第 5 章

pandas 入门

pandas 是贯穿本书后续内容的主要工具。它包含多种数据结构和数据操作工具，可以使数据清洗和分析工作更快、更简单。pandas 经常和其他工具一同使用，如数值计算工具 NumPy 和 SciPy，分析库 statsmodels 和 scikit-learn，以及数据可视化库 matplotlib。pandas 从 NumPy 汲取了大量基于数组计算的语言风格，特别是基于数组的函数，以及不使用 for 循环进行数据处理。

虽然 pandas 采用了大量的 NumPy 编码风格，但二者最大的不同是，pandas 是专门为处理表格和异构数据设计的，而 NumPy 更适合处理同构类型的数值数组数据。

自从 2010 年成为开源项目，pandas 逐渐成长为一个非常庞大的库，并应用于许多领域的真实案例。开发者社区已经有 2500 位独立的贡献者，他们在解决日常数据问题的同时为这个项目提供贡献。活跃的 pandas 开发者和用户社区是 pandas 取得成功的重要一环。

许多人可能不清楚，自从 2013 年，我就远离了日复一日的 pandas 开发工作。pandas 自那以后就一直是完全由社区管理的项目。谨代表所有人对核心开发者和所有贡献者的辛勤工作表示由衷的感谢！

在本书后续章节中，我将使用下面的约定导入 NumPy 和 pandas：

```
In [1]: import numpy as np

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

因此，只要你在代码中看到 `pd.`，就知道引用的是 `pandas`。因为 Series 和 DataFrame 用的次数非常多，所以将其导入本地命名空间中会更加方便：

```
In [3]: from pandas import Series, DataFrame
```

5.1 pandas 的数据结构介绍

要使用 pandas，需要熟悉它的两个主要数据结构：Series 和 DataFrame。它们虽然并不能解决所有问题，但为大部分数据工作提供了一个可靠、易于使用的基础。

5.1.1 Series

Series 是一种类似于一维数组的对象，它由一组（类似 NumPy 数据类型的）数据以及一组与之相关的数据标签（即索引）组成。仅由一个数组即可创建最简单的 Series：

```
In [14]: obj = pd.Series([4, 7, -5, 3])

In [15]: obj
Out[15]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series 字符串以交互式的方式呈现，索引位于左边，值位于右边。由于我们没有为数据指定索引，因此会自动创建一个 0 到 `N - 1`（*N* 为数据的长度）的整数型索引。可以通过 Series 的 `array` 和 `index` 属性获取其数组值和索引对象：

```
In [16]: obj.array
Out[16]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64

In [17]: obj.index
Out[17]: RangeIndex(start=0, stop=4, step=1)
```

`.array` 属性的结果是 PandasArray 对象，它通常封装了一个 NumPy 数组，但也可以包含特殊的扩展数组类型，7.3 节会进一步详解。

经常需要创建带有索引的 Series，用标签指明各个数据点：

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])

In [19]: obj2
Out[19]:
d    4
b    7
a   -5
c    3
dtype: int64
```

```
In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

与 NumPy 数组相比，你可以通过索引的标签选取 Series 中的单个或一组值：

```
In [21]: obj2["a"]
Out[21]: -5

In [22]: obj2["d"] = 6

In [23]: obj2[["c", "a", "d"]]
Out[23]:
c    3
a   -5
d    6
dtype: int64
```

`["c", "a", "d"]` 是索引列表，它包含的是字符串而不是数字。

使用 NumPy 函数或类似 NumPy 的运算，比如用布尔型数组进行过滤、标量乘法、应用数学函数等，都会保留索引值的链接：

```
In [24]: obj2[obj2 > 0]
Out[24]:
d    6
b    7
c    3
dtype: int64

In [25]: obj2 * 2
Out[25]:
d    12
b    14
a   -10
c     6
dtype: int64

In [26]: import numpy as np

In [27]: np.exp(obj2)
Out[27]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

还可以将 Series 看作长度固定的有序字典，因为它是索引值对数据值的映射。在可能使用字典的场景中，也可以使用 Series：

```
In [28]: "b" in obj2
Out[28]: True

In [29]: "e" in obj2
Out[29]: False
```

如果数据已经存放在一个 Python 字典中，也可以通过传入这个字典来创建 Series：

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [31]: obj3 = pd.Series(sdata)

In [32]: obj3
Out[32]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

通过 to_dict 方法，Series 也可以转换回字典：

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

如果只传入一个字典，则结果 Series 中的索引会遵循字典的键的顺序，而键的顺序依据的是字典的 keys 方法，keys 方法又取决于键的插入顺序。你可以将字典键按照想要的顺序传给构造函数，从而使生成的 Series 的索引顺序符合你的预期：

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]

In [35]: obj4 = pd.Series(sdata, index=states)

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

在这个例子中，sdata 中跟 states 索引相匹配的三个值放到了相应的位置上，但由于 "California" 没有对应的值，所以其结果为 NaN（即 not a number，非数字），在 pandas 中，它用于标记缺失值或 NA 值。因为 "Utah" 不在 states 中，所以它会从结果中除去。

我将交替使用“缺失”“NA”或“空值”表示缺失数据。pandas 的 isnull 和 notnull 函数可用于检测缺失数据：

```
In [37]: pd.isna(obj4)
Out[37]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [38]: pd.notna(obj4)
Out[38]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series 也将其作为实例方法：

```
In [39]: obj4.isna()
Out[39]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

我将在第 7 章详细讲解如何处理缺失数据。

对于许多应用而言，Series 最实用的一个功能是它在算数运算中能自动对齐索引标签：

```
In [40]: obj3
Out[40]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64

In [41]: obj4
Out[41]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [42]: obj3 + obj4
Out[42]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

数据对齐功能将在后面详细讲解。如果你使用过数据库，可以认为这类似于 join 操作。

Series 对象本身及其索引都有 name 属性，该属性与 pandas 其他功能的关系非常密切：

```
In [43]: obj4.name = "population"

In [44]: obj4.index.name = "state"

In [45]: obj4
Out[45]:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Series 的索引可以通过赋值的方式就地修改：

```
In [46]: obj
Out[46]:
0    4
1    7
2   -5
3    3
dtype: int64

In [47]: obj.index = ["Bob", "Steve", "Jeff", "Ryan"]

In [48]: obj
Out[48]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

5.1.2 DataFrame

DataFrame 是矩形的数据表，它含有一组有序且有命名的列，每一列可以是不同的数据类型（数值、字符串、布尔值等）。DataFrame 既有行索引也有列索引，可以看作由共用同一个索引的 Series 组成的字典。

虽然 DataFrame 是二维的，但利用层次化索引，你仍然可以用其表示更高维度的表格型数据。层次化索引是高级数据处理特性，我们会在第 8 章进行讨论。

有多种创建 DataFrame 的方式，最常用的是传入一个由等长列表或 NumPy 数组构成的

字典：

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

生成的 DataFrame 会自动加上索引，跟 Series 一样，且全部列会按照 `data` 键（键的顺序取决于在字典中插入的顺序）的顺序有序排列：

```
In [50]: frame
Out[50]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

如果你使用的是 Jupyter notebook，pandas 的 DataFrame 对象将会展示为对浏览器更为友好的 HTML 表格。如图 5-1 的示例所示。

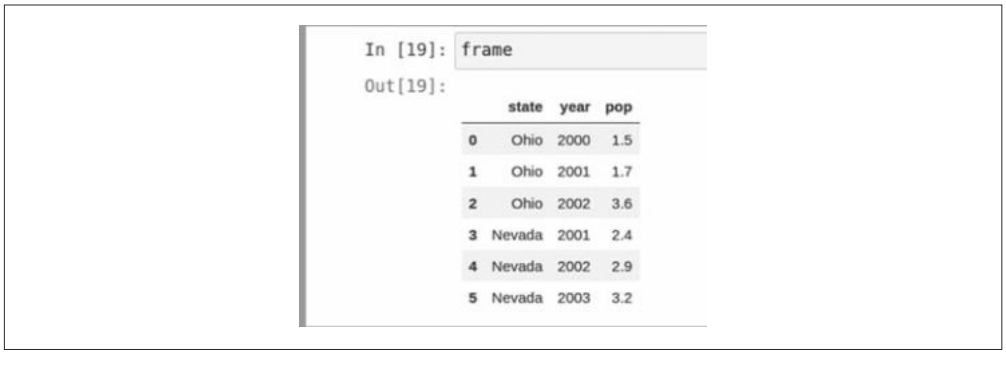

图 5-1：pandas 的 DataFrame 在 Jupyter 中的呈现形式

对于特别大的 DataFrame，`head` 方法会只选取前 5 行：

```
In [51]: frame.head()
Out[51]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

相似地，tail 方法会返回最后 5 行：

```
In [52]: frame.tail()
Out[52]:
    state  year  pop
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

如果指定了列的序列，则 DataFrame 的列就会按照指定顺序进行排列：

```
In [53]: pd.DataFrame(data, columns=["year", "state", "pop"])
Out[53]:
   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
5  2003  Nevada  3.2
```

如果字典不包含传入的列，就会在结果中产生缺失值：

```
In [54]: frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])

In [55]: frame2
Out[55]:
   year   state  pop debt
0  2000    Ohio  1.5  NaN
1  2001    Ohio  1.7  NaN
2  2002    Ohio  3.6  NaN
3  2001  Nevada  2.4  NaN
4  2002  Nevada  2.9  NaN
5  2003  Nevada  3.2  NaN

In [56]: frame2.columns
Out[56]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

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

```
In [57]: frame2["state"]
Out[57]:
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object

In [58]: frame2.year
```

```
Out[58]:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

IPython 提供了类似属性的访问（即 `frame2.year`）和 tab 补全，使用更为方便。`frame2[column]` 适用于任意列的名，但是 `frame2.column` 只有在列名是合理的 Python 变量名时才适用，并且不能与任意的 DataFrame 的方法名冲突。例如，如果列名包含空格或下划线以外的符号，就不能用点属性的方式访问。

注意，返回的 Series 拥有与原 DataFrame 相同的索引，且其 `name` 属性也已经合理地设置好了。

通过 `iloc` 和 `loc` 属性，也可以通过位置或名称的方式进行获取行（稍后将在节对此进行详细讲解）：

```
In [59]: frame2.loc[1]
Out[59]:
year     2001
state    Ohio
pop       1.7
debt      NaN
Name: 1, dtype: object

In [60]: frame2.iloc[2]
Out[60]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: 2, dtype: object
```

通过赋值的方式可以修改列。例如，我们可以将空的列 `debt` 赋值为标量值或值数组：

```
In [61]: frame2["debt"] = 16.5

In [62]: frame2
Out[62]:
   year   state  pop  debt
0  2000    Ohio  1.5  16.5
1  2001    Ohio  1.7  16.5
2  2002    Ohio  3.6  16.5
3  2001  Nevada  2.4  16.5
4  2002  Nevada  2.9  16.5
5  2003  Nevada  3.2  16.5
```

```
In [63]: frame2["debt"] = np.arange(6.)

In [64]: frame2
Out[64]:
   year   state  pop  debt
0  2000    Ohio  1.5   0.0
1  2001    Ohio  1.7   1.0
2  2002    Ohio  3.6   2.0
3  2001  Nevada  2.4   3.0
4  2002  Nevada  2.9   4.0
5  2003  Nevada  3.2   5.0
```

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，它的标签就会精确匹配 DataFrame 的索引，所有的空缺都将填上缺失值：

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

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

In [67]: frame2
Out[67]:
   year   state  pop  debt
0  2000    Ohio  1.5   NaN
1  2001    Ohio  1.7   NaN
2  2002    Ohio  3.6   NaN
3  2001  Nevada  2.4   NaN
4  2002  Nevada  2.9   NaN
5  2003  Nevada  3.2   NaN
```

为不存在的列赋值会创建一个新的列。

关键字 `del` 可以像在字典中那样删除列。作为例子，我先添加一个新的布尔值的列，条件为 `state` 列是否为 `"Ohio"`：

```
In [68]: frame2["eastern"] = frame2["state"] == "Ohio"

In [69]: frame2
Out[69]:
   year   state  pop  debt  eastern
0  2000    Ohio  1.5   NaN     True
1  2001    Ohio  1.7   NaN     True
2  2002    Ohio  3.6   NaN     True
3  2001  Nevada  2.4   NaN    False
4  2002  Nevada  2.9   NaN    False
5  2003  Nevada  3.2   NaN    False
```

不能用 `frame2.eastern` 的方式创建新的列。

可以用 del 方法来删除这一列：

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

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

通过索引方式从 DataFrame 返回的列只是底层数据的视图而已，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 DataFrame 上。应当通过 Series 的 copy 方法来复制列。

另一种常见的数据形式是嵌套字典的字典：

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

如果将嵌套字典传给 DataFrame，pandas 就会将外层字典的键解释为列，将内层字典的键解释为行索引：

```
In [73]: frame3 = pd.DataFrame(populations)

In [74]: frame3
Out[74]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
2002   3.6     2.9
```

使用类似 NumPy 数组的方法，可以对 DataFrame 进行转置（交换行和列）：

```
In [75]: frame3.T
Out[75]:
        2000  2001  2002
Ohio     1.5   1.7   3.6
Nevada   NaN   2.4   2.9
```

注意，如果列没有所有相同的数据类型，转置则会丢弃列的数据类型，因此转置以及再次转置返回原先的矩阵会导致丢失先前的类型信息。在这个例子中，列变成了纯 Python 对象的数组。

内层字典的键被合并后形成了结果的索引。如果明确指定了索引，则不会这样：

```
In [76]: pd.DataFrame(populations, index=[2001, 2002, 2003])
Out[76]:
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
2003   NaN     NaN
```

由 Series 组成的字典差不多也是一样的用法：

```
In [77]: pdata = {"Ohio": frame3["Ohio"][:-1],
   ....:          "Nevada": frame3["Nevada"][:2]}

In [78]: pd.DataFrame(pdata)
Out[78]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
```

表 5-1 列出了 DataFrame 构造函数所能接收的各种数据。

表 5-1：可以向 DataFrame 构造器输入的数据

类型	说明
二维 ndarray	数据矩阵，传入可选的行标签和列标签
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 / 记录化数组	处理方式与“由数组组成的字典”一致
由 Series 组成的字典	每个值成为一列。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	各内部字典成为一列。键会被合并成结果的行索引，处理方式与“由 Series 组成的字典”一致
字典或 Series 的列表	各项会成为 DataFrame 的一行。字典键或 Series 索引合并后成为 DataFrame 的列标签
由列表或元组组成的列表	处理方式与“二维 ndarray”一致
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	处理方式与“二维 ndarray”一致，只是在 DataFrame 结果中缺少掩码值

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

```
In [79]: frame3.index.name = "year"

In [80]: frame3.columns.name = "state"

In [81]: frame3
Out[81]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

不同于 Series，DataFrame 本身没有 name 属性。DataFrame 的 `to_numpy` 方法将数据以

二维 ndarray 的 DataFrame 形式返回：

```
In [82]: frame3.to_numpy()
Out[82]:
array([[1.5, nan],
       [1.7, 2.4],
       [3.6, 2.9]])
```

如果 DataFrame 各列的数据类型不同，则返回数组会选用能兼容所有列的数据类型：

```
In [83]: frame2.to_numpy()
Out[83]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, nan],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, nan],
       [2002, 'Nevada', 2.9, nan],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

5.1.3 索引对象

pandas 的索引对象负责存储轴标签（包括 DataFrame 的列名）和其他元数据（比如轴名称或标签）。构建 Series 或 DataFrame 时，所用到的任何数组或其他标签序列都会转换成索引对象：

```
In [84]: obj = pd.Series(np.arange(3), index=["a", "b", "c"])

In [85]: index = obj.index

In [86]: index
Out[86]: Index(['a', 'b', 'c'], dtype='object')

In [87]: index[1:]
Out[87]: Index(['b', 'c'], dtype='object')
```

Index 对象是不可变的，因此用户不能对其进行修改：

```
index[1] = "d"  # TypeError
```

不可变性可以使索引对象在多个数据结构之间安全共享：

```
In [88]: labels = pd.Index(np.arange(3))

In [89]: labels
Out[89]: Int64Index([0, 1, 2], dtype='int64')

In [90]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [91]: obj2
Out[91]:
```

```
0    1.5
1   -2.5
2    0.0
dtype: float64

In [92]: obj2.index is labels
Out[92]: True
```

虽然用户不需要经常使用索引的功能，但是因为一些操作生成的结果会包含索引化的数据，所以理解它们的工作原理也很重要。

除了类似于数组，索引也类似于一个大小固定的集合：

```
In [93]: frame3
Out[93]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9

In [94]: frame3.columns
Out[94]: Index(['Ohio', 'Nevada'], dtype='object', name='state')

In [95]: "Ohio" in frame3.columns
Out[95]: True

In [96]: 2003 in frame3.index
Out[96]: False
```

与 Python 集合不同，pandas 的索引可以包含重复的标签：

```
In [97]: pd.Index(["foo", "foo", "bar", "bar"])
Out[97]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

选择重复的标签，会选取所有对应的结果。

每个索引都有一些集合逻辑的方法和属性，可用于处理索引所包含数据的常见问题。表 5-2 总结了其中一些方法和属性。

表 5-2：索引的方法和属性

方法	说明
`append()`	连接另一个索引对象，产生一个新的索引
`difference()`	计算差集，并得到一个索引
`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

表 5-2：索引的方法和属性（续）

方法	说明
delete()	删除索引 i 处的元素，并得到新的索引
drop()	删除传入的值，并得到新的索引
insert()	将元素插入索引 i 处，并得到新的索引
is_monotonic	当各元素均大于等于前一个元素时，返回 True
is_unique	当索引没有重复值时，返回 True
unique()	计算索引中唯一值的数组

5.2 基本功能

本节，我将介绍 Series 和 DataFrame 中数据的基本操作方法。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，而是关注最重要的功能，对于不常用的内容（比如那些更深奥的内容）就留给读者自行探索。

5.2.1 重建索引

pandas 对象的一个重要方法是 reindex，其作用是创建一个数据根据新索引重新排列的新对象。看下面的例子：

```
In [98]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])

In [99]: obj
Out[99]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

对该 Series 调用 reindex 将会根据新索引进行重排。如果某个索引值当前不存在，就导入缺失值：

```
In [100]: obj2 = obj.reindex(["a", "b", "c", "d", "e"])

In [101]: obj2
Out[101]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

对于时间序列这样的有序数据，重建索引时可能需要做一些插值或填值处理。method 选

项可以达到此目的，例如，使用 ffill 可以实现前向填充值：

```
In [102]: obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])

In [103]: obj3
Out[103]:
0      blue
2    purple
4    yellow
dtype: object

In [104]: obj3.reindex(np.arange(6), method="ffill")
Out[104]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

借助 DataFrame，reindex 可以修改（行）索引、列，也可以同时修改。只传入一个序列时，会重建索引结果中的行：

```
In [105]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   .....:                      index=["a", "c", "d"],
   .....:                      columns=["Ohio", "Texas", "California"])

In [106]: frame
Out[106]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [107]: frame2 = frame.reindex(index=["a", "b", "c", "d"])

In [108]: frame2
Out[108]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

列可以用 columns 关键字重建索引：

```
In [109]: states = ["Texas", "Utah", "California"]

In [110]: frame.reindex(columns=states)
Out[110]:
   Texas  Utah  California
a      1   NaN           2
```

```
c      4   NaN           5
d      7   NaN           8
```

因为 "Ohio" 不在 stats 中，所以这一列从结果中删除。

另一种重建索引的方式是传入新的轴标签作为位置参数，然后用 axis 关键字对指定轴进行重建索引：

```
In [111]: frame.reindex(states, axis="columns")
Out[111]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

表 5-3 列出了 reindex 函数的参数及说明。

表 5-3：reindex 函数的参数及说明

参数	说明
labels	用作索引的新序列。既可以是索引实例，也可以是其他序列型的 Python 数据结构。索引会被直接使用，无须复制
index	使用传入的序列作为新的索引标签
columns	使用传入的序列作为新的列标签
axis	进行重建索引的轴，可以是索引（行）也可以是列。默认为索引。既可以 reindex(index=new_labels)，也可以 reindex(columns=new_labels)。
method	插值（填充）方式，"ffill" 表示前向填充，"bfill" 表示后向填充
fill-value	在重建索引的过程中导入了缺失值，用作替换的值。使用 fill_value="missing" 可以对结果中不存在的标签填入缺失值。
limit	前向或后向填充值时，最大的填充区间（以元素计数）
tolerance	前向或后向填充值时，填充不准确匹配项的最大区间（绝对值距离）
level	在多层索引的指定层级上匹配索引，否则选取其子集
copy	如果为 True，即新索引等于旧索引，总是复制底层数据；如果是 False，则在索引相同时不复制数据

还可以用 loc 运算符重建索引，这也是多数人更为常用的方式，详见节。只有当新索引的标签在 DataFrame 中已经存在时，才能这么做（否则的话，reindex 将会给新标签插入缺失值）：

```
In [112]: frame.loc[["a", "d", "c"], ["California", "Texas"]]
Out[112]:
   California  Texas
a           2      1
d           8      7
c           5      4
```

5.2.2 删除指定轴上的项

删除某条轴上的一个或多个项很简单，只要有一个索引数组或不包含这些项的列表，就可以使用 reindex 方法或基于 .loc 的索引进行删除。由于需要执行一些数据整理和集合逻辑操作，因此 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 [113]: obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])

In [114]: obj
Out[114]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [115]: new_obj = obj.drop("c")

In [116]: new_obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [117]: obj.drop(["d", "c"])
Out[117]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

对于 DataFrame，可以删除任意轴上的索引值。为了演示，首先创建一个 DataFrame 实例：

```
In [118]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=["Ohio", "Colorado", "Utah", "New York"],
   .....:                     columns=["one", "two", "three", "four"])

In [119]: data
Out[119]:
          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
```

用标签序列调用 drop 会从行标签（axis 0）删除值：

```
In [120]: data.drop(index=["Colorado", "Ohio"])
Out[120]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

通过传入 `columns` 关键字，可以删除列的标签：

```
In [121]: data.drop(columns=["two"])
Out[121]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
```

还可以传入 `axis=1`（类似于 NumPy）或 `axis="columns"`，从列删除值：

```
In [122]: data.drop("two", axis=1)
Out[122]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [123]: data.drop(["two", "four"], axis="columns")
Out[123]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

5.2.3 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 的索引值可以不仅仅是整数。下面是几个例子：

```
In [124]: obj = pd.Series(np.arange(4.), index=["a", "b", "c", "d"])

In [125]: obj
Out[125]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [126]: obj["b"]
Out[126]: 1.0

In [127]: obj[1]
```

```
Out[127]: 1.0

In [128]: obj[2:4]
Out[128]:
c    2.0
d    3.0
dtype: float64

In [129]: obj[["b", "a", "d"]]
Out[129]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [130]: obj[[1, 3]]
Out[130]:
b    1.0
d    3.0
dtype: float64

In [131]: obj[obj < 2]
Out[131]:
a    0.0
b    1.0
dtype: float64
```

虽然可以用这种方式用标签选取数据，但是更可取的方式是使用特殊的 loc 运算符选取索引值：

```
In [132]: obj.loc[["b", "a", "d"]]
Out[132]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

loc 的方式更为可取，因为用 [] 进行索引时，针对整数的处理有所不同。如果索引包含整数，常规的基于 [] 的索引会将整数用作标签，因此索引的选取取决于数据类型。例如：

```
In [133]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])

In [134]: obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [135]: obj1
Out[135]:
2    1
0    2
1    3
dtype: int64

In [136]: obj2
```

```
Out[136]:
a    1
b    2
c    3
dtype: int64

In [137]: obj1[[0, 1, 2]]
Out[137]:
0    2
1    3
2    1
dtype: int64

In [138]: obj2[[0, 1, 2]]
Out[138]:
a    1
b    2
c    3
dtype: int64
```

使用 loc 时，如果索引中不含整数，则表达式 obj.loc[[0, 1, 2]] 会失效：

```
In [134]: obj2.loc[[0, 1]]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/ipykernel_804589/4185657903.py in <module>
----> 1 obj2.loc[[0, 1]]

^ LONG EXCEPTION ABBREVIATED ^

KeyError: "None of [Int64Index([0, 1], dtype="int64")] are in the [index]"
```

loc 运算符只使用标签，iloc 运算符只使用整数。无论索引是否包含整数，都能使用 iloc：

```
In [139]: obj1.iloc[[0, 1, 2]]
Out[139]:
2    1
0    2
1    3
dtype: int64

In [140]: obj2.iloc[[0, 1, 2]]
Out[140]:
a    1
b    2
c    3
dtype: int64
```

还可以使用标签进行切片，但区别于普通 Python 的切片方式，loc 的切片是包含末端的：

```
In [141]: obj2.loc["b":"c"]
Out[141]:
b    2
c    3
dtype: int64
```

使用以上切片方法可以对 Series 的相应部分进行赋值：

```
In [142]: obj2.loc["b":"c"] = 5

In [143]: obj2
Out[143]:
a    1
b    5
c    5
dtype: int64
```

像函数那样调用 loc 和 iloc 是新手常犯的错误，正确的方式是用方括号进行索引。方括号不仅用于切片索引，还用于对 DataFrame 的多个轴进行索引。

用单个值或序列对 DataFrame 进行索引，以获取单列或多列：

```
In [144]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=["Ohio", "Colorado", "Utah", "New York"],
   .....:                     columns=["one", "two", "three", "four"])

In [145]: data
Out[145]:
          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 [146]: data["two"]
Out[146]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [147]: data[["three", "one"]]
Out[147]:
          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用法。首先，可以通过切片或布尔型数组选取数据：

```
In [148]: data[:2]
Out[148]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [149]: data[data["three"] > 5]
Out[149]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

选取行的语法 `data[:2]` 十分方便。向 `[]` 传入单个元素或列表，就可以选择列。

另一种用法是通过布尔型 DataFrame 进行索引，比如由标量比较运算得出的 DataFrame。来看下面的 DataFrame，它的所有值都是与一个标量比较得出的布尔值：

```
In [150]: data < 5
Out[150]:
             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
```

我们可以用这个 DataFrame 将 0 赋值给等于 `True` 的位置：

```
In [151]: data[data < 5] = 0

In [152]: data
Out[152]:
          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
```

用 loc 和 iloc 选取 DataFrame

与 Series 一样，DataFrame 有两个特殊属性 `loc` 和 `iloc`，分别用于标签索引和整数索引。由于 DataFrame 是二维的，因此可以用 NumPy 风格的语法，使用轴标签（`loc`）或整数（`iloc`）从 DataFrame 选取行和列的子集。

作为一个初步示例，让我们通过标签选取一行：

```
In [153]: data
Out[153]:
```

```
          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

In [154]: data.loc["Colorado"]
Out[154]:
one      0
two      5
three    6
four     7
Name: Colorado, dtype: int64
```

取出的单独一行是 Series，它的索引是 DataFrame 的列标签。要选取多行，创建一个新的 DataFrame，可以传入标签序列：

```
In [155]: data.loc[["Colorado", "New York"]]
Out[155]:
          one  two  three  four
Colorado    0    5      6     7
New York   12   13     14    15
```

如果要用 loc 同时选取行和列，可以用逗号将选取过程分隔开，如下所示：

```
In [156]: data.loc["Colorado", ["two", "three"]]
Out[156]:
two      5
three    6
Name: Colorado, dtype: int64
```

我们再用 iloc 和整数索引做一些相似的选取操作：

```
In [157]: data.iloc[2]
Out[157]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64

In [158]: data.iloc[[2, 1]]
Out[158]:
          one  two  three  four
Utah        8    9     10    11
Colorado    0    5      6     7

In [159]: data.iloc[2, [3, 0, 1]]
Out[159]:
four    11
one      8
two      9
Name: Utah, dtype: int64
```

```
In [160]: data.iloc[[1, 2], [3, 0, 1]]
Out[160]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

除了单个标签或多个标签，这两个索引函数也可以使用切片：

```
In [161]: data.loc[:"Utah", "two"]
Out[161]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [162]: data.iloc[:, :3][data.three > 5]
Out[162]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

`loc` 可以使用布尔型数组，但 `iloc` 不能使用：

```
In [163]: data.loc[data.three >= 2]
Out[163]:
          one  two  three  four
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

有多种方式可以选取和重排存储在 pandas 中的数据。对于 DataFrame 的数据选取方法，表 5-4 做了简短总结。后面会看到，还有更多的方法用于处理层次化索引。

表 5-4：DataFrame 的索引选项

类型	说明
`df[column]`	从 DataFrame 选取单列或多列；在特殊情况下更为便利：布尔型数组（过滤行）、切片（行切片）、布尔型 DataFrame（根据条件设置值）
`df.loc[rows]`	通过标签，选取 DataFrame 的单行或多行
`df.loc[:, cols]`	通过标签，选取单列或多列
`df.loc[rows, cols]`	通过标签，同时选取行和列
`df.iloc[rows]`	通过整数索引，从 DataFrame 选取单行或多行
`df.iloc[:, cols]`	通过整数索引，选取单列或多列
`df.iloc[rows, cols]`	通过整数索引，同时选取行和列
`df.at[row, col]`	通过行和列标签，选取单个标量值
`df.iat[row, col]`	通过行和列的索引（整数），选取单个标量值
`reindex` 方法	通过标签选取行或列

整数索引中的陷阱

处理整数索引的 pandas 对象常常会难住新手，因为它与 Python 内置的数据结构不同，比如列表和元组。例如，你可能不认为下面的代码会出错：

```
In [164]: ser = pd.Series(np.arange(3.))

In [165]: ser
Out[165]:
0    0.0
1    1.0
2    2.0
dtype: float64

In [166]: ser[-1]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/indexes/range.p
y in get_loc(self, key, method, tolerance)
    384                 try:
--> 385                     return self._range.index(new_key)
    386                 except ValueError as err:
ValueError: -1 is not in range
The above exception was the direct cause of the following exception:
KeyError                                  Traceback (most recent call last)
<ipython-input-166-44969a759c20> in <module>
----> 1 ser[-1]
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/series.py in __
getitem__(self, key)
    956
    957         elif key_is_scalar:
--> 958             return self._get_value(key)
    959
    960         if is_hashable(key):
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/series.py in _g
et_value(self, label, takeable)
   1067
   1068         # Similar to Index.get_value, but we do not fall back to position
al
-> 1069         loc = self.index.get_loc(label)
   1070         return self.index._get_values_for_loc(self, loc, label)
   1071
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/indexes/range.p
y in get_loc(self, key, method, tolerance)
    385                     return self._range.index(new_key)
    386                 except ValueError as err:
--> 387                     raise KeyError(key) from err
    388             self._check_indexing_error(key)
    389             raise KeyError(key)
KeyError: -1
```

在这个例子中，pandas 会“回退”到整数索引，但是这样的方式难免会向代码中导入一些细微问题。索引包含了 0、1、2，但让 pandas 来猜用户的意向（是标签索引还是位置

索引）是很难的：

```
In [167]: ser
Out[167]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

另外，对于非整数索引，则不会产生歧义：

```
In [168]: ser2 = pd.Series(np.arange(3.), index=["a", "b", "c"])

In [169]: ser2[-1]
Out[169]: 2.0
```

如果轴索引含有整数，则数据选取将始终基于标签。就像前文说的，使用 loc（标签）或 iloc（整数）能让选取数据更准确：

```
In [170]: ser.iloc[-1]
Out[170]: 2.0
```

而整数切片总是基于整数的：

```
In [171]: ser[:2]
Out[171]:
0    0.0
1    1.0
dtype: float64
```

为了避免错误，最好使用 loc 和 iloc 来选取数据。

链式索引中的陷阱

在上一节中，我们学习了如何使用 loc 和 iloc 灵活地选取 DataFramce 中的数据。这两个索引属性也可以用于就地修改 DataFrame，但需要谨慎操作。

例如，以上面的 DataFrame 为例，我们可以通过标签或整数索引对列或行赋值：

```
In [172]: data.loc[:, "one"] = 1

In [173]: data
Out[173]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    1    5      6     7
Utah        1    9     10    11
New York    1   13     14    15

In [174]: data.iloc[2] = 5
```

```
In [175]: data
Out[175]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    1    5      6     7
Utah        5    5      5     5
New York    1   13     14    15

In [176]: data.loc[data["four"] > 5] = 3

In [177]: data
Out[177]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      5     5
New York    3    3      3     3
```

pandas 新用户的常见问题是在赋值时进行链式选取，如下所示：

```
In [177]: data.loc[data.three == 5]["three"] = 6
<ipython-input-11-0ed1cf2155d5>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

根据数据内容，链式选取可能抛出 `SettingWithCopyWarning`，这是警告用户正在修改临时值（即 `data.loc[data.three == 5]` 的非空结果），而非修改原始 DataFrame 的 `data`，后者才是本来想修改的。在这个例子中，`data` 没有被修改：

```
In [179]: data
Out[179]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      5     5
New York    3    3      3     3
```

对于此类场景，正确的做法是重写链式赋值以使用单独的 `loc` 操作：

```
In [180]: data.loc[data.three == 5, "three"] = 6

In [181]: data
Out[181]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      6     5
New York    3    3      3     3
```

因此，首要的原则是在赋值时避免链式索引。此外还存在其他场景，pandas 会生成与链

式索引相关的 SettingsWithCopyWarning。关于这个主题，建议读者参考 pandas 的在线文档。

5.2.4 算术运算和数据对齐

用 pandas 来处理不同索引的对象要简单得多。例如，当将对象相加时，如果存在不同的索引对，则结果中的索引是所有索引的并集。来看一个例子：

```
In [182]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=["a", "c", "d", "e"])

In [183]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=["a", "c", "e", "f", "g"])

In [184]: s1
Out[184]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [185]: s2
Out[185]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

将它们相加就会得到：

```
In [186]: s1 + s2
Out[186]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

对于不重叠的标签，内部数据对齐会导入缺失值。缺失值会在以后的算术运算过程中传播。

对于 DataFrame，对齐操作会同时发生在行和列上：

```
In [187]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),
   .....:                    index=["Ohio", "Texas", "Colorado"])

In [188]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
   .....:                    index=["Utah", "Ohio", "Texas", "Oregon"])
```

```
In [189]: df1
Out[189]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [190]: df2
Out[190]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

把这两个相加后会返回 DataFrame，其索引和列为原来两个 DataFrame 的并集：

```
In [191]: df1 + df2
Out[191]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

因为 "c" 和 "e" 列均不在两个 DataFrame 对象中，在结果中以缺失值呈现。行也是同样，如果行标签在两个 DataFrame 对象中都没有，也会导致缺失值。

如果将没有共用的列或行标签的 DataFrame 对象相加，结果将全部为空：

```
In [192]: df1 = pd.DataFrame({"A": [1, 2]})

In [193]: df2 = pd.DataFrame({"B": [3, 4]})

In [194]: df1
Out[194]:
   A
0  1
1  2

In [195]: df2
Out[195]:
   B
0  3
1  4

In [196]: df1 + df2
Out[196]:
    A   B
0 NaN NaN
1 NaN NaN
```

带有填充值的算术方法

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的某个轴标签不在另一个对象中时，填充一个特殊值（比如 0）。下面的例子使用了 np.nan 赋值给 NA 值：

```
In [197]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list("abcd"))

In [198]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list("abcde"))

In [199]: df2.loc[1, "b"] = np.nan

In [200]: df1
Out[200]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [201]: df2
Out[201]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

将它们相加时，没有重叠的位置就会产生 NA 值：

```
In [202]: df1 + df2
Out[202]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 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，用传入值替换运算中的缺失值：

```
In [203]: df1.add(df2, fill_value=0)
Out[203]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

表 5-5 列出了 Series 和 DataFrame 的算术方法。每个方法都有一个以字母 r 开头的副本，会将参数翻转。因此下面两个语句是等价的：

```
In [204]: 1 / df1
Out[204]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909

In [205]: df1.rdiv(1)
Out[205]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909
```

表 5-5：灵活的算术方法

方法	说明
add、radd	加法（+）
sub、rsub	减法（-）
div、rdiv	除法（/）
floordiv、rfloordiv	底除（//）
mul、rmul	乘法（*）
pow、rpow	乘方（**）

相关地，对 Series 和 DataFrame 重建索引时，也可以指定不同的填充值：

```
In [206]: df1.reindex(columns=df2.columns, fill_value=0)
Out[206]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

DataFrame 和 Series 间的运算

与不同维度的 NumPy 数组一样，DataFrame 和 Series 之间的算术运算也要遵守一定的规则。先来看一个具有启发性的例子，计算一个二维数组与某一行的差：

```
In [207]: arr = np.arange(12.).reshape((3, 4))

In [208]: arr
Out[208]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])

In [209]: arr[0]
Out[209]: array([0., 1., 2., 3.])
```

```
In [210]: arr - arr[0]
Out[210]:
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

当我们从 arr 减去 arr[0] 时，每一行都会执行这个操作。这就是广播机制，因为广播与 NumPy 数组关系密切，附录 A 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 [211]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list("bde"),
   .....:                      index=["Utah", "Ohio", "Texas", "Oregon"])

In [212]: series = frame.iloc[0]

In [213]: frame
Out[213]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [214]: series
Out[214]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

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

```
In [215]: frame - series
Out[215]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

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重建索引以形成并集：

```
In [216]: series2 = pd.Series(np.arange(3), index=["b", "e", "f"])

In [217]: series2
Out[217]:
b    0
e    1
```

```
f    2
dtype: int64

In [218]: frame + series2
Out[218]:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

如果你希望在列上广播且匹配行，则必须使用算术运算方法并指定匹配索引。例如：

```
In [219]: series3 = frame["d"]

In [220]: frame
Out[220]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [221]: series3
Out[221]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [222]: frame.sub(series3, axis="index")
Out[222]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

传入的轴就是用于匹配的 `axis`。在本例中，我们想匹配 DataFrame 的行索引（`axis="index"`）并进行列广播。

5.2.5 函数应用和映射

NumPy 的通用函数（元素级数组方法）也可用于操作 pandas 对象：

```
In [223]: frame = pd.DataFrame(np.random.standard_normal((4, 3)),
   .....:                      columns=list("bde"),
   .....:                      index=["Utah", "Ohio", "Texas", "Oregon"])

In [224]: frame
Out[224]:
```

```
               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 [225]: np.abs(frame)
Out[225]:
               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各行所形成的一维数组上。DataFrame 的 `apply` 方法可以实现此功能：

```
In [226]: def f1(x):
   .....:     return x.max() - x.min()

In [227]: frame.apply(f1)
Out[227]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

这里的函数 `f` 计算了 Series 最大值和最小值的差，在 `frame` 的每列都执行了一次。结果是一个 Series，使用 `frame` 的列作为索引。

如果传递 `axis="columns"` 给 `apply` 函数，这个函数会在每行执行一次，可以将其当作“跨列处理”：

```
In [228]: frame.apply(f1, axis="columns")
Out[228]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

许多最为常见的数组统计功能（如 `sum` 和 `mean`）都是 DataFrame 的方法，因此无须使用 `apply` 方法。

传递到 `apply` 的函数不一定返回单个标量值，还可以返回由多个值组成的 Series：

```
In [229]: def f2(x):
   .....:     return pd.Series([x.min(), x.max()], index=["min", "max"])

In [230]: frame.apply(f2)
Out[230]:
```

```
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

还可以使用元素级的 Python 函数。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 [231]: def my_format(x):
   .....:     return f"{x:.2f}"

In [232]: frame.applymap(my_format)
Out[232]:
            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
```

之所以叫作 applymap，是因为 Series 有一个用于元素级函数的 map 方法：

```
In [233]: frame["e"].map(my_format)
Out[233]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

5.2.6 排序和排名

根据特定条件对数据集排序也是一种重要的内置运算。要对行或列标签按字典顺序排序，可使用 sort_index 方法，它将返回一个排好序的新对象：

```
In [234]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])

In [235]: obj
Out[235]:
d    0
a    1
b    2
c    3
dtype: int64

In [236]: obj.sort_index()
Out[236]:
a    1
b    2
c    3
d    0
dtype: int64
```

对于 DataFrame，可以根据任意一个轴上的索引进行排序：

```
In [237]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=["three", "one"],
   .....:                      columns=["d", "a", "b", "c"])

In [238]: frame
Out[238]:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

In [239]: frame.sort_index()
Out[239]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [240]: frame.sort_index(axis="columns")
Out[240]:
       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 [241]: frame.sort_index(axis="columns", ascending=False)
Out[241]:
       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 `sort_values` 方法：

```
In [242]: obj = pd.Series([4, 7, -3, 2])

In [243]: obj.sort_values()
Out[243]:
2   -3
3    2
0    4
1    7
dtype: int64
```

在排序时，任何缺失值默认都会放到 Series 的末尾：

```
In [244]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [245]: obj.sort_values()
Out[245]:
4   -3.0
5    2.0
0    4.0
```

```
2    7.0
1    NaN
3    NaN
dtype: float64
```

使用 na_position 选项可以将缺失值排在最前面：

```
In [246]: obj.sort_values(na_position="first")
Out[246]:
1    NaN
3    NaN
4   -3.0
5    2.0
0    4.0
2    7.0
dtype: float64
```

当对 DataFrame 排序时，可以使用一列或多列中的数据作为排序键。将一个或多个列名传递给 sort_values 即可实现：

```
In [247]: frame = pd.DataFrame({"b": [4, 7, -3, 2], "a": [0, 1, 0, 1]})

In [248]: frame
Out[248]:
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1

In [249]: frame.sort_values("b")
Out[249]:
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

要根据多个列进行排序，传入列名的列表即可：

```
In [250]: frame.sort_values(["a", "b"])
Out[250]:
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

排名从数组中的最小值开始，从 1 一直到数组中有效数据的数量。接下来介绍 Series 和 DataFrame 的 rank 方法。默认情况下，rank 是通过“为各组分配平均排名”的方式破坏平级关系的：

```
In [251]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [252]: obj.rank()
Out[252]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

也可以根据值在原数据中出现的顺序给出排名：

```
In [253]: obj.rank(method="first")
Out[253]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

这里，条目 0 和 2 没有使用平均排名 6.5，它们被设成了 6.0 和 7.0，因为数据中标签 0 位于标签 2 的前面。

你也可以按降序进行排名：

```
In [254]: obj.rank(ascending=False)
Out[254]:
0    1.5
1    7.0
2    1.5
3    3.5
4    5.0
5    6.0
6    3.5
dtype: float64
```

表 5-6 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 [255]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],
   .....:                       "c": [-2, 5, 8, -2.5]})

In [256]: frame
Out[256]:
     b  a    c
0  4.3  0 -2.0
```

```
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5

In [257]: frame.rank(axis="columns")
Out[257]:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

表 5-6：排名中打破平级关系的方法

方法	说明
"averaget"	默认：在每个组中分配平均排名
"min"	对整组使用最小排名
"max"	对整组使用最大排名
"first"	按值在原始数据中出现的顺序分配排名
"dense"	类似于 method="min"，但排名在组间增加 1, 而不是组中相等元素的数量

5.2.7 带有重复标签的轴索引

直到目前为止，几乎所有示例的轴标签（索引值）都是唯一的。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。观察下面这个带有重复索引值的 Series：

```
In [258]: obj = pd.Series(np.arange(5), index=["a", "a", "b", "b", "c"])

In [259]: obj
Out[259]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

索引的 is_unique 属性可以告诉我们索引值是不是唯一的：

```
In [260]: obj.index.is_unique
Out[260]: False
```

对于带有重复值的索引，数据选取操作将会有些不同。如果某个标签对应多个项，则返回 Series；如果对应单个项，则返回标量值：

```
In [261]: obj["a"]
Out[261]:
```

```
a    0
a    1
dtype: int64

In [262]: obj["c"]
Out[262]: 4
```

这样会使代码变复杂，因为索引的输出类型会根据标签是不是重复的而发生变化。

对 DataFrame 的行（或列）进行索引时也是如此：

```
In [263]: df = pd.DataFrame(np.random.standard_normal((5, 3)),
   .....:                   index=["a", "a", "b", "b", "c"])

In [264]: df
Out[264]:
          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
c -0.577087  0.124121  0.302614

In [265]: df.loc["b"]
Out[265]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [266]: df.loc["c"]
Out[266]:
0   -0.577087
1    0.124121
2    0.302614
Name: c, dtype: float64
```

5.3 描述性统计的汇总和计算

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或汇总统计，用于从 Series 中提取单个值（如 sum 或 mean）或从 DataFrame 的行或列中提取 Series。与对应的 NumPy 数组方法相比，它们都内置有处理缺失数据的功能。来看一个小型 DataFrame：

```
In [267]: df = pd.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 [268]: df
Out[268]:
```

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

调用 DataFrame 的 sum 方法会返回包含列之和的 Series：

```
In [269]: df.sum()
Out[269]:
one    9.25
two   -5.80
dtype: float64
```

传入 axis="columns" 或 axis=1 会对各行进行跨列求和运算：

```
In [270]: df.sum(axis="columns")
Out[270]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

如果某行或某列全为 NA 值，则和为 0。但如果既有 NA 值也有非 NA 值，则会自动跳过 NA 值。如果不想跳过，可以通过 skipna 选项设置。如果某行或某列有 NA 值，则结果就是 NA 值：

```
In [271]: df.sum(axis="index", skipna=False)
Out[271]:
one   NaN
two   NaN
dtype: float64

In [272]: df.sum(axis="columns", skipna=False)
Out[272]:
a     NaN
b    2.60
c     NaN
d   -0.55
dtype: float64
```

某些连接方法，比如 mean，要求至少有一个非 NA 值：

```
In [273]: df.mean(axis="columns")
Out[273]:
a    1.400
b    1.300
c      NaN
d   -0.275
dtype: float64
```

表 5-7 列出了这些约简方法的常用选项。

表 5-7：约简方法的常用选项

方法	说明
axis	约简的轴，DataFrame 的行用 "index"，列用 "columns"
skipna	排除缺失值，默认为 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 [274]: df.idxmax()
Out[274]:
one    b
two    d
dtype: object
```

另一些方法则是累计型的：

```
In [275]: df.cumsum()
Out[275]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

还有一些方法既不是约简型也不是累计型。describe 就属于此类，它能一次性生成多个汇总统计：

```
In [276]: df.describe()
Out[276]:
            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 [277]: obj = pd.Series(["a", "a", "b", "c"] * 4)

In [278]: obj.describe()
Out[278]:
count     16
unique     3
```

```
top        a
freq       8
dtype: object
```

表 5-8 列出了所有与描述性统计和汇总统计相关的方法。

表 5-8：描述性统计和汇总统计

方法	说明
count	非 NA 值的数量
describe	计算汇总统计信息
min、max	计算最小值和最大值
argmin、argmax	计算最小值和最大值的索引位置（整数）；在 Data Frame 对象上不可用
idxmin、idxmax	计算最小值和最大值的索引标签
quantile	计算样本 0～1 之间的分位数（默认为 0.5）
sum	所有值的总和
mean	所有值的平均数
median	所有值的算术中位数（50% 分位数）
mad	平均值的平均绝对偏差
var	所有值的样本方差
std	所有值的样本标准差
skew	所有值的样本偏度（三阶矩）
kurt	所有值的样本峰度（四阶矩）
cumsum	所有值的累计和
cummin、cummax	所有值的累计最小值和累计最大值
cumprod	所有值的累计积
diff	计算一阶差分（对时间序列很有用）
pct change	计算百分比变化

5.3.1 相关系数与协方差

有些汇总统计（如相关系数和协方差）是通过计算成对的参数得到的。我们来看几个 DataFrame，数据来自 Yahoo! Finance 的股票价格和成交量，可以从本书附带的数据集找到二进制 Python 序列化文件：

```
In [279]: price = pd.read_pickle("examples/yahoo_price.pkl")

In [280]: volume = pd.read_pickle("examples/yahoo_volume.pkl")
```

现在计算股价的百分比变化，有关股票这类时间序列的操作会在第 11 章展开介绍：

```
In [281]: returns = price.pct_change()

In [282]: returns.tail()
Out[282]:
                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

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。相应地，cov 用于计算协方差：

```
In [283]: returns["MSFT"].corr(returns["IBM"])
Out[283]: 0.49976361144151144

In [284]: returns["MSFT"].cov(returns["IBM"])
Out[284]: 8.870655479703546e-05
```

因为 MSTF 是一个合理的 Python 变量名，我们还可以用更简洁的语法选择列：

```
In [285]: returns["MSFT"].corr(returns["IBM"])
Out[285]: 0.49976361144151144
```

另一方面，DataFrame 的 corr 和 cov 方法将以 DataFrame 的形式分别返回完整的相关系数和协方差矩阵：

```
In [286]: returns.corr()
Out[286]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [287]: returns.cov()
Out[287]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

利用 DataFrame 的 corrwith 方法，可以计算一个 DataFrame 中列或行与另一个 Series 或 DataFrame 之间的相关系数。传入一个 Series 时，将会返回一个按列计算的相关系数值的 Series：

```
In [288]: returns.corrwith(returns["IBM"])
Out[288]:
```

```
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

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

```
In [289]: returns.corrwith(volume)
Out[289]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

传入 `axis="columns"` 即可按行进行计算。无论哪种情况，在计算相关系数之前，所有的数据项都会按标签对齐。

5.3.2 唯一值、计数以及成员属性

另一类方法是提取一维 Series 中包含的值的信息。看下面的例子：

```
In [290]: obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
```

第一个函数是 `unique`，用于生成 Series 中的唯一值的数组：

```
In [291]: uniques = obj.unique()

In [292]: uniques
Out[292]: array(['c', 'a', 'd', 'b'], dtype=object)
```

返回的唯一值不一定是排好序的，如果需要的话，可以对结果再次进行排序（使用 `uniques.sort()`）。相应地，`value_counts` 用于计算 Series 中各值出现的频次：

```
In [293]: obj.value_counts()
Out[293]:
c    3
a    3
b    2
d    1
dtype: int64
```

为了便于查看，结果 Series 是按频次降序排列的。`value_counts` 是顶级的 pandas 方法，可用于 NumPy 数组或其他 Python 序列：

```
In [294]: pd.value_counts(obj.to_numpy(), sort=False)
Out[294]:
```

```
c    3
a    3
d    1
b    2
dtype: int64
```

isin 可以执行向量化的成员属性检查，可用于以 Series 或 DataFrame 中一列的形式，将数据集过滤为子集：

```
In [295]: obj
Out[295]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [296]: mask = obj.isin(["b", "c"])

In [297]: mask
Out[297]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [298]: obj[mask]
Out[298]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

与 isin 相关的是 Index.get_indexer 方法，它可以提供一个索引数组，将可能包含重复值的数组转换为唯一值的数组：

```
In [299]: to_match = pd.Series(["c", "a", "b", "b", "c", "a"])

In [300]: unique_vals = pd.Series(["c", "b", "a"])
```

```
In [301]: indices = pd.Index(unique_vals).get_indexer(to_match)

In [302]: indices
Out[302]: array([0, 2, 1, 1, 0, 2])
```

表 5-9 列出了这些方法的参考。

表 5-9：唯一值、计数、成员属性方法

方法	说明
isin	计算表示 Series 各值是否包含于传入序列的布尔型数组
match	计算数组中的各值到另一个唯一值数组的整数索引，对于数据对齐和连接类型的操作十分有用
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，唯一值作为索引，频次作为值，频次按降序排列

某些情况下，你可能想计算 DataFrame 中多个相关列的直方图。例如：

```
In [303]: data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],
   .....:                      "Qu2": [2, 3, 1, 2, 3],
   .....:                      "Qu3": [1, 5, 2, 4, 4]})

In [304]: data
Out[304]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

可以计算其中一列的计数，如下所示：

```
In [305]: data["Qu1"].value_counts().sort_index()
Out[305]:
1    1
3    2
4    2
Name: Qu1, dtype: int64
```

要计算所有列的计数，可以将 pandas.value_counts 传给该 DataFrame 的 apply 方法：

```
In [306]: result = data.apply(pd.value_counts).fillna(0)

In [307]: result
Out[307]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
```

```
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

这里，结果中的行标签是所有列的不同值，结果的值是每列中不同值的相应计数。

还有一个 DataFrame.value_counts 方法，它将 DataFrame 的每行当作元组，计算不同行的计数：

```
In [308]: data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})

In [309]: data
Out[309]:
   a  b
0  1  0
1  1  0
2  1  1
3  2  0
4  2  0

In [310]: data.value_counts()
Out[310]:
a  b
1  0    2
2  0    2
1  1    1
dtype: int64
```

这个示例中，结果的索引是将不同的行作为层次化索引，第 8 章会深入讲解层次化索引。

5.4 总结

在下一章中，我们将讨论用 pandas 读取（或加载）和写入数据集的工具。之后，我们会更深入地讨论使用 pandas 进行数据清洗、规整、分析和可视化工具。