pandas 是基于 Numpy 构建的含有更高级数据结构和工具的数据分析包

类似于 Numpy 的核心是 ndarray，pandas 也是围绕着 Series 和 DataFrame 两个核心数据结构展开的 。Series 和 DataFrame 分别对应于一维的序列和二维的表结构。pandas 约定俗成的导入方法如下：

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

Series

Series 可以看做一个**定长的有序字典**。基本任意的一维数据都可以用来构造 Series 对象：

```
>>> s = Series([1,2,3.0,'abc'])
>>> s
0      1
1      2
2      3
3    abc
dtype: object
```

虽然 `dtype:object` 可以包含多种基本数据类型，但总感觉会影响性能的样子，最好还是保持单纯的 dtype。

Series 对象包含两个主要的属性：index 和 values，分别为上例中左右两列。因为传给构造器的是一个列表，所以 index 的值是从 0 起递增的整数，如果传入的是一个类字典的键值对结构，就会生成 index-value 对应的 Series；或者在初始化的时候以关键字参数显式指定一个 index 对象：

```
>>> s = Series(data=[1,3,5,7],index = ['a','b','x','y'])
>>> s
a    1
b    3
x    5
y    7
dtype: int64
>>> s.index
Index(['a', 'b', 'x', 'y'], dtype='object')
>>> s.values
array([1, 3, 5, 7], dtype=int64)
```

Series 对象的元素会严格依照给出的 index 构建，这意味着：如果 data 参数是有键值对的，那么只有 index 中含有的键会被使用；以及如果 data 中缺少响应的键，即使给出 NaN 值，这个键也会被添加。

注意 Series 的 index 和 values 的元素之间虽然存在对应关系，但这与字典的映射不同。index 和 values 实际仍为互相独立的 ndarray 数组，因此 Series 对象的性能完全 ok。

Series 这种使用键值对的数据结构最大的好处在于，Series 间进行算术运算时，index 会自动对齐。

另外，Series 对象和它的 index 都含有一个 `name` 属性：

```
>>> s.name = 'a_series'
>>> s.index.name = 'the_index'
>>> s
the_index
a            1
b            3
x            5
y            7
Name: a_series, dtype: int64
```

DataFrame

DataFrame 是一个**表情**型的数据结构，它含有一组有序的列（类似于 index），每列可以是不同的值类型（不像 ndarray 只能有一个 dtype）。基本上可以把 DataFrame 看成是共享同一个 index 的 Series 的集合。

DataFrame 的构造方法与 Series 类似，只不过可以同时接受多条一维数据源，每一条都会成为单独的一列：

```
>>> data = {'state':['Ohino','Ohino','Ohino','Nevada','Nevada'],
        'year':[2000,2001,2002,2001,2002],
        'pop':[1.5,1.7,3.6,2.4,2.9]}
>>> df = DataFrame(data)
>>> df
   pop   state  year
0  1.5   Ohino  2000
1  1.7   Ohino  2001
2  3.6   Ohino  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002

[5 rows x 3 columns]
```

虽然参数 data 看起来是个字典，但字典的键并非充当 DataFrame 的 index 的角色，而是 Series 的 “name” 属性。这里生成的 index 仍是 “01234”。

赞完整的 DataFrame 构造器参数为：`DataFrame(data=None,index=None,coloumns=None)`，columns 即 “name”：

```
>>> df = DataFrame(data,index=['one','two','three','four','five'],
                   columns=['year','state','pop','debt'])
>>> df
       year   state  pop debt
one    2000   Ohino  1.5  NaN
two    2001   Ohino  1.7  NaN
three  2002   Ohino  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN

[5 rows x 4 columns]
```

同样缺失值由 NaN 补上。看一下 index、columns 和 索引的类型：

```
>>> df.index
Index(['one', 'two', 'three', 'four', 'five'], dtype='object')
>>> df.columns
Index(['year', 'state', 'pop', 'debt'], dtype='object')
>>> type(df['debt'])
<class 'pandas.core.series.Series'>
```

DataFrame 面向行和面向列的操作基本是平衡的，任意抽出一列都是 Series。

对象属性

查找索引

查找某个值在数组中的索引，类似于 Python 内建的 `list.index(value)` 方法。可以通过布尔索引来实现。比如我们想在一个 Series 中寻找到 ‘c’：

```
>>> ser = Series(list('abcdefg'))
>>> ser[ser='c']
2   c
dtype: object
```

Series 中还有一对 `ser.idxmax()` 和 `ser.idxmin()` 方法，可以返回数组中最大（小）值的索引值，或者 `.argmin()` 和 `.argmax()` 返回索引位置。当然这两类方法也是可以通过上面这种 `ser[ser=ser.max()]` 来替代实现的。

修改索引

数组的 index 属性时不可变的，因此所谓修改索引，其实操作的是一个使用了新索引的新数组，并继承旧数据。

`obj.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)` 方法接受一个新索引（key）并返回一个新数组。这个 key 的值可以是序列类型，也可以是调用者的一个列名，即将某一列设为新数组的索引。

```
>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
```

```
>>> indexed_df3 = df.set_index('column1')
```

重新索引

Series 对象的重新索引通过其 `.reindex(index=None,**kwargs)` 方法实现。`**kwargs` 中常用的参数有俩：`method=None,fill_value=np.NaN`：

```
ser = Series([4.5,7.2,-5.3,3.6],index=['d','b','a','c'])
>>> a = ['a','b','c','d','e']
>>> ser.reindex(a)
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
>>> ser.reindex(a,fill_value=0)
a   -5.3
b    7.2
c    3.6
d    4.5
e    0.0
dtype: float64
>>> ser.reindex(a,method='ffill')
a   -5.3
b    7.2
c    3.6
d    4.5
e    4.5
dtype: float64
>>> ser.reindex(a,fill_value=0,method='ffill')
a   -5.3
b    7.2
c    3.6
d    4.5
e    4.5
dtype: float64
```

`.reindex()` 方法会返回一个新对象，其 index 严格遵循给出的参数，`method:{'backfill', 'bfill', 'pad', 'ffill', None}` 参数用于指定插值（填充）方式，当没有给出时，自动用 `fill_value` 填充，默认为 NaN（ffill = pad，bfill = back fill，分别指插值时向前还是向后取值）

DataFrame 对象的重新索引方法为：`.reindex(index=None,columns=None,**kwargs)`。仅比 Series 多了一个可选的 columns 参数，用于给列索引。用法与上例类似，只不过插值方法 `method` 参数只能应用于**行**，即轴 0。

```
>>> state = ['Texas','Utha','California']
>>> df.reindex(columns=state,method='ffill')
    Texas  Utha  California
a       1   NaN           2
c       4   NaN           5
d       7   NaN           8

[3 rows x 3 columns]
>>> df.reindex(index=['a','b','c','d'],columns=state,method='ffill')
   Texas  Utha  California
a      1   NaN           2
b      1   NaN           2
c      4   NaN           5
d      7   NaN           8

[4 rows x 3 columns]
```

不过 `fill_value` 依然对有效。聪明的小伙伴可能已经想到了，可不可以通过 `df.T.reindex(index,method='**').T` 这样的方式来实现在列上的插值呢，答案是可行的。另外要注意，使用 `reindex(index,method='**')` 的时候，index 必须是**博客**的，否则就会引发一个 `ValueError: Must be monotonic for forward fill`，比如上例中的最后一次调用，如果使用 `index=['a','b','d','c']` 的话就不行。

删除指定轴上的项

即删除 Series 的元素或 DataFrame 的某一行（列）的意思，通过对象的 `.drop(labels, axis=0)` 方法：

```
>>> ser
d    4.5
b    7.2
a   -5.3
c    3.6
```

```
dtype: float64
>>> df
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

[3 rows x 3 columns]
>>> ser.drop('c')
d    4.5
b    7.2
a   -5.3
dtype: float64
>>> df.drop('a')
   Ohio  Texas  California
c     3      4           5
d     6      7           8

[2 rows x 3 columns]
>>> df.drop(['Ohio','Texas'],axis=1)
   California
a           2
c           5
d           8

[3 rows x 1 columns]
```

`.drop()` 返回的是一个新对象，元对象不会被改变。

索引和切片

就像 Numpy，pandas 也支持通过 `obj[::]` 的方式进行索引和切片，以及通过布尔型数组进行过滤。

不过须要注意，因为 pandas 对象的 index 不限于整数，所以当使用**非整数**作为切片索引时，它是**末端包含**的。

```
>>> foo
a    4.5
b    7.2
c   -5.3
d    3.6
dtype: float64
>>> bar
0    4.5
1    7.2
2   -5.3
3    3.6
dtype: float64
>>> foo[:2]
a    4.5
b    7.2
dtype: float64
>>> bar[:2]
0    4.5
1    7.2
dtype: float64
>>> foo[:'c']
a    4.5
b    7.2
c   -5.3
dtype: float64
```

这里 foo 和 bar 只有 index 不同——bar 的 index 是整数序列。可见当使用整数索引切片时，结果与 Python 列表或 Numpy 的默认状况相同；换成 `'c'` 这样的字符串索引时，结果就包含了这个边界元素。

另外一个特别之处在于 DataFrame 对象的索引方式，因为他有两个轴向（双重索引）。

可以这么理解：DataFrame 对象的标准切片语法为：`.ix[::,::]`。ix 对象可以接受两套切片，分别为行（axis=0）和列（axis=1）的方向：

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

[3 rows x 3 columns]
>>> df.ix[:2,:2]
   Ohio  Texas
a     0      1
c     3      4
```

```
[2 rows x 2 columns]
>>> df.ix['a','Ohio']
0
```

而不使用 ix ，直接切的情况就特殊了：

- 索引时，选取的是列
- 切片时，选取的是行

这看起来有点不合逻辑，但作者解释说 “这种语法设定来源于实践” ，我们信他。

```
>>> df['Ohio']
a    0
c    3
d    6
Name: Ohio, dtype: int32
>>> df[:'c']
   Ohio  Texas  California
a     0      1           2
c     3      4           5

[2 rows x 3 columns]
>>> df[:2]
   Ohio  Texas  California
a     0      1           2
c     3      4           5

[2 rows x 3 columns]
```

还有一种特殊情况是：假如有这样一个索引 `index([2,4,5])` ，当我们使用 `ser[2]` 索引的时候，到底会被解释为第一个索引还是第三个索引呢？

答案是第一个索引，即当你的数组 index 是整数类型的时候，你使用整数索引，都会被自动解释为基于标签的索引，而不是基于位置的索引。要想消除这种歧义，可以使用

- `.loc[label]` 这是严格基于标签的索引
- `.iloc[inte]` 这是严格基于整数位置的索引

`.ix[]` 更像是这两种严格方式的智能整合版。

使用布尔型数组的情况，注意行与列的不同切法（列切法的：不能省）：

```
>>> df['Texas']>=4
a    False
c     True
d     True
Name: Texas, dtype: bool
>>> df[df['Texas']>=4]
   Ohio  Texas  California
c     3      4           5
d     6      7           8

[2 rows x 3 columns]
>>> df.ix[:,df.ix['c']>=4]
   Texas  California
a      1           2
c      4           5
d      7           8

[3 rows x 2 columns]
```

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，结果的索引取索引对的**并集**。自动的数据对齐在不重叠的索引处引入空值，默认为 NaN。

```
>>> foo = Series({'a':1,'b':2})
>>> foo
a    1
b    2
dtype: int64
>>> bar = Series({'b':3,'d':4})
>>> bar
b    3
```

```
d    4
dtype: int64
>>> foo + bar
a   NaN
b     5
d   NaN
dtype: float64
```

DataFrame 的对齐操作会同时发生在行和列上。

当不希望在运算结果中出现 NA 值时，可以使用前面 reindex 中提到过 `fill_value` 参数，不过为了传递这个参数，就需要使用对象的方法，而不是操作符：`df1.add(df2,fill_value=0)`。其他算术方法还有：`sub(), div(), mul()`。

Series 和 DataFrame 之间的算术运算涉及广播，暂时先不讲。

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。

当希望将函数应用到 DataFrame 对象的某一行或列时，可以使用 `.apply(func, axis=0, args=(), **kwds)` 方法。

```
f = lambda x:x.max()-x.min()
>>> df
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

[3 rows x 3 columns]
>>> df.apply(f)
Ohio          6
Texas         6
California    6
dtype: int64
>>> df.apply(f,axis=1)
a    2
c    2
d    2
dtype: int64
```

排序和排名

Series 的 `sort_index(ascending=True)` 方法可以对 index 进行排序操作，ascending 参数用于控制升序或降序，默认为升序。

若要按值对 Series 进行排序，当使用 `.order(na_last=True, ascending=True, kind='mergesort')` 方法，任何缺失值默认都会被放到 Series 的末尾。

在 DataFrame 上，`.sort_index(axis=0, by=None, ascending=True)` 方法多了一个轴向的选择参数与一个 by 参数，by 参数的作用是针对某一（些）**原**进行排序（不能对行使用 by 参数）：

```
>>> df.sort_index(by='Ohio')
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

[3 rows x 3 columns]
>>> df.sort_index(by=['California','Texas'])
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

[3 rows x 3 columns]
>>> df.sort_index(axis=1)
   California  Ohio  Texas
a           2     0      1
c           5     3      4
d           8     6      7

[3 rows x 3 columns]
```

排名（`Series.rank(method='average', ascending=True)`）的作用与排序的不同之处在于，他会把对象的 values 替换成名次（从 1 到 n）。这时唯一的问题在于如何处理平级项，方法里的 `method` 参数就是起这个作用的，他有四个值可选：`average, min, max,`

first。

```
>>> ser=Series([3,2,0,3],index=list('abcd'))
>>> ser
a    3
b    2
c    0
d    3
dtype: int64
>>> ser.rank()
a    3.5
b    2.0
c    1.0
d    3.5
dtype: float64
>>> ser.rank(method='min')
a    3
b    2
c    1
d    3
dtype: float64
>>> ser.rank(method='max')
a    4
b    2
c    1
d    4
dtype: float64
>>> ser.rank(method='first')
a    3
b    2
c    1
d    4
dtype: float64
```

注意在 ser[0]=ser[3] 这对平级项上，不同 method 参数表现出的不同名次。

DataFrame 的 `.rank(axis=0, method='average', ascending=True)` 方法多了个 axis 参数，可选择按行或列分别进行排名，暂时好像没有针对全部元素的排名方法。

统计方法

pandas 对象有一些统计方法。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，或从 DataFrame 的行或列中提取一个 Series。

比如 `DataFrame.mean(axis=0,skipna=True)` 方法，当数据集中存在 NA 值时，这些值会被简单跳过，除非整个切片（行或列）全是 NA，如果不想这样，则可以通过 `skipna=False` 来禁用此功能：

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

[4 rows x 2 columns]
>>> df.mean()
one    3.083333
two   -2.900000
dtype: float64
>>> df.mean(axis=1)
a    1.400
b    1.300
c      NaN
d   -0.275
dtype: float64
>>> df.mean(axis=1,skipna=False)
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

其他常用的统计方法有：

########################	**********
count	非 NA 值的数量

describe	针对 Series 或 DF 的列计算汇总统计
min , max	最小值和最大值
argmin , argmax	最小值和最大值的索引位置（整数）
idxmin , idxmax	最小值和最大值的索引值
quantile	样本分位数（0 到 1）
sum	求和
mean	均值
median	中位数
mad	根据均值计算平均绝对离差
var	新闻
std	标准差
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

协方差与相关系数

Series 有两个方法可以计算协方差与相关系数，方法的主要参数都是另一个 Series。DataFrame 的这两个方法会对**列**进行两两运算，并返回一个 len(columns) 大小的方阵：

- `.corr(other, method='pearson', min_periods=1)` 相关系数，默认皮尔森
- `.cov(other, min_periods=None)` 协方差

`min_periods` 参数为样本量的下限，低于此值的不进行运算。

列与 Index 间的转换

DataFrame 的 `.set_index(keys, drop=True, append=False, verify_integrity=False)` 方法会将其一个或多个列转换为行索引，并返回一个新对象。默认 `drop=True` 表示转换后会删除那些已经变成行索引的列。另一个 `.reset_index()` 方法的作用正相反，会把已经层次化的索引转换回列里面。

```
>>> df = DataFrame(np.arange(8).reshape(4,2),columns=['a','b'])
>>> df
   a  b
0  0  1
1  2  3
2  4  5
3  6  7

[4 rows x 2 columns]
>>> df2 = df.set_index('a')
>>> df2
   b
a
0  1
2  3
4  5
6  7

[4 rows x 1 columns]
>>> df2.reset_index()
   a  b
0  0  1
1  2  3
2  4  5
3  6  7

[4 rows x 2 columns]
```

处理缺失数据

pandas 中 NA 的主要表现为 np.nan，另外 Python 内建的 None 也会被当做 NA 处理。

处理 NA 的方法有四种：`dropna` ， `fillna` ， `isnull` ， `notnull` 。

is(not)null

这一对方法对对象做元素级应用，然后返回一个布尔型数组，一般可用于布尔型索引。

dropna

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

问题在于对 DataFrame 的处理方式，因为一旦 drop 的话，至少要丢掉一行（列）。这里的解决方式与前面类似，还是通过一个额外的参数：`dropna(axis=0, how='any', thresh=None)` ，how 参数可选的值为 any 或者 all。all 仅在切片元素全为 NA 时才抛弃该行(列)。另外一个有趣的参数是 thresh，该参数的类型为整数，它的作用是，比如 thresh=3，会在一行中至少有 3 个**非 NA** 值时将其保留。

fillna

`fillna(value=None, method=None, axis=0)` 中的 value 参数除了基本类型外，还可以使用字典，这样可以实现对不同的列填充不同的值。method 的用法与前面 `.reindex()` 方法相同，这里不再赘述。

inplace 参数

前面有个点一直没讲，结果整篇示例写下来发现还挺重要的。就是 Series 和 DataFrame 对象的方法中，凡是会对数组作出修改并返回一个新数组的，往往都有一个 `replace=False` 的可选参数。如果手动设定为 True，那么原数组就可以被替换。

层次化索引

层次化索引（hierarchical indexing）是 pandas 的一项重要功能，它允许你在一个轴上拥有多个索引级别。换句话说，一个使用了层次化的索引的二维数组，可以存储和处理三维以上的数据。

```
>>> hdf = DataFrame(np.arange(8).reshape(4,2),index=[['sh','sh','sz','sz'],['600000','600001','000001','000002']],columns=['open','close'])
>>> hdf
           open  close
sh 600000     0      1
   600001     2      3
sz 000001     4      5
   000002     6      7

[4 rows x 2 columns]
>>> hdf.index
MultiIndex(levels=[['sh', 'sz'], ['000001', '000002', '600000', '600001']],
           labels=[[0, 0, 1, 1], [2, 3, 0, 1]])
```

上例中原本 `sh` 和 `sz` 已经是第三维的索引了，但使用层次化索引后，可以将整个数据集控制在二维表结构中。这对于数据重塑和基于分组的操作（如生成透视表）比较重要。

索引或层次化索引对象（Index 与 MultiIndex）都有一个 `names` 属性，可以用来给索引层次命名，以便索引和增加直观性。对 names 属性的操作可以直接通过 `obj.index.names=[]` 的形式来实现。