01Pandas基本

一、数据结构

`Pandas` 将读取到的数据加载到自己的叫做 `Series` 和 `DataFrame` 的数据结构框架当中，数据一旦进入这两种框架，我们就可以按照这些框架自己的处理方法进行处理，这是对数据的一种高级抽象。

维数	名称	描述
1	Series	带标签的一维同构数组
2	DataFrame	带标签的，大小可变的，二维异构表格

> *注：*
>
> 1. *之前支持三维的面板（Panel）结构已经不再支持。*
> 2. *此数据结构指的是Pandas的数据结构框架，不是《数据结构》课程的数据结构。*

1.理解数据结构

数据结构就像一个存放数据的架子，有多行多列，每个数据在一个格子里，每个格子有自己的编号。比如一个剧场的座位，我们在横向编成 1排、2排、3排等，在纵向编成 1号、2号、3号等，那一个具体的位置就如 4排18号、6排1号等，我们每个人落座后就像一个具体的数据。

数据结构提供了一个数据框架，`Pandas` 不用关注你给它的是什么业务数据，只要符合这个框架就能放进去，它会提供各种针对这个框架的处理方法，你只需要根据你的数据分析需求去使用它。

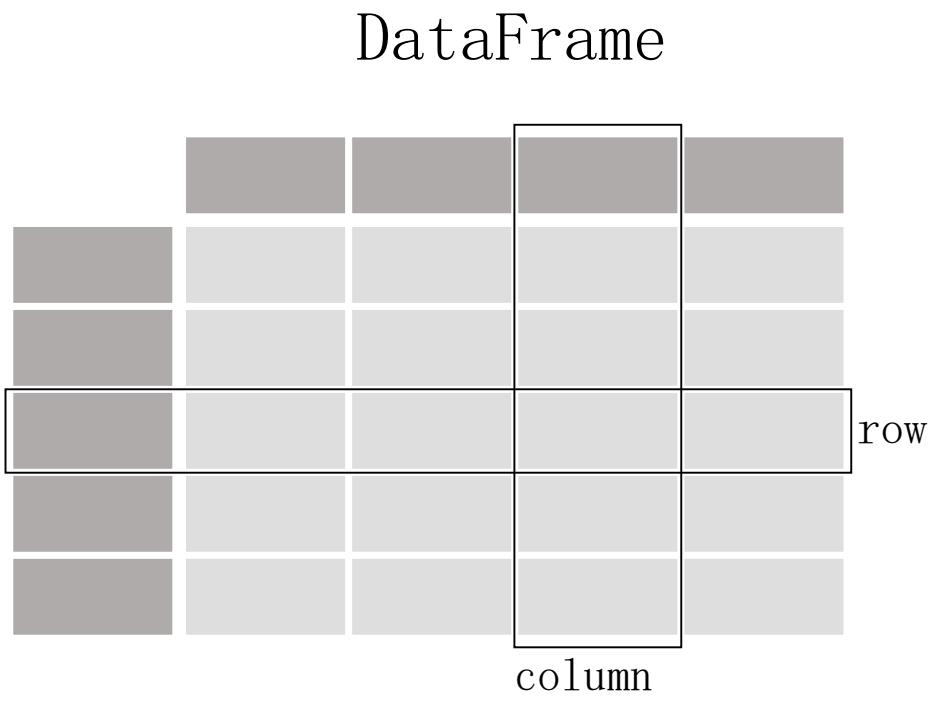
2.Series

`Series`（系列、数列、序列）是一个带有标签的一维数组。以下各国的 GDP 就是一个典型的 `Series`，国家是标签、索引，不是具体的数据，起到解释数据的作用。

```
中国  14.22
美国  21.34
日本  5.18
dtype: float64
```

带有同样标签和索引的 `Series` 可以组成一个 `DataFrame`，比如我们再增加一个国家的人口，下边会讲到。

3.DataFrame



`DataFrame` 是 `Pandas` 定义的一个二维数据结构。如上图所示：

- 横向的称作行（row），我们所说的一条数据，就是指其中的一行
- 纵向的称作列（column），或者可以叫一个字段，是一条数据的某个值
- 第一行是表头，或者可叫字段名，类型 `Python` 字典里的 key，代码数据的属性
- 第一列是索引（index），就是这行数据所描述的主体，也是这条数据的关键
- 表头和索引在一些场景下也有称列索引和行索引的
- 行索引和列索引可能会出外多层索引的情况，后边会遇到

熟悉了上边的概念，我们来看看一个具体的 `DataFrame`：

```
      人口    GDP
中国  14.33  14.22
美国   3.29  21.34
日本   1.26   5.18
```

这个例子是在 `Series` 的例子上演化而来，其中：

- 共有三行两列（不包含索引）数据
- 国家那一列是这个表的索引，每一行数据就是针对这个国家的
- 每条数据有两个值，分别是人口和 GDP

这就是一个典型的 `DataFrame` 结构。

4.Numpy

`Numpy` 是一个高效的科学计算库，`Pandas` 的这些数据结构是构建在 `Numpy` 数组之上，所以处理速度非常快。我们在之前的课程中已经学习了`Numpy`的基本用法。

5.特点

`Series` 和 `DataFrame` 里边的值都是可变的，它们都可以增加行，并排序，`Series` 只有一列不能再增加，`DataFrame` 可以增加列。

我们在处理数据时，不要对原始数据及转入进来的初始数据（如 `DataFrame` 一般变量为 `df`）进行改动，而是复制生成新的对象，或者使用方法链，这样比较稳妥。试想，你如果改变了上边两个步骤的数据如果分析方法错误就会增加调整成本，如果是源数据集有可能造成无源挽回的损失。

二、快速入门

今后我们处理的数据基本上是 `Pandas` 的 `DataFrame` 和 `Series`，其中 `DataFrame` 是 `Series` 的容器，所以需要掌握数据生成方法。现在我们学习一下制造一些简单数据放入 `DataFrame` 和 `Series`，后边我们会单独讲解从文件如 Excel 中读取生成数据。

1.引入 Pandas

我们在使用 `Pandas` 时，需要将导入：

```
import pandas as pd
```

这里我们给它取了一个别名 `pd`,起别名是因为 `Pandas` 这个单词有点长，在代码中会经常出现，这样会简化些，减少代码量。别名可以自由取名，但 `pd` 是 `Pandas` 的缩写，已经约定俗成，方便自己和别人能看懂你的代码，所以建议不要起别的名字。

如果出现 `ModuleNotFoundError: No module named 'pandas'` 错误，说明没有安装 `Pandas` 模块，可以使用 `pip install pandas` 或`conda install pandas` 安装。

如果执行导入库后，没有任务返回内容，说明库导入正常，可以用 `pd` 使用 `Pandas` 了。

同样，有时如果需要 `Numpy` 则用以下代码引入并起别名 `np`

```
import numpy as np
```

2.创建DataFrame

使用 `pd.DataFrame()` 可以创建一个 `DataFrame`，然后用 `df` 做为变量赋值给它。`df` 是指 `DataFrame`，也是约定俗成建议尽量使用。

```
df = pd.DataFrame({'国家': ['中国', '美国', '日本'],
                   '地区': ['亚洲', '北美', '亚洲'],
                   '人口': [14.33, 3.29, 1.26],
                   'GDP': [14.22, 21.34, 5.18],
                  })
df
'''
   国家  地区     人口    GDP
0  中国  亚洲  14.33  14.22
1  美国  北美   3.29  21.34
2  日本  亚洲   1.26   5.18
'''
```

可以看到，我们成功生成了一个 `DataFrame`：

- 共有四列数据， 国家、地区、人口、GDP
- 四列数据国家和地区是文本类型，人口和 GDP 是数字
- 共三行数据，系统为我们自动加了索引 0、1、2

我们知道，`DataFrame` 可以容纳 `Series`，所以我们在定义 `DataFrame` 时可以使用 `Series`，同时也可以利用 `Numpy` 的方法：

```
df2 = pd.DataFrame({'A': 1.,
                    'B': pd.Timestamp('20130102'),
                    'C': pd.Series(1, index=list(range(4)),
dtype='float32'),
```

```
                    'D': np.array([3] * 4, dtype='int32'),
                    'E': pd.Categorical(["test", "train",
"test", "train"]),
                    'F': 'foo'})

df2
'''
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
'''
```

注：`df.<TAB>` 即输入变量名并输入 `.` 后按 tab 键会提示此对象的所有方法和属性，这是 `IPython` 的功能，也是一个通用的功能，所有的对象和变量都支持。

3.生成Series

我们从上例 `df` 中取一列：

```
df['人口']
'''
0    14.33
1     3.29
2     1.26
Name: 人口, dtype: float64
'''
```

从 `DataFrame` 中选取一列就会返回一个 `Series`，当然选择多列的话依然是 `DataFrame`。下边我们单独创建一个 `Series`：

```
gdp = pd.Series([14.22, 21.34, 5.18], name='gdp')
gdp
'''
0    14.22
1    21.34
2     5.18
Name: gdp, dtype: float64
'''
```

我们生成了一个名叫 `gdp` 的 `Series`，我们没有指定索引，系统自动给我们补上 0、1、2 ... 作为索引。

4.对象的操作

上边我们定义了两种对象，对这两种对象，我们可以进行相关的操作，以下是两种类型的 `describe` 方法，用于对数据进行整体描述：

```
df.describe()
'''
              人口        GDP
count   3.000000   3.000000
mean    6.293333  13.580000
std     7.033579   8.098988
min     1.260000   5.180000
25%     2.275000   9.700000
50%     3.290000  14.220000
75%     8.810000  17.780000
max    14.330000  21.340000
'''

gdp.describe()
'''
count     3.000000
mean     13.580000
std       8.098988
min       5.180000
25%       9.700000
50%      14.220000
75%      17.780000
max      21.340000
Name: gdp, dtype: float64
'''
```

最大值：

```
df.max()
'''
国家         美国
地区         北美
人口      14.33
GDP     21.34
dtype: object
'''

gdp.max()
# 21.34
```

查看类型：

```
type(s) # pandas.core.series.Series
type(df) # pandas.core.frame.DataFrame
```

所以，对于 `Series` 和 `DataFrame` 很多方法都适用，但计算的维度不一样。

三、Series序列

`Series` 是一个一维的带有标签的数组，这个数据可以由任何类型数据构成，包括整型、浮点、字符、`Python` 对象等。它的轴标签被称为「索引」，它是 `Pandas` 最基础的数据结构。

1.创建

`Series` 的创建方式如下：

```
s = pd.Series(data, index=[])
```

其中：

- `data` 可以是 `Python` 对象、`Numpy` 的 `ndarray` 、一个标量（定值，如 8）
- `index` 索引是轴上的一个列表，必须和 `data` 的长度相同，如果没有指定则自动从 0 开始，`[0, ..., len(data) - 1]`

2.数据

上文中的 `data` 可以取以下方法的值：

（1）列表元组

列表和元组可以直接放入 `pd.Series()`：

```
pd.Series(['a', 'b', 'c', 'd', 'e'])
pd.Series(('a', 'b', 'c', 'd', 'e'))
```

（2）ndarray

可以是 `Numpy` 的 `ndarray`:

```
# 由索引为 a、b.. ，  五个随机浮点数数组组成
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd',
'e'])
s.index # 查看索引
s = pd.Series(np.random.randn(5)) # 未指定索引
```

（3）字典 dict

使用字典来生成后，key 为索引，value 为内容，顺序为字典内容的顺序：

```
d = {'b': 1, 'a': 0, 'c': 2}
s = pd.Series(d)
```

```
'''
[out]:
b    1
a    0
c    2
dtype: int64
'''


#  如果指定索引，则会按索引顺序，如无法与索引对应的会产生缺失值
pd.Series(d, index=['b', 'c', 'd', 'a'])
'''
Out:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
'''

```

（4）标量（scalar value）

一个具体的值，如果不指定索引长度为 1，指定索引后长度为索引的数量，每个索引的值都是它。

```
pd.Series(5.)
'''
Out:
0    5.0
'''
dtype: float64
# 指定索引
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
'''
Out:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
'''

```

3.Series 操作

`Series` 的操作非常灵活，以下介绍的操作方法是对照类似的其他数据类型，可以应用在所有 `Series` 上，不管 `Series` 是从何种方式定义生成的。

（1）类似 ndarray 操作

`Series` 操作与 `ndarray` 类似，支持切片。

```
s = pd.Series([1,2,3,4,5,6,7,8])
s[3] # 类似列表切片
s[2:]
s.median() # 平均值，包括其他的数学函数
s[s > s.median()] # 筛选大于平均值的内容
s[[1, 2, 1]] # 指定索引的内容，括号的列表是索引
s.dtype # 数据类型
s.array # 返回值的数列
s.to_numpy() # 转为 numpy 的 ndarray
3 in s # 逻辑运算，检测索引
```

（2）类似字典的操作

```
s = pd.Series([14.22, 21.34, 5.18],
              index=['中国', '美国', '日本'],
              name='人口')

s['中国'] # 14.22 # 根 key 进行取值，如果没有报 KeyError
s['印度'] = 13.54 # 类似字典一样增加一个数据
'法国' in s # False 逻辑运算，检测索引
```

（3）向量计算和标签对齐

```
s = pd.Series([1,2,3,4])
s + s # 同索引相加，无索引位用 NaN 补齐
s * 2 # 同索引相乘
s[1:] + s[:-1] # 选取部分进行计算
np.exp(s) # 求e的幂次方
```

（4）名称属性

`Series` 可以指定一个名称，如无名称不返回内容（`NoneType`）。

```
s = pd.Series([1,2,3,4], name='数字')
s.name # '数字'
s = s.rename("number") # 修改名称
s2 = s.rename("number") # 修改名称并赋值给一个新变量
```

4.其他操作

```
s = pd.Series([1,2,3,4], name='数字')
s.add(1) # 每个元素加1
s.add_prefix(3) # 给索引前加个3，升位
s.add_suffix(4) # 同上，在后增加
s.sum() # 总和
s.count() # 数量，长度
s.agg('std') # 聚合，仅返回标准差，与 s.std() 相同
s.agg(['min', 'max']) # 聚合，返回最大最小值
s.any() # 是否有为假的
s.all() # 是否全是真
s.append(s2) # 追加另外一个 Series
s.apply(lambda x:x+1) # 应用方法
s.empty # 是否为空
s3 = s.copy() # 深拷贝
# 等等，以上是常用的，方法非常多
```

四、DataFrame数据框

数据框/数据帧（`DataFrame`）是二维数据结构，数据以行和列的形式排列表达一定的数据意义。类似于 CSV、Excel 、SQL 结果表，或者由 `Series` 组成。

1.定义方法

```
df = pd.DataFrame(
    data=,
    index = ,
    columns = ,
    dtype = ,
    copy = ,
)
```

说明：

- `data`: 具体数据，结构化或者同构的 `ndarray`、可迭代对象、字典或者 `DataFrame`
- `index`: 索引，类似数组的对象，支持解包，如果没有指定会自动生成 `RangeIndex (0, 1, 2, …, n)`
- `columns`: 列索引，表头，如果没有指定会自动生成 `RangeIndex (0, 1, 2, …, n)`
- `copy`: `bool` 或者 `None`, 默认为 `None`，从输入复制数据。对于 `dict` 数据，默认值 `None` 的行为类似于 `copy=True`。对于 `DataFrame` 或 `2d ndarray` 输入，默认值 `None` 的行为类似于 `copy=False`。

此外还可以 `dtype` 指定数据类型，如果未指定，系统会自动推断。

2.可定义的结构

和 `Series` 一样，`DataFrame` 可以由多种数据类型来定义：

- 一维 `numpy.ndarray`、字典、列表
- 二维 `numpy.ndarray`，三维及以上的不可以
- 结构化的 `ndarray`
- 一个或者多个 `Series`
- 一个或者多个 `DataFrame` 也可以定义

3.创建生成

下边介绍的是利用现有的 `Python` 数据创建一个 `DataFrame`，但是大多数情况下我们都是从数据文件如 CSV、Excel 中取得数据，不过，了解这部分知道可以让我们更好地理解 `DataFrame` 的数据机制。

（1）字典

```
d = {'国家': ['中国', '美国', '日本'],
     '人口': [14.33, 3.29, 1.26]}
df = pd.DataFrame(d)

df
'''
   国家     人口
0  中国  14.33
1  美国   3.29
2  日本   1.26
'''
```

可以指定索引，会覆盖原有的索引：

```
df = pd.DataFrame(d, index=['a', 'b', 'c'])
df
'''
   国家     人口
a  中国  14.33
b  美国   3.29
c  日本   1.26
'''
```

（2）Series 组成的字典

```
d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
     'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c',
'd'])}
df = pd.DataFrame(d)
df
'''
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
'''
```

指定索引和列名，会覆盖原有的列名：

```
pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
'''
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
'''
```

（3）ndarrays 或列表组成的字典

```
d = {'one': [1., 2., 3., 4.],
     'two': [4., 3., 2., 1.]}

pd.DataFrame(d)
'''
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
'''

pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
'''
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
'''
```

（4）同构的数组数据

```
# 创建一个空的 2x3 数组
data = np.zeros((2, ), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])

# 给这个数据填入具体数据值
data[:] = [(1, 2., 'Hello'), (2, 3., "World")]

# 生成 DataFrame
pd.DataFrame(data)
'''
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'
'''

# 指定索引
pd.DataFrame(data, index=['first', 'second'])
'''
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'
'''

# 指定列名
pd.DataFrame(data, columns=['C', 'A', 'B'])
'''
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
'''
```

（5）字典和列表

```
# 定义一个字典列表
data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

# 生成 DataFrame 对象
pd.DataFrame(data2)
'''
   a   b     c
0  1   2   NaN
1  5  10  20.0
'''

# 指定索引
pd.DataFrame(data2, index=['first', 'second'])
'''
        a   b     c
first   1   2   NaN
```

```
second  5  10  20.0
'''

# 指定列名
pd.DataFrame(data2, columns=['a', 'b'])
'''
   a   b
0  1   2
1  5  10
'''
```

（6）元组组成的字典

```
# 一个双索引的例子
pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
              ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
              ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
              ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
              ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
'''
       a             b
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
'''
```

（7）由 Series 生成

可以将多个同索引的 `Series`，生成 `DataFrame`：

```
s1 = pd.Series(['a', 'b', 'c', 'd', 'e'])
pd.DataFrame(s1)
```

五、数据类型

`Pandas` 的数据类型是指某一列的里所有的数据的共性，如果全是数字那么就是类型数字型，其中一个不是数据那么就没法是数字型了。我们知道 `Pandas` 里的一列可以由 `NumPy` 数组组成，事实上大多 `NumPy` 的数据类型就是 `Pandas` 的类型，`Pandas` 也会有自己特有的数据类型。

1.常用 Pandas 特有类型

（1）DatetimeTZDtype

带有时区的日期时间格式。

```
 # 时区为北京时间，单位支持纳秒
d = pd.DatetimeTZDtype("ns", tz='Asia/Shanghai')
pd.Series(['20200501 22:23:22.3432'], dtype=d)
# 0   2020-05-01 22:23:22.343200+08:00
#  dtype: datetime64[ns, Asia/Shanghai]
```

也可以用字符串去指定类型，此字符串可用在所有指定数据类型的地方，所有数据类型道理一样。

```
pd.Series(['20200501 22:23:22.3432'], dtype='datetime64[ns, Asia/Shanghai]')
pd.Series(['20200501 22:23:22.3432'], dtype='datetime64[ns]') # 无时区
```

如果需要指定一个时间定值，可以用 `pd.Timestamp()`:

```
# 用字符形式
pd.Timestamp('2017-01-01T12')
# Timestamp('2017-01-01 12:00:00')

# Unix epoch 指定时间单和时区
pd.Timestamp(1513393355.5, unit='s')
# Timestamp('2017-12-16 03:02:35.500000')
pd.Timestamp(1513393355, unit='s', tz='US/Pacific')
# Timestamp('2017-12-15 19:02:35-0800', tz='US/Pacific')

# 用 datetime.datetime 的方法
pd.Timestamp(2017, 1, 1, 12)
# Timestamp('2017-01-01 12:00:00')
pd.Timestamp(year=2017, month=1, day=1, hour=12)
# Timestamp('2017-01-01 12:00:00')
```

后续会有关于时间日期数据的专门介绍。

（2）CategoricalDtype

定义一个有限的字符枚举类型：

```
t = pd.CategoricalDtype(categories=['b', 'a'], ordered=True)
pd.Series(['a', 'b', 'a', 'c'], dtype=t) # 'c' 不在列表是值会为 NaN
'''
0      a
1      b
2      a
```

```
3    NaN
dtype: category
Categories (2, object): [b < a]
'''

pd.Categorical([1, 2, 3, 1, 2, 3])
# [1, 2, 3, 1, 2, 3]
# Categories (3, int64): [1, 2, 3]

pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
# [a, b, c, a, b, c]
# Categories (3, object): [a, b, c]

c = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'],
ordered=True,
                   categories=['c', 'b', 'a'])
c
# [a, b, c, a, b, c]
# Categories (3, object): [c < b < a]
c.min()
# 'c'
```

（3）PeriodDtype

支持时间周期

```
pd.PeriodDtype(freq='D') # 按天
# period[D]
pd.PeriodDtype(freq=pd.offsets.MonthEnd()) # 按月，最后一天
# period[M]
```

（4）StringDtype

> *警告*
>
> *`pd.StringDtype` 和 `pd.arrays.StringArray` 是 Pandas 1.0 版本新增加的实验性功能，可能在下个版本不兼容，可以尝试试用，不过从官方的态度来看还是极力推荐的。*

支持字符类型数据。

```
pd.Series(['a','b','c'], dtype=pd.StringDtype())
'''
0    a
1    b
2    c
dtype: string
'''
```

后续会有关于文本数据处理的专门介绍。

2.数据类型的检测

可以使用类型测试数据的类型：

```
pd.api.types.is_bool_dtype(s)
pd.api.types.is_categorical_dtype(s)
pd.api.types.is_datetime64_any_dtype(s)
pd.api.types.is_datetime64_ns_dtype(s)
pd.api.types.is_datetime64_dtype(s)
pd.api.types.is_float_dtype(s)
pd.api.types.is_int64_dtype(s)
pd.api.types.is_numeric_dtype(s)
pd.api.types.is_object_dtype(s)
pd.api.types.is_string_dtype(s)
pd.api.types.is_timedelta64_dtype(s)
pd.api.types.is_bool_dtype(s)
```

3.数据类型的转换

数据分析前我们就需要对数据分配好适合的类型，这才能理工高效地处理数据，不用的数据类型可以用不同的处理方法。注意，一个列只能有一个总数据类型，但具体值可以是不同的数据类型。

（1）数据初始化时指定

```
df = pd.DataFrame(data, dtype='float32') # 对所的字段指定类型
 # 每个字段分别指定
df = pd.read_excel(data, dtype={'team': 'string', 'Q1':
'int32'})
```

（2）自动推定类型

`Pandas` 可以用以下方法智能地推定各列的数据类型，以下方法不妨一试：

```
# 自动转换合适的数据类型
df.convert_dtypes() # 推荐！新的方法，支持 string 类型
df.infer_objects()

# 按大体类型推定
m = ['1', 2, 3]
s = pd.to_numeric(s) # 转成数字
pd.to_datetime(m) # 转成时间
pd.to_timedelta(m) # 转成时差
pd.to_datetime(m, errors='coerce') # 错误处理
pd.to_numeric(m, errors='ignore')
pd.to_numeric(m errors='coerce').fillna(0) # 兜底填充
pd.to_datetime(df[['year', 'month', 'day']]) # 组合成日期
```

```
pd.to_numeric(m, downcast='integer')
'''
erros:
    -ignore:只对数字字符串转换，其他忽略不转换
    -raise:遇到非数字字符串报错，时间类型可以转换成int
    -coerce:将时间字符串和bool类型转换成数字，其他均转换成NaN

downcast:指定转换的类型，默认返回float64
    -integer
    -signed
    -unsigned
    -float
'''

```

（3）类型转换 astype()

和`Numpy`的数据类型转换一样，这也是最常见的数据类型转换方式，各数据类型可以参考`Numpy`的数据类型

```
df.dtypes # 查看数据类型
df.index.astype('int64') # 索引类型转换
df.astype('int32') # 所有数据转换为 int32
df.astype({'col1': 'int32'}) # 指定字段转指定类型
s.astype('int64')
s.astype('int64', copy=False) # 不与原数据关联
s.astype(np.uint8)
df['name'].astype('object')
data['Q4'].astype('float')
s.astype('datetime64[ns]')
data['状态'].astype('bool')
```