简介

问题：Numpy已经能够帮助我们处理数据，能够结合Matplotlib解决数据分析的展示问题，那么为什么还需要学习Pandas呢？

- numpy 能够帮我们处理处理数值型数据，但是这还不够
- 很多时候，我们的数据除了数值之外，还有**字符串，还有时间序列等**

1．Pandas是什么?

- Pandas 的名称来自于面板数据（panel data）和 Python 数据分析（data analysis）。
- Pandas 是一个强大的分析结构化数据的工具集，基于 NumPy 构建，提供了 **高级**数据结构和**数据操作工具**，它是使 Python 成为强大而高效的数据分析环境的重要因素之一。
- pandas 是一个开源、BSD 许可的库，为 Python 编程语言提供高性能、易于使用的数据结构和数据分析工具。

2．Pandas有哪些特点?

- 灵活的数据结构：支持 Series（一维）、DataFrame（二维），适配 `90%+` 结构化数据场景
- 高效数据操作：一行代码实现筛选、排序、分组等复杂操作，比原生 Python 效率提升 `10-100` 倍
- 丰富的 IO 能力：支持读取 / 保存 CSV、Excel、SQL、JSON 等 `10 +` 种格式数据
- 无缝集成生态：与 NumPy（数值计算）、Matplotlib（可视化）、Scikit-learn（机器学习）完美兼容

安装

可以通过 pip命令安装 pandas

```
# 安装
$ pip install pandas


# 导入（惯例用pd作为别名）
$ import pandas as pd
```

核心数据结构

Pandas 有两大基础数据结构：**Series（一维）** 和 **DataFrame（二维）**，所有操作均围绕二者展开。

Series

Series：带标签的一维数组。

定义与创建

Series 由“索引（index）”和“数据（values）”组成，索引默认从 `0` 开始递增。

创建主要有以下两种方式：

- 通过列表创建（可以手动指定索引，也可以不指定索引）
- 通过字典创建（key是索引，value是值）

```
import pandas as pd

# Series的定义
# 1. 列表创建，默认索引
s1 = pd.Series([10, 20, 30, 40, 50])
# print(s1)

# 2. 列表创建，自定义索引
s2 = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd',
'e'])
# print(s2)

# 3. 字典创建(key是索引，value是值)
s3 = pd.Series({"Alice":85, "Bob":90, "Charlie":78, "David":92})
# print(s3)
```

```
# 输出:
# Alice      85
# Bob        90
# Charlie    78
# David      92
# dtype: int64   # 数据类型
```

核心属性

核心属性有:

- index:索引
- values:数据
- dtype:数据类型
- shape:形状

```
s3 = pd.Series({"Alice":85, "Bob":90, "Charlie":78, "David":92})


print(s3.index)    # 索引 → Index(['Alice', 'Bob', 'Charlie',
'David'], dtype='object')
print(s3.values)   # 数据 → [85 90 78 92]
print(s3.dtype)    # 数据类型 → int64
print(s3.shape)    # 形状（长度）→ (4,)
```

基本操作

```
# 通过索引获取值
s3 = pd.Series({"Alice":85, "Bob":90, "Charlie":78, "David":92})
print(s3["Alice"])
print(s3["Bob"])

# 通过索引修改值
s3["Bob"] = 100
print(s3)
```

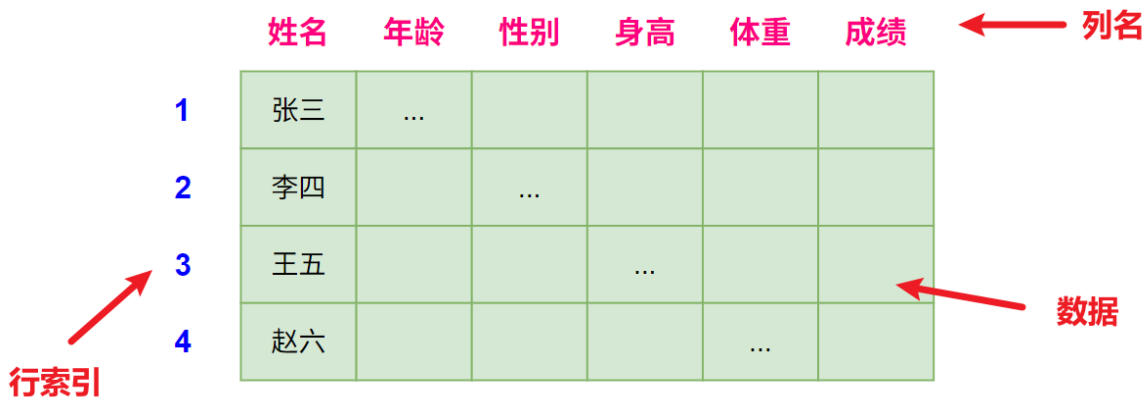
总结:

- Series对象本质上由两个列表构成
- 一个数组构成对象的键(index，索引)
- 一个数组构建对象的值(values)

DataFrame

DataFrame可以理解为是：带标签的二维表格。

DataFrame 类似 Excel 表格，由 “行索引（index）”、“列名（columns）” 和 “数据（values）” 组成，是 Pandas 最常用的数据结构。



定义与创建

DataFrame本质其实就是二维数组（ndarray），创建方式有两种：

- 通过字典创建（键为列名，值为列数据，行索引可以自定义，也可以默认）
- 通过二维的ndarray创建

```python
import pandas as pd
import numpy as np

data = {
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [20, 22, 21, 20],
    "score": [85, 90, 78, 92],
    "gender": ["女", "男", "男", "男"]
}
# 1. 字典创建（字典键为列名，值为列数据）
df = pd.DataFrame(data)
print(df)


# 2. 自定义行索引
df = pd.DataFrame(data,index=['s1', 's2', 's3', 's4'])
print(df)


# 3. 通过ndarray构建Dataframe
```

```python
arr1 = np.arange(12).reshape(3,4)
df2 = pd.DataFrame(arr1)
print(df2)

# df2输出:
#        0   1    2    3              # 行索引
# 0      0   1    2    3
# 1      4   5    6    7
# 2      8   9   10   11
# 列索引


# 注意: 依然具有广播的特性
data2 = {
    "name":['zs','ls','ww'],
    "gender":"男",
    "height":[180,180,190]
}
df2 = pd.DataFrame(data2)
```

核心属性

核心属性有:

- index: 行索引
- columns: 列索引（列名）
- values: 数据
- shape: 形状
- info: 整体概览

```python
import pandas as pd
import numpy as np

# 1. 创建一个DataFrame
data = {
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [20, 22, 21, 20],
    "score": [85, 90, 78, 92],
    "gender": ["女", "男", "男", "男"]
}

df = pd.DataFrame(data)

# 2. 输出核心属性
# 行索引
```

```
print(df.index)      # 输出: RangeIndex(start=0, stop=4, step=1)

# 列索引
print(df.columns)    # 输出: Index(['name', 'age', 'score',
'gender'], dtype='object')

# 数据
print(df.values)     # 输出: [['Alice' 20 85 '女'] ['Bob' 22 90
'男'] ['Charlie' 21 78 '男'] ['David' 20 92 '男']]

# 形状
print(df.shape)      # 输出: (4, 4)

# 数据概览
print(df.info())     # 输出包括: 类型、行索引、列索引、数据、数据类型、内存占
用等
```

基本操作

1. 获取前几行(后几行)
2. 通过索引获取列数据
3. 增加列
4. 删除列

```
dict = {
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [20, 22, 21, 20],
    "score": [85, 90, 78, 92],
    "gender": ["女", "男", "男", "男"]
}
df = pd.DataFrame(dict)

# 1. 获取前几行数据
print(df.head(2))
print('-'*20)
# 2. 获取后几行数据
print(df.tail(1))
print('-'*20)

# 3. 获取指定列的数据
name_data = df["name"]
print(name_data)
print(type(name_data))       # 单列数据是 Series
print('-'*20)

```

```python
# 4. 获取指定多列的数据
age_data = df[["age", "gender"]]
print(age_data)
print(type(age_data))        # 多列数据是 DataFrame
print('-'*20)

# 5. 增加列
df["new_column"] = [1, 2, 3, 4]
# print(df)
df["new_column2"] = df["new_column"] + 20
print(df)
print('-'*20)

# 6. 删除列
del df["new_column"]
print(df)
```

索引操作

索引是 Pandas 高效定位数据的核心，是非常重要的。

学习目标:

1. 掌握Series中的索引的操作
2. 掌握 **label 索引（loc）** 和 **位置索引（iloc）** 的使用与区别
3. 掌握索引切片

Series中的索引操作

1. Index指定索引的名称

```python
import pandas as pd
import numpy as np

# 创建一个Series
s=pd.Series(['A','B','C','D'])

# 1. Index指定索引的名称
s.index = [1,2,3,4]
print(s)
```

2. 根据索引名取值

```
s2 = pd.Series(['A','B','C','D'],index=['001','002','003','004'])

# 通过索引名取值
print(s2['002'])              # B
print(s2.loc['002'])          # B      #推荐的方式
```

3. 根据位置取值

```
s2 = pd.Series(['A','B','C','D'],index=['001','002','003','004'])

#  通过位置取值
# print(s2[1])            # FutureWarning: 已被废弃
print(s2.iloc[1])         # 推荐的方式
```

4. 取多个值

```
# 3. 通过多个索引取多个值
# print(s2[['002','004']])
# print('-'*20)

print(s2.loc[['002','004']])
print('-'*20)

# print(s2[[1,3]])

print(s2.iloc[[1,3]])
print('-'*30)
```

5. 根据切片取值

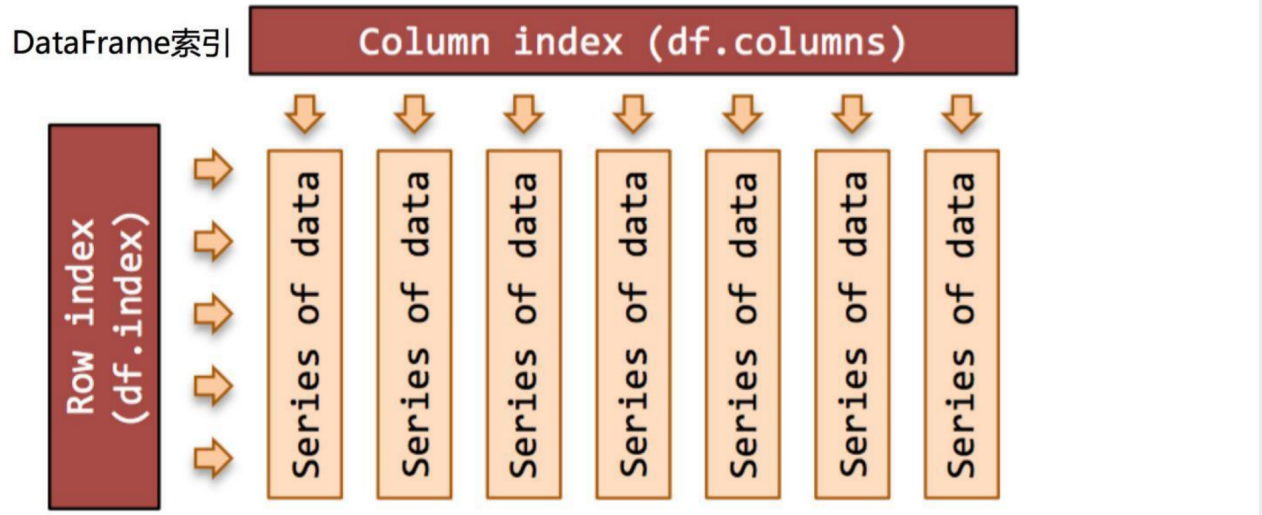
```
print(s2.loc['002':'004'])       # 通过索引名取值是【左闭右闭】
print('-'*20)
print(s2.iloc[0:2])               # 通过索引位置取值是【左闭右开】
```

总结：Series中的取值有两种方式：

1. 根据索引的名字取值
   - 可以直接 `对象名[索引名]` 来取值
   - 建议使用 `对象名.loc[索引名]` 来取值
2. 根据索引的位置取值
   - 可以直接使用 `对象名[位置值]` 来取值
   - 建议使用 `对象名.iloc[位置值]` 来取值
3. 支持使用切片的方式来取值

DataFrame中的索引操作

首先，认识一下DataFrame的索引，分为行索引和列索引，每一列都构成了一个Series对象。




```
df = pd.DataFrame(np.arange(12).reshape(3,4),index=['s1','s2','s3'],columns=['c1','c2','c3','c4'])
print("原始数据:")
print(df)

# 1. 通过列索引获取值
print("通过列索引获取值:")
print(df['c2'])

# 2. 通过多个列索引获取多个列的值
print("通过多个列索引获取多个列的值")
print(df[['c2','c4']])

# 3. 通过行索引获取行数据
print("通过行索引获取行数据:")
print(df.loc['s2'])

# 4. 通过多个行索引获取多个行数据
print("通过多个行索引获取多个行数据:")
print(df.loc[['s1','s3']])

# 5. 通过位置索引获取一行的值
print("通过位置索引获取一行的值:")
print(df.iloc[1])

# 6. 通过位置索引获取多行数据
print("通过位置索引获取多行数据:")
print(df.iloc[[0,2]])

# 7. 获取具体位置的值
print("获取具体位置的值:")
```

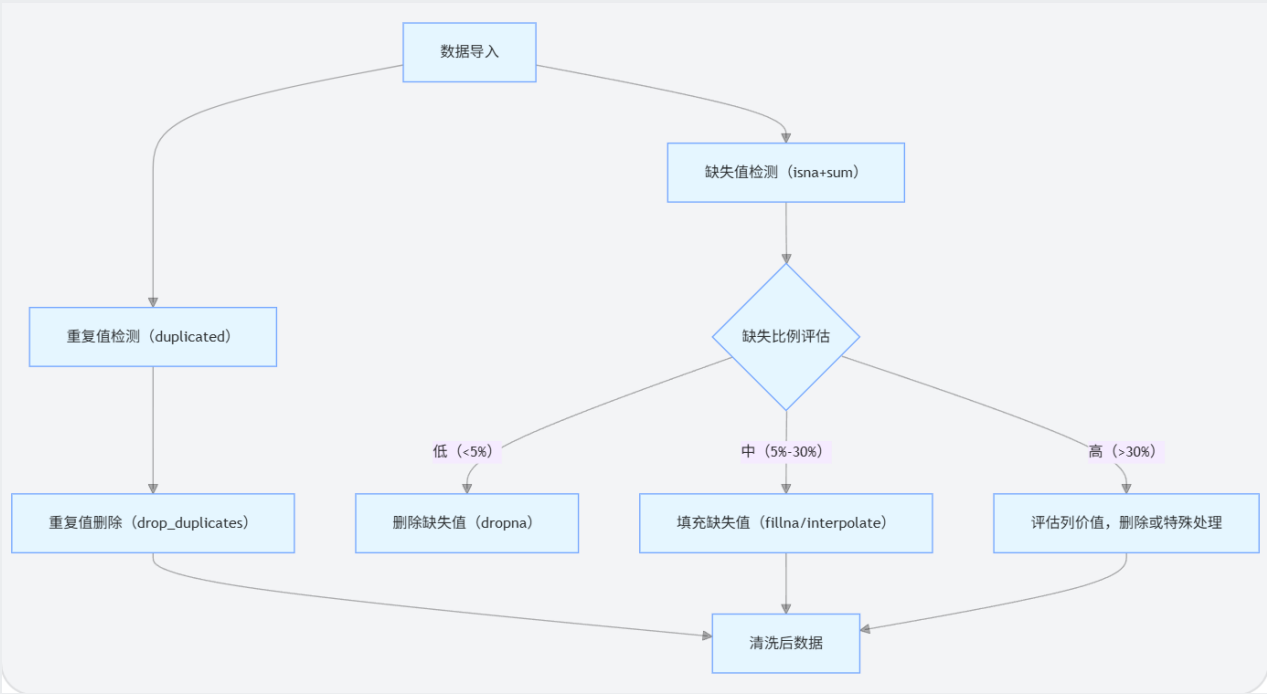
```python
print(df.iloc[1,2])           # 前面是行，后面是列
# print(df.iloc[1][2])        # FutureWarning

# 8. 通过位置索引切片获取数据
print("通过位置索引切片获取数据:")
print(df.iloc[0:2,1:3])       # 前面是行，后面是列，左闭右开

# 9. 通过名字索引切片获取数据
print("通过名字索引切片获取数据:")
print(df.loc['s1':'s3','c2':'c4'])   # 前面是行，后面是列，左闭右闭

# 10. 一次取多个不连续的行或者列
print(df.iloc[[1,2],[0,3]])   # 前面是行，后面是列
#改为loc实现上一行
print(df.loc[['s2','s3'],['c1','c4']])
```

总结

1. `df.loc[]`：根据索引名来取值，如果是切片，左闭右闭
2. `df.iloc[]`：根据索引的位置来取值，如果是切片，左闭右开
3. 在DataFrame中，`df.loc()`和`df.iloc()`默认都是取**行数据**

数据清洗

当我们采集到数据之后，数据清洗是提高数据质量的关键步骤，包括处理缺失值、重复值等。



重复值处理

重复值会影响分析结果准确性，需先检测再处理，核心函数：`duplicated()`（检测）和`drop_duplicates()`（删除）

1. 检测重复值

```
# 创建一个数据表
data = {
    "name": ["Alice", "Alice", "Charlie", "David", "Alice", "Charlie"],
    "age": [20, 22, 21, 20, 20, 21],
    "score": [85, 90, 78, 92, 85, 78],
    "gender": ["女", "男", "男", "男", "女", "男"]
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5', 's6'], columns=['name', 'age', 'score', 'gender'])
print(df)

# 1. 检测重复值
print(df.duplicated())       # 返回布尔Series，True表示重复行
print('-'*20)

# 2. 指定列检测重复值(可以指定多列)
# 参数 keep，默认值为'first'，表示只保留第一个重复行，'last'表示只保留最后一个重复行，False表示标记所有重复行
df_dup_by_col = df.duplicated(subset=['name', 'age'], keep='first')
```

```
print(df_dup_by_col)
```

2. 删除重复值

```
# 创建一个数据表
data = {
    "name": ["Alice", "Alice", "Charlie", "David", "Alice", "Charlie"],
    "age": [20, 22, 21, 20, 20, 21],
    "score": [85, 90, 78, 92, 85, 78],
    "gender": ["女", "男", "男", "男", "女", "男"]
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5', 's6'], columns=['name', 'age', 'score', 'gender'])
print("原数据:")
print(df)

# 1. 删除完全重复的行
df_drop_dup = df.drop_duplicates()
print("删除完全重复的行:")
print(df_drop_dup)

# 2. 删除部分列重复的行
df_drop_dup_by_col = df.drop_duplicates(subset=['name'], keep='first')
print("删除部分列重复的行:")
print(df_drop_dup_by_col)

# 3. 直接在原数据上删除
df.drop_duplicates(inplace=True)
print("直接在原数据上删除:")
print(df)
```

缺失值处理

缺失值（NaN，Not a Number）是数据采集常见问题（如漏填、传感器故障），处理核心是“先评估缺失程度，再选修复策略”（删除 / 填充 / 插值）。

1. 检测缺失值 `isnull()`

```
# 1. 创建一个数据表
data = {
    "name": ["Alice", "Bob", "Charlie", "David", "Eva", "Frank"],
    "age": [20, 22, 21, 20, 20, np.nan],
    "score": [85, 90, np.nan, 92, 85, np.nan],
    "gender": ["女", "男", "男", np.nan, np.nan,np.nan]
```

```
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5',
's6'], columns=['name', 'age', 'score', 'gender'])
print(df)

# 2. 检测缺失值
print(df.isnull())

# 3. 计算缺失个数
print("每一列缺失个数:")
print(df.isnull().sum())

print("每一行缺失个数:")
print(df.isnull().sum(axis=1))

# 4. 计算缺失率
print("每一列缺失率:")
print(df.isnull().sum() / len(df))

print("每一行缺失率:")
print(df.isnull().sum(axis=1) / len(df.columns))
```

2. 删除缺失值

适用于缺失值占比极低（如 <5%），且删除后不影响样本代表性的场景。

```
# 1. 创建一个数据表
data = {
    "name": ["Alice", "Bob", "Charlie", "David", "Eva", "Frank"],
    "age": [20, 22, 21, 20, 20, np.nan],
    "score": [85, 90, np.nan, 92, 85, np.nan],
    "gender": ["女", "男", "男", np.nan, np.nan,np.nan]
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5',
's6'], columns=['name', 'age', 'score', 'gender'])
print("原数据:")
print(df)

# 2. 删除含有缺失值的行或列(谨慎操作，可能会丢失大量数据)
df_drop_na = df.dropna()
print("删除含有缺失值的行:")
print(df_drop_na)

df_drop_na_by_col = df.dropna(axis=1)
print("删除含有缺失值的列:")
print(df_drop_na_by_col)

# 3. 删除 score数据缺失的行
df_drop_na_by_col = df.dropna(subset=['score'])
print("删除 score数据缺失的行:")
```

```
print(df_drop_na_by_col)
```

3. 填充缺失值

```
# 1. 创建一个数据表
data = {
    "name": ["Alice", "Bob", "Charlie", "David", "Eva", "Frank"],
    "age": [20, 22, 21, 20, 20, np.nan],
    "score": [85, 90, np.nan, 92, 85, np.nan],
    "gender": ["女", "男", "男", np.nan, np.nan,np.nan]
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5', 's6'], columns=['name', 'age', 'score', 'gender'])
print("原数据:")
print(df)

# 2. 填充全局缺失值
df_fill_na = df.fillna(0)
print("填充缺失值为0:")
print(df_fill_na)

# 3. 填充某一列的缺失值
df_fill_na_by_col = df.fillna({'gender': '男'})
print("填充【gender】缺失值为男:")
print(df_fill_na_by_col)
```

数据转换：函数应用

当内置函数无法满足需求时，需用自定义函数批量处理数据，核心工具：apply()、map()、applymap()等

apply

apply 是 Pandas 中最灵活的转换函数，**既支持 Series（一维），也支持 DataFrame（二维）**，核心作用是“按规则批量处理数据”，可处理复杂逻辑（多参数、多列联动）。

1. 处理Series数据

```
# 语法格式
Series.apply(func, args=(), **kwargs)
```

```
# 使用案例
# 创建一个学生表
data = {
```

```
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [20, 22, 21, 20],
    "score": [85, 90, 78, 92],
    "gender": ["女", "男", "男", "男"]
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4'], columns=['name', 'age', 'score', 'gender'])
print(df)

# 比如需要把年龄数据乘以2
def mul_two(x):
    return x * 2
df['age'] = df['age'].apply(mul_two)
# 再次打印一下看看变化
print(df)

# 也可以直接使用匿名函数
df['age'] = df['age'].apply(lambda x: x * 2)
print(df)
```

2. 处理DataFrame数据

```
# 语法格式
DataFrame.apply(func, axis=0, args=(), **kwargs)
```

```
# apply也可以处理DataFrame数据
# 和Series的区别在于，操作DataFrame的时候要指定轴

df = pd.DataFrame(np.arange(12).reshape(3,4), index=['s1', 's2', 's3'], columns=['c1', 'c2', 'c3', 'c4'])
print(df)

# 1. 统计每个列的数据个数
print(df.apply(lambda x: x.count(), axis=0))          # 默认轴是0，即列

# 2. 统计每个列的最大值
print(df.apply(lambda x: x.max(), axis=0))           # 默认轴是0，即列

# 3. 统计每个行的数据个数
print(df.apply(lambda x: x.count(), axis=1))         # 轴1, 是行
```

map

map 映射函数，核心作用是 “按规则逐元素转换数据”，一般用于Series

```
# 语法格式
# arg: 可传入字典，函数(匿名函数)
# na_action: 缺失值处理参数，默认为None
Series.map(arg, na_action=None)
```

```
# 使用案例
# 创建一个学生二维表
data = {
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [20, 22, 21, 20],
    "score": [85, 90, 78, 92],
    "gender": ["女", "男", "男", "男"]
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4'], columns=['name', 'age', 'score', 'gender'])
print(df)

# 案例1: 将性别数据映射为male和female
# 通过字典映射
gender_map = {
    "男": "male",
    "女": "female"
}
df['gender'] = df['gender'].map(gender_map)
print(df)

# 案例2: 将性别数据映射为1和0
# 通过函数映射
df['gender'] = df['gender'].map(lambda x: 1 if x == "male" else 0)
print(df)

# 案例3: 给所有的数据乘以2
df2 = pd.DataFrame(np.arange(12).reshape(3,4), index=['s1', 's2', 's3'], columns=['c1', 'c2', 'c3', 'c4'])
print(df2)
df2 = df2.map(lambda x: x * 2)
print(df2)
```

数据分析：统计与计算

Pandas 提供丰富的统计函数，覆盖 “描述性统计”“聚合计算” 核心需求。

重点掌握 `df.describe()`

描述性统计

```python
# 创建一个数据表
data = {
    "name": ["Alice", "Bob", "Charlie", "David", "Eva", "Frank"],
    "age": [20, 22, 21, 20, 20, 30],
    "score": [85, 90, 88, 92, 85, 72],
    "gender": ["女", "男", "男", '男', '女', '男']
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5', 's6'], columns=['name', 'age', 'score', 'gender'])
print(df)

# 1. 单列统计
print("单列统计:")
print(df['age'].describe())
print('-'*20)

# 2. 多列统计
print("多列统计:")
print(df[['age', 'score']].describe())
print('-'*20)

# 3. 所有列进行统计(只会对，也只能对数值列进行统计)
print(df.describe())

# 4. 常用统计函数
print("常用统计函数:")
print(df['age'].mean())             # 平均数
print(df['age'].median())           # 中位数
print(df['age'].min())              # 最小值
print(df['age'].max())              # 最大值
print(df['age'].std())              # 标准差
print(df['age'].sum())              # 求和

```

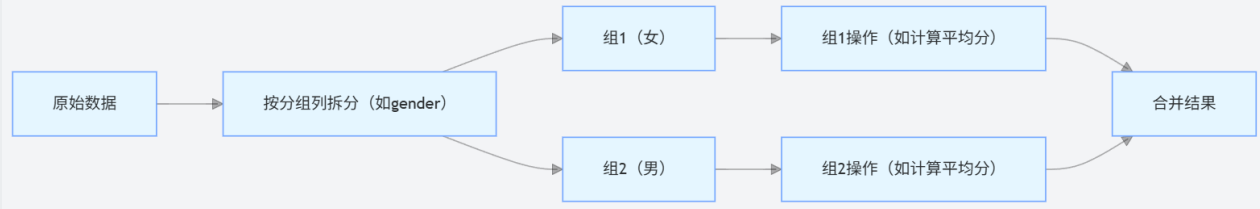
聚合计算（自己了解）

对同一列 / 多列同时应用多个统计函数，高效生成汇总结果。

```python
# 1. 单列多函数聚合
score_stats = df['score'].agg([
    "mean", "max", "min"
])
print(score_stats)

# 2. 多列分别聚合
multi_col_stats = df.agg({
    "age": ["mean", "count"],     # age列：均值、非空计数
    "score": ["median", "std"]    # score列：中位数、标准差
})
print(multi_col_stats)
```

进阶操作: 分组

分组是“按类别拆分数据→对每组应用操作→合并结果”的过程，是数据分析的核心手段。




```python
# 创建一个数据表
data = {
    "name": ["Alice", "Bob", "Charlie", "David", "Eva", "Frank"],
    "age": [20, 22, 21, 20, 20, 30],
    "score": [85, 90, 88, 92, 82, 72],
    "gender": ["女", "男", "男", '男', '女', '男']
}
df = pd.DataFrame(data, index=['s1', 's2', 's3', 's4', 's5',
's6'], columns=['name', 'age', 'score', 'gender'])
```

```python
print(df)

# 1. 分组
group_by_gender = df.groupby('gender')
print(group_by_gender)

# 2. 分组之后统计
print("分组之后统计:")
print(group_by_gender.describe())

# 3. 分组之后求均值
print("分组之后求均值:")
mean_score = df.groupby('gender')['score'].mean().add_prefix('mean_')
print(mean_score)

# 4. 分组之后进行转换
# 需求：按性别分组，求每个组的最高分(传入已有聚合函数)
df["max_score"] = df.groupby("gender")["score"].transform(max)
print(df[["name", "gender", "score", "max_score"]])

# 需求：按性别分组，计算每个学生分数在组内的降序排名(传入自定义函数)
# ascending=False 表示降序
df["score_rank"] = df.groupby("gender")["score"].transform(lambda group: group.rank(ascending=False))
print(df[["name", "gender", "score", "score_rank"]])
```