# 1. 处理缺失数据

在数据分析的工作中, 有缺失数据再正常不过了, 如何正常地处理缺失数据是数据分析的必修课。

有两种丢失数据：

None np.nan(NaN)

None是Python自带的，其类型为python object。因此，None不能参与到任何计算中。

np.nan是浮点类型，能参与到计算中。但计算的结果总是NaN。

值得庆幸的是, pandas中None与np.nan都视作np.nan.

pandas处理空值的几个方法操作:

- isnull()
- notnull()
- dropna(): 过滤丢失数据
- fillna(): 填充丢失数据

## isnull和notnull

In [1]:

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

In [2]:

```
df = DataFrame(data=np.random.randint(10,50,size=(8,8)))
df.iloc[1,3] = None
df.iloc[2,2] = None
df.iloc[4,2] = None
df.iloc[6,7] = np.nan
```

In [3]:

```
df
```

Out[3]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| 1 | 34 | 40 | 15.0 | NaN | 36 | 36 | 24 | 36.0 |
| 2 | 21 | 30 | NaN | 26.0 | 40 | 32 | 26 | 35.0 |
| 3 | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| 4 | 43 | 15 | NaN | 16.0 | 39 | 27 | 48 | 36.0 |
| 5 | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| 6 | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | NaN |
| 7 | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [4]:

```
df.isnull()
```

Out[4]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | False | False | False | False | False | False | False | False |
| 1 | False | False | False | True | False | False | False | False |
| 2 | False | False | True | False | False | False | False | False |
| 3 | False | False | False | False | False | False | False | False |
| 4 | False | False | True | False | False | False | False | False |
| 5 | False | False | False | False | False | False | False | False |
| 6 | False | False | False | False | False | False | False | True |
| 7 | False | False | False | False | False | False | False | False |

In [5]:

```
# isnull() 一般常和any() 结合使用, any() 的参数指定轴向
df.isnull().any(axis=1)
```

Out[5]:

```
0    False
1     True
2     True
3    False
4     True
5    False
6     True
7    False
dtype: bool
```

In [8]:

```
# 选出有None的行数据，在numpy中可以使用bool索引去筛选行或列，在Series中也可以通过bool索引去筛选
df.loc[df.isnull().any(axis=1)]
```

Out[8]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 34 | 40 | 15.0 | NaN | 36 | 36 | 24 | 36.0 |
| 2 | 21 | 30 | NaN | 26.0 | 40 | 32 | 26 | 35.0 |
| 4 | 43 | 15 | NaN | 16.0 | 39 | 27 | 48 | 36.0 |
| 6 | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | NaN |

In [9]:

```
# notnull() 一般和all() 结合使用， all() 的参数指定轴向
df.notnull()
```

Out[9]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | True | True | True | True | True | True | True | True |
| 1 | True | True | True | False | True | True | True | True |
| 2 | True | True | False | True | True | True | True | True |
| 3 | True | True | True | True | True | True | True | True |
| 4 | True | True | False | True | True | True | True | True |
| 5 | True | True | True | True | True | True | True | True |
| 6 | True | True | True | True | True | True | True | False |
| 7 | True | True | True | True | True | True | True | True |

In [10]:

```
df.notnull().all(axis=1)
```

Out[10]:

```
0     True
1    False
2    False
3     True
4    False
5     True
6    False
7     True
dtype: bool
```

In [12]:

```
# 选出没有None的行
df.loc[df.notnull().all(axis=1)]
```

Out[12]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

# dropna

df.dropna() 可以选择过滤的是行还是列（默认为行）：axis中0表示行，1表示的列. 之前说axis=0表示的列，这里的axis=0怎么却表示行呢？这里需要注意一点：在drop系列的方法中，和之前的轴向恰好是相反的

In [13]:

```
df
```

Out[13]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 34 | 40 | 15.0 | NaN | 36 | 36 | 24 | 36.0 |
| **2** | 21 | 30 | NaN | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 43 | 15 | NaN | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | NaN |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [14]:

```
# 删除有None的行
df.dropna(axis=0)
```

Out[14]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [15]:

```
# 删除有None的列
df.dropna(axis=1)
```

Out[15]:

| | 0 | 1 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| **0** | 17 | 33 | 46 | 14 | 20 |
| **1** | 34 | 40 | 36 | 36 | 24 |
| **2** | 21 | 30 | 40 | 32 | 26 |
| **3** | 10 | 32 | 23 | 43 | 37 |
| **4** | 43 | 15 | 39 | 27 | 48 |
| **5** | 34 | 21 | 40 | 34 | 43 |
| **6** | 36 | 17 | 29 | 13 | 40 |
| **7** | 19 | 19 | 46 | 26 | 28 |

这里既然说到了dropna, 那么也说一下drop方法.

df.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')

其中labels：single label or list-like， 配合axis使用可以删除某些行或某些列。

In [19]:

```
df
```

Out[19]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 34 | 40 | 15.0 | NaN | 36 | 36 | 24 | 36.0 |
| **2** | 21 | 30 | NaN | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 43 | 15 | NaN | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | NaN |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [20]:

```
df.drop(labels=0, axis=1)
```

Out[20]:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **0** | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 40 | 15.0 | NaN | 36 | 36 | 24 | 36.0 |
| **2** | 30 | NaN | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 15 | NaN | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 17 | 19.0 | 18.0 | 29 | 13 | 40 | NaN |
| **7** | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [21]:

```
df.drop(labels=[0, 2], axis=1)
```

Out[21]:

| | 1 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| **0** | 33 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 40 | NaN | 36 | 36 | 24 | 36.0 |
| **2** | 30 | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 32 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 15 | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 21 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 17 | 18.0 | 29 | 13 | 40 | NaN |
| **7** | 19 | 42.0 | 46 | 26 | 28 | 23.0 |

# fillna

df.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, **kwargs)

一般不会去指定value进行填充, 比如说有多处空值，如果都用同样的一个value去填充的话, 那么肯定是不合适的。一般用这个空值所在的列的上一个或下一个元素去进行填充, 因为在同一列的数据一般含义是相同的, 所以这样才是符合实际情况的。

method : {‘backfill’, ‘bfill’, ‘pad’, ‘ffill’, None}, pad/ffill：用前一个非缺失值去填充该缺失值，backfill/bfill：用下一个非缺失值填充该缺失值。所谓的ffill(向前填充)就是用前一个值进行填充。

一般使用的method：ffill(用前一个值填充) 和 bfill(用后一个值填充)

In [16]:

```
df
```

Out[16]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 34 | 40 | 15.0 | NaN | 36 | 36 | 24 | 36.0 |
| **2** | 21 | 30 | NaN | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 43 | 15 | NaN | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | NaN |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [17]:

```
# 选择axis=0 用列进行填充数据
df.fillna(method="ffill", axis=0)
```

Out[17]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 34 | 40 | 15.0 | 42.0 | 36 | 36 | 24 | 36.0 |
| **2** | 21 | 30 | 15.0 | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 43 | 15 | 29.0 | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | 28.0 |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

In [18]:

```
df.fillna(method="bfill", axis=0)
```

Out[18]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | 17 | 33 | 20.0 | 42.0 | 46 | 14 | 20 | 39.0 |
| **1** | 34 | 40 | 15.0 | 26.0 | 36 | 36 | 24 | 36.0 |
| **2** | 21 | 30 | 29.0 | 26.0 | 40 | 32 | 26 | 35.0 |
| **3** | 10 | 32 | 29.0 | 34.0 | 23 | 43 | 37 | 29.0 |
| **4** | 43 | 15 | 12.0 | 16.0 | 39 | 27 | 48 | 36.0 |
| **5** | 34 | 21 | 12.0 | 36.0 | 40 | 34 | 43 | 28.0 |
| **6** | 36 | 17 | 19.0 | 18.0 | 29 | 13 | 40 | 23.0 |
| **7** | 19 | 19 | 32.0 | 42.0 | 46 | 26 | 28 | 23.0 |

# 2. pandas的拼接操作

pandas的拼接分为两种：

级联：pd.concat, pd.append

合并：pd.merge, pd.join

# 使用pd.concat()级联

pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=None, copy=True)

需要关注的参数一般是以下几种：objs、axis、join和ignore_index。其中默认是列级联, 且使用外连接。

既然级联需要的是多个DataFrame, 那么这些DataFrame的形状可能相同可能又不同。

## 1. 匹配级联

行索引和列索引完全匹配, 这种情况下join是outer和inner的结果是完全一致的

In [1]:

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

In [2]:

```
df1 = DataFrame(data=np.random.randint(0,100,size=(3,3)),index=['a','b','c'],columns
df2 = DataFrame(data=np.random.randint(0,100,size=(3,3)),index=['a','b','c'],columns
```

In [3]:

```
# 默认使用列级联
pd.concat((df1, df2), axis=0)
```

Out[3]:

| | A | B | C |
|---|---|---|---|
| **a** | 2 | 93 | 43 |
| **b** | 90 | 79 | 89 |
| **c** | 39 | 72 | 76 |
| **a** | 60 | 50 | 13 |
| **b** | 72 | 23 | 95 |
| **c** | 40 | 7 | 90 |

In [4]:

```
pd.concat((df1, df2), join="inner")
```

Out[4]:

| | A | B | C |
|---|---|---|---|
| **a** | 2 | 93 | 43 |
| **b** | 90 | 79 | 89 |
| **c** | 39 | 72 | 76 |
| **a** | 60 | 50 | 13 |
| **b** | 72 | 23 | 95 |
| **c** | 40 | 7 | 90 |

## 2. 不匹配级联

不匹配指的是级联的维度的索引不一致。例如纵向级联时列索引不一致，横向级联时行索引不一致

有2种连接方式：

外连接：补NaN（默认模式）

内连接：只连接匹配的项，含义就是说如果指定axis=0,那么只对列索引相同的进行级联，行索引累加即可。

In [5]:

```
df1 = DataFrame(data=np.random.randint(0,100,size=(3,3)),index=['a','b','c'],columns
df2 = DataFrame(data=np.random.randint(0,100,size=(3,3)),index=['a','d','c'],columns
```

In [12]:

```
pd.concat((df1, df2))
```

```
/Users/guwanhua/venv36/lib/python3.6/site-packages/ipykernel_launcher.
py:1: FutureWarning: Sorting because non-concatenation axis is not ali
gned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=Tru
e'.

  """Entry point for launching an IPython kernel.
```

Out[12]:

| | A | B | C | d |
|---|---|---|---|---|
| **a** | 70 | 41.0 | 6 | NaN |
| **b** | 87 | 41.0 | 40 | NaN |
| **c** | 28 | 8.0 | 41 | NaN |
| **a** | 97 | NaN | 64 | 27.0 |
| **d** | 49 | NaN | 72 | 93.0 |
| **c** | 11 | NaN | 78 | 74.0 |

In [10]:

```
# 想要取上述outer之后的不为None的行和列，使用inner
pd.concat((df1, df2), join="inner")
```

Out[10]:

| | A | C |
|---|---|---|
| **a** | 70 | 6 |
| **b** | 87 | 40 |
| **c** | 28 | 41 |
| **a** | 97 | 64 |
| **d** | 49 | 72 |
| **c** | 11 | 78 |

## 使用df.append()函数添加

由于在后面级联的使用非常普遍，因此有一个函数append专门用于在后面添加.append就是添加行元素，是concat(axis=0, join="outer")的缩写形式

In [13]:

```
df1
```

Out[13]:

| | A | B | C |
|---|---|---|---|
| **a** | 70 | 41 | 6 |
| **b** | 87 | 41 | 40 |
| **c** | 28 | 8 | 41 |

In [14]:

```
df2
```

Out[14]:

| | A | d | C |
|---|---|---|---|
| **a** | 97 | 27 | 64 |
| **d** | 49 | 93 | 72 |
| **c** | 11 | 74 | 78 |

In [15]:

```
df1.append(df2)
```

```
/Users/guwanhua/venv36/lib/python3.6/site-packages/pandas/core/frame.p
y:6211: FutureWarning: Sorting because non-concatenation axis is not a
ligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=Tru
e'.

  sort=sort)
```

Out[15]:

| | A | B | C | d |
|---|---|---|---|---|
| **a** | 70 | 41.0 | 6 | NaN |
| **b** | 87 | 41.0 | 40 | NaN |
| **c** | 28 | 8.0 | 41 | NaN |
| **a** | 97 | NaN | 64 | 27.0 |
| **d** | 49 | NaN | 72 | 93.0 |
| **c** | 11 | NaN | 78 | 74.0 |

# pd.merge()合并

merge与concat的区别在于, merge需要依据某一共同的列来进行合并. merge的合并方法参数没有axis, 也就是说merge其实是用于补充列数据的一种方法。

使用pd.merge()合并时，会自动根据两者相同column名称的那一列，作为key来进行合并。

注意每一列元素的顺序不要求一致。

pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('*x*', 'y'), copy=True, indicator=False, validate=None)

需要注意的参数是left, right, how, on, left_on, right_on。

on、left_on、right_on: 一般如果两个DataFrame的要merge的列的名称相同，使用on即可。如果列名不同，就用left_on和right_on分别指明列名。

how：指的是合并(连接)的方式有inner(内连接),left(左外连接),right(右外连接),outer(全外连接);默认为inner

pandas的merge类似于msyql数据库的join, 本人之前在项目开发中通过read_sql读取数据库数据之后就经常使用merge方法。

sql中的：

```
SELECT *
FROM df1
INNER JOIN df2
  ON df1.key = df2.key;
或
SELECT *
FROM df1,df2 where df1.key=df2.key
```

对应pandas中使用 `pd.merge(df1, df2, on='key')`

In [16]:

```
df1=DataFrame([{"id":0,"name":'lxh',"age":20,"cp":'lm'},{"id":1,"name":'xiao',"age":
df2=DataFrame([{"id":100,"name":'lxh','cs':10},{"id":101,"name":'xiao','cs':40},{"id
df3=DataFrame([{"id":0,"name":'lxh','cs':10},{"id":101,"name":'xiao','cs':40},{"id":
```

In [17]:

```
df1
```

Out[17]:

| | age | cp | id | name |
|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh |
| **1** | 40 | ly | 1 | xiao |
| **2** | 4 | yry | 2 | hua |
| **3** | 70 | old | 3 | be |

In [18]:

```
df2
```

Out[18]:

| | cs | id | name |
|---|---|---|---|
| **0** | 10 | 100 | lxh |
| **1** | 40 | 101 | xiao |
| **2** | 50 | 102 | hua2 |

In [19]:

```
df3
```

Out[19]:

| | cs | id | name |
|---|---|---|---|
| **0** | 10 | 0 | lxh |
| **1** | 40 | 101 | xiao |
| **2** | 50 | 102 | hua2 |

## 内连接

内连接就是两张表互相迁就

In [20]:

```
# 单个列名做为内链接的连接键

pd.merge(df1, df2, on="name")
```

Out[20]:

| | age | cp | id_x | name | cs | id_y |
|---|---|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh | 10 | 100 |
| **1** | 40 | ly | 1 | xiao | 40 | 101 |

In [21]:

```
# 多列名做为内链接的连接键
pd.merge(df1, df3, on=("name", "id"))
```

Out[21]:

| | age | cp | id | name | cs |
|---|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh | 10 |

In [22]:

```
# 不指定on则以两个DataFrame的列名交集做为连接键,这里使用了id与name
pd.merge(df1, df3)
```

Out[22]:

| | age | cp | id | name | cs |
|---|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh | 10 |

In [23]:

```
# 使用右边的DataFrame的行索引做为连接键
# 先设置上行索引的名称
df1_index = df1.set_index("name")
```

In [24]:

```
df1
```

Out[24]:

| | age | cp | id | name |
|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh |
| **1** | 40 | ly | 1 | xiao |
| **2** | 4 | yry | 2 | hua |
| **3** | 70 | old | 3 | be |

In [25]:

```
df1_index
```

Out[25]:

| | age | cp | id |
|---|---|---|---|
| **name** | | | |
| **lxh** | 20 | lm | 0 |
| **xiao** | 40 | ly | 1 |
| **hua** | 4 | yry | 2 |
| **be** | 70 | old | 3 |

In [26]:

```
# 这里的right_index 其实功效就是 right_on
pd.merge(df1, df1_index, left_on="name", right_index=True)
```

Out[26]:

| | age_x | cp_x | id_x | name | age_y | cp_y | id_y |
|---|---|---|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh | 20 | lm | 0 |
| **1** | 40 | ly | 1 | xiao | 40 | ly | 1 |
| **2** | 4 | yry | 2 | hua | 4 | yry | 2 |
| **3** | 70 | old | 3 | be | 70 | old | 3 |

## 左右连接

左连接就是以左边为主，右连接就是以右表为主.

In [27]:

```
df1
```

Out[27]:

| | age | cp | id | name |
|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh |
| **1** | 40 | ly | 1 | xiao |
| **2** | 4 | yry | 2 | hua |
| **3** | 70 | old | 3 | be |

In [28]:

```
df2
```

Out[28]:

| | cs | id | name |
|---|---|---|---|
| **0** | 10 | 100 | lxh |
| **1** | 40 | 101 | xiao |
| **2** | 50 | 102 | hua2 |

In [30]:

```
# 左连接
pd.merge(df1, df2, on="name", how="left", suffixes=('_a','_b'))
```

Out[30]:

| | age | cp | id_a | name | cs | id_b |
|---|---|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh | 10.0 | 100.0 |
| **1** | 40 | ly | 1 | xiao | 40.0 | 101.0 |
| **2** | 4 | yry | 2 | hua | NaN | NaN |
| **3** | 70 | old | 3 | be | NaN | NaN |

In [31]:

```
# 上面是id_a, id_y 这里是id_x id_y
pd.merge(df1, df2, on="name", how="left")
```

Out[31]:

| | age | cp | id_x | name | cs | id_y |
|---|---|---|---|---|---|---|
| **0** | 20 | lm | 0 | lxh | 10.0 | 100.0 |
| **1** | 40 | ly | 1 | xiao | 40.0 | 101.0 |
| **2** | 4 | yry | 2 | hua | NaN | NaN |
| **3** | 70 | old | 3 | be | NaN | NaN |

In [32]:

```
# 右连接
pd.merge(df1, df2, how="right", on="name")
```

Out[32]:

| | age | cp | id_x | name | cs | id_y |
|---|---|---|---|---|---|---|
| **0** | 20.0 | lm | 0.0 | lxh | 10 | 100 |
| **1** | 40.0 | ly | 1.0 | xiao | 40 | 101 |
| **2** | NaN | NaN | NaN | hua2 | 50 | 102 |

## 外连接

外连接是左连接和右连接的结合, 保证两张表都能兼顾到

In [33]:

```
pd.merge(df1, df2, how="outer", on="name")
```

Out[33]:

| | age | cp | id_x | name | cs | id_y |
|---|---|---|---|---|---|---|
| **0** | 20.0 | lm | 0.0 | lxh | 10.0 | 100.0 |
| **1** | 40.0 | ly | 1.0 | xiao | 40.0 | 101.0 |
| **2** | 4.0 | yry | 2.0 | hua | NaN | NaN |
| **3** | 70.0 | old | 3.0 | be | NaN | NaN |
| **4** | NaN | NaN | NaN | hua2 | 50.0 | 102.0 |

## df.join

join方法提供了一个简便的方法用于将两个DataFrame中的不同的列索引合并成为一个DataFrame, join方法的调用者是DataFrame对象，而不是pd.

其中参数的意义与merge方法基本相同,只是join方法默认为左外连接how=left。

In [ ]:

```

```