

D17 pandas 效能調校

- 📖 簡報閱讀
- 📄 範例與作業
- 💬 問題討論
- 📝 學習心得(完成)

- >
- 重要知識點>
- 效能調校>
- 大型資料集處理>
- 知識點回顧>
- >



重要知識點



- 效能調校
- 大型資料集處理

效能調校

大家如果熟悉 pandas 了之後，邏輯都可以做出來的情況下，接下來老闆就會要求你速度要快，也就是快狠準才是 coding 的王道。

接下來有三個方法帶給大家，可以大幅減少程式的執行時間

- 讀取資料型態選最快速的
- 多使用內建函數
- 向量化的資料處理

資料型態非常多種，在大數據的情況第一關往往都是資料讀取，以下四種資料型態進行實測，可以發現讀取速度以 pickle 為最快，是平常讀 csv 的 6 倍速，當然不同環境與不同資料會有所差距，不過資料越多改善會越明顯。

文件格式	運行時間 (mean ± std)	速度倍數 (以csv為基準)
xlsx	1min 19s ± 2.82 s	無視
csv	581 ms ± 16.6 ms	1
pkl	98.4 ms ± 1.9 ms	5.90
hdf	120 ms ± 1.79 ms	4.84

- 之後如果遇到讀取資料慢時，不妨使用 pickle 檔讀取看看。

- 在使用 agg 和 transform 進行操作時，儘量使用 Python 的內建函式，能夠提高執行效率
- 下圖可看出 groupby+agg+ 內建函數是最快的
- 因為 pandas 的內建函數皆有經過加速後的算法
- 如果非得需要用到自己的函式，那盡量使用 agg 會比 transform 來的快速

```
star_time = time.time()
score_df.groupby('class').agg('mean')
end_time = time.time()
end_time - star_time

0.0032105445861816406

star_time = time.time()
score_df.groupby('class').agg(lambda x: x.mean())
end_time = time.time()
end_time - star_time

0.01680469512939453

star_time = time.time()
score_df.groupby('class').transform('mean')
end_time = time.time()
end_time - star_time

0.019934415817260742

star_time = time.time()
score_df.groupby('class').transform(lambda x: x.mean())
end_time = time.time()
end_time - star_time

0.027081966400146484
```

下圖可以看到，採用 isin() 篩選出對應資料室最快的，速度快是因為它採用了向量化的資料處理方式 (這裡的 isin() 是其中一種方式，還有其他方式，大家可以嘗試一下)

```
score_df1 = score_df.copy()
star_time = time.time()
score_df1['Pass_math'] = score_df1.math_score>=60
end_time = time.time()
end_time - star_time

0.02496051788330078

score_df2 = score_df.copy()
star_time = time.time()
score_df2['Pass_math'] = score_df2.math_score.apply(lambda x : x>=60)
end_time = time.time()
end_time - star_time

0.0016407966613769531

score_df3 = score_df.copy()
star_time = time.time()
score_df3['Pass_math'] = score_df3.math_score.isin(range(60, 100))
end_time = time.time()
end_time - star_time

0.0014753341674804688
```

大型資料集處理

- 遇到大資料集時，常有記憶體不足的問題，還有速度上變慢，此時我們可以將欄位的型態降級，不需要存太多元素在一個數字中
- 首先先生成大資料，因為改善部分不同所以分成浮點數 float 與整數 int 的資料集，可以看到不管浮點數還是整數都佔了 800128bytes

```
float_data = pd.DataFrame(np.random.uniform(0,5,100000).reshape(1000,100))
int_data = pd.DataFrame(np.random.randint(0,1000,100000).reshape(1000,100))
int_data.memory_usage(deep=True).sum(), float_data.memory_usage(deep=True).sum()
```

(800128, 800128)

將整數型態 int 改成 uint 減少記憶體正用空間，使用前 800128bytes，使用後剩下 200128bytes，原因是因為原本有 100 個欄位是 int64，經過 downcast 變成了 100 個欄位的 uint16，因此只用了 1/4 倍左右的空間(int64 uint16 差了 4 倍)

```
downcast_int = int_data.apply(pd.to_numeric, downcast='unsigned')
int_data.memory_usage(deep=True).sum(), downcast_int.memory_usage(deep=True).sum()
```

(800128, 200128)

```
compare_int = pd.concat([int_data.dtypes, downcast_int.dtypes], axis=1)
compare_int.columns = ['before', 'after']
compare_int.apply(pd.value_counts)
```

	before	after
uint16	NaN	100.0
int64	100.0	NaN

將浮點數型態 float64 改成 float32 減少記憶體正用空間，使用前 800128bytes，使用後剩下 400128bytes，原因是因為原本有 100 個欄位是 float64，經過 downcast 變成了 100 個欄位的 float32，因此只用了 1/2 倍左右的空間(float64 → float32 差了 2 倍)

```
downcast_float = float_data.apply(pd.to_numeric, downcast='float')
float_data.memory_usage(deep=True).sum(), downcast_float.memory_usage(deep=True).sum()
```

(800128, 400128)

```
compare_int = pd.concat([float_data.dtypes, downcast_float.dtypes], axis=1)
compare_int.columns = ['before', 'after']
compare_int.apply(pd.value_counts)
```

	before	after
float32	NaN	100.0
float64	100.0	NaN

知識點回顧

- 三個加速方法
 - 讀取資料型態選最快速的
 - 多使用內建函數
 - 向量化的資料處理
- 欄位的型態降級有助於減少記憶體佔用空間

參考資料

Pandas 效能優化方法

網站：[iter01.com](#)

可以看到，對同一份資料，pickle 格式的資料的讀取速度最快，是讀取 csv 格式資料的近 6 倍，其次是 hdf 格式的資料，速度最慘不忍睹的是讀取 xlsx 格式的資料（這僅僅是一份只有 15M 左右大小的資料集呀）。

所以對於日常的資料集（大多為 csv 格式），可以先用 pandas 讀入，然後將資料轉存為 pickle 或者 hdf 格式，之後每次讀取資料時候，便可以節省一些時間。程式碼如下：

```
import pandas as pd
#讀取csv
df = pd.read_csv('xxx.csv')

#pkl格式
df.to_pickle('xxx.pkl') #格式另存
df = pd.read_pickle('xxx.pkl') #讀取

#hdf格式
df.to_hdf('xxx.hdf', 'df') #格式另存
df = pd.read_hdf('xxx.pkl', 'df') #讀取
```

二、進行聚合操作時的優化

在使用 agg 和 transform 進行操作時，儘量使用 Python 的內建函式，能夠提高執行效率。（資料用的還是上面的測試用例）