zhuanlan.zhihu.com

提速百倍的Pandas性能优化方法，让你的Pandas飞起来！

13-16 minutes

Pandas是Python中用于数据处理与分析的屠龙刀，想必大家也都不陌生，但Pandas在使用上有一些技巧和需要注意的地方，尤其是对于较大的数据集而言，如果你没有适当地使用，那么可能会导致Pandas的运行速度非常慢。

对于程序猿/媛而言，时间就是生命，这篇文章给大家总结了一些pandas常见的性能优化方法，希望能对你有所帮助！

一、数据读取的优化

读取数据是进行数据分析前的一个必经环节，pandas中也内置了许多数据读取的函数，最常见的就是用`pd.read_csv()`函数从csv文件读取数据，那不同格式的文件读取起来有什么区别呢？哪种方式速度更快呢？我们做个实验对比一下。

这里采用的数据共59万行，分别保存为xlsx、csv、hdf以及pkl格式，每种格式进行10次读取测试，得到下面的结果。

文件格式	运行时间（mean ± std）	速度倍数（以csv为基准）
xlsx	1min 19s ± 2.82 s	无视
csv	581 ms ± 16.6 ms	1
pkl	98.4 ms ± 1.9 ms	5.90
hdf	120 ms ± 1.79 ms	4.84

可以看到，对同一份数据，**pkl格式**的数据的读取速度最快，是读取csv格式数据的近6倍，其次是hdf格式的数据，速度最惨不忍睹的是读取xlsx格式的数据（这仅仅是一份只有15M左右大小的数据集呀）。

所以对于日常的数据集（大多为csv格式），可以先用pandas读入，然后将数据转存为pkl或者hdf格式，之后每次读取数据时候，便可以节省一些时间。代码如下：

```
import pandas as pd
#读取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') #读取

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

二、进行聚合操作时的优化

在使用 agg 和 transform进行操作时，尽量使用Python的内置函数，能够提高运行效率。（数据用的还是上面的测试用例）

1、agg+Python内置函数

```
%%timeit
df.groupby("ProductCD")["TransactionAmt"].agg(sum)
df.groupby("ProductCD")["TransactionAmt"].agg(max)
df.groupby("ProductCD")["TransactionAmt"].agg(min)
df.groupby("ProductCD")["TransactionAmt"].agg("count")
df.groupby("ProductCD")["TransactionAmt"].agg("mean")
```

```
152 ms ± 2.09 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

2、agg+非内置函数

```
%%timeit
df.groupby("ProductCD")["TransactionAmt"].agg(lambda x:x.sum())
df.groupby("ProductCD")["TransactionAmt"].agg(lambda x:x.max())
df.groupby("ProductCD")["TransactionAmt"].agg(lambda x:x.min())
df.groupby("ProductCD")["TransactionAmt"].agg(lambda x:x.count())
df.groupby("ProductCD")["TransactionAmt"].agg(lambda x:x.mean())
```

```
242 ms ± 5.06 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

可以看到对 agg 方法，使用内置函数时运行效率提升了60%。

3、transform+Python内置函数

```
%%timeit
df.groupby("ProductCD")["TransactionAmt"].transform(sum)
df.groupby("ProductCD")["TransactionAmt"].transform(max)
df.groupby("ProductCD")["TransactionAmt"].transform(min)
df.groupby("ProductCD")["TransactionAmt"].transform("count")
df.groupby("ProductCD")["TransactionAmt"].transform("mean")
```

```
197 ms ± 5.39 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

4、transform+非内置函数

```
%%timeit
df.groupby("ProductCD")["TransactionAmt"].transform(lambda x:x.sum())
df.groupby("ProductCD")["TransactionAmt"].transform(lambda x:x.max())
df.groupby("ProductCD")["TransactionAmt"].transform(lambda x:x.min())
df.groupby("ProductCD")["TransactionAmt"].transform(lambda x:x.count())
df.groupby("ProductCD")["TransactionAmt"].transform(lambda x:x.mean())
```

```
663 ms ± 11.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```


对 transform方法而言，使用内置函数时运行效率提升了两倍。

三、对数据进行逐行操作时的优化

假设我们现在有这样一个电力消耗数据集，以及对应时段的电费价格，如下图所示：

	date_time	energy_kwh
	2001/1/13 0:00	0.586
	2001/1/13 1:00	0.58
	2001/1/13 2:00	0.572
	2001/1/13 3:00	0.596
	2001/1/13 4:00	0.592
	2001/1/13 5:00	0.592
	2001/1/13 6:00	0.596
	2001/1/13 7:00	0.239
	2001/1/13 8:00	0.566
	2001/1/13 9:00	0.557

时段	单位电费（元/kwh）	时间段
高峰期	0.75	17:00-24:00
正常期	0.68	7:00-17:00
低峰期	0.60	0:00-7:00

数据集记录着每小时的电力消耗，如第一行代表2001年1月13日零点消耗了0.586kwh的电。不同使用时段的电费价格不一样，我们现在的目的是求出总的电费，那么就需要将对应时段的单位电费×消耗电量。下面给出了三种写法，我们分别测试这三种处理方式，对比一下这三种写法有什么不同，代码效率上有什么差异。

```
#编写求得相应结果的函数
def get_cost(kwh, hour):
    if 0 <= hour < 7:
        rate = 0.6
```

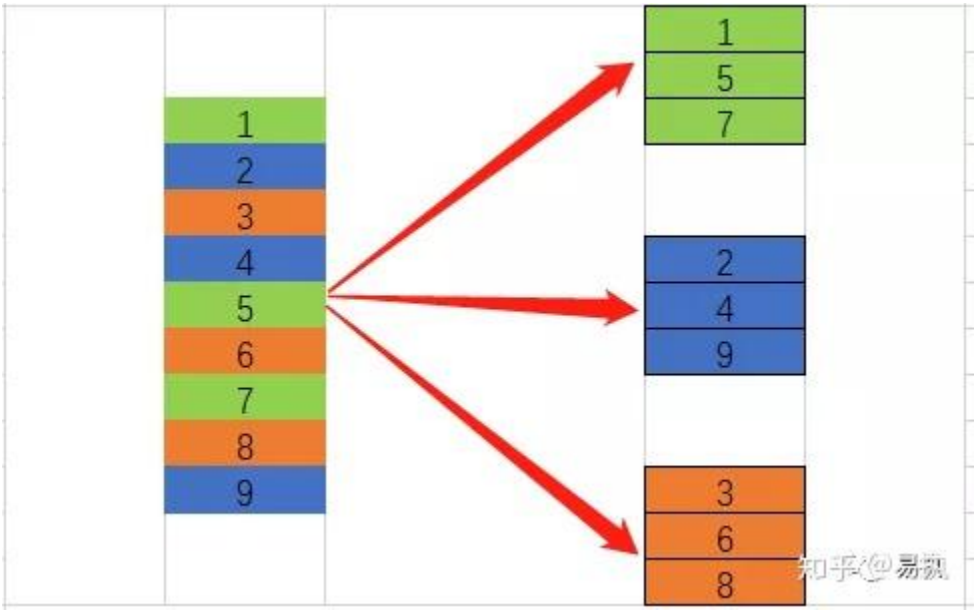
```
    elif 7 <= hour < 17:
        rate = 0.68
    elif 17 <= hour < 24:
        rate = 0.75
    else:
        raise ValueError(f'Invalid hour: {hour}')
    return rate * kwh

#方法一：简单循环
def loop(df):
    cost_list = []
    for i in range(len(df)):
        energy_used = df.iloc[i]['energy_kwh']
        hour = df.iloc[i]['date_time'].hour
        energy_cost = get_cost(energy_used, hour)
        cost_list.append(energy_cost)
    df['cost'] = cost_list

#方法二：apply方法
def apply_method(df):
     df['cost'] = df.apply(
         lambda row: get_cost(
             kwh=row['energy_kwh'],
             hour=row['date_time'].hour),
         axis=1)

#方法三：采用isin筛选出各时段，分段处理
df.set_index('date_time', inplace=True)
def isin_method(df):
    peak_hours = df.index.hour.isin(range(17, 24))
    simple_hours = df.index.hour.isin(range(7, 17))
    off_peak_hours = df.index.hour.isin(range(0, 7))

    df.loc[peak_hours, 'cost'] = df.loc[peak_hours, 'energy_kwh'] * 0.75
    df.loc[simple_hours,'cost'] = df.loc[simple_hours, 'energy_kwh'] * 0.68
    df.loc[off_peak_hours,'cost'] = df.loc[off_peak_hours, 'energy_kwh'] * 0.6
```

测试结果：

方法	运行时间（mean ± std）	速度倍数（以循环为基准）
简单循环	8.3 s ± 510 ms	1
apply	471 ms ± 13.4 ms	17.6
.isin()	13.7 ms ± 935 µs	605.8

可以看到，采用 `isin()` 筛选出对应数据后分开计算的速度是简单循环的近606倍，这并不是说 `isin()` 有多厉害，方法三速度快是因为它采用了向量化的数据处理方式（这里的`isin()` 是其中一种方式，还有其他方式，大家可以尝试一下），这才是重点。什么意思呢?

这里简单画了个图，大家可以结合这个图和代码好好体会是一个一个处理快，还是把能进行相同操作的分开然后批量处理快。



四、使用numba进行加

如果在你的数据处理过程涉及到了大量的**数值计算**，那么使用numba可以大大加快代码的运行效率，numba使用起来也很简单，下面给大家演示一下。（代码处理不具有实际意义，只是展示一下效果）

首先需要安装numba模块

我们用一个简单的例子测试一下numba的提速效果

```
import numba


@numba.vectorize
def f_with_numba(x):
```

```
    return x * 2


def f_without_numba(x):
    return x * 2


#方法一：apply逐行操作
df["double_energy"] = df.energy_kwh.apply(f_without_numba)


#方法二：向量化运行
df["double_energy"] = df.energy_kwh*2


#方法三：运用numba加速
#需要以numpy数组的形式传入
#否则会报错
df["double_energy"] = f_with_numba(df.energy_kwh.to_numpy())
```

	运行时间（mean ± std）	速度倍数（以apply为基准）
apply逐行	4.58 ms ± 188 µs	1
向量化处理	675 µs ± 25.3 µs	6785
使用numba	294 µs ± 11.6 µs	15578

从测试结果来看，再次凸显出向量化处理的优势，同时numba对原本速度已经很快的向量化处理也能提高一倍多的效率。更多numba的使用方法请参考numba的使用文档。

参考资料：

1、https://pandas.pydata.org/pandasdocs/stable/user_guide/enhancingperf.html

2、https://realpython.com/fast-flexible-pandas/

3、https://www.cnblogs.com/wkang/p/9794678.html

相关文章：

1. Pandas数据处理三板斧——map、apply、applymap详解
2. Pandas数据分析——超好用的Groupby详解
3. Pandas数据分析——Merge数据拼接图文详解
4. Pandas数据处理——玩转时间序列数据
5. Pandas数据处理——盘点那些常用的函数（上）

6. Pandas数据处理——盘点那些常用的函数（下）
7. 天秀！Pandas还能用来写爬虫？
8. 提高数据的颜值！一起看看Pandas中的那些Style

原创不易，如果觉得有点用，希望可以随手点个赞，拜谢各位老铁。