法律声明

本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

关注 小象学院

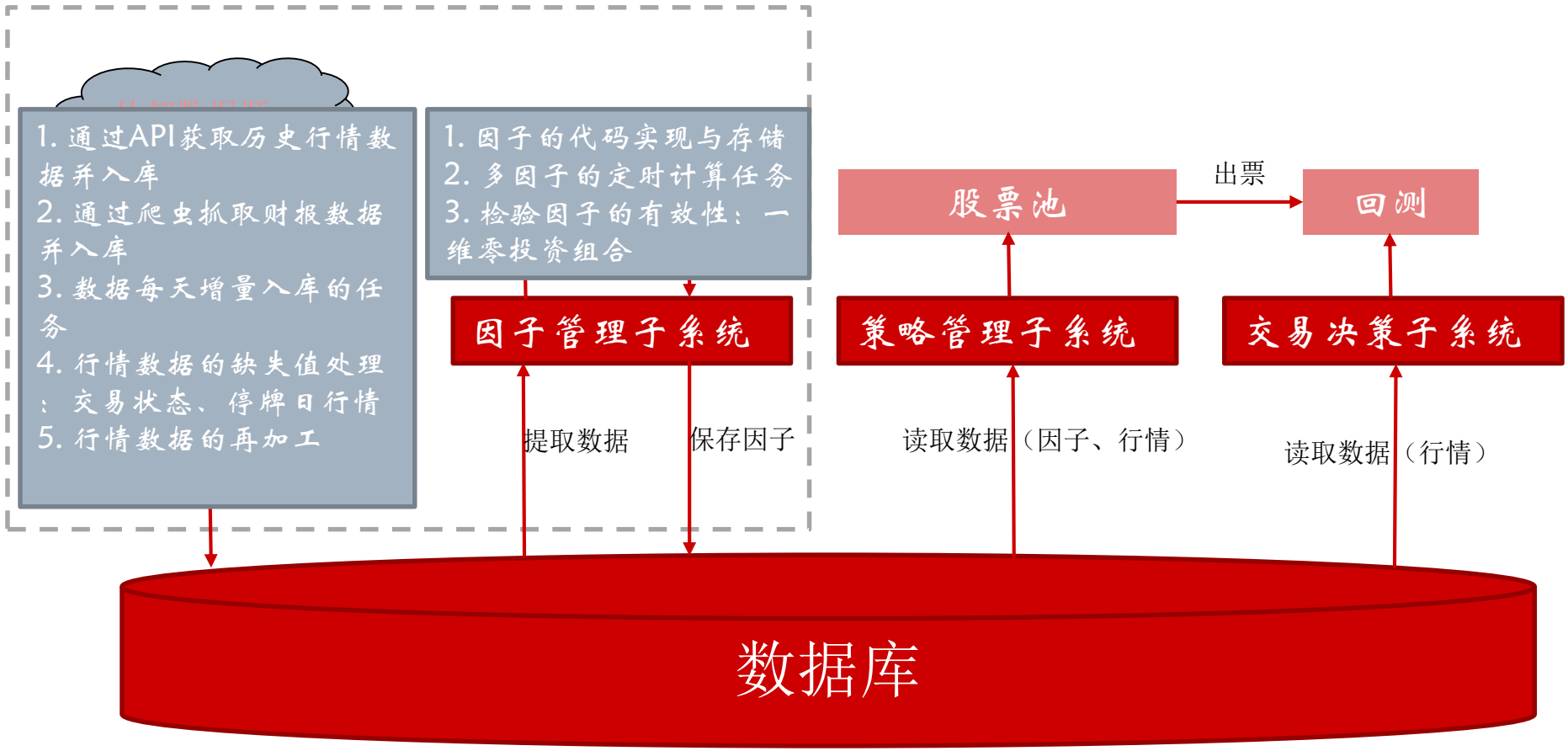
第五课

数据管理和因子管理子系统的实现

系统化构建量化交易体系：

模块2：**搭建自己的股票回测及交易平台**

模块化交易系统架构图

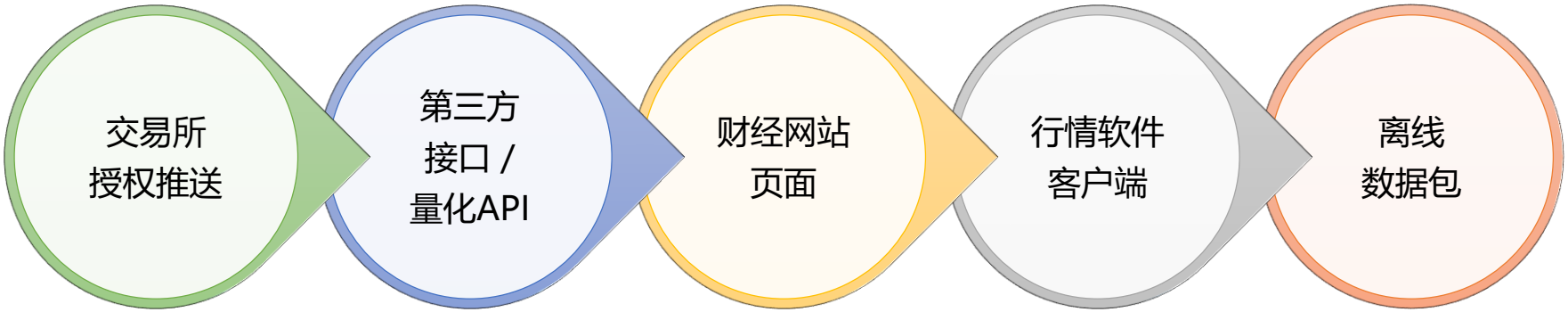

1. 通过API获取历史行情数据并入库
2. 通过爬虫抓取财报数据并入库
3. 数据每天增量入库的任务
4. 行情数据的缺失值处理：交易状态、停牌日行情
5. 行情数据的再加工
1. 因子的代码实现与存储
2. 多因子的定时计算任务
3. 检验因子的有效性：一维零投资组合
因子管理子系统
提取数据
保存因子
股票池
出票
回测
策略管理子系统
交易决策子系统
读取数据（因子、行情）
读取数据（行情）
数据库

内容介绍

- 对接开源数据源接口，获取历史行情数据
- 自动抓取股票基本数据和财报数据
- 数据异常和修正
- 数据的再加工
- 因子编写源码实例
- 因子计算任务和存储
- 因子的有效性检验：一维零投资组合

对接开源数据接口，获取历史行情

行情数据来源（例）



上海
证券交易所
深圳
证券交易所

Wind
东财Choice
掘金量化
Tushare
通联数据
JQData

新浪财经
东方财富
巨潮资讯

通达信
东方财富
同花顺
飞狐交易师
图表交易软件
券商软件

交易所
Wind
淘宝
预测者

今天的剧本

- 从Tushare获取历史行情
- 从东方财富抓取财务报表数据

任务

- 实现从TuShare获取日K数据
  - 前复权
  - 后复权
  - 不复权
- 实现盘后抓取当日K线数据
  - 交易日15：30

本地数据集

- 三个数据集
  - daily – 未复权
  - daily_qfq – 前复权
  - daily_hfq – 后复权
- 指数保存在daily中
  - index = true

code：股票代码
date：日期
index：是否为指数
open：开盘价
close：收盘价
high：最高价
low：最低价
volume：成交量

获取指数日K数据并保存

```
#  -*- coding: utf-8 -*-

from pymongo import UpdateOne
from util.database import DB_CONN
import tushare as ts
from datetime import datetime

"""
从tushare获取日K数据，保存到本地的MongoDB数据库中
"""”

class DailyCrawler:
    def __init__(self):
        self.daily = DB_CONN['daily']
        self.daily_qfq = DB_CONN['daily_qfq']
        self.daily_hfq = DB_CONN['daily_hfq']

    def crawl_index(self, begin_date=None, end_date=None):
        """
        抓取指数的日线数据，并保存到本地数据数据库中
        抓取的日期范围从2008-01-01至今
        """
        index_codes = ['000001', '000300', '399001', '399005', '399006']

        # 设置默认的日期范围
        if begin_date is None:
            begin_date = '2008-01-01'

        if end_date is None:
            end_date = datetime.now().strftime('%Y-%m-%d')

        for code in index_codes:
            df_daily = ts.get_k_data(code, index=True, start=begin_date, end=end_date)
            self.save_data(code, df_daily, self.daily, {'index': True})
```

保存到数据库

```
def save_data(self, code, df_daily, collection, extra_fields=None):
    """
    将从网上抓取的数据保存到本地MongoDB中

    :param code: 股票代码
    :param df_daily: 包含日线数据的DataFrame
    :param collection: 要保存的数据
    :param extra_fields: 除了K线数据中保存的字段，需要额外保存的字段
    """
    update_requests = []
    for df_index in df_daily.index:
        daily_obj = df_daily.loc[df_index]
        doc = self.daily_obj_2_doc(code, daily_obj)

        if extra_fields is not None:
            doc.update(extra_fields)

        update_requests.append(
            UpdateOne(
                {'code': doc['code'], 'date': doc['date']},
                {'$set': doc},
                upsert=True)
        )

    # 批量写入，提高访问效率
    if len(update_requests) > 0:
        update_result = collection.bulk_write(update_requests, ordered=False)
        print('保存日线数据，代码： %s, 插入：%4d条, 更新：%4d条' %
              (code, update_result.upserted_count, update_result.modified_count),
              flush=True)
```

获取股票日K数据

```
def crawl(self, begin_date=None, end_date=None):
    """
    获取所有股票从2008-01-01至今的K线数据（包括前复权、后复权和不复权三种），保存到数据库中
    """

    # 获取所有股票代码
    stock_df = ts.get_stock_basics()
    codes = list(stock_df.index)

    # 设置默认的日期范围
    if begin_date is None:
        begin_date = '2008-01-01'

    if end_date is None:
        end_date = datetime.now().strftime('%Y-%m-%d')

    for code in codes:
        # 抓取不复权的价格
        df_daily = ts.get_k_data(code, autype=None, start=begin_date, end=end_date)
        self.save_data(code, df_daily, self.daily, {'index': False})

        # 抓取前复权的价格
        df_daily_qfq = ts.get_k_data(code, autype='qfq', start=begin_date, end=end_date)
        self.save_data(code, df_daily_qfq, self.daily_qfq, {'index': False})

        # 抓取后复权的价格
        df_daily_hfq = ts.get_k_data(code, autype='hfq', start=begin_date, end=end_date)
        self.save_data(code, df_daily_hfq, self.daily_hfq, {'index': False})
```

数据对象转换

```
@staticmethod
def daily_obj_2_doc(code, daily_obj):
    return {
        'code': code,
        'date': daily_obj['date'],
        'close': daily_obj['close'],
        'open': daily_obj['open'],
        'high': daily_obj['high'],
        'low': daily_obj['low'],
        'volume': daily_obj['volume']
    }
```

盘后定时抓取当日K线数据

- 系统级别
  - Windows的定时任务
  - Linux的crontab
- 代码级别
  - schedule：轻量级定时任务调度框架

每天定时抓取

```
#  -*- coding: utf-8 -*-

import schedule
from data.daily_crawler import DailyCrawler
import time
from datetime import datetime

"""
每天下午15:30执行抓取，只有周一到周五才真正执行抓取任务
"""

def crawl_daily():
    dc = DailyCrawler()
    now_date = datetime.now()
    weekday = now_date.strftime('%w')
    if 0 < weekday < 6:
        now = now_date.strftime('%Y-%m-%d')
        dc.crawl_index(begin_date=now, end_date=now)
        dc.crawl(begin_date=now, end_date=now)


if __name__ == '__main__':
    schedule.every().day.at("15:30").do(crawl_daily)
    while True:
        schedule.run_pending()
        time.sleep(10)
```

自动抓取股票基本数据和财报数据

任务

- 从TuShare获取股票的基本情况数据
  - get_stock_basics
- 从东方财富抓取财报数据

基本数据的特点

- 基本数据包含的信息（例）
  - 股票代码
  - 股票名称
  - 股本（总股本、流通股本）
  - 上市日期
- 存在的变化
  - 新股上市，增加新代码和新名称
  - ST戴帽和摘帽，名称变化
  - 退市，代码和名称消失
  - 除权，股本增加
  - 市值（总市值和流通市值）变化，市值=股本 * 股价

为了留下变化的痕迹，每只股票每日保存一条记录

股本单位不一致

outstanding,流通股本(亿)

totals,总股本(亿)

```
"code" : "600000", "date" : "2016-08-12", "totals" : 2161828, "name" : "浦发银行"
"code" : "600000", "date" : "2016-08-15", "totals" : 2161828, "name" : "浦发银行"
```

▶ 股本构成

股份构成(万股)\变更日期	2017-12-31	2017-09-04	2017-06-30	2017-05-25	2017-03-20	2016-12-31	2016-06-30
股份总数	2,935,208.04	2,935,208.04	2,810,376.39	2,810,376.39	2,161,827.99	2,161,827.99	2,161,827.99
已上市流通A股	2,810,376.39	2,810,376.39	2,810,376.39	2,810,376.39	2,161,827.99	2,051,881.86	2,051,881.86
受限流通股份	124,831.65	124,831.65	0.00	0.00	0.00	109,946.14	109,946.14

```
"code" : "600000", "date" : "2017-12-26", "name" : "浦发银行", "totals" : 293.52 }
"code" : "600000", "date" : "2017-12-25", "name" : "浦发银行", "totals" : 293.52 }
"code" : "600000", "date" : "2017-12-22", "name" : "浦发银行", "totals" : 293.52 }
"code" : "600000", "date" : "2017-12-21", "name" : "浦发银行", "totals" : 293.52 }
```

```
#  -*- coding: utf-8 -*-

from pymongo import UpdateOne
from util.database import DB_CONN
from util.stock_util import get_trading_dates
import tushare as ts
from datetime import datetime, timedelta

"""
从tushare获取股票基础数据，保存到本地的MongoDB数据库中
"""


class BasicCrawler:
    def __init__(self):
        self.db = DB_CONN['basic']

    def crawl_basic(self, begin_date=None, end_date=None):
        """
        抓取指定时间范围内的股票基础信息
        :param begin_date: 开始日期
        :param end_date: 结束日期
        """

        if begin_date is None:
            begin_date = (datetime.now() - timedelta(days=1)).strftime('%Y-%m-%d')

        if end_date is None:
            end_date = (datetime.now() - timedelta(days=1)).strftime('%Y-%m-%d')

        all_dates = get_trading_dates(begin_date, end_date)

        for date in all_dates:
            try:
                self.crawl_basic_at_date(date)
            except:
                print('抓取股票进本信息时出错，日期：%s' % date, flush=True)
```

获取基本信息

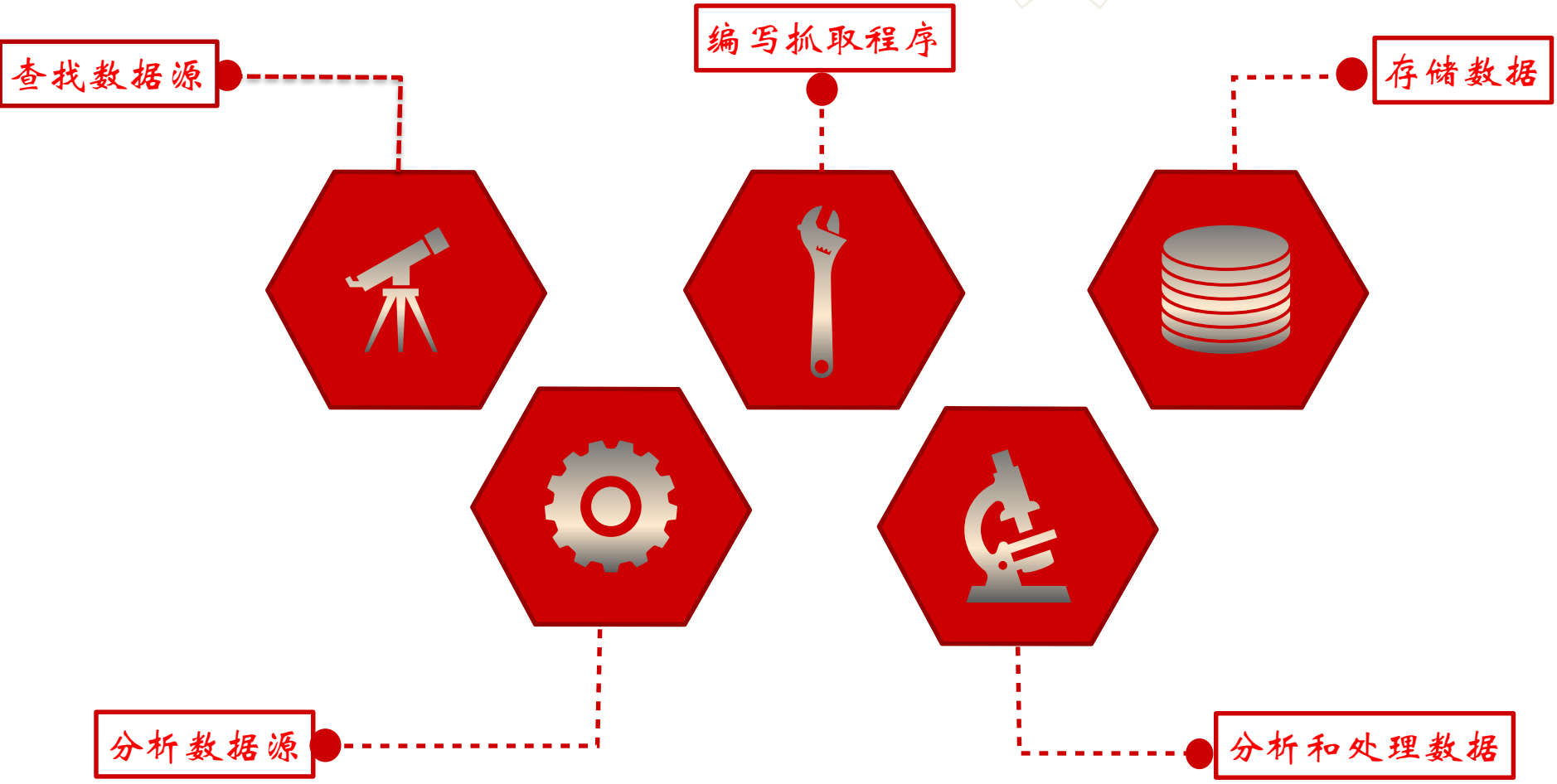
```
def crawl_basic_at_date(self, date):
    """
    从Tushare抓取指定日期的股票基本信息
    :param date: 日期
    """
    # 默认推送上一个交易日的数据
    df_basics = ts.get_stock_basics(date)

    # 如果当日没有基础信息，在不做操作
    if df_basics is None:
        return

    update_requests = []
    for code in df_basics.index:
        doc = dict(df_basics.loc[code])
        try:
            # 将20180101转换为2018-01-01的形式
            time_to_market = datetime\
                .strptime(str(doc['timeToMarket']), '%Y%m%d')\
                .strftime('%Y-%m-%d')

            # 解决流通股本和总股本单位不一致的情况
            totals = float(doc['totals'])
            # 这里假设最大规模的股本不超过5000亿，股本规模的最大工商银行是3564亿
            if totals > 5000:
                totals *= 1E4
            else:
                totals *= 1E8

            outstanding = float(doc['outstanding'])
            # 这里假设最大规模的股本不超过5000亿，股本规模的最大工商银行是3564亿
            if outstanding > 5000:
                outstanding *= 1E4
            else:
                outstanding *= 1E8
```

获取基本信息

```
        # 保存时增加date字段，因为明天都会有一条数据
        doc.update({
            'code': code,
            'date': date,
            'timeToMarket': time_to_market,
            'outstanding': outstanding,
            'totals': totals
        })

        update_requests.append(
            UpdateOne(
                {'code': code, 'date': date},
                {'$set': doc}, upsert=True))
    except:
        print('发生异常，股票代码：%s', code, flush=True)
        print(doc, flush=True)

if len(update_requests) > 0:
    update_result = self.db.bulk_write(update_requests, ordered=False)

    print('抓取股票基本信息，日期：%s，插入：%4d条，更新：%4d条' %
          (date, update_result.upserted_count, update_result.modified_count), flush=True)
```

获取基本信息

从网页抓取财报数据

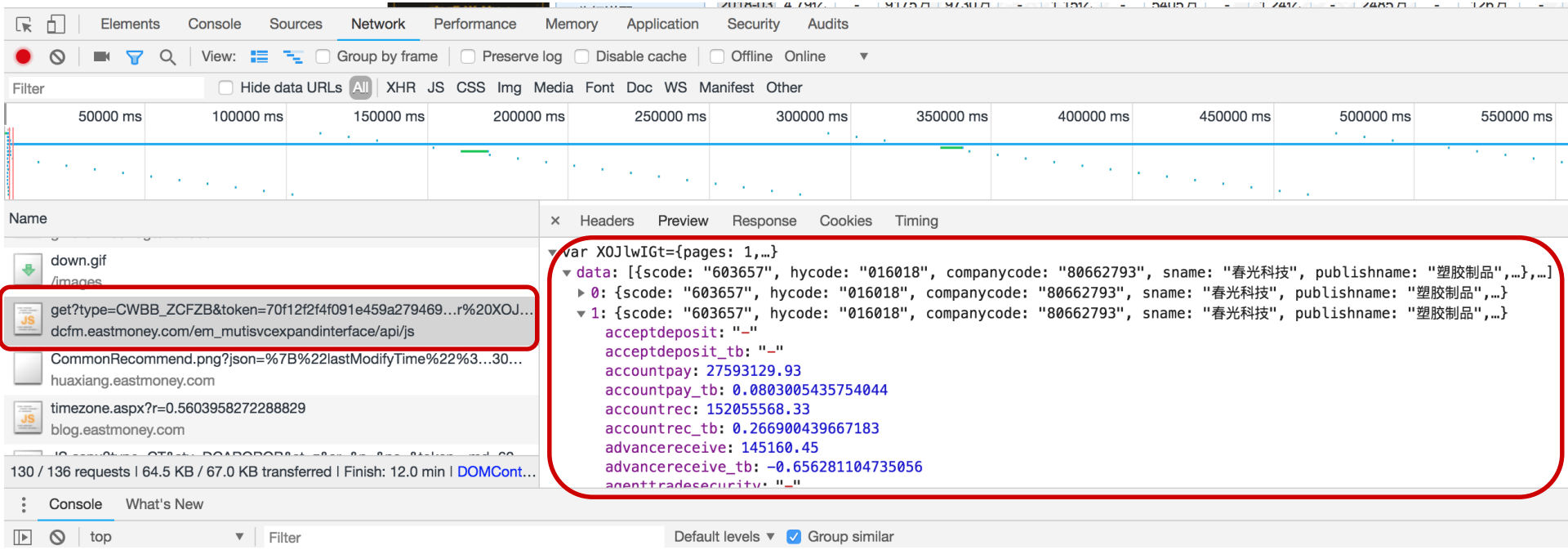


网页分析

资产负债表 | 利润表 | 现金流量表

报告期	总资产(元)	总资产同比(%)	固定资产(元)	货币资金		应收账款		存货		总负债(元)	总负债同比(%)	应付账款		预收账款		股东权益合计(元)	股东权益同比(%)	资产负债率(%)	公告日期
				货币资金(元)	同比(%)	应收账款(元)	同比(%)	存货(元)	同比(%)			应付账款(元)	同比(%)	预收账款(元)	同比(%)				
2018-03	4.79亿	-	9175万	9730万	-	1.15亿	-	5405万	-	1.24亿	-	2485万	-	126万	-	3.56亿	-	25.80	07-10
2017-12	4.76亿	22.9	9316万	7886万	10.4	1.52亿	26.7	3659万	3.12	1.32亿	-0.19	2759万	8.03	14.52万	-65.6	3.44亿	34.9	27.77	07-10
2016-12	3.88亿	-5.35	8284万	7143万	13.5	1.20亿	-4.82	3548万	21.8	1.33亿	-22.1	2554万	-2.90	42.23万	378	2.55亿	6.57	34.19	07-10
2015-12	4.09亿	30.5	6245万	6293万	20.0	1.26亿	9.48	2914万	-30.6	1.70亿	19.3	2631万	-19.7	8.84万	-85.6	2.39亿	39.8	41.56	07-10
2014-12	3.14亿	-	5404万	5245万	-	1.15亿	-	4201万	-	1.43亿	-	3277万	-	61.43万	-	1.71亿	-	45.46	07-14

获取数据抓取地址




```
http://dcfm.eastmoney.com/em_mutisvcexpandinterface/api/js/get?type=CWBB_ZCFZB&token=70f12f2f4f091e459a279469fe49eca5&st=reportdate&sr=-1&p=1&ps=50&filter=(scode=%27603657%27)&js=var%20XOJlwIGt={pages:(tp),data:%20(x)}&rt=51044855
```

实际抓取使用的网址

http://dcfm.eastmoney.com/em_mutisvcexpandinterface/api/js/get

?type=CWBB_ZCFZB 报告类型

&token=70f12f2f4f091e459a279469fe49eca5

&st=reportdate&sr=-1 排序字段和排序方式

&p=1&ps=50 分页参数

&filter=(scode=%27603657%27) 股票代码

&js={%22pages%22:(tp),%22data%22:%20(x)} 返回的结果形式

&rt=51044775#

抓取财报数据

```
#  -*- coding: utf-8 -*-

from pymongo import UpdateOne
import tushare as ts
from util.database import DB_CONN
from util.stock_util import get_all_codes
import urllib3
import json

"""
从东财获取财务数据，保存到数据库中
"""

class FinanceReportCrawler:
    def __init__(self):
        self.db = DB_CONN

    def crawl_finance_report(self):
        """
        从东方财富网站抓取三张财务报表
        :return:
        """
        # 先获取所有的股票列表
        codes = get_all_codes()

        # 创建连接池
        conn_pool = urllib3.PoolManager()

        # 抓取的网址，两个替换参数 {1} - 财报类型 {2} - 股票代码
        url = 'http://dcfm.eastmoney.com/em_mutisvcexpandinterface/api/js/get?' \
              'type={1}&token=70f12f2f4f091e459a279469fe49eca5&' \
              'st=reportdate&sr=-1&p=1&ps=500&filter=(scode=%27{2}%27)' \
              '&js={%22pages%22:(tp),%22data%22:%20(x)}&rt=51044775#'
```

抓取财报数据

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/66.0.3359.139 Safari/537.36'

# 对应的类型，分别资产负债表、现金流量表和利润表
report_types = ['CWBB_ZCFZB', 'CWBB_XJLLB', 'CWBB_LRB']

for code in codes:
    for report_type in report_types:
        print('开始抓取财报数据，股票：%s，财报类型：%s' % (code, report_type), flush=True)
        response = conn_pool.request('GET',
                                     url.replace('{1}', report_type).replace('{2}', code),
                                     headers={
                                         'User-Agent': user_agent})

        # 解析抓取结果
        result = json.loads(response.data.decode('UTF-8'), 'UTF-8')

        reports = result['data']

        update_requests = []
```

```
for report in reports:
    # 更新字段端
    try:
        report.update({
            # 公告日和报告期只保留年月日
            'announced_date': report['noticedate'][0:10],
            'report_date': report['reportdate'][0:10],
            # 股票名称和股票代码的字段名和系统设计保持一致
            'code': code,
            'name': report['sname']
        })

        update_requests.append(
            UpdateOne(
                {
                    'code': code,
                    'report_date': report['report_date'],
                    'announced_date': report['announced_date']},
                {'$set': report},
                upsert=True))
    except:
        print('解析出错，股票：%s 财报类型：%s' % (code, report_type))

if len(update_requests) > 0:
    update_result = self.db[report_type].bulk_write(update_requests, ordered=False)
    print('股票 %s, 财报类型：%s，更新： %4d, 新增： %4d' %
          (code, report_type, update_result.modified_count, update_result.upserted_count))
```

抓取财报数据

数据的异常和修正

常见的问题

- 异常值
  - 财报的公告日无法获取精确日期
  - 数字的不存在表示方法：空字符串、-、N/A
  - 数据的单位不统一：亿（万）股、亿（万）元
- 数据重复
  - 多数据源融合时，数据重复
- 缺失
  - 字段缺失
  - 记录缺失

补充停牌的日K数据

- 问题：
  - 从TuShare获取的数据，停牌日没有数据
- 影响
  - 回测时，不能直接参与账户的净值计算，导致账户的净值以及收益计算不准确
- 解决方法
  - 增加is_trading字段，区分停牌日和交易日
  - 补充停牌日的日K数据，根据当前数据现状，填充的数据为：open、close、high、low为停牌前最后一个交易日的close，volume为0，is_trading为false。

```
@staticmethod
def fill_single_date_is_trading(date, collection_name):
    """
    填充某一个日行情的数据集的is_trading
    :param date: 日期
    :param collection_name: 集合名称
    """
    print('填充字段， 字段名: is_trading，日期：%s，数据集：%s' %
          (date, collection_name), flush=True)
    daily_cursor = DB_CONN[collection_name].find(
        {'date': date},
        projection={'code': True, 'volume': True, '_id': False},
        batch_size=1000)

    update_requests = []
    for daily in daily_cursor:
        # 默认是交易
        is_trading = True
        # 如果交易量为0，则认为是停牌
        if daily['volume'] == 0:
            is_trading = False

        update_requests.append(
            UpdateOne(
                {'code': daily['code'], 'date': date},
                {'$set': {'is_trading': is_trading}}))

    if len(update_requests) > 0:
        update_result = DB_CONN[collection_name].bulk_write(update_requests, ordered=False)
        print('填充字段， 字段名: is_trading，日期：%s，数据集：%s，更新：%4d条' %
              (date, collection_name, update_result.modified_count), flush=True)
```

更新某日某个数据集的is_trading

```
def fill_is_trading_between(self, begin_date=None, end_date=None):
    """
    填充指定时间段内的is_trading字段
    :param begin_date: 开始日期
    :param end_date: 结束日期
    """
    all_dates = get_trading_dates(begin_date, end_date)

    for date in all_dates:
        self.fill_single_date_is_trading(date, 'daily')
        self.fill_single_date_is_trading(date, 'daily_hfq')
        self.fill_single_date_is_trading(date, 'daily_qfq')

def fill_is_trading(self, date=None):
    """
    为日线数据增加is_trading字段，表示是否交易的状态，True - 交易  False - 停牌
    从Tushare来的数据不包含交易状态，也不包含停牌的日K数据，为了系统中使用的方便，我们需要填充停牌是的K数据。
    一旦填充了停牌的数据，那么数据库中就同时包含了停牌和交易的数据，为了区分这两种数据，就需要增加这个字段。

    在填充该字段时，要考虑到是否最坏的情况，也就是数据库中可能已经包含了停牌和交易的数据，但是却没有is_trading
    字段。这个方法通过交易量是否为0，来判断是否停牌
    """

    if date is None:
        all_dates = get_trading_dates()
    else:
        all_dates = [date]

    for date in all_dates:
        self.fill_single_date_is_trading(date, 'daily')
        self.fill_single_date_is_trading(date, 'daily_hfq')
        self.fill_single_date_is_trading(date, 'daily_qfq')
```

批量更新is_trading

```
def fill_is_trading_between(self, begin_date=None, end_date=None):
    """
    填充指定时间段内的is_trading字段
    :param begin_date: 开始日期
    :param end_date: 结束日期
    """
    all_dates = get_trading_dates(begin_date, end_date)

    for date in all_dates:
        self.fill_single_date_is_trading(date, 'daily')
        self.fill_single_date_is_trading(date, 'daily_hfq')
        self.fill_single_date_is_trading(date, 'daily_qfq')

def fill_is_trading(self, date=None):
    """
    为日线数据增加is_trading字段，表示是否交易的状态，True - 交易  False - 停牌
    从Tushare来的数据不包含交易状态，也不包含停牌的日K数据，为了系统中使用的方便，我们需要填充停牌是的K数据。
    一旦填充了停牌的数据，那么数据库中就同时包含了停牌和交易的数据，为了区分这两种数据，就需要增加这个字段。

    在填充该字段时，要考虑到是否最坏的情况，也就是数据库中可能已经包含了停牌和交易的数据，但是却没有is_trading
    字段。这个方法通过交易量是否为0，来判断是否停牌
    """

    if date is None:
        all_dates = get_trading_dates()
    else:
        all_dates = [date]

    for date in all_dates:
        self.fill_single_date_is_trading(date, 'daily')
        self.fill_single_date_is_trading(date, 'daily_hfq')
        self.fill_single_date_is_trading(date, 'daily_qfq')
```

批量更新方法

数据的再加工处理

行情数据补充字段

- 量比
- 换手率
- 涨跌停
- MA
- …

涨跌停

- 涨停：
  - 新股：（1+44%） * ipo_price
  - ST、*ST：(1+5%) * pre_close
  - 正常：(1+10%) * pre_close
- 跌停：
  - 新股：(1-36%) * ipo_price
  - ST、*ST：(1-5%) * pre_close
  - 正常：(1-10%) * pre_close

前提

- 前收pre_close
  - 先用后复权价格计算复权因子：
    hfq_close = au_factor x close
  - 基于当日价格为基准，填充前复权的前收
    $close_{-1}$ x au_factor_{-1} = pre_close_0 x au_factor_0
- 发行价
  - Tushare：new_stocks() → price
- 某一个交易日的股票状态
  - 新股：股票基本信息中的timeToMarket
  - ST和正常：通过股票名称判断
    - Tushare可以拿到2016-08-09之后的历史数据

```
@staticmethod
def fill_au_factor_pre_close(begin_date, end_date):
    """
    为daily数据集填充：
    1. 复权因子au_factor，复权的因子计算方式：au_factor = hfq_close/close
    2. pre_close = close(-1) * au_factor(-1)/au_factor
    :param begin_date: 开始日期
    :param end_date: 结束日期
    """
    all_codes = get_all_codes()

    for code in all_codes:
        hfq_daily_cursor = DB_CONN['daily_hfq'].find(
            {'code': code, 'date': {'$lte': end_date, '$gte': begin_date}, 'index': False},
            sort=[('date', ASCENDING)],
            projection={'date': True,  'close': True})

        date_hfq_close_dict = dict([(x['date'], x['close']) for x in hfq_daily_cursor])

        daily_cursor = DB_CONN['daily'].find(
            {'code': code, 'date': {'$lte': end_date, '$gte': begin_date}, 'index': False},
            sort=[('date', ASCENDING)],
            projection={'date': True,  'close': True}
        )

        last_close = -1
        last_au_factor = -1
```

补充复权因子和前收

```
update_requests = []
for daily in daily_cursor:
    date = daily['date']
    try:
        close = daily['close']

        doc = dict()

        au_factor = round(date_hfq_close_dict[date]/close, 2)
        doc['au_factor'] = au_factor
        if last_close != -1 and last_au_factor != -1:
            pre_close = last_close * last_au_factor / au_factor
            doc['pre_close'] = round(pre_close, 2)

        last_au_factor = au_factor
        last_close = close

        update_requests.append(
            UpdateOne(
                {'code': code, 'date': date, 'index': False},
                {'$set': doc}))
    except:
        print('计算复权因子时发生错误，股票代码：%s，日期：%s' % (code, date), flush=True)
        # 恢复成初始值，防止用错
        last_close = -1
        last_au_factor = -1

if len(update_requests) > 0:
    update_result = DB_CONN['daily'].bulk_write(update_requests, ordered=False)
    print('填充复权因子和前收，股票：%s，更新：%4d条' %
          (code, update_result.modified_count), flush=True)
```

补充复权因子和前收

```
@staticmethod
def fill_high_limit_low_limit(begin_date, end_date):
    """
    为daily数据集填充涨停价和跌停价
    :param begin_date: 开始日期
    :param end_date: 结束日期
    """
    # 从tushare获取新股数据
    df_new_stocks = ts.new_stocks()

    code_ipo_price_dict = dict()
    code_ipo_date_set = set()
    for index in df_new_stocks.index:
        ipo_price = df_new_stocks.loc[index]['price']
        code = df_new_stocks.loc[index]['code']
        ipo_date = df_new_stocks.loc[index]['ipo_date']
        code_ipo_price_dict[code + '_' + ipo_date] = ipo_price
        code_ipo_date_set.add(code + '_' + ipo_date)

    all_codes = get_all_codes()

    basic_cursor = DB_CONN['basic'].find(
        {'date': {'$gte': begin_date, '$lte': end_date}},
        projection={'code': True, 'date': True, 'name': True, '_id': False},
        batch_size=1000)

    code_date_basic_dict = dict([(x['code'] + '_' + x['date'], x['name']) for x in basic_cursor])
    code_date_key_sets = set(code_date_basic_dict.keys())

    print(code_date_basic_dict)
```

计算涨停和跌停

```
for code in all_codes:
    daily_cursor = DB_CONN['daily'].find(
        {'code': code, 'date': {'$lte': end_date, '$gte': begin_date}, 'index': False},
        sort=[('date', ASCENDING)],
        projection={'date': True,  'pre_close': True}
    )

    update_requests = []
    for daily in daily_cursor:
        date = daily['date']
        code_date_key = code + '_' + daily['date']
        try:
            high_limit = -1
            low_limit = -1
            pre_close = daily['pre_close']

            if code_date_key in code_ipo_date_set:
                high_limit = round(code_ipo_price_dict[code_date_key] * 1.44, 2)
                low_limit = round(code_ipo_price_dict[code_date_key] * 0.64, 2)
            elif code_date_key in code_date_key_sets and code_date_basic_dict[code_date_key][0:2]\
                    in ['ST', '*S'] and pre_close > 0:
                high_limit = round(pre_close * 1.04, 2)
                low_limit = round(pre_close * 0.95, 2)
            elif pre_close > 0:
                high_limit = round(pre_close * 1.10, 2)
                low_limit = round(pre_close * 0.9, 2)
```

计算涨停和跌停

```
        if high_limit > 0 and low_limit > 0:
            update_requests.append(
                UpdateOne(
                    {'code': code, 'date': date, 'index': False},
                    {'$set': {'high_limit': high_limit, 'low_limit': low_limit}}))
    except:
        print('填充涨跌停时发生错误，股票代码：%s，日期：%s' % (code, date), flush=True)

if len(update_requests) > 0:
    update_result = DB_CONN['daily'].bulk_write(update_requests, ordered=False)
    print('填充涨跌停，股票：%s，更新：%4d条' %
          (code, update_result.modified_count), flush=True)
```

计算涨停和跌停

休息一下

5分钟后回来

因子源码实例编写

实现规模因子

- 总市值

  总市值 = 总股本 * 股价

- 数据来源
  - 总股本：基本信息（basic - totals）
  - 股价：不复权的日行情（daily - close）

计算规模因子 – 总市值

```
#  -*- coding: utf-8 -*-

from pymongo import UpdateOne
from base_factor import BaseFactor
from util.stock_util import get_all_codes, get_trading_dates
from data.data_module import DataModule
from datetime import datetime

“”“
实现规模因子的计算和保存
"""


class MktCapFactor(BaseFactor):
    def __init__(self):
        BaseFactor.__init__(self, name='mkt_cap')

    def compute(self, begin_date=None, end_date=None):
        """
        计算指定时间段内所有股票的该因子的值，并保存到数据库中
        :param begin_date:  开始时间
        :param end_date: 结束时间
        """
        dm = DataModule()
```

```
# 如果没有指定日期范围，则默认为计算当前交易日的数据
if begin_date is None:
    begin_date = datetime.now().strftime('%Y-%m-%d')

if end_date is None:
    end_date = datetime.now().strftime('%Y-%m-%d')

dates = get_trading_dates(begin_date, end_date)

for date in dates:
    # 查询出股票在某一交易日的总股本
    df_basics = dm.get_stock_basic_at(date)

    if df_basics.index.size == 0:
        continue

    # 将索引改为code
    df_basics.set_index(['code'], 1, inplace=True)

    # 查询出股票在某一个交易日的收盘价
    df_dailies = dm.get_one_day_k_data(autype=None, date=date)

    if df_dailies.index.size == 0:
        continue

    # 将索引设为code
    df_dailies.set_index(['code'], 1, inplace=True)

    update_requests = []
```

计算规模因子 – 总市值

```
            for code in df_dailies.index:
                try:
                    # 股价
                    close = df_dailies.loc[code]['close']
                    # 总股本
                    total_shares = df_basics.loc[code]['totals']
                    # 总市值 = 股价 * 总股本
                    total_capital = close * total_shares

                    print('%s, %s, mkt_cap: %15.2f' %
                          (code, date, total_capital),
                          flush=True)

                    update_requests.append(
                        UpdateOne(
                            {'code': code, 'date': date},
                            {'$set': {'code': code, 'date': date, self.name: total_capital}},
upsert=True))

                except:
                    print('计算规模因子时发生异常，股票代码：%s，日期：%s'
                          % (code, date),
                          flush=True)

            if len(update_requests) > 0:
                save_result = self.collection.bulk_write(update_requests, ordered=False)
                print('股票代码: %s, 因子: %s, 插入: %4d, 更新: %4d' %
                      (code, self.name, save_result.upserted_count, save_result.modified_count),
flush=True)
```

计算规模因子 – 总市值

因子的存储和计算实现

因子的存储（1）

- 使用场景（例）
  - 股票池：
    - 查询条件：日期、因子值范围
    - 结果：股票代码列表
  - 单因子或者多因子：
    - 查询条件：时间范围、因子值范围
    - 结果：股票代码和日期列表

因子的存储

- 数据读写特征
  - 写入：一次性
  - 查询：经常性
  - 更新：少量
- 存储方式
  - 每只股票每个交易日每个因子保存一条数据，即使因子值和之前相比，没有发生任何变化

因子的定时计算

- 保证每天都有数据
- 盘后运行，当日数据已固定
- 统一执行，降低复杂度

```
#  -*- coding: utf-8 -*-

from factor import *
from datetime import datetime
import schedule, time

"""
因子的计算任务，主要完成每天收盘后的因子计算任务
"""


def computing():
    weekday = datetime.now().strftime('%w')

    if weekday == 0 or weekday == 6:
        return

    # 所有的因子实例
    factors = [
        PEFactor(),
        MktCapFactor()
    ]

    now = datetime.now().strftime('%Y-%m-%d')
    for factor in factors:
        print('开始计算因子：%s，日期：%s' % (factor.name(), now), flush=True)
        factor.compute(begin_date=now, end_date=now)
        print('结束计算因子：%s，日期：%s' % (factor.name(), now), flush=True)


if __name__ == '__main__':
    # 每天下午四点定时运行
    schedule.every().day.at('16:00').do(computing)
    while True:
        schedule.run_pending()
        time.sleep(10)
        computing()
```

周一至周五的下午16:00计算因子值

因子的有效性检验：一维零投资组合

一维零投资组合的基本概念

Step1

- 选定研究因子，并使用每一时期的该因子数口对股票进行排序，对排序结果进行五（或十）分为划割

Step2

- 在第一个五分位里构造相同权重的股票组合，同时在最后一个五分位里相同权重的股票组合，第一个五分位是根据该因子排序的前1/5的股票，最后一个五分位是根据该因子的最后1/5的股票

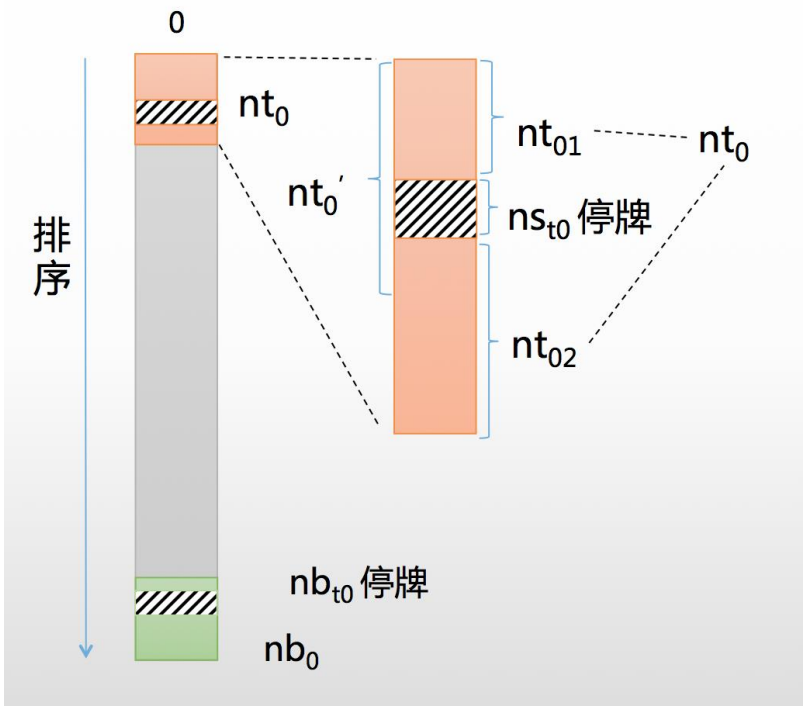
Step3

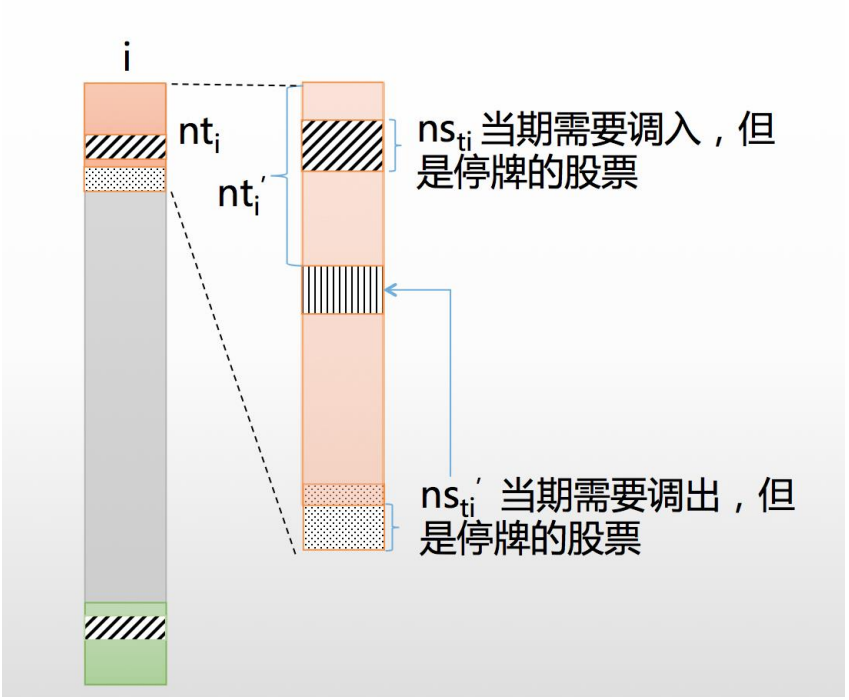
- 计算两个股票组合的月度收益，并计算两个组合收益率的差值，该差值即为零投资组合的收益。这里的零投资组合是通过买进最顶层划分中的投资组合并卖空最底层划分中的投资组合得到的，它被称为零投资组合是因为从理论上讲构造该组合不需要资本。

Step4

- 在计算过零投资组合的收益后，我们进行统计检验，以考察等权重股票组合与零投资组合收益是否存在显著不同

档位调整示意图


0
排序
nt_0
nt_0'
nt_{01}
nt_0
ns_{t0} 停牌
nt_{02}
nb_{t0} 停牌
nb_0


i
nt_i
nt_i'
ns_{ti} 当期需要调入，但是停牌的股票
ns_{ti}' 当期需要调出，但是停牌的股票

组合构建的关键点

- 计算每档数量，基数应该当日全市场的股票数
- 首期组合构建时，首末两档都不包含停牌数据
- 非首期组合构建时，保留停牌的股票

收益计算

当期首末档收益

1 $$P_i = \frac{1}{n}\sum_{m=0}^{n}\left(\frac{Close_{mi}}{Close_{m(i-1)}} - 1\right)$$

组合收益

2 $$P_{pi} = P_{ti} - P_{bi}$$

累计收益（复利）

3 $$P = \prod_{i=1}^{m}(1+P_i) - 1,\ m \in \{1,2,3,\ldots,n\}$$

只有当第n期恰好结束时，m = n

```
#  -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
import pandas as pd

from data.data_module import DataModule
from factor.factor_module import FactorModule
from util.database import DB_CONN
from util.stock_util import get_trading_dates


class ZeroInvestmentPortfolioAnalysis:
    def __init__(self, factor, begin_date, end_date, interval, position=10, ascending=True):
        """
        零投资组合的初始化方法

        :param factor:  因子名字
        :param begin_date: 分析的开始日期
        :param end_date: 分析的结束日期
        :param interval: 调整周期，交易日数
        :param position: 档位数，默认划分为10档
        :param ascending: 是否按照因子值正序排列，默认为正序
        """
        # 单期收益的DataFrame
        self.profit_df = pd.DataFrame(columns={
            'top', 'bottom', 'portfolio'})
        # 累计收益的DataFrame
        self.cumulative_profit = pd.DataFrame(columns={
            'top', 'bottom', 'portfolio'})
        # 单期股次数的DataFrame
        self.count_df = pd.DataFrame(columns={
            'top', 'bottom'})
```

初始化方法

初始化方法

```
# 净值
self.last_top_net_value = 1
self.last_bottom_net_value = 1
self.last_portfolio_net_value = 1

# 因子名字
self.factor = factor
# 因子的数据集
self.factor_collection = DB_CONN[factor]

# 分析的日期范围
self.begin_date = begin_date
self.end_date = end_date
# 调整周期
self.interval = interval
# 排序方式
self.ascending = ascending
# 档位数
self.position = position
```

首档调整方法

```
def adjust_top_position(self, top_dailies, df_factor, df_dailies, single_position_count):
    """
    调整首档组合
    1. 移除上期调入且当前非停牌股票
    2. 加入当期应跳入且当前非停牌股票
    3. 加入时，应按照排序从头逐个加入，直到满足数量要求
    :param top_dailies:
    :param df_factor: 排序后的因子值
    :param df_dailies:
    :param single_position_count:
    :return:
    """
    # 移除首档非停牌股票
    self.remove_stocks(top_dailies, df_dailies)

    # 首档股票，保留后复权的价格
    top_size = len(top_dailies.keys())
    all_codes = list(df_factor.index)
    # 所有处于交易状态的股票
    all_trading_codes = set(df_dailies.index)
    for code in all_codes:
        # 可能已经存在股票，所以需要先判断是否已经满足了数量要求
        if top_size == single_position_count:
            break

        # 只有是交易状态的股票才被纳入组合
        if code in all_trading_codes:
            top_dailies[code] = df_dailies.loc[code]['close']
            top_size += 1
```

```
def adjust_bottom_position(self, bottom_dailies, df_factor, df_dailies, single_position_count):
    """
    调整末档组合
    1. 移除上期调入且当前非停牌股票
    2. 加入当期应跳入且当前非停牌股票
    3. 加入时，应按照排序从末尾逐个加入，直到满足数量要求

    :param bottom_dailies:
    :param df_dailies: 日K数据
    :param df_factor: 排序后的因子值
    :param single_position_count: 每个档位的股票数量
    :return:
    """
    # 移除首档非停牌股票
    self.remove_stocks(bottom_dailies, df_dailies)
    # 末档股票，保留后复权的价格
    bottom_size = len(bottom_dailies.keys())
    # 将所有股票的顺序反转
    all_codes = list(df_factor.index)
    all_codes.reverse()

    # 所有处于交易状态的股票
    all_trading_codes = set(df_dailies.index)
    for code in all_codes:
        # 可能已经存在股票，所以需要先判断是否已经满足了数量要求
        if bottom_size == single_position_count:
            break

        # 只有是交易状态的股票才被纳入组合
        if code in all_trading_codes:
            bottom_dailies[code] = df_dailies.loc[code]['close']
            bottom_size += 1
```

末档调整方法

```
def compute_average_profit(self, df_dailies, position_code_close_dict):
    """
    计算某一档的平均收益
    :param df_dailies: 日行情的DataFrame
    :param position_code_close_dict:
    :return: 收益
    """
    # 提取股票列表
    position_codes = list(position_code_close_dict.keys())

    # 只有存在股票时，才进行计算
    if len(position_codes) > 0:
        # 所有股票代码
        codes = set(df_dailies.index)

        # 所有股票的累计收益
        profit_sum = 0
        # 实际参与统计的股票数
        count = 0
        # 计算所有股票的收益
        for code in position_codes:
            count += 1
            buy_close = position_code_close_dict[code]

            # 计算所有股票的累计收益
            if code in codes:
                profit_sum += (df_dailies.loc[code]['close'] - buy_close) / buy_close

        # 计算单期平均收益
        return round(profit_sum * 100 / count, 2), count

    # 没有数据时，返回None
    return None
```

计算平均收益

总结

- 数据处理子系统
  - 通过API获取历史行情，以及每天定时执行
  - 从网页抓取财报数据
  - 数据的异常处理
  - 数据的再加工
- 因子管理子系统
  - 因子的编写
  - 因子的定时计算任务
  - 因子的有效性检验

课后练习

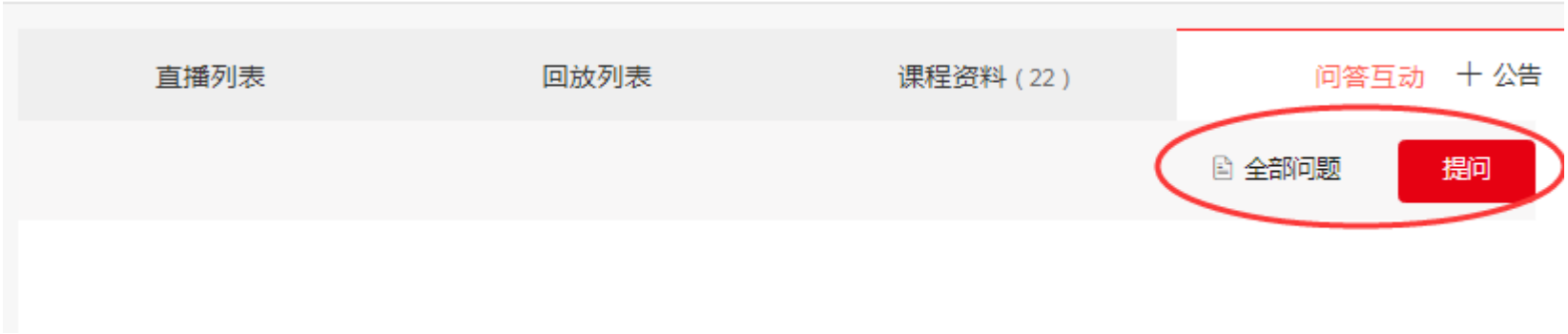
- 从基本数据或者财务报表中提取数据做一个因子：
  - 编写源码
  - 添加到定时计算任务中
  - 完成一维零投资组合检验
  - 对检验结果进行分析（需附结果图）

下节课预告

- 题目：交易决策子系统的实现—信号计算、仓位管理、风险管理
  - 编程语言和运行平台：
    Python、MongoDB

问答互动

在所报课的课程页面，
1、点击“全部问题”显示本课程所有学员提问的问题。
2、点击“提问”即可向该课程的老师和助教提问问题。



联系我们

小象学院：互联网新技术在线教育领航者

– 微信公众号：**小象学院**

THANKS