API文档

感谢您使用JoinQuant（聚宽）量化交易平台，以下内容主要介绍聚宽量化交易平台的API使用方法，目录中带有"♠"标识的API是 `"回测环境/模拟"` 的专用API，**不能在研究模块中调用。**

内容较多，可使用 `Ctrl+F` 进行搜索。

如果以下内容仍没有解决您的问题，请您通过社区提问的方式告诉我们，谢谢。

开始写策略

简单但是完整的策略

先来看一个简单但是完整的策略:

```
def initialize(context):
    g.security = '000001.XSHE'

def handle_data(context, data):
    if g.security not in context.portfolio.positions:
        order(g.security, 1000)
    else:
        order(g.security, -800)
```

一个完整策略只需要两步:

- 设置初始化函数: initialize,上面的例子中, 只操作一支股票: '000001.XSHE', 平安银行
- 实现一个函数: handle_data, 来根据历史数据调整仓位.

这个策略里, 每当我们没有股票时就买入1000股, 每当我们有股票时又卖出800股, 具体的下单API请看order函数.

这个策略里, 我们有了交易, 但是只是无意义的交易, 没有依据当前的数据做出合理的分析

下面我们来看一个真正实用的策略

实用的策略

在这个策略里, 我们会根据历史价格做出判断:

- 如果上一时间点价格高出五天平均价1%, 则全仓买入
- 如果上一时间点价格低于五天平均价, 则空仓卖出

```
# 初始化函数，设定要操作的股票、基准等等
def initialize(context):
    # 定义一个全局变量， 保存要操作的股票
    # 000001(股票:平安银行)
    g.security = '000001.XSHE'
    #设定沪深300作为基准
    set_benchmark('000300.XSHG')

# 每个单位时间(如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次)调用一次
def handle_data(context, data):
    security = g.security
    # 获取股票的收盘价
    close_data = attribute_history(security, 5, '1d', ['close'])
    # 取得过去五天的平均价格
    MA5 = close_data['close'].mean()
    # 取得上一时间点价格
    current_price = close_data['close'][-1]
    # 取得当前的现金
    cash = context.portfolio.cash

    # 如果上一时间点价格高出五天平均价1%， 则全仓买入
    if current_price > 1.01*MA5:
        # 用所有 cash 买入股票
        order_value(security, cash)
        # 记录这次买入
        log.info("Buying %s" % (security))
    # 如果上一时间点价格低于五天平均价， 则空仓卖出
    elif current_price < MA5 and context.portfolio.positions[security].closeab
le_amount > 0:
        # 卖出所有股票,使这只股票的最终持有量为0
        order_target(security, 0)
        # 记录这次卖出
        log.info("Selling %s" % (security))
    # 画出上一时间点价格
    record(stock_price=current_price)
```

用户需要实现的函数

initialize

```
initialize(context)
```

初始化方法，在整个回测、模拟实盘中最开始执行一次，用于初始一些全局变量

参数

context: Context对象, 存放有当前的账户/股票持仓信息

返回

None

示例

```
def initialize(context):
    # g为全局变量
    g.security = "000001.XSHE"
```

handle_data

```
handle_data(context, data)
```

该函数每个单位时间会调用一次, 如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次

该函数在回测中的非交易日是不会触发的（如回测结束日期为1月5日，则程序在1月1日-3日时，handle_data不会运行，4日继续运行）。

参数
context: Context对象, 存放有当前的账户/标的持仓信息
data: 一个字典(dict), key是股票代码, value是当时的SecurityUnitData 对象. 存放前一个单位时间(按天回测, 是前一天, 按分钟回测, 则是前一分钟) 的数据. **注意**:

- 为了加速, data 里面的数据是按需获取的, 每次 handle_data 被调用时, data 是空的 dict, 当你使用 `data[security]` 时该 security 的数据才会被获取.
- data 只在这一个时间点有效, 请不要存起来到下一个 handle_data 再用
- 注意, 要获取回测当天的开盘价/是否停牌/涨跌停价, 请使用 get_current_data

返回
None

示例

```
def handle_data(context, data):
    order("000001.XSHE",100)
```

before_trading_start, 可选

```
before_trading_start(context)
```

该函数会在每天开始交易前被调用一次, 您可以在这里添加一些每天都要初始化的东西.

参数
context: Context对象, 存放有当前的账户/股票持仓信息

返回
None

示例

```
def before_trading_start(context):
    log.info(str(context.current_dt))
```

after_trading_end, 可选

```
after_trading_end(context)
```

该函数会在每天结束交易后被调用一次, 您可以在这里添加一些每天收盘后要执行的内容. 这个时候所有未完成的订单已经取消.

参数
context: Context对象, 存放有当前的账户/股票持仓信息

返回
None

示例

```
def after_trading_end(context):
    log.info(str(context.current_dt))
```

process_initialize, 可选

```
process_initialize(context)
```

该函数会在每次模拟盘/回测进程重启时执行, 一般用来初始化一些**不能持久化保存**的内容. 在 initialize 后执行.

因为模拟盘会每天重启, 所以这个函数会每天都执行.

参数
context: Context对象, 存放有当前的账户/股票持仓信息

返回
None

示例

```
def process_initialize(context):
    # query 对象不能被 pickle 序列化, 所以不能持久保存, 所以每次进程重启时都给它初始化
    # 以两个下划线开始, 系统序列化 [g] 时就会自动忽略这个变量, 更多信息, 请看 [g] 和 [模拟盘注意事项]
    g.__q = query(valuation)

def handle_data(context, data):
    get_fundamentals(g.__q)
```

after_code_changed, 可选

```
after_code_changed(context)
```

模拟盘在每天的交易时间结束后会休眠，第二天开盘时会恢复，如果在恢复时发现代码已经发生了修改，则会在恢复时执行这个函数。
具体的使用场景：可以利用这个函数修改一些模拟盘的数据。

注意: 因为一些原因, 执行回测时这个函数也会被执行一次, 在 process_initialize 执行完之后执行.

参数
context: Context对象, 存放有当前的账户/股票持仓信息

返回
None

示例

```
def after_code_changed(context):
    g.stock = '000001.XSHE'
```

回测引擎介绍

回测环境

1. 回测引擎运行在Python2.7之上, 请您的策略也兼容Python2.7
2. 我们支持所有的Python标准库和部分常用第三方库, 具体请看: python库. 另外您可以把.py文件放在研究根目录, 回测中可以直接import, 具体请看: 自定义python库
3. 安全是平台的重中之重, 您的策略的运行也会受到一些限制, 具体请看: 安全

回测过程

1. 准备好您的策略, 选择要操作的股票池, 实现handle_data函数
2. 选定一个回测开始和结束日期, 选择初始资金、调仓间隔(每天还是每分钟), 开始回测
3. 引擎根据您选择的股票池和日期, 取得股票数据, 然后每一天或者每一分钟调用一次您的handle_data函数, 同时告诉您现金、持仓情况和股票在上一天或者分钟的数据. 在此函数中, 您还可以调用函数获取任何多天的历史数据, 然后做出调仓决定.
4. 当您下单后, 我们会根据接下来时间的实际交易情况, 处理您的订单. 具体细节参见订单处理
5. 下单后您可以调用get_open_orders取得所有未完成的订单, 调用cancel_order取消订单
6. 您可以在handle_data里面调用record()函数记录某些数据, 我们会以图表的方式显示在回测结果页面
7. 您可以在任何时候调用log.info/debug/warn/error函数来打印一些日志
8. 回测结束后我们会画出您的收益和基准(参见set_benchmark)收益的曲线, 列出每日持仓,每日交易和一系列风险数据。

数据

1. 股票数据：我们拥有所有A股上市公司2005年以来的股票行情数据、市值数据、财务数据、上市公司基本信息、融资融券信息等。为了避免幸存者偏差，我们包括了已经退市的股票数据。
2. 基金数据：我们目前提供了600多种在交易所上市的基金的行情、净值等数据，包含ETF、LOF、分级A/B基金以及货币基金的完整的行情、净值数据等，请点击基金数据查看。
3. 金融期货数据：我们提供中金所推出的所有金融期货产品的行情数据，并包含历史产品的数据。
4. 股票指数：我们支持近600种股票指数数据，包括指数的行情数据以及成分股数据。为了避免未来函数，我们支持获取历史任意时刻的指数成分股信息，具体见get_index_stocks。注意：指数不能买卖
5. 行业板块：我们支持按行业、按板块选股，具体见get_industry_stocks
6. 概念板块：我们支持按概念板块选股，具体见get_concept_stocks
7. 宏观数据：我们提供全方位的宏观数据，为投资者决策提供有力数据支持。
8. 所有的行情数据我们均已处理好前复权信息。
9. 我们当日的回测数据会在收盘后通过多数据源进行校验，并在T+1（第二天）的00:01更新。

安全

1. 保证您的策略安全是我们的第一要务
2. 在您使用我们网站的过程中, 我们全程使用https传输
3. 策略会加密存储在数据库
4. 回测时您的策略会在一个安全的进程中执行, 我们使用了进程隔离的方案来确保系统不会被任何用户的代码攻击, 每个用户的代码都运行在一个有很强限制的进程中:

    - 只能读指定的一些python库文件
    - 不能写和执行任何文件, 如果您需要保存和读取私有文件, 请看[write_file]/[read_file]
    - 不能创建进程或者线程
    - 限制了cpu和内存, 堆栈的使用
    - 可以访问网络, 但是对带宽做了限制, 下载最大带宽为500KB/s, 上传带宽为10KB/s
    - 有严格的超时机制, 如果handle_data超过十分钟则立即停止运行
      对于读取回测所需要的数据, 和输出回测结果, 我们使用一个辅助进程来帮它完成, 两者之间通过管道连接.

    我们使用了linux内核级别的apparmer技术来实现这一点.
    有了这些限制我们确保了任何用户不能侵入我们的系统, 更别提盗取他人的策略了.

订单处理

对于您在某个单位时间下的单, 我们会做如下处理:

- 市价单:
    - 按天回测
        - 交易价格: 开盘价 + 滑点
        - 最大成交量: 每次下单成交量不会超过该股票当天的总成交量. 可通过选项 order_volume_ratio 设置每天最大的成交量, 例如: 0.25 表示下单成交量不会超过当天成交量的 25%
        - 注意:
            - context.portfolio中的持仓价格会使用当天开盘价更新
            - data 是昨天的按天数据, 要想拿到当天开盘价, 请使用 get_current_data 拿取

day_open 字段

    - 分钟回测
        - 交易价格: 因为我们是在每分钟的第一秒钟执行代码, 所以价格是上一分钟的最后一个价格 + 滑点
        - 同按天回测规则, 每次下单成交量不会超过该股票当天的总成交量, order_volume_ratio 同样有效. 注意: 这是限制了每个订单的成交量, 当然, 你可以通过一天多次下单来超过一天成交量, 但是, 为了对你的回测负责, 请不要这么做.
    - 模拟交易
        - 交易价格: 同回测
        - 最大交易量: 不管是按天, 按分钟, 还是按tick, 由于市价单都是同步完成, 下单那一刻无法知道当天成交量, 所以市价单都不考虑成交量, 全量成交.
    - 所有市价单下单之后同步完成(也即 order_XXX 系列函数返回时完成), context.portfolio 会同步变化

- 限价单
    - 回测和模拟交易保持一致
    - 不是立即完成, 下单之后 context.portfolio.cash 和 context.portfolio.positions 不会同步变化.
    - 下单(按天, 是09:30:00下单)之后每分钟根据这一分钟的分价表撮合一次, 直到完全成交或者当天收盘为止. 撮合时会考虑 order_volume_ratio 选项, 具体撮合机制见 order_volume_ratio 选项的说明.
    - 按tick, 不是立即完成, 而是下单之后每个tick根据这个tick的分价表撮合一次, 直到完全成交或者当天收盘为止. 同样考虑 order_volume_ratio 选项.
    - 按天模拟交易暂时不支持限价单

- 上述过程中, 如果实际价格已经涨停或者跌停, 则相对应的买单或卖单不成交, 市价单直接取消(log中有警告信息), 限价单会挂单直到可以成交.
- 一天结束后, 所有未完成的订单会被取消
- 每次订单完成(完全成交)或者取消后, 我们会根据成交量计算手续费(参见set_order_cost), 减少您的现金
- 更多细节, 请看order函数

拆分合并和分红

当股票发生拆分，合并或者分红时，股票价格会受到影响，为了保证价格的连续性, 我们使用前复权来处理之前的股票价格，给您的所有股票价格已经是前复权的价格。

滑点

在实战交易中，往往最终成交价和预期价格有一定偏差，因此我们加入了滑点模式来帮助您更好地模拟真实市场的表现。

您可以通过set_slippage来设置回测具体的滑点参数。

交易税费

交易税费包含券商手续费和印花税。您可以通过set_order_cost来设置具体的交易税费的参数。

券商手续费

中国A股市场目前为双边收费，券商手续费系默认值为万分之三，即0.03%，最少5元。

印花税

印花税对卖方单边征收，对买方不再征收，系统默认为千分之一，即0.1%。

风险指标

风险指标数据有利于您对策略进行一个客观的评价。

注意: 无论是回测还是模拟, 所有风险指标(alpha/beta/sharpe/max_drawdown等指标)都只会**每天更新一次, 也只根据每天收盘后的收益计算, 并不考虑每天盘中的收益情况**. 例外:

- 分钟和TICK模拟盘每分钟会更新策略收益和基准收益
- 按天模拟盘每天开盘后和收盘后会更新策略收益和基准收益

那么可能会造成这种现象: 模拟时收益曲线中有回撤, 但是 max_drawdown 可能为0.

Total Returns（策略收益）

$$Total\ Returns = (P_{end} - P_{start})/P_{start} * 100\%$$

P_{end} = 策略最终股票和现金的总价值

P_{start} = 策略开始股票和现金的总价值

Total Annualized Returns（策略年化收益）

$$Total\ Annualized\ Returns = R_p = ((1 + P)^{\frac{250}{n}} - 1) * 100\%$$

P = 策略收益

n = 策略执行天数

Benchmark Returns（基准收益）

$$Benchmark\ Returns = (M_{end} - M_{start})/M_{start} * 100\%$$

M_{end} = 基准最终价值

M_{start} = 基准开始价值

Benchmark Annualized Returns（基准年化收益）

$$Benchmark\ Annualized\ Returns = R_m = ((1 + M)^{\frac{250}{n}} - 1) * 100\%$$

M = 基准收益

n = 策略执行天数

Alpha（阿尔法）

投资中面临着系统性风险（即Beta）和非系统性风险（即Alpha），Alpha是投资者获得与市场波动无关的回报。比如投资者获得了15%的回报，其基准获得了10%的回报，那么Alpha或者价值增值的部分就是5%。

$$Alpha = \alpha = R_p - [R_f + \beta_p(R_m - R_f)]$$

$R_p =$ 策略年化收益率

$R_m =$ 基准年化收益率

$R_f =$ 无风险利率（默认0.04）

$\beta_p =$ 策略$beta$值

Alpha值	解释
α>0	策略相对于风险，获得了超额收益
α=0	策略相对于风险，获得了适当收益
α<0	策略相对于风险，获得了较少收益

Beta（贝塔）

表示投资的系统性风险，反映了策略对大盘变化的敏感性。例如一个策略的Beta为1.5，则大盘涨1%的时候，策略可能涨1.5%，反之亦然；如果一个策略的Beta为-1.5，说明大盘涨1%的时候，策略可能跌1.5%，反之亦然。

$$Beta = \beta_p = \frac{Cov(D_p, D_m)}{Var(D_m)}$$

$D_p =$ 策略每日收益

$D_m =$ 基准每日收益

$Cov(D_p, D_m) =$ 策略每日收益与基准每日收益的协方差

$Var(D_m) =$ 基准每日收益的方差

Beta值	解释
β<0	投资组合和基准的走向通常反方向，如空头头寸类
β=0	投资组合和基准的走向没有相关性，如固定收益类
0<β<1	投资组合和基准的走向相同，但是比基准的移动幅度更小
β=1	投资组合和基准的走向相同，并且和基准的移动幅度贴近
β>1	投资组合和基准的走向相同，但是比基准的移动幅度更大

Sharpe（夏普比率）

表示每承受一单位总风险，会产生多少的超额报酬，可以同时对策略的收益与风险进行综合考虑。

$$Sharpe\ Ratio = \frac{R_p - R_f}{\sigma_p}$$

R_p = 策略年化收益率

R_f = 无风险利率（默认0.04）

σ_p = 策略收益波动率

Sortino（索提诺比率）

表示每承担一单位的下行风险，将会获得多少超额回报。

$$Sortino\ Ratio = \frac{R_p - R_f}{\sigma_{pd}}$$

R_p = 策略年化收益率

R_f = 无风险利率（默认0.04）

σ_{pd} = 策略下行波动率

Information Ratio（信息比率）

衡量单位超额风险带来的超额收益。信息比率越大，说明该策略单位跟踪误差所获得的超额收益越高，因此，信息比率较大的策略的表现要优于信息比率较低的基准。合理的投资目标应该是在承担适度风险下，尽可能追求高信息比率。

$$Information\ Ratio = \frac{R_p - R_m}{\sigma_t}$$

R_p = 策略年化收益率

R_m = 基准年化收益率

σ_t = 策略与基准每日收益差值的年化标准差

Algorithm Volatility（策略波动率）

用来测量策略的风险性，波动越大代表策略风险越高。

$$Algorithm\ Volatility = \sigma_p = \sqrt{\frac{250}{n}\sum_{i=1}^{n}(r_p - \bar{r_p})^2}$$

r_p = 策略每日收益率

$\bar{r_p}$ = 策略每日收益率的平均值 $= \frac{1}{n}\sum_{i=1}^{n} r_p$

n = 策略执行天数

Benchmark Volatility（基准波动率）

用来测量基准的风险性，波动越大代表基准风险越高。

$$Benchmark\ Volatility = \sigma_m = \sqrt{\frac{250}{n}\sum_{i=1}^{n}(r_m - \bar{r_m})^2}$$

r_m = 基准每日收益率

$$\bar{r_m} = \text{基准每日收益率的平均值} = \frac{1}{n}\sum_{i=1}^{n} r_m$$

$$n = \text{基准执行天数}$$

Max Drawdown（最大回撤）

描述策略可能出现的最糟糕的情况，最极端可能的亏损情况。

$$Max\ Drawdown = Max(P_x - P_y)/P_x$$

$$P_x, P_y = \text{策略某日股票和现金的总价值，} y > x$$

Downside Risk（下行波动率）

策略收益下行波动率。和普通收益波动率相比，下行标准差区分了好的和坏的波动。

$$Downside\ Risk = \sigma_{pd} = \sqrt{\frac{250}{m}\sum_{i=1}^{n}(r_p - \bar{r_{pi}})^2 f(t)}$$

$$r_p = \text{策略每日收益率}$$

$$\bar{r_{pi}} = \text{策略至第} i \text{日平均收益率} = \frac{1}{i}\sum_{j=1}^{i} r_j$$

$$n = \text{策略执行天数}$$

$$m = \text{策略收益低于} \bar{r_{pi}} \text{天数}$$

$$f(t) = 1\ if\ r_p < \bar{r_{pi}}$$

$$f(t) = 0\ if\ r_p >= \bar{r_{pi}}$$

胜率(%)

盈利次数在总交易次数中的占比。

$$\text{胜率} = \frac{\text{盈利交易次数}}{\text{总交易次数}}$$

日胜率(%)

盈利超过基准的日数在总交易数中的占比。

$$\text{日胜率} = \frac{\text{盈利超过基准的日数}}{\text{总交易日数}}$$

盈亏比

周期盈利亏损的比例。

$$\text{盈亏比} = \frac{\text{总盈利额}}{\text{总亏损额}}$$

运行时间

- 开盘前(9:20)运行:

    - run_monthly/run_weekly/run_daily中指定time='before_open'运行的函数
    - before_trading_start

- 盘中运行:
    - run_monthly/run_weekly/run_daily中在指定交易时间执行的函数, 执行时间为这分钟的第一秒. 例如: `run_daily(func, '14:50')` 会在每天的14:50:00(精确到秒)执行
    - handle_data
        - 按日回测/模拟, 在9:30:00(精确到秒)运行, data为昨天的天数据
        - 按分钟回测/模拟, 在每分钟的第一秒运行, 每天执行240次, 不包括11:30和15:00这两分钟, data是上一分钟的分钟数据. 例如: 当天第一次执行是在9:30:00, data是昨天14:59这一分钟的分钟数据, 当天最后一次执行是在14:59:00, data是14:58这一分钟的分钟数据.
        - 按tick模拟, 上午从9:30:00到11:29:55, 下午从13:00:00到14:59:55, 每5秒钟执行一次, 一天执行240*12次. 例如: 当天第一次执行时在9:30:00, data为昨天最后5秒(14:59:55-15:00:00)的数据(统计高开低收+成交量+成交额)

- 收盘后(15:00后半小时内)运行:
    - run_monthly/run_weekly/run_daily中指定time='after_close'运行的函数
    - after_trading_end

- 同一个时间点, 总是先运行 run_XXX 指定的函数, 然后是 before_trading_start, handle_data 和 after_trading_end
- 注意:
    - run_XXX 指定的函数只能有一个参数 context, data 不再提供, 请使用 history/attribute_history 获取
    - initialize / before_trading_start / after_trading_end / handle_data 都是可选的, 如果不是必须的, 不要实现这些函数, 一个空函数会降低运行速度.

模拟盘注意事项

- 模拟盘在每天运行结束后会保存状态, 结束进程(相当于休眠). 然后在第二天恢复.
- 进程结束时会保存这些状态:
    - 用户账户, 持仓
    - 使用 pickle 保存 g 对象. 注意
        - g 中以 '__' 开头的变量将被忽略, 不会被保存
        - g 中不能序列化的变量不会被保存, 重启后会不存在. 如果你写了如下的代码:

```
def initialize(context):
    g.query = query(valuation)
```

g 将不能被保存, 因为 query() 返回的对象并不能被持久化. 重启后也不会再执行 initialize, 使用 g.query 将会抛出 AttributeError 异常
正确的做法是, 在 process_initialize 中初始化它, 并且名字以 '__' 开头.

```
def process_initialize(context):
    g.__query = query(valuation)
```

        - 注意: 涉及到IO(打开的文件, 网络连接, 数据库连接)的对象是不能被序列化的:
            - `query(valuation)` : 数据库连接
            - `open("some/path")` : 打开的文件
            - `requests.get('https://www.joinquant.com')` : 网络连接
    - 使用 pickle 保存 context 对象, 处理方式跟 g 一样
    - 为了兼容老旧的代码, 我们会保存在函数外定义的全局变量. 但是不推荐大家直接使用全局变量.
        - 示例:

```
a = 1
b = '000001.XSHE'
def handle_data(context, data):
    g.d = 2
    context.c = 1
```

        a, b 同 g, context 一样, 都会被保存
        - 请注意, 下面的变量 xxx 不会被保存:

```
def handle_data(context, data):
    global xxx
    xxx = 1
```

        必须在 handle_data 函数外定义一下 xxx, 比如: 在 handle_data 函数前加入 `xxx = 0`
        - 建议大家不要在函数外定义全局变量, 取而代之, 使用 g 保存全局变量. 因为我们保存所有的函数外变量其实有一定风险, 比如你写了如下的代码:

```
from numpy import *
```

        我们将需要保存所有的从 numpy 导入的对象, 而其中可能有一些变量是不能 pickle.dumps 序列化的, 这会导致所有全局变量都没有持久保存
    - 为了防止恶意攻击, 序列化之后的状态大小不能超过 30M, 如果超出将在保存状态时运行失败. 当超过 20M 时日志中会有警告提示, 请注意日志.

- 恢复过程是这样的:
    1. 加载策略代码, 因为python是动态语言, 编译即运行, 所以全局的(在函数外写的)代码会被执行一遍.
    2. 使用保存的状态恢复 g, context, 和函数外定义的全局变量.
    3. 执行 process_initialize, 每次启动时都会执行这个函数.
    4. 如果策略代码和上一次运行时发生了修改，而且代码中定义了 after_code_changed 函数，则会运行 after_code_changed 函数。

- 重启后不再执行 initialize 函数, initialize 函数在整个模拟盘的生命周期中只执行一次. 即使是更改回测后, initialize 也不会执行.

- 模拟盘更改回测之后上述的全局变量(包括 g 和 context 中保存的)不会丢失. 新代码中 initialize 不会执行.
  如果需要修改原来的值, 可以在 after_code_changed 函数里面修改, 比如, 原来代码是:

  ```
  a = 1
  def initialize(context):
      g.stock = '000001.XSHE'
  ```

  代码改成:

  ```
  a = 2
  def initialize(context):
      g.stock = '000002.XSHE'
  ```

  执行时, a 仍然是 1, g.stock 仍然是 '000001.XSHE', 要修改他们的值, 必须定义 after_code_changed:

  ```
  def after_code_changed(context):
      global a
      a = 2
      g.stock = '000002.XSHE'
  ```

- 创建模拟交易时, 如果选择的日期是今天, 则从今天当前时间点开始运行, 应该在当前时间点之前运行的函数不再运行. 比如: 今天10:00创建了**按天**的模拟交易, 选择日期是今天, 代码中实现了 handle_data 和 after_trading_end, 则 handle_data 今天不运行, 而 after_trading_end 会在 15:10 运行
- 当模拟交易在A时间点失败后, 然后在B时间点"重跑", 那么 A-B 之间的时间点应该运行的函数不再运行
- 因为模拟盘资源有限, 为了防止用户创建之后长期搁置浪费资源, 我们做出如下限制: 如果一个模拟盘同时满足下面条件, 则暂缓运行:
  - 该用户连续30天没有使用JoinQuant网站
  - 没有开启微信通知

  当用户重新使用网站后, 第二天会继续运行(会把之前的交易日执行一遍, 并不会跳过日期)
- 强烈建议模拟盘使用真实价格成交, 即调用 `set_option('use_real_price', True)` . 更多细节请看 set_option

模拟交易和回测的差别

因为一些原因, 模拟交易现在和回测还是有些微小的差别, 具体如下:

- 市价单的处理:
  - 回测: 成交量不会超过当天成交量(或当天成交量的一部分, 见order_volume_ratio 选项)
  - 模拟: 全部成交

  这会导致同样的日期同样的程序回测结果可能会和模拟交易结果不一样, 请注意
- 按天模拟交易暂时不支持限价单, 所有限价单会自动转成市价单

- 理论上对运行结果不会有影响: 模拟交易进程每天会重启(请看[模拟交易注意事项]). 回测进程一般不会重启, 如果需要重启(比如机器宕机了) 也会做和模拟交易同样的处理.

期货交割日

期货持仓到交割日，没有手动交割，系统会以当天结算价平仓, 没有手续费, 不会有交易记录.

API介绍

注意事项

- 【取数据函数】【其它函数】目录中带有"♠" 标识的API是 `"回测环境/模拟"` 专用的API，**不能在研究模块中调用。**整个 【jqdata 模块】在研究环境与回测环境下都可以使用.
- 所有价格单位是元
- 时间表示:
    - 所有时间都是北京时间, 时区:UTC+8
    - 所有时间都是datetime.datetime对象
- 每日结束时自动撤销所有未完成订单
- 下文中提到 Context, SecurityUnitData, Portfolio, Position, Order 对象都是只读的, 尝试修改他们会报错或者无效.
- 没有python基础的同学请注意, 有的函数的定义中, 某些参数是有值的, 这个值是参数的默认值, 这个参数是可选的, 可以不传.

如何开始金融期货交易

初始化的仓位是**不允许**直接买卖金融期货的，因为初始默认 subportfolios[0] 中 SubPortfolioConfig 的 type = 'stock'，只允许买卖股票与基金等。

因此要买卖金融期货，您需要设定 SubPortfolioConfig 的 type = 'index_futures'，**单仓操作的设定方法**具体方法如下：

```
def initialize(context):
    # 在初始换函数 initialize 中加入下面设定语句
    set_subportfolios([SubPortfolioConfig(cash=context.portfolio.starting_cash
,type='index_futures')])
```

如需使用**分仓操作**，请看账户分仓操作.

策略设置函数

set_benchmark - 设置基准

```
set_benchmark(security)
```

默认我们选定了沪深300指数的每日价格作为判断您策略好坏和一系列风险值计算的基准. 您也可以使用set_benchmark指定其他股票/指数/ETF的价格作为基准。注意：这个函数只能在initialize中调用。

参数

- security:股票/指数/ETF代码

返回
None

示例

```
set_benchmark('600000.XSHG')
```

set_order_cost - 设置佣金/印花税

```
set_order_cost(cost, type)
```

指定每笔交易要收取的手续费, 系统会根据用户指定的费率计算每笔交易的手续费

参数

- cost: OrderCost 对象
    - open_tax，买入时印花税 (只股票类标的收取，基金与期货不收)
    - close_tax，卖出时印花税 (只股票类标的收取，基金与期货不收)
    - open_commission，买入时佣金
    - close_commission, 卖出时佣金
    - close_today_commission, 平今仓佣金
    - min_commission, 最低佣金，不包含印花税
- type: 股票、基金或金融期货，’stock’/ ‘fund’ / ‘index_futures’

默认与示例

```
# 股票类每笔交易时的手续费是：买入时佣金万分之三，卖出时万佣金分之三加千分之一印花税, 每笔交易佣金最低扣5块钱
set_order_cost(OrderCost(open_tax=0, close_tax=0.001, open_commission=0.0003, close_commission=0.0003, close_today_commission=0, min_commission=5), type='stock')

# 期货类每笔交易时的手续费是：买入时万分之0.23,卖出时万分之0.23,平今仓为万分之23
set_order_cost(OrderCost(open_tax=0, close_tax=0, open_commission=0.000023, close_commission=0.000023, close_today_commission=0.000023, min_commission=0), type='index_futures')
```

注：期货持仓到交割日会以当天结算价平仓, 没有手续费, 不会有交易记录.

~~set_commission(已废弃)~~

```
set_commission(object)
```

指定每笔交易要收取的手续费, 系统会根据用户指定的费率计算每笔交易的手续费

参数
object: 一个PerTrade对象

- PerTrade.buy_cost，买入时手续费
- PerTrade.sell_cost，卖出时手续费
- PerTrade.min_cost，最少的手续费

默认：PerTrade(buy_cost=0.0003, sell_cost=0.0013, min_cost=5)
每笔交易时的手续费是, 买入时万分之三，卖出时万分之三加千分之一印花税, 每笔交易最低扣5块钱

set_slippage - 设置滑点

```
set_slippage(object)
```

设定滑点，回测/模拟时有效.

当您下单后, 真实的成交价格与下单时预期的价格总会有一定偏差, 因此我们加入了滑点模式来帮您更好的模拟真实市场的表现. 我们暂时只支持固定滑点

固定滑点
当您使用固定滑点的时候, 我们认为您的落单的多少并不会影响您最后的成交价格. 您只需要指定一个价差,
当您下达一个买单指令的时候, 成交的价格等于当时(您执行order函数所在的单位时间)的平均价格加上价差的一半；当您下达一个卖出指令的时候，卖出的价格等于当时的平均价格减去价差的一半.
价差可以设定为一个固定值或者按照百分比设定。

- 固定值：
  这个价差可以是一个固定的值(比如0.02元, 交易时加减0.01元), 设定方式为：FixedSlippage(0.02)
- 百分比：
  这个价差可以是是当时价格的一个百分比(比如0.2%, 交易时加减当时价格的0.1%), 设定方式为：PriceRelatedSlippage(0.002)

```
# 设定滑点为固定值
set_slippage(FixedSlippage(0.02))

# 设定滑点为百分比
set_slippage(PriceRelatedSlippage(0.002))
```

注：如果您没有调用 set_slippage 函数, 系统默认的滑点是 PriceRelatedSlippage(0.00246)

set_option - 设置其他项

```
set_option(name, value)
```

设置 **真实价格、成交量比例、期货保证金比例** 选项, 其中name='use_real_price'时必须在initialize中调用，其它name没有这样的限制。

参数

- name: 选项名字, 字符串
- value: 选项的值, 根据name的不同, 是不同的类型

共有如下选项:

- **1.use_real_price(真实价格回测)**: value是True/False. 是否使用真实价格回测. 原理讲解图示见帖子。默认是False(主要是为了让旧的策略不会出错). 两者的区别是:
    - True: 回测过程中:
        - 每天看到的当天的价格都是真实的(不复权的)
        - 使用真实的价格下单, 交易详情和持仓详情里看到的都是真实价格
        - 为了让编写代码简单, 通过 history/attribute_history/get_price/SecurityUnitData.mavg/vwap 等API **拿到的都是基于当天日期的前复权价格**. 比如: 回测运行到了2015-01-01这一天, 那么 `history(3, '1d', 'close')` 取得的就是你穿越到2015-01-01这一天所看到的前复权价格. 另一方面, 你在不同日期调用 history/attribute_history/get_price/SecurityUnitData.mavg/vwap 返回的价格可能是不一样的, 因为我们在不同日期看到的前复权价格是不一样的. 所以**不要跨日期缓存这些API返回的结果.**
        - 每到新的一天, **如果持仓中有股票发生了拆合或者分红或者其他可能影响复权因子的情形, 我们会根据复权因子自动调整股票的数量, 如果调整后的数量是小数, 则向下取整到整数,** 最后为了保证context.portfolio.portfolio_value不变, context.portfolio.cash可能有略微调整.
        - 注意事项:
            - 如上所说, 不要跨日期缓存history/attribute_history/get_price这些API返回的结果
            - 开启真实价格回测之后, 回测结果可能会之前不一样, 因为交易时买入数量必须是100的倍数, 使用前复权价格和实际价格能买入的数量是不一样的.
            - 如果想通过 history 拿到昨天的真实价格, 还是需要用取得价格除以factor, 因为可能今天发生了拆合分红, 导致拿到的昨天的价格是相对于今天的前复权价格.

              ```
              s = '000001.XSHE'
              df = attribute_history(s, 1, '1d', fields=['close', 'factor'])
              real_close = df['close'][-1] / df['factor'][-1]
              ```
    - False: 此选项的核心是选定一个日期作为基准, 保证这个日期的价格是真实价格, 然后调整其他日期的价格. 最终保证所有价格是连续的, 在回测或者模拟交易过程中不同日期看到的价格是一致的. 下面分回测和模拟交易单独做介绍:
        - 回测: 基准日期是建立回测的日期, 回测过程中所看到的所有价格都是基于此日期的前复权价格. 比如说, 我昨天跑了一个回测, 那么回测过程所有价格都是在昨天所看到的前复权价格. 这会导致两个问题:
            - 回测过程中使用了前复权价格下单, 这是违背真实场景的, 不能对接实盘的.
            - 不同的日期建立的回测跑出来的结果可能会有差异, 因为如果这两次回测之间回测的股票发生了拆合或者分红, 会导致回测中看到前复权价格会不一致.

        - 模拟交易: 基准日期是建立模拟交易的日期, 模拟交易过程所看到的所有价格都是基于此日期调整过的. 为了方便计算, 我举一个虚拟的例子: 某只股票在如下三个日期的实际价格和后复权因子分别是:

日期	价格	后复权因子
2015-09-01	1	1
2015-10-01	2	2
2015-11-01	4	4

            - 如果你在 09-01 建立了一个模拟交易, 你在不同日期看到的所有价格都是 1
            - 如果你在 10-01 建立了一个模拟交易, 你在不同日期看到的所有价格都是 2
            - 如果你在 11-01 建立了一个模拟交易, 你在不同日期看到的所有价格都是 4
    - 为了更好的模拟, 建议大家都设成 True. 将来对接实盘交易时, 此选项会强制设成 True

    注意：设置 use_real_price 为 True之后, 如下的按天回测的代码是不对的：

    ```
    def initialize(context):
        g.cached_data = []
        g.s = '000001.XSHE'

    def handle_data(content, data):
        g.cached_data.append(data)
        if len(g.cached_data) > 1:
            # 如果昨天收盘价比前天涨了5%, 则买入. 这是不对的, 如果昨天早上发生了拆合, 则昨天和前天的股价不具可比性.
            if g.cached_data[-1][g.s].close > g.cached_data[-2][g.s].close * 1.05:
                order(g.s, 1000)
    ```

- 2.**order_volume_ratio(设定成交量比例)**: value 是一个 float 值, 根据实际行情限制每个订单的成交量. 无论天还是分钟回测, 对于每一笔订单
    - 如果是市价单, 成交量不超过: 每日成交量 * value
    - 如果是限价单, 限价单撮合时设定分价表中每一个价格的成交量的比率, 假设某一分钟分价表如下:

价格	成交量
10.0	10
10.1	11
10.2	12

      撮合时, 按价格 10.0 成交 10 * value 股, 按价格 10.1 成交 11 * value 股, 按价格 10.2 成交 12 * value 股
- 3.**futures_margin_rate(期货保证金比例)**: value 是一个 float 值, 设定期货的保证金比例

示例

```
# 使用真实价格回测
set_option('use_real_price', True)

# 设定成交量比例
set_option('order_volume_ratio', 0.25) # 成交量不超过总成交量的四分之一

# 设定期货保证金比例
set_option('futures_margin_rate', 0.25) # 设定期货保证金比例为25%
```

~~set_universe(已废弃)~~

```
set_universe(security_list)
```

设置或者更新此策略要操作的股票池. 请注意:

- **该函数已废弃，现在只用于设定history函数的默认security_list, 除此之外并无其他用处。**
- 当前持仓的股票仍然会在股票池里, 所以最终的股票池包括security_list和持仓股票
- 可以通过context.universe拿到当前的股票池, 如上所述, 调用set_universe之后, context.universe不一定等于security_list

参数

- security_list: 股票列表

返回
None

示例

```
set_universe(['000001.XSHE', '600000.XSHG'])
```

取数据函数

get_price - 获取历史数据

```
get_price(security, start_date=None, end_date=None, frequency='daily', field
s=None, skip_paused=False, fq='pre', count=None)
```

获取一支或者多只股票的行情数据, 按天或者按分钟，这里请在使用时注意防止未来函数

关于停牌: 因为此API可以获取多只股票的数据, 可能有的股票停牌有的没有, 为了保持时间轴的一致, 我们默认没有跳过停牌的日期, 停牌时使用停牌前的数据填充(请看 SecurityUnitData 的 paused 属性). 如想跳过, 请使用 skip_paused=True 参数, 同时只取一只股票的信息

参数

- security: 一支股票代码或者一个股票代码的list
- count: **与 start_date 二选一，不可同时使用**. 数量, 返回的结果集的行数, 即表示获取 end_date 之前几个 frequency 的数据

- start_date: **与 count 二选一，不可同时使用**. 字符串或者 datetime.datetime/datetime.date 对象, 开始时间.
    - 如果 count 和 start_date 参数都没有, 则 start_date 生效, 值是 '2015-01-01'. 注意:
    - 当取分钟数据时, 时间可以精确到分钟, 比如: 传入 `datetime.datetime(2015, 1, 1, 10, 0, 0)` 或者 `'2015-01-01 10:00:00'`.
    - 当取分钟数据时, 如果只传入日期, 则日内时间是当日的 00:00:00.
    - 当取天数据时, 传入的日内时间会被忽略

- end_date: 格式同上, 结束时间, 默认是'2015-12-31', 包含此日期. **注意: 当取分钟数据时, 如果 end_date 只有日期, 则日内时间等同于 00:00:00, 所以返回的数据是不包括 end_date 这一天的**.
- frequency: 单位时间长度, 几天或者几分钟, 现在支持'Xd','Xm', 'daily'(等同于'1d'), 'minute'(等同于'1m'), X是一个正整数, 分别表示X天和X分钟(不论是按天还是按分钟回测都能拿到这两种单位的数据), 注意, 当X > 1时, fields只支持['open', 'close', 'high', 'low', 'volume', 'money']这几个标准字段. 默认值是daily
- fields: 字符串list, 选择要获取的行情数据字段, 默认是None(表示['open', 'close', 'high', 'low', 'volume', 'money']这几个标准字段), 支持SecurityUnitData里面的所有基本属性, , 包含：['open', ' close', 'low', 'high', 'volume', 'money', 'factor', 'high_limit',' low_limit', 'avg', ' pre_close', 'paused']
- skip_paused: 是否跳过不交易日期(包括停牌, 未上市或者退市后的日期). 如果不跳过, 停牌时会使用停牌前的数据填充(具体请看SecurityUnitData的paused属性), 上市前或者退市后数据都为 nan, 但要注意:

    - 默认为 False
    - 当 skip_paused 是 True 时, 只能取一只股票的信息

- fq: 复权选项:

    - `'pre'`: 前复权(根据'use_real_price'选项不同含义会有所不同, 参见set_option), 默认是前复权
    - `None`: 不复权, 返回实际价格
    - `'post'`: 后复权

返回

- **请注意, 为了方便比较一只股票的多个属性, 同时也满足对比多只股票的一个属性的需求, 我们在security参数是一只股票和多只股票时返回的结构完全不一样**
- 如果是一支股票, 则返回pandas.DataFrame对象, 行索引是datetime.datetime对象, 列索引是行情字段名字, 比如'open'/'close'. 比如: `get_price('000300.XSHG')[:2]` 返回:

	open	close	high	low	volume	money
2015-01-05 00:00:00	3566.09	3641.54	3669.04	3551.51	451198098.0	519849817448.0
2015-01-06 00:00:00	3608.43	3641.06	3683.23	3587.23	420962185.0	498529588258.0

- 如果是多支股票, 则返回pandas.Panel对象, 里面是很多pandas.DataFrame对象, 索引是行情字段(open/close/...), 每个pandas.DataFrame的行索引是datetime.datetime对象, 列索引是股票代号. 比如 `get_price(['000300.XSHG', '000001.XSHE'])['open'][:2]` 返回:

	000300.XSHG	000001.XSHE
2015-01-05 00:00:00	3566.09	13.21

2015-01-06 00:00:00	3608.43	13.09

示例

```
# 获取一支股票
df = get_price('000001.XSHE') # 获取000001.XSHE的2015年的按天数据
df = get_price('000001.XSHE', start_date='2015-01-01', end_date='2015-01-31 23:00:00', frequency='minute', fields=['open', 'close']) # 获得000001.XSHG的2015年01月的分钟数据，只获取open+close字段
df = get_price('000001.XSHE', count = 2, end_date='2015-01-31', frequency='daily', fields=['open', 'close']) # 获取获得000001.XSHG在2015年01月31日前2个交易日的数据
df = get_price('000001.XSHE', start_date='2015-12-01 14:00:00', end_date='2015-12-02 12:00:00', frequency='1m') # 获得000001.XSHG的2015年12月1号14:00-2015年12月2日12:00的分钟数据

# 获取多只股票
panel =  get_price(get_index_stocks('000903.XSHG')) # 获取中证100的所有成分股的2015年的天数据，返回一个[pandas.Panel]
df_open = panel['open']  # 获取开盘价的[pandas.DataFrame]，  行索引是[datetime.datetime]对象，列索引是股票代号
df_volume = panel['volume']  # 获取交易量的[pandas.DataFrame]

df_open['000001.XSHE'] # 获取平安银行的2015年每天的开盘价数据
```

history ♠ - 获取历史数据

```
history(count, unit='1d', field='avg', security_list=None, df=True, skip_paused=False, fq='pre')
```

回测环境/模拟专用API

查看历史的行情数据。

关于停牌: 因为获取了多只股票的数据, 可能有的股票停牌有的没有, 为了保持时间轴的一致, 我们默认没有跳过停牌的日期, 停牌时使用停牌前的数据填充(请看SecurityUnitData的paused属性). 如想跳过, 请使用skip_paused=True 参数

当取天数据时, 不包括当天的, 即使是在收盘后

参数

- count: 数量, 返回的结果集的行数
- unit: 单位时间长度, 几天或者几分钟, 现在支持'Xd','Xm', X是一个正整数, 分别表示X天和X分钟(不论是按天还是按分钟回测都能拿到这两种单位的数据), 注意, 当X > 1时, field只支持['open', 'close', 'high', 'low', 'volume', 'money']这几个标准字段.
- field: 要获取的数据类型, 支持SecurityUnitData里面的所有基本属性, , 包含：['open', ' close', 'low', 'high', 'volume', 'money', 'factor', 'high_limit',' low_limit', 'avg', ' pre_close', 'paused']
- security_list: 要获取数据的股票列表, None表示universe中选中的所有股票
- df: 若是True, 返回pandas.DataFrame, 否则返回一个dict, 具体请看下面的返回值介绍. 默认是True. 我们之所以增加df参数, 是因为pandas.DataFrame创建和操作速度太慢, 很多情况并不需要使用它. 为了保持向上兼容, df默认是True, 但是如果你的回测速度很慢, 请考虑把df设成False.

- skip_paused: 是否跳过不交易日期(包括停牌, 未上市或者退市后的日期). 如果不跳过, 停牌时会使用停牌前的数据填充(具体请看SecurityUnitData的paused属性), 上市前或者退市后数据都为 nan, 但要注意:
    - 默认为 False
    - 如果跳过, 则行索引不再是日期, 因为不同股票的实际交易日期可能不一样
- fq: 复权选项:
    - `'pre'`: 前复权(根据'use_real_price'选项不同含义会有所不同, 参见set_option), 默认是前复权
    - `None`: 不复权, 返回实际价格
    - `'post'`: 后复权

返回

- df=True:
  pandas.DataFrame对象, 行索引是datetime.datetime对象, 列索引是股票代号. 比如: 如果当前时间是2015-01-07, universe是['000300.XSHG', '000001.XSHE'], `history(2, '1d', 'open')`将返回:

	000300.XSHG	000001.XSHE
2015-01-05 00:00:00	3566.09	13.21
2015-01-06 00:00:00	3608.43	13.09

  关于numpy和pandas, 请看下面的第三方库介绍
- df=False:
  dict, key是股票代码, 值是一个numpy数组numpy.ndarray, 对应上面的DataFrame的每一列, 例如`history(2, '1d', 'open', df=False)`将返回:

```
{
    '000300.XSHG': array([ 3566.09,  3608.43]),
    '000001.XSHE': array([ 13.21,  13.09])
}
```

示例

```
h = history(5, security_list=['000001.XSHE', '000002.XSHE'])
h['000001.XSHE'] #000001(平安银行)过去5天的每天的平均价, 一个pd.Series对象, index是datatime
h['000001.XSHE'][-1] #000001(平安银行)昨天(数组最后一项)的平均价
h.iloc[-1] #所有股票在昨天的平均价, 一个pd.Series对象, index是股票代号
h.iloc[-1]['000001.XSHE'] #000001(平安银行)昨天(数组最后一项)的平均价
h.mean() # 取得每一列的平均值
```

```
## set_universe 之后可以，调用 history 可以不用指定 security_list
set_universe(['000001.XSHE']) # 设定universe
history(5) # 获取universe中股票的过去5天(不包含今天)的每天的平均价
history(5, '1m') # 获取universe中股票的过去5分钟(不包含当前分钟)的每分钟的平均价
history(5, '1m', 'price') # 获取universe中股票的过去5分钟(不包含当前分钟)的每分钟的平均价
history(5, '1m', 'volume') # 获取universe中股票的过去5分钟(不包含当前分钟)的每分钟的交易额
history(5, '1m', 'price', ['000001.XSHE']) # 获取平安银行的过去5分钟(不包含当前分钟)的每分钟的平均价
```

```
h = history(5, security_list=['000001.XSHE', '000002.XSHE'], df=False)
h['000001.XSHE']
h['000001.XSHE'][0]
h['000001.XSHE'][-1]
h['000001.XSHE'].sum()
h['000001.XSHE'].mean()
# 因为h本身是一个dict, 下列panda.DataFrame的特性将不可用:
# h.illoc[-1]
# h.sum()
```

attribute_history ♠ - 获取历史数据

```
attribute_history(security, count, unit='1d',
            fields=['open', 'close', 'high', 'low', 'volume', 'money'],
            skip_paused=True, df=True, fq='pre')
```

回测环境/模拟专用API

查看某一支股票的历史数据, 可以选这只股票的多个属性, **默认跳过停牌日期**.

当取天数据时, 不包括当天的, 即使是在收盘后

参数

- security: 股票代码
- count: 数量, 返回的结果集的行数
- unit: 单位时间长度, 几天或者几分钟, 现在支持 'Xd', 'Xm', X是一个正整数, 分别表示X天和X分钟(不论是按天还是按分钟回测都能拿到这两种单位的数据), 注意, 当 X > 1 时, field 只支持 ['open', 'close', 'high', 'low', 'volume', 'money'] 这几个标准字段.
- fields: 股票属性的list, 支持SecurityUnitData里面的所有基本属性，包含：['open', ' close', 'low', 'high', 'volume', 'money', 'factor', 'high_limit',' low_limit', 'avg', ' pre_close', 'paused']
- skip_paused: 是否跳过不交易日期(包括停牌, 未上市或者退市后的日期). 如果不跳过, 停牌时会使用停牌前的数据填充(具体请看SecurityUnitData的paused属性), 上市前或者退市后数据都为 nan, **默认是True**
- df: 若是True, 返回pandas.DataFrame, 否则返回一个dict, 具体请看下面的返回值介绍. 默认是True. 我们之所以增加df参数, 是因为pandas.DataFrame创建和操作速度太慢, 很多情况并不需要使用它. 为了保持向上兼容, df默认是True, 但是如果你的回测速度很慢, 请考虑把df设成False.
- fq: 复权选项:
    - `'pre'`: 前复权(根据'use_real_price'选项不同含义会有所不同, 参见set_option), 默认是前

复权
  - `None`: 不复权, 返回实际价格
  - `'post'`: 后复权

返回

- df=True
  pandas.DataFrame对象, 行索引是datetime.datetime对象, 列索引是属性名字. 比如: 如果当前时间是2015-01-07, `attribute_history('000300.XSHG', 2)` 将返回:

	open	close	high	low	volume	money
2015-01-05 00:00:00	3566.09	3641.54	3669.04	3551.51	451198098.0	519849817448.0
2015-01-06 00:00:00	3608.43	3641.06	3683.23	3587.23	420962185.0	498529588258.0

- df=False:
  dict, key是股票代码, 值是一个numpy数组numpy.ndarray, 对应上面的DataFrame的每一列, 例如 `attribute_history('000300.XSHG', 2, df=False)` 将返回:

```
{
    'volume': array([  4.51198098e+08,   4.20962185e+08]),
    'money': array([  5.19849817e+11,   4.98529588e+11]),
    'high': array([ 3669.04,  3683.23]),
    'low': array([ 3551.51,  3587.23]),
    'close': array([ 3641.54,  3641.06]),
    'open': array([ 3566.09,  3608.43])
}
```

示例

```
stock = '000001.XSHE'
h = attribute_history(stock, 5, '1d', ('open','close', 'volume', 'factor')) #
取得000001(平安银行)过去5天的每天的开盘价，收盘价，交易量，复权因子
# 不管df等于True还是False，下列用法都是可以的
h['open'] #过去5天的每天的开盘价，一个pd.Series对象，index是datatime
h['close'][-1] #昨天的收盘价
h['open'].mean()

# 下面的pandas.DataFrame的特性，df=False时将不可用
# 行的索引可以是整数，也可以是日期的各种形式:
h['open']['2015-01-05']
h['open'][datetime.date(2015, 1, 5)]
h['open'][datetime.datetime(2015, 1, 5)]

# 按行取数据
h.iloc[-1] #昨天的开盘价和收盘价，一个pd.Series对象，index是字符串:'open'/'close'
h.iloc[-1]['open'] #昨天的开盘价
h.loc['2015-01-05']['open']

# 高级运算
h = h[h['volume'] > 1000000] # 只保留交易量>1000000股的行
h['open'] = h['open']/h['factor'] #让open列都跟factor列相除，把价格都转化成原始价格
h['close'] = h['close']/h['factor']
```

get_current_data ♠ - 获取当前时间数据

```
get_current_data()
```

回测环境/模拟专用API

获取当前单位时间（当天/当前分钟）的涨跌停价, 是否停牌，当天的开盘价等。

回测时, 通过 API 获取到的是前一个单位时间(天/分钟)的数据, 而有些数据, 我们在这个单位时间是知道的, 比如涨跌停价, 是否停牌, 当天的开盘价. 我们添加了这个API用来获取这些数据.

参数

- 现在不需要传入, 即使传入了, 返回的 dict 也是空的, dict 的 value 会按需获取.

返回值
一个dict, 其中 key 是股票代码, value 是拥有如下属性的对象

- high_limit: 涨停价
- low_limit: 跌停价
- paused: 是否停止或者暂停了交易, 当停牌、未上市或者退市后返回 True
- is_st: 是否是 ST(包括ST, *ST)，是则返回 True，否则返回 False
- day_open: 当天开盘价
- name: 股票现在的名称, 可以用这个来判断股票当天是否是 ST, *ST, 是否快要退市
- industry_code: 股票现在所属行业代码, 参见 行业概念数据

注意

- 为了加速, 返回的 dict 里面的数据是按需获取的, dict 初始是空的, 当你使用

`current_data[security]` 时(假设 current_data 是返回的 dict), 该 security 的数据才会被获取.
- 返回的结果只在当天有效, 请不要存起来到隔天再用

示例

```
def handle_data(context, data):
    current_data = get_current_data()
    print current_data
    print current_data['000001.XSHE']
    print current_data['000001.XSHE'].paused
    print current_data['000001.XSHE'].day_open
```

get_extras - 获取基金净值/期货结算价等

```
get_extras(info, security_list, start_date='2015-01-01', end_date='2015-12-31', df=True, count=None)
```

得到多只标的在一段时间的如下额外的数据:

- is_st: 是否是ST，是则返回 True，否则返回 False
- acc_net_value: 基金累计净值
- unit_net_value: 基金单位净值
- futures_sett_price: 期货结算价
- futures_positions: 期货持仓量

参数

- info: ['is_st', 'acc_net_value', 'unit_net_value', 'futures_sett_price', 'futures_positions'] 中的一个
- security_list: 股票列表
- start_date/end_date: 开始结束日期, 同 get_price
- df: 返回pandas.DataFrame对象还是一个dict, 同 history
- count: 数量, **与 start_date 二选一, 不可同时使用**, 必须大于 0. 表示取 end_date 往前的 count 个交易日的数据

返回值

- df=True:
  pandas.DataFrame对象, 列索引是股票代号, 行索引是datetime.datetime, 比如 `get_extras('acc_net_value', ['510300.XSHG', '510050.XSHG'], start_date='2015-12-01', end_date='2015-12-03')` 返回:

	510300.XSHG	510050.XSHG
2015-12-01 00:00:00	1.395	3.119
2015-12-02 00:00:00	1.4432	3.251
2015-12-03 00:00:00	1.4535	3.254

`get_extras('is_st', ['000001.XSHE', '000018.XSHE'], start_date='2013-12-01', end_date='2013-12-03')` 返回:

	000001.XSHE	000018.XSHE
2013-12-02 00:00:00	False	True
2013-12-03 00:00:00	False	True

- df=False
  一个dict, key是基金代号, value是numpy.ndarray, 比如 `get_extras('acc_net_value', ['510300.XSHG', '510050.XSHG'], start_date='2015-12-01', end_date='2015-12-03', df=False)` 返回:

```
{
    u'510050.XSHG': array([ 3.119,  3.251,  3.254]),
    u'510300.XSHG': array([ 1.395 ,  1.4432,  1.4535])
}
```

get_fundamentals - 查询财务数据

```
get_fundamentals(query_object, date=None, statDate=None)
```

查询财务数据，详细的数据字段描述请点击财务数据文档查看

date和statDate参数只能传入一个:

- 传入date时, 查询**指定日期date收盘后所能看到的最近(对市值表来说, 最近一天, 对其他表来说, 最近一个季度)的数据**, 我们会查找上市公司在这个日期之前(包括此日期)发布的数据, 不会有未来函数.
- 传入statDate时, 查询 **statDate 指定的季度或者年份的财务数据**. 注意:
  1. 由于公司发布财报不及时, 一般是看不到当季度或年份的财务报表的, 回测中使用这个数据可能会有未来函数, 请注意规避.
  2. 由于估值表每天更新, 当按季度或者年份查询时, 返回季度或者年份最后一天的数据
  3. **对于年报数据, 我们目前只有现金流表和利润表**, 当查询其他表时, 会返回该年份最后一个季报的数据

当 date 和 statDate 都不传入时, 相当于使用 date 参数, date 的默认值下面会描述.

参数

- query_object: 一个sqlalchemy.orm.query.Query对象 (http://docs.sqlalchemy.org/en/rel_1_0/orm/query.html), 可以通过全局的 query 函数获取 Query 对象
- date: 查询日期, 一个字符串(格式类似'2015-10-15')或者datetime.date/datetime.datetime对象, 可以是None, 使用默认日期. 这个默认日期在回测和研究模块上有点差别:

  1. 回测模块: 默认值会随着回测日期变化而变化, 等于 context.current_dt 的前一天(实际生活中我们只能看到前一天的财报和市值数据, 所以要用前一天)
  2. 研究模块: 使用平台财务数据的最新日期, 一般是昨天.
     如果传入的 date 不是交易日, 则使用这个日期之前的最近的一个交易日

- statDate: 财报统计的季度或者年份, 一个字符串, 有两种格式:

  1. 季度: 格式是: **年 + 'q' + 季度序号**, 例如: '2015q1', '2013q4'.

2. 年份: 格式就是年份的数字, 例如: '2015', '2016'.

返回

返回一个 pandas.DataFrame, 每一行对应数据库返回的每一行(可能是几个表的联合查询结果的一行), 列索引是你查询的所有字段

注意：

1. 为了防止返回数据量过大, 我们每次最多返回10000行
2. 当相关股票上市前、退市后，财务数据返回各字段为空

示例

```
# 查询'000001.XSHE'的所有市值数据, 时间是2015-10-15
q = query(
    valuation
).filter(
    valuation.code == '000001.XSHE'
)
df = get_fundamentals(q, '2015-10-15')
# 打印出总市值
log.info(df['market_cap'][0])
```

```
# 获取多只股票在某一日期的市值, 利润
df = get_fundamentals(query(
        valuation, income
    ).filter(
        # 这里不能使用 in 操作, 要使用in_()函数
        valuation.code.in_(['000001.XSHE', '600000.XSHG'])
    ), date='2015-10-15')
```

```
# 选出所有的总市值大于1000亿元, 市盈率小于10, 营业总收入大于200亿元的股票
df = get_fundamentals(query(
        valuation.code, valuation.market_cap, valuation.pe_ratio, income.tota
l_operating_revenue
    ).filter(
        valuation.market_cap > 1000,
        valuation.pe_ratio < 10,
        income.total_operating_revenue > 2e10
    ).order_by(
        # 按市值降序排列
        valuation.market_cap.desc()
    ).limit(
        # 最多返回100个
        100
    ), date='2015-10-15')
```

```
# 使用 or_ 函数: 查询总市值大于1000亿元 **或者** 市盈率小于10的股票
from sqlalchemy.sql.expression import or_
get_fundamentals(query(
        valuation.code
    ).filter(
        or_(
            valuation.market_cap > 1000,
            valuation.pe_ratio < 10
        )
    ))
```

```
# 查询平安银行2014年四个季度的季报, 放到数组中
q = query(
        income.statDate,
        income.code,
        income.basic_eps,
        balance.cash_equivalents,
        cash_flow.goods_sale_and_service_render_cash
    ).filter(
        income.code == '000001.XSHE',
    )

rets = [get_fundamentals(q, statDate='2014q'+str(i)) for i in range(1, 5)]
```

```
# 查询平安银行2014年的年报
q = query(
        income.statDate,
        income.code,
        income.basic_eps,
        cash_flow.goods_sale_and_service_render_cash
    ).filter(
        income.code == '000001.XSHE',
    )

ret = get_fundamentals(q, statDate='2014')
```

get_index_stocks - 获取指数成份股

```
get_index_stocks(index_symbol, date=None)
```

获取一个指数给定日期在平台可交易的成分股列表，请点击指数列表查看指数信息

参数

- index_symbol: 指数代码
- date: 查询日期, 一个字符串(格式类似'2015-10-15')或者datetime.date/datetime.datetime对象, 可以是None, 使用默认日期. 这个默认日期在回测和研究模块上有点差别:
    1. 回测模块: 默认值会随着回测日期变化而变化, 等于context.current_dt
    2. 研究模块: 默认是今天

返回
返回股票代码的list

示例

```
# 获取所有沪深300的股票
stocks = get_index_stocks('000300.XSHG')
log.info(stocks)
```

get_industry_stocks - 获取行业成份股

```
get_industry_stocks(industry_code, date=None)
```

获取在给定日期一个行业的所有股票，行业分类列表见数据页面-行业概念数据。

参数

- industry_code: 行业编码
- date: 查询日期, 一个字符串(格式类似'2015-10-15')或者datetime.date/datetime.datetime对象, 可以是None, 使用默认日期. 这个默认日期在回测和研究模块上有点差别:
    1. 回测模块: 默认值会随着回测日期变化而变化, 等于context.current_dt
    2. 研究模块: 默认是今天

返回
返回股票代码的list

示例

```
# 获取计算机/互联网行业的成分股
stocks = get_industry_stocks('I64')
```

get_concept_stocks - 获取概念成份股

```
get_concept_stocks(concept_code, date=None)
```

获取在给定日期一个概念板块的所有股票，概念板块分类列表见数据页面-行业概念数据。

参数

- concept_code: 概念板块编码
- date: 查询日期, 一个字符串(格式类似'2015-10-15')或者datetime.date/datetime.datetime对象, 可以是None, 使用默认日期. 这个默认日期在回测和研究模块上有点差别:
    1. 回测模块: 默认值会随着回测日期变化而变化, 等于context.current_dt
    2. 研究模块: 默认是今天

返回
返回股票代码的list

示例

```
# 获取风力发电概念板块的成分股
stocks = get_concept_stocks('GN036')
```

get_all_securities - 获取所有标的信息

```
get_all_securities(types=[], date=None)
```

获取平台支持的所有股票、基金、指数、期货信息

参数

- types: list: 用来过滤securities的类型, list元素可选: ‘stock’, ‘fund’, ‘index’, ‘futures’, ‘etf’, ‘lof’, ‘fja’, ‘fjb’. **types为空时返回所有股票, 不包括基金,指数和期货**
- date: 日期, 一个字符串或者 datetime.datetime/datetime.date 对象, 用于获取某日期还在上市的股票信息. 默认值为 None, 表示获取所有日期的股票信息

返回
pandas.DataFrame, 比如: `get_all_securities()[:2]` 返回:

	display_name	name	start_date	end_date	type
000001.XSHE	平安银行	PAYH	1991-04-03	9999-01-01	stock
000002.XSHE	万 科 A	WKA	1991-01-29	9999-01-01	stock

- display_name: 中文名称
- name: 缩写简称
- start_date: 上市日期
- end_date: 退市日期，如果没有退市则为2200-01-01
- type: 类型，stock(股票)，index(指数)，etf(ETF基金)，fja（分级A），fjb（分级B）

示例

```
def initialize(context):
    #获得所有股票列表
    log.info(get_all_securities())
    log.info(get_all_securities(['stock']))

    #将所有股票列表转换成数组
    stocks = list(get_all_securities(['stock']).index)

    #获得所有指数列表
    get_all_securities(['index'])

    #获得所有基金列表
    df = get_all_securities(['fund'])

    #获取所有期货列表
    get_all_securities(['futures'])

    #获得etf基金列表
    df = get_all_securities(['etf'])
    #获得lof基金列表
    df = get_all_securities(['lof'])
    #获得分级A基金列表
    df = get_all_securities(['fja'])
    #获得分级B基金列表
    df = get_all_securities(['fjb'])

    #获得2015年10月10日还在上市的所有股票列表
    get_all_securities(date='2015-10-10')
    #获得2015年10月10日还在上市的 etf 和 lof 基金列表
    get_all_securities(['etf', 'lof'], '2015-10-10')
```

get_security_info - 获取单个标的信息

```
get_security_info(code)
```

获取股票/基金/指数的信息.

参数

- code: 证券代码

返回值

- 一个对象, 有如下属性:
    - display_name: 中文名称
    - name: 缩写简称
    - start_date: 上市日期, datetime.date 类型
    - end_date: 退市日期， datetime.date 类型, 如果没有退市则为2200-01-01
    - type: 类型，stock(股票)，index(指数)，etf(ETF基金)，fja（分级A），fjb（分级B）
    - parent: 分级基金的母基金代码

示例

```
# 获取基金的母基金，下面的判断为真.
assert get_security_info('502050.XSHG').parent == '502048.XSHG'
```

jqdata模块

我们新增了 jqdata 模块用来提供更多数据, 如果要使用下面的 API 请先导入 jqdata 模块, 如下所示:

```
from jqdata import *
```

get_all_trade_days - 获取所有交易日

```
get_all_trade_days()
```

获取所有交易日, 不需要传入参数, 返回一个包含所有交易日的 numpy.ndarray, 每个元素为一个 datetime.date 类型.

get_trade_days - 获取指定范围交易日

```
get_trade_days(start_date=None, end_date=None, count=None)
```

获取指定日期范围内的所有交易日, 返回 numpy.ndarray, 包含指定的 start_date 和 end_date, 默认返回至 datatime.date.today() 的所有交易日

参数

- start_date: 开始日期, **与 count 二选一, 不可同时使用**. str/datetime.date/datetime.datetime 对象
- end_date: 结束日期, str/datetime.date/datetime.datetime 对象, 默认为 datetime.date.today()
- count: 数量, **与 start_date 二选一, 不可同时使用**, 必须大于 0. 表示取 end_date 往前的 count 个交易日，包含 end_date 当天。

get_mtss - 获取融资融券信息

```
get_mtss(security_list, start_date=None, end_date=None, fields=None, count=Non
e)
```

获取一只或者多只股票在一个时间段内的融资融券信息

参数

- security_list: 一只股票代码或者一个股票代码的 list
- start_date: 开始日期, **与 count 二选一, 不可同时使用**. 一个字符串或者 datetime.datetime/datetime.date 对象, 默认为平台提供的数据的最早日期
- end_date: 结束日期, 一个字符串或者 datetime.date/datetime.datetime 对象, 默认为 datetime.date.today()
- count: 数量, **与 start_date 二选一，不可同时使用**, 必须大于 0. 表示返回 end_date 之前 count 个交易日的数据, 包含 end_date
- fields: 字段名或者 list, 可选. 默认为 None, 表示取全部字段, 各字段含义如下：

字段名	含义
date	日期
sec_code	股票代码
fin_value	融资余额
fin_buy_value	融资买入额
fin_refund_value	融资偿还额
sec_value	融券余额
sec_sell_value	融券卖出额
sec_refund_value	融券偿还额
fin_sec_value	融资融券余额

返回

返回一个 pandas.DataFrame 对象，默认的列索引为取得的全部字段. 如果给定了 fields 参数, 则列索引与给定的 fields 对应.

示例

```
# 获取一只股票的融资融券信息
get_mtss('000001.XSHE', '2016-01-01', '2016-04-01')
get_mtss('000001.XSHE', '2016-01-01', '2016-04-01', fields=["date", "sec_cod
e", "fin_value", "fin_buy_value"])
get_mtss('000001.XSHE', '2016-01-01', '2016-04-01', fields="sec_sell_value")

# 获取多只股票的融资融券信息
get_mtss(['000001.XSHE', '000002.XSHE', '000099.XSHE'], '2015-03-25', '2016-0
1-25')
get_mtss(['000001.XSHE', '000002.XSHE', '000099.XSHE'], '2015-03-25', '2016-0
1-25', fields=["date", "sec_code", "sec_value", "fin_buy_value", "sec_sell_val
ue"])

# 获取股票 000001.XSHE 在日期 2016-06-30 往前 20 个交易日的融资融券信息
get_mtss('000001.XSHE', end_date="2016-06-30", count=20)
# 获取股票 000001.XSHE 往前 20 个交易日的融资融券信息
get_mtss('000001.XSHE', count=20)
```

get_money_flow - 获取资金流信息

```
get_money_flow(security_list, start_date=None, end_date=None, fields=None, cou
nt=None)
```

获取一只或者多只股票在一个时间段内的资金流向数据

参数

- security_list: 一只股票代码或者一个股票代码的 list
- start_date: 开始日期, **与 count 二选一, 不可同时使用**, 一个字符串或者 datetime.datetime/datetime.date 对象, 默认为平台提供的数据的最早日期
- end_date: 结束日期, 一个字符串或者 datetime.date/datetime.datetime 对象, 默认为 datetime.date.today()
- count: 数量, **与 start_date 二选一，不可同时使用**, 必须大于 0. 表示返回 end_date 之前 count 个交易日的数据, 包含 end_date
- fields: 字段名或者 list, 可选. 默认为 None, 表示取全部字段, 各字段含义如下：

字段名	含义	备注
date	日期	
sec_code	股票代码	
change_pct	涨跌幅(%)	
net_amount_main	主力净额(万)	主力净额 = 超大单净额 + 大单净额
net_pct_main	主力净占比(%)	主力净占比 = 主力净额 / 成交额
net_amount_xl	超大单净额(万)	超大单：大于等于50万股或者100万元的成交单
net_pct_xl	超大单净占比(%)	超大单净占比 = 超大单净额 / 成交额
net_amount_l	大单净额(万)	大单：大于等于10万股或则20万元且小于50万股或者100万元的成交单
net_pct_l	大单净占比(%)	大单净占比 = 大单净额 / 成交额
net_amount_m	中单净额(万)	中单：大于等于2万股或者4万元且小于10万股或则20万元的成交单
net_pct_m	中单净占比(%)	中单净占比 = 中单净额 / 成交额
net_amount_s	小单净额(万)	小单：小于2万股或者4万元的成交单
net_pct_s	小单净占比(%)	小单净占比 = 小单净额 / 成交额

返回

返回一个 pandas.DataFrame 对象，默认的列索引为取得的全部字段. 如果给定了 fields 参数, 则列索引与给定的 fields 对应.

示例

```
# 获取一只股票在一个时间段内的资金流量数据
get_money_flow('000001.XSHE', '2016-02-01', '2016-02-04')
get_money_flow('000001.XSHE', '2015-10-01', '2015-12-30', fields="change_pct")
get_money_flow(['000001.XSHE'], '2010-01-01', '2010-01-30', ["date", "sec_cod
e", "change_pct", "net_amount_main", "net_pct_l", "net_amount_m"])

# 获取多只股票在一个时间段内的资金流向数据
get_money_flow(['000001.XSHE', '000040.XSHE', '000099.XSHE'], '2010-01-01', '2
010-01-30')
# 获取多只股票在某一天的资金流向数据
get_money_flow(['000001.XSHE', '000040.XSHE', '000099.XSHE'], '2016-04-01', '2
016-04-01')

# 获取股票 000001.XSHE 在日期 2016-06-30 往前 20 个交易日的资金流量数据
get_money_flow('000001.XSHE', end_date="2016-06-30", count=20)
# 获取股票 000001.XSHE 往前 20 个交易日的资金流量数据
get_money_flow('000001.XSHE', count=20)
```

gta.run_query - 查询国泰安数据

```
from jqdata import gta
gta.run_query(query_object)
```

查询国泰安数据，详细的数据字段描述请点击国泰安数据查看

注意未来函数，建议使用filter进行过滤

参数

- query_object: 一个sqlalchemy.orm.query.Query对象 (http://docs.sqlalchemy.org/en/rel_1_0/orm/query.html), 可以通过全局的query函数获取Query对象

返回
返回一个pandas.DataFrame, 每一行对应数据库返回的每一行, 列索引是你查询的所有字段

注意：

1. 为了防止返回数据量过大, 我们每次最多返回3000行
2. 不能进行连表查询，即同时查询多张表内数据

示例

```
# 查询'股票基本信息表 - STK_STOCKINFO'的数据, 并返回前10条数据
df = gta.run_query(query(gta.STK_STOCKINFO).limit(10))

# 打印出股票简称
log.info(df['SHORTNAME'])
```

```
# 选出所有的发行价格大于10元, 股票类别为A类的的股票,并返回前20条记录
df = gta.run_query(query(
        gta.STK_STOCKINFO
    ).filter(
        gta.STK_STOCKINFO.ISSUEPRICE > 10,
        gta.STK_STOCKINFO.SHARETYPE == 'A'
    ).limit(20))
```

```
# 将国泰安数据取到的股票代码转化为聚宽所使用的代码形式
df = gta.run_query(query(gta.STK_STOCKINFO))
# 获取国泰安提供的6为股票代码
symbol = df['SYMBOL'][0]
# 转化为带后缀的股票代码
stock_code = normalize_code(symbol)

# symbol输出为'000971'
# stock_code输出为'000971.XSHE'
```

下单函数

order - 按股数下单

```
order(security, amount, style=None, side='long', pindex=0)
```

买卖标的。调用成功后, 您将可以调用get_open_orders取得所有未完成的交易, 也可以调用cancel_order取消交易

参数

- security: 标的代码
- amount: 交易数量, 正数表示买入, 负数表示卖出
- style: 参见order styles, None代表MarketOrder
- side: 'long'/'short'，开空单还是多单。默认为多单，**股票、基金暂不支持开空单。**
- pindex: 在使用set_subportfolios创建了多个仓位时，指定subportfolio 的序号, 从 0 开始, 比如 0 指定第一个 subportfolio, 1 指定第二个 subportfolio，**默认为0。**

返回
Order对象或者None, 如果创建订单成功, 则返回Order对象, 失败则返回None

示例

```
#买入平安银行股票100股
order('000001.XSHE', 100) # 下一个市价单
order('000001.XSHE', 100, MarketOrderStyle()) # 下一个市价单，功能同上
order('000001.XSHE', 100, LimitOrderStyle(10.0)) # 以10块价格下一个限价单

# 在仓位1中开一手沪深300指数期货的空单
order('IF1412.CCFX', 1 , side='short', pindex=1)
# 在仓位1中以3600的限价单，平一手沪深300指数期货的空单
order('IF1412.CCFX', -1 , LimitOrderStyle(3600.0), side='short', pindex=1)
```

可能的失败原因:

1. 股票数量经调整后变成0 (请看下面的说明)
2. 股票停牌
3. 股票未上市或者退市
4. 股票不存在
5. 为股票、基金开了空单
6. 选择了不存在的仓位号，如没有建立多个仓位，而设定pindex的数大于0

对于原因4, 我们会抛出异常停止运行, 因为我们认为这是您代码的bug.

注意:

- 因为下列原因, 有时候实际买入或者卖出的股票数量跟您设置的不一样，这个时候我们会在您的log中添加警告信息。

    1. 买入时会根据您当前的现金来限制您买入的数量
    2. 卖出时会根据您持有股票的数量来限制您卖出的数量
    3. 我们会遵守A股交易规则: 每次交易数量只能是100的整数倍, 但是卖光所有股票时不受这个限制

- 根据交易所规则, 每天结束时会取消所有未完成交易

order_target - 目标股数下单

```
order_target(security, amount, style=None, side='long', pindex=0)
```

买卖标的, 使最终标的的数量达到指定的amount

参数

- security: 标的代码
- amount: 期望的最终数量
- style: 参见order styles, None代表MarketOrder
- side: 'long'/'short'，平空单还是多单。默认为多单，**股票、基金暂不支持开空单。**
- pindex: 在使用set_subportfolios创建了多个仓位时，指定subportfolio 的序号, 从 0 开始, 比如 0为指定第一个 subportfolio, 1 为指定第二个 subportfolio，**默认为0。**

返回
Order对象或者None, 如果创建委托成功, 则返回Order对象, 失败则返回None

示例

```
# 卖出平安银行所有股票
order_target('000001.XSHE', 0)
# 买入平安银行所有股票到100股
order_target('000001.XSHE', 100)
# 平掉仓位1中的空单标的
order_target('IF1412.CCFX', 0 , side='short', pindex=1)
```

order_value - 按价值下单

```
order_value(security, value, style=None, side='long', pindex=0)
```

买卖价值为value的股票，**金融期货暂不支持该API**

参数

- security: 股票名字
- value: 股票价值
- style: 参见order styles, None代表MarketOrder
- side: 'long'/'short'，平空单还是多单。默认为多单，**股票、基金暂不支持开空单。**
- pindex: 在使用set_subportfolios创建了多个仓位时，指定subportfolio 的序号, 从 0 开始, 比如 0为指定第一个 subportfolio, 1 为指定第二个 subportfolio，**默认为0。**

返回
Order对象或者None, 如果创建委托成功, 则返回Order对象, 失败则返回None

示例

```
#卖出价值为10000元的平安银行股票
order_value('000001.XSHE', -10000)
#买入价值为10000元的平安银行股票
order_value('000001.XSHE', 10000)
```

order_target_value - 目标价值下单

```
order_target_value(security, value, style=None, side='long', pindex=0)
```

调整股票仓位到value价值，**金融期货暂不支持该API**

参数

- security: 股票名字
- value: 期望的股票最终价值
- style: 参见order styles, None代表MarketOrder
- side: 'long'/'short'，平空单还是多单。默认为多单，**股票、基金暂不支持开空单。**
- pindex: 在使用set_subportfolios创建了多个仓位时，指定subportfolio 的序号, 从 0 开始, 比如 0为指定第一个 subportfolio, 1 为指定第二个 subportfolio，**默认为0。**

返回
Order对象或者None, 如果创建委托成功, 则返回Order对象, 失败则返回None

示例

```
#卖出平安银行所有股票
order_target_value('000001.XSHE', 0)
#调整平安银行股票仓位到10000元价值
order_target_value('000001.XSHE', 10000)
```

cancel_order - 撤单

```
cancel_order(order)
```

取消订单

参数

- order: Order对象或者order_id

返回
Order对象或者None, 如果取消委托成功, 则返回Order对象, 委托不存在返回None

示例

```
#每个交易日结束运行
def after_trading_end(context):
    # 得到当前未完成订单
    orders = get_open_orders()
    # 循环，撤销订单
    for _order in orders.values():
        cancel_order(_order)
```

get_open_orders - 获取未完成订单

```
get_open_orders()
```

获得当天的所有未完成的订单

参数
无

返回
返回一个dict, key是order_id, value是Order对象

示例

```
#每个交易日结束运行
def after_trading_end(context):
    #得到当前未完成订单
    orders = get_open_orders()
    for _order in orders.values():
        log.info(_order.order_id)
```

get_orders - 获取订单信息

```
get_orders()
```

获取当天的所有订单

参数
无

返回
返回一个dict, key是order_id, value是Order对象

示例

```
#每个交易日结束运行
def after_trading_end(context):
    #得到当天所有订单
    orders = get_orders()
    for _order in orders.values():
        log.info(_order.order_id)
```

get_trades - 获取成交信息

```
get_trades()
```

获取当天的所有成交记录, 一个订单可能分多次成交

参数
无

返回
返回一个dict, key是trade_id, value是Trade对象

示例

```
#每个交易日结束运行
def after_trading_end(context):
    #得到当天所有成交记录
    trades = get_trades()
    for _trade in trades.values():
        log.info(_trade.trade_id)
```

其他函数

定时运行 ♠

- run_monthly
- run_weekly
- run_daily

```
# 按月运行
run_monthly(func, monthday, time='open', reference_security)
# 按周运行
run_weekly(func, weekday, time='open', reference_security)
# 每天内何时运行
run_daily(func, time='open', reference_security)
```

回测环境/模拟专用API

指定每月, 每周或者每天要运行的函数, 可以在具体每月/周的第几个交易日(或者倒数第几天)的某一分钟执行. 可以在任意时间运行.

调用这些函数后, handle_data可以不实现

参数

- func: 一个函数, 此函数必须接受一个参数, 就是 context
- monthday: 每月的第几个交易日, 可以是负数, 表示倒数第几个交易日
- weekday: 每周的第几个交易日, 可以是负数, 表示倒数第几个交易日
- time: 日内执行时间, 具体到分钟, 一个字符串, 可以是具体执行时间, 比如 “10:00”, 或者 “every_bar”, “open”, “before_open” 和 “after_close”, 他们的含义是:
    - every_bar: 在每一个 bar 结束后调用. **按天**会在每天的开盘时调用一次，**按分钟**会在每天的每分钟运行
    - open: 开盘的第一分钟(等同于”9:30”)
    - before_open: 开盘前(等同于”9:20”, 但是请不要使用直接使用此时间, 后续可能会调整)
    - after_close: 收盘后(半小时内运行).
    - 注：当time不等于open/before_open/after_close时, 必须使用**分钟级回测**
- reference_security: 时间的参照标的。如参照’000001.XSHG’，交易时间为9:30-15:00。如参照’IF1512.CCFX’，2016-01-01之后的交易时间为9:30-15:00，在此之前为9:15-15:15。

返回值
None

注意

- **参数 func 必须是一个全局的函数, 不能是类的成员函数**, 示例:

```
def on_week_start(context):
    pass

class MyObject(object):
    def on_week_start2(self, context):
        pass


# OK

run_weekly(on_week_start, 1)

# 错误，下面的语句会报错

run_weekly(MyObject().on_week_start2, 1)
```

- 当 time 指定具体的几点几分时必须使用**分钟级回测**
- 通过history/attribute_history取天数据时，**是不包括当天的数据的**(即使在15:00和after_close里面也是如此)，要取得当天数据，只能取分钟的
- 这些函数可以**重复调用**，比如下面的代码可以在每周的第一个交易日和最后一个交易日分别调用两个函数:

```
def on_week_start(context):
    pass
def on_week_end(context):
    pass
def initialize(context):
    run_weekly(on_week_start, 1)
    run_weekly(on_week_end, -1)
```

- 每次调用这些函数都会产生一个新的定时任务，如果想修改或者删除旧的定时任务，请先调用unschedule_all 来删除所有定时任务，然后再添加新的.
- 在一月/一周交易日数不够以致于monthday/weekday无法满足时，我们会找这周内最近的一个日期来执行，比如，如果某一周只有4个交易日:
    - 若 weekday == 5，我们会在第4个交易日执行
    - 若 weekday == -5，我们会在第1个交易日执行

  如果要避免这样的行为，您可以这样做:

```
def initialize(context):
    run_weekly(weekly, 1)

def weekly(context):
    if context.current_dt.isoweekday() != 1:
        # 不在周一，跳过执行
        return
```

示例

```
def weekly(context):
    print 'weekly %s %s' % (context.current_dt, context.current_dt.isoweekda
y())


def monthly(context):
    print 'monthly %s %s' % (context.current_dt, context.current_dt.month)


def daily(context):
    print 'daily %s' % context.current_dt


def initialize(context):

    # 指定每月第一个交易日，在开盘第一分钟执行
    run_monthly(monthly, 1, 'open')

    # 指定每周倒数第一个交易日，在开盘前执行，此函数中不能下单
    run_weekly(weekly, -1, 'before_open')

    # 指定每天收盘后执行，此函数中不能下单
    run_daily(daily, 'after_close')

    # 指定在每天的10:00运行，必须选择分钟回测，否则不会执行
    run_daily(daily, '10:00')

    # 指定在每天的14:00运行，必须选择分钟回测，否则不会执行
    run_daily(daily, '14:00')
```

取消所有定时运行 ♠

```
# 取消所有定时运行
unschedule_all()
```

画图函数record ♠

```
record(**kwargs)
```

回测环境/模拟专用API

我们会帮您在图表上画出收益曲线和基准的收益曲线，您也可以调用record函数来描画额外的曲线。
因为我们是按天展现的，如果您使用按分钟回测，我们画出的点是您最后一次调用record的值。

参数
很多key=>value形式的参数，key曲线名称，value为值

返回
None

示例

```
# d是一个SecurityUnitData结构体，会画出每个单元时间(天或分钟)的平均价,开始价,结束价
record(price=d.price, open=d.open, close=d.close)
# 也可以画一条100的直线
record(price=100)
```

发送自定义消息 ♠

```
send_message(message, channel='weixin')
```

回测环境/模拟专用API

给用户自己发送消息, 暂时只支持微信消息.

参数
message: 消息内容. 字符串.
channel: 消息渠道, 暂时只支持微信: weixin. 默认值是 weixin

返回值
True/False, 表示是否发送成功. 当发送失败时, 会在日志中显示错误信息.

注意

- 要使用功能, 必须开启模拟交易的 微信通知.
- 此功能只能在 **模拟交易** 中使用, 回测中使用会直接忽略, 无任何提示.
- 微信消息每人每天不超过 **5** 条, 超出会失败.
- 微信消息主页只显示前 **200** 个字符, 点击详情可查看全部消息, 全部消息不得超过 **10000** 个字符.

示例

```
send_message("测试消息")
```

日志log

```
log.error(content)
log.warn(content)
log.info(content)
log.debug(content)
print content1, content2, ...
```

分级别打log,跟python的logging模块一致
print输出的结果等同于log.info, 但是print后面的每一个元素会占用一行

参数
参数可以是字符串、对象等

返回
None

示例

```
log.info(history(10)) # 打印出 history(10)返回的结果
log.info("Selling %s, amount=%s", security, amount) # 打印出一个格式化后的字符串
print history(10), data, context.portfolio
```

设定log级别：log.set_level

```
log.set_level(name, level)
```

设置不同种类的log的级别, 低于这个级别的log不会输出. 所有log的默认级别是debug

参数
name: 字符串, log种类, 必须是'order', 'history', 'strategy'中的一个, 含义分别是:

- order: 调用order系列API产生的log
- history: 调用history系列API(history/attribute_history/get_price)产生的log
- strategy: 您自己在策略代码中打的log

level: 字符串, 必须是'debug', 'info', 'warning', 'error'中的一个, 级别: debug < info < warning < error

返回
None

示例

```
# 过滤掉order系列API产生的比error级别低的log
log.set_level('order', 'error')
```

write_file - 写文件

```
write_file(path, content, append=False)
```

写入内容到研究模块path文件, 写入后, 您可以立即在研究模块中看到这个文件

参数

- path: 相对路径, 相对于您的私有空间的根目录的路径
- content: 文件内容, str或者unicode, 如果是unicode, 则会使用UTF-8编码再存储.可以是二进制内容.
- append: 是否是追加模式, 当为False会清除原有文件内容，默认为False.

返回
None
如果写入失败(一般是因为路径不合法), 会抛出异常

示例

```
write_file("test.txt", "hello world")

# 写入沪深300的股票到HS300.stocks.json文件中
import json
write_file('HS300.stocks.json', json.dumps(get_index_stocks('000300.XSHG')))

# 把 DataFrame 表保存到文件
df = attribute_history('000001.XSHE', 5, '1d') #获取DataFrame表
write_file('df.csv', df.to_csv(), append=False) #写到文件中
```

read_file - 读文件

```
read_file(path)
```

读取你的私有文件(您的私有文件可以在研究模块中看到)

参数
path: 相对路径, 相对于您的私有空间的根目录的路径

返回
返回文件的原始内容, 不做任何decode.

示例

```
#解析json文件
import json
content = read_file('HS300.stocks.json')
securities = json.loads(content)
log.info(securities)

#解析csv文件
import pandas as pd
from six import StringIO
body=read_file("open.csv")
data=pd.read_csv(StringIO(body))
```

自定义python库

您可以在把.py文件放在'研究'的根目录, 然后在回测中就可以通过import的方式来引用此文件. 比如

研究根目录/mylib.py:

```
#-*- coding: utf-8 -*-
# 如果你的文件包含中文，请在文件的第一行使用上面的语句指定你的文件编码

# 用到回测API请加入下面的语句
from kuanke.user_space_api import *

my_stocks = get_index_stocks('000300.XSHG')
```

在策略代码中:

```
# 导入自己创建的库
from mylib import *

def initialize(context):
    log.info(my_stocks)
```

注意: 暂时只能import研究根目录下的.py文件, 还不能import子目录下的文件(比如通过 import a.b.c 来引用 a/b/c.py)

研究中创建回测函数

1. create_backtest

```
create_backtest(algorithm_id, start_date, end_date, frequency="day", initial_cash=10000, initial_positions=None, extras=None, name=None)
```

通过一个策略ID从研究中创建回测，暂不支持期货回测

参数：

- algorithm_id: 策略ID，从策略编辑页的 url 中获取, 比如 ‘https://www.joinquant.com/algorithm/index/edit?algorithmId=xxxx‘，则策略ID为 `xxxx`
- start_date: 回测开始日期
- end_date: 回测结束日期
- frequency: 数据频率，支持 day，minute
- initial_cash: 初始资金
- extras: 额外参数，一个 dict， 用于设置全局的 g 变量，如 extras={‘x’:1, ‘y’:2}，则回测中 g.x = 1, g.y = 2，需要注意的是，该参数的值是 `在 initialize 函数执行之后` 才设置给 g 变量的，所以这会覆盖掉 initialize 函数中 g 变量同名属性的值
- name: 回测名, 用于指定回测名称, 如果没有指定则默认采用策略名作为回测名
- initial_positions: 初始持仓。持仓会根据价格换成现金加到初始资金中，如果没有给定价格则默认获取股票最近的价格。格式如下:

```
initial_positions = [
    {
        'security':'000001.XSHE',
        'amount':'100',
    },
    {
        'security':'000063.XSHE',
        'amount':'100',
        'avg_cost': '1.0'
    },
]
```

返回：

一个字符串, 即 backtest_id

示例：

```
algorithm_id = "xxxx"
extra_vars = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
initial_positions = [
    {
        'security':'000001.XSHE',
        'amount':'100',
    },
    {
        'security':'000063.XSHE',
        'amount':'100',
        'avg_cost': '1.0'
    },
]

params = {
    "algorithm_id": algorithm_id,
    "start_date": "2015-10-01",
    "end_date": "2016-07-31",
    "frequency": "day",
    "initial_cash": "1000000",
    "initial_positions": initial_positions,
    "extras": extra_vars,
}

created_bt_id = create_backtest(**params)
print(created_bt_id)
```

2. get_backtest

```
gt = get_backtest(backtest_id)
```

- 返回：
  - gt.get_status():**获取回测状态**. 返回一个字符串，其含义分别为：
    - none: 未开始
    - running: 正在进行
    - done: 完成
    - failed: 失败
    - canceled: 取消
    - paused: 暂停
    - deleted: 已删除
  - gt.get_params()：**获得回测参数**. 返回一个 dict, 包含调用 create_backtest 时传入的所有信息. (注： algorithm_id，initial_positions，extras 只有在研究中创建的回测才能取到)
  - gt.get_results()：**获得收益曲线**. 返回一个 list，每个交易日是一个 dict，键的含义如下：
    - time: 时间
    - returns: 收益
    - benchmark_returns: 基准收益
    - 如果没有收益则返回一个空的 list
  - gt.get_positions()：**获得持仓详情**. 返回一个 list，每个交易日为一个 dict，键的含义为：
    - time: 时间
    - security: 证券代码
    - security_name: 证券名称

        - amount: 持仓数量
        - price: 股票价格
        - avg_cost: 买入股票平均每股所花的钱
        - closeable_amount: 可平仓数量
        - 如果没有持仓则返回一个空的 list

    - gt.get_orders()：**获得交易详情**. 返回一个 list，每个交易日为一个 dict，键的含义为：
        - time: 时间
        - security: 证券代码
        - security_name: 证券名称
        - action: 交易类型, 开仓('open')/平仓('close')
        - amount: 下单数量
        - filled: 成交数量
        - price: 平均成交价格
        - commission: 交易佣金
        - 如果没有持仓则返回一个空的 list

    - gt.get_records()：**获得所有 record 记录**. 返回一个 list，每个交易日为一个 dict，键是 time 以及调用 record() 函数时设置的值.
    - gt.get_risk()：**获得总的风险指标**. 返回一个 dict，键是各类收益指标数据，如果没有风险指标则返回一个空的 dict.
    - gt.get_period_risks()：**获得分月计算的风险指标**. 返回一个 dict，键是各类指标, 值为一个 pandas.DataFrame. 如果没有风险指标则返回一个空的 dict.

示例：

```
gt = get_backtest("xxxx")

gt.get_status()         # 获取回测状态
gt.get_params()         # 获取回测参数
gt.get_results()        # 获取收益曲线
gt.get_positions()      # 获取所有持仓列表
gt.get_orders()         # 获取交易列表
gt.get_records()        # 获取所有record()记录
gt.get_risk()           # 获取总的风险指标
gt.get_period_risks()   # 获取分月计算的风险指标
```

股票代码格式转换

```
normalize_code()
```

将其他形式的股票代码转换为聚宽可用的股票代码形式。

仅适用于A股市场股票代码以及基金代码

示例

```
#输入
for code in ('000001', 'SZ000001', '000001SZ', '000001.sz', '000001.XSHE'):
        print normalize_code(code)

#输出
000001.XSHE
000001.XSHE
000001.XSHE
000001.XSHE
000001.XSHE
```

性能分析 ♠

```
enable_profile()
```

回测环境/模拟专用API

开启性能分析功能, **请在所有代码之前调用这句话(即在便以页面的最上方放置该代码)**, 只在点击 '运行回测' 运行的时候才能看到性能分析结果.
开启性能分析之后, 你会在回测结果页面看到性能分析结果.
请注意, 不需要时, 请不要调用此函数, 因为它本身会影响程序性能.

结果示例(真实输出中没有中文说明):

```
// 时间单位: 微秒
Timer unit: 1e-06 s

// 函数执行总时间
Total time: 0.00277 s
// 文件名
File: user_code.py
// 函数名
Function: initialize at line 3

//  行号, 这一行执行次数,总执行时间,每次执行时间, 这一行执行时间在整个函数的比例
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def initialize(context):
     4                                               # 定义一个全局变量, 保存要操
作的股票
     5                                               # 000001(股票:平安银行)
     6         1           31     31.0      1.1      g.security = '000001.XSH
E'
     7                                               # 初始化此策略
     8                                               # 设置我们要操作的股票池, 这
里我们只操作一支股票
     9         1         2739   2739.0     98.9      set_universe([g.securit
y])

Total time: 0.426325 s
File: user_code.py
Function: handle_data at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           def handle_data(context, dat
a):
    13       122          398      3.3      0.1      security = g.security
    14                                               # 取得过去五天的平均价格
    15       122       168565   1381.7     39.5      average_price = data[secu
rity].mavg(5)
    16                                               # 取得上一时间点价格
    17       122          493      4.0      0.1      current_price = data[secu
rity].price
    18                                               # 取得当前的现金
    19       122          240      2.0      0.1      cash = context.portfoli
o.cash
    20
    21                                               # 如果上一时间点价格高出五天平
均价1%, 则全仓买入
    22       122          396      3.2      0.1      if current_price > 1.01*a
verage_price:
    23                                                   # 计算可以买多少只股票
    24        30          124      4.1      0.0          number_of_shares = in
t(cash/current_price)
    25                                                   # 购买量大于0时, 下单
    26        30           56      1.9      0.0          if number_of_shares >
0:
    27                                                       # 买入股票
    28        30        83190   2773.0     19.5              order(security,
```

```
+number_of_shares)
    29                                                  # 记录这次买入
    30        30        16202    540.1      3.8          log.info("Buying
%s" % (security))
    31                                              # 如果上一时间点价格低于五天平
均价, 则空仓卖出
    32        92         1119     12.2      0.3      elif current_price < aver
age_price and context.portfolio.positions[security].amount > 0:
    33                                                  # 卖出所有股票,使这只股票
的最终持有量为0
    34        13        86702   6669.4     20.3          order_target(securit
y, 0)
    35                                                  # 记录这次卖出
    36        13         8210    631.5      1.9          log.info("Selling %s"
% (security))
    37                                              # 画出上一时间点价格
    38       122        60630    497.0     14.2      record(stock_price=data[s
ecurity].price)
```

期货保证金预警

判断指定仓位，是否低于指定的保证金比率，高于该比例返回`False`，低于该比例返回`True`.

```
context.subportfolios[i].is_dangerous(margin_rate)
```

其中 i 是要查询的仓位编号, margin_rate 是要查询的保证金比例

返回： True 或 False

示例：

```
# 查询 subportfolios[1] 中保证金是否低于 20%
context.subportfolios[1].is_dangerous(0.2)

# 低于则返回True， 高则返回 False
```

账户分仓操作

初始化仓位 subportfolios

```
set_subportfolios(portfolio_configs)
```

初始化或者修改 subportfolios 的配置，只能在 initialize 中调用, 每个 portfolio_config 中 cash 的和应该等于总的初始资金

portfolio_configs 设置方法

```
SubPortfolioConfig(cash,type)
```

- cash: 仓位初始资金
- type: 可操作标的的类型，'stock' / 'index_futures', 其中 stock 包括股票和基金，index_futures 指金融期货

示例

```
init_cash = 500000 # 定义一个变量
# 设定subportfolios[0]为股票和基金仓位，初始资金为 init_cash 变量代表的数值
# 设定subportfolios[1]为金融期货仓位，初始资金为初始资金减去 init_cash
set_subportfolios([SubPortfolioConfig(cash=init_cash ,type='stock'),\
                   SubPortfolioConfig(cash=context.portfolio.starting_cash- in
it_cash,type='index_futures')])
```

SubPortfolio

某个仓位的资金，标的信息，如不使用 SubPortfolioConfig 设置多仓位，默认只有subportfolios[0]一个仓位，Portfolio 指向该仓位。

有关 SubPortfolio 详情见对象 - SubPortfolio

仓位间转移资金

```
transfer_cash(from_pindex, to_pindex, cash)
```

从序号为 from_pindex 的 subportfolio 转移 cash 到序号为 to_pindex 的 subportfolio
资金转移及时到账

示例

```
从subportfolio[0] 向 subportfolio[1] 转移 500000
transfer_cash(from_pindex=0, to_pindex=1, cash=500000)
```

对象

全局对象 g

全局对象 g，用来存储用户的各类可被pickle.dumps函数序列化的全局数据

在模拟盘中，如果中途进程中断，我们会使用pickle.dumps序列化所有的g下面的变量内容, 保存到磁盘中，再启动的时候模拟盘就不会有任何数据影响。如果没有用g声明，会出现模拟盘重启后，变量数据丢失的问题。

如果不想 g 中的某个变量被序列化, 可以让变量以 '__' 开头, 这样, 这个变量在序列化时就会被忽略

更多模拟盘细节, 请看 模拟盘注意事项.

```
def initialize(context):
    g.security = "000001.XSHE"
    g.count = 1
    g.flag = 0

def process_initialize(context):
    # 保存不能被序列化的对象，进程每次重启都初始化，更多信息，请看 [process_initialize]
    g.__q = query(valuation)

def handle_data(context, data):
    log.info(g.security)
    log.info(g.count)
    log.info(g.flag)
```

Context

- subportfolios: 当前单个操作仓位的资金、标的信息，是一个SubPortfolio 的数组
- portfolio: 账户信息，即subportfolios 的汇总信息, Portfolio对象，单个操作仓位时，portfolio 指向 subportfolios[0]
- current_dt: 当前单位时间的开始时间, datetime.datetime对象,
    - 按天回测时, hour = 9, minute = 30, second = microsecond = 0,
    - 按分钟回测时, second = microsecond = 0

- previous_date: 前一个交易日, datetime.date对象, 注意, 这是一个日期, 是 date, 而不是 datetime
- universe: 查询set_universe()设定的股票池, 比如: ['000001.XSHE', '600000.XSHG']
- run_params: 表示此次运行的参数, 有如下属性

    - start_date: 回测/模拟开始日期, datetime.date对象
    - end_date: 回测/模拟结束日期, datetime.date对象
    - type: 运行方式, 如下三个字符串之一
        - 'simple_backtest': 回测, 通过点击'编译运行'运行
        - 'full_backtest': 回测, 通过点击'运行回测'运行
        - 'sim_trade': 模拟交易

    - frequency: 运行频率, 如下三个字符串之一
        - 'day'
        - 'minute'
        - 'tick'

- 为了让从其他平台迁移过来的同学更顺手的使用系统, 我们对此对象也做了和 g 一样的处理:

    - 可以添加自己的变量, 每次进程关闭时持久保存, 进程重启时恢复.
    - 以 '__' 开头的变量不会被持久保存
    - 如果添加的变量与系统的冲突, 将覆盖掉系统变量, 如果想恢复系统变量, 请删除自己的变量. 示例:

```
def handle_data(context, data):
    # 执行下面的语句之后, context.portfolio 的整数 1
    context.portfolio = 1
    log.info(context.portfolio)
    # 要恢复系统的变量, 只需要使用下面的语句即可
    del context.portfolio
    # 此时, context.portfolio 将变成账户信息.
    log.info(context.portfolio.portfolio_value)
```

- 我们以后可能会往 context 添加新的变量来支持更多功能, 为了减少不必要的迷惑, 还是建议大家使用 g

示例

```
def handle_data(context, data):

    #获得当前回测相关时间
    year = context.current_dt.year
    month = context.current_dt.month
    day = context.current_dt.day
    hour = context.current_dt.hour
    minute = context.current_dt.minute
    second = context.current_dt.second
    #得到"年-月-日"格式
    date = context.current_dt.strftime("%Y-%m-%d")
    #得到周几
    weekday = context.current_dt.isoweekday()

    # 获取账户的持仓价值
    positions_value = context.portfolio.positions_value

    # 获取仓位subportfolios[0]的可用资金
    available_cash = context.subportfolios[0].available_cash

    # 获取subportfolios[0]中多头仓位的security的持仓成本
    hold_cost = context.subportfolios[0].long_positions[security].hold_cost
```

SubPortfolio

某个仓位的资金，标的信息，如未使用 SubPortfolioConfig 设置多仓位，默认只有subportfolios[0]一个仓位，Portfolio 指向该仓位。

- inout_cash: 累计出入金, 比如初始资金 1000, 后来转移出去 100, 则这个值是 1000 - 100
- available_cash: 可用资金, 可用来购买证券的资金
- transferable_cash: 可取资金, 即可以提现的资金, 不包括今日卖出证券所得资金
- locked_cash: 挂单锁住资金
- margin: 保证金，股票、基金保证金都为100%；期货保证金会实时更新, 总是等于当前期货价值 乘以 保证金比率, 当保证金不足时, 强制平仓. 平仓顺序是: 亏损多的(相对于开仓均价)先平仓
- long_positions: 多单的仓
- 位, 一个 dict, key 只证券代码, value 是 Position对象
- short_positions: 空单的仓位, 一个 dict, key 只证券代码, value 是 Position对象
- total_value: 总的权益, 包括现金, 保证金, 仓位的总价值, 可用来计算收益
- positions_value: 持仓价值, 股票基金才有持仓价值, 期货为0

Portfolio

账户当前的资金，标的信息，即所有标的操作仓位的信息汇总。如未使用 SubPortfolioConfig 设置多仓位，默认只有subportfolios[0]一个仓位，Portfolio 指向该仓位。

- inout_cash: 累计出入金, 比如初始资金 1000, 后来转移出去 100, 则这个值是 1000 - 100
- available_cash: 可用资金, 可用来购买证券的资金
- transferable_cash: 可取资金, 即可以提现的资金, 不包括今日卖出证券所得资金
- locked_cash: 挂单锁住资金
- margin: 保证金，股票、基金保证金都为100%
- positions: 等同于 long_positions
- long_positions: 多单的仓位, 一个 dict, key 只证券代码, value 是 Position对象
- short_positions: 空单的仓位, 一个 dict, key 只证券代码, value 是 Position对象
- total_value: 总的权益, 包括现金, 保证金, 仓位的总价值, 可用来计算收益
- returns: 总权益的累计收益
- starting_cash: 初始资金, 现在等于 inout_cash
- positions_value: 持仓价值, 股票基金才有持仓价值, 期货为0
- capital_used: 已用资金, 等于 starting_cash - cash
- ~~cash: **已过时**，等价于 available_cash~~
- ~~portfolio_value: **已过时**，等价于 total_value~~
- ~~unsell_positions: **已过时, 请使用 positions 代替**, 当前持有的不可以卖出的持仓(比如在A股T+1市场, 今天购票的股票), 并没有考虑股票今天是否停牌, 一个dict, key是股票代码, value是Position对象.~~

Position

持有的某个标的的信息

- security: 标的代码
- price: 最新行情价格
- avg_cost: 开仓均价，买入标的的加权平均价, 计算方法是:
  (buy_volume1 * buy_price1 + buy_volume2 * buy_price2 + …) / (buy_volume1 + buy_volume2 + …)
  每次买入后会调整avg_cost, 卖出时avg_cost不变. 这个值也会被用来计算浮动盈亏.
- hold_cost: 持仓成本，**针对期货有效。**
- total_amount: 总仓位, 但不包括挂单冻结仓位
- closeable_amount: 可卖出的仓位
- today_amount: 今天开的仓位
- locked_amount: 挂单冻结仓位
- value: 标的价值，计算方法是: price * total_amount * multiplier, 其中股票、基金的multiplier为1，期货为相应的合约乘数
- side: 多/空，'long' or 'short'
- pindex: 仓位索引，subportfolio index
- ~~sellable_amount: **已过时**, 为了向前兼容, 等同于 closeable_amount~~
- ~~amount: **已过时**, 为了向前兼容, 等同于 closeable_amount~~

SecurityUnitData

一个单位时间内的股票的数据

基本属性

以下属性也能通过history/attribute_history/get_price获取到

- open: 时间段开始时价格
- close: 时间段结束时价格
- low: 最低价
- high: 最高价
- volume: 成交的股票数量
- money: 成交的金额
- factor: 前复权因子, 我们提供的价格都是前复权后的, 但是利用这个值可以算出原始价格, 方法是价格除以factor, 比如: `close/factor`
- high_limit: 涨停价
- low_limit: 跌停价
- avg: 这段时间的平均价, 等于`money/volume`
- ~~price: **已经过时**, 为了向前兼容, 等同于 avg~~
- pre_close: 前一个单位时间结束时的价格, 按天则是前一天的收盘价, 按分钟这是前一分钟的结束价格
- paused: bool值, 这只股票是否停牌, 停牌时open/close/low/high/pre_close依然有值,都等于停牌前的收盘价, volume=money=0

额外的属性和方法

- security: 股票代码, 比如'000001.XSHE'
- returns: 股票在这个单位时间的相对收益比例, 等于`(close-pre_close)/pre_close`
- isnan(): 数据是否有效, 当股票未上市或者退市时, 无数据, isnan()返回True
- `mavg(days, field='close')`: 过去days天的每天收盘价的平均值, 把field设成'avg'(等同于已过时的'price')则为每天均价的平均价, 下同
- `vwap(days)`: 过去days天的每天均价的加权平均值, 以days=2为例子, 算法是:

```
(price1 * volume1 + price2 * volume2) / (volume1 + volume2)
```

- `stddev(days)`: 过去days天的每天收盘价的标准差
- 注：mavg/vwap/stddev:都会跳过停牌日期, 如果历史交易天数不足, 则返回nan

Trade对象

订单的一次交易记录,一个订单可能分多次交易.

- time: 交易时间, datetime.datetime对象
- amount: 交易数量
- price: 交易价格
- trade_id: 交易记录id
- order_id: 对应的订单id

Order对象

买卖订单

- status: 状态, 一个OrderStatus值
- add_time: 订单添加时间, datetime.datetime对象
- is_buy: bool值, 买还是卖，对于期货:
    - 开多/平空 -> 买
    - 开空/平多 -> 卖
- amount: 下单数量, 不管是买还是卖, 都是正数

- filled: 已经成交的股票数量, 正数
- security: 股票代码
- order_id: 订单ID
- price: 平均成交价格, 已经成交的股票的平均成交价格(一个订单可能分多次成交)
- avg_cost: 卖出时表示下卖单前的此股票的持仓成本, 用来计算此次卖出的收益. 买入时表示此次买入的均价(等同于price).
- side: 多/空，'long'/'short'
- action: 开/平，'open'/'close'

OrderStatus - 订单状态

订单状态, Enum特性使用的第三方库(https://pypi.python.org/pypi/enum34)

```
class OrderStatus(Enum):
    # 订单未完成，无任何成交
    open = 0
    # 订单未完成，部分成交
    filled = 1
    # 订单完成，已撤销，可能有成交，需要看 Order.filled 字段
    canceled = 2
    # 订单完成，交易所已拒绝，可能有成交，需要看 Order.filled 字段
    rejected = 3
    # 订单完成，全部成交，Order.filled 等于 Order.amount
    held = 4
```

OrderStyle - 下单方式

下单方式, 有如下子类

- 市价单: 不论价格, 直接下单, 直到交易全部完成

```
class MarketOrderStyle(OrderStyle):
    pass
```

- 限价单: 指定一个价格, 买入时不能高于它, 卖出时不能低于它, 如果不满足, 则等待满足后再交易

```
class LimitOrderStyle(OrderStyle):
    def __init__(self, limit_price):
        self.limit_price = limit_price
```

Python库

标准库

我们支持所有Python标准库(https://docs.python.org/2/library/index.html)，您可以通过import的方式进行引入, 下面列出了一些常用的库：

库名	帮助文档
array	https://docs.python.org/2.7/library/array.html
cmath	https://docs.python.org/2.7/library/cmath.html
collections	https://docs.python.org/2.7/library/collections.html
copy	https://docs.python.org/2.7/library/copy.html
datetime	https://docs.python.org/2.7/library/datetime.html
dateutil	https://pypi.python.org/pypi/dateutils/0.6.6
functools	https://docs.python.org/2.7/library/functools.html
heapq	https://docs.python.org/2.7/library/heapq.html
itertools	https://docs.python.org/2.7/library/itertools.html
json	https://docs.python.org/2.7/library/json.html
math	https://docs.python.org/2.7/library/math.html
operator	https://docs.python.org/2.7/library/operator.html
pytz	https://pypi.python.org/pypi/pytz/2015.2
random	https://docs.python.org/2.7/library/random.html
re	https://docs.python.org/2.7/library/re.html
string	https://docs.python.org/2.7/library/string.html
time	https://docs.python.org/2.7/library/time.html
xml	https://docs.python.org/2.7/library/xml.html

第三方库

我们支持以下Python第三方库，您可以通过import的方式进行引入:
在研究模拟中, 您可以运行 `!pip list` 来查看所有安装的第三库和版本.

回测模块：

> **anyjson, arch, graphviz, Lasagne, numpy, pandas, pybrain, scipy, seaborn, sklearn, statsmodels, talib, tushare, Theanom, requests, hmm, hmmlearn, pywt**

研究模块：

> **anyjson, arch, cvxopt, graphviz, gensim, jieba, Lasagne, matplotlib, mpl_toolkits, numpy, pandas, pybrain, pymc, pillow, scipy, seaborn, sklearn, statsmodels, tables, talib, hmm, hmmlearn, tushare, theano, Lasagne, requests, pywt, zipline, xlrd, xlwt, openpyxl, snownlp**

主要模块介绍：

模块名称	版本	简介	网址
arch	3.1	Arch模型的lib库	https://pypi.python.org/pypi/arch
cvxopt	1.1.8	cvxopt是一个最优化计算包，进行线性规划、二次规划、半正定规划等的计算	https://pypi.python.org/pypi/cvxopt/1.1.8
gensim	0.12.2	gensim用于计算文本相似度，依赖NumPy和SciPy这两大Python科学计算工具包	http://radimrehurek.com/gensim/tutorial.html
jieba	0.37	jieba是一个中文分词组件	https://pypi.python.org/pypi/jieba
matplotlib	1.4.3	matplotlib可能是Python 2D绘图领域使用最广泛的库。它能让使用者很轻松地将数据图形化，并且提供多样化的输出格式	http://matplotlib.org/contents.html
mpl_toolkits	1.4.3	mpl_toolkits是一个Phyton 3D绘图领域函数库	http://matplotlib.org/mpl_toolkits/index.html
NumPy	1.9.3	NumPy系统是Python的一种开源的数值计算扩展。NumPy（Numeric Python）提供了许多高级的数值编程工具，如：矩阵数据类型、矢量处理，以及精密的运算库。专为进行严格的数字处理而产生	http://www.numpy.org/
pandas	0.16.2	Python Data Analysis Library 或 pandas 是基于NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。pandas提供了大量能使我们快速便捷地处理数据	http://pandas.pydata.org/pandas-docs/version/0.16.2/

		的函数和方法	
pybrain	0.3	pybrain一个开源的Python神经网络库	http://pybrain.org/docs/
pymc	2.3.6	pymc是机器学习中一个图模型的Python库	https://pypi.python.org/pypi/pymc/
SciPy	0.15.1	SciPy是一款方便、易于使用、专为科学和工程设计的Python工具包。它包括统计，优化，整合，线性代数模块，傅里叶变换，信号和图像处理,常微分方程求解器等等	http://www.scipy.org/
seaborn	0.6.0	该模块是一个统计数据可视化库	http://web.stanford.edu/~mwaskom/software/seaborn/
sklearn	0.18	Scikit-Learn是基于python的机器学习模块，基于BSD开源许可证。scikit-learn的基本功能主要被分为六个部分，分类，回归，聚类，数据降维，模型选择，数据预处理。Scikit-Learn中的机器学习模型非常丰富，包括SVM，决策树，GBDT，KNN等等，可以根据问题的类型选择合适的模型	http://scikit-learn.org/stable/
Statsmodels	0.6.1	Statismodels是一个Python包，提供一些互补scipy统计计算的功能，包括描述性统计和统计模型估计和推断	http://statsmodels.sourceforge.net/
PyTables	3.2.2	PyTables提供了一些用于结构化数组的高级查询功能，而且还能添加列索引以提升查询速度，这跟关系型数据库所提供的表索引功能非常类似	http://www.pytables.org/usersguide/tutorials.html
TALib	0.4.9	TALib是一个处理金融数据和技术分析的开	http://mrjbq7.github.io/ta-lib/funcs.html

		放代码库	
hmmlearn	0.2.0	是在python上实现隐马可夫模型的一个组件包	https://github.com/hmmlearn/hmmlearn
Theano	0.8.1	Pyhton深度学习库	http://deeplearning.net/software/theano/
Lasagne	0.1	Pyhton深度学习库	http://lasagne.readthedocs.org/en/latest/
requests	2.7.0	网络访问模块	http://docs.python-requests.org/en/v2.7.0/
pywt	0.4.0	小波工具箱	http://pywavelets.readthedocs.io/en/v0.4.0/
Zipline	0.9.0	开源的交易算法库，目前作为Quantopian的回溯检验引擎	https://github.com/quantopian/zipline
xlrd	1.0.0	Python语言中，读取Excel的扩展工具	https://pypi.python.org/pypi/xlrd/
xlwt	1.1.2	Python语言中，写入Excel文件的扩展工具	https://pypi.python.org/pypi/xlwt/
openpyxl	2.4.0	Openpyxl是一个python读写Excel 2010文件的库	http://openpyxl.readthedocs.io/en/default/
snownlp	0.12.3	处理中文文本的Python库	https://pypi.python.org/pypi/snownlp/

策略示例

以下为常见的量化基础算法示例，您也可以访问社区寻找更多量化策略。

双均线策略

最基础的量化策略之一，当五日均线高于十日均线时买入，当五日均线低于十日均线时卖出。

```
# 初始化函数，设定要操作的股票、基准等等
def initialize(context):
    # 定义一个全局变量， 保存要操作的股票
    # 000001(股票:平安银行)
    g.security = '000001.XSHE'
    # 设定沪深300作为基准
    set_benchmark('000300.XSHG')

# 每个单位时间(如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次)调用一次
def handle_data(context, data):
    security = g.security
    # 获取股票的收盘价
    close_data = attribute_history(security, 10, '1d', ['close'],df=False)
    # 取得过去五天的平均价格
    ma5 = close_data['close'][-5:].mean()
    # 取得过去10天的平均价格
    ma10 = close_data['close'].mean()
    # 取得当前的现金
    cash = context.portfolio.cash

    # 如果当前有余额，并且五日均线大于十日均线
    if ma5 > ma10:
        # 用所有 cash 买入股票
        order_value(security, cash)
        # 记录这次买入
        log.info("Buying %s" % (security))

    # 如果五日均线小于十日均线，并且目前有头寸
    elif ma5 < ma10 and context.portfolio.positions[security].closeable_amoun
t> 0:
        # 全部卖出
        order_target(security, 0)
        # 记录这次卖出
        log.info("Selling %s" % (security))

    # 绘制五日均线价格
    record(ma5=ma5)
    # 绘制十日均线价格
    record(ma10=ma10)
```

均线回归策略

当价格低于5日均线平均价格*0.95时买入，当价格高于5日平均价格*1.05时卖出。

```
# 初始化函数，设定要操作的股票、基准等等
def initialize(context):
    # 定义一个全局变量, 保存要操作的股票
    # 000001(股票:平安银行)
    g.security = '000001.XSHE'
    # 设定沪深300作为基准
    set_benchmark('000300.XSHG')

# 每个单位时间(如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次)调用一次
def handle_data(context, data):
    security = g.security
    # 获取股票的收盘价
    close_data = attribute_history(security, 5, '1d', ['close'])
    # 取得过去五天的平均价格
    MA5 = close_data['close'].mean()
    # 取得上一时间点价格
    current_price = close_data['close'][-1]
    # 取得当前的现金
    cash = context.portfolio.cash

    # 如果上一时间点价格高出五天平均价1%, 则全仓买入
    if current_price > 1.05*MA5:
        # 用所有 cash 买入股票
        order_value(security, cash)
        # 记录这次买入
        log.info("Buying %s" % (security))
    # 如果上一时间点价格低于五天平均价, 则空仓卖出
    elif current_price < 0.95*MA5 and context.portfolio.positions[security].cl
oseable_amount > 0:
        # 卖出所有股票,使这只股票的最终持有量为0
        order_target(security, 0)
        # 记录这次卖出
        log.info("Selling %s" % (security))
    # 画出上一时间点价格
    record(stock_price=current_price)
```

多股票持仓示例

这是一个较简单的多股票操作示例，当价格高于三天平均价*1.005则买入100股，当价格小于三天平均价*0.995则卖出。

```
def initialize(context):
    # 初始化此策略
    # 设置我们要操作的股票池
    g.stocks = ['000001.XSHE','000002.XSHE','000004.XSHE','000005.XSHE']
    # 设定沪深300作为基准
    set_benchmark('000300.XSHG')

# 每个单位时间(如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次)调用一次
def handle_data(context, data):
    # 循环每只股票
    for security in g.stocks:
        # 得到股票之前3天的平均价
        vwap = data[security].vwap(3)
        # 得到上一时间点股票平均价
        price = data[security].close
        # 得到当前资金余额
        cash = context.portfolio.cash

        # 如果上一时间点价格小于三天平均价*0.995，并且持有该股票，卖出
        if price < vwap * 0.995 and context.portfolio.positions[security].clos
eable_amount > 0:
            # 下入卖出单
            order(security,-100)
            # 记录这次卖出
            log.info("Selling %s" % (security))
        # 如果上一时间点价格大于三天平均价*1.005，并且有现金余额，买入
        elif price > vwap * 1.005 and cash > 0:
            # 下入买入单
            order(security,100)
            # 记录这次买入
            log.info("Buying %s" % (security))
```

多股票追涨策略

当股票在当日收盘30分钟内涨幅到达9.5%~9.9%时间段的时候，我们进行买入，在第二天开盘卖出。注意：**请按照分钟进行回测该策略。**

```python
# 初始化程序，整个回测只运行一次
def initialize(context):

    # 每天买入股票数量
    g.daily_buy_count  = 5

    # 设置我们要操作的股票池，这里我们操作多只股票，下列股票选自计算机信息技术相关板块
    g.stocks = get_industry_stocks('I64') + get_industry_stocks('I65')

    # 防止板块之间重复包含某只股票，排除掉重复的，g.stocks 现在是一个集合(set)
    g.stocks = set(g.stocks)

    # 让每天早上开盘时执行 morning_sell_all
    run_daily(morning_sell_all, 'open')

def morning_sell_all(context):
    # 将目前所有的股票卖出
    for security in context.portfolio.positions:
        # 全部卖出
        order_target(security, 0)
        # 记录这次卖出
        log.info("Selling %s" % (security))

def before_trading_start(context):
    # 今天已经买入的股票
    g.today_bought_stocks = set()

    # 得到所有股票昨日收盘价，每天只需要取一次，所以放在 before_trading_start 中
    g.last_df = history(1,'1d','close',g.stocks)

# 在每分钟的第一秒运行，data 是上一分钟的切片数据
def handle_data(context, data):

    # 判断是否在当日最后的2小时，我们只追涨最后2小时满足追涨条件的股票
    if context.current_dt.hour < 13:
        return

    # 每天只买这么多个
    if len(g.today_bought_stocks) >= g.daily_buy_count:
        return

    # 只遍历今天还没有买入的股票
    for security in (g.stocks - g.today_bought_stocks):

        # 得到当前价格
        price = data[security].close

        # 获取这只股票昨天收盘价
        last_close = g.last_df[security][0]

        # 如果上一时间点价格已经涨了9.5%~9.9%
        # 今天的涨停价格区间大于1元，今天没有买入该支股票
        if price/last_close > 1.095 \
                and price/last_close < 1.099 \
                and data[security].high_limit - last_close >= 1.0:
```

```
            # 得到当前资金余额
            cash = context.portfolio.cash

            # 计算今天还需要买入的股票数量
            need_count = g.daily_buy_count - len(g.today_bought_stocks)

            # 把现金分成几份,
            buy_cash = context.portfolio.cash / need_count

            # 买入这么多现金的股票
            order_value(security, buy_cash)

            # 放入今日已买股票的集合
            g.today_bought_stocks.add(security)

            # 记录这次买入
            log.info("Buying %s" % (security))

            # 买够5个之后就不买了
            if len(g.today_bought_stocks) >= g.daily_buy_count:
                break
```

万圣节效应策略

股市投资中的“万圣节效应”是指在北半球的冬季(11月至4月份)，股市回报通常明显高於夏季(5月至10月份)。这里我们选取了中国蓝筹股，采用10月15日后买入，5月15日后卖出的简单策略进行示例。

```
def initialize(context):
    # 初始化此策略
    # 设置我们要操作的股票池，这里我们选择蓝筹股
    g.stocks = ['000001.XSHE','600000.XSHG','600019.XSHG','600028.XSHG','60003
0.XSHG','600036.XSHG','600519.XSHG','601398.XSHG','601857.XSHG','601988.XSHG']

# 每个单位时间(如果按天回测,则每天调用一次,如果按分钟,则每分钟调用一次)调用一次
def handle_data(context, data):
    # 得到每只股票可以花费的现金，这里我们使用总现金股票数数量
    cash = context.portfolio.cash / len(g.stocks)
    # 获取数据
    hist = history(1,'1d','close',g.stocks)
    # 循环股票池
    for security in g.stocks:
        # 得到当前时间
        today = context.current_dt
        # 得到该股票上一时间点价格
        current_price = hist[security][0]
        # 如果当前为10月且日期大于15号，并且现金大于上一时间点价格，并且当前该股票空仓
        if today.month == 10 and today.day > 15 and cash > current_price and c
ontext.portfolio.positions[security].closeable_amount == 0:
            order_value(security, cash)
            # 记录这次买入
            log.info("Buying %s" % (security))
        # 如果当前为5月且日期大于15号，并且当前有该股票持仓，则卖出
        elif today.month == 5 and today.day > 15 and context.portfolio.positio
ns[security].closeable_amount > 0:
            # 全部卖出
            order_target(security, 0)
            # 记录这次卖出
            log.info("Selling %s" % (security))
```

如果以上内容仍没有解决您的问题，请您通过社区提问的方式告诉我们，谢谢