vn.py 2.0.7源代码深入分析

我这人手欠。看英文资料时会随手翻译，编程序和读程序时都会随手整理文档。本文是学习vn.py源代码时记录并整理的。

原来这个文档叫“分析入门”，只包含本文档的第一部分。当时正在全面调研各类量化交易平台，对vn.py只做初步了解。现在准备进一步学习vn.py，也就有了本文档的第二部分。希望以后还会有第三、四部分。说到调研，我没资格评价哪个好哪个稍差，此处只是汇报我个人的一个情况：在调研了一圈之后，现在又回过头来研究vn.py了。

与vn.py相关的每个小专题在网上都能找到相应介绍，但不成系统。整理这个文档花了挺大功夫，发出来有点儿舍不得，但想想人家源码都能分享，我一破文档还有什么舍不得的呀。希望本文档对大家能有帮助，但还是说一句：真正深入的分析，只有靠自己。

随着分析的深入可以看到，vn.py的编程技术很精妙，程序结构非常清晰。但毕竟是复杂系统，各部分相互交叉，要想一开始就从代码入手，涉及哪个代码文件就把它吃透，会非常困难。本文采用的方法是从功能入手，从多个代码文件中找出与该功能相关的代码，串成该功能的实现方法。用这种方法需要分析者有一定的编程经验，如果您现在编程经验还不是太充分，只需要跟随本文档的线索先分析起来，等对系统有了一定了解，脱离文档自己独立分析也不会有问题。

本文档不是教学稿。本人水平有限，初学Python，初学vn.py，文档中肯定有很多问题和错误，敬请批评指正。

第一部分 初步分析

学习vn.py有一个星期了，时间上绝对是新手。但以前除了自己编程，还有挺多分析别人程序的经历，包括SQLite、MySQL、PostgreSQL、Weka、Tanagra等，有了这些经验，分析vn.py时可以少走不少弯路。

学习vn.py这么好的程序，一个星期肯定不够，我也会继续学习，也会继续整理文档。但把这个时间点的新手文档发出来，或许对同辈新手更有帮助，咱们可能正走着同样的路，而且可能就是肩并着肩。

不同人分析源码的方法不同，有的人是直奔目标模块深入分析，而我喜欢先搞清代码的整体结构。本部分主要介绍以数据结构为基础的各模块之间的调用关系。如果您不喜欢这种方式，就不用再往下看了。

本部分是后续分析工作的基础。从vn.py源码的下载、安装、程序主函数、主窗口入手，先跟您一起把源码运行起来。再聚焦于vn.py的一个重要应用“CTA回测”，从各个层次上分析其源码，包括相关的数据库操作、多线程机制、事件引擎机制等，把这个应用从顶到底，再从底到顶贯通起来。明白了这些内容，再分析其它的功能就会事半功倍。

本部分目标：理清vn.py程序的整体架构。

# 准备工作

本部分只是简述，每项内容在网上都有很多帖子可以参考。

## 源码下载

vn.py的官网是<http://www.vnpy.com/>。点击主页上的“获取源码”，转到GitHub上去下载。跳转到的页面是vn.py的介绍（vn.py在GitHub上的主页），其中讲到安装时，有超链跳转到下载页面，如下图。



当前的最新版本是vn.py 2.0.7，我是Windows环境，下载Zip文件，如下图。



吐槽一下，GitHub的下载速度慢点不要紧，讨厌的是经常中断，中断了就得重下。我下了一天多，下了无数次，最终也没有下载成功。后来在码云上下载，速度飞快，链接是<https://gitee.com/mirrors/vn-py>。

## 源码安装

安装vn.py源码之前，您需要先安装Python环境，这个环境相信您已经有了。vn.py 2.0.7要求Python版本在3.7以上。

将下载的Zip文件解压到一个文件夹，我是解压到

D:\vnpy207

后面的介绍都会以此为基础进行。

执行D:\vnpy207中的install.bat批处理文件进行安装，自动下载安装需要的包。

我电脑中装的是Anaconda3，包比较全，安装过程很快成功。感觉vn.py适应性挺强，我试了在纯的Python环境下安装，需要下载的包很多，需要很长时间，有些还出错，没有成功。后来出差没有再试，相信对于高手/老手来说这不是问题。

## vn.py的启动

在D:\vnpy207中创建一个新的Python文件run.py，内容如下：

from vnpy.event import EventEngine

from vnpy.trader.engine import MainEngine

from vnpy.trader.ui import MainWindow, create\_qapp

from vnpy.gateway.ctp import CtpGateway

from vnpy.app.cta\_strategy import CtaStrategyApp

from vnpy.app.cta\_backtester import CtaBacktesterApp

def main（）:

"""Start VN Trader"""

qapp = create\_qapp（）

event\_engine = EventEngine（）

main\_engine = MainEngine（event\_engine）

main\_engine.add\_gateway（CtpGateway）

main\_engine.add\_app（CtaStrategyApp）

main\_engine.add\_app（CtaBacktesterApp）

main\_window = MainWindow（main\_engine, event\_engine）

main\_window.showMaximized（）

qapp.exec（）

if \_\_name\_\_ == "\_\_main\_\_":

main（）

打开CMD，转到D:\vnpy207目录，运行下面命令：

python run.py

就可以启动vn.py。

使用CMD有多种方法，如：

1-使用Windows的“运行”功能，运行CMD。

2-使用Windows的资源管理器，在D:\vnpy207目录下，按住Shift键，点击鼠标右键，在弹出菜单中选择“在此处打开PowerShell窗口”。

如果上述方法都不行，比如可能出现以下出错信息：

Traceback (most recent call last):

File "run.py", line 5, in <module>

from vnpy.gateway.da import DaGateway

File "D:\vnpy207\vnpy\gateway\da\\_\_init\_\_.py", line 1, in <module>

from .da\_gateway import DaGateway

File "D:\vnpy207\vnpy\gateway\da\da\_gateway.py", line 7, in <module>

import wmi

File "D:\vnstudio\lib\site-packages\wmi.py", line 88, in <module>

from win32com.client import GetObject, Dispatch

File "D:\vnstudio\lib\site-packages\win32com\\_\_init\_\_.py", line 5, in <module>

import win32api, sys, os

ImportError: DLL load failed: 找不到指定的程序。

此时可以使用Anaconda3的Anaconda Prompt，应该可以成功。

## 使用IDE

要分析原代码，最好还是在IDE中进行。使用IDE，不仅方便查看源码，还可以在其中调试执行，有利于加快对源码的理解。

我用PyCharm，把D:\vnpy207目录作为一个项目打开，执行其中的run.py，可正常执行。这样就可以对代码进行修改和调试。我还测试了，用eric6也可以，相信用其它的IDE也一样。

## vn.py的使用

得先会使用，才可能理解人家的程序。

按我初步的理解，vn.py包括三大功能模块：交易（VN Trader）、CTA策略和CTA回测。其中VN Trader是主程序，CTA策略和CTA回测以非模式窗口的形式执行。后来发现，vn.py其实还有挺多功能，只是没有加载，不过作为初步分析，上面三个模块就够了。

VN Trader界面比较一般，典型的CTP示例程序界面，没有分时图和K线图等直观界面。

回测结果形式倒是比较直观，有图表有K线图，但K线图比较难看。

从外观上看，vn.py界面单调，功能比较少。但为什么人家这么有名呢？估计是这比较少的功能做得比较透。

vn.py的使用方法不再详述。为什么没一本书（或文档）纯讲vn.py的使用？都是把编程跟使用混在一起讲。假设vn.py作为平台已经很好了，那用户可能并不需要了解平台的程序架构，更不需要去改源代码，只需要告诉人家平台的使用方法以及策略怎么编怎么测怎么实盘就行了。

类似的提问本文档后面还有，没有质问谁对谁不尊重的意思，纯粹是想记录下来，以后自己回答，或者请大师们帮助解答。

## 需要的知识

学习vn.py的必备知识很多文章都有讲，我目前大多没有用到，随着分析的深入应该能用到吧。本部分需要用到：

* Python基本知识，这个相信您比我强。
* PyQt：vn.py的GUI界面是用PyQt做的。
* 数据库及ORM的概念：vn.py可以接多种数据库，为了简化编程，使用了一个简单小巧的Python ORM（Object Relational Mapping）——peewee。

# VN Trader初步

有了前面的基础，就可以直接从VN Trader开始分析vn.py的源代码了。

本章是对VN Trader的初步分析，服务于“理清vn.py整体架构”的目标。我们将在第二部分对VN Trader进行深入分析。

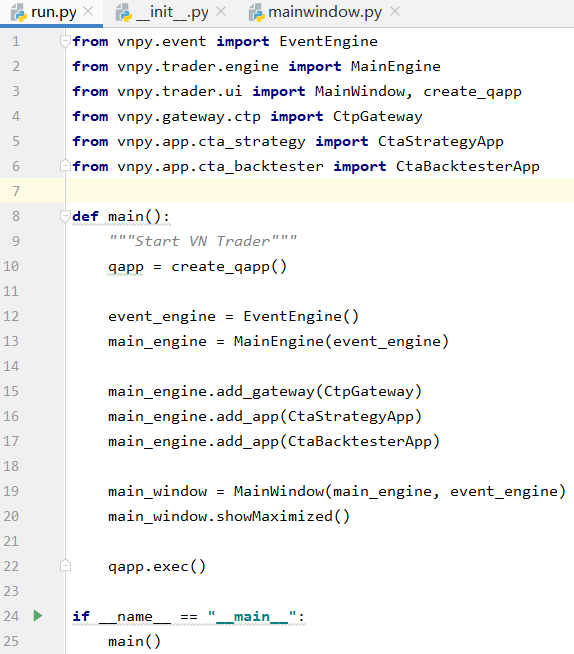
## 程序主函数

程序执行的入口也是分析的入口。

本文不用顺序阅读，文中说“参”哪儿，就可以先看哪儿。

假设vn.py源代码的安装目录为D:\vnpy207。

程序的执行入口（主函数）在D:\vnpy207\run.py中，其内容如下：



对各行程序进行说明。

10：创建一个应用程序对象，参“创建应用程序”一节。

12：创建一个事件引擎。事件引擎的内容参“事件引擎 – 事件类”一节。

13：用上一行创建的事件引擎作参数，创建主引擎。主引擎的内容参“主引擎”一节。

15：将CTP底层接口加入到主引擎。

16：将“CTA策略”上层应用加入到主引擎。

17：将“CTA回测”上层应用加入到主引擎。

19：创建主窗口。

20：主窗口最大化。

22：执行应用程序。

## 主引擎

从程序主函数一节可以看出，在主函数不多的几行代码中，大多都在操作一个叫main\_engine的东西，它是MainEngine类的一个实例，我们首先来介绍与之相关的内容。

vn.py的体系结构参“vn.py体系结构”一节，在整个体系结构中，处于中层的主引擎是整个系统的核心。

MainEngine主引擎在D:\vnpy207\vnpy\trader目录下的engine.py文件中定义。

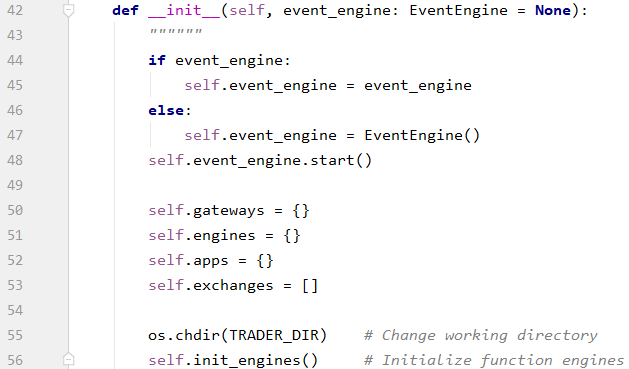
有的资料中也称主引擎为“核心引擎”，感觉还是叫主引擎合适。

作为老程序员，我也不记得“引擎”这个术语是怎么深化的啦，反正现在什么都叫引擎。按说引擎应该产生动力呀，但vn.py的主引擎好像只是一个容器，装了一大堆互不相干的东西。但既然人家叫引擎，咱也就叫跟着引擎吧。

### 初始化函数

通过初始化函数可以看出主引擎是一个容器，把对程序运行至关重要的一些东西容纳进来，包括：

* 一个事件引擎
* 一个底层接口集合
* 一个功能引擎集合
* 一个上层应用集体
* 一个交易所列表



可以看出，主引擎的功能就是将各类引擎组织在一起。

44~47：如果参数传来了一个事件引擎，则将其赋给成员变量event\_engine；如果没有参数，则为该成员变量创建一个事件引擎。

48：启动事件引擎。事件引擎是主引擎乃至整个vn.py的核心组件，详细说明参“事件引擎”一章。

50：底层接口集合。

51：功能引擎集合。

52：上层应用集体。

53：交易所列表，向底层接口集合增加接口时，同时向本列表增加该接口支持的交易所，参“增加底层接口”一节。

55：将工作路径改到C:\Users\admin\.vntrader。工作路径TRADER\_DIR在同目录的utility.py文件中定义，与工作路径有关的内容参“工作目录”一节。

56：调用init\_engines()函数，初始化功能引擎。

老版本中还会把负责数据的数据引擎也放在主引擎中，现版本好像把数据操作局限在了底层，这样更合理。

下面介绍主引擎的几个成员函数。

### 初始化功能引擎

def init\_engines(self):

self.add\_engine(LogEngine)

self.add\_engine(OmsEngine)

self.add\_engine(EmailEngine)

将日志引擎、Oms引擎和邮件引擎加入到功能引擎集合中。

这三个引擎与主引擎在同一个文件中定义，它们都是以BaseEngine为基类，而在BaseEngine的注释中称其为功能引擎，所以我们将这三个引擎统称为功能引擎。

注：主引擎的功能引擎集合中不只这三个引擎，还包括所有上层应用引擎（参“增加上层应用（函数）”一节）。所以功能引擎这个词是否准确，再研究。

### 增加功能引擎

本函数被init\_engines(self)函数所调用。

def add\_engine(self, engine\_class: Any):

传入的参数是类名称。

engine = engine\_class(self, self.event\_engine)

根据类名称创建对象。注意，第一个参数是主引擎本身，第二个参数是事件引擎。这两个参数将传递给相应功能引擎类的构造函数。

self.engines[engine.engine\_name] = engine

将新创建的对象加入到功能引擎集合。

return engine

### 增加底层接口

def add\_gateway(self, gateway\_class: Type[BaseGateway]):

参数是底层接口类名称。

gateway = gateway\_class(self.event\_engine)

根据类名称，创建一个底层接口类实例。

self.gateways[gateway.gateway\_name] = gateway

加入到主引擎的网关集合中。

# Add gateway supported exchanges into engine

for exchange in gateway.exchanges:

if exchange not in self.exchanges:

self.exchanges.append(exchange)

将本底层接口支持的交易所加入到交易所列表中。

return gateway

主引擎的底层接口集合和交易所列表都通过本函数建立。

### 增加上层应用

def add\_app(self, app\_class: Type[BaseApp]):

app = app\_class()

创建应用对象。

self.apps[app.app\_name] = app

加入到上层应用集合。

engine = self.add\_engine(app.engine\_class)

注意，此处会将上层应用对应的引擎类加入到功能引擎集合中。由此可见，功能引擎集合中不只前述init\_engines()函数增加的几个引擎，还包括各类上层应用引擎等。

return engine

## 增加上层应用

从“程序主函数”一节可以看出，程序中只加入了两个上层应用。执行程序，在“功能”菜单下面只有CTA策略和CTA回测两个菜单项。

如果想再增加其它的上层应用，假设要把“CSV载入”（此功能我们后面要用）加入到“功能”菜单，可以在run.py中增加两行：

from vnpy.app.csv\_loader import CsvLoaderApp

……

main\_engine.add\_app(CsvLoaderApp)

除CSV载入外，还有多个现成的上层应用可以增加，可参考D:\vnpy207\examples\vn\_trader目录下的run.py文件。

## 主界面

### 创建应用程序

create\_qapp()函数在D:\vnpy207\vnpy\trader\ui的\_\_init\_\_.py中定义。

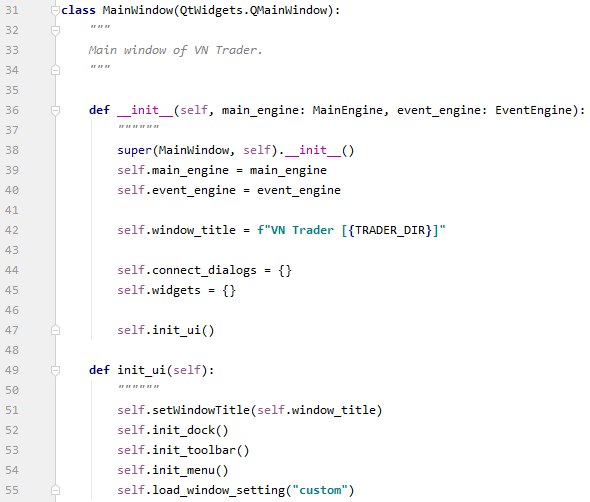
该\_\_init\_\_.py作用如下：

1-定义本系统的异常弹出窗口。当系统捕捉到异常时，用此窗口弹出提示信息，界面更友好。

2-定义create\_qapp()函数。create\_qapp()的功能是创建一个应用程序对象，设置其图标、字体等。

### 主窗口

主窗口在D:\vnpy207\vnpy\trader\ui的mainwindow.py中定义。



39~40：记住主引擎和事件引擎。这两个引擎作为参数在程序主函数中创建主容器时传入。

42：将主窗口标题改为程序名+工作路径，TRADER\_DIR前文有说明。

47：类的构造函数主要是调用init\_ui()函数对窗口进行初始化。

53：创建工具栏。注意，只创建工具栏对象，并没有在其上增加按钮。增加按钮的工作在init\_menu()中完成。所以，init\_toolbar()一定要在init\_menu()的前面。

本文件中是典型的PyQt窗体代码，略。

### 基本窗口控件

程序主窗口包含多个组成部分，都在一个文件中定义难免零乱。这些组成部分被定义成窗口控件，在D:\vnpy207\vnpy\trader\ui目录下的widget.py中定义。

略。

## 工作目录

程序执行后，会将工作目录切换到C:\Users\admin（其中的admin是操作系统用户名，在您的系统中可能有所不同），称为vn.py的工作目录。vn.py运行产生的数据存放在该目录下的.vntrader目录中，包括：

* connect\_ctp.json：保存CTP连接信息，在“连接CTP”功能中使用。
* vt\_setting.json：存放全局配置参数，在“全局配置”功能中使用。
* database.db：保存行情数据的SQLite3文件。相关的功能包括“CSV载入”等。
* cta\_strategy\_data.json：现在为空，还不知是何作用。
* cta\_strategy\_setting.json：现在为空，还不知是何作用。
* 等等。

# 数据库操作

使用和分析vn.py都离不开行情数据。vn.py默认从ricequant（米筐）下载行情数据，然后存储到数据库中。我没有米筐帐户，所以先研究从CSV文件中加载数据到数据库。

把行情数据放到数据库中，在性能上也许不是好方案，放在文件中可能更好。行情的Meta数据及其它系统数据倒是可以放在数据库中。但既然vn.py现在这么实现，就先这么学习吧。

## vn.py支持的数据库

vn.py将行情数据保存到数据库中。vn.py既可支持SQL数据库也可支持NoSQL数据库，包括：

* sql
  + sqlite
  + mysql
  + postgresql
* nosql
  + mongodb

vn.py默认使用sqlite，下面就以sqlite为例介绍vn.py的数据库操作。

对于sql数据库，vn.py使用peewee。peewee是一个简单小巧的Python ORM，它非常容易学习，并且使用起来很直观。关于ORM的概念及peewee，本文档不介绍。

## 数据库结构

用SQLite的可视化工具，如SQLiteStudio，打开C:\Users\admin\.vntrader目录下的database.db。

可以看到库中有两个表：dbbardata和dbtickdata。

dbbardata存放K线数据，结构如下：

CREATE TABLE dbbardata (

id INTEGER NOT NULL

PRIMARY KEY,

symbol VARCHAR (255) NOT NULL,

exchange VARCHAR (255) NOT NULL,

datetime DATETIME NOT NULL,

interval VARCHAR (255) NOT NULL,

volume REAL NOT NULL,

open\_interest REAL NOT NULL,

open\_price REAL NOT NULL,

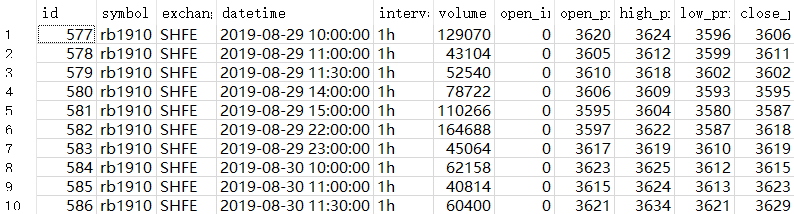
high\_price REAL NOT NULL,

low\_price REAL NOT NULL,

close\_price REAL NOT NULL

);

在symbol,exchange,interval,datetime这四个字段上建立了索引，其部分数据如下图。



dbtickdata存放Tick数据，结构如下：

CREATE TABLE dbtickdata (

id INTEGER NOT NULL

PRIMARY KEY,

symbol VARCHAR (255) NOT NULL,

exchange VARCHAR (255) NOT NULL,

datetime DATETIME NOT NULL,

name VARCHAR (255) NOT NULL,

volume REAL NOT NULL,

open\_interest REAL NOT NULL,

last\_price REAL NOT NULL,

last\_volume REAL NOT NULL,

limit\_up REAL NOT NULL,

limit\_down REAL NOT NULL,

open\_price REAL NOT NULL,

high\_price REAL NOT NULL,

low\_price REAL NOT NULL,

pre\_close REAL NOT NULL,

bid\_price\_1 REAL NOT NULL,

bid\_price\_2 REAL,

bid\_price\_3 REAL,

bid\_price\_4 REAL,

bid\_price\_5 REAL,

ask\_price\_1 REAL NOT NULL,

ask\_price\_2 REAL,

ask\_price\_3 REAL,

ask\_price\_4 REAL,

ask\_price\_5 REAL,

bid\_volume\_1 REAL NOT NULL,

bid\_volume\_2 REAL,

bid\_volume\_3 REAL,

bid\_volume\_4 REAL,

bid\_volume\_5 REAL,

ask\_volume\_1 REAL NOT NULL,

ask\_volume\_2 REAL,

ask\_volume\_3 REAL,

ask\_volume\_4 REAL,

ask\_volume\_5 REAL

);

在symbol,exchange,datetime这三个字段上建立了复合索引。

上述结构可与CTP的Tick数据结构做一下对比。

///深度行情

struct CThostFtdcDepthMarketDataField

{

///交易日

TThostFtdcDateType TradingDay;

///合约代码

TThostFtdcInstrumentIDType InstrumentID;

///交易所代码

TThostFtdcExchangeIDType ExchangeID;

///合约在交易所的代码

TThostFtdcExchangeInstIDType ExchangeInstID;

///最新价

TThostFtdcPriceType LastPrice;

///上次结算价

TThostFtdcPriceType PreSettlementPrice;

///昨收盘

TThostFtdcPriceType PreClosePrice;

///昨持仓量

TThostFtdcLargeVolumeType PreOpenInterest;

///今开盘

TThostFtdcPriceType OpenPrice;

///最高价

TThostFtdcPriceType HighestPrice;

///最低价

TThostFtdcPriceType LowestPrice;

///数量

TThostFtdcVolumeType Volume;

///成交金额

TThostFtdcMoneyType Turnover;

///持仓量

TThostFtdcLargeVolumeType OpenInterest;

///今收盘

TThostFtdcPriceType ClosePrice;

///本次结算价

TThostFtdcPriceType SettlementPrice;

///涨停板价

TThostFtdcPriceType UpperLimitPrice;

///跌停板价

TThostFtdcPriceType LowerLimitPrice;

///昨虚实度

TThostFtdcRatioType PreDelta;

///今虚实度

TThostFtdcRatioType CurrDelta;

///最后修改时间

TThostFtdcTimeType UpdateTime;

///最后修改毫秒

TThostFtdcMillisecType UpdateMillisec;

///申买价一

TThostFtdcPriceType BidPrice1;

///申买量一

TThostFtdcVolumeType BidVolume1;

///申卖价一

TThostFtdcPriceType AskPrice1;

///申卖量一

TThostFtdcVolumeType AskVolume1;

///申买价二

TThostFtdcPriceType BidPrice2;

///申买量二

TThostFtdcVolumeType BidVolume2;

///申卖价二

TThostFtdcPriceType AskPrice2;

///申卖量二

TThostFtdcVolumeType AskVolume2;

///申买价三

TThostFtdcPriceType BidPrice3;

///申买量三

TThostFtdcVolumeType BidVolume3;

///申卖价三

TThostFtdcPriceType AskPrice3;

///申卖量三

TThostFtdcVolumeType AskVolume3;

///申买价四

TThostFtdcPriceType BidPrice4;

///申买量四

TThostFtdcVolumeType BidVolume4;

///申卖价四

TThostFtdcPriceType AskPrice4;

///申卖量四

TThostFtdcVolumeType AskVolume4;

///申买价五

TThostFtdcPriceType BidPrice5;

///申买量五

TThostFtdcVolumeType BidVolume5;

///申卖价五

TThostFtdcPriceType AskPrice5;

///申卖量五

TThostFtdcVolumeType AskVolume5;

///当日均价

TThostFtdcPriceType AveragePrice;

///业务日期

TThostFtdcDateType ActionDay;

};

## 数据库管理器类

本节涉及与数据库操作有关的两个重要类，但没有深入。其中部分操作在后续“初始化”一节中讨论，目的是把数据库操作的调用关系理清；至于具体的操作，典型的ORM方式，本文略。

系统与数据库操作有关的源程序文件和类及其继承关系如下图所示。



### BaseDatabaseManager抽象类

数据库管理器基类，在D:\vnpy207\vnpy\trader\database下的database.py文件中定义。BaseDatabaseManager是一个抽象类，定义了一些数据库操作的抽象方法，是SQL数据库管理器类和NoSQL数据库管理器类的基类。

略。

### SqlManager类

SQL数据库管理器类SqlManager，在D:\vnpy207\vnpy\trader\database下的database\_sql.py文件中定义，用于操作所有SQL数据库，包括sqlite、mysql和postgresql。

略。

## 初始化

系统启动时，对数据库操作进行初始化。

在D:\vnpy207\vnpy\trader\database目录下的\_\_init\_\_.py文件中的第二个if语句上设断点，发现程序启动时在此中断。

其后

settings = get\_settings("database.")

取得有关数据库的全局配置。

database\_manager: "BaseDatabaseManager" = init(settings=settings)

创建并初始化一个数据库管理器（BaseDatabaseManager对象）。

数据库管理器的初始化函数在同目录的initialize.py文件中，代码如下：

def init(settings: dict) -> BaseDatabaseManager:

driver = Driver(settings["driver"])

if driver is Driver.MONGODB:

return init\_nosql(driver=driver, settings=settings)

else:

return init\_sql(driver=driver, settings=settings)

def init\_sql(driver: Driver, settings: dict):

from .database\_sql import init

keys = {'database', "host", "port", "user", "password"}

settings = {k: v for k, v in settings.items() if k in keys}

\_database\_manager = init(driver, settings)

return \_database\_manager

可见，如果是SQLite数据库，则调用init\_sql()函数。而init\_sql()函数则会调用同目录database\_sql.py文件的init()函数，代码如下：

def init(driver: Driver, settings: dict):

init\_funcs = {

Driver.SQLITE: init\_sqlite,

Driver.MYSQL: init\_mysql,

Driver.POSTGRESQL: init\_postgresql,

}

assert driver in init\_funcs

db = init\_funcs[driver](settings)

bar, tick = init\_models(db, driver)

return SqlManager(bar, tick)

根据数据库类型的不同，调用不同的初始化函数，并返回数据库对象到db。在db中初始化两个表，参“数据库结构”一节。bar和tick是两个库表model，最后返回用这两个model创建的SQL数据库管理器对象（SqlManager对象，SqlManager是BaseDatabaseManager的子类），作为全局的数据库管理器。

初始化SQLite数据库调用下面函数：

def init\_sqlite(settings: dict):

database = settings["database"]

path = str(get\_file\_path(database))

db = SqliteDatabase(path)

return db

最终的db = SqliteDatabase(path)一句，是典型的peewee创建数据库对象的方法。

## 全局的数据库管理器

有些文章说数据库管理器也会加入到主引擎中。

那可能是以前，在2.0.7版中我没看到。在上一节的

database\_manager: "BaseDatabaseManager" = init(settings=settings)

一句中，创建了一个数据库管理器的全局变量。在程序中搜索该变量，没有找到将该管理器加入某个引擎的语句。在使用该变量前，一般都是用下面语句

from vnpy.trader.database import database\_manager

导入该变量，然后直接使用。

估计是数据库操作只在底层内部调用，还有待于继续研究。

## 加载数据

本节讨论如何将数据写入到数据库。

注：本节内容参考了网上文章《【vn.py】量化策略历史回测（基于本地csv数据）》。

### 获取数据

编程从新浪财经获取行情数据，将得到的数据保存为csv格式，以获取rb1910的1h行情为例，代码如下：

from urllib import request

import json

import pandas as pd

def get\_data(id):

url\_60m = 'http://stock2.finance.sina.com.cn/futures/api/json.php/IndexService.getInnerFuturesMiniKLine60m?symbol='

url = url\_60m + id

req = request.Request(url)

rsp = request.urlopen(req)

res = rsp.read()

res\_json = json.loads(res)

bar\_list = []

res\_json.reverse()

for line in res\_json:

bar = {}

bar['Datetime'] = line[0]

bar['Open'] = float(line[1])

bar['High'] = float(line[2])

bar['Low'] = float(line[3])

bar['Close'] = float(line[4])

bar['Volume'] = int(line[5])

bar\_list.append(bar)

df = pd.DataFrame(data=bar\_list)

print(df)

df.to\_csv('d:/data.csv', index=None)

if \_\_name\_\_ == '\_\_main\_\_':

get\_data('rb1910')

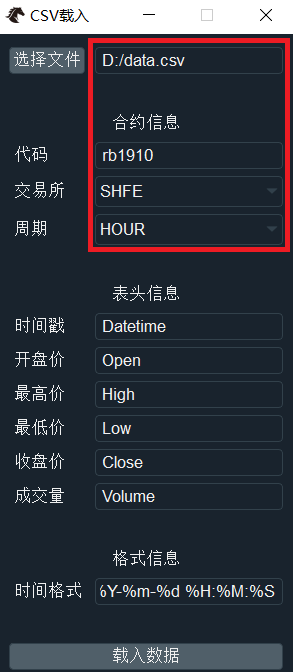
生成的csv文件保存在d:/data.csv中。

注：本例中获取rb1910的数据。如果该合约过期取不到数据，请酌情修改程序，后文中的相关内容也需要修改。

### CSV载入

确保已经按本文“增加上层应用”一节的方法，增加了“CSV载入”功能。

启动VN Trader，执行“CSV载入”功能，按下图进行设置，按“载入数据”按钮，将CSV中的数据加载到数据库。



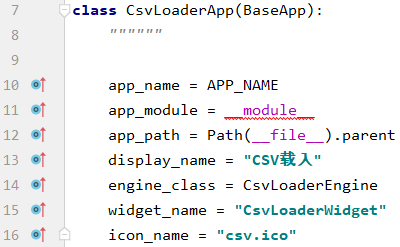
下面通过分析与上述功能相关的代码，研究vn.py加载数据的实现方法。

另，看网上文章对CSV文件格式都有严格要求。根据上述界面判断，由于通过表头定义了各列的含义，应该允许CSV文件格式具有一定的灵活性。这应该是新版本的增强吧，等有机会分析一下。

### CsvLoaderApp类

先看CSV载入应用程序类的定义。这部分与前文“VN Trader”一章结合看，可以搞清VN Trader是如何管理上层应用，如何把上层应用加入到菜单以及如何与相应的操作引擎相关联。

CsvLoaderApp类在在D:\vnpy207\vnpy\app\csv\_loader目录下的\_\_init\_\_.py文件中定义，代码如下：



CsvLoaderApp继承看成BaseApp类，相关说明参第三部分“类结构”一节。

14：指明引擎类。

15：指明窗体类。

VN Trader根据这些信息生成菜单项，并将相关的引擎加入到主引擎当中。具体过程是：程序主函数中执行

main\_engine.add\_app(CsvLoaderApp)

语句，在主引擎的add\_app函数中会调用add\_engine函数，增加功能引擎。

### 窗体代码

“CSV载入”的窗体文件在D:\vnpy207\vnpy\app\csv\_loader\ui目录下的widget.py文件中。

界面的创建是典型的PyQt代码，略。

与数据库操作相关的代码，在初始化函数中：

def \_\_init\_\_(self, main\_engine: MainEngine, event\_engine: EventEngine):

super().\_\_init\_\_()

self.engine = main\_engine.get\_engine(APP\_NAME)

self.init\_ui()

第一个参数是主引擎，参“主引擎”一节。从主引擎中取到CSV载入引擎类，保存到成员变量engine中。

“载入数据”按钮的事件处理函数为load\_data()，完成三项任务：

1-取界面参数。

2-调用engine.load()函数，加载数据。这个load()调用的是D:\vnpy207\vnpy\app\csv\_loader目录下的engine.py中的load()，以下逐级调用文件和数据库操作，此处不再深入。

3-回显CSV载入成功信息。

## 使用数据

本节讨论如何使用数据库中的行情数据。

据说：当执行CTA回测时，先下载数据（默认好像是到米筐），如果下载失败，就到数据库中取数据。

执行CTA回测。

假设rb1910的小时数据已经从CSV文件加载，可按如下界面设置进行回测。



“开始回测”按钮的处理事件在D:\vnpy207\vnpy\app\cta\_backtester\ui目录下的widget.py文件中的start\_backtesting(self)函数中。

在该函数中设断点，发现函数执行结束后没什么反应。可以想象，是“result = self.backtester\_engine.start\_backtesting()”一句发出信号，启动了其它的线程进行处理。

backtester\_engine.start\_backtesting()函数中如何执行回测，将在后续的“CTA回测”一节详细讨论，现在不用管。

在D:\vnpy207\vnpy\app\cta\_strategy目录下的backtesting.py文件中的load\_data(self)函数中设断点，发现数据在函数的while循环中加载。

再深入的调用就是典型的ORM操作，不再继续分析。

# CTA策略初步

对主界面有了基本了解，就可以从一个功能入手，进行专项分析，我选的是CTA回测。

## 策略入门

没找到哪里介绍vn.py的CTA策略如何使用，都是讲怎么编程的。等硬做一遍后，发现还真就是编程。

vn.py没有策略管理模块，没有一个界面让用户自己新增、修改、删除策略。只要用户编好程序并把程序放到特定的目录中，vn.py就自然能发现并管理它。

本节介绍如何从一个已有策略克隆出新策略，在此基础上，就可以继续深入研究如何写自己的策略。

### 创建策略程序文件

新策略可以放在程序工作目录中（参“工作目录”一节），在该目录下创建strategies文件夹（C:\Users\admin\strategies）；也可以放在D:\vnpy207\vnpy\app\cta\_strategy\strategies文件夹内。策略文件命名建议采用下划线模式，如gao\_strategy.py，而策略类命名采用驼峰模式，如GaoStrategy。

在D:\vnpy207\vnpy\app\cta\_strategy\strategies目录下创建一个Python程序文件gao\_strategy.py，并将同目录下double\_ma\_strategy.py的内容完全复制过来。

只需要做两处改动，即把两处类名DoubleMaStrategy替换为GaoStrategy即可。

### 执行

不需要做其它工作，重新启动VN Trader。

在VN Trader的主界面上，无论打开CTA策略还是CTA回测，都能在策略列表中找到新的策略GaoStrategy，回测的方法与原有策略完全相同。

## CTA回测

CTA策略和CTA回测使用同样的策略。同一个策略，在“CTA回测”功能中是用历史数据进行回测，在“CTA策略”功能中则是应用于实盘。

当前还在关注于vn.py的技术实现，不太重视实盘，所以本节主要研究CTA回测功能的实现。

请先按“数据库操作 – 加载数据”一节的方法，将rb1910的小时行情加载到数据库中。

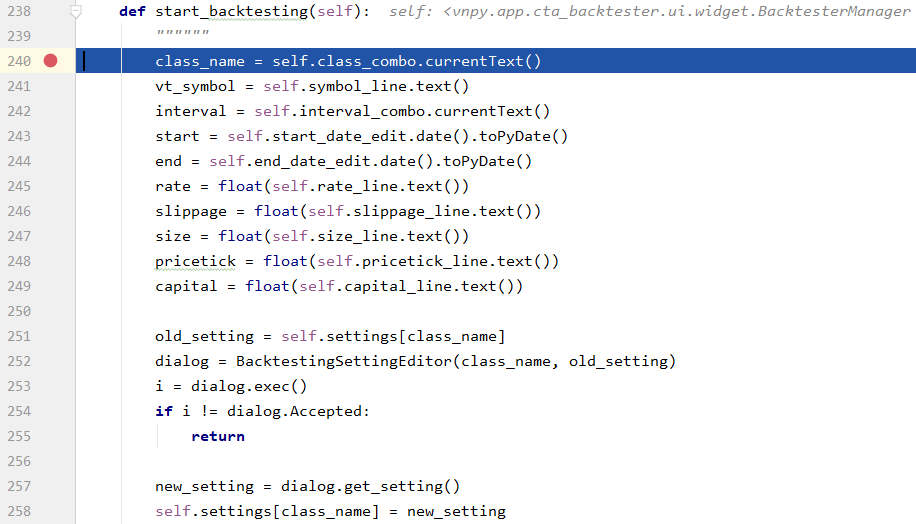
在“CTA回测”功能中按下图设置，按“开始回测”按钮开始回测。



同样是“CTA回测”功能，在“使用数据”一节中也有讨论，那里侧重如何从数据库中取数据，本节侧重如何调用回测引擎执行回测。

### “开始回测”按钮事件处理函数

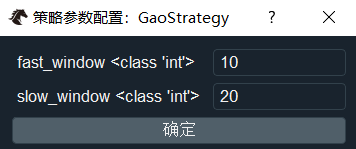
将断点设在按钮处理事件的第一行，如下图。



240~249：取界面上的回测参数。

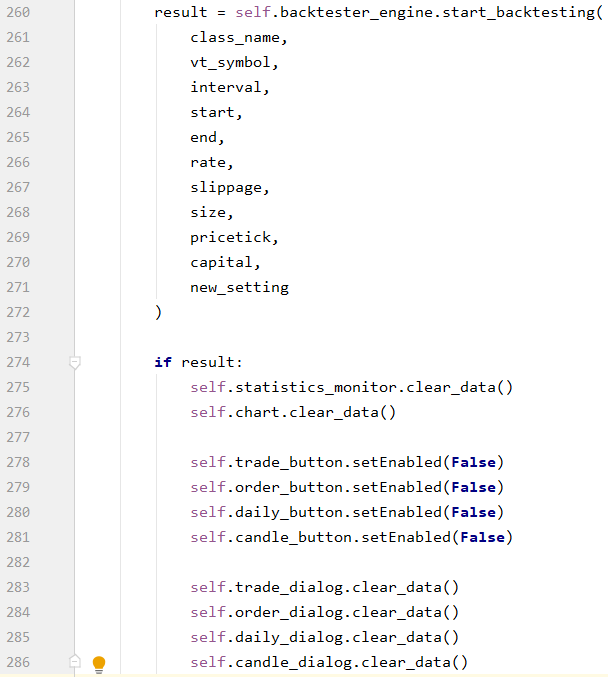
251：取GaoStrategy类的两个策略参数。

252~255：打开如下图的对话框，可对两个参数进行定制。



如果单击对话框右上的叉叉，或者按ESC键退出，则终止回测。如果按“确定”按钮，则继续。

257~258：将改过的策略参数保存下来，下次再执行时默认用新的策略参数。但新策略参数不保存到外部文件，如果程序关闭，则定制的策略参数将丢失。



260：启动回测线程，详见后续“执行回测”一节。注意，返回结果result表示线程启动是否成功，而不是线程的执行结果。

274：如果线程成功启动，继续执行以下工作。

278~281：把“成交记录”到“K线图表”4个按钮改为不可用。在回测期间不允许按这4个按钮。

283~286：清除4个结果对话框中的数据，待重新使用。

就结束了，真正的回测工作在其它的线程中执行。

### 执行回测

在继续本节内容之前，请先看一下后续“多线程”一章的内容。这不是建议，而是必须，否则无法理解本节后续内容。

真正的回测工作在回测线程中执行。BacktesterEngine是回测引擎类，在D:\vnpy207\vnpy\app\cta\_backtester目录下的engine.py中定义，其线程函数如下：

def run\_backtesting(

self,

class\_name: str,

vt\_symbol: str,

interval: str,

start: datetime,

end: datetime,

rate: float,

slippage: float,

size: int,

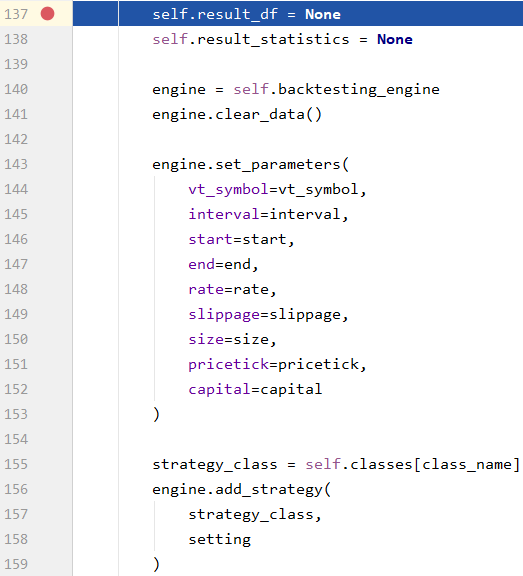
pricetick: float,

capital: int,

setting: dict

):

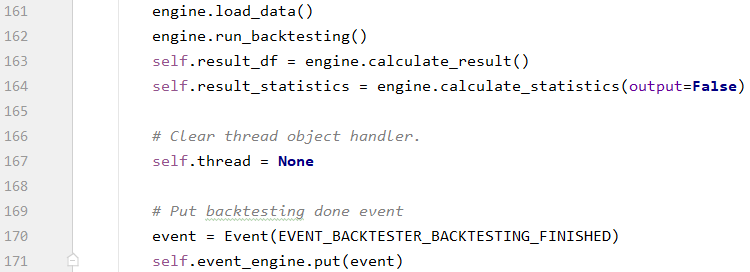
函数参数就是所有的界面配置项和策略参数。



140~153：创建一个回测引擎，并设置参数。

155：取得策略类（这里是GaoStrategy）。

156：将策略类和策略参数加载到回测引擎。



161：加载行情数据，详参“加载数据”一节。执行的结果是从数据库中取rb1910的小时行情。

162：回测引擎执行回测任务。包括策略初始化及回放历史行情数据等，详参“”。（注：还没继续分析，这是后期工作的重点）

163：计算回测的结果。

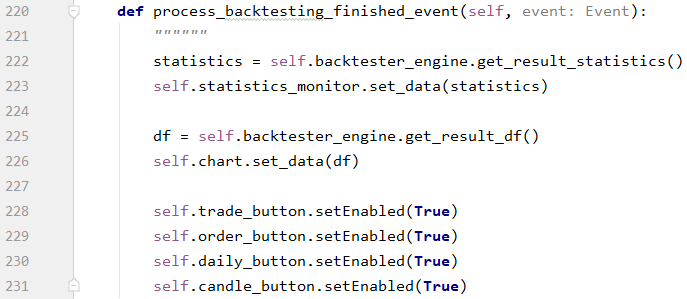
164：对回测结果进行统计计算。

167：将thread置为None，允许新的回测线程执行。参“多线程”一章。

170~171：推送回测结束事件。参“事件引擎”一章。

### 回测结束事件处理

回测结束事件由窗体类的process\_backtesting\_finished\_event()函数进行处理。参“事件引擎”一章。



# 多线程

vn.py采用多线程机制处理费时操作。如回测操作，启动专用的回测线程执行，在执行期间不影响界面响应。

我们以回测线程为例，介绍vn.py的线程机制。

## 类结构

涉及回测的有如下几个类。



BacktesterManager是窗体类，在主线程中使用。

BacktesterEngine是线程类，其线程函数为run\_backtesting()，由本类的start\_backtesting()函数调用。因此，上图两个线程的分界在start\_backtesting()与run\_backtesting()之间。

BacktestingEngine是回测执行类，在回测线程内部使用。

## 执行流程

**一、BacktesterManager窗体类：**

BacktesterManager是窗体类，在D:\vnpy207\vnpy\app\cta\_strategy\ui\widget.py中定义。它有一个成员变量backtester\_engine，指向一个BacktesterEngine类实例。在BacktesterManager的“开始回测”按钮事件处理函数start\_backtesting()中，调用backtester\_engine.start\_backtesting()。

**二、BacktesterEngine回测引擎类：**

BacktesterEngine是回测引擎类，在D:\vnpy207\vnpy\app\cta\_backtester目录下的engine.py中定义。它有一个成员变量backtesting\_engine，指向一个BacktestingEngine类实例。还有一个线程指针（借用指针的概念）tread，该指针平常值为None；当有回测线程执行时，该指针指向回测线程；线程执行结束前，将该指针重新置为None。

当start\_backtesting()函数被调用时，做三项工作：

1-检查tread是否为None。如果不为None，说明有回测线程正在执行，禁止启动新的线程。

2-将run\_backtesting()函数指定为线程，tread指向该线程。

3-启动线程。

在线程函数run\_backtesting()中，执行以下操作：

1-传递策略参数。

2-调用backtesting\_engine的run\_backtesting()函数，执行真正的回测操作。参“CTA回测”一节。

3-回测结束，将tread置为None，允许新的回测线程执行。

4-推送回测结束事件。

其中第3步与线程控制有关。

需要说明的是，这种线程控制方式是不“优雅”的。参本章“存在问题”一节。

**三、BacktestingEngine回测执行类**

BacktestingEngine是回测执行类，在backtesting.py中定义。

成员函数run\_backtesting()执行真正的回测操作，本处不做介绍。

## 存在问题

需要说明的是，这种线程控制方式是不“优雅”的。如果在BacktesterEngine.run\_backtesting()函数的第1、2步处理中发生异常，线程就可能异常中止（虽然没有return语句），tread就无法重新置为None，新的回测线程就无法启动。

简单地使用Python的异常处理语句，将相关代码改为：

try:

engine = self.backtesting\_engine

……

self.result\_statistics = engine.calculate\_statistics(output=False)

except:

self.write\_log("回测过程中发生错误，回测线程已中止")

finally:

# Clear thread object handler.

self.thread = None

就可以解决上述问题。也许有更好的方法，没仔细研究。

另外，搜索了程序代码，好像没有发现finally关键字，我对此表示困惑。

# 事件引擎

## vn.py体系结构

vn.py的体系结构，也称软件架构，有多种说法，都对。此处引用《Python量化交易》（张扬飞著）中的说法，将vn.py的体系结构分为三层。

* 上层应用：GUI图形界面、CTA策略、行情数据记录等。
* 中层引擎：事件引擎、订单路由等。
* 底层接口：CTP网关、LTS网关、IB网关等底层接口。

在中层，通过一个叫主引擎（MainEngine类）的结构，把所有这些引擎组织起来，便于上层GUI图形界面调用，主引擎是整个系统的核心。

事件引擎是主引擎乃至整个vn.py的核心组件，也是大多数交易系统或回测引擎、甚至大多数交互程序的设计基础。本章讨论事件引擎的相关内容。

## 事件引擎的作用

对于编程新人来说，“事件引擎”这个词可能比较抽象。具体地说，对于像vn.py这样的应用，它需要监听一些事件，当某些事件发生的时候，要分配相对应的方法进行处理，完成这个过程的机制就叫事件引擎。概括地说，一个事件引擎应该具有以下功能：

* 有一个容器，暂存待处理的事件。事件发生时，可以先放到这个容器中等待处理。
* 将事件类型与处理方法关联。一类事件可以由多个方法来处理，一个方法也可以处理多类事件。
* 监听容器中是否有事件，如果有，就取出并调用预定的方法来处理。

作为vn.py的核心组件，事件引擎完成上述功能，实现方法是：

* 维护一个事件队列（作为容器）。vn.py为所有待处理的事件创建一个事件对象，存储在该队列中等待处理。队列的特点是先进先出（FIFO），先发生的事件先处理。
* 为不同类型的事件注册处理函数。
* 创建守护线程，对队列进行监听和处理。

明确了事件引擎是干什么的，下面来看具体的代码。

## 事件引擎类

事件引擎类在D:\vnpy207\vnpy\event目录下的engine.py文件中定义。其中定义了两个类，分别是EventEngine和Event。下面按照代码顺序进行介绍。

EVENT\_TIMER = "eTimer"

定义时钟事件的类型（type成员）。

class Event:

def \_\_init\_\_(self, type: str, data: Any = None):

""""""

self.type = type

self.data = data

定义Event事件类。事件类比较简单，只有两个成员，type（字符串类型）和data（任意类型）。

# Defines handler function to be used in event engine.

HandlerType = Callable[[Event], None]

声明事件处理函数的类型。

下面是EventEngine事件引擎类的定义。

**初始化函数：**

def \_\_init\_\_(self, interval: int = 1):

"""

Timer event is generated every 1 second by default, if

interval not specified.

"""

self.\_interval = interval

self.\_queue = Queue()

self.\_active = False

self.\_thread = Thread(target=self.\_run)

self.\_timer = Thread(target=self.\_run\_timer)

self.\_handlers = defaultdict(list)

self.\_general\_handlers = []

\_interval：时钟的时间间隔，单位秒。

\_queue：事件队列，先进先出。事件引擎中只有这一个队列，说明vn.py的事件处理是没有优先级的。其实加上优先级也不难，多用几个队列就是了，估计是不需要。

\_active：引擎是否活动的标志。如果为False，则不处理队列中的事件；如果为True，如果发现队列中有事件，则处理。

\_thread：引擎的守护线程。指定本类的\_run函数为守护线程，该线程从事件队列中取事件并处理。

\_timer：时钟线程。指定本类的\_run\_timer函数为时钟线程，定时往事件队列中追加时钟事件。

\_handlers：事件处理函数字典。

\_general\_handlers：通用事件处理函数列表。

对于一个事件，先到\_handlers中找它对应的处理函数。通用事件处理函数对所有事件进行处理。

**守护线程：**

def \_run(self):

while self.\_active:

try:

event = self.\_queue.get(block=True, timeout=1)

self.\_process(event)

except Empty:

pass

如果本引擎是活动的，就循环处理：从队列中取事件。如果取到，就调用事件处理函数进行处理。

使用get方法从队列中取事件。参数block为True，表示使用阻塞方式，也就是当队列为空时等待timeout秒（此处是1秒）；等待之后如果还为空，就抛出Empty异常。这个异常被下面的except捕获，但并未做处理，也就是直接进入下一次循环等待。

说明：如果参数block为False，当队列为空时直接抛出Empty异常，直接进入下次循环，浪费资源。

上述代码中指明由本类的\_process函数处理事件。

**事件处理函数：**

def \_process(self, event: Event):

if event.type in self.\_handlers:

[handler(event) for handler in self.\_handlers[event.type]]

if self.\_general\_handlers:

[handler(event) for handler in self.\_general\_handlers]

如果在事件处理函数字典中有本类事件的处理函数，则处理之。每类事件的处理函数可能有多个，串行执行每一个。

如果通用事件处理函数列表不为空，则调用通用处理函数处理。通用处理函数可能有多个，串行执行每一个。

在此可以回顾一下初始化函数中的相关内容：

self.\_handlers = defaultdict(list)

self.\_general\_handlers = []

\_handlers是一个字典，key是事件类型，value是一个list，里存的是处理该事件的函数。\_general\_handlers没有key，就是一个list，是一个通用处理函数的列表，也就是不管是哪类事件，都要用这些方法处理一下。

注：目前通用函数只有对rpc\_service的事件进行处理，还没研究，略。所以，现在可以看成，只对各类事件调用特定的处理函数。

**时钟线程：**

在初始化函数中指定了本类的\_run\_timer函数为时钟线程。

def \_run\_timer(self):

while self.\_active:

sleep(self.\_interval)

event = Event(EVENT\_TIMER)

self.put(event)

功能是定时往事件队列中追加时钟事件。

从局部看，时钟线程定时向队列中插入时钟事件，但到目前为止还没有看到注册时钟事件的处理函数，好像时钟事件没有意义。

以后随着分析的深入，就会看到其它功能，如CTP接口，会为时钟事件注册特定的处理函数，那时时钟事件就是有意义的啦。参“CTP底层接口的使用”一章。

**启动函数：**

从前面守护线程和时钟线程可以看出，\_active是启停标志，不是暂停标志。如果\_active设为False，上述两个线程就会结束。只简单地将\_active设为True，不能重启这两个线程。所以要有专门的启动函数。

def start(self):

self.\_active = True

self.\_thread.start()

self.\_timer.start()

将引擎设为活动，并启动守护线程和时钟线程。

**停止函数：**

def stop(self):

self.\_active = False

self.\_timer.join()

self.\_thread.join()

将引擎设为不活动，并挂起两个线程。

**推送事件函数：**

def put(self, event: Event):

self.\_queue.put(event)

将事件推送到引擎的事件队列。

**注册事件处理函数：**

def register(self, type: str, handler: HandlerType):

"""

Register a new handler function for a specific event type. Every

function can only be registered once for each event type.

"""

handler\_list = self.\_handlers[type]

if handler not in handler\_list:

handler\_list.append(handler)

每个事件类型对应一个函数列表，可以有多个函数。同一函数在同一事件类型的列表中只能出现一次。

**注销事件处理函数：**

def unregister(self, type: str, handler: HandlerType):

if handler in handler\_list:

handler\_list.remove(handler)

if not handler\_list:

self.\_handlers.pop(type)

如果处理函数在对应类型的列表中，则从列表中删除。

如果列表为空，则从字典中删除对应类型的字典项。

**注册通用事件处理函数：**

def register\_general(self, handler: HandlerType):

if handler not in self.\_general\_handlers:

self.\_general\_handlers.append(handler)

**注销通用事件处理函数：**

def unregister\_general(self, handler: HandlerType):

if handler in self.\_general\_handlers:

self.\_general\_handlers.remove(handler)

了解了事件引擎类的定义之后，下面以CTA回测为例，讨论底层接口是如何将产生的事件推送给中层的事件引擎，而上层应用又是如何接收并处理这些事件的。

## 底层接口（引擎）推送事件到事件引擎

在BacktesterEngine回测引擎类的线程函数run\_backtesting()中（参“多线程 – 执行流程”一节），开始时有一句：

engine = self.backtesting\_engine

当执行完主要工作后，需要执行以下两句：

event = Event(EVENT\_BACKTESTER\_BACKTESTING\_FINISHED)

self.event\_engine.put(event)

综合起来，这些语句创建一个回测结束事件（该事件只有类型没有数据），并推送到本回测引擎的回测事件队列中。

## 上层应用处理事件

BacktesterManager是窗体类，它有一个成员变量backtester\_engine，指向一个BacktesterEngine类实例（参“多线程 – 类结构”一节），这是前文介绍过的。

窗体类首先定义一个静态成员

signal\_backtesting\_finished = QtCore.pyqtSignal(Event)

是一个PyQt消息。

在register\_event()函数中，

self.signal\_backtesting\_finished.connect(

self.process\_backtesting\_finished\_event)

指明process\_backtesting\_finished\_event()为该消息的处理函数。

以下两行

self.event\_engine.register(

EVENT\_BACKTESTER\_BACKTESTING\_FINISHED, self.signal\_backtesting\_finished.emit)

将signal\_backtesting\_finished消息的处理函数注册为EVENT\_BACKTESTER\_BACKTESTING\_FINISHED事件的处理函数。

其实上面有些绕，可以只用下面两行完成上述功能：

self.event\_engine.register(

EVENT\_BACKTESTER\_BACKTESTING\_FINISHED, self.process\_backtesting\_finished\_event)

其实，如果上层应用是窗体应用，可以直接给窗体发消息，不经过中间层的事件引擎，效率还能高一些，但这样的程序层次结构不清楚，两种方式各有利弊。

要使用统一的中间层引擎，就不会再去触发窗体消息了，否则一个事件会处理两次。估计是程序一路发展而来，没有重构得那么细致吧，比如某个时间段使用了消息处理机制，后来改成中间层引擎机制后，程序改得不彻底。我不是在挑vn.py的毛病，说实话，vn.py的代码水平真的很高，至少不比我看过的其它任何源代码差。

先写到这儿。感觉一周的努力很有收获，虽然分析还不够深入，但以此为基础，对继续分析很有信心。

此时是2019年10月20日，此时的南京桂花飘香。

第二部分 VN Trader

写完第二部分时突然意识到，这可能不是一个好的文档。

我是一个想做交易的程序员，学习vn.py的目的很明确，就是深度使用，真正用vn.py做量化交易。要做量化交易，就要能写自己的策略。如果现有平台不能满足自己策略的要求怎么办（比如我想要更高的效率，更强的策略功能）？就要对平台进行定制（这是我不使用网站型交易平台的原因）。要合理地定制甚至是优化，对原平台的体系结构一定要了解透彻，否则就是乱改。研究程序结构的重点是策略部分，但为了能真正深入研究，必须先扫清外围。

如果您也跟我一样把编程作为一个非常重要的目的，那咱们的做法很可能相似，本文档会对您有用。但这样的人很少，大多数人只为一般使用，简单了解程序结构即可，那您看完第一部分就可以。前文已经简单讨论过CTA策略，对于以使用为目的的人来说，那已经足够。

要深入研究策略，需要先掌握行情接口和交易平台的内容，本部分就完成这项工作。

本部分首先研究行情接口。vn.py支持多种行情接口，但默认的，也最常用的是CTP，本部分就以CTP为例进行介绍。

# CTP基础

综合交易平台（Comprehensive Transaction Platform）简称CTP，是专门为期货公司开发的一套期货经纪业务管理系统，由交易、风险控制和结算三大系统组成，交易系统主要负责订单处理、行情转发及银期转账业务，结算系统负责交易管理、帐户管理、经纪人管理、资金管理、费率设置、日终结算、 信息查询以及报表管理等，风控系统则主要在盘中进行高速的实时试算，以及时揭示并控制风险。（来自百度百科）

## 在C++中使用CTP

到上海期货信息技术有限公司（简称上期技术）主页<http://www.sfit.com.cn/>下载CTP接口文件及文档。

CTP的API使用建立在TCP协议之上的FTD协议（《期货交易数据交换协议》）与交易托管系统进行通讯，而交易托管系统负责投资者的交易业务处理。

我本来习惯用C++编程，这部分内容对我来说很容易，写本章是出于兴趣而不是出于需要。如果您对C++不感兴趣或者不熟悉，可以跳过本章，不会影响后续章节的阅读。

### CTP接口文件

CTP接口包含以下内容：

* ThostFtdcTraderApi.h：C++头文件，包含交易相关的指令，如报单。
* ThostFtdcMdApi.h：C++头文件，包含获取行情相关的指令。
* ThostFtdcUserApiStruct.h：包含了所有用到的数据结构。
* ThostFtdcUserApiDataType.h：包含了所有用到的数据类型和枚举描述。
* thosttraderapi.lib、thosttraderapi.dll：交易部分的动态链接库和静态链接库。
* thostmduserapi.lib、thostmduserapi.dll：行情部分的动态链接库和静态链接库。
* error.dtd、error.xml：包含所有可能的错误信息。

整个开发包有2个核心头文件——CThostFtdcMdApi和CThostFtdcTraderApi，一个处理行情，一个处理交易。

（1）处理行情的CThostFtdcMdApi接口有两个类，分别是CThostFtdcMdApi和CthostFtdcMdSpi，以Api结尾的用来下命令，以Spi结尾的用来响应命令的回调。

（2）处理交易的CThostFtdcTraderApi接口也有两个类，分别是CThostFtdcTraderApi和CthostFtdcTraderSpi，通过CThostFtdcTraderApi向CTP发送操作请求，通过CThostFtdcTraderSpi接收CTP的操作响应。

开发者通过CThostFtdcTraderApi就可以完成交易接口的初始化，登入，确认结算结果，查询合约，查询资金，查询持仓，报单，撤单等业务操作；通过CthostFtdcTraderSpi获取相应回报

开发者也可以通过CThostFtdcMdApi完成行情接口的初始化，登入，订阅，收行情等业务；通过CThostFtdcMdSpi获取相应的行情业务操作的回报。

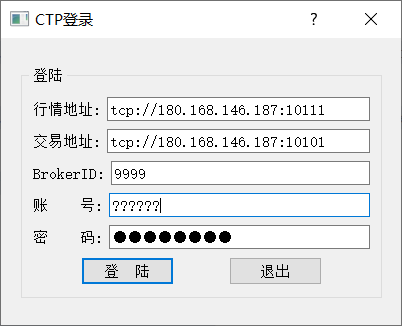
### 使用CTP接口

CTP没有好的官方文档和示例。网上相关的技术帖五花八门，技术水平参差不齐，我建议还是仔细分析官方文档和示例。

为了学习，我用VS2019+QT5.13.1做了一个示例，之所以选择QT而没有选择纯的C++，是基于以下原因：

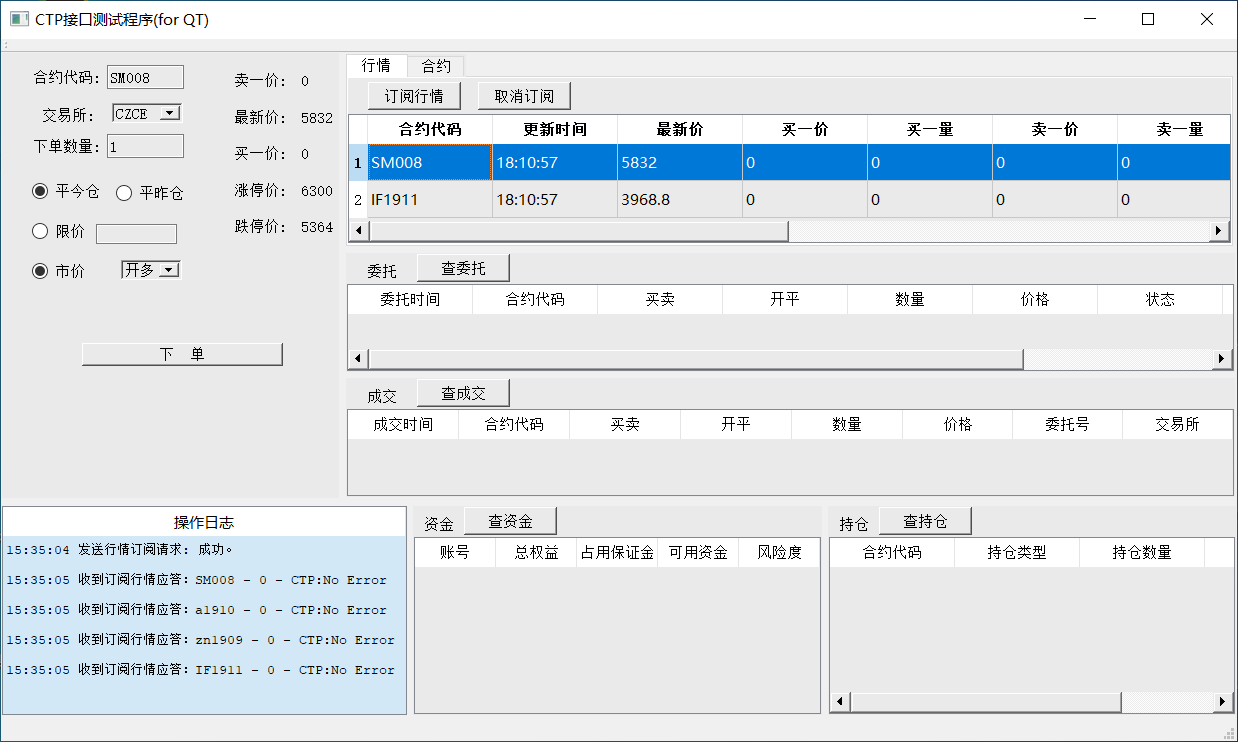
* 与纯C++中使用CTP极其相似。
* 有界面，可以使用QT的信号/槽机制。
* vn.py使用PyQT，可以互相参照。

程序的登录界面如下。



注：如果还没有仿真账号，可登录网站http://simnow.sfit.com.cn/申请，更改密码，第二天才能使用。

程序主界面如下。



这不是一个实用的程序，主界面模仿了VN Trader，但为了测试功能，各类请求都由按钮触发。程序太初级，不好意思分享给大家，仅以它的部分代码作原理性介绍。

要使用CTP接口，需要从CthostFtdcMdSpi和CthostFtdcTraderSpi继承两个类，类名可以任意。因为我使用QT，还需要从QObject继承，以能够使用QT的信号/槽机制，所以代码不太纯。好在以下代码不需要太仔细看，大致知道类的功能就行了。代码放到这儿，主要是方便与后面的Python代码比较。

MdSpi.h的代码如下：

#include <QObject>

#include "ThostFtdcMdApi.h"

typedef struct

{

char FRONT\_ADDR[100]; // 前置地址

TThostFtdcBrokerIDType BROKER\_ID; // 经纪公司代码

} MDStruct;

class MdSpi : public CThostFtdcMdSpi

{

public:

MdSpi();

~MdSpi();

void Init();

public:

MDStruct hq;

signals:

void sendMdLogin(int);

void sendData(QString);

public:

///错误应答

virtual void OnRspError(CThostFtdcRspInfoField \*pRspInfo,

int nRequestID, bool bIsLast);

///当客户端与交易后台通信连接断开时，该方法被调用。当发生这个情况后，API会自动重新连接，客户端可不做处理。

///@param nReason 错误原因

/// 0x1001 网络读失败

/// 0x1002 网络写失败

/// 0x2001 接收心跳超时

/// 0x2002 发送心跳失败

/// 0x2003 收到错误报文

virtual void OnFrontDisconnected(int nReason);

///心跳超时警告。当长时间未收到报文时，该方法被调用。

///@param nTimeLapse 距离上次接收报文的时间

virtual void OnHeartBeatWarning(int nTimeLapse);

///当客户端与交易后台建立起通信连接时（还未登录前），该方法被调用。

virtual void OnFrontConnected();

///登录请求响应

virtual void OnRspUserLogin(CThostFtdcRspUserLoginField \*pRspUserLogin, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///订阅行情应答

virtual void OnRspSubMarketData(CThostFtdcSpecificInstrumentField \*pSpecificInstrument, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///取消订阅行情应答

virtual void OnRspUnSubMarketData(CThostFtdcSpecificInstrumentField \*pSpecificInstrument, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///深度行情通知

virtual void OnRtnDepthMarketData(CThostFtdcDepthMarketDataField \*pDepthMarketData);

public:

// 以下函数调用用户接口，发送请求

// 订阅行情

void SubscribeMarketData();

private:

// 请求登录

void ReqUserLogin();

// 是否收到了出错的响应信息

bool IsErrorRspInfo(CThostFtdcRspInfoField \*pRspInfo);

private:

CThostFtdcMdApi\* pMdUserApi;

};

TdSpi.h的代码如下：

#include <QObject>

#include "ThostFtdcTraderApi.h"

typedef struct

{

char FRONT\_ADDR[100];

TThostFtdcBrokerIDType BROKER\_ID;

TThostFtdcInvestorIDType INVESTOR\_ID;

TThostFtdcPasswordType PASSWORD;

} TDStruct;

class TdSpi : public QObject, public CThostFtdcTraderSpi

{

Q\_OBJECT

public:

TDStruct jy;

//合约结构

typedef struct

{

int hycs; //合约数量乘数（合约乘数）

double hyds; //最小变动价位（合约点数）

} HYStruct;

void Init();

signals:

void sendTdLogin(int);

void sendCJ(QString);

void sendWT(QString);

void sendCC(QString);

void sendZJ(QString);

void sendHY(QString);

void sendDELCC(QString);

public:

TdSpi(QObject \*parent=NULL);

~TdSpi();

HYStruct hy(QString);

private:

public:

//报单录入请求

void ReqOrderInsert(QString dm, QString jys, QString lx, int lots, double price, QString pclx);

//报单操作请求(注：原来应该是通用报单，本示例改成了撤单。可参考上期原来的DEMO)

void ReqOrderAction(QString brokerid,QString wth,QString jys);

///当客户端与交易后台建立起通信连接时（还未登录前），该方法被调用。

virtual void OnFrontConnected();

///登录请求响应

virtual void OnRspUserLogin(CThostFtdcRspUserLoginField \*pRspUserLogin, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///投资者结算结果确认响应

virtual void OnRspSettlementInfoConfirm(CThostFtdcSettlementInfoConfirmField \*pSettlementInfoConfirm, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///请求查询合约响应

virtual void OnRspQryInstrument(CThostFtdcInstrumentField \*pInstrument, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///请求查询资金账户响应

virtual void OnRspQryTradingAccount(CThostFtdcTradingAccountField \*pTradingAccount, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///请求查询投资者持仓响应

virtual void OnRspQryInvestorPosition(CThostFtdcInvestorPositionField \*pInvestorPosition, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///报单录入请求响应

virtual void OnRspOrderInsert(CThostFtdcInputOrderField \*pInputOrder, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///报单操作请求响应

virtual void OnRspOrderAction(CThostFtdcInputOrderActionField \*pInputOrderAction, CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///错误应答

virtual void OnRspError(CThostFtdcRspInfoField \*pRspInfo, int nRequestID, bool bIsLast);

///当客户端与交易后台通信连接断开时，该方法被调用。当发生这个情况后，API会自动重新连接，客户端可不做处理。

virtual void OnFrontDisconnected(int nReason);

///心跳超时警告。当长时间未收到报文时，该方法被调用。

virtual void OnHeartBeatWarning(int nTimeLapse);

///报单通知

virtual void OnRtnOrder(CThostFtdcOrderField \*pOrder);

///成交通知

virtual void OnRtnTrade(CThostFtdcTradeField \*pTrade);

///请求查询资金账户

void ReqQryTradingAccount();

///请求查询投资者持仓

void ReqQryInvestorPosition();

///请求查询合约

void ReqQryInstrument();

private:

///用户登录请求

void ReqUserLogin();

///查询投资者结算结果确认

void ReqSettlementInfoConfirm();

// 是否收到了出错的响应信息

bool IsErrorRspInfo(CThostFtdcRspInfoField \*pRspInfo);

// 是否我的报单回报

bool IsMyOrder(CThostFtdcOrderField \*pOrder);

// 是否正在交易的报单

bool IsTradingOrder(CThostFtdcOrderField \*pOrder);

// USER\_API参数

CThostFtdcTraderApi\* pTdUserApi;

};

### 线程的使用

CTP我也是初学者，但经验上不认可网上的某些说法，比如说应该启动多少个线程什么的。

其它的任务需要启动多少个线程不管，单CTP通信，可能不需要自己控制启动额外的线程。

CTP既然是基于TCP的通信，那在类的初始化过程中，接口内部很可能已经启动了监听线程，对TCP连接进行监听，否则无法进行回调。网上有些示例可能就是受了错误说法的影响，生硬地启动一些不必要的线程，大家参考时要明辨。

写这段话不是较劲儿，先放到这儿提醒自己，然后继续学习，可能后面会发现是自己错了。

## CTP的Python封装

C++的API无法直接在Python中使用，所以需要为Python进行封装：

* C++ API中很多函数的调用参数是ApiStruct.h中定义的结构体，在Python中既无法直接创建这些结构体（主动函数），也无法提取结构体中包含的数据（回调函数）。
* Python虚拟机是基于C语言实现的，所有的Python对象，哪怕只是一个整数或者字符串，在C的环境中都是一个PyObject对象。如果在Python中直接传递一个参数到C++环境里，C++是无法识别的。

这部分我还没研究，暂略，不影响后面内容的理解。

# CTP接口的定义

从本章开始讨论，vn.py中如何使用CTP接口获取行情数据并进行交易。

## VN Trader的全局定义

VN Trader是一个支持手工交易的平台，它的全局定义大多与交易有关。

在D:\vnpy207\vnpy\trader目录下：

* event.py定义各类事件，包括EVENT\_TICK、EVENT\_ORDER、EVENT\_TRADE、EVENT\_POSITION、EVENT\_ACCOUNT、EVENT\_CONTRACT和EVENT\_LOG等。
* object.py定义各类数据结构，包括TickData、OrderData、TradeData、PositionData、AccountData、ContractData、LogData、OrderRequest、CancelRequest、SubscribeRequest和HistoryRequest等。

## 底层接口基类

在D:\vnpy207\vnpy\trader目录下的gateway.py文件中定义了所有底层接口的基类BaseGateway。代码比较重要，基本全部引用，并使用注释的形式进行说明，读者应该通读。

吐槽一下，在老版本的源程序中有很好的中文注释，现在改成很少的英文注释。官方说这是进步，反正看代码会更费劲。你一中文软件，界面都是中文的，用什么英文注释呀？被谁忽悠啦？或者另有深意。

gateway.py的代码如下：

from abc import ABC, abstractmethod

from typing import Any, Sequence

from copy import copy

from vnpy.event import Event, EventEngine

from .event import (

EVENT\_TICK,

EVENT\_ORDER,

EVENT\_TRADE,

EVENT\_POSITION,

EVENT\_ACCOUNT,

EVENT\_CONTRACT,

EVENT\_LOG,

)

from .object import (

TickData,

OrderData,

TradeData,

PositionData,

AccountData,

ContractData,

LogData,

OrderRequest,

CancelRequest,

SubscribeRequest,

HistoryRequest

)

class BaseGateway(ABC):

"""

抽象接口类，用于创建连接不同交易系统的接口。

如何实现一个接口：

## 基础

一个接口应该满足：

\* 这个类必须是线程安全的

\* 所有的方法都要线程安全

\* 实例间没有可变的共享属性

\* 所有方法都是非阻塞的

\* satisfies all requirements written in docstring for every method and callbacks.

\* 意外断链时能够自动连接

## 所有的@abstractmethod方法都必须实现

## 以下回调函数必须手工实现：

\* on\_tick

\* on\_trade

\* on\_order

\* on\_position

\* on\_account

\* on\_contract

传给回调函数的XxxData参数必须是不可变的，也就是说，当这些对象传递给on\_xxxx方法后不能被修改。

So if you use a cache to store reference of data, use copy.copy to create a new object

before passing that data into on\_xxxx

"""

# connect函数需要用到的配置参数字典

default\_setting = {}

# 接口支持的交易所

exchanges = []

def \_\_init\_\_(self, event\_engine: EventEngine, gateway\_name: str):

"""创建接口实例时，需要传入事件引擎和接口名称"""

self.event\_engine = event\_engine

self.gateway\_name = gateway\_name

def on\_event(self, type: str, data: Any = None):

"""通用函数，将事件加入事件引擎"""

event = Event(type, data)

self.event\_engine.put(event)

def on\_tick(self, tick: TickData):

"""

市场行情推送

Tick event push.

Tick event of a specific vt\_symbol is also pushed.

"""

self.on\_event(EVENT\_TICK, tick)

self.on\_event(EVENT\_TICK + tick.vt\_symbol, tick)

def on\_trade(self, trade: TradeData):

"""

成交信息推送

Trade event push.

Trade event of a specific vt\_symbol is also pushed.

"""

self.on\_event(EVENT\_TRADE, trade)

self.on\_event(EVENT\_TRADE + trade.vt\_symbol, trade)

def on\_order(self, order: OrderData):

"""

订单变化推送

Order event push.

Order event of a specific vt\_orderid is also pushed.

"""

self.on\_event(EVENT\_ORDER, order)

self.on\_event(EVENT\_ORDER + order.vt\_orderid, order)

def on\_position(self, position: PositionData):

"""

持仓信息推送

Position event push.

Position event of a specific vt\_symbol is also pushed.

"""

self.on\_event(EVENT\_POSITION, position)

self.on\_event(EVENT\_POSITION + position.vt\_symbol, position)

def on\_account(self, account: AccountData):

"""

账户信息推送

Account event push.

Account event of a specific vt\_accountid is also pushed.

"""

self.on\_event(EVENT\_ACCOUNT, account)

self.on\_event(EVENT\_ACCOUNT + account.vt\_accountid, account)

def on\_log(self, log: LogData):

"""

日志推送

Log event push.

"""

self.on\_event(EVENT\_LOG, log)

def on\_contract(self, contract: ContractData):

"""

合约基础信息推送

Contract event push.

"""

self.on\_event(EVENT\_CONTRACT, contract)

def write\_log(self, msg: str):

"""

Write a log event from gateway.

"""

log = LogData(msg=msg, gateway\_name=self.gateway\_name)

self.on\_log(log)

@abstractmethod

def connect(self, setting: dict):

"""

连接

要实现这个方法，你必须：

\* 连接服务器（如果需要）

\* 当所有需要的连接都建立之后，创建connected日志

\* 执行以下查询，并在on\_xxxx中对返回做出响应，并写日志

\* contracts : on\_contract

\* account asset : on\_account

\* account holding: on\_position

\* orders of account: on\_order

\* trades of account: on\_trade

\* 如果上述任何查询失败，写日志

未来计划:

response callback/change status instead of write\_log

"""

pass

@abstractmethod

def close(self):

"""

关闭连接

"""

pass

@abstractmethod

def subscribe(self, req: SubscribeRequest):

"""

订阅行情

"""

pass

@abstractmethod

def send\_order(self, req: OrderRequest) -> str:

"""

发送新的订单到服务器

实现时需要完成以下工作：

\* 使用OrderRequest.create\_order\_data，从req创建一个OrderData

\* 为OrderData.orderid分配一个接口实例范围内唯一的值

\* 向服务器发送请求

\* 如果请求发送成功，OrderData.status应该设为Status.SUBMITTING

\* 如果请求发送失败，OrderData.status应该设为Status.REJECTED

\* response on\_order:

\* 返回OrderData.vt\_orderid

:return str vt\_orderid for created OrderData

"""

pass

@abstractmethod

def cancel\_order(self, req: CancelRequest):

"""

撤消一个已经存在的订单

实现时需要完成以下工作：

\* 向服务器发送请求

"""

pass

def send\_orders(self, reqs: Sequence[OrderRequest]):

"""

发送一批订单到服务器

默认在for循环中调用send\_order函数。

如果服务器支持批量下单，可以重写这个函数。

"""

vt\_orderids = []

for req in reqs:

vt\_orderid = self.send\_order(req)

vt\_orderids.append(vt\_orderid)

return vt\_orderids

def cancel\_orders(self, reqs: Sequence[CancelRequest]):

"""

批量撤单

默认在for循环中调用cancel\_order函数。

如果服务器支持批量下单，可以重写这个函数。

"""

for req in reqs:

self.cancel\_order(req)

@abstractmethod

def query\_account(self):

"""

查询账户资金

"""

pass

@abstractmethod

def query\_position(self):

"""

查询持仓

"""

pass

def query\_history(self, req: HistoryRequest):

"""

查询历史K线数据

"""

pass

def get\_default\_setting(self):

"""

返回配置参数字典。

"""

return self.default\_setting

将上述代码与“在C++中使用CTP”一节内容进行对照，很容易理解。也说明，大部分接口的功能都是相似的。

在同一程序文件中还定义了一个LocalOrderManager类，好像只用于火币，不再分析。

## CtpGateway类

vn.py中实现CTP通信和类是CtpGateway，继承自BaseGateway。

CtpGateway类在D:\vnpy207\vnpy\gateway\ctp目录下的ctp\_gateway.py文件中定义。本节只介绍初始化部分，具体的功能在后续章节中介绍。

class CtpGateway(BaseGateway):

"""

VN Trader的CTP接口

"""

default\_setting = {

"用户名": "",

"密码": "",

"经纪商代码": "",

"交易服务器": "",

"行情服务器": "",

"产品名称": "",

"授权编码": "",

"产品信息": ""

}

exchanges = list(EXCHANGE\_CTP2VT.values())

def \_\_init\_\_(self, event\_engine):

"""构造函数"""

super().\_\_init\_\_(event\_engine, "CTP")

self.td\_api = CtpTdApi(self)

self.md\_api = CtpMdApi(self)

可以看到，在初始化函数中用两个成员变量来保存CtpTdApi类和CtpMdApi类的实例。这两个类也在本文件中定义，是CTP接口的Python封装。分析其代码，与CTP原始接口的代码非常相似。其实在多层次封装中，代码都是非常相似的。封装的层次如下图所示。



调用的层次正好反过来，从上向下。

# CTP底层接口的使用

与具体底层接口相关的程序都在D:\vnpy207\vnpy\gateway目录中，该目录下子目录的名称就是具体底层接口的名称。如CTP接口在ctp子目录中。

CTP接口类CtpGateway在ctp\_gateway.py中定义。先不管它如何定义，先看VN Trader主窗口如何使用它。

## 加载CTP底层接口

vn.py的当前版本编程水平很高，使用了大量的通用处理。这样，对一件事情的处理，程序就可能分布在多处，下面分别介绍。

1-程序主函数中

程序主函数中调用add\_gateway函数将CtpGateway加载到主引擎，参“程序主函数”和“主引擎”两节。

main\_engine.add\_gateway(CtpGateway)

2-主窗口中创建菜单时

函数init\_menu负责为主窗口创建菜单，其中与CTP接口相关的代码如下：

def init\_menu(self):

# System menu

sys\_menu = bar.addMenu("系统")

gateway\_names = self.main\_engine.get\_all\_gateway\_names()

for name in gateway\_names:

func = partial(self.connect, name)

self.add\_menu\_action(sys\_menu, f"连接{name}", "connect.ico", func)

在主引擎中取出所有底层接口（包括CTP接口）。

用本类的connect函数，使用接口名（'CTP'）作为默认参数生成一个新函数，用新函数作为菜单项的槽函数。

注：Python functools模块

functools 模块中主要包含了一些函数装饰器和便捷的功能函数。

functools.partial(func, \*args, \*\*keywords)：该函数用于为 func 函数的部分参数指定参数值，从而得到一个转换后的函数，程序以后调用转换后的函数时，就可以少传入那些己指定值的参数。

3-增加菜单项

主窗口的add\_menu\_action函数用于增加菜单项。

def add\_menu\_action(

self,

menu: QtWidgets.QMenu,

action\_name: str,

icon\_name: str,

func: Callable,

):

""""""

icon = QtGui.QIcon(get\_icon\_path(\_\_file\_\_, icon\_name))

action = QtWidgets.QAction(action\_name, self)

action.triggered.connect(func)

action.setIcon(icon)

menu.addAction(action)

调用add\_menu\_action函数后，在主界面的“系统”菜单中增加了一项“连接CTP”，该项的处理函数是connect('CTP')。

## “连接CTP”的处理函数

在菜单中选择了“连接CTP”，会调用connect('CTP')，connect函数的代码如下：

def connect(self, gateway\_name: str):

dialog = self.connect\_dialogs.get(gateway\_name, None)

if not dialog:

dialog = ConnectDialog(self.main\_engine, gateway\_name)

dialog.exec\_()

其功能是打开如下图的对话框。



程序先从主窗口的连接对话框集合中找对应的对话框，如果找到就使用它；如果找不到，就使用通用的ConnectDialog对话框。

connect\_dialogs中没有与CTP相关的对话框，所以打开通用连接对话框ConnectDialog。

## 通用连接对话框类ConnectDialog

ConnectDialog在D:\vnpy207\vnpy\trader\ui目录下的widget.py中定义。

其功能是取对应接口类的配置参数，用这些配置参数生成界面，并将接口类的connect函数指定为“连接”按钮的槽函数。

最终结果是：当按对话框中的“连接”按钮时，调用了CTP接口的connect函数。

## CTP接口的connect函数

后续的执行就进入了CtpGateway类内部，CtpGateway类在D:\vnpy207\vnpy\gateway\ctp目录下的ctp\_gateway.py文件中定义。

def connect(self, setting: dict):

……

self.td\_api.connect(td\_address, userid, password, brokerid, auth\_code, appid, product\_info)

self.md\_api.connect(md\_address, userid, password, brokerid)

#初始化账户查询函数

self.init\_query()

功能是先通过两个成员变量连接交易服务器和行情服务器，再调用init\_query函数对账户查询函数进行初始化。

def init\_query(self):

""""""

self.count = 0

# 将查询函数设为td\_api的资金和持仓查询函数

self.query\_functions = [self.query\_account, self.query\_position]

# 注册事件引擎中时钟事件的处理函数

self.event\_engine.register(EVENT\_TIMER, self.process\_timer\_event)

将本类的process\_timer\_event函数注册为事件引擎的时钟事件处理函数。

## 时钟事件的处理函数

事件引擎的时钟事件可以注册多个处理函数，与行情查询有关的处理函数如下。

def process\_timer\_event(self, event):

self.count += 1

if self.count < 2:

return

self.count = 0

func = self.query\_functions.pop(0)

func()

self.query\_functions.append(func)

看这意思，是每两秒钟刷新一次账户、持仓信息。

经过上述步骤，就连接到了CTP行情服务器，并可定时查询账户信息。

# Tick数据的使用

VN Trader是一个交易平台，主要功能就是发送交易请求和不断显示（刷新）行情、账户信息。

本章以Tick数据为例，对Tick数据的使用进行分析。其它类型数据的使用方法基本相同。

如果通过CTP接口订阅了行情，就会定时收到特定合约的Tick数据。CtpGateway的回调函数会将Tick数据加入到事件引擎中等待处理。

## 在主窗口上订阅行情

如下图是VN Trader主窗口的左上部分。



这部分窗口在D:\vnpy207\vnpy\trader\ui目录下的widget.py文件中的TradingWidget类中定义。

在类的初始化部分，相关代码如下：

class TradingWidget(QtWidgets.QWidget):

"""

主窗口的交易子窗口，完成一般的手工交易

"""

def \_\_init\_\_(self, main\_engine: MainEngine, event\_engine: EventEngine):

……

self.init\_ui()

……

def init\_ui(self):

……

# 创建“代码”输入框，并关联其回车键的槽函数

self.symbol\_line = QtWidgets.QLineEdit()

self.symbol\_line.returnPressed.connect(self.set\_vt\_symbol)

当在“代码”输入框中按回车键时，调用set\_vt\_symbol函数。set\_vt\_symbol看的相关代码如下：

def set\_vt\_symbol(self):

……

# 创建行情订阅请求

req = SubscribeRequest(

symbol=symbol, exchange=Exchange(exchange\_value)

)

self.main\_engine.subscribe(req, gateway\_name)

调用主引擎的subscribe函数订阅行情。主引擎中相关代码如下：

def subscribe(self, req: SubscribeRequest, gateway\_name: str):

gateway = self.get\_gateway(gateway\_name)

if gateway:

gateway.subscribe(req)

绕了一大圈，还是调用CtpGateway类中的subscribe函数：

def subscribe(self, req: SubscribeRequest):

"""订阅行情"""

self.md\_api.subscribe(req)

CtpGateway类中的subscribe函数则是简单地调用CtpMdApi类的subscribe函数。

使用的是交易接口的典型调用方法，参“CtpGateway类”一节中与封装有关的内容。

向行情服务器发送subscribe请求之后，行情服务器就会定时回送所请求合约的Tick数据。

## Tick数据的接收

当CTP接口接收到Tick数据后，首先回调CtpMdApi类的onRtnDepthMarketData函数，代码如下：

def onRtnDepthMarketData(self, data: dict):

"""

收到Tick数据时的回调函数

"""

……

self.gateway.on\_tick(tick)

看起来是调用CtpGateway的on\_tick函数，其实CtpGateway没有重写on\_tick函数，因此，调用的是BaseGateway类的on\_tick函数，相关代码如下：

def on\_event(self, type: str, data: Any = None):

"""通用函数，将事件加入事件引擎"""

event = Event(type, data)

self.event\_engine.put(event)

def on\_tick(self, tick: TickData):

"""

市场行情推送

向事件引擎推送一个Tick事件，同时推送一个特定的vt\_symbol Tick事件。

"""

self.on\_event(EVENT\_TICK, tick)

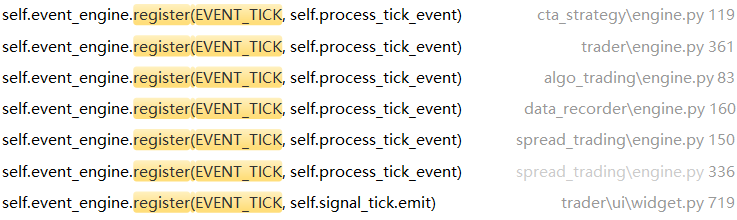
self.on\_event(EVENT\_TICK + tick.vt\_symbol, tick)

将EVENT\_TICK事件推送到事件引擎中，等待为该事件所注册的处理函数进行处理。

## Tick数据使用在哪里

本章以Tick数据为例，研究行情数据的使用。如果要研究Tick数据在某项功能中的使用，可从具体功能入手。如果是笼统地了解，可用本节的方法。

如前文所述，所有的事件都要放到事件引擎中等待处理。在整个项目中搜索字符串“register(EVENT\_TICK”，可以看到所有“直接”为EVENT\_TICK事件注册的函数调用。



上述函数调用分别对应：

* CTA策略引擎
* 交易引擎
* 算法交易(Algo Trading)
* RecorderEngine中用Tick数据生成Bar数据。
* 点差交易 （Spread Trading）
* VN Trader界面

如果在项目中搜索EVENT\_TICK，得到的结果会非常多，分析每个结果，就可以找到一些使用“间接”方法为EVENT\_TICK事件注册函数的地方，从而找出Tick数据所有的使用场合。

## 在CTA策略引擎中使用Tick数据

如前图所示，打开D:\vnpy207\vnpy\app\cta\_strategy目录下的engine.py文件，找到process\_tick\_event函数。在CTA策略引擎中，Tick处理函数代码如下：

def process\_tick\_event(self, event: Event):

tick = event.data

strategies = self.symbol\_strategy\_map[tick.vt\_symbol]

if not strategies:

return

self.check\_stop\_order(tick)

for strategy in strategies:

if strategy.inited:

self.call\_strategy\_func(strategy, strategy.on\_tick, tick)

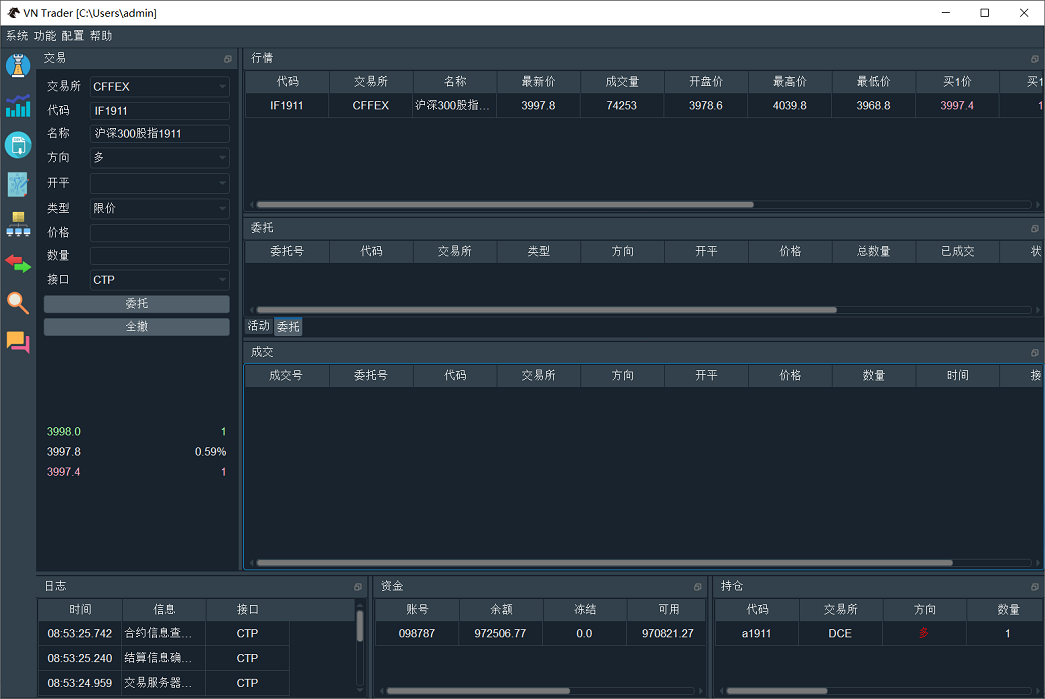
可以看到，CTA策略引擎会对Tick数据应用多种策略。

由于EVENT\_TICK既可能由回测策略产生，也可能来自实时行情，所以上述代码既用于策略回测，也用于实盘。

这部分内容在后文中继续研究。

## 在界面上使用Tick数据

先看一下VN Trader的主界面。



### 在交易子窗口中使用

在D:\vnpy207\vnpy\trader\ui目录下的widget.py文件中搜索“register(EVENT\_TICK”，发现该代码在TradingWidget(QtWidgets.QWidget)类中，对应主窗口的“交易”子窗口。要达到的效果是不断刷新交易子窗口中合约的价格信息。

def process\_tick\_event(self, event: Event):

"""EVENT\_TICK事件处理"""

# 取tick数据

# 显示最新价和买一卖一

# 显示涨跌百分比

# 显示十档行情

### 在行情子窗口中使用

观察VN Trader的运行界面，发现“行情”子窗口中合约的最新价、成交量等也会实时变化，说明该子窗口也使用Tick数据，该子窗口使用间接注册的方法注册EVENT\_TICK事件处理函数。

在D:\vnpy207\vnpy\trader\ui目录下的widget.py中定义主窗口的各组成部分（子窗口）。

仔细观察可以看出，在主窗口上，行情、委托、成交、资金和持仓等子窗口在风格上具有相似性，都是一个标题栏加一个QTableWidget控件，可以使用相同的方法实现。vn.py为所有此类子窗口定义基类BaseMonitor，上述子窗口都是BaseMonitor的子类，QTableWidget中显示不同的列，对不同的事件进行响应。行情子窗口就是对EVENT\_TICK事件进行响应。

以下略。

第三部分 CTA策略

CTA (Commodity Trading Advisor Strategy)策略称为商品交易顾问策略，也称作管理期货。商品交易顾问对商品等投资标的走势做出预判，通过期货期权等衍生品在投资中进行做多、做空或多空双向的投资操作，为投资者获取来自于传统股票、债券等资产类别之外的投资回报。

狭义上来说，CTA策略的研究对象只包括期货，像国内的股指期货，大宗商品期货和国债期货（利率期货），这些品种是目前国内CTA策略的主要研究对象和利润来源。广义上来说，可以是大宗商品期货，国债期货（利率期货），股票，外汇（包括spots和futures），甚至期权等任何有一定历史公开量价数据的品种。

CTA策略研究周期：通常来说，以分钟、小时和日线数据为主。少部分CTA策略也会用到tick数据，包括level2的bidprice，askprice，bidvolume，askvolume。

CTA策略研究方法：对单个品种历史上的量价数据进行分析，包括开盘价、收盘价、最高价、最低价、成交量、持仓量这些数据，提炼出具有概率优势的规律，这就是通常意义上的因子或者策略，并将此规律用代码实现，并假设这类规律在未来会依然存在。最后用此类规律来判断品种未来的方向，进行开仓、平仓、加仓、减仓等操作，并以此来获利。通常来说，演化至今的CTA策略基本都是全自动交易，但也依然有辅之以手工判断的交易存在。

（来自百度知道）

研究vn.py的策略部分是本文的最主要目的，采取的方法是：先在第一部分简单接触CTA并建立概念，第二部分扫清外围，本部分深入研究。

# CTA回测界面

CTA回测是vn.py的上层应用程序。本章研究CTA回测的界面程序。

## 类结构

在D:\vnpy207\vnpy\trader目录下的app.py文件中，定义所有上层App的基类。

class BaseApp(ABC):

"""

App的基类。

"""

app\_name = "" # 用于创建引擎和窗口的唯一名称

app\_module = "" # App的模块字符串，在import模块时使用

app\_path = "" # App的绝对路径

display\_name = "" # 显示在菜单中的名称

engine\_class = None # App的引擎类

widget\_name = "" # 窗口的类名称

icon\_name = "" # 窗口图标的文件名

可以看到，只有成员变量，没有方法。成员变量用于注册应用程序和创建菜单等，其中engine\_class用于指明引擎类。

引擎类在D:\vnpy207\vnpy\app\cta\_backtester目录下的\_\_init\_\_.py文件中定义

class CtaBacktesterApp(BaseApp):

app\_name = APP\_NAME

app\_module = \_\_module\_\_

app\_path = Path(\_\_file\_\_).parent

display\_name = "CTA回测"

engine\_class = BacktesterEngine

widget\_name = "BacktesterManager"

icon\_name = "backtester.ico"

执行的结果：

app\_name = "CtaBacktester"

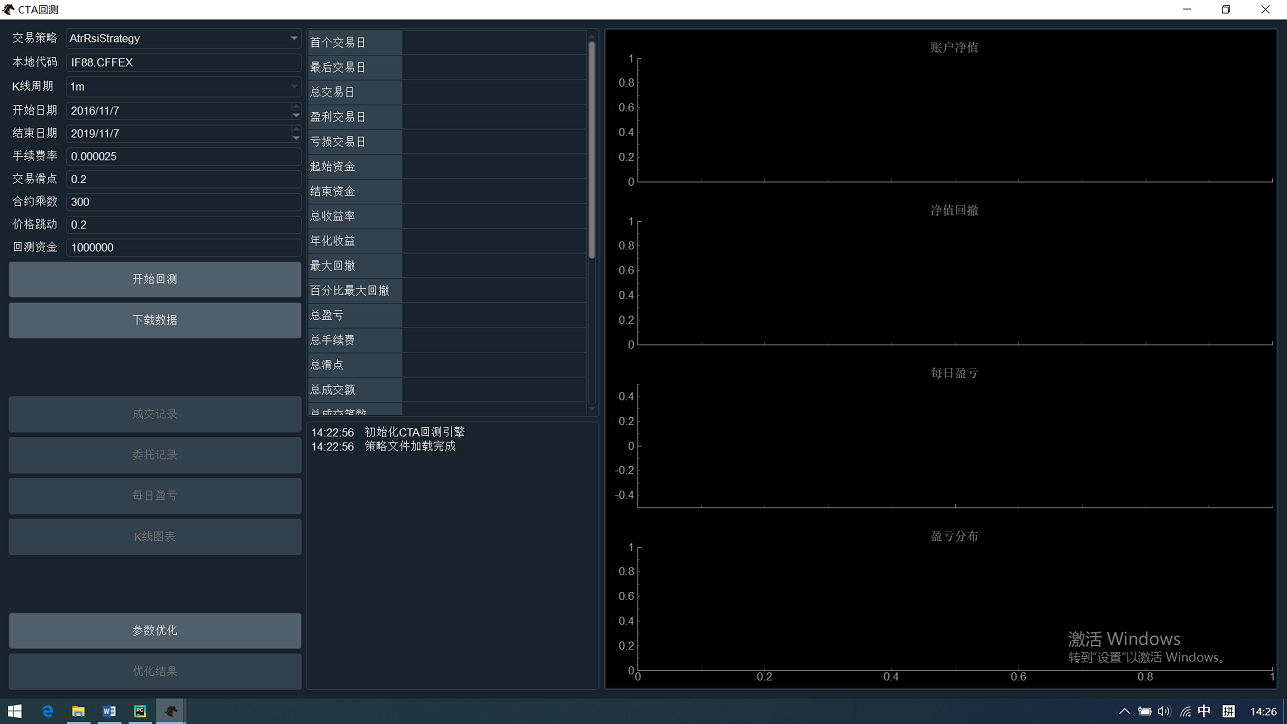
app\_module = vnpy.app.cta\_backtester

app\_path = D:\vnpy207\vnpy\app\cta\_backtester

engine\_class = BacktesterEngine

## CTA回测窗口

执行时CTA回测窗口的界面如下图所示。



在D:\vnpy207\vnpy\app\cta\_backtester\ui目录下的widget.py文件中定义回测窗口类BacktesterManager(QtWidgets.QWidget)。

在VN Trader的init\_menu函数中，将App增加到菜单和工具栏。

### 加载策略

加载策略到“交易策略”列表，到哪些目录中找。

略。

### 各窗口控件

略。

# CTA策略

## CTA策略模板

在D:\vnpy207\vnpy\app\cta\_strategy目录下的template.py文件中，定义所有CTA策略的基类CtaTemplate。我们为其增加必要的注释并去掉执行代码，如下（读者应该通读源代码）：

class CtaTemplate(ABC):

"""CTA策略模板"""

author = ""

parameters = [] # 默认的策略参数

variables = [] # 变量列表

def \_\_init\_\_(

self,

cta\_engine: Any,

strategy\_name: str,

vt\_symbol: str,

setting: dict,

):

""""""

def update\_setting(self, setting: dict):

"""用配置字典内的值作为策略属性"""

@classmethod

def get\_class\_parameters(cls):

"""取默认的策略参数"""

def get\_parameters(self):

"""取本策略具体的策略参数"""

def get\_variables(self):

"""取策略变量字典"""

def get\_data(self):

"""取策略数据"""

@virtual

def on\_init(self):

"""策略初始化时的回调函数（必须由用户继承实现）"""

@virtual

def on\_start(self):

"""策略启动时的回调函数（必须由用户继承实现）"""

@virtual

def on\_stop(self):

"""策略停止时的回调函数（必须由用户继承实现）"""

@virtual

def on\_tick(self, tick: TickData):

"""收到行情TICK推送（必须由用户继承实现）"""

@virtual

def on\_bar(self, bar: BarData):

"""收到Bar推送（必须由用户继承实现）"""

@virtual

def on\_trade(self, trade: TradeData):

"""收到成交推送（必须由用户继承实现）"""

@virtual

def on\_order(self, order: OrderData):

"""收到委托变化推送（必须由用户继承实现）"""

@virtual

def on\_stop\_order(self, stop\_order: StopOrder):

"""收到停止单推送（必须由用户继承实现）"""

def buy(self, price: float, volume: float, stop: bool = False, lock: bool = False):

"""买开：发送买单开多仓"""

def sell(self, price: float, volume: float, stop: bool = False, lock: bool = False):

"""卖平：发送卖单平多仓"""

def short(self, price: float, volume: float, stop: bool = False, lock: bool = False):

"""卖开：Send short order to open as short position"""

def cover(self, price: float, volume: float, stop: bool = False, lock: bool = False):

"""买平：Send cover order to close a short position"""

def send\_order(

self,

direction: Direction,

offset: Offset,

price: float,

volume: float,

stop: bool = False,

lock: bool = False

):

"""发送委托"""

def cancel\_order(self, vt\_orderid: str):

"""撤消一个已经存在的委托"""

def cancel\_all(self):

"""撤消由策略发出的所有委托"""

def write\_log(self, msg: str):

"""写一条日志信息"""

def get\_engine\_type(self):

"""返回CTA引擎是在回测还是实盘（live trading）"""

def load\_bar(

self,

days: int,

interval: Interval = Interval.MINUTE,

callback: Callable = None,

):

"""在初始化策略时加载历史Bar数据"""

def load\_tick(self, days: int):

"""在初始化策略时加载历史Tick数据"""

def put\_event(self):

"""发出由界面变化引起的策略数据事件"""

def send\_email(self, msg):

"""向默认的接收者发送邮件"""

理解了上述代码，就知道在vn.py中，一个策略需要定义哪些内容，也就会写自己的策略了。

## CTA策略示例

下面以一个系统自带的策略为例。打开D:\vnpy207\vnpy\app\cta\_strategy\strategies目录下的文件double\_ma\_strategy.py，代码如下。

略。

## 目标持仓模板

允许直接通过修改目标持仓来实现交易的策略模板。

class TargetPosTemplate(CtaTemplate):

"""

允许直接通过修改目标持仓来实现交易的策略模板

开发策略时，无需再调用buy/sell/cover/short这些具体的委托指令，

只需在策略逻辑运行完成后调用setTargetPos设置目标持仓，底层算法

会自动完成相关交易，适合不擅长管理交易挂撤单细节的用户。

使用该模板开发策略时，请在以下回调方法中先调用母类的方法：

onTick

onBar

onOrder

假设策略名为TestStrategy，请在onTick回调中加上：

super(TestStrategy, self).onTick(tick)

其他方法类同。

"""

# 结语

文档写到这儿，突然不想写了。

随着对系统越来越熟悉，分析的速度大大加快。在分析的过程还会随手记录，但整理已经成为负担。

把这个文档发出来，主要是表达对vn.py的感谢。但太详细的技术文档对vn.py不一定好，还是不写了吧。

此时是2019年11月13日，此时的滇池，红嘴鸥自由飞翔。

稳转