vn.py 2.2.0源代码深入分析

作者：稳转

前言

本文档是《Python量化交易从入门到实战》的附配资源之一，其他的附配资源可加入QQ群881645236下载。

《Python量化交易从入门到实战》一书由清华大学出版社出版，该书既是一本针对所有层次读者的Python编程教学书籍，又是一本利用Python解决量化交易实际问题的专业书籍，共分为四个部分。第一部分是Python语言基础，主要介绍Python的基础编程、数据结构、结构化编程、函数以及模块和包等内容，掌握这一部分可以算是Python基本入门。第二部分是Python编程进阶，包括面向对象的编程、面向数据的分析与可视化以及数据持久化等内容，掌握了这一部分可以进行Python的专业编程实践。第三部分是使用PyQt进行界面开发。PyQt是一种常用而强大的图形用户界面(GUI)设计工具，使用它可以设计出美观、易用的用户界面。掌握这一部分，可以在大型项目团队中完成比较核心的工作。第四部分是vn.py量化交易平台，为读者提供高水平实践机会，在巩固专业程序员水平的同时，也在量化金融这个Python的重要应用领域中进行深入探索。该书目录为：

第一部分 Python语言基础

第1章 准备工作

第2章 初识Python编程

第3章 数据结构

第4章 结构化编程

第5章 函数

第6章 模块和包

第二部分 Python编程进阶

第7章 面向对象编程

第8章 数据分析与可视化

第9章 数据持久化

第三部分 使用PyQt进行界面开发

第10章 PyQt基础

第11章 PyQt5界面编程

第12章 PyQt5控件

第13章 Qt Designer的使用

第14章 PyQt5绘图

第四部分 vn.py量化交易平台

第15章 vn.py的使用

第16章 VN Trader分析

第17章 数据库操作

第18章 CTA回测

该书内容一脉相承，第四部分与前三部分是有机结合而不是割裂的。例如Python的数据分析和可视化工具有很多种，该书选择的NumPy、Pandas、matplotlib和PyQtGraph本身就是其中最常用的，也都在vn.py中确实得到了应用。虽然该书第四部分只是vn.py代码分析的基础，但vn.py编程涉及的所有技术，在前三部分都有介绍。

作为技术书籍，《Python量化交易从入门到实战》从技术上来说已经包含了vn.py系统实现的核心内容，并已经认真分析了所涉及的代码，但不可能涉及所有细节，受篇幅限制也不可能过分深入。从功能上来说，vn.py作为一个专业化平台系统，《Python量化交易从入门到实战》只涉及了其中的一小部分功能，本文档与大家一起继续深入分析vn.py的其他部分。

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。本文档是对vn.py的全面分析，希望对大家有所帮助。

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。本文第一、二部分采用的方法是从功能入手，从多个代码文件中找出与特定功能相关的代码，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。

随着分析的深入，已经不再允许任何含糊其辞。每个函数，每个函数的每个参数都要搞清楚，对系统要达到通晓的程度，将来才可能改写系统。本文第三、四部分，很多内容以源码的形式呈现，在源码上增加详细的注释。遇到具体的功能，就会一直深入分析到底。其中第三部分采用自顶向下的分析方法，第四部分采用自底向上的分析方法，确保读者融会贯通。

本文档不是书稿，在编排上可能不够严谨。本人水平有限，文档中肯定会存在问题和错误，敬请批评指正。

第一部分 初步分析

本部分对应《Python量化交易从入门到实战》一书的第四部分。对于Python和vn.py的初学者，需要先系统学习教材，因为教材在内容编排上更完整合理。教材上有的这里就不再重复，免得混乱，本部分仅包含对教材对应内容的补充与修正（随着软件版本而变化）。

不方便买教材的朋友，我在vn.py社区的经验分享板块还有一篇置顶文章《vn.py 2.0.7源代码深入分析》，内容虽然略旧但结构比较完整，可以参考。

本部分目标：理清vn.py程序的整体架构。

# vn.py的使用

对应《Python量化交易从入门到实战》的第15章。

本章学习方法：以教材为主线进行学习，同时对照参考本文档。

## 量化交易基础

量化交易涉及多方面的知识，包括但不限于：

* 一般分析技术，如MACD等指标
* 波浪理论/缠论等高级分析技术
* 基于机器学习的技术，如线性回归和SVM等
* 基于人工智能的技术，如循环神经网络、卷集神经网络和强化学习等
* 基于概率论的时间序列分析方法，如ARIMA、GARCH等
* 基于金融工程学的方法，如定价方法、套利方法等

从专业学科的角度看，量化交易涉及多个学科，包括：证券投资学、计算机科学、数学（包括线性代数、概率论和数理统计等）和机器学习/人工智能等。

量化交易是科学而不是奇淫巧计，不要学点方法就以为拿到了打开财富之门的秘诀，只有打好基础，才可能靠谱地做量化。如果刚刚接触量化，推荐两本书，有利于建立对量化交易的基本认识：

* 《宽客：华尔街顶级数量金融大师的另类人生》
* 《宽客人生：从物理学家到数量金融大师的传奇》

根据美国量化发展的经验，成功的策略都是由精英团队经过多年研究才能盈利，盈利之后仍然需要精英团队来运维。vn.py能够为您解决平台的问题，使您将更多的精力放到知识上。本文档帮助您更有效地使用vn.py平台。

## vn.py的安装与运行

### 源码下载与安装

vn.py的官网是http://www.vnpy.com/。

主页上增加了“Gitee仓库”的下载链接，建议使用这个，下载速度要快得多。

执行D:\vnpy220中的install.bat批处理文件进行安装，自动下载安装需要的包。

安装需要一定的经验，以下是一些安装经验的记录：

* 强烈建议参考书籍第6章的补充内容，使用镜像站点，可以极大地加快pip的下载速度。下载速度快了，一次安装成功的概率会大很多。
* 安装的时候需要大量的网络下载，最好选择网络较好的环境。还与人家网站的情况有关，所以不一定能一次安装成功。一次不成功就再试一次，今天不成功就等明天再试试看。
* 一遍一遍地试也不是个事儿。如果是有经验的Python编程人员，可以根据安装时的报错信息，手动补充安装需要的包，特别是在Win7下，有些包可能根本就不能自动安装。
* 在Win10上安装更容易成功。在Win7上安装，需要手工安装一些包，有些甚至不能用pip直接安装，需要下载轮子之类的东西再手动安装。
* Python的版本太高也不好，可能与vn.py的DLL有冲突。据说如果非要用高版本的Python，需要将CTP的DLL重新编译。
* 推荐的方法：安装Anaconda3，在Anaconda3上使用“conda create -n vnpy220 python=3.7.6”命令，创建一个名为vnpy220的虚拟环境，然后切换到新的虚拟环境中进行安装及今后的定制开发。
* vn.py的2.2.0版本，用最新版的PyCharm，将vn.py源目录作为项目打开，如果没事先选择虚拟环境，默认BASE环境，PyCharm会自动对虚拟环境进行配置，使vn.py可执行。退出PyCharm，在Anaconda Prompt下也就能直接运行了。
* D:\vnpy220中的文件requirements.txt中列出了所有需要安装的第三方包，有些还指明了版本。您可以对照pip list的结果，把这些包先分别装好，然后再执行install.bat，会减少一些重复工作。

### vn.py的启动

vn.py提供的源码中没有主程序。vn.py在启动过程中可以只加载必要的模块，因此需要使用者自己根据需要创建主程序。在D:\vnpy220中创建一个新的Python文件run.py，代码为：

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

from vnpy.app.data\_manager import DataManagerApp

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\_engine.add\_app(DataManagerApp)

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

main\_window.showMaximized()

qapp.exec()

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

main()

以上是书中给出的示例主程序，但随着vn.py版本的不同，建议的主程序略有差别，可以在下载页面上找到。

随着用户使用经验的积累，可能需要在系统中加载不同的模块，这就需要对主程序进行修改定制。比如本书给出的示例中增加了数据库管理器的加载。

打开CMD，转到D:\vnpy220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:\vnpy220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:\vnpy220\vnpy\gateway\da\\_\_init\_\_.py", line 1, in <module>

from .da\_gateway import DaGateway

File "D:\vnpy220\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。

教材推荐使用PyCharm，把D:\vnpy220目录作为一个项目打开，执行其中的run.py，可正常运行。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。如果是Anaconda环境，也可以使用Anaconda自带的Spyder。

## VN Trader

### 准备交易账号

打开SimNow官网http://www.simnow.com.cn，单击右上角的“注册账号”，填写一些基础信息完成注册。注册完成后，回到SimNow首页点击右上角的“投资者登录”，输入手机号和密码登录进去。

请牢记investorId才是您SimNow环境的CTP用户名，而不是登录网站用的手机号！同时CTP密码则就是您登录网站用的密码。

接下来需要修改一次密码才能使用API进行交易，是的，刚注册就要改。

### 交易接口登录

交易和行情服务器一共有三组选择，前两组只能在交易时段登录（周一到周五，日盘和夜盘时段），提供和实盘环境一致的行情和撮合。

**选择1（对应SimNow第一套第二组）：**

* 交易服务器：180.168.146.187:10101
* 行情服务器：180.168.146.187:10111

**选择2（对应SimNow第一套第三组）：**

* 交易服务器：218.202.237.33:10102
* 行情服务器：218.202.237.33:10112

第三组则是只能在非交易时段登录，提供最近交易时段行情的回放和撮合。

**选择3（对应SimNow第二套）：**

* 交易服务器：180.168.146.187:10130
* 行情服务器：180.168.146.187:10131

### 实盘交易

当您已经对SimNow的仿真测试环境足够熟悉后，可能已经做好了使用CTP柜台进行实盘交易的准备。对于CTP实盘交易：

* 用户名和密码，就是您开户后直接拿到的信息
* 经纪商编号和交易行情服务器地址，可以联系您的客户经理获取
* 产品名称和授权编码，需要完成穿透式认证获取

## CTA回测

策略写好后，下一步就是回测：把历史上的价格数据（K线或者Tick），推送给策略去运行交易逻辑，并把策略产生的交易记录下来，然后分析这些回测的交易记录，从而判断该策略的潜在盈利能力。

在开始之前，先讲几个量化策略研究中（不管是否用vn.py）需要记住的重要原则：

* 所有量化程序的回测功能，永远都只能尽量接近实盘交易中的各项细节，而无法做到100%一样，关键点在于误差的大小（是否能容忍）；
* 回测效果好的策略，并不代表实盘交易就一定盈利，可能存在交易成本误差、参数过度拟合、策略逻辑有未来函数或者市场特征变化等原因；
* 回测效果烂的策略，实盘交易基本可以保证会更烂，绝对不要抱有侥幸心理。

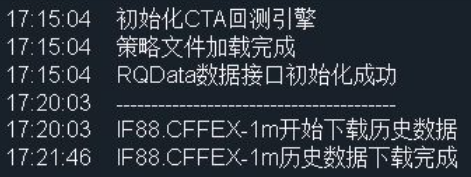
### 获取行情数据

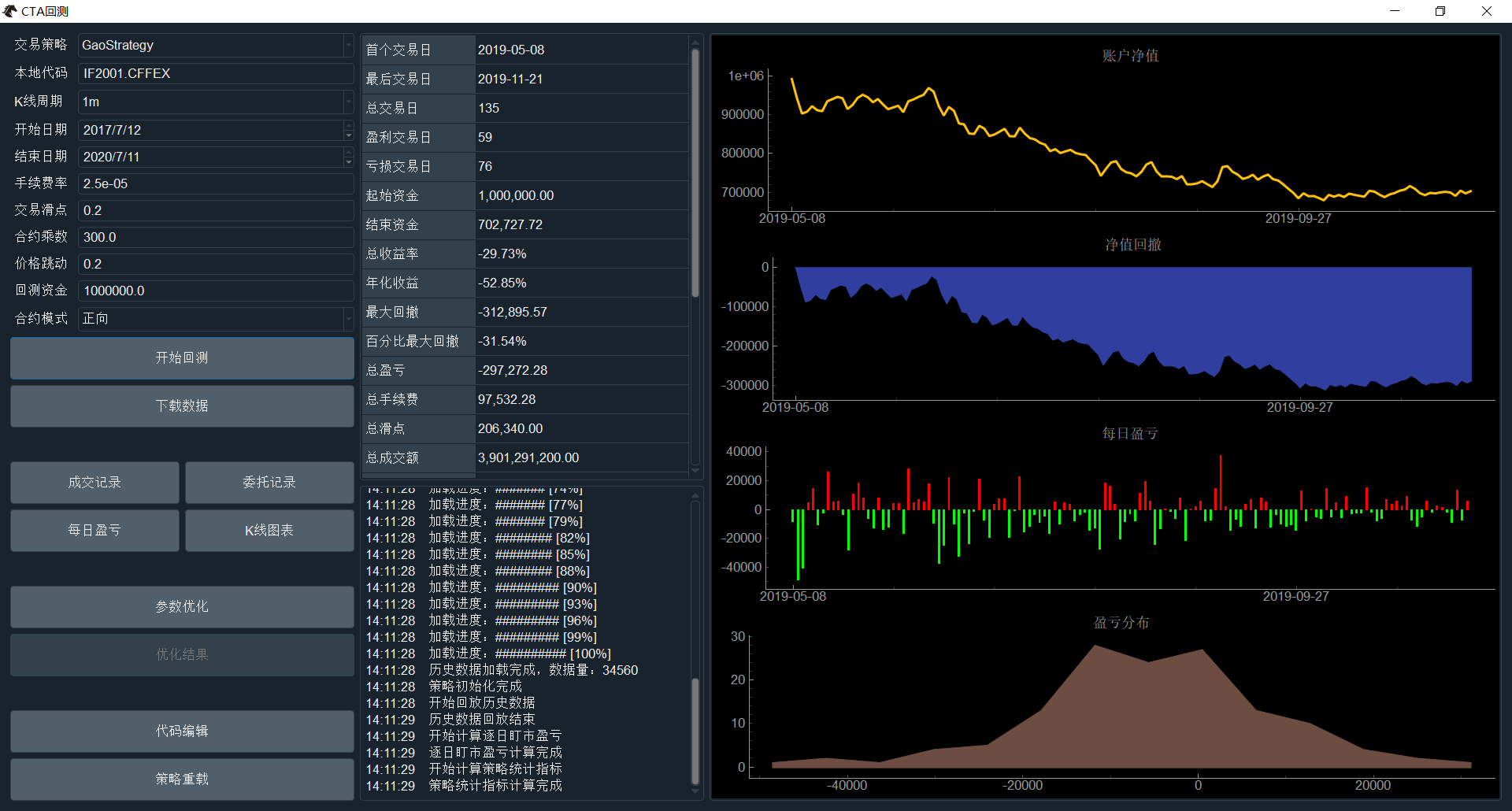
要跑历史数据回测，第一步就是要先准备好历史数据。以国内期货数据为例，vn.py默认使用米筐的RQData来下载获取。

RQData目前提供30天的免费试用期，网站申请非常方便。

申请成功后，运行VN Trader，在主界面顶部的菜单栏，选择“配置”菜单，打开VN Trader的全局配置对话框。将之前已经准备好的RQData用户名和密码，分别填入到rqdata.username和rqdata.password两个字段中，然后点击“确定”按钮，弹出提示重启的对话框。

此时即可关闭VN Trader并重启。重启后点击菜单栏“功能 - CTA回测”，启动接下来要用到的CTA策略回测图形界面。如果上一步的RQData账号密码配置正确，此时可以在中间底部的日志输出框中看到“RQData数据接口初始化成功”的信息。如果没有看到就说明配置有问题，回去重来吧。





窗口左上方的一系列编辑框和下拉框，用来控制和管理回测功能。在本地代码编辑框中输入IF2001.CFFEX，K线周期选择1m（即1分钟K线），然后选择要下载数据的开始日期和结束日期，点击“下载数据”按钮。

此时CtaBacktester模块就会自动从RQData服务器下载历史数据，在完成数据结构转化后插入到VN Trader的数据库中（默认使用SQLite，数据文件位于.vntrader目录下的database.db），下载完成后同样会在日志输出框中看到相应信息。

# VN Trader分析

对应《Python量化交易从入门到实战》的第16章。

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，我们将在本文档第二部分对VN Trader进行深入分析。

本章学习方法：按教材学习，本文档没有补充更多内容。

## 主引擎

### vn.py体系结构

注：本小节内容教材中也有。因为这一小节太重要，所以复制过来供不方便买书的同学参考。

分层是解决复杂编程问题的常用方法，将不同的功能放在不同的层次上。每个层次上的模块只调用下层功能，并对上层提供接口。vn.py的体系结构分为三层，自上至下分别是：应用层、引擎层和接口层，如图16-1所示。



图16-1 vn.py的体系结构

接口层负责对接行情和交易API，将行情数据推送到系统，发送交易指令(如下单、数据请求)等。按照vn.py官网的说法，vn.py目前支持CTP、中泰XTP等超过30种交易接口。vn.py的底层接口程序在D:\vnpy220\vnpy\api目录下，每个子目录对应一种接口。

引擎层包括主引擎中的事件引擎和功能引擎等公共引擎，还包括CTA回测、CTA策略等专用引擎。向下对接各种交易接口，向上服务于各种上层应用。中层引擎将系统的各个组成部分结合成一个有机整体。除各类引擎外，数据库管理器为系统提供基础的数据存储功能，被各个引擎所调用。在vn.py的早期版本中，数据库管理器被实现为数据库引擎，也在引擎层与其他引擎并列，但这样就形成了同层之间的调用，会造成层次关系的混乱。现在将其实现为公共的数据库管理器模块，可以看做是处于引擎层底部的一个子层，为所有引擎提供服务。

在引擎层，主引擎是整个系统的核心，而事件引擎又是主引擎乃至整个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。本节介绍主引擎，事件引擎将在教材18.1节介绍。

应用层至少包含GUI图形界面模块VN Trader，其他可选的上层应用包括CTA回测、CTA策略、数据管理、价差套利、算法交易、期权策略、脚本策略、交易风险管理、投资组合模块和RPC服务模块等。这些应用是系统与用户的接口，为用户直接提供功能。

# 数据库操作

对应《Python量化交易从入门到实战》的第17章。

正如教材中所预言：“数据管理作为发展中的功能，在后续的版本中很可能会不断优化，当读者阅读本书时，数据管理功能很可能已经不是现在的样子。”

本章学习方法：

如果使用vn.py 2.1.4，可以教材为主线进行学习，同时对照参考《vn.py 2.1.4源代码深入分析》。

从vn.py的2.1.9版本开始重构了数据库管理模块。如果使用vn.py的最新版本，由于本部分功能变化较大，可按照本文档进行学习。本文档复制了教材对应章节的完整结构，并进行了改进和补充。

如果条件允许，建议使用新版本软件进行学习。老版本的数据库管理是新功能，体系结构不合理，程序很绕，不容易看懂。大道至简，新版本在功能增强的基础上程序更加简单清晰，不但执行速度快，程序读起来也更容易。

交易平台离不开行情数据。回测时需要回放历史数据，实盘时除了接收实时行情外，也需要回放一定数量的历史数据。本书不是介绍交易本身的书籍，作为技术书籍，本章以数据库为核心，对行情数据的处理进行介绍。因此本章称为“数据库操作”，淡化行情的概念。

vn.py默认从ricequant(米筐)下载行情数据，也可以从CSV格式的行情数据文件中提取数据。获得的行情数据被存储到数据库中，需要时取出使用。关于行情数据是放在数据库中还是放在文件中好？这个问题历来有争论。具体到vn.py这个应用场合，只保存部分合约、部分周期、部分时段的行情数据，使用数据库是恰当的。

## vn.py支持的数据库

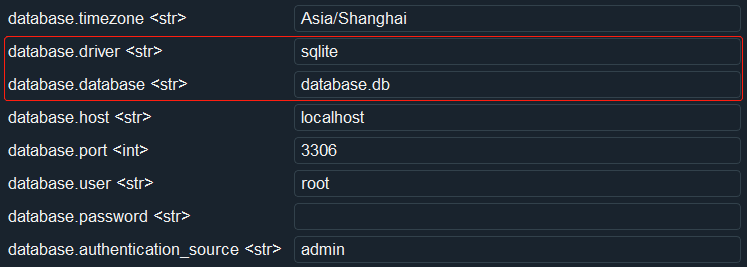
vn.py既支持SQL数据库，包括SQLite、MySQL和PostgreSQL，也支持NoSQL数据库，包括MongoDB。

教材编写时vn.py只支持上述4类数据库。从vn.py的2.1.9开始，重构了几大数据库管理模块，实现了更好的数据库读写性能和未来的新数据库扩展支持，所支持的数据库也增加到了5种，包括：

* SQL类
  + SQLite（sqlite）：轻量级单文件数据库，无需安装和配置数据服务程序，vn.py的默认选项，适合入门新手用户；
  + MySQL（mysql）：世界最流行的开源关系型数据库，文档资料极为丰富，且可替换其他高NewSQL兼容实现（如TiDB）；
  + PostgreSQL（postgresql）：特性更为丰富的开源关系型数据库，支持通过扩展插件来新增功能，只推荐熟手使用；
* NoSQL类
  + MongoDB（mongodb）：基于分布式文件储存（bson格式）的非关系型数据库，内置的热数据内存缓存实现更快读写速度；
  + InfluxDB（influxdb）：针对时序数据专门设计的非关系型数据库，列式数据储存提供极高的读写效率和外围分析应用。

增加了一类NoSQL数据库InfluxDB，估计后续版本很快还会增加对其它数据库的支持。

先看关于数据库的全局配置。在主菜单中选择“配置”，打开“全局配置”功能。与数据库相关的配置如下图所示。



上图表示默认数据库是SQLite，数据库文件是C:\Users\admin\.vntrader目录下的database.db文件。在您的计算机中，路径可能会有所不同，比如其中的admin应该换成您的系统用户名，参教材16.2节关于“工作目录”的说明。

SQLite3数据库只需要配置这两个字段，如果使用其它种类的数据库，还需要配置host和port等几个字段。

当前版本的vn.py默认使用SQLite数据库，下面就以SQLite为例介绍vn.py的数据库操作，教材9.4和9.5节介绍的知识将在本章得到应用。

用SQLite的可视化工具，如SQLiteStudio，打开C:\Users\admin\.vntrader目录下的database.db。可以看到库中有三个表：dbbardata、dbbaroverview和dbtickdata。注：早期版本只有两个表，没有概览信息表。

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,

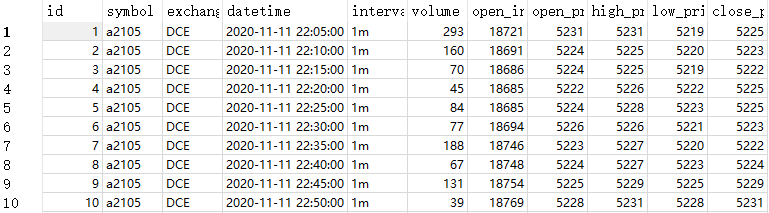
high\_price REAL NOT NULL,

low\_price REAL NOT NULL,

close\_price REAL NOT NULL

);

在symbol、exchange、interval和datetime这四个字段上建立了复合索引，其部分数据如下图。



dbbaroverview存放K线数据的概览信息，结构如下：

CREATE TABLE dbbaroverview (

id INTEGER NOT NULL

PRIMARY KEY,

symbol VARCHAR (255) NOT NULL,

exchange VARCHAR (255) NOT NULL,

interval VARCHAR (255) NOT NULL,

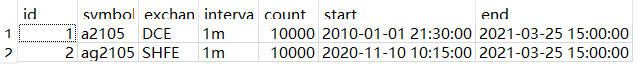
count INTEGER NOT NULL,

start DATETIME NOT NULL,

[end] DATETIME NOT NULL

);

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

将上述Tick数据表结构与CTP接口的Tick数据结构进行对比（关于CTP接口的说明见本文档第二部分）：

///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;

};

可以看出，两者的结构非常相似，说明该结构具有通用性。

## 下载数据(补充内容)

vn.py默认的行情数据接口是米筐的rqdata。但rqdata的免费试用期只有1个月，严重不够。在学习期就要为行情付费，不合适。

本节讨论如何编程从新浪财经或聚宽（JoinQuant）等来源获取行情数据，并保存到CSV格式文件。

### 从新浪财经获取行情数据

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。

创建Python程序文件DownFromSina.py，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。

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。

从新浪财经下载数据比较简单，但新浪财经目前不提供1分钟级别的行情数据，使用时受限。

### 从聚宽获取行情数据

JQData是聚宽数据团队专门为金融机构、学术团体和量化研究者们提供的本地量化金融数据服务。使用JQData，可快速查看和计算金融数据，无障碍解决本地、Web、金融终端调用数据的需求。

目前JQData的免费试用期是1年，对于学习来说足够了。

创建Python程序文件DownFromJQ.py，代码如下：

import pandas as pd

import jqdatasdk

def get\_data(id):

jqdatasdk.auth('username','password')

df=jqdatasdk.get\_price(id, start\_date="2019-4-22", end\_date="2019-11-22", frequency='minute', fields=['close', 'high', 'low', 'open', 'volume'])

df["Datetime"] = df.index

df = df.loc[:,["Datetime",'close', 'high', 'low', 'open', 'volume']]

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

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

get\_data('IF2001.CCFX')

#get\_data('RB2001.XSGE')

注意：

* JQData需要执行“pip install jqdatasdk”进行安装。jqdatasdk的详细说明见：<https://www.joinquant.com/help/api/help?name=JQData>。
* 安装完成后，在有的IDE中可能找不到jqdatasdk。可以在NotePad++中，或者在原生的Python环境中执行程序。
* 免费试用，JQData有每天100万条的下载限制。
* 程序中的用户名和密码需要自行申请。
* 还要根据需要修改合约代码及起止时间。如果起始时间早于合约的上市日，可能会得到很多空数据，造成回测时加载数据失败。
* 上述程序我测试得比较早，现在试用到期了我就没再试。如果有问题，请同学们自行学习修改。

与本节介绍的其他方法相比，使用JQData有一个明显的好处，就是JQData的调用方法与rqdata非常相似，只要对vn.py的代码稍加修改，就能改成使用JQData。

vn.py的rqdata接口在D:\vnpy220\vnpy\trader的rqdata.py文件中定义。下面代码是针对vn.py的2.0.7版本所做的修改，如果是2.0.7版，用下面代码覆盖rqdata.py的内容即可正常执行（这样做对原系统的影响最小）。2.2.0版该文件的内容稍微复杂一些直接覆盖可能不行，但还是那个原因，我的试用到期了，就没针对2.2.0版进行修改，请读者自行修改测试吧。

from datetime import datetime, timedelta

from typing import List

from jqdatasdk import auth as rqdata\_init

from jqdatasdk import get\_all\_securities as rqdata\_all\_instruments

from jqdatasdk import get\_price as rqdata\_get\_price

#from rqdatac import init as rqdata\_init

#from rqdatac.services.basic import all\_instruments as rqdata\_all\_instruments

#from rqdatac.services.get\_price import get\_price as rqdata\_get\_price

#from rqdatac.share.errors import AuthenticationFailed

from .setting import SETTINGS

from .constant import Exchange, Interval

from .object import BarData, HistoryRequest

INTERVAL\_VT2RQ = {

Interval.MINUTE: "1m",

Interval.HOUR: "60m",

Interval.DAILY: "1d",

}

INTERVAL\_ADJUSTMENT\_MAP = {

Interval.MINUTE: timedelta(minutes=1),

Interval.HOUR: timedelta(hours=1),

Interval.DAILY: timedelta() # no need to adjust for daily bar

}

class RqdataClient:

"""

Client for querying history data from RQData.

"""

def \_\_init\_\_(self):

""""""

self.username = SETTINGS["rqdata.username"]

self.password = SETTINGS["rqdata.password"]

self.inited = False

self.symbols = set()

def init(self, username="", password=""):

""""""

if self.inited:

return True

if username and password:

self.username = username

self.password = password

if not self.username or not self.password:

return False

rqdata\_init(

self.username,

self.password

)

try:

df = rqdata\_all\_instruments(['futures'], date=datetime.now())

for ix, row in df.iterrows():

self.symbols.add(row[1])

except Exception:

return False

self.inited = True

return True

def to\_rq\_symbol(self, symbol: str, exchange: Exchange):

"""

CZCE product of RQData has symbol like "TA1905" while

vt symbol is "TA905.CZCE" so need to add "1" in symbol.

"""

if exchange == Exchange.CFFEX:

rq\_symbol = f"{symbol.upper()}.CCFX"

elif exchange == Exchange.SHFE:

rq\_symbol = f"{symbol.upper()}.XSGE"

elif exchange == Exchange.CZCE:

rq\_symbol = f"{symbol.upper()}.XZCE"

elif exchange == Exchange.DCE:

rq\_symbol = f"{symbol.upper()}.XDCE"

return rq\_symbol

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

"""

Query history bar data from RQData.

"""

symbol = req.symbol

exchange = req.exchange

interval = req.interval

start = req.start

end = req.end

if symbol not in self.symbols:

return None

rq\_interval = INTERVAL\_VT2RQ.get(interval)

if not rq\_interval:

return None

# For adjust timestamp from bar close point (RQData) to open point (VN Trader)

adjustment = INTERVAL\_ADJUSTMENT\_MAP[interval]

# For querying night trading period data

end += timedelta(1)

rq\_symbol = self.to\_rq\_symbol(symbol, exchange)

df = rqdata\_get\_price(

rq\_symbol,

frequency=rq\_interval,

fields=["open", "high", "low", "close", "volume"],

start\_date=start,

end\_date=end

)

data: List[BarData] = []

if df is not None:

for ix, row in df.iterrows():

bar = BarData(

symbol=symbol,

exchange=exchange,

interval=interval,

datetime=row.name.to\_pydatetime() - adjustment,

open\_price=row["open"],

high\_price=row["high"],

low\_price=row["low"],

close\_price=row["close"],

volume=row["volume"],

gateway\_name="RQ"

)

data.append(bar)

return data

rqdata\_client = RqdataClient()

### 从Tushere Pro取行情数据

Tushere也不提供1分钟级别的行情数据，但新版的Tushere Pro是可以的。按照Tushere官网的说法：“在Tushare旧版运行了3年后，Tushare Pro在大家的期待下，终于要跟大家见面了。可以略显激动地说，Pr版数据更稳定质量更好了，但Pro依然是个开放的，免费的平台，不带任何商业性质和目的。”

Tushare Pro的用法与旧版有所不同，现在Tushare已经开始对旧版给出提示：“本接口即将停止更新，请尽快使用Pro版接口：<https://waditu.com/document/2>。”

与老版本不同，Tushare Pro增加了一些限制，比如只有Tushare社区用户才能使用Tushare Pro，请使用下面链接注册：

<https://tushare.pro/register?reg=406959>

教材中多处用到Tushere下载行情数据，但使用的是旧版。Tushare Pro的用法可到上述链接页面中学习，我们也针对教材和本节下载行情数据的需要，编写了文档《Tushere Pro使用说明》。如果读者想继续使用Tushare Pro下载行情数据，可参考该文档，本节不再重复。

### 其它方法

行情数据必不可少，怎样获得各有高招儿。前文介绍了使用Python与vn.py容易结合的方法。

对于具有一定编程基础的人，可以方便地通过其它方法获得行情数据。事实上，只要能通过其它方法得到行情数据的CSV文件，就可以通过vn.py的数据库管理器导入到系统当中。

我使用自编的软件“普吸金 - 缠论/波浪理论分析/训练软件(2.0)”（可以到QQ群881645236下载）从通达信中获取数据并转换成CSV文件。这种方法还是比较麻烦的，vn.py开发团队在社区动向帖“终极解决方案：再也不为量化数据而烦恼（<https://www.vnpy.com/forum/topic/10-zhong-ji-jie-jue-fang-an-:zai-ye-bu-wei-liang-hua-shu-ju-er-fan-nao>）”中对此观点有论证，总之vn.py建议大家使用rqdata。

## 数据库管理器

新版本的数据库管理器采用类似gateway（底层接口）和app（上层应用）的设计模式。与数据库管理有关的程序主要在三个地方，如下图所示。

vnpy/

├─app/ 注：第三部分

│ ├─data\_manager/

│ │ ├─ui/

│ │ │ ├─\_\_init\_\_.py

│ │ │ └─widget.py

│ │ ├─\_\_init\_\_.py

│ │ └─engine.py

│ ...

├─database/ 注：第二部分

│ ├─influxdb/

│ │ ├─\_\_init\_\_.py

│ │ └─influxdb\_database.py

│ ├─mongodb/

│ │ ├─\_\_init\_\_.py

│ │ └─mongodb\_database.py

│ ├─mysql/

│ │ ├─\_\_init\_\_.py

│ │ └─mysql\_database.py

│ ├─postgresql/

│ │ ├─\_\_init\_\_.py

│ │ └─postgresql\_database.py

│ ├─sqlite/

│ │ ├─\_\_init\_\_.py

│ │ └─sqlite\_database.py

│ └─\_\_init\_\_.py

│ ...

├─trader/

│ ├─database.py 注：第一部分

... ...

第一部分：vnpy.trader.database模块

在D:\vnpy220\vnpy\trader目录下的database.py文件中，定义数据库管理的通用接口，包括：

* 抽象模板类BaseDatabase
* 数据库时区常量DB\_TZ
* 时区转换函数convert\_tz
* 以及K线数据整体概况BarOverview类，用于大幅提高DataManager组件的数据库概况查询速度

第二部分：vnpy.database子包

在D:\vnpy220\vnpy\database目录中定义。

该子包下又有5个子包，每个子包对应一种数据库，例如与SQLite有关的操作在D:\vnpy220\vnpy\database\sqlite下的sqlite\_database.py中定义。每种数据库都从BaseDatabase继承，实现具体的数据库管理功能。

数据库管理器类在第一第二部分中定义，文件和类及其继承关系如图17-1（按教材中的图编号）所示。



图17-1 数据库管理器类

第三部分：vnpy.app.data\_manager子包

在D:\vnpy220\vnpy\app\data\_manager中定义，实现数据管理的界面。

### 数据库管理器基类

本小节对应上述第一部分代码。

数据库管理器的基类BaseDatabase在D:\vnpy220\vnpy\trader下的database.py中定义。在介绍BaseDatabase之前，先做一些准备工作。

为走向国际化，vn.py在新的数据管理中增加了时区转换功能，相关代码为：

from datetime import datetime

# 从全局配置信息中取得时区

DB\_TZ = timezone(SETTINGS["database.timezone"])

# 时区转换函数

def convert\_tz(dt: datetime) -> datetime:

"""

将datetime对象dt的时区改为DB\_TZ。

"""

dt = dt.astimezone(DB\_TZ)

return dt.replace(tzinfo=None)

为了方便统计和显示K线数据的概览信息，新的数据管理中增加了一个BarOverview类，用于存储K线数据的概览信息，相关代码为：

@dataclass

class BarOverview:

"""

数据库中K线数据的概览信息，用于在“数据管理”应用左侧的列表中显示

"""

symbol: str = ""

exchange: Exchange = None

interval: Interval = None

count: int = 0

start: datetime = None

end: datetime = None

BaseDatabase是一个抽象类(抽象类概念见教材7.5节)，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数据库管理器类共同的基类，其部分代码为

class BaseDatabase(ABC):

"""

数据库基类，用于连接到不同的数据库

"""

@abstractmethod

def save\_bar\_data(self, bars: List[BarData]) -> bool:

"""

将一批K线数据保存到数据库

"""

pass

@abstractmethod

def save\_tick\_data(self, ticks: List[TickData]) -> bool:

"""

将一批Tick数据保存到数据库

"""

pass

@abstractmethod

def load\_bar\_data(

self,

symbol: str,

exchange: Exchange,

interval: Interval,

start: datetime,

end: datetime

) -> List[BarData]:

"""

取某个合约某周期某段时间的K线数据

"""

pass

@abstractmethod

def load\_tick\_data(

self,

symbol: str,

exchange: Exchange,

start: datetime,

end: datetime

) -> List[TickData]:

"""

取某个合约某周期某段时间的Tick数据

"""

pass

@abstractmethod

def delete\_bar\_data(

self,

symbol: str,

exchange: Exchange,

interval: Interval

) -> int:

"""

从数据库中删除某个合约某周期的所有K线数据

"""

pass

@abstractmethod

def delete\_tick\_data(

self,

symbol: str,

exchange: Exchange

) -> int:

"""

从数据库中删除某个合约的所有Tick数据

"""

pass

@abstractmethod

def get\_bar\_overview(self) -> List[BarOverview]:

"""

取数据库中所有K线数据的概览信息

"""

pass

# 从全局配置信息中取数据库类型

driver: str = SETTINGS["database.driver"]

# 生成该类数据库的模块名称

module\_name: str = f"vnpy.database.{driver}"

try:

# 根据模块名称取该类数据库的管理器

database\_manager: BaseDatabase = import\_module(module\_name).database\_manager

except ModuleNotFoundError:

# 如果全局配置信息有误，默认使用SQLite数据库

print(f"找不到数据库驱动{module\_name}，使用默认的SQLite数据库")

database\_manager: BaseDatabase = import\_module("vnpy.database.sqlite").database\_manager

后面将会看到，在每种数据库的模块（如sqlite\_database.py）中都会定义一个变量database\_manager。上述代码从相应模块中取得该变量，并赋给一个全局变量database\_manager。在需要使用数据库的主调模块中，使用如下语句取得该全局变量：

from vnpy.trader.database import database\_manager

然后就可以通过该变量操作数据库。教材“图16-1 vn.py的体系结构”中的“数据库管理器”模块对应此变量。此变量不同于一般的数值变量，它是一个完整的数据库操作模块。

### SQLite数据库管理器

本小节对应上述第二部分代码。

BaseDatabase是抽象类，所有数据库管理器都从该抽象类继承。如SQLite数据库管理器类SqliteDatabase，在D:\vnpy220\vnpy\database\sqlite下的sqlite\_database.py中定义，继承自BaseDatabase，用于操作SQLite数据库。该模块的定义分为两部分，第一部分定义peewee模板类，第二部分定义SQLite数据库类。

#### peewee模板类

vn.py在操作SQL数据库时使用peewee，包括SQLite、MySQL和PostgreSQL等，要先为三个库表定义peewee模板类，相关代码为：

# 取得数据库文件名

path = str(get\_file\_path("database.db"))

# 连接数据库

db = PeeweeSqliteDatabase(path)

# 注：教材中连接数据库用的是peewee的SqliteDatabase()函数。

# 注意在本程序开关的import语句中已将SqliteDatabase()函数重名句

# 为PeeweeSqliteDatabase()函数。

class DbBarData(Model):

"""K线数据的peewee模板类"""

id = AutoField()

symbol: str = CharField()

exchange: str = CharField()

datetime: datetime = DateTimeField()

interval: str = CharField()

volume: float = FloatField()

open\_interest: float = FloatField()

open\_price: float = FloatField()

high\_price: float = FloatField()

low\_price: float = FloatField()

close\_price: float = FloatField()

class Meta:

database = db

indexes = ((("symbol", "exchange", "interval", "datetime"), True),)

# 可以看到，DbBarData类的属性与前述数据库表dbbardata的字段完全对应。

class DbTickData(Model):

"""Tick数据的peewee模板类"""

id = AutoField()

symbol: str = CharField()

exchange: str = CharField()

datetime: datetime = DateTimeField()

name: str = CharField()

volume: float = FloatField()

open\_interest: float = FloatField()

last\_price: float = FloatField()

last\_volume: float = FloatField()

limit\_up: float = FloatField()

limit\_down: float = FloatField()

open\_price: float = FloatField()

high\_price: float = FloatField()

low\_price: float = FloatField()

pre\_close: float = FloatField()

bid\_price\_1: float = FloatField()

bid\_price\_2: float = FloatField(null=True)

bid\_price\_3: float = FloatField(null=True)

bid\_price\_4: float = FloatField(null=True)

bid\_price\_5: float = FloatField(null=True)

ask\_price\_1: float = FloatField()

ask\_price\_2: float = FloatField(null=True)

ask\_price\_3: float = FloatField(null=True)

ask\_price\_4: float = FloatField(null=True)

ask\_price\_5: float = FloatField(null=True)

bid\_volume\_1: float = FloatField()

bid\_volume\_2: float = FloatField(null=True)

bid\_volume\_3: float = FloatField(null=True)

bid\_volume\_4: float = FloatField(null=True)

bid\_volume\_5: float = FloatField(null=True)

ask\_volume\_1: float = FloatField()

ask\_volume\_2: float = FloatField(null=True)

ask\_volume\_3: float = FloatField(null=True)

ask\_volume\_4: float = FloatField(null=True)

ask\_volume\_5: float = FloatField(null=True)

class Meta:

database = db

indexes = ((("symbol", "exchange", "datetime"), True),)

# 可以看到，DbTickData类的属性与前述数据库表dbtickdata的字段完全对应。

class DbBarOverview(Model):

"""K线数据的概览信息peewee模板类"""

id = AutoField()

symbol: str = CharField()

exchange: str = CharField()

interval: str = CharField()

count: int = IntegerField()

start: datetime = DateTimeField()

end: datetime = DateTimeField()

class Meta:

database = db

indexes = ((("symbol", "exchange", "interval"), True),)

# 用于在“数据管理”应用左侧的列表中显示

为每个表定义一个继承自Model的模板类，比如DbBarData类的属性与前述数据库表dbbardata的字段完全对应，其它模板类也是如此。

#### SQLite数据库类

SQLite数据库类SqliteDatabase实现BaseDatabase中所有的抽象方法，都使用最典型最基本的peewee实现方法，部分方法将在后续章节(如教材17.3.4节，本文档3.4.4节)涉及到时进行自顶向下的分析，未涉及到的方法请读者自行分析。 SQLite数据库类的框架代码如下：

class SqliteDatabase(BaseDatabase):

"""继承自BaseDatabase的Sqlite数据库类"""

def \_\_init\_\_(self) -> None:

"""初始化"""

# 连接数据库

self.db = db

self.db.connect()

# 根据需要创建数据库表

self.db.create\_tables([DbBarData, DbTickData, DbBarOverview])

def save\_bar\_data(self, bars: List[BarData]) -> bool:

"""将一批K线数据保存到数据库"""

……

def save\_tick\_data(self, ticks: List[TickData]) -> bool:

"""将一批Tick数据保存到数据库"""

……

def load\_bar\_data(

self,

symbol: str,

exchange: Exchange,

interval: Interval,

start: datetime,

end: datetime

) -> List[BarData]:

"""取某个合约某周期某段时间的K线数据"""

……

def load\_tick\_data(

self,

symbol: str,

exchange: Exchange,

start: datetime,

end: datetime

) -> List[TickData]:

"""取某个合约某周期某段时间的Tick数据"""

……

def delete\_bar\_data(

self,

symbol: str,

exchange: Exchange,

interval: Interval

) -> int:

"""从数据库中删除某个合约某周期的所有K线数据"""

……

def delete\_tick\_data(

self,

symbol: str,

exchange: Exchange

) -> int:

"""从数据库中删除某个合约的所有Tick数据"""

……

def get\_bar\_overview(self) -> List[BarOverview]:

"""

取数据库中所有K线数据的概览信息

"""

……

def init\_bar\_overview(self) -> None:

"""

初始化K线数据的概览信息

"""

……

# 创建一个全局变量database\_manager，供主调模块使用

database\_manager = SqliteDatabase()

## 数据管理

本节对应上述第三部分代码。

数据管理是vn.py一个新的上层应用，它将原来的CSV载入等功能组合在一起，形成一个综合的数据管理平台。启动VN Trader，在菜单中选择“功能 - 数据管理”，或者单击左侧功能导航栏的数据库按钮，启动“数据管理”上层应用，界面如图17-2所示。本节不对数据管理做全面介绍，仅介绍其中与数据库操作相关的部分。

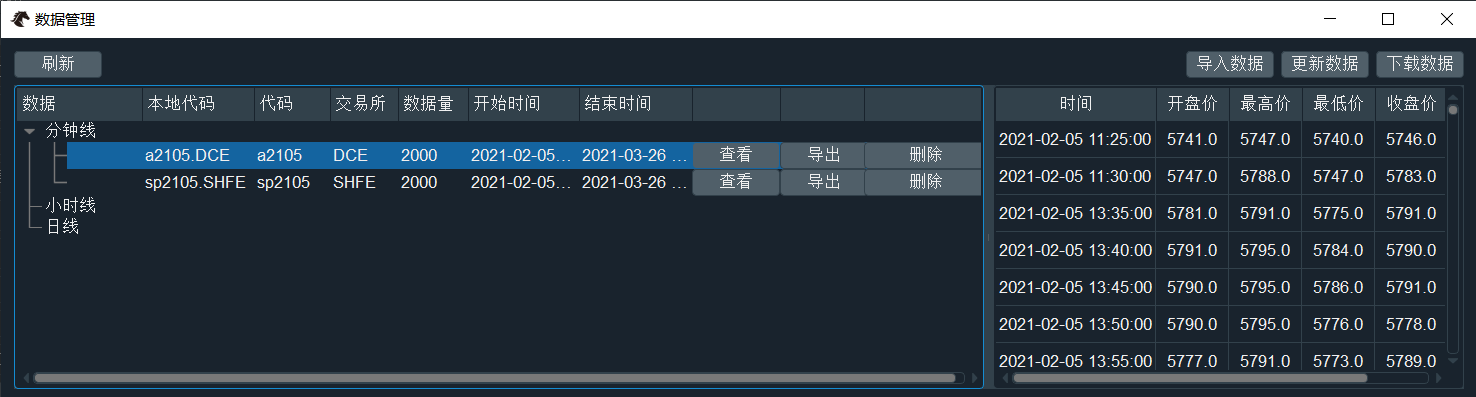


图17-2 “数据管理”上层应用

vn.py的维护团队是一个非常有朝气的团队，版本更新很快。而数据管理作为发展中的功能，在后续的版本中很可能会不断优化。数据管理虽然是新功能，但其体系结构在后续版本中应该不会大改，如图17-3所示，该结构其实就是图16-1所示vn.py体系结构的局部。



图17-3 数据管理的体系结构

在图17-3的应用层包括数据管理应用类和数据管理窗体类，引擎层包括数据管理引擎类。数据库层的数据库管理器类教材17.2节（本文档3.3节）已经介绍，本节介绍应用层和引擎层的实现方法。

### 数据管理应用类

请读者先复习教材16.5节内容，回顾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功能引擎相关联。

现在看数据管理上层应用类的定义。与CTA回测上层应用类相同，DataManagerApp类同样继承自BaseApp，在D:\vnpy220\vnpy\app\data\_manager目录下的\_\_init\_\_.py中定义，代码如下：

class DataManagerApp(BaseApp):

app\_name = APP\_NAME

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

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

display\_name = "数据管理"

engine\_class = ManagerEngine

widget\_name = "ManagerWidget"

icon\_name = "manager.ico"

可以看到，数据管理的引擎类是ManagerEngine，窗体类是ManagerWidget。

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，具体方法是在程序主函数中执行：

main\_engine.add\_app(DataManagerApp)

### 数据管理窗体类

“数据管理”的窗体类在D:\vnpy220\vnpy\app\data\_manager\ui目录下的widget.py中定义。

界面的左侧是一个树型的行情分类列表，与教材12.8节的树型列表相同。列表的内容是数据库中行情数据的种类，分为分钟线、小时线和日线三大类，每类当中又按合约继续分类。该树型列表的初始化工作由init\_tree和init\_child两个方法完成。

开始时左侧列表为空，按“刷新”按钮，可根据对数据库中数据的统计结果刷新列表。如果在左侧列表中选中某类行情，按该类行情的“查看”按钮，会将该类行情逐条显示在右侧的行情数据表格中。右侧表格由init\_table方法初始化。

widget.py是典型的PyQt5代码，组织方式比主窗口简单，不再详述。

### 下载数据

按“下载数据”按钮可从网上下载行情数据。



教材8.3和14.6节已经讨论过如何从Tushare获取历史行情数据，但Tushare不提供1分钟级别的行情数据。除Tushare之外，还可以从其他多个来源获取行情数据。vn.py默认的行情数据接口是米筐的rqdata，rqdata可以自由注册，有一个月的免费试用期。

如果感觉rqdata一个月的免费试用期太短，还可以选择使用聚宽的JQData，但需要自己编写一些程序。JQData是聚宽数据团队专门为金融机构、学术团体和量化研究者们提供的本地量化金融数据服务。使用JQData，可快速查看和计算金融数据，无障碍解决本地、Web、金融终端调用数据的需求。JQData的免费试用期是1年，可以满足学习需求。参本文档3.2节。

### 导入数据

按“导入数据”按钮，可从CSV文件导入行情数据并存入数据库，界面如图17-4所示。



图17-4 CSV载入

在图17-4界面上，按“选择文件”按钮选择包含行情数据的CSV文件。CSV文件中仅包含行情数据，不包含合约信息，合约信息要手工输入，包括合约代码、交易所和周期等。无论CSV文件包含多少字段，vn.py只关心图17-4中列出的字段，也即数据库表dbbardata所包含的字段。vn.py对CSV文件格式有较强的适应性，可以指定各字段对应的CSV文件表头信息，需要注意的是，图17-4中指定的表头信息大小写是敏感的。

在数据管理主界面上，“导入数据”按钮对应的槽函数为import\_data()，其代码为：

def import\_data(self) -> None:

"""“导入数据”按钮对应的槽函数"""

# 打开“从CSV文件导入数据”对话框

dialog = ImportDialog()

n = dialog.exec\_()

if n != dialog.Accepted:

return

# 取对话框上用户输入的信息

file\_path = dialog.file\_edit.text()

symbol = dialog.symbol\_edit.text()

exchange = dialog.exchange\_combo.currentData()

interval = dialog.interval\_combo.currentData()

datetime\_head = dialog.datetime\_edit.text()

open\_head = dialog.open\_edit.text()

low\_head = dialog.low\_edit.text()

high\_head = dialog.high\_edit.text()

close\_head = dialog.close\_edit.text()

volume\_head = dialog.volume\_edit.text()

open\_interest\_head = dialog.open\_interest\_edit.text()

datetime\_format = dialog.format\_edit.text()

# 调用管理引擎功能，从CSV文件中读取行情数据并写入数据库

start, end, count = self.engine.import\_data\_from\_csv(

file\_path,

symbol,

exchange,

interval,

datetime\_head,

open\_head,

high\_head,

low\_head,

close\_head,

volume\_head,

open\_interest\_head,

datetime\_format

)

# 反馈导入成功信息

msg = f"\

CSV载入成功\n\

代码：{symbol}\n\

交易所：{exchange.value}\n\

周期：{interval.value}\n\

起始：{start}\n\

结束：{end}\n\

总数量：{count}\n\

"

QtWidgets.QMessageBox.information(self, "载入成功！", msg)

上述代码中调用数据库管理引擎的import\_data\_from\_csv方法加载数据。数据库管理引擎在D:\vnpy220\vnpy\app\data\_manager的engine.py中定义，其import\_data\_from\_csv方法的代码为：

def import\_data\_from\_csv(

self,

file\_path: str,

symbol: str,

exchange: Exchange,

interval: Interval,

datetime\_head: str,

open\_head: str,

high\_head: str,

low\_head: str,

close\_head: str,

volume\_head: str,

open\_interest\_head: str,

datetime\_format: str

) -> Tuple:

"""从CSV文件中读取行情数据并写入数据库"""

# 打开CSV文件

with open(file\_path, "rt") as f:

buf = [line.replace("\0", "") for line in f]

# 创建一个csv模块的reader对象

reader = csv.DictReader(buf, delimiter=",")

bars = [] # 保存K线数据的列表

start = None # 起始时间，初始时为None

count = 0 # 数据量

# 读取CSV文件的每一行

for item in reader:

# 取日期时间字符串

if datetime\_format:

# 如果界面上有日期时间字符串格式的定义

dt = datetime.strptime(item[datetime\_head], datetime\_format)

else:

# 如果界面上没有日期时间字符串格式的定义，按标准格式进行解析

dt = datetime.fromisoformat(item[datetime\_head])

# 持仓量

open\_interest = item.get(open\_interest\_head, 0)

# 用读取的数据创建一个K线数据对象

bar = BarData(

symbol=symbol,

exchange=exchange,

datetime=dt,

interval=interval,

volume=float(item[volume\_head]),

open\_price=float(item[open\_head]),

high\_price=float(item[high\_head]),

low\_price=float(item[low\_head]),

close\_price=float(item[close\_head]),

open\_interest=float(open\_interest),

gateway\_name="DB",

)

# 加入K线数据列表

bars.append(bar)

# 数据量+1

count += 1

# 如果是第一条数据，记录起始时间

if not start:

start = bar.datetime

# 将K线数据列表中的数据写入数据库

database\_manager.save\_bar\_data(bars)

# 记录结束时间

end = bar.datetime

# 返回读取数据的统计信息

return start, end, count

其中CSV文件的读取操作已在教材9.2节介绍，本节不再赘述。

SQLite数据库类SqliteDatabase的框架代码见本文档3.3.2节，其save\_bar\_data方法的详细代码为：

def save\_bar\_data(self, bars: List[BarData]) -> bool:

"""将一批K线数据保存到数据库"""

# 从第0条数据中取关键字信息

bar = bars[0]

symbol = bar.symbol # 合约代码

exchange = bar.exchange # 交易所

interval = bar.interval # K线周期

# 将bar对象转换为带有时区信息的字典对象

data = [] # 字典对象列表

for bar in bars:

# 时区转换

bar.datetime = convert\_tz(bar.datetime)

d = bar.\_\_dict\_\_

d["exchange"] = d["exchange"].value

d["interval"] = d["interval"].value

d.pop("gateway\_name")

d.pop("vt\_symbol")

data.append(d)

# 根据情况，将数据插入到数据库，或对数据库中的原有数据进行更新

with self.db.atomic():

for c in chunked(data, 50):

DbBarData.insert\_many(c).on\_conflict\_replace().execute()

# 更新K线概览信息

# 从数据库中取原有概览信息。如果数据库中没有该合约该周期的概览信息，则返回None

overview: DbBarOverview = DbBarOverview.get\_or\_none(

DbBarOverview.symbol == symbol,

DbBarOverview.exchange == exchange.value,

DbBarOverview.interval == interval.value,

)

if not overview:

# 如果数据库中没有该合约该周期的概览信息

# 创建一个新的概览信息对象

overview = DbBarOverview()

overview.symbol = symbol

overview.exchange = exchange.value

overview.interval = interval.value

overview.start = bars[0].datetime

overview.end = bars[-1].datetime

overview.count = len(bars)

else:

# 如果数据库中有该合约该周期的概览信息

# 对概览信息进行更新

overview.start = min(bars[0].datetime, overview.start)

overview.end = max(bars[-1].datetime, overview.end)

s: ModelSelect = DbBarData.select().where(

(DbBarData.symbol == symbol)

& (DbBarData.exchange == exchange.value)

& (DbBarData.interval == interval.value)

)

overview.count = s.count()

# 保存概览信息

overview.save()

除了保存K线行情数据外，还需要对概览信息进行更新。其中具体的数据库增、改和查操作可参考教材9.5节内容，也可上网查询相关资料。

## 使用数据

在数据管理中取行情数据并显示实现起来比较简单，请读者自行分析。本节讨论在其他上层应用中如何使用数据库中的行情数据。历史行情数据的一个作用就是支持CTA回测，本节通过跟踪CTA回测功能的执行，研究数据库中数据的读取。

在菜单中选择“功能 - CTA回测”，或者单击左侧功能导航栏的图表按钮，启动“CTA回测”上层应用，其操作说明见教材15.4节。确保数据库中已经加载了所需的行情数据，在CTA回测界面上单击“开始回测”按钮，执行回测操作。

“开始回测”按钮的槽函数为D:\vnpy220\vnpy\app\cta\_backtester\ui目录下的widget.py中的start\_backtesting方法。在该方法的开始处设断点并跟踪，发现方法执行结束后没什么反应。仔细分析，是“result = self.backtester\_engine.start\_backtesting()”一句启动了其他线程进行实际的回测操作。backtester\_engine.start\_backtesting方法中如何执行回测，将在第教材18章（本文档第4章）讨论，现在暂时忽略。

在D:\vnpy220\vnpy\app\cta\_strategy目录下的backtesting.py文件中的load\_data()方法中设断点，发现数据在该方法的while循环中加载。BacktestingEngine类的load\_data()方法的代码分为三部分，第一部分是整体准备工作，代码为：

def load\_data(self):

"""

加载历史数据

回测使用行情数据的方法是一次性地将所需数据加载到回测执行引擎中来

本方法在参数优化时在本类中调用，一般情况是在回测引擎类中调用

"""

self.output("开始加载历史数据") # ①

# 如果未指定回测结束日期，则回测到现在

if not self.end: # ②

self.end = datetime.now()

# 检查日期的合法性

if self.start >= self.end: # ③

self.output("起始日期必须小于结束日期")

return

# 清空原有的历史数据

self.history\_data.clear() # ④

①：调用本类的output方法输出信息。

②：如果没有结束时间，默认为“现在”。

③：判断起止时间的合法性，起始时间不能大于等于终止时间。

④：清空前面加载的历史数据。history\_data是BacktestingEngine类的属性，其类型是一个列表，用于存放取得的行情数据，元素类型是BarData或者TickData。

第二部分为循环处理做准备，代码为：

# 注：新版本的vn.py回测增加了进度显示的功能，

# 每加载10%天数的数据，在界面上显示一条信息。

total\_days = (self.end - self.start).days # ①

progress\_days = max(int(total\_days / 10), 1) # ②

progress\_delta = timedelta(days=progress\_days)

interval\_delta = INTERVAL\_DELTA\_MAP[self.interval] # ③

start = self.start # ④

end = self.start + progress\_delta # ⑤

progress = 0 # ⑥

①：计算需要加载行情数据的天数。

②：进度增量。progress\_days是以天为单位的增量，可以看出vn.py每次从数据库中取10%天数的数据。progress\_delta为由天数换算得到的时间跨度。progress\_days用于在界面上显示进度，progress\_delta作为查询条件到数据库中取数据。

③：将K线周期影射为时间跨度，如果选择的回测K线周期是1分钟，得到的时间跨度就是“1分钟”。

每次从数据库中取10%的数据(称为进度增量)，这个时间跨度由起止时间start和end限定。当下次取数时，不能从end开始取，这样时间为end的数据就会被取两次，而是要从end + interval\_delta开始取。

④：每次取数的起始时间。

⑤：每次取数的结束时间。

⑥：进度值，取值在0~1之间。

第三部分循环从数据库中取数，并显示结果。

while start < self.end:

# 显示当前进度

progress\_bar = "#" \* int(progress \* 10 + 1)

self.output(f"加载进度：{progress\_bar} [{progress:.0%}]")

end = min(end, self.end) # ①

if self.mode == BacktestingMode.BAR: # ②

data = load\_bar\_data(

self.symbol,

self.exchange,

self.interval,

start,

end

)

else: # ③

data = load\_tick\_data(

self.symbol,

self.exchange,

start,

end

)

self.history\_data.extend(data) # ④

progress += progress\_days / total\_days # ⑤

progress = min(progress, 1)

start = end + interval\_delta # ⑥

end += progress\_delta

self.output(f"历史数据加载完成，数据量：{len(self.history\_data)}") # ⑦

①：确保取数的终止时间在指定范围内。

②：如果是基于K线数据的回测，则从数据库中取K线数据。

③：如果是基于Tick数据的回测，则从数据库中取分时数据。

④：将新取到的行情数据追加到history\_data列表中。

⑤：重新计算进度值。

⑥：重新计算下次加载数据的起止时间。

⑦：循环结束之后，显示加载结果。

在上述程序中，调用本类的load\_bar\_data方法加载K线数据。

def load\_bar\_data(

symbol: str,

exchange: Exchange,

interval: Interval,

start: datetime,

end: datetime

):

""""""

return database\_manager.load\_bar\_data(

symbol, exchange, interval, start, end

)

如果是SQLite数据库，最终调用的是前述SqliteDatabase类(见本文档3.3.2节)的load\_bar\_data方法，其代码为：

def load\_bar\_data(

self,

symbol: str,

exchange: Exchange,

interval: Interval,

start: datetime,

end: datetime

) -> List[BarData]:

"""取某个合约某周期某段时间的K线数据"""

# 从数据库中取数据

s: ModelSelect = (

DbBarData.select().where(

(DbBarData.symbol == symbol)

& (DbBarData.exchange == exchange.value)

& (DbBarData.interval == interval.value)

& (DbBarData.datetime >= start)

& (DbBarData.datetime <= end)

).order\_by(DbBarData.datetime)

)

# 根据取到的数据创建列表

vt\_symbol = f"{symbol}.{exchange.value}"

bars: List[BarData] = []

for db\_bar in s:

db\_bar.datetime = DB\_TZ.localize(db\_bar.datetime)

db\_bar.exchange = Exchange(db\_bar.exchange)

db\_bar.interval = Interval(db\_bar.interval)

db\_bar.gateway\_name = "DB"

db\_bar.vt\_symbol = vt\_symbol

bars.append(db\_bar)

# 返回行情数据列表

return bars

从数据库中取数据是典型的peewee代码，其中的select和where等方法在教材9.5节已经介绍，不再赘述。

# CTA回测

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，本章选择CTA回测。

对应《Python量化交易从入门到实战》的第18章。

本章学习方法：以教材为主线进行学习，同时对照参考资料。

需要对教材第18章做的补充很少。也可以说需要补充的内容很多，所以在本文档中专门用一个大部分——第三部分 CTA回测深入分析——对相关程序进行分析。

我们的原则是尽量不动教材的主线。以下是针对教材第18章，针对版本2.2.0所补充的内容。

### 18.2.2 执行流程(勘误)

**1.BacktesterManager窗体类**

原文：BacktesterManager是回测窗体类，在D:\vnpy214\vnpy\app\cta\_strategy\ui目录下的widget.py中定义。

更正：BacktesterManager是回测窗体类，在D:\vnpy214\vnpy\app\cta\_backtester\ui目录下的widget.py中定义。

### 18.2.3 存在问题

关于教材中提到的线程控制方式不“优雅”的问题，经我们提出后，vn.py的维护团队已经做出了修改。但总感觉改得还是不彻底，比如教材中提到的“搜索程序代码没有发现finally保留字”的问题，在vn.py的新增功能中已经可以看到finally保留字，但原有功能中都未做此优化。这是可以理解的，估计等下一次代码全面重构时会考虑此工作。

### 18.3.2 回测操作

回测线程函数增加了注释的相关代码如下（如果现在看不明白也没关系，后面还会详细分析）：

def run\_backtesting(

self,

class\_name: str, # 策略类名称

vt\_symbol: str, # 合约本地代码

interval: str, # K线周期

start: datetime, # 开始日期

end: datetime, # 结束日期

rate: float, # 手续费率

slippage: float, # 交易滑点

size: int, # 合约乘数

pricetick: float, # 价格跳动

capital: int, # 回测资金

inverse: bool, # 合约模式

setting: dict # 策略参数

):

"""回测线程函数"""

# 初始化回测结果和统计信息

self.result\_df = None

self.result\_statistics = None

# 指定回测执行引擎

engine = self.backtesting\_engine

engine.clear\_data()

# 根据“K线周期”，确定是使用K线数据还是Tick数据

if interval == Interval.TICK.value:

mode = BacktestingMode.TICK

else:

mode = BacktestingMode.BAR

# 设置回测参数

engine.set\_parameters(

vt\_symbol=vt\_symbol,

interval=interval,

start=start,

end=end,

rate=rate,

slippage=slippage,

size=size,

pricetick=pricetick,

capital=capital,

inverse=inverse,

mode=mode

)

# 根据策略名称取策略类（策略名称来自于回测界面左上的“交易策略”列表

# 如回测DoubleMaStrategy策略时，策略类为：

# <class 'vnpy.app.cta\_strategy.strategies.double\_ma\_strategy.DoubleMaStrategy'>

strategy\_class = self.classes[class\_name]

# 将策略类和策略参数加载到回测引擎

engine.add\_strategy(

strategy\_class,

setting

)

# 加载历史行情数据

engine.load\_data()

try:

# 调用回测执行引擎的run\_backtesting方法执行回测任务

# 包括策略初始化及回放历史行情数据等

engine.run\_backtesting()

except Exception:

# 如果在回测过程中发生异常

msg = f"策略回测失败，触发异常：\n{traceback.format\_exc()}"

self.write\_log(msg)

# 将thread置为None，允许新的回测线程执行，参18.2节

self.thread = None

return

# 计算回测的结果

self.result\_df = engine.calculate\_result()

# 对回测结果进行统计计算

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

# 将thread置为None，允许新的回测线程执行，参18.2节

self.thread = None

# 创建一个回测结束事件(该事件只有类型没有数据)，并追加到事件引擎的事件队列中

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

self.event\_engine.put(event)

### 18.3.3 回测结束事件处理

以下是教材原文：

现在详细说明回测线程中第⑨部分的含义。

BacktesterManager是回测窗体类，在D:\vnpy220\vnpy\app\cta\_backtester\ui目录下的widget.py中定义。在窗体类中首先定义一个PyQt5信号：

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

在回测窗体类的register\_event方法中：

self.signal\_backtesting\_finished.connect(

self.process\_backtesting\_finished\_event)

指明process\_backtesting\_finished\_event为该信号的槽函数。后续代码：

self.event\_engine.register(

EVENT\_BACKTESTER\_BACKTESTING\_FINISHED,

self.signal\_backtesting\_finished.emit)

将发出signal\_backtesting\_finished信号的函数注册为EVENT\_BACKTESTER\_BACKTESTING\_FINISHED事件的处理函数。

以下为补充说明：

对于上述内容，可能有读者会疑问：为什么不直接将process\_backtesting\_finished\_event注册为事件处理函数呢？还要先将其先关联为信号槽函数，然后再发出信号呢？

Qt编程中的一个重要知识点：不能在Qt事件循环以外的线程中，直接调用对Qt图形组件进行修改操作，否则可能因为冲突导致程序崩溃。如果要这么做，只能通过Signal/Slot机制来实现这种跨线程的通知调用。

增加了注释的信号槽函数相关代码如下：

def process\_backtesting\_finished\_event(self, event: Event):

""""""

# 取统计结果信息，并显示到界面中间顶部的表格中

statistics = self.backtester\_engine.get\_result\_statistics()

self.statistics\_monitor.set\_data(statistics)

# 取回测结果，并显示到界面右侧图表的四个子图中

df = self.backtester\_engine.get\_result\_df()

self.chart.set\_data(df)

# 将“成交记录”到“K线图表”等四个按钮改为可用

self.trade\_button.setEnabled(True)

self.order\_button.setEnabled(True)

self.daily\_button.setEnabled(True)

# Tick数据不能使用K线图显示

interval = self.interval\_combo.currentText()

if interval != Interval.TICK.value:

self.candle\_button.setEnabled(True)

可以看到，该函数包含界面操作，不能在Qt事件循环以外的线程中直接调用。

第二部分 VN Trader深入分析

从本部分开始是《Python量化交易从入门到实战》一书所没有的内容，是对教材的整体补充。

本文档第2章已经对VN Trader的代码进行了分析，那时主要针对VN Trader的界面。本部分继续深入，侧重于分析功能，特别是交易功能的实现方法。

无论是学习编程的交易员，还是想做交易的程序员，学习vn.py的目的都是深度使用，真正用vn.py做量化交易。要做量化交易就要写自己的策略，如果现有平台不能满足自己策略的要求怎么办？比如想要更高的效率，更强的策略功能等，就要对平台进行定制，这也是教材作者不推荐网站型量化交易平台的原因。要合理地定制甚至优化，对原平台的体系结构一定要透彻了解。研究程序结构的重点是策略，研究策略之前必须先扫清外围。

要深入研究策略，需要先掌握行情接口和交易平台，本部分就完成这项工作,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。

本部分目标：搞清VN Trader交易平台。

# 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，交易系统主要负责委托单处理、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，以及时提示并控制风险。

## 在C++中使用CTP

上海期货信息技术有限公司（简称上期技术）提供CTP的C++开发接口，可到其主页<http://www.sfit.com.cn/>下载相关文件及文档。

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信，而交易托管系统负责投资者的交易业务处理。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——ThostFtdcTraderApi.h和ThostFtdcMdApi.h，一个处理交易，一个处理行情。

（1）处理交易的ThostFtdcTraderApi.h有两个类，分别是CThostFtdcTraderApi和CThostFtdcTraderSpi，以Api结尾的用来下命令，以Spi结尾的用来响应命令的回调。通过CThostFtdcTraderApi向CTP发送操作请求，通过CThostFtdcTraderSpi接收CTP的操作响应。

（2）处理行情的ThostFtdcMdApi.h也有两个类，分别是CThostFtdcMdApi和CThostFtdcMdSpi。

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。也可以通过CThostFtdcMdApi完成行情接口的初始化，登入，订阅，接收行情等业务；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（尽管不好）。

为了方便大家学习，我用Visual Studio 2019 + Qt 5.13.1做了一个示例，示例源代码可加入QQ群881645236下载。之所以选择Qt而没有使用纯的C++，是基于以下原因：

* 与纯C++中使用CTP极其相似，不会混淆
* 在图形界面上可以使用Qt的信号/槽机制
* vn.py使用PyQt，可以互相参照，便于vn.py的学习

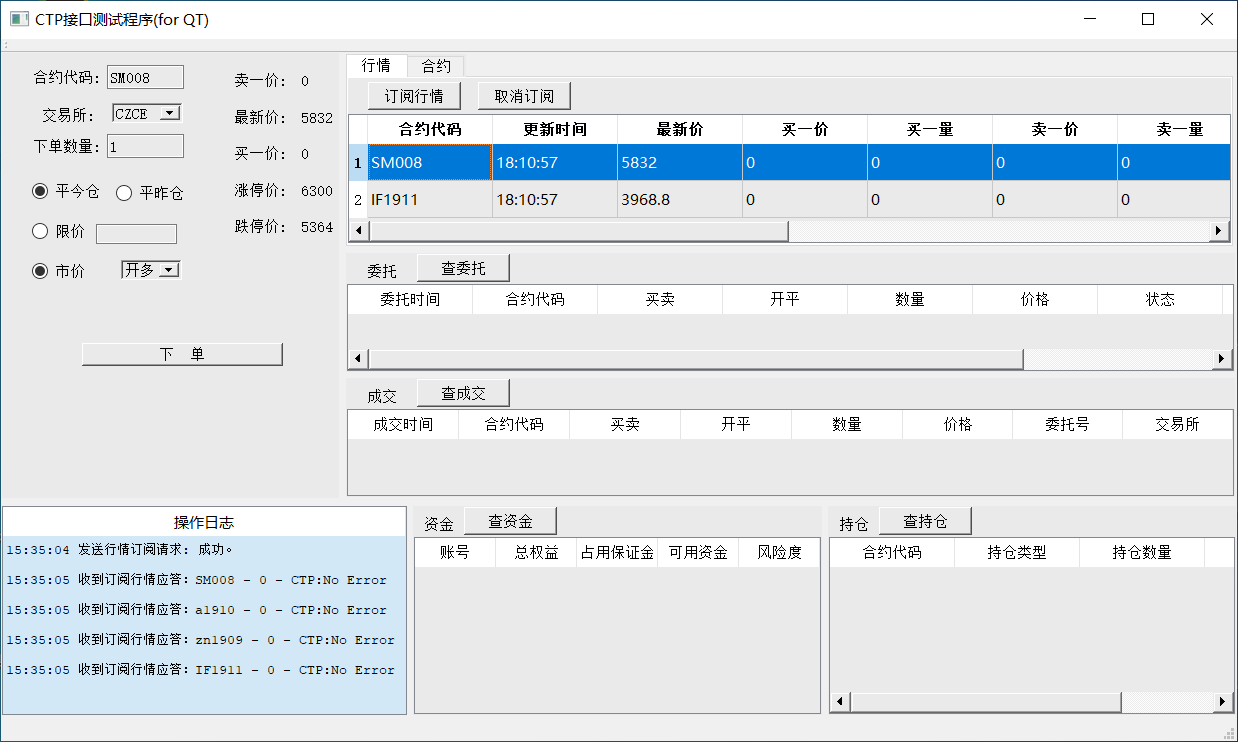
程序的登录界面如下。



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。

如果确认在登录界面上所填信息正确，但按“登录”按钮仍然没有反应，应该是时段不对，上期技术经常会停服务器维护。可用vn.py验证一下，如果也不能登录，可以等一段时间再试。

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。程序很初级，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。代码放到这里，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;

};

MdSpi主要有两方面的功能：

1-实现CThostFtdcMdSpi的各种响应回调函数，对服务器返回的行情信息进行处理。

2-通过成员变量CThostFtdcMdApi\* pMdUserApi，向服务器发出行情请求。

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;

};

TdSpi类的实现方法与MdSpi相似。

### 线程的使用

CTP我也是初学者，但根据以往经验并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。

其他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，可能后面会发现自己是错的。如果有阅读此文档的老师确定我是错的，请告诉我。

### 用户登录

用户登录时输入账号和密码即可，编程处理则需要知道API内部的工作流程。交易接口的登录流程如下：

* 调用Init，开始连接
* 收到OnFrontConnected，确认连接成功
* 调用ReqAuthenticate，这一步填入AppID和AuthCode，进行认证
* 收到OnRspAuthenticate，确认认证成功
* 调用ReqUserLogin，这一步同样需要填入AppID，进行登录
* 收到OnRspUserLogin，确认登录成功

如果是登录行情接口，则不需要中间的认证一步，连接后直接登录就行。每一步出错的话都会有相应的报错输出提示。本文档9.4节分析vn.py中交易接口的登录过程，流程与本小节相同。

## CTP的Python封装

vn.py在2015年诞生之初，只是单纯对C++交易API接口进行Python封装，也就是说，本节是vn.py的最原始最基础的部分。

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。

具体流程：

1. 需要将两个定义了数据类型的头文件转换成python的格式，方便后续gateway映射使用
2. 将原生ctpapi接口用C++继承实现一次
3. 利用pybind11对C++继承好的类进行转换

封装结果：

形成D:\vnpy220\vnpy\api\ctp下的三个文件。

ctp\_constant.py：定义与CTP通信有关的所有枚举量，与CTP接口文件ThostFtdcUserApiDataType.h相对应。

vnctpmd.pyd包含类MdApi的定义。继承自CTP接口中CThostFtdcMdSpi类和CThostFtdcMdApi类，作用与前述CTP示例的MdSpi相似。

vnctptd.pyd：包含类TdApi的定义。继承自CTP接口中CThostFtdcTraderSpi类和CThostFtdcTraderApi类，作用与前述CTP示例的TdSpi相似。

在D:\vnpy220\vnpy\api\ctp下的的\_\_init\_\_.py文件中：

from .vnctpmd import MdApi

from .vnctptd import TdApi

from .ctp\_constant import \*

将上述封装的成果集成到模块中。

# 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:\vnpy220\vnpy\trader目录下：

* event.py定义VN Trader中用到的事件类型字符串，包括EVENT\_TICK、EVENT\_ORDER、EVENT\_TRADE、EVENT\_POSITION、EVENT\_ACCOUNT、EVENT\_CONTRACT和EVENT\_LOG等。
* constant.py定义VN Trader中用到的字符串常量，包括Direction、Offset、Status、Product、OrderType、OptionType、Exchange、Currency和Interval等。
* object.py定义VN Trader中用于交易功能的基础数据结构，包括TickData、BarData、OrderData、TradeData、PositionData、AccountData、ContractData、LogData、OrderRequest、CancelRequest、SubscribeRequest和HistoryRequest等。

更详细的说明在本文第四部分。

## 底层接口基类BaseGateway

在D:\vnpy220\vnpy\trader目录下的gateway.py文件中定义了所有底层接口的基类BaseGateway，由有两大类函数构成。

**回调函数**

名称形如on\_xxx的方法，如on\_event、on\_tick等，为回调函数。包括事件、tick数据、成交数据、委托单数据、持仓数据、账户数据、日志数据、合约数据等的回调函数，用于向上层推送已经按照vn.py内部标准封装好的数据对象。

**功能函数**

功能函数都是抽象函数，必须继承实现。包括：

* connect(self,setting:dict):连接接口。如果已经连接，则输出日志，成功连接后进行必要的查询操作。
* close(self):关闭接口连接。
* subscribe(self,req:SubscribeRequest):订阅时必须传入vn.py标准订阅请求类参数。
* send\_order(self,req:OrderRequest):发单必须传入vn.py标准报单请求类参数，对于报单，有额外的要求：
  + 利用OrderRequest生成OrderData，即从报单请求生成报单数据。
  + 给OrderData赋值一个与接口类有关的unique\_id。
  + 如果order已经发出，则更改报单状态为“已提交”，如果发送失败，则应该为“拒单”。
  + 利用on\_order回调，send\_order应返回OrderData.vt\_orderid。
* cancel\_order(self,reqs:Sequence[CancelRequest]):撤销一系列委托单。
* query\_account(self):查询账户信息。
* query\_position(self):查询持仓信息。
* query\_history(self):查询历史信息。
* get\_default\_setting(self):返回默认配置信息。

BaseGateway类的定义比较重要，使用代码+注释的形式进行说明，读者应该通读。gateway.py的代码如下：

from abc import ABC, abstractmethod

from typing import Any, Sequence, Dict, List, Optional, Callable

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,

Exchange,

BarData

)

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: Dict[str, Any] = {}

# 接口支持的交易所

exchanges: List[Exchange] = []

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: EventEngine = event\_engine

self.gateway\_name: str = gateway\_name

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

"""

深度行情推送

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)

def on\_trade(self, trade: TradeData) -> None:

"""

成交信息推送

1-将一个成交事件推送给事件引擎

2-还要将一个带有本地代码的成交事件推送给事件引擎

"""

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

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

def on\_order(self, order: OrderData) -> None:

"""

委托单变化推送

发送委托请求后执行

委托请求不在这儿发，在接口的send\_order()中发，此处执行委托请求发送完成后需要做的工作，包括：

1-将一个委托单事件推送给事件引擎

2-还要将一个带有内部委托号的委托单事件推送给事件引擎

"""

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

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

def on\_position(self, position: PositionData) -> None:

"""

持仓信息推送

1-将一个仓位事件推送给事件引擎

2-还要将一个带有本地代码的仓位事件推送给事件引擎

"""

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

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

def on\_account(self, account: AccountData) -> None:

"""

账户信息推送

1-将一个账户信息事件推送给事件引擎

2-还要将一个带有本地账户代码的账户信息事件推送给事件引擎

"""

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

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

def on\_quote(self, quote: QuoteData) -> None:

"""

询价单推送

1-将一个询价单事件推送给事件引擎

2-还要将一个带有本地代码的询价单事件推送给事件引擎

高注：应该是正在开发中的功能，目前还没有哪个功能使用此数据。

"""

self.on\_event(EVENT\_QUOTE, quote)

self.on\_event(EVENT\_QUOTE + quote.vt\_symbol, quote)

def on\_log(self, log: LogData) -> None:

"""

日志推送

将一个日志事件推送给事件引擎

"""

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

def on\_contract(self, contract: ContractData) -> None:

"""

合约基础信息推送

"""

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

def write\_log(self, msg: str) -> None:

"""

从接口发起，写一个日志事件

"""

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

self.on\_log(log)

@abstractmethod

def connect(self, setting: dict) -> None:

"""

连接

要实现这个方法，您必须：

\* 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) -> None:

"""

关闭连接

"""

pass

@abstractmethod

def subscribe(self, req: SubscribeRequest) -> None:

"""

订阅行情

传入vn.py标准订阅请求类参数。

"""

pass

@abstractmethod

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

"""

发送新的委托单到服务器

传入一个vn.py标准报单请求类对象作为参数。

实现时需要完成以下工作：

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

\* 也就是根据定单请求数据创建定单数据

\* 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

\* 由on\_order响应:

\* 返回OrderData.vt\_orderid

:返回为OrderData创建的vt\_orderid

"""

pass

@abstractmethod

def cancel\_order(self, req: CancelRequest) -> None:

"""

撤消一个已经存在的委托单

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]) -> List[str]:

"""

发送一批委托单到服务器

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]) -> None:

"""

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)

def send\_quote(self, req: QuoteRequest) -> str:

"""

发送一个新的双向询价单到服务器

高注：应该是正在开发中的功能，目前还没有哪个具体的接口实现此功能

Send a new two-sided quote to server.

implementation should finish the tasks blow:

\* create an QuoteData from req using QuoteRequest.create\_quote\_data

\* assign a unique(gateway instance scope) id to QuoteData.quoteid

\* send request to server

\* if request is sent, QuoteData.status should be set to Status.SUBMITTING

\* if request is failed to sent, QuoteData.status should be set to Status.REJECTED

\* response on\_quote:

\* return vt\_quoteid

:return str vt\_quoteid for created QuoteData

"""

return ""

def cancel\_quote(self, req: CancelRequest) -> None:

"""

撤消一个存在的询价单

高注：应该是正在开发中的功能，目前还没有哪个具体的接口实现此功能

Cancel an existing quote.

implementation should finish the tasks blow:

\* send request to server

"""

pass

@abstractmethod

def query\_account(self) -> None:

"""

查询账户资金

"""

pass

@abstractmethod

def query\_position(self) -> None:

"""

查询持仓

"""

pass

def query\_history(self, req: HistoryRequest) -> List[BarData]:

"""

查询历史K线数据

"""

pass

def get\_default\_setting(self) -> Dict[str, Any]:

"""

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，用于其他几个接口，CTP中没有用到，不再分析。

## CTP接口类CtpGateway

vn.py中实现CTP通信的类是CtpGateway，继承自BaseGateway。

CtpGateway类在D:\vnpy220\vnpy\gateway\ctp目录下的ctp\_gateway.py文件中定义。

在开头的导入部分，相关代码如下：

from vnpy.api.ctp import (

MdApi,

TdApi,

THOST\_FTDC\_OAS\_Submitted,

THOST\_FTDC\_OAS\_Accepted,

THOST\_FTDC\_OAS\_Rejected,

……

)

从模块vnpy.api.ctp导入CTP接口封装的所有成果。

下面生成几个转换字典。vn.py定义了一些自己的常量，在D:\vnpy220\vnpy\trader目录下的constant.py文件中定义，详见本文档15.1节。每个具体的接口有自己独特的常量定义，CTP接口的常量在D:\vnpy219\vnpy\api\ctp下的ctp\_constant.py中定义，这是CTP接口封装的成果。我们需要生成几个转换字典，对vn.py常量和CTP接口常量进行转换。在后续的处理中使用vn.py常量，以达到对所有接口使用相同程序的效果。相关代码如下：

# 在D:\vnpy220\vnpy\trader目录下的constant.py中定义了VN Trader中用到的字符串常量

# 各类接口有自己的规定，需要进行转换

# 委托单状态转换

STATUS\_CTP2VT = {

THOST\_FTDC\_OAS\_Submitted: Status.SUBMITTING,

THOST\_FTDC\_OAS\_Accepted: Status.SUBMITTING,

THOST\_FTDC\_OAS\_Rejected: Status.REJECTED,

THOST\_FTDC\_OST\_NoTradeQueueing: Status.NOTTRADED,

THOST\_FTDC\_OST\_PartTradedQueueing: Status.PARTTRADED,

THOST\_FTDC\_OST\_AllTraded: Status.ALLTRADED,

THOST\_FTDC\_OST\_Canceled: Status.CANCELLED

}

# 方向转换

DIRECTION\_VT2CTP = {

Direction.LONG: THOST\_FTDC\_D\_Buy,

Direction.SHORT: THOST\_FTDC\_D\_Sell

}

DIRECTION\_CTP2VT = {v: k for k, v in DIRECTION\_VT2CTP.items()}

DIRECTION\_CTP2VT[THOST\_FTDC\_PD\_Long] = Direction.LONG

DIRECTION\_CTP2VT[THOST\_FTDC\_PD\_Short] = Direction.SHORT

# 委托单类型转换

ORDERTYPE\_VT2CTP = {

OrderType.LIMIT: THOST\_FTDC\_OPT\_LimitPrice,

OrderType.MARKET: THOST\_FTDC\_OPT\_AnyPrice

}

ORDERTYPE\_CTP2VT = {v: k for k, v in ORDERTYPE\_VT2CTP.items()}

# 开平方向转换

OFFSET\_VT2CTP = {

Offset.OPEN: THOST\_FTDC\_OF\_Open,

Offset.CLOSE: THOST\_FTDC\_OFEN\_Close,

Offset.CLOSETODAY: THOST\_FTDC\_OFEN\_CloseToday,

Offset.CLOSEYESTERDAY: THOST\_FTDC\_OFEN\_CloseYesterday,

}

OFFSET\_CTP2VT = {v: k for k, v in OFFSET\_VT2CTP.items()}

# 交易所转换

EXCHANGE\_CTP2VT = {

"CFFEX": Exchange.CFFEX,

"SHFE": Exchange.SHFE,

"CZCE": Exchange.CZCE,

"DCE": Exchange.DCE,

"INE": Exchange.INE

}

# 在D:\vnpy220\vnpy\trader目录下的constant.py中定义了VNTrader支持所有交易所，包括中国和国际的

# CTP只支持上述交易所

# 产品类型转换

PRODUCT\_CTP2VT = {

THOST\_FTDC\_PC\_Futures: Product.FUTURES,

THOST\_FTDC\_PC\_Options: Product.OPTION,

THOST\_FTDC\_PC\_SpotOption: Product.OPTION,

THOST\_FTDC\_PC\_Combination: Product.SPREAD

}

# 期权类型转换

OPTIONTYPE\_CTP2VT = {

THOST\_FTDC\_CP\_CallOptions: OptionType.CALL,

THOST\_FTDC\_CP\_PutOptions: OptionType.PUT

}

下面介绍CtpGateway类的初始化部分，代码如下：

class CtpGateway(BaseGateway):

"""

VN Trader的CTP接口

"""

# 默认配置字典，key是配置字段名，value是默认值

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": ""

}

# 交易所列表

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。在创建成员变量时，将self作为参数传入。在这两个类中，通过属性gateway来访问本类实例，比如调用本类中的成员函数。这两个类也在本文件中定义，是Python CTP的接口。两个类定义的框架代码为：

class CtpMdApi(MdApi):

""""""

def \_\_init\_\_(self, gateway):

"""构造函数"""

……

# 接口的连接状态。True-已连接；False-未连接

self.connect\_status = False

# 接口的登录状态。True-已登录；False-未登录

self.login\_status = False

# 已订阅合约集合

self.subscribed = set()

……

def onFrontConnected(self):

"""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。"""

def onFrontDisconnected(self, reason: int):

"""行情服务器连接断开"""

def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool):

"""登录请求响应"""

def onRspError(self, error: dict, reqid: int, last: bool):

"""行情接口报错"""

def onRspSubMarketData(self, data: dict, error: dict, reqid: int, last: bool):

"""订阅行情应答"""

class CtpTdApi(TdApi):

""""""

def \_\_init\_\_(self, gateway):

"""构造函数"""

def onFrontConnected(self):

"""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。"""

def onFrontDisconnected(self, reason: int):

"""交易服务器连接断开"""

def onRspAuthenticate(self, data: dict, error: dict, reqid: int, last: bool):

"""交易服务器授权验证"""

def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool):

"""登录请求响应"""

def onRspOrderInsert(self, data: dict, error: dict, reqid: int, last: bool):

"""报单录入请求响应"""

def onRspOrderAction(self, data: dict, error: dict, reqid: int, last: bool):

"""报单操作请求响应"""

可以看到，这两个类分别从MdApi和TdApi继承。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。



调用时从上层向下层调用。

# CTP接口的使用

与具体底层接口相关的程序都在D:\vnpy220\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接口

vn.py的编程水平很高，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，参教材的“16.1. 程序主函数”和“16.2.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，参教材的“16.3.主界面”和“16.5.菜单”两节，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) -> None:

"""为连接接口打开“连接”对话框"""

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

dialog.exec\_()

该函数适用于所有接口。对于CTP接口来说，其功能是打开如下图的对话框。



其中ConnectDialog是一个继承自QtWidgets.QDialog的窗口类，它根据接口名称取得连接参数，动态生成上述“接连”对话框。

## 通用连接对话框类ConnectDialog

ConnectDialog在D:\vnpy220\vnpy\trader\ui目录下的widget.py中定义。

class ConnectDialog(QtWidgets.QDialog):

"""

对话框：连接到特定的交易接口

"""

def \_\_init\_\_(self, main\_engine: MainEngine, gateway\_name: str):

"""初始化"""

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

self.main\_engine: MainEngine = main\_engine

# 接口名

self.gateway\_name: str = gateway\_name

# 生成接口连接配置信息文件名，如connect\_ctp.json

self.filename: str = f"connect\_{gateway\_name.lower()}.json"

# 字段控件字典，key为字段名，value为一个元组，其值为(输入控件，值类型)

self.widgets: Dict[str, QtWidgets.QWidget] = {}

# 窗口初始化

self.init\_ui()

def init\_ui(self) -> None:

"""窗口初始化"""

# 修改窗口标题

self.setWindowTitle(f"连接{self.gateway\_name}")

# 得到默认配置字典，包括各配置项的字段名和默认值（根据默认值可确定类型）

# 对于CTP接口，其值为{'用户名': '', '密码': '', ...}

# 该字典来自CtpGateway类的类属性default\_setting

default\_setting = self.main\_engine.get\_default\_setting(

self.gateway\_name)

# 取上次连接时保存下来的连接配置信息

# 对于CTP接口，这些配置信息保存在文件connect\_ctp.json中

loaded\_setting = load\_json(self.filename)

# 根据配置信息初始化行编辑控件布局

form = QtWidgets.QFormLayout()

# 针对每个配置项进行操作

for field\_name, field\_value in default\_setting.items():

# 默认值类型

field\_type = type(field\_value)

if field\_type == list: # 如果是列表

widget = QtWidgets.QComboBox()

widget.addItems(field\_value)

if field\_name in loaded\_setting:

saved\_value = loaded\_setting[field\_name]

ix = widget.findText(saved\_value)

widget.setCurrentIndex(ix)

else: # 如果是字符串类型

# 使用默认值创建QLineEdit控件

widget = QtWidgets.QLineEdit(str(field\_value))

# 如果该字段有保存值

if field\_name in loaded\_setting:

# 取得保存值

saved\_value = loaded\_setting[field\_name]

# 修改QLineEdit控件的显示值（覆盖默认值）

widget.setText(str(saved\_value))

# 如果是"密码"，设为显示不回显

if "密码" in field\_name:

widget.setEchoMode(QtWidgets.QLineEdit.Password)

# 布局中增加一行

form.addRow(f"{field\_name} <{field\_type.\_\_name\_\_}>", widget)

# 加入到字段控件字典

self.widgets[field\_name] = (widget, field\_type)

button = QtWidgets.QPushButton("连接")

# 关联“连接”按钮与槽函数

button.clicked.connect(self.connect)

form.addRow(button)

self.setLayout(form)

# “连接”按钮的槽函数

def connect(self) -> None:

"""从输入控件上取连接配置信息，并连接接口"""

# 连接配置信息字典

setting = {}

for field\_name, tp in self.widgets.items():

widget, field\_type = tp

if field\_type == list:

field\_value = str(widget.currentText())

else:

field\_value = field\_type(widget.text())

setting[field\_name] = field\_value

# 写接口连接配置信息文件

save\_json(self.filename, setting)

# 根据连接配置信息，调用主引擎的connect方法，连接接口

self.main\_engine.connect(setting, self.gateway\_name)

self.accept()

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。

按“连接”按钮时，调用主引擎的connect方法。主引擎的connect方法的代码为：

class MainEngine:

……

def connect(self, setting: dict, gateway\_name: str) -> None:

"""

连接一个特定接口

"""

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

if gateway:

gateway.connect(setting)

最终调用的是CtpGateway类的connect函数。

## CTP接口的connect函数

后续的执行就进入到CtpGateway类内部，CtpGateway类在D:\vnpy220\vnpy\gateway\ctp目录下的ctp\_gateway.py文件中定义，其connect函数代码为：

def connect(self, setting: dict):

"""连接CTP接口"""

# 取连接配置信息

userid = setting["用户名"]

password = setting["密码"]

brokerid = setting["经纪商代码"]

td\_address = setting["交易服务器"]

md\_address = setting["行情服务器"]

appid = setting["产品名称"]

auth\_code = setting["授权编码"]

product\_info = setting["产品信息"]

# 对服务器地址进行处理

if (

(not td\_address.startswith("tcp://"))

and (not td\_address.startswith("ssl://"))

):

td\_address = "tcp://" + td\_address

if (

(not md\_address.startswith("tcp://"))

and (not md\_address.startswith("ssl://"))

):

md\_address = "tcp://" + md\_address

# 连接交易服务器

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

# 连接行情服务器

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):

"""初始化账户查询函数"""

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CTP接口有关的处理函数如下。

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

"""注册与CTP接口有关的时钟事件处理函数，定时执行账户查询"""

# 计数，每2秒执行一次查询

self.count += 1

if self.count < 2:

return

self.count = 0

# 从查询函数列表中取第一个函数

# 注：列表中有两个函数，一个查询资金，一个查询持仓

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

# 执行取出的查询函数

func()

# 再将查询函数追加到尾部

self.query\_functions.append(func)

# 上述代码达到循环执行列表中函数的目的，每次只执行一个

# 更新行情接口的当前日期

# 该日期用于在接收深度行情通知时生成时间戳

self.md\_api.update\_date()

与CTP接口有关的处理函数也可以有多个，每2秒钟执行一次。但每次不是执行所有函数，而是循环地执行其中一个。分析本节和前一节代码，功能效果就是每2秒钟触发一次，这次查询账户，下次查询持仓，再下次还是查询账户，如此往复。

经过上述步骤，就连接到了CTP行情服务器，并可定时查询账户和持仓信息。

# 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、账户信息等。

本章以Tick数据为线索，对Tick数据的使用进行分析。其他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

下图是VN Trader主窗口的左上部分。



这部分窗口在D:\vnpy220\vnpy\trader\ui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：

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

""""初始化"""

……

# 当前监控合约本地代码和价格显示精度

self.vt\_symbol: str = ""

self.price\_digits: int = 0 # 显示价格时精确到小数点后多少位

# 界面初始化

self.init\_ui()

def init\_ui(self) -> None:

"""界面初始化"""

……

# 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的相关代码如下：

def set\_vt\_symbol(self) -> None:

"""

“代码”输入框中按回车键时，调用此函数

对vt\_symbol的深度Tick数据进行监控

"""

# 取合约代码

symbol = str(self.symbol\_line.text())

if not symbol:

return

# 取交易所，并与合约代码合成本地代码

exchange\_value = str(self.exchange\_combo.currentText())

vt\_symbol = f"{symbol}.{exchange\_value}"

# 如果本地代码未改变，什么都不做；如果已改变，保存到本实例属性

if vt\_symbol == self.vt\_symbol:

return

self.vt\_symbol = vt\_symbol

# Update name line widget and clear all labels

# 从主引擎中取对应的合约

contract = self.main\_engine.get\_contract(vt\_symbol)

if not contract:

# 如果未找到对应合约，清理显示

self.name\_line.setText("")

gateway\_name = self.gateway\_combo.currentText()

else:

# 如果找到对应合约

# 在界面上显示合约名称

self.name\_line.setText(contract.name)

# 在合约中取得接口名称

gateway\_name = contract.gateway\_name

# 更新界面上的“接口”列表

ix = self.gateway\_combo.findText(gateway\_name)

self.gateway\_combo.setCurrentIndex(ix)

# 计算价格显示精度

self.price\_digits = get\_digits(contract.pricetick)

# 清空所有价格标签的显示文本

self.clear\_label\_text()

# 清空“数量”输入框的内容

self.volume\_line.setText("")

# 清空“价格”输入框的内容

self.price\_line.setText("")

# 创建行情订阅请求

req = SubscribeRequest(

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

)

# 订阅行情

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) -> None:

"""

订阅特定接口的行情数据

"""

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)

调用行情接口的subscribe函数：

def subscribe(self, req: SubscribeRequest):

"""

订阅行情

"""

# 如果接口处于已登录状态，则订阅req.symbol的行情

if self.login\_status:

self.subscribeMarketData(req.symbol)

# 加入到已订阅合约集合

self.subscribed.add(req.symbol)

使用的是交易接口的典型调用方法，参“6.3 CTP接口类CtpGateway”一节。

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，详见下一节。

### 交易所

每个交易所的合约命名规则有所区别：

中金所CFFEX：字母部分大写，年份数字为2位，举例IF1908

上期所SHFE：字母部分小写，年份数字为2位，举例rb1910

能源交易所INE：字母部分小写，年份数字为2位，举例sc1910

大商所DCE：字母部分小写，年份数字为2位，举例m1911

郑商所CZCE：字母部分大写，年份数字为1位，举例TA910

对上述内容不熟悉的读者，可先执行VN Trader的查询合约功能，从中找到当前合法合约的代码。

### 本地代码vt\_symbol

字符型变量，合约在vn.py系统中的唯一代码，通常是“合约代码.交易所代码”。

在vn.py中，合约代码一般用symbol表示，是该合约在某家交易所的唯一标识；交易所一般用exchange表示，是该交易所在VN Trader内的唯一标识。

将合约代码和交易所合起来，用“.”分隔，被称为本地代码，用vt\_symbol表示，如rb2001.SHFE、IF1912.CFFEX等。使用本地代码是因为跨交易所的代码可能存在重复，比如000001在上交所代表的是上证指数，在深交所代表的则是平安银行。

vt\_symbol是操作合约的关键标识。

## 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):

"""

深度行情通知

收到Tick数据时的回调函数

"""

# 如果数据中没有UpdateTime属性，则不处理

if not data["UpdateTime"]:

return

# 取得合约代码和交易所代码

symbol = data["InstrumentID"]

exchange = symbol\_exchange\_map.get(symbol, "")

if not exchange:

return

# 生成行情时间，并进行时区转换

timestamp = f"{self.current\_date} {data['UpdateTime']}.{int(data['UpdateMillisec']/100)}"

dt = datetime.strptime(timestamp, "%Y%m%d %H:%M:%S.%f")

dt = CHINA\_TZ.localize(dt)

# 创建TickData对象保存行情数据

tick = TickData(

symbol=symbol,

exchange=exchange,

datetime=dt,

name=symbol\_name\_map[symbol],

volume=data["Volume"],

open\_interest=data["OpenInterest"],

last\_price=adjust\_price(data["LastPrice"]),

limit\_up=data["UpperLimitPrice"],

limit\_down=data["LowerLimitPrice"],

open\_price=adjust\_price(data["OpenPrice"]),

high\_price=adjust\_price(data["HighestPrice"]),

low\_price=adjust\_price(data["LowestPrice"]),

pre\_close=adjust\_price(data["PreClosePrice"]),

bid\_price\_1=adjust\_price(data["BidPrice1"]),

ask\_price\_1=adjust\_price(data["AskPrice1"]),

bid\_volume\_1=data["BidVolume1"],

ask\_volume\_1=data["AskVolume1"],

gateway\_name=self.gateway\_name

)

# 如果有十档行情，则保存

if data["BidVolume2"] or data["AskVolume2"]:

tick.bid\_price\_2 = adjust\_price(data["BidPrice2"])

tick.bid\_price\_3 = adjust\_price(data["BidPrice3"])

tick.bid\_price\_4 = adjust\_price(data["BidPrice4"])

tick.bid\_price\_5 = adjust\_price(data["BidPrice5"])

tick.ask\_price\_2 = adjust\_price(data["AskPrice2"])

tick.ask\_price\_3 = adjust\_price(data["AskPrice3"])

tick.ask\_price\_4 = adjust\_price(data["AskPrice4"])

tick.ask\_price\_5 = adjust\_price(data["AskPrice5"])

tick.bid\_volume\_2 = data["BidVolume2"]

tick.bid\_volume\_3 = data["BidVolume3"]

tick.bid\_volume\_4 = data["BidVolume4"]

tick.bid\_volume\_5 = data["BidVolume5"]

tick.ask\_volume\_2 = data["AskVolume2"]

tick.ask\_volume\_3 = data["AskVolume3"]

tick.ask\_volume\_4 = data["AskVolume4"]

tick.ask\_volume\_5 = data["AskVolume5"]

# 调用接口的on\_tick函数处理行情数据

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) -> 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) -> None:

"""

深度行情推送

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)

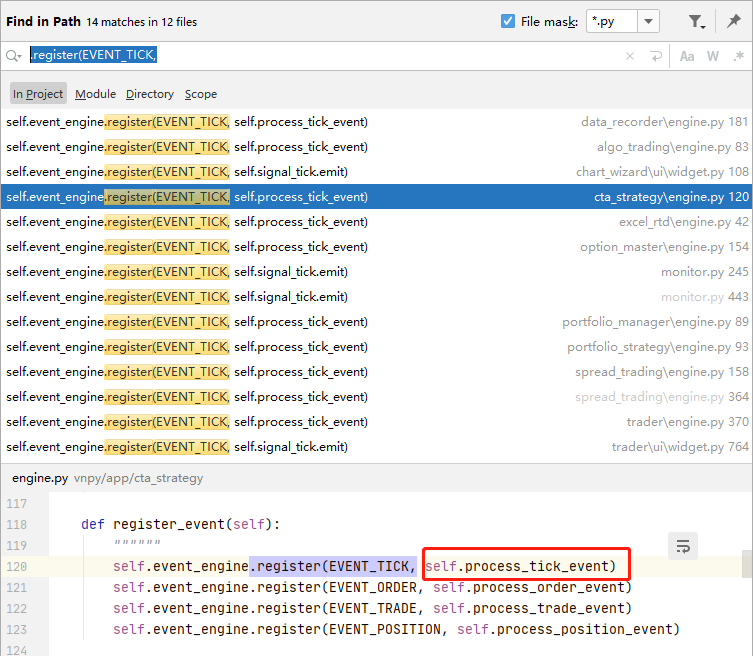
self.on\_event(EVENT\_TICK + tick.vt\_symbol, tick)

将收到的Tick数据作为事件数据，向事件引擎推送一个EVENT\_TICK事件，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

本节以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。



可以看到，系统在多处注册EVENT\_TICK事件处理函数，也就是说，从行情服务器返回的Tick数据可能在系统的多处被使用。例如：

* 数据记录器
* 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界面等

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，还可以找到一些使用“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。

假设我们关心在CTA策略引擎中如何使用Tick数据。

如前图所示，打开D:\vnpy220\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):

"""处理EVENT\_TICK事件"""

# 取得Tick数据

tick = event.data

# 取使用tick.vt\_symbol的所有策略

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

if not strategies:

return

# 检查相关的停止单

self.check\_stop\_order(tick)

# 调用各相关策略的on\_tick方法

for strategy in strategies:

if strategy.inited:

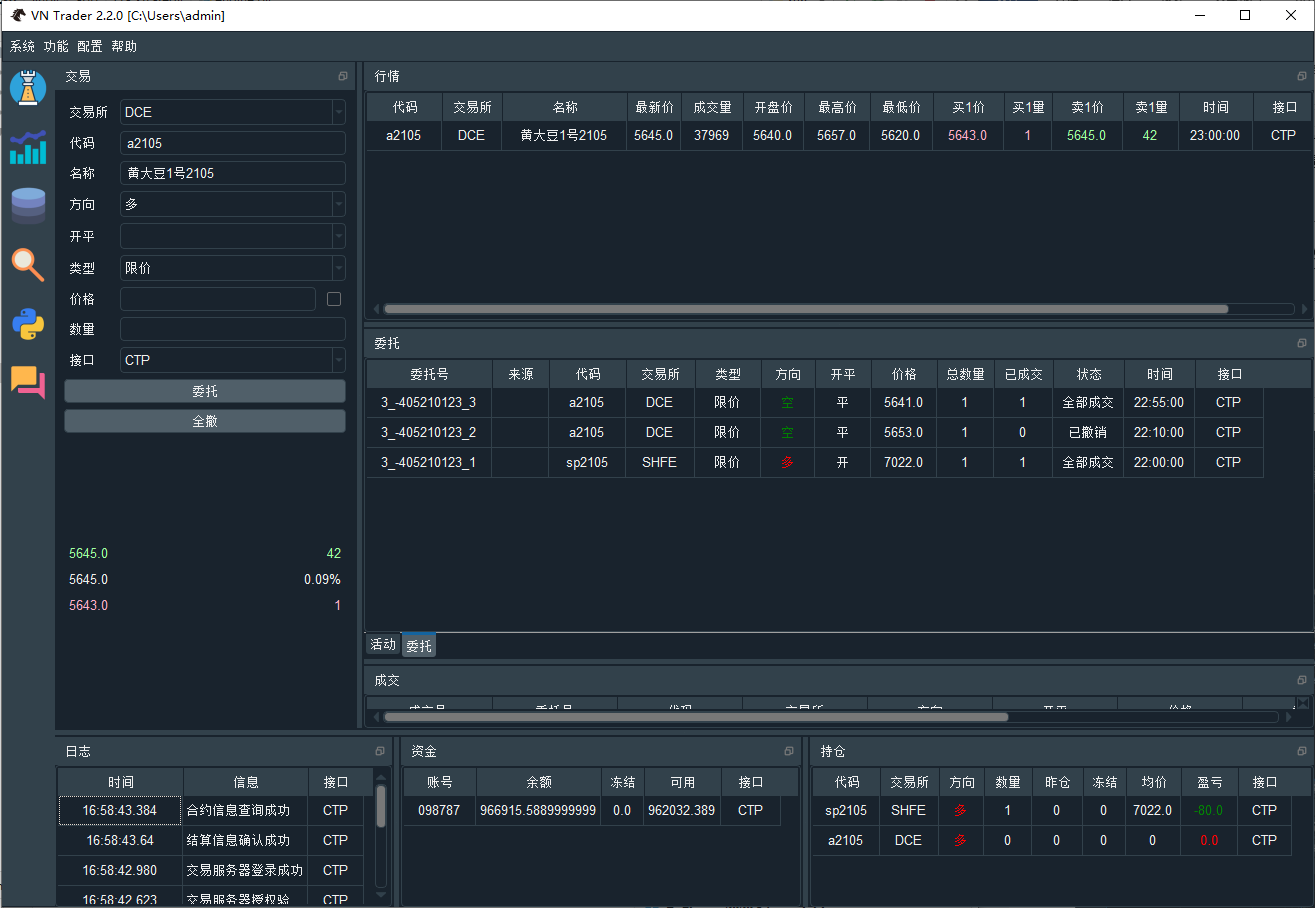
self.call\_strategy\_func(strategy, strategy.on\_tick, tick)

可以看到，CTA策略引擎可能将Tick数据应用于多个策略。

这部分内容在后文中继续研究。

## 在界面上使用Tick数据

先看VN Trader的主界面。



### 在交易子窗口中使用

在D:\vnpy220\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。

class TradingWidget(QtWidgets.QWidget):

"""

生成主窗口的交易子窗口，完成一般的手工交易

"""

# 定义Tick数据信号

signal\_tick = QtCore.pyqtSignal(Event)

……

def register\_event(self) -> None:

"""注册事件"""

# 将signal\_tick信号与槽函数相关联

self.signal\_tick.connect(self.process\_tick\_event)

# 将signal\_tick信号的发出作为EVENT\_TICK事件的处理函数

self.event\_engine.register(EVENT\_TICK, self.signal\_tick.emit)

# 为什么不直接将self.process\_tick\_event注册为事件处理函数呢？

# Qt编程中的一个重要知识点：

# 不能在Qt事件循环以外的线程中，直接调用对Qt图形组件进行修改操作，

# 否则可能因为冲突导致程序崩溃。

# 如果要这么做，只能通过Signal/Slot机制来实现这种跨线程的通知调用。

当收到EVENT\_TICK事件时会触发signal\_tick信号，signal\_tick信号的槽函数为process\_tick\_event()。process\_tick\_event()函数的代码为：

def process\_tick\_event(self, event: Event) -> None:

"""EVENT\_TICK事件处理"""

# 取tick数据

tick = event.data

if tick.vt\_symbol != self.vt\_symbol:

return

# 取得数据精度

price\_digits = self.price\_digits

# 显示最新价和买一卖一

self.lp\_label.setText(f"{tick.last\_price:.{price\_digits}f}")

self.bp1\_label.setText(f"{tick.bid\_price\_1:.{price\_digits}f}")

self.bv1\_label.setText(str(tick.bid\_volume\_1))

self.ap1\_label.setText(f"{tick.ask\_price\_1:.{price\_digits}f}")

self.av1\_label.setText(str(tick.ask\_volume\_1))

# 显示涨跌百分比

if tick.pre\_close:

r = (tick.last\_price / tick.pre\_close - 1) \* 100

self.return\_label.setText(f"{r:.2f}%")

# 显示十档行情

if tick.bid\_price\_2:

self.bp2\_label.setText(f"{tick.bid\_price\_2:.{price\_digits}f}")

self.bv2\_label.setText(str(tick.bid\_volume\_2))

self.ap2\_label.setText(f"{tick.ask\_price\_2:.{price\_digits}f}")

self.av2\_label.setText(str(tick.ask\_volume\_2))

self.bp3\_label.setText(f"{tick.bid\_price\_3:.{price\_digits}f}")

self.bv3\_label.setText(str(tick.bid\_volume\_3))

self.ap3\_label.setText(f"{tick.ask\_price\_3:.{price\_digits}f}")

self.av3\_label.setText(str(tick.ask\_volume\_3))

self.bp4\_label.setText(f"{tick.bid\_price\_4:.{price\_digits}f}")

self.bv4\_label.setText(str(tick.bid\_volume\_4))

self.ap4\_label.setText(f"{tick.ask\_price\_4:.{price\_digits}f}")

self.av4\_label.setText(str(tick.ask\_volume\_4))

self.bp5\_label.setText(f"{tick.bid\_price\_5:.{price\_digits}f}")

self.bv5\_label.setText(str(tick.bid\_volume\_5))

self.ap5\_label.setText(f"{tick.ask\_price\_5:.{price\_digits}f}")

self.av5\_label.setText(str(tick.ask\_volume\_5))

if self.price\_check.isChecked():

self.price\_line.setText(f"{tick.last\_price:.{price\_digits}f}")

### 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:\vnpy220\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，参教材16.4节。

教材的16.4节主要介绍与界面相关的知识，除显示数据外，各监控组件还可以对不同的事件进行响应，其中行情子窗口就可以对EVENT\_TICK事件进行响应。

class TickMonitor(BaseMonitor):

"""

行情子窗口

监控Tick数据

"""

event\_type = EVENT\_TICK

data\_key = "vt\_symbol"

sorting = True

headers = {

"symbol": {"display": "代码", "cell": BaseCell, "update": False},

"exchange": {"display": "交易所", "cell": EnumCell, "update": False},

"name": {"display": "名称", "cell": BaseCell, "update": True},

"last\_price": {"display": "最新价", "cell": BaseCell, "update": True},

"volume": {"display": "成交量", "cell": BaseCell, "update": True},

"open\_price": {"display": "开盘价", "cell": BaseCell, "update": True},

"high\_price": {"display": "最高价", "cell": BaseCell, "update": True},

"low\_price": {"display": "最低价", "cell": BaseCell, "update": True},

"bid\_price\_1": {"display": "买1价", "cell": BidCell, "update": True},

"bid\_volume\_1": {"display": "买1量", "cell": BidCell, "update": True},

"ask\_price\_1": {"display": "卖1价", "cell": AskCell, "update": True},

"ask\_volume\_1": {"display": "卖1量", "cell": AskCell, "update": True},

"datetime": {"display": "时间", "cell": TimeCell, "update": True},

"gateway\_name": {"display": "接口", "cell": BaseCell, "update": False},

}

如果子类中定义了event\_type，基类BaseMonitor的register\_event方法会进行注册，相关代码为：

class BaseMonitor(QtWidgets.QTableWidget):

"""

监控组件基类

"""

event\_type: str = ""

signal: QtCore.pyqtSignal = QtCore.pyqtSignal(Event)

def register\_event(self) -> None:

"""

注册事件处理函数

"""

# 如果定义了event\_type，则注册事件处理函数

if self.event\_type:

self.signal.connect(self.process\_event)

self.event\_engine.register(self.event\_type, self.signal.emit)

事件处理函数是一个通用函数，看上去适应所有事件类型，对所有监控子窗口通用，代码如下：

def process\_event(self, event: Event) -> None:

"""

事件处理函数

"""

# 暂时禁止排序功能，以防止不可预知的错误

if self.sorting:

self.setSortingEnabled(False)

# 取得事件数据

data = event.data

if not self.data\_key:

# 如果没有关键字段，直接插入新行

self.insert\_new\_row(data)

else:

# 如果有关键字段

# 取得新数据的关键字

key = data.\_\_getattribute\_\_(self.data\_key)

if key in self.cells:

# 如果该数据已经在列表中，更新原有行

self.update\_old\_row(data)

else:

# 如果该数据不在列表中，插入新行

self.insert\_new\_row(data)

# 重新开放排序功能

if self.sorting:

self.setSortingEnabled(True)

# OMS引擎

回顾教材“16.2. 主引擎”的内容，在主引擎的初始化中，初始化功能引擎的代码如下：

def init\_engines(self) -> None:

self.add\_engine(LogEngine)

self.add\_engine(OmsEngine)

self.add\_engine(EmailEngine)

调用add\_engine方法，将日志引擎、OMS引擎和邮件引擎加入到功能引擎字典中。OMS引擎在系统的运行过程中起着重要作用。

## OMS引擎的定义

OMS引擎类OmsEngine在D:\vnpy220\vnpy\trader目录下的engine.py文件中定义，与MainEngine主引擎在同一个文件中。其初始化代码为：

class OmsEngine(BaseEngine):

"""

为VN Trader提供委托单管理系统（order management system, OMS）

"""

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

""""""

super(OmsEngine, self).\_\_init\_\_(main\_engine, event\_engine, "oms")

# 行情字典，为每个vt\_symbol保留最新的Tick数据

self.ticks: Dict[str, TickData] = {}

# 委托单字典，每个vt\_orderid一个委托单

self.orders: Dict[str, OrderData] = {}

# 成交字典，每个vt\_tradeid一个成交

self.trades: Dict[str, TradeData] = {}

# 持仓字典，每个vt\_positionid一个持仓

self.positions: Dict[str, PositionData] = {}

# 账号字典，每个vt\_accountid一条账号信息

self.accounts: Dict[str, AccountData] = {}

# 合约字典，每个vt\_symbol对应一个合约

self.contracts: Dict[str, ContractData] = {}

# 活动委托单字典

self.active\_orders: Dict[str, OrderData] = {}

# 将查询函数加到主引擎

self.add\_function()

# 注册事件处理函数

self.register\_event()

## OMS引擎与主引擎的关系

把委托单交易相关的内容放到OMS引擎中，但又要通过主引擎来调用。

def add\_function(self):

"""将查询函数加到主引擎"""

self.main\_engine.get\_tick = self.get\_tick

self.main\_engine.get\_order = self.get\_order

self.main\_engine.get\_trade = self.get\_trade

self.main\_engine.get\_position = self.get\_position

self.main\_engine.get\_account = self.get\_account

self.main\_engine.get\_contract = self.get\_contract

self.main\_engine.get\_all\_ticks = self.get\_all\_ticks

self.main\_engine.get\_all\_orders = self.get\_all\_orders

self.main\_engine.get\_all\_trades = self.get\_all\_trades

self.main\_engine.get\_all\_positions = self.get\_all\_positions

self.main\_engine.get\_all\_accounts = self.get\_all\_accounts

self.main\_engine.get\_all\_contracts = self.get\_all\_contracts

self.main\_engine.get\_all\_active\_orders = self.get\_all\_active\_orders

经过上述操作，有时候看起来是调用主引擎的方法，其实调用的是OMS引擎的方法。

上述代码中的红字与合约有关，将在下一节中用到。

注册事件处理函数的代码如下：

def register\_event(self) -> None:

"""注册事件处理函数"""

self.event\_engine.register(EVENT\_TICK, self.process\_tick\_event)

self.event\_engine.register(EVENT\_ORDER, self.process\_order\_event)

self.event\_engine.register(EVENT\_TRADE, self.process\_trade\_event)

self.event\_engine.register(EVENT\_POSITION, self.process\_position\_event)

self.event\_engine.register(EVENT\_ACCOUNT, self.process\_account\_event)

self.event\_engine.register(EVENT\_CONTRACT, self.process\_contract\_event)

通过上述两段代码，将操作接口交给了主引擎，将事件处理函数注册到事件引擎。

## 查询合约功能

当vn.py通过某个交易接口（如CTP）连接成功后能够交易哪些合约呢？可以通过VN Trader的“帮助 – 查询合约”功能查询。



本节介绍“查询合约”功能背后的合约管理机制。本节的分析是按照数据使用与生成的反次序进行，先看在查询合约时如何使用数据，再看收到接口的合约数据时如何处理，再看何时向接口发送合约查询请求。

“查询合约”窗口在D:\vnpy220\vnpy\trader\ui下的widget.py文件的ContractManager类中定义。分析其“查询”按钮的处理函数show\_contracts，并没有向服务器请求查询，而是直接在主引擎中取。

def show\_contracts(self) -> None:

"""

根据代码过滤字符串显示合约

"""

# 代码过滤字符串

flt = str(self.filter\_line.text())

# 从主引擎中取所有合约

all\_contracts = self.main\_engine.get\_all\_contracts()

# 对合约进行过滤

if flt:

contracts = [

contract for contract in all\_contracts if flt in contract.vt\_symbol

]

else:

contracts = all\_contracts

# 清空合约列表

self.contract\_table.clearContents()

self.contract\_table.setRowCount(len(contracts))

# 显示所有合约

for row, contract in enumerate(contracts):

for column, name in enumerate(self.headers.keys()):

value = getattr(contract, name)

if isinstance(value, Enum):

cell = EnumCell(value, contract)

else:

cell = BaseCell(value, contract)

self.contract\_table.setItem(row, column, cell)

# 调整合约列表的显示

self.contract\_table.resizeColumnsToContents()

说明不是用户想查询时才向服务器请求，而是一旦连接了交易接口就维护一个合约字典。

通过上一节的分析可知，有时候看起来是调用主引擎的方法，其实调用的是OMS引擎的方法。如上述代码中的：

# 从主引擎中取所有合约

all\_contracts = self.main\_engine.get\_all\_contracts()

调用的其实就是OMS引擎的get\_all\_contracts方法。

## 交易接口的登录过程

合约查询不是通过行情接口，而是通过交易接口。

vn.py的处理是这样的：当交易接口登录成功后，自动查询合约信息，并将获得的回送信息放到OMS引擎中进行管理。

本节分析交易接口的登录过程，分成若干步骤，各步的程序主要在CtpTdApi(TdApi)类中实现。在D:\vnpy220\vnpy\gateway\ctp目录下，分析ctp\_gateway.py文件中CtpTdApi类的定义。

### 调用connect方法

在主界面上连接CTP接口，经过多级调用后，最终调用的是CtpTdApi类的connect方法。

调用connect方法连接交易服务器。

def connect(……):

if not self.connect\_status:

self.init()

self.connect\_status = True

else:

self.authenticate()

如果当前没有连接，则调用init函数初始化运行环境；如果已连接，进行客户端认证。

### onFrontConnected回调函数

如果连接成功，触发onFrontConnected回调函数。

def onFrontConnected(self):

if self.auth\_code:

self.authenticate()

else:

self.login()

如果self.auth\_code有值则进行客户端认证，否则进行用户登录。

认证结果返回后，触发onRspAuthenticate回调函数。

def onRspAuthenticate(self, data: dict, error: dict, reqid: int, last: bool):

"""交易服务器授权验证"""

if not error['ErrorID']:

self.auth\_status = True

self.gateway.write\_log("交易服务器授权验证成功")

self.login()

else:

self.gateway.write\_error("交易服务器授权验证失败", error)

如果认证成成功则进行用户登录。也就是终究会调用self.login()进行用户登录。

### 交易用户登录

def login(self):

……

self.reqid += 1

self.reqUserLogin(req, self.reqid)

向服务器发送登录请求。

### 登录响应

def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool):

if not error["ErrorID"]:

……

self.reqid += 1

self.reqSettlementInfoConfirm(req, self.reqid)

如果登录成功，还要进行协议确认。

### 协议确认响应

def onRspSettlementInfoConfirm(self, data: dict, error: dict, reqid: int, last: bool):

……

self.reqQryInstrument({}, self.reqid)

协议确认通过，请求查询合约。

### 查询合约响应

def onRspQryInstrument(self, data: dict, error: dict, reqid: int, last: bool):

……

self.gateway.on\_contract(contract)

收到合约信息后，调用接口的on\_contract函数进行处理。

### 交易接口中的处理

CtpGateway类中没有重写on\_contract，执行的是父类BaseGateway中的。

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

"""收到合约基础信息推送"""

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

就是简单地将收到的合约信息加入到事件引擎等待处理。

### 合约事件处理

在OMS引擎中，合约事件的处理函数为：

def process\_contract\_event(self, event: Event) -> None:

"""合约事件的处理函数"""

contract = event.data

self.contracts[contract.vt\_symbol] = contract

EVENT\_CONTRACT事件的处理很简单，就是加入到OMS引擎的合约字典中。至此，就跟前述“9.1 OMS引擎的定义”一节的内容对接起来了。

## 委托单的处理流程

前两节通过分析交易接口的登录和合约的查询，对OMS引擎有了初步了解。本节继续通过分析委托单的处理流程，进一步了解OMS引擎在系统中的作用。

vn.py有多处可以发出委托单，比如在CTA策略中自动委托。本节以VN Trader上的手工委托为例进行分析。操作方法如下，在VN Trader主界面上先连接CTP，然后在交易子窗口中选择交易所并输入合约代码，回车。输入开平方向、类型、价格和数量等，按“委托”按钮。



下面就分析按了“委托”按钮后程序是如何执行的。经过多层调用，再经过跟交易服务器的交互并经过多级返回，委托单的处理流程如下图所示。



下面分步骤进行分析。

### 主界面上的处理

在主界面上，交易子窗口在D:\vnpy220\vnpy\trader\ui\widget.py文件中的TradingWidget类中定义。其部分初始化代码如下：

class TradingWidget(QtWidgets.QWidget):

"""

生成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):

"""初始化"""

……

# 界面初始化

self.init\_ui()

def init\_ui(self) -> None:

"""界面初始化"""

……

# “委托”按钮

send\_button = QtWidgets.QPushButton("委托")

send\_button.clicked.connect(self.send\_order)

可以看到，“委托”按钮的clicked事件由函数send\_order()处理。send\_order()的代码如下：

def send\_order(self) -> None:

"""

手工发送委托单

"""

……

# 创建委托单请求

req = OrderRequest(……)

# 取接口名称

# 注：可能同时连接多个接口，需要确定向哪个接口发送委托单请求

gateway\_name = str(self.gateway\_combo.currentText())

# 调用主引擎的send\_order()函数

self.main\_engine.send\_order(req, gateway\_name)

可以看到，最终调用主引擎的send\_order()函数发送委托单。

### 主引擎中的处理

VN Trader可以同时连接多个交易接口，主引擎会选择特定的接口发送委托单。在主引擎中，相关代码如下：

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

"""

向特定接口发送一个新的委托单请求

"""

# 根据接口名称，取得接口类

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

if gateway:

# 调用接口类的send\_order()函数

return gateway.send\_order(req)

else:

return ""

对于CTP接口来说，调用的是CTP接口的send\_order()函数发送委托单。

### 在CTP接口类中的处理

send\_order()函数在接口基类中只是一个虚函数，具体实现在CTP接口类中。在D:\vnpy220\vnpy\gateway\ctp\ctp\_gateway.py文件中的CtpGateway类中定义，代码如下：

def send\_order(self, req: OrderRequest):

"""发送委托单"""

if req.type == OrderType.RFQ: # 如果是询价单

vt\_orderid = self.td\_api.send\_rfq(req)

else: # 一般的委托单

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

return vt\_orderid

CTP接口类的send\_order()函数区分一般委托单和询价单，一般委托单的发送是调用CtpTdApi类的send\_order()函数。

CtpTdApi类同样在D:\vnpy220\vnpy\gateway\ctp\ctp\_gateway.py文件中定义，其send\_order()函数代码如下：

def send\_order(self, req: OrderRequest):

"""

发送委托单

"""

# 检查开平方向

if req.offset not in OFFSET\_VT2CTP:

self.gateway.write\_log("请选择开平方向")

return ""

# 检查委托单类型

if req.type not in ORDERTYPE\_VT2CTP:

self.gateway.write\_log(f"当前接口不支持该类型的委托{req.type.value}")

return ""

# 累加委托单ID

self.order\_ref += 1

# 使用vn.py的委托单请求，创建CTP的委托单请求

ctp\_req = {……}

……

# 累加请求ID

self.reqid += 1

# 调用底层功能：

# 调用ThostFtdcTraderApi类的“报单录入请求”功能，向交易服务器发送请求

self.reqOrderInsert(ctp\_req, self.reqid)

# 生成委托号

orderid = f"{self.frontid}\_{self.sessionid}\_{self.order\_ref}"

# 根据定单请求数据创建定单数据

order = req.create\_order\_data(orderid, self.gateway\_name)

self.gateway.on\_order(order)

# 返回OrderData.vt\_orderid

return order.vt\_orderid

可以看出，最后要做两方面的工作。

1-调用底层的reqOrderInsert()函数，向服务器发送请求。

2-调用接口类的on\_order()函数。这个函数在接口基类中定义。该调用的结果是向事件引擎推送一个EVENT\_ORDER事件，此时委托单的状态为默认的“提交中”。

### 交易服务器端的处理

CTP终端报单指令ReqOrderInsert报入CTP后台，要经过数据同步状态、会话、报单字段、合约、经纪公司、投资者、是否确认结算单、交易权限、持仓资金检查和冻结、只能平仓权限检查及交易所会话检查等，任何一项检查失败则通过OnRspOrderInsert返回报单错误，在vn.py中一旦收到OnRspOrderInsert则认为委托单已被“拒单”。

通过报单检查的报单指令（ReqOrderInsert），CTP后台会向客户端返回OnRtnOrder消息，其中OrderSubmitStatus为“已经提交”，OrderStatus为“未知”。同时CTP后台将该报单指令转发至对应的交易所系统。交易所系统同样会对报单进行相应的检查，如价格是否超出涨跌停板、报单指令是否试用等等，未通过交易所系统检查的报单，CTP收到交易所系统响应后也会向客户端返回OnRtnOrder消息，其中OrderSubmitStatus为“报单已经被拒绝”，OrderStatus为“撤单”。

通过交易所系统检查的报单，交易所系统会将对应的报单插入报单薄，并通知CTP后台，CTP收到交易所系统响应后也会向客户端返回OnRtnOrder消息，其OrderSubmitStatus为“已经接受”，OrderStatus为“未成交还在队列中”。

当成交发生后（全部成交或部分成交），CTP后台将向客户端返回OnRtnTrade消息，同时也会返回OnRtnOrder消息，其中OrderSubmitStatus为“已经接受”，OrderStatus为“全部成交”或“部分成交还在队列中”。

### CTP封装类中的响应

针对委托单，在CTP封装类中要响应两类消息。回调函数onRspOrderInsert()响应OnRspOrderInsert消息，代码为：

def onRspOrderInsert(self, data: dict, error: dict, reqid: int, last: bool):

"""报单录入请求响应"""

order\_ref = data["OrderRef"]

orderid = f"{self.frontid}\_{self.sessionid}\_{order\_ref}"

symbol = data["InstrumentID"]

exchange = symbol\_exchange\_map[symbol]

order = OrderData(

symbol=symbol,

exchange=exchange,

orderid=orderid,

direction=DIRECTION\_CTP2VT[data["Direction"]],

offset=OFFSET\_CTP2VT.get(data["CombOffsetFlag"], Offset.NONE),

price=data["LimitPrice"],

volume=data["VolumeTotalOriginal"],

status=Status.REJECTED,

gateway\_name=self.gateway\_name

)

self.gateway.on\_order(order)

self.gateway.write\_error("交易委托失败", error)

如果收到此消息，说明没有通过CTP服务器端的检查，状态为“拒单”。收到的消息在处理之后，同样交给接口类的on\_order()函数处理。

回调函数onRtnOrder()响应OnRtnOrder消息，代码为：

def onRtnOrder(self, data: dict):

"""

报单通知

当委托单状态发生变化时，从服务器返回该消息

"""

if not self.contract\_inited:

self.order\_data.append(data)

return

symbol = data["InstrumentID"]

exchange = symbol\_exchange\_map[symbol]

frontid = data["FrontID"]

sessionid = data["SessionID"]

order\_ref = data["OrderRef"]

orderid = f"{frontid}\_{sessionid}\_{order\_ref}"

timestamp = f"{data['InsertDate']} {data['InsertTime']}"

dt = datetime.strptime(timestamp, "%Y%m%d %H:%M:%S")

dt = CHINA\_TZ.localize(dt)

order = OrderData(

symbol=symbol,

exchange=exchange,

orderid=orderid,

type=ORDERTYPE\_CTP2VT[data["OrderPriceType"]],

direction=DIRECTION\_CTP2VT[data["Direction"]],

offset=OFFSET\_CTP2VT[data["CombOffsetFlag"]],

price=data["LimitPrice"],

volume=data["VolumeTotalOriginal"],

traded=data["VolumeTraded"],

status=STATUS\_CTP2VT[data["OrderStatus"]],

datetime=dt,

gateway\_name=self.gateway\_name

)

self.gateway.on\_order(order)

self.sysid\_orderid\_map[data["OrderSysID"]] = orderid

委托单的状态通过CTP接口的OrderStatus字段转换获得，处理之后同样交给接口类的on\_order()函数处理。

### 接口类的on\_order()函数

综上所述，所有的委托单响应都会交给接口类的on\_order()函数处理。该函数在接口基类中定义，代码如下：

def on\_order(self, order: OrderData) -> None:

"""

委托单变化推送

发送委托请求后执行

委托请求不在这儿发，在接口的send\_order()中发，此处执行委托请求发送完成后需要做的工作，包括：

1-将一个委托单事件推送给事件引擎

2-还要将一个带有内部委托号的委托单事件推送给事件引擎

"""

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

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

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

"""

通用函数，将事件

"""

event = Event(type, data)

self.event\_engine.put(event)

可以看到，就是将处理后的委托单数据加入到事件引擎。

事件到了事件引擎后，可能被多个函数所处理，我们只分析其中两个去处：1-在主窗口的委托子窗口中显示（如下图）；2-加入到OMS引擎进行统一管理。



### 委托子窗口中的处理

委托子窗口在D:\vnpy220\vnpy\trader\ui\widget.py文件中的OrderMonitor类中定义。其部分初始化代码如下：

class OrderMonitor(BaseMonitor):

"""

委托子窗口

"""

event\_type = EVENT\_ORDER

data\_key = "vt\_orderid"

sorting = True

可以看到，在委托子窗口上会处理EVENT\_ORDER事件，以vt\_orderid为关键字。后面的处理在“8.4.2. 在行情子窗口中使用”一节已经详细介绍，这里不赘述。

### OMS引擎中的处理

在OMS引擎中，为EVENT\_ORDER事件注册了处理函数process\_order\_event()，代码为：

def register\_event(self) -> None:

"""注册事件处理函数"""

self.event\_engine.register(EVENT\_TICK, self.process\_tick\_event)

self.event\_engine.register(EVENT\_ORDER, self.process\_order\_event)

self.event\_engine.register(EVENT\_TRADE, self.process\_trade\_event)

self.event\_engine.register(EVENT\_POSITION, self.process\_position\_event)

self.event\_engine.register(EVENT\_ACCOUNT, self.process\_account\_event)

self.event\_engine.register(EVENT\_CONTRACT, self.process\_contract\_event)

def process\_order\_event(self, event: Event) -> None:

"""委托单事件处理函数"""

# 加入到委托单字典

order = event.data

self.orders[order.vt\_orderid] = order

# 如果委托是活动的，加入到活动委托单字典

if order.is\_active():

self.active\_orders[order.vt\_orderid] = order

# 否则（不活动），从活动委托单字典中移除

elif order.vt\_orderid in self.active\_orders:

self.active\_orders.pop(order.vt\_orderid)

将委托单存入委托单字典，再根据该委托是否活动进行相应处理。

第三部分 CTA回测深入分析

商品交易顾问(Commodity Trading Advisor，CTA)是指通过为客户提供期货、期权方面的交易建议，或者通过受管理的期货账户参与实际交易，来获得收益的机构或个人。1949年，美国证券经纪人理查德.唐川(Richard Donchuan)设立第一个公开发售的期货基金，标志着CTA基金的诞生。1971年，管理期货行业协会(Managed Future Association)的建立，标志着CTA正式成为业界所接受的一种投资策略。

广义上，CTA策略能够分为三大类。其中，趋势跟踪策略约占70%，均值回归（也叫价差套利）约占25%，反趋势策略约占5%。由于趋势跟踪策略所占比重巨大，国内习惯把趋势跟踪策略称为CTA策略（狭义理解），vn.py的CTA策略指的就是趋势跟踪策略，在本文的第三、四部分进行研究。vn.py也支持专门的价差套利 (Spread Trading)策略，详见第五部分。

研究vn.py的策略部分是本文的最主要目的，重点是CTA策略，采用的方法是：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的回测机制，下一部分研究CTA策略如何用于实盘。

阅读到这里，读者应该已经具备了通读代码的能力。本部分内容会大量以源码的形式呈现，在源码上增加详细的注释。

本部分采用自顶向下的分析方法，从程序界面开始，逐渐深入到引擎的内部执行。

本部分目标：从回测的角度，学习CTA策略编程。

# CTA回测界面

CTA回测是vn.py的上层应用程序。本章研究CTA回测的界面实现。

## 上层应用程序类CtaBacktesterApp

在D:\vnpy220\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: str = "" # 用于创建引擎和窗口的唯一名称

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

app\_path: str = "" # App的绝对路径

display\_name: str = "" # 显示在菜单中的名称

engine\_class = None # App的引擎类

widget\_name: str = "" # 窗口的类名称

icon\_name: str = "" # 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。

CTA回测应用类在D:\vnpy220\vnpy\app\cta\_backtester目录下的\_\_init\_\_.py文件中定义：

class CtaBacktesterApp(BaseApp):

"""CTA回测应用"""

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"

程序执行之后，部分结果如下：

app\_name = "CtaBacktester"

app\_module = "vnpy.app.cta\_backtester"

app\_path = "D:\vnpy220\vnpy\app\cta\_backtester"

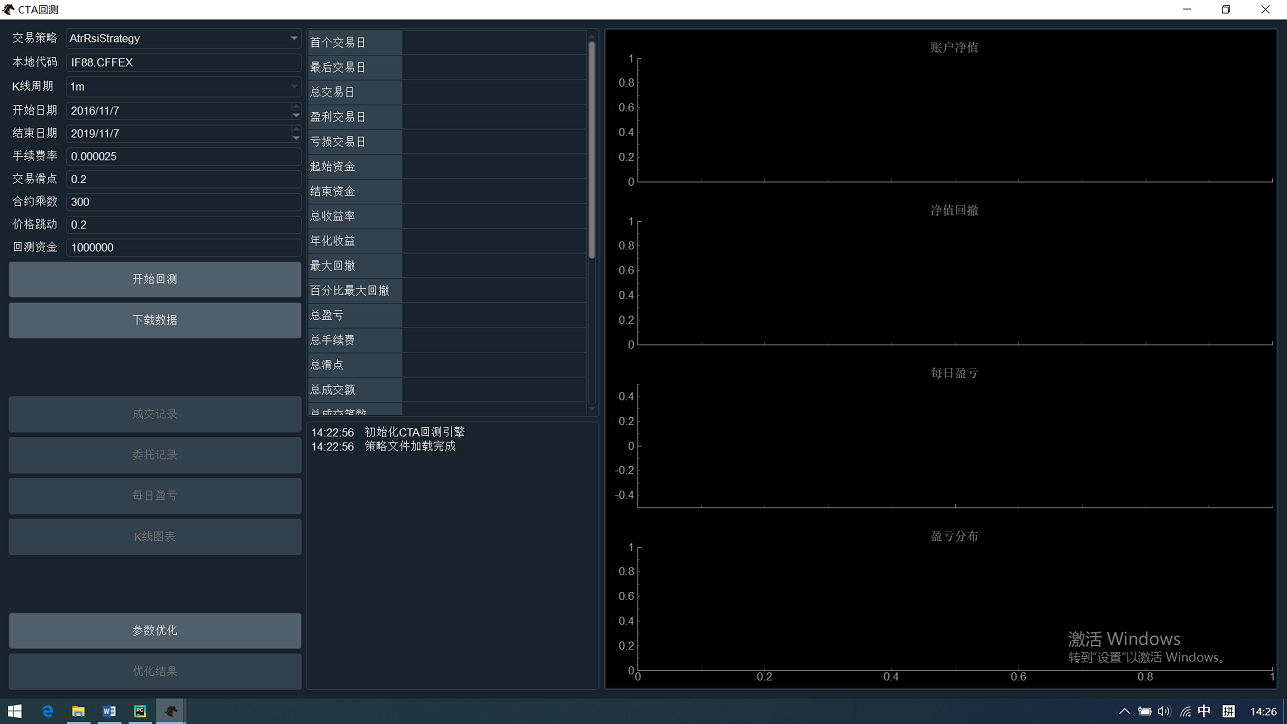
engine\_class = BacktesterEngine

widget\_name = "BacktesterManager"

执行之后，指明了引擎类是BacktesterEngine，回测窗口类是BacktesterManager。

## 回测窗口类BacktesterManager

执行时CTA回测窗口的界面如下图所示。



在D:\vnpy220\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。

### 窗口初始化

窗口初始化的代码为：

class BacktesterManager(QtWidgets.QWidget):

"""回测窗体类"""

# 保存回测配置信息的文件

setting\_filename = "cta\_backtester\_setting.json"

# 定义几个信号

# 日志信号，需要显示日志信息时发出

signal\_log = QtCore.pyqtSignal(Event)

# 回测结束信号，在回测结束时发出

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

# 优化结束信号，在优化结束时发出

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

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

"""初始化"""

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

self.main\_engine = main\_engine

self.event\_engine = event\_engine

# 取得回测引擎，执行时值为<vnpy.app.cta\_backtester.engine.BacktesterEngine>

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

# 策略名称列表，包括两个目录（参教材15.4节）中所有可供回测的策略

self.class\_names = []

# 策略字典，key是策略名称，与self.class\_names对应，

# value又是一个字典，包括对应策略的所有参数和默认值

self.settings = {}

# 优化目标

self.target\_display = ""

# 界面初始化

self.init\_ui()

# 注册事件：注册回测日志事件、回测完成事件和回测优化完成事件的处理函数

self.register\_event()

# 初始化回测引擎

self.backtester\_engine.init\_engine()

# 加载策略：功能是加载self.class\_names和self.settings

self.init\_strategy\_settings()

# 取界面配置参数

self.load\_backtesting\_setting()

与界面初始化相关代码比较简单，对阅读到这里的读者已经不成问题，不再详述。

其中，init\_strategy\_settings方法从两个目录（参教材15.4节）中所有策略的源代码中加载策略，并保存到self.class\_names和self.settings中。

在上图“CTA回测窗口”的左上部，如下图所示。



上一次执行回测操作时该部分界面的设置信息保存在系统工作目录的cta\_backtester\_setting.json文件中。load\_backtesting\_setting方法取出该设置，并修改界面，相关代码请读者自行分析。

### 加载策略

BacktesterManager类的init\_strategy\_settings方法从两个目录（参教材15.4节）中所有策略的源代码中加载策略，并保存到self.class\_names和self.settings中。

def init\_strategy\_settings(self):

"""加载策略"""

# 从回测引擎取策略名，保存到self.class\_names列表

self.class\_names = self.backtester\_engine.get\_strategy\_class\_names()

# 对于每个策略

for class\_name in self.class\_names:

# 取策略的默认配置参数

setting = self.backtester\_engine.get\_default\_setting(class\_name)

# 保存到self.settings字典

self.settings[class\_name] = setting

# 将所有策略名称加入到下拉式列表self.class\_combo中，对应界面的“交易策略”列表

self.class\_combo.addItems(self.class\_names)

主要是调用回测引擎的get\_strategy\_class\_names函数，相关代码为：

class BacktesterEngine(BaseEngine):

"""回测引擎类"""

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

self.classes = {} # 策略类字典

def load\_strategy\_class(self):

"""

从源代码中加载策略类，结果存储在self.classes中

"""

……

def get\_strategy\_class\_names(self):

"""策略名称列表"""

return list(self.classes.keys())

可见，程序的关键是load\_strategy\_class方法，将在本文第16章详细说明。

存在问题：

如果在load\_strategy\_class(self)函数中加载策略失败，会推送一个EVENT\_BACKTESTER\_LOG事件。而EVENT\_BACKTESTER\_LOG事件的处理函数在窗口（BacktesterManager）初始化时注册，处理也是将日志显示在界面上。

由于加载策略时窗口还未初始化，所以加载策略失败的日志信息不会被显示。

### 各窗口控件

比较简单，略。

# 通用实用功能utility.py

vn.py将一些底层的、通用的功能集中在D:\vnpy220\vnpy\trader目录的utility.py文件中，本章就对该文件进行分析。该文件中定义了一些通用函数，还定义了两个类，本章主要分析这两个类。

## K线合成器BarGenerator

代码中有详细的注释，相关代码如下：

class BarGenerator:

"""

K线合成器，支持：

1. 基于Tick合成1分钟K线

2. 基于1分钟K线合成X分钟K线/X小时K线

注意：

对于X分钟K线，X必须能被60整除，如：2, 3, 5, 6, 10, 15, 20, 30

对于X小时K线，X可以是任意值

"""

def \_\_init\_\_(

self,

on\_bar: Callable, # on\_bar回调函数作为参数传，意味着用外部函数作为on\_bar回调

window: int = 0, # 数据窗口大小，对应要生成的“X分钟K线”的X

on\_window\_bar: Callable = None, # “X分钟K线”生成后的回调函数

interval: Interval = Interval.MINUTE # K线周期，指定是生成分钟线还是小时线，默认为分钟

):

"""构造函数"""

self.bar: BarData = None # 1分钟K线对象

self.on\_bar: Callable = on\_bar # 1分钟K线回调函数

self.interval: Interval = interval # （目标）K线周期

self.interval\_count: int = 0 # K线周期计数，在0~self.window之间循环

self.hour\_bar: BarData = None # 存储生成的小时线数据

self.window: int = window # 数据窗口大小，对应要生成的“X分钟K线”的X

self.window\_bar: BarData = None # 一个BarData对象，存储生成的“X分钟K线”

self.on\_window\_bar: Callable = on\_window\_bar # “X分钟K线”生成后的回调函数

self.last\_tick: TickData = None # 上一Tick缓存对象

self.last\_bar: BarData = None # 上一K线缓存对象

def update\_tick(self, tick: TickData) -> None:

"""

更新一个新的Tick数据到合成器

功能：根据Tick数据生成1分钟K线数据

"""

new\_minute = False # 默认不是新的一分钟

# 过滤掉last price为0的Tick数据

if not tick.last\_price:

return

# 过滤掉比“上一Tick”还老的数据

if self.last\_tick and tick.datetime < self.last\_tick.datetime:

return

# 如果尚未创建对象

if not self.bar:

new\_minute = True

elif (

(self.bar.datetime.minute != tick.datetime.minute)

or (self.bar.datetime.hour != tick.datetime.hour)

):

# 如果新收到的Tick与上一Tick的分钟不同（新的一分钟）

# 生成上一分钟K线的时间戳

self.bar.datetime = self.bar.datetime.replace(

second=0, microsecond=0

) # 将秒和微秒设为0

# 推送已经结束的上一分钟K线

self.on\_bar(self.bar)

new\_minute = True

if new\_minute:

# 如果新的分钟，初始化新一分钟的K线数据

self.bar = BarData(

symbol=tick.symbol,

exchange=tick.exchange,

interval=Interval.MINUTE,

datetime=tick.datetime,

gateway\_name=tick.gateway\_name,

open\_price=tick.last\_price,

high\_price=tick.last\_price,

low\_price=tick.last\_price,

close\_price=tick.last\_price,

open\_interest=tick.open\_interest

)

else:

# 如果不是新的分钟，累加更新老一分钟的K线数据

self.bar.high\_price = max(self.bar.high\_price, tick.last\_price)

if tick.high\_price > self.last\_tick.high\_price:

self.bar.high\_price = max(self.bar.high\_price, tick.high\_price)

self.bar.low\_price = min(self.bar.low\_price, tick.last\_price)

if tick.low\_price < self.last\_tick.low\_price:

self.bar.low\_price = min(self.bar.low\_price, tick.low\_price)

self.bar.close\_price = tick.last\_price

self.bar.open\_interest = tick.open\_interest

self.bar.datetime = tick.datetime

# 如果有上一Tick

if self.last\_tick:

# 当前K线内的成交量

volume\_change = tick.volume - self.last\_tick.volume

# 避免夜盘开盘lastTick.volume为昨日收盘数据，导致成交量变化为负的情况

self.bar.volume += max(volume\_change, 0)

# 缓存Tick

self.last\_tick = tick

def update\_bar(self, bar: BarData) -> None:

"""

更新一个新的1分钟K线数据到合成器

根据要生成X分钟线还是X小时线，进行不同的处理

"""

if self.interval == Interval.MINUTE:

# 如果要生成X分钟线

self.update\_bar\_minute\_window(bar)

else:

# 如果要生成X小时线

self.update\_bar\_hour\_window(bar)

def update\_bar\_minute\_window(self, bar: BarData) -> None:

"""

更新一个新的1分钟K线数据到合成器

生成X分钟数据

"""

# 如果还没有创建，现在创建一个window bar对象

if not self.window\_bar:

# 生成K线数据的时间戳（秒以下级别都为0）

dt = bar.datetime.replace(second=0, microsecond=0)

# 创建window bar对象

self.window\_bar = BarData(

symbol=bar.symbol,

exchange=bar.exchange,

datetime=dt,

gateway\_name=bar.gateway\_name,

open\_price=bar.open\_price,

high\_price=bar.high\_price,

low\_price=bar.low\_price

)

# 如果已经创建window bar对象，更新其最高/最低价

else:

self.window\_bar.high\_price = max(

self.window\_bar.high\_price,

bar.high\_price

)

self.window\_bar.low\_price = min(

self.window\_bar.low\_price,

bar.low\_price

)

# 更新收盘价和成交量

self.window\_bar.close\_price = bar.close\_price

self.window\_bar.volume += int(bar.volume)

self.window\_bar.open\_interest = bar.open\_interest

# 检查window bar是否已经完成

if not (bar.datetime.minute + 1) % self.window:

# 如果window bar已经完成，调用on\_window\_bar()

# 注意：on\_window\_bar()是回调函数，由主调模块指定

# 也就是本模块可以为主调模块生成X分钟K线数据

self.on\_window\_bar(self.window\_bar)

# 清除window\_bar

# 如果再有1分钟K线到来，会生成新的window\_bar

self.window\_bar = None

# 缓存新的1分钟K线数据

self.last\_bar = bar

# 高注：本函数虽然没有明说，但默认是接受1分钟数据

# 如果来的是5分钟数据，则上述“检查window bar是否已经完成”的条件永远都不会满足

# 永远都生成不了window bar

def update\_bar\_hour\_window(self, bar: BarData) -> None:

"""

更新一个新的1分钟K线数据到合成器

生成X小时线数据

注：本函数中先生成1小时线，再调用on\_hour\_bar()生成X小时数据

"""

# 如果还没有创建，现在创建一个hour\_bar对象

if not self.hour\_bar:

# 生成小时K线数据的时间戳（分钟以下级别都为0）

dt = bar.datetime.replace(minute=0, second=0, microsecond=0)

# 创建hour\_bar对象

self.hour\_bar = BarData(

symbol=bar.symbol,

exchange=bar.exchange,

datetime=dt,

gateway\_name=bar.gateway\_name,

open\_price=bar.open\_price,

high\_price=bar.high\_price,

low\_price=bar.low\_price,

volume=bar.volume

)

return

# finished\_bar缓存生成的小时线，默认还没有

finished\_bar = None

# 如果当前分钟是59，用分钟数据更新小时数据并推送

# If minute is 59, update minute bar into window bar and push

if bar.datetime.minute == 59:

# 更新最高价

self.hour\_bar.high\_price = max(

self.hour\_bar.high\_price,

bar.high\_price

)

# 更新最低价

self.hour\_bar.low\_price = min(

self.hour\_bar.low\_price,

bar.low\_price

)

# 更新收盘价和成交量

self.hour\_bar.close\_price = bar.close\_price

self.hour\_bar.volume += int(bar.volume)

self.hour\_bar.open\_interest = bar.open\_interest

# 缓存生成的小时线数据，并清除self.hour\_bar

finished\_bar = self.hour\_bar

self.hour\_bar = None

# 如果是新的小时，推送已经存在的hour\_bar

elif bar.datetime.hour != self.hour\_bar.datetime.hour:

finished\_bar = self.hour\_bar

# 上面一句可以适应两种情况：

# 1-前一分钟是59，则前一分钟已推送，hour\_bar为None，不会重复推送

# 2-前一分钟不是59（行情数据缺失），hour\_bar为为None，则会在此处推送

# 因为是新的小时，所以在此创建hour\_bar

# 生成小时K线数据的时间戳（分钟以下级别都为0）

dt = bar.datetime.replace(minute=0, second=0, microsecond=0)

# 创建hour\_bar对象

self.hour\_bar = BarData(

symbol=bar.symbol,

exchange=bar.exchange,

datetime=dt,

gateway\_name=bar.gateway\_name,

open\_price=bar.open\_price,

high\_price=bar.high\_price,

low\_price=bar.low\_price,

volume=bar.volume

)

# 如果既不是59分，也不是新小时的开始，只更新小时数据

else:

# 更新最高价

self.hour\_bar.high\_price = max(

self.hour\_bar.high\_price,

bar.high\_price

)

# 更新最低价

self.hour\_bar.low\_price = min(

self.hour\_bar.low\_price,

bar.low\_price

)

# 更新收盘价和成交量

self.hour\_bar.close\_price = bar.close\_price

self.hour\_bar.volume += int(bar.volume)

self.hour\_bar.open\_interest = bar.open\_interest

# 如果有新的小时数据，推送给on\_hour\_bar()

if finished\_bar:

# 调用on\_hour\_bar()生成X小时数据

self.on\_hour\_bar(finished\_bar)

# 缓存新的1分钟K线数据

self.last\_bar = bar

def on\_hour\_bar(self, bar: BarData) -> None:

"""用1小时K线生成X小时K线"""

if self.window == 1:

# 如果X==1，直接返回小时线

self.on\_window\_bar(bar)

else:

# 如果还没有创建，现在创建一个window bar对象

if not self.window\_bar:

# 创建window bar对象

self.window\_bar = BarData(

symbol=bar.symbol,

exchange=bar.exchange,

datetime=bar.datetime,

gateway\_name=bar.gateway\_name,

open\_price=bar.open\_price,

high\_price=bar.high\_price,

low\_price=bar.low\_price

)

else:

# 如果已经创建window bar对象

# 更新其最高/最低价

self.window\_bar.high\_price = max(

self.window\_bar.high\_price,

bar.high\_price

)

self.window\_bar.low\_price = min(

self.window\_bar.low\_price,

bar.low\_price

)

# 更新收盘价和成交量

self.window\_bar.close\_price = bar.close\_price

self.window\_bar.volume += int(bar.volume)

self.window\_bar.open\_interest = bar.open\_interest

# K线周期计数+1

self.interval\_count += 1

if not self.interval\_count % self.window:

# 如果已经达到X小时，重新计数

self.interval\_count = 0

# 调用on\_window\_bar()

# 注意：on\_window\_bar()是回调函数，由主调模块指定

# 也就是本模块可以为主调模块生成X小时K线数据

self.on\_window\_bar(self.window\_bar)

# 清除window\_bar

# 如果再有1小时K线到来，会生成新的window\_bar

self.window\_bar = None

def generate(self) -> Optional[BarData]:

"""

强制立即完成K线合成并回调

"""

bar = self.bar

if self.bar:

bar.datetime = bar.datetime.replace(second=0, microsecond=0)

self.on\_bar(bar)

self.bar = None

return bar

BarData的定义参“VN Trader的全局定义”一节。

存在问题：update\_bar函数只能基于1分钟K线合成X分钟K线/X小时K线。假设遇到这种情况：回测策略是基于5分钟的，数据库中也有5分钟的数据。但是，从数据库中取K线时，每个K线被当成一个1分钟K线，按照上面update\_bar\_minute\_window()函数的逻辑（if not (bar.datetime.minute + 1) % self.window），永远都不能生成。

解决方法：

1-修改上面函数。看起来倒是比较简单，但我不想在人家源码的基础上改，至少目前不想。等人家下次更新了版本，还得把我改过的源码再改一遍。如果将来我准备基于某个版本大改程序时，就无所谓了。

2-增强数据源的能力。比如申请rq账号，比如取通达信的本地行情。其实有第一部分“加载数据”一节的内容，也差不多了。

3-（也是我现在暂时用的方法）修改自己的策略，在on\_bar()中不调用self.bg.update\_bar(bar)，而是直接使用。

上述方法读者现在还不容易理解，等实际开发策略时就能理解。如果编程能力强，无论遇到什么问题都可以编程解决；如果不想费编程的劲，解决好数据源，就不会出现这些问题了。

## K线序列管理工具ArrayManager

如果策略中需要使用一些K线数据进行指标计算，就需要用到ArrayManager。ArrayManager的结构如下图所示。



在ArrayManager中维护self.open\_array等几个缓存数组，用于存储一定数量的K线数据。通过这些数据可以计算各种技术指标，用于支持策略的逻辑。

缓存数组的大小为self.size。self.count为数据计数，开始时为0，随着数据的到来而增加。新来的数据从数组的尾部（图中右侧）进入，原有数据需要左移一位，为新来的数据腾出位置。当self.count小于self.size时缓存数组未满，self.inited为False，表示尚未初始化完成，此时策略无法触发交易。当self.count大于等于self.size时缓存数组已满，self.inited置为True，此时为给新来的数据腾出空间，最左侧的数据会丢失，所以在设置self.size时要能保证所需技术指标的计算，太大太小都不好。当self.inited为True时是否允许策略进行交易，还要看其它条件。

ArrayManager的部分代码如下：

class ArrayManager(object):

"""

K线序列管理工具，负责：

1. K线时间序列的维护

2. 常用技术指标的计算

"""

def \_\_init\_\_(self, size: int = 100):

"""构造函数"""

self.count: int = 0 # 缓存计数

self.size: int = size # 缓存大小

self.inited: bool = False # 当count>=size时为True

# OHLC、成交量、持仓量数组

self.open\_array: np.ndarray = np.zeros(size)

self.high\_array: np.ndarray = np.zeros(size)

self.low\_array: np.ndarray = np.zeros(size)

self.close\_array: np.ndarray = np.zeros(size)

self.volume\_array: np.ndarray = np.zeros(size)

self.open\_interest\_array: np.ndarray = np.zeros(size)

def update\_bar(self, bar: BarData) -> None:

"""

将K线更新到array manager

新K线放到数组的尾部，原K线前移

当数组中的K线放满后，self.inited才设为True

"""

self.count += 1

if not self.inited and self.count >= self.size:

self.inited = True

# 向前移动一个K线

self.open\_array[:-1] = self.open\_array[1:]

self.high\_array[:-1] = self.high\_array[1:]

self.low\_array[:-1] = self.low\_array[1:]

self.close\_array[:-1] = self.close\_array[1:]

self.volume\_array[:-1] = self.volume\_array[1:]

self.open\_interest\_array[:-1] = self.open\_interest\_array[1:]

# 将新K线放到最后一个元素中

self.open\_array[-1] = bar.open\_price

self.high\_array[-1] = bar.high\_price

self.low\_array[-1] = bar.low\_price

self.close\_array[-1] = bar.close\_price

self.volume\_array[-1] = bar.volume

self.open\_interest\_array[-1] = bar.open\_interest

@property

def open(self) -> np.ndarray:

"""

获取开盘价序列

"""

return self.open\_array

@property

def high(self) -> np.ndarray:

"""

获取最高价序列

"""

return self.high\_array

@property

def low(self) -> np.ndarray:

"""

获取最低价序列

"""

return self.low\_array

@property

def close(self) -> np.ndarray:

"""

获取收盘价序列

"""

return self.close\_array

@property

def volume(self) -> np.ndarray:

"""

获取成交量序列

"""

return self.volume\_array

@property

def open\_interest(self) -> np.ndarray:

"""

获取持仓量序列

"""

return self.open\_interest\_array

def sma(self, n: int, array: bool = False) -> Union[float, np.ndarray]:

"""

简单移动平均线

"""

result = talib.SMA(self.close, n)

if array:

return result

return result[-1]

def ema(self, n: int, array: bool = False) -> Union[float, np.ndarray]:

"""

指数移动平均线

"""

result = talib.EMA(self.close, n)

if array:

return result

return result[-1]

def kama(self, n: int, array: bool = False) -> Union[float, np.ndarray]:

"""

KAMA指标

"""

result = talib.KAMA(self.close, n)

if array:

return result

return result[-1]

……还有多种指标，参差版本的更新还会越来越多。

注意其中的size参数，该参数决定的缓存的大小，默认为100。如果在策略中计算指标需要更多的K线，就应该指定更大的size值，否则策略执行时会出错，得不到交易信号。

vn.py所支持的技术指标每个版本都有增加。如果仍然不够，需要自己写程序。一方面可以进一步利用talib包的功能，另一方面可以写自己的指标，如缠论的买卖点等。

# 一般CTA策略

vn.py支持的CTA策略有两种写法，一种是一般CTA策略，一种是目标持仓策略。本章先讨论一般CTA策略。

## CTA策略模板CtaTemplate

在D:\vnpy220\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，读者应该通读源代码。

class CtaTemplate(ABC):

"""CTA策略模板"""

author = "" # 策略作者

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,

):

"""构造函数"""

self.cta\_engine = cta\_engine # CTA引擎

self.strategy\_name = strategy\_name # 策略名称

self.vt\_symbol = vt\_symbol # 合约

self.inited = False # 是否已初始化

self.trading = False # 是否正在交易

self.pos = 0 # 持仓数量

# 复制一个新的变量列表，这样，就可以在用相同的策略类创建了多个策略实例之后，

# 进行重复的插入。

# 注：此处是基类，变量列表为空，看起来下面复制一句没用。

# 具体的策略带有变量，此句就有用了。

self.variables = copy(self.variables)

self.variables.insert(0, "inited")

self.variables.insert(1, "trading")

self.variables.insert(2, "pos")

# 设置策略的参数

self.update\_setting(setting)

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

"""

用配置字典内的值作为策略属性

"""

for name in self.parameters:

if name in setting:

setattr(self, name, setting[name])

@classmethod

def get\_class\_parameters(cls):

"""

取策略类默认的参数（供回测引擎调用）

"""

class\_parameters = {}

for name in cls.parameters:

class\_parameters[name] = getattr(cls, name)

return class\_parameters

def get\_parameters(self):

"""

取本策略具体的参数

"""

strategy\_parameters = {}

for name in self.parameters:

strategy\_parameters[name] = getattr(self, name)

return strategy\_parameters

def get\_variables(self):

"""

取策略变量字典

"""

strategy\_variables = {}

for name in self.variables:

strategy\_variables[name] = getattr(self, name)

return strategy\_variables

def get\_data(self):

"""

取策略数据"

"""

strategy\_data = {

"strategy\_name": self.strategy\_name,

"vt\_symbol": self.vt\_symbol,

"class\_name": self.\_\_class\_\_.\_\_name\_\_,

"author": self.author,

"parameters": self.get\_parameters(),

"variables": self.get\_variables(),

}

return strategy\_data

@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）

"""

pass

@virtual

def on\_start(self):

"""

策略启动时的回调函数（必须由用户继承实现）

"""

pass

@virtual

def on\_stop(self):

"""

策略停止时的回调函数（必须由用户继承实现）

"""

pass

@virtual

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

"""

收到行情TICK推送的回调函数（必须由用户继承实现）

"""

pass

@virtual

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

"""

收到Bar推送的回调函数（必须由用户继承实现）

"""

pass

@virtual

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

"""

收到成交推送的回调函数（必须由用户继承实现）

"""

pass

@virtual

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

"""

收到委托变化推送的回调函数（必须由用户继承实现）

"""

pass

@virtual

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

"""

收到停止单推送的回调函数（必须由用户继承实现）

"""

pass

def buy(

self,

price: float,

volume: float,

stop: bool = False,

lock: bool = False,

net: bool = False

):

"""

买开：发送多单开多仓

"""

return self.send\_order(

Direction.LONG,

Offset.OPEN,

price,

volume,

stop,

lock,

net

)

def sell(

self,

price: float,

volume: float,

stop: bool = False,

lock: bool = False,

net: bool = False

):

"""

卖平：发送空单平多仓

"""

return self.send\_order(

Direction.SHORT,

Offset.CLOSE,

price,

volume,

stop,

lock,

net

)

def short(

self,

price: float,

volume: float,

stop: bool = False,

lock: bool = False,

net: bool = False

):

"""

卖开：发送空单开空仓

"""

return self.send\_order(

Direction.SHORT,

Offset.OPEN,

price,

volume,

stop,

lock,

net

)

def cover(

self,

price: float,

volume: float,

stop: bool = False,

lock: bool = False,

net: bool = False

):

"""

买平：发送多单平空仓

"""

return self.send\_order(

Direction.LONG,

Offset.CLOSE,

price,

volume,

stop,

lock,

net

)

def send\_order(

self,

direction: Direction,

offset: Offset,

price: float,

volume: float,

stop: bool = False,

lock: bool = False,

net: bool = False

):

"""

发送委托

"""

if self.trading:

vt\_orderids = self.cta\_engine.send\_order(

self, direction, offset, price, volume, stop, lock, net

)

return vt\_orderids

else:

return []

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

"""

撤消一个已经存在的委托

"""

if self.trading:

self.cta\_engine.cancel\_order(self, vt\_orderid)

def cancel\_all(self):

"""

撤消由策略发出的所有委托

"""

if self.trading:

self.cta\_engine.cancel\_all(self)

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

"""

写一条日志信息

"""

self.cta\_engine.write\_log(msg, self)

def get\_engine\_type(self):

"""

返回CTA引擎是在回测还是实盘（live trading）

"""

return self.cta\_engine.get\_engine\_type()

def get\_pricetick(self):

"""

返回合约的价格跳动

"""

return self.cta\_engine.get\_pricetick(self)

def load\_bar(

self,

days: int,

interval: Interval = Interval.MINUTE,

callback: Callable = None,

use\_database: bool = False

):

"""

在初始化策略时加载历史Bar数据

"""

# 如果参数callback为None，指定self.on\_bar为回调函数

if not callback:

callback = self.on\_bar

self.cta\_engine.load\_bar(

self.vt\_symbol,

days,

interval,

callback,

use\_database

)

# 注意：在要求self.cta\_engine加载数据时，必须指明回调函数，

# 用于将数据推送回本策略

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

"""

在初始化策略时加载历史Tick数据

"""

self.cta\_engine.load\_tick(self.vt\_symbol, days, self.on\_tick)

# 注意：在要求self.cta\_engine加载数据时，必须指明回调函数，

# 用于将数据推送回本策略

def put\_event(self):

"""

发出由界面变化引起的策略数据事件

"""

if self.inited:

self.cta\_engine.put\_strategy\_event(self)

def send\_email(self, msg):

"""

向默认的接收者发送邮件

"""

if self.inited:

self.cta\_engine.send\_email(msg, self)

def sync\_data(self):

"""

同步策略变量到磁盘存储器

"""

if self.trading:

self.cta\_engine.sync\_strategy\_data(self)

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。

在同一个文件中还定义了CTA信号类和目标持仓模板类，详细说明见“目标持仓策略”一章。

### 关于变量列表

在CtaTemplate模板类定义的开始处，定义了一个类属性variables，该属性会存储一系列与策略相关的变量。

variables = [] # 变量列表

在该类的构造函数中有如下一句，比较不好理解。

self.variables = copy(self.variables)

事实上，该句的作用是将类属性值复制到实例属性中。关于类属性的说明请参考教材“7.3.类属性和类方法”一节。对教材该节的示例稍加修改如下：

class Person:

count = 0 # 声明一个类属性

def \_\_init\_\_(self):

Person.count = Person.count + 1

def test(self): # 测试功能

# 赋值号右侧取的是类属性，但左侧的self.count又创建了一个新的实例属性

self.count = self.count

print(self.count) # 输出实例属性值，与类属性相等

self.count = 0 # 改变实例属性的值

print(self.count) # 输出改变后的实例属性的值

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

zhang = Person() # 创建第一个实例

print(Person.count)

print(zhang.count)

wang = Person() # 创建第二个实例

print(Person.count) # 通过类名称访问，值为2

print(wang.count) # 通过实例名称访问，值也为2，访问的是类属性

wang.test() # 执行测试功能

print(wang.count) # 访问的是改变后的实例属性，值为0

print(Person.count) # 通过类名访问类属性，值仍为2

print(zhang.count) # 通过另一个实例访问的仍然是类属性，值仍为2

执行结果：

1

1

2

2

2

0

0

2

2

这样下面一句就好理解了。

self.variables = copy(self.variables)

它的作用是将类属性的值复制到实例属性，这样每个策略实例就有了自己的变量复本。

### 是否正在交易标志trading

在策略的初始化阶段往往需要先加载若干天（一般用days表示）的行情数据。例如，在有的策略的on\_start事件中会调用self.load\_bar(10)。这样可以避免在数据不足时发出错误的交易信号。

这些数据也用通常的方法加载到策略中，也可能产生交易信号，这些信号需要忽略掉。所以在策略中使用一个是否正在交易标志trading。开始时trading被设为False，此时即使产生交易信号，也不进行实际的交易。当策略初始化完成进入到交易阶段后，trading被设为True，此时交易信号才会触发真正的交易。注意，在上述模板类的send\_order和cancel\_order等函数中，都会对trading标志进行判断。

注：当trading为True时是否允许交易还要看其它条件，如ArrayManager的inited是否为True，详见本文档11.2节。

### 策略数据事件

策略模板类提供的put\_event函数，在策略中会被经常调用。

该函数直接调用cta\_engine的put\_strategy\_event函数。cta\_engine根据应用场合，可能是CTA回测执行引擎类，也可能是CTA策略引擎类。在CTA回测执行引擎类中，该函数什么都不执行；在CTA策略引擎类中，该函数将策略数据打包成事件数据，交给事件引擎处理。

因此，put\_event函数虽然没有参数，看起来好像事件是无差别的，其实不是。

### 交易函数的参数说明

buy、sell、short和cover等交易函数具有相同的参数列表，本小节统一说明。

* price：委托价格。vn.py不支持市价单，因此委托时必须指明一个价格。从后面的分析中可以看到，策略中可以用涨/跌停价格来做委托价格，实际起到市价单的效果。
* volume：委托的数量。
* stop：布尔型参数，默认为False，表示委托的是否为停止单。
* lock：布尔型参数，默认为False，表示委托的是否为锁仓单。

这些参数在策略引擎的各个层次都会用到，含义相同。

## 一般CTA策略示例DoubleMaStrategy

下面以一个系统自带的策略为例，介绍一般CTA策略的写法。DoubleMaStrategy为双均线策略，该策略用到两条均线，分别是快均线（10天均线）和慢均线（20天均线）。当快均线上升穿过慢均线时，形成金叉，给出买入信号。反之，当快均线下跌穿过慢均线时，形成死叉，给出卖出信号。

打开D:\vnpy220\vnpy\app\cta\_strategy\strategies目录下的文件double\_ma\_strategy.py，增加了注释的代码如下。

from vnpy.app.cta\_strategy import (

CtaTemplate,

StopOrder,

TickData,

BarData,

TradeData,

OrderData,

BarGenerator,

ArrayManager,

)

"""

这个Demo是一个最简单的双均线策略实现

"""

class DoubleMaStrategy(CtaTemplate):

author = "用Python的交易员"

fast\_window = 10

slow\_window = 20

fast\_ma0 = 0.0

fast\_ma1 = 0.0

slow\_ma0 = 0.0

slow\_ma1 = 0.0

parameters = ["fast\_window", "slow\_window"]

variables = ["fast\_ma0", "fast\_ma1", "slow\_ma0", "slow\_ma1"]

def \_\_init\_\_(self, cta\_engine, strategy\_name, vt\_symbol, setting):

"""构造函数"""

super().\_\_init\_\_(cta\_engine, strategy\_name, vt\_symbol, setting)

# K线合成器，在本策略中用于由Tick数据生成K线数据

self.bg = BarGenerator(self.on\_bar)

# K线序列管理工具，在本策略中用于求简单均线

self.am = ArrayManager()

# 以下实现基类中规定必须在策略中实现的回调函数

def on\_init(self):

"""

策略初始化时的回调函数

"""

self.write\_log("策略初始化")

# 在策略初始化时先加载10天的行情数据

self.load\_bar(10)

def on\_start(self):

"""

策略启动时的回调函数

"""

self.write\_log("策略启动")

self.put\_event()

def on\_stop(self):

"""

策略停止时的回调函数

"""

self.write\_log("策略停止")

self.put\_event()

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

"""

收到行情Tick推送的回调函数

"""

# 将Tick数据交给K线合成器，用于生成K线数据

self.bg.update\_tick(tick)

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

"""

收到新K线数据的回调函数

"""

# 将K线数据交给K线序列管理工具进行处理

am = self.am

am.update\_bar(bar)

if not am.inited:

# 如果am初始化未完成，即出，不进行交易

return

# 求快均线

fast\_ma = am.sma(self.fast\_window, array=True)

self.fast\_ma0 = fast\_ma[-1]

self.fast\_ma1 = fast\_ma[-2]

# 求慢均线

slow\_ma = am.sma(self.slow\_window, array=True)

self.slow\_ma0 = slow\_ma[-1]

self.slow\_ma1 = slow\_ma[-2]

# 是否金叉

cross\_over = self.fast\_ma0 > self.slow\_ma0 and self.fast\_ma1 < self.slow\_ma1

# 是否死叉

cross\_below = self.fast\_ma0 < self.slow\_ma0 and self.fast\_ma1 > self.slow\_ma1

if cross\_over:

# 如果金叉

if self.pos == 0:

# 如果当前仓位为0，买开

self.buy(bar.close\_price, 1)

elif self.pos < 0:

# 如果当前有空仓，先买平再买开

self.cover(bar.close\_price, 1)

self.buy(bar.close\_price, 1)

# 如果已经持有多仓，则什么都不做

elif cross\_below:

# 如果死叉

if self.pos == 0:

# 如果当前仓位为0，卖开

self.short(bar.close\_price, 1)

elif self.pos > 0:

# 如果当前有多仓，先卖平再卖开

self.sell(bar.close\_price, 1)

self.short(bar.close\_price, 1)

self.put\_event()

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

"""

收到委托变化推送的回调函数

"""

pass

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

"""

收到成交推送的回调函数

"""

self.put\_event()

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

"""

收到停止单推送的回调函数

"""

pass

策略的主要思想就是：金叉就买平买开，死叉就卖平卖开。

在策略中买卖都是1手，实际运行中应该有仓位管理。

## 其他一般CTA策略

不能指望找到一个万能策略，可以适应所有市场、所有时段和所有的品种，实际交易中可能需要根据市场情况在不同的策略间切换，同一个策略可能也需要不断调整参数。真正的量化交易，一定要对各种策略透彻了解。

vn.py的示例CTA策略看起来挺多，其实每个都各有特点，没有重复，都应该仔细阅读。

限于篇幅，本节只包含各示例策略的介绍，不再放带注释的源码。

### Dual Thrust策略

策略文件：dual\_thrust\_strategy.py

Dual Thrust是一个趋势跟踪系统，由迈克尔.查莱克（Michael Chalek）在20世纪80年代开发，曾被Furures Truth杂志评为最赚钱的策略之一。Dual Thrust系统具有简单易用、适用度广的特点，其思路简单、参数很少，配合不同的参数、止损止盈和仓位管理，可以为投资者带来长期稳定的收益，被投资者广泛应用于股票、货币、贵金属、债券、能源及股指期货市场等。在Dual Thrust策略中，对于震荡区间的定义非常关键，这也是该交易策略的核心和精髓。Dual Thrust策略采用：

Range = Max(HH - LC, HC - LL)

来描述震荡区间的大小。其中，HH是N日的最高价，LC是N日收盘的最低价，HC是N日收盘价的最高价，LL是N日最低价的最低价。

本系统取N值为1，即认为只有昨天股价的变化会对当天股价有影响。

构造系统：当价格向上突破上轨时，如果当时持有空仓，则先平仓再开多仓；如果没有仓位，则直接开多仓。当价格向下突破下轨时，如果当时持有多仓，则先平仓再开空仓；如果没有仓位，则直接开空仓。

Dual Thrust对于多头和空头的触发条件，考虑了非对称的幅度，即做多和做空参考的Range可以选择不同的周期数，也可以通过参数k1和k2来确定。

* 当k1 < k2时，多头相对容易被触发。
* 当k1 > k2时，空头相对容易被触发。

因此，在使用该策略时，一方面可以参考历史数据测试的最优参数；另一方面，则可以根据自己对后势的判断，或从其他大周期的技术指标入手，阶段性地动态调整k1和k2的值。

国内行情具有涨慢跌快的特点，所以一般设置k1 < k2。

增加额外的条件力求过滤震荡行情：

* 规定每天多开交易只做一次，避免在震荡行情中，日内不断开平仓操作损失手续费。
* 只当Tick行情推送合成的分钟K线高于日开盘价时判断交易方向为多头，设置突破的停止买入单，同理，分钟K线低于日开盘价时判断交易方向为空头，设置突破的停止卖出单。

止损：固定点位止损策略，如多头仓位在价格下跌到下轨时自己平仓，空头仓位在价格突破到上轨时自动平仓。

此策略大量使用了停止单，限制每天只交易一次。策略还可以优化，如基于5分钟判断。

### AtrRsi策略

策略文件：atr\_rsi\_strategy.py

ATR指标：平均真实波动范围(Average True Range)，简称ATR指标，是由威尔斯.韦尔德(J. Welles Wilder)发明的。ATR指标主要用来衡量市场波动的强烈度，即用来显示市场变化率的指标。注意，这一指标主要用来衡量价格的波动，并不能直接反映价格趋势及其趋势稳定性。

RSI指标：相对强弱指标(Rekatuve Strength Index)，简称RSI指标，也是由威尔斯.韦尔德发明的。RSI指标是根据市场上供求关系平稳的原理，通过比较一段时期内单个标的物或整个市场指数的涨跌幅度来分析判断市场上多空双方买卖力量的强弱程度，从而判断未来市场趋势的技术指标。

策略原理：ATR用于过滤，RSI用于产生交易信号，固定百分比点位移动止损。

该策略只用到两个指标。ATR用于过滤，当ATR>ATRMa时，显示市场波动性增大，趋势正在增强。只有在市场出现趋势的时候做单（追涨杀跌），盈利的机会才会增大。RSI用于产生交易信号，当RSI>规定上限时，开仓做多；反之，当RSI<规定下限时，开创做空。开仓之后就需要考虑如何盈利离场或止损离场，该策略用固定百分比点位移动止损。

此策略中用到了固定百分比点位移动止损，可以参考。

### 金肯特纳通道策略

策略文件：king\_keltner\_strategy.py

金肯特纳通道策略是一个典型的通道突破策略，即当价格突破通道上轨时做多，当价格超低突破通道下轨时做空。若价格在通道内上下走动，则不进行开仓操作。固定百分比点位移动止损。

轨道计算的思路也相对简单，先计算移动均线（MA），并且统计ATR指标，设置一定的通道宽度偏差X，如下：

* 上轨 = MA + X \* ATR
* 下轨 = MA - X \* ATR

因为ATR指标相对于标准差能捕捉到K线的跳空高开或者跳空低开的情况，所以更适合于一些在短期内有较大波动的品种，如股指期货或者有“小股指”之称的螺纹钢。

OCO委托的全称是“One Cancels the Other Order”，意思是二选一委托，即在K线内同时发出条件买单和条件卖单：若价格突破上轨，则触发条件买单同时取消条件卖单；若价格突破下轨，则触发条件卖单同时取消条件买单。这种挂单方式在国内交易所比较少见，一般多用于外汇市场。因为货币在短时间内会有很强的震荡，比较难以判断趋势，这时候OCO的优点就显现出来了。当盘整震荡的行情接近结束，而要进入一个上涨或下跌的趋势时，可用OCO挂单捕捉趋势。一个例子是发生重大行情如非农或者利率决议公布，若不确定接下来的行情，则可用OCO挂单。

OCO委托实现方法如下：

创建3个空的列表long\_vt\_orderids、short\_vt\_orderids和vt\_orderids。long\_vt\_orderids用于缓存条件买入单的委托，分别插入委托价格、合约手数。short\_vt\_orderids用于缓存条件卖出单的委托，分别插入委托价格、合约手数。vt\_orderids用于缓存所有发出的委托单，用extend()方法把上面两个列表添加进来。

在on\_5min\_bar函数中：为保证委托的唯一性，同样要撤销之前尚未成交的委托。但这一次没有采用cancelAll方法，而是先在vt\_orderids中遍历，然后删除，以保证清空vt\_orderids的目标。

本策略的特点：使用了5分钟K线，使用了OCO委托，对on\_trade事件进行了处理。

# 目标持仓策略

一般的CTA策略在on\_tick或on\_bar中进行策略计算，确定买卖方向及数量。

vn.py还支持一种策略形式，称为目标持仓策略。当Tick数据或K线数据到来时，各信号对象单独进行计算，算出该对象建议的目标仓位。而目标持仓策略综合各信号对象的计算结果，执行买卖操作。这样可能会给复杂策略的编写带来方便，如跨时间周期交易等。

## CTA信号类CtaSignal

目标持仓策略综合处理各种信号，因此需要首先分析CTA信号对象。

CtaSignal类与CtaTemplate类在同一文件中定义，在D:\vnpy220\vnpy\app\cta\_strategy目录下的template.py文件中。

class CtaSignal(ABC):

"""CTA策略信号，负责纯粹的信号生成（目标仓位），不参与具体交易管理"""

def \_\_init\_\_(self):

"""构造函数"""

self.signal\_pos = 0

@virtual

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

"""

收到新Tick数据的回调函数

"""

pass

@virtual

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

"""

收到新K线数据的回调函数

"""

pass

def set\_signal\_pos(self, pos):

"""设置信号仓位"""

self.signal\_pos = pos

def get\_signal\_pos(self):

"""获取信号仓位"""

return self.signal\_pos

信号类看起来就是一个小策略，只不过策略的结果不是买卖，而是设置信号仓位。

## 目标持仓模板类TargetPosTemplate

目标持仓策略模板类（TargetPosTemplate类）继承自CTA策略的模板类CtaTemplate，与CtaTemplate类在同一文件中定义，在D:\vnpy220\vnpy\app\cta\_strategy目录下的template.py文件中。

class TargetPosTemplate(CtaTemplate):

"""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"""

tick\_add = 1 # 委托时相对基准价格的超价

last\_tick = None # 最新tick数据

last\_bar = None # 最新bar数据

target\_pos = 0 # 目标持仓

def \_\_init\_\_(self, cta\_engine, strategy\_name, vt\_symbol, setting):

"""构造函数"""

super().\_\_init\_\_(cta\_engine, strategy\_name, vt\_symbol, setting)

self.active\_orderids = [] # 活动委托单号列表

self.cancel\_orderids = [] # 撤销委托单号列表

# 策略变量中增加target\_pos

self.variables.append("target\_pos")

@virtual

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

"""

收到新Tick数据的回调函数

"""

self.last\_tick = tick

# 实盘模式下，启动交易后，需要根据tick的实时推送执行自动开平仓操作

if self.trading:

self.trade()

@virtual

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

"""

收到新K线数据的回调函数

"""

self.last\_bar = bar

@virtual

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

"""

收到委托推送回调函数

"""

# 委托单号

vt\_orderid = order.vt\_orderid

# 如果该委托不再活动

if not order.is\_active():

# 从两个委托单号列表中移除

if vt\_orderid in self.active\_orderids:

self.active\_orderids.remove(vt\_orderid)

if vt\_orderid in self.cancel\_orderids:

self.cancel\_orderids.remove(vt\_orderid)

def check\_order\_finished(self):

"""检查委托单是否全部完成"""

if self.active\_orderids:

return False # 还有活动的委托，就是未完成

else:

return True

def set\_target\_pos(self, target\_pos):

"""设置目标仓位"""

self.target\_pos = target\_pos

self.trade()

def trade(self):

"""执行交易"""

if not self.check\_order\_finished():

# 如果前面的委托未全部完成，撤消所有老的委托

self.cancel\_old\_order()

else:

# 如果前面的委托已全部完成，发送新委托

self.send\_new\_order()

def cancel\_old\_order(self):

"""撤消所有老的委托"""

# 如果目标仓位和实际仓位一致，则不进行任何操作

for vt\_orderid in self.active\_orderids:

if vt\_orderid not in self.cancel\_orderids:

self.cancel\_order(vt\_orderid)

self.cancel\_orderids.append(vt\_orderid)

def send\_new\_order(self):

"""发送新委托"""

pos\_change = self.target\_pos - self.pos

if not pos\_change:

return

# 确定委托基准价格，有Tick数据时优先使用，否则使用K线数据

long\_price = 0

short\_price = 0

if self.last\_tick:

if pos\_change > 0:

long\_price = self.last\_tick.ask\_price\_1 + self.tick\_add

if self.last\_tick.limit\_up:

# 涨停价检查

long\_price = min(long\_price, self.last\_tick.limit\_up)

else:

short\_price = self.last\_tick.bid\_price\_1 - self.tick\_add

if self.last\_tick.limit\_down:

# 跌停价检查

short\_price = max(short\_price, self.last\_tick.limit\_down)

else:

if pos\_change > 0:

long\_price = self.last\_bar.close\_price + self.tick\_add

else:

short\_price = self.last\_bar.close\_price - self.tick\_add

# 回测模式下，采用合并平仓和反向开仓委托的方式

if self.get\_engine\_type() == EngineType.BACKTESTING:

if pos\_change > 0:

vt\_orderids = self.buy(long\_price, abs(pos\_change))

else:

vt\_orderids = self.short(short\_price, abs(pos\_change))

self.active\_orderids.extend(vt\_orderids)

# 实盘模式下，首先确保之前的委托都已经结束（全成、撤销）

# 然后先发平仓委托，等待成交后，再发送新的开仓委托

else:

# 检查之前委托都已结束

if self.active\_orderids:

return

# 买入

if pos\_change > 0:

# 若当前有空头持仓

if self.pos < 0:

if pos\_change < abs(self.pos):

# 若买入量小于空头持仓，则直接平空买入量

vt\_orderids = self.cover(long\_price, pos\_change)

else:

# 否则先平所有的空头仓位

vt\_orderids = self.cover(long\_price, abs(self.pos))

else:

# 若没有空头持仓，则执行开仓操作

vt\_orderids = self.buy(long\_price, abs(pos\_change))

# 卖出和以上相反

else:

if self.pos > 0:

if abs(pos\_change) < self.pos:

vt\_orderids = self.sell(short\_price, abs(pos\_change))

else:

vt\_orderids = self.sell(short\_price, abs(self.pos))

else:

vt\_orderids = self.short(short\_price, abs(pos\_change))

self.active\_orderids.extend(vt\_orderids)

可以看出，买卖操作集中在模板类的set\_target\_pos函数中，其好处是允许直接通过修改目标持仓来实现交易的策略模板。用户在开发策略时，无需再调用buy、sell、cover和short这些具体的委托指令，只需在策略逻辑运行完成后调用set\_target\_pos设置目标持仓，CTA策略模板类（父类）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。

## 目标持仓策略示例MultiSignalStrategy

本节以另一个系统自带的策略MultiSignalStrategy为例，这是一个目标持仓策略。打开D:\vnpy220\vnpy\app\cta\_strategy\strategies目录下的文件multi\_signal\_strategy.py，代码如下。

"""

一个多信号组合策略，基于的信号包括：RSI（1分钟）、CCI（1分钟）和MA（5分钟）

"""

from vnpy.app.cta\_strategy import (

StopOrder,

TickData,

BarData,

TradeData,

OrderData,

BarGenerator,

ArrayManager,

CtaSignal,

TargetPosTemplate

)

class RsiSignal(CtaSignal):

"""

RSI（1分钟）：大于rsi\_long为多头、低于rsi\_short为空头

"""

def \_\_init\_\_(self, rsi\_window: int, rsi\_level: float):

"""构造函数"""

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

self.rsi\_window = rsi\_window

self.rsi\_level = rsi\_level

self.rsi\_long = 50 + self.rsi\_level

self.rsi\_short = 50 - self.rsi\_level

self.bg = BarGenerator(self.on\_bar)

self.am = ArrayManager()

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

"""

收到新Tick数据的回调函数

"""

self.bg.update\_tick(tick)

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

"""

收到新K线数据的回调函数

"""

self.am.update\_bar(bar)

if not self.am.inited:

self.set\_signal\_pos(0)

rsi\_value = self.am.rsi(self.rsi\_window)

if rsi\_value >= self.rsi\_long:

self.set\_signal\_pos(1)

elif rsi\_value <= self.rsi\_short:

self.set\_signal\_pos(-1)

else:

self.set\_signal\_pos(0)

class CciSignal(CtaSignal):

"""

CCI（1分钟）：大于cci\_long为多头、低于cci\_short为空头

"""

def \_\_init\_\_(self, cci\_window: int, cci\_level: float):

"""构造函数"""

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

self.cci\_window = cci\_window

self.cci\_level = cci\_level

self.cci\_long = self.cci\_level

self.cci\_short = -self.cci\_level

self.bg = BarGenerator(self.on\_bar)

self.am = ArrayManager()

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

"""

收到新Tick数据的回调函数

"""

self.bg.update\_tick(tick)

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

"""

收到新K线数据的回调函数

"""

self.am.update\_bar(bar)

if not self.am.inited:

self.set\_signal\_pos(0)

cci\_value = self.am.cci(self.cci\_window)

if cci\_value >= self.cci\_long:

self.set\_signal\_pos(1)

elif cci\_value <= self.cci\_short:

self.set\_signal\_pos(-1)

else:

self.set\_signal\_pos(0)

class MaSignal(CtaSignal):

"""

MA（5分钟）：快速大于慢速为多头、低于慢速为空头

"""

def \_\_init\_\_(self, fast\_window: int, slow\_window: int):

"""构造函数"""

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

self.fast\_window = fast\_window

self.slow\_window = slow\_window

self.bg = BarGenerator(self.on\_bar, 5, self.on\_5min\_bar)

self.am = ArrayManager()

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

"""

收到新Tick数据的回调函数

"""

self.bg.update\_tick(tick)

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

"""

收到新K线数据的回调函数

"""

self.bg.update\_bar(bar)

def on\_5min\_bar(self, bar: BarData):

"""5分钟K线更新"""

self.am.update\_bar(bar)

if not self.am.inited:

self.set\_signal\_pos(0)

fast\_ma = self.am.sma(self.fast\_window)

slow\_ma = self.am.sma(self.slow\_window)

if fast\_ma > slow\_ma:

self.set\_signal\_pos(1)

elif fast\_ma < slow\_ma:

self.set\_signal\_pos(-1)

else:

self.set\_signal\_pos(0)

class MultiSignalStrategy(TargetPosTemplate):

"""跨时间周期交易策略"""

author = "用Python的交易员"

# 策略参数

rsi\_window = 14

rsi\_level = 20

cci\_window = 30

cci\_level = 10

fast\_window = 5

slow\_window = 20

# 策略变量

signal\_pos = {} # 信号仓位

parameters = ["rsi\_window", "rsi\_level", "cci\_window",

"cci\_level", "fast\_window", "slow\_window"]

variables = ["signal\_pos", "target\_pos"]

def \_\_init\_\_(self, cta\_engine, strategy\_name, vt\_symbol, setting):

"""构造函数"""

super().\_\_init\_\_(cta\_engine, strategy\_name, vt\_symbol, setting)

self.rsi\_signal = RsiSignal(self.rsi\_window, self.rsi\_level)

self.cci\_signal = CciSignal(self.cci\_window, self.cci\_level)

self.ma\_signal = MaSignal(self.fast\_window, self.slow\_window)

self.signal\_pos = {

"rsi": 0,

"cci": 0,

"ma": 0

}

def on\_init(self):

"""

策略初始化时的回调函数

"""

self.write\_log("策略初始化")

self.load\_bar(10)

def on\_start(self):

"""

策略启动时的回调函数

"""

self.write\_log("策略启动")

def on\_stop(self):

"""

策略停止时的回调函数

"""

self.write\_log("策略停止")

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

"""

收到行情Tick推送的回调函数

"""

super(MultiSignalStrategy, self).on\_tick(tick)

self.rsi\_signal.on\_tick(tick)

self.cci\_signal.on\_tick(tick)

self.ma\_signal.on\_tick(tick)

self.calculate\_target\_pos()

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

"""

收到新K线数据的回调函数

"""

super(MultiSignalStrategy, self).on\_bar(bar)

self.rsi\_signal.on\_bar(bar)

self.cci\_signal.on\_bar(bar)

self.ma\_signal.on\_bar(bar)

self.calculate\_target\_pos()

def calculate\_target\_pos(self):

"""计算目标仓位"""

self.signal\_pos["rsi"] = self.rsi\_signal.get\_signal\_pos()

self.signal\_pos["cci"] = self.cci\_signal.get\_signal\_pos()

self.signal\_pos["ma"] = self.ma\_signal.get\_signal\_pos()

target\_pos = 0

for v in self.signal\_pos.values():

target\_pos += v

# 设置目标仓位，在模板类中会引发执行交易

self.set\_target\_pos(target\_pos)

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

"""

收到委托变化推送的回调函数

"""

super(MultiSignalStrategy, self).on\_order(order)

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

"""

收到成交推送的回调函数

"""

self.put\_event()

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

"""

收到停止单推送的回调函数

"""

pass

先定义三个信号，策略根据这三个信号，综合确定目标仓位。策略中不用执行具体的买卖，只需要在确定了目标仓位后调用set\_target\_pos函数即可。注意，本策略并没有重写set\_target\_pos函数，执行的是模板类（基类）中的该函数。

从此示例可以看出，CTA信号与目标持仓策略配合得比较好。但并不是只有它们两个才能一起配合。一般CTA策略中也可以使用CTA信号，只是需要策略自己来执行买卖操作罢了。

另外，除了信号之外，策略本身的逻辑也可以发出买卖信号，可以与CTA信号相结合，CTA信号和策略不是割裂的。

# 回测的执行过程

以简单策略DoubleMaStrategy为例，详细分析回测的执行过程。一般的分析参第一部分“CTA回测”一章。

本章从回测执行引擎类BacktestingEngine开始分析。

## 回测执行引擎的初始化

回测执行引擎类BacktestingEngine在D:\vnpy220\vnpy\app\cta\_strategy下的backtesting.py中定义，初始化部分代码如下：

class BacktestingEngine:

"""回测执行引擎类"""

engine\_type = EngineType.BACKTESTING # 引擎类型为回测

gateway\_name = "BACKTESTING"

def \_\_init\_\_(self):

"""初始化"""

self.vt\_symbol = "" # 本地代码

self.symbol = "" # 合约代码

self.exchange = None # 交易所

self.start = None # 回测开始时间

self.end = None # 回测结束时间

self.rate = 0 # 手续费率

self.slippage = 0 # 交易滑点

self.size = 1 # 合约乘数

self.pricetick = 0 # 价格跳动

self.capital = 1\_000\_000 # 回测资金

self.mode = BacktestingMode.BAR # 回测模式：基于K线

self.inverse = False # 合约模式？

self.strategy\_class = None # 策略类

self.strategy = None # 策略

self.tick: TickData # Tick数据

self.bar: BarData # K线数据

self.datetime = None # 当前（正在处理的K线）的日期时间

self.interval = None # K线周期

self.days = 0 # 初始化策略需要数据的天数

self.callback = None # 加载数据时的回调函数

# 注：回测时此回调函数不能为None。策略在调用load\_bar()或load\_tick()时，

# 必然要指定回调函数，用于将数据推送回策略

self.history\_data = [] # 历史行情数据列表

# 注：回测使用行情数据的方法是一次性地将所需数据加载到回测执行引擎中来

self.stop\_order\_count = 0 # 停止单数量

self.stop\_orders = {} # 停止单字典

self.active\_stop\_orders = {} # 活动停止单字典

self.limit\_order\_count = 0 # 限价单数量

self.limit\_orders = {} # 限价单字典

self.active\_limit\_orders = {} # 活动限价单字典

self.trade\_count = 0 # 成交数量

self.trades = {} # 成交字典

self.logs = [] # 日志列表

self.daily\_results = {} # 逐日盯市盈亏：字典

self.daily\_df = None # 逐日盯市盈亏：DataFrame，以Date为索引

## 回测执行函数

教材第18章已经分析了回测的执行流程，指出回测执行引擎类的成员函数run\_backtesting()执行真正的回测操作。回测执行引擎类的run\_backtesting()函数代码如下：

def run\_backtesting(self):

"""回测执行"""

# 指定策略函数

# 根据回测模式，确定新数据的处理函数

if self.mode == BacktestingMode.BAR:

func = self.new\_bar

else:

func = self.new\_tick

# 初始化策略

self.strategy.on\_init()

# 在历史行情数据中取前days天的数据，用于初始化策略。这些数据不产生交易信号

# 注：执行此方法前，已经调用本类的load\_data()方法，

# 将行情数据加载到了self.history\_data中

day\_count = 1 # 天数计数

ix = 0 # 数据量计数

for ix, data in enumerate(self.history\_data):

# 如果是新的一天，天数计数+1

if self.datetime and data.datetime.day != self.datetime.day:

day\_count += 1

if day\_count >= self.days:

break

# 保存当前日期时间

self.datetime = data.datetime

try:

# 调用回调函数推送数据

self.callback(data)

except Exception:

self.output("触发异常，回测终止")

self.output(traceback.format\_exc())

return

# days天的数据回话完成，策略初始化完成

self.strategy.inited = True

self.output("策略初始化完成")

# 执行策略启动时的操作

self.strategy.on\_start()

# 设置允许策略进行交易的标志

self.strategy.trading = True

self.output("开始回放历史数据")

# 用剩余的历史行情数据进行回测

backtesting\_data = self.history\_data[ix + 1:]

if not backtesting\_data:

self.output("历史数据不足，回测终止")

return

# 注：新版本的vn.py回测增加了进度显示的功能，

# 每回放10%的数据，在界面上显示一条信息。

# 回放数据总量

total\_size = len(backtesting\_data)

# 每一批的数据量

batch\_size = max(int(total\_size / 10), 1)

for ix, i in enumerate(range(0, total\_size, batch\_size)):

# 取一批数据

batch\_data = backtesting\_data[i: i + batch\_size]

# 回放每一条数据

for data in batch\_data:

try:

# 执行func策略函数处理数据

func(data)

except Exception:

self.output("触发异常，回测终止")

self.output(traceback.format\_exc())

return

# 处理完一批数据后显示进度

progress = min(ix / 10, 1)

progress\_bar = "=" \* (ix + 1)

self.output(f"回放进度：{progress\_bar} [{progress:.0%}]")

# 执行策略停止时的操作

self.strategy.on\_stop()

self.output("历史数据回放结束")

可以看出，回测执行被分为两个阶段。第一阶段是初始化。在这个阶段，反复调用callback函数，将days天的行情数据加载到策略中。callback函数由策略指定，通常是指向策略的on\_bar函数或on\_tick函数，在忽略了交易信号后只是将数据加载到策略。

第二阶段是真正的回测，其中策略函数函数func明确地指向了本类的new\_bar或者new\_tick函数，这两个函数的代码如下：

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

"""策略函数：处理一个K线数据"""

self.bar = bar

# 保存当前日期时间

self.datetime = bar.datetime

# 撮合限价单

self.cross\_limit\_order()

# 撮合停止单

self.cross\_stop\_order()

# 执行策略的on\_bar()函数

self.strategy.on\_bar(bar)

# 更新每日盯盘的收盘数据

self.update\_daily\_close(bar.close\_price)

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

"""策略函数：处理一个Tick数据"""

self.tick = tick

self.datetime = tick.datetime

self.cross\_limit\_order()

self.cross\_stop\_order()

self.strategy.on\_tick(tick)

self.update\_daily\_close(tick.last\_price)

在上面的分析中提到了两个重要函数，回调函数callback和策略函数func。其中func明确地指向了本类的new\_bar或者new\_tick，而callback在本文件中搞不清是指的哪个函数，要在多个文件中进行分析。下面先对callback指向哪个函数进行分析。

关于撮合限价委托单和撮合本地停止单(条件单)：根据最新的行情K线或者TICK，对策略之前下达的所有委托进行检查，如果能够撮合成交，则返回并记录数据。

## 回调函数

当有新数据时如何处理呢，也就是调用哪个函数？因为策略要同时服务于回测和实盘，所以有些东西就会显得很绕。

以回测为例，以K线数据为例，需要在多处寻找相关的代码段。

### 在回测引擎类中

回测引擎类在D:\vnpy220\vnpy\app\cta\_backtester下的engine.py中定义。其中回测线程函数完整的源代码在本文档第4章已经给出，本节只给出需要重点说明的部分，相关代码如下：

class BacktesterEngine(BaseEngine):

"""回测引擎类"""

def init\_engine(self):

"""初始化回测引擎"""

# 创建回测执行引擎

self.backtesting\_engine = BacktestingEngine()

def run\_backtesting(

……

):

"""回测线程函数"""

# 指定回测执行引擎

engine = self.backtesting\_engine

# 根据策略名称取策略类

strategy\_class = self.classes[class\_name]

# 将策略类和策略参数加载到回测引擎

engine.add\_strategy(

strategy\_class,

setting

)

# 加载历史行情数据

engine.load\_data()

try:

# 调用回测执行引擎的run\_backtesting方法执行回测任务

# 包括策略初始化及回放历史行情数据等

engine.run\_backtesting()

except Exception:

……

可以看到，在回测引擎类中先创建回测执行引擎。在回测线程函数中，根据策略名称取策略类。策略名称来自于回测界面左上的“交易策略”列表，如果回测的是DoubleMaStrategy策略，则策略类为<class 'vnpy.app.cta\_strategy.strategies.double\_ma\_strategy.DoubleMaStrategy'>。

调用回测执行引擎的add\_strategy()函数加载策略，再调用回测执行引擎的load\_data()函数加载历史数据，最后调用回测执行引擎的run\_backtesting()函数执行真正的回测操作。注意，所有需要的历史行情数据已经在调用load\_data()函数时一次性地加载，存到了回测执行引擎的history\_data中，执行run\_backtesting()函数的过程中已经不再需要加载数据。

回测执行引擎的load\_data()函数的代码已经在本文档“3.5. 使用数据”一节详细分析，读者可以回顾一下，这样就可以把不同的内容关联到一起。

### 在回测执行引擎类中

回测执行引擎类在D:\vnpy220\vnpy\app\cta\_strategy下的backtesting.py文件中定义，相关代码如下：

class BacktestingEngine:

"""回测执行引擎类"""

def \_\_init\_\_(self):

self.callback = None # 回调函数

def add\_strategy(self, strategy\_class: type, setting: dict):

"""加载策略"""

self.strategy\_class = strategy\_class

self.strategy = strategy\_class(

self, strategy\_class.\_\_name\_\_, self.vt\_symbol, setting

)

在加载策略时，会将本回测执行类实例的指针作为参数传给策略类。

### 在策略模板类中

策略模板类在D:\vnpy220\vnpy\app\cta\_strategy目录下的template.py文件中定义，相关代码如下：

class CtaTemplate(ABC):

"""CTA策略模板"""

def \_\_init\_\_(

self,

cta\_engine: Any,

strategy\_name: str,

vt\_symbol: str,

setting: dict,

):

self.cta\_engine = cta\_engine # CTA引擎

def load\_bar(

self,

days: int,

interval: Interval = Interval.MINUTE,

callback: Callable = None,

use\_database: bool = False

):

"""

在初始化策略时加载历史Bar数据

"""

if not callback:

callback = self.on\_bar

self.cta\_engine.load\_bar(

self.vt\_symbol,

days,

interval,

callback,

use\_database

)

在策略模板类中，cta\_engine参数接收到的是回测执行引擎类实例的指针。

当load\_bar函数被调用时，会将本类的on\_bar函数传给回测执行引擎类，作为回测执行引擎类的回调函数。

至此，我们就知道了，回测执行引擎类的回调函数其实是策略类的on\_bar函数。这种机制虽然绕，但可以保证无论是回测还是实盘，策略都可以通过on\_bar函数得到K线数据。当然，如果策略模式是基于Tick数据，那回调函数就是on\_tick。

那么，load\_bar函数何时被调用呢？

### 在策略类中

在具体策略类的on\_init函数中，一般都会调用load\_bar函数，取一定天数的行情数据，用于策略初始化。如在DoubleMaStrategy策略中：

class DoubleMaStrategy(CtaTemplate):

author = "用Python的交易员"

def on\_init(self):

"""

策略初始化时的回调函数

"""

self.write\_log("策略初始化")

# 在策略初始化时先加载10天的行情数据

self.load\_bar(10) # 取10天的K线数据

调用的是本策略类的load\_bar函数（该函数在策略模板类中定义），但在load\_bar函数中会调用回测执行引擎类中的load\_bar函数，并将本类的on\_bar函数作为参数传递。详细代码见“12.1. CTA策略模板CtaTemplate”一节。

### 在回测执行引擎类中（二）

回测执行引擎类中的load\_bar函数代码如下：

def load\_bar(

self,

vt\_symbol: str,

days: int,

interval: Interval,

callback: Callable,

use\_database: bool

):

"""加载K线数据"""

self.days = days

self.callback = callback

以后，回测执行引擎类中调用callback函数时，调用的其实是策略类的on\_bar函数。另外还可以看出，由于所有数据的加载都在load\_data()中统一进行，此处只需要设置self.days和self.callback即可，并不需要真正地加载数据。

## 回测执行

分析回测的执行过程，如限价单的处理等。

略。（有了前面的内容，应该一看就懂）

第四部分 CTA策略深入分析

研究vn.py终究是为了实盘，vn.py的实盘由“CTA策略”上层应用完成。

本文第三部分分析CTA回测，采用的仍然是自顶向下的分析方法，但比前两部分深入，已经接触到了一些类的源代码。本部分采用自底向上的分析方法，从基础类开始，分析到程序界面。

本部分目标：从实盘的角度，搞清CTA策略的执行机制。

# 一些基础类

分析深入到这个程度时，已经容不得任何含糊其辞。每个函数，每个函数的每个参数都要搞清楚，对系统要达到通晓的程度，将来才可能改写系统。本章分析一些基础类，回测和实盘都以这些类为基础。本章内容既补前面的缺（在分析回测时没有深入到这些类），也为下面CTA策略（实盘）的分析做准备。

## 在constant.py中定义常量

在D:\vnpy220\vnpy\trader目录下的constant.py文件中，定义了VN Trader中用到的字符串常量。代码如下：

"""

VN Trader中用到的字符串常量

"""

from enum import Enum

class Direction(Enum):

"""

方向，用于委托/成交/仓位

"""

LONG = "多"

SHORT = "空"

NET = "净"

class Offset(Enum):

"""

开平，用于委托/成交

"""

NONE = ""

OPEN = "开"

CLOSE = "平"

CLOSETODAY = "平今"

CLOSEYESTERDAY = "平昨"

class Status(Enum):

"""

委托单状态

"""

SUBMITTING = "提交中"

NOTTRADED = "未成交"

PARTTRADED = "部分成交"

ALLTRADED = "全部成交"

CANCELLED = "已撤销"

REJECTED = "拒单"

class Product(Enum):

"""

产品类型

"""

EQUITY = "股票"

FUTURES = "期货"

OPTION = "期权"

INDEX = "指数"

FOREX = "外汇"

SPOT = "现货"

ETF = "ETF"

BOND = "债券"

WARRANT = "权证"

SPREAD = "价差"

FUND = "基金"

class OrderType(Enum):

"""

委托单类型

"""

LIMIT = "限价"

MARKET = "市价"

STOP = "STOP"

FAK = "FAK"

FOK = "FOK"

RFQ = "询价"

class OptionType(Enum):

"""

期权类型

"""

CALL = "看涨期权"

PUT = "看跌期权"

class Exchange(Enum):

"""

交易所

"""

# 中国的

CFFEX = "CFFEX" # China Financial Futures Exchange

SHFE = "SHFE" # Shanghai Futures Exchange

CZCE = "CZCE" # Zhengzhou Commodity Exchange

DCE = "DCE" # Dalian Commodity Exchange

INE = "INE" # Shanghai International Energy Exchange

SSE = "SSE" # Shanghai Stock Exchange

SZSE = "SZSE" # Shenzhen Stock Exchange

SGE = "SGE" # Shanghai Gold Exchange

WXE = "WXE" # Wuxi Steel Exchange

CFETS = "CFETS" # China Foreign Exchange Trade System

# 全球的

SMART = "SMART" # Smart Router for US stocks

NYSE = "NYSE" # New York Stock Exchnage

NASDAQ = "NASDAQ" # Nasdaq Exchange

ARCA = "ARCA" # ARCA Exchange

EDGEA = "EDGEA" # Direct Edge Exchange

ISLAND = "ISLAND" # Nasdaq Island ECN

BATS = "BATS" # Bats Global Markets

IEX = "IEX" # The Investors Exchange

NYMEX = "NYMEX" # New York Mercantile Exchange

COMEX = "COMEX" # COMEX of CME

GLOBEX = "GLOBEX" # Globex of CME

IDEALPRO = "IDEALPRO" # Forex ECN of Interactive Brokers

CME = "CME" # Chicago Mercantile Exchange

ICE = "ICE" # Intercontinental Exchange

SEHK = "SEHK" # Stock Exchange of Hong Kong

HKFE = "HKFE" # Hong Kong Futures Exchange

HKSE = "HKSE" # Hong Kong Stock Exchange

SGX = "SGX" # Singapore Global Exchange

CBOT = "CBT" # Chicago Board of Trade

CBOE = "CBOE" # Chicago Board Options Exchange

CFE = "CFE" # CBOE Futures Exchange

DME = "DME" # Dubai Mercantile Exchange

EUREX = "EUX" # Eurex Exchange

APEX = "APEX" # Asia Pacific Exchange

LME = "LME" # London Metal Exchange

BMD = "BMD" # Bursa Malaysia Derivatives

TOCOM = "TOCOM" # Tokyo Commodity Exchange

EUNX = "EUNX" # Euronext Exchange

KRX = "KRX" # Korean Exchange

OTC = "OTC" # OTC Forex Broker

IBKRATS = "IBKRATS" # Paper Trading Exchange of IB

# 加密货币

BITMEX = "BITMEX"

OKEX = "OKEX"

HUOBI = "HUOBI"

BITFINEX = "BITFINEX"

BINANCE = "BINANCE"

BYBIT = "BYBIT" # bybit.com

COINBASE = "COINBASE"

DERIBIT = "DERIBIT"

GATEIO = "GATEIO"

BITSTAMP = "BITSTAMP"

# 特殊功能

LOCAL = "LOCAL" # For local generated data

class Currency(Enum):

"""

货币

"""

USD = "USD"

HKD = "HKD"

CNY = "CNY"

class Interval(Enum):

"""

K线周期

"""

MINUTE = "1m"

HOUR = "1h"

DAILY = "d"

WEEKLY = "w"

### 关于委托单类型

什么是限价单？举个例子，以做多为例，一般我们是想要价格跌到多少的时候就买进，比如Long170表示如果价格低于170就买入，这就是限价单。

什么是停止单？如果想要价格涨到多少的时候就买进，比如Long170表示如果价格高于170就买入，这就是停止单。停止单也称条件单，英文是Stop Order，所以本系统称停止单。其实英文原意是止损单，就是跌破多少时就卖出。这样看来，叫条件单其实更合适，对买入和卖出都合适。

vn.py支持停止单，但并不是所有的交易接口（服务器）都支持。如果服务器支持，停止单就发给服务器；如果服务器不支持，就保存在本地。

在D:\vnpy220\vnpy\app\cta\_strategy目录下的base.py文件中定义了用于回测和实盘的一些常量字符串，还定义了停止单的数据结构。代码如下：

@dataclass

class StopOrder:

vt\_symbol: str

direction: Direction

offset: Offset

price: float

volume: float

stop\_orderid: str

strategy\_name: str

lock: bool = False

vt\_orderids: list = field(default\_factory=list)

status: StopOrderStatus = StopOrderStatus.WAITING

### 关于Long 和Short

需要注意的是，并不是说Long就代表买入。如果持有空仓，平仓是Long，多头建仓也是Long。如果持有多仓，平仓是short，空头建仓也是short。下面是来自网络的一张图，比较说明问题。



## 在object.py中定义数据结构

在D:\vnpy220\vnpy\trader目录下的object.py文件中，定义了VN Trader中用于交易功能的基础数据结构。代码如下：

"""

VN Trader中用于交易功能的基础数据结构

"""

from dataclasses import dataclass

from datetime import datetime

from logging import INFO

from .constant import Direction, Exchange, Interval, Offset, Status, Product, OptionType, OrderType

ACTIVE\_STATUSES = set([Status.SUBMITTING, Status.NOTTRADED, Status.PARTTRADED])

@dataclass

class BaseData:

"""

回调函数推送数据的基础类，其他数据类继承于此

任何数据对象都需要一个接口名（gateway\_name）

"""

gateway\_name: str

@dataclass

class TickData(BaseData):

"""

Tick行情数据类，包含以下信息：

\* 市场上的最后成交

\* 委托单快照

\* 盘中市场统计

"""

# 代码相关

symbol: str # 合约代码

exchange: Exchange # 交易所代码

datetime: datetime # 日期时间

name: str = ""

volume: float = 0 # 今天总成交量

open\_interest: float = 0 # 持仓量

last\_price: float = 0 # 最新成交价

last\_volume: float = 0 # 最新成交量

limit\_up: float = 0 # 涨停价格

limit\_down: float = 0 # 跌停价格

# 常规行情

open\_price: float = 0 # 今日开盘价

high\_price: float = 0 # 今日最高价

low\_price: float = 0 # 今日最低价

pre\_close: float = 0 # 今日收盘价

# 十档行情

bid\_price\_1: float = 0

bid\_price\_2: float = 0

bid\_price\_3: float = 0

bid\_price\_4: float = 0

bid\_price\_5: float = 0

ask\_price\_1: float = 0

ask\_price\_2: float = 0

ask\_price\_3: float = 0

ask\_price\_4: float = 0

ask\_price\_5: float = 0

bid\_volume\_1: float = 0

bid\_volume\_2: float = 0

bid\_volume\_3: float = 0

bid\_volume\_4: float = 0

bid\_volume\_5: float = 0

ask\_volume\_1: float = 0

ask\_volume\_2: float = 0

ask\_volume\_3: float = 0

ask\_volume\_4: float = 0

ask\_volume\_5: float = 0

def \_\_post\_init\_\_(self):

""""""

# 本地代码：合约在vt系统中的唯一代码，通常是 合约代码.交易所代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

@dataclass

class BarData(BaseData):

"""

K线数据

"""

# 代码相关

symbol: str # 合约代码

exchange: Exchange # 交易所代码

datetime: datetime # 日期时间

interval: Interval = None # K线周期

volume: float = 0 # 成交量

open\_interest: float = 0 # 持仓量

open\_price: float = 0 # 开盘价

high\_price: float = 0 # 最高价

low\_price: float = 0 # 最低价

close\_price: float = 0 # 收盘价

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

@dataclass

class OrderData(BaseData):

"""

委托单数据类

包含某个委托单最新的状态

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

orderid: str # 委托单编号 gateway内部自己生成的编号

type: OrderType = OrderType.LIMIT # 委托类型

direction: Direction = None # 报单方向

offset: Offset = Offset.NONE # 报单开平仓

price: float = 0 # 报单价格

volume: float = 0 # 报单总数量

traded: float = 0 # 报单成交数量

status: Status = Status.SUBMITTING # 报单状态：提交中

datetime: datetime = None # 发单时间

reference: str = ""

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

# 委托单在vt系统中的唯一编号，通常是 Gateway名.委托单编号

self.vt\_orderid = f"{self.gateway\_name}.{self.orderid}"

def is\_active(self) -> bool:

"""

返回委托单是否活动

"""

if self.status in ACTIVE\_STATUSES:

return True

else:

return False

def create\_cancel\_request(self) -> "CancelRequest":

"""

根据委托单信息创建撤单申请

"""

req = CancelRequest(

orderid=self.orderid, symbol=self.symbol, exchange=self.exchange

)

return req

@dataclass

class TradeData(BaseData):

"""

成交数据类

一般来说，一个VtOrderData可能对应多个VtTradeData：一个委托单可能多次部分成交

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

orderid: str # 委托单编号 gateway内部自己生成的编号

tradeid: str # 成交编号 gateway内部自己生成的编号

direction: Direction = None # 成交方向

offset: Offset = Offset.NONE # 成交开平仓

price: float = 0 # 成交价格

volume: float = 0 # 成交数量

datetime: datetime = None # 成交时间

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

# 委托单在vt系统中的唯一编号，通常是 Gateway名.委托单编号

self.vt\_orderid = f"{self.gateway\_name}.{self.orderid}"

# 成交在vt系统中的唯一编号，通常是 Gateway名.成交编号

self.vt\_tradeid = f"{self.gateway\_name}.{self.tradeid}"

@dataclass

class PositionData(BaseData):

"""

持仓数据类，用于跟踪每个具体的持仓

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

direction: Direction # 持仓方向

volume: float = 0 # 持仓量

frozen: float = 0 # 冻结数量

price: float = 0 # 持仓均价

pnl: float = 0 # 盈亏额(profit and loss)

yd\_volume: float = 0 # 昨持仓

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

# 持仓在vt系统中的唯一代码，通常是vtSymbol.方向

self.vt\_positionid = f"{self.vt\_symbol}.{self.direction.value}"

@dataclass

class AccountData(BaseData):

"""

账户数据类

包括净值、冻结值和可用资金

"""

accountid: str # 账户代码

balance: float = 0 # 账户净值

frozen: float = 0 # 冻结值

def \_\_post\_init\_\_(self):

""""""

# 可用资金

self.available = self.balance - self.frozen

# 账户在vt中的唯一代码，通常是 Gateway名.账户代码

self.vt\_accountid = f"{self.gateway\_name}.{self.accountid}"

@dataclass

class LogData(BaseData):

"""

日志数据类

用于记录在界面上显示或存入文件的日志信息

"""

msg: str # 日志信息

level: int = INFO # 日志级别

def \_\_post\_init\_\_(self):

""""""

# 日志生成时间

self.time = datetime.now()

@dataclass

class ContractData(BaseData):

"""

合约详细信息类

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

name: str # 合约中文名

product: Product # 合约类型

size: float # 合约乘数

pricetick: float # 价格跳动（最小变动价位）

min\_volume: float = 1 # 最小委托量

stop\_supported: bool = False # 服务器是否支持停止单

net\_position: bool = False # 接口是否使用净持仓值

history\_data: bool = False # 接口是否提供历史K线数据

# 以下仅针对期权

option\_strike: float = 0

option\_underlying: str = "" # vt\_symbol of underlying contract

option\_type: OptionType = None # 期权类型

option\_expiry: datetime = None

option\_portfolio: str = ""

option\_index: str = "" # for identifying options with same strike price

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

@dataclass

class QuoteData(BaseData):

"""

询价单数据

包含一个询价的最后状态的追踪信息

高注：应该是正在开发中的功能，目前还没有哪个功能使用此数据。

"""

symbol: str

exchange: Exchange

quoteid: str

bid\_price: float = 0.0

bid\_volume: int = 0

ask\_price: float = 0.0

ask\_volume: int = 0

bid\_offset: Offset = Offset.NONE

ask\_offset: Offset = Offset.NONE

status: Status = Status.SUBMITTING

datetime: datetime = None

reference: str = ""

def \_\_post\_init\_\_(self):

""""""

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

self.vt\_quoteid = f"{self.gateway\_name}.{self.quoteid}"

def create\_cancel\_request(self) -> "CancelRequest":

"""

根据询价信息创建一个撤消单

"""

req = CancelRequest(

orderid=self.quoteid, symbol=self.symbol, exchange=self.exchange

)

return req

@dataclass

class SubscribeRequest:

"""

订阅行情请求

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

@dataclass

class OrderRequest:

"""

委托单请求

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

direction: Direction # 买卖方向

type: OrderType # 价格类型

volume: float # 数量

price: float = 0 # 价格

offset: Offset = Offset.NONE # 开平

reference: str = ""

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

def create\_order\_data(self, orderid: str, gateway\_name: str) -> OrderData:

"""

根据委托请求创建委托单数据

"""

order = OrderData(

symbol=self.symbol,

exchange=self.exchange,

orderid=orderid,

type=self.type,

direction=self.direction,

offset=self.offset,

price=self.price,

volume=self.volume,

reference=self.reference,

gateway\_name=gateway\_name,

)

return order

@dataclass

class CancelRequest:

"""

撤单请求

"""

orderid: str # 报单号

symbol: str # 合约代码

exchange: Exchange # 交易所代码

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

@dataclass

class HistoryRequest:

"""

查询历史数据请求

"""

symbol: str # 合约代码

exchange: Exchange # 交易所代码

start: datetime # 起始时间datetime对象

end: datetime = None # 结束时间datetime对象

interval: Interval = None # K线周期

def \_\_post\_init\_\_(self):

""""""

# 本地代码

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

@dataclass

class QuoteRequest:

"""

询价请求

发送给特定的接口，用于生成询价单

高注：应该是正在开发中的功能，目前还没有哪个功能使用此数据。

"""

symbol: str

exchange: Exchange

bid\_price: float

bid\_volume: int

ask\_price: float

ask\_volume: int

bid\_offset: Offset = Offset.NONE

ask\_offset: Offset = Offset.NONE

reference: str = ""

def \_\_post\_init\_\_(self):

""""""

self.vt\_symbol = f"{self.symbol}.{self.exchange.value}"

def create\_quote\_data(self, quoteid: str, gateway\_name: str) -> QuoteData:

"""

用询价申请创建一个询价单数据

"""

quote = QuoteData(

symbol=self.symbol,

exchange=self.exchange,

quoteid=self.quoteid,

bid\_price=self.bid\_price,

bid\_volume=self.bid\_volume,

ask\_price=self.ask\_price,

ask\_volume=self.ask\_volume,

bid\_offset=self.bid\_offset,

ask\_offset=self.ask\_offset,

reference=self.reference,

gateway\_name=gateway\_name,

)

return quote

## 开平转换器

交易接口接受的委托只能指定方向及开平等，要使用限价单（Limit Order）和停止单（Stop Order）等，需要由本系统将其转换成为交易接口能接受的委托单。开平转换器完成此功能。

vn.py在策略中用一种统一的方法处理交易。回测时不需要真正往交易服务器发请求，所以不需要这个转换器，实盘时才需要。

在D:\vnpy220\vnpy\trader下的converter.py文件中定义了开平转换器类和持仓类两个类。先介绍持仓类。

### 持仓类PositionHolding

通过接口可以取账户信息，但取得的原始账户信息不方便在自动交易过程中进行管理，不方便基于它执行策略。

本系统基于持仓类对账户进行管理。持仓类的每个实例对应一个vt\_symbol，可管理一个合约的仓位。代码如下：

class PositionHolding:

"""持仓类"""

def \_\_init\_\_(self, contract: ContractData):

"""初始化"""

self.vt\_symbol: str = contract.vt\_symbol # 本地代码

self.exchange: Exchange = contract.exchange # 交易所代码

self.active\_orders: Dict[str, OrderData] = {} # 活动委托单字典

self.long\_pos: float = 0 # 多头持仓量

self.long\_yd: float = 0 # 昨多头持仓量

self.long\_td: float = 0 # 今多头持仓量

self.short\_pos: float = 0 # 空头持仓量

self.short\_yd: float = 0 # 昨空头持仓量

self.short\_td: float = 0 # 今空头持仓量

self.long\_pos\_frozen: float = 0 # 多头冻结数量

self.long\_yd\_frozen: float = 0 # 昨多头冻结

self.long\_td\_frozen: float = 0 # 今多头冻结

self.short\_pos\_frozen: float = 0 # 空头冻结数量

self.short\_yd\_frozen: float = 0 # 昨空头冻结

self.short\_td\_frozen: float = 0 # 今空头冻结

def update\_position(self, position: PositionData) -> None:

"""根据仓位数据更新持仓信息"""

if position.direction == Direction.LONG:

self.long\_pos = position.volume

self.long\_yd = position.yd\_volume

self.long\_td = self.long\_pos - self.long\_yd

else:

self.short\_pos = position.volume

self.short\_yd = position.yd\_volume

self.short\_td = self.short\_pos - self.short\_yd

def update\_order(self, order: OrderData) -> None:

"""根据委托单数据更新持仓信息"""

# 如果委托单是活动的，就加入到活动委托单字典；如果不活动，则从活动委托单字典中弹出

if order.is\_active():

self.active\_orders[order.vt\_orderid] = order

else:

if order.vt\_orderid in self.active\_orders:

self.active\_orders.pop(order.vt\_orderid)

self.calculate\_frozen() # 计算冻结数量

def update\_order\_request(self, req: OrderRequest, vt\_orderid: str) -> None:

"""根据委托请求更新持仓信息"""

gateway\_name, orderid = vt\_orderid.split(".")

# 根据委托请求创建委托单数据

order = req.create\_order\_data(orderid, gateway\_name)

self.update\_order(order)

def update\_trade(self, trade: TradeData) -> None:

"""根据成交数据更新持仓信息"""

if trade.direction == Direction.LONG: # 方向为多

if trade.offset == Offset.OPEN: # 开多

self.long\_td += trade.volume

elif trade.offset == Offset.CLOSETODAY: # 平今多

self.short\_td -= trade.volume

elif trade.offset == Offset.CLOSEYESTERDAY: # 平昨多

self.short\_yd -= trade.volume

elif trade.offset == Offset.CLOSE: # 平多

if trade.exchange in [Exchange.SHFE, Exchange.INE]:

self.short\_yd -= trade.volume

else:

self.short\_td -= trade.volume

if self.short\_td < 0:

self.short\_yd += self.short\_td

self.short\_td = 0

else:

if trade.offset == Offset.OPEN:

self.short\_td += trade.volume

elif trade.offset == Offset.CLOSETODAY:

self.long\_td -= trade.volume

elif trade.offset == Offset.CLOSEYESTERDAY:

self.long\_yd -= trade.volume

elif trade.offset == Offset.CLOSE:

if trade.exchange in [Exchange.SHFE, Exchange.INE]:

self.long\_yd -= trade.volume

else:

self.long\_td -= trade.volume

if self.long\_td < 0:

self.long\_yd += self.long\_td

self.long\_td = 0

self.long\_pos = self.long\_td + self.long\_yd

self.short\_pos = self.short\_td + self.short\_yd

def calculate\_frozen(self) -> None:

"""计算冻结数量"""

self.long\_pos\_frozen = 0

self.long\_yd\_frozen = 0

self.long\_td\_frozen = 0

self.short\_pos\_frozen = 0

self.short\_yd\_frozen = 0

self.short\_td\_frozen = 0

# 对于所有活动委托单

for order in self.active\_orders.values():

# 忽略开仓单

if order.offset == Offset.OPEN:

continue

# 冻结数量 = 报单量 - 成交数量

frozen = order.volume - order.traded

if order.direction == Direction.LONG: # 方向为多

if order.offset == Offset.CLOSETODAY: # 平今多

self.short\_td\_frozen += frozen

elif order.offset == Offset.CLOSEYESTERDAY: # 平昨多

self.short\_yd\_frozen += frozen

elif order.offset == Offset.CLOSE: # 平多

self.short\_td\_frozen += frozen

if self.short\_td\_frozen > self.short\_td:

self.short\_yd\_frozen += (self.short\_td\_frozen

- self.short\_td)

self.short\_td\_frozen = self.short\_td

elif order.direction == Direction.SHORT:

if order.offset == Offset.CLOSETODAY:

self.long\_td\_frozen += frozen

elif order.offset == Offset.CLOSEYESTERDAY:

self.long\_yd\_frozen += frozen

elif order.offset == Offset.CLOSE:

self.long\_td\_frozen += frozen

if self.long\_td\_frozen > self.long\_td:

self.long\_yd\_frozen += (self.long\_td\_frozen

- self.long\_td)

self.long\_td\_frozen = self.long\_td

self.long\_pos\_frozen = self.long\_td\_frozen + self.long\_yd\_frozen

self.short\_pos\_frozen = self.short\_td\_frozen + self.short\_yd\_frozen

def convert\_order\_request\_shfe(self, req: OrderRequest) -> List[OrderRequest]:

"""转换委托单请求（针对上海期货交易所）"""

if req.offset == Offset.OPEN:

return [req]

if req.direction == Direction.LONG:

pos\_available = self.short\_pos - self.short\_pos\_frozen

td\_available = self.short\_td - self.short\_td\_frozen

else:

pos\_available = self.long\_pos - self.long\_pos\_frozen

td\_available = self.long\_td - self.long\_td\_frozen

if req.volume > pos\_available:

return []

elif req.volume <= td\_available:

req\_td = copy(req)

req\_td.offset = Offset.CLOSETODAY

return [req\_td]

else:

req\_list = []

if td\_available > 0:

req\_td = copy(req)

req\_td.offset = Offset.CLOSETODAY

req\_td.volume = td\_available

req\_list.append(req\_td)

req\_yd = copy(req)

req\_yd.offset = Offset.CLOSEYESTERDAY

req\_yd.volume = req.volume - td\_available

req\_list.append(req\_yd)

return req\_list

def convert\_order\_request\_lock(self, req: OrderRequest) -> List[OrderRequest]:

"""转换委托单请求（针对锁仓单）"""

if req.direction == Direction.LONG:

td\_volume = self.short\_td

yd\_available = self.short\_yd - self.short\_yd\_frozen

else:

td\_volume = self.long\_td

yd\_available = self.long\_yd - self.long\_yd\_frozen

# If there is td\_volume, we can only lock position

if td\_volume:

req\_open = copy(req)

req\_open.offset = Offset.OPEN

return [req\_open]

# If no td\_volume, we close opposite yd position first

# then open new position

else:

close\_volume = min(req.volume, yd\_available)

open\_volume = max(0, req.volume - yd\_available)

req\_list = []

if yd\_available:

req\_yd = copy(req)

if self.exchange in [Exchange.SHFE, Exchange.INE]:

req\_yd.offset = Offset.CLOSEYESTERDAY

else:

req\_yd.offset = Offset.CLOSE

req\_yd.volume = close\_volume

req\_list.append(req\_yd)

if open\_volume:

req\_open = copy(req)

req\_open.offset = Offset.OPEN

req\_open.volume = open\_volume

req\_list.append(req\_open)

return req\_list

def convert\_order\_request\_net(self, req: OrderRequest) -> List[OrderRequest]:

"""转换委托单请求（针对净仓单）"""

if req.direction == Direction.LONG:

pos\_available = self.short\_pos - self.short\_pos\_frozen

else:

pos\_available = self.long\_pos - self.long\_pos\_frozen

if not pos\_available:

req.offset = Offset.OPEN

return [req]

elif req.volume <= pos\_available:

req.offset = Offset.CLOSE

return [req]

else:

close\_volume = pos\_available

open\_volume = req.volume - close\_volume

close\_req = copy(req)

close\_req.offset = Offset.CLOSE

close\_req.volume = close\_volume

open\_req = copy(req)

open\_req.offset = Offset.OPEN

open\_req.offset = open\_volume

return [close\_req, open\_req]

### 开平转换器类OffsetConverter

除委托单转换的功能外，开平转换器维护一个持仓字典，实际上起到账户仓位管理的作用。

self.holdings = {} # 持仓字典，以本地代码vt\_symbol为索引，每个vt\_symbol对应一个持仓类实例

详细代码如下：

class OffsetConverter:

"""开平转换器"""

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

"""初始化"""

self.main\_engine: MainEngine = main\_engine

# 持仓字典，以本地代码vt\_symbol为索引，每个vt\_symbol对应一个持仓类实例

self.holdings: Dict[str, "PositionHolding"] = {}

def update\_position(self, position: PositionData) -> None:

"""根据仓位数据更新持仓信息"""

if not self.is\_convert\_required(position.vt\_symbol): # 净仓位不处理

return

holding = self.get\_position\_holding(position.vt\_symbol)

holding.update\_position(position)

def update\_trade(self, trade: TradeData) -> None:

"""根据成交数据更新持仓信息"""

if not self.is\_convert\_required(trade.vt\_symbol):

return

holding = self.get\_position\_holding(trade.vt\_symbol)

holding.update\_trade(trade)

def update\_order(self, order: OrderData) -> None:

"""根据委托单数据更新持仓信息"""

if not self.is\_convert\_required(order.vt\_symbol):

return

holding = self.get\_position\_holding(order.vt\_symbol)

holding.update\_order(order)

def update\_order\_request(self, req: OrderRequest, vt\_orderid: str) -> None:

"""根据委托请求更新持仓信息"""

if not self.is\_convert\_required(req.vt\_symbol):

return

holding = self.get\_position\_holding(req.vt\_symbol)

holding.update\_order\_request(req, vt\_orderid)

def get\_position\_holding(self, vt\_symbol: str) -> "PositionHolding":

"""取得本地代码对应的持仓实例"""

# 从持仓字典中取

holding = self.holdings.get(vt\_symbol, None)

# 如果没有取到，就创建一个，并加入到持仓字典中

if not holding:

contract = self.main\_engine.get\_contract(vt\_symbol)

holding = PositionHolding(contract)

self.holdings[vt\_symbol] = holding

return holding

def convert\_order\_request(

self,

req: OrderRequest,

lock: bool,

net: bool = False

) -> List[OrderRequest]:

"""转换委托单请求"""

# 净仓位，直接返回原请求

if not self.is\_convert\_required(req.vt\_symbol):

return [req]

# 取得持仓

holding = self.get\_position\_holding(req.vt\_symbol)

if lock:

# 如果是锁仓单

return holding.convert\_order\_request\_lock(req)

elif net:

# 如果是净仓单

return holding.convert\_order\_request\_net(req)

elif req.exchange in [Exchange.SHFE, Exchange.INE]:

# 如果是上海期货交易所

return holding.convert\_order\_request\_shfe(req)

else:

# 如果不是上述两种情况，则不需要转换

return [req]

def is\_convert\_required(self, vt\_symbol: str) -> bool:

"""

合约是否需要开平转换

"""

contract = self.main\_engine.get\_contract(vt\_symbol)

# 只有多/空仓位需要转换（净仓位不需要）

if not contract:

return False

elif contract.net\_position:

return False

else:

return True

# CTA引擎CtaEngine

## CtaEngine源代码分析

CTA引擎类在D:\vnpy220\vnpy\app\cta\_strategy目录下的engine.py文件中定义。

900多行程序，按源代码顺序，分为四个部分。

### 初始化部分

初始化部分代码为：

class CtaEngine(BaseEngine):

"""CTA策略引擎"""

engine\_type = EngineType.LIVE # 用于实盘

setting\_filename = "cta\_strategy\_setting.json"

data\_filename = "cta\_strategy\_data.json"

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

""""""

super(CtaEngine, self).\_\_init\_\_(

main\_engine, event\_engine, APP\_NAME)

self.strategy\_setting = {} # 策略参数字典（strategy\_name: dict）

self.strategy\_data = {} # 策略数据字典（strategy\_name: dict）

# 策略类字典（class\_name: stategy\_class），包含VN支持的（两个目录中）所有的策略类

self.classes = {}

# 策略字典（strategy\_name: strategy），仅包含已“添加”的策略

self.strategies = {}

# 本地代码-策略字典（vt\_symbol: strategy list）

self.symbol\_strategy\_map = defaultdict(list)

# 委托单号-策略字典（vt\_orderid: strategy）

self.orderid\_strategy\_map = {}

# 策略-委托单字典（strategy\_name: orderid list）

self.strategy\_orderid\_map = defaultdict(set)

self.stop\_order\_count = 0 # 用于生成stop\_orderid

self.stop\_orders = {} # 停止单字典（stop\_orderid: stop\_order）

# 线程池，使多个策略的初始化可以在多个线程中进行

self.init\_executor = ThreadPoolExecutor(max\_workers=1)

self.rq\_client = None

self.rq\_symbols = set()

self.vt\_tradeids = set() # 用于过滤重复的成交

# 开平转换器，对真实账户仓位进行管理

self.offset\_converter = OffsetConverter(self.main\_engine)

def init\_engine(self):

"""

初始化引擎，在“CTA策略”窗口初始化时被调用，

在VN Trader的生命周期中，“CTA策略”窗口只会被初始化一次。

"""

self.init\_rqdata() # 初始化RQData客户端

self.load\_strategy\_class() # 加载所有的策略类，注意是“所有”的，不只是已“添加”的

self.load\_strategy\_setting() # 从json文件中加载策略配置信息

self.load\_strategy\_data() # 从json文件中加载策略数据

self.register\_event() # 注意事件处理函数

self.write\_log("CTA策略引擎初始化成功")

def close(self):

"""在程序关闭时被调用"""

# 停止所有的策略

self.stop\_all\_strategies()

def register\_event(self):

"""注册事件"""

self.event\_engine.register(EVENT\_TICK, self.process\_tick\_event)

self.event\_engine.register(EVENT\_ORDER, self.process\_order\_event)

self.event\_engine.register(EVENT\_TRADE, self.process\_trade\_event)

self.event\_engine.register(EVENT\_POSITION, self.process\_position\_event)

def init\_rqdata(self):

"""

初始化RQData客户端

"""

result = rqdata\_client.init()

if result:

self.write\_log("RQData数据接口初始化成功")

### 与交易接口交互部分

与交易接口交互部分代码为：

def query\_bar\_from\_rq(

self, symbol: str, exchange: Exchange, interval: Interval, start: datetime, end: datetime

):

"""

从RQData查询K线数据

"""

req = HistoryRequest(

symbol=symbol,

exchange=exchange,

interval=interval,

start=start,

end=end

)

data = rqdata\_client.query\_history(req)

return data

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

"""处理EVENT\_TICK事件"""

# 取得Tick数据

tick = event.data

# 取使用tick.vt\_symbol的所有策略

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

if not strategies:

return

# 收到tick行情后，先处理本地停止单（检查是否要立即发出）

self.check\_stop\_order(tick)

# 调用各相关策略的on\_tick方法

for strategy in strategies:

if strategy.inited: # 如果策略已经初始化

# 调用各相关策略的on\_tick方法

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

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

"""处理委托推送"""

order = event.data

self.offset\_converter.update\_order(order)

# 取与vt\_symbol相关的所有策略

strategy = self.orderid\_strategy\_map.get(order.vt\_orderid, None)

if not strategy:

return

# 如果委托单已经不再活动，将vt\_orderid移除

vt\_orderids = self.strategy\_orderid\_map[strategy.strategy\_name]

if order.vt\_orderid in vt\_orderids and not order.is\_active():

vt\_orderids.remove(order.vt\_orderid)

# 如果是停止单，调用策略的on\_stop\_order函数

if order.type == OrderType.STOP:

so = StopOrder(

vt\_symbol=order.vt\_symbol,

direction=order.direction,

offset=order.offset,

price=order.price,

volume=order.volume,

stop\_orderid=order.vt\_orderid,

strategy\_name=strategy.strategy\_name,

status=STOP\_STATUS\_MAP[order.status],

vt\_orderids=[order.vt\_orderid],

)

self.call\_strategy\_func(strategy, strategy.on\_stop\_order, so)

# 调用策略的on\_order函数

self.call\_strategy\_func(strategy, strategy.on\_order, order)

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

"""处理成交推送"""

trade = event.data

# 过滤已经收到过的成交回报

if trade.vt\_tradeid in self.vt\_tradeids:

return

self.vt\_tradeids.add(trade.vt\_tradeid)

self.offset\_converter.update\_trade(trade)

strategy = self.orderid\_strategy\_map.get(trade.vt\_orderid, None)

if not strategy:

return

# 在调用on\_trade方法前，先修改策略仓位

if trade.direction == Direction.LONG:

strategy.pos += trade.volume

else:

strategy.pos -= trade.volume

self.call\_strategy\_func(strategy, strategy.on\_trade, trade)

# 将策略变量同步到数据文件

self.sync\_strategy\_data(strategy)

# 刷新界面

self.put\_strategy\_event(strategy)

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

"""处理仓位变化事件"""

position = event.data

self.offset\_converter.update\_position(position)

def check\_stop\_order(self, tick: TickData):

"""收到行情后处理本地停止单（检查是否要立即发出）"""

# 对于每个停止单

for stop\_order in list(self.stop\_orders.values()):

if stop\_order.vt\_symbol != tick.vt\_symbol:

continue

# 多头停止单是否被触发

long\_triggered = (

stop\_order.direction == Direction.LONG and tick.last\_price >= stop\_order.price

)

# 空头停止单是否被触发

short\_triggered = (

stop\_order.direction == Direction.SHORT and tick.last\_price <= stop\_order.price

)

if long\_triggered or short\_triggered:

strategy = self.strategies[stop\_order.strategy\_name]

# 买入和卖出分别以涨停跌停价发单（模拟市价单）

# 对于没有涨跌停价格的市场则使用5档报价

if stop\_order.direction == Direction.LONG:

if tick.limit\_up:

price = tick.limit\_up

else:

price = tick.ask\_price\_5

else:

if tick.limit\_down:

price = tick.limit\_down

else:

price = tick.bid\_price\_5

contract = self.main\_engine.get\_contract(stop\_order.vt\_symbol)

# 发出限价单

vt\_orderids = self.send\_limit\_order(

strategy,

contract,

stop\_order.direction,

stop\_order.offset,

price,

stop\_order.volume,

stop\_order.lock,

stop\_order.net

)

# 如果委托成功，修改停止单状态

if vt\_orderids:

# 从相关的字典中移除

self.stop\_orders.pop(stop\_order.stop\_orderid)

strategy\_vt\_orderids = self.strategy\_orderid\_map[strategy.strategy\_name]

if stop\_order.stop\_orderid in strategy\_vt\_orderids:

strategy\_vt\_orderids.remove(stop\_order.stop\_orderid)

# Change stop order status to cancelled and update to strategy.

stop\_order.status = StopOrderStatus.TRIGGERED

stop\_order.vt\_orderids = vt\_orderids

self.call\_strategy\_func(

strategy, strategy.on\_stop\_order, stop\_order

)

self.put\_stop\_order\_event(stop\_order)

def send\_server\_order(

self,

strategy: CtaTemplate,

contract: ContractData,

direction: Direction,

offset: Offset,

price: float,

volume: float,

type: OrderType,

lock: bool,

net: bool

):

"""向服务器发送一个新的委托单"""

#生成一个原始的委托单请求

original\_req = OrderRequest(

symbol=contract.symbol,

exchange=contract.exchange,

direction=direction,

offset=offset,

type=type,

price=price,

volume=volume,

reference=f"{APP\_NAME}\_{strategy.strategy\_name}"

)

# 用转换器转换委托单请求

req\_list = self.offset\_converter.convert\_order\_request(original\_req, lock, net)

# 发单

vt\_orderids = []

for req in req\_list:

vt\_orderid = self.main\_engine.send\_order(req, contract.gateway\_name)

# 检查委托单是否发送成功

if not vt\_orderid:

continue

vt\_orderids.append(vt\_orderid) # 增加到单号列表

# 根据委托请求更新持仓信息

self.offset\_converter.update\_order\_request(req, vt\_orderid)

# 保存单号与策略之间的联系

self.orderid\_strategy\_map[vt\_orderid] = strategy

self.strategy\_orderid\_map[strategy.strategy\_name].add(vt\_orderid)

return vt\_orderids

def send\_limit\_order(

self,

strategy: CtaTemplate,

contract: ContractData,

direction: Direction,

offset: Offset,

price: float,

volume: float,

lock: bool,

net: bool

):

"""

向服务器发送一个限价单

"""

return self.send\_server\_order(

strategy,

contract,

direction,

offset,

price,

volume,

OrderType.LIMIT,

lock,

net

)

def send\_server\_stop\_order(

self,

strategy: CtaTemplate,

contract: ContractData,

direction: Direction,

offset: Offset,

price: float,

volume: float,

lock: bool,

net: bool

):

"""

向服务器发送一个停止单请求

只能向支持停止单的交易服务器发送。

"""

return self.send\_server\_order(

strategy,

contract,

direction,

offset,

price,

volume,

OrderType.STOP,

lock,

net

)

def send\_local\_stop\_order(

self,

strategy: CtaTemplate,

direction: Direction,

offset: Offset,

price: float,

volume: float,

lock: bool,

net: bool

):

"""

创建一个新的本地停止单

"""

self.stop\_order\_count += 1

stop\_orderid = f"{STOPORDER\_PREFIX}.{self.stop\_order\_count}"

stop\_order = StopOrder(

vt\_symbol=strategy.vt\_symbol,

direction=direction,

offset=offset,

price=price,

volume=volume,

stop\_orderid=stop\_orderid,

strategy\_name=strategy.strategy\_name,

lock=lock,

net=net

)

self.stop\_orders[stop\_orderid] = stop\_order

vt\_orderids = self.strategy\_orderid\_map[strategy.strategy\_name]

vt\_orderids.add(stop\_orderid)

self.call\_strategy\_func(strategy, strategy.on\_stop\_order, stop\_order)

self.put\_stop\_order\_event(stop\_order)

return [stop\_orderid]

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

"""

根据vt\_orderid，撤消一个已经存在的委托单

"""

order = self.main\_engine.get\_order(vt\_orderid)

if not order:

self.write\_log(f"撤单失败，找不到委托{vt\_orderid}", strategy)

return

req = order.create\_cancel\_request()

self.main\_engine.cancel\_order(req, order.gateway\_name)

def cancel\_local\_stop\_order(self, strategy: CtaTemplate, stop\_orderid: str):

"""

撤消一个本地停止单

"""

stop\_order = self.stop\_orders.get(stop\_orderid, None)

if not stop\_order:

return

strategy = self.strategies[stop\_order.strategy\_name]

# Remove from relation map.

self.stop\_orders.pop(stop\_orderid)

vt\_orderids = self.strategy\_orderid\_map[strategy.strategy\_name]

if stop\_orderid in vt\_orderids:

vt\_orderids.remove(stop\_orderid)

# Change stop order status to cancelled and update to strategy.

stop\_order.status = StopOrderStatus.CANCELLED

self.call\_strategy\_func(strategy, strategy.on\_stop\_order, stop\_order)

self.put\_stop\_order\_event(stop\_order)

def send\_order(

self,

strategy: CtaTemplate,

direction: Direction,

offset: Offset,

price: float,

volume: float,

stop: bool,

lock: bool,

net: bool

):

"""

发送一个新的委托单

本函数被策略所调用。在策略中，种类交易函数统一成一个send\_order（策略的），

并在其中调用本引擎的此函数。

本函数中又会根据不同的委托单类型，调用不同的交易函数。

"""

contract = self.main\_engine.get\_contract(strategy.vt\_symbol)

if not contract:

self.write\_log(f"委托失败，找不到合约：{strategy.vt\_symbol}", strategy)

return ""

# Round order price and volume to nearest incremental value

price = round\_to(price, contract.pricetick)

volume = round\_to(volume, contract.min\_volume)

if stop:

if contract.stop\_supported:

return self.send\_server\_stop\_order(

strategy, contract, direction, offset, price, volume, lock, net

)

else:

return self.send\_local\_stop\_order(

strategy, direction, offset, price, volume, lock, net

)

else:

return self.send\_limit\_order(

strategy, contract, direction, offset, price, volume, lock, net

)

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

"""

撤销一个委托单

"""

if vt\_orderid.startswith(STOPORDER\_PREFIX):

self.cancel\_local\_stop\_order(strategy, vt\_orderid)

else:

self.cancel\_server\_order(strategy, vt\_orderid)

def cancel\_all(self, strategy: CtaTemplate):

"""

撤销一个策略中所有活动的委托单

"""

vt\_orderids = self.strategy\_orderid\_map[strategy.strategy\_name]

if not vt\_orderids:

return

for vt\_orderid in copy(vt\_orderids):

self.cancel\_order(strategy, vt\_orderid)

def get\_engine\_type(self):

"""取引擎类型，返回本CTA引擎是回测还是实盘"""

return self.engine\_type

def get\_pricetick(self, strategy: CtaTemplate):

"""

返回合约的价格跳动

"""

contract = self.main\_engine.get\_contract(strategy.vt\_symbol)

if contract:

return contract.pricetick

else:

return None

### 取行情数据部分

CTA引擎取行情数据部分代码如下：

def load\_bar(

self,

vt\_symbol: str,

days: int,

interval: Interval,

callback: Callable[[BarData], None],

use\_database: bool

):

"""在初始化策略时加载历史K线数据"""

symbol, exchange = extract\_vt\_symbol(vt\_symbol)

end = datetime.now(get\_localzone())

start = end - timedelta(days)

bars = []

# Pass gateway and RQData if use\_database set to True

if not use\_database:

# Query bars from gateway if available

contract = self.main\_engine.get\_contract(vt\_symbol)

if contract and contract.history\_data:

req = HistoryRequest(

symbol=symbol,

exchange=exchange,

interval=interval,

start=start,

end=end

)

bars = self.main\_engine.query\_history(req, contract.gateway\_name)

# Try to query bars from RQData, if not found, load from database.

else:

bars = self.query\_bar\_from\_rq(symbol, exchange, interval, start, end)

if not bars:

bars = database\_manager.load\_bar\_data(

symbol=symbol,

exchange=exchange,

interval=interval,

start=start,

end=end,

)

for bar in bars:

callback(bar)

def load\_tick(

self,

vt\_symbol: str,

days: int,

callback: Callable[[TickData], None]

):

"""在初始化策略时加载历史Tick数据"""

symbol, exchange = extract\_vt\_symbol(vt\_symbol)

end = datetime.now()

start = end - timedelta(days)

ticks = database\_manager.load\_tick\_data(

symbol=symbol,

exchange=exchange,

start=start,

end=end,

)

for tick in ticks:

callback(tick)

def call\_strategy\_func(

self, strategy: CtaTemplate, func: Callable, params: Any = None

):

"""

调用策略的函数，若触发异常则捕捉

"""

try:

if params:

func(params)

else:

func()

except Exception:

# 停止策略，修改状态为未初始化

strategy.trading = False

strategy.inited = False

# 发出日志

msg = f"触发异常已停止\n{traceback.format\_exc()}"

self.write\_log(msg, strategy)

可以看到，在CTA引擎中先在线取数据，如果取不到，就到数据库中取。在数据库中取可能会遇到两种情况：

1-数据库中的数据实时性差，可能最新的数据没有，造成策略在过期数据上做决策。

2-数据库中无该合约数据，那就要等策略运行够了load\_bar指定的天数后（还是ArrayManager指定的K线数？），才能真正开始交易。

所以，如果已经到了策略仿真或实盘阶段，最好能解决在线获取行情数据的问题。

### 与界面交互部分

与界面交互部分代码为：

def add\_strategy(

self, class\_name: str, strategy\_name: str, vt\_symbol: str, setting: dict

):

"""将策略添加到self.strategies中"""

# 如果该策略已经存在，写错误日志

if strategy\_name in self.strategies:

self.write\_log(f"创建策略失败，存在重名{strategy\_name}")

return

# 取策略类

strategy\_class = self.classes.get(class\_name, None)

if not strategy\_class:

self.write\_log(f"创建策略失败，找不到策略类{class\_name}")

return

if "." not in vt\_symbol:

self.write\_log("创建策略失败，本地代码缺失交易所后缀")

return

\_, exchange\_str = vt\_symbol.split(".")

if exchange\_str not in Exchange.\_\_members\_\_:

self.write\_log("创建策略失败，本地代码的交易所后缀不正确")

return

# 创建策略实例，并加入到策略字典

strategy = strategy\_class(self, strategy\_name, vt\_symbol, setting)

self.strategies[strategy\_name] = strategy

# 加入到本地代码字典

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

strategies.append(strategy)

# 更新配置文件cta\_strategy\_setting.json

self.update\_strategy\_setting(strategy\_name, setting)

# 刷新界面

self.put\_strategy\_event(strategy)

def init\_strategy(self, strategy\_name: str):

"""

初始化一个策略

"""

self.init\_executor.submit(self.\_init\_strategy, strategy\_name)

def \_init\_strategy(self, strategy\_name: str):

"""

在线程中初始化一个策略

"""

strategy = self.strategies[strategy\_name]

if strategy.inited:

self.write\_log(f"{strategy\_name}已经完成初始化，禁止重复操作")

return

self.write\_log(f"{strategy\_name}开始执行初始化")

# 调用策略的on\_init函数

self.call\_strategy\_func(strategy, strategy.on\_init)

# 恢复策略数据（变量）

data = self.strategy\_data.get(strategy\_name, None)

if data:

for name in strategy.variables:

value = data.get(name, None)

if value:

setattr(strategy, name, value)

# 订阅行情数据

contract = self.main\_engine.get\_contract(strategy.vt\_symbol)

if contract:

req = SubscribeRequest(

symbol=contract.symbol, exchange=contract.exchange)

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

else:

self.write\_log(f"行情订阅失败，找不到合约{strategy.vt\_symbol}", strategy)

# 推送事件，将策略状态改为“已初始化”

strategy.inited = True

self.put\_strategy\_event(strategy)

self.write\_log(f"{strategy\_name}初始化完成")

def start\_strategy(self, strategy\_name: str):

"""

启动一个策略

"""

strategy = self.strategies[strategy\_name]

if not strategy.inited:

self.write\_log(f"策略{strategy.strategy\_name}启动失败，请先初始化")

return

if strategy.trading:

self.write\_log(f"{strategy\_name}已经启动，请勿重复操作")

return

# 调用策略的on\_start函数

self.call\_strategy\_func(strategy, strategy.on\_start)

strategy.trading = True

# 刷新界面

self.put\_strategy\_event(strategy)

def stop\_strategy(self, strategy\_name: str):

"""

停止一个策略

"""

strategy = self.strategies[strategy\_name]

if not strategy.trading:

return

# 调用策略的on\_stop函数

self.call\_strategy\_func(strategy, strategy.on\_stop)

# 改变策略的交易状态

strategy.trading = False

# 取消策略的所有委托单

self.cancel\_all(strategy)

# 将策略变量保存到数据文件

self.sync\_strategy\_data(strategy)

# 刷新界面

self.put\_strategy\_event(strategy)

def edit\_strategy(self, strategy\_name: str, setting: dict):

"""

编辑一个策略的参数

"""

strategy = self.strategies[strategy\_name]

strategy.update\_setting(setting)

self.update\_strategy\_setting(strategy\_name, setting)

# 刷新界面

self.put\_strategy\_event(strategy)

def remove\_strategy(self, strategy\_name: str):

"""

移除一个策略

"""

strategy = self.strategies[strategy\_name]

if strategy.trading:

self.write\_log(f"策略{strategy.strategy\_name}移除失败，请先停止")

return

# Remove setting

self.remove\_strategy\_setting(strategy\_name)

# Remove from symbol strategy map

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

strategies.remove(strategy)

# Remove from active orderid map

if strategy\_name in self.strategy\_orderid\_map:

vt\_orderids = self.strategy\_orderid\_map.pop(strategy\_name)

# Remove vt\_orderid strategy map

for vt\_orderid in vt\_orderids:

if vt\_orderid in self.orderid\_strategy\_map:

self.orderid\_strategy\_map.pop(vt\_orderid)

# Remove from strategies

self.strategies.pop(strategy\_name)

return True

### 通用操作部分

通用操作部分代码为

def load\_strategy\_class(self):

"""

从源码中加载策略类

"""

path1 = Path(\_\_file\_\_).parent.joinpath("strategies")

self.load\_strategy\_class\_from\_folder(

path1, "vnpy.app.cta\_strategy.strategies")

path2 = Path.cwd().joinpath("strategies")

self.load\_strategy\_class\_from\_folder(path2, "strategies")

def load\_strategy\_class\_from\_folder(self, path: Path, module\_name: str = ""):

"""

从特定目录中加载策略类

"""

for dirpath, dirnames, filenames in os.walk(str(path)):

for filename in filenames:

if filename.split(".")[-1] in ("py", "pyd", "so"):

strategy\_module\_name = ".".join([module\_name, filename.split(".")[0]])

self.load\_strategy\_class\_from\_module(strategy\_module\_name)

def load\_strategy\_class\_from\_module(self, module\_name: str):

"""

从模块文件中加载策略类

"""

try:

module = importlib.import\_module(module\_name)

for name in dir(module):

value = getattr(module, name)

if (isinstance(value, type) and issubclass(value, CtaTemplate) and value is not CtaTemplate):

self.classes[value.\_\_name\_\_] = value

except: # noqa

msg = f"策略文件{module\_name}加载失败，触发异常：\n{traceback.format\_exc()}"

self.write\_log(msg)

def load\_strategy\_data(self):

"""

从json文件中加载策略数据

从cta\_strategy\_data.json取策略数据。

策略数据不只包括已添加的策略，只要曾经添加过的策略，都有记录。

"""

self.strategy\_data = load\_json(self.data\_filename)

def sync\_strategy\_data(self, strategy: CtaTemplate):

"""

将策略数据同步到json文件

"""

data = strategy.get\_variables()

data.pop("inited") # Strategy status (inited, trading) should not be synced.

data.pop("trading")

self.strategy\_data[strategy.strategy\_name] = data

save\_json(self.data\_filename, self.strategy\_data)

def get\_all\_strategy\_class\_names(self):

"""

返回所有已加载策略类的名称

"""

return list(self.classes.keys())

def get\_strategy\_class\_parameters(self, class\_name: str):

"""

取一个策略类的默认参数

"""

strategy\_class = self.classes[class\_name]

parameters = {}

for name in strategy\_class.parameters:

parameters[name] = getattr(strategy\_class, name)

return parameters

def get\_strategy\_parameters(self, strategy\_name):

"""

取一个策略的参数

"""

strategy = self.strategies[strategy\_name]

return strategy.get\_parameters()

def init\_all\_strategies(self):

"""

初始化所有策略

"""

for strategy\_name in self.strategies.keys():

self.init\_strategy(strategy\_name)

def start\_all\_strategies(self):

"""

启动所有策略

"""

for strategy\_name in self.strategies.keys():

self.start\_strategy(strategy\_name)

def stop\_all\_strategies(self):

"""

停止所有策略

"""

for strategy\_name in self.strategies.keys():

self.stop\_strategy(strategy\_name)

def load\_strategy\_setting(self):

"""

加载配置文件

从cta\_strategy\_setting.json中加载策略配置信息。

该文件中包括所有已添加的策略，及添加时进行的配置，包括策略名、本地代码及参数等。

"""

self.strategy\_setting = load\_json(self.setting\_filename)

for strategy\_name, strategy\_config in self.strategy\_setting.items():

self.add\_strategy(

strategy\_config["class\_name"],

strategy\_name,

strategy\_config["vt\_symbol"],

strategy\_config["setting"]

)

def update\_strategy\_setting(self, strategy\_name: str, setting: dict):

"""

更新配置文件cta\_strategy\_setting.json

"""

strategy = self.strategies[strategy\_name]

self.strategy\_setting[strategy\_name] = {

"class\_name": strategy.\_\_class\_\_.\_\_name\_\_,

"vt\_symbol": strategy.vt\_symbol,

"setting": setting,

}

save\_json(self.setting\_filename, self.strategy\_setting)

def remove\_strategy\_setting(self, strategy\_name: str):

"""

从配置文件中移除菶策略的配置信息

"""

if strategy\_name not in self.strategy\_setting:

return

self.strategy\_setting.pop(strategy\_name)

save\_json(self.setting\_filename, self.strategy\_setting)

def put\_stop\_order\_event(self, stop\_order: StopOrder):

"""

推送停止单事件

"""

event = Event(EVENT\_CTA\_STOPORDER, stop\_order)

self.event\_engine.put(event)

def put\_strategy\_event(self, strategy: CtaTemplate):

"""

推送一个事件，以更新策略状态

"""

data = strategy.get\_data()

event = Event(EVENT\_CTA\_STRATEGY, data)

self.event\_engine.put(event)

def write\_log(self, msg: str, strategy: CtaTemplate = None):

"""

创建CTA引擎的日志事件

"""

if strategy:

msg = f"{strategy.strategy\_name}: {msg}"

log = LogData(msg=msg, gateway\_name=APP\_NAME)

event = Event(type=EVENT\_CTA\_LOG, data=log)

self.event\_engine.put(event)

def send\_email(self, msg: str, strategy: CtaTemplate = None):

"""

向默认的接收者发送邮件

"""

if strategy:

subject = f"{strategy.strategy\_name}"

else:

subject = "CTA策略引擎"

self.main\_engine.send\_email(subject, msg)

## 进一步说明

### 仓位管理

vn.py采用真实账户仓位与具体策略仓位分别管理的方法。

一、真实账户仓位：

真实账户仓位是具体策略执行的基础，如策略不能平已冻结的仓位。

真实账户仓位用前述开平转换器进行管理，在CTA引擎的初始化函数中有如下代码：

# 开平转换器，对真实账户仓位进行管理

self.offset\_converter = OffsetConverter(self.main\_engine)

策略代码中没有操作offset\_converter的代码，但策略执行中，如果产生交易，则会对offset\_converter产生影响。

在引擎初始化时，为四类事件注册的处理函数：

def register\_event(self):

"""注册事件"""

self.event\_engine.register(EVENT\_TICK, self.process\_tick\_event)

self.event\_engine.register(EVENT\_ORDER, self.process\_order\_event)

self.event\_engine.register(EVENT\_TRADE, self.process\_trade\_event)

self.event\_engine.register(EVENT\_POSITION, self.process\_position\_event)

除EVENT\_TICK事件外，其他三个事件的处理函数都会对offset\_converter进行操作。另外，在向服务器发送委托请求时，也会对offset\_converter进行操作。特别是EVENT\_POSITION事件，每2秒钟就会被触发一次，所以vn.py基本上能够实时地对账户仓位进行管理。

二、具体策略仓位：

具体策略在运行的过程中，需要结合策略中的风控思路，进行自己的仓位管理。这个仓位管理局限于策略内部，不与其他策略相关联。vn.py可以启动多个策略，多个策略可以操作相同的合约。这多个策略分别管理自己的仓位，互不相干，避免不同策略发出相互矛盾的买卖信号，造成仓位的混乱，影响具体策略的执行逻辑。

具体策略的委托与成交，都会对账户仓位产生影响；账户仓位的现实情况也会对具体策略的委托单类型及成交产生影响。

### 关于“策略数据”

在CTA引擎中：

* 当策略处理成交推送时（在process\_trade\_event函数中），调用sync\_strategy\_data函数将策略数据保存到json文件cta\_strategy\_data.json中。
* 当某个策略停止时，也要保存策略数据。
* 当策略初始化时，从json文件中加载策略数据

这样做的目的是什么？下面以策略数据中的pos为例进行说明。

vn.py的CTA策略不是根据账户的实际仓位进行操作，而是在策略中用一个成员变量pos来保存仓位。

策略不可能永远执行，当因程序退出等原因，策略停止执行后，下一次再执行时，应该在上次策略执行的基础上继续，如上一次策略停止时的仓位情况。如果不这样，一旦程序退出，账户中的仓位就会成为死仓位，不会对策略今后的执行产生影响。

采用上述保存和加载策略数据的机制，就是保证策略的持续有效。

### CTA引擎与策略的关系

从CtaEngine的代码可以看出，CTA引擎与策略的直接交互非常少。

它们都是通过界面进行管理，通过底层接口进行驱动。

接口的Tick数据和K线数据驱动策略进行运算，根据运算结果发出委托。委托成交的结果返回来，驱动引擎对仓位进行更新。

# CTA策略界面

## 上层应用程序类CtaStrategyApp

上层应用程序类在D:\vnpy220\vnpy\app\cta\_strategy目录下的\_\_init\_\_.py文件中定义：

class CtaStrategyApp(BaseApp):

"""CTA策略应用"""

app\_name = APP\_NAME

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

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

display\_name = "CTA策略"

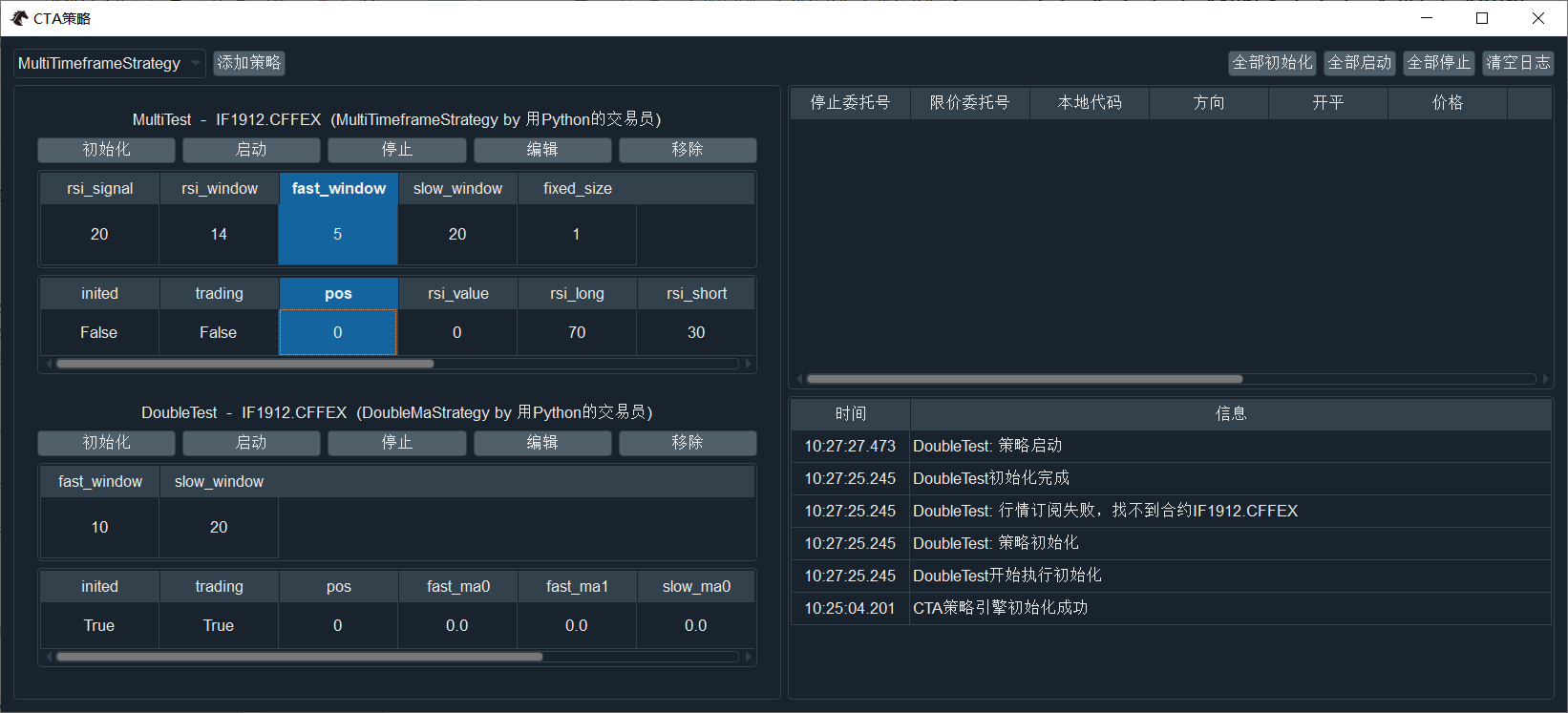
engine\_class = CtaEngine

widget\_name = "CtaManager"

icon\_name = "cta.ico"

## CTA策略窗口类CtaManager

执行时CTA策略窗口的界面如下图所示。



在D:\vnpy220\vnpy\app\cta\_strategy\ui目录下的widget.py文件中定义CTA策略窗口类CtaManager (QtWidgets.QWidget)。

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

在“CTA策略”窗口初始化时，调用CTA引擎的init\_engine函数对CTA引擎进行初始化，初始化的内容见函数注释。

CtaManager没有太实质化的功能，所有按钮的处理函数都在CTA引擎CtaEngine中。

CtaManager使用继承自BaseMonitor（参“在界面上使用Tick数据”一节）的子窗口StopOrderMonitor显示停止单，用同样继承自BaseMonitor的LogMonitor子窗口显示日志。

用StrategyManager子窗口管理单个策略。

对于大家目前的水平来说，CTA策略窗口类的代码相对简单，不再详述。

第五部分 其他策略

# 价差套利 (SpreadTrading)

价差套利策略，通过捕捉市场的不合理价差，买入被低估的资产，卖出被高估的资产，获得回归收益，达到资本盈利或避险的目的。套利交易风险小、回报稳定，对于大资金而言，如果单边重仓介入，将面临持仓成本较高、风险较大的不足；反之，如果单边轻仓介入，虽然可能降低风险，但其机会成本、时间成本也较高。因此整体而言，大资金单边重仓或单边轻仓介入均难以获得较为稳定和理想的回报。而大资金如以多空双向持仓介入，也就是进行套利交易，则既可回避单边持仓所面临的风险，又可能获取较为稳定的回报。

价差套利进一步细分为跨期套利、期现套利、跨品种套利和跨市场套利四种。

# 算法交易 (AlgoTrading)

略。

# 期权策略 (OptionMaster)

略。

# 脚本策略模块(script\_trader)

script\_trader：脚本策略模块，针对多标的组合类交易策略设计，同时也可以直接在命令行中实现REPL指令形式的交易，不支持回测功能

# 交易风险管理(RiskManager)

略。

# 投资组合模块(portfolio\_manager)

portfolio\_manager：投资组合模块，面向各类基本面交易策略，以独立的策略子账户为基础，提供交易仓位的自动跟踪以及盈亏实时统计功能

# RPC服务模块(rpc\_service)

rpc\_service：RPC服务模块，允许将某一VN Trader进程启动为服务端，作为统一的行情和交易路由通道，允许多客户端同时连接，实现多进程分布式系统