
vn.py 3.0.0 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，其他的附配资源可在购书后加入 QQ 群下载。

《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 VeighNa 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。

本文档的写作目的：

- 主要是对选用《Python 量化交易从入门到实战》一书作为教材的老师进行支持，方便他们针对 vn.py 的最新版本进行教学。
- 本文档的内容编排会与教材相对照，便于教材的读者学习。以教材为主线以文档为参考，或者以文档为主线以教材为参考都可以。
- 本文档是对教材内容的有效补充。随着 vn.py 的版本升级，书中有些代码与 vn.py 新版本会有所不同。对于这样的内容，本文档会重新编写，读者学习时请以本文档为主。本文档与教材的不同之处会加以标注并特别说明。
- **推销书籍不是目的。无需购买书籍，借助本文档以及一点点探索精神，完全能够掌握 vn.py。**
- 写作本文档的另一个主要目的是对 vn.py 的开源精神做出一点支持，希望本文档能够对大家学习使用 vn.py 有所帮助。

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。

本部分目标：理清 vn.py 程序的整体架构。

第 0 章 概述

0.1. 量化交易基础

本节是对教材 15.1 节内容的补充。

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、卷积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，不能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。

0.2. vn.py 版本变迁

本节是本文档新增内容。

要深入了解一个系统，可以从了解它的发展史开始。对于技术基础比较扎实的人，通过 vn.py 的版本变迁可以了解它的功能变化、技术路线、背景知识等，从而对 vn.py 建立一个全局认识。本节还包括我对各个版本的一些学习体会，仅是一家之言，供参考。

0.2.1. 1.x 版

vn.py 开源量化交易平台由上海韦纳软件科技有限公司开发，于 2015 年 3 月首次发布于 Github。

vn.py 的早期版本基于 Python2，基于 Python2 的最后一个版本是 v1.9.2 LTS，发布于 2019 年 1 月 2 日。

我学 Python 的目的很明确，就是量化交易。从一开始就有关关注 vn.py，但我学的是 Python3，那时 vn.py 还处于版本 1.x 时期，所以只能望 vn.py 兴叹。

0.2.2. 2.0 版

vn.py 从版本 2 开始基于 Python3。2.0.0 版发布于 2019 年 2 月 27 日，除 Python 版本的升级外，主要增强包括：

- 完成整个框架的搭建升级。
- 全新开发的量化交易 GUI 管理工具 VN Station。

vn.py 2.0 出来之后我并没有及时注意，等反应过来已经是 2.0.7 版。很兴奋，认真研究，并将心得写成《vn.py 2.0.7 源代码深入分析》，分享在 vn.py 社区的经验分享板块，至今仍被置顶。

出于对量化交易的爱好，出于对 Python 在量化交易中作用的认同，一定程度受 vn.py 强大功能的鼓舞，我与同事合写了《Python 量化交易从入门到实战》一书，对 vn.py 的讨论是其中很重要的一部分内容。

后续又写了《vn.py 2.1.4 源代码深入分析》和《vn.py 2.2.0 源代码深入分析》两个文档。当时 vn.py 自己的文档还不健全，据说有不少同道中人都是看着我的这些文档对 vn.py 由陌生到熟识，也算是我对 vn.py 开源精神的一点支持吧。

后来 vn.py 本身提供了很好的文档支持，比我权威，我就不准备再写了。但对 vn.py 的关注始终未变，而且发现 vn.py 本身的文档偏重使用，而我主要是从程序员的角度来分析代码，并且我以单一文档的形式提供，各部分之间更具相关性。另外，我也要选用《Python 量化交易从入门到实战》作为教材的老师进行支持，所以当 vn.py 的版本变化积累到一定程度时，我还是会写针对某个版本的代码深入分析文档。

vn.py 的版本号由三位数字组成，原来的版本变化是变第 3 位，所以从 2.0.0 到 2.2.0 经历了 21 个版本。从那之后，vn.py 的版本是变第 2 位，经历了从 2.3.0 到 2.9.0 共 7 个版本。每个版本的变化都比较大，特别是从版本 2.5.0 开始。

0.2.3. 2.5.0 版

发布于 2021-08-16，主要变化包括：

- 实现了 Web 应用后端服务
- 包含了对底层数据库结构的修改

Web 应用后端服务据说在社区呼声已久，想法很好，应该是激动人心的功能。但目前只实现了 Web 应用的后端（提供了给浏览器访问数据的接口），而前端页面（也就是浏览器中看到的网页）则交给社区用户来实现。我这些年已经不再编写 Web 应用程序，所以对我暂时用处不大。而且目前的服务功能也还不太完备，目前只支持基础的手动交易功能，策略交易相关的管理功能（比如 CtaStrategy 的相关调用）要在后面陆续增加。所以我暂时不研究。

对底层数据库的结构进行了修改。原来数据库中有 3 个表，一个 K 线数据表，一个 Tick 数据表和一个概览表。现在数据库中虽然还是这 3 个表，但字段数都有所增加。数据属性更丰富当然是好的，但如果需要继续使用以前版本的数据，则需要对数据库进行手动迁移操作，好在不麻烦。

除上述两个改变之外，还有一个令我激动的加强，就是增加了 TTS 交易接口。

CTP API 在过去 10 年间，已经成为了国内金融市场的交易 API 标准。但 CTP 动不动就停，一旦停了，vn.py 的很多功能就无法执行；即使服务没有停，也经常会因为连接的用户太多而返回 4097 错误，我可以说是苦 CTP 久矣！

知乎网友 krenx 推出了 OpenCTP 项目（CTP 开放平台），同样是在兼容 CTP API 的基础上（只需替换 dll），自主实现了整套 CTP 柜台的仿真交易功能（除了期货外，还引入了指数、股票等更多品种的支持），让大家在 SimNow 之外又多了一个选择。2.5.0 版本增加了对 OpenCTP 交易系统的支持，接口名称为 TtsGateway（Tick Trading System）。

但是，我使用 TTS 交易接口还没有连接成功过，比较失望。

0.2.4. 2.6.0 版

发布于 2021-09-25，新增了一系列专门针对金融时序数据的高性能数据库支持，包括：DolphinDB、Arctic 和 LevelDB，大幅提高各类量化策略回测研究的效率。

DolphinDB

DolphinDB 是由浙江智奥科技有限公司研发的一款高性能分布式时序数据库，特别适用于对速度要求极高的低延时或实时性任务，在国内外金融投资领域有着丰富的应用案例。

Arctic (MongoDB)

由英国量化对冲基金 Man AHL 基于 MongoDB 开发的高性能金融时序数据库，支持直接存储 pandas 的 DataFrame 和 numpy 的 ndarray 对象，在量化投研中非常实用。

LevelDB

由 Google 推出的高性能 Key/Value 数据库，基于 LSM 算法实现进程内存存储引擎，支持数十亿级别的海量数据。LevelDB 的定位是通用性数据存储方案，对于金融领域的时序数据存储没有特别大的优势，但也比一般 SQL 类数据库要多。

对我来说，2.6.0 版最好的增强是对数据服务的支持更加灵活。以前 vn.py 默认只支持 RQData，2.6.0 版增加了对 Udata、TuShare 和 TQSDK 的支持。

0.2.5. 2.7.0 版

发布于 2021-10-26，新增了一系列金融机构投资者常用的数据服务接口，包括：万得 Wind、同花顺 iFinD 和天软 Tinysoft，满足机构用户在使用 vn.py 过程中的数据获取需求。

0.2.6. 2.8.0 版

发布于 2021-12-10，对证券相关的 API 接口进行了更新和升级，包括：华鑫证券奇点极速柜台接口升级 4.0 版本，新增东方证券 OST 极速柜台接口，满足合格机构投资者使用 vn.py 进行证券程序化交易的需求。

0.2.7. 2.9.0 版

发布于 2021-12-30，完成了整个模块剥离计划。至此 vn.py 的交易接口 (gateway)、应用模块 (app)、数据库适配器 (database)、数据服务接口 (datafeed) 等都已经完成剥离，并支持通过 pip 按需安装和快速升级。

事实上，从 2.3.0 开始，vn.py 一直都在做着一项工作——模块剥离。

原来 vn.py 的源代码都是打包在一起的，包括 vn.py 的所有功能。从 2.3.0 开始，逐渐将一些模块实现为单独的项目，并可使用 pip 单独安装。开始时我对 vn.py 的这项工作持否定态度，后来我也逐渐认识到了该项工作的必要性。vn.py 从 2.0 版本开始因为功能太多导致打包极为复杂。而剥离后的各单独项目由于只包含该接口的相关代码，且没有复杂的依赖库，可以非常方便地进行发行版打包上传 pypi，也同时实现了 Windows 和 Ubuntu 系统下的 pip install 快速安装。

顺便说一下，以前下载的 vn.py 源代码压缩包都是几十 M，从 2.9.0 开始只有几百 K。

剥离后的模块一览表

交易接口模块（包括API封装和接口实现）		
功能模块	对接系统	支持市场
vnpy_ctp	CTP柜台	期货、期货期权
vnpy_mini	CTP MINI柜台	期货、期货期权
vnpy_femas	飞马柜台	期货
vnpy_uft	恒生UFT柜台	期货、期货期权、ETF期权
vnpy_esunny	易盛柜台	期货、黄金TD、外盘期货
vnpy_nhtd	南华NHTD柜台	期货、期货期权、ETF期权
vnpy_sopt	CTP证券柜台	ETF期权
vnpy_sec	顶点飞创柜台	ETF期权
vnpy_hts	顶点HTS柜台	ETF期权
vnpy_xtp	中泰XTP柜台	股票、两融、ETF期权
vnpy_tora	华鑫奇点柜台	股票、ETF期权
vnpy_hft	国泰君安证券 统一接入网关	股票、两融
vnpy_ost	东证OST柜台	股票
vnpy_sgit	飞鼠柜台	黄金TD
vnpy_ksgold	金仕达黄金柜台	黄金TD
vnpy_rohon	融航资管系统	期货、期货期权
vnpy_comstar	ComStar交易系统	债券
vnpy_tap	易盛外盘柜台	外盘期货
vnpy_da	直达柜台	外盘期货
vnpy_ib	Interactive Brokers	外盘市场
vnpy_tts	TTS仿真系统	仿真（期货、股票）

策略应用模块		
功能模块	模块名称	应用领域
vnpy_ctastrategy	CTA策略模块	策略模板、历史回测、参数优化、实盘交易
vnpy_ctabacktester	CTA回测模块	基于图形界面实现CTA策略投研功能
vnpy_spreadtrading	价差交易模块	自定义价差、价差盘口计算、价差执行算法
vnpy_optionmaster	期权交易模块	波动率跟踪、希腊值风控、电子眼算法
vnpy_portfoliostrategy	组合策略模块	多标的组合策略的开发、回测和实盘
vnpy_algotrading	算法交易模块	算法交易执行：TWAP、Sniper、Iceberg
vnpy_scripttrader	脚本策略模块	命令行REPL交互式交易、脚本化策略交易
vnpy_paperaccount	本地仿真模块	本地模拟撮合、委托成交推送、持仓数据记录
vnpy_chartwizard	K线图表模块	K线历史数据显示、实时Tick推送更新
vnpy_portfoliomanager	组合管理模块	策略委托成交记录、仓位跟踪、实时盈亏计算
vnpy_rpcservice	RPC服务模块	跨进程RPC服务端、标准化RPC接口
vnpy_datamanager	数据管理模块	历史数据下载、CSV数据读写、数据库管理
vnpy_datarecorder	行情录制模块	Tick数据录制、K线合成录制
vnpy_excelrtd	Excel RTD模块	基于pyxll的Excel数据实时推送更新
vnpy_riskmanager	风险管理模块	交易流控、单笔上限控制、撤单数量控制
vnpy_webtrader	Web服务模块	提供Web服务的REST API、Websocket推送

数据库适配器模块		
数据库分类	功能模块	对接数据库
SQL	vnpy_sqlite	SQLite
	vnpy_mysql	MySQL
	vnpy_postgresql	PostgreSQL
NoSQL	vnpy_dolphindb	DolphinDB
	vnpy_arctic	Arctic
	vnpy_mongodb	MongoDB
	vnpy_influxdb	InfluxDB
	vnpy_leveldb	LevelDB

数据服务模块		
服务分类	功能模块	对接服务
云端	vnpy_rqdata	米筐RQData
	vnpy_udata	恒生UData
	vnpy_tushare	TuShare
	vnpy_tqsdsk	天勤TQSDK
客户端	vnpy_tinysoft	天软TinySoft
	vnpy_wind	万得Wind
	vnpy_ifind	同花顺iFinD

另外，从 2.9.0 开始，VN Station 改称为 Veighna Station。Veighna 是上海韦纳软件科技有限公司申请注册的文字商标。因为我一直都是使用源代码，对 VN Station 的版本变化没有关心。好像其中包含一些有用的功能，有兴趣的读者可以安装试用。

同时，原来的主 GUI 界面“VN Trader”也改称为“Veighna Trader 社区版”。

0.2.8. 3.0.0 版

发布于 2022-03-23，主要增强包括：

- 采用 Python 3.10 作为核心支持（也保持了对 3.7、3.8、3.9 的兼容），同时对周边插件模块进行了相应的编译升级。
- 重构 VeighNa Station。趁这次大版本的升级，对整个 VeighNa Station 进行了一次重构，一方面致力于解决过去发现的各种问题，另一方面也尽可能利用当下 Python 的新技术特性，来打造一款称得上“好用”的产品。
- 在安装使用上，以 vnpy_开头的各模块不再默认安装。
- 在编程实现上，用 PySide6 代替了 PyQt5。

后面两点与安装编程有关，将在后文详细说明。

这是我一直期待的一个版本，所以它刚一推出，我就立刻开始试用，并着手整理本文档。

第 1 章 vn.py 的使用

对应《Python 量化交易从入门到实战》一书的第 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。

1.1. vn.py 的安装与运行

本节是对教材 15.2 节内容的补充。本节内容比教材更具体，学习时可以本节内容为主。

1.1.1. 准备环境

vn.py 3.0.0 版采用 Python 3.10 作为核心支持，同时也保持了对 Python3.7、3.8 和 3.9 的兼容。

对于初学者，推荐用下面方法准备环境。

安装 Anaconda3。Anaconda 的安装使用方法参《Python 量化交易从入门到实战》一书的第 6 章。

在 Anaconda3 上使用“conda create -n vnp300 python=3.10”命令，创建一个名为 vnp300 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。

使用如下命令：

```
activate vnp300
```

切换到 vnp300 虚拟环境。

1.1.2. 源码下载与安装

vn.py 的官网是 <http://www.vnpy.com/>。

单击主页上的“Gitee 仓库”链接，转到 Gitee（码云）去下载。如果读者使用的是 Windows 操作系统，请下载 Zip 文件。

下载的 Zip 文件中包含一个目录，将该目录解压到 D: 盘的根目录并改名为 D:\vnp300，本文档后续内容都假设 vn.py 安装在此目录当中。该目录中应该包含 install.bat 等文件。

注意，执行下面安装步骤前，需要先将 Anaconda 虚拟环境切换到 vnp300。

将目录切换到 D:\vnp300，执行其中的 install.bat 批处理文件进行安装，自动下载安装需要的包。

根据计算机操作系统的不同及网络情况的差异，安装过程中可能会遇到一些问题，导致安装不成功，可以多尝试几次。

安装经验

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。除本人的网络环境外，还与 pypi 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。
- D:\vnp300 中的文件 requirements.txt 中列出了所有需要安装的第三方包，每个包都指明了版

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。

PySide 和 PyQt5

如前所述，`vn.py 3.0.0` 版在编程实现上用 PySide6 代替了 PyQt5，下面是一篇网络文章，对两者的区别进行了说明。

文章标题：PySide2 和 PyQt5 区别

作者：生化环材

原谅链接：<https://xw.qq.com/cmsid/20210507A013GM00>

1 开源协议不同

PySide2 使用的是 LGPL 协议，通过一定的手段（调用库、wrapper 等），可以在发布程序时合法合理地闭源或使用。

PyQt5 有两种授权协议：GPL 开源协议或商用闭源协议。前者意味着直接使用 PyQt5 的程序也需要基于 GPL 协议开源，而后者需要支付购买费用才能闭源使用。购买 PyQt5 的商业授权按年收费。

2 资料丰富程度不同

目前 PySide2 的资料明显少于 PyQt5。不过也许因为大家对于版权意识的增强，目前较新的教程资料 PySide2 和 PyQt5 不相上下。

另一方面，PySide2 是 Qt 公司的亲儿子，利益使然，PySide2 的官方资料比较少。不过幸运的是，很多时候把代码里的“PyQt5”改成“PySide2”就能完美将相关代码迁移。因此 PySide2 官方文档少的缺点无关痛痒。

3 选哪个？

要让我选，我当然会选开源程度更高的 PySide2。如前文所述，PySide2 显然更加开放，而且因为两种 Python 图形化框架相似，PyQt5 的资料也可以应用到 PySide2 上，一定程度上弥补了 PySide2 资料少的缺点。

以上是生化环材文章内容的摘要。

与 PyQt 相比，PySide 的问世比较晚。当时 Nokia 与 Riverbank Computing 商谈，希望 PyQt 能添加对 LGPL 协议的支持，这对于很多商业用户会更加友好，但是 Riverbank Computing 不同意。

背景知识：Trolltech 是 Qt 的开发者，后来 Nokia 收购了 Trolltech，因此当时 Nokia 是 Qt 的实际拥有者。PyQt 是 Python 对 Qt 的包装，Riverbank Computing 是 PyQt 的开发者。Qt 的开源版本支持的是 LGPL 协议，因此 Nokia 希望 PyQt 也能支持 LGPL 协议。

由于商谈未成，Nokia 决定开发自己的 Python 开发工具包，并于 2009 年 8 月发布了支持了 LGPL 协议的 PySide（PyQt 的对标产品）。

2011 年，Nokia 将 Qt 的商业许可卖给 Digia。

2012 年，Nokia 将 Qt 完全卖给 Digia，后者在 2012 年底推出了 Qt5。

2015 年 10 月 14 日 PySide 1.2.4 发布，支持 Qt 4.8.7 框架。兼容 Python2.6 和 2.7（采用 MSVC2008 构建），同时兼容 Python3.3 和 3.4（采用 MSVC2010 构建）。

反观 PyQt，在 Qt5 推出的半年内（2013 年 6 月）就发布了支持 Qt5 的 PyQt5，比 PySide 要快得多。

PySide 对 Qt5 提供支持的计划从 2014 年开始筹备，也就是 2015 年上马的 Qt for Python 项目。该

项目开发的包命名为PySide2,以表示与老一代PySide的不同。PySide对标PyQt4,而PySide2对标PyQt5。

2020年12月8日,Qt6正式发布,这是一个里程碑式的新版本,它的使命是使Qt成为面向未来的开发平台。Qt6增加了很多新特性,但也有部分Qt5的特性不再支持。所以Qt5的程序不一定能顺利升级到Qt6。我看了一下列表,Qt5中不再支持的特性都是我不太使用的,但我还是没敢对自己的Qt程序进行升级。

支持Qt6的PyQt版本PyQt6于2021年1月发布,支持Qt6的PySide版本PySide6于2020年12月底发布。这一次双方的动作都很快,从中也可以看出PySide的进步。

vn.py的GUI原来使用PyQt5开发,从3.0.0开始全面转换为PySide6。由于vn.py 3.0.0版的Python版本和Gui工具包都有较大变化,这也是我推荐大家在全新的Anaconda虚拟环境中安装vn.py 3.0.0版的原因。

关于vnpy_包

在每个版本的vn.py源代码目录中,都有一个requirements.txt,该文件的意义请参考教材。

如前文所述,vn.py 2.9.0已经完成了整个模块的剥离计划。在vn.py 2.9.0的requirements.txt中,会要求安装所有打包的vn.py模块,内容如下:

```
six
wheel
tzlocal==2.0.0
PyQt5==5.14.1
pyqtgraph
qdarkstyle
numpy==1.18.2
pandas
matplotlib
seaborn
PyCryptodome==3.9.9
rqdatac==2.9.38
ta-lib==0.4.17
ibapi
deap
pyzmq
wmi
QScintilla==2.11.4
plotly
exchange-calendars
psycpg2-binary

vnpy_rest==1.0.3
```

```
vnpy_websocket==1.0.2
vnpy_ctp==6.5.1.7
vnpy_ctptest==6.5.1.3
vnpy_xtp==2.2.27.7
vnpy_tts==6.5.1.1
.....

vnpy_ctastrategy==1.0.6
vnpy_ctabacktester==1.0.4
vnpy_riskmanager==1.0.2
vnpy_datamanager==1.0.4
.....

vnpy_sqlite==1.0.0
.....

vnpy_rqdata==2.9.38.1
vnpy_tushare==1.2.64.2
.....
```

vn.py 3.0.0 版，requirements.txt 文件的完整内容如下：

```
tzlocal==2.0.0
PySide6==6.2.3
pyqtgraph==0.12.3
qdarkstyle==3.0.3
numpy==1.21.5
pandas==1.3.5
matplotlib==3.5.1
seaborn==0.11.2
ta-lib==0.4.24
deap==1.3.1
pyzmq==22.3.0
plotly==5.5.0
importlib-metadata==4.10.1
```

首先可以看到不再有 PyQt5，而是换成了 PySide6 的 6.2.3 版本。

另外，就是不再包含任何 vnpy_包的自动安装，也就是说，在新建的 Anaconda 虚拟环境中，这些包需要手工安装。这样做的好处是，您可以根据自己的需要，用到哪个包就下载哪个包，既可以使环境小型化，还有利于避免冲突。这样做也不麻烦，如果缺包，启动系统时自然会抛出异常，按照提示使用 pip install 命令安装一下就是了。

根据本文档后续的内容，您可能需要安装如下的包：

```
vnpy_ctastrategy
```

```
vnpy_ctabacktester
vnpy_ctp
vnpy_datamanager
vnpy_mongodb
vnpy_sqlite
vnpy_rqdata
vnpy_tts
vnpy_tushare
```

如果您的 Python 环境中安装过 vn.py 的早期版本，比如 2.9.0，如果环境中已经安装了这些包，可以选择：

- 不重新安装。因为原来的包既然装上了，就还能正常运行。
- 重新安装。有些包是带有界面的，如回测应用包 vnpy_ctabacktester，这些包的最新版本也已经更新使用 PySide6。

如果您按照本文档学习 vn.py，建议重新安装这些包：

- 重新安装可以感受系统的最新效果。
- 本文档需要针对这些包的最新版本分析其源代码。

如果您重新安装了这些包，使用 pip list 命令查看，可以看到下面内容：

```
.....
vnpy                3.0.0
vnpy-ctabacktester  1.0.5
vnpy-ctastrategy    1.0.8
vnpy-ctp            6.5.1.10
vnpy-datamanager    1.0.5
vnpy-mongodb        1.0.1
vnpy-rqdata         2.9.44.1
vnpy-sqlite         1.0.0
vnpy-tts            6.5.1.2
vnpy-tushare        1.2.64.3
.....
```

可以看出，与界面相关的包，版本与 2.9.0 要求的版本相比都有所更新。

1.1.3. 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:\vnpy300 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_ctp import CtpGateway
from vnpy_ctastrategy import CtaStrategyApp
```

```
from vnpy_ctabacktester import CtaBacktesterApp
from vnpy_datamanager import DataManagerApp
```

```
def main():
    """Start VeighNa 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()
```

vn.py 建议的主程序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。在目录 D:\vnpy300\examples\veighna_trader 下有一个示例程序 run.py，其中包含了绝大多数 vn.py 模块的加载方法，可以参考。例如，如果要加载前文所述的 TTS 交易接口，需要在程序中增加如下两句：

```
from vnpy_tts import TtsGateway
.....
main_engine.add_gateway(TtsGateway)
```

打开 CMD，切换到 vnpy300 虚拟环境，转到 D:\vnpy300 目录，运行下面命令：

```
python run.py
```

就可以启动 vn.py。

上述方法比较笼统，如果您按本文档前文建议的方法安装了 Anaconda3 并创建的虚拟环境，可用如下方法启动 vn.py：

- 在 Windows 的系统菜单中选择 Anaconda3，执行其中的 Anaconda Prompt (Anaconda3) 命令。
- 将虚拟环境切换到 vnpy300。
- 将目录切换到 D:\vnpy300。
- 执行 python run.py 命令。

如果您需要经常使用 `vn.py`，每次执行上述步骤比较麻烦，可以将上述步骤做到批处理文件中。

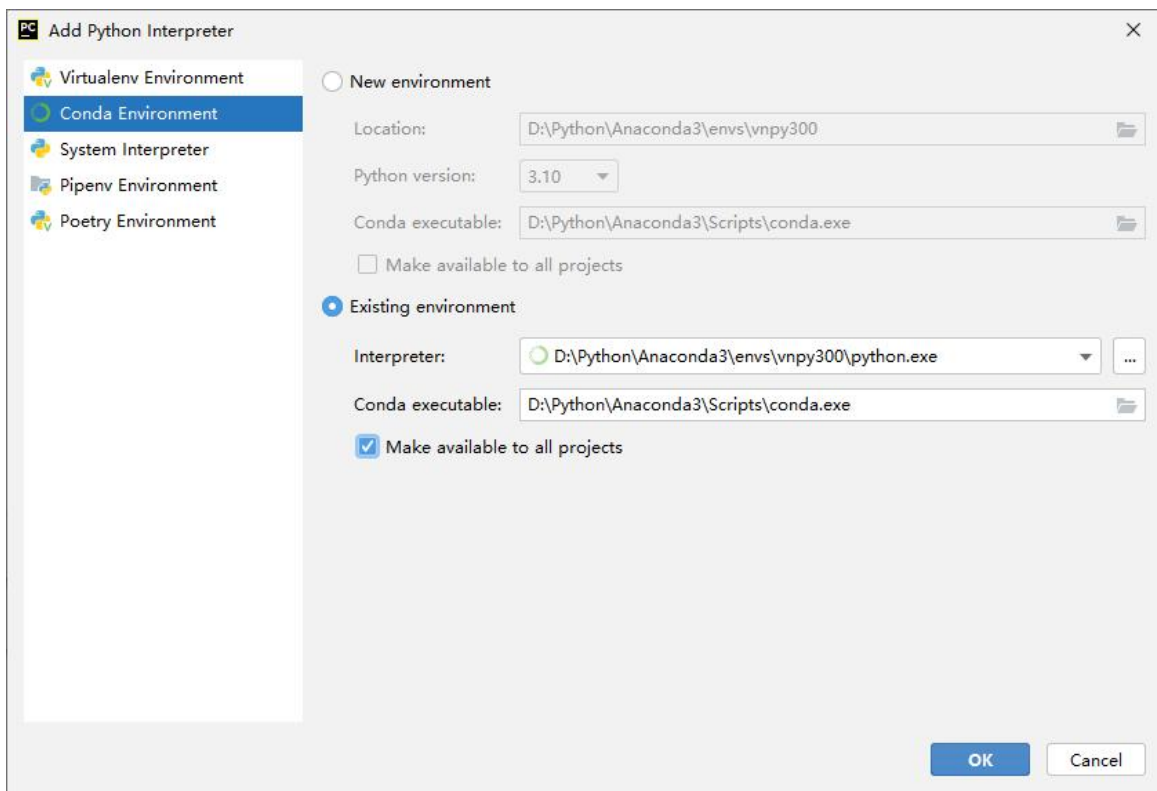
注：代码中红色的部分，是与教材不同的部分，也就是 `vn.py 3.0.0` 与 `vn.py 2.1.4` 不同的部分。本文档后面都会遵循此规则。

从代码中红色的部分可以看出，原来导入本地模块，现在是导入第三方包。这就是前面所述“模块剥离计划”的表现。从后文可以看出，`vnpy_`包提供的仍然是源代码，对我们的分析、学习和定制没有太大影响。

1.1.4. 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。

关于 IDE 的选择，《Python 量化交易从入门到实战》一书推荐使用 PyCharm。把 `D:\vnpy300` 目录作为一个项目打开，按照教材 3.5 节的方法，将 PyCharm 的 Python 解释器设为虚拟环境 `vnpy300` 的解释器，如下图所示。

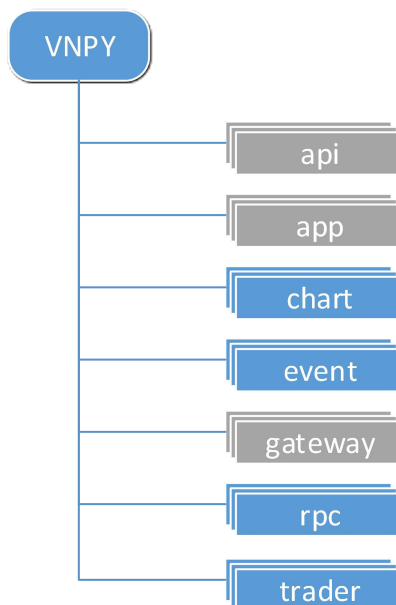


执行项目主目录中的 `run.py`，可正常运行。这样就可以对代码进行修改和调试。

1.1.5. 代码目录结构

如前所述，假设 `vn.py` 的源代码解压缩到 `D:\vnpy300` 目录，则该目录下有 3 个子目录，分别是 `docs`、`examples` 和 `vnpy`。如果还有其它目录，那些目录是在安装和运行过程中自动创建，不属于源代码。`docs` 目录中存放扩展名为 `.md` 的说明文档。`examples` 目录中存放一些示例程序，供学习研究用。`vn.py` 的程序都在 `vnpy` 子目录中。

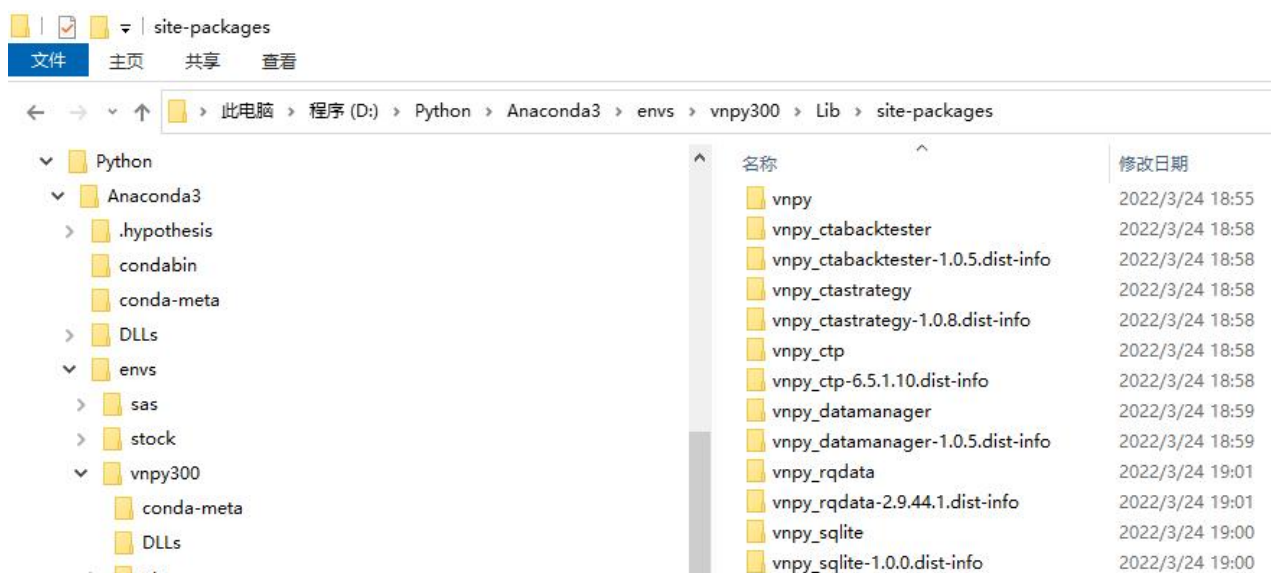
在 `vn.py` 的早期版本中，`vnpy` 目录下面有 7 个子目录，如下图所示。



其中，api 目录中存放真正的交易接口程序，gateway 目录中存入 vn.py 与具体接口的接口类程序，app 目录中存入所有的上层应用。

如前所述，“以前下载的 vn.py 源代码压缩包都是几十 M，从 2.9.0 开始只有几百 K。”vn.py 3.0.0 的源代码中，已经没有了 api、app 和 gateway 等 3 个目录，原来在这 3 个目录中的模块都已经做成了 vnpy_ 包，使用 pip install 命令安装。

那么，这些包安装在哪里呢？如果您按照教材推荐的方法安装 Anaconda，按照本文档推荐的方法创建虚拟环境，这些包的程序在 D:\Python\Anaconda3\envs\vnpy300\Lib\site-packages 目录中，如下图所示。



参考教材第 6.5 节内容，上述结构很容易理解。

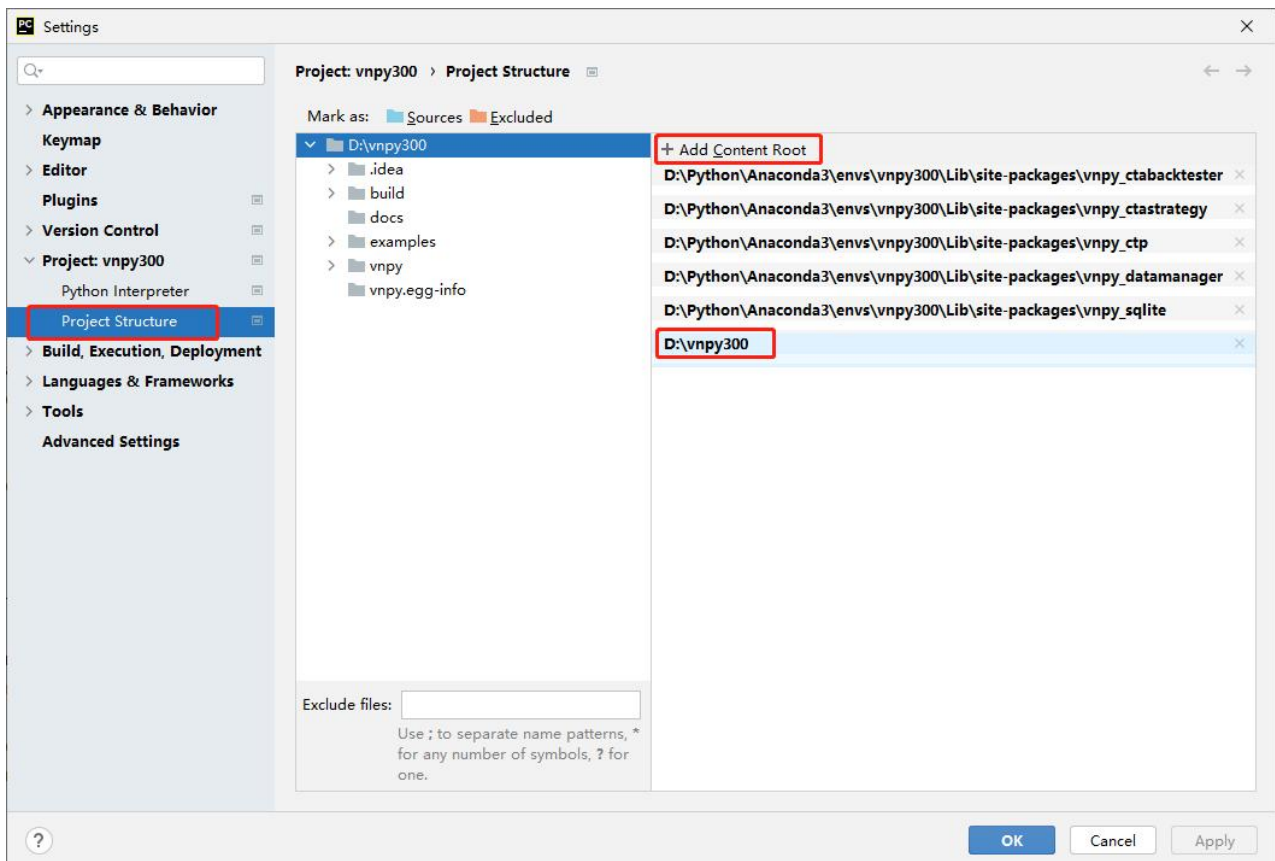
任意打开一个目录，比如 vnpy_ctabacktester，发现其中就是源程序，文件和目录结构与早期版本的 app 子目录下的模块完全相同，如下图所示。

ui	2022/3/24 18:58	文件夹	
init.py	2022/3/24 18:58	PY 文件	2 KB
engine.py	2022/3/24 18:58	PY 文件	14 KB

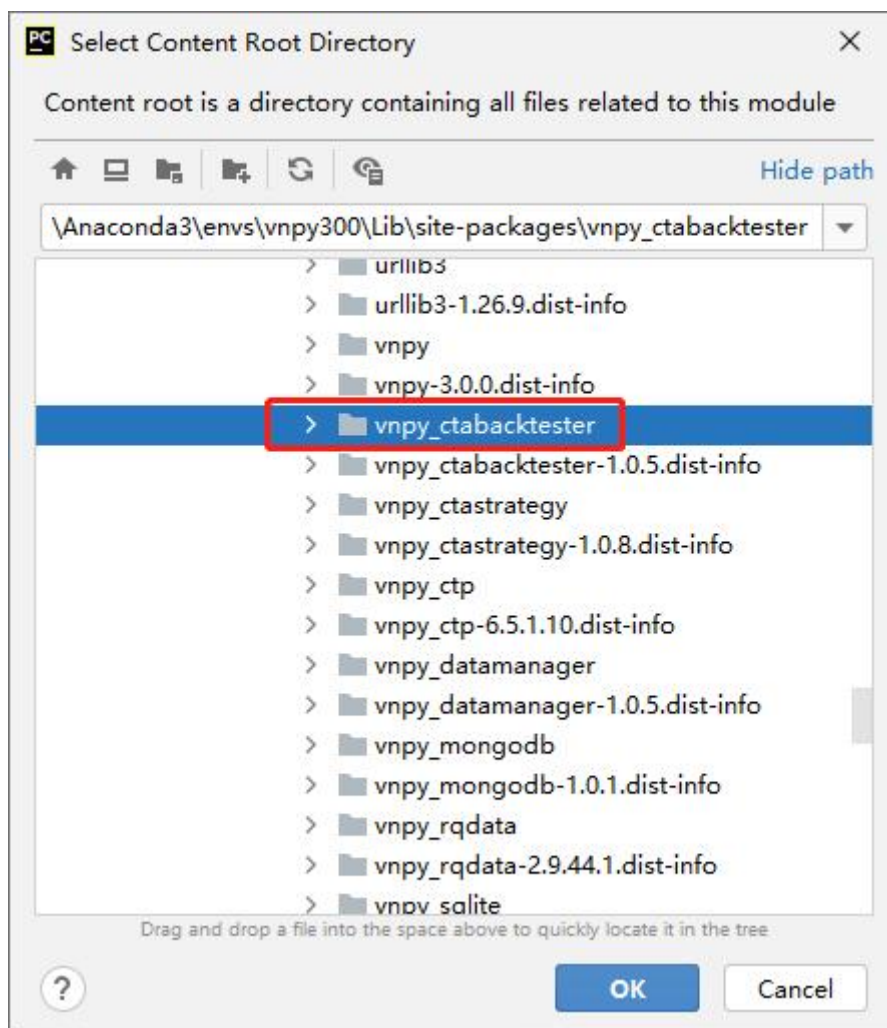
1.1.6. 使用 IDE（续）

在 PyCharm 中，用前述方法打开项目，只能看到从 zip 文件解压出来的源代码，不包含 vnpy_包中的代码。为了学习、分析方便，可用下述方法将 vnpy_包的源码加入到 PyCharm 项目中来。

在 PyCharm 的菜单中选择“File - Settings...”项，打开“Settings”界面，如下图所示。

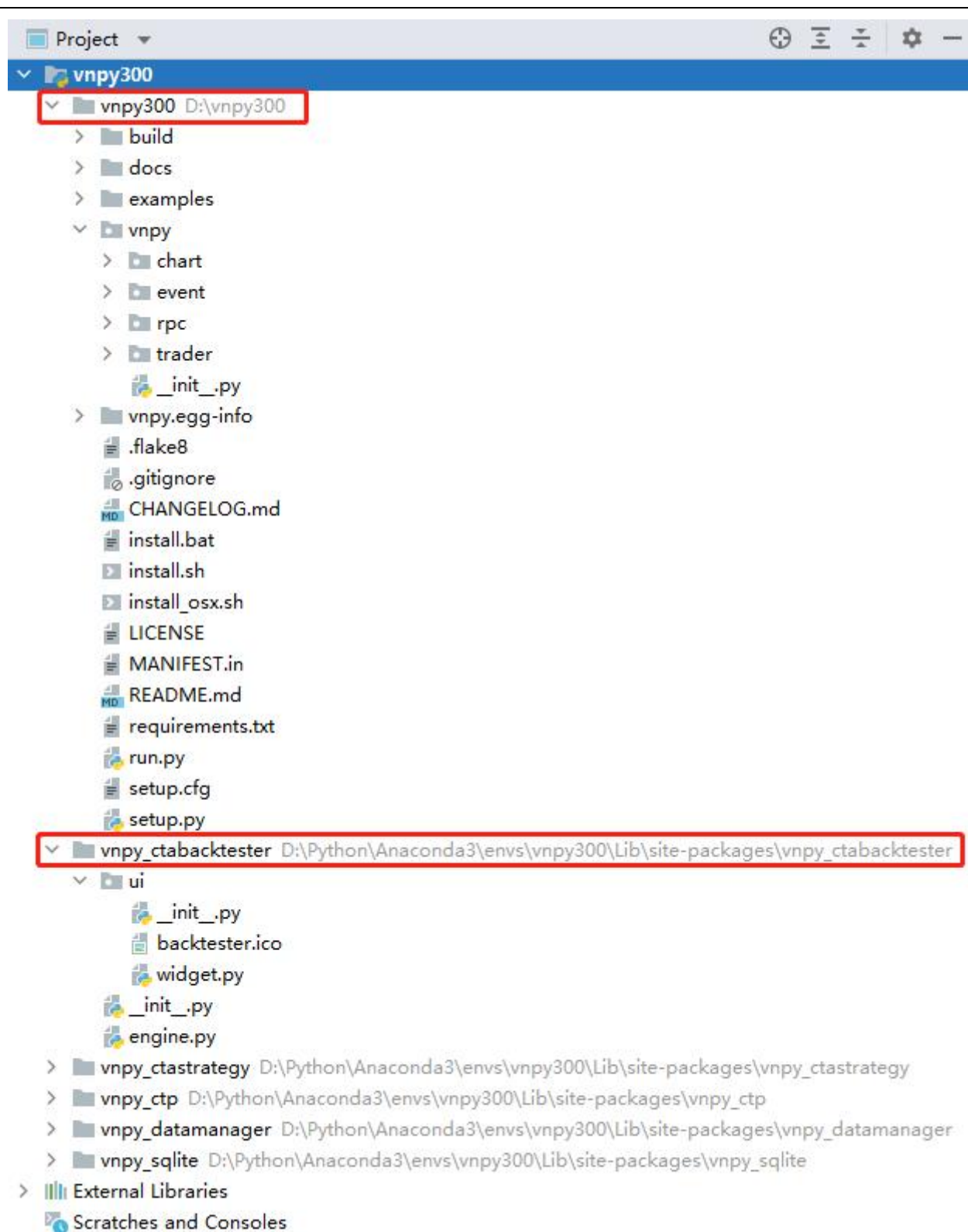


在“Settings”界面左侧树形列表中选择“Project Structure”，显示项目结构。此时右侧列表中应该已经有“D:\vnpy300”。单击右上的“Add Content Root”按钮，打开“Select Content Root Directory”对话框，如下图所示。



找到 D:\Python\Anaconda3\envs\vnpy300\Lib\site-packages\vnpy_ctabacktester 目录，按“OK”按钮，把 vnpy_ctabacktester 包加入到 PyCharm 项目当中来。

用同样的方法把其它几个 vnpy_包也加入到项目中来，最终结果如前图（“Settings” 界面）所示。退回到 PyCharm 主界面，在左侧的项目导航窗口中，可以同时看到 vn.py 主目录以及各 vnpy_包的文件，如下图所示。



经过上述配置，在 PyCharm 中同时分析 `vn.py` 程序及所有 `vnpy_` 包的程序非常方便。

因为每个人的 Python 环境各不相同，所以 `vnpy_` 包的存储路径也会各不相同。后续分析代码时，如果涉及 `vnpy_` 包中的代码，本文档只指明 `vnpy_` 包的名称，如 `vnpy_ctabacktester`，而不再指明具体的操作系统路径，读者使用 PyCharm 打开相关代码不会产生误解。

1.2. VeighNa Trader 的使用

本节是对教材 15.3 节内容的补充。本节最好先看教材，再看本文档内容。

1.2.1. 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。

1.2.2. 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

1.2.3. 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

1.3. CTA 回测

本节是对教材 15.4 节内容的补充。本节最好先看本文档（是针对新版本的说明），再看教材，教材的内容更全面。

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（无论是否用 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。

1.3.1. 配置数据服务

要跑历史数据回测，第一步就是要先准备好历史数据。

如前文所述，vn.py 支持多种数据服务。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 VeighNa Trader，在主界面顶部的菜单栏，选择“配置”菜单，打开 VeighNa Trader 的“全局配置”对话框。按下图所示的方式填写配置信息。

datafeed.name <str>	rqdata
datafeed.username <str>	license
datafeed.password <str>	RQData提供的token

填写完成后，单击对话框下部的“确定”按钮，弹出对话框，提示配置要重启系统之后才能生效。

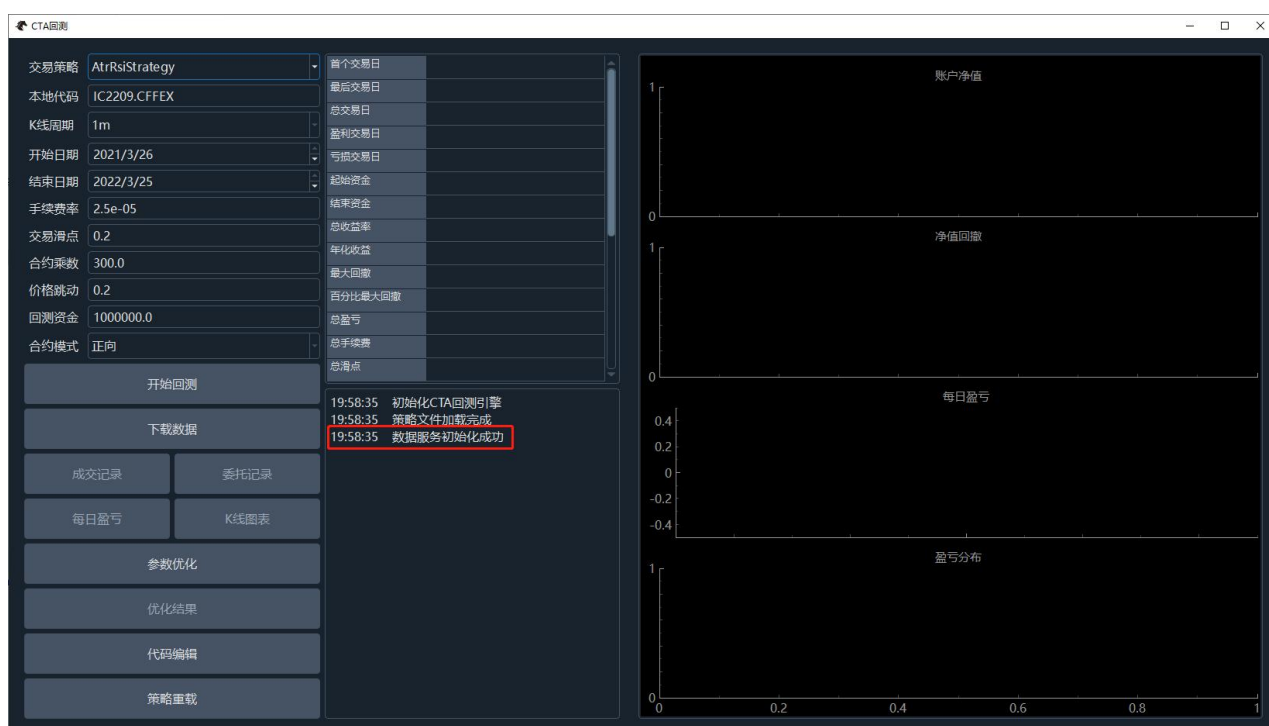
对于个人量化交易者，还有一个常用的数据服务是 tushare。如何注册 tushare 用户可参考教材的另一个附配文档《8-Tushare Pro 使用说明》。在“全局配置”对话框中，按下图所示的方式填写配置信息。

datafeed.name <str>	tushare
datafeed.username <str>	token
datafeed.password <str>	您的tushare token

RQData 要收费，tushare 要积分，免费的数据服务越来越难找了。

1.3.2. 执行回测

完成数据服务的全局配置之后，关闭 VeighNa Trader 并重启。重启后选择菜单“功能 - CTA 回测”，启动如下图所示的“CTA 回测”界面。



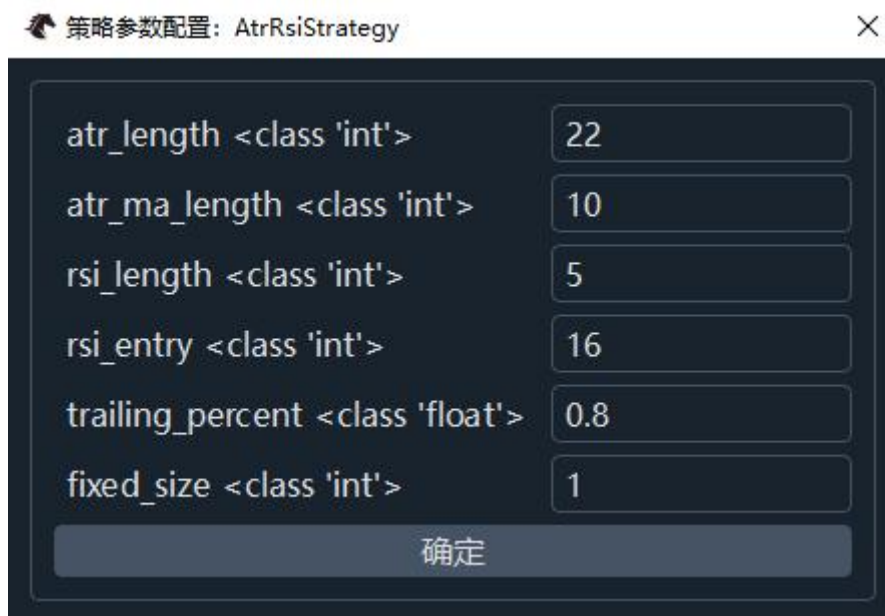
如果上一步的数据服务配置正确，此时可以在界面中间底部的日志输出框中看到“数据服务初始化

成功”的信息。如果没有看到就说明配置有问题，回去重来吧。

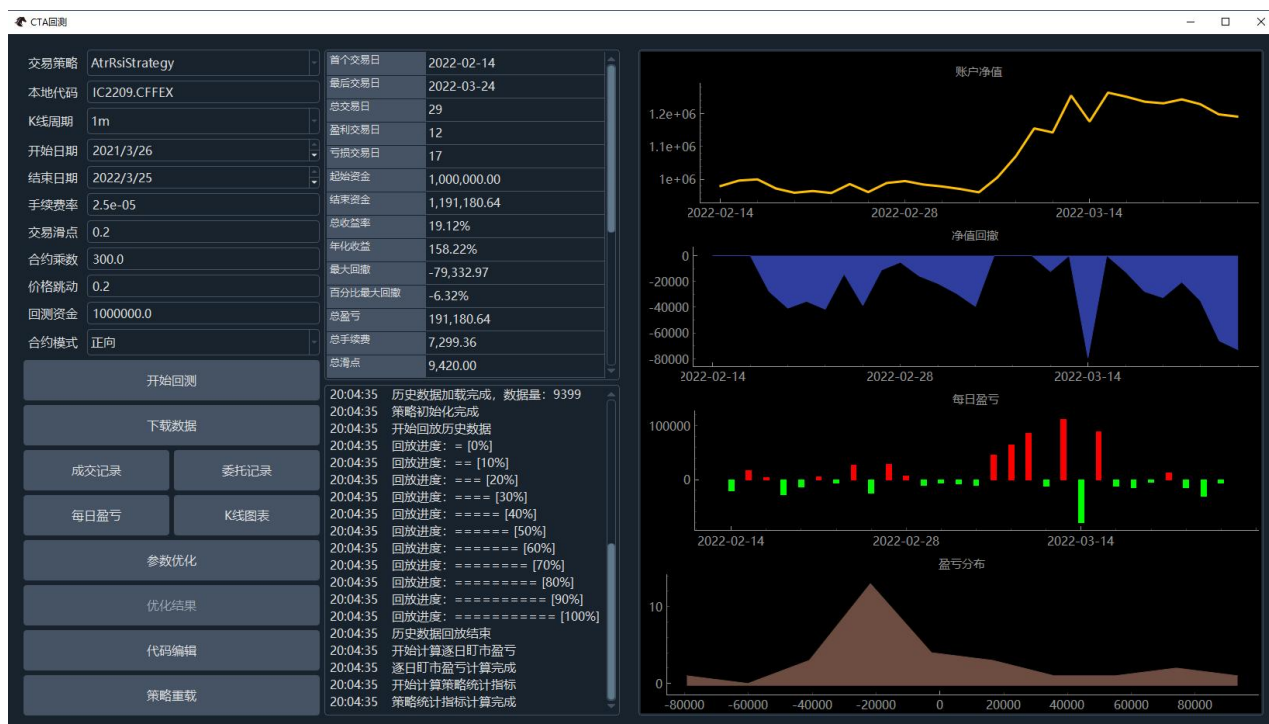
窗口左上方的一系列编辑框和下拉框，用来控制和管理回测功能。在本地代码编辑框中输入 IF2001.CFFEX, K 线周期选择 1m (即 1 分钟 K 线)，然后选择要下载数据的开始日期和结束日期，点击“下载数据”按钮。

此时 CtaBacktester 模块就会自动从所配置的数据服务器下载历史数据，在完成数据结构转化后插入到 VeighNa Trader 的数据库中。默认使用 SQLite，数据文件位于 vn.py 工作目录的 vntrader 子目录下的 database.db 文件中，“vn.py 工作目录”见教材 16.2.2 节。下载完成后同样会在日志输出框中看到相应信息。

按“开始回测”按钮，弹出对话框如下图所示。



接受默认的策略参数，按“确定”按钮开始回测，回测结果如下图所示。



1.4. CTA 策略

详见教材 15.5 节。

第 2 章 VeighNa 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 VeighNa Trader 开始分析 vn.py 的源代码了。

本章是对 VeighNa 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 VeighNa 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。

2.1. 程序主函数

详见教材 15.5 节。

2.2. 主引擎

详见教材 16.2 节。

2.2.1. 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，如下图所示。

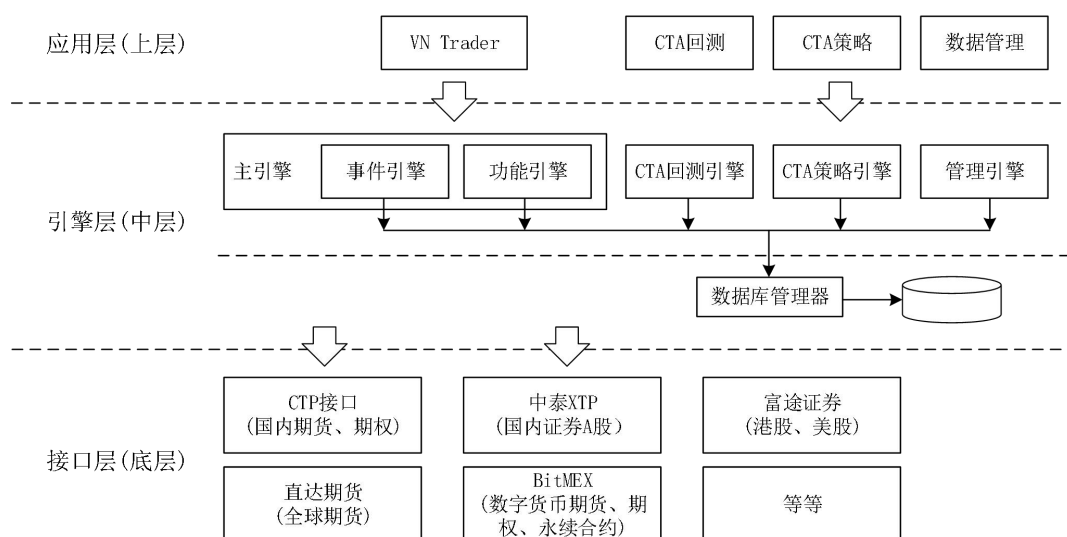


图 vn.py 的体系结构

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:\vnpy300\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 VeighNa 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。

2.2.2. 初始化函数

详见教材。

2.2.3. 初始化功能引擎

详见教材。

2.2.4. 增加功能引擎

详见教材。

2.2.5. 增加底层接口

注：再次说明，代码以及正文中的红色部分，有的是重点强调的部分，有的是与教材不同的部分，也就是 vn.py 3.0.0 与 vn.py 2.1.4 不同的部分。读者可与教材参照学习。

事实上，分析这些部分，不但可以了解版本之间的差异，还可以通过分析，理解开发者为何这么做，从而增强学习效果，更快地提高编程水平。

增加底层接口的代码为

```
def add_gateway(self, gateway_class: Type[BaseGateway], gateway_name: str = "") \
    -> BaseGateway:                                # ①
    # 如果没有传入 gateway_name，则使用默认名称
    if not gateway_name:
        gateway_name = gateway_class.default_name

    gateway = gateway_class(self.event_engine, gateway_name)    # ②
    self.gateways[gateway_name] = gateway                      # ③

    # ④将该底层接口支持的交易所加入到交易所列表中
    for exchange in gateway.exchanges:
        if exchange not in self.exchanges:
            self.exchanges.append(exchange)
```

```
return gateway
```

```
# ⑤
```

①：传入的参数是底层接口类名称。

②：根据类名称，创建一个底层接口实例。

③：将新创建的接口实例加入到底层接口字典。

④：将该底层接口支持的交易所加入到交易所列表中。主引擎的底层接口字典和交易所列表都通过本方法建立。

⑤：返回创建的底层接口实例。

新版本此方法增加了一个参数 `gateway_name`。加了此参数，在功能上就允许同类接口以不同的名字加载多次，可以用同样的接口连接不同的交易商。该参数有默认值，相关程序不需要修改即可保持原有功能。

2.2.6. 增加上层应用

详见教材。

2.3. 主界面

详见教材 16.3 节。

对系统的体系结构及主引擎有所了解之后，本节分析 VN Trader 的主界面编程。

2.3.1. 创建应用程序

主函数中调用 `create_qapp()` 函数创建应用程序，`create_qapp()` 函数在 `D:\vnpy300\vnpy\trader\ui` 的 `qt.py` 中定义。`qt.py` 有三个作用：

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

2-作为 GUI 模块导入的中间模块。在本模块中包括语句：

```
from PySide6 import QtGui, QtWidgets, QtCore
```

可以看出，使用的是 PySide6 图形界面包，但导入的类名与 PyQt5 完全相同。有这样一种说法，将 PyQt5 程序移植到 PySide6，只需要将导入部分的 PyQt5 改为 PySide6 就可以了。其实不完全是这样，但在大部分情况下是可以的。教材中关于 PyQt5 的讲解只涉及 PyQt5 的核心部分，在 PySide6 中有同样的实现，教材的内容仍然适用。

在本模块中导入了 PySide6 类后，在其它模块中就可以使用如下的方式从本模块导入（见本模块下的其它程序）：

```
from .qt import QtCore, QtGui, QtWidgets
```

这样就将本模块作为了其它 GUI 模块的导入中间模块，增加了以后程序移植的方便性。

3-定义 `create_qapp()` 函数。`create_qapp()` 的功能是创建一个应用程序对象，设置其图标、字体等。其核心代码是：

```
qapp = QtWidgets.QApplication([])
```

可见创建的是一个 PySide6 应用程序。

2.3.2. 主窗口的初始化

与前期版本只有一处不同：

```
self.window_title: str = f"VeighNa Trader 社区版 - {vnpy.__version__} [{TRADER_DIR}]"
```

将主 GUI 界面 VeighNa Trader 的名称改为“Veighna Trader 社区版”。

2.4. 窗口组件

详见教材 16.4 节。

2.4.1. 单元格类

详见教材。

2.4.2. 监控组件类

新版本在监控组件基类中增加了配置项的保存和加载功能：

```
def __init__(self, main_engine: MainEngine, event_engine: EventEngine):
    """

    super(BaseMonitor, self).__init__()

    self.main_engine: MainEngine = main_engine
    self.event_engine: EventEngine = event_engine
    self.cells: Dict[str, dict] = {}

    self.init_ui()
    self.load_setting()
    self.register_event()
```

其效果是：您对监控组件的调整，比如调整了某列的列宽，下次启动程序时可以重现。

2.4.3. 初始化悬浮窗口

详见教材。

2.5. 菜单

详见教材 16.5 节。

2.5.1. 底层接口加入菜单

用于添加菜单项的方法改名为 add_action：

```
def add_action(
    self,
    menu: QtWidgets.QMenu,
    action_name: str,
    icon_name: str,
    func: Callable,
    toolbar: bool = False
) -> None:
    """
```

```
icon = QtGui.QIcon(icon_name)

action = QtWidgets.QAction(action_name, self)
action.triggered.connect(func)
action.setIcon(icon)

menu.addAction(action)

if toolbar:
    self.toolbar.addAction(action)
```

该方法增加了一个布尔型参数 `toolbar`，该参数确定是否将创建的行为对象同时加入到工具栏。

2.5.2. 上层应用加入菜单

与教材的不同是，CTA 回测的应用类在 `vnpy_ctabacktester` 包的 `__init__.py` 中定义。

在创建“功能”菜单部分，新版本的相同之处是：

```
for app in all_apps:
    ui_module = import_module(app.app_module + ".ui")
    widget_class = getattr(ui_module, app.widget_name)

    func = partial(self.open_widget, widget_class, app.app_name)

    self.add_action(app_menu, app.display_name, app.icon_name, func, True)
```

使用参数 `toolbar=True` 调用方法 `add_action`，就不再需要像早期版本那样专门调用 `add_toolbar_action` 方法，即可将行为对象同时加入到工具栏。

第 3 章 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》。

`vn.py` 从 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。

（以下为复制、改进的教材内容）

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。

3.1. 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。vn.py 3.0.0 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 8 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。
 - DolphinDB：DolphinDB 是由浙江智奥科技有限公司研发的一款高性能分布式时序数据库，特别适用于对速度要求极高的低延时或实时性任务，在国内外金融投资领域有着丰富的应用案例。
 - Arctic：由英国量化对冲基金 Man AHL 基于 MongoDB 开发的高性能金融时序数据库，支持直接存储 pandas 的 DataFrame 和 numpy 的 ndarray 对象，在量化投研中非常实用。
 - LevelDB：由 Google 推出的高性能 Key/Value 数据库，基于 LSM 算法实现进程内存储引擎，支持数十亿级别的海量数据。LevelDB 的定位是通用性数据存储方案，对于金融领域的时序数据存储没有特别大的优势，但也比一般 SQL 类数据库要多。

先看 vn.py 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。

database.timezone <str>	Asia/Shanghai
database.name <str>	sqlite
database.database <str>	database.db
database.host <str>	
database.port <int>	
database.user <str>	
database.password <str>	

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。详细的配置方法见 vn.py 的官方文档 <https://www.vnpy.com/docs/cn/database.html>。

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,  
    turnover      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。

	id	symbol	exchange	datetime	interval	volume	turnover	open_inte	open_pric	high_pric	low_price	close_pri
1	1	IC2209	CFFEX	2022-01-24 00:00:00	d	9945	1355800.652	9816	6829.6	6844.2	6770	6807
2	2	IC2209	CFFEX	2022-01-25 00:00:00	d	11626	1557411.932	17384	6799	6818	6584.8	6588.4
3	3	IC2209	CFFEX	2022-01-26 00:00:00	d	10061	1327283.696	21938	6613.2	6633	6552.2	6586.6
4	4	IC2209	CFFEX	2022-01-27 00:00:00	d	10978	1421569.784	26069	6581.8	6586.4	6391.4	6420
5	5	IC2209	CFFEX	2022-01-28 00:00:00	d	10805	1376719.152	29264	6480	6480	6273.2	6335
6	6	IC2209	CFFEX	2022-02-07 00:00:00	d	6914	889858.008	31481	6392.8	6462.6	6392.8	6428.4
7	7	IC2209	CFFEX	2022-02-08 00:00:00	d	6971	898097.56	32572	6430	6522	6372.2	6517
8	8	IC2209	CFFEX	2022-02-09 00:00:00	d	5230	686979.912	32605	6519.6	6615	6507.2	6605.2
9	9	IC2209	CFFEX	2022-02-10 00:00:00	d	4858	641166.656	33804	6609	6625	6570.8	6609.2
10	10	IC2209	CFFEX	2022-02-11 00:00:00	d	6379	839564.904	35476	6590	6618	6535.8	6554.8

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。

	id	symbol	exchange	interval	count	start	end
1	1	IC2209	CFFEX	d	39	2022-01-24 00:00:00	2022-03-24 00:00:00
2	2	IC2206	CFFEX	d	38	2022-01-26 00:00:00	2022-03-25 00:00:00
3	3	IC2209	CFFEX	1m	9399	2022-01-24 09:29:00	2022-03-24 14:59:00

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,
    turnover    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,
        localtime       DATETIME
    );

```

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。

3.2. 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。

vn.py 的数据管理功能允许从 CSV 文件导入数据，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，本章的后续内容将会讨论如何将 CSV 文件中的数据导入到数据库。

3.2.1. 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。

3.2.2. 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，就能用 JQData 替换 rqdata。

3.2.3. 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 - 缠论分析/训练软件(2.1)”从通达信中获取数据并转换成 CSV 文件。

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>)”中对此问题有论证，总之 vn.py 建议大家使用 rqdata。

3.3. 数据管理（基础代码）

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。

第一部分：vnpy.trader.database 模块

在 D:\vnpy300\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：具体的数据库包，如 vnpy_sqlite

每类数据库都有其对应的数据库包，如 SQLite 数据库对应的包是 vnpy_sqlite，其源代码在 D:\Python\Anaconda3\envs\vnpy300\Lib\site-packages\vnpy_sqlite 目录中（与 vnpy_包相关的内容详见“vn.py 的安装与运行”一节）。

再例如，与 MongoDB 数据库对应的包是 vnpy_mongodb。

在每个数据库包中，都会定义具体的数据库类，如 SQLite 数据库的类是 SqliteDatabase，再如 MongoDB 数据库的类是 MongodbDatabase。

如果是 SQL 数据库，因为要使用 peewee（关于 peewee 的详细说明见教材 9.5 节），还会定义与三个表相对应的模板类。相关模块、文件和类及其继承关系如下图所示。

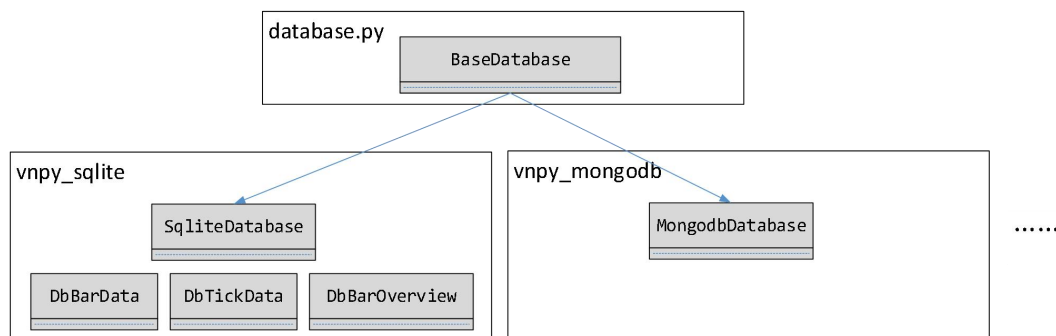


图 17-1 数据库管理器类

第三部分：vnpy_datamanager 包

vnpy_datamanager 包实现“数据管理”应用的 GUI 界面。

3.3.1. 数据库管理器基类

本小节对应前述第一部分代码。

数据库管理器的基类 BaseDatabase 在 D:\vnpy300\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
from pytz import timezone

# 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，是所有数据库管理器类（如 SqliteDatabase、MongodbDatabase 等）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

# 创建一个全局数据库实例变量
database: BaseDatabase = None

def get_database() -> BaseDatabase:
    """得到全局数据库实例"""
```

```

# 如果已经创建，直接返回全局数据库实例
global database

if database:
    return database

# 如果还没有创建
# 读取与数据库相关的全局配置项
database_name: str = SETTINGS["database.name"]
module_name: str = f"vnpy_{database_name}"

# 尝试导入具体的数据库模块
try:
    module = import_module(module_name)
except ModuleNotFoundError:
    # 如果导入具体的数据库模块失败，则导入默认的 SQLite 数据库模块
    print(f"找不到数据库驱动 {module_name}，使用默认的 SQLite 数据库")
    module = import_module("vnpy_sqlite")

# 从数据库模块中创建全局数据库实例，并返回
database = module.Database()

return database

```

3.3.2. vnpy_sqlite 包

本小节对应前述第二部分代码。

__init__.py 文件

__init__.py 文件的作用见教材 6.2 节。

BaseDatabase 是抽象类，所有数据库类都从该抽象类继承。每类数据库都有其对应的数据库包，如 SQLite 数据库对应的包是 vnpy_sqlite，其源代码在 D:\Python\Anaconda3\envs\vnpy300\Lib\site-packages\vnpy_sqlite 目录中（与 vnpy_包相关的内容详见“vn.py 的安装与运行”一节）。

在每个数据库包中，都会定义具体的数据库类，如 SQLite 数据库的类是 SqliteDatabase（继承自 BaseDatabase）。

在 vnpy_sqlite 包的 __init__.py 文件中有如下定义：

```
from .sqlite_database import SqliteDatabase as Database
```

无论是哪类数据库，对外都会重命名为 Database。可样就可以理解，上一节 BaseDatabase 类的代码：

```
database = module.Database()
```

是创建一个具体数据库类的实例，并返回。

peewee 模板类

与 SQLite 数据库相关的定义在 `vnpy_sqlite` 包中的 `sqlite_database.py` 中，用于操作 SQLite 数据库。该文件的定义分为两部分，第一部分定义 peewee 模板类，第二部分定义 SQLite 数据库类。

`vn.py` 在操作 SQL 数据库时使用 peewee，包括 SQLite、MySQL 和 PostgreSQL 等，都需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。
# 注意在本程序开头的 import 语句中已将 SqliteDatabase 函数重命名为 PeeweeSqliteDatabase。
```

```
class DbBarData(Model):
    """K 线数据表映射对象"""

    id = AutoField()

    symbol: str = CharField()
    exchange: str = CharField()
    datetime: datetime = DateTimeField()
    interval: str = CharField()

    volume: float = FloatField()
    turnover: 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 数据表映射对象"""
```

```
id = AutoField()

symbol: str = CharField()
exchange: str = CharField()
datetime: datetime = DateTimeField()

name: str = CharField()
volume: float = FloatField()
turnover: 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)

localtime: datetime = DateTimeField(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 线汇总数据表映射对象"""

    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，或本文档 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
    """
    .....

```

3.4. 数据管理（应用）

本节对应前述第三部分代码。

数据管理是 vn.py 的一个上层应用，是一个综合的数据管理平台。启动 VeighNa 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，界面如下图所示。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。

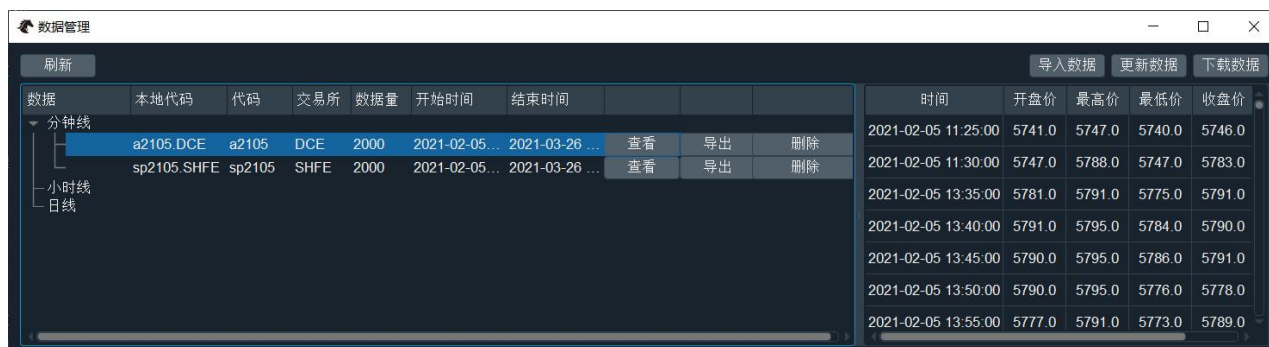


图 “数据管理”上层应用

数据管理应用的体系结构如下图所示，该结构其实就是本文档“vn.py 体系结构”一节所示 vn.py 体系结构的局部。

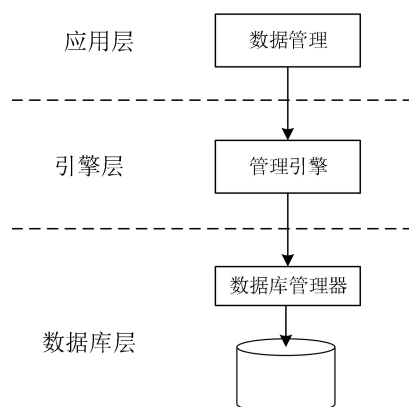


图 数据管理的体系结构

上图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。数据库层的数据库类在教材 17.2 节（或本文档 3.3 节）已经介绍，本节介绍应用层和引擎层的实现方法。

3.4.1. 数据管理应用类

请读者先复习教材 16.5 节（本文档 2.5 节）内容，回顾 VeighNa Trader 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 vnpy_datamanager 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。

VeighNa 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)
```

3.4.2. 数据管理窗体类

“数据管理”的窗体类 ManagerWidget 在 vnpy_datamanager 包 ui 子目录下的 widget.py 中定义。

界面的左侧是一个树型的行情分类列表，编程方法与教材 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 PySide6 代码，组织方式比主窗口简单，不再详述。掌握《Python 量化交易从入门到实战》一书中相关内容的读者，看这部分代码不会有问题。

3.4.3. 下载数据

按“下载数据”按钮可从数据服务下载行情数据。



如本文档 1.3 节所述，`vn.py` 默认的行情数据服务是米筐的 `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。

3.4.4. 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，界面如下图所示。

从CSV文件导入数据

选择文件

合约信息

代码

交易所

周期

时区

CFFEX

MINUTE

Asia/Shanghai

表头信息

时间戳

开盘价

最高价

最低价

收盘价

成交量

成交额

持仓量

datetime

open

high

low

close

volume

turnover

open_interest

格式信息

时间格式

%Y-%m-%d %H:%M:%S

确定

图 从 CSV 文件导入数据

在上图界面上，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 只关心上图中列出的字段，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，上图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()
    tz_name = dialog.tz_combo.currentText()
    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()
    turnover_head = dialog.turnover_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,
        tz_name,
        datetime_head,
        open_head,
        high_head,
        low_head,
        close_head,
        volume_head,
        turnover_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 `vnpy_datamanager` 包的 `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,
    tz_name: str,
    datetime_head: str,
    open_head: str,
    high_head: str,
    low_head: str,
    close_head: str,
    volume_head: str,
    turnover_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

# 时区
tz = timezone(tz_name)

# 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])

    # 进行时区转换
    dt = tz.localize(dt)

    # 成交额
    turnover = item.get(turnover_head, 0)

    # 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]),
    turnover=float(turnover),
    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，本文档不再赘述。

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条数据中读取主键参数
    bar = bars[0]

    symbol = bar.symbol          # 合约代码
    exchange = bar.exchange      # 交易所
    interval = bar.interval      # K线周期

    # 将BarData数据转换为字典，并调整时区
    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)

```

```
# 使用 upsert 操作将数据更新到数据库中
# 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 线汇总数据
# 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汇总数据，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汇总数据
    # 创建一个新的汇总数据对象
    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汇总数据
    # 对汇总数据进行更新
    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()


# 保存汇总数据
overview.save()
```

```
return True
```

除了保存 K 线行情数据外，还需要对汇总数据进行更新。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。

3.5. 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按钮，启动“CTA 回测”上层应用，其操作说明见教材 15.4 节以及本文档 1.3 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 `vnpy_ctabacktester` 包的 `ui` 子目录下的 `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，将在教材第 18 章（本文档第 4 章）讨论，现在暂时忽略。

在 `vnpy_ctastrategy` 包的 `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:
    data = load_tick_data(
        self.symbol,
        self.exchange,
        start,
        end
    )

    self.history_data.extend(data)

    progress += progress_days / total_days
    progress = min(progress, 1)

    start = end + interval_delta
    end += progress_delta

```

```

self.output(f"历史数据加载完成，数据量: {len(self.history_data)}")

```

- 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
    bars: List[BarData] = []
    for db_bar in s:
        bar = BarData(
            symbol=db_bar.symbol,
            exchange=Exchange(db_bar.exchange),
            datetime=datetime.fromtimestamp(db_bar.datetime.timestamp(), DB_TZ),
            interval=Interval(db_bar.interval),
            volume=db_bar.volume,
            turnover=db_bar.turnover,
            open_interest=db_bar.open_interest,
            open_price=db_bar.open_price,
            high_price=db_bar.high_price,
            low_price=db_bar.low_price,
            close_price=db_bar.close_price,
            gateway_name="DB"
        )
        bars.append(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。

第 4 章 CTA 回测

对主界面有了基本了解，就可以从一个具体功能入手，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。

为了方便教学，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 3.0.0 所补充的内容。

4.1. 事件引擎

详见教材 18.1 节。

4.2. 回测线程

对应教材 18.2 节。

4.2.1. 类结构

本小节看教材。

4.2.2. 执行流程

本小节看本文档。

本小节只是按照调用关系简单分析 CTA 回测的执行流程，详细的执行代码见教材 18.3 节或本文档 4.3 节。

1. BacktesterManager 窗体类

BacktesterManager 是回测窗体类，在 vnpy_ctabacktester 包的 ui 目录下的 widget.py 中定义。它有一个属性 backtester_engine，指向一个 BacktesterEngine 类实例。在 BacktesterManager 的“开始回测”按钮事件处理函数 start_backtesting() 中，调用 backtester_engine.start_backtesting()。

2. BacktesterEngine 回测引擎类

BacktesterEngine 是回测引擎类，在 vnpy_ctabacktester 包的 engine.py 中定义。它有一个成员变量 backtesting_engine，指向一个 BacktestingEngine 类实例。还有一个属性 thread 是回测线程，该属性平时值为 None；当有回测任务需要执行时，创建并启动回测线程，将回测线程的引用保存到 thread 属性中；线程执行结束时，将该指针重新置为 None。

当 start_backtesting 方法被调用时做三项工作：

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

2-以 run_backtesting() 函数为线程函数创建线程，并将 thread 指向该线程。

3-启动线程。

在线程函数 `run_backtesting()` 中执行以下操作：

1-传递回测及策略参数。

2-调用 `backtesting_engine` 的 `run_backtesting()` 函数，执行真正的回测操作。

3-回测结束，将 `thread` 置为 `None`，允许新的回测线程执行。

4-推送回测结束事件。

其中第 3 步与线程控制有关。

3. BacktestingEngine 回测执行引擎类

`BacktestingEngine` 是回测执行引擎类，在 `vnpy_ctastrategy` 包的 `backtesting.py` 中定义。其成员函数 `run_backtesting()` 执行真正的回测操作。

4.2.3. 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。

4.3. 回测执行

对应教材 18.3 节。

4.3.1. “开始回测”按钮的槽函数

本小节看教材。

4.3.2. 18.3.2 回测操作

本小节看本文档。

真正的回测操作在回测线程中执行。`BacktesterEngine` 是回测引擎类，在 `vnpy_ctabacktester` 包的 `engine.py` 中定义。

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

本小节教材与文档对照学习。

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 vnpy_ctabacktester 包的 ui 目录下的 widget.py 中定义。在窗体类中首先定义一个 PySide6 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，该函数包含大量界面操作，不能在 Qt 事件循环以外的线程中直接调用。

第二部分 VeighNa 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 VeighNa Trader 的代码进行了分析，那时主要针对 VeighNa 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 VeighNa Trader 交易平台。

第 5 章 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。

5.1. 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。

5.1.1. 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。

5.1.2. 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，可以到《Python 量化交易从入门到实战》一书的资源下载群下载。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 PySide，可以互相参照，便于 vn.py 的学习

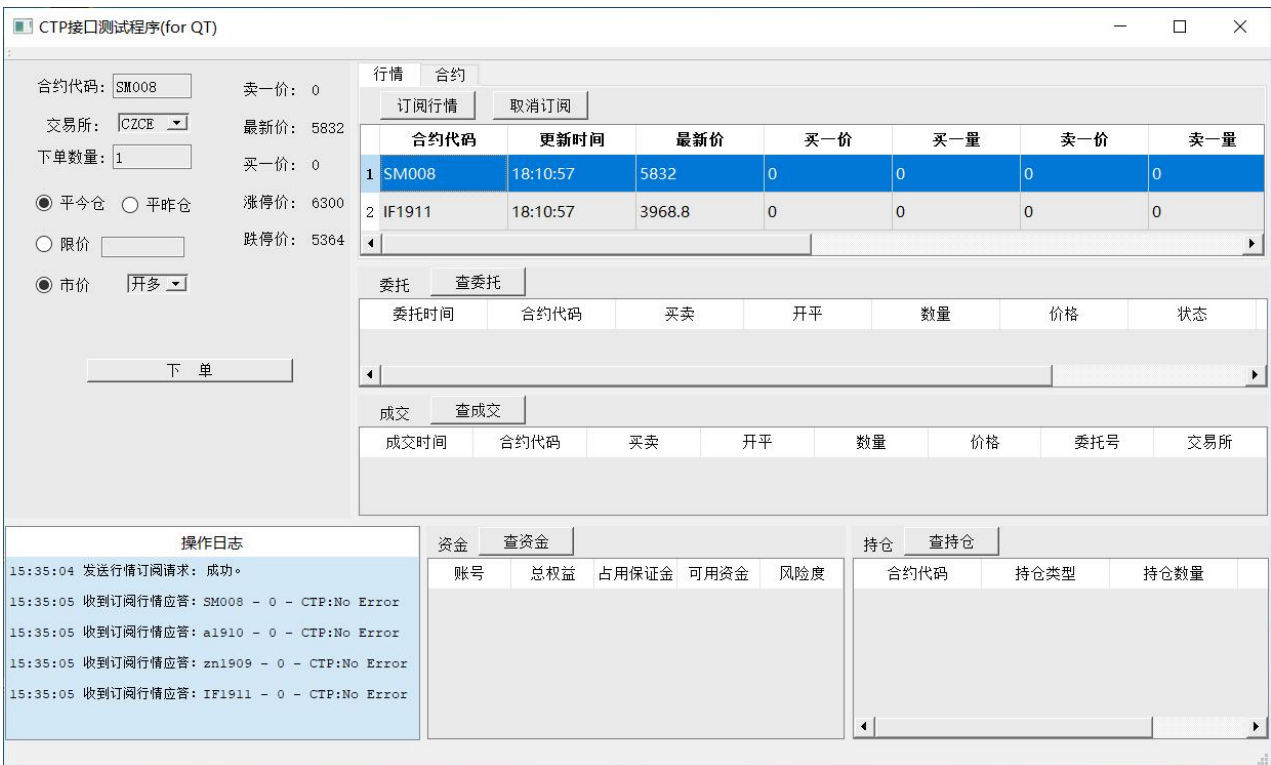
程序的登录界面如下。



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 VeighNa 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。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 "CThostFtdcMdApi.h"

typedef struct
{
    char FRONT_ADDR[100];           // 前置地址
    CThostFtdcBrokerIDType 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。

5.1.3. 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对的。如果有阅读此文档的老师确定我是错的，请告诉我。

5.1.4. 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，确认证认证成功

- 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。

5.2. 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 ctapi 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 vnpy_ctp 包中 api 目录下的三个文件。

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。

vnctpm.d.cp310-win_amd64.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。

注：在上述文件名中，中间部分 cp310-win_amd64 与操作系统以及 Python 版本有关，在执行 pip install 命令时由系统自动区分，在读者的电脑中可能会有所不同。

vnctptd.cp310-win_amd64.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 vnpy_ctp 包的 __init__.py 文件中：

```
from .gateway import CtpGateway
```

将上述封装的成果集成到模块中。

第 6 章 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。

6.1. VeighNa Trader 的全局定义

VeighNa 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:\vnpy300\vnpy\trader 目录下有三个文件：

- event.py 定义 VeighNa 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 VeighNa 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 VeighNa 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。

基于由浅入深的原则，本节只做提示，更详细的说明在本文档第四部分。

6.2. 底层接口基类 BaseGateway

在 D:\vnpy300\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、报价数据、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, 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,
```

```
    EVENT_QUOTE,
```

```
)
```

```
from .object import (
```

```
    TickData,
```

```
    OrderData,
```

```
    TradeData,
```

```
    PositionData,
```

```
    AccountData,
```

```
    ContractData,
```

```
    LogData,
```

```
    QuoteData,
```

```
    OrderRequest,
```

```
    CancelRequest,
```

```
    SubscribeRequest,
```

```
    HistoryRequest,
```

```
    QuoteRequest,
```

```
    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

```
"""
```

```
# 交易接口的默认名称
default_name: str = ""
```

```
# 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
    """
    self.on_event(EVENT_QUOTE, quote)
    self.on_event(EVENT_QUOTE + quote.vt_symbol, quote)

def on_log(self, log: LogData) -> None:
    """
    日志事件推送
    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:
    """
    从交易接口发起，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:
```

```
"""

开始连接接口

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:
    """
    关闭交易接口连接
    """
    pass


@abstractmethod
def subscribe(self, req: SubscribeRequest) -> None:
    """
    订阅行情（订阅 Tick 行情数据）
    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 创建的字符串类型的 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_quote(self, req: QuoteRequest) -> str:
        """
        发送一个新的双向询价单到服务器

        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
    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。

6.3. CTP 接口类 CtpGateway

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

CtpGateway 类在 vnpy_ctp 包的 gateway 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 ..api import (
    MdApi,
    TdApi,
    THOST_FTDC_OAS_Submitted,
    THOST_FTDC_OAS_Accepted,
    THOST_FTDC_OAS_Rejected,
    .....
)
```

从 vnpy_ctp 包的 api 目录的 __init__.py 文件中导入 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:\vnpy300\vnpy\trader 目录下的 constant.py 文件中定义，详见本文档 16.1 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 vnpy_ctp 包的 api 目录的 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:\vnpy300\vnpy\trader 目录下的 constant.py 中定义了 VeighNa 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

# 委托状态映射
STATUS_CTP2VT: Dict[str, Status] = {
    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
}

# 多空方向映射
DIRECTION_VT2CTP: Dict[Direction, str] = {
    Direction.LONG: THOST_FTDC_D_Buy,
    Direction.SHORT: THOST_FTDC_D_Sell
}

DIRECTION_CTP2VT: Dict[str, Direction] = {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
```

```
# 委托类型映射
```

```
ORDERTYPE_VT2CTP: Dict[OrderType, Tuple] = {
    OrderType.LIMIT: (THOST_FTDC_OPT_LimitPrice, THOST_FTDC_TC_GFD, THOST_FTDC_VC_AV),
    OrderType.MARKET: (THOST_FTDC_OPT_AnyPrice, THOST_FTDC_TC_GFD, THOST_FTDC_VC_AV),
    OrderType.FAK: (THOST_FTDC_OPT_LimitPrice, THOST_FTDC_TC_IOC, THOST_FTDC_VC_AV),
    OrderType.FOK: (THOST_FTDC_OPT_LimitPrice, THOST_FTDC_TC_IOC, THOST_FTDC_VC_CV),
}

ORDERTYPE_CTP2VT: Dict[Tuple, OrderType] = {v: k for k, v in ORDERTYPE_VT2CTP.items()}
```

```
# 开平方向映射
```

```
OFFSET_VT2CTP: Dict[Offset, str] = {
    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: Dict[str, Offset] = {v: k for k, v in OFFSET_VT2CTP.items()}
```

```
# 交易所映射
```

```
EXCHANGE_CTP2VT: Dict[str, Exchange] = {
    "CFFEX": Exchange.CFFEX,
    "SHFE": Exchange.SHFE,
    "CZCE": Exchange.CZCE,
    "DCE": Exchange.DCE,
    "INE": Exchange.INE
}
```

在 D:\vnpy300\vnpy\trader 目录下的 constant.py 中定义了 VeighNa Trader 支持所有交易所，包括中国和国际的

```
# CTP 只支持上述交易所
```

```
# 产品类型映射
```

```
PRODUCT_CTP2VT: Dict[str, Product] = {
    THOST_FTDC_PC_Futures: Product.FUTURES,
    THOST_FTDC_PC_Options: Product.OPTION,
    THOST_FTDC_PC_SpotOption: Product.OPTION,
    THOST_FTDC_PC_Combination: Product.SPREAD
}
```

```
# 期权类型映射
OPTIONTYPE_CTP2VT: Dict[str, OptionType] = {
    THOST_FTDC_CP_CallOptions: OptionType.CALL,
    THOST_FTDC_CP_PutOptions: OptionType.PUT
}

# 其他常量
MAX_FLOAT = sys.float_info.max          # 浮点数极限值
CHINA_TZ = pytz.timezone("Asia/Shanghai")  # 中国时区
```

```
# 合约数据全局缓存字典
symbol_contract_map: Dict[str, ContractData] = {}
```

下面介绍 CtpGateway 类的初始化部分，代码如下：

```
class CtpGateway(BaseGateway):
    """
    vn.py 用于对接期货 CTP 柜台的交易接口。
    """

    default_name: str = "CTP"

    # 默认配置字典，key 是配置字段名，value 是默认值
    default_setting: Dict[str, str] = {
        "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": ""
    }

    # 交易所列表
    exchanges: List[str] = list(EXCHANGE_CTP2VT.values())

    def __init__(self, event_engine: EventEngine, gateway_name: str) -> None:
        """构造函数"""
        super().__init__(event_engine, gateway_name)

        self.td_api: "CtpTdApi" = CtpTdApi(self)
        self.md_api: "CtpMdApi" = 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: CtpGateway) -> None:
    """构造函数"""
    .....
    # 接口的连接状态。True-已连接；False-未连接
    self.connect_status: bool = False
    # 接口的登录状态。True-已登录；False-未登录
    self.login_status: bool = False
    # 已订阅合约集合
    self.subscribed: Set = set()
    .....

def onFrontConnected(self) -> None:
    """服务器连接成功回报"""
    # 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) -> None:
    """服务器连接断开回报"""

def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """用户登录请求回报"""

def onRspError(self, error: dict, reqid: int, last: bool) -> None:
    """请求报错回报"""

def onRspSubMarketData(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """订阅行情回报"""
    # 订阅行情应答

def onRtnDepthMarketData(self, data: dict) -> None:
    """行情数据推送"""

def connect(self, address: str, userid: str, password: str, brokerid: int) -> None:
    """连接服务器"""
```

```
def login(self) -> None:
    """用户登录"""

def subscribe(self, req: SubscribeRequest) -> None:
    """订阅行情"""

def close(self) -> None:
    """关闭连接"""

def update_date(self) -> None:
    """更新当前日期"""

class CtpTdApi(TdApi):
    """

    def __init__(self, gateway: CtpGateway) -> None:
        """构造函数"""

    def onFrontConnected(self) -> None:
        """服务器连接成功回报"""
        # 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) -> None:
        """服务器连接断开回报"""

    def onRspAuthenticate(self, data: dict, error: dict, reqid: int, last: bool) -> None:
        """用户授权验证回报"""

    def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool) -> None:
        """用户登录请求回报"""

    def onRspOrderInsert(self, data: dict, error: dict, reqid: int, last: bool) -> None:
        """委托下单失败回报"""

    def onRspOrderAction(self, data: dict, error: dict, reqid: int, last: bool) -> None:
        """委托撤单失败回报"""

    def onRspSettlementInfoConfirm(self, data: dict, error: dict, reqid: int, last: bool) -> None:
        """确认结算单回报"""
```

```
def onRspQryInvestorPosition(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """持仓查询回报"""

def onRspQryTradingAccount(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """资金查询回报"""

def onRspQryInstrument(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """合约查询回报"""

def onRtnOrder(self, data: dict) -> None:
    """委托更新推送"""

def onRtnTrade(self, data: dict) -> None:
    """成交数据推送"""

def onRspForQuoteInsert(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """询价请求回报"""

def connect(
    .....
) -> None:
    """连接服务器"""

def authenticate(self) -> None:
    """发起授权验证"""

def login(self) -> None:
    """用户登录"""

def send_order(self, req: OrderRequest) -> str:
    """委托下单"""

def cancel_order(self, req: CancelRequest) -> None:
    """委托撤单"""

def query_account(self) -> None:
    """查询资金"""

def query_position(self) -> None:
```

```
""" 查询持仓 """
```

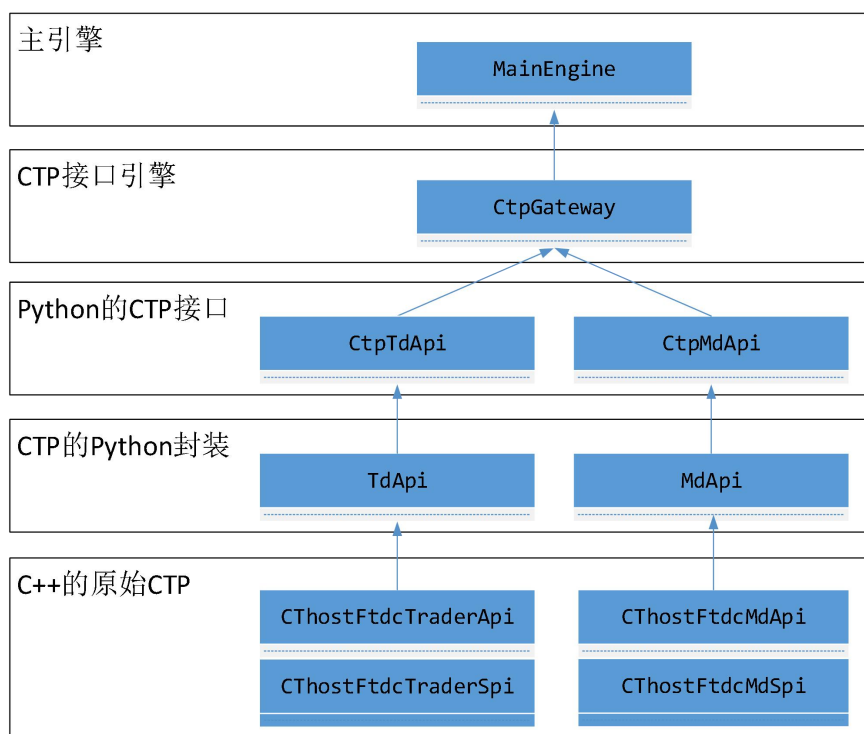
```
def close(self) -> None:
```

```
""" 关闭连接 """
```

```
def adjust_price(price: float) -> float:
```

```
""" 将异常的浮点数最大值 (MAX_FLOAT) 数据调整为 0 """
```

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。

据我观察，这两个类的代码在每个版本中都有所变化，这其实表现了这部分编程的难度。这两个类既是通信程序又要与业务相结合，需要考虑的情况很多。我自己写的交易策略只考虑了其中很小的一部分，感谢 vn.py 提供了这样丰富的经验参考。

第 7 章 CTP 接口的使用

前一章已经介绍了 CTP 接口类 CtpGateway，本章分析 VeighNa Trader 主窗口如何使用它。

7.1. 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，或者本文档的 2.1 和 2.2 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，或者本文档的 2.3 和 2.5 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_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，就可以少传入那些已指定值的参数。

3-增加菜单项

主窗口的 `add_action` 函数用于增加菜单项。

```
def add_action(
    self,
    menu: QtWidgets.QMenu,
    action_name: str,
    icon_name: str,
    func: Callable,
    toolbar: bool = False
) -> None:
    """

    icon = QtGui.QIcon(icon_name)

    action = QtWidgets.QAction(action_name, self)
    action.triggered.connect(func)
    action.setIcon(icon)

    menu.addAction(action)
```

```

if toolbar:
    self.toolbar.addAction(action)

```

调用 `add_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')`。

7.2. “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“连接”对话框。

7.3. 通用连接对话框类 `ConnectDialog`

`ConnectDialog` 在 `D:\vnpy300\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。

7.4. CTP 接口的 connect 函数

后续的执行就进入到 CtpGateway 类内部，CtpGateway 类在 vnpy_ctp 包的 gateway 目录的 ctp_gateway.py 文件中定义，其 connect 函数代码为：

```

def connect(self, setting: dict) -> None:
    """连接交易接口"""

    # 取连接配置信息
    userid: str = setting["用户名"]
    password: str = setting["密码"]
    brokerid: str = setting["经纪商代码"]
    td_address: str = setting["交易服务器"]
    md_address: str = setting["行情服务器"]
    appid: str = setting["产品名称"]
    auth_code: str = setting["授权编码"]

    # 对服务器地址进行处理
    if (
        (not td_address.startswith("tcp://"))
        and (not td_address.startswith("ssl://"))
    ):

```

```

        and (not td_address.startswith("socks"))
    ):
        td_address = "tcp://" + td_address

    if (
        (not md_address.startswith("tcp://"))
        and (not md_address.startswith("ssl://"))
        and (not md_address.startswith("socks"))
    ):
        md_address = "tcp://" + md_address

    # 连接交易服务器
    self.td_api.connect(td_address, userid, password, brokerid, auth_code, appid)

    # 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) -> None:
    """初始化查询任务"""
    self.count: int = 0

    # 将查询函数设为 td_api 的资金和持仓查询函数
    self.query_functions: list = [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。

7.5. 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(`process_timer_event` 也是 `CtpGateway` 类的方法)：

```

def process_timer_event(self, event) -> None:
    """定时事件处理"""

    # 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)

# 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。

第 8 章 Tick 数据的使用

VeighNa 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。

8.1. 在主窗口上订阅行情

下图是 VeighNa Trader 主窗口的左上部分。

交易

交易所 CFFEX

代码

名称

方向 多

开平

类型 限价

价格

数量

接口 CTP

委托

全撤

这部分窗口在 D:\vnpy300\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：

```
class TradingWidget(QtWidgets.QWidget):  
    """  
    通用的手工交易窗口  
    """  
  
    signal_tick = QtCore.pyqtSignal(Event)  
  
    def __init__(self, main_engine: MainEngine, event_engine: EventEngine):  
        """初始化"""  
        super().__init__()  
  
        self.main_engine: MainEngine = main_engine  
        self.event_engine: EventEngine = event_engine  
  
        # 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  
  
        # 界面初始化
```

```
self.init_ui()

self.register_event()
```

```
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（假设使用 CTP 接口），相关代码如下：

```

def subscribe(self, req: SubscribeRequest):
    """订阅行情"""
    self.md_api.subscribe(req)

```

调用行情接口的 subscribe 函数，相关代码如下：

```

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，参本文档“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。

8.1.1. 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 VeighNa Trader 的查询合约功能，从中找到当前合法合约的代码。

8.1.2. 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 VeighNa 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。

8.2. 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) -> None:
    """
    行情数据推送
    收到 Tick 数据时的回调函数
    """

    # 过滤没有时间戳的异常行情数据
    if not data["UpdateTime"]:
        return
```

```
# 过滤还没有收到合约数据前的行情推送
symbol: str = data["InstrumentID"]
contract: ContractData = symbol_contract_map.get(symbol, None)
if not contract:
    return

# 对大商所的交易日字段取本地日期
if not data["ActionDay"] or contract.exchange == Exchange.DCE:
    date_str: str = self.current_date
else:
    date_str: str = data["ActionDay"]

# 生成行情时间，并进行时区转换
timestamp: str = f"{date_str} {data['UpdateTime']}. {int(data['UpdateMillisec']/100)}"
dt: datetime = datetime.strptime(timestamp, "%Y%m%d %H:%M:%S.%f")
dt: datetime = CHINA_TZ.localize(dt)

# 创建 TickData 对象保存行情数据
tick: TickData = TickData(
    symbol=symbol,
    exchange=contract.exchange,
    datetime=dt,
    name=contract.name,
    volume=data["Volume"],
    turnover=data["Turnover"],
    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)

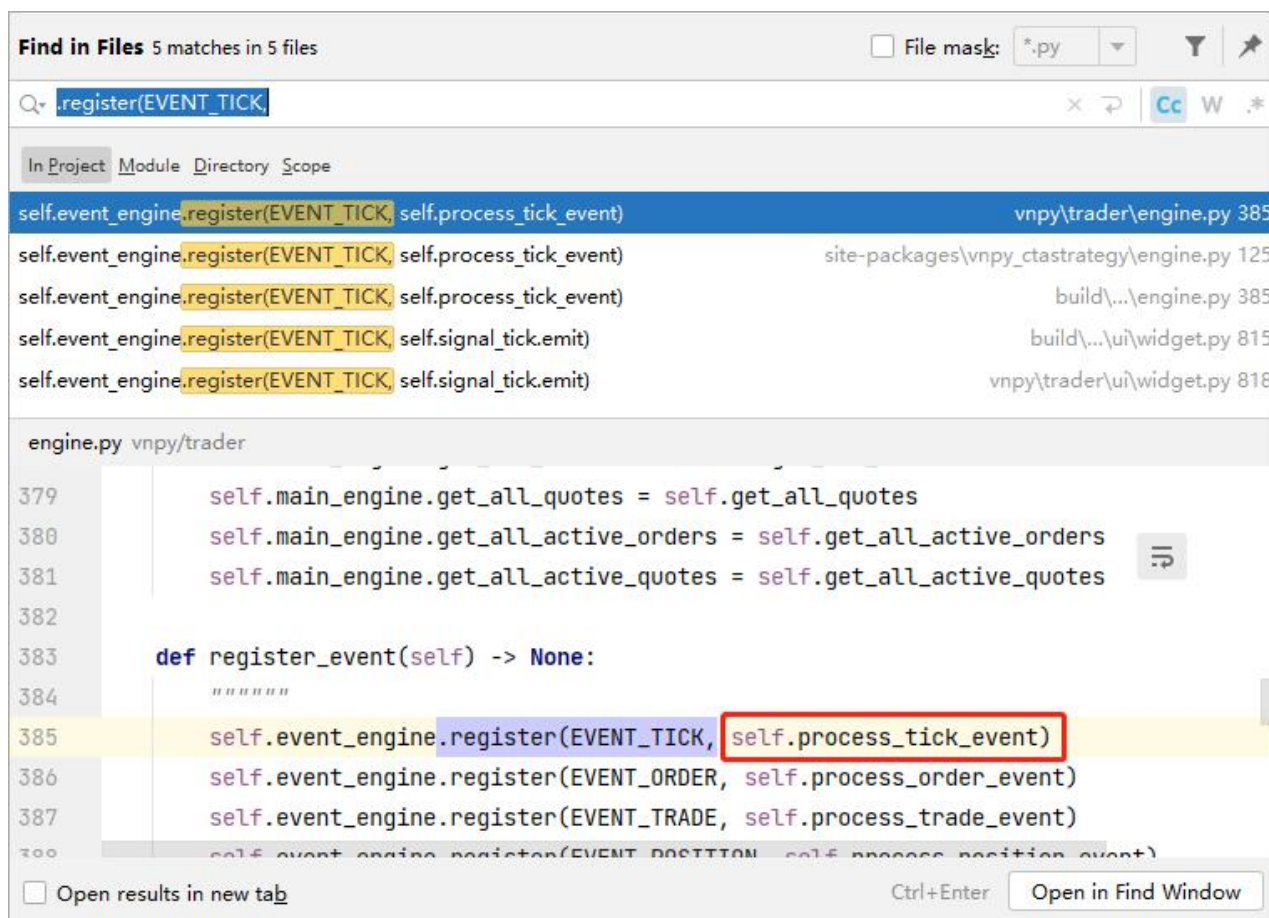
```

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。

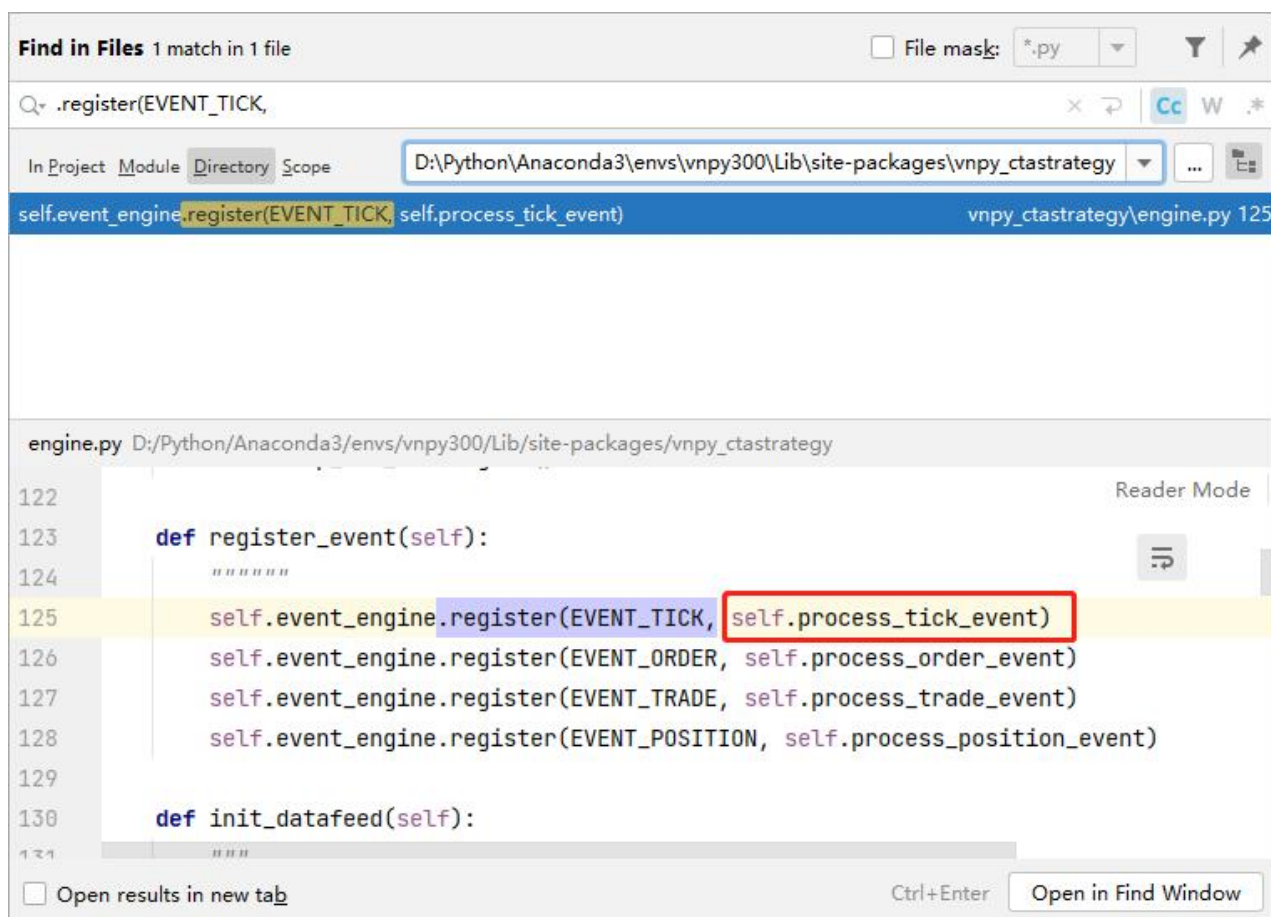
8.3. Tick 数据使用在哪里

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。下图是在 vnpy300 项目中查找的结果。



下图是在 vnpy_ctastrategy 包中查找的结果。



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
- 价差交易（Spread Trading）
- VeighNa 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 vnpy_ctastrategy 包中的 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)

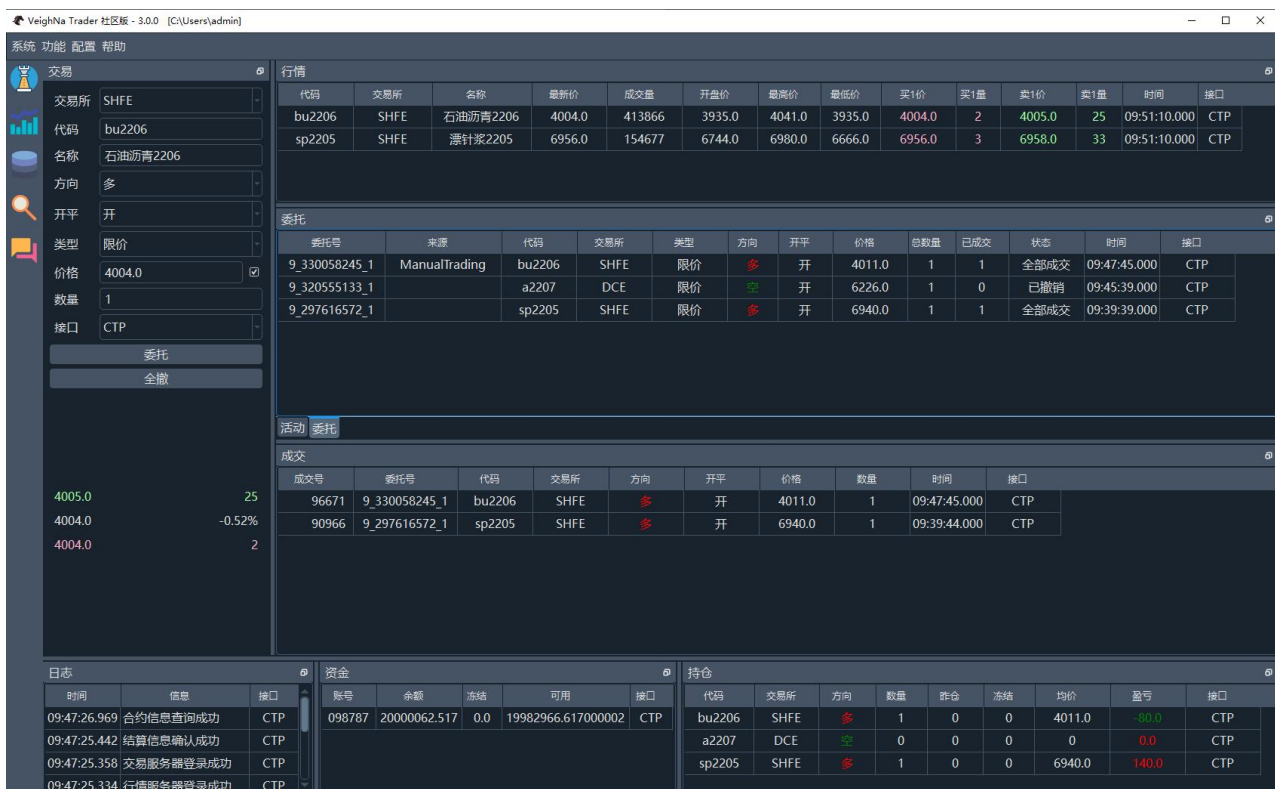
```

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。

8.4. 在界面上使用 Tick 数据

先看 VeighNa Trader 的主界面。



8.4.1. 在交易子窗口中使用

在 D:\vnpy300\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，相关代码如下：

```
class TradingWidget(QtWidgets.QWidget):
```

```
"""
```

```
通用的手工交易窗口
```

```
"""
```

```
# 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))

# 如果价格输入框右侧的复选框被选中，将当前价格写入价格输入框
# 这样可以方便委托下单
if self.price_check.isChecked():
    self.price_line.setText(f"{tick.last_price:. {price_digits}f}")

```

8.4.2. 在行情子窗口中使用

观察 VeighNa 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:\vnpy300\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 QTableWidgetItem 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，QTableWidgetItem 中显示不同的列，参教材 16.4 节以及本文档的 2.4 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:
    """
    Tick 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.__getattr____(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)
```

第 9 章 OMS 引擎

回顾教材“16.2. 主引擎”以及本文档 2.2 节的内容，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。

9.1. OMS 引擎的定义

OMS 引擎类 `OmsEngine` 在 `D:\vnpy300\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 VeighNa Trader 提供委托单管理系统 (order management system, OMS)
    """

    def __init__(self, main_engine: MainEngine, event_engine: EventEngine):
        """初始化"""
        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] = {}

        # 询价单字典，每个 vt_orderid 一个询价单
        self.quotes: Dict[str, QuoteData] = {}

        # 活动委托单字典
        self.active_orders: Dict[str, OrderData] = {}

        # 活动委询价单字典
        self.active_quotes: Dict[str, QuoteData] = {}

        # 将查询函数加到主引擎
        self.add_function()

        # 注册事件处理函数
        self.register_event()
```

9.2. OMS 引擎与主引擎的关系

把与委托单相关的功能放到 OMS 引擎中实现，但又要通过主引擎来调用，相关代码如下：

```
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_quote = self.get_quote

    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_quotes = self.get_all_quotes
    self.main_engine.get_all_active_orders = self.get_all_active_orders
    self.main_engine.get_all_active_quotes = self.get_all_active_quotes
```

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)
    self.event_engine.register(EVENT_QUOTE, self.process_quote_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。

9.3. 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 VeighNa Trader 的“帮助 - 查询合约”功能查询。

本地代码 vt_symbol	代码 symbol	交易所 exchange	名称 name	合约分类 product	合约乘数 size	价格跳动 pricetick	最小委托量 min_volume	交易接口 gateway_name
IF2012.CFFEX	IF2012	CFFEX	沪深300股指2012	期货	300	0.2	1	CTP
IF2101.CFFEX	IF2101	CFFEX	沪深300股指2101	期货	300	0.2	1	CTP
IF2106.CFFEX	IF2106	CFFEX	沪深300股指2106	期货	300	0.2	1	CTP
IF2103.CFFEX	IF2103	CFFEX	沪深300股指2103	期货	300	0.2	1	CTP

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:\vnpy300\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，相关代码如下：

```
def show_contracts(self) -> None:
    """
    根据代码过滤字符串显示合约
    """
    # 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。

9.4. 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 `vnpy_ctp` 包的 `gateway` 目录下，分析 `ctp_gateway.py` 文件中 `CtpTdApi` 类的定义。

9.4.1. 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(
    self,
    address: str,
    userid: str,
    password: str,
    brokerid: int,
    auth_code: str,
    appid: str
) -> None:
    """连接服务器"""
    self.userid = userid
    self.password = password
    self.brokerid = brokerid
    self.auth_code = auth_code
    self.appid = appid
```

```
if not self.connect_status:
    path: Path = get_folder_path(self.gateway_name.lower())
    self.createFtdcTraderApi((str(path) + "\\Td").encode("GBK"))

    self.subscribePrivateTopic(0)
    self.subscribePublicTopic(0)

    self.registerFront(address)
    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。

9.4.2. onFrontConnected 回调函数

如果连接成功，触发 onFrontConnected 回调函数。

```
def onFrontConnected(self) -> None:
    """服务器连接成功回报"""
    # 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。
    self.gateway.write_log("交易服务器连接成功")

    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) -> None:
    """用户授权验证回报"""
    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.auth_failed = True

        self.gateway.write_error("交易服务器授权验证失败", error)
```

如果认证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。

9.4.3. 交易用户登录

```
def login(self) -> None:
    """用户登录"""
    if self.login_failed:
        return

    ctp_req: dict = {
        "UserID": self.userid,
        "Password": self.password,
        "BrokerID": self.brokerid,
        "AppID": self.appid
    }

    self.reqid += 1
    self.reqUserLogin(ctp_req, self.reqid)
```

向服务器发送登录请求。

9.4.4. 登录响应

```
def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool):
    """用户登录请求回报"""
    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。

9.4.5. 协议确认响应

```
def onRspSettlementInfoConfirm(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """确认结算单回报"""
    self.gateway.write_log("结算信息确认成功")

    # 由于流控，单次查询可能失败，通过 while 循环持续尝试，直到成功发出请求
    while True:
        self.reqid += 1
        n: int = self.reqQryInstrument({}, self.reqid)

        if not n:
            break
        else:
```

```
sleep(1)
```

协议确认通过，请求查询合约。

9.4.6. 查询合约响应

```
def onRspQryInstrument(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """合约查询回报"""
    .....

    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。

9.4.7. 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。

9.4.8. 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，就跟本文档前述“9.1 OMS 引擎的定义”一节的内容对接起来了。

9.5. 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 VeighNa Trader 上的手工委托为例进行分析。操作方法如下，在 VeighNa 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。

交易

交易所

SHFE

代码

bu2206

名称

石油沥青2206

方向

多

开平

开

类型

限价

价格

3999.0

☒

数量

1

接口

CTP

委托

全撤

4000.0

8

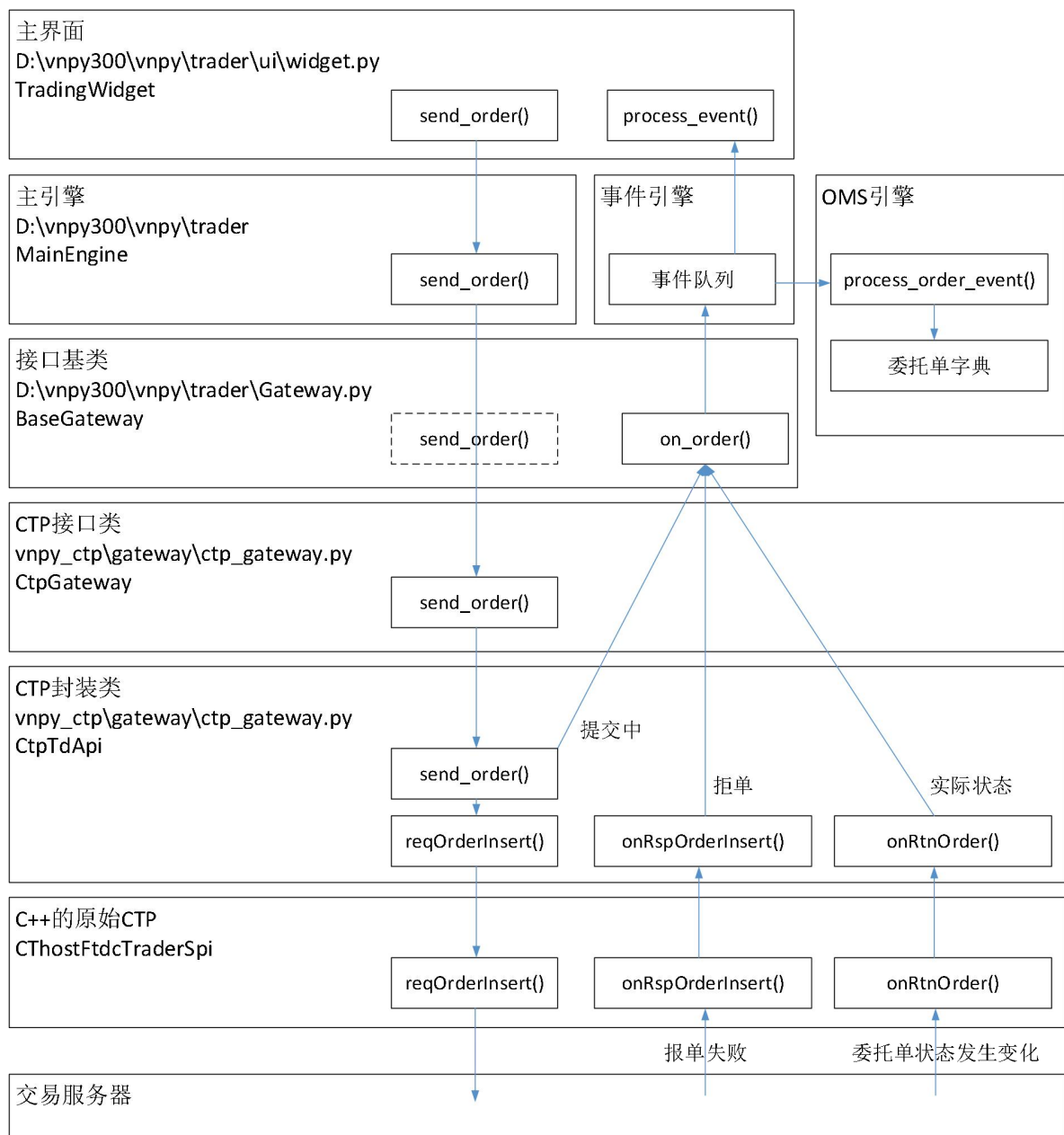
3999.0

-0.65%

3999.0

16

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。

9.5.1. 主界面上的处理

在主界面上，交易子窗口在 D:\vnpy300\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(QWidgets.QWidget):
    """
    通用的手工交易窗口
    """

    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。

9.5.2. 主引擎中的处理

VeighNa 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。

9.5.3. 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 `vnpy_ctp\gateway\ctp_gateway.py` 文件中的 `CtpGateway` 类中定义，代码如下：

```
def send_order(self, req: OrderRequest) -> str:
    """委托下单"""
    return self.td_api.send_order(req)
```

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 `vnpy_ctp\gateway\ctp_gateway.py` 文件中定义，其 `send_order()` 函数代码如下：

```
def send_order(self, req: OrderRequest) -> str:
    """委托下单"""

    # 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

    tp = ORDERTYPE_VT2CTP[req.type]
    price_type, time_condition, volume_condition = tp

    # 使用 vn.py 的委托单请求，创建 CTP 的委托单请求
    ctp_req: dict = {
        "InstrumentID": req.symbol,
        "ExchangeID": req.exchange.value,
        "LimitPrice": req.price,
        "VolumeTotalOriginal": int(req.volume),
        "OrderPriceType": price_type,
        "Direction": DIRECTION_VT2CTP.get(req.direction, ""),
        "CombOffsetFlag": OFFSET_VT2CTP.get(req.offset, ""),
```

```

        "OrderRef": str(self.order_ref),
        "InvestorID": self.userid,
        "UserID": self.userid,
        "BrokerID": self.brokerid,
        "CombHedgeFlag": THOST_FTDC_HF_Speculation,
        "ContingentCondition": THOST_FTDC_CC_Immediately,
        "ForceCloseReason": THOST_FTDC_FCC_NotForceClose,
        "IsAutoSuspend": 0,
        "TimeCondition": time_condition,
        "VolumeCondition": volume_condition,
        "MinVolume": 1
    }

    # 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)

    # 生成委托单号
    orderid: str = f"{self.frontid}_{self.sessionid}_{self.order_ref}"

    # 根据定单请求数据创建委托单数据
    order: OrderData = 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”。

9.5.4. 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”。

如果报单指令（ReqOrderInsert）通过了报单检查，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

适用等等，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”。

如果报单通过了交易所系统的检查，交易所系统会将对应的报单插入报单簿，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”。

9.5.5. 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。

需要响应的第一类消息是在回调函数 onRspOrderInsert() 中响应 OnRspOrderInsert 消息，代码为：

```
def onRspOrderInsert(self, data: dict, error: dict, reqid: int, last: bool) -> None:
    """委托下单失败回报"""
    order_ref: str = data["OrderRef"]
    orderid: str = f"{self.frontid}_{self.sessionid}_{order_ref}"

    symbol: str = data["InstrumentID"]
    contract: ContractData = symbol_contract_map[symbol]

    order: OrderData = OrderData(
        symbol=symbol,
        exchange=contract.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。

需要响应的第二类消息是在回调函数 onRtnOrder() 中响应 OnRtnOrder 消息，代码为：

```
def onRtnOrder(self, data: dict) -> None:
    """
    委托更新推送
```

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: str = data["InstrumentID"]
contract: ContractData = symbol_contract_map[symbol]

frontid: int = data["FrontID"]
sessionid: int = data["SessionID"]
order_ref: str = data["OrderRef"]
orderid: str = f"{frontid}_{sessionid}_{order_ref}"

timestamp: str = f"{data['InsertDate']} {data['InsertTime']}"
dt: datetime = datetime.strptime(timestamp, "%Y%m%d %H:%M:%S")
dt: datetime = CHINA_TZ.localize(dt)

tp = (data["OrderPriceType"], data["TimeCondition"], data["VolumeCondition"])

order: OrderData = OrderData(
    symbol=symbol,
    exchange=contract.exchange,
    orderid=orderid,
    type=ORDERTYPE_CTP2VT[tp],
    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。

9.5.6. 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												
委托号	来源	代码	交易所	类型	方向	开平	价格	总数量	已成交	状态	时间	接口
3_1331826221_6		sp2105	SHFE	限价	空	平今	7068.0	1	0	未成交	14:50:13	CTP
3_1331826221_5		a2105	DCE	限价	多	开	5672.0	1	1	全部成交	14:45:01	CTP
3_1331826221_4		a2105	DCE	限价	空	平	5668.0	1	1	全部成交	14:29:03	CTP
3_1331826221_3		sp2105	SHFE	限价	多	开	7060.0	1	1	全部成交	14:09:42	CTP

9.5.7. 委托子窗口中的处理

委托子窗口在 D:\vnpy300\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。后面的处理在本文档“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。

9.5.8. 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)
    self.event_engine.register(EVENT_QUOTE, self.process_quote_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。

第 10 章 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。

10.1. 上层应用程序类 CtaBacktesterApp

在 D:\vnpy300\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 vnpy_ctabacktester 包的 __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"
```

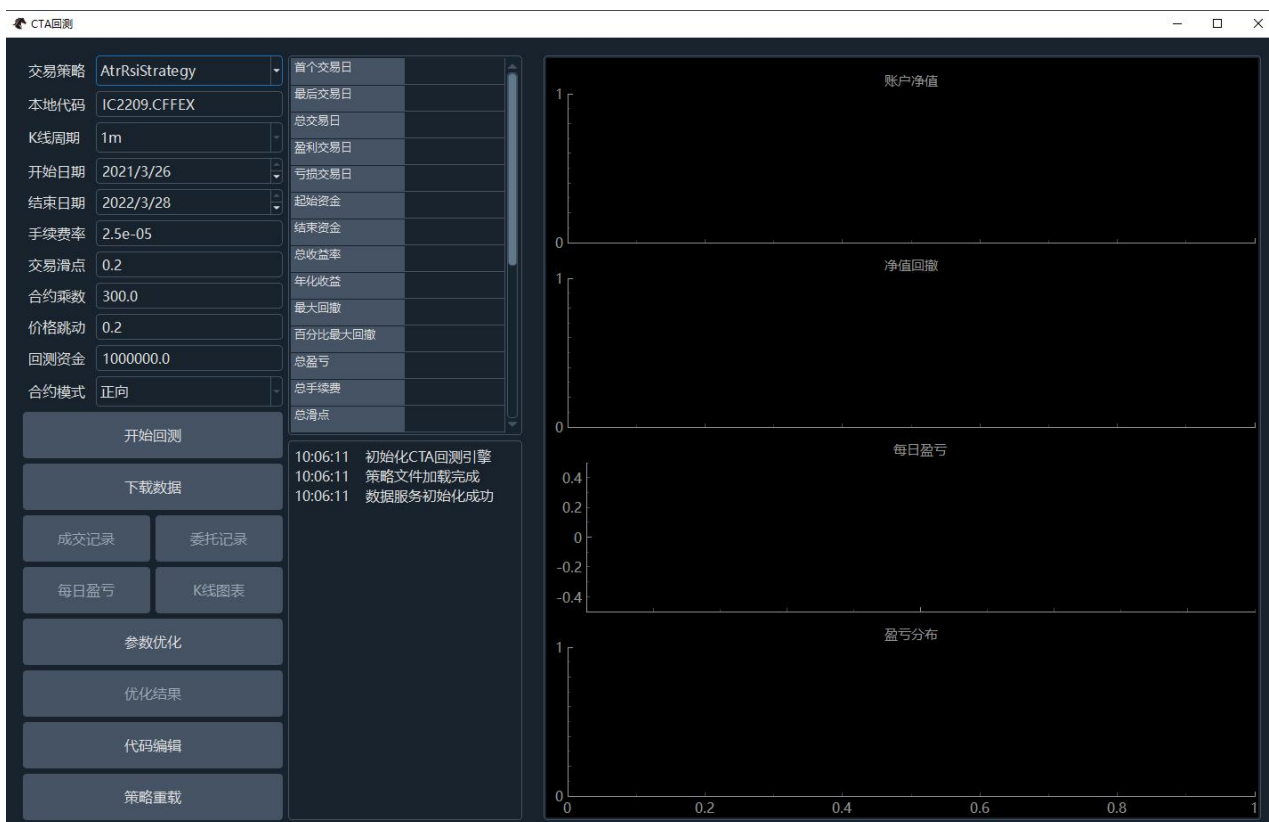
程序执行之后，部分结果如下：

```
app_name = "CtaBacktester"
app_module = "vnpy.app.cta_backtester"
app_path = "D:\vnpy300\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。

10.2. 回测窗口类 BacktesterManager

执行时 CTA 回测窗口的界面如下图所示。



在 `vnpy_ctabacktester` 包的 `ui` 目录下的 `widget.py` 文件中定义回测窗口类 `BacktesterManager(QtWidgets.QWidget)`。

在 `VeighNa Trader` 的 `init_menu` 函数中，将 App 增加到菜单和工具栏。

10.2.1. 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。

交易策略	AtrRsiStrategy
本地代码	IC2209.CFFEX
K线周期	1m
开始日期	2021/3/26
结束日期	2022/3/28
手续费率	2.5e-05
交易滑点	0.2
合约乘数	300.0
价格跳动	0.2
回测资金	1000000.0
合约模式	正向

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。

10.2.2. 界面初始化

界面初始化部分的代码如下：

```
def init_ui(self):
    """界面初始化"""
    self.setWindowTitle("CTA 回测")

    .....

    self.candle_button = QtWidgets.QPushButton("K 线图表")
    self.candle_button.clicked.connect(self.show_candle_chart)
    self.candle_button.setEnabled(False)
    .....

    # K 线图表
    self.candle_dialog = CandleChartDialog()
```

可以看到，“K 线图表”按钮的槽函数为 `show_candle_chart()`。

这里只是先关注一下，希望将来有机会定制自己的 K 线图表，并将缠论的分析结果加入进去。

10.2.3. 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.addItem(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):
        """返回策略名称列表"""
        return list(self.classes.keys())
```

可见，程序的关键是 `load_strategy_class` 方法，将在本文档第 17 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。

10.2.4. 各窗口控件

比较简单，略。

第 11 章 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:\vnpy300\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。

11.1. K 线合成器 BarGenerator

前面提到了订阅行情。vn.py 通过 CTP 接口连接交易所，订阅行情后就能几乎实时地收到 Tick 数据。如果策略是基于 Tick 数据的，可以直接使用；如果是基于 K 线的，K 线数据需要在本地生成，这个任务由“K 线合成器”完成。

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

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)

    # 当前 K 线内的成交额
    turnover_change = tick.turnover - self.last_tick.turnover
    self.bar.turnover += max(turnover_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、成交量、成交额和持仓量
```

```
self.window_bar.close_price = bar.close_price
self.window_bar.volume += bar.volume
self.window_bar.turnover += bar.turnover
self.window_bar.open_interest = bar.open_interest

# C 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

# 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,
            close_price=bar.close_price,
            volume=bar.volume,
```

```
        turnover=bar.turnover,
        open_interest=bar.open_interest
    )
    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、成交量、成交额和持仓量
    self.hour_bar.close_price = bar.close_price
    self.hour_bar.volume += bar.volume
    self.hour_bar.turnover += bar.turnover
    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 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,
    close_price=bar.close_price,
    volume=bar.volume,
    turnover=bar.turnover,
    open_interest=bar.open_interest
)

# 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、成交量、成交额和持仓量
    self.hour_bar.close_price = bar.close_price
    self.hour_bar.volume += bar.volume
    self.hour_bar.turnover += bar.turnover
    self.hour_bar.open_interest = bar.open_interest

# P 缓存新的小时线数据
if finished_bar:
    self.on_hour_bar(finished_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、成交量、成交额和持仓量
    self.window_bar.close_price = bar.close_price
    self.window_bar.volume += bar.volume
    self.window_bar.turnover += bar.turnover
    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

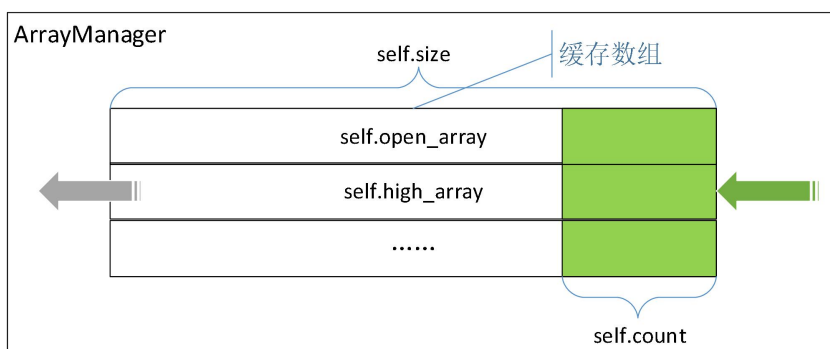
    return bar

```

BarData 的定义参“VeighNa Trader 的全局定义”一节。

11.2. K 线序列管理工具 ArrayManager

有的策略 K 线数据用完就完了, 不需要保存。大多数策略需要保存一定数量的数据, 比如用于计算均线值等。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 时缓存数组未滿, 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 时缓存数组已滿, 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.turnover_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.turnover_array[:-1] = self.turnover_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.turnover_array[-1] = bar.turnover
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:
    """
    G 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 turnover(self) -> np.ndarray:
    """
    获取成交额序列
    """
    return self.turnover_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。

第 12 章 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。

12.1. CTA 策略模板 `CtaTemplate`

在 `vnpy_ctastrategy` 包的 `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

    bars = 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

    for bar in bars:
        callback(bar)

def load_tick(self, days: int):
    """
    在初始化策略时加载历史 Tick 数据
    """
    ticks = 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

    for tick in ticks:
        self.on_tick(tick)

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。

12.1.1. 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副本。

12.1.2. 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。

12.1.3. 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。

12.1.4. 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。
- net: 布尔型参数，默认为 False，表示委托的是否为净仓单。

这些参数在策略引擎的各个层次都会用到，含义相同。

12.2. 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 日均线）和慢均线（20 日均线）。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 vnpy_ctastrategy 包的 strategies 目录下的文件 double_ma_strategy.py, 增加了注释的代码如下。

```
from vnpy_ctastrategy 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):
    """
    C 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)

    def 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。

12.3. 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。

12.3.1. 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 Futures 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：

$$\text{Range} = \text{Max}(\text{HH} - \text{LC}, \text{HC} - \text{LL})$$

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。

增加了注释的策略代码如下：

```
from datetime import time
from vnpy.app.cta_strategy import (
    CtaTemplate,
    StopOrder,
    TickData,
    BarData,
    TradeData,
    OrderData,
    BarGenerator,
    ArrayManager,
)

class DualThrustStrategy(CtaTemplate):
    """Dual Thrust 策略"""

    author = "用 Python 的交易员"

    fixed_size = 1      # 每次固定的买卖数量
    # 用于控制多头和空头的触发条件的系数
    k1 = 0.4
    k2 = 0.6
```

```
# 用于计算震荡区间的K线
bars = []

# 日内开盘、最高、最低价
day_open = 0
day_high = 0
day_low = 0

# 震荡区间和多/空开仓价
day_range = 0
long_entry = 0
short_entry = 0

# 退出时间
# 注：本策略是日内策略，在收盘前要平掉所有持仓
exit_time = time(hour=22, minute=50)

# 当天是否开过多仓或空仓
long_entered = False
short_entered = False

parameters = ["k1", "k2", "fixed_size"]
variables = ["day_range", "long_entry", "short_entry"]

def __init__(self, cta_engine, strategy_name, vt_symbol, setting):
    """初始化"""
    super().__init__(cta_engine, strategy_name, vt_symbol, setting)

    self.bg = BarGenerator(self.on_bar)
    self.am = ArrayManager()
    self.bars = []

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
    """

    # 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
    """

    # 撤消由策略发出的所有活动的委托
    self.cancel_all()

    # self.bars 中保存 2 个 K 线
    self.bars.append(bar)
    if len(self.bars) <= 2:
        return
    else:
        self.bars.pop(0)
    last_bar = self.bars[-2]

    # 如果日期发生了改变
    if last_bar.datetime.date() != bar.datetime.date():
        # 计算震荡区间和多/空开仓价
        if self.day_high:
            self.day_range = self.day_high - self.day_low
            self.long_entry = bar.open_price + self.k1 * self.day_range
            self.short_entry = bar.open_price - self.k2 * self.day_range
```

```
# 重置当日内开盘、最高、最低价
self.day_open = bar.open_price
self.day_high = bar.high_price
self.day_low = bar.low_price

# 当天多空都未开
self.long_entered = False
self.short_entered = False
else:
    # 如果日期未改变, 更新当日最高最低价
    self.day_high = max(self.day_high, bar.high_price)
    self.day_low = min(self.day_low, bar.low_price)

# 如果还没有震荡区间, 什么都不做
if not self.day_range:
    return

# 如果还没到退出时间 (可以交易)
if bar.datetime.time() < self.exit_time:
    # 如果当前无持仓
    if self.pos == 0:
        # 如果当前价高于当日开盘价
        if bar.close_price > self.day_open:
            # 如果今天还没有开多仓
            if not self.long_entered:
                # 发出一个开多的停止单
                self.buy(self.long_entry, self.fixed_size, stop=True)
        # 如果当前价低于当日开盘价
        else:
            # 如果今天还没有开空仓
            if not self.short_entered:
                # 发出一个开空的停止单
                self.short(self.short_entry,
                           self.fixed_size, stop=True)

# 如果持有多仓
elif self.pos > 0:
    # 置已开多仓标志
```

```
        self.long_entered = True

    # 预埋一个位于震荡区间下轨的平多条件单，效果是当价格向下突破下轨时平多仓
    self.sell(self.short_entry, self.fixed_size, stop=True)

    # 如果今天还没有开空仓
    if not self.short_entered:
        # 预埋一个位于震荡区间下轨的买空条件单
        self.short(self.short_entry, self.fixed_size, stop=True)

    # 如果持有空仓
    elif self.pos < 0:
        # 置已开空仓标志
        self.short_entered = True

    # 预埋一个位于震荡区间上轨的平空条件单，效果是当价格向上突破上轨时平空仓
    self.cover(self.long_entry, self.fixed_size, stop=True)

    # 如果今天还没有开多仓
    if not self.long_entered:
        # 预埋一个位于震荡区间上轨的买多条件单
        self.buy(self.long_entry, self.fixed_size, stop=True)

    # 如果已经到了退出时间，平掉所有持仓
    else:
        if self.pos > 0:
            self.sell(bar.close_price * 0.99, abs(self.pos))
        elif self.pos < 0:
            self.cover(bar.close_price * 1.01, abs(self.pos))

    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

```

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。

12.3.2. 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 (Relative 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 $ATR > ATR_{Ma}$ 时，显示市场波动性增大，趋势正在增强。只有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，开仓做空。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。

增加了注释的策略代码如下：

```

from vnpy.app.cta_strategy import (
    CtaTemplate,
    StopOrder,
    TickData,
    BarData,
    TradeData,
    OrderData,
    BarGenerator,
    ArrayManager,
)

class AtrRsiStrategy(CtaTemplate):
    """AtrRsi 策略"""

```

```
author = "用Python的交易员"

# ATR 指标计算天数
atr_length = 22
# 取 ATR 移动平均的天数
atr_ma_length = 10
# RSI 指标计算天数
rsi_length = 5
# RSI 指标的入口范围
rsi_entry = 16
# 回撤止损百分比
trailing_percent = 0.8
# 每次固定的买卖数量
fixed_size = 1

# ATR 指标值
atr_value = 0
# ATR 指标移动平均值
atr_ma = 0
# RSI 指标值
rsi_value = 0
# RSI 指标开多阈值
rsi_buy = 0
# RSI 指标开空阈值
rsi_sell = 0
# 持仓期间达到的最高价，用于计算止损/溢价
intra_trade_high = 0
# 持仓期间达到的最低价，用于计算止损/溢价
intra_trade_low = 0

parameters = [
    "atr_length",
    "atr_ma_length",
    "rsi_length",
    "rsi_entry",
    "trailing_percent",
    "fixed_size"
]
variables = [
```

```
        "atr_value",
        "atr_ma",
        "rsi_value",
        "rsi_buy",
        "rsi_sell",
        "intra_trade_high",
        "intra_trade_low"
    ]

def __init__(self, cta_engine, strategy_name, vt_symbol, setting):
    """初始化"""
    super().__init__(cta_engine, strategy_name, vt_symbol, setting)
    self.bg = BarGenerator(self.on_bar)
    self.am = ArrayManager()

def on_init(self):
    """
    策略初始化时的回调函数
    """
    self.write_log("策略初始化")

    # RSI 指标开多/空阈值
    # rsi_entry 作为参数在回测之前可能修改，因此在此处计算
    self.rsi_buy = 50 + self.rsi_entry
    self.rsi_sell = 50 - self.rsi_entry

    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
    """
    # 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
    """
    # 撤消由策略发出的所有活动的委托
    self.cancel_all()

    am = self.am
    am.update_bar(bar)
    if not am.inited:
        return

    # 计算指标值
    atr_array = am.atr(self.atr_length, array=True)
    self.atr_value = atr_array[-1]
    self.atr_ma = atr_array[-self.atr_ma_length:].mean()
    self.rsi_value = am.rsi(self.rsi_length)

    # 如果当前空仓
    if self.pos == 0:
        # 计算持仓期间达到的最高/低价
        self.intra_trade_high = bar.high_price
        self.intra_trade_low = bar.low_price

        # 当 ATR>ATRMa 时，显示市场波动性增大，趋势正在增强
        if self.atr_value > self.atr_ma:
            # 当 RSI>规定上限时，开仓做多
            if self.rsi_value > self.rsi_buy:
                self.buy(bar.close_price + 5, self.fixed_size)
            # 当 RSI<规定下限时，开仓做空
            elif self.rsi_value < self.rsi_sell:
                self.short(bar.close_price - 5, self.fixed_size)
```

```
# 如果持有多仓
elif self.pos > 0:
    # 重新计算持仓期间最高/低价
    self.intra_trade_high = max(self.intra_trade_high, bar.high_price)
    self.intra_trade_low = bar.low_price

    # 多仓的止损/溢价
    long_stop = self.intra_trade_high * \
        (1 - self.trailing_percent / 100)
    # 按止损/溢价发出停止单
    self.sell(long_stop, abs(self.pos), stop=True)

# 如果持有空仓
elif self.pos < 0:
    # 重新计算持仓期间最高/低价
    self.intra_trade_low = min(self.intra_trade_low, bar.low_price)
    self.intra_trade_high = bar.high_price

    # 空仓的止损/溢价
    short_stop = self.intra_trade_low * \
        (1 + self.trailing_percent / 100)
    # 按止损/溢价发出停止单
    self.cover(short_stop, abs(self.pos), stop=True)

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

此策略中用到了固定百分比点位移动止损，可以参考。

12.3.3. 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。

第 13 章 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。

13.1. 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 vnpy_ctastrategy 包的 template.py 文件中。CtaSignal 类的代码如下：

```
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。

13.2. 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 vnpy_ctastrategy 包的 template.py 文件中。TargetPosTemplate 类的代码如下：

```
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。

13.3. 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 `vnpy_ctastrategy` 包的 `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)

    def 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。

第 14 章 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。

14.1. 回测执行引擎的初始化

回测执行引擎类 `BacktestingEngine` 在 `vnpy_ctastrategy` 包的 `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.risk_free: float = 0         # 无风险利率?
self.annual_days: int = 240       # 每年的交易天数?
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

```

14.2. 回测执行函数

教材第 18 章 (本文档第 4 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 = 0    # 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 天的数据回调完成，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

# 进度显示
# 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 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。

14.3. 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。

14.3.1. 在回测引擎类中

回测引擎类在 `vnpy_ctabacktester` 包的 `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):
    """CTA 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。

14.3.2. 在回测执行引擎类中

回测执行引擎类在 `vnpy_ctastrategy` 包的 `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  # 加载数据时的回调函数

    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。

14.3.3. 在策略模板类中

策略模板类已经在本文档 12.1 节进行了详细分析，本小节再对其中与回调函数相关的部分重点分析一下。策略模板类在 `vnpy_ctastrategy` 包的 `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？

14.3.4. 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。

14.3.5. 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。

14.4. 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 15 章 K 线图表

略。（等有时间再完成）

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。

第 16 章 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。

16.1. 在 constant.py 中定义常量

在 D:\vnpy300\vnpy\trader 目录下的 constant.py 文件中，定义了 VeighNa Trader 中用到的字符串常量。代码如下：

```
"""
VeighNa 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
    BSE = "BSE" # Beijing Stock Exchange
    SGE = "SGE" # Shanghai Gold Exchange
    WXE = "WXE" # Wuxi Steel Exchange
    CFETS = "CFETS" # CFETS Bond Market Maker Trading System
    XBOND = "XBOND" # CFETS X-Bond Anonymous Trading 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
    SGX = "SGX" # Singapore Global Exchange
    CBOT = "CBOT" # 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 Product (Forex/CFD/Pink Sheet Equity)
IBKRATS = "IBKRATS"   # Paper Trading Exchange of IB

# 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"

```

16.1.1. 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 vn.py 中称停止单。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 vnpy_ctastrategy 包的 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
    datetime: datetime
    lock: bool = False
    net: bool = False
    vt_orderids: list = field(default_factory=list)
    status: StopOrderStatus = StopOrderStatus.WAITING
```

16.1.2. 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。

指令	buy	sell	short	cover
意义	多头建仓	多头平仓	空头建仓	空头平仓
方向direction	long	short	short	long
开平offset	开仓	平仓	开仓	平仓
盈利（按指令）	buy.price<sell.price		short.price>cover.price	
亏损（按指令）	buy.price>sell.price		short.price<cover.price	
盈利（按direction）	short.price>long.price			
亏损（按direction）	short.price<long.price			

16.2. 在 object.py 中定义数据结构

在 D:\vnpy300\vnpy\trader 目录下的 object.py 文件中，定义了 VeighNa Trader 中用于交易功能的基础数据结构。代码如下：

```
"""
VeighNa 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
    turnover: float = 0   # 成交额
    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

localtime: datetime = None

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
    turnover: float = 0    # 成交额
    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
    """
    return self.status in ACTIVE_STATUSES
```

```
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_listed: datetime = None
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
```

16.3. 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:\vnpy300\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。

16.3.1. 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
        td_available = self.short_td - self.short_td_frozen
        yd_available = self.short_yd - self.short_yd_frozen
    else:
        pos_available = self.long_pos - self.long_pos_frozen
        td_available = self.long_td - self.long_td_frozen
        yd_available = self.long_yd - self.long_yd_frozen

    # Split close order to close today/yesterday for SHFE/INE exchange
    if req.exchange in {Exchange.SHFE, Exchange.INE}:
        reqs = []
        volume_left = req.volume

        if td_available:

```

```
        td_volume = min(td_available, volume_left)
        volume_left -= td_volume

        td_req = copy(req)
        td_req.offset = Offset.CLOSETODAY
        td_req.volume = td_volume
        reqs.append(td_req)

    if volume_left and yd_available:
        yd_volume = min(yd_available, volume_left)
        volume_left -= yd_volume

        yd_req = copy(req)
        yd_req.offset = Offset.CLOSEYESTERDAY
        yd_req.volume = yd_volume
        reqs.append(yd_req)

    if volume_left > 0:
        open_volume = volume_left

        open_req = copy(req)
        open_req.offset = Offset.OPEN
        open_req.volume = open_volume
        reqs.append(open_req)

    return reqs

# Just use close for other exchanges
else:
    reqs = []
    volume_left = req.volume

    if pos_available:
        close_volume = min(pos_available, volume_left)
        volume_left -= pos_available

        close_req = copy(req)
        close_req.offset = Offset.CLOSE
        close_req.volume = close_volume
        reqs.append(close_req)
```

```
        if volume_left > 0:
            open_volume = volume_left

            open_req = copy(req)
            open_req.offset = Offset.OPEN
            open_req.volume = open_volume
            reqs.append(open_req)

    return reqs
```

16.3.2. 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

```

第 17 章 CTA 引擎 CtaEngine

本章分析 CTA 引擎类 CtaEngine。

17.1. CtaEngine 源代码分析

CTA 引擎类在 vnpy_ctastrategy 包的 engine.py 文件中定义。

900 多行程序，按源代码顺序，分为四个部分。

17.1.1. 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: strategy_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)
```

```
self.database: BaseDatabase = get_database()

self.datafeed: BaseDatafeed = get_datafeed()


def init_engine(self):
    """
    初始化引擎，在“CTA 策略”窗口初始化时被调用，
    在VeighNa Trader 的生命周期中，“CTA 策略”窗口只会被初始化一次。
    """

    self.init_datafeed()          # 初始化数据服务
    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()         # 注册事件处理函数
    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)


def init_datafeed(self):
    """
    初始化数据服务
    """
    result = self.datafeed.init()
    if result:
        self.write_log("数据服务初始化成功")


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")
```

灰色代码表示已经不再使用的代码。可以看出，新版本用通用的数据服务代替了原来默认的 rqdata。

17.1.2. 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_datafeed(
    self, symbol: str, exchange: Exchange, interval: Interval, start: datetime, end: datetime
):
    """
    从数据服务查询 K 线数据
    """

    req = HistoryRequest(
        symbol=symbol,
        exchange=exchange,
        interval=interval,
        start=start,
        end=end
    )

    data = self.datafeed.query_bar_history(req)
    return data


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，各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
```

17.1.3. 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
) -> List[BarData]:
    """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,
    )

    return bars

def load_tick(
    self,
    vt_symbol: str,
    days: int,
    callback: Callable[[TickData], None]
) -> List[TickData]:
    """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,
    )

    return ticks

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。

17.1.4. 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

```

17.1.5. 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_ctastrategy.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 suffix in [".py", ".pyd", ".so"]:
        pathname: str = str(path.joinpath(f"*.{suffix}"))
        for filepath in glob(pathname):

```

```
        filename: str = Path(filepath).stem
        name: str = f"{module_name}.{filename}"
        self.load_strategy_class_from_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)

        # 重载模块，确保如果策略文件中有任何修改，能够立即生效。
        importlib.reload(module)

        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)
```

17.2. 进一步说明

17.2.1. 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，例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。

17.2.2. 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。

17.2.3. 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。

下一章分析 CTA 策略界面。

第 18 章 CTA 策略界面

18.1. 上层应用程序类 CtaStrategyApp

上层应用程序类在 vnpy_ctastrategy 包的 __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"
```

18.2. CTA 策略窗口类 CtaManager

执行时 CTA 策略窗口的界面如下图所示。



在 `vnpy_ctastrategy` 包的 `ui` 目录下的 `widget.py` 文件中定义 CTA 策略窗口类 `CtaManager` (`QtWidgets.QWidget`)。

在 `VeighNa 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。

以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

第 19 章 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。

第 20 章 算法交易 (AlgoTrading)

略。

第 21 章 期权策略 (OptionMaster)

略。

第 22 章 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

第 23 章 交易风险管理(RiskManager)

略。

第 24 章 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

第 25 章 RPC 服务模块(rpc_service)

rpc_service: RPC 服务模块, 允许将某一 VeighNa 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