# 第 3 章

# vn.py 基础

本章将对 vn.py 基础进行介绍，首先从 vn.py 整体架构入手，再分别介绍 vn.py 的交易链路中的部分，最后结合 VeighNa Trader 的界面对该软件的各部分功能进行说明。通过本章的学习，读者将会对 vn.py 的整个数据链路有一定的了解，对于量化交易软件的构成部分有深刻的理解。


## 3.1 vn.py 的整体架构

vn.py 的架构主要分为三层，首先是底层接口，其负责与各交易所通信并提供各品种交易的接口；在底层接口之上，vn.py 提供了一套事件或时间驱动的中层引擎，其用于完成事件的分发、订单的路由与数据的分发等；最上层的接口用于实现不同的应用，例如 CTA 策略模块、GUI 界面、行情记录等，接下来将分别介绍这三层架构的内容。

### 3.1.1 底层接口

vn.py 的底层接口主要提供行情与交易的 API，将行情数据发送到中层的引擎并接收来自引擎的订单数据并将其发送。对于目前 vn.py 3.x 版本而言，其对不同厂商的接口进行了分离，表 3-1 列出了部分 vn.py 3.x 版本支持的部分交易接口与品种。

**表 3-1 vn.py 3.x 版本支持的部分交易接口与品种**

| 接　口 | 品　种 | 接　口 | 品　种 |
|---|---|---|---|
| CTP | 国内期货、期权 | 中泰 XTP | A 股、ETF 期权 |
| CTP Mini | 国内期货、期权 | 华鑫奇点 | A 股、ETF |
| CTP 证券 | ETF 期权 | 国泰君安 | A 股、两融 |
| 飞马 | 国内期货 | 东证 OST | A 股 |
| 恒生 UFT | 国内期货、ETF 期权 | 东方财富 EMT | A 股 |
| 易盛 | 国内期货、黄金 TD | 飞鼠 | 黄金 TD、国内期货 |
| 顶点飞创 | ETF 期权 | 金仕达黄金 | 黄金 TD |
| 顶点 HTS | ETF 期权 | 融航 | 期货资管 |

续表

| 接　口 | 品　种 | 接　口 | 品　种 |
|---|---|---|---|
| 杰宜斯 | 期货资管 | 恒生云 UF | A 股仿真 |
| 中汇亿达 | 银行间市场 | TTS | 国内期货仿真 |
| 掘金 | A 股仿真 | 火象 | 国内期货仿真 |

从表 3-1 不难看出，vn.py 支持许多交易接口，当用户需要交易特定的品种时，可以自由选择对应的接口完成交易等功能。得益于好的抽象，用户在切换交易接口时是无感的，因为 vn.py 已经对不同的交易接口进行了统一的封装。

### 3.1.2 中层引擎

vn.py 文件中的中层引擎包括事件引擎、订单路由及数据引擎等，其中事件引擎负责将底层接口的行情、交易执行情况等消息推送到订阅了该信息的上层应用，而订单路由则负责将上层应用的下单请求推送到底层接口的路由；数据引擎则负责处理上层应用对数据库数据的读取与处理。

中层引擎将程序中的各组件（不同的底层接口、数据库接口等）处理为统一的数据格式，以便上层应用来进行调用。

#### 1. 主引擎 MainEngine

主引擎 MainEngine 中包含交易中所需要的一系列方法，方法及其用途如表 3-2 所示。

**表 3-2 MainEngine 提供的方法及其用途**

| 方　法 | 用　途 |
|---|---|
| add_engine | 添加引擎（如日志引擎、订单管理引擎、邮件引擎等） |
| add_gateway | 添加底层交易接口（如 CTP 接口、UFT 接口等） |
| add_app | 添加上层应用（如实盘交易应用、回测应用等） |
| init_engines | 初始化所有已添加的引擎（默认包含日志引擎、订单管理引擎、邮件引擎） |
| write_log | 写日志，向日志引擎的队列中放入日志消息 |
| get_gateway | 根据名称获取底层交易接口实例 |
| get_engine | 根据名称获取引擎实例 |
| get_default_setting | 根据接口名称获取底层接口的默认配置信息 |
| get_all_gateway_names | 获取所有底层接口的名称 |
| get_all_apps | 获取所有应用的实例 |
| get_all_exchanges | 获取所有交易所 |
| connect | 根据名称完成底层接口的连接 |
| subscribe | 根据名称订阅品种行情信息 |
| send_order | 根据名称使用底层接口发送订单 |
| cancel_order | 根据名称使用底层接口取消订单 |

续表

| 方　法 | 用　途 |
| --- | --- |
| send_quote | 根据名称使用底层接口发送双边报价请求 |
| cancel_quote | 根据名称使用底层接口取消双边报价请求 |
| query_history | 根据名称使用底层接口查询历史 K 线信息 |
| close | 停止各引擎和各底层接口 |

从表 3-2 中不难看出，MainEngine 是一个聚合了各底层接口（gateway）和上层应用（app）的类，起到了承上启下的作用，并且由于其支持添加多个底层接口与上层应用，MainEngine 支持的交互过程变得十分灵活，读者可以很方便地在 MainEngine 中定制自己的应用与接口的交互逻辑。

### 2. 事件引擎 EventEngine

事件引擎 EventEngine 负责根据事件的类型将其分发到对应的处理函数（handler）上处理，目前 EventEngine 支持的事件类型如表 3-3 所示。

**表 3-3　EventEngine 支持的事件类型**

| 类　型 | 说　明 |
| --- | --- |
| EVENT_TIMER | 定时任务 |
| EVENT_TICK | 收到 tick 数据的事件 |
| EVENT_TRADE | 收到成交回报的事件 |
| EVENT_ORDER | 收到下单回报的事件 |
| EVENT_POSITION | 收到持仓更改消息的事件 |
| EVENT_QUOTE | 收到双边报价回报的事件 |
| EVENT_CONTRACT | 收到合约信息查询回报的事件 |
| EVENT_LOG | 收到写日志请求的事件 |

在订单管理引擎 OmsEngine 中（主引擎 MainEngine 默认注册的引擎之一）完成了大部分不同类型事件与其处理函数的绑定，代码如下：

```
//ch3/trader_engine.py
def register_event(self) -> None:
    """"""
    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)
```

当上层应用或底层接口产生了表 3-3 中的类型事件时，通过事件引擎 EventEngine 的 put 方法将待处理事件 Event 放入 EventEngine 的队列_queue 中，与此同时，EventEngine 中有一个启动的单独线程不断地从队列_queue 中获取待处理事件并传给 handler 进行处理，代码如下：

```
//ch3/event_engine.py
…
#将待处理事件放入队列
def put(self, event: Event) -> None:
    """
    Put an event object into event queue.
    """
    self._queue.put(event)
…
…
#不断地从队列中获取事件
def _run(self) -> None:
    """
    Get event from queue and then process it.
    """
    while self._active:
        try:
            event: Event = self._queue.get(block=True, timeout=1)
            self._process(event)
        except Empty:
            pass
…
#将不同类型的事件分发给不同的 handler
def _process(self, event: Event) -> None:
    """
    First distribute event to those handlers registered listening
    to this type.

    Then distribute event to those general handlers which listens
    to all types.
    """
    if event.type in self._handlers:
        [handler(event) for handler in self._handlers[event.type]]

    if self._general_handlers:
        [handler(event) for handler in self._general_handlers]
…
```

3. 数据引擎 BaseDatabase

vn.py 支持很多数据库作为后端，其默认使用 SQLite，除此之外目前其还支持 MySQL、PostgreSQL、DolphinDB、Arctic、TDengine、TimescaleDB、MongoDB、InfluxDB 和 LevelDB。由于 vn.py 目前以分模块设计，所以具体数据库对应的模块都位于具体的名为 vnpy_[数据库名]的包中（如 vnpy_mysql）。

本节以数据引擎的基类 BaseDatabase 为例讲解其包含的方法，不涉及具体某个数据库的执行逻辑。表 3-4 列出了 BaseDatabase 中包含的方法及其说明。

**表 3-4 BaseDatabase 包含的方法及说明**

| 方　法 | 说　明 |
|---|---|
| save_bar_data | 保存 K 线数据 |
| save_tick_data | 保存 tick 数据 |
| load_bar_data | 读取某品种特定时间与周期内的 K 线数据 |
| load_tick_data | 读取某品种特定时间内的 tick 数据 |
| delete_bar_data | 删除某品种特定周期的 K 线数据 |
| delete_tick_data | 删除某品种的 tick 数据 |
| get_bar_overview | 获取所有品种的 K 线统计数据 |
| get_tick_overview | 获取所有品种的 tick 统计数据 |

从表 3-4 可以看出，数据引擎主要负责对 K 线与 tick 数据进行增、删、查、改操作。

### 3.1.3 上层应用

vn.py 文件中的上层应用仅与中层引擎进行交互，通过引擎调用底层接口的具体方法完成交易链路与回报链路的通信。vn.py 文件中的上层应用众多，本节不举例进行讲解，本章的 3.4 节~3.14 节将对不同的上层应用进行更为详细的讲解。

## 3.2 vn.py 文件中的交易接口

如 3.1.1 节中所述，vn.py 支持众多交易接口，本节以 CTP 和 UFT 接口为例进行讲解。

### 3.2.1 CTP 接口

综合交易平台（ComprehensiveTransactionPlatform，CTP）是上海期货信息技术有限公司开发的一套柜面系统。交易者通过 CTP 可以将报文发送给各个交易所，在 vnpy_ctp 模块中，可以看到其包含一个 api 文件夹和一个 gateway 文件夹，其中 api 文件夹中包含 CTP 系统的一些库文件，其用 C++编写或以 dll 或 lib 文件出现。

以 api 文件夹下的 include/ctp 文件夹中的文件为例，其中的 C++头文件 ThostFtdcMdApi.h

包含获取与行情相关的指令，C++头文件 ThostFtdcTraderApi.h 包含与交易相关的指令，C++头文件ThostFtdcUserApiDataType.h包含所有用到的数据类型，C++头文件ThostFtdcUserApiStruct.h包含了所有用到的数据结构。

在 api/libs 下包含两个 lib 文件 thostmduserapi_se.lib 和 thosttraderapi_se.lib，其分别是行情部分和交易部分的静态链接库，由于 lib 文件无法直接打开，因此用户在进行 CTP 编程时只需关注 C++头文件中定义的方法。

在 gateway 文件夹下有一个名为 ctp_gateway.py 的文件，这个文件对 CTP 的 C++ API 进行了 Python 层面的接口封装，通过 pybind11 使 Python 与 C++之间的数据对象进行互相转换。在此不过多分析 C++层面的编程细节，主要分析 ctp_gateway.py 文件中的 Python API 封装。

下面选取了部分 ctp_gateway.py 文件中的代码：

```
    //ch3/ctp_gateway.py
    #委托状态映射
    STATUS_CTP2VT: Dict[str, Status] = {
        THOST_FTDC_OST_NoTradeQueueing: Status.NOTTRADED,
        THOST_FTDC_OST_PartTradedQueueing: Status.PARTTRADED,
        THOST_FTDC_OST_AllTraded: Status.ALLTRADED,
        THOST_FTDC_OST_Canceled: Status.CANCELLED,
        THOST_FTDC_OST_Unknown: Status.SUBMITTING
    }

    #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,
        "GFEX": Exchange.GFEX
    }

    #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
    }
```

以上代码主要定义了 CTP 接口使用的数据字典到 vn.py 中使用的数据字典之间的映射，包含下单、行情、产品类型等的映射。经过 vn.py 的映射，不同接口的行情或从 vn.py 下单到不同接口的数据类型都会被转换为 vn.py 内部使用的统一格式。

在 ctp_gateway.py 文件中包含了一个核心类 CtpGateway，其封装了 CTP 接口与 vn.py 之间交互的各种方法，代码如下：

```
    //ch3/ctp_gateway.py
    class CtpGateway(BaseGateway):
```

```
    """
    VeighNa 用于对接期货 CTP 柜台的交易接口
    """

    default_name: str = "CTP"

    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": ""
    }

    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)

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

        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

        self.td_api.connect(td_address, userid, password, brokerid, auth_
code, appid)
        self.md_api.connect(md_address, userid, password, brokerid)

        self.init_query()

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

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

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

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

    def query_position(self) -> None:
        """查询持仓"""
        self.td_api.query_position()

    def close(self) -> None:
        """关闭接口"""
        self.td_api.close()
        self.md_api.close()

    def write_error(self, msg: str, error: dict) -> None:
        """输出错误信息日志"""
        error_id: int = error["ErrorID"]
        error_msg: str = error["ErrorMsg"]
        msg: str = f"{msg}，代码：{error_id}，信息：{error_msg}"
        self.write_log(msg)
```

```
    def process_timer_event(self, event) -> None:
        """定时事件处理"""
        self.count += 1
        if self.count < 2:
            return
        self.count = 0

        func = self.query_functions.pop(0)
        func()
        self.query_functions.append(func)

        self.md_api.update_date()

    def init_query(self) -> None:
        """初始化查询任务"""
        self.count: int = 0
        self.query_functions: list = [self.query_account, self.query_position]
        self.event_engine.register(EVENT_TIMER, self.process_timer_event)
```

代码中包含连接 CTP 柜台的配置，包括行情、交易服务器地址（以 TCP 进行连接）、登录账号和密码等信息。不同的行为，例如订阅行情（subscribe）、委托下单（send_order）须调用行情 API 或交易 API 的具体接口完成。

读者容易在 ctp_gateway.py 文件中看到 CtpMdApi 和 CtpTdApi 类，其分别对应着 CTP 接口的行情和交易类，以 CtpMdApi 为例，代码如下：

```
//ch3/ctp_gateway.py
class CtpMdApi(MdApi):
    """"""

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

        self.gateway: CtpGateway = gateway
        self.gateway_name: str = gateway.gateway_name

        self.reqid: int = 0

        self.connect_status: bool = False
        self.login_status: bool = False
        self.subscribed: set = set()
```

```
            self.userid: str = ""
            self.password: str = ""
            self.brokerid: str = ""

            self.current_date: str = datetime.now().strftime("%Y%m%d")

        def onFrontConnected(self) -> None:
            """服务器连接成功回报"""
            self.gateway.write_log("行情服务器连接成功")
            self.login()

        def onFrontDisconnected(self, reason: int) -> None:
            """服务器连接断开回报"""
            self.login_status = False
            self.gateway.write_log(f"行情服务器连接断开，原因{reason}")

        def onRspUserLogin(self, data: dict, error: dict, reqid: int, last: bool)
-> None:
            """用户登录请求回报"""
            if not error["ErrorID"]:
                self.login_status = True
                self.gateway.write_log("行情服务器登录成功")

                for symbol in self.subscribed:
                    self.subscribeMarketData(symbol)
            else:
                self.gateway.write_error("行情服务器登录失败", error)

        def onRspError(self, error: dict, reqid: int, last: bool) -> None:
            """请求报错回报"""
            self.gateway.write_error("行情接口报错", error)

        def onRspSubMarketData(self, data: dict, error: dict, reqid: int, last:
bool) -> None:
            """订阅行情回报"""
            if not error or not error["ErrorID"]:
                return

            self.gateway.write_error("行情订阅失败", error)

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

        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 = dt.replace(tzinfo=CHINA_TZ)

        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
        )

        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"]

        self.gateway.on_tick(tick)

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

        #禁止重复发起连接，会导致异常，从而崩溃
        if not self.connect_status:
            path: Path = get_folder_path(self.gateway_name.lower())
            self.createFtdcMdApi((str(path) + "\\Md").encode("GBK"))

            self.registerFront(address)
            self.init()

            self.connect_status = True

    def login(self) -> None:
        """用户登录"""
        ctp_req: dict = {
            "UserID": self.userid,
```

```
            "Password": self.password,
            "BrokerID": self.brokerid
        }

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

    def subscribe(self, req: SubscribeRequest) -> None:
        """订阅行情"""
        if self.login_status:
            self.subscribeMarketData(req.symbol)
        self.subscribed.add(req.symbol)

    def close(self) -> None:
        """关闭连接"""
        if self.connect_status:
            self.exit()

    def update_date(self) -> None:
        """更新当前日期"""
        self.current_date = datetime.now().strftime("%Y%m%d")
```

CtpMdApi 类中主要包含三类方法（除构造函数__init__以外），一类是如上代码所示的login、connect、close 等方法的 CTP 行情接口连接状态管理函数；其次是像代码中的 subscribe、update_date 等方法，用于管理类中的成员变量值；剩下的以 on 开头的函数则是回调函数，其继承了 MdApi 中的方法，在行情接口推送了已订阅的标的行情后，以函数 onRtnDepthMarketData 为例，对于底层 C++接口推送来的逐笔行情，CtpMdApi 将其转换为 vn.py 文件中定义的 TickData 对象并最终通过 BaseGateway 类（CtpGateway 的父类）中的 on_tick 方法把 tick 行情通过引擎进行推送。其他的回调函数的执行逻辑与 onRtnDepthMarketData 方法类似，读者可以自行学习。

### 3.2.2 UFT 接口

UFT 是一个由恒生开发的极速交易系统，其依托于统一接入的系统 UFX，可以认为 UFT 是一个快速交易的订单子系统，其专门负责进行极速交易，而 UFX 的后台业务系统可以基于 UFT 实现。

vn.py 的 UFT 接口相关文件位于 vnpy_uft 项目中。与 CTP 接口类似，UFT 接口同样也维护了一套自身的行情与交易服务器，与 CTP 接口不同的是，UFT 接口有一套不同的登录所需的字段，代码如下：

```
//ch3/uft_gateway.py
def connect(self, setting: dict) -> None:
```

```
        """连接交易接口"""
        userid: str = setting["用户名"]
        password: str = setting["密码"]
        md_address: str = setting["行情服务器"]
        td_address: str = setting["交易服务器"]
        self.server: str = setting["服务器类型"]
        appid: str = setting["产品名称"]
        auth_code: str = setting["授权编码"]
        application_type: str = setting["委托类型"]

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

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

        license_path: Path = TRADER_DIR.joinpath("license.dat")

        if license_path.exists():
            server_license: str = str(license_path)
        else:
            if self.server == "期货":
                server_license: str = FUTURES_LICENSE
            else:
                server_license: str = OPTION_LICENSE

        …
    )
```

从代码中可以看出，与 CTP 接口连接不同，UFT 接口的 connect 函数需要额外的服务器类型、委托类型字段，并需要根据不同的服务器类型验证证书。剩余代码与处理逻辑与 CTP 接口基本类似，在本节不再赘述。

## 3.3 vn.py 文件中的数据库

在 3.1.2 节中的第 3 部分已经简要介绍了 vn.py 支持的数据库后端，其默认的数据库为 SQLite，基于文件系统的 SQLite 单文件最大支持 128TB 数据，并且其在索引列上的读写性能并不差，因此本节将讲解 SQLite 作为 vn.py 数据库后端的应用。

vn.py 文件中与 SQLite 数据库交互的模块叫作 vnpy_sqlite，使用 vnpy_sqlite 会在用户目录下的.vntrader 文件夹内生成一个 database.db 文件，vn.py 产生的 K 线和 tick 数据都存储于其中。核心类为 SqliteDatabase，SQLite 中的 ORM 继承了 peewee 包中的 Model 进行实现。

它的基类为 3.1.2 节中所介绍的 BaseDatabase，因此其实现了表 3-4 中方法的具体逻辑，下面将逐一分析这些方法的实现。

1. save_bar_data

save_bar_data 的具体逻辑代码如下：

```
//ch3/sqlite_database.py
def save_bar_data(self, bars: List[BarData], stream: bool = False) -> bool:
    """保存 K 线数据"""
    #读取主键参数
    bar: BarData = bars[0]
    symbol: str = bar.symbol
    exchange: Exchange = bar.exchange
    interval: Interval = bar.interval

    #将 BarData 数据转换为字典，并调整时区
    data: list = []

    for bar in bars:
        bar.datetime = convert_tz(bar.datetime)

        d: dict = bar.__dict__
        d["exchange"] = d["exchange"].value
        d["interval"] = d["interval"].value
        d.pop("gateway_name")
        d.pop("vt_symbol")3
        data.append(d)

    #使用 upsert 操作将数据更新到数据库中
    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
    overview: DbBarOverview = DbBarOverview.get_or_none(
        DbBarOverview.symbol == symbol,
        DbBarOverview.exchange == exchange.value,
        DbBarOverview.interval == interval.value,
    )

    if not overview:
        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)
        elif stream:
            overview.end = bars[-1].datetime
            overview.count += len(bars)
        else:
            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()

        overview.save()

        return True
```

其中，DbBarData 类的定义如下：

```
//ch3/sqlite_database.py
class DbBarData(Model):
    """K 线数据表映射对象"""

    id: AutoField = 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: PeeweeSqliteDatabase = db
        indexes: tuple = ((("symbol", "exchange", "interval", "datetime"), True),)
```

从 DbBarData 的代码可以看出，其对于一个 K 线对象不同的数据域进行了定义，例如标的代码、时间、OHLC 等。在 save_bar_data 中，其取出一个待存储的 K 线对象，并获取了待存储对象的相同数据（save_bar_data 接收的一批 K 线对象是按照同一标的进行聚合的）。

接着将每个 K 线对象转换为字典对象，并使用 upsert 方法将字典对象按照列名存入 SQLite 数据库。除此之外，save_bar_data 还在数据库中维护了一个 K 线汇总对象，便于获取与 K 线相关的统计信息。

2. save_tick_data

与 save_bar_data 类似，save_tick_data 的代码如下：

```
//ch3/sqlite_database.py
def save_tick_data(self, ticks: List[TickData], stream: bool = False) -> bool:
    """保存 tick 数据"""
    #读取主键参数
    tick: TickData = ticks[0]
    symbol: str = tick.symbol
    exchange: Exchange = tick.exchange

    #将 TickData 数据转换为字典，并调整时区
    data: list = []

    for tick in ticks:
        tick.datetime = convert_tz(tick.datetime)

        d: dict = tick.__dict__
        d["exchange"] = d["exchange"].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
    with self.db.atomic():
        for c in chunked(data, 10):
            DbTickData.insert_many(c).on_conflict_replace().execute()

    #更新 tick 汇总数据
    overview: DbTickOverview = DbTickOverview.get_or_none(
        DbTickOverview.symbol == symbol,
        DbTickOverview.exchange == exchange.value,
    )
```

```
    if not overview:
        overview: DbTickOverview = DbTickOverview()
        overview.symbol = symbol
        overview.exchange = exchange.value
        overview.start = ticks[0].datetime
        overview.end = ticks[-1].datetime
        overview.count = len(ticks)
    elif stream:
        overview.end = ticks[-1].datetime
        overview.count += len(ticks)
    else:
        overview.start = min(ticks[0].datetime, overview.start)
        overview.end = max(ticks[-1].datetime, overview.end)

        s: ModelSelect = DbTickData.select().where(
            (DbTickData.symbol == symbol)
            & (DbTickData.exchange == exchange.value)
        )
        overview.count = s.count()

    overview.save()

    return True
```

save_tick_data 的代码结构与 save_bar_data 相同，大致也分为获取 tick 数据的共同部分数据、存储 tick 数据（tick 数据的域定义与 K 线定义自然不同）与修改或创建 tick 数据对象。

### 3. load_bar_data

load_bar_data 根据传入的筛选条件（标的代码、交易所、K 线周期、起止日期）返回数据库中的 K 线数据，其代码如下：

```
//ch3/sqlite_database.py

def load_bar_data(
    self,
    symbol: str,
    exchange: Exchange,
    interval: Interval,
    start: datetime,
    end: datetime
) -> List[BarData]:
    """读取 K 线数据"""
    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)
        )

        bars: List[BarData] = []
        for db_bar in s:
            bar: BarData = 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)

        return bars
```

使用 ORM 对象 SELECT 后，其还对筛选出的数据进行了数据类型转换，使其返回的结果为 vn.py 文件中的 BarData 对象列表，这么做的好处是能统一不同数据库同一 API 的返回值，便于上层应用的进一步处理。

4. load_tick_data

与 load_bar_data 类似，load_tick_data 对数据库中的 tick 数据按照筛选条件进行获取并返回一个 TickData 对象的列表，代码如下：

```
    //ch3/sqlite_database.py
    def load_tick_data(
        self,
        symbol: str,
        exchange: Exchange,
        start: datetime,
        end: datetime
```