ctaTemplate类所在路径vnpy/trader/app/ctaStrategy/ctaTemplate.py

```
'''
本文件包含了CTA引擎中的策略开发用模板，开发策略时需要继承CtaTemplate类。
'''

import numpy as np
import pandas as  pd
from datetime import datetime,timedelta,time
import talib
import requests
from collections import defaultdict
from vnpy.trader.vtConstant import *
from vnpy.trader.vtObject import VtBarData
from vnpy.trader.vtUtility import BarGenerator, ArrayManager
from vnpy.trader.utils.email import mail

from .ctaBase import *

########################################################################
class CtaTemplate(object):
    """CTA策略模板"""

    # 策略类的名称和作者
    className = 'CtaTemplate'
    author = EMPTY_UNICODE

    # MongoDB数据库的名称，K线数据库默认为1分钟
    tickDbName = TICK_DB_NAME
    barDbName = MINUTE_DB_NAME

    # 策略的基本参数
    name = EMPTY_UNICODE  # 策略实例名称
    vtSymbol = EMPTY_STRING        # 交易的合约vt系统代码
    productClass = EMPTY_STRING  # 产品类型（只有IB接口需要）
    currency = EMPTY_STRING  # 货币（只有IB接口需要）

    # 策略的基本变量，由引擎管理
    inited = False  # 是否进行了初始化
    trading = False  # 是否启动交易，由引擎管理
    symbolList = []  # 策略的标的列表

    # 参数列表，保存了参数的名称,列表里的内容最后会显示在qt界面上
    paramList = ['name',
                 'className',
                 'author',
                 'symbolList']

    # 变量列表，保存了变量的名称，列表里的内容最后会显示在qt界面上
    varList = ['inited',
               'trading',
               'posDict']

    # 同步列表，保存了需要保存到数据库的变量名称
    syncList = ['posDict',
                'eveningDict',
                'accountDict']
```

```
    # ----------------------------------------------------------------------
    def __init__(self, ctaEngine, setting):
        """Constructor"""
        self.ctaEngine = ctaEngine #
        self.posDict = {}#仓位字典，形式如同{"eosusd_long":1,"eosusd_short":1}表明多空都有
，自动在引擎内部生成
        self.eveningDict = {}#保证金字典，一般不用
        self.accountDict = {}#账户字典，暂时没用上还不知道干啥的
        # 设置策略的参数
```

通过setting里的key和value自动在类内部创建属性，通过往self.__dict__[key]=value的方式 不懂的话可以自行百度self.__dict__

```
        if setting:
            d = self.__dict__
            for key in self.paramList:
                if key in setting:
                    d[key] = setting[key]

        # self.posDict = {}
        # self.eveningDict = {}
```

NotImplementedError可以实现c++的pure virtual的功能，就是说子类必须继承父类的方法并加以实现，否则就报错 引擎上会有四个功能键 初始化->onInit 启动->onStart 停止->onStop 恢复->onRestore

```
    def onInit(self):
        """初始化策略（必须由用户继承实现）"""
        raise NotImplementedError

    def onStart(self):
        """启动策略（必须由用户继承实现）"""
        raise NotImplementedError

    def onStop(self):
        """停止策略（必须由用户继承实现）"""
        raise NotImplementedError

    def onRestore(self):
        """恢复策略（必须由用户继承实现）"""
        raise NotImplementedError
```

程序通过websocket与交易所建立持久化连接，交易所会推送给tick数据 引擎内部会把tick数据处理成固定的数据格式 tick的数据格式如下

```
    class VtTickData(VtBaseData):
    """Tick行情数据类"""

    def __init__(self):
        """Constructor"""
        super(VtTickData, self).__init__()

        # 代码相关
```

```
        self.symbol = EMPTY_STRING              # 合约代码
        self.exchange = EMPTY_STRING            # 交易所代码
        self.vtSymbol = EMPTY_STRING            # 合约在vt系统中的唯一代码，通常是 合约代码
.交易所代码

        # 成交数据
        self.lastPrice = EMPTY_FLOAT            # 最新成交价
        self.lastVolume = EMPTY_INT             # 最新成交量
        self.volume = EMPTY_INT                 # 今天总成交量
        self.openInterest = EMPTY_INT           # 持仓量
        self.time = EMPTY_STRING                # 时间 11:20:56.5
        self.date = EMPTY_STRING                # 日期 20151009
        self.datetime = None                    # python的datetime时间对象

        # 常规行情
        self.openPrice = EMPTY_FLOAT            # 今日开盘价
        self.highPrice = EMPTY_FLOAT            # 今日最高价
        self.lowPrice = EMPTY_FLOAT             # 今日最低价
        self.preClosePrice = EMPTY_FLOAT

        self.upperLimit = EMPTY_FLOAT           # 涨停价
        self.lowerLimit = EMPTY_FLOAT           # 跌停价

        # 五档行情
        self.bidPrice1 = EMPTY_FLOAT
        self.bidPrice2 = EMPTY_FLOAT
        self.bidPrice3 = EMPTY_FLOAT
        self.bidPrice4 = EMPTY_FLOAT
        self.bidPrice5 = EMPTY_FLOAT
        self.askPrice1 = EMPTY_FLOAT
        self.askPrice2 = EMPTY_FLOAT
        self.askPrice3 = EMPTY_FLOAT
        self.askPrice4 = EMPTY_FLOAT
        self.askPrice5 = EMPTY_FLOAT
        self.bidVolume1 = EMPTY_INT
        self.bidVolume2 = EMPTY_INT
        self.bidVolume3 = EMPTY_INT
        self.bidVolume4 = EMPTY_INT
        self.bidVolume5 = EMPTY_INT
        self.askVolume1 = EMPTY_INT
        self.askVolume2 = EMPTY_INT
        self.askVolume3 = EMPTY_INT
        self.askVolume4 = EMPTY_INT
        self.askVolume5 = EMPTY_INT
```

会把tick传入如下函数，在策略里继承实现

```
    def onTick(self, tick):
        """收到行情TICK推送（必须由用户继承实现）"""
        raise NotImplementedError
```

onOrder会在订单发生变化的时候，进行推送，通过websocket传来 websocket会订阅很多频道，订单这个单独有一个频道 行情数据也有单独的频道，传来的是json格式，内部处理成order的数据格式， 格式如下

```
    class VtOrderData(VtBaseData):
    """订单数据类"""
```

```
    #----------------------------------------------------------------------
    def __init__(self):
        """Constructor"""
        super(VtOrderData, self).__init__()

        # 代码编号相关
        self.symbol = EMPTY_STRING              # 合约代码
        self.exchange = EMPTY_STRING            # 交易所代码
        self.vtSymbol = EMPTY_STRING  # 索引，统一格式：f"{symbol}.{exchange}"

        self.orderID = EMPTY_STRING             # 订单编号 gateway内部自己生成的编号
        self.vtOrderID = EMPTY_STRING  # 索引，统一格式：f"{gatewayName}.{orderId}"

        # 报单相关
        self.direction = EMPTY_UNICODE          # 报单方向
        self.offset = EMPTY_UNICODE             # 报单开平仓
        self.price = EMPTY_FLOAT                # 报单价格
        self.totalVolume = EMPTY_INT            # 报单总数量
        self.tradedVolume = EMPTY_INT           # 报单成交数量
        self.status = EMPTY_UNICODE             # 报单状态

        self.orderTime = EMPTY_STRING           # 发单时间
        self.cancelTime = EMPTY_STRING          # 撤单时间

        # CTP/LTS相关
        self.frontID = EMPTY_INT                # 前置机编号
        self.sessionID = EMPTY_INT
```

交易所会发来一个orderID作为唯一标志，传递某一个订单的状态order.status 有“未成交”，“部分成交”，“全部成交”，“已撤销”，“未知”（未知是引擎内部处理的，用于应付掉线情况） vtOrderID是引擎内部根据orderID维护的一个列表，也是唯一的，按从1开始的顺序递增，方便处理， 所以一般在策略里都使用order.vtOrderID order.vtSymbol是你交易的品种 order.direction是你下的这单的方向,多或空 order.offset是你下的这单是开仓还是平仓 buy->“多” “开仓” sell->“空” “平仓” short->“空” “开仓” cover->“多” “平仓” order.price报单价格 order.totalVolume报单总数量 order.tradedVolume成交数量

```
    def onOrder(self, order):
        """收到委托变化推送（必须由用户继承实现）"""
        raise NotImplementedError
```

如果是完全成交或部分成交的order.status会被推到onTrade里，顺序在onOrder之后， 所以如果要在onTrade里写东西，最好不要写交易逻辑，不然会要先运行完onOrder再过来，速度跟不上 或者onOrder自己实现异步的，不阻塞到onTrade。 trade的数据格式如下

```
    class VtTradeData(VtBaseData):
    """
    成交数据类
    一般来说，一个VtOrderData可能对应多个VtTradeData：一个订单可能多次部分成交
    """

    #----------------------------------------------------------------------
    def __init__(self):
        """Constructor"""
        super(VtTradeData, self).__init__()
```

```
        # 代码编号相关
        self.symbol = EMPTY_STRING              # 合约代码
        self.exchange = EMPTY_STRING            # 交易所代码
        self.vtSymbol = EMPTY_STRING            # 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.tradeID = EMPTY_STRING  # 成交编号 gateway内部自己生成的编号
        self.vtTradeID = EMPTY_STRING           # 成交在vt系统中的唯一编号，通常是 Gateway
名.成交编号

        self.orderID = EMPTY_STRING             # 订单编号
        self.vtOrderID = EMPTY_STRING           # 订单在vt系统中的唯一编号，通常是 Gateway
名.订单编号

        # 成交相关
        self.direction = EMPTY_UNICODE          # 成交方向
        self.offset = EMPTY_UNICODE             # 成交开平仓
        self.price = EMPTY_FLOAT                # 成交价格
        self.volume = EMPTY_INT                 # 成交数量
        self.tradeTime = EMPTY_STRING           # 成交时间
```

部分成交和全部成交会被推送到onTrade函数里

```
    def onTrade(self, trade):
        """收到成交推送（必须由用户继承实现）"""
        raise NotImplementedError
```

bar就是传统的k线，数据格式如下

```
    class VtBarData(VtBaseData):
    """K线数据"""

    #----------------------------------------------------------------------
    def __init__(self):
        """Constructor"""
        super(VtBarData, self).__init__()

        self.vtSymbol = EMPTY_STRING        # vt系统代码
        self.symbol = EMPTY_STRING          # 代码
        self.exchange = EMPTY_STRING        # 交易所

        self.open = EMPTY_FLOAT             # OHLC
        self.high = EMPTY_FLOAT
        self.low = EMPTY_FLOAT
        self.close = EMPTY_FLOAT

        self.date = EMPTY_STRING            # bar开始的时间，日期
        self.time = EMPTY_STRING            # 时间
        self.datetime = None                # python的datetime时间对象

        self.volume = EMPTY_INT             # 成交量
        self.openInterest = EMPTY_INT       # 持仓量
        self.interval = EMPTY_UNICODE       # K线周期
```

onBar默认是一分钟推送一次，第一次推送在第一个整分钟。

```
    def onBar(self, bar):
        """收到Bar推送（必须由用户继承实现）"""
        raise NotImplementedError
```

停止单，目前还没用过，等用过再来补充

```
    def onStopOrder(self, so):
        """收到停止单推送（必须由用户继承实现）"""
        raise NotImplementedError
```

buy，sell，short，cover是四种下单方式，在onOrder里介绍了 这里要说明的是这四种下单方式的函数会返回一个list，list里只有一个元素，这个元素是vtOrderID

```
    def buy(self, vtSymbol, price, volume, priceType = PRICETYPE_LIMITPRICE,
    stop=False):
        """买开"""
        return self.sendOrder(CTAORDER_BUY, vtSymbol, price, volume,
        priceType, stop)

        # ----------------------------------------------------------------------
    def sell(self, vtSymbol, price, volume, priceType = PRICETYPE_LIMITPRICE,
    stop=False):
        """卖平"""
        return self.sendOrder(CTAORDER_SELL, vtSymbol, price, volume,
        priceType, stop)

        # ----------------------------------------------------------------------
    def short(self, vtSymbol, price, volume, priceType =
    PRICETYPE_LIMITPRICE, stop=False):
        """卖开"""
        return self.sendOrder(CTAORDER_SHORT, vtSymbol, price, volume,
        priceType, stop)

        # ----------------------------------------------------------------------
    def cover(self, vtSymbol, price, volume, priceType =
    PRICETYPE_LIMITPRICE, stop=False):
        """买平"""
        return self.sendOrder(CTAORDER_COVER, vtSymbol, price, volume,
        priceType, stop)
```

buy，sell，short，cover调用此函数发单

```
    def sendOrder(self, orderType, vtSymbol, price, volume, priceType =
    PRICETYPE_LIMITPRICE, stop=False):
        """发送委托"""
        if self.trading:
            # 如果stop为True，则意味着发本地停止单
            if stop:
                vtOrderIDList = self.ctaEngine.sendStopOrder(vtSymbol, orderType, pric
e, volume, priceType, self)
            else:
                vtOrderIDList = self.ctaEngine.sendOrder(vtSymbol, orderType, price, v
olume, priceType, self)
            return vtOrderIDList
```

```
        else:
            # 交易停止时发单返回空字符串
            return []
```

cancelOrder取消订单，传入参数是vtOrderID，每次发单buy，sell，cover，short返回值都是一个列表 列表里只有一个元素就是vtOrderID，把那个vtOrderID传入cancelOrder函数，完成撤单

```
    def cancelOrder(self, vtOrderID):
        """撤单"""
        # 如果发单号为空字符串，则不进行后续操作
        if not vtOrderID:
            return

        if STOPORDERPREFIX in vtOrderID:
            self.ctaEngine.cancelStopOrder(vtOrderID)
        else:
            self.ctaEngine.cancelOrder(vtOrderID)
```

cancelAll是循环vtOrderID去发撤单，所以为了程序的精细化处理一般不用cancelAll

```
    def cancelAll(self):
        """全部撤单"""
        self.ctaEngine.cancelAll(self.name)
```

如上，暂没有使用过StopOrder

```
    def cancelAllStopOrder(self):
        self.ctaEngine.cancelAllStopOrder(self.name)
```

batchCancelOrder批量撤单，如果单特别多，使用cancelOrder会慢，所以用batchCancelOrder

```
    def batchCancelOrder(self,vtOrderIDList):
        if len(vtOrderIDList)>5:
            self.writeCtaLog(u'策略发送批量撤单委托失败，单量超过5张')
            return
        self.ctaEngine.batchCancelOrder(vtOrderIDList)
```

回测中使用，但实盘没有tick可以load所以不loadtick

```
    def loadTick(self, hours=1):
        """读取tick数据"""
        return self.ctaEngine.loadTick(self.tickDbName, self.symbolList,
        hours)
```

加载bar数据

```
    def loadBar(self, hours=1):
        """读取bar数据"""
        return self.ctaEngine.loadBar(self.barDbName, self.symbolList, hours)
```

非常重要的功能，会把日志打下来，方便复盘，但记住别放在交易逻辑之前，尽量放在之后

```
    def writeCtaLog(self, content):
        """记录CTA日志"""
        content = self.name + ':' + content
        self.ctaEngine.writeCtaLog(content)
```

loadHistoryBar可以在实盘中通过交易所提供的restful Api获取一定数量的bar，

```
    def loadHistoryBar(self,vtSymbol,type_,size= None,since = None):
        """策略开始前下载历史数据"""

        if type_ in ["1min","5min","15min","30min","60min","120min","240min",
        "360min","480min","1day","1week","1month"]:
            data = self.ctaEngine.loadHistoryBar(vtSymbol,type_,size,since)
            lastbar = data[-1]
            if 'min' in type_:
                minute = int(type_[:-3])

            if datetime.now() < (lastbar.datetime + timedelta(seconds =
            60*minute)):
                self.writeCtaLog(u'加载历史数据抛弃最后一个非完整K线，频率%s，时
                间%s'%(type_, lastbar.datetime))
                data = data[:-1]

            return data

        else:
            self.writeCtaLog(
                u'下载历史数据参数错误，请参考以下参数["1min","5min","15min",
                "30min","60min","120min","240min","360min","480min","1day",
                "1week","1month"]，同时size建议不大于2000')
            return
```

可以给自己发邮件，在实盘中启用

```
    def mail(self,my_context):
        """邮件发送模块"""
        if self.ctaEngine.engineType == ENGINETYPE_BACKTESTING:
            pass
        else:
            msg = mail(my_context,self)
            self.writeCtaLog('%s'%msg)
```

注册一个函数onXmimBar写函数名字 xmin写分钟数，必须是整分钟，比如5，15 这种，可以参考行情软件上的k线时间 然后会自动生成两个变量通过setattr(key,value) 一个是BarGenerator对象的 一个是ArrayManager对象的 名字是

```
  # if xmin:
    #         variable = "bg%sDict"%xmin
    #         variable2 = "am%sDict"%xmin
    #     else:
    #         variable = "bgDict"
    #         variable2 = "amDict"
```

```
    def generateBarDict(self, onBar, xmin=0, onXminBar=None, size = 100,
    alignment='sharp', marketClose = (23,59)):
        if xmin:
            variable = "bg%sDict"%xmin
            variable2 = "am%sDict"%xmin
        else:
            variable = "bgDict"
            variable2 = "amDict"
        bgDict= {
            sym: BarGenerator(onBar,xmin,onXminBar, alignment=alignment,
            marketClose=marketClose)
            for sym in self.symbolList }

        amDict = {
            sym: ArrayManager(size)
            for sym in self.symbolList }

        setattr(self, variable, bgDict)
        setattr(self, variable2, amDict)
```

秒级别的bar，类似分钟级别的参数，一般是给高频，而且秒级别的噪声很大，一般需要很长的研究过程

```
    def generateHFBar(self,xSecond,size = 60):
        self.hfDict = {sym: BarGenerator(self.onHFBar,xSecond = xSecond)
                       for sym in self.symbolList}
        self.amhfDict = {sym: ArrayManager(size) for sym in self.symbolList}
```