# 第 5 章

# 基于指标的交易策略

从本章开始将会为读者介绍基于指标的交易策略，指标通常由人为启发式地提出，当指标到达某一特定值时应该做出不同的决策。在介绍具体的指标思想与其计算方法之前，笔者将介绍交易策略的整体框架，该框架在后面几章的内容中同样适用。

## 5.1 交易策略框架


在 vn.py 文件中，交易策略（仅代指用于 3.5 节中自动交易模块的策略，下同）的基类为 CtaTemplate，其定义代码如下：

```
//ch5/cta_template.py
class CtaTemplate(ABC):
    """"""

    author: str = ""
    parameters: list = []
    variables: list = []

    def __init__(
        self,
        cta_engine: Any,
        strategy_name: str,
        vt_symbol: str,
        setting: dict,
    ) -> None:
        """"""
        self.cta_engine: Any = cta_engine
        self.strategy_name: str = strategy_name
        self.vt_symbol: str = vt_symbol

        self.inited: bool = False
        self.trading: bool = False
```

```
        self.pos: int = 0

        #Copy a new variables list here to avoid duplicate insert when multiple
        #strategy instances are created with the same strategy class.
        self.variables = copy(self.variables)
        self.variables.insert(0, "inited")
        self.variables.insert(1, "trading")
        self.variables.insert(2, "pos")

        self.update_setting(setting)
    …
    @virtual
    def on_init(self) -> None:
        """
        Callback when strategy is inited.
        """
        pass

    @virtual
    def on_start(self) -> None:
        """
        Callback when strategy is started.
        """
        pass

    @virtual
    def on_stop(self) -> None:
        """
        Callback when strategy is stopped.
        """
        pass

    @virtual
    def on_tick(self, tick: TickData) -> None:
        """
        Callback of new tick data update.
        """
        pass

    @virtual
    def on_bar(self, bar: BarData) -> None:
        """
        Callback of new bar data update.
```

```
        """
        pass

    @virtual
    def on_trade(self, trade: TradeData) -> None:
        """
        Callback of new trade data update.
        """
        pass

    @virtual
    def on_order(self, order: OrderData) -> None:
        """
        Callback of new order data update.
        """
        pass

    @virtual
    def on_stop_order(self, stop_order: StopOrder) -> None:
        """
        Callback of stop order update.
        """
        pass
…
```

在 vn.py 的设计中，每个策略都有自己的参数（parameter）和变量（variable），读者可以将参数理解为用户从策略外部单独设置的值，与策略本身无关，而通过策略自身的内部逻辑结合用户输入的参数值计算得到的由一系列策略自身维护的值即为变量，变量便于用户对当前策略的运行情况进行监控。例如在代码的__init__方法中使用 self.variables.insert 方法插入 3 个变量：inited、trading 和 pos，其分别表示策略的初始化状态、交易状态和当前持仓值，用户通过这 3 个变量即可知道策略的运行状态和持仓量。

vn.py 文件中的策略主要以回调函数的形式执行各种情形下的逻辑，如上代码所示使用 virtual 注解的方法：on_init、on_start、on_stop、on_tick、on_bar、on_trade、on_order 及 on_stop_order，这些方法表示策略中支持的回调函数，下面将分别介绍这些回调函数的含义。

（1）on_init：策略初始化时的回调函数。

（2）on_start：策略启动时的回调函数。

（3）on_stop：策略停止时的回调函数。

（4）on_tick：收到新的 tick 消息的回调函数。

（5）on_bar：收到新的 bar 消息的回调函数。

（6）on_trade：收到新的成交回报的回调函数。

（7）on_order：收到新的委托请求的回调函数。

（8）on_stop_order：收到新的停止单[①]委托请求的回调函数。

通常而言，交易执行的逻辑在 on_tick 或者 on_bar 中执行，收到新的行情信息后，当策略通过计算得到此时应该进行交易时，执行 buy/sell/short/cover 相应的交易动作。在本章接下来的策略中，读者将看到根据不同的交易信号进行下单的示例。


14min

## 5.2 双均线交易策略

根据双均线进行交易可以说是最简单的一种策略。从名称不难看出，其本质由两条均线构成，对于时间序列而言，均线的计算一般使用移动平均线进行计算，下文使用 MA(*t*)表示计算过去长度为 *t* 的时间序列的平均值，根据定义容易知道 MA(*t*)使用式(5-1)进行计算：

$$\mathrm{MA}(t)=\frac{1}{t}\sum_{i=-t}^{-1}x_i \tag{5-1}$$

式(5-1)中所使用的下标表示形式为 Python 中列表的形式，表示取序列 *x* 中的最后 *t* 个元素的平均值（假定时间序列 *x* 为正序排列）。如下代码使用手动计算了移动平均线的值：

```
//ch5/ma_strategy/demo.py
test_list = [1, 3, 2, 5, 4, 9, 8, 7]
period = 3

ma_list = []
for i in range(len(test_list) + 1):
    if i >= period:
        #计算下标 i 及其前 period 个元素的均值
        ma = sum(test_list[i - period: i]) / period
        ma_list.append(ma)

print(ma_list)
```

代码比较简单，在此不进行说明。如上代码加粗行展示了一种最简单的均值计算方法，读者可以自行尝试其他均值计算方法，例如几何平均、调和平均或指数移动平均等。

幸运的是，移动平均值的计算逻辑不需要自己实现，TA-Lib 已经实现了许多指标的计算，其中就包含各种均值的计算方法。在 TA-Lib 中，使用 SMA（Simple Moving Average）方法即可完成式(5-1)的计算，函数接收一个名为 timeperiod 的参数，表示回溯数据的长度，如果需要使用 TA-Lib 完成同样的计算，则需要如下的代码：

```
//ch5/ma_strategy/demo.py
```

① 停止单是一种特殊的委托方式，其将委托消息缓存在本地，当市场价格符合委托价格时才真正将委托报送到交易柜台，避免过早报单至柜台而暴露交易意图。

```
import talib
import numpy as np

print(
    talib.SMA(
        np.asarray(test_list).astype(np.float64),
        timeperiod=period
    )
)
```

运行以上代码，不难得出与手动计算的相同结果。不同之处在于 TA-Lib 的方法需要接收数据类型为 float 的 NumPy 数组作为参数，并且其返回与输入数组相同长度的结果，在本示例中，由于 period 值为 3，因此结果中的前两个值为 NaN。

那么如何使用均值进行交易呢？首先读者需要了解均值的特性，其相当于对历史值的平滑操作，并且随着均值计算周期越长，平滑程度越高，图 5-1 为螺纹连续日 K 线与 5、8、13、60 和 250 日的均线图，从图中不难发现除了 60 和 250 日均线，其他均线都随着 K 线一同有较大的波动。


图 5-1　螺纹连续日 K 线及均线

在信噪比低的市场中，过滤无用波动而留下主要的趋势走向是至关重要的，而均线恰好就能做到这一点，然而如果使用较长周期的均线，则会导致数据计算的滞后性，反而会导致均线走势跟不上实际市场走势的变化，因此周期的选取是均线策略的关键。

读者不难理解，对于 $t_1 < t_2$ 来讲，MA($t_1$)的变化会比 MA($t_2$)的变化更为剧烈，更接近市场的实际走势，因此，如果在市场下跌或者横盘时发现短期均线突然从下向上穿过长周期均

线，则说明短期向上趋势强烈，可以认为此时后市可能会上涨；反之，如果在市场上涨或横盘了一段时间后，短期的均线突然从上向下穿过了长周期均线，则说明后市有下跌的可能，这两种交叉分别称作“金叉”和“死叉”，图 5-2 展示了螺纹连续日 K 中的一组“金叉”和“死叉”，其中均线是以 5 日和 10 日线进行判定的。


图 5-2　螺纹连续日线及均线交叉

通过策略逻辑判断，在“金叉”时买入并持有到“死叉”卖出，则能赚取这一波上涨行情的利润，同时不至于被中途的回撤“清洗”出局。在 vn.py 文件中实现双均线交易策略十分简单，代码如下：

```
//ch5/ma_strategy/ma_strategy.py
from vnpy_ctastrategy import (
    CtaTemplate,
    StopOrder,
    TickData,
    BarData,
    TradeData,
    OrderData,
    BarGenerator,
    ArrayManager,
)
from vnpy.trader.object import Interval

class MaStrategy(CtaTemplate):
    """ 均线策略 """
```

```
        author = "ouyangpengcheng"

        fast_window = 5
        slow_window = 10
        fixed_size = 1

        fast_ma0 = 0.0
        fast_ma1 = 0.0

        slow_ma0 = 0.0
        slow_ma1 = 0.0

        parameters = ["fast_window", "slow_window", "fixed_size"]
        variables = ["fast_ma0", "fast_ma1", "slow_ma0", "slow_ma1"]

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

            #默认使用分钟线
            self.bar_generator = BarGenerator(self.on_bar)
            self.array_manager = ArrayManager(2 * max(self.fast_window,
self.slow_window))

        def on_init(self):
            """
            Callback when strategy is inited.
            """
            self.write_log("策略初始化")
            self.load_bar(2 * max(self.fast_window, self.slow_window),
use_database=True)

        def on_start(self):
           …

        def on_stop(self):
           …

        def on_tick(self, tick: TickData):
            """
            Callback of new tick data update.
            """
            self.bar_generator.update_tick(tick)
```

```
    def on_bar(self, bar: BarData):
        """
        Callback of new bar data update.
        """
        array_manager = self.array_manager
        array_manager.update_bar(bar)
        if not array_manager.inited:
            return

        #ArrayManager 中的 sma 方法从底层直接调用 talib.SMA
        #计算 5 日均线
        fast_ma = array_manager.sma(self.fast_window, array=True)
        #判断交叉至少需要两个点
        #因此获取最近一天和最近第二天的 5 日均线值
        self.fast_ma0 = fast_ma[-1]
        self.fast_ma1 = fast_ma[-2]

        #计算 10 日均线
        slow_ma = array_manager.sma(self.slow_window, array=True)
        #获取最近一天和最近第二天的 10 日均线值
        self.slow_ma0 = slow_ma[-1]
        self.slow_ma1 = slow_ma[-2]

        #如果最近一天 5 日均线值大于 10 日均线值
        #并且最近第二天的 5 日均线值小于 10 日均线值
        #则说明最近一天 5 日均线完成了对 10 日均线的上穿(金叉)
        cross_over = self.fast_ma0 > self.slow_ma0 and self.fast_ma1 <
self.slow_ma1

        #如果最近一天 5 日均线值小于 10 日均线值
        #并且最近第二天的 5 日均线值大于 10 日均线值
        #则说明最近一天 5 日均线完成了对 10 日均线的下穿(死叉)
        cross_below = self.fast_ma0 < self.slow_ma0 and self.fast_ma1 >
self.slow_ma1

        #如果发生了金叉
        if cross_over:
            if self.pos == 0:
                #如果无持仓，则开多仓
                self.buy(bar.close_price, self.fixed_size)
            elif self.pos < 0:
                #如果持有空仓，则先平仓再开多仓
```

```
                self.cover(bar.close_price, abs(self.pos))
                self.buy(bar.close_price, self.fixed_size)
            #如果发生了死叉
            elif cross_below:
                if self.pos == 0:
                    #如果无持仓，则开空仓
                    self.short(bar.close_price, self.fixed_size)
                elif self.pos > 0:
                    #如果持仓多仓，则先平仓再开空仓
                    self.sell(bar.close_price, abs(self.pos))
                    self.short(bar.close_price, self.fixed_size)

            self.put_event()

        def on_order(self, order: OrderData):
            …

        def on_trade(self, trade: TradeData):
            …

        def on_stop_order(self, stop_order: StopOrder):
            …
```

在 vn.py 的策略中，通常会初始化一个 BarGenerator 和 ArrayManager 的实例，其中 BarGenerator 用于完成 tick 数据到 K 线数据的合成，参数需要指定合成好 K 线时的回调函数（如上代码中为 self.on_bar），在 on_tick 函数（收到 tick 数据的回调）中调用 BarGenerator 的 update_tick 方法将 tick 数据传入，以便进行 K 线数据的更新；ArrayManager 则负责存储最新和历史的 K 线数据（OHLCV、换手和持仓量），ArrayManager 将 TA-Lib 中大量指标计算的方法又封装了一层，因此在以上代码中使用 ArrayManager 的 SMA 方法进行均线的计算。均线“金叉”与“死叉”的逻辑在代码注释中已经详细说明，在此不再赘述。在 on_init 方法中使用了 load_bar 方法，其会预加载部分历史数据，通过回调 on_bar 完成 ArrayManager 的初始化。

在进行回测之前需要确保 vn.py 使用的数据库（默认为 SQLite）中有足够的数据（足够意味着能够完成 load_bar 的数据预加载和至少一次开仓操作）完成回测，如果读者的数据库已经有足够的数据，则可以跳过本节。笔者在本章代码的文件夹内放置了 PTA 连续和螺纹连续的历史日线数据（TA888.csv、rb888.csv），使用 3.7 节中的历史数据管理模块对 CSV 数据进行导入即可。

使用 3.4 节中介绍的 CTA 回测模块对本节的均线进行默认参数（5 日和 10 日均线）回测，可以得到如图 5-3 所示的结果，笔者使用的是 PTA 连续数据完成的回测，在回测之前需

要设置品种相关参数，将合约乘数与价格跳动改为交易品种的特定值（PTA 的合约乘数为 5，价格跳动为 2）。

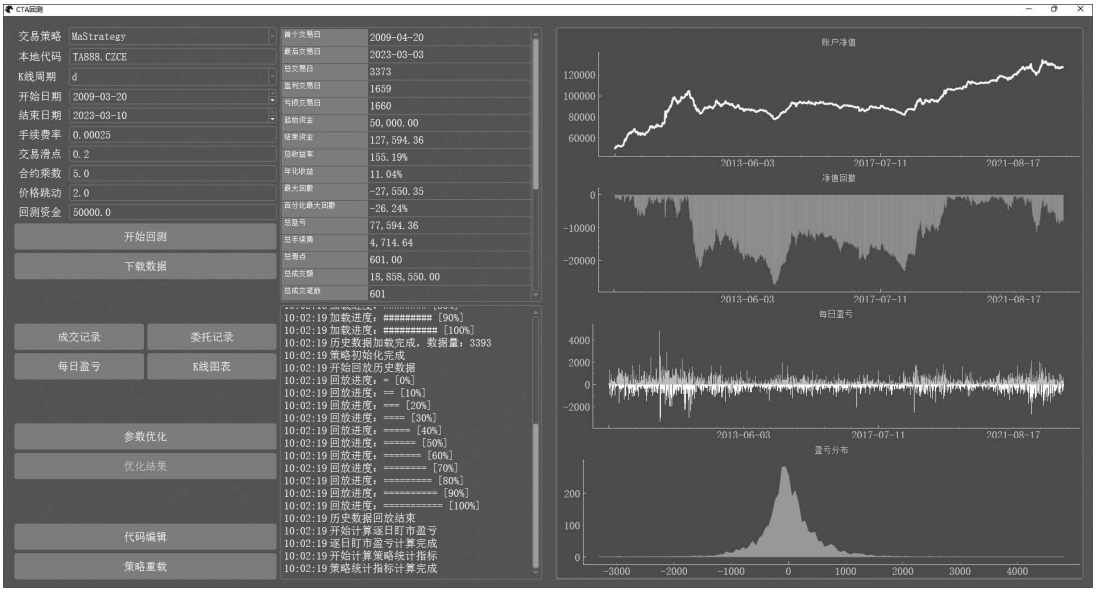


图 5-3　PTA 连续均线策略回测结果

从图 5-3 可以看出，在回测周期内策略取得了年化 11%的收益，总收益为 155%。那么对于 PTA 而言，默认的 5 日与 10 日均线是否是最好的参数？vn.py 提供了一个参数优化的工具，可以对策略参数进行暴力优化或遗传算法的优化，并且可以指定优化目标（默认为总收益率）。上述策略中的参数仅包括两条均线的周期和单次的下单量，在交易信号相同的情况下，更大的下单量势必会带来更高的收益/回撤，因此下单量参数通常不作为优化的对象，在此仅将优化目标设置为“总收益率”，优化对象为长短期均线的周期，其中短期均线的优化范围为[1,20]，步长为 1；长期均线优化范围为[1,60]，步长为 1，使用多进程进行暴力优化，优化参数配置如图 5-4 所示。

需要注意的问题是，在 vn.py 的优化过程中，不会检查“短期均线周期小于长期均线周期”这一限制，通常来讲人工在优化结果中将不符合限制的结果剔除或在策略代码中进行参数的校验即可。单击图 5-4 中的“多进程优化”并等待一段时间后单击“优化结果”可以得到如图 5-5 所示的参数优化结果。

将最优参数填入进行回测可以得到如图 5-6 所示的回测结果。