## **vn.py社区精选19 - 福音收尾，MongoDB分表数据读取！**

原创 用Python的交易员 VNPY  2020-02-25

收录于话题

#vn.py社区精选

25个

**原文作者：庸木**

上一期文章[《vn.py社区精选18 - 老用户福音，MongDB分表重构！》](http://mp.weixin.qq.com/s?__biz=MzI1MTQ2Njc5OQ==&mid=2247484211&idx=1&sn=16c0ccc409d6b04b5df9d69cef79f46a&chksm=e9f3c673de844f65c1eb0ff700d56ec0480126e0c28d0232e0f6ec8c8caff87e42ee82b5c7e4&scene=21" \l "wechat_redirect" \t "/Users/huangning/Documents\\x/_blank)介绍了如何对v2.0版本vn.py中的MongoDB数据库进行分表处理，但在数据读写方面上只讲到了“写”的部分（将历史数据写入重构后的数据库）。

本期文章就来补完之前缺失的“读”的部分，讲解如何从重构后的数据库中读取数据用于策略的回测和优化。整体内容分为两块：

数据库对接层中，对于读取函数实现的修改

CTA回测引擎中，对于读取函数调用的修改

**数据库对接层改造**

**database.py文件**

找到vn.py源代码所在的路径，使用VN Studio的情况下应该位于C:\vnstudio\Lib\site-packages\vnpy，进入到目录vnpy\trader\database下找到database.py 文件。

我们需要对load\_bar\_data和load\_tick\_data这两个函数进行修改，增加了一个可选参数collection\_name用于指明需要读取数据的特定集合名称：

@abstractmethod def load\_bar\_data( self, symbol: str, exchange: "Exchange", interval: "Interval", start: datetime, end: datetime, collection\_name: str = None ) -> Sequence["BarData"]: pass  
 @abstractmethod def load\_tick\_data( self, symbol: str, exchange: "Exchange", start: datetime, end: datetime, collection\_name: str = None ) -> Sequence["TickData"]: pass

**database\_mongo.py文件**

然后打开位于同一目录下的database\_mongo.py文件，同样我们需要对save\_bar\_data和save\_tick\_data进行修改。

改动后的数据读取逻辑如下：

若没有指定collection\_name，则从db\_bar\_data或db\_tick\_data中读取数据；

若指定了collection\_name，则调用switch\_collection函数，从指定的集合中读取数据。

def load\_bar\_data( self, symbol: str, exchange: Exchange, interval: Interval, start: datetime, end: datetime, collection\_name: str = None, ) -> Sequence[BarData]: if collection\_name is None: s = DbBarData.objects( symbol=symbol, exchange=exchange.value, interval=interval.value, datetime\_\_gte=start, datetime\_\_lte=end, ) else: with switch\_collection(DbBarData, collection\_name): s = DbBarData.objects( symbol=symbol, exchange=exchange.value, interval=interval.value, datetime\_\_gte=start, datetime\_\_lte=end, ) data = [db\_bar.to\_bar() for db\_bar in s] return data  
  
 def load\_tick\_data( self, symbol: str, exchange: Exchange, start: datetime, end: datetime, collection\_name: str = None, ) -> Sequence[TickData]: if collection\_name is None: s = DbTickData.objects( symbol=symbol, exchange=exchange.value, datetime\_\_gte=start, datetime\_\_lte=end, ) else: with switch\_collection(DbTickData, collection\_name): s = DbTickData.objects( symbol=symbol, exchange=exchange.value, datetime\_\_gte=start, datetime\_\_lte=end, ) data = [db\_tick.to\_tick() for db\_tick in s] return data

**CTA回测引擎改造**

**backtesting.py文件**

进入目录vnpy\app\cta\_strategy打开backtesting.py文件，这次我们要修改的函数比较多，为了方便整理，基于函数功能大致把它们分成两块：

策略回测部分

参数优化部分

**策略回测部分**

1. 在load\_bar\_data 和 load\_tick\_data增加一个入参collection\_name：

@lru\_cache(maxsize=999)def load\_bar\_data( symbol: str, exchange: Exchange, interval: Interval, start: datetime, end: datetime, collection\_name: str = None): """""" return database\_manager.load\_bar\_data( symbol, exchange, interval, start, end, collection\_name )  
  
@lru\_cache(maxsize=999)def load\_tick\_data( symbol: str, exchange: Exchange, start: datetime, end: datetime, collection\_name: str = None): """""" return database\_manager.load\_tick\_data( symbol, exchange, start, end, collection\_name )

2. 在BacktestingEngine类的\_\_init\_\_函数中增加新的类属性collection\_name用于自定义要使用的MongoDB集合名称：

def \_\_init\_\_(self): """""" self.vt\_symbol = "" self.symbol = "" self.exchange = None self.start = None self.end = None self.rate = 0 self.slippage = 0 self.size = 1 self.pricetick = 0 self.capital = 1\_000\_000 self.mode = BacktestingMode.BAR self.inverse = False self.collection\_name = None

3）对set\_parameters进行修改，增加一个入参collection\_name，并把该参数的值绑定到类属性collection\_name：

def set\_parameters( self, vt\_symbol: str, interval: Interval, start: datetime, rate: float, slippage: float, size: float, pricetick: float, capital: int = 0, end: datetime = None, mode: BacktestingMode = BacktestingMode.BAR, inverse: bool = False, collection\_name: str = None ): """""" self.mode = mode self.vt\_symbol = vt\_symbol self.interval = Interval(interval) self.rate = rate self.slippage = slippage self.size = size self.pricetick = pricetick self.start = start  
 self.symbol, exchange\_str = self.vt\_symbol.split(".") self.exchange = Exchange(exchange\_str)  
 self.capital = capital self.end = end self.mode = mode self.inverse = inverse self.collection\_name = collection\_name

4. 对load\_data函数进行修改，增加一个类属性collection\_name。由于load\_data函数涉及逻辑比较多，下面只显示修改代码部分，其他用省略号(...)来代替：

... if self.mode == BacktestingMode.BAR: data = load\_bar\_data( self.symbol, self.exchange, self.interval, start, end, self.collection\_name, ) else: data = load\_tick\_data( self.symbol, self.exchange, start, end, self.collection\_name,                )···

**参数优化部分**

在参数优化的代码中，也需要对数据的读取位置进行修改，因为默认优化函数读取的数据表/集合仍然是默认的da\_bar\_data或db\_tick\_data。

1. 首先增加全局变量collection\_name, 并对它进行赋值：

global ga\_collection\_name  
ga\_collection\_name = self.collection\_name

2. 修改\_ga\_optimize和optimize函数，增加入参collection\_name：

def optimize( target\_name: str, strategy\_class: CtaTemplate, setting: dict, vt\_symbol: str, interval: Interval, start: datetime, rate: float, slippage: float, size: float, pricetick: float, capital: int, end: datetime, mode: BacktestingMode, inverse: bool, collection\_name: str = None): """ Function for running in multiprocessing.pool """ engine = BacktestingEngine()  
 engine.set\_parameters( vt\_symbol=vt\_symbol, interval=interval, start=start, rate=rate, slippage=slippage, size=size, pricetick=pricetick, capital=capital, end=end, mode=mode, inverse=inverse, collection\_name=collection\_name )  
 engine.add\_strategy(strategy\_class, setting) engine.load\_data() engine.run\_backtesting() engine.calculate\_result() statistics = engine.calculate\_statistics(output=False)  
 target\_value = statistics[target\_name] return (str(setting), target\_value, statistics)  
  
@lru\_cache(maxsize=1000000)def \_ga\_optimize(parameter\_values: tuple): """""" setting = dict(parameter\_values)  
 result = optimize( ga\_target\_name, ga\_strategy\_class, setting, ga\_vt\_symbol, ga\_interval, ga\_start, ga\_rate, ga\_slippage, ga\_size, ga\_pricetick, ga\_capital, ga\_end, ga\_mode, ga\_inverse, ga\_collection\_name, ) return (result[1],)

**Jupyter Notebook使用示例**

最后我们同样可以通过Jupyter Notebook来测试下使用效果，在上一期文章的结尾，我们已经把XBTUSD数据导入到新的集合【XBTUSD】中。

现在我们要从这个新的集合读取数据进行回测，需要修改的部分同样很简单，只要调用set\_parameters函数时，增加参数collection\_name="XBTUSD"即可：

engine = BacktestingEngine()engine.set\_parameters( vt\_symbol="XBTUSD.BITMEX", interval="1h", start=datetime(2018, 1, 1), end=datetime(2019, 1, 1), rate=1/10000, slippage=0.5, size=10, pricetick=0.5, capital=1\_000\_000, collection\_name = "XBTUSD")engine.add\_strategy(AtrRsiStrategy, {})