QuantLAB

使用指南

盛冠达资产

# 前言

QuantLAB是基于.NET 5的量化回测平台。支持日线、分钟线、Tick级回测，支持的市场有国内期货和股票。

建议使用WINDOWS来开发和运行QuantLab策略，因为Microsoft官方有完善的IDE。编写好的QuantLab策略也可以在纯Linux环境下运行。

如果您需要在Linux下开发或编译.NET项目，可考虑使用Visual Studio Code或JetBrains Rider。请查阅相关网站。

# 系统安装(Windows)

1. 策略开发，请首先安装Visual Studio 2022，并勾选安装**.NET桌面开发**，以及**.NET 5.0**
2. 解压QuantLAB的压缩包到任何目录，这里建议D:\QuantLab3，以便示例代码可以不加修改运行。
3. 配置数据路径  
   确认Config\data\_sources.json中的数据目录是正确的

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

如果配置正确，您可以通过文件管理器来打开“Path”中对应的目录来查看数据。

1. 系统运行有两种方式，图形界面和命令行。
   1. 运行SunnyTrader.QuantLab.Client\_x64.exe，启动图形界面，可看到回测系统主界面

图形用户界面, 应用程序, Word

描述已自动生成

* 1. 运行SunnyTrader.QuantLab.Console.exe，将启动命令行执行。系统运行并不是交互的，您需要在参数中指定要运行的策略。详情请参照“命令行参数”

# 系统安装(Linux)

1. 安装 **.NET 5.0**

https://docs.microsoft.com/zh-cn/dotnet/core/install/linux-ubuntu

1. 解压QuantLAB的压缩包到任何目录
2. 配置数据路径  
   确认Config/data\_sources.json中的数据目录是正确的。
   1. 运行dotnet SunnyTrader.QuantLab.Console.dll可启动回测。系统运行并不是交互的，您需要在参数中指定要运行的策略。详情请参照“命令行参数”

# 示例项目(Windows)

附件中的MyStrategy.rar有一段示例代码。这是一个布林带突破策略，每次在沪深300指数突破布林带的时候，对主力合约开仓。

为省去配置项目的麻烦，您的新策略可以在此策略的基础上做修改。

下面介绍策略如何运行

1. 解压到任意目录可以用Visual Studio直接打开。

电脑萤幕的截图

描述已自动生成

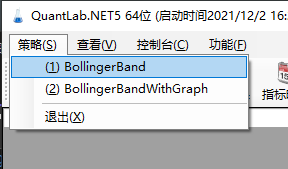
2、为确保项目可以直接运行，请将QuantLab解压到了D:\QuantLab3，

3、项目打开后，按F5键可以启动调试，按Ctrl+F5可以不带调试器运行。运行后的界面如下：

图形用户界面, 应用程序, Word

描述已自动生成

1. 现在点击左上方的策略菜单，并选择BollingerBand。



1. 程序会列出所有参数组。默认所有参数组均为选中。请将时间调整为2012.1.1-2012.3.1。并点击确定

表格

描述已自动生成

1. 单击“全部开始”按钮开始运行策略

图形用户界面, 表格

描述已自动生成

1. 运行完成后，将下方tab切换到收益曲线，可以看各个参数组的收益情况。

图表, 折线图

描述已自动生成

1. 在详情tab中，可以查看每天的成交记录

图形用户界面, 应用程序, 表格

描述已自动生成

# 手动建立项目(Windows)

如果不想基于示例代码修改，您也可以考虑手动建立项目，过程如下

1. 建立项目

截图里有图片

描述已自动生成

电脑屏幕的截图

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

电脑萤幕的截图

描述已自动生成

1. 修改调试配置

右键单击项目，选择“属性”，并打开调试选项卡，将可执行文件设置为您的QuantLab主程序，并将工作目录设置为您的QuantLab安装目录

电脑萤幕的截图

描述已自动生成

1. 修改项目属性

左键单击项目，将在代码区打开策略的项目文件。请做下列编辑：

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFramework>net5.0</TargetFramework>

<AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOutputPath>

</PropertyGroup>

<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">

<OutputPath>D:\QuantLab3\Strategies</OutputPath>

</PropertyGroup>

<ItemGroup>

<Reference Include="Google.Protobuf">

<HintPath>D:\QuantLab3\Google.Protobuf.dll</HintPath>

<Private>false</Private>

</Reference>

<Reference Include="SunnyTrader.Common">

<HintPath>D:\QuantLab3\SunnyTrader.Common.dll</HintPath>

<Private>false</Private>

</Reference>

<Reference Include="SunnyTrader.QuantLab.Common">

<HintPath>D:\QuantLab3\SunnyTrader.QuantLab.Common.dll</HintPath>

<Private>false</Private>

</Reference>

<Reference Include="SunnyTrader.Repository">

<HintPath>D:\QuantLab3\SunnyTrader.Repository.dll</HintPath>

<Private>false</Private>

</Reference>

<Reference Include="SunnyTrader.Repository.Utils">

<HintPath>D:\QuantLab3\SunnyTrader.Repository.Utils.dll</HintPath>

<Private>false</Private>

</Reference>

</ItemGroup>

</Project>

上述代码作了如下修改：

1. 修改输出目录为QuantLab下的Strategies目录。
2. 增加必要的dll引用

其中“D:\QuantLab3”为QuantLab解压的目录，应修改为您自己的安装目录

电脑萤幕的截图

描述已自动生成

1. 编写策略代码

电脑萤幕画面

描述已自动生成

# 编写策略

示例代码中的策略，是一个基于TestStrategy的抽象类。所有的策略都是基于这个类。一个策略可以改写下列函数：

Init()

在策略开始运行时会被调用，您可以在这个函数里做一些初始化工作。例如，如果您的策略需要写文件，可以在这里初始化文件夹。当然，很多情况下，您并不需要实现这个函数。

BeforeTrading();

在每个交易日开始时调用。您可以在这个函数里订阅当天要推送的合约，配置每个合约的手续费。

如果您通过SubMarketData订阅了合约，系统将逐条推送tick数据到OnDataArrive。

如果您通过SubCandleData订阅了合约，系统将逐条推送K线数据到OnCandleArrive。

注意：Tick和K线策略是互斥的 ，您不可以同时通过SubMarketData/SubCandleData订阅两种数据。但是不同时间尺度的K线数据是可以同时订阅的。比如5分钟和1分钟的K线是可以同时推送的

OnDataArrive(FastQuotaData data);

在每个Tick数据到达时调用。您可以在这个函数里下单交易，查询持仓，等等。

OnCandleArrive(Instrument instrument, CandleType type, CandleData candle);

在K线数据到达时调用。

AfterTrading();

在每个交易日结束时调用。

AfterSettlement(DailyInfo todayInfo, DailyDetail todayDetail);

在每个交易日结算完成后调用。

Finish();

在策略执行完毕后调用。

# 策略参数枚举

对于同一个策略，您可能想测试不同的参数。例如，您可能想测试不同的股票标的的表现，可能想试一下不同的布林带宽度给策略收益带来的影响。

## 值枚举

QuantLAB支持通过“StrategyParams”修饰策略的字段来进行参数枚举。例如：

public class BollingerBand : TestStrategy

{

[StrategyParams(ParamName = "均线周期", ValueList = new object[] { 60, 90, 120, 180, 240, 300 })]

public int Interval;

….

}

其中看到ValueList里有7个值，那么系统会自动生成7个策略对象，每个对象的Width被初始化成对应的值。

目前char, int, long, float, double, string类型的字段都支持枚举，Enum也支持。

## 循环枚举

您还可以通过循环的方式来枚举，例如：

[StrategyParams(ParamName = "均线周期", StartPoint = 60, EndPoint = 180, Step = 30)]

public int Interval;

## 文件枚举

如果您的参数放在文件里，可以通过**File**StrategyParams指定文件名来实现枚举，例如：

[**File**StrategyParams(ParamName = "股票", FileName = @"d:\data\stock\_list.csv", FirstLine = 1)]

public string Stock;

代码将读取一个csv文件，将文件的每一行作为一个参数值。其中FirstLine=1代表略过第一行。此外，您还可以通过Separator指定文件中是用什么来分割不同值的。默认是回车符，即每行一个值。

## 自定义枚举

如果已有的枚举方式依然无法满足需要 ，可以通过继承BaseStrategyParamsAttribute来实现自定义枚举器，然后改写GenerateValues()方法，例如，下面的例子通过读取文件中的每一行来生成枚举值：

class **StockParamsAttribute** : BaseStrategyParamsAttribute

{

protected override IEnumerable GenerateValues()

{

return File.ReadAllLines("corrs\\codes.txt");

}

}

然后，在相应的字段用上述自定义的枚举器进行修饰：

[**StockParams**(ParamName = "StockCode")]

public string Code;

# 撮合系统

默认情况下，通过MarketOrder下单时，系统会根据参数中的量价立即成交（无论指定的价格和量是否合理）。对于一些日线级模型，配合滑点，这种方式是足够的。但是，对于高频模型，可能需要判断盘口目前的状态甚至结合逐笔数据。因此，系统开放了撮合接口，可以由用户自定义撮合规则。

## 实现撮合器

您可以通过实现IOrderMatcher接口实现一个撮合器

public interface IOrderMatcher

{

void PreInit()

void PostInit()

void PreBeforeTrading()

void PostBeforeTrading()

void PreDataArrive(FastQuotaData q)

void PostDataArrive(FastQuotaData q)

void PreAfterTrading()

void PostAfterTrading()

void PreFinish()

void PostFinish()

void AddOrder(OrderInfo orderInfo);

void CancelOrder(OrderInfo orderInfo);

}

其中PreInit()是在策略的Init()函数前调用，PostInit()是在策略的Init()函数后调用。后面的函数类似。

AddOrder()函数用来接收策略的MarketOrder报单，

CancelOrder()用来接收策略的CancelOrder撤单。

系统的默认撮合器是这样实现的：

public class DefaultOrderMatcher : IOrderMatcher

{

private IAccount \_account;

public DefaultOrderMatcher(IAccount account)

{

\_account = account;

}

public void AddOrder(OrderInfo orderInfo)

{

\_account.AddTradeInfo(

orderInfo, null, orderInfo.InsertTime,

orderInfo.VolumeRequested, orderInfo.LimitPrice);

\_account.NotifyOrderUpdated(orderInfo);

}

public void CancelOrder(OrderInfo orderInfo)

{

}

}

可以看到，对于任何的报单，都是立即成交

## 切换撮合器

实现撮合器后，您可以在策略中建立一个撮合器对象，并分配给当前的TestCase

TestCase.OrderMatcher = new MyStockOrderMatcher(TestCase, \_stock);

之后，撮合器就由默认撮合器切换为您的自定义撮合器。

## 使用撮合器

撮合系统是集成到回测系统中的，MarketOrder将自动将委托派给当前撮合器。因此，请不要在策略中调用撮合器的任何函数，这将导致不可预测的行为。

您应当依旧使用MarketOrder函数来下单。MarketOrder会返回一个委托对象：

var orderInfo = MarketOrder(\_stock, EnumDirectionType.Buy, EnumOffsetFlagType.Close, RemainingBuyCloseVol, buyLimitPrice);

OrderInfo中有如下字段：

public class OrderInfo

{

public string OrderId;

public Instrument Instr;

public DateTime InsertTime;

public EnumDirectionType Direction;

public EnumOffsetFlagType Offset;

public double LimitPrice;

public int VolumeTraded;

public double AmountTraded;

public int VolumeRequested;

public bool Cancelled;

public double Fee;

public string Payload;

}

}

撮合器会随时更新orderInfo对象中的值。因此，请不要在策略中修改orderInfo对象中的字段，否则会干扰撮合器的运行，导致不可预测的行为。

## 委托与成交回报

撮合器还负责发送委托和成交回报。在策略中实现下列两个函数，可以接收委托与成交回报。

public virtual void OnTradeAdded(IAccount account, OrderInfo orderInfo, TradeInfo tradeInfo)

public virtual void OnOrderUpdated(IAccount account, OrderInfo orderInfo)

# 策略API

1. 生成股票/期货对象。

您可以通过Instrument类来生成股票、期货对象。例如：

var stock = Instrument.stock(“600036”)

var future = Instrument.future(“IF2103”)

1. 订阅行情

在BeforeTrading()中使用SubMarketData订阅行情，系统将通过OnDataArrive推送当日的Tick：

SubMarketData(Instrument.stock(“600036”), 0.0008)

您可以多次调用SubMarketData订阅多个合约，系统将按照时间顺序推送数据。

1. 获取策略初始参数
   1. 通过TestCase.InitialValue获取到初始资金。这个初始资金是在策略加载框中指定的。同时，您也可以随时更改这个值。

InitivalValue仅用来计算最终收益率。因为每个市场保证金计算不同，回测系统无法告知您当前“可用资金”

* 1. TestCase.StartDate和TestCase.EndDate用来获取回测的日期区间。同样也是在策略加载框中选择的。这个值您也可以在Init()函数中修改。

1. 下单

MarketOrder函数用来下单。默认撮合系统下，您的任何委托都会直接按照您指定的价格成交。

1. 查询持仓

GetPosition函数用来获取您的合约持仓情况。包括昨仓，今仓，多头，空头。

# 数据API

回测系统在运行的时候会根据您的订阅自动推送Tick和Candle数据，但是，您也可以自己通过DataUtils类来获取数据。例如，通过DataUtils.GetTicks可以获取Tick级数据。DataUtils.GetDaily获取日线级数据。这些函数已经经过LRU缓存优化，最近访问过的数据会被放在内存中。重复调用不会反复读取硬盘。

PbLoader中有更加底层的数据读取实现。是直接解析磁盘中的PB文件，如果需要用其他语言（如java，c++）读取PB文件，可以参考PBLoader.cs的源代码。

数据读取的相关源代码是公开的，请参考QuantLab下的Src子目录。

# 数据文件结构

所有的行情数据均为文件存储，分为三级目录。第一级为市场大类。

图形用户界面, 应用程序, 表格

描述已自动生成

第二级为日期

图形用户界面, 应用程序, 表格

描述已自动生成

第三级为数据格式类别。

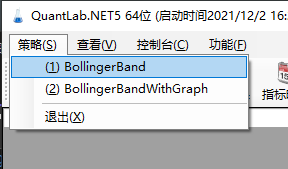
图形用户界面, 应用程序, 表格

描述已自动生成

每一类数据均有两个存储方式：csv和protocol buffer。前者给python的pandas读取，后者给java/c#/c++之类的语言读取。

# 系统运行流程

回测系统在启动后，会在您当前工作目录的Strategies目录下动态加载所有的dll，并搜索基于TestStrategy的类，并显示在策略菜单里。如图所示：

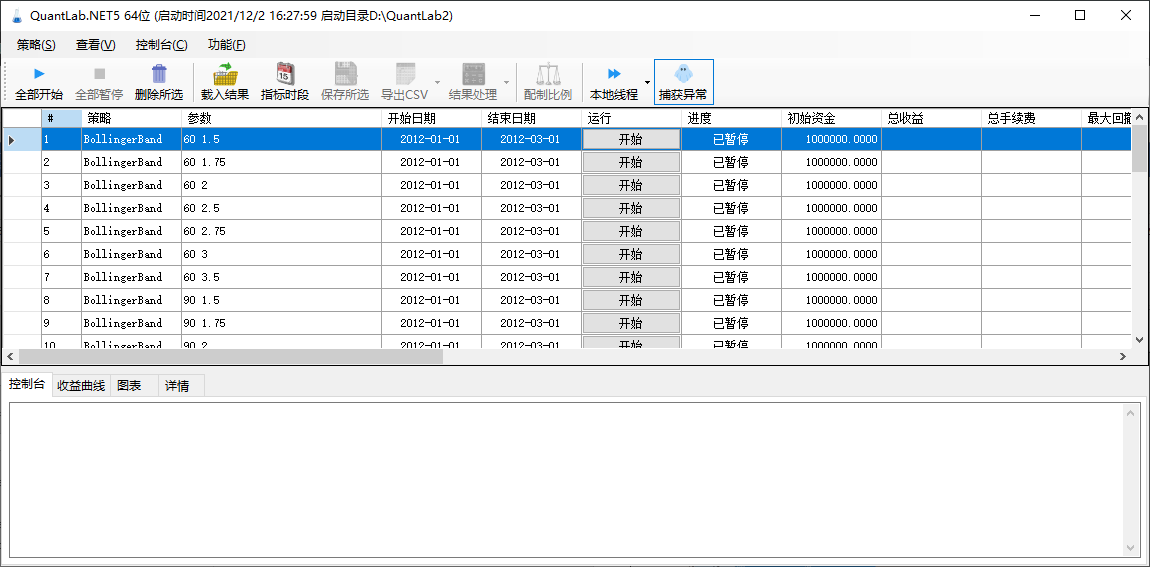


点击策略后，系统会枚举该策略所有的参数组合，并弹框让您选择需要回测哪些组合，同时您还可以选择初始资金。以及回测的时间。

表格

描述已自动生成

点击确定之后，这些策略对象将被生成，并显示在主窗口中，所有的参数也会显示出来，方便区分



当点击策略对象的“开始”后，策略对象s会被这样调用：

s.Init();

For each day from startDate to endDate:

s.BeforeTrading();

Loads all subscribed ticks/candles and sort by time

For each tick or candle:

s.OnDataArrive(tick)/s.OnCandleArrive(candle)

AfterTrading();

Caculates profit and other information of the day.

AfterSettlement()

Finish()

系统支持多线程回测，您可以同时启动多个策略对象，它们将被分配的不同的CPU核心上运行。您可以通过“本地线程”按钮来选择并发数量。

# 编写回测结果处理器

## 概要

很多情况下，我们需要在回测结束后对回测结果做进一步处理，比如，找到收益率最优的个体，把成交回报存档，等等。 因此，回测系统提供了一个IResultProcessor接口，您需要实现如下函数：

void void ProcessAll(List<TaskResult> results, CancellationToken ct, Action<int, int> onProgress, string option)

参数说明：

1. results是所有已完成任务的结果，里面将包含任务的概略信息，每日盈亏，每日的委托和成交记录。
2. ct是终止信标，当用户点击取消按钮时，该信标将被设置。如果您的程序执行时间很长，可考虑响应该信标。
3. onProgress可用来向系统报告执行进度，系统将以进度条的形式显示您的进度。
4. option是从命令行传入的选项，具体可参考“命令行参数”一章

## 数据说明

### TaskResult结构定义

public abstract class TaskResult

{

/// <summary>

/// 回测任务编号，从1开始

/// </summary>

public int TaskId;

/// <summary>

/// 策略名称

/// </summary>

public string StrategyName;

/// <summary>

/// 参数名列表

/// </summary>

public string[] ParamNames;

/// <summary>

/// 参数值列表

/// </summary>

public string[] ParamValues;

/// <summary>

/// 所有指标

/// </summary>

public Indicators Inds;

/// <summary>

/// 每日回测概况，包括盈亏，手续费等信息

/// </summary>

public abstract List<DailyInfo> DailyInfoList { get; }

/// <summary>

/// 每日回测详情，包括交易记录，仓位记录

/// </summary>

public abstract List<DailyDetail> DailyDetailList { get; }

/// <summary>

/// 日期过滤器

/// </summary>

public Func<DateTime, bool> DateFilter;

/// <summary>

/// 说明

/// </summary>

public string Description;

/// <summary>

/// 提示

/// </summary>

public string Tooltip;

/// <summary>

/// 选项，参见命令行参数

/// </summary>

public string Option;

/// <summary>

/// 删除标志，如果您的处理器将其设为true，则运行完毕后对应的策略实例将从主列表中消失

/// </summary>

public bool Delete = false;

/// <summary>

/// 选择标志，如果您的处理器将其设为true，则运行完毕后对应的策略实例将从主列表中被选中

/// </summary>

public bool Select = true;

}

### DailyInfo对象结构定义

/// <summary>

/// 每日交易概况

/// </summary>

public class DailyInfo

{

/// <summary>

/// 交易日

/// </summary>

public DateTime TradingDay;

/// <summary>

/// 持仓总量

/// </summary>

public Position TotalPosition;

/// <summary>

/// 盈亏（未扣除手续费）

/// </summary>

public double Profit;

/// <summary>

/// 手续费

/// </summary>

public double Fee;

/// <summary>

/// 成交量

/// </summary>

public long Volume;

/// <summary>

/// 成交额

/// </summary>

public double TurnAmount;

/// <summary>

/// 持有额

/// </summary>

public double HoldingAmount;

/// <summary>

/// 备注信息

/// </summary>

public string Note;

/// <summary>

/// 委托数

/// </summary>

public int OrderCount;

/// <summary>

/// 撤单数

/// </summary>

public int CancelCount;

}

### DailyDetail对象结构定义

/// <summary>

/// 每日交易详情

/// </summary>

public class DailyDetail

{

/// <summary>

/// 持仓表

/// </summary>

public Dictionary<Instrument, Position> PositionMap;

/// <summary>

/// 委托表

/// </summary>

public List<OrderInfo> OrderList;

/// <summary>

/// 成交表

/// </summary>

public List<TradeInfo> TradeList;

/// <summary>

/// 收盘价

/// </summary>

public Dictionary<Instrument, double> EndPrices;

/// <summary>

/// 图表对象

/// </summary>

public GraphDataType GraphData;

/// <summary>

/// 额外的信息

/// </summary>

public string Payload;

}

### Position对象定义

public class Position

{

/// <summary>

/// 多头持仓

/// </summary>

public int LongVolume;

/// <summary>

/// 空头持仓

/// </summary>

public int ShortVolume;

}

### OrderInfo对象定义

/// <summary>

/// 委托对象

/// </summary>

public class OrderInfo

{

/// <summary>

/// 委托号

/// </summary>

public string OrderId;

/// <summary>

/// 股票或期货对象

/// </summary>

public Instrument Instr;

/// <summary>

/// 下单时间

/// </summary>

public DateTime InsertTime;

/// <summary>

/// 买卖

/// </summary>

public EnumDirectionType Direction;

/// <summary>

/// 开平

/// </summary>

public EnumOffsetFlagType Offset;

/// <summary>

/// 限价

/// </summary>

public double LimitPrice;

/// <summary>

/// 成交量

/// </summary>

public int VolumeTraded;

/// <summary>

/// 成交额

/// </summary>

public double AmountTraded;

/// <summary>

/// 挂单手数

/// </summary>

public int VolumeRequested;

/// <summary>

/// 取消标志

/// </summary>

public bool Cancelled;

/// <summary>

/// 手续费

/// </summary>

public double Fee;

/// <summary>

/// 用户自定义信息

/// </summary>

public string Payload;

}

### TradeInfo对象定义

/// <summary>

/// 成交记录

/// </summary>

public class TradeInfo

{

/// <summary>

/// 委托号

/// </summary>

public string TradeId;

/// <summary>

/// 委托号

/// </summary>

public string OrderId;

/// <summary>

/// 股票或期货对象

/// </summary>

public Instrument Instr;

/// <summary>

/// 成交时间

/// </summary>

public DateTime Time;

/// <summary>

/// 买卖

/// </summary>

public EnumDirectionType Direction;

/// <summary>

/// 开平

/// </summary>

public EnumOffsetFlagType Offset;

/// <summary>

/// 成交价格

/// </summary>

public double Price;

/// <summary>

/// 成交量

/// </summary>

public int Volume;

/// <summary>

/// 用户自定义信息

/// </summary>

public string Payload;

}

## 使用方法

系统在启动时，会在Strategies目录下的dll中搜索基于IResultProcessor的类，显示在“处理结果”菜单中：



选中一个或多个已经完成的策略实例，同时点击一个处理模块，系统将把这些实例的回测结果发送给您的处理模块中。您可以访问到每日的盈亏、手续费、持仓和成交记录。

范例中有一个GenTradeRecords的处理模块，会将成交记录写到文件里。可以参考其实现。

# 命令行参数

为实现回测的自动化运行，回测系统支持通过命令行参数来自动运行策略和结果处理器。

|  |  |  |
| --- | --- | --- |
| 参数 | 简写 | 说明 |
| --strategy <策略类名> | -s | 要运行的策略类名，该策略所有的参数将被枚举运行。  可用多个—strategy指定多个策略依次运行 |
| --start\_date <日期> |  | 回测开始日期，格式YYYYMMDD。  如果该参数出现在—strategy之后，则为指定该策略的开始日期。否则指定所有策略的开始日期。  为方便一些日结策略的运行，日期可以是负数，例如-1代表昨天，-2代表前天 |
| --end\_date <日期> |  | 回测结束日期，格式YYYYMMDD |
| --initial\_value <资金> |  | 回测起始资金，如果该参数出现在—strategy之后，则为指定该策略的起始资金。否则指定所有策略的起始资金 |
| --processor <处理器类名> | -p | 在回测结束后，要运行的处理器类。该处理器类将处理所有的运行结果。可用多个—processor指定多个处理器类。 |
| --option <选项字符串> | -o | 选项。  如果此参数出现在所有—strategy之前，则会填入TestStrategy.GlobalOption静态字段中，所有策略均可访问。  如果此参数出现在某个—strategy之后，则填入对应策略实例的Option字段中。  如果此参数出现在某个—processor之后，则会在调用该Processor时通过option参数传入。 |
| --auto\_exit |  | 运行完毕自动退出 |
| --load <存盘文件名> |  | 启动时先载入tls存盘文件 |
| --save <存盘文件名> |  | 回测完成后存盘tls文件 |
| --error\_tolerance |  | 错误容忍率，超过容忍率的，不会自动退出，需要等待用户回车键。 |
| --strategy\_dir <路径名> | -d | 指定策略dll搜索目录，默认为./Strategies |
| --log\_file |  | 指定每个策略示例完成情况的记录文件，默认输出到屏幕。 |
| --jobs <线程数> | -j | 并发线程数，默认为您当前cpu的内核数 |