EPI量化操作手册

EPI策略，通过以下流程进行策略的开发、测试、仿真和实盘。

1.新建策略：编辑策略代码，调试代码。

2.策略回测：代码加密后提交至服务端，由服务端进行测试，并将策略结果反馈给用户（支持全品种测试）。

3.策略分析：所有回测过的策略可进行组合分析，并生成组合策略报告，从而找到一个稳定性最好的策略组合。

4.仿真运行：挑选优异的策略，申请资源在服务端自动运行。

5.实盘跟单：择时择策略进行跟踪信号，并将信号发送至场内。

开始量化

在本文档中，我们详细介绍了平台的各项功能和使用方法。由于内容较多，在浏览本文档的时候，我们建议您多使用 "ctrl + f" 的快捷键组合，快速定位到感兴趣的内容上。

千里之行，始于足下。在学习编程的时候，我们都是从打印一句 "Hello World" 开始，踏入到奇妙的程序世界；同样，在本文档的第一部分，我们准备了一个类似 "Hello World" 的简单策略实例，帮助新用户了解量化策略，以及如何使用我们的平台。

1. "数据"部分，介绍了平台上可供使用的、丰富多样的数据
2. "回测设置"部分，介绍了回测系统的各项默认设置（例如撮合方式、滑点和期货交易费用等）
3. "进行回测"部分，介绍用户如何使用我们的在线IDE进行策略开发和回测，以及回测过程中可供使用的各个函数和对象；
4. "回测结果分析"部分，我们介绍了策略评估的各类核心量化指标（收益、风险和风险调整后收益），以及进阶分析功能（月度收益、持仓情况等）
5. "策略实例"部分，介绍了更多、更具体的策略实例，帮助你了解常见的量化策略开发思路，和如何灵活使用我们提供的数据和功能
6. "实时模拟交易"部分，介绍如何设定实时模拟交易，以更好地评估策略的实盘表现

希望通过我们的文档和平台，您能够逐步成长为一个优秀的量化策略开发者。如果您对我们文档或平台有任何建议，欢迎随时联系我们；如果希望分享编写策略过程中的疑问或心得体会，欢迎来我们的用户社区一起交流讨论。实践，思考，交流，成长，翌派科技与你一路同行。

开发环境

操作系统：Win7及以上操作系统

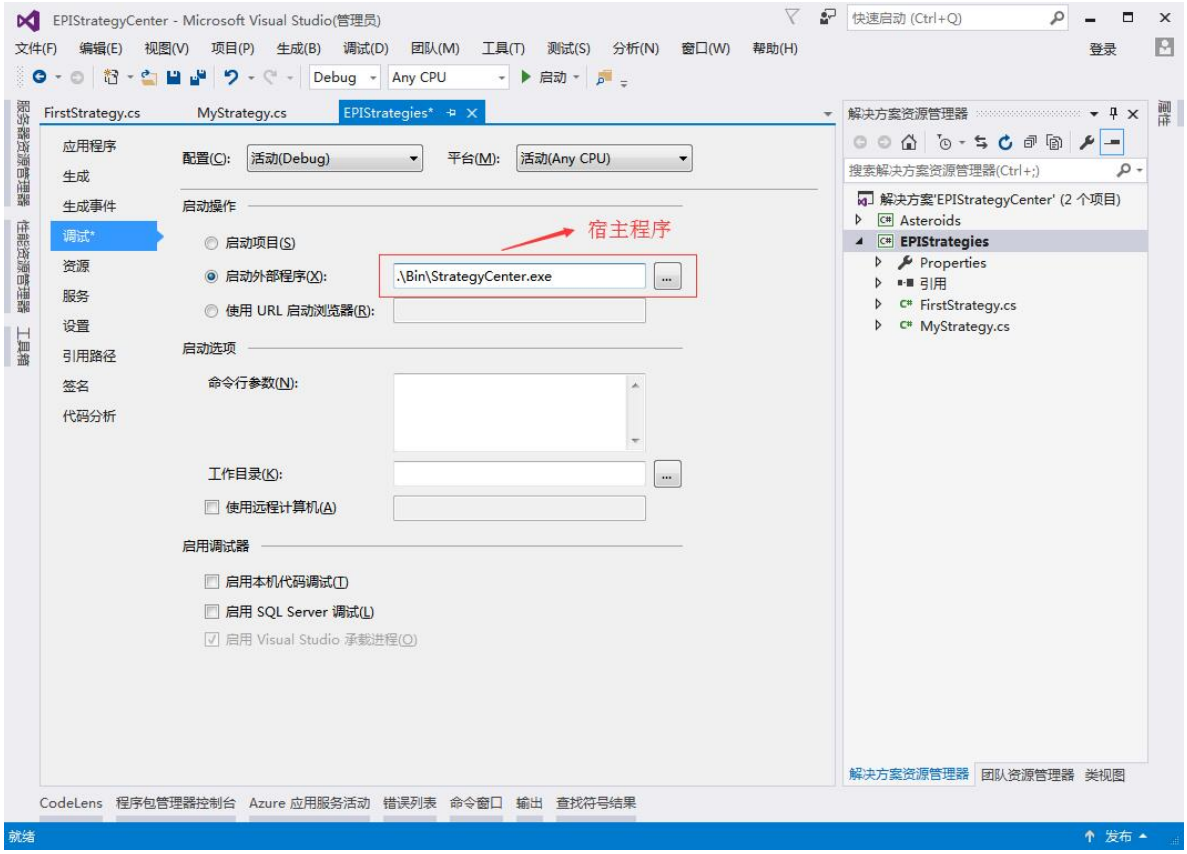
开发工具：VS2015及以上IDE，或者SharpDevelop5.1

回测工具：EPIStrategyCenter（用于本地调试、回测和查看报表），云端回测请登录u.epai.tech

本地策略编写

开始工作

解压EPIStrategyCenter.zip到任意磁盘路径，使用IDE打开EPIStrategyCenter.sln打开VS项目，项目结构如下图：



项目结构说明

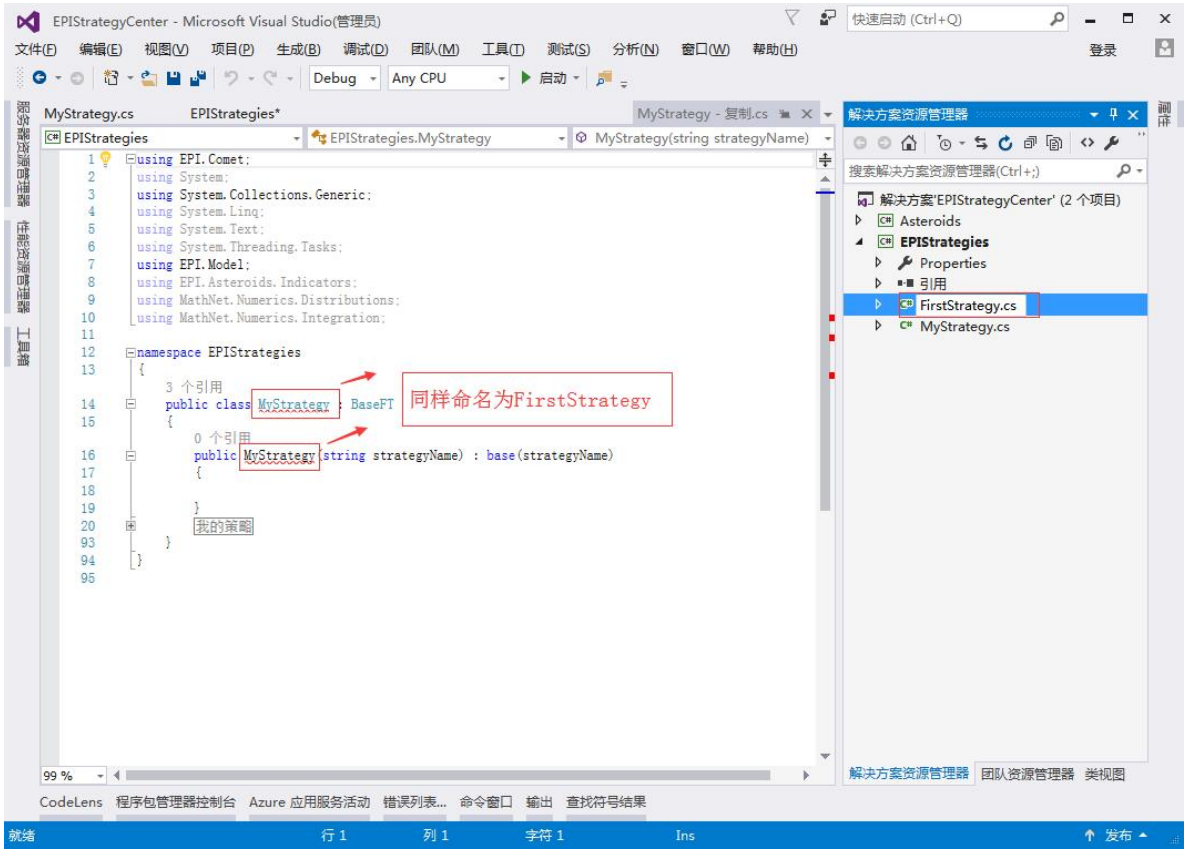
Asteroids 指标项目：用户可以使用EPI开源指标类，也可以写自己的指标。

EPIStrategies策略项目：用户所有的策略都可以新建类的形式放置于该项目中；为了方便调试策略，EPI提供了EPIStrategyCenter宿主程序（默认已设置好启动外部程序）。

MyStrategy类：策略的模版类，用户可以复制该类来新建策略。

编写策略

复制并粘贴MyStrategy策略，并命名为FirstStrategy，如下图：



策略代码如下：

`

```
using EPI.Comet;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using EPI.Model;
using EPI.Asteroids.Indicators;
using MathNet.Numerics.Distributions;
using MathNet.Numerics.Integration;

namespace EPIStrategies
{
    public class FirstStrategy : BaseFT
    {
        public FirstStrategy(string strategyName) : base(strategyName)
        {

        }
        #region 第一个策略
        //-------------------------------------------------------------------------
--
        // 简称: FirstStrategy
        // 类别: Demo
        // 说明: 第一个策略
        //-------------------------------------------------------------------------
--
        /// <summary>
        /// 初始化策略
```

```
        /// </summary>
        /// <param name="sender">策略参数</param>
        /// <returns></returns>
        public override bool InitStrategy(object sender)
        {
            return base.InitStrategy(sender);
        }
        /// <summary>
        /// 数据加载完成
        /// </summary>
        /// <param name="contract">合约</param>
        /// <param name="cycle">周期</param>
        /// <param name="barDatas">Bar数据</param>
        /// <param name="isLast">是否最后一个数据</param>
        public override void FinishedLoadData(string contract, string cycle,
List<BarData> barDatas, bool isLast)
        {
            base.FinishedLoadData(contract, cycle, barDatas, isLast);
        }
        /// <summary>
        /// Tick行情
        /// </summary>
        /// <param name="tickData"></param>
        public override void RtnTickData(TickData tickData)
        {
            base.RtnTickData(tickData);
        }
        /// <summary>
        /// Bar行情
        /// </summary>
        /// <param name="barData">最新Bar数据</param>
        /// <param name="isNewBar">是否下一周期的Bar（Tick实时刷新Bar时用于判断是否下
一周期的Bar）</param>
        public override void RtnBarData(BarData barData, bool isNewBar)
        {
            base.RtnBarData(barData, isNewBar);
        }
        /// <summary>
        /// 委托回报
        /// </summary>
        /// <param name="rOrder">委托单</param>
        public override void RtnOrder(RspOrders rOrder)
        {
            base.RtnOrder(rOrder);
        }
        /// <summary>
        /// 成交回报
        /// </summary>
        /// <param name="rTrade">成交单</param>
        public override void RtnTrade(RspTrades rTrade)
        {
            base.RtnTrade(rTrade);
        }
        /// <summary>
        /// 消息回报
        /// </summary>
        /// <param name="rMsg">消息</param>
        public override void RtnMessage(JRspMessage rMsg)
```

```
        {
            base.RtnMessage(rMsg);
        }
        //------------------------------------------------------------------------
--
        // 编译版本 EPI2019.11.19
        // 版权所有 EPAI.TECH 2019-2029
        // 更改声明 EPAI.TECH保留对EPI平台每一版本的EPI策略的修改和重写的权利
        //------------------------------------------------------------------------
--
        #endregion
      }
}
```

`

修改代码编写KDJ金叉死叉策略，策略代码如下：

```
using EPI.Comet;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using EPI.Model;
using EPI.Asteroids.Indicators;
using MathNet.Numerics.Distributions;
using MathNet.Numerics.Integration;

namespace EPIStrategies
{
    public class FirstStrategy : BaseFT
    {
        public FirstStrategy(string strategyName) : base(strategyName)
        {

    }
        #region 第一个策略
        //------------------------------------------------------------------------
--
        // 简称: FirstStrategy
        // 类别: Demo
        // 说明: 第一个策略
        //------------------------------------------------------------------------
--
        KDJ kdj;                    //创建KDJ指标变量
        int kdjLength = 9;          //创建KDJ指标Length变量
        int kdjM1 = 3;              //创建KDJ指标M1变量
        int kdjM2 = 3;              //创建KDJ指标M2变量

        /// <summary>
        /// 初始化策略
        /// </summary>
        /// <param name="sender">策略参数,如:（9,3,3）</param>
        /// <returns></returns>
        public override bool InitStrategy(object sender)
        {
```

```
            try
            {
                var usrParams = ConvertParams(sender);
                if (usrParams.Length == 3)
                {
                    kdjLength = int.Parse(usrParams[0]);
                    kdjM1 = int.Parse(usrParams[1]);
                    kdjM2 = int.Parse(usrParams[2]);
                    //加载数据次数，对应LoadBarDatas调用次数
                    LoadDataCount = 1;
                    //加载Bar数据，用于初始化指标历史数据（取模拟数据前天数据，不需要过
多）
                    var result = LoadBarDatas(Setting.Contract, Setting.Cycle,
100, false, false);
                    return result;
                }
                else
                {
                    Log("初始化策略失败，传入参数错误");
                    return false;
                }
            }
            catch (Exception ex)
            {
                Log("初始化策略失败", ex);
                return false;
            }
        }
        /// <summary>
        /// 数据加载完成
        /// </summary>
        /// <param name="contract">合约</param>
        /// <param name="cycle">周期</param>
        /// <param name="barDatas">Bar数据</param>
        /// <param name="isLast">是否最后一个数据</param>
        public override void FinishedLoadData(string contract, string cycle,
List<BarData> barDatas, bool isLast)
        {
            try
            {
                if (contract == Setting.Contract)
                {
                    kdj = new KDJ(barDatas, kdjLength, kdjM1, kdjM2);
                    //获取合约保证金率及手续情况
                    var cTpDetail = GetContractTpDetail(contract);
                    //获取合约信息
                    Contracts cInfo = GetContract(contract);
                    Log(string.Format("合约:{0},合约乘数:{1},最小变动价位:{2},手续
费:{3},保证金:{4}",
                        cInfo.Contract, cInfo.VolumeMultiple, cInfo.PriceTick,
                        cTpDetail.FeeValue, cTpDetail.MarginValue));
                }
                //判断是否最后一笔数据，否则等待后续数据过来
                if (isLast)
                {
                    //至此整个策略初始化完成，启动策略
                    StartStrategy();
                }
```

```
            }
            catch (Exception ex)
            {
                Log("处理回报数据错误", ex);
            }
        }
        /// <summary>
        /// Tick行情
        /// </summary>
        /// <param name="tickData"></param>
        public override void RtnTickData(TickData tickData)
        {
            base.RtnTickData(tickData);
        }
        /// <summary>
        /// Bar行情
        /// </summary>
        /// <param name="barData">最新Bar数据</param>
        /// <param name="isNewBar">是否下一周期的Bar（Tick实时刷新Bar时用于判断是否下
一周期的Bar）</param>
        public override void RtnBarData(BarData barData, bool isNewBar)
        {
            try
            {
                if (barData.Contract == Setting.Contract)
                {
                    //获取当前持仓
                    var position = GetPosition();
                    //添加最新Bar至指标
                    kdj.AddBarData(barData);

                    bool isCurCrossKdjUp = CrossKdjUp();
                    bool isCurCrossKdjDown = CrossKdjDown();

                    if (isCurCrossKdjDown)
                    {
                        Log(string.Format("[{0}] KDJ死叉出现",
barData.Contract));
                        Sell(1);
                    }
                    else if (isCurCrossKdjUp)
                    {
                        Log(string.Format("[{0}] KDJ金叉出现",
barData.Contract));
                        Buy(1);
                    }
                }
            }
            catch (Exception ex)
            {
                Log("推动Bar数据时处理过程错误", ex);
            }
        }
        /// <summary>
        /// 委托回报
        /// </summary>
        /// <param name="rOrder">委托单</param>
        public override void RtnOrder(RspOrders rOrder)
```

```
        {
            base.RtnOrder(rOrder);
        }
        /// <summary>
        /// 成交回报
        /// </summary>
        /// <param name="rTrade">成交单</param>
        public override void RtnTrade(RspTrades rTrade)
        {
            base.RtnTrade(rTrade);
        }
        /// <summary>
        /// 消息回报
        /// </summary>
        /// <param name="rMsg">消息</param>
        public override void RtnMessage(JRspMessage rMsg)
        {
            base.RtnMessage(rMsg);
        }

        /// <summary>
        /// KDJ金叉
        /// </summary>
        /// <returns></returns>
        bool CrossKdjUp()
        {
            //获取前一个D值，由于索引从0开始，所以总长度(kdj.Count)减2即为最后第二个
            var preDValue = kdj.GetDValue(kdj.Count - 2);
            var preJValue = kdj.GetJValue(kdj.Count - 2);
            //获取当前D值
            var dValue = kdj.GetDValue(kdj.Count - 1);
            var jValue = kdj.GetJValue(kdj.Count - 1);
            //判断数据是否有效，JPR为内部属性，可用于判断数值是否为正常数值
            if (!JPR.IsNaN(preDValue) && !JPR.IsNaN(preJValue) &&
!JPR.IsNaN(dValue) && !JPR.IsNaN(jValue))
            {
                //前J小于D值 并且 当前J大于D值，说明J上穿D
                return preJValue < preDValue && jValue > dValue;
            }
            return false;
        }
        /// <summary>
        /// KDJ死叉
        /// </summary>
        /// <returns></returns>
        bool CrossKdjDown()
        {
            var preDValue = kdj.GetDValue(kdj.Count - 2);
            var preJValue = kdj.GetJValue(kdj.Count - 2);
            var dValue = kdj.GetDValue(kdj.Count - 1);
            var jValue = kdj.GetJValue(kdj.Count - 1);
            if (!JPR.IsNaN(preDValue) && !JPR.IsNaN(preJValue) &&
!JPR.IsNaN(dValue) && !JPR.IsNaN(jValue))
            {
                return preJValue > preDValue && jValue < dValue;
            }
            return false;
        }
```

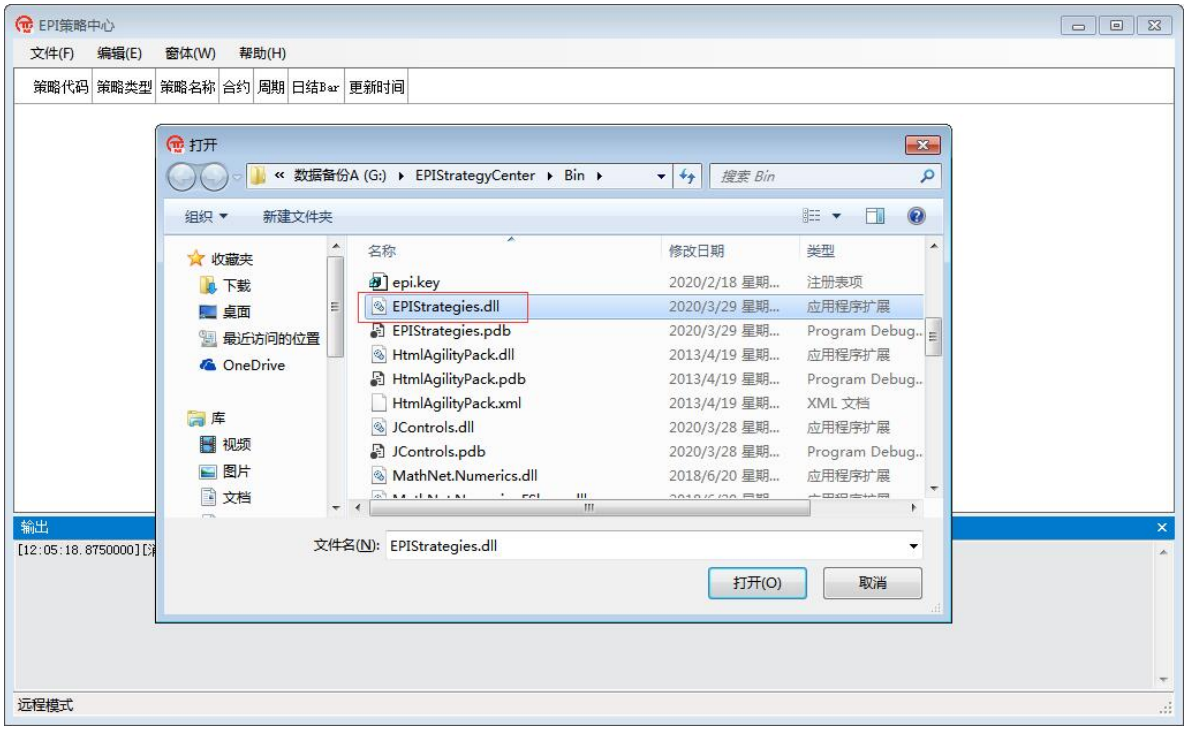
```
        //------------------------------------------------------------------------
--
        // 编译版本 EPI2019.11.19
        // 版权所有 EPAI.TECH 2019-2029
        // 更改声明 EPAI.TECH保留对EPI平台每一版本的EPI策略的修改和重写的权利
        //------------------------------------------------------------------------
--
        #endregion
    }
}
```

调试策略

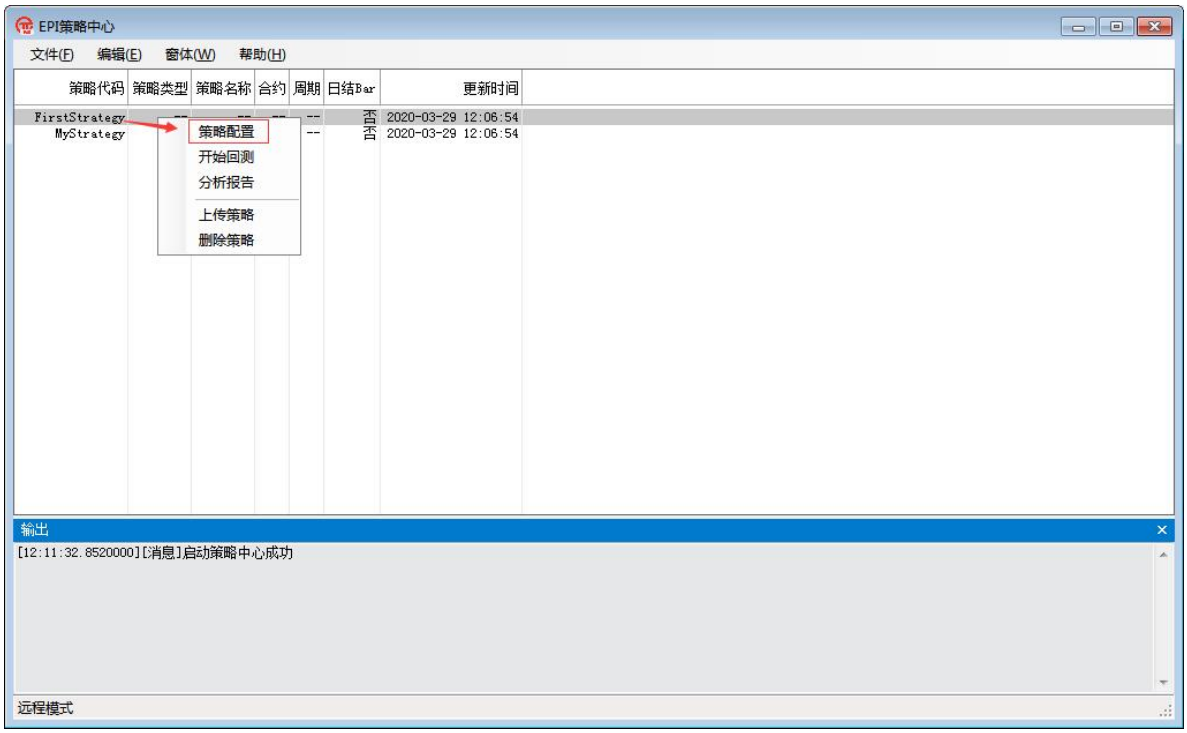
点击运行策略，如下图：

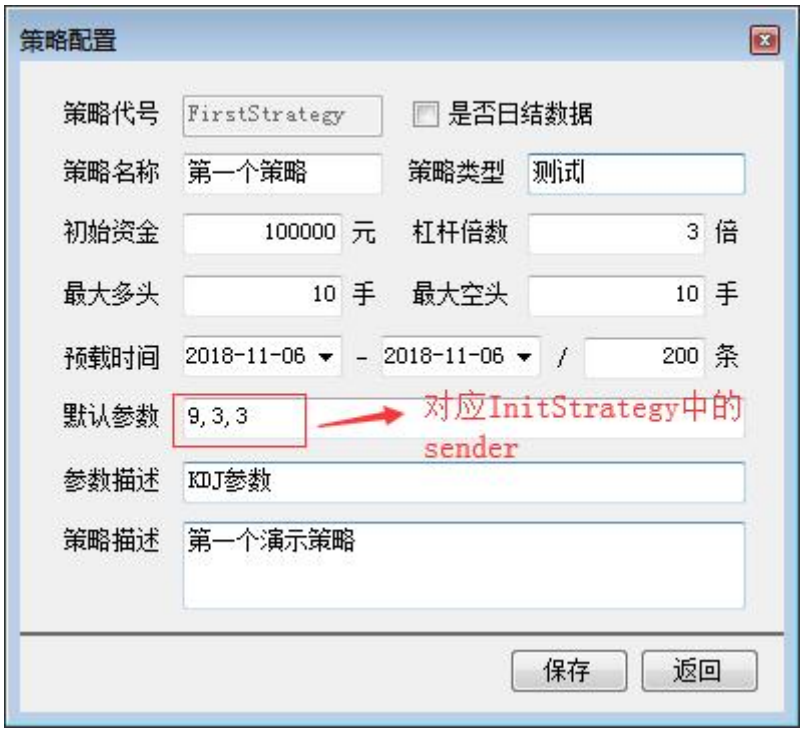
注：如果没有账号的用户，可以访问[翌派量化官网]，进行注册。

登录后，点击文件->加载策略库，选择Bin目录下的EPIStrategies，如下图：

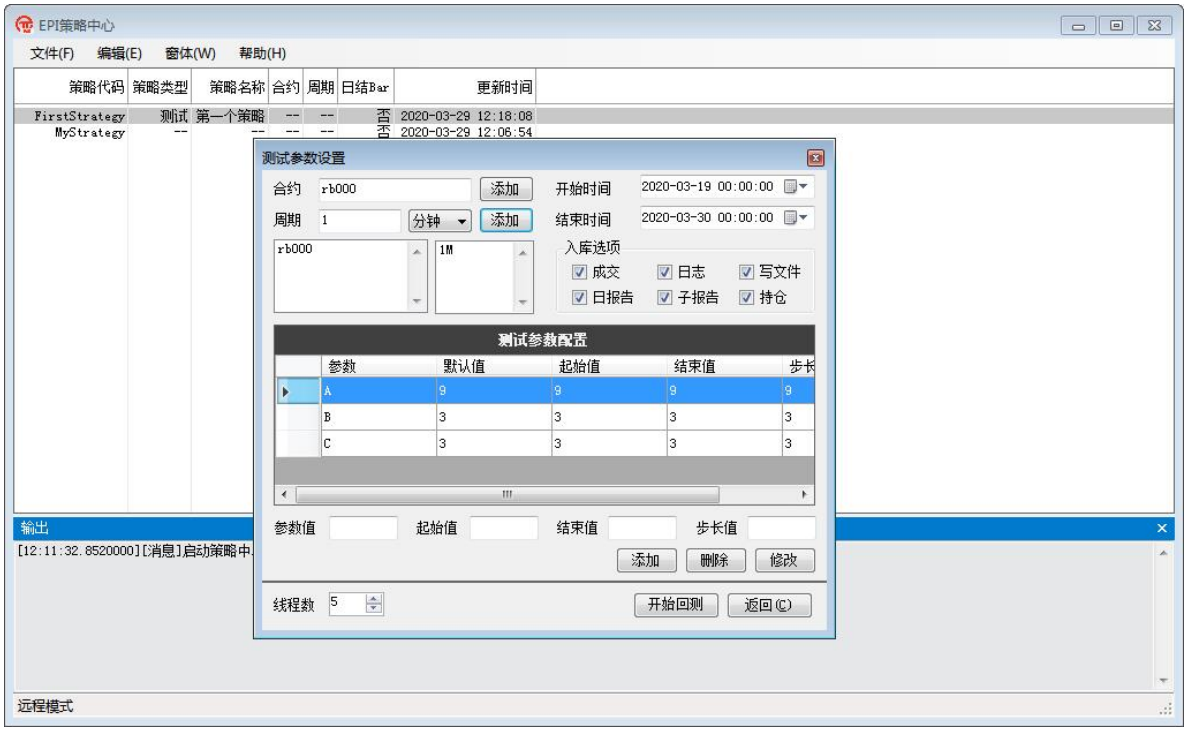


加载完成后，选择配置策略，根据自己定义的策略参数进行策略配置，如下图：

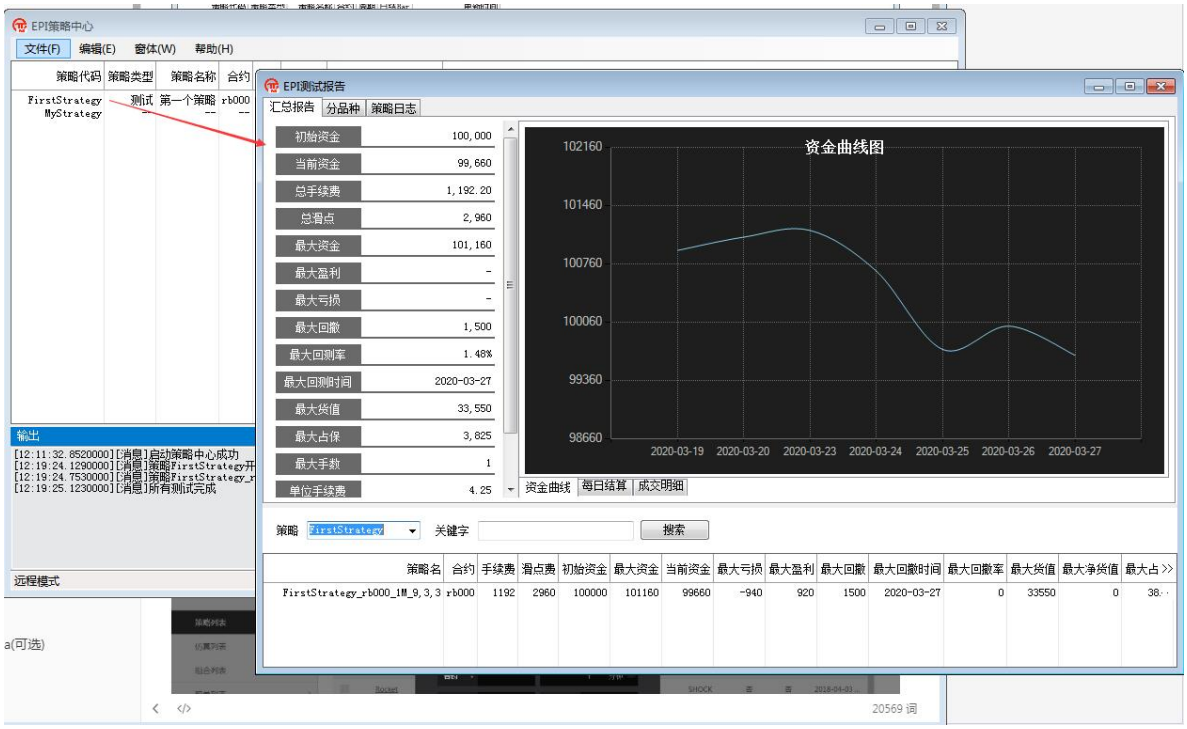




保存后右键进行策略回测，如下图：

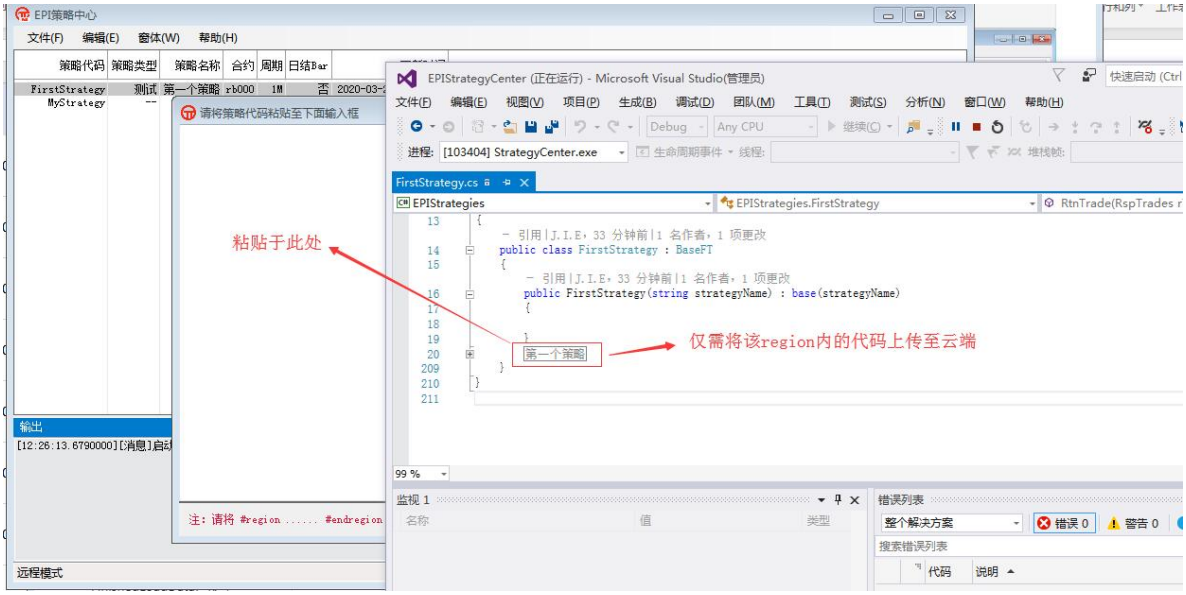


回测结束后双击策略可查看策略报告，如下图：



上传策略

回测完成想进行批量测试或者想进行仿真运行，需上传策略至云端，右键上传策略，并粘贴策略代码，如下图：

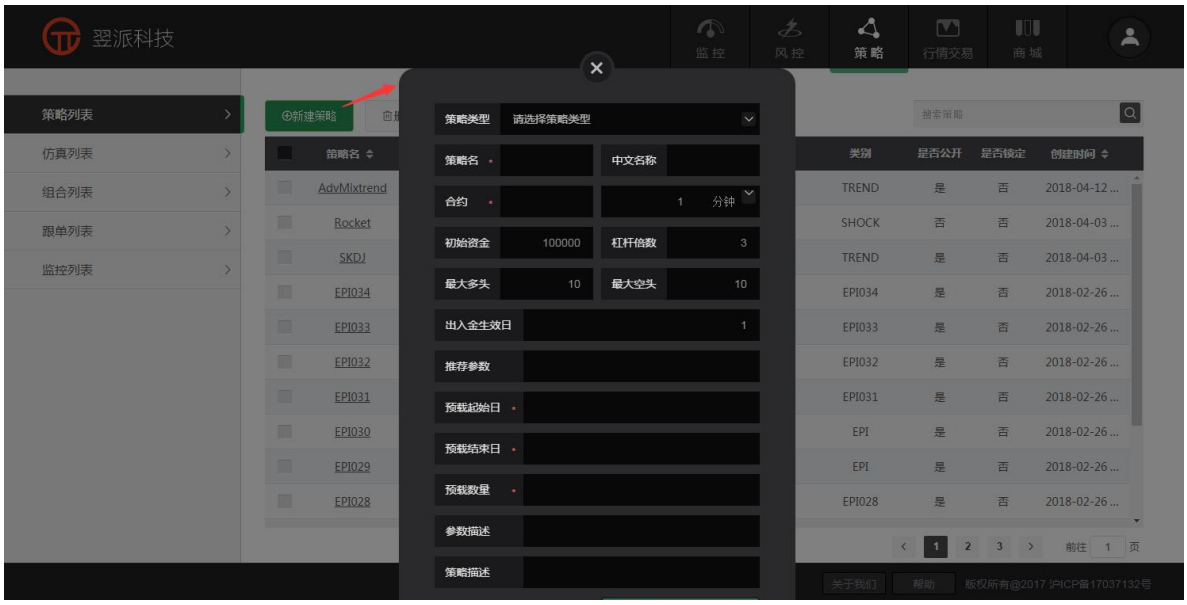


此时在云端就可以看到上传的策略，下面操作参见下面云端策略的说明。

云端策略编写

创建策略并配置参数

您可以通过在'策略列表'下点击'新建策略'来创建一个新的策略，如下图：



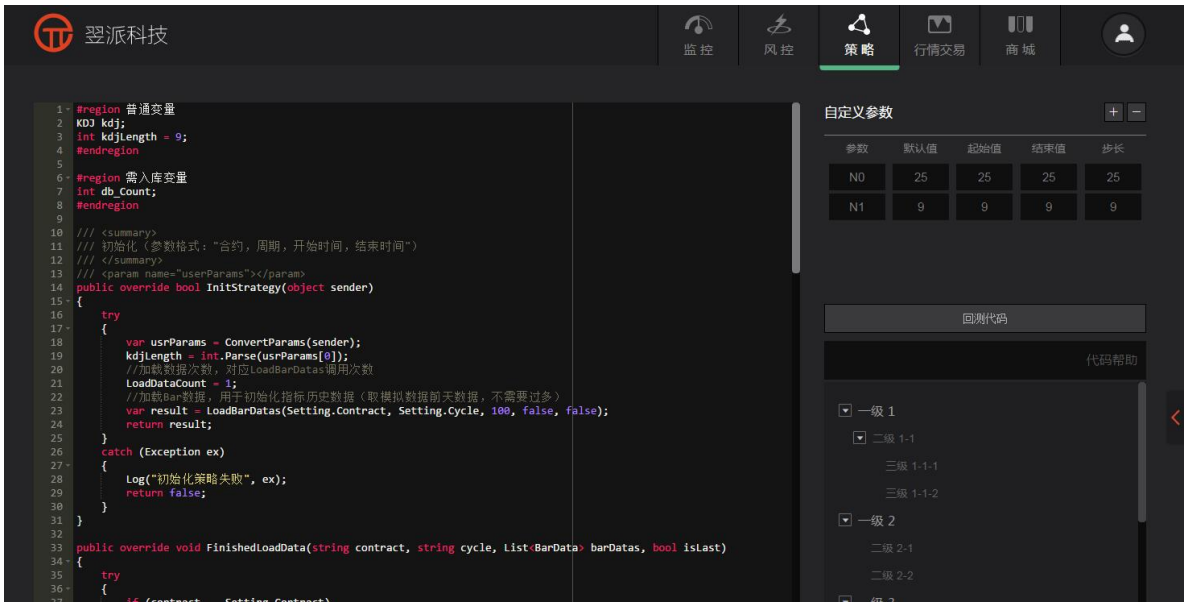
可回测合约清单（更新中）

合约代码	合约名
FG000	玻璃000
CF000	郑棉000
TA000	PTA000
SR000	白糖000
bu000	沥青000
ZC000	动煤000
a000	豆一000
au000	黄金000
ag000	白银000
al000	沪铝000
RM000	菜粕000
MA000	郑醇000
OI000	菜油000
SF000	硅铁000
SM000	猛硅000
hc000	热卷000
sn000	沪锡000
y000	豆油000
i000	铁矿000
ni000	沪镍000
p000	棕榈000
c000	玉米000
pb000	沪铅000
l000	塑料000
jm000	焦煤000
j000	焦炭000
jd000	鸡蛋000
zn000	沪锌000
cs000	淀粉000
cu000	沪铜000

合约代码	合约名
ru000	橡胶000
AP000	鲜苹果000
pp000	PP000
rb000	螺纹000
m000	豆粕000
v000	PVC000
sc000	原油000
sp000	纸浆000

编写策略代码及回测

策略编辑页面是回测的入口，如下图所示：



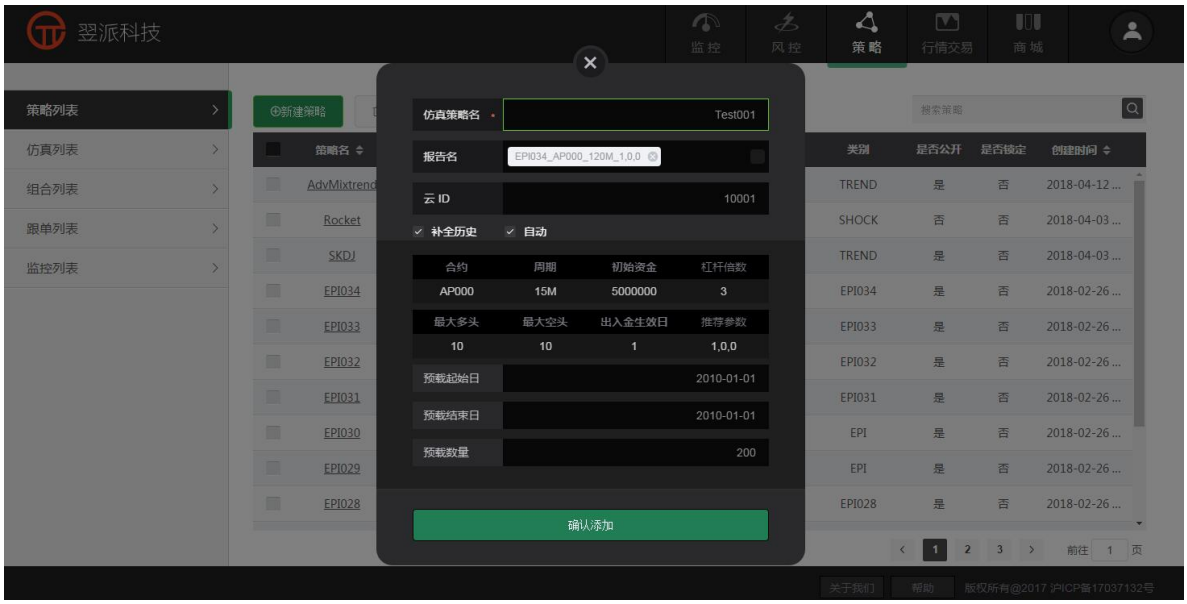
您可以在策略编辑页面中进行以下参数的设置：

- 开始日期：回测期间的开始日期
- 结束日期：回测期间的结束日期 - 如果回测结束日期在*今天*之后，将会自动使用最后一天的历史数据
- 测试参数：设置策略测试参数
- 全品种测试：批量全品种测试

在设置完回测参数之后，您可以点击'回测'来对策略进行测试，系统会返回策略错误信息和测试结果，点击测试详情可看到完整的测试报告。

策略仿真运行

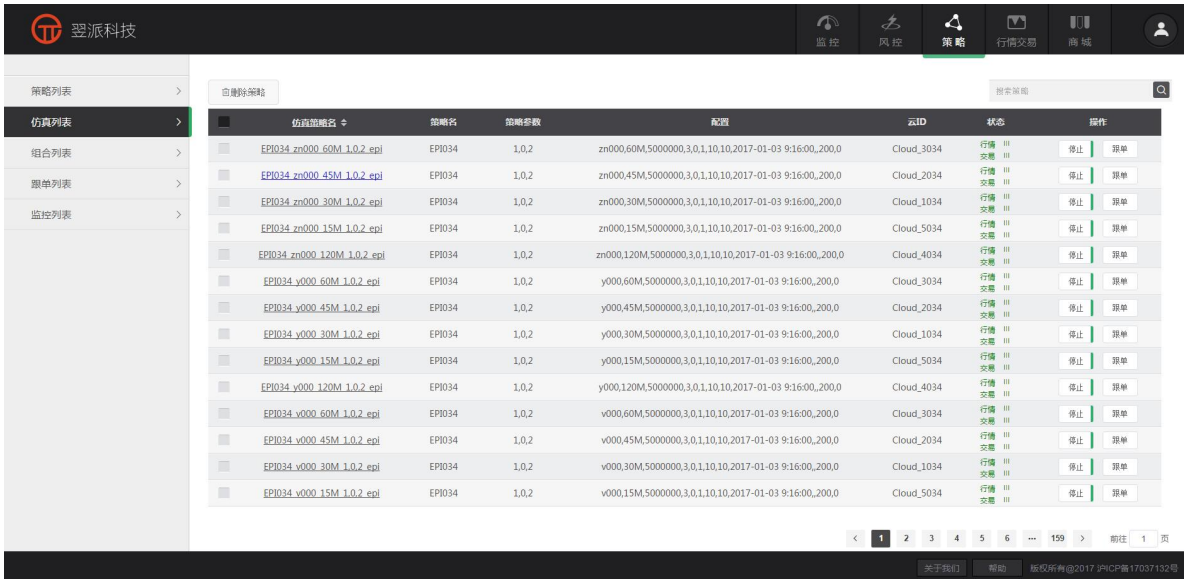
在完成回测并产生报表后，可以选择对应的测试报告进行仿真运行，如下图：



仿真配置如下：

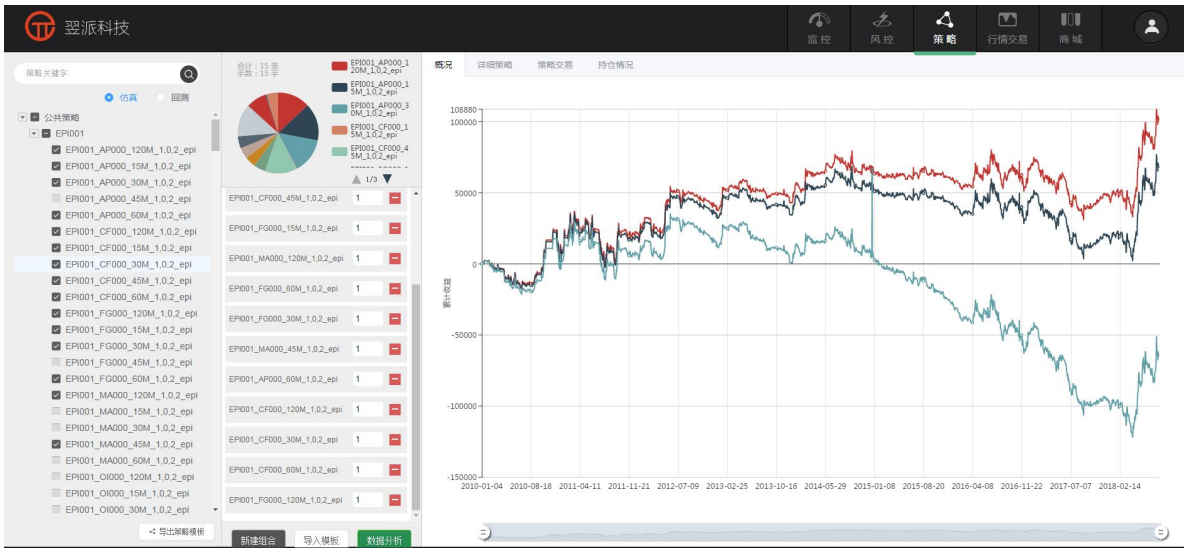
- 仿真策略名：用于仿真跟踪的策略名称
- 对应回测报告：根据策略对应测试报告来决定仿真哪组策略配置（合约、周期、初始资金等）
- 云ID：云端的资源编号
- 是否补全历史：用于回测策略数据和仿真策略数据的无缝衔接
- 是否自动：运行云端每天自动启动策略
- 推荐参数：本次策略运行的自定义参数

添加仿真成功后，可以在仿真队列中进行策略的状态管理，如下图：



策略组合

我们还提供策略组合分析工具，更便捷快速的分析策略间组合的情况，并可以自由配比，组合的策略组也可以进行实盘运行。



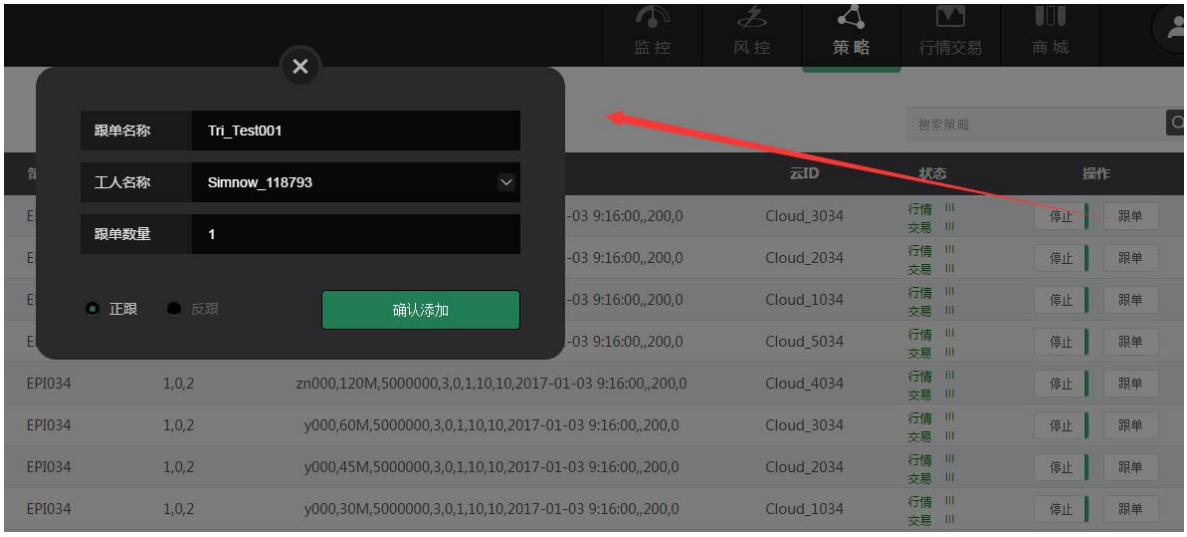
资金分析	
累计收益	67990.88
累计手续费	33054.12
累计滑点费	132715.00
历史最大回撤金额	65930.90
历史最大回撤完成时间	2018-08-06
历史最大回撤持续时间	1411
当前回撤金额	11795.10
当前回撤撤持续时间	6
手续费/收益	0.49
日均收益金额	32.38
交易日数	2100
累计收益/最大历史回撤	1.03
多头最大占用保证金	98513.40
空头最大占用保证金	70320.40
多空最大占用保证金	109169.40
当前多头保证金	0.00
当前空头保证金	0.00
当前总保证金	4235.92
当前回撤/最大回撤	0.18

资金分析(含滑点)	
累计收益	-64724.12
累计手续费	33054.12
累计滑点费	132715.00
历史最大回撤金额	41030.00
历史最大回撤完成时间	2012-06-04
历史最大回撤持续时间	447
当前回撤金额	189542.90
当前回撤撤持续时间	1209
手续费/收益	-0.51
日均收益金额	-30.82
交易日数	2100
累计收益/最大历史回撤	-1.58
多头最大占用保证金	98513.40
空头最大占用保证金	70320.40
多空最大占用保证金	109169.40
当前多头保证金	0.00
当前空头保证金	0.00
当前总保证金	
当前年度累计收益	33614.08

年份	当前回撤	当前回撤持续时间	历史最大回撤	历史最大回撤持续时间	历史最大回撤完成时间
2010	20386.50	23	20348.00	283	2010-10-25
2011	38030.00	140	20348.00	283	2010-10-25
2012	17934.00	207	20348.00	283	2010-10-25
2013	19056.50	224	20348.00	283	2010-10-25
2014	16935.00	49	20348.00	283	2010-10-25
2015	11344.40	174	20348.00	283	2010-10-25
2016	31409.50	150	20348.00	283	2010-10-25
2017	48043.40	228	20348.00	283	2010-10-25
2018	11795.10	6	20673.10	97	2018-05-14
whole	11795.10	6	65930.90	1411	2018-08-06

策略实盘运行

根据策略仿真运行的情况，可以实时切换到实盘运行，只需要点击跟单即可，如下图：



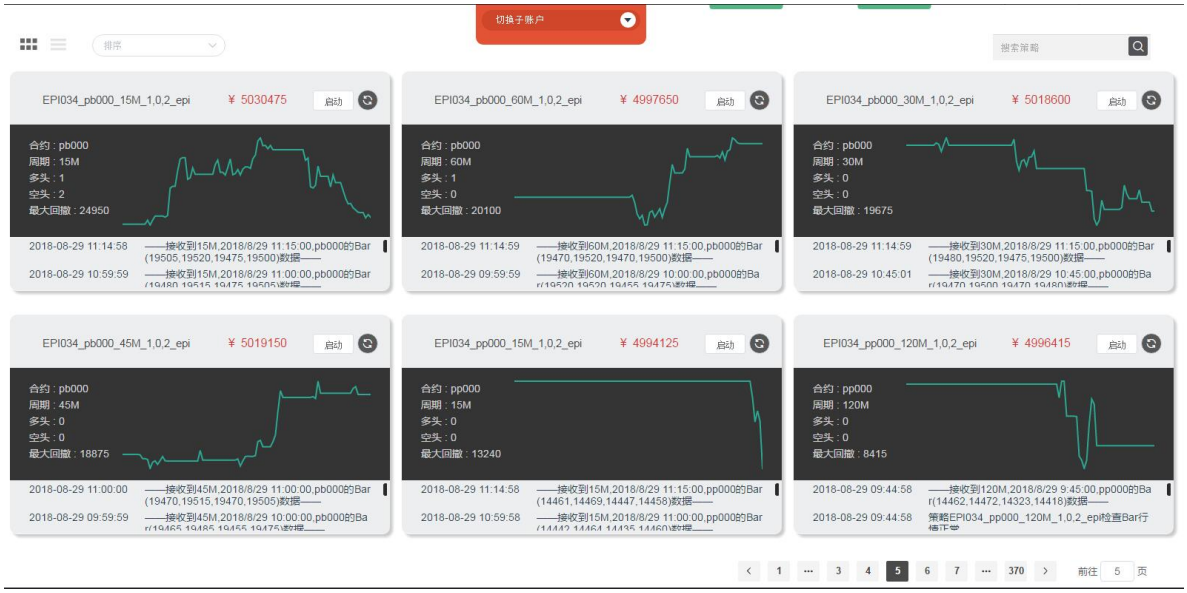
添加跟单后，即可在跟单队列中维护实盘运行状况，实盘运行有以下几个步骤完成：

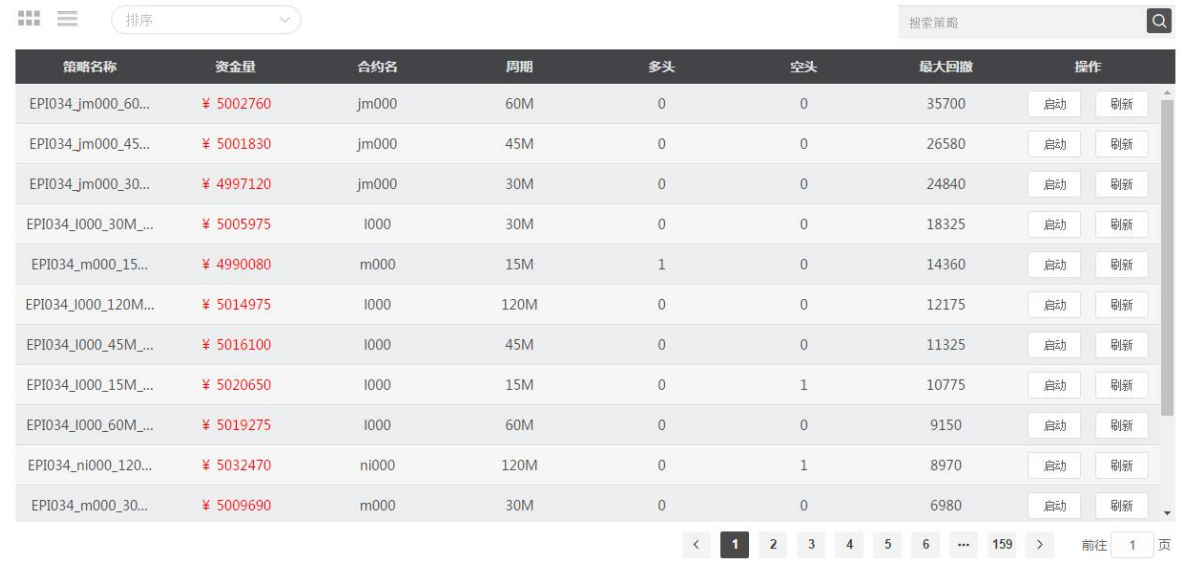
1. 读取跟单配置（名称、倍数、方向、实盘账号）
2. 同步策略持仓至实盘（将策略虚拟持仓同步至实盘）
3. 信号订阅并下单

当用户停止跟单后系统会自动将仓位清空。

策略监控

所有的仿真运行策略都可以通过策略监控界面进行实盘查看，如下图：





EPI SDK 简介

基础方法

InitStrategy(必须实现)

```
public override bool InitStrategy(object sender)
```

初始化方法 - 只会在启动的时候触发一次。你的策略会使用这个方法来设置你需要的各种初始化配置。`sender` 用于传入用户自定义参数。

参数

参数	类型	注释
sender	string	用户传入字符串形式的自定义参数，每个值间以‘，’分割。

返回

布尔

范例

```
/// <summary>
/// 初始化策略
/// </summary>
/// <param name="sender">如:（9,3,3）</param>
public override bool InitStrategy(object sender)
{
    try
    {
        var usrParams = ConvertParams(sender);
        if (usrParams.Length == 3)
        {
            kdjLength = int.Parse(usrParams[0]);
            kdjM1 = int.Parse(usrParams[1]);
            kdjM2 = int.Parse(usrParams[2]);
            //加载数据次数，对应LoadBarDatas调用次数
            LoadDataCount = 1;
            //加载Bar数据，用于初始化指标历史数据（取模拟数据前天数据，不需要过多）
            var result = LoadBarDatas(Setting.Contract, Setting.Cycle, 100,
false, false);
            return result;
        }
        else
        {
            Log("初始化策略失败，传入参数错误");
            return false;
        }
    }
    catch (Exception ex)
    {
        Log("初始化策略失败", ex);
        return false;
    }
}
```

FinishedLoadData(可选)

```
public override void FinishedLoadData(string contract, string cycle,
List<BarData> barDatas, bool isLast)
```

数据完整完成后回调方法 - 根据**LoadBarDatas**的次数来决定调用几次。该方法用于获取历史行情数据，并且进行初始化参数或指标等工作。

参数

参数	类型	注释
contract	string	本次返回数据的合约名
cycle	string	本次返回数据的合约周期
barDatas	List	本次返回的历史数据
isLast	bool	是否最后一次返回（LoadDataCount与LoadBarDatas调用次数必须一致，否则导致判断错误）

返回

无

范例

```
/// <summary>
/// 完成加载数据
/// </summary>
/// <param name="contract">合约</param>
/// <param name="cycle">周期</param>
/// <param name="barDatas">数据集合</param>
public override void FinishedLoadData(string contract, string cycle,
List<BarData> barDatas, bool isLast)
{
    try
    {
        if (contract == Setting.Contract)
        {
            kdj = new KDJ(barDatas, kdjLength, kdjM1, kdjM2);
            //获取合约保证金率及手续情况
            UserTradeTpDetails tplDetail = UserTradeTpDetails(contract);
            //获取合约信息
            Contracts cInfo = GetContract(contract);
            Log(string.Format("合约:{0},合约乘数:{1},最小变动价位:{2},手续费:{3},保证
金:{5}",
                cInfo.Contract, cInfo.VolumeMultiple, cInfo.PriceTick,
                tplDetail.FeeValue , cTpDetail.MarginValue));
        }
        //判断是否最后一笔数据，否则等待后续数据过来
        if (isLast)
        {
            //至此整个策略初始化完成，启动策略
            StartStrategy();
        }
    }
    catch (Exception ex)
    {
        Log("处理回报数据错误", ex);
    }
}
```

RtnBarData(可选)

```
public override void RtnBarData(BarData barData, bool isNewBar)
```

生成新Bar后的回调方法 - 每根Bar过来都会调用。

参数

参数	类型	注释
barData	BarData	Bar数据（开、高、低、收、量等）
isNewBar	bool	是否完整Bar（当启用Tick更新Bar时，来区分是否完整的Bar）

返回

无

范例

```
/// <summary>
/// 新数据主动推送
/// </summary>
/// <param name="barData">Bar数据</param>
/// <param name="isNewBar">是否完整Bar（当启用Tick更新Bar时，区分是否完整的Bar）
</param>
public override void RtnBarData(BarData barData, bool isNewBar)
{
    try
    {
        if (barData.Contract == Setting.Contract)
        {
            //获取当前持仓
            var position = GetPosition();
            //添加最新Bar至指标
            kdj.AddBarData(barData);

            bool isCurCrossKdjUp = CrossKdjUp();
            bool isCurCrossKdjDown = CrossKdjDown();

            if (isCurCrossKdjDown)
            {
                Log(string.Format("[{0}] KDJ死叉出现", barData.Contract));
                Sell(1);
            }
            else if (isCurCrossKdjUp)
            {
                Log(string.Format("[{0}] KDJ金叉出现", barData.Contract));
                Buy(1);
            }
        }
    }
    catch (Exception ex)
    {
        Log("推动Bar数据时处理过程错误", ex);
    }
}
```

RtnTickData(可选)

```
public override void RtnTickData(TickData tickData)
```

接收最新Tick行情数据后的回调方法 - 每Tick数据过来都会调用。

参数

参数	类型	注释
tickData	TickData	Tick数据（最新价、卖一价、卖一量、买一价、买一量等）

返回

无

范例

```
/// <summary>
/// Tick实时数据
/// </summary>
/// <param name="tickData">Tick数据</param>
public override void RtnTickData(TickData tickData)
{
    if (tickData.Contract == Setting.Contract)
    {
        var lastPrice = tickData.LastPrice;
    }
}
```

RtnMessage(可选)

```
public override void RtnMessage(JRspMessage rMsg)
```

接收最新Tick行情数据后的回调方法 - 每Tick数据过来都会调用。

参数

参数	类型	注释
rMsg	JRspMessage	通常会收到一些错误类消息（发单错误、撤单错误、限制消息等）

返回

无

范例

```
/// <summary>
/// 消息回报
/// </summary>
/// <param name="rMsg">消息体</param>
public override void RtnMessage(JRspMessage rMsg)
{
    Log(string.Format("[0]{1}", rMsg.MessageType.ToString(),
rMsg.MessageContent));
}
```

RtnOrder(可选)

```
public override void RtnOrder(RspOrders rOrder)
```

接收委托回报的回调方法 - 每次有订单回报过来都会调用。

参数

参数	类型	注释
rOrder	RspOrders	委托单

返回

无

范例

```
/// <summary>
/// 委托回报
/// </summary>
/// <param name="rOrder">委托单</param>
public override void RtnOrder(RspOrders rOrder)
{
    if (rOrder.EOrderStatus == EnumOrderStatus.AllTraded)
    {
        //全部成交
    }
}
```

RtnTrade(可选)

```
public override void RtnTrade(RspTrades rTrade)
```

接收成交回报的回调方法 - 每次有成交回报过来都会调用。

参数

参数	类型	注释
rTrade	RspTrades	成交单

返回

无

范例

```
/// <summary>
/// 成交回报
/// </summary>
/// <param name="rTrade">成交单</param>
public override void RtnTrade(RspTrades rTrade)
{
    if (rTrade.Contract == Setting.Contract)
    {
        //单笔成交记录
    }
}
```

交易相关函数

StartStrategy(必须调用)

用于启动策略。

参数

无

返回

bool

StopStrategy(可选)

用于停止策略。

参数

无

返回

bool

LoadBarDatas(可选)

加载历史Bar数据。

参数

参数	类型	注释
contract	string	合约
cycle	string	周期
takeNumber 或 start,end	int	获取最近N条 或 开始时间，结束时间
isFuzzy	bool	是否模糊查询(用于查询同月合约)
isBarBaseOnTick	bool	是否基于Tick动态生成Bar

返回

无 - 异步回调

ConvertParams(可选)

用于将初始化策略中的object参数转换为字符串数组。

参数

参数	类型	注释
sender	object	初始化策略参数

返回

string[] - 字符串参数数组

SetNoWaitingMode(可选)

设置非等待模式（不等待成交回报,直接以盘口价作为成交价来更新持仓）。

参数

无

返回

无

Buy(可选)

买入。

参数

参数	类型	注释
number	int	报单数量
price	double	报单价格

参数	类型	注释
number	int	数量
sendPriceType	int	发单类型（0-对价 1-超价 2-挂量小价）
advPoint	int	超价打单点数

返回

string - 报单编号

Sell(可选)

买入。

参数

参数	类型	注释
number	int	报单数量
price	double	报单价格

参数	类型	注释
number	int	数量
sendPriceType	int	发单类型（0-对价 1-超价 2-挂量小价）
advPoint	int	超价打单点数

返回

string - 报单编号

Cancel(可选)

用于撤销订单。

参数

参数	类型	注释
clientOrderId	string	订单编号

返回

bool - 是否发送撤单成功

GetContract(可选)

获取合约信息。

参数

参数	类型	注释
contract	string	合约代码

返回

Contracts- 合约信息

GetContractTpDetail(可选)

获取交易模版信息。

参数

参数	类型	注释
contract	string	合约代码

返回

UserTradeTpDetails- 用户交易模版信息

GetPosition(可选)

获取策略持仓。

参数

无

返回

StrategyPositions

GetReport(可选)

获取策略报告。

参数

无

返回

StrategyReports

Log(可选)

写日志。

参数

参数	类型	注释
msg	int	消息
isError	bool	是否错误

参数	类型	注释
title	int	标题
ex	Exception	异常

返回

string - 报单编号

SendMail(可选)

发送邮件。

参数

参数	类型	注释
title	string	标题
content	string	内容
toAccount	string	邮件账户

返回

bool - 是否发送成功

InitDBFields(可选)

初始化入库字段，如果用户想将变量保存至服务器，只需以"db_"开头声明变量，如下：

```
double db_double1;
string db_str1;
int db_int1;
public override bool InitStrategy(object sender)
{
    //从服务器初始数据库字段
    InitDBFields();
}
```

SaveDBFields(可选)

保存入库字段，如果用户想将变量保存至服务器，只需以"db_"开头声明变量，如下：

```
double db_double1;
string db_str1;
int db_int1;
public override void RtnBarData(BarData barData, bool isNewBar)
{
    //修改数据库字段值
    db_double1 = barData.Close;
    db_str1 = "test";
    db_int1++;
    //保存数据库字段至服务器
    SaveDBFields();
}
```

交易相关属性

属性	类型	注释
LoadDataCount	*int*	加载数据次数（需要与调用LoadBarDatas次数一致）
Setting	*StrategySettings*	策略配置
TradeDay	*string*	交易日
IsClosingTime	*bool*	是否收盘中

枚举对象

EnumSide对象

属性	类型	注释
Buy	*byte*	买入
Sell	*byte*	卖出

EnumOrderStatus对象

属性	类型	注释
AllTraded	*byte*	全部成交
PartTradedQueueing	*byte*	卖部分成交还在队列中
PartTradedNotQueueing	byte	部分成交不在队列中
NoTradeQueueing	byte	未成交还在队列中
NoTradeNotQueueing	byte	未成交不在队列中
Canceled	byte	撤单
Limit	byte	限制
Unknown	byte	未知
NotTouched	byte	尚未触发
Touched	byte	已触发
CenterReceived	byte	交易中心已收到
AlgServerReceived	byte	算法服务器已接收
Error	byte	错误
WaitCancel	byte	待撤

EnumMessageType对象

属性	类型	注释
SendError	*byte*	发单错误
CancelError	*byte*	撤单错误
CloudMsg	byte	云端消息
AuditMsg	byte	审核消息
AccountMsg	byte	出入金消息
LimitMsg	byte	限制消息（挂撤次数已满）

重要对象

Contracts对象

属性	类型	注释
Contract	*string*	合约代码
ContractName	*string*	合约名
ContractType	*string*	合约类别
Exchange	*string*	交易所
VolumeMultiple	*double*	合约乘数
PriceTick	*double*	最小变动价位
OpenInterest	*double*	持仓量
Offset	*double*	偏移量(按成交量)
Offset1	*double*	偏移量(按持仓量)
TimeTpDetailList	List	交易时间模版明细

TimeTpDetails对象

属性	类型	注释
StartTime	*string*	开始时间
BiddingTime	*string*	集合竞价时间
EndTime	*string*	结束时间
IsOpen	*bool*	是否开盘点
IsClose	*bool*	是否收盘点

UserTradeTpDetails对象

属性	类型	注释
Contract	*string*	合约
MarginValue	*double*	保证金值
MarginType	*int*	保证金类型（0-百分比，1-绝对值）
FeeValue	*double*	是否数字
FeeType	*int*	手续费类型（0-百分比，1-绝对值）

BarData对象

属性	类型	注释
Contract	*string*	合约代码
Open	*double*	开盘价
Close	*double*	收盘价
High	*double*	最高价
Low	*double*	最低价
Volume	*double*	成交量
OpenInterest	*double*	持仓量
Amount	*double*	成交金额
TradingDay	*string*	交易日期（yyyyMMdd）
RealDay	*string*	真实日期（yyyyMMdd）
UpdateTime	*string*	更新时间(HH:mm:ss)
RealDateTime	*DateTime*	最新时间
Offset	*double*	偏移量
Cycle	*string*	周期（1M,3M,5M,15M,1H等）

TickData对象

属性	类型	注释
Contract	*string*	合约代码
LastPrice	*double*	最新价
AskPrice1	*double*	卖一价
AskPrice2	*double*	卖二价
AskPrice3	*double*	卖三价
AskPrice4	*double*	卖四价
AskPrice5	*double*	卖五价
AskVolume1	*double*	卖一量
AskVolume2	*double*	卖二量
AskVolume3	*double*	卖三量
AskVolume4	*double*	卖四量
AskVolume5	*double*	卖五量
BidPrice1	*double*	买一价
BidPrice2	*double*	买二价
BidPrice3	*double*	买三价
BidPrice4	*double*	买四价
BidPrice5	*double*	买五价
Volume	*double*	成交量
OpenInterest	*double*	持仓量

属性	类型	注释
Amount	*double*	成交金额
TradingDay	*string*	交易日期（yyyyMMdd）
RealDay	*string*	真实日期（yyyyMMdd）
UpdateTime	*string*	更新时间(HH:mm:ss)
UpdateMillisec	*int*	毫秒
RealDateTime	*DateTime*	最新时间
OpenPrice	*double*	开盘价
PreClosePrice	*double*	前收盘
PreSettlementPrice	*double*	前结算
HighestPrice	*double*	当日最高价
LowestPrice	*double*	当日最低价
UpperLimitPrice	*double*	涨停价
LowerLimitPrice	*double*	跌停价

StrategySettings对象

属性	类型	注释
StrategyName	*string*	策略名称
Client	*string*	用户
Contract	*string*	合约
Cycle	*string*	周期
InitMoney	*double*	初始资金
Leverage	double	杠杆比例
MaxLongVolumeAllowed	*int*	多头最大允许头寸
MaxShortVolumeAllowed	*int*	空头最大允许头寸
InOutMoneyDays	*int*	出入金有效时间
CloseOffsetSeconds	*int*	收盘钱N秒清仓
UserParams	*string*	默认自定义策略参数
Worker	*string*	交易工人
PreLoadStart	string	预加载开始时间
PreLoadEnd	string	预加载结束时间
PreLoadNumber	int	预加载数量
IsEndOfDay	bool	Bar数据是否日结算

StrategyReports对象

属性	类型	注释
StrategyName	*string*	策略名称
Client	*string*	用户
Contract	*string*	合约
Cycle	*string*	周期
InitMoney	*double*	初始资金
Money	double	当前资金
Commission	*double*	手续费
SlippageFee	double	滑点费
MaxMoney	*double*	最大资金
MaxLoss	*double*	最大亏损
MaxWin	*double*	最大盈利
MaxBack	*double*	最大回撤
MaxBackRate	*double*	最大回撤率
MaxBackTime	*DateTime*	最大回撤时间
MaxMarketValue	double	最大货值
MaxNetMarketValue	double	最大净货值
MaxUsedMargin	double	最大占用保证金
MaxVolume	*int*	最大手数
TradeDayCount	*int*	总交易天数
CurHoldDayCount	*int*	当前持仓天数
MaxHoldDayCount	*int*	最大持仓天数
TotalInOutMoney	double	总出入金

StrategyPositions对象

属性	类型	注释
StrategyName	*string*	策略名称
Client	*string*	用户
Contract	*string*	合约
LongPrice	*double*	多头均价
LongVolume	*int*	多头数量
ShortPrice	double	空头均价
ShortVolume	*int*	空头数量
TotalLongPrice	double	累计多头均价
TotalLongVolume	*int*	累计多头数量
TotalShortPrice	*double*	累计空头均价
TotalShortVolume	*int*	累计空头数量
PendingLongVolume	*int*	多头挂单中数量
PendingShortVolume	*int*	空头挂单中数量
PreLongPrice	*double*	前一次多头成交价
PreShortPrice	double	前一次空头成交价
NetVolume	int	净头寸

RspOrders对象

属性	类型	注释
ClientOrderId	*string*	订单编号
OrderName	*string*	订单名称
ESide	*EnumSide*	买卖(Buy,Sell)
InsertPrice	*double*	报单价格
InsertVolume	*int*	报单数量
InsertTime	string	报单时间
TradedPrice	*double*	成交均价
TradedVolume	int	成交数量
TradedTime	*string*	成交时间
EOrderStatus	*EnumOrderStatus*	订单状态
SpecialContract	*int*	特殊合约（000,666,777,999等）
SCOffset	*int*	偏移量

属性	类型	注释
UsrContract	*int*	还原后的合约（如用主力合约，rb1805还原后即rb000）
UsrInsertPrice	*double*	还原后报单价格（真实合约价格+偏移量）
UsrTradedPrice	*double*	还原后成交价格（真实合约价格+偏移量）
StrategyId	*string*	策略标识
Client	*string*	用户名
Sender	*string*	发送者
Worker	*string*	工人账号
TradeDay	*string*	交易日
IsFinished	*bool*	是否订单已完成
IsQueueing	*bool*	是否在队列中
IsCanceled	*bool*	是否已撤单

RspTrades对象

属性	类型	注释
ClientOrderId	*string*	订单编号
Contract	*string*	单合约名称
ESide	*EnumSide*	买卖(Buy,Sell)
TradedPrice	*double*	成交价格
TradedVolume	*int*	成交数量
Commission	double	手续费
SlippageFee	*double*	滑点费
TradedTime	string	成交时间
UsrTradedPrice	*double*	还原后成交价格（真实合约价格+偏移量）
TradeDay	*string*	交易日
SpecialContract	*int*	特殊合约（000,666,777,999等）
SCOffset	*int*	偏移量

属性	类型	注释
StrategyId	*int*	策略名
Client	*double*	用户
Sender	*double*	发送者
Worker	*string*	工人账号

指标对象

创建对象

```
// 创建KDJ指标对象
// barDatas为K线数据
// （9,3,3）为指标参数
// 具体参数说明请参阅指标清单
KDJ kdj = new KDJ(barDatas, 9, 3, 3);
```

公共属性

属性	类型	注释
Name	string	指标名称
Description	string	指标描述
IsShowInMain	bool	是否在主图上显示
Tag	string	标签
ValueDict	Dictionary<string, List>	数据字典
GraphDict	Dictionary<string, IndicatorGraph>	图形字典
LastDateTime	DateTime	最新时间
FirstDateTime	DateTime	初始时间
Count	int	数值数量

公共方法

GetBarData(可选)

根据时间或者索引（0->最新一根，1->前一根,以此推类）获取Bar数据。

参数

参数	类型	注释
index	int	索引

参数	类型	注释
dateTime	DateTime	时间

返回

BarData- Bar数据

BindData(可选)

绑定数据，初始化指标时候用。

参数

参数	类型	注释
datas	List	bar数据

返回

无

AddBarData(可选)

添加Bar数据至最后。

参数

参数	类型	注释
bar	BarData	bar数据

返回

无

UpdateBarData(可选)

更新Bar数据。

参数

参数	类型	注释
bar	BarData	bar数据

返回

无

InsertBar(可选)

插入Bar数据。

参数

参数	类型	注释
index	int	索引
bar	BarData	bar数据

返回

无

指标清单

指标代码	指标名称
ADTM	动态买卖器指标(&ADTM)
AMA	佩里·考夫曼自适应移动平均线(&AMA)
ARBR	人气意愿指标(&ARBR)
ATR	平均真实区域(&ATR)
ATRGrid1	波动网格1(&ATRGrid1)
ATRGrid2	波动网格2(&ATRGrid2)
B3612	三减六日乖离(&B3612)
BBI	牛熊指数(&BBI)
BIAS	乖离率(&BIAS)
BOLL	布林通道(&BOLL)
BOLLA	布林通道A(&BOLLA)
BSM	BSM(&BSM)
CCI	商品通道指标(&CCI)
CDP	CDP逆势操作(&CDP)
CP	CR能量指标(&CP)
Channel	通道(&Channel)
DBCD	异同离差乖离率(&DBCD)
DDI	方向标准离差指数(&DDI)
DKX	多空线(&DKX)
DMA	平行线差(&DMA)
DMAB	日均线回归(&DMAB)
DMI	趋向指标(&DMI)
EMA	指数移动平均(&EMA)
Envalops	轨道线(&Envalops)
HCL	均线通道(&HCL)
HISVAR	历史潜在最大损失值(&HISVAR)
HighLow	最高最低价(&HighLow)
IMPVAR	近期潜在最大损失值(&IMPVAR)
KDJ	随机指标(&KDJ)
Kelther	肯特纳通道(&Kelther)

指标代码	指标名称
LLT	低延迟趋势线(&LLT)
MACD	指数平滑异同平均线(&MACD)
MTM	动力指标(&MTM)
Mike	均线通道(&Mike)
OBV	能量潮(&OBV)
OpenInst	持仓量
PUBU	瀑布线(&PUBU)
RC	变化率指标(&RC)
RJX	日均线(&RJX)
RSI	相对强弱指数(&RSI)
SAR	抛物线转向(&SAR)
SMA	简单移动平均(&SMA)
SPL	分笔分段(&SPL)
TRA	三角形态(&TRA)
VOL	成交量(&VOL)
VSD	波动标准差(&VSD)
Volatility	波动率(&Volatility)
VolatilityChannel	波动率通道(&VolatilityChannel)
WMA	权重移动平均(&WMA)
Williams	威廉指标(&Williams)

注：EPI所有策略内部索引都是按顺序规则来取值，并且索引从0开始，特殊情况会有特殊说明。

ADTM

动态买卖器指标

原理

1、如果开盘价 <= 昨日开盘价，DTM = 0，如果开盘价 > 昨日开盘价，DTM = (最高价 - 开盘价)和(开盘价 - 昨日开盘价)的较大值。 2、如果开盘价 >= 昨日开盘价，DBM = 0，如果开盘价 < 昨日开盘价，DBM = (开盘价 - 最低价)和（开盘价 - 昨日开盘价）的较大值。 3、STM = DTM在N个周期内的和。 4、SBM = DBM在N个周期内的和。 5、如果STM > SBM,ADTM = (STM - SBM) / STM,如果STM = SBM,ADTM = 0, 如果STM < SBM,ADTM = (STM - SBM) / SBM。 6、ADTMMA为ADTM在某周期内的简单移动平均。

用法

1、该指标在 + 1到 - 1之间波动； 2、低于 - 0.5时为很好的买入点,高于 + 0.5时需注意风险。

公共函数

```
//构造函数
ADTM(List<BarData> bars, int n = 23, int m = 8, bool isShowInMain = false,
string tag = "");
//取值函数
List<double> GetADTMValues();
List<double> GetADTMMAValues();
double GetADTMValue(int index);
double GetADTMMAValue(int index);
double GetLastADTM();
double GetLastADTMMA();
```

AMA

佩里·考夫曼自适应移动平均线

原理

无

用法

无

公共函数

```
//构造函数
AMA(List<BarData> bars, int length = 5, double smoothlength=3.1,bool
isShowInMain = true, string tag = "1");
//取值函数
List<double> GetAMAValues();
List<double> GetAMAEfratio();
double GetAMAValue(int index);
double GetEfratioValue(int index);
double GetLastAMA();
double GetLastEfratio();
```

ARBR

人气意愿指标

原理： 最近N个周期内最高价与开盘价的差的和除以开盘价与最低价的差的和，所得的比值放大100。 **用法：** （1）介于60至120，盘整；过高，回落；过低，反弹。 （2）BR意愿指标 **原理：** 最近N个周期内，若某日的最高价高于前一天的收盘价，将该日最高价与前收的差累加到强势和 中，若某日的最低价低于前收，则将前收与该日最低价的差累加到弱势和中。最后用强势和除以弱 势和，所得比值放大100。 **用法:** 介于70至150，盘整；高于400，回调；低于50，反弹。 **综合用法:** （1）AR和BR同时急速上升，意味价格峰位已近，应注意及时获利了结。 （2）BR比AR低，且指标处于低于100以下时，可考虑逢低买进。 （3）BR从高峰回跌，跌幅达120时，若AR无警戒讯号出现，应逢低买进。 （4）BR急速上升，AR盘整小回时，应逢高卖出，及时了结。

公共函数

```
//构造函数
ARBR(List<BarData> bars, int n = 26, bool isShowInMain = false, string tag =
"");
//取值函数
List<double> GetARValues();
List<double> GetBRValues();
double GetARValue(int index);
double GetBRValue(int index);
double GetLastAR();
double GetLastBR();
```

ATR

平均真实区域

原理： 无 **用法：** 无

公共函数

```
//构造函数
ATR(List<BarData> barDatas, int length = 14, bool isShowInMain = false, string
tag = "1");
//取值函数
List<double> GetTRValues();
List<double> GetATRValues();
double GetATRValue(int index);
double GetTRValue(int index);
double GetLastTR();
double GetLastATR();
```

ATRGrid1

波动网格1

原理： 无 **用法：** 无

公共函数

```
//构造函数
ATRGrid1(List<BarData> bars, int length = 20, double num = 1.5, int theme = 1,
bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetUpValues();
List<double> GetDownValues();
double GetUpValue(int index);
double GetDownValue(int index);
double GetLastUp();
double GetLastDown();
```

ATRGrid2

波动网格2

原理： 无 **用法：** 无

公共函数

```
//构造函数
ATRGrid2(List<BarData> bars, int length = 20, double num = 1.5, int theme = 1,
bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetUpValues();
List<double> GetDownValues();
double GetUpValue(int index);
double GetDownValue(int index);
double GetLastUp();
double double GetLastDown();
```

B3612

三减六日乖离

原理： B36 收盘价的3日均线与6日均线的差 B612 收盘价的6日均线与12日均线的差 注：日为周期中的一个，也可以用其他周期 **用法：** 乖离值围绕多空平衡点零上下波动，正数达到某个程度无法再往上升时，是卖出时机； 反之，是买进时机。多头走势中，行情回档多半在三减六日乖离达到零附近获得支撑， 即使跌破，也很快能够拉回。

公共函数

```
//构造函数
B3612(List<BarData> bars, EnumBarStruct objBarStruct = EnumBarStruct.Close, bool
isShowInMain = false, string tag = "");
//取值函数
List<double> GetB36Values();
List<double> GetB612Values();
double GetB36Value(int index);
double GetB612Value(int index);
double GetLastB36();
double GetLastB612();
```

BBI

牛熊指数

原理： 无 **用法：** 无

公共函数

```
//构造函数
BBI(List<BarData> barDatas, Color lineColor, int num1 = 3, int num2 = 6, int
num3 = 12, int num4 = 24, bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

BIAS

乖离率

原理： 无 **用法：** 无

公共函数

```
//构造函数
BIAS(List<BarData> barDatas, int num = 10, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

BOLL

布林通道

原理 无 **用法** 无

迭代属性

EnumBollLine

属性	类型	注释
Mid	*byte*	中轨
Up	*byte*	上轨
Down	byte	下轨

公共函数

```
//构造函数
BOLL(List<BarData> bars, int length = 26, double offset = 2, int theme = 1 , bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetValues(EnumBollLine bl);
double GetValue(EnumBollLine bl, DateTime dateTime);
double GetUpValue(int index);
double GetMidValue(int index);
double GetDownValue(int index);
double GetLast(EnumBollLine bl);
```

BSM

BSM

原理： 无 **用法：** 无

公共函数

```
//构造函数
BSM(List<BarData> barDatas, int length = 26, double step = 0.04, double limit = 0.2, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
int GetPosition(int index); //获取持仓（0-多,1-空）
```

CCI

商品通道指标

原理： 无 **用法：** 无

公共函数

```
//构造函数
CCI(List<BarData> bars, int length = 14, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

CDP

CDP逆势操作

原理 CDP 为最高价、最低价、收盘价的均值，称中价；中价与前一天的振幅的和、差分别记为AH(最高值)、AL(最低值)；两倍中价与最低价的差称NH(近高值)，与最高价的差称NL(近低值)。 **用法** 1.一种极短线的操作方法。在波动并不很大的情况下，即开市价处在近高值与近低值之间，通 常交易者可以在近低值的价们买进，而在近高期的价位卖出；或在近高值的价位卖出，近低值的价位买进。 2.在波动较大的情况下，即开市价开在最高值或最低值附近时，意味着跳空开高或跳空开低， 是一个大行情的发动开始，因此交易者可在最高值的价位去追买，最低值的价位去追卖。通常一个 跳空，意味着一个强烈的涨跌，应有相当的利润。

公共函数

```
//构造函数
CDP(List<BarData> bars, int n=3, bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetAHValues();
List<double> GetALValues();
List<double> GetNHValue();
List<double> GetNLValues();
double GetAHValue(int index);
double GetALValue(int index);
double GetNHValue(int index);
double GetNLValue(int index);
double GetLastAH();
double GetLastAL();
double GetLastNH();
double GetLastNL();
```

CR

CR能量指标

原理 在N日内，若某日最高价高于前一日中价(最高、最低价的均值)，将二者的差累加到强势和中； 若某日最低价低于前中价，将前中价与最低价的差累加到弱势和中。强势和除以弱势和，再乘100， 即得CR。同时绘制CR的M1周期、M2周期、M3周期均线。 **用法** 该指标用于判断买卖时机。能够测量人气的热度和价格动量的潜能； 显示压力带和支撑带，以辅助BRAR的不足。实际用来进行价格预测时：（1）a、b两线所夹的区域称为 副地震带，当CR由下往上欲穿越副地震带时，价格相对将遭次 级压力干扰；当CR欲由上往下贯穿副地震带时，价格相对将遭遇次级支撑干扰。（2）c、d两线所夹成的区域称为 主地震带，当CR由下往上欲穿越主地震带时，价格相对将遭 遇强大压力干扰；当CR由上往下欲贯穿主地震带时，价格相对将遭遇强大支撑干扰。（3）CR相对价格也会产生背离现象。特别是在高价区。（4）CR跌至a、b、c、d四条线的下方，再度由低点向上爬升160 % 时，为短线卖出时机。 例如

从CR100升到160。（5）CR下跌至40以下时，价格形成底部的机会相当高。（6）CR高于300 - 400之间时，价格很容易向下反转。

公共函数

```
//构造函数
CR(List<BarData> bars, int n = 26, int m1 = 5, int m2 = 10, int m3 = 20, bool
isShowInMain = false, string tag = "");
//取值函数
List<double> GetCRValues();
List<double> GetCRMA1Values();
List<double> GetCRMA2Values();
List<double> GetCRMA3Values();
double GetCRValue(int index);
double GetCRMA1Value(int index);
double GetCRMA2Value(int index);
double GetCRMA3Value(int index);
double GetLastCR();
double GetLastCRMA1();
double GetLastCRMA2();
double GetLastCRMA3();
```

Channel

通道

原理 无 **用法** 无

公共函数

```
//构造函数
Channel(List<BarData> barDatas, Color lineColor, int minSpan = 5, int mpe = 0,
bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetUpValues();
List<double> GetDownValues();
double GetUpValue(int index);
double GetDownValue(int index);
double GetLastUp();
double GetLastDown();
```

DBCD

异同离差乖离率

原理 先计算乖离率BIAS，然后计算不同日的乖离率之间的离差，最后对离差进行指数移动平滑处理。**用法** 与乖离率相同。(优点是能够保持指标紧密同步，而且线条光滑，信号明确，能够有效的过滤掉伪信号。)

公共函数

```
//构造函数
DBCD(List<BarData> bars, EnumBarStruct objBarStruct = EnumBarStruct.Close, int n
= 5, int m = 16,int t=76, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetDBCDValues();
List<double> GetMMValues();
double GetDBCDValue(int index);
double GetMMValue(int index);
double GetLastDBCD();
double GetLastMM();
```

DDI

方向标准离差指数

原理 1、TR =（最高价 - 昨日最高价）的绝对值与（最低价 - 昨日最低价）的绝对值两者之间较大者。 2、如果（最高价 + 最低价）<=（昨日最高价 + 昨日最低价），DMZ = 0， 如果（最高价 + 最低价）>（昨日最高价 + 昨日最低价），DMZ =（最高价 - 昨日最低价）的绝对值与（最低价 - 昨日最低价）的绝对值中较大值。 3、如果（最高价 + 最低价）>=（昨日最高价 + 昨日最低价），DMF = 0， 如果（最高价 + 最低价）<（昨日最高价 + 昨日最低价），DMZ =（最高价 - 昨日最低价）的绝对值与（最低价 - 昨日最低价）的绝对值中较大值。 4、DIZ = N个周期DMZ的和 /（N个周期DMZ的和 + N个周期DMF的和） 5、DIF = N个周期DMF的和 /（N个周期DMF的和 + N个周期DMZ的和） 6、DDI = DIZ - DIF 7、ADDI = DDI在一定周期内的加权平均 8、AD = ADDI在一定周期内的简单移动平均 **用法** （1）分析DDI柱状线，由红变绿(正变负)，卖出信号；由绿变红，买入信号。 （2）ADDI与AD的交叉情况以及背离情况。

公共函数

```
//构造函数
DDI(List<BarData> bars, int n = 13, int n1 = 30,int m=10,int m1=5, bool
isShowInMain = false, string tag = "");
//取值函数
List<double> GetDDIValues();
List<double> GetADDIValues();
List<double> GetADValues();
double GetDDIValue(int index);
double GetADDIValue(int index);
double GetADValue(int index);
double GetLastDDI();
double GetLastADDI();
double GetLastAD();
```

DKX

多空线

原理 多空线（DKX）是一个统计性指标。它是将主动买、主动卖的成交按时间区间分别统计而形成的一个曲线。 多空线有两条线，以交叉方式提示买入卖出。 **用法** （1）当DKX指标在超卖区上穿其均线时，为买入信号；当DKX指标在超买区下穿其均线时，为卖出信号 （2）在DKX指标之上，为多头市场，采取做多策略；股价在DKX指标之下，为空头市场，采取做空策略

公共函数

```
//构造函数
DKX(List<BarData> bars, int m = 10,bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetBValues();
List<double> GetDValues();
double GetBValue(int index);
double GetDValue(int index);
double GetLastB();
double GetLastD();
```

DMA

平行线差

原理 无 **用法** 无

公共函数

```
//构造函数
DMA(List<BarData> barDatas, int num1 = 10, int num2 = 50, int numMADMA = 6, bool
isShowInMain = false, string tag = "1");
//取值函数
List<double> GetDMAValues();
List<double> GetMADMAValues();
double GetDMAValue(int index);
double GetMADMAValue(int index);
double GetLastDMA();
double GetLastMADMA();
```

DMAB

日均线回归

原理 无 **用法** 无

公共函数

```
//构造函数
DMAB(List<BarData> barDatas, int length = 3, double validRange = 0.2, int
isFullDay = 0, int theme = 1, bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetLVValues();
List<double> GetSVValues();
List<double> GetDVValues();
double GetLVValue(int index);
double GetSVValue(int index);
double GetDVValue(int index);
double GetLastLV();
double GetLastSV();
double GetLastDV();
```

DMI

趋向指标

原理 无 **用法** 无

公共函数

```
//构造函数
DMI(List<BarData> barDatas, int length = 12, bool isShowInMain = false, string
tag = "1");
//取值函数
List<double> GetPDIValues();
List<double> GetMDIValues();
List<double> GetADXValues();
List<double> GetADXRValues();
double GetPDIValue(int index);
double GetMDIValue(int index);
double GetADXValue(int index);
double GetADXRValue(int index);
double GetLastPDI();
double GetLastMDI();
double GetLastADX();
double GetLastADXR();
```

EMA

指数移动平均

原理 无 **用法** 无

公共函数

```
//构造函数
EMA(List<BarData> barDatas, Color lineColor, EnumBarStruct objBarStruct =
EnumBarStruct.Close, int length = 20, bool isShowInMain = true, string tag =
"1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

Envalops

轨道线

原理

收盘价在N1个周期内的简单移动平均向上浮动N2%得UPPER线，向下浮动N2%得LOWER线。

参数：N1设定计算移动平均的天数，一般为14天

用法

UPPER线为压力线， LOWER线起支撑作用。当价格上试UPPER线时应考虑卖出；

反之，当价格下穿LOWER线或得到支撑时应考虑买入。

公共函数

```
//构造函数
Envalops(List<BarData> bars,EnumBarStruct objBarStruct = EnumBarStruct.Close,int
n1 = 14, int n2 = 6, bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetUValues();
List<double> GetLValues();
double GetUValue(int index);
double GetLValue(int index);
double GetLastU();
double GetLastL();
```

HCL

均线通道

原理

K线图叠加三条均线：最高价均线、最低价均线、收盘价均线;

用法

无

公共函数

```
//构造函数
HCL(List<BarData> bars,int n = 10,bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetHValues();
List<double> GetLValues();
List<double> GetCValues();
double GetHValue(int index);
double GetCValue(int index);
double GetLValue(int index);
double GetLastH();
double GetLastC();
double GetLastL();
```

HISVAR

历史潜在最大损失值

原理

无

用法

无

公共函数

```
//构造函数
HISVAR(List<BarData> barDatas, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetHISVARValues();
double GetHISVARValue(int index);
double GetLastHISVAR();
```

HighLow

最高最低价

原理

无

用法

无

公共函数

```
//构造函数
HighLow(List<BarData> barDatas, int length = 20, int theme = 1, bool
isShowInMain = true, string tag = "1");
//取值函数
List<double> GetHighestValues();
List<double> GetLowestValues();
double GetHighestValue(DateTime dateTime)
double GetHighestValue(int index);
double GetLowestValue(DateTime dateTime)
double GetLowestValue(int index);
double GetLastHighest();
double GetLastLowest();
```

IMPVAR

近期潜在最大损失值

原理

无

用法

无

公共函数

```
//构造函数
IMPVAR(List<BarData> barDatas, int length = 20, bool isShowInMain = false,
string tag = "1");
//取值函数
List<double> GetIMPVARValues();
double GetIMPVARValue(int index);
double GetLastIMPVAR();
```

KDJ

随机指标

原理

无

用法

无

公共函数

```
//构造函数
KDJ(List<BarData> bars, int length = 9, int ma1 = 3, int ma2 = 3, bool
isShowInMain = false, string tag = "1");
//取值函数
List<double> GetKValues();
List<double> GetDValues();
List<double> GetJValues();
double GetKValue(int index);
double GetDValue(int index);
double GetJValue(int index);
double GetLastK();
double GetLastD();
double GetLastJ();
```

Kelther

肯特纳通道

原理

无

用法

无

迭代属性

EnumKeltherLine

属性	类型	注释
Mid	*byte*	中轨
Up	*byte*	上轨
Down	byte	下轨

公共函数

```
//构造函数
Kelther(List<BarData> bars, int length = 20, double offset = 1.5, int theme = 1,
bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetValues(EnumKeltherLine kl);
double GetUpValue(int index);
double GetMidValue(int index);
double GetDownValue(int index);
double GetLast(EnumKeltherLine kl);
```

LLT

低延迟趋势线

原理

无

用法

无

公共函数

```
//构造函数
LLT(List<BarData> barDatas, Color lineColor, EnumBarStruct objBarStruct =
EnumBarStruct.Close, int length = 20, bool isShowInMain = true, string tag =
"1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

MACD

指数平滑异同平均线

原理

无

用法

无

迭代属性

EnumValueType

属性	类型	注释
MACDValue	*byte*	MACD
AvgMACD	*byte*	Avg MACD
MACDDiff	byte	MACD Diff

公共函数

```
//构造函数
MACD(List<BarData> barDatas, int fastLength = 12, int slowLength = 26, int
macdLength = 9, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetValues(EnumValueType valueType);
double GetMACDValue(int index);
double GetAvgValue(int index);
double GetDiffValue(int index);
double GetLast(EnumValueType valueType);
```

MTM

动力指标

原理

无

用法

无

公共函数

```
//构造函数
MTM(List<BarData> barDatas, int numMTM = 12, int numMTMMA = 6, bool isShowInMain
= false, string tag = "1");
//取值函数
List<double> GetMTMValues();
List<double> GetMTMMAValues();
double GetMTMValue(int index);
double GetMTMMAValue(int index);
double GetMTMLast();
double GetMTMMALast();
```

Mike

麦克支撑压力

原理 MIKE指标是一种路径指标，依据Typical Price为计算基准，求其Weak、Medium、 Strong三条带状支撑与压力，并且叠加到主图上。 **用法** 1.Weak - s、Medium - s、Strong - s三条线代表初级、中级及强力支撑。 2.Weak - r、Medium - r、Strong - r三条线代表初级、中级及强力压力。

公共函数

```
//构造函数
Mike(List<BarData> bars, int n = 12, bool isShowInMain = true, string tag =
"1");
//取值函数
List<double> GetWRValues();
List<double> GetMRValues();
List<double> GetSRValues();
List<double> GetWSValues();
List<double> GetMSValues();
List<double> GetSSValues();
double GetWRValue(int index);
double GetMRValue(int index);
double GetSRValue(int index);
double GetWSValue(int index);
double GetMSValue(int index);
double GetSSValue(int index);
double GetWRLast();
double GetMRLast();
double GetSRLast();
double GetWSLast();
double GetMSLast();
double GetSSLast();
```

OBV

能量潮

原理

无

用法

无

公共函数

```
//构造函数
OBV(List<BarData> barDatas, int length = 20, bool isShowInMain = false, string
tag = "1");
//取值函数
List<double> GetOBVValues();
List<double> GetMAOBVValues();
double GeOBVValue(int index);
double GetMAOBVValue(int index);
double GetLastOBV();
double GetLastMAOBV();
```

OpenInst

持仓量

原理

无

用法

无

公共函数

```
//构造函数
OpenInst(List<BarData> barDatas, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

PUBU

瀑布线

原理 瀑布线是由均线系统优化得来的趋势性指标，因其在运行的过程中， 形态与瀑布极其相似，故得名为瀑布线。 **用法** 1.瀑布线在低位粘合，短线瀑布线向上穿越长线瀑布线并向上发散，买入。 2.瀑布线在高位粘合，短线瀑布线向下穿越长线瀑布线并向下发散，卖出。 3.当长中短期瀑布线依次由下向上排列形成多头排列时，可持多单。 4.当长中短期瀑布线依次由上向下排列形成空头排列时，可持空单。

公共函数

```
//构造函数
PUBU(List<BarData> bars, EnumBarStruct objBarStruct = EnumBarStruct.Close,
                     int m1 = 4, int m2 = 6, int m3 = 9,
                     int m4 = 13, int m5 = 18, int m6 = 24,
                     bool isShowInMain = true, string tag = "");
//取值函数
List<double> GetValues(int lineidx);//用于获得第i条线的所有时间序列值
double GetValue(int lineidx,int index);//用于获得第i条线 索引为index的值
double GetLast(int i);//用于获得第i条线的最后值
```

RC

变化率指标

原理 当天收盘价与N天前收盘价的比值再进行移动平均。 **用法** 如果价格始终在上升，则变化率指数RC始终能保持在1线以上，如果变化率指数RC继续向上发展 ，则说明价格上升的速度在加快。反之，如果价格始终下降，则RC在1以下，如果RC继续向下发展， 则说明价格下降的速度在加快。

公共函数

```
//构造函数
RC(List<BarData> bars, EnumBarStruct objBarStruct = EnumBarStruct.Close, int n =
50, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetARCValues();
double GetARCValue(int index);
double GetLast();
```

RJX

日间均线

原理 无 **用法** 无

公共函数

```
//构造函数
RJX(List<BarData> barDatas, Color dtLineColor, Color fdLineColor, int length =
3,
            bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetFDValues();
List<double> GetDTValues();
double GetFDValue(int index);
double GetDTValue(int index);
double GetLastFD();
double GetLastDT();
```

RSI

相对强弱指数

原理 无 **用法** 无

公共函数

```
//构造函数
RSI(List<BarData> bars, int length = 14, bool isShowInMain = false, string tag =
"1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
```

SAR

抛物线转向

原理 无 **用法** 无

公共函数

```
//构造函数
SAR(List<BarData> barDatas, double step = 0.04, double limit = 0.2, int theme =
1, bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
int GetPosition(int index);
```

SMA

简单移动平均

原理 无 **用法** 无

公共函数

```
//构造函数
SMA(List<BarData> barDatas, Color lineColor, EnumBarStruct objBarStruct =
EnumBarStruct.Close, int length = 10, bool isShowInMain = true, string tag =
"1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
int GetPosition(int index);
```

VOL

成交量

原理 无 **用法** 无

公共函数

```
//构造函数
VOL(List<BarData> barDatas, int length = 20, bool isShowInMain = false, string
tag = "1");
//取值函数
List<double> GetVOLValues();
List<double> GetMAVOLValues();
List<double> GetOPIDValues();
double GetVOLValue(int index);
double GetMAVOLValue(int index);
double GetOPIDValue(int index);
double GetLastVOL();
double GetLastMAVOL();
double GetLastOPID();
```

VSD

波动标准差

原理 无 **用法** 无

公共函数

```
//构造函数
VSD(List<BarData> barDatas, int length = 20,int showPec=0, bool isShowInMain =
false, string tag = "1");
//取值函数
List<double> GetPecValues();
List<double> GetAvgValues();
List<double> GetStdValues();
double GetPecValue(int index);
double GetAvgValue(int index);
double GetStdValue(int index);
double GetLastPec();
double GetLastAvg();
double GetLastStd();
```

Volatility

波动率

原理 无 **用法** 无

公共函数

```
//构造函数
Volatility(List<BarData> barDatas, int length1 = 5, int length2 = 10, int
length3 = 20, int showLength = 1, bool isShowInMain = false, string tag = "1");
//取值函数
List<double> GetValues1();
List<double> GetValues2();
List<double> GetValues3();
double GetValue1(int index);
double GetValue2(int index);
double GetValue3(int index);
double GetLast1();
double GetLast2();
double GetLast3();
```

VolatilityChannel

波动率通道

原理 无 **用法** 无

公共函数

```
//构造函数
VolatilityChannel(List<BarData> barDatas, int length = 20, int predictedLength =
5,int theme = 1, bool isShowInMain = true, string tag = "1");
//取值函数
List<double> GetUpValues();
List<double> GetDownValues();
double GetUpValue(int index);
double GetDownValue(int index);
double GetLastUp();
double GetLastDown();
```

WMA

权重移动平均

原理 无 **用法** 无

公共函数

```
//构造函数
WMA(List<BarData> barDatas, Color lineColor, EnumBarStruct objBarStruct =
EnumBarStruct.Close, int length = 20, bool isShowInMain = true, string tag =
"1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
int GetPosition(int index);
```

Williams

威廉指标

原理 无 **用法** 无

公共函数

```
//构造函数
Williams(List<BarData> barDatas, int num = 7, bool isShowInMain = false, string
tag = "1");
//取值函数
List<double> GetValues();
double GetValue(int index);
double GetLast();
int GetPosition(int index);
```

C#教程

C# 是一个简单的、现代的、通用的、面向对象的编程语言，它是由微软（Microsoft）开发的。

本教程将告诉您基础的 C# 编程，同时将向您讲解 C# 编程语言相关的各种先进理念。

基础语法

C# 是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种相互交互的对象组成。相同种类的对象通常具有相同的类型，或者说，是在相同的 class 中。

例如，以 Rectangle（矩形）对象为例。它具有 length 和 width 属性。根据设计，它可能需要接受这些属性值、计算面积和显示细节。

让我们来看看一个 Rectangle（矩形）类的实现，并借此讨论 C# 的基本语法：

```
using System;
namespace RectangleApplication
{
    class Rectangle
    {
        // 成员变量
        double length;
        double width;
        public void Acceptdetails()
        {
```

```
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

数据类型

类型	描述	范围	默认值
bool	布尔值	True 或 False	False
byte	8 位无符号整数	0 到 255	0
char	16 位 Unicode 字符	U +0000 到 U +ffff	'\0'
decimal	128 位精确的十进制值，28-29 有效位数	(-7.9 x 1028 到 7.9 x 1028) / 100 到 28	0.0M
double	64 位双精度浮点型	(+/-)5.0 x 10-324 到 (+/-)1.7 x 10308	0.0D
float	32 位单精度浮点型	-3.4 x 1038 到 + 3.4 x 1038	0.0F
int	32 位有符号整数类型	-2,147,483,648 到 2,147,483,647	0
long	64 位有符号整数类型	-923,372,036,854,775,808 到 9,223,372,036,854,775,807	0L
sbyte	8 位有符号整数类型	-128 到 127	0
short	16 位有符号整数类型	-32,768 到 32,767	0
uint	32 位无符号整数类型	0 到 4,294,967,295	0
ulong	64 位无符号整数类型	0 到 18,446,744,073,709,551,615	0
ushort	16 位无符号整数类型	0 到 65,535	0

类型转换

序号	方法 & 描述
1	**ToBoolean**如果可能的话，把类型转换为布尔型。
2	**ToByte**把类型转换为字节类型。
3	**ToChar**如果可能的话，把类型转换为单个 Unicode 字符类型。
4	**ToDateTime**把类型（整数或字符串类型）转换为 日期-时间 结构。
5	**ToDecimal**把浮点型或整数类型转换为十进制类型。
6	**ToDouble**把类型转换为双精度浮点型。
7	**ToInt16**把类型转换为 16 位整数类型。
8	**ToInt32**把类型转换为 32 位整数类型。
9	**ToInt64**把类型转换为 64 位整数类型。
10	**ToSbyte**把类型转换为有符号字节类型。
11	**ToSingle**把类型转换为小浮点数类型。
12	**ToString**把类型转换为字符串类型。
13	**ToType**把类型转换为指定类型。
14	**ToUInt16**把类型转换为 16 位无符号整数类型。
15	**ToUInt32**把类型转换为 32 位无符号整数类型。
16	**ToUInt64**把类型转换为 64 位无符号整数类型。

变量

一个变量只不过是一个供程序操作的存储区的名字。在 C# 中，每个变量都有一个特定的类型，类型决定了变量的内存大小和布局。范围内的值可以存储在内存中，可以对变量进行一系列操作。

我们已经讨论了各种数据类型。C# 中提供的基本的值类型大致可以分为以下几类：

类型	举例
整数类型	sbyte、byte、short、ushort、int、uint、long、ulong 和 char
浮点型	float 和 double
十进制类型	decimal
布尔类型	true 或 false 值，指定的值
空类型	可为空值的数据类型

C# 允许定义其他值类型的变量，比如 **enum**，也允许定义引用类型变量，比如 **class**。

运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C# 有丰富的内置运算符，分类如下：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其他运算符

本教程将逐一讲解算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

算术运算符

下表显示了 C# 支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

请看下面的实例，了解 C# 中所有可用的算术运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int b = 10;
            int c;

            c = a + b;
            Console.WriteLine("Line 1 - c 的值是 {0}", c);
            c = a - b;
            Console.WriteLine("Line 2 - c 的值是 {0}", c);
            c = a * b;
            Console.WriteLine("Line 3 - c 的值是 {0}", c);
            c = a / b;
            Console.WriteLine("Line 4 - c 的值是 {0}", c);
            c = a % b;
            Console.WriteLine("Line 5 - c 的值是 {0}", c);
```

```
            // ++a 先进行自增运算再赋值
            c = ++a;
            Console.WriteLine("Line 6 - c 的值是 {0}", c);

            // 此时 a 的值为 22
            // --a 先进行自减运算再赋值
            c = --a;
            Console.WriteLine("Line 7 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 22
Line 7 - c 的值是 21
```

- **c = a++**: 先将 a 赋值给 c，再对 a 进行自增运算。
- **c = ++a**: 先将 a 进行自增运算，再将 a 赋值给 c 。
- **c = a--**: 先将 a 赋值给 c，再对 a 进行自减运算。
- **c = --a**: 先将 a 进行自减运算，再将 a 赋值给 c 。

逻辑运算符

下表显示了 C# 支持的所有逻辑运算符。假设变量 **A** 为布尔值 true，变量 **B** 为布尔值 false，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

实例

请看下面的实例，了解 C# 中所有可用的逻辑运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            bool a = true;
            bool b = true;

            if (a && b)
            {
               Console.WriteLine("Line 1 - 条件为真");
            }
            if (a || b)
            {
                Console.WriteLine("Line 2 - 条件为真");
            }
            /* 改变 a 和 b 的值 */
            a = false;
            b = true;
            if (a && b)
            {
                Console.WriteLine("Line 3 - 条件为真");
            }
            else
            {
                Console.WriteLine("Line 3 - 条件不为真");
            }
            if (!(a && b))
            {
                Console.WriteLine("Line 4 - 条件为真");
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

赋值运算符

下表列出了 C# 支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
|=	按位或且赋值运算符	C |= 2 等同于 C = C | 2

实例

请看下面的实例，了解 C# 中所有可用的赋值运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 21;
            int c;

            c = a;
            Console.WriteLine("Line 1 - =  c 的值 = {0}", c);

            c += a;
            Console.WriteLine("Line 2 - += c 的值 = {0}", c);

            c -= a;
            Console.WriteLine("Line 3 - -=  c 的值 = {0}", c);

            c *= a;
            Console.WriteLine("Line 4 - *=  c 的值 = {0}", c);
```

```
            c /= a;
            Console.WriteLine("Line 5 - /=  c 的值 = {0}", c);

            c = 200;
            c %= a;
            Console.WriteLine("Line 6 - %=  c 的值 = {0}", c);

            c <<= 2;
            Console.WriteLine("Line 7 - <<=  c 的值 = {0}", c);

            c >>= 2;
            Console.WriteLine("Line 8 - >>=  c 的值 = {0}", c);

            c &= 2;
            Console.WriteLine("Line 9 - &=  c 的值 = {0}", c);

            c ^= 2;
            Console.WriteLine("Line 10 - ^=  c 的值 = {0}", c);

            c |= 2;
            Console.WriteLine("Line 11 - |=  c 的值 = {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =     c 的值 = 21
Line 2 - +=    c 的值 = 42
Line 3 - -=    c 的值 = 21
Line 4 - *=    c 的值 = 441
Line 5 - /=    c 的值 = 21
Line 6 - %=    c 的值 = 11
Line 7 - <<=    c 的值 = 44
Line 8 - >>=    c 的值 = 11
Line 9 - &=    c 的值 = 2
Line 10 - ^=    c 的值 = 0
Line 11 - |=    c 的值 = 2
```

运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 * 2，在这里，x 被赋值为 13，而不是 20，因为运算符 * 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {
         int a = 20;
         int b = 10;
         int c = 15;
         int d = 5;
         int e;
         e = (a + b) * c / d;      // ( 30 * 15 ) / 5
         Console.WriteLine("(a + b) * c / d 的值是 {0}", e);

         e = ((a + b) * c) / d;   // (30 * 15 ) / 5
         Console.WriteLine("((a + b) * c) / d 的值是 {0}", e);

         e = (a + b) * (c / d);   // (30) * (15/5)
         Console.WriteLine("(a + b) * (c / d) 的值是 {0}", e);

         e = a + (b * c) / d;     //  20 + (150/5)
         Console.WriteLine("a + (b * c) / d 的值是 {0}", e);
```

```
            Console.ReadLine();
        }
    }
}
```

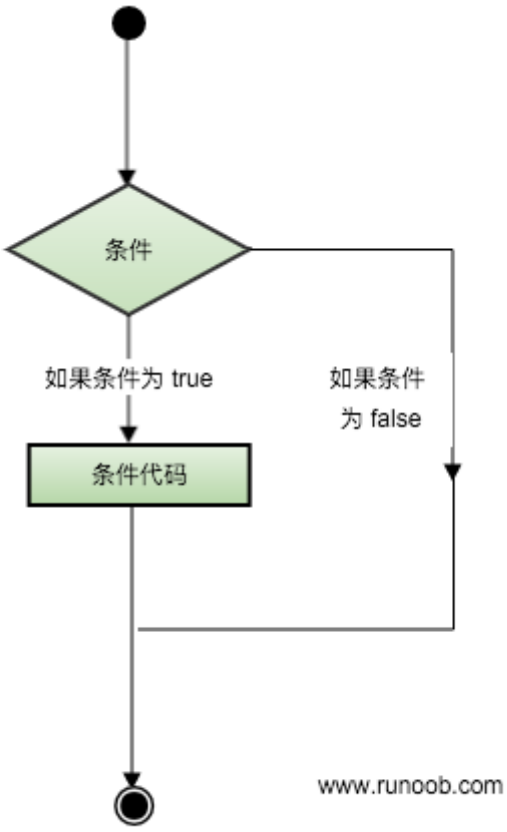
当上面的代码被编译和执行时，它会产生下列结果：

```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

下面是大多数编程语言中典型的判断结构的一般形式：



判断语句

C# 提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
if 语句	一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。
switch 语句	一个 **switch** 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 **switch** 语句内使用另一个 **switch** 语句。

? : 运算符

我们已经在前面的章节中讲解了 **条件运算符 ? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

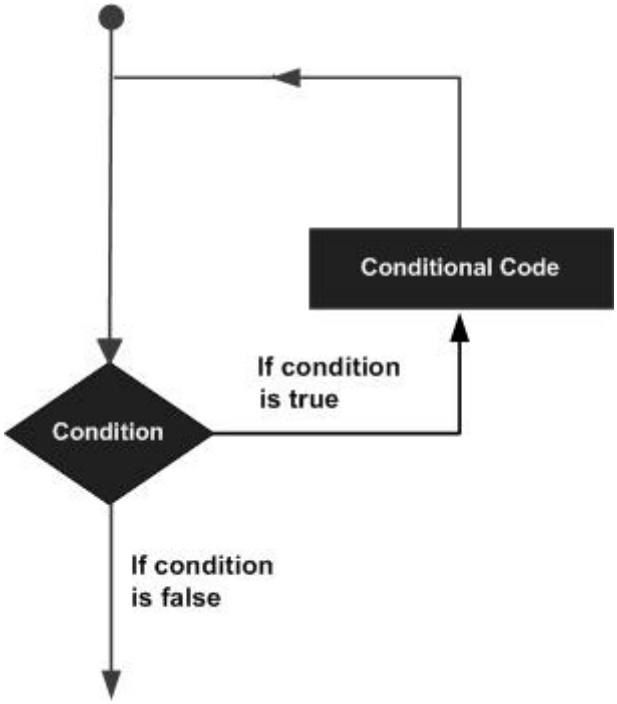
? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

循环

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



循环类型

C# 提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for/foreach 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C# 提供了下列的控制语句。点击链接查看每个语句的细节。

控制语句	描述
break 语句	终止 **loop** 或 **switch** 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
continue 语句	引起循环跳过主体的剩余部分，立即重新开始测试条件。

无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Hey! I am Trapped");
            }

        }
    }
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，程序员偏向于使用 for(;;) 结构来表示一个无限循环。

方法

一个方法是把一些相关的语句组织在一起，用来执行一个任务的语句块。每一个 C# 程序至少有一个带有 Main 方法的类。

要使用一个方法，您需要：

- 定义方法
- 调用方法

C# 中定义方法

当定义一个方法时，从根本上说是在声明它的结构的元素。在 C# 中，定义方法的语法如下：

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

下面是方法的各个元素：

- **Access Specifier**：访问修饰符，这个决定了变量或方法对于另一个类的可见性。
- **Return type**：返回类型，一个方法可以返回一个值。返回类型是方法返回的值的数据类型。如果方法不返回任何值，则返回类型为 **void**。
- **Method name**：方法名称，是一个唯一的标识符，且是大小写敏感的。它不能与类中声明的其他标识符相同。
- **Parameter list**：参数列表，使用圆括号括起来，该参数是用来传递和接收方法的数据。参数列表是指方法的参数类型、顺序和数量。参数是可选的，也就是说，一个方法可能不包含参数。
- **Method body**：方法主体，包含了完成任务所需的指令集。

实例

下面的代码片段显示一个函数 *FindMax*，它接受两个整数值，并返回两个中的较大值。它有 public 访问修饰符，所以它可以使用类的实例从类的外部进行访问。

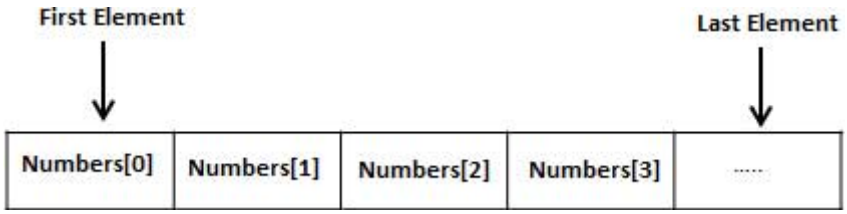
```
class NumberManipulator
{
   public int FindMax(int num1, int num2)
   {
      /* 局部变量声明 */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

数组

数组是一个存储相同类型元素的固定大小的顺序集合。数组是用来存储数据的集合，通常认为数组是一个同一类型变量的集合。

声明数组变量并不是声明 number0、number1、...、number99 一个个单独的变量，而是声明一个就像 numbers 这样的变量，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

所有的数组都是由连续的内存位置组成的。最低的地址对应第一个元素，最高的地址对应最后一个元素。



声明数组

在 C# 中声明一个数组，您可以使用下面的语法：

```
datatype[] arrayName;
```

其中，

- *datatype* 用于指定被存储在数组中的元素的类型。
- *[ ]* 指定数组的秩（维度）。秩指定数组的大小。
- *arrayName* 指定数组的名称。

例如：

```
double[] balance;
```

初始化数组

声明一个数组不会在内存中初始化数组。当初始化数组变量时，您可以赋值给数组。

数组是一个引用类型，所以您需要使用 **new** 关键字来创建数组的实例。

例如：

```
double[] balance = new double[10];
```

赋值给数组

您可以通过使用索引号赋值给一个单独的数组元素，比如：

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

您可以在声明数组的同时给数组赋值，比如：

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

您也可以创建并初始化一个数组，比如：

```
int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

在上述情况下，你也可以省略数组的大小，比如：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

您也可以赋值一个数组变量到另一个目标数组变量中。在这种情况下，目标和源会指向相同的内存位置：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

当您创建一个数组时，C# 编译器会根据数组类型隐式初始化每个数组元素为一个默认值。例如，int 数组的所有元素都会被初始化为 0。

访问数组元素

元素是通过带索引的数组名称来访问的。这是通过把元素的索引放置在数组名称后的方括号中来实现的。例如：

```
double salary = balance[9];
```

下面是一个实例，使用上面提到的三个概念，即声明、赋值、访问数组：

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */
         int i,j;


         /* 初始化数组 n 中的元素 */
         for ( i = 0; i < 10; i++ )
         {
            n[ i ] = i + 100;
         }

         /* 输出每个数组元素的值 */
         for (j = 0; j < 10; j++ )
         {
            Console.WriteLine("Element[{0}] = {1}", j, n[j]);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

字符串

在 C# 中，您可以使用字符数组来表示字符串，但是，更常见的做法是使用 **string** 关键字来声明一个字符串变量。string 关键字是 **System.String** 类的别名。

创建 String 对象

您可以使用以下方法之一来创建 string 对象：

- 通过给 String 变量指定一个字符串
- 通过使用 String 类构造函数
- 通过使用字符串串联运算符（ + ）
- 通过检索属性或调用一个返回字符串的方法
- 通过格式化方法来转换一个值或对象为它的字符串表示形式

下面的实例演示了这点：

```
using System;

namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //字符串，字符串连接
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //通过使用 string 构造函数
            char[] letters = { 'H', 'e', 'l', 'l','o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);

            //方法返回字符串
            string[] sarray = { "Hello", "From", "Tutorials", "Point" };
            string message = String.Join(" ", sarray);
            Console.WriteLine("Message: {0}", message);

            //用于转化值的格式化方法
            DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
            string chat = String.Format("Message sent at {0:t} on {0:D}",
```

```
            waiting);
            Console.WriteLine("Message: {0}", chat);
            Console.ReadKey() ;
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Full Name: RowanAtkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 17:58 on Wednesday, 10 October 2012
```

String 类的属性

String 类有以下两个属性：

序号	属性名称 & 描述
1	**Chars**在当前 *String* 对象中获取 *Char* 对象的指定位置。
2	**Length**在当前的 *String* 对象中获取字符数。

String 类的方法

String 类有许多方法用于 string 对象的操作。下面的表格提供了一些最常用的方法：

序号	方法名称 & 描述
1	**public static int Compare(string strA, string strB)** 比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。该方法区分大小写。
2	**public static int Compare(string strA, string strB, bool ignoreCase)** 比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。但是，如果布尔参数为真时，该方法不区分大小写。
3	**public static string Concat(string str0, string str1)** 连接两个 string 对象。
4	**public static string Concat(string str0, string str1, string str2)** 连接三个 string 对象。
5	**public static string Concat(string str0, string str1, string str2, string str3)** 连接四个 string 对象。
6	**public bool Contains(string value)** 返回一个表示指定 string 对象是否出现在字符串中的值。
7	**public static string Copy(string str)** 创建一个与指定字符串具有相同值的新的 String 对象。
8	**public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)** 从 string 对象的指定位置开始复制指定数量的字符到 Unicode 字符数组中的指定位置。
9	**public bool EndsWith(string value)** 判断 string 对象的结尾是否匹配指定的字符串。
10	**public bool Equals(string value)** 判断当前的 string 对象是否与指定的 string 对象具有相同的值。
11	**public static bool Equals(string a, string b)** 判断两个指定的 string 对象是否具有相同的值。
12	**public static string Format(string format, Object arg0)** 把指定字符串中一个或多个格式项替换为指定对象的字符串表示形式。
13	**public int IndexOf(char value)** 返回指定 Unicode 字符在当前字符串中第一次出现的索引，索引从 0 开始。
14	**public int IndexOf(string value)** 返回指定字符串在该实例中第一次出现的索引，索引从 0 开始。
15	**public int IndexOf(char value, int startIndex)** 返回指定 Unicode 字符从该字符串中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
16	**public int IndexOf(string value, int startIndex)** 返回指定字符串从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
17	**public int IndexOfAny(char[] anyOf)** 返回某一个指定的 Unicode 字符数组中任意字符在该实例中第一次出现的索引，索引从 0 开始。
18	**public int IndexOfAny(char[] anyOf, int startIndex)** 返回某一个指定的 Unicode 字符数组中任意字符从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。

序号	方法名称 & 描述
19	**public string Insert(int startIndex, string value)** 返回一个新的字符串，其中，指定的字符串被插入在当前 string 对象的指定索引位置。
20	**public static bool IsNullOrEmpty(string value)** 指示指定的字符串是否为 null 或者是否为一个空的字符串。
21	**public static string Join(string separator, string[] value)** 连接一个字符串数组中的所有元素，使用指定的分隔符分隔每个元素。
22	**public static string Join(string separator, string[] value, int startIndex, int count)** 连接接一个字符串数组中的指定位置开始的指定元素，使用指定的分隔符分隔每个元素。
23	**public int LastIndexOf(char value)** 返回指定 Unicode 字符在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。
24	**public int LastIndexOf(string value)** 返回指定字符串在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。
25	**public string Remove(int startIndex)** 移除当前实例中的所有字符，从指定位置开始，一直到最后一个位置为止，并返回字符串。
26	**public string Remove(int startIndex, int count)** 从当前字符串的指定位置开始移除指定数量的字符，并返回字符串。
27	**public string Replace(char oldChar, char newChar)** 把当前 string 对象中，所有指定的 Unicode 字符替换为另一个指定的 Unicode 字符，并返回新的字符串。
28	**public string Replace(string oldValue, string newValue)** 把当前 string 对象中，所有指定的字符串替换为另一个指定的字符串，并返回新的字符串。
29	**public string[] Split(params char[] separator)** 返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。
30	**public string[] Split(char[] separator, int count)** 返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。int 参数指定要返回的子字符串的最大数目。
31	**public bool StartsWith(string value)** 判断字符串实例的开头是否匹配指定的字符串。
32	**public char[] ToCharArray()**返回一个带有当前 string 对象中所有字符的 Unicode 字符数组。
33	**public char[] ToCharArray(int startIndex, int length)** 返回一个带有当前 string 对象中所有字符的 Unicode 字符数组，从指定的索引开始，直到指定的长度为止。
34	**public string ToLower()**把字符串转换为小写并返回。
35	**public string ToUpper()**把字符串转换为大写并返回。
36	**public string Trim()**移除当前 String 对象中的所有前导空白字符和后置空白字符。

上面的方法列表并不详尽，请访问 MSDN 库，查看完整的方法列表和 String 类构造函数。

实例

下面的实例演示了上面提到的一些方法：

比较字符串

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str1 = "This is test";
         string str2 = "This is text";

         if (String.Compare(str1, str2) == 0)
         {
            Console.WriteLine(str1 + " and " + str2 +  " are equal.");
         }
         else
         {
            Console.WriteLine(str1 + " and " + str2 + " are not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is test and This is text are not equal.
```

字符串包含字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "This is test";
         if (str.Contains("test"))
         {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The sequence 'test' was found.
```

获取子字符串：

```
using System;
namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "Last night I dreamt of San Pedro";
            Console.WriteLine(str);
            string substr = str.Substring(23);
            Console.WriteLine(substr);
            Console.ReadKey() ;
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Last night I dreamt of San Pedro
San Pedro
```

枚举

枚举是一组命名整型常量。枚举类型是使用 **enum** 关键字声明的。

C# 枚举是值类型。换句话说，枚举包含自己的值，且不能继承或传递继承。

声明 *enum* 变量

声明枚举的一般语法：

```
enum <enum_name>
{
    enumeration list
};
```

其中，

- *enum_name* 指定枚举的类型名称。
- *enumeration list* 是一个用逗号分隔的标识符列表。

枚举列表中的每个符号代表一个整数值，一个比它前面的符号大的整数值。默认情况下，第一个枚举符号的值是 0.例如：

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

实例

下面的实例演示了枚举变量的用法：

```
using System;
namespace EnumApplication
{
```

```
   class EnumProgram
   {
      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

      static void Main(string[] args)
      {
         int WeekdayStart = (int)Days.Mon;
         int WeekdayEnd = (int)Days.Fri;
         Console.WriteLine("Monday: {0}", WeekdayStart);
         Console.WriteLine("Friday: {0}", WeekdayEnd);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Monday: 1
Friday: 5
```

数学函数库

Math库包含的函数如下：

函数原型	功能	返回值	说明
int Abs(int x)	求整数x的绝对值	绝对值	
double Acos(double x)	计算arccos(x)的值	计算结果	-1≤x≤1
double Asin(double x)	计算arcsin(x)的值	计算结果	-1≤x≤1
double Atan(double x)	计算arctan(x)的值	计算结果	
double atan2(double y, double x);	计算arctan(y/x)的值	计算结果	
long BigMul(int x, int y)	计算x*y的值	计算结果	
int Ceiling(double x)	返回大于或等于所给数字表达式x的最小整数	最小整数	

函数原型	功能	返回值	说明
double Cos(double x)	计算cos(x)的值	计算结果	x的单位为弧度
double Cosh(double x)	计算x的双曲余弦cosh(x)的值	计算结果	
int DivRem(int x,int y,int z)	计算x与y的商，并将余数作为输出参数进行传递	x与y的商，z为余数	
double Exp(double x)	求ex的值	计算结果	
int Floor (double x)	返回小于或等于所给数字表达式x的最大整数	最大整数	
int IEEERemainder(int x, int y)	返回x/y的余数	计算结果	
double Log(double x)	计算ln(x)的值	计算结果	
double Log10(double x)	计算log10(x)的值	计算结果	
double Max(double x, double y)	返回x,y中的较大者	计算结果	
double Min(double x, double y)	返回x,y中的较小者	计算结果	
double Pow(double x,double y)	求xy的值	计算结果	
int Round(double x)	将x四舍五入到最接近的整数	计算结果	
double Round(double x,int y)	将x四舍五入到由y指定的小数位数	计算结果	
int Sign(double x)	返回表示x符号的值	数值x大于0，返回1；数值x等于0返回0；数值x小于0，返回-1	
double Sin(double x)	计算sin(x)的值	计算结果	x的单位为弧度
double Sinh(double x)	计算x的双曲正弦sinh(x)的值	计算结果	

函数原型	功能	返回值	说明
double Sqrt(double x)	求的值	计算结果	x≥0
double Tan(double x)	计算tan(x)的值	计算结果	x的单位为弧度
double Tanh(double x)	计算x的双曲正切tanh(x)的值	计算结果	

这些数学函数的基本使用方法在表1中均有说明，都是静态函数，调用的时候用算术类直接调用，例如：

double d = Math.Sin(123.0);

对于个别函数的计算结果要注意。

例如，对于数字12.9273，Ceiling（12.9273）将返回 13，Floor(12.9273) 将返回 12。