

以下资料均属网上搜集，由东方华尔街论坛整理

## 东方华尔街，您最好的金融 E 家

bbs.moneyeast.com

如何在交易开拓者中编写技术指标？

1、在面板中[TB 公式]组里点击[新建技术指标按钮]；



2、在新建技术指标对话框中输入相应的信息；

新建 - [技术指标]

简称:  
FirstIndicator

名称:  
我的第一个技术指标

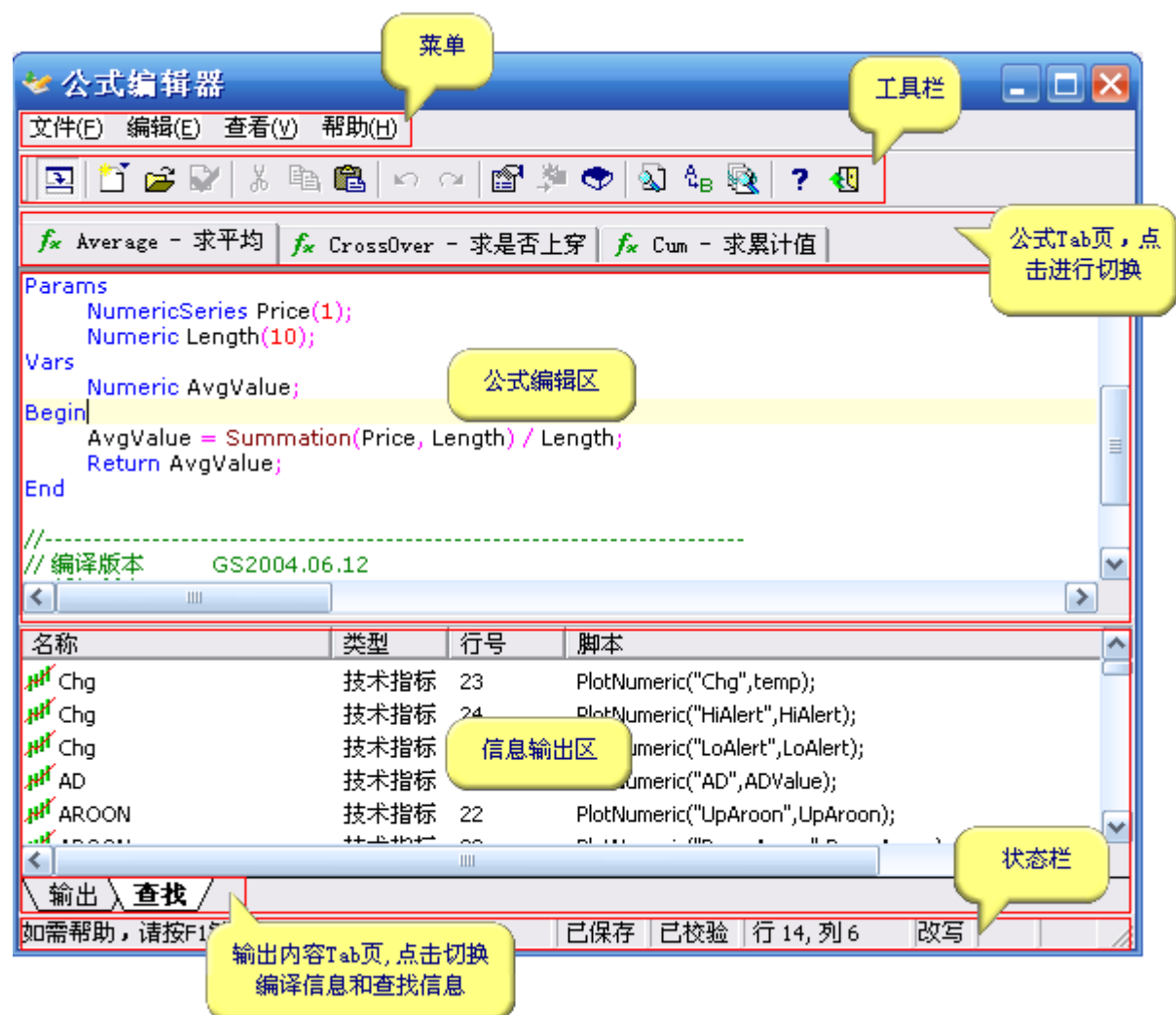
分类: 振荡类 模板: 空

注释:  
示范如何新建技术指标

确定 (O) 取消 (C) 帮助 (H)

输入公式的简称，只能是数字，字母和下划线的组合。该名称作为将以后调用技术指标的标识

3、点击[确定]按钮之后，将会进入公式编辑器界面；



4、以下以标准的MACD指标脚本为例，讲解公式的语法；

<b>Params</b> Numeric FastLength(12); Numeric SlowLength(26); Numeric MACDLength(9);	公式参数段，用Params宣告参数代码段的开始
<b>Vars</b> NumericSeries MACDValue; Numeric AvgMACD; Numeric MACDDiff;	公式变量段，用Params宣告变量代码段的开始
<b>Begin</b> MACDValue = XAverage(Close, FastLength) - XAverage(Close, SlowLength); AvgMACD = XAverage(MACDValue, MACDLength); MACDDiff = MACDValue - AvgMACD; PlotNumeric("MACD", MACDValue); PlotNumeric("MACDAvg", AvgMACD); If (MACDDiff >= 0) PlotNumeric("MACDDiff", MACDDiff, Red); Else PlotNumeric("MACDDiff", MACDDiff, Green); PlotNumeric("零线", 0); <b>End</b>	公式脚本段，用Begin和End宣告代码的起始和结束，一般技术指标输出一条线用PlotNumeric函数

## 5、关于参数；

### 参数声明

在使用参数之前，必须对参数进行声明，TradeBlazer公式使用关键字"Params"来进行参数宣告，并指定参数类型。可以选择，也可以不赋默认值。如果某个参数没有赋予默认值，则这个参数之前的其他参数的默认值都将被忽略。

参数定义的语法如下：

```
Params
    参数类型 参数名1(初值);
    参数类型 参数名2(初值);
    参数类型 参数名3(初值);
```

下面是一些参数定义的例子：

```
Params
    Bool          bTest(False);           //定义布尔型参数bTest，默认值为False;
    Numeric        Length(10);             //定义数值型参数Length，默认值为10;
    NumericSeries  Price(0);               //定义数值型序列参数Price，默认值为0;
    NumericRef     output(0);              //定义数值型引用参数output，默认值为0;
    String         strTmp("Hello");        //定义字符串参数strTmp,默认值为Hello;
```

参数名称的命名规范详细说明参见[命名规则](#)。

整个公式中只能出现一个Params宣告，并且要放到公式的开始部分，在变量定义之前。

## 6、关于变量；

## 变量声明

在使用变量之前，必须对变量进行声明，TradeBlazer公式使用关键字"Vars"来进行变量宣告，并指定变量类型。可以选择赋也可以不赋默认值。

变量定义的语法如下：

```
Vars
    变量类型 变量名1(初值);
    变量类型 变量名2(初值);
    变量类型 变量名3(初值);
```

下面是一些变量定义的例子：

```
Vars
    NumericSeries    MyVal1(0);           //定义数值型序列变量MyVal1，默认值为0；
    Numeric          MyVal2(0);           //定义数值型变量MyVal2，默认值为0；
    Bool              MyVal3(False);       //定义布尔型变量MyVal3，默认值为False；
    String            MyVal4("Test");      //定义字符串变量MyVal4，默认值为Test。
```

变量定义的个数没有限制，变量名称的命名规范详细说明参见[命名规则](#)。

整个公式中只能出现一个Vars宣告，并且要放到公式的开始部分，在参数定义之后，正文之前。

## 7、关于公式属性；

在公式编辑器中可通过点击工具栏的[属性设置]按钮打开公式属性对话框。

对于技术指标而言，主要是设置技术指标各个线条的线型，颜色，粗细等内容。

2007-7-20 17:06

step7.png (9.92 KB)



8、在编写公式脚本之后，点击公式编辑器工具栏的第五个按钮[校验保存公式]，编译成功之后，信息输出区会显示[成功保存当前公式信息！]字样。如果编译失败，也会在信息输出区显示出错的信息以及行号，根据这些提示信息，您可以检查代码，并找出脚本的编辑错误进行修改。

9、成功编译公式之后，您可以在超级图表的窗体中直接输入该公式的简称，即可通过[我的键盘]调用。或者通过工具栏插入该公式。

10、关于更详细的公式使用请参照帮助文件。

## TradeBlazer 公式入门教程(1)

**Step 1**、在开始写公式之前，我们先了解以下基本概念

### Bar 数据：

公式在进行计算时，都是建立在基本数据源(Bar 数据)之上，我们这里所谓的 Bar 数据，是指商品在不同周期下形成的序列数据，在单独的每个 Bar 上面包含开盘价、收盘价、最高价、最低价、成交量及时间。期货等品种还有持仓量等数据。所有的 Bar 按照不同周期组合，并按照时间从先到后进行排列，由此形成序列数据，整个序列称之为 Bar 数据。

### 公式如何执行：

TradeBlazer 公式在计算时按照 Bar 数据的 Bar 数目，从第一个 Bar 到最后一个 Bar，依次进行计算，如果公式中出现了调用 Bar 数据函数的，则取出当前 Bar 的相应值，进行运算。公式执行从上至下，Bar 从左到右执行。

**Step 2**、接下来，我们从 TradeBlazer 公式的 HelloWorld 开始

### TradeBlazer 公式的 HelloWorld!

从第一版 C 语言推出，“Hello World”这个经典程序，就成为全世界程序员挥之不去的情结。那么在 TB 中的 HELLO WORLD 是怎么写的呢？

首先，为了方便讨论，我们先设置一下环境：打开超级图表，选择一个当前没有行情的品种，比如 IF0705。再鼠标右键菜单中选择“商品设置”，只显示最后 5 个样本。

然后，新建立一个指标，取名为 HelloWorld,输入如下代码并保存：

[Copy to clipboard] [ - ]

#### CODE:

```
Begin
    FileAppend("c:\\Formula.log","hello world");
End
```

最后，把指标 HelloWorld 插入到 超级图表中。

图表上并不会有什么输出，但是 C 盘根目录下会产生一个 Formula.log 文件。该文件的内容为：

#### QUOTE:

```
hello world
hello world
hello world
hello world
hello world
```

如果你能执行到这一步，看见文件中的 5 行 hello world，那么恭喜你，你的第一个 TB 语言公式已经完成了！具体有多少行 hello world 取决于你在超级图表中设置的样本数目，刚才设置了 5 个样本，所以是 5 行 hello world 字符串。

FileAppend 函数是 TB 中的写文件函数，可以在指定文件中追加一行字符串。该函数的语法原型为：

```
Bool FileAppend(String strPath,String strText);
```

参数 `strPath`: 指定文件的路径, 请使用全路径表示, 并使用\\做路径分割符。  
参数: `strText` 输出的字符串内容。

这个函数非常重要! 它不仅仅是写文件这么简单, 因为 TB 中没有公式的单步执行调试工具, 所以公式的调试往往是通过把你要查看的变量值输出到文件来完成的。比如, 你要查看 `CLOSE` 的值, 那么

[Copy to clipboard] [ - ]

CODE:

FileAppend("c:\\Formula.log","Close = "+Text(Close));

其中 `Text` 函数可以把数值类型转换成字符串。  
我们可以进一步把 `HelloWorld` 中的内容改为:

[Copy to clipboard] [ - ]

CODE:

Begin  
FileAppend("c:\\Formula.log","Bar"+Text(CurrentBar)+"hello world");  
End

`CurrentBar` 函数返回的是当前 `BAR` 的索引值, 该值从 0 开始递增。如果图表中的样本数是 5, 那么这 5 根 `BAR` 的索引从左到右分别是: 0、1、2、3、4。保存公式之后, 文件 `Formula.log` 中的内容将是:

QUOTE:	
Bar0	hello world
Bar1	hello world
Bar2	hello world
Bar3	hello world
Bar4	hello world

这就清楚地显示了 `FileAppend` 函数分别在每个一个 `BAR` 上都执行了一遍, 一共执行了 5 遍。

你可能认为公式理所当然地应该执行 5 遍, 因为有 5 个 `BAR` 啊。并非如此! 不同的软件, 不同的语言公式架构是不同的。类分析家语言, 比如文华、飞狐等等, 都只执行一遍!

我们把一个公式看成是一个整体黑盒, 类分析家语言是把 5 根 `BAR` 作为一个整体输入, 公式里的每个语句都只执行一遍, 整个公式也只执行一遍, 然后便输出了。所以, 在类分析家语言中是无法实现 `IF` 语句和 `WHILE` 循环语句的, 所有的复合语句都无法实现。要做就只能在底层用 C 语言遍成函数做特殊处理。如果你是程序员, 你大概早就会很奇怪为什么几乎所有语言都有 `IF` 语句和 `WHILE`、`FOR` 语句, 而分析家、文华、飞狐中却只有 `IF` 函数呢? 原因就就在这里了。

而 TB 不是这样。TB 是把第一根 `BAR` 作为输入传给公式, 得到一个输出。然后再传入第二根, 第三根..... 有多少根 `BAR`, 公式就会被执行多少次。用这样一种机制, 就可以实现公式和算法的精确控制。

很多用惯了类分析家语言平台的投资者总会觉得 TB 语言很难学。其实并不是语法难学, 而在于处理机制



不同导致的编程思维模式的不同。如果你要学习 TB 语言，那么了解它的这个处理框架，从原有的编程思维中跳出来就显得非常重要。

**Step 3**、建立一个简单的指标：成交量

对于交易开拓者界面不熟悉的朋友可以参看以下帖子：  
[如何在交易开拓者中编写技术指标？](#)

新建指标简称: MyVol

[Copy to clipboard] [ - ]

**CODE:**  
Begin  
    PlotNumeric("Vol",Vol);  
End

**Begin** 和 **End** 宣告公式正文的开始和结束，公式语句应该放到 **Begin** 和 **End** 之间。  
并且总是以";"作为语句结束的标志。  
**PlotNumeric** 表示输出一个数值型组成的数组;公式中""内所引用的是字符串的常量，内容文字即在图表中所输出的技术指标的名称

**关于 PlotNumeric 的使用**

函数原形：  
    Numeric PlotNumeric(String Name,Numeric Number,Integer Color=-1,Integer BarsBack=0)  
参数：  
    Name 输出值的名称，不区分大小写；  
    Number 输出的数值；  
    Color 输出值的显示颜色，默认表示使用属性设置框中的颜色；  
    BarsBack 从当前 Bar 向前回溯的 Bar 数，默认值为当前 Bar。

**技术指标属性的设置**

在属性里的常规下填写公式的简称、名称、分类以及注释。也可更改参数等设置：



在“线型”里选择更改技术指标的输出形态，如线条、柱状图、十字图等，并且选择自己爱好的颜色，以更加个性化的表现形式来迎合您的看盘习惯。



## TradeBlazer 公式入门教程(2)

### Step 4:

前面我们所建的技术指标 MyVol,可以输出成交量，但成交量只能设置为属性所选的一种颜色。如下图：



很多朋友习惯于看红绿色表示涨跌的成交量。

下面我们来实现带红绿颜色的成交量指标，代码如下：

[\[Copy to clipboard\]](#) [\[-\]](#)

**CODE:**

Begin

```
PlotNumeric("Vol",Vol,IIf(Close>=Open,Red,Green));
```

End

使用的情形如下：



[关于 IIF](#)

函数原形:

Numeric IIF(Bool Conditon,Numeric TrueValue,Numeric FalseValue)

参数:

Conditon 条件表达式;

TrueValue 条件为 True 时的返回值;

FalseValue 条件为 False 时的返回值。

针对上面的使用 IIF 进行成交量颜色指定的脚本， 我们还有另外一种写法:

[Copy to clipboard] [ - ]

CODE:

Begin

If(Close>=Open)

PlotNumeric("Vol",Vol,Red);

Else

PlotNumeric("Vol",Vol,Green);

End

PlotNumeric 由输出的名字来区分是否为同一条线!

## 关于 IF 语句

If 语句是一个条件语句，当特定的条件满足后执行一部分操作。

语法如下:

If (Condition)

{

TradeBlazer 公式语句;

}

TradeBlazer 公式语句是一些语句的组合，如果 TradeBlazer 公式语句是单条，您可以省略{}，二条或者二条以上的语句必须使用{}。

## TradeBlazer 公式入门教程(3)

### Step5

## 关于条件表达式

o

## 逻辑操作符

逻辑运算符常常用于比较两个 True/False 的表达式，共有三个逻辑操作符: AND(&&)，OR(||)，NOT(!)。

表达式 1 AND 表达式 2o

表达式 1 OR 表达式 2o

o NOT 表达式 1

如下图所示可以让大家更清晰地理解逻辑操作符在表达式中的运算结果

下表列出AND逻辑操作符的应用情况：

表达式1	表达式2	表达式1 AND 表达式2
True	True	True
True	False	False
False	True	False
False	False	False

下表列出OR逻辑操作符的应用情况：

表达式1	表达式2	表达式1 OR 表达式2
True	True	True
True	False	True
False	True	True
False	False	False

下表列出NOT逻辑操作符的应用情况：

表达式1	NOT表达式1
True	False
False	True

## TradeBlazer 公式入门教程(4)

### Step6

前面第一贴已经讲过了 IF 语句,接下来要讲解条件语句的另外三种表达方式:

**If-Else**

**If-Else-If**

**If-Else 的嵌套**

### 关于 If-Else 语句

If-Else 语句是对指定条件进行判断，如果条件满足执行 If 后的语句。否则执行 Else 后面的语句。

语法如下：

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```
If (Condition)
{
    TradeBlazer 公式语句 1;
}Else
{
    TradeBlazer 公式语句 2;
}
```

Condition 是一个逻辑表达式，当 Condition 为 True 的时候，TradeBlazer 公式语句 1 将会被执行；Condition 为 False 时，TradeBlazer 公式语句 2 将会被执行。Condition 可以是多个条件表达式的逻辑组合，Condition 必须用()括起来。

TradeBlazer 公式语句是一些语句的组合，如果 TradeBlazer 公式语句是单条，您可以省略{}，二条或者二条以上的语句必须使用{}。

例如，比较当前 Bar 和上一个 Bar 的收盘价，如果 Close > Close[1]，Value1 = Value1 + Vol；否则 Value1 = Value1 - Vol，脚本如下：

[Copy to clipboard] [ - ]

**CODE:**

```
If (Close > Close[1])
    Value1 = Value1 + Vol;
Else
    Value1 = Value1 - Vol;
```

### 关于 If-Else-If 的语句

If-Else-If 是在 If-Else 的基础上进行扩展，支持条件的多重分支。

语法如下：

[Copy to clipboard] [ - ]

**CODE:**

```
If (Condition1)
{
    TradeBlazer 公式语句 1;
}Else If(Condition2)
{
    TradeBlazer 公式语句 2;
}Else
{
    TradeBlazer 公式语句 3;
}
```

Condition1 是一个逻辑表达式，当 Condition1 为 True 的时候，TradeBlazer 公式语句 1 将会被执行，Condition1 为 False 时，将会继续判断 Condition2 的值，当 Condition2 为 True 时，TradeBlazer 公式语句 2 将会被执行。Condition2 为 False 时，TradeBlazer 公式语句 3 将会被执行。Condition1，Condition2 可以是多个条件表达式的逻辑组合，条件表达式必须用()括起来。

TradeBlazer 公式语句是一些语句的组合，如果 TradeBlazer 公式语句是单条，您可以省略{}，二条或者二条以上的语句必须使用{}。

If-Else-If 的语句可以根据需要一直扩展，在最后的 Else 之后再加 If(Condition)和新的执行代码即可。当然您也可以省略最后的 Else 分支，语法如下：

[Copy to clipboard] [ - ]

**CODE:**

```
If (Condition1)
{
    TradeBlazer 公式语句 1;
}Else If(Condition2)
{
    TradeBlazer 公式语句 2;
}
```

### If-Else 的嵌套

If-Else 的嵌套是在 If-Else 的执行语句中包含新的条件语句，即一个条件被包含在另一个条件中。

语法如下：

[Copy to clipboard] [ - ]

**CODE:**

```
If (Condition1)
{
    If (Condition2)
    {
        TradeBlazer 公式语句 1;
    }Else
    {
        TradeBlazer 公式语句 2;
    }
}Else
{
    If (Condition3)
    {
        TradeBlazer 公式语句 3;
    }Else
    {
        TradeBlazer 公式语句 4;
    }
}
```

Condition1 是一个逻辑表达式，当 Condition1 为 True 的时候，将会继续判断 Condition2 的值，当 Condition2 为 True 时，TradeBlazer 公式语句 1 将会被执行。Condition2 为 False 时，TradeBlazer 公式语句 2 将会被执行。当 Condition1 为 False 的时候，将会继续判断 Condition3 的值，当 Condition3 为 True 时，TradeBlazer 公式语句 3 将会被执行。Condition3 为 False 时，TradeBlazer 公式语句 4 将会被执行。Condition1，Condition2，

Condition3 可以是多个条件表达式的逻辑组合，条件表达式必须用()括起来。

TradeBlazer 公式语句是一些语句的组合，如果 TradeBlazer 公式语句是单条，您可以省略{}，二条或者二条以上的语句必须使用{}。

例如，在一个交易指令中，条件设置如下：当前行情上涨的时候，如果收盘价高于开盘价时，则产生一个以收盘价买入 1 张合约；否则产生一个以开盘价买入 1 张合约。当前行情没有上涨的时候，如果收盘价高于开盘价，则产生一个以收盘价卖出 1 张合约；否则产生一个以开盘价卖出 1 张合约。脚本如下：

[\[Copy to clipboard\]](#) [\[ - \]](#)

**CODE:**

```
If (Open > High[1])
{
    If (Close>Open)
    {
        Buy(1,close);
    }Else
    {
        Buy(1,open);
    }
}Else
{
    If (Close > Open)
    {
        Sell(1,close);
    }Else
    {
        Sell (1,open);
    }
}
```

## TradeBlazer 公式入门教程(5)

### Step7

现在再回到成交量指标

有人喜欢在成交量指标上加均线，我们来看如何实现这样的功能。

[\[Copy to clipboard\]](#) [\[ - \]](#)

**CODE:**

```
Begin
    PlotNumeric("Vol",Vol);
    PlotNumeric("AvgVol5",AverageFC(Vol,5));
End
```



[Copy to clipboard] [ - ]

**CODE:**

```
Params
    NumericSeries Price(1);
    Numeric Length(10);

Vars
    Numeric AvgValue;

Begin
    AvgValue = SummationFC(Price, Length) / Length;
    Return AvgValue;

End
```

Average 和 AverageFC 有什么不同呢？AverageFC 是指 FastCalculate,即快速计算。当这两个函数的第二个变量,即 N 个 Bar 是常量时,使用 AverageFC,提高计算效率。当 N 是不确定的变量时,则必须使用 Average,否则会出现计算问题。

单看 Average 和 AverageFC 似乎是一样的,唯一不同的是 AvgValue 的计算方式用到的是 Summation 和 SumamtionFC。

### Summation 和 SumamtionFC

现在再来看看 Summation 与 SumamtionFC 的不同之处。公式表达如下：

### Summation

[\[Copy to clipboard\]](#) [\[ - \]](#)**CODE:**

```
Params
    NumericSeries Price(1);
    Numeric Length(10);

Vars
    Numeric SumValue(0);
    Numeric i;

Begin
    If (CurrentBar >= Length-1)
    {
        for i = 0 to Length - 1
        {
            SumValue = SumValue + Price[i];
        }
    }Else
    {
        SumValue = InvalidNumeric;
    }
    Return SumValue;

End
```

## SummationFC

[\[Copy to clipboard\]](#) [\[ - \]](#)

### CODE:

#### Params

```
NumericSeries Price(1);  
Numeric Length(10);
```

#### Vars

```
NumericSeries SumValue(0);  
Numeric i;
```

#### Begin

```
    If ( CurrentBar < Length || Price[Length] == InvalidNumeric || SumValue[1] ==  
InvalidNumeric )  
    {  
        for i = 0 to Length - 1  
        {  
            SumValue = SumValue + Price[i];  
        }  
    }Else  
    {  
        SumValue = SumValue[1] + Price - Price[Length] ;  
    }  
    Return SumValue;
```

#### End

关于 [Average](#) 函数的参数

Numeric Average(NumericSeries Price, Numeric Length);

Price 需要进行平均的序列变量

Length 平均时回溯的 Bar 数量

## TradeBlazer 公式入门教程(6)

### Step9

接下来我们再说一下 [常量](#)与[变量](#)的定义

#### 常量

是用来代替一个数或字符串的名称，在公式整个执行过程中不发生改变。

#### 变量

是一个存储值的地址，当变量被声明之后，就可以在脚本中使用变量，可以对其赋值，也可以在其他地方引用变量的值进行计算，要对变量进行操作，直接使用变量名称即可。

变量的主要用处在于它可以存放计算或比较的结果，以方便在之后的脚本中直接引用运算的值，而无需重现计算过程。

例如，我们定义一个变量 Y，我们把一个收盘价(Close)乘上 8% 的所得的值存储在 Y 中，即  $Y = \text{Close} * 8\%$ 。那么一旦计算出  $\text{Close} * 8\%$  的值，便赋给变量 Y。而无需在公式中输入计算过程，只需调用变量名称即可引用变量的值。

变量有助于程序的优化，这是 TradeBlazer 公式必须重复调用一些数据，这些数据可能是某些函数（如：Bar 数据），或通过表达式执行计算和比较的值。因此，在表达式频繁使用的地方使用变量可提高程序的运行速度和节约内存空间。

使用变量也可以避免输入错误，使程序的可读性提高，示例如下：

未使用变量的公式代码：

[Copy to clipboard] [ - ]

**CODE:**

```
If(Close > High[1] + Average(Close,10)*0.5)
{
    Buy(100, High[1] + Average(Close,10)*0.5);
}
```

如果使用变量，则整个代码变得简洁：

[Copy to clipboard] [ - ]

**CODE:**

```
Value1 = High[1] + Average(Close,10)*0.5;
If (Close > Value1)
{
    Buy(100,Value1);
}
```

如果一些表达式的组合经常在不同的公式中被调用，这个时候变量就不能实现功能，变量只能在单个公式的内部使用，这个时候我们需要建立用户函数来完成这些功能，详细说明参见[用户函数](#)（在 TB 软件里按 F1 便会出现联机帮助--公式系统--公式应用--用户函数）。

## 变量类型

TradeBlazer 公式支持有三种基本数据类型：数值型(Numeric)、字符串(String)、布尔型(Bool)。为了通过用户函数返回多个值，我们对三种数据类型进行了扩展，增加了引用数据类型。另外，为了对变量，参数进行回溯，我们增加了序列数据类型。因此，我们的数据类型共有九种。但对于变量定义，引用类型是无效的，剩余六种数据类型中分为简单和序列两大类，简单类型变量是单个的值，不能对其进行回溯，序列类型变量是和 Bar 长度一致的数据排列，我们可以通过回溯来获取当前 Bar 以前的任意值。

## 9 种数据类型

Bool	布尔型。
BoolRef	布尔型引用。
BoolSeries	和周期长度一致的 Bool 型序列值。
Numeric	数值型。
NumericRef	数值型引用。
NumericSeries	和周期长度一致的 Numeric 型序列值。
String	字符串。
StringRef	字符串引用。
StringSeries	和周期长度一致的 String 型序列值。

## 变量声明

在使用变量之前，必须对变量进行声明，TradeBlazer 公式使用关键字"Vars"来进行变量宣告，并指定变量类型。可以选择赋默认值，也可以不赋默认值。

变量定义的语法如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    变量类型 变量名 1(初值);
    变量类型 变量名 2(初值);
    变量类型 变量名 3(初值);
```

下面是一些变量定义的例子：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    NumericSeries    MyVal1(0);           //定义数值型序列变量 MyVal1，默认值为 0;
    Numeric          MyVal2(0);           //定义数值型变量 MyVal2，默认值为 0;
    Bool             MyVal3(False);        //定义布尔型变量 MyVal3，默认值为 False;
    String           MyVal4("Test");       //定义字符串变量 MyVal4，默认值为 Test。
```

变量定义的个数没有限制，变量名称的命名规范详细说明参见[命名规则](#)。

整个公式中**只能出现一个Vars宣告，并且要放到公式的开始部分，在参数定义之后，正文之前。**

## 变量的默认值

在声明变量时，通常会赋给变量一个默认值。例如上例中的 0，False，"Test"等就是变量的默认值。如果某个变量没有赋予默认值，系统将会自动给该变量赋予默认值。数值型变量的默认值为 0，布尔型变量的默认值为False，字符串的默认值为空串。

变量的默认值是在当公式在执行时，给该变量赋予的初值，使该变量在引用时存在着有效的值。在该公式每个Bar的执行过程中，改变量的默认值都会被重新赋值。

## 变量赋值

变量声明完成之后，您可以在脚本正文中给变量指定一个值。

语法如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Name = Expression;
```

"Name"是变量的名称，表达式的类型可以是数值型、布尔型、字符串中的任何一种。不过表达式的类型一定要和变量的数据类型相匹配。如果变量被指定为是数值型的，那么表达式一定要是数值型的表达式。

例如：下面的语句将 Close 的 10 周期平均值赋值给变量 Value1：

[Copy to clipboard] [ - ]

**CODE:**

```
Value1 = Average(Close , 10);
```

在下面这个语句中，声明了一个名为"KeyReversal"的逻辑型变量，然后又把计算的值赋给它。

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    Bool      KeyReversal(False);
Begin
    KeyReversal = Low < Low[1] AND Close > High[1];
    ...
End
```

## 变量使用

变量定义、赋值之后，在表达式中直接使用变量名就可以引用变量的值。例如在下面的语句中计算了买入价格后，把值赋给数值型变量 EntryPrc，在买入指令中便可直接应用变量名，通过变量名便可引用变量的值：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    Numeric EntryPrc(0);
```

```

Begin
    EntryPrc = Highest(High,10);
    If (MarkerPosition <> 1)
    {
        Buy(1,EntryPrc);
    }
End

```

接下来的例子，我们计算最近 10 个 Bar 最高价中的最大值（不包括当前 Bar），对比当前 High，然后通过 If 语句，产生报警信息。

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```

Vars
    Bool    Con1(False);
Begin
    Con1 = High > Highest(High[1],10);
    If(Con1)
    {
        Alert("New 10-bar high");
    }
End

```

其实我们并不一定都要应用条件为 True 的情况，有时候我们需要判断条件为 False 的时候执行某些代码，如下的例子：

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```

Vars
    Bool    Con1(False);
Begin
    Con1 = High < Highest(High[1],10) AND Low > Lowest(Low[1],10);
    If(Con1==False)
    {
        Alert("New high or low");
    }
End

```

## 序列变量

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```

Vars
    NumericSeries MyNumSVal(0);

```

```
BoolSeries      MyBoolVal(False);
StringSeries    MyStrVal("");
```

序列变量和简单变量一样，可以对其赋予默认值。

序列变量定义之后，您可以象简单变量一样的对其使用，不会有任何的不同。除了支持全部简单变量的功能之外，序列变量还可以通过"[nOffset]"来回溯以前的变量值，详细说明参见[变量回溯](#)。

对于序列变量，TradeBlazer公式在内部针对其回溯的特性作了很多的特殊处理，也需要为序列变量保存相应的历史数据，因此，和简单变量相比，执行的速度和占用内存空间方面都作了一些牺牲。因此，尽管您可以定义一个序列变量，把它当作简单变量来使用，但是，我们强烈建议您只将需要进行回溯的变量定义为序列变量。

## 成交量指标的扩充

回到成交量指标上,在成交量和均线的基础上再增加输出一条线，显示成交量和 5 日均线的差值。

[\[Copy to clipboard\]](#) [\[ - \]](#)

### CODE:

```
Params
    Numeric Length(5);
Begin
    PlotNumeric("Vol",Vol);
    PlotNumeric("AvgVol",AverageFC(Vol, Length));
    PlotNumeric("VolDiff",Vol-AverageFC(Vol,Length));
End
```

这样的写法效率较低，因此我们引入变量来进行处理。

## 对成交量指标的优化

我们通过使用变量来使代码变得简洁，并提高执行效率。上面的公式修改如下：

[\[Copy to clipboard\]](#) [\[ - \]](#)

### CODE:

```
Params
    Numeric Length(5);
Vars
    Numeric AvgVol5;
Begin
    AvgVol5 = AverageFC(Vol, Length);
    PlotNumeric("Vol",Vol);
    PlotNumeric("AvgVol",AvgVol5);
    PlotNumeric("VolDiff",Vol- AvgVol5);
End
```



如下图所示是增加了差值的成交量：



TradeBlazer 公式入门教程(7)

Step10

回到加均线的成交量指标

刚才的成交量指标输出的 5 周期均线，有的人希望使用 10 周期均线，为了方便使用，我们使用参数来处理这种需求。

[Copy to clipboard] [ - ]

```
CODE:
Params
    Numeric Length(5);
Begin
    PlotNumeric("Vol",Vol);
    PlotNumeric("AvgVol",AverageFC(Vol, Length));
End
```

在上述公式中只要将参数Numeric Length后面的数值（5）改成（10）便可以啦。

关于参数

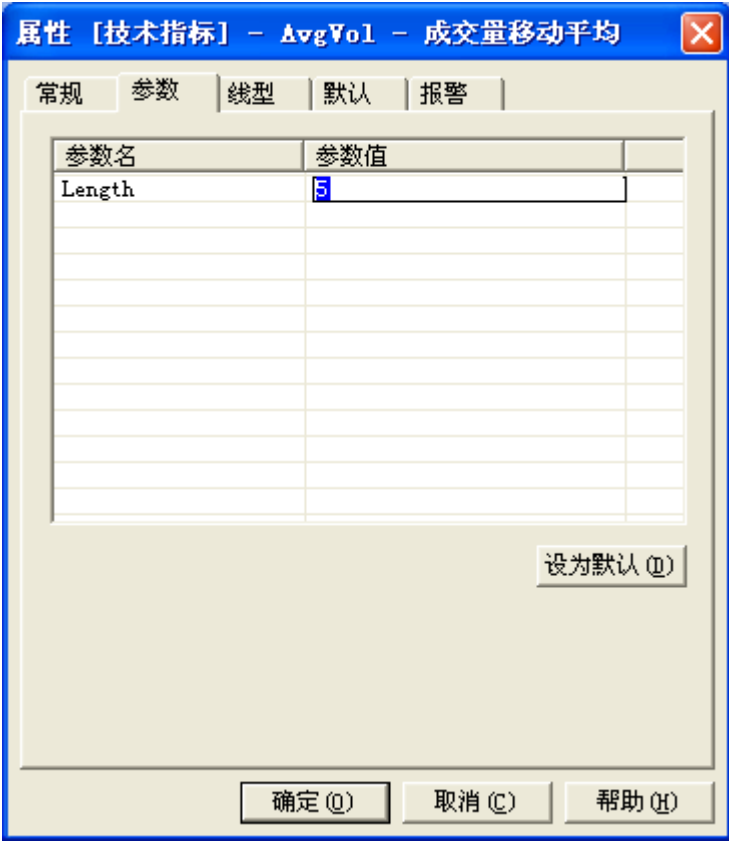
参数

参数是一个预先声明的地址，用来存放输入参数的值，在声明之后，您就可以在接下来的公式中使用该参数的名称来引用其值。

参数的值在公式的内部是不能够被修改，在整个程序中一直保持不变，不能对参数进行赋值操作(引用参数是个特例)。参数的好处在于您可以在调用执行技术分析，交易指令的时候才指定相应的参数，而不需要重新编译。

例如，我们常用的移动平均线指标，就是通过不同的Length来控制移动平均线的周期，在调用指标时可以随意修改各个Length的值，使之能够计算出相对应的移动平均线。您可以指定 4 个参数为 5,10,20,30 计算出这 4 条移动平均线，也可以修改 4 个参数为 10，22，100，250 计算出另外的 4 条移动平均线。

参数的修改很简单，在超级图表调用指标的过程中，您可以打开指标的属性设置框，切换到参数页面，手动修改各项参数的值，然后应用即可（如下图），交易开拓者将根据新的参数设置计算出新的结果，在超级图表中反映出来。



另外，参数的一个额外的优点是，我们可以通过修改交易指令不同的参数，计算交易指令组合的优劣，达到优化参数的目的。

### 参数类型

在介绍参数类型之前，我们需要对于TradeBlazer公式的六种类型作一些说明，用户函数是六种公式中比较特殊的一类，它自身不能被超级图表，行情报价这样的模块调用，只能被其他五类公式或者用户函数调用，因此它的参数类型也和其他几种不一样。

用户函数的参数类型可以包含TradeBlazer公式的九种类型，而其他五类公式只能使用三种简单的基本类型。

三种简单类型参数通过传值的方式将参数值传入公式，公式内部通过使用参数名称，将参数值用来进行计算或赋值。

引用参数是在调用的时候传入一个变量的地址，在用户函数内部会修改参数的值，在函数执行完毕，上层调用的公式会通过变量获得修改后的值，引用参数对于需要通过用户函数返回多个值的情况非常有用。

序列参数可以通过回溯获取以前Bar的值，具体介绍可参见[参数回溯](#)。

### 参数声明

在使用参数之前，必须对参数进行声明，TradeBlazer公式使用关键字"Params"来进行参数宣告，并指定参数类型。可以选择赋默认值，也可以不赋默认值。如果某个参数没有赋予默认值，则这个参数之前的其他参数的默认值都将被忽略。

参数定义的语法如下：

[Copy to clipboard] [ - ]

**CODE:**

Params  
    参数类型 参数名 1(初值);  
    参数类型 参数名 2(初值);  
    参数类型 参数名 3(初值);

下面是一些参数的例子：

[Copy to clipboard] [ - ]

**CODE:**

Params  
    Bool          bTest(False);      //定义布尔型参数 bTest，默认值为 False;  
    Numeric      Length(10);         //定义数值型参数 Length，默认值为 10;  
    NumericSeries  Price(0);          //定义数值型序列参数 Price，默认值为 0;  
    NumericRef     output(0);         //定义数值型引用参数 output，默认值为 0;  
    String        strTmp("Hello");    //定义字符串参数 strTmp,默认值为 Hello;

参数名称的命名规范详细说明参见[命名规则](#)。

整个公式中只能出现一个Params宣告，并且要放到公式的开始部分，在变量定义之前

参数的默认值

在声明参数时，通常会赋给参数一个默认值。例如上例中的False，10，0 等就是参数的默认值。用户函数的默认值是在当用户函数被其他公式调用，省略参数时作为参数的输入值，其他五种公式的默认值是用于图表，报价等模块调用公式时默认的输入值。

参数的默认值的类型在定义的时候指定，默认值在公式调用的时候传入作为参数进行计算。只能够对排列在后面的那些参数提供默认参数，例如：

[Copy to clipboard] [ - ]

CODE:

Params

Numeric

MyVal1;

Numeric

MyVal2(0);

Numeric

MyVal3(0);

您不能够使用以下方式对参数的默认值进行设定：

[Copy to clipboard] [ - ]

CODE:

Params

Numeric

MyVal1(0);

Numeric

MyVal2(0);

Numeric

MyVal3;

参数的使用

在声明参数之后，我们可以在脚本正文中通过参数名称使用该参数，在使用的过程中要注意保持数据类型的匹配，示例如下：

[Copy to clipboard] [ - ]

CODE:

Params

NumericSeries

Price(1);

Vars

Numeric

CumValue(0);

Begin

CumValue = CumValue[1] + Price;

Return CumValue;

End

在以上的公式中，首先定义了一个数值型序列参数Price，并将其默认值设置为 1。接着定义了一个变量CumValue。脚本正文中，将CumValue的上一个Bar值加上Price，并将值赋给CumValue，最后返回CumValue。

通过上述的公式可以看到，我们只需要调用参数名，就可以使用参数的值进行计算了，如果要对序列参数进行回溯，请参见[参数回溯](#)。

### 引用参数

TradeBlazer公式的用户函数可以通过返回值，返回函数的计算结果，返回值只能是三种简单类型。当我们需要通过函数进行计算，返回多个值的时候，单个的返回值就不能满足需求了。在这种情况下，我们提出了引用参数的概念，引用参数是在调用的时候传入一个变量的地址，在用户函数内部会修改参数的值，在函数执行完毕，上层调用的公式会通过变量获得修改后的值。因为引用参数的使用是没有个数限制，因此，我们可以通过引用参数返回任意多个值。

例如，用户函数MyFunc如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Params
    NumericSeries   Price(0);
    NumericRef       oHigher(0);
    NumericRef       oLower(0);
Vars
    Numeric          Tmp(0);
Begin
    Tmp = Average(Price,10);
    oHigher = IIf(Tmp > High,Tmp,High);
    oLower = IIf(Tmp < Low,Tmp,Low);
    Return Tmp;
End
```

以上代码通过两个数值型引用参数返回 10 个周期的 Price 平均值和最高价的较大值 oHigher，以及 10 个周期的 Price 平均值和最低价的较小值 oLower，并且通过函数返回值输出 10 个周期的 Price 平均值。在调用该用户函数的公式中，可以通过调用该函数获得 3 个计算返回值，示例如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    Numeric AvgValue;
    Numeric HigherValue;
    Numeric LowerValue;
Begin
    AvgValue = MyFunc(Close,HigherValue,LowerValue);
    ...
End
```

## TradeBlazer 公式入门教程(8)

### Step11

#### 创建稍复杂的指标

我们以 **MACD** 指标为例，介绍一般指标的写法。

**MACD** 的技术指标包含参数，变量和正文三部分。  
如下所示：

Params	公式参数段，用Params宣告参数代码段的开始
<code>Numeric FastLength(12);</code> <code>Numeric SlowLength(26);</code> <code>Numeric MACDLength(9);</code>	
Vars	公式变量段，用Params宣告变量代码段的开
<code>NumericSeries MACDValue;</code> <code>Numeric AvgMACD;</code> <code>Numeric MACDDiff;</code>	
Begin	公式脚本段，用Begin和End宣告代码的起始和结束，一般技术指标输出一条线用PlotNumeric函数
<code>MACDValue = XAverage( Close, FastLength ) - XAverage( Close, SlowLength );</code> <code>AvgMACD = XAverage(MACDValue,MACDLength);</code> <code>MACDDiff = MACDValue - AvgMACD;</code> <code>PlotNumeric("MACD",MACDValue);</code> <code>PlotNumeric("MACDAvg",AvgMACD);</code> <code>If (MACDDiff &gt;= 0)</code> <code>    PlotNumeric("MACDDiff",MACDDiff,Red);</code> <code>Else</code> <code>    PlotNumeric("MACDDiff",MACDDiff,Green);</code> <code>PlotNumeric("零线",0);</code>	
End	

MACD 输出 4 条线。包括线条和变色柱状线及零线。



## TradeBlazer公式入门教程(9)

### Step12

#### 关于价差指标-Spread

##### 如何进行商品叠加？

交易开拓者的超级图表支持商品叠加的显示，当叠加的图表调用各项公式时，可能有需要使用叠加的商品对应的基础数据，针对这样的需求，TradeBlazer公式提供了叠加数据的支持。详细请参照贴子[如何在超级图表中叠加商品，并进行价差运算](#)

##### 价差指标的公式代码

下面我们就举一个简单例子，两个商品以收盘价来计算的价差：

[\[Copy to clipboard\]](#) [\[ - \]](#)

##### CODE:

Begin

```
If(Data0.Close != InvalidNumeric && Data1.Close != InvalidNumeric)
{
    PlotNumeric("Spread",Data0.Close - Data1.Close);
}
```

```
}  
End
```

有些人习惯将价差作为 K 线形式表现出来，如下图所示：



图中蓝线就代表了两个商品收盘价的价差

### 关于在价差基础上扩展出的 SpreadEX

有些人还习惯于将不同商品的开盘价、收盘价、最高价以及最低价的价差再集中体现在同一图表上，如下例：

[\[Copy to clipboard\]](#) [\[-\]](#)

#### CODE:

```
Begin  
If(Data0.Close != InvalidNumeric && Data1.Close != InvalidNumeric0  
{  
    PlotNumeric("Open", Data0.Open - Data1.Open);  
    PlotNumeric("Close", Data0.Close - Data1.Close);  
    PlotNumeric("High", max(Data0.High - Data1.High, Data0.Low - Data1.Low));  
    PlotNumeric("Low", min(Data0.High - Data1.High, Data0.Low - Data1.Low));  
}
```



```
}  
End
```

最后这四个指标在两个商品中的价差得到的 K 线图示如下：



## TradeBlazer 公式入门教程(10)

### Step13

#### 数据回溯

在 TradeBlazer 公式中有三种类型的数据回溯：变量回溯、参数回溯和函数回溯。

如何使用回溯表达？ XXX[nOffset]

nOffset 是要回溯引用的 Bar 相对于当前 Bar 的偏移值，该值必须大于等于 0，当 nOffset = 0 时，即为获取当前 Bar 的参数值。并且 nOffset 不能大于当时的 CurrentBar，这样会导致数据访问越界。造成不可预知的计算结果。

#### 变量回溯

TradeBlazer 公式共支持九种数据类型，但对于变量定义，引用类型是无效的，剩余六种数据类型中分为简单和序列两大类，简单类型变量是单个的值，不能对其进行回溯，序列类型变量是和 Bar 长度一致的数据排列，我们可以通过回溯来获取当前 Bar 以前的任意值。

要使用变量回溯，需要在变量的后面，使用中括号"[nOffset]"，nOffset 是要回溯引用的 Bar 相对于当前 Bar 的偏移值，该值必须大于等于 0，当 nOffset = 0 时，即为获取当前 Bar 的变量值。

例如，我们定义如下技术指标：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    NumericSeries MyVal;
Begin
    MyVal = Average(Close,10);
    PlotNumeric("MyVal",MyVal[3]);
End
```

以上公式定义数值型序列变量 MyVal，MyVal 等于收盘价的 10 个周期的平均值，然后将序列变量 MyVal 的前 3 个 Bar 数据输出。

以上公式 MyVal 的前 9 个数据因为需要计算的 Bar 数据不足，返回无效值，从第 10 个 Bar 开始，MyVal 获取到正确的平均值，但是我们需要输出的数据是 MyVal[3]，即前 3 个 Bar 的数据，因此，直到第 12 个 Bar，有效的数据才会被输出。以上公式的 12 是该公式需要的最少引用周期数，如果将输出信息画到超级图表中，前 11 个 Bar 是没有图形显示的。

当 nOffset>CurrentBar 或者 nOffset<0 时，对于变量的回溯都将越界，这种情况下，将返回无效值。

参数回溯

TradeBlazer 公式支持的九种基本类型，在用户函数的参数定义中全部支持，在其他的公式中参数定义只支持三种简单类型。因此，关于参数的回溯问题，只对用户函数有效，下面我们举例说明用户函数序列参数的使用。

要使用参数回溯，需要在参数的后面，使用中括号"[nOffset]"，nOffset 是要回溯引用的 Bar 相对于当前 Bar 的偏移值，该值必须大于等于 0，当 nOffset = 0 时，即为获取当前 Bar 的参数值。

例如，我们定义一个用户函数 MyFunc，脚本如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Params
    NumericSeries Price(0);
    Numeric Length(10);
```

```

Vars
    Numeric      MyAvg;
    Numeric      MyDeviation;
Begin
    MyAvg = Summation(Price,Length)/Length;
    MyDeviation = MyAvg - Price[Length];
    Return MyDeviation;
End

```

以上的例子，对输入的 Price 我们求其 10 个周期的平均值，然后求出该平均值和 Price 的前 Length 个 Bar 的值之间的差值，将其返回。对于 Price[Length] 这样的参数回溯引用，其实现原理和上节所描述的变量回溯引用基本一致。

## 函数回溯

函数回溯分为系统函数的回溯和用户函数的回溯。

系统函数中回溯的使用主要是针对 Bar 数据。比如我们需要获取上 2 个 Bar 的收盘价，脚本为 Close[2]；又或者我们需要获取 10 个 Bar 前的成交量，脚本为 Vol[10]。对于 Bar 数据的回溯是系统函数中最常用的，虽然也可以对行情数据和交易数据等进行回溯，但是大部分并无实质的意义，返回的结果和不回溯是一样的，因此，不推荐如此使用。

要对函数回溯引用，我们可以通过在函数名称后面添加 "[nOffset]" 获取其回溯值，nOffset 是要回溯引用的 Bar 相对于当前 Bar 的偏移值，该值必须大于等于 0，当 nOffset = 0 时，即为获取当前 Bar 的参数值。

带有参数的函数回溯，需要将 "[nOffset]" 放到参数之后，另外，无参数和使用默认参数的情况下，函数调用的括号可以省略。例如:Close[2] 等同于 Close()[2]。

用户函数的回溯和系统函数原理基本一致，但考虑到系统的执行速度和效率等因素，目前，TradeBlazer 公式不支持对用户函数的回溯，如果您想要获取用户函数的回溯值，建议您将函数返回值赋值给一个序列变量，通过对序列变量的回溯来达到相同的目的。

如下面的脚本所示，取 Close 的 10 个 Bar 平均值的 4 个周期前的回溯值：

[\[Copy to clipboard\]](#) [\[ - \]](#)

```

CODE:
Vars
    NumericSeries AvgValue;
    Numeric      TmpValue;
Begin
    AvgValue = Average(Close,10);
    TmpValue = AvgValue[4];
    ...
End

```

## TradeBlazer 公式入门教程(11)

### Step14

TradeBlazer 公式包含的公式类型如下：

**用户函数**    **用户字段(暂时未用)**    **技术指标**    **K 线型态**    **特征走势**    **交易指令**

**用户函数：**用户函数是能够通过函数名称进行引用的指令集，它执行一系列操作并返回一个值。您可以在其他任何公式中使用用户函数进行计算；

**用户字段：**用户字段是 TradeBlazer 公式为交易开拓者报价类窗体提供的一项数据输出公式，通过用户字段执行一系列语言指令，给报价窗体返回一个特定的显示值；

**技术指标：**技术指标是基于基础数据，通过一系列的数学运算，在每个 Bar 返回相应的结果值的一类公式，这些值在图表模块中输出为线条、柱状图、点等表现形式；

**K 线型态：**K 线型态是类似于技术指标的一类公式，它主要着重于反映一段 K 线的特定型态，并通过不同的技术指标的方式输出到图表；

**特征走势：**特征走势是类似于技术指标的一类公式，它主要着重于反映整个价格曲线的趋势、变化特征，并通过特定的表达方式输出到图表；

**交易指令：**交易指令是包含买、卖、平仓，头寸，仓位控制的并执行交易指令的一类公式，它主要帮助您将您的交易思想转化为计算机的操作。

通过调用 TradeBlazer 公式，您可以在交易开拓者中进行技术分析、交易策略优化测试、公式报警、自动交易等操作。

### Step15

#### 用户函数

用户函数是可以通过名称进行调用的一组语句的集合，用户函数返回一个值，这个值可以是 Numeric, Bool, String 三种类型中的任何一种。您可以在需要的任何地方调用用户函数来完成相应的功能。

例如，在 TradeBlazer 公式中经常使用的一个用户函数 Summation，Summation 通过输入 Price 序列数据，以及 Length 统计周期数，计算 Price 最近 Length 周期的和，每次用户需要进行求和计算的时候，都可以调用 Summation 代替冗长的求和代码，输入参数并获取返回值。

Summation 是 TradeBlazer 公式中一个比较简单的用户函数，TradeBlazer 公式提供了上百个内建用户函数，

当然，您也可以编写您自己的用户函数。

用户函数通过参数传递输入数据，通过引用参数或返回值传递输出数据，以上例子中的 Summation 函数，在被调用的时候格式如下：

#### CODE:

```
Value1 = Summation(Close,10);
```

[\[Copy to clipboard\]](#) [\[ - \]](#)

在调用Summation的时候，需要根据定义时候的参数列表和顺序，输入相应的输入参数，有默认值的参数可以省略输入参数。

用户函数在交易开拓者中使用有如下规则：

- \*支持九种类型的参数定义，支持指定参数默认值；
- \*支持使用引用参数，可通过引用参数返回多个数据；
- \*支持六种类型的变量定义，支持指定变量的默认值；
- \*可以访问行情数据、属性数据；
- \*必须通过Return返回数据，返回数据类型为三种基本类型之一；
- \*脚本中的返回数据类型必须和属性界面设置中一致；
- \*用户函数之间可以相互调用，用户函数自身也可以递归调用；
- \*用户函数可以根据设置调用部分的系统函数。

## 用户函数的类型

用户函数按照返回值类型不同可以分为数值型(Numeric)，布尔型(Bool)，字符串(String)三种基本类型，三种类型用户函数在调用时需要将返回值赋予类型相同的变量。

按照用户函数属性不同，用户函数可以分为内建用户函数和其他用户函数两种，内建用户函数是交易开拓者提供的，用于支持公式系统运行的预置公式，您可以查看和调用内建用户函数，但是不能删除和修改内建公式。

## 使用内建用户函数

TradeBlazer公式中提供上百个内建用户函数，一部分用户函数提供类似于求和，求平均，求线性回归等算法方面的功能，另外一些函数提供技术分析的一些算法，比如：RSI，CCI，DMI等,这些用户函数辅助完成技术分析。

在创建自己的技术分析和交易系统时，如果需要自己写一些算法，您可以首先在用户函数中查找是否有相应的内建用户函数，尽可能的多使用内建用户函数，减少出错的可能。您也可以编写自己的算法，以供在技术分析和交易系统中使用。

### 用户函数的参数

大部分用户函数都需要接受输入的信息进行计算，这些输入的信息，我们称之为参数。关于用户函数参数的使用详细说明参见[参数](#)。

### 如何编写用户函数

一个用户函数由三部分组成，参数定义，变量定义，脚本正文。

语法如下：

[Copy to clipboard] [ - ]

**CODE:**

Params  
    参数定义语句;  
Vars  
    变量定义语句;  
Begin  
    脚本正文;  
End

参数定义和变量定义部分在前面已经详细叙述过，脚本的正文部分将输入参数进行计算，得出函数的返回值，并通过 Return 返回。

例如，我们以 Average 为例，Average 计算 Price 在 Length 周期内的平均值。Average 调用 Summation 求和，并计算平均值，然后返回结果，脚本如下：

[Copy to clipboard] [ - ]

**CODE:**

Params  
    NumericSeries Price(1);  
    Numeric Length(10);  
Vars  
    Numeric AvgValue;  
Begin  
    AvgValue = Summation(Price, Length) / Length;  
    Return AvgValue;  
End

对于使用多个输出的情况，即使用引用参数的情况，我们以求 N 周期最大值为例进行描述，假定我们需要编写一个用户函数，该函数需要求出序列变量 Price 在最近 Length 周期内的最大值，并且要求出最大值出现的 Bar 和当前 Bar 的偏移值。脚本如下：

**CODE:**

## Params

```
NumericSeries Price(1);  
Numeric Length(10);  
NumericRef HighestBar(0);
```

## Vars

```
Numeric MyVal;  
Numeric MyBar;  
Numeric i;
```

## Begin

```
MyVal = Price;  
MyBar = 0;  
For i = 1 to Length - 1  
{  
    If ( Price[i] > MyVal)  
    {  
        MyVal = Price[i];  
        MyBar = i;  
    }  
}  
HighestBar = MyBar;  
Return MyVal;
```

## End

## 用户函数的调用

用户函数成功创建之后（编译/保存成功），您可以在其他的用户函数、技术分析、交易指令等公式中调用用户函数，调用用户函数时需要注意保持参数类型的匹配，即用户函数参数的声明数据类型需和调用时传入参数的数据匹配，这是所指的匹配是指基本数据类型：数值型，布尔型，字符串三种类型匹配，并且保持序列参数和传入变量类型的对应。我们可以对用户函数定义为 **Numeric** 或者 **NumericRef** 的参数使用 **Numeric** 类型的变量作为传入参数；但不能将在定义为 **NumericSeries** 类型的参数时传入 **Numeric**。具体的对应关系如下表：

函数参数声明类型	可传入的变量类型
<b>Numeric</b>	Numeric , NumericRef , NumericSeries
<b>NumericRef</b>	Numeric , NumericRef , NumericSeries
<b>NumericSeries</b>	NumericSeries
<b>Bool</b>	Bool , BoolRef , BoolSeries
<b>BoolRef</b>	Bool , BoolRef , BoolSeries
<b>BoolSeries</b>	BoolSeries
<b>String</b>	String , StringRef , StringSeries
<b>StringRef</b>	String , StringRef , StringSeries
<b>StringSeries</b>	StringSeries

对于函数的返回值，您也可以将用户函数的 **Numeric** 返回值赋值给 **NumericSeries** 或 **NumericRef** 变量。即在用户函数的返回值使用时，忽略其扩展数据类型。比如我们在调用 **Average** 求平均值时，可以这样调用：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    Numeric Value1;
Begin
    Value1 = Average(Close,10);
    ...
End
```

我们也可以按照以下方式进行调用：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    NumericSeries Value1;
Begin
    Value1 = Average(CloseTmp,10);
    ...
End
```

**A**用户函数调用自身，我们称之为直接递归；**A**用户函数可以调用**B**用户函数，同时**B**用户函数也可以调用**A**用户函数，对于这种情况，我们称之为间接递归；

不管是直接递归还是间接递归，用户函数在执行的时候，都可能遇到递归调用没有出口，导致死循环的情况。因此，我们在编写公式的时候，要注意避免使用递归算法，如果一定需要使用递归算法，要注意保证递归算法都有出口。



## 用默认参数调用用户函数

用户函数在被调用的时候，如果传入的参数和参数的默认值一样，可以省略输出参数，使用默认值来调用用户参数。只能对排列在后面的那些参数使用默认参数，默认参数的定义参见[参数](#)。

对于用户函数的直接递归调用，默认参数调用有一些特殊的意义，如下所示，用户函数Fun1：

[Copy to clipboard] [ - ]

**CODE:**

```
Params
    NumericSeries Price(1);
Vars
    Numeric CumValue(0);
Begin
    If(CurrentBar == 0)
    {
        CumValue = Price;
    }else
    {
        CumValue = Fun1[1] + Price;
    }
    Return CumValue;
End
```

技术指标 Ind1 调用 Fun1 的代码如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Value1 = Fun1(Close);
```

以上的用户函数通过默认参数调用 Fun1 的意思不是调用 Fun1(1)，而是将 Ind1 调用 Fun1 的 Close 传递下去，即求 Fun1(Close)的上一个 Bar 的值。以上 Ind1 调用 Fun1 的计算结果和调用如下的 Fun2 计算结果一致。

用户函数 Fun2：

[Copy to clipboard] [ - ]

**CODE:**

```
Params
    NumericSeries Price(1);
Vars
    Numeric CumValue(0);
Begin
    If(CurrentBar == 0)
    {
```

```
CumValue = Price;  
}else  
{  
    CumValue = Fun1(Close)[1] + Price;  
}  
Return CumValue;  
End
```

## TradeBlazer 公式入门教程(12)

### Step16

#### 用户字段

用户字段因为暂时还未用到，所以这部分就先跳过不讲

#### 技术指标

技术指标是最常用的一类公式，它通过计算一系列的数学公式，在每个 Bar 都返回值，这些值在图表模块中输出为线条、柱状图、点等表现形式，通过分析图形特点、走势和曲线帮助客户分析行情走势，得出合理的交易判断。

当技术指标应用在图表中时，您可以设置技术指标各输出值的表现形式，以及颜色、粗细等，如下图的点，线，柱状图所示：



技术指标的使用规则归纳如下：

支持三种基本类型的参数定义，支持指定参数默认值；

不支持使用引用参数；

支持六种类型的变量定义，支持指定变量的默认值；

可以访问 Data0-Data49 个数据源的 Bar 数据；

可以访问行情数据、属性数据；

必须通过 PlotNumeric、PlotBool、PlotString 返回数据，返回数据类型为三种基本类型的组合；

可以输出多组数据，通过 PlotNumeric、PlotBool、PlotString 的第一个参数，即输出名称来区分输出数据；

可以支持 Alert 来进行报警；

技术指标可以调用所有的用户函数进行计算；

技术指标可以根据设置调用部分的系统函数；

技术指标在执行时，必须要指定相应的数据源和周期，需要调用历史数据的，只有历史数据存在的情

况下才能返回正确的值。

示例，技术指标 RSI，脚本如下：

[\[Copy to clipboard\]](#) [\[ - \]](#)

**CODE:**

Params

```
Numeric Length(14);  
Numeric OverSold(20);  
Numeric OverBought (80);
```

Vars

```
Numeric RSIValue(0);  
Numeric RSIColor(-1);
```

Begin

```
RSIValue = RSI(Close,Length);  
If (RSIValue > OverBought)  
{  
    RSIColor = RED;  
}Else If (RSIValue < OverSold)  
{  
    RSIColor = CYAN;  
}  
PlotNumeric("RSI1", RSIValue, RSIColor);  
PlotNumeric("超卖", OverSold);  
PlotNumeric("超买", OverBought);  
  
If CrossOver(RSIValue,OverSold)  
{  
    Alert("Indicator exiting oversold zone");  
}  
If CrossUnder(RSIValue, OverBought)  
{  
    Alert("Indicator exiting overbought zone");  
}
```

End

技术指标 RSI 调用 RSI 内建用户函数计算出结果，然后判断其返回值和超买，超卖的关系，设置显示颜色，并产生报警信息。

技术指标在输出数据时，我们是通过输出值的名称来进行识别，名称相同则认为是一个数据，如下的代码，后面语句的输出数据将会覆盖前面语句的输出数据。

[\[Copy to clipboard\]](#) [\[ - \]](#)

**CODE:**

```
PlotNumeric("Test",10);  
PlotNumeric("Test",20);
```

最后"Test"输出的数据为 20，而不是 10。

## K 线型态

K 线型态是另外一种形式的技术分析公式，它对满足设定条件的 Bar 进行标记，使之醒目，便于客户进行分析。

当 K 线型态应用在图表中时，您可以设置其输出值的颜色、风格和粗细，如图所示：



K 线型态的使用规则归纳如下：

支持三种基本类型的参数定义，支持指定参数默认值；

不支持使用引用参数；

支持六种类型的变量定义，支持指定变量的默认值；

可以访问 Data0-Data49 个数据源的 Bar 数据；

可以访问行情数据、属性数据；

必须通过 PlotBar 返回数据；

只能输出一组数据，用名称进行区分；

可以支持 Alert 来进行报警；

K 线型态可以调用所有的用户函数进行计算；

K 线型态可以根据设置调用部分的系统函数；

K 线型态在执行时，必须要指定相应的数据源和周期，需要调用历史数据的，只有历史数据存在的情况下才能返回正确的值。

示例，K 线型态十字星，脚本如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Vars
    Bool Condition(False);
Begin
    Condition = (Abs(Close-Open)*10<(High-Low)) And
                (High <> Close) And (Low <> Close);
    If (Condition)
    {
        PlotBar("SZX",High,Low)
    }
End
```

K 线型态十字星判断条件，条件满足的情况下用 PlotBar 输出信息。

## 特征走势

特征走势是另外一种形式的技术分析公式，它对满足设定条件的 Bar 进行标记，使之醒目，便于客户进行分析。特征走势和 K 线型态有很多相似之处，最大的不同在于，K 线型态和特征走势的数据输出方式。

当特征走势应用在图表中时，您可以设置其输出值的表现形式，以及颜色、风格和粗细，如图所示：



特征走势的使用规则归纳如下：

支持三种基本类型的参数定义，支持指定参数默认值；

支持使用引用参数；

支持六种类型的变量定义，支持指定变量的默认值；

可以访问 Data0-Data49 个数据源的 Bar 数据；

可以访问行情数据、属性数据；

必须通过 PlotNumeric、PlotBool、PlotString 返回数据，返回数据类型为三种基本类型的组合；

只能输出一组数据，用名称进行区分；

可以支持 Alert 来进行报警；

特征走势可以调用所有的用户函数进行计算；

特征走势可以根据设置调用部分的系统函数；

特征走势在执行时，必须要指定相应的数据源和周期，需要调用历史数据的，只有历史数据存在的情况下才能返回正确的值。

示例，特征走势创历史新高，脚本如下：

[Copy to clipboard] [ - ]

**CODE:**

```
Params
    Numeric Length(5);
Vars
    Bool Condition(False);
Begin
    Condition = (High == Highest(High,Length)) ;
    If (Condition)
    {
        PlotNumeric("CLSXG",High)
    }
End
```

特征走势创历史新高判断条件，条件满足的情况下用 PlotNumeric、PlotBool、PlotString 输出信息。

交易指令

TradeBlazer 公式提供一种简单的方法表达您的交易思想，那就是使用交易指令，一个简单的交易指令如下：

[Copy to clipboard] [ - ]

**CODE:**

```
If (Condition)
    Buy (1,Close);
```

以上的语句表达的意思是：当某些条件满足了，将用当前 Bar 的收盘价买入 1 手指定商品。就像您平时通过经纪商进行交易操作一样，TradeBlazer 公式提供四个系统函数和现实中的四种交易动作进行对应，如下：

函数名	描述
Buy	平掉所有空头持仓，开多头仓位。
Sell	平掉指定的多头持仓。
SellShort	平掉所有多头持仓，开空头仓位。
BuyToCover	平掉指定的空头持仓。

交易指令的使用规则归纳如下：

支持三种基本类型的参数定义，支持指定参数默认值；

不支持使用引用参数；



支持六种类型的变量定义，支持指定变量的默认值；

可以访问 Data0-Data49 个数据源的 Bar 数据；

可以访问行情数据、属性数据；

通过 Buy、Sell、SellShort 和 BuyToCover 产生交易动作，也可以使用各种内建平仓指令产生交易动作；

每个交易指令至少包含一个交易动作；

交易指令可以调用所有的用户函数进行计算；

交易指令可以根据设置调用部分的系统函数；

交易指令在执行时，必须要指定相应的数据源和周期，需要调用历史数据的，只有历史数据存在的情况下才能返回正确的值。

示例，交易指令 MACD\_LE，脚本如下：

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

##### Params

```
Numeric FastLength( 12 );  
Numeric SlowLength( 26 );  
Numeric MACDLength( 9 );  
Numeric BuyLots(1);
```

##### Vars

```
NumericSeries MACDValue;  
NumericSeries AvgMACD;  
Numeric MACDDiff;  
Bool Condition1;  
Bool Condition2;
```

##### Begin

```
MACDValue = XAverage( Close, FastLength ) -  
            XAverage( Close, SlowLength ) ;  
AvgMACD = XAverage(MACDValue,MACDLength);  
MACDDiff = MACDValue - AvgMACD;  
Condition1 = CrossOver(MACDValue, AvgMACD) ;  
Condition2 = MACDValue > 0;  
if (Condition1 And Condition2)  
{  
    Buy(BuyLots,Close);  
}
```

##### End

MACD\_LE 在零轴之上,当 MACDValue 向上穿过 AvgMACD 值时为产生多头买入指令。

### 关于 Delay

默认情况下，4 个交易函数产生的委托单即时发送；当参数 Delay=True 时，委托单将延迟到下一个 Bar 发送，这样设计的原因在于：只有延迟的委托单才会保证发送的交易指令的正确性。

假定在某商品 A 的周期为 5 分钟的数据上应用交易指令，A 商品每 1 秒钟会产生一个 Tick 数据，因此一段时间内（<5 分钟）A 商品最后一个 Bar 的数据的收盘价，最高价，最低价以及成交量等数据，会随着 Tick 的变化和累计而产生相应的变化。在某种情况下，上一个 Tick 更新时，Buy 的预设条件可能为 False，下一个 Tick 更新时，Buy 的预设条件为 True。如果不延迟，将会马上发送该委托单到交易所。但是，当更多的 Tick 累计，产生一个新的 Bar 时，Buy 的预设条件可能会变成 False。在这种情况下，前面产生的委托单将会丢失，不会在测试和优化报表中出现。该委托单实际上是由于噪音数据产生的错误讯号导致，为了避免这种情况的出现，一定要等最后 Bar 数据更新结束之后，新 Bar 产生第一个 Tick 时，才会发送上一个 Bar 产生的委托单。

当交易函数的延迟设置为 False。将会实时发送产生的委托单，按 Tick 进行更新。在使用该参数时，需要确认自己所编写的公式不会用到这些无效的中间数据，从而影响交易结果。

**QUOTE:**

注意: Delay 参数对交易会产生重要的影响，请在确认理解含义之后才进行真实的交易。

### 关于内建平仓指令

除了上节的 Sell 和 BuyToCover 可以进行平仓之外，TradeBlazer 公式提供了额外的八种平仓函数，通过合理的应用内建平仓函数，可以帮助您有效的锁定风险并及时获利。

您可以组合使用内建平仓函数，也可以在自己的交易指令中调用内建平仓函数进行平仓，八个内建平仓函数如下：

函数名	描述
<a href="#">SetExitOnClose</a>	该平仓函数用来在当日收盘后产生一个平仓动作，将当前所有的持仓按当日收盘价全部平掉。
<a href="#">SetBreakEven</a>	该平仓函数在获利条件满足的情况下启动，当盈利回落达到保本时产生平仓动作，平掉指定的仓位。
<a href="#">SetStopLoss</a>	该平仓函数在亏损达到设定条件时产生平仓动作，平掉指定的仓位。
<a href="#">SetProfitTarget</a>	该平仓函数在盈利达到设定条件时产生平仓动作，平掉指定的仓位。
<a href="#">SetPeriodTrailing</a>	该平仓函数在盈利回落到设定条件时产生平仓动作，平掉指定的仓位。
<a href="#">SetPercentTrailing</a>	该平仓函数在盈利回落到设定条件时产生平仓动作，平掉指定的仓位。
<a href="#">SetDollarTrailing</a>	该平仓函数在盈利回落到设定条件时产生平仓动作，平掉指定的仓位。
<a href="#">SetInactivate</a>	该平仓函数在设定时间内行情一直在某个幅度内波动时产生平仓动作，平掉指定的仓位。

## 关于 ExitPosition

上述多个平仓函数都用到了参数 ExitPosition，作为平仓函数仓位控制的重要参数，有必要对该参数进行单独说明。

ExitPosition 是布尔型参数，当 ExitPosition=True 时，表示将当前所有的持仓作为一个整体，根据其平均建仓成本，计算各平仓函数的盈亏，当条件满足时，会将所有仓位一起平掉；当 ExitPosition=False 时，表示单独对每个建仓位置进行平仓，单独计算各平仓函数盈亏时，当单个建仓位置条件满足后，平掉该建仓位置即可。

## TradeBlazer 公式入门教程(13)

### Step17

#### 在技术分析中设置报警

公式报警

TradeBlazer 公式提供报警功能，您可以在 3 类技术分析(技术指标，K 线型态，特征走势)中通过 Alert 函数来实现报警。

您可以在这 3 类公式中按照以下方式编写自己的报警：

[Copy to clipboard] [ - ]

```
CODE:
Vars
    Bool Condition1;
Begin
    Condition1 = 您设定的条件表达式;
    If(AlertEnabled AND Condition1)
    {
        Alert("报警信息...");
    }
End
```

当公式编译保存成功之后，您可以将其应用在超级图表中，通过[报警属性页](#)（[帮助--公式系统--TradeBlazer 公式应用--公式属性](#)）启动报警。当条件满足之后，将会产生报警信息，并发送到[消息中心](#)（[帮助--系统基础--消息中心](#)）。

### Step18

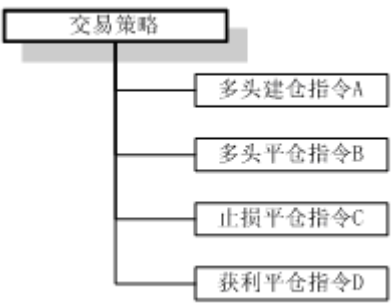
#### 交易策略-完整的交易系统

通常单个交易指令只完成建仓或平仓的单个动作，而一个完整的交易策略应该至少包含建仓、平仓交易指

令，并且根据需要加上止损，获利等锁定风险和收益的交易指令。多个交易指令的组合才能更加有效的帮助我们完整的进行交易，因此，我们将多个交易指令的有效组合称之为交易策略。

### 交易策略的运行机制

假定我们创建一个交易策略，该交易策略由以下交易指令组成，并按照如下顺序应用到超级图表中。



当我们将该交易策略应用到超级图表上时，TradeBlazer 公式将会从图表的第一个 Bar 开始执行交易策略，在第一个 Bar 上首先执行多头建仓指令 A，可能会产生交易委托（开仓），该委托可能被设置为在当前 Bar 执行，也可以被设置为延迟到下一个 Bar 执行。当多头建仓指令 A 执行完成之后，将按顺序调用多头平仓指令 B，同时该指令会判断当前的持仓状态，仓位等信息，当条件满足的时候会产生交易委托（平仓）。依次执行止损平仓指令 C 和获利平仓指令 D，当四个交易指令在第一个 Bar 上都执行完之后，将会移到第二个 Bar 执行，这时候，系统会首先读取上一个 Bar 是否有延迟的交易委托，如果有延迟的交易委托，对这些委托先进行处理，然后像第一个 Bar 一样，依次调用各个交易指令。以此类推，从图表的第一个 Bar 到最后一个 Bar，全部执行完成之后，整个交易策略执行完毕。在整个执行过程产生的所有交易委托被保存下来供超级图表模块显示或进行性能测试分析。

当交易策略应用在超级图表中时，您可以设置交易策略开平仓的显示风格以及颜色、线条等，使之显示在超级图表中，如下图所示：



## 交易策略测试引擎

为了真实准确的模拟交易策略在过去时段的表现，并能在实时数据更新时使交易策略沿着预定的方向发展，TradeBlazer 公式提供了一个强大的交易策略测试引擎，该处理引擎收集交易策略在历史过程中产生的所有委托单，将其应用在对应的图表中，并能根据交易设置创建交易策略性能测试报表供客户参考。

交易策略测试引擎包括了两大功能：历史数据测试和实时自动交易。历史数据测试分析交易策略在历史过程中的交易动作并计算出交易盈亏，收益等性能指数。实时自动交易收集实时数据，并根据实时数据生成相应的交易动作，条件满足时，将委托单直接发送到交易券商。

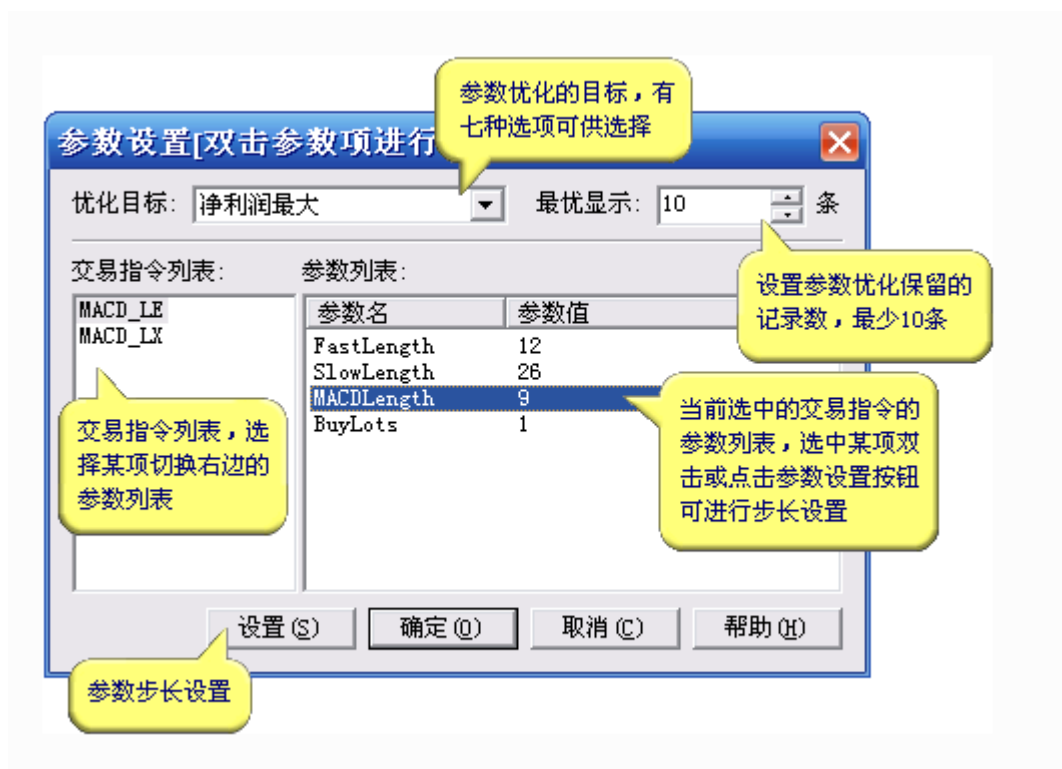
## TradeBlazer 公式入门教程(14)

### Step19

## 交易策略参数优化

在超级图表中插入一个或一个以上交易指令后，菜单和工具栏中的交易策略参数优化选项将会有效，您可以通过点击菜单项或工具栏使用该功能。

交易策略参数优化模块可对多个交易指令组合的所有参数进行优化，您可以通过参数设置界面需要对需要优化的参数进行设置，界面如下：



参数优化目标有以下七种选项：

净利润最大：以优化结果中的净利润最大为目标，保留指定数量的记录数；

交易次数最大：以优化结果中的交易次数最大为目标，保留指定数量的记录数；

平均净利润最大：以优化结果中的平均净利润最大为目标，保留指定数量的记录数；

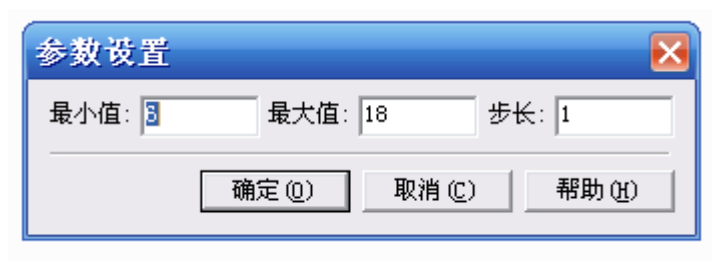
盈利因子最大：以优化结果中的盈利因子最大为目标，保留指定数量的记录数；

收益率最大：以优化结果中的收益率最大为目标，保留指定数量的记录数；

盈亏比率最大：以优化结果中的盈亏比率最大为目标，保留指定数量的记录数；

回报率最大：以优化结果中的回报率最大为目标，保留指定数量的记录数。

通过双击参数项或点击参数设置按钮，对选中的参数项进行步长设置，界面如下：



您可以设置参数的最小值，步长及最大值，系统将会根据设置，产生从最小值到最大值之间按步长分布的参数列表。

在各项参数设置完成之后，点击确定按钮，将会进行参数优化的计算，参数优化计算过程中会显示优化的进程，时间，当前参数，最优参数以及优化目标等信息，您可以通过点击取消按钮终止计算。

**寻找最优的参数。需防止过度优化。**

**寻找最稳定的系统。而不是最大化的系统。**

**QUOTE:**

注意：根据优化参数设置的多少，优化时间可能长达几小时甚至数天，优化过程中，CPU 将会大量被占用，您可以选择空闲的时间进行大量参数优化计算。

参数优化计算完成之后，将会显示出最优的记录，您可以通过工具栏的交易设置按钮修改交易参数，也可以通过重新优化按钮进行重新计算。

## Step20

### 交易策略性能测试

在超级图表中插入一个或一个以上交易指令后，菜单和工具栏中的交易策略测试报表选项将会有效，您可以通过点击菜单项或工具栏使用该功能。

交易策略测试报表界面如下：

性能概要			
统计指标	全部交易	多头	空头
净利润	2470.00	2470.00	0.00
总盈利	2940.00	2940.00	0.00
总亏损	(470.00)	(470.00)	0.00
总盈利/总亏损	6.26	6.26	0.00
期末持仓盈亏	0.00	0.00	0.00
交易次数	11	11	0
盈利比率	63.64%	63.64%	0.00%
盈利次数	7	7	0
亏损次数	4	4	0
持平次数	0	0	0
平均利润	224.55	224.55	0.00
平均盈利	420.00	420.00	0.00
平均亏损	(117.50)	(117.50)	0.00
平均盈利/平均亏损	3.57	3.57	0.00

帐户分析按照五个方面对帐户进行分析，包括交易汇总、交易分析、交易记录、平仓分析、阶段总结、资产变化、图表分析和系统设置。

交易汇总：按照多头交易、空头交易和全部交易列出当前交易策略的交易统计信息。

交易分析：对当前策略的交易情况进行分析，包括交易分析、盈亏分析和连续盈亏分析。

交易记录：按开仓平仓对所有交易进行配对组合，并计算盈亏及累计盈亏。

平仓分析：按平仓记录对交易情况进行分析和汇总。

阶段总结：按年、月对交易盈亏及次数进行统计。

资产变化：列出资产的变化记录及统计信息。

图表分析：按资产图表和盈亏图表两大类共十三小类对帐户进行图表分析。

系统设置：显示交易策略的参数，设置以及数据等内容。

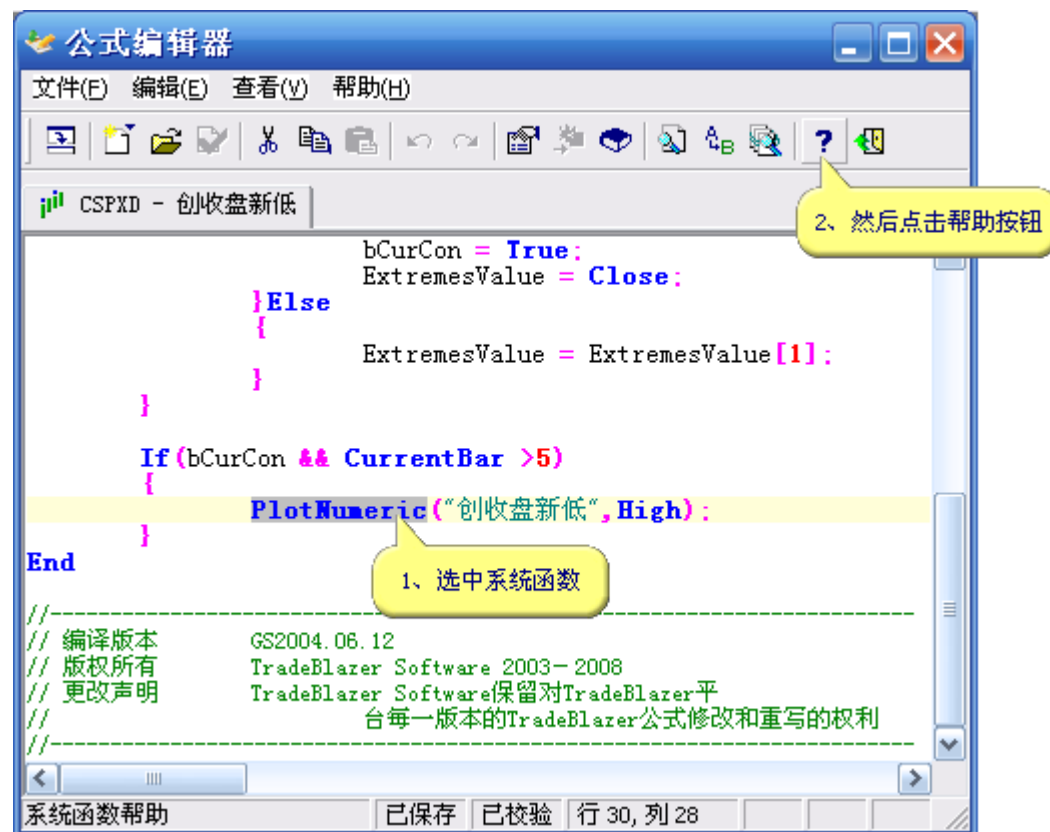


## TB 公式之常见问题(Q&A)!

Q1: 如何在公式编辑器中快速显示系统函数帮助?

**A1:**

- 1、在公式编辑器中，用鼠标选中系统函数；
- 2、点击工具栏的帮助按钮，将会打开对应的帮助文件。



Q2: 我自己建的公式没有编译，但在公式编辑器中不能编译啊，编译按钮是灰的，我该怎么办？

**A2:**

您只需要在公式代码中任意位置加入一个空格，然后再删掉这个空格，就可以进行编译了。  
这是因为该公式以前被保存过，如果代码没有被修改过，就不能触发进行保存编译。

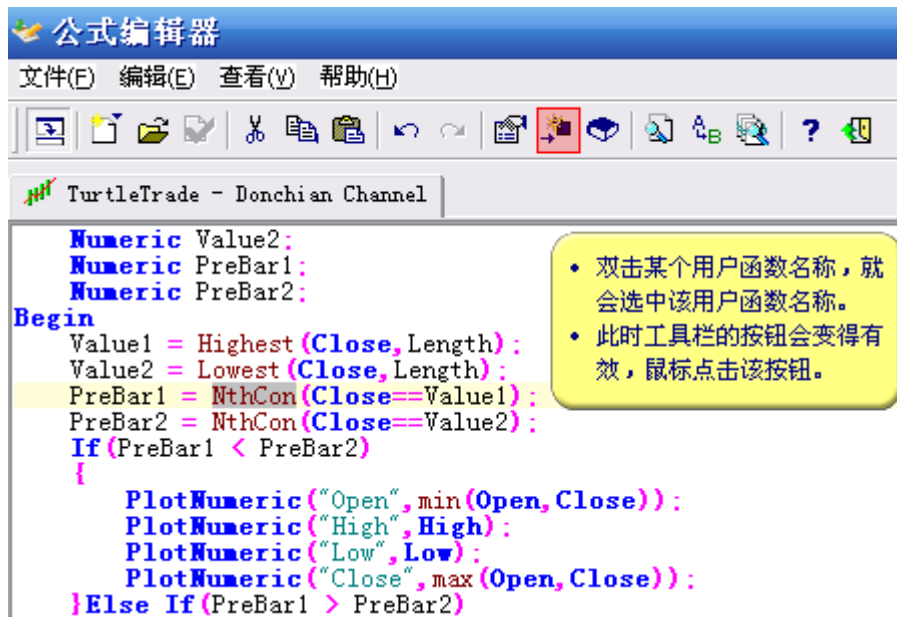
Q3:保存公式的时候，需要时间很长是为什么？

**A3:**

保存公式的时候，需要做大量的编译工作，所以会比较慢，请耐心等待几秒钟。

Q4:如何快捷的在公式编辑器中跳转到调用的用户函数中？

**A4:**鼠标双击函数名，选中，点工具栏的一个按钮，就可以转到用户函数的代码中。



Q5:我的公式死活编译不过，还没有任何提示？

**A5: 有两种情况可能导致这样的问题：**

- 1、应该是公式的简称里面有中文，TB公式不支持中文的函数或公式名称，所以您只能删掉这个公式，重新新建一个，不要用包含中文的名字。
- 2、您的公式里面可能有中文的符号，或者其它不可显示的字符，您需要仔细检查一下代码。也可以采取一行一行加注释的方式来排除问题。

Q6:总是报“锁定编译目标文件超时”是什么原因？

**A6:**

有两种可能。

- 1、已经打开的图表调用了技术指标或交易指令，并且行情更新较快，导致编译时覆盖旧文件失败。这个时候，您可以关闭先所有的图表窗体在试试看。
- 2、可能是公式的写法有问题，是系统现在还不能识别的错误。您可以另外写一个简单的公式看看能不能编译通过，如果能通过，那就证明是这个公式有问题。如果不是，那我也不知道具体原因：(。您可以考虑导出您自己的公式，备份自己的工作区，然后删掉User目录重新登录。

Q7:我新建的用户函数编译提示：Return语句的返回值类型与公式定义的返回值类型不符。我该怎么办？

**A7:**

您只需要在公式编辑器中打开属性设置对话框，在最后一页[返回类型]中将返回之类型设置为正确的类型。  
如果您希望函数返回一个数字，那您可以选择数值型。  
如果您希望函数返回一个条件，那您可以选择布尔型。  
如果您希望函数返回一个字符串，那您可以选择字符串。

Q8:我建了一个指标，怎样让它在副图上显示，在公式属性里设置，我看到有“主图”和“子图”两个选项，但是是灰色的，不能用

**A8:**

- 1、您可以在图表中直接选中该指标，把它拖到图表下面的时间横坐标轴的位置，然后放开鼠标。这只是暂时解决显示的问题，指标的默认还是主图显示。
- 2、您可以在公式管理器中，选中技术指标，点属性按钮，在默认页面，设置主图，子图。
- 3、您也可以可以在技术指标的公式编辑器中，点属性按钮，进行设置。

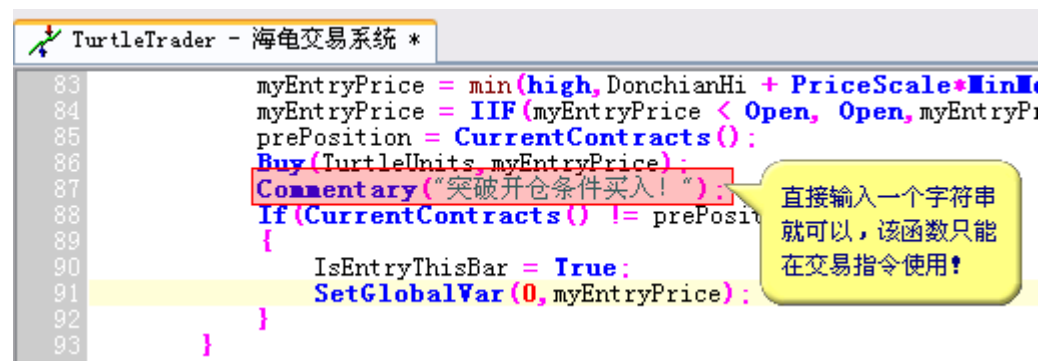
以下是相关的内容：

<http://www.tradeblazer.net/forum/thread-719-1-3.html>

**Q9:如何用Commentary进行调试?**

**A9:**

- 1、代码里面这么写：



The screenshot shows the TurtleTrader software interface. The title bar reads "TurtleTrader - 海龟交易系统 \*". The code editor displays the following code:

```
83 myEntryPrice = min(high, DonchianHi + PriceScale*MinMo
84 myEntryPrice = IIF(myEntryPrice < Open, Open, myEntryP
85 prePosition = CurrentContracts();
86 Buy(TurtleUnits, myEntryPrice);
87 Commentary("突破开仓条件买入!");
88 If (CurrentContracts() != prePosition)
89 {
90     IsEntryThisBar = True;
91     SetGlobalVar(0, myEntryPrice);
92 }
93 }
```

A yellow callout box points to the `Commentary` function, containing the text: "直接输入一个字符串就可以，该函数只能在交易指令使用！"

- 2、在图表里面双击鼠标左键：



**Q10:为什么出现最终目标文件编译错误?**

#### A10:

目前发现有以下几种情况会导致这个问题出现:

- 1、整体公式比较多, 累计到一定程度, 就会出现这个问题, 对于这种情况, 暂时没法解决, 只好先把不用的公式备份出来, 然后删掉一些公式。
- 2、用了一些C++的关键字来命名变量, 比如.switch,case,int,Public,protected,class,long,double....有好几百个, 可以考虑加上一些前缀, 比如My\*\*\*\*, 这样就可以了。

**Q1:关于全局变量的处理, 能否举例通俗的解释一下 SetGlobalVar 和 GetGlobalVar?**

#### A1:

系统公式目前提供 50 个全局变量。这 50 个全局变量附着在超级图表上, 即一个图表的各个交易指令可以有 50 个全局变量。

您可以通过这些全局变量在交易指令中进行数据交换。关掉超级图表之后及全部删除, 新建一个超级图表, 新建出 50 个初值为无效值的变量。用户自行通过 GetGlobalVar,SetGlobalVar 进行保存及获取数据的操作。全局变量不会因为当前 Bar 的变化而变化。

每一个图表的单个技术指标, K 线型态, 特征走势都有 50 个全局变量。一个图表的所有交易指令共有 50 个全局变量。

**Q:如何处理讯号出现又消失的情况?**

#### A:有以下两种方式:

1、使用 Buy(1,Close,True)这样的格式, 将信号延迟到该 Bar 走完, 下一个 Bar 的第一个 Tick 出现的时候发送。

这个时候您可以用 Buy(1,Close,True), 表示的意思是用当前 Bar 的收盘价在下一个 Bar 开始时候交易。

您也可以使用 Buy(1,NextOpen,True), 表示用下一个 Bar 的开盘价交易。

对于上面的这种处理方式, 还有另外一种变通的处理, 那就是取上一个 Bar 的条件或数据, 只用当前 Bar 的 Open 价来进行判断。这样公式会写得更清晰。以下两段代码效果是相同的。

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```
Condition = 您的交易条件;  
If(Condition)  
{  
    Buy(1,NextOpen,True);  
}
```

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```
Condition = 您的交易条件;  
If(Condition[1])  
{  
    Buy(1,Open);  
}
```

2、第一种处理方式对于时间敏感性不高的系统，是可以采取的，但有些系统，如果选择延迟发送，则会导致比较大的性能下降。

此时我们需要选择另外一种处理方式。使用 **High,Low,Open** 这样能够保持住的价格来进行条件判断。

是价格往上的突破形成的交易操作用 **High** 来判断。是价格下上的突破形成的交易操作用 **Low** 来判断。其他不确定方向的情况最好用 **Open** 来判断。

比如，以下两种情况在实时交易方面同样迅速，但后面的就不会出现讯号消失的问题。

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```
AvgClose = AverageFC(Close,10);
If(CrossOver(Close,AvgClose))
{
    Buy(1,AvgClose+MinMove*PriceScale);
}
```

[\[Copy to clipboard\]](#) [\[ - \]](#)

#### CODE:

```
AvgClose = AverageFC(Close,10);
If(CrossOver(High,AvgClose))
{
    Buy(1,AvgClose+MinMove*PriceScale);
}
```

#### 附海龟 TB 源码: Params

```
Numeric RiskRatio(1);           // % Risk Per N ( 0 - 100)
Numeric ATRLength(20);           // 平均波动周期 ATR Length
Numeric boLength(20);            // 短周期 BreakOut Length
Numeric fsLength(55);            // 长周期 FailSafe Length
Numeric teLength(10);            // 离市周期 Trailing Exit Length
Bool LastProfitableTradeFilter(True); // 使用入市过滤条件
```

#### Vars

```
Numeric MinPoint;                // 最小变动单位
Numeric N;                       // N 值
Numeric TotalEquity;              // 按最新收盘价计算出的总资产
Numeric TurtleUnits;              // 交易单位
NumericSeries DonchianHi;         // 唐奇安通道上轨，延后 1 个 Bar
NumericSeries DonchianLo;         // 唐奇安通道下轨，延后 1 个 Bar
NumericSeries fsDonchianHi;       // 唐奇安通道上轨，延后 1 个 Bar，长周期
NumericSeries fsDonchianLo;       // 唐奇安通道下轨，延后 1 个 Bar，长周期
Numeric ExitHighestPrice;         // 离市时判断需要的 N 周期最高价
Numeric ExitLowestPrice;          // 离市时判断需要的 N 周期最低价
Numeric myEntryPrice;             // 开仓价格
```

```

Numeric myExitPrice;           // 平仓价格
Bool SendOrderThisBar(False);  // 当前 Bar 有过交易
    NumericSeries preEntryPrice(0);    // 前一次开仓的价格，存放到全局变量 0 号位置
    BoolSeries PreBreakoutFailure(false); // 前一次突破是否失败
Begin
    If(BarStatus == 0)
    {
        preEntryPrice = InvalidNumeric;
        PreBreakoutFailure = false;
    }Else
    {
        preEntryPrice = preEntryPrice[1];
        PreBreakoutFailure = PreBreakoutFailure[1];
    }

    MinPoint = MinMove*PriceScale;
    N = AverageFC(TrueRange,ATRLength);

    TotalEquity = CurrentCapital() +
Abs(CurrentContracts()*Close*ContractUnit()*BigPointValue()*MarginRatio());
    TurtleUnits = (TotalEquity*RiskRatio/100) /(N * ContractUnit()*BigPointValue());
    TurtleUnits = IntPart(TurtleUnits); // 对小数取整

    DonchianHi = HighestFC(High[1],boLength);
    DonchianLo = LowestFC(Low[1],boLength);

    fsDonchianHi = HighestFC(High[1],fsLength);
    fsDonchianLo = LowestFC(Low[1],fsLength);

    Commentary("N="+Text(N));
    Commentary("preEntryPrice="+Text(preEntryPrice));
    Commentary("PreBreakoutFailure="+IIFString(PreBreakoutFailure,"True","False"));

    // 当不使用过滤条件，或者使用过滤条件并且条件为 PreBreakoutFailure 为 True 进行后续操作
    If(MarketPosition == 0 && ((!LastProfitableTradeFilter) Or (PreBreakoutFailure)))
    {
        // 突破开仓
        If(CrossOver(High,DonchianHi) && TurtleUnits >= 1)
        {
            // 开仓价格取突破上轨+一个价位和最高价之间的较小值，这样能更接近真实情况，并能尽量保
            证成交
            myEntryPrice = min(high,DonchianHi + MinPoint);
            myEntryPrice = IIF(myEntryPrice < Open, Open,myEntryPrice); // 大跳空的时候用开盘价代替
            preEntryPrice = myEntryPrice;
            Buy(TurtleUnits,myEntryPrice);

```

```

        SendOrderThisBar = True;
        PreBreakoutFailure = False;
    }

    If(CrossUnder(Low,DonchianLo) && TurtleUnits >= 1)
    {
        // 开仓价格取突破下轨-一个价位和最低价之间的较大值，这样能更接近真实情况，并能尽量保
证成交
        myEntryPrice = max(low,DonchianLo - MinPoint);
        myEntryPrice = IIF(myEntryPrice > Open, Open,myEntryPrice); // 大跳空的时候用开盘价代替
        preEntryPrice = myEntryPrice;
        SendOrderThisBar = True;
        SellShort(TurtleUnits,myEntryPrice);
        SendOrderThisBar = True;
        PreBreakoutFailure = False;
    }
}

// 长周期突破开仓 Failsafe Breakout point
If(MarketPosition == 0)
{
    If(CrossOver(High,fsDonchianHi) && TurtleUnits >= 1)
    {
        // 开仓价格取突破上轨+一个价位和最高价之间的较小值，这样能更接近真实情况，并能尽量保
证成交
        myEntryPrice = min(high,fsDonchianHi + MinPoint);
        myEntryPrice = IIF(myEntryPrice < Open, Open,myEntryPrice); // 大跳空的时候用开盘价代替
        preEntryPrice = myEntryPrice;
        Buy(TurtleUnits,myEntryPrice);
        SendOrderThisBar = True;
        PreBreakoutFailure = False;
    }

    If(CrossUnder(Low,fsDonchianLo) && TurtleUnits >= 1)
    {
        // 开仓价格取突破下轨-一个价位和最低价之间的较大值，这样能更接近真实情况，并能尽量保
证成交
        myEntryPrice = max(low,fsDonchianLo - MinPoint);
        myEntryPrice = IIF(myEntryPrice > Open, Open,myEntryPrice); // 大跳空的时候用开盘价代替
        preEntryPrice = myEntryPrice;
        SellShort(TurtleUnits,myEntryPrice);
        SendOrderThisBar = True;
        PreBreakoutFailure = False;
    }
}

```

```

}

If(MarketPosition == 1) // 有多仓的情况
{
    // 求出持多仓时离市的条件比较值
    ExitLowestPrice = Lowest(Low[1],teLength);
    Commentary("ExitLowestPrice="+Text(ExitLowestPrice));
    If(Low < ExitLowestPrice)
    {
        myExitPrice = max(Low,ExitLowestPrice - MinPoint);
        myExitPrice = IIF(myExitPrice > Open, Open,myExitPrice ); // 大跳空的时候用开
盘价代替
        Sell(0,myExitPrice); // 数量用 0 的情况下将全部平仓
    }Else
    {
        If(preEntryPrice!=InvalidNumeric && TurtleUnits >= 1)
        {
            If(Open >= preEntryPrice + 0.5*N) // 如果开盘就超过设定的 1/2N,则直接用开盘价增仓。
            {
                myEntryPrice = Open;
                preEntryPrice = myEntryPrice;
                Buy(TurtleUnits,myEntryPrice);
                SendOrderThisBar = True;
            }

            while(High >= preEntryPrice + 0.5*N) // 以最高价为标准，判断能进行几次增仓
            {
                myEntryPrice = preEntryPrice + 0.5 * N;
                preEntryPrice = myEntryPrice;
                Buy(TurtleUnits,myEntryPrice);
                SendOrderThisBar = True;
            }
        }
    }

    // 止损指令
    If(Low <= preEntryPrice - 2 * N && SendOrderThisBar == false) // 加仓 Bar 不止
    损
    {
        myExitPrice = preEntryPrice - 2 * N;
        Sell(0,myExitPrice); // 数量用 0 的情况下将全部平仓
        PreBreakoutFailure = True;
    }
}

}Else If(MarketPosition ==-1) // 有空仓的情况

```



```

{
    // 求出持空仓时离市的条件比较值
    ExitHighestPrice = Highest(High[1],teLength);
    Commentary("ExitHighestPrice="+Text(ExitHighestPrice));
    If(High > ExitHighestPrice)
    {
        myExitPrice = Min(High,ExitHighestPrice + MinPoint);
        myExitPrice = IIF(myExitPrice < Open, Open,myExitPrice ); // 大跳空的时候用开盘价代替
        BuyToCover(0,myExitPrice);    // 数量用 0 的情况下将全部平仓
    }Else
    {
        If(preEntryPrice!=InvalidNumeric && TurtleUnits >= 1)
        {
            If(Open <= preEntryPrice - 0.5*N) // 如果开盘就超过设定的 1/2N,则直接用开盘价增仓。
            {
                myEntryPrice = Open;
                preEntryPrice = myEntryPrice;
                SellShort(TurtleUnits,myEntryPrice);
                SendOrderThisBar = True;
            }

            while(Low <= preEntryPrice - 0.5*N) // 以最低价为标准，判断能进行几次增仓
            {
                myEntryPrice = preEntryPrice - 0.5 * N;
                preEntryPrice = myEntryPrice;
                SellShort(TurtleUnits,myEntryPrice);
                SendOrderThisBar = True;
            }
        }
    }

    // 止损指令
    If(High >= preEntryPrice + 2 * N &&SendOrderThisBar==false) // 加仓 Bar 不止损
    {
        myExitPrice = preEntryPrice + 2 * N;
        BuyToCover(0,myExitPrice); // 数量用 0 的情况下将全部平仓
        PreBreakoutFailure = True;
    }
}
}
End

```