

交易开拓者（TradeBlazer）公式详细介绍

概述

本章节内容是 TradeBlazer 公式的全面参考手册，详细介绍了 TradeBlazer 公式的结构、语法、特点、使用方法及功能等。

通过阅读该参考手册，您能够了解 TradeBlazer 公式的基本语法、操作符、表达式及控制语句等，通过手册提供的各种示例程序，掌握各种 TradeBlazer 公式的编写要领，最终达到能够熟练将自己的思想转化为 TradeBlazer 公式，并在交易开拓者中应用。

什么是 TradeBlazer 公式？

TradeBlazer 公式是一种专为分析金融数据-时间序列而设计的高级语言，它提供直接、强大的框架将交易思想转化为用户函数、用户字段、技术分析，交易指令等计算机能够识别的代码。

TradeBlazer 公式是一门语法简单但是功能强大的语言，它能帮助您创建自己的交易和技术分析工具。通过组合普通的交易指令和简单的语句，TradeBlazer 公式使您能够很容易并且直接的用简单语句表达自己的交易规则和行为。

交易开拓者能够读取您开发的 TradeBlazer 公式，在历史价格数据基础上进行评估，并能自动执行特定的交易动作，将您的交易思想转化为实际的交易操作。

TradeBlazer 公式能做什么？

通过 TradeBlazer 公式，您能够创建自己的交易指令、技术指标、K 线型态、特征走势、用户函数以及用户字段。您也可以拷贝，修改并使用系统内置几百个函数、字段、技术分析和交易指令。

TradeBlazer 公式包含的公式类型如下：

- **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。

各类数据

Bar 数据

在介绍 Bar 数据之前，首先，我们需要讨论一下 TradeBlazer 公式的计算方法，针对上面介绍的各种公式类型，包含用户函数，技术分析，交易指令等，在公式进行计算时，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。

以下列出所有的 Bar 数据系统函数：

函数名	简写	描述
Date	D	当前 Bar 的日期。
Time	T	当前 Bar 的时间。
Open	O	当前 Bar 的开盘价。
High	H	当前 Bar 的最高价。
Low	L	当前 Bar 的最低价。
Close	C	当前 Bar 的收盘价。
Vol	V	当前 Bar 的成交量。
OpenInt	无	当前 Bar 的持仓量。
CurrentBar	无	当前 Bar 的索引值，从 0 开始计数。
BarStatus	无	当前 Bar 的状态值，0 表示为第一个 Bar，1 表示为中间的普通 Bar，2 表示最后一个 Bar。

计算方法

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。如下图箭头所示，公式执行从上至下，Bar 从左到右执行。



例如，现在有如下语句需要执行，Bar 数据如下表所示：

Value1 = Close - Open;

CurrentBar	Date	Time	Open	High	Low	Close	Vol	Value1
0	2005/04/04	15:00	2970	2979	2951	2974	18	4
1	2005/04/05	15:00	2960	2960	2946	2960	14	0
2	2005/04/06	15:00	2951	2980	2951	2963	30	12
3	2005/04/07	15:00	3048	3048	2968	2995	120	-57
4	2005/04/08	15:00	2985	2987	2985	2987	10	2
...

如上表所示，从 CurrentBar = 0 开始，依次计算每个 Bar 进行计算；

在公式的编写中，经常会遇到当前 Bar 的数据和上一个 Bar，上 N 个 Bar 数据进行比较，计算的情况，针对这种情况，TradeBlazer 公式提供了一种处理机制：回溯。即对数据的向前引用，比如，获取上一个 Bar 的收盘价：Close[1]，获取 10 天前的成交量：Vol[10]。以下提供一个简单的例子来说明如何进行回溯处理。

假定有如下语句：

```
If (Close > Close[1])
{
    Buy(1,Close);
}
```

以上公式执行一个简单的操作，当前 Bar 的收盘价大于上一个 Bar 的收盘价，即执行按照当前收盘价买入 1 手的动作。根据上表的数据，公式将在 CurrentBar 为 2 和 3 的时候调用 Buy 指令。

如果您足够仔细的话，您会发现：对于上面的一段公式的执行，有一个小小的问题，当第一次计算公式时，即 CurrentBar = 0 时，这个时候需要获取上一个 Bar 的数据，但是当前 Bar 已经是第一个 Bar，这个时候就存在着问题，如何来获取此时的 Close[1]呢，TradeBlazer 公式将默认 Close[1]为无效值，即系统函数中的 InvalidNumeric，Close > Close[1]的表达式计算结果是一个 Bool 值，其结果也是一个无效值，对于 Bool 值，我们将 False 作为其无效值。因此，第一个 Bar 计算时，Buy 指令不会被执行。

对于技术分析这类公式来说，假定 Bar 数据的总数共有 100，相同的代码将从 CurrentBar = 0 到 CurrentBar = 99 共执行 100 遍，分别输出公式中的结果值。

注意：在执行 TradeBlazer 公式时，可能出现数据不存在的情况，可通过系统函数 HistoryDataExist 进行判断，如果该函数返回 True，即为 Bar 数据有效。

叠加数据

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。

假定，我们新建一个超级图表模块，其主数据对应的商品为：cu0503，在此基础上，我们叠加了 cu0504 和 cu0505。此时，根据叠加操作的先后顺序，cu0503 为 Data0，cu0504 为 Data1，cu0505 为 Data2，在 TradeBlazer 公式中，我们可以通过 Data1.Close()，Data2.Vol()类似方法调用叠加 Bar 数据，叠加 Bar 数据的函数和 Bar 数据一样，只是需要在调用的时候加上数据源。

我们也可以使用 Data0.Open()来调用 Bar 数据，默认情况下，可以省略对主数据源的指定，为了方便，一般直接使用 Open()来代替 Data0.Open()。

行情数据

除了 Bar 数据之外，TradeBlazer 公式还可以支持实时行情数据的调用，行情数据是指当前商品最新的报价数据，该数据和 Bar 无关，行情数据的回溯没有意义。

行情数据只在最后 Bar 是有意义的，其他 Bar 会返回无效值。因此，在调用行情数据函数时，为了提高效率，最好按照以下方法：

```
If(BarStatus()==2)
{
    //调用行情数据函数
}
```

行情数据函数都按照以下格式命名 Q_XXXXX，比如 Q_Close，Q_BidPrice。在调用行情数据的时候，需要判断当前行情数据是否有效，系统提供函数 QuoteDataExist 来对有效性进行判断。如果行情数据已经准备好，返回 True，否则，返回 False。

属性数据

除了以上的各项数据之外，TradeBlazer 公式还提供一组重要的属性数据，反映了该商品的一些基本信息，比如当前数据周期，买卖盘个数、保证金设置等信息。在所有的 Bar 上面获取的市场属性数据都是一样的，属性数据的回溯没有意义。

关于属性数据的详细说明参见帮助文件附录。

数据类型

TradeBlazer 公式支持有三种基本数据类型：数值型、字符串、布尔型。

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，如下表所示：

名称	说明
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。

命名规则

公式名称规则：

- 不区分大小写；
- 不能超过 32 个英文字符；
- 每一类公式不能出现相同的名称；
- 公式名称不能出现字母、数字、下划线以外的其他字符；
- 公式名称不能和系统保留字，系统函数等重名。

变量，参数规则：

- 不区分大小写；
- 不能超过 32 个英文字符；
- 每一个公式内部能不能重复命名；
- 名称不能出现字母、数字、下划线以外的其他字符；
- 名称不能和系统保留字，系统函数等重名；
- 不能使用已定义的用户函数名。

注意：建议采取匈牙利命名规则命名变量，匈牙利命名规则是 **Charles Simonyi** 发明的一种给变量取名字的方式。他 在变量前加上变量的类型，这样，看看变量的名字就知道变量的类型了。

语句

一个语句代表一个完全的指示或描述，语句中包含有保留字、操作符、符号。并且语句总是以";"作为语句结束的标志。

以下为语句的一些例子：

1. **This is one statement;**
2. **This is another statement;**
3. **This is
one statement;**
4. **This is
another
statement**
;
5. **This is yet another;**
6. **This is one statement;This is another;**

赋值语句

赋值语句用于给公式变量指定一个具体的值的语句，赋值语句使用赋值操作符(=)进行处理。

以下为赋值语句的一些例子：

```
Vars
    Bool b;
Begin
    B = true;
    ...
End
Vars
    Numeric Value1;
Begin
    Value1 = (Close + Open)/2;
    ...
End
Vars
    String str;
Begin
    str ="It Is A Test!";
    ...
End
```

变量在赋值的时候忽略其扩展数据类型，只考虑其基本数据类型，即 `NumericSeries`，`NumericRef`，`Numeric` 之间可以相互赋值。此时序列数据类型只是对当前 `Bar` 的值进行操作。

保留字

保留字都有自己独特的意思或用途，主要是一些功能关键字，系统函数，以及数据类型等。

下面分类列举出系统主要的保留字。

数据类型

包含 3 种基本类型，共 9 种数据类型能够，详细说明参见[数据类型](#)。

运算符

类型	保留字
算术运算符	+ - * / % ^
关系运算符	> >= < <= == <>
逻辑运算符	AND/∧ OR/∨ NOT/!
括号	(){}[]

其它	· ,
----	-----

功能关键字

保留字	说明
Params	用该关键字宣告参数定义的起始，参数必须填写默认值。
Vars	用该关键字宣告变量定义的起始(可以赋初值)，变量不填写初值时,系统将自动为其填充初值。
If	条件语句。
Else	条件语句。
Begin	用该关键字宣告程序主体的起始。
End	用该关键字宣告程序主体的结束。
For	循环语句。
To	循环语句。
DownTo	循环语句。
While	循环语句。
Break	循环语句。
Continue	循环语句。
True	真。
False	假。

数据源

保留字	说明
Data0-Data49	支持 50 个数据源。

数据输出

保留字	说明
FieldBool	用户字段的布尔型返回数据。
FieldNumeric	用户字段的数值型返回数据。
FieldString	用户字段的字符串返回数据。

PlotBool	输出布尔型值。
PlotNumeric	输出数值型值。
PlotString	输出字符串值。
PlotBar	画 K 线型态。
UnPlot	取消指定位置的输出。
Alert	报警输出。
Buy	买入操作。
Sell	卖出操作。
...	其他系统函数。

操作符

操作符是一些象征具体操作运算行为的符号，例如操作符"+"代表对两个数求和，这些操作符适用于数值型、字符串、布尔型的数据。

TradeBlazer 公式为您提供了多种操作运算符，便于您对保留字的操作和生成更复杂的数据类型、逻辑型、字符串类型的值。下面有四种不同类型的操作符可用于逻辑表达式、数值表达式、字符串表达式中。

数学操作符

数值型表达式的操作符有几种，如下表所示：

操作符	说明
+	加
-	减
*	乘
/	除
%	求模
^	求幂
()	括号

这些数学操作按其特定的优先级来进行计算，"^"(求幂)最先，其次是"*(乘法)，"/"(除法)和"%(求模)，加和减最后，如果有多个乘法/除发(或者是加法或减法)，那么计算顺序是从左边到右边。

例如，在数值型的表达式中：

```
High+2*range/2;
```

它首先计算的是 **range**(此处 **range** 是指 **High-Low**)与 2 的积,接着计算与 2 的商(除法),最后求 $2 * \text{range} / 2$ 与最高价(**High**)的和。

如果要找到一个 **Bar** 的中间位置,可以尝试写成如下语句:

```
High+Low/2;
```

然而在上面语句中,首先运算的是以最低价(**Low**)除以 2,然后再与最高价求和。最后的值不是我们所需要的,并且和原来预想中的值是不一致的。

为了处理上述这样的情况,我们在运算符中引入了括号"**()**",可以用括号来操作和控制运算的规则,先计算括号里面的表达式,不考虑外面的操作符和常量。因此,获取某一个 **Bar** 上的中间位置(**MidPoint**)的语句可如下:

```
(High+Low)/2;
```

该语句就是返回最高价和最低价之和的 $1/2$,即 **Bar** 的中间位置。

对于除法,有一些特别的提示,众所周知,0 不能作为除数,否则将会导致系统溢出。**TradeBlazer** 公式在对脚本进行执行时,碰到除法符号时,都需要对除数进行是否为 0 的检查,以保证脚本能够正确的执行下去,当遇到除数为 0 的情况下,我们将会返回一个无效值。

上面描述到求中间位置(**MidPoint**)的表达式,其实我们可以用另外一种形式来代替它:

```
(High+Low)*0.5;
```

对于 **A/B** 这样的表达式,因为不知道 **B** 是否为 0,所以总是需要对其进行有效性验证,计算的速度会受到一定的影响,因此,对于除法表达式,我们强烈建议使用时尽可能转换为乘法处理,即提高执行速度,也可以避免未知的错误发生。

字符串操作符

"+"是唯一的可以应用于控制字符串表达式的数学操作符,它可用于连接两个文本字符串为一个字符串。如下:

```
"这是一个字符串表达式, "+"返回值为数值型。";  
"This is expression A"+" and this is expression B.";
```

上面字符串表达式的输出值,应该是"这是一个字符串表达式,返回值为数值型。"和"This is expression A and this is expression B."。

关系操作符

逻辑运算符使用下列标准的比较符号,大于、小于、等于、小于等于、大于等于和不等。

下列的关系操作符号都可以应用到逻辑表达式中。

操作符	说明
<	小于
>	大于
<=	小于等于
>=	大于等于
<>	不等于
==	等于

应用上述的关系运算符，我们可以对两个数值或字符串表达式进行对比，在下列的语句中，我们就是找到一个 Bar，它的前一个 Bar 收盘价要高于前一个 Bar 最高价：

```
Close>High[1];
```

在字符串的比较运算中，首先是把每一个字符用它的 ASCII 来代替，其次对两个表达式中的字符逐一比较其 ASCII 值，从第一个开始，直到两个表达式中的所有字符已经被计算完为止。例如：

```
"abcd" < "zyxw";
```

在这个例子中，我们对把第一个字符串表达式中的字符和第二个表达式中的字符进行比较运算，字母"a"的 ASCII 值是小于"z"的，同样其它的字符也是一样，所有该表达式的值为 True。

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(!)。

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

在上面的表格中，应用 OR 可以增加表达式的值为 True 的可能性，仅仅只要两个表达式中，只要有一个的值为 True，那么整个表达式的值就为 True。

其实在应用的过程中，还包含有一些复杂的组合运算。为了获得一个的关键反转 Bar，可以使用如下的表达式：

```
Low < Low[1] AND Close > High[1];
```

在上面的表达式中，我们使用了 AND 逻辑运算符，因而要表达式的值为 True，那么当前 Bar 的最低价一定要小于前一个 Bar 的最低价，而且当前 Bar 的收盘价还必须高于前一个 Bar 的最高价。只有当这两个条件都满足的时候，表达式的值才为 True。

再看下面一个例子：

```
High > 10 OR Vol > 5000;
```

在上面的表达式中，如果要其值为 True，那么只需要任意一个条件满足即值为 True，那么表达式的值便为 True，如果当前 bar 的最高价大于 10，或者成交量大于 5000，那么表达式的值便为 True。而如果需要表达式的值为 False 时，则两个条件都必须为 False，表达式的值才为 False。

逻辑操作符的优先级低于数学操作符和关系操作符。逻辑操作符也遵循先括号的原则，如果没有括号，那么其运算顺序也是从左边到右边。

对于逻辑表达式中不同条件的先后顺序，可能会产生不同的运算逻辑，执行的效率也会有所不同。

以 Con1 AND Con2 这样的表达式举例，系统从左到右进行逻辑判断，当 Con1 为 True 时，需要继续判断 Con2 是否为 True，只有当 Con1，Con2 都为 True 时，整个表达式才为 True。但是只要当 Con1 为 False 时，就不再需要判断 Con2 的值，而是直接返回 False。

因此，以下的两个表达式在执行效率方面是有差异的：

```
5 < 4 AND Close > Open;
Close > Open AND 5 < 4;
```

第一条语句的执行速度大部分情况下都比第二条要快。

对于 Con1 OR Con2 表达式，情况也比较类似，当 Con1 为 False 时，需要继续判断 Con2 是否为 False，只有当 Con1，Con2 都为 False 时，整个表达式才为 False。但是只要当 Con1 为 True 时，就不再需要判断 Con2 的值，而是直接返回 True。



以下两条语句的执行效率也是不一样的：

```
5 > 4 OR Close > Open;  
Close > Open OR 5 > 4;
```

通过上述的说明，我们应该知道，逻辑表达式的组合时，应该尽可能的把容易判别整个表达式逻辑的条件放在前面，以减少整个表达式的计算时间。

表达式

表达式是操作符和保留字的有机组合，任意表达式都代表了一个值，表达式的值只能是以下的三种类型：

- **数值型**：即 Numeric, NumericRef, NumericSeries 三种数据类型。
- **布尔型**：即 Bool, BoolRef, BoolSeries 三种数据类型。
- **字符串**：即 String, StringRef, StringSeries 三种数据类型。

当您使用 TradeBlazer 公式的过程中，便可应用三种广泛数据类型的表达式，来完成您自己的程序。

数值型的表达式可以是一个数字，当然，他们也可以是一个数值型的保留字，例如：保留字"Close"。或者是通过运算符组合的一个计算表达式，该表达式的结果是数值型。下文中的例子都是数值型的表达式。

```
154;  
Vol;  
(High+Low)/2;
```

布尔型的表达式能够返回一个值 True/False，或者说它可以对表达式的值进行评估，然后返回其值，要么为 True，要么为 False。逻辑表达式永远要涉及到比较运算，下面我们有一些逻辑表达式，它也就是对一个表达式的值是 True 还是 False 进行判断。

```
Close > Open;  
5 < 2;
```

字符串的表达式可以是单个字符串，也可以字符串的组合，或者返回值为字符串的系统函数，如下，就是一个字符串的表达式：

```
"This is a test string expression" ;  
"Hello"+"World" ;  
Trim(" I Love This Game! ");
```

使用注释

注释可以标注解释语句，公式在编译执行时会忽略注释语句。

注释分为单行注释和多行注释。

单行注释

在单行中对需要注释部分之前通过添加//，使//之后的语句形成注释。以下是注释的例子：

```
//这是一个注释语句；
```

多行注释

在需要注释的部分之前添加/*,之后添加*/, 形成注释。以下是注释的例子：

```
/*这是一个多行  
注释的  
语句；  
*/
```

系统函数

TradeBlazer 公式的系统函数，可根据使用范围在相应类型的公式中直接调用，计算后返回结果值。

目前的系统函数支持四种数据类型，除了 TradeBlazer 公式中定义的三种基本数据类型：Bool，Numeric，String 之外，新加入 Long（长整型）类型，使系统函数能够更加快捷的进行计算，TradeBlazer 公式在处理的时候自动将 Numeric 和 Long 进行转化，用户无需进行特别的处理。



TradeBlazer 公式现有的系统函数主要分为：数据函数、时间函数、数学函数、其它函数、交易函数、属性函数、账户函数、颜色函数、字符串函数等。每个系统函数都有自己的适用范围和使用规范，详细说明参见附录。

标点符号

通常，在写语句的过程中，会用到很多的标点符号。可用来定义参数、定义变量、创建规则的优先权。

例如，TradeBlazer 公式用";"来标注一个语句结束。标点符号也是一个保留字，因为符号也是语言结构的一部分，在下表中列出了 TradeBlazer 公式中所用到的标点符号，和该标点符号所表达的意思：

符号	名称	说明
;	分号	语句结束的标志。
,	逗号	当函数带有多个参数时，用于分隔多个参数。
()	小括号	括号之内的表达式有计算的优先权。
""	双引号	字符串常量。
[]	中括号	回溯数据，引用以前的数据，或者数组中的元素。

	大括号	控制语句的起始。
	点	扩展数据源的数据调用。

控制语句

TradeBlazer 公式支持两大类的控制语句：[条件语句](#)和[循环语句](#)。

条件语句

条件语句包括以下四类表达方式：

If

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;
}
```

Condition 是一个逻辑表达式，当 Condition 为 True 的时候，TradeBlazer 公式语句将会被执行，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，您可以计算图表中上升缺口（当前 Bar 的开盘价高于上一个 Bar 的最高价）出现了多少次，只要在图表中使用 If 语句，当找到一个满足条件的 Bar 时，即条件为真时，变量加 1，脚本如下：

```
Vars
    NumericSeries Counter(0);
Begin
    If ( Open > High[1])
    {
        Counter = Counter[1] + 1;
        ...
    }
    ...
End
```

在 TradeBlazer 公式中，If 语句被广泛使用，如 K 线形态和特征走势，都需要大量的使用 If 语句，当条件满足的时候，在满足条件的 Bar 上面进行标记。例如，下面的语句就是特征走势的例子：

```
If(High > High[1] AND Low < Low[1])
{
    PlotNumeric("Outside Bar",High);
}
```

If 语句在不是用括号的情况，只执行下面的第一条语句，如下的语句，Alert 不会只在条件为 True 时执行，而是每次都执行。

```
If(High > High[1] AND Low < Low[1])
    PlotNumeric("Outside Bar",High);
Alert("Outside Bar");
```

要想 Alert 只在条件为 True 时执行，您需要按照下面的格式编写：

```
If(High > High[1] AND Low < Low[1])
{
    PlotNumeric("Outside Bar",High);
    Alert("Outside Bar");
}
```

If-Else

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：

```
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:

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

If-Else-If

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:

```
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:

```
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：

```
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：

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

```

    }Else
    {
        Buy(1,Close);
    }
}Else
{
    If (Close > Open)
    {
        Sell(1,Open);
    }Else
    {
        Sell (1,Close);
    }
}

```

循环语句

循环语句包括两种表达方式：**For** 和 **While**。

For

For 语句是一个循环语句，重复执行某项操作，直到循环结束。

语法如下：

```

For 循环变量 = 初始值 To 结束值
{
    TradeBlazer 公式语句;
}

```

循环变量为在之前已经定义的一个数值型变量，**For** 循环的执行是从循环变量从初始值到结束值，按照步长为 1 递增，依次执行 **TradeBlazer** 公式语句。结束值必须大于或等于初始值才有意义，初始值和结束值可以使用浮点数，但是在执行过程中会被直接取整。只计算其整数部分。

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{}。

第一次执行时，首先将循环变量赋值为初始值，然后判断循环变量是否小于等于结束值，如果满足条件，则执行 **TradeBlazer** 公式语句，同时循环变量加 1。接着重新判断循环变量是否小于等于结束值，一直到条件为 **False**，退出循环。

例如，以下的用户计算 **Price** 最近 **Length** 周期的和。

```

Params

```

```

NumericSeries Price(1);
Numeric Length(10);
Vars
    Numeric SumValue(0);
    Numeric i;
Begin
    for i = 0 to Length - 1
    {
        SumValue = SumValue + Price[i];
    }
    Return SumValue;
End

```

如果希望 For 语句从大到小进行循环，可以使用以下的语法：

```

For 循环变量 = 初始值 DownTo 结束值
{
    TradeBlazer 公式语句;
}

```

For-DownTo 让循环变量从结束值每次递减 1 直到等于结束值，依次调用 TradeBlazer 公式语句执行，初始值必须大于或等于结束值才有意义。

For 语句是比较常用的一种循环控制语句，它应用于知道循环次数的地方，很多内建用户函数中都使用 For 语句来完成相应的功能，比如 Summation, Highest, Lowest, LinearReg 等。

While

While 语句在条件为真的时候重复执行某一项操作。即，只要条件表达式的值为真(True)时，就重复执行某个动作。直到行情信息改变以致条件为假(False)时，循环才结束。

语法如下：

```

While (Condition)
{
    TradeBlazer 公式语句;
}

```

Condition 是一个逻辑表达式，当 Condition 为 True 的时候，TradeBlazer 公式语句将会被循环执行，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，以下的公式用来计算要产生大于 100000 成交量需要最近 Bar 的个数：

```

Vars
    Numeric      SumVolume (0);
    Numeric      Counter  (0);
Begin
    While (SumVolume < 100000)
    {
        SumVolume = SumVolume + Vol[Counter]
        Counter = Counter + 1;
    }
End

```

首先，我们定义两个变量 **SumVolume** 和 **Counter**，并将其默认值设为 0。当 **SumVolume < 100000** 这个表达式为 **True** 时，**While** 内的 **TradeBlazer** 公式语句一直被调用，将前 **Counter** 个 **Bar** 的 **Vol** 加到 **SumVolume** 中，当 **SumVolume** 大于等于 100000 时，退出循环。

在使用 **While** 循环的时候，有可能会遇到循环一直执行，永远不能退出的情况，这种情况我们称之为死循环，比如下面的语句：

```

While (True)
{
    TradeBlazer 公式语句;
}

```

在这种情况下，循环将一直执行，导致程序不能继续工作，在这种情况下，我们可以使用 **Break** 来跳出循环，详细情况参加下节。

Break

针对上节的例子，要想从死循环中跳出，**我们可以在循环之中添加 Break 语句**，如下：

```

While (True)
{
    TradeBlazer 公式语句;
    If (Condition)
        Break;
}

```

循环在每次执行后，都将判断 **Condition** 的值，当 **Condition** 为 **True** 时，则执行 **Break** 语句，跳出整个循环。

Continue

有的时候在循环中，我们可能希望跳过后面的代码，进入下一次循环，在这种情况下，可以使用 **Continue**

语句来达到目的，如下：

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

当 Condition1 满足时，循环被执行，在执行完 TradeBlazer 公式语句 1 后，将判断 Condition2 的值，当 Condition2 为 True，将跳过 TradeBlazer 公式语句 2，重新判断 Condition1 的值，进入下一次循环。否则将继续执行 TradeBlazer 公式语句 2。

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，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：

```
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("Hi");  //定义字符串参数 strTmp，默认值为 Hi;
```

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：

```
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：

```
Params
    Numeric      MyVal1(0);
    Numeric      MyVal2(0);
    Numeric      MyVal3;
```

参数使用

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：

```
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：

```
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：

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

变量

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：

```
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：

```
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](#)。

变量类型

TradeBlazer 公式支持九种数据类型，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。

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：

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。

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：

```
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**：

```
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。

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：

```

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。

```

Vars
    Bool    Con1(False);
Begin
    Con1 = High > Highest(High,10)[1];
    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：

```

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

```

```
}  
End
```

序列变量

序列变量是变量中的一种，可以对序列变量进行回溯获取以前 **Bar** 的变量数据。序列变量的声明和简单变量一样，只是定义的数据类型不同，您必须选择以下的 3 种类型来定义序列变量：

NumericSeries/BoolSeries/StringSeries。例如：

```
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。

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](#)。

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：

```
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：

```
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：

```

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

```

各类公式

TradeBlazer 公式包含的公式类型如下：

[用户函数](#)，[用户字段](#)，[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：

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

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 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 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,这些用户函数用户辅助完成技术分析。

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：

```
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：

```
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：

```
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
Numeric	Numeric, NumericRef, NumericSeries

NumericRef	Numeric, NumericRef, NumericSeries
NumericSeries	NumericSeries
Bool	Bool, BoolRef, BoolSeries
BoolRef	Bool, BoolRef, BoolSeries
BoolSeries	BoolSeries
String	String, StringRef, StringSeries
StringRef	String, StringRef, StringSeries
StringSeries	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：

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

我们也可以按照以下方式进行调用：

```
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。只能够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**：

```

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:

```
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:

```

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

```

用户字段

用户字段作为一种特殊用途的公式, 它主要用于在报价类窗体中使用, 最大的特点在于: 用户字段不像其他公式, 它只需要最后一个 Bar 的输出数据, 而不是所有 Bar 的输出数据。因此, 对于用户字段的处理和

其他公式都不相同，我们对用户字段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 FieldNumeric、FieldBool、FieldString 返回数据，返回数据类型为三种基本类型的组合；
- 可以支持 Alert 来进行报警；
- 用户字段可以调用所有的用户函数进行计算；
- 用户字段可以根据设置调用部分的系统函数；
- 用户字段在执行时，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。

例如，用户字段 Field1 计算最近 5 个 Bar 的收盘价的最高价,用户函数 Highest 计算最大值。脚本如下：

```
Params
    Numeric Length(5);
Vars
    Numeric HighValue(0);
Begin
    HighValue = Highest(Close,Length);
    FieldNumeric("Highest Bar", HighValue);
End
```

在用户字段中只能输出一组数据，该组数据通过 FieldNumeric/FieldBool/FieldString 的第一个参数：输出值的名称来进行识别，**在一个用户字段中，不能出现两个不同的输出值名称**，以下为错误的代码：

```
FieldNumeric("Test1",Close);
FieldNumeric("Test2",Open);
FieldBool("Test1",True);
```

以下为正确的代码：

```
FieldNumeric("Test1",10);
FieldBool("Test1",True);
FieldString("Test1","Test String");
```

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：

```
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。

```

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：

```
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；
- 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；
- 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：

```
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：

```
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：

```
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。

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。

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
SetExitOnClose	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。
SetBreakEven	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。
SetStopLoss	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。
SetProfitTarget	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。
SetPeriodTrailing	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。
SetPercentTrailing	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。
SetDollarTrailing	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。
SetInactivate	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。

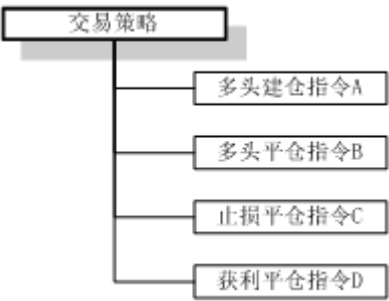
ExitPosition 是布尔型参数，当 ExitPosition=True 时，表示将当前所有的持仓作为一个整体，根据其平均建仓成本，计算各平仓函数的盈亏，当条件满足时，会将所有仓位一起平掉；当 ExitPosition=False 时，表示单独对每个建仓位置进行平仓，单独计算各平仓函数盈亏时，当单个建仓位置条件满足后，平掉该建仓位置即可。

交易策略

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。

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。