

# 简介

---



# Pine Script

Pine Script 是 TradingView 的编程语言。它允许交易者创建自己的交易工具并在我们的服务器上运行。我们将 Pine 设计成一种轻量级但功能强大的语言，用于开发指标和策略，然后可以进行回测。TradingView 的大多数内置指标都是用 Pine 编写的，我们活跃的 Pine 开发者社区已经发布了超过 10 万个社区脚本。

我们的明确目标是让尽可能多的人了解 Pine，并使之易于理解。Pine 是基于云的，因此与客户端编程语言不同。虽然我们很可能不会把 Pine 发展成一种成熟的语言，但我们确实在不断地改进它，并且总是乐于考虑对新功能的要求。

由于每个脚本都使用云中的计算资源，我们必须加以限制，以便在我们的用户中公平地分享这些资源。我们努力设置尽可能少的限制，但当然也要根据平台顺利运行的需要来实施。限制适用于额外请求的数据量、执行时间、内存使用和脚本大小。

## 第一步

---

### 介绍

欢迎来到《Pine 脚本用户手册》，它将伴随你学习用 Pine 编程自己的交易工具的旅程。也欢迎你加入 TradingView 上非常活跃的 Pine 开发者社区。

在本页面中，我们将介绍一种循序渐进的方法，你可以按照这种方法逐步熟悉 [TradingView](#) 上用 Pine 编程语言编写的指标和策略（也称为脚本）。我们将让你开始你的旅程：

1. 使用平台上现有的脚本中的一些。
2. 阅读现有脚本的 Pine 代码。
3. 写 Pine 的脚本。

如果您已经熟悉了 TradingView 上 Pine 脚本的使用，并且现在准备学习如何编写自己的脚本，那么请跳到本页面的 [编写脚本](#) 部分。

如果您是我们平台的新手，那么请继续阅读！

# 使用脚本

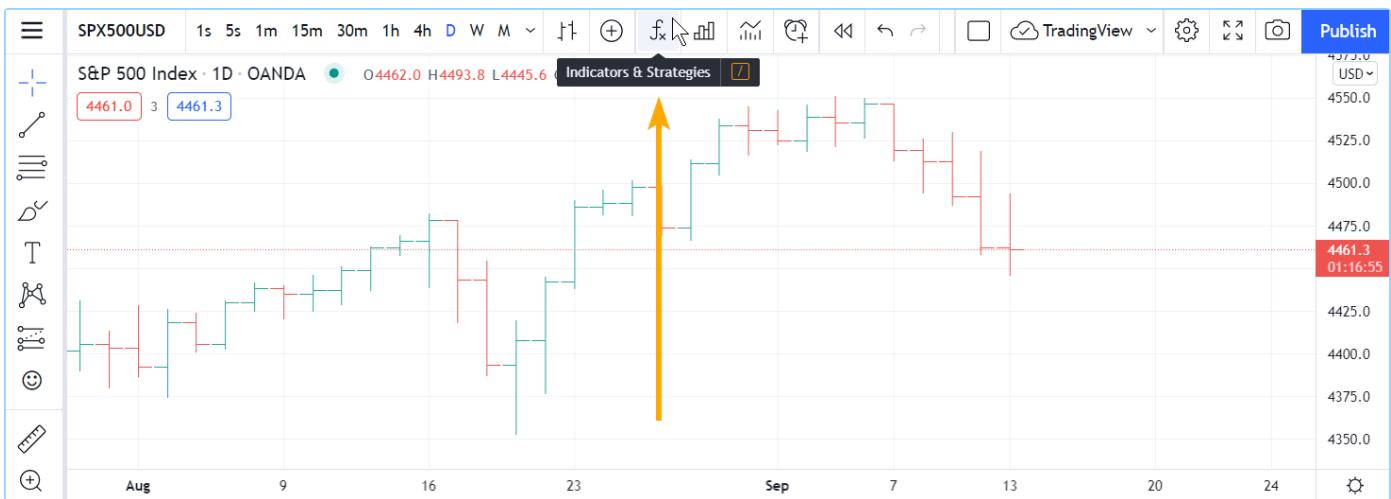
如果您有兴趣在TradingView上使用技术指标或策略，您可以首先开始探索我们平台上已有的数千个指标。你可以通过两种不同的方式访问平台上的现有指标。

- 通过使用图表的 "指标和策略" 按钮，或
- 通过浏览TradingView的[社区脚本](#)，这是世界上最大的交易脚本库，有超过10万个脚本，其中大部分是免费和开源的，这意味着你可以看到它们的Pine代码。

如果你能找到已经为你写好的你所需要的工具，这可能是一个很好的开始，并逐渐成为熟练的脚本用户，直到你准备好在Pine中开始你的编程之旅。

## 从图表中加载脚本

要探索并从你的图表中加载脚本，请使用 "指标和策略" 按钮。



对话框的左窗格中显示了不同类别的脚本。

- **收藏夹 (Favorites)** 列出了你已经 "收藏" 的脚本，你可以点击它的名字左边的星星，当你把鼠标放在它上面时，它就会出现。
- **我的脚本 (My scripts)** 显示您在Pine Editor中编写并保存的脚本。它们被保存在TradingView的云中。
- **内置插件 (Built-ins)** 将所有TradingView内置插件分为四类：指标、策略、蜡烛图和成交量信息。大多数是用Pine编写的，可免费使用。
- **社区脚本 (Community Scripts)** 是您可以从TradingView用户编写的100,000多个已发布的脚本中搜索的地方。
- **仅限邀请的脚本 (Invite-only scripts)** 包含您已被其作者授予访问权的仅限邀请的脚本列表。

在这里，包含TradingView内置脚本的部分被选中。

The screenshot shows the TradingView interface with a sidebar on the left containing links for Favorites, My scripts, Built-ins (which is highlighted in blue), Community Scripts, and Invite-only scripts. The main area has tabs for Indicators, Strategies, Candlestick patterns, and Volume profile. Under the Indicators tab, there is a section for SCRIPT NAME with several options: Accumulation/Distribution, Advance Decline Line, Advance Decline Ratio, and Advance/Decline Ratio (Bars).

当你点击其中一个指标或策略（名字后面有绿色和红色箭头的那些），它就会加载到你的图表上。

## 浏览社区脚本

从[TradingView的主页](#)，你可以从“社区”菜单中调出社区脚本流。这里，我们指的是“编辑精选”部分，但还有许多其他类别，你可以选择。

The screenshot shows the TradingView homepage with a search bar and navigation links for Chart, Trade, Markets, Screeners, and a dropdown menu. The dropdown menu is open, showing sections like Community (highlighted) and More, with sub-options like Ideas, Scripts, Streams, and More. A large orange box highlights the “Community” section. Below the search bar, there's a “SCRIPTS” section with a “Monthly Returns in PineScript Strategies” chart by QuantNomad. To the right, there's a “Top authors: Scripts” section listing authors like ChrisMoody, LazyBear, HPotter, KivancOzbilic, RicardoSantos, and vdubus, each with a “Following” button. Further down, there's a detailed description of the Public Library's scripts and a note about Pine Wizards and Pine Coders.

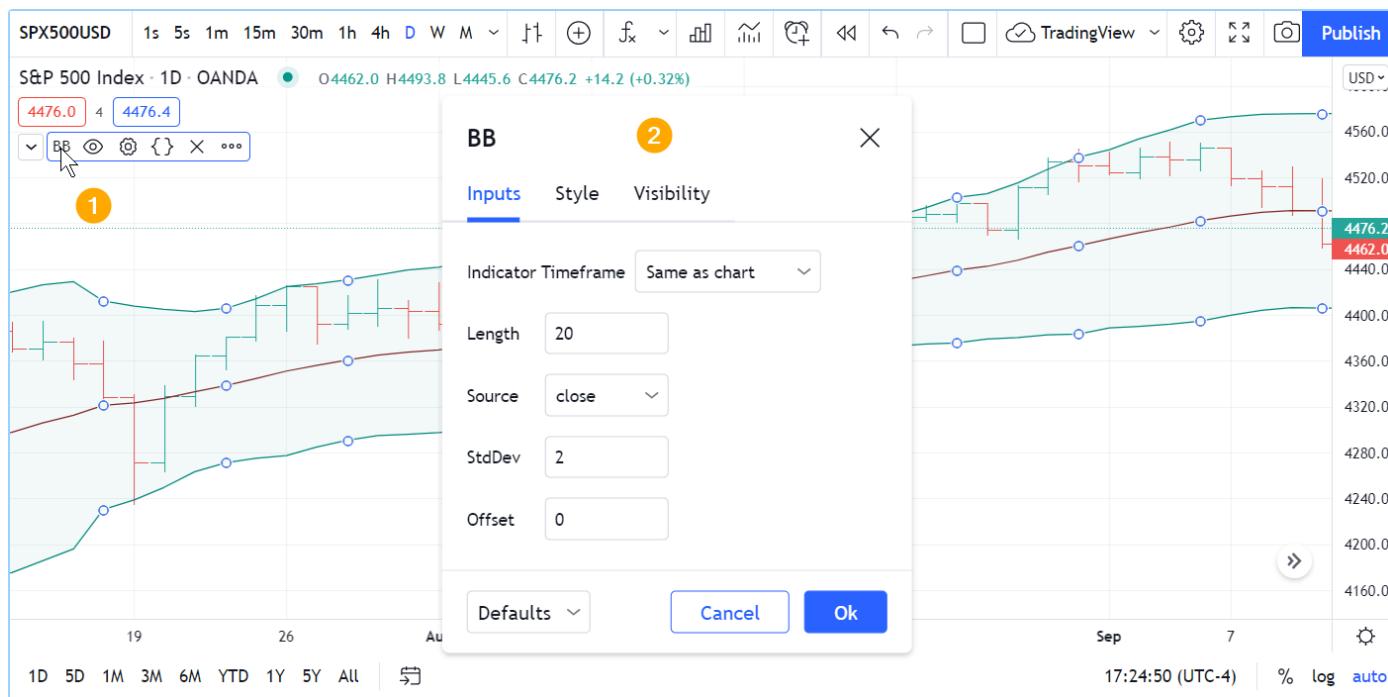
你也可以用主页上的“搜索”栏来搜索脚本，并使用不同的标准来过滤脚本。帮助中心有一个页面解释了[不同类型的脚步](#)。

脚步流显示了脚步的图标，即占位符，显示了每个出版物的图表和描述以及作者的小视图。通过点击它，你将打开脚步的页面，在那里你可以在图表上看到该脚步，阅读作者的描述，喜欢该脚步，留下评论或阅读该脚步的源代码（如果它是开源发布的）。

一旦你在社区脚步中找到了一个有趣的脚步，请按照帮助中心的指示来[把它加载到你的图表上](#)。

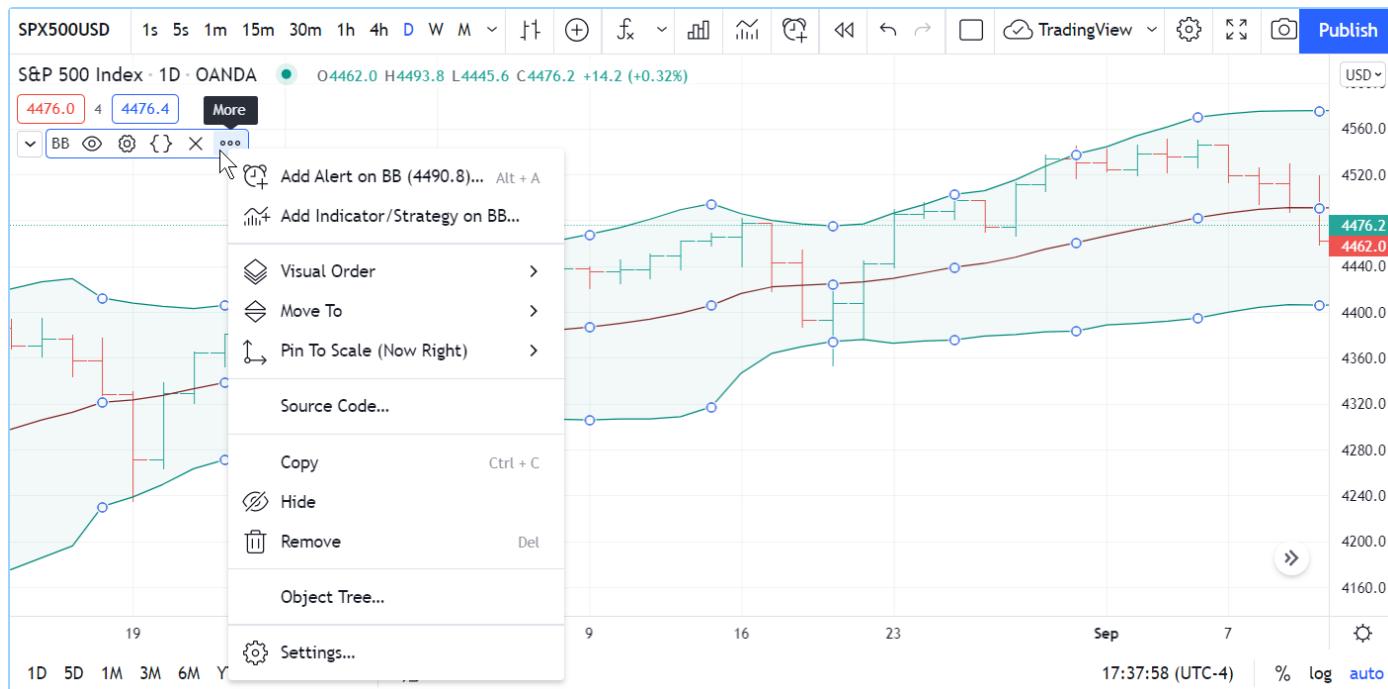
# 改变脚本设置

一旦一个脚本被加载到图表上，你可以双击它的名字(#1)来调出它的“设置/输入”(Settings/Inputs)标签(#2)。



在“输入”(Inputs)选项卡中，你可以编辑作者设置可编辑的设置。你可以用同一对话框中的“风格”(Style)选项卡来配置脚本的一些视觉效果，用“可见性”(Visibility)选项卡来配置脚本应该出现在哪个时间段。

其他的设置可以从所有脚本的名字右边的按钮中获得，当你把鼠标放在它上面时，也可以从“更多”菜单(三个点)中获得。



## 阅读脚本

阅读由好的程序员编写的代码是发展你对语言理解的最好方法。对于Pine来说，这和其他所有的编程语言一样，都是如此。找到好的开源Pine代码是相对容易的。这些都是由TradingView上的优秀程序员编写的可靠的代码来源：

- TradingView的内置指标
- 被选为[编辑推荐](#)的脚本
- 由[PineCoders官方账户关注的作者](#)编写的脚本
- 许多脚本由具有较高声誉和开源出版物的作者编写。

阅读你在[开放库](#)中找到的脚本的代码是很容易的；如果你在脚本小部件的右上角没有看到灰色或红色的“锁”图标，这表明该脚本是开源的。打开它的脚本页面，你就可以看到它的源代码。

要查看TradingView内置程序的代码，请在您的图表上加载该指标，然后将鼠标悬停在其名称上并选择“源代码”大括号图标（如果您没有看到它，那是因为该指标的源代码不可用）。当你点击该图标时，Pine编辑器将打开，从那里你可以看到脚本的代码。如果你想调试它，你需要使用编辑器窗格右上方的“更多”菜单按钮，并选择“复制一份...”（Make a copy...）。然后你将能够修改和保存代码。因为你已经创建了一个不同版本的脚本，所以你需要使用编辑器的“添加到图表”按钮，将这个新副本添加到图表中。

这显示了我们在图表上选择了指标的“查看来源”（View source）按钮后，Pine编辑器刚刚打开。我们即将对其源文件进行复制，因为它现在是只读的（在编辑器中其文件名附近的“锁”图标表示）。



你也可以通过使用“打开/新的默认内置脚本.....”（Open/New default built-in script...）菜单选择，从Pine编辑器（可从图表底部的“Pine Editor”标签进入）打开TradingView内置指标。

## 写脚本

我们建立Pine Script的目的是为了让初出茅庐的交易者和经验丰富的交易者都能创建自己的交易工具。我们的设计使它对于第一次学习的程序员来说相对容易--尽管学习第一种编程语言，如交易，对任何人来说都很容易--但对于有知识的程序员来说，它又足够强大，可以建立中等复杂度的工具。

Pine允许你编写三种类型的脚本。

- 指标如RSI、MACD等。
- 策略，包括发出交易指令的逻辑，可以进行回测和正向测试。
- 库，由更高级的程序员使用，将经常使用的函数打包，可以被其他脚本重复使用。

我们推荐的下一步是编写你的[第一个指标](#)。)

# 第一个指标

## Pine编辑器

Pine编辑器是你处理你的脚本的地方。虽然你可以使用任何你想要的文本编辑器来编写你的Pine脚本，但使用我们的编辑器有很多好处。

- 它可以按照Pine的语法突出显示你的代码。
- 当你把鼠标悬停在内置和库函数上时，它会弹出语法提醒。
- 当你用ctrl+单击（Windows）或者cmd+单击（MAC）Pine关键词时，它提供了快速访问Pine参考手册的弹出窗口。
- 它提供了一个自动完成功能，你可以用ctrl + space / cmd + space来激活。
- 它使写/编译/运行周期变得很快，因为保存加载在图表上的新脚本会立即执行它。
- 虽然不像外面的顶级编辑器那样功能丰富，但它提供了关键的功能，如搜索和替换，多光标和版本管理。

要打开编辑器，请点击TradingView图表底部的 "Pine Editor" 标签。这将打开编辑器的窗口。

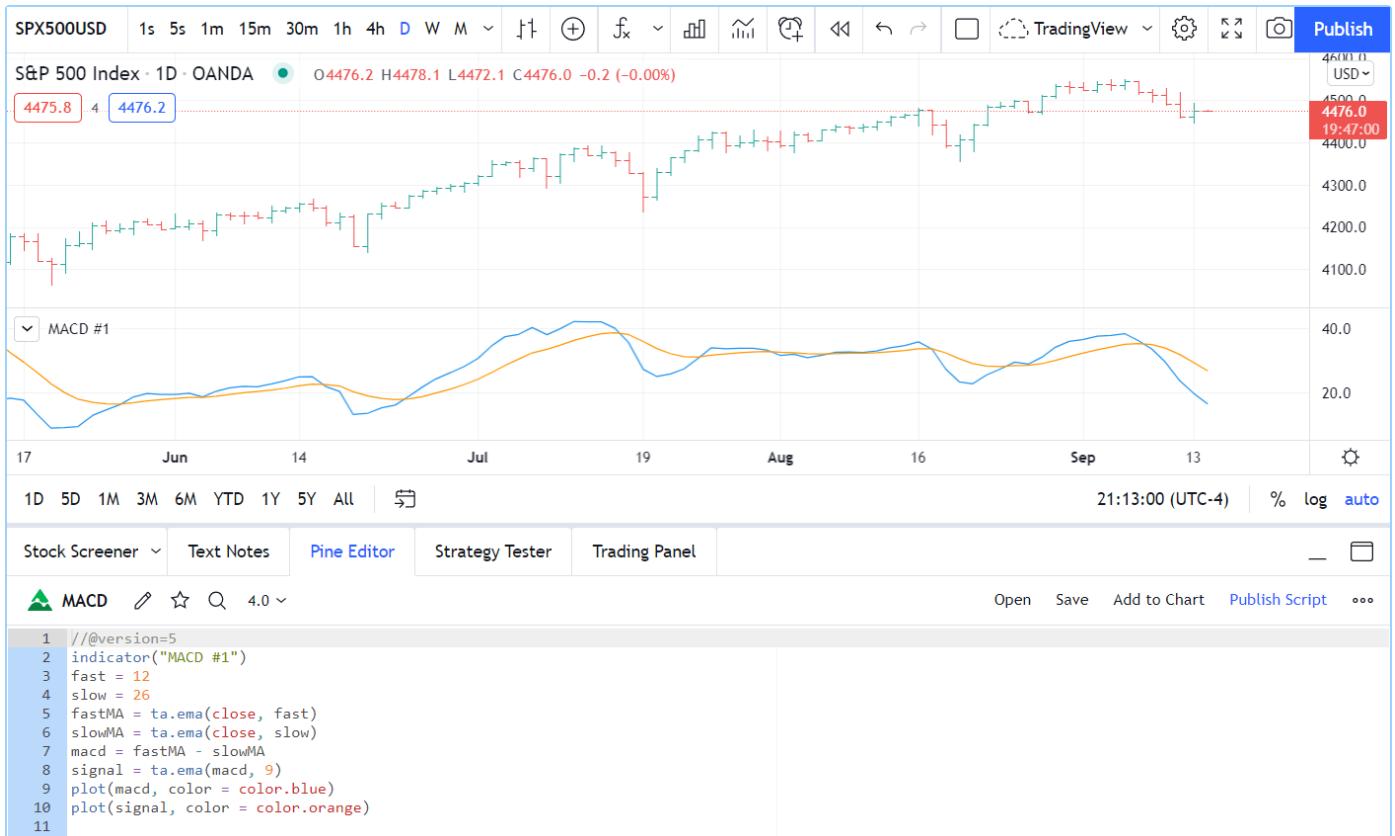
# 第一个版本

我们现在将创建我们的第一个工作的Pine脚本，一个在Pine中实现[MACD](#)指标的脚本。

```
//@version=5
indicator("MACD #1")
fast = 12
slow = 26
fastMA = ta.ema(close, fast)
slowMA = ta.ema(close, slow)
macd = fastMA - slowMA
signal = ta.ema(macd, 9)
plot(macd, color = color.blue)
plot(signal, color = color.orange)
```

- 首先调出编辑器右上方的 "打开"(Open)下拉菜单，选择 "新建空白指标"(New blank indicator)。
- 然后复制上面的例子脚本，注意不要把行号包括在你的选择中。
- 选择已经在编辑器中的所有代码，用示例脚本替换它。
- 点击 "保存"(Save)并为你的脚本选择一个名称。你的脚本现在被保存在TradingView的云端，但在你的账户名下。除了你，没有人可以使用它。
- 在编辑器的菜单栏中点击 "添加到图表"(Add to Chart)。MACD指标出现在你的图表下的一个单独的窗口中。

你的第一个Pine脚本正在你的图表上运行，它应该看起来像这样。



让我们逐行看一下我们的脚本代码。

- 第1行: `//@version=5`

这是一个包含编译器指令的注释，告诉编译器脚本将使用Pine的第5版。

- 第2行。 `indicator("MACD #1")`

定义了脚本的名称，它将在图表上显示为 "MACD"。

- 第3行。 `fast = 12`

定义了一个 `fast` 整数变量，它将是快速EMA的长度。

- 第4行。 `slow = 26`

定义一个 `slow` 整数变量，它将是慢速EMA的长度。

- 第5行: `fastMA = ta.ema(close, fast)`

定义变量 `fastMA`，包含EMA计算的结果（指数移动平均线），其长度等于 `fast` (12)，根据每日收盘价而来。

- 第6行: `slowMA = ta.ema(close, slow)`

定义变量 `slowMA`，包含长度等于 `slow` (26) 的EMA计算结果，来自 `close`。

- 第7行: `macd = fastMA - slowMA`

定义变量 `macd` 为两个EMA的差值。

- 第8行: `signal = ta.ema(macd, 9)`

定义变量 `signal` 为 `macd` 的平滑值，使用EMA算法（指数移动平均），长度为9。

- 第9行。 `plot(macd, color = color.blue)`

调用 `plot` 函数，使用蓝线输出变量 `macd`。

- 第10行: `plot(signal, color = color.orange)` 调用 `plot` 函数，用蓝色线条输出变量 `macd`  
调用 `plot` 函数，用一条橙色的线来输出变量。

## 第二个版本

我们脚本的第一个版本是 手动 计算MACD，但由于Pine是为编写指标和策略而设计的，许多常见的指标都有内置的Pine函数，包括一个用于MACD的函数: [ta.macd\(\)](#)。

这是我们脚本的第二个版本：

```
//@version=5
indicator("MACD #2")
fastInput = input(12, "Fast length")
slowInput = input(26, "Slow length")
[macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)
plot(macdLine, color = color.blue)
plot(signalLine, color = color.orange)
```

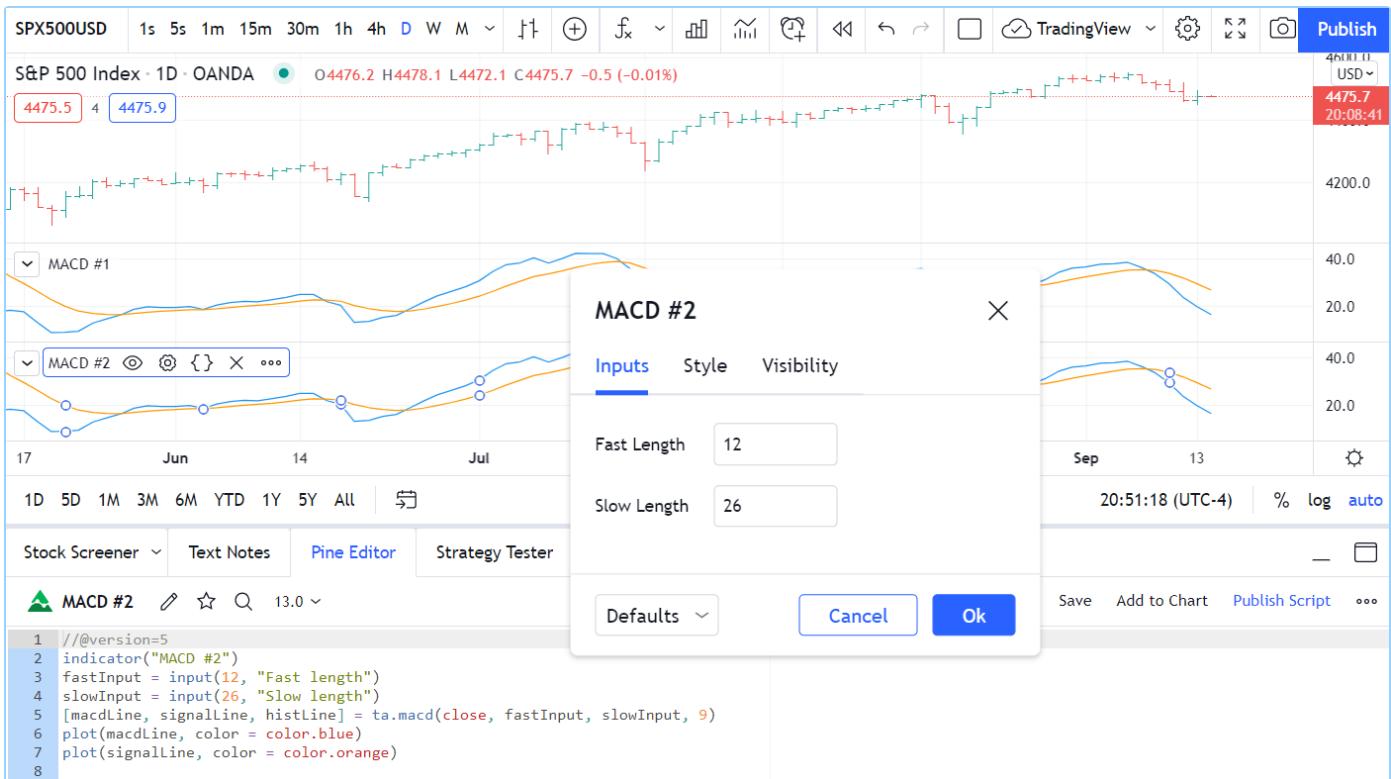
请注意，我们已经：

- 增加了输入，所以我们可以改变MAs的长度
- 我们现在使用[ta.macd\(\)](#)Pine内置函数来计算我们的MACD，这为我们节省了三行，使我们的代码更容易阅读。

让我们重复之前的过程，在一个新的指标中复制该代码：

- 首先调出编辑器右上方的 "打开" (Open) 下拉菜单，选择 "新建空白指标" (New blank indicator) 。
- 然后复制上面的例子脚本，同样注意不要把行号复制进去。
- 选择已经在编辑器中的所有代码，用我们的脚本的第二个版本替换它。
- 点击 "保存" (Save)，为你的脚本选择一个与前一个不同的名称。
- 在编辑器的菜单栏中点击 "添加到图表"(Add to Chart)。 `MACD #2` 指标出现在 `MACD #1` 指标下的一个单独的窗格中。

你的第二个Pine脚本正在你的图表上运行。如果你在图表上双击该指标的名称，你将调出该脚本的 "设置/输入" 选项卡，现在你可以改变慢速和快速的长度。



让我们来看看在我们的脚本的第二个版本中，有哪些行发生了变化。

- 第2行: `indicator("MACD #2")`

我们将 `#1` 改为 `#2`，这样我们的第二个版本的指标就会在图表上显示一个不同的名称。

- 第3行: `fastInput = input(12, "Fast length")`

我们没有给变量分配一个常量值，而是使用了 `input()` 函数，因此我们可以在脚本的“设置/输入”(Settings/Inputs)标签中改变数值。`12` 将是默认值，该字段的标签将是“快速长度”。如果在“输入”选项卡中改变了数值，`fastInput` 变量的内容将包含新的数值，脚本将用这个新的数值在图表上重新执行。请注意，正如我们的 [Pine Style Guide](#) 所建议的，我们在变量名称的后面加上 `input`，以便在以后的脚本中提醒我们，它的值来自于用户的输入。

- 第4行: `slowInput = input(26, "Slow length")`

我们对慢速长度做同样的处理，注意使用不同的变量名、默认值和字段标签。

- 第5行: `[macdLine, signalLine, histLine] = ta.macd(close, fastInput, slowInput, 9)`

这就是我们调用 `ta.macd()` 内置函数，只用一行就能完成第一版的所有计算。该函数需要四个参数（函数名称后面的值，用圆括号括起来）。它在三个变量中返回三个值，而不是像我们之前使用的函数那样只返回一个值，这就是为什么我们需要将接收函数结果的三个变量列表用方括号括起来，在 `=` 符号的左边。注意，我们传递给函数的两个值是包含快速和慢速长度的“输入”变量：`fastInput` 和 `slowInput`。

- 第6行和第7行。

我们绘制的变量名称已经改变了，但这几行所做的事情与我们第一个版本中的相同。

我们的第二个版本所做的计算与第一个版本相同，但我们可以改变用于计算的两个变量。我们的代码也更简单，短了三行。我们已经改进了我们的脚本。

## 接下来

我们现在建议你去我们的[下一步](#)页面。

# 接下来的步骤

---

在你的[第一步](#)和你的[第一个指标](#)之后，让我们通过分享一些指导你学习Pine的要点，来探索更多的Pine用法。

## "指标"与"策略"

---

Pine的[策略](#)是用来对历史数据进行回测和对公开市场进行正向测试。除了指标计算外，它们还包含[strategy.\\*\(\)](#)调用，用于向Pine的交易所模拟器发送交易指令，然后可以模拟其执行。策略在图表底部的"策略测试器" (Strategy Tester) 标签中显示回测结果，该标签紧挨着 "Pine编辑器" (Pine Editor) 标签。

Pine指标也包含计算，但不能用于回测。因为它们不需要交易所模拟器，它们使用的资源更少，运行速度也会更快。因此，只要有可能，使用指标是很有利的。

指标和策略都可以以叠加模式（在图表的条形图上）或窗格模式（在图表下方或上方的单独部分）运行。两者都可以在各自的空间中绘制信息，并且都可以生成[警报事件](#)。

## 脚本是如何执行的

---

Pine脚本不像许多编程语言中的程序，执行一次就停止了。在Pine的*runtime*环境中，脚本的运行相当于一个无形的循环，它在你所处的任何图表的每个K线上从左到右执行一次。脚本执行时已经关闭的图表条被称为历史条。当执行到图表的最后一个条形图，并且是开市的，它是在实时条形图。然后脚本在每次检测到价格或成交量变化时执行一次，在关闭时对该实时条进行最后一次执行。然后，该实时条就变成了一个失效的实时条。请注意，当脚本实时执行时，它不会在每次价格/成交量更新时重新计算图表上的所有历史条。它已经对这些条形图进行了一次计算，所以它不需要在每一个图表刻度上重新计算。更多信息请参见[执行模式](#)页面。

当一个脚本在历史条上执行时，[close](#)内置变量保持该条的收盘价。当脚本在实时条上执行时，[close](#)返回符号的当前价格，直到该条关闭。

与指标相反，Pine策略通常只在实时条上执行一次，当它们收盘时。如果你需要，它们也可以被配置为在每次价格/成交量更新时执行。更多信息请参见[策略](#)页面，了解策略的计算方式与指标不同。

## 时间序列

---

Pine脚本中使用的主要数据结构被称为[时间序列](#)。时间序列对脚本执行的每一个柱状体都包含一个值，所以当脚本在更多的柱状体上执行时，它们会不断扩展。时间序列过去的值可以用Pine的历史引用操作符来引用，例如：`□.close[1]`，指的是在执行脚本中前一个k线的[close](#)的值。

虽然这种索引机制可能会让很多程序员联想到数组，但是时间序列是不同的，用数组的方式思考将不利于理解这个关键的Pine概念。对[执行模型](#)和[时间序列](#)的良好理解是理解Pine脚本如何工作的关键。如果你以前从未处理过以时间序列组织的数据，你将需要练习使它们为你工作。一旦你熟悉了这些关键的概念，你就会发现，通过将时间序列的使用与我们设计的内置函数相结合，只需很少的几行Pine代码就可以完成很多工作。

## 发布脚本

---

TradingView是一个庞大的Pine开发员社区和来自世界各地的数百万名交易者的家园。一旦你对Pine足够熟练，你可以选择与其他交易者分享你的脚本。在这样做之前，请花时间充分学习Pine，以便为交易者提供一个原创的、可靠的工具。所有公开发布的脚本都会被我们的审核团队分析，并且必须遵守我们的[脚本发布规则](#)，该规则要求这些脚本必须是原创的并且有好的文档说明。

如果想使用Pine脚本供自己使用，只需在Pine编辑器中编写，并从那里将其添加到你的图表中；你不必发布它们来使用它们。如果你想与几个朋友分享你的脚本，你可以私下发布它们，并把浏览器的链接发给你的朋友，让他们看到你的私人脚本。更多信息请参见发布页面。

## 了解Pine的文档

---

尽管从已发布的脚本中阅读代码无疑是有效的，但要达到对Pine的很高的熟练程度，花时间在我们的文档中是必要的。我们在Pine上的两个主要文档来源是：

- 这个Pine用户手册
- 我们的[Pine参考手册](#)

[Pine参考手册](#) 记录了每个变量、函数或Pine关键字的作用。它是所有Pine开发者的必备工具；如果你不参考它而试图编写复杂的脚本，你的过程将是痛苦的。它有两种访问方式：我们刚刚链接到的HTML，以及弹出式版本，可以从Pine编辑器中访问，方法是ctrl+点击一个关键字，或者使用编辑器的 "更多/Pine脚本参考 弹出式" (More/Pine Script reference (pop-up)) 菜单。参考手册被翻译成了其他语言。

Pine有五个不同的版本。确保你使用的文档与你正在编码的Pine版本相一致。

## 如何阅读本文档？

---

这本[Pine用户手册](#)包含了许多代码实例。通过阅读它，你既可以学习Pine的基础，又可以研究示例脚本。阅读关键概念并立即用代码进行尝试，是学习任何编程语言的有效方法。希望你已经在[第一个指标](#)页面中尝试了，把文档中示例代码复制到编辑器中然后调试他们。大胆探索！你不会破坏任何东西。

这就是你正在阅读的Pine用户手册的组织方式。

- [语言](#)部分解释了Pine语言的主要组成部分以及脚本如何执行。
- [Concepts](#) 部分更多的是面向任务。它解释了如何在Pine中做事情。
- [Writing](#) 部分探讨了帮助你编写和发布脚本的工具和技巧。
- [FAQ](#) 部分回答了Pine编码者的常见问题。
- [错误信息](#) 页面记录了最常见的运行时和编译器错误的原因和修复方法。
- [发布说明](#) 页面是你可以跟踪Pine的频繁更新的地方。
- [迁移指南](#) 部分解释了如何在不同版本的Pine之间移植。
- [Where can I get more information](#) 页面列出了其他与Pine相关的有用内容，包括当你在代码上遇到困难时可以向哪里提问。

我们祝愿你在使用Pine的过程中获得成功.....并进行交易

## 执行模式

---

Pine运行时的执行模型与Pine的[时间序列](#)和[类型系统](#)密切相关。了解这三者是充分利用Pine的关键。

执行模型决定了你的脚本如何在图表上执行，从而决定了你在脚本中编写的代码如何工作。如果没有Pine的运行时，你的Pine代码将无所作为，它在你的代码编译后启动，并在你的图表上执行，因为其中一个[触发脚本执行的事件](#)已经发生。

当Pine脚本被加载到图表上时，它在每个历史条上执行一次，使用每个条的可用OHLCV（开盘open、最高high、最低low、收盘close、成交量volume）值。一旦脚本的执行到达数据集的最右边的条形，如果目前交易活跃，那么Pine 指标将在每次更新发生时执行一次，即价格或成交量变化。Pine 策略默认只在最右边的条形图收盘时执行，但它们也可以被配置为在每次更新时执行，就像指标那样。

所有的符号/时间框架对都有一个数据集包括有限数量的条形图。当你向左滚动图表以查看数据集的早期条形时，相应的条形会在图表上加载。当该符号/时间框架对没有更多的条形图或您的账户类型允许的最大条形图数量已被加载时，加载过程就会停止[[1](#)]。你可以向左滚动图表，直到数据集的第一个条形，它的索引值为0（见[bar\\_index](#)）。

当脚本第一次在图表上运行时，数据集中的所有条形图都是历史条形图，除了最右边的条形图，如果交易时段是活跃的。当交易进行时候在最右边的条形图上，它被称为实时条形图。当检测到价格或成交量的变化时，实时条就会更新。当实时条形图关闭时，它将成为一个失效的实时条形图，并打开一个新的实时条形图。

## 基于历史条的计算

让我们来看看一个简单的脚本，并跟踪它在历史条上的执行情况。

```
//@version=5
indicator("My Script", overlay = true)
src = close
a = ta.sma(src, 5)
b = ta.sma(src, 50)
c = ta.cross(a, b)
plot(a, color = color.blue)
plot(b, color = color.black)
plotshape(c, color = color.red)
```

在历史条上，当OHLCV值都是已知的时候，脚本会在相当于条的收盘时执行。在执行脚本之前，内置的变量如 "开盘"、"高点"、"低点"、"收盘"、"成交量" 和 "时间" 都被设置为与该条线相应的值。脚本在每个历史条上执行一次。

我们的例子脚本首先在数据集的第一个条形索引0处执行。每条语句的执行都使用当前条形的数值。因此，在数据集的第一个条形图上，下面的语句：

```
src = close
```

用第一个条形图的 `close` 值初始化变量 `src`，然后依次执行下面的每一行。因为该脚本对每个历史条形图只执行一次，所以该脚本总是用特定历史条形图的相同 `close` 值进行计算。

脚本中每一行的执行都会产生计算结果，进而产生指标的输出值，然后可以在图表上绘制出来。我们的例子在脚本的最后使用了 `plot` 和 `plotshape` 调用来输出一些数值。在策略的情况下，计算的结果可以用来绘制数值或决定要下的订单。

在第一个条形图上执行和绘制之后，脚本在数据集的第二个条形图上执行，这个条形图的索引是1。然后这个过程重复进行，直到数据集的所有历史条形图被处理，脚本到达图表的最右边。



## 基于实时条形图的计算

Pine脚本在实时条上的行为与在历史条上的行为有很大不同。回顾一下，当图表符号上的交易活跃时，实时条是图表上最右边的条。另外，记得策略在实时条上可以有两种不同的行为方式。默认情况下，它们只在实时条形图关闭时执行，但是`strategy`声明语句的`calc_on_every_tick`参数可以设置为true，以修改策略的行为，使其在每次实时条形图更新时执行，就像指标那样。因此这里描述的指标行为只适用于使用`calc_on_every_tick=true`的策略。

脚本在历史条上的执行和在实时条上的执行最重要的区别是，在历史条上只执行一次，而在实时条上每次更新都会执行脚本。这就意味着内置的变量，如“高”、“低”和“收”，在历史条形图上是不会改变的，但在实时条形图上，脚本的每一次迭代都可以改变。脚本计算中使用的内置变量的变化，反过来也会引起这些计算结果的变化。这是脚本跟随实时价格行动所必需的。因此，同一个脚本在实时条形图中每次执行都可能产生不同的结果。

**注意：**在实时条形图中，`close`变量总是代表当前价格。同样，`high`和`low`内置变量代表自实时条形图开始以来达到的最高价和最低价。Pine内置变量将只代表实时条形图最后更新时的最终值。

让我们在实时条形图中遵循我们的脚本例子。

当脚本到达实时条形图时，它第一次执行。它使用内置变量的当前值来产生一组结果，并在需要时绘制它们。在下一次更新发生时，在脚本执行另一次之前，它的用户定义的变量被重置到一个已知的状态，对应于上一个条形收盘时的最后一次`commit`的状态。如果没有对变量进行提交，因为它们在每个柱状图上都被初始化了，那么它们将被重新初始化。在这两种情况下，它们最后的计算状态都会丢失。绘图的标签和线条的状态也会被重置。在实时条形图中的脚本的每一次新的迭代之前，对脚本的用户定义的变量和绘图的这种重设被称为回滚。它的作用是将脚本重置到与实时条打开时相同的已知状态，所以实时条中的计算总是从一个干净的状态开始执行。

随着实时条形图中价格或成交量的变化，脚本值的不断重新计算会导致这样一种情况：在我们的例子中，变量`c`因为发生了交叉而变成了真，因此脚本的最后一条线所绘制的红色标记会出现在图表上。如果在下一次价格更新时，价格的变化使`close`值不再产生计算，使`c`成为真实，因为不再有交叉，那么之前绘制的标记将消失。

当实时条形图关闭时，脚本会执行最后一次。像往常一样，变量在执行前被回滚。然而，由于这次迭代是实时条形图上的最后一次迭代，当计算完成后，变量将被承诺为条形图的最终值。

总结一下实时条形图的过程。

- 脚本在实时条形图打开时执行，然后每次更新时执行一次。
- 变量在每次实时更新前\*\*回滚。
- 变量在收盘时被提交\*\*一次更新。

## 触发执行脚本的事件

当下列事件之一发生时，一个脚本将在图表上的全部条形上执行。

- 在图表上加载一个新的符号或时间框架。
- 从松树编辑器或图表的 "指标和策略" 对话框中保存或添加一个脚本到图表中。
- 在脚本的 "设置/输入" 对话框中修改一个数值。
- 在策略的 "设置/属性" 对话框中修改了一个值。
- 检测到一个浏览器刷新事件。

当交易活跃时，一个脚本在实时条上被执行，并且。

- 上述条件之一发生，导致脚本在实时条形图的打开处执行，或
- 由于检测到价格或成交量的变化，实时条形图被更新。

请注意，当市场处于活跃状态时，如果一个图表没有被触动，一连串已经打开然后关闭的实时条将会跟踪当前的实时条。虽然这些过期的实时条形图已经被确认，因为它们的变量都已经提交，但是脚本还没有在它们的历史\*状态下执行，因为当脚本最后一次在图表的数据集上运行时，它们并不存在。

当一个事件触发了脚本在图表上的执行，并导致它在那些现在已经成为历史条形的条形上运行时，脚本的计算结果有时会与它们在最后一次收盘时的计算结果不同，当时它们还是实时条形。这可能是由于实时条形图收盘时保存的OHLCV值与相同条形图成为历史条形图时从数据馈送中获取的OHLCV值之间的微小差异造成的。这种行为是重绘的可能原因之一。

## 更多信息

- Pine内置的 `barstate.*` 变量提供了关于脚本执行的[条形图类型或事件](#)的信息。记录这些变量的页面还包含一个脚本，可以让你直观地看到实时和历史条形图之间的差异，例如。
- [策略](#)页面解释了策略计算的细节，这与指标的计算不尽相同。

## Pine函数的执行和函数块内的历史背景

Pine函数内部使用的系列变量的历史是通过对函数的每一次连续调用而产生的。如果不在脚本运行的每一个柱状图上调用该函数，这将导致在函数的本地块内部和外部的系列的历史值之间出现差异。因此，如果不在每个柱状图上调用该函数，那么在函数内部和外部使用相同索引值的系列将不会指的是历史上的同一个点。

让我们看看这个脚本的例子，`f()` 和 `f2()` 函数每隔一段时间就被调用。

```

//@version=5
indicator("My Script", overlay = true)

//返回2个柱形前最后一次调用函数时的 "a" 值。
f(a) => a[1].
// 如预期的那样，返回上一栏的 "close" 的值。
f2() => close[1].

oneBarInTwo = bar_index % 2 == 0
plot(oneBarInTwo ? f(close) : na, color = color.maroon, linewidth = 6, style =
plot.style_cross)
plot(oneBarInTwo ? f2() : na, color = color.lime, linewidth = 6, style =
plot.style_circles)
plot(close[2], color = color.maroon)
plot(close[1], color = color.lime)

```



从结果图中可以看出，`a[1]` 返回函数上下文中`a`的前一个值，所以最后一次调用`f()`是在两个柱状体之前--而不是像`f2()`中的`close[1]`那样，是前一个柱状体的收盘价。这导致函数块中的`a[1]`指的是与`close[1]`不同的过去值，尽管它们使用相同的索引1。

## 为什么有这种行为？

之所以需要这种行为，是因为在每个柱形上强制执行函数会导致意想不到的结果，比如在if分支内调用`label.new()`函数，除非`if`条件需要，否则不得执行。

另一方面，这种行为会导致某些内置函数出现意想不到的结果，这些函数需要在每个柱形图上执行以正确计算其结果。如果这些函数被放置在不是每个柱子都要执行的环境中，例如`if`分支，那么它们将不会返回预期的结果。

在这些情况下，解决方案是将这些函数调用放在其上下文之外，这样它们就可以在每个柱状图上执行。

在这个脚本中，`ta.barssince()`不是在每个柱形上都被调用，因为它在一个三元运算符的条件分支中。

```
//@version=5
指标("Barssince", overlay = false)
res = close > close[1] ? ta.barssince(close < close[1]) : -1
plot(res, style = plot.style_histogram, color=res >= 0 ? color.red : color.blue)
```

这导致了不正确的结果，因为[ta.barssince\(\)](#)不是在每个柱子上执行的。



解决办法是在条件分支之外调用[ta.barssince\(\)](#)，以强制在每个柱子上执行。

```
//@version=5
indicator("Barssince", overlay = false)
b = ta.barssince(close < close[1])
res = close > close[1] ? b : -1
plot(res, style = plot.style_histogram, color = res >= 0 ? color.red : color.blue)
```

使用这种技术，我们得到了预期的输出。



## Exceptions

并非所有的内置函数都需要每条都执行。这些是不需要的函数，所以不需要特别处理。

```
dayofmonth, dayofweek, hour, linebreak, math.abs, math.acos, math.asin, math.atan,
math.ceil,
math.cos, math.exp, math.floor, math.log, math.log10, math.max, math.min, math.pow,
math.round,
math.sign, math.sin, math.sqrt, math.tan, minute, month, na, nz, second, str.tostring,
ticker.heikinashi, ticker.kagi, ticker.new, ticker.renko, time, Timestamp, weekofyear,
year
```

### 注意

在[for](#)循环中调用的函数，在循环的每一次迭代中使用相同的上下文。在下面的例子中，对同一个酒吧的每个[ta.lowest\(\)](#)调用都使用传递给它的值，即[bar\\_index](#)，所以循环中使用的函数调用不需要特殊处理。

```
//@version=5
indicator("My Script")
va = 0.0
for i = 1 to 2 by 1
    如果 (i + bar_index) % 2 == 0
        va := ta.lowers(bar_index, 10) //每次调用时都是相同的环境。
plot(va)
```

## 时间序列

Pine的许多功能源于这样一个事实，即它被设计用来有效地处理时间序列。时间序列不是一种形式或类型；它们是Pine用来存储一个变量在一段时间内的连续值的基本结构，其中每个值都与一个时间点相联系。由于图表是由条形图组成的，每个条形图代表一个特定的时间点，因此时间序列是处理可能随时间变化的数值的理想数据结构。

时间序列的概念与Pine的[执行模型](#)和[类型系统](#)概念紧密相连。理解所有这三个概念是充分利用Pine的力量的关键。

以内置的`open`变量为例，它包含了数据集中每个条形的“open”价格，数据集是任何特定图表上的所有条形。如果你的脚本是在5分钟的图表上运行，那么`open`时间序列中的每个值都是连续5分钟图表中的“开盘”价格。当你的脚本提到`open`时，它指的是脚本正在执行的条形图的“开盘”价格。为了引用时间序列中的过去值，我们使用`[]`历史引用操作符。当一个脚本在一个给定的条上执行时，`open[1]`指的是前一个条上的`open`时间序列的值。

虽然时间序列可能会让程序员想起数组，但它们是完全不同的。Pine确实使用了一个数组数据结构，但它与时间序列是完全不同的概念。

Pine中的时间序列，结合其特殊类型的运行时引擎和内置函数，才使得计算`close`值的累积总数变得容易，不需要使用`for`循环，只需要`ta.cum(close)`。这是可能的，因为尽管`ta.cum(close)`在脚本中看起来相当静态，但事实上它是在每个柱子上执行的，所以它的值会随着每个新柱子的`close`值被添加到其中而变得越来越大。当脚本到达图表的最右边时，`ta.cum(close)`返回图表中所有条形的`close`值的总和。

同样，过去14个高和低值之间的差值的平均值可以表示为`ta.sma(high - low, 14)`，或者表示自上次图表连续出现五个高点以来的条形距离为`barssince(rise(high, 5))`。即使是在连续的条形图上调用函数的结果，也会在时间序列中留下数值的痕迹，可以使用`[]`历史参考操作符来引用。这可能很有用，例如，当测试当前条形图的`close`是否突破了过去10个条形图中的最高`high`，但不包括当前条形图，我们可以写成`breach = close > highest(close, 10)[1]`。同样的语句也可以写成`breach = close > highest(close[1], 10)`。

同样的循环逻辑也适用于诸如`plot(open)`这样的函数调用，它将在每个柱子上重复，在图表上连续绘制出每个柱子的`open`的值。

不要将“时间序列”与“序列”形式混淆。[时间序列](#)的概念解释了变量的连续值是如何存储在Pine中的；“系列”形式表示其值可以逐条变化的变量。例如，考虑`timeframe.period`内置变量，它的形式是“simple”，类型是“string”，所以是“simple string”。简单”的形式意味着该变量的值在第0条（脚本执行的第一个条）就已经知道，并且在脚本执行过程中不会在图表的所有条上发生变化。变量的值是以字符串格式表示的图表的时间框架，例如，对于一个1D图表来说，就是“D”。尽管它的值在脚本中不能改变，但在Pine中使用`timeframe.period[10]`来指代10个条形前的值在语法上是正确的（尽管不是很有用）。这是有可能的，因为每个柱子的`timeframe.period`的连续值被存储在一个时间序列中，尽管该特定时间序列中的所有值都是相似的。然而，请注意，当`[]`操作符被用来访问一个变量的过去值时，它产生的结果是“系列”形式，即使没有偏移的变量是另一种形式，比如在`timeframe.period`的情况下是“简单”。

当你掌握了如何使用Pine的语法和它的[执行模型](#)有效地处理时间序列时，你可以用很少的代码来定义复杂的计算。

## 脚本结构

Pine中的脚本遵循这个一般结构：

```
<version>
<declaration_statement>
<code>
```

# 注释

双斜线（//）定义了Pine的注释。注释可以在行的任何地方开始。除了在被包裹的行上，它们也可以在同一行上跟随Pine代码。

```
//@version=5
indicator("")
// This line is a comment
a = close // This is also a comment
plot(a)
```

Pine编辑器有一个评论/取消评论行的键盘快捷键：ctrl + /，你可以在多行上使用它，首先突出显示它们。

# 版本

以下形式的编译器指令告诉编译器该脚本是用哪个版本的Pine编写的：

```
//@version=5
```

- 版本号可以是1到5。
- 编译器指令不是强制性的，但如果省略的话，就会假定版本1。强烈建议总是使用最新的版本。
- 虽然把版本指令放在脚本的任何地方在协同上都是正确的，但如果放在脚本的顶部，对读者来说更有用。

当前版本的Pine的显著变化记录在[发行说明](#)中。

# 声明语句

所有的Pine脚本都必须包含一个声明语句，也就是对这些函数之一的调用。

- [indicator\(\)](#)
- [strategy\(\)](#)
- [library\(\)](#)

声明语句。

- 确定脚本的类型，这又决定了其中允许哪些内容，以及如何使用和执行。
- 设置脚本的关键属性，比如它的名称，当它被添加到图表中时，它将出现在哪里，它所显示的数值的精度和格式，以及管理其运行时行为的某些数值，比如它将在图表中显示的最大绘图对象数量。对于策略，属性包括控制回测的参数，如初始资本、佣金、滑点等。

每种类型的脚本都有不同的要求。

- 指标必须包含至少一个在图表上产生输出的函数调用（例如，[plot\(\)](#), [plot\(\)](#), [plotshape\(\)](#), [barcolor\(\)](#), [line.new\(\)](#)，等等）。
- 策略必须包含至少一个 "strategy.\*()" 调用，例如，[strategy.entry\(\)](#)。
- 库必须至少包含一个库函数声明。

# 代码

脚本中不是注释或编译器指令的行是语句，它实现了脚本的算法。一个语句可以是这些内容之一。

- 变量声明
- 变量的重新赋值
- 函数声明
- 内置函数调用，[用户定义的函数调用](#)或[一个库函数调用](#)
- [if](#), [for](#), [while](#) 或 [switch](#) 结构。

语句可以以多种方式排列。

- 有些语句可以用一行来表达，比如大多数变量声明、只包含一个函数调用的行或单行函数声明。其他的，像结构，总是需要多行，因为它们需要一个局部的块。
- 脚本的全局范围内的语句（即不属于局部块的部分）不能以白色空间（空格或制表符）开始。它们的第一个字符也必须是该行的第一个字符。在行的第一个位置开始的行，根据定义成为脚本的全局范围的一部分。
- 结构或多行函数声明总是需要一个*local block*。一个本地块必须缩进一个制表符或四个空格。每个局部块定义了一个不同的局部范围。
- 多个单行语句可以通过使用逗号（,）作为分隔符在一行中串联起来。
- 行可以包含注释，也可以是注释。
- 行也可以被包起来（在多行上继续）。

通过使用 "打开" 按钮并选择 "新建空白指标"，可以在Pine编辑器中生成一个简单有效的Pine v5指标。

```
//@version=5
indicator("My Script")
plot(close)
```

这个指标包括三个局部块，一个在 `f()` 函数声明中，两个在变量声明中使用[if](#)结构。

```
//@version=5
indicator("", "", true) //声明语句(全局范围)

barIsUp() => // 函数声明(全局范围)
    close > open // 本地块(本地范围)

plotColor = if barIsUp() // 变量声明 (全局范围)
    color.green // 本地块 (本地范围)
else
    color.red // 本地块 (本地范围)

bgcolor(color.new(plotColor, 70)) // 调用一个内置函数 (全局范围)
```

你可以通过选择 "新建空白策略" 来调出一个简单的Pine v5策略。

```
//@version=5
strategy("My Strategy", overlay=true, margin_long=100, margin_short=100)

longCondition = crossover(sma(close, 14), sma(close, 28))
if(longCondition)
    strategy.entry("My Long Entry Id", strategy.long)

shortCondition = crossunder(sma(close, 14), sma(close, 28))
if(shortCondition)
    strategy.entry("My Short Entry Id", strategy.short)
```

## 换行代码

长行可以被分割在多行上，或被 "包裹"起来。被包裹的行必须缩进任何数量的空格，只要它不是4的倍数（这些边界用于缩进局部块）。

```
a = open + high + low + close
```

可以被包装成：

```
a = open +
    high +
    low +
    close
```

一个长的[plot\(\)](#)调用可以被包装成。

```
plot(ta.correlation(src, ovr, length),
      color = color.new(color.purple, 40),
      style = plot.style_area,
      trackprice = true)
```

用户定义的函数声明中的语句也可以被包装。但是，由于局部块在语法上必须以缩进开始（4个空格或1个制表符），当把它分割到下一行时，语句的延续部分必须以一个以上的缩进开始（不等于4个空格的倍数）。比如说

```

updown(s) =>
    isEqual = s == s[1] .
    isGrowing = s > s[1] .
    ud = isEqual ?
        0 :
        isGrowing ?
            (nz(ud[1]) <= 0 ?
                1 :
                nz(ud[1])+1) :
            (nz(ud[1]) >= 0 ?
                -1 :
                NZ(UD[1])-1)

```

不要在换行代码中使用注释。下面的代码不会被编译。

```

//@version=5
indicator("")
c = open > close ? color.red :
    high > high[1] ? color.lime : // 一个注释导致编译错误。
    low < low[1] ? color.blue : color.black
bgcolor(c)

```

## 标识符

标识符是用于用户定义的变量和函数的名称。

- 它们必须以大写字母(A-Z)或小写字母(a-z)开头，或下划线(\_)。
- 接下来的字符可以是字母、下划线或数字(0-9)。
- 它们是区分大小写的。

下面是一些例子。

```

myVar
_myVar
my123Var
functionName
MAX_LEN
max_len
maxLen
3barsDown // 不是有效的!

```

[Pine Style Guide](#) 官方建议对常量使用大写的蛇形，对其他标识符使用驼峰写法。

```
GREEN_COLOR = #4CAF50
MAX_LOOKBACK = 100
int fastLength = 7
// 如果参数是`true`，则返回1，如果是`false`或`na`，则返回0。
zeroOne(boolValue) => boolValue ? 1 : 0
```

# 运算符

## 简介

一些运算符被用来建立表达式，并返回一个结果。

- 算术运算符
- 比较运算符
- 逻辑运算符
- [?: 三元运算符。](#)
- [历史参考运算符](#)

其他运算符用于为变量赋值。

- `=` 用来给变量赋值，但只在声明变量时（第一次使用时）。
- `:=` 用来给一个\*\*以前声明的变量赋值。下列运算符也可以这样使用：`+=`, `-=`, `*=`, `/=`, `%=`。

正如在[类型系统](#)页面中解释的那样，形式和类型在决定表达式产生的结果的类型方面起着关键作用。这反过来又影响到你将被允许使用这些结果的方式和函数。表达式总是返回表达式中使用的最强的一种形式，例如，如果你将一个 "input int" 与一个 "series int" 相乘，表达式将产生一个 "series int" 的结果，你将不能使用它作为[ta.ema\(\)](#)中 `length` 的参数。

这个脚本会产生一个编译错误。

```
//@version=5
indicator("")
lenInput = input.int(14, "Length")
factor = year > 2020 ? 3 : 1
adjustedLength = lenInput * factor
ma = ta.ema(close, adjustedLength) // 编译错误!
plot(ma)
```

编译器会报错：无法调用'ta.ema'，参数'length'='adjustedLength'。使用了一个'series int'类型的参数，但预期是一个'简单int'。这是因为 `lenInput` 是一个 "input int"，但 `factor` 是一个 "series int"（它只能通过查看每个柱状图上的`year`的值来确定）。因此，`adjustedLength` 变量被分配了一个 "series int" 值。我们的问题是[ta.ema\(\)](#)的参考手册条目告诉我们，它的 `length` 参数需要 "simple" 形式的值，这是一个比 "series" 更弱的形式，所以 "series int" 值是不允许的。

解决我们的难题需要。

- 使用另一个支持 "series int" 长度的移动平均线函数，例如[ta.sma\(\)](#)，或者
- 不使用为我们的长度产生 "series int" 值的计算。

## 算术运算符

在Pine Script中，有五个算术运算符。

<code>+</code>	加法和字符串连接
<code>-</code>	减法
<code>*</code>	乘法运算
<code>/</code>	除法
<code>%</code>	Modulo (除法后的余数)

上面的算术运算符都是二进制的（意味着它们需要两个操作数来工作，如 "1+2"）。+和-也是单数运算符（意味着它们对一个操作数工作，如-1或+1）。

如果两个操作数都是数字，但其中至少有一个是 "float" 类型，结果也将是 "float"。如果两个操作数都是 "int" 类型，结果也将是 "int"。如果至少有一个操作数是[na](#)，其结果也是[na](#)。

+ 运算符也可以作为字符串的连接运算符。`"EUR "+"USD"` 得到 `"EURUSD"` 字符串。

## 比较运算符

在Pine Script中，有六个比较运算符。

<code>&lt;</code>	小于
<code>&lt;=</code>	小于或等于
<code>!=</code>	不等于
<code>==</code>	等于
<code>&gt;</code>	大于
<code>&gt;=</code>	大于或等于

比较操作是二进制的。如果两个操作数都是数值，结果将是 `bool` 类型，即 "true"、"false" 或 [na](#)。

例子。

```
1 > 2 // false
1 != 1 // false
close >= open // 取决于`close`和`open`的值。
```

## 逻辑运算符

在Pine Script中，有三个逻辑运算符。

<code>not</code>	<b>Negation</b>
<code>and</code>	逻辑连接体
<code>or</code>	逻辑分离

操作符 "not" 是单项的。当应用于一个 "真" 的操作数时，结果是 "假"，反之亦然。

和 运算符真值表。

<b>a</b>	<b>b</b>	<b>a and b</b>
true	true	true
true	false	false
false	true	false
false	false	false

`or` 运算符真值表。

<b>a</b>	<b>b</b>	<b>a or b</b>
true	true	true
true	false	true
false	true	true
false	false	false

## ? : 三元运算符

? : 三元运算符用于创建形式的表达：

条件 ? 如果为真返回值 : 如果为假返回值

三元运算符返回的结果取决于 条件 的值。如果它是 "真"，那么返回 第二个参数的值。如果 条件 是 假 或 `na`，那么将返回 第三个参数的值。

三元表达式的组合可以达到与 [switch](#) 结构相同的效果，例如：

```
timeframe.isintraday ? color.red : timeframe.isdaily ? color.green :  
timeframe.ismonthly ? color.blue : na
```

这个例子从左到右计算：

- 如果`timeframe.isintraday`是`true`，那么就会返回`color.red`。如果是`"false"`，那么`timeframe.isdaily`被评估。
- 如果`timeframe.isdaily`为`true`，则返回`color.green`。如果是`false`，则评估`timeframe.ismonthly`。
- 如果`timeframe.ismonthly`为`true`，则返回`color.blue`，否则返回`na`。

## [ ] 历史引用操作符

可以使用`[]`历史引用操作符来引用[时间序列](#)的过去值。过去的数值是指变量在当前脚本执行的条形图之前的数值，即当前条形图。更多关于脚本的执行方式的信息，请参见[执行模式](#)页面。

在变量、表达式或函数调用之后使用`[]`操作符。操作符的方括号内使用的值是我们要引用的过去的偏移量。要引用`volume`内置变量的值，离当前柱状体有两个柱状体，我们可以使用`volume[2]`。

因为序列是动态增长的，当脚本在连续的柱状体上移动时，与操作符一起使用的偏移量将指代不同的柱状体。让我们看看同一个偏移量返回的值是如何动态的，以及为什么系列与数组有很大的不同。在Pine中，`close`变量，或`close[0]`相当于，持有当前条形的“收盘”值。如果你的代码现在是在`dataset`（图表上所有条形的集合）的\*第三条上执行，`close`将包含该条形收盘时的价格，`close[1]`将包含前一条形（数据集的第二条形）收盘时的价格，`close[2]`是第一条形。`close[3]`将返回`na`，因为该位置不存在条形图，因此其值是不可用的。

当同样的代码在下一个条形图上执行时，即数据集中的\*\*第四个条形图，`close``现在将包含该条形图的收盘价，在你的代码中使用的同样的`close[1]`现在将指数据集中的第三个条形图的“收盘”。数据集中第一个条形图的收盘价现在将是`close[3]`，而这次`close[4]``将返回`na`。

在Pine的运行环境中，由于你的代码对数据集中的每个历史条形图都要执行一次，从图表的左边开始，Pine在索引0处增加了一个新的系列元素，并将系列中已有的元素推远了一个索引。相比之下，数组可以有恒定或可变的大小，其内容或索引结构不会被运行时环境修改。因此，松系列与数组有很大的不同，只是通过其索引语法与数组共享熟悉感。

当图表的符号的市场是开放的，脚本在图表的最后一个条形上执行时，[实时条形](#)，`close`返回当前价格的值。它将只包含实时条的实际收盘价，即最后一次在该条上执行脚本时的收盘价。

Pine有一个变量，包含了脚本所执行的条形图的编号。`bar_index`。在第一个条形图上，`bar_index`等于0，在脚本执行的每个连续条形图上，它都会增加1。在最后一个条形图上，`bar_index`等于数据集中条形图的数量减去1。

在Pine中使用“`[]`”操作符时，还有一个重要的考虑因素要牢记。我们看到的情况是，一个历史参考可能会返回`na`的值。`na`代表一个不是数字的值，在任何表达式中使用它都会产生一个同样是`na`的结果（类似于`Nan`）。这种情况经常发生在数据集的早期条形图的脚本计算过程中，但在某些条件下也可能发生在后期条形图中。如果你的Pine代码没有明确规定如何处理这些特殊情况，它们会在你的脚本计算中引入无效的结果，这些结果会一直影响到实时条。`na`和`nz`函数被设计为允许处理这种情况。

这些都是`[]`运算符的有效使用。

```
high[10]
ta.sma(close, 10)[1]
ta.highest(high, 10)[20]
close > nz(close[1], open)
```

请注意，`[]`操作符只能在同一个值上使用一次。这是不允许的。

```
close[1][2] // 错误：不正确使用 [] 操作符
```

## 运算符优先级

计算的顺序是由运算符的优先级决定的。优先级较高的运算符会先被计算。下面是一个按优先级递减排序的运算符列表。

优先级	操作符
9	<code>[]</code>
8	<code>+, -, not</code>
7	<code>*, %</code>
6	<code>+, -</code>
5	<code>&gt;, &lt;, &gt;=, &lt;=</code>
4	<code>==, !=</code>
3	<code>and</code>
2	<code>or</code>
1	<code>? :</code>

如果在一个表达式中，有几个具有相同优先级的运算符，那么它们将被从左到右计算。

如果表达式必须以不同于优先级的顺序进行计算，那么表达式的部分内容可以用圆括号分组。

## = 赋值运算符

`=` 操作符用于在初始化--或声明--时分配一个变量，即第一次使用它时。它说这是我要使用的一个新变量，我希望它在每个柱子上以这个值开始。

这些都是有效的变量声明。

```
i = 1
ms_in_one_minute = 1000 * 60
showPlotInput = input.bool(true, "Show plots")
pHi = pivothigh(5, 5)
plotColor = color.green
```

关于如何声明变量的更多信息，请参见[变量声明](#)页面。

## := 重新赋值运算符

`:=`是用来为一个现有的变量重新赋值的。它说使用这个在我的脚本中早先声明的变量，并给它一个新的值。

首先声明的变量，然后用`:=`重新赋值的变量，被称为可变变量。下面所有的例子都是有效的变量重新赋值。你可以在[ref` var 声明模式 <PageVariableDeclarations\\_Var>](#)一节中找到更多关于[var](#)如何工作的信息。

```
//@version=5
indicator("", "", true)
// Declare `pHi` and initialize it on the first bar only.
var float pHi = na
// Reassign a value to `pHi`
pHi := nz(ta.pivothigh(5, 5), pHi)
plot(pHi)
```

请注意。

- 我们用这个代码声明`pHi`。`var float pHi = na`。`var`关键字告诉Pine，我们只想在数据集的第一个条形图上用`na`来初始化该变量。`float`关键字告诉编译器我们正在声明一个"float"类型的变量。这是必要的，因为与大多数情况相反，编译器不能自动确定`=`符号右边的数值的类型。
- 虽然变量声明只在第一个条形图上执行，因为它使用了`var`，但`pHi := nz(ta.pivothigh(5, 5), pHi)`一行将在图表的所有条形图上执行。在每个条形图上，如果`pivothigh()`调用返回`na`，它就会进行评估，因为这是该函数在没有找到新枢轴时的做法。`nz()`函数是做"检查`na`"的部分。当它的第一个参数`(ta.pivothigh(5, 5))`是`na`时，它返回第二个参数（`pHi`）而不是第一个。当`pivothigh()`返回新发现的枢轴的价格点时，该值被分配给`pHi`。当它返回`na`时，因为没有找到新的枢轴，我们将`pHi`的先前值分配给它自己，实际上是保留了它先前的值。

我们脚本的输出看起来像这样。



请注意。

- 在发现新的枢轴之前，该线保留其先前的值。
- 枢轴是在枢轴实际发生后的五个小节后被检测到的，因为我们的 `ta.pivothigh(5, 5)` 调用说我们需要在一个高点的两边有五个较低的高点才能被检测为枢轴。

参见[变量重新赋值](#)部分，了解更多关于如何为变量重新赋值的信息。

## 变量声明

### 介绍

变量是保存值的[标识符](#)。在使用之前，必须在代码中对它们进行声明。变量声明的语法如下：

```
[<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
```

或者

```
<tuple_declaration> = <function_call> | <structure>
```

其中：

- `|` 表示“或”，方括号 (`[]`) 中的部分可以出现零次或一次。
- `<declaration_mode>` 是变量的[声明模式](#)。可以是 `var` 或 `varip`，或者为空。
- `<type>` 是可选的，几乎所有 Pine Script™ 变量声明都是可选的（参见[类型](#)）。
- `<identifier>` 是变量的[名称](#)。
- `<expression>` 可以是文字、变量、表达式或函数调用。
- `<structure>` 可以是 `if`、`for`、`while` 或 `switch` 结构。
- `<tuple_declaration>` 是用方括号 (`[]`) 括起的以逗号分隔的变量名称列表，例如 `[ma, upperBand, lowerBand]`。

以下是所有有效的变量声明。最后一个需要四行：

```

BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
var barRange = float(na)
var firstBarOpen = open
varip float lastClose = na
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red

```

## 注意

上述语句都包含 `=` 赋值运算符，因为它们是变量声明。当看到类似的行使用 `=` 重新分配运算符时，代码正在重新分配一个已经声明的变量的值。这是对 Pine Script™ 新手来说常见的障碍，请确保您理解这个区别。详细信息请参阅下一个[变量重新分配](#)部分。

变量声明的形式语法为：

```

<variable_declarator>
  [<declaration_mode>] [<type>] <identifier> = <expression> | <structure>
  |
<tuple_declarator> = <function_call> | <structure>

<declaration_mode>
  var | varip

<type>
  int | float | bool | color | string | line | linefill | label | box | table |
  array<type> | matrix<type> | UDF

```

## 使用 `na` 进行初始化

在大多数情况下，显式类型声明是多余的，因为类型会在编译时从等号右边的值中自动推断出来，所以使用它们通常是个喜好的问题。例如：

```

baseLine0 = na          // 编译时错误!
float baseLine1 = na    // OK
baseLine2 = float(na)   // OK

```

在示例的第一行中，编译器无法确定 `baseLine0` 变量的类型，因为 `na` 是一个没有特定类型的通用值。`baseLine1` 变量的声明是正确的，因为显式声明了其 `float` 类型。`baseLine2` 变量的声明也是正确的，因为其类型可以从表达式 `float(na)` 推导出来，这是 `na` 值到 `float` 类型的显式转换。`baseLine1` 和 `baseLine2` 的声明是等价的。

## 元组声明

函数调用或结构允许返回多个值。当我们调用它们并想要存储它们返回的值时，必须使用元组声明，它是由括号括起的一个或多个值的逗号分隔集合。这允许我们同时声明多个变量。例如，用于 Bollinger 带的内置函数 `ta.bb()` 返回三个值：

```
[bbMiddle, bbUpper, bbLower] = ta.bb(close, 5, 4)
```

## 变量重新分配

变量重新分配是使用 `=` 重新分配运算符完成的。只能在变量首次声明并赋予初始值之后进行重新分配。在计算中经常需要重新分配一个变量的新值，当一个全局作用域中的变量需要从结构的本地块内赋予新值时，这是必需的。例如：

```
//@version=5
indicator("", "", true)
sensitivityInput = input.int(2, "Sensitivity", minval = 1, tooltip = "Higher values
make color changes less sensitive.")
ma = ta.sma(close, 20)
maUp = ta.rising(ma, sensitivityInput)
maDn = ta.falling(ma, sensitivityInput)

// 仅在第一根K线上，将颜色初始化为灰色
var maColor = color.gray
if maUp
    // MA 上涨了两根K线；将其设为绿色。
    maColor := color.lime
else if maDn
    // MA 下跌了两根K线；将其设为品红色。
    maColor := color.fuchsia

plot(ma, "MA", maColor, 2)
```

注意：

- 我们仅在第一根K线上初始化 `maColor`，因此它在各个K线之间保持其值不变。
- 在每根K线上，`if` 语句检查MA是否在用户指定的K线数（默认为2）上涨或下跌。当发生这种情况时，必须使用 `if` 本地块内的新值重新分配 `maColor` 的值。为此，我们使用了 `=` 重新分配运算符。
- 如果不使用 `=`，效果将是初始化一个新的 `maColor` 本地变量，该变量与全局作用域的同名变量相同，但实际上是一个非常令人困惑的独立实体，仅在本地块的长度内存在，然后在没有痕迹的情况下消失。

在 Pine Script™ 中，所有用户定义的变量都是可变的，这意味着它们的值可以使用 `=` 重新分配运算符进行更改。在一根K线上，一个变量可以被分配新值多次，因此一个脚本可以包含对一个变量的任意次数的重新分配。变量的 [声明模式](#) 决定了如何保存分配给变量的新值。

## 声明模式

理解声明模式对变量行为的影响需要先了解 Pine Script™ 的[执行模型](#)。

当声明变量时，如果指定了声明模式，它必须首先出现。可以使用三种模式：

- 在没有指定时是“每根K线”，即没有使用 `var` 或 `varip` 关键字。
- `var`
- `varip`

## 每根K线

当没有显式声明模式时，即没有使用 `var` 或 `varip` 关键字时，该变量会在每根K线上进行声明和初始化，例如，此页面介绍的第一组示例中的以下声明：

```
BULL_COLOR = color.lime
i = 1
len = input(20, "Length")
float f = 10.5
closeRoundedToTick = math.round_to_mintick(close)
st = ta.supertrend(4, 14)
[macdLine, signalLine, histLine] = ta.macd(close, 12, 26, 9)
plotColor = if close > open
    color.green
else
    color.red
```

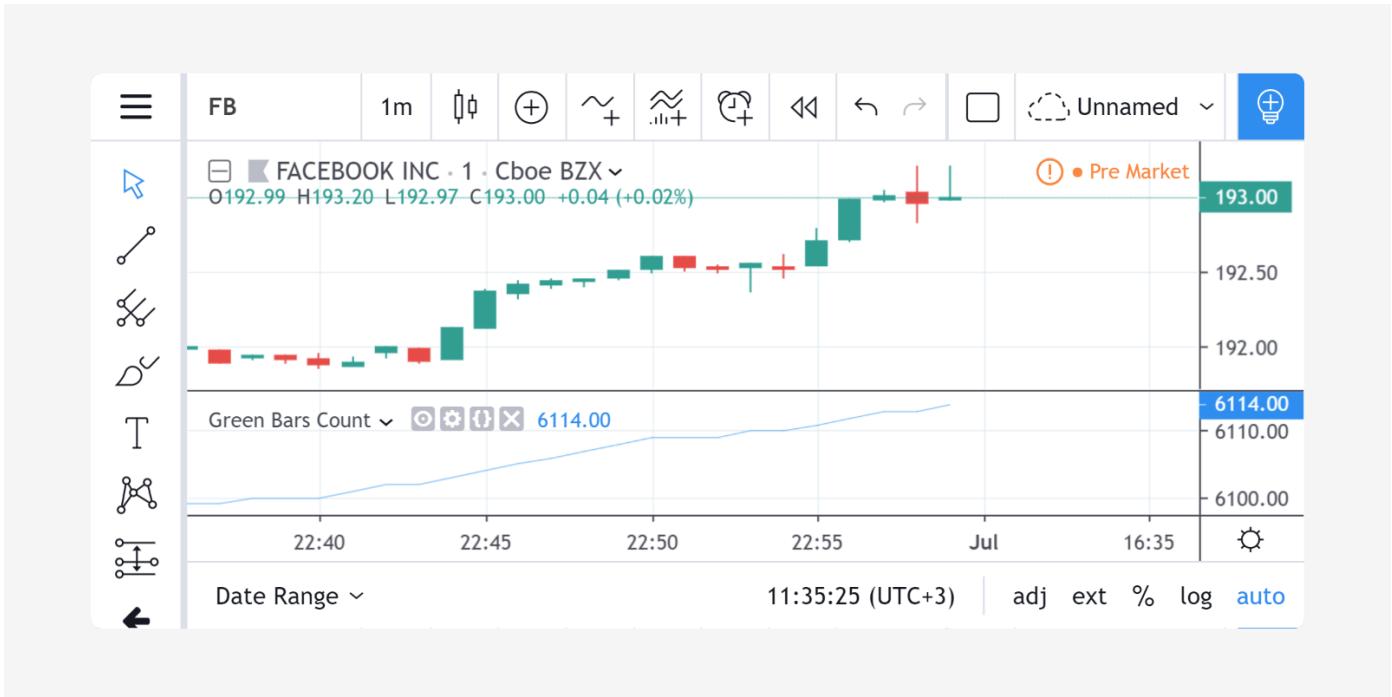
## var

当使用 `var` 关键字时，变量仅在第一根K线上初始化，如果声明在全局作用域中，或者在局部块中的声明在执行局部块的第一次时。之后，它将保持其在连续的K线上的最后一个值，直到我们重新分配新值给它。这种行为在许多情况下都很有用，其中变量的值必须在脚本在连续的K线上的迭代中保持不变。例如，假设我们想要统计图表上绿色K线的数量：

```
//@version=5
indicator("绿色K线计数")
var count = 0
isGreen = close >= open
if isGreen
    count := count + 1
plot(count)
```

如果没有 `var` 修饰符，变量 `count` 将在每次新的K线触发脚本重新计算时重置为零，从而丢失其值。

```
if isGreen
    count := count + 1
plot(count)
```



使用 `var` 修饰符，`count` 将保持在连续K线上的最后一个值，不会在每次脚本重新计算时重置为零。

```
//@version=5
indicator("绿色K线计数")
var count = 0
isGreen = close >= open
if isGreen
    count := count + 1
plot(count)
```

这样，`count` 的值将在每个新K线上追踪绿色K线的数量。这种行为在许多场景中很有用，例如跟踪满足某些条件的事件发生的次数。

## varip

理解使用 `varip` 声明模式的变量行为需要先了解 Pine Script™ 的[执行模型](#)和[栏状态](#)。

`varip` 关键字可用于声明变量，这些变量逃离回滚过程，这在 Pine Script™ 的[执行模型](#)页面中有解释。

虽然脚本仅在历史K线的收盘时执行一次，但在实时运行时，每当图表的数据源检测到价格或成交量更新时，脚本就会执行一次。在每个实时更新时，Pine Script™ 的运行时通常会将脚本变量的值重置为它们的上一个提交的值，即在前一根K线关闭时它们的值。这通常很方便，因为每个实时脚本执行都从一个已知状态开始，从而简化了脚本逻辑。

然而，有时脚本逻辑要求代码能够在实时K线的不同执行之间保存变量值。使用 `varip` 声明变量使这成为可能。`varip` 中的 "ip" 代表 *intrabar persist*。

让我们看看以下的代码，它不使用 `varip`：

```
//@version=5
indicator("")
int updateNo = na
if barstate.isnew
    updateNo := 1
else
    updateNo := updateNo + 1

plot(updateNo, style = plot.style_circles)
```

在历史K线上，`barstate.isnew` 总是为 `true`，因此绘图显示值为 "1"，因为 `if` 结构的 `else` 部分永远不会执行。在实时K线上，只有当脚本首次在K线的“开盘”上执行时，`barstate.isnew` 才为 `true`。然后，绘图将会短暂地显示 "1"，直到后续执行发生。在实时脚本执行的后续执行期间，`barstate.isnew` 不再为 `true`。由于 `updateNo` 在每次执行时都初始化为 `na`，`updateNo + 1` 表达式将产生 `na`，因此在脚本的后续实时执行中不会绘制任何内容。

现在，如果我们使用 `varip` 来声明 `updateNo` 变量，脚本的行为将大不相同：

```
//@version=5
indicator("")
varip int updateNo = na
if barstate.isnew
    updateNo := 1
else
    updateNo := updateNo + 1

plot(updateNo, style = plot.style_circles)
```

现在的区别在于 `updateNo` 跟踪每个实时K线上发生的实时更新次数。这是可能的，因为 `varip` 声明允许 `updateNo` 的值在实时更新之间保持，不再在脚本的每次实时执行时被回滚。对 `barstate.isnew` 的测试允许我们在新的实时K线到来时重置更新计数。

由于 `varip` 仅影响脚本在实时K线上的行为，因此基于 `varip` 变量的逻辑设计的策略的回测结果将无法在历史K线上重现该行为，这将使其在测试结果上失效。这也意味着在历史K线上，图表将无法复现脚本在实时执行时的行为。

## 条件结构

- [介绍](#)
- `if` 结构
  - [`if` 用于它的副作用](#)
  - [`if` 用于返回一个值](#)
- ‘开关’结构
  - [带有表达式的“switch”](#)
  - [不带表达式的“switch”](#)

- 匹配本地块类型要求

## 介绍

Pine Script™ 中的条件结构是 `if` 和 `switch`。它们可以用于：

- 对于它们的副作用，即当它们不返回值但执行某些操作时，例如将值重新分配给变量或调用函数。
- 返回一个值或一个元组，然后可以将其分配给一个（或多个，如果是元组）变量。

条件结构，如 `for` 和 `while` 结构，可以嵌入；您可以在另一个结构中使用 `if` 或 `switch`。

某些 Pine Script™ 内置函数无法从条件结构的本地块内调用。它们是：`alertcondition()`、`barcolor()`、`fill()`、`hline()`、`indicator()`、`library()`、`plot()`、`plotbar()`、`plotcandle()`、`plotchar()`、`plotshape()`、`strategy()`。这并不意味着它们的功能不能通过脚本评估的条件来控制——只是不能通过将它们包含在条件结构中来完成。请注意，虽然 `input*.*()` 本地块中允许函数调用，但它们的功能与脚本全局范围内的功能相同。

条件结构中的局部块必须缩进四个空格或制表符。

## `if` 结构

### `if` 用于它的副作用

用于其副作用的 `if` 结构具有以下语法：

```
if <expression>
  <local_block>
{else if <expression>
  <local_block>}
[else
  <local_block>]
```

在哪里：

- 方括号 (`[]`) 中的部分可以出现零次或一次，大括号 (`{}`) 中的部分可以出现零次或多次。
- <表达式> 必须是“bool”类型或者可以自动转换为该类型，这仅适用于“int”或“float”值（请参阅[类型系统](#)页面）。
- <local\_block> 由零个或多个语句组成，后跟一个返回值，该返回值可以是值的元组。它必须缩进四个空格或制表符。
- 可以有零个或多个子句。`else if`
- 可以有零个或一个 `else` 子句。

当 `if` 后面的 计算结果为 `true` 时，将执行第一个本地块，`if` 结构的执行结束，并返回在本地块末尾计算的值。

当 `if` 后面的 计算结果为 `false` 时，将计算后续子句（如果有）。当 `one` 的 计算结果为 `true` 时，执行其本地块，`if` 结构的执行结束，并返回在本地块末尾计算的值。`else if`

当没有 `true` 并且 `else` 存在子句时，将执行其本地块，`if` 结构的执行结束，并返回在本地块末尾计算的值。

当没有 的计算结果为`true` 并且不 `else` 存在子句时，将返回`na`。

例如，使用`if` 结构的副作用对于管理策略中的订单流很有用。虽然通常可以使用调用 `when` 中的参数 来实现相同的功能 `strategy.*()`，但使用`if` 结构的代码更易于阅读：

```
if (ta.crossover(source, lower))
    strategy.entry("BBandLE", strategy.long, stop=lower,
                  oca_name="BollingerBands",
                  oca_type=strategy.oca.cancel, comment="BBandLE")
else
    strategy.cancel(id="BBandLE")
```

可以使用 `if` 结构将代码的执行限制在特定的柱上，就像我们在这里将标签的更新限制在图表的最后一个柱上一样：

```
//@version=5
indicator("", "", true)
var ourLabel = label.new(bar_index, na, na, color = color(na), textcolor =
color.orange)
if barstate.islast
    label.set_xy(ourLabel, bar_index + 2, h12[1])
    label.set_text(ourLabel, str.tostring(bar_index + 1, "# bars in chart"))
```

注意：

- `ourLabel` 我们仅在脚本的第一个栏上 初始化变量，因为我们使用`var`声明模式。用于初始化变量的值由 `label.new()` 函数调用提供，该函数返回一个指向它创建的标签的标签 ID。我们使用该调用来设置标签的属性，因为一旦设置，它们将持续存在，直到我们更改它们为止。
- 接下来发生的是，在每个连续的柱上，Pine Script™ 运行时将跳过 的初始化 `ourLabel`，并评估`if` 结构的条件 (`barstate.islast`)。它在所有柱上返回 `false`，直到最后一个柱，因此脚本在零柱之后的大多数历史柱上不执行任何操作。
- 在最后一个柱上，`barstate.islast` 变为 `true` 并且结构的本地块执行，在每个图表上修改更新我们的标签的属性，该属性显示数据集中的柱数。
- 我们希望在没有背景的情况下显示标签的文本，因此我们在`label.new()` 函数调用中将标签的背景设置为`na`，并用于标签的y位置，因为我们不希望它一直移动。通过使用前一个柱的高值 和低值 的平均值，标签不会移动，直到下一个实时柱打开。`h12[1]`
- 我们在`label.set_xy()` 调用中使用 将标签向右偏移两个条。`bar_index + 2`

## if 用于返回一个值

用于返回一个或多个值的`if`结构 具有以下语法：

```
[<declaration_mode>] [<type>] <identifier> = if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

在哪里：

- 方括号 ( [ ]) 中的部分可以出现零次或一次，大括号 ( {} ) 中的部分可以出现零次或多次。
- <declaration\_mode> 是变量的[声明模式](#)
- 是可选的，就像在几乎所有 Pine Script™ 变量声明中一样（请参阅[types](#)）
- 是变量的[名称](#)
- <表达式> 可以是文字、变量、表达式或函数调用。
- <local\_block> 由零个或多个语句组成，后跟一个返回值，该返回值可以是值的元组。它必须缩进四个空格或制表符。
- 分配给该变量的值是 <local\_block> 的返回值，如果没有执行本地块，则为[na](#)。

这是一个例子：

```
//@version=5
indicator("", "", true)
string barState = if barstate.islastconfirmedhistory
    "islastconfirmedhistory"
else if barstate.isnew
    "isnew"
else if barstate.isrealtime
    "isrealtime"
else
    "other"

f_print(_text) =>
    var table _t = table.new(position.middle_right, 1, 1)
    table.cell(_t, 0, 0, _text, bgcolor = color.yellow)
f_print(barState)
```

可以省略 `else` 块。在这种情况下，如果 `condition` 为 `false`，则会将空值 (`na`、`false` 或 `" "`) 分配给该 `var_declarationx` 变量。

这是一个示例，显示当没有执行本地块时如何返回 `na`。如果在这里，则返回 `na`：`close > open``false`

```
x = if close > open
    close
```

脚本可以包含 `if` 嵌套结构 `if` 和其他条件结构。例如：

```
if condition1
    if condition2
        if condition3
            expression
```

但是，从性能角度来看，不建议嵌套这些结构。如果可能，通常更优化的做法是使用多个逻辑运算符组成单个 `if` 语句，而不是多个嵌套 `if` 块：

```
if condition1 and condition2 and condition3
    expression
```

## switch 结构

开关 结构有两种形式 可以切换键表达式的不同值：

```
[ [ <declaration_mode> ] [ <type> ] <identifier> = ]switch <expression>
    {<expression> => <local_block>}
    => <local_block>
```

另一种形式不使用表达式作为键；它打开不同表达式的求值：

```
[ [ <declaration_mode> ] [ <type> ] <identifier> = ]switch
    {<expression> => <local_block>}
    => <local_block>
```

在哪里：

- 方括号 (`[]`) 中的部分可以出现零次或一次，大括号 (`{}`) 中的部分可以出现零次或多次。
- `<declaration_mode>` 是变量的[声明模式](#)
- 是可选的，就像在几乎所有 Pine Script™ 变量声明中一样（请参阅[types](#)）
- 是变量的[名称](#)
- `<表达式>` 可以是文字、变量、表达式或函数调用。
- `<local_block>` 由零个或多个语句组成，后跟一个返回值，该返回值可以是值的元组。它必须缩进四个空格或制表符。
- 分配给该变量的值是 `<local_block>` 的返回值，如果没有执行本地块，则为[na](#)。
- 最后的允许您指定一个返回值，该返回值充当结构中没有执行其他情况时使用的默认值。`=>`  
`<local_block>`

仅执行 `switch` 结构的一个本地块。因此，它是一个不会因情况而异的\*\*结构化交换机。因此，声明是不必要的。`break`

这两种形式都可以作为用于初始化变量的值。

与 [if](#) 结构一样，如果没有执行任何本地块，则返回 [na](#)。

## 带有表达式的 [switch](#)

让我们看一个 使用表达式的 [switch](#)示例：

```
//@version=5
indicator("Switch using an expression", "", true)

string maType = input.string("EMA", "MA type", options = ["EMA", "SMA", "RMA", "WMA"])
int maLength = input.int(10, "MA length", minval = 2)

float ma = switch maType
    "EMA" => ta.ema(close, maLength)
    "SMA" => ta.sma(close, maLength)
    "RMA" => ta.rma(close, maLength)
    "WMA" => ta.wma(close, maLength)
=>
    runtime.error("No matching MA type found.")
    float(na)

plot(ma)
```

注意：

- 我们要打开的表达式是变量，它是“input int”类型（有关“[input](#) maType”限定符是什么的解释，请参见此处）。由于它在脚本执行期间无法更改，这保证了用户选择的任何 MA 类型都将在每个柱上执行，这是像 [ta.ema\(\)](#)这样的函数的要求，这些函数需要一个“简单 int”参数作为其参数。[length](#)
- 如果没有找到 的匹配值 [maType](#)，则 [交换机](#) 将执行由 引入的最后一个本地块 [=>](#)，该块充当包罗万象的角色。我们在该块中生成运行时错误。我们还以 [float\(na\)](#) 这样的方式结束它，以便本地块返回一个其类型与结构中其他本地块的类型兼容的值，以避免编译错误。

## 不带表达式的 [switch](#)

这是不使用表达式的 [switch](#)结构的示例：

```
//@version=5
strategy("Switch without an expression", "", true)

bool longCondition = ta.crossover(ta.sma(close, 14), ta.sma(close, 28))
bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))

switch
    longCondition => strategy.entry("Long ID", strategy.long)
    shortCondition => strategy.entry("Short ID", strategy.short)
```

注意：

- 我们使用 [开关](#) 来选择要发出的适当策略顺序，具体取决于 `longCondition` 或 `shortCondition` “bool” 变量是否为 `true`。
- `longCondition` 和的建筑条件 `shortCondition` 是排他性的。虽然它们可以 `false` 同时存在，但它们不能 `true` 同时存在。因此，仅执行[switch结构的一个本地块](#)这一事实对我们来说不是问题。
- 我们在进入 [switch结构之前评估对ta.crossover\(\)](#) 和 [ta.crossunder\(\)](#) 的调用。不这样做（如以下示例所示）将阻止在每个柱上执行函数，这将导致编译器警告和不稳定的行为：

```
//@version=5
strategy("Switch without an expression", "", true)

switch
    // Compiler warning! Will not calculate correctly!
    ta.crossover( ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Long ID",
strategy.long)
    ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Short ID",
strategy.short)
```

## 匹配本地块类型要求

当结构体中使用多个局部块时，其所有局部块的返回值类型必须匹配。这仅适用于在声明中使用结构体为变量赋值的情况，因为变量只能有一种类型，如果语句在其分支中返回两种不兼容的类型，则无法正确确定变量类型。如果该结构未分配到任何位置，则其分支可以返回不同的值。

这段代码编译得很好，因为 `close` 和 `open` 都是以下类型 `float`：

```
x = if close > open
    close
else
    open
```

此代码无法编译，因为第一个本地块返回一个 `float` 值，而第二个本地块返回 a `string`，并且 - 语句的结果 `if` 分配给 `x` 变量：

```
// Compilation error!
x = if close > open
    close
else
    "open"
```

## 循环

### 介绍

## 当不需要循环时

Pine Script™ 的运行时及其内置函数使得在许多情况下不需要循环。尚未熟悉 Pine Script™ 运行时和内置程序的新晋 Pine Script™ 程序员如果想要计算最后 10 个 [接近](#) 值的平均值，通常会编写如下代码：

```
//@version=5
indicator("Inefficient MA", "", true)
MA_LENGTH = 10
sumOfCloses = 0.0
for offset = 0 to MA_LENGTH - 1
    sumOfCloses := sumOfCloses + close[offset]
inefficientMA = sumOfCloses / MA_LENGTH
plot(inefficientMA)
```

在 Pine 中完成此类任务时，[for](#) 循环是不必要的且效率低下。应该这样做。此代码更短并且运行速度更快，因为它不使用循环并使用 [ta.sma\(\)](#) 内置函数来完成任务：

```
//@version=5
indicator("Efficient MA", "", true)
thePineMA = ta.sma(close, 10)
plot(thePineMA)
```

计算最后一个条中某个条件的出现次数也是 Pine Script™ 初学者经常认为必须使用循环完成的一项任务。要计算最后 10 个柱中的上涨柱数，他们将使用：

```
//@version=5
indicator("Inefficient sum")
MA_LENGTH = 10
upBars = 0.0
for offset = 0 to MA_LENGTH - 1
    if close[offset] > open[offset]
        upBars := upBars + 1
plot(upBars)
```

在 Pine 中编写此代码的有效方法（对于程序员来说，因为它可以节省时间，实现最快加载图表，并最公平地共享我们的公共资源），是使用 [math.sum\(\)](#) 内置函数来完成任务：

```
//@version=5
indicator("Efficient sum")
upBars = math.sum(close > open ? 1 : 0, 10)
plot(upBars)
```

那里发生的事情是：

- 我们使用[?:](#) 三元运算符构建一个表达式，在向上的柱上生成 1，在其他柱上生成 0。
- 我们使用[math.sum\(\)](#) 内置函数来保存最后 10 个柱的该值的运行总和。

## 何时需要循环

循环的存在有充分的理由，因为即使在 Pine Script™ 中，它们在某些情况下也是必要的。这些案例通常包括：

- [集合（数组、矩阵和映射）](#) 的操作。
- 回顾历史，使用只能在当前柱上知道的参考值来分析柱，例如，找出有多少过去的高点高于 [当前柱的高点](#)。由于当前柱的[最高价](#) 仅在脚本运行的柱上已知，因此需要一个循环来及时返回并分析过去的柱。
- 对过去的柱进行无法使用内置函数完成的计算。

### for

[for结构](#) 允许使用计数器重复执行语句。其语法为：

```
[ [<declaration_mode>] [<type>] <identifier> = ]for <identifier> = <expression> to  
<expression>[ by <expression>]  
    <local_block_loop>
```

在哪里：

- 方括号 ([]) 中的部分可以出现零次或一次，大括号 ({} ) 中的部分可以出现零次或多次。
- <declaration\_mode> 是变量的[声明模式](#)
- 是可选的，就像在几乎所有 Pine Script™ 变量声明中一样（请参阅[types](#)）
- 是变量的[名称](#)
- <表达式> 可以是文字、变量、表达式或函数调用。
- <local\_block\_loop> 由零个或多个语句组成，后跟一个返回值，该返回值可以是值的元组。它必须缩进四个空格或制表符。它可以包含 `break` 退出循环的语句，或 `continue` 退出当前迭代并继续下一个迭代的语句。
- 分配给该变量的值是<local\_block\_loop>的返回值，即循环最后一次迭代计算的最后一个值，或者如果循环未执行则为[na\\_](#)。
- 中的标识符是循环的计数器初始值。`for <identifier>`
- 中的表达式是计数器的起始值。`= <expression>`
- 中的表达式是计数器的最终值。\*\*它仅在进入循环时才被评估\*\*。`to <expression>`
- 中的表达式是可选的。这是循环计数器在循环的每次迭代中增加或减少的步骤。当时，其默认值为 1。时为 -1。用作默认值的步长 (+1 或 -1) 由开始值和结束值确定。`by <expression>``start value < end value``start value > end value`

此示例使用[for](#) 语句来回顾用户定义的柱形数量，以确定有多少柱形的[最高点](#)高于或低于 图表上最后一个柱形的[最高点](#)。这里需要一个[for](#) 循环，因为脚本只能访问图表最后一个柱上的参考值。这里，Pine Script™ 的运行时不能用于动态计算，因为脚本正在逐条执行：

```
//@version=5  
indicator(`for` loop)  
lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
```

```

higherBars = 0
lowerBars = 0
if barstate.islast
    var label lbl = label.new(na, na, "", style = label.style_label_left)
    for i = 1 to lookbackInput
        if high[i] > high
            higherBars += 1
        else if high[i] < high
            lowerBars += 1
    label.set_xy(lbl, bar_index, high)
    label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") +
str.tostring(lowerBars, "# lower bars"))

```

此示例在其函数中使用循环 `checkLinesForBreaches()` 来遍历枢轴线数组，并在价格穿过枢轴线时将其删除。这里需要一个循环，因为必须在每个柱上检查每个 `hiPivotLines` 和数组中的所有行 `loPivotLines`，并且没有内置函数可以为我们执行此操作：

```

//@version=5
MAX_LINES_COUNT = 100
indicator("Pivot line breaches", "", true, max_lines_count = MAX_LINES_COUNT)

color hiPivotColorInput = input(color.new(color.lime, 0), "High pivots")
color loPivotColorInput = input(color.new(color.fuchsia, 0), "Low pivots")
int pivotLegsInput = input.int(5, "Pivot legs")
int qtyOfPivotsInput = input.int(50, "Quantity of last pivots to remember", minval =
= 0, maxval = MAX_LINES_COUNT / 2)
int maxLineLengthInput = input.int(400, "Maximum line length in bars", minval = 2)

// —— Queues a new element in an array and de-queues its first element.
qDq(array, qtyOfElements, arrayElement) =>
    array.push(array, arrayElement)
    if array.size(array) > qtyOfElements
        // Only dequeue if array has reached capacity.
        array.shift(array)

// —— Loop through an array of lines, extending those that price has not crossed
and deleting those crossed.
checkLinesForBreaches(arrayOfLines) =>
    int qtyOfLines = array.size(arrayOfLines)
    // Don't loop in case there are no lines to check because "to" value will be `na` then`.
    for lineNumber = 0 to (qtyOfLines > 0 ? qtyOfLines - 1 : na)
        // Need to check that array size still warrants a loop because we may have
        deleted array elements in the loop.
        if lineNumber < array.size(arrayOfLines)
            line currentLine = array.get(arrayOfLines, lineNumber)
            float lineLevel = line.get_price(currentLine, bar_index)
            bool lineWasCrossed = math.sign(close[1] - lineLevel) != math.sign(close -
lineLevel)

```

```

        bool lineIsTooLong = bar_index - line.get_x1(currentLine) >
maxLineLengthInput
        if lineWasCrossed or lineIsTooLong
            // Line stays on the chart but will no longer be extend on further
bars.
            array.remove(arrayOfLines, lineNo)
            // Force type of both local blocks to same type.
            int(na)
        else
            line.set_x2(currentLine, bar_index)
            int(na)

// Arrays of lines containing non-crossed pivot lines.
var array<line> hiPivotLines = array.new_line(qtyOfPivotsInput)
var array<line> loPivotLines = array.new_line(qtyOfPivotsInput)

// Detect new pivots.
float hiPivot = ta.pivothigh(pivotLegsInput, pivotLegsInput)
float loPivot = ta.pivotlow(pivotLegsInput, pivotLegsInput)

// Create new lines on new pivots.
if not na(hiPivot)
    line.newLine = line.new(bar_index[pivotLegsInput], hiPivot, bar_index, hiPivot,
color = hiPivotColorInput)
    line.delete(qDq(hiPivotLines, qtyOfPivotsInput, newLine))
else if not na(loPivot)
    line.newLine = line.new(bar_index[pivotLegsInput], loPivot, bar_index, loPivot,
color = loPivotColorInput)
    line.delete(qDq(loPivotLines, qtyOfPivotsInput, newLine))

// Extend lines if they haven't been crossed by price.
checkLinesForBreaches(hiPivotLines)
checkLinesForBreaches(loPivotLines)

```

## while

while 结构允许重复执行语句，直到条件为假为止。其语法为：

```
[[<declaration_mode>] [<type>] <identifier> = ]while <expression>
    <local_block_loop>
```

在哪里：

- 方括号 ([ ]) 中的部分可以出现零次或一次。
- <declaration\_mode> 是变量的[声明模式](#)
- 是可选的，就像在几乎所有 Pine Script™ 变量声明中一样（请参阅[types](#)）

- 是变量的名称
- <表达式> 可以是文字、变量、表达式或函数调用。它在循环的每次迭代时进行评估。当它的计算结果为 `true`，循环就会执行。当它求值时 `false` 循环停止。请注意，表达式的计算仅在每次迭代之前完成。循环内表达式值的更改只会影响下一次迭代。
- <local\_block\_loop> 由零个或多个语句组成，后跟一个返回值，该返回值可以是值的元组。它必须缩进四个空格或制表符。它可以包含 `break` 退出循环的语句，或 `continue` 退出当前迭代并继续下一个迭代的语句。
- 分配给变量的值是<local\_block\_loop>的返回值，即循环最后一次迭代计算的最后一个值，或者如果循环没有执行则为na\_。

这是使用 `while` 结构而不是 for one 编写的for 部分的第一个代码示例：

```
//@version=5
indicator(`for` loop)
lookbackInput = input.int(50, "Lookback in bars", minval = 1, maxval = 4999)
higherBars = 0
lowerBars = 0
if barstate.islast
    var label lbl = label.new(na, na, "", style = label.style_label_left)
    // Initialize the loop counter to its start value.
    i = 1
    // Loop until the `i` counter's value is <= the `lookbackInput` value.
    while i <= lookbackInput
        if high[i] > high
            higherBars += 1
        else if high[i] < high
            lowerBars += 1
            // Counter must be managed "manually".
            i += 1
        label.set_xy(lbl, bar_index, high)
        label.set_text(lbl, str.tostring(higherBars, "# higher bars\n") +
str.tostring(lowerBars, "# lower bars"))
```

注意：

- 计数器必须在 `while` 的本地块 `i` 内显式递增 1。
- 我们使用 `+=` 运算符将计数器加一。相当于 `lowerBars += 1` `` `lowerBars := lowerBars + 1`

让我们使用while结构来计算阶乘函数：

```
//@version=5
indicator("")
int n = input.int(10, "Factorial of", minval=0)

factorial(int val = na) =>
    int counter = val
    int fact = 1
    result = while counter > 0
        fact := fact * counter
```

```

counter := counter - 1
fact

// Only evaluate the function on the first bar.
var answer = factorial(n)
plot(answer)

```

注意：

- 我们使用`input.int()`作为输入，因为我们需要指定一个`minval`值来保护我们的代码。虽然`input()`也支持int类型值的输入，但不支持参数`minval`。
- 我们将脚本的功能封装在一个`factorial()`函数中，该函数接受必须计算其阶乘的值作为参数。我们已经习惯了声明函数的参数，这表示如果调用函数时不带参数（如`in`），则参数将初始化为`na`，这将阻止while循环的执行，因为其表达式将返回`na`。因此，`while`结构会将变量初始化为`na`。反过来，因为的初始化是函数本地块的返回值，所以该函数将返回`na`。`int val = na``factorial()``val``counter > 0``result``result`
- 请注意while本地块的最后一行：`fact`。它是本地块的返回值，即`while`结构最后一次迭代时的值。
- 我们的初始化`result`不是必需的；我们这样做是为了可读性。我们也可以使用：

```

while counter > 0
    fact := fact * counter
    counter := counter - 1
fact

```

# 类型系统

## 介绍

Pine Script™ 类型系统确定脚本值与各种函数和操作的兼容性。虽然在不了解类型系统的情况下也可以编写简单的脚本，但要达到一定程度的语言熟练程度，必须对其有合理的理解，并且深入了解其微妙之处将使您能够充分利用其潜力。

Pine Script™ 使用类型对所有值进行分类，并使用限定符来确定值是恒定的、在第一次脚本迭代时建立的还是在整个脚本执行过程中动态的。该系统适用于所有 Pine 值，包括来自文字、变量、表达式、函数返回和函数参数的值。

类型系统与 Pine 的执行模型 和时间序列概念紧密相连。了解这三者对于充分利用 Pine Script™ 的强大功能至关重要。

### 笔记

为了简洁起见，我们经常使用“类型”来指代“限定类型”。

## 限定词

Pine Script™ 限定符标识何时可以在脚本执行中访问某个值：

- 限定为const的值是在编译时建立的（即，在 Pine 编辑器中保存脚本或将其添加到图表时）。
- 符合输入条件的值在输入时可用（即，当更改脚本的“设置/输入”选项卡中的值时）。
- 限定为简单的值在零小节（脚本执行的第一个小节）处建立。
- 作为系列限定的值可以在脚本执行过程中发生变化。

Pine Script™ 将类型限定符的主导地位建立在以下层次结构上：**const < input < simple < series**，其中“const”是最弱的限定符，“series”是最强的。限定符层次结构转化为以下规则：每当变量、函数或操作与特定限定类型兼容时，也允许具有较弱限定符的值。

脚本始终根据计算中的主要限定符限定其表达式的返回值。例如，计算涉及“const”和“series”值的表达式将返回限定为“series”的值。此外，脚本无法将值的限定符更改为层次结构中较低的限定符。如果一个值获得更强的限定符（例如，最初推断为“简单”的值在脚本执行的后期变成“系列”），则该状态是不可逆的。

请注意，只有限定为“系列”的值可以在脚本执行过程中发生变化，其中包括来自各种内置函数的值，例如[close](#)和[volume](#)，以及涉及“系列”值的任何操作的结果。限定为“const”、“input”或“simple”的值在整个脚本执行过程中是一致的。

## 常量

限定为“const”的值是在编译时、脚本开始执行之前建立的。当在 Pine 编辑器中保存脚本时，最初会发生编译，这不需要它在图表上运行。带有“const”限定符的值在脚本迭代之间永远不会改变，甚至在其执行的初始栏上也不会改变。

脚本可以通过使用文字值或从仅使用文字值或其他限定为“const”的变量的表达式计算值来将值限定为“const”。

这些是文字值的示例：

- 字面量 int: `1, -1, 42`
- 字面浮点数: `1., 1.0, 3.14, 6.02E-23, 3e8`
- 文字布尔: `true, false`
- 字面颜色: `#FF55C6, #FF55C6ff`
- 文字字符串: `,"A text literal"``"Embedded single quotes 'text'"``'Embedded double quotes "text"'`

`const` 用户可以通过在声明中包含关键字来显式定义仅接受“const”值的变量和参数。

我们的[风格指南](#)建议使用大写的 SNAKE\_CASE 来命名“const”变量以提高可读性。虽然这不是必需的，但在声明“const”变量时也可以使用[var](#) 关键字，以便脚本仅在数据集的第一个柱上初始化它们。请参阅我们的用户手册的[这一部分](#)以获取更多信息。

[下面是一个在indicator\(\)](#) 和 [plot\(\)](#) 函数中使用“const”值的示例，这两个函数都需要“const string”限定类型的值作为它们的 `title` 参数：

```

//@version=5

// The following global variables are all of the "const string" qualified type:

//@variable The title of the indicator.
INDICATOR_TITLE = "const demo"
//@variable The title of the first plot.
var PLOT1_TITLE = "High"
//@variable The title of the second plot.
const string PLOT2_TITLE = "Low"
//@variable The title of the third plot.
PLOT3_TITLE = "Midpoint between " + PLOT1_TITLE + " and " + PLOT2_TITLE

indicator(INDICATOR_TITLE, overlay = true)

plot(high, PLOT1_TITLE)
plot(low, PLOT2_TITLE)
plot(hl2, PLOT3_TITLE)

```

以下示例将引发编译错误，因为它使用 [syminfo.ticker](#)，它返回一个“简单”值，因为它依赖于只有在脚本开始执行后才能访问的图表信息：

```

//@version=5

//@variable The title in the `indicator()` call.
var NAME = "My indicator for " + syminfo.ticker

indicator(NAME, "", true) // Causes an error because `NAME` is qualified as a "simple
// string".
plot(close)

```

## 输入

限定为“输入”的值是在通过 `input.*()` 函数初始化后建立的。这些函数生成的值用户可以在脚本设置的“输入”选项卡中进行修改。当更改此选项卡中的任何值时，脚本将从图表历史记录的开头重新执行，以确保其输入值在整个执行过程中保持一致。

### 笔记

`input.source()` 函数是命名空间中的一个例外 `input.*()`，因为它返回“系列”限定值而不是“输入”，因为内置变量（例如 [open](#)、[close](#) 等）以及来自另一个脚本绘图的值，被称为“系列”。

以下脚本绘制了和上下文 `sourceInput` 中 `a` 的值。`request.security()` 调用在此脚本中有效，因为它的 `和` 参数允许“简单字符串”参数，这意味着它们也可以接受“输入字符串”值，因为“输入”限定符在层次结构中较低：`symbolInput``timeframeInput``symbol``timeframe`

```
//@version=5
```

```

indicator("input demo", overlay = true)

//@variable The symbol to request data from. Qualified as "input string".
symbolInput = input.symbol("AAPL", "Symbol")
//@variable The timeframe of the data request. Qualified as "input string".
timeframeInput = input.timeframe("D", "Timeframe")
//@variable The source of the calculation. Qualified as "series float".
sourceInput = input.source(close, "Source")

//@variable The `sourceInput` value from the requested context. Qualified as "series
float".
requestedSource = request.security(symbolInput, timeframeInput, sourceInput)

plot(requestedSource)

```

## 简单的

仅当脚本开始在其历史记录的第一个图表条上执行时，限定为“简单”的值才可用，并且它们在脚本执行期间保持一致。

`simple` 用户可以通过在声明中包含关键字来显式定义接受“简单”值的变量和参数。

许多内置变量返回“简单”限定值，因为它们依赖于脚本只有在开始执行后才能获取的信息。此外，许多内置函数需要不随时间变化的“简单”参数。只要脚本允许“简单”值，它也可以接受限定为“输入”或“const”的值。

此脚本突出显示背景以警告用户他们正在使用非标准图表类型。它使用`chart.is_standard`的值来计算`isNonStandard`变量，然后使用该变量的值来计算`warningColor`也引用“简单”值的`a. bgcolor() color`的参数允许“系列颜色”参数，这意味着它也可以接受“简单颜色”值，因为“简单”在层次结构中较低：

```

//@version=5
indicator("simple demo", overlay = true)

//@variable Is `true` when the current chart is non-standard. Qualified as "simple
bool".
isNonStandard = not chart.is_standard
//@variable Is orange when the current chart is non-standard. Qualified as "simple
color".
simple color warningColor = isNonStandard ? color.new(color.orange, 70) : na

// Colors the chart's background to warn that it's a non-standard chart type.
bgcolor(warningColor, title = "Non-standard chart color")

```

## 系列

限定为“系列”的值在脚本中提供了最大的灵活性，因为它们可以在任何柱上更改，甚至在同一柱上多次更改。

`series` 用户可以通过在声明中包含关键字来显式定义接受“系列”值的变量和参数。

内置变量（例如[open](#)、[high](#)、[low](#)、[close](#)、[Volume](#)、[time](#)和[bar\\_index](#)）以及使用此类内置变量的任何表达式的结果都被限定为“系列”。返回动态值的任何函数或操作的结果将始终是一个“系列”，使用历史引用运算符`[]`访问历史值的结果也是如此。只要脚本允许使用“series”值，它也将接受带有任何其他限定符的值，因为“series”是层次结构中的最高限定符。

此脚本显示上方柱线的[最高](#) 和[最低](#)值。分配给和变量的值是“series float”限定类型，因为它们可以在脚本执行过程中发生变化：`sourceInput``lengthInput``highest``lowest`

```
//@version=5
indicator("series demo", overlay = true)

//@variable The source value to calculate on. Qualified as "series float".
series float sourceInput = input.source(close, "Source")
//@variable The number of bars in the calculation. Qualified as "input int".
lengthInput = input.int(20, "Length")

//@variable The highest `sourceInput` value over `lengthInput` bars. Qualified as
"series float".
series float highest = ta.highest(sourceInput, lengthInput)
//@variable The lowest `sourceInput` value over `lengthInput` bars. Qualified as
"series float".
lowest = ta.lowest(sourceInput, lengthInput)

plot(highest, "Highest source", color.green)
plot(lowest, "Lowest source", color.red)
```

## 类型

Pine Script™ 类型对值进行分类并确定它们兼容的函数和操作。他们包括：

- 基本类型：[int](#)、[float](#)、[bool](#)、[color](#)和[string](#)
- 特殊类型：[plot](#)、[hline](#)、[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)、[chart.point](#)、[array](#)、[matrix](#)和[map](#)
- [用户定义类型 \(UDT\)](#)
- [空白](#)

基本类型是指值的基本性质，例如，值 1 属于“int”类型，1.0 属于“float”类型，“AAPL”属于“string”类型等。特殊类型和用户定义的类型使用引用特定类的对象的ID。例如，“label”类型的值包含一个 ID，充当引用“label”对象的指针。“void”类型是指不返回可用值的函数或[方法的输出](#)。

Pine Script™ 可以自动将某些类型的值转换为其他类型。自动转换规则为： **int** → **float** → **bool**。有关详细信息，请参阅本页的[类型转换部分](#)。

在大多数情况下，Pine Script™ 可以自动确定值的类型。但是，我们还可以使用 **type** 关键字显式指定类型，以提高可读性和需要显式定义的代码（例如，声明分配给 [na](#) 的变量）。例如：

```
//@version=5
indicator("Types demo", overlay = true)

//@variable A value of the "const string" type for the `ma` plot's title.
string MA_TITLE = "MA"

//@variable A value of the "input int" type. Controls the length of the average.
int lengthInput = input.int(100, "Length", minval = 2)

//@variable A "series float" value representing the last `close` that crossed over the
`ma`.
var float crossValue = na

//@variable A "series float" value representing the moving average of `close`.
float ma = ta.sma(close, lengthInput)
//@variable A "series bool" value that's `true` when the `close` crosses over the `ma`.
bool crossUp = ta.crossover(close, ma)
//@variable A "series color" value based on whether `close` is above or below its `ma`.
color maColor = close > ma ? color.lime : color.fuchsia

// Update the `crossValue`.
if crossUp
    crossValue := close

plot(ma, MA_TITLE, maColor)
plot(crossValue, "Cross value", style = plot.style_circles)
plotchar(crossUp, "Cross Up", "▲", location.belowbar, size = size.small)
```

## 整数

“int”类型的值表示整数，即没有任何小数的整数。

整数文字是以十进制表示法编写的数值。例如：

```
1
-1
750
```

[bar\\_index](#)、[time](#)、[timenow](#)、[dayofmonth](#)和[Strategy.wintrades](#)等内置变量 均返回“int”类型的值。

## 漂浮

“float”类型的值表示浮点数，即可以包含整数和小数的数字。

浮点文字是用 . 分隔符编写的数值。它们还可能包含符号 e or E (这意味着“10 的 X 次方”，其中 X 是 e or E 符号后面的数字)。例如：

```
3.14159    // Rounded value of Pi ( $\pi$ )
- 3.0
6.02e23    //  $6.02 * 10^{23}$  (a very large value)
1.6e-19    //  $1.6 * 10^{-19}$  (a very small value)
```

Pine Script™ 中“浮点”值的内部精度为 1e-16。

[close](#)、[hlcc4](#)、[volume](#)、[ta.vwap](#)和[strategy.position\\_size](#)等内置变量 均返回“float”类型的值。

## 布尔

“bool”类型的值表示比较或条件的真值，脚本可以在[条件结构](#)和其他表达式中使用它。

只有两个文字表示布尔值：

```
true    // true value
false   // false value
```

当“bool”类型的表达式返回[na](#)时，脚本将其值视为 `false` 评估条件语句和运算符时的值。

内置变量如 [barstate.isfirst](#)、[Chart.is\\_heikinashi](#)、[session.ismarket](#)和[timeframe.isdaily](#)都返回“bool”类型的值。

## 颜色

颜色文字具有以下格式：`#RRGGBB` 或 `#RRGGBBAA`。字母对代表 和之间的十六进制值（十进制 0 到 255），其中： `00``FF`

- `RR`、`GG` 和 `BB` 对分别表示颜色的红色、绿色和蓝色分量的值。
- `AA` 是颜色不透明度（或alpha分量）的可选值，其中 `00` 不可见且 `FF` 不透明。当文字不包含对时 `AA`，脚本将其视为完全不透明（与 using 相同 `FF`）。
- 文字中的十六进制字母可以是大写或小写。

这些是“颜色”文字的示例：

```
#000000      // black color
#FF0000      // red color
#00FF00      // green color
#0000FF      // blue color
#FFFFFF      // white color
#808080      // gray color
#3ff7a0      // some custom color
#FF000080    // 50% transparent red color
#FF0000ff    // same as #FF0000, fully opaque red color
#FF000000    // completely transparent red color
```

Pine Script™ 还具有[内置颜色常量](#)，包括 `color.green`、`color.red`、`color.orange`、`color.blue`（函数中的默认颜色以及[绘图类型](#) `plot*` 中许多默认的与颜色相关的属性）等。

当使用内置颜色常量时，可以通过 `color.new()` 函数向它们添加透明度信息。

请注意，在 `color.*()` 函数中指定红色、绿色或蓝色分量时，我们使用“int”或“float”参数，其值介于 0 到 255 之间。指定透明度时，我们使用介于 0 到 100 之间的值，其中 0 表示完全不透明，100 表示完全不透明。意思是完全透明。例如：

```
//@version=5
indicator("Shading the chart's background", overlay = true)

//@variable A "const color" value representing the base for each day's color.
color BASE_COLOR = color.rgb(0, 99, 165)

//@variable A "series int" value that modifies the transparency of the `BASE_COLOR` in
`color.new()` .
int transparency = 50 + int(40 * dayofweek / 7)

// Color the background using the modified `BASE_COLOR`.
bgcolor(color.new(BASE_COLOR, transparency))
```

有关在脚本中使用颜色的更多信息，请参阅[用户手册中有关颜色的页面](#)。

## 细绳

“字符串”类型的值表示字母、数字、符号、空格和其他字符的序列。

Pine 中的字符串文字是用单引号或双引号括起来的字符。例如：

```
"This is a string literal using double quotes."
'This is a string literal using single quotes.'
```

单引号和双引号在 Pine Script™ 中的功能是等效的。双引号内的“字符串”可以包含任意数量的单引号，反之亦然：

```
"It's an example"  
'The "Star" indicator'
```

脚本可以使用反斜杠字符 () 对“字符串”中的封闭分隔符进行转义 \。例如：

```
'It\'s an example'  
"The \"Star\" indicator"
```

我们可以创建包含新行转义字符 () 的“字符串”值，用于使用[绘图类型的函数](#)和对象显示 \n 多行文本。例如：`plot*()``log.*()`

```
"This\nString\nHas\nOne\nWord\nPer\nLine"
```

我们可以使用 [+ 运算符](#) 来连接“字符串”值：

```
"This is a " + "concatenated string."
```

命名空间中的内置函数 `str.*()` 使用专门的操作创建“字符串”值。例如，此脚本创建一个格式化字符串来表示“浮动”价格值，并使用标签显示结果：

```
//@version=5  
indicator("Formatted string demo", overlay = true)  
  
//@variable A "series string" value representing the bar's OHLC data.  
string ohlcString = str.format("Open: {0}\nHigh: {1}\nLow: {2}\nClose: {3}", open,  
high, low, close)  
  
// Draw a label containing the `ohlcString`.  
label.new(bar_index, high, ohlcString, textcolor = color.white)
```

有关显示脚本中“字符串”值的更多信息，请参阅我们的用户手册中有关[文本和形状的页面](#)。

`syminfo.tickerid`、`syminfo.currency` 和 `timeframe.period` 等内置变量 返回“string”类型的值。

## 绘图和水平线

Pine Script™ 的 `plot()` 和 `hline()` 函数返回分别引用“plot”和“hline”类型实例的 ID。这些类型在图表上显示计算值和水平水平，并且可以将它们的 ID 分配给变量以与内置 `fill()` 函数一起使用。

例如，此脚本在图表上绘制两个 EMA，并使用 `fill()` 函数填充它们之间的空间：

```
//@version=5  
indicator("plot fill demo", overlay = true)  
  
//@variable A "series float" value representing a 10-bar EMA of `close`.
```

```

float emaFast = ta.ema(close, 10)
//@variable A "series float" value representing a 20-bar EMA of `close`.
float emaSlow = ta.ema(close, 20)

//@variable The plot of the `emaFast` value.
emaFastPlot = plot(emaFast, "Fast EMA", color.orange, 3)
//@variable The plot of the `emaSlow` value.
emaSlowPlot = plot(emaSlow, "Slow EMA", color.gray, 3)

// Fill the space between the `emaFastPlot` and `emaSlowPlot`.
fill(emaFastPlot, emaSlowPlot, color.new(color.purple, 50), "EMA Fill")

```

需要注意的是，与其他特殊类型不同，Pine 中没有 `plot` 或 `hline` 关键字来显式声明变量的类型为“`plot`”或“`hline`”。

用户可以通过命名空间中的变量来控制脚本绘图的显示位置 `display.*`。此外，一个脚本可以通过 `input.source()` 函数使用另一个脚本绘图中的值作为外部输入（请参阅我们的用户手册中有关[源输入](#)的部分）。

## 绘图类型

Pine Script™ 绘图类型允许脚本在图表上创建自定义绘图。它们包括以下内容：[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#) 和 [table](#)。

每种类型还有一个命名空间，其中包含创建和管理绘图实例的所有内置函数。例如，以下 `*.new()` 构造函数在脚本中创建这些类型的新对象：[line.new\(\)](#)、[linefill.new\(\)](#)、[box.new\(\)](#)、[polyline.new\(\)](#)、[label.new\(\)](#) 和 [table.new\(\)](#)。

这些函数中的每一个都会返回一个 `ID`，该 `ID` 是唯一标识绘图对象的引用。`ID` 始终限定为“系列”，这意味着它们的限定类型为“系列线”、“系列标签”等。绘图 `ID` 的作用类似于指针，因为每个 `ID` 在绘图名称空间的所有函数中都引用绘图的特定实例。例如，当需要使用[line.delete\(\)](#) 删除该对象时，稍后将使用 [line.new\(\)](#) 调用返回的行的 `ID` 来引用该特定对象。

## 图表点

图表点是表示图表上坐标的特殊类型。脚本使用来自[Chart.point](#)对象的信息来确定[线条](#)、[方框](#)、[折线](#)和[标签](#)的图表位置。

这种类型的对象包含三个字段：`time`、`index` 和 `price`。绘图实例是否使用[Chart.point](#)中的 `time` 或 `price` 字段作为 `x` 坐标取决于绘图的属性。`price``xloc`

我们可以使用以下任意函数在脚本中创建图表点：

- [Chart.point.new\(\)](#) - 使用指定的 `time` 和 `index` 来创建一个新的[Chart.point](#)。`time``index``price`
- [Chart.point.now\(\)](#) - 创建一个具有指定 `y` 坐标的新的[Chart.point](#)。`price` 和 `index` 字段包含函数执行所在柱的 `time` 时间和 `bar_index`。`index`
- [Chart.point\\_from\\_index\(\)](#) - 创建一个带有 `x` 坐标和 `y` 坐标的新的[图表.point](#)。生成的实例的 `na` 字段是 `na`，这意味着它不适用于使用 `xloc.bar_time` 值的绘图对象。`index``price``time``xloc`

- [Chart.point.from\\_time\(\)](#) - 创建一个带有x坐标和y坐标的新[Chart.point](#)。生成的实例的字段是na，这意味着它不适用于使用[xloc.bar\\_index](#)值的绘图对象。`time``price``index``xloc`
- [Chart.point.copy\(\)](#) - 创建一个新的[chart.point](#)，其中包含与函数调用中相同的time、index、和price信息。`id`

此示例在每个图表条形上绘制连接前一个条形的最高点和当前条形的最低点的线。它还在每条线的两个点处显示标签。线条和标签从firstPoint和secondPoint变量获取信息，这些变量引用使用[Chart.point\\_from\\_index\(\)](#)和[Chart.point.now\(\)](#)创建的图表点：

```
//@version=5
indicator("Chart points demo", overlay = true)

//@variable A new `chart.point` at the previous `bar_index` and `high`.
firstPoint = chart.point.from_index(bar_index - 1, high[1])
//@variable A new `chart.point` at the current bar's `low`.
secondPoint = chart.point.now(low)

// Draw a new line connecting coordinates from the `firstPoint` and `secondPoint`.
// This line uses the `index` fields from the points as x-coordinates.
line.new(firstPoint, secondPoint, color = color.purple, width = 3)
// Draw a label at the `firstPoint`. Uses the point's `index` field as its x-
coordinate.
label.new(
    firstPoint, str.tostring(firstPoint.price), color = color.green,
    style = label.style_label_down, textcolor = color.white
)
// Draw a label at the `secondPoint`. Uses the point's `index` field as its x-
coordinate.
label.new(
    secondPoint, str.tostring(secondPoint.price), color = color.red,
    style = label.style_label_up, textcolor = color.white
)
```

## 收藏

Pine Script™ 中的集合（[数组](#)、[矩阵](#)和[映射](#)）使用引用 ID，与其他特殊类型（例如标签）非常相似。ID 的类型定义了集合将包含的元素的类型。在 Pine 中，我们通过将[类型模板](#)附加到array、matrix 或 map 关键字来指定[array](#)、[matrix](#)和[map](#)类型：

- `array<int>` 定义一个包含“int”元素的数组。
- `array<label>` 定义一个包含“标签”ID 的数组。
- `array<UDT>` 定义一个数组，其中包含引用[用户定义类型 \(UDT\)](#)对象的 ID。
- `matrix<float>` 定义一个包含“float”元素的矩阵。
- `matrix<UDT>` 定义一个包含引用[用户定义类型 \(UDT\)](#)对象的 ID 的矩阵。

- `map<string, float>` 定义一个包含“字符串”键和“浮点”值的映射。
- `map<int, UDT>` 定义一个映射，其中包含“int”键和[用户定义类型\(UDT\)](#)实例的 ID 作为值。

例如，可以通过以下任一等效方式声明单个元素值为 10 的“int”数组：

```
a1 = array.new<int>(1, 10)
array<int> a2 = array.new<int>(1, 10)
a3 = array.from(10)
array<int> a4 = array.from(10)
```

- 注意：

该 `int[]` 语法还可以指定“int”元素的数组，但不鼓励使用它。不存在以这种方式指定矩阵或映射类型的等效方法。数组存在特定于类型的内置函数，例如 `array.new_int()`，但更通用的 `array.new` 形式是首选，这将 `array.new<int>()` 创建一个“int”元素的数组。

## 用户定义类型

`type` 关键字允许创建[用户定义类型\(UDT\)](#)，[脚本](#)可以从中创建[对象](#)。UDT 是复合类型；它们包含任意数量的可以是任何类型的字段，包括其他用户定义的类型。定义用户定义类型的语法是：

```
[export] type <UDT_identifier>
  <field_type> <field_name> [= <value>]
  ...
  ...
```

在哪里：

- `export` 是[库](#)脚本用于导出用户定义类型的关键字。要了解有关导出 UDT 的更多信息，请参阅我们的用户手册的 [库](#)页面。
- `<UDT_identifier>` 是用户定义类型的名称。
- `<field_type>` 是字段的类型。
- `<field_name>` 是字段的名称。
- `<value>` 是该字段的可选默认值，脚本将在创建该 UDT 的新对象时为其分配该默认值。如果未提供值，则该字段的默认值为 `na`。与管理函数签名中参数默认值的规则相同的规则也适用于字段的默认值。例如，UDT 的默认值不能使用历史引用运算符 `[]` 或表达式的结果。

此示例声明一个 `pivotPoint` 带有“int”`pivotTime` 字段和“float”`priceLevel` 字段的 UDT，分别保存有关计算的枢轴的时间和价格信息：

```
//@type          A user-defined type containing pivot information.
//@field pivotTime Contains time information about the pivot.
//@field priceLevel Contains price information about the pivot.

type pivotPoint
  int  pivotTime
  float  priceLevel
```

用户定义类型支持类型递归，即一个UDT的字段可以引用同一个UDT的对象。在这里，我们 `nextPivot` 在之前的 `pivotPoint` 类型中添加了一个引用另一个 `pivotPoint` 实例的字段：

```
//@type          A user-defined type containing pivot information.  
//@field pivotTime Contains time information about the pivot.  
//@field priceLevel Contains price information about the pivot.  
//@field nextPivot A `pivotPoint` instance containing additional pivot information.  
  
type pivotPoint  
    int      pivotTime  
    float   priceLevel  
    pivotPoint nextPivot
```

脚本可以使用两种内置方法来创建和复制 UDT：`new()` 和 `copy()`。请参阅我们的用户手册的[对象](#)页面，了解有关使用 UDT 的更多信息。

## 空白

Pine Script™ 中有一个“void”类型。仅具有副作用且不返回可用结果的函数返回“void”类型。此类函数的一个示例是[alert\(\)](#)；它会执行某些操作（触发警报事件），但不会返回任何可用值。

脚本不能在表达式中使用“void”结果或将它们分配给变量。Pine Script™ 中不 `void` 存在关键字，因为无法声明“void”类型的变量。

## na\_值

Pine Script™ 中有一个特殊值，称为 `na`，它是 *not available* 的缩写。我们使用 `na` 来表示变量或表达式中未定义的值。它 `null` 与 Java 和 Python 中的类似 `None`。

脚本可以自动将 `na` 值转换为几乎任何类型。但是，在某些情况下，编译器无法推断与 `na` 值关联的类型，因为可能适用多个类型转换规则。例如：

```
// Compilation error!  
myVar = na
```

上面的代码行会导致编译错误，因为编译器无法确定变量的性质 `myVar`，即变量是否会引用用于绘图的数值、用于在标签中设置文本的字符串值，或者稍后用于其他目的的其他值。脚本的执行。

要解决此类错误，我们必须显式声明与变量关联的类型。假设该 `myVar` 变量将在后续脚本迭代中引用“浮点”值。我们可以通过使用 `float` 关键字声明变量来解决该错误：

```
float myVar = na
```

或者通过 `float()` 函数将 `na` 值显式转换为“float”类型：

```
myVar = float(na)
```

为了测试变量或表达式的值是否为[na](#), 我们调用[na\(\)](#)函数, `true`如果该值未定义, 该函数将返回。例如:

```
//@variable Is 0 if the `myVar` is `na`, `close` otherwise.  
float myClose = na(myVar) ? 0 : close
```

不要使用`==`比较运算符来测试[na](#)值, 因为脚本无法确定未定义值的相等性:

```
//@variable Returns the `close` value. The script cannot compare the equality of `na`  
values, as they're undefined.  
float myClose = myVar == na ? 0 : close
```

最佳编码实践通常涉及处理[na](#)值以防止计算中出现未定义的值。

例如, 这行代码检查当前柱的[收盘价](#)是否大于前一个柱的值:

```
//@variable Is `true` when the `close` exceeds the last bar's `close`, `false`  
otherwise.  
bool risingClose = close > close[1]
```

在第一个图表条上, 的值为`risingClose na`, 因为没有过去的[收盘价](#)可供参考。

通过将未定义的过去值替换为当前柱中的值, 我们可以确保表达式还在第一个柱上返回可操作的值。当值为[na](#)时, 这行代码使用[nz\(\)](#)函数将过去柱的[收盘价](#)替换为当前柱的[开盘价](#):

```
//@variable Is `true` when the `close` exceeds the last bar's `close` (or the current  
`open` if the value is `na`).  
bool risingClose = close > nz(close[1], open)
```

保护脚本免受[na](#)实例的影响有助于防止未定义的值在计算结果中传播。例如, 此脚本`allTimeHigh`在第一个柱上声明一个变量。然后, 它使用和柱形[最高点](#)之间的[math.max\(\)](#)来更新整个执行过程: `allTimeHigh``allTimeHigh`

```

//@version=5
indicator("na protection demo", overlay = true)

//@variable The result of calculating the all-time high price with an initial value of
`na`.
var float allTimeHigh = na

// Reassign the value of the `allTimeHigh`.
// Returns `na` on all bars because `math.max()` can't compare the `high` to an
undefined value.
allTimeHigh := math.max(allTimeHigh, high)

plot(allTimeHigh) // Plots `na` on all bars.

```

该脚本在所有柱上绘制na值，因为我们没有在代码中包含任何na保护。为了修复该行为并绘制预期结果（即图表价格的历史最高点），我们可以使用 `nz()` 替换该系列中的na值： `allTimeHigh`

```

//@version=5
indicator("na protection demo", overlay = true)

//@variable The result of calculating the all-time high price with an initial value of
`na`.
var float allTimeHigh = na

// Reassign the value of the `allTimeHigh`.
// We've used `nz()` to prevent the initial `na` value from persisting throughout the
calculation.
allTimeHigh := math.max(nz(allTimeHigh), high)

plot(allTimeHigh)

```

## 类型模板

[类型模板指定集合](#)（[数组](#)、[矩阵](#)和[映射](#)）可以包含的数据类型。

[数组](#)和[矩阵](#)的模板由尖括号括起来的单个类型标识符组成，例如 `<int>`、`<label>` 和 `<PivotPoint>`（其中 `PivotPoint` 是 [用户定义类型 \(UDT\)](#)）。

[映射](#)模板由两个括在尖括号中的类型标识符组成，其中第一个指定每个键值对中键的类型，第二个指定值类型。例如，是保存键和值的映射的类型模板。`<string, float>``string``float`

用户可以通过以下方式构建类型模板：

- 基本类型：[int](#)、[float](#)、[bool](#)、[color](#)和[string](#)
- 以下特殊类型：[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)和[Chart.point](#)
- [用户定义类型 \(UDT\)](#)

- 注意：

[映射](#)可以使用任何这些类型作为值，但它们只能接受基本类型作为键。

脚本在创建新的集合实例时使用类型模板来声明指向集合的变量。例如：

```
//@version=5
indicator("Type templates demo")

//@variable A variable initially assigned to `na` that accepts arrays of "int" values.
array<int> intArray = na
//@variable An empty matrix that holds "float" values.
floatMatrix = matrix.new<float>()
//@variable An empty map that holds "string" keys and "color" values.
stringColorMap = map.new<string, color>()
```

## 类型转换

Pine Script™ 包含自动类型转换机制，可在必要时将“int”值\*转换\*（转换）为“float”。需要“float”值的变量或表达式也可以使用“int”值，因为任何整数都可以表示为浮点数，其小数部分等于 0。

为了向后兼容，Pine Script™ 还会在必要时自动将“int”和“float”值转换为“bool”。当将数值传递给需要“bool”类型的函数和操作的参数时，Pine 会自动将它们转换为“bool”。但是，我们不建议依赖此行为。大多数自动将数值转换为“bool”类型的脚本都会产生[编译器警告](#)。通过使用[bool\(\)](#)函数，可以避免编译器警告并提高代码可读性，该函数将数值显式转换为“bool”类型。

将“int”或“float”转换为“bool”时，值 0 会转换为 `false`，任何其他数值始终会转换为 `true`。

下面的代码演示了 Pine 中已弃用的自动转换行为。它在每个柱上创建一个 `randomValue` 具有“系列浮点”值的变量，并将其传递给 `if condition` 结构中的参数和 `plotchar()` 函数调用中的参数。由于这两个参数都接受“bool”值，因此脚本在评估它们时会自动将其转换为“bool”：`series``randomValue`

```
//@version=5
indicator("Auto-casting demo", overlay = true)

//@variable A random rounded value between -1 and 1.
float randomValue = math.round(math.random(-1, 1))
//@variable The color of the chart background.
color bgColor = na

// This raises a compiler warning since `randomValue` is a "float", but `if` expects a
// "bool".
if randomValue
    bgColor := color.new(color.blue, 60)
// This does not raise a warning, as the `bool()` function explicitly casts the
// `randomValue` to "bool".
if bool(randomValue)
    bgColor := color.new(color.blue, 60)
```

```

// Display unicode characters on the chart based on the `randomValue`.
// Whenever `math.random()` returns 0, no character will appear on the chart because 0
// converts to `false`.
plotchar(randomValue)
// We recommend explicitly casting the number with the `bool()` function to make the
// type transformation more obvious.
plotchar(bool(randomValue))

// Highlight the background with the `bgColor`.
bgcolor(bgColor)

```

当自动转换规则不够时，有时需要将一种类型转换为另一种类型。对于这种情况，可以使用以下类型转换函数：[int\(\)](#)、[float\(\)](#)、[bool\(\)](#)、[color\(\)](#)、[string\(\)](#)、[line\(\)](#)、[linefill\(\)](#)、[label\(\)](#)、[box\(\)](#)和[桌子 \(\)](#)。

下面的示例显示了尝试使用“const float”值作为 [ta.sma\(\)](#) `length` 函数调用中的参数的代码。该脚本将无法编译，因为它无法自动将“float”值转换为所需的“int”类型：

```

//@version=5
indicator("Explicit casting demo", overlay = true)

//@variable The length of the SMA calculation. Qualified as "const float".
float LENGTH = 10.0

float sma = ta.sma(close, LENGTH) // Compilation error. The `length` parameter requires
// an "int" value.

plot(sma)

```

该代码引发以下错误：“无法使用参数“length”=“LENGTH”调用“ta.sma”。使用了“const float”类型的参数，但需要使用“series int”。”

编译器告诉我们代码在需要“int”的地方使用“float”值。没有自动转换规则可以将“float”转换为“int”，因此我们必须自己完成这项工作。在此版本的代码中，我们使用 [int\(\)函数](#)在 [ta.sma\(\)](#) 调用中将“float”值显式转换 `LENGTH` 为“int”类型：

```

//@version=5
indicator("explicit casting demo")

//@variable The length of the SMA calculation. Qualified as "const float".
float LENGTH = 10.0

float sma = ta.sma(close, int(LENGTH)) // Compiles successfully since we've converted
// the `LENGTH` to "int".

plot(sma)

```

当声明分配给 [na](#) 的变量时，显式类型转换也很方便，如上一 [节](#) 所述。

例如，once 可以通过以下任一等效方式将值为 `na` 的变量显式声明为“标签”类型：

```
// Explicitly specify that the variable references "label" objects:  
label myLabel = na  
  
// Explicitly cast the `na` value to the "label" type:  
myLabel = label(na)
```

## 元组

元组是一组用括号括起来的、以逗号分隔的表达式。当函数、[方法](#)或其他本地块返回多个值时，脚本会以元组的形式返回这些值。

例如，以下[用户定义函数](#)返回两个“浮点”值的和与积：

```
//@function Calculates the sum and product of two values.  
calcSumAndProduct(float a, float b) =>  
    // @variable The sum of `a` and `b`.  
    float sum = a + b  
    // @variable The product of `a` and `b`.  
    float product = a * b  
    // Return a tuple containing the `sum` and `product`.  
    [sum, product]
```

当我们稍后在脚本中调用此函数时，我们使用元组声明来声明与函数调用返回的值相对应的多个变量：

```
// Declare a tuple containing the sum and product of the `high` and `low`,  
// respectively.  
[hlSum, hlProduct] = calcSumAndProduct(high, low)
```

请记住，与声明单个变量不同，我们无法显式定义元组变量（`hlSum` 在 `hlProduct` 本例中）将包含的类型。编译器自动推断与元组中的变量关联的类型。

在上面的示例中，生成的元组包含相同类型（“float”）的值。但是，需要注意的是，元组可以包含多种类型的值。例如，`chartInfo()` 下面的函数返回一个包含“int”、“float”、“bool”、“color”和“string”值的元组：

```
//@function Returns information about the current chart.  
chartInfo() =>  
    // @variable The first visible bar's UNIX time value.  
    int firstVisibleTime = chart.left_visible_bar_time  
    // @variable The `close` value at the `firstVisibleTime`.  
    float firstVisibleClose = ta.valuewhen(ta.cross(time, firstVisibleTime), close, 0)  
    // @variable Is `true` when using a standard chart type, `false` otherwise.  
    bool isStandard = chart.is_standard  
    // @variable The foreground color of the chart.  
    color fgColor = chart.fg_color
```

```
//@variable The ticker ID of the current chart.
string symbol = syminfo.tickerid
// Return a tuple containing the values.
[firstVisibleTime, firstVisibleClose, isStandard, fgColor, symbol]
```

[元组对于在一次request.security\(\)](#)调用中请求多个值特别方便。

例如，此函数返回一个包含 OHLC 值的元组，该值四舍五入到可被交易品种的[最小刻度](#) roundedOHLC() 值整除的最接近价格。我们将此函数称为[request.security\(\)](#)中的参数，以请求包含每日 OHLC 值的元组：expression

```
//@function Returns a tuple of OHLC values, rounded to the nearest tick.
roundedOHLC() =>
    [math.round_to_mintick(open), math.round_to_mintick(high),
math.round_to_mintick(low), math.round_to_mintick(close)]

[op, hi, lo, cl] = request.security(syminfo.tickerid, "D", roundedOHLC())
```

我们还可以通过直接传递一个舍入值的元组来实现相同的结果，如[request.security\(\)](#) expression 调用中的：

```
[op, hi, lo, cl] = request.security(
    syminfo.tickerid, "D",
    [math.round_to_mintick(open), math.round_to_mintick(high),
math.round_to_mintick(low), math.round_to_mintick(close)])
)
```

[条件结构](#)的局部块（包括 [if](#) 和 [switch](#) 语句）可以返回元组。例如：

```
[v1, v2] = if close > open
    [high, close]
else
    [close, low]
```

和：

```
[v1, v2] = switch
close > open => [high, close]
=>           [close, low]
```

但是，三元不能包含元组，因为三元语句中的返回值不被视为本地块：

```
// Not allowed.
[v1, v2] = close > open ? [high, close] : [close, low]
```

请注意，从函数返回的元组中的所有项目都被限定为“简单”或“系列”，具体取决于其内容。如果元组包含“series”值，则元组中的所有其他元素也将采用“series”限定符。例如：

```
//@version=5
```

```

indicator("Qualified types in tuples demo")

makeTicker(simple string prefix, simple string ticker) =>
    tId = prefix + ":" + ticker // simple string
    source = close // series float
    [tId, source]

// Both variables are series now.
[tId, source] = makeTicker("BATS", "AAPL")

// Error cannot call 'request.security' with 'series string' tId.
r = request.security(tId, "", source)

plot(r)

```

# 内置

## 介绍

Pine Script™ 具有数百个内置变量和函数。它们为您的脚本提供有价值的信息并为您进行计算，从而使您无需编写代码。您对内置函数了解得越多，您就能用 Pine 脚本做更多的事情。

在本页中，我们概述了 Pine Script™ 的一些内置变量和函数。本手册中涉及特定主题的页面将更详细地介绍它们。

所有内置变量和函数均在 Pine Script™ [v5 参考手册](#)中定义。它被称为“参考手册”，因为它是 Pine Script™ 语言的权威参考。无论您是初学者还是专家，它都是您在 Pine 中编码时随时陪伴的必备工具。如果您正在学习您的第一门编程语言，请将[参考手册](#)作为您的朋友。忽略它将使您使用 Pine Script™ 的编程体验变得困难和令人沮丧——就像使用任何其他编程语言一样。

同一族中的变量和函数共享相同的命名空间，即函数名称的前缀。例如，`ta.sma()` 函数位于命名空间中，[代表](#) `ta` “技术分析”。命名空间可以包含变量和函数。

有些变量也有函数版本，例如：

- `ta.tr` 变量返回当前柱的“真实范围”。`ta.tr(true)` 函数调用也返回“真实范围”，但是当通常需要计算它的前一个 [收盘值是na](#) 时，它会改为使用 `high - low` 进行计算。
- 时间[变量](#)给出当前柱的[开盘](#)时间。`time(timeframe)` 函数返回指定的柱线[开盘](#)时间 `timeframe`，即使图表的时间范围不同。`time(timeframe, session)` 函数返回指定柱的[开盘](#)时间 `timeframe`，但前提是该时间在指定 `session` 时间内。`time(timeframe, session, timezone)` 函数返回指定的柱线[开盘](#)时间 `timeframe`，但前提是该时间 `session` 在指定的时间内 `timezone`。

## 内置变量

内置变量的存在有不同的目的。以下是一些示例：

- 与价格和成交量相关的变量：[开盘价](#)、[最高价](#)、[最低价](#)、[收盘价](#)、[hl2](#)、[hlc3](#)、[ohlc4](#) 和 [成交量](#)。

- 命名空间中与符号相关的信息 `syminfo`: `syminfo.basecurrency`、`syminfo.currency`、`syminfo.description`、`syminfo.mintick`、`syminfo.pointvalue`、`syminfo.prefix`、`syminfo.root`、`syminfo.session`、`syminfo.ticker`、`syminfo.tickerid`、`syminfo`。时区和 `syminfo.type`。
- 命名空间中的时间帧（又名“间隔”或“分辨率”，例如 15 秒、30 分钟、60 分钟、1D、3M）变量 `timeframe`: `timeframe.isseconds`、`timeframe.is分钟`、`timeframe.isintraday`、`timeframe.isdaily`、`timeframe.isweekly`、`timeframe.ismonthly`、`timeframe.isdwm`、`timeframe.multiplier` 和 `timeframe.period`。
- 命名空间中的 Bar 状态 `barstate` (请参阅[Bar states](#)页面) : `barstate.isconfirmed`、`barstate.isfirst`、`barstate.ishistory`、`barstate.islast`、`barstate.islastconfirmedhistory`、`barstate.isnew` 和 `barstate.isrealtime`。
- 命名空间中与策略相关的信息 `strategy`: `strategy.equity`、`strategy.initial_capital`、`strategy.grossloss`、`strategy.grossprofit`、`strategy.wintrades`、`strategy.losstrades`、`strategy.position_size`、`strategy.position_avg_price`、`strategy.wintrades`等。

## 内置函数

---

许多函数用于返回结果。以下是一些示例：

- 命名空间中与数学相关的函数 `math`: `math.abs()`、`math.log()`、`math.max()`、`math.random()`、`math.round_to_mintick()`等。
- 命名空间中的技术指标 `ta`: `ta.sma()`、`ta.ema()`、`ta.macd()`、`ta.rsi()`、`ta.supertrend()`等。
- 命名空间中常用于计算技术指标的支持函数 `ta`: `ta.barssince()`、`ta.crossover()`、`ta.highest()`等。
- 从命名空间中的其他交易品种或时间范围请求数据的函数 `request`: `request.dividends()`、`request.earnings()`、`request.financial()`、`request.quandl()`、`request.security()`、`request.splits()`。
- 操作 `str` 命名空间中字符串的函数: `str.format()`、`str.length()`、`str tonumber()`、`str.tostring()`等。
- 用于定义脚本用户可以在脚本的“设置/输入”选项卡中修改的输入值的函数，命名空间为 `input`: `input()`、`input.color()`、`input.int()`、`input.session()`、`input.符号()`等
- 用于操作 `color` 命名空间中颜色的函数: `color.from_gradient()`、`color.new()`、`color.rgb()`等。

有些函数不返回结果，而是因其副作用而被使用，这意味着即使它们不返回结果，它们也会执行某些操作：

- 用作定义三种 Pine 脚本类型之一及其属性的声明语句的函数。每个脚本必须以调用以下函数之一开始：`indicator()`、`strategy()`或 `library()`。
- 绘图或着色函数: `bgcolor()`、`plotbar()`、`plotcandle()`、`plotchar()`、`plotshape()`、`fill()`。
- 下单策略函数，`strategy` 命名空间为： `strategy.cancel()`、`strategy.close()`、`strategy.entry()`、`strategy.exit()`、`strategy.order()`等。
- 策略函数返回有关单个过去交易的信息，在 `strategy` 命名空间中：  
`strategy.closetrades.entry_bar_index()`、`strategy.linedtrades.entry_price()`、  
`strategy.latedtrades.entry_time()`、`strategy.linedtrades.exit_bar_index()`、  
`strategy.linedtrades.max_drawdown()`、`Strategy.Closedtrades.max_runup()`、  
`Strategy.Closedtrades.profit()`等
- 生成警报事件的函数: `alert()`和 `alertcondition()`。

其他函数返回结果，但我们并不总是使用它，例如：[hline\(\)](#)、[plot\(\)](#)、[array.pop\(\)](#)、[label.new\(\)](#)等。

所有内置函数均在 Pine Script™ [v5 参考手册](#)中定义。您可以单击此处列出的任何函数名称以转到参考手册中的条目，该条目记录了函数的签名，即它接受的参数列表以及它返回的值的限定类型（函数可以返回多个结果）。参考手册条目还将列出每个参数：

- 其名称。
- 它所需的值的限定类型（我们使用参数来命名调用函数时传递给函数的值）。
- 是否需要该参数。

所有内置函数都在其签名中定义了一个或多个参数。并非每个函数都需要所有参数。

让我们看一下[ta.vwma\(\)](#)函数，它返回源值的成交量加权移动平均值。这是参考手册中的条目：

The screenshot shows the Pine Script v5 Reference Manual interface. A search bar at the top left contains the text "vwma". Below it, a sidebar on the left lists several functions, with "ta.vwma" highlighted in blue. The main content area on the right is titled "ta.vwma". It includes a brief description: "The vwma function returns volume-weighted moving average of `source` for `length` bars back. It is the same as: sma(source \* volume, length) / sma(volume, length)." Below this is a code snippet: "ta.vwma(source, length) → series float". A "EXAMPLE" section shows a Pine script example: "plot(ta.vwma(close, 15))". Further down, under "RETURNS", it says "Volume-weighted moving average of `source` for `length` bars back." Under "ARGUMENTS", it defines "source (series int/float) Series of values to process." and "length (series int) Number of bars (length)". At the bottom, a "SEE ALSO" section lists related functions: ta.sma, ta.emma, ta.rma, ta.wma, ta.swma, and ta.alma.

该条目为我们提供了使用它所需的信息：

- 该函数的作用是什么。
- 它的签名（或定义）：

```
ta.vwma(source, length) → series float
```

- 它包括的参数：`source` 和 `length`
- 它返回结果的限定类型：“series float”。
- 显示其使用情况的示例：。`plot(ta.vwma(close, 15))`
- 一个显示其功能的示例，但以长形式显示，以便您可以更好地理解其计算。请注意，这只是为了解释——而不是作为可用的代码，因为它更复杂并且执行时间更长。使用长格式只有缺点。
- “RETURNS”部分准确解释了函数返回的值。

- “ARGUMENTS”部分列出了每个参数，并提供了有关调用函数时使用的参数所需的限定类型的关键信息。
- “另请参阅”部分可让您参考相关的参考手册条目。

```
myVwma ~ 这是对声明变量并将结果分配给它的代码行中的函数的调用: `ta.vwma(close, 20)
```

Pine Script™

```
CopiedmyVwma = ta.vwma(close, 20)
```

注意:

- 我们使用内置变量`close`作为参数的参数`source`。
- 我们用作参数`20`的实参`length`。
- 如果放置在全局范围内（即，从一行的第一个位置开始），它将由 Pine Script™ 运行时在图表的每个柱上执行。

我们还可以在调用函数时使用参数名称。在函数调用中使用时，参数名称称为关键字参数：

```
myVwma = ta.vwma(source = close, length = 20)
```

使用关键字参数时，您可以更改参数的位置，但前提是您将它们用于所有参数。当调用具有多个参数的函数（例如[indicator\(\)](#)）时，您还可以放弃第一个参数的关键字参数，只要不跳过任何参数即可。如果您跳过某些参数，则必须使用关键字参数，以便 Pine Script™ 编译器能够找出它们对应的参数，例如：

```
indicator("Example", "Ex", true, max_bars_back = 100)
```

以这种方式混合是不允许的：

```
indicator(precision = 3, "Example") // Compilation error!
```

调用内置函数时，确保您使用的参数具有所需的限定类型至关重要，该类型因每个参数而异。

要了解如何做到这一点，需要了解 Pine Script™ 的[类型系统](#)。每个内置函数的参考手册条目都包含一个“参数”部分，其中列出了提供给每个函数参数的参数所需的限定类型。

## 用户自定义函数

### 介绍

用户定义函数是您编写的函数，与 Pine Script™ 中的内置函数不同。它们对于定义必须重复执行的计算或想要与脚本的主要计算部分隔离的计算非常有用。当没有内置函数可以满足您的需要时，请将用户定义的函数视为扩展 Pine Script™ 功能的一种方式。

您可以通过两种方式编写函数：

- 在一行中，当它们很简单时，或者
- 在多行上

函数可以位于两个地方：

- 如果某个函数仅在一个脚本中使用，您可以将其包含在使用该函数的脚本中。有关在脚本中放置函数的位置的建议，请参阅我们的[样式指南](#)。
- 您可以创建一个 Pine Script™[库](#)来包含您的函数，这使得它们可以在其他脚本中重用，而无需复制其代码。对库函数存在不同的要求。它们在[库](#)页面中进行了解释。

无论使用一行还是多行，用户定义函数都具有以下特征：

- 它们无法嵌入。所有函数都在脚本的全局范围内定义。
- 他们不支持递归。不允许函数从自己的代码中调用自身。
- 函数返回值的类型是自动确定的，并且取决于每个特定函数调用中使用的参数类型。
- 函数的返回值是函数体中的最后一个值。
- 脚本中函数调用的每个实例都维护其自己独立的历史记录。

## 单行函数

简单的函数通常可以用一行编写。这是单行函数的正式定义：

```
<function_declaration>
  <identifier>(<parameter_list>) => <return_value>

<parameter_list>
  {<parameter_definition>{}, <parameter_definition>{}}

<parameter_definition>
  [<identifier> = <default_value>]

<return_value>
  <statement> | <expression> | <tuple>
```

这是一个例子：

```
f(x, y) => x + y
```

声明函数后 `f()`，可以使用不同类型的参数调用它：

```
a = f(open, close)
b = f(2, 2)
c = f(open, 2)
```

在上面的示例中，变量的类型 `a` 是 *Series*，因为参数都是 *series*。变量的类型 `b` 是整数，因为参数都是文字整数。变量的类型 `c` 是系列，因为\*\*系列和文字整数相加会产生系列结果。

## 多行函数

Pine Script™ 还支持具有以下语法的多行函数：

```
<identifier>(<parameter_list>) =>
  <local_block>

<identifier>(<list of parameters>) =>
  <variable declaration>
  ...
  <variable declaration or expression>
```

在哪里：

```
<parameter_list>
  {<parameter_definition>{, <parameter_definition>}}

<parameter_definition>
  [<identifier> = <default_value>]
```

多行函数的主体由多个语句组成。每条语句都放在单独的行上，并且前面必须有 1 个缩进（4 个空格或 1 个制表符）。语句之前的缩进表明它是函数体的一部分，而不是脚本全局范围的一部分。在函数代码之后，第一个没有缩进的语句表示函数体已经结束。

表达式或声明的变量应该是函数体的最后一条语句。该表达式（或变量）的结果将是函数调用的结果。例如：

```
geom_average(x, y) =>
  a = x*x
  b = y*y
  math.sqrt(a + b)
```

该函数 `geom_average` 有两个参数，并在主体中创建两个变量：`a` 和 `b`。最后一条语句调用该函数 `math.sqrt`（提取平方根）。该 `geom_average` 调用将返回最后一个表达式的值：`。(math.sqrt(a + b))`

## 脚本中的范围

在函数体或其他局部块之外声明的变量属于全局范围。用户声明的函数和内置函数以及内置变量也属于全局范围。

每个函数都有自己的局部作用域。函数内声明的所有变量以及函数的参数都属于该函数的作用域，这意味着不可能从外部引用它们——例如，从全局作用域或另一个函数的局部作用域。

另一方面，由于可以从函数作用域引用全局作用域中声明的任何变量或函数（自引用递归调用除外），因此可以说局部作用域嵌入到全局作用域中。

在 Pine Script™ 中，不允许嵌套函数，即一个函数不能在另一个函数内声明一个函数。所有用户函数都在全局范围内声明。局部作用域不能相互交叉。

## 返回多个结果的函数

在大多数情况下，函数仅返回一个结果，但可以返回结果列表（类似元组的结果）：

```
fun(x, y) =>
    a = x+y
    b = x-y
    [a, b]
```

调用此类函数需要特殊语法：

```
[res0, res1] = fun(open, close)
plot(res0)
plot(res1)
```

## 限制

用户定义的函数可以使用任何 Pine Script™ 内置函数，除了：[barcolor\(\)](#)、[fill\(\)](#)、[hline\(\)](#)、[Indicator\(\)](#)、[library\(\)](#)、[plot\(\)](#)、[plotbar\(\)](#)、[plotcandle\(\)](#)、[plotchar\(\)](#)、[plotshape\(\)](#)和[strategy\(\)](#)。

## 对象

提示：

此页面包含高级材料。如果您是一名初级 Pine Script™ 程序员，我们建议您在冒险之前熟悉其他更易于使用的 Pine Script™ 功能。

## 介绍

Pine Script™ 对象是用户定义类型(UDT) 的实例。它们相当于包含称为*fields*的部分的变量，每个变量都能够保存各种类型的独立值。

有经验的程序员可以将 UDT 视为无方法类。它们允许用户创建自定义类型，在一个逻辑实体下组织不同的值。

## 创建对象

在创建对象之前，必须定义其类型。[类型系统](#)页面的用户[定义类型](#)部分 解释了如何执行此操作。

让我们定义一个 `pivotPoint` 类型来保存枢轴信息：

```
type pivotPoint
    int x
    float y
    string xloc = xloc.bar_time
```

注意：

- 我们使用[type](#)关键字来声明 UDT 的创建。
- 我们将新的 UDT 命名为 `pivotPoint`。

- 在第一行之后，我们创建一个包含每个字段的类型和名称的本地块。
- 该 `x` 字段将保存枢轴的 `x` 坐标。它被声明为“`int`”，因为它将保存“`int`”类型的时间戳或条形索引。
- `y` 是一个“浮动”，因为它将保持枢轴的价格。
- `xloc``x` 是一个字段，用于指定 `xloc.bar_index` 或 `xloc.bar_time` 的单位。我们使用运算符将其默认值设置为 `xloc.bar_time`。当从该 UDT 创建对象时，其字段将设置为该值。`=``xloc`

现在我们的 `pivotPoint` UDT 已定义，我们可以继续从中创建对象。我们使用 UDT 的 `new()` 内置方法创建对象。要从 UDT 创建新 `foundPoint` 对象 `pivotPoint`，我们使用：

```
foundPoint = pivotPoint.new()
```

我们还可以使用以下命令为创建的对象指定字段值：

```
foundPoint = pivotPoint.new(time, high)
```

或者等价的：

```
foundPoint = pivotPoint.new(x = time, y = high)
```

此时，该 `foundPoint` 对象的字段将包含创建时内置的 `时间` `x` 值，将包含 `high` 值，并且该字段将包含其默认值 `xloc.bar_time`，因为创建时没有为其定义值。目的。`y``xloc`

还可以通过使用以下内容声明对象名称来创建对象占位符：

```
pivotPoint foundPoint = na
```

此示例显示检测到高枢轴的标签。枢轴 `legsInput` 在出现后会被检测到，因此我们必须绘制过去的标签，以便它出现在枢轴上：

```
//@version=5
indicator("Pivot labels", overlay = true)
int legsInput = input(10)

// Define the `pivotPoint` UDT.
type pivotPoint
    int x
    float y
    string xloc = xloc.bar_time

// Detect high pivots.
pivotHighPrice = ta.pivothigh(legsInput, legsInput)
if not na(pivotHighPrice)
    // A new high pivot was found; display a label where it occurred `legsInput` bars
    // back.
    foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
    label.new(
```

```
foundPoint.x,  
foundPoint.y,  
str.toString(foundPoint.y, format.mintick),  
foundPoint.xloc,  
textcolor = color.white)
```

请注意上面示例中的这一行：

```
foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

也可以使用以下形式编写：

```
pivotPoint foundPoint = na  
foundPoint := pivotPoint.new(time[legsInput], pivotHighPrice)
```

[当使用var或varip创建对象时](#)，这些关键字适用于该对象的所有字段：

```
//@version=5  
indicator("")  
type barInfo  
    int i = bar_index  
    int t = time  
    float c = close  
  
// Created on bar zero.  
var firstBar = barInfo.new()  
// Created on every bar.  
currentBar = barInfo.new()  
  
plot(firstBar.i)  
plot(currentBar.i)
```

## 更改字段值

[可以使用:=重新赋值运算符](#)更改对象字段的值。

我们前面的例子中的这一行：

```
foundPoint = pivotPoint.new(time[legsInput], pivotHighPrice)
```

可以使用以下方式编写：

```
foundPoint = pivotPoint.new()  
foundPoint.x := time[legsInput]  
foundPoint.y := pivotHighPrice
```

## 收集对象

Pine Script™ 集合（[数组](#)、[矩阵](#)和[映射](#)）可以包含对象，允许用户向其数据结构添加虚拟维度。要声明对象集合，[请将 UDT 名称传递到其类型模板中](#)。

此示例声明一个空[数组](#)，该数组将保存 `pivotPoint` 用户定义类型的对象：

```
pivotHighArray = array.new<pivotPoint>()
```

要将变量的类型显式声明为[用户定义类型的数组](#)、[矩阵](#)或[映射](#)，请使用集合的 `type` 关键字，后跟其[类型模板](#)。例如：

```
var array<pivotPoint> pivotHighArray = na  
pivotHighArray := array.new<pivotPoint>()
```

让我们利用所学到的知识来创建一个检测高枢轴点的脚本。该脚本首先将历史数据透视表信息收集到一个[数组](#)中。然后，它循环遍历最后一个历史柱上的数组，为每个枢轴创建一个标签，并用线连接枢轴：



```
//@version=5  
indicator("Pivot Points High", overlay = true)  
  
int legsInput = input(10)  
  
// Define the `pivotPoint` UDT containing the time and price of pivots.  
type pivotPoint  
    int openTime  
    float level  
  
// Create an empty `pivotPoint` array.
```

```

var pivotHighArray = array.new<pivotPoint>()

// Detect new pivots (`na` is returned when no pivot is found).
pivotHighPrice = ta.pivothigh(legsInput, legsInput)

// Add a new `pivotPoint` object to the end of the array for each detected pivot.
if not na(pivotHighPrice)
    // A new pivot is found; create a new object of `pivotPoint` type, setting its
    `openTime` and `level` fields.
    newPivot = pivotPoint.new(time[legsInput], pivotHighPrice)
    // Add the new pivot object to the array.
    array.push(pivotHighArray, newPivot)

// On the last historical bar, draw pivot labels and connecting lines.
if barstate.islastconfirmedhistory
    var pivotPoint previousPoint = na
    for eachPivot in pivotHighArray
        // Display a label at the pivot point.
        label.new(eachPivot.openTime, eachPivot.level, str.tostring(eachPivot.level,
format.mintick), xloc.bar_time, textcolor = color.white)
        // Create a line between pivots.
        if not na(previousPoint)
            // Only create a line starting at the loop's second iteration because lines
connect two pivots.
            line.new(previousPoint.openTime, previousPoint.level, eachPivot.openTime,
eachPivot.level, xloc = xloc.bar_time)
            // Save the pivot for use in the next iteration.
            previousPoint := eachPivot

```

## 复制对象

在 Pine 中，对象是通过引用分配的。当现有对象分配给新变量时，两者都指向同一个对象。

在下面的示例中，我们创建一个 `pivot1` 对象并将其 `x` 字段设置为 1000。然后，我们声明一个 `pivot2` 包含对该 `pivot1` 对象的引用的变量，因此两者都指向同一个实例。因此，更改 `pivot2.x` 也会更改 `pivot1.x`，因为两者都引用 `x` 同一对象的字段：

```

//@version=5
indicator("")
type pivotPoint
    int x
    float y
pivot1 = pivotPoint.new()
pivot1.x := 1000
pivot2 = pivot1
pivot2.x := 2000
// Both plot the value 2000.
plot(pivot1.x)
plot(pivot2.x)

```

要创建独立于原始对象的副本，在这种情况下我们可以使用内置 `copy()` 方法。

在此示例中，我们声明 `pivot2` 引用 `pivot1` 对象的复制实例的变量。现在，改变 `pivot2.x` 不会改变 `pivot1.x`，因为它指的是一个单独对象的字段：

```

//@version=5
indicator("")
type pivotPoint
    int x
    float y
pivot1 = pivotPoint.new()
pivot1.x := 1000
pivot2 = pivotPoint.copy(pivot1)
pivot2.x := 2000
// Plots 1000 and 2000.
plot(pivot1.x)
plot(pivot2.x)

```

需要注意的是，内置 `copy()` 方法会生成对象的浅表副本。如果对象具有特殊类型的字段（[array](#)、[matrix](#)、[map](#)、[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#) 或 [Chart.point](#)），则该对象的浅拷贝中的这些字段将指向与该对象相同的实例。原来的。

在下面的示例中，我们定义了一个 `InfoLabel` 类型，其中一个字段为标签。该脚本实例化对象 `shallow` 的副本 `parent`，然后调用用户定义的 `set()` 方法来更新每个对象的 `info` 和 `lbl` 字段。由于 `lbl` 两个对象的字段都指向同一标签实例，因此任一对象中此字段的更改都会影响另一个对象：

```

//@version=5
indicator("Shallow Copy")

type InfoLabel
    string info
    label lbl

method set(InfoLabel this, int x = na, int y = na, string info = na) =>
    if not na(x)

```

```

    this.lbl.set_x(x)
    if not na(y)
        this.lbl.set_y(y)
    if not na(info)
        this.info := info
    this.lbl.set_text(this.info)

var parent = InfoLabel.new("", label.new(0, 0))
var shallow = parent.copy()

parent.set(bar_index, 0, "Parent")
shallow.set(bar_index, 1, "Shallow Copy")

```

要生成对象的深层副本，并使其所有特殊类型字段都指向独立实例，我们还必须显式复制这些字段。

在这个例子中，我们定义了一个 `deepCopy()` 方法来实例化一个新 `InfoLabel` 对象，其 `lbl` 字段指向原始字段的副本。对 `deep` 副本字段的更改 `lbl` 不会影响该 `parent` 对象，因为它指向一个单独的实例：

```

//@version=5
indicator("Deep Copy")

type InfoLabel
    string info
    label lbl

method set(InfoLabel this, int x = na, int y = na, string info = na) =>
    if not na(x)
        this.lbl.set_x(x)
    if not na(y)
        this.lbl.set_y(y)
    if not na(info)
        this.info := info
    this.lbl.set_text(this.info)

method deepCopy(InfoLabel this) =>
    InfoLabel.new(this.info, this.lbl.copy())

var parent = InfoLabel.new("", label.new(0, 0))
var deep   = parent.deepCopy()

parent.set(bar_index, 0, "Parent")
deep.set(bar_index, 1, "Deep Copy")

```

## 提示

避免未来添加到 Pine Script™ 的命名空间与现有脚本中的 UDT 或对象名称发生冲突的潜在冲突；通常，UDT 和对象名称会影响语言的命名空间。例如，UDT 或对象可以使用内置类型的名称，例如 [line](#) 或 [table](#)。

只有该语言的五种基本类型不能用于命名 UDT 或对象：[int](#)、[float](#)、[string](#)、[bool](#)和[color](#)。

# 方法

提示：

此页面包含高级材料。如果您是一名初级 Pine Script™ 程序员，我们建议您在冒险之前熟悉其他更易于使用的 Pine Script™ 功能。

## 介绍

Pine Script™ 方法是与内置或用户定义类型的特定实例相关的专用函数。它们在大多数方面与常规函数本质上相同，但提供了更短、更方便的语法。用户可以直接使用变量上的点表示法来访问方法，就像访问 Pine Script™[对象](#)的字段一样。

## 内置方法

Pine Script™ 包含适用于所有特殊类型的内置方法，包括[数组](#)、[矩阵](#)、[地图](#)、[线](#)、[线填充](#)、[框](#)、[折线](#)、[标签](#)和[表格](#)。这些方法为用户提供了一种更简洁的方法来在其脚本中调用这些类型的专用例程。

当使用这些特殊类型时，表达式：

```
<namespace>.<functionName>([paramName =] <objectName>, ...)
```

和：

```
<objectName>.<functionName>(...)
```

是等价的。例如，而不是使用：

```
array.get(id, index)
```

`id`要从指定的数组中获取值`index`，我们可以简单地使用：

```
id.get(index)
```

以达到同样的效果。这种表示法消除了用户引用函数的名称空间的需要，因为`get()``id`是此上下文中的一种方法。

下面编写一个实际示例来演示如何使用内置方法代替函数。

以下脚本根据每条`n`柱采样一次的指定数量的价格来计算布林线。它调用`array.push()`和`array.shift()`通过排列`sourceInput`值`sourceArray`，然后调用`array.avg()`和`array.stdev()`来计算`sampleMean`和`sampleDev`。然后，该脚本使用这些值来计算`highBand`和`lowBand`，并将其与一起绘制在图表上`sampleMean`：



```
//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput = input.source(close, "Source")
int samplesInput = input.int(20, "Samples")
int n = input.int(10, "Bars")
float multiplier = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)
var float sampleMean = na
var float sampleDev = na

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    array.push(sourceArray, sourceInput)
    array.shift(sourceArray)
    // Update the mean and standard deviation values.
    sampleMean := array.avg(sourceArray)
    sampleDev := array.stdev(sourceArray) * multiplier

// Calculate bands.
float highBand = sampleMean + sampleDev
float lowBand = sampleMean - sampleDev

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)
```

让我们重写这段代码以使用方法而不是内置函数。在此版本中，我们已将脚本中的所有内置array\_\*[函数](#)替换为等效方法：

```

//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput = input.source(close, "Source")
int samplesInput = input.int(20, "Samples")
int n = input.int(10, "Bars")
float multiplier = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)
var float sampleMean = na
var float sampleDev = na

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.push(sourceInput)
    sourceArray.shift()
    // Update the mean and standard deviaiton values.
    sampleMean := sourceArray.avg()
    sampleDev := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand = sampleMean - sampleDev

plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)

```

- 注意：

`sourceArray.*` 我们使用而不是引用数组命名空间来调用数组方法。当我们调用这些方法时，我们不会将其 `sourceArray` 作为参数包含在内，因为它们已经引用了该对象。

## 用户定义的方法

Pine Script™ 允许用户定义与任何内置或用户定义类型的对象一起使用的自定义方法。定义方法本质上与定义函数相同，但有两个关键区别：

- method关键字必须包含在函数名称之前。
- 签名中第一个参数的类型必须显式声明，因为它表示该方法将与之关联的对象的类型。

```

[export] method <functionName>(<paramType> <paramName> [= <defaultValue>], ...) =>
    <functionBlock>

```

让我们将用户定义的方法应用到之前的布林线示例中，以封装全局范围内的操作，这将简化代码并提高可重用性。请参阅示例中的这一部分：

```

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.push(sourceInput)
    sourceArray.shift()
    // Update the mean and standard deviation values.
    sampleMean := sourceArray.avg()
    sampleDev  := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand  = sampleMean - sampleDev

```

我们将首先定义一个简单的方法，在一次调用中通过数组对值进行排队。

当为 true 时，此 `maintainQueue()` 方法调用 `push()` 和 `shift()` 方法并返回对象： `srcArray``takeSample`

```

// @function      Maintains a queue of the size of `srcArray`.
//                It appends a `value` to the array and removes its oldest element
at position zero.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param value   (float) The new value to be added to the queue.
//                The queue's oldest value is also removed, so its size is constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this is
true.
// @returns        (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

```

- 注意：

就像用户定义的函数一样，我们使用 `@function` 编译器注释来记录方法描述。

现在我们可以在示例中替换 `sourceArray.push()` and

`: sourceArray.shift()``sourceArray.maintainQueue()`

```

// Identify if `n` bars have passed.
if bar_index % n == 0
    // Update the queue.
    sourceArray.maintainQueue(sourceInput)
    // Update the mean and standard deviaiton values.
    sampleMean := sourceArray.avg()
    sampleDev := sourceArray.stdev() * multiplier

// Calculate band values.
float highBand = sampleMean + sampleDev
float lowBand = sampleMean - sampleDev

```

从这里开始，我们将通过定义一个处理其范围内所有布林带计算的方法来进一步简化我们的代码。

当为true时，此 calcBB() 方法调用 avg() 和 stdev() 方法来 srcArray 更新 mean 和 dev 值。 calculate 该方法使用这些值返回一个分别包含基础值、上限值和下限值的元组：

```

// @function      Computes Bollinger Band values from an array of data.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviaiton multiplier.
// @param calculate (bool) The method will only calculate new values when this is
// true.
// @returns        A tuple containing the basis, upper band, and lower band
// respectively.
method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
    var float mean = na
    var float dev = na
    if calculate
        // Compute the mean and standard deviation of the array.
        mean := srcArray.avg()
        dev := srcArray.stdev() * mult
    [mean, mean + dev, mean - dev]

```

通过这种方法，我们现在可以从全局范围内删除布林带计算并提高代码可读性：

```

// Identify if `n` bars have passed.
bool newSample = bar_index % n == 0

// Update the queue and compute new BB values on each new sample.
[sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput,
newSample).calcBB(multiplier, newSample)

```

- 注意：

我们没有 if 在全局范围内使用块，而是定义了一个 newSample 每条柱只为 true 一次的变量 n。和方法将此值用作各自的 maintainQueue() 和参数。 calcBB() `` takeSample `` calculate 由于该 maintainQueue() 方法返回它引用的对象，因此我们可以 calcBB() 从同一行代码进行调用，因为这两种方法都适用于 array<float> 实例。

现在我们已经应用了用户定义的方法，完整的脚本示例如下所示：

```
//@version=5
indicator("Custom Sample BB", overlay = true)

float sourceInput = input.source(close, "Source")
int samplesInput = input.int(20, "Samples")
int n = input.int(10, "Bars")
float multiplier = input.float(2.0, "StdDev")

var array<float> sourceArray = array.new<float>(samplesInput)

// @function      Maintains a queue of the size of `srcArray`.
//                 It appends a `value` to the array and removes its oldest element
at position zero.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param value    (float) The new value to be added to the queue.
//                 The queue's oldest value is also removed, so its size is constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this is
true.
// @returns        (array<float>) `srcArray` object.
method maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

// @function      Computes Bollinger Band values from an array of data.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param multiplier (float) Standard deviation multiplier.
// @param calculate (bool) The method will only calculate new values when this is
true.
// @returns        A tuple containing the basis, upper band, and lower band
respectively.
method calcBB(array<float> srcArray, float mult, bool calculate = true) =>
    var float mean = na
    var float dev = na
    if calculate
        // Compute the mean and standard deviation of the array.
        mean := srcArray.avg()
        dev := srcArray.stdev() * mult
    [mean, mean + dev, mean - dev]

    // Identify if `n` bars have passed.
    bool newSample = bar_index % n == 0

    // Update the queue and compute new BB values on each new sample.
    [sampleMean, highBand, lowBand] = sourceArray.maintainQueue(sourceInput,
newSample).calcBB(multiplier, newSample)
```

```
plot(sampleMean, "Basis", color.orange)
plot(highBand, "Upper", color.lime)
plot(lowBand, "Lower", color.red)
```

## 方法重载

用户定义的方法可以覆盖和重载具有相同标识符的现有内置方法和用户定义的方法。此功能允许用户在同一方法名称下定义与不同参数签名关联的多个例程。

作为一个简单的例子，假设我们想要定义一个方法来识别变量的类型。由于我们必须显式指定与用户定义方法关联的对象类型，因此我们需要为我们希望它识别的每种类型定义重载。

下面，我们定义了一个 `getType()` 方法，该方法返回变量类型的字符串表示形式，并具有五种基本类型的重载：

```
// @function Identifies an object's type.
// @param this Object to inspect.
// @returns (string) A string representation of the type.
method getType(int this) =>
    na(this) ? "int(na)" : "int"

method getType(float this) =>
    na(this) ? "float(na)" : "float"

method getType(bool this) =>
    na(this) ? "bool(na)" : "bool"

method getType(color this) =>
    na(this) ? "color(na)" : "color"

method getType(string this) =>
    na(this) ? "string(na)" : "string"
```

现在我们可以使用这些重载来检查一些变量。此脚本使用 `str.format()` 将在五个不同变量上调用该方法的结果格式化 `getType()` 为单个字符串，然后使用内置 `set_text()` `results` 方法在标签中显示该字符串： `lbl`



```

indicator("Type Inspection")

// @function Identifies an object's type.
// @param this Object to inspect.
// @returns (string) A string representation of the type.
method getType(int this) =>
    na(this) ? "int(na)" : "int"

method getType(float this) =>
    na(this) ? "float(na)" : "float"

method getType(bool this) =>
    na(this) ? "bool(na)" : "bool"

method getType(color this) =>
    na(this) ? "color(na)" : "color"

method getType(string this) =>
    na(this) ? "string(na)" : "string"

a = 1
b = 1.0
c = true
d = color.white
e = "1"

// Inspect variables and format results.
results = str.format(
    "a: {0}\nb: {1}\nnc: {2}\nnd: {3}\nne: {4}",
    a.getType(), b.getType(), c.getType(), d.getType(), e.getType()
)

var label lbl = label.new(0, 0)
lbl.set_x(bar_index)
lbl.set_text(results)

```

- 注意：

每个变量的基础类型决定 `getType()` 编译器将使用哪种重载。当变量要标定它为空时，该方法会将“(na)”附加到输出字符串。

## 高级示例

让我们应用我们所学到的知识来构建一个脚本，用于估计数组中元素的累积分布，即数组中小于或等于任何给定值的元素的分数。

我们可以选择多种方式来实现这一目标。对于这个例子，我们将首先定义一个方法来替换数组的元素，这将帮助我们计算某个值范围内元素的出现次数。

下面写的是实例的内置 `fill()` 方法的重载 `array<float>`。此重载将 `a` 中的元素替换为和之间 `srcArray` 的范围内的元素，并用替换该范围之外的所有元素： `lowerBound``upperBound``innerValue``outerValue`

```
// @function      Replaces elements in a `srcArray` between `lowerBound` and
//                `upperBound` with an `innerValue`,
//                and replaces elements outside the range with an `outerValue`.
// @param srcArray (array<float>) Array to modify.
// @param innerValue (float) Value to replace elements within the range with.
// @param outerValue (float) Value to replace elements outside the range with.
// @param lowerBound (float) Lowest value to replace with `innerValue`.
// @param upperBound (float) Highest value to replace with `innerValue`.
// @returns        (array<float>) `srcArray` object.

method fill(array<float> srcArray, float innerValue, float outerValue, float
lowerBound, float upperBound) =>
    for [i, element] in srcArray
        if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or
na(upperBound))
            srcArray.set(i, innerValue)
        else
            srcArray.set(i, outerValue)
    srcArray
```

使用此方法，我们可以按值范围过滤数组以生成出现次数的数组。例如，表达式：

```
srcArray.copy().fill(1.0, 0.0, min, val)
```

复制 `srcArray` 对象，将 `1.0 min` 和 `1.0` 之间的所有元素替换 `val` 为 `1.0`，然后将上面的所有元素替换 `val` 为 `0.0`。从这里，很容易估计 处累积分布函数的输出 `val`，因为它只是结果数组的平均值：

```
srcArray.copy().fill(1.0, 0.0, min, val).avg()
```

- 注意：

`fill()` 当用户在调用中提供 `innerValue`、`outerValue`、`lowerBound` 和参数时，编译器将仅使用此重载而不是内置重载 `upperBound`。如果 或 `lowerBound` 是 `upperBound`，`na` 则在过滤填充范围时忽略其值。我们能够在同一行代码上连续调用 `copy()`、`fill()`、 和，因为前两个方法返回一个实例。`avg()``array<float>`

我们现在可以用它来定义一种方法来计算我们的经验分布值。以下方法从 `a` 的累积分布函数 `eCDF()` 估计多个均匀分布的升序，并将结果推入 `a`：`steps``srcArray``cdfArray`

```
// @function      Estimates the empirical CDF of a `srcArray`.
// @param srcArray (array<float>) Array to calculate on.
// @param steps    (int) Number of steps in the estimation.
// @returns        (array<float>) Array of estimated CDF ratios.

method eCDF(array<float> srcArray, int steps) =>
    float min = srcArray.min()
    float rng = srcArray.range() / steps
```

```

array<float> cdfArray = array.new<float>()
// Add averages of `srcArray` filtered by value region to the `cdfArray`.
float val = min
for i = 1 to steps
    val += rng
    cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
cdfArray

```

最后，为了确保我们的 `eCDF()` 方法对于包含小值和大值的数组正常运行，我们将定义一个方法来规范化我们的数组。

此 `featureScale()` 方法使用数组 `min()` 和 `range()` 方法来生成 `srcArray`。在调用该方法之前，我们将使用它来标准化我们的数组 `eCDF()`：

```

// @function      Rescales the elements within a `srcArray` to the interval [0, 1].
// @param srcArray (array<float>) Array to normalize.
// @returns       (array<float>) Normalized copy of the `srcArray`.
method featureScale(array<float> srcArray) =>
    float min = srcArray.min()
    float rng = srcArray.range()
    array<float> scaledArray = array.new<float>()
    // Push normalized `element` values into the `scaledArray`.
    for element in srcArray
        scaledArray.push((element - min) / rng)
    scaledArray

```

- 注意：

此方法不包括对除零条件的特殊处理。如果 `rng` 为 0，则数组元素的值为 `na`。

下面的完整示例使用我们之前的方法 `sourceArray` 对 size 的 `length` 值进行排队，使用该方法对数组的元素进行标准化，然后调用该方法来获取分布上均匀间隔步长的估计值数组。然后，该脚本调用用户定义的函数以在图表右侧的标签中显示估计值和价

格：`sourceInput``maintainQueue()``featureScale()``eCDF()``n``makeLabel()`



```

//@version=5
indicator("Empirical Distribution", overlay = true)

float sourceInput = input.source(close, "Source")
int length      = input.int(20, "Length")
int n           = input.int(20, "Steps")

// @function      Maintains a queue of the size of `srcArray`.
//                It appends a `value` to the array and removes its oldest element
// at position zero.
// @param srcArray (array<float>) The array where the queue is maintained.
// @param value    (float) The new value to be added to the queue.
// 
//                The queue's oldest value is also removed, so its size is constant.
// @param takeSample (bool) A new `value` is only pushed into the queue if this is
// true.
// @returns        (array<float>) `srcArray` object.
method array<float> maintainQueue(array<float> srcArray, float value, bool takeSample = true) =>
    if takeSample
        srcArray.push(value)
        srcArray.shift()
    srcArray

// @function      Replaces elements in a `srcArray` between `lowerBound` and
// `upperBound` with an `innerValue`,
//                and replaces elements outside the range with an `outerValue`.
// @param srcArray (array<float>) Array to modify.
// @param innerValue (float) Value to replace elements within the range with.
// @param outerValue (float) Value to replace elements outside the range with.
// @param lowerBound (float) Lowest value to replace with `innerValue`.
// @param upperBound (float) Highest value to replace with `innerValue`.
// @returns        (array<float>) `srcArray` object.

```

```

method fill(array<float> srcArray, float innerValue, float outerValue, float
lowerBound, float upperBound) =>
    for [i, element] in srcArray
        if (element >= lowerBound or na(lowerBound)) and (element <= upperBound or
na(upperBound))
            srcArray.set(i, innerValue)
        else
            srcArray.set(i, outerValue)
srcArray

// @function      Estimates the empirical CDF of a `srcArray`.
// @param srcArray (array<float>) Array to calculate on.
// @param steps     (int) Number of steps in the estimation.
// @returns        (array<float>) Array of estimated CDF ratios.
method eCDF(array<float> srcArray, int steps) =>
    float min = srcArray.min()
    float rng = srcArray.range() / steps
    array<float> cdfArray = array.new<float>()
    // Add averages of `srcArray` filtered by value region to the `cdfArray`.
    float val = min
    for i = 1 to steps
        val += rng
        cdfArray.push(srcArray.copy().fill(1.0, 0.0, min, val).avg())
    cdfArray

// @function      Rescales the elements within a `srcArray` to the interval [0, 1].
// @param srcArray (array<float>) Array to normalize.
// @returns        (array<float>) Normalized copy of the `srcArray`.
method featureScale(array<float> srcArray) =>
    float min = srcArray.min()
    float rng = srcArray.range()
    array<float> scaledArray = array.new<float>()
    // Push normalized `element` values into the `scaledArray`.
    for element in srcArray
        scaledArray.push((element - min) / rng)
    scaledArray

// @function      Draws a label containing eCDF estimates in the format "{price}:
// {percent}%"
// @param srcArray (array<float>) Array of source values.
// @param cdfArray (array<float>) Array of CDF estimates.
// @returns        (void)
makeLabel(array<float> srcArray, array<float> cdfArray) =>
    float max      = srcArray.max()
    float rng      = srcArray.range() / cdfArray.size()
    string results = ""
    var label lbl  = label.new(0, 0, "", style = label.style_label_left,
text_font_family = font.family_monospace)
    // Add percentage strings to `results` starting from the `max`.

```

```

cdfArray.reverse()
for [i, element] in cdfArray
    results += str.format("{0}: {1}%\n", max - i * rng, element * 100)
// Update `lbl` attributes.
lbl.set_xy(bar_index + 1, srcArray.avg())
lbl.set_text(results)

var array<float> sourceArray = array.new<float>(length)

// Add background color for the last `length` bars.
bgcolor(bar_index > last_bar_index - length ? color.new(color.orange, 80) : na)

// Queue `sourceArray`, feature scale, then estimate the distribution over `n` steps.
array<float> distArray = sourceArray.maintainQueue(sourceInput).featureScale().eCDF(n)
// Draw label.
makeLabel(sourceArray, distArray)

```

# 数组

---

提示：

此页面包含高级材料。如果您是一名初级 Pine Script™ 程序员，我们建议您在冒险之前熟悉其他更易于使用的 Pine Script™ 功能。

## 介绍

---

Pine Script™ 数组是可以保存多个值引用的一维集合。将它们视为处理需要显式声明一组相似变量（例如 `price00, , , , ...`）`price01` 的情况的更好方法 `price02`。

数组中的所有元素必须属于同一类型，可以是 [内置类型](#) 或 [用户定义类型](#)，始终限定为“系列”。脚本使用类似于行、标签和其他特殊类型的 ID 的数组 ID 来引用数组。Pine Script™ 不使用索引运算符来引用各个数组元素。相反，包括 [array.get\(\)](#) 和 [array.set\(\)](#) 在内的函数读取和写入数组元素的值。我们可以在允许“系列”值的表达式和函数中使用数组值。

脚本使用索引引用数组的元素，索引从 0 开始，扩展到数组中元素的数量减一。Pine Script™ 中的数组可以具有随条形变化的动态大小，因为可以在脚本的每次迭代中更改数组中的元素数量。脚本可以包含多个数组实例。数组的大小限制为 100,000 个元素。

### 笔记

我们将使用数组的开头来指定索引 0，使用数组的\*\*结尾来指定具有最高索引值的数组元素。为了简洁起见，我们还将扩展数组的含义以包括数组 ID。

## 声明数组

---

Pine Script™ 使用以下语法来声明数组：

```
[var/varip ][array<type>/<type[ ]> ]<identifier> = <expression>
```

其中 `<type>` 是数组的 [类型模板](#)，声明它将包含的值的类型，并且 `<expression>` 返回指定类型的数组或 `na`。

将变量声明为数组时，我们可以使用 `array` 关键字，后跟 [类型模板](#)。或者，我们可以使用 `type` 名称后跟 `[]` 修饰符（不要与 `[]` 历史引用运算符混淆）。

由于 Pine 总是使用特定于类型的函数来创建数组，因此 `array<type>/type[]` 声明部分是多余的，除非声明分配给的数组变量 `na`。即使不需要，显式声明数组类型也有助于向读者清楚地说明意图。

这行代码声明了一个名为 `prices` 的数组变量，`prices` 该变量指向 `na`。在这种情况下，我们必须指定类型来声明变量可以引用包含“float”值的数组：

```
array<float> prices = na
```

我们还可以将上面的例子写成这样的形式：

```
float[] prices = na
```

声明数组且 `<expression>` is not `na`，请使用以下函数之一：[array.new\(size, initial\\_value\)](#)、[array.from\(\)](#) 或[array.copy\(\)](#)。对于函数，和参数的参数可以是“系列”，以允许动态调整数组元素的大小和初始化。以下示例创建一个包含零个“float”元素的数组，这次，将[array.new\(\)](#) 函数调用返回的数组 ID 分配给：`array.new<type>(size, initial_value)` `size` `initial_value` `prices`

```
prices = array.new<float>(0)
```

## 笔记

命名 `array.*` 空间还包含用于创建数组的特定于类型的函数，包括 [array.new\\_int\(\)](#)、[array.new\\_float\(\)](#)、[array.new\\_bool\(\)](#)、[array.new\\_color\(\)](#)、[array.new\\_string\(\)](#)、[array.new\\_line\(\)](#)、[array.new\\_linefill\(\)](#)、[array.new\\_label\(\)](#)、[array.new\\_box\(\)](#) 和 [array.new\\_table\(\)](#)。`array.new()` 函数可以创建任何类型的数组，包括[用户定义的类型](#)。

`initial_value` 函数的参数允许 `array.new*` 用户将数组中的所有元素设置为指定值。如果没有为 提供参数 `initial_value`，则数组将填充 `na` 值。

此行声明一个名为 `prices` 的数组 ID，该数组 ID `prices` 指向包含两个元素的数组，每个元素都分配给条的 `close` 值：

```
prices = array.new<float>(2, close)
```

要创建数组并使用不同的值初始化其元素，请使用 [array.from\(\)](#)。该函数根据函数调用中的参数推断数组的大小和它将保存的元素类型。与 `array.new*` 函数一样，它接受“系列”参数。提供给函数的所有值必须具有相同的类型。

例如，所有这三行代码将创建具有相同两个元素的相同“bool”数组：

```
statesArray = array.from(close > open, high != close)
bool[] statesArray = array.from(close > open, high != close)
array<bool> statesArray = array.from(close > open, high != close)
```

## 使用 `var` 和 `varip` 关键字

用户可以利用`var`和`varip`关键字指示脚本在第一个图表条上的脚本的第一次迭代中仅声明一次数组变量。使用这些关键字声明的数组变量指向相同的数组实例，直到显式重新分配为止，从而允许数组及其元素引用跨柱保留。

当使用这些关键字声明数组变量并将新值推入每个柱上引用数组的末尾时，该数组将在每个柱上增长一，并且在脚本执行时其大小（`bar_index`从零开始）最后一栏，如以下代码所示：`bar_index + 1`

```
//@version=5
indicator("Using `var`")
//@variable An array that expands its size by 1 on each bar.
var a = array.new<float>(0)
array.push(a, close)

if barstate.islast
    //@variable A string containing the size of `a` and the current `bar_index` value.
    string labelText = "Array size: " + str.tostring(a.size()) + "\nbar_index: " +
str.tostring(bar_index)
    // Display the `labelText`.
    label.new(bar_index, 0, labelText, size = size.large)
```

没有`var`关键字的相同代码将在每个柱上重新声明数组。在这种情况下，执行`array.push()`调用后，`a.size()`调用将返回值1。

### 笔记

使用`varip`声明的数组变量的行为与在历史数据上使用`var`的行为相同，但它们会在每个新的价格变动时更新实时柱（即自脚本上次编译以来的柱）的值。分配给`varip`变量的数组只能保存`int`、`float`、`bool`、`color`或`string`类型或[用户定义类型](#)，这些类型在其字段中专门包含这些类型或这些类型的集合（数组、[矩阵](#)或[映射](#)）。

## 读写数组元素

脚本可以使用`array.set(id, index, value)`将值写入现有的各个数组元素，并使用`array.get(id, index)`读取值。使用这些函数时，函数调用中的必须`index`始终小于或等于数组的大小（因为数组索引从零开始）。要获取数组的大小，请使用`array.size(id)`函数。

以下示例使用`set()`方法使用不同透明度级别的一种基色实例填充数组。然后，它使用`array.get()`根据最后一根柱中价格最高的柱的位置从数组中检索一种颜色`lookbackInput`：



```
//@version=5
indicator("Distance from high", "", true)
lookbackInput = input.int(100)
FILL_COLOR = color.green
// Declare array and set its values on the first bar only.
var fillColors = array.new<color>(5)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill
    // color.
    fillColors.set(0, color.new(FILL_COLOR, 70))
    fillColors.set(1, color.new(FILL_COLOR, 75))
    fillColors.set(2, color.new(FILL_COLOR, 80))
    fillColors.set(3, color.new(FILL_COLOR, 85))
    fillColors.set(4, color.new(FILL_COLOR, 90))

// Find the offset to highest high. Change its sign because the function returns a
// negative value.
lastHiBar = - ta.highestbars(high, lookbackInput)
// Convert the offset to an array index, capping it to 4 to avoid a runtime error.
// The index used by `array.get()` will be the equivalent of `floor(fillNo)`.
fillNo = math.min(lastHiBar / (lookbackInput / 5), 4)
// Set background to a progressively lighter fill with increasing distance from
// location of highest high.
bgcolor(array.get(fillColors, fillNo))
// Plot key values to the Data Window for debugging.
plotchar(lastHiBar, "lastHiBar", "", location.top, size = size.tiny)
plotchar(fillNo, "fillNo", "", location.top, size = size.tiny)
```

初始化数组中元素的另一种技术是创建一个空数组（没有元素的数组），然后使用[array.push\(\)](#)将新元素追加到数组末尾，从而将数组的大小增加一每次通话。以下代码在功能上与前面脚本中的初始化部分相同：

```
// Declare array and set its values on the first bar only.
var fillColors = array.new<color>(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill
color.
    array.push(fillColors, color.new(FILL_COLOR, 70))
    array.push(fillColors, color.new(FILL_COLOR, 75))
    array.push(fillColors, color.new(FILL_COLOR, 80))
    array.push(fillColors, color.new(FILL_COLOR, 85))
    array.push(fillColors, color.new(FILL_COLOR, 90))
```

此代码与上面的代码等效，但它使用[array.unshift\(\)](#)在数组的开头插入新元素 `fillColors`：

```
// Declare array and set its values on the first bar only.
var fillColors = array.new<color>(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill
color.
    array.unshift(fillColors, color.new(FILL_COLOR, 90))
    array.unshift(fillColors, color.new(FILL_COLOR, 85))
    array.unshift(fillColors, color.new(FILL_COLOR, 80))
    array.unshift(fillColors, color.new(FILL_COLOR, 75))
    array.unshift(fillColors, color.new(FILL_COLOR, 70))
```

我们还可以使用[array.from\(\)](#) `fillColors` 通过单个函数调用创建相同的数组：

```
//@version=5
indicator("Using `var`")
FILL_COLOR = color.green
var array<color> fillColors = array.from(
    color.new(FILL_COLOR, 70),
    color.new(FILL_COLOR, 75),
    color.new(FILL_COLOR, 80),
    color.new(FILL_COLOR, 85),
    color.new(FILL_COLOR, 90)
)
// Cycle background through the array's colors.
bgcolor(array.get(fillColors, bar_index % (fillColors.size())))
```

`array.fill (id, value, index_from, index_to)` `index_from` 函数将所有数组元素或 `to` 范围内的元素 `index_to` 指向指定的 `value`。如果没有最后两个可选参数，该函数将填充整个数组，因此：

```
a = array.new<float>(10, close)
```

和：

```
a = array.new<float>(10)
a.fill(close)
```

是等价的，但是：

```
a = array.new<float>(10)
a.fill(close, 1, 3)
```

仅用 填充数组的第二个和第三个元素（位于索引 1 和 2 处）`close`。请注意`array.fill()`的最后一个参数`index_to`必须比函数将填充的最后一个索引大 1。其余元素将保存`na`值，因为`array.new()`函数调用不包含`initial_value`参数。

## 循环遍历数组元素

当循环遍历数组的元素索引并且数组的大小未知时，可以使用`array.size()`函数来获取最大索引值。例如：

```
//@version=5
indicator("Protected `for` loop", overlay = true)
//@variable An array of `close` prices from the 1-minute timeframe.
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

//@variable A string representation of the elements in `a`.
string labelText = ""
for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
    labelText += str.tostring(array.get(a, i)) + "\n"

label.new(bar_index, high, text = labelText)
```

- 注意：

我们使用`request.security_lower_tf()`函数，该函数返回该时间范围内的收盘价数组。`1 minute`如果您在小于的图表时间范围上使用此代码示例，则会引发错误。`1 minute``to`如果表达式为`na`，则不会执行`for`循环。请注意，该`to`值仅在输入时评估一次。

循环数组的另一种方法是使用`for...in`循环。此方法是标准`for`循环的变体，可以迭代数组中的值引用和索引。下面是我们如何使用循环编写上面的代码示例的示例`for...in`：

```

//@version=5
indicator(`for...in` loop, overlay = true)
//@variable An array of `close` prices from the 1-minute timeframe.
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

//@variable A string representation of the elements in `a`.
string labelText = ""
for price in a
    labelText += str.tostring(price) + "\n"

label.new(bar_index, high, text = labelText)

```

- 注意：

`for...in` 循环可以返回一个包含每个索引和相应元素的元组。例如，返回中每个元素的索引和值。`for [i, price] in a`i``price``a`

也可以使用while循环语句：

```

//@version=5
indicator(`while` loop, overlay = true)
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

string labelText = ""
int i = 0
while i < array.size(a)
    labelText += str.tostring(array.get(a, i)) + "\n"
    i += 1

label.new(bar_index, high, text = labelText)

```

## 范围

用户可以在脚本的全局范围内声明数组，也可以在[函数](#)、[方法](#)和[条件结构](#)的局部范围内声明数组。与其他一些内置类型（即基本类型）不同，脚本可以在本地范围内修改全局分配的数组，从而允许用户实现脚本中的任何函数都可以直接与之交互的全局变量。我们使用此处的功能来计算逐渐降低或升高的价格水平：



```

//@version=5
indicator("Bands", "", true)
//@variable The distance ratio between plotted price levels.
factorInput = 1 + (input.float(-2., "Step %") / 100)
//@variable A single-value array holding the lowest `ohlc4` value within a 50 bar
window from 10 bars back.
level = array.new<float>(1, ta.lowest(ohlc4, 50)[10])

nextLevel(val) =>
    newLevel = level.get(0) * val
    // Write new level to the global `level` array so we can use it as the base in the
    next function call.
    level.set(0, newLevel)
    newLevel

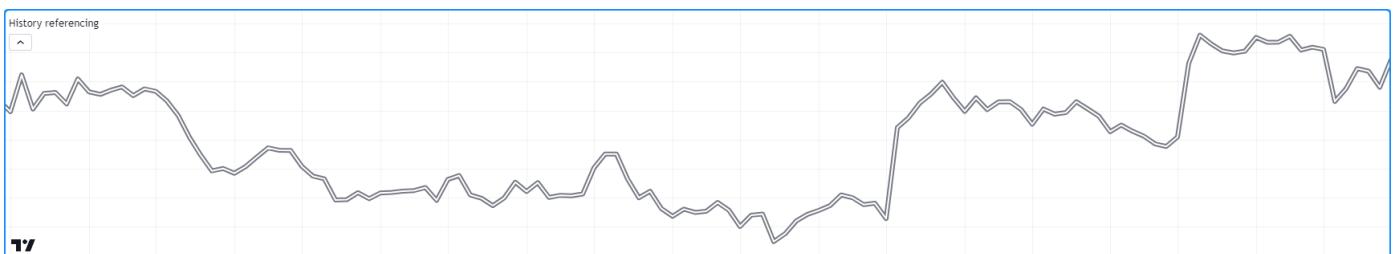
plot(nextLevel(1))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))

```

## 历史参考

Pine Script™ 的历史引用运算符 `[]` 可以访问数组变量的历史记录，从而允许脚本与先前分配给变量的过去数组实例进行交互。

为了说明这一点，让我们创建一个简单的示例来展示如何以 `close` 两种等效的方式获取前一柱的值。该脚本使用 `[]` 运算符获取分配给 `a` 前一柱的数组实例，然后使用 `get()` 方法检索第一个元素 () 的值 `previousClose1`。对于 `previousClose2`，我们直接对变量使用历史引用运算符 `close` 来检索值。正如我们从图中看到的，`previousClose1` 两者 `previousClose2` 都返回相同的值：



```

//@version=5
indicator("History referencing")

//@variable A single-value array declared on each bar.
a = array.new<float>(1)
// Set the value of the only element in `a` to `close`.
array.set(a, 0, close)

//@variable The array instance assigned to `a` on the previous bar.
previous = a[1]

```

```

previousClose1 = na(previous) ? na : previous.get(0)
previousClose2 = close[1]

plot(previousClose1, "previousClose1", color.gray, 6)
plot(previousClose2, "previousClose2", color.white, 2)

```

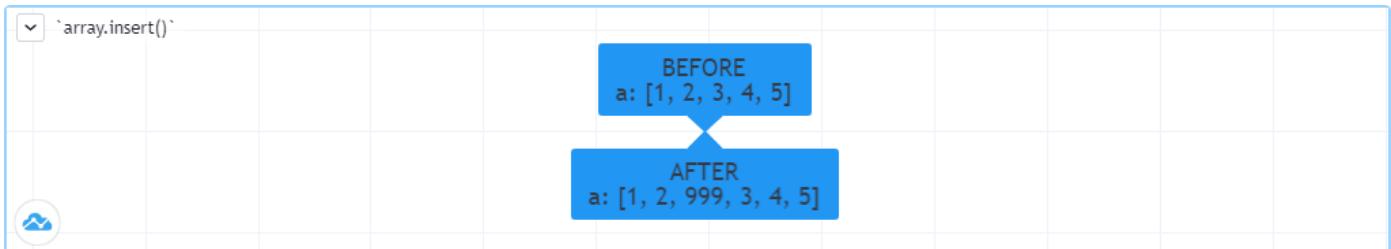
## 插入和删除数组元素

### 插入

以下三个函数可以向数组中插入新元素。

[array.unshift\(\)](#) 在数组的开头插入一个新元素（索引 0），并将任何现有元素的索引值加一。

[array.insert\(\)](#) 在指定位置插入一个新元素 `index`，并将该位置或之后的现有元素的索引增加 `index` 1。



```

//@version=5
indicator(`array.insert()`)
a = array.new<float>(5, 0)
for i = 0 to 4
    array.set(a, i, i + 1)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a), size = size.large)
    array.insert(a, 2, 999)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a), style =
label.style_label_up, size = size.large)

```

[array.push\(\)](#) 在数组末尾添加一个新元素。

### 删除

这四个函数从数组中删除元素。前三个还返回被删除元素的值。

[array.remove\(\)](#) 删除指定位置的元素 `index` 并返回该元素的值。

[array.shift\(\)](#) 从数组中删除第一个元素并返回其值。

[array.pop\(\)](#) 删除数组的最后一个元素并返回其值。

[array.clear\(\)](#) 从数组中删除所有元素。请注意，清除数组不会删除其元素引用的任何对象。请参阅下面的示例来说明其工作原理：

```
//@version=5
indicator(`array.clear()` example`, overlay = true)

// Create a label array and add a label to the array on each new bar.
var a = array.new<label>()
label lbl = label.new(bar_index, high, "Text", color = color.red)
array.push(a, lbl)

var table t = table.new(position.top_right, 1, 1)
// Clear the array on the last bar. This doesn't remove the labels from the chart.
if barstate.islast
    array.clear(a)
    table.cell(t, 0, 0, "Array elements count: " + str.tostring(array.size(a)), bgcolor = color.yellow)
```

## 使用数组作为堆栈

堆栈是 LIFO（后进先出）结构。它们的行为有点像一堆垂直的书，一次只能添加或删除一本书，而且总是从顶部开始。Pine Script™ 数组可以用作堆栈，在这种情况下，我们使用 [array.push\(\)](#) 和 [array.pop\(\)](#) 函数在数组末尾添加和删除元素。

`array.push(prices, close)` 将在数组末尾添加一个新元素 `prices`，从而将数组的大小增加一。

`array.pop(prices)` 将从数组中删除末尾元素 `prices`，返回其值并将数组大小减一。

了解此处如何使用这些函数来跟踪反弹中的连续低点：



```
//@version=5
indicator("Lows from new highs", "", true)
var lows = array.new<float>(0)
```

```

flushLows = false

// Remove last element from the stack when `_cond` is true.
array_pop(id, cond) => cond and array.size(id) > 0 ? array.pop(id) : float(na)

if ta.rising(high, 1)
    // Rising highs; push a new low on the stack.
    lows.push(low)
    // Force the return type of this `if` block to be the same as that of the next
block.
    bool(na)
else if lows.size() >= 4 or low < array.min(lows)
    // We have at least 4 lows or price has breached the lowest low;
    // sort lows and set flag indicating we will plot and flush the levels.
    array.sort(lows, order.ascending)
    flushLows := true

// If needed, plot and flush lows.
lowLevel = array_pop(lows, flushLows)
plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2,
plot.style_linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3,
plot.style_linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4,
plot.style_linebr)
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 4", low > lowLevel ? color.silver : color.purple, 5,
plot.style_linebr)

if flushLows
    // Clear remaining levels after the last 4 have been plotted.
    lows.clear()

```

## 使用数组作为队列

队列是 FIFO（先进先出）结构。它们的行为有点像汽车闯红灯。新车在队列末端排队，最先离开的车将是第一个到达红灯的车。

在下面的代码示例中，我们让用户通过脚本的输入决定他们想要在图表上使用多少个标签。我们使用该数量来确定随后创建的标签数组的大小，并将数组的元素初始化为 `na`。

当检测到新的主元时，我们为其创建一个标签，并将标签的 ID 保存在变量中 `pLabel`。然后，我们使用 [array.push\(\)](#) 对该标签的 ID 进行排队，将新标签的 ID 附加到数组的末尾，使数组大小比图表上要保留的最大标签数大 1。

最后，我们通过使用 `array.shift()` 删除数组的第一个元素 并删除该数组元素的值引用的标签来使最旧的标签出队。由于我们现在已经从队列中出列了一个元素，因此数组 `pivotCountInput` 再次包含元素。请注意，在数据集的第一个条上，我们将删除 `na` 标签 ID，直到创建了最大数量的标签，但这不会导致运行时错误。让我们看看我们的代码：



```
//@version=5
MAX_LABELS = 100
indicator("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS)

pivotCountInput = input.int(5, "How many pivots to show", minval = 0, maxval =
MAX_LABELS)
pivotLegsInput = input.int(3, "Pivot legs", minval = 1, maxval = 5)

// Create an array containing the user-selected max count of label IDs.
var labelIds = array.new<label>(pivotCountInput)

pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput)
if not na(pHi)
    // New pivot found; plot its label `i_pivotLegs` bars back.
    pLabel = label.new(bar_index[pivotLegsInput], pHi, str.tostring(pHi,
format.mintick), textColor = color.white)
    // Queue the new label's ID by appending it to the end of the array.
    array.push(labelIds, pLabel)
    // De-queue the oldest label ID from the queue and delete the corresponding label.
    label.delete(array.shift(labelIds))
```

## 数组计算

系列变量可以被视为时间上延伸的一组水平值，而 Pine Script™ 的一维数组可以被视为驻留在每个条形上的垂直结构。由于数组的元素集不是时间序列，因此它们不允许使用 Pine Script™ 通常的数学函数。必须使用专用函数来操作数组的所有值。可用的函数有：[array.abs\(\)](#)、[array.avg\(\)](#)、[array.covariance\(\)](#)、[array.min\(\)](#)、[array.max\(\)](#)、[array.median\(\)](#)、[array.mode\(\)](#)、[array.percentile\\_linear\\_interpolation\(\)](#)、[array.percentile\\_nearest\\_rank\(\)](#)、[array.percentrank\(\)](#)、[array.range\(\)](#)、[array.standardize\(\)](#)、[array.stdev\(\)](#)、[array.sum\(\)](#)、[array.variance\(\)](#)。

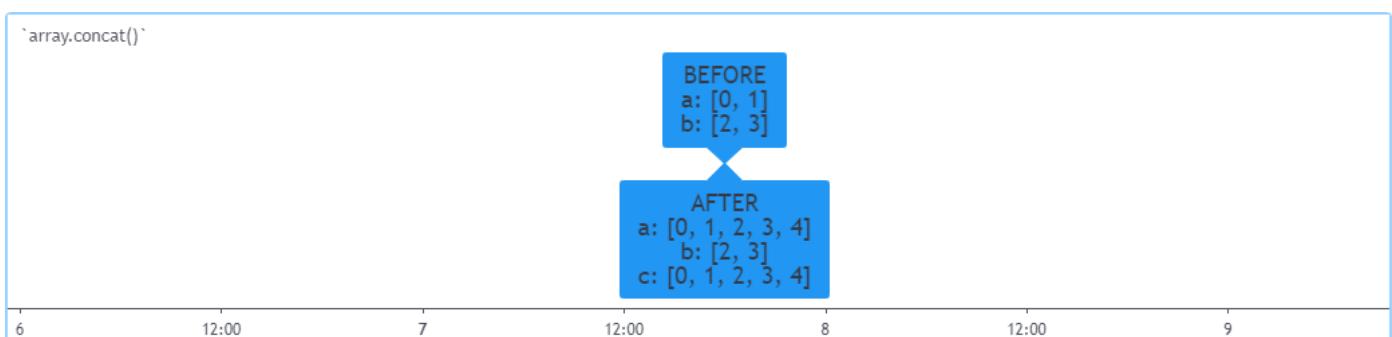
`na` 请注意，与 Pine Script™ 中常见的数学函数相反，当它们计算的某些值具有值时，用于数组的数学函数不会返回 `na`。此规则有一些例外：

- 当所有数组元素都有 `na` 值或数组不包含元素时，`na` 返回。`array.standardize()` 但是，将返回一个空数组。
- `array.mode()` 当没有找到模式时将返回 `na`。

## 操作数组

### 连接

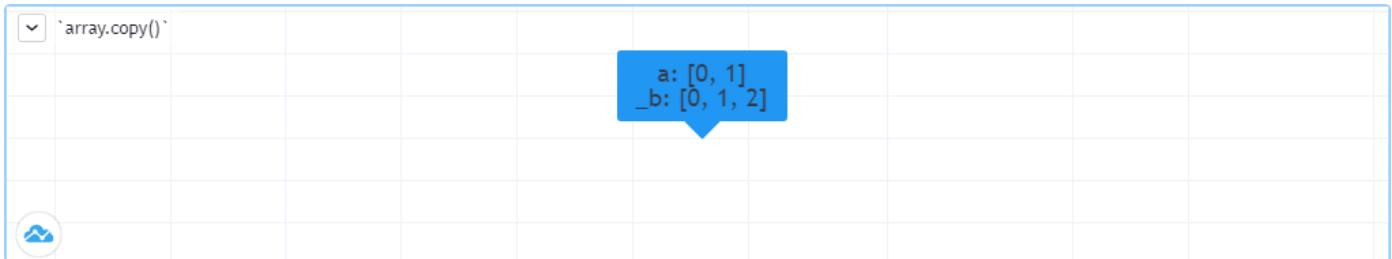
可以使用 `array.concat()` 合并或连接两个数组。当数组连接时，第二个数组将附加到第一个数组的末尾，因此第一个数组被修改，而第二个数组保持不变。该函数返回第一个数组的数组 ID：



```
//@version=5
indicator(`array.concat()`)
a = array.new<float>(0)
b = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(b, 2)
array.push(b, 3)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nb: " +
str.tostring(b), size = size.large)
    c = array.concat(a, b)
    array.push(c, 4)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nb: " + str.tostring(b) +
"\nc: " + str.tostring(c), style = label.style_label_up, size = size.large)
```

### 复制

您可以使用 `array.copy()` 复制数组。这里我们将数组复制 `a` 到一个名为 `_b` 的新数组：



```
//@version=5
indicator(`array.copy()`)
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
if barstate.islast
    b = array.copy(a)
    array.push(b, 2)
    label.new(bar_index, 0, "a: " + str.tostring(a) + "\nb: " + str.tostring(b), size =
size.large)
```

请注意，仅在前面的示例中使用不会复制数组，而只会复制其 ID。从那时起，两个变量将指向同一个数组，因此使用任一变量都会影响同一个数组。`_b = a`

## 添加

使用[array.join\(\)](#)将数组中的所有元素连接成一个字符串，并使用指定的分隔符分隔这些元素：

```
//@version=5
indicator("")
v1 = array.new<string>(10, "test")
v2 = array.new<string>(10, "test")
array.push(v2, "test1")
v3 = array.new_float(5, 5)
v4 = array.new_int(5, 5)
l1 = label.new(bar_index, close, array.join(v1))
l2 = label.new(bar_index, close, array.join(v2, ","))
l3 = label.new(bar_index, close, array.join(v3, ","))
l4 = label.new(bar_index, close, array.join(v4, ","))
```

## 排序

包含“int”或“float”元素的数组可以使用[array.sort\(\)](#)按升序或降序排序。该 `order` 参数是可选的，默认为[orderascending](#)。与所有 `array.*()` 函数参数一样，它被限定为“系列”，因此可以在运行时确定，就像这里所做的那样。请注意，在示例中，对哪个数组进行排序也是在运行时确定的：



```
//@version=5
indicator(`array.sort()`)

a = array.new<float>(0)
b = array.new<float>(0)
array.push(a, 2)
array.push(a, 0)
array.push(a, 1)
array.push(b, 4)
array.push(b, 3)
array.push(b, 5)
if barstate.islast
    barUp = close > open
    array.sort(barUp ? a : b, barUp ? order.ascending : order.descending)
    label.new(bar_index, 0,
        "a " + (barUp ? "is sorted ▲: " : "is not sorted: ") + str.tostring(a) + "\n\n"
    +
        "b " + (barUp ? "is not sorted: " : "is sorted ▼: ") + str.tostring(b), size =
    size.large)
```

对数组进行排序的另一个有用选项是使用 [array.sort\\_indices\(\)](#) 函数，该函数获取对原始数组的引用并返回包含原始数组索引的数组。请注意，此函数不会修改原始数组。该 `order` 参数是可选的，默认为 [order.ascending](#)。

## 反转

使用`array.reverse()`反转数组：

```
//@version=5
indicator(`array.reverse()`)
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
if barstate.islast
    array.reverse(a)
label.new(bar_index, 0, "a: " + str.tostring(a))
```

## 切片

使用`array.slice()`对数组进行切片 会创建父数组子集的浅表副本。您可以使用 `index_from` 和参数确定要切片的子集的大小 `index_to`。该 `index_to` 参数必须比要切片的子集末尾大 1。

由切片创建的浅表副本就像父数组内容上的窗口一样。用于切片的索引定义窗口在父数组上的位置和大小。如果如下例所示，从数组的前三个元素（索引 0 到 2）创建切片，则无论对父数组进行更改，只要它至少包含三个元素，浅层数组copy 将始终包含父数组的前三个元素。

此外，一旦创建了浅表副本，副本上的操作就会镜像到父数组上。将一个元素添加到浅表副本的末尾（如以下示例所示）会将窗口加宽一个元素，并将该元素插入到父数组中的索引 3 处。在此示例中，从索引 0 开始对子集进行切片对于数组的索引 2 `a`，我们必须使用：`_sliceOfA = array.slice(a, 0, 3)`



```
//@version=5
indicator(`array.slice()`)
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
array.push(a, 3)
if barstate.islast
    // Create a shadow of elements at index 1 and 2 from array `a`.
    sliceOfA = array.slice(a, 0, 3)
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nsliceOfA: " +
str.tostring(sliceOfA))
```

```

// Remove first element of parent array `a`.
array.remove(a, 0)
// Add a new element at the end of the shallow copy, thus also affecting the
original array `a`.
array.push(sliceOfA, 4)
label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nsliceOfA: " +
str.tostring(sliceOfA), style = label.style_label_up)

```

## 搜索数组

我们可以使用[array.includes\(\)](#)函数测试某个值是否是数组的一部分，如果找到该元素，该函数将返回 true。我们可以使用[array.indexOf\(\)](#)函数查找数组中某个值的第一次出现。第一次出现的是索引最低的那个。我们还可以使用[array.lastIndexOf\(\)](#)查找最后一次出现的值：

```

//@version=5
indicator("Searching in arrays")
valueInput = input.int(1)
a = array.new<float>(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
array.push(a, 1)
if barstate.islast
    valueFound      = array.includes(a, valueInput)
    firstIndexFound = array.indexOf(a, valueInput)
    lastIndexFound  = array.lastIndexOf(a, valueInput)
    label.new(bar_index, 0, "a: " + str.tostring(a) +
        "\nFirst " + str.tostring(valueInput) + (firstIndexFound != -1 ? " value was
        found at index: " + str.tostring(firstIndexFound) : " value was not found.") +
        "\nLast " + str.tostring(valueInput) + (lastIndexFound != -1 ? " value was
        found at index: " + str.tostring(lastIndexFound) : " value was not found."))

```

我们还可以对数组执行二分搜索，但请注意，对数组执行二分搜索意味着该数组首先需要仅按升序排序。如果找到该值，则array.binary\_search()[\(\)](#)函数将返回该值的索引；如果未找到，则返回 -1。如果我们希望始终从数组中返回现有索引，即使未找到我们选择的值，那么我们可以使用其他可用的二分搜索函数之一。

array.binary\_search\_leftmost()[\(\)](#)函数，如果找到该值则返回索引，或者返回找到该值的左侧第一个索引。

array.binary\_search\_rightmost()[\(\)](#)函数几乎相同，如果找到该值，则返回一个索引，或者返回找到该值的右侧的第一个索引。

## 错误处理

当您保存脚本时，Pine 脚本中格式错误的array.\*()[\(\)](#)调用语法将导致常见的编译器错误消息出现在 Pine 编辑器的控制台（位于窗口底部）中。如果对函数调用的确切语法有疑问，请参阅 Pine Script™ [v5 参考手册](#)。

使用数组的脚本还可能引发运行时错误，这些错误在图表上的指标名称旁边显示为感叹号。我们在本节中讨论这些运行时错误。

## 索引 xx 超出范围。数组大小为 yy

这很可能是您遇到的最常见的错误。当您引用不存在的数组索引时就会发生这种情况。“xx”值将是您尝试使用的错误索引的值，“yy”将是数组的大小。回想一下，数组索引从零开始，而不是从一开始，到数组大小减一结束。因此，大小为 3 的数组的最后一个有效索引是 2。

为了避免此错误，您必须在代码逻辑中做出规定，以防止使用位于数组索引边界之外的索引。此代码将生成错误，因为我们在循环中使用的最后一个索引超出了数组的有效索引范围：

```
//@version=5
indicator("Out of bounds index")
a = array.new<float>(3)
for i = 1 to 3
    array.set(a, i, i)
plot(array.pop(a))
```

正确的 for 说法是：

```
for i = 0 to 2
```

要循环未知大小数组中的所有数组元素，请使用：

```
//@version=5
indicator("Protected `for` loop")
sizeInput = input.int(0, "Array size", minval = 0, maxval = 100000)
a = array.new<float>(sizeInput)
for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
    array.set(a, i, i)
plot(array.pop(a))
```

当您使用脚本的“设置/输入”选项卡中的字段动态调整数组大小时，请使用 `input.int(minval)` 和参数保护该值的边界 `maxval`：

```
//@version=5
indicator("Protected array size")
sizeInput = input.int(10, "Array size", minval = 1, maxval = 100000)
a = array.new<float>(sizeInput)
for i = 0 to sizeInput - 1
    array.set(a, i, i)
plot(array.size(a))
```

有关详细信息，请参阅本页的 [循环部分](#)。

## 当数组 ID 为 'na' 时无法调用数组方法

当数组 ID 初始化为 `na` 时，不允许对其进行操作，因为不存在数组。此时存在的只是一个包含该 `na` 值的数组变量，而不是指向现有数组的有效数组 ID。请注意，创建的数组中没有任何元素（就像使用 `na` 时所做的那样），但仍然具有有效的 ID。这段代码将抛出我们正在讨论的错误：`a = array.new_int(0)`

```
//@version=5
indicator("Out of bounds index")
array<int> a = na
array.push(a, 111)
label.new(bar_index, 0, "a: " + str.tostring(a))
```

为了避免这种情况，请使用以下命令创建一个大小为零的数组：

```
array<int> a = array.new_int(0)
```

或者：

```
a = array.new_int(0)
```

## 数组太大。最大大小为 100000

如果您的代码尝试声明大小大于 100,000 的数组，则会出现此错误。如果在动态追加元素到数组时，新元素将数组的大小增加到超过最大值，也会发生这种情况。

## 无法创建负数数组

我们还没有发现负大小数组的任何用途，但如果您发现了，我们可能会允许它们：)

## 如果数组为空，则无法使用 `shift()`。

如果调用 `array.shift()` 来删除空数组的第一个元素，则会发生此错误。

## 如果数组为空，则无法使用 `pop()`。

如果调用 `array.pop()` 来删除空数组的最后一个元素，则会发生此错误。

## 索引“from”应该小于索引“to”

当在array.slice()等函数中使用两个索引时，第一个索引必须始终小于第二个索引。

## 切片超出父数组的范围

每当父数组的大小被修改而导致由切片点创建的浅表副本位于父数组边界之外时，就会出现此消息。此代码将重现它，因为在创建从索引 3 到 4（我们的五元素父数组的最后两个元素）的切片后，我们删除父元素的第一个元素，使其大小为 4，最后一个索引为 3。从那一刻起，仍然指向父数组索引 3 到 4 处的“窗口”的浅拷贝，指向父数组的边界之外：

```
//@version=5
indicator("Slice out of bounds")
a = array.new<float>(5, 0)
b = array.slice(a, 3, 5)
array.remove(a, 0)
c = array.indexof(b, 2)
plot(c)
```

# 矩阵

提示：

此页面包含高级材料。如果您是一名初级 Pine Script™ 程序员，我们建议您在冒险之前熟悉其他更易于使用的 Pine Script™ 功能。

## 介绍

Pine Script™ 矩阵是以矩形格式存储值引用的集合。它们本质上相当于二维数组 [对象](#)，具有用于检查、修改和专门计算的函数和方法。与 [数组](#)一样，所有矩阵元素必须具有相同[类型](#)，可以是 [内置类型](#)或 [用户定义类型](#)。

矩阵使用两个索引引用其元素：一个索引用于行，另一个索引用于列。每个索引从 0 开始，并扩展到矩阵中的行/列数减一。Pine 中的矩阵可以具有随条形变化的动态行数和列数。矩阵内的[元素总数](#)是[行数](#)和[列数](#)的乘积（例如，5x5 矩阵共有 25 个元素）。与 [数组](#)一样，矩阵中的元素总数不能超过 100,000。

## 声明一个矩阵

Pine Script™ 使用以下语法进行矩阵声明：

```
[var/varip ][matrix<type> ]<identifier> = <expression>
```

其中 `<type>` 是矩阵的[类型模板](#)，声明它将包含的值的类型，并且 `<expression>` 返回类型或 的矩阵实例 `na`。

将矩阵变量声明为 时，用户必须通过在[类型模板](#)后包含[矩阵](#) `na` 关键字来指定标识符将引用特定类型的矩阵。

该行声明了一个 `myMatrix` 值为 的新变量 `na`。它将变量显式声明为 `matrix<float>`，这告诉编译器该变量只能接受 包含浮点值的矩阵 对象：

```
matrix<float> myMatrix = na
```

当矩阵变量未分配给 时 `na`，矩阵 关键字及其类型模板是可选的，因为编译器将使用变量引用的对象中的类型信息。

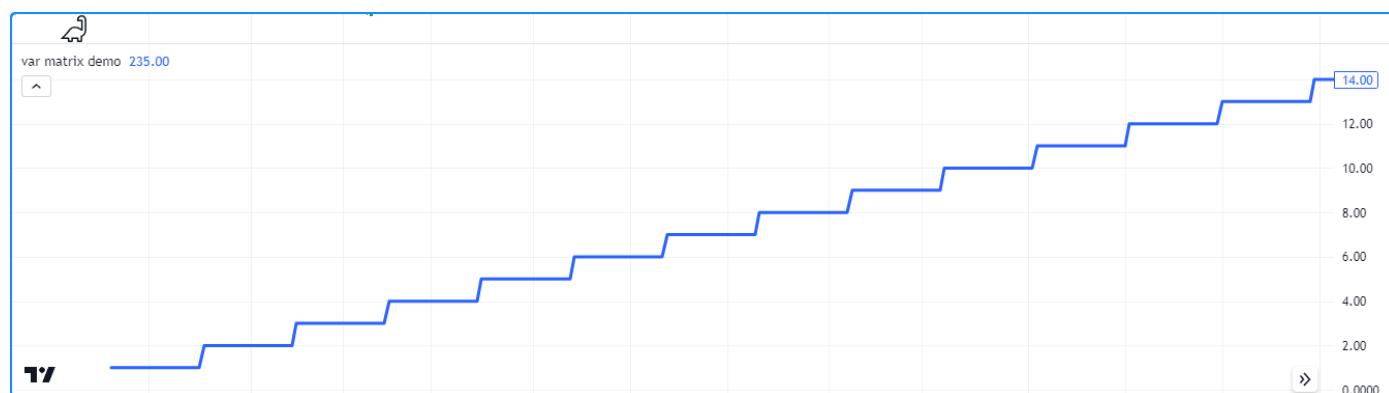
在这里，我们声明一个 `myMatrix` 变量引用一个 `matrix<float>` 具有两行、两列和 `initial_value 0` 的新实例。在这种情况下，该变量从新对象获取其类型信息，因此不需要显式类型声明：

```
myMatrix = matrix.new<float>(2, 2, 0.0)
```

## 使用 `var` 和 `varip` 关键字

与其他变量一样，用户可以包含 `var` 或 `varip` 关键字来指示脚本仅声明一次矩阵变量，而不是在每个柱上声明一次。使用此关键字声明的矩阵变量将在整个图表范围内指向同一实例，除非脚本显式为其分配另一个矩阵，从而允许矩阵及其元素引用在脚本迭代之间保留。

此脚本 使用 `var m` 关键字声明一个分配给矩阵的变量，该矩阵保存单行两个 `int` 元素。在每 20 个柱上，脚本将 1 添加到矩阵第一行的第一个元素。 `plot()` 调用在图表上显示该元素。正如我们从图中看到的，`m.get(0, 0)` 的值在柱之间持续存在，永远不会返回到初始值 0： `m`



```
//@version=5
indicator("var matrix demo")

//@variable A 1x2 rectangular matrix declared only at `bar_index == 0` , i.e., the first bar.
var m = matrix.new<int>(1, 2, 0)

//@variable Is `true` on every 20th bar.
bool update = bar_index % 20 == 0

if update
    int currentValue = m.get(0, 0) // Get the current value of the first row and column.
```

```

m.set(0, 0, currentValue + 1) // Set the first row and column element value to
`currentValue + 1`.

plot(m.get(0, 0), linewidth = 3) // Plot the value from the first row and column.

```

## 笔记

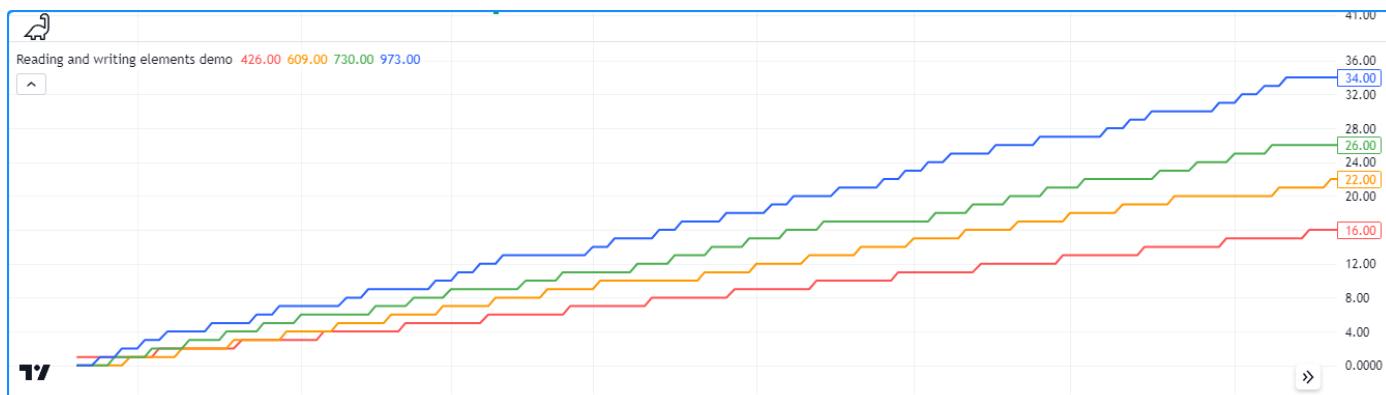
使用`varip`声明的矩阵变量的行为与在历史数据上使用`var`的 行为相同，但它们会在每个新的价格变动时更新实时柱（即自脚本上次编译以来的柱）的值。分配给`varip` 变量的矩阵只能保存`int`、`float`、`bool`、`color`或`string`类型或 [用户定义类型](#)，这些类型在其字段中专门包含这些类型或这些类型的集合（`arrays`、`matrices` 或`maps`）。

## 读写矩阵元素

### `matrix.get()` 和 `matrix.set()`

要从指定索引处的矩阵中检索值 `column`，请使用 `matrix.get()`。该函数查找指定的矩阵元素并返回其值。类似地，要覆盖特定元素的值，请使用 `matrix.set()` 将指定的元素分配 `row` 给 `column` 新的 `value`。

下面的示例定义了一个 `m` 具有两行和两列的方阵，并且 `initial_value` 第一个条上的所有元素的值为 0。该脚本使用`m.get()` 和`m.set()`方法将不同柱上每个元素的值加 1。它每 11 个柱更新一次第一行的第一个值，每 7 个柱更新一次第一行的第二个值，每 5 个柱更新一次第二行的第一个值，每 3 个柱更新一次第二行的第二个值。该脚本在图表上绘制每个元素的值：



```

//@version=5
indicator("Reading and writing elements demo")

//@variable A 2x2 square matrix of `float` values.
var m = matrix.new<float>(2, 2, 0.0)

switch
    bar_index % 11 == 0 => m.set(0, 0, m.get(0, 0) + 1.0) // Adds 1 to the value at row
0, column 0 every 11th bar.
    bar_index % 7 == 0 => m.set(0, 1, m.get(0, 1) + 1.0) // Adds 1 to the value at row
0, column 1 every 7th bar.

```

```

bar_index % 5 == 0 => m.set(1, 0, m.get(1, 0) + 1.0) // Adds 1 to the value at row
1, column 0 every 5th bar.
bar_index % 3 == 0 => m.set(1, 1, m.get(1, 1) + 1.0) // Adds 1 to the value at row
1, column 1 every 3rd bar.

plot(m.get(0, 0), "Row 0, Column 0 Value", color.red, 2)
plot(m.get(0, 1), "Row 0, Column 1 Value", color.orange, 2)
plot(m.get(1, 0), "Row 1, Column 0 Value", color.green, 2)
plot(m.get(1, 1), "Row 1, Column 1 Value", color.blue, 2)

```

## [matrix.fill\(\)](#)

要使用特定值覆盖所有矩阵元素，请使用 [matrix.fill\(\)](#)。该函数将整个矩阵中或索引范围内的所有项目 `from_row/column` 指向 `to_row/column` 调用 `value` 中指定的。例如，此代码片段声明一个 4x4 方阵，然后用 [随机](#) 值填充其元素：

```

myMatrix = matrix.new<float>(4, 4)
myMatrix.fill(math.random())

```

请注意，当对包含特殊类型 ([line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#) 或 [chart.point](#)) 或 [UDT](#) 的矩阵使用 [matrix.fill\(\)](#) 时，所有替换元素将指向函数调用中传递的同一对象。

此脚本声明一个包含四行四列 [标签](#) 引用的矩阵，并在第一个栏上 填充新的[标签](#) 对象。在每个柱上，脚本将 `x` 第 0 行第 0 列引用的标签属性设置为 `bar_index`，并将 `text` 第 3 行第 3 列引用的标签属性设置为图表上的标签数量。尽管矩阵可以引用 16 (4x4) 个标签，但每个元素都指向 [同一个实例](#)，导致图表上只有一个标签更新每个条形上的 `x` 和属性：`text`



```

//@version=5
indicator("Object matrix fill demo")

//@variable A 4x4 label matrix.
var matrix<label> m = matrix.new<label>(4, 4)

// Fill `m` with a new label object on the first bar.

```

```

if bar_index == 0
    m.fill(label.new(0, 0, textcolor = color.white, size = size.huge))

//@variable The number of label objects on the chart.
int numLabels = label.all.size()

// Set the `x` of the label from the first row and column to `bar_index`.
m.get(0, 0).set_x(bar_index)
// Set the `text` of the label at the last row and column to the number of labels.
m.get(3, 3).set_text(str.format("Total labels on the chart: {0}", numLabels))

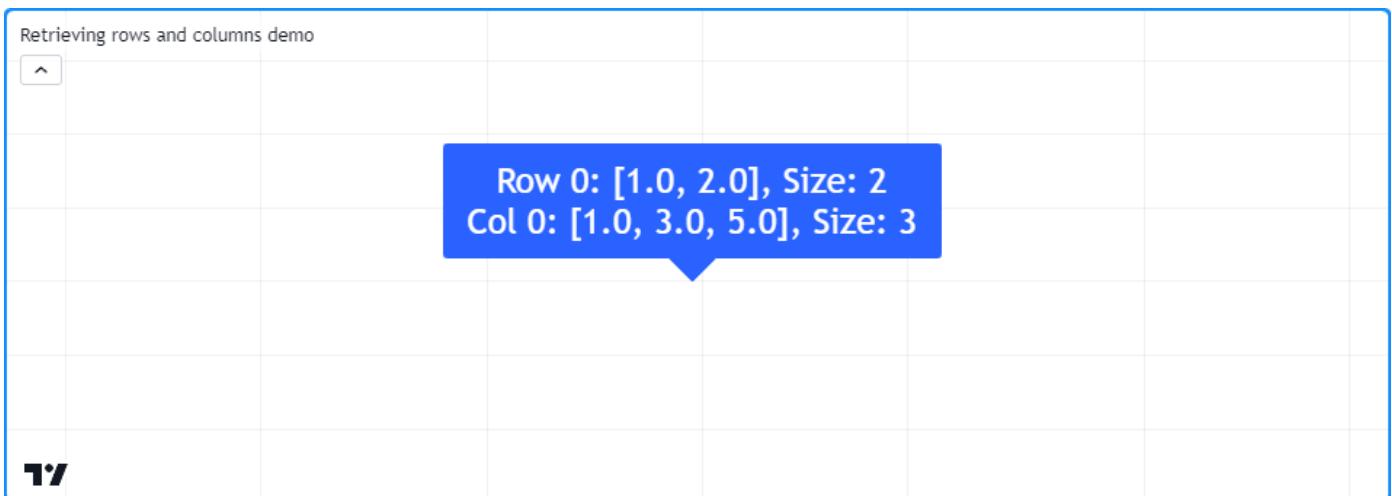
```

## 行和列

### 检索

矩阵有助于通过 `matrix.row()` 和 `matrix.col()` 函数检索特定行或列中的所有值。这些函数将值作为 [数组对象](#) 返回，该数组对象的大小根据矩阵的其他维度进行调整，即，`matrix.row()` 数组的大小等于 [列数](#)，`matrix.col()` 数组的大小等于 [列数行数](#)。

下面的脚本使用 `m` 第一个图表栏上的值 1 - 6 填充  $3 \times 2$  矩阵。它调用 `m.row()` 和 `m.col()` 方法来访问矩阵中的第一行和第一列数组，并将它们与数组大小一起显示在图表上的标签中：



```

//@version=5
indicator("Retrieving rows and columns demo")

//@variable A 3x2 rectangular matrix.
var matrix<float> m = matrix.new<float>(3, 2)

if bar_index == 0
    m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
    m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.

```

```

m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.
m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.

//@variable The first row of the matrix.
array<float> row0 = m.row(0)
//@variable The first column of the matrix.
array<float> column0 = m.col(0)

//@variable Displays the first row and column of the matrix and their sizes in a label.
var label debugLabel = label.new(0, 0, color = color.blue, textcolor = color.white,
size = size.huge)
debugLabel.set_x(bar_index)
debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0,
m.columns(), column0, m.rows())))

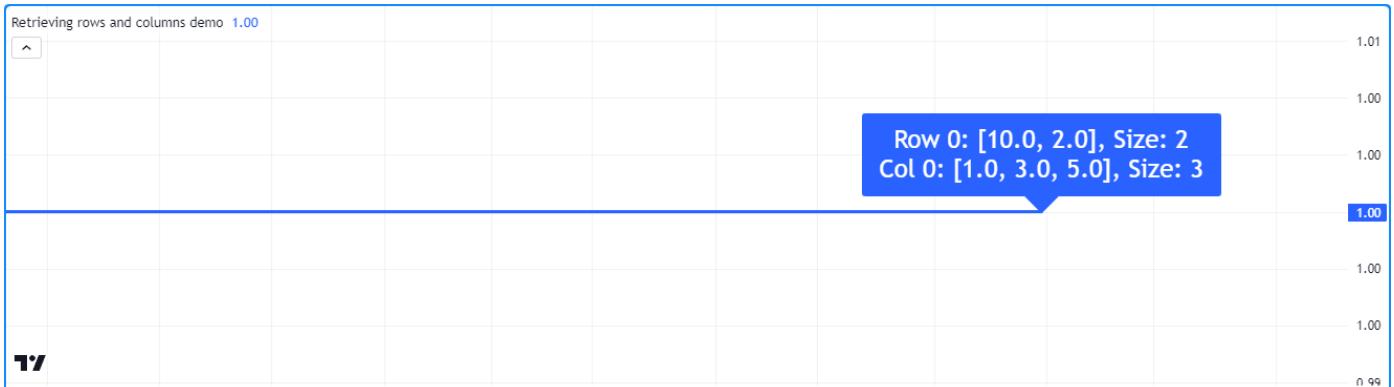
```

- 注意：

为了获取标签中显示的数组的大小，我们使用 [rows\(\)](#) 和 [columns\(\)](#) 方法而不是 [array.size\(\)](#) 来演示 `row0` 数组的大小等于列数，并且数组的大小 `column0` 等于行数。

[matrix.row\(\)](#) 和 [matrix.col\(\)](#) 将行/列中的引用复制到新的数组中。对这些函数返回的数组的修改不会直接影响矩阵的元素或形状。

在这里，我们修改了之前的脚本，在显示标签之前通过 [array.set\(\)](#) 方法将第一个元素设置 `row0` 为 10。该脚本还绘制了第 0 行、第 0 列的值。正如我们所见，标签显示数组的第一个元素是 10。但是，该图显示相应的矩阵元素的值仍然为 1： `row0`



```

//@version=5
indicator("Retrieving rows and columns demo")

//@variable A 3x2 rectangular matrix.
var matrix<float> m = matrix.new<float>(3, 2)

if bar_index == 0
    m.set(0, 0, 1.0) // Set row 0, column 0 value to 1.
    m.set(0, 1, 2.0) // Set row 0, column 1 value to 2.
    m.set(1, 0, 3.0) // Set row 1, column 0 value to 3.
    m.set(1, 1, 4.0) // Set row 1, column 1 value to 4.
    m.set(2, 0, 5.0) // Set row 1, column 0 value to 5.

```

```

m.set(2, 1, 6.0) // Set row 1, column 1 value to 6.

//@variable The first row of the matrix.
array<float> row0 = m.row(0)
//@variable The first column of the matrix.
array<float> column0 = m.col(0)

// Set the first `row` element to 10.
row0.set(0, 10)

//@variable Displays the first row and column of the matrix and their sizes in a label.
var label debugLabel = label.new(0, m.get(0, 0), color = color.blue, textcolor =
color.white, size = size.huge)
debugLabel.set_x(bar_index)
debugLabel.set_text(str.format("Row 0: {0}, Size: {1}\nCol 0: {2}, Size: {3}", row0,
m.columns(), column0, m.rows()))

// Plot the first element of `m`.
plot(m.get(0, 0), linewidth = 3)

```

虽然对 [array.row\(\)](#) 或 [matrix.col\(\)](#) 返回的数组进行更改不会直接影响父矩阵，但请务必注意包含 [UDT](#) 或特殊类型（包括 [line](#)、[linefill](#)、[box](#)、[polyline](#)）的矩阵所生成的数组、[label](#)、[table](#) 或 [Chart.point](#)，表现为行/列的浅表副本，即从这些函数返回的数组中的元素指向与相应矩阵元素相同的对象。

此脚本包含一个自定义 `myUDT` 类型，其中包含 `value` 一个初始值为 0 的字段。它声明一个  $1 \times 1$  矩阵来保存 `myUDT` 第一个柱上的单个实例，然后调用 `m.row(0)` 将矩阵的第一行复制为 [array](#)。在每个图表条上，脚本将 1 添加到 `value` 第一个 `row` 数组元素的字段。在这种情况下，`value` 矩阵元素的字段也会在每个条上增加，因为两个元素引用相同的对象：

```

//@version=5
indicator("Row with reference types demo")

//@type A custom type that holds a float value.
type myUDT
    float value = 0.0

//@variable A 1x1 matrix of `myUDT` type.
var matrix<myUDT> m = matrix.new<myUDT>(1, 1, myUDT.new())
//@variable A shallow copy of the first row of `m`.
array<myUDT> row = m.row(0)
//@variable The first element of the `row`.
myUDT firstElement = row.get(0)

firstElement.value += 1.0 // Add 1 to the `value` field of `firstElement`. Also affects
the element in the matrix.

plot(m.get(0, 0).value, linewidth = 3) // Plot the `value` of the `myUDT` object from
the first row and column of `m`.

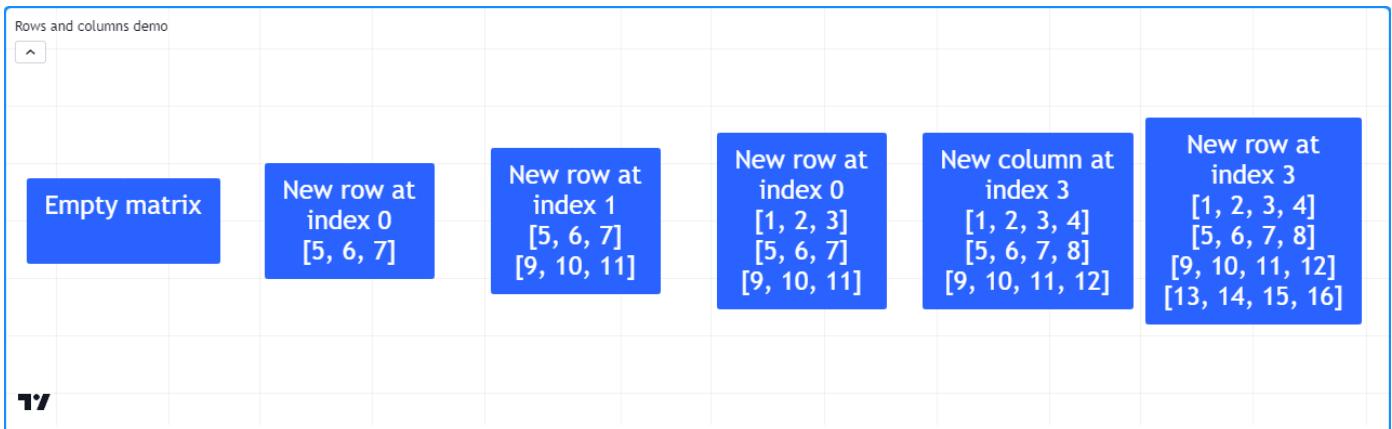
```

## 插入

脚本可以通过`matrix.add_row()`和`matrix.add_col()`向矩阵添加新的行和列。这些函数将数组中的值引用插入到矩阵的指定索引处`row/column`。如果`id`矩阵为空（没有行或列），则`array_id`调用中的可以是任意大小。如果指定索引处存在行/列，则矩阵会将现有行/列及其后所有行/列的索引值加1。

下面的脚本声明一个空矩阵并使用`m.add_row()`和`m.add_col()`方法插入行和列。它首先在第0行插入一个包含三个元素的数组，将其变为 $1 \times 3$ 矩阵，然后在第1行插入另一个数组，将形状更改为 $2 \times 3$ 。之后，脚本在第0行插入另一个数组，这会将形状更改为 $3 \times 3$ ，并将之前所有行的索引移动到索引0及更高位置。它在最后一列索引处插入另一个数组，将形状更改为 $3 \times 4$ 。最后，它在末尾行索引处添加一个包含四个值的数组。`m^~m`

生成的矩阵有四行四列，包含按升序排列的值1-16。该脚本`m`使用用户定义的函数显示每行/列插入后的行`debugLabel()`以可视化该过程：



```
//@version=5
indicator("Rows and columns demo")

//@function Displays the rows of a matrix in a label with a note.
//@param this The matrix to display.
//@param barIndex The `bar_index` to display the label at.
//@param bgColor The background color of the label.
//@param textColor The color of the label's text.
//@param note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//Create an empty matrix.
var m = matrix.new<float>()
```

```

if bar_index == last_bar_index - 1
    debugLabel(m, bar_index - 30, note = "Empty matrix")

    // Insert an array at row 0. `m` will now have 1 row and 3 columns.
    m.add_row(0, array.from(5, 6, 7))
    debugLabel(m, bar_index - 20, note = "New row at\nindex 0")

    // Insert an array at row 1. `m` will now have 2 rows and 3 columns.
    m.add_row(1, array.from(9, 10, 11))
    debugLabel(m, bar_index - 10, note = "New row at\nindex 1")

    // Insert another array at row 0. `m` will now have 3 rows and 3 columns.
    // The values previously on row 0 will now be on row 1, and the values from row 1
    will be on row 2.
    m.add_row(0, array.from(1, 2, 3))
    debugLabel(m, bar_index, note = "New row at\nindex 0")

    // Insert an array at column 3. `m` will now have 3 rows and 4 columns.
    m.add_col(3, array.from(4, 8, 12))
    debugLabel(m, bar_index + 10, note = "New column at\nindex 3")

    // Insert an array at row 3. `m` will now have 4 rows and 4 columns.
    m.add_row(3, array.from(13, 14, 15, 16))
    debugLabel(m, bar_index + 20, note = "New row at\nindex 3")

```

## 笔记

正如从 [line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)、[Chart.point](#) 或 [UDT](#) 实例的矩阵中检索的行或列数组表现  
为浅拷贝一样，包含此类类型的矩阵元素引用与数组相同的对象 插入其中。在这种情况下，对任一对象中元素值的  
修改都会影响另一个对象。

## 删除

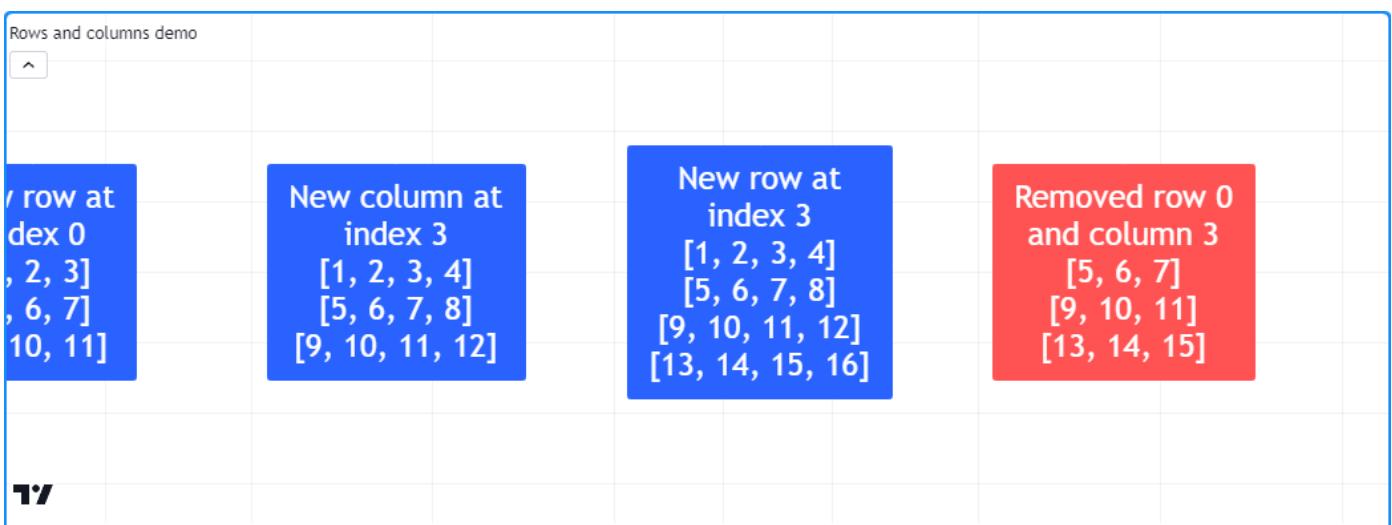
要从矩阵中删除特定的行或列，请使用 [matrix.remove\\_row\(\)](#) 和 [matrix.remove\\_col\(\)](#)。这些函数删除指定的行/  
列，并将其之后的所有行/列的索引值减 1。

对于此示例，我们将这些代码行添加到 [上面部分](#) 中的“行和列演示”脚本中：

```
// Removing example

// Remove the first row and last column from the matrix. `m` will now have 3 rows
and 3 columns.

m.remove_row(0)
m.remove_col(3)
debugLabel(m, bar_index + 30, color.red, note = "Removed row 0\nand column 3")
m`此代码使用 [m.remove_row()] (https://www.tradingview.com/pine-script-reference/v5/#fun\_matrix.remove\_row) 和 [m.remove_col()] (https://www.tradingview.com/pine-script-reference/v5/#fun\_matrix.remove\_col) 方法删除矩阵的第一行和最后一列，并在 处的标签中显示行。正如我们所看到的，执行此块后形状为 3x3，并且所有现有行的索引值均减 1: `bar_index + 30` `m
```



## 交换

要交换矩阵的行和列而不改变其维度，请使用 [matrix.swap\\_rows\(\)](#) 和[matrix.swap\\_columns\(\)](#)。这些函数交换 `row1/column1` 和索引处元素的位置 `row2/column2`。

让我们将以下行添加到[前面的示例](#)中，这些行交换 的第一行和最后一行 `m` 并在标签中显示更改：`bar_index + 40`

```
// Swapping example

// Swap the first and last row. `m` retains the same dimensions.
m.swap_rows(0, 2)
debugLabel(m, bar_index + 40, color.purple, note = "Swapped rows 0\nand 2")
```

在新标签中，我们看到矩阵的行数与之前相同，并且第一行和最后一行交换了位置：

## Rows and columns demo

row at  
ex 0  
2, 3]  
6, 7]  
0, 11]

New column at  
index 3  
[1, 2, 3, 4]  
[5, 6, 7, 8]  
[9, 10, 11, 12]

New row at  
index 3  
[1, 2, 3, 4]  
[5, 6, 7, 8]  
[9, 10, 11, 12]  
[13, 14, 15, 16]

Removed row 0  
and column 3  
[5, 6, 7]  
[9, 10, 11]  
[13, 14, 15]

Swapped rows 0  
and 2  
[13, 14, 15]  
[9, 10, 11]  
[5, 6, 7]

17

## 替换

在某些情况下，可能需要完全替换矩阵中的行或列。为此，请在所需位置[插入新数组 row/column](#)，并[删除该索引处先前的旧元素](#)。

在下面的代码中，我们定义了一个 `replaceRow()` 方法，该方法使用 `add_row(values)` 方法在索引处插入新行 `row`，并使用 `remove_row()` 方法删除移动到索引的旧行。此脚本使用该方法用数字 1-9 填充 3x3 矩阵的行。它在使用自定义方法替换行之前和之后在图表上绘制标签：`row + 1``replaceRow()``debugLabel()`

## Replacing rows demo

17

Original  
[0, 0, 0]  
[0, 0, 0]  
[0, 0, 0]

Replaced rows  
[1, 2, 3]  
[4, 5, 6]  
[7, 8, 9]

17

```
//@version=5
indicator("Replacing rows demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
```

```

matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
color textColor = color.white, string note = ""
) =>
labelText = note + "\n" + str.tostring(this)
if barstate.ishistory
    label.new(
        barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
        textcolor = textColor, size = size.huge
    )

//@function Replaces the `row` of `this` matrix with a new array of `values`.
//@param    row The row index to replace.
//@param    values The array of values to insert.
method replaceRow(matrix<float> this, int row, array<float> values) =>
    this.add_row(row, values) // Inserts a copy of the `values` array at the `row`.
    this.remove_row(row + 1) // Removes the old elements previously at the `row`.

//@variable A 3x3 matrix.
var matrix<float> m = matrix.new<float>(3, 3, 0.0)

if bar_index == last_bar_index - 1
    m.debugLine(note = "Original")
    // Replace each row of `m`.
    m.replaceRow(0, array.from(1.0, 2.0, 3.0))
    m.replaceRow(1, array.from(4.0, 5.0, 6.0))
    m.replaceRow(2, array.from(7.0, 8.0, 9.0))
    m.debugLine(bar_index + 10, note = "Replaced rows")

```

## 循环矩阵

### for

当脚本只需要迭代矩阵中的行/列索引时，最常见的方法是使用 `for` 循环。例如，此行创建一个循环，`row` 其值从 0 开始并增加 1，直达到到比矩阵中的行数 `m`（即最后一行索引）少 1 为止：

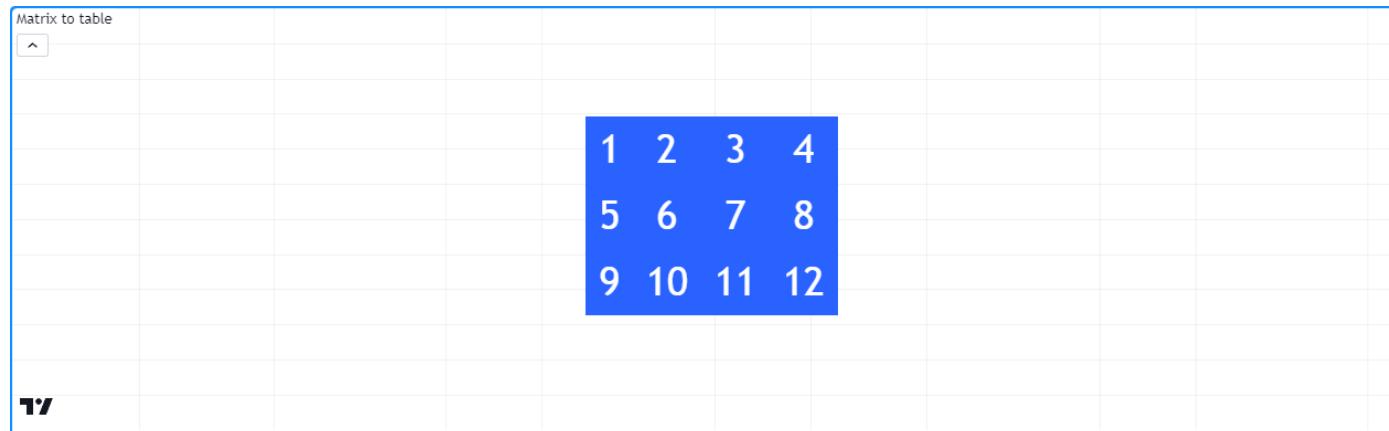
```
for row = 0 to m.rows() - 1
```

要迭代 `m` 矩阵中的所有索引值，我们可以创建一个嵌套循环，迭代每个值 `column` 的每个索引 `row`：

```
for row = 0 to m.rows() - 1
    for column = 0 to m.columns() - 1
```

让我们使用这个嵌套结构来创建一个可视化矩阵元素的方法。在下面的脚本中，我们定义了一个在表`toTable()`对象中显示矩阵元素的方法。它迭代每个索引以及每个上的每个索引。在循环内，它将每个元素转换为字符串以显示在相应的表格单元格中。`row``column``row`

在第一个栏上，脚本创建一个空`m`矩阵，用行填充它，并调用`m.toTable()`以显示其元素：



```
//@version=5
indicator("for loop demo", "Matrix to table")

//@function Displays the elements of `this` matrix in a table.
//@param    this The matrix to display.
//@param    position The position of the table on the chart.
//@param    bgColor The background color of the table.
//@param    textColor The color of the text in each cell.
//@param    note A note string to display on the bottom row of the table.
//@returns  A new `table` object with cells corresponding to each element of `this` matrix.

method toTable(
    matrix<float> this, string position = position.middle_center,
    color bgColor = color.blue, color textColor = color.white,
    string note = na
) =>
    // @variable The number of rows in `this` matrix.
    int rows = this.rows()
    // @variable The number of columns in `this` matrix.
    int columns = this.columns()
    // @variable A table that displays the elements of `this` matrix with an optional `note` cell.
    table result = table.new(position, columns, rows + 1, bgColor)

    // Iterate over each row index of `this` matrix.
    for row = 0 to rows - 1
        // Iterate over each column index of `this` matrix on each `row`.
        for col = 0 to columns - 1
            // @variable The element from `this` matrix at the `row` and `col` index.
            float element = this.get(row, col)
            // Initialize the corresponding `result` cell with the `element` value.
```

```

        result.cell(col, row, str.tostring(element), text_color = textColor,
text_size = size.huge)

    // Initialize a merged cell on the bottom row if a `note` is provided.
    if not na(note)
        result.cell(0, rows, note, text_color = textColor, text_size = size.huge)
        result.merge_cells(0, rows, columns - 1, rows)

    result // Return the `result` table.

//@variable A 3x4 matrix of values.
var m = matrix.new<float>()

if bar_index == 0
    // Add rows to `m`.
    m.add_row(0, array.from(1, 2, 3))
    m.add_row(1, array.from(5, 6, 7))
    m.add_row(2, array.from(9, 10, 11))
    // Add a column to `m`.
    m.add_col(3, array.from(4, 8, 12))
    // Display the elements of `m` in a table.
    m.toTable()

```

## for...in

当脚本需要迭代并检索矩阵的行时，使用 `for...in` 结构通常优于标准 `for` 循环。此结构直接引用矩阵中的行[数组](#)，使其成为此类用例的更方便的选择。例如，此行创建一个循环，`row` 为矩阵中的每一行返回一个数组 `m`：

```
for row in m
```

以下指标通过输入计算 OHLC 数据的移动平均值，`length` 并将值显示在图表上。自定义 `rowWiseAvg()` 方法使用结构体循环遍历矩阵的行，以生成包含每个的 `array.avg()` 的 `for...in` 数组。`row`

在第一个图表条上，脚本创建一个 `m` 具有四行和四列的新矩阵，通过每个后续条上的 `m.add_col()` 和 `m.remove_col()` `length` 方法将新的 OHLC 数据列排队到该矩阵中。它用于计算 row-wise 数组，然后在图表上绘制元素值：`m.rowWiseAvg()``averages`



```

//@version=5
indicator("for...in loop demo", "Average OHLC", overlay = true)

//@variable The number of terms in the average.
int length = input.int(20, "Length", minval = 1)

//@function Calculates the average of each matrix row.
method matrix<float> rowWiseAvg(matrix<float> this) =>
    //@variable An array with elements corresponding to each row's average.
    array<float> result = array.new<float>()
    // Iterate over each `row` of `this` matrix.
    for row in this
        // Push the average of each `row` into the `result`.
        result.push(row.avg())
    result // Return the resulting array.

//@variable A 4x`length` matrix of values.
var matrix<float> m = matrix.new<float>(4, length)

// Add a new column containing OHLC values to the matrix.
m.add_col(m.columns(), array.from(open, high, low, close))
// Remove the first column.
m.remove_col(0)

//@variable An array containing averages of `open`, `high`, `low`, and `close` over
`length` bars.
array<float> averages = m.rowWiseAvg()

plot(averages.get(0), "Average Open", color.blue, 2)
plot(averages.get(1), "Average High", color.green, 2)
plot(averages.get(2), "Average Low", color.red, 2)

```

```
plot(averages.get(3), "Average Close", color.orange, 2)
```

- 注意：

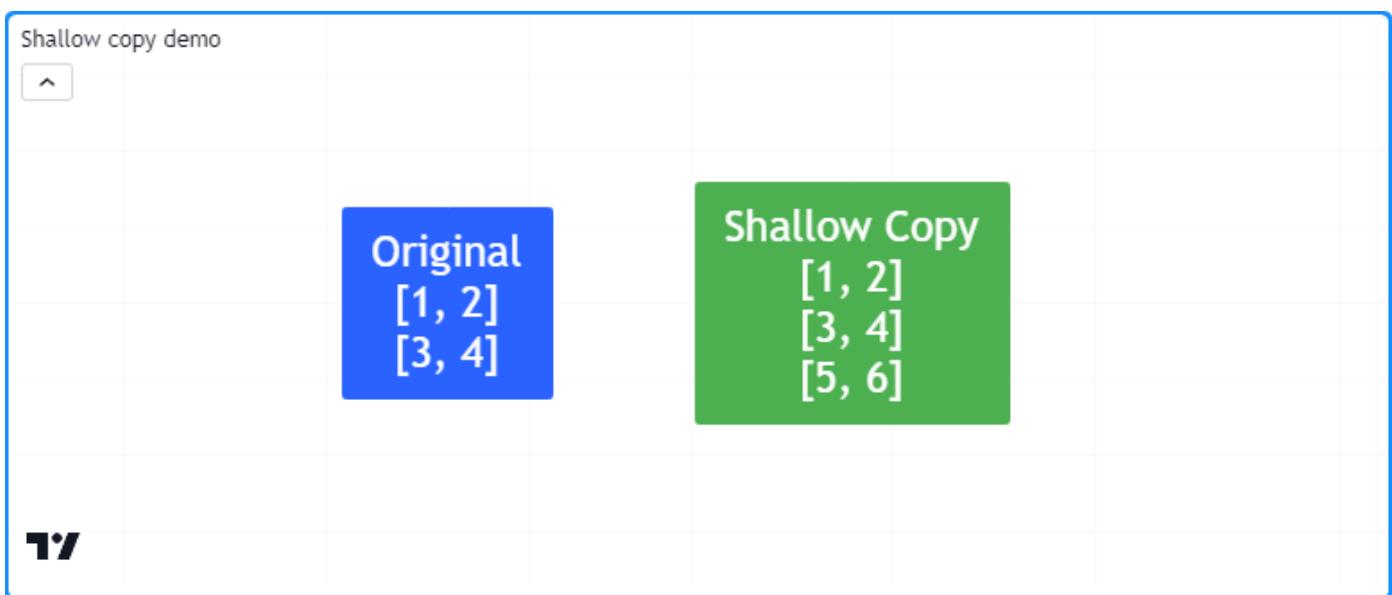
`for...in` 循环还可以引用每行的索引值。例如，在每次循环迭代时创建一个包含行索引和矩阵中相应数组的元组。`for [i, row] in m``i``row``m`

## 复制矩阵

### 浅拷贝

Pine脚本可以通过`matrix.copy()`复制矩阵。此函数返回矩阵的浅表副本，该副本不会影响原始矩阵或其引用的形状。

例如，此脚本将一个新矩阵分配给`myMatrix`变量并添加两列。它使用`myMatrix.copy()`方法创建一个新`myCopy`矩阵，然后添加一个新行。它通过用户定义的函数在标签中显示两个矩阵的行：`myMatrix``debugLabel()`



```
//@version=5
indicator("Shallow copy demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
```

```

if barstate.ishistory
    label.new(
        barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
        textColor = textColor, size = size.huge
    )

//@variable A 2x2 `float` matrix.
matrix<float> myMatrix = matrix.new<float>()
myMatrix.add_col(0, array.from(1.0, 3.0))
myMatrix.add_col(1, array.from(2.0, 4.0))

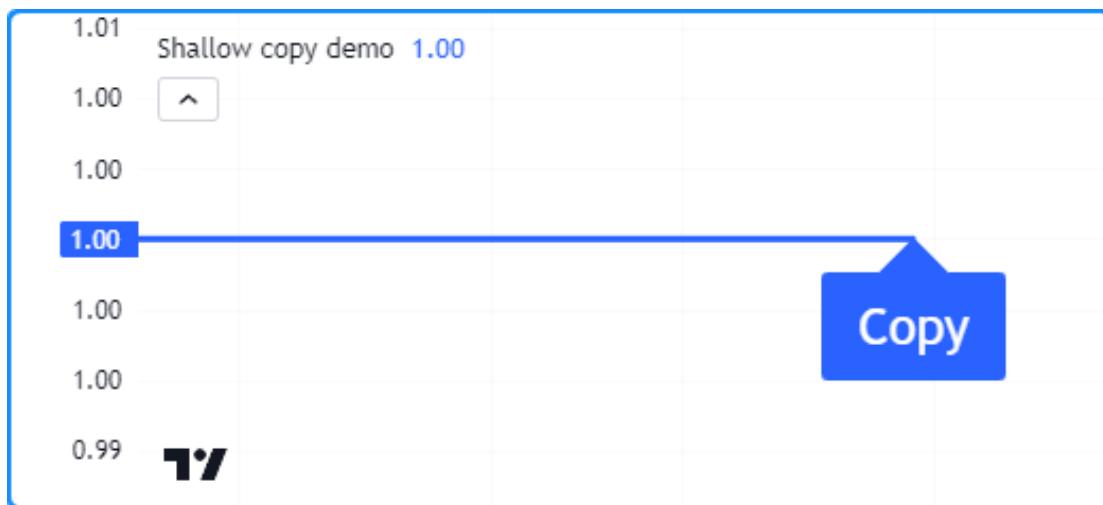
//@variable A shallow copy of `myMatrix`.
matrix<float> myCopy = myMatrix.copy()
// Add a row to the last index of `myCopy`.
myCopy.add_row(myCopy.rows(), array.from(5.0, 6.0))

if bar_index == last_bar_index - 1
    // Display the rows of both matrices in separate labels.
    myMatrix.debugLine(note = "Original")
    myCopy.debugLine(bar_index + 10, color.green, note = "Shallow Copy")

```

值得注意的是，矩阵浅拷贝中的元素指向与原始矩阵相同的值。当矩阵包含特殊类型（[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)或[chart.point](#)）或[用户定义类型](#)时，浅表副本的元素引用与原始对象相同的对象。

该脚本声明一个 `myMatrix` 以 `a newLabel` 作为初始值的变量。然后它 通过`myMatrix.copy()` `myMatrix` 复制到变量并绘制标签数量。如下所示，图表上只有一个[标签](#)，因为 中的元素与 中的元素引用相同的对象。因此，元素值的更改会影响两个矩阵中的值：`myCopy``myCopy``myMatrix``myCopy`



```

//@version=5
indicator("Shallow copy demo")

//@variable Initial value of the original matrix elements.
var label newLabel = label.new(
    bar_index, 1, "Original", color = color.blue, textColor = color.white, size =
    size.huge
)

```

```

//@variable A 1x1 matrix containing a new `label` instance.
var matrix<label> myMatrix = matrix.new<label>(1, 1, nameLabel)
//@variable A shallow copy of `myMatrix`.
var matrix<label> myCopy = myMatrix.copy()

//@variable The first label from the `myCopy` matrix.
label testLabel = myCopy.get(0, 0)

// Change the `text`, `style`, and `x` values of `testLabel`. Also affects the
// `newLabel`.
testLabel.set_text("Copy")
testLabel.set_style(label.style_label_up)
testLabel.set_x(bar_index)

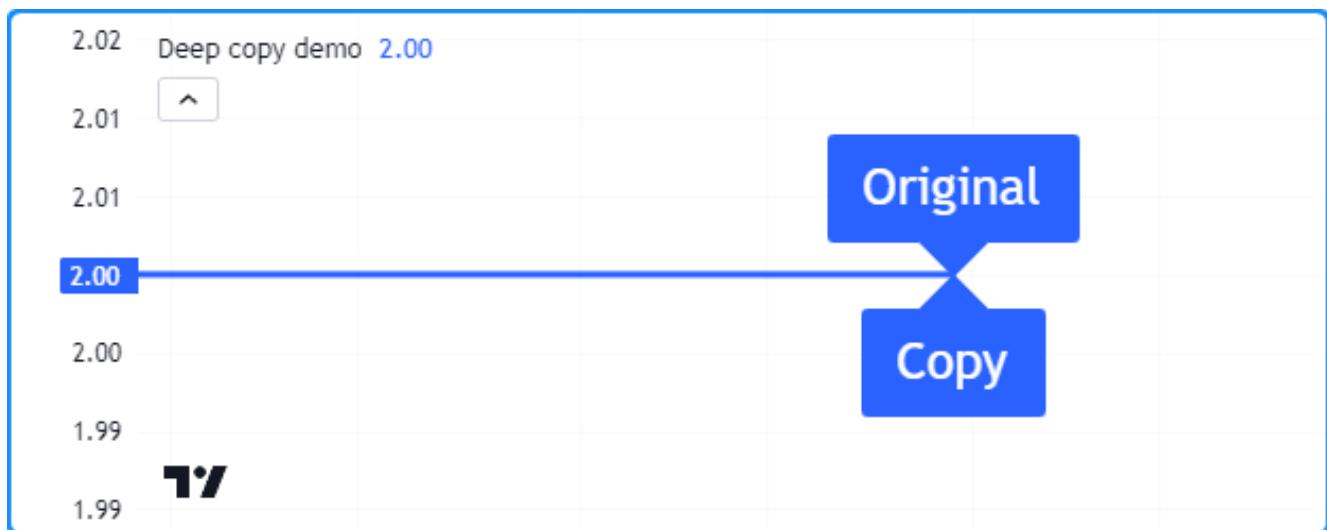
// Plot the total number of labels.
plot(label.all.size(), linewidth = 3)

```

## 深拷贝

通过显式复制矩阵引用的每个对象，可以生成矩阵的深层副本（即，其元素指向原始值的副本的矩阵）。

在这里，我们 `deepCopy()` 在之前的脚本中添加了一个用户定义的方法。该方法创建一个新矩阵并使用 [嵌套 for 循环](#) 将所有元素分配给原始矩阵的副本。当脚本调用此方法而不是内置的 `copy()` 时，我们看到图表上现在有两个标签，并且对标签 `from` 的任何更改都 `myCopy` 不会影响 `from` 标签 `myMatrix`：



```

//@version=5
indicator("Deep copy demo")

//@function Returns a deep copy of a label matrix.
method matrix<label> deepCopy(matrix<label> this) =>
    //@variable A deep copy of `this` matrix.
    matrix<label> that = this.copy()
    for row = 0 to that.rows() - 1

```

```

for column = 0 to that.columns() - 1
    // Assign the element at each `row` and `column` of `that` matrix to a copy
    // of the retrieved label.
    that.set(row, column, that.get(row, column).copy())
that

//@variable Initial value of the original matrix.
var label newLabel = label.new(
    bar_index, 2, "Original", color = color.blue, textcolor = color.white, size =
size.huge
)

//@variable A 1x1 matrix containing a new `label` instance.
var matrix<label> myMatrix = matrix.new<label>(1, 1, newLabel)
//@variable A deep copy of `myMatrix`.
var matrix<label> myCopy = myMatrix.deepcopy()

//@variable The first label from the `myCopy` matrix.
label testLabel = myCopy.get(0, 0)

// Change the `text`, `style`, and `x` values of `testLabel`. Does not affect the
// `newLabel`.
testLabel.set_text("Copy")
testLabel.set_style(label.style_label_up)
testLabel.set_x(bar_index)

// Change the `x` value of `newLabel`.
newLabel.set_x(bar_index)

// Plot the total number of labels.
plot(label.all.size(), linewidth = 3)

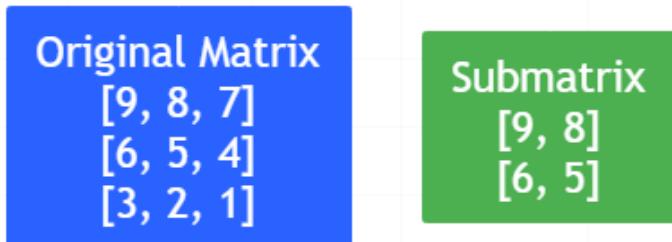
```

## 子矩阵

在 Pine 中，子矩阵是现有矩阵的浅表副本 `from_row/column`，仅包含和参数指定的行和列 `to_row/column`。本质上，它是矩阵的切片副本。

例如，下面的脚本通过 `m.submatrix()` `mSub` 方法从矩阵创建一个矩阵，然后调用我们的用户定义函数在标签中显示两个矩阵的行： `m``debugLabel()`

## Submatrix demo



17

```
//@version=5
indicator("Submatrix demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )
    else
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )

//@variable A 3x3 matrix of values.
var m = matrix.new<float>()

if bar_index == last_bar_index - 1
    // Add columns to `m`.
    m.add_col(0, array.from(9, 6, 3))
    m.add_col(1, array.from(8, 5, 2))
    m.add_col(2, array.from(7, 4, 1))
    // Display the rows of `m`.
    m.debugLabel(note = "Original Matrix")

    //@variable A 2x2 submatrix of `m` containing the first two rows and columns.
    matrix<float> mSub = m.submatrix(from_row = 0, to_row = 2, from_column = 0,
    to_column = 2)
    // Display the rows of `mSub`
```

```
debugLabel(mSub, bar_index + 10, bgColor = color.green, note = "Submatrix")
```

## 范围和历史

矩阵变量在每个柱上留下历史轨迹，允许脚本使用历史引用运算符 `[]` 与先前分配给变量的过去矩阵实例进行交互。此外，脚本可以在[函数](#)、[方法](#)和[条件结构](#)的范围内修改分配给全局变量的矩阵。

该脚本计算实体和烛线距离相对于柱上柱范围的平均比率 `length`。它 `length` 在表格中显示数据以及柱之前的值。用户定义的 `addData()` 函数将当前和历史比率的列添加到 `globalMatrix`，并且该 `calcAvg()` 函数 `previous` 引用分配给 `globalMatrix` 使用 `[]` 运算符的矩阵来计算平均值矩阵：



```
//@version=5  
indicator("Scope and history demo", "Bar ratio comparison")
```

```
int length = input.int(10, "Length", 1)
```

```
//@variable A global matrix.
```

```
matrix<float> globalMatrix = matrix.new<float>()
```

```
//@function Calculates the ratio of body range to candle range.
```

```
bodyRatio() =>
```

```
    math.abs(close - open) / (high - low)
```

```
//@function Calculates the ratio of upper wick range to candle range.
```

```
upperWickRatio() =>
```

```
    (high - math.max(open, close)) / (high - low)
```

```
//@function Calculates the ratio of lower wick range to candle range.
```

```
lowerWickRatio() =>
```

```

        (math.min(open, close) - low) / (high - low)

//@function Adds data to the `globalMatrix`.
addData() =>
    // Add a new column of data at `column` 0.
    globalMatrix.add_col(0, array.from(bodyRatio(), upperWickRatio(),
lowerWickRatio()))
    //@variable The column of `globalMatrix` from index 0 `length` bars ago.
    array<float> pastValues = globalMatrix.col(0)[length]
    // Add `pastValues` to the `globalMatrix`, or an array of `na` if `pastValues` is
`na`.
    if na(pastValues)
        globalMatrix.add_col(1, array.new<float>(3))
    else
        globalMatrix.add_col(1, pastValues)

//@function Returns the `length`-bar average of matrices assigned to `globalMatrix` on
historical bars.
calcAvg() =>
    //@variable The sum historical `globalMatrix` matrices.
    matrix<float> sums = matrix.new<float>(globalMatrix.rows(), globalMatrix.columns(),
0.0)
    for i = 0 to length - 1
        //@variable The `globalMatrix` matrix `i` bars before the current bar.
        matrix<float> previous = globalMatrix[i]
        // Break the loop if `previous` is `na`.
        if na(previous)
            sums.fill(na)
            break
        // Assign the sum of `sums` and `previous` to `sums`.
        sums := matrix.sum(sums, previous)
    // Divide the `sums` matrix by the `length`.
    result = sums.mult(1.0 / length)

// Add data to the `globalMatrix`.
addData()

//@variable The historical average of the `globalMatrix` matrices.
globalAvg = calcAvg()

//@variable A `table` displaying information from the `globalMatrix`.
var table infoTable = table.new(
    position.middle_center, globalMatrix.columns() + 1, globalMatrix.rows() + 1,
bgcolor = color.navy
)

// Define value cells.
for [i, row] in globalAvg
    for [j, value] in row

```

```

        color textColor = value > 0.333 ? color.orange : color.gray
        infoTable.cell(j + 1, i + 1, str.toString(value), text_color = textColor,
text_size = size.huge)

// Define header cells.
infoTable.cell(0, 1, "Body ratio", text_color = color.white, text_size = size.huge)
infoTable.cell(0, 2, "Upper wick ratio", text_color = color.white, text_size =
size.huge)
infoTable.cell(0, 3, "Lower wick ratio", text_color = color.white, text_size =
size.huge)
infoTable.cell(1, 0, "Current average", text_color = color.white, text_size =
size.huge)
infoTable.cell(2, 0, str.format("{0} bars ago", length), text_color = color.white,
text_size = size.huge)

```

- 注意：

和函数没有参数，因为它们直接与外部作用域中声明的 和 变量 `addData()` 交互。`calcAvg()``globalMatrix``length``calcAvg()` 通过使用 `matrix.sum()` 将矩阵相加并使用 `matrix.mult()` 将所有元素相乘来计算平均值。我们在下面的 [矩阵计算](#) 部分讨论这些和其他专用函数。

## 检查矩阵

检查矩阵的形状及其元素内的模式的能力至关重要，因为它有助于揭示有关矩阵的重要信息及其与各种计算和转换的兼容性。Pine Script™ 包含多个用于矩阵检查的内置函数，包括 [matrix.is\\_square\(\)](#)、[matrix.is\\_identity\(\)](#)、[matrix.is\\_diagonal\(\)](#)、[matrix.is\\_antidiagonal\(\)](#)、[matrix.is\\_symmetry\(\)](#)、[matrix.is\\_antisymmetry\(\)](#)、[matrix.is\\_triangle\(\)](#)、[matrix.is\\_stochastic\(\)](#)、[matrix.is\\_binary\(\)](#) 和 [matrix.is\\_zero\(\)](#)。

为了演示这些功能，此示例包含一个自定义 `inspect()` 方法，该方法使用带有 `matrix.is_*` 函数的条件块来返回有关矩阵的信息。它显示矩阵的字符串表示形式以及从图表上的标签 `m` 返回的描述：`m.inspect()`

The screenshot shows a Pine Script editor window titled "Matrix inspection demo". It displays a 4x4 matrix with values [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], and [0, 0, 0, 1]. A callout box highlights the matrix and lists its properties: it is square, contains only 1s and 0s, is diagonal, has nonzero values only on the main diagonal (making it symmetric and antisymmetric), equals its transpose, equals the negative of its transpose, and is the identity matrix.

```

//@version=5
indicator("Matrix inspection demo")

```

```

//@function Inspects a matrix using `matrix.is_*()` functions and returns a `string`  

describing some of its features.  

method inspect(matrix<int> this)=>  

    //@variable A string describing `this` matrix.  

    string result = "This matrix:\n"  

    if this.is_square()  

        result += "- Has an equal number of rows and columns.\n"  

    if this.is_binary()  

        result += "- Contains only 1s and 0s.\n"  

    if this.is_zero()  

        result += "- Is filled with 0s.\n"  

    if this.is_triangular()  

        result += "- Contains only 0s above and/or below its main diagonal.\n"  

    if this.is_diagonal()  

        result += "- Only has nonzero values in its main diagonal.\n"  

    if this.is_antidiagonal()  

        result += "- Only has nonzero values in its main antidiagonal.\n"  

    if this.is_symmetric()  

        result += "- Equals its transpose.\n"  

    if this.is_antisymmetric()  

        result += "- Equals the negative of its transpose.\n"  

    if this.is_identity()  

        result += "- Is the identity matrix.\n"  

    result  
  

//@variable A 4x4 identity matrix.  

matrix<int> m = matrix.new<int>()  
  

// Add rows to the matrix.  

m.add_row(0, array.from(1, 0, 0, 0))  

m.add_row(1, array.from(0, 1, 0, 0))  

m.add_row(2, array.from(0, 0, 1, 0))  

m.add_row(3, array.from(0, 0, 0, 1))  
  

if bar_index == last_bar_index - 1  

    // Display the `m` matrix in a blue label.  

    label.new(  

        bar_index, 0, str.tostring(m), color = color.blue, style =  

label.style_label_right,  

        textcolor = color.white, size = size.huge  

    )  

    // Display the result of `m.inspect()` in a purple label.  

    label.new(  

        bar_index, 0, m.inspect(), color = color.purple, style =  

label.style_label_left,  

        textcolor = color.white, size = size.huge  

    )

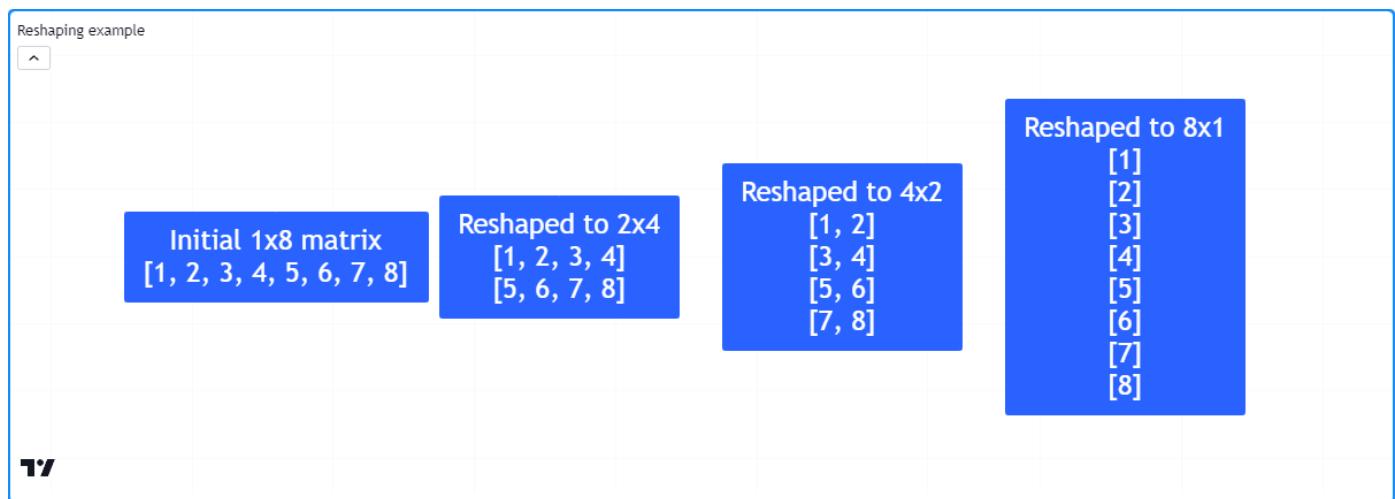
```

# 操作矩阵

## 重塑

矩阵的形状可以决定其与各种矩阵运算的兼容性。在某些情况下，有必要在不影响元素数量或它们引用的值的情况下更改矩阵的维度，也称为重塑。要在 Pine 中重塑矩阵，请使用 [matrix.reshape\(\)](#) 函数。

此示例演示了对矩阵进行多次整形操作的结果。初始 `m` 矩阵具有  $1 \times 8$  形状（一行八列）。通过连续调用 [m.reshape\(\)](#) 方法，脚本将形状更改 `m` 为  $2 \times 4$ 、 $4 \times 2$  和  $8 \times 1$ 。它使用自定义方法在图表上的标签中显示每个重塑矩阵 `debugLabel()`：



```
//@version=5
indicator("Reshaping example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )

//@variable A matrix containing the values 1-8.
matrix<int> m = matrix.new<int>()
```

```

if bar_index == last_bar_index - 1
    // Add the initial vector of values.
    m.add_row(0, array.from(1, 2, 3, 4, 5, 6, 7, 8))
    m.debugLine(note = "Initial 1x8 matrix")

    // Reshape. `m` now has 2 rows and 4 columns.
    m.reshape(2, 4)
    m.debugLine(bar_index + 10, note = "Reshaped to 2x4")

    // Reshape. `m` now has 4 rows and 2 columns.
    m.reshape(4, 2)
    m.debugLine(bar_index + 20, note = "Reshaped to 4x2")

    // Reshape. `m` now has 8 rows and 1 column.
    m.reshape(8, 1)
    m.debugLine(bar_index + 30, note = "Reshaped to 8x1")

```

- 注意：

每次调用时元素的顺序 `m` 不会改变 `m.reshape()`。`rows` 重塑矩阵时，和参数的乘积 `columns` 必须等于 `matrix.elements_count()` 值，因为 `matrix.reshape()` 无法更改矩阵中的元素数量。

## 反转

可以使用 `matrix.reverse()` 反转矩阵中所有元素的顺序。该函数将  $m \times n$  矩阵 `id` 的第  $i$  行第  $j$  列的引用移动到第  $m - 1 - i$  行和  $n - 1 - j$  列。

例如，此脚本创建一个  $3 \times 3$  矩阵，其中包含按升序排列的值 1-9，然后使用 `reverse()` 方法反转其内容。它通过以下方式在图表上的标签中显示矩阵的原始版本和修改版本 `m.debugLine()`：



```

//@version=5
indicator("Reversing demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.

```

```

//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )
    )

//@variable A 3x3 matrix.
matrix<float> m = matrix.new<float>()

// Add rows to `m`.
m.add_row(0, array.from(1, 2, 3))
m.add_row(1, array.from(4, 5, 6))
m.add_row(2, array.from(7, 8, 9))

if bar_index == last_bar_index - 1
    // Display the contents of `m`.
    m.debugLabel(note = "Original")
    // Reverse `m`, then display its contents.
    m.reverse()
    m.debugLabel(bar_index + 10, color.red, note = "Reversed")

```

## 转置

转置矩阵是一项基本操作，它围绕 主对角线（行索引等于列索引的所有值的对角向量）翻转矩阵中的所有行和列。此过程会产生一个具有相反行和列维度的新矩阵，称为转置。脚本可以使用[matrix.transpose\(\)](#)计算矩阵的转置。

对于任何 m 行、n 列的矩阵，从[matrix.transpose\(\)](#)返回的矩阵 将具有 n 行和 m 列。矩阵中第 i 行第 j 列的所有元素对应于其转置中第 j 行第 i 列的元素。

此示例声明一个 2x4 m 矩阵，使用 [m.transpose\(\)](#)方法计算其转置，并使用我们的自定义方法在图表上显示两个矩阵 `debugLabel()`。如下所示，转置矩阵的形状为 4x2，转置后的行与原始矩阵的列相匹配：

## Transpose example



Original  
[1, 2, 3, 4]  
[5, 6, 7, 8]

Transpose  
[1, 5]  
[2, 6]  
[3, 7]  
[4, 8]

17

```
//@version=5
indicator("Transpose example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )
    else
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )

//@variable A 2x4 matrix.
matrix<int> m = matrix.new<int>()

// Add columns to `m`.
m.add_col(0, array.from(1, 5))
m.add_col(1, array.from(2, 6))
m.add_col(2, array.from(3, 7))
m.add_col(3, array.from(4, 8))

//@variable The transpose of `m`. Has a 4x2 shape.
matrix<int> mt = m.transpose()

if bar_index == last_bar_index - 1
    m.debugLabel(note = "Original")
```

```
mt.debugLabel(bar_index + 10, note = "Transpose")
```

## 排序

脚本可以通过`matrix.sort()`对矩阵的内容进行排序。与对元素进行排序的`array.sort()`不同，此函数根据指定的值按指定的顺序（默认为`order.ascending`）组织矩阵中的所有行。`order^`column`

此脚本声明一个 3x3 矩阵，根据第一列对副本的行进行升序排序，然后根据第二列对副本的行进行降序排序。它使用我们的方法显示原始矩阵和标签中的排序副本：`m1``m2``debugLabel()`

Sorting rows example

Original  
[3, 2, 4]  
[1, 9, 6]  
[7, 8, 9]

Sorted using col 0  
(Ascending)  
[1, 9, 6]  
[3, 2, 4]  
[7, 8, 9]

Sorted using col 1  
(Descending)  
[1, 9, 6]  
[7, 8, 9]  
[3, 2, 4]

```
//@version=5
indicator("Sorting rows example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )
    else
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )

//@variable A 3x3 matrix.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
```

```

// Add rows to `m`.
m.add_row(0, array.from(3, 2, 4))
m.add_row(1, array.from(1, 9, 6))
m.add_row(2, array.from(7, 8, 9))
m.debugLine(note = "Original")

// Copy `m` and sort rows in ascending order based on the first column (default).
matrix<int> m1 = m.copy()
m1.sort()
m1.debugLine(bar_index + 10, color.green, note = "Sorted using col 0\n(Ascending)")

// Copy `m` and sort rows in descending order based on the second column.
matrix<int> m2 = m.copy()
m2.sort(1, order.descending)
m2.debugLine(bar_index + 20, color.red, note = "Sorted using col 1\n(Descending)")

```

需要注意的是，[matrix.sort\(\)](#) 不会对矩阵的列进行排序。然而，我们可以使用这个函数在[matrix.transpose\(\)](#)的帮助下对矩阵列进行排序。

例如，此脚本包含一个 `sortColumns()` 方法，该方法使用 [sort\(\)](#) 方法使用与原始矩阵 对应的列对矩阵的转置进行排序。该脚本使用此方法根据第一行的内容对矩阵 `row` 进行排序：

### Sorting columns example



Original  
[3, 2, 4]  
[1, 9, 6]  
[7, 8, 9]

Sorted using row 0  
(Ascending)  
[2, 3, 4]  
[9, 1, 6]  
[8, 7, 9]



```

//@version=5
indicator("Sorting columns example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""

```

```

) =>
    labelText = note + "\n" + str.toString(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )

//@function Sorts the columns of `this` matrix based on the values in the specified
`row`.

method sortColumns(matrix<int> this, int row = 0, bool ascending = true) =>
    // @variable The transpose of `this` matrix.
    matrix<int> thisT = this.transpose()
    // @variable Is `order.ascending` when `ascending` is `true`, `order.descending`
otherwise.
    order = ascending ? order.ascending : order.descending
    // Sort the rows of `thisT` using the `row` column.
    thisT.sort(row, order)
    // @variable A copy of `this` matrix with sorted columns.
    result = thisT.transpose()

// @variable A 3x3 matrix.
matrix<int> m = matrix.new<int>()

if bar_index == last_bar_index - 1
    // Add rows to `m`.
    m.add_row(0, array.from(3, 2, 4))
    m.add_row(1, array.from(1, 9, 6))
    m.add_row(2, array.from(7, 8, 9))
    m.debugLabel(note = "Original")

    // Sort the columns of `m` based on the first row and display the result.
    m.sortColumns(0).debugLabel(bar_index + 10, note = "Sorted using row
0\n(Ascending)")

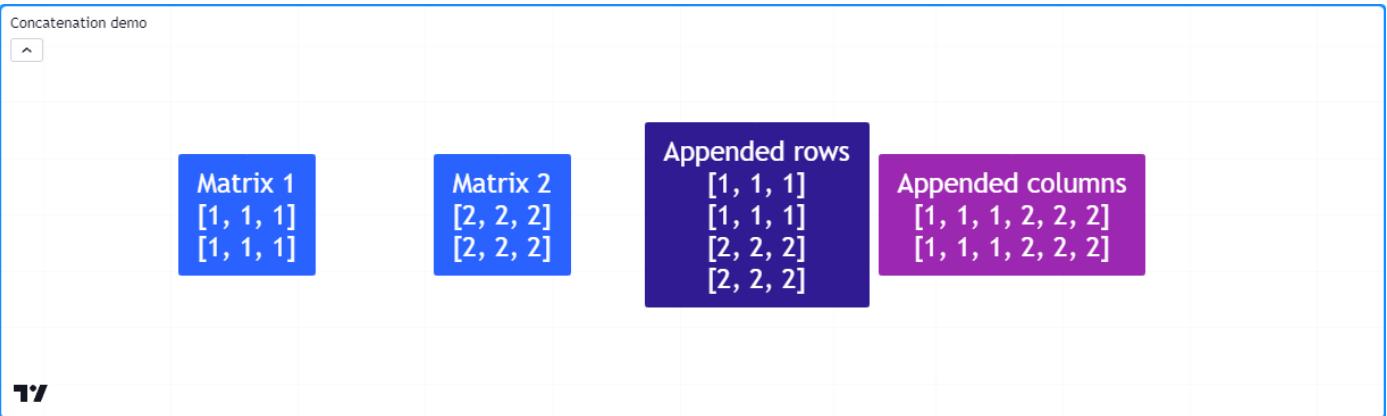
```

## 连接

脚本可以使用[matrix.concat\(\)](#)连接两个矩阵。此函数将矩阵的行附加到具有相同列数的矩阵的末尾。`id2``id1`

要创建一个矩阵，其中的元素表示附加到另一个矩阵的列，[请转置](#)两个矩阵，对转置后的矩阵使用[matrix.concat\(\)](#)，[然后转置\(\)](#)结果。

例如，此脚本将矩阵的行附加 `m2` 到矩阵，并使用矩阵的转置副本 `m1` 附加其列。它使用自定义方法在标签中连接行和列后显示和矩阵以及结果：`m1``m2``debugLabel()`



77

```
//@version=5
indicator("Concatenation demo")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x3 matrix filled with 1s.
matrix<int> m1 = matrix.new<int>(2, 3, 1)
//@variable A 2x3 matrix filled with 2s.
matrix<int> m2 = matrix.new<int>(2, 3, 2)

//@variable The transpose of `m1`.
t1 = m1.transpose()
//@variable The transpose of `m2`.
t2 = m2.transpose()

if bar_index == last_bar_index - 1
    // Display the original matrices.
    m1.debugLabel(note = "Matrix 1")
    m2.debugLabel(bar_index + 10, note = "Matrix 2")
    // Append the rows of `m2` to the end of `m1` and display `m1`.
    m1.concat(m2)
    m1.debugLabel(bar_index + 20, color.blue, note = "Appended rows")
    // Append the rows of `t2` to the end of `t1`, then display the transpose of `t1`.
```

```
t1.concat(t2)
t1.transpose().debugLabel(bar_index + 30, color.purple, note = "Appended columns")
```

## 矩阵计算

### 逐元素计算

Pine脚本可以通过 `matrix.avg()`、`matrix.min()`、`matrix.max()` 和 `matrix.mode()` 计算矩阵中所有元素的平均值、最小值、最大值和众数。这些函数的操作与其等效函数相同，允许用户使用相同的语法对矩阵、其子矩阵及其行和列运行逐元素计算。例如，在具有值 1-9 的 3x3 矩阵和具有相同九个元素的数组上调用的内置函数都将返回值 5。`array.*`~*.avg()`

下面的脚本使用 `*.avg()`、`*.max()` 和 `*.min()` 方法来计算一段时间内 OHLC 数据的发展平均值和极值。无论何时，它都会在矩阵末尾添加一列新的开盘价、最高价、最低价和收盘价。当时，脚本使用 `get()` 和 `set()` 矩阵方法调整最后一列中的元素，以得出当前周期的 HLC 值。它使用矩阵、`submatrix()`、`row()` 和 `col()` 数组来计算一段时间内发展中的 OHLC4 和 HL2 平均值、一段时间内的最大高点和最小低点，以及当前时期的发展中 OHLC4 价格：`ohlcData`~queueColumn`~true`~false`~ohlcData`~length`~length`



```
//@version=5
indicator("Element-wise calculations example", "Developing values", overlay = true)

//@variable The number of data points in the averages.
int length = input.int(3, "Length", 1)
//@variable The timeframe of each reset period.
string timeframe = input.timeframe("D", "Reset Timeframe")

//@variable A 4x`length` matrix of OHLC values.
```

```

var matrix<float> ohlcData = matrix.new<float>(4, length)

//@variable Is `true` at the start of a new bar at the `timeframe`.
bool queueColumn = timeframe.change(timeframe)

if queueColumn
    // Add new values to the end column of `ohlcData`.
    ohlcData.add_col(length, array.from(open, high, low, close))
    // Remove the oldest column from `ohlcData`.
    ohlcData.remove_col(0)
else
    // Adjust the last element of column 1 for new highs.
    if high > ohlcData.get(1, length - 1)
        ohlcData.set(1, length - 1, high)
    // Adjust the last element of column 2 for new lows.
    if low < ohlcData.get(2, length - 1)
        ohlcData.set(2, length - 1, low)
    // Adjust the last element of column 3 for the new closing price.
    ohlcData.set(3, length - 1, close)

//@variable The `matrix.avg()` of all elements in `ohlcData`.
avgOHLC4 = ohlcData.avg()
//@variable The `matrix.avg()` of all elements in rows 1 and 2, i.e., the average of
all `high` and `low` values.
avgHL2 = ohlcData.submatrix(from_row = 1, to_row = 3).avg()
//@variable The `matrix.max()` of all values in `ohlcData`. Equivalent to
`ohlcData.row(1).max()`.
maxHigh = ohlcData.max()
//@variable The `array.min()` of all `low` values in `ohlcData`. Equivalent to
`ohlcData.min()`.
minLow = ohlcData.row(2).min()
//@variable The `array.avg()` of the last column in `ohlcData`, i.e., the current
OHLC4.
ohlc4Value = ohlcData.col(length - 1).avg()

plot(avgOHLC4, "Average OHLC4", color.purple, 2)
plot(avgHL2, "Average HL2", color.navy, 2)
plot(maxHigh, "Max High", color.green)
plot(minLow, "Min Low", color.red)
plot(ohlc4Value, "Current OHLC4", color.blue)

```

- 注意：

在此示例中，我们交替使用[array.\\*\(\)](#)和[matrix.\\*\(\)](#)方法来演示它们在语法和行为上的相似性。用户可以通过将[matrix.avg\(\)](#)乘以[matrix.elements\\_count\(\)](#)来计算[array.sum\(\)](#)的等价矩阵。

## 特殊计算

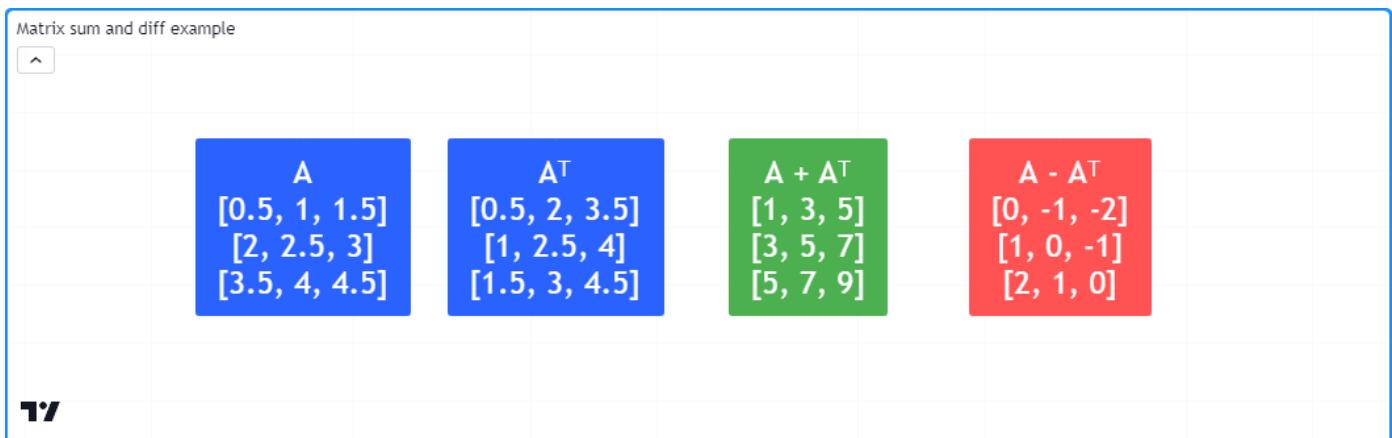
Pine Script™ 具有多个用于执行基本矩阵算术和线性代数运算的内置函数，包括[matrix.sum\(\)](#)、[matrix.diff\(\)](#)、[matrix.mult\(\)](#)、[matrix.pow\(\)](#)、[matrix.det\(\)](#)、[矩阵.inv\(\)](#)、[矩阵.pinv\(\)](#)、[矩阵.rank\(\)](#)、[矩阵.trace\(\)](#)、[矩阵特征值\(\)](#)、[矩阵特征向量\(\)](#)和[矩阵.kron\(\)](#)。这些函数是高级功能，可促进各种矩阵计算和转换。

下面，我们通过一些基本示例来解释一些基本功能。

### [matrix.sum\(\)](#) 和 [matrix.diff\(\)](#)

脚本可以使用[matrix.sum\(\)](#)和[matrix.diff\(\)](#)函数对具有相同形状的两个矩阵或一个矩阵和一个标量值执行加法和减法。这些函数使用矩阵或标量中的值 `id2` 来添加或减去 中的元素 `id1`。

该脚本演示了 Pine 中矩阵加法和减法的简单示例。它创建一个3x3矩阵，计算其转置，然后计算 两个矩阵的 [matrix.sum\(\)](#) 和 [matrix.diff\(\)](#)。此示例在图表上的标签中显示原始矩阵、其 转置以及生成的和矩阵和差矩阵：



```
//@version=5
indicator("Matrix sum and diff example")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )
    else
        label.set_text(labelText, barIndex, 0, color = bgColor, style = label.style_label_center,
            textColor = textColor, size = size.huge
        )
    end
)

//@variable A 3x3 matrix.
```

```

m = matrix.new<float>()

// Add rows to `m`.
m.add_row(0, array.from(0.5, 1.0, 1.5))
m.add_row(1, array.from(2.0, 2.5, 3.0))
m.add_row(2, array.from(3.5, 4.0, 4.5))

if bar_index == last_bar_index - 1
    // Display `m`.
    m.debugLine(note = "A")
    // Get and display the transpose of `m`.
    matrix<float> t = m.transpose()
    t.debugLine(bar_index + 10, note = "AT")
    // Calculate the sum of the two matrices. The resulting matrix is symmetric.
    matrix.sum(m, t)..debugLine(bar_index + 20, color.green, note = "A + AT")
    // Calculate the difference between the two matrices. The resulting matrix is
    antisymmetric.
    matrix.diff(m, t)..debugLine(bar_index + 30, color.red, note = "A - AT")

```

- 注意：

在此示例中，我们将原始矩阵标记为“A”，将转置矩阵标记为“AT”。将“A”和“AT”相加生成对称矩阵，将它们相减生成反对称矩阵。

## matrix.mult()

脚本可以通过matrix.mult()函数将两个矩阵相乘。此函数还有助于矩阵与数组或标量值的乘法。

在将两个矩阵相乘的情况下，与加法和减法不同，矩阵乘法不需要两个矩阵共享相同的形状。但是，第一个矩阵中的列数必须等于第二个矩阵中的行数。matrix.mult()返回的结果矩阵将包含与相同的行数 id1 和相同的列数 id2。例如，2x3 矩阵乘以 3x4 矩阵将产生一个两行四列的矩阵，如下所示。生成的矩阵中的每个值都是中相对应行和列的点积：id1``id2

Matrix mult example

Matrix mult example		
<b>A</b> [1, 2, 3] [4, 5, 6]	<b>B</b> [4.5, 5, 5.5, 6] [0.5, 1, 1.5, 2] [2.5, 3, 3.5, 4]	<b>A * B</b> [13, 16, 19, 22] [35.5, 43, 50.5, 58]

17

```

//@version=5
indicator("Matrix mult example")

```

```

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.

method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )
    else
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x3 matrix.
a = matrix.new<float>()
//@variable A 3x4 matrix.
b = matrix.new<float>()

// Add rows to `a`.
a.add_row(0, array.from(1, 2, 3))
a.add_row(1, array.from(4, 5, 6))

// Add rows to `b`.
b.add_row(0, array.from(0.5, 1.0, 1.5, 2.0))
b.add_row(1, array.from(2.5, 3.0, 3.5, 4.0))
b.add_row(2, array.from(4.5, 5.0, 5.5, 6.0))

if bar_index == last_bar_index - 1
    // @variable The result of `a` * `b`.
    matrix<float> ab = a.mult(b)
    // Display `a`, `b`, and `ab` matrices.
    debugLabel(a, note = "A")
    debugLabel(b, bar_index + 10, note = "B")
    debugLabel(ab, bar_index + 20, color.green, note = "A * B")

```

- 注意：

与标量乘法相反，矩阵乘法是不可交换的，即不一定产生与相同的结果。在我们的示例中，后者将引发运行时错误，因为中的列数不等于中的行数。`matrix.mult(a, b)` `matrix.mult(b, a)` `b``a`

当矩阵和数组`id1`相乘时，该函数将运算视为与单列矩阵相乘，但它返回一个数组，其元素数与中的行数相同`id1`。当`matrix.mult()`传递一个标量作为其`id2`值时，该函数返回一个新矩阵，其元素是`in`中的元素`id1`乘以该`id2`值。

## matrix.det()

行列式是与方阵相关的标量值，描述了方阵的一些特征，即其可逆性。如果矩阵有逆矩阵，则其行列式非零。否则，矩阵是奇异的（不可逆的）。脚本可以通过[matrix.det\(\)](#)计算矩阵的行列式。

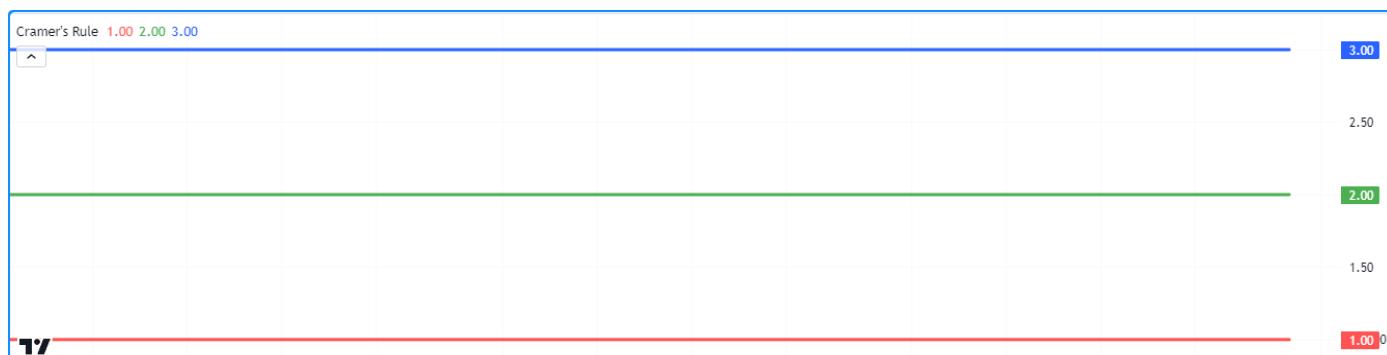
程序员可以使用行列式来检测矩阵之间的相似性、识别满秩矩阵和缺秩矩阵、求解线性方程组等应用。

例如，此脚本利用行列式通过[克莱默规则](#)求解具有匹配数量的未知值的线性方程组。用户定义[solve\(\)](#)函数返回一个[数组](#)，其中包含系统中每个未知值的解，其中数组的第n个元素是系数矩阵的行列式，第n列替换为常量列除以行列式的原始系数。

在此脚本中，我们定义了m保存这三个方程的系数和常数的矩阵：

```
3 * x0 + 4 * x1 - 1 * x2 = 8  
5 * x0 - 2 * x1 + 1 * x2 = 4  
2 * x0 - 2 * x1 + 1 * x2 = 1
```

该系统的解决方案是.该脚本通过via计算这些值并将它们绘制在图表上：(x0 = 1, x1 = 2, x2 = 3)``m``m.solve()



```
//@version=5  
indicator("Determinants example", "Cramer's Rule")  
  
//@function Solves a system of linear equations with a matching number of unknowns  
using Cramer's rule.  
//@param    this An augmented matrix containing the coefficients for each unknown and  
the results of  
//          the equations. For example, a row containing the values 2, -1, and 3  
represents the equation  
//          `2 * x0 + (-1) * x1 = 3` , where `x0` and `x1` are the unknown values in the  
system.  
//@returns  An array containing solutions for each variable in the system.  
solve(matrix<float> this) =>  
    //@variable The coefficient matrix for the system of equations.  
    matrix<float> coefficients = this.submatrix(from_column = 0, to_column =  
this.columns() - 1)  
    //@variable The array of resulting constants for each equation.  
    array<float> constants = this.col(this.columns() - 1)  
    //@variable An array containing solutions for each unknown in the system.  
    array<float> result = array.new<float>()
```

```

//@variable The determinant value of the coefficient matrix.
float baseDet = coefficients.det()
matrix<float> modified = na
for col = 0 to coefficients.columns() - 1
    modified := coefficients.copy()
    modified.add_col(col, constants)
    modified.remove_col(col + 1)

    // Calculate the solution for the column's unknown by dividing the determinant
    // of `modified` by the `baseDet`.
    result.push(modified.det() / baseDet)

result

//@variable A 3x4 matrix containing coefficients and results for a system of three
// equations.
m = matrix.new<float>()

// Add rows for the following equations:
// Equation 1: 3 * x0 + 4 * x1 - 1 * x2 = 8
// Equation 2: 5 * x0 - 2 * x1 + 1 * x2 = 4
// Equation 3: 2 * x0 - 2 * x1 + 1 * x2 = 1
m.add_row(0, array.from(3.0, 4.0, -1.0, 8.0))
m.add_row(1, array.from(5.0, -2.0, 1.0, 4.0))
m.add_row(2, array.from(2.0, -2.0, 1.0, 1.0))

//@variable An array of solutions to the unknowns in the system of equations
// represented by `m`.
solutions = solve(m)

plot(solutions.get(0), "x0", color.red, 3) // Plots 1.
plot(solutions.get(1), "x1", color.green, 3) // Plots 2.
plot(solutions.get(2), "x2", color.blue, 3) // Plots 3.

```

- 注意：

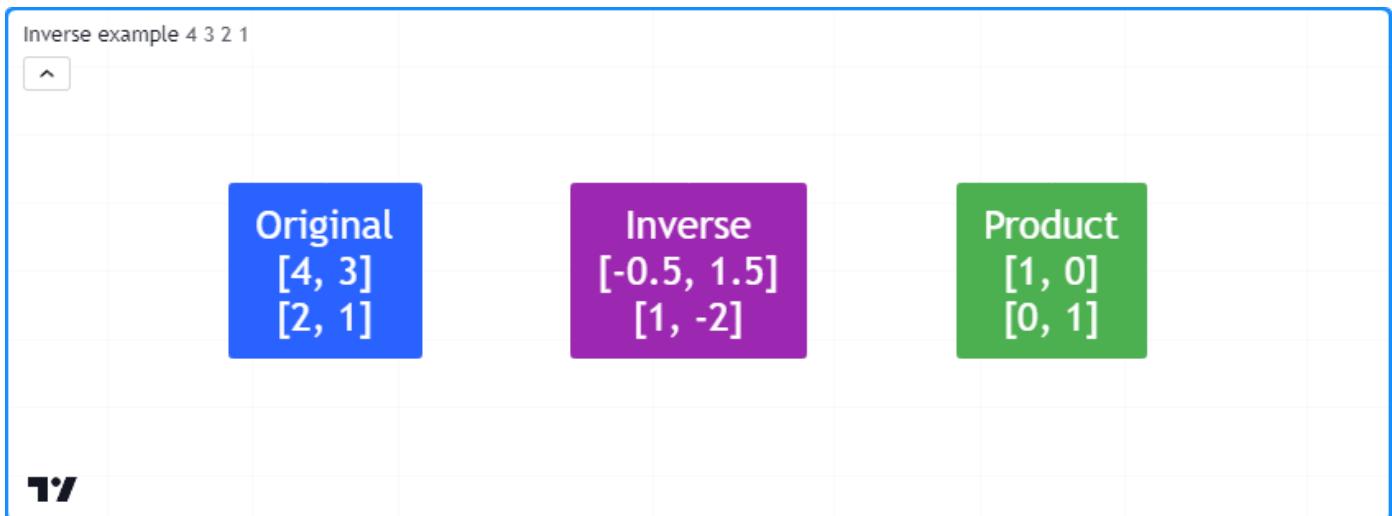
求解方程组对于回归分析特别有用，例如线性回归和多项式回归。克莱默规则适用于小型方程组。然而，在较大的系统上，它的计算效率很低。对于此类用例，通常首选其他方法，例如[高斯消除法](#)。

## [matrix.inv\(\)](#) 和 [matrix.pinv\(\)](#)

对于任何非奇异方阵，都有一个逆矩阵，当乘以原始矩阵时会产生单位矩阵。逆矩阵在各种矩阵变换和求解方程组中都有用处。当矩阵存在时，脚本可以通过[matrix.inv\(\)](#)函数计算矩阵的逆。

对于奇异（不可逆）矩阵，无论矩阵是方阵还是具有非零行列式，都可以通过[matrix.pinv\(\)](#)函数计算广义逆（伪逆）。请记住，与真逆不同，伪逆与原始矩阵的乘积不一定等于单位矩阵，除非原始矩阵是可逆的。

以下示例根据用户输入形成  $2 \times 2$  矩阵，然后使用 `m.inv()` 和 `m.pinv()` 方法计算的逆或伪逆。该脚本在图表上的标签中显示原始矩阵、其逆矩阵或伪逆矩阵及其乘积：



```
//@version=5
indicator("Inverse example")

// Element inputs for the 2x2 matrix.
float r0c0 = input.float(4.0, "Row 0, Col 0")
float r0c1 = input.float(3.0, "Row 0, Col 1")
float r1c0 = input.float(2.0, "Row 1, Col 0")
float r1c1 = input.float(1.0, "Row 1, Col 1")

//@function Displays the rows of a matrix in a label with a note.
//@param    this The matrix to display.
//@param    barIndex The `bar_index` to display the label at.
//@param    bgColor The background color of the label.
//@param    textColor The color of the label's text.
//@param    note The text to display above the rows.
method debugLabel(
    matrix<float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    labelText = note + "\n" + str.tostring(this)
    if barstate.ishistory
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )
    else
        label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )

//@variable A 2x2 matrix of input values.
m = matrix.new<float>()

// Add input values to `m`.
m.add_row(0, array.from(r0c0, r0c1))
m.add_row(1, array.from(r1c0, r1c1))
```

```

//@variable Is `true` if `m` is square with a nonzero determinant, indicating
invertibility.
bool isInvertible = m.is_square() and m.det()

//@variable The inverse or pseudoinverse of `m`.
mInverse = isInvertible ? m.inv() : m.pinv()

//@variable The product of `m` and `mInverse`. Returns the identity matrix when
`isInvertible` is `true`.
matrix<float> product = m.mult(mInverse)

if bar_index == last_bar_index - 1
    // Display `m`, `mInverse`, and their `product`.
    m.debugLine(note = "Original")
    mInverse.debugLine(bar_index + 10, color.purple, note = isInvertible ? "Inverse" :
"Pseudoinverse")
    product.debugLine(bar_index + 20, color.green, note = "Product")

```

- 注意：

该脚本仅在为时调用`m.inv()`，即当为平方且行列式为非零时。否则，它使用`m.pinv()`计算广义逆。`isInvertible``true``m`

### `matrix.rank()`

矩阵的秩表示它包含的线性无关向量（行或列）的数量。本质上，矩阵秩衡量的是无法表达为其他向量的线性组合的向量的数量，或者换句话说，是包含唯一信息的向量的数量。脚本可以通过`matrix.rank()`计算矩阵的秩。

`m1` 此脚本识别两个  $3 \times 3$  矩阵 (和) 中线性无关向量的数量 `m2`，并在单独的窗格中绘制这些值。正如我们在图表中看到的，`m1.rank()` 值为 3，因为每个向量都是唯一的。另一方面，`m2.rank()` 值为 1，因为它只有一个唯一向量：



```

//@version=5
indicator("Matrix rank example")

//@variable A 3x3 full-rank matrix.
m1 = matrix.new<float>()

```

```

// @variable A 3x3 rank-deficient matrix.
m2 = matrix.new<float>()

// Add linearly independent vectors to `m1`.
m1.add_row(0, array.from(3, 2, 3))
m1.add_row(1, array.from(4, 6, 6))
m1.add_row(2, array.from(7, 4, 9))

// Add linearly dependent vectors to `m2`.
m2.add_row(0, array.from(1, 2, 3))
m2.add_row(1, array.from(2, 4, 6))
m2.add_row(2, array.from(3, 6, 9))

// Plot `matrix.rank()` values.
plot(m1.rank(), color = color.green, linewidth = 3)
plot(m2.rank(), color = color.red, linewidth = 3)

```

- 注意：

矩阵可以具有的最高等级值是其行数和列数中的最小值。具有最大可能秩的矩阵称为满秩矩阵，任何没有满秩的矩阵称为缺秩矩阵。满秩方阵的行列式非零，并且此类矩阵具有逆矩阵。相反，秩亏矩阵的行列式始终为0。对于每个元素只包含相同值的任何矩阵（例如，用0填充的矩阵），秩始终为0，因为没有向量保存唯一信息。对于具有不同值的任何其他矩阵，最小可能的秩为1。

## 错误处理

除了在脚本编译期间由于语法不正确而发生的常见编译器错误之外，使用矩阵的脚本还可能在执行期间引发特定的运行时错误。当脚本引发运行时错误时，它会在脚本标题旁边显示一个红色感叹号。用户可以通过单击该图标来查看错误消息。

在本节中，我们讨论用户在脚本中使用矩阵时可能遇到的运行时错误。

### 行/列索引(xx)越界，行/列大小为(yy)。

当尝试使用包含[math.get\(\)](#)、[math.set\(\)](#)、[math.fill\(\)](#)和[math.submatrix\(\)](#)的函数以及一些与矩阵的行和列。

例如，此代码包含两行将产生此运行时错误。`m.set()`方法引用了`row`不存在的索引(2)。`m.submatrix()`方法引用直到的所有列索引。值4会导致运行时错误，因为引用的最后一个列索引(3)不存在于：`to_column - 1` `to_column` `m`

```

//@version=5
indicator("Out of bounds demo")

//@variable A 2x3 matrix with a max row index of 1 and max column index of 2.
matrix<float> m = matrix.new<float>(2, 3, 0.0)

m.set(row = 2, column = 0, value = 1.0)      // The `row` index is out of bounds on this
line. The max value is 1.
m.submatrix(from_column = 1, to_column = 4) // The `to_column` index is invalid on this
line. The max value is 3.

if bar_index == last_bar_index - 1
    label.new(bar_index, 0, str.tostring(m), color = color.navy, textcolor =
color.white, size = size.huge)

```

用户可以通过确保其函数调用不引用大于或等于行/列数的索引来避免脚本中的此错误。

## 数组大小与矩阵中的行/列数不匹配。

当使用[matrix.add\\_row\(\)](#) 和[matrix.add\\_col\(\)](#) 函数将 行和列插入非空矩阵时，插入数组的大小必须与矩阵维度对齐。插入的行的大小必须与列的数量匹配，并且插入的列的大小必须与行的数量匹配。否则，脚本将引发此运行时错误。例如：

```

//@version=5
indicator("Invalid array size demo")

// Declare an empty matrix.
m = matrix.new<float>()

m.add_col(0, array.from(1, 2))      // Add a column. Changes the shape of `m` to 2x1.
m.add_col(1, array.from(1, 2, 3)) // Raises a runtime error because `m` has 2 rows, not
3.

plot(m.col(0).get(1))

```

- 注意：

当 `m` 为空时，可以插入任意大小的行或列数组，如第一 `m.add_col()` 行所示。

## 当矩阵的 ID 为“na”时，无法调用矩阵方法。

当矩阵变量被分配给 时 `na`，意味着该变量不引用现有对象。因此，不能使用内置 `matrix.*()` 函数和方法。例如：

```
//@version=5
indicator("na matrix methods demo")

//@variable A `matrix` variable assigned to `na`.
matrix<float> m = na

mCopy = m.copy() // Raises a runtime error. You can't copy a matrix that doesn't exist.

if bar_index == last_bar_index - 1
    label.new(bar_index, 0, str.tostring(mCopy), color = color.navy, textcolor =
color.white, size = size.huge)
```

要解决此错误，请在使用函数之前分配给有效的矩阵实例 `matrix.*()`。

## 矩阵太大。矩阵的最大大小为 100,000 个元素。

矩阵 (`matrix.elements_count()`) 中的元素总数不能超过**100,000**，无论其形状如何。例如，此脚本将引发错误，因为它将包含 101 个元素的 1000 行插入 `m` 到矩阵中：

```
//@version=5
indicator("Matrix too large demo")

var matrix<float> m = matrix.new<float>()

if bar_index == 0
    for i = 1 to 1000
        // This raises an error because the script adds 101 elements on each iteration.
        // 1000 rows * 101 elements per row = 101000 total elements. This is too large.
        m.add_row(m.rows(), array.new<float>(101, i))

plot(m.get(0, 0))
```

## 行/列索引必须为 0 <= from\_row/column < to\_row/column。

当使用带有和索引的 `matrix.*()` 函数时，这些值必须小于相应的值，最小可能值为 0。否则，脚本将引发运行时错误。`from_row/column``to_row/column``from_*``to_*`

例如，此脚本显示尝试从 4x4 矩阵声明一个值为 2 和值为 2 的子矩阵，这将导致错误：`m``from_row``to_row`

```

//@version=5
indicator("Invalid from_row, to_row demo")

//@variable A 4x4 matrix filled with a random value.
matrix<float> m = matrix.new<float>(4, 4, math.random())

matrix<float> mSub = m.submatrix(from_row = 2, to_row = 2) // Raises an error.
`from_row` can't equal `to_row`.

plot(mSub.get(0, 0))

```

## 矩阵“id1”和“id2”必须具有相同数量的要添加的行和列。

使用[matrix.sum\(\)](#) 和[matrix.diff\(\)](#) 函数时，`id1` 和 `id2` 矩阵必须具有相同的行数和相同的列数。尝试对两个维度不匹配的矩阵进行相加或相减将会引发错误，如以下代码所示：

```

//@version=5
indicator("Invalid sum dimensions demo")

//@variable A 2x3 matrix.
matrix<float> m1 = matrix.new<float>(2, 3, 1)
//@variable A 3x4 matrix.
matrix<float> m2 = matrix.new<float>(3, 4, 2)

mSum = matrix.sum(m1, m2) // Raises an error. `m1` and `m2` don't have matching
dimensions.

plot(mSum.get(0, 0))

```

## “id1”矩阵中的列数必须等于“id2”矩阵中的行数（或数组中的元素数）。

当使用[matrix.mult\(\)](#) `id1` 将矩阵乘以 `id2` 矩阵或数组时，[matrix.rows\(\)](#) 或 [array.size\(\)](#) 必须 `id2` 等于 [matrix.columns\(\)](#) `id1`。如果它们不对齐，脚本将引发此错误。

例如，此脚本尝试将两个 2x3 矩阵相乘。虽然可以将这些矩阵相加，但不能将它们相乘：

```
//@version=5
indicator("Invalid mult dimensions demo")

//@variable A 2x3 matrix.
matrix<float> m1 = matrix.new<float>(2, 3, 1)
//@variable A 2x3 matrix.
matrix<float> m2 = matrix.new<float>(2, 3, 2)

mSum = matrix.mult(m1, m2) // Raises an error. The number of columns in `m1` and rows
in `m2` aren't equal.

plot(mSum.get(0, 0))
```

## 运算不适用于非方阵。

一些矩阵运算，包括[matrix.inv\(\)](#)、[matrix.det\(\)](#)、[matrix.eigenvalues\(\)](#)和[matrix.eigenvectors\(\)](#)仅适用于方阵，即具有相同行数和列数的矩阵。当尝试在非方阵上执行此类函数时，脚本将引发错误，指出该操作不可用或无法计算矩阵的结果 `id`。例如：

```
//@version=5
indicator("Non-square demo")

//@variable A 3x5 matrix.
matrix<float> m = matrix.new<float>(3, 5, 1)

plot(m.det()) // Raises a runtime error. You can't calculate the determinant of a 3x5
matrix.
```

# 映射 Map

提示：

此页面包含高级材料。如果您是一名初级 Pine Script™ 程序员，我们建议您在冒险之前熟悉其他更易于使用的 Pine Script™ 功能。

## 介绍

Pine Script™ 映射是以键值对形式存储元素的集合。它们允许脚本收集与唯一标识符（键）关联的多个值引用。

与数组和矩阵不同，映射被视为无序集合。脚本通过引用放入其中的键值对中的键而不是遍历内部索引来快速访问映射的值。

映射的键可以是任何基本类型，其值可以是任何内置或[用户定义的](#)类型。映射不能直接使用其他集合（映射、数组或矩阵）作为值，但它们可以在其字段中保存包含这些数据结构的[UDT](#)实例。请参阅[本节](#)了解更多信息。

与其他集合一样，映射总共最多可以包含 100,000 个元素。由于映射中的每个键值对都由两个元素（唯一键及其关联值）组成，因此映射可以容纳的最大键值对数量为 50,000。

## 声明一个映射关系

Pine Script™ 使用以下语法来声明映射：

```
[var/varip ][map<keyType, valueType> ]<identifier> = <expression>
```

映射的[类型模板](#)在哪里，它声明它将包含的键和值的类型，并且返回映射实例或。`<keyType, valueType>``<expression>``na`

在声明分配给 的映射变量时 `na`，用户必须包含 `map`关键字，后跟 [类型模板](#)，以告诉编译器该变量可以接受带有 `keyType` 键和 `valueType` 值的映射。

例如，这行代码声明一个新 `myMap` 变量，它可以接受保存 [字符串](#)键和 [浮点](#)值对的映射实例：

```
map<string, float> myMap = na
```

当 `<expression>` 不是时 `na`，编译器不需要显式类型声明，因为它将从分配的映射对象推断类型信息。

此行声明一个 `myMap` 分配给带有 [字符串](#)键和 [浮点](#)值的空映射的变量。稍后分配给该变量的任何映射都必须具有相同的键和值类型：

```
myMap = map.new<string, float>()
```

## 使用 `var` 和 `varip` 关键字

用户可以包含`var`或`varip`关键字来指示其脚本仅在第一个图表栏上声明映射变量。使用这些关键字的变量在每次脚本迭代中都指向相同的映射实例，直到显式重新分配为止。

例如，此脚本声明一个 `colorMap` 分配给映射的变量，该映射保存 第一个图表栏上的[字符串](#)键和 [颜色](#)值对。该脚本在图表上显示，并使用它放入第一个条形图上的值来 `oscillator` 为 所有条形图上的图着色：`colorMap`



```

//@version=5
indicator("var map demo")

//@variable A map associating color values with string keys.
var colorMap = map.new<string, color>()

// Put `<string, color>` pairs into `colorMap` on the first bar.
if bar_index == 0
    colorMap.put("Bull", color.green)
    colorMap.put("Bear", color.red)
    colorMap.put("Neutral", color.gray)

//@variable The 14-bar RSI of `close`.
float oscillator = ta.rsi(close, 14)

//@variable The color of the `oscillator`.
color oscColor = switch
    oscillator > 50 => colorMap.get("Bull")
    oscillator < 50 => colorMap.get("Bear")
    => colorMap.get("Neutral")

// Plot the `oscillator` using the `oscColor` from our `colorMap`.
plot(oscillator, "Histogram", oscColor, 2, plot.style_histogram, histbase = 50)
plot(oscillator, "Line", oscColor, 3)

```

## 笔记

使用[varip声明的映射变量的行为与在历史数据上使用var的](#)行为相同，但它们会在每个新的价格变动时更新实时柱（即自脚本上次编译以来的柱）的键值对。分配给[varip](#)变量的映射只能保存[int](#)、[float](#)、[bool](#)、[color](#)或[string](#)类型或[用户定义类型](#)的值，这些类型在其字段中专门包含这些类型或这些类型的集合（[array](#)、[matrices](#)或[Map](#)）。

# 读写

## 放置和获取键值对

`map.put()`函数是映射用户经常使用的函数，因为它是将新的键值对放入映射的主要方法。它将`key`参数与`value`调用中的参数相关联，并将该对添加到映射中`id`。

如果`map.put()` key 调用中的参数已经存在于映射的`keys`中，则传递到函数中的新对将替换现有的对。

`id`要从与给定关联的映射中检索值`key`，请使用`map.get()`。如果`id`映射包含`key`，否则，它返回`na`。

以下示例借助`map.put()` 和`map.get()`方法计算给定值上一次收盘价上涨和下跌时的`bar_index`值之间的差异。当价格上涨时，脚本将一对放入映射中；当价格下跌时，脚本将一对放入映射中。然后，它将包含“上升”和“下降”值之间的“差异”的一对放入映射中，并将其值绘制在图表上：`length``("Rising", bar_index)` `data``("Falling", bar_index)`



```
//@version=5
indicator("Putting and getting demo")

//@variable The length of the `ta.rising()` and `ta.falling()` calculation.
int length = input.int(2, "Length")

//@variable A map associating `string` keys with `int` values.
var data = map.new<string, int>()

// Put a new ("Rising", `bar_index`) pair into the `data` map when `close` is rising.
if ta.rising(close, length)
    data.put("Rising", bar_index)

// Put a new ("Falling", `bar_index`) pair into the `data` map when `close` is falling.
if ta.falling(close, length)
    data.put("Falling", bar_index)

// Get the difference between the last "Rising" and "Falling" bar indices.
float diff = data.get("Rising") - data.get("Falling")
plot(diff, "Difference", color=blue, linewidth=2)
```

```

// Put a new ("Falling", `bar_index`) pair into the `data` map when `close` is falling.
if ta.falling(close, length)
    data.put("Falling", bar_index)

// Put the "Difference" between current "Rising" and "Falling" values into the `data` map.
data.put("Difference", data.get("Rising") - data.get("Falling"))

//@variable The difference between the last "Rising" and "Falling" `bar_index`.
int index = data.get("Difference")

//@variable Returns `color.green` when `index` is positive, `color.red` when negative, and `color.gray` otherwise.
color indexColor = index > 0 ? color.green : index < 0 ? color.red : color.gray

plot(index, color = indexColor, style = plot.style_columns)

```

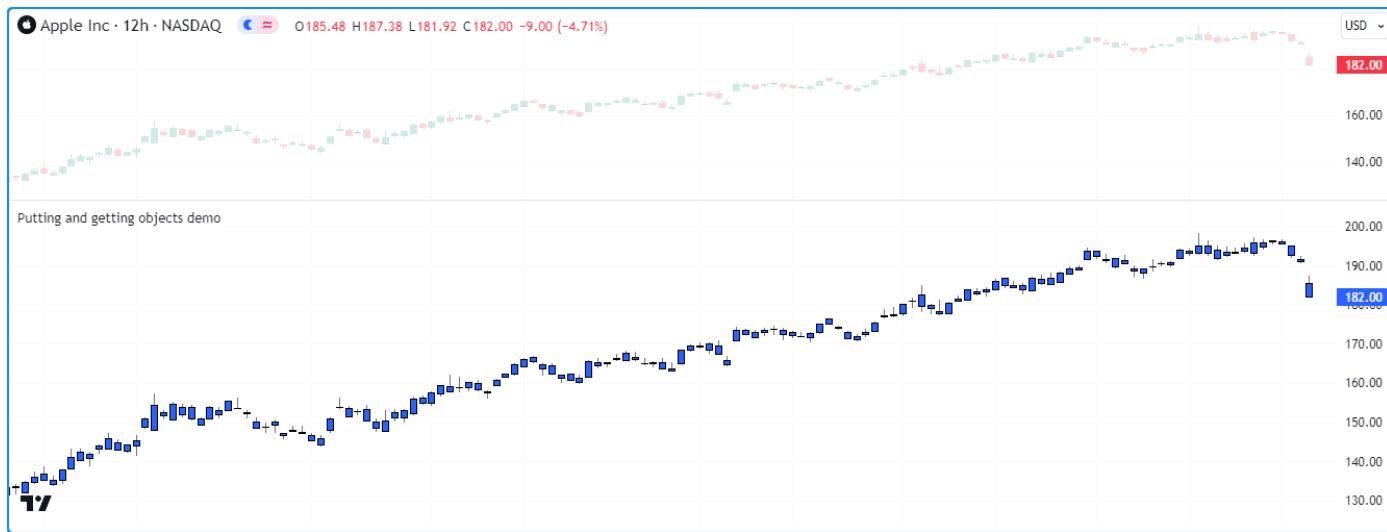
- 注意：

[此脚本替换连续](#)`data.put()`调用中与“Rising”、“Falling”和“Difference”键关联的值，因为每个键都是唯一的，并且只能在映射中出现一次`data`。替换映射中的对不会更改其键的内部插入顺序。我们将在[下一节](#)中进一步讨论这一点。

与使用其他集合类似，将特殊类型（[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)或[Chart.point](#)）或[用户定义类型的](#)值放入映射时，重要的是要注意插入的对的`value`指向同一个对象而不复制它。修改键值对引用的值也会影响原始对象。

例如，此脚本包含`ChartData`带有`o`、`h`、`l`和`c`字段的自定义类型。在第一个图表栏上，脚本声明一个`myMap`变量并添加对，其中是初始字段值为的实例。它通过用户定义的方法在每个柱上添加该对并更新该对的对象。`("A", myData)``myData``ChartData``na``("B", myData)``myMap``update()`

使用“B”键对对象进行的每次更改都会影响“A”键引用的对象，如“A”对象字段的蜡烛图所示：



```

//@version=5
indicator("Putting and getting objects demo")

```

```

//@type A custom type to hold OHLC data.
type ChartData
    float o
    float h
    float l
    float c

//@function Updates the fields of a `ChartData` object.
method update(ChartData this) =>
    this.o := open
    this.h := high
    this.l := low
    this.c := close

//@variable A new `ChartData` instance declared on the first bar.
var myData = ChartData.new()
//@variable A map associating `string` keys with `ChartData` instances.
var myMap = map.new<string, ChartData>()

// Put a new pair with the "A" key into `myMap` only on the first bar.
if bar_index == 0
    myMap.put("A", myData)

// Put a pair with the "B" key into `myMap` on every bar.
myMap.put("B", myData)

//@variable The `ChartData` value associated with the "A" key in `myMap`.
ChartData oldest = myMap.get("A")
//@variable The `ChartData` value associated with the "B" key in `myMap`.
ChartData newest = myMap.get("B")

// Update `newest`. Also affects `oldest` and `myData` since they all reference the
// same `ChartData` object.
newest.update()

// Plot the fields of `oldest` as candles.
plotcandle(oldest.o, oldest.h, oldest.l, oldest.c)

```

- 注意：

如果该脚本将 的副本传递 `myData` 到每个 `myMap.put()` 调用中，则其行为会有所不同。有关更多信息，请参阅我们的用户手册的[对象](#)页面的[这一部分](#)。

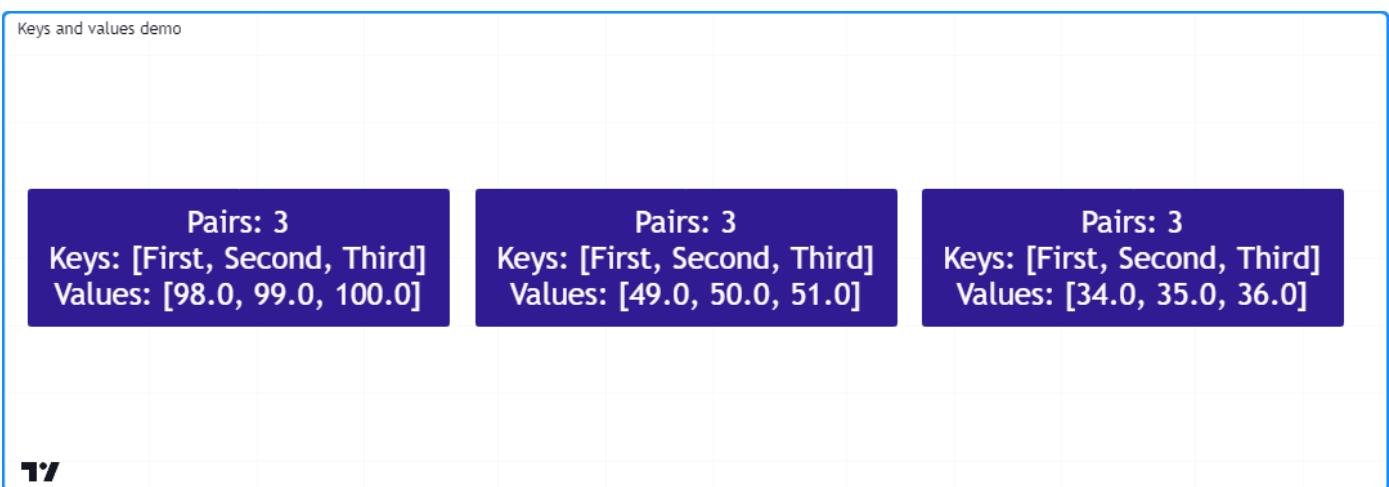
## 检查键和值

### `map.keys()` 和 `map.values()`

要检索放入映射中的所有键和值，请使用 `map.keys()` 和 `map.values()`。这些函数将映射中的所有键/值引用复制到新的 [数组](#) 对象。修改从这两个函数返回的数组不会影响映射 `id`。

尽管映射是无序集合，但 Pine Script™ 在内部维护映射键值对的插入顺序。因此，`map.keys()` 和 `map.values()` 函数始终返回 [数组](#)，其元素根据 `id` 映射的插入顺序进行排序。

下面的脚本通过每 50 个条 `m` 在标签中 [显示](#) 一次映射中的键和值数组来演示这一点。正如我们在图表中看到的，`m.keys()` 和返回的每个数组中元素的顺序 `m.values()` 与中键值对的插入顺序一致 `m`：



```
//@version=5
indicator("Keys and values demo")

if bar_index % 50 == 0
    // @variable A map containing pairs of `string` keys and `float` values.
    m = map.new<string, float>()

    // Put pairs into `m`. The map will maintain this insertion order.
    m.put("First", math.round(math.random(0, 100)))
    m.put("Second", m.get("First") + 1)
    m.put("Third", m.get("Second") + 1)

    // @variable An array containing the keys of `m` in their insertion order.
    array<string> keys = m.keys()
    // @variable An array containing the values of `m` in their insertion order.
    array<float> values = m.values()

    // @variable A label displaying the `size` of `m` and the `keys` and `values` arrays.
    label debugLabel = label.new(
        bar_index, 0,
        str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys, values),
    )
```

```

        color = color.navy, style = label.style_label_center,
        textcolor = color.white, size = size.huge
    )

```

- 注意：

“First”键的值是0到100之间的随机整数。“Second”值比“First”大1，“Third”值比“Second”大1。

需要注意的是，替换映射的键值对时，映射的内部插入顺序不会改变。在这种情况下，`keys()`和`values()`数组中新元素的位置将与旧元素相同。唯一的例外是脚本事先完全删除了密钥。

下面，我们添加了一行代码，将带有“Second”键的新值放入映射中，`m`覆盖与该键关联的先前值。尽管脚本将这个新对放入带有“第三”键的映射之后，但该对的键和值仍然位于`keys`和数组中的第二位，因为该键在更改之前`values`已经存在：`m`

The screenshot shows a "Keys and values demo" interface. It displays three cards, each containing a summary of key-value pairs:

- Pairs: 3**  
Keys: [First, Second, Third]  
Values: [12.0, -2.0, 14.0]
- Pairs: 3**  
Keys: [First, Second, Third]  
Values: [32.0, -2.0, 34.0]
- Pairs: 3**  
Keys: [First, Second, Third]  
Values: [55.0, -2.0, 57.0]

TY

```

//@version=5
indicator("Keys and values demo")

if bar_index % 50 == 0
    // @variable A map containing pairs of `string` keys and `float` values.
    m = map.new<string, float>()

    // Put pairs into `m`. The map will maintain this insertion order.
    m.put("First", math.round(math.random(0, 100)))
    m.put("Second", m.get("First") + 1)
    m.put("Third", m.get("Second") + 1)

    // Overwrite the "Second" pair in `m`. This will NOT affect the insertion order.
    // The key and value will still appear second in the `keys` and `values` arrays.
    m.put("Second", -2)

    // @variable An array containing the keys of `m` in their insertion order.
    array<string> keys = m.keys()
    // @variable An array containing the values of `m` in their insertion order.
    array<float> values = m.values()

```

```

//@variable A label displaying the `size` of `m` and the `keys` and `values`
arrays.

label debugLabel = label.new(
    bar_index, 0,
    str.format("Pairs: {0}\nKeys: {1}\nValues: {2}", m.size(), keys, values),
    color = color.navy, style = label.style_label_center,
    textcolor = color.white, size = size.huge
)

```

## 笔记

[map.values\(\)](#)数组中的元素 指向与 map 相同的值 `id`。因此，当映射的值是引用类型（包括[line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)、[Chart.point](#)或[UDTs](#)）时，修改[map.values\(\)](#) 数组引用的实例也会影响映射 `id`，因为两个集合的内容都指向相同的对象。

## [map.contains\(\)](#)

`key` 要检查map 中是否存在特定内容 `id`，请使用 [map.contains\(\)](#)。此函数是对从 [map.keys\(\)](#)返回的数组调用 `array.includes ()`的便捷替代方法。

例如，此脚本检查映射中是否存在各种键，然后在[标签](#) `m` 中显示结果：

Inspecting keys demo

Tested keys: [A, B, C, D, E, F]  
Keys found: [A, C, E]



```

//@version=5
indicator("Inspecting keys demo")

//@variable A map containing `string` keys and `string` values.
m = map.new<string, string>()

// Put key-value pairs into the map.
m.put("A", "B")
m.put("C", "D")
m.put("E", "F")

//@variable An array of keys to check for in `m`.
array<string> testKeys = array.from("A", "B", "C", "D", "E", "F")

//@variable An array containing all elements from `testKeys` found in the keys of `m`.

```

```

array<string> mappedKeys = array.new<string>()

for key in testKeys
    // Add the `key` to `mappedKeys` if `m` contains it.
    if m.contains(key)
        mappedKeys.push(key)

//@variable A string representing the `testKeys` array and the elements found within
//the keys of `m`.
string testText = str.format("Tested keys: {0}\nKeys found: {1}", testKeys, mappedKeys)

if bar_index == last_bar_index - 1
    //@variable Displays the `testText` in a label at the `bar_index` before the last.
    label debugLabel = label.new(
        bar_index, 0, testText, style = label.style_label_center,
        textColor = color.white, size = size.huge
    )

```

## 删除键值对

要从映射中删除特定的键值对 `id`，请使用 `map.remove()`。此函数 `key` 从映射中删除及其关联值，同时保留其他键值对的插入顺序。如果映射 `包含` `key`，否则，它返回 `na`。

`id` 要一次从映射中删除所有键值对，请使用 `map.clear()`。

以下脚本创建一个新 `m` 映射，将 键值对放入映射中，在循环中使用 `m.remove()` 删除数组 `key` 中列出的每个有效键 `removeKeys`，然后调用 `m.clear()` 删除所有剩余的键值对。它使用自定义 `debugLabel()` 方法来显示每次更改后的 `size`、`keys` 和 `value[m]`：

The screenshot shows a script editor window with the title "Removing key-value pairs demo". Below the title, there is a table with three columns. The first column contains the status message "Removed key-value pairs demo". The second column, which has a green background, displays the state after adding pairs: "Added pairs", "Size: 5", "Keys: [A, B, C, D, E]", and "Values: [0, 1, 2, 3, 4]". The third column, which has a red background, displays the state after removing pairs: "Removed pairs", "Size: 3", "Keys: [A, C, E]", and "Values: [0, 2, 4]". The final column, which has a purple background, displays the state after clearing the map: "Cleared the map", "Size: 0", "Keys: []", and "Values: []". At the bottom left of the editor, there is a small icon with the letters "TV".

```

//@version=5
indicator("Removing key-value pairs demo")

//@function Returns a label to display the keys and values from a map.
method label debugLabel(
    map<string, int> this, int barIndex = bar_index,
    color bgColor = color.blue, string note = ""
) =>

```

```

//@variable A string representing the size, keys, and values in `this` map.
string repr = str.format(
    "{0}\nSize: {1}\nKeys: {2}\nValues: {3}",
    note, this.size(), str.tostring(this.keys()), str.tostring(this.values())
)
label.new(
    barIndex, 0, repr, color = bgColor, style = label.style_label_center,
    textcolor = color.white, size = size.huge
)

if bar_index == last_bar_index - 1
    // @variable A map containing `string` keys and `int` values.
    m = map.new<string, int>()

    // Put key-value pairs into `m`.
    for [i, key] in array.from("A", "B", "C", "D", "E")
        m.put(key, i)
    m.debugLine(bar_index, color.green, "Added pairs")

    // @variable An array of keys to remove from `m`.
    array<string> removeKeys = array.from("B", "B", "D", "F", "a")

    // Remove each `key` in `removeKeys` from `m`.
    for key in removeKeys
        m.remove(key)
    m.debugLine(bar_index + 10, color.red, "Removed pairs")

    // Remove all remaining keys from `m`.
    m.clear()
    m.debugLine(bar_index + 20, color.purple, "Cleared the map")

```

- 注意：

并非数组中的所有字符串都 `removeKeys` 出现在 键中 `m`。尝试删除不存在的键（本例中的“F”、“a”和第二个“B”）不会影响映射的内容。

## 组合映射

脚本可以通过 `map.put_all()` 组合两个映射。该函数将映射中的所有键值 `id2` 对按照插入顺序放入 `id1` 映射中。与 `map.put()` `id2` 一样，如果 中 中也存在任何键 `id1`，则此函数将替换包含这些键的键值对，而不影响它们的初始插入顺序。

此示例包含一个用户定义的 `hexMap()` 函数，该函数将十进制 `int` 键映射到 其 十六进制形式的字符串 表示形式。该脚本使用此函数创建两个映射 和，然后使用 `mapA.put_all(mapB)` 将所有键值对放入 中。`mapA``mapB``mapB``mapA`

该脚本使用自定义 `debugLabel()` 函数来显示 显示 和 的键和 值 的标签，然后在将所有键值对放入其中后显示另一个标签 显示 的内容：`mapA``mapB``mapA``mapB`

A  
Decimal: [101, 202, 303, 404]  
Hex: [65, CA, 12F, 194]

B  
Decimal: [303, 404, 505, 606, 707, 808]  
Hex: [12F, 194, 1F9, 25E, 2C3, 328]

Merge B into A  
Decimal: [101, 202, 303, 404, 505, 606, 707, 808]  
Hex: [65, CA, 12F, 194, 1F9, 25E, 2C3, 328]

77

```
//@version=5
indicator("Combining maps demo", "Hex map")

//@variable An array of string hex digits.
var array<string> hexDigits = str.split("0123456789ABCDEF", " ")

//@function Returns a hexadecimal string for the specified `value`.
hex(int value) =>
    //@variable A string representing the hex form of the `value`.
    string result = ""
    //@variable A temporary value for digit calculation.
    int tempValue = value
    while tempValue > 0
        //@variable The next integer digit.
        int digit = tempValue % 16
        // Add the hex form of the `digit` to the `result`.
        result := hexDigits.get(digit) + result
        // Divide the `tempValue` by the base.
        tempValue := int(tempValue / 16)
    result

//@function Returns a map holding the `numbers` as keys and their `hex` strings as
values.
hexMap(array<int> numbers) =>
    //@variable A map associating `int` keys with `string` values.
    result = map.new<int, string>()
    for number in numbers
        // Put a pair containing the `number` and its `hex()` representation into the
`result`.
        result.put(number, hex(number))
    result

//@function Returns a label to display the keys and values of a hex map.
debugLabel(
    map<int, string> this, int barIndex = bar_index, color bgColor = color.blue,
    string style = label.style_label_center, string note = ""
) =>
    string repr = str.format(
```

```

    "{0}\nDecimal: {1}\nHex: {2}",
    note, str.toString(this.keys()), str.toString(this.values())
)
label.new(
    barIndex, 0, repr, color = bgColor, style = style,
    textcolor = color.white, size = size.huge
)

if bar_index == last_bar_index - 1
    // @variable A map with decimal `int` keys and hexadecimal `string` values.
    map<int, string> mapA = hexMap(array.from(101, 202, 303, 404))
    debugLabel(mapA, bar_index, color.navy, label.style_label_down, "A")

    // @variable A map containing key-value pairs to add to `mapA`.
    map<int, string> mapB = hexMap(array.from(303, 404, 505, 606, 707, 808))
    debugLabel(mapB, bar_index, color.maroon, label.style_label_up, "B")

    // Put all pairs from `mapB` into `mapA`.
    mapA.put_all(mapB)
    debugLabel(mapA, bar_index + 10, color.purple, note = "Merge B into A")

```

## 循环遍历映射

脚本可以通过多种方式迭代访问映射中的键和值。例如，可以循环遍历映射的 `keys()` 数组并 `get()` 每个的值 `key`，如下所示：

```

for key in thisMap.keys()
    value = thisMap.get(key)

```

但是，我们建议 `for...in` 直接在映射上使用循环，因为它按插入顺序迭代映射的键值对，并在每次迭代时返回包含下一对的键和值的元组。

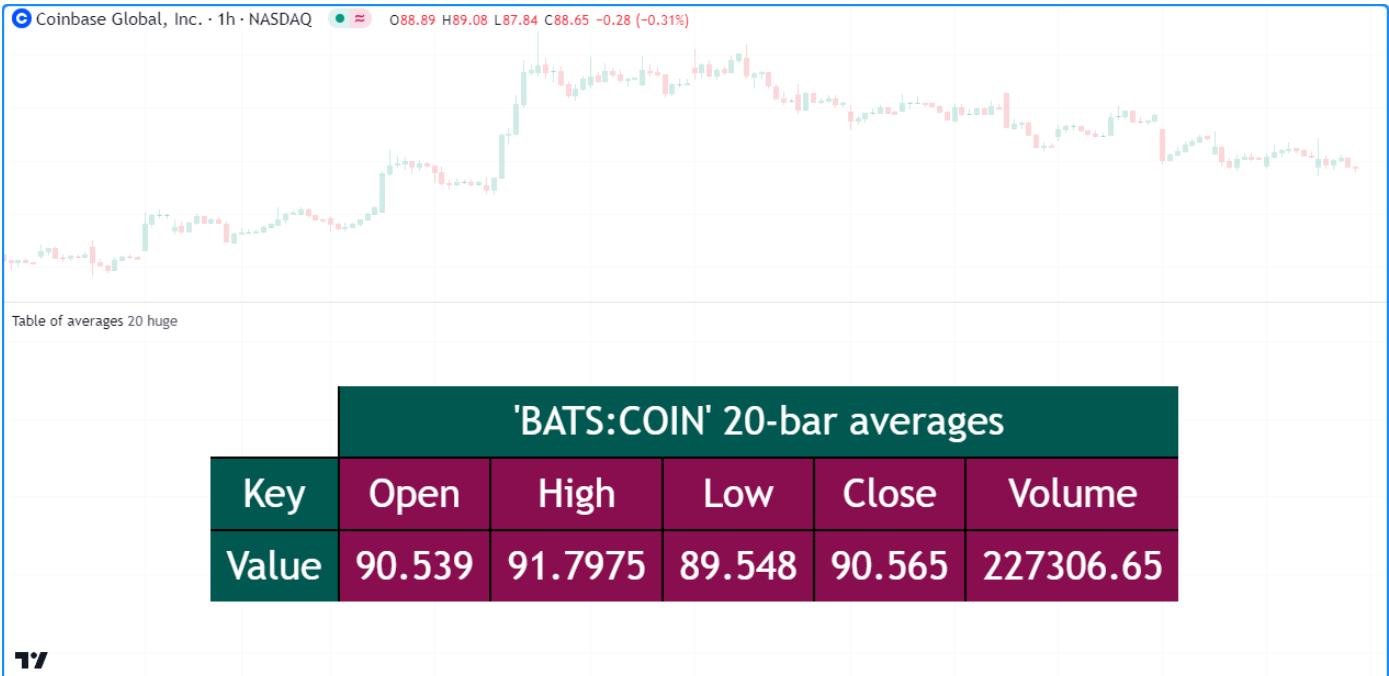
例如，这行代码从放入其中的第一个键值对开始循环遍历 `every key 和 value in : thisMap`

```

for [key, value] in thisMap

```

[让我们使用此结构编写一个脚本，在表中显示映射的键值对](#)。在下面的示例中，我们定义了一个自定义 `toTable()` 方法来创建 `table`，然后使用 `for...in` 循环迭代映射的键值对并填充表的单元格。该脚本使用此方法来可视化包含价格和交易量数据的 `length-bar` 的映射 `averages`：



```
//@version=5
indicator("Looping through a map demo", "Table of averages")

//@variable The length of the moving average.
int length = input.int(20, "Length")
//@variable The size of the table text.
string txtSize = input.string(
    size.huge, "Text size",
    options = [size.auto, size.tiny, size.small, size.normal, size.large, size.huge]
)

//@function Displays the pairs of `this` map within a table.
//@param this A map with `string` keys and `float` values.
//@param position The position of the table on the chart.
//@param header The string to display on the top row of the table.
//@param textSize The size of the text in the table.
//@returns A new `table` object with cells displaying each pair in `this`.
method table toTable(
    map<string, float> this, string position = position.middle_center, string header =
na,
    string textSize = size.huge
) =>
    // Color variables
    borderColor = #000000
    headerColor = color.rgb(1, 88, 80)
    pairColor = color.maroon
    textColor = color.white

    //@variable A table that displays the key-value pairs of `this` map.
    table result = table.new(
        position, this.size() + 1, 3, border_width = 2, border_color = borderColor
    )
    for (string key, float value) in this
        result.cell(key, 1).text(key)
        result.cell(value, 2).text(value)
        result.cell(pairColor, 3).text("")
    end
    return result
)
```

```

    )
    // Initialize top and side header cells.
    result.cell(1, 0, header, bgcolor = headerColor, text_color = textColor, text_size
= textSize)
    result.merge_cells(1, 0, this.size(), 0)
    result.cell(0, 1, "Key", bgcolor = headerColor, text_color = textColor, text_size =
textSize)
    result.cell(0, 2, "Value", bgcolor = headerColor, text_color = textColor, text_size
= textSize)

    //@variable The column index of the table. Updates on each loop iteration.
    int col = 1

    // Loop over each `key` and `value` from `this` map in the insertion order.
    for [key, value] in this
        // Initialize a `key` cell in the `result` table on row 1.
        result.cell(
            col, 1, str.tostring(key), bgcolor = color.maroon,
            text_color = color.white, text_size = textSize
        )
        // Initialize a `value` cell in the `result` table on row 2.
        result.cell(
            col, 2, str.tostring(value), bgcolor = color.maroon,
            text_color = color.white, text_size = textSize
        )
        // Move to the next column index.
        col += 1
    result // Return the `result` table.

    //@variable A map with `string` keys and `float` values to hold `length`-bar averages.
    averages = map.new<string, float>()

    // Put key-value pairs into the `averages` map.
    averages.put("Open", ta.sma(open, length))
    averages.put("High", ta.sma(high, length))
    averages.put("Low", ta.sma(low, length))
    averages.put("Close", ta.sma(close, length))
    averages.put("Volume", ta.sma(volume, length))

    //@variable The text to display at the top of the table.
    string headerText = str.format("{0} {1}-bar averages", "" + syminfo.tickerid + "", length)
    // Display the `averages` map in a `table` with the `headerText`.
    averages.toTable(header = headerText, textSize = txtSize)

```

## 复制映射

## 浅拷贝

脚本可以使用 `map.copy()` 函数制作映射的浅表副本。对浅拷贝的修改不会影响原始映射或其内部插入顺序。`id``id`

例如，此脚本构建一个 `m` 映射，其中将键“A”、“B”、“C”和“D”分配给 0 到 10 之间的四个随机 `mCopy` 值。然后，它创建一个映射作为 的浅表副本 `m` 并更新与其键关联的值。该脚本使用我们的自定义方法在图表中 `m` 和 `mCopy` 图表上显示键值对 `debugLabel()`：



```
//@version=5
indicator("Shallow copy demo")

//@function Displays the key-value pairs of `this` map in a label.
method debugLabel(
    map<string, float> this, int barIndex = bar_index, color bgColor = color.blue,
    color textColor = color.white, string note = ""
) =>
    //@variable The text to display in the label.
    labelText = note + "\n{"
    for [key, value] in this
        labelText += str.format("{0}: {1}, ", key, value)
    labelText := str.replace(labelText, ", ", "}", this.size() - 1)

    if barstate.ishistory
        label result = label.new(
            barIndex, 0, labelText, color = bgColor, style = label.style_label_center,
            textcolor = textColor, size = size.huge
        )

    if bar_index == last_bar_index - 1
        //@variable A map of `string` keys and random `float` values.
        m = map.new<string, float>()

        // Assign random values to an array of keys in `m`.
        for key in array.from("A", "B", "C", "D")
            m.put(key, math.random(0, 10))
```

```

//@variable A shallow copy of `m`.
mCopy = m.copy()

// Assign the insertion order value `i` to each `key` in `mCopy`.
for [i, key] in mCopy.keys()
    mCopy.put(key, i)

// Display the labels.
m.debugLineLabel(bar_index, note = "Original")
mCopy.debugLineLabel(bar_index + 10, color.purple, note = "Copied and changed")

```

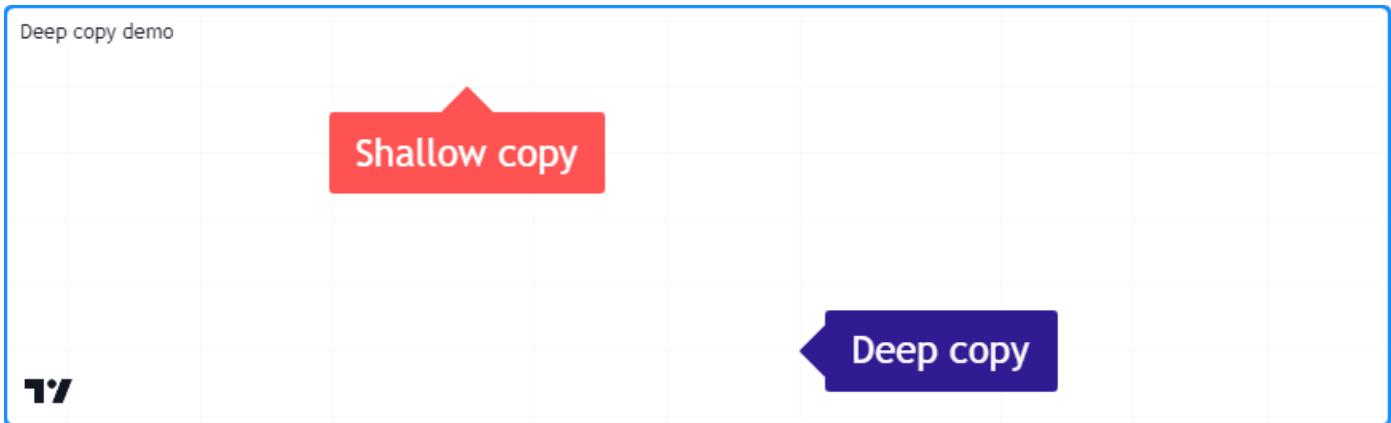
## 深拷贝

虽然在复制具有 基本类型值的映射时浅复制就足够了，但请务必记住，保存引用类型值 ([line](#)、[linefill](#)、[box](#)、[polyline](#)、[label](#)、[table](#)、[Chart.point](#) 或 UDT) 指向与原始对象相同的对象。修改浅拷贝引用的对象将影响原始映射引用的实例，反之亦然。

为了确保对复制映射引用的对象所做的更改不会影响其他位置引用的实例，可以通过创建一个新映射来进行深层复制，该新映射的键值对包含原始映射中每个值的副本。

此示例创建字符串键和标签值 `original` 的映射，并将键值对放入其中。该脚本通过内置的[copy\(\)](#)方法将映射复制到变量，然后使用自定义方法复制到变量。`shallow``deep``deepCopy()`

正如我们从图表中看到的，对从副本中检索到的标签的更改 `shallow` 也会影响映射引用的实例 `original`，但对副本中的标签的更改 `deep` 不会：



```

//@version=5
indicator("Deep copy demo")

//@function Returns a deep copy of `this` map.
method deepCopy(map<string, label> this) =>
    // @variable A deep copy of `this` map.
    result = map.new<string, label>()
    // Add key-value pairs with copies of each `value` to the `result`.
    for [key, value] in this
        result.put(key, value.copy())

```

```

result //Return the `result`.

//@variable A map containing `string` keys and `label` values.
var original = map.new<string, label>()

if bar_index == last_bar_index - 1
    // Put a new key-value pair into the `original` map.
    map.put(
        original, "Test",
        label.new(bar_index, 0, "Original", textcolor = color.white, size = size.huge)
    )

    // @variable A shallow copy of the `original` map.
    map<string, label> shallow = original.copy()
    // @variable A deep copy of the `original` map.
    map<string, label> deep = original.deepCopy()

    // @variable The "Test" label from the `shallow` copy.
    label shallowLabel = shallow.get("Test")
    // @variable The "Test" label from the `deep` copy.
    label deepLabel = deep.get("Test")

    // Modify the "Test" label's `y` attribute in the `original` map.
    // This also affects the `shallowLabel`.
    original.get("Test").set_y(label.all.size())

    // Modify the `shallowLabel`. Also modifies the "Test" label in the `original` map.
    shallowLabel.set_text("Shallow copy")
    shallowLabel.set_color(color.red)
    shallowLabel.set_style(label.style_label_up)

    // Modify the `deepLabel`. Does not modify any other label instance.
    deepLabel.set_text("Deep copy")
    deepLabel.set_color(color.navy)
    deepLabel.set_style(label.style_label_left)
    deepLabel.set_x(bar_index + 5)

```

- 注意：

该 `deepCopy()` 方法循环遍历 `original` 映射，复制每个映射 `value` 并将 包含副本的键值对放入新的 映射实例中。

## 范围和历史

[与 Pine 中的其他集合一样，映射变量在每个条上留下历史轨迹，允许脚本使用历史引用运算符]访问分配给变量的过去映射实例。脚本还可以将映射分配给全局变量，并在函数、方法和条件结构的范围内与它们交互。

例如，此脚本使用全局映射及其历史记录来计算 EMA的聚合集。它声明了一个int键和 float globalData 值 的映射，其中映射中的每个键对应于每个 EMA 计算的长度。用户定义的函数通过将分配给的映射中的值与当前值混合来计算每个-length EMA 。 update(`key` `previous` `globalData` `source`)

该脚本绘制全局映射的 value()数组中的最大值和 最小值以及来自 (即 50 条 EMA) 的值： globalData.get(50)



```
//@version=5
indicator("Scope and history demo", overlay = true)

//@variable The source value for EMA calculation.
float source = input.source(close, "Source")

//@variable A map containing global key-value pairs.
globalData = map.new<int, float>()

//@function Calculates a set of EMAs and updates the key-value pairs in `globalData`.
update() =>
    //@variable The previous map instance assigned to `globalData`.
    map<int, float> previous = globalData[1]

    // Put key-value pairs with keys 10-200 into `globalData` if `previous` is `na`.
    if na(previous)
        for i = 10 to 200
            globalData.put(i, source)
    else
        // Iterate each `key` and `value` in the `previous` map.
        for [key, value] in previous
            //@variable The smoothing parameter for the `key`-length EMA.
            float alpha = 2.0 / (key + 1.0)
            //@variable The `key`-length EMA value.
            float ema = (1.0 - alpha) * value + alpha * source
            // Put the `key`-length `ema` into the `globalData` map.
            globalData.put(key, ema)

    // Update the `globalData` map.
```

```

update()

//@variable The array of values from `globalData` in their insertion order.
array<float> values = globalData.values()

// Plot the max EMA, min EMA, and 50-bar EMA values.
plot(values.max(), "Max EMA", color.green, 2)
plot(values.min(), "Min EMA", color.red, 2)
plot(globalData.get(50), "50-bar EMA", color.orange, 3)

```

## 其他集合的映射

映射不能直接使用其他映射、[数组](#)或[矩阵](#)作为值，但它们可以保存在其字段中包含集合的[用户定义类型](#)的值。

例如，假设我们要创建一个“2D”映射，它使用[字符串](#)键来访问包含[字符串键和浮点值对的嵌套映射](#)。由于映射不能使用其他集合作为值，因此我们首先创建一个包装类型，其中包含一个字段来保存实例，如下所示：

```

//@type A wrapper type for maps with `string` keys and `float` values.
type Wrapper
    map<string, float> data

```

定义类型后 `Wrapper`，我们可以创建[字符串](#)键和 `Wrapper` 值的映射，其中 `data` 映射中每个值的字段都指向一个实例：

```
mapOfMaps = map.new<string, Wrapper>()
```

下面的脚本使用此概念构建一个映射，其中包含保存多个代码请求的 OHLCV 数据的映射。用户定义的 `requestData()` 函数从股票代码请求价格和交易量数据，创建映射，将数据放入其中，然后返回包含新映射的实例。`<string, float>``Wrapper`

该脚本将每次调用的结果 `requestData()` 放入中 `mapOfMaps`，然后使用用户定义的方法创建嵌套映射的[字符串](#) `toString()` 表示形式，并将其显示在图表上的 `label` 中：

Nested map demo D EURUSD GBPUSD EURGBP

```
{FX:EURUSD: {Open: 1.09491, High: 1.10419, Low: 1.09332, Close: 1.10088, Volume: 267301},
FX:GBPUSD: {Open: 1.27082, High: 1.27921, Low: 1.2685, Close: 1.27503, Volume: 314092},
FX:EURGBP: {Open: 0.8615, High: 0.86478, Low: 0.8601, Close: 0.86337, Volume: 193472}}
```

17

```
//@version=5
indicator("Nested map demo")
```

```

//@variable The timeframe of the requested data.
string tf = input.timeframe("D", "Timeframe")
// Symbol inputs.
string symbol1 = input.symbol("EURUSD", "Symbol 1")
string symbol2 = input.symbol("GBPUSD", "Symbol 2")
string symbol3 = input.symbol("EURGBP", "Symbol 3")

//@type A wrapper type for maps with `string` keys and `float` values.
type Wrapper
    map<string, float> data

//@function Returns a wrapped map containing OHLCV data from the `tickerID` at the
`timeframe`.
requestData(string tickerID, string timeframe) =>
    // Request a tuple of OHLCV values from the specified ticker and timeframe.
    [o, h, l, c, v] = request.security(
        tickerID, timeframe,
        [open, high, low, close, volume]
    )
    //@variable A map containing requested OHLCV data.
    result = map.new<string, float>()
    // Put key-value pairs into the `result`.
    result.put("Open", o)
    result.put("High", h)
    result.put("Low", l)
    result.put("Close", c)
    result.put("Volume", v)
    //Return the wrapped `result`.
    Wrapper.new(result)

//@function Returns a string representing `this` map of `string` keys and `Wrapper`
values.
method toString(map<string, Wrapper> this) =>
    //@variable A string representation of `this` map.
    string result = "{"

    // Iterate over each `key1` and associated `wrapper` in `this`.
    for [key1, wrapper] in this
        // Add `key1` to the `result`.
        result += key1

        //@variable A string representation of the `wrapper.data` map.
        string innerStr = ": {"
        // Iterate over each `key2` and associated `value` in the wrapped map.
        for [key2, value] in wrapper.data
            // Add the key-value pair's representation to `innerStr`.
            innerStr += str.format("{0}: {1}, ", key2, str.tostring(value))

```

```

// Replace the end of `innerStr` with ")" and add to `result`.
result += str.replace(innerStr, ", ", "},\n", wrapper.data.size() - 1)

// Replace the blank line at the end of `result` with ")".
result := str.replace(result, ",\n", "}", this.size() - 1)
result

//@variable A map of wrapped maps containing OHLCV data from multiple tickers.
var mapOfMaps = map.new<string, Wrapper>()

//@variable A label showing the contents of the `mapOfMaps`.
var debugLabel = label.new(
    bar_index, 0, color = color.navy, textcolor = color.white, size = size.huge,
    style = label.style_label_center, text_font_family = font.family_monospace
)

// Put wrapped maps into `mapOfMaps`.
mapOfMaps.put(symbol1, requestData(symbol1, tf))
mapOfMaps.put(symbol2, requestData(symbol2, tf))
mapOfMaps.put(symbol3, requestData(symbol3, tf))

// Update the label.
debugLabel.set_text(mapOfMaps.toString())
debugLabel.set_x(bar_index)

```

## 警报

### 介绍

TradingView 警报在我们的服务器上 24x7 运行，不需要用户登录即可执行。警报是从图表用户界面 (*UI*) 创建的。您将在帮助中心的[关于 TradingView 警报](#)页面的图表 UI 中找到了解警报如何工作以及如何创建警报所需的所有信息。

TradingView 上提供的一些警报类型（通用警报、绘图警报和订单执行事件脚本警报）是根据图表上加载的交易品种或脚本创建的，不需要特定的编码。任何用户都可以从图表 UI 创建这些类型的警报。

其他类型的警报（在`alert()`函数调用上触发的\*\*脚本警报和`alertcondition()`警报）需要脚本中存在特定的Pine Script™代码来创建警报事件，然后脚本用户可以使用图表UI从中创建警报。此外，虽然脚本用户可以在其图表上加载的任何策略的图表 UI 上创建触发订单填写事件的\*\*脚本警报，但程序员可以在其脚本中为经纪商模拟器填写的每种订单类型指定显式订单填写警报消息。

本页介绍了 Pine Script™ 程序员编写脚本来创建警报事件的不同方式，脚本用户又可以通过图表 UI 创建警报。我们将涵盖：

- 如何使用[alert\(\)](#)函数在指标或策略中调用`alert()` 函数，然后将其包含在从图表UI 创建的脚本警报中。
- 如何添加自定义警报消息以包含在触发策略的订单填充事件的\*\*脚本警报中。

- 如何使用[alertcondition\(\)](#)函数仅在指标中生成*alertcondition()* 事件，然后可以使用该事件从图表UI创建*alertcondition()* 警报。

请记住：

- 任何与警报相关的 Pine Script™ 代码都无法在图表 UI 中创建正在运行的警报；它只是创建警报事件，然后脚本用户可以使用这些事件从图表 UI 创建运行警报。
- 警报仅在实时栏中触发。因此，处理任何类型警报的 Pine Script™ 代码的操作范围仅限于实时柱。
- 当在图表 UI 中创建警报时，TradingView 会保存脚本及其输入的镜像，以及图表的主要交易品种和时间范围，以在其服务器上运行警报。因此，对脚本输入或图表的后续更改不会影响之前根据它们创建的运行警报。如果您希望对上下文所做的任何更改反映在正在运行的警报的行为中，则需要删除该警报并在新上下文中创建一个新警报。

## 背景

如今，Pine 程序员可以使用不同的方法在其脚本中创建警报事件，这些方法是在 Pine Script™ 发展过程中不断部署的增强功能的结果。Alertcondition () 函数仅适用于指标，是第一个允许 Pine Script™ 程序员创建警报事件的功能。然后是策略的订单填充警报，当经纪商模拟器创建订单填充事件时触发。订单执行事件不需要脚本用户编写特殊代码来创建警报，但通过 `alert_message` 订单生成 `strategy.*()` 函数的参数，程序员可以通过为任意数量的订单定义不同的警报消息来自定义在订单执行事件上触发的警报消息。订单履行事件。

Alert () 函数是 Pine Script™ 的最新添加功能。它或多或少取代了 [alertcondition\(\)](#)，并且在策略中使用时，为订单执行事件警报提供了有用的补充。

## 哪种类型的警报最好？

对于 Pine Script™ 程序员来说，[alert\(\)](#) 函数通常使用起来更容易、更灵活。与 [alertcondition\(\)](#) 相反，它允许动态警报消息，适用于指标和策略，并且程序员决定 alert() 事件的频率。

虽然可以在 Pine 中的任何可编程逻辑上生成[alert\(\)调用](#)，包括当订单发送到策略中的经纪商模拟器时，但它们无法编码为在订单执行（或成交）时触发，因为在订单发送到经纪商模拟器之后，模拟器控制它们的执行，并且不会将填充事件直接报告回脚本。

当脚本用户想要生成有关策略订单执行事件的警报时，他必须在“创建警报”对话框中创建有关该策略的脚本警报时包含这些事件。用户无需在脚本中编写特殊代码即可执行此操作。然而，程序员可以通过 `alert_message` 在订单生成函数调用中使用参数来自定义与订单填写事件一起发送的消息 `strategy.*()`。结合使用[alert\(\)](#) `alert_message` 调用和在订单生成调用中使用自定义参数 `strategy.*()`，程序员可以在脚本执行中发生的大多数情况下生成警报事件。

为了向后兼容，[alertcondition \(\)](#) 函数保留在 Pine Script™ 中，但它也可以有利地用于生成可在“创建警报”对话框的“条件”字段中作为单独项目选择的不同警报。

## 脚本警报

当脚本用户使用“创建警报”对话框创建脚本警报时，能够触发警报的事件将根据警报是从指标还是策略创建而有所不同。

从指标创建的脚本警报将在以下情况下触发：

- 该指标包含[alert\(\)](#)调用。

- 该代码的逻辑允许执行特定的[alert\(\)调用](#)。
- [在alert\(\)调用中指定的频率](#)允许触发警报。

从策略创建的脚本警报可以在\*\*`alert()`函数调用、订单执行事件或两者上触发。创建策略警报的脚本用户决定他希望在脚本警报中包含哪种类型的事件。虽然用户可以在订单执行事件上创建脚本警报，而不需要包含特殊代码的策略，但它必须包含[alert\(\)调用](#)，以便用户在其脚本警报中包含 `alert()` 函数调用。

## alert() 函数事件

`Alert()`函数具有以下签名：

```
alert(message, freq)
```

- `message`

表示警报触发时发送的消息文本的“系列字符串”。因为此参数允许“系列”值，所以它可以在运行时生成，并且条形图与条形图不同，从而使其动态。

- `freq`

指定警报触发频率的“输入字符串”。有效参数是：`alert.freq_once_per_bar`：只有每个实时栏的第一个调用才会触发警报（默认值）。`alert.freq_once_per_bar_close`：仅当实时栏关闭并且在该脚本迭代期间执行[alert\(\)调用](#)时才会触发警报。`alert.freq_all`：实时栏内的所有来电都会触发警报。

`Alert()`函数可用于指标和策略。对于要触发在[alert\(\)函数调用](#)上配置的脚本警报的`alert()`调用，脚本的逻辑必须允许执行[alert\(\)调用](#)，并且参数确定的频率必须允许触发警报。`freq`

请注意，默认情况下，策略会在柱线收盘时重新计算，因此如果在策略中使用频率为或的[alert\(\)函数](#)，则在柱线收盘时调用该函数的频率不会超过一次。为了使[alert\(\)函数](#)能够在bar构建过程中被调用，您需要启用该选项。`alert.freq_all` `alert.freq_once_per_bar` `calc_on_every_tick`

## 使用所有“alert()”调用

让我们看一个检测 RSI 中心线交叉的示例：

```
//@version=5
indicator("All `alert()` calls")
r = ta.rsi(close, 20)

// Detect crosses.
xUp = ta.crossover(r, 50)
xDn = ta.crossunder(r, 50)
// Trigger an alert on crosses.
if xUp
    alert("Go long (RSI is " + str.tostring(r, "#.00")"))
else if xDn
    alert("Go short (RSI is " + str.tostring(r, "#.00")"))

plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
hline(50)
```

```
plot(r)
```

如果从此脚本创建脚本警报：

- 当 RSI 向上穿过中心线时，脚本警报将触发，并显示“做多...”消息。当 RSI 向下穿过中心线时，脚本警报将触发，并显示“做空...”消息。
- 由于没有为 `alert(freq)` 调用中的参数指定参数，因此将使用默认值，因此警报仅在实时条期间第一次执行每个 `alert()` 调用时触发。`alert.freq_once_per_bar`
- 与警报一起发送的消息由两部分组成：一个常量字符串，然后是 `str.toString()` 调用的结果，其中包括脚本执行 `alert()` 调用时的 RSI 值。交叉上涨的警报消息如下所示：“做多 (RSI 为 53.41) ”。
- 由于脚本警报总是在调用 `alert()` 时触发，只要调用中使用的频率允许，该特定脚本不允许脚本用户将其脚本警报限制为仅长整型，例如。

注意：

- 与始终放置在第 0 列（在脚本的全局作用域中）的 `alertcondition()` 调用相反，`alert()` 调用放置在 `if` 分支的本地作用域中，因此它仅在满足触发条件时才执行。如果将 `alert()` 调用放置在脚本全局范围的第 0 列中，它将在所有柱上执行，这可能不是所需的行为。
- 由于使用了 `str.toString()` 调用，`alertcondition()` 无法接受与我们用于警报消息的相同字符串。`Alertcondition()` 消息必须是常量字符串。

最后，因为 `alert()` 消息可以在运行时动态构造，所以我们可以使用以下代码来生成警报事件：

```
// Trigger an alert on crosses.  
if xUp or xDn  
    firstPart = (xUp ? "Go long" : "Go short") + " (RSI is "  
    alert(firstPart + str.toString(r, "#.00"))
```

## 使用选择性的“`alert()`”调用

当用户在 `alert()` 函数调用上创建脚本警报时，只要满足其频率限制，该警报就会在脚本对 `alert()` 函数进行的任何调用时触发。如果您希望允许脚本的用户选择脚本中的哪个 `alert()` 函数调用将触发脚本警报，您将需要为他们提供在脚本输入中表明他们的偏好的方法，并在您的脚本中编写适当的逻辑。脚本。这样，脚本用户将能够从单个脚本创建多个脚本警报，每个脚本警报的行为根据在图表 UI 中创建警报之前在脚本输入中所做的选择而有所不同。

假设，对于我们的下一个示例，我们希望提供仅针对多头、仅空头或两者触发警报的选项。您可以像这样编写脚本：

```
//@version=5  
indicator("Selective `alert()` calls")  
detectLongsInput = input.bool(true, "Detect Longs")  
detectShortsInput = input.bool(true, "Detect Shorts")  
repaintInput = input.bool(false, "Allow Repainting")  
  
r = ta.rsi(close, 20)  
// Detect crosses.  
xUp = ta.crossover(r, 50)  
xDn = ta.crossunder(r, 50)
```

```

// Only generate entries when the trade's direction is allowed in inputs.
enterLong = detectLongsInput and xUp and (repaintInput or barstate.isconfirmed)
enterShort = detectShortsInput and xDn and (repaintInput or barstate.isconfirmed)
// Trigger the alerts only when the compound condition is met.
if enterLong
    alert("Go long (RSI is " + str.tostring(r, "#.00")"))
else if enterShort
    alert("Go short (RSI is " + str.tostring(r, "#.00")))

plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(enterShort, "Go Short", "▼", location.top, color.red, size = size.tiny)
hline(50)
plot(r)

```

注意如何：

- 我们创建一个复合条件，仅当用户的选择允许朝该方向输入时才满足该条件。仅当脚本的输入中启用了长条目时，中心线交叉处的长条目才会触发警报。
- 我们让用户表明他的重画偏好。当他不允许重新绘制计算时，我们会等到柱形确认后触发复合条件。这样，警报和标记仅出现在实时栏的末尾。
- 如果该脚本的用户想要从此脚本创建两个不同的脚本警报，即一个仅在多头触发，一个仅在空头触发，那么他需要：
  - 在输入中仅选择“检测长整型”，并在脚本上创建第一个脚本警报。
  - 在输入中仅选择“检测短裤”，并在脚本上创建另一个脚本警报。

## 在策略中

[Alert\(\)](#) 函数调用也可以在策略中使用，但默认情况下，策略仅在实时柱线[收盘](#)时执行。除非在[strategy\(\)](#) 声明语句中使用，否则所有[alert\(\)](#)调用都将使用频率，无论使用什么参数。`calc_on_every_tick = true``alert.freq_once_per_bar_close``freq`

虽然策略上的脚本警报将使用订单填写事件在经纪商模拟器填写订单时触发警报，但可以有利地使用[alert\(\)](#)来生成[策略中的其他警报事件。](#)

当 RSI 连续三个柱线走势不利于交易时，此策略会创建[alert\(\)](#)函数调用：

```

//@version=5
strategy("Strategy with selective `alert()` calls")
r = ta.rsi(close, 20)

// Detect crosses.
xUp = ta.crossover(r, 50)
xDn = ta.crossunder(r, 50)
// Place orders on crosses.
if xUp
    strategy.entry("Long", strategy.long)
else if xDn
    strategy.entry("Short", strategy.short)

```

```

// Trigger an alert when RSI diverges from our trade's direction.
divInLongTrade = strategy.position_size > 0 and ta.falling(r, 3)
divInShortTrade = strategy.position_size < 0 and ta.rising( r, 3)
if divInLongTrade
    alert("WARNING: Falling RSI", alert.freq_once_per_bar_close)
if divInShortTrade
    alert("WARNING: Rising RSI", alert.freq_once_per_bar_close)

plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
plotchar(divInLongTrade, "WARNING: Falling RSI", "●", location.top, color.red,
size = size.tiny)
plotchar(divInShortTrade, "WARNING: Rising RSI", "●", location.bottom, color.lime,
size = size.tiny)
hline(50)
plot(r)

```

如果用户从此策略创建脚本警报，并在其警报中包含\*\*订单成交事件和`alert()`函数调用，则每当执行订单或脚本执行其中一个`alert()`调用时，警报就会触发在实时栏的结束迭代上，即，当`barstate.isrealtime`和`barstate.isconfirmed`都为 true 时。脚本中的`alert()`函数事件只会在实时柱关闭时触发警报，因为是`alert()`调用中参数`alert.freq_once_per_bar_close`所使用的参数。`freq`

## 订单填写事件

当从指标创建脚本警报时，它只能在调用\*\*`alert()`函数时触发。但是，当从策略创建脚本警报时，用户可以指定\*\*订单执行事件也触发脚本警报。订单填写事件是由经纪模拟器生成的导致执行模拟订单的任何事件。它相当于由经纪商/交易所填写的交易订单。订单下达后不一定被执行。在策略中，订单的执行只能在事后通过分析内置变量（例如`strategy.opentrades`或`strategy.position_size`）的变化来间接检测。因此，在订单执行事件上配置的脚本警报非常有用，因为它们允许在脚本逻辑检测到订单执行的精确时刻触发警报。

Pine Script™ 程序员可以自定义执行特定命令时发送的警报消息。虽然这不是触发订单执行事件的先决条件，但自定义警报消息可能很有用，因为它们允许将自定义语法包含在警报中，以便将实际订单路由到第三方执行引擎等。为特定订单成交事件指定自定义警报消息是通过`alert_message`可以生成订单的函数中的参数来完成的：`strategy.close()`、`strategy.entry()`、`strategy.exit()`和`strategy.order()`。

用于参数的参数`alert_message`是“系列字符串”，因此可以使用脚本可用的任何变量动态构造它，只要将其转换为字符串格式即可。

`alert_message`让我们看一下在[策略.entry\(\)](#)调用中使用参数的策略：

```

//@version=5
strategy("Strategy using `alert_message`")
r = ta.rsi(close, 20)

// Detect crosses.
xUp = ta.crossover( r, 50)
xDn = ta.crossunder(r, 50)
// Place order on crosses using a custom alert message for each.

```

```

if xUp
    strategy.entry("Long", strategy.long, stop = high, alert_message = "Stop-buy
executed (stop was " + str.tostring(high) + ")")
else if xDn
    strategy.entry("Short", strategy.short, stop = low, alert_message = "Stop-sell
executed (stop was " + str.tostring(low) + ")")

plotchar(xUp, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Go Short", "▼", location.top, color.red, size = size.tiny)
hline(50)
plot(r)

```

注意：

- `stop` 我们在 `strategy.entry()` 调用中使用该参数，这会创建止损买入和止损卖出订单。这意味着只有当价格高于 `high` 下达订单的柱上的价格时，买入订单才会执行，只有当价格低于下达订单的柱上的最低价时，卖出订单才会执行。
- 我们用 `plotchar()` 绘制的向上/向下箭头是在下订单时绘制的。在订单实际执行之前，可能会经过任意数量的柱，并且在某些情况下，由于价格不满足所需条件，订单将永远不会被执行。
- 因为我们 `id` 对所有买单使用相同的参数，所以在满足前一个订单的条件之前下的任何新买单都将替换该订单。这同样适用于卖单。
- 参数中包含的变量 `alert_message` 在订单执行时（即警报触发时）进行评估。

当该 `alert_message` 参数用于策略的订单生成 `strategy.*()` 函数调用时，脚本用户在创建有关订单成交事件的脚本警报时，必须在“\*\*创建 {{strategy.order.alert\_message}} 警报”对话框的“消息”字段中包含占位符。这是必需的，因此订单生成函数调用中使用的参数将在每个订单填写事件触发的警报消息中使用。当仅在“消息”字段中使用占位符且该参数仅出现在策略中的部分订单生成函数调用中时，空字符串将替换由任何订单生成函数调用触发的警报消息中的占位符不使用参数。`alert_message``strategy.*`  
(`)`{{strategy.order.alert_message}}` `alert_message``strategy.*()``strategy.*`  
(`)`alert_message`

虽然用户可以在“创建警报”对话框的“消息”字段中使用其他占位符来创建有关订单成交事件的警报，但它们不能在的参数中使用 `alert_message`。

## [alertcondition\(\)](#) 事件

`Alertcondition()` 函数允许程序员在其指标中创建单独的 `alertcondition` 事件。一个指示器可能包含多个 `alertcondition()` 调用。脚本中每次调用 `alertcondition()` 都会创建一个相应的警报，可在“创建警报”对话框的“条件”下拉菜单中进行选择。

虽然策略脚本中存在 `alertcondition()` 调用不会导致编译错误，但无法从中创建警报。

`Alertcondition()` 函数具有以下签名：

```
alertcondition(condition, title, message)
```

- `condition`

一个“series bool”值 (`true` 或 `false`)，决定何时触发警报。这是一个必需的参数。当该值为 `true` 时，`true` 警报将触发。当该值为 `false` 时，`false` 警报将不会触发。与 [alert\(\)](#) 函数调用相反，[alertcondition\(\)](#) 调用必须从一行的零列开始，因此不能放置在条件块中。

- `title`

“常量字符串”可选参数，用于设置警报条件的名称，该名称将显示在图表 UI 中“创建警报”对话框的“条件”字段中。如果未提供参数，将使用“Alert”。

- `message`

“const string”可选参数，指定警报触发时要显示的文本消息。该文本将出现在“创建警报”对话框的“消息”字段中，脚本用户可以在创建警报时对其进行修改。由于此参数必须是“**const string**”，因此必须在编译时已知，因此不能逐条变化。但是，它可以包含占位符，这些占位符将在运行时被动态值替换，动态值可能会更改条形。有关列表，请参阅本页的[占位符部分](#)。

`Alertcondition()` 函数不包含 `freq` 参数。`Alertcondition()` 警报的频率由用户在“创建警报”对话框中确定。

## 使用一个条件

以下是创建 `alertcondition()` 事件的代码示例：

```
//@version=5
indicator(`alertcondition()` on single condition")
r = ta.rsi(close, 20)

xUp = ta.crossover(r, 50)
xDn = ta.crossunder(r, 50)

plot(r, "RSI")
hline(50)
plotchar(xUp, "Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(xDn, "Short", "▼", location.top, color.red, size = size.tiny)

alertcondition(xUp, "Long Alert", "Go long")
alertcondition(xDn, "Short Alert", "Go short")
```

因为我们的脚本中有两个 [alertcondition\(\)](#) 调用，所以“创建警报”对话框的“条件”字段中将提供两个不同的警报：“长警报”和“短警报”。

如果我们想在交叉发生时包含 RSI 的值，我们不能像在 [alert\(\)](#) 调用或策略中的参数中那样简单地 `message` 使用将其值添加到字符串中。但是，我们可以使用占位符将其包含在内。这显示了两种选择：`str.toString(r)` ` alert_message`

```
alertcondition(xUp, "Long Alert", "Go long. RSI is {{plot_0}}")
alertcondition(xDn, "Short Alert", 'Go short. RSI is {{plot("RSI")}}')
```

注意：

- 第一行使用  `{{plot_0}}`  占位符，其中绘图编号对应于脚本中绘图的顺序。

- 第二行使用 `{{plot("[plot_title]")}}` 占位符类型，其中必须包含脚本 `title` 中用于绘制 RSI 的 `plot()` 调用。双引号用于将绘图标题括在 `{{plot("RSI")}}` 占位符内。这就要求我们使用单引号来包裹 `message` 字符串。
- 使用其中一种方法，我们可以包含指标绘制的任何数值，但由于无法绘制字符串，因此无法使用字符串变量。

## 使用复合条件

如果我们想为脚本用户提供使用多个 `alertcondition()` 调用从指标创建单个警报的可能性，我们需要在脚本的输入中提供选项，用户可以通过这些选项在创建警报之前指示他们想要触发警报的条件。

该脚本演示了一种方法：

```
//@version=5
indicator(`alertcondition()` on multiple conditions`)
detectLongsInput = input.bool(true, "Detect Longs")
detectShortsInput = input.bool(true, "Detect Shorts")

r = ta.rsi(close, 20)
// Detect crosses.
xUp = ta.crossover(r, 50)
xDn = ta.crossunder(r, 50)
// Only generate entries when the trade's direction is allowed in inputs.
enterLong = detectLongsInput and xUp
enterShort = detectShortsInput and xDn

plot(r)
plotchar(enterLong, "Go Long", "▲", location.bottom, color.lime, size = size.tiny)
plotchar(enterShort, "Go Short", "▼", location.top, color.red, size = size.tiny)
hline(50)
// Trigger the alert when one of the conditions is met.
alertcondition(enterLong or enterShort, "Compound alert", "Entry")
```

请注意如何允许在两个条件之一上触发 `alertcondition()` 调用。仅当用户在创建警报之前在脚本输入中启用每个条件时，才能触发警报。

## 占位符

这些占位符可以在 `alertcondition()` `message` 调用的参数中使用。当警报触发时，它们将被动态值替换。它们是在 `alertcondition()` 消息中包含动态值（可以逐条变化的值）的唯一方法。

请注意，从图表 UI 中的“创建警报”对话框创建 `alertcondition()` 警报的用户也可以在对话框的“消息”字段中使用这些占位符。

- `{{exchange}}`

警报中使用的交易代码（NASDAQ、NYSE、MOEX 等）。请注意，对于延迟符号，交换将以“`DL`”或“`DLY`”结束。例如，“`NYMEX_DL`”。

- `{{interval}}`

返回创建警报的图表的时间范围。请注意，范围图表是根据 1m 数据计算的，因此在范围图表上创建的任何警报上，占位符将始终返回“1”。

- `{{open}} , {{high}} , {{low}} , {{close}} , {{volume}}`

触发警报的柱的相应值。

- `{{plot_0}} 、 {{plot_1}} [...] 、 {{plot_19}}`

相应地块编号的值。绘图按照脚本中出现的顺序从 0 到 19 编号，因此只能使用前 20 个绘图中的一个。例如，内置的“成交量”指标有两个输出系列：成交量和成交量 MA，因此您可以使用以下内容：

```
alertcondition(volume > ta.sma(volume,20), "Volume alert", "Volume {{plot_0}} > average {{plot_1}}")
```

- `{{plot("[plot_title])}}`

当需要使用`plot()` title 调用中使用的参数 来引用绘图时，可以使用此占位符。请注意，占位符内必须使用双引号 () 来包裹参数。这需要使用单引号 () 来包裹字符串：`"`title`" message`

```
//@version=5
indicator("")
r = ta.rsi(close, 14)
xUp = ta.crossover(r, 50)
plot(r, "RSI", display = display.none)
alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')
```

- `{{ticker}}`

警报中使用的交易品种代码 (AAPL、BTCUSD 等)。

- `{{time}}`

返回小节开头的时间。时间为 UTC，格式为 `yyyy-MM-ddTHH:mm:ssZ`，例如：`2019-08-27T09:56:00Z`。

- `{{timenow}}`

警报触发的当前时间，格式与相同  `{{time}}`。无论图表的时间范围如何，精度都精确到秒。

## 避免使用警报重新绘制

重绘交易者希望通过警报避免的最常见情况是，他们必须防止在实时柱期间的某个点触发警报，而该警报在收盘时不会触发。当满足以下条件时就会发生这种情况：

- 触发警报的条件中使用的计算在实时条期间可能会有所不同。例如，任何使用 `high`、`low` 或 的计算都会出现这种情况 `close`，其中包括几乎所有内置指标。当较高时间范围的当前柱尚未关闭时，使用比图表更高时间范围的任何`request.security()`调用的结果也会出现这种情况。
- 警报可以在实时柱收盘之前触发，因此可以使用除“每柱收盘一次”之外的任何频率。

避免此类重画的最简单方法是配置警报的触发频率，以便它们仅在实时柱收盘时触发。没有灵丹妙药；避免这种类型的重画总是需要等待确认的信息，这意味着交易者必须牺牲即时性来获得可靠性。

请注意，其他类型的重画（例如我们的重画部分中记录的重画）可能无法通过简单地在实时柱收盘时触发警报来预防。

# 背景

bgcolor() 函数更改脚本背景的颜色。如果脚本在模式下运行，那么它将为图表的背景着色。`overlay = true` 该函数的签名是：

```
bgcolor(color, offset, editable, show_last, title) → void
```

它的 `color` 参数允许使用“系列颜色”作为其参数，因此可以在表达式中动态计算它。

如果正确的透明度不是要使用的颜色的一部分，则可以使用[color.new\(\)](#)函数生成。

下面是一个为交易时段的背景着色的脚本（例如，在 30 分钟 EURUSD 上尝试）：

```
//@version=5
indicator("Session backgrounds", overlay = true)

// Default color constants using transparency of 25.
BLUE_COLOR      = #0050FF40
PURPLE_COLOR    = #0000FF40
PINK_COLOR      = #5000FF40
NO_COLOR        = color(na)

// Allow user to change the colors.
preMarketColor  = input.color(BLUE_COLOR, "Pre-market")
regSessionColor = input.color(PURPLE_COLOR, "Pre-market")
postMarketColor = input.color(PINK_COLOR, "Pre-market")

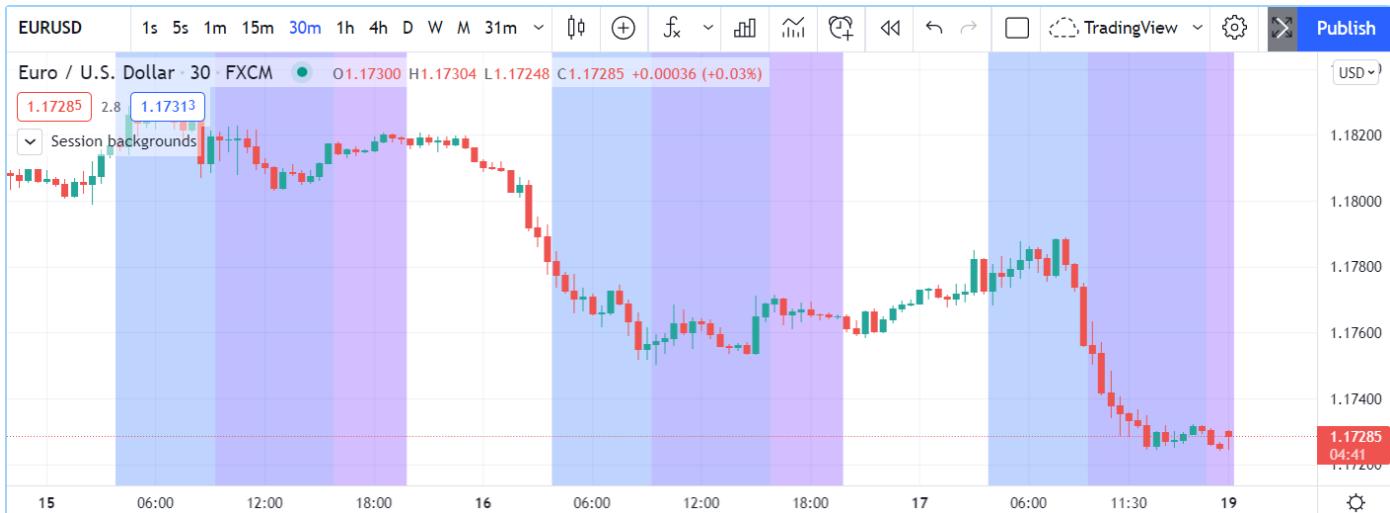
// Function returns `true` when the bar's time is
timeInRange(tf, session) =>
    time(tf, session) != 0

// Function prints a message at the bottom-right of the chart.
f_print(_text) =>
    var table _t = table.new(position.bottom_right, 1, 1)
    table.cell(_t, 0, 0, _text, bgcolor = color.yellow)

var chartIs30MinOrLess = timeframe.isseconds or (timeframe.isintraday and
timeframe.multiplier <=30)
sessionColor = if chartIs30MinOrLess
    switch
        timeInRange(timeframe.period, "0400-0930") => preMarketColor
        timeInRange(timeframe.period, "0930-1600") => regSessionColor
        timeInRange(timeframe.period, "1600-2000") => postMarketColor
    => NO_COLOR
else
    f_print("No background is displayed.\nChart timeframe must be <= 30min.")
```

```
NO_COLOR
```

```
bgcolor(sessionColor)
```



注意：

- 该脚本仅适用于 30 分钟或更短的图表时间范围。当图表的时间范围高于 30 分钟时，它会打印一条错误消息。
- 当由于图表的时间范围不正确而使用 if 结构的 else 分支时，本地块将返回颜色，NO\_COLOR 以便在这种情况下不显示背景。
- 我们首先使用基色初始化常量，其中包括 40 最后的十六进制表示法的透明度。透明度的反向 00-FF 刻度上的十六进制表示法中的 40 对应于 Pine Script™ 透明度的 0-100 十进制刻度中的 75。
- 我们提供颜色输入，允许脚本用户更改我们建议的默认颜色。

在下一个示例中，我们为 CCI 线的背景生成渐变：

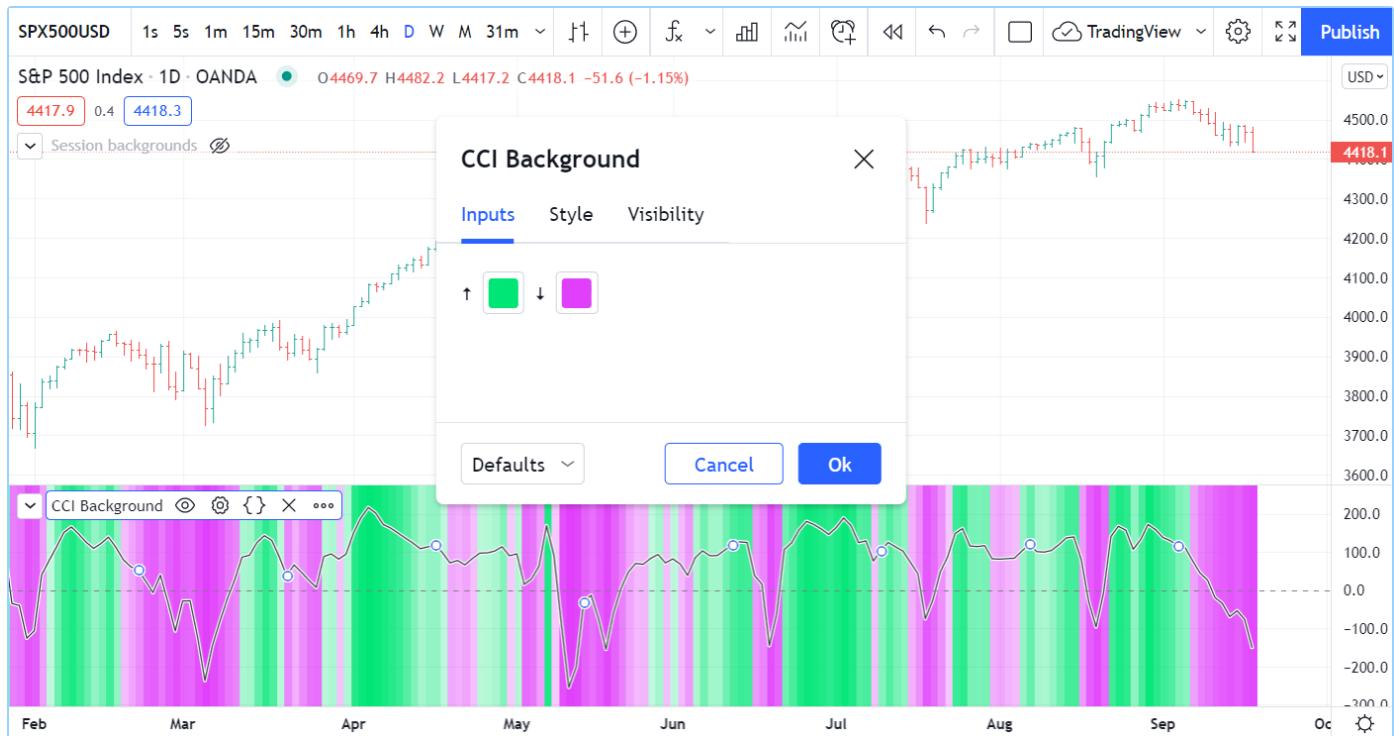
```
//@version=5
indicator("CCI Background")

bullColor = input.color(color.lime, "█", inline = "1")
bearColor = input.color(color.fuchsia, "█", inline = "1")

// Calculate CCI.
myCCI = ta.cci(hlc3, 20)
// Get relative position of CCI in last 100 bars, on a 0-100% scale.
myCCIPosition = ta.percentrank(myCCI, 100)
// Generate a bull gradient when position is 50-100%, bear gradient when position is 0-50%.
backgroundColor = if myCCIPosition >= 50
    color.from_gradient(myCCIPosition, 50, 100, color.new(bullColor, 75), bullColor)
else
    color.from_gradient(myCCIPosition, 0, 50, bearColor, color.new(bearColor, 75))

// Wider white line background.
plot(myCCI, "CCI", color.white, 3)
```

```
// Think black line.
plot(myCCI, "CCI", color.black, 1)
// Zero level.
hline(0)
// Gradient background.
bgcolor(backgroundColor)
```



注意：

- 我们使用`ta.cci()`内置函数来计算指标值。
- 我们使用`ta.percentrank()`内置函数来计算，即最后100个柱中过去值低于当前值的`myCCIPosition`百分比。`myCCI` `myCCI`
- 为了计算梯度，我们使用内置的`color.from_gradient()``myCCIPosition`的两种不同调用：一种是在50-100%范围内时的公牛梯度，这意味着更多过去的值低于其当前值，另一种是对于熊市梯度，当`myCCIPosition`处于0-49.99%范围内时，这意味着更多过去的值高于它。
- 我们提供输入，以便用户可以更改牛市/熊市颜色，并且我们将两个颜色输入小部件放置在同一行上，并在两个`input.color()`调用中使用。`inline = "1"`
- 我们使用两个`plot()`调用来绘制CCI信号，以在繁忙的背景上实现最佳对比度：第一个图是3像素宽的白色背景，第二个`plot()`调用绘制细的1像素宽的黑线。

有关背景的更多示例，请参阅[颜色页面](#)。

## 条形着色

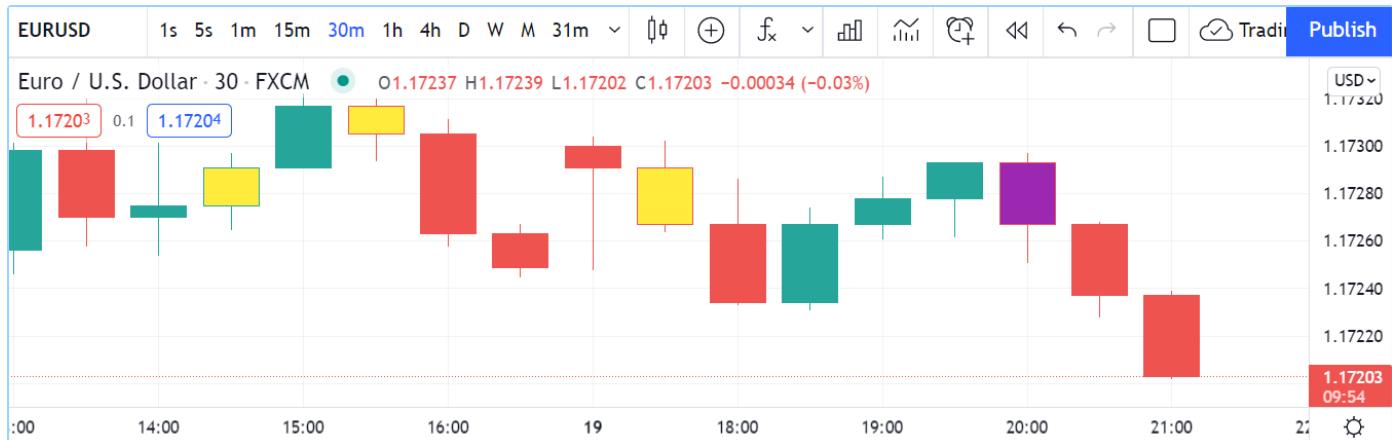
`barcolor()`函数可让您为图表条形着色。它是唯一允许在窗格中运行的脚本影响图表的 Pine Script™ 功能。

该函数的签名是：

```
barcolor(color, offset, editable, show_last, title) → void
```

着色可以是有条件的，因为 `color` 参数接受“系列颜色”参数。

以下脚本以不同的颜色呈现内部和外部条形：



```
//@version=5
indicator("barcolor example", overlay = true)
isUp = close > open
isDown = close <= open
isOutsideUp = high > high[1] and low < low[1] and isUp
isOutsideDown = high > high[1] and low < low[1] and isDown
isInside = high < high[1] and low > low[1]
barcolor(isInside ? color.yellow : isOutsideUp ? color.aqua : isOutsideDown ?
color.purple : na)
```

注意：

- `na` 值使条形保持原样。
- 在 `barcolor()` 调用中，我们使用嵌入的`?:` 三元运算符表达式来选择颜色。

## 条形图

### 介绍

`plotcandle()` 内置函数用于绘制蜡烛图。 `plotbar()` 用于绘制常规条形图。

这两个函数都需要四个参数，这些参数将用于它们将绘制的柱的OHLC 价格（[开盘价](#)、[最高价](#)、[最低价](#)、[收盘价](#)）。如果其中之一是 `na`，则不会绘制条形图。

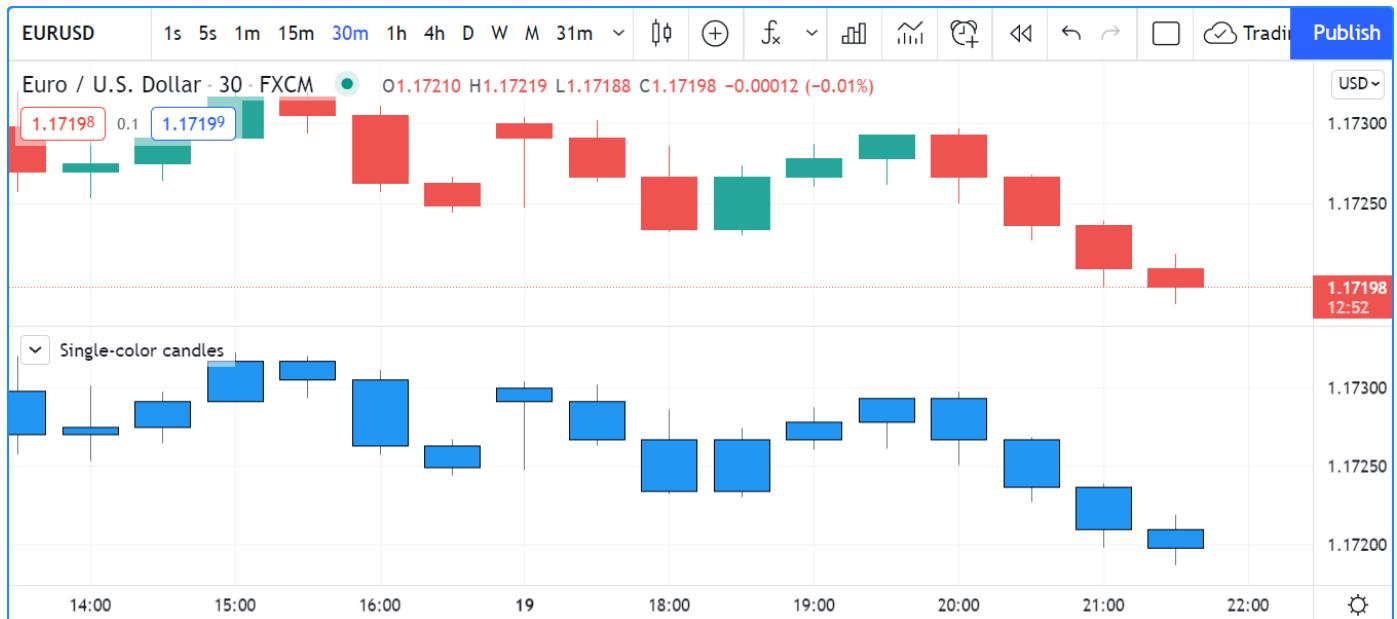
### 使用“`plotcandle()`”绘制蜡烛图

`plotcandle()` 的签名是：

```
plotcandle(open, high, low, close, title, color, wickcolor, editable, show_last, bordercolor, display) → void
```

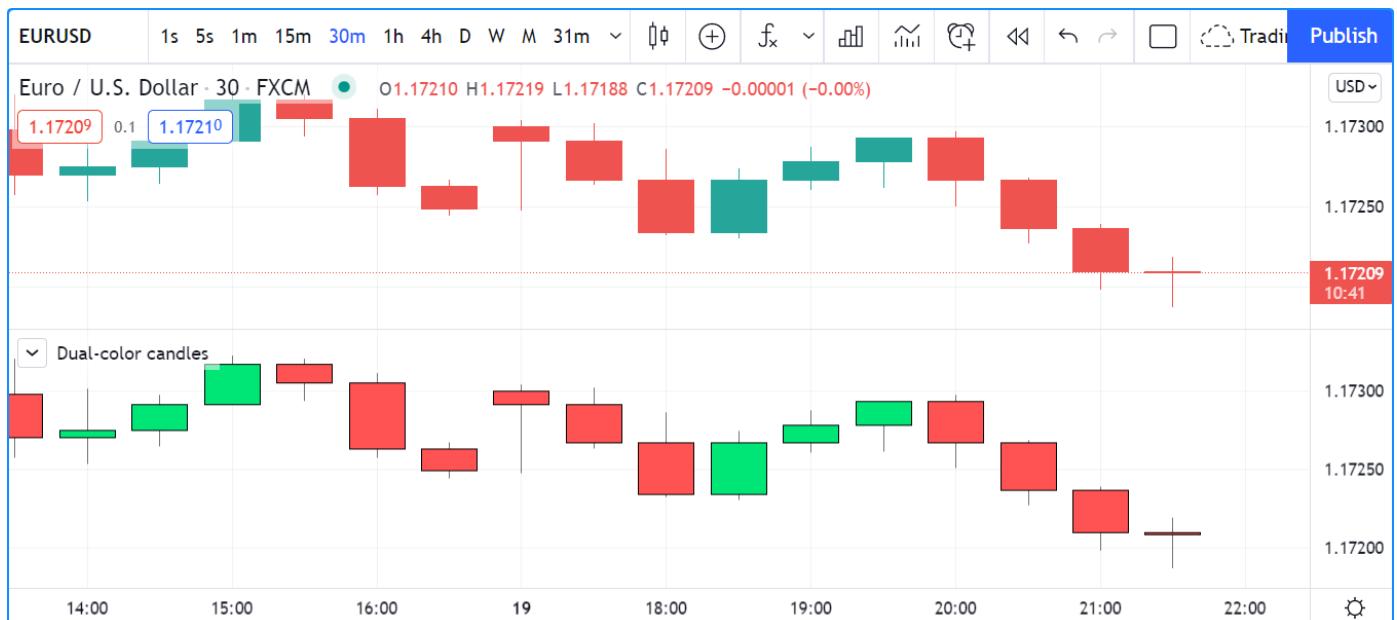
这将在单独的窗格中使用习惯的 OHLC 值绘制简单的蜡烛图，全部为蓝色：

```
//@version=5
indicator("Single-color candles")
plotcandle(open, high, low, close)
```



要将它们着色为绿色或红色，我们可以使用以下代码：

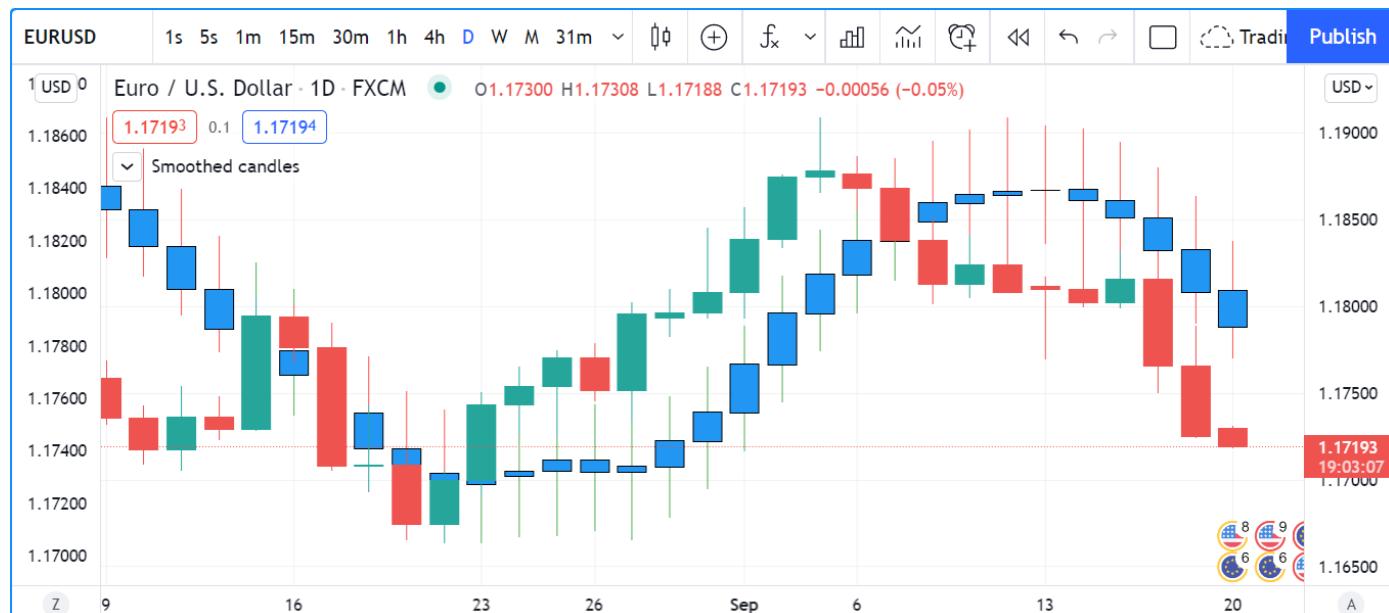
```
//@version=5
indicator("Example 2")
paletteColor = close >= open ? color.lime : color.red
plotbar(open, high, low, close, color = paletteColor)
```



请注意，该 `color` 参数接受“系列颜色”参数，因此常量值（例如 `color.red`、`color.lime`、`"#FF9090"`）以及在运行时计算颜色的表达式（如此处对 `paletteColor` 变量所做的那样）都将起作用。

您可以使用实际 OHLC 值以外的值构建条形图或蜡烛图。例如，您可以使用以下代码计算和绘制平滑收盘价，该代码还根据收盘价相对于我们指标的平滑收盘价 () 的位置为烛芯着色：

```
//@version=5
indicator("Smoothed candles", overlay = true)
lenInput = input.int(9)
smooth(source, length) =>
    ta.sma(source, length)
o = smooth(open, lenInput)
h = smooth(high, lenInput)
l = smooth(low, lenInput)
c = smooth(close, lenInput)
ourWickColor = close > c ? color.green : color.red
plotcandle(o, h, l, c, wickcolor = ourWickColor)
```



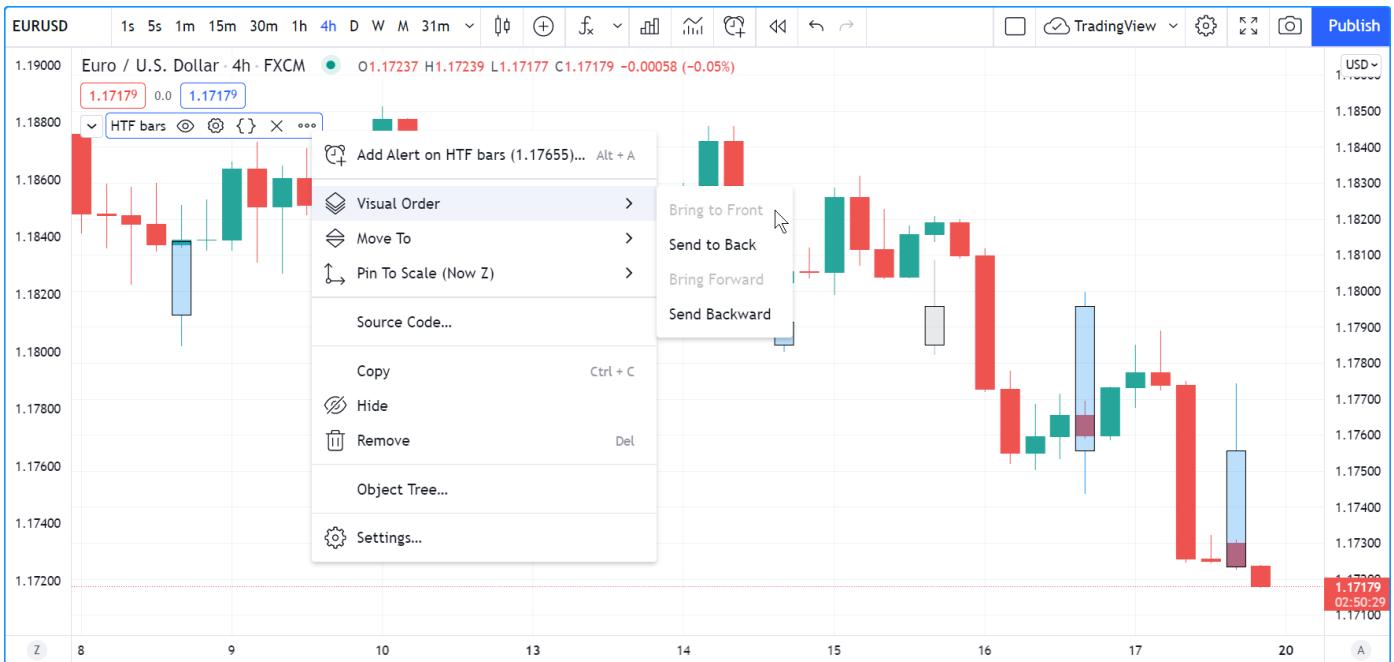
您可能会发现绘制取自较高时间范围的 OHLC 值很有用。例如，您可以在日内图表上绘制日线图：

```
// NOTE: Use this script on an intraday chart.
//@version=5
indicator("Daily bars")

// Use gaps to only return data when the 1D timeframe completes, `na` otherwise.
[o, h, l, c] = request.security(syminfo.tickerid, "D", [open, high, low, close], gaps = barmerge.gaps_on)

var color UP_COLOR = color.silver
var color DN_COLOR = color.blue
color wickColor = c >= o ? UP_COLOR : DN_COLOR
color bodyColor = c >= o ? color.new(UP_COLOR, 70) : color.new(DN_COLOR, 70)
// Only plot candles on intraday timeframes,
```

```
// and when non `na` values are returned by `request.security()` because a HTF has completed.
plotcandle(timeframe.isintraday ? o : na, h, l, c, color = bodyColor, wickcolor = wickColor)
```



注意：

- 我们在使用脚本的“更多”菜单中的“视觉顺序/置于前面”后显示脚本的情节。这会导致我们脚本的蜡烛图出现在图表蜡烛图的顶部。
- 该脚本仅在满足两个条件时才会显示蜡烛：
  - 该图表使用日内时间范围（请参阅`plotcandle()` `timeframe.isintraday` 调用中的检查）。我们这样做是因为在高于或等于 1D 的时间范围内显示每日值没有用。
  - `request.security()` 函数返回非`na`值（请参阅函数调用）。`gaps = barmerge.gaps_on`
- 我们使用元组() 和 `request.security()` 在一次调用中获取四个值。`[open, high, low, close]`
- 我们仅使用`var`来声明 零柱上的常量 `UP_COLOR` 和颜色常量。`DN_COLOR` 我们使用常量是因为这些颜色在我们的代码中不止一处使用。这样，如果我们需要更改它们，只需在一处进行即可。
- 我们在变量初始化中为蜡烛体创建了较浅的透明度 `bodyColor`，因此它们不会遮挡图表的蜡烛。

## 使用“`plotbar()`”绘制条形图

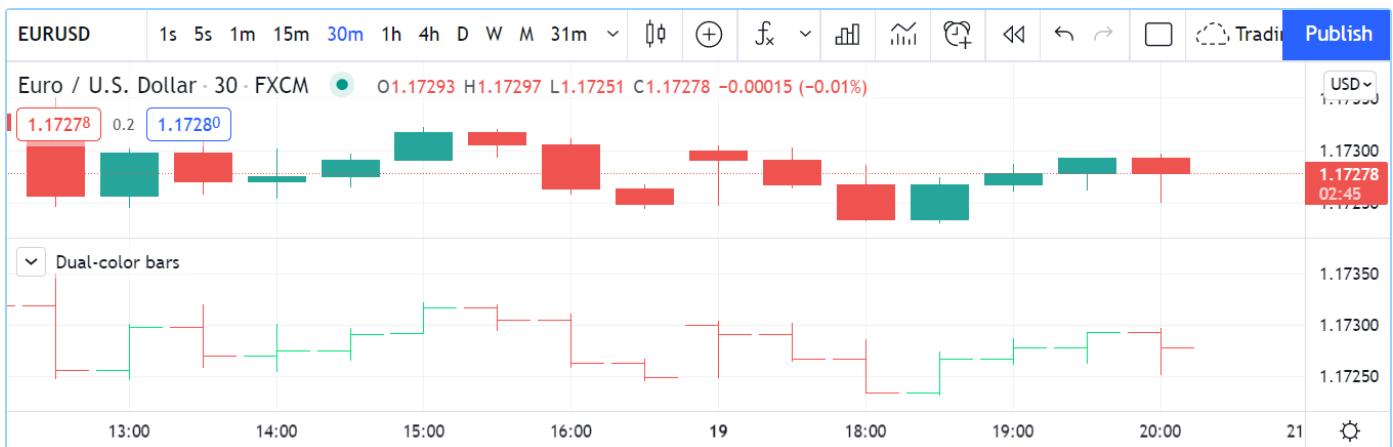
`plotbar()`的签名是：

```
plotbar(open, high, low, close, title, color, editable, show_last, display) → void
```

请注意，`plotbar()` 没有 `bordercolor` 或参数 `wickcolor`，因为传统条形图上没有边框或影线。

这使用与上一节的第二个示例相同的着色逻辑绘制传统条形：

```
//@version=5
indicator("Dual-color bars")
paletteColor = close >= open ? color.lime : color.red
plotbar(open, high, low, close, color = paletteColor)
```



## 条形图状态

### 介绍

命名空间中的一组内置变量 `barstate` 允许您的脚本检测当前正在执行脚本的栏的不同属性。

这些状态可用于将代码的执行或逻辑限制到特定的栏。

一些内置函数返回当前柱所属交易时段的信息。它们在[会话状态](#)部分进行了解释。

### 条形状态内置变量

请注意，虽然指标和库实时运行所有价格或交易量更新，但不使用的策略则 `calc_on_every_tick` 不会；它们只会在实时栏关闭时执行。这将影响该类型脚本中条形状态的检测。例如，在开放市场上，此代码在实时关闭之前不会显示背景，因为那是策略运行的时间：

```
//@version=5
strategy("S")
bgcolor(barstate.islast ? color.silver : na)
```

#### [barstate.isfirst](#)

[barstate.isfirst](#) 仅 `true` 位于数据集的第一个柱上，即当 `bar_index` 为零时。

仅初始化第一个柱上的变量可能很有用，例如：

```
// Declare array and set its values on the first bar only.
FILL_COLOR = color.green
var fillColors = array.new_color(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill
    color.
    array.push(fillColors, color.new(FILL_COLOR, 70))
    array.push(fillColors, color.new(FILL_COLOR, 75))
    array.push(fillColors, color.new(FILL_COLOR, 80))
    array.push(fillColors, color.new(FILL_COLOR, 85))
    array.push(fillColors, color.new(FILL_COLOR, 90))
```

## barstate.islast

barstate.islast 表示 `true` 当前柱是否为图表上的最后一个，无论该柱是否为实时柱。

它可用于将代码的执行限制在图表的最后一个柱，这在绘制线条、标签或表格时通常很有用。在这里，我们用它来确定何时更新我们只想出现在最后一个栏上的标签。我们只创建标签一次，然后使用 `label.set_*` 函数更新其属性，因为它更有效：

```
//@version=5
indicator("", "", true)
// Create label on the first bar only.
var label hiLabel = label.new(na, na, "")
// Update the label's position and text on the last bar,
// including on all realtime bar updates.
if barstate.islast
    label.set_xy(hiLabel, bar_index, high)
    label.set_text(hiLabel, str.tostring(high, format.mintick))
```

## barstate.ishistory

barstate.ishistory 存在 `true` 于所有历史酒吧中。当 barstate.isrealtime 也为 `true` 时，它永远不会出现 `true` 在柱上，并且当 barstate.isconfirmed 变为 `true` 时，它不会出现在实时柱的收盘更新上。在封闭市场上，它可以位于 barstate.islast 所在的同一柱上。`true``true``true``true``true`

## barstate.isrealtime

barstate.isrealtime 表示 `true` 当前数据更新是否是实时柱更新，`false` 否则（因此是历史数据）。请注意，barstate.islast 也出现 `true` 在所有实时柱上。

## barstate.isnew

barstate.isnew 出现 `true` 在所有历史柱和实时柱的第一次（开盘）更新上。

所有历史柱都被视为新柱，因为 Pine Script™ 运行时在每个柱上按顺序执行脚本，从图表的第一个柱到最后一个柱。因此，您的脚本在执行时会逐条发现每个历史柱。

[barstate.isnew](#) 可用于在新的实时柱出现时重置 `varip updateNo` 变量。以下代码将在所有历史柱上以及每个实时柱的开头重置为 1。它计算每个实时柱期间实时更新的数量：

```
//@version=5
indicator("")
updateNo() =>
    varip int updateNo = na
    if barstate.isnew
        updateNo := 1
    else
        updateNo += 1
plot(updateNo())
```

## [barstate.isconfirmed](#)

[barstate.isconfirmed](#) 位于 `true` 所有历史柱和实时柱的最后（收盘）更新上。

通过要求在条件变为 `之前` 关闭实时栏来避免重新绘制可能很有用 `true`。我们在这里使用它来保存 RSI 的绘图，直到实时柱关闭并成为经过的实时柱。它将绘制在历史柱上，因为 [barstate.isconfirmed](#) 始终 `true` 位于历史柱上：

```
//@version=5
indicator("")
myRSI = ta.rsi(close, 20)
plot(barstate.isconfirmed ? myRSI : na)
```

[barstate.isconfirmed](#) 在 `request.security()` 调用 中使用时将不起作用。

## [barstate.islastconfirmedhistory](#)

[barstate.islastconfirmedhistory](#) 是 `true` 指脚本是否在市场收盘时在数据集的最后一个柱上执行，或者在市场开盘时在实时柱之前的柱上执行。

它可用于检测第一个实时柱 `barstate.islastconfirmedhistory[1]`，或将服务器密集型计算推迟到最后一个历史柱，否则在公开市场上将无法检测到。

## 例子

以下是使用变量的脚本示例 `barstate.*`：

```
//@version=5
indicator("Bar States", overlay = true, max_labels_count = 500)

stateText() =>
    string txt = ""
    txt += barstate.isfirst ? "isfirst\n" : ""
    txt += barstate.islast ? "islast\n" : ""
    txt += barstate.ishistory ? "ishistory\n" : ""
    txt += barstate.isrealtime ? "isrealtime\n" : ""
```

```

txt += barstate.isnew ? "isnew\n" : ""
txt += barstate.isconfirmed ? "isconfirmed\n" : ""
txt += barstate.islastconfirmedhistory ? "islastconfirmedhistory\n" : ""

labelColor = switch
    barstate.isfirst => color.fuchsia
    barstate.islastconfirmedhistory => color.gray
    barstate.ishistory => color.silver
    barstate.isconfirmed => color.orange
    barstate.isnew => color.red
=> color.yellow

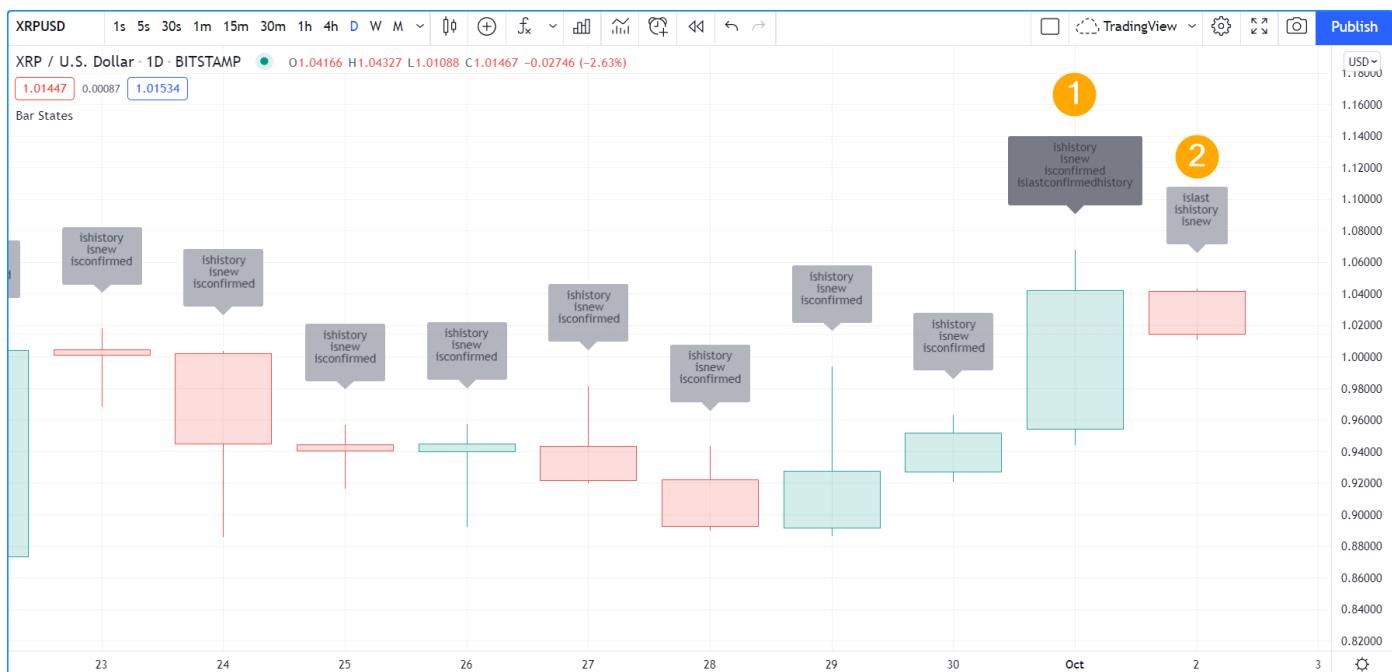
label.new(bar_index, na, stateText(), yloc = yloc.abovebar, color = labelColor)

```

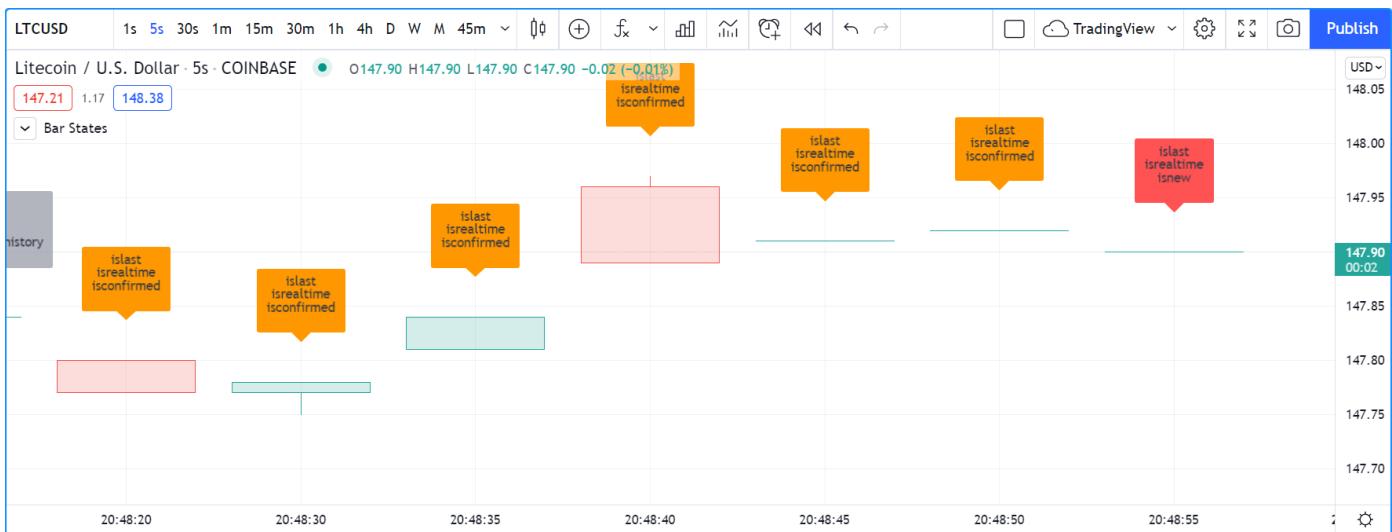
注意：

- 当为时，每个州的名称将出现在标签文本中true。
- 标签的背景有五种可能的颜色：
  - 第一个栏上紫红色
  - 历史金条上的白银
  - 最后确认的历史柱呈灰色
  - 当实时条被确认时（当它关闭并成为经过的实时条时）为橙色
  - 实时栏第一次执行时呈红色
  - 黄色表示实时栏的其他执行

我们首先将指标添加到公开市场的图表中，但在收到任何实时更新之前。请注意#1中如何识别最后一个已确认的历史柱，以及如何将最后一个柱识别为最后一个，但仍被视为历史柱，因为尚未收到实时更新。



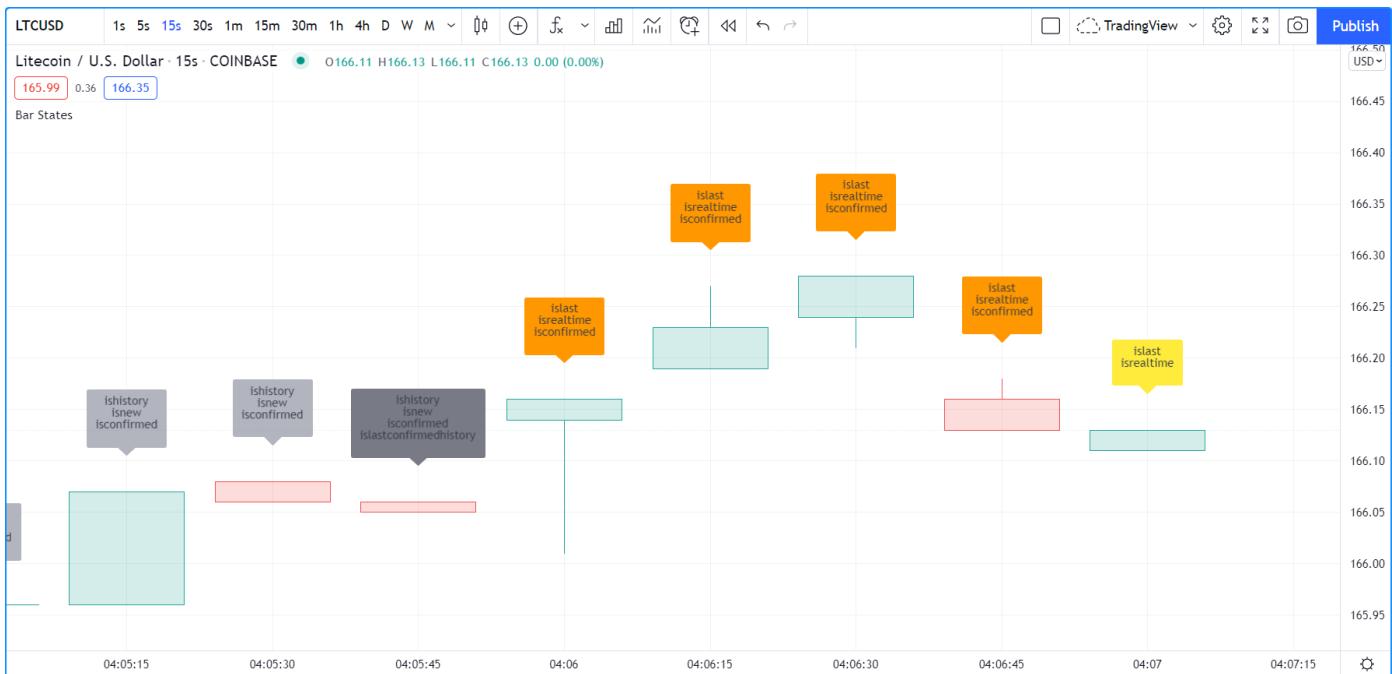
让我们看看当实时更新开始出现时会发生什么：



注意：

- 实时条是红色的，因为它是第一次执行，因为 `barstate.isnew` 是 `true` 并且 `barstate.ishistory` 不再是 `true`，所以我们的 开关 结构确定我们的颜色使用分支。这通常不会持续很长时间，因为在下一次更新时将不再如此，因此标签的颜色将变成黄色。`barstate.isnew => color.red``barstate.isnew``true`
- 经过的实时柱的标签是橙色，因为这些柱在关闭时不是历史柱。因此，switch 结构中的分支没有被执行，但下一个分支被执行。`barstate.ishistory => color.silver``barstate.isconfirmed => color.orange`

最后一个示例显示了实时栏的标签在栏上第一次执行后如何变成黄色。这是标签通常出现在实时栏上的方式：



## 图表信息

### 介绍

脚本可以通过 Pine Script™[内置变量的子集](#)获取有关其当前运行的图表和交易品种的信息。我们在这里介绍的脚本允许脚本访问以下相关信息：

- 图表的价格和成交量
- 图表的符号
- 图表的时间范围
- 交易品种交易的时段（或时间段）

## 价格和数量

---

OHLCV 值的内置变量是：

- [open](#)：酒吧的开盘价。
- [high](#)：柱形的最高价格，或实时柱形经过时间内达到的最高价格。
- [low](#)：柱形的最低价格，或实时柱形经过时间内达到的最低价格。
- [close](#)：柱的收盘价，或实时柱中的当前价格。
- [成交量](#)：柱中的交易量，或实时柱的运行时间内的交易量。体积信息的单位因仪器而异。它体现在股票的股票、外汇的手数、期货合约、加密货币的基础货币等中。

其他值可通过以下方式获得：

- [hl2](#)：柱形图[最高值](#)和[最低值](#)的平均值。
- [hlc3](#)：[柱线最高价](#)、[最低价](#)和[收盘价](#)的平均值。
- [ohlc4](#)：[柱的开盘价](#)、[最高价](#)、[最低价](#)和[收盘价](#)的平均值。

在历史柱上，上述变量的值在柱期间不会变化，因为它们仅提供 OHLCV 信息。当在历史柱上运行时，脚本在柱的收盘价处执行，此时所有柱的信息都是已知的，并且在脚本在柱上执行期间不能更改。

实时酒吧则完全是另一回事。当指标（或使用的策略）实时运行时，上述变量的值（除了[open](#)）将在实时柱上的脚本的连续迭代之间变化，因为它们代表了进程进程中某个时间点的当前值。实时酒吧。这可能会导致一种形式的重新绘制。有关更多详细信息，请参阅 Pine Script™[执行模型](#)页面。`calc_on_every_tick = true`

[历史引用运算符](#)可用于引用内置变量的过去值，例如，`close[1]`引用前一柱的[收盘价](#)（相对于脚本正在执行的特定柱）。

## 符号信息

---

命名空间中的内置变量为 `syminfo` 脚本提供有关脚本运行的图表符号的信息。每次脚本用户更改图表符号时，此信息都会更改。然后，该脚本使用内置变量的新值在所有图表的柱上重新执行：

- [syminfo.basecurrency](#)：基础货币，例如“BTCUSD”中的“BTC”，或“EURUSD”中的“EUR”。
- [syminfo.currency](#)：报价货币，例如“BTCUSD”中的“USD”，或“USDCAD”中的“CAD”。
- [syminfo.description](#)：符号的详细描述。
- [syminfo.mintick](#)：交易品种的价格变动值，或者可以变动的最小增量价格。不要与 *pips* 或 *point*混淆。关于“ES1！”（“S&P 500 E-Mini”）刻度大小为 0.25，因为这是价格变动的最小增量。

- [syminfo.pointvalue](#): 点值是确定合约价值的基础资产的倍数。关于“ES1!” (“S&P 500 E-Mini”) 点值为 50, 因此合约的价值是该工具价格的 50 倍。
- [syminfo.prefix](#): 前缀是交易所或经纪商的标识符：“NASDAQ”或“BATS”代表“AAPL”，“CME\_MINI\_DL”代表“ES1!”。
- [syminfo.root](#): 它是结构化股票代码（如期货）的股票代码前缀。“ES1!”是“ES”，“ZW1!”是“ZW”。
- [syminfo.session](#): 它反映了图表上该交易品种的会话设置。如果“图表设置/交易品种/会话”字段设置为“扩展”，则仅当交易品种和用户的提要允许扩展会话时，它才会返回“扩展”。它很少显示，主要用作[ticker.new\(\)](#) session 中参数 的参数。
- [syminfo.ticker](#): 这是交易品种的名称，不包含交易部分 ([syminfo.prefix](#)) : “BTCUSD”、“AAPL”、“ES1!”、“USDCAD”。
- [syminfo.tickerid](#): 该字符串很少显示。它主要用作 [request.security\(\)](#)参数的参数 `symbol`。它包括会话、前缀和代码信息。
- [syminfo.timezone](#): 交易品种的时区。该字符串是IANA 时区数据库名称 (例如“America/New\_York”)。
- [syminfo.type](#): 交易品种所属的市场类型。值包括“股票”、“期货”、“指数”、“外汇”、“加密货币”、“基金”、“dr”、“CFD”、“债券”、“权证”、“结构性”和“权利”。

该脚本将在图表上显示这些内置变量的值：

```
//@version=5
indicator(`syminfo.*` built-ins", "", true)
printTable(txtLeft, txtRight) =>
    var table t = table.new(position.middle_right, 2, 1)
    table.cell(t, 0, 0, txtLeft, bgcolor = color.yellow, text_align =
text.align_right)
    table.cell(t, 1, 0, txtRight, bgcolor = color.yellow, text_align =
text.align_left)

nl = "\n"
left =
    "syminfo.basecurrency: " + nl +
    "syminfo.currency: " + nl +
    "syminfo.description: " + nl +
    "syminfo.mintick: " + nl +
    "syminfo.pointvalue: " + nl +
    "syminfo.prefix: " + nl +
    "syminfo.root: " + nl +
    "syminfo.session: " + nl +
    "syminfo.ticker: " + nl +
    "syminfo.tickerid: " + nl +
    "syminfo.timezone: " + nl +
    "syminfo.type: " + nl

right =
    syminfo.basecurrency + nl +
    syminfo.currency + nl +
    syminfo.description + nl +
```

```

str.tostring(syminfo.mintick)      + nl +
str.tostring(syminfo.pointvalue)   + nl +
syminfo.prefix                   + nl +
syminfo.root                     + nl +
syminfo.session                  + nl +
syminfo.ticker                   + nl +
syminfo.tickerid                 + nl +
syminfo.timezone                + nl +
syminfo.type                     + nl +

printTable(left, right)

```

## 图表时间范围

脚本可以使用这些内置函数获取图表上使用的时间范围类型的信息，这些内置函数都返回“简单布尔”结果：

- [timeframe.isseconds](#)
- [timeframe.isminutes](#)
- [timeframe.isintraday](#)
- [timeframe.isdaily](#)
- [timeframe.isweekly](#)
- [timeframe.ismonthly](#)
- [timeframe.isdwm](#)

两个额外的内置函数返回更具体的时间范围信息：

- [timeframe.multiplier](#) 返回一个“简单 int”，其中包含时间帧单位的乘数。将返回一小时的图表时间范围，[60](#) 因为日内时间范围以分钟表示。将返回 30 秒时间范围 [30](#) (秒)、每日图表将返回 [1](#) (日)、季度图表将返回 [3](#) (月)、年度图表将返回 [12](#) (月)。此变量的值不能用作 `timeframe` 内置函数中参数的参数，因为它们需要时间范围规范格式的字符串。
- [timeframe.period](#) 返回 Pine Script™ 时间范围规范格式的字符串。

有关详细信息，请参阅[时间范围](#)页面。

## 会话信息

会话信息有多种形式：

- `syminfo.session` 内置变量返回一个 [值](#)[session.regular](#) 或 [session.extended](#)。它反映了图表上该交易品种的会话设置。如果“图表设置/交易品种/会话”字段设置为“扩展”，则仅当交易品种和用户的提要允许扩展会话时，它才会返回“扩展”。当需要会话类型时使用它，例如作为 [ticker.new\(\)](#) `session` 中参数 的参数。
- [会话状态内置](#)提供有关柱所属交易会话的信息。

# 颜色

## 介绍

脚本视觉效果对于我们在 Pine Script™ 中编写的指标的可用性起着至关重要的作用。精心设计的图表和绘图使指标更易于使用和理解。良好的视觉设计建立了一种视觉层次结构，使更重要的信息脱颖而出，而不太重要的信息则不会受到阻碍。

在 Pine 中使用颜色可以像您想要的那样简单，也可以像您的概念需要的那样复杂。Pine Script™ 中提供的 4,294,967,296 种可能的颜色和透明度组合可应用于：

- 您可以在指标的视觉空间中绘制或绘制的任何元素，无论是线条、填充、文本还是蜡烛。
- 脚本可视空间的背景，无论脚本是在其自己的窗格中运行，还是在图表上以覆盖模式运行。
- 图表上出现的条形或蜡烛主体的颜色。

脚本只能为它放置在自己的视觉空间中的元素着色。此规则的唯一例外是窗格指示器可以为图表条形图或蜡烛图着色。

Pine Script™ 具有内置颜色（例如[color.green](#)）以及[color.rgb\(\)](#)等函数，可让您动态生成 RGBA 颜色空间中的任何颜色。

## 透明度

Pine Script™ 中的每种颜色均由四个值定义：

- 其红、绿、蓝分量（0-255），遵循[RGB颜色模型](#)。
- 它的透明度（0-100），通常称为 Pine 之外的 Alpha 通道，如[RGBA颜色模型](#)中所定义。尽管透明度以 0-100 范围表示，但在函数中使用时其值可以是“浮点”，这使您可以访问 Alpha 通道的 256 个基础值。

颜色的透明度定义了它的不透明程度：0 表示完全不透明，100 表示颜色（无论它是什么）不可见。在涉及更多的颜色视觉效果或使用背景时，调节透明度至关重要，可以控制哪些颜色占主导地位，以及它们在叠加时如何混合在一起。

## Z 索引

当您将元素放置在脚本的视觉空间中时，它们在 z 轴上具有相对深度；有些会出现在其他之上。`z-index` 是表示元素在\*\*z 轴上的位置的值。`z-index` 最高的元素显示在顶部。

在 Pine Script™ 中绘制的元素被分为几组。每个组在 z 空间中都有自己的位置，并且在同一组中，脚本逻辑中最后创建的元素将出现在同一组中其他元素的顶部。一个组的元素不能放置在属于其组的 z 空间区域之外，因此图永远不会出现在表格的顶部，例如，因为表格具有最高的 z 索引。

此列表包含视觉元素组，按 z 索引递增排序，因此背景颜色始终位于 z 空间的底部，表格始终显示在所有其他元素的顶部：

- 背景颜色
- 填充
- 地块

- 线
- 线条填充
- 线路
- 盒子
- 标签
- 表格

请注意，通过使用`indicator()`或`strategy()`，您可以使用脚本中的顺序来控制、`hline()`和`fill()`视觉效果的相对z索引。`explicit_plot_zorder = true` `plot*()`

## 恒定颜色

Pine Script™ 有 17 种内置颜色。此表列出了它们的名称、十六进制等效值以及作为`color.rgb()`参数的 RGB 值：

姓名	十六进制	RGB 值
水色	#00BCD4	<code>color.rgb(0, 188, 212)</code>
黑色	#363A45	<code>color.rgb(54, 58, 69)</code>
蓝色	#2196F3	<code>color.rgb(33, 150, 243)</code>
紫红色	#E040FB	<code>color.rgb(224, 64, 251)</code>
灰色	#787B86	<code>color.rgb(120, 123, 134)</code>
绿色	#4CAF50	<code>color.rgb(76, 175, 80)</code>
石灰	#00E676	<code>color.rgb(0, 230, 118)</code>
栗色	#880E4F	<code>color.rgb(136, 14, 79)</code>
海军蓝	#311B92	<code>color.rgb(49, 27, 146)</code>
橄榄色	#808000	<code>color.rgb(128, 128, 0)</code>
橙色	#FF9800	<code>color.rgb(255, 152, 0)</code>
紫色	#9C27B0	<code>color.rgb(156, 39, 176)</code>
红色	#FF5252	<code>color.rgb(255, 82, 82)</code>
银色	#B2B5BE	<code>color.rgb(178, 181, 190)</code>
青色	#00897B	<code>color.rgb(0, 137, 123)</code>
白色	#FFFFFF	<code>color.rgb(255, 255, 255)</code>
黄色	#FFEB3B	<code>color.rgb(255, 235, 59)</code>

在下面的脚本中，所有绘图都使用相同的`color.olive` 颜色，透明度为 40，但以不同的方式表示。所有五种方法在功能上都是等效的：



```
//@version=5
indicator("", "", true)
// — Transparency (#99) is included in the hex value.
plot(ta.sma(close, 10), "10", #80800099)
// — Transparency is included in the color-generating function's arguments.
plot(ta.sma(close, 30), "30", color.new(color.olive, 40))
plot(ta.sma(close, 50), "50", color.rgb(128, 128, 0, 40))
    // — Use `transp` parameter (deprecated and advised against)
plot(ta.sma(close, 70), "70", color.olive, transp = 40)
plot(ta.sma(close, 90), "90", #808000, transp = 40)
```

## 笔记

最后两个`plot()`调用使用参数指定透明度`transp`。应避免这种使用，因为`transp`在 Pine Script™ v5 中已弃用。使用`transp`参数定义透明度不太灵活，因为它需要输入整数类型的参数，这意味着必须在执行脚本之前知道它，因此无法动态计算，因为您的脚本执行条到条。此外，如果您使用的`color`参数已经包含透明度信息（如接下来的三个`plot()`调用中所做的那样），则用于参数的任何参数`transp`都将无效。对于其他带参数的函数也是如此`transp`。

当脚本逐条执行时，上一个脚本中的颜色不会发生变化。然而，有时，需要在脚本在每个柱上执行时创建颜色，因为它们取决于编译时或脚本在零柱上开始执行时未知的条件。对于这些情况，程序员有两种选择：

1. 使用条件语句从一些预先确定的基色中选择颜色。
2. 例如，通过在脚本逐条执行时计算它们来动态构建新颜色，以实现颜色渐变。

## 条件着色

假设您想要根据您定义的某些条件，用不同的颜色对移动平均线进行着色。为此，您可以使用条件语句为每个状态选择不同的颜色。首先，当移动平均线上涨时，将其着色为牛市颜色；当移动平均线未上涨时，将其着色为熊市颜色：



```
//@version=5
indicator("Conditional colors", "", true)
int lengthInput = input.int(20, "Length", minval = 2)
color maBullColorInput = input.color(color.green, "Bull")
color maBearColorInput = input.color(color.maroon, "Bear")
float ma = ta.sma(close, lengthInput)
// Define our states.
bool maRising = ta.rising(ma, 1)
// Build our color.
color c_ma = maRising ? maBullColorInput : maBearColorInput
plot(ma, "MA", c_ma, 2)
```

注意：

- 我们为脚本用户提供牛市/熊市颜色的选择。
- 我们定义一个 `maRising` 布尔变量，`true` 当当前柱上的移动平均线高于上一个柱上时，该变量将保持不变。
- 我们定义一个 `c_ma` 颜色变量，根据布尔值分配两种颜色之一 `maRising`。我们使用[?:三元运算符](#)来编写我们的条件语句。

您还可以使用条件颜色来避免在某些条件下进行绘图。在这里，我们使用一条线绘制高枢轴和低枢轴，但我不想在新枢轴出现时绘制任何内容，以避免出现枢轴过渡中的关节。为此，我们测试枢轴变化，并在检测到变化时使用 `na` 作为颜色值，以便在该条上不绘制线条：



```
//@version=5
indicator("Conditional colors", "", true)
int legsInput = input.int(5, "Pivot Legs", minval = 1)
```

```

color pHIColorInput = input.color(color.olive, "High pivots")
color pLoColorInput = input.color(color.orange, "Low pivots")
// Initialize the pivot level variables.
var float pHi = na
var float pLo = na
// When a new pivot is detected, save its value.
pHi := nz(ta.pivothigh(legsInput, legsInput), pHi)
pLo := nz(ta.pivotlow(legsInput, legsInput), pLo)
// When a new pivot is detected, do not plot a color.
plot(pHi, "High", ta.change(pHi) ? na : pHIColorInput, 2, plot.style_line)
plot(pLo, "Low", ta.change(pLo) ? na : pLoColorInput, 2, plot.style_line)

```

要理解这段代码是如何工作的，首先必须知道[ta.pivothigh\(\)](#)和[ta.pivotlow\(\)](#)在这里使用时不带参数参数 `source`，当它们找到 高/低 枢轴时将返回一个值，否则他们回来 `na`。

当我们使用[nz\(\)](#)函数测试枢轴函数返回的值是否为 `na` 时，只有当返回的值不是 `na` 时，我们才允许将返回值分配给或变量，否则变量的先前值将简单地重新分配给它，这对其价值没有影响。请记住，`和` 的先前值会逐条保留，因为我们在初始化它们时使用 `var` 关键字，这会导致初始化仅发生在第一个条上。`pHi``pLo``pHi``pLo`

接下来要做的就是，当我们绘制线条时，插入一个三元条件语句，当主元值更改时，该语句将生成 `na` 颜色，或者当主元级别不更改时，在脚本输入中选择的颜色。

## 计算颜色

使用[color.new\(\)](#)、[color.rgb\(\)](#)和[color.from\\_gradient\(\)](#)等函数，可以在脚本逐条执行时即时构建颜色。

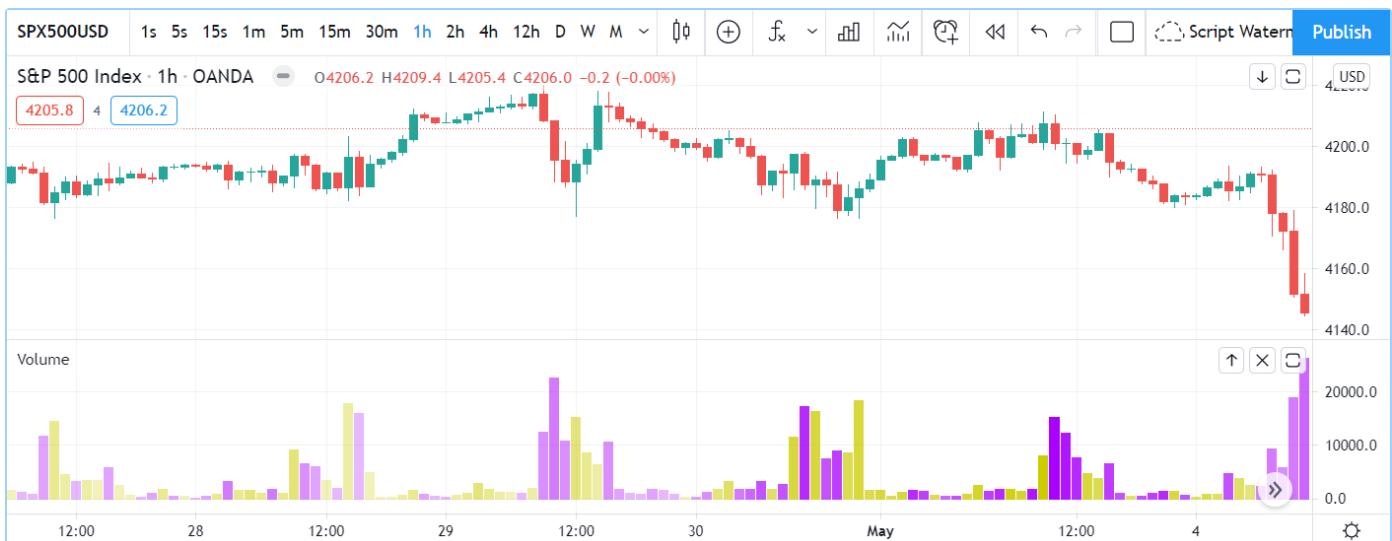
当您需要从基色生成不同的透明度级别时，[color.new\(\)最有用。](#)

当您需要从红色、绿色、蓝色或透明组件动态构建颜色时，[color.rgb\(\)](#)非常有用。当[color.rgb\(\)](#)创建颜色时，它的姊妹函数[color.r\(\)](#)、[color.g\(\)](#)、[color.b\(\)](#)和[color.t\(\)](#) 可用于提取红色、绿色、蓝色或透明度值来自颜色，该颜色又可用于生成变体。

[color.from\\_gradient\(\)](#) 对于在两个基色之间创建线性渐变非常有用。它通过根据最小值和最大值评估源值来确定要使用哪种中间颜色。

## 颜色.new()

让我们使用[color.new\(color, transp\)](#) 使用两种牛市/熊市基色之一为成交量列创建不同的透明度：



```
//@version=5
indicator("Volume")
// We name our color constants to make them more readable.
var color GOLD_COLOR      = #CCCC00ff
var color VIOLET_COLOR    = #AA00FFff
color bullColorInput = input.color(GOLD_COLOR,      "Bull")
color bearColorInput = input.color(VIOLET_COLOR, "Bear")
int levelsInput = input.int(10, "Gradient levels", minval = 1)
// We initialize only once on bar zero with `var`, otherwise the count would reset to
// zero on each bar.
var float riseFallCnt = 0
// Count the rises/falls, clamping the range to: 1 to `i_levels`.
riseFallCnt := math.max(1, math.min(levelsInput, riseFallCnt + math.sign(volume -
nz(volume[1]))))
// Rescale the count on a scale of 80, reverse it and cap transparency to <80 so that
// colors remains visible.
float transparency = 80 - math.abs(80 * riseFallCnt / levelsInput)
// Build the correct transparency of either the bull or bear color.
color volumeColor = color.new(close > open ? bullColorInput : bearColorInput,
transparency)
plot(volume, "Volume", volumeColor, 1, plot.style_columns)
```

注意：

- 在脚本的倒数第二行中，我们通过改变所使用的基色（具体取决于条形向上还是向下）和透明度级别（根据交易量的累积上升或下降计算得出）来动态计算列颜色。
- 我们为脚本用户提供不仅可以控制所使用的基本牛市/熊市颜色，还可以控制所使用的亮度级别的数量。我们使用该值来确定我们将跟踪的最大上涨或下跌次数。让用户能够管理该值，使他们能够根据他们使用的时间范围或市场调整指标的视觉效果。
- 我们小心控制所使用的最大透明度级别，使其永远不会高于 80。这确保我们的颜色始终保持一定的可见性。
- 我们还将输入中级别数的最小值设置为 1。当用户选择 1 时，成交量列将采用最大亮度的牛市或熊市颜色，或透明度为零。

## [color.rgb\(\)](#)

在下一个示例中，我们使用[color.rgb\(red, green, blue, transp\)](#)从 RGBA 值构建颜色。我们将结果用作送给朋友的节日礼物，这样他们就可以将他们的 TradingView 图表带到聚会上：



```
//@version=5
indicator("Holiday candles", "", true)
float r = math.random(0, 255)
float g = math.random(0, 255)
float b = math.random(0, 255)
float t = math.random(0, 100)
color holidayColor = color.rgb(r, g, b, t)
plotcandle(open, high, low, close, color = c_holiday, wickcolor = holidayColor,
bordercolor = c_holiday)
```

注意：

- 我们为红色、绿色和蓝色通道生成 0 到 255 范围内的值，为透明度生成 0 到 100 范围内的值。另请注意，由于[math.random\(\)](#)返回浮点值，因此浮点 0.0-100.0 范围提供对底层 Alpha 通道的完整 0-255 透明度值的访问。
- 我们使用[math.random\(min, max, seeds\)](#) 函数来生成伪随机值。我们不使用函数的第三个参数的实参：`seed`。当您想要确保函数结果的可重复性时，使用它会很方便。使用相同的种子调用，它将产生相同的值序列。

## [color.from\\_gradient\(\)](#)

我们最后的颜色计算示例将使用 [color.from\\_gradient\(value, Bottom\\_value, top\\_value, Bottom\\_color, top\\_color\)](#)。让我们首先以最简单的形式使用它，为指标版本中的 CCI 信号着色，否则该指标看起来就像内置的：



```

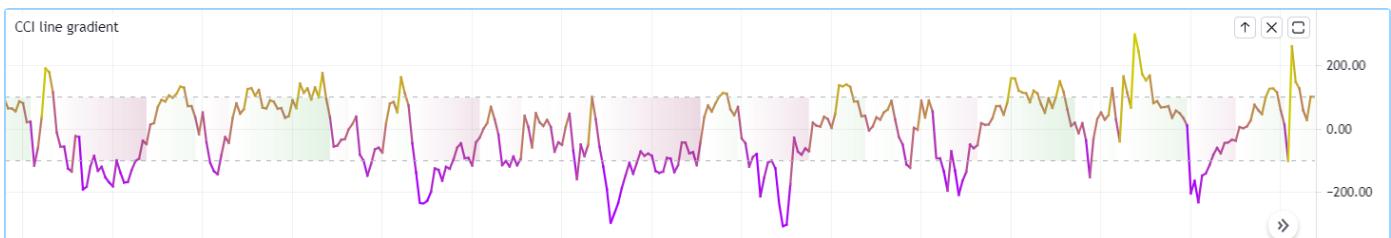
//@version=5
indicator(title="CCI line gradient", precision=2, timeframe="")
var color GOLD_COLOR      = #CCCC00
var color VIOLET_COLOR   = #AA00FF
var color BEIGE_COLOR    = #9C6E1B
float srcInput = input.source(close, title="Source")
int lenInput = input.int(20, "Length", minval = 5)
color bullColorInput = input.color(GOLD_COLOR, "Bull")
color bearColorInput = input.color(BEIGE_COLOR, "Bear")
float signal = ta.cci(srcInput, lenInput)
color signalColor = color.from_gradient(signal, -200, 200, bearColorInput,
bullColorInput)
plot(signal, "CCI", signalColor)
bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed)
bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed)
fill(bandTopPlotID, bandBotPlotID, color.new(BEIGE_COLOR, 90), "Background")

```

注意：

- 要计算渐变，`color.from_gradient()` value 需要与用于参数的参数进行比较的最小值和最大值。事实上，我们想要像 CCI 这样的无界信号的梯度（即没有固定边界，例如 RSI，它总是在 0-100 之间振荡），但这并不意味着我们不能使用`color.from_gradient()`。在这里，我们通过提供 -200 和 200 值作为参数来解决我们的难题。它们并不代表 CCI 的真实最小值和最大值，但它们处于我们不介意颜色不再变化的水平，因为每当系列超出 `bottom_value` 和 `top_value` 限制时，所使用的颜色 `bottom_color` 和 `top_color` 将适用。
- `color.from_gradient()` 计算的颜色渐变是线性的。如果该系列的值介于 `bottom_value` 和 `top_value` 参数之间，则生成的颜色的 RGBA 分量也将介于 `bottom_color` 和 `top_color` 之间。
- 许多常见的指标计算可在 Pine Script™ 中作为内置函数使用。这里我们使用`ta.cci()`而不是长期计算它。

`color.from_gradient()` value 中使用的参数 不一定是我们正在计算的线的值。只要可以提供的参数，就可以使用我们想要的任何东西。在这里，由于信号一直高于/低于中心线，我们通过使用条形数量对带进行着色来增强 CCI 指标： `bottom_value`^`top_value`



```

//@version=5
indicator(title="CCI line gradient", precision=2, timeframe="")
var color GOLD_COLOR      = #CCCC00
var color VIOLET_COLOR   = #AA00FF
var color GREEN_BG_COLOR = color.new(color.green, 70)
var color RED_BG_COLOR   = color.new(color.maroon, 70)
float srcInput      = input.source(close, "Source")
int lenInput       = input.int(20, "Length", minval = 5)
int stepsInput     = input.int(50, "Gradient levels", minval = 1)

```

```

color bullColorInput = input.color(GOLD_COLOR, "Line: Bull", inline = "11")
color bearColorInput = input.color(VIOLET_COLOR, "Bear", inline = "11")
color bullBgColorInput = input.color(GREEN_BG_COLOR, "Background: Bull", inline = "12")
color bearBgColorInput = input.color(RED_BG_COLOR, "Bear", inline = "12")

// Plot colored signal line.
float signal = ta.cci(srcInput, lenInput)
color signalColor = color.from_gradient(signal, -200, 200, color.new(bearColorInput,
0), color.new(bullColorInput, 0))
plot(signal, "CCI", signalColor, 2)

// Detect crosses of the centerline.
bool signalX = ta.cross(signal, 0)
// Count no of bars since cross. Capping it to the no of steps from inputs.
int gradientStep = math.min(stepsInput, nz(ta.barssince(signalX)))
// Choose bull/bear end color for the gradient.
color endColor = signal > 0 ? bullBgColorInput : bearBgColorInput
// Get color from gradient going from no color to `c_endColor`
color bandColor = color.from_gradient(gradientStep, 0, stepsInput, na, endColor)
bandTopPlotID = hline(100, "Upper Band", color.silver, hline.style_dashed)
bandBotPlotID = hline(-100, "Lower Band", color.silver, hline.style_dashed)
fill(bandTopPlotID, bandBotPlotID, bandColor, title = "Band")

```

注意：

- 信号图使用与我们之前的示例相同的基色和渐变。然而，我们将线条的宽度从默认的 1 增加到 2。它是我们视觉效果中最重要的组成部分；增加其宽度是使其更加突出的一种方式，并确保用户不会被带子分散注意力，因为带子比原来的扁平米色颜色变得更加繁忙。
- 由于两个原因，填充物必须保持不显眼。首先，它对于视觉效果来说是次要的，因为它提供了补充信息，即信号处于牛市/熊市区域的持续时间。其次，由于填充的 z 索引大于绘图，因此填充将覆盖信号图。由于这些原因，我们将填充的基色设置为相当透明，为 70，这样它们就不会遮盖绘图。用于带的渐变从根本没有颜色开始（请参阅在 [color.from\\_gradient\(\)](#) 调用中用作参数的 `na`），然后从输入转到基础牛市/熊市颜色，其中条件颜色变量包含。`bottom_color`^c_endColor`
- 我们为用户提供不同的牛市/熊市颜色选择。
- 当我们计算 `gradientStep` 变量时，我们在 [ta.barssince\(\)](#) 上使用 `nz()`，因为在数据集的早期柱中，当测试的条件尚未发生时，[ta.barssince\(\)](#) 将返回 `na`。因为我们使用 `nz()`，所以在这些情况下返回的值将替换为零。

## 混合透明胶片

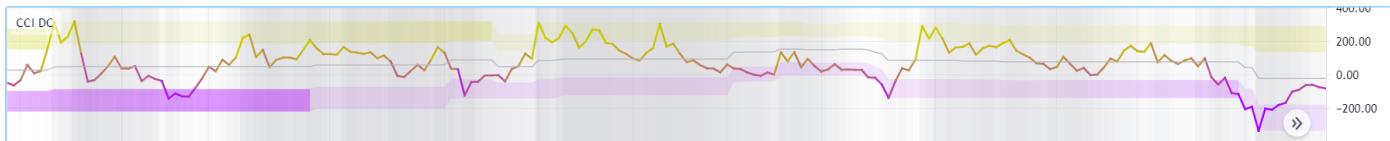
在此示例中，我们将 CCI 指标转向另一个方向。我们将使用根据 CCI 计算的唐奇安通道（历史高点/低点）构建动态调整极值区域缓冲区。我们通过将顶部/底部带设为 DC 高度的 1/4 来构建它们。我们将使用动态调整回顾来计算 DC。为了调整回顾，我们将通过保持短期 ATR 与长期 ATR 的比率来计算波动率的简单衡量标准。当该比率高于最后 100 个值中的 50 时，我们认为波动性很高。当波动性高/低时，我们减少/增加回溯。

我们的目标是为指标的用户提供：

- 正如我们在最近的示例中所示，CCI 线使用牛市/熊市梯度着色。
- 唐奇安通道的顶部和底部带的填充方式使得它们的颜色随着历史高点/低点变得越来越旧而变暗。

- 一种了解波动率测量状态的方法，我们将通过用一种颜色绘制背景来实现，当波动率增加时，该颜色的强度会增加。

这是我们的指标使用浅色主题时的样子：



以及黑暗主题：



```
//@version=5
indicator("CCI DC", precision = 6)
color GOLD_COLOR = #CCCC00ff
color VIOLET_COLOR = #AA00FFFF
int lengthInput = input.int(20, "Length", minval = 5)
color bullColorInput = input.color(GOLD_COLOR, "Bull")
color bearColorInput = input.color(VIOLET_COLOR, "Bear")

// ----- Function clamps `val` between `min` and `max`.
clamp(val, min, max) =>
    math.max(min, math.min(max, val))

// ----- Volatility expressed as 0-100 value.
float v = ta.atr(lengthInput / 5) / ta.atr(lengthInput * 5)
float vPct = ta.percentrank(v, lengthInput * 5)

// ----- Calculate dynamic lookback for DC. It increases/decreases on low/high
// volatility.
bool highVolatility = vPct > 50
var int lookBackMin = lengthInput * 2
var int lookBackMax = lengthInput * 10
var float lookBack = math.avg(lookBackMin, lookBackMax)
lookBack += highVolatility ? -2 : 2
lookBack := clamp(lookBack, lookBackMin, lookBackMax)

// ----- Dynamic lookback length Donchian channel of signal.
float signal = ta.cci(close, lengthInput)
// `lookBack` is a float; need to cast it to int to be used a length.
float hiTop = ta.highest(signal, int(lookBack))
float loBot = ta.lowest(signal, int(lookBack))
// Get margin of 25% of the DC height to build high and low bands.
float margin = (hiTop - loBot) / 4
float hiBot = hiTop - margin
float loTop = loBot + margin
// Center of DC.
```

```

float center = math.avg(hiTop, loBot)

// ----- Create colors.
color signalColor = color.from_gradient(signal, -200, 200, bearColorInput,
bullColorInput)
// Bands: Calculate transparencies so the longer since the hi/lo has changed,
//         the darker the color becomes. Cap highest transparency to 90.
float hiTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(hiTop)) + 1)) /
255), 60, 90)
float loTransp = clamp(100 - (100 * math.max(1, nz(ta.barssince(ta.change(loBot)) + 1)) /
255), 60, 90)
color hiColor = color.new(bullColorInput, hiTransp)
color loColor = color.new(bearColorInput, loTransp)
// Background: Rescale the 0-100 range of `vPct` to 0-25 to create 75-100
transparencies.
color bgColor = color.new(color.gray, 100 - (vPct / 4))

// ----- Plots
// Invisible lines for band fills.
hiTopPlotID = plot(hiTop, color = na)
hiBotPlotID = plot(hiBot, color = na)
loTopPlotID = plot(loTop, color = na)
loBotPlotID = plot(loBot, color = na)
// Plot signal and centerline.
p_signal = plot(signal, "CCI", signalColor, 2)
plot(center, "Centerline", color.silver, 1)

// Fill the bands.
fill(hiTopPlotID, hiBotPlotID, hiColor)
fill(loTopPlotID, loBotPlotID, loColor)

// ----- Background.
bgcolor(bgColor)

```

注意：

- 我们将背景的透明度限制在 100-75 的范围内，这样它就不会压倒一切。我们还使用不会过多分散注意力的中性颜色。背景越暗，我们衡量的波动性就越高。
- 我们还将带填充的透明度值限制在 60 和 90 之间。我们使用 90，以便当找到新的高/低并且渐变重置时，起始透明度使颜色有些可见。我们不使用低于 60 的透明度，因为我们不希望这些波段隐藏信号线。
- 我们使用非常方便的`ta.percentrank()`函数根据衡量波动性的 ATR 比率生成 0-100 的值。将比例未知的值转换为可用于生成透明度的已知值非常有用。
- 因为我们必须在脚本中对值进行三次钳位，所以我们编写了一个`f_clamp()`函数，而不是对逻辑进行三次显式编码。

## 尖端

## 设计可用的配色方案

如果您编写供其他交易者使用的脚本，请尽量避免在某些环境中无法正常工作的颜色，无论是用于绘图、标签、表格还是填充。至少，测试您的视觉效果，以确保它们在浅色和深色 TradingView 主题下都能令人满意地运行；它们是最常用的。例如，应避免使用黑色和白色等颜色。

构建适当的输入，为脚本用户提供灵活性，使脚本的视觉效果适应其特定环境。

请注意构建您使用的颜色的视觉层次结构，以匹配脚本视觉组件的相对重要性。优秀的设计师了解如何实现颜色和重量的最佳平衡，因此眼睛自然会被设计中最重要的元素所吸引。当你让一切都脱颖而出时，什么都不会。通过淡化周围的视觉效果，为某些元素腾出空间来脱颖而出。

在输入中提供一系列颜色预设（而不是可以更改的单一颜色）可以帮助有颜色问题的用户。我们的[技术评级](#)展示了实现这一目标的一种方法。

## 绘制清晰的线条

最好使用零透明度来绘制视觉效果中的重要线条，以保持它们清晰。这样，它们将通过填充更精确地显示。请记住，填充的 z-index 高于绘图，因此它们放置在绘图之上。稍微增加线条宽度也可以使其脱颖而出。

如果您希望某个特殊图脱颖而出，您还可以通过在同一条线上使用多个图来赋予其更多重要性。这些是我们调整绘图的连续宽度和透明度以实现此目的的示例：



```
//@version=5
indicator("")
plot(high, "", color.new(color.orange, 80), 8)
plot(high, "", color.new(color.orange, 60), 4)
plot(high, "", color.new(color.orange, 00), 1)

plot(hl2, "", color.new(color.orange, 60), 4)
plot(hl2, "", color.new(color.orange, 00), 1)

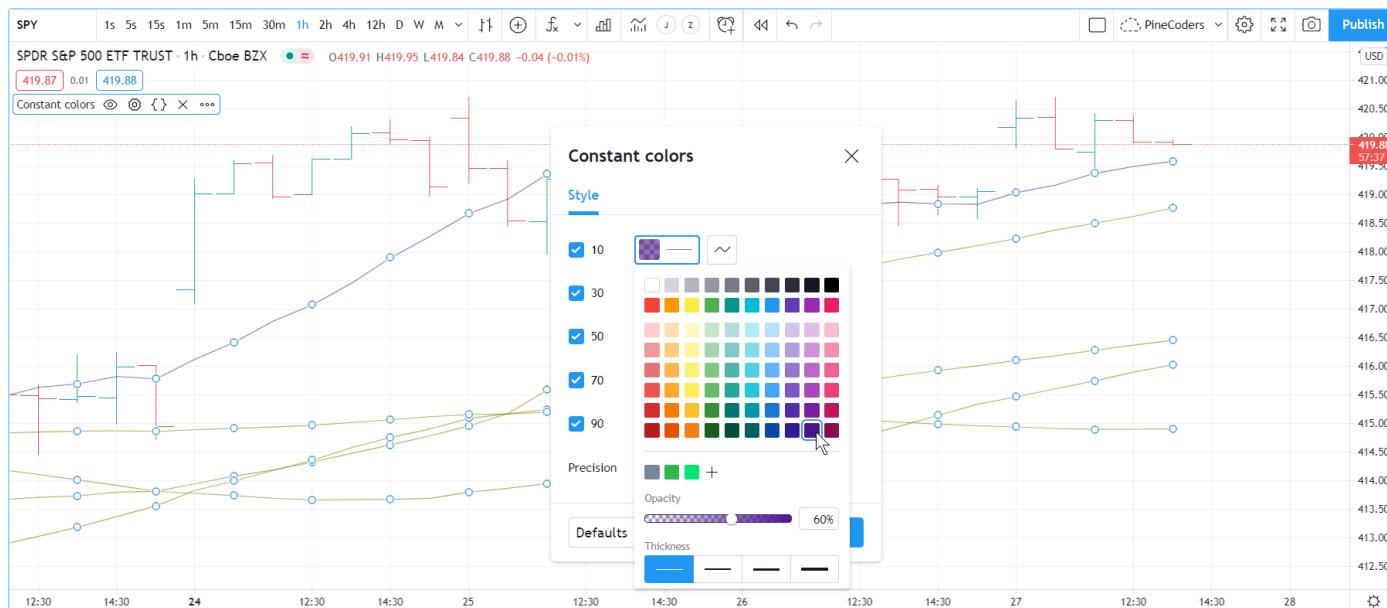
plot(low, "", color.new(color.orange, 0), 1)
```

## 自定义渐变

构建渐变时，请使其适应所应用的视觉效果。例如，如果您使用渐变为蜡烛着色，通常最好将渐变的步数限制为十或更少，因为眼睛更难以感知离散对象的强度变化。正如我们在示例中所做的那样，限制最小和最大透明度级别，以便您的视觉元素保持可见，并且在必要时不会淹没。

## 通过脚本设置选择颜色

您在脚本中使用的颜色类型会影响脚本用户如何更改脚本视觉效果的颜色。只要您不使用必须在运行时计算 RGBA 分量的颜色，脚本用户就可以通过转到脚本的“设置/样式”选项卡来修改您使用的颜色。本页上的第一个示例脚本满足该标准，以下屏幕截图显示了我们如何使用脚本的“设置/样式”选项卡来更改第一个移动平均线的颜色：

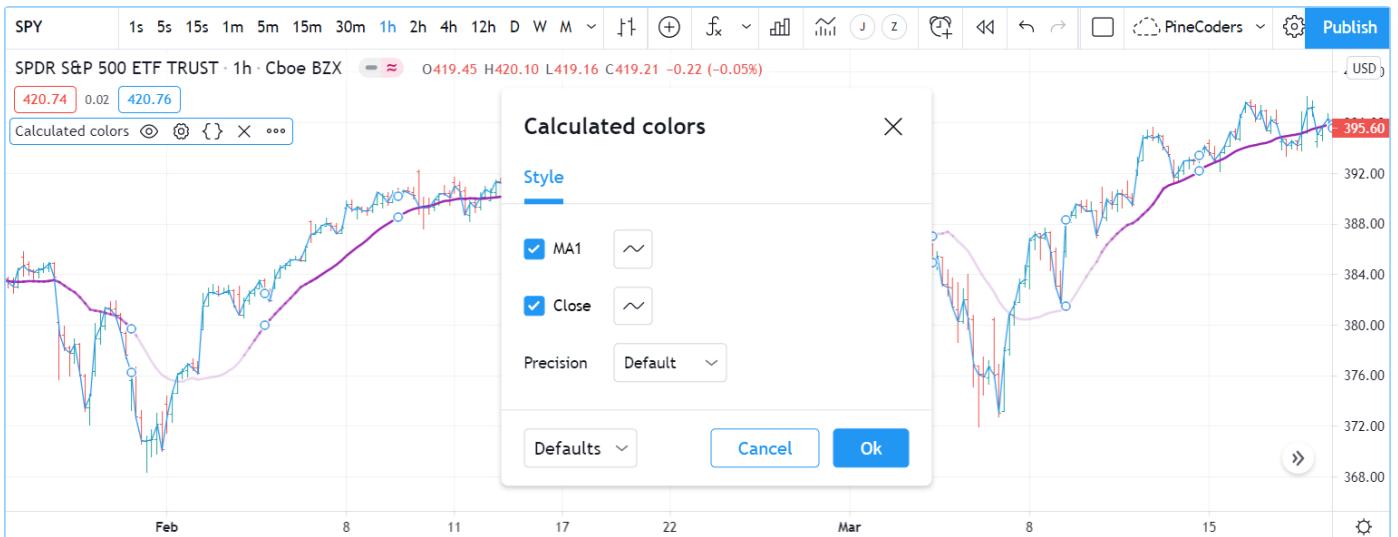


如果您的脚本使用计算的颜色，即一种颜色，其中至少一个 RGBA 分量只能在运行时知道，那么“设置/样式”选项卡将不会为用户提供可用于修改绘图的常用颜色小部件颜色。不使用计算颜色的同一脚本的绘图也会受到影响。例如，在此脚本中，我们的第一个 `plot()` 调用使用计算出的颜色，而第二个则不使用：

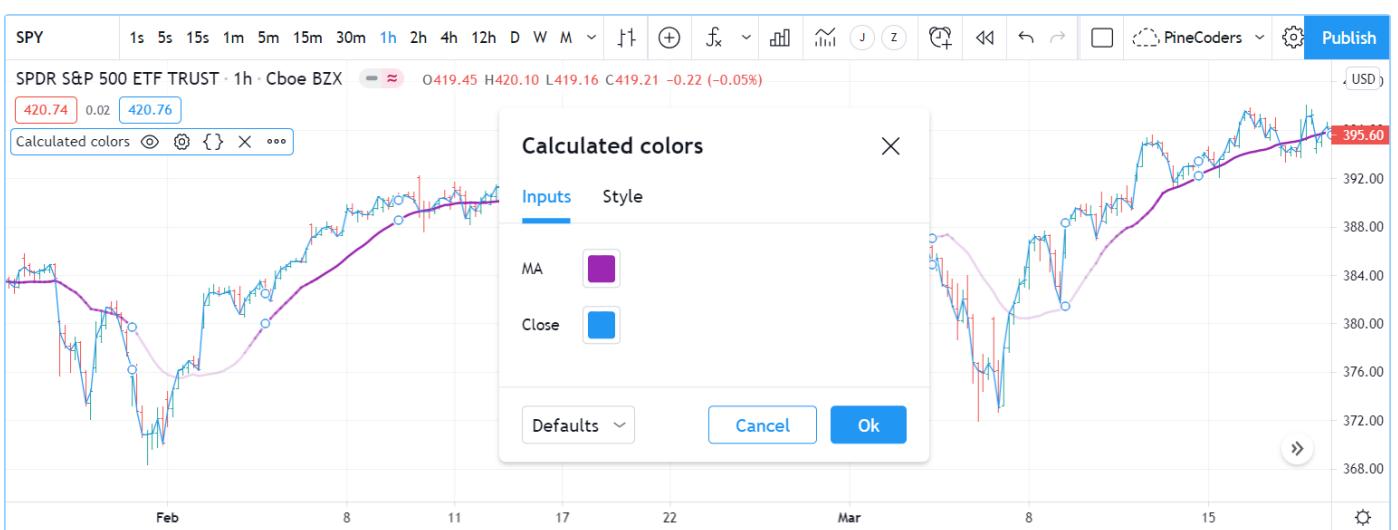
```
//@version=5
indicator("Calculated colors", "", true)
float ma = ta.sma(close, 20)
float maHeight = ta.percentrank(ma, 100)
float transparency = math.min(80, 100 - maHeight)
// This plot uses a calculated color.
plot(ma, "MA1", color.rgb(156, 39, 176, transparency), 2)
// This plot does not use a calculated color.
plot(close, "Close", color.blue)
```

第一个图中使用的颜色是计算出的颜色，因为它的透明度只能在运行时知道。它是使用移动平均线相对于其过去 100 个值的相对位置来计算的。过去值低于当前值的百分比越大，0-100 的值 `maHeight` 就越高。由于我们希望颜色在 100 时最暗 `maHeight`，因此我们从中减去 100 以获得零透明度。我们还将计算 `transparency` 值限制为最大值 80，以便它始终保持可见。

由于我们的脚本中使用了计算出的颜色，因此“设置/样式”选项卡将不会显示任何颜色小部件：



使脚本用户能够控制所使用的颜色的解决方案是为他们提供自定义输入，就像我们在这里所做的那样：



```
//@version=5
indicator("Calculated colors", "", true)
color maInput = input.color(color.purple, "MA")
color closeInput = input.color(color.blue, "Close")
float ma = ta.sma(close, 20)
float maHeight = ta.percentrank(ma, 100)
float transparency = math.min(80, 100 - maHeight)
// This plot uses a calculated color.
plot(ma, "MA1", color.new(maInput, transparency), 2)
// This plot does not use a calculated color.
plot(close, "Close", closeInput)
```

请注意我们的脚本的“设置”现在如何显示“输入”选项卡，我们在其中创建了两个颜色输入。第一个使用`color.purple`作为默认值。无论脚本用户是否更改该颜色，它都将在`color.new()`调用中使用，以在`plot()`调用中生成计算出的透明度。第二个输入使用我们之前在`plot()`调用中使用的内置`color.blue`作为默认颜色，并且只需在第二个`plot()`调用中使用它即可。

# 填充

## 介绍

有两种不同的机制专门用于填充 Pine 视觉效果之间的空间：

- [fill \(\)](#)函数允许您为使用[plot\(\)](#)绘制的两个图或使用[hline\(\)](#)绘制的两条水平线之间的背景着色。
- [linefill.new \(\)](#)函数填充使用[line.new\(\)](#)创建的行之间的空间。

## [plot\(\)](#) 和 [hline\(\)](#) 填充

[fill \(\)](#)函数有两个签名：

```
fill(plot1, plot2, color, title, editable, show_last, fillgaps) → void  
fill(hline1, hline2, color, title, editable, fillgaps) → void
```

用于 `plot1`、`plot2` 和参数 的参数必须是由[plot\(\)](#) `hline1` 和[hline\(\)](#) `hline2` 调用返回的ID。 [fill \(\)](#)函数是唯一使用这些 ID 的内置函数。

请参阅第一个示例，了解如何在、、、和、和变量中捕获由[plot\(\)](#)和[hline\(\)](#)调用返回的 ID，以便重用为[fill\(\)](#)参数：`p1``p2``p3``h1``h2``h3``h4`



```
//@version=5
indicator("Example 1")
p1 = plot(math.sin(high))
p2 = plot(math.cos(low))
p3 = plot(math.sin(close))
fill(p1, p3, color.new(color.red, 90))
fill(p2, p3, color.new(color.blue, 90))
h1 = hline(0)
h2 = hline(1.0)
h3 = hline(0.5)
h4 = hline(1.5)
fill(h1, h2, color.new(color.yellow, 90))
fill(h3, h4, color.new(color.lime, 90))
```

由于`fill()`需要来自同一函数的两个 ID，因此我们有时需要使用`plot()`调用，否则我们会使用`hline()`调用，如下例所示：



```
//@version=5
indicator("Example 2")
src = close
ma = ta.sma(src, 10)
osc = 100 * (ma - src) / ma
oscPlotID = plot(osc)
// An `hline()` would not work here because two `plot()` calls are needed.
zeroPlotID = plot(0, "Zero", color.silver, 1, plot.style_circles)
fill(oscPlotID, zeroPlotID, color.new(color.blue, 90))
```

因为“系列颜色”可以用作`fill()``color`中的参数的参数，所以您可以使用像或之类的常量，以及计算每个条形上颜色的表达式，如下例所示：`color.red``#FF001A`



```
//@version=5
indicator("Example 3", "", true)
line1 = ta.sma(close, 5)
line2 = ta.sma(close, 20)
p1PlotID = plot(line1)
p2PlotID = plot(line2)
fill(p1PlotID, p2PlotID, line1 > line2 ? color.new(color.green, 90) :
color.new(color.red, 90))
```

## 线条填充

线条填充是允许您填充通过[line.new\(\)](#)函数创建的两个线条图之间的空间的对象。当调用[linefill.new\(\)](#)函数时，图表上会显示一个 `linefill` 对象。该函数具有以下签名：

```
linefill.new(line1, line2, color) → series linefill
```

和参数是要在其间填充 `line1` 的 `line2` 两行的行 ID。参数 `color` 是填充的颜色。任何两行对之间只能有一个行填充，因此在同一对行上连续调用[linefill.new\(\)](#)会将前一个行填充替换为新的行填充。该函数返回它创建的对象的 ID `linefill`，该 ID 可以保存在变量中，以便在[linefill.set\\_color\(\)](#)调用中使用，该调用将更改现有线条填充的颜色。

行填充的行为取决于它们所附加的行。无法直接移动换行符；它们的坐标遵循它们所绑定的线的坐标。如果两条线朝同一方向延伸，则线填充也会延伸。

请注意，为了使直线延伸正常工作，直线的 `x1` 坐标必须小于其 `x2` 坐标。如果一条线的 `x1` 参数大于其 `x2` 参数并被 `extend.left` 使用，则该线实际上会向右延伸，因为 `x2` 假定它是最右边的 `x` 坐标。

在下面的示例中，我们的指标绘制了两条线连接图表的最后两个高点和低点枢轴点。我们将线条向右延伸以预测图表的短期走势，并填充线条之间的空间以增强线条创建的通道的可见性：



```

//@version=5
indicator("Channel", overlay = true)

LEN_LEFT = 15
LEN_RIGHT = 5
pH = ta.pivothigh(LEN_LEFT, LEN_RIGHT)
pL = ta.pivotlow(LEN_LEFT, LEN_RIGHT)

// Bar indices of pivot points
pH_x1 = ta.valuewhen(pH, bar_index, 1) - LEN_RIGHT
pH_x2 = ta.valuewhen(pH, bar_index, 0) - LEN_RIGHT
pL_x1 = ta.valuewhen(pL, bar_index, 1) - LEN_RIGHT
pL_x2 = ta.valuewhen(pL, bar_index, 0) - LEN_RIGHT

// Price values of pivot points
pH_y1 = ta.valuewhen(pH, pH, 1)
pH_y2 = ta.valuewhen(pH, pH, 0)
pL_y1 = ta.valuewhen(pL, pL, 1)
pL_y2 = ta.valuewhen(pL, pL, 0)

if barstate.islastconfirmedhistory
    // Lines
    lH = line.new(pH_x1, pH_y1, pH_x2, pH_y2, extend = extend.right)
    lL = line.new(pL_x1, pL_y1, pL_x2, pL_y2, extend = extend.right)
    // Fill
    fillColor = switch
        pH_y2 > pH_y1 and pL_y2 > pL_y1 => color.green
        pH_y2 < pH_y1 and pL_y2 < pL_y1 => color.red
        => color.silver
    linefill.new(lH, lL, color.new(fillColor, 90))

```

# 输入

## 介绍

输入允许脚本接收用户可以更改的值。将它们用作键值将使您的脚本更适合用户首选项。

以下脚本使用绘制 20 周期简单移动平均线 (SMA)。虽然写起来很简单，但它不是很灵活，因为它只会绘制特定的 MA: `ta.sma(close, 20)`

```
//@version=5
indicator("MA", "", true)
plot(ta.sma(close, 20))
```

相反，如果我们以这种方式编写脚本，它会变得更加灵活，因为它的用户将能够选择源和他们想要用于 MA 计算的长度：

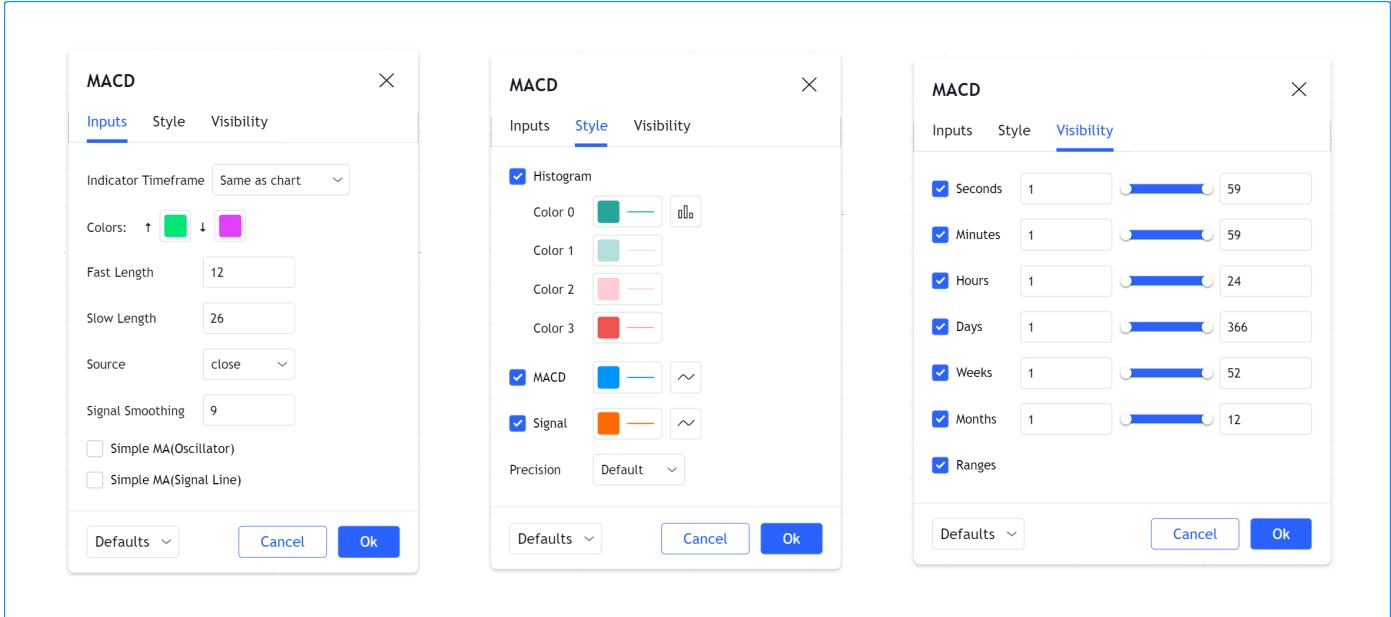
```
//@version=5
indicator("MA", "", true)
sourceInput = input(close, "Source")
lengthInput = input(20, "Length")
plot(ta.sma(sourceInput, lengthInput))
```

仅当脚本在图表上运行时才能访问输入。脚本用户通过脚本的“设置”对话框访问它们，可以通过以下任一方式访问：

- 双击图表上指标的名称
- 右键单击脚本名称并从下拉菜单中选择“设置”项
- 将鼠标悬停在图表上指标名称上时出现的“更多”菜单图标（三个点）中选择“设置”项
- 双击数据窗口中的指标名称（图表右侧下方的第四个图标）

“设置”对话框始终包含“样式”和“可见性”选项卡，允许用户指定脚本视觉效果以及脚本可见的图表时间范围的首选项。

当脚本包含对 `input.*()` 函数的调用时，“设置”对话框中会出现“输入”选项卡。



在脚本执行流程中，当脚本已位于图表上并且用户更改“输入”选项卡中的值时，将处理输入。这些更改会触发所有图表栏上脚本的重新执行，因此当用户更改输入值时，您的脚本将使用该新值重新计算。

## 输入功能

可以使用以下输入功能：

- [input\(\)](#)
- [input.int\(\)](#)
- [input.float\(\)](#)
- [input.bool\(\)](#)
- [input.颜色\(\)](#)
- [input.string\(\)](#)
- [input.timeframe\(\)](#)
- [input.symbol\(\)](#)
- [input.price\(\)](#)
- [input.source\(\)](#)
- [input.session\(\)](#)
- [input.time\(\)](#)

在“输入”选项卡中创建一个特定的输入小部件来接受每种类型的输入。除非在 `input.*()` 调用中另有指定，否则每个输入都会按照调用 `input.*()` 在脚本中出现的顺序显示在“输入”选项卡的新行中。

我们的[风格指南](#)建议将调用 `input.*()` 放在脚本的开头。

输入函数定义通常包含许多参数，这些参数允许您控制输入的默认值、它们的限制以及它们在“输入”选项卡中的组织。

调用 `input.*()` 只是 Pine Script™ 中的另一个函数调用，其结果可以与 算术、比较、逻辑或三元 运算符组合以形成要分配给变量的表达式。在这里，我们将调用 `input.string()` 的结果与 `string` 进行比较 "On"。然后表达式的結果存储在 `plotDisplayInput` 变量中。由于该变量保存一个 `true` or `false` 值，因此它是"输入 bool"类型：

```
//@version=5
indicator("Input in an expression`", "", true)
bool plotDisplayInput = input.string("On", "Plot Display", options = ["On", "Off"]) ==
"On"
plot(plotDisplayInput ? close : na)
```

除"源"值外，函数返回的所有值 `input.*()` 都是"输入"限定值。有关更多信息，请参阅我们的用户手册中有关[类型限定符](#)的部分。

## 输入函数参数

所有输入函数共有的参数是：`defval`、`title`、`tooltip` 和。其他输入函数使用一些参数：`、`、`和`。`inline``group``options``minval``maxval``step``confirm`

所有这些参数都需要"const"参数（除非它是用于"源"的输入，它返回"系列浮点数"结果）。这意味着它们必须在编译时已知，并且在脚本执行期间不能更改。由于函数的结果 `input.*()` 始终被限定为"input"或"series"，因此一个 `input.*()` 函数调用的结果不能用作后续 `input.*()` 调用中的参数，因为"input"限定符比"const"强。

让我们看一下每个参数：

- `defval` 是所有输入函数的第一个参数。这是将出现在输入小部件中的默认值。它需要一个函数所使用的输入值类型的参数。
- `title` 需要一个"const string"参数。这是该字段的标签。
- `tooltip` 需要一个"const string"参数。使用该参数时，字段右侧会出现一个问号图标。当用户将鼠标悬停在其上时，将显示工具提示的文本。请注意，如果使用 将多个输入字段分组在一行上 `inline`，则工具提示将始终显示在最右侧字段的右侧，并显示 `tooltip` 该行中使用的最后一个参数的文本。`\n` 参数字符串支持换行符`()`。
- `inline` 需要一个"const string"参数。在多个调用中使用相同的参数作为参数 `input.*()` 会将它们的输入小部件分组在同一行上。"输入"选项卡展开的宽度有限制，因此一行中可以容纳有限数量的输入字段。使用 `input.*()` 带有唯一参数的一次调用 `inline`，可以将输入字段紧接在标签之后向左移动，从而放弃在不 `inline` 使用参数时使用的所有输入字段的默认左对齐方式。
- `group` 需要一个"const string"参数。它用于将任意数量的输入分组到同一部分中。用作参数的字符串 `group` 将成为该节的标题。要组合在一起的所有 `input.*()` 调用必须使用相同的字符串作为其 `group` 参数。
- `options` 需要一个用方括号括起来的以逗号分隔的元素列表（例如，`["ON", "OFF"]`）。它用于创建一个下拉菜单，以菜单选择的形式提供列表的元素。只能选择一个菜单项。使用列表时，`value` 必须是列表的元素之一。当在允许、或的输入函数中使用时，这些参数不能同时使用。
- `minval` 需要一个"const int/float"参数，具体取决于值的类型 `defval`。它是输入字段的最小有效值。
- `maxval` 需要一个"const int/float"参数，具体取决于值的类型 `defval`。它是输入字段的最大有效值。
- `step` 是使用小部件的向上/向下箭头时字段值将移动的增量。

- `confirm` 需要一个“`const bool`” (`true` 或 `false`) 参数。此参数会影响脚本添加到图表时的行为。当脚本添加到图表时，`input.*()` 使用调用将导致弹出“设置/输入”选项卡。对于确保用户配置特定字段很有用。`confirm = true` `` `confirm`
- 、`minval` 和参数仅出现在 `input.int()` `maxval` 和 `input.float()` 函数的 `step` 签名中。

## 输入类型

接下来的部分将解释每个输入函数的作用。在我们继续进行的过程中，我们将探索使用输入功能和组织其显示的不同方式。

### 简单输入

`input()` 是一个简单的通用函数，支持基本的 Pine Script™ 类型：“int”、“float”、“bool”、“color”和“string”。它还支持“源”输入，这些输入是与价格相关的值，例如 `close`、`hl2`、`hlc3` 和 `hlcc4`，或者可用于接收另一个脚本的输出值。

它的签名是：

```
input(defval, title, tooltip, inline, group) → input int/float/bool/color/string |
series float
```

`defval` 该函数通过分析函数调用中使用的参数类型来自动检测输入的类型。此脚本显示了所有支持的类型以及与 `defval` 不同类型的参数一起使用时函数返回的限定类型：

```
//@version=5
indicator(`input()`, "", true)
a = input(1, "input int")
b = input(1.0, "input float")
c = input(true, "input bool")
d = input(color.orange, "input color")
e = input("1", "input string")
f = input(close, "series float")
plot(na)
```

### 整数输入

`input.int()` 函数存在两个签名；一个 `options` 是不使用时，另一个是：

```
input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm) → input
int
input.int(defval, title, options, tooltip, inline, group, confirm) → input int
```

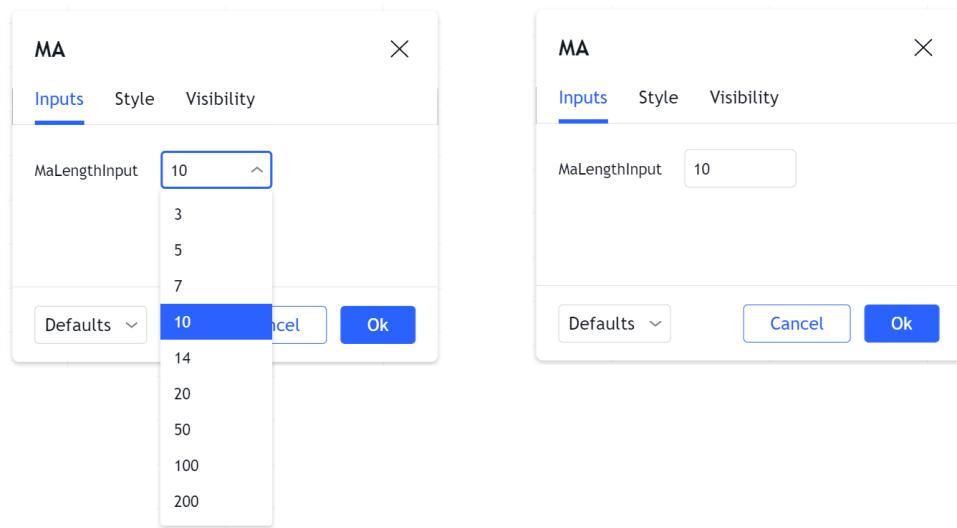
此调用使用 `options` 参数来建议 MA 的预定义长度列表：

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, options = [3, 5, 7, 10, 14, 20, 50, 100, 200])
ma = ta.sma(close, maLengthInput)
plot(ma)
```

这个使用 `minval` 参数来限制长度：

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, minval = 2)
ma = ta.sma(close, maLengthInput)
plot(ma)
```

带有列表的版本 `options` 对其小部件使用下拉菜单。当 `options` 不使用参数时，使用简单的输入小部件来输入值。



## 浮点输入

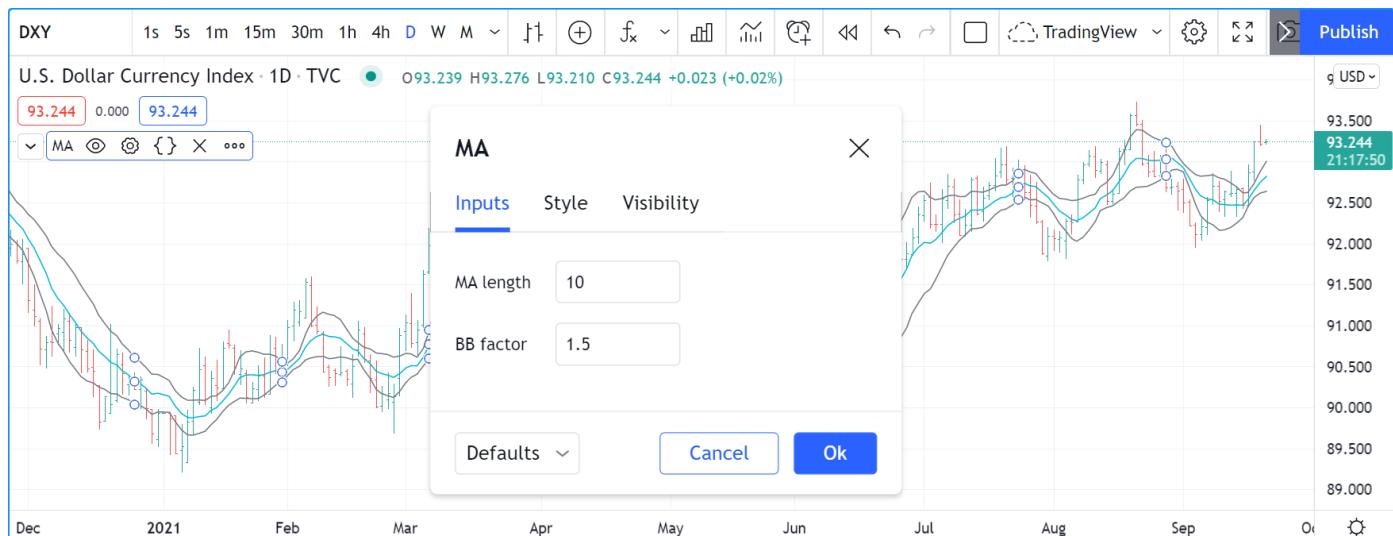
`input.float()` 函数存在两个签名；一个 `options` 是不使用时，另一个是：

```
input.int(defval, title, minval, maxval, step, tooltip, inline, group, confirm) → input int
input.int(defval, title, options, tooltip, inline, group, confirm) → input int
```

在这里，我们使用“浮点”输入作为乘以标准差的因子，以计算布林线：

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, minval = 1)
bbFactorInput = input.float(1.5, minval = 0, step = 0.5)
ma = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi = ma + bbWidth
bbLo = ma - bbWidth
plot(ma)
plot(bbHi, "BB Hi", color.gray)
plot(bbLo, "BB Lo", color.gray)
```

浮点数的输入小部件与用于整数输入的输入小部件类似。



## 布尔输入

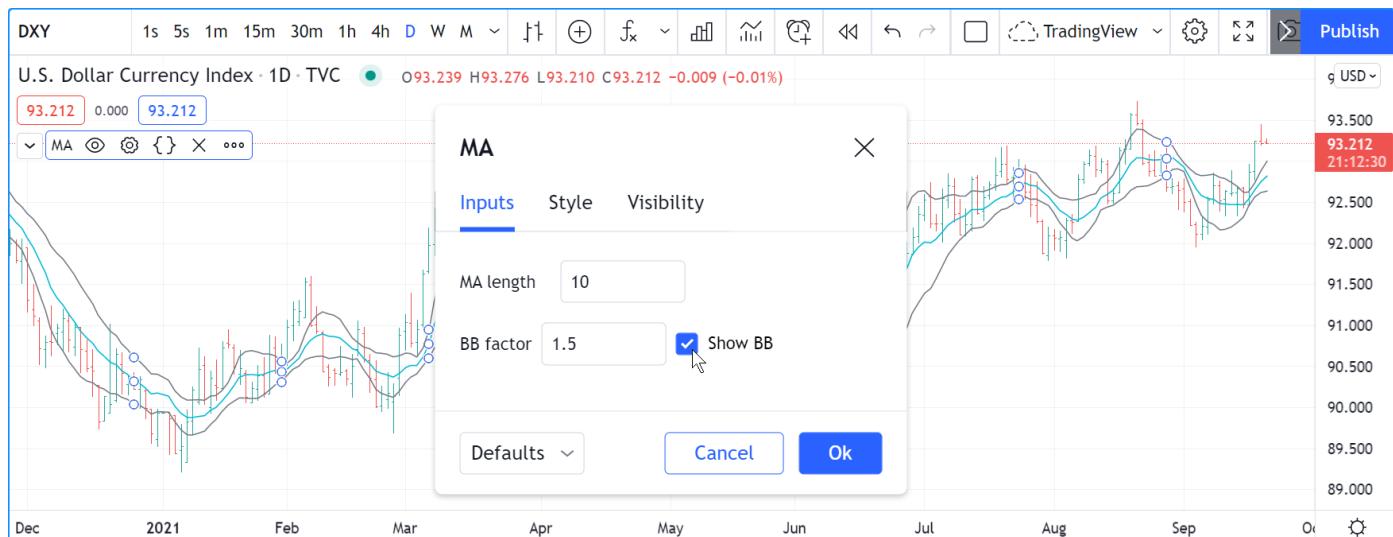
让我们继续进一步开发我们的脚本，这次添加一个布尔输入以允许用户切换 BB 的显示：

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, "MA length", minval = 1)
bbFactorInput = input.float(1.5, "BB factor", inline = "01", minval = 0, step = 0.5)
showBBInput = input.bool(true, "Show BB", inline = "01")
ma = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi = ma + bbWidth
bbLo = ma - bbWidth
plot(ma, "MA", color.aqua)
plot(showBBInput ? bbHi : na, "BB Hi", color.gray)
plot(showBBInput ? bbLo : na, "BB Lo", color.gray)
```

注意：

- 我们使用 `input.bool()` 添加了一个输入来设置的值 `showBBInput`。

- 我们使用 `inline` 该输入中的参数和 `for` 中的参数 `bbFactorInput` 将它们放在同一行。我们 "01" 在这两种情况下都使用它的参数。这就是 Pine Script™ 编译器识别它们属于同一行的方式。用作参数的特定字符串并不重要，并且不会出现在“输入”选项卡中的任何位置；它仅用于识别哪些输入位于同一行。
- 我们垂直对齐了调用 `title` 的参数 `input.*()`，以使它们更易于阅读。
- 我们 `showBBInput` 在两次 `plot()` 调用中使用该变量来有条件地绘制。当用户取消选中输入的复选框时 `showBBInput`，变量的值变为 `false`。当这种情况发生时，我们的 `plot()` 调用绘制 `na` 值，该值不显示任何内容。我们使用 `true` 作为输入的默认值，因此默认绘制 BB。
- 因为我们使用变量 `inline` 的参数 `bbFactorInput`，所以它在“输入”选项卡中的输入字段不会与 `maLengthInput` 不使用的输入字段垂直对齐 `inline`。



## 颜色输入

正如“颜色”页面的通过脚本设置进行颜色选择部分中所述，通常出现在“设置/样式”选项卡中的颜色选择并不总是可用。在这种情况下，脚本用户将无法更改脚本使用的颜色。对于这些情况，如果您希望可以通过脚本的“设置”修改脚本的颜色，则必须提供颜色输入。然后，您将允许脚本用户使用调用 `input.color()` 来更改颜色，而不是使用“设置/样式”选项卡来更改颜色。

假设当高值和低值高于/低于BB时，我们希望将BB绘制为较浅的阴影。您可以使用这样的代码来创建颜色：

```
bbHiColor = color.new(color.gray, high > bbHi ? 60 : 0)
bbLoColor = color.new(color.gray, low < bbLo ? 60 : 0)
```

当使用动态（或“系列”）颜色组件（例如此处的透明度）时，“设置/样式”中的颜色小部件将不再出现。让我们创建自己的，它将出现在我们的“输入”选项卡中：

```
//@version=5
indicator("MA", "", true)
maLengthInput = input.int(10, "MA length", inline = "01", minval = 1)
maColorInput = input.color(color.aqua, "", inline = "01")
bbFactorInput = input.float(1.5, "BB factor", inline = "02", minval = 0, step = 0.5)
bbColorInput = input.color(color.gray, "", inline = "02")
showBBInput = input.bool(true, "Show BB", inline = "02")
```

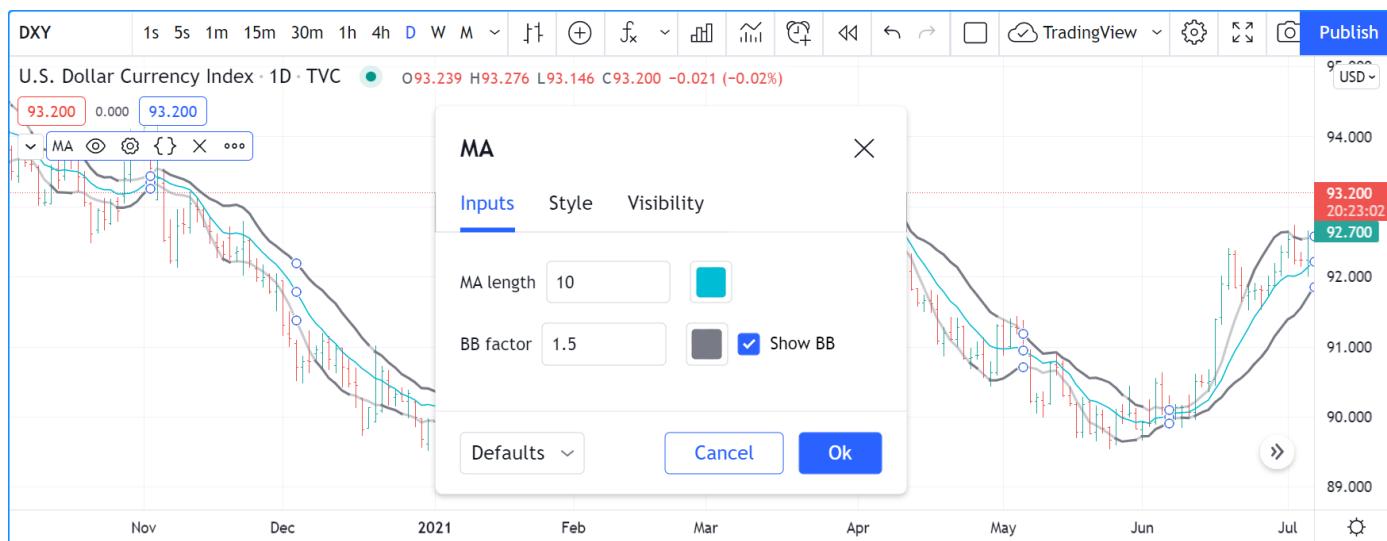
```

ma      = ta.sma(close, maLengthInput)
bbWidth = ta.stdev(ma, maLengthInput) * bbFactorInput
bbHi    = ma + bbWidth
bbLo    = ma - bbWidth
bbHiColor = color.new(bbColorInput, high > bbHi ? 60 : 0)
bbLoColor = color.new(bbColorInput, low < bbLo ? 60 : 0)
plot(ma, "MA", maColorInput)
plot(showBBInput ? bbHi : na, "BB Hi", bbHiColor, 2)
plot(showBBInput ? bbLo : na, "BB Lo", bbLoColor, 2)

```

注意：

- 我们添加了两次对`input.color()`的`maColorInput`调用来收集和变量的值`bbColorInput`。我们`maColorInput`直接在调用中使用，并用于构建和变量，这些变量使用价格相对于 BB 的位置来调节透明度。我们对调用`color.new()`的值使用条件值，以生成相同基色的不同透明度。`plot(ma, "MA", maColorInput)`~`bbColorInput`~`bbHiColor`~`bbLoColor`~`transp`
- 我们不`title`为新的颜色输入使用参数，因为它们与其他输入位于同一行，允许用户了解它们应用到哪些绘图。
- 我们重新组织了我们的`inline`论点，以便它们反映了我们的输入分为两条不同的线的事实。



## 时间范围输入

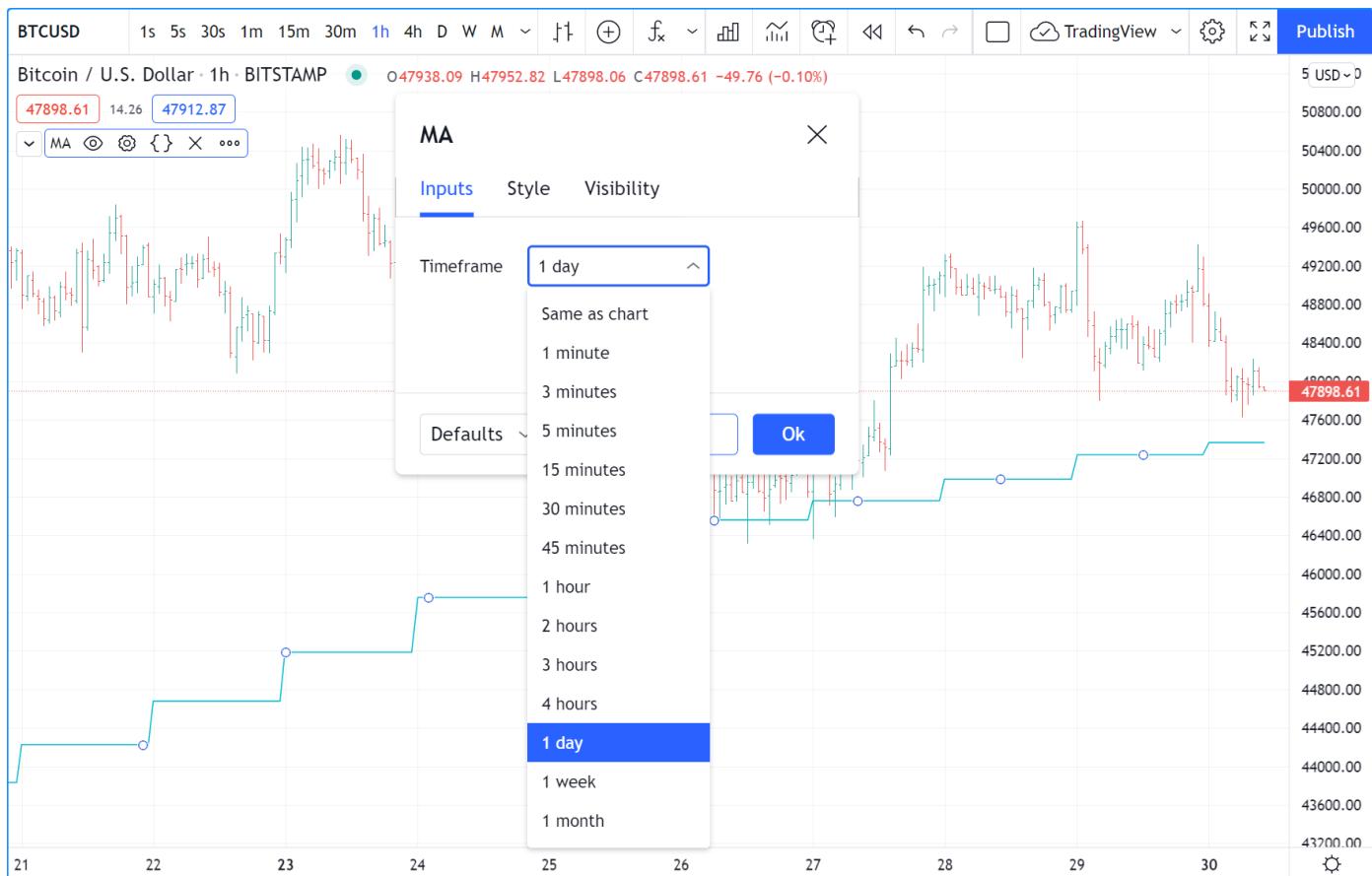
当您希望能够更改用于计算脚本中的值的时间范围时，时间范围输入会很有用。

让我们去掉前面部分中的 BB，并将时间范围输入添加到简单的 MA 脚本中：

```
//@version=5
indicator("MA", "", true)
tfInput = input.timeframe("D", "Timeframe")
ma = ta.sma(close, 20)
securityNoRepaint(sym, tf, src) =>
    request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])[barstate.isrealtime ? 0 : 1]
maHTF = securityNoRepaint(syminfo.tickerid, tfInput, ma)
plot(maHTF, "MA", color.aqua)
```

注意：

- 我们使用`input.timeframe()` 函数来接收时间范围输入。
- 该函数创建一个下拉小部件，其中建议了一些标准时间范围。时间范围列表还包括您在图表用户界面中喜欢的任何时间范围。
- `tfInput` 我们在`request.security()`调用中使用。我们还在调用中使用，因此该函数仅在较高时间范围完成时返回数据。`gaps = barmerge.gaps_on`



## 符号输入

`input.symbol()` 函数创建一个小部件，允许用户像在图表的用户界面中一样搜索和选择品种。

让我们在脚本中添加一个符号输入：

```

//@version=5
indicator("MA", "", true)
tfInput = input.timeframe("D", "Timeframe")
symbolInput = input.symbol("", "Symbol")
ma = ta.sma(close, 20)
securityNoRepaint(sym, tf, src) =>
    request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])[barstate.isrealtime ? 0
: 1]
maHTF = securityNoRepaint(symbolInput, tfInput, ma)
plot(maHTF, "MA", color.aqua)

```

注意：

- 我们使用的参数 `defval` 是一个空字符串。这会导致 `request.security()`（我们在其中使用 `symbolInput` 包含该输入的变量）默认使用图表的符号。如果用户选择另一个交易品种并希望使用图表交易品种返回到默认值，则他需要使用“输入”选项卡的“默认”菜单中的“重置设置”选项。
- 我们使用 `securityNoRepaint()` 用户定义的函数来使用 `request.security()`，这样它就不会重绘；它仅在较高时间范围完成时返回值。

## 会话输入

会话输入对于收集一段时间内的起止值非常有用。`input.session()` 内置函数创建一个输入小部件，允许用户指定会话的开始和结束时间。可以使用下拉菜单或通过输入“hh:mm”格式的时间值来进行选择。

[input.session\(\)](#) 返回的值是会话格式的有效字符串。有关详细信息，请参阅手册中有关[会话的页面](#)。

会话信息还可以包含有关会话有效日期的信息。我们在这里使用[input.string\(\)](#) 函数调用来输入当天的信息：

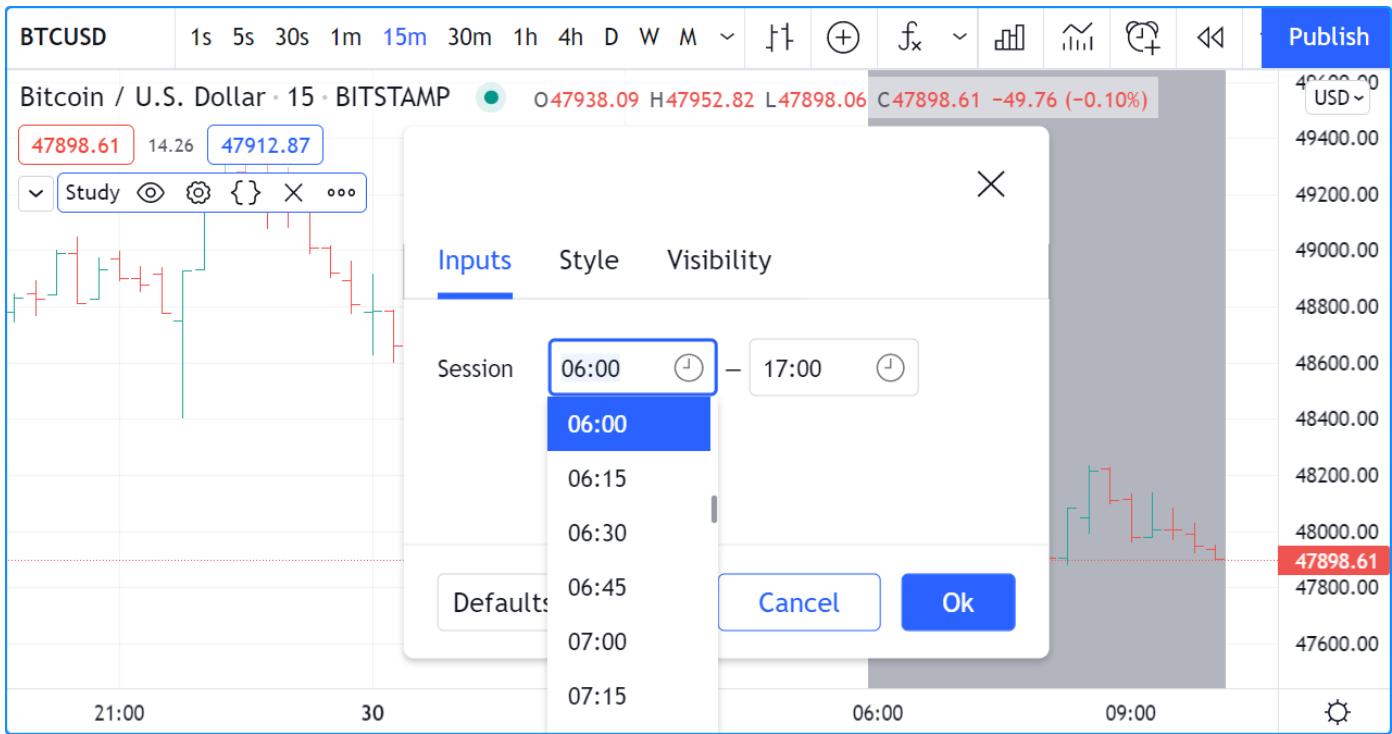
```

//@version=5
indicator("Session input", "", true)
string sessionInput = input.session("0600-1700", "Session")
string daysInput = input.string("1234567", tooltip = "1 = Sunday, 7 = Saturday")
sessionString = sessionInput + ":" + daysInput
inSession = not na(time(timeframe.period, sessionString))
bgcolor(inSession ? color.silver : na)

```

注意：

- 该脚本建议默认会话为“0600-1700”。
- `input.string()` 调用使用工具提示为用户提供有关输入日期信息的格式的帮助。
- 通过连接脚本接收的两个字符串作为输入来构建完整的会话字符串。
- [我们使用string关键字显式声明两个输入的类型](#)，以明确这些变量将包含字符串。
- [我们通过使用会话字符串调用time\(\)](#) 来检测图表栏是否位于用户定义的会话中。如果当前柱的[时间](#)值（柱开盘的时间）不在会话中，则`time()`返回 `na`，因此每当`time()`返回非 `na` `inSession` 的值时都会返回 `na`。`true`



## 源输入

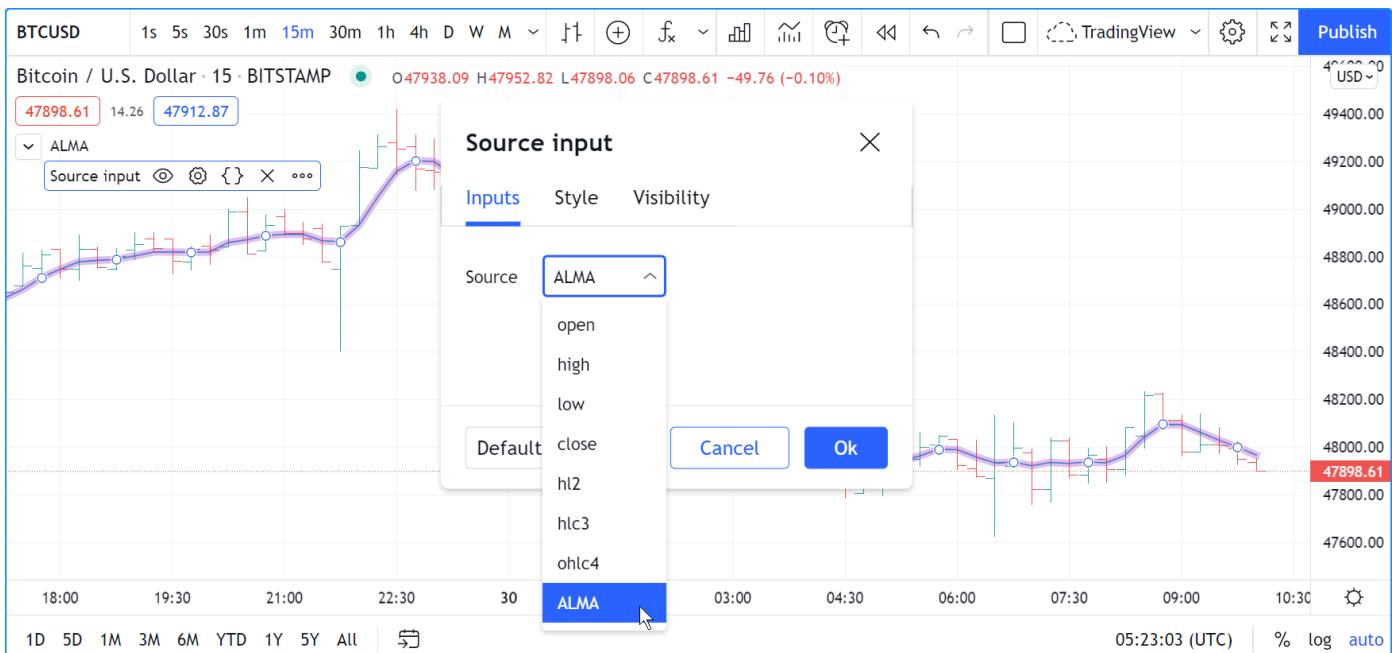
源输入可用于提供两种类型源的选择：

- 价格值，即：[open](#)、[high](#)、[low](#)、[close](#)、[hl2](#)、[hlc3](#)和[ohlc4](#)。
- 由其他脚本在图表上绘制的值。通过将一个脚本的输出作为输入发送到另一个脚本，这对于将两个或多个脚本“链接”在一起非常有用。

该脚本仅绘制用户对源的选择。我们建议将[高](#)值作为默认值：

```
//@version=5
indicator("Source input", "", true)
srcInput = input.source(high, "Source")
plot(srcInput, "Src", color.new(color.purple, 70), 6)
```

这显示了一个图表，除了我们的脚本之外，我们还加载了“Arnaud Legoux 移动平均线”指标。请参阅此处，了解我们如何使用脚本的源输入小部件来选择 ALMA 脚本的输出作为脚本的输入。因为我们的脚本以浅紫色粗线绘制该源，所以您会看到两个脚本的图重叠，因为它们绘制了相同的值：



## 时间输入

时间输入使用`input.time()` 函数。该函数返回以毫秒为单位的 Unix 时间（有关更多信息，请参阅[时间](#)页面）。这种类型的数据还包含日期信息，因此 `input.time()` 函数返回时间和日期。这就是为什么它的小部件允许选择两者的原因。

在这里，我们根据输入值测试柱的时间，并在输入值较大时绘制箭头：

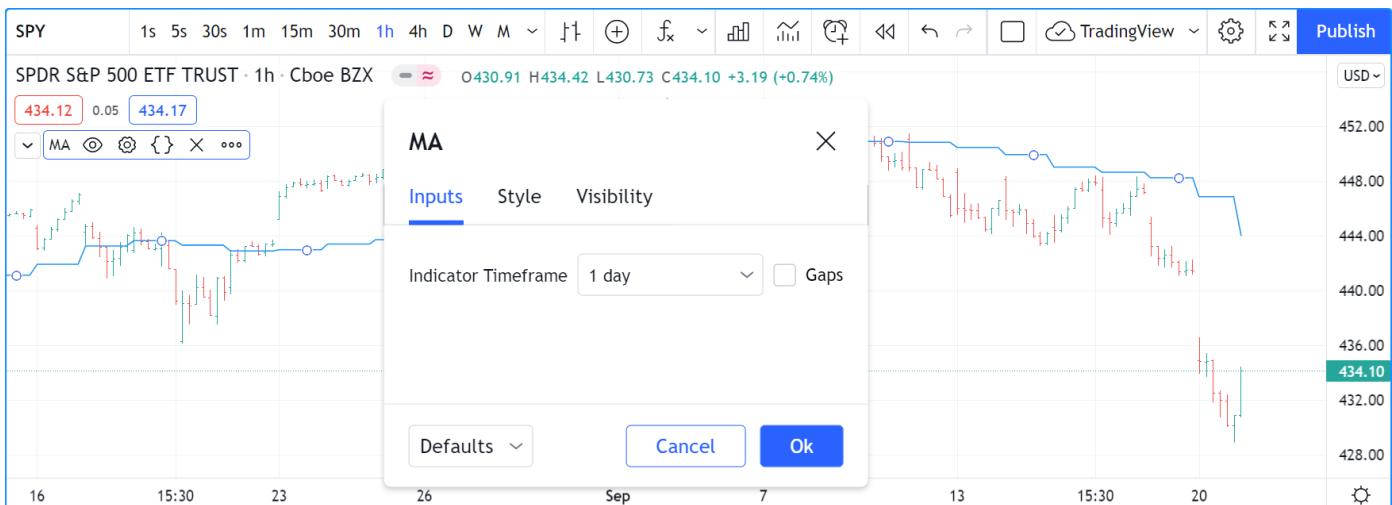
```
//@version=5
indicator("Time input", "T", true)
timeAndDateInput = input.time(timestamp("1 Aug 2021 00:00 +0300"), "Date and time")
barIsLater = time > timeAndDateInput
plotchar(barIsLater, "barIsLater", "↑", location.top, size = size.tiny)
```

请注意，我们使用的值是对`timestamp()` `defval` 函数的调用。

## 影响输入的其他功能

[Indicator\(\)](#)函数的某些参数 在使用时将使用字段填充脚本的“输入”选项卡。参数是 `timeframe` 和 `timeframe_gaps`。一个例子：

```
//@version=5
indicator("MA", "", true, timeframe = "D", timeframe_gaps = false)
plot(ta.vwma(close, 10))
```



## 尖端

脚本输入的设计对脚本的可用性具有重要影响。精心设计的输入可以更直观地使用，并带来更好的用户体验：

- 选择清晰简洁的标签（您输入的 `title` 参数）。
- 仔细选择您的默认值。
- 提供 `minval` 和 `maxval` 值，以防止您的代码产生意外结果，例如，根据您使用的 MA 类型，将长度的最小值限制为 1 或 2。
- 提供 `step` 与您正在捕获的值一致的值。例如，在 0-200 范围内，步长为 5 可能更有用，或者在 0.0-1.0 范围内，步长为 0.05 更有用。
- `inline` 使用；将相关输入分组在同一行上例如，牛市和熊市的颜色，或者线条的宽度和颜色。
- 当您有许多输入时，请使用 将它们分组为有意义的部分 `group`。将最重要的部分放在顶部。
- 对部分内的各个输入执行相同的操作。

`input.*()` 在代码中垂直对齐多个调用的不同参数可能会很有利。当您需要进行全局更改时，这将允许您使用编辑器的多光标功能同时对所有行进行操作。

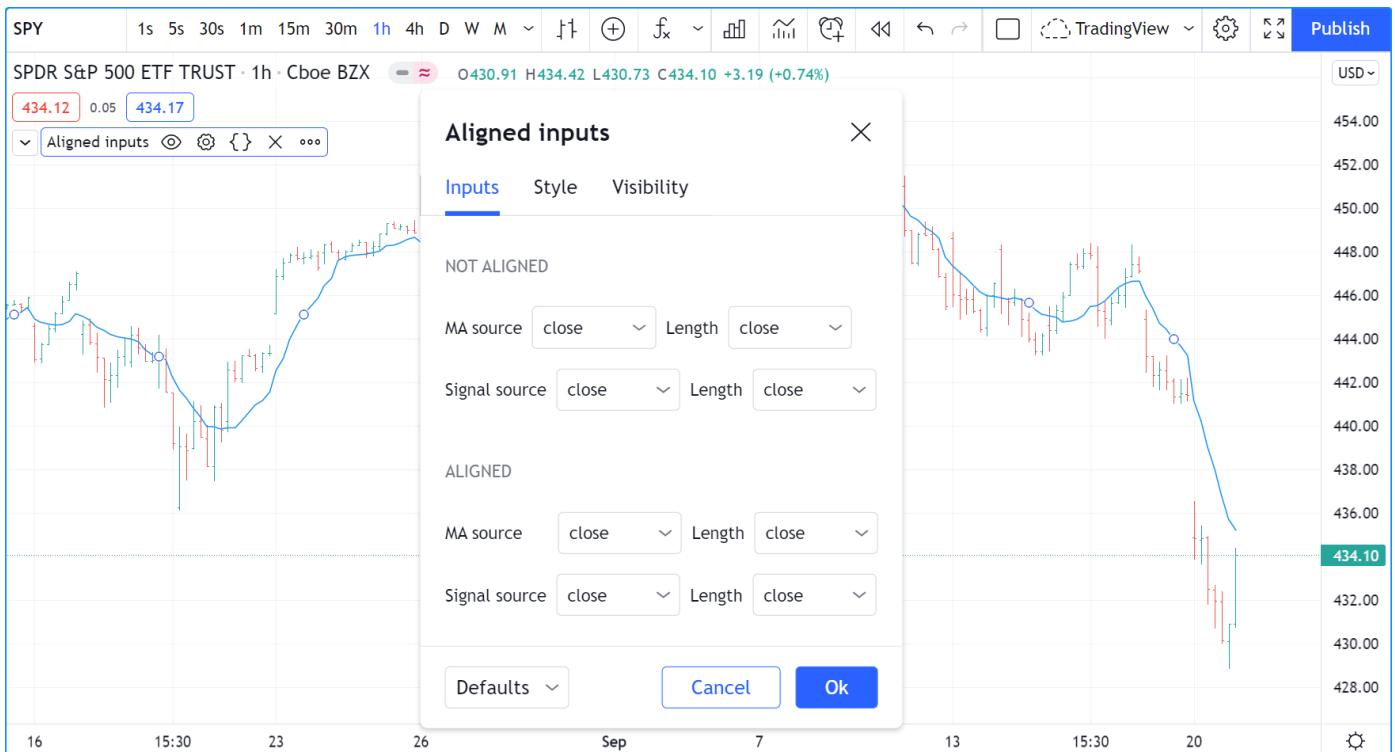
因为有时需要使用 Unicode 空格来实现输入的最佳对齐。这是一个例子：

```
//@version=5
indicator("Aligned inputs", "", true)

var GRP1 = "Not aligned"
ma1SourceInput = input(close, "MA source", inline = "11", group = GRP1)
ma1LengthInput = input(close, "Length", inline = "11", group = GRP1)
long1SourceInput = input(close, "Signal source", inline = "12", group = GRP1)
long1LengthInput = input(close, "Length", inline = "12", group = GRP1)

var GRP2 = "Aligned"
// The three spaces after "MA source" are Unicode EN spaces (U+2002).
ma2SourceInput = input(close, "MA source    ", inline = "21", group = GRP2)
ma2LengthInput = input(close, "Length", inline = "21", group = GRP2)
long2SourceInput = input(close, "Signal source", inline = "22", group = GRP2)
long2LengthInput = input(close, "Length", inline = "22", group = GRP2)
```

```
plot(ta.vwma(close, 10))
```



注意：

- 我们使用 `group` 参数来区分输入的两部分。我们使用常量来保存组的名称。这样，如果我们决定更改组的名称，我们只需要在一个地方更改它。
- 第一部分输入小部件不垂直对齐。我们正在使用 `inline`，它将输入小部件立即放置在标签的右侧。由于 `ma1SourceInput` 和输入的标签 `long1SourceInput` 长度不同，因此标签位于不同的y位置。
- 为了弥补错位，我们 `title` 在行中填充了 `ma2SourceInput` 三个 Unicode EN 空格 (U+2002) 的参数。Unicode 空格是必要的，因为普通空格会从标签中去除。您可以通过组合不同数量和类型的 Unicode 空格来实现精确对齐。有关不同宽度的[Unicode 空格](#)的列表，请参阅[此处链接](#)。

## 级别

### hline() 级别

级别是使用 `hline()` 函数绘制的线。它被设计为使用单一颜色绘制\*\*水平\*\*水平，即它在不同的条上不会改变。当 `hline()` 不能满足您的需要时，请参阅 [plot\(\)](#) 页面的“[级别](#)”部分，了解绘制级别的替代方法。

该函数具有以下签名：

```
hline(price, title, color, linestyle, linewidth, editable) → hline
```

与 [plot\(\)](#) 相比，[hline\(\)](#) 有一些限制：

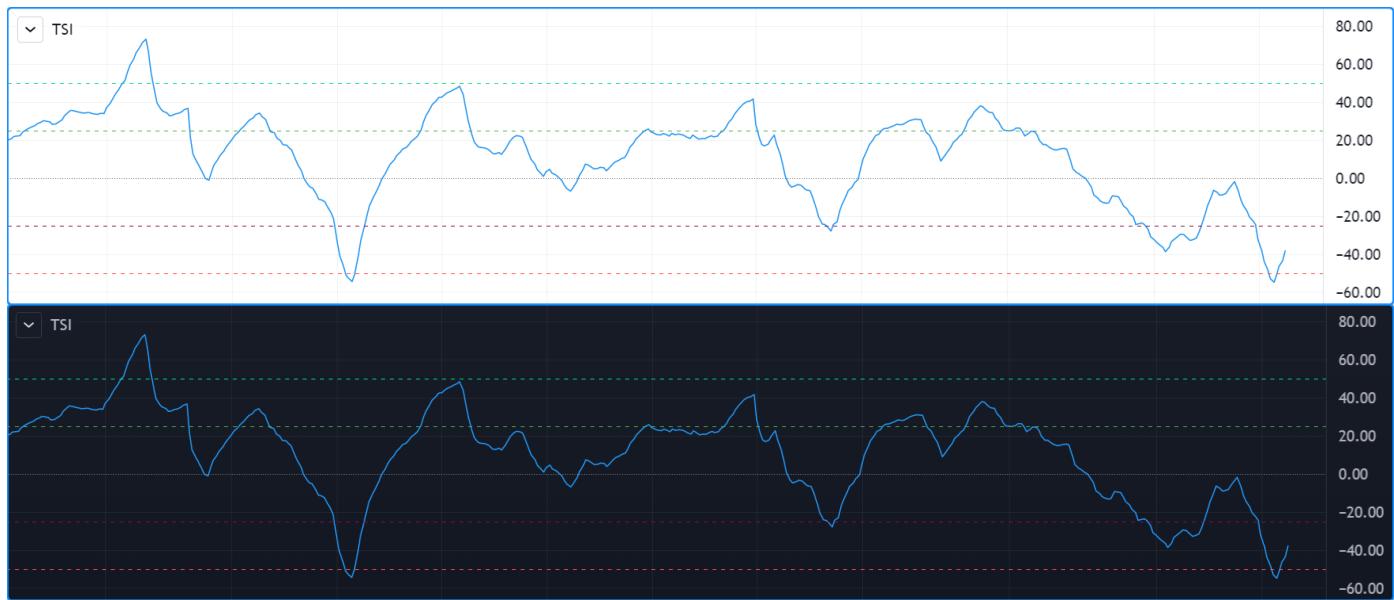
- 由于该函数的目标是绘制水平线，因此其参数需要“输入 int/float”参数，这意味着不能使用 `price` “系列浮点”值，例如接近值或动态计算值。
- 它的 `color` 参数需要一个“input int”参数，这排除了动态颜色的使用，即在每个条上计算的颜色 - 或“系列颜色”值。
- 通过参数支持三种不同的线型 `linestyle`: `hline.style_solid`、`hline.style_dotted` 和 `hline.style_dashed`。

让我们看看[hline\(\)](#) 在“真实强度指数”指标中的作用：

```
//@version=5
indicator("TSI")
myTSI = 100 * ta.tsi(close, 25, 13)

hline( 50, "+50", color.lime)
hline( 25, "+25", color.green)
hline( 0, "zero", color.gray, linestyle = hline.style_dotted)
hline(-25, "-25", color.maroon)
hline(-50, "-50", color.red)

plot(myTSI)
```



注意：

- 我们显示 5 个级别，每个级别都有不同的颜色。
- 我们对零中心线使用不同的线条样式。
- 我们选择适合浅色和深色主题的颜色。
- 指标值的通常范围是 +100 到 -100。由于[ta.tsi\(\)](#) 内置返回 +1 到 -1 范围内的值，因此我们在代码中进行了调整。

## 级别之间的填充

使用[hline\(\)](#)绘制的两个级别之间的空间 可以使用[fill\(\)](#)进行着色。请记住，这两个图必须是用 [hline\(\)](#)绘制的。

让我们在 TSI 指标中添加一些背景颜色：

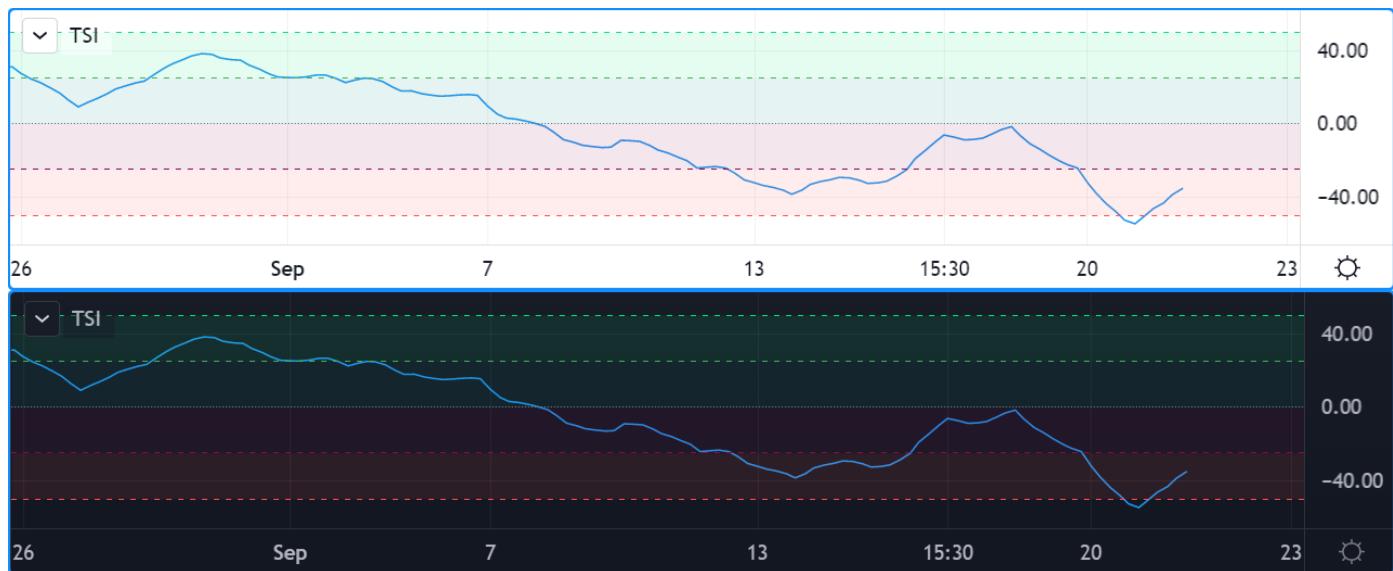
```
//@version=5
indicator("TSI")
myTSI = 100 * ta.tsi(close, 25, 13)

plus50Hline = hline( 50, "+50", color.lime)
plus25Hline = hline( 25, "+25", color.green)
zeroHline   = hline( 0, "Zero", color.gray, linestyle = hline.style_dotted)
minus25Hline = hline(-25, "-25", color.maroon)
minus50Hline = hline(-50, "-50", color.red)

// —— Function returns a color in a light shade for use as a background.
fillColor(color col) =>
    color.new(col, 90)

fill(plus50Hline, plus25Hline, fillColor(color.lime))
fill(plus25Hline, zeroHline,   fillColor(color.teal))
fill(zeroHline,   minus25Hline, fillColor(color.maroon))
fill(minus25Hline, minus50Hline, fillColor(color.red))

plot(myTSI)
```



注意：

- 现在我们已经使用了 `hline()` 函数调用的返回值，它是 `hline` 特殊类型。我们使用 `plus50Hline`、`plus25Hline`、`zeroHline` 和 `minus25Hline` 变量来存储这些“`hline`”ID，因为稍后我们将在 `fill()` 调用中需要它们。`minus50Hline`
- 为了生成背景颜色较浅的色调，我们声明一个 `fillColor()` 接受颜色并返回其 90 透明度的函数。我们 `color` 在 `fill()` 调用中使用对该函数的调用作为参数。
- 我们在四对不同的级别之间对我们想要的四种不同填充中的每一种进行 `fill()` 调用。
- 我们在第二次填充中使用它 `color.teal`，因为它产生的绿色比 `color.green` 25 级别使用的绿色更适合配色方案。

# 库函数

## 介绍

Pine Script™ 库是包含可在指标、策略或其他库中重用的函数的出版物。它们对于定义常用函数很有用，因此不必将其源代码包含在需要它们的每个脚本中。

库必须先发布（私下或公开），然后才能在另一个脚本中使用。所有库都是开源发布的。公共脚本只能使用公共库，并且它们必须是开源的。保存在 Pine Script™ 编辑器中的私有脚本或个人脚本可以使用公共或私有库。一个库可以使用其他库，甚至可以使用其自身的早期版本。

库程序员应该熟悉 Pine Script™ 的类型命名法、范围和用户定义的函数。如果您需要温习合格的类型，请参阅[类型系统](#)上的用户手册页面。有关用户定义函数和作用域的更多信息，请参阅[用户定义函数](#)页面。

您可以在 TradingView[社区脚本](#)中浏览成员公开发布的库脚本。

## 创建库

库是一种特殊类型的脚本，以[library\(\)](#)声明语句开头，而不是[indicator\(\)](#)或[strategy\(\)](#)。库包含可导出的函数定义，当另一个脚本使用库时，它们构成库的唯一可见部分。库还可以在其全局范围内使用其他 Pine Script™ 代码，就像普通指示器一样。该代码通常用于演示如何使用库的函数。

库脚本具有以下结构，其中必须定义一个或多个可导出函数：

```
//@version=5

// @description <library_description>
library(title, overlay)

<script_code>

// @function <function_description>
// @param <parameter> <parameter_description>
// @returns <return_value_description>
export <function_name>([simple/series] <parameter_type> <parameter_name> [=
<default_value>] [, ...]) =>
    <function_code>

<script_code>
```

注意：

- 、和[编译器注释](#)是可选的，但我们强烈建议您使用它们。它们有双重目的：记录库的代码并填充作者在发布库时可以使用的默认库描述。`// @description``// @function``// @param``// @returns`
- 导出关键字是强制性的。
- <parameter\_type> 是强制性的，这与指标或策略中用户定义的函数参数定义相反，后者是无类型的。

- <script\_code> 可以是您通常在指标中使用的任何代码，包括输入或绘图。

这是一个示例库：

```
//@version=5

// @description Provides functions calculating the all-time high/low of values.
library("AllTimeHighLow", true)

// @function Calculates the all-time high of a series.
// @param val Series to use (`high` is used if no argument is supplied).
// @returns The all-time high for the series.
export hi(float val = high) =>
    var float ath = val
    ath := math.max(ath, val)

// @function Calculates the all-time low of a series.
// @param val Series to use (`low` is used if no argument is supplied).
// @returns The all-time low for the series.
export lo(float val = low) =>
    var float atl = val
    atl := math.min(atl, val)

plot(hi())
plot(lo())
```

## 库函数

库中的函数定义与指标和策略中的用户定义函数略有不同。库函数体内可以包含的内容是有限制的。

在库函数签名中（第一行）：

- 导出关键字是强制性的。
- 必须明确提及每个参数所需的参数类型。
- simple或 [series关键字可以限制允许的参数限定](#) 类型（下一节将解释它们的用途）。

这些是对库函数施加的约束：

- 它们不能使用库全局范围内的变量，除非它们被限定为“const”。这意味着您不能使用例如从脚本输入初始化的全局变量或全局声明的数组。
- `request.*()` 不允许打电话。
- `input.*()` 不允许打电话。
- `plot*()`, `fill()` 并且 `bgcolor()` 不允许打电话。

库函数总是返回被限定为“简单”或“系列”的结果。您不能在需要“const”或“input”限定值的地方使用它们，就像某些内置函数的情况一样。例如，库函数不能用于计算 `plot().show_last` 调用中参数的参数，因为它需要“输入int”值。

## 合格的类型控制

调用库函数时提供的参数的限定类型是根据每个参数在函数内部的使用方式自动检测的。如果论证可以用作“系列”，那么它就被限定为“系列”。如果不能，则尝试使用“simple”类型限定符。这解释了为什么这段代码：

```
export myEma(int x) =>
    ta.ema(close, x)
```

`myCustomLibrary.myEma(20)` 即使`ta.ema()`的`length`参数需要一个“简单 int”参数，使用`using`调用时也会起作用。当 Pine Script™ 编译器检测到“系列”长度不能与`ta.ema()`一起使用时，它会尝试“简单”限定符，在本例中这是允许的。

虽然库函数不能返回“常量”或“输入”值，但可以编写它们来产生“简单”结果。这使得它们比返回“系列”结果的函数在更多上下文中有用，因为某些内置函数不允许“系列”参数。例如，`request.security()`需要一个“简单字符串”作为其`symbol`参数。如果我们编写一个库函数来按以下方式组装参数`symbol`，则该函数的结果将不起作用，因为它是“系列字符串”限定类型：

```
export makeTickerid(string prefix, string ticker) =>
    prefix + ":" + ticker
```

然而，通过将参数限定符限制为“simple”，我们可以强制函数产生“simple”结果。我们可以通过在参数类型前加上`simple`关键字来实现这一点：

```
export makeTickerid(simple string prefix, simple string ticker) =>
    prefix + ":" + ticker
```

请注意，对于返回“简单”值的函数，在其计算中不能使用“系列”值；否则结果将是一个“系列”值。

还可以使用`series`关键字作为库函数参数类型的前缀。但是，由于默认情况下参数被限定为“系列”，因此使用`系列`修饰符是多余的。

## 用户定义的类型和对象

您可以从库中导出[用户定义类型 \(UDT\)](#)，并且库函数可以返回[对象](#)。

要导出 UDT，请在其定义前加上[导出](#)关键字，就像导出函数一样：

```
//@version=5
library("Point")

export type point
    int x
    float y
    bool isHi
    bool wasBreached = false
```

导入该库并从其 `point` UDT 创建对象的脚本看起来有点像这样：

```
//@version=5
indicator("")
import userName/Point/1 as pt
newPoint = pt.point.new()
```

注意：

- 此代码无法编译，因为没有发布“Point”库，并且脚本不显示任何内容。
- `userName` 需要替换为库发布者的 TradingView 用户名。
- 我们使用内置 `new()` 方法从 UDT 创建对象 `point`。
- [我们使用`import`语句中定义的别名](#) 作为对库 `point` UDT 的引用的前缀，就像使用导入库中的函数一样。`pt`

如果库中使用的 UDT 的任何导出函数使用参数或返回该用户定义类型的结果，则必须导出该 UDT。

当库仅在内部使用 UDT 时，无需将其导出。以下库 `point` 内部使用了 UDT，但仅 `drawPivots()` 导出其函数，不使用参数也不返回 `point` 类型的结果：

```
//@version=5
library("PivotLabels", true)

// We use this `point` UDT in the library, but it does NOT require exporting because:
// 1. The exported function's parameters do not use the UDT.
// 2. The exported function does not return a UDT result.

type point
    int x
    float y
    bool isHi
    bool wasBreached = false

fillPivotsArray(qtyLabels, leftLegs, rightLegs) =>
    // Create an array of the specified qty of pivots to maintain.
    var pivotsArray = array.new<point>(math.max(qtyLabels, 0))

    // Detect pivots.
    float pivotHi = ta.pivothigh(leftLegs, rightLegs)
    float pivotLo = ta.pivotlow(leftLegs, rightLegs)

    // Create a new `point` object when a pivot is found.
    point foundPoint = switch
        pivotHi => point.new(time[rightLegs], pivotHi, true)
        pivotLo => point.new(time[rightLegs], pivotLo, false)
        => na

    // Add new pivot info to the array and remove the oldest pivot.
    if not na(foundPoint)
```

```

array.push(pivotsArray, foundPoint)
array.shift(pivotsArray)

array<point> result = pivotsArray


detectBreaches(pivotsArray) =>
  // Detect breaches.
  for [i, eachPoint] in pivotsArray
    if not na(eachPoint)
      if not eachPoint.wasBreached
        bool hiWasBreached = eachPoint.isHi and high[1] <= eachPoint.y and
high > eachPoint.y
        bool loWasBreached = not eachPoint.isHi and low[1] >= eachPoint.y and
low < eachPoint.y
        if hiWasBreached or loWasBreached
          // This pivot was breached; change its `wasBreached` field.
          point p = array.get(pivotsArray, i)
          p.wasBreached := true
          array.set(pivotsArray, i, p)


drawLabels(pivotsArray) =>
  for eachPoint in pivotsArray
    if not na(eachPoint)
      label.new(
        eachPoint.x,
        eachPoint.y,
        str.tostring(eachPoint.y, format.mintick),
        xloc.bar_time,
        color = eachPoint.wasBreached ? color.gray : eachPoint.isHi ? color.teal
: color.red,
        style = eachPoint.isHi ? label.style_label_down: label.style_label_up,
        textcolor = eachPoint.wasBreached ? color.silver : color.white)

// @function      Displays a label for each of the last `qtyLabels` pivots.
//                 Colors high pivots in green, low pivots in red, and breached pivots
in gray.
// @param qtyLabels (simple int) Quantity of last labels to display.
// @param leftLegs  (simple int) Left pivot legs.
// @param rightLegs (simple int) Right pivot legs.
// @returns        Nothing.
export drawPivots(int qtyLabels, int leftLegs, int rightLegs) =>
  // Gather pivots as they occur.
  pointsArray = fillPivotsArray(qtyLabels, leftLegs, rightLegs)

  // Mark breached pivots.
  detectBreaches(pointsArray)

```

```

// Draw labels once.
if barstate.islastconfirmedhistory
    drawLabels(pointsArray)

// Example use of the function.
drawPivots(20, 10, 5)

```

如果 TradingView 用户发布了上述库，则可以这样使用：

```

//@version=5
indicator("")
import TradingView/PivotLabels/1 as dpl
dpl.drawPivots(20, 10, 10)

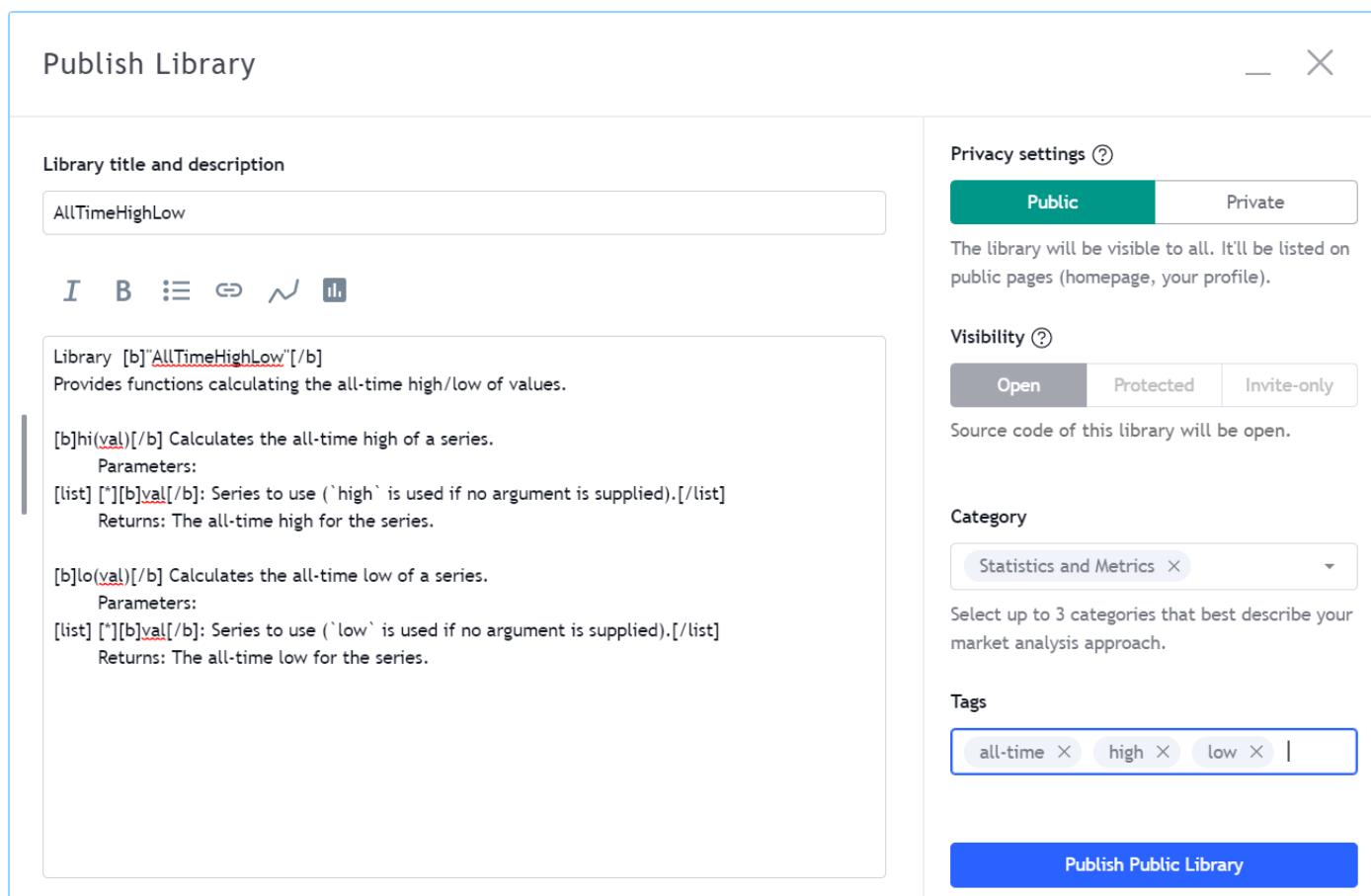
```

## 发布库

在您或其他 Pine Script™ 程序员可以重用任何库之前，必须先发布该库。如果您想与所有 TradingViewers 共享您的库，请公开发布。要私人使用它，请使用私人出版物。与指标或策略一样，当您发布库时，活动图表将同时出现在其小部件（表示 TradingView 脚本流中的库的小占位符）和脚本页面（用户单击小部件时看到的页面）中。

私有库可以在公共受保护或仅限邀请的脚本中使用。

将我们的示例库添加到图表并设置一个干净的图表以我们想要的方式显示我们的库绘图后，我们使用 Pine 编辑器的“发布脚本”按钮。出现“发布库”窗口：



注意：

- 我们保留库的标题不变（`library()` `title` 声明语句中的参数用作默认值）。虽然您可以更改出版物的标题，但最好保留其默认值，因为该参数用于引用 [导入](#)语句中的导入库。当您的出版物标题与图书馆的实际名称相匹配时，图书馆用户的生活会更轻松。`title`
- 默认描述是根据我们在库中使用的[编译器注释](#)构建的。我们将发布该库而不对其进行修改。
- 我们选择公开发布我们的库，因此所有 TradingViewers 都可以看到它。
- 我们无法选择“开放”以外的可见性类型，因为库始终是开源的。
- 图书馆的类别列表与指标和策略的类别列表不同。我们选择了“统计和指标”类别。
- 我们添加了一些自定义标签：“历史上”、“最高”和“最低”。

公共库的目标用户是其他 Pine 程序员；您对库的功能解释和记录得越好，其他人使用它们的机会就越大。提供示例来演示如何在出版物的代码中使用库的函数也会有所帮助。

## 家庭规则

在我们的[《脚本发布内部规则》](#)中，Pine 库被视为“公共领域”代码，这意味着如果您在开源脚本中调用它们的函数或重用它们的代码，则不需要获得其作者的许可。但是，如果您打算在公共受保护或仅限邀请的出版物中重用 Pine Script™ 库函数中的代码，则需要获得其作者的明确许可才能以该形式重用。

无论是使用库的函数还是重用其代码，您都必须在出版物的描述中注明作者。这也是在开源评论中值得赞扬的好形式。

## 使用库

使用另一个脚本中的库（可以是指标、策略或另一个库）是通过[import](#)语句完成的：

```
import <username>/<libraryName>/<libraryVersion> [as <alias>]
```

在哪里：

- <用户名>/<库名称>/<库版本> 路径将唯一标识该库。
- 必须显式指定。为了确保使用库的脚本的可靠性，无法自动使用最新版本的库。每次作者发布库更新时，库的版本号都会增加。如果您打算使用最新版本的库，则需要在[导入](#)语句中更新 值。
- 该部分是可选的。使用时，它定义将引用库函数的名称空间。例如，如果您像下面的示例中那样使用别名导入库，您将将该库的函数称为`.as`。当未定义别名时，库的名称将成为其命名空间。`as`  
`<alias>`allTime`<function_name>()`

要使用我们在上一节中发布的库，我们的下一个脚本将需要一个[import](#)语句：

```
import PineCoders/AllTimeHighLow/1 as allTime
```

当您键入库作者的用户名时，您可以使用编辑器的ctrl+ space/ cmd+ space“自动完成”命令显示一个弹出窗口，提供与可用库匹配的选择：

```
1 //@version=5
2 indicator("Using AllTimeHighLow library")
3 import PineCoders
4 import PineCoders/AllTimeHighLow/1 Provides functions calculating the all-time
5
```

这是一个重用我们库的指标：

```
//@version=5
indicator("Using AllTimeHighLow library", "", true)
import PineCoders/AllTimeHighLow/1 as allTime

plot(allTime.hi())
plot(allTime.lo())
plot(allTime.hi(close))
```

注意：

- 我们选择在脚本中使用库实例的“allTime”别名。在编辑器中输入该别名时，将出现一个弹出窗口，帮助您从库中选择要使用的特定函数。
- 我们使用不带参数的库 `hi()` 和 `lo()` 函数，因此默认的 `high` 和 `low` 内置变量将分别用于它们的系列。
- 我们第二次调用 `allTime.hi()`，但这次使用 `close` 作为其参数，以绘制图表历史中的最高收盘价。

## 线条和方框

### 介绍

Pine Script™ 有助于使用 `line`、`box` 和 `polyline` 类型从 [代码](#) 中绘制线、框和其他几何形状。这些类型提供了以编程方式在图表上绘制支撑位和阻力位、趋势线、价格范围和其他自定义形态的实用程序。

与 [绘图](#) 不同，这些类型的灵活性使它们特别适合在图表上几乎任何可用点可视化当前计算的数据，而不管脚本在哪个图表条上执行。

[线](#)、[框](#) 和 [折线](#) 都是对象，就像 [标签](#)、[表格](#) 和其他特殊类型一样。脚本使用 ID 引用这些类型的对象，其作用类似于指针。与其他对象一样，[line](#)、[box](#) 和 [polyline](#) ID 被限定为“系列”值，并且管理这些对象的所有函数都接受“系列”参数。

笔记：

使用我们在本页讨论的 [类型](#) 通常涉及 [数组](#)，特别是在使用 [折线](#) 时，它需要一个 [图表点实例数组](#)。因此，我们建议您熟悉 [数组](#)，以便在脚本中充分利用这些绘图类型。

脚本绘制的线条可以是垂直的、水平的或有角度的。盒子总是长方形的。折线顺序连接多个垂直、水平、有角度或曲线的线段。尽管所有这些绘图类型都有不同的特征，但它们确实有一些共同点：

- [Lines](#)、[Box](#)和[Polyline](#)可以在图表上的任何可用位置具有坐标，包括最后一个图表栏之外的未来时间的坐标。
- 这些类型的对象可以使用[Chart.point](#)实例来设置它们的坐标。
- 每个对象的x坐标可以是条形索引或时间值，具体取决于其指定的xloc属性。
- 每个对象可以具有多种预定义线条样式之一。
- 脚本可以在循环和条件结构的范围内调用管理这些对象的函数，从而允许对其绘图进行迭代和条件控制。
- 脚本可以引用并在图表上显示的这些对象的数量是有限制的。单个脚本实例最多可以显示500条线、500个框和100条折线。用户可以通过脚本的[indicator\(\)](#)或[strategy\(\)](#)声明语句的max\_lines\_count、max\_boxes\_count、和参数指定每种类型允许的最大数量。如果未指定，默认值为~50。与[标签](#)和[表格](#)类型一样，线条、方框和折线利用垃圾收集机制，当绘图总数超过脚本限制时，该机制会删除图表上最旧的对象。max\_polylines\_count

## 笔记

在TradingView图表上，一整套绘图工具允许用户使用鼠标操作创建和修改绘图。虽然它们有时可能类似于使用Pine Script™代码创建的绘图对象，但它们是不相关的实体。Pine脚本无法与图表用户界面中的绘图工具交互，并且鼠标操作不会直接影响Pine绘图对象。

## 线路

命名空间中的内置函数控制[线](#)`line.*`对象的创建和管理：

- `line.new()`函数创建一个新行。
- 这些`line.set_*`()函数修改线属性。
- 这些`line.get_*`()函数从线路实例中检索值。
- `line.copy()`函数克隆一个线实例。
- `line.delete()`函数删除现有的线路实例。
- `line.all`变量引用一个只读数组，其中包含脚本显示的所有行的ID。数组的大小取决于[indicator\(\)](#)或[strategy\(\)](#)声明语句的大小以及脚本绘制的行数。`max_lines_count`

脚本可以将`line.set_*`()、`line.get_*`()、`line.copy()`和`line.delete()`内置函数作为函数或方法调用。

## 创建线条

`line.new()`函数创建一个新的[线条](#)实例以显示在图表上。它有以下签名：

```
line.new(first_point, second_point, xloc, extend, color, style, width) → series line

line.new(x1, y1, x2, y2, xloc, extend, color, style, width) → series line
```

该函数的第一个重载包含 `first_point` 和 `second_point` 参数。是代表线条起点的图表点，是 `first_point` 代表线条 `second_point` 终点的图表点。该函数从这些图表点复制信息以确定线条的坐标。是否使用 `和` 中的 `index` 或字段作为 x 坐标取决于函数的值。`time``first_point``second_point``xloc`

第二个重载独立指定 `x1`、`y1`、`x2` 和 `y2` 值，其中 `x1` 和 `x2` 是表示线的起始和结束 x 坐标的 `int` `y1` 值，和 `y2` 是表示 y 坐标的 `浮点` 值。该行是否将这些 `x` 值视为柱索引或时间戳取决于 `xloc` 函数调用中的值。

两个重载共享相同的附加参数：

- `xloc`

控制新线的 x 坐标是否使用柱索引或时间值。它的默认值为 `xloc.bar_index`。当调用第一个重载时，使用 `xloc.bar_index` `xloc` 的值告诉函数使用 `和` 的字段，并且 `xloc.bar_time` 的值告诉函数使用点的字段。`index``first_point``second_point``time` 调用第二个重载时，`xloc.bar_index` `xloc` 的值会提示函数将 `和` 和参数视为条形索引值。当使用 `xloc.bar_time` 时，该函数会将 `和` 视为时间值。`x1``x2``x1``x2` 当指定的 x 坐标表示柱索引值时，请务必注意允许的最小 x 坐标为 。对于较大的偏移量，可以使用 `xloc.bar_time`。`bar_index - 9999`

- `extend`

确定绘制的线是否无限延伸超出其定义的起点和终点坐标。它接受以下值之一：`extend.left`、`extend.right`、`extend.both` 或 `extend.none`（默认）。

- `color`

指定线条图的颜色。默认为 `color.blue`。

- `style`

指定线条的样式，可以是本页的 [线条样式](#) 部分中列出的任何选项。默认值为 `line.style_solid`。

- `width`

控制线的宽度（以像素为单位）。默认值为 1。

下面的示例演示了如何以最简单的形式绘制线条。此脚本在每个图表条的水平中心绘制一条连接 [开盘价](#) 和 [收盘价](#) 的新垂直线：



```
//@version=5
indicator("Creating lines demo", overlay = true)
```

```

//@variable The `chart.point` for the start of the line. Contains `index` and `time` information.
firstPoint = chart.point.now(open)
//@variable The `chart.point` for the end of the line. Contains `index` and `time` information.
secondPoint = chart.point.now(close)

// Draw a basic line with a `width` of 5 connecting the `firstPoint` to the `secondPoint`.
// This line uses the `index` field from each point for its x-coordinates.
line.new(firstPoint, secondPoint, width = 5)

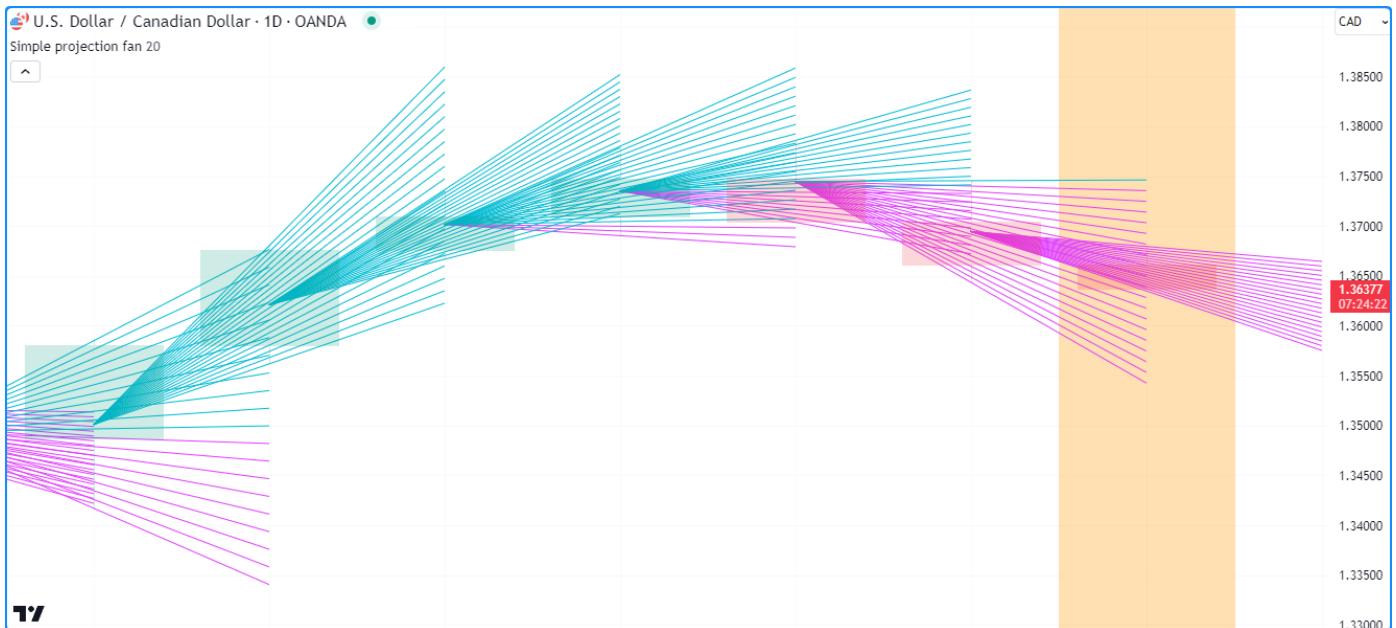
// Color the background on the unconfirmed bar.
bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

```

- 注意：

如果 `firstPoint` 和 `secondPoint` 引用相同的坐标，脚本将不会显示一条线，因为它们之间没有要绘制的距离。但是，线路 ID 仍然存在。该脚本将仅显示图表上大约最后 50 条线，因为它没有 `max_lines_count` 在 [Indicator\(\)](#) 函数调用中指定。线条图一直保留在图表上，直到使用 [line.delete\(\)](#) 删除 或被垃圾收集器删除。该脚本会在打开的图表栏（即带有橙色背景突出显示的栏）上重新绘制线条，直到其关闭。栏关闭后，将不再更新绘图。

让我们看一个更复杂的例子。该脚本使用前一个柱的 [hl2](#) 价格和当前柱的 [最高价](#) 和 [最低价](#) 来绘制一个扇形，并用用户指定的线条数来投影下一个图表柱的假设价格值范围。它在 [for](#) 循环中调用 [line.new\(\)](#) 在每个柱上创建线条：



```

//@version=5
indicator("Creating lines demo", "Simple projection fan", true, max_lines_count = 500)

//@variable The number of fan lines drawn on each chart bar.

```

```

int linesPerBar = input.int(20, "Line drawings per bar", 2, 100)

//@variable The distance between each y point on the current bar.
float step = (high - low) / (linesPerBar - 1)

//@variable The `chart.point` for the start of each line. Does not contain `time` information.
firstPoint = chart.point.from_index(bar_index - 1, hl2[1])
//@variable The `chart.point` for the end of each line. Does not contain `time` information.
secondPoint = chart.point.from_index(bar_index + 1, float(na))

//@variable The stepped y value on the current bar for `secondPoint.price` calculation, starting from the `low`.
float barValue = low
// Loop to draw the fan.
for i = 1 to linesPerBar
    // Update the `price` of the `secondPoint` using the difference between the `barValue` and `firstPoint.price`.
    secondPoint.price := 2.0 * barValue - firstPoint.price
    //@variable Is `color.aqua` when the line's slope is positive, `color.fuchsia` otherwise.
    color lineColor = secondPoint.price > firstPoint.price ? color.aqua : color.fuchsia
    // Draw a new `lineColor` line connecting the `firstPoint` and `secondPoint` coordinates.
    // This line uses the `index` field from each point for its x-coordinates.
    line.new(firstPoint, secondPoint, color = lineColor)
    // Add the `step` to the `barValue`.
    barValue += step

// Color the background on the unconfirmed bar.
bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed bar highlight")

```

- 注意：

我们已包含在[Indicator\(\)](#)函数调用中，这意味着脚本在图表上最多保留 500 条线。`max_lines_count = 500` 每个[line.new\(\)](#)调用都会从和变量引用的[Chart.point](#)复制信息。因此，脚本可以在每次循环迭代时更改的字段，而不会影响其他行中的 y 坐标。`firstPoint``secondPoint``price``secondPoint`

## 修改行

命名 `line.*` 空间包含多个修改线实例属性的setter函数：

- [line.set\\_first\\_point\(\)](#) 和 [line.set\\_second\\_point\(\)](#) `id` 分别使用指定的信息更新线的起点和终点 `point`。
- [line.set\\_x1\(\)](#) 和 [line.set\\_x2\(\)](#) 将线的 x 坐标之一设置 `id` 为新 `x` 值，该值可以表示柱索引或时间值，具体取决于线的 `xloc` 属性。

- `line.set_y1()`和`line.set_y2()`将线的y坐标之一设置`id`为新`y`值。
- `line.set_xy1()`和`line.set_xy2()``id`用新的`x`和值更新线的点之一`y`。
- `line.set_xloc()`设置线`xloc`的并用新值和值`id`更新其`x`坐标。`x1``x2`
- `line.set_extend()`设置线`extend`的属性`id`。
- `line.set_color()`更新`id`线条的`color`值。
- `line.set_style()`更改线条`style`的样式`id`。
- `line.set_width()`设置线`width`的宽度`id`。

所有setter函数都直接修改`id`传递到调用中的行，并且不返回任何值。每个setter函数都接受“系列”参数，因为脚本可以在整个执行过程中更改行的属性。

以下示例绘制了将`a`的开盘价`timeframe`与其收盘价连接起来的线。该脚本使用`var`关键字声明和仅在第一个图表栏上引用`Chart.point`值(和)`periodLine`的变量，并在执行过程中为这些变量分配新值。检测到上的`更改`后，它使用`line.set_color()`设置现有的，使用`Chart.point.now()`为和创建新值，然后使用这些点将`newLine`分配给。`openPoint``closePoint``timeframe``color``periodLine``openPoint``closePoint``periodLine`

`periodLine`在值不是`na`的其他柱上，脚本将新的`Chart.point`分配给`closePoint`，然后使用`line.set_second_point()`和`line.set_color()`作为更新线条属性的方法：



```
//@version=5
indicator("Modifying lines demo", overlay = true)

//@variable The size of each period.
string timeframe = input.timeframe("D", "Timeframe")

//@variable A line connecting the period's opening and closing prices.
var line periodLine = na

//@variable The first point of the line. Contains `time` and `index` information.
var chart.point openPoint = chart.point.now(open)
//@variable The closing point of the line. Contains `time` and `index` information.
```

```

var chart.point closePoint = chart.point.now(close)

if timeframe.change(timeframe)
    // @variable The final color of the `periodLine`.
    color finalColor = switch
        closePoint.price > openPoint.price => color.green
        closePoint.price < openPoint.price => color.red
        => color.gray

    // Update the color of the current `periodLine` to the `finalColor`.
    line.set_color(periodLine, finalColor)

    // Assign new points to the `openPoint` and `closePoint`.
    openPoint := chart.point.now(open)
    closePoint := chart.point.now(close)
    // Assign a new line to the `periodLine`. Uses `time` fields from the `openPoint`
    and `closePoint` as x-coordinates.
    periodLine := line.new(openPoint, closePoint, xloc.bar_time, style =
line.style_arrow_right, width = 3)

else if not na(periodLine)
    // Assign a new point to the `closePoint`.
    closePoint := chart.point.now(close)

    // @variable The color of the developing `periodLine`.
    color developingColor = switch
        closePoint.price > openPoint.price => color.aqua
        closePoint.price < openPoint.price => color.fuchsia
        => color.gray

    // Update the coordinates of the line's second point using the new `closePoint`.
    // It uses the `time` field from the point for its new x-coordinate.
    periodLine.set_second_point(closePoint)
    // Update the color of the line using the `developingColor`.
    periodLine.set_color(developingColor)

```

- 注意：

此示例中的每个线条图都使用 [line.style\\_arrow\\_right](#) 样式。有关所有可用样式设置的概述，请参阅下面的[线条样式部分](#)。

## 线条样式

`style` 用户可以通过在[line.new\(\)](#)或[line.set\\_style\(\)](#)函数调用中传递以下变量之一作为参数来控制脚本线条图的样式：

争论	线	争论	线
<code>line.style_solid</code>		<code>line.style_arrow_left</code>	
<code>line.style_dotted</code>		<code>line.style_arrow_right</code>	
<code>line.style_dashed</code>		<code>line.style_arrow_both</code>	

- 注意：

多段线还可以使用这些变量中的任何一个作为其 `line_style` 值。请参阅本页的[创建折线部分](#)。

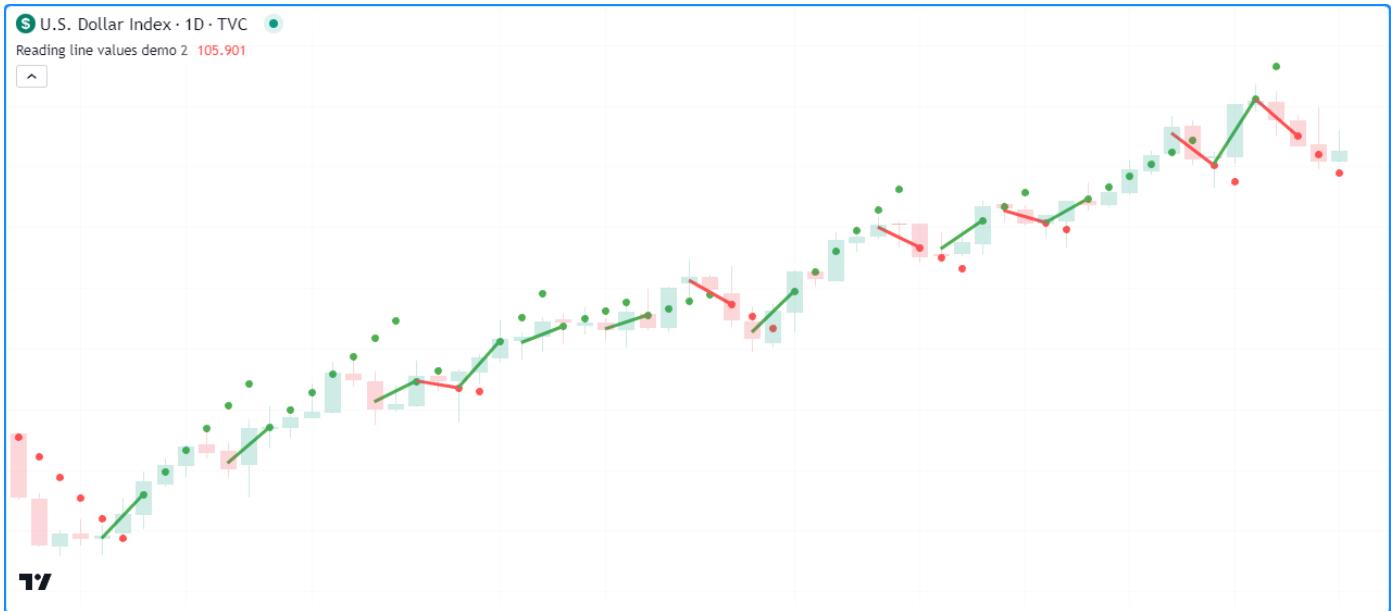
## 读取行值

命名空间 `line.*` 包括 `getter` 函数，它允许脚本从 `行` 对象中检索值以供进一步使用：

- `line.get_x1()` 和 `line.get_x2()` 分别获取该行的第一个和第二个 `x` 坐标 `id`。返回的值是否表示柱索引或时间值取决于线的 `xloc` 属性。
- `line.get_y1()` 和 `line.get_y2()` 分别获取该 `id` 行的第一个和第二个 `y` 坐标。
- `line.get_price()` `id` 检索指定值处的线的价格 (`y` 坐标) `x`，包括线起点和终点之外的柱索引处的价格。此函数仅与使用 `xloc.bar_index` 作为 `xloc` 值的行兼容。

下面的脚本在柱形上形成[上涨](#) 或[下跌](#) 价格模式时绘制一条新线 `length`。它使用 `var directionLine` 关键字在第一个图表栏上声明变量。分配给的 ID `directionLine` 在后续柱中持续存在，直到 `newDirection` 条件发生，在这种情况下，脚本会为变量分配新行。

在每个柱上，脚本调用 `line.get_y2()`、`line.get_y1()`、`line.get_x2()` 和 `line.get_x1()` `getter` 作为从当前值检索值并计算其的方法，用于确定每幅图画和情节的颜色。它使用 `line.get_price()` 从第二个点之外检索的扩展值，并将它们绘制在图表上： `directionLine``slope``directionLine`



```

//@version=5
indicator("Reading line values demo", overlay = true)

//@variable The number of bars for rising and falling calculations.
int length = input.int(2, "Length", 2)

//@variable A line that's drawn whenever `hlc3` starts rising or falling over `length` bars.
var line directionLine = na

//@variable Is `true` when `hlc3` is rising over `length` bars, `false` otherwise.
bool rising = ta.rising(hlc3, length)
//@variable Is `true` when `hlc3` is falling over `length` bars, `false` otherwise.
bool falling = ta.falling(hlc3, length)
//@variable Is `true` when a rising or falling pattern begins, `false` otherwise.
bool newDirection = (rising and not rising[1]) or (falling and not falling[1])

// Update the `directionLine` when `newDirection` is `true`. The line uses the default
// `xloc.bar_index`.
if newDirection
    directionLine := line.new(bar_index - length, hlc3[length], bar_index, hlc3, width
= 3)

//@variable The slope of the `directionLine`.
float slope = (directionLine.get_y2() - directionLine.get_y1()) /
(directionLine.get_x2() - directionLine.get_x1())
//@variable The value extrapolated from the `directionLine` at the `bar_index`.
float lineValue = line.get_price(directionLine, bar_index)

//@variable Is `color.green` when the `slope` is positive, `color.red` otherwise.
color slopeColor = slope > 0 ? color.green : color.red

// Update the color of the `directionLine`.

```

```

directionLine.set_color(slopeColor)
// Plot the `lineValue`.
plot(lineValue, "Extrapolated value", slopeColor, 3, plot.style_circles)

```

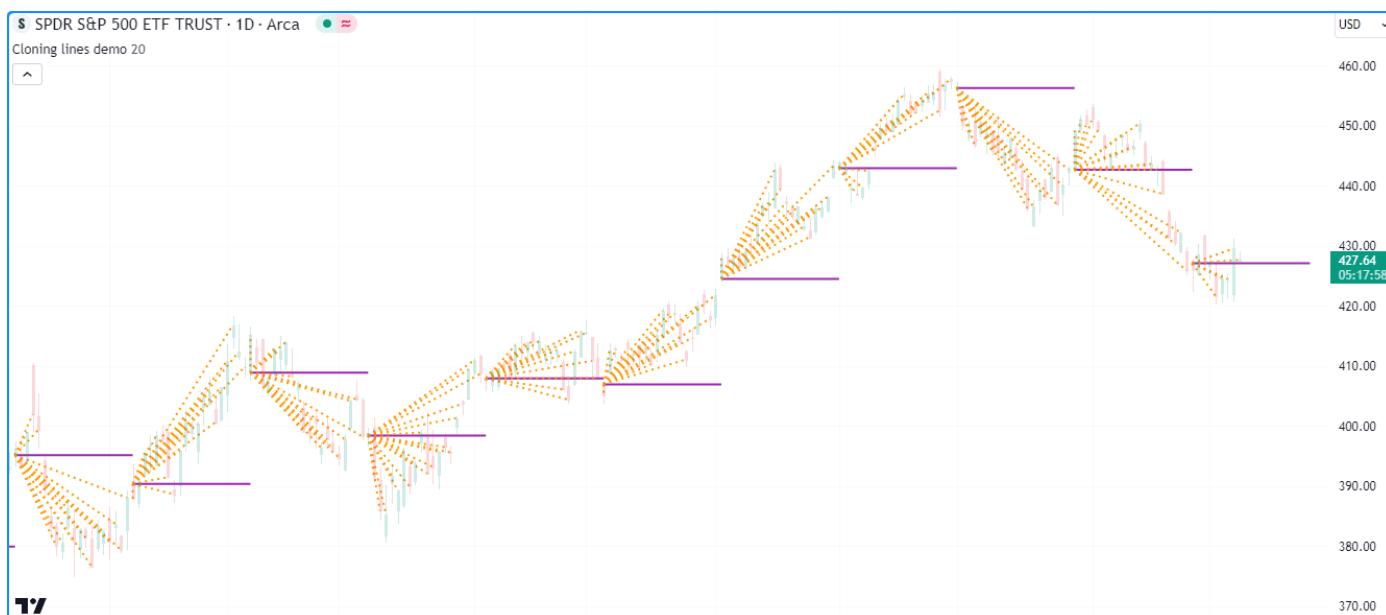
- 注意：

此示例调用 `line.new()` 函数的第二个重载，该函数使用 `x1`、`y1`、`x2` 和 `y2` 参数来定义线条的起点和终点。该 `x1` 值是 `length` 当前 `bar_index` 后面的柱，并且该 `y1` 值是该索引处的 `hlc3` 值。函数调用中的 `x2` 和使用当前柱的 `bar_index` 和 `hlc3` 值。`y2` `line.get_price()` 函数调用将 `directionLine` 视为无限扩展，无论其 `extend` 属性如何。该脚本仅显示图表上大约最后 50 条线，但推断值的绘图跨越整个图表的历史记录。

## 克隆系

脚本可以使用 `line.copy()` 函数克隆一行 `id` 及其所有属性。对复制的线实例进行的任何更改都不会影响原始线实例。

例如，此脚本在每个柱的 `开盘价` `length` 处创建一条水平线，并将其分配给一个 `mainLine` 变量。在所有其他栏上，它 `copiedLine` 使用 `line.copy()` 创建一个并调用 `line.set_*`() 函数来修改其属性。如下所示，更改 `copiedLine` 不会 `mainLine` 以任何方式影响：



```

//@version=5
indicator("Cloning lines demo", overlay = true, max_lines_count = 500)

//@variable The number of bars between each new mainLine assignment.
int length = input.int(20, "Length", 2, 500)

//@variable The first `chart.point` used by the `mainLine`. Contains `index` and `time` information.
firstPoint = chart.point.now(open)
//@variable The second `chart.point` used by the `mainLine`. Does not contain `time` information.
secondPoint = chart.point.from_index(bar_index + length, open)

```

```

//@variable A horizontal line drawn at the `open` price once every `length` bars.
var line mainLine = na

if bar_index % length == 0
    // Assign a new line to the `mainLine` that connects the `firstPoint` to the
    `secondPoint`.
    // This line uses the `index` fields from both points as x-coordinates.
    mainLine := line.new(firstPoint, secondPoint, color = color.purple, width = 2)

//@variable A copy of the `mainLine`. Changes to this line do not affect the original.
line copiedLine = line.copy(mainLine)

// Update the color, style, and second point of the `copiedLine`.
line.set_color(copiedLine, color.orange)
line.set_style(copiedLine, line.style_dotted)
line.set_second_point(copiedLine, chart.point.now(close))

```

- 注意：

`index` 的字段是 `secondPoint` 超出 `length` 当前 `bar_index` 的柱。由于 `xloc.bar_index` 允许的最大 x 坐标为，因此我们将输入的设为 500。`bar_index + 500``maxval``length`

## 删除行

`id` 要删除脚本绘制的线条，请使用 [line.delete\(\)](#) 函数。此函数从脚本中删除线条实例及其在图表上的绘制。

当人们只想在任何给定时间在图表上保留特定数量的线条或随着图表的进展有条件地删除图形时，删除线条实例通常很方便。

例如，每当 [RSI](#) 穿过其 [EMA](#) 时，[此脚本就会使用 extend.right 属性绘制一条水平线](#)。

该脚本将所有行 ID 存储在一个 `lines` 数组中，该 [数组用作队列](#) `numberOfLines`，仅在图表上显示最后一个。当 [数组的大小](#) 超过指定的值时，脚本会使用 [array.shift\(\)](#) 删除数组中最旧的行 ID，并使用 [line.delete\(\)](#) 删除它：`numberOfLines`



```

//@version=5

//@variable The maximum number of lines allowed on the chart.
const int MAX_LINES_COUNT = 500

indicator("Deleting lines demo", "RSI cross levels", max_lines_count = MAX_LINES_COUNT)

//@variable The length of the RSI.
int rsiLength = input.int(14, "RSI length", 2)
//@variable The length of the RSI's EMA.
int emaLength = input.int(28, "RSI average length", 2)
//@variable The maximum number of lines to keep on the chart.
int numberofLines = input.int(20, "Lines on the chart", 0, MAX_LINES_COUNT)

//@variable An array containing the IDs of lines on the chart.
var array<line> lines = array.new<line>()

//@variable An `rsiLength` RSI of `close`.
float rsi = ta.rsi(close, rsiLength)
//@variable A `maLength` EMA of the `rsi`.
float rsiMA = ta.ema(rsi, emaLength)

if ta.cross(rsi, rsiMA)
    //@variable The color of the horizontal line.
    color lineColor = rsi > rsiMA ? color.green : color.red
    // Draw a new horizontal line. Uses the default `xloc.bar_index`.
    newLine = line.new(bar_index, rsiMA, bar_index + 1, rsiMA, extend = extend.right,
color = lineColor, width = 2)
    // Push the `newLine` into the `lines` array.
    lines.push(newLine)
    // Delete the oldest line when the size of the array exceeds the specified
    `numberofLines`.
    if array.size(lines) > numberofLines

```

```

line.delete(lines.shift())

// Plot the `rsi` and `rsiMA`.

plot(rsi, "RSI", color.new(color.blue, 40))
plot(rsiMA, "EMA of RSI", color.new(color.gray, 30))

```

- 注意：

我们声明了一个 `MAX_LINES_COUNT` 具有“const int”限定类型的变量，脚本将其用作 `indicator()` `max_lines_count` 函数中的 和 分配给该变量的 `input.int()` 的。 `maxval``numberOfLines` 此示例使用 `line.new()` 函数的第二个重载，该函数独立指定 `x1`、`y1`、`x2` 和 `y2` 坐标。

## 盒子

命名空间中的内置函数 `box.*` 创建和管理 [盒子](#) 对象：

- `box.new()` 函数创建一个新盒子。
- 这些 `box.set_*` 函数修改框属性。
- 这些 `box.get_*` 函数从盒子实例中检索值。
- `box.copy()` 函数克隆一个盒子实例。
- `box.delete()` 函数删除一个盒子实例。
- `box.all` 变量引用一个只读 [数组](#)，[其中](#) 包含脚本显示的所有框的 ID。数组的 [大小](#) 取决于 `indicator()` 或 `strategy()` 声明语句的大小以及脚本绘制的框的数量。`max_boxes_count`

与 `lines` 一样，用户可以调用 `box.set_*`、`box.get_*`、`box.copy()` 和 `box.delete()` 内置函数或 [方法](#)。

## 创建盒子

`box.new()` 函数创建一个新的 [框](#) 对象以显示在图表上。它有以下签名：

```

box.new(top_left, bottom_right, border_color, border_width, border_style, extend, xloc,
bgcolor, text, text_size, text_color, text_halign, text_valign, text_wrap,
text_font_family) → series box

```

```

box.new(left, top, right, bottom, border_color, border_width, border_style, extend,
xloc, bgcolor, text, text_size, text_color, text_halign, text_valign, text_wrap,
text_font_family) → series box

```

该函数的第一个重载包括 `top_left` 和 `bottom_right` 参数，它们接受分别代表框的左上角和右下角的 [Chart.point](#) 对象。该函数从这些图表点复制信息来设置框角的坐标。是否使用和点的 `index` 或字段作为 `x` 坐标取决于函数的值。`time``top_left``bottom_right``xloc`

第二个重载指定框的 `left`、`top`、`right` 和边缘。`bottom` 参数接受指定框的左、右 x 坐标的 `int left` 值，该值可以是条形索引或时间值，具体取决于函数调用中的值。和参数接受代表框顶部和底部 y 坐标的浮点值。`right``xloc``top``bottom`

该函数的附加参数在两个重载中是相同的：

- `border_color`

指定所有四个框边框的颜色。默认为 [color.blue](#)。

- `border_width`

指定边框的宽度（以像素为单位）。其默认值为 1。

- `border_style`

指定边框的样式，可以是本页[“框样式”](#)部分中的任何选项。

- `extend`

确定框的边框是否无限延伸超出左或右 x 坐标。它接受以下值之一：[extend.left](#)、[extend.right](#)、[extend.both](#) 或 [extend.none](#)（默认）。

- `xloc`

确定框的左边缘和右边缘是否使用条形索引或时间值作为 x 坐标。默认为 [xloc.bar\\_index](#)。在第一个重载中，[xloc.bar\\_index](#) `xloc` 的值意味着该函数将使用和图表点的字段，[xloc.bar\\_time](#) 的值意味着它将使用它们的字段。`index``top_left``bottom_right``xloc``time` 在第二个重载中，使用 [xloc.bar\\_index](#) `xloc` 值意味着该函数将和值视为柱索引，而 [xloc.bar\\_time](#) 意味着它将将它们视为时间戳。`left``right` 当指定的 x 坐标表示柱索引值时，请务必注意允许的最小 x 坐标为。对于较大的偏移量，可以使用 [xloc.bar\\_time](#)。`bar_index - 9999`

- `bgcolor`

指定框内空间的背景颜色。默认值为 [color.blue](#)。

- `text`

要在框中显示的文本。默认情况下，其值为空字符串。

- `text_size`

指定框中文本的大小。它接受以下值之一：[size.tiny](#)、[size.small](#)、[size.normal](#)、[size.large](#)、[size.huge](#) 或 [size.auto](#)（默认）。

- `text_color`

控制文本的颜色。它的默认值是 [color.black](#)。

- `text_halign`

指定文本在框边界内的水平对齐方式。它接受以下其中一项：[text.align\\_left](#)、[text.align\\_right](#) 或 [text.align\\_center](#)（默认）。

- `text_valign`

指定文本在框边界内的垂直对齐方式。它接受以下其中一项：[text.align\\_top](#)、[text.align\\_bottom](#) 或 [text.align\\_center](#)（默认）。

- `text_wrap`

确定框是否将文本包裹在其中。如果其值为 `text.wrap_auto`，则该框将包裹文本以确保它不会超出其垂直边框。当包裹的文本延伸超过底部时，它还会剪辑它。如果值为 `text.wrap_none`，则该框将文本显示在可以超出其边框的单行上。默认为`text.wrap_none`。

- `text_font_family`

定义框文本的字体系列。使用 `font.family_default` 以系统默认字体显示框的文本。`font.family_monospace` 以等宽格式显示 文本。默认值为 `font.family_default`。

让我们编写一个简单的脚本来在图表上显示方框。下面的示例绘制一个框，将每个条形的 **最高值** 和 **最低值** 从当前条形的水平中心投影到下一个可用条形的中心。

在每个柱上，脚本通过`Chart.point.now()`和`Chart.point_from_index()` `topLeft` 创建并 `bottomRight` 指向，然后调用`box.new()`构造一个新框并将其显示在图表上。它还使用`bgcolor()`突出显示未经确认的图表栏上的背景，以指示它重新绘制该框，直到该栏的最后一次更新：



```
//@version=5
indicator("Creating boxes demo", overlay = true)

//@variable The `chart.point` for the top-left corner of the box. Contains `index` and
`time` information.
topLeft = chart.point.now(high)
//@variable The `chart.point` for the bottom-right corner of the box. Does not contain
`time` information.
bottomRight = chart.point.from_index(bar_index + 1, low)

// Draw a box using the `topLeft` and `bottomRight` corner points. Uses the `index`
fields as x-coordinates.
box.new(topLeft, bottomRight, color.purple, 2, bgcolor = color.new(color.gray, 70))

// Color the background on the unconfirmed bar.
bgcolor(barstate.isconfirmed ? na : color.new(color.orange, 70), title = "Unconfirmed
bar highlight")
```

- 注意：

该 `bottomRight` 点的场比中的 `index` 场大一格。如果角的 `x` 坐标相等，脚本将在每个条的水平中心绘制一条垂直线，类似于本页的[创建线](#)部分中的示例。`index``topLeft` 与 `lines` 类似，如果 `topLeft` 和 `bottomRight` 包含相同的坐标，则该框不会显示在图表上，因为它们之间没有空间可供绘制。然而，它的 ID 仍然存在。该脚本仅显示图表上大约最后 50 个框，因为我们没有 `max_boxes_count` 在[Indicator\(\)](#) 函数调用中指定 `a`。

## 修改框

命名空间中存在多个 `setter` 函数，允许脚本修改 [盒子](#) `box.*` 对象的属性：

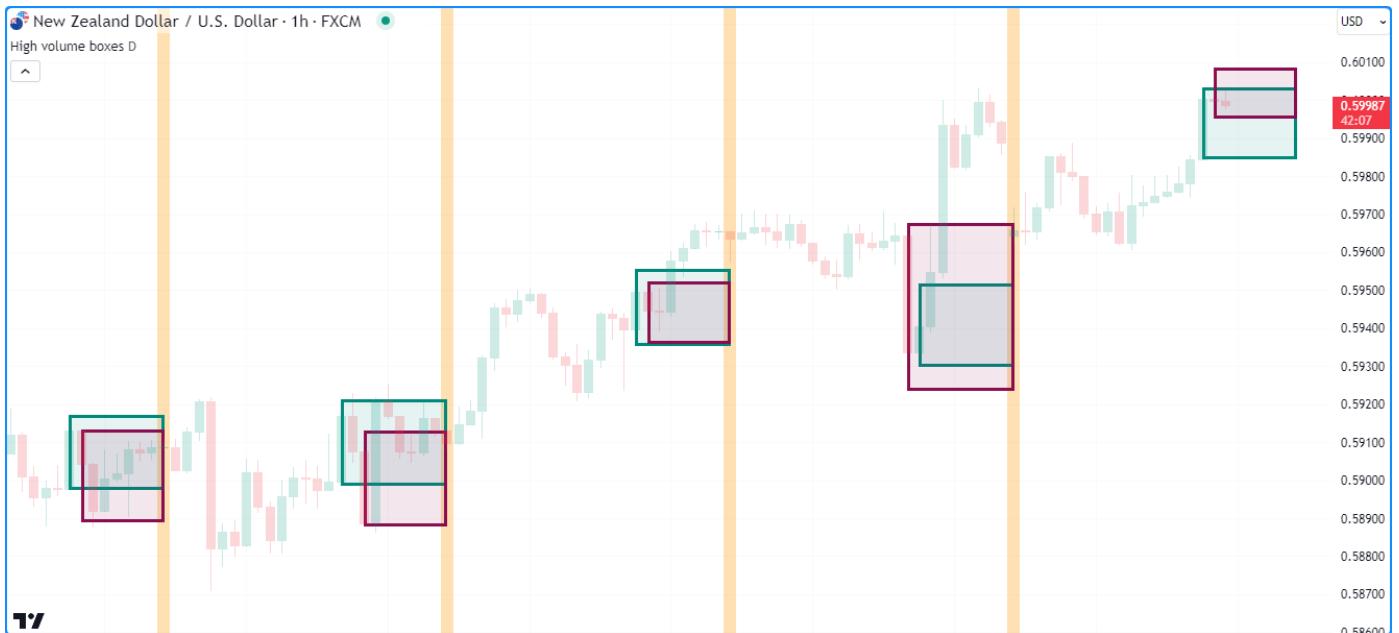
- `box.set_top_left_point()` 和 `box.set_bottom_right_point()` `id` 分别使用指定的信息更新框的左上角和右下角坐标 `point`。
- `box.set_left()` 和 `box.set_right()` 将框的左或右 `x` 坐标设置 `id` 为新 `left/right` 值，该值可以是条形索引或时间值，具体取决于框的 `xloc` 属性。
- `box.set_top()` 和 `box.set_bottom()` 将框的顶部或底部 `y` 坐标设置 `id` 为新 `top/bottom` 值。
- `box.set_lefttop()` 设置框的 `left` 和 `top` 坐标 `id`，`box.set_rightbottom()` 设置框的 `right` 和 `bottom` 坐标。
- `box.set_border_color()`、`box.set_border_width()` 和 `box.set_border_style()` 分别更新框边框的 `color`、`width`、和。 `style``id`
- `box.set_extend()` `extend` 设置盒子的水平属性 `id`。
- `box.set_bgcolor()` 将框内空间的颜色设置 `id` 为新的 `color`。
- `box.set_text()`、`box.set_text_size()`、`box.set_text_color()`、`box.set_text_halign()`、`box.set_text_valign()`、`box.set_text_wrap()` 和 `box.set_text_font_family()` 更新 `id` 框的文本相关属性。

与 `line.*` 命名空间中的 `setter` 函数一样，所有框 `setter` 都直接修改 `id` 框而不返回值，并且每个 `setter` 函数都接受“系列”参数。

请注意，与 `lines` 不同，`box.*` 命名空间不包含用于修改框的 `setter` 函数 `xloc`。用户必须针对此类情况[创建](#)一个具有所需设置的新框。`xloc`

此示例使用框来可视化用户定义的最高 `交易量` `timeframe` 的向上和向下柱的范围。当脚本检测到 中的更 改 `timeframe` 时，它会为其和变量分配 [新框](#)，重置其值，并突出显示图表背景。`upBox``downBox``upVolume``downVolume`

当向上或向下柱的 `交易量` 超过 `upVolume` 或时 `downVolume`，脚本会更新交易量跟踪变量并调用 `box.set_top_left_point()` 和 `box.set_bottom_right_point()` 来更新 `upBox` 或 `downBox` 坐标。设置器使用使用 `Chart.point.now()` 和 `Chart.point.from_time()` 创建的 [图表点](#) 的信息来投影该柱从当前时间到 [收盘时间](#) 的 [最高值](#) 和 [最低值](#)：`timeframe`



```

//@version=5
indicator("Modifying boxes demo", "High volume boxes", true, max_boxes_count = 100)

//@variable The timeframe of the calculation.
string timeframe = input.timeframe("D", "Timeframe")

//@variable A box projecting the range of the upward bar with the highest `volume` over
//the `timeframe`.
var box upBox = na
//@variable A box projecting the range of the downward bar with the lowest `volume`
//over the `timeframe`.
var box downBox = na
//@variable The highest volume of upward bars over the `timeframe`.
var float upVolume = na
//@variable The highest volume of downward bars over the `timeframe`.
var float downVolume = na

// Color variables.
var color upBorder    = color.teal
var color upFill      = color.new(color.teal, 90)
var color downBorder = color.maroon
var color downFill   = color.new(color.maroon, 90)

//@variable The closing time of the `timeframe`.
int closeTime = time_close(timeframe)
//@variable Is `true` when a new bar starts on the `timeframe`.
bool changeTF = timeframe.change(timeframe)

//@variable The `chart.point` for the top-left corner of the boxes. Contains `index`
//and `time` information.
topLeft = chart.point.now(high)

```

```

//@variable The `chart.point` for the bottom-right corner of the boxes. Does not
contain `index` information.
bottomRight = chart.point.from_time(closeTime, low)

if changeTF and not na(volume)
    if close > open
        // Update `upVolume` and `downVolume` values.
        upVolume := volume
        downVolume := 0.0
        // Draw a new `upBox` using `time` and `price` info from the `topLeft` and
        `bottomRight` points.
        upBox := box.new(topLeft, bottomRight, upBorder, 3, xloc = xloc.bar_time,
        bgcolor = upFill)
        // Draw a new `downBox` with `na` coordinates.
        downBox := box.new(na, na, na, na, downBorder, 3, xloc = xloc.bar_time, bgcolor
= downFill)
    else
        // Update `upVolume` and `downVolume` values.
        upVolume := 0.0
        downVolume := volume
        // Draw a new `upBox` with `na` coordinates.
        upBox := box.new(na, na, na, na, upBorder, 3, xloc = xloc.bar_time, bgcolor =
upFill)
        // Draw a new `downBox` using `time` and `price` info from the `topLeft` and
        `bottomRight` points.
        downBox := box.new(topLeft, bottomRight, downBorder, 3, xloc = xloc.bar_time,
        bgcolor = downFill)
// Update the ``upVolume`` and change the ``upBox`` coordinates when volume increases
on an upward bar.
else if close > open and volume > upVolume
    upVolume := volume
    box.set_top_left_point(upBox, topLeft)
    box.set_bottom_right_point(upBox, bottomRight)
// Update the ``downVolume`` and change the ``downBox`` coordinates when volume
increases on a downward bar.
else if close <= open and volume > downVolume
    downVolume := volume
    box.set_top_left_point(downBox, topLeft)
    box.set_bottom_right_point(downBox, bottomRight)

// Highlight the background when a new `timeframe` bar starts.
bgcolor(changeTF ? color.new(color.orange, 70) : na, title = "Timeframe change
highlight")

```

- 注意:

Indicator () 函数调用包含，这意味着脚本将保留图表上的最后 100 个框。`max_boxes_count = 100` 在此示例中，我们使用了 `box.new()` 的两个重载。在的第一个柱上，当柱上升时，脚本调用第一个重载，当柱下降时，脚本调用该重载。它使用第二个重载将具有 `na` 值的新框分配给该条上的另一个框变量。`timeframe``upBox``downBox`

## 盒子样式

`line.style_*` 用户可以在 `box.new()` 或 `box.set_border_style()` 函数调用中包含以下变量之一，以设置脚本绘制的框的边框样式：

争论	盒子
<code>line.style_solid</code>	
<code>line.style_dotted</code>	
<code>line.style_dashed</code>	

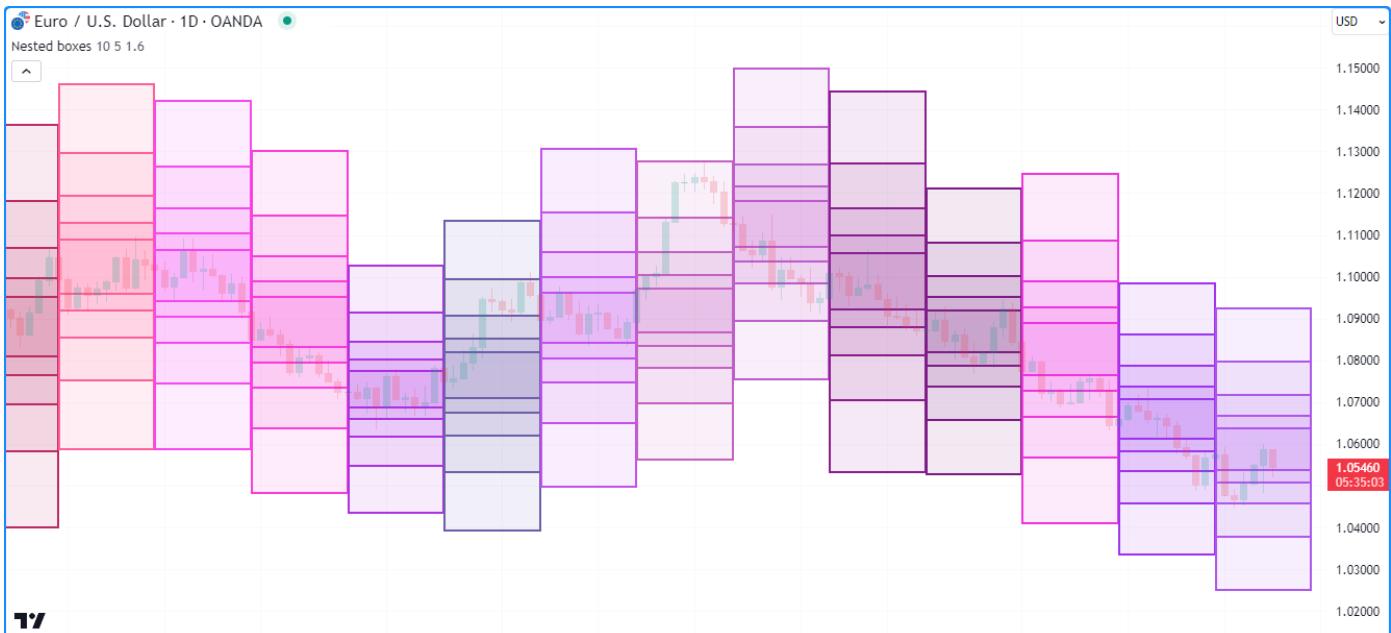
## 读取框值

命名 `box.*` 空间具有 `getter` 函数，允许脚本从盒子实例检索坐标值：

- `box.get_left()` 和 `box.get_right()` 分别获取盒子左边缘和右边缘的 x 坐标 `id`。返回的值是否表示柱索引或时间值取决于框的 `xloc` 属性。
- `box.get_top()` 和 `box.get_bottom()` 分别获取盒子的顶部和底部 y 坐标 `id`。

下面的示例绘制了方框来可视化一段时间内的假设价格范围 `length`。在每个新周期开始时，它使用平均蜡烛范围乘以输入来计算以 `h12` 价格为中心且具有一定高度 `scaleFactor` 的盒子的角点。绘制第一个框后，它会在 `for` 循环内创建新框。`initialRange``numberOfBoxes - 1`

在每次循环迭代中，脚本通过从只读 `box.all` 数组中检索最后一个 `lastBoxDrawn` 元素来获取，然后调用 `box.get_top()` 和 `box.get_bottom()` 来获取其 y 坐标。它使用这些值来计算比以前高一倍的新框的坐标：`scaleFactor`



```

//@version=5
indicator("Reading box values demo", "Nested boxes", overlay = true, max_boxes_count = 500)

//@variable The number of bars in the range calculation.
int length = input.int(10, "Length", 2, 500)
//@variable The number of nested boxes drawn on each period.
int numberOfBoxes = input.int(5, "Nested box count", 1)
//@variable The scale factor applied to each box.
float scaleFactor = input.float(1.6, "Scale factor", 1)

//@variable The initial box range.
float initialRange = scaleFactor * ta.sma(high - low, length)

if bar_index % length == 0
    //@variable The top-left `chart.point` for the initial box. Does not contain `time` information.
    topLeft = chart.point.from_index(bar_index, h12 + initialRange / 2)
    //@variable The bottom-right `chart.point` for the initial box. Does not contain `time` information.
    bottomRight = chart.point.from_index(bar_index + length, h12 - initialRange / 2)

    // Calculate border and fill colors of the boxes.
    borderColor = color.rgb(math.random(100, 255), math.random(0, 100),
    math.random(100, 255))
    bgColor = color.new(borderColor, math.max(100 * (1 - 1/numberOfBoxes), 90))

    // Draw a new box using the `topLeft` and `bottomRight` points. Uses their `index` fields as x-coordinates.
    box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor)

    if numberOfBoxes > 1

```

```

// Loop to create additional boxes.
for i = 1 to numberOfBoxes - 1
    // @variable The last box drawn by the script.
    box lastBoxDrawn = box.all.last()

    // @variable The top price of the last box.
    float top = box.get_top(lastBoxDrawn)
    // @variable The bottom price of the last box.
    float bottom = box.get_bottom(lastBoxDrawn)

    // @variable The scaled range of the new box.
    float newRange = scaleFactor * (top - bottom) * 0.5
    // @variable The midpoint between the `bottom` and `top`.
    float middle = 0.5 * (top + bottom)

    // Update the `price` fields of the `topLeft` and `bottomRight` points.
    // This does not affect the coordinates of previous boxes.
    topLeft.price := h12 + newRange
    bottomRight.price := h12 - newRange

    // Draw a new box using the updated `topLeft` and `bottomRight` points.
    box.new(topLeft, bottomRight, borderColor, 2, bgcolor = bgColor)

```

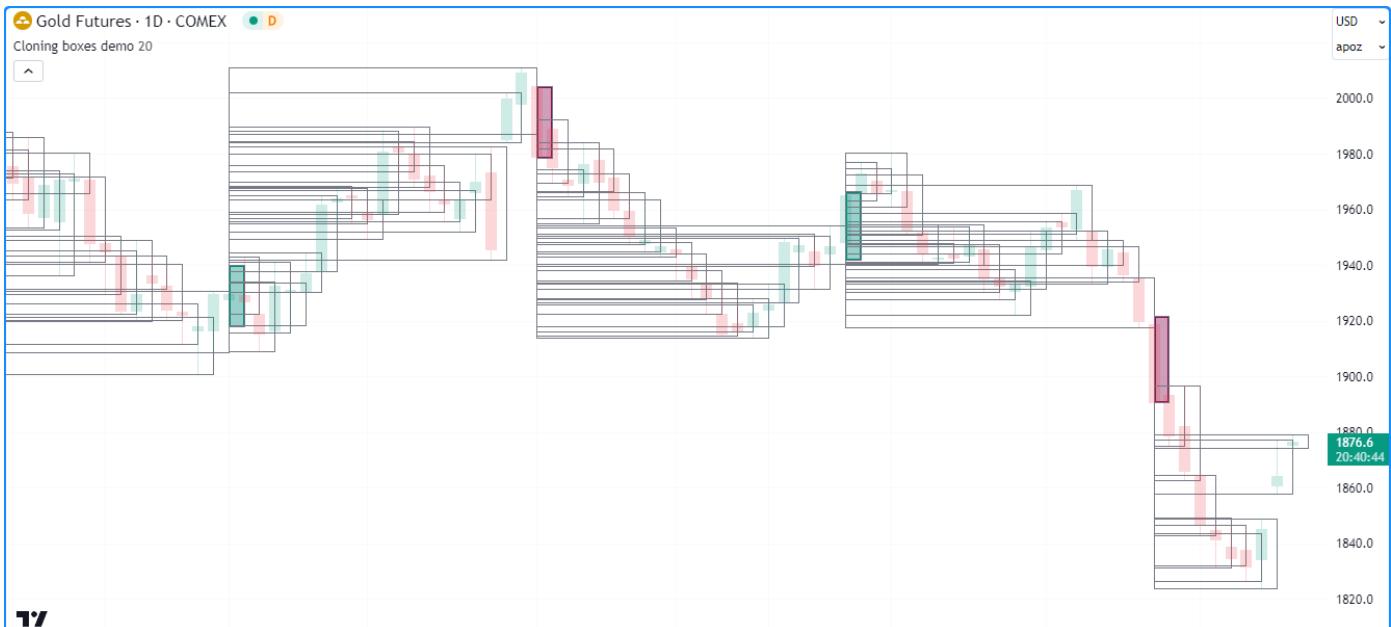
- 注意:

Indicator () 函数调用使用，这意味着脚本可以在图表上显示最多 500 个框。max\_boxes\_count = 500 每张图在索引之外都有一个 right 索引 length 条 left。由于这些绘图的 x 坐标未来最多可达 500 个柱，因此我们将输入 maxval 的值设置 length 为 500。在每个新周期，脚本使用随机的 color.rgb() 值作为框的 border\_color 和。bgcolor 每个 box.new() 调用都会从分配给和变量的 Chart.point 对象中复制坐标，这就是为什么脚本可以在每次循环迭代中修改它们的字段而不影响其他框。topLeft``bottomRight``price

## 克隆盒

要克隆特定的 box id，请使用 [box.copy\(\)](#)。此函数复制框及其属性。对复制框的任何更改都不会影响原始框。

例如，此脚本 originalBox 在第一个柱上声明一个变量，并为每个柱分配一个 [新框 length](#)。在其他栏上，它使用 [box.copy\(\)](#) 创建 copiedBox 并调用 [box.set\\_\\*\(\)](#) 函数来 [修改](#) 其属性。如下图所示，这些更改不会修改 originalBox：



```

//@version=5
indicator("Cloning boxes demo", overlay = true, max_boxes_count = 500)

//@variable The number of bars between each new mainLine assignment.
int length = input.int(20, "Length", 2)

//@variable The `chart.point` for the top-left of the `originalBox`. Contains `time` and `index` information.
topLeft = chart.point.now(high)
//@variable The `chart.point` for the bottom-right of the `originalBox`. Does not contain `time` information.
bottomRight = chart.point.from_index(bar_index + 1, low)

//@variable A new box with `topLeft` and `bottomRight` corners on every `length` bars.
var box originalBox = na

//@variable Is teal when the bar is rising, maroon when it's falling.
color originalColor = close > open ? color.teal : color.maroon

if bar_index % length == 0
    // Assign a new box using the `topLeft` and `bottomRight` info to the `originalBox`.
    // This box uses the `index` fields from the points as x-coordinates.
    originalBox := box.new(topLeft, bottomRight, originalColor, 2, bgcolor =
color.new(originalColor, 60))
else
    //@variable A clone of the `originalBox`.
    box copiedBox = box.copy(originalBox)
    // Modify the `copiedBox`. These changes do not affect the `originalBox`.
    box.set_top(copiedBox, high)
    box.set_bottom_right_point(copiedBox, bottomRight)
    box.set_border_color(copiedBox, color.gray)

```

```
box.set_border_width(copiedBox, 1)
box.set_bgcolor(copiedBox, na)
```

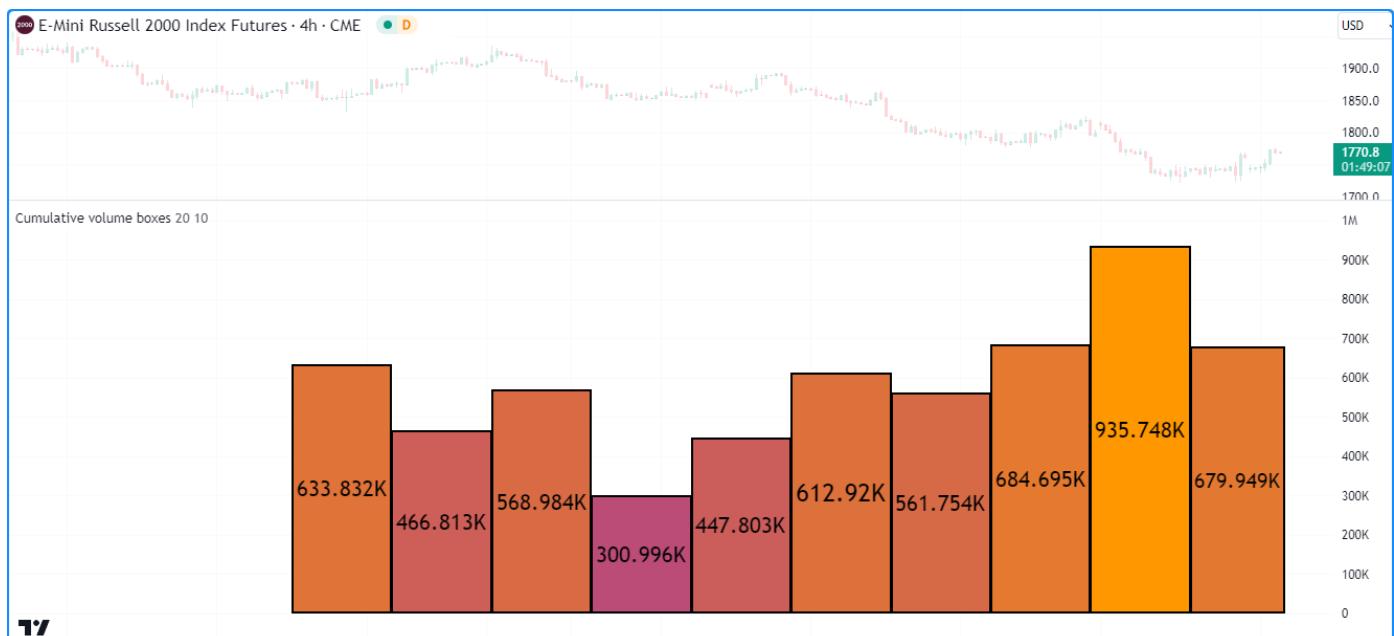
## 删除框

要删除脚本绘制的框，请使用`box.delete()`。与`*.delete()`其他绘图命名空间中的函数一样，此函数可以方便地有条件地删除框或维护图表上特定数量的框。

此示例显示表示周期性累积体积值的框。该脚本 创建一个新的盒子 ID，并将其存储在`boxes`数组中`length`。如果数组的大小 超过指定的值，脚本将使用`array.shift()``numberOfBoxes`从数组中删除最旧的框，并使用`box.delete()`删除它。

在其他柱上，它 通过修改 数组中最后一个框的 来累积 每个周期的交易量。然后，该脚本使用`for 循环`查找所有数组框的，并根据相对于的`box.get_top()` 值设置每个框的 渐变色

色：`top``boxes``highestTop``bgcolor``highestTop`



```
//@version=5

//@variable The maximum number of boxes to show on the chart.
const int MAX_BOXES_COUNT = 500

indicator("Deleting boxes demo", "Cumulative volume boxes", format = format.volume,
max_boxes_count = MAX_BOXES_COUNT)

//@variable The number of bars in each period.
int length = input.int(20, "Length", 1)
//@variable The maximum number of volume boxes in the calculation.
int numberOfBoxes = input.int(10, "Number of boxes", 1, MAX_BOXES_COUNT)

//@variable An array containing the ID of each box displayed by the script.
var boxes = array.new<box>()
```

```

if bar_index % length == 0
    // Push a new box into the `boxes` array. The box has the default `xloc.bar_index`
    // property.
    boxes.push(box.new(bar_index, 0, bar_index + 1, 0, #000000, 2, text_color =
    #000000))
    // Shift the oldest box out of the array and delete it when the array's size
    exceeds the `numberOfBoxes`.
    if boxes.size() > numberOfBoxes
        box.delete(boxes.shift())

//@variable The last box drawn by the script as of the current chart bar.
box lastBox = boxes.last()
// Add the current bar's volume to the top of the `lastBox` and update the `right`
// index.
lastBox.set_top(lastBox.get_top() + volume)
lastBox.set_right(bar_index + 1)
// Display the top of the `lastBox` as volume-formatted text.
lastBox.set_text(str.tostring(lastBox.get_top(), format.volume))

//@variable The highest `top` of all boxes in the `boxes` array.
float highestTop = 0.0
for id in boxes
    highestTop := math.max(id.get_top(), highestTop)

// Set the `bgcolor` of each `id` in `boxes` with a gradient based on the ratio of its
// `top` to the `highestTop`.
for id in boxes
    id.set_bgcolor(color.from_gradient(id.get_top() / highestTop, 0, 1, color.purple,
    color.orange))

```

- 注意：

在代码的顶部，我们声明了一个 `MAX_BOXES_COUNT` 具有“const int”限定类型的变量。我们使用这个值作为 `indicator()` `max_boxes_count` 函数中的 和输入的最大可能值。`numberOfBoxes` 此脚本使用 `box.new()` 函数的第二个重载，该函数分别指定框的 `left`、`top`、`right` 和 `bottom` 坐标。我们已将 `format.volume` 作为 `format` 参数包含在 `indicator()` 调用中，它告诉脚本图表窗格的 y 轴代表交易量值。每个框还将其 最高值 显示为 体积格式的文本。

## 折线

Pine Script™ 多段线是高级绘图，它 使用直线或曲线段按顺序连接图表.点实例数组 中的坐标。

这些功能强大的绘图可以在图表上的任何可用位置连接多达 10,000 个点，从而允许脚本绘制自定义系列、多边形和其他复杂的几何结构，否则使用 线条 或 框 对象很难或不可能绘制这些几何结构。

该 `polyline.*` 命名空间具有以下用于创建和管理 折线 对象的内置函数：

- `polyline.new()` 函数创建一个新的折线实例。

- `polyline.delete()` 函数删除现有的折线实例。
- `polyline.all` 变量引用一个只读数组，其中包含脚本显示的所有折线的 ID。数组的大小取决于 `indicator()` 或 `strategy()` 声明语句的大小以及脚本绘制的折线条数。`max_polylines_count`

与 [lines](#) 或 [box](#) 不同，`polyline` 没有修改或读取其属性的功能。要在图表上重绘多段线，可以删除现有实例并创建一条具有所需更改的新多段线。

## 创建折线

`polyline.new()` 函数创建一个新的折线实例以显示在图表上。它具有以下签名：

```
polyline.new(points, curved, closed, xloc, line_color, fill_color, line_style,
line_width) → series polyline
```

以下八个参数影响折线绘制的行为：

- `points`

接受一个 [Chart.point](#) 对象数组，这些对象确定折线中每个点的坐标。该图从第一个开始，按顺序连接数组中每个元素的坐标。折线是否使用每个 [图表点](#) 的或字段作为其 x 坐标取决于函数调用中的值。`index``time``xloc`

- `curved`

指定绘图是否使用曲线段来连接数组中的每个 [图表点](#) `points`。默认值为 `false`，表示它使用直线段。

- `closed`

控制折线是否将数组中的最后一个 [图表点](#) `points` 连接到第一个图表点，形成闭合折线。默认值为 `false`。

- `xloc`

指定折线使用数组中每个 [图表点](#) 的哪个字段 `points` 作为其 x 坐标。当其值为 `xloc.bar_index` 时，该函数使用这些 `index` 字段来创建折线。当其值为 `xloc.bar_time` 时，该函数使用这些 `time` 字段。默认值为 `xloc.bar_index`。

- `line_color`

指定多段线绘图中所有线段的颜色。默认为 `color.blue`。

- `fill_color`

控制由折线图填充的封闭空间的颜色。它的默认值为 `na`。

- `line_style`

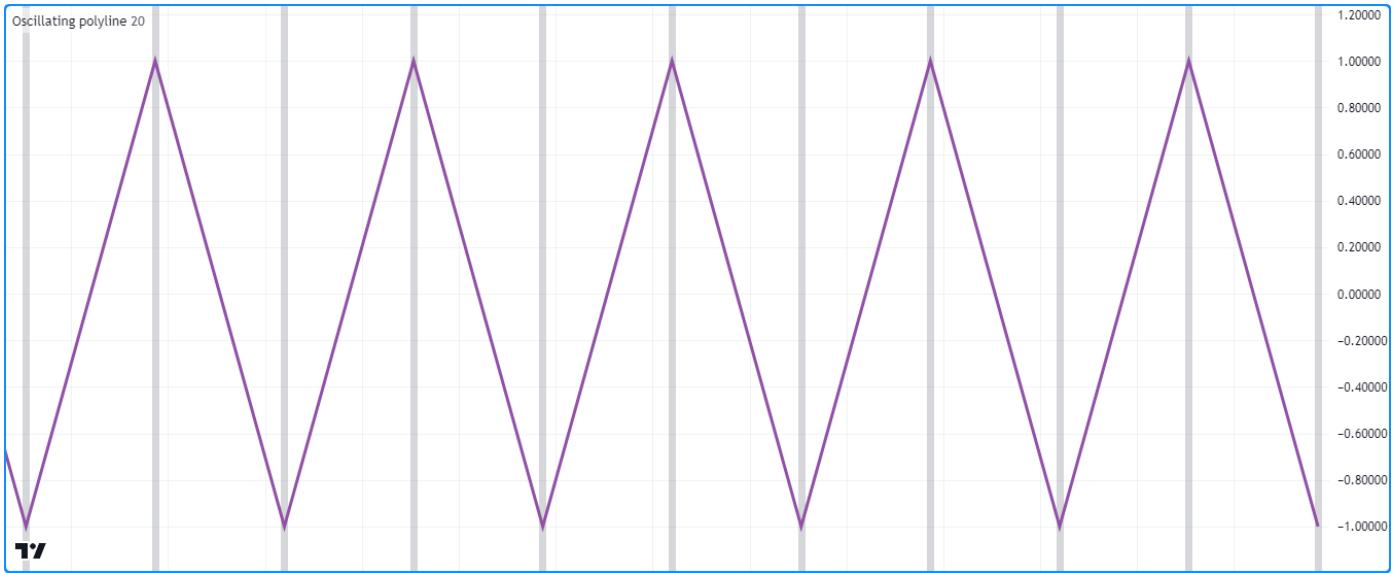
指定多段线的样式，可以是此页面的 [线条样式](#) 部分中的任何可用选项。默认为 `line.style_solid`。

- `line_width`

指定折线的宽度（以像素为单位）。默认值为 1。

此脚本演示了在图表上绘制折线的简单示例。它将具有交替值的新 [图表点推](#) 送到数组中，并在每条柱上使用 `bgcolor()` 为背景着色。`price``points``length`

在 [最后一个确认的历史柱上](#)，脚本在图表上绘制一条新的折线，从第一个开始连接数组中每个 [图表点](#) 的坐标：



```

//@version=5
indicator("Creating polylines demo", "Oscillating polyline")

//@variable The number of bars between each point in the drawing.
int length = input.int(20, "Length between points", 2)

//@variable An array of `chart.point` objects to sequentially connect with a polyline.
var points = array.new<chart.point>()

//@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1
on each `newPoint`.
var int yValue = 1

//@variable Is `true` once every `length` bars, `false` otherwise.
bool newPoint = bar_index % length == 0

if newPoint
    // Push a new `chart.point` into the `points`. The new point contains `time` and
    `index` info.
    points.push(chart.point.now(yValue))
    // Change the sign of the `yValue`.
    yValue *= -1

// Draw a new `polyline` on the last confirmed historical chart bar.
// The polyline uses the `time` field from each `chart.point` in the `points` array as
x-coordinates.
if barstate.islastconfirmedhistory
    polyline.new(points, xloc = xloc.bar_time, line_color = #9151A6, line_width = 3)

// Highlight the chart background on every `newPoint` condition.
bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

```

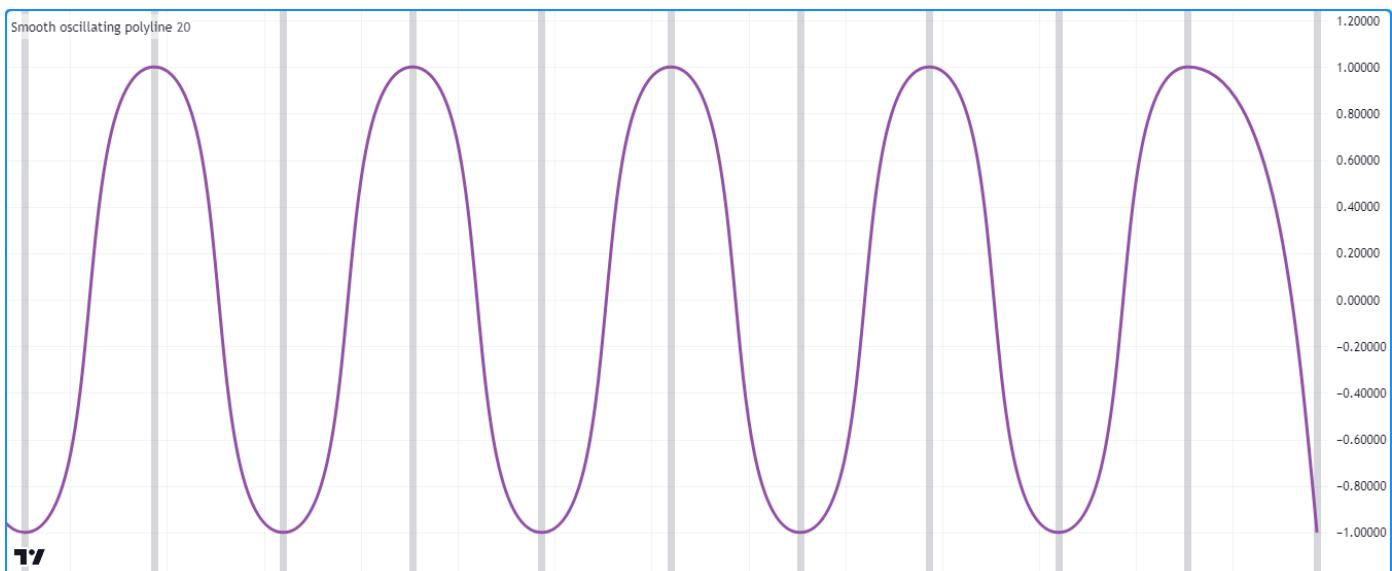
- 注意:

该脚本仅使用一条折线将数组中的每个图表点与直线段连接起来，并且该绘图从第一个条形图开始跨越整个可用图表数据。虽然使用lines可以达到类似的效果，但这样做需要在每次出现条件时都需要一个新的线newPoint实例，并且这样的绘图最多只能有500条线段。另一方面，该单个未闭合折线图最多可以包含9,999条线段。

## 曲线图

折线可以绘制用直线或方框无法绘制的曲线。启用polyline.new()函数的参数时，生成的折线会在其数组中每个图表点的坐标之间插入非线性值，以生成曲线效果。curved` `points

例如，我们前面示例中的“振荡折线”脚本使用直线段来生成类似于三角波的图形，这意味着波形在波峰和波谷之间呈锯齿形。如果我们将该示例中的polyline.new()调用curved中的参数设置为，则生成的绘图将使用曲线段连接点，产生类似于正弦波的平滑非线性形状：true



```
//@version=5
indicator("Curved drawings demo", "Smooth oscillating polyline")

//@variable The number of bars between each point in the drawing.
int length = input.int(20, "Length between points", 2)

//@variable An array of `chart.point` objects to sequentially connect with a polyline.
var points = array.new<chart.point>()

//@variable The y-coordinate of each point in the `points`. Alternates between 1 and -1
on each `newPoint`.
var int yValue = 1

//@variable Is `true` once every `length` bars, `false` otherwise.
bool newPoint = bar_index % length == 0

if newPoint
    // Push a new `chart.point` into the `points`. The new point contains `time` and
    `index` info.
```

```

points.push(chart.point.now(yValue))
// Change the sign of the `yValue`.
yValue *= -1

// Draw a new curved `polyline` on the last confirmed historical chart bar.
// The polyline uses the `time` field from each `chart.point` in the `points` array as
x-coordinates.
if barstate.islastconfirmedhistory
    polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6,
line_width = 3)

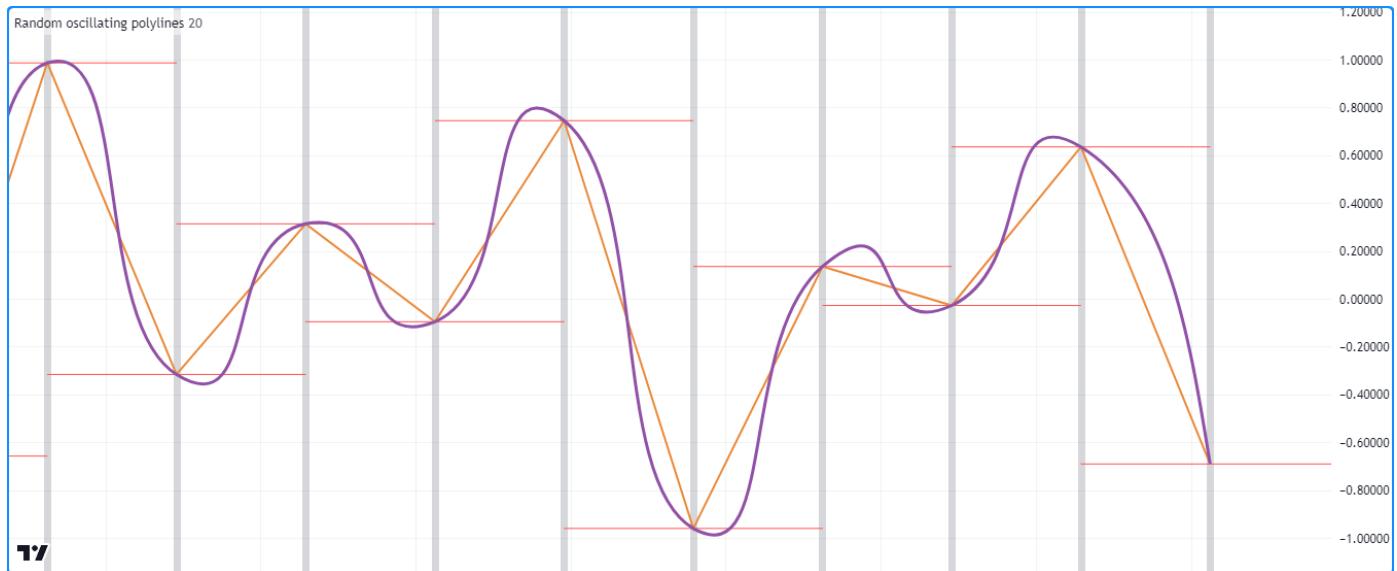
// Highlight the chart background on every `newPoint` condition.
bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

```

请注意，在此示例中，平滑曲线具有相对一致的行为，并且绘图的任何部分都不会超出其定义的坐标，但绘制弯曲多段线时情况并非总是如此。用于构造折线的数据严重影响其在点之间插值的平滑分段函数。在某些情况下，插值曲线可能超出其实际坐标。

让我们向示例数组中的图表点 `points` 添加一些变化来演示此行为。在下面的版本中，脚本将 `yValue` 与 图表 `.point.now()` 调用中的 随机 值相乘。

为了可视化该行为，此脚本还在数组中每个图表点的值处创建一条水平线，并显示另一条用直线段连接相同点的折线。正如我们在图表上看到的，两条折线都经过数组中的所有坐标。然而，弯曲折线有时会超出水平线指示的垂直边界，而使用直线段绘制的折线则不会： `price``points``points`



```

//@version=5
indicator("Curved drawings demo", "Random oscillating polylines")

//@variable The number of bars between each point in the drawing.
int length = input.int(20, "Length between points", 2)

//@variable An array of `chart.point` objects to sequentially connect with a polyline.
var points = array.new<chart.point>()

```

```

//@variable The sign of each `price` in the `points`. Alternates between 1 and -1 on
//each `newPoint`.
var int yValue = 1

//@variable Is `true` once every `length` bars.
bool newPoint = bar_index % length == 0

if newPoint
    // Push a new `chart.point` with a randomized `price` into the `points`.
    // The new point contains `time` and `index` info.
    points.push(chart.point.now(yValue * math.random()))
    // Change the sign of the `yValue`.
    yValue *= -1

    //@variable The newest `chart.point`.
    lastPoint = points.last()
    // Draw a horizontal line at the `lastPoint.price`. This line uses the default
    `xloc.bar_index`.
    line.new(lastPoint.index - length, lastPoint.price, lastPoint.index + length,
    lastPoint.price, color = color.red)

    // Draw two `polyline` instances on the last confirmed chart bar.
    // Both polylines use the `time` field from each `chart.point` in the `points` array as
    x-coordinates.
    if barstate.islastconfirmedhistory
        polyline.new(points, curved = false, xloc = xloc.bar_time, line_color = #EB8A3B,
        line_width = 2)
        polyline.new(points, curved = true, xloc = xloc.bar_time, line_color = #9151A6,
        line_width = 3)

    // Highlight the chart background on every `newPoint` condition.
    bgcolor(newPoint ? color.new(color.gray, 70) : na, title = "New point highlight")

```

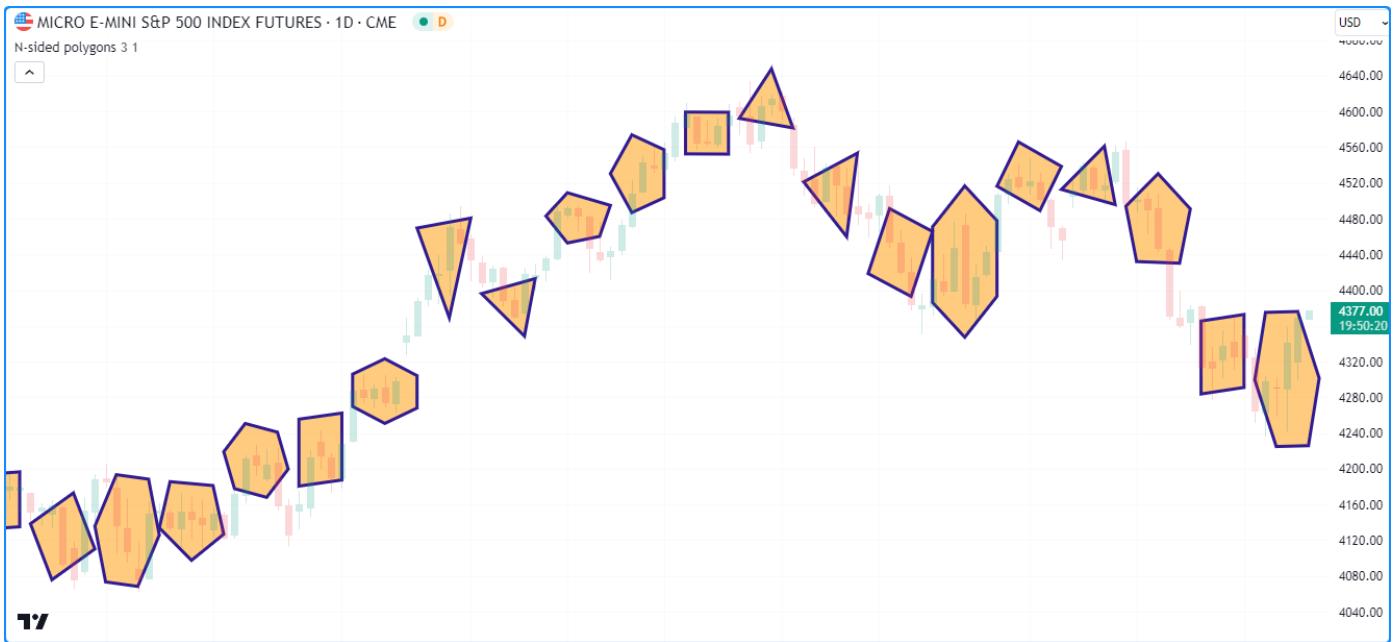
## 闭合形状

由于一条多段线可以包含许多直线或曲线段，并且该 `closed` 参数允许绘图连接其 数组中 第一个和最后一个图表点 的坐标，因此我们可以使用多段线来绘制许多不同类型的闭合多边形形状。`points`

让我们在 Pine 中绘制一些多边形。以下脚本定期绘制以 `hl2` 价格值为中心的随机多边形。

每次出现该 `newPolygon` 条件时，它都会清除 数组 `points`，根据 `math.random()` 值 计算新多边形绘图的 `numberOfSides` 和，然后使用`for 循环`将新图表点推入 包含来自椭圆路径的步进坐标的数组 中带和半轴。该脚本通过使用带有直线段的闭合折线连接数组 中的每个图表点 来绘制多边形

`: rotationOffset``numberOfSides``xScale``yScale``points`



```

//@version=5
indicator("Closed shapes demo", "N-sided polygons", true)

//@variable The size of the horizontal semi-axis.
float xScale = input.float(3.0, "X scale", 1.0)
//@variable The size of the vertical semi-axis.
float yScale = input.float(1.0, "Y scale") * ta.atr(2)

//@variable An array of `chart.point` objects containing vertex coordinates.
var points = array.new<chart.point>()

//@variable The condition that triggers a new polygon drawing. Based on the horizontal
axis to prevent overlaps.
bool newPolygon = bar_index % int(math.round(2 * xScale)) == 0 and barstate.isconfirmed

if newPolygon
    // Clear the `points` array.
    points.clear()

    //@variable The number of sides and vertices in the new polygon.
    int numberOfSides = int(math.random(3, 7))
    //@variable A random rotation offset applied to the new polygon, in radians.
    float rotationOffset = math.random(0.0, 2.0) * math.pi
    //@variable The size of the angle between each vertex, in radians.
    float step = 2 * math.pi / numberOfSides

    //@variable The counter-clockwise rotation angle of each vertex.
    float angle = rotationOffset

    for i = 1 to numberOfSides
        // @variable The approximate x-coordinate from an ellipse at the `angle`,
        rounded to the nearest integer.

```

```

int xValue = int(math.round(xScale * math.cos(angle))) + bar_index
//@variable The y-coordinate from an ellipse at the `angle`.
float yValue = yScale * math.sin(angle) + h12

// Push a new `chart.point` containing the `xValue` and `yValue` into the
`points` array.
// The new point does not contain `time` information.
points.push(chart.point.from_index(xValue, yValue))
// Add the `step` to the `angle`.
angle += step

// Draw a closed polyline connecting the `points`.
// The polyline uses the `index` field from each `chart.point` in the `points`
array.
polyline.new(
    points, closed = true, line_color = color.navy, fill_color =
color.new(color.orange, 50), line_width = 3
)

```

- 注意：

此示例显示了图表上最后约 50 条折线，因为我们没有 `max_polylines_count` 在 [Indicator\(\)](#) 函数调用中指定值。该 `yScale` 计算将 `input.float()` 乘以 `ta.atr(2)` 以使绘图的垂直比例适应最近的价格范围。生成的多边形的最大宽度是水平半轴 () 的两倍，四舍五入到最接近的整数。该条件使用该值来防止多边形绘图重叠。`2 * xScale``newPolygon` 该脚本将计算四舍五入 `xValue` 为最接近的整数，因为 [图表 index](#) 的 x 轴不包括小数条形索引，所以图表的字段仅接受 [int 值。](#)

## 删除折线

要删除特定的折线 `id`，请使用 [polyline.delete\(\)](#)。此函数从脚本中删除 [折线](#) 对象及其在图表上的绘制。

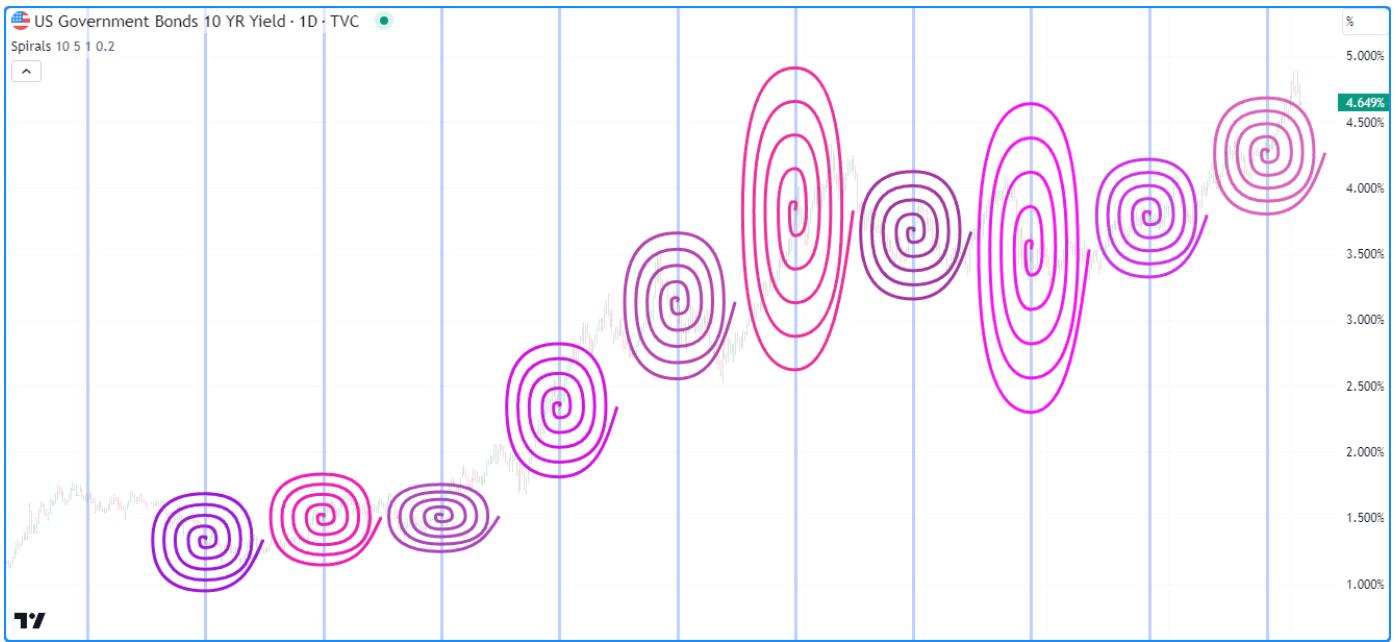
与其他绘图对象一样，我们可以使用 [polyline.delete\(\)](#) 来维护特定数量的折线绘图或有条件地从图表中删除绘图。

例如，下面的脚本定期绘制近似算术螺旋并将其折线 ID 存储在数组中，[该数组用作队列](#) 来管理其显示的绘图数量。

当 `newSpiral` 条件发生时，脚本会创建一个 `points` 数组并在 [for 循环](#) 中添加 [图表点](#)。在每次循环迭代中，它调用 [用户定义的函数](#) 来创建一个新的 [Chart.point](#)，其中包含来自相对于增长的椭圆路径的步进值。然后，该脚本创建一条随机颜色的弯曲多段线，连接来自 `points` 的坐标，并将其 ID 推入 [数组](#)。`spiralPoint()`

`angle``points``polylines`

当数组的 [大小](#) 超过指定的值时 `numberOfSpirals`，脚本使用 [array.shift\(\)](#) [删除最旧的折线](#)，[并使用 polyline.delete\(\) 删除对象](#)：



```

//@version=5

//@variable The maximum number of polylines allowed on the chart.
const int MAX_POLYLINES_COUNT = 100

indicator("Deleting polylines example", "Spirals", true, max_polyline_count =
MAX_POLYLINES_COUNT)

//@variable The number of spiral drawings on the chart.
int numberOoSpirals = input.int(10, "Spirals shown", 1, MAX_POLYLINES_COUNT)
//@variable The number of full spiral rotations to draw.
int rotations = input.int(5, "Rotations", 1)
//@variable The scale of the horizontal semi-axis.
float xScale = input.float(1.0, "X scale")
//@variable The scale of the vertical semi-axis.
float yScale = input.float(0.2, "Y scale") * ta.atr(2)

//@function Calculates an approximate point from an elliptically-scaled arithmetic
//spiral.
//@returns A `chart.point` with `index` and `price` information.
spiralPoint(float angle, int xOffset, float yOffset) =>
    result = chart.point.from_index(
        int(math.round(angle * xScale * math.cos(angle))) + xOffset,
        angle * yScale * math.sin(angle) + yOffset
    )

//@variable An array of polylines.
var polyline = array.new<polyline>()

//@variable The condition to create a new spiral.
bool newSpiral = bar_index % int(math.round(4 * math.pi * rotations * xScale)) == 0

```

```

if newSpiral
    // @variable An array of `chart.point` objects for the `spiral` drawing.
    points = array.new<chart.point>()
    // @variable The counter-clockwise angle between calculated points, in radians.
    float step = math.pi / 2
    // @variable The rotation angle of each calculated point on the spiral, in radians.
    float theta = 0.0
    // Loop to create the spiral's points. Creates 4 points per full rotation.
    for i = 0 to rotations * 4
        // @variable A new point on the calculated spiral.
        chart.point newPoint = spiralPoint(theta, bar_index, ohlc4)
        // Add the `newPoint` to the `points` array.
        points.push(newPoint)
        // Add the `step` to the `theta` angle.
        theta += step

        // @variable A random color for the new `spiral` drawing.
        color spiralColor = color.rgb(math.random(150, 255), math.random(0, 100),
math.random(150, 255))
        // @variable A new polyline connecting the spiral points. Uses the `index` field
from each point as x-coordinates.
        polyline spiral = polyline.new(points, true, line_color = spiralColor, line_width =
3)

        // Push the new `spiral` into the `polylines` array.
        polylines.push(spiral)
        // Shift the first polyline out of the array and delete it when the array's size
exceeds the `numberOfSpirals`.
        if polylines.size() > numberOfSpirals
            polyline.delete(polylines.shift())

// Highlight the background when `newSpiral` is `true`.
bgcolor(newSpiral ? color.new(color.blue, 70) : na, title = "New drawing highlight")

```

- 注意：

我们声明了一个 `MAX_POLYLINES_COUNT` 常量值为 100 的全局变量。脚本使用该常量作为 `Indicator()` `max_polyline_count` 函数中的值 和输入的值。`maxval` `numberOfSpirals` 与上一节 中的“N 边多边形”示例一样，我们将 x 坐标的计算四舍五入为最接近的整数，因为 `Chart.point` `index` 的字段只能接受 `int` 值。尽管绘图外观平滑，但每个折线的 `points` 数组每次螺旋旋转仅包含四个 `图表点` 对象。由于 `polyline.new()` 调用包含，因此每条折线都使用平滑曲线来连接它们，从而产生螺旋实际曲率的视觉近似值。`curved = true` `points` 每个螺旋的宽度约为，四舍五入到最接近的整数。我们在条件中使用该值来间隔每个绘图并防止重叠。`4 * math.pi * rotations * xScale` `newSpiral`

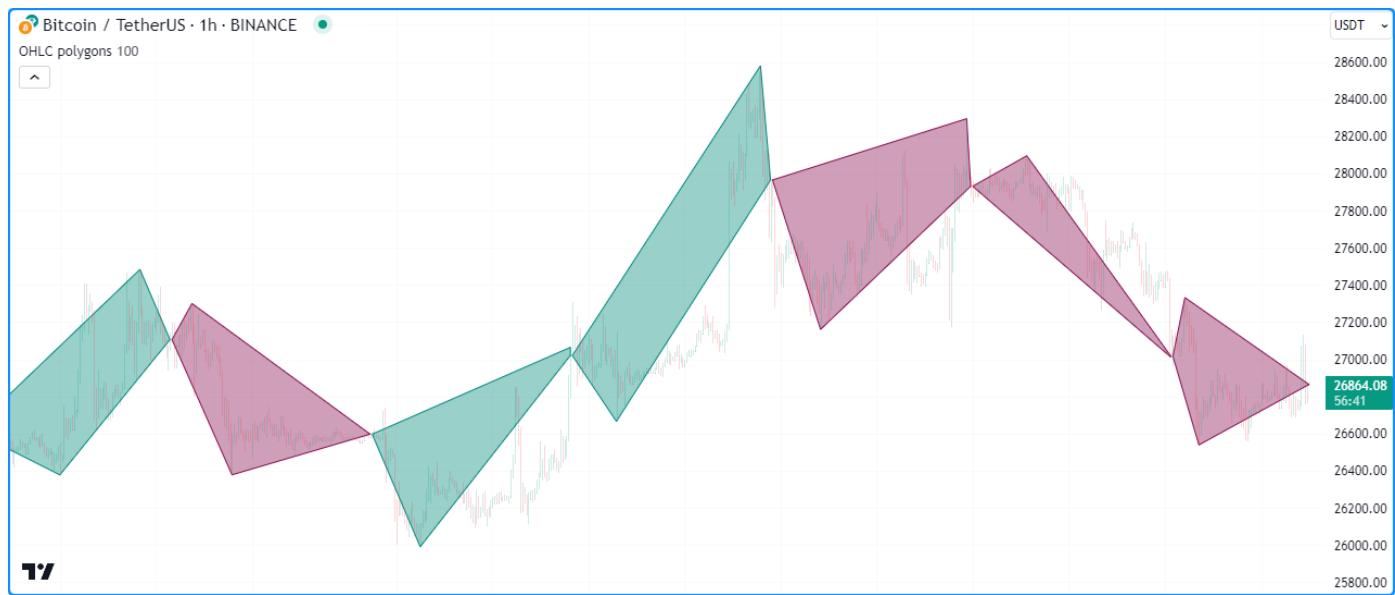
## 重画折线

在某些情况下，可能需要在脚本执行过程中更改多段线绘制。虽然 `polyline.*` 命名空间不包含内置的 setter 函数，但我们可以删除现有折线并分配具有所需更改的新实例\*\*来重新绘制变量或集合引用的折线。

以下示例使用 `polyline.delete()` 和 `polyline.new()` 调用来更新折线变量的值。

该脚本绘制闭合多段线，连接包含 `length` 柱线的周期的开盘点、最高点、最低点和收盘点。它在第一个条上创建一个 `currentDrawing` 变量，并在每个图表条上为其分配一个折线 ID。它使用 `openPoint`、`highPoint`、`lowPoint` 和 `closePoint` 变量来参考 [图表点](#)，以跟踪该时期发展中的 OHLC 值。当新值出现时，脚本将新的 `Chart.point` 对象分配给变量，使用 `array.from` 将它们收集在数组中，然后创建一条连接数组点坐标的 [新折线](#) 并将其分配给 `currentDrawing`

当 `newPeriod` 条件为 `false`（即当前周期未完成）时，脚本 [将删除](#) `currentDrawing` before [创建新的折线](#) 所引用的折线，从而生成随开发周期变化的动态绘图：



```
//@version=5
indicator("Redrawing polylines demo", "OHLC polygons", true, max_polylines_count = 100)

//@variable The length of the period.
int length = input.int(100, "Length", 1)

//@variable A `chart.point` representing the start of each period.
var chart.point openPoint = na
//@variable A `chart.point` representing the highest point of each period.
var chart.point highPoint = na
//@variable A `chart.point` representing the lowest point of each period.
var chart.point lowPoint = na
//@variable A `chart.point` representing the current bar's closing point.
closePoint = chart.point.now(close)

//@variable The current period's polyline drawing.
var polyline currentDrawing = na
```

```

//@variable Is `true` once every `length` bars.
bool newPeriod = bar_index % length == 0

if newPeriod
    // Assign new chart points to the `openPoint`, `highPoint`, and `closePoint`.
    openPoint := chart.point.now(open)
    highPoint := chart.point.now(high)
    lowPoint := chart.point.now(low)
else
    // Assign a new `chart.point` to the `highPoint` when the `high` is greater than
    its `price`.
    if high > highPoint.price
        highPoint := chart.point.now(high)
    // Assign a new `chart.point` to the `lowPoint` when the `low` is less than its
    `price`.
    if low < lowPoint.price
        lowPoint := chart.point.now(low)

//@variable Is teal when the `closePoint.price` is greater than the `openPoint.price`,
maroon otherwise.
color drawingColor = closePoint.price > openPoint.price ? color.teal : color.maroon

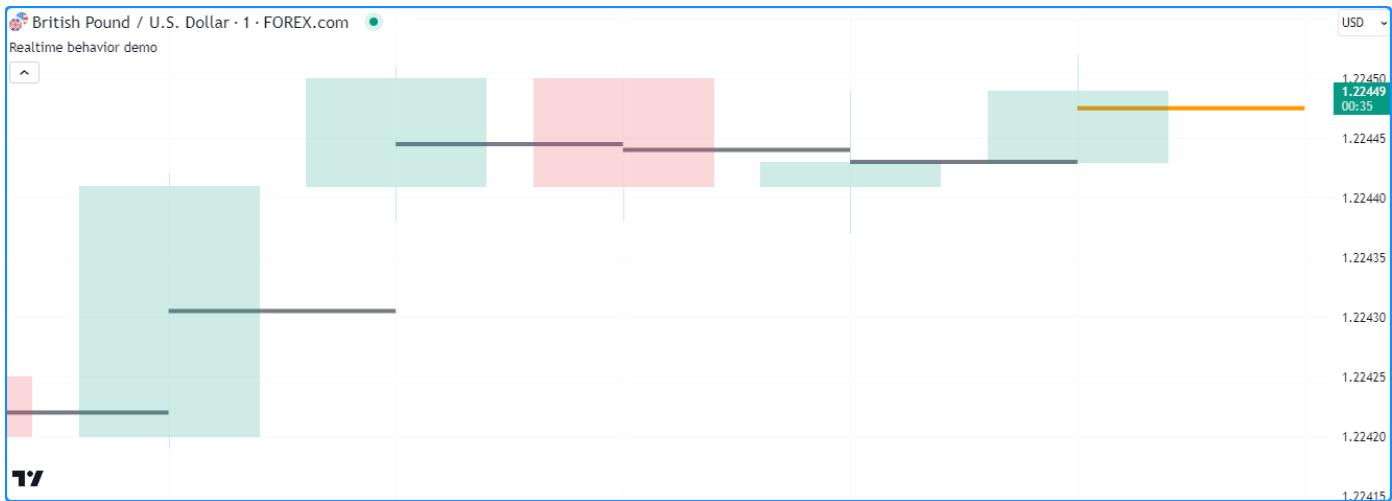
// Delete the polyline assigned to the `currentDrawing` if it's not a `newPeriod`.
if not newPeriod
    polyline.delete(currentDrawing)
// Assign a new polyline to the `currentDrawing`.
// Uses the `index` field from each `chart.point` in its array as x-coordinates.
currentDrawing := polyline.new(
    array.from(openPoint, highPoint, closePoint, lowPoint), closed = true,
    line_color = drawingColor, fill_color = color.new(drawingColor, 60)
)

```

## 实时行为

[Lines](#)、[Box](#)和[Polylines](#)都会受到提交和回滚操作的影响，这会影响脚本在实时栏上执行时的行为。[请参阅有关Pine Script™执行模型的页面。](#)

此脚本演示了在实时、未确认的图表条上执行时回滚的效果：



```
//@version=5
indicator("Realtime behavior demo", overlay = true)

//@variable Is orange when the `line` is subject to rollback and gray after the `line`
//is committed.
color lineColor = barstate.isconfirmed ? color.gray : color.orange

line.new(bar_index, h12, bar_index + 1, h12, color = lineColor, width = 4)
```

当未确认柱上的值发生变化时，本示例中的 `line.new()` 调用会在每次迭代中创建一个新的 [线条ID](#)。由于每次迭代之前的回滚，脚本会自动删除在该栏中每次更改时创建的对象。它仅提交在柱关闭之前创建的最后一行，并且该 [行](#) 实例是在已确认的柱上持续存在的实例。

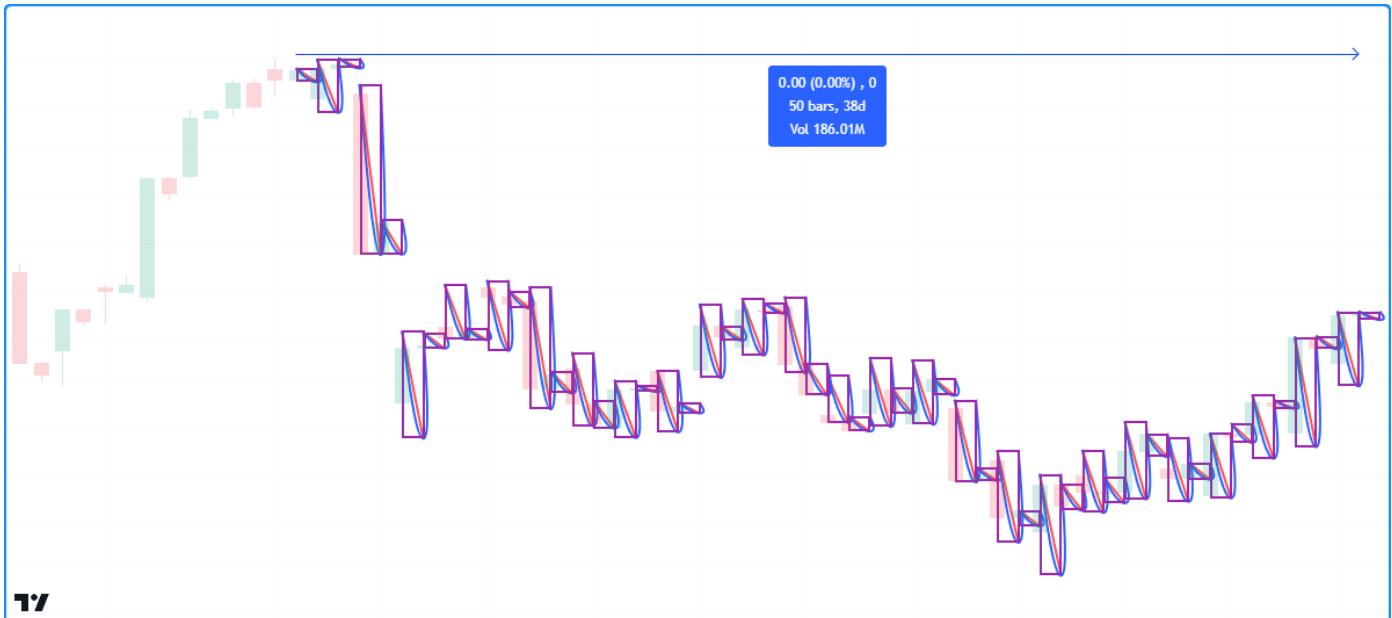
## 限制

### 对象总数

[Lines](#)、[Box](#)和[Polylines](#)会消耗服务器资源，这就是每个脚本的绘图总数受到限制的原因。当脚本创建的绘图对象多于允许的限制时，Pine Script™ 运行时会在称为 [垃圾收集](#)的过程中自动删除最旧的绘图对象。

单个脚本最多可以包含 500 条线、500 个框和 100 条折线。用户可以通过在脚本的 `indicator()`或 `strategy()` 声明语句中指定 `max_lines_count`、`max_boxes_count` 和值来控制垃圾收集限制。`max_polyline_count`

该脚本演示了 Pine 中垃圾收集的工作原理。它在每个图表栏上创建新的线条、框和折线。我们没有在[Indicator\(\)](#) `max_lines_count` 函数调用中指定、`max_boxes_count` 或 `max_polyline_count` 参数的值，因此脚本将维护图表上最近的约 50 条线、框和折线，因为这是每个参数的默认设置：



```
//@version=5
indicator("Garbage collection demo", overlay = true)

//@variable A new `chart.point` at the current `bar_index` and `high`.
firstPoint = chart.point.now(high)
//@variable A new `chart.point` one bar into the future at the current `low`.
secondPoint = chart.point.from_index(bar_index + 1, low)
//@variable A new `chart.point` one bar into the future at the current `high`.
thirdPoint = chart.point.from_index(bar_index + 1, high)

// Draw a new `line` connecting the `firstPoint` to the `secondPoint`.
line.new(firstPoint, secondPoint, color = color.red, width = 2)
// Draw a new `box` with the `firstPoint` top-left corner and `secondPoint` bottom-right corner.
box.new(firstPoint, secondPoint, color.purple, 2, bgcolor = na)
// Draw a new `polyline` connecting the `firstPoint`, `secondPoint`, and `thirdPoint` sequentially.
polyline.new(array.from(firstPoint, secondPoint, thirdPoint), true, line_width = 2)
```

- 注意:

我们使用 TradingView 的“Measure”绘图工具来测量脚本绘图对象覆盖的柱形数量。

## 未来对 xloc.bar\_index 的引用

使用xloc.bar\_index定位的对象 可以包含未来不超过 500 个柱的 x 坐标。

## 其他上下文

脚本不能在函数中使用`lines`、`boxes`或多段`request.*()`线。这些类型的实例可以使用调用中的值`request.*()`，但脚本只能在图表的上下文中创建和绘制它们。

`timeframe` 此限制也是为什么在 `indicator()` 声明语句中使用参数时绘图对象无法工作的原因。

## 历史缓冲区和 `max_bars_back`

将`barstate.isrealtime` 与绘图结合使用有时可能会产生意想不到的结果。例如，此脚本的目的是忽略所有历史柱并在实时柱上绘制跨越 300 个柱的水平线：

```
//@version=5
indicator("Historical buffer demo", overlay = true)

//@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close`.
firstPoint = chart.point.from_index(bar_index[300], close)
//@variable The current bar's `chart.point` containing the current `close`.
secondPoint = chart.point.now(close)

// Draw a new line on realtime bars.
if barstate.isrealtime
    line.new(firstPoint, secondPoint)
```

但是，它会在运行时失败并引发错误。该脚本失败，因为它无法确定基础`时间`序列历史值的缓冲区大小。尽管代码不包含内置`时间`变量，但内置`bar_index`在其内部工作中使用`时间`序列。因此，要访问300个柱之前的历史`bar_index`值，需要`时间`序列的历史缓冲区至少为300个柱。

Pine Script™ 包含一种在大多数情况下自动检测所需历史缓冲区大小的机制。它的工作原理是让脚本在有限的时间内访问任意数量的柱的历史值。在此脚本的情况下，使用`barstate.isrealtime`控制线条的绘制会阻止其访问历史系列，因此无法推断所需的历史缓冲区大小，并且脚本失败。

解决此问题的简单方法是在评估`条件结构`之前使用`max_bars_back()` 函数显式定义`时间`序列的历史缓冲区：

```
//@version=5
indicator("Historical buffer demo", overlay = true)

//@variable A `chart.point` at the `bar_index` from 300 bars ago and current `close`.
firstPoint = chart.point.from_index(bar_index[300], close)
//@variable The current bar's `chart.point` containing the current `close`.
secondPoint = chart.point.now(close)

// Explicitly set the historical buffer of the `time` series to 300 bars.
max_bars_back(time, 300)

// Draw a new line on realtime bars.
if barstate.isrealtime
    line.new(firstPoint, secondPoint)
```

此类问题可能会令人困惑，但非常罕见。 Pine Script™ 团队希望随着时间的推移消除它们。

# 非标准图表数据

## 介绍

这些函数允许脚本从非标准条形图或图表类型获取信息，无论脚本运行的图表类型如何。它们是：[ticker.heikinashi\(\)](#)、[ticker.renko\(\)](#)、[ticker.linebreak\(\)](#)、[ticker.kagi\(\)](#)和[ticker.pointfigure\(\)](#)。它们都以相同的方式工作；他们创建一个特殊的代码标识符，用作[request.security\(\)](#)函数调用中的第一个参数。

### [ticker.heikinashi\(\)](#)

*Heikin-Ashi*在日语中的意思是“平均酒吧”。 Heikin-Ashi 蜡烛图的开盘价/最高价/最低价/收盘价是合成的；它们不是实际的市场价格。它们是通过对当前柱和前一柱的实际 OHLC 值的组合进行平均来计算的。使用的计算使 Heikin-Ashi 柱形图比普通烛形图的噪音要小。它们对于进行视觉评估很有用，但不适合回溯测试或自动交易，因为订单按市场价格执行，而不是平均足价格。

`ticker.heikinashi()` 函数创建一个特殊的代码标识符，用于使用[request.security\(\)](#)函数请求 Heikin-Ashi 数据。

此脚本请求 Heikin-Ashi 柱的收盘值并将它们绘制在正常蜡烛图的顶部：

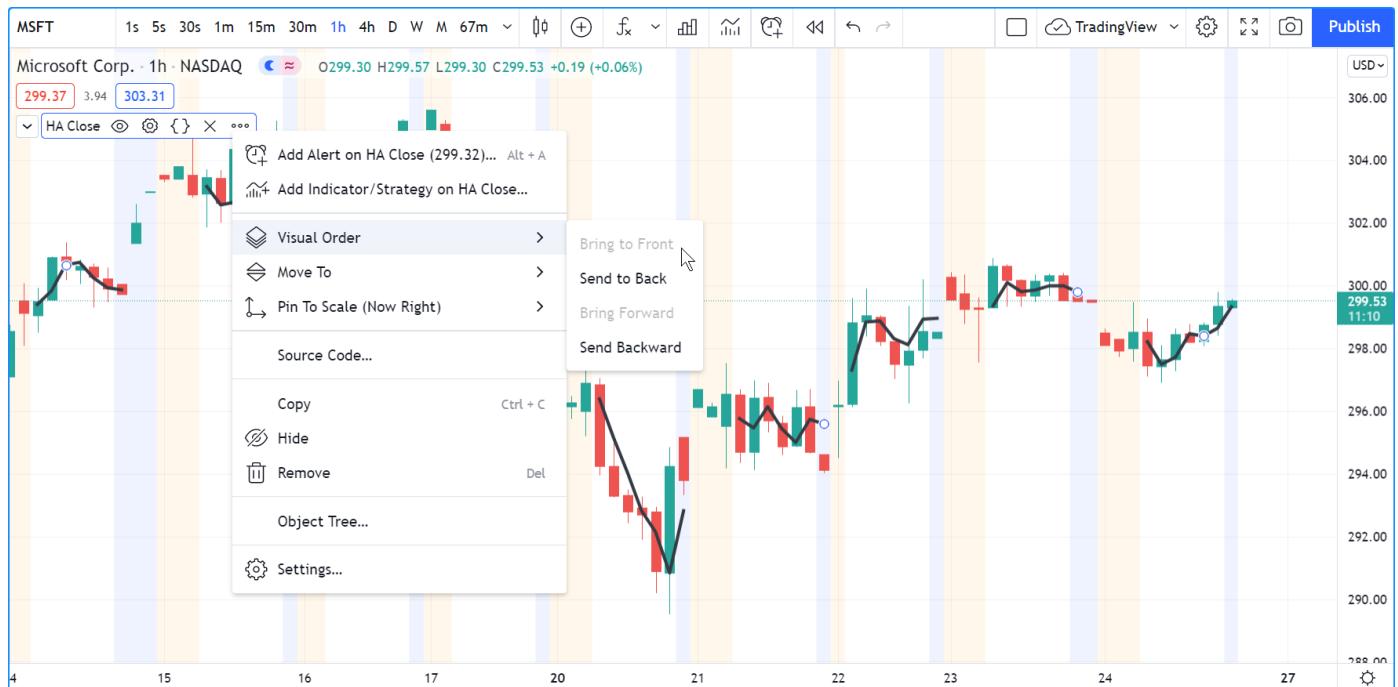


```
//@version=5
indicator("HA Close", "", true)
haTicker = ticker.heikinashi(syminfo.tickerid)
haClose = request.security(haTicker, timeframe.period, close)
plot(haClose, "HA Close", color.black, 3)
```

注意：

- 绘制为黑线的 Heikin-Ashi 柱的收盘价与使用市场价格的真实蜡烛的收盘价非常不同。它们的表现更像是移动平均线。
- 黑线出现在图表条上，因为我们从脚本的“更多”菜单中选择了“视觉顺序/置于前面”。

如果您想在上一个示例中省略延长时间的值，则需要首先创建一个没有延长会话信息的中间代码：

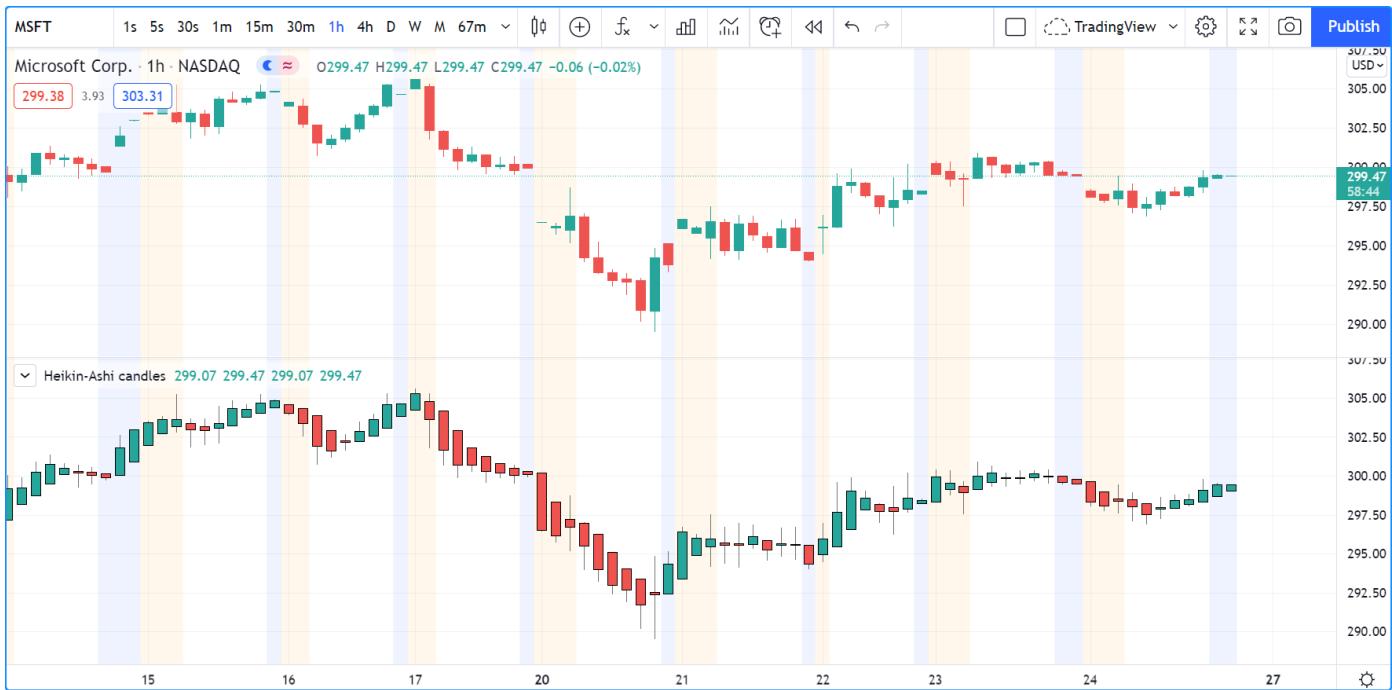


```
//@version=5
indicator("HA Close", "", true)
regularSessionTicker = ticker.new(syminfo.prefix, syminfo.ticker, session.regular)
haTicker = ticker.heikinashi(regularSessionTicker)
haClose = request.security(haTicker, timeframe.period, close, gaps = barmerge.gaps_on)
plot(haClose, "HA Close", color.black, 3, plot.style_linebr)
```

注意：

- 我们首先使用 `ticker.new()` 函数来创建一个没有扩展会话信息的股票代码。
- 我们在 `ticker.heikinashi()` 调用中使用该股票而不是 `syminfo.tickerid`。
- 在 `request.security()` 调用中，我们将 `gaps` 参数的值设置为 `barmerge.gaps_on`。这指示函数不要使用以前的值来填充缺少数据的槽。这使得它可以在常规会话之外返回 `na` 值。
- 为了能够在图表上看到这一点，我们还需要使用特殊的样式，它会破坏 `plot.style_linebr` 值上的绘图。

此脚本在图表下绘制 Heikin-Ashi 蜡烛图：



```
//@version=5
indicator("Heikin-Ashi candles")
CANDLE_GREEN = #26A69A
CANDLE_RED = #EF5350

haTicker = ticker.heikinashi(syminfo.tickerid)
[haO, haH, haL, haC] = request.security(haTicker, timeframe.period, [open, high, low, close])
candleColor = haC >= haO ? CANDLE_GREEN : CANDLE_RED
plotcandle(haO, haH, haL, haC, color = candleColor)
```

注意：

- 我们使用带有 [request.security\(\)](#) 的元组 通过同一调用获取四个值。
- 我们使用 [plotcandle\(\)](#) 来绘制蜡烛图。有关详细信息，请参阅[条形图绘制页面](#)。

## [ticker.renko\(\)](#)

砖形图仅绘制价格变动，而不考虑时间或交易量。它们看起来就像堆叠在相邻列中的砖块[1]。仅当价格超过顶部或底部预定金额后才会绘制新砖。 `ticker.renko()` 函数创建一个股票代码 id，可与 [request.security\(\)](#) 一起使用 来获取砖形图值，但没有 Pine Script™ 函数可以在图表上绘制砖形图：

```
//@version=5
indicator("", "", true)
renkoTicker = ticker.renko(syminfo.tickerid, "ATR", 10)
renkoLow = request.security(renkoTicker, timeframe.period, low)
plot(renkoLow)
```

## [ticker.linebreak\(\)](#)

折线图表类型显示一系列基于价格变化的垂直框[1]。ticker.linebreak() 函数创建一个股票代码 id，可与 [request.security\(\)](#)一起使用 来获取“换行符”值，但没有 Pine Script™ 函数可以在图表上绘制此类条形：

```
//@version=5
indicator("", "", true)
lineBreakTicker = ticker.linebreak(syminfo.tickerid, 3)
lineBreakClose = request.security(lineBreakTicker, timeframe.period, close)
plot(lineBreakClose)
```

## [ticker.kagi\(\)](#)

卡吉图由改变方向的连续线组成。当价格变化[1] 超出预定金额时，方向就会发生变化。ticker.kagi() 函数创建一个股票代码 id，可与 [request.security\(\)](#)一起使用 来获取“Kagi”值，但没有 Pine Script™ 函数可以在图表上绘制此类条形：

```
//@version=5
indicator("", "", true)
kagiBreakTicker = ticker.linebreak(syminfo.tickerid, 3)
kagiBreakClose = request.security(kagiBreakTicker, timeframe.period, close)
plot(kagiBreakClose)
```

## [ticker.pointfigure\(\)](#)

点数图(PnF) 图表仅绘制价格变动[1]，而不考虑时间。当价格上涨时绘制 X 列，当价格下跌时绘制 O 列。ticker.pointfigure() 函数创建一个股票代码 id，可与 [request.security\(\)](#)一起使用 来获取“PnF”值，但没有 Pine Script™ 函数可以在图表上绘制此类条形。每列 X 或 O 都由四个数字表示。您可以将它们视为合成的 OHLC PnF 值：

```
//@version=5
indicator("", "", true)
pnfTicker = ticker.pointfigure(syminfo.tickerid, "hl", "ATR", 14, 3)
[pnfO, pnfC] = request.security(pnfTicker, timeframe.period, [open, close],
barmerge.gaps_on)
plot(pnfO, "PnF Open", color.green, 4, plot.style_linebr)
plot(pnfC, "PnF Close", color.red, 4, plot.style_linebr)
```

## 脚注

[1] ([1](#), [2](#), [3](#), [4](#))在 TradingView 上，Renko、Line Break、Kagi 和 PnF 图表类型是根据较低时间范围内的 OHLC 值生成的。因此，这些图表类型仅代表从报价数据生成时的近似值。

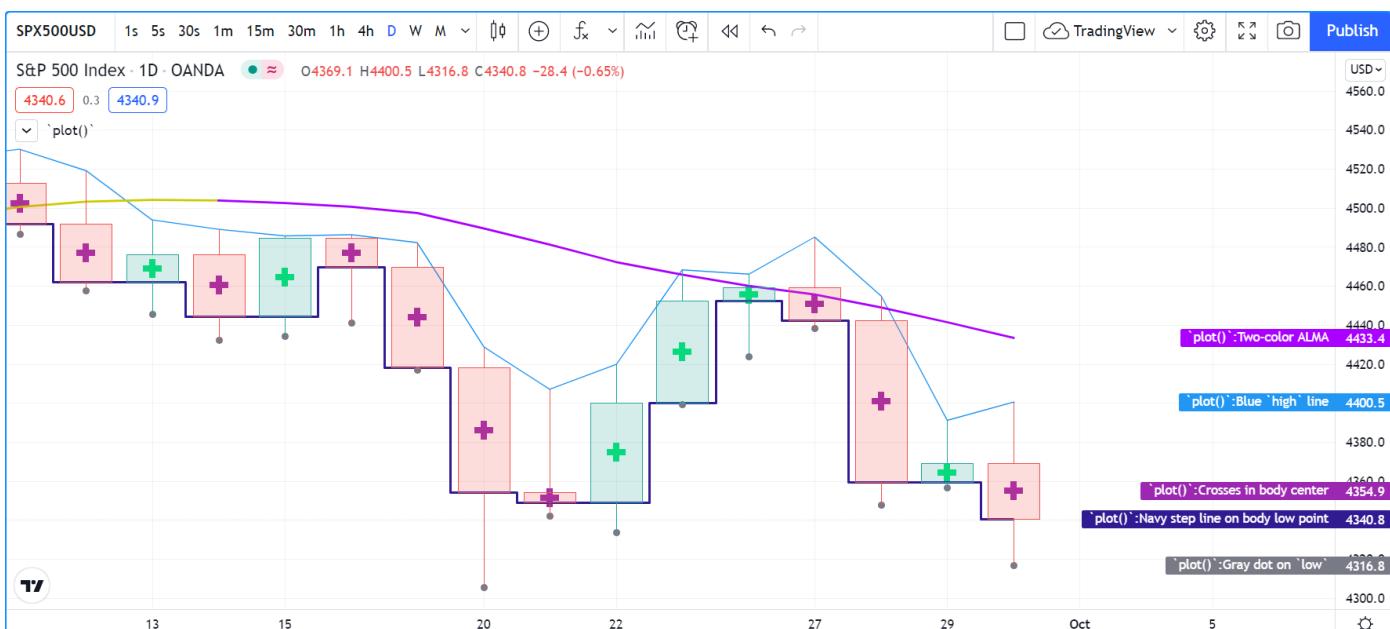
# 绘图

# 介绍

`plot()` 函数是最常用的函数，用于显示使用 Pine 脚本计算的信息。它用途广泛，可以绘制不同样式的线条、直方图、面积、柱形（如体积柱）、填充、圆形或十字形。

[Fills](#) 页面中解释了如何使用 `plot()` 创建填充。

该脚本展示了在覆盖脚本中 `plot()` 的几种不同用法：



```
//@version=5
indicator(`plot()`, "", true)
plot(high, "Blue `high` line")
plot(math.avg(close, open), "Crosses in body center", close > open ? color.lime :
color.purple, 6, plot.style_cross)
plot(math.min(open, close), "Navy step line on body low point", color.navy, 3,
plot.style_stepline)
plot(low, "Gray dot on `low`", color.gray, 3, plot.style_circles)

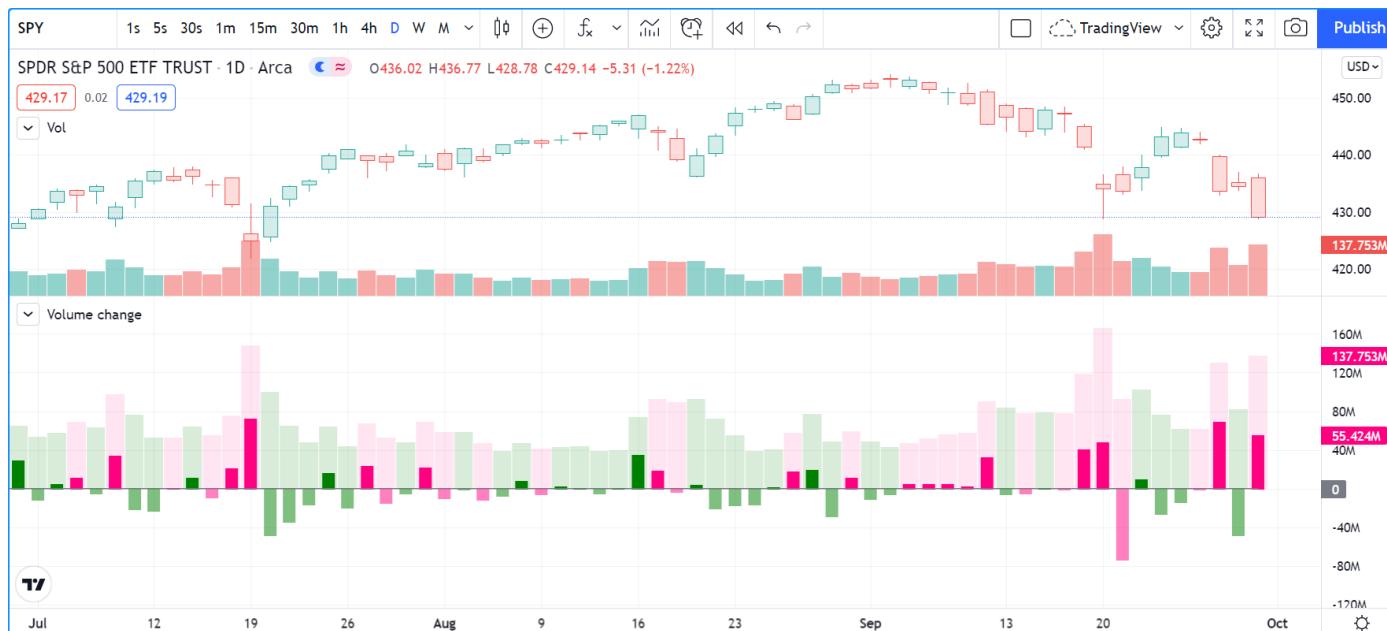
color VIOLET = #AA00FF
color GOLD   = #CCCC00
ma = ta.alma(hl2, 40, 0.85, 6)
var almaColor = color.silver
almaColor := ma > ma[2] ? GOLD : ma < ma[2] ? VIOLET : almaColor
plot(ma, "Two-color ALMA", almaColor, 2)
```

注意：

- 第一个 `plot()` 调用绘制了一条穿过条形高点的 1 像素蓝线。
- 第二个图在实体的中点交叉。当条形向上时，十字架呈石灰色；当条形向下时，十字架呈紫色。用于的参数 `linewidth` 是 6 但它不是像素值；只是相对大小。
- 第三次调用绘制了一条跟随物体低点的 3 像素宽的阶梯线。
- 第四次调用在柱的最低点绘制了一个灰色圆圈。

- 最后一个情节需要一些准备。我们首先定义牛市/熊市颜色，计算[Arnaud Legoux 移动平均线](#)，然后进行颜色计算。我们使用`var`仅在零柱上初始化颜色变量。我们将其初始化为`color.silver`，因此在数据集的第一个条上，直到我们的一个条件导致颜色发生变化，该线将为银色。改变线条颜色的条件要求它高于/低于两根柱前的值。与我们仅仅寻找比前一个值更高/更低的值相比，这使得颜色过渡的噪音更少。

此脚本在窗格中显示了[plot\(\)](#)的其他用法：



```
//@version=5
indicator("Volume change", format = format.volume)

color GREEN      = #008000
color GREEN_LIGHT = color.new(GREEN, 50)
color GREEN_LIGHTER = color.new(GREEN, 85)
color PINK       = #FF0080
color PINK_LIGHT = color.new(PINK, 50)
color PINK_LIGHTER = color.new(PINK, 90)

bool barUp = ta.rising(close, 1)
bool barDn = ta.falling(close, 1)
float volumeChange = ta.change(volume)

volumeColor = barUp ? GREEN_LIGHTER : barDn ? PINK_LIGHTER : color.gray
plot(volume, "Volume columns", volumeColor, style = plot.style_columns)

volumeChangeColor = barUp ? volumeChange > 0 ? GREEN : GREEN_LIGHT : volumeChange > 0 ?
PINK : PINK_LIGHT
plot(volumeChange, "Volume change columns", volumeChangeColor, 12,
plot.style_histogram)

plot(0, "Zero line", color.gray)
```

注意：

- 我们将正常体积值绘制为零线上方的宽柱（请参阅我们的[plot\(\)](#)调用中的）。`style = plot.style_columns`
- 在绘制列之前，我们使用和布尔变量 `volumeColor` 的值来计算。当当前柱的收盘价 高于/低于前一柱时，它们分别变为。请注意，内置的“Volume”不使用相同的条件；它用 标识一个向上的柱。我们对体积列使用和颜色。`barUp``barDn``true``close > open``GREEN_LIGHTER``PINK_LIGHTER`
- 因为第一个图绘制了列，所以我们不使用该 `linewidth` 参数，因为它对列没有影响。
- 我们脚本的第二个图是体积的变化，我们之前使用 计算过 `ta.change(volume)`。该值绘制为直方图，其中 `linewidth` 参数控制列的宽度。我们设置这个宽度 `12`，使直方图元素比第一个图的列更细。正/负值 `volumeChange` 绘制在零线上方/下方；无需任何操作即可实现此效果。
- 在绘制值的直方图之前 `volumeChange`，我们计算其颜色值，它可以是四种不同颜色之一。当条向上/向下并且成交量自最后一个条 ()以来增加时，我们使用明亮 `GREEN` 或颜色。因为在本例中为正，所以直方图的元素将绘制在零线上方。当条向上/向下并且自上一个条以来成交量没有增加时，我们使用明亮或颜色。因为在这种情况下为负，所以直方图的元素将绘制在零线下方。`PINK``volumeChange > 0``volumeChange``GREEN_LIGHT``PINK_LIGHT``volumeChange`
- 最后，我们绘制一条零线。我们也可以 `hline(0)` 在那里使用。
- 我们在[indicator\(\)](#)调用中使用，以便该脚本显示的大值被缩写，就像内置的“交易量”指标一样。`format = format.volume`

[plot\(\)](#) 调用必须始终放置在行的第一个位置，这意味着它们始终位于脚本的全局范围内。它们不能放置在用户定义的函数或结构中，如[if](#)、[for](#)等。但是，对[plot\(\)](#) 的调用可以设计为以两种方式进行条件绘图，我们将在 本页的条件绘图部分中介绍这些方法。

脚本只能在其自己的视觉空间中进行绘制，无论是在窗格中还是在图表上作为叠加层。在窗格中运行的脚本只能在图表区域中显示[颜色条](#)。

## plot() 参数

[plot\(\)](#) 函数具有以下签名：

```
plot(series, title, color, linewidth, style, trackprice, histbase, offset, join,
editable, show_last, display) → plot
```

[绘图\(\)](#) 的参数是：

- `series`

它是唯一的强制参数。它的参数必须是“series int/float”类型。请注意，由于 Pine Script™ 中的自动转换规则以 `int`  $\square$  `float`  $\square$  `bool` 方向进行转换，因此“`bool`”类型变量不能按原样使用；它必须转换为“`int`”或“`float`”才能用作参数。例如，`if newDay` 是“`bool`”类型，则当变量为 `1` 时，可用于绘制 `1`，当变量为 `0` 时，可用于绘制 `0`。`newDay ? 1 : 0``true``false`

- `title`

需要一个“`const string`”参数，因此它必须在编译时已知。出现字符串：当选中“图表设置/刻度/指标名称标签”字段时，在脚本的刻度中。在数据窗口中。在“设置/样式”选项卡中。[在input.source\(\)](#) 字段的下拉列表中。当选择脚本时，在“创建警报”对话框的“条件”字段中。作为将图表数据导出到 CSV 文件时的列标题。

- `color`

接受“系列颜色”，因此可以逐条即时计算。使用`na`作为颜色或透明度为 100 的任何颜色进行绘图是在不需要绘图时隐藏绘图的一种方法。

- `linewidth`

是绘制元素的大小，但它并不适用于所有样式。绘制线条时，单位是像素。使用`plot.style_columns`时没有影响。

- `style`

可用的参数有：`plot.style_line`（默认值）：它使用`linewidth`以像素为单位的参数绘制一条连续线的宽度。`na`值不会绘制为一条线，但当非`na`值出现时，它们将被桥接。非`na`值仅当它们在图表上可见时才被桥接。`plot.style_linebr`：允许通过不在`na`值上绘制、不连接间隙（即桥接`na`值）来绘制不连续线。`plot.style_stepline`：使用楼梯效果进行绘图。值变化之间的转换是使用在条形中间绘制的垂直线来完成的，而不是使用连接条形中点的点对点对角线。也可用于实现与`plot.style_linebr`类似的效果，但前提是注意在`na`值上不绘制颜色。`plot.style_area`：绘制一条`linewidth`宽度的线，填充线和之间的区域`histbase`。该`color`参数用于线条和填充。您可以使用另一个`plot()`调用使线条具有不同的颜色。正值绘制在上方`histbase`，负值绘制在下方。`plot.style_areabr`：这与`plot.style_area`类似，但它不会跨越`na`值。另一个区别是指标规模的计算方式。只有绘制的值才用于计算脚本视觉空间的y范围。例如，如果仅绘制远离的高值`histbase`，则这些值将用于计算脚本视觉空间的y比例。正值绘制在上方`histbase`，负值绘制在下方。`plot.style_columns`：绘制类似于“Volume”内置指标的列。该`linewidth`值不影响列的宽度。正值绘制在上方`histbase`，负值绘制在下方。始终`histbase`在脚本视觉空间的y比例中包含的值。`plot.style_histogram`：绘制与“Volume”内置指标类似的列，不同之处在于该`linewidth`值用于确定直方图条形的宽度（以像素为单位）。请注意，由于`linewidth`需要“输入 int”值，因此直方图条形的宽度不能随条形变化。正值绘制在上方`histbase`，负值绘制在下方。始终`histbase`在脚本视觉空间的y比例中包含的值。`plot.style_circles`和`plot.style_cross`：这些绘制的形状不会跨条连接，除非也使用了。对于这些样式，参数成为相对大小度量 - 它的单位不是像素。`join = true`` linewidth`

- `trackprice`

该值的默认值为`false`。当它是`true`时，将在脚本视觉空间的整个宽度上绘制一条由小方块组成的虚线。它经常与`histbase`结合使用，以隐藏实际情节，只留下残留的虚线。`show_last = 1, offset = -99999`

- `histbase`

它是与`plot.style_area`、`plot.style_columns`和`plot.style_histogram`一起使用的参考点。它决定了论证的正值和负值的分隔程度`series`。它不能动态计算，因为需要“输入 int/float”。

- `offset`

这允许使用柱中的负/正偏移来移动过去/未来的图。该值在脚本执行期间不能更改。

- `join`

这将影响样式`plot.style_circles`或`plot.style_cross`。当时`true`，形状通过单像素线连接。

- `editable`

该布尔参数控制是否可以在“设置/样式”选项卡中编辑绘图的属性。它的默认值为`true`。

- `show_last`

允许控制最后有多少条绘制值可见。需要一个“input int”参数，因此无法动态计算。

- `display`

默认为`display.all`。当它设置为`display.none`时，绘制的值不会影响脚本视觉空间的比例。该图将不可见，并且不会出现在指标值或数据窗口中。它在用作其他脚本的外部输入的绘图中很有用，或者`plot(" [plot_title]")`在`alertcondition()`调用中与占位符一起使用的绘图中很有用，例如：

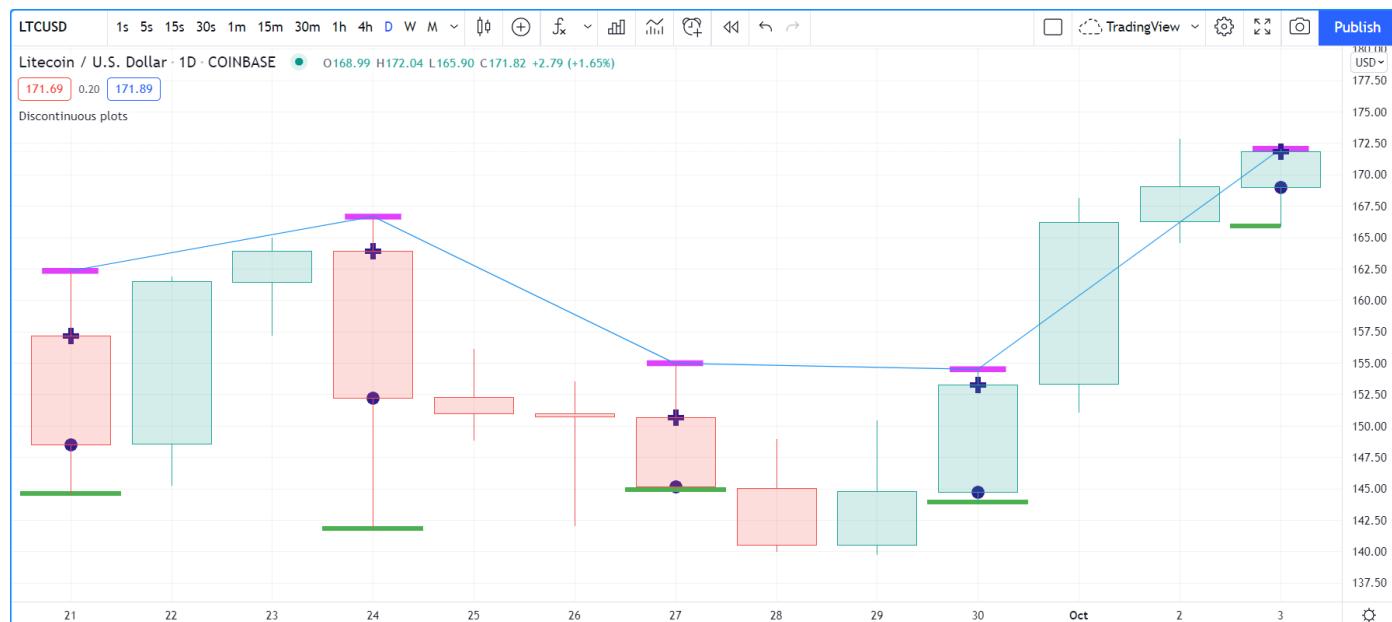
```
//@version=5
indicator("")
r = ta.rsi(close, 14)
xUp = ta.crossover(r, 50)
plot(r, "RSI", display = display.none)
alertcondition(xUp, "xUp alert", message = 'RSI is bullish at: {{plot("RSI")}}')
```

## 有条件地绘图

`plot()`调用不能在条件结构（例如`if`）中使用，但可以通过改变其绘制值或颜色来控制它们。当不需要绘图时，您可以绘制`na`值，或使用`na`颜色或透明度为 100 的任何颜色（这也使其不可见）绘制值。

## 值控制

控制图显示的一种方法是在不需要图时绘制`na`值。有时，`request.security()`等函数返回的值将在使用时返回`na`值。在这两种情况下，绘制不连续线有时很有用。该脚本展示了几种执行此操作的方法：`gaps = barmerge.gaps_on`



```
//@version=5
indicator("Discontinuous plots", "", true)
bool plotValues = bar_index % 3 == 0
plot(plotValues ? high : na, color = color.fuchsia, linewidth = 6, style =
plot.style_linebr)
plot(plotValues ? high : na)
plot(plotValues ? math.max(open, close) : na, color = color.navy, linewidth = 6, style =
= plot.style_cross)
plot(plotValues ? math.min(open, close) : na, color = color.navy, linewidth = 6, style =
= plot.style_circles)
plot(plotValues ? low : na, color = plotValues ? color.green : na, linewidth = 6, style =
= plot.style_stepline)
```

注意：

- 我们使用 `bar_index % 3 == 0` 来定义确定绘图时的条件，即当条形索引除以 3 的余数为零时。每三小节就会发生一次这种情况。
- 在第一个图中，我们使用 `plot.style_linebr`，它在高点上绘制紫红色线。它以条形的水平中点为中心。
- 第二个图显示了绘制相同值的结果，但没有特别小心地打破线条。这里发生的情况是，普通 `plot()` 调用的细蓝线会自动桥接 `na` 值（或间隙），因此绘图不会中断。
- 然后，我们在身体的顶部和底部绘制海军蓝色的十字和圆圈。`plot.style_circles` 和 `plot.style_cross` 样式是绘制不连续值的简单方法，例如，止损或止盈水平，或支撑和阻力水平。
- 条形最低点上的最后一个绿色图是使用 `plot.style_stepline` 完成的。请注意它的线段比用 `plot.style_linebr` 绘制的紫红色线段更宽。另请注意，在最后一个柱上，它仅绘制到下一个柱出现之前的一半。
- 每个情节的绘制顺序由它们在脚本中出现的顺序控制。看

此脚本展示了如何将绘图限制为用户定义日期之后的条形图。我们使用 `input.time()` 函数创建一个输入小部件，允许脚本用户选择日期和时间，并使用 2021 年 1 月 1 日作为其默认值：

```
//@version=5
indicator("", "", true)
startInput = input.time(timestamp("2021-01-01"))
plot(time > startInput ? close : na)
```

## 颜色控制

颜色页面的条件着色部分讨论了绘图的颜色控制。我们将看几个例子。

`plot(color)` 中参数的值可以是常量，例如内置常量颜色之一或 [颜色文字](#)。在 Pine Script™ 中，此类颜色的限定类型称为“常量颜色”（请参阅[类型系统](#)页面）。它们在编译时是已知的：

```
//@version=5
indicator("", "", true)
plot(close, color = color.gray)
```

也可以使用仅当脚本开始在图表的第一个历史柱（柱零，即或）上执行时才知道的信息来确定绘图的颜色，就像确定颜色所需的信息时的情况一样取决于脚本运行的图表。在这里，我们使用 `syminfo.type` 内置变量计算绘图颜色，该变量返回图表符号的类型。在这种情况下，合格的类型将是“**simple color**”：`bar_index == 0` `barstate.isfirst == true` `plotColor`

```
//@version=5
indicator("", "", true)
plotColor = switch syminfo.type
    "stock"      => color.purple
    "futures"    => color.red
    "index"      => color.gray
    "forex"      => color.fuchsia
    "crypto"     => color.lime
    "fund"       => color.orange
    "dr"         => color.aqua
    "cfd"        => color.blue
plot(close, color = plotColor)
printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t, 0, 0, txt, bgcolor = color.yellow)
printTable(syminfo.type)
```

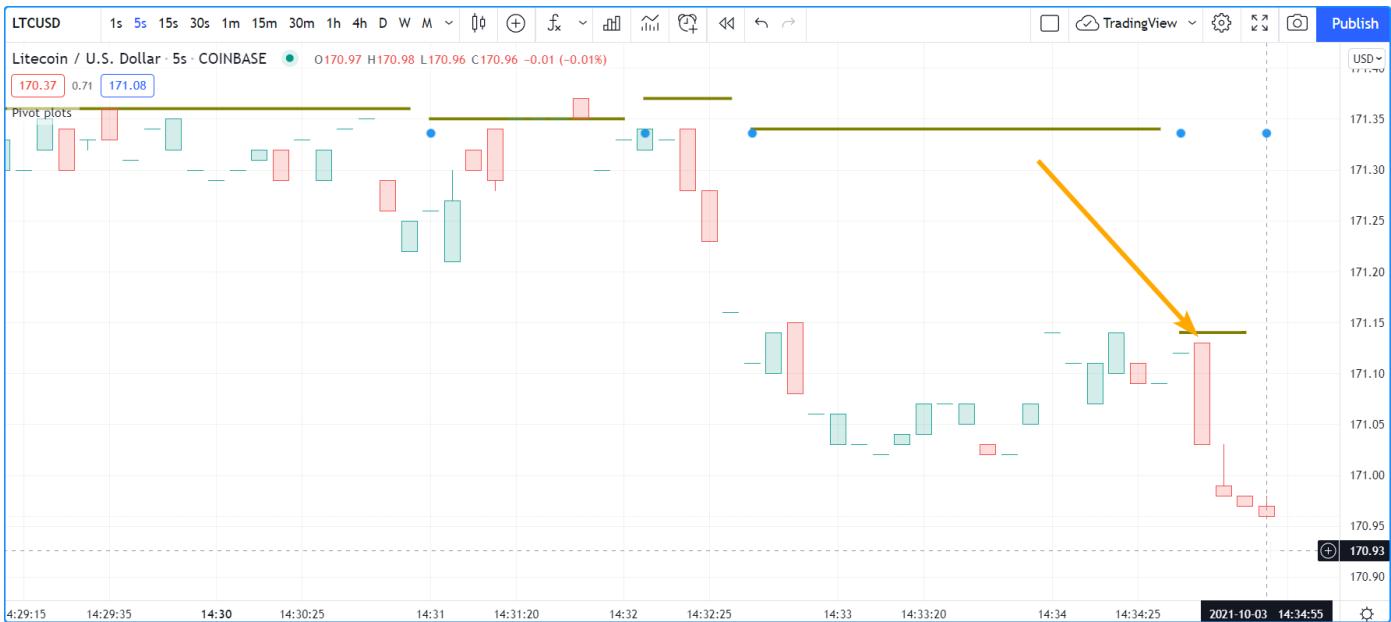
绘图颜色也可以通过脚本的输入来选择。在本例中，`lineColorInput` 变量是“**输入颜色**”类型：

```
//@version=5
indicator("", "", true)
color lineColorInput = input(#1848CC, "Line color")
plot(close, color = lineColorInput)
```

最后，绘图颜色也可以是动态值，即可以在每个条上改变的计算值。这些值属于“**系列颜色**”类型：

```
//@version=5
indicator("", "", true)
plotColor = close >= open ? color.lime : color.red
plot(close, color = plotColor)
```

绘制枢轴水平时，一项常见要求是避免绘制水平转换。使用 [线条](#) 是一种替代方法，但您也可以使用 [plot\(\)](#)，如下所示：



```
//@version=5
indicator("Pivot plots", "", true)
pivotHigh = fixnan(ta.pivothigh(3,3))
plot(pivotHigh, "High pivot", ta.change(pivotHigh) ? na : color.olive, 3)
plotchar(ta.change(pivotHigh), "ta.change(pivotHigh)", "•", location.top, size =
size.small)
```

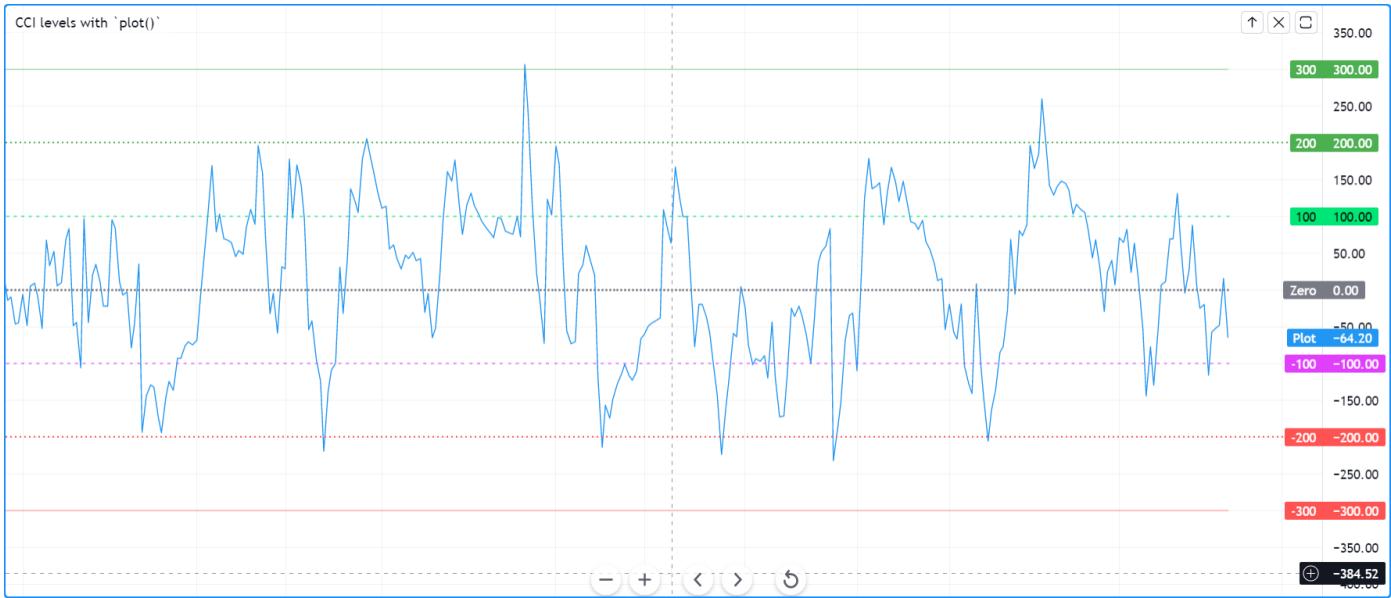
注意：

- 我们习惯于保持我们的核心价值观。因为`ta.pivothigh()`仅在找到新的主元时返回一个值，所以我们使用`fixnan()`用最后返回的主元值来填充空白。这里的间隙指的是当没有找到新的主元时`ta.pivothigh()`返回的`na`值。`pivotHigh = fixnan(ta.pivothigh(3,3))`
- 我们的枢轴在出现后三个柱被检测到，因为我们在`ta.pivothigh(3)`调用中使用了参数。`leftbars` `rightbars`
- 最后一个图绘制了一个连续值，但当枢轴的值发生变化时，它将图的颜色设置为`na`，因此此时图不可见。因此，可见的图只会出现在我们使用`na`颜色绘制的图之后的条形图上。
- 蓝点表示何时检测到新的高枢轴点，并且在前一柱和该柱之间没有绘制任何图。请注意如何在实时条形图上检测到箭头指示的条形图上的枢轴（三个条形图之后），以及如何未绘制任何图。该图仅出现在下一个柱上，使得该图在实际枢轴之后的四个柱上可见。

## 级别

Pine Script™ 有一个`hline()` 函数来绘制水平线（请参阅“[级别](#)”页面）。`hline()` 很有用，因为它有一些`plot()`无法提供的线条样式，但它也有一些限制，即它不接受“系列颜色”，并且它的`price` 参数需要“输入 int/float”，因此不能在脚本执行过程中会发生变化。

您可以 通过几种不同的方式使用`plot()`绘制级别。这显示了`CCI` 指标，其水平是使用`plot()`绘制的：



```
//@version=5
indicator("CCI levels with `plot()`")
plot(ta.cci(close, 20))
plot(0, "Zero", color.gray, 1, plot.style_circles)
plot(bar_index % 2 == 0 ? 100 : na, "100", color.lime, 1, plot.style_linebr)
plot(bar_index % 2 == 0 ? -100 : na, "-100", color.fuchsia, 1, plot.style_linebr)
plot( 200, "200", color.green, 2, trackprice = true, show_last = 1, offset = -99999)
plot(-200, "-200", color.red, 2, trackprice = true, show_last = 1, offset = -99999)
plot( 300, "300", color.new(color.green, 50), 1)
plot(-300, "-300", color.new(color.red, 50), 1)
```

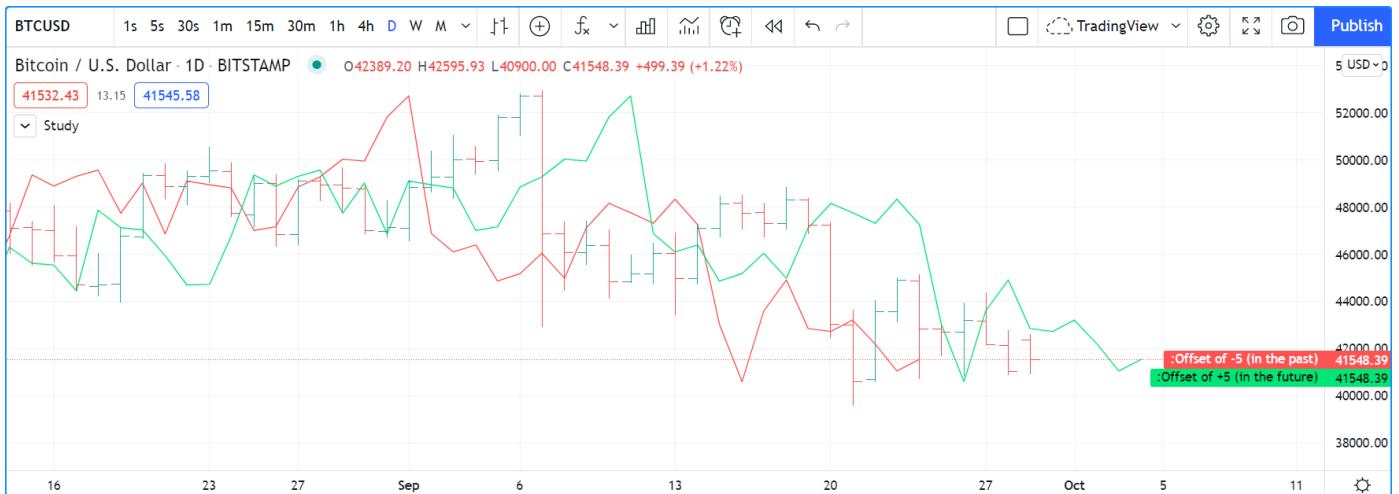
注意：

- 零水平是使用[plot.style\\_circles](#)绘制的。
- 100 个级别是使用仅每隔两个条绘制一次的条件值绘制的。为了防止`na`值被桥接，我们使用[plot.style\\_linebr](#)线条样式。
- 200 个级别的绘制用于绘制独特的小方块图案，该图案延伸了脚本视觉空间的整个宽度。那里只显示最后一个绘制的值，如果没有使用下一个技巧，该值将显示为单条直线：将单条线段推到过去很远的地方，使其永远不可见。`trackprice = true``show_last = 1``offset = -99999`
- 300 个级别使用连续线绘制，但使用较浅的透明度使它们不那么突出。

## 偏移量

该 `offset` 参数指定绘制线条时使用的偏移（负值偏移到过去，正值偏移到未来）。例如：

```
//@version=5
indicator("", "", true)
plot(close, color = color.red, offset = -5)
plot(close, color = color.lime, offset = 5)
```



从屏幕截图中可以看出，红色系列已向左移动（因为参数的值为负），而绿色系列已向右移动（其值为正）。

## 绘图计数限制

每个脚本的最大绘图数限制为 64。所有 `plot*()` 调用和 `alertcondition()` 调用都计入脚本的绘图数。某些类型的调用在总绘图计数中的数量不只一次。

如果他们使用“`const color`”参数作为参数，`plot()` 会在总绘图计数中调用 `count 1 color`，这意味着它在编译时是已知的，例如：

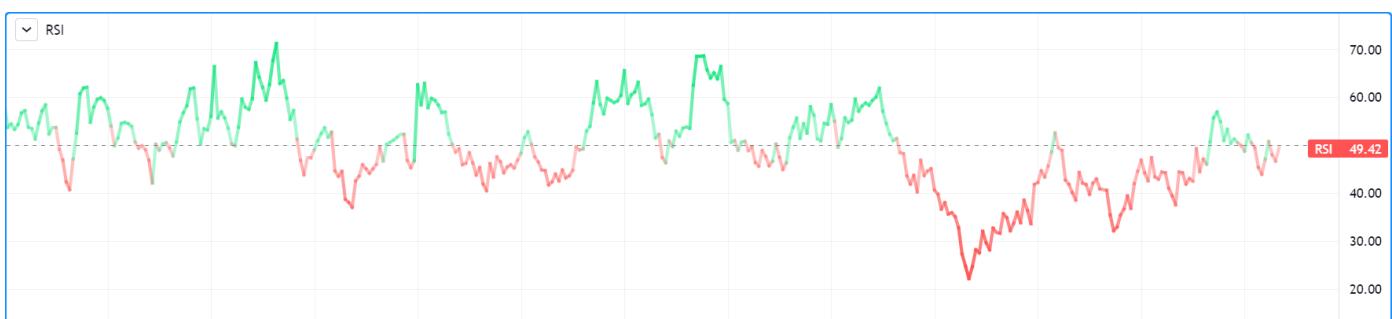
```
plot(close, color = color.green)
```

当他们使用另一种限定类型（例如其中任何一种）时，它们将在总绘图计数中计数：

```
plot(close, color = syminfo.mintick > 0.0001 ? color.green : color.red) // "simple color"
plot(close, color = input.color(color.purple)) // "input color"
plot(close, color = close > open ? color.green : color.red) // "series color"
plot(close, color = color.new(color.silver, close > open ? 40 : 0)) // "series color"
```

## 规模

并非所有值都可以随处绘制。脚本的视觉空间始终受到上限和下限的限制，上限和下限会根据绘制的值进行动态调整。RSI 指标将绘制 0 到 100 之间的值，这就是为什么它通常显示在图表上方或下方的不同窗格或区域中的原因。如果 RSI 值在图表上绘制为叠加，则效果将扭曲该符号的正常价格范围，除非它恰好接近 RSI 的 0 到 100 范围。这显示了 RSI 信号线和 50 水平的中心线，脚本在单独的窗格中运行：



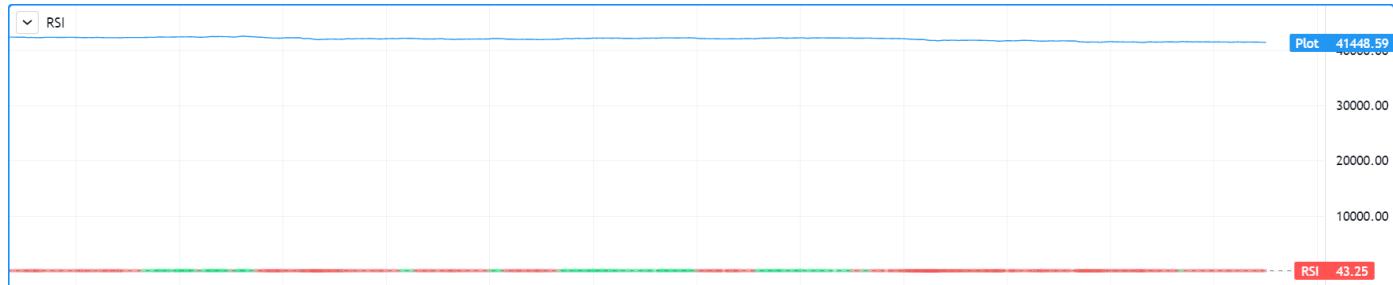
```
//@version=5
indicator("RSI")
myRSI = ta.rsi(close, 20)
bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70),
color.new(color.lime, 0))
bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0),
color.new(color.red, 70))
myRSIColor = myRSI > 50 ? bullColor : bearColor
plot(myRSI, "RSI", myRSIColor, 3)
hline(50)
```

请注意，我们的脚本视觉空间的y轴是使用绘制的值范围（即 RSI 值）自动调整大小的。有关脚本中使用的 [color.from\\_gradient\(\)](#) 函数的更多信息，请参阅[颜色](#)页面。

如果我们尝试通过将以下行添加到脚本中来在同一空间中绘制交易品种的[收盘价](#)：

```
plot(close)
```

发生的情况是这样的：

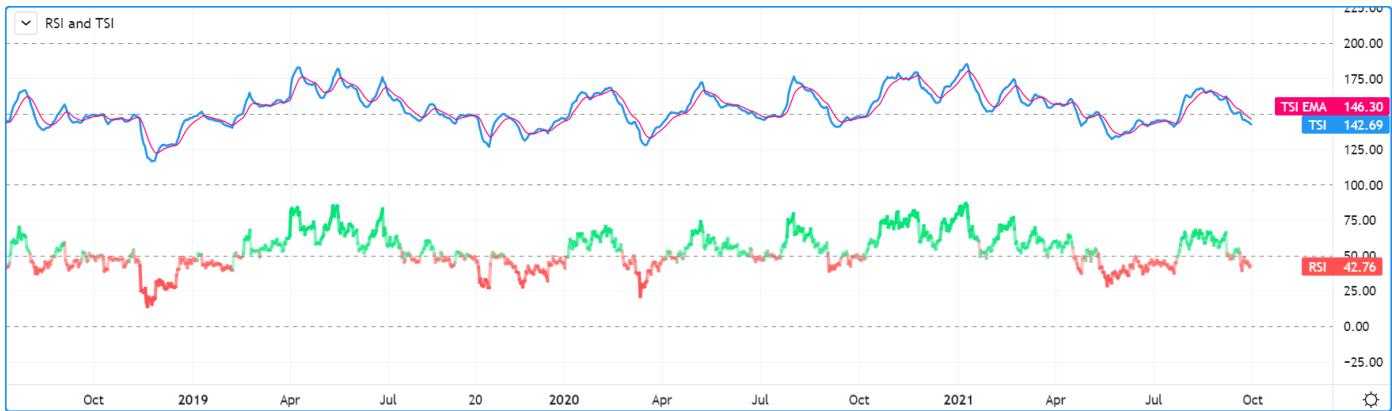


该图表显示的是 BTCUSD 符号，在此期间[收盘](#)价约为 40000 点。在 40000 范围内绘制值使得我们在 0 到 100 范围内的 RSI 图难以辨别。如果我们将[RSI](#)指标作为叠加层放置在图表上，也会出现同样的扭曲图。

## 合并两个指标

如果您计划将两个信号合并到一个脚本中，请首先考虑每个信号的规模。例如，不可能在同一脚本的视觉空间中正确绘制[RSI](#)和[MACD](#)，因为 RSI 有固定范围（0 到 100），而 MACD 则没有，因为它绘制根据价格计算的移动平均线。

如果您的两个指标都使用固定范围，您可以移动其中之一的值，使它们不重叠。例如，我们可以通过替换其中之一来绘制[RSI](#)（0 到 100）和[真实强度指标 \(TSI\)](#)（-100 到 +100）。我们的策略是压缩和移动[TSI](#)值，以便它们在[RSI](#)上绘制：



```
//@version=5
indicator("RSI and TSI")
myRSI = ta.rsi(close, 20)
bullColor = color.from_gradient(myRSI, 50, 80, color.new(color.lime, 70),
color.new(color.lime, 0))
bearColor = color.from_gradient(myRSI, 20, 50, color.new(color.red, 0),
color.new(color.red, 70))
myRSIColor = myRSI > 50 ? bullColor : bearColor
plot(myRSI, "RSI", myRSIColor, 3)
hline(100)
hline(50)
hline(0)

// 1. Compress TSI's range from -100/100 to -50/50.
// 2. Shift it higher by 150, so its -50 min value becomes 100.
myTSI = 150 + (100 * ta.tsi(close, 13, 25) / 2)
plot(myTSI, "TSI", color.blue, 2)
plot(ta.ema(myTSI, 13), "TSI EMA", #FF006E)
hline(200)
hline(150)
```

注意：

- 我们使用添加了级别 来定位两个信号。
- 为了使两条信号线在相同的 100 范围内振荡，我们将`TSI`值除以 2，因为它的范围为 200 (-100 到 +100)。然后我们将该值上移 150，使其在 100 和 200 之间振荡，使 150 成为其中心线。
- 我们在这里所做的操作是将两个具有不同尺度的指标置于同一视觉空间中所需的典型折衷，即使它们的值（与`MACD`相反）限制在固定范围内。

## 重画

## 介绍

我们将重绘定义为：导致历史与实时计算或绘图表现不同的脚本行为。

重新喷漆行为非常普遍，并且有多种因素可能导致这种行为。根据我们的定义，我们估计超过 95% 的现有指标都表现出某种形式的重绘行为。例如，MACD 和 RSI 等常用指标在历史柱上显示已确认的值，但会在实时、未经确认的图表柱上波动，直至收盘。因此，它们在历史状态和实时状态下的行为有所不同。

**并非所有重画行为本质上都是无用或具有误导性的**，此类行为也不会阻止知识渊博的交易者使用具有此类行为的指标。例如，谁会仅仅因为交易量剖面指标更新实时柱上的值而认为它不可信？

人们可能会在他们使用的脚本中遇到以下任何形式的重画，具体取决于脚本的计算所涉及的内容：

- **广泛但通常可以接受**：脚本可以使用随未确认柱上的实时价格变化而更新的值。例如，如果在打开的图表柱上执行的计算中使用[收盘](#)变量，则其值将反映柱中的最新价格。然而，一旦柱线关闭，脚本只会向其历史系列提交新的数据点。另一种常见情况是使用[request.security\(\)](#)获取实时柱上较高时间范围的数据，如[其他时间范围和数据](#)页面的[历史和实时行为](#)部分中所述。与图表时间范围内的未确认图表条一样，[request.security\(\)](#)可以跟踪实时条上较高时间范围上下文中的未确认值，这可能导致脚本重新开始执行后重新绘制。只要您了解它们的工作原理，使用此类脚本通常不会有任何问题。然而，当选择使用此类脚本来发出警报或交易订单时，重要的是要了解它们的实时行为和历史行为之间的差异，并自行决定它是否可以满足您的需求。
- **可能具有误导性**：将值绘制到过去、在实时柱上计算无法在历史柱上复制的结果或重新定位过去事件的脚本可能会产生误导。例如，Ichimoku、大多数基于枢轴的脚本、大多数使用的策略、使用[request.security\(\)](#)的脚本（当它在实时柱上表现不同时）、许多使用[varip](#)的脚本、许多使用[timenow](#)的脚本以及一些使用变量的脚本可能会出现误导性的重画行为。`calc_on_every_tick = true` `barstate.*`
- **不可接受**：将未来信息泄漏到过去的脚本、在[非标准图表](#)上执行的策略以及使用实时柱内图生成警报或订单的脚本，都是可能产生严重误导性重画行为的示例。
- **不可避免的**：来自提供商的数据馈送的修订以及图表历史记录起始栏的变化可能会导致脚本中不可避免的重画行为。

如果满足以下条件，前两种类型的重新绘制是完全可以接受的：

1. 您了解该行为。
2. 你可以忍受它，或者
3. 你可以绕过它。

现在应该清楚的是，并非所有重画行为都是错误的，需要不惜一切代价避免。在许多情况下，某些形式的重画可能正是脚本所需要的。重要的是要知道什么时候重画行为不能满足一个人的需要。为了避免不可接受的重新绘制，了解工具的工作原理或应该如何设计您构建的工具非常重要。如果您[发布](#)脚本，请确保在出版物的描述中提及任何潜在的误导行为以及脚本的其他限制。

## 笔记

我们不会讨论在非标准图表上使用策略的危险，因为这个问题与重新绘制无关。请参阅[非标准图表上的回测：小心！](#)讨论该主题的脚本。

## 对于脚本用户

如果人们了解该行为以及该行为是否满足其分析要求，则可以决定使用重绘指标。不要成为那些在已发表的剧本上“重画”句子以试图抹黑它们的新人之一，因为这样做暴露了对该主题缺乏基础知识。

考虑到脚本中存在完全可以接受的某些形式的重绘行为，简单地询问脚本是否重绘是相对没有意义的。因此，这样的问题不会产生有意义的答案。人们应该询问有关脚本潜在重画行为的具体问题，例如：

- 脚本在历史柱和实时柱上的计算/显示方式是否相同？

- 脚本中的警报是否会等待实时条结束后再触发？
- 脚本显示的信号标记是否会等待实时条的末尾才显示？
- 脚本是否将值绘制/绘制到过去？
- 策略使用吗？`calc_on_every_tick = true`
- 脚本的[request.security\(\)](#)调用是否会将未来的信息泄漏到历史柱上的过去？

重要的是您了解您使用的工具如何工作，以及它们的行为是否与您的目标兼容，无论是否重新绘制。如果您阅读本页，您将会了解到，重新绘制是一件复杂的事情。它有很多面孔和很多原因。即使您不使用 Pine Script™ 进行编程，此页面也将帮助您了解可能导致重新绘制的一系列原因，并希望能够与脚本作者进行更有意义的讨论。

## 对于 Pine Script™ 程序员

如上所述，并非所有形式的重画行为都必须不惜一切代价避免，也并非所有潜在的重画行为都一定可以避免。我们希望本页面可以帮助您更好地了解正在发生的动态，以便您可以在设计交易工具时考虑到这些行为。此页面的内容应帮助您了解会产生误导性重画结果的常见编码错误。

无论您的设计决策是什么，如果您[发布](#)脚本，请向交易者解释该脚本，以便他们了解其行为方式。

本页涵盖了重画原因的三大类：

- [历史与实时计算](#)
- [过去的情节](#)
- [数据集变化](#)

## 历史与实时计算

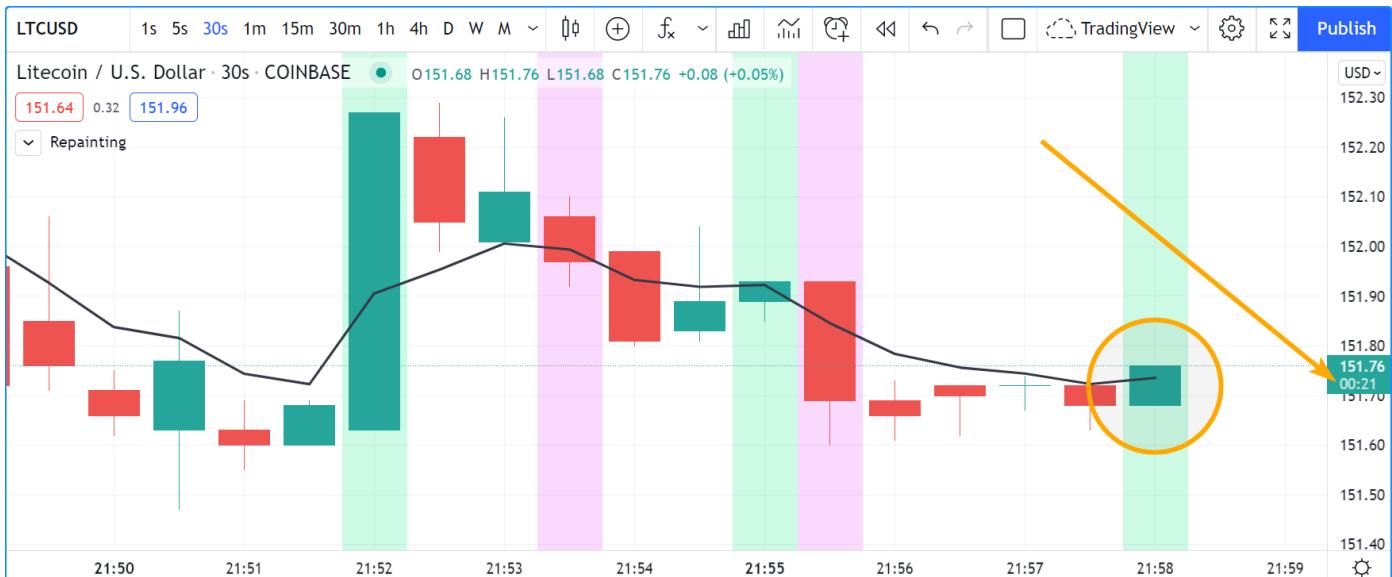
### 流体数据值

历史数据不包括金条中间价格变动的记录；仅 [开盘价](#)、[最高价](#)、[最低价](#) 和 [收盘价](#) (OHLC)。

然而，在实时柱（工具市场开盘时运行的柱）上，[最高价](#)、[最低价](#) 和 [收盘价](#) 不是固定的；在实时柱关闭且其 HLC 值固定之前，它们可以多次更改值。它们是流动的。这导致脚本有时对历史数据和实时数据的工作方式不同，其中只有开盘价[在](#)柱期间不会改变。

[任何实时使用最高价](#)、[最低价](#) 和 [收盘价](#) 等值的脚本都会产生可能无法在历史柱上重复的计算 — 因此需要重新绘制。

让我们看一下这个简单的脚本。它检测 [EMA之上和之下的收盘价](#)（在实时柱中，这对应于工具的当前价格）的交叉：



```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma)
xDn = ta.crossunder(close, ma)
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

- 注意：

该脚本使用[bgcolor\(\)](#)在收盘价穿过 EMA 时将背景着色为绿色，在 EMA 下方交叉时将背景着色为红色。屏幕快照在 30 秒图表上实时显示脚本。已检测到 EMA 交叉，因此实时柱的背景为绿色。这里的问题是，没有任何东西可以保证这个条件在实时条结束之前一直成立。箭头指向的计时器显示实时栏中还剩 21 秒，在此之前任何事情都可能发生。我们正在见证一个重新绘制的脚本。

为了防止这种重新绘制，我们必须重写脚本，使其不使用实时柱期间波动的值。这将需要使用已过去的柱（通常是最前一个柱）的值，或开盘价。[这些值不会实时变化。](#)

我们可以通过多种方式实现这一目标。此方法为我们的交叉检测添加了一个条件，这要求脚本在柱的最后一次迭代（即收盘价和价格确认时）执行。这是避免重画的简单方法：[and barstate.isconfirmed](#)

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma) and barstate.isconfirmed
xDn = ta.crossunder(close, ma) and barstate.isconfirmed
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

这使用在前一柱上检测到的交叉：

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma)[1]
xDn = ta.crossunder(close, ma)[1]
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

这仅使用确认的[收盘价](#) 和 EMA 值进行计算：

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close[1], 5)
xUp = ta.crossover(close[1], ma)
xDn = ta.crossunder(close[1], ma)
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

这会检测实时柱的[开盘价](#) 与前一柱的 EMA 值之间的交叉。请注意，EMA 是使用 [close](#) 计算的，因此它会重新绘制。我们必须确保使用确认值来检测交叉，因此 `ma[1]` 在交叉检测逻辑中：

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(open, ma[1])
xDn = ta.crossunder(open, ma[1])
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

所有这些方法都有一个共同点：虽然它们阻止重绘，但它们也会晚于重绘脚本触发信号。如果想避免重新粉刷，这是不可避免的妥协。你不能鱼与熊掌兼得。

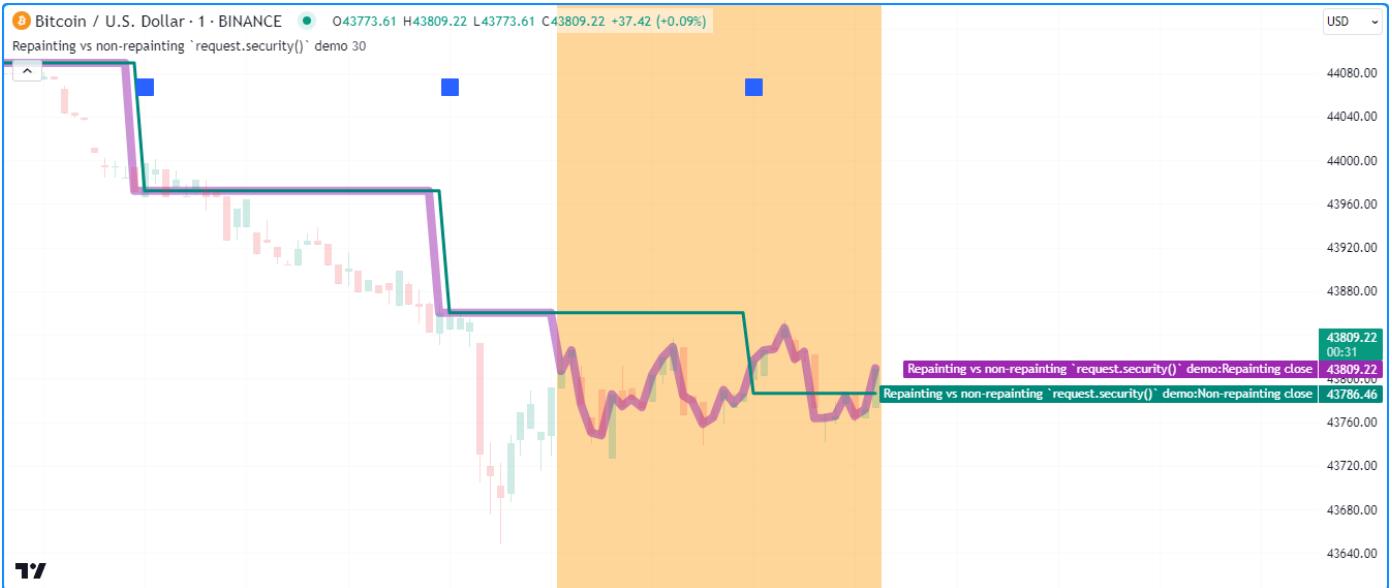
## 重新绘制 [request.security\(\)](#) 调用

`request.security()` 函数在历史柱和实时柱上的行为不同。在历史柱上，它仅从其请求的上下文中返回 已确认 的值，而在实时柱上，它可以返回未确认的值。当脚本重新开始执行时，具有实时状态的柱形图将变为历史柱形图，因此将仅包含它在这些柱形图上确认的值。如果 [request.security\(\)](#) 返回的值 在没有上下文确认的情况下在实时柱上波动，脚本将在重新启动执行时重新绘制它们。有关详细说明，请参阅[其他时间范围和数据页面](#) 的[历史和实时行为部分](#)。

通过使用历史引用运算符 `[expression]` 将参数偏移至少一根柱，并使用 [barmerge.lookahead\\_on](#) 作为 请求中的参数，可以确保较高时间范围的数据请求仅返回所有柱上的确认值，而不管柱状态如何。[security\(\)](#) 调用，如此处所述。`lookahead`

下面的脚本演示了重绘和非重绘 HTF 数据请求之间的区别。它包含两个`request.security()`调用。第一个函数调用请求来自的[关闭](#)数据而无需额外指定，第二个函数调用请求具有偏移量和`barmerge.lookahead_on` `higherTimeframe` 的相同系列。

正如我们在所有[实时](#)柱（具有橙色背景的柱）上看到的，`repaintingClose` 包含未经确认而波动的值，这意味着当脚本重新启动其执行时 `higherTimeframe` 它将重新绘制。另一方面 `nonRepaintingClose`，在实时和历史柱上的行为相同，即，它仅在新的、已确认的数据可用时更改其值：



```
//@version=5
indicator("Repainting vs non-repainting `request.security()` demo", overlay = true)

//@variable The timeframe to request data from.
string higherTimeframe = input.timeframe("30", "Timeframe")

if timeframe.in_seconds() > timeframe.in_seconds(higherTimeframe)
    runtime.error("The 'Timeframe' input is smaller than the chart's timeframe. Choose a higher timeframe.")

//@variable The current `close` requested from the `higherTimeframe`. Fluctuates without confirmation on realtime bars.
float repaintingClose = request.security(syminfo.tickerid, higherTimeframe, close)
//@variable The last confirmed `close` requested from the `higherTimeframe`.
// Behaves the same on historical and realtime bars.
float nonRepaintingClose = request.security(
    syminfo.tickerid, higherTimeframe, close[1], lookahead = barmerge.lookahead_on
)

// Plot the values.
plot(repaintingClose, "Repainting close", color.new(color.purple, 50), 8)
plot(nonRepaintingClose, "Non-repainting close", color.teal, 3)
// Plot a shape when a new `higherTimeframe` starts.
plotshape(timeframe.change(higherTimeframe), "Timeframe change marker", shape.square,
location.top, size = size.small)
// Color the background on realtime bars.
```

```
bgcolor(barstate.isrealtime ? color.new(color.orange, 60) : na, title = "Realtime bar highlight")
```

- 注意：

当 `higherTimeframe` 如果低于图表的时间范围，此脚本会产生[运行时错误](#)。`higherTimeframe` 在历史柱上，和在每个时间范围结束 `repaintingClose` 时都有一个新值，并且在每个时间范围开始时都有一个新值。`nonRepaintingClose`

为了便于重用，下面是一个简单的 `noRepaintSecurity()` 函数，可以在脚本中应用该函数来请求非重绘更高的时间范围值：

```
//@function Requests non-repainting `expression` values from the context of the
`symbol` and `timeframe`.
noRepaintSecurity(symbol, timeframe, expression) =>
    request.security(symbol, timeframe, expression[1], lookahead =
barmerge.lookahead_on)
```

- 注意：

系列的偏移 `[1]` 量和 的使用是相互依赖的。在不损害功能完整性的情况下无法删除其中一个。`lookahead = barmerge.lookahead_on` 与普通的[request.security\(\)](#) 调用不同，此包装函数不能接受元组 `expression` 参数。对于多元素用例，可以传递[用户定义的类型](#)，其字段包含要请求的所需元素。

## 在较低的时间范围内使用 [request.security\(\)](#)

某些脚本使用[request.security\(\)](#) 来请求低于图表时间范围的数据。当专门设计用于处理较低时间范围内的柱线的函数在时间范围内发送时，这会很有用。当这种类型的用户定义函数需要检测柱内的第一个柱时（大多数情况都是如此），该技术将仅适用于历史柱。这是因为实时内部条尚未排序。其影响是此类脚本无法实时重现其在历史柱上的行为。例如，任何生成警报的逻辑都会有缺陷，并且需要不断刷新才能将经过的实时柱重新计算为历史柱。

当在低于图表的时间范围使用时，没有能够区分内部柱的专门函数，[request.security\(\)](#) 将仅返回图表柱扩张中最后一个内部柱的值，这通常没有用，也不会重现实时，因此导致重新绘制。

[出于所有这些原因，除非您了解在低于图表的时间范围内使用request.security\(\)](#) 的微妙之处，否则最好避免在这些时间范围内使用该函数。更高质量的脚本将具有检测此类异常的逻辑，并防止显示在使用较低时间范围时无效的结果。

对于更可靠的较低时间范围数据请求，请使用 `request.security_lower_tf()`，如[其他时间范围和数据页面的本](#) 节中所述。

## 未来通过 [request.security\(\)](#) 进行泄漏

当[request.security\(\)](#) 与一起使用来获取价格而不用 抵消系列时，它将返回历史柱上的未来数据，这是危险的误导。`lookahead = barmerge.lookahead_on``[1]`

虽然历史柱会神奇地在知道未来价格之前显示它们，但实时预测是不可能的，因为未来是未知的，而它应该是未知的，所以不存在未来柱。

这是一个例子：



```
// FUTURE LEAK! DO NOT USE!
//@version=5
indicator("Future leak", "", true)
futureHigh = request.security(syminfo.tickerid, "1D", high, lookahead =
barmerge.lookahead_on)
plot(futureHigh)
```

请注意较高的时间范围线如何显示时间范围发生之前的高值。避免这种影响的解决方案是使用本节中演示的函数。

脚本出版物中不允许使用前瞻来产生误导性结果，如[其他时间范围和数据页面的前瞻部分](#)中所述。使用这种误导性技术的脚本出版物将受到审核。

## 瓦里普

使用变量的[varip](#)声明模式的脚本（请参阅有关[varip](#)的部分了解更多信息）跨实时更新保存信息，这些信息无法在仅可用 OHLC 信息的历史柱上重现。此类脚本可能在实时中很有用，包括生成警报，但它们的逻辑无法进行回溯测试，它们在历史柱上的图也不能反映将实时完成的计算。

## 条形状态内置

使用[条形状态](#)的脚本可能会也可能不会重新绘制。正如我们在上一节中看到的，使用[barstate.isconfirmed](#)实际上是避免重绘的一种方法，重绘会在历史柱上重现，而历史柱总是“已确认”。但是，使用其他栏状态（例如[barstate.isnew](#)）将导致重新绘制。原因是，在历史柱上，[barstate.isnew](#)位于 `true` 柱的收盘价处，但在实时中，它位于 `true` 柱的开盘价上。使用其他柱状态变量通常会导致历史柱和实时柱之间出现某种类型的行为差异。

## 现在时间

内置的`timenow`返回当前时间。使用此变量的脚本无法显示一致的历史和实时行为，因此它们必须重新绘制。

## 策略

策略在每次实时更新时执行，而策略则在历史柱收盘时运行。它们很可能不会生成相同的顺序执行，因此需要重新绘制。请注意，发生这种情况时，回测结果也会失效，因为它们不能实时代表策略的行为。`calc_on_every_tick = true`

## 过去的绘图

在 5 个柱过去后检测枢轴的脚本通常会回到过去，以在过去 5 个柱的实际枢轴上绘制枢轴水平或值。这通常会导致毫无戒心的交易者查看历史柱上的图来推断，当枢轴实时发生时，相同的图将出现在枢轴发生时，而不是检测到时。

让我们看一个显示高枢轴价格的脚本，该脚本将价格置于过去（检测到枢轴后 5 个柱）：

```
//@version=5
indicator("Plotting in the past", "", true)
pHi = ta.pivothigh(5, 5)
if not na(pHi)
    label.new(bar_index[5], na, str.tostring(pHi, format.mintick) + "\n", yloc =
yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal)
```



注意：

- 此脚本会重新绘制，因为如果未显示价格的已过去实时柱被识别为枢轴（在实际枢轴发生后 5 个柱），则可能在其上放置价格。
- 显示看起来很棒，但可能会产生误导。

为其他人开发脚本时解决此问题的最佳解决方案是默认情况下不带偏移量地进行绘图，但为脚本用户提供过去通过输入打开绘图的选项，因此他们必须知道脚本正在做什么，例如：

```
//@version=5
indicator("Plotting in the past", "", true)
plotInThePast = input(false, "Plot in the past")
pHi = ta.pivothigh(5, 5)
if not na(pHi)
    label.new(bar_index[plotInThePast ? 5 : 0], na, str.tostring(pHi, format.mintick) +
"\n\"", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size =
size.normal)
```

## 数据集变化

### 起点

脚本开始在图表的第一个历史柱上执行，然后按顺序在每个柱上执行，如本手册有关 Pine Script™ 执行[模型](#)的页面中所述。如果第一个柱发生变化，则脚本通常不会像数据集在不同时间点开始时那样进行计算。

以下因素会影响您在图表上看到的柱形数量及其起点：

- 您持有的账户类型
- 数据供应商提供的历史数据
- 数据集的对齐要求，决定了其起点

这些是特定于账户的酒吧限制：

- 高级计划有 20000 个历史柱。
- Pro 和 Pro+ 计划的 10000 个历史柱。
- 其他计划的 5000 个历史酒吧。

起点是使用以下规则确定的，这些规则取决于图表的时间范围：

- **1、5、10、15、30 秒**：与一天的开始对齐。
- **1 - 14 分钟**：与一周的开始对齐。
- **15 - 29 分钟**：与月初对齐。
- **30 - 1439 分钟**：与年初对齐。
- **1440 分钟及以上**：与第一个可用历史数据点对齐。

随着时间的推移，这些因素会导致图表的历史记录在不同的时间点开始。这通常会对您的脚本计算产生影响，因为早期柱中计算结果的变化可能会波及数据集中的所有其他柱。例如，使用[ta.valuewhen\(\)](#)、[ta.barssince\(\)](#)或[ta.ema\(\)](#)等函数将产生随早期历史而变化的结果。

## 历史数据的修改

历史和实时柱是使用交易所/经纪商提供的两种不同的数据源构建的：历史数据和实时数据。当实时金条过去时，交易所/经纪商有时会对金条价格进行通常较小的调整，然后将其写入其历史数据。当刷新图表或在那些已过去的实时柱上重新执行脚本时，将使用历史数据构建和计算它们，其中将包含那些通常较小的价格修正（如果有）。

历史数据也可能因其他原因而被修改，例如股票分割。

## 会话

## 介绍

会话信息在 Pine Script™ 中可以通过三种不同的方式使用：

1. 包含从开始时间到结束时间和日期信息的会话字符串，可在[time\(\)](#)和[time\\_close\(\)](#)等函数中使用，以检测柱线何时处于特定时间段，并可选择将有效会话限制为特定日期。[input.session\(\)](#)函数提供了一种允许脚本用户通过脚本的“输入”选项卡定义会话值的方法（有关更多信息，请参阅[会话输入](#)部分）。
2. 会话状态内置变量（例如[session.ismarket](#)）可以识别柱属于哪个会话。
3. 使用[request.security\(\)](#)获取数据时，您还可以选择仅从常规会话或扩展会话返回数据。在这种情况下，**常规会话**和**扩展会话**的定义就是交换。它是仪器属性的一部分 - 不是用户定义的，如第 1 点所示。常规和扩展会话的概念与图表界面中使用的概念相同，例如在“图表设置/交易品种/会话”字段中。

以下部分介绍了在 Pine Script™ 中使用会话信息的两种方法。

注意：

- 并非 TradingView 上的所有用户帐户都有权访问扩展会话信息。
- Pine Script™ 中没有特殊的“会话”类型。相反，会话字符串是“string”类型，但必须符合会话字符串语法。

## 会话字符串

### 会话字符串规范

与[time\(\)](#)和[time\\_close\(\)](#)一起使用的会话字符串必须具有特定的格式。它们的语法是：

```
<time_period>:<days>
```

在哪里：

- <time\_period> 使用“hhmm”格式的时间，“hh”采用 24 小时格式，因此 1700 为下午 5 点。时间段的格式为“hhmm-hhmm”，逗号可以分隔多个时间段以指定离散时间段的组合。
- 例如，- 是一组从 1 到 7 的数字，指定会话的有效日期。  
1 是星期日，7 是星期六。

笔记

**默认日期为:**，这在 Pine Script™ v5 中与使用 (工作日) 的 `1234567` 早期版本不同。`23456` 对于要重现以前版本的行为的 v5 代码，它应该明确提及工作日，如 中所示 `"0930-1700:23456"`。

这些是会话字符串的示例：

- `"24x7"`

为期 7 天、24 小时的会议从午夜开始。

- `"0000-0000:1234567"`

相当于前面的例子。

- `"0000-0000"`

相当于前两个示例，因为默认天数为 `1234567`。

- `"0000-0000:23456"`

与前面的示例相同，但仅限周一至周五。

- `"2000-1630:1234567"`

隔夜时段从 20:00 开始，到次日 16:30 结束。它在一周中的所有日子都有效。

- `"0930-1700:146"`

每周日 (1)、周三 (4) 和周五 (6) 的会议于 9:30 开始并于 17:00 结束。

- `"1700-1700:23456"`

通宵会议。周一会议于周日 17:00 开始，周一 17:00 结束。周一至周五有效。

- `"1000-1001:26"`

周一 (2) 和周五 (6) 的奇怪会议仅持续一分钟。

- `"0900-1600,1700-2000"`

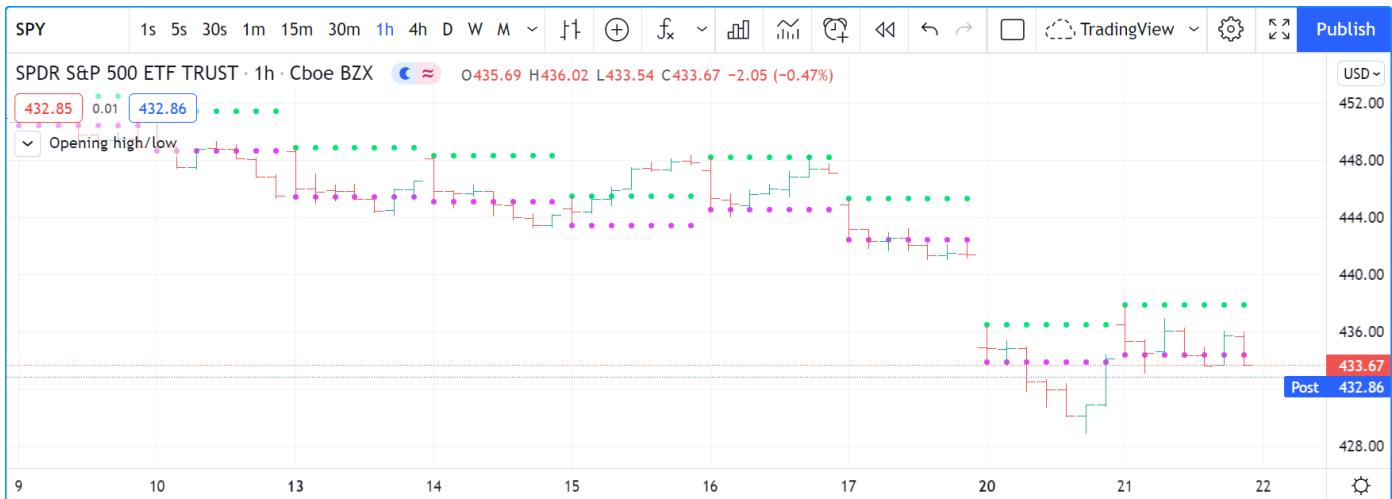
会议从 9:00 开始，从 16:00 到 17:00 休息，一直持续到 20:00。适用于一周中的每一天。

## 使用会话字符串

使用会话字符串定义的会话属性独立于交易所定义的会话（确定工具何时可以交易）。程序员可以完全自由地创建适合其目的的任何会话定义，这通常是检测柱线何时属于特定时间段。这是通过使用[time\(\)](#)函数的以下两个签名之一在 Pine Script™ 中完成的：

```
time(timeframe, session, timezone) → series int  
time(timeframe, session) → series int
```

在这里，我们使用带有参数的[time\(\)](#)在日内图表上 `session` 显示市场的开盘 [高点](#) 和 [低点值](#)：



```
//@version=5
indicator("Opening high/low", overlay = true)

sessionInput = input.session("0930-0935")

sessionBegins(sess) =>
    t = time("", sess)
    timeframe.isintraday and (not barstate.isfirst) and na(t[1]) and not na(t)

var float hi = na
var float lo = na
if sessionBegins(sessionInput)
    hi := high
    lo := low

plot(lo, "lo", color.fuchsia, 2, plot.style_circles)
plot(hi, "hi", color.lime, 2, plot.style_circles)
```

注意：

- 我们使用会话输入来允许用户指定他们想要检测的时间。我们仅在柱上查找会话的开始时间，因此我们在 "0930-0935" 默认值的开始时间和结束时间之间使用五分钟的间隔。
- 我们创建一个 `sessionBegins()` 函数来检测会话的开始。它的调用使用空字符串作为函数的参数，这意味着它使用图表的时间范围，无论是什么。该函数在以下情况下返回：`time("", sess)` `timeframe` `true`
  - 该图表使用日内时间范围（秒或分钟）。
  - 该脚本不在图表的第一个柱上，我们用来确保这一点。此检查可防止代码始终检测从第一个柱开始的会话，因为它始终存在。`(not barstate.isfirst)` `na(t[1]) and not na(t)` `true`
  - `time()` 调用在前一个柱上返回了 `na`，因为它不在会话的时间段内，并且它返回的值不是当前柱上的 `na`，这意味着该柱在会话的时间段内。

## 会话状态

三个内置变量允许您区分当前柱所属的会话类型。它们仅对日内时间范围有帮助：

- `true` 当柱属于常规交易时间时, `session.ismarket` 返回。
- `true` 当柱属于常规交易时间之前的延长时段时, `session.ispremarket` 返回。
- `true` 当柱属于常规交易时间之后的延长时段时, `session.ispostmarket` 返回。

## 使用带有 `request.security()` 的会话

当您的 TradingView 账户提供扩展会话访问权限时, 您可以选择使用“设置/交易品种/会话”字段查看其柱形。有两种类型的会话：

- **常规** (不包括上市前和上市后数据), 以及
- **扩展** (包括上市前和上市后数据)。

使用`request.security()` 函数访问数据的脚本可以返回扩展会话数据, 也可以不返回。这是仅获取常规会话数据的示例：



```
//@version=5
indicator("Example 1: Regular Session Data")
regularSessionData = request.security("NASDAQ:AAPL", timeframe.period, close,
barmerge.gaps_on)
plot(regularSessionData, style = plot.style_linebr)
```

如果您希望`request.security()`调用返回扩展会话数据, 则必须首先使用`ticker.new()`函数构建`request.security()`调用的第一个参数:



```
//@version=5
indicator("Example 2: Extended Session Data")
t = ticker.new("NASDAQ", "AAPL", session.extended)
extendedSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on)
plot(extendedSessionData, style = plot.style_linebr)
```

请注意，脚本情节中先前图表的空白现已填充。另外，请记住，我们的示例脚本不会在图表上产生背景颜色；这是由于图表的设置显示延长的时间。

`ticker.new()` 函数具有以下签名：

```
ticker.new(prefix, ticker, session, adjustment) → simple string
```

在哪里：

- `prefix` 是交换前缀，例如，`"NASDAQ"`
- `ticker` 是一个符号名称，例如，`"AAPL"`
- `session` 可以是 `session.extended` 或 `session.regular`。请注意，这不是会话字符串。
- `adjustment` 使用不同的标准调整价  
格：`adjustment.none`、`adjustment.splits`、`adjustment.dividends`。

我们的第一个例子可以重写为：

```
//@version=5
indicator("Example 1: Regular Session Data")
t = ticker.new("NASDAQ", "AAPL", session.regular)
regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on)
plot(regularSessionData, style = plot.style_linebr)
```

如果您想使用与图表主要交易品种相同的会话规范，请省略`ticker.new()`中的第三个参数；它是可选的。如果您希望代码明确声明您的意图，请使用`syminfo.session` 内置变量。它保存图表主要交易品种的会话类型：

```
//@version=5
indicator("Example 1: Regular Session Data")
t = ticker.new("NASDAQ", "AAPL", syminfo.session)
regularSessionData = request.security(t, timeframe.period, close, barmerge.gaps_on)
plot(regularSessionData, style = plot.style_linebr)
```

# 策略

## 介绍

Pine Script™ 策略根据历史和实时数据模拟交易的执行，以促进交易系统的回溯测试和前向测试。它们包含许多与 Pine Script™ 指标相同的功能，同时提供下达、修改和取消假设订单以及分析结果的能力。

当脚本使用 [strategy\(\)](#) 函数进行声明时，它可以访问 `strategy.*` 命名空间，在其中可以调用函数和变量来模拟订单并访问基本策略信息。此外，该脚本将在“策略测试器”选项卡中向外部显示信息和模拟结果。

## 一个简单的策略示例

以下脚本是一个简单的策略，模拟在两条移动平均线交叉时建立多头或空头头寸：

```
//@version=5
strategy("test", overlay = true)

// Calculate two moving averages with different lengths.
float fastMA = ta.sma(close, 14)
float slowMA = ta.sma(close, 28)

// Enter a long position when `fastMA` crosses over `slowMA`.
if ta.crossover(fastMA, slowMA)
    strategy.entry("buy", strategy.long)

// Enter a short position when `fastMA` crosses under `slowMA`.
if ta.crossunder(fastMA, slowMA)
    strategy.entry("sell", strategy.short)

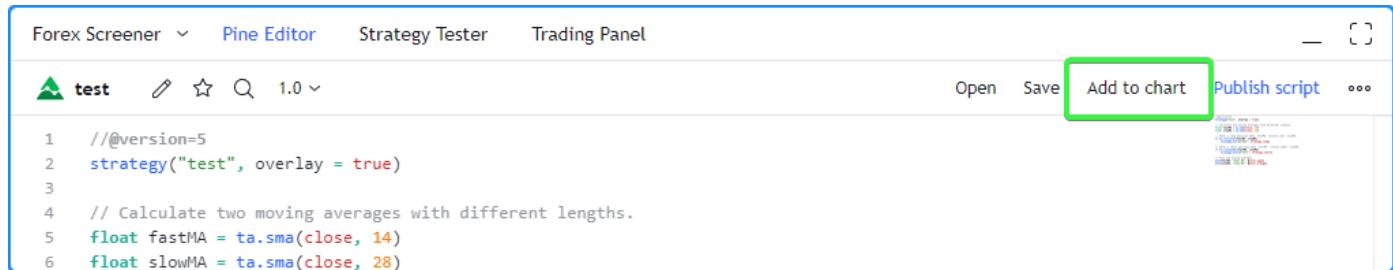
// Plot the moving averages.
plot(fastMA, "Fast MA", color.aqua)
plot(slowMA, "Slow MA", color.orange)
```

- 注意：

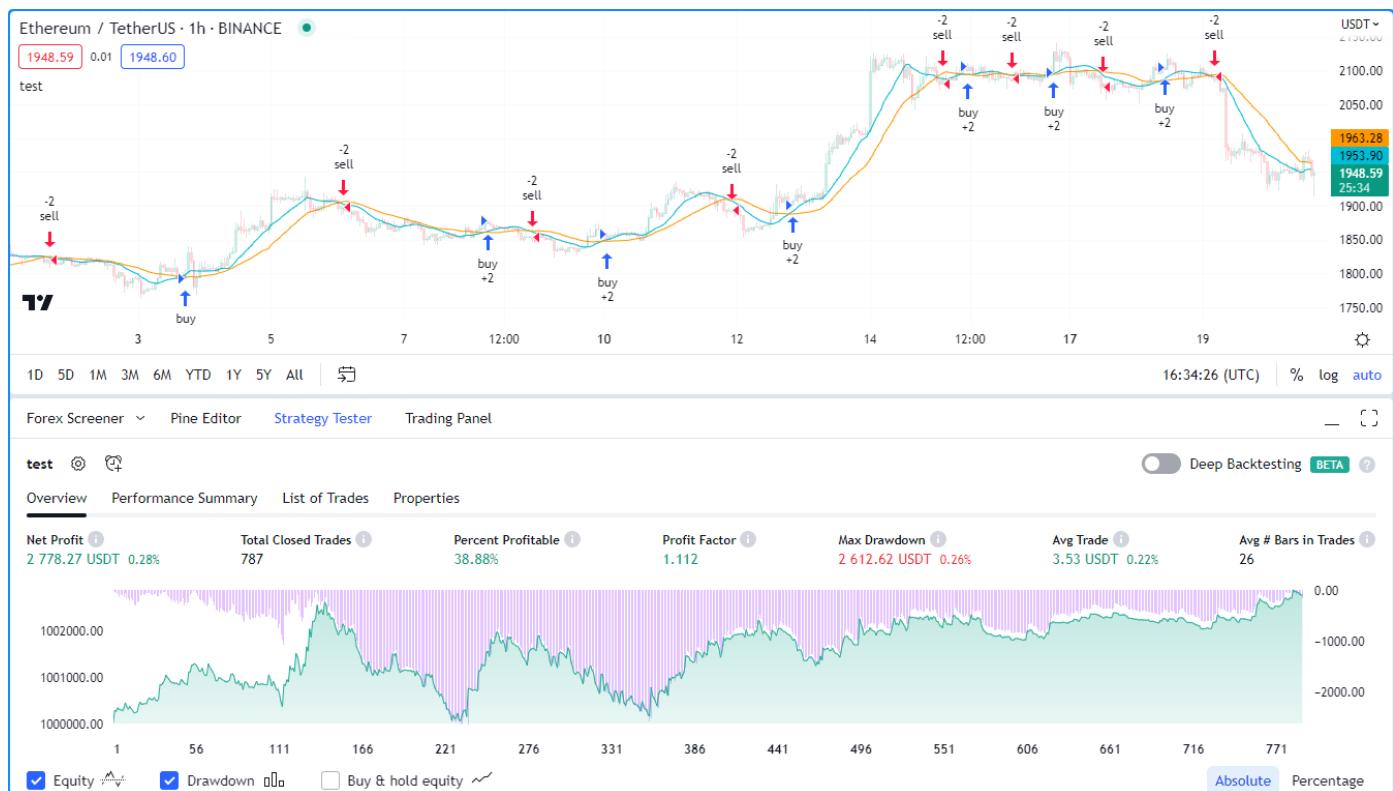
该行声明该脚本是一个名为“test”的策略，其视觉输出覆盖在主图表窗格上。`strategy("test" overlay = true)` [strategy.entry\(\)](#) 是脚本用来模拟“买入”和“卖出”订单的命令。当脚本下订单时，它还会 `id` 在图表上绘制订单并用箭头指示方向。两个 [plot\(\)](#) 函数用两种不同的颜色绘制移动平均值以供视觉参考。

## 将策略应用于图表

要测试策略，请将其应用于图表。您可以使用“指标和策略”对话框中的内置策略，也可以在 Pine 编辑器中编写自己的策略。单击“Pine Editor”选项卡中的“添加到图表”，将脚本应用到图表：



策略脚本编译并应用于图表后，它将在主图表窗格上绘制订单标记，并在下面的“策略测试器”选项卡中显示模拟性能结果：



## 笔记

默认情况下，应用于非标准图表（[Heikin Ashi](#)、[Renko](#)、[Line Break](#)、[Kagi](#)、[Point & Figure](#)和[Range](#)）的策略结果并不反映实际市场状况。策略脚本将在模拟过程中使用这些图表中的合成价格值，这些值通常与实际市场价格不一致，从而产生不切实际的回测结果。因此，我们强烈建议使用标准图表类型进行回测策略。或者，在 Heikin Ashi 图表上，用户可以通过启用“[策略属性](#)”中的“使用标准 OHLC 执行订单”选项，使用实际价格模拟订单。

## 策略测试器

策略测试器模块可用于使用 [策略\(\)](#) 函数声明的所有脚本。用户可以从图表下方的“策略测试器”选项卡访问此模块，在这里他们可以方便地可视化他们的策略并分析假设的绩效结果。

## 概述

策略测试器的“概述”选项卡显示了模拟交易序列的基本绩效指标以及净值和回撤曲线，可以快速查看策略绩效，而无需深入了解具体细节。本节中的图表将策略的净值曲线显示为以初始值为中心的基线图，买入并持有净值曲线显示为折线图，回撤曲线显示为直方图。用户可以使用图表下方的选项切换这些图并将其缩放为绝对值或百分比。



- 注意：

概览图使用两种尺度：左边是净值曲线，右边是回撤曲线。当用户单击这些图上的某个点时，这会将主图表视图定向到交易平仓的点。

## 性能总结

该模块的“绩效摘要”选项卡全面概述了策略的绩效指标。它显示三列：一列代表所有交易，一列代表所有多头，一列代表所有空头，为交易者提供有关策略的多头、空头和整体模拟交易表现的更详细的见解。

test ⌂ ⌂ ⌂ ⌂ ⌂ Deep Backtesting BETA ?

Title	All	Long	Short
Net Profit	2 778.27 USDT 0.28%	2 049.48 USDT 0.2%	728.79 USDT 0.07%
Gross Profit	27 631.14 USDT 2.76%	14 308.03 USDT 1.43%	13 323.11 USDT 1.33%
Gross Loss	24 852.87 USDT 2.49%	12 258.55 USDT 1.23%	12 594.32 USDT 1.26%
Max Run-up	3 054.07 USDT 0.3%		
Max Drawdown	2 612.62 USDT 0.26%		
Buy & Hold Return	1 527 822.71 USDT 152.78%		
Sharpe Ratio	-2.456		
Sortino Ratio	-0.927		
Profit Factor	1.112	1.167	1.058
Max Contracts Held	1	1	1
Open PL	143.09 USDT 0.01%		
Commission Paid	0.00 USDT	0.00 USDT	0.00 USDT
Total Closed Trades	787	394	393
Total Open Trades	1	0	1
Number Winning Trades	306	161	145
Number Losing Trades	480	232	248
Drawdown	38.80%	10.82%	26.82%

## 交易列表

交易列表选项卡提供了策略模拟交易的详细视图，其中包含基本信息，包括执行日期和时间、使用的订单类型（进场或出场）、交易的合约/股票/手数/单位数量、价格以及一些关键的贸易绩效指标。

test ⌂ ⌂ ⌂ ⌂ ⌂ Deep Backtesting BETA ?

Overview	Performance Summary	List of Trades	Properties						
Trade # ↑	Type	Signal	Date/Time	Price	Contracts	Profit	Cum. Profit	Run-up	Drawdown
	1	Exit Long	sell	2021-01-05 16:00	1 052.83 USDT	1	282.06 USDT 36.59%	282.06 USDT 0.03%	392.20 USDT 50.88%
2	Entry Long	buy	2021-01-02 14:00	770.77 USDT					
	Exit Short	buy	2021-01-05 19:00	1 074.00 USDT	1	-21.17 USDT -2.01%	260.89 USDT 0%	16.54 USDT 1.57%	42.05 USDT 3.99%
3	Entry Short	sell	2021-01-05 16:00	1 052.83 USDT					
	Exit Long	sell	2021-01-08 06:00	1 148.29 USDT	1	74.29 USDT 6.92%	335.18 USDT 0.01%	215.00 USDT 20.02%	34.00 USDT 3.17%
4	Entry Long	buy	2021-01-05 19:00	1 074.00 USDT					
	Exit Short	buy	2021-01-08 19:00	1 182.55 USDT	1	-34.26 USDT -2.98%	300.92 USDT 0%	22.88 USDT 1.99%	125.46 USDT 10.93%
5	Entry Short	sell	2021-01-08 06:00	1 148.29 USDT					
	Exit Long	sell	2021-01-09 06:00	1 185.40 USDT	1	2.85 USDT 0.24%	303.77 USDT 0%	52.12 USDT 4.41%	47.04 USDT 3.98%
6	Entry Long	buy	2021-01-08 19:00	1 182.55 USDT					
	Exit Short	buy	2021-01-09 16:00	1 219.96 USDT	1	-34.56 USDT -2.92%	269.21 USDT 0%	11.75 USDT 0.99%	51.60 USDT 4.35%
7	Entry Short	sell	2021-01-09 06:00	1 185.40 USDT					
	Exit Long	sell	2021-01-10 22:00	1 270.59 USDT	1	50.63 USDT 4.15%	319.84 USDT 0.01%	128.37 USDT 10.52%	49.96 USDT 4.1%

● 注意：

用户可以通过单击此列表中的特定交易来浏览图表上的特定交易时间。通过点击列表上方的“交易号”字段，用户可以按从第一个开始的升序或从最后一个开始的降序排列交易。

## 特性

“属性”选项卡提供有关策略配置及其应用的数据集的详细信息。它包括策略的日期范围、品种信息、脚本设置和策略属性。

- **日期范围**- 包括模拟交易的日期范围和总可用回测范围。
- **交易品种信息**- 包含交易品种名称和经纪商/交易所、图表的时间范围和类型、刻度大小、图表的点值以及基础货币。
- **策略输入**- 概述脚本设置的“输入”选项卡中提供的策略脚本中使用的各种参数和变量。
- **策略属性**- 提供交易策略配置的概述。它包括基本细节，例如初始资本、基础货币、订单大小、保证金、金字塔、佣金和滑点。此外，本节还重点介绍了对策略计算行为所做的任何修改。

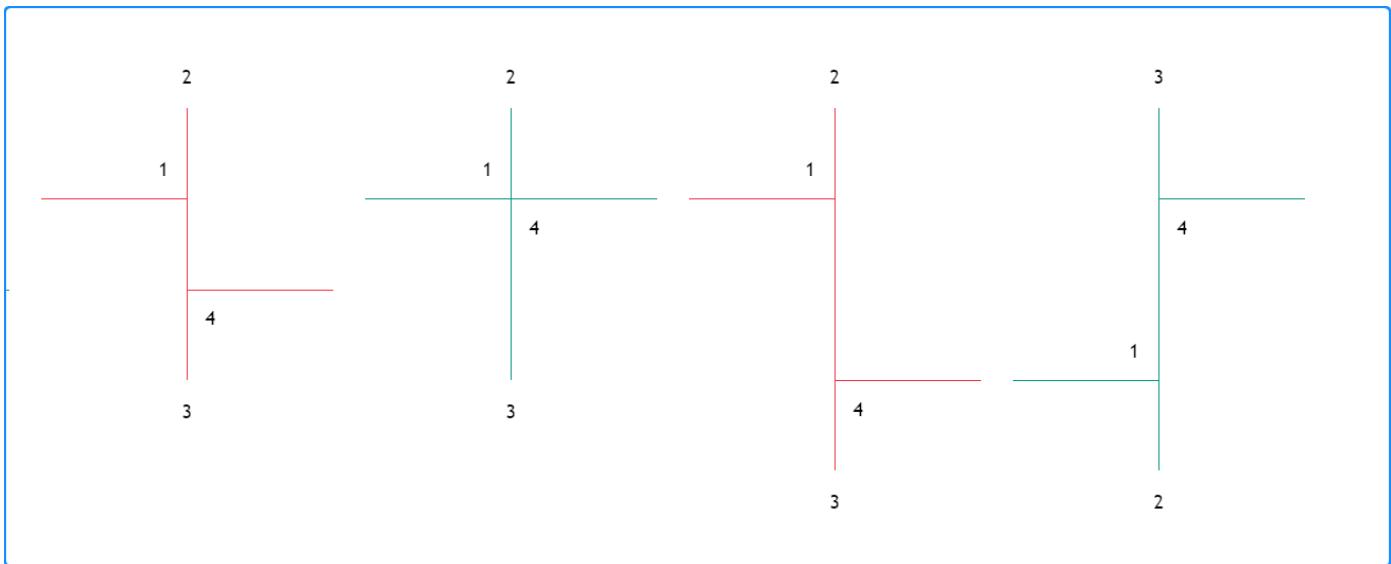
The screenshot shows the TradingView interface with the 'Properties' tab selected. At the top, there are tabs for 'Overview', 'Performance Summary', 'List of Trades', and 'Properties'. A 'Deep Backtesting' button is also visible. The main content area displays strategy properties under three sections: 'Date range' (Trading range: 2021-01-02 14:00 – 2023-04-20 16:00, Backtesting range: 2021-01-01 00:00 – 2023-04-20 16:00), 'Symbol info' (Symbol: BINANCE:ETHUSDT, Timeframe: 1 hour, Chart type: Candles, Currency: USDT, Tick Size: 0.01, Precision: Default), and 'Strategy properties' (Initial capital: 1000000 USDT, Order size: 1 Contracts, Pyramiding: 1 orders, Commission: 0%, Slippage: 0 ticks).

## 经纪商模拟器

TradingView 利用经纪商模拟器来模拟交易策略的性能。与现实生活中的交易不同，模拟器严格使用可用的图表价格进行订单模拟。因此，模拟只能在柱线收盘后进行历史交易，并且只能在新的价格变动上进行实时交易。有关此行为的更多信息，请参阅 Pine Script™[执行模型](#)。

由于模拟器只能使用图表数据，因此它会对柱内价格变动做出假设。它使用柱的开盘价、最高价和最低价来推断柱内活动，同时使用以下逻辑计算订单填充：

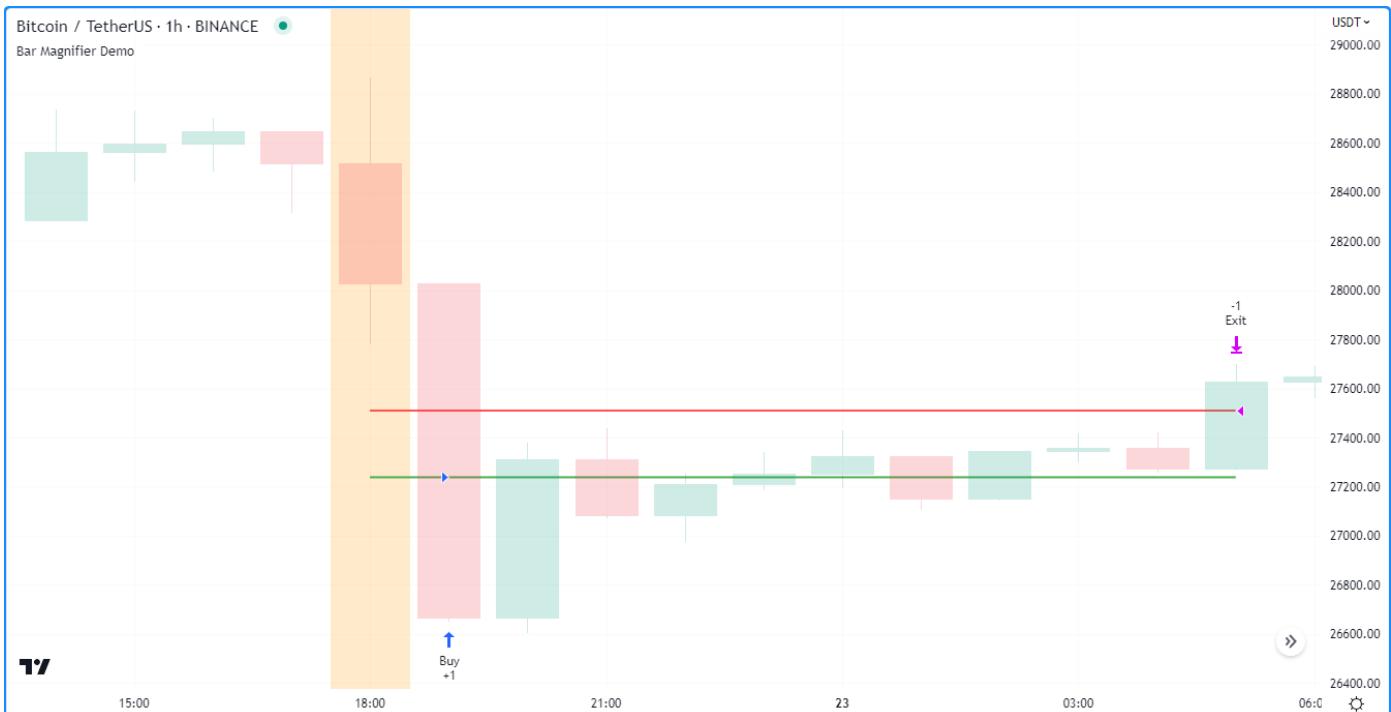
- 如果最高价比最低价更接近开盘价，则假定价格在柱上按以下顺序移动：开盘→最高价→最低价→收盘价。
- 如果最低价比最高价更接近开盘价，则假定价格在柱上按以下顺序移动：开盘→最低价→最高价→收盘价。
- 经纪商模拟器假设条形内的价格之间不存在差距；在模拟器的“眼睛”中，所有的柱内价格均可用于订单执行。



## 条形放大镜

高级账户持有者可以通过 `strategy("Bar Magnifier Demo", overlay = true, use_bar_magnifier = false)` 函数的参数 或脚本设置的“属性”选项卡中的“使用柱形放大镜”输入来覆盖经纪商模拟器的柱内假设。条形放大镜检查比图表更小的时间范围内的数据，以获得有关条形内价格行为的更精细信息，从而在模拟过程中实现更精确的订单填充。

为了进行演示，以下脚本在 `time` 值穿过时 在 `entryPrice` 处放置“买入”限价单 `entryPrice`，并在 `exitPrice` 处放置“退出”限价单，并绘制两条水平线以可视化订单价格。该脚本还使用 `orderTime` 和 `orderColor` 来突出显示背景，以指示策略下订单的时间：



```
//@version=5
strategy("Bar Magnifier Demo", overlay = true, use_bar_magnifier = false)

//@variable The UNIX timestamp to place the order at.
```

```

int orderTime = timestamp("UTC", 2023, 3, 22, 18)

//@variable Returns `color.orange` when `time` crosses the `orderTime`, false
otherwise.
color orderColor = na

// Entry and exit prices.
float entryPrice = h12 - (high - low)
float exitPrice = entryPrice + (high - low) * 0.25

// Entry and exit lines.
var line entryLine = na
var line exitLine = na

if ta.cross(time, orderTime)
    // Draw new entry and exit lines.
    entryLine := line.new(bar_index, entryPrice, bar_index + 1, entryPrice, color =
color.green, width = 2)
    exitLine := line.new(bar_index, exitPrice, bar_index + 1, exitPrice, color =
color.red, width = 2)

    // Update order highlight color.
    orderColor := color.new(color.orange, 80)

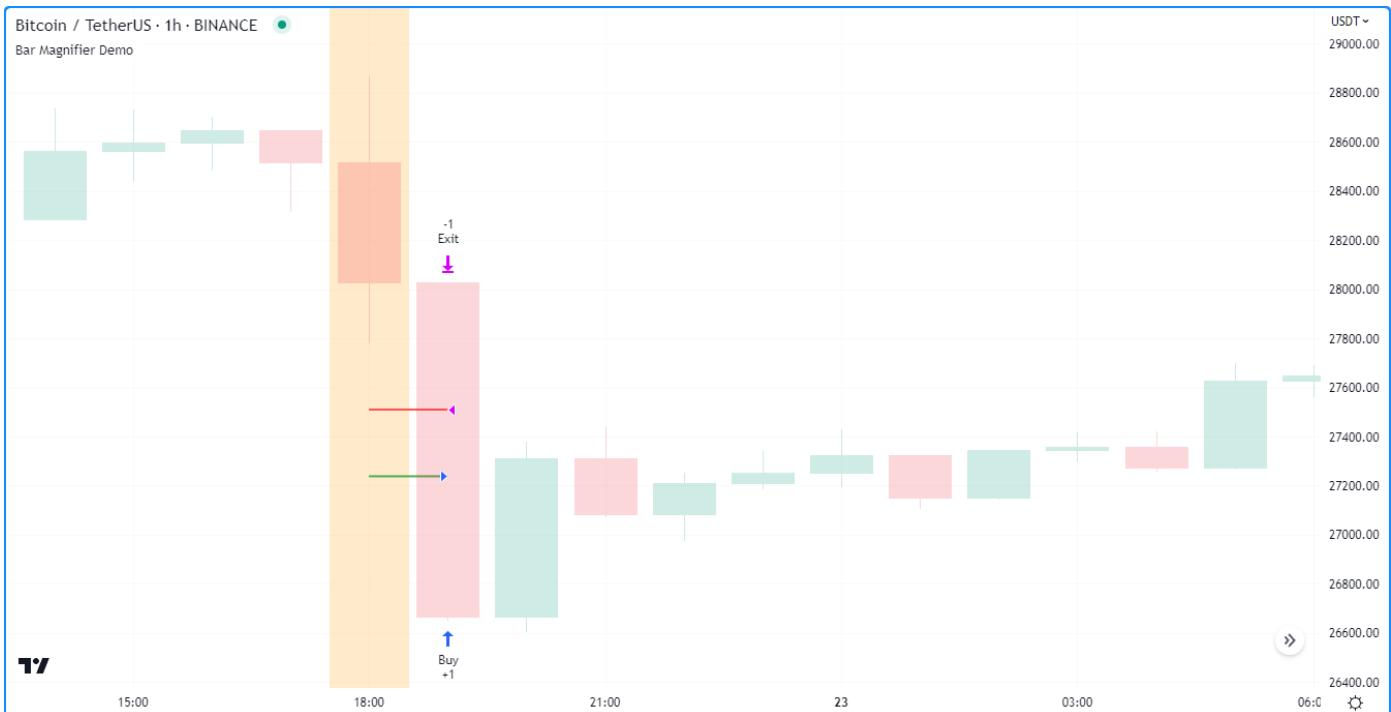
    // Place limit orders at the `entryPrice` and `exitPrice`.
    strategy.entry("Buy", strategy.long, limit = entryPrice)
    strategy.exit("Exit", "Buy", limit = exitPrice)

// Update lines while the position is open.
else if strategy.position_size > 0.0
    entryLine.set_x2(bar_index + 1)
    exitLine.set_x2(bar_index + 1)

bgcolor(orderColor)

```

正如我们在上图中看到的，经纪商模拟器假设柱内价格从开盘到高点，然后从高点到低点，然后从低点到收盘，这意味着模拟器假设“退出”订单无法在同一栏上填写。然而，在声明声明中包含之后，我们看到了不同的情  
况： `use_bar_magnifier = true`



## 笔记

脚本可以请求的最大内柱数为 200,000。由于此限制，一些具有较长历史记录的品种可能无法完全覆盖其开始图表条形，这意味着这些条形图上的模拟交易不会受到条形放大镜的影响。

## 订单和条目

就像现实生活中的交易一样，Pine 策略使用订单来管理仓位。在这种情况下，**订单**是模拟市场行为的命令，**交易**是订单执行后的结果。因此，要使用 Pine 建仓或平仓，用户必须创建带有指定其行为方式的参数的订单。

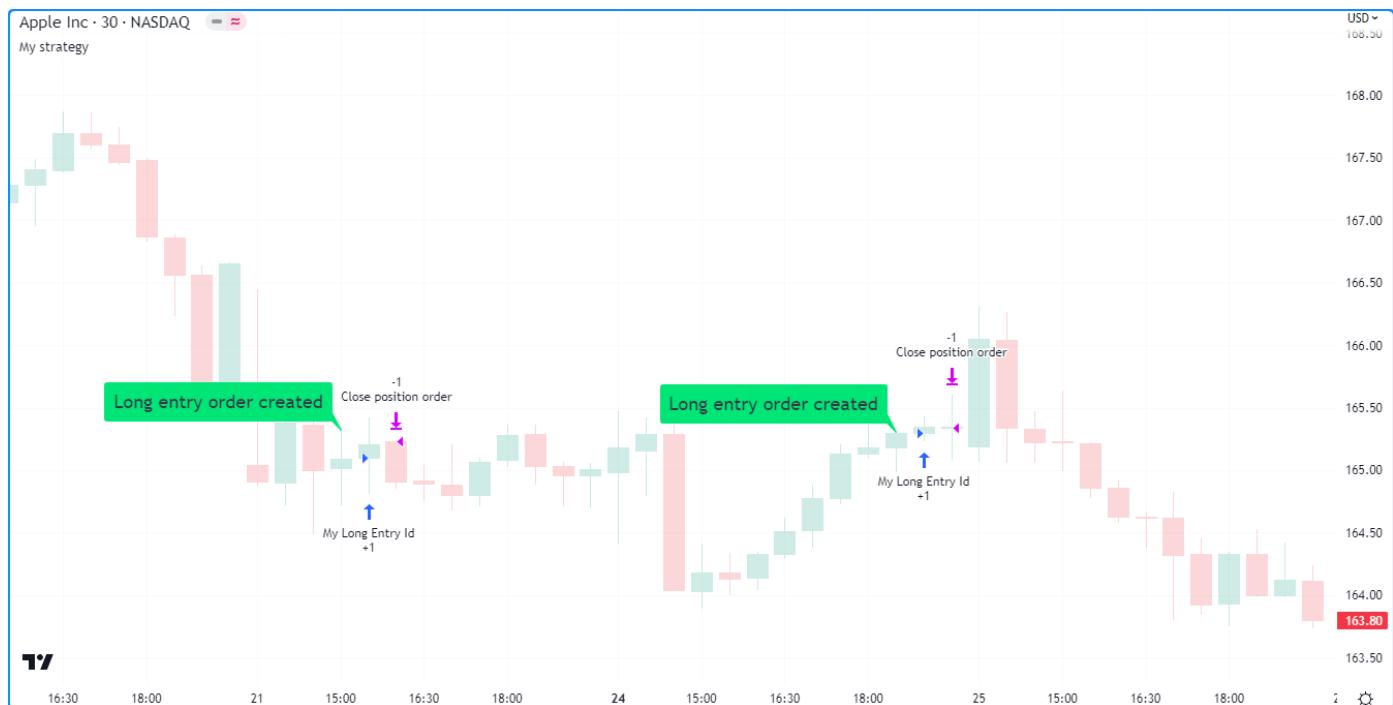
为了更仔细地了解订单如何运作以及它们如何成为交易，让我们编写一个简单的策略脚本：

```
//@version=5
strategy("My strategy", overlay = true, margin_long = 100, margin_short = 100)

//@function Displays text passed to `txt` when called.
debugLabel(txt) =>
    label.new(
        bar_index, high, text = txt, color=color.lime, style =
label.style_label_lower_right,
        textcolor = color.black, size = size.large
    )

longCondition = bar_index % 20 == 0 // true on every 20th bar
if (longCondition)
    debugLabel("Long entry order created")
    strategy.entry("My Long Entry Id", strategy.long)
strategy.close_all()
```

在此脚本中，我们定义了 `a`，`longCondition` 只要可以被 20 整除，即每 20 个柱，则 `a` 为真 `bar_index`。该策略在 `if` 结构中使用此条件，通过 `Strategy.entry()` 模拟入场订单，并使用用户定义的 `debugLabel()` 函数在入场价格处绘制标签。该脚本从全局范围调用 `strategy.close_all()` 来模拟平掉任何未平仓头寸的市价订单。让我们看看将脚本添加到图表后会发生什么：



图表上的蓝色箭头表示入场位置，紫色箭头表示策略平仓的点。请注意，标签位于实际入场点之前，而不是出现在同一柱上 - 这是正在执行的订单。默认情况下，Pine 策略会在执行订单之前等待下一个可用的价格变动，因为在同一价格变动上执行订单是不现实的。此外，它们会在每个历史柱的收盘价时重新计算，这意味着在这种情况下，下一个可用的订单执行价格是下一个柱的开盘价。因此，默认情况下，所有订单都会延迟一个图表栏。

需要注意的是，虽然脚本从全局范围调用 `strategy.close_all()`，强制在每个柱上执行，但如果策略不模拟未平仓头寸，则函数调用不会执行任何操作。如果有未平仓头寸，该命令会发出市价单来平仓，该订单在下一个可用报价时执行。例如，当 `longCondition` 20 柱上的为 `true` 时，该策略将下一个挂单以在下一个价格变动处（即第 21 柱的开盘价）处成交。一旦脚本在该柱的收盘价上重新计算其值，该函数就会下一个订单平仓，该仓位在第 22 条柱的开盘价处填充。

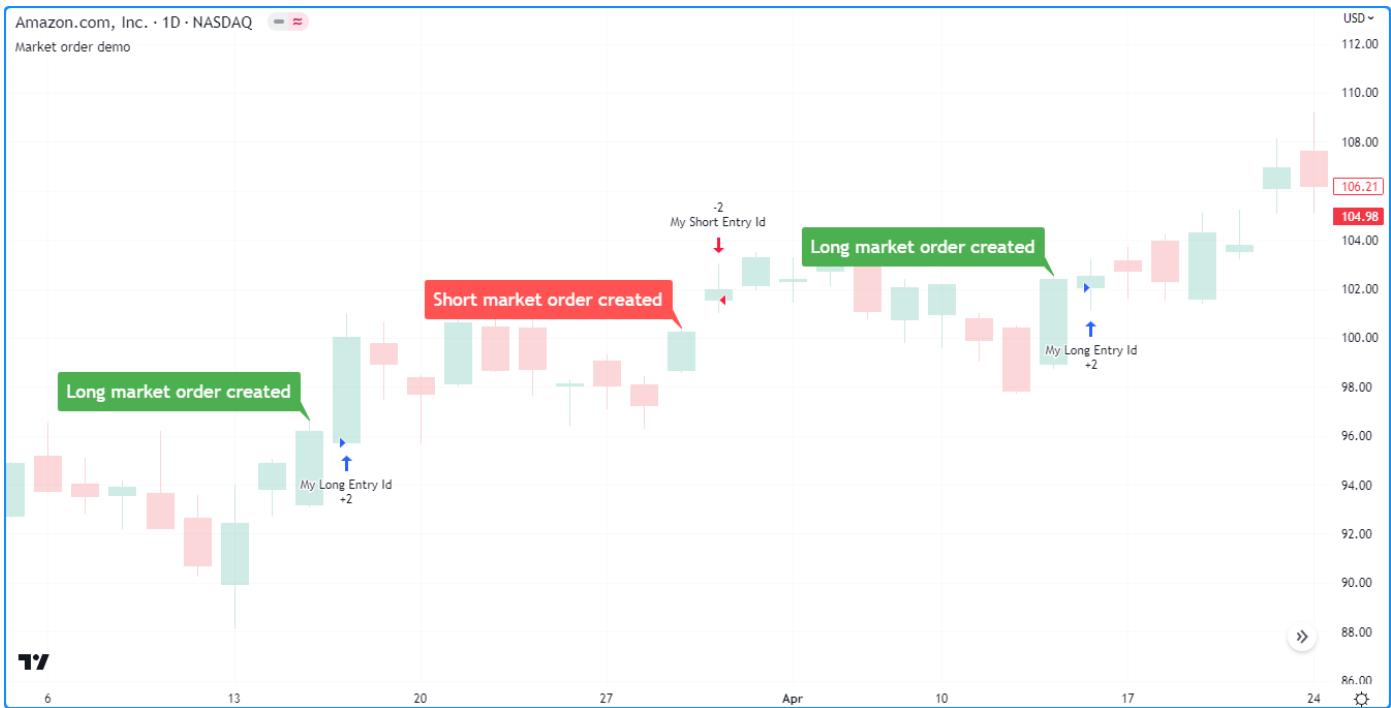
## 订单类型

Pine Script™ 策略允许用户根据其特定需求模拟不同的订单类型。主要值得注意的类型有 **市价**、**限价**、**止损**和**止损限价**。

### 市价订单

市价订单是最基本的订单类型。他们制定了尽快买入或卖出证券的策略，无论价格如何。因此，它们总是在下一个可用价格变动时执行。默认情况下，所有 `strategy.*()` 生成订单的函数都会专门生成市价订单。

以下脚本模拟可以被 `bar_index` 整除时的多头市价单。否则，当可以被 整除时，它会模拟空头市价单，从而产生每条柱交替进行多头和空头交易的策略：`2 * cycleLength``bar_index``cycleLength``cycleLength`



```

//@version=5
strategy("Market order demo", overlay = true, margin_long = 100, margin_short = 100)

//@variable Number of bars between long and short entries.
cycleLength = input.int(10, "Cycle length")

//@function Displays text passed to `txt` when called.
debugLabel(txt, lblColor) => label.new(
    bar_index, high, text = txt, color = lblColor, textcolor = color.white,
    style = label.style_label_lower_right, size = size.large
)

//@variable Returns `true` every `2 * cycleLength` bars.
longCondition = bar_index % (2 * cycleLength) == 0
//@variable Returns `true` every `cycleLength` bars.
shortCondition = bar_index % cycleLength == 0

// Generate a long market order with a `color.green` label on `longCondition`.
if longCondition
    debugLabel("Long market order created", color.green)
    strategy.entry("My Long Entry Id", strategy.long)
// Otherwise, generate a short market order with a `color.red` label on
// `shortCondition`.
else if shortCondition
    debugLabel("Short market order created", color.red)
    strategy.entry("My Short Entry Id", strategy.short)

```

## 限价订单

限价订单命令策略以特定价格或更好的价格建仓（多头订单低于指定价格，空头订单高于指定价格）。当当前市场价格优于订单命令的 `limit` 参数时，订单将被执行，而无需等待市场价格达到限价水平。

要在脚本中模拟限价单，请使用参数将价格值传递给下单命令 `limit`。以下示例在之前 100 根柱线收盘价下方 800 个报价点处设置限价单 `last_bar_index`：



```
//@version=5
strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100)

//@function Displays text passed to `txt` and a horizontal line at `price` when called.
debugLabel(price, txt) =>
    label.new(
        bar_index, price, text = txt, color = color.teal, textcolor = color.white,
        style = label.style_label_lower_right, size = size.large
    )
    line.new(
        bar_index, price, bar_index + 1, price, color = color.teal, extend =
        extend.right,
        style = line.style_dashed
    )

// Generate a long limit order with a label and line 100 bars before the
// `last_bar_index`.
if last_bar_index - bar_index == 100
    limitPrice = close - syminfo.mintick * 800
    debugLabel(limitPrice, "Long Limit order created")
    strategy.entry("Long", strategy.long, limit = limitPrice)
```

请注意脚本如何放置标签并在交易前几根柱线开始该线。只要价格保持在 `limitPrice` 价值之上，订单就无法执行。一旦市场价格达到限制，该策略就会执行中线交易。如果我们将 `limitPrice` 800 个报价点设置为高于收盘价而不是低于800 个报价点，则订单将立即成交，因为价格已经处于更好的值：



```
//@version=5
strategy("Limit order demo", overlay = true, margin_long = 100, margin_short = 100)

//@function Displays text passed to `txt` and a horizontal line at `price` when called.
debugLabel(price, txt) =>
    label.new(
        bar_index, price, text = txt, color = color.teal, textcolor = color.white,
        style = label.style_label_lower_right, size = size.large
    )
    line.new(
        bar_index, price, bar_index + 1, price, color = color.teal, extend =
        extend.right,
        style = line.style_dashed
    )

// Generate a long limit order with a label and line 100 bars before the
// `last_bar_index`.
if last_bar_index - bar_index == 100
    limitPrice = close + syminfo.mintick * 800
    debugLabel(limitPrice, "Long Limit order created")
    strategy.entry("Long", strategy.long, limit = limitPrice)
```

## 止损单和止损限价单

止损订单命令一种策略在价格达到指定 `stop` 价格或更差的值（多头订单高于指定价格，空头订单低于指定价格）后模拟另一个订单。它们本质上与限价订单相反。当当前市场价格差于 `stop` 参数时，策略将触发后续订单，而无需等待当前价格达到止损水平。如果下单命令包含 `limit` 参数，则后续订单将是指定值的限价订单。否则，这将是市价订单。

下面的脚本在 100 根柱线上方 800 个点处设置了止损单 `close`。在此示例中，策略 `stop` 在下单后市场价格与价格交叉一些柱时进入多头头寸。请注意，下订单时的初始价格优于传递给 的价格 `stop`。等效的限价订单将在以下图表栏上填写：



```
//@version=5
strategy("Stop order demo", overlay = true, margin_long = 100, margin_short = 100)

//@function Displays text passed to `txt` when called and shows the `price` level on
//the chart.
debugLabel(price, txt) =>
    label.new(
        bar_index, high, text = txt, color = color.teal, textcolor = color.white,
        style = label.style_label_lower_right, size = size.large
    )
    line.new(bar_index, high, bar_index, price, style = line.style_dotted, color =
color.teal)
    line.new(
        bar_index, price, bar_index + 1, price, color = color.teal, extend =
extend.right,
        style = line.style_dashed
    )

// Generate a long stop order with a label and lines 100 bars before the last bar.
if last_bar_index - bar_index == 100
```

```

stopPrice = close + syminfo.mintick * 800
debugLabel(stopPrice, "Long Stop order created")
strategy.entry("Long", strategy.long, stop = stopPrice)

```

`limit` 同时使用和参数的下单命令 `stop` 会生成止损限价订单。该订单类型等待价格突破止损水平，然后以指定 `limit` 价格下限价单。

让我们修改之前的脚本来模拟和可视化止损限价单。在此示例中，我们使用 `low` 100 个柱之前的值作为 `limit` 入场命令中的价格。该脚本还显示标签和价格水平，以指示策略何时穿越 `stopPrice`，即策略何时激活限价单。请注意市场价格最初如何达到限价水平，但该策略不会模拟交易，因为价格必须穿过 才能将 `stopPrice` 挂起的限价订单放置在 `limitPrice`：



```

//@version=5
strategy("Stop-Limit order demo", overlay = true, margin_long = 100, margin_short = 100)

//@function Displays text passed to `txt` when called and shows the `price` level on the chart.
debugLabel(price, txt, lblColor, lineWidth = 1) =>
    label.new(
        bar_index, high, text = txt, color = lblColor, textColor = color.white,
        style = label.style_label_lower_right, size = size.large
    )
    line.new(bar_index, close, bar_index, price, style = line.style_dotted, color = lblColor, width = lineWidth)
    line.new(
        bar_index, price, bar_index + 1, price, color = lblColor, extend =
        extend.right,
        style = line.style_dashed, width = lineWidth
    )

```

```

var float stopPrice = na
var float limitPrice = na

// Generate a long stop-limit order with a label and lines 100 bars before the last
bar.
if last_bar_index - bar_index == 100
    stopPrice := close + syminfo.mintick * 800
    limitPrice := low
    debugLabel(limitPrice, "", color.gray)
    debugLabel(stopPrice, "Long Stop-Limit order created", color.teal)
    strategy.entry("Long", strategy.long, stop = stopPrice, limit = limitPrice)

// Draw a line and label once the strategy activates the limit order.
if high >= stopPrice
    debugLabel(limitPrice, "Limit order activated", color.green, 2)
    stopPrice := na

```

## 下单命令

Pine Script™ 策略具有多种模拟下单功能，称为 **下单命令**。每个命令都有其独特的用途，其行为也与其他命令不同。

### [strategy.entry\(\)](#)

该命令模拟挂单。默认情况下，策略在调用此函数时下达市价单，但在使用 `stop` 和参数时，它们也可以创建止损单、限价单和止损限价单 `limit`。

为了简化开仓，[strategy.entry\(\)](#) 具有多种独特的行为。其中一种行为是，该命令可以反转公开市场头寸，而无需额外的函数调用。当使用[strategy.entry\(\)](#)下的订单被执行时，该函数将自动计算该策略需要平仓的金额，并 `qty` 默认以相反方向交易合约/股票/手数/单位。例如，如果某个策略在 `strategy.long` 方向上有 15 股的未平仓合约，并调用 [strategy.entry\(\)](#) 在 `strategy.short` 方向下市价单，则该策略下单时交易的金额为 15 股股票加上 `qty` 新空头订单的。

下面的示例演示了仅使用[strategy.entry\(\)](#) 调用来下挂单的策略。它每 100 根柱创建一次价值 15 股的多头市价订单 `qty`，每 25 根柱创建一次价值 5 股的空头市价订单 `qty`。该脚本针对出现的 `buyCondition` 和分别突出显示背景蓝色和红色 `sellCondition`：



```
//@version=5
strategy("Entry demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by 100.
sellCondition = bar_index % 25 == 0 and not buyCondition

if buyCondition
    strategy.entry("buy", strategy.long, qty = 15)
if sellCondition
    strategy.entry("sell", strategy.short, qty = 5)

bgcolor(buyCondition ? color.new(color.blue, 90) : na)
bgcolor(sellCondition ? color.new(color.red, 90) : na)
```

正如我们在上图中看到的，订单标记显示该策略在每次订单执行时交易了 20 股，而不是 15 股和 5 股。由于 [strategy.entry\(\)](#) 会自动反转头寸，除非通过 [strategy.risk.allow\\_entry\\_in\(\)](#) 另有指定函数中，当方向改变时，它将未平仓头寸规模（多头入场 15 股）添加到新订单的规模（空头入场 5 股）中，从而导致交易数量为 20 股。

请注意，在上面的示例中，尽管 `sellCondition` 在另一个之前出现了 3 次 `buyCondition`，但该策略仅在第一次出现时下达“卖出”订单。[Strategy.entry\(\)](#) 命令的另一个独特行为是它受脚本金字塔设置的影响。金字塔指定策略可以在同一方向上填充的连续订单的数量。默认情况下，其值为 1，这意味着该策略仅允许一个连续订单在任一方向上成交。用户可以通过[strategy\(\)](#) `pyramiding` 函数调用的参数或脚本设置的“属性”选项卡中的“金字塔”输入来设置策略金字塔值。

如果我们添加到之前脚本的声明语句中，该策略将允许同一方向最多三个连续交易，这意味着它可以在每次出现时模拟新的市价订单： `pyramiding = 3``sellCondition`



### [strategy.order\(\)](#)

该命令模拟基本命令。与大多数下单命令包含内部逻辑以简化与策略的接口不同，[strategy.order\(\)](#)使用指定的参数，而不考虑大多数其他策略设置。通过[strategy.order\(\)](#)下的订单可以开立新仓位并修改或平仓现有仓位。

以下脚本仅使用[strategy.order\(\)](#)调用来创建和修改条目。该策略模拟每 100 个柱状图 15 个单位的多头市价订单，然后每 25 个柱状图模拟 5 个单位的三个空头订单。该脚本突出显示背景蓝色和红色，以指示策略何时模拟“买入”和“卖出”订单：



```
//@version=5
strategy("Order demo", "test", overlay = true)
```

```

//@variable Is `true` on every 100th bar.
buyCond = bar_index % 100 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by 100.
sellCond = bar_index % 25 == 0 and not buyCond

if buyCond
    strategy.order("buy", strategy.long, qty = 15) // Enter a long position of 15
units.

if sellCond
    strategy.order("sell", strategy.short, qty = 5) // Exit 5 units from the long
position.

bgcolor(buyCond ? color.new(color.blue, 90) : na)
bgcolor(sellCond ? color.new(color.red, 90) : na)

```

这个特定的策略永远不会模拟空头头寸，因为与 [strategy.entry\(\)](#) 不同，[strategy.order\(\)](#) 不会自动反转头寸。使用该命令时，得出的市场持仓量为当前市场持仓量与已成交订单数量的净和。策略填写 15 个单位的“买入”订单后，它会执行三个“卖出”订单，每个订单将未平仓头寸减少 5 个单位，即  $15 - 5 * 3 = 0$ 。使用 [Strategy.entry\(\)](#) 相同的脚本会有不同的 [行为](#)，如上节所示的示例。

## [strategy.exit\(\)](#)

该命令模拟退出命令。它的独特之处在于它允许策略通过 `loss`、`stop`、`profit`、`limit` 和参数以止损、止盈和追踪止损订单的形式退出市场头寸或形成多个退出 `trail_*`。

[Strategy.exit\(\)](#) 命令最基本的用途是创建级别，在该级别中，策略将因损失过多资金（止损）、赚取足够资金（止盈）或两者（括号）而退出仓位。

该命令的止损和获利功能与两个参数相关。该函数的 `loss` 和 `profit` 参数将止损和止盈值指定为距挂单价格的定义数量，而其 `stop` 和 `limit` 参数则提供特定的止损和止盈价格值。函数调用中的绝对参数取代相对参数。如果 [strategy.exit\(\)](#) 调用包含 `profit` 和 `limit` 参数，该命令将优先考虑该 `limit` 值并忽略该 `profit` 值。同样，只有 `stop` 当函数调用包含 `stop` 和 `loss` 参数时，它才会考虑该值。

### 笔记

尽管与 [strategy.entry\(\)](#) 和 [strategy.order\(\)](#) 命令中的参数共享相同的名称，但在 [strategy.exit\(\)](#) `limit` 中，和 `stop` 参数的工作方式有所不同。在第一种情况下，在命令中使用和将创建一个止损限价订单，该订单会在超过止损价格后打开一个限价订单。在第二种情况下，该命令将创建一个单独的限价和止损订单以从未平仓头寸退出。`limit``stop`

带有参数的 [strategy.exit\(\)](#) 中的所有退出订单 `from_entry` 都绑定到 `id` 相应的入市订单；当没有与 `from_entry` ID 关联的公开市场头寸或活跃挂单时，策略无法模拟退出订单。

以下策略通过 [strategy.entry\(\)](#) 下达“买入”挂单，并通过 [strategy.exit\(\)](#) 命令每 100 个柱下达止损和止盈订单。请注意，该脚本调用 [strategy.exit\(\)](#) 两次。“exit1”命令引用“buy1”挂单，“exit2”引用“buy”订单。该策略将仅模拟来自“exit2”的退出订单，因为“exit1”引用了不存在的订单 ID：



```

//@version=5
strategy("Exit demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0

//@variable Stop-loss price for exit commands.
var float stopLoss    = na
//@variable Take-profit price for exit commands.
var float takeProfit = na

// Place orders upon `buyCondition`.
if buyCondition
    if strategy.position_size == 0.0
        stopLoss    := close * 0.99
        takeProfit := close * 1.01
        strategy.entry("buy", strategy.long)
        strategy.exit("exit1", "buy1", stop = stopLoss, limit = takeProfit) // Does
nothing. "buy1" order doesn't exist.
        strategy.exit("exit2", "buy", stop = stopLoss, limit = takeProfit)

// Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the
strategy simulates an exit.
if low <= stopLoss or high >= takeProfit
    stopLoss    := na
    takeProfit := na

plot(stopLoss, "SL", color.red, style = plot.style_circles)
plot(takeProfit, "TP", color.green, style = plot.style_circles)

```

- 注意：

每个退出命令发出的限价单和止损单不一定以指定价格成交。策略可以以更好的价格填写限价单，以更差的价格止损单，具体取决于经纪商模拟器可用的值范围。

如果用户未 `from_entry` 在 `strategy.exit()` 调用中提供参数，该函数将为每个打开的条目创建退出订单。

在此示例中，策略创建“buy1”和“buy2”挂单，并每 100 个柱调用 `strategy.exit()`（不带参数）。`from_entry` 从图表上的订单标记中我们可以看到，一旦市场价格达到或 `stopLoss` 值 `takeProfit`，该策略就会为“buy1”和“buy2”条目填写退出订单：



```
//@version=5
strategy("Exit all demo", "test", overlay = true, pyramiding = 2)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0

//@variable Stop-loss price for exit commands.
var float stopLoss    = na
//@variable Take-profit price for exit commands.
var float takeProfit = na

// Place orders upon `buyCondition`.
if buyCondition
    if strategy.position_size == 0.0
        stopLoss    := close * 0.99
        takeProfit := close * 1.01
        strategy.entry("buy1", strategy.long)
        strategy.entry("buy2", strategy.long)
        strategy.exit("exit", stop = stopLoss, limit = takeProfit) // Places orders to exit
all open entries.
```

```

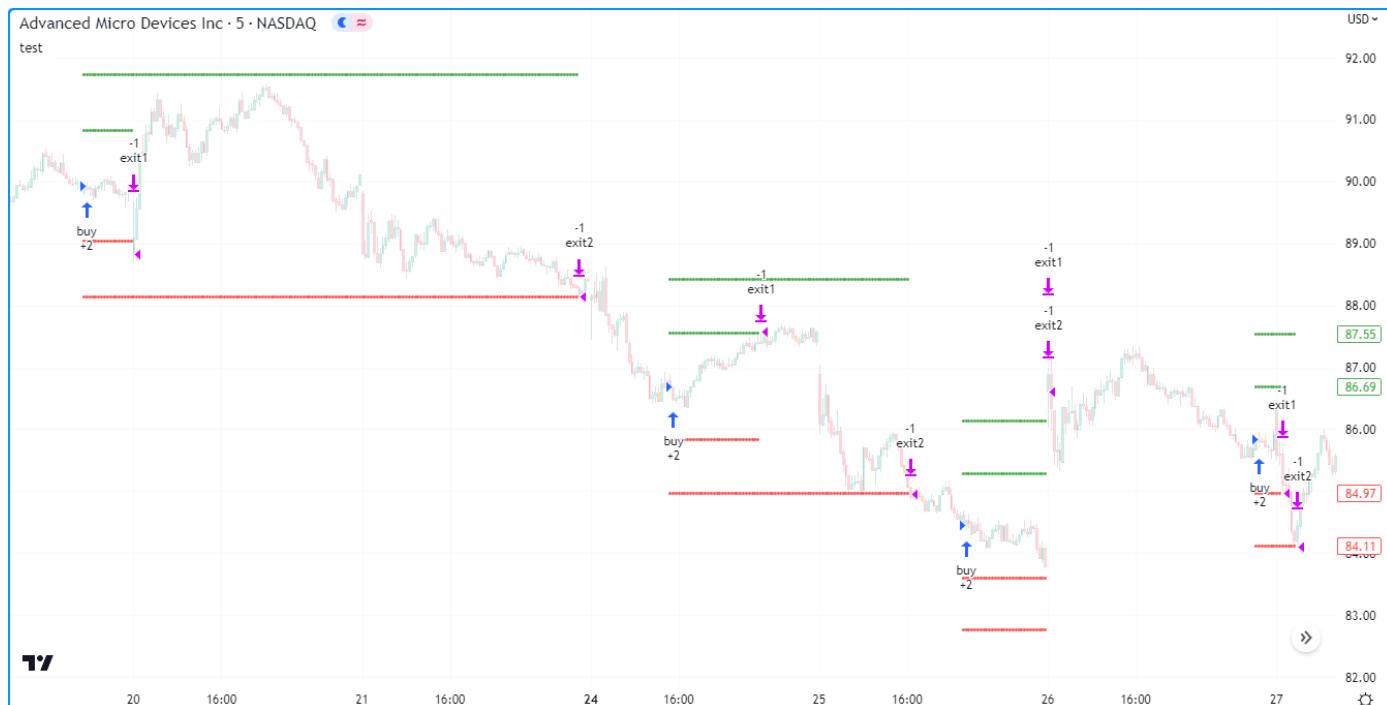
// Set `stopLoss` and `takeProfit` to `na` when price touches either, i.e., when the
// strategy simulates an exit.
if low <= stopLoss or high >= takeProfit
    stopLoss := na
    takeProfit := na

plot(stopLoss, "SL", color.red, style = plot.style_circles)
plot(takeProfit, "TP", color.green, style = plot.style_circles)

```

一个策略可以从同一个入口ID多次退出，这有利于形成多级退出策略。执行多个平仓指令时，每笔订单数量必须是成交数量的一部分，且总和不得超过持仓量。如果 `qty` 函数的小于当前市场头寸的规模，该策略将模拟部分退出。如果 `qty` 价值超过未平仓头寸数量，它将减少订单，因为它无法填充比未平仓头寸更多的合约/股票/手数/单位。

在下面的示例中，我们修改了之前的“退出演示”脚本，以模拟每个入场点的两个止损和止盈订单。该策略下达 `qty` 两股的“买入”订单，一股的“退出1”止损和止盈订单，以及三股的 `qty` “退出2”止损和止盈订单： `qty`



```

//@version=5
strategy("Multiple exit demo", "test", overlay = true)

//@variable Is `true` on every 100th bar.
buyCondition = bar_index % 100 == 0

//@variable Stop-loss price for "exit1" commands.
var float stopLoss1 = na
//@variable Stop-loss price for "exit2" commands.
var float stopLoss2 = na
//@variable Take-profit price for "exit1" commands.
var float takeProfit1 = na
//@variable Take-profit price for "exit2" commands.
var float takeProfit2 = na

```

```

// Place orders upon `buyCondition`.
if buyCondition
    if strategy.position_size == 0.0
        stopLoss1 := close * 0.99
        stopLoss2 := close * 0.98
        takeProfit1 := close * 1.01
        takeProfit2 := close * 1.02
    strategy.entry("buy", strategy.long, qty = 2)
    strategy.exit("exit1", "buy", stop = stopLoss1, limit = takeProfit1, qty = 1)
    strategy.exit("exit2", "buy", stop = stopLoss2, limit = takeProfit2, qty = 3)

// Set `stopLoss1` and `takeProfit1` to `na` when price touches either.
if low <= stopLoss1 or high >= takeProfit1
    stopLoss1 := na
    takeProfit1 := na

// Set `stopLoss2` and `takeProfit2` to `na` when price touches either.
if low <= stopLoss2 or high >= takeProfit2
    stopLoss2 := na
    takeProfit2 := na

plot(stopLoss1, "SL1", color.red, style = plot.style_circles)
plot(stopLoss2, "SL2", color.red, style = plot.style_circles)
plot(takeProfit1, "TP1", color.green, style = plot.style_circles)
plot(takeProfit2, "TP2", color.green, style = plot.style_circles)

```

从图表上的订单标记可以看出，尽管指定 `qty` 值超过了交易金额，该策略还是执行了“exit2”订单。该脚本没有使用此数量，而是减小了订单大小以匹配剩余头寸。

- 注意：

通过[strategy.exit\(\)](#)调用生成的所有订单都 属于同一个[strategy.oca.reduce](#)组，这意味着当任一订单成交时，该策略会减少所有其他订单以匹配未平仓头寸。

值得注意的是，该命令生成的订单会保留一部分公开市场头寸以供退出。[strategy.exit\(\)](#)无法发出订单以退出已由另一个退出命令保留用于退出的部分仓位。

以下脚本模拟 100 根柱前 20 股的“买入”市价订单，其中“限价”订单和“止损”订单分别为 19 股和 20 股。正如我们在图表上看到的，该策略首先执行“止损”订单。但成交量仅为一股。由于脚本首先放置了“限价”订单，因此该策略保留其 `qty`（19 股）来平仓，仅留下一股由“止损”订单平仓：



```
//@version=5
strategy("Reserved exit demo", "test", overlay = true)

//@variable "stop" exit order price.
var float stop    = na
//@variable "limit" exit order price
var float limit   = na
//@variable Is `true` 100 bars before the `last_bar_index`.
longCondition = last_bar_index - bar_index == 100

if longCondition
    stop  := close * 0.99
    limit := close * 1.01
    strategy.entry("buy", strategy.long, 20)
    strategy.exit("limit", limit = limit, qty = 19)
    strategy.exit("stop", stop = stop, qty = 20)

bool showPlot = strategy.position_size != 0
plot(showPlot ? stop : na, "Stop", color.red, 2, plot.style_linebr)
plot(showPlot ? limit : na, "Limit 1", color.green, 2, plot.style_linebr)
```

[Strategy.exit\(\)](#)函数的另一个关键功能是，它可以创建追踪止损，即每当价格朝有利方向移动到更好的值时，止损订单会落后于市场价格指定的金额。这些订单有两个组成部分：激活级别和跟踪偏移量。激活水平是市场价格必须交叉才能激活追踪止损计算的值，通过参数以价格变动表示 `trail_points` 或通过参数表示为价格值 `trail_price`。如果退出调用包含两个参数，则该 `trail_price` 参数优先。追踪偏移量是止损点跟随市场价格的距离，通过参数以价格变动来表示 `trail_offset`。对于要创建和激活追踪止损的[strategy.exit\(\)](#)，函数调用必须包含 `trail_offset` 和 `trail_price` 或 `trail_points` 参数。

下面的示例显示了正在运行的追踪止损并可视化其行为。该策略模拟图表上最后一个柱之前 100 个柱的多头入场订单，然后在下一个柱上设置追踪止损。该脚本有两个输入：一个控制激活水平偏移（即，超过激活止损所需的入场价格的金额），另一个控制追踪偏移（即，当市场价格移动到所需方向上的更好值）。

图表上的绿色虚线显示市场价格必须穿越的水平才能触发追踪止损单。价格穿过该水平后，脚本会绘制一条蓝线来表示追踪止损。当价格上升到新的高值时，这对策略有利，因为这意味着头寸的价值正在增加，止损也会上升，以保持 `trailingStopOffset` 当前价格后面的价格变动距离。当价格下跌或没有达到新的高点时，止损值保持不变。最终，价格跌破止损，触发退出：



```
//@version=5
strategy("Trailing stop order demo", overlay = true, margin_long = 100, margin_short = 100)

//@variable Offset used to determine how far above the entry price (in ticks) the activation level will be located.
activationLevelOffset = input(1000, "Activation Level Offset (in ticks)")
//@variable Offset used to determine how far below the high price (in ticks) the trailing stop will trail the chart.
trailingStopOffset = input(2000, "Trailing Stop Offset (in ticks)")

//@function Displays text passed to `txt` when called and shows the `price` level on the chart.
debugLabel(price, txt, lblColor, hasLine = false) =>
    label.new(
        bar_index, price, text = txt, color = lblColor, textColor = color.white,
        style = label.style_label_lower_right, size = size.large
    )
    if hasLine
        line.new(
            bar_index, price, bar_index + 1, price, color = lblColor, extend =
            extend.right,
```

```

        style = line.style_dashed
    )

//@variable The price at which the trailing stop activation level is located.
var float trailPriceActivationLevel = na
//@variable The price at which the trailing stop itself is located.
var float trailingStop = na
//@variable Calculates the value that Trailing Stop would have if it were active at the
moment.
theoreticalStopPrice = high - trailingStopOffset * syminfo.mintick

// Generate a long market order to enter 100 bars before the last bar.
if last_bar_index - bar_index == 100
    strategy.entry("Long", strategy.long)

// Generate a trailing stop 99 bars before the last bar.
if last_bar_index - bar_index == 99
    trailPriceActivationLevel := open + syminfo.mintick * activationLevelOffset
    strategy.exit(
        "Trailing Stop", from_entry = "Long", trail_price = trailPriceActivationLevel,
        trail_offset = trailingStopOffset
    )
    debugLabel(trailPriceActivationLevel, "Trailing Stop Activation Level",
color.green, true)

// Visualize the trailing stop mechanic in action.
// If there is an open trade, check whether the Activation Level has been achieved.
// If it has been achieved, track the trailing stop by assigning its value to a
variable.
if strategy.opentrades == 1
    if na(trailingStop) and high > trailPriceActivationLevel
        debugLabel(trailPriceActivationLevel, "Activation level crossed", color.green)
        trailingStop := theoreticalStopPrice
        debugLabel(trailingStop, "Trailing Stop Activated", color.blue)

    else if theoreticalStopPrice > trailingStop
        trailingStop := theoreticalStopPrice

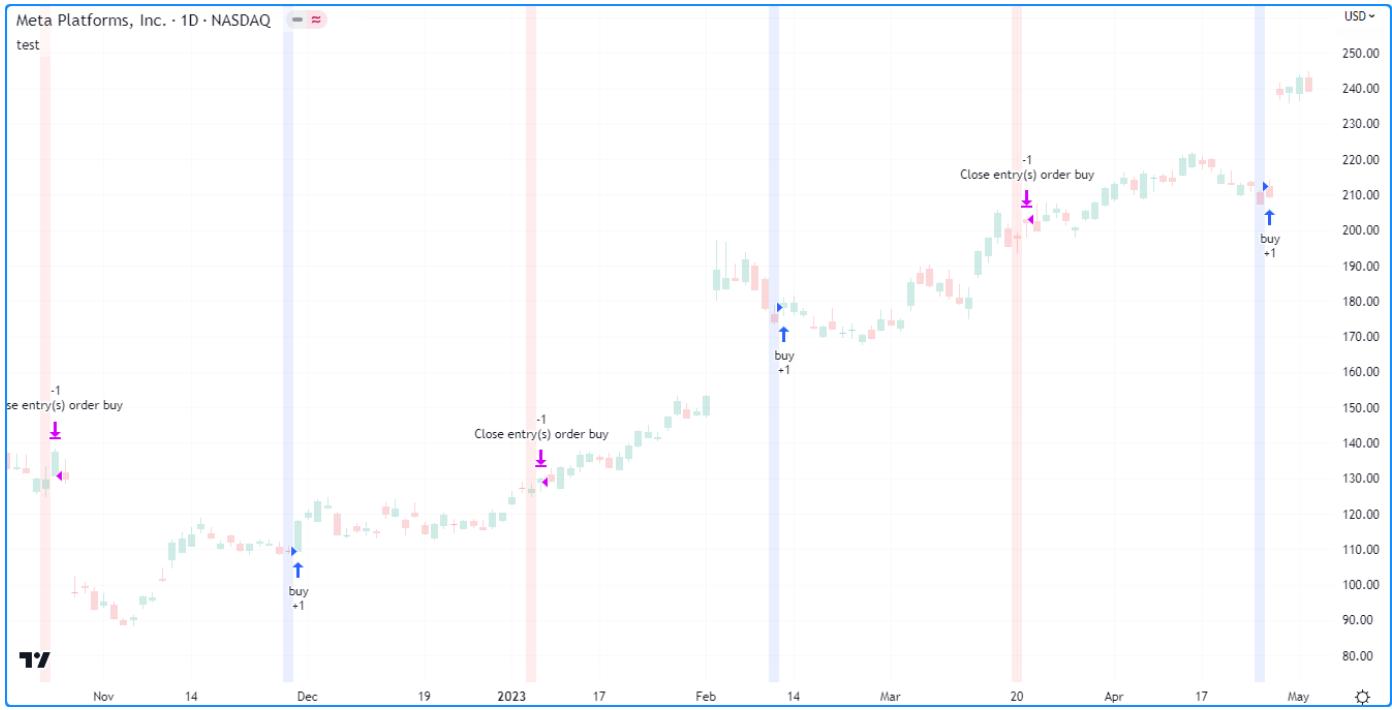
// Visualize the movement of the trailing stop.
plot(trailingStop, "Trailing Stop")

```

## [strategy.close\(\)](#) 和 [strategy.close\\_all\(\)](#)

这些命令使用市价订单模拟退出头寸。这些函数在被调用时关闭交易，而不是以特定价格关闭交易。

下面的示例演示了一个简单的策略，该策略每 50 根柱线通过 [strategy.entry\(\)](#) 下一次“买入”订单，并在 25 根柱线后使用[strategy.close\(\)](#)执行市价订单平仓：



```
//@version=5
strategy("Close demo", "test", overlay = true)

//@variable Is `true` on every 50th bar.
buyCond = bar_index % 50 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by 50.
sellCond = bar_index % 25 == 0 and not buyCond

if buyCond
    strategy.entry("buy", strategy.long)
if sellCond
    strategy.close("buy")

bgcolor(buyCond ? color.new(color.blue, 90) : na)
bgcolor(sellCond ? color.new(color.red, 90) : na)
```

与大多数其他下单命令不同，`strategy.close()` 的参数引用现有的条目 ID 来关闭。如果指定的不存在，该命令将不会执行命令。如果持仓由具有相同 ID 的多个条目组成，该命令将同时退出所有条目。`id`

为了进行演示，以下脚本每 25 个柱线下一次“买入”订单。该脚本每 100 个柱关闭一次所有“买入”条目。我们在`strategy()` 声明语句中加入了允许策略模拟最多三个相同方向的订单：`pyramiding = 3`



```
//@version=5
strategy("Multiple close demo", "test", overlay = true, pyramiding = 3)

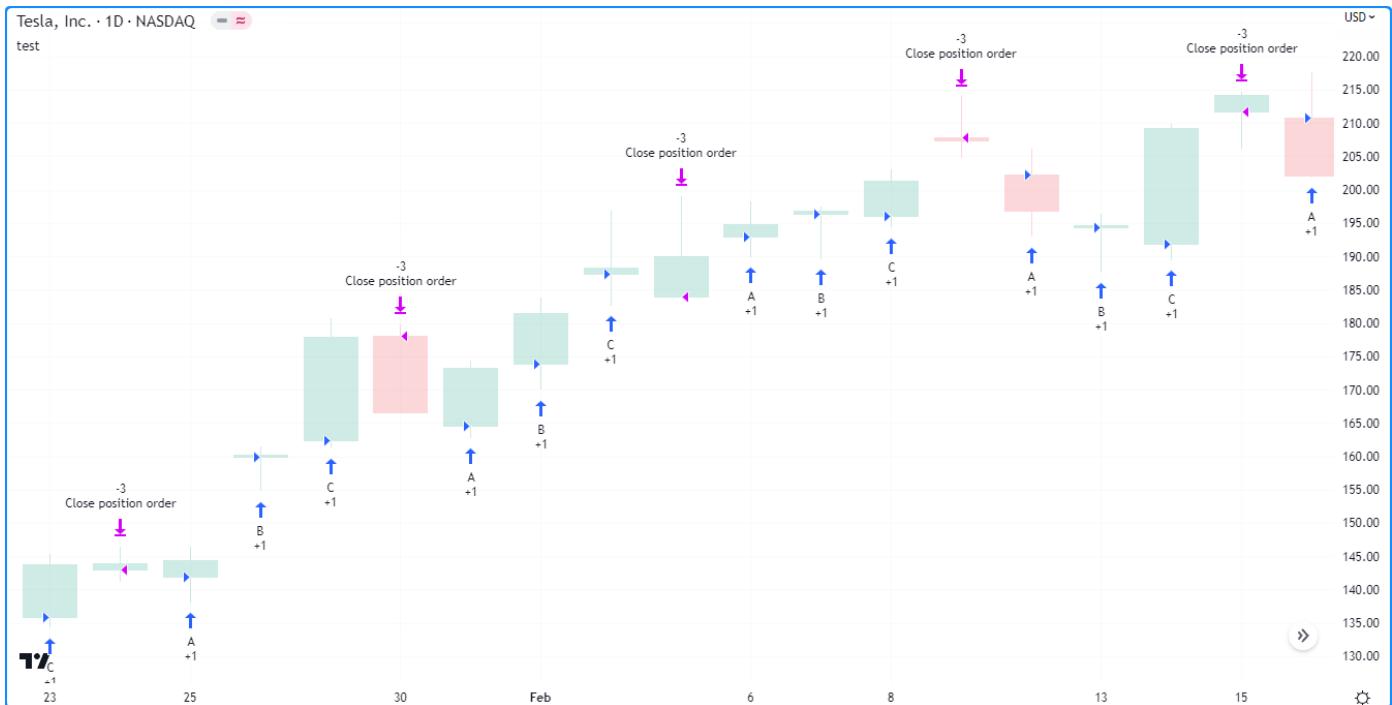
//@variable Is `true` on every 100th bar.
sellCond = bar_index % 100 == 0
//@variable Is `true` on every 25th bar except for those that are divisible by 100.
buyCond = bar_index % 25 == 0 and not sellCond

if buyCond
    strategy.entry("buy", strategy.long)
if sellCond
    strategy.close("buy")

bgcolor(buyCond ? color.new(color.blue, 90) : na)
bgcolor(sellCond ? color.new(color.red, 90) : na)
```

对于脚本具有多个具有不同 ID 的条目的情况，[strategy.close\\_all\(\)](#) 命令会派上用场，因为它会关闭所有条目，无论其 ID 为何。

下面的脚本根据未平仓交易的数量依次放置“A”、“B”和“C”挂单，然后用单个市价订单平仓所有订单：



```
//@version=5
strategy("Close multiple ID demo", "test", overlay = true, pyramiding = 3)

switch strategy.opentrades
    0 => strategy.entry("A", strategy.long)
    1 => strategy.entry("B", strategy.long)
    2 => strategy.entry("C", strategy.long)
    3 => strategy.close_all()
```

### [strategy.cancel\(\)](#) 和 [strategy.cancel\\_all\(\)](#) .

这些命令允许策略取消挂单，即当使用或参数时由[strategy.exit\(\)](#) 或[strategy.order\(\)](#) 或[strategy.entry\(\)](#)生成的挂单。`limit`stop`

以下策略模拟 100 个柱前收盘价下方 500 个点的“买入”限价单，然后取消下一个柱的订单。该脚本在处绘制一条水平线，并将 `limitPrice` 背景颜色设置为绿色和橙色，以分别指示何时下达和取消限价单。正如我们所看到的，一旦市场价格越过，什么也没有发生，`limitPrice` 因为该策略已经取消了订单：



```

//@version=5
strategy("Cancel demo", "test", overlay = true)

//@variable Draws a horizontal line at the `limit` price of the "buy" order.
var line limitLine = na

//@variable Returns `color.green` when the strategy places the "buy" order,
//`color.orange` when it cancels the order.
color bgColor = na

if last_bar_index - bar_index == 100
    float limitPrice = close - syminfo.mintick * 500
    strategy.entry("buy", strategy.long, limit = limitPrice)
    limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend =
extend.right)
    bgColor := color.new(color.green, 50)

if last_bar_index - bar_index == 99
    strategy.cancel("buy")
    bgColor := color.new(color.orange, 50)

bgcolor(bgColor)

```

与 [strategy.close\(\)](#) 一样，[strategy.cancel\(\)](#) `id` 的参数指的是现有条目的ID。如果参数引用不存在的 ID，则此命令将不执行任何操作。当有多个相同ID的挂单时，该命令将一次性取消所有挂单。`id`

在此示例中，我们修改了之前的脚本，在从 100 根柱之前开始的三个连续柱上放置“买入”限价单。该策略在距最近柱线 97 根柱线后取消所有柱线 `bar_index`，导致当价格穿过任何线时它不执行任何操作：



```
//@version=5
strategy("Multiple cancel demo", "test", overlay = true, pyramiding = 3)

//@variable Draws a horizontal line at the `limit` price of the "buy" order.
var line limitLine = na

//@variable Returns `color.green` when the strategy places the "buy" order,
//`color.orange` when it cancels the order.
color bgColor = na

if last_bar_index - bar_index <= 100 and last_bar_index - bar_index >= 98
    float limitPrice = close - syminfo.mintick * 500
    strategy.entry("buy", strategy.long, limit = limitPrice)
    limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice, extend = extend.right)
    bgColor := color.new(color.green, 50)

if last_bar_index - bar_index == 97
    strategy.cancel("buy")
    bgColor := color.new(color.orange, 50)

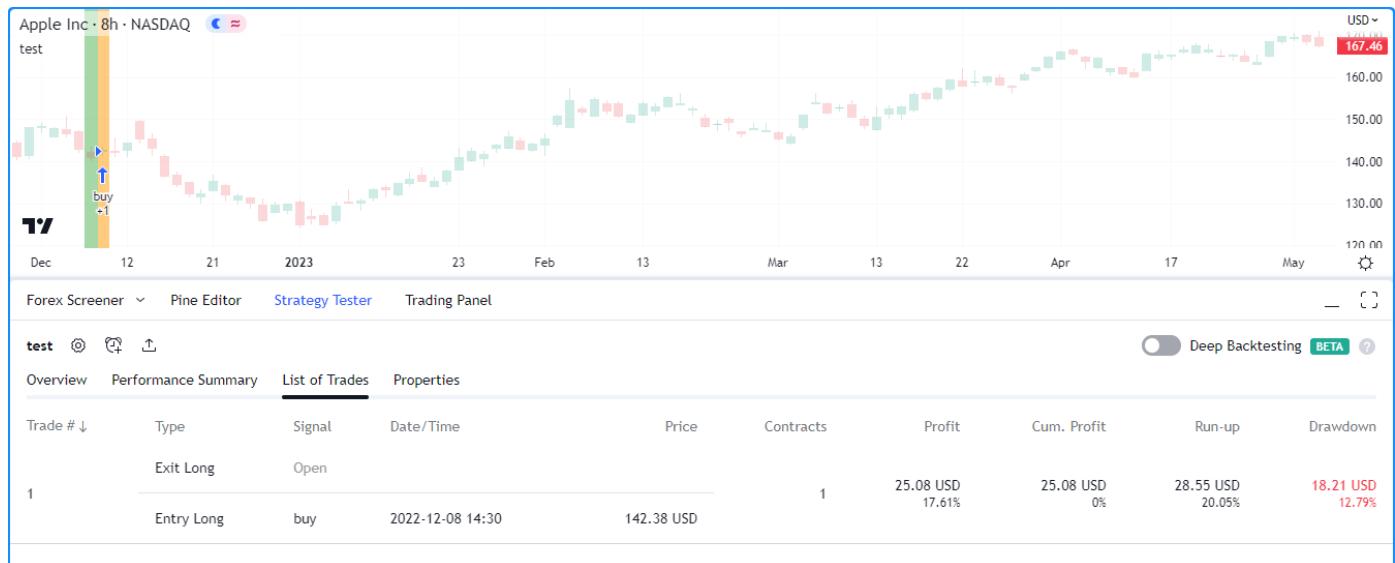
bgcolor(bgColor)
```

- 注意：

我们在脚本的声明语句中添加了允许执行三个 `strategy.entry()` 订单的功能。或者，脚本可以使用 `strategy.order()` 实现相同的输出，因为它对设置不敏感。`pyramiding = 3``pyramiding`

需要注意的是，`strategy.cancel()` 和 `strategy.cancel_all()` 都不能取消市价订单，因为策略会在下一个价格变动时立即执行它们。策略在成交后无法取消订单。要平仓，请使用 `strategy.close()` 或 `strategy.close_all()`。

此示例模拟 100 个柱前的“买入”市价单，然后尝试取消下一个柱上的所有挂单。由于策略已经填写了“买入”订单，因此在这种情况下，[strategy.cancel\\_all\(\)](#)命令不执行任何操作，因为没有待取消的订单：



```
//@version=5
strategy("Cancel market demo", "test", overlay = true)

//@variable Returns `color.green` when the strategy places the "buy" order,
//`color.orange` when it tries to cancel.
color bgColor = na

if last_bar_index - bar_index == 100
    strategy.entry("buy", strategy.long)
    bgColor := color.new(color.green, 50)

if last_bar_index - bar_index == 99
    strategy.cancel_all()
    bgColor := color.new(color.orange, 50)

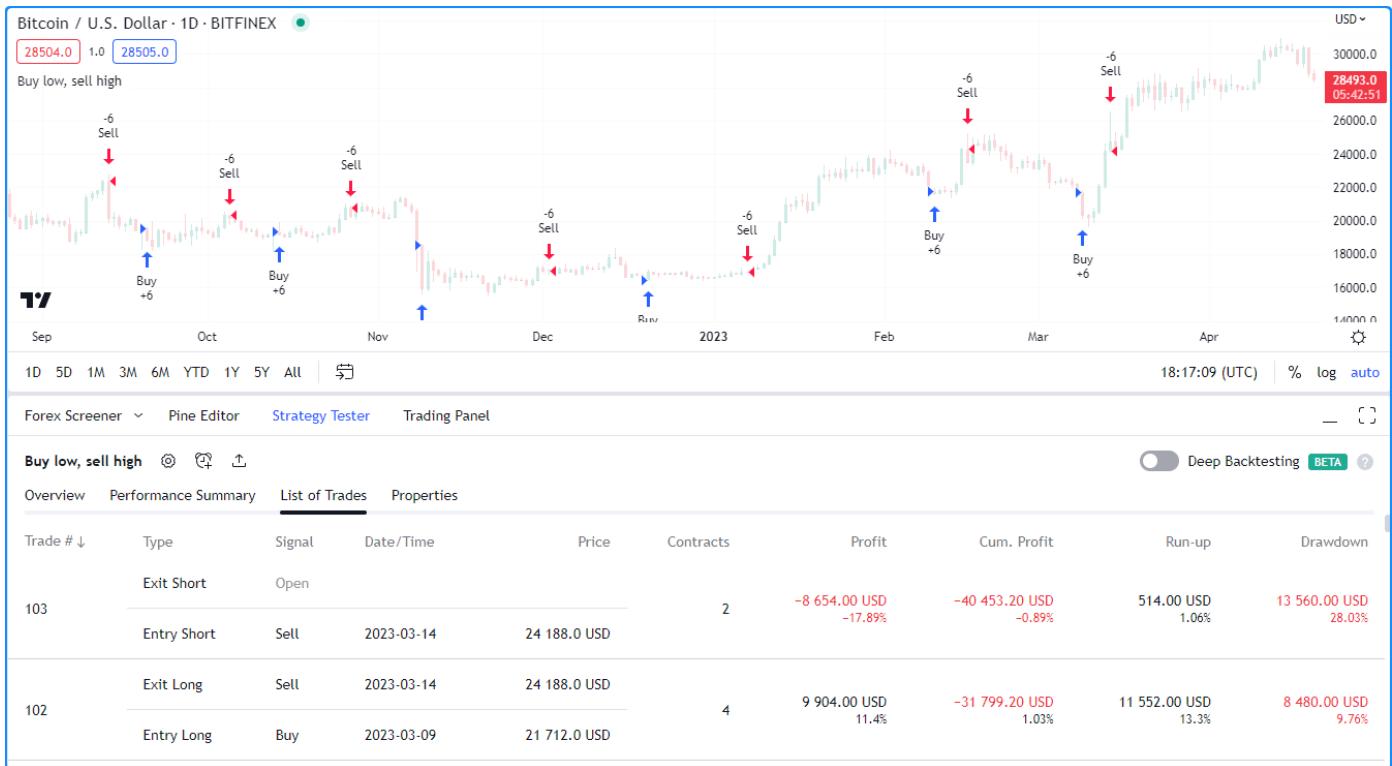
bgcolor(bgColor)
```

## 头寸规模

Pine Script™ 策略有两种控制模拟交易规模的方法：

- 使用[strategy\(\)](#) `default_qty_type` 函数中的和 `default_qty_value` 参数为所有订单设置默认的固定数量类型和值，该函数还会在脚本设置的“属性”选项卡中设置默认值。
- 调用[strategy.entry\(\)](#) `qty` 时指定参数。当用户向函数提供此参数时，脚本会忽略策略的默认数量值和类型。

以下示例模拟 `longAmount` 每当 `low` 价格等于价值时的“买入”订单规模，以及当 `high` 价格等于价值时的 `lowest` “卖出”订单规模： `shortAmount``high``highest`



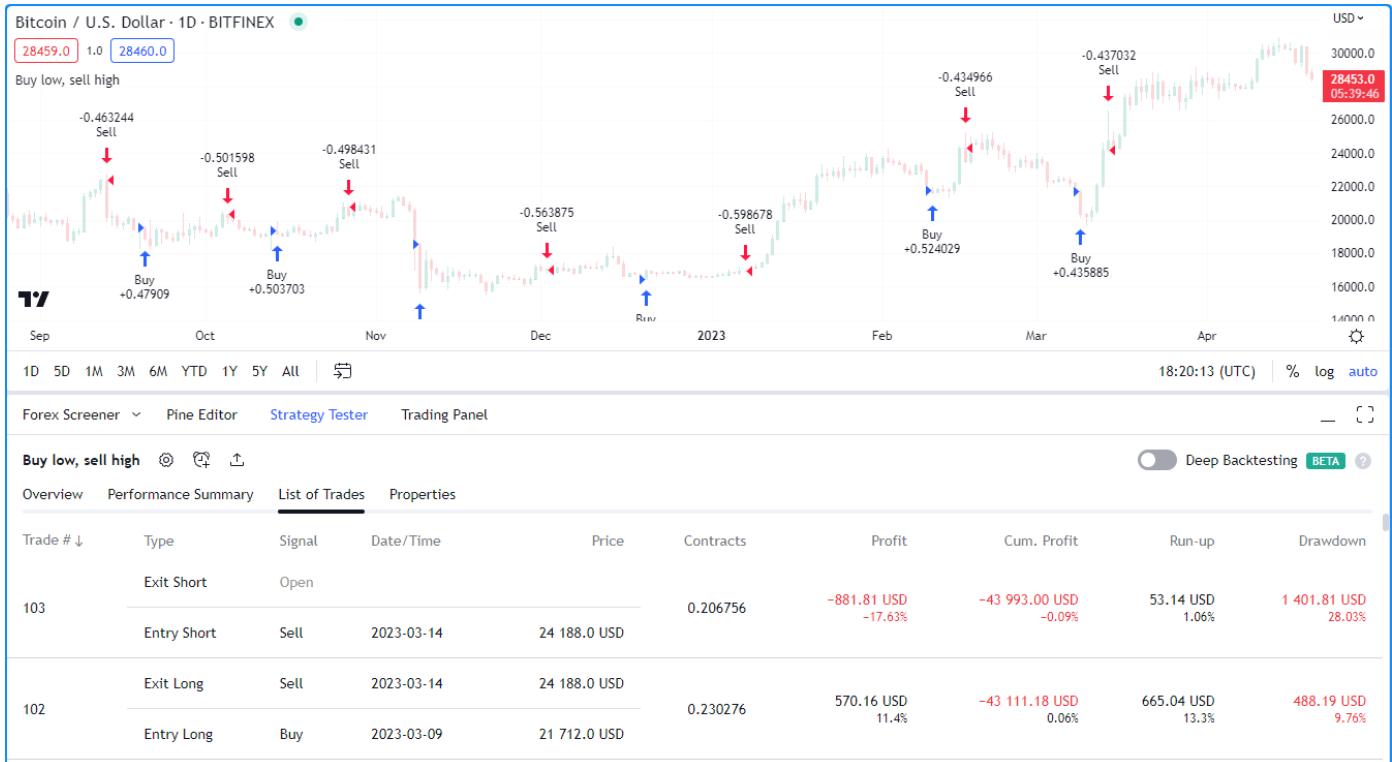
```
//@version=5
strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash,
default_qty_value = 5000)

int    length      = input.int(20, "Length")
float longAmount = input.float(4.0, "Long Amount")
float shortAmount = input.float(2.0, "Short Amount")

float highest = ta.highest(length)
float lowest  = ta.lowest(length)

switch
    low == lowest  => strategy.entry("Buy", strategy.long, longAmount)
    high == highest => strategy.entry("Sell", strategy.short, shortAmount)
```

请注意，在上面的示例中，尽管我们在声明语句中指定了 `default_qty_type` 和 `default_qty_value` 参数，但脚本并未将这些默认值用于模拟订单。相反，它将它们的大小设置为单位 `longAmount` 和 `shortAmount` 单位。如果我们希望脚本使用默认类型和值，则必须 `qty` 从 [strategy.entry\(\)](#) 调用中删除该规范：



```
//@version=5
strategy("Buy low, sell high", overlay = true, default_qty_type = strategy.cash,
default_qty_value = 5000)

int length = input.int(20, "Length")

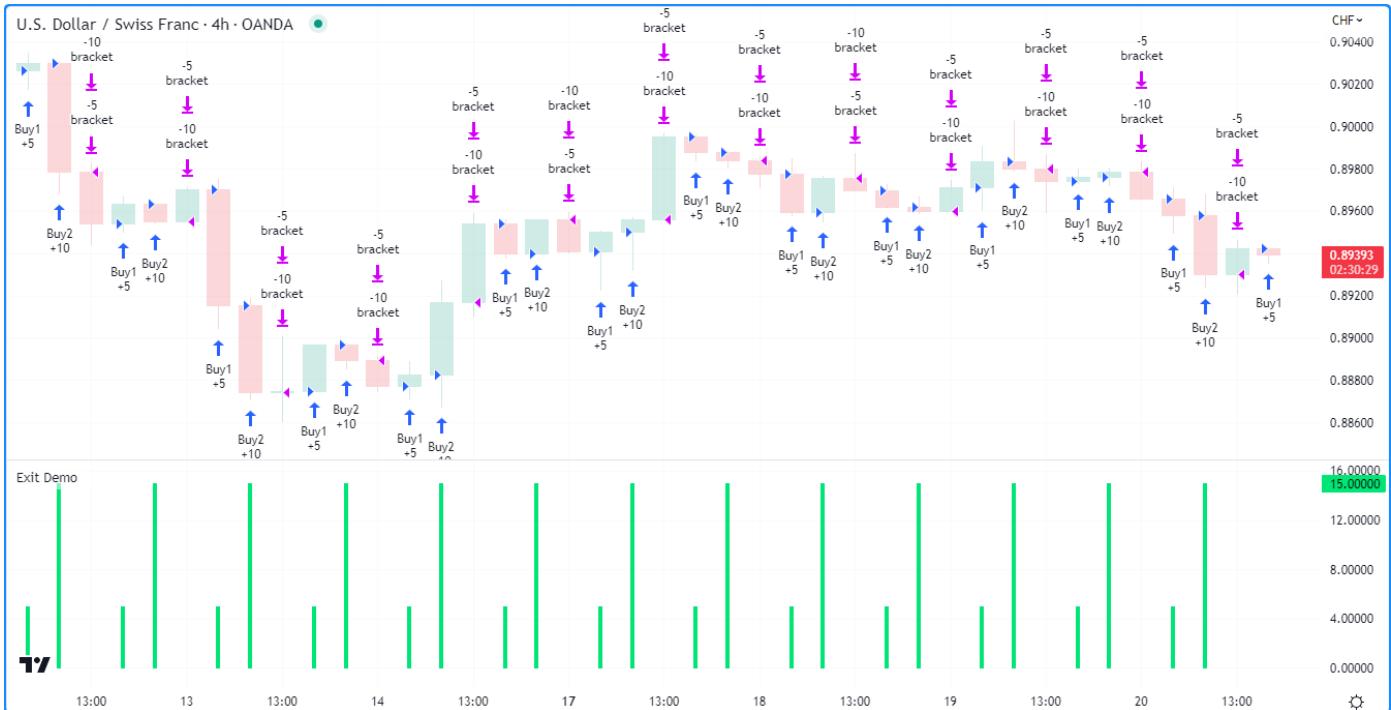
float highest = ta.highest(length)
float lowest = ta.lowest(length)

switch
    low == lowest => strategy.entry("Buy", strategy.long)
    high == highest => strategy.entry("Sell", strategy.short)
```

## 平仓市场

尽管可以模拟[策略测试器](#)模块交易列表选项卡中显示的特定挂单的退出，但所有订单均根据 FIFO（先进先出）规则链接。如果用户未指定[strategy.exit\(\)](#)调用的参数，策略将从开仓的第一个挂单开始退出未平仓市场头寸。[from\\_entry](#)

以下示例按顺序模拟两个订单：以最后 100 个柱的市场价格“Buy1”，一旦头寸大小与“Buy1”的大小匹配，则“Buy2”。该策略仅在 15 个单位时下达退出指令[positionSize](#)。该脚本不[from\\_entry](#)向[strategy.exit\(\)](#)命令提供参数，因此该策略每次调用该函数时都会为所有未平仓头寸下达退出指令，从第一个指令开始。它将在单独的窗格中绘制[positionSize](#)以供视觉参考：



```
//@version=5
strategy("Exit Demo", pyramiding = 2)

float positionSize = strategy.position_size

if positionSize == 0 and last_bar_index - bar_index <= 100
    strategy.entry("Buy1", strategy.long, 5)
else if positionSize == 5
    strategy.entry("Buy2", strategy.long, 10)
else if positionSize == 15
    strategy.exit("bracket", loss = 10, profit = 10)

plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4,
plot.style_histogram)
```

- 注意：

我们在脚本的声明语句中包含了它，以允许它模拟同一方向的两个连续订单。`pyramiding = 2`

假设我们想在“Buy1”之前退出“Buy2”。让我们看看如果我们指示策略在满足两个订单时在“Buy1”之前关闭“Buy2”会发生什么：

Exit Demo ⌂ ⌂ ⌂ Deep Backtesting BETA ?

Overview	Performance Summary	List of Trades	Properties						
Trade # ↓	Type	Signal	Date/Time	Price	Contracts	Profit	Cum. Profit	Run-up	Drawdown
100	Exit Long	Open			5	-0.00 CHF -0.03%	-0.16 CHF 0%	0.00 CHF 0.01%	0.00 CHF 0.08%
	Entry Long	Buy1	2023-04-20 17:00	0.89423 CHF					
99	Exit Long	bracket	2023-04-20 13:00	0.89300 CHF	5	-0.01 CHF -0.31%	-0.15 CHF 0%	0.01 CHF 0.11%	0.02 CHF 0.39%
	Entry Long	Buy2	2023-04-20 09:00	0.89582 CHF					
98	Exit Long	Close entry(s) order Buy2	2023-04-20 13:00	0.89300 CHF	5	-0.01 CHF -0.31%	-0.14 CHF 0%	0.01 CHF 0.11%	0.02 CHF 0.39%
	Entry Long	Buy2	2023-04-20 09:00	0.89582 CHF					
97	Exit Long	Close entry(s) order Buy2	2023-04-20 13:00	0.89300 CHF	5	-0.02 CHF -0.4%	-0.13 CHF 0%	0.00 CHF 0.06%	0.02 CHF 0.48%
	Entry Long	Buy1	2023-04-20 05:00	0.89660 CHF					

```
//@version=5
strategy("Exit Demo", pyramiding = 2)

float positionSize = strategy.position_size

if positionSize == 0 and last_bar_index - bar_index <= 100
    strategy.entry("Buy1", strategy.long, 5)
else if positionSize == 5
    strategy.entry("Buy2", strategy.long, 10)
else if positionSize == 15
    strategy.close("Buy2")
    strategy.exit("bracket", "Buy1", loss = 10, profit = 10)

plot(positionSize == 0 ? na : positionSize, "Position Size", color.lime, 4,
plot.style_histogram)
```

正如我们在策略测试器的“交易列表”选项卡中看到的，它不是使用[Strategy.close\(\)](#)关闭“Buy2”头寸，而是首先关闭“Buy1”的数量，这是平仓订单数量的一半，然后关闭一半的“Buy2”头寸，因为经纪商模拟器默认遵循 FIFO 规则。用户可以通过在[strategy\(\)](#)函数中指定来更改此行为。`close_entries_rule = "ANY"`

## OCA 团体

一取消全部 (OCA) 组允许策略在执行下单命令时全部或部分取消其他订单，包括[strategy.entry\(\)](#) 和 [strategy.order\(\)](#)，具有相同的 `oca_name`，具体取决于 `oca_type` 用户在函数调用中提供。

### [strategy.oca.cancel](#)

`strategy.oca.cancel` OCA 类型在订单组中的订单执行或部分执行后 取消所有具有相同订单的订单 `oca_name`

例如，以下策略在 `ma1` 交叉时执行订单 `ma2`。当 `strategy.position_size` 为 0 时，它会在柱的 `high` 和 `low` 处放置多头和空头止损订单。`low` 否则，它调用 `strategy.close_all()` 以市价订单平仓所有未平仓头寸。根据价格行为，策略可能会在发出平仓订单之前填写两个订单。此外，如果经纪商模拟器的柱内假设支持它，则两个订单可能在同一柱上填充。在这种情况下，`strategy.close_all()` 命令不会执行任何操作，因为在执行完两个订单之前，脚本无法调用该操作：



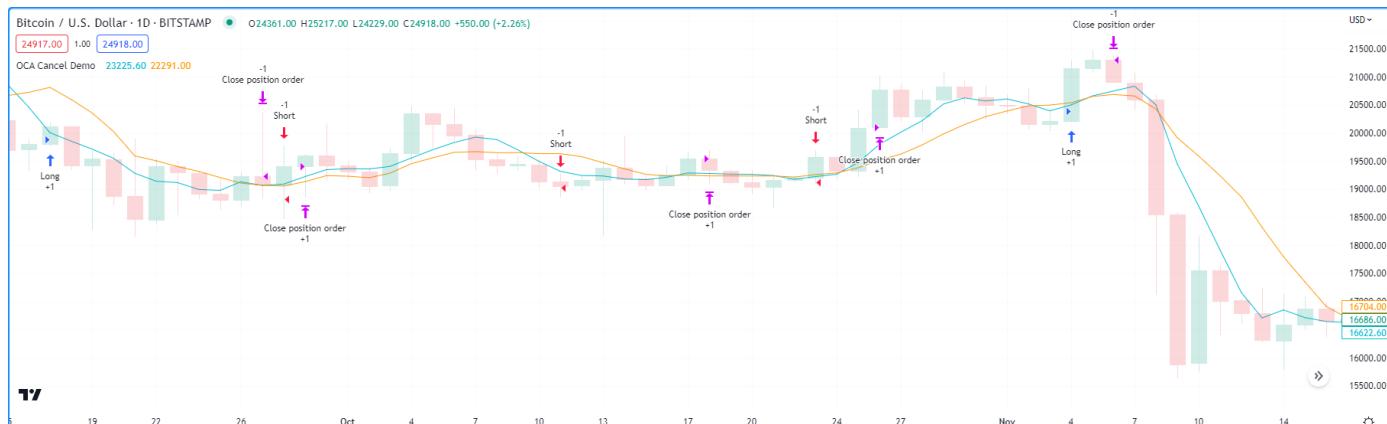
```
//@version=5
strategy("OCA Cancel Demo", overlay=true)

float ma1 = ta.sma(close, 5)
float ma2 = ta.sma(close, 9)

if ta.cross(ma1, ma2)
    if strategy.position_size == 0
        strategy.order("Long", strategy.long, stop = high)
        strategy.order("Short", strategy.short, stop = low)
    else
        strategy.close_all()

plot(ma1, "Fast MA", color.aqua)
plot(ma2, "Slow MA", color.orange)
```

为了消除策略在平仓订单之前填写多头和空头订单的情况，我们可以指示它在执行另一订单后取消另一订单。在此示例中，我们将 `Strategy.order()` `oca_name` 命令的 `和它们的分别设置为“Entry”` 和： `oca_type` `strategy.oca.cancel`



```

//@version=5
strategy("OCA Cancel Demo", overlay=true)

float ma1 = ta.sma(close, 5)
float ma2 = ta.sma(close, 9)

if ta.cross(ma1, ma2)
    if strategy.position_size == 0
        strategy.order("Long", strategy.long, stop = high, oca_name = "Entry",
oca_type = strategy.oca.cancel)
        strategy.order("Short", strategy.short, stop = low, oca_name = "Entry",
oca_type = strategy.oca.cancel)
    else
        strategy.close_all()

plot(ma1, "Fast MA", color.aqua)
plot(ma2, "Slow MA", color.orange)

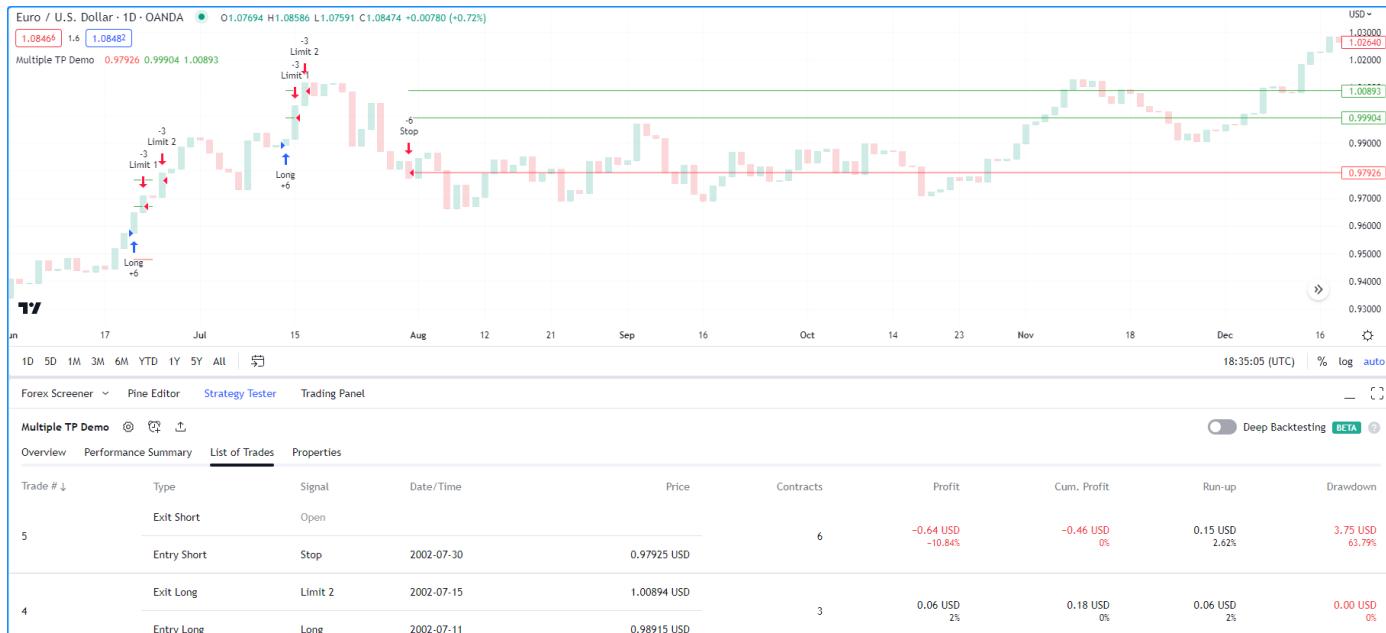
```

## strategy.oca.reduce

Strategy.oca.reduce OCA 类型不会取消订单。相反，它会根据已平仓合约/股票/手数/单位的数量减少每次新成交时相同订单的规模 `oca_name`，这对于退出策略特别有用。

以下示例演示了仅多头退出策略的尝试，该策略为每个新入场生成一个止损订单和两个止盈订单。在两条移动平均线交叉时，它使用[Strategy.entry\(\)](#)模拟 6 个单位的“多头”挂单，然后使用[Strategy.order\(\)](#) `qty` 模拟 6、3 和 3 个单位的止损/限价单。和分别是价格。`stop``limit1``limit2`

将策略添加到图表后，我们发现它没有按预期工作。该脚本的问题是，与 [strategy.exit\(\)](#) 不同，[strategy.order\(\)](#) 默认不属于 OCA 组。由于我们没有明确地将订单分配给 OCA 组，因此该策略在执行订单时不会取消或减少订单，这意味着可以交易比未平仓头寸更大的数量并反转方向：



```

//@version=5
strategy("Multiple TP Demo", overlay = true)

var float stop    = na
var float limit1 = na
var float limit2 = na

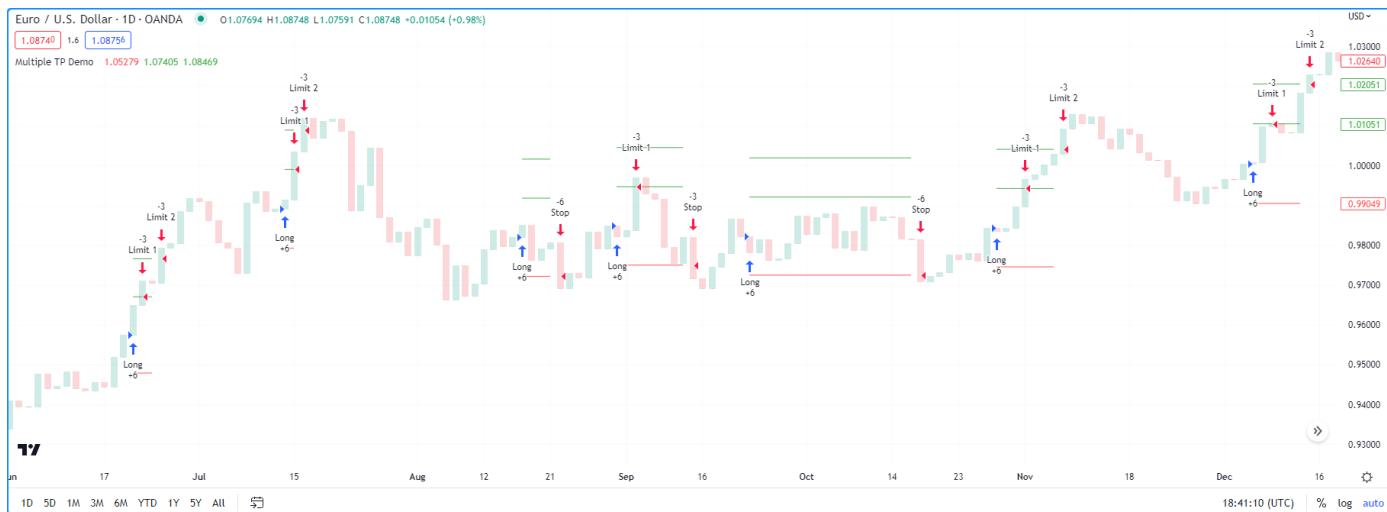
bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9))
if longCondition and strategy.position_size == 0
    stop    := close * 0.99
    limit1 := close * 1.01
    limit2 := close * 1.02
    strategy.entry("Long", strategy.long, 6)
    strategy.order("Stop", strategy.short, stop = stop, qty = 6)
    strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3)
    strategy.order("Limit 2", strategy.short, limit = limit2, qty = 3)

bool showPlot = strategy.position_size != 0
plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr)
plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr)
plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

```

为了使我们的策略按预期发挥作用，我们必须指示其减少其他止损/止盈订单的单位数量，以便它们不超过剩余未平仓头寸的规模。

在下面的示例中，我们将 `oca_name` 退出策略中的每个订单设置为“Bracket”，并设置 `oca_type` 为 `strategy.oca.reduce`。这些设置告诉策略在执行其中一个订单时，`qty` 将“Bracket”组中的订单值减少为已 `qty` 成交，以防止其交易过多单位并导致逆转：



```

//@version=5
strategy("Multiple TP Demo", overlay = true)

var float stop    = na
var float limit1 = na
var float limit2 = na

```

```

bool longCondition = ta.crossover(ta.sma(close, 5), ta.sma(close, 9))
if longCondition and strategy.position_size == 0
    stop := close * 0.99
    limit1 := close * 1.01
    limit2 := close * 1.02
    strategy.entry("Long", strategy.long, 6)
    strategy.order("Stop", strategy.short, stop = stop, qty = 6, oca_name = "Bracket",
oca_type = strategy.oca.reduce)
    strategy.order("Limit 1", strategy.short, limit = limit1, qty = 3, oca_name =
"Bracket", oca_type = strategy.oca.reduce)
    strategy.order("Limit 2", strategy.short, limit = limit2, qty = 6, oca_name =
"Bracket", oca_type = strategy.oca.reduce)

bool showPlot = strategy.position_size != 0
plot(showPlot ? stop : na, "Stop", color.red, style = plot.style_linebr)
plot(showPlot ? limit1 : na, "Limit 1", color.green, style = plot.style_linebr)
plot(showPlot ? limit2 : na, "Limit 2", color.green, style = plot.style_linebr)

```

- 注意：

我们将 `qty` “Limit 2”订单的值从 3 更改为 6，因为该策略在执行“Limit 1”订单时会将其价值减少 3。保留 `qty` 值 3 会导致其降至 0，并且在填写第一个限价订单后永远不会填写。

## strategy.oca.none

`strategy.oca.none` OCA 类型指定订单独立于任何 OCA 组执行。[该值是](#)`strategy.order()` 和`strategy.entry()` 下单命令 `oca_type` 的默认值。

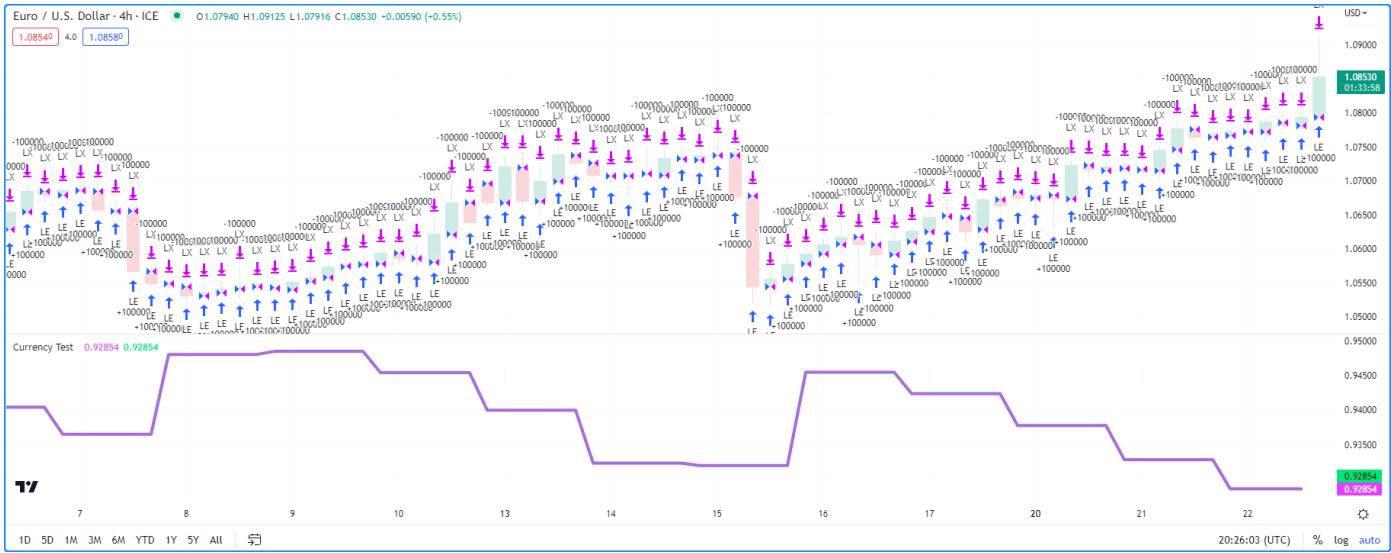
### 笔记

如果两个下单命令具有相同 `oca_name` 但不同的 `oca_type` 值，则该策略认为它们来自两个不同的组。即，OCA 组不能组合 [strategy.oca.cancel](#)、[strategy.oca.reduce](#) 和[strategy.oca.none](#) OCA 类型。

## 货币

Pine Script™ 策略可以使用与其计算工具不同的基础货币。用户可以通过在 `strategy()` 函数中包含一个 `currency.*` 变量作为参数来指定模拟账户的基础货币，这将更改脚本的 `strategy.account_currency` 值。策略的默认值为，这意味着脚本使用图表上工具的基础货币。`currency` `currency` `currency.NONE`

当策略脚本使用指定的基础货币时，它将模拟利润乘以前一个交易日的 FX\_IDC 汇率。例如，下面的策略下一个标准手（100,000 单位）的挂单，在最后 500 个图表柱上每个都设定利润目标和 1 点止损，然后将净利润与反向日收盘价一起绘制出来。符号在单独的窗格中。我们已将基础货币设置为 `currency.EUR`。当我们将其添加到 `FX_IDC:EURUSD` 时，两个图对齐，确认该策略使用该交易品种前一天的汇率进行计算：



```
//@version=5
strategy("Currency Test", currency = currency.EUR)

if last_bar_index - bar_index < 500
    strategy.entry("LE", strategy.long, 100000)
    strategy.exit("LX", "LE", profit = 1, loss = 1)
plot(math.abs(ta.change(strategy.netprofit)), "1 Point profit", color = color.fuchsia,
linewidth = 4)
plot(request.security(syminfo.tickerid, "D", 1 / close)[1], "Previous day's inverted
price", color = color.lime)
```

- 注意：

当交易时间范围高于每日时，该策略将使用柱线收盘前一个交易日的收盘价来计算历史柱线的交叉汇率。例如，在每周时间范围内，交叉汇率将基于上周四的收盘价，但该策略仍将使用实时柱线的每日收盘价。

## 改变计算行为

策略在图表中所有可用的历史柱上执行，然后在新数据可用时自动继续实时计算。默认情况下，策略脚本仅对每个已确认的柱计算一次。我们可以通过更改[策略\(\)](#)函数的参数 或单击脚本“属性”选项卡的“重新计算”部分中的复选框来更改此行为。

### calc\_on\_every\_tick

`calc_on_every_tick` 是一个可选设置，用于控制实时数据的计算行为。启用此参数后，脚本将在每个新的价格变动时重新计算其值。默认情况下，其值为 `false`，这意味着脚本仅在确认柱后执行计算。

启用此计算行为在前向测试时可能特别有用，因为它有利于精细的实时策略模拟。然而，值得注意的是，这种行为会在实时和历史模拟之间引入数据差异，因为历史柱不包含报价信息。用户应谨慎对待此设置，因为数据差异可能会导致策略重新绘制其历史记录。

```

close`每次达到输入的`highest`或`lowest`值时，以下脚本将模拟一个新订单。由于在策略声明中启用，因此脚本将在编译后在每个新的实时价格变动上模拟新订单：`lowest``length``calc_on_every_tick`

Pine Script™
Copied//@version=5
strategy("Donchian Channel Break", overlay = true, calc_on_every_tick = true,
pyramiding = 20)

int length = input.int(15, "Length")

float highest = ta.highest(close, length)
float lowest = ta.lowest(close, length)

if close == highest
    strategy.entry("Buy", strategy.long)
if close == lowest
    strategy.entry("Sell", strategy.short)

//@variable The starting time for real-time bars.
var realTimeStart = timenow

// Color the background of real-time bars.
bgcolor(time_close >= realTimeStart ? color.new(color.orange, 80) : na)

plot(highest, "Highest", color = color.lime)
plot(lowest, "Lowest", color = color.red)

```

- 注意：

该脚本在其声明中使用 `pyramiding` 值 20，这允许策略最多模拟同一方向的 20 笔交易。为了直观地划分策略将哪些柱形处理为实时柱形，脚本为自上次编译以来所有柱形的背景 [着色](#)。

将脚本应用到图表并让它计算一些实时柱后，我们可能会看到如下输出：



该脚本在条件有效的每个新的实时报价上放置“买入”订单，从而导致每条柱有多个订单。然而，不熟悉此行为的用户在重新编译脚本后看到策略的输出发生变化可能会感到惊讶，因为之前执行实时计算的柱现在是历史柱，它们不保存报价信息：

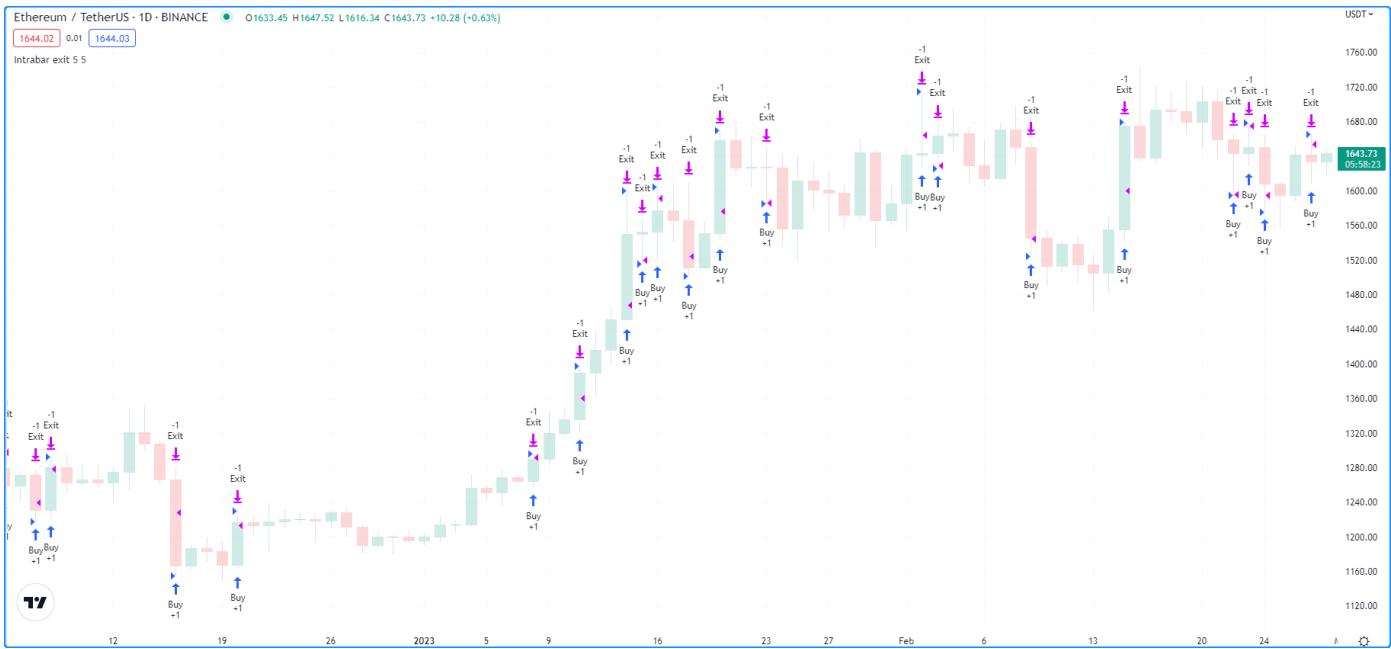


## [calc\\_on\\_order\\_fills](#)

可选 `calc_on_order_fills` 设置允许在模拟订单执行后立即重新计算策略，这允许脚本使用更细粒度的价格并下额外的订单，而无需等待柱线被确认。

启用此设置可以为脚本提供附加数据，否则这些数据在柱关闭后才可用，例如未确认柱上模拟头寸的当前平均价格。

下面的示例显示了一个使用启用声明的简单策略，当 `strategy.position_size` 为 0 时 `calc_on_order_fills` 模拟“买入”订单。该脚本使用 `strategy.position_avg_price` 来计算 a 和并在价格交叉时模拟“退出”订单，无论价格如何该栏是否已确认。因此，一旦触发退出，策略就会重新计算并下达新的入场订单，因为 `strategy.position_size` 再次等于 0。策略在退出发生后下订单，并在退出后的下一个价格变动时执行该订单。退出，这将是柱的 OHLC 值之一，具体取决于模拟的柱内运动： `stopLoss``takeProfit`



```
//@version=5
strategy("Intrabar exit", overlay = true, calc_on_order_fills = true)

float stopSize    = input.float(5.0, "SL %", minval = 0.0) / 100.0
float profitSize = input.float(5.0, "TP %", minval = 0.0) / 100.0

if strategy.position_size == 0.0
    strategy.entry("Buy", strategy.long)

float stopLoss    = strategy.position_avg_price * (1.0 - stopSize)
float takeProfit = strategy.position_avg_price * (1.0 + profitSize)

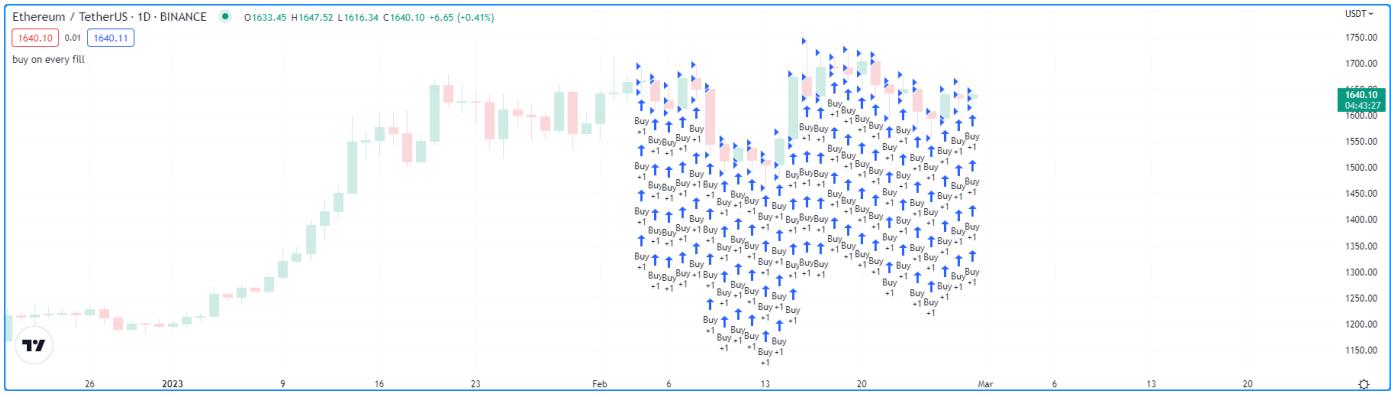
strategy.exit("Exit", stop = stopLoss, limit = takeProfit)
```

- 注意：

关闭后 `calc_on_order_fills`，相同的策略在触发退出订单后将仅进入一根柱线。首先，会发生中线退出，但不会出现进场指令。然后，该策略将在柱线收盘后模拟入场订单，并将在下一个价格变动（即下一个柱线开盘）时填充该订单。

值得注意的是，启用 `calc_on_order_fills` 可能会产生不切实际的策略结果，因为经纪商模拟器可能会假设实时交易时不可能的订单价格。用户必须谨慎对待此设置，并仔细考虑脚本中的逻辑。

当脚本加载到图表上时，以下示例模拟从 `last_bar_index` 开始的 25 条柱窗口中每次新订单成交和柱确认后的“买入”订单。启用该设置后，该策略会模拟每个柱形图的四个条目，因为模拟器认为每个柱形图有四个价格变动（开盘价、最高价、最低价、收盘价），这是不切实际的行为，因为订单通常不可能在准确的位置填写柱线的高点或低点：



```
//@version=5
strategy("buy on every fill", overlay = true, calc_on_order_fills = true, pyramiding = 100)

if last_bar_index - bar_index <= 25
    strategy.entry("Buy", strategy.long)
```

## process\_orders\_on\_close

默认策略行为模拟每个柱线收盘时的订单，这意味着最早执行订单并执行策略计算和警报的机会是在下一个柱线开盘时。交易者可以通过启用该设置来更改此行为，以使用每个柱的收盘价来处理策略 `process_orders_on_close`。

在回测手动策略（交易者在柱线收盘前退出头寸）或非 24x7 市场中的算法交易者设置盘后交易功能以便收盘后发送的警报仍有希望在后续交易之前成交的情况下，此行为非常有用。天。

- 注意：

重要的是要意识到，`process_orders_on_close` 在实时交易环境中使用策略可能会导致重新绘制策略，因为收盘时仍然会出现柱线收盘警报，并且订单可能要等到下一个市场开盘时才会填写。

## 模拟交易成本

为了使策略绩效报告包含相关的、有意义的数据，交易者应努力在策略结果中考虑潜在的现实成本。忽视这一点可能会影响交易者对策略表现产生不切实际的看法，并损害测试结果的可信度。如果不对其交易相关的潜在成本进行建模，交易者可能会高估策略的历史盈利能力，从而可能导致实时交易中的决策不理想。Pine Script™ 策略包括用于在绩效结果中模拟交易成本的输入和参数。

## 委员会

佣金是指经纪商/交易所在执行交易时收取的费用。根据经纪商/交易所的不同，有些经纪商/交易所可能会对每笔交易或合约/股票/手数/单位收取固定费用，而另一些经纪商/交易所可能会收取总交易价值的一定百分比。用户可以通过在`strategy()` `commission_type` 函数中包含 `参数` 或在策略设置的“属性”选项卡中设置“佣金”输入来设置其策略的佣金属性。`commission_value`

以下脚本是一个简单的策略，当 `close` 等于 `highest` 的值时，模拟 2% 权益的“多头”头寸 `length`，并在等于该值时平仓 `lowest`：



```
//@version=5
strategy("Commission Demo", overlay=true, default_qty_value = 2, default_qty_type =
strategy.percent_of_equity)

length = input.int(10, "Length")

float highest = ta.highest(close, length)
float lowest = ta.lowest(close, length)

switch close
    highest => strategy.entry("Long", strategy.long)
    lowest => strategy.close("Long")

plot(highest, color = color.new(color.lime, 50))
plot(lowest, color = color.new(color.red, 50))
```

在检查策略测试器中的结果后，我们发现该策略在测试范围内的净值正增长为 17.61%。然而，回测结果并未考虑经纪商/交易所可能收取的费用。让我们看看当我们在策略模拟中的每笔交易中加入少量佣金时，这些结果会发什么。在这个例子中，我们在 `strategy()` 声明中包含了 `and`，这意味着它将模拟所有执行订单的 1% 佣金： `commission_type = strategy.commission.percent``commission_value = 1`



```
//@version=5
strategy(
    "Commission Demo", overlay=true, default_qty_value = 2, default_qty_type =
strategy.percent_of_equity,
    commission_type = strategy.commission.percent, commission_value = 1
)

length = input.int(10, "Length")

float highest = ta.highest(close, length)
float lowest = ta.lowest(close, length)

switch close
    highest => strategy.entry("Long", strategy.long)
    lowest => strategy.close("Long")

plot(highest, color = color.new(color.lime, 50))
plot(lowest, color = color.new(color.red, 50))
```

正如我们在上面的例子中看到的，在回测中应用 1% 的佣金后，该策略模拟净利润显著下降，仅为 1.42%，并且股票曲线波动更大，最大回撤也更高，这凸显了佣金模拟的影响策略的测试结果。

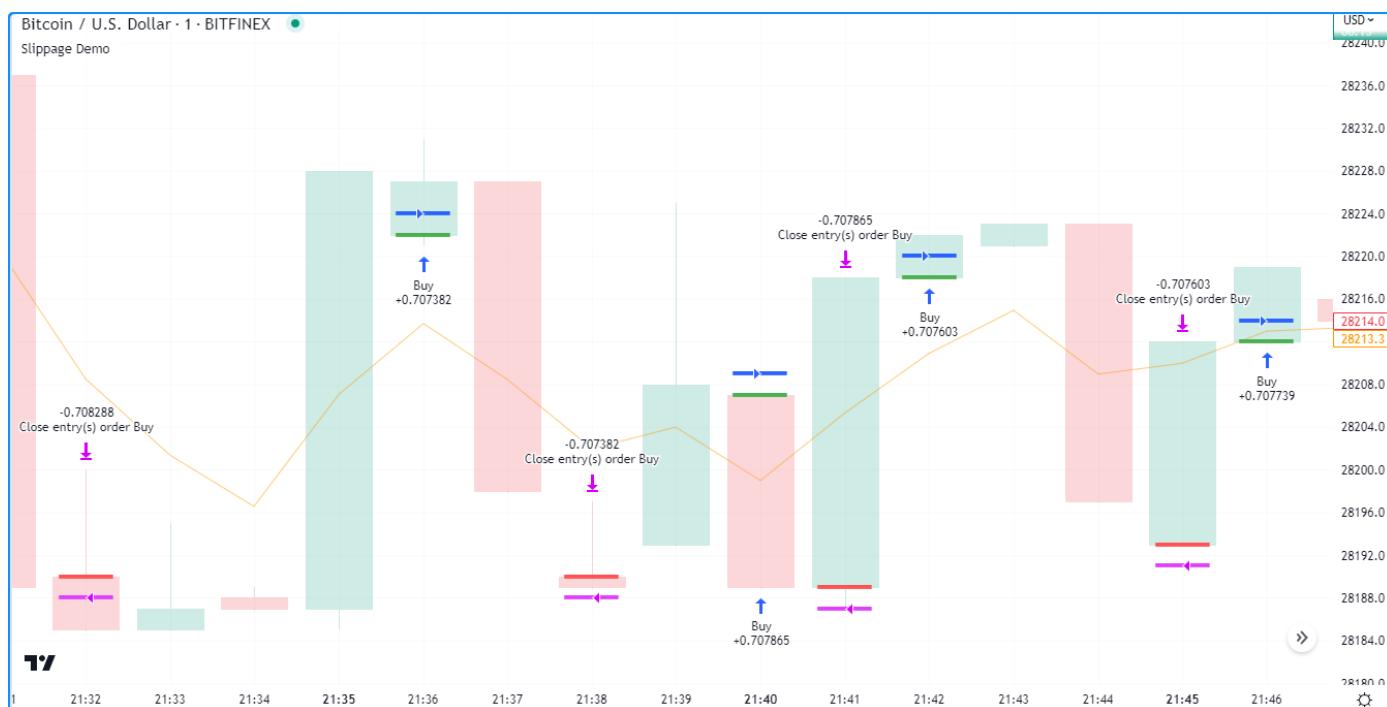
## 滑点和未履行限额

在现实交易中，由于波动性、流动性、订单规模和其他市场因素，经纪商/交易所可能会以与交易者预期略有不同的价格执行订单，这可能会对策略的表现产生深远的影响。经纪商/交易所执行交易的预期价格与实际价格之间的差异就是我们所说的滑点。滑点是动态的且不可预测的，因此无法精确模拟。然而，在回溯测试或前向测试期间考虑每笔交易的少量滑点可能有助于结果更好地符合现实。用户可以通过 `slippage` 在 `strategy()` 声明中包含参数或在策略设置的“属性”选项卡中设置“滑点”输入，在其策略结果中对滑点进行建模，大小为固定数量的价格变动。

以下示例演示了滑点模拟如何影响策略测试中市价订单的执行价格。当市场价格高于 EMA 且 EMA 上升时，下面的脚本会下达 2% 股票的“买入”市价单；当价格低于 EMA 且下降时，则平仓。我们已将其纳入 `strategy()` 函数中，该函数声明每个模拟订单的价格将沿交易方向滑动 20 个报价点。该脚本使用 `strategy.opentrades.entry_bar_index()` 和 `strategy.closetrades.exit_bar_index()` 来获取和，并利用它来获取订单的。当柱索引位于 0 时，是第一个 `strategy.opentrades.entry_price()` 值。处，是上次平仓交易的 `strategy.closetrades.exit_price()` 值。该脚本绘制了预期成交价格以及滑点后的模拟成交价格，以直观地比较差异：

```
slippage =
```

```
20``entryIndex``exitIndex``fillPrice``entryIndex``fillPrice``exitIndex``fillPrice
```



```
//@version=5
strategy(
    "Slippage Demo", overlay = true, slippage = 20,
    default_qty_value = 2, default_qty_type = strategy.percent_of_equity
)

int length = input.int(5, "Length")

//@variable Exponential moving average with an input `length`.
float ma = ta.ema(close, length)

//@variable Returns `true` when `ma` has increased and `close` is greater than it,
//`false` otherwise.

```

```

bool longCondition = close > ma and ma > ma[1]
//@variable Returns `true` when `ma` has decreased and `close` is less than it, `false` otherwise.
bool shortCondition = close < ma and ma < ma[1]

// Enter a long market position on `longCondition`, close the position on
// `shortCondition`.
if longCondition
    strategy.entry("Buy", strategy.long)
if shortCondition
    strategy.close("Buy")

//@variable The `bar_index` of the position's entry order fill.
int entryIndex = strategy.opentrades.entry_bar_index(0)
//@variable The `bar_index` of the position's close order fill.
int exitIndex = strategy.closedtrades.exit_bar_index(strategy.closedtrades - 1)

//@variable The fill price simulated by the strategy.
float fillPrice = switch bar_index
    entryIndex => strategy.opentrades.entry_price(0)
    exitIndex => strategy.closedtrades.exit_price(strategy.closedtrades - 1)

//@variable The expected fill price of the open market position.
float expectedPrice = fillPrice ? open : na

color expectedColor = na
color filledColor = na

if bar_index == entryIndex
    expectedColor := color.green
    filledColor := color.blue
else if bar_index == exitIndex
    expectedColor := color.red
    filledColor := color.fuchsia

plot(ma, color = color.new(color.orange, 50))

plotchar(fillPrice ? open : na, "Expected fill price", "-", location.absolute,
expectedColor)
plotchar(fillPrice, "Fill price after slippage", "-", location.absolute, filledColor)

```

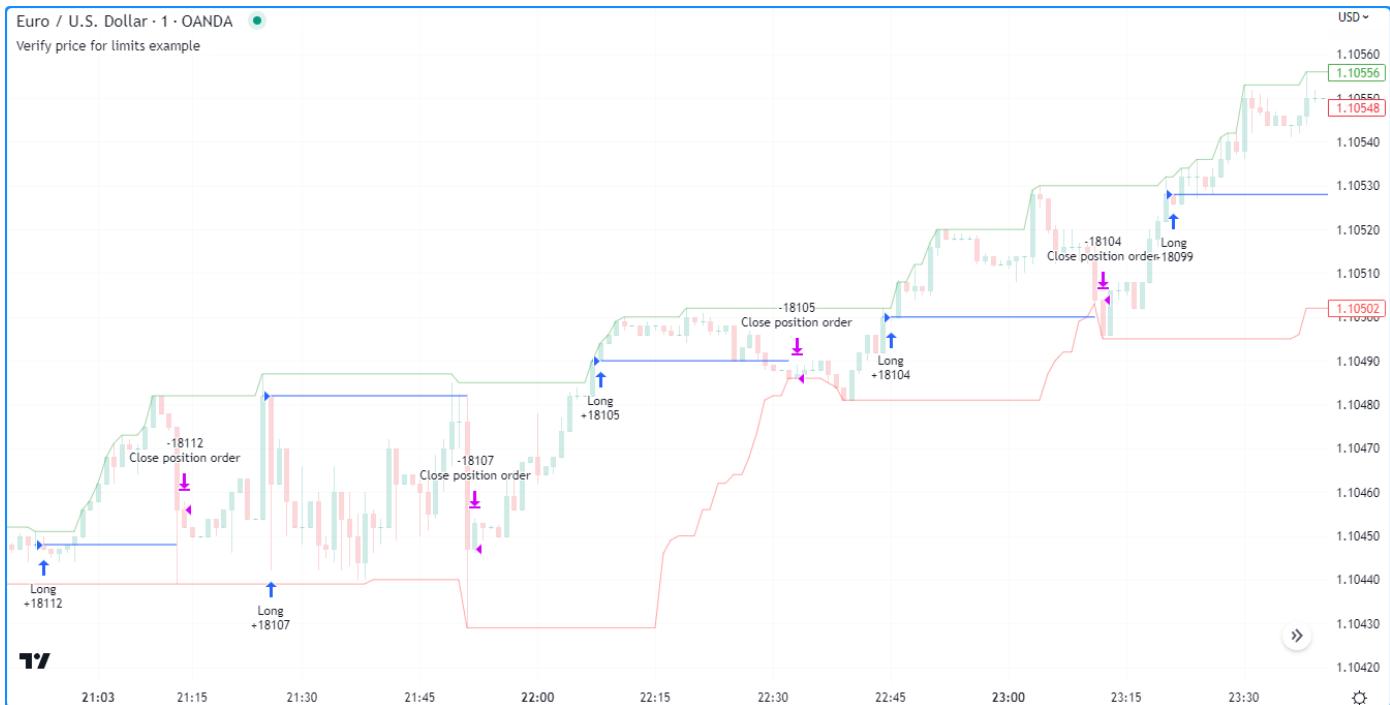
- 注意：

由于该策略对所有订单成交应用恒定滑点，因此某些订单可能会在模拟中的蜡烛范围之外成交。因此，用户应谨慎使用此设置，因为过度的模拟滑点可能会产生不切实际的更差的测试结果。

一些交易者可能认为他们可以通过使用限价单来避免滑点的不利影响，因为与市价单不同，限价单不能以比指定值更差的价格执行。然而，根据现实市场的状况，即使市场价格达到订单价格，限价订单也有可能无法成交，因为限价订单只有在证券具有足够的流动性和价格行为时才能成交。为了考虑回测中未成交订单的可能性，用户可以 `backtest_fill_limits_assumption` 在声明语句中指定该值，或使用“属性”选项卡中输入的“验证限价订单价

格”来指示策略仅在价格变动后才填写限价订单过去订单价格的指定数量的价格变动。

以下示例在柱上放置 2% 净值的限价单，`hlcc4` 此时 `high` 是 `highest` 过去 `length` 柱的值并且没有待处理的条目。当 `low` 为该值时，该策略会平仓并取消所有订单 `lowest`。每次策略触发订单时，它都会在处绘制一条水平线 `limitPrice`，并在每个柱上更新该水平线，直到平仓或取消订单：



```
//@version=5
strategy(
    "Verify price for limits example", overlay = true,
    default_qty_type = strategy.percent_of_equity, default_qty_value = 2
)

int length = input.int(25, title = "Length")

//@variable Draws a line at the limit price of the most recent entry order.
var line limitLine = na

// Highest high and lowest low
highest = ta.highest(length)
lowest = ta.lowest(length)

// Place an entry order and draw a new line when the the `high` equals the `highest` value and `limitLine` is `na`.
if high == highest and na(limitLine)
    float limitPrice = hlcc4
    strategy.entry("Long", strategy.long, limit = limitPrice)
    limitLine := line.new(bar_index, limitPrice, bar_index + 1, limitPrice)

// Close the open market position, cancel orders, and set `limitLine` to `na` when the `low` equals the `lowest` value.
if low == lowest
```

```

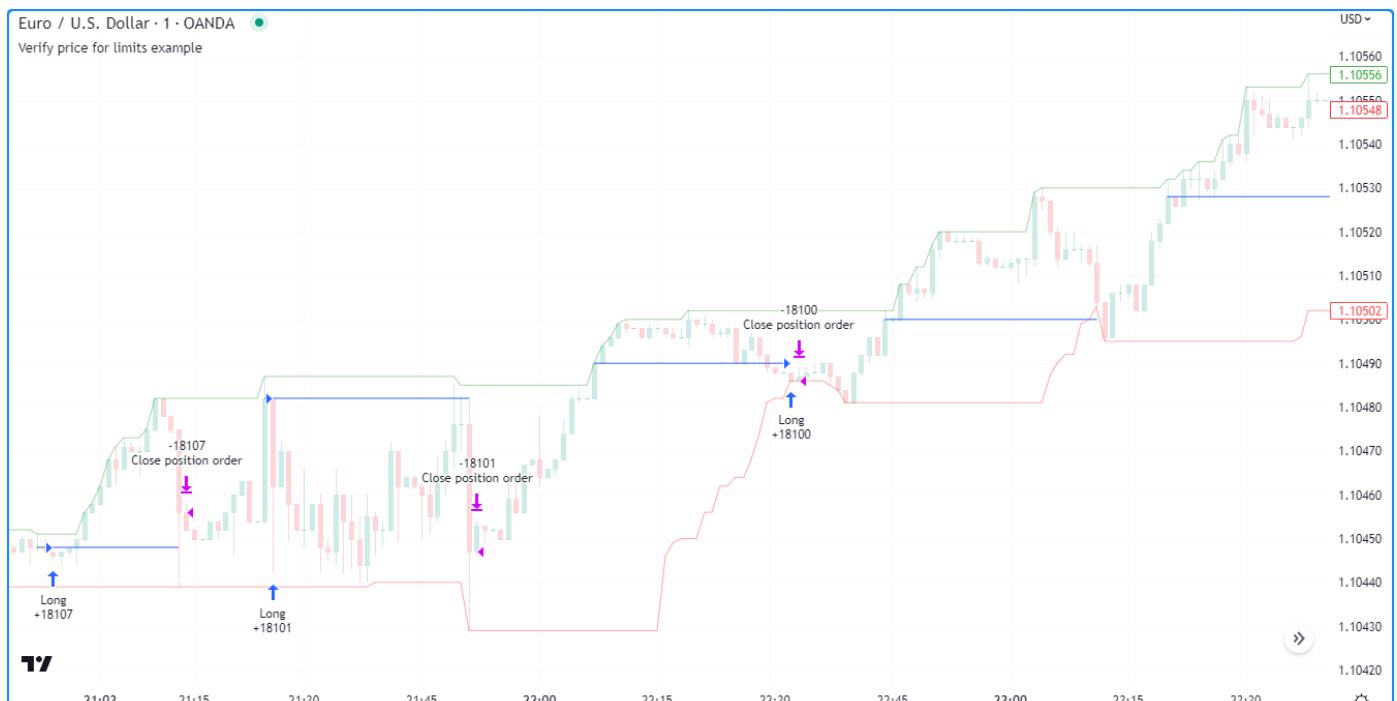
strategy.cancel_all()
limitLine := na
strategy.close_all()

// Update the `x2` value of `limitLine` if it isn't `na`.
if not na(limitLine)
    limitLine.set_x2(bar_index + 1)

plot(highest, "Highest High", color = color.new(color.green, 50))
plot(lowest, "Lowest Low", color = color.new(color.red, 50))

```

默认情况下，脚本假定所有限价单都保证成交。然而，现实交易中的情况往往并非如此。让我们在限价订单中添加价格验证，以解决可能未成交的订单。在这个例子中，我们已经包含在 `strategy()` 函数调用中。正如我们所看到的，使用限价验证会省略一些模拟订单成交并更改其他订单的时间，因为挂单现在只能在价格突破限价三个跳动点后才能成交：`backtest_fill_limits_assumption = 3`



## 笔记

值得注意的是，尽管限价验证改变了某些订单执行的时间，但该策略以相同的价格模拟它们。这种“时间扭曲”效应是一种折衷方案，可以保留已验证限价订单的价格，但它可能会导致策略有时模拟其填充，而这在现实世界中不一定可能的。用户在分析策略结果时应谨慎对待此设置并了解其局限性。

## 风险管理

设计一种表现良好的策略是一项具有挑战性的任务，更不用说在广泛的市场中执行的策略了。大多数是针对特定市场模式/条件而设计的，当应用于其他数据时可能会产生无法控制的损失。因此，策略的风险管理质量对其绩效至关重要。用户可以使用带有前缀的特殊命令在策略脚本中设置风险管理标准 `strategy.risk`。

策略可以包含任意数量的风险管理标准的任意组合。所有风险管理命令都会在每个价格变动和订单执行事件上执行，无论策略的计算行为发生任何变化。无法在脚本运行时禁用任何这些命令。无论风险规则位于何处，它都将始终适用于策略，除非用户从代码中删除调用。

- [Strategy.risk.allow\\_entry\\_in\(\)](#)

该命令会覆盖[strategy.entry\(\)](#)命令允许的市场方向。当用户使用此函数指定交易方向（例如，[strategy.direction.long](#)）时，策略将仅在该方向上输入交易。但是，需要注意的是，如果脚本在有未平仓头寸的情况下调用相反方向的入场命令，则该策略将模拟市价订单以退出该头寸。

- [Strategy.risk.max\\_cons\\_loss\\_days\(\)](#)

在策略模拟指定数量的交易日连续亏损后，此命令会取消所有挂单、平仓市场头寸并停止所有其他交易操作。

- [Strategy.risk.max\\_drawdown\(\)](#)

在策略的回撤达到函数调用中指定的金额后，此命令将取消所有挂单、平仓市场头寸并停止所有其他交易操作。

- [Strategy.risk.max\\_intraday\\_filled\\_orders\(\)](#)

此命令指定每个交易日（或每个图表栏，如果时间范围高于每日）执行订单的最大数量。一旦该策略执行了当天的最大数量的订单，它就会取消所有挂单，平仓，并停止交易活动，直到当前交易时段结束。

- [Strategy.risk.max\\_intraday\\_loss\(\)](#)

此命令控制策略在每个交易日（或每个图表条，如果时间范围高于每日）允许的最大损失。当策略的损失达到此阈值时，它将取消所有挂单，关闭公开市场头寸，并停止所有交易活动，直到当前交易时段结束。

- [Strategy.risk.max\\_position\\_size\(\)](#)

该命令指定使用[strategy.entry\(\)](#)命令时可能的最大头寸大小。如果进场命令的数量导致市场头寸超过此阈值，策略将减少订单数量，以使结果头寸不超过限制。

## 利润

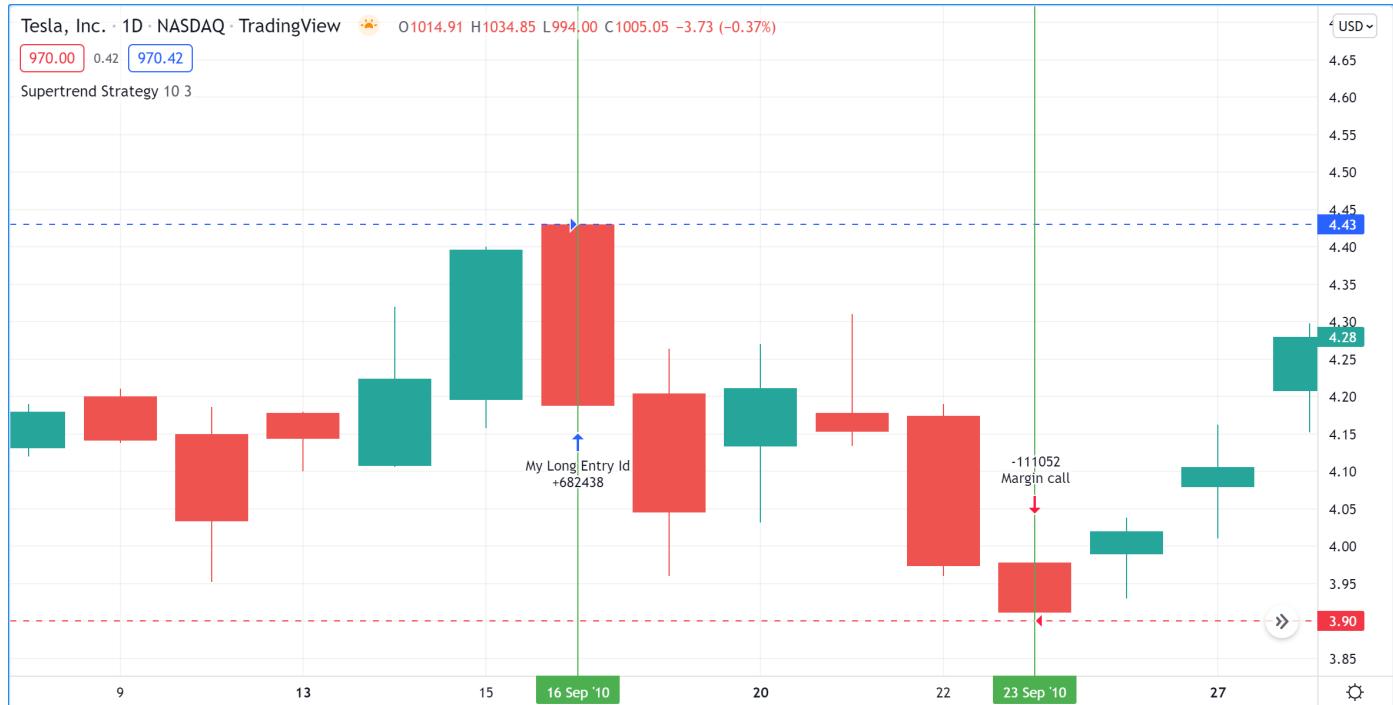
保证金是交易者必须在其账户中作为抵押品持有的市场头寸的最低百分比，以从经纪人处接收和维持贷款，以实现其所需的杠杆率。[strategy\(\)](#)声明的 `margin_long` 和参数 以及脚本设置的“属性”选项卡中的“多头/空头头寸保证金”输入允许策略指定多头和空头头寸的保证金百分比。例如，如果交易者将多头头寸的保证金设置为 25%，则他们必须有足够的资金来覆盖未平仓多头头寸的 25%。这一保证金百分比还意味着交易者有可能将高达 400% 的资产用于交易。`margin_short`

如果策略的模拟资金无法弥补保证金交易的损失，经纪商模拟器会触发追加保证金通知，强制清算全部或部分头寸。模拟器清算的合约/股票/手数/单位的确切数量是弥补损失所需的四倍，以防止后续金条持续追加保证金。模拟器使用以下算法计算金额：

1. 计算该头寸花费的资本金额：`Money Spent = Quantity * Entry Price`
2. 计算证券的市场价值 (MVS)：`MVS = Position Size * Current Price`
3. `MVS` 将未平仓利润计算为和之间的差值。如果头寸是空头，我们将其乘以-1。`Money Spent`
4. 计算策略的股权价值：`Equity = Initial Capital + Net Profit + Open Profit`
5. 计算保证金比例：`Margin Ratio = Margin Percent / 100`
6. 计算保证金值，即平仓交易者部分头寸所需的现金：`Margin = MVS * Margin Ratio`

7. 计算可用资金: Available Funds = Equity - Margin
8. 计算交易者损失的总金额: Loss = Available Funds / Margin Ratio
9. 计算交易者需要清算多少合约/股票/手数/单位来弥补损失。我们将此值截断为与当前交易品种的最小头寸大小相同的小数精度: Cover Amount = TRUNCATE(Loss / Current Price).
10. 计算经纪人将清算多少单位来弥补损失: Margin Call = Cover Amount \* 4

为了详细检查此计算, 让我们将内置的超级趋势策略添加到 NASDAQ:TSLA 图表上的 1D 时间范围, 并将“订单大小”设置为权益的 300%, 将“多头头寸保证金”设置为 25%。策略设置的“属性”选项卡:



第一次入场发生在 2010 年 9 月 16 日柱的开盘价。该策略以 4.43 美元 (入场价) 买入 682,438 股 (头寸规模)。然后, 在2010年9月23日, 当价格跌至3.9 (当前价格) 时, 模拟器通过追加保证金强制平仓了111,052股。

```

Money spent: 682438 * 4.43 = 3023200.34
MVS: 682438 * 3.9 = 2661508.2
Open Profit: -361692.14
Equity: 1000000 + 0 - 361692.14 = 638307.86
Margin Ratio: 25 / 100 = 0.25
Margin: 2661508.2 * 0.25 = 665377.05
Available Funds: 638307.86 - 665377.05 = -27069.19
Money Lost: -27069.19 / 0.25 = -108276.76
Cover Amount: TRUNCATE(-108276.76 / 3.9) = TRUNCATE(-27763.27) = -27763
Margin Call Size: -27763 * 4 = - 111052

```

## 策略提醒

常规 Pine Script™ 指标有两种不同的机制来设置自定义警报条件: alertcondition() 函数, 跟踪每个函数调用的一个特定条件; alert() 函数, 同时跟踪其所有调用, 但在以下方面提供了更大的灵活性: 来电数量、警报消息等。

Pine Script™ 策略不适用于 [alertcondition\(\)调用](#)，[但它们支持通过alert\(\)函数](#)生成自定义警报。除此之外，每个创建订单的函数还带有自己的内置警报功能，不需要任何额外的代码来实现。因此，任何使用下单命令的策略都可以在订单执行时发出警报。我们的用户手册中[警报](#)页面的订单执行事件部分 描述了此类内置策略警报的精确机制。

当策略 `alert()` 同时使用创建订单的函数和函数时，警报创建对话框提供了触发条件之间的选择：它可以在 `alert()` 事件、订单执行事件或两者上触发。

对于许多交易策略来说，触发条件和实时交易之间的延迟可能是一个关键的性能因素。默认情况下，策略脚本只能在实时柱线收盘时 执行[alert\(\)函数调用](#)，[考虑它们使用alert.freq\\_once\\_per\\_bar\\_close freq](#)，无论调用中的参数如何。在创建警报之前，用户还可以通过 在[策略 \(\)](#) 调用中包含或选择策略设置的“属性”选项卡中的“每次报价时重新计算”选项来更改警报频率。但是，根据脚本的不同，这也可能会对策略的行为产生不利影响，因此在使用此方法时请务必谨慎并注意其局限性。`calc_on_every_tick = true`

当向第三方发送警报以实现策略自动化时，我们建议使用订单填写警报而不是[alert\(\) 函数](#)，因为它们没有相同的限制；来自订单执行事件的警报立即执行，不受脚本 `calc_on_every_tick` 设置的影响。用户可以通过编译器注释设置订单执行警报的默认消息 `@strategy_alert_message`。此注释提供的文本将填充“消息”字段，以便填写警报创建对话框。

以下脚本显示了默认订单填写警报消息的简单示例。在[strategy\(\)](#)声明语句上方，它与消息文本中的交易操作、头寸规模、股票代码和填充价格值的占位符 `@strategy_alert_message` 一起使用：

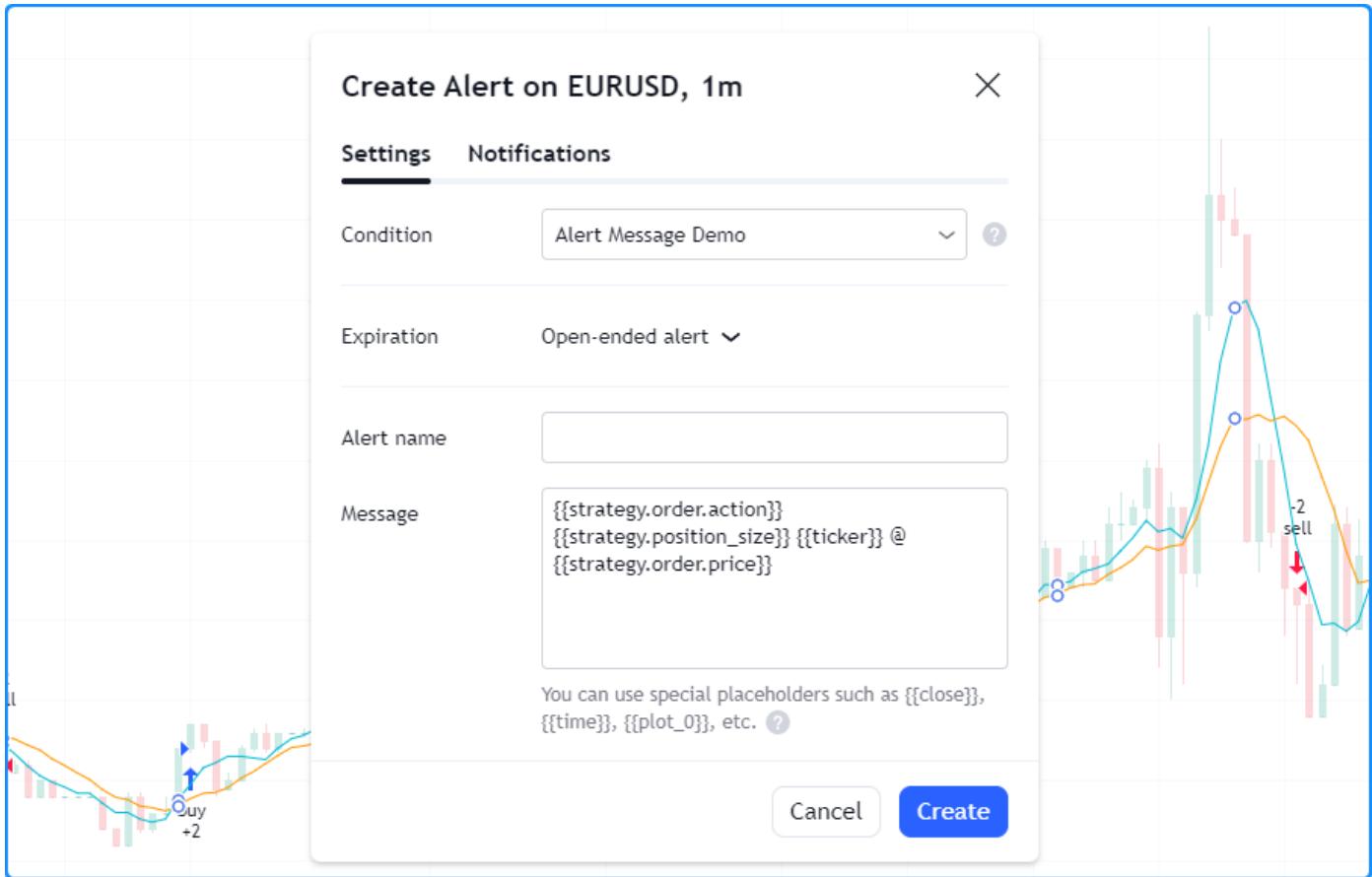
```
//@version=5
//@strategy_alert_message {{strategy.order.action}} {{strategy.position_size}}
{{ticker}} @ {{strategy.order.price}}
strategy("Alert Message Demo", overlay = true)
float fastMa = ta.sma(close, 5)
float slowMa = ta.sma(close, 10)

if ta.crossover(fastMa, slowMa)
    strategy.entry("buy", strategy.long)

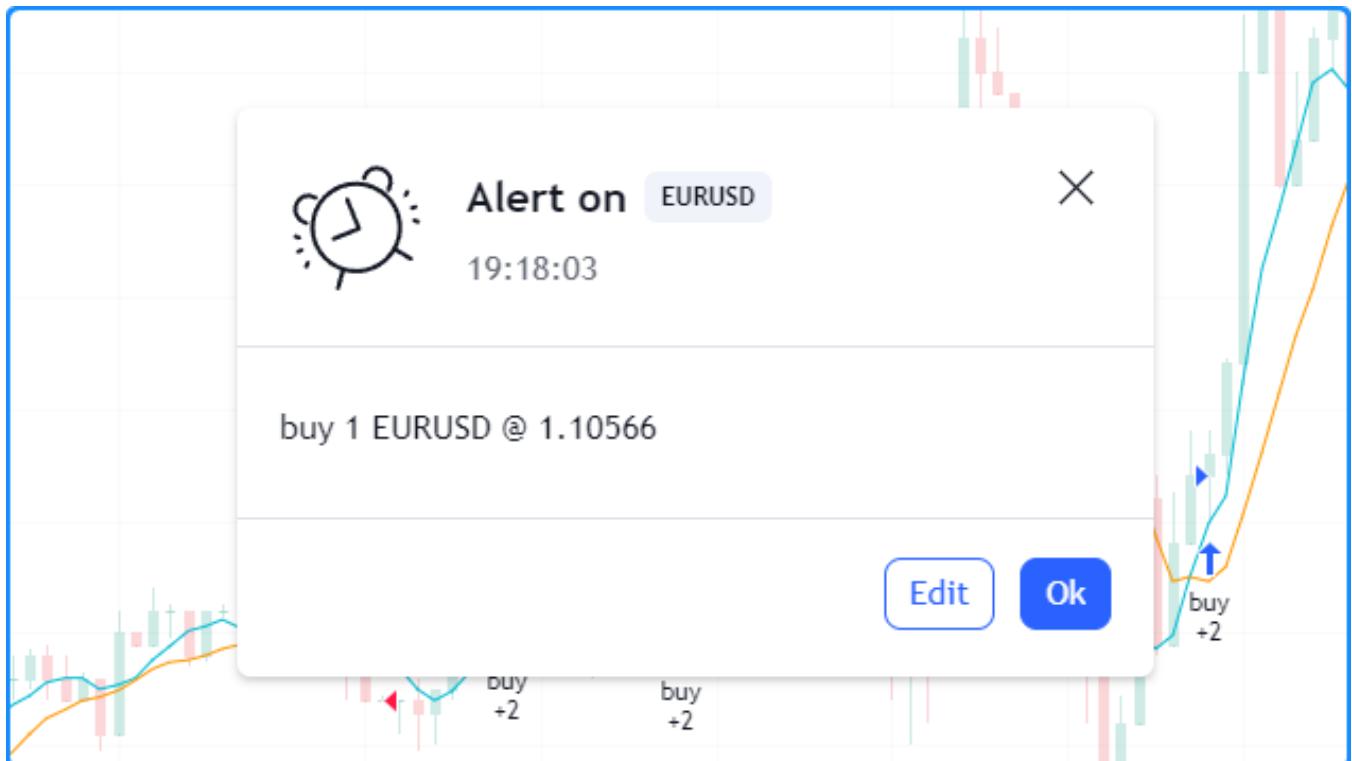
if ta.crossunder(fastMa, slowMa)
    strategy.entry("sell", strategy.short)

plot(fastMa, "Fast MA", color.aqua)
plot(slowMa, "Slow MA", color.orange)
```

当用户从“条件”下拉选项卡中选择其名称时，此脚本将使用其默认消息填充警报创建对话框：



警报触发后，策略将使用相应的值填充警报消息中的占位符。例如：



## 测试策略注意事项

交易者通常会在历史和实时市场条件下测试和调整他们的策略，因为许多人认为分析结果可以提供有关策略特征、潜在弱点以及可能的未来潜力的宝贵见解。然而，交易者应始终意识到模拟策略结果的偏差和局限性，特别是在使用结果支持实时交易决策时。本节概述了与策略验证和调整相关的一些注意事项以及减轻其影响的可能解决方案。

## 笔记

虽然根据现有数据测试策略可能会为交易者提供有关策略质量的有用信息，但重要的是要注意，过去和现在都不能保证未来。金融市场可能会快速且不可预测地变化，这可能会导致策略承受无法控制的损失。此外，模拟结果可能无法完全考虑可能影响交易表现的其他现实因素。因此，我们建议交易者在评估回溯测试和前向测试时彻底了解局限性和风险，并在验证过程中将它们视为“整体的一部分”，而不是仅根据结果做出决策。

## 回溯测试和前向测试

回溯测试是交易者通过模拟和分析历史市场数据的过去结果来评估交易策略或模型的历史表现的技术；该技术假设对过去数据的策略结果进行分析可以深入了解其优势和劣势。在回测时，许多交易者会调整策略的参数以尝试优化其结果。历史结果的分析和优化可以帮助交易者更深入地了解策略。然而，交易者在根据优化回测结果做出决策时应始终了解风险和限制。

与回溯测试并行，审慎的交易系统开发通常还涉及将实时分析作为前瞻性评估交易系统的工具。前瞻性测试旨在衡量策略在实时、真实市场条件下的表现，其中交易成本、滑点和流动性等因素可能会对其表现产生重大影响。前向测试具有不受某些类型偏差（例如，前瞻偏差或“未来数据泄漏”）影响的明显优点，但缺点是要测试的数据量受到限制。因此，它通常不是用于策略验证的独立解决方案，但它可以提供有关策略在当前市场条件下的表现的有用见解。

回溯测试和前向测试是同一枚硬币的两个方面，因为这两种方法都旨在验证策略的有效性并确定其优点和缺点。通过结合回溯测试和前瞻测试，交易者也许能够弥补一些限制，并更清晰地了解其策略的表现。然而，交易者有责任清理他们的策略和评估流程，以确保见解尽可能与现实保持一致。

## 前瞻偏差

回测某些策略（即请求备用时间范围数据、使用重绘变量（例如`timenow`）或更改柱内订单填充的计算行为的策略）的一个典型问题是在评估期间未来数据泄漏到过去，这称为前瞻偏差。这种偏差不仅是导致不切实际的策略结果的常见原因，因为未来实际上是事先无法得知的，而且它也是策略重绘的典型原因之一。交易者通常可以通过前向测试他们的系统来确认这种偏差，因为前瞻偏差不适用于当前柱线之外不存在已知数据的实时数据。用户可以通过确保不使用将未来泄漏到过去的重绘变量来消除策略中的这种偏差，`request.*()` 函数不包括 `barmerge.lookahead_on` 而不偏移数据系列，如 [重绘](#) 页面的这一部分所述，他们使用现实的计算行为。

## 选择偏差

选择偏差是许多交易者在测试策略时遇到的常见问题。当交易者仅分析特定工具或时间范围的结果而忽略其他工具时，就会发生这种情况。这种偏差可能会导致策略稳健性的扭曲，从而可能影响交易决策和性能优化。交易者可以通过在多个、理想情况下多样化的交易品种和时间框架上评估其策略来减少选择偏差的影响，因此不要忽视其分析或精选测试范围中的不良表现结果。

## 过度拟合

优化回测时的一个常见陷阱是可能出现过度拟合（“曲线拟合”），当策略针对特定数据量身定制并且无法很好地概括新的、未见过的数据时，就会发生这种情况。一种广泛使用的方法有助于减少过度拟合的可能性并促进更好的泛化，即将仪器的数据分成两个或多个部分，以测试用于优化的样本之外的策略，也称为“样本内”(IS) 和“样本外”(OOS) 回测。在这种方法中，交易者使用 IS 数据进行策略优化，而 OOS 部分用于在新数据上测试和评估 IS 优化的性能，而无需进一步优化。虽然这种方法和其他更强大的方法可以让您了解优化后策略的表现，但交易者应该谨慎行事，因为未来本质上是不可知的。无论用于测试和优化的数据如何，任何交易策略都无法保证未来的表现。

## 表格

### 介绍

表格是可用于将信息定位在脚本可视空间中特定且固定位置的对象。与 Pine Script™ 中绘制的所有其他绘图或对象相反，表格并不锚定到特定的条形；它们漂浮在脚本的空间中，无论是在覆盖模式还是窗格模式下，在研究还是策略中，与正在查看的图表栏或使用的缩放系数无关。

表格包含按列和行排列的单元格，非常类似于电子表格。它们是通过两个不同的步骤创建和填充的：

- 表的结构和关键属性是使用[table.new\(\)](#)定义的，它返回一个表 ID，其作用类似于指向表的指针，就像标签、行或数组 ID 一样。[table.new\(\)](#)调用将创建表对象但不显示它。
- 创建后，为了使其显示，必须对每个单元格使用一次[table.cell\(\)](#)调用来填充表格。表格单元格可以包含文本，也可以不包含文本。第二步是定义单元格的宽度和高度。

先前创建的表的大多数属性都可以使用 `table.set_*`() setter 函数进行更改。可以使用 `table.cell_set_*`() 函数修改先前填充的单元格的属性。

通过将表格锚定到九个参考之一来将表格定位在指示器的空间中：四个角或中点，包括中心。通过从锚点展开表格来定位表格，因此锚定到 [position.middle\\_right](#) 引用的表格 将通过从该锚点向上、向下和向左展开来绘制。

有两种模式可用于确定表格单元格的宽度/高度：

- 默认自动模式使用其中最宽/最高的文本来计算列/行中单元格的宽度/高度。
- 显式模式允许程序员使用指示器可用 x/y 空间的百分比来定义单元格的宽度/高度。

显示的表内容始终代表表的最后状态，因为它是在数据集的最后一个柱上的脚本上次执行时绘制的。与数据窗口或指标值中显示的值相反，当脚本用户将光标移到特定图表栏上时，表中显示的变量内容不会改变。因此，强烈建议始终将所有 `table.*()` 调用的执行限制为数据集的第一个或最后一个柱。因此：

- 使用[var](#)关键字来声明表。
- 将所有其他调用包含在[if barstate.islast](#)块内。

一个脚本中可以使用多个表，只要它们各自锚定到不同的位置即可。每个表对象都由其自己的 ID 标识。所有表中单元格数量的限制由一个脚本中使用的单元格总数决定。

## 创建表

使用[table.new\(\)](#)创建表时，三个参数是必需的：表的位置及其列数和行数。其他五个参数是可选的：表格的背景颜色、表格外框的颜色和宽度以及所有单元格周围边框的颜色和宽度（不包括外框）。除了列数和行数之外的所有表格属性都可以使用 setter 函数进行修改：[table.set\\_position\(\)](#)、[table.set\\_bgcolor\(\)](#)、[table.set\\_frame\\_color\(\)](#)、[table.set\\_frame\\_width\(\)](#)、[table.set\\_border\\_color\(\)](#)和[table.set\\_border\\_width\(\)](#)。

可以使用[table.delete\(\)](#)删除表，并且可以使用[table.clear\(\)](#)有选择地删除表的内容。

使用[table.cell\(\)](#)填充单元格时，必须为四个强制参数提供参数：单元格所属的表 ID、使用从零开始的索引的列索引和行索引，以及单元格包含的文本字符串，这可以为空。其他七个参数是可选的：单元格的宽度和高度、文本的属性（颜色、水平和垂直对齐方式、大小）以及单元格的背景颜色。所有单元格属性都可以使用 setter 函数修改：[table.cell\\_set\\_text\(\)](#)、[table.cell\\_set\\_width\(\)](#)、[table.cell\\_set\\_height\(\)](#)、[table.cell\\_set\\_text\\_color\(\)](#)、[table.cell\\_set\\_text\\_halign\(\)](#)、[table.cell\\_set\\_text\\_valign\(\)](#)、[table.cell\\_set\\_text\\_size\(\)](#) 和 [table.cell\\_set\\_bgcolor\(\)](#)。

请记住，每次连续调用[table.cell\(\)](#)都会重新定义所有单元格的属性，删除先前对同一单元格调用[table.cell\(\)](#)设置的任何属性。

## 将单个值放置在固定位置

让我们创建第一个表，它将 ATR 的值放在图表的右上角。我们首先创建一个单单元格表格，然后填充该单元格：

```
//@version=5
indicator("ATR", "", true)
// We use `var` to only initialize the table on the first bar.
var table atrDisplay = table.new(position.top_right, 1, 1)
// We call `ta.atr()` outside the `if` block so it executes on each bar.
myAtr = ta.atr(14)
if barstate.islast
    // We only populate the table on the last bar.
    table.cell(atrDisplay, 0, 0, str.tostring(myAtr))
```



注意：

- 我们在使用[table.new\(\)](#)创建表时 使用[var](#)关键字。
- 我们使用[table.cell\(\)](#)填充[if barstate.islast](#)块内的单元格。
- 填充单元格时，我们不指定[width](#)或[height](#)。因此，单元格的宽度和高度将自动调整以适应其包含的文本。

- `ta.atr(14)` 我们在进入`if`块之前调用，以便它在每个柱上进行评估。如果我们在`if str.tostring(ta.atr(14))`块内使用，该函数将无法正确计算，因为它将在数据集的最后一个柱上调用，而没有根据前一个柱计算出必要的值。

让我们提高脚本的可用性和美观性：

```
//@version=5
indicator("ATR", "", true)
atrPeriodInput = input.int(14, "ATR period", minval = 1, tooltip = "Using a period of
1 yields True Range.")

var table atrDisplay = table.new(position.top_right, 1, 1, bgcolor = color.gray,
frame_width = 2, frame_color = color.black)
myAtr = ta.atr(atrPeriodInput)
if barstate.islast
    table.cell(atrDisplay, 0, 0, str.tostring(myAtr, format.mintick), text_color =
color.white)
```



注意：

- 我们使用`table.new()`来定义背景颜色、框架颜色及其宽度。
- 当使用`table.cell()`填充单元格时，我们将文本设置为以白色显示。
- 我们将`format.mintick`作为第二个参数传递给`str.tostring()` 函数，以将 ATR 的精度限制为图表的刻度精度。
- 我们现在使用输入来允许脚本用户指定 ATR 的周期。输入还包括一个工具提示，当用户将鼠标悬停在脚本的“设置/输入”选项卡中的“i”图标上时可以看到该工具提示。

## 为图表的背景着色

此示例使用单单元格表格为 RSI 牛市/熊市状态的图表背景着色：

```

//@version=5
indicator("Chart background", "", true)
bullColorInput = input.color(color.new(color.green, 95), "Bull", inline = "1")
bearColorInput = input.color(color.new(color.red, 95), "Bear", inline = "1")
// —— Function colors chart bg on RSI bull/bear state.
colorChartBg(bullColor, bearColor) =>
    var table bgTable = table.new(position.middle_center, 1, 1)
    float r = ta.rsi(close, 20)
    color bgColor = r > 50 ? bullColor : r < 50 ? bearColor : na
    if barstate.islast
        table.cell(bgTable, 0, 0, width = 100, height = 100, bgcolor = bgColor)

colorChartBg(bullColorInput, bearColorInput)

```

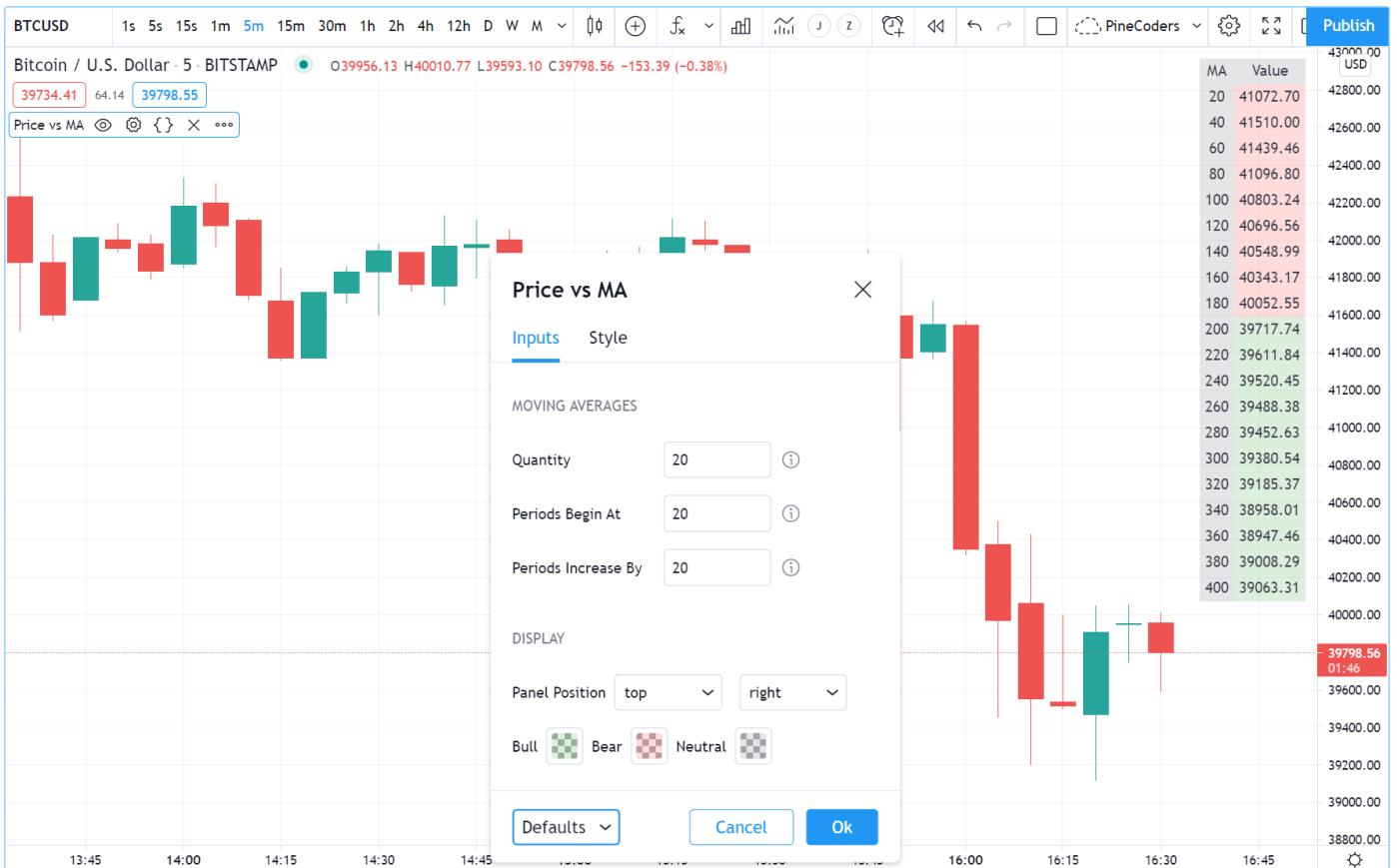
注意：

- 我们为用户提供输入，允许他们指定用于背景的牛市/熊市颜色，并将这些输入颜色作为参数发送给我们的 `colorChartBg()` 函数。
- 我们只创建一个新表一次，使用`var`关键字来声明该表。
- 我们仅在最后一个栏上使用`table.cell()`来指定单元格的属性。我们将单元格设置为指标空间的宽度和高度，以便它覆盖整个图表。

## 创建显示面板

桌子是制作精致展示面板的理想选择。它们不仅使显示面板始终在固定位置可见，而且还提供更灵活的格式设置，因为每个单元格的属性都是单独控制的：背景、文本颜色、大小和对齐方式等。

在这里，我们创建一个基本显示面板，显示用户选择的 MA 值数量。我们在第一列中显示它们的周期，然后用绿色/红色/灰色背景显示它们的值，该值随着价格相对于每个 MA 的位置而变化。当价格高于/低于 MA 时，单元格的背景颜色为牛市/熊市颜色。当 MA 落在当前柱的 [开盘价](#) 和 [收盘价](#) 之间时，单元格的背景为中性色：



```

//@version=5
indicator("Price vs MA", "", true)

var string GP1 = "Moving averages"
int      masQtyInput      = input.int(20, "Quantity", minval = 1, maxval = 40, group = GP1, tooltip = "1-40")
int      masStartInput    = input.int(20, "Periods begin at", minval = 2, maxval = 200, group = GP1, tooltip = "2-200")
int      masStepInput     = input.int(20, "Periods increase by", minval = 1, maxval = 100, group = GP1, tooltip = "1-100")

var string GP2 = "Display"
string  tableYposInput = input.string("top", "Panel position", inline = "11", options = ["top", "middle", "bottom"], group = GP2)
string  tableXposInput = input.string("right", "", inline = "11", options = ["left", "center", "right"], group = GP2)
color   bullColorInput  = input.color(color.new(color.green, 30), "Bull", inline = "12", group = GP2)
color   bearColorInput  = input.color(color.new(color.red, 30), "Bear", inline = "12", group = GP2)
color   neutColorInput  = input.color(color.new(color.gray, 30), "Neutral", inline = "12", group = GP2)

var table panel = table.new(tableYposInput + "_" + tableXposInput, 2, masQtyInput + 1)
if barstate.islast
    // Table header.

```

```

table.cell(panel, 0, 0, "MA", bgcolor = neutColorInput)
table.cell(panel, 1, 0, "Value", bgcolor = neutColorInput)

int period = masStartInput
for i = 1 to masQtyInput
    // ----- Call MAs on each bar.
    float ma = ta.sma(close, period)
    // ----- Only execute table code on last bar.
    if barstate.islast
        // Period in left column.
        table.cell(panel, 0, i, str.tostring(period), bgcolor = neutColorInput)
        // If MA is between the open and close, use neutral color. If close is
        lower/higher than MA, use bull/bear color.
        bgColor = close > ma ? open < ma ? neutColorInput : bullColorInput : open > ma
        ? neutColorInput : bearColorInput
        // MA value in right column.
        table.cell(panel, 1, i, str.tostring(ma, format.mintick), text_color =
color.black, bgcolor = bgColor)
    period += masStepInput

```

注意：

- 用户可以从输入中选择表格的位置，以及用于右列单元格背景的牛市/熊市/中性颜色。
- 表的行数由用户选择显示的 MA 数确定。我们为列标题添加一行。
- 即使我们仅填充最后一个柱上的表格单元格，我们也在每个柱上执行对 `ta.sma()` 的调用，以便它们产生正确的结果。可以安全地忽略编译代码时出现的编译器警告。
- 我们使用将输入分成两个部分 `group`，并使用将相关输入连接到同一行 `inline`。我们提供工具提示来记录某些字段的限制 `tooltip`。

## 显示热图

我们的下一个项目是热图，它将表明当前价格相对于过去价值的牛市/熊市关系。为此，我们将使用位于图表底部的表格。我们将仅显示颜色，因此我们的表格将不包含文本；我们只需为其单元格的背景着色即可生成热图。热图使用用户可选择的回顾期。它循环遍历该时期，以确定价格是否高于/低于过去的每个柱，并随着我们进一步了解过去，显示逐渐变浅的牛市/熊市颜色强度：



```

//@version=5
indicator("Price vs Past", "", true)

var int MAX_LOOKBACK = 300

int    lookBackInput = input.int(150, minval = 1, maxval = MAX_LOOKBACK, step = 10)
color  bullColorInput = input.color(#00FF00ff, "Bull", inline = "11")
color  bearColorInput = input.color(#FF0080ff, "Bear", inline = "11")

// ----- Function draws a heatmap showing the position of the current `_src` relative
// to its past `_lookBack` values.
drawHeatmap(src, lookBack) =>
    // float src      : evaluated price series.
    // int   lookBack: number of past bars evaluated.
    // Dependency: MAX_LOOKBACK

    // Force historical buffer to a sufficient size.
    max_bars_back(src, MAX_LOOKBACK)
    // Only run table code on last bar.
    if barstate.islast
        var heatmap = table.new(position.bottom_center, lookBack, 1)
        for i = 1 to lookBackInput
            float transp = 100. * i / lookBack
            if src > src[i]
                table.cell(heatmap, lookBack - i, 0, bgcolor =
color.new(bullColorInput, transp))
            else
                table.cell(heatmap, lookBack - i, 0, bgcolor =
color.new(bearColorInput, transp))

    drawHeatmap(high, lookBackInput)

```

注意：

- 我们将最大回溯期定义为 `MAX_LOOKBACK` 常数。这是一个重要的值，我们将其用于两个目的：指定我们将在单行表中创建的列数，以及指定 `_src` 函数中参数所需的回溯期，以便我们强制使用 Pine Script™ 创建历史缓冲区大小，使我们能够在 `for _src` 循环中引用过去值的所需数量。
- 我们为用户提供了在输入中配置牛市/熊市颜色的可能性，并且我们用来 `inline` 将颜色选择放在同一行上。
- 在我们的函数中，我们将表创建代码包含在 `if barstate.islast` 构造中，以便它仅在图表的最后一个柱上运行。
- 表的初始化是在 `if` 语句内完成的。因此，以及它使用 `var` 关键字的事实，初始化仅在脚本第一次在最后一个柱上执行时发生。请注意，此行为与脚本全局范围内的常见 `var` 声明不同，其中初始化发生在数据集的第一个柱上，即 `bar_index` 零处。
- 我们没有为 `table.cell()` 调用中的参数指定参数，因此使用空字符串。
- 我们计算透明度的方式是，随着历史的深入，颜色的强度会降低。
- 我们使用动态颜色生成根据需要创建不同透明度的基色。

- 与 Pine 脚本中显示的其他对象相反，此热图的单元格不链接到图表栏。配置的回溯期决定热图包含多少个表格单元格，并且当图表水平平移或缩放时，热图不会改变。
- 脚本视觉空间中可以显示的最大单元格数量取决于您的查看设备的分辨率以及图表使用的显示部分。更高分辨率的屏幕和更宽的窗口将允许显示更多的表格单元格。

## 提示

---

- 在策略脚本中创建表时，请记住，除非策略使用，否则`if barstate.islast`块中包含的表代码不会在每次实时更新时执行，因此表不会按您的预期显示。`calc_on_every_tick = true`
- 请记住，对`table.cell()`的连续调用会覆盖先前`table.cell()`调用指定的单元格属性。使用`setter`函数修改单元格的属性。
- 请记住通过将其限制为仅必要的栏来明智地控制表代码的执行。这可以节省服务器资源，并且您的图表将显示得更快，因此每个人都是赢家。

# 文本和形状

---

## 介绍

---

您可以使用 Pine Script™ 使用五种不同的方式显示文本或形状：

- [plotchar\(\)](#)
- [plotshape\(\)](#)
- [plotarrow\(\)](#)
- 使用[label.new\(\)](#)创建的标签
- 使用[table.new\(\)](#)创建的表（参见[Tables](#)）

使用哪一种取决于您的需求：

- 表格可以在图表上的各种相对位置显示文本，当用户水平滚动或缩放图表时，这些文本不会移动。他们的内容不受酒吧限制。相比之下，使用`plotchar()`、`plotshape()`或`label.new()`显示的文本始终与特定的条形图相连，因此它将随着条形图在图表上的位置而移动。有关[表格](#)的更多信息，请参阅表格页面。
- 包括三个函数能够显示预定义的形状：[plotshape\(\)](#)、[plotarrow\(\)](#)和使用[label.new\(\)](#)创建的标签。
- [plotarrow\(\)](#)不能显示文本，只能显示向上或向下箭头。
- [plotchar\(\)](#)和[plotshape\(\)](#)可以在图表的任何条形或所有条形上显示非动态文本。
- [plotchar\(\)](#)只能显示一个字符，而[plotshape\(\)](#)可以显示字符串，包括换行符。
- [label.new\(\)](#)最多可以在图表上显示 500 个标签。其文本可以包含动态文本或“系列字符串”。标签文本也支持换行符。
- 虽然[plotchar\(\)](#)和[plotshape\(\)](#)可以以过去或未来的固定偏移量显示文本，并且在脚本执行期间不能更改，但每个[label.new\(\)](#)调用可以使用可以在飞。

关于 Pine Script™ 字符串，需要记住以下几点：

- 由于 `plotchar()` 和 `plotshape()` 中的参数都需要“const string”参数，因此它不能包含只能在柱上知道的价格等值（“系列字符串”）。
- 要在使用 `label.new()` 显示的文本中包含“系列”值，首先需要使用 `str.tostring()` 将它们转换为字符串。
- Pine 中字符串的连接运算符是 `+`。它用于将字符串组件连接到一个字符串中，例如（其中 `syminfo.tickerid` 是一个内置变量，它以字符串格式返回图表的交易所和交易品种信息）。`msg = "Chart symbol: " + syminfo.tickerid`
- 所有这些函数显示的字符都可以是 Unicode 字符，其中可能包括 Unicode 符号。请参阅此[探索 Unicode](#) 脚本以了解可以使用 Unicode 字符执行哪些操作。
- 有时可以使用函数参数控制文本的颜色或大小，但无法进行内联格式（粗体、斜体、等宽字体等）。
- 来自 Pine 脚本的文本始终以 Trebuchet MS 字体显示在图表上，该字体在许多 TradingView 文本中使用，包括本文本。

该脚本使用 Pine Script™ 中可用的四种方法显示文本：

```
//@version=5
indicator("Four displays of text", overlay = true)
plotchar(ta.rising(close, 5), "`plotchar()``", "□", location.belowbar, color.lime, size = size.small)
plotshape(ta.falling(close, 5), "`plotshape()``", location = location.abovebar, color = na, text = "•`plotshape()•`\n□", textcolor = color.fuchsia, size = size.huge)

if bar_index % 25 == 0
    label.new(bar_index, na, "•LABEL•\nHigh = " + str.tostring(high, format.mintick) +
"\\n□", yloc = yloc.abovebar, style = label.style_none, textcolor = color.black, size = size.normal)

printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
0, 0, txt, bgcolor = color.yellow)
printTable("•TABLE•\n" + str.tostring(bar_index + 1) + " bars\nin the dataset")
```



注意：

- 用于显示每个文本字符串的方法与文本一起显示，但使用 [plotchar\(\)](#) 显示的石灰向上箭头除外，因为它只能显示一个字符。
- 标签和表调用可以插入条件结构中以控制它们的执行时间，而 [plotchar\(\)](#) 和 [plotshape\(\)](#) 则不能。它们的条件绘图必须使用第一个参数进行控制，该参数是一个“series bool”，其值决定 `true` 何时 `false` 显示文本。
- 表和标签中显示的数值首先使用 [str.tostring\(\)](#) 转换为字符串。
- 我们使用 `+` 运算符来连接字符串组件。
- [plotshape\(\)](#) 旨在显示带有文本的形状。它的 `size` 参数控制形状的大小，而不是文本的大小。我们使用 `na` 作为其 `color` 参数，以便形状不可见。
- 与其他文本相反，当您滚动或缩放图表时，表格文本不会移动。
- 某些文本字符串包含 `□` Unicode 箭头 (U+1F807)。
- 某些文本字符串包含 `\n` 表示新行的序列。

## [plotchar\(\)](#)

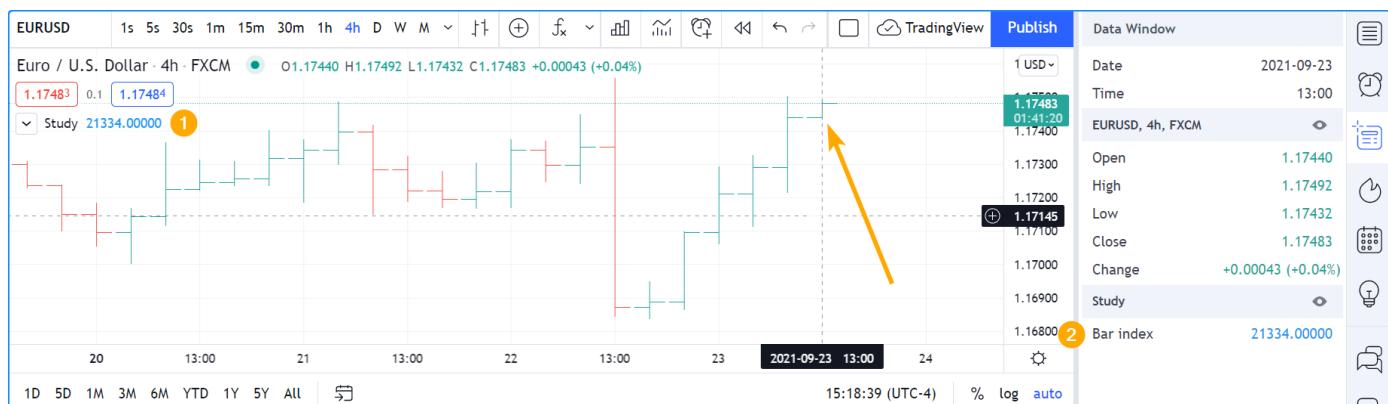
此函数对于在条上显示单个字符很有用。它具有以下语法：

```
plotchar(series, title, char, location, color, offset, text, textColor, editable, size,
showLast, display) → void
```

有关其参数的详细信息，请参阅[plotchar\(\) 的参考手册条目。](#)

正如调试页面的[何时必须保留脚本的比例](#) 部分中所述，该函数可用于显示和检查数据窗口中的值或图表上脚本名称右侧显示的指标值中的值：

```
//@version=5
indicator("", "", true)
plotchar(bar_index, "Bar index", "", location.top)
```



注意：

- 光标位于图表的最后一个柱上。
- 该柱上的 `bar_index` 值显示在指标值 (1) 和数据窗口 (2) 中。
- 我们使用 `location.top` 因为默认的 `location.abovebar` 会将价格按照脚本的比例发挥作用，这通常会干扰其他图。

`plotchar()` 还可以很好地识别图表上的特定点或验证条件是否 `true` 符合我们的预期。此示例在柱形下方显示一个向上箭头，其中 收盘价、最高价和 交易量 在两根柱形中均呈上升趋势：

```
//@version=5
indicator("", "", true)
bool longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or
ta.rising(volume, 2))
plotchar(longSignal, "Long", "▲", location.belowbar, color = na(volume) ? color.gray :
color.blue, size = size.tiny)
```



注意：

- 我们使用这样我们的脚本就可以在没有 体积 数据的符号上运行。如果我们没有在没有 交易量 数据时做出规定（这就是在没有交易量时所做的），那么变量的值将永远不会是，因为在这些情况下会产生收益。`((na(volume) or ta.rising(volume,
2))``na(volume)``true``longSignal``true``ta.rising(volume, 2)` `false`
- 当没有交易量时，我们将箭头显示为灰色，以提醒我们所有三个基本条件均未满足。
- 因为 `plotchar()` 现在在图表上显示一个字符，所以我们用它来控制它的大小。`size = size.tiny`
- 我们已经调整了 `location` 参数以在条形下显示字符。

如果您不介意只绘制圆形，您也可以使用 `plot()` 来实现类似的效果：

```
//@version=5
indicator("", "", true)
longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or
ta.rising(volume, 2))
plot(longSignal ? low - ta.tr : na, "Long", color.blue, 2, plot.style_circles)
```

此方法的不便之处在于，由于 `plot()` 没有相对定位机制，因此必须使用类似 `ta.tr` (条形图的“真实范围”) 之类的方法将圆圈向下移动：



## plotshape()

此功能对于在条形上显示预定义的形状和/或文本非常有用。它具有以下语法：

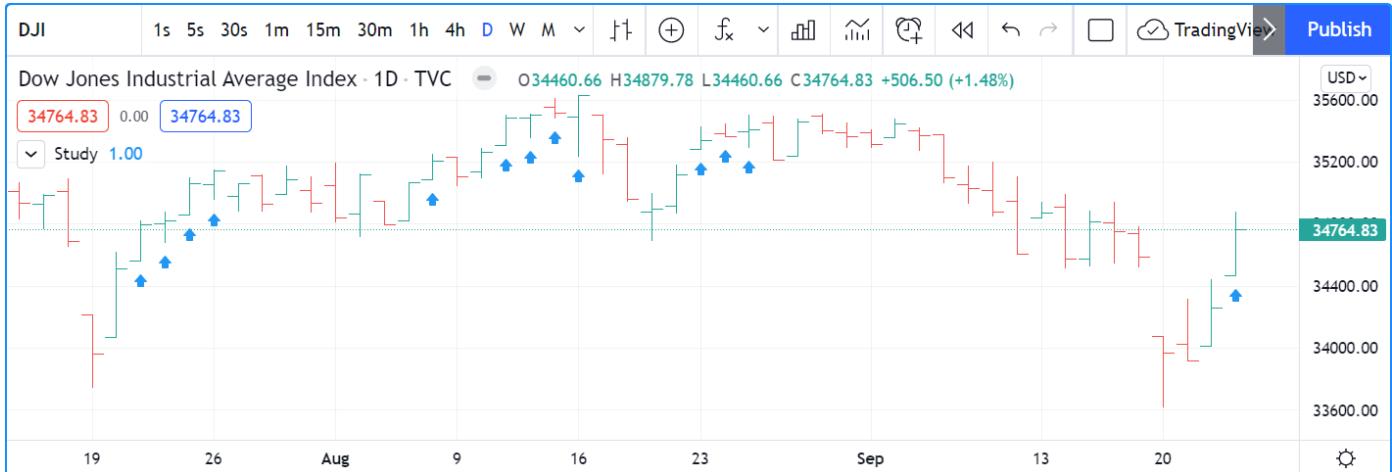
```
plotshape(series, title, style, location, color, offset, text, textcolor, editable,
size, show_last, display) → void
```

有关其参数的详细信息，请参阅[plotshape\(\) 的参考手册条目。](#)

让我们使用该函数来实现与上一节的第二个示例大致相同的结果：

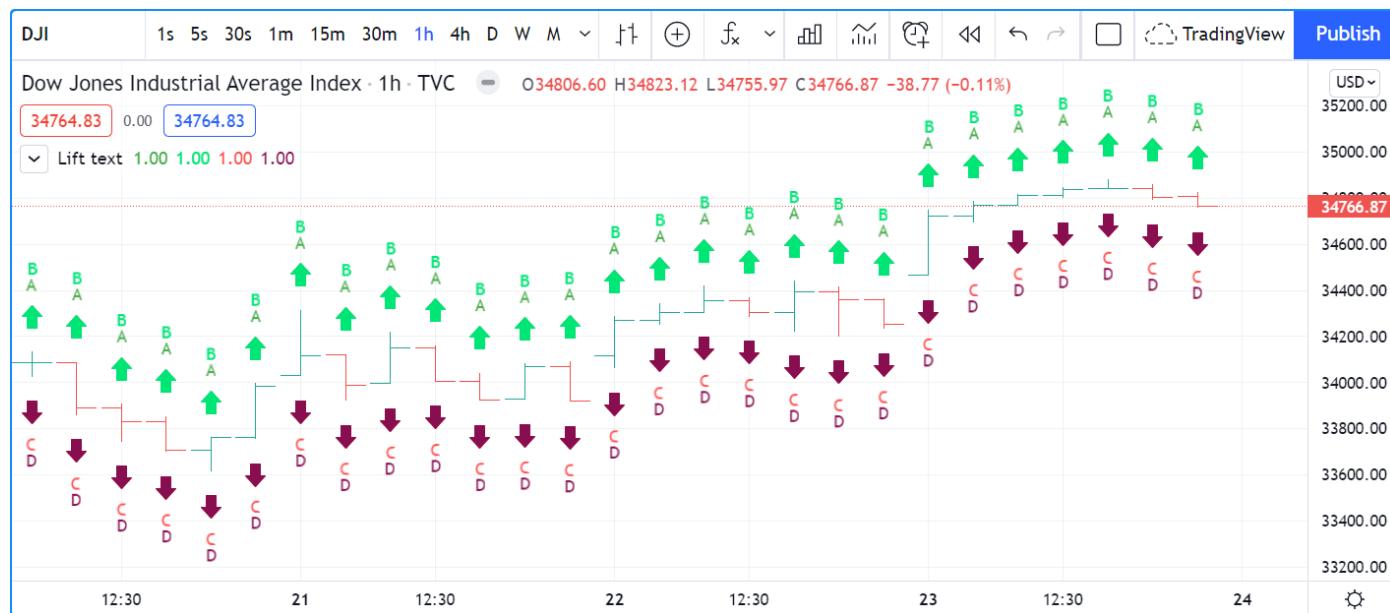
```
//@version=5
indicator("", "", true)
longSignal = ta.rising(close, 2) and ta.rising(high, 2) and (na(volume) or
ta.rising(volume, 2))
plotshape(longSignal, "Long", shape.arrowup, location.belowbar)
```

请注意，这里我们没有使用箭头字符，而是使用参数 `shape.arrowup` 的实参 `style`。



可以使用不同的`plotshape()`调用在条形图上叠加文本。您将需要使用`\n`后跟一个特殊的非打印字符，该字符不会被删除以保留换行符的功能。这里我们使用 Unicode 零宽度空格 (U+200E)。虽然您在以下代码的字符串中看不到它，但它确实存在并且可以复制/粘贴。特殊的 Unicode 字符必须是字符串中的最后一个字符，以便文本向上，并且当您在条形图下方绘图且文本向下时，第一个字符必须是：

```
//@version=5
indicator("Lift text", "", true)
plotshape(true, "", shape.arrowup, location.abovebar, color.green, text = "A")
plotshape(true, "", shape.arrowup, location.abovebar, color.lime, text = "B\n")
plotshape(true, "", shape.arrowdown, location.belowbar, color.red, text = "C")
plotshape(true, "", shape.arrowdown, location.belowbar, color.maroon, text = "\nD")
```



您可以与该参数一起使用的可用形状 `style` 有：

争论	形状	带文字		争论	形状	带文字
shape.xcross	X			shape.arrowup		
shape.cross	+			shape.arrowdown		
shape.circle	●			shape.square		
shape.triangleup	▲			shape.diamond		
shape.triangledown	▼			shape.labelup		
shape.flag	旗			shape.labellow		

## plotarrow()

plotarrow 函数根据函数第一个参数中使用的系列的相对值显示可变长度的向上或向下箭头。[它具有以下语法：](#)

```
plotarrow(series, title, colorup, colordown, offset, minheight, maxheight, editable,
show_last, display) → void
```

有关其参数的详细信息，请参阅[plotarrow\(\) 的参考手册条目。](#)

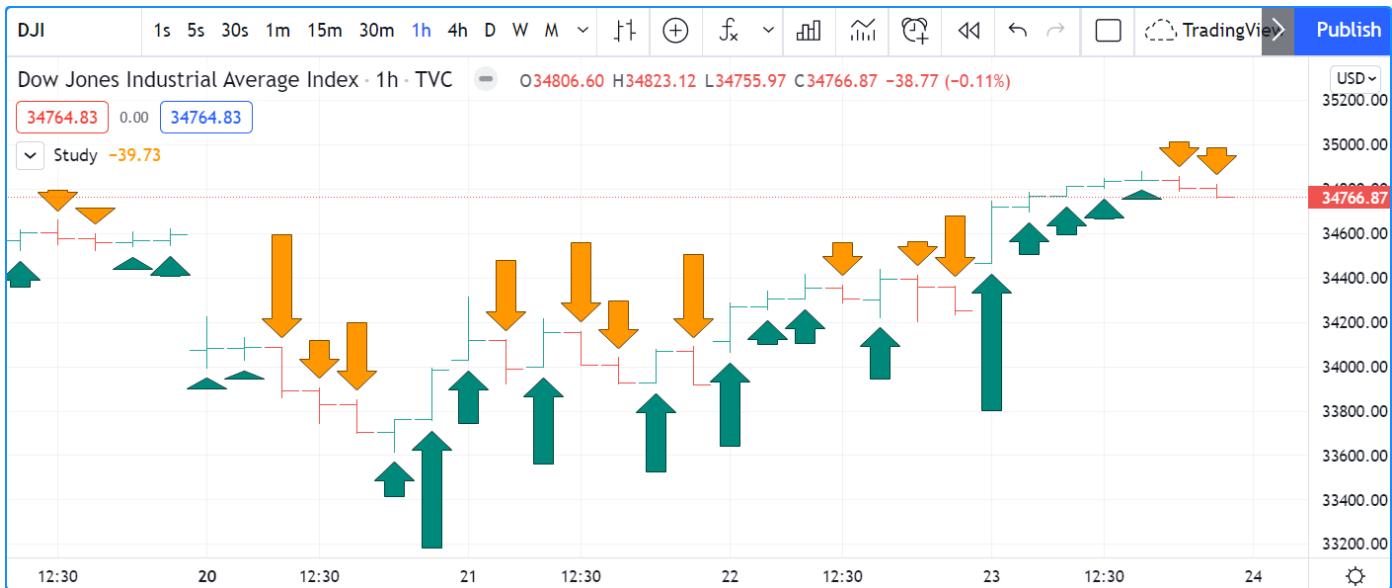
[plotarrow\(\)](#) series 中的参数不是像[plotchar\(\)](#)和[plotshape\(\)](#)中那样的“series bool”；它是一个“series int/float”，它不仅仅是一个简单的或确定何时绘制箭头的值。这是控制提供的参数如何影响[plotarrow\(\)](#) 的行为的逻辑：`true``false``series`

- `series > 0`：显示向上箭头，其长度与该条上的系列相对于其他系列值的相对值成正比。
- `series < 0`：显示向下箭头，使用相同的规则按比例调整大小。
- `series == 0 or na(series)`：不显示箭头。

`minheight` 可以使用和参数控制箭头的最大和最小可能尺寸（以像素为单位）`maxheight`。

这是一个简单的脚本，说明了[plotarrow\(\)](#)的工作原理：

```
//@version=5
indicator("", "", true)
body = close - open
plotarrow(body, colorup = color.teal, colordown = color.orange)
```



请注意箭头的高度与条形体的相对尺寸成正比。

您可以使用任何系列来绘制箭头。这里我们使用“Chaikin Oscillator”的值来控制箭头的位置和大小：

```
// @version=5
indicator("Chaikin Oscillator Arrows", overlay = true)
fastLengthInput = input.int(3, minval = 1)
slowLengthInput = input.int(10, minval = 1)
osc = ta.ema(ta.accdist, fastLengthInput) - ta.ema(ta.accdist, slowLengthInput)
plotarrow(osc)
```



请注意，我们在图表下方的窗格中显示实际的“柴金振荡器”，因此您可以看到使用哪些值来确定箭头的位置和大小。

## 标签

标签仅在 v4 及更高版本的 Pine Script™ 中可用。它们的工作方式与 [plotchar\(\)](#) 和 [plotshape\(\)](#) 非常不同。

标签是对象，例如[线条和框](#)或[表格](#)。与它们一样，它们也使用 ID 来引用，ID 的作用类似于指针。标签 ID 是“标签”类型。与其他对象一样，标签 ID 是“时间序列”，所有用于管理它们的函数都接受“序列”参数，这使得它们非常灵活。

## 笔记

在 TradingView 图表上，一整套绘图工具 允许用户使用鼠标操作创建和修改绘图。虽然它们有时看起来类似于使用 Pine Script™ 代码创建的绘图对象，但它们是不相关的实体。使用 Pine 代码创建的绘图对象无法通过鼠标操作进行修改，并且图表用户界面中的手绘绘图在 Pine 脚本中不可见。

标签是有利的，因为：

- 它们允许将“系列”值转换为文本并放置在图表上。这意味着它们非常适合显示之前无法知道的值，例如脚本计算的任何其他值的价格值、支撑位和阻力位。
- 它们的定位选项比功能的定位选项更灵活 `plot*()`。
- 他们提供更多的显示模式。
- 与 `plot*()` 函数相反，标签处理函数可以插入条件或循环结构中，从而更容易控制其行为。
- 您可以向标签添加工具提示。

与 [plotchar\(\)](#) 和 [plotshape\(\)](#) 相比，使用标签的一个缺点 是您只能在图表上绘制有限数量的标签。默认值为~50，但您可以 `max_labels_count` 在 [indicator\(\)](#) 或 [strategy\(\)](#) 声明语句中使用该参数来指定最多500个。标签（如[lines](#) 和 [boxes](#)）使用垃圾收集机制进行管理，该机制会删除最旧的标签图表，这样只有最近绘制的标签可见。

用于管理标签的内置工具箱都位于 `label` 命名空间中。他们包括：

- [label.new\(\)](#) 创建标签。
- `label.set_*()` 函数来修改现有标签的属性。
- `label.get_*()` 函数读取现有标签的属性。
- [label.delete\(\)](#) 删除标签
- `label.all` 数组始终包含图表上 所有可见标签的 ID。数组的大小取决于脚本的最大标签计数以及您绘制的标签数量。`array.size(label.all)` 将返回数组的大小。

## 创建和修改标签

`label.new()` 函数创建一个新标签。它具有以下签名：

```
label.new(x, y, text, xloc, yloc, color, style, textcolor, size, textalign, tooltip) →  
series label
```

允许您更改标签属性的setter函数有：

- [label.set\\_x\(\)](#)
- [label.set\\_y\(\)](#)
- [label.set\\_xy\(\)](#)
- [label.set\\_text\(\)](#)

- [label.set\\_xloc\(\)](#)
- [label.set\\_yloc\(\)](#)
- [label.set\\_color\(\)](#)
- [label.set\\_style\(\)](#)
- [label.set\\_textcolor\(\)](#)
- [label.set\\_size\(\)](#)
- [label.set\\_textalign\(\)](#)
- [label.set\\_tooltip\(\)](#)

他们都有一个相似的签名。[label.set\\_color\(\)](#)的一个是：

```
label.set_color(id, color) → void
```

在哪里：

- `id` 是要修改属性的标签的ID。
- 下一个参数是要修改的标签的属性。这取决于所使用的 setter 函数。[label.set\\_xy\(\)](#)更改了两个属性，因此它有两个这样的参数。

您可以通过以下方式以最简单的形式创建标签：

```
//@version=5
indicator("", "", true)
label.new(bar_index, high)
```



注意：

- 该标签是使用参数（当前柱的索引 `bar_index`）和（柱的最高值）创建的。`x = bar_index``y = high`
- 我们不为函数的 `text` 参数提供实参。它的默认值为空字符串，不显示任何文本。
- 没有逻辑控制我们的[label.new\(\)](#)调用，因此在每个柱上创建标签。
- 仅显示最后 54 个标签，因为我们的[Indicator\(\)](#)调用不使用 `max_labels_count` 参数来指定默认值 ~50 以外的值。
- [标签会一直保留在条上，直到您的脚本使用label.delete\(\)删除它们，或者垃圾收集将它们删除。](#)

在下一个示例中，我们在最近 50 个柱中具有最高高值的柱上显示一个标签：

```
//@version=5
indicator("", "", true)

// Find the highest `high` in last 50 bars and its offset. Change it's sign so it is
// positive.
LOOKBACK = 50
hi = ta.highest(LOOKBACK)
highestBarOffset = - ta.highestbars(LOOKBACK)

// Create label on bar zero only.
var lbl = label.new(na, na, "", color = color.orange, style =
label.style_label_lower_left)
// When a new high is found, move the label there and update its text and tooltip.
if ta.change(hi)
    // Build label and tooltip strings.
    labelText = "High: " + str.tostring(hi, format.mintick)
    tooltipText = "Offset in bars: " + str.tostring(highestBarOffset) + "\nLow: " +
str.tostring(low[highestBarOffset], format.mintick)
    // Update the label's position, text and tooltip.
    label.set_xy(lbl, bar_index[highestBarOffset], hi)
    label.set_text(lbl, labelText)
    label.set_tooltip(lbl, tooltipText)
```

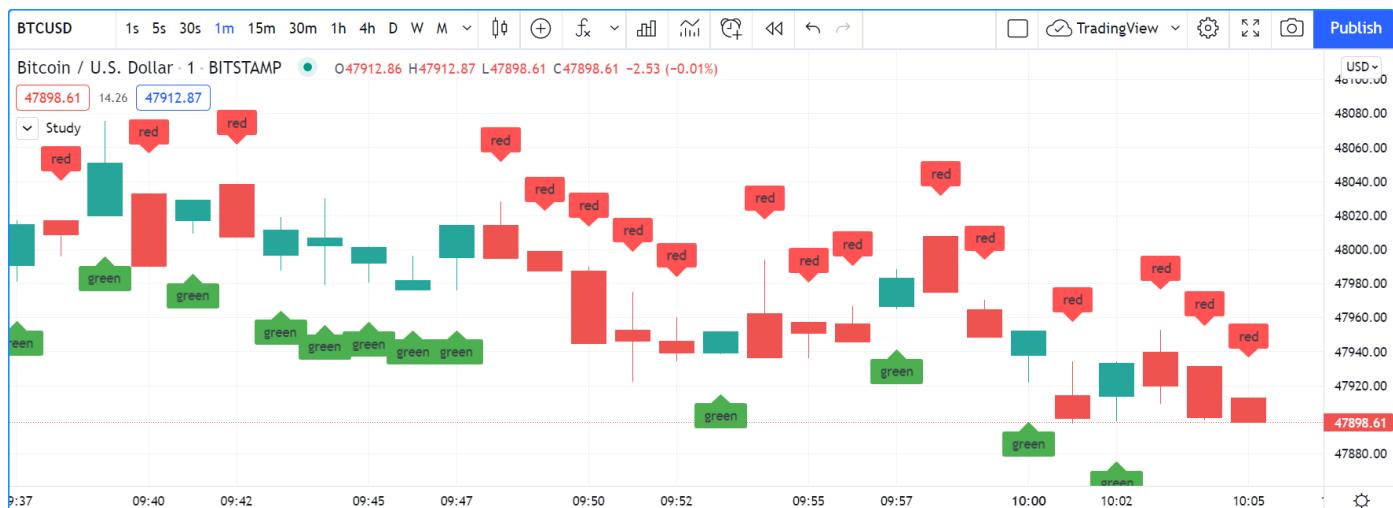


注意：

- 我们仅通过使用 `var` 关键字来声明 `lbl` 包含标签 ID 的变量，从而在第一个栏上创建标签。`label.new()` 调用中的 `x`、`y` 和参数是无关紧要的，因为标签将在以后的柱上更新。但是，我们确实小心地使用了和我们想要的标签，因此以后不需要更新它们。`text``color``style`
- 在每个柱上，我们通过测试值的变化来检测是否发现新高 `hi`
- 当高值发生变化时，我们会用新信息更新标签。为此，我们使用三个 `label.set*()` 调用来更改标签的相关信息。我们使用 `lbl` 变量引用我们的标签，该变量包含我们标签的 ID。因此，该脚本在所有柱中保持相同的标签，但在检测到新高时移动它并更新其信息。

在这里，我们在每个条形上创建一个标签，但我们根据条形的极性有条件地设置其属性：

```
//@version=5
indicator("", "", true)
lbl = label.new(bar_index, na)
if close >= open
    label.set_text(lbl, "green")
    label.set_color(lbl, color.green)
    label.set_yloc(lbl, yloc.belowbar)
    label.set_style(lbl, label.style_label_up)
else
    label.set_text(lbl, "red")
    label.set_color(lbl, color.red)
    label.set_yloc(lbl, yloc.abovebar)
    label.set_style(lbl, label.style_label_down)
```



## 定位标签

标签根据 `x` (柱形) 和 `y` (价格) 坐标放置在图表上。有五个参数影响此行为：`x`、`y`、`xloc` 和：`yloc``style`

- `x`

是柱形索引或时间值。当使用柱线索引时，该值可以在过去或未来偏移（未来最多 500 个柱线）。使用时间值时也可以计算过去或未来的偏移量。可以使用 `label.set_x()` 或 `label.set_xy()` `x` 修改现有标签的值。

- `xloc`

是 `xloc.bar_index` (默认值) 或 `xloc.bar_time`。它确定必须与一起使用哪种类型的参数 `x`。对于 `xloc.bar_index`, `x` 必须是绝对柱索引。对于 `xloc.bar_time`, 必须是与柱的 `开盘时间` `x` 值相对应的 UNIX 时间 (以毫秒为单位)。可以使用 `label.set_xloc()` 修改现有标签的值。`xloc`

- `y`

是标签所在的价格水平。仅在使用默认 `yloc` 值时才考虑它 `yloc.price`。如果 `yloc` 是 `yloc.abovebar` 或 `yloc.belowbar` 则 `y` 忽略该参数。可以使用 `label.set_y()` 或 `label.set_xy()` `y` 修改现有标签的值。

- `yloc`

可以是 `yloc.price` (默认) 、 `yloc.abovebar` 或 `yloc.belowbar`。用于的参数仅在 `yloc.price` 中考虑。可以使用 `label.set_yloc()` 修改现有标签的值。 `yloc`

- `style`

当使用 `yloc.abovebar` 或 `yloc.belowbar` 时，使用的参数会影响标签的视觉外观及其相对于由值 `y` 或条的顶部/底部确定的参考点的位置。可以使用 `label.set_style()` 修改现有标签的值。 `style`

这些是可用的 `style` 参数：

参数	标签	带文字的标签	参数	标签	带文字的标签
<code>label.style_xcross</code>			<code>label.style_label_up</code>		
<code>label.style_cross</code>			<code>label.style_label_down</code>		
<code>label.style_flag</code>			<code>label.style_label_left</code>		
<code>label.style_circle</code>			<code>label.style_label_right</code>		
<code>label.style_square</code>			<code>label.style_label_lower_left</code>		
<code>label.style_diamond</code>			<code>label.style_label_lower_right</code>		
<code>label.style_triangleup</code>			<code>label.style_label_upper_left</code>		
<code>label.style_triangledown</code>			<code>label.style_label_upper_right</code>		
<code>label.style_arrowup</code>			<code>label.style_label_center</code>		
<code>label.style_arrowdown</code>			<code>label.style_none</code>		<code>text</code>

使用 `xloc.bar_time` 时，该 `x` 值必须是以毫秒为单位的 UNIX 时间戳。请参阅“[时间](#)”页面以获取更多信息。当前柱的开始时间可以从 `time` 内置变量中获取。之前柱的柱时间为 `time[1]`，`time[2]` 依此类推。还可以使用 `时间戳` 函数将时间设置为绝对值。您可以添加或减去时间段以实现相对时间偏移。

让我们在最后一个柱上的日期前一天放置一个标签：

```
//@version=5
indicator("")
daysAgoInput = input.int(1, tooltip = "Use negative values to offset in the future")
if barstate.islast
    MS_IN_ONE_DAY = 24 * 60 * 60 * 1000
    oneDayAgo = time - (daysAgoInput * MS_IN_ONE_DAY)
    label.new(oneDayAgo, high, xloc = xloc.bar_time, style = label.style_label_right)
```

请注意，由于市场休市时的时间间隔和缺失柱的变化不同，标签的位置可能并不总是准确的。此类时间偏移在24x7市场上往往更可靠。

您还可以使用值的柱索引进行偏移 `x`，例如：

```
label.new(bar_index + 10, high)
label.new(bar_index - 10, high[10])
label.new(bar_index[10], high[10])
```

## 读取标签属性

以下`getter`函数可用于标签：

- [标签.get\\_x\(\)](#)
- [标签.get\\_y\(\)](#)
- [标签.get\\_text\(\)](#)

他们都有一个相似的签名。[label.get\\_text\(\)](#)的一个是：

```
label.get_text(id) → series string
```

其中 `id` 是要检索其文本的标签。

## 克隆标签

`label.copy()` 函数用于克隆标签。其语法为：

```
label.copy(id) → void
```

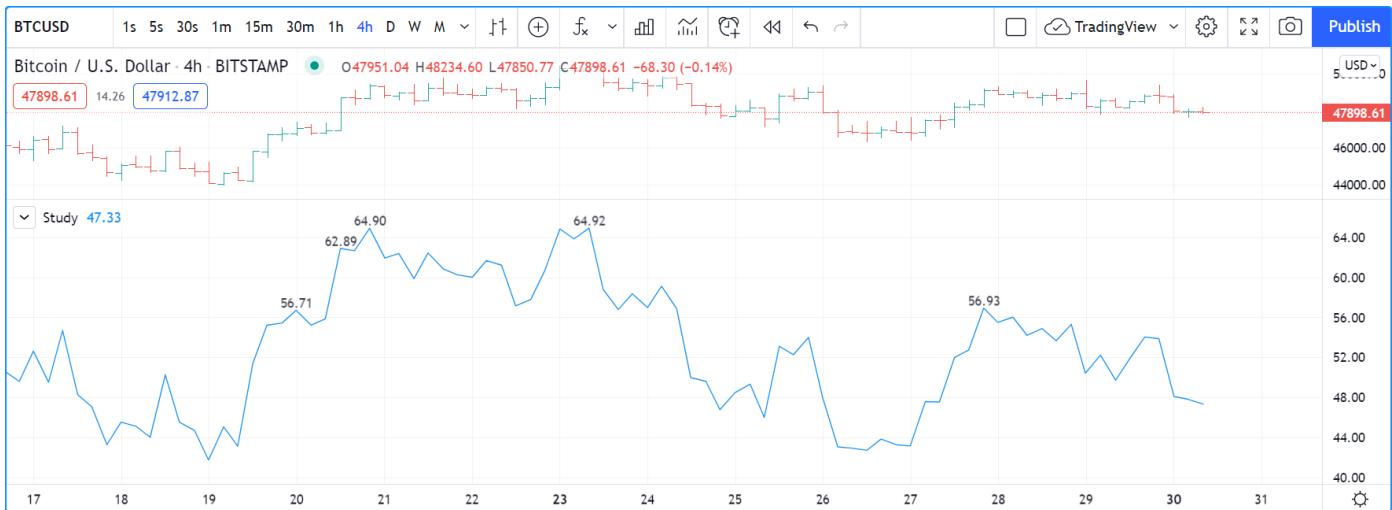
## 删除标签

`label.delete()` 函数用于删除标签。其语法为：

```
label.delete(id) → void
```

要仅在图表上保留用户定义数量的标签，可以使用如下代码：

```
//@version=5
MAX_LABELS = 500
indicator("", max_labels_count = MAX_LABELS)
qtyLabelsInput = input.int(5, "Labels to keep", minval = 0, maxval = MAX_LABELS)
myRSI = ta.rsi(close, 20)
if myRSI > ta.highest(myRSI, 20)[1]
    label.new(bar_index, myRSI, str.tostring(myRSI, "#.00"), style = label.style_none)
    if array.size(label.all) > qtyLabelsInput
        label.delete(array.get(label.all, 0))
plot(myRSI)
```



注意：

- 我们定义一个 `MAX_LABELS` 常量来保存脚本可以容纳的最大标签数量。我们使用该值在我们的 `indicator()` 调用中设置 `max_labels_count` 参数的值，并且也作为我们在 `input.int()` 调用中的值来限制用户值。`maxval`
- 当 RSI 突破最后 20 根柱线的最高值时，我们创建一个新标签。注意 [1] 我们在 中使用的偏移量。这是必要的。如果没有它，`ta.highest()` 返回的值将始终包含 的当前值，因此永远不会高于函数的返回值。`if myRSI > ta.highest(myRSI, 20)[1]` `myRSI` `myRSI`
- 之后，我们删除 `label.all` 数组中最旧的标签，该数组由 Pine Script™ 运行时自动维护，并包含我们的脚本绘制的所有可见标签的 ID。我们使用 `array.get()` 函数来检索索引零处的数组元素（最旧的可见标签 ID）。然后我们使用 `label.delete()` 删除与该 ID 链接的标签。

请注意，如果只想将标签放置在最后一个柱上，则当脚本在所有柱上执行时创建和删除标签是不必要且低效的，因此仅保留最后一个标签：

```
// INEFFICIENT!
//@version=5
indicator("", "", true)
lbl = label.new(bar_index, high, str.tostring(high, format.mintick))
label.delete(lbl[1])
```

这是实现相同任务的有效方法：

```
//@version=5
indicator("", "", true)
if barstate.islast
    // Create the label once, the first time the block executes on the last bar.
    var lbl = label.new(na, na)
    // On all iterations of the script on the last bar, update the label's information.
    label.set_xy(lbl, bar_index, high)
    label.set_text(lbl, str.tostring(high, format.mintick))
```

## 实时行为

标签会受到提交和回滚操作的影响，这会影响脚本在实时栏中执行时的行为。请参阅有关 Pine Script™ 执行模型的页面。

该脚本演示了在实时栏运行时回滚的效果：

```
//@version=5
indicator("", "", true)
label.new(bar_index, high)
```

在实时柱上，[label.new\(\)](#) 在每次脚本更新时创建一个新标签，但由于回滚过程，同一柱上先前更新创建的标签将被删除。只有在实时柱关闭之前创建的最后一个标签才会被提交，从而持续存在。

## 时间

### 介绍

#### 四个参考文献

在 Pine Script™ 中使用日期和时间值时，有四种不同的参考发挥作用：

1. **UTC 时区**：Pine Script™ 中时间值的本机格式是**Unix 时间（以毫秒为单位）**。 Unix 时间是自**1970 年 1 月 1 日 Unix Epoch**以来经过的时间。请参阅[此处了解当前 Unix 时间（以秒为单位）](#)，并参阅[此处了解有关 Unix 时间](#)的更多信息。 Unix 时间的值称为**时间戳**。 Unix 时间戳始终以 UTC（或“GMT”或“GMT+0”）时区表示。它们是根据固定参考（即 Unix 纪元）测量的，并且不随时区变化。一些内置程序使用 UTC 时区作为参考。
2. **交易所时区**：交易者的第二个与时间相关的关键参考是交易工具的交易所的时区。一些内置函数例如默认情况下以交易所时区返回[小时值](#)。
3. **timezone 参数**：某些通常返回交易所时区值的函数（例如[hour\(\)](#)）包含一个 `timezone` 参数，允许您将函数的结果调整为另一个时区。其他函数（如[time\(\)](#)）包含 `session` 和 `timezone` 参数。在这些情况下，`timezone` 参数适用于 `session` 参数的解释方式，而不是函数返回的时间值。
4. **图表时区**：这是用户使用“图表设置/交易品种/时区”字段从图表中选择的时区。此设置仅影响图表上日期和时间的显示。它不会影响 Pine 脚本的行为，并且它们对此设置不可见。

在讨论变量或函数时，我们会注意它们是否返回 UTC 或交换时区的日期或时间。脚本无法查看用户图表上的时区设置。

## 时间内置

Pine Script™ 具有内置变量，用于：

- 从当前柱（UTC 时区）获取时间戳信息：[time](#) 和 [time\\_close](#)
- 获取当前交易日开始的时间戳信息（UTC 时区）：[time\\_tradingday](#)
- 以一秒为增量获取当前时间（UTC 时区）：[timenow](#)

- 从栏检索日历和时间值（交换时区）：[year](#)、[month](#)、[weekofyear](#)、[dayofmonth](#)、[dayofweek](#)、[hour](#)、[min](#)和[Second](#)
- [使用syminfo.timezone](#)返回图表交易品种的时区

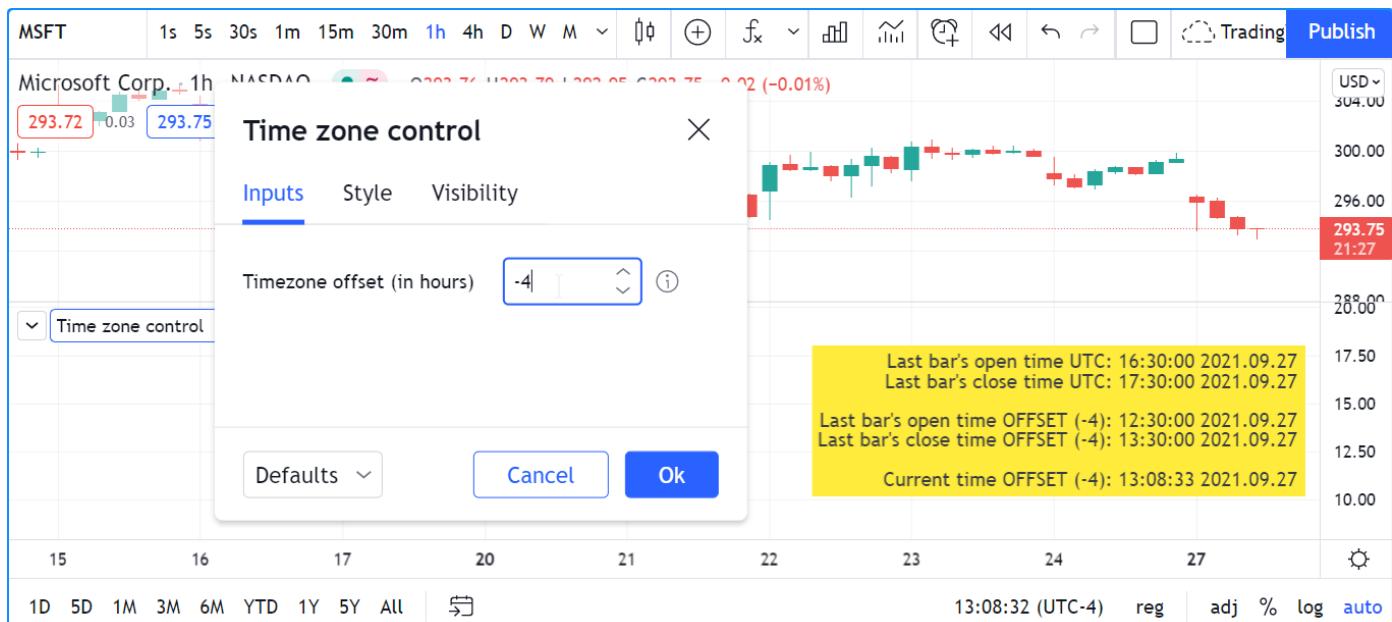
还有一些内置函数可以：

- [使用time\(\)](#)和[time\\_close\(\)](#)从其他时间帧返回柱的时间戳，无需调用request.security()
- 从任何时间戳检索日历和时间值，可以使用时区进行偏移：[year\(\)](#)、[month\(\)](#)、[weekofyear\(\)](#)、[dayofmonth\(\)](#)、[dayofweek\(\)](#)、[hour\(\)](#)、[min\(\)](#)和[Second\(\)](#)
- [使用timestamp\(\)](#)创建时间戳
- [使用str.format\(\)](#)将时间戳转换为格式化的日期/时间字符串以供显示
- 输入数据和时间值。请参阅有关[输入](#)的部分。
- [使用会话信息](#)。

## 时区

TradingViewers 可以更改用于在其图表上显示柱时间的时区。 Pine 脚本对此设置不可见。虽然有一个[syminfo.timezone](#) 变量可以返回图表工具交易的交易所的时区，但没有等效 `chart.timezone` 的变量。

当在图表上显示时间时，这显示了一种为用户提供将脚本的时间值调整为图表时间值的方法。这样，您显示的时间就可以与交易者在图表上使用的时区相匹配：



```
//@version=5
indicator("Time zone control")
MS_IN_1H = 1000 * 60 * 60
TOOLTIP01 = "Enter your time zone's offset (+ or -), including a decimal fraction if needed."
hoursOffsetInput = input.float(0.0, "Timezone offset (in hours)", minval = -12.0,
maxval = 14.0, step = 0.5, tooltip = TOOLTIP01)

printTable(txt) =>
    var table t = table.new(position.middle_right, 1, 1)
```

```

    table.cell(t, 0, 0, txt, text_halign = text.align_right, bgcolor = color.yellow)

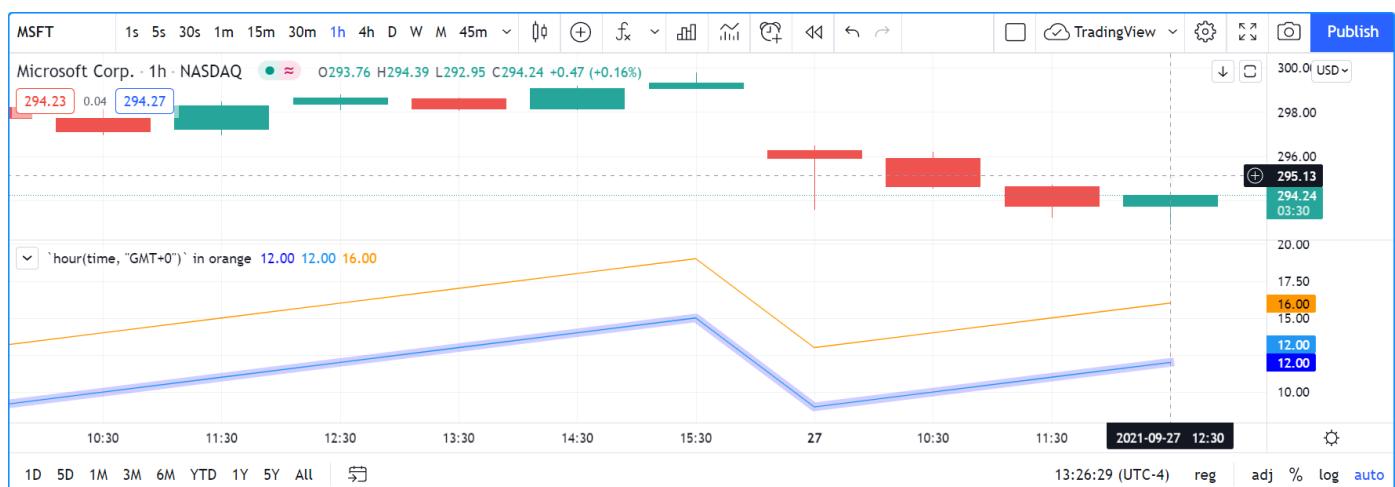
msOffsetInput = hoursOffsetInput * MS_IN_1H
printTable(
    str.format("Last bar's open time UTC: {0,date,HH:mm:ss yyyy.MM.dd}", time) +
    str.format("\nLast bar's close time UTC: {0,date,HH:mm:ss yyyy.MM.dd}", time_close) +
    str.format("\n\nLast bar's open time EXCHANGE: {0,date,HH:mm:ss yyyy.MM.dd}", time(timeframe.period, syminfo.session, syminfo.timezone)) +
    str.format("\nLast bar's close time EXCHANGE: {0,date,HH:mm:ss yyyy.MM.dd}", time_close(timeframe.period, syminfo.session, syminfo.timezone)) +
    str.format("\n\nLast bar's open time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}", hoursOffsetInput, time + msOffsetInput) +
    str.format("\nLast bar's close time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}", hoursOffsetInput, time_close + msOffsetInput) +
    str.format("\n\nCurrent time OFFSET ({0}): {1,date,HH:mm:ss yyyy.MM.dd}", hoursOffsetInput, timenow + msOffsetInput))

```

注意：

- 我们使用将以小时为单位的用户偏移量转换为毫秒 `msOffsetInput`。然后，我们将该偏移量添加到 UTC 格式的时间戳中，然后再将其转换为显示格式，例如和。`time + msOffsetInput` `timenow + msOffsetInput`
- 我们使用工具提示向用户提供说明。
- 我们提供 `minval` 和 `maxval` 值来保护输入字段，并提供 `step` 值 0.5，这样当他们使用字段的向上/向下箭头时，他们可以直观地找出可以使用分数。
- `str.format()` 函数格式化我们的时间值，即最后一根柱的时间和当前时间。

一些通常返回交易所时区值的函数提供了通过 `timezone` 参数将其结果适应另一个时区的方法。此脚本说明了如何使用[hour\(\)](#)执行此操作：



```
//@version=5
indicator(`hour(time, "GMT+0")` in orange)
color BLUE_LIGHT = #0000FF30
plot(hour, "", BLUE_LIGHT, 8)
plot(hour(time, syminfo.timezone))
plot(hour(time, "GMT+0"), "UTC", color.orange)
```

注意：

- hour变量和hour()函数通常返回交易所时区的值。因此，hour 和两者的图都呈蓝色重叠。因此，如果需要交换时间，则使用函数形式with是多余的。hour(time, syminfo.timezone)` `syminfo.timezone
- 然而，绘制的橙色线返回柱形图在UTC或“GMT+0”时间的时间，在本例中比交易所时间少四个小时，因为MSFT在时区为UTC-4的纳斯达克进行交易。hour(time, "GMT+0")

## 时区字符串

[time\(\)](#)、[timestamp\(\)](#)、[hour\(\)](#) `timezone`等函数中的参数使用的参数可以采用不同的格式，您可以在[IANA时区数据库名称参考页面](#)中找到这些格式。可以使用该页面表的“TZ数据库名称”、“UTC偏移±hh:mm”和“UTC DST偏移±hh:mm”列中的内容。

为了表示距UTC+5.5小时的偏移量，参考页中找到的这些字符串都是等效的：

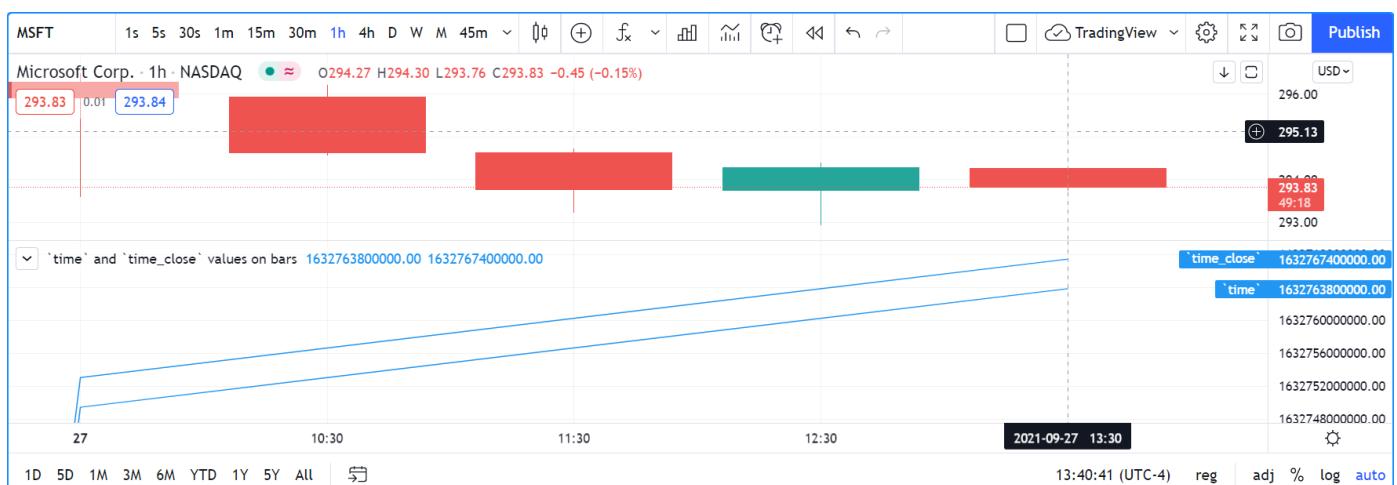
- "GMT+05:30"
- "Asia/Calcutta"
- "Asia/Colombo"
- "Asia/Kolkata"

非小数偏移量可以用以下形式表示 "GMT+5"。"GMT+5.5" 不允许。

## 时间变量

### time 和 time\_close

让我们首先绘制[time](#)和[time\\_close](#)，即柱状图开盘和收盘时间的 Unix 时间戳（以毫秒为单位）：



```
//@version=5
indicator(`time` and `time_close` values on bars")
plot(time, "`time`")
plot(time_close, "`time_close`")
```

注意：

- time和`time_close`变量返回UNIX时间中的时间戳，该时间戳与用户在图表上选择的时区无关。在这种情况下，图表的时区设置是交易所时区，因此无论图表上有什么符号，其交易所时区都将用于在图表光标上显示日期和时间值。纳斯达克的时区是UTC-4，但这仅影响图表的日期/时间值的显示；它不会影响脚本绘制的值。
- 比例尺中显示的绘图的最后一个时间值是从1970年1月1日00:00:00 UTC（世界标准时间）到酒吧开放时间所经过的毫秒数。它对应于2021年9月27日的17:30。但是，由于该图表使用UTC-4时区（纳斯达克时区），因此它显示的是13:30时间，比UTC时间早四个小时。
- 最后一根柱上两个值之间的差异是一小时内的毫秒数( $1000 * 60 * 60 = 3,600,000$ )，因为我们使用的是1H图表。

## 时间交易日

当品种在不同日历日开始和结束的隔夜交易时段进行交易时，`time_tradingday`非常有用。例如，这种情况发生在外汇市场中，其中一个交易时段可以在周日17:00开盘，并在周一17:00收盘。

当在1D或更小的时间范围内使用时，该变量返回以UNIX时间表示的交易日开始时间。当用于高于1D的时间范围时，它将返回柱中最后一个交易日的开始时间（例如，在1W时，它将返回本周最后一个交易日的开始时间）。

## 现在时间

`timenow`返回UNIX时间中的当前时间。它可以实时工作，也可以在历史柱上执行脚本时工作。实时地，您的脚本仅在执行提要更新时才会感知更改。当没有发生更新时，脚本处于空闲状态，因此无法更新其显示。有关更多信息，请参阅Pine Script™执行模型页面。

该脚本使用`timenow`和`time_close`的值来计算日内柱的实时倒计时。与图表上的倒计时相反，只有当提要更新导致脚本执行另一次迭代时，此倒计时才会更新：

```
//@version=5
indicator("", "", true)

printTable(txt) =>
    var table t = table.new(position.middle_right, 1, 1)
    table.cell(t, 0, 0, txt, text_halign = text.align_right, bgcolor = color.yellow)

printTable(str.format("{0,time,HH:mm:ss.SSS}", time_close - timenow))
```

## 日历日期和时间

日历日期和时间变量（例如`year`、`month`、`weekofyear`、`dayofmonth`、`dayofweek`、`hour`、`min`和`sec`）可用于测试特定日期或时间，并作为`timestamp()`的参数。

在测试特定日期或时间时，需要考虑脚本在无法检测到测试条件的时间范围内执行的可能性，或者在不存在具有特定要求的柱的情况下。例如，假设我们想要检测该月的第一个交易日。此脚本显示了当使用周图表或当月 1 日没有交易发生时，仅使用`dayofmonth`不起作用：



```
//@version=5
indicator("", "", true)
firstDayIncorrect = dayofmonth == 1
firstDay = ta.change(time("M"))
plotchar(firstDayIncorrect, "firstDayIncorrect", "•", location.top, size = size.small)
bgcolor(firstDay ? color.silver : na)
```

注意：

- 使用`ta.change(time("M"))`更加稳健，因为它适用于所有月份（#1 和 #2），显示为银色背景，而检测到使用的蓝点在 9 月第一个交易日发生在 2 号时不起作用（#1）。`dayofmonth == 1`
- 该条件将出现在该月第一天的所有内部柱上，但仅出现在第一天。`dayofmonth == 1``true``ta.change(time("M"))``true`

如果您希望脚本仅在 2020 年及以后显示，您可以使用：

```
//@version=5
indicator("", "", true)
plot(year >= 2020 ? close : na, linewidth = 3)
```

## [syminfo.timezone\(\)](#)

`syminfo.timezone` 返回图表交易品种的时区。当`timezone`函数中的参数可用时，并且您想明确指出您正在使用交易所的时区，这会很有帮助。它通常是多余的，因为当没有向`timezone`提供参数时，将假定交换的时区。

## 时间函数

## [time\(\)](#) 和 [time\\_close\(\)](#)

time () 和 [time\\_close\(\)](#) 函数具有以下签名：

```
time(timeframe, session, timezone) → series int  
time_close(timeframe, session, timezone) → series int
```

他们接受三个论点：

- `timeframe`

[timeframe.period](#) 格式的字符串。

- `session`

会话规范格式的可选字符串： "hhmm–hhmm[:days]"，其中该 [:days] 部分是可选的。请参阅[会话](#)页面以获取更多信息。

- `timezone`

`session`一个可选值，用于限定使用时的参数。

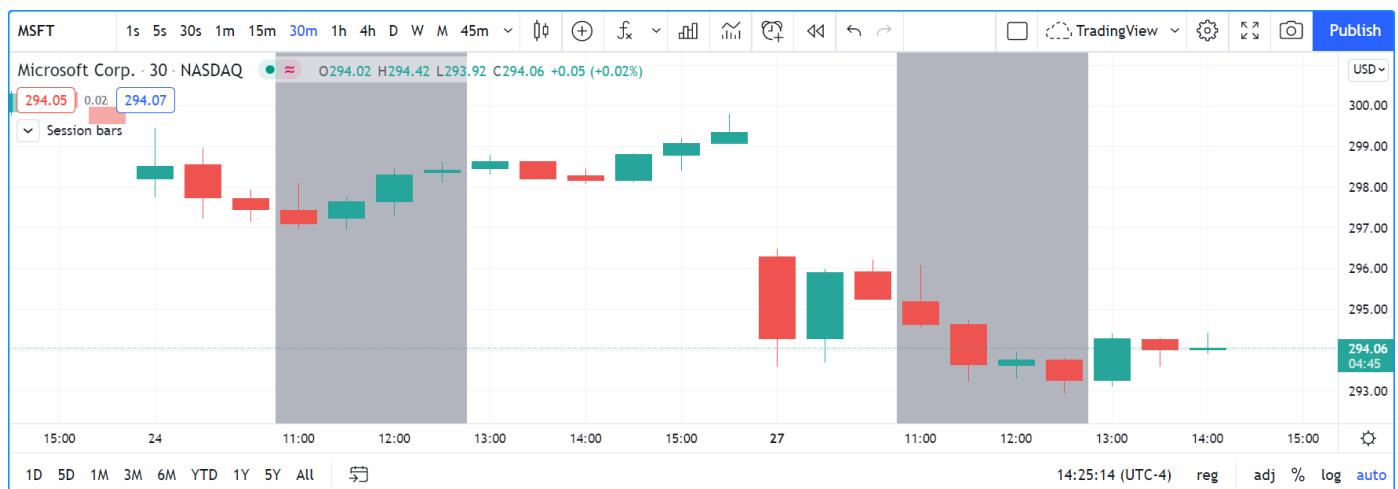
有关详细信息，请参阅《参考手册》中的[time\(\)](#)和[time\\_close\(\)](#)条目。

time () 函数最常用于：

1. 测试柱是否处于特定时间段内，这需要使用该 `session` 参数。在这些情况下，`timeframe.period` 通常将，即图表的时间范围用作第一个参数。当以这种方式使用该函数时，我们依赖这样一个事实：当柱形图不属于参数中指定的周期时，它将返回 `na session`。
2. 通过使用参数的更高时间范围来检测比图表更高的时间范围内的变化 `timeframe`。当为此目的使用该函数时，我们正在寻找返回值的变化，这意味着较高的时间范围柱已发生变化。这通常需要使用 `ta.change()` 进行测试，例如，`ta.change(time("D"))` 当新的更高时间范围柱出现时，将返回时间变化，因此在条件表达式中使用时，表达式的结果将转换为“bool”值。`true` 当有变化和 `false` 没有变化时，结果将是“bool”。

## 测试会话

让我们看一下第一种情况的示例，我们想要确定柱的开始时间是否属于 11:00 到 13:00 之间的时间段的一部分：



```
//@version=5
indicator("Session bars", "", true)
inSession = not na(time(timeframe.period, "1100-1300"))
bgcolor(inSession ? color.silver : na)
```

注意：

- 我们使用，它表示：“如果当前柱的开盘时间在 11:00 到 13:00 之间（含 11:00 和 13:00），则检查图表的时间范围”。如果柱在会话中，该函数将返回其开盘时间。如果不是，该函数返回 `na`。`time(timeframe.period, "1100-1300")`
- [我们有兴趣识别`time\(\)`不返回`na`的实例](#)，因为这意味着柱形图位于会话中，因此我们测试。当`time()`不是`na`时，我们不使用它的实际返回值；我们只关心它是否返回`na`。`not na(...)`

## 测试更高时间范围内的变化

在较长的时间范围内检测变化通常很有帮助。例如，您可能希望在日内图表上检测交易日的变化。对于这些情况，您可以使用返回 1D 柱的开盘时间的事实 `time("D")`，即使图表处于日内时间范围（例如 1H）：



```
//@version=5
indicator("", "", true)
bool newDay = ta.change(time("D"))
bgcolor(newDay ? color.silver : na)

newExchangeDay = ta.change(dayofmonth)
plotchar(newExchangeDay, "newExchangeDay", "□", location.top, size = size.small)
```

注意：

- 该 `newDay` 变量检测一维柱线开盘时间的变化，因此它遵循图表符号的约定，即使用 17:00 到 17:00 的隔夜时段。当新会话进入时它会更改值。
- 因为 `newExchangeDay` 检测 日历日中的`dayofmonth`的变化，所以当图表上的日期变化时它也会变化。

- 只有当有几天没有交易时，这两种变化检测方法才会在图表上重合。例如，在周日，两种检测方法都会检测到变化，因为日历日从最后一个交易日（周五）更改为新一周的第一个日历日（周日），即周一隔夜交易时段从17:00开始。

## 日历日期和时间

日历日期和时间函数，例如[year\(\)](#)、[month\(\)](#)、[weekofyear\(\)](#)、[dayofmonth\(\)](#)、[dayofweek\(\)](#)、[hour\(\)](#)、[min\(\)](#)和[second\(\)](#)可用于测试特定日期或时间。它们都具有与此处所示的[dayofmonth\(\)](#)类似的签名：

```
dayofmonth(time) → series int
dayofmonth(time, timezone) → series int
```

这将绘制柱线开盘日，其中2021年1月1日00:00时间位于其[time](#)和[time\\_close](#)值之间：

```
//@version=5
indicator("")
exchangeDay = dayofmonth(timestamp("2021-01-01"))
plot(exchangeDay)
```

该值将是31日或1日，具体取决于图表符号上会话开始的日历日。使用UTC时区的交易所24x7交易品种的日期将为第一天。对于在UTC-4交易所交易的品种，该日期将为31日。

## 时间戳()

[timestamp\(\)](#)函数有几个不同的签名：

```
timestamp(year, month, day, hour, minute, second) → simple/series int
timestamp(timezone, year, month, day, hour, minute, second) → simple/series int
timestamp(dateString) → const int
```

前两者之间的唯一区别是[timezone](#)参数。它的默认值为[syminfo.timezone](#)。有关有效值，请参阅本页的[时区字符串部分](#)。

第三种形式用作[input.time\(\) defval](#)中的值。有关详细信息，请参阅参考手册中的[timestamp\(\)条目](#)。

[timestamp\(\)](#)对于生成特定日期的时间戳非常有用。要生成2021年1月1日的时间戳，请使用以下方法之一：

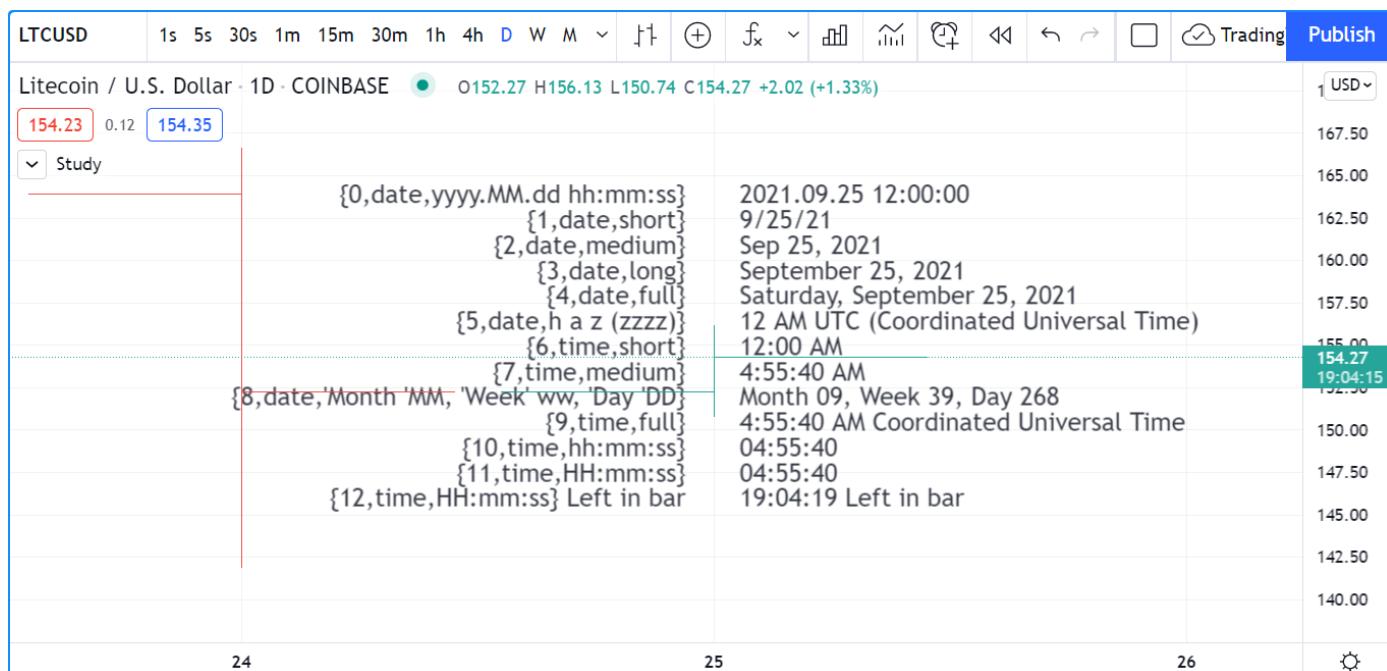
```
//@version=5
indicator("")
yearBeginning1 = timestamp("2021-01-01")
yearBeginning2 = timestamp(2021, 1, 1, 0, 0)
printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
0, 0, txt, bgcolor = color.yellow)
printTable(str.format("yearBeginning1: {0,date,yyyy.MM.dd hh:mm}\nyearBeginning2:
{1,date,yyyy.MM.dd hh:mm}", yearBeginning1, yearBeginning2))
```

您可以在`timestamp()`参数中使用偏移量。在这里，我们从为其`day`参数提供的值中减去 2，以获取图表两天前最后一个柱的日期/时间。请注意，由于各种工具上的柱线对齐方式不同，图表上标识的柱线可能并不总是正好位于 48 小时之外，尽管函数的返回值是正确的：

```
//@version=5
indicator("")
twoDaysAgo = timestamp(year, month, dayofmonth - 2, hour, minute)
printTable(txt) => var table t = table.new(position.middle_right, 1, 1), table.cell(t,
0, 0, txt, bgcolor = color.yellow)
printTable(str.format("{0,date,yyyy.MM.dd hh:mm}", twoDaysAgo))
```

## 设置日期和时间的格式

可以使用`str.format()`格式化时间戳。这些是各种格式的示例：



```
//@version=5
indicator("", "", true)

print(txt, styl) =>
    var alignment = styl == label.style_label_right ? text.align_right :
text.align_left
    var lbl = label.new(na, na, "", xloc.bar_index, yloc.price, color(na), styl,
color.black, size.large, alignment)
    if barstate.islast
        label.set_xy(lbl, bar_index, h12[1])
        label.set_text(lbl, txt)

var string format =
    "{0,date,yyyy.MM.dd hh:mm:ss}\n" +
    "{1,date,short}\n" +
    "{2,date,medium}\n" +
```

```

" {3,date,long}\n" +
" {4,date,full}\n" +
" {5,date,h a z (zzzz)}\n" +
" {6,time,short}\n" +
" {7,time,medium}\n" +
" {8,date,'Month 'MM, 'Week' ww, 'Day 'DD}\n" +
" {9,time,full}\n" +
" {10,time,hh:mm:ss}\n" +
" {11,time,HH:mm:ss}\n" +
" {12,time,HH:mm:ss} Left in bar\n"

print(format, label.style_label_right)
print(str.format(format,
    time, time, time, time, time, time, time,
    timenow, timenow, timenow, timenow,
    timenow - time, time_close - timenow), label.style_label_left)

```

# 时间框架

## 介绍

图表的时间范围有时也称为其间隔或分辨率。它是图表上用一根柱表示的时间单位。所有标准图表类型都使用时间范围：“条形”、“蜡烛”、“空心蜡烛”、“线”、“面积”和“基线”。一种非标准图表类型也使用时间范围：“Heikin Ashi”。

对访问多个时间范围内的数据感兴趣的程序员需要熟悉时间范围在 Pine Script™ 中的表达方式以及如何使用它们。

**时间范围字符串**在不同的上下文中发挥作用：

- 当从另一个交易品种和/或时间范围请求数据时，必须在[request.security\(\)](#)中使用它们。请参阅[其他时间范围和数据](#)页面以探索[request.security\(\)](#)的使用。
- 它们可以用作[time\(\)](#)和[time\\_close\(\)](#)函数的参数，以返回较高时间范围柱的时间。反过来，这可以用于检测图表时间范围内较高时间范围的变化，而无需使用[request.security\(\)](#)。请参阅[测试较高时间范围内的更改](#)部分以了解如何执行此操作。
- [input.timeframe\(\)](#)函数提供了一种允许脚本用户通过脚本的“输入”选项卡定义时间范围的方法（有关更多信息，请参阅[时间范围输入](#)部分）。
- [Indicator\(\)](#)声明语句有一个可选 `timeframe` 参数，可用于为简单脚本提供多时间范围功能，而无需使用[request.security\(\)](#)。
- 许多内置变量提供有关脚本运行所在图表所使用的时间范围的信息。有关它们的更多信息，请参阅[图表时间范围部分，包括timeframe.period](#)，它返回 Pine Script™ 时间范围规范格式的字符串。

## 时间范围字符串规范

时间范围字符串遵循以下规则：

- 它们由乘数和时间单位组成，例如“1S”、“30”（30 分钟）、“1D”（一天），“3M”（三个月）。

- 单位由单个字母表示，分钟不使用字母：“S”表示秒，“D”表示天，“W”表示周，“M”表示月。
- 当不使用乘数时，假设为1：“S”相当于“1S”，“D”相当于“1D”等。如果仅使用“1”，则被解释为“1min”，因为没有单位字母标识符使用几分钟。
- 没有“小时”单位；“1H”无效。一小时的正确格式是“60”（记住没有为分钟指定单位字母）。
- 每个时间范围单位的有效乘数有所不同：
  - 对于秒，只有离散的1、5、10、15和30乘法器有效。
  - 分钟为1到1440。
  - 天数为1到365。
  - 周数为1到52。
  - 月份为1至12。

## 比较时间范围

比较不同的时间范围字符串可能很有用，例如，可以确定图表上使用的时间范围是否低于脚本中使用的较高时间范围。

将时间范围字符串转换为以小数分钟为单位的表示形式提供了一种使用通用单位来比较它们的方法。此脚本使用[timeframe.in\\_seconds\(\)](#)函数将时间范围转换为浮点数秒，然后将结果转换为分钟：

```
//@version=5
indicator("Timeframe in minutes example", "", true)
string tfInput = input.timeframe(defval = "", title = "Input TF")

float chartTFInMinutes = timeframe.in_seconds() / 60
float inputTFInMinutes = timeframe.in_seconds(tfInput) / 60

var table t = table.new(position.top_right, 1, 1)
string txt = "Chart TF: " + str.tostring(chartTFInMinutes, "#.##### minutes") +
"\nInput TF: " + str.tostring(inputTFInMinutes, "#.##### minutes")
if barstate.isfirst
    table.cell(t, 0, 0, txt, bgcolor = color.yellow)
else if barstate.islast
    table.cell_set_text(t, 0, 0, txt)

if chartTFInMinutes > inputTFInMinutes
    runtime.error("The chart's timeframe must not be higher than the input's
timeframe.")
```

注意：

- 我们使用内置的[timeframe.in\\_seconds\(\)](#)函数将图表和[input.timeframe\(\)](#)函数转换为秒，然后除以60转换为分钟。

- 我们在和变量的初始化中使用了两次对`timeframe.in_seconds()`函数的调用。在第一个实例中，我们不为其参数提供参数，因此该函数返回图表的时间范围（以秒为单位）。在第二次调用中，我们通过调用`input.timeframe()`提供脚本用户选择的时间范围。
  - `chartTFInMinutes``inputTFInMinutes``timeframe`
- 接下来，我们验证时间范围以确保输入时间范围等于或高于图表的时间范围。如果不是，我们会生成一个运行时错误。
- 我们最终打印转换为分钟的两个时间范围值。

# 风格指南

## 介绍

本风格指南提供了有关如何命名变量以及如何以有效的标准方式组织 Pine 脚本的建议。遵循我们最佳实践的脚本将更易于阅读、理解和维护。

您可以查看使用平台上的[TradingView](#)和[PineCoders](#)帐户发布的这些指南的脚本。

## 命名约定

我们建议使用：

- `camelCase` 对于所有标识符，即变量或函数名称：`ma`, `maFast`, `maLengthInput`, `maColor`, `roundedOHLC()`, `pivotHi()`。
- `SNAKE_CASE` 常量全部大写：`BULL_COLOR`, `BEAR_COLOR`, `MAX_LOOKBACK`.
- 当限定后缀提供有关变量的类型或出处的有价值的线索时使用：`maShowInput`, `bearColor`, `bearColorInput`, `volumesArray`, `maPlotID`, `resultsTable`, `levelsColorArray`。

## 脚本组织

Pine Script™ 编译器对脚本中特定语句或版本[编译器注释](#)的定位相当宽容。虽然其他安排在语法上是正确的，但我们建议这样组织脚本：

```
<license>
<version>
<declaration_statement>
<import_statements>
<constant_declarations>
<inputs>
<function_declarations>
<calculations>
<strategy_calls>
<visuals>
<alerts>
```

## <许可证>

如果您在 TradingView 上公开发布开源脚本（脚本也可以私下发布），则您的开源代码默认受 Mozilla 许可证保护。您可以选择您喜欢的任何其他许可证。

[这些脚本中代码的重用受我们的脚本发布内部规则](#)的约束，该规则优先于作者的许可。

出现在脚本开头的标准许可证注释是：

```
// This source code is subject to the terms of the Mozilla Public License 2.0 at  
https://mozilla.org/MPL/2.0/  
// © username
```

## <版本>

这是定义脚本将使用的 Pine Script™ 版本的[编译器注释](#)。如果不存在，则使用 v1。对于 v5，请使用：

```
//@version=5
```

## <声明语句>

这是定义脚本类型的强制声明语句。它必须是对[Indicator\(\)](#)、[Strategy\(\)](#)或[Library\(\)](#)的调用。

## <导入语句>

如果您的脚本使用一个或多个 Pine Script™ 库，则您的[导入](#)语句属于此处。

## <常量声明>

脚本可以声明限定为“const”的变量，即引用常量值的变量。

当变量满足以下条件时，我们将变量称为“常量”：

- 它们的声明使用可选关键字（有关更多信息，`const` 请参阅我们的用户手册中有关[类型限定符的部分](#)）。
- 它们使用文字（例如，`100` 或 `"AAPL"`）或内置限定为“const”（例如，`color.green`）来初始化。
- 它们的值在脚本执行期间不会改变。

我们用来 `SNAKE_CASE` 命名这些变量并将它们的声明分组在脚本顶部附近。例如：

```
// —— Constants  
int    MS_IN_MIN    = 60 * 1000  
int    MS_IN_HOUR   = MS_IN_MIN * 60  
int    MS_IN_DAY    = MS_IN_HOUR * 24  
  
color  GRAY        = #808080ff  
color  LIME        = #00FF00ff  
color  MAROON      = #800000ff  
color  ORANGE      = #FF8000ff  
color  PINK        = #FF0080ff
```

```

color  TEAL      = #008080ff
color  BG_DIV    = color.new(ORANGE, 90)
color  BG_RESETS = color.new(GRAY, 90)

string RST1      = "No reset; cumulate since the beginning of the chart"
string RST2      = "On a stepped higher timeframe (HTF)"
string RST3      = "On a fixed HTF"
string RST4      = "At a fixed time"
string RST5      = "At the beginning of the regular session"
string RST6      = "At the first visible chart bar"
string RST7      = "Fixed rolling period"

string LTF1      = "Least precise, covering many chart bars"
string LTF2      = "Less precise, covering some chart bars"
string LTF3      = "More precise, covering less chart bars"
string LTF4      = "Most precise, 1min intrabars"

string TT_TOTVOL = "The 'Bodies' value is the transparency of the total volume
candle bodies. Zero is opaque, 100 is transparent."
string TT_RST_HTF = "This value is used when '" + RST3 + "' is selected."
string TT_RST_TIME = "These values are used when '" + RST4 + "' is selected.

A reset will occur when the time is greater or equal to the bar's open time, and less
than its close time.\nHour: 0-23\nMinute: 0-59"
string TT_RST_PERIOD = "This value is used when '" + RST7 + "' is selected."

```

在这个例子中：

- 和常量将用作调用参数中 `RST*` 的元组元素。`LTF*``options``input.*()`
- 这些 `TT_*` 常量将用作调用 `tooltip` 中的参数 `input.*()`。请注意我们如何对长字符串文字使用续行。
- 我们不使用 `var` 来初始化常量。Pine Script™ 运行时经过优化，可以处理每个柱上的声明，但仅在第一次声明变量时 使用 `var` 来初始化变量，这会对脚本性能造成较小的损失，因为 `var` 变量需要在更多柱上进行维护。

注意：

- 脚本中多处使用的文字应始终声明为常量。如果给定一个有意义的名称，则使用常量而不是文字可以使其更具可读性，并且这种做法使代码更易于维护。尽管一天中的毫秒数将来不太可能改变，但 `MS_IN_DAY` 比更有意义。`1000 * 60 * 60 * 24`
- 仅在函数的本地块或 `if`、`while` 等语句中使用的常量可以在该本地块中声明。

## <输入

当所有输入都位于同一代码段中时，读取脚本会容易得多。将该部分放在脚本的开头还反映了它们在运行时（即在执行脚本的其余部分之前）的处理方式。

输入变量名称后缀 `input` 使它们在稍后在脚本中使用时更容易识别：

`maLengthInput`、`bearColorInput`、`showAvgInput` 等。

```

// ----- Inputs
string resetInput           = input.string(RST2,          "CVD Resets",
                                             inline = "00", options = [RST1, RST2, RST3, RST4, RST5, RST6, RST7])
string fixedTfInput         = input.timeframe("D",        " Fixed HTF: ",
                                             tooltip = TT_RSTHTF)
int    hourInput             = input.int(9,              " Fixed time hour: ",
                                             inline = "01", minval = 0, maxval = 23)
int    minuteInput           = input.int(30,             "minute",
                                             inline = "01", minval = 0, maxval = 59, tooltip = TT_RSTTIME)
int    fixedPeriodInput      = input.int(20,             " Fixed period: ",
                                             inline = "02", minval = 1, tooltip = TT_RSTPERIOD)
string ltfModeInput         = input.string(LTF3,          "Intrabar precision",
                                             inline = "03", options = [LTF1, LTF2, LTF3, LTF4])

```

## <函数声明>

所有用户定义的函数必须在脚本的全局范围内定义； Pine Script™ 中不允许嵌套函数定义。

最佳函数设计应尽量减少函数范围内全局变量的使用，因为它们会破坏函数的可移植性。当无法避免时，这些函数必须遵循代码中的全局变量声明，这意味着它们不能总是放置在 `<function_declarations>` 部分中。理想情况下，这种对全局变量的依赖性应该记录在函数的注释中。

如果您记录该函数的目标、参数和结果，也会对读者有所帮助。[库](#)中使用的相同语法可用于记录您的函数。如果您决定这样做，这可以使您更轻松地将函数移植到库中：

```

//@version=5
indicator("<function_declarations>", "", true)

string SIZE_LARGE  = "Large"
string SIZE_NORMAL = "Normal"
string SIZE_SMALL  = "Small"

string sizeInput = input.string(SIZE_NORMAL, "Size", options = [SIZE_LARGE,
SIZE_NORMAL, SIZE_SMALL])

// @function      Used to produce an argument for the `size` parameter in built-in
functions.
// @param userSize (simple string) User-selected size.
// @returns        One of the `size.*` built-in constants.
// Dependencies: SIZE_LARGE, SIZE_NORMAL, SIZE_SMALL
getSize(simple string userSize) =>
    result =
        switch userSize
            SIZE_LARGE  => size.large
            SIZE_NORMAL => size.normal
            SIZE_SMALL   => size.small
            => size.auto

```

```
if ta.rising(close, 3)
    label.new(bar_index, na, yloc = yloc.abovebar, style = label.style_arrowup, size =
getSize(sizeInput))
```

## <计算

这是脚本的核心计算和逻辑应该放置的地方。当变量声明放置在使用变量的代码段附近时，代码可以更容易阅读。一些程序员喜欢将所有非常量变量声明放在本节的开头，这并不总是适用于所有变量，因为有些变量可能需要在声明之前执行一些计算。

## <策略调用

当策略调用分组在脚本的同一部分时，策略更易于阅读。

## <视觉效果

理想情况下，此部分应包括生成脚本视觉效果的所有语句，无论它们是绘图、绘图、背景颜色、蜡烛图等。请参阅此处的 Pine Script™ 用户手册部分，了解有关视觉效果的相对深度如何的更多信息决定。

## <警报>

警报代码通常要求脚本的计算在其之前执行，因此将其放在脚本的末尾是有意义的。

## 间距

除一元运算符 () 外，所有运算符两侧均应使用空格 -1。还建议在所有逗号之后以及使用命名函数参数时使用空格，如下所示：`plot(series = close)`

```
int a = close > open ? 1 : -1
var int newLen = 2
newLen := min(20, newlen + 1)
float a = -b
float c = d > e ? d - e : d
int index = bar_index % 2 == 0 ? 1 : 2
plot(close, color = color.red)
```

## 换行

换行可以使长行更易于阅读。换行是通过使用不是四的倍数的缩进级别来定义的，因为四个空格或制表符用于定义本地块。这里我们使用两个空格：

```
plot(
    series = close,
    title = "Close",
    color = color.blue,
    show_last = 10
)
```

## 垂直对齐

使用制表符或空格进行垂直对齐在包含许多类似行（例如常量声明或输入）的代码节中非常有用。他们可以使用 Pine 编辑器的多光标功能 (`ctrl+ alt+ ⌘/ ⌘`) 使批量编辑变得更加容易：

```
// Colors used as defaults in inputs.  
color COLOR_AQUA  = #0080FFff  
color COLOR_BLACK = #000000ff  
color COLOR_BLUE   = #013BCAff  
color COLOR_CORAL  = #FF8080ff  
color COLOR_GOLD   = #CCCC00ff
```

## 显式输入

在声明变量时包含变量的类型不是必需的，并且对于小脚本来说通常是过度的；我们没有系统地使用它。它可以使函数结果的类型更加清晰，并区分变量的声明（使用 `=`）与其重新赋值（使用 `:=`）。使用显式类型还可以让读者更轻松地在较大的脚本中找到自己的方式。

## 调试

### 介绍

TradingView 在 Pine Script™ 编辑器和图表之间紧密集成，可实现 Pine Script™ 代码的高效、交互式调试。一旦程序员了解了在每种情况下使用的最合适的技术，他们将能够快速、彻底地调试脚本。本页演示了调试 Pine Script™ 代码的最有用的技术。

如果您还不熟悉 Pine Script™ 的执行模型，请务必阅读本用户手册的执行模型页面，以便了解调试代码在 Pine Script™ 环境中的行为方式。

## 显示概况

Pine 脚本绘制的值可以显示在四个不同的位置：

1. 脚本名称旁边（由“图表设置/状态线”选项卡中的“指标值”复选框控制）。
2. 在脚本的窗格中，无论您的脚本是图表叠加层还是在单独的窗格中。
3. 在比例中（仅显示最后一个柱的值，并由“图表设置/比例”选项卡中的“指标最后值标签”复选框控制）。
4. 在数据窗口中（您可以使用图表右侧第四个图标将其调出）。

请注意前面屏幕截图中的以下内容：

- 图表的光标位于数据集的第一个条形图上，其中 `bar_index` 为零。该值反映在指标名称旁边和数据窗口中。将光标移动到其他条形图上会更新这些值，因此它们始终代表该条形图上的值。这是随着脚本执行逐条进行而检查变量值的好方法。

- `title` 我们的`plot()`调用的参数“条形索引”被用作数据窗口中值的图例。
- 数据窗口中显示的值的精度取决于图表符号的刻度值。您可以通过两种方式修改它：
  - 通过更改脚本的“设置/样式”选项卡中“精度”字段的值。使用此方法可以获得最多八位数字的精度。
  - `precision` 通过在脚本的`indicator()`或`strategy()`声明语句中使用参数。此方法允许指定高达 16 位的精度。
- 我们脚本中的`plot()`调用在指标窗格中绘制`bar_index`的值，该值显示了变量的增加值。
- 脚本窗格的比例会自动调整大小，以适应脚本中所有`plot()`调用绘制的最小和最大值。

## 显示数值

### 当脚本的规模不重要时

前面的屏幕截图中的脚本使用最简单的方法来检查数值：`plot()`调用，它在脚本的显示区域中绘制与变量值相对应的线。我们的示例脚本绘制了`bar_index`内置变量的值，其中包含柱的编号，该值在数据集的第一个柱上从零开始，并在每个后续柱上增加 1。我们使用了`plot()`调用来绘制要检查的变量，因为我们的脚本没有绘制任何其他内容；我们并没有专注于保留其他地块的比例以继续正常绘制。这是我们使用的脚本：

```
//@version=5
indicator("Plot `bar_index`")
plot(bar_index, "Bar Index")
```

### 当必须保留脚本的比例时

在脚本的显示区域中绘制值并不总是可行。当我们已经有其他绘图正在进行时，添加其值超出脚本绘图边界的变量的调试绘图会使绘图不可读，如果我们想保留其他绘图的比例，则必须使用另一种技术来检查值。

[假设我们想继续检查`bar\_index`的值，但这次是在脚本中绘制 RSI：](#)

```
//@version=5
indicator("Plot RSI and `bar_index`")
r = ta.rsi(close, 20)
plot(r, "RSI", color.black)
plot(bar_index, "Bar Index")
```

在包含大量条形的数据集上运行脚本会产生以下显示：

在哪里：

1. [黑色的 RSI 线是平坦的，因为它在 0 到 100 之间变化，但指标的窗格会缩放以显示`bar\_index`的最大值，即 25692.0000。](#)
2. [光标所在柱上的`bar\_index`值显示在指标名称旁边，并且其在脚本窗格中的蓝色图是平坦的。](#)
3. [比例尺中显示的 25692.0000 `bar\_index` 值表示其在最后一个柱上的值，因此数据集包含 25693 个柱。](#)

4. 光标所在柱上的`bar_index`值以及位于其上方的柱的 RSI 值也显示在数据窗口中。

为了保留 RSI 绘图，同时仍然能够检查值或`bar_index`，我们将使用`plotchar()`绘制变量，如下所示：

```
//@version=5
indicator("Plot RSI and `bar_index`")
r = ta.rsi(close, 20)
plot(r, "RSI", color.black)
plotchar(bar_index, "Bar index", "", location.top)
```

在哪里：

- 由于`bar_index`的值不再绘制在脚本的窗格中，因此窗格的边界现在是 RSI 的边界，可以正常显示。
- 使用`plotchar()`绘制的值显示在脚本名称旁边和数据窗口中。
- 我们没有使用`plotchar()`调用来绘制字符，因此第三个参数是空字符串( `""` )。我们还指定`location.top`作为`location`参数，这样我们就不会在显示区域边界的计算中考虑符号的价格。

## 显示字符串

必须使用 Pine Script™ 标签来显示字符串。标签仅出现在脚本的显示区域；标签中显示的字符串不会出现在数据窗口或其他任何地方。

### 每个栏上的标签

以下脚本演示了重复绘制显示符号名称的标签的最简单方法：

```
//@version=5
indicator("Simple label", "", true)
label.new(bar_index, high, syminfo.ticker)
```

默认情况下，图表上仅显示最后 50 个标签。您可以通过使用`max_labels_count`脚本的`indicator()`或`strategy()`声明语句中的参数将此数量增加到最大 500。例如：

```
indicator("Simple label", "", true, max_labels_count = 500)
```

### 最后一个栏上的标签

由于 Pine 脚本中操作的字符串通常不会更改条形，因此最常用的可视化方法是在数据集的最后一个条形上绘制标签。在这里，我们使用一个函数创建一个仅出现在图表最后一个柱上的标签。我们的`f_print()`函数只有一个参数，即要显示的文本字符串：

```

//@version=5
indicator("print()", "", true)
print(txt) =>
    // Create label on the first bar.
    var lbl = label.new(bar_index, na, txt, xloc.bar_index, yloc.price, color(na),
label.style_none, color.gray, size.large, text.align_left)
    // On next bars, update the label's x and y position, and the text it displays.
    label.set_xy(lbl, bar_index, ta.highest(10)[1])
    label.set_text(lbl, txt)

print("Multiplier = " + str.tostring(timeframe.multiplier) + "\nPeriod = " +
timeframe.period + "\nHigh = " + str.tostring(high))
print("Hello world!\n\n\n")

```

请注意我们最后一个代码示例中的以下内容：

- 我们使用该 `print()` 函数来封装标签绘制代码。虽然在每个柱上调用该函数，但仅在数据集的第一个柱上创建标签，因为我们在函数内声明变量时使用了 `var` 关键字。`lbl` 创建后，我们仅更新标签的 `x` 和 `y` 坐标及其每个连续条上的文本。如果我们不更新这些值，标签将保留在数据集的第一个条上，并且仅在该条上显示文本字符串的值。最后，请注意，我们使用垂直定位标签，通过使用前 `ta.highest(10)[1]` 10 个柱的最高点，我们可以防止标签在实时柱中移动。您可能需要在其他情况下调整此 `y` 位置。
- 我们调用该 `print()` 函数两次，以表明如果您进行多次调用，因为这样可以更轻松地调试多个字符串，您可以使用正确数量的换行符 (`\n`) 来分隔每个字符串来叠加它们的文本。
- 我们使用 `str.tostring()` 函数将数值转换为字符串，以便包含在要显示的文本中。

## 调试条件

### 单一条件

可以使用许多方法来显示满足条件的事件。此代码显示了识别 RSI 小于 30 的柱的六种方法：

```

//@version=5
indicator("Single conditions")
r = ta.rsi(close, 20)
rIsLow = r < 30
hline(30)

// Method #1: Change the plot's color.
plot(r, "RSI", rIsLow ? color.fuchsia : color.black)
// Method #2: Plot a character in the bottom region of the display.
plotchar(rIsLow, "rIsLow char at bottom", "▲", location.bottom, size = size.small)
// Method #3: Plot a character on the RSI line.
plotchar(rIsLow ? r : na, "rIsLow char on line", "•", location.absolute, color.red,
size = size.small)
// Method #4: Plot a shape in the top region of the display.
plotshape(rIsLow, "rIsLow shape", shape.arrowup, location.top)

```

```
// Method #5: Plot an arrow.
plotarrow(rIsLow ? 1 : na, "rIsLow arrow")
// Method #6: Change the background's color.
bgcolor(rIsLow ? color.new(color.green, 90) : na)
```

注意：

- 我们在布尔变量中定义条件 `rIsLow`，并在每个柱上对其进行评估。用于为变量赋值的表达式的计算结果为 `or`（或者当 `is` 时，如数据集第一条中的情况）。`r < 30``true``false``na``r``na`
- 方法 #1 根据条件使用 RSI 图的颜色变化。每当绘图的颜色发生变化时，它都会从前一栏开始为绘图着色。
- 方法 #2 使用 `plotchar()` 在指标显示的底部绘制一个向上的三角形。使用位置和字符的不同组合可以同时识别单个条上的多个条件。这是我们识别图表上的条件的首选方法之一。
- 方法 #3 还使用了 `plotchar()` 调用，但这次角色位于 RSI 线上。为了实现这一点，我们使用 `location.absolute` 和 Pine Script™ 的 `?:` 三元条件运算符来定义一个条件表达式，其中仅当条件为真时才使用 `rIsLow` 位置。如果不为 `true`，`na` 则使用，因此不显示任何字符。
- 当满足我们的条件时，方法 #4 使用 `plotshape()` 在指标显示区域的顶部绘制一个蓝色向上箭头。
- 当满足我们的条件时，方法 #5 使用 `plotarrow()` 在显示屏底部绘制一个绿色向上箭头。
- 当满足我们的条件时，方法 #6 使用 `bgcolor()` 更改背景颜色。再次使用三元运算符来评估我们的条件。`color.green` 当为 `true` 时它将返回 `rIsLow`，当为 `false` 或 `na` 时返回颜色（不为背景着色）。`rIsLow``na`
- 最后，请注意带有值的布尔变量如何在数据窗口中 `true` 显示。值由零值表示。`1``false`

## 复合条件

需要识别满足多个条件的情况的程序员必须通过使用 `and` 逻辑运算符聚合各个条件来构建复合条件。由于只有在各个条件正确触发的情况下，复合条件才会按预期执行，因此如果您在代码中使用复合条件之前验证各个条件的行为，您将会省去很多麻烦。

可以使用像这样的技术来显示多个单独条件的状态，其中使用四个单独条件来构建我们的 `bull` 复合条件：

```
//@version=5
indicator("Compound conditions")
periodInput = input.int(20)
bullLevelInput = input.int(55)

r = ta.rsi(close, periodInput)

// Condition #1.
rsiBull = r > bullLevelInput
// Condition #2.
hiChannel = ta.highest(r, periodInput * 2)[1]
aboveHiChannel = r > hiChannel
// Condition #3.
channelIsOld = hiChannel >= hiChannel[periodInput]
// Condition #4.
```

```

historyIsBull = math.sum(rsiBull ? 1 : -1, periodInput * 3) > 0
// Compound condition.
bull = rsiBull and aboveHiChannel and channelIsOld and historyIsBull

hline(bullLevelInput)
plot(r, "RSI", color.black)
plot(hiChannel, "High Channel")

plotchar(rsiBull ? bullLevelInput : na, "rIsBull", "1", location.absolute, color.green,
size = size.tiny)
plotchar(aboveHiChannel ? r : na, "aboveHiChannel", "2", location.absolute, size =
size.tiny)
plotchar(channelIsOld, "channelIsOld", "3", location.bottom, size = size.tiny)
plotchar(historyIsBull, "historyIsBull", "4", location.top, size = size.tiny)
bgcolor(bull ? not bull[1] ? color.new(color.green, 50) : color.new(color.green, 90) :
na)

```

注意：

- 我们使用 `plotchar()` 调用来显示每个条件的数字，注意将它们分布在指标的 y 空间上，这样它们就不会重叠。
- 前两个 `plotchar()` 调用使用绝对定位来放置条件编号，以便帮助我们记住相应的条件。例如，当 RSI 高于用户定义的牛市水平时，第一个显示“1”，将“1”定位在牛市水平上。
- 我们使用两种不同深浅的绿色来为背景着色：较亮的绿色表示复合条件变为 `true` 的第一个条形，较浅的绿色表示复合条件继续为 `true` 的后续条形。
- 虽然并不总是严格需要将单个条件分配给变量，因为它们可以直接在布尔表达式中使用，但当您将条件分配给变量名称时，它会提高代码的可读性，从而提醒您和您的读者它代表什么。在这种情况下，应始终考虑可读性，因为将条件分配给变量名对性能的影响很小或为空。

## 从内部函数调试

函数中的变量是函数的本地变量，因此不可用于从脚本的全局范围进行绘图。在此脚本中，我们编写了 `hlca()` 计算加权平均值的函数：

```

//@version=5
indicator("Debugging from inside functions", "", true)
hlca() =>
    var float avg = na
    hlca = math.avg(high, low, close, nz(avg, close))
    avg := ta.sma(hlca, 20)

h = hlca()
plot(h)

```

`hlca` 我们需要在函数计算时逐条检查函数局部范围内的值。我们无法 `hlca` 从脚本的全局范围访问函数内部使用的变量。因此，我们需要另一种机制来从函数的局部范围内提取该变量的值，同时仍然能够使用函数的结果。我们可以使用 Pine Script™ 的功能让函数返回元组来访问变量：

```

//@version=5
indicator("Debugging from inside functions", "", true)
hlca() =>
    var float avg = na
    instantVal = math.avg(high, low, close, nz(avg, close))
    avg := ta.sma(instantVal, 20)
    // Return two values instead of one.
    [avg, instantVal]

[h, instantVal] = hlca()
plot(h, "h")
plot(instantVal, "instantVal", color.black)

```

与全局范围变量相反，全局定义数组的数组元素可以在函数内修改。我们可以利用这个特性来编写一个功能等效的脚本：

```

//@version=5
indicator("Debugging from inside functions", "", true)
// Create an array containing only one float element.
instantValGlobal = array.new_float(1)
hlca() =>
    var float avg = na
    instantVal = math.avg(high, low, close, nz(avg, close))
    // Set the array's only element to the current value of `instantVal`.
    array.set(instantValGlobal, 0, instantVal)
    avg := ta.sma(instantVal, 20)

h = hlca()
plot(h, "h")
// Retrieve the value of the array's only element which was set from inside the
function.
plot(array.get(instantValGlobal, 0), "instantValGlobal", color.black)

```

## 从 [for](#) 循环内部调试

[for](#) 循环内的值无法使用循环中的 [plot\(\)](#) 调用来绘制。与在函数中一样，此类变量也是循环范围的局部变量。在这里，我们从以下代码示例开始，探索三种不同的技术来检查源自 [for](#) 循环的变量值，该代码示例计算回溯期内真实范围值高于/低于当前柱的柱的平衡：

```
//@version=5
indicator("Debugging from inside `for` loops")
lookbackInput = input.int(20, minval = 0)

float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])

hline(0)
plot(trBalance)
```

## 提取单个值

如果我们想在循环中的单个点检查变量的值，我们可以保存它并在退出循环后绘制它。在这里，我们将`tr`的值保存在`val`循环最后一次迭代的变量中：

```
//@version=5
indicator("Debugging from inside `for` loops", max_lines_count = 500, max_labels_count
= 500)
lookbackInput = input.int(20, minval = 0)

float val = na
float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])
    if i == lookbackInput
        val := ta.tr[i]
hline(0)
plot(trBalance)
plot(val, "val", color.black)
```

## 使用线条和标签

当我们想要从多个循环迭代中提取值时，我们可以使用线条和标签。这里我们画一条线，对应于每次循环迭代中使用的`ta.tr`的值。我们还使用标签来显示每行的循环索引和行的值。这让我们对每个循环迭代中使用的值有一个大致的了解：

```

//@version=5
indicator("Debugging from inside `for` loops", max_lines_count = 500, max_labels_count
= 500)
lookbackInput = input.int(20, minval = 0)

float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])
    line.new(bar_index[1], ta.tr[i], bar_index, ta.tr[i], color = color.black)
    label.new(bar_index, ta.tr[i], str.tostring(i) + "•" + str.tostring(ta.tr[i]),
style = label.style_none, size = size.small)

hline(0)
plot(trBalance)

```

注意：

- 为了显示更多细节，前面的屏幕截图中的比例已通过单击并拖动比例区域来手动扩展。
- 我们在[indicator\(\)](#)声明语句中使用来显示最大行数和标签数。`max_lines_count = 500,`  
`max_labels_count = 500`
- 每次循环迭代不一定会产生不同的[ta.tr](#)值，这就是为什么我们可能看不到每个柱有 20 条不同的线。
- 如果我们只想显示一个级别，我们可以使用相同的技术，同时隔离特定的循环迭代，就像我们在前面的示例中所做的那样。

## 提取多个值

我们还可以通过构建单个字符串从循环迭代中提取多个值，我们将在循环执行后使用标签显示该字符串：

```

//@version=5
indicator("Debugging from inside `for` loops", max_lines_count = 500, max_labels_count
= 500)
lookbackInput = input.int(20, minval = 0)

string = ""
float trBalance = 0
for i = 1 to lookbackInput
    trBalance := trBalance + math.sign(ta.tr - ta.tr[i])
    string := string + str.tostring(i, "00") + "•" + str.tostring(ta.tr[i]) + "\n"

label.new(bar_index, 0, string, style = label.style_none, size = size.small, textalign
= text.align_left)
hline(0)
plot(trBalance)

```

注意：

- 前面的屏幕截图中的刻度已通过单击并拖动刻度区域来手动扩展，因此指标显示区域的内容可以垂直移动以仅显示其相关部分。
- 我们用来强制循环索引的显示以零填充两位数，以便它们整齐地对齐。`str.tostring(i, "00")`

当具有多次迭代的循环导致显示其所有值不切实际时，您可以对迭代的子集进行采样。此代码使用`%`（模）运算符来包含来自每个第二个循环迭代的值：

```
for i = 1 to i_lookBack
    lowerRangeBalance := lowerRangeBalance + math.sign(ta.tr - ta.tr[i])
    if i % 2 == 0
        string := string + str.tostring(i, "00") + "•" + str.tostring(ta.tr[i]) + "\n"
```

## 提示

我们最常用于调试 Pine Script™ 代码的两种技术是：

```
plotchar(v, "v", "", location.top, size = size.tiny)
```

在指标值和数据窗口中绘制`float`、`int`或`bool` `print()`类型的变量，以及用于调试字符串的函数的单行版本：

```
print(txt) => var _label = label.new(bar_index, na, txt, xloc.bar_index, yloc.price,
color(na), label.style_none, color.gray, size.large, text.align_left),
label.set_xy(_label, bar_index, ta.highest(10)[1]), label.set_text(_label, txt)
print(stringName)
```

当我们使用 Windows 版 AutoHotkey 来加速重复性任务时，我们将这些行包含在 AutoHotkey 脚本中（这不是 Pine Script™ 代码）：

```
; ----- This is AHK code, not Pine Script™. -----
^+f:: SendInput plotchar(^v, "v", "", location.top, size = size.tiny){Return}
^+p:: SendInput print(txt) => var lbl = label.new(bar_index, na, txt, xloc.bar_index,
yloc.price, color(na), label.style_none, color.gray, size.large, text.align_left),
label.set_xy(lbl, bar_index, highest(10)[1]), label.set_text(lbl, txt)`nprint(){Left}
```

第二行将键入调试`plotchar()`调用，其中包括先前在使用++时复制到剪贴板的表达式或变量名称f。将`variableName` 变量名称或条件表达式复制到剪贴板并点击++将分别产生：`close > open` `ctrlshiftf`

```
plotchar(variableName, "variableName", "", location.top, size = size.tiny)
plotchar(close > open, "close > open", "", location.top, size = size.tiny)
```

第三行在`ctrl+ shift+p`上触发p。它在脚本中键入我们的一行`print()`函数，然后在第二行上对函数进行空调用，并将光标放置在其中，因此剩下要做的就是键入我们要显示的字符串：

```
print(txt) => var lbl = label.new(bar_index, na, txt, xloc.bar_index, yloc.price,
color(na), label.style_none, color.gray, size.large, text.align_left),
label.set_xy(lbl, bar_index, ta.highest(10)[1]), label.set_text(lbl, txt)
print()
```

注意：AutoHotkey 仅适用于 Windows 系统。Apple 系统上可以用 Keyboard Maestro 或其他键盘替代。

## 发布脚本

希望与其他交易者分享 Pine 脚本的程序员可以发布它们。

提示：

如果您编写的脚本供个人使用，则无需发布；您可以将它们保存在 Pine 编辑器中，然后使用“添加到图表”按钮将脚本添加到图表中。

## 脚本可见性和访问

当您发布脚本时，您可以控制其可见性和访问权限：

- 通过选择公开或私下发布来控制可见性。请参阅[私人想法和脚本与公共想法和脚本有何不同？](#)请在帮助中心了解更多详细信息。当您编写了您认为对 TradingViewers 有用的脚本时，请公开展示。公共脚本需要经过审核。为了避免审核，请确保您的出版物符合我们的[内部规则](#)和[脚本出版规则](#)。当您不希望所有其他用户看到您的脚本，但希望与一些朋友共享时，可以私下发布。
- 访问权限决定用户是否会看到您的源代码，以及他们如何使用您的脚本。共有三种访问类型：开放、受保护（保留在付费帐户）或仅限邀请（保留在高级帐户）。请参阅[已发布的脚本有哪些不同类型？](#)请在帮助中心了解更多详细信息。

## 当您发布脚本时

- `title` 出版物的标题由脚本的 [indicator\(\)](#) 或 [strategy\(\)](#) 声明语句中的参数所使用的参数确定。当 TradingViewers 搜索脚本名称时也会使用该标题。
- 图表上脚本的名称将是用于 `shorttitle` 脚本的 [dicator\(\)](#) 或 [strategy\(\)](#) 声明语句中的参数的参数，或者是 [library\(\)](#) 中的 `title` 参数。
- 您的脚本必须有一个说明，解释您的脚本的用途以及如何使用它。
- 您发布时使用的图表将在您的出版物中可见，包括其中的任何其他脚本或绘图。在发布脚本之前，从图表中删除不相关的脚本或绘图。
- 您的脚本代码稍后可以更新。每个更新都可以包含发行说明，这些发行说明将显示在您的原始描述下方并注明日期。
- 脚本可以被其他用户点赞、分享、评论或举报。
- 您发布的脚本将显示在您的用户个人资料的“脚本”选项卡下。
- 为您的脚本创建脚本小部件和脚本页面。脚本小部件是脚本的占位符，显示在平台上的脚本源中。它包含脚本的标题、图表和描述的前几行。当用户单击您的脚本小部件时，脚本的页面将打开。它包含与您的脚本相关的所有信息。

## 可见性

### 民众

当您发布公共脚本时：

- 您的脚本将包含在我们的[社区脚本](#)中，该脚本对网站所有国际化版本上的数百万 TradingViewers 可见。
- 您的出版物必须遵守[《内部规则》](#)和[《剧本出版规则》](#)。
- 如果您的脚本是仅限邀请的脚本，您必须遵守我们的[供应商要求](#)。
- 它可以通过脚本的搜索功能进行访问。
- 您将无法编辑原始描述或其标题，也无法更改其公共/私人可见性及其访问类型（开源、受保护、仅限邀请）。
- 您将无法删除您的出版物。

### 私人的

当您发布私有脚本时：

- 除非您与其他用户共享其 URL，否则其他用户将看不到它。
- 您可以从用户个人资料的“脚本”选项卡中看到它。
- 私有脚本可以通过其小部件右上角的“X”和“锁定”图标来识别。“X”用于删除它。
- 它不会受到监管，除非您出售对它的访问权或将其公开提供，因为这样它就不再是“私有的”。
- 您可以更新其原始描述和标题。
- 您不能在任何公共 TradingView 内容（想法、脚本描述、评论、聊天等）中链接或提及它。
- 无法通过脚本搜索功能访问它。

## 使用权

可以使用三种访问类型之一发布公共或私人脚本：开放、受保护或仅限邀请。您可以选择的访问类型将根据您持有的帐户类型而有所不同。

### 打开

公开发布的脚本的 Pine Script™ 代码对所有用户都可见。TradingView 上的开源脚本默认使用 Mozilla 许可证，但您可以选择任何您想要的许可证。您可以在[GitHub](#)上找到有关许可的信息。

### 受保护

受保护脚本的代码是隐藏的，除了其作者之外，没有人可以访问它。虽然脚本的代码不可访问，但任何用户都可以自由使用受保护的脚本。只有 Pro、Pro+ 或 Premium 帐户可以发布公共受保护的脚本。

### 只有邀请

仅限邀请的访问类型可以保护脚本的代码及其使用。仅限邀请的脚本的发布者必须明确向各个用户授予访问权限。仅限邀请的脚本主要由脚本供应商使用，提供对其脚本的付费访问。只有高级帐户才能发布仅限邀请的脚本，并且它们必须符合我们的[供应商要求](#)。

TradingView 不会从脚本销售中受益。仅限邀请的脚本的交易严格在用户和供应商之间进行；他们不涉及 TradingView。

仅限公开邀请的脚本是唯一允许供应商在 TradingView 上要求付款的脚本。

在仅限邀请的脚本页面上，作者将看到“管理访问”按钮。“管理访问”窗口允许作者控制谁有权访问其脚本。

The screenshot shows a TradingView script page for the "Aroon with dynamic colors" study. At the top, there's a chart showing price action over time. Below the chart, there's a "Trend Analysis" section. The main content area contains sections for "Invite-only script" (with a note about protected code), "Author's instructions" (requesting a private message), and "Want to use this script on a chart?" (with a warning to read the script before requesting access). Two buttons are present: "★ Add to favorite indicators" and "Manage Access". An orange arrow points from the "Manage Access" button to the "Manage Access" link in the "Author's instructions" section. Below these buttons is the Pine script code:

```
1 //version=4
2 study("Aroon with dynamic colors", "Aroon", false, format.percent, 2)
3 i_length = input(25, minval = 1)
4 float upper = 100 * (highestbars(high, i_length + 1) + i_length) / i_length
5 float lower = 100 * (lowestbars(low, i_length + 1) + i_length) / i_length
6 plot(upper, "Upper", upper > lower ? #00FF00FF : #00FF0060)
7 plot(lower, "Lower", lower > upper ? #FF0080FF : #FF008060)
8
```

At the bottom, there are links for tools and ideas, a comments section, and a "Comments" section.

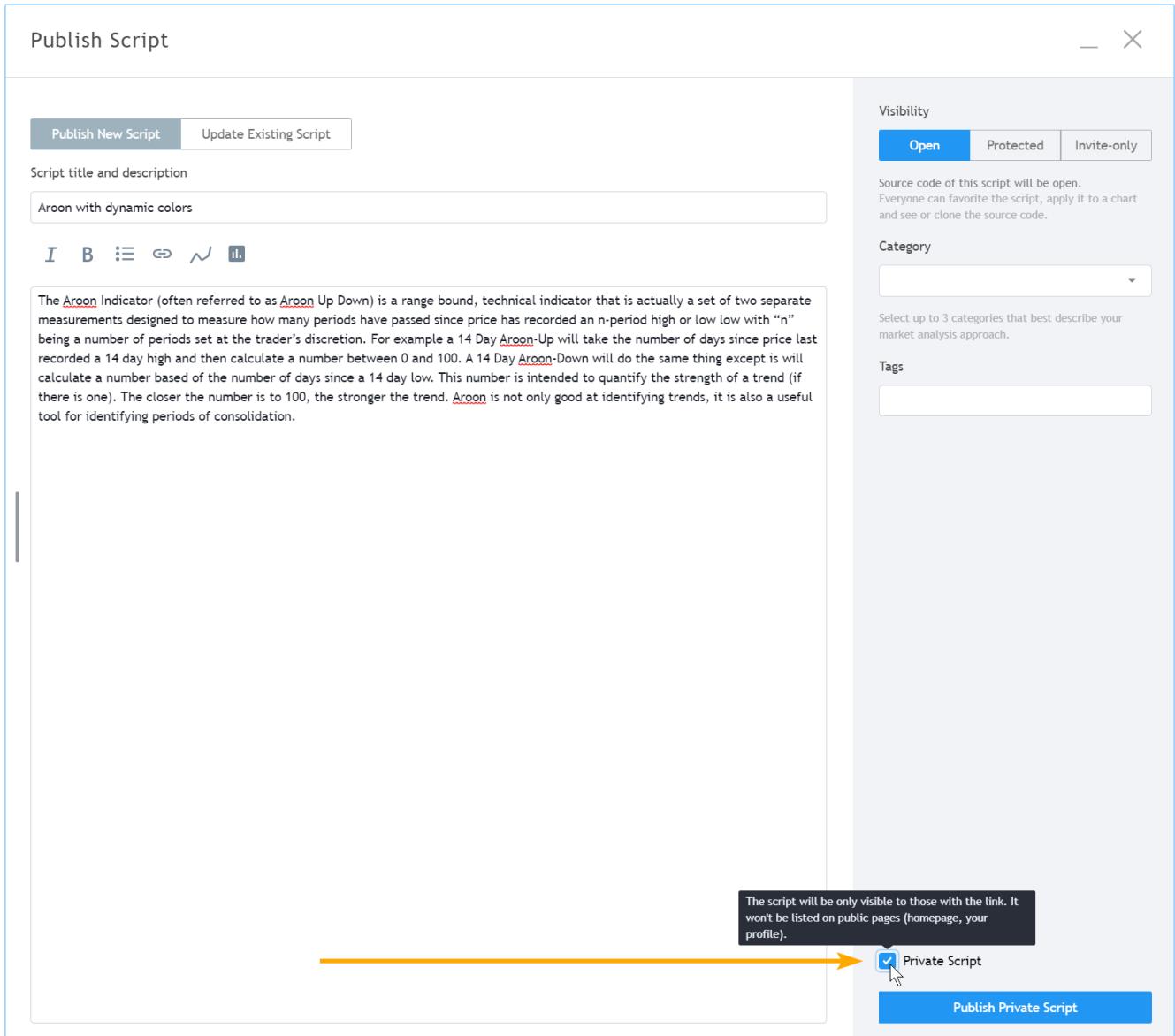
## 准备出版物

- 即使您打算公开发布，最好从私人出版物开始，因为您可以使用它来验证最终出版物的外观。您可以编辑私人出版物的标题、描述、代码或图表，与公共脚本相反，您可以在不再需要私人脚本时将其删除，因此它们是公开共享脚本之前练习的完美方式。您可以在[《我们如何编写和格式化脚本描述》](#)出版物中阅读有关准备脚本描述的更多信息。
- 准备你的图表。将您的脚本加载到图表上，并删除其他无法帮助用户理解您的脚本的脚本或绘图。您的脚本的绘图应该很容易在将随其发布的图表上识别。
- 如果尚未在 Pine 编辑器中加载您的代码。在编辑器中，单击“发布脚本”按钮：

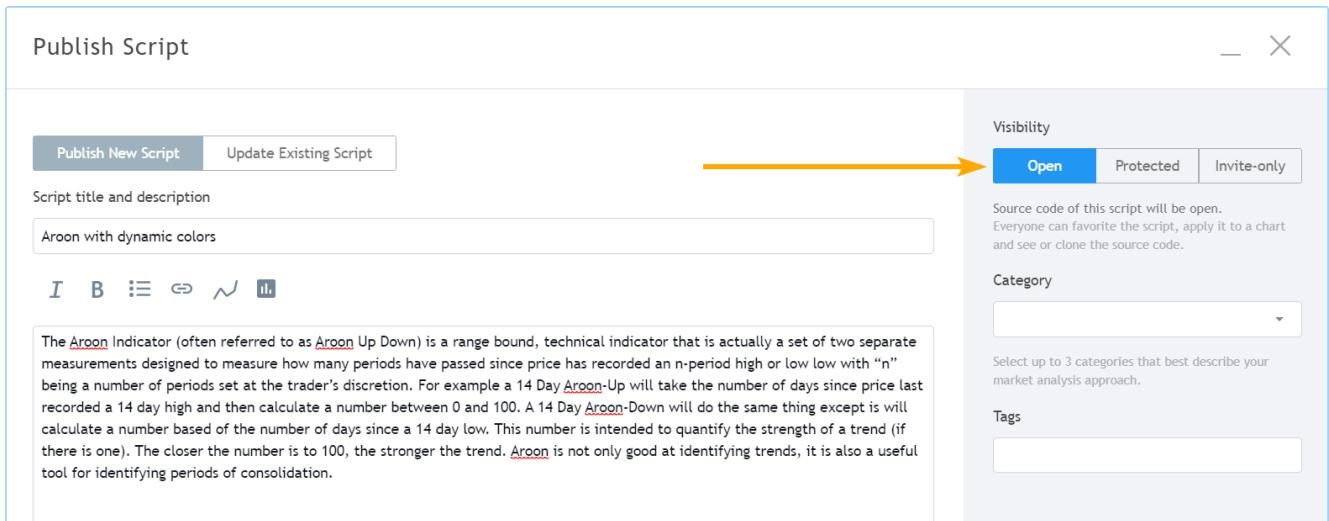
The screenshot shows the Pine Editor interface. The tabs at the top are Stock Screener, Text Notes, Pine Editor (which is active), Strategy Tester, and Trading Panel. Below the tabs, there's a search bar with the text "Aroon" and a dropdown showing "13.0 ~". On the right side of the editor, there are buttons for Open, Save, Add to Chart, and Publish Script. An orange arrow points from the "Publish Script" button to the "Publish Script" link in the "Comments" section of the previous screenshot. The Pine script code is visible in the editor:

```
1 //version=4
2 study("Aroon with dynamic colors", "Aroon", false, format.percent, 2)
3 i_length = input(25, minval = 1)
4 float upper = 100 * (highestbars(high, i_length + 1) + i_length) / i_length
5 float lower = 100 * (lowestbars(low, i_length + 1) + i_length) / i_length
6 plot(upper, "Upper", upper > lower ? #00FF00FF : #00FF0060)
7 plot(lower, "Lower", lower > upper ? #FF0080FF : #FF008060)
8
```

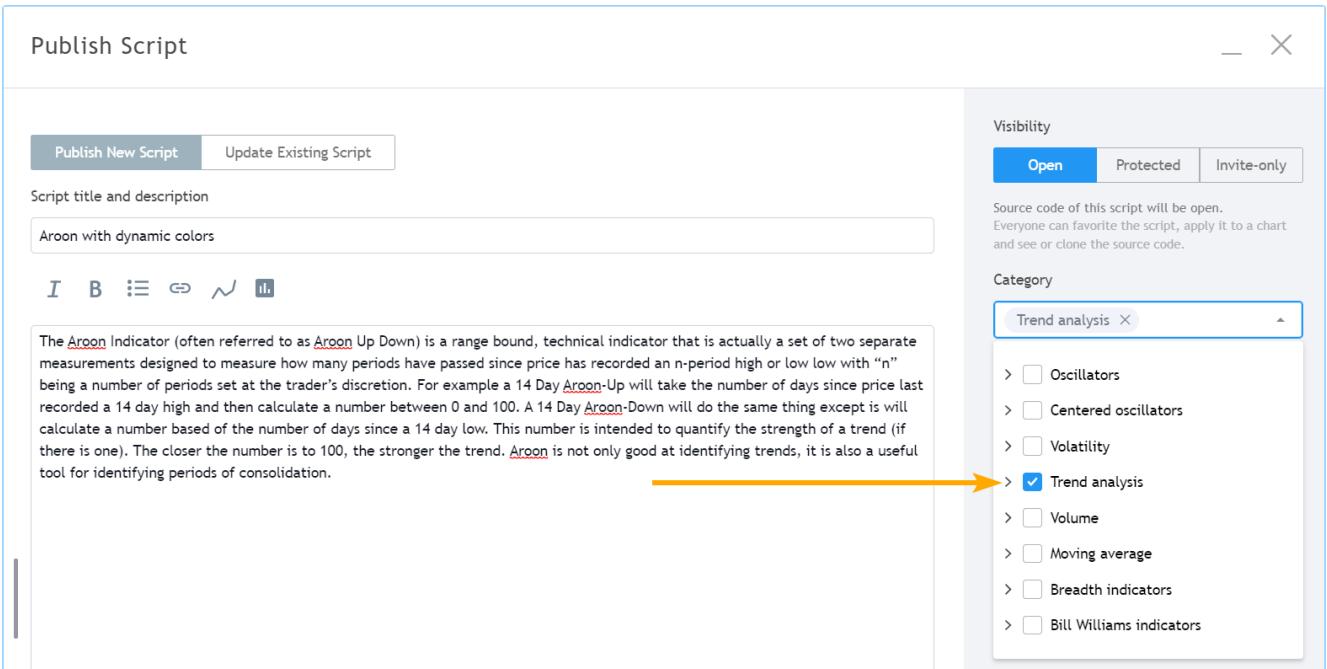
4. 弹出窗口会提醒您，如果您公开发表文章，那么您的出版物必须遵守众议院规则，这一点很重要。完成弹出窗口后，将您的描述放入脚本标题下方的字段中。为您的出版物建议的默认标题是 title 脚本代码中的字段。最好使用该标题；如果您的脚本是公开的，它可以让用户更轻松地搜索您的脚本。选择您的出版物的可见性。我们想要发布私人出版物，因此我们选中“发布脚本”窗口右下角的“私人脚本”复选框：



5. 选择您想要的脚本访问类型：开放、受保护或仅限邀请。我们为开源选择了“开放”。



6. 为您的脚本选择适当的类别（至少有一个是强制性的）并输入可选的自定义标签。



7. 单击窗口右下角的“发布私有脚本”按钮。发布完成后，将出现您发布的脚本页面。你完成了！你可以通过转到您的用户个人资料并查看“脚本”选项卡来确认发布。从那里，您将能够打开脚本页面并使用脚本页面右上角的“编辑”按钮编辑您的私人出版物。请注意，您也可以更新私人出版物，就像更新公共出版物一样。如果您想与朋友分享您的私人出版物，请私下向她发送脚本页面的网址。请记住，您不得在公共 TradingView 内容中共享私人出版物的链接。

## 发布脚本

无论您打算私下发布还是公开发布，请首先按照上一节中的步骤进行操作。如果您打算私下发布，就可以了。如果您打算公开发布并且对验证私人出版物的准备过程感到满意，请按照与上述相同的步骤操作，但不要选中“私人脚本”复选框，然后单击页面右下角的“发布公共脚本”按钮“发布脚本”页面。

当您发布新的公共脚本时，您有 15 分钟的时间来更改描述或删除出版物。此后，您将无法再更改出版物的标题、描述、可见性或访问类型。如果您犯了错误，请向[PineCoders](#)版主帐户发送消息；他们负责审核剧本出版物并提供帮助。

## 更新出版物

您可以更新公共或私人脚本出版物。更新脚本时，其代码必须与之前发布版本的代码不同。您可以在更新中添加发行说明。它们将出现在脚本页面中脚本的原始描述之后。

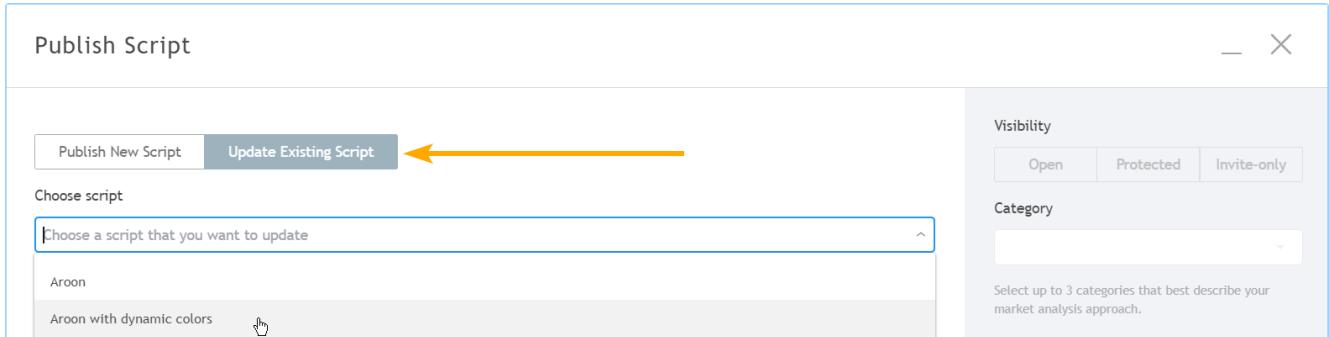
默认情况下，更新时使用的图表将替换脚本页面中以前的图表。但是，您可以选择不更新脚本页面的图表。请注意，虽然您可以更新脚本页面中显示的图表，但脚本小部件中的图表不会更新。

同样，您可以通过首先发布私有脚本来验证公共发布，您也可以先验证私有发布的更新，然后再在公共发布上进行更新。对于公共脚本和私有脚本，更新已发布脚本的过程是相同的。

如果您打算更新已发布脚本的代码和图表，请按照与新出版物相同的方式准备图表。在以下示例中，我们将**不会**更新出版物的图表：

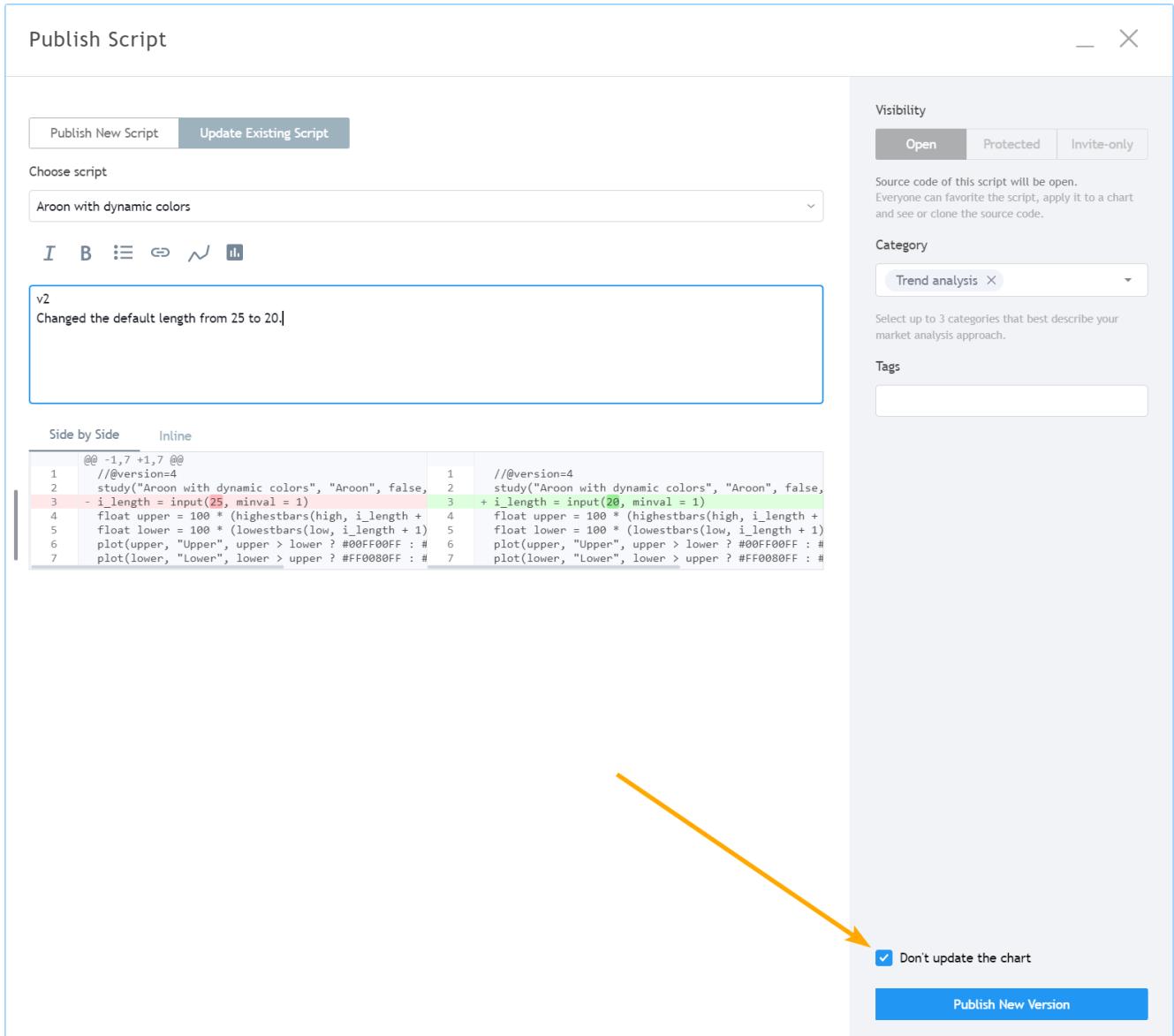
1. 就像发布新出版物一样，在编辑器中加载脚本并单击“发布脚本”按钮。

2. 进入“发布脚本”窗口后，选择“更新现有脚本”按钮。然后从“选择脚本”下拉菜单中选择要更新的脚本：



3. 在文本字段中输入您的发行说明。代码中的差异在发行说明下方突出显示。

4. 我们不想更新出版物的图表，因此我们选中“不更新图表”复选框：



5. 单击“发布新版本”按钮。你完成了。

## 限制

## 介绍

正如我们的[欢迎](#)页面中提到的：

由于每个脚本都使用云中的计算资源，因此我们必须施加限制，以便在用户之间公平地共享这些资源。我们努力设置尽可能少的限制，但当然必须实施尽可能多的限制，以使平台顺利运行。限制适用于从附加符号请求的数据量、执行时间、内存使用和脚本大小。

如果您使用 Pine Script™ 开发复杂的脚本，迟早您会遇到我们施加的一些限制。本节概述了您可能遇到的限制。目前，Pine Script™ 程序员无法获取有关其脚本消耗的资源的数据。我们希望这种情况将来会改变。

同时，当您考虑大型项目时，最安全的做法是进行概念验证，以评估脚本在项目后期遇到限制的可能性。

下面，我们描述了 Pine Script™ 环境中施加的限制。

## 时间

---

### 脚本编译

脚本必须先编译，然后才能在图表上执行。当您从 Pine 编辑器保存脚本或将脚本添加到图表时，会发生编译。编译时间有两分钟的限制，这取决于脚本的大小和复杂性，以及先前编译的缓存版本是否可用。当编译超过两分钟限制时，会发出警告。通过缩短脚本来注意该警告，因为连续三次警告后，将强制执行一小时禁止编译尝试。优化代码首先要考虑的是避免重复，用函数封装常用的段，调用函数而不是重复代码。

### 脚本执行

脚本编译完成后就可以执行。有关[触发脚本执行的事件](#)列表，请参阅触发脚本执行的事件。分配给脚本在数据集的所有柱上执行的时间因账户类型而异。基本帐户的限制为 20 秒，其他帐户为 40 秒。

### 循环执行

任何单个柱上的任何循环的执行时间限制为 500 毫秒。嵌入循环的外层循环算作一个循环，因此会先超时。请记住，即使循环可能在给定柱上的 500 毫秒时间限制内执行，但在所有数据集的柱上执行所需的时间可能会导致您的脚本超出总执行时间限制。例如，总执行时间的限制将使您的脚本无法在 20,000 个柱数据集的每个柱上执行 400 毫秒的循环，因为您的脚本将需要 8000 秒来执行。

### 图表视觉效果

---

## 绘图限制

每个脚本最多允许 64 个绘图计数。生成绘图计数的函数是：

- [plot\(\)](#)
- [plotarrow\(\)](#)
- [plotbar\(\)](#)
- [plotcandle\(\)](#)
- [plotchar\(\)](#)
- [plotshape\(\)](#)
- [alertcondition\(\)](#)
- [bgcolor\(\)](#)
- [fill\(\)](#), 但前提是它 `color` 是 [series](#) 形式。

以下函数不会生成绘图计数：

- [hline\(\)](#)
- [line.new\(\)](#)
- [label.new\(\)](#)
- [table.new\(\)](#)
- [box.new\(\)](#)

一次函数调用最多可以生成七个绘图计数，具体取决于函数及其调用方式。当您的脚本超过 64 个绘图计数的最大值时，运行时错误消息将显示脚本生成的绘图计数。到达该点后，您可以通过在脚本中注释掉函数调用来确定函数调用生成的绘图计数。只要您的脚本仍然抛出错误，您就可以看到注释掉一行后实际绘图计数如何减少。

以下示例显示了不同的函数调用以及每个函数调用将生成的绘图计数数：

```
//@version=5
indicator("Plot count example")

bool isUp = close > open
color isUpColor = isUp ? color.green : color.red
bool isDn = not isUp
color isDnColor = isDn ? color.red : color.green

// Uses one plot count each.
p1 = plot(close, color = color.white)
p2 = plot(open, color = na)

// Uses two plot counts for the `close` and `color` series.
plot(close, color = isUpColor)

// Uses one plot count for the `close` series.
plotarrow(close, colorup = color.green, colordown = color.red)
```

```
// Uses two plot counts for the `close` and `colorup` series.  
plotarrow(close, colorup = isUpColor)  
  
// Uses three plot counts for the `close`, `colorup`, and the `colordown` series.  
plotarrow(close - open, colorup = isUpColor, colordown = isDnColor)  
  
// Uses four plot counts for the `open`, `high`, `low`, and `close` series.  
plotbar(open, high, low, close, color = color.white)  
  
// Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series.  
plotbar(open, high, low, close, color = isUpColor)  
  
// Uses four plot counts for the `open`, `high`, `low`, and `close` series.  
plotcandle(open, high, low, close, color = color.white, wickcolor = color.white,  
bordercolor = color.purple)  
  
// Uses five plot counts for the `open`, `high`, `low`, `close`, and `color` series.  
plotcandle(open, high, low, close, color = isUpColor, wickcolor = color.white,  
bordercolor = color.purple)  
  
// Uses six plot counts for the `open`, `high`, `low`, `close`, `color`, and  
`wickcolor` series.  
plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor ,  
bordercolor = color.purple)  
  
// Uses seven plot counts for the `open`, `high`, `low`, `close`, `color`, `wickcolor`,  
and `bordercolor` series.  
plotcandle(open, high, low, close, color = isUpColor, wickcolor = isUpColor ,  
bordercolor = isUp ? color.lime : color.maroon)  
  
// Uses one plot count for the `close` series.  
plotchar(close, color = color.white, text = "|", textcolor = color.white)  
  
// Uses two plot counts for the `close`` and `color` series.  
plotchar(close, color = isUpColor, text = "-", textcolor = color.white)  
  
// Uses three plot counts for the `close`, `color`, and `textcolor` series.  
plotchar(close, color = isUpColor, text = "O", textcolor = isUp ? color.yellow :  
color.white)  
  
// Uses one plot count for the `close` series.  
plotshape(close, color = color.white, textcolor = color.white)  
  
// Uses two plot counts for the `close` and `color` series.  
plotshape(close, color = isUpColor, textcolor = color.white)  
  
// Uses three plot counts for the `close`, `color`, and `textcolor` series.  
plotshape(close, color = isUpColor, textcolor = isUp ? color.yellow : color.white)
```

```
// Uses one plot count.
alertcondition(close > open, "close > open", "Up bar alert")

// Uses one plot count.
bgcolor(isUp ? color.yellow : color.white)

// Uses one plot count for the `color` series.
fill(p1, p2, color = isUpColor)
```

此示例生成的绘图计数为 56。如果我们要添加最后一次调用 [plotcandle\(\)](#) 的另外两个实例，脚本将抛出一个错误，指出脚本现在使用 70 个绘图计数，因为每次额外调用 [plotcandle\(\)](#) 生成 7 个绘图计数， $56 + (7 * 2)$  为 70。

## 线、框、折线和标签限制

与可覆盖图表的整个数据集的[绘图](#)相反，脚本默认仅显示图表上的最后 50 条线、框、折线和标签。可以通过脚本的 [Indicator\(\)](#) 或 [Strategy\(\)](#) 声明语句的、和参数来增加每种[绘图类型的最大数量](#)。[line](#)、[box](#) 和 [label](#) ID 的最大数量为 500，[折线ID](#) 的最大数量为

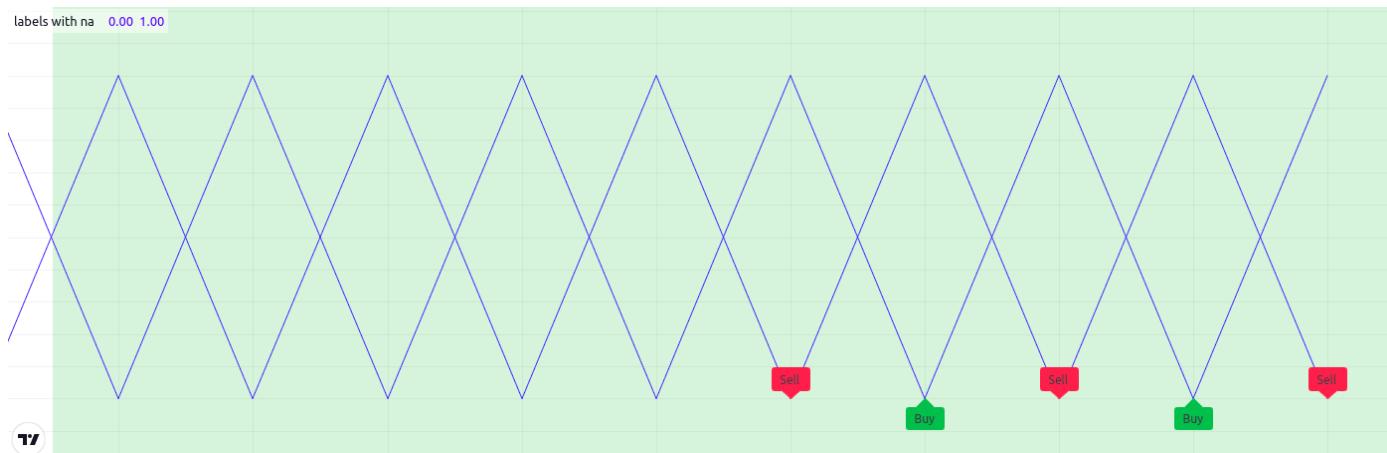
100。`max_lines_count``max_boxes_count``max_polylines_count``max_labels_count`

在此示例中，我们将图表上显示的最近标签的最大数量设置为 100：

```
//@version=5
indicator("Label limits example", max_labels_count = 100, overlay = true)
label.new(bar_index, high, str.tostring(high, format.mintick))
```

需要注意的是，将绘图对象的任何属性设置为 [na](#) 时，其 ID 仍然存在，从而影响脚本的绘图总数。为了演示此行为，以下脚本在每个柱上绘制“买入”和“卖出”[标签](#)，其值由和变量 [x](#) 确定。`longCondition``shortCondition`

当柱索引为偶数时，“买入”标签的 [x](#) 值为 [na](#)；当柱索引为奇数时，“卖出”标签的 [x](#) 值为 [na](#)。尽管本示例中的 [x](#) 为 10，但我们可以看到脚本在图表上显示的[标签](#) `max_labels_count` 少于 10 个，因为具有 [na](#) 值的标签也计入总数：



```
//@version=5
// Approximate maximum number of label drawings
```

```

MAX_LABELS = 10

indicator("labels with na", overlay = false, max_labels_count = MAX_LABELS)

// Add background color for the last MAX_LABELS bars.
bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na)

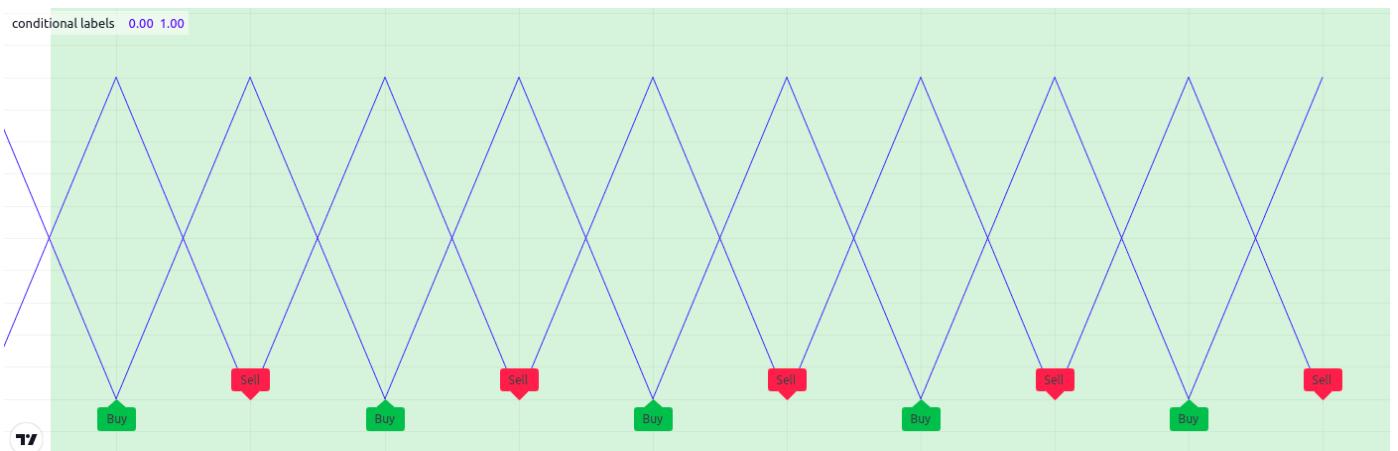
longCondition = bar_index % 2 != 0
shortCondition = bar_index % 2 == 0

// Add "Buy" and "Sell" labels on each new bar.
label.new(longCondition ? bar_index : na, 0, text = "Buy", color =
color.new(color.green, 0), style = label.style_label_up)
label.new(shortCondition ? bar_index : na, 0, text = "Sell", color =
color.new(color.red, 0), style = label.style_label_down)

plot(longCondition ? 1 : 0)
plot(shortCondition ? 1 : 0)

```

要显示所需数量的标签，我们必须消除不想显示的标签绘图，而不是将其属性设置为 `na`。下面的示例使用 `if` 结构有条件地绘制“买入”和“卖出”标签，防止脚本在不必要时创建新的标签 ID：



```

//@version=5

// Approximate maximum number of label drawings
MAX_LABELS = 10

indicator("conditional labels", overlay = false, max_labels_count = MAX_LABELS)

// Add background color for the last MAX_LABELS bars.
bgcolor(bar_index > last_bar_index - MAX_LABELS ? color.new(color.green, 80) : na)

longCondition = bar_index % 2 != 0
shortCondition = bar_index % 2 == 0

// Add a "Buy" label when `longCondition` is true.
if longCondition

```

```

label.new(bar_index, 0, text = "Buy", color = color.new(color.green, 0), style =
label.style_label_up)
// Add a "Sell" label when `shortCondition` is true.
if shortCondition
    label.new(bar_index, 0, text = "Sell", color = color.new(color.red, 0), style =
label.style_label_down)

plot(longCondition ? 1 : 0)
plot(shortCondition ? 1 : 0)

```

## 表限制

脚本最多可以在图表上 显示九个表格，每个表格对应一个可能的位置：[position.bottom\\_center](#)、[position.bottom\\_left](#)、[position.bottom\\_right](#)、[position.middle\\_center](#)、[position.middle\\_left](#)、[position.middle\\_right](#)、[position.top\\_center](#)、[position.top\\_left](#)和[position.top\\_right](#)。当尝试将两个表放置在同一位置时，图表上只会显示最新的实例。

## [request.\\*\(\)](#) 调用

### 调用次数

一个脚本不能包含超过 40 个对 `request.()` 命名空间中函数的调用。这些函数的所有实例都计入此限制，即使包含在脚本主逻辑未使用的[用户定义函数](#)的本地块中也是如此。此限制适用于[其他时间范围和数据](#)页面中讨论的所有功能，包括：

- [request.security\(\)](#)
- [request.security\\_lower\\_tf\(\)](#)
- [request.currency\\_rate\(\)](#)
- [request.dividends\(\)](#)
- [request.splits\(\)](#)
- [request.earnings\(\)](#)
- [request.quandl\(\)](#)
- [request.financial\(\)](#)
- [request.economic\(\)](#)
- [request.seed\(\)](#)

## 内栏

脚本可以通过[request.security\(\)](#)或[request.security\\_lower\\_tf\(\)](#)函数检索最多 100,000 个内部柱（较低时间范围柱）。

图表时间范围内 100,000 个内部柱所覆盖的柱数随每个图表柱所包含的内部柱的数量而变化。例如，在 60 分钟图表上运行脚本时请求 1 分钟时间范围内的数据意味着每个图表条形最多可以包含 60 个内部条形图。在这种情况下，柱内请求覆盖的最小图表柱数为 1,666，即  $100,000 / 60 = 1,666.67$ 。但值得注意的是，提供商可能不会在一小时内每分钟报告数据。因此，此类请求可能会覆盖更多图表条形，具体取决于可用数据。

## 元组元素限制

脚本中的所有 `request.*()` 函数调用返回的元组元素总数不能超过 127 个。当所有 `request.*()` 调用的组合元组大小超过 127 个元素时，可以使用[用户定义类型\(UDT\)](#)来请求更多数量的值。

下面的示例概述了此限制以及解决该限制的方法。第一个 [request.security\(\)](#) 调用表示使用包含 128 个元素的元组作为 `expression` 参数。由于元素数量大于 127，因此会导致错误。

为了避免错误，我们可以使用这些相同的值作为 UDT 对象中的 [字段](#)，并将其 ID 传递给：`expression`

```
//@version=5
indicator("Tuple element limit")

s1 = close
s2 = close * 2
...
s128 = close * 128

// Causes an error.
[v1, v2, v3, ..., v128] = request.security(syminfo.tickerid, "1D", [s1, s2, s3, ..., s128])

// Works fine:
type myType
    float v1
    float v2
    float v3
    ...
    float v128

myObj = request.security(syminfo.tickerid, "1D", myType.new(s1, s2, s3, ..., s128))
```

- 注意：

此示例概述了脚本尝试在单个[request.security\(\)](#)调用中评估 128 个元组元素的场景。如果我们要将元组请求拆分为多个调用，则同样的限制也适用。例如，两次[request.security\(\)](#)调用（每次调用都检索具有 64 个元素的元组）也会导致错误。

# 脚本大小和内存

## 编译的令牌

在执行脚本之前，编译器会将其翻译为标记化的中间语言(IL)。使用 IL 允许 Pine Script™ 通过应用各种内存和性能优化来容纳更大的脚本。编译器根据IL 形式中的标记数量来确定脚本的大小，而不是Pine 编辑器中可查看的代码中的字符数或行数。

每个指标、策略和库脚本的编译形式仅限于 68,000 个代币。当脚本导入库时，所有导入库的令牌总数不能超过 100 万个。无法检查脚本的编译形式及其 IL 标记计数。因此，只有当编译器达到脚本大小限制时，您才会知道脚本超出了大小限制。

在大多数情况下，脚本的编译大小可能不会达到限制。但是，如果编译的脚本确实达到了令牌限制，那么减少编译令牌的最有效方法是减少重复代码、将冗余调用封装在函数中并尽可能使用[库](#)。

需要注意的是，编译过程会从最终的 IL 形式中省略任何未使用的变量、函数、类型等，其中“未使用”指的\*\*是不影响脚本输出的任何内容。此优化可防止代码中多余的元素影响脚本的 IL 令牌计数。

例如，下面的脚本声明一个[用户定义类型](#)和一个[用户定义方法](#)，并定义使用它们的调用序列：

```
//@version=5
indicator("My Script")
plot(close)

type myType
    float field = 10.0

method m(array<myType> a, myType v) =>
    a.push(v)

var arr = array.new<myType>()
arr.push(myType.new(25))
arr.m(myType.new())
```

尽管脚本中包含了[array.new\(\)](#)、[myType.new\(\)](#)、和 [arr.m\(\)](#) 调用，但脚本实际输出 [plot\(close\)](#) 的唯一内容是。其余代码不影响输出。因此，该脚本的编译形式将具有与以下相同数量的标记：

```
//@version=5
indicator("My Script")
plot(close)
```

## 每个作用域的变量

脚本的每个作用域中最多可以包含 1,000 个变量。 Pine 脚本始终包含一个全局作用域，由非缩进代码表示，并且它们可能包含零个或多个局部作用域。局部作用域是缩进代码的一部分，表示在[函数](#)和[方法](#)以及[if](#)、[switch](#)、[for](#)、[for...in](#)和[while](#)结构中执行的过程，这些结构允许一个或多个局部块。每个本地块都算作一个本地范围。

使用 [?:](#)三元运算符的条件表达式的分支不算作本地块。

## 范围计数

脚本中的作用域总数，包括其全局作用域以及来自[用户定义的函数](#)、[方法](#)、[条件结构](#)或它使用的[循环](#)的每个局部作用域，不能超过 500。

值得注意的是，[request.security\(\)](#)、[request.security\\_lower\\_tf\(\)](#)和[request.seed\(\)](#)函数 重复了在另一个上下文中评估其 `expression` 参数值所需的范围。每次调用这些 `request.*()` 函数所产生的作用域也会计入脚本的作用域限制。

例如，假设我们创建了一个带有全局变量的脚本，该变量取决于 250 个 [if](#) 结构的局部范围。该脚本的总作用域计数为 251 (1 个全局作用域 + 250 个局部作用域)：

```
//@version=5
indicator("Scopes demo")

var x = 0

if close > 0
    x += 0
if close > 1
    x += 1
// ... Repeat this `if close > n` pattern until `n = 249`.
if close > 249
    x += 249

plot(x)
```

由于作用域总数在限制范围内，因此可以编译成功。现在，假设我们调用[request.security\(\)](#) 来评估另一个上下文中的值并[绘制](#)它的值。在这种情况下，它实际上会使脚本的作用域计数加倍，因为 的值 `x` 取决于所有脚本的作用域：

```
//@version=5
indicator("Scopes demo")

var x = 0

if close > 0
    x += 0
if close > 1
```

```

x += 1
// ... Repeat this `if close > n` pattern until `n = 249`.
if close > 249
    x += 249

plot(x)
plot(request.security(syminfo.tickerid, "1D", x) // Causes compilation error since the
scope count is now 502.

```

我们可以通过将`if`块封装在[用户定义的函数](#)中来解决此问题，因为函数的作用域算作一个嵌入作用域：

```

//@version=5
indicator("Scopes demo")

f() =>
    var x = 0

    if close > 0
        x += 0
    if close > 1
        x += 1
    // ... Repeat this `if close > n` pattern until `n = 249`.
    if close > 249
        x += 249

plot(f())
plot(request.security(syminfo.tickerid, "1D", f()) // No compilation error.

```

## 收藏

Pine Script™ 集合（[数组](#)、[矩阵](#)和[映射](#)）最多可以有 100,000 个元素。映射中的每个键值对包含两个元素，这意味着[映射](#)最多可以包含 50,000 个键值对。

## 其他限制

---

### 最大后退条数

[使用][历史引用运算符](#)对过去值的引用 取决于 Pine Script™ 运行时维护的历史缓冲区的大小，该缓冲区的最大限制为 5000 个柱。[此帮助中心页面](#) 讨论历史缓冲区以及如何使用 `max_bars_back` 参数或 `max_bars_back()` 函数更改其大小。

## 最大向前柱数

使用 定位绘图时 `xloc.bar_index`，可以使用大于当前条形索引值的条形索引值作为x坐标。未来最多可参考 500 个柱。

此示例展示了我们如何在 [input.int\(\)函数调用中使用](#) `maxval`参数 来限制用户定义的柱数，我们绘制一条投影线，使其永远不会超过限制：

```
//@version=5
indicator("Max bars forward example", overlay = true)

// This function draws a `line` using bar index x-coordinates.
drawLine(bar1, y1, bar2, y2) =>
    // Only execute this code on the last bar.
    if barstate.islast
        // Create the line only the first time this function is executed on the last
bar.
        var line lin = line.new(bar1, y1, bar2, y2, xloc.bar_index)
        // Change the line's properties on all script executions on the last bar.
        line.set_xy1(lin, bar1, y1)
        line.set_xy2(lin, bar2, y2)

// Input determining how many bars forward we draw the `line`.
int forwardBarsInput = input.int(10, "Forward Bars to Display", minval = 1, maxval =
500)

// Calculate the line's left and right points.
int    leftBar   = bar_index[2]
float  leftY     = high[2]
int    rightBar  = leftBar + forwardBarsInput
float  rightY    = leftY + (ta.change(high)[1] * forwardBarsInput)

// This function call is executed on all bars, but it only draws the `line` on the last
bar.
drawLine(leftBar, leftY, rightBar, rightY)
```

## 图表栏

图表上显示的条形数量取决于图表符号和时间范围的可用历史数据量以及您持有的账户类型。当所需的历史日期可用时，图表条的最小数量为：

- 高级计划有 20,000 条。
- Pro 和 Pro+ 计划有 10,000 个条。
- 其他计划有 5000 个条。

## 回测中的交易订单

回测策略时最多可下单9000单。使用深度回测时，限制为 200,000。

## 常问问题

### 在 Heikin Ashi 图表上获取真实 OHLC 价格

假设我们有一个 Heikin Ashi 图表（或 Renko、Kagi、PriceBreak 等），并且我们在上面添加了一个 Pine 脚本：

```
//@version=5
indicator("Visible OHLC", overlay=true)
c = close
plot(c)
```

您可能会看到该变量 `c` 是 Heikin Ashi 收盘价，与真实 OHLC 价格不同。因为 `close` 内置变量始终是与图表上的可见柱（或蜡烛）相对应的值。

那么，如果当前图表类型是非标准的，我们如何在 Pine Script™ 代码中获取真实的 OHLC 价格？我们应该将 `request.security` 功能与功能结合起来使用 `ticker.new`。这是一个例子：

```
//@version=5
indicator("Real OHLC", overlay = true)
t = ticker.new(syminfo.prefix, syminfo.ticker)
realC = request.security(t, timeframe.period, close)
plot(realC)
```

以类似的方式，我们可以获得其他 OHLC 价格：开盘价、最高价和最低价。

### 在标准图表上获取非标准 OHLC 值

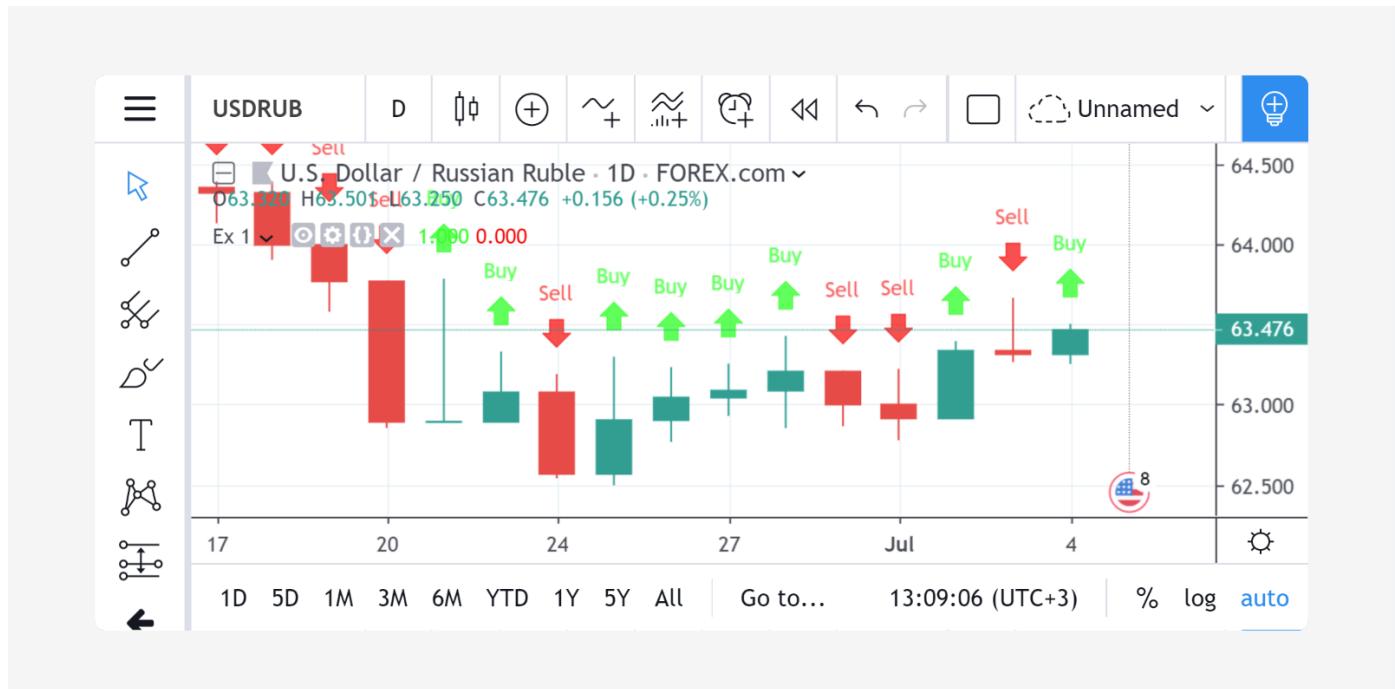
不建议对非标准图表类型（例如 Heikin Ashi 或 Renko）进行回溯测试，因为这些图表类型上的柱形图并不代表您在交易时会遇到的真实价格变动。如果您希望策略根据实际价格进入和退出，但仍使用基于 Heikin Ashi 的信号，则可以使用相同的方法在常规蜡烛图上获取 Heikin Ashi 值：

```
//@version=5
strategy("BarUpDn Strategy", overlay = true, default_qty_type =
strategy.percent_of_equity, default_qty_value = 10)
maxIdLossPcntInput = input.float(1, "Max Intraday Loss(%)")
strategy.risk.max_intraday_loss(maxIdLossPcntInput, strategy.percent_of_equity)
needTrade() => close > open and open > close[1] ? 1 : close < open and open < close[1]
? -1 : 0
trade = request.security(ticker.heikinashi(syminfo.tickerid), timeframe.period,
needTrade())
if trade == 1
    strategy.entry("BarUp", strategy.long)
if trade == -1
    strategy.entry("BarDn", strategy.short)
```

## 在图表上绘制箭头

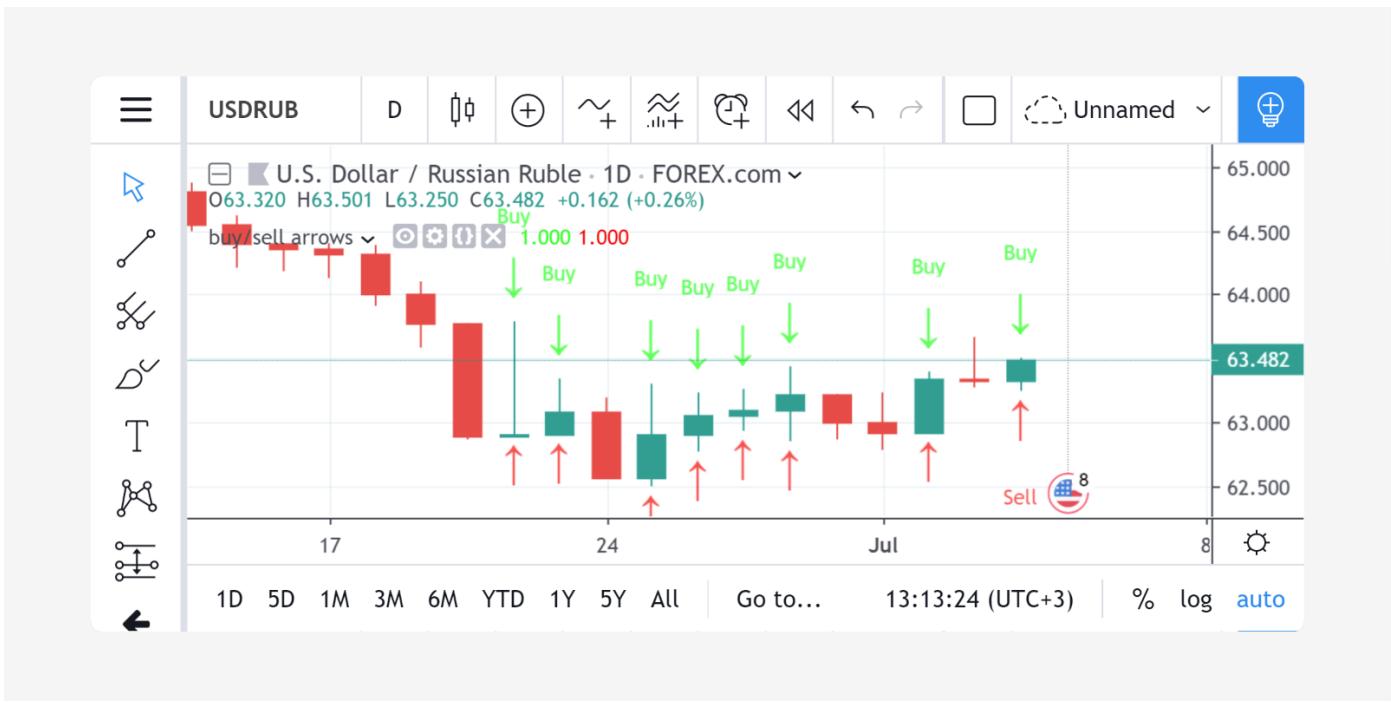
`shape.arrowup` 您可以使用带有 `style` 和 `的 plotshape shape.arrowdown`:

```
//@version=5
indicator('Ex 1', overlay = true)
condition = close >= open
plotshape(condition, color = color.lime, style = shape.arrowup, text = "Buy")
plotshape(not condition, color = color.red, style = shape.arrowdown, text = "Sell")
```



您可以将该 `plotchar` 函数与任何 unicode 字符一起使用:

```
//@version=5
indicator('buy/sell arrows', overlay = true)
condition = close >= open
plotchar(not condition, char='↓', color = color.lime, text = "Buy")
plotchar(condition, char='↑', location = location.belowbar, color = color.red, text =
"Sell")
```



## 绘制动态水平线

Pine Script™ 中有此功能 `hline`，但仅限于绘制常数值。这是一个简单的脚本，其中包含绘制变化的 `hline` 的解决方法：

```
//@version=5
indicator("Horizontal line", overlay = true)
plot(close[10], trackprice = true, offset = -9999)
// `trackprice = true` plots horizontal line on close[10]
// `offset = -9999` hides the plot
plot(close, color = #FFFFFF) // forces display
```

## 根据条件绘制垂直直线

```

//@version=5
indicator("Vertical line", overlay = true, scale = scale.none)
// scale.none means do not resize the chart to fit this plot
// if the bar being evaluated is the last bar in the chart (the most recent bar), then
cond is true
cond = barstate.islast
// when cond is true, plot a histogram with a line with height value of
100,000,000,000,000,000.00
// (10 to the power of 20)
// when cond is false, plot no numeric value (nothing is plotted)
// use the style of histogram, a vertical bar
plot(cond ? 10e20 : na, style = plot.style_histogram)

```

## 访问之前的值

```

//@version=5
//...
s = 0.0
s := nz(s[1]) // Accessing previous values
if (condition)
    s := s + 1

```

## 创 5 日新高

从当前柱线回溯 5 天，找到最高柱线，在当前柱线上方的价格水平处绘制星形字符



```

//@version=5
indicator("High of last 5 days", overlay = true)

```

```

// Milliseconds in 5 days: millisecs * secs * mins * hours * days
MS_IN_5DAYS = 1000 * 60 * 60 * 24 * 5

// The range check begins 5 days from the current time.
leftBorder = timenow - time < MS_IN_5DAYS
// The range ends on the last bar of the chart.
rightBorder = barstate.islast

// ----- Keep track of highest `high` during the range.
// Initialize `maxHi` with `var` on bar zero only.
// This way, its value is preserved, bar to bar.
var float maxHi = na
if leftBorder
    if not leftBorder[1]
        // Range's first bar.
        maxHi := high
    else if not rightBorder
        // On other bars in the range, track highest `high`.
        maxHi := math.max(maxHi, high)

// Plot level of the highest `high` on the last bar.
plotchar(rightBorder ? maxHi : na, "Level", "-", location.absolute, size = size.normal)
// When in range, color the background.
bgcolor(leftBorder and not rightBorder ? color.new(color.aqua, 70) : na)

```

## 计算数据集中的条数

获取加载数据集中所有条形的计数。对于根据柱数计算灵活的回溯期可能有用。

```

//@version=5
indicator("Bar Count", overlay = true, scale = scale.none)
plot(bar_index + 1, style = plot.style_histogram)

```

## 枚举一天内的柱数

```

//@version=5
indicator("My Script", overlay = true, scale = scale.none)

isNewDay() =>
    d = dayofweek
    na(d[1]) or d != d[1]

plot(ta.barssince(isNewDay()), style = plot.style_cross)

```

## 查找整个数据集的最高值和最低值

```
//@version=5
indicator("", "", true)

allTimetHi(source) =>
    var atHi = source
    atHi := math.max(atHi, source)

allTimetLo(source) =>
    var atLo = source
    atLo := math.min(atLo, source)

plot(allTimetHi(close), "ATH", color.green)
plot(allTimetLo(close), "ATL", color.red)
```

## 查询最后一个非na值

您可以使用下面的脚本来避免系列中出现间隙：

```
//@version=5
indicator("")
series = close >= open ? close : na
vw = fixnan(series)
plot(series, style = plot.style_linebr, color = color.red) // series has na values
plot(vw) // all na values are replaced with the last non-empty value
```

## 错误消息

### if语句太长： The if statement is too long

当[if语句](#)内的缩进代码对于编译器来说太大时，就会出现此错误。由于编译器的工作方式，您不会收到一条消息，告诉您到底有多少行代码超出了限制。现在唯一的解决方案是将[if语句](#)分解为更小的部分（函数或更小的[if语句](#)）。下面的示例显示了一个相当长的[if语句](#)；理论上，这会抛出：`line 4: if statement is too long`

```
//@version=5
indicator("My script")
var e = 0
if barstate.islast
    a = 1
    b = 2
    c = 3
    d = 4
    e := a + b + c + d
plot(e)
```

要修复此代码，您可以将这些行移动到它们自己的函数中：

```

//@version=5
indicator("My script")
var e = 0
doSomeWork() =>
    a = 1
    b = 2
    c = 3
    d = 4
    result = a + b + c + d
if barstate.islast
    e := doSomeWork()
plot(e)

```

## 脚本请求太多证券： Script requesting too many securities

脚本中的最大证券数限制为 40。如果将变量声明为 `request.security` 函数调用，然后使用该变量作为其他变量和计算的输入，则不会导致多次 `request.security` 调用。但是，如果您要声明一个调用的函数 `request.security`，则对该函数的每次调用都将被视为一次 `request.security` 调用。

从源代码来看，很难说会调用多少证券。以下示例 `request.security` 在编译后恰好有 3 次调用：

```

//@version=5
indicator("Securities count")
a = request.security(syminfo.tickerid, '42', close) // (1) first unique security call
b = request.security(syminfo.tickerid, '42', close) // same call as above, will not
produce new security call after optimizations

plot(a)
plot(a + 2)
plot(b)

sym(p) => // no security call on this line
    request.security(syminfo.tickerid, p, close)
plot(sym('D')) // (2) one indirect call to security
plot(sym('W')) // (3) another indirect call to security

request.security(syminfo.tickerid, timeframe.period, open) // result of this line is
never used, and will be optimized out

```

## 脚本无法翻译自null： Script could not be translated from: null

```
study($)
```

通常此错误出现在版本 1 Pine 脚本中，并且意味着代码不正确。Pine Script™ 版本 2（及更高版本）更擅长解释此类错误。所以你可以尝试通过在第一行添加特殊属性来切换到版本2。你会得到：`line 2: no viable alternative at character '$'`

```
// @version=2
study($)
```

## 第 2 行：字符 '\$' 处没有可行的替代方案：line 2: no viable alternative at character '\$'

此错误消息提示了错误所在。`$` 代替带有脚本标题的字符串。例如：

```
// @version=2
study("title")
```

## 不匹配的输入 <...> 期望： Mismatched input <...> expecting

与 `no viable alternative` 相同，但知道那个地方应该有什么。例子：`no viable alternative`

```
//@version=5
indicator("My Script")
plot(1)
```

`line 3: mismatched input 'plot' expecting 'end of line without line continuation'`

`plot` 要解决此问题，您应该在没有缩进的情况下以新行开始：

```
//@version=5
indicator("My Script")
plot(1)
```

## 循环太长 (> 500 毫秒) :Loop is too long (> 500 ms)

我们限制每个历史柱和实时报价的循环计算时间，以保护我们的服务器免受无限或非常长的循环的影响。此限制还导致计算时间过长的快速失败指标。例如，如果您有 5000 个柱，并且指标需要 500 毫秒来计算每个柱，则加载时间将超过 16 分钟：

```
//@version=5
indicator("Loop is too long", max_bars_back = 101)
s = 0
for i = 1 to 1e3 // to make it longer
    for j = 0 to 100
        if timestamp(2017, 02, 23, 00, 00) <= time[j] and time[j] < timestamp(2017, 02,
23, 23, 59)
            s := s + 1
plot(s)
```

或许可以优化算法来克服这个错误。在这种情况下，算法可以这样优化：

```
//@version=5
indicator("Loop is too long", max_bars_back = 101)
bar_back_at(t) =>
    i = 0
    step = 51
    for j = 1 to 100
        if i < 0
            i := 0
            break
        if step == 0
            break
        if time[i] >= t
            i := i + step
            i
        else
            i := i - step
            i
        step := step / 2
        step
    i

s = 0
for i = 1 to 1e3 // to make it longer
    s := s - bar_back_at(timestamp(2017, 02, 23, 23, 59)) +
        bar_back_at(timestamp(2017, 02, 23, 00, 00))
    s
plot(s)
```

## 脚本有太多局部变量:Script has too many local variables

如果脚本太大而无法编译，则会出现此错误。语句 `var=expression` 为创建一个局部变量 `var`。除此之外，需要注意的是，辅助变量可以在脚本编译过程中隐式创建。该限制适用于显式和隐式创建的变量。1000 个变量的限制单独应用于每个函数。事实上，放置在脚本全局作用域中的代码也隐式包装到主函数中，并且 1000 个变量的限制也适用于它。您可以尝试进行一些重构来避免此问题：

```
var1 = expr1  
var2 = expr2  
var3 = var1 + var2
```

可以转换为：

```
var3 = expr1 + expr2
```

## Pine Script™ 无法确定系列的引用长度。尝试在指标或策略函数中使用 max\_bars\_back:Pine Script™ cannot determine the referencing length of a series. Try using max\_bars\_back in the indicator or strategy function!.

如果 Pine Script™ 错误地自动检测脚本中使用的系列所需的最大长度，则会出现此错误。当脚本的执行流程不允许 Pine Script™ 检查条件语句（、`if` 或 `iff`）分支中系列的使用情况时？，就会发生这种情况，并且 Pine Script™ 无法自动检测引用了多远的系列。以下是导致此问题的脚本示例：

```
//@version=5  
indicator("Requires max_bars_back")  
test = 0.0  
if bar_index > 1000  
    test := ta.roc(close, 20)  
plot(test)
```

为了帮助 Pine Script™ 进行检测，您应该将 `max_bars_back` 参数添加到脚本 `indicator` 或 `strategy` 函数中：

```
//@version=5  
indicator("Requires max_bars_back", max_bars_back = 20)  
test = 0.0  
if bar_index > 1000  
    test := ta.roc(close, 20)  
plot(test)
```

您还可以通过将有问题的表达式从条件分支中取出来解决问题，在这种情况下 `max_bars_back` 不需要参数：

```
//@version=5  
indicator("My Script")  
test = 0.0  
roc20 = ta.roc(close, 20)  
if bar_index > 1000  
    test := roc20  
plot(test)
```

如果问题是由于变量而不是内置函数引起的（`vwma` 在我们的示例中），您可以使用该 `max_bars_back` 函数仅显式定义该变量的引用长度。这样做的优点是需要较少的运行时资源，但需要您识别有问题的变量，例如 `s` 以下示例中的变量：

```
//@version=5
indicator("My Script")
f(off) =>
    t = 0.0
    s = close
    if bar_index > 242
        t := s[off]
    t
plot(f(301))
```

这种情况可以通过使用 `max_bars_back` 函数仅定义变量的引用长度来解决 `s`，而不是定义脚本的所有变量：

```
//@version=5
indicator("My Script")
f(off) =>
    t = 0.0
    s = close
    max_bars_back(s, 301)
    if bar_index > 242
        t := s[off]
    t
plot(f(301))
```

当使用引用之前柱形图至 `bar_index[n]` 和的绘图时，从该柱形图接收的时间序列将用于在时间轴上定位绘图。因此，如果无法确定缓冲区的正确大小，则可能会发生此错误。为了避免这种情况，您需要使用.有关[绘图](#)的部分更详细地描述了此行为。`xloc = xloc.bar_index max_bars_back(time, n)`

## 至 Pine Script™ 版本 5

### 介绍

本指南记录了Pine Script™ 从 v4 到 v5 所做的更改。它将指导您将现有 Pine 脚本改编为 Pine Script™ v5。请参阅我们的[发行说明](#)，了解 Pine Script™ v5 中的新功能列表。

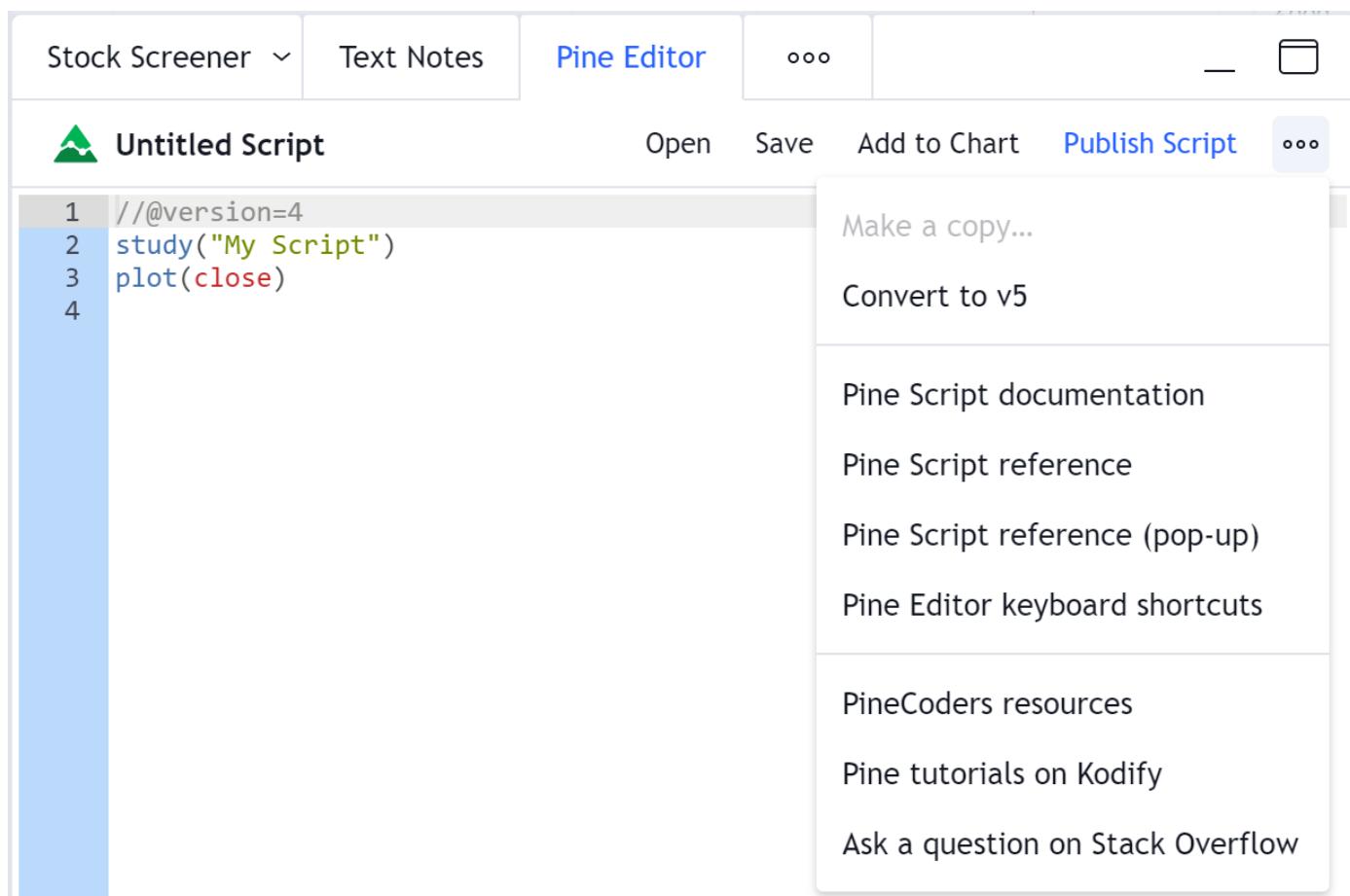
将旧脚本转换为 v5 所需的最常见的调整是：

- 将[Study\(\)](#)更改为[Indicator\(\)](#)（函数的签名未更改）。
- 重命名内置函数调用以包含新的命名空间（例如，v4 中的[Highest\(\)](#)变为v5 中的[ta.highest\(\)](#)）。
- 重组输入以使用更专业的 `input.*()` 功能。
- `transp` 通过使用[color.new\(\)](#)同时定义与参数一起使用的颜色和透明度，消除了对已弃用 `color` 参数的使用。

- 如果您在 v4 的 `Study()` resolution 中使用了和参数，则需要将它们更改为 v5 的 `Indicator()` 中的和。`resolution_gaps``timeframe``timeframe_gaps`

## v4 到 v5 转换器

Pine 编辑器包含一个实用程序，可自动将 v4 脚本转换为 v5。要访问它，请打开其中的脚本 `//@version=4`，然后在编辑器窗格右上角由三个点标识的“更多”菜单中选择“转换为 v5”选项：



并非所有脚本都可以自动从 v4 转换为 v5。如果您想要手动转换脚本或者您的指标在转换后返回编译错误，请使用以下部分来确定如何完成转换。您可以在本指南的 [常见脚本转换错误](#) 部分找到自动转换过程中可能遇到的一些错误以及如何修复这些错误的列表。

## 重命名的函数和变量

为了清晰和一致性，许多内置函数和变量在 v5 中被重命名。大多数更改的原因是在新命名空间中包含 v4 函数名称。例如，v4 中的 `sma()` 函数被移动到 `ta.` v5 中的命名空间：`ta.sma()`。不必记住新的命名空间；如果您在编辑器中键入不带命名空间的旧函数名称，然后按“自动完成”热键（`Ctrl+Space` 或 `MacOS` 上的 `Cmd+ Space`），则会出现一个显示匹配建议的弹出窗口：

[Stock Screener](#)[Text Notes](#)[Pine Editor](#)

ooo

[Untitled Script](#)[Open](#)[Save](#)[Add to Chart](#)

```
1 //@version=5
2 indicator("My Script")
3 v1 = highest
4 plot( ta.highest builtin function
5 ta.highestbars builtin function
```

不包括移动到新命名空间的函数，只有两个函数被重命名：

- `study()` 现在是[Indicator\(\)](#)。
- `tickerid()` 现在是[ticker.new\(\)](#)。

重命名的函数和变量的完整列表可以在本指南的所有变量、函数和参数名称更改部分中找到。

## 重命名的函数参数

更改了一些内置函数的参数名称以改进命名法。这对大多数脚本没有影响，但如果您在调用函数时使用这些参数名称，则它们将需要调整。例如，我们对所有提及进行了标准化：

```
// Valid in v4. Not valid in v5.
timev4 = time(resolution = "1D")
// Valid in v5.
timev5 = time(timeframe = "1D")
// Valid in v4 and v5.
timeBoth = time("1D")
```

重命名的函数参数的完整列表可以在本指南的所有变量、函数和参数名称更改部分中找到。

## 删除了 `rsi()` 重载

在 v4 中，[rsi\(\)](#) 函数有两个不同的重载：

- `rsi(series float, simple int)` 对于正常的 RSI 计算，以及
- `rsi(series float, series float)` 对于 MFI 指标中使用的过载，其计算相当于  $100.0 - (100.0 / (1.0 + arg1 / arg2))$

这导致单个内置函数以两种截然不同的方式表现，并且很难区分应用哪一种，因为它取决于第二个参数的类型。结果，许多指标滥用了该功能并显示了错误的结果。为了避免这种情况，在 v5 中删除了第二个重载。

v5 中的 `ta.rsi()` 函数仅接受“简单 int”参数作为其 `length` 参数。如果您的 v4 代码使用了现已弃用的带有第二个参数的函数重载 `float`，您可以使用以下公式替换整个 `rsi()` 调用，该公式等效：

```
100.0 - (100.0 / (1.0 + arg1 / arg2))
```

请注意，当您的 v4 代码使用“series int”值作为 `rsi()` 的第二个参数时，它会自动转换为“series float”并使用该函数的第二个重载。虽然这在语法上是正确的，但它很可能不会产生您期望的结果。在 v5 中，`ta.rsi()` 需要一个“简单 int”作为参数 `length`，这排除了动态（或“系列”）长度。原因是 RSI 计算使用 `ta.rma()`（移动平均线，这与 `ta.ema()` 类似），因为它依赖于使用先前柱线值的长度相关递归过程。这使得不可能通过逐条变化的“系列”长度来获得正确的结果。

如果您的 v4 代码使用的长度为“const int”、“input int”或“simple int”，则无需进行任何更改。

## 保留关键字

许多单词是保留的，不能用于变量或函数名称。他们是：`catch`, `class`, `do`, `ellipse`, `in`, `is`, `polygon`, `range`, `return`, `struct`, `text`, `throw`, `try`。如果您的 v4 指标使用其中任何一个，请重命名变量或函数以使脚本在 v5 中运行。

## 删除了 `iff()` 和 `offset()`

`iff()` 和 `offset()` 函数已被删除。使用 `iff()` 函数的代码可以使用三元运算符重写：

```
// iff(<condition>, <return_when_true>, <return_when_false>)
// Valid in v4, not valid in v5
barColorIff = iff(close >= open, color.green, color.red)
// <condition> ? <return_when_true> : <return_when_false>
// Valid in v4 and v5
barColorTernary = close >= open ? color.green : color.red
```

请注意，三元运算符是“惰性”计算的；仅计算所需的值（取决于条件对 `true` 或的评估 `false`）。这与 `iff()` 不同，后者总是评估两个值但仅返回相关的值。

有些函数需要对每个柱进行评估才能正确计算，因此您需要通过在三元之前对它们进行预评估来为这些函数做出特殊规定：

```
// `iff()` in v4: `highest()` and `lowest()` are calculated on every bar
v1 = iff(close > open, highest(10), lowest(10))
plot(v1)
// In v5: forced evaluation on every bar prior to the ternary statement.
h1 = ta.highest(10)
l1 = ta.lowest(10)
v1 = close > open ? h1 : l1
plot(v1)
```

`offset()` 函数已被弃用，因为更具可读性的 `[]` 运算符是等效的：

```
// Valid in v4. Not valid in v5.  
prevClosev4 = offset(close, 1)  
// Valid in v4 and v5.  
prevClosev5 = close[1]
```

## 将 `input()` 拆分为多个函数

v4 `input()` 函数变得挤满了过多的重载和参数。我们将其功能拆分为不同的功能，以清理空间并提供更强大的结构来容纳计划添加的输入。每个新函数都使用它所替换的 `input.*`` v4 调用的类型名称。`input()` 例如，现在有一个专门的 `input.float()` 函数取代了 v4 调用。请注意，在 v5 中仍然可以使用，但由于只有 `input.float()` 允许使用 `、` 等参数，因此功能更强大。另请注意，`input.int()` 是唯一不使用其等效 v4 名称的专用输入函数。这些常量已被删除，因为它们被用作参数的实参，而该参数已被弃用。`input(1.0, type = input.float)`` `input(1.0)`` `minval`` `maxval`` `input.integer`` `input.*`` `type`

例如，要转换使用类型的输入的 v4 脚本 `input.symbol`，则必须在 v5 中使用 `input.symbol()` 函数：

```
// Valid in v4. Not valid in v5.  
aaplTicker = input("AAPL", type = input.symbol)  
// Valid in v5  
aaplTicker = input.symbol("AAPL")
```

`input()` 函数在 v5 中仍然存在，但形式更简单，参数更少。它的优点是可以从用于的参数中自动检测输入类型“bool/color/int/float/string/source” `defval`：

```
// Valid in v4 and v5.  
// While "AAPL" is a valid symbol, it is only a string here because `input.symbol()` is  
// not used.  
tickerString = input("AAPL", title = "Ticker string")
```

## 一些函数参数现在需要内置参数

在 v4 中，内置常量（例如 `plot.style_area` 在调用 Pine Script™ 函数时用作参数）对于特定类型的预定义值。例如，`barmerge.lookahead_on` 的值是 `true`，因此在 `security()` 函数调用中向形参提供实参时，`true` 您可以使用它来代替命名常量。我们发现这是一个常见的混乱根源，它导致毫无戒心的程序员编写出产生意想不到结果的代码。`true`` `lookahead`

在 v5 中，必须使用正确的内置命名常量作为需要它们的函数参数的实参：

```

// Not valid in v5: `true` is used as an argument for `lookahead`.
request.security(syminfo.tickerid, "1D", close, lookahead = true)
// Valid in v5: uses a named constant instead of `true`.
request.security(syminfo.tickerid, "1D", close, lookahead = barmerge.lookahead_on)

// Would compile in v4 because `plot.style_columns` was equal to 5.
// Won't compile in v5.
a = 2 * plot.style_columns
plot(a)

```

要将脚本从 v4 转换为 v5, 请确保使用正确命名的内置常量作为函数参数。

## 弃用了 `transp` 参数

`transp=`许多 v4 绘图函数的签名中使用的参数已被弃用, 因为它会干扰 RGB 功能。现在必须将透明度与颜色一起指定为 `color`、`textcolor` 等参数的参数。在这些情况下, 将需要[color.new\(\)](#)或[color.rgb\(\)](#)函数来连接颜色及其透明度。

请注意, 在 v4 中, `bgcolor()`和`fill()`函数有一个可选 `transp` 参数, 使用默认值 90。这意味着下面的代码可以显示布林带, 在两个带之间具有半透明填充和半透明填充即使 `transp` 在其`bgcolor()`和`fill()`调用中没有为参数使用参数, 带交叉价格处的背景颜色:

```

//@version=4
study("Bollinger Bands", overlay = true)
[middle, upper, lower] = bb(close, 5, 4)
plot(middle, color=color.blue)
p1PlotID = plot(upper, color=color.green)
p2PlotID = plot(lower, color=color.green)
crossUp = crossover(high, upper)
crossDn = crossunder(low, lower)
// Both `fill()` and `bgcolor()` have a default `transp` of 90
fill(p1PlotID, p2PlotID, color = color.green)
bgcolor(crossUp ? color.green : crossDn ? color.red : na)

```

在 v5 中, 我们需要明确提及颜色的 90 透明度, 产生:

```

//@version=5
indicator("Bollinger Bands", overlay = true)
[middle, upper, lower] = ta.bb(close, 5, 4)
plot(middle, color=color.blue)
p1PlotID = plot(upper, color=color.green)
p2PlotID = plot(lower, color=color.green)
crossUp = ta.crossover(high, upper)
crossDn = ta.crossunder(low, lower)
var TRANSP = 90
// We use `color.new()` to explicitly pass transparency to both functions
fill(p1PlotID, p2PlotID, color = color.new(color.green, TRANSP))
bgcolor(crossUp ? color.new(color.green, TRANSP) : crossDn ? color.new(color.red, TRANSP) : na)

```

## 更改了 `time()` 和 `time_close()` 的默认会话天数

`time()` 和 `time_close()` `session` 函数中使用的字符串的默认日期集（由 `input.session()` 返回）已从（星期一到星期五）更改为（星期日到星期六）： "23456" `` "1234567"

```

// On symbols that are traded during weekends, this will behave differently in v4 and
// v5.
t0 = time("1D", "1000-1200")
// v5 equivalent of the behavior of `t0` in v4.
t1 = time("1D", "1000-1200:23456")
// v5 equivalent of the behavior of `t0` in v5.
t2 = time("1D", "1000-1200:1234567")

```

这种行为变化应该不会对在周末休市的传统市场上运行的脚本产生太大影响。如果确保会话定义在 v5 代码中保留其 v4 行为对您很重要，请添加 ":23456" 到会话字符串中。有关详细信息，请参阅本手册中 [有关会话的页面](#)。

## `strategy.exit()` 现在必须做一些事情

允许 `strategy.exit()` 函数闲逛的日子已经一去不复返了。现在，它实际上必须通过使用以下参数中的至少一个来对策略产生影响：`profit`、`limit`、`loss`、`stop` 或以下对之一：与或 `trail_offset` 组合。当在将策略转换为 v5 时使用不满足这些条件的 `Strategy.exit()` 时会触发错误，您可以安全地消除这些行，因为它们无论如何都不会在您的代码中执行任何操作。`trail_price``trail_points`

## 常见的脚本转换错误

### “plot”/“hline”调用中的参数“style”/“linestyle”无效

为了使其工作，您需要更改用于内置常量的 `plot()` 和 `hline()` `style` 中的和 `linestyle` 参数的 “int” 参数：

```

// Will cause an error during conversion
plotStyle = input(1)

```

```

hlineStyle = input(1)
plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)

// Will work in v5
//@version=5
indicator("")
plotStyleInput = input.string("Line", options = ["Line", "Stepline", "Histogram",
"Cross", "Area", "Columns", "Circles"])
hlineStyleInput = input.string("Solid", options = ["Solid", "Dashed", "Dotted"])

plotStyle = plotStyleInput == "Line" ? plot.style_line :
    plotStyleInput == "Stepline" ? plot.style_stepline :
    plotStyleInput == "Histogram" ? plot.style_histogram :
    plotStyleInput == "Cross" ? plot.style_cross :
    plotStyleInput == "Area" ? plot.style_area :
    plotStyleInput == "Columns" ? plot.style_columns :
    plot.style_circles

hlineStyle = hlineStyleInput == "Solid" ? hline.style_solid :
    hlineStyleInput == "Dashed" ? hline.style_dashed :
    hline.style_dotted

plot(close, style = plotStyle)
hline(100, linestyle = hlineStyle)

```

有关详细信息，请参阅本指南的[某些函数参数现在需要内置参数部分](#)。

## 未声明的标识符 'input.%input\_name%'

要解决此问题，请将 `input.*` 从代码中删除常量：

```

// Will cause an error during conversion
_integer = input.integer
_bool = input.bool
i1 = input(1, "Integer", _integer)
i2 = input(true, "Boolean", _bool)

// Will work in v5
i1 = input.int(1, "Integer")
i2 = input.bool(true, "Boolean")

```

有关详细信息，请参阅用户手册的[输入](#)页面，以及本指南的[某些函数参数现在需要内置参数部分](#)。

## “strategy.close”调用中的参数“when”无效

这是由于[strategy.entry\(\)](#)和[strategy.close\(\)](#)之间的混淆造成的。

[Strategy.close\(\)](#)的第二个参数是`when`，它需要一个“bool”参数。在v4中，允许使用`strategy.long`参数，因为它是“bool”。然而，在v5中，命名内置常量必须用作参数，因此`strategy.long`不再允许作为参数的参数`when`。

此代码中的调用相当于，这是v5中必须使用的：`strategy.close("Short", strategy.long)` `strategy.close("Short")`

```
// Will cause an error during conversion
if (longCondition)
    strategy.close("Short", strategy.long)
    strategy.entry("Long", strategy.long)

// Will work in v5:
if (longCondition)
    strategy.close("Short")
    strategy.entry("Long", strategy.long)
```

有关详细信息，请参阅本指南的[某些函数参数现在需要内置参数部分](#)。

## 无法使用参数“minval”=“%value%”调用“input.int”。使用了“literal float”类型的参数，但需要“const int”

`minval`在v4中，当输入“int”值时可以传递“float”参数。这在v5中不再可能：“int”输入需要“int”值：

```
// Works in v4, will break on conversion because minval is a 'float' value
int_input = input(1, "Integer", input.integer, minval = 1.0)

// Works in v5
int_input = input.int(1, "Integer", minval = 1)
```

有关详细信息，请参阅用户手册的[输入](#)页面，以及本指南的[某些函数参数现在需要内置参数部分](#)。

## 所有变量、函数和参数名称更改

### 删除的函数和变量

v4	v5
input.bool 输入	取而代之 input.bool()
input.color 输入	取而代之 input.color()
input.float 输入	取而代之 input.float()
input.integer 输入	取而代之 input.int()
input.resolution 输入	取而代之 input.timeframe()
input.session 输入	取而代之 input.session()
input.source 输入	取而代之 input.source()
input.string 输入	取而代之 input.string()
input.symbol 输入	取而代之 input.symbol()
input.time 输入	取而代之 input.time()
iff()	使用 ?: 运算符代替
offset()	使用 [ ] 运算符代替

## 重命名的函数和参数

### 没有命名空间变化

v4	v5
study(<...>, resolution, resolution_gaps, <...>)	indicator(<...>, timeframe, timeframe_gaps, <...>)
strategy.entry(long)	strategy.entry(direction)
strategy.order(long)	strategy.order(direction)
time(resolution)	time(timeframe)
time_close(resolution)	time_close(timeframe)
nz(x, y)	nz(source, replacement)

### 用于技术分析函数和变量的“ta”命名空间

v4	v5
指示函数和变量	
accdist	ta.accdist
alma()	ta.alma()

atr()	ta.atr()
bb()	ta.bb()
bbw()	ta.bbw()
cci()	ta.cci()
cmo()	ta.cmo()
cog()	ta.cog()
dmi()	ta.dmi()
ema()	ta.ema()
hma()	ta.hma()
iii	ta.iii
kcc()	ta.kcc()
kcw()	ta.kcw()
linreg()	ta.linreg()
macd()	ta.macd()
mfi()	ta.mfi()
mom()	ta.mom()
nvi	ta.nvi
obv	ta.obv
pvi	ta.pvi
pvt	ta.pvt
rma()	ta.rma()
roc()	ta.roc()
rsi(x, y)	ta.rsi(source, length)
sar()	ta.sar()
sma()	ta.sma()
stoch()	ta.stoch()
supertrend()	ta.supertrend()
swma(x)	ta.swma(source)
tr	ta.tr
tr()	ta.tr()
tsi()	ta.tsi()
vwap	ta.vwap
vwap(x)	ta.vwap(source)
vwma()	ta.vwma()
wad	ta.wad
wma()	ta.wma()
wpr()	ta.wpr()
wvad	ta.wvad
配套功能	
barsince()	ta.barsince()

change()	ta.change()
correlation(source_a, source_b, length)	ta.correlation(source1, source2, length)
cross(x, y)	ta.cross(source1, source2)
crossover(x, y)	ta.crossover(source1, source2)
crossunder(x, y)	ta.crossunder(source1, source2)
cum(x)	ta.cum(source)
dev()	ta.dev()
falling()	ta.falling()
highest()	ta.highest()
highestbars()	ta.highestbars()
lowest()	ta.lowest()
lowestbars()	ta.lowestbars()
median()	ta.median()
mode()	ta.mode()
percentile_linear_interpolation()	ta.percentile_linear_interpolation()
percentile_nearest_rank()	ta.percentile_nearest_rank()
percentrank()	ta.percentrank()
pivothigh()	ta.pivothigh()
pivotlow()	ta.pivotlow()
range()	ta.range()
rising()	ta.rising()
stdev()	ta.stdev()
valuewhen()	ta.valuewhen()
variance()	ta.variance()

## 数学相关函数和变量的“math”命名空间

v4	v5
abs(x)	math.abs(number)
acos(x)	math.acos(number)
asin(x)	math.asin(number)
atan(x)	math.atan(number)
avg()	math.avg()
ceil(x)	math.ceil(number)
cos(x)	math.cos(angle)
exp(x)	math.exp(number)
floor(x)	math.floor(number)
log(x)	math.log(number)
log10(x)	math.log10(number)
max()	math.max()
min()	math.min()
pow()	math.pow()
random()	math.random()
round(x, precision)	math.round(number, precision)
round_to_mintick(x)	math.round_to_mintick(number)
sign(x)	math.sign(number)
sin(x)	math.sin(angle)
sqrt(x)	math.sqrt(number)
sum()	math.sum()
tan(x)	math.tan(angle)
todegrees()	math.todegrees()
toradians()	math.toradians()

## 请求外部数据的函数的“request”命名空间

v4	v5
financial()	request.financial()
quandl()	request.quandl()
security(<...>, resolution, <...>)	request.security(<...>, timeframe, <...>)
splits()	request.splits()
dividends()	request.dividends()
earnings()	request.earnings()

## 帮助创建股票代码的函数的“股票代码”命名空间

v4	v5
heikinashi()	ticker.heikinashi()
kagi()	ticker.kagi()
linebreak()	ticker.linebreak()
pointfigure()	ticker.pointfigure()
renko()	ticker.renko()
tickerid()	ticker.new()

## 操作字符串的函数的“str”命名空间

v4	v5
tostring(x, y)	str.tostring(value, format)
tonumber(x)	str.tonumber(string)

# 到 Pine Script™ 版本 4

这是将 Pine Script™ 代码 @version=3 从 @version=4 .

## 转换器

Pine 编辑器附带了一个实用程序，可以自动将 v3 指标和策略转换为 v4。要访问它，请打开其中的脚本 // @version=3，然后选择下拉菜单中的选项： Convert to v4 More

```
//@version=3
study("My Script")
plot(close)
```

Pine Editor context menu options:

- Make a copy...
- Convert to v4
- Pine Script documentation
- Pine Script reference
- Pine Script reference (pop-up)
- Pine Editor keyboard shortcuts
- PineCoders resources
- Pine tutorials on Kodify
- Ask a question on Stack Overflow

并非所有脚本都可以自动从 v3 转换为 v4。如果您想手动转换脚本或者您的指标在转换后返回编译错误，请参阅下面的指南以获取更多信息。

## 重命名内置常量、变量和函数

在 Pine Script™ v4 中，以下内置常量、变量和函数已重命名：

- 颜色常量（例如 `red`）被移动到 `color.*` 命名空间（例如 `color.red`）。
- 该 `color` 函数已重命名为 `color.new`。
- 类型常量 `input()`（例如 `integer`）被移动到 `input.*` 命名空间（例如 `input.integer`）。
- 绘图样式常量（例如 `histogram style`）被移动到 `plot.style_*` 命名空间（例如 `plot.style_histogram`）。
- 函数的样式常量 `hline`（例如 `dotted` 样式）被移动到 `hline.style_*` 命名空间（例如 `hline.style_dotted`）。
- 星期几的常量（例如 `sunday`）被移动到 `dayofweek.*` 命名空间（例如 `dayofweek.sunday`）。
- 当前图表时间范围的变量（例如 `period`, `isintraday`）被移动到 `timeframe.*` 命名空间（例如 `timeframe.period`, `timeframe.isintraday`）。
- 该 `interval` 变量已重命名为 `timeframe.multiplier`。
- 和变量分别重命名为  `tickerid` `syminfo.ticker` `syminfo.tickerid`
- 包含柱索引值的变量 `n` 已重命名为 `bar_index`。

重命名上述所有内容的原因是为了构建标准语言工具并使代码处理更容易。新名称根据公共前缀下的分配进行分组。例如，如果您在编辑器中键入“颜色”并按 Ctrl + 空格键，您将看到包含所有可用颜色常量的列表。

## 显式变量类型声明

在 Pine Script™ v4 中，不再可能在声明时创建具有未知数据类型的变量。这样做是为了避免变量类型在使用 na 值初始化后发生更改时出现的许多问题。从现在开始，`float` 在声明具有 na 值的变量时，您需要使用关键字或类型函数（例如）显式指定它们的类型：

```
//@version=4
study("Green Candle Close")
// We expect `src` to hold float values, so we declare it with the `float` keyword
float src = na
if close > open
    src := close
plot(src)
```

## 至 Pine Script™ 版本 3

本文档有助于将 Pine Script™ 代码 `@version=2` 从 `@version=3`。

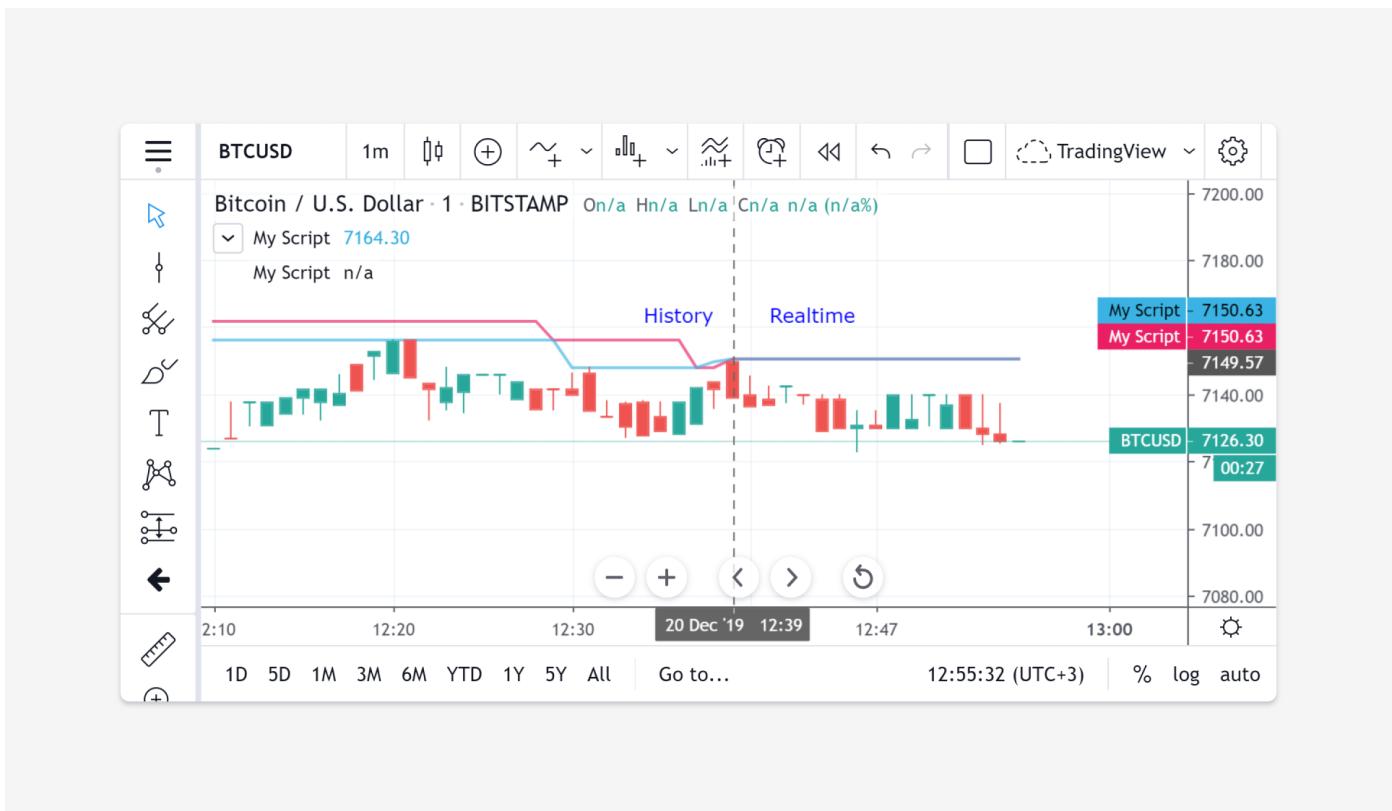
### 安全功能的默认行为已更改

让我们看一下简单的 `security` 函数用例。在日内图表上添加该指标：

```
// Add this indicator on an intraday (e.g., 30 minutes) chart
//@version=2
study("My Script", overlay=true)
s = security(tickerid, 'D', high, false)
plot(s)
```

该指标是根据历史数据计算的，并在一定程度上展望了未来。在每个交易时段的第一根柱上，指标会绘制全天的最高价格。这在某些情况下对于分析可能很有用，但对于回溯测试策略不起作用。

我们对此进行了研究，并在 Pine Script™ 版本 3 中进行了更改。如果使用 `//@version=3` 指令编译该指标，我们会得到完全不同的图片：



但旧的行为仍然可用。我们向 `security` 名为 的函数（第五个）添加了一个参数 `lookahead`。

它可以采用两种不同值的形式：（`barmerge.lookahead_off` 这是 Pine Script™ 版本 3 的默认值）或 `barmerge.lookahead_on`（这是 Pine Script™ 版本 2 的默认值）。

## 自引用变量被删除

Pine Script™ 版本 2 代码段，包含自引用变量：

```
//@version=2
//...
s = nz(s[1]) + close
```

使用 Pine Script™ 版本 3 编译这段代码将会出现错误。它应该重写为：`Undeclared identifier 's'`

```
//@version=3
//...
s = 0.0
s := nz(s[1]) + close
```

`s` 现在是一个在第 3 行初始化的可变变量。在第 3 行，初始值向 Pine Script™ 编译器提供有关变量类型的信息。在这个例子中它是一个浮点数。

在某些情况下，您可以 `s` 使用值初始化该可变变量（例如）`na`。但在复杂的情况下这是行不通的。

## 前向引用变量被删除

```
//@version=2
//...
d = nz(f[1])
e = d + 1
f = e + close
```

本例中 `f` 是一个前向引用变量，因为它在声明和初始化之前在第 3 行被引用。在 Pine Script™ 版本 3 中，这会给您一个错误。此示例应在 Pine Script™ 版本 3 中重写，如下所示： Undeclared identifier 'f'

```
//@version=3
//...
f = 0.0
d = nz(f[1])
e = d + 1
f := e + close
```

## 解决安全表达式中可变变量的问题

当您将脚本迁移到版本 3 时，删除自引用和前向引用变量后，Pine Script™ 编译器可能会给出错误：

```
//@version=3
//...
s = 0.0
s := nz(s[1]) + close
t = security(tickerid, period, s)
Cannot use mutable variable as an argument for security function!
```

自从 Pine Script™（即版本 2）中引入了可变变量以来，就存在此限制。它可以像以前一样解决：在函数中使用可变变量包装代码：

```
//@version=3
//...
calcS() =>
    s = 0.0
    s := nz(s[1]) + close
t = security(tickerid, period, calcS())
```

## 禁止使用布尔值进行数学运算

在 Pine Script™ v2 中，存在将布尔值隐式转换为数字类型的规则。在 v3 中这是被禁止的。相反，将数字类型转换为布尔值（0 和 `na` 值是 `false`，所有其他数字都是 `true`）。示例（在 v2 中，此代码可以正常编译）：

```
//@version=2
study("My Script")
s = close >= open
s1 = close[1] >= open[1]
s2 = close[2] >= open[2]
sum = s + s1 + s2
col = sum == 1 ? white : sum == 2 ? blue : sum == 3 ? red : na
bgcolor(col)
```

变量 `s`、`s1` 和 `s2` 均为 `bool` 类型。但在第 6 行，我们添加了其中的三个并将结果存储在变量中 `sum`。`sum` 是一个数字，因为我们不能添加布尔值。布尔值被隐式转换为数字（`true` 值 to `1.0` 和 `false` to `0.0`），然后将它们相加。

这种方法会导致更复杂的脚本中出现无意的错误。这就是为什么我们不再允许布尔值到数字的隐式转换。

如果您尝试将此示例编译为 Pine Script™ v3 代码，您将收到错误：这意味着您不能将加法运算符与布尔值一起使用。要使该示例在 Pine Script™ v3 中运行，您可以执行以下操作：`Cannot call operator + with arguments (series_bool, series_bool); <...>`

```
//@version=3
study("My Script")
bton(b) =>
    b ? 1 : 0
s = close >= open
s1 = close[1] >= open[1]
s2 = close[2] >= open[2]
sum = bton(s) + bton(s1) + bton(s2)
col = sum == 1 ? white : sum == 2 ? blue : sum == 3 ? red : na
bgcolor(col)
```

如果您确实需要，函数 `bton`（布尔到数字的缩写）可以显式地将任何布尔值转换为数字。

## 我在哪里可以获得更多信息？

- 所有 Pine Script™ 运算符、变量和函数的描述可以在[参考手册](#)中找到。
- 使用 TradingView 的内置脚本之一中的代码开始。打开一个新图表，然后单击工具栏上的“Pine Editor”按钮。进入编辑器窗口后，单击“打开”按钮，然后从下拉列表中选择“内置脚本...”以打开包含 TradingView 内置脚本列表的对话框。
- 有一个 TradingView 公共聊天室专门用于 Pine Script™[回答](#)，我们社区的活跃开发人员可以互相帮助。
- 有关 Pine Script™（以及其他功能）的主要版本和修改的信息定期发布在[TradingView 的博客](#)上。
- TradingView 的[社区脚本](#)包含所有用户发布的脚本。还可以使用“指标和策略”按钮和脚本搜索对话框的“社区脚本”选项卡从图表中访问它们。

## [外部资源](#)

- [TradingView 上的 PineCoders](#)帐户为 Pine Script™ 程序员发布有用的信息。他们的[网站](#)上也有内容。
- [Kodify](#)为初学者和经验丰富的程序员提供了有关各种主题的 TradingView 教程。主题包括绘图、警报、策略订单以及完整的示例指标和策略。
- [Backtest Rookies](#)发布了高质量的博客文章，重点关注在 Pine Script™ 中实现特定任务。
- 您可以在[StackOverflow](#) [pine-script] 上的标签中询问有关 Pine Script™ 编程的问题。