Contents

AccessControlKind.Allow

AccessControlKind.Deny

Access.Database

ActiveDirectory.Domains

AdobeAnalytics.Cubes

AdoDotNet.DataSource

AdoDotNet.Query

AnalysisServices.Database

AnalysisServices.Databases

AzureStorage.BlobContents

AzureStorage.Blobs

AzureStorage.DataLake

AzureStorage.DataLakeContents

AzureStorage.Tables

Cdm.Contents

Csv.Document

CsvStyle.QuoteAfterDelimiter

CsvStyle.QuoteAlways

Cube.AddAndExpandDimensionColumn

Cube.AddMeasureColumn

Cube.ApplyParameter

Cube.AttributeMemberId

Cube.AttributeMemberProperty

Cube.CollapseAndRemoveColumns

Cube.Dimensions

Cube.DisplayFolders

Cube.MeasureProperties

Cube.MeasureProperty

Cube.Measures

Cube.Parameters

Cube.Properties

Cube.PropertyKey

Cube.ReplaceDimensions

Cube.Transform

DB2.Database

Essbase.Cubes

Excel.CurrentWorkbook

Excel.Workbook

Exchange.Contents

Facebook.Graph

File.Contents

Folder.Contents

Folder.Files

GoogleAnalytics.Accounts

Hdfs.Contents

Hdfs.Files

HdInsight.Containers

HdInsight.Contents

HdInsight.Files

Html.Table

Identity.From

Identity.IsMemberOf

IdentityProvider.Default

Informix.Database

Json.Document

Json.FromValue

MySQL.Database

OData.Feed

ODataOmitValues.Nulls

Odbc.DataSource

Odbc.InferOptions

Odbc.Query

OleDb.DataSource

OleDb.Query

Oracle.Database

Parquet.Document

Pdf.Tables

PostgreSQL.Database

RData.FromBinary

Salesforce.Data

Salesforce.Reports

SapBusinessWarehouse.Cubes

SapBusinessWarehouseExecutionMode.DataStream

SapBusinessWarehouseExecutionMode.BasXml

SapBusinessWarehouseExecutionMode.BasXmlGzip

SapHana.Database

SapHanaDistribution.All

SapHanaDistribution.Connection

SapHanaDistribution.Off

SapHanaDistribution.Statement

SapHanaRangeOperator.Equals

SapHanaRangeOperator.GreaterThan

SapHanaRangeOperator.GreaterThanOrEquals

SapHanaRangeOperator.LessThan

SapHanaRangeOperator.LessThanOrEquals

SapHanaRangeOperator.NotEquals

SharePoint.Contents

SharePoint.Files

SharePoint.Tables

Soda.Feed

Sql.Database

Sql.Databases

Sybase.Database

Teradata.Database

WebAction.Request

Web.BrowserContents

Date.IsInNextDay

Date.IsInNextMonth

Date.IsInNextNDays

Date.IsInNextNMonths

Date.IsInNextNQuarters

Date.IsInNextNWeeks

Date.IsInNextNYears

Date.IsInNextQuarter

Date.IsInNextWeek

Date.IsInNextYear

Date.IsInPreviousDay

Date.IsInPreviousMonth

Date.IsInPreviousNDays

Date.IsInPreviousNMonths

Date.IsInPreviousNQuarters

Date.IsInPreviousNWeeks

Date.IsInPreviousNYears

Date.IsInPreviousQuarter

Date.IsInPreviousWeek

Date.IsInPreviousYear

Date.IsInYearToDate

Date.IsLeapYear

Date.Month

Date.MonthName

Date.QuarterOfYear

Date.StartOfDay

Date.StartOfMonth

Date.StartOfQuarter

Date.StartOfWeek

Date.StartOfYear

Date.ToRecord

Date.ToText

List.Last

List.LastN

List.MatchesAll

List.MatchesAny

List.Max

List.MaxN

List.Median

List.Min

List.MinN

List.Mode

List.Modes

List.NonNullCount

List.Numbers

List.PositionOf

List.PositionOfAny

List.Positions

List.Product

List.Random

List.Range

List.RemoveFirstN

List.RemoveItems

List.RemoveLastN

List.RemoveMatchingItems

List.RemoveNulls

List.RemoveRange

List.Repeat

List.ReplaceMatchingItems

List.ReplaceRange

List.ReplaceValue

List.Reverse

List.Select

List.Single

Number.BitwiseNot

Number.BitwiseOr

Number.BitwiseShiftLeft

Number.BitwiseShiftRight

Number.BitwiseXor

Number.Combinations

Number.Cos

Number.Cosh

Number.E

Number.Epsilon

Number.Exp

Number.Factorial

Number.From

Number.FromText

Number.IntegerDivide

Number.IsEven

Number.IsNaN

Number.IsOdd

Number.Ln

Number.Log

Number.Log10

Number.Mod

Number.NaN

Number.NegativeInfinity

Number.Permutations

Number.PI

Number.PositiveInfinity

Number.Power

Number.Random

Number.RandomBetween

Number.Round

Number.RoundAwayFromZero

ItemExpression.From

ItemExpression.Item

JoinAlgorithm.Dynamic

JoinAlgorithm.LeftHash

JoinAlgorithm.LeftIndex

JoinAlgorithm.PairwiseHash

JoinAlgorithm.RightHash

JoinAlgorithm.RightIndex

JoinAlgorithm.SortMerge

JoinKind.FullOuter

JoinKind.Inner

JoinKind.LeftAnti

JoinKind.LeftOuter

JoinKind.RightAnti

JoinKind.RightOuter

JoinSide.Left

JoinSide.Right

Occurrence.All

Occurrence.First

Occurrence.Last

Order.Ascending

Order.Descending

RowExpression.Column

RowExpression.From

RowExpression.Row

Table.AddColumn

Table.AddIndexColumn

Table.AddJoinColumn

Table.AddKey

Table.AggregateTableColumn

Table.AlternateRows

Table.Buffer

Table.Column

Table.ColumnCount

Table.ColumnNames

Table.ColumnsOfType

Table.Combine

Table.CombineColumns

Table.Contains

Table.ContainsAll

Table.ContainsAny

Table.DemoteHeaders

Table.Distinct

Table.DuplicateColumn

Table.ExpandListColumn

Table.ExpandRecordColumn

Table.ExpandTableColumn

Table.FillDown

Table.FillUp

Table.FilterWithDataTable

Table.FindText

Table.First

Table.FirstN

Table.FirstValue

Table.FromColumns

Table.FromList

Table.FromPartitions

Table.FromRecords

Table.FromRows

Table.FromValue

Table.FuzzyGroup

Table.FuzzyJoin

Table.FuzzyNestedJoin

Table.Group

Table.HasColumns

Table.InsertRows

Table.IsDistinct

Table.IsEmpty

Table.Join

Table.Keys

Table.Last

Table.LastN

Table.MatchesAllRows

Table.MatchesAnyRows

Table.Max

Table.MaxN

Table.Min

Table.MinN

Table.NestedJoin

Table.Partition

Table.PartitionValues

Table.Pivot

Table.PositionOf

Table.PositionOfAny

Table.PrefixColumns

Table.Profile

Table.PromoteHeaders

Table.Range

Table.RemoveColumns

Table.RemoveFirstN

Table.RemoveLastN

Table.RemoveMatchingRows

Table.RemoveRows

Table.RemoveRowsWithErrors

Table.RenameColumns

Table.ReorderColumns

Table.Repeat

Table.ReplaceErrorValues

Table.ReplaceKeys

Table.ReplaceMatchingRows

Table.ReplaceRelationshipIdentity

Table.ReplaceRows

Table.ReplaceValue

Table.Reverse

Table.ReverseRows

Table.RowCount

Table.Schema

Table.SelectColumns

Table.SelectRows

Table.SelectRowsWithErrors

Table.SingleRow

Table.Skip

Table.Sort

Table.Split

Table.SplitColumn

Table.ToColumns

Table.ToList

Table.ToRecords

Table.ToRows

Table.TransformColumnNames

Table.TransformColumns

Table.TransformColumnTypes

Table.TransformRows

Table.Transpose

Table.Unpivot

Table.UnpivotOtherColumns

Table.View

Table.ViewFunction

Power Query M 公式语言的快速教程

2019/12/9 •

本快速教程介绍如何创建 Power Query M 公式语言查询。

> **NOTE**
> M 是区分大小写的语言。

使用查询编辑器创建查询

若要创建高级查询，请使用“查询编辑器”。混合查询由“let”表达式封装的变量、表达式和值组成。变量可以通过使用 # 标识符来包含空格（名称在引号中），例如 #"Variable name"。

“let”表达式遵循此结构：

```
let
   Variablename = expression,
   #"Variable name" = expression2
in
   Variablename
```

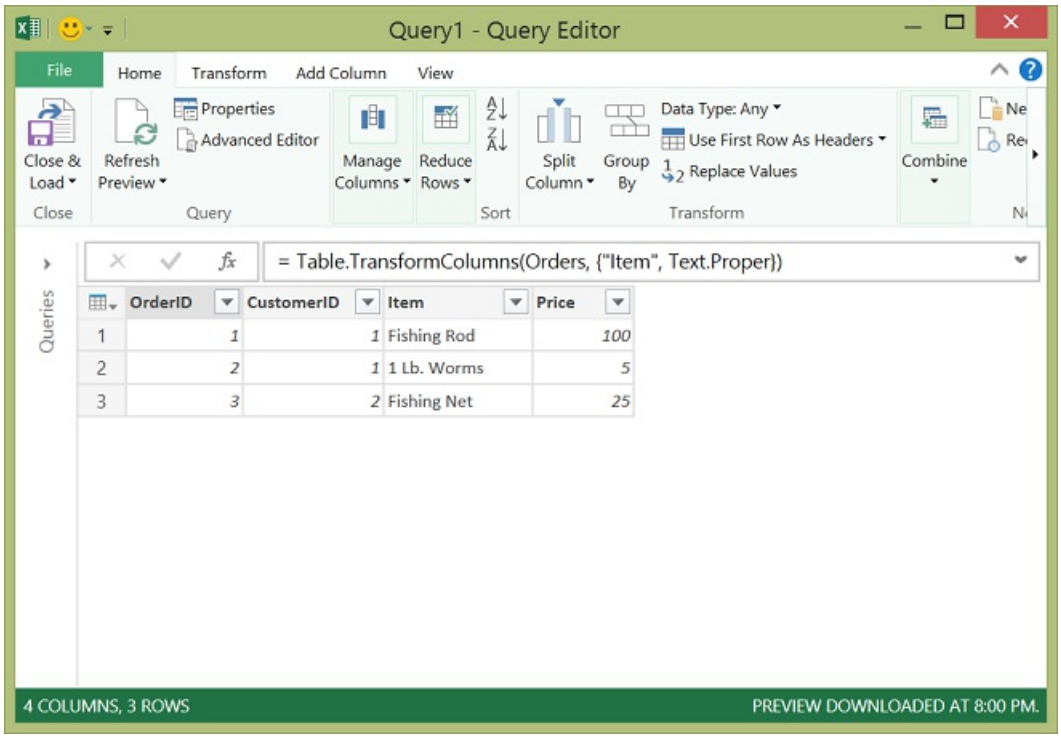
若要在“查询编辑器”中创建 M 查询，请遵循以下基本过程：

- 创建一系列以“let”语句开头的查询公式步骤。每个步骤都由步骤变量名称定义。M“变量”可以通过使用 # 字符（如 #"Step Name"）来包含空格。公式步骤可以是自定义公式。请注意，Power Query 公式语言区分大小写。
- 每个查询公式步骤都以前一个步骤为基础，通过变量名引用一个步骤。
- 使用“in”语句输出查询公式步骤。通常，将最后一个查询步骤用作 in 最终数据集结果。

若要了解有关表达式和值的详细信息，请参阅表达式、值和 let 表达式。

简单的 Power Query M 公式步骤

假设你在“查询编辑器”中创建了以下转换，以将产品名称转换为正确的大小写。



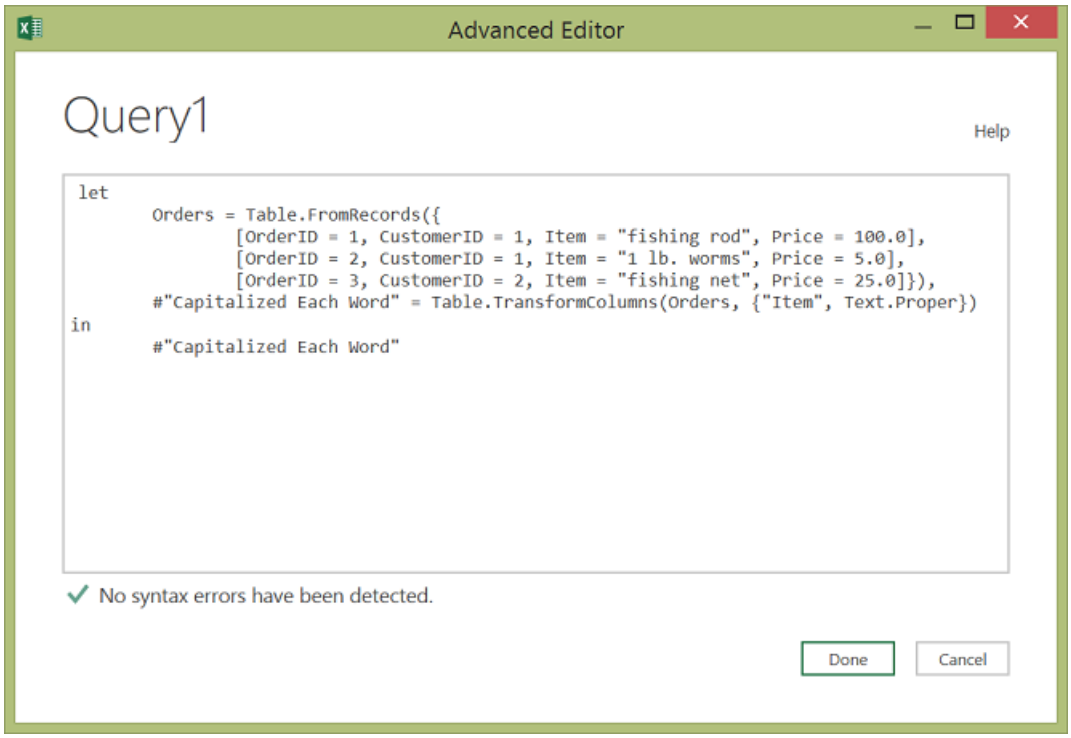
你有一张表，如下：

ORDERID	CUSTOMERID	[illegible]	[illegible]
1	1	钓鱼竿	100
2	1	1 磅蠕虫	5
3	2	渔网	25

而且，你希望将"项"列中的每个单词的首字母大写，以生成下表：

ORDERID	CUSTOMERID	[illegible]	[illegible]
1	1	钓鱼竿	100
2	1	1 磅 蠕虫	5
3	2	渔网	25

将原始表投射到结果表中的 M 公式步骤如下所示：



下面是可以粘贴到"查询编辑器"中的代码：

```
let Orders = Table.FromRecords({
    [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
    [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
    [OrderID = 3, CustomerID = 2, Item = "fishing net", Price = 25.0]}),
    #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item", Text.Proper})
in
    #"Capitalized Each Word"
```

我们来回顾一下每个公式步骤。

1. **订单** - 使用订单数据创建一个 [Table](#_Table_value)。
2. **#"每个词首字母大写"** - 要将每个单词的首字母大写，请使用 Table.TransformColumns()。
3. **in #"每个词首字母大写"** - 输出每个单词首字母都大写的表格。

另请参阅

表达式、值和 let 表达式
运算符
类型转换

Power Query M 语言规范

2020/4/26 •

该规范介绍了构成 Power Query M 语言的基本概念的值、表达式、环境和变量、标识符和计算模型。

该规范包含在以下主题中。

简介

2020/4/23 •

概述

Microsoft Power Query 提供了包含许多功能的强大"获取数据"体验。Power Query 的核心功能是筛选和合并，即从支持的数据源的一个或多个丰富集合中"混合"数据。任何此类数据混合都使用 Power Query 公式语言（通常也称为"M"）来表示。Power Query 在 Excel 和 Power BI 工作簿中嵌入 M 文档以启用数据的可重复混合。

本文档提供了 M 规范。在简要介绍了如何构建一些初步直觉并熟悉语言后，本文档将通过几个渐进式步骤精确介绍此语言：

1. 词法结构定义一组在词法上有效的文本 。
2. 此语言的基本概念的值、表达式、环境和变量、标识符和计算模型 。
3. 值（包括基元值和结构化值）的详细规范定义了语言的目标域 。
4. 值具有类型，它们本身就是一种特殊类型的值，既表示基本类型的值，又携带特定于结构化值的形状的其他元数据 。
5. M 中的一组运算符定义可以形成的表达式类型 。
6. 函数是另一种特殊值，为 M 提供丰富标准库的基础，并允许添加新抽象 。
7. 在表达式计算过程中应用运算符或函数时，可能会出现错误 。虽然错误不是值，但可以通过多种方法将错误映射回值来处理错误 。
8. Let 表达式可以引入辅助定义，以便在更小的步骤中生成复杂的表达式 。
9. If 表达式支持条件计算 。
10. 节提供简单的模块化机制 。（Power Query 尚未使用节。）
11. 最后，合并语法将此文档的所有其他节中的语法片段收集到一个完整的定义中 。

对于计算机语言理论家而言：本文档中指定的公式语言是一个大致纯粹的、高阶的、动态类型化的、部分延迟的函数语言。

表达式和值

M 中的中央构造是表达式 。可以对表达式求值（计算），然后得到单个值 。

尽管许多值都可以按字面形式写成表达式，但值不是表达式。例如，表达式 `1` 的计算结果为值 1；表达式 `1+1` 的计算结果为值 2 。这种区别很细微，但很重要。表达式是计算的方法；值是计算的结果。

下面的示例演示了 M 中可用的不同类型的值。通常，使用文本形式编写值，它们将显示在计算结果为仅该值的表达式中。（请注意，`//` 指示注释的开头，注释延续至行尾。）

- "基元"值是单个部分值，如数字、逻辑、文本或 NULL 。NULL 值可用于指示缺少数据。

```
123                 // A number
true                // A logical
"abc"               // A text
null                // null value
```

- "列表"值是值的有序序列。M 支持无限列表，但如果作为文本写入，则列表具有固定长度。大括号字符 `{` 和 `}` 表示列表的开头和结尾。

```
{123, true, "A"}     // list containing a number, a logical, and
                     //     a text
{1, 2, 3}            // list of three numbers
```

- "记录"是一组字段。字段是名称/值对，其中名称是在字段的记录中唯一的文本值。记录值的文本语法允许将名称写成不带引号的形式，这种形式也称为"标识符"。下面显示了一个记录，其中包含名为 `A`、`B` 和 `C` 的三个字段，这些字段具有值 `1`、`2` 和 `3`。

```
[
      A = 1,
      B = 2,
      C = 3
]
```

- "表"是一组按列（按名称标识）和行组织的值。没有用于创建表的文本语法，但有几个标准函数可用于从列表或记录创建表。

  例如：

```
#table( {"A", "B"}, { {1, 2}, {3, 4} } )
```

  这将创建一个形状如下所示的表：

A	B
1	2
3	4

- "函数"是一个值，当带着参数进行调用时，将生成一个新值。函数编写的方法是在括号中列出函数的"参数"，后跟"转到"符号 `=>` 和定义函数的表达式。该表达式通常引用参数（按名称）。

```
(x, y) => (x + y) / 2`
```

计算

M 语言的计算模型是根据电子表格中常见的计算模型建模的，这种模型可以基于单元格中公式之间的依赖关系来确定计算顺序。

如果你已经在 Excel 等电子表格中编写过公式，你可能会意识到左侧的公式在计算时会生成右侧的值：

	A
1	=A2 * 2
2	=A3 + 1
3	1

	A
1	4
2	2
3	1

在 M 中，表达式的各个部分可以按名称引用表达式的其他部分，并且计算过程会自动确定所引用表达式的计算顺

序。

我们可以使用一个记录来生成一个与上述电子表格示例等效的表达式。初始化字段的值时，可以通过使用字段名称引用记录中的其他字段，如下所示：

```
[
    A1 = A2 * 2,
    A2 = A3 + 1,
    A3 = 1
]
```

上述表达式等效于以下表达式（因为两者都计算出相等的值）：

```
[
    A1 = 4,
    A2 = 2,
    A3 = 1
]
```

记录可以包含在其他记录中，也可以嵌套在其他记录中 。我们可以使用“查找运算符”（`[]`）按名称访问记录的字段 。例如，以下记录具有一个名为 `Sales` 的字段（包含一个记录）和一个名为 `Total` 的字段（用于访问 `Sales` 记录的 `FirstHalf` 和 `SecondHalf` 字段）：

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

计算后，上述表达式等效于以下表达式：

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

记录也可以包含在列表中。我们可以使用“位置索引运算符”（`{}`）按其数字索引访问列表中的项目 。从列表的开头开始，使用从零开始的索引来引用列表中的值。例如，索引 `0` 和 `1` 用于引用下面列表中的第一和第二项：

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // 2500
            ]
        },
    TotalSales = Sales{0}[Total] + Sales{1}[Total] // 4600
]
```

列表和记录成员表达式（以及后面引入的 let 表达式）使用“延迟计算”进行计算，这意味着它们只会根据需要进行计

算 。所有其他表达式都使用"迫切计算"进行计算，这意味着如果在计算过程中遇到它们，则将立即对其进行计算 。考虑这一点的一种好方法是记住计算列表或记录表达式将返回一个列表或记录值，该值本身会记住在请求时（查找或索引运算符）需如何计算其列表项或记录字段。

函数

在 M 中，函数是一组输入值到单个输出值的映射 。函数的编写方法是，首先命名所需的一组输入值（函数的参数），然后提供一个表达式，该表达式将使用"转到"（`=>`）符号后面的这些输入值（函数的主体）来计算函数的结果。例如：

```
(x) => x + 1                    // function that adds one to a value
(x, y) =>  x + y                // function that adds two values
```

函数是一个值，就像数字或文本值一样。以下示例演示一个函数，该函数是随后将从其他几个字段调用或执行的 Add 字段的值 。调用函数时，将指定一组值，这些值会在逻辑上替换函数正文表达式中所需的输入值集。

```
[
    Add = (x, y) => x + y,
    OnePlusOne = Add(1, 1),     // 2
    OnePlusTwo = Add(1, 2)      // 3
]
```

库

M 包括一组通用的定义，可用于称为标准库（简称为库）的表达式 。这些定义包含一组命名值。库提供的值的名称可在表达式中使用，无需通过表达式显式定义。例如：

```
Number.E                        // Euler's number e (2.7182...)
Text.PositionOf("Hello", "ll")  // 2
```

运算符

M 包含一组可在表达式中使用的运算符。将运算符运用于操作数即形成符号表达式 。例如，在表达式 `1 + 2` 中，数字 `1` 和 `2` 是操作数，而运算符是加法运算符（`+`）。

运算符的含义可以根据操作数值的类型而变化。例如，加号运算符可用于数字以外的值类型：

```
1 + 2                   // numeric addition: 3
#time(12,23,0) + #duration(0,0,2,0)
                        // time arithmetic: #time(12,25,0)
```

另一个含义依赖于操作数的运算符示例是组合运算符（`&`）：

```
"A" & "BC"              // text concatenation: "ABC"
{1} & {2, 3}            // list concatenation: {1, 2, 3}
[ a = 1 ] & [ b = 2 ]   // record merge: [ a = 1, b = 2 ]
```

请注意，运算符不一定支持某些值的组合。例如：

```
1 + "2"  // error: adding number and text is not supported
```

如果表达式在计算时遇到未定义的运算符条件，计算结果将为错误。稍后将介绍有关 M 中的错误的详细信息。

元数据

元数据是某个与值相关联的值的相关信息。元数据被表示为一个记录值，称为元数据记录。元数据记录的字段可用于存储值的元数据。

每个值都有一个元数据记录。如果尚未指定元数据记录的值，元数据记录则为空（没有字段）。

元数据记录提供了一种以非介入式方式将附加信息与任何类型的值相关联的方法。将元数据记录与值相关联不会更改该值或其行为。

使用语法 `x meta y` 将元数据记录值 `y` 与现有的值 `x` 相关联。例如，以下将带有 `Rating` 和 `Tags` 字段的元数据记录与文本值 `"Mozart"` 相关联：

```
"Mozart" meta [ Rating = 5, Tags = {"Classical"} ]
```

对于已经包含非空元数据记录的值，应用 meta 的结果是计算现有和新的元数据记录的记录合并的结果。例如，下面两个表达式等效于彼此和前面的表达式：

```
("Mozart" meta [ Rating = 5 ]) meta [ Tags = {"Classical"} ]
"Mozart" meta ([ Rating = 5 ] & [ Tags = {"Classical"} ])
```

可以使用 Value.Metadata 函数访问一个给定值的元数据记录。在下面的示例中，`ComposerRating` 字段中的表达式访问 `Composer` 字段中值的元数据记录，然后访问元数据记录的 `Rating` 字段。

```
[
    Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],
    ComposerRating = Value.Metadata(Composer)[Rating] // 5
]
```

Let 表达式

到目前为止，很多示例都在表达式的结果中包含了表达式的所有文本值。“let”表达式允许一组值进行计算、分配名称，然后在“in”后面的后续表达式中使用。例如，在我们的销售数据示例中，可以执行以下操作：

```
let
    Sales2007 =
        [
            Year = 2007,
            FirstHalf = 1000,
            SecondHalf = 1100,
            Total = FirstHalf + SecondHalf // 2100
        ],
    Sales2008 =
        [
            Year = 2008,
            FirstHalf = 1200,
            SecondHalf = 1300,
            Total = FirstHalf + SecondHalf // 2500
        ]
  in Sales2007[Total] + Sales2008[Total] // 4600
```

上述表达式的结果是一个数字值（`4600`），该值是根据绑定到名称 `Sales2007` 和 `Sales2008` 的值计算得出的。

If 表达式

`if` 表达式根据逻辑条件在两个表达式之间进行选择。例如：

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

如果逻辑表达式 (`2 > 1`) 为 true，则选择第一个表达式 (`2 + 2`)；如果为 false，则选择第二个表达式 (`1 + 1`)。将对选定的表达式（在本例中为 `2 + 2`）进行计算，并成为 `if` 表达式 (`4`) 的结果。

Errors

错误指示计算表达式的过程不能产生值 。

错误是由运算符和函数遇到错误条件，或使用了错误表达式导致的。可以使用 try 表达式来处理错误。引发某一错误时，将指定一个值，此值可用于指示错误发生的原因。

```
let Sales =
    [
        Revenue = 2000,
        Units = 1000,
        UnitPrice = if Units = 0 then error "No Units"
                    else Revenue / Units
    ],
    UnitPrice = try Number.ToText(Sales[UnitPrice])
in "Unit Price: " &
    (if UnitPrice[HasError] then UnitPrice[Error][Message]
    else UnitPrice[Value])
```

上面的示例访问 `Sales[UnitPrice]` 字段并格式化产生结果的值：

```
"Unit Price: 2"
```

如果 `Units` 字段为零，`UnitPrice` 字段会引发错误，而 `try` 表达式则会处理此错误。结果值将为：

```
"No Units"
```

`try` 表达式将正确的值和错误转换为一个记录值，此值指示 try 表达式是否处理了错误，以及在处理错误时所提取的是正确值还是错误记录。例如，请考虑以下引发错误，然后立即进行处理的表达式：

```
try error "negative unit count"
```

此表达式计算结果为以下嵌套的记录值，解释之前单价示例中的 `[HasError]`、`[Error]` 和 `[Message]` 字段查找。

```
[
    HasError = true,
    Error =
        [
            Reason = "Expression.Error",
            Message = "negative unit count",
            Detail = null
        ]
]
```

常见的情况是使用默认值替换错误。`try` 表达式可以与一个可选的 `otherwise` 子句一起使用，从而以紧凑的形式

实现：

```
try error "negative unit count" otherwise 42
// 42
```

词法结构

2020/4/26 •

文档

M 文档是 Unicode 字符的有序序列 。M 允许在 M 文档的不同部分使用不同类别的 Unicode 字符。有关 Unicode 字符类的信息，请参阅 *Unicode 标准，版本 3.0* 中的第 4.5 节。

文档要么由一个表达式组成，要么由组织成节的多组定义构成 。第 10 章对节进行了详细说明。从概念上讲，以下步骤用于从文档中读取表达式：

1. 文档根据其字符编码方案被解码为一个 Unicode 字符序列。

2. 执行词法分析，从而将 Unicode 字符流转换为令牌流。本节余下的小节将介绍词法分析。

3. 执行词法分析，从而将令牌流转换为可计算的形式。后续部分将介绍此过程。

语法约定

词法和句法用文法产生式来表示 。每个文法产生式定义一个非终端符号，以及该非终端符号可能扩展成一系列非终端或终端符号。在文法产生式中，_非终端+ 符号以斜体显示，终端符号以固定宽度字体显示 。

文法产生式的第一行是定义的非终端符号的名称，后跟冒号。每一个后续缩进行都包含一个非终端符的可能扩展，该非终端符由一系列非终端符或终端符符号组成。例如，产生式：

if-expression:
`if` *if-condition* `then` *true-expression* `else` *false-expression*

定义一个 if-expression 由令牌 `if` 后跟 if-condition，令牌 `then` 后跟 true-expression 以及令牌 `else` 后跟 false-expression 组成 。

当非终端符号有多个可能的扩展时，替代项在单独的行中列出。例如，产生式：

variable-list:
variable
variable-list `,` *variable*

定义一个 variable-list 由一个变量或 variable-list 后跟 variable 组成 。换言之，这个定义是递归的，它指定变量列表由一个或多个(用逗号分隔的)变量组成。

下标后缀"$_{opt}$"用于指示可选符号。产生式：

field-specification:
`optional`$_{opt}$ *identifier* `=` *field-type*

为以下的简写：

field-specification:
identifier `=` *field-type*
`optional` *identifier* `=` *field-type*

并定义了 field-specification 可选地以终端符号 `optional` 开头，后跟标识符、终端符号 `=` 和字段类型 。

替代项通常在单独的行中列出，但在有许多替代项的情况下，"one of"一词可能出现在单行上给出的一列扩展之前。这只是对在单独的行中列出每个替代项的简化。例如，产生式：

decimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`

为以下的简写：

decimal-digit:

`0`
`1`
`2`
`3`
`4`
`5`
`6`
`7`
`8`
`9`

词法分析

lexical-unit 产生式定义了 M 文档的词法语法 。每个有效的 M 文档都遵循此语法。

lexical-unit:
    lexical-elements$_{opt}$
lexical-elements:
    lexical-element
    lexical-element
    lexical-elements
lexical-element:
    whitespace
    token comment

在词法级别，M 文档由一系列空白、注释和令牌元素组成 。以下各节将介绍这些产生式。在语法中，只有令牌元素是有意义的 。

空格

空格用于分隔 M 文档中的注释和令牌。空格包括空格字符（它是 Unicode 类 Zs 的一部分），以及水平和垂直制表符、换页符和换行符序列。换行字符序列包括回车符、换行符、后跟换行符的回车符、下一行和段落分隔符。

空格:
    带有 Unicode 类 Zs 的任何字符
    水平制表符字符 (`U+0009`)
    垂直制表符 (`U+000B`)
    换页符 (`U+000C`)
    后跟换行符 (`U+000A`) 的回车符 (`U+000D`)
    new-line-character
new-line-character:
    回车符 (`U+000D`)
    换行符 (`U+000A`)
    换行符 (`U+0085`)
    行分隔符 (`U+2028`)
    段落分隔符 (`U+2029`)

为了与添加 end-of-file 标记的源代码编辑工具兼容，并使文档能够被看作正确终止的行序列，将按顺序对 M 文档

应用以下转换：

- 如果文档的最后一个字符是 Control-Z 字符（`U+001A`），则删除此字符。
- 如果文档非空并且文档的最后一个字符不是回车符（`U+000D`）、换行符（`U+000A`）、行分隔符（`U+2028`）或段落分隔符（`U+2029`），则在文档末尾添加回车符（`U+000D`）。

注释

支持两种形式的注释：单行注释和分隔注释。S 注释以字符 `//` 开头，并扩展到源行的末尾 。分隔注释以字符 `/*` 开头，以字符 `*/` 结尾 。

分隔注释可能跨多行。

comment:
    single-line-comment
    delimited-comment
single-line-comment:
    `//` *single-line-comment-characters*$_{opt}$
single-line-comment-characters:
    single-line-comment-character single-line-comment-characters$_{opt}$
single-line-comment-character:
    除 new-line-character
delimited-comment 之外的任何 Unicode 字符：
    `/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`
delimited-comment-text:
    delimited-comment-section delimited-comment-text$_{opt}$
delimited-comment-section:
    `/`
    asterisks$_{opt}$ *not-slash-or-asterisk*
asterisks:
    `*` *asterisks*$_{opt}$
not-slash-or-asterisk:
    除 `*` 或 `/` 之外的任何 Unicode 字符

注释不嵌套。字符序列 `/*` 和 `*/` 在单行注释中没有特殊含义，字符序列 `//` 和 `/*` 在分隔注释中没有特殊含义。

不在文字串中处理注释。示例

```
/* Hello, world
*/
    "Hello, world"
```

包含分隔注释。

示例

```
// Hello, world
//
"Hello, world" // This is an example of a text literal
```

显示若干单行注释。

令牌

令牌是标识符、关键字、文字、运算符或标点符号 。空白和注释用于分隔标记，但不会将其视为令牌。

token:
    identifier
    keyword
    literal
    operator-or-punctuator

字符转义序列

M 文本值可以包含任意 Unicode 字符。然而，文字文本仅限于图形字符，需要对非图形字符使用转义序列 。例如，要在文字文本中包含回车符、换行符或制表符，可以分别使用 `#(cr)`、`#(lf)` 和 `#(tab)` 转义序列。若要在文字文本中嵌入转义序列开始字符 `#(`，`#` 本身需要进行转义：

```
#(#)(
```

转义序列也可以包含短（四个十六进制数字）或长（八个十六进制数字）Unicode 码位值。因此，以下三个转义序列是等效的：

```
#(000D)     // short Unicode hexadecimal value
#(0000000D) // long Unicode hexadecimal value
#(cr)       // compact escape shorthand for carriage return
```

单个转义序列中可以包含多个转义码，用逗号分隔；因此，以下两个序列是等效的：

```
#(cr,lf)
#(cr)#(lf)
```

下面介绍 M 文档中字符转义的标准机制。

character-escape-sequence:
    `#(` *escape-sequence-list* `)`
escape-sequence-list:
    single-escape-sequence
    single-escape-sequence `,` *escape-sequence-list*
single-escape-sequence:
    long-unicode-escape-sequence
    short-unicode-escape-sequence
    control-character-escape-sequence
    escape-escape
long-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
short-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit
control-character-escape-sequence:
    control-character
control-character:
    `cr`
    `lf`
    `tab`
escape-escape:
    `#`

文字

文本是值的源代码表示形式 。

literal:
    logical-literal
    number-literal
    text-literal
    null-literal

null 文本

null 文本用于写入 `null` 值。`null` 值表示不存在的值。

null-literal:
    `null`

逻辑文本

逻辑文本用于写入值 `true` 和 `false`，并生成逻辑值。

logical-literal:
    `true`
    `false`

数字文本

数字文字用于写入数值并生成数值。

number-literal:
    decimal-number-literal
    hexadecimal-number-literal
decimal-number-literal:
    decimal-digits `.` *decimal-digits* *exponent-part*$_{opt}$
    `.` *decimal-digits* *exponent-part*$_{opt}$
    decimal-digits *exponent-part*$_{opt}$
decimal-digits:
    decimal-digit *decimal-digits*$_{opt}$
decimal-digit: 以下之一
    `0 1 2 3 4 5 6 7 8 9`
exponent-part:
    `e` *sign*$_{opt}$ *decimal-digits*
    `E` *sign*$_{opt}$ *decimal-digits*
sign: one of
    `+ -`
hexadecimal-number-literal:
    `0x` *hex-digits*
    `0X` *hex-digits*
hex-digits:
    hex-digit *hex-digits*$_{opt}$
hex-digit: 以下之一
    `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

通过在十六进制数字前面加上字符 `0x`，可以十六进制格式指定一个数字 。例如：

```
0xff // 255
```

请注意，如果数字文本中包含小数点，则它后面必须至少有一个数字。例如，`1.3` 是数字文本，但 `1.` 和 `1.e3` 不是。

文本文字

文本文字用于写入 Unicode 字符序列并生成文本值。

text-literal:
" *text-literal-characters*$_{opt}$ "
text-literal-characters:
text-literal-character text-literal-characters$_{opt}$
text-literal-character:
single-text-character
character-escape-sequence
double-quote-escape-sequence
single-text-character:
除后跟 ((U+0028) 的 " (U+0022) 或 # (U+0023) 外的任何字符
double-quote-escape-sequence:
"" (U+0022 , U+0022)

若要在文本值中包含引号，请重复使用引号，如下所示：

```
"The ""quoted"" text" // The "quoted" text
```

可使用 *character-escape-sequence* 产生式在文本值中写入字符，而无需在文档中将它们直接编码为 Unicode 字符。 例如，回车和换行可以用文本值写入：

```
"Hello world#(cr,lf)"
```

标识符

标识符是用于引用值的名称 。标识符可以是常规标识符，也可以是带引号的标识符。

identifier:
regular-identifier
quoted-identifier
regular-identifier:
available-identifier
available-identifier dot-character regular-identifier
available-identifier:
A *keyword-or-identifier* that is not a *keyword*
keyword-or-identifier:
identifier-start-character identifier-part-characters$_{opt}$
identifier-start-character:
letter-character
underscore-character
identifier-part-characters:
identifier-part-character identifier-part-characters$_{opt}$
identifier-part-character:
letter-character
decimal-digit-character
underscore-character
connecting-character
combining-character
formatting-character
dot-character:
. (U+002E)

underscore-character:
`_` (`U+005F`)
letter-character:
Lu、Ll、Lt、Lm、Lo 或 Nl 类的 Unicode 字符
combining-character:
Mn 或 Mc 类的 Unicode 字符
decimal-digit-character:
Nd 类的 Unicode 字符
connecting-character:
Pc 类的 Unicode 字符
formatting-character:
Cf 类的 Unicode 字符

带引号的标识符可用于允许零个或多个 Unicode 字符的任何序列用作标识符，包括关键字、空格、注释、运算符和标点符号 。

quoted-identifier:
`#"` *text-literal-characters*$_{opt}$ `"`

注意，转义序列和用于转义引号的双引号可以在带引号的标识符中使用，就像在 text-literal 中一样 。

以下示例对包含空格字符的名称使用标识符引号：

```
[
    #"1998 Sales" = 1000,
    #"1999 Sales" = 1100,
    #"Total Sales" = #"1998 Sales" + #"1999 Sales"
]
```

以下示例使用标识符引号将 `+` 运算符包含在标识符中：

```
[
    #"A + B" = A + B,
    A = 1,
    B = 2
]
```

通用化标识符
在 M 中有两个地方没有由包含空格或关键字或数字文字的标识符引起的歧义。这些位置是记录文本中记录字段的名称，在字段访问运算符 (`[ ]`) 中，M 允许这样的标识符，而不必使用带引号的标识符。

```
[
    Data = [ Base Line = 100, Rate = 1.8 ],
    Progression = Data[Base Line] * Data[Rate]
]
```

用于命名和访问字段的标识符称为通用标识符，定义如下 ：

generalized-identifier :
generalized-identifier-part
generalized-identifier 仅用空格分隔 (`U+0020`)
generalized-identifier-part
generalized-identifier-part:
generalized-identifier-segment
decimal-digit-character generalized-identifier-segment
generalized-identifier-segment:

keyword-or-identifier
keyword-or-identifier dot-character keyword-or-identifier

关键字

_关键字_是保留的类似标识符的字符序列，不能用作标识符，除非使用标识引用机制或允许使用通用标识符。

keyword: one of `and as each else error false if in is let meta not otherwise or`
`section shared then true try type #binary #date #datetime`
`#datetimezone #duration #infinity #nan #sections #shared #table #time`

运算符和标点符号

有多种运算符和标点符号。表达式中使用运算符来描述涉及一个或多个操作数的操作。例如，表达式 `a + b` 使用 `+` 运算符添加两个操作数 `a` 和 `b`。标点符号用于分组和分隔。

operator-or-punctuator: one of
`, ; = < <= > >= <> + - * / & ( ) [ ] { } @ ! ? => .. ...`

基本概念

2020/4/23 •

本节讨论在后续节中显示的基本概念。

值

单个数据称为_值_。广义上讲，有两个常规类别的值：*基元值* 和*结构化值*。前者是值的最基本形式，后者由基元值和其他结构化值构成。例如，值 \

```
1
true
3.14159
"abc"
```

是基元，因为它们不由其他值构成。但是，值

```
{1, 2, 3}
[ A = {1}, B = {2}, C = {3} ]
```

是使用基元值进行构造的，在这条记录中，是使用其他结构化值构造的。

表达式

_表达式_是用于构造值的公式。表达式可以使用多种语法结构形成。下面是一些表达式示例。每一行都是一个单独的表达式。

```
"Hello World"             // a text value
123                       // a number
1 + 2                     // sum of two numbers
{1, 2, 3}                 // a list of three numbers
[ x = 1, y = 2 + 3 ]      // a record containing two fields:
                          //        x and y
(x, y) => x + y           // a function that computes a sum
if 2 > 1 then 2 else 1    // a conditional expression
let x = 1 + 1  in x * 2   // a let expression
error "A"                 // error with message "A"
```

如上所示，最简单的表达式形式是表示值的文本。

更复杂的表达式由其他表达式（称为 *sub-expressions*）组成。例如：

```
1 + 2
```

上述表达式实际上由三个表达式组成。`1` 和 `2` 文本是父级表达式 `1 + 2` 的子表达式。

执行表达式中使用的语法结构所定义的算法称为_计算_表达式。每种类型的表达式都具有其计算规则。例如，文本表达式（如 `1`）将生成一个常数值，而表达式 `a + b` 将通过计算其他两个表达式（`a` 和 `b`）来获取生成的值，并根据一组规则将它们相加。

环境和变量

表达式在指定环境中进行计算。*环境_是一组名为_变量_命名值。*环境中的每个变量在环境中都有一个唯一名称，称为_标识符。

在_全局环境_中计算顶级(或_根_)表达式。全局环境由表达式计算器提供，而不是根据要计算的表达式的内容来确定。全局环境的内容包括标准库定义，并且可能受到从某些文档集的节导出的影响。(为简单起见，本节的示例将假定一个空的全局环境。也就是说，假设没有标准库，也没有任何其他基于节的定义。)

用于计算子表达式的环境由父表达式确定。大多数父表达式类型将在其计算所用的环境中计算子表达式，而有些则使用不同的环境。全局环境是在其中计算全局表达式的_父环境_。

例如，*record-initializer-expression* 使用修改后的环境计算每个字段的子表达式。修改后的环境包含记录的每个字段的变量，但正在初始化的字段除外。如果包含记录的其他字段，则该字段取决于字段的值。例如：

```
[
    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
]
```

同样，*let-expression* 使用包含 let 的每个变量(正在初始化的变量除外)的环境来计算每个变量的子表达式。*let-expression* 使用包含所有变量的环境来计算后面的表达式：

```
let

    x = 1,          // environment: y, z
    y = 2,          // environment: x, z
    z = x + y       // environment: x, y
in
    x + y + z       // environment: x, y, z
```

(结果是，*record-initializer-expression* 和 *let-expression* 实际定义_两个_环境，其中一种环境确实包含正在初始化的变量。这有助于高级递归定义，会在标识符引用中进行介绍。

为了形成子表达式的环境，新变量会与父环境中的变量进行“合并”。以下示例演示了嵌套记录的环境：

```
[
    a =
    [

        x = 1,      // environment: b, y, z
        y = 2,      // environment: b, x, z
        z = x + y   // environment: b, x, y
    ],
    b = 3           // environment: a
]
```

以下示例显示了嵌套在 let 中的记录的环境：

```
Let
    a =
    [
        x = 1,       // environment: b, y, z
        y = 2,       // environment: b, x, z
        z = x + y    // environment: b, x, y
    ],
    b = 3            // environment: a
in
    a[z] + b         // environment: a, b
```

合并变量与环境可能会导致变量之间产生冲突(因为环境中的每个变量必须具有唯一的名称)。解决冲突的方法如下:如果要合并的新变量的名称与父环境中的现有变量的名称相同,则新环境中将优先使用新变量。在下面的示例中,内部(更深层嵌套的)变量 `x` 将优先于外部变量 `x`。

```
[
    a =
    [
        x = 1,       // environment: b, x (outer), y, z
        y = 2,       // environment: b, x (inner), z
        z = x + y    // environment: b, x (inner), y
    ],
    b = 3,           // environment: a, x (outer)
    x = 4            // environment: a, b
]
```

标识符引用

identifier-reference 用于引用环境中的变量。

identifier-expression:
    identifier-reference
identifier-reference:
    exclusive-identifier-reference
    inclusive-identifier-reference

最简单的标识符引用形式是 *exclusive-identifier-reference*:

exclusive-identifier-reference:
    identifier

xclusive-identifier-reference 引用的变量不属于该标识符所在的表达式环境的一部分,或者引用当前正在初始化的标识符是错误的。

inclusive-identifier-reference 可用于获取访问包含正在初始化的标识符的环境的权限。如果在没有初始化标识符的上下文中使用此方法,则它相当于 *exclusive-identifier-reference*。

inclusive-identifier-reference:
    `@` *identifier*

这有助于定义递归函数,因为函数名称通常不在范围内。

```
[
    Factorial = (n) =>
        if n <= 1 then
            1
        else
            n * @Factorial(n - 1),  // @ is scoping operator

    x = Factorial(5)
]
```

与 *record-initializer-expression* 一样,*inclusive-identifier-reference* 可用于 *let-expression*,访问包含正在初始化的标识符的环境。

评估顺序

请考虑以下初始化记录的表达式:

```
[
    C = A + B,
    A = 1 + 1,
    B = 2 + 2
]
```

计算时，此表达式会产生以下记录值：

```
[
    C = 6,
    A = 2,
    B = 4
]
```

表达式表明，若要为字段 `C` 执行 `A + B` 计算，则必须知道字段 `A` 和字段 `B` 的值。该示例是计算的_依赖项排序_，由表达式提供。M 计算器遵循表达式提供的依赖项顺序，但可以按其选择的任何顺序自由执行其余计算。例如，计算顺序可以是：

```
A = 1 + 1
B = 2 + 2
C = A + B
```

也可以是：

```
B = 2 + 2
A = 1 + 1
C = A + B
```

或者，由于 `A` 和 `B` 并不相互依赖，因此可以同时计算：

`B = 2 + 2` *concurrently with* `A = 1 + 1`
`C = A + B`

副作用

如果表达式没有声明显式依赖项，则允许表达式计算器自动计算计算顺序，这是一个简单而强大的计算模型。

但它确实依赖于能够重新排列计算。由于表达式可以调用函数，并且这些函数可以通过发出外部查询来观察表达式的外部状态，因此可以构造一个场景，其中计算顺序确实很重要，但不会在表达式的部分顺序中捕获。例如，函数可以读取文件的内容。如果重复调用该函数，则可以观察到对该文件的外部更改，因此，重新排序可能会导致程序行为出现明显差异。根据此类观察到的计算顺序，M 表达式的正确性导致对特定的实现选择产生依赖，这些选择可能因不同的计算器而有所不同，甚至在不同的情况下也可能会有所不同。

不可变性

计算某个值后，它就是_不可变的_，这意味着无法再对其进行更改。这简化了计算表达式的模型，推理结果也更容易，因为一旦用该值来计算表达式的后续部分，则再无法更改该值。例如，仅当需要时才计算记录字段。但是，一旦计算后，它在记录的生命周期内保持不变。即使尝试计算字段时引发了错误，也会在每次尝试访问该记录字段时再次引发同样的错误。

immutable-once-calculated 规则的一种重要的例外情况应用于列值和表值。两者都具有_流式语义_。也就是说，重复枚举列表中的项或表中的行可以产生不同的结果。流式语义使用 M 表达式结构，这些表达式可以转换无法同时存在于内存的数据集。

另外，请注意，函数应用程序与值构造_不同_。库函数可能会公开外部状态（如当前时间或查询数据库的结果随时间

而变化），从而呈现_不确定性_。虽然 M 中定义的函数不会公开任何此类不确定性行为，但如果将其定义为调用其他非确定性函数，则其可以公开。

M 中不确定性的最后一个来源是_错误_。错误发生时将停止计算（直到达到由 try 表达式处理的级别为止）。通常无法观察到 `a + b` 是在 `b` 之前计算 `a`，还是在 `a` 之前计算 `b`（为了简单，可以忽略并发性）。但是，如果首先计算的子表达式引发错误，则可以确定两个表达式中哪个是首先计算的。

值

2020/4/26 •

值是通过计算表达式生成的数据。本部分介绍了 M 语言的值种类。每种类型的值都与一种文字语法、一组该类型的值、一组在该组值上定义的运算符以及一种归属于新构造的值的内部类型相关联。

Null	`null`
逻辑	`true` `false`
数字	`0` `1` `-1` `1.5` `2.3e-5`
时间	`#time(09,15,00)`
Date	`#date(2013,02,26)`
DateTime	`#datetime(2013,02,26, 09,15,00)`
DateTimeZone	`#datetimezone(2013,02,26, 09,15,00, 09,00)`
Duration	`#duration(0,1,30,0)`
文本	`"hello"`
二进制	`#binary("AQID")`
列表	`{1, 2, 3}`
记录	`[ A = 1, B = 2 ]`
表格	`#table({"X","Y"},{{0,1},{1,0}})`
Function	`(x) => x + 1`
类型	`type { number }` `type table [ A = any, B = text ]`

以下各节详细介绍了每种值种类。类型和类型归属在类型中正式定义。函数值在函数中定义。以下各节列出了为每种值类型定义的运算符，并提供了示例。运算符语义的完整定义如运算符中所示。

Null

_null 值_用于表示缺失值，或者值的状态不确定或未知。使用文本 `null` 写入 null 值。为 null 值定义了以下运算符：

“”	“”
`x > y`	大于
`x >= y`	大于或等于
`x < y`	小于
`x <= y`	小于或等于
`x = y`	等于
`x <> y`	不等于

`null` 值的本机类型是内部类型 `null`。

逻辑

逻辑值用于布尔运算，值为 true 或 false 。使用文本 `true` 和 `false` 写入逻辑值。为逻辑值定义了以下运算符：

“”	“”
`x > y`	大于
`x >= y`	大于或等于
`x < y`	小于
`x <= y`	小于或等于
`x = y`	等于
`x <> y`	不等于
`x or y`	条件逻辑或
`x and y`	条件逻辑和
`not x`	逻辑非

逻辑值（`true` 和 `false`）的本机类型是内部类型 `logical`。

数字

数值用于数字和算术运算 。下面是数字文本的示例：

```
3.14  // Fractional number
-1.5  // Fractional number
1.0e3 // Fractional number with exponent
123   // Whole number
1e3   // Whole number with exponent
0xff  // Whole number in hex (255)
```

数字至少以双精度表示(但可以保留更高的精度)。双精度表示形式与 [IEEE 754-2008] 中定义的二进制浮点运算的 IEEE 64 位双精度标准一致 。(双精度表示形式的动态范围约为 5.0 x 10^{324} 到 1.7 x 10^{308}，精度为 15-16 位。)

以下特殊值也被视为数值：

- 正零和负零。在大多数情况下，正零和负零的行为与简单值零相同，但是某些运算区分这两个值。
- 正无穷 (`#infinity`) 和负无穷 (`-#infinity`)。无穷大是由非零数除以零等运算产生的。例如，`1.0 / 0.0` 产生正无穷，`-1.0 / 0.0` 产生负无穷。
- _非数字_值 (`#nan`)，通常缩写为 NaN。NaN 是由无效的浮点运算生成的，例如用零除以零。

使用精度执行二进制数学运算 。精度决定将操作数舍入到的域和执行操作的域。如果没有显式指定的精度，则使用双精度执行此类操作 。

- 如果数学运算的结果对于目标格式来说太小，则运算结果将变为正零或负零。
- 如果数学运算的结果对于目标格式来说太大，则运算的结果将变为正无穷或负无穷。
- 如果数学运算无效，则运算结果变为 NaN。
- 如果浮点运算的一个或两个操作数都是 NaN，则运算结果变为 NaN。

为数值定义了以下运算符：

IIII	II
`x > y`	大于
`x >= y`	大于或等于
`x < y`	小于
`x <= y`	小于或等于
`x = y`	等于
`x <> y`	不等于
`x + y`	求和
`x - y`	差
`x * y`	产品
`x / y`	商
`+x`	一元加

[illegible]	[illegible]
`-x`	否定

数值的本机类型是内部类型 `number`。

时间

时间值存储一天中时间的不透明表示形式。时间编码为自午夜起的时钟周期数，即 24 小时制中已经过去的 100 纳秒的数量。自午夜起的最大时钟周期数对应 23:59:59.9999999 这一时刻。

时间值可以使用 #time 内部函数来构造。

```
#time(hour, minute, second)
```

以下内容必须保持不变，否则将引发原因代码为 `Expression.Error` 的错误：

0 ≤ hour ≤ 24
0 ≤ minute ≤ 59
0 ≤ second ≤ 59

此外，如果 hour = 24，则 minute 和 second 必须为零。

为时间值定义了以下运算符：

[illegible]	[illegible]
`x = y`	等于
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于

以下运算符允许其一个或两个操作数为日期：

[illegible]	[illegible]	[illegible]	[illegible]
`x + y`	`time`	`duration`	持续时间的日期偏移
`x + y`	`duration`	`time`	持续时间的日期偏移
`x - y`	`time`	`duration`	求反持续时间的日期偏移
`x - y`	`time`	`time`	日期之间的持续时间

[illegible]	[illegible]	[illegible]	[illegible]
`x & y`	`date`	`time`	合并的 DateTime

时间值的本机类型是内部类型 `time`。

日期

日期值存储特定日期的不透明表示形式。日期编码为有史以来经过的天数，从公历 0001 年 1 月 1 日开始。有史以来的最大天数为 3652058，对应 9999 年 12 月 31 日。

日期值可以使用 `#date` 内部函数来构造。

```
#date(year, month, day)
```

以下内容必须保持不变，否则将引发原因代码为 `Expression.Error` 的错误：

1 ≤ year ≤ 9999
1 ≤ month ≤ 12
1 ≤ day ≤ 31

此外，日期必须对选定的月份和年份有效。

为日期值定义了以下运算符：

[illegible]	[illegible]
`x = y`	等于
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于

以下运算符允许其一个或两个操作数为日期：

[illegible]	[illegible]	[illegible]	[illegible]
`x + y`	`date`	`duration`	持续时间的日期偏移
`x + y`	`duration`	`date`	持续时间的日期偏移
`x - y`	`date`	`duration`	求反持续时间的日期偏移
`x - y`	`date`	`date`	日期之间的持续时间

[illegible]	[illegible]	[illegible]	[illegible]
`x & y`	`date`	`time`	合并的 DateTime

日期值的本机类型是内部类型 `date`。

DateTime

日期时间值同时包含日期和时间 。

日期时间值可以使用 `#datetime` 内部函数来构造。

```
#datetime(year, month, day, hour, minute, second)
```

以下内容必须保持不变，否则将引发原因代码为 Expression.Error 的错误：1 ≤ year ≤ 9999
1 ≤ month ≤ 12
1 ≤ day ≤ 31
0 ≤ hour ≤ 23
0 ≤ minute ≤ 59
0 ≤ second ≤ 59

此外，日期必须对选定的月份和年份有效。

为日期时间值定义了以下运算符：

[illegible]	[illegible]
`x = y`	等于
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于

以下运算符允许其一个或两个操作数为日期时间：

[illegible]	[illegible]	[illegible]	[illegible]
`x + y`	`datetime`	`duration`	持续时间的日期时间偏移
`x + y`	`duration`	`datetime`	持续时间的日期时间偏移
`x - y`	`datetime`	`duration`	求反持续时间的日期时间偏移

[illegible]	[illegible]	[illegible]	[illegible]
`x - y`	`datetime`	`datetime`	日期时间之间的持续时间

日期时间值的本机类型是内部类型 `datetime`。

时区

时区值包含日期时间和时区。时区编码为从 UTC 偏移的分钟数，即日期时间的时间部分从全球协调时 (UTC) 偏移的分钟数。UTC 偏移分钟数的最小值为 -840，表示 UTC 偏移量 -14:00，或者说比 UTC 早 14 小时。UTC 偏移分钟数的最大值为 840，对应于 UTC 偏移量 14:00。

时区值可以使用 `#datetimezone` 内部函数来构造。

```
#datetimezone(
       year, month, day,
       hour, minute, second,
       offset-hours, offset-minutes)
```

以下内容必须保持不变，否则将引发原因代码为 `Expression.Error` 的错误：

1 ≤ year ≤ 9999
1 ≤ month ≤ 12
1 ≤ day ≤ 31
0 ≤ hour ≤ 23
0 ≤ minute ≤ 59
0 ≤ second ≤ 59
-14 ≤ offset-hours ≤ 14
-59 ≤ offset-minutes ≤ 59

此外，日期必须对所选月份和年份有效，如果偏移小时数 = 14，则偏移分钟数<= 0，如果偏移小时数 = -14，则偏移分钟数 >= 0。

为时区值定义了以下运算符：

[illegible]	[illegible]
`x = y`	等于
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于

以下运算符允许其一个或两个操作数为时区：

[illegible]	[illegible]	[illegible]	[illegible]
`x + y`	`datetimezone`	`duration`	持续时间的 Datetimezone 偏移
`x + y`	`duration`	`datetimezone`	持续时间的 Datetimezone 偏移
`x - y`	`datetimezone`	`duration`	求反持续时间的 Datetimezone 偏移
`x - y`	`datetimezone`	`datetimezone`	datetimezone 之间的持续时间

时区值的本机类型是内部类型 `datetimezone`。

持续时间

持续时间值存储以 100 纳秒为单位的时间轴上两点之间的距离的不透明表示。持续时间的大小可以是正的，也可以是负的，正值表示时间向前推进，负值表示时间向后推进。在持续时间中可以存储的最小值是 -9,223,372,036,854,775,808 个时钟周期，即往前 10,675,199 天 2 小时 48 分 05.4775808 秒。在持续时间中可以存储的最大值是 9,223,372,036,854,775,807 个时钟周期，即往后 10,675,199 天 2 小时 48 分 05.4775807 秒。

可以使用 `#duration` 内部函数构造持续时间值：

```
#duration(0, 0, 0, 5.5)          // 5.5 seconds
#duration(0, 0, 0, -5.5)         // -5.5 seconds
#duration(0, 0, 5, 30)           // 5.5 minutes
#duration(0, 0, 5, -30)          // 4.5 minutes
#duration(0, 24, 0, 0)           // 1 day
#duration(1, 0, 0, 0)            // 1 day
```

为持续时间值定义了以下运算符：

[illegible]	[illegible]
`x = y`	等于
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于

此外，以下运算符允许其一个或两个操作数为持续时间值：

ㄱㄱㄱ	ㄱㄱㄱㄱ	ㄱㄱㄱㄱ	ㄱㄱ
`x + y`	`datetime`	`duration`	持续时间的日期时间偏移
`x + y`	`duration`	`datetime`	持续时间的日期时间偏移
`x + y`	`duration`	`duration`	持续时间之和
`x - y`	`datetime`	`duration`	求反持续时间的日期时间偏移
`x - y`	`datetime`	`datetime`	日期时间之间的持续时间
`x - y`	`duration`	`duration`	持续时间之差
`x * y`	`duration`	`number`	持续时间的 N 倍
`x * y`	`number`	`duration`	持续时间的 N 倍
`x / y`	`duration`	`number`	持续时间的分数

持续时间值的本机类型是内部类型 `duration` 。

文本

文本值表示 Unicode 字符序列 。文本值具有符合以下语法的文本形式：

_text-literal：
    `"` text-literal-characters$_{opt}$ `"`
text-literal-characters：
    text-literal-character text-literal-characters$_{opt}$
text-literal-character：
    single-text-character
    character-escape-sequence
    double-quote-escape-sequence
single-text-character：
    除后跟 `(` (`U+0028`) 的 `"` (`U+0022`) 或 `#` (`U+0023`) 外的任何字符
double-quote-escape-sequence:
    `""` (`U+0022` , `U+0022`)

以下是文本值的示例：

```
"ABC" // the text value ABC
```

为文本值定义了以下运算符：

ㄱㄱㄱ	ㄱㄱ
`x = y`	等于

[illegible]	[illegible]
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于
`x & y`	串联

文本值的本机类型是内部类型 `text`。

二进制

二进制值表示字节序列。没有文字格式。提供了几个标准库函数来构造二进制值。例如，`#binary` 可用于从字节列表构造二进制值：

```
#binary( {0x00, 0x01, 0x02, 0x03} )
```

在二进制值上定义了以下运算符：

[illegible]	[illegible]
`x = y`	等于
`x <> y`	不等于
`x >= y`	大于或等于
`x > y`	大于
`x < y`	小于
`x <= y`	小于或等于

二进制值的本机类型是内部类型二进制。

列表

列表值是在枚举时生成一系列值的值。列表生成的值可以包含任何类型的值，包括列表。可以使用初始化语法构造列表，如下所示：

list-expression:
　　{ *item-list*$_{opt}$ }
item-list:

item
item `,` *item-list*
item:
expression
expression `..` *expression*

下面是一个列表表达式的示例，它定义了包含三个文本值的列表：`"A"`、`"B"` 和 `"C"` 。

```
{"A", "B", "C"}
```

值 `"A"` 是列表中的第一项，值 `"C"` 是列表中的最后一项。

- 在访问列表项之前，不会对其计算。
- 虽然使用 list 语法构造的列表值将按其在 item-list 中出现的顺序生成项，但通常，从库函数返回的列表在每次枚举时可能生成不同的集合或不同数量的值 。

若要在列表中包含整数序列，可以使用 `a..b` 形式：

```
{ 1, 5..9, 11 }     // { 1, 5, 6, 7, 8, 9, 11 }
```

列表中的项数，即列表计数，可以使用 `List.Count` 函数来确定 。

```
List.Count({true, false})  // 2
List.Count({})             // 0
```

列表可以有效地包含无限个项;这样的列表的 `List.Count` 未定义，可能引发错误或无法终止。

如果列表不包含任何项，则将其称为空列表 。空列表写为：

```
{}  // empty list
```

为列表定义了以下运算符：

`x = y`	等于
`x <> y`	不等于
`x & y`	Concatenate

例如：

```
{1, 2} & {3, 4, 5}   // {1, 2, 3, 4, 5}
{1, 2} = {1, 2}      // true
{2, 1} <> {1, 2}     // true
```

列表值的本机类型是内部类型 `list`，它指定项类型 `any` 。

记录

记录值是字段的有序序列 。字段由字段名和字段值组成 。字段名是在记录中定义字段的独特文本值。字段值可以是任何类型的值，包括记录。可以使用初始化语法构造记录，如下所示：

record-expression:
    `[` *field-list*$_{opt}$ `]`
field-list:
    field
    field `,` *field-list*
field:
    field-name `=` *expression*
field-name:
    generalized-identifier

下面的示例使用名为 `x` 、值为 `1` 的字段和名为 `y` 、值为 `2` 的字段构造记录。

```
[ x = 1, y = 2 ]
```

下面的示例构造了一条记录，其中包含一个名为 a、值为嵌套记录的 `a` 字段。嵌套记录包含名为 `b` 、值为 `2` 的字段。

```
[ a = [ b = 2 ] ]
```

在计算记录表达式时，以下条件适用：

- 分配给每个字段名的表达式用于确定关联字段的值。
- 如果分配给字段名的表达式在计算时生成一个值，则该值将成为结果记录的字段值。
- 如果分配给字段名的表达式在计算时引发错误，将记录引发了错误这一事实，还将记录该字段以及引发的错误值。对该字段的后续访问将导致重新引发具有记录的错误值的错误。
- 表达式是在类似父环境的环境中计算的，只合并了与记录的每个字段（正在初始化的字段除外）的值相对应的变量。
- 在访问相应字段之前，不会计算记录中的值。
- 记录中的值最多计算一次。
- 表达式的结果是包含空元数据记录的记录值。
- 记录中字段的顺序由它们在 record-initializer-expression 中出现的顺序定义 。
- 指定的每个字段名在记录中都必须是唯一的，否则为错误。使用序号比较来比较名称。

```
    [ x = 1, x = 2 ] // error: field names must be unique
    [ X = 1, x = 2 ] // OK
```

不含字段的记录称为空记录，并按如下方式写入 ：

```
[] // empty record
```

虽然在访问一个字段或比较两个记录时，记录字段的顺序并不重要，但在其他上下文中（如枚举记录字段时），它是重要的。

获得字段时，这两条记录会产生不同的结果：

```
Record.FieldNames([ x = 1, y = 2 ]) // [ "x", "y" ]
Record.FieldNames([ y = 1, x = 2 ]) // [ "y", "x" ]
```

可以使用 `Record.FieldCount` 函数确定记录中的字段数。例如：

```
Record.FieldCount([ x = 1, y = 2 })  // 2
Record.FieldCount([])                // 0
```

除了使用记录初始化语法 `[ ]`，还可以从值列表、字段名列表或记录类型构造记录。例如：

```
Record.FromList({1, 2}, {"a", "b"})
```

以上等效于：

```
[ a = 1, b = 2 ]
```

为记录值定义了以下运算符：

`x = y`	等于
`x <> y`	不等于
`x & y`	合并

以下示例说明了上述运算符。请注意，如果字段名有重叠，记录合并使用来自右操作数的字段替代来自左操作数的字段。

```
[ a = 1, b = 2 ] & [ c = 3 ]    // [ a = 1, b = 2, c = 3 ]
[ a = 1, b = 2 ] & [ a = 3 ]    // [ a = 3, b = 2 ]
[ a = 1, b = 2 ] = [ b = 2, a = 1 ]         // true
[ a = 1, b = 2, c = 3 ] <> [ a = 1, b = 2 ] // true
```

记录值的本机类型是内部类型 `record`，它指定一个打开的空字段列表。

表

表值是行的有序序列 。行是值的有序序列 。表的类型决定了表中所有行的长度、表列的名称、表列的类型以及表键（如果有的话）的结构。

表没有文字语法。提供了几个标准库函数来构造二进制值。例如，可使用 `#table` 从一列行列表和一列头名称构造表：

```
#table({"x", "x^2"}, {{1,1}, {2,4}, {3,9}})
```

上面的示例构造了包含两列的表，这两列都是 `type any`。

`#table` 也可用于指定完整表类型：

```
#table(
    type table [Digit = number, Name = text],
    {{1,"one"}, {2,"two"}, {3,"three"}}
    )
```

在这里，新表值具有指定列名和列类型的表类型。

为表值定义了以下运算符：

`x = y`	等于
`x <> y`	不等于
`x & y`	串联

表串联将对齐名称相似的列，并为仅出现在一个操作数表中的列填充 `null`。下面的示例演示表串联：

```
  #table({"A","B"}, {{1,2}})
& #table({"B","C"}, {{3,4}})
```

A	B	C
`1`	`2`	`null`
`null`	`3`	`4`

表值的本机类型是一个自定义表类型（派生自内在类型 `table`），它列出列名，指定所有列类型为任意，并且没有键。（有关表类型的详细信息，请参见表类型。）

函数

函数值是将一组参数映射到单个值的值 。函数值的详细信息在函数中描述 。

类型

类型值是一个对其他值进行分类的值 。类型值的详细信息在类型中描述 。

类型

2020/4/26 •

类型值是一个对其他值进行分类的值 。认为按类型分类的值符合 该类型。M 类型系统由以下类型组成：

- 基元类型可以对基元值（`binary`、`date`、`datetime`、`datetimezone`、`duration``list`、`logical`、`null`、`number`、`record`、`text`、`time`、`type`）进行分类，并且基元类型还包括许多抽象类型（`function`、`table`、`any` 和 `none`）
- 记录类型（根据字段名称和值类型对记录值进行分类）
- 列表类型（使用单一项基类型对列表进行分类）
- 函数类型（根据其参数和返回值类型对函数值进行分类）
- 表类型（根据列名、列类型和键对表值进行分类）
- 可为 null 的类型（不仅可以按基类型对所有值进行分类，还可以对 null 值进行分类）
- 类型类型（对属于类型的值进行分类）

一组基元类型 包含基元值的类型、许多抽象类型，以及不对任何值进行唯一分类的类型：`function`、`table`、`any` 和 `none`。所有函数值都符合抽象类型 `function`，所有表值都符合抽象类型 `table`，所有值都符合抽象类型 `any`，没有值符合抽象类型 `none`。类型为 `none` 的表达式必须引发错误或无法终止，因为无法生成符合类型 `none` 的值。请注意，基元类型 `function` 和 `table` 是抽象的，因为任何函数或表都不直接属于这些类型。基元类型 `record` 和 `list` 是非抽象的，因为它们分别表示一个没有定义字段的 open 记录和一个 any 类型的列表。

所有不属于封闭基元类型集成员的类型统称为_自定义类型_。可以使用 `type-expression` 编写自定义类型：

type-expression:
    primary-expression
    `type` *primary-type*
type:
    parenthesized-expression
    primary-type
primary-type:
    primitive-type
    record-type
    list-type
    function-type
    table-type
    nullable-type
primitive-type: one of
    `any binary date datetime datetimezone duration function list logical`
    `none null number record table text time type`

基元类型名称是上下文关键字，仅在类型上下文中识别 。在类型上下文中使用括号将语法移回正则表达式上下文，需要使用关键字类型才能返回到类型上下文 。例如，要在类型上下文中调用函数，可以使用括号：

```
type nullable ( Type.ForList({type number}) )
// type nullable {number}
```

括号还可用于访问名称与基元类型名称相冲突的变量：

```
let  record = type [ A = any ]  in  type {(record)}
// type {[ A = any ]}
```

以下示例定义了对数字列表进行分类的类型：

```
type { number }
```

同样，以下示例定义了一个自定义类型，该类型使用名为 `X` 和 `Y` 的必需字段（其值为数字）对记录进行分类：

```
type [ X = number, Y = number ]
```

值的归属类型是使用标准库函数 `Value.Type` 获取的，如以下示例所示：

```
Value.Type( 2 )                // type number
Value.Type( {2} )              // type list
Value.Type( [ X = 1, Y = 2 ] )  // type record
```

`is` 运算符用于确定值的类型是否与给定类型兼容，如以下示例所示：

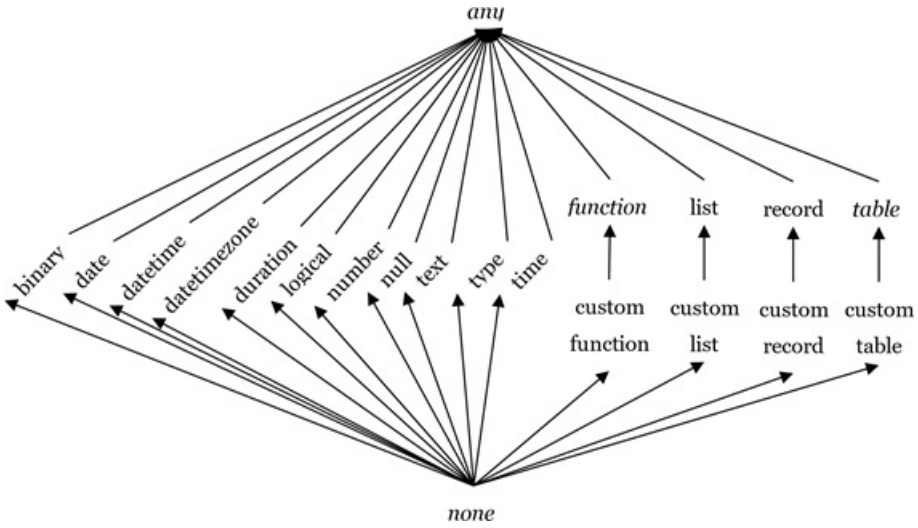
```
1 is number          // true
1 is text            // false
{2} is list          // true
```

`as` 运算符检查该值是否与给定类型兼容，如果不兼容，则引发错误。否则，它将返回原始值。

```
Value.Type( 1 as number )   // type number
{2} as text                 // error, type mismatch
```

请注意，`is` 和 `as` 运算符仅接受基元类型作为其正确的操作数。M 不提供用于检查值是否符合自定义类型的方法。

当且仅当符合 `X` 的所有值也符合 `Y` 时，类型 `X` 与类型 `Y` *兼容*。所有类型都与类型 `any` 兼容，没有类型（除了 `none` 本身）与类型 `none` 兼容。下图显示了兼容性关系。（类型总是与其自身兼容，且类型兼容性可传递。它形成了一个点阵，类型 `any` 在最上，类型 `none` 在最下。）抽象类型的名称设置为_斜体_。



为类型值定义了以下运算符：

[[[	[[
`x = y`	等于
`x <> y`	不等于

类型值的本机类型是内部类型 `type`。

基元类型

M 语言中的类型形成一个源于类型 `any` 的不相交的层次结构，该类型对所有值进行分类。任何 M 值都符合 `any` 的恰好一种基元子类型。从类型 `any` 派生的基元类型闭集如下：

- `type null` 可以对 null 值进行分类。
- `type logical` 可以将值 true 和 false 进行分类。
- `type number` 可以对数字值进行分类。
- `type time` 可以对时间值进行分类。
- `type date` 可以对日期值进行分类。
- `type datetime` 可以对日期/时间值进行分类。
- `type datetimezone` 可以对时区值进行分类。
- `type duration` 可以对持续时间值进行分类。
- `type text` 可以对文本值进行分类。
- `type binary` 可以对二进制值进行分类。
- `type type` 可以对类型值进行分类。
- `type list` 可以对列表值进行分类。
- `type record` 可以对记录值进行分类。
- `type table` 可以对表值进行分类。
- `type function` 可以对函数值进行分类。
- `type anynonnull` 可以对 null 以外的所有值进行分类。内部类型 `none` 不对任何值进行分类。

任何类型

类型 `any` 是抽象的，它对 M 中的所有值进行分类，并且 M 中的所有类型都与 `any` 兼容。`any` 类型的变量可以绑定到所有可能的值。由于 `any` 是抽象的，因此不能将其归属于值，也就是说，没有值直接属于 `any` 类型。

列表类型

任何属于列表的值都符合内部类型 `list`，该类型不会对列表值中的项施加任何限制。

list-type:
    `{` *item-type* `}`
item-type:
    type

计算 *list-type* 的结果是一个基类型为 `list` 的_列表类型值_。

以下示例演示了用于声明同类列表类型的语法：

```
type { number }        // list of numbers type
     { record }        // list of records type
     {{ text }}        // list of lists of text values
```

如果值是一个列表，并且该列表值中的每一项都符合列表类型的项类型，则该值符合列表类型。

列表类型的项类型暗示着一种绑定：符合的列表中所有项都符合项类型。

记录类型

任何记录值都符合内部类型 record，该类型不会对记录值中的字段名称或值施加任何限制。使用 _record-type 值_ 限制有效名称集，以及允许与这些名称相关联的值类型。

record-type:
    `[` *open-record-marker* `]`
    `[` *field-specification-list* `]`
    `[` *field-specification-list* , *open-record-marker* `]`
field-specification-list:
    field-specification
    field-specification `,` *field-specification-list*
field-specification:
    `optional`$_{opt}$ *identifier field-type-specification*$_{opt}$
field-type-specification:
    `=` *field-type*
field-type:
    type
open-record-marker:
    `...`

计算 *record-type* 的结果是一个基类型为 `record` 的类型值。

以下示例演示了用于声明记录类型的语法：

```
type [ X = number, Y = number]
type [ Name = text, Age = number ]
type [ Title = text, optional Description = text ]
type [ Name = text, ... ]
```

记录类型默认为 *closed*，这意味着符合值中不可以存在 *fieldspecification-list* 中没有的字段。在记录类型中包含 *openrecord-marker*，可将该类型声明为 *open*，允许其包含字段规范列表中没有的字段。以下两个表达式等效：

```
type record   // primitive type classifying all records
type [ ... ]  // custom type classifying all records
```

如果值是一条记录，且满足记录类型中的每项字段规范，则该值符合记录类型。若以下任何一项属实，则满足字段规范：

- 记录中存在与规范的标识符匹配的字段名称，并且关联的值符合规范的类型
- 该规范标记为可选，并且在记录中没有相应的字段名称

当且仅当记录类型为 open 时，符合值可以包含字段规范列表中未列出的字段名。

函数类型

任何函数值都符合基元类型 `function`，它不对函数的形式参数的类型或函数的返回值施加任何限制。自定义 _function-type 值_用于对共形函数值的签名施加类型限制。

function-type:
    `function (` *parameter-specification-list*$_{opt}$ `)` *function-return-type*
parameter-specification-list:
    required-parameter-specification-list
    required-parameter-specification-list `,` *optional-parameter-specification-list*
    optional-parameter-specification-list
required-parameter-specification-list:
    required-parameter-specification
    required-parameter-specification `,` *required-parameter-specification-list*
required-parameter-specification:
    parameter-specification
optional-parameter-specification-list:
    optional-parameter-specification
    optional-parameter-specification `,` *optional-parameter-specification-list*
optional-parameter-specification:
    `optional` *parameter-specification*
parameter-specification:
    parameter-name parameter-type
function-return-type:
    assertion
assertion:
    `as` *nullable-primitve-type*

计算 *function-type* 的结果是一个基类型为 `function` 的类型值。

以下示例说明了声明函数类型的语法：

```
type function (x as text) as number
type function (y as number, optional z as text) as any
```

如果函数值的返回类型与函数类型的返回类型兼容，并且函数类型的每个参数规范与函数的位置对应的形式参数兼容，则函数值符合函数类型。如果指定的 *parameter-type* 类型与形式参数的类型兼容，并且如果形式参数是可选的，则参数规范与形式参数兼容。

为了确定函数类型符合性，会忽略形式参数名称。

表类型

使用 _table-type 值_来定义表值的结构。

table-type:
    `table` *row-type*
row-type:
    `[` *field-specification-list* `]`

计算 *table-type* 的结果是一个基类型为 `table` 的类型值。

表的_行类型_将表的列名和列类型指定为封闭记录类型。从而所有表值都符合类型 `table`，其行类型为类型 `record`（空的开放记录类型）。因此，类型表是抽象的，这是因为没有表值可以具有 `table` 的行类型（但是所有表值都具有与类型 `table` 的行类型兼容的行类型）。以下示例显示了表类型的构造：

```
type table [A = text, B = number, C = binary]
// a table type with three columns named A, B, and C
// of column types text, number, and binary, respectively
```

表类型的值还具有表值的_键_的定义。一个键就是一组列名称。最多只有一个键可以指定为表的_主键_。（在 M 中，表键没有语义意义。然而，外部数据源 [如数据库或 OData 反馈] 常在表上定义键。Power Query 使用键信息来提高跨源连接操作等高级功能的性能。）

标准库函数 `Type.TableKeys`、`Type.AddTableKey` 和 `Type.ReplaceTableKeys` 分别可用于获取表类型的键，向表类型添加一个键，替换表类型的所有键。

```
Type.AddTableKey(tableType, {"A", "B"}, false)
// add a non-primary key that combines values from columns A and B
Type.ReplaceTableKeys(tableType, {})
// returns type value with all keys removed
```

可为 null 的类型

对于任何 `type T`，可以使用 *nullable-type* 来派生可为 null 的变量：

nullable-type:
    `nullable` *type*

结果是一个抽象类型，允许类型 *T* 的值或值 `null`。

```
42 is nullable number            // true null is
nullable number                  // true
```

对 `type nullable` *T* 的描述归结为对 `type null` 或 `type` *T* 的描述。（回想一下，可为 null 的类型是抽象的，任何值都不能直接为抽象类型。）

```
Value.Type(42 as nullable number)       // type number
Value.Type(null as nullable number)     // type null
```

标准库函数 `Type.IsNullable` 和 `Type.NonNullable` 可用于测试类型是否可为 null，并使类型不可为 null。

以下条件适用（对于任何 `type T`）：

- `type T` 与 `type nullable T` 兼容
- `Type.NonNullable(type T)` 与 `type T` 兼容

以下是成对等效（对于任何 `type T`）：

```
  type nullable any
  any
  Type.NonNullable(type any) type
  anynonnull
    type nullable
  none type null     Type.NonNullable(type null)
  type none type nullable
  nullable T type nullable T
  Type.NonNullable(Type.NonNullable(type T))
  Type.NonNullable(type T)     Type.NonNullable(type nullable T)
  Type.NonNullable(type T)
  type nullable (Type.NonNullable(type
  T)) type nullable T
```

值的归属类型

值的_归属类型_是_声明_一个值是否符合的类型。当一个值归属为一个类型时，只会进行有限的符合性检查。*M 在可为 null 的基类型之外不执行符合性检查。如果 M 程序作者选择归属类型定义与可为 null 的基元类型相比更复杂的值，则必须确保这些值符合这些类型。*

使用库函数 `Value.ReplaceType` 可以将值归属于类型。如果新类型与值的本机基元类型不兼容，则函数返回一个具有已归属类型的新值，或者引发一个错误。特别地，当尝试归属抽象类型（如 `any`）时，该函数会引发错误。

库函数可以根据输入值的归属类型来选择计算复杂类型并将其归属于结果。

可以使用库函数 `Value.Type` 获得值的归属类型。例如：

```
Value.Type( Value.ReplaceType( {1}, type {number} )
// type {number}
```

类型等效性和兼容性

M 中未定义类型等效性。比较等效性的任何两个类型值可能返回，也可能不返回 `true`。然而，这两种类型（无论是 `true` 还是 `false`）之间的关系总是相同的。

可以使用库函数 `Type.Is` 来确定指定类型和可为 null 的基元类型之间的兼容性，该函数接受任意类型值作为其第一个参数，接受可为 null 的基元类型值作为其第二个参数：

```
Type.Is(type text, type nullable text)  // true
Type.Is(type nullable text, type text)  // false
Type.Is(type number, type text)         // false
Type.Is(type [a=any], type record)      // true
Type.Is(type [a=any], type list)        // false
```

M 中不支持确定指定类型与自定义类型的兼容性。

标准库确实包含一个函数集合，用于从自定义类型中提取定义特征，因此特定的兼容性测试可以实现为 M 表达式。以下是一些示例；有关详细信息，请参阅 M 库规范。

```
Type.ListItem( type {number} )
  // type number
Type.NonNullable( type nullable text )
  // type text
Type.RecordFields( type [A=text, B=time] )
  // [ A = [Type = type text, Optional = false],
  //   B = [Type = type time, Optional = false] ]
Type.TableRow( type table [X=number, Y=date] )
  // type [X = number, Y = date]
Type.FunctionParameters(
        type function (x as number, optional y as text) as number)
  // [ x = type number, y = type nullable text ]
Type.FunctionRequiredParameters(
        type function (x as number, optional y as text) as number)
  // 1
Type.FunctionReturn(
        type function (x as number, optional y as text) as number)
  // type number
```

运算符

2020/4/26 •

本部分定义各种 M 运算符的行为。

运算符优先级

如果某个表达式包含多个运算符，则运算符的优先顺序控制各个运算符的计算顺序。例如，表达式 `x + y * z` 的计算结果为 `x + (y * z)`，因为 `*` 运算符的优先级高于二进制 `+` 运算符。运算符的优先级由其关联的文法产生式的定义来确定。例如，additive-expression 由 `+` 或 `-` 运算符分隔的一系列 multiplicative-expression 组成，因此 `+` 和 `-` 运算符的优先级低于 `*` 和 `/` 运算符。

parenthesized-expression 产生式可用于更改默认优先级排序。

parenthesized-expression:
`(` *expression* `)`

例如：

```
1 + 2 * 3       // 7
(1 + 2) * 3     // 9
```

下表汇总了 M 运算符，并按从高到低的优先级顺序列出了运算符类别。同一类别的运算符具有相等的优先级。

主	*i* @*i*	标识符表达式
	(*x*)	带圆括号表达式
	x[*i*]	LookUp
	x{*y*}	项访问
	x(...)	函数调用
	{*x*, *y*, ...}	列表初始化
	[*i* = *x*, ...]	记录初始化
	...	未实现
一元	+*x*	身份
	-*x*	否定
	`not` *x*	逻辑非
元数据	*x* `meta` *y*	关联元数据

乘法性的	*x* * *y*	乘法
	x / *y*	除法
累加性	*x* + *y*	加法
	x - *y*	减法
关系	*x* <> *y*	小于
	x > *y*	大于
	x <= > *y*	小于或等于
	x >= *y*	大于或等于
等式	*x* = *y*	等于
	x <> *y*	不等于
类型断言	*x* `as` *y*	为兼容的可为 null 的基元类型或错误
类型一致性	*x* `is` *y*	测试是否为兼容的可为 null 的基元类型
逻辑与	*x* `and` *y*	短路合取
逻辑或	*x* `or` *y*	短路析取

运算符和元数据

每个值都有一个关联的记录值，该记录值可以包含有关该值的其他信息。此记录被称为值的元数据记录 。元数据记录可以与任何种类的值（甚至是 `null`）相关联。此类关联的结果是具有给定元数据的新值。

元数据记录只是一个常规记录，可以包含常规记录可包含的任何字段和值，且其本身具有元数据记录。将元数据记录与值关联是一种非侵入性行为。此操作不会更改值在计算中的行为，除非是显式检查元数据记录。

每个值都有默认的元数据记录，即使是未指定的值也是如此。默认元数据记录为空。以下示例演示如何使用 `Value.Metadata` 标准库函数访问文本值的元数据记录：

```
Value.Metadata( "Mozart" )   // []
```

当值与构造新值的运算符或函数一起使用时，通常不保留元数据记录 。例如，如果使用 `&` 运算符来连接两个文本值，那么生成的文本值的元数据为空记录 `[]`。以下表达式等效：

```
"Amadeus " & ("Mozart" meta [ Rating = 5 ])
"Amadeus " & "Mozart"
```

标准库函数 `Value.RemoveMetadata` 和 `Value.ReplaceMetadata` 可用于从值中删除所有元数据，并替换值的元数据（而不是将元数据合并到可能已存在的元数据中）。

返回带有元数据的结果的唯一运算符是元运算符。

结构递归运算符

值可以循环。例如：

```
let l = {0, @l} in l
// {0, {0, {0, ... }}}
[A={B}, B={A}]
// [A = {{ ... }}, B = {{ ... }}]
```

M 通过将记录、列表和表的结构保持为延迟来处理循环值。如果尝试构造循环值，而该值并不会得益于突然插入的结构延迟的值，则会生成错误：

```
[A=B, B=A]
// [A = Error.Record("Expression.Error",
//         "A cyclic reference was encountered during evaluation"),
//  B = Error.Record("Expression.Error",
//         "A cyclic reference was encountered during evaluation"),
// ]
```

M 中的一些运算符由结构递归定义。例如，记录和列表的相等性分别由相应的记录字段和项列表的结合相等性定义。

对于非循环值，应用结构递归会生成值的有限扩展：将重复遍历共享嵌套值，但递归过程始终终止。

在应用结构递归时，循环值存在无限扩展。M 的语义不会针对此类无限扩展作出任何特殊适应 — 例如，尝试比较循环值的相等性通常会耗尽资源并异常终止。

选择和投影运算符

选择和投影运算符允许从列表和记录值中提取数据。

项访问

可使用 item-access-expression，根据某个值在列表或表中从零开始的位置从该列表或表中选择该值。

item-access-expression:
    item-selection
    optional-item-selection
item-selection:
    primary-expression `{` *item-selector* `}`
optional-item-selection:
    primary-expression `{` *item-selector* `}?`
item-selector:
    expression

item-access-expression `x{y}` 返回：

- 对于列表 `x` 和数字 `y`，列表 `x` 的项位于 `y`。列表的第一项被视为具有序号索引零。如果列表中不存在请求的位置，则会引发错误。
- 对于表 `x` 和数字 `y`，表 `x` 的行位于 `y`。表的第一行被视为具有序号索引零。如果表中不存在请求的位置，则会引发错误。
- 对于表 `x` 和记录 `y`，表 `x` 的行与字段记录 `y` 的字段值匹配，并且字段名称与相应的表列名称匹配。如果表中没有唯一的匹配行，则会引发错误。

例如：

```
{"a","b","c"}{0}                        // "a"
{1, [A=2], 3}{1}                        // [A=2]
{true, false}{2}                        // error
#table({"A","B"},{{0,1},{2,1}}){0}      // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}  // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]}  // error
#table({"A","B"},{{0,1},{2,1}}){[B=1]}  // error
```

item-access-expression 还支持 `x{y}?` 形式，如果列表或表 `x` 中不存在位置（或匹配项）`y`，这种形式将返回 `null` 。如果存在多个 `y` 匹配项，仍会引发错误。

例如：

```
{"a","b","c"}{0}?                       // "a"
{1, [A=2], 3}{1}?                       // [A=2]
{true, false}{2}?                       // null
#table({"A","B"},{{0,1},{2,1}}){0}      // [A=0,B=1]
#table({"A","B"},{{0,1},{2,1}}){[A=2]}  // [A=2,B=1]
#table({"A","B"},{{0,1},{2,1}}){[B=3]}  // null
#table({"A","B"},{{0,1},{2,1}}){[B=1]}  // error
```

项访问不会强制对列表或表项（正在访问的项除外）进行求值。例如：

```
{ error "a", 1, error "c"}{1}  // 1
{ error "a", error "b"}{1}     // error "b"
```

对项访问运算符 `x{y}` 求值时，存在以下情况：

- 传播在计算表达式 `x` 或 `y` 期间引发的错误。
- 表达式 `x` 生成列表或表值。
- 表达式 `y` 生成数值，或在 `x` 生成表值时生成记录值。
- 如果 `y` 生成数值，并且 `y` 的值为负，则会引发一个原因代码为 `"Expression.Error"` 的错误。
- 如果 `y` 生成数值，并且 `y` 的值大于或等于 `x` 的计数，则除非使用了可选运算符形式 `x{y}?`（在这种情况下，将返回值 `null`），否则将引发原因代码为 `"Expression.Error"` 的错误。
- 如果 `x` 生成表值，`y` 生成记录值，并且 `x` 中的 `y` 没有匹配项，则除非使用了可选运算符形式 `x{y}?`（在这种情况下，将返回值 `null`），否则将引发原因代码为 `"Expression.Error"` 的错误。
- 如果 `x` 生成表值，`y` 生成记录值，并且 `x` 中的 `y` 具有多个匹配项，则会引发原因代码为 `"Expression.Error"` 的错误。

在项选择过程中，不会计算 `x` 中的任何项（位于 `y` 位置的项除外）。（对于流式处理列表或表，会跳过位置 `y` 前面的项或行，这可能会导致计算这些项或行，具体取决于列表或表的源。）

字段访问

field-access-expression 用于从记录中选择值，或将记录或表分别投影到具有较少字段或列的记录或表 。

field-access-expression:
    field-selection
    implicit-target-field-selection
    projection
    implicit-target-projection
field-selection:

primary-expression field-selector
field-selector:
required-field-selector
optional-field-selector
required-field-selector:
`[` *field-name* `]`
optional-field-selector:
`[` *field-name* `]?`
field-name:
generalized-identifier
implicit-target-field-selection:
field-selector
projection:
primary-expression required-projection
primary-expression optional-projection
required-projection:
`[` *required-selector-list* `]`
optional-projection:
`[` *required-selector-list* `]?`
required-selector-list:
required-field-selector
required-selector-list `,` *required-field-selector*
implicit-target -projection:
projection

字段访问的最简单形式是必需字段选择。它使用运算符 `x[y]` 按字段名称在记录中查找字段。如果 `x` 中不存在字段 `y`，则会引发错误。`x[y]?` 形式用于执行可选字段选择，如果记录中不存在请求的字段，则返回 `null`。

例如：

```
[A=1,B=2][B]       // 2
[A=1,B=2][C]       // error
[A=1,B=2][C]?      // null
```

对于必需记录投影和可选记录投影，运算符支持对多个字段进行集体访问。运算符 `x[[y1],[y2],...]` 将记录投影到具有更少字段的新记录（通过 `y1`、`y2`、`...` 选择）。如果不存在选定字段，则会引发错误。运算符 `x[[y1],[y2],...]` 将记录投影到具有通过 `y1`、`y2`、`...` 选择的字段的新记录：如果某一字段缺失，则改用 `null`。例如：

```
[A=1,B=2][[B]]          // [B=2]
[A=1,B=2][[C]]          // error
[A=1,B=2][[B],[C]]?     // [B=2,C=null]
```

支持将形式 `[y]` 和 `[y]?` 作为对标识符 `_`（下划线）的速记引用。以下两个表达式等效：

```
[A]
_[A]
```

以下示例演示了字段访问的速记形式：

```
let _ = [A=1,B=2] in [A] //1
```

还支持将形式 `[[y1],[y2],...]` 和 `[[y1],[y2],...]?` 作为速记，以下两个表达式同样等效：

```
[[A],[B]]
_[[A],[B]]
```

在结合使用 `each` 速记（一种引入单个参数 `_` 的函数的方法）时，速记形式尤其有用（有关详细信息，请参阅简化的声明）。这两种速记共同简化了常见的高阶函数表达式：

```
List.Select( {[a=1, b=1], [a=2, b=4]}, each [a] = [b])
// {[a=1, b=1]}
```

上面的表达式与以下这种看上去更为费解的普通写法等效：

```
List.Select( {[a=1, b=1], [a=2, b=4]}, (_) => _[a] = _[b])
// {[a=1, b=1]}
```

字段访问不会强制对字段（正在访问的字段除外）进行求值。例如：

```
[A=error "a", B=1, C=error "c"][B]  // 1
[A=error "a", B=error "b"][B]       // error "b"
```

对字段访问运算符 `x[y]``x[y]?`、`x[[y]]` 或 `x[[y]]?` 求值时，存在以下情况：

- 传播在计算表达式 `x` 期间引发的错误。
- 计算字段 `y` 时引发的错误与字段 `y` 永久关联，然后进行传播。将来对字段 `y` 进行的任何访问都将引发相同的错误。
- 表达式 `x` 生成记录或表值，或引发错误。
- 如果标识符 `y` 命名 `x` 中不存在的字段，则除非使用了可选运算符形式 `...?`（在这种情况下，将返回值 `null`），否则将引发原因代码为 `"Expression.Error"` 的错误。

在字段访问过程中，不会计算 `x` 的任何字段（由 `y` 命名的字段除外）。

元数据运算符

值的元数据记录使用元运算符 (`x meta y`) 进行修正 。

metadata-expression:
unary-expression
unary-expression `meta` *unary-expression*

以下示例使用 `meta` 运算符构造一个包含元数据记录的文本值，然后使用 `Value.Metadata` 访问生成的值的元数据记录：

```
Value.Metadata( "Mozart" meta [ Rating = 5 ] )
// [Rating = 5 ]
Value.Metadata( "Mozart" meta [ Rating = 5 ] )[Rating]
// 5
```

在应用元数据组合运算符 `x meta y` 时，存在以下情况：

- 传播在计算 `x` 或 `y` 表达式时引发的错误。

- `y` 表达式必须为记录，否则将引发原因代码为 `"Expression.Error"` 的错误。
- 生成的元数据记录是与 `y` 合并的 `x` 的元数据记录。（有关记录合并的语义，请参阅记录合并。）
- 生成的值是 `x` 表达式中的值（不包含其元数据），并且附加了新计算的元数据记录。

标准库函数 `Value.RemoveMetadata` 和 `Value.ReplaceMetadata` 可用于从值中删除所有元数据，并替换值的元数据（而不是将元数据合并到可能已存在的元数据中）。以下表达式等效：

```
x meta y
Value.ReplaceMetadata(x, Value.Metadata(x) & y)
Value.RemoveMetadata(x) meta (Value.Metadata(x) & y)
```

相等运算符

相等运算符 `=` 用于确定两个值是否相等 。不相等运算符 `<>` 用于确定两个值是否不相等 。

equality-expression:
relational-expression
relational-expression `=` *equality-expression*
relational-expression `<>` *equality-expression*

例如：

```
1 = 1           // true
1 = 2           // false
1 <> 1          // false
1 <> 2          // true
null = true     // false
null = null     // true
```

元数据不属于相等比较或不相等比较。例如：

```
(1 meta [ a = 1 ]) = (1 meta [ a = 2 ]) // true
(1 meta [ a = 1 ]) = 1                  // true
```

在应用相等运算符 `x = y` 和 `x <> y` 时，存在以下情况：

- 传播在计算 `x` 或 `y` 表达式时引发的错误。
- 如果值相等，则 `=` 运算符的结果为 `true`，否则为 `false`。
- 如果值相等，则 `<>` 运算符的结果为 `false`，否则为 `true`。
- 比较中不包含元数据记录。
- 如果 `x` 和 `y` 表达式的计算结果不是同一类值，则这些值不相等。
- 如果 `x` 和 `y` 表达式的计算结果是同一类值，则可以应用一些特定的规则来确定它们是否相等，定义如下。
- 以下情况始终存在：

```
    (x = y) = not (x <> y)
```

相等运算符针对以下类型定义：

- `null` 值仅等于其自身。

```
null = null    // true
null = true    // false
null = false   // false
```

- 逻辑值 `true` 和 `false` 仅等于其自身。例如：

```
true = true      // true
false = false    // true
true = false     // false
true = 1         // false
```

- 使用指定的精度比较数字：
  - 如果任一数字为 `#nan`，则这两个数字不相同。
  - 如果这两个数字都不为 `#nan`，则将使用数值的按位比较对这两个数字进行比较。
  - `#nan` 是唯一不等于其自身的值。

    例如：

```
1 = 1,              // true
1.0 = 1             // true
2 = 1               // false
#nan = #nan         // false
#nan <> #nan        // true
```

- 如果两个持续时间均表示 100 毫微秒的时间刻度，则这两个持续时间相等。
- 如果两个时间的各个部分（时、分、秒）的度量值相等，则这两个时间相等。
- 如果两个日期的各个部分（年、月、日）的度量值相等，则这两个日期相等。
- 如果两个日期时间的各个部分（年、月、日、时、分、秒）的度量值相等，则这两个日期时间相等。
- 如果两个 DateTimeZone 时区的相应 UTC 日期时间相等，则这两个 DateTimeZone 相等。要到达相应的 UTC 日期时间，需要从 DateTimeZone 的日期时间部分减去小时/分钟偏移量。
- 如果使用序号、区分大小写和不区分区域性的比较，两个文本值在相应位置具有相同长度和相同字符，则这两个文本值相等。
- 如果满足以下所有条件，则两个列表值相等：
  - 这两个列表均包含相同数量的项。
  - 列表中每个值在位置上相对应的项都相等。这意味着，不仅列表需要包含相等的项，而且这些项需要采用相同的顺序。

    例如：

```
{1, 2} = {1, 2}     // true
{2, 1} = {1, 2}     // false
{1, 2, 3} = {1, 2}  // false
```

- 如果满足以下所有条件，则两条记录相等：
  - 字段数相同。
  - 一条记录的每个字段名称也出现在另一条记录中。

- 一条记录的每个字段的值等于另一条记录中名称相同的字段。

  例如：

```
[ A = 1, B = 2 ] = [ A = 1, B = 2 ]        // true
[ B = 2, A = 1 ] = [ A = 1, B = 2 ]        // true
[ A = 1, B = 2, C = 3 ] = [ A = 1, B = 2 ] // false
[ A = 1 ] = [ A = 1, B = 2 ]               // false
```

- 如果满足以下所有条件，则两个表相等：
  - 列数相同。
  - 一个表中的每个列名称也出现在另一个表中。
  - 行数相同。
  - 每行的相应单元格中的值相等。

    例如：

```
#table({"A","B"},{{1,2}}) = #table({"A","B"},{{1,2}}) // true
#table({"A","B"},{{1,2}}) = #table({"X","Y"},{{1,2}}) // false
#table({"A","B"},{{1,2}}) = #table({"B","A"},{{2,1}}) // true
```

- 函数值等于其自身，但可能等于也可能不等于另一函数值。如果两个函数值被视为相等，则调用时它们将具有相同的行为。

  两个给定函数值将始终具有相同的相等关系。

- 类型值等于其自身，但可能等于也可能不等于另一类型值。如果两个类型值被视为相等，则查询一致性时它们将具有相同的行为。

  两个给定类型值将始终具有相同的相等关系。

关系运算符

`<`、`>`、`<=` 和 `>=` 运算符称为关系运算符 。

relational-expression:
additive-expression
additive-expression `<` *relational-expression*
additive-expression `>` *relational-expression*
additive-expression `<=` _relational-expression
additive-expression `>=` *relational-expression*

这些运算符用于确定两个值之间的相对排序关系，如下表所示：

"	"
`x < y`	如果 `x` 小于 `y`，则为 `true`；否则为 `false`
`x > y`	如果 `x` 大于 `y`，则为 `true`；否则为 `false`
`x <= y`	如果 `x` 小于等于 `y`，则为 `true`；否则为 `false`

“	“
`x >= y`	如果 `x` 大于等于 `y`，则为 `true`；否则为 `false`

例如：

```
0 <= 1           // true
null < 1         // null
null <= null     // null
"ab" < "abc"     // true
#nan >= #nan     // false
#nan <= #nan     // false
```

在计算包含关系运算符的表达式时，存在以下情况：

- 传播在计算 `x` 或 `y` 操作数表达式时引发的错误。
- 通过计算 `x` 和 `y` 表达式而生成的值必须是数字、日期、日期时间、datetimezone、持续时间、逻辑、null 或时间值。否则，将引发原因代码为 `"Expression.Error"` 的错误。
- 如果其中一个操作数或两个操作数都为 `null`，则结果为 `null` 值。
- 如果两个操作数都是逻辑操作数，则值 `true` 被视为大于 `false`。
- 如果两个操作数均为持续时间，则将根据它们所表示的 100 毫微秒时间刻度的总数来比较这些值。
- 通过比较两个时间的小时部分（如果相等，则比较分钟部分，如果分钟部分也相等，则比较秒数部分）来比较这两个时间。
- 通过比较两个日期的年份部分（如果相等，则比较月份部分，如果月份部分也相等，则比较日部分）来比较这两个日期。
- 通过比较两个日期时间的年份部分（如果相等，则比较月份部分，如果月份部分也相等，则比较日部分，如果日部分也相等，则比较小时部分，如果小时部分也相等，则比较分钟部分，如果分钟部分也相等，则比较秒数部分）来比较这两个日期时间。
- 通过减去两个 datetimezone 的小时/分钟偏移量将其规范化为 UTC，然后比较其日期时间部分，来比较这两个 datetimezone。
- 根据 IEEE 754 标准的规则比较 `x` 和 `y` 这两个数字：
  - 如果任一操作数为 `#nan`，则所有关系运算符的结果都为 `false`。
  - 如果这两个操作数都不为 `#nan`，运算符将根据排序比较这两个浮点操作数的值

    ```
    <code>-&#8734; < -max < ... < -min < -0.0 = +0.0 < +min < ... < +max < +&#8734;</code>
    ```

    其中，最小值和最大值可使用最小和最大正有限值来表示。`-∞` 和 `+∞` 的 M 名称为 `-#infinity` 和 `#infinity`。

    此排序的明显效果是：

    - 负零和正零被视为相等。
    - `-#infinity` 值被视为小于其他所有数值，但等于另一 `-#infinity`。
    - `#infinity` 值被视为大于其他所有数值，但等于另一 `#infinity`。

条件逻辑运算符

`and` 和 `or` 运算符称为条件逻辑运算符。

logical-or-expression:
    logical-and-expression
    logical-and-expression `or` *logical-or-expression*
logical-and-expression:
    is-expression
    is-expression `and` *logical-and-expression*

`or` 运算符在至少一个操作数为 `true` 时返回 `true`。当且仅当左操作数不为 `true` 时，才计算右操作数。

`and` 运算符在至少一个操作数为 `false` 时返回 `false`。当且仅当左操作数不为 `false` 时，才计算右操作数。

下面显示了 `or` 和 `and` 运算符的真值表，其中包含对垂直轴上左操作数表达式的求值结果和对水平轴上右操作数表达式的求值结果。

`and`	`true`	`false`	`null`	`error`
`true`	`true`	`false`	`null`	`error`
`false`	`false`	`false`	`false`	`false`
`null`	`null`	`false`	`null`	`error`
`error`	`error`	`error`	`error`	`error`

`or`	`true`	`false`	`null`	`error`
`true`	`true`	`true`	`true`	`true`
`false`	`true`	`false`	`null`	`error`
`null`	`true`	`null`	`null`	`error`
`error`	`error`	`error`	`error`	`error`

在计算包含条件逻辑运算符的表达式时，存在以下情况：

- 传播在计算 `x` 或 `y` 表达式时引发的错误。
- 条件逻辑运算符通过类型 `logical` 和 `null` 定义。如果操作数的值不是这些类型，则会引发原因代码为 `"Expression.Error"` 的错误。
- 结果为逻辑值。
- 在 `x` 或 `y` 表达式中，当且仅当 `x` 的计算结果不为 `true` 时，才会计算 `y` 表达式。
- 在 `x` 和 `y` 表达式中，当且仅当 `x` 的计算结果不为 `false` 时，才会计算 `y` 表达式。

最后两个属性赋予了条件逻辑运算符"条件"资格;属性也称为"短路"。这些属性对于编写精简的受保护的谓词非常有用 。例如，以下表达式等效：

```
d <> 0 and n/d > 1 if d <> 0 then n/d > 1 else false
```

算术运算符

`+`、`-`、`*` 和 `/` 运算符是算术运算符 。

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*
multiplicative-expression:
metadata- expression
multiplicative-expression `*` *metadata-expression*
multiplicative-expression `/` *metadata-expression*

精度

M 中的数字使用多种表示形式进行存储，以尽可能多地保留来自各种源的数字相关信息。应用于数字的运算符只会根据需要将数字从一种表示形式转换为另一种表示形式。M 支持两种精度：

`Precision.Decimal`	128 位十进制表示形式，范围为 ±1.0 x 10-28 至 ±7.9 x 1028，有效位数为 28-29。
`Precision.Double`	使用尾数和指数的科学表示形式;符合 64 位二进制双精度 IEEE 754 算术标准 IEEE 754-2008。

执行算术运算的方式如下：选择一个精度，将两个操作数都转换为该精度（如有必要），然后执行实际运算，最后返回一个采用所选精度的数字。

内置算术运算符（`+`、`-`、`*`、`/`）使用双精度。标准库函数（`Value.Add`、`Value.Subtract`、`Value.Multiply`、`Value.Divide`）可用于使用特定精度模型请求这些操作。

- 不可能出现数值溢出：`#infinity` 或 `-#infinity` 表示因度量值过大而无法表示的值。
- 不可能出现数值下溢：`0` 和 `-0` 表示因度量值过小而无法表示的值。
- IEEE 754 特殊值 `#nan`（NaN — 不是数字）用于涵盖在算数上无效的案例，例如零除以零。
- 从十进制精度到双精度的转换通过将十进制数字舍入到最接近的等效双精度值来实现。
- 从双精度到十进制精度的转换通过将双精度数字舍入到最接近的等效十进制值（如有必要，溢出到 `#infinity` 或 `-#infinity` 值）来实现。

加法运算符

加法运算符 (`x + y`) 的解释取决于计算表达式 x 和 y 所得到的值的类型，如下所示：

X	Y		
`type number`	`type number`	`type number`	数值求和

x	y	结果	结果
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type duration`	`type duration`	度量值的数值求和
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	
`type` 日期时间	`type duration`	`type` 日期时间	持续时间的日期时间偏移
`type duration`	`type` 日期时间	`type` 日期时间	
`type` 日期时间	`null`	`null`	
`null`	`type` 日期时间	`null`	

在表中，`type` 日期时间代表 `type date`、`type datetime`、`type datetimezone` 或 `type time` 中的任何一项 。在添加某些日期时间类型的持续时间和值时，生成的值为相同类型 。

对于表中所列值之外的其他值组合，将引发原因代码为 `"Expression.Error"` 的错误。以下各部分将介绍每种组合。

传播在计算任一操作数时引发的错误。

数值求和

两个数字的求和使用加法运算符进行计算，并生成一个数字 。

例如：

```
1 + 1            // 2
#nan + #infinity  // #nan
```

对数字采用的加法运算符 `+` 使用双精度；标准库函数 `Value.Add` 可用于指定十进制精度。计算数字之和时，存在以下情况：

- 采用双精度的求和根据 64 位二进制双精度 IEEE 754 算法 IEEE 754-2008 的规则计算得出。下表列出了非零有限值、零、无限值和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 是非零有限值，`z` 是 `x + y` 的结果。如果 `x` 和 `y` 度量值相同但异号，则 `z` 为正零。如果 `x + y` 由于过大而无法采用目标类型表示，则 `z` 为与 `x + y` 具有相同符号的无穷值。

+	y	+0	-0	+∞	-∞	NAN
x	z	x	x	+∞	-∞	NaN
+0	y	+0	+0	+∞	-∞	NaN
-0	y	+0	-0	+∞	-∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN

+	Y	+0	-0	+°	-°	NAN
-°	-°	-°	-°	NaN	-°	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 采用十进制精度计算总和时不会丢失精度。结果的小数位数是两个操作数的小数位数中较大的一个。

持续时间之和

两个持续时间之和是指这两个持续时间所表示的 100 毫微秒时间刻度数的总和。例如：

```
#duration(2,1,0,15.1) + #duration(0,1,30,45.3)
// #duration(2, 2, 31, 0.4)
```

持续时间的日期时间偏移

可使用 `x + y` 添加日期时间 `x` 和持续时间 `y`，以计算其与线性时间线上 `x` 的距离的度良值完全等同于 `y` 的新日期时间 。此处，日期时间表示 `Date`、`DateTime`、`DateTimeZone` 或 `Time` 中的任何一项，并且非 null 结果将为同一类型 。可按如下所示计算持续时间的日期时间偏移：

- 如果指定了时期值后的日期时间天数，请使用以下信息元素构造新的日期时间：
  - 计算时期后的新天数，相当于将 y 的度量值除以 24 小时内的 100 毫微秒时间刻度数，然后截断结果的小数部分，并将此值添加到时期后的 x 天数。
  - 计算午夜后的新时间刻度，相当于将 y 的量级与午夜后 x 的时间刻度相加，除以 24 小时内的 100 毫微秒时间刻度数。如果 x 自午夜后未指定任何时间刻度值，则假定一个值 0。
  - 复制 x 的值，表示相对于 UTC 的分钟偏移量未更改。
- 如果未指定时期值后的日期时间天数，请使用以下指定的信息元素构造新的日期时间：
  - 计算午夜后的新时间刻度，相当于将 y 的量级与午夜后 x 的时间刻度相加，除以 24 小时内的 100 毫微秒时间刻度数。如果 x 自午夜后未指定任何时间刻度值，则假定一个值 0。
  - 复制 x 的值，表示 epoch 后的天数和相对于 UTC 的分钟偏移量未更改。

以下示例显示如何在日期时间指定时期后的天数时，计算绝对时间总和 ：

```
#date(2010,05,20) + #duration(0,8,0,0)
    //#datetime( 2010, 5, 20, 8, 0, 0 )
    //2010-05-20T08:00:00

#date(2010,01,31) + #duration(30,08,0,0)
    //#datetime(2010, 3, 2, 8, 0, 0)
    //2010-03-02T08:00:00

#datetime(2010,05,20,12,00,00,-08) + #duration(0,04,30,00)
    //#datetime(2010, 5, 20, 16, 30, 0, -8, 0)
    //2010-05-20T16:30:00-08:00

#datetime(2010,10,10,0,0,0,0) + #duration(1,0,0,0)
   //#datetime(2010, 10, 11, 0, 0, 0, 0, 0)
   //2010-10-11T00:00:00+00:00
```

以下示例显示如何计算给定时间的持续时间的日期时间偏移：

```
#time(8,0,0) + #duration(30,5,0,0)
   //#time(13, 0, 0)
   //13:00:00
```

减法运算符

减法运算符 (`x - y`) 的解释取决于计算表达式 `x` 和 `y` 所得到的值的类型，如下所示：

X	Y	结果	结果
`type number`	`type number`	`type number`	数值差
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type duration`	`type duration`	度量值的数值差
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	
`type` 日期时间	`type` 日期时间	`type duration`	日期时间之间的持续时间
`type` 日期时间	`type duration`	`type` 日期时间	求反持续时间的日期时间偏移
`type` 日期时间	`null`	`null`	
`null`	`type` 日期时间	`null`	

在表中，`type` 日期时间代表 `type date`、`type datetime`、`type datetimezone` 或 `type time` 中的任何一项 。在从某些日期时间类型的值中减去持续时间时，生成的值为相同类型 。

对于表中所列值之外的其他值组合，将引发原因代码为 `"Expression.Error"` 的错误。以下各部分将介绍每种组合。

传播在计算任一操作数时引发的错误。

数值差

两个数字之间的差使用减法运算符进行计算，并生成一个数字 。例如：

```
1 - 1              // 0
#nan - #infinity   // #nan
```

对数字采用的减法运算符 `-` 使用双精度；标准库函数 `Value.Subtract` 可用于指定十进制精度。计算数字之差时，存在以下情况：

- 采用双精度的求差根据 64 位二进制双精度 IEEE 754 算法 IEEE 754-2008 的规则计算得出。下表列出了非零有限值、零、无限值和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 是非零有限值，`z` 是 `x - y` 的结果。如果 `x` 和 `y` 相等，则 `z` 为正零。如果 `x - y` 由于过大而无法采用目标类型表示，则 `z` 为与 `x - y` 具有相同符号的无穷值。

-	Y	+0	-0	+°	-°	NAN
x	z	x	x	-°	+°	NaN
+0	-y	+0	+0	-°	+°	NaN
-0	-y	-0	+0	-°	+°	NaN
+°	+°	+°	+°	NaN	+°	NaN
-°	-°	-°	-°	-°	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 采用十进制精度计算差值时不会丢失精度。结果的小数位数是两个操作数的小数位数中较大的一个。

持续时间之差

两个持续时间之差是指每个持续时间所表示的 100 毫微秒时间刻度数之间的差值。例如:

```
#duration(1,2,30,0) - #duration(0,0,0,30.45)
// #duration(1, 2, 29, 29.55)
```

求反持续时间的日期时间偏移

可使用 `x - y` 减去日期时间 `x` 和持续时间 `y`，以计算新的日期时间 。此处，日期时间表示 `date`、`datetime`、`datetimezone` 或 `time` 中的任何一项 。生成的日期时间与线性时间线上 `x` 的距离的度量值与 `y` 完全相同，方向与 `y` 符号相反 。减去正数持续时间将生成在时间上相对于 `x` 向后推移的结果，而减去负值将生成在时间上向前推移的结果。

```
#date(2010,05,20) - #duration(00,08,00,00)
   //#datetime(2010, 5, 19, 16, 0, 0)
   //2010-05-19T16:00:00
#date(2010,01,31) - #duration( 30,08,00,00)
   //#datetime(2009, 12, 31, 16, 0, 0)
   //2009-12-31T16:00:00
```

两个日期时间之间的持续时间

可使用 `t - u` 将两个日期时间 `t` 和 `u` 相减，以计算它们之间的持续时间 。此处，日期时间表示 `date`、`datetime`、`datetimezone` 或 `time` 中的任何一项 。从 `t` 中减去 `u` 所得到的持续时间在添加 `u` 时必须生成 `t` 。

```
#date(2010,01,31) - #date(2010,01,15)
// #duration(16,00,00,00)
// 16.00:00:00

#date(2010,01,15)- #date(2010,01,31)
// #duration(-16,00,00,00)
// -16.00:00:00

#datetime(2010,05,20,16,06,00,-08,00) -
#datetime(2008,12,15,04,19,19,03,00)
// #duration(521,22,46,41)
// 521.22:46:41
```

当 `u > t` 结果为负持续时间时，减去 `t - u`:

```
#time(01,30,00) - #time(08,00,00)
// #duration(0, -6, -30, 0)
```

使用 `t - u` 将两个日期时间相减时，存在以下情况：

- u + (t - u) = t

乘法运算符

乘法运算符 (`x * y`) 的解释取决于计算表达式 x 和 y 所得到的值的类型，如下所示：

X	Y	结果	说明
`type number`	`type number`	`type number`	数值乘积
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type number`	`type duration`	持续时间倍数
`type number`	`type duration`	`type duration`	持续时间倍数
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	

对于表中所列值之外的其他值组合，将引发原因代码为 `"Expression.Error"` 的错误。以下各部分将介绍每种组合。

传播在计算任一操作数时引发的错误。

数值乘积

两个数字的乘积使用乘法运算符进行计算，并生成一个数字 。例如：

```
2 * 4               // 8
6 * null            // null
#nan * #infinity    // #nan
```

对数字采用的乘法运算符 `*` 使用双精度；标准库函数 `Value.Multiply` 可用于指定十进制精度。计算数字的乘积时，存在以下情况：

- 采用双精度的求积根据 64 位二进制双精度 IEEE 754 算法 IEEE 754-2008 的规则计算得出。下表列出了非零有限值、零、无限值和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 为正有限值。`x * y` 的结果为 `z` 。如果结果相对于目标类型而言太大，则 `z` 为无穷值。如果结果相对于目标类型而言太小，则 `z` 为零。

*	+Y	-Y	+0	-0	+∞	-∞	NAN
+x	+z	-Z	+0	-0	+∞	-∞	NaN
-x	-Z	+z	-0	+0	-∞	+∞	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN

*	+Y	-Y	+0	-0	+°	-°	NAN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+°	+°	-°	NaN	NaN	+°	-°	NaN
-°	-°	+°	NaN	NaN	-°	+°	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 采用十进制精度计算乘积时不会丢失精度。结果的小数位数是两个操作数的小数位数中较大的一个。

持续时间的倍数

持续时间和数字的乘积是指由持续时间操作数乘以数字操作数表示的 100 毫微秒时间刻度数。例如：

```
#duration(2,1,0,15.1) * 2
// #duration(4, 2, 0, 30.2)
```

除法运算符

除法运算符 (`x / y`) 的解释取决于计算表达式 `x` 和 `y` 所得到的值的类型，如下所示：

X	Y	ㄖ	ㄖ
`type number`	`type number`	`type number`	数值商
`type number`	`null`	`null`	
`null`	`type number`	`null`	
`type duration`	`type number`	`type duration`	持续时间的分数
`type duration`	`type duration`	`type duration`	持续时间的数值商
`type duration`	`null`	`null`	
`null`	`type duration`	`null`	

对于表中所列值之外的其他值组合，将引发原因代码为 `"Expression.Error"` 的错误。以下各部分将介绍每种组合。

传播在计算任一操作数时引发的错误。

数值商

两个数字的商使用除法运算符进行计算，并生成一个数字 。例如：

```
8 / 2              // 4
8 / 0              // #infinity
0 / 0              // #nan
0 / null           // null
#nan / #infinity   // #nan
```

对数字采用的除法运算符 `/` 使用双精度；标准库函数 `Value.Divide` 可用于指定十进制精度。计算数字的商时，存

在以下情况：

- 采用双精度的求商根据 64 位二进制双精度 IEEE 754 算法 IEEE 754-2008 的规则计算得出。下表列出了非零有限值、零、无限值和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 为正有限值。`x / y` 的结果为 `z`。如果结果相对于目标类型而言太大，则 `z` 为无穷值。如果结果相对于目标类型而言太小，则 `z` 为零。

/	+Y	-Y	+0	-0	+°	-°	NAN
+x	+z	-Z	+°	-°	+0	-0	NaN
-x	-Z	+z	-°	+°	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+°	+°	-°	+°	-°	NaN	NaN	NaN
-°	-°	+°	-°	+°	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 采用十进制精度计算总和时不会丢失精度。结果的小数位数是两个操作数的小数位数中较大的一个。

持续时间之商

两个持续时间之商是指这两个持续时间所表示的 100 毫微秒时间刻度数的商。例如：

```
#duration(2,0,0,0) / #duration(0,1,30,0)
// 32
```

缩放的持续时间

持续时间 `x` 和数字 `y` 的商是表示商的持续时间，该商是由持续时间 `x` 和数字 `y` 所表示的 100 毫微秒时间刻度数的商。例如：

```
#duration(2,0,0,0) / 32
// #duration(0,1,30,0)
```

结构组合

组合运算符 (`x & y`) 通过以下类型的值定义：

X	Y	结果	结果
`type text`	`type text`	`type text`	串联
`type text`	`null`	`null`	
`null`	`type text`	`null`	
`type date`	`type time`	`type datetime`	合并

x	y	结果	结果
`type date`	`null`	`null`	
`null`	`type time`	`null`	
`type list`	`type list`	`type list`	串联
`type record`	`type record`	`type record`	合并
`type table`	`type table`	`type table`	串联

串联

可使用 `x & y` 串联两个文本、两个列表或两个表值。

下面的示例演示文本值串联：

```
"AB" & "CDE"     // "ABCDE"
```

下面的示例演示列表串联：

```
{1, 2} & {3}     // {1, 2, 3}
```

在使用 `x & y` 串联两个值时，存在以下情况：

- 传播在计算 `x` 或 `y` 表达式时引发的错误。
- 如果 `x` 或 `y` 的项包含错误，则不会传播任何错误。
- 串联两个文本值将生成一个文本值，该值包含 x 的值，随后紧跟 y 的值。如果其中一个操作数为 null，另一个为文本值，则结果为 null。
- 串联两个列表将生成一个列表，该列表包含 `x` 的所有项，后跟 `y` 的所有项。
- 串联两个表将生成一个表，该表包含这两个操作数表的所有列。保留 `x` 的列排序，后跟仅显示在 `y` 中的列，保留其相对顺序。对于仅出现在其中一个操作数中的列，`null` 用于填充另一个操作数的单元格值。

合并

记录**合并**

可使用 `x & y` 来合并两条记录，并生成一条记录，其中包括来自 `x` 和 `y` 的字段。

以下示例说明了如何合并记录：

```
[ x = 1 ] & [ y = 2 ]               // [ x = 1, y = 2 ]
[ x = 1, y = 2 ] & [ x = 3, z = 4 ] // [ x = 3, y = 2, z = 4 ]
```

在使用 `x + y` 合并两条记录时，存在以下情况：

- 传播在计算 `x` 或 `y` 表达式时引发的错误。
- 如果某一字段同时出现在 `x` 和 `y` 中，则使用 `y` 中的值。
- 生成的记录中字段的顺序为 `x`，后跟 `y` 中不属于 `x` 的字段，其顺序与它们在 `y` 中的显示顺序相同。

- 合并记录不会导致对值进行计算。
- 由于字段包含错误，因此不会引发错误。
- 结果是一条记录。

日期时间合并

可使用 `x & y` 合并日期 `x` 与时间 `y`，并生成一个将 `x` 和 `y` 中的各个部分组合在一起的日期时间。

以下示例演示了如何合并日期和时间：

```
#date(2013,02,26) & #time(09,17,00)
// #datetime(2013,02,26,09,17,00)
```

在使用 `x + y` 合并两条记录时，存在以下情况：

- 传播在计算 `x` 或 `y` 表达式时引发的错误。
- 结果是一个日期时间。

一元运算符

`+`、`-` 和 `not` 运算符是一元运算符。

unary-expression:
  type-expression
  `+` *unary expression*
  `-` *unary expression*
  `not` *unary expression*

一元加运算符

一元加运算符 (`+x`) 针对以下类型的值定义：

x	"	"
`type number`	`type number`	一元加
`type duration`	`type duration`	一元加
`null`	`null	

对于其他值，将引发原因代码为 `"Expression.Error"` 的错误。

一元加运算符允许将 `+` 符号应用于数字、日期时间或 null 值。结果是该相同值。例如：

```
+ - 1                // -1
+ + 1                // 1
+ #nan               // #nan
+ #duration(0,1,30,0) // #duration(0,1,30,0)
```

在计算一元加运算符 `+x` 时，存在以下情况：

- 传播在计算 `x` 时引发的错误。
- 如果计算 `x` 所得到的结果不是数字值，则会引发原因代码为 `"Expression.Error"` 的错误。

一元减运算符

一元减运算符 (`-x`) 针对以下类型的值定义：

x	结果	备注
`type number`	`type number`	否定
`type duration`	`type duration`	否定
`null`	`null`	

对于其他值，将引发原因代码为 `"Expression.Error"` 的错误。

一元减运算符用于更改数字或持续时间的符号。例如：

```
- (1 + 1)       // -2
- - 1           // 1
- - - 1         // -1
- #nan          // #nan
- #infinity     // -#infinity
- #duration(1,0,0,0)  // #duration(-1,0,0,0)
- #duration(0,1,30,0) // #duration(0,-1,-30,0)
```

在计算一元减运算符 `-x` 时，存在以下情况：

- 传播在计算 `x` 时引发的错误。
- 如果表达式为数字，则结果为来自表达式 `x` 的数字值，但符号发生了更改。如果该值为 NaN，则结果也为 NaN。

逻辑求反运算符

逻辑求反运算符 (`not`) 针对以下类型的值定义：

x	结果	备注
`type logical`	`type logical`	否定
`null`	`null`	

此运算符计算指定逻辑值的逻辑 `not` 运算。例如：

```
not true            // false
not false           // true
not (true and true) // false
```

在计算逻辑求反运算符 `not x` 时，存在以下情况：

- 传播在计算 `x` 时引发的错误。
- 计算表达式 x 所生成的值必须是逻辑值，否则将引发原因代码为 `"Expression.Error"` 的错误。如果该值为 `true`，则结果为 `false`。如果操作数为 `false`，则结果为 `true`。

结果为逻辑值。

类型运算符

运算符 `is` 和 `as` 称为类型运算符。

类型兼容性运算符

类型兼容性运算符 `x is y` 针对以下类型的值定义：

X	Y	结果
`type any`	*nullable-primitive-type*	`type logical`

如果 `x` 的先赋类型与 `y` 兼容，则表达式 `x is y` 返回 `true`，如果 `false` 的先赋类型与 `x` 不兼容，则返回 `y`。`y` 必须是可为 null 的基元类型 。

is-expression:
    as-expression
    is-expression `is` *nullable-primitive-type*
nullable-primitive-type:
    `nullable`$_{opt}$ *primitive-type*

`is` 运算符支持的类型兼容性是常规类型兼容性的子集，并使用以下规则定义：

- 如果 `x` 为 null，则当且仅当 y 是可为 null 的类型或 `any` 类型时，它才是兼容的。
- 如果 `x` 不为 null，则仅当 `x` 的基元类型与 `y` 相同时，它才是兼容的。

在计算表达式 `x is y` 时，存在以下情况：

- 传播在计算表达式 `x` 时引发的错误。

类型断言运算符

类型断言运算符 `x as y` 针对以下类型的值定义：

X	Y	结果
`type any`	*nullable-primitive-type*	`type any`

表达式 `x as y` 断言，根据 `is` 运算符，值 `x` 与 `y` 兼容。如果不兼容，则会出现错误。`y` 必须是可为 null 的基元类型 。

as-expression:
    equality-expression
    as-expression `as` *nullable-primitive-type*

表达式 `x as y` 的计算如下：

- 执行类型兼容性检查 `x is y`，如果测试成功，则断言返回未更改的 `x`。
- 如果兼容性检查失败，则会引发原因代码为 `"Expression.Error"` 的错误。

示例：

```
1 as number               // 1
"A" as number             // error
null as nullable number   // null
```

在计算表达式 `x as y` 时，存在以下情况：

- 传播在计算表达式 `x` 时引发的错误。

Let

2020/4/26 •

Let 表达式

Let 表达式可用于从变量中的中间计算中捕获值。

let-expression:
    `let` *variable-list* `in` *expression*
variable-list:
    variable
    variable `,` *variable-list*
variable:
    variable-name `=` *expression*
variable-name:
    identifier

下面的示例显示要计算的中间结果，这些结果存储在变量 `x`、`y` 和 `z` 中，以供在后续计算 `x + y + z` 中使用：

```
let     x = 1 + 1,
        y = 2 + 2,
        z = y + 1
in
        x + y + z
```

此表达式的结果为：

```
11  // (1 + 1) + (2 + 2) + (2 + 2 + 1)
```

在计算 let-expression 中的表达式时，存在以下情况：

- 变量列表中的表达式定义了一个新的作用域，其中包含来自 variable-list 产生式的标识符，并且在计算 variable-list 产生式中的表达式时必须存在 。variable-list 中的表达式可能相互引用 。
- 在计算 let-expression 中的表达式之前，必须先计算 variable-list 中的表达式 。
- 除非使用了 variable-list 中的表达式，否则不能对其进行计算 。
- 传播在计算 let-expression 中的表达式期间引发的错误 。

let 表达式可以看作是隐式记录表达式的语法糖。下面的表达式与上面的表达式等效：

```
[     x = 1 + 1,
      y = 2 + 2,
      z = y + 1,
      result = x + y + z
][result]
```

条件语句

2020/4/23 •

if-expression 基于逻辑输入值的值从两个表达式中进行选择，并仅对所选表达式进行计算 。

if-expression:
    `if` *if-condition* `then` *true-expression* `else` *false-expression*
if-condition:
    expression
true-expression:
    expression
false-expression:
    expression

下面是 if-expressions 的示例：

```
if 2 > 1 then 2 else 1          // 2
if 1 = 1 then "yes" else "no"   // "yes"
```

在计算 if-expression 时，存在以下情况：

- 如果通过计算 if-condition 生成的值不是逻辑值，则会引发原因代码为 `"Expression.Error"` 的错误 。
- 只有当 if-condition 计算为 `true` 值时，才会计算 true-expression 。
- 只有当 if-condition 计算为 `false` 值时，才会计算 false-expression 。
- 如果 if-condition 是 `true`，则 if-expression 的值是 true-expression，如果 if-condition 是 `false`，则值是 false-expression 。
- 传播在计算 if-condition、true-expression 或 falseexpression 时引发的错误 。

函数

2020/4/26 •

函数是一个值，该值表示从一组参数值到单个值的映射 。通过提供一组输入值（参数值）来调用函数，并生成单个输出值（返回值）。

写入函数

使用 function-expression 写入函数：

function-expression:
    `(` *parameter-list*$_{opt}$ `)` *function-return-type*$_{opt}$ `=>` *function-body*
function-body:
    expression
parameter-list:
    fixed-parameter-list
    fixed-parameter-list `,` *optional-parameter-list*
    optional-parameter-list
fixed-parameter-list:
    parameter
    parameter `,` *fixed-parameter-list*
parameter:
    parameter-name parameter-type$_{opt}$
parameter-name:
    identifier
parameter-type:
    assertion
function-return-type:
    assertion
assertion:
    `as` *nullable-primitve-type*
optional-parameter-list:
    optional-parameter
    optional-parameter `,` *optional-parameter-list*
optional-parameter:
    `optional` *parameter*
nullable-primitve-type
    `nullable` opt primitive-type_

下面是一个函数的示例，该函数正好需要两个值 `x` 和 `y`，并生成对这些值应用 `+` 运算符的结果。`x` 和 `y` 是参数，是函数的 formal-parameter-list 的一部分，`x + y` 是函数体：

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

计算 function-expression 的结果是生成函数值（而不是计算 function-body）。作为本文档中的约定，函数值（而不是函数表达式）与 formal-parameter-list 一起显示，但带有省略号（`...`），而不是 function-body 。例如，计算上述函数表达式后，它将显示为以下函数值：

```
(x, y) => ...
```

为函数值定义了以下运算符：

""	""
`x = y`	等于
`x <> y`	不等于

函数值的本机类型是一种自定义函数类型（派生自内部类型 `function`），它列出参数名称并指定所有参数类型和返回类型为 `any`。（有关函数类型的详细信息，请参阅函数类型。）

调用函数

函数的 function-body 是通过使用 invokeexpression 调用函数值来执行的。调用函数值意味着将计算函数值的 function-body 并返回值或引发错误。

invoke-expression:
    primary-expression `(` *argument-list*$_{opt}$ `)`
argument-list:
    expression-list

每次调用函数值时，都会将一组值指定为 argument-list，称为函数的参数。

argument-list 用于将固定数量的参数直接指定为表达式列表。下面的示例定义一个在字段中具有函数值的记录，然后从该记录的另一个字段调用函数：

```
[
    MyFunction = (x, y, z) => x + y + z,
    Result1 = MyFunction(1, 2, 3)           // 6
]
```

调用函数时，以下条件适用：

- 用于计算函数的 function-body 的环境包含与每个参数对应的变量，其名称与参数相同。每个参数的值对应于从 invokeexpression的 argument-list 构造的值（如参数中所定义）。
- 在计算 function-body 之前，计算与函数参数对应的所有表达式。
- 传播在 expression-list 或 functionexpression 中计算表达式时引发的错误。
- 从 argument-list 构造的参数数目必须与函数的形参兼容，否则将引发错误，原因代码为 `"Expression.Error"`。确定兼容性的过程在参数中定义。

参数

formal-parameter-list 中可能存在两种形参：

- 必需参数指示在调用函数时，必须始终指定与形参相对应的实参。必须先在 formal-parameter-list 中指定必需参数。以下示例中的函数定义必需参数 `x` 和 `y`：

```
[
    MyFunction = (x, y) => x + y,

    Result1 = MyFunction(1, 1),     // 2
    Result2 = MyFunction(2, 2)      // 4
]
```

- 可选参数指示在调用函数时，可以指定与形参相对应的实参，但不是必需指定 。如果在调用函数时未指定与可选形参对应的实参，则改为使用 `null` 值。可选参数必须出现在 formal-parameter-list 中的任何必需参数之后 。以下示例中的函数定义固定参数 `x` 和可选参数 `y`：

```
[
    MyFunction = fn(x, optional y) =>
                    if (y = null) x else x + y,
    Result1 = MyFunction(1),        // 1
    Result2 = MyFunction(1, null),  // 1
    Result3 = MyFunction(2, 2),     // 4
]
```

调用函数时指定的参数数目必须与形参列表兼容。函数 `F` 的一组参数 `A` 的兼容性计算如下：

- 让值 N 表示从 argumentlist 构造的参数 `A` 的数目 。例如：

```
MyFunction()             // N = 0
MyFunction(1)            // N = 1
MyFunction(null)         // N = 1
MyFunction(null, 2)      // N = 2
MyFunction(1, 2, 3)      // N = 3
MyFunction(1, 2, null)   // N = 3
MyFunction(1, 2, {3, 4}) // N = 3
```

- 让值 Required 表示 `F` 的固定参数的数目，Optional 表示 `F` 的可选参数的数目 。例如：

```
()              // Required = 0, Optional = 0
(x)             // Required = 1, Optional = 0
(optional x)    // Required = 0, Optional = 1
(x, optional y) // Required = 1, Optional = 1
```

- 如果以下为 true，则参数 `A` 与函数 `F` 兼容：
  - (N >= Fixed) 和 (N <= (Fixed + Optional))
  - 实参类型与 `F` 的相应形参类型兼容
- 如果函数具有已声明的返回类型，则函数 `F` 的主体的结果值与 `F` 的返回类型兼容，前提是以下为 true：
  - 通过使用为函数形参提供的实参计算函数体得到的值的类型与返回类型兼容。
- 如果函数体产生了与函数的返回类型不兼容的值，则会引发错误，原因代码为 `"Expression.Error"` 。

递归函数

若要编写递归函数值，必须使用范围运算符 (`@`) 在其范围内引用函数。例如，下面的记录包含一个定义 `Factorial` 函数的字段和调用该函数的另一个字段：

```
[
    Factorial = (x) =>
                if x = 0 then 1 else x * @Factorial(x - 1),
    Result = Factorial(3)  // 6
]
```

同样，只要需要访问的每个函数都具有名称，就可以编写相互递归函数。在下面的示例中，部分 `Factorial` 函数已重构为第二个 `Factorial2` 函数。

```
[
    Factorial = (x) => if x = 0 then 1 else Factorial2(x),
    Factorial2 = (x) => x * Factorial(x - 1),
    Result = Factorial(3)     // 6
]
```

闭包

函数可以将另一个函数作为值返回。此函数也可以依赖于原始函数的一个或多个参数。在下面的示例中，与字段 `MyFunction` 关联的函数返回一个函数，该函数返回指定给它的参数：

```
[
    MyFunction = (x) => () => x,
    MyFunction1 = MyFunction(1),
    MyFunction2 = MyFunction(2),
    Result = MyFunction1() + MyFunction2()  // 3
]
```

每次调用函数时，将返回维护参数值的新函数值，以便在调用时返回参数值。

函数和环境

除了参数外，function-expression 的 function-body 还可以引用函数初始化时环境中存在的变量 。例如，字段 `MyFunction` 定义的函数访问封闭记录 `A` 的字段 `C`：

```
[
A =
    [
        MyFunction = () => C,
        C = 1
    ],
B = A[MyFunction]()           // 1
]
```

调用 `MyFunction` 时，它将访问变量 `C` 的值，即使它是从不包含变量 `C` 的环境 (`B`) 中调用的。

简化声明

each-expression 是使用名为 `_`（下划线）的单个形参声明非类型化函数的语法简写形式 。

each-expression:
    `each` *each-expression-body*
each-expression-body:
    function-body

简化的声明通常用于提高高阶函数调用的可读性。

例如，以下声明对在语义上是等效的：

```
each _ + 1
(_) => _ + 1
each [A]
(_) => _[A]

Table.SelectRows( aTable, each [Weight] > 12 )
Table.SelectRows( aTable, (_) => _[Weight] > 12 )
```

错误处理

2020/4/23 •

计算 M 表达式的结果将生成以下结果之一：

- 生成单个值。
- 引发错误指示计算表达式的过程不能生成值 。错误包含单个记录值，可用于提供有关导致不完整计算的原因的其他信息。

可从表达式中引发错误，也可从其中处理错误。

引发错误

引发错误的语法如下所示：

error-raising-expression:
`error` *expression*

文本值可用作错误值的简写形式。例如：

```
error "Hello, world" // error with message "Hello, world"
```

完整的错误值是记录，可以使用 `Error.Record` 函数进行构造：

```
error Error.Record("FileNotFound", "File my.txt not found",
     "my.txt")
```

上述表达式等效于：

```
error [
    Reason = "FileNotFound",
    Message = "File my.txt not found",
    Detail = "my.txt"
]
```

引发错误将导致当前表达式计算停止，并将展开表达式计算堆栈，直到发生以下情况之一：

- 已达到记录字段、节成员或 let 变量—统称为：条目 —。该条目被标记为有错误，该错误值将与该条目一起保存，然后传播。对该条目的任何后续访问都将导致引发相同的错误。记录、节或 let 表达式的其他条目不一定会受到影响（除非它们访问之前标记为有错误的条目）。
- 已达到顶级表达式。在这种情况下，计算顶级表达式的结果是一个错误而不是一个值。
- 已达到 `try` 表达式。在这种情况下，将捕获错误并以值的形式返回。

处理错误

error-handling-expression 用于处理错误：

_error-handling-expression:
`try` *protected-expression otherwise-clause*$_{opt}$
protected-expression:

expression

otherwise-clause:

`otherwise` *default-expression*

default-expression:

expression

在不使用 otherwiseclause 计算 error-handling-expression 时，存在以下情况：

- 如果计算 protected-expression 没有导致错误并生成值 x，则由 error-handling-expression 生成的值为以下形式的记录：

```
[ HasErrors = false, Value = x ]
```

- 如果计算 protected-expression 引发错误值 e，则 error-handling-expression 的结果为以下形式的记录：

```
[ HasErrors = true, Error = e ]
```

在使用 otherwiseclause 计算 error-handling-expression 时，存在以下情况：

- 必须在 otherwise-clause 之前计算 protected-expression。
- 当且仅当计算 protectedexpression 引发错误时，才必须计算 otherwise-clause。
- 如果计算 protected-expression 引发错误，则由 error-handling-expression 生成的值为计算 otherwise-clause 的结果 。
- 传播在计算 otherwise-clause 期间引发的错误。

下面的示例演示了在没有引发错误的情况下的 error-handling-expression：

```
let
    x = try "A"
in
    if x[HasError] then x[Error] else x[Value]
// "A"
```

下面的示例演示了引发错误，然后对其进行处理：

```
let
    x = try error "A"
in
    if x[HasError] then x[Error] else x[Value]
// [ Reason = "Expression.Error", Message = "A", Detail = null ]
```

otherwise 子句可用于将 try 表达式处理的错误替换为替代值：

```
try error "A" otherwise 1
// 1
```

如果 otherwise 子句也引发错误，那么整个 try 表达式也会引发错误：

```
try error "A" otherwise error "B"
// error with message "B"
```

记录和 let 初始值设定项中的错误

下面的示例显示了一个记录初始值设定项，其中字段 `A` 引发错误，并且由两个其他字段 `B` 和 `C` 访问。字段 `B` 不处理 `A` 引发的错误，但是 `C` 会处理此错误。最终字段 `D` 不访问 `A`，因此不受 `A` 中的错误影响。

```
[
    A = error "A",
    B = A + 1,
    C = let x =
            try A in
                if not x[HasError] then x[Value]
                else x[Error],
    D = 1 + 1
]
```

计算以上表达式的结果是：

```
[
    A = // error with message "A"
    B = // error with message "A"
    C = "A",
    D = 2
]
```

M 中的错误处理应在接近错误原因的位置执行，以处理延迟字段初始化和延迟闭包计算的影响。下面的示例演示了使用 `try` 表达式处理错误时的失败尝试：

```
let
    f = (x) => [ a = error "bad", b = x ],
    g = try f(42) otherwise 123
in
    g[a]  // error "bad"
```

在此示例中，定义 `g` 用于处理调用 `f` 时引发的错误。但是，此错误由仅在需要时（即从 f 返回记录并通过 `try` 表达式传递后）才运行的字段初始值设定项引发。

未实现的错误

当开发表达式时，作者可能希望省略表达式某些部分的实现，但可能仍希望能够执行表达式。处理这种情况的一种方法是引发未实现部件的错误。例如：

```
(x, y) =>
     if x > y then
         x - y
     else
         error Error.Record("Expression.Error",
            "Not Implemented")
```

省略号符号 (`...`) 可用作 `error` 的快捷方式。

not-implemented-expression:
    `...`

例如，下面的示例等效于前面的示例：

```
(x, y) => if x > y then x - y else ...
```

章节

2020/4/26 •

section-document 是由多个命名表达式组成的 M 程序。

section-document:
    section
section:
    literal-attributes$_{opt}$ `section` *section-name* `;` *section-members*$_{opt}$
section-name:
    identifier
section-members:
    section-member section-members$_{opt}$
section-member:
    literal-attributes$_{opt}$ `shared`$_{opt}$ *section-member-name* `=` *expression* `;`
section-member-name:
    identifier

在 M 中，节是一个组织概念，可以在文档中进行命名和分组相关表达式。每个节都有一个 *section-name*，用于标识该节并限定在该节中声明的 *section-members* 的名称。*sectionmember* 由 *member-name* 和 *expression* 组成。节成员表达式可以直接按成员名称引用同一节中的其他节成员。

以下示例演示包含一个节的 section-document：

```
section Section1;

A = 1;                          //1
B = 2;                          //2
C = A + B;                      //3
```

节成员表达式可以通过 *section-access-expression* 来引用位于其他节中的节成员，该表达式使用包含节的名称来限定节成员名称。

section-access-expression:
    identifier `!` *identifier*

以下示例演示一个包含两个相互引用的节的文档：

```
section Section1;
A = "Hello";                    //"Hello"
B = 1 + Section2!A;             //3

section Section2;
A = 2;                          //2
B = Section1!A & " world!";     /"Hello, world"
```

可以选择性地将节成员声明为 `shared`，这样在引用包含节之外的共享成员时，就不需要使用 *section-access-expression*。只要引用节中没有声明同名的成员，并且其他节中都没有名称相似的共享成员，就可以按非限定成员名称来引用外部节中的共享成员。

以下示例演示了在同一文档的多个节中使用共享成员时的行为：

```
section Section1;
shared A = 1;        // 1

section Section2;
B = A + 2;           // 3 (refers to shared A from Section1)

section Section3;
A = "Hello";         // "Hello"
B = A + " world";    // "Hello world" (refers to local A)
C = Section1!A + 2;  // 3
```

在各个节中定义同名的共享成员，可生成有效的全局环境，但访问该共享成员会引发错误。

```
section Section1;
shared A = 1;

section Section2;
shared A = "Hello";

section Section3;
B = A;    //Error: shared member A has multiple definitions
```

计算 section-document，以下条件适用：

- 每个 *section-name* 在全局环境中必须唯一。
- 在一个节中，每个 *section-member* 必须具有唯一的 *section-member-name*。
- 具有多个定义的共享节成员在访问共享成员时引发错误。
- 在访问节成员之前，无法计算 *section-member* 的表达式组件。
- 在计算 *section-member* 的表达式组件时引发的错误在向外传播之前与该节成员相关联，然后在每次访问该节成员时再次引发错误。

文档链接

一组 M 节文档可以链接到一个不透明的记录值，该值在节文档的每个共享成员中都有一个字段。如果共享成员具有不明确的名称，则会引发错误。

生成的记录值对于执行链接过程的全局环境完全封闭。因此，此类记录是用于从其他(链接的)M 文档集组合 M 文档的合适组件。没有命名冲突的机会。

标准库函数 `Embedded.Value` 可用于检索与重复使用的 M 组件相对应的"嵌入式"记录值。

文档自检

M 通过 `#sections` 和 `#shared` 关键字提供对全局环境的编程访问。

#sections

`#sections` 内部变量将全局环境中的所有节作为记录返回。此记录按节名称设置键，每个值都是按节成员名称进行索引的相应节的记录表示形式。

以下示例显示由两个节组成的文档，以及通过在该文档上下文中计算 `#sections` 内在变量而生成的记录：

```
section Section1;
A = 1;
B = 2;

section Section2;
C = "Hello";
D = "world";

#sections
//[
//  Section1 = [ A = 1, B = 2],
//  Section2 = [ C = "Hello", D = "world" ]
//]
```

计算 `#sections` 时，以下条件适用：

- `#sections` 内部变量保留文档内所有节成员表达式的计算状态。
- `#sections` 内部变量不强制计算任何未计算的节成员。

#shared

`#shared` 内部变量返回一个记录，其中包含当前范围内所有共享节成员的名称和值。

以下示例显示了一个包含两个共享成员的文档，以及通过在该文档上下文中计算 #shared 内在变量而生成的相应记录：

```
section Section1;
shared A = 1;
B = 2;

Section Section2;
C = "Hello";
shared D = "world";

//[
//  A = 1,
//  D = "world"
//]
```

计算 `#shared` 时，以下条件适用：

- `#shared` 内部变量保留文档内所有共享成员表达式的计算状态。
- `#shared` 内部变量不强制计算任何未计算的节成员。

合并语法

2020/4/26 •

词法语法

lexical-unit:
    lexical-elements$_{opt}$
lexical-elements:
    lexical-element lexical-elements$_{opt}$
lexical-element:
    whitespace
    token comment

空格

空格:
    带有 Unicode 类 Zs 的任何字符
    水平制表符字符 (`U+0009`)
    垂直制表符 (`U+000B`)
    换页符 (`U+000C`)
    后跟换行符 (`U+000A`) 的回车符 (`U+000D`)　　*new-line-character*
new-line-character:
    回车符 (`U+000D`)
    换行符 (`U+000A`)
    换行符 (`U+0085`)
    行分隔符 (`U+2028`)
    段落分隔符 (`U+2029`)

评论

comment:
    single-line-comment
    delimited-comment
single-line-comment:
    `//` *single-line-comment-characters*$_{opt}$
single-line-comment-characters:
    single-line-comment-character single-line-comment-characters$_{opt}$
single-line-comment-character:
    除 new-line-character
delimited-comment 之外的任何 Unicode 字符:
    `/*` *delimited-comment-text*$_{opt}$ *asterisks* `/`
delimited-comment-text:
    delimited-comment-section delimited-comment-text$_{opt}$
delimited-comment-section:
    `/`
    asterisks$_{opt}$ *not-slash-or-asterisk*
asterisks:
    `*` *asterisks*$_{opt}$
not-slash-or-asterisk:
    除 `*` 或 `/` 之外的任何 Unicode 字符

令牌

token:
    identifier
    keyword
    literal
    operator-or-punctuator

字符转义序列

character-escape-sequence:
    `#(` *escape-sequence-list* `)`
escape-sequence-list:
    single-escape-sequence
    escape-sequence-list `,` *single-escape-sequence*
single-escape-sequence:
    long-unicode-escape-sequence
    short-unicode-escape-sequence
    control-character-escape-sequence
    escape-escape
long-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
short-unicode-escape-sequence:
    hex-digit hex-digit hex-digit hex-digit
control-character-escape-sequence:
    control-character
control-character:
    `cr`
    `lf`
    `tab`
escape-escape:
    `#`

文字

literal:
    number-literal
    text-literal
    null-literal
number-literal:
    decimal-number-literal
    hexadecimal-number-literal
decimal-digits:
    decimal-digit decimal-digits$_{opt}$
decimal-digit: 以下之一
    `0 1 2 3 4 5 6 7 8 9`
hexadecimal-number-literal:
    `0x` *hex-digits*
    `0X` *hex-digits*
hex-digits:
    hex-digit hex-digits$_{opt}$
hex-digit: 以下之一
    `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`
decimal-number-literal:
    decimal-digits `.` *decimal-digits exponent-part*$_{opt}$

`.` *decimal-digits exponent-part*$_{opt}$

decimal-digits exponent-part$_{opt}$

exponent-part:

`e` *sign*$_{opt}$ *decimal-digits*

`E` *sign*$_{opt}$ *decimal-digits*

sign: one of

`+ -`

text-literal:

`"` *text-literal-characters*$_{opt}$ `"`

text-literal-characters:

text-literal-character text-literal-characters$_{opt}$

text-literal-character:

single-text-character

character-escape-sequence

double-quote-escape-sequence

single-text-character:

除后跟 `(` (`U+0028`) 的 `"` (`U+0022`) 或 `#` (`U+0023`) 外的任何字符

double-quote-escape-sequence:

`""` (`U+0022`, `U+0022`)

null-literal:

`null`

标识符

identifier:

regular-identifier

quoted-identifier

regular-identifier:

available-identifier

available-identifier dot-character regular-identifier

available-identifier:

A *keyword-or-identifier* that is not a *keyword*

keyword-or-identifier:

letter-character

underscore-character

identifier-start-character identifier-part-characters

identifier-start-character:

letter-character

underscore-character

identifier-part-characters:

identifier-part-character identifier-part-characters$_{opt}$

identifier-part-character:

letter-character

decimal-digit-character

underscore-character

connecting-character

combining-character

formatting-character

generalized-identifier:

generalized-identifier-part

generalized-identifier separated only by blanks (`U+0020`) *generalized-identifier-part*

generalized-identifier-part:

generalized-identifier-segment

decimal-digit-character generalized-identifier-segment

generalized-identifier-segment:

keyword-or-identifier

keyword-or-identifier dot-character keyword-or-identifier

dot-character:

`.` (`U+002E`)

underscore-character:

`_` (`U+005F`)

letter-character:

Lu、Ll、Lt、Lm、Lo 或 Nl 类的 Unicode 字符

combining-character:

Mn 或 Mc 类的 Unicode 字符

decimal-digit-character:

Nd 类的 Unicode 字符

connecting-character:

Pc 类的 Unicode 字符

formatting-character:

Cf 类的 Unicode 字符

quoted-identifier:

`#"` *text-literal-characters*$_{opt}$ `"`

关键字和预定义标识符

无法重新定义预定义的标识符和关键字。带引号的标识符可用于处理与预定义标识符或关键字冲突的标识符。

keyword: one of

```
and as each else error false if in is let meta not otherwise or
section shared then true try type #binary #date #datetime
#datetimezone #duration #infinity #nan #sections #shared #table #time
```

运算符和标点符号

operator-or-punctuator: one of

```
, ; = < <= > >= <> + - * / & ( ) [ ] { } @ ? => .. ...
```

语法语法

文档

document:

section-document

expression-document

节文档

section-document:

section

section:

literal-attributes$_{opt}$ `section` *section-name* `;` *section-members*$_{opt}$

section-name:

identifier

section-members:

section-member section-members$_{opt}$

section-member:

literal-attributes$_{opt}$ *shared*$_{opt}$ *section-member-name* `=` *expression* `;`

section-member-name:

identifier

表达式文档

表达式

expression-document:

expression

expression:

logical-or-expression

each-expression

function-expression

let-expression

if-expression

error-raising-expression

error-handling-expression

逻辑表达式

logical-or-expression:

logical-and-expression

logical-and-expression `or` *logical-or-expression*

logical-and-expression:

is-expression

logical-and-expression `and` *is-expression*

Is 表达式

is-expression:

as-expression

is-expression `is` *nullable-primitive-type*

nullable-primitive-type:

`nullable`$_{opt}$ *primitive-type*

As 表达式

as-expression:

equality-expression

as-expression `as` *nullable-primitive-type*

相等表达式

equality-expression:

relational-expression

relational-expression `=` *equality-expression*

relational-expression `<>` *equality-expression*

关系表达式

relational-expression:

additive-expression

additive-expression `<` *relational-expression*

additive-expression `>` *relational-expression*

additive-expression `<=` *relational-expression*

additive-expression `>=` *relational-expression*

算术表达式

additive-expression:

multiplicative-expression

multiplicative-expression `+` *additive-expression*

multiplicative-expression `-` *additive-expression*

multiplicative-expression `&` _additive-expression

multiplicative-expression:

metadata-expression

metadata-expression `*` *multiplicative-expression*
metadata-expression `/` *multiplicative-expression*

元数据表达式

metadata-expression:
unary-expression
unary-expression `meta` *unary-expression*

一元表达式

unary-expression:
type-expression
`+` *unary-expression*
`-` *unary-expression*
`not` *unary-expression*

主表达式

primary-expression:
literal-expression
list-expression
record-expression
identifier-expression
section-access-expression
parenthesized-expression
field-access-expression
item-access-expression
invoke-expression
not-implemented-expression

文本表达式

literal-expression:
literal

标识符表达式

identifier-expression:
identifier-reference
identifier-reference:
exclusive-identifier-reference
inclusive-identifier-reference
exclusive-identifier-reference:
identifier
inclusive-identifier-reference:
`@` *identifier*

Section-access 表达式

section-access-expression:
identifier `!` *identifier*

带圆括号表达式

parenthesized-expression:
`(` *expression* `)`

Not-implemented 表达式

not-implemented-expression:
`...`

调用表达式

invoke-expression:

primary-expression `(` *argument-list*$_{opt}$ `)`

argument-list:

expression

expression `,` *argument-list*

列表表达式

list-expression:

`{` *item-list*$_{opt}$ `}`

item-list:

item

item `,` *item-list*

item:

expression

expression `..` *expression*

记录表达式

record-expression:

`[` *field-list*$_{opt}$ `]`

field-list:

field

field `,` *field-list*

field:

field-name `=` *expression*

field-name:

generalized-identifier

项访问表达式

item-access-expression:

item-selection

optional-item-selection

item-selection:

primary-expression `{` *item-selector* `}`

optional-item-selection:

primary-expression `{` *item-selector* `}?`

字段访问表达式

field-access-expression:

field-selection

implicit-target-field-selection

projection

implicit-target-projection

field-selection:

primary-expression field-selector

field-selector:

required-field-selector

optional-field-selector

required-field-selector:

`[` *field-name* `]`

optional-field-selector:

`[` *field-name* `]?`

field-name:

generalized-identifier

implicit-target-field-selection:

field-selector

projection:
    primary-expression required-projection
    primary-expression optional-projection
required-projection:
    `[` *required-selector-list* `]`
optional-projection:
    `[` *required-selector-list* `] ?`
required-selector-list:
    required-field-selector
    required-field-selector `,` *required-selector-list*
implicit-target-projection:
    projection

函数表达式

function-expression:
    `(` *parameter-list*$_{opt}$ `)` *return-type*$_{opt}$ `=>` *function-body*
function-body:
    expression
parameter-list:
    fixed-parameter-list
    fixed-parameter-list `,` *optional-parameter-list*
    optional-parameter-list
fixed-parameter-list:
    parameter
    parameter `,` *fixed-parameter-list*
parameter:
    parameter-name parameter-type$_{opt}$
parameter-name:
    identifier
parameter-type:
    assertion
return-type:
    assertion
assertion:
    `as` *type*
optional-parameter-list:
    optional-parameter
    optional-parameter `,` *optional-parameter-list*
optional-parameter:
    `optional` *parameter*

每个表达式

each-expression:
    `each` *each-expression-body*
each-expression-body:
    function-body

Let 表达式

let-expression:
    `let` *variable-list* `in` *expression*
variable-list:
    variable
    variable `,` *variable-list*

variable:
    variable-name `=` *expression*
variable-name:
    identifier

If 表达式

if-expression:
    `if` *if-condition* `then` *true-expression* `else` *false-expression*
if-condition:
    expression
true-expression:
    expression
false-expression:
    expression

类型表达式

type-expression:
    primary-expression
    `type` *primary-type*
type:
    parenthesized-expression
    primary-type
primary-type:
    primitive-type
    record-type
    list-type
    function-type
    table-type
    nullable-type
primitive-type: one of
    `any anynonnull binary date datetime datetimezone duration function`
    `list logical none null number record table text type`
record-type:
    `[` *open-record-marker* `]`
    `[` *field-specification-list* `]`
    `[` *field-specification-list* `,` *open-record-marker* `]`
field-specification-list:
    field-specification
    field-specification `,` *field-specification-list*
field-specification:
    `optional`$_{opt}$ *identifier field-type-specification*$_{opt}$
field-type-specification:
    `=` *field-type*
field-type:
    type
open-record-marker:
    `...`
list-type:
    `{` *item-type* `}`
item-type:
    type
function-type:

`function (` *parameter-specification-listopt* `)` *return-type*

parameter-specification-list:

required-parameter-specification-list

required-parameter-specification-list `,` *optional-parameter-specification-list*

optional-parameter-specification-list

required-parameter-specification-list:

required-parameter-specification

required-parameter-specification `,` *required-parameter-specification-list*

required-parameter-specification:

parameter-specification

optional-parameter-specification-list:

optional-parameter-specification

optional-parameter-specification `,` *optional-parameter-specification-list*

optional-parameter-specification:

`optional` *parameter-specification*

parameter-specification:

parameter-name parameter-type

table-type:

`table` *row-type*

row-type:

`[` *field-specification-list* `]`

nullable-type:

`nullable` *type*

错误引发表达式

error-raising-expression:

`error` expression_

错误处理表达式

error-handling-expression:

`try` *protected-expression otherwise-clause*$_{opt}$

protected-expression:

expression

otherwise-clause:

`otherwise` *default-expression*

default-expression:

expression

文本属性

literal-attributes:

record-literal

record-literal:

`[` *literal-field-list*$_{opt}$ `]`

literal-field-list:

literal-field

literal-field `,` *literal-field-list*

literal-field:

field-name `=` *any-literal*

list-literal:

`{` *literal-item-list*$_{opt}$ `}`

literal-item-list:

any-literal `,` *literal-item-list*

any-literal:

record-literal
list-literal
logical-literal
number-literal
text-literal
null-literal

Power Query M 公式语言中的类型

2020/4/14 •

Power Query M 公式语言是一种很有用且富有表现力的数据混合语言。但它确实有一些局限性。例如，没有对类型系统的强有力的执行。在某些情况下，需要进行更严格的验证。幸运的是，M 提供了一个内置库来支持类型，从而使更强的验证成为可能。

开发人员应对类型系统有一个全面的了解，以便通过任何通用性来实现此目的。而且，虽然 Power Query M 语言规范对类型系统进行了说明，但也确实留下了一些惊喜。例如，验证函数实例需要一种比较类型兼容性的方法。

通过更仔细地研究 M 类型系统，可以阐明其中的许多问题，并授权开发人员创建他们需要的解决方案。

谓词演算和朴素集理论的知识应足以理解所使用的表示法。

预备知识

(1) $B := \{ true, false \}$
B 是典型的布尔值集

(2) $N := \{ \text{valid M identifiers} \}$
N 是 M 中所有有效名称的集合。此定义来源于别处。

(3) $P := [B, T]$
P 是函数参数的集合。每个选项都可能是可选的，且具有一个类型。参数名不相关。

(4) $P^n := \bigcup_{0 \le i \le n} [i, P^i]$
P^n 是 n 个函数参数的所有序序列的集合 。

(5) $P^* := \bigcup_{0 \le i \le \infty} P^i$
P^* 是所有可能的函数参数序列的集合，从长度 0 开始 。

(6) $F := [B, N, T]$
F 是所有记录字段的集合。每个字段可能是可选的，具有一个名称和类型 。

(7) $F^n := \prod_{0 \le i \le n} F$
F^n 是 n 个记录字段的所有集合 。

(8) $F^* := \left(\bigcup_{0 \le i \le \infty} F^i \right) \setminus \{ F \mid [b_1, n_1, t_1], [b_2, n_2, t_2] \in F \wedge n_1 = n_2 \}$
F^* 是所有记录字段集（任何长度）的集合，但多个字段具有相同名称的集合除外。

(9) $C := [N, T]$
C 是表的一组列类型。每个列都具有一个名称和类型 。

(10) $C^n \subset \bigcup_{0 \le i \le n} [i, C]$
C^n 是 n 列类型的所有有序序列的集合 。

(11) $C^* := \left(\bigcup_{0 \le i \le \infty} C^i \right) \setminus \{ C^m \mid [a, n_1, t_1], [b, n_2, t_2] \in C^m \wedge n_1 = n_2 \}$
C^* 是列类型的所有组合（任意长度）的集合，但不包括多个列具有相同名称的组合 。

M 类型

(12) $T_F := [P, P^*]$
函数类型由返回类型和由零或多个函数参数组成的有序列表组成 。

(13) T_L := ⟦ T ⟧
列表类型由用大括号括起来的给定类型（称为"项目类型"）表示。由于元语言中使用了大括号，因此本文档使用 ⟦ ⟧ 括号 。

(14) T_R := ⟦ B, F^* ⟧
记录类型有一个标志（指示是否为 open 类型），以及零个或多个无序的记录字段 。

(15) T_R^o := ⟦ *true*, F ⟧

(16) $T_R^•$:= ⟦ *false*, F ⟧
T_R^o 和 $T_R^•$ 分别指 open 和 closed 记录类型的表示法快捷方式 __ 。

(17) T_T := C^*
表类型是零或多个列类型的有序序列，其中不存在名称冲突 。

(18) T_P := { any; none; null; logical; number; time; date; datetime; datetimezone; duration; text; binary; type; list; record; table; function; anynonnull }
基元类型是 M 关键字列表中的一个 。

(19) T_N := { t_n , u ∈ T | t_n = u+null } = nullable t
通过使用"nullable"关键字，任何类型都可以标记为 nullable 。

(20) T := T_F ⋃ T_L ⋃ T_R ⋃ T_T ⋃ T_P ⋃ T_N
所有 M 类型的集合是这六组类型的并集：函数类型、列表类型、记录类型、表类型、基元类型和可空类型 。

函数

需要定义一个函数：NonNullable : T ← T
此函数接受一个类型，并返回一个等效类型，但它不符合 null 值。

标识

需要一些标识来定义一些特殊的情况，这也有助于阐明上述内容。

(21) nullable any = any
(22) nullable anynonnull = any
(23) nullable null = null
(24) nullable none = null
(25) nullable nullable t ∈ T = nullable t
(26) *NonNullable*(nullable t ∈ T) = *NonNullable*(t)
(27) *NonNullable*(any) = anynonnull

类型兼容性

如其他地方所定义的，当且仅当符合第一个类型的所有值也符合第二个类型时，M 类型才可与另一个 M 类型兼容。

此处定义了一个不依赖于符合值的兼容关系，且它基于类型本身的属性。值得注意的是，本文档中定义的这种关系完全等同于最初的语义定义。

"兼容"关系：≤ : B ← T × T
在下面的部分中，小写的 t 将始终表示 M 类型，即 T 的一个元素 。

Φ 将代表 F^* 的一个子集或 C^* 。

(28) t ≤ t
此关系为自反关系 。

(29) t_a ≤ t_b ∧ t_b ≤ t_c → t_a ≤ t_c

此关系为可传递关系 。

(30) none ≤ t ≤ any
M 类型在这个关系上形成一个点阵；没有一个是底部，任何一个都是顶部 。

(31) t_a , t_b ∈ T_N ∧ t_a ≤ t_a → *NonNullable*(t_a) ≤ *NonNullable*(t_b)
如果两种类型是兼容的，那么 NonNullable 的等效项也是兼容的 。

(32) null ≤ t ∈ T_N
基元类型 null 与所有可为 null 的类型兼容 。

(33) t ∉ T_N ≤ anynonnull
所有不可为 null 的类型都与 anynonnull 兼容 。

(34) *NonNullable*(t) ≤ t
NonNullible 类型与可为 null 的等效类型兼容 。

(35) t ∈ T_F → t ≤ function
所有函数类型都与函数兼容 。

(36) t ∈ T_L → t ≤ list
所有列表类型都与列表兼容 。

(37) t ∈ T_R → t ≤ record
所有记录类型都与记录兼容 。

(38) t ∈ T_T → t ≤ table
所有表类型都与表兼容 。

(39) t_a ≤ t_b ↔ ⟦t_a ⟧ ≤ ⟦t_b ⟧
如果项类型兼容，则列表类型可以与其他列表类型兼容，反之亦然 。

(40) t_a ∈ T_F = □ p_a , p^* □, t_b ∈ T_F = □ p_b , p^* □ ∧ p_a ≤ p_b → t_a ≤ t_b
如果返回类型兼容，且参数列表相同，则函数类型与另一种函数类型兼容 。

(41) t_a ∈ T_R^o, t_b ∈ $T_R^\bullet$ → t_a ⊀ t_b
open 记录类型永远与 closed 记录类型不兼容 。

(42) t_a ∈ $T_R^\bullet$ = □*false*, *Φ*□, t_b ∈ T_R^o = □*true*, *Φ*□ → t_a ≤ t_b
closed 记录类型与其他相同的 open 记录类型兼容 。

(43) t_a ∈ T_R^o = □*true*, (*Φ* , □*true*, *n*, any□)□, t_b ∈ T_R^o = □*true*, *Φ*□ → t_a ≤ t_b ∧ t_b ≤ t_a
在比较两个 open 记录类型时，任何类型的可选字段都可能被忽略 。

(44) t_a ∈ T_R = □*b*, (Φ, □*β*, *n*, u_a□)□, t_b ∈ T_R = □*b*, (Φ, □*β*, *n*, u_b□)□ ∧ u_a ≤ u_b → t_a ≤ t_b
如果字段的名称和可选性相同，且所述字段的类型是兼容的，则仅相差一个字段的两个记录类型是兼容的 。

(45) t_a ∈ T_R = □*b*, (Φ, □*false*, *n*, *u*□)□, t_b ∈ T_R = □*b*, (Φ, □*true*, *n*, *u*□)□ → t_a ≤ t_b
带有非可选字段的记录类型与相同的记录类型兼容，但该字段是可选的 。

(46) t_a ∈ T_R^o = □*true*, (*Φ* , □*b*, *n*, *u*□)□, t_b ∈ T_R^o = □*true*, *Φ*□ → t_a ≤ t_b
open 记录类型与另一个少一个字段的 open 记录类型兼容 。

(47) t_a ∈ T_T = (*Φ* , □*i*, □*n*, u_a □□), t_b ∈ T_T = (*Φ* , □*i*, □*n*, u_b □□) ∧ u_a ≤ u_b → t_a ≤ t_b
第一种表类型与第二种表类型兼容，但对于具有不同类型的一列，如果该列的类型兼容，则这两种表类型相同 。

REFERENCES

Microsoft Corporation（2015 年 8 月）
Microsoft Power Query for Excel 公式语言规范 [PDF]

检索自 https://msdn.microsoft.com/library/mt807488.aspx

Microsoft Corporation (n.d.)
Power Query M 函数参考 [网页]
检索自 https://msdn.microsoft.com/library/mt779182.aspx

表达式、值和 let 表达式

2019/12/9 •

Power Query M 公式语言查询由创建糅合查询的公式“表达式”步骤组成 。可以对公式表达式求值(计算)，然后得到一个值。“let”表达式封装一组要计算、分配名称的值，然后在“in”语句后面的后续表达式中使用 。例如，let 表达式可能包含一个“源”变量，该变量等于“Text.Proper()”的值，并以正确的大小写生成文本值 。

Let 表达式

```
let
    Source = Text.Proper("hello world")
in
    Source
```

在上面的示例中，Text.Proper("hello world") 计算为“Hello World”。

下一节介绍语言中的值类型。

基元值

“基元”值是单个部分值，如数字、逻辑、文本或 NULL 。NULL 值可用于指示缺少数据。

“	“
二进制	00 00 00 02 // 点数 (2)
日期	2015/5/23
日期/时间	2015/5/23 凌晨 12:00:00
时区	2015/5/23 凌晨 12:00:00 到上午 08:00
持续时间	15:35:00
逻辑	true 和 false
Null	null
数字	0、1、-1、1.5 和 2.3e-5
Text	“abc”
时间	下午 12:34:12

函数值

“函数”是一个值，当带着参数进行调用时，将生成一个新值 。函数编写的方法是在括号中列出函数的“参数”，后跟“转到”符号 => 和定义函数的表达式 。例如，若要创建一个名为“MyFunction”的函数，该函数具有两个参数，并对 parameter1 和 parameter2 执行计算：

```
let
    MyFunction = (parameter1, parameter2) => (parameter1 + parameter2) / 2
in
    MyFunction

Calling the MyFunction() returns the result:

let
    Source = MyFunction(2, 4)
in
    Source
```

此代码生成值 3。

结构化数据值

M 语言支持以下结构化数据值：

- 列表
- 记录
- 表格
- 其他结构化数据示例

> **NOTE**
> 结构化数据可包含任何 M 值。要查看其他示例，请参阅其他结构化数据示例。

列表

列表是用大括号字符 { } 括起来的从零开始的有序值序列。大括号字符 { } 还用于按索引位置检索列表中的项。请参阅 [List value](#_List_value)。

> **NOTE**
> Power Query M 支持无限的列表大小，但如果列表是以文本形式编写的，则列表的长度是固定的。例如，{1, 2, 3} 的固定长度为 3。

下面是一些“列表”示例 。

{123, true, "A"}	包含数字、逻辑和文本的列表。
{1, 2, 3}	数字列表

{
{1, 2, 3},
{4, 5, 6}
}
	数字列表

{
[CustomerID = 1, Name = "Bob", Phone = "123-4567"],
[CustomerID = 2, Name = "Jim", Phone = "987-6543"]
}
	记录列表

[	[[
{123, true, "A"}{0}	获取列表中第一项的值。此表达式将返回值 123。
{ {1, 2, 3}, {4, 5, 6} }{0}{1}	从第一个列表元素获取第二个项的值。此表达式将返回值 2。

记录

"记录"是一组字段。"字段"是名称/值对，其中名称是在字段的记录中唯一的文本值。记录值的语法允许将名称写成不带引号的形式，这种形式也称为"标识符"。标识符可以采用以下两种形式：

- identifier_name，例如 OrderID。
- #"标识符名称"，如 #"今天的数据为："。

下面是一个记录，其中包含名为"OrderID"、"CustomerID"、"项"和"价格"的字段，其值为 1、1、"钓鱼竿"和 100.00。方括号字符 [ ] 表示记录表达式的开始和结束，用于从记录中获取字段值。下面的示例演示了一个记录以及如何获取项字段值。

下面是一个示例记录：

```
let Source =
        [
              OrderID = 1,
              CustomerID = 1,
              Item = "Fishing rod",
              Price = 100.00
        ]
in Source
```

若要获取项的值，请依照 Source[Item] 这样使用方括号：

```
let Source =
    [
          OrderID = 1,
          CustomerID = 1,
          Item = "Fishing rod",
          Price = 100.00
    ]
in Source[Item] //equals "Fishing rod"
```

表

"表"是一组按命名的列和行组织的值。列类型可以是隐式或显式的。你可以使用 #table 来创建列名列表和行列表。值"表"是"列表"中的列表。大括号 { } 还用于按索引位置从"表"中检索行，（参阅示例 3 - 按索引位置从表中获取行）。

示例 1 - 创建具有隐式列类型的表

```
let
  Source = #table(
    {"OrderID", "CustomerID", "Item", "Price"},
      {
          {1, 1, "Fishing rod", 100.00},
          {2, 1, "1 lb. worms", 5.00}
      })
in
    Source
```

示例 2 - 创建具有显式列类型的表

```
let
    Source = #table(
    type table [OrderID = number, CustomerID = number, Item = text, Price = number],
        {
                {1, 1, "Fishing rod", 100.00},
              {2, 1, "1 lb. worms", 5.00}
        }
    )
in
    Source
```

上述两个示例都创建了一个具有以下形状的表：

ORDERID	CUSTOMERID	▮	▮▮
1	1	钓鱼竿	100.00
2	1	1 磅蠕虫	5.00

示例 3 - 按索引位置从表中获取行

```
let
    Source = #table(
    type table [OrderID = number, CustomerID = number, Item = text, Price = number],
        {
              {1, 1, "Fishing rod", 100.00},
              {2, 1, "1 lb. worms", 5.00}
         }
    )
in
    Source{1}
```

此表达式返回以下记录：

OrderID	2
CustomerID	1
▮	1 磅蠕虫
■	5

其他结构化数据示例

结构化数据可包含任何 M 值。下面是一些示例：

示例 1 - 具有 [(#_Primitive_value_1) 值、[(#_Function_value) 和 [Record](#_Record_value) 的列表

```
let
    Source =
{
   1,
   "Bob",
   DateTime.ToText(DateTime.LocalNow(), "yyyy-MM-dd"),
   [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
}
in
    Source
```

此表达式的计算过程可以直观表示为：

A List containing a Record
1
"Bob"
2015-05-22

OrderID	1
CustomerID	1
Item	"Fishing rod"
Price	100.0

示例 2 - 包含基元值和嵌套记录的记录

```
let
    Source = [CustomerID = 1, Name = "Bob", Phone = "123-4567", Orders =
        {
              [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
            [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0]
        }]
in
    Source
```

此表达式的计算过程可以直观表示为：

A record containing a List of Records	
CustomerID	1
Name	"Bob"
Phone	"123-4567"
Orders	OrderID: 1 CustomerID: 1 Item: "Fishing rod" Price: 100.0
	OrderID: 2 CustomerID: 1 Item: "1 lb. worms" Price: 5.0

> **NOTE**
>
> 尽管许多值都可以按字面形式写成表达式，但值不是表达式。例如，表达式 1 的计算结果为值 1；表达式 1 + 1 的计算结果为值 2。这种区别很细微，但很重要。表达式是计算的方法；值是计算的结果。

If 表达式

“if”表达式根据逻辑条件在两个表达式之间进行选择 。例如：

```
if 2 > 1 then
    2 + 2
else
    1 + 1
```

如果逻辑表达式 (2 > 1) 为 true，则选择第一个表达式 (2 + 2)；如果为 false，则选择第二个表达式 (1 + 1)。将对选定的表达式（在本例中为2 + 2）进行计算，并成为“if”表达式 (4) 的结果 。

注释

2019/12/3 •

可以使用单行注释 `//` 或以 `/*` 开头，以 `*/` 结尾的多行注释将注释添加到代码中。

示例 - 单行注释

```
let
   //Convert to proper case.
    Source = Text.Proper("hello world")
in
    Source
```

示例 - 多行注释

```
/* Capitalize each word in the Item column in the Orders table. Text.Proper
is evaluated for each Item in each table row. */
let
    Orders = Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "fishing rod", Price = 100.0],
          [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
          [OrderID = 3, CustomerID = 2, Item = "fishing net", Price = 25.0]}),
        #"Capitalized Each Word" = Table.TransformColumns(Orders, {"Item", Text.Proper})
in
        #"Capitalized Each Word"
```

计算模型

2019/12/9 •

Power Query M 公式语言的计算模型是根据电子表格中常见的计算模型建模的，这种模型可以基于单元格中公式之间的依赖关系来确定计算顺序。
如果你已经在 Excel 等电子表格中编写过公式，你可能会意识到左侧的公式在计算时会生成右侧的值：

	A
1	=A2 * 2
2	=A3 + 1
3	1

	A
1	4
2	2
3	1

在 M 中，表达式可以按名称引用以前的表达式，并且计算过程会自动确定所引用表达式的计算顺序。

我们将使用一个记录来生成一个与上述电子表格示例等效的表达式。初始化字段的值时，请按字段名称引用记录中的其他字段，如下所示：

```
[
    A1 = A2 * 2,
    A2 = A3 + 1,
    A3 = 1
]
```

上面的表达式计算为以下记录：

```
[
    A1 = 4,
    A2 = 2,
    A3 = 1
]
```

记录可以包含在其他记录中，也可以嵌套在其他记录中 。你可以使用“查找运算符” ([]) 按名称访问记录的字段 。例如，以下记录具有一个名为 Sales 的字段（包含一个记录）和一个名为 Total 的字段（用于访问 Sales 记录的 FirstHalf 和 SecondHalf 字段）：

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = Sales[FirstHalf] + Sales[SecondHalf]
]
```

上面的表达式计算为以下记录：

```
[
    Sales = [ FirstHalf = 1000, SecondHalf = 1100 ],
    Total = 2100
]
```

你可以使用“位置索引运算符” ({ }) 按其数字索引访问列表中的项目 。从列表的开头开始，使用从零开始的索引来引

用列表中的值。例如，索引 0 和 1 用于引用下面列表中的第一和第二项：

```
[
    Sales =
        {
            [
                Year = 2007,
                FirstHalf = 1000,
                SecondHalf = 1100,
                Total = FirstHalf + SecondHalf // equals 2100
            ],
            [
                Year = 2008,
                FirstHalf = 1200,
                SecondHalf = 1300,
                Total = FirstHalf + SecondHalf // equals 2500
            ]
        },
    #"Total Sales" = Sales{0}[Total] + Sales{1}[Total] // equals 4600
]
```

延迟计算和迫切计算

“列表”、“记录”和“表”成员表达式以及 let 表达式（请参阅表达式、值和 let 表达式）使用“延迟计算”进行计算：在需要时进行计算 。所有其他表达式都使用“迫切计算”进行计算：如果在计算过程中遇到它们，则将立即对其进行计算 。考虑这一点的一种好方法是记住评估列表或记录表达式将返回一个列表或记录值，该值知道在请求时（查找或索引运算符）需如何计算其列表项或记录字段。

运算符

2019/12/4 •

Power Query M 公式语言包括可在表达式中使用的一组运算符。将运算符运用于操作数即形成符号表达式 。例如，在表达式 1 + 2 中，数字 1 和 2 是操作数，而运算符是加法运算符 (+)。

运算符的含义可以根据操作数值的类型而变化。该语言具有以下运算符：

加法运算符 (+)

1 + 2	数值相加：3
#time(12,23,0) + #duration(0,0,2,0)	时间运算：#time(12,25,0)

组合运算符 (&)

"A" & "BC"	文本串联："ABC"
{1} & {2, 3}	列表串联：{1, 2, 3}
[a = 1] & [b = 2]	记录合并：[a = 1, b = 2]

M 运算符列表

普通运算符（适用于 NULL、逻辑值、数字、时间、日期、datetime、datetimezone、duration、text、binary）

>	大于
>=	大于或等于
<	小于
<=	小于或等于
=	等于
<>	不等于

逻辑运算符（用于补充普通运算符）

或	条件逻辑或
和	条件逻辑和

""	""
不是	逻辑非

数字运算符（用于补充通用运算符外）

""	""
+	求和
-	差
*	产品
/	商
+x	一元加
-x	否定

文本运算符（ 用于补充通用运算符外）

""	""
&	串联

列表、记录、表运算符

""	""
=	等于
<>	不等于
&	串联

记录查**找运算符**

""	""
[]	按名称访问记录的字段。

列表索引器运算符

""	""
{}	按从零开始的数字索引访问列表中的项。

类型**兼容性和断言运算符**

""	""

ㄩㄩ	ㄩ
是	如果 x 的类型与 y 兼容，则表达式 x is y 返回 true；如果 x 的类型与 y 不兼容，则返回 false。
作为	表达式 x as y 断言，根据 is 运算符，值 x 与 y 兼容。

日期运算符

ㄩㄩ	ㄩㄩㄩ	ㄩㄩㄩㄩ	ㄩ
x + y	time	duration	持续时间的日期偏移
x + y	duration	time	持续时间的日期偏移
x - y	time	duration	求反持续时间的日期偏移
x - y	time	time	日期之间的持续时间
x & y	日期	time	合并的 DateTime

Datetime 运算符

ㄩㄩ	ㄩㄩㄩ	ㄩㄩㄩㄩ	ㄩ
x + y	datetime	duration	持续时间的日期时间偏移
x + y	duration	datetime	持续时间的日期时间偏移
x - y	datetime	duration	求反持续时间的日期时间偏移
x - y	datetime	datetime	日期时间之间的持续时间

Datetimezone 运算符

ㄩㄩ	ㄩㄩㄩ	ㄩㄩㄩㄩ	ㄩ
x + y	datetimezone	duration	持续时间的 Datetimezone 偏移
x + y	duration	datetimezone	持续时间的 Datetimezone 偏移
x - y	datetimezone	duration	求反持续时间的 Datetimezone 偏移
x - y	datetimezone	datetimezone	datetimezone 之间的持续时间

运算符

[illegible]	[illegible]	[illegible]	[illegible]
x + y	datetime	duration	持续时间的日期时间偏移
x + y	duration	datetime	持续时间的日期时间偏移
x + y	duration	duration	持续时间之和
x - y	datetime	duration	求反持续时间的日期时间偏移
x - y	datetime	datetime	日期时间之间的持续时间
x - y	duration	duration	持续时间之差
x * y	duration	数字	持续时间的 N 倍
x * y	数字	duration	持续时间的 N 倍
x / y	duration	数字	持续时间的分数

NOTE

运算符不一定支持某些值的组合。如果表达式在计算时遇到未定义的运算符条件，计算结果将为错误。有关 M 中错误的详细信息，请参见错误

错误示例::

[illegible]	[illegible]
1 + "2"	错误：不支持添加数字和文本

类型转换

2019/12/3 •

Power Query M 公式语言包含用于在类型之间进行转换的公式。下面是 M 中转换公式的汇总。

数字

[illegible]	[illegible]
Number.FromText(text as text), 数字	从文本值返回一个数值。
Number.ToText(number as number), 文本	从数值返回一个文本值。
Number.From(value as any), 数字	从某个值返回一个数字。
Int32.From(value as any), 数字	从给定的值返回一个 32 位整数值。
Int64.From(value as any), 数字	从给定的值返回一个 64 位整数值。
Single.From(value as any), 数字	从给定的值返回单个数值。
Double.From(value as any), 数字	从给定的值返回一个双精度数值。
Decimal.From(value as any), 数字	从给定的值返回一个十进制数值。
Currency.From(value as any), 数字	从给定的值返回货币数值。

文本

[illegible]	[illegible]
Text.From(value as any), 文本	返回数字、日期、时间、日期时间、datetimezone、逻辑、持续时间或二进制值的文本表示形式。

逻辑

[illegible]	[illegible]
Logical.FromText(text as text), 逻辑	从文本值返回逻辑值 true 或 false。
Logical.ToText(logical as logical), 文本	从逻辑值返回文本值。
Logical.From(value as any), 逻辑	从某个值返回逻辑值。

Date、Time、DateTime 和 DateTimeZone

[illegible]	[illegible]
.FromText(text as text), 作为 Date、Time、DateTime 和 DateTimeZone	从一组日期格式和区域性值返回 date、time、datetime 和 datetimezone 值。
.ToText(date, time, dateTime, or dateTimeZone as date, time, datetime, or datetimezone), 作为文本	从 date、time、datetime 或 datetimezone 值返回文本值。
.From(value as any)	从某个值返回 date、time、datetime 或 datetimezone 值。
.ToRecord(date, time, dateTime, or dateTimeZone as date, time, datetime, or datetimezone)	返回包含 date、time、datetime 或 datetimezone 值的各个部分的记录。

元数据

2019/12/3 •

元数据是某个与值相关联的值的相关信息 。元数据被表示为一个记录值，称为元数据记录 。元数据记录的字段可用于存储值的元数据 。每个值都有一个元数据记录。如果尚未指定元数据记录的值，元数据记录则为空（没有字段）。将元数据记录与值关联不会更改值在计算中的行为，除非是显式检查元数据记录。

使用语法值 meta [record] 将元数据记录值与值 x 相关联。例如，以下将带有 Rating 和 Tags 字段的元数据记录与文本值“Mozart”相关联：

```
"Mozart" meta [ Rating = 5,
Tags = {"Classical"} ]
```

可以使用 `Value.Metadata` 函数访问一个值的元数据记录。在下面的示例中，ComposerRating 字段中的表达式访问 Composer 字段中值的元数据记录，然后访问元数据记录的 Rating 字段。

```
[
    Composer = "Mozart" meta [ Rating = 5, Tags = {"Classical"} ],
    ComposerRating = Value.Metadata(Composer)[Rating]   // 5
]
```

当值与构造新值的运算符或函数一起使用时，不保留元数据记录。例如，如果使用 & 运算符来连接两个文本值，那么生成的文本值的元数据为空记录 []。

标准库函数 `Value.RemoveMetadata` 和 `Value.ReplaceMetadata` 可用于从值中删除所有元数据，并替换值的元数据。

错误

2019/12/4 •

Power Query M 公式语言中的 error 指示计算表达式的过程不能产生值 。错误是由运算符和函数遇到错误条件，或使用了错误表达式导致的 。可以使用 try 表达式来处理错误 。引发某一错误时，将指定一个值，此值可用于指示错误发生的原因。

Try 表达式

try 表达式将值和错误转换为一个记录值，此值指示 try 表达式是否处理了错误，以及在处理错误时所提取的是正确值还是错误记录。例如，请考虑以下引发错误，然后立即进行处理的表达式：

```
try error "negative unit count"
```

此表达式计算结果为以下嵌套的记录值，解释之前单价示例中的 `[HasError]`、`[Error]` 和 `[Message]` 字段查找。

错误记录

```
[
    HasError = true,
    Error =
        [
            Reason = "Expression.Error",
            Message = "negative unit count",
            Detail = null
        ]
]
```

常见的情况是使用默认值替换错误。try 表达式可以与一个可选的 otherwise 子句一起使用，从而以紧凑的形式实现：

```
try error "negative unit count" otherwise 42
// equals 42
```

错误示例

```
let Sales =
        [
        ProductName = "Fishing rod",
            Revenue = 2000,
            Units = 1000,
            UnitPrice = if Units = 0 then error "No Units"
                    else Revenue / Units
        ],

    //Get UnitPrice from Sales record
        textUnitPrice = try Number.ToText(Sales[UnitPrice]),
    Label = "Unit Price: " &
        (if textUnitPrice[HasError] then textUnitPrice[Error][Message]
        //Continue expression flow
            else textUnitPrice[Value])
in
    Label
```

上面的示例访问 `Sales[UnitPrice]` 字段并格式化产生结果的值：

```
"Unit Price: 2"
```

如果 Units 字段为零，`UnitPrice` 字段会引发错误，而 try 表达式则会处理此错误。结果值将为：

```
"No Units"
```

Power Query M 函数参考

2019/12/4 •

Power Query M 函数参考包括 700 多个函数中每个函数的文章。你在 docs.microsoft.com 中看到的参考文章是产品内帮助自动生成的。要详细了解函数以及它们在表达式中的使用，请参阅了解 Power Query M 函数。

按类别列出的函数

- 数据访问函数
- 二进制函数
- 组合器函数
- 比较器函数
- 日期函数
- 日期/时间函数
- 日期/时间/时区函数
- 持续时间函数
- 错误处理
- 表达式函数
- 函数值
- 列表函数
- 行函数
- 逻辑函数
- 数字函数
- 记录函数
- 替换器函数
- 拆分器函数
- 表函数
- 文本函数
- 时间函数
- 类型函数
- Uri 函数
- 值函数

了解 Power Query M 函数

2019/12/3 •

在 Power Query M 公式语言中，函数是一组输入值到单个输出值的映射 。要编写一个函数，首先需命名函数参数，然后提供一个表达式来计算函数的结果。函数正文在“转到”(=>) 符号之后。根据需要，类型信息可以包含在参数和函数返回值中。函数在 let 语句的正文中定义和调用 。参数和/或返回值可以是隐式或显式。隐式参数和/或返回值属于 any 类型 。any 类型类似于其他语言中的 object 类型 。M 中的所有类型都派生自 any 类型 。

函数是与数字或文本值类似的值，并且可以像其他任何表达式一样通过嵌入的方式包含在内 。以下示例演示一个函数，该函数是随后将从其他几个变量调用或执行的 Add 变量的值。调用函数时，将指定一组值，这些值会在逻辑上替换函数正文表达式中所需的输入值集。

示例 - 显式参数和返回值

```
let
    AddOne = (x as number) as number => x + 1,
    //additional expression steps
    CalcAddOne = AddOne(5)
in
    CalcAddOne
```

示例 - 隐式参数和返回值

```
let
    Add = (x, y) => x + y,
    AddResults =
        [
            OnePlusOne = Add(1, 1),     // equals 2
            OnePlusTwo = Add(1, 2)      // equals 3
    ]
in
    AddResults
```

查找列表中大于 5 的第一个元素，否则返回 NULL

```
let
    FirstGreaterThan5 = (list) =>
        let
            GreaterThan5 = List.Select(list, (n) => n> 5),
            First = List.First(GreaterThan5)
        in
            First,
    Results =
    [
            Found    = FirstGreaterThan5({3,7,9}),  // equals 7
            NotFound = FirstGreaterThan5({1,3,4})   // equals null
    ]
in
    Results
```

函数可以以递归方式使用。若要以递归方式引用函数，请在标识符前面加上 @。

```
let
    fact = (num) => if num = 0 then 1 else num * @fact (num-1)
in
    fact(5) // equals 120
```

Each 关键字

Each 关键字用于轻松创建简单函数 。“each …”是一个采用 _ 参数的函数签名“(_) => …”的语法糖

与默认应用于 _ 的查询运算符结合使用时，Each 非常有用
例如，each [CustomerID] 与 each _[CustomerID] 相同，后者与 (_) => _[CustomerID] 相同

示例 - 在表行筛选器中使用 each

```
Table.SelectRows(
      Table.FromRecords({
            [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
            [CustomerID = 2, Name = "Jim", Phone = "987-6543"] ,
            [CustomerID = 3, Name = "Paul", Phone = "543-7890"] ,
            [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
      }),
      each [CustomerID] = 2
)[Name]

// equals "Jim"
```

数据访问函数

2020/4/26 •

访问数据

这些函数可访问数据并返回表值。这些函数中的大多数会返回名为"导航表"的表值 。导航表主要由 Power Query 用户界面使用，用于提供针对可能返回的大型分层数据集的导航体验。

AccessControlEntry.ConditionToIdentities	返回此条件将接受的标识列表。
AccessControlKind.Allow	允许访问。
AccessControlKind.Deny	访问被拒绝。
Access.Database	返回 Microsoft Access 数据库的结构表示形式。
ActiveDirectory.Domains	返回与指定域或当前计算机的域（如果未指定任何域）处于同一个林中的 Active Directory 域的列表。
AdobeAnalytics.Cubes	在 Adobe Analytics 中返回报表套件。
AdoDotNet.DataSource	返回 ADO.NET 数据源的架构集合。
AdoDotNet.Query	返回 ADO.NET 数据源的架构集合。
AnalysisServices.Database	从 Analysis Services 数据库中返回多维数据集或表格模型的表。
AnalysisServices.Databases	将 Analysis Services 数据库返回到特定的主机上。
AzureStorage.BlobContents	从 Azure 存储库返回指定 blob 的内容。
AzureStorage.Blobs	返回一个导航表，其中包含在 Azure 存储帐户中找到的所有容器。每一行都有容器名称和指向容器 blob 的一个链接。
AzureStorage.DataLake	返回一个导航表，其中包含在 Azure Data Lake Storage 的指定容器及其子文件夹中找到的文档。
AzureStorage.DataLakeContents	从 Azure Data Lake Storage 文件系统返回指定文件的内容。
AzureStorage.Tables	返回一个导航表，对于从 Azure 存储库中的帐户 URL 处找到的每个表，都作为一行包含在此表中。每一行都包含一个指向 Azure 表的链接。
Cdm.Contents	此函数不可用，因为它需要使用 .NET 4.5。
Csv.Document	使用指定的编码以表的形式返回 CSV 文档的内容。

[	[
CsvStyle.QuoteAfterDelimiter	字段中的引号只有紧跟分隔符之后才有意义。
CsvStyle.QuoteAlways	不论在何处出现，字段中的引号始终有意义。
Cube.AddAndExpandDimensionColumn	将指定维度表 dimensionSelector 合并到多维数据集的 cube 筛选上下文中，并通过展开维度属性的指定集 attributeNames 来更改维度粒度。
Cube.AddMeasureColumn	向多维数据集添加包含名称列的列，其中包含在每行的行上下文中应用的度量值 measureSelector 的结果。
Cube.ApplyParameter	将形参与实参应用于多维数据集后返回多维数据集。
Cube.AttributeMemberId	从成员属性值返回唯一的成员标识符。
Cube.AttributeMemberProperty	返回维度属性 `attribute` 的属性 `propertyName`。
Cube.CollapseAndRemoveColumns	通过折叠映射至指定列 columnNames 的属性，为多维数据集更改筛选上下文的维度粒度。
Cube.Dimensions	返回包含多维数据集中可用维度集的表。
Cube.DisplayFolders	返回一个表嵌套树，表示可在多维数据集中使用的对象(如维度和度量值)的文件夹层次结构。
Cube.MeasureProperties	返回一个表，此表包含在多维数据集中扩展的度量值的可用属性集。
Cube.MeasureProperty	返回度量值的属性。
Cube.Measures	返回包含多维数据集中可用度量值集的表。
Cube.Parameters	返回一个表，此表包含可应用于多维数据集的参数集。
Cube.Properties	返回一个表，此表包含在多维数据集中扩展的维度的可用属性集。
Cube.PropertyKey	返回属性 `property` 的键。
Cube.ReplaceDimensions	
Cube.Transform	在多维数据集上应用列表多维数据集函数 transforms。
DB2.Database	返回 Db2 数据库中可用的 SQL 表和视图的表。
Essbase.Cubes	返回 Essbase 实例中按 Essbase 服务器分组的多维数据集。
Excel.CurrentWorkbook	返回当前 Excel 工作簿中的表。
Excel.Workbook	返回表示给定 excel 工作簿中工作表的表。

Exchange.Contents	返回来自 Microsoft Exchange 帐户的目录。
Facebook.Graph	返回一条包含 Facebook 图形中的内容的记录。
File.Contents	返回位于某一路径处的文件的二进制内容。
Folder.Contents	返回一个表，其中包含在路径处找到的文件和文件夹的属性和内容。
Folder.Files	返回一个表，其中包含在文件夹路径处和子文件夹中找到的每个文件的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。
GoogleAnalytics.Accounts	返回当前凭据的 Google Analytics 帐户。
Hdfs.Contents	返回一个表，其中包含在 Hadoop 文件系统的文件夹 URL 和 {0} 中找到的每个文件夹和文件的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。
Hdfs.Files	返回一个表，其中包含在 Hadoop 文件系统的文件夹 URL、{0} 和子文件夹中找到的每个文件的行。每一行都包含文件的属性以及指向其内容的链接。
HdInsight.Containers	返回一个导航表，其中包含在 HDInsight 帐户中找到的所有容器。每一行都有容器名称和包含其文件的表。
HdInsight.Contents	返回一个导航表，其中包含在 HDInsight 帐户中找到的所有容器。每一行都有容器名称和包含其文件的表。
HdInsight.Files	返回一个表，其中包含在 HDInsight 帐户的容器 URL 处和子文件夹中找到的每个文件夹和文件的行。每一行都包含文件/文件夹的属性以及指向其内容的链接。
Html.Table	返回一个表，其中包含针对所提供的 html 运行指定 CSS 选择器的结果
Identity.From	创建标识。
Identity.IsMemberOf	确定某标识是否为某一标识集合的成员。
IdentityProvider.Default	当前主机的默认标识提供程序。
Informix.Database	返回 SQL 表和视图的一个表，这些表和视图在名为 `database` 数据库实例中服务器 `server` 上的 Informix 数据库中可用。
Json.Document	返回 JSON 文档的内容。这些内容可能会直接作为文本传递给函数，或者可能是由 File.Contents 等函数返回的二进制值。
Json.FromValue	使用 encoding 指定的文本编码生成给定值的 JSON 表示形式。
MySQL.Database	返回一个表，其中包含与指定 MySQL 数据库中的表相关的数据。

OData.Feed	返回 OData serviceUri 提供的 OData 数据源表。
ODataOmitValues.Nulls	允许 OData 服务忽略 NULL 值。
Odbc.DataSource	从连接字符串 `connectionString` 指定的 ODBC 数据源中返回 SQL 表和视图的表。
Odbc.InferOptions	返回尝试推断 ODBC 驱动程序的 SQL 功能的结果。
Odbc.Query	使用给定的连接字符串连接到通用提供程序并返回评估查询的结果。
OleDb.DataSource	从连接字符串指定的 OLE DB 数据源中返回 SQL 表和视图的表。
OleDb.Query	返回在 OLE DB 数据源上运行本机查询的结果。
Oracle.Database	返回一个表，其中包含与指定 Oracle Database 中的表相关的数据。
Parquet.Document	返回 Parquet 文档的内容作为表。
Pdf.Tables	返回 pdf 中找到的任何表。
PostgreSQL.Database	返回一个表，其中包含与指定 PostgreSQL 数据库中的表相关的数据。
RData.FromBinary	从 RData 文件返回数据帧记录。
Salesforce.Data	连接到 Salesforce Objects API 并返回一组可用对象(即帐户)。
Salesforce.Reports	连接到 Salesforce Reports API 并返回一组可用报表。
SapBusinessWarehouse.Cubes	返回 SAP Business Warehouse 系统中按 InfoArea 分组的 InfoCubes 和查询。
SapBusinessWarehouseExecutionMode.DataStream	用于在 SAP Business Warehouse 中执行 MDX 的“DataStream 平展模式”选项。
SapBusinessWarehouseExecutionMode.BasXml	用于在 SAP Business Warehouse 中执行 MDX 的“bXML 平展模式”选项。
SapBusinessWarehouseExecutionMode.BasXmlGzip	用于在 SAP Business Warehouse 中执行 MDX 的“Gzip 压缩 bXML 平展模式”选项。建议用于低延迟或高容量查询。
SapHana.Database	返回 SAP HANA 数据库中的包。
SapHanaDistribution.All	返回 SAP HANA 数据库中的包。
SapHanaDistribution.Connection	SAP HANA 的“Connection”分发选项。
SapHanaDistribution.Off	SAP HANA 的“Off”分发选项。

SapHanaDistribution.Statement	SAP HANA 的“Statement”分发选项。
SapHanaRangeOperator.Equals	SAP HANA 输入参数的“等于”范围运算符。
SapHanaRangeOperator.GreaterThan	SAP HANA 输入参数的“大于”范围运算符。
SapHanaRangeOperator.GreaterThanOrEquals	SAP HANA 输入参数的“大于或等于”范围运算符。
SapHanaRangeOperator.LessThan	SAP HANA 输入参数的“小于”范围运算符。
SapHanaRangeOperator.LessThanOrEquals	SAP HANA 输入参数的“小于或等于”范围运算符。
SapHanaRangeOperator.NotEquals	SAP HANA 输入参数的“不等于”范围运算符。
SharePoint.Contents	返回一个表，其中包含在 SharePoint 站点 URL 处找到的每个文件夹和文档的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。
SharePoint.Files	返回一个表，其中包含在 SharePoint 站点 URL 处和子文件夹中找到的每个文档的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。
SharePoint.Tables	返回一个表，其中包含作为 OData 源的 SharePoint 列表的结果。
Soda.Feed	返回可使用 SODA 2.0 API 访问的 CSV 文件的结果表。URL 必须指向与 SODA 兼容的且以 .csv 扩展名结尾的有效的源。
Sql.Database	返回一个表，其中包含位于 SQL Server 实例数据库上的 SQL 表。
Sql.Databases	返回一个表，其中包含对位于 SQL Server 实例上的数据库的引用。返回导航表。
Sybase.Database	返回一个表，其中包含与指定 Sybase 数据库中的表相关的数据。
Teradata.Database	返回一个表，其中包含与指定 Teradata 数据库中的表相关的数据。
WebAction.Request	创建以下操作：执行后，将使用 HTTP 针对 URL 执行方法请求的结果作为二进制值返回。
Web.BrowserContents	返回由 Web 浏览器查看的指定 URL 的 HTML。
Web.Contents	将从 Web URL 下载的内容作为二进制值返回。
Web.Page	将 HTML 网页的内容作为表返回。
WebMethod.Delete	指定 HTTP 的 DELETE 方法。
WebMethod.Get	指定 HTTP 的 GET 方法。

WebMethod.Head	指定 HTTP 的 HEAD 方法。
WebMethod.Patch	指定 HTTP 的 PATCH 方法。
WebMethod.Post	指定 HTTP 的 POST 方法。
WebMethod.Put	指定 HTTP 的 PUT 方法。
Xml.Document	返回 XML 文档的内容作为层次结构表(记录列表)。
Xml.Tables	返回 XML 文档的内容作为平展表的嵌套集合。

AccessControlEntry.ConditionToIdentities

2019/12/3 •

语法

```
AccessControlEntry.ConditionToIdentities(identityProvider as function, condition as function) as list
```

关于

使用指定的 `identityProvider`，将 `condition` 转换为一个标识列表；对于该列表，`condition` 在以 `identityProvider` 作为标识提供程序的所有授权上下文中都将返回 `true`。如果无法将 `condition` 转换为标识列表，例如，如果 `condition` 参考用户或组标识以外的属性来做出决定，则会引发错误。

请注意，标识列表会按标识出现在 `condition` 中的顺序表示它们，且不对其进行规范化（例如组扩展）。

AccessControlKind.Allow

2019/12/3 •

关于

允许访问。

AccessControlKind.Deny

2019/12/4 •

关于

访问**被拒**绝。

Access.Database

2019/12/3 •

语法

```
Access.Database(database as binary, optional options as nullable record) as table
```

关于

返回 Access 数据库 `database` 的结构表示形式。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 false）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。

例如，记录参数指定为 [option1 = value1, option2 = value2...]。

ActiveDirectory.Domains

2019/12/4 •

语法

```
ActiveDirectory.Domains(optional forestRootDomainName as nullable text) as table
```

关于

返回与指定域或当前计算机的域（如果未指定任何域）处于同一个林中的 Active Directory 域的列表。

AdobeAnalytics.Cubes

2019/12/3 •

语法

```
AdobeAnalytics.Cubes(optional options as nullable record) as table
```

关于

从 Adobe Analytics 返回多维包的表。可以指定可选记录参数 `options` 来控制以下选项：

- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。
- `MaxRetryCount`：在轮询查询的结果时要执行的重试次数。默认值为 120。
- `RetryInterval`：重试尝试之间的持续时间。默认值为 1 秒钟。

AdoDotNet.DataSource

2019/12/3 •

语法

```
AdoDotNet.DataSource(providerName as text, connectionString as any, optional options as nullable record) as table
```

关于

返回 ADO.NET 数据源的架构集合，其中包含提供程序名称 `providerName` 和连接字符串 `connectionString`。 `connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `SqlCompatibleWindowsAuth`：一个逻辑值 (true/false)，用于确定是否为 Windows 身份验证生成与 SQL Server 兼容的连接字符串选项。默认值为 true。
- `TypeMap`

AdoDotNet.Query

2019/12/4 •

语法

```
AdoDotNet.Query(providerName as text, connectionString as any, query as text, optional options as nullable record) as table
```

关于

返回使用 ADO.NET 提供程序 `providerName` 通过连接字符串 `connectionString` 运行 `query` 的结果。 `connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `SqlCompatibleWindowsAuth`：一个逻辑值 (true/false)，用于确定是否为 Windows 身份验证生成与 SQL Server 兼容的连接字符串选项。默认值为 true。

AnalysisServices.Database

2019/12/3 •

语法

```
AnalysisServices.Database(server as text, database as text, optional options as nullable record)
as table
```

关于

从服务器 `server` 上的 Analysis Services 数据库 `database` 中返回多维数据集或表格模型的表。可以指定可选记录参数 `options` 来控制以下选项：

- `Query`：用于检索数据的本机 MDX 查询。
- `TypedMeasureColumns`：一个逻辑值，指示多维模型或表格模型中指定的类型是否将用于添加的度量值列的类型。当设置为 false 时，类型 "number" 将用于所有度量值列。此选项的默认值为 false。
- `Culture`：指定数据区域性的区域性名称。这对应于“区域设置标识符”连接字符串属性。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值与驱动程序相关。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `SubQueries`：一个数字（0、1 或 2），用于设置连接字符串中 "SubQueries" 属性值。这将控制子选定对象或子多维数据集上计算成员的行为。（默认值为 2）。
- `Implementation`

AnalysisServices.Databases

2019/12/4 •

语法

```
AnalysisServices.Databases(server as text, optional options as nullable record) as table
```

关于

返回 Analysis Services 实例 `server` 上的数据库。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `TypedMeasureColumns`：一个逻辑值，指示多维模型或表格模型中指定的类型是否将用于添加的度量值列的类型。当设置为 false 时，类型 "number" 将用于所有度量值列。此选项的默认值为 false。
- `Culture`：指定数据区域性的区域性名称。这对应于“区域设置标识符”连接字符串属性。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值与驱动程序相关。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `SubQueries`：一个数字（0、1 或 2），用于设置连接字符串中 "SubQueries" 属性值。这将控制子选定对象或子多维数据集上计算成员的行为。（默认值为 2）。
- `Implementation`

AzureStorage.BlobContents

2019/12/3 •

语法

```
AzureStorage.BlobContents(url as text, optional options as nullable record) as binary
```

关于

从 Azure 存储库返回 URL 处的 blob 的内容 `url`。可以指定 `options` 来控制以下选项：

- `BlockSize`：等待数据使用者前要读取的字节数。默认值为 4 MB。
- `RequestSize`：要在对服务器的单个 HTTP 请求中尝试读取的字节数。默认值为 4 MB。
- `ConcurrentRequests`：ConcurrentRequests 选项通过指定要并行执行的请求数来支持更快速的数据下载，这是以内存利用率为代价的。所需的内存为 (ConcurrentRequest * RequestSize)。默认值为 16。

AzureStorage.Blobs

2019/12/4 •

语法

```
AzureStorage.Blobs(account as text, optional options as nullable record) as table
```

关于

返回一个导航表，对于从 Azure 存储库中的帐户 URL `account` 处找到的每个容器，都作为一行包含在此表中。每一行都包含一个指向容器 blob 的链接。可以指定 `options` 来控制以下选项：

- `BlockSize`：等待数据使用者前要读取的字节数。默认值为 4 MB。
- `RequestSize`：要在对服务器的单个 HTTP 请求中尝试读取的字节数。默认值为 4 MB。
- `ConcurrentRequests`：ConcurrentRequests 选项通过指定要并行执行的请求数来支持更快速的数据下载，这是以内存利用率为代价的。所需的内存为 (ConcurrentRequest * RequestSize)。默认值为 16。

AzureStorage.DataLake

2019/12/4 •

语法

```
AzureStorage.DataLake(endpoint as text, optional options as nullable record) as table
```

关于

返回一个导航表，其中包含在 Azure Data Lake Storage 文件系统的指定容器及其帐户 URL `endpoint` 处的子文件夹中找到的文档。可以指定 `options` 来控制以下选项：

- `BlockSize`：等待数据使用者前要读取的字节数。默认值为 4 MB。
- `RequestSize`：要在对服务器的单个 HTTP 请求中尝试读取的字节数。默认值为 4 MB。
- `ConcurrentRequests`：ConcurrentRequests 选项通过指定要并行执行的请求数来支持更快速的数据下载，这是以内存利用率为代价的。所需的内存为 (ConcurrentRequest * RequestSize)。默认值为 16。
- `HierarchicalNavigation`：逻辑 (true/false)，用于控制是以类似树的目录视图还是以简单列表的形式返回文件。默认值为 false。

AzureStorage.DataLakeContents

2019/12/3 •

语法

```
AzureStorage.DataLakeContents(url as text, optional options as nullable record) as binary
```

关于

从 Azure Data Lake Storage 文件系统返回 URL 处文件的内容 `url`。可以指定 `options` 来控制以下选项：

- `BlockSize`：等待数据使用者前要读取的字节数。默认值为 4 MB。
- `RequestSize`：要在对服务器的单个 HTTP 请求中尝试读取的字节数。默认值为 4 MB。
- `ConcurrentRequests`：ConcurrentRequests 选项通过指定要并行执行的请求数来支持更快速的数据下载，这是以内存利用率为代价的。所需的内存为 (ConcurrentRequest * RequestSize)。默认值为 16。

AzureStorage.Tables

2019/12/4 •

语法

```
AzureStorage.Tables(account as text) as table
```

关于

返回一个导航表，它对于在 Azure 存储库的帐户 URL `account` 上找到的每个表包含一行。每一行都包含一个指向 Azure 表的链接。

Cdm.Contents

2020/4/30 •

语法

```
Cdm.Contents(table as table) as table
```

关于

此函数不可用，因为它需要使用 .NET 4.5。

Csv.Document

2020/4/30 •

语法

```
Csv.Document(source as any, optional columns as any, optional delimiter as any, optional
extraValues as nullable number, optional encoding as nullable number) as table
```

关于

返回 CSV 文档的内容作为表。

- `columns` 可以为 null、列数、列名称列表、表类型或选项记录。（有关选项记录的详细信息，请参阅下文。）
- `delimiter` 可以是单个字符或字符列表。默认：`","`。
- 请参阅 `ExtraValues.Type`，了解 `extraValues` 的支持值。
- `encoding` 指定文本编码类型。

如果为 `columns` 指定了记录（且 `delimiter`、`extraValues` 和 `encoding` 为 null），则可能会提供以下记录字段：

- `Delimiter`：列分隔符。默认：`","`。
- `Columns`：可以为 null、列数、列名称列表或表类型。如果列数小于在输入中找到的数，则将忽略其他列。如果列数大于在输入中找到的数，则其他列将为 null。如果未指定，则列数将取决于输入中找到的数。
- `Encoding`：文件的文本编码。默认值：65001 (UTF-8)。
- `CsvStyle`：指定如何处理引号。`CsvStyle.QuoteAfterDelimiter`（默认）：字段中的引号只有紧跟分隔符之后才有意义。`CsvStyle.QuoteAlways`：不论在何处出现，字段中的引号始终有意义。
- `QuoteStyle`：指定如何处理带引号的换行符。`QuoteStyle.None`（默认）：所有换行符都视为当前行的末尾，即使它们出现在带引号的值中。`QuoteStyle.Csv`：带引号的换行符视为数据的一部分，而不视为当前行的末尾。

示例 1

处理包含列标题的 CSV 文本。

```
let
    csv = Text.Combine({"OrderID,Item", "1,Fishing rod", "2,1 lb. worms"}, "#(cr)#(lf)")
in
    Table.PromoteHeaders(Csv.Document(csv))
```

ORDERID	ı
1	钓鱼竿
2	1 磅蠕虫

CsvStyle.QuoteAfterDelimiter

2019/12/4 •

语法

```
CsvStyle.QuoteAfterDelimiter
```

关于

字段中的引号只有紧跟分隔符之后才有意义。

CsvStyle.QuoteAlways

2019/12/3 •

语法

```
CsvStyle.QuoteAlways
```

关于

不论在何处出现，字段中的引号始终有意义。

Cube.AddAndExpandDimensionColumn

2019/12/4 •

语法

```
Cube.AddAndExpandDimensionColumn(**cube** as table, **dimensionSelector** as any,
**attributeNames** as list, optional **newColumnNames** as any) as table
```

关于

将指定维度表 `dimensionSelector` 合并到多维数据集 `cube` 的筛选上下文中，并通过展开指定维度属性集 `attributeNames` 来更改维度粒度。将维度属性添加到包含名为 `newColumnNames` 的列的表格视图中，如果未指定，则添加 `attributeNames`。

Cube.AddMeasureColumn

2019/12/3 •

语法

```
Cube.AddMeasureColumn(**cube** as table, **column** as text, **measureSelector** as any) as table
```

关于

向 `cube` 添加包含名称 `column` 的列，其中包含在每行的行上下文中应用的度量值 `measureSelector` 的结果。度量值的应用受维度粒度和切片的变化的影响。执行特定多维数据集操作后，系统会调整度量值。

Cube.ApplyParameter

2019/12/3 •

语法

```
Cube.ApplyParameter(cube as table, parameter as any, optional arguments as nullable list) as table
```

关于

通过将 `parameter` 与 `arguments` 应用到 `cube` 后返回多维数据集。

Cube.AttributeMemberId

2019/12/3 •

语法

```
Cube.AttributeMemberId(attribute as any) as any
```

关于

从成员属性值返回唯一的成员标识符。 `attribute` . 对于任何其他值，返回 NULL。

Cube.AttributeMemberProperty

2019/12/3 •

语法

```
Cube.AttributeMemberProperty(attribute as any, propertyName as text) as any
```

关于

返回维度属性 `attribute` 的属性 `propertyName`。

Cube.CollapseAndRemoveColumns

2019/12/3 •

语法

```
Cube.CollapseAndRemoveColumns(**cube** as table, **columnNames** as list) as table
```

关于

通过折叠映射至指定列 `columnNames` 的属性，更改 `cube` 筛选上下文的维度粒度。还会从多维数据集的表格视图中删除列。

Cube.Dimensions

2019/12/4 •

语法

```
Cube.Dimensions(**cube** as table) as table
```

关于

返回包含 `cube` 中可用维度集的表。每个维度都是包含维度属性集的表，每个维度属性的表示形式是维度表中的列。可以使用 Cube.AddAndExpandDimensionColumn 在多维数据集中展开维度。

Cube.DisplayFolders

2019/12/3 •

语法

```
Cube.DisplayFolders(**cube** as table) as table
```

关于

返回一个表嵌套树，表示可在 `cube` 中使用的对象（如维度和度量值）的文件夹层次结构。

Cube.MeasureProperties

2019/12/3 •

语法

```
Cube.MeasureProperties(cube as table) as table
```

关于

返回一个表，此表包含在多维数据集中扩展的度量值的可用属性集。

Cube.MeasureProperty

2019/12/4 •

语法

```
Cube.MeasureProperty(measure as any, propertyName as text) as any
```

关于

返回度量值 `measure` 的属性 `propertyName`。

Cube.Measures

2019/12/3 •

语法

```
Cube.Measures(**cube** as any) as table
```

关于

返回包含 `cube` 中可用度量值集的表。每个度量值都表示为一个函数。可以使用 Cube.AddMeasureColumn 将度量值应用到多维数据集。

Cube.Parameters

2019/12/4 •

语法

```
Cube.Parameters(cube as table) as table
```

关于

返回一个表，此表包含可应用到 `cube` 的参数集。每个参数都是一个函数，可调用此函数以应用形参及其实参来获取 `cube`。

Cube.Properties

2019/12/4 •

语法

```
Cube.Properties(cube as table) as table
```

关于

返回一个表，此表包含在多维数据集中扩展的维度的可用属性集。

Cube.PropertyKey

2019/12/4 •

语法

```
Cube.PropertyKey(property as any) as any
```

关于

返回属性 `property` 的键。

Cube.ReplaceDimensions

2019/12/3 •

语法

```
Cube.ReplaceDimensions(cube as table, dimensions as table) as table
```

关于

Cube.ReplaceDimensions

Cube.Transform

2019/12/3 •

语法

```
Cube.Transform(cube as table, transforms as list) as table
```

关于

在 `cube` 上应用多维数据集函数列表 `transforms`。

DB2.Database

2019/12/4 •

语法

```
DB2.Database(server as text, database as text, optional options as nullable record) as table
```

关于

返回 SQL 表和视图的表，此表在名为 `database` 的数据库实例中的服务器 `server` 上的 Db2 数据库中可用。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。
- `Implementation`：指定要使用的内部数据库提供程序实现。有效值为："IBM" 和 "Microsoft"。
- `BinaryCodePage`：CCSID（编码字符集标识符）的一个数字，用于将 Db2 FOR BIT 二进制数据解码为字符串。适用于 Implementation = "Microsoft"。设置为 0 将禁用转换（默认值）。设置为 1 将基于数据库编码进行转换。设置其他 CCSID 编号可转换为应用程序编码。
- `PackageCollection`：为包集合指定字符串值（默认值为 "NULLID"）以启用处理 SQL 语句所需的共享包。适用于 Implementation = "Microsoft"。
- `UseDb2ConnectGateway`：指定是否通过 Db2 连接网关进行连接。适用于 Implementation = "Microsoft"。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

Essbase.Cubes

2019/12/4 •

语法

```
Essbase.Cubes(url as text, optional options as nullable record) as table
```

关于

返回一个多维数据集表，其中的数据集由 Essbase 服务器根据 APS 服务器 `url` 上的 Essbase 实例进行分组。可以指定可选记录参数 `options` 来控制以下选项：

- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。

Excel.CurrentWorkbook

2019/12/3 •

语法

```
Excel.CurrentWorkbook() as table
```

关于

返回当前 Excel 工作簿中的表

Excel.Workbook

2019/12/3 •

语法

```
Excel.Workbook(workbook as binary, optional useHeaders as nullable logical, optional delayTypes as nullable logical) as table
```

关于

从 Excel 工作簿返回工作表的记录。

Exchange.Contents

2019/12/4 •

语法

```
Exchange.Contents (optional mailboxAddress as nullable text) as table
```

关于

返回来自 Microsoft Exchange 帐户 `mailboxAddress` 的目录。如果未指定 `mailboxAddress`，则将使用凭据的默认帐户。

Facebook.Graph

2020/2/3 •

IMPORTANT

■ Facebook ▬▬

在 Excel 中从 Facebook 导入和刷新数据将于 2020 年 4 月不再有效。在此之前，你仍可以使用 Facebook"获取和转换"(Power Query) 连接器，但自 2020 年 4 月起，你将无法连接到 Facebook，并会看到错误消息。建议尽快更正或删除现有任何使用 Facebook 连接器的"获取和转换"(Power Query) 查询，以免出现意外结果。

语法

```
Facebook.Graph(url as text) as any
```

关于

返回一条记录，它包含在位于指定 URL `url` 处的 Facebook 图形中找到的一组表。

File.Contents

2019/12/4 •

语法

```
File.Contents(path as text, optional options as nullable record) as binary
```

关于

以二进制形式返回文件 `path` 的内容。

Folder.Contents

2019/12/3 •

语法

```
Folder.Contents(path as text, optional options as nullable record) as table
```

关于

返回一个表，其中包含在文件夹路径 `path` 中找到的每个文件夹和文件的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。

Folder.Files

2019/12/4 •

语法

```
Folder.Files(path as text, optional options as nullable record) as table
```

关于

返回一个表，其中包含在文件夹路径 `path` 和子文件夹中找到的每个文件的行。每一行都包含文件的属性以及指向其内容的链接。

GoogleAnalytics.Accounts

2019/12/3 •

语法

```
GoogleAnalytics.Accounts() as table
```

关于

返回可通过当前凭据进行访问的 Google Analytics 帐户。

Hdfs.Contents

2019/12/4 •

语法

```
Hdfs.Contents(url as text) as table
```

关于

返回一个表，其中包含在 Hadoop 文件系统的文件夹 URL 和 `url` 中找到的每个文件夹和文件的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。

Hdfs.Files

2019/12/4 •

语法

```
Hdfs.Files(url as text) as table
```

关于

返回一个表，其中包含在 Hadoop 文件系统的文件夹 URL、`url` 和子文件夹中找到的每个文件的行。每一行都包含文件的属性以及指向其内容的链接。

HdInsight.Containers

2019/12/3 •

语法

```
HdInsight.Containers(account as text) as table
```

关于

返回一个导航表，对于从 Azure 存储库中的帐户 URL `account` 处找到的每个容器，都作为一行包含在此表中。每一行都包含一个指向容器 blob 的链接。

HdInsight.Contents

2019/12/3 •

语法

```
HdInsight.Contents(account as text) as table
```

关于

返回一个导航表，对于从 Azure 存储库中的帐户 URL `account` 处找到的每个容器，都作为一行包含在此表中。每一行都包含一个指向容器 blob 的链接。

HdInsight.Files

2019/12/3 •

语法

```
HdInsight.Files(account as text, containerName as text) as table
```

关于

返回一个表，对于从 Azure 存储库中的容器 URL `account` 处找到的每个 blob 文件，都作为一行包含在此表中。每一行都包含文件的属性以及指向其内容的链接。

Html.Table

2020/2/28 •

语法

```
Html.Table(html as any, columnNameSelectorPairs as list, optional options as nullable record) as table
```

关于

此函数不可用，因为它需要使用 .NET 4.5。

Identity.From

2019/12/3 •

语法

```
Identity.From(identityProvider as function, value as any) as record
```

关于

创建标识。

Identity.IsMemberOf

2019/12/3 •

语法

```
Identity.IsMemberOf(identity as record, collection as record) as logical
```

关于

确定某标识是否为某一标识集合的成员。

IdentityProvider.Default

2019/12/3 •

语法

```
IdentityProvider.Default() as any
```

关于

当前主机的默认标识提供程序。

Informix.Database

2019/12/3 •

语法

```
Informix.Database(server as text, database as text, optional options as nullable record) as table
```

关于

返回 SQL 表和视图的一个表，这些表和视图在名为 `database` 数据库实例中服务器 `server` 上的 Informix 数据库中可用。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

Json.Document

2019/11/29 •

语法

```
Json.Document(jsonText as any, optional encoding as nullable number) as any
```

关于

返回 JSON 文档的内容。

Json.FromValue

2020/4/30 •

语法

```
Json.FromValue(value as any, optional encoding as nullable number) as binary
```

关于

使用 `value` `encoding` 指定的文本编码生成给定值的 JSON 表示形式。如果省略 `encoding`，则使用 UTF8。值按如下方式表示：

- Null、文本和逻辑值表示为相应的 JSON 类型
- 编号表示为 JSON 中的数字，但 `#infinity`、`-#infinity` 和 `#nan` 都转换为 null
- 列表表示为 JSON 数组
- 记录表示为 JSON 对象
- 表格表示为对象数组
- 日期、时间、日期/时间、日期时间时区和持续时间表示为 ISO-8601 文本
- 二进制值表示为 base-64 编码文本
- 类型和函数产生错误

示例 1

将复杂值转换为 JSON。

```
Text.FromBinary(Json.FromValue([A = {1, true, "3"}, B = #date(2012, 3, 25)]))
```

```
"{""A"":[1,true,""3""],""B"":""2012-03-25""}"
```

MySQL.Database

2019/12/3 •

语法

```
MySQL.Database(server as text, database as text, optional options as nullable record) as table
```

关于

在名为 `database` 的数据库实例中，返回服务器 `server` 上 MySQL 数据库中可用的 SQL 表、视图和存储标量函数的表。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `Encoding`：一个 TextEncoding 值，指定用于对发送到服务器的所有查询进行编码的字符集（默认值为 NULL）。
- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `TreatTinyAsBoolean`：一个逻辑值 (true/false)，用于确定是否将服务器上的 tinyint 列强制设置为逻辑值。默认值为 true。
- `OldGuids`：一个逻辑值 (true/false)，用于设置将 char(36) 列（如果为 false）还是 binary(16) 列（如果为 true）视为 GUID。默认值为 false。
- `ReturnSingleDatabase`：一个逻辑值 (true/false)，用于设置是返回所有数据库的所有表（如果为 false），还是返回指定数据库的表和视图（如果为 true）。默认值为 false。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

OData.Feed

2020/1/16 •

语法

```
OData.Feed(serviceUri as text, optional headers as nullable record, optional options as any) as any
```

关于

从 URI `serviceUri`、标头 `headers` 返回 OData 服务提供的 OData 源表。可以指定一个布尔值来指定使用并发连接还是可选的记录参数 `options` 控制以下选项：

- `Query`：以编程方式将查询参数添加到 URL，无需担心转义。
- `Headers`：将此值指定为记录将为 HTTP 请求提供额外的标头。
- `ExcludedFromCacheKey`：将此值指定为列表会将这些 HTTP 标头键排除在对缓存数据的计算之外。
- `ApiKeyName`：如果目标站点具有 API 密钥的概念，则可以使用此参数来指定必须在 URL 中使用的密钥参数的名称（而不是值）。凭据中提供了实际的密钥值。
- `Timeout`：将此值指定为持续时间将更改 HTTP 请求的超时值。默认值为 600 秒。
- `EnableBatch`：用于设置在超过 MaxUriLength 时是否允许生成 OData $batch 请求的逻辑值 (true/false)，默认为 false。
- `MaxUriLength`：指示发送到 OData 服务的允许的 URI 的最大长度的数字。如果超过且 EnableBatch 为 true，则将向 OData $batch 端点发出请求，否则将失败（默认为 2048）。
- `Concurrent`：逻辑值 (true/false)，设置为 true 时，将同时发出对服务的请求。如果设置为 false，则按顺序发出请求。如果未指定，将由服务的 AsynchronousRequestsSupported 注释确定值。如果服务未指定是否支持 AsynchronousRequestsSupported，则将按顺序发出请求。
- `ODataVersion`：指定要用于此 OData 服务的 OData 协议版本的数字（3 或 4）。如果未指定，则将请求所有支持版本。服务版本将由服务返回的 OData-版本标头确定。
- `FunctionOverloads`：如果逻辑 (true/false) 设置为 true，则函数导入过载将作为单独条目列在导航器中，如果设置为 false，则函数导入过载将作为一个 union 函数列在导航器中。V3 的默认值: false。V4 的默认值: true。
- `MoreColumns`：如果逻辑 (true/false) 设置为 true，则会向每个包含开放类型和多态类型的实体源添加“更多列”列。这将包含基类型中未声明的字段。如果设置为 false，则此字段不存在。默认为 false。
- `IncludeAnnotations`：要包括的命名空间限定术语名或模式的逗号分隔列表，其中“*”作为通配符。默认情况下，不包括任何注释。
- `IncludeMetadataAnnotations`：元数据文档请求中要包括的命名空间限定术语名或模式的逗号分隔列表，其中“*”作为通配符。默认情况下，包括与 IncludeAnnotations 相同的注释。
- `OmitValues`：允许 OData 服务避免在响应中写出某些值。如果已确认，我们将从省略的字段中推断出这些值。选项包括：
- `ODataOmitValues.Nulls`：允许 OData 服务忽略 NULL 值。
- `Implementation`：指定要使用的 OData 连接器实现。有效值为“2.0”或 NULL。

ODataOmitValues.Nulls

2019/12/3 •

关于

允许 OData 服务忽略 NULL 值。

Odbc.DataSource

2019/12/4 •

语法

```
Odbc.DataSource(connectionString as any, optional options as nullable record) as table
```

关于

从连接字符串 `connectionString` 指定的 ODBC 数据源中返回 SQL 表和视图的表。`connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值为 15 秒。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `SqlCompatibleWindowsAuth`：一个逻辑值 (true/false)，用于确定是否为 Windows 身份验证生成与 SQL Server 兼容的连接字符串选项。默认值为 true。

Odbc.InferOptions

2019/12/4 •

语法

```
Odbc.InferOptions(connectionString as any) as record
```

关于

返回后列推断的结果：尝试通过使用 ODBC 的连接字符串 `connectionString` 来推断 SQL 功能。`connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。

Odbc.Query

2019/12/4 •

语法

```
Odbc.Query(connectionString as any, query as text, optional options as nullable record) as table
```

关于

返回在 ODBC 中使用连接字符串 `connectionString` 运行 `query` 的结果。`connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值为 15 秒。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `SqlCompatibleWindowsAuth`：一个逻辑值 (true/false)，用于确定是否为 Windows 身份验证生成与 SQL Server 兼容的连接字符串选项。默认值为 true。

OleDb.DataSource

2019/12/4 •

语法

```
OleDb.DataSource(connectionString as any, optional options as nullable record) as table
```

关于

返回 SQL 表的表并从由连接字符串 `connectionString` 指定的 OLE DB 数据源进行查看。`connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `HierarchicalNavigation`：一个逻辑值(true/false)，用于设置是否查看按架构名称分组的表(默认值为 true)。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `SqlCompatibleWindowsAuth`：一个逻辑值 (true/false)，用于确定是否为 Windows 身份验证生成与 SQL Server 兼容的连接字符串选项。默认值为 True。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

OleDb.Query

2019/12/4 •

语法

```
OleDb.Query(connectionString as any, query as text, optional options as nullable record) as table
```

关于

返回在 OLE DB 中使用连接字符串 `connectionString` 运行 `query` 的结果。 `connectionString` 可以是文本，也可以是属性值对的记录。属性值可以是文本，也可以是数字。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `SqlCompatibleWindowsAuth`：一个逻辑值 (true/false)，用于确定是否为 Windows 身份验证生成与 SQL Server 兼容的连接字符串选项。默认值为 true。

Oracle.Database

2019/12/4 •

语法

```
Oracle.Database(server as text, optional options as nullable record) as table
```

关于

从服务器 `server` 上的 Oracle Database 中返回 SQL 表和视图的表。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

Parquet.Document

2020/2/28 •

语法

```
Parquet.Document(binary as binary) as any
```

关于

此函数不可用，因为它需要使用 .NET 4.5。

Pdf.Tables

2020/2/28 •

语法

```
Pdf.Tables(pdf as binary, optional options as nullable record) as table
```

关于

此函数不可用，因为它需要使用 .NET 4.5。

PostgreSQL.Database

2019/12/4 •

语法

```
PostgreSQL.Database(server as text, database as text, optional options as nullable record) as
table
```

关于

返回包含 SQL 表和视图的表，这些表和视图在名为 `database` 数据库实例中服务器 `server` 上的 PostgreSQL 数据库中可用。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

RData.FromBinary

2019/12/4 •

语法

```
RData.FromBinary(stream as binary) as any
```

关于

从 RData 文件返回数据帧记录。

Salesforce.Data

2020/2/28 •

语法

```
Salesforce.Data(optional loginUrl as any, optional options as nullable record) as table
```

关于

返回凭据中提供的 Salesforce 帐户的对象。将通过提供的环境 `loginUrl` 连接帐户。如果没有提供环境，则帐户连接到生产 (https://login.salesforce.com))。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 false）。
- `ApiVersion`：用于此查询的 Salesforce API 版本。如果未指定，则使用 API 版本 29.0。
- `Timeout`：持续时间，用于控制在放弃向服务器发出的请求前等待的时长。默认值是特定于源的。

Salesforce.Reports

2020/2/28 •

语法

```
Salesforce.Reports(optional loginUrl as nullable text, optional options as nullable record) as
table
```

关于

返回凭据中提供的 Salesforce 帐户的报表。将通过提供的环境 `loginUrl` 连接帐户。如果没有提供环境，则帐户连接到生产 (https://login.salesforce.com))。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `ApiVersion`：用于此查询的 Salesforce API 版本。如果未指定，则使用 API 版本 29.0。
- `Timeout`：持续时间，用于控制在放弃向服务器发出的请求前等待的时长。默认值是特定于源的。

SapBusinessWarehouse.Cubes

2019/12/3 •

语法

```
SapBusinessWarehouse.Cubes(server as text, systemNumberOrSystemId as text, clientId as text,
optional optionsOrLogonGroup as any, optional options as nullable record) as table
```

关于

返回一个表，此表包含在服务器 `server` 处的一个 SAP Business Warehouse 实例中按 InfoArea 分组的 InfoCubes 和查询，系统编号为 `systemNumberOrSystemId`，客户端 ID 为 `clientId`。可以指定可选记录参数 `optionsOrLogonGroup` 来控制选项。

sapbusinesswarehouseexecutionmode.datastream

2019/12/3 •

关于

用于在 SAP Business Warehouse 中执行 MDX 的“DataStream 平展模式”选项。

SapBusinessWarehouseExecutionMode.BasXml

2019/12/4 •

关于

用于在 SAP Business Warehouse 中执行 MDX 的“bXML 平展模式”选项。

SapBusinessWarehouseExecutionMode.BasXmlGzip

2019/12/4 •

关于

用于在 SAP Business Warehouse 中执行 MDX 的“Gzip 压缩 bXML 平展模式”选项。建议用于低延迟或高容量查询。

SapHana.Database

2019/12/4 •

语法

```
SapHana.Database(**server** as text, optional **options** as nullable record) as table
```

关于

从 SAP HANA 数据库 `server` 返回多维包的表。可以指定可选记录参数 `options` 来控制以下选项：

- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `Distribution`：SapHanaDistribution，用于设置连接字符串中 "Distribution" 属性的值。语句路由是在语句执行之前计算正确的分布式系统服务器节点的方法。默认值为 SapHanaDistribution.All。

SapHanaDistribution.All

2019/12/3 •

关于

SAP HANA 的“All”分发选项。

SapHanaDistribution.Connection

2019/12/4 •

关于

SAP HANA 的“Connection”分发选项。

SapHanaDistribution.Off

2019/12/3 •

关于

SAP HANA 的“Off”分发选项。

SapHanaDistribution.Statement

2019/12/4 •

关于

SAP HANA 的“Statement”分发选项。

SapHanaRangeOperator.Equals

2019/12/3 •

关于

SAP HANA 输入参数的“等于”范围运算符。

SapHanaRangeOperator.GreaterThan

2019/12/4 •

关于

SAP HANA 输入参数的“大于”范围运算符。

SapHanaRangeOperator.GreaterThanOrEquals

2019/12/3 •

关于

SAP HANA 输入参数的“大于或等于”范围运算符。

SapHanaRangeOperator.LessThan

2019/12/4 •

关于

SAP HANA 输入参数的“小于”范围运算符。

SapHanaRangeOperator.LessThanOrEquals

2019/12/4 •

关于

SAP HANA 输入参数的“小于或等于”范围运算符。

SapHanaRangeOperator.NotEquals

2019/12/4 •

关于

SAP HANA 输入参数的"不等于"范围运算符。

SharePoint.Contents

2019/12/4 •

语法

```
SharePoint.Contents(url as text, optional options as nullable record) as table
```

关于

返回一个表，该表包含在指定 SharePoint 站点 `url` 中找到的每个文件夹和文档的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。可以指定 `options` 来控制以下选项：

- `ApiVersion`：一个数字（14 或 15）或文本 "Auto"，用于指定此站点要使用的 SharePoint API 版本。如果未指定，则使用 API 版本 14。如果指定 Auto，则将自动发现服务器版本（如果可能），否则版本默认为 14。非英语的 SharePoint 站点至少需要版本 15。

SharePoint.Files

2019/12/4 •

语法

```
SharePoint.Files(url as text, optional options as nullable record) as table
```

关于

返回一个表，其中包含在指定 SharePoint 站点 `url` 和子文件夹中找到的每个文档的行。每一行都包含文件夹或文件的属性以及指向其内容的链接。可以指定 `options` 来控制以下选项：

- `ApiVersion`：一个数字（14 或 15）或文本 "Auto"，用于指定此站点要使用的 SharePoint API 版本。如果未指定，则使用 API 版本 14。如果指定 Auto，则将自动发现服务器版本（如果可能），否则版本默认为 14。非英语的 SharePoint 站点至少需要版本 15。

SharePoint.Tables

2019/12/4 •

语法

```
SharePoint.Tables(url as text, optional options as nullable record) as table
```

关于

返回一个表，该表包含在指定 SharePoint 列表 `url` 处找到的每个列表项的行。每一行都包含该列表的属性。可以指定 `options` 来控制以下选项：

- `ApiVersion`：一个数字（14 或 15）或文本 "Auto"，用于指定此站点要使用的 SharePoint API 版本。如果未指定，则使用 API 版本 14。如果指定 Auto，则将自动发现服务器版本（如果可能），否则版本默认为 14。非英语的 SharePoint 站点至少需要版本 15。

Soda.Feed

2019/12/3 •

语法

```
Soda.Feed(url as text) as table
```

关于

从位于指定 URL `url`（根据 SODA 2.0 API 进行格式化）的内容中返回一个表。URL 必须指向与 SODA 兼容的且以 .csv 扩展名结尾的有效的源。

Sql.Database

2019/12/9 •

语法

```
Sql.Database(server as text, database as text, optional options as nullable record) as table
```

关于

从服务器 `server` 上的 SQL Server 数据库 `database` 中返回 SQL 表、视图和存储函数的表。使用服务器可以选择性指定端口，并用冒号或逗号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `MaxDegreeOfParallelism`：一个数字，用于设置生成的 SQL 查询中 maxdop 查询子句的值""。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。
- `MultiSubnetFailover`：一个逻辑值 (true/false)，用于设置连接字符串中 MultiSubnetFailover 属性的值（默认值为 false）""。
- `UnsafeTypeConversions`
- `ContextInfo`：一个二进制值，用于在运行每个命令之前设置 CONTEXT_INFO。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

Sql.Databases

2020/1/27 •

语法

```
Sql.Databases(server as text, optional options as nullable record) as table
```

关于

返回指定的 SQL Server `server` 上的数据库表。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `MaxDegreeOfParallelism`：一个数字，用于设置生成的 SQL 查询中 maxdop 查询子句的值""。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。
- `MultiSubnetFailover`：一个逻辑值 (true/false)，用于设置连接字符串中 MultiSubnetFailover 属性的值（默认值为 false）""。
- `UnsafeTypeConversions`
- `ContextInfo`：一个二进制值，用于在运行每个命令之前设置 CONTEXT_INFO。

例如，记录参数指定为 [option1 = value1, option2 = value2...]。
不支持将 SQL 查询设置为在服务器上运行。`Sql.Database` 应改为用于运行 SQL 查询。

Sybase.Database

2019/12/4 •

语法

```
Sybase.Database(server as text, database as text, optional options as nullable record) as table
```

关于

返回 SQL 表和视图的表，该表在名为 `database` 的数据库实例中的服务器 `server` 上的 Sybase 数据库中可用。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

Teradata.Database

2019/12/4 •

语法

```
Teradata.Database(server as text, optional options as nullable record) as table
```

关于

从服务器 `server` 上的 Teradata 数据库中返回 SQL 表和视图的表。可以使用服务器选择性指定端口，并用冒号分隔。可以指定可选记录参数 `options` 来控制以下选项：

- `CreateNavigationProperties`：一个逻辑值 (true/false)，用于在返回的值上设置是否生成导航属性（默认值为 true）。
- `NavigationPropertyNameGenerator`：一个函数，用于创建导航属性的名称。
- `Query`：用于检索数据的本机 SQL 查询。如果该查询产生多个结果集，则仅返回第一个。
- `CommandTimeout`：一个时间段，用于控制在取消服务器端查询之前允许该查询运行的时间。默认值为十分钟。
- `ConnectionTimeout`：一个时间段，用于控制在放弃尝试与服务器建立连接之前等待的时间。默认值与驱动程序相关。
- `HierarchicalNavigation`：一个逻辑值 (true/false)，用于设置是否查看按架构名称分组的表（默认值为 false）。

例如，记录参数指定为 [option1 = value1, option2 = value2...] 或 [Query = "select ..."]。

WebAction.Request

2019/12/4 •

语法

```
WebAction.Request(method as text, url as text, optional options as nullable record) as action
```

关于

创建以下操作：执行后，将使用 HTTP 针对 `url` 执行 `method` 请求的结果作为二进制值返回。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `Query`：以编程方式将查询参数添加到 URL，无需担心转义。
- `ApiKeyName`：如果目标站点具有 API 密钥的概念，则可以使用此参数来指定必须在 URL 中使用的密钥参数的名称（而不是值）。凭据中提供了实际的密钥值。
- `Content`：使用 `Content` 字段的值作为 POST 的内容，指定此值会将 Web 请求从 GET 更改为 POST。
- `Headers`：将此值指定为记录将为 HTTP 请求提供额外的标头。
- `Timeout`：将此值指定为持续时间将更改 HTTP 请求的超时值。默认值为 100 秒。
- `IsRetry`：如果将此逻辑值指定为 true，在提取数据时则将忽略缓存中的任何现有响应。
- `ManualStatusHandling`：将此值指定为列表将防止对响应具有以下状态代码之一的 HTTP 请求进行任何内置处理。
- `RelativePath`：如果将此值指定为文本，那么在发出请求前会将此值追加到基 URL。

Web.BrowserContents

2020/2/28 •

语法

```
Web.BrowserContents(url as text, optional options as nullable record) as text
```

关于

此函数不可用，因为它需要使用 .NET 4.5。

Web.Contents

2019/12/4 •

语法

```
Web.Contents(url as text, optional options as nullable record) as binary
```

关于

以二进制形式返回从 `url` 下载的内容。可以提供可选记录参数 `options` 来指定额外的属性。记录可以包含以下字段：

- `Query`：以编程方式将查询参数添加到 URL，无需担心转义。
- `ApiKeyName`：如果目标站点具有 API 密钥的概念，则可以使用此参数来指定必须在 URL 中使用的密钥参数的名称（而不是值）。凭据中提供了实际的密钥值。
- `Content`：使用 `Content` 字段的值作为 POST 的内容，指定此值会将 Web 请求从 GET 更改为 POST。
- `Headers`：将此值指定为记录将为 HTTP 请求提供额外的标头。
- `Timeout`：将此值指定为持续时间将更改 HTTP 请求的超时值。默认值为 100 秒。
- `ExcludedFromCacheKey`：将此值指定为列表会将这些 HTTP 标头键排除在对缓存数据的计算之外。
- `IsRetry`：如果将此逻辑值指定为 true，在提取数据时则将忽略缓存中的任何现有响应。
- `ManualStatusHandling`：将此值指定为列表将防止对响应具有以下状态代码之一的 HTTP 请求进行任何内置处理。
- `RelativePath`：如果将此值指定为文本，那么在发出请求前会将此值追加到基 URL。

Web.Page

2019/12/4 •

语法

```
Web.Page(html as any) as table
```

关于

返回 HTML 文档的内容(分解为其组成结构)，以及删除标记后的完整文档及其文本的表示形式。

WebMethod.Delete

2019/12/3 •

关于

指定 HTTP 的 DELETE 方法。

WebMethod.Get

2019/12/4 •

关于

指定 HTTP 的 GET 方法。

WebMethod.Head

2019/12/3 •

关于

指定 HTTP 的 HEAD 方法。

WebMethod.Patch

2019/12/4 •

关于

指定 HTTP 的 PATCH 方法。

WebMethod.Post

2019/12/3 •

关于

指定 HTTP 的 POST 方法。

WebMethod.Put

2019/12/3 •

关于

指定 HTTP 的 PUT 方法。

Xml.Document

2020/1/16 •

语法

```
Xml.Document(contents as any, optional encoding as nullable number) as table
```

关于

返回 XML 文档的内容作为层次结构表。

Xml.Tables

2019/12/4 •

语法

```
Xml.Tables(contents as any, optional options as nullable record, optional encoding as nullable number) as table
```

关于

返回 XML 文档的内容作为平展表的嵌套集合。

二进制函数

2020/4/26 •

这些函数创建并操纵二进制数据。

二进制格式

读取数值

"	"
BinaryFormat.7BitEncodedSignedInteger	一种二进制格式，读取使用 7 位可变长度编码进行编码的 64 位带符号整数。
BinaryFormat.7BitEncodedUnsignedInteger	一种二进制格式，读取使用 7 位可变长度编码进行编码的 64 位无符号整数。
BinaryFormat.Binary	返回读取二进制值的二进制格式。
BinaryFormat.Byte	读取 8 位无符号整数的二进制格式。
BinaryFormat.Choice	返回一个二进制格式，它基于已读取的值选择下一个二进制格式。
BinaryFormat.Decimal	读取 .NET 16 字节十进制值的二进制格式。
BinaryFormat.Double	读取 8 字节 IEEE 双精度浮点值的二进制格式。
BinaryFormat.Group	返回读取一组项的二进制格式。每个项值的前面都有一个唯一的键值。结果是项值列表。
BinaryFormat.Length	返回一个二进制格式，它限制可读取的数据量。BinaryFormat.List 和BinaryFormat.Binary 均可用于读取，直至数据结束。BinaryFormat.Length 可以用于限制所读取的字节数。
BinaryFormat.List	返回读取项序列并返回一个列表的二进制格式。
BinaryFormat.Null	读取零字节并且返回 NULL 的二进制格式。
BinaryFormat.Record	返回读取记录的二进制格式。记录中的每个字段都可以有不同的二进制格式。
BinaryFormat.SignedInteger16	读取 16 位带符号整数的二进制格式。
BinaryFormat.SignedInteger32	读取 32 位带符号整数的二进制格式。
BinaryFormat.SignedInteger64	读取 64 位带符号整数的二进制格式。
BinaryFormat.Single	读取 4 字节 IEEE 单精度浮点值的二进制格式。

[illegible]	[illegible]
BinaryFormat.Text	返回读取文本值的二进制格式。可选的编码值指定文本的编码。
BinaryFormat.Transform	返回一个二进制格式，该二进制格式将转换由另一个二进制格式读取的值。
BinaryFormat.UnsignedInteger16	读取 16 位无符号整数的二进制格式。
BinaryFormat.UnsignedInteger32	读取 32 位无符号整数的二进制格式。
BinaryFormat.UnsignedInteger64	读取 64 位无符号整数的二进制格式。

[illegible]	[illegible]
BinaryFormat.ByteOrder	以函数指定的字节顺序返回二进制格式。
Table.PartitionValues	返回有关如何对表进行分区的信息。

二进制

[illegible]	[illegible]
Binary.Buffer	缓冲内存中的二进制值。此调用的结果是一个稳定的二进制值，这意味着它将具有确定性的字节长度和顺序。
Binary.Combine	将一系列二进制值合并成单个二进制值。
Binary.Compress	使用给定的压缩类型压缩二进制值。
Binary.Decompress	使用给定压缩类型解压缩二进制值。
Binary.From	返回给定值的二进制值。
Binary.FromList	将一系列数值转换为一个二进制值
Binary.FromText	将来自文本格式的数据解码为二进制值。
Binary.InferContentType	返回一条记录，其中的 Content.Type 字段包含推理出的 MIME 类型。
Binary.Length	返回二进制值的长度。
Binary.ToList	将一个二进制值转换为一系列数值
Binary.ToText	将二进制数据解码为文本格式。
BinaryEncoding.Base64	在要求 base-64 编码时要用作编码类型的常量。
BinaryEncoding.Hex	在要求使用十六进制编码时要用作编码类型的常量。

BinaryOccurrence.Optional	要求该项在输入中不出现或出现一次。
BinaryOccurrence.Repeating	要求该项在输入中不出现或出现多次。
BinaryOccurrence.Required	要求该项在输入中出现一次。
ByteOrder.BigEndian	`BinaryFormat.ByteOrder` 中 `byteOrder` 参数的可能值。最高有效字节首先以 Big Endian 字节顺序显示。
ByteOrder.LittleEndian	`BinaryFormat.ByteOrder` 中 `byteOrder` 参数的可能值。最低有效字节首先以 Little Endian 字节顺序显示。
Compression.Deflate	压缩数据为“Deflate”格式。
Compression.GZip	压缩数据为“GZip”格式。
Occurrence.Optional	要求该项在输入中不出现或出现一次。
Occurrence.Repeating	要求该项在输入中不出现或出现多次。
Occurrence.Required	要求该项在输入中出现一次。
#binary	从数字或文本创建一个二进制值。

Binary.Buffer

2019/12/3 •

语法

```
Binary.Buffer(binary as nullable binary) as nullable binary
```

关于

缓冲内存中的二进制值。此调用的结果是一个稳定的二进制值，这意味着它将具有确定性的字节长度和顺序。

示例 1

创建二进制值的稳定版本。

```
Binary.Buffer(Binary.FromList({0..10}))
```

```
#binary({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
```

Binary.Combine

2019/12/3 •

语法

```
Binary.Combine(binaries as list) as binary
```

关于

将一系列二进制值合并成单个二进制值。

Binary.Compress

2019/12/4 •

语法

```
Binary.Compress(binary as nullable binary, compressionType as number) as nullable binary
```

关于

使用给定的压缩类型压缩二进制值。此调用的结果是输入的压缩副本。压缩类型包括：

- `Compression.GZip`
- `Compression.Deflate`

示例 1

压缩二进制值。

```
Binary.Compress(Binary.FromList(List.Repeat({10}, 1000)), Compression.Deflate)
```

```
#binary({227, 226, 26, 5, 163, 96, 20, 12, 119, 0, 0})
```

Binary.Decompress

2019/12/3 •

语法

```
Binary.Decompress(binary as nullable binary, compressionType as number) as nullable binary
```

关于

使用给定压缩类型解压缩二进制值。此调用的结果是输入内容的解压缩副本。压缩类型包括：

- `Compression.GZip`
- `Compression.Deflate`

示例 1

解压缩二进制值。

```
Binary.Decompress(#binary({115, 103, 200, 7, 194, 20, 134, 36, 134, 74, 134, 84, 6, 0}), Compression.Deflate)
```

```
#binary({71, 0, 111, 0, 111, 0, 100, 0, 98, 0, 121, 0, 101, 0})
```

Binary.From

2019/12/4 •

语法

```
Binary.From(value as any, optional encoding as nullable number) as nullable binary
```

关于

从给定的 `value` 返回 `binary` 值。如果给定的 `value` 为 `null`，则 `Binary.From` 返回 `null`。如果给定的 `value` 为 `binary`，则返回 `value`。可以将以下类型的值转换为 `binary` 值：

- `text`：文本表示形式的 `binary` 值。有关详细信息，请参阅 `Binary.FromText`。

如果 `value` 为任何其他类型，则返回错误。

示例 1

获取 `"1011"` 的 `binary` 值。

```
Binary.From("1011")
```

`Binary.FromText("1011", BinaryEncoding.Base64)`

Binary.FromList

2019/12/4 •

语法

```
Binary.FromList(list as list) as binary
```

关于

将一组数值转换为一个二进制值。

Binary.FromText

2019/12/3 •

语法

```
Binary.FromText(text as nullable text, optional encoding as nullable number) as nullable binary
```

关于

返回将文本值 `text` 转换为二进制的结果（ `number` 列表）。可以将 `encoding` 指定为表示文本值中使用的编码。以下 `BinaryEncoding` 值可以用于 `encoding` 。

- `BinaryEncoding.Base64` : Base64 编码
- `BinaryEncoding.Hex` : 十六进制编码

示例 1

将 `"1011"` 解码为二进制。

```
Binary.FromText("1011")
```

```
Binary.FromText("1011", BinaryEncoding.Base64)
```

示例 2

将 `"1011"` 解码为具有十六进制编码的二进制值。

```
Binary.FromText("1011", BinaryEncoding.Hex)
```

```
Binary.FromText("EBE=", BinaryEncoding.Base64)
```

Binary.InferContentType

2019/12/4 •

语法

```
Binary.InferContentType(source as binary) as record
```

关于

返回一条记录，其中的 Content.Type 字段包含推理出的 MIME 类型。如果推理出的内容类型为 text/*，且检测到编码代码页，则还会返回包含数据流编码的 Content.Encoding 字段。如果推理出的内容类型为 text/csv，且格式是分隔的，则还会返回 Csv.PotentialDelimiter 字段，其中包含用于分析潜在分隔符的表。如果推理出的内容类型为 text/csv，且格式是固定宽度的，则还会返回 Csv.PotentialPositions 字段，其中包含用于分析潜在固定宽度列位置的列表。

Binary.Length

2019/12/4 •

语法

```
Binary.Length(binary as nullable binary) as nullable number
```

关于

返回字符数。

Binary.ToList

2019/12/3 •

语法

```
Binary.ToList(binary as binary) as list
```

关于

将一个二进制值转换为一组数值。

Binary.ToText

2019/12/3 •

语法

```
Binary.ToText(binary as nullable binary, optional encoding as nullable number) as nullable text
```

关于

返回将数字的二进制列表 `binary` 转换为文本值的结果。根据选择，可以指定 `encoding`，以指示要在生成的文本值中使用的编码，以下 `BinaryEncoding` 值可以用于 `encoding`。

- `BinaryEncoding.Base64`：Base64 编码
- `BinaryEncoding.Hex`：十六进制编码

BinaryEncoding.Base64

2019/12/4 •

关于

在要求 base-64 编码时要用作编码类型的常量。

BinaryEncoding.Hex

2019/12/4 •

关于

在要求使用十六进制编码时要用作编码类型的常量。

BinaryFormat.7BitEncodedSignedInteger

2019/12/3 •

语法

```
BinaryFormat.7BitEncodedSignedInteger(binary as binary) as any
```

关于

一种二进制格式，读取使用 7 位可变长度编码进行编码的 64 位带符号整数。

BinaryFormat.7BitEncodedUnsignedInteger

2019/12/4 •

语法

```
BinaryFormat.7BitEncodedUnsignedInteger(binary as binary) as any
```

关于

一种二进制格式，读取使用 7 位可变长度编码进行编码的 64 位无符号整数。

BinaryFormat.Binary

2019/12/4 •

语法

```
BinaryFormat.Binary(optional length as any) as function
```

关于

返回读取二进制值的二进制格式。如果指定了 `length`，则二进制值将包含多个字节。如果未指定 `length`，则二进制值将包含剩余字节。 `length` 可指定为数字或二进制数据之前的长度的二进制格式。

BinaryFormat.Byte

2019/12/4 •

语法

```
BinaryFormat.Byte(binary as binary) as any
```

关于

读取 8 位无符号整数的二进制格式。

BinaryFormat.ByteOrder

2019/12/4 •

语法

```
BinaryFormat.ByteOrder(binaryFormat as function, byteOrder as number) as function
```

关于

以 `binaryFormat` 指定的字节顺序返回二进制格式。默认字节顺序为 `ByteOrder.BigEndian`。

BinaryFormat.Choice

2020/4/30 •

语法

```
BinaryFormat.Choice(binaryFormat as function, chooseFunction as function, optional type as
nullable type, optional combineFunction as nullable function) as function
```

关于

返回一个二进制格式，它基于已读取的值选择下一个二进制格式。此函数按如下步骤生成二进制格式值：

- 使用 `binaryFormat` 参数指定的二进制格式读取一个值。
- 将该值传递到 `chooseFunction` 参数指定的选择函数。
- 该选择函数检查该值并返回第二个二进制格式。
- 使用第二个二进制格式读取第二个值。
- 如果指定了组合函数，则将第一个值和第二个值传递到该组合函数，然后返回生成的值。
- 如果未指定该组合函数，则返回第二个值。
- 返回第二个值。

可选的 `type` 参数指示选择函数将返回的二进制格式的类型。可以指定 `type any`、`type list` 或 `type binary` 中的任何一个。如果未指定 `type` 参数，则使用 `type any`。如果使用 `type list` 或 `type binary`，则系统可能会返回流式 `binary` 或 `list` 值，而不是缓冲后的值，这可以减少读取该格式所需的内存量。

示例 1

读取字节的列表，其中的元素数目由第一个字节确定。

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.List(BinaryFormat.Byte, length)
    )
in
    listFormat(binaryData)
```

3
4

示例 2

读取字节的列表，其中的元素数目由第一个字节确定，并且保留读取的第一个字节。

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.Record([
            length = length,
            list = BinaryFormat.List(BinaryFormat.Byte, length)
        ])
    )
in
    listFormat(binaryData)
```

"	2
"	[列表]

示例 3

使用流式列表读取字节的列表，其中的元素数目由第一个字节确定。

```
let
    binaryData = #binary({2, 3, 4, 5}),
    listFormat = BinaryFormat.Choice(
        BinaryFormat.Byte,
        (length) => BinaryFormat.List(BinaryFormat.Byte, length),
        type list
    )
in
    listFormat(binaryData)
```

3
4

BinaryFormat.Decimal

2019/12/4 •

语法

```
BinaryFormat.Decimal(binary as binary) as any
```

关于

读取 .NET 16 字节十进制值的二进制格式。

BinaryFormat.Double

2019/12/4 •

语法

```
BinaryFormat.Double(binary as binary) as any
```

关于

读取 8 字节 IEEE 双精度浮点值的二进制格式。

BinaryFormat.Group

2020/4/30 •

语法

```
BinaryFormat.Group(binaryFormat as function, group as list, optional extra as nullable function,
optional lastKey as any) as function
```

关于

参数如下所示：

- `binaryFormat` 参数指定键值的二进制格式。
- `group` 参数提供有关已知项组的信息。
- 可选的 `extra` 参数可用于指定函数，该函数会为任何非预期键之后的值返回二进制格式值。如果未指定 `extra` 参数，则会在存在非预期键值时引发错误。

`group` 参数指定项定义的列表。每个项定义都是一个包含 3-5 个值的列表，如下所示：

- 键值。与项对应的键的值。此值在项集中必须唯一。
- 项格式。与项值对应的二进制格式。此值支持各项具有不同的格式。
- 项出现次数。项应在组中出现的次数的 `BinaryOccurrence.Type` 值。必需项不存在时会导致错误。对必需或可选重复项的处理方式类似于非预期键值。
- 默认项值（可选）。如果默认项值出现在项定义列表中，且不为 NULL，则会使用它代替默认值。重复或可选项的默认值为 NULL，重复值的默认值为空列表 { }。
- 项值转换（可选）。如果项值转换函数出现在项定义列表中，且不为 NULL，则会在返回它之前，调用它来转换项值。仅当项出现在输入中时才调用转换函数（永远不会使用默认值调用它）。

示例 1

以下函数假设键值是单个字节，且组中有 4 个预期项，所有项在键之后都跟有一个字节的数据。项在输入中的显示方式如下所示：

- 键 1 为必需项，显示时值为 11。
- 键 2 为重复项，显示两次，值为 22，且结果为值 { 22, 22 }。
- 键 3 为可选项，不显示，且结果为 NULL 值。
- 键 4 为重复项，但不显示，结果为 { } 值。
- 键 5 不属于该组，但显示一次，值为 55。使用键值 5 调用额外的函数，并返回对应该值 (BinaryFormat.Byte) 的格式。已读取并弃用值 55。

```
let
    b = #binary({
        1, 11,
        2, 22,
        2, 22,
        5, 55,
        1, 11
    }),
    f = BinaryFormat.Group(
        BinaryFormat.Byte,
        {
            {1, BinaryFormat.Byte, BinaryOccurrence.Required},
            {2, BinaryFormat.Byte, BinaryOccurrence.Repeating},
            {3, BinaryFormat.Byte, BinaryOccurrence.Optional},
            {4, BinaryFormat.Byte, BinaryOccurrence.Repeating}
        },
        (extra) => BinaryFormat.Byte
    )
in
    f(b)
```

11
[列表]
[列表]

示例 2

下例解释了项值转换和默认项值。带有键 1 的重复项使用 List.Sum 对已读值的列表执行求和。带有键 2 的可选项的默认值为 123，而非 NULL。

```
let
    b = #binary({
        1, 101,
        1, 102
    }),
    f = BinaryFormat.Group(
        BinaryFormat.Byte,
        {
            {1, BinaryFormat.Byte, BinaryOccurrence.Repeating,
              0, (list) => List.Sum(list)},
            {2, BinaryFormat.Byte, BinaryOccurrence.Optional, 123}
        }
    )
in
    f(b)
```

203
123

BinaryFormat.Length

2020/4/30 •

语法

```
BinaryFormat.Length(binaryFormat as function, length as any) as function
```

关于

返回一个二进制格式，它限制可读取的数据量。`BinaryFormat.List` 和 `BinaryFormat.Binary` 均可用于读取至数据结尾。`BinaryFormat.Length` 可以用于限制所读取的字节数。`binaryFormat` 参数指定要限制的二进制格式。`length` 参数指定要读取的字节数。`length` 参数可以是数字值，也可以是二进制格式值，用于指定在要读取的值之前显示的长度值格式。

示例 1

在读取字节列表时，将读取字节数限制为 2。

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.Length(
        BinaryFormat.List(BinaryFormat.Byte),
        2
    )
in
    listFormat(binaryData)
```

```
1
```

```
2
```

示例 2

在读取字节列表时将读取字节数限制为此列表前面的字节值。

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.Length(
        BinaryFormat.List(BinaryFormat.Byte),
        BinaryFormat.Byte
    )
in
    listFormat(binaryData)
```

```
2
```

BinaryFormat.List

2020/4/30 •

语法

```
BinaryFormat.List(binaryFormat as function, optional countOrCondition as any) as function
```

关于

返回可读取项序列的二进制格式并且返回一个 `list`。 `binaryFormat` 参数指定每个项的二进制格式。有三种方法可以确定读取的项数：

- 如果未指定 `countOrCondition`，则将读取二进制格式，直到没有其他项为止。
- 如果 `countOrCondition` 是一个数字，那么二进制格式将读取与该数字相同量的项。
- 如果 `countOrCondition` 是函数，那么将为每个读取的项调用该函数。函数返回 true 则继续，返回 false 则停止读取项。列表中包含最终项。
- 如果 countOrCondition 是二进制格式，则项目计数应位于列表之前，并使用指定的格式读取计数。

示例 1

读取字节，直到到达数据末尾。

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte)
in
    listFormat(binaryData)
```

1
2
3

示例 2

读取两个字节。

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte, 2)
in
    listFormat(binaryData)
```

1
2

示例 3

读**取字**节，**直到字**节值**大于或等于** 2。

```
let
    binaryData = #binary({1, 2, 3}),
    listFormat = BinaryFormat.List(BinaryFormat.Byte, (x) => x < 2)
in
    listFormat(binaryData)
```

1
2

BinaryFormat.Null

2019/12/3 •

语法

```
BinaryFormat.Null(binary as binary) as any
```

关于

读取零字节并且返回 NULL 的二进制格式。

BinaryFormat.Record

2020/4/30 •

语法

```
BinaryFormat.Record(record as record) as function
```

关于

返回读取记录的二进制格式。`record` 参数指定记录的格式。记录中的每个字段都可以有不同的二进制格式。如果某个字段包含的值不是二进制格式的值，则不会读取该字段的任何数据，并且该字段的值将反映到结果中。

示例 1

读取一个记录，该记录包含一个 16 位整数和一个 32 位整数。

```
let
    binaryData = #binary({
        0x00, 0x01,
        0x00, 0x00, 0x00, 0x02
    }),
    recordFormat = BinaryFormat.Record([
        A = BinaryFormat.UnsignedInteger16,
        B = BinaryFormat.UnsignedInteger32
    ])
in
    recordFormat(binaryData)
```

A	1
B	2

BinaryFormat.SignedInteger16

2019/12/4 •

语法

```
BinaryFormat.SignedInteger16(binary as binary) as any
```

关于

读取 16 位带符号整数的二进制格式。

BinaryFormat.SignedInteger32

2019/12/4 •

语法

```
BinaryFormat.SignedInteger32(binary as binary) as any
```

关于

读取 32 位带符号整数的二进制格式。

BinaryFormat.SignedInteger64

2019/12/3 •

语法

```
BinaryFormat.SignedInteger64(binary as binary) as any
```

关于

读取 64 位带符号整数的二进制格式。

BinaryFormat.Single

2019/12/3 •

语法

```
BinaryFormat.Single(binary as binary) as any
```

关于

读取 4 字节 IEEE 单精度浮点值的二进制格式。

BinaryFormat.Text

2020/4/30 •

语法

```
BinaryFormat.Text(length as any, optional encoding as nullable number) as function
```

关于

返回读取文本值的二进制格式。`length` 指定要解码的字节数，或文本前长度的二进制格式。可选的 `encoding` 值指定文本的编码。如果未指定 `encoding`，则根据 Unicode 字节顺序标记确定编码。如果没有字节顺序标记，则使用 `TextEncoding.Utf8`。

示例 1

将两个字节解码为 ASCII 文本。

```
let
    binaryData = #binary({65, 66, 67}),
    textFormat = BinaryFormat.Text(2, TextEncoding.Ascii)
in
    textFormat(binaryData)
```

`"AB"`

示例 2

对 ASCII 文本进行解码，其中，以字节为单位的文本长度作为一个字节出现在文本之前。

```
let
    binaryData = #binary({2, 65, 66}),
    textFormat = BinaryFormat.Text(
        BinaryFormat.Byte,
        TextEncoding.Ascii
    )
in
    textFormat(binaryData)
```

`"AB"`

BinaryFormat.Transform

2020/4/30 •

语法

```
BinaryFormat.Transform(binaryFormat as function, function as function) as function
```

关于

返回一个二进制格式，该二进制格式将转换由另一个二进制格式读取的值。`binaryFormat` 参数指定将用于读取值的二进制格式。将以读取的值调用 `function`，并返回转换后的值。

示例 1

读取一个字节并向其加 1。

```
let
    binaryData = #binary({1}),
    transformFormat = BinaryFormat.Transform(
        BinaryFormat.Byte,
        (x) => x + 1
    )
in
    transformFormat(binaryData)
```

```
2
```

BinaryFormat.UnsignedInteger16

2019/12/4 •

语法

```
BinaryFormat.UnsignedInteger16(binary as binary) as any
```

关于

读取 16 位无符号整数的二进制格式。

BinaryFormat.UnsignedInteger32

2019/12/4 •

语法

```
BinaryFormat.UnsignedInteger32(binary as binary) as any
```

关于

读取 32 位无符号整数的二进制格式。

BinaryFormat.UnsignedInteger64

2019/12/4 •

语法

```
BinaryFormat.UnsignedInteger64(binary as binary) as any
```

关于

读取 64 位无符号整数的二进制格式。

BinaryOccurrence.Optional

2019/12/4 •

关于

要求该项在输入中不出现或出现一次。

BinaryOccurrence.Repeating

2019/12/3 •

关于

要求该项**在**输**入中不出**现**或出**现**多次**。

BinaryOccurrence.Required

2019/12/3 •

关于

要求该项在输入中出现一次。

ByteOrder.BigEndian

2019/12/3 •

关于

`BinaryFormat.ByteOrder` 中 `byteOrder` 参数的可能值。最高有效字节首先以 Big Endian 字节顺序显示。

ByteOrder.LittleEndian

2019/12/3 •

关于

`BinaryFormat.ByteOrder` 中 `byteOrder` 参数的可能值。最低有效字节首先以 Little Endian 字节顺序显示。

Compression.Deflate

2019/12/3 •

关于

压缩数据为“Deflate”格式。

Compression.GZip

2019/12/4 •

关于

压缩**数据**为“GZip”**格式**。

Occurrence.Optional

2019/12/4 •

关于

要求该项**在**输**入**中**不**出现**或**出现**一次**。

Occurrence.Repeating

2019/12/3 •

关于

要求该项**在**输**入**中**不**出现**或**出现**多次**。

Occurrence.Required

2019/12/4 •

关于

要求该项在输入中出现一次。

#binary

2019/12/4 •

语法

```
#binary(value as any) as any
```

关于

从数字列表或一个 Base 64 编码文本值创建一个二进制值。

示例 1

从数字列表创建一个二进制值。

```
#binary({0x30, 0x31, 0x32})
```

`Text.ToBinary("012")`

示例 2

从一个 Base 64 编码文本值创建一个二进制值。

```
#binary("1011")
```

`Binary.FromText("1011", BinaryEncoding.Base64)`

合并器函数

2020/4/26 •

这些函数由合并值的其他库函数使用。例如，`Table.ToList` 和 `Table.CombineColumns` 将组合程序函数应用于表中的每一行，以为每一行生成单个值。

组合器

"	"
Combiner.CombineTextByDelimiter	返回一个函数，它使用指定的分隔符将文本列表合并成单个文本。
Combiner.CombineTextByEachDelimiter	返回一个函数，它按顺序使用每个指定的分隔符将文本列表合并成单个文本。
Combiner.CombineTextByLengths	返回一个函数，它使用指定的长度将文本列表合并成单个文本。
Combiner.CombineTextByPositions	返回一个函数，它使用指定的位置将文本列表合并成单个文本。
Combiner.CombineTextByRanges	返回一个函数，它使用指定的位置和长度将文本列表合并成单个文本。

Combiner.CombineTextByDelimiter

2019/12/3 •

语法

```
Combiner.CombineTextByDelimiter(delimiter as text, optional quoteStyle as nullable number) as
function
```

关于

返回一个函数，它使用指定的分隔符将文本列表合并成单个文本。

Combiner.CombineTextByEachDelimiter

2019/12/3 •

语法

```
Combiner.CombineTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number) as function
```

关于

返回一个函数，它按顺序使用每个指定的分隔符将文本列表合并成单个文本。

Combiner.CombineTextByLengths

2019/12/4 •

语法

```
Combiner.CombineTextByLengths(lengths as list, optional template as nullable text) as function
```

关于

返回一个函数，它使用指定的长度将文本列表合并成单个文本。

Combiner.CombineTextByPositions

2019/12/3 •

语法

```
Combiner.CombineTextByPositions(positions as list, optional template as nullable text) as function
```

关于

返回一个函数，它使用指定的位置将文本列表合并成单个文本。

Combiner.CombineTextByRanges

2019/12/4 •

语法

```
Combiner.CombineTextByRanges(ranges as list, optional template as nullable text) as function
```

关于

返回一个函数，它使用指定的位置和长度将文本列表合并成单个文本。

比较器函数

2020/4/26 •

这些函数测试等同性并确定顺序。

比较器

"	"
Comparer.Equals	在对两个给定值进行同等性检查的基础上返回一个逻辑值。
Comparer.FromCulture	返回一个比较器函数，得出用于比较的区分大小写的区域性和逻辑值。ignoreCase 的默认值为 false。区域性值是 .NET Framework 中所使用的区域设置的已知文本表示形式。
Comparer.Ordinal	返回一个使用 Ordinal 规则的比较器函数以比较值。
Comparer.OrdinalIgnoreCase	返回使用 Ordinal 规则来比较提供的值 x 和 y 的不区分大小写的比较器函数。
Culture.Current	返回系统的当前区域性。

Comparer.Equals

2019/12/4 •

语法

```
Comparer.Equals(comparer as function, x as any, y as any) as logical
```

关于

使用提供的 `comparer`，在对两个给定值（`x` 和 `y`）进行同等性检查的基础上返回一个 `logical` 值。

`comparer` 是用于控制比较的 `Comparer`。比较器可用于提供不区分大小写或区分区域性和区域设置的比较。
以下内置比较器支持公式语言：

- `Comparer.Ordinal`：用于执行精确的序号比较
- `Comparer.OrdinalIgnoreCase`：用于执行精确的、不区分大小写的序号比较
- `Comparer.FromCulture`：用于执行区分区域性的比较

示例 1

使用"en-US"区域设置比较"1"和"A"以确定这些值是否相等。

```
Comparer.Equals(Comparer.FromCulture("en-us"), "1", "A")
```

`false`

Comparer.FromCulture

2019/12/3 •

语法

```
Comparer.FromCulture(culture as text, optional ignoreCase as nullable logical) as function
```

关于

返回一个比较器函数，得出用于比较的区分大小写的 `culture` 和逻辑值 `ignoreCase`。 `ignoreCase` 的默认值为 false。区域性值是 .NET framework 中所使用的区域设置的已知文本表示形式。

示例 1

使用“en-US”区域设置比较“a”和“A”以确定这些值是否相等。

```
Comparer.FromCulture("en-us")("a", "A")
```

```
-1
```

示例 2

使用忽略大小写的“en-US”区域设置比较“a”和“A”以确定这些值是否相等。

```
Comparer.FromCulture("en-us", true)("a", "A")
```

```
0
```

Comparer.Ordinal

2019/12/3 •

语法

```
Comparer.Ordinal(x as any, y as any) as number
```

关于

返回使用 Ordinal 规则来比较提供的值 `x` 和 `y` 的比较器函数。

示例 1

使用 Ordinal 规则，比较 encyclopædia 和 encyclopaedia 是否等效。请注意，这些在使用 `Comparer.FromCulture("en-us")` 时是等效的。

```
Comparer.Equals(Comparer.Ordinal, "encyclopædia", "encyclopaedia")
```

```
false
```

Comparer.OrdinalIgnoreCase

2019/12/4 •

语法

```
Comparer.OrdinalIgnoreCase(x as any, y as any) as number
```

关于

返回使用 Ordinal 规则来比较提供的值 `x` 和 `y` 的不区分大小写的比较器函数。

示例

使用不区分大小写的 Ordinal 规则来比较“Abc”与“abc”。请注意，使用 `Comparer.Ordinal` 时，“Abc”小于“abc”。

```
Comparer.OrdinalIgnoreCase("Abc", "abc")
```

```
0
```

Culture.Current

2019/12/4 •

关于

返回应用程序的当前区域性的名称。

日期函数

2020/4/26 •

这些函数创建并操纵 date、datetime 和 datetimezone 值的日期部分。

日期

"	"
Date.AddDays	返回一个 Date/DateTime/DateTimeZone 值，其中天数部分是按提供的天数递增的。它还会根据需要处理值的月份和年份部分的递增。
Date.AddMonths	返回一个 DateTime 值，其中月份部分是按 n 个月递增的。
Date.AddQuarters	返回一个 Date/DateTime/DateTimeZone 值，此值是按提供的季度数递增的。每个季度定义为为期三个月的一段时间。它还会根据需要处理值的年份部分的递增。
Date.AddWeeks	返回一个 Date/DateTime/DateTimeZone 值，此值是按提供的周数递增的。每周定义为为期七天的一段时间。它还会根据需要处理值的月份和年份部分的递增。
Date.AddYears	返回一个 DateTime 值，其中年份部分是按 n 年递增的。
Date.Day	返回 DateTime 值的日。
Date.DayOfWeek	返回数字（介于 0 到 6 之间）以指明提供的值是星期几。
Date.DayOfWeekName	返回星期几。
Date.DayOfYear	从 DateTime 值返回一个数字，此数字表示一年中的第几日。
Date.DaysInMonth	从 DateTime 值返回所在月份的天数。
Date.EndOfDay	返回一天结束时的 DateTime 值。
Date.EndOfMonth	返回月份结束时的 DateTime 值。
Date.EndOfQuarter	返回一个 Date/DateTime/DateTimeZone 值，此值表示季度的结束。日期和时间部分将重置为该季度的终止值。时区信息会保留。
Date.EndOfWeek	返回一周结束时的 DateTime 值。
Date.EndOfYear	返回年份结束时的 DateTime 值。
Date.From	返回值中的日期值。
Date.FromText	返回一组日期格式和区域性值的 Date 值。

Date.IsInCurrentDay	指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于当日内。
Date.IsInCurrentMonth	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于当前月份内。
Date.IsInCurrentQuarter	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于当前季度内。
Date.IsInCurrentWeek	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于当周内。
Date.IsInCurrentYear	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于当年内。
Date.IsInNextDay	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于下一天内。
Date.IsInNextMonth	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于下一月内。
Date.IsInNextNDays	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于下一天数内。
Date.IsInNextNMonths	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于下一月数内。
Date.IsInNextNQuarters	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于下一季度数内。
Date.IsInNextNWeeks	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于下一周数内。
Date.IsInNextNYears	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于下一年数内。
Date.IsInNextQuarter	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于下一季度内。
Date.IsInNextWeek	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于下一周内。
Date.IsInNextYear	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于下一年内。

Date.IsInPreviousDay	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于前一天内。
Date.IsInPreviousMonth	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于上一月内。
Date.IsInPreviousNDays	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于上一天数内。
Date.IsInPreviousNMonths	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于上一月数内。
Date.IsInPreviousNQuarters	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于上一季度数内。
Date.IsInPreviousNWeeks	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于上一周数内。
Date.IsInPreviousNYears	指示给定的日期/时间值 dateTime 是否按系统当前日期和时间所确定的那样处于上一年数内。
Date.IsInPreviousQuarter	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于上一季度内。
Date.IsInPreviousWeek	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于上一周内。
Date.IsInPreviousYear	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于上一年内。
Date.IsInYearToDate	返回一个逻辑值，此值指示给定的 Date/DateTime/DateTimeZone 是否按系统当前日期和时间所确定的那样处于以当前年份 1 月 1 日开始至当前日期结束的时间段内。
Date.IsLeapYear	返回一个逻辑值，此值指示 DateTime 值的年份部分是否为闰年。
Date.Month	返回 DateTime 值中的月份。
Date.MonthName	返回月份部分的名称。
Date.QuarterOfYear	返回介于 1 到 4 之间的一个数字，此数字表示 DateTime 值中该年份的季度。
Date.StartOfDay	返回一天开始时的 DateTime 值。
Date.StartOfMonth	返回表示月份开始时的 DateTime 值。
Date.StartOfQuarter	返回表示季度开始时的 DateTime 值。

Date.StartOfWeek	返回表示一周开始时的 DateTime 值。
Date.StartOfYear	返回表示年份开始时的 DateTime 值。
Date.ToRecord	返回包含 Date 值的各个部分的记录。
Date.ToText	返回 Date 值中的文本值。
Date.WeekOfMonth	返回一个数字，此数字表示当前月份中的周数。
Date.WeekOfYear	返回一个数字，此数字表示当前年份中的周数。
Date.Year	返回 DateTime 值中的年份。
#date	从年、月和日创建一个日期值。

Day.Sunday	表示星期日。
Day.Monday	表示星期一。
Day.Tuesday	表示星期二。
Day.Wednesday	表示星期三。
Day.Thursday	表示星期四。
Day.Friday	表示星期五。
Day.Saturday	表示星期六。

Date.AddDays

2019/12/4 •

语法

```
Date.AddDays(dateTime as any, numberOfDays as number) as any
```

关于

返回将 `numberOfDays` 天数添加到 `datetime` 值 `dateTime` 所得到的 `date`、`datetime` 或 `datetimezone` 结果。

- `dateTime`：要向其中添加天数的 `date`、`datetime` 或 `datetimezone` 值。
- `numberOfDays`：要添加的天数。

示例 1

将 5 天添加到表示日期 2011/5/14 的 `date`、`datetime` 或 `datetimezone` 值。

```
Date.AddDays(#date(2011, 5, 14), 5)
```

`#date(2011, 5, 19)`

Date.AddMonths

2019/12/3 •

语法

```
Date.AddMonths(dateTime as any, numberOfMonths as number) as any
```

关于

返回将 `numberOfMonths` 月份添加到 `datetime` 值 `dateTime` 所得到的 `date`、`datetime` 或 `datetimezone` 结果。

- `dateTime`：要向其中添加月份的 `date`、`datetime` 或 `datetimezone` 值。
- `numberOfMonths`：要添加的月数。

示例 1

将 5 个月添加到表示日期 2011/5/14 的 `date`、`datetime` 或 `datetimezone` 值。

```
Date.AddMonths(#date(2011, 5, 14), 5)
```

```
#date(2011, 10, 14)
```

示例 2

将 18 个月添加到表示日期 2011/5/14 和时间上午 08:15:22 的 `date`、`datetime` 或 `datetimezone` 值。

```
Date.AddMonths(#datetime(2011, 5, 14, 8, 15, 22), 18)
```

```
#datetime(2012, 11, 14, 8, 15, 22)
```

Date.AddQuarters

2019/12/3 •

语法

```
Date.AddQuarters(dateTime as any, numberOfQuarters as number) as any
```

关于

返回将 `numberOfQuarters` 个季度添加到 `datetime` 值 `dateTime` 所得到的 `date`、`datetime` 或 `datetimezone` 结果。

- `dateTime`：要向其中添加季度的 `date`、`datetime` 或 `datetimezone` 值。
- `numberOfQuarters`：要添加的季度数。

示例 1

将1 个季度添加到表示日期 2011/5/14 的 `date`、`datetime` 或 `datetimezone` 值。

```
Date.AddQuarters(#date(2011, 5, 14), 1)
```

`#date(2011, 8, 14)`

Date.AddWeeks

2019/12/4 •

语法

```
Date.AddWeeks(dateTime as any, numberOfWeeks as number) as any
```

关于

返回向 `datetime` 值 `dateTime` 添加 `numberOfWeeks` 周后所得的 `date`、`datetime` 或 `datetimezone` 结果。

- `dateTime`：要向其中添加周的 `date`、`datetime` 或 `datetimezone` 值。
- `numberOfWeeks`：要添加的周数。

示例 1

向表示日期 2011/5/14 的 `date`、`datetime` 或 `datetimezone` 值添加 2 周。

```
Date.AddWeeks(#date(2011, 5, 14), 2)
```

`#date(2011, 5, 28)`

Date.AddYears

2019/12/3 •

语法

```
Date.AddYears(dateTime as any, numberOfYears as number) as any
```

关于

返回将 `numberOfYears` 加上 `datetime` 值 `dateTime` 的 `date`、`datetime` 或 `datetimezone` 结果。

- `dateTime`：向其增加年份值的 `date`、`datetime` 或 `datetimezone` 值。
- `numberOfYears`：要添加的年数。

示例 1

将 4 年添加到表示日期 2011/5/14 的 `date`、`datetime` 或 `datetimezone` 值。

```
Date.AddYears(#date(2011, 5, 14), 4)
```

`#date(2015, 5, 14)`

示例 2

将 10 年添加到表示日期 2011/5/14 和时间上午 08:15:22 的 `date`、`datetime` 或 `datetimezone` 值。

```
Date.AddYears(#datetime(2011, 5, 14, 8, 15, 22), 10)
```

`#datetime(2021, 5, 14, 8, 15, 22)`

Date.Day

2019/12/3 •

语法

```
Date.Day(dateTime as any) as nullable number
```

关于

返回 `date`、`datetime` 或 `datetimezone` 值的天数部分。

- `dateTime`：从中提取天数部分的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

获取表示日期和时间值为 2011/5/14 下午 05:00:00 的 `date`、`datetime` 或 `datetimezone` 值的天数部分。

```
Date.Day(#datetime(2011, 5, 14, 17, 0, 0))
```

`14`

Date.DayOfWeek

2019/12/4 •

语法

```
Date.DayOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

关于

返回数字（介于 0 到 6 之间），以指明提供的 `dateTime` 是星期几。

- `dateTime`：`date`、`datetime` 或 `datetimezone` 值。
- `firstDayOfWeek`：`Day` 值，指示应将星期几视为一周的第一天。允许的值为 Day.Sunday、Day.Monday、Day.Tuesday、Day.Wednesday、Day.Thursday、Day.Friday 或 Day.Saturday。如果未指定，则使用与区域性相关的默认值。

示例 1

获取 2011 年 2 月 21 日（星期一）代表的是星期几（将星期日视为一周的第一天）。

```
Date.DayOfWeek(#date(2011, 02, 21), Day.Sunday)`
```

`1`

示例 2

获取 2011 年 2 月 21 日（星期一）代表的是星期几（将星期一视为一周的第一天）。

```
Date.DayOfWeek(#date(2011, 02, 21), Day.Monday)
```

`0`

Date.DayOfWeekName

2019/12/4 •

语法

```
Date.DayOfWeekName(date as any, optional culture as nullable text)
```

关于

针对所提供的 `date` 和可选的区域性 `culture` 返回星期几。

示例 1

获取星期几。

```
Date.DayOfWeekName(#date(2011, 12, 31), "en-US")
```

`"Saturday"`

Date.DayOfYear

2019/12/3 •

语法

```
Date.DayOfYear(dateTime as any) as nullable number
```

关于

返回一个数字，代表所提供的 `date`、`datetime` 或 `datetimezone` 值 `dateTime` 中的一年中的某一天。

示例 1

2011 年 3 月 1 日这一天所代表的数值（`#date(2011, 03, 01)`）。

```
Date.DayOfYear(#date(2011, 03, 01))
```

`60`

Date.DaysInMonth

2019/12/3 •

语法

```
Date.DaysInMonth(dateTime as any) as nullable number
```

关于

返回 `date`、`datetime` 或 `datetimezone` 值 `dateTime` 中月份的天数。

- `dateTime`:会为其返回月份天数的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

由 `#date(2011, 12, 01)` 表示的十二月的天数。

```
Date.DaysInMonth(#date(2011, 12, 01))
```

```
31
```

Date.EndOfDay

2019/12/3 •

```
Date.EndOfDay(dateTime as any) as any
```

关于

返回一个 `date`、`datetime` 或 `datetimezone` 值，表示 `dateTime` 中一天结束值。保留时区信息。

- `dateTime`：用于计算一天结束值的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

获取 2011/5/14 下午 05:00:00 的一天结束值。

```
Date.EndOfDay(#datetime(2011, 5, 14, 17, 0, 0))
```

```
#datetime(2011, 5, 14, 23, 59, 59.9999999)
```

示例 2

获取 2011/5/17 下午 05:00:00-7:00 的一天结束值。

```
Date.EndOfDay(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

```
#datetimezone(2011, 5, 17, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfMonth

2019/12/3 •

语法

```
Date.EndOfMonth(dateTime as any) as any
```

关于

返回 `dateTime` 中月份的最后一天。

- `dateTime`：用于计算月份结束值的 `date`、`datetime` 或 `datetimezone` 值

示例 1

获取 2011/5/14 的月份结束值。

```
Date.EndOfMonth(#date(2011, 5, 14))
```

`#date(2011, 5, 31)`

示例 2

获取 2011/5/17 下午 05:00:00-7:00 的月份结束值。

```
Date.EndOfMonth(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

`#datetimezone(2011, 5, 31, 23, 59, 59.9999999, -7, 0)`

Date.EndOfQuarter

2019/12/4 •

语法

```
Date.EndOfQuarter(dateTime as any) as any
```

关于

返回一个 `date`、`datetime` 或 `datetimezone` 值，表示 `dateTime` 中的季度结束。保留时区信息。

- `dateTime`：用于计算季度结束值的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

查找 2011 年 10 月 10 日上午 8:00 (`#datetime(2011, 10, 10, 8, 0, 0)`) 的季度结束值。

```
Date.EndOfQuarter(#datetime(2011, 10, 10, 8, 0, 0))
```

`#datetime(2011, 12, 31, 23, 59, 59.9999999)`

Date.EndOfWeek

2019/12/4 •

语法

```
Date.EndOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as any
```

关于

返回所提供的 `date`、`datetime` 或 `datetimezone` `dateTime` 中的星期的最后一天。此函数采用可选的 `Day` (`firstDayOfWeek`) 为此相关计算设置一周的第一天。默认值为 `Day.Sunday`。

- `dateTime`：用于计算星期的最后一天的 `date`、`datetime` 或 `datetimezone` 值
- `firstDayOfWeek`：*[可选]* 表示星期的最后一天的 `Day.Type` 值。可能的值为 `Day.Sunday`、`Day.Monday`、`Day.Tuesday`、`Day.Wednesday`、`Day.Thursday`、`Day.Friday` 和 `Day.Saturday.`。默认值为 `Day.Sunday`。

示例 1

获取 5/14/2011 的星期结束值。

```
Date.EndOfWeek(#date(2011, 5, 14))
```

```
#date(2011, 5, 14)
```

示例 2

获取 5/17/2011 05:00:00 PM -7:00 的星期结束值，Sunday 作为该星期的第一天。

```
Date.EndOfWeek(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0), Day.Sunday)
```

```
#datetimezone(2011, 5, 21, 23, 59, 59.9999999, -7, 0)
```

Date.EndOfYear

2019/12/3 •

语法

```
Date.EndOfYear(dateTime as any) as any
```

关于

返回一个值，以 `dateTime` 形式(含小数秒)表示年份结束值。保留时区信息。

- `dateTime`:用于计算年份结束值的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

获取 5/14/2011 05:00:00 PM 的年份结束值。

```
Date.EndOfYear(#datetime(2011, 5, 14, 17, 0, 0))
```

`#datetime(2011, 12, 31, 23, 59, 59.9999999)`

示例 2

获取 5/17/2011 05:00:00 PM -7:00 的小时结束值。

```
Date.EndOfYear(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

`#datetimezone(2011, 12, 31, 23, 59, 59.9999999, -7, 0)`

Date.From

2019/12/3 •

语法

```
Date.From(value as any, optional culture as nullable text) as nullable date
```

关于

从给定的 `value` 返回 `date` 值。如果给定的 `value` 为 `null`，则 `Date.From` 返回 `null`。如果给定的 `value` 为 `date`，则返回 `value`。可以将以下类型的值转换为 `date` 值：

- `text`：文本表示形式的 `date` 值。有关详细信息，请参阅 `Date.FromText`。
- `datetime`：`value` 的日期部分。
- `datetimezone`：与 `value` 等效的本地日期/时间的日期部分。
- `number`：与 `value` 所示的 OLE 自动化日期等效的日期/时间的日期部分。

如果 `value` 为任何其他类型，则返回错误。

示例 1

将 `43910` 转换为 `date` 值。

```
Date.From(43910)
```

`#date(2020, 3, 20)`

示例 2

将 `#datetime(1899, 12, 30, 06, 45, 12)` 转换为 `date` 值。

```
Date.From(#datetime(1899, 12, 30, 06, 45, 12))
```

`#date(1899, 12, 30)`

Date.FromText

2019/12/4 •

语法

```
Date.FromText(text as nullable text, optional culture as nullable text) as nullable date
```

关于

根据 ISO 8601 格式标准，从文本表示形式 `text` 创建 `date` 值。

- `Date.FromText("2010-02-19")` // 日期，yyyy-MM-dd

示例 1

将 `"December 31, 2010"` 转换为日期值。

```
Date.FromText("2010-12-31")
```

`#date(2010, 12, 31)`

示例 2

将 `"December 31, 2010"` 转换为不同格式的日期值

```
Date.FromText("2010, 12, 31")
```

`#date(2010, 12, 31)`

示例 3

将 `"December, 2010"` 转换为日期值。

```
Date.FromText("2010, 12")
```

`#date(2010, 12, 1)`

示例 4

将 `"2010"` 转换为日期值。

```
Date.FromText("2010")
```

`#date(2010, 1, 1)`

Date.IsInCurrentDay

2019/12/3 •

语法

```
Date.IsInCurrentDay(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于当日内。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例

确定当前系统时间是否处于当日内。

```
Date.IsInCurrentDay(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentMonth

2019/12/4 •

语法

```
Date.IsInCurrentMonth(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为当前这一个月的时间(由系统上的当前日期和时间确定)。

- `dateTime`:要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否为当前这一个月的时间。

```
Date.IsInCurrentMonth(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentQuarter

2019/12/4 •

语法

```
Date.IsInCurrentQuarter(dateTime as any) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于当季度内。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否处于当前季度。

```
Date.IsInCurrentQuarter(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentWeek

2019/12/4 •

语法

```
Date.IsInCurrentWeek(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于当周内。

- `dateTime`:要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否处于当周内。

```
Date.IsInCurrentWeek(DateTime.FixedLocalNow())
```

`true`

Date.IsInCurrentYear

2019/12/4 •

语法

```
Date.IsInCurrentYear(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于当前年份内。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否处于当前年份。

```
Date.IsInCurrentYear(DateTime.FixedLocalNow())
```

`true`

Date.IsInNextDay

2019/12/4 •

语法

```
Date.IsInNextDay(dateTime as any) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于下一天内。注意，如果传递的值处于当天内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之后的那一天是否处于下一天。

```
Date.IsInNextDay(Date.AddDays(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInNextMonth

2019/12/3 •

语法

```
Date.IsInNextMonth(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于下一月内。注意，如果传递的值处于当前月份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之后的月份是否处于下一月。

```
Date.IsInNextMonth(Date.AddMonths(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInNextNDays

2019/12/3 •

语法

```
Date.IsInNextNDays(dateTime as any, days as number) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于接下来的天数内。注意，如果传递的值处于当天内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `days`：天数。

示例 1

确定当前系统时间的后一天是否在接下来的两天中。

```
Date.IsInNextNDays(Date.AddDays(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextNMonths

2019/12/4 •

```
Date.IsInNextNMonths(dateTime as any, months as number) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于接下来的月份数内。注意，如果传递的值处于当前月份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `months`：月数。

示例 1

确定当前系统时间之后的月份是否在接下来的两个月中。

```
Date.IsInNextNMonths(Date.AddMonths(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextNQuarters

2019/12/4 •

语法

```
Date.IsInNextNQuarters(dateTime as any, quarters as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于下一季度数内。注意，如果传递的值处于当前季度内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `quarters`：季度数。

示例 1

确定当前系统时间之后的季度是否处于接下来的两个季度中。

```
Date.IsInNextNQuarters(Date.AddQuarters(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextNWeeks

2019/12/4 •

语法

```
Date.IsInNextNWeeks(dateTime as any, weeks as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为下几周的时间(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一周的时间，此函数将返回 false。

- `dateTime` :要计算的 `date` 、`datetime` 或 `datetimezone` 值。
- `weeks` :周数。

示例 1

确定当前系统时间之后的周值是否为下两周的时间。

```
Date.IsInNextNWeeks(Date.AddDays(DateTime.FixedLocalNow(), 7), 2)
```

`true`

Date.IsInNextNYears

2019/12/4 •

语法

```
Date.IsInNextNYears(dateTime as any, years as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为下几年的时间(由系统上的当前日期和时间确定)。注意，如果传递的值处于当前年份内，此函数将返回 false。

- `dateTime` :要计算的 `date` 、`datetime` 或 `datetimezone` 值。
- `years` :年数。

示例 1

确定当前系统时间之后的年值是否为下两年的时间。

```
Date.IsInNextNYears(Date.AddYears(DateTime.FixedLocalNow(), 1), 2)
```

`true`

Date.IsInNextQuarter

2019/12/4 •

语法

```
Date.IsInNextQuarter(dateTime as any) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于下季度内。注意，如果传递的值处于当前季度内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

####示例 1 确定当前系统时间之后的那一季度是否处于下一季度。

```
Date.IsInNextQuarter(Date.AddQuarters(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInNextWeek

2019/12/4 •

语法

```
Date.IsInNextWeek(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 (`dateTime`) 是否为下周的时间(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一周的时间，此函数将返回 false。

- `dateTime` :要计算的 `date` 、 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之后的周值是否为下周的时间。

```
Date.IsInNextWeek(Date.AddDays(DateTime.FixedLocalNow(), 7))
```

`true`

Date.IsInNextYear

2019/12/3 •

语法

```
Date.IsInNextYear(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于下一年份内。注意，如果传递的值处于当前年份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之后的那一年是否处于下一年份内。

```
Date.IsInNextYear(Date.AddYears(DateTime.FixedLocalNow(), 1))
```

`true`

Date.IsInPreviousDay

2019/12/4 •

语法

```
Date.IsInPreviousDay(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于前一天内。注意，如果传递的值处于当天内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的那一天是否处于前一天。

```
Date.IsInPreviousDay(Date.AddDays(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInPreviousMonth

2019/12/3 •

语法

```
Date.IsInPreviousMonth(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于上一月内。注意，如果传递的值处于当前月份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的月份是否处于上个月。

```
Date.IsInPreviousMonth(Date.AddMonths(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInPreviousNDays

2019/12/3 •

语法

```
Date.IsInPreviousNDays(dateTime as any, days as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于之前的天数内。注意，如果传递的值处于当天内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `days`：天数。

示例 1

确定当前系统时间之前的那一天是否在之前的两天中。

```
Date.IsInPreviousNDays(Date.AddDays(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousNMonths

2019/12/3 •

语法

```
Date.IsInPreviousNMonths(dateTime as any, months as number) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于之前的月份数内。注意，如果传递的值处于当前月份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `months`：月数。

示例 1

确定当前系统时间之前的月份是否在之前的两个月中。

```
Date.IsInPreviousNMonths(Date.AddMonths(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousNQuarters

2019/12/3 •

语法

```
Date.IsInPreviousNQuarters(dateTime as any, quarters as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为前几个季度的时间(由系统上的当前日期和时间确定)。注意，如果传递的值处于当前季度内，此函数将返回 false。

- `dateTime`:要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `quarters`:季度数。

示例 1

确定当前系统时间之前的季度是否在之前的两个季度中。

```
Date.IsInPreviousNQuarters(Date.AddQuarters(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousNWeeks

2019/12/3 •

语法

```
Date.IsInPreviousNWeeks(dateTime as any, weeks as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为前几周的时间(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一周的时间，此函数将返回 false。

- `dateTime` :要计算的 `date` 、`datetime` 或 `datetimezone` 值。
- `weeks` :周数。

示例 1

确定当前系统时间之前的周是否是上两周中的时间。

```
Date.IsInPreviousNWeeks(Date.AddDays(DateTime.FixedLocalNow(), -7), 2)
```

`true`

Date.IsInPreviousNYears

2019/12/4 •

语法

```
Date.IsInPreviousNYears(dateTime as any, years as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于之前的年数内。注意，如果传递的值处于当前年份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。
- `years`：年数。

示例 1

确定当前系统时间之前的年份是否在之前的两年中。

```
Date.IsInPreviousNYears(Date.AddYears(DateTime.FixedLocalNow(), -1), 2)
```

`true`

Date.IsInPreviousQuarter

2019/12/3 •

语法

```
Date.IsInPreviousQuarter(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于上一季度内。注意，如果传递的值处于当前季度内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的季度是否处于上个季度。

```
Date.IsInPreviousQuarter(Date.AddQuarters(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInPreviousWeek

2019/12/3 •

语法

```
Date.IsInPreviousWeek(dateTime as any) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于上一周内。注意，如果传递的值为当前这一周的时间，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的那个星期是否处于上个星期。

```
Date.IsInPreviousWeek(Date.AddDays(DateTime.FixedLocalNow(), -7))
```

`true`

Date.IsInPreviousYear

2019/12/4 •

语法

```
Date.IsInPreviousYear(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于上一年份内。注意，如果传递的值处于当前年份内，此函数将返回 false。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的年份是否处于前一年。

```
Date.IsInPreviousYear(Date.AddYears(DateTime.FixedLocalNow(), -1))
```

`true`

Date.IsInYearToDate

2019/12/4 •

语法

```
Date.IsInYearToDate(dateTime as any) as nullable logical
```

关于

指示在当前年份中该给定日期值 `dateTime` 是否出现以及该日期是否就在当天或早于当天，它由系统上的当前日期和时间确定。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否处于此年份至今的日期内。

```
Date.IsInYearToDate(DateTime.FixedLocalNow())
```

`true`

Date.IsLeapYear

2019/12/4 •

语法

```
Date.IsLeapYear(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否处于闰年中。

- `dateTime`：要计算的 `date`、`datetime` 或 `datetimezone` 值。

示例 1

确定由 `#date(2012, 01, 01)` 表示的 2012 年是否为闰年。

```
Date.IsLeapYear(#date(2012, 01, 01))
```

`true`

Date.Month

2019/12/3 •

语法

```
Date.Month(dateTime as any) as nullable number
```

关于

返回所提供的 `datetime` 值的月份部分 `dateTime`。

示例 1

查找 #datetime(2011, 12, 31, 9, 15, 36) 中的月份。

```
Date.Month(#datetime(2011, 12, 31, 9, 15, 36))
```

```
12
```

Date.MonthName

2019/12/4 •

语法

```
Date.MonthName(date as any, optional culture as nullable text) as nullable text
```

关于

返回所提供的 `date` 的月份部分名称，或者可选择返回区域性 `culture`。

示例

获取月份名称。

```
Date.MonthName(#datetime(2011, 12, 31, 5, 0, 0), "en-US")
```

`"December"`

Date.QuarterOfYear

2019/12/4 •

语法

```
Date.QuarterOfYear(dateTime as any) as nullable number
```

关于

返回一个介于 1 到 4 之间的数值，该数值指示日期 `dateTime` 属于年份中的哪一季度。`dateTime` 可以是 `date`、`datetime` 或 `datetimezone` 值。

示例 1

查找日期 #date(2011, 12, 31) 属于年份中的哪个季度。

```
Date.QuarterOfYear(#date(2011, 12, 31))
```

`4`

Date.StartOfDay

2019/12/3 •

语法

```
Date.StartOfDay(dateTime as any) as any
```

关于

返回 `dateTime` 那天的第一个值。 `dateTime` 必须是 `date` 、 `datetime` 或 `datetimezone` 值。

示例 1

查找 2011 年 10 月 10 日上午 8:00 (`#datetime(2011, 10, 10, 8, 0, 0)`) 的一天开始时的值。

```
Date.StartOfDay(#datetime(2011, 10, 10, 8, 0, 0))
```

`#datetime(2011, 10, 10, 0, 0, 0)`

Date.StartOfMonth

2019/12/4 •

语法

```
Date.StartOfMonth(dateTime as any) as any
```

关于

返回给定的 `date` 或 `datetime` 类型的月份的第一个值。

示例 1

查找 2011 年 10 月 10 日上午 8:10:32 (`#datetime(2011, 10, 10, 8, 10, 32)`) 的月份开始值。

```
Date.StartOfMonth(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 10, 1, 0, 0, 0)`

Date.StartOfQuarter

2019/12/3 •

语法

```
Date.StartOfQuarter(dateTime as any) as any
```

关于

返回季度的第一个值 < `dateTime`。 `dateTime` 必须是 `date`、`datetime` 或 `datetimezone` 值。

示例 1

查找 2011 年 10 月 10 日上午 8:00 (`#datetime(2011, 10, 10, 8, 0, 0)`) 的季度开始值。

```
Date.StartOfQuarter(#datetime(2011, 10, 10, 8, 0, 0))
```

`#datetime(2011, 10, 1, 0, 0, 0)`

Date.StartOfWeek

2019/12/4 •

语法

```
Date.StartOfWeek(dateTime as any, optional firstDayOfWeek as nullable number) as any
```

关于

给定 `date`、`datetime` 或 `datetimezone` 值，返回一周的第一个值。

示例 1

查找 2011 年 10 月 10 日上午 8:10:32 (`#datetime(2011, 10, 10, 8, 10, 32)`) 的一周开始时的值。

```
Date.StartOfWeek(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 10, 9, 0, 0, 0)`

Date.StartOfYear

2019/12/3 •

语法

```
Date.StartOfYear(dateTime as any) as any
```

关于

给定 `date`、`datetime` 或 `datetimezone` 值，返回年份的第一个值。

示例 1

查找 2011 年 10 月 10 日上午 8:10:32 (`#datetime(2011, 10, 10, 8, 10, 32)`) 的年份开始值。

```
Date.StartOfYear(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 1, 1, 0, 0, 0)`

Date.ToRecord

2019/12/3 •

语法

```
Date.ToRecord(date as date) as record
```

关于

返回包含给定日期值 `date` 的各个部分的记录。

- `date`：用于计算其部分的记录的 `date` 值。

示例 1

将 `#date(2011, 12, 31)` 值转换为包含日期值的各个部分的记录。

```
Date.ToRecord(#date(2011, 12, 31))
```

[Year]	2011
[Month]	12
[Day]	31

Date.ToText

2019/12/4 •

语法

```
Date.ToText(date as nullable date, optional format as nullable text, optional culture as nullable text) as nullable text
```

关于

返回日期值 `date` 的文本表示形式，`date` 此函数采用可选格式参数 `format`。有关所支持格式的完整列表，请参阅库规范文档。

示例 1

获取 #date(2010, 12, 31) 的文本表示形式。

```
Date.ToText(#date(2010, 12, 31))
```

`"12/31/2010"`

示例 2

使用格式选项获取 #date(2010, 12, 31) 的文本表示形式。

```
Date.ToText(#date(2010, 12, 31), "yyyy/MM/dd")
```

`"2010/12/31"`

Date.WeekOfMonth

2019/12/3 •

语法

```
Date.WeekOfMonth(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

关于

返回一个介于 1 到 5 之间的数值，该数值指示日期 `dateTime` 属于月份中的哪一周。

- `dateTime`：`datetime` 值，用于确定是月份的第几个星期。

示例 1

确定 3 月 15 日属于 2011 年（`#date(2011, 03, 15)`）的哪一周。

```
Date.WeekOfMonth(#date(2011, 03, 15))
```

`3`

Date.WeekOfYear

2019/12/9 •

语法

```
Date.WeekOfYear(dateTime as any, optional firstDayOfWeek as nullable number) as nullable number
```

关于

返回一个介于 1 到 54 之间的数值，该数值指示日期 `dateTime` 属于年份中的哪一周。

- `dateTime`：用于确定一年中的某一周的 `datetime` 值。
- `firstDayOfWeek`：可选 `Day.Type` 值，指示哪一天被视为新周的开始，例如，`Day.Sunday`。如果未指定，则使用与区域性相关的默认值。

示例 1

确定 2011 年 3 月 27 日属于该年的哪一周（`#date(2011, 03, 27)`）。

```
Date.WeekOfYear(#date(2011, 03, 27))
```

```
14
```

示例 2

确定 2011 年 3 月 27 日属于该年的哪一周（`#date(2011, 03, 27)`），使用"星期一"作为新的一周的开始。

```
Date.WeekOfYear(#date(2011, 03, 27), Day.Monday)
```

```
13
```

Date.Year

2019/12/4 •

```
Date.Year(dateTime as any) as nullable number
```

关于

返回所提供的 `datetime` 值 `dateTime` 的年份部分。

示例 1

查找 #datetime(2011, 12, 31, 9, 15, 36) 中的年份。

```
Date.Year(#datetime(2011, 12, 31, 9, 15, 36))
```

`2011`

Day.Friday

2019/12/3 •

关于

返回 6，此数值表示星期五。

Day.Monday

2019/12/3 •

关于

返回 2，该数值表示星期一。

Day.Saturday

2019/12/3 •

关于

返回 7，此数值表示星期六。

Day.Sunday

2019/12/3 •

关于

返回 1，此数值表示星期日。

Day.Thursday

2019/12/3 •

关于

返回 5，该数值表示星期四。

Day.Tuesday

2019/12/4 •

关于

返回 3，此数值表示星期二。

Day.Wednesday

2019/12/4 •

关于

返回 4，该数值表示星期三。

#date

2019/12/3 •

语法

```
#date(year as number, month as number, day as number) as date
```

关于

从年 `year`、月 `month` 和日 `day` 创建一个日期值。如果不满足以下条件，则会引发错误：

- 1 ≤ year ≤ 9999
- 1 ≤ month ≤ 12
- 1 ≤ day ≤ 31

日期/时间函数

2020/4/26 •

这些函数创建并操纵 datetime 值和 datetimezone 值。

DateTime

"	"
DateTime.AddZone	将 timezonehours 添加为输入日期时间值的偏移量，并返回新的 datetimezone 值。
DateTime.Date	返回 DateTime 值的日期部分
DateTime.FixedLocalNow	返回设置为系统上的当前日期和时间的 DateTime 值。
DateTime.From	返回值中的日期时间值。
DateTime.FromFileTime	返回提供的表中的 DateTime 值。
DateTime.FromText	返回一组日期格式和区域性值的 DateTime 值。
DateTime.IsInCurrentHour	指示给定的日期/时间值是否为当前这一小时的时间(由系统上的当前日期和时间确定)。
DateTime.IsInCurrentMinute	指示给定的日期/时间值是否为当前这一分钟的时间(由系统上的当前日期和时间确定)。
DateTime.IsInCurrentSecond	指示给定的日期/时间值是否为当前这一秒的时间(由系统上的当前日期和时间确定)。
DateTime.IsInNextHour	指示给定的日期/时间值是否为下一小时的时间(由系统上的当前日期和时间确定)。
DateTime.IsInNextMinute	指示给定的日期/时间值是否为下一分钟的时间(由系统上的当前日期和时间确定)。
DateTime.IsInNextNHours	指示给定的日期/时间值是否为下几个小时的时间(由系统上的当前日期和时间确定)。
DateTime.IsInNextNMinutes	指示给定的日期/时间值是否为接下来的几分钟时间(由系统上的当前日期和时间确定)。
DateTime.IsInNextNSeconds	指示给定的日期/时间值是否为接下来的几秒钟时间(由系统上的当前日期和时间确定)。
DateTime.IsInNextSecond	指示给定的日期/时间值是否为下一秒的时间(由系统上的当前日期和时间确定)。
DateTime.IsInPreviousHour	指示给定的日期/时间值是否为上一小时的时间(由系统上的当前日期和时间确定)。

DateTime.IsInPreviousMinute	指示给定的日期/时间值是否为上一分钟的时间(由系统上的当前日期和时间确定)。
DateTime.IsInPreviousNHours	指示给定的日期/时间值是否为前几个小时的时间(由系统上的当前日期和时间确定)。
DateTime.IsInPreviousNMinutes	指示给定的日期/时间值是否为前几分钟的时间(由系统上的当前日期和时间确定)。
DateTime.IsInPreviousNSeconds	指示给定的日期/时间值是否为前几秒钟的时间(由系统上的当前日期和时间确定)。
DateTime.IsInPreviousSecond	指示给定的日期/时间值是否为前一秒的时间(由系统上的当前日期和时间确定)。
DateTime.LocalNow	返回设置为系统上的当前日期和时间的 DateTime 值。
DateTime.Time	返回 DateTime 值的时间部分。
DateTime.ToRecord	返回包含 DateTime 值的各个部分的记录。
DateTime.ToText	从 DateTime 值返回文本值。
#datetime	使用年、月、日、小时、分钟和秒创建一个日期/时间值。

DateTime.AddZone

2019/12/3 •

语法

```
DateTime.AddZone(dateTime as nullable datetime, timezoneHours as number, optional timezoneMinutes
as nullable number) as nullable datetimezone
```

关于

设置日期/时间值 `dateTime` 的时区信息。时区信息将包括 `timezoneHours` 和 `timezoneMinutes`（可选）。

示例 1

将 #datetime(2010, 12, 31, 11, 56, 02) 的时区信息设置为 7 小时，30 分钟。

```
DateTime.AddZone(#datetime(2010, 12, 31, 11, 56, 02), 7, 30)
```

`#datetimezone(2010, 12, 31, 11, 56, 2, 7, 30)`

DateTime.Date

2019/12/4 •

语法

```
DateTime.Date(dateTime as any) as nullable date
```

关于

返回给定的 `date`、`datetime` 或 `datetimezone` 值 `dateTime` 的日期部分。

示例 1

查找 #datetime(2010, 12, 31, 11, 56, 02) 的日期值。

```
DateTime.Date(#datetime(2010, 12, 31, 11, 56, 02))
```

```
#date(2010, 12, 31)
```

DateTime.FixedLocalNow

2019/12/3 •

语法

```
DateTime.FixedLocalNow() as datetime
```

关于

返回设置为系统上的当前日期和时间的 `datetime` 值。该值是固定的，因此将不会随着连续调用而更改，这与 DateTime.LocalNow 不同，后者可能会在表达式的执行过程中返回不同值。

DateTime.From

2019/12/4 •

语法

```
DateTime.From(value as any, optional culture as nullable text) as nullable datetime
```

关于

从给定的 `value` 返回 `datetime` 值。如果给定的 `value` 为 `null`，则 `DateTime.From` 返回 `null`。如果给定的 `value` 为 `datetime`，则返回 `value`。可以将以下类型的值转换为 `datetime` 值：

- `text`：文本表示形式的 `datetime` 值。有关详细信息，请参阅 `DateTime.FromText`。
- `date`：一个 `datetime`，具有作为日期部分的 `value` 和作为时间部分的 `12:00:00 AM`。
- `datetimezone`：`value` 的本地 `datetime` 等效值。
- `time`：一个 `datetime`，具有作为日期部分的 `0` 的 OLE 自动化日期的等效日期，以及作为时间部分的 `value`。
- `number`：等效于 `value` 表示的 OLE 自动化日期的 `datetime`。

如果 `value` 为任何其他类型，则返回错误。

示例 1

将 `#time(06, 45, 12)` 转换为 `datetime` 值。

```
DateTime.From(#time(06, 45, 12))
```

`#datetime(1899, 12, 30, 06, 45, 12)`

示例 2

将 `#date(1975, 4, 4)` 转换为 `datetime` 值。

```
DateTime.From(#date(1975, 4, 4))
```

`#datetime(1975, 4, 4, 0, 0, 0)`

DateTime.FromFileTime

2019/12/3 •

语法

```
DateTime.FromFileTime(fileTime as nullable number) as nullable datetime
```

关于

根据 `fileTime` 值创建 `datetime` 值，并将其转换为本地时区。fileTime 是一个 Windows 文件时间值，表示自公元 1601 年 1 月 1 日午夜 12:00 开始经过的 100 纳秒间隔数 。协调世界时 (UTC)。

示例 1

将 `129876402529842245` 转换为日期/时间值。

```
DateTime.FromFileTime(129876402529842245)
```

```
#datetime(2012, 7, 24, 14, 50, 52.9842245)
```

DateTime.FromText

2019/12/3 •

语法

```
DateTime.FromText(text as nullable text, optional culture as nullable text) as nullable datetime
```

关于

根据 ISO 8601 格式标准，从文本表示形式 `text` 创建 `datetime` 值。

- `DateTime.FromText("2010-12-31T01:30:00")` // yyyy-MM-ddThh:mm:ss

示例 1

将 `"2010-12-31T01:30:25"` 转换为日期/时间值。

```
DateTime.FromText("2010-12-31T01:30:25")
```

`#datetime(2010, 12, 31, 1, 30, 25)`

示例 2

将 `"2010-12-31T01:30"` 转换为日期/时间值。

```
DateTime.FromText("2010-12-31T01:30")
```

`#datetime(2010, 12, 31, 1, 30, 0)`

示例 3

将 `"20101231T013025"` 转换为日期/时间值。

```
DateTime.FromText("20101231T013025")
```

`#datetime(2010, 12, 31, 1, 30, 25)`

示例 4

将 `"20101231T01:30:25"` 转换为日期/时间值。

```
DateTime.FromText("20101231T01:30:25")
```

`#datetime(2010, 12, 31, 1, 30, 25)`

示例 5

将 `"20101231T01:30:25.121212"` 转换为**日期**/时间值。

```
DateTime.FromText("20101231T01:30:25.121212")
```

`#datetime(2010, 12, 31, 1, 30, 25.121212)`

DateTime.IsInCurrentHour

2019/12/4 •

语法

```
DateTime.IsInCurrentHour(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为当前这一小时的时间(由系统上的当前日期和时间确定)。

- `dateTime` :要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否为当前这一个小时的时间。

```
DateTime.IsInCurrentHour(DateTime.FixedLocalNow())
```

`true`

DateTime.IsInCurrentMinute

2019/12/4 •

语法

```
DateTime.IsInCurrentMinute(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为当前这一分钟的时间(由系统上的当前日期和时间确定)。

- `dateTime`:要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间是否为当前这一分钟的时间。

```
DateTime.IsInCurrentMinute(DateTime.FixedLocalNow())
```

`true`

DateTime.IsInCurrentSecond

2019/12/4 •

语法

```
DateTime.IsInCurrentSecond(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为当前这一秒的时间（由系统上的当前日期和时间确定）。

- `dateTime` :要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间为当前这一秒的时间。

```
DateTime.IsInCurrentSecond(DateTime.FixedLocalNow())
```

`true`

DateTime.IsInNextHour

2020/4/30 •

语法

```
DateTime.IsInNextHour(dateTime as any) as nullable logical
```

关于

指示给定的日期时间值（`dateTime`）是否按系统当前日期和时间所确定的那样处于下一小时内。注意，如果传递的值为当前这一小时的时间，此函数将返回 false。

- `dateTime`：要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间后的小时是否处于接下来的一小时内。

```
DateTime.IsInNextHour(DateTime.FixedLocalNow() + #duration(0, 1, 0, 0))
```

`true`

DateTime.IsInNextMinute

2020/4/30 •

语法

```
DateTime.IsInNextMinute(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否处于下一分钟内(由系统上的当前日期和时间确定)。注意，如果传递的值处于当前分钟内，此函数将返回 false。

- `dateTime`:要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间后的一分钟是否处于接下来的一分钟内。

```
DateTime.IsInNextMinute(DateTime.FixedLocalNow() + #duration(0, 0, 1, 0))
```

`true`

DateTime.IsInNextNHours

2020/4/30 •

语法

```
DateTime.IsInNextNHours(dateTime as any, hours as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否处于接下来的几小时时间内(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一小时的时间，此函数将返回 false。

- `dateTime` :要计算的 `datetime` 或 `datetimezone` 值。
- `hours` :小时数。

示例 1

确定当前系统时间后的一小时是否处于接下来的两个小时内。

```
DateTime.IsInNextNHours(DateTime.FixedLocalNow() + #duration(0, 2, 0, 0), 2)
```

`true`

DateTime.IsInNextNMinutes

2020/4/30 •

语法

```
DateTime.IsInNextNMinutes(dateTime as any, minutes as number) as nullable logical
```

关于

指示给定的日期时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于接下来的分钟数内。注意，如果传递的值处于当前分钟内，此函数将返回 false。

- `dateTime`：要计算的 `datetime` 或 `datetimezone` 值。
- `minutes`：分钟数。

示例 1

确定当前系统时间后的分钟值是否处于接下来的两分钟内。

```
DateTime.IsInNextNMinutes(DateTime.FixedLocalNow() + #duration(0, 0, 2, 0), 2)
```

`true`

DateTime.IsInNextNSeconds

2020/4/30 •

语法

```
DateTime.IsInNextNSeconds(dateTime as any, seconds as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为下几秒的时间(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一秒的时间，此函数将返回 false。

- `dateTime`:要计算的 `datetime` 或 `datetimezone` 值。
- `seconds`:秒数。

示例 1

确定当前系统时间之后的秒值是否为下两秒的时间。

```
DateTime.IsInNextNSeconds(DateTime.FixedLocalNow() + #duration(0, 0, 0, 2), 2)
```

`true`

DateTime.IsInNextSecond

2020/4/30 •

语法

```
DateTime.IsInNextSecond(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 (`dateTime`) 是否为下一秒的时间(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一秒的时间，此函数将返回 false。

- `dateTime` :要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之后的秒值是否为下一秒的时间。

```
DateTime.IsInNextSecond(DateTime.FixedLocalNow() + #duration(0, 0, 0, 1))
```

`true`

DateTime.IsInPreviousHour

2020/4/30 •

语法

```
DateTime.IsInPreviousHour(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否为上一小时的时间(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一小时的时间，此函数将返回 false。

- `dateTime` :要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的小时是否为上一小时的时间。

```
DateTime.IsInPreviousHour(DateTime.FixedLocalNow() - #duration(0, 1, 0, 0))
```

`true`

DateTime.IsInPreviousMinute

2020/4/30 •

语法

```
DateTime.IsInPreviousMinute(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于上一分钟内。注意，如果传递的值处于当前分钟内，此函数将返回 false。

- `dateTime`：要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间之前的一分钟是否处于前一分钟内。

```
DateTime.IsInPreviousMinute(DateTime.FixedLocalNow() - #duration(0, 0, 1, 0))
```

`true`

DateTime.IsInPreviousNHours

2020/4/30 •

语法

```
DateTime.IsInPreviousNHours(dateTime as any, hours as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于之前的小时数内。注意，如果传递的值为当前这一小时的时间，此函数将返回 false。

- `dateTime`：要计算的 `datetime` 或 `datetimezone` 值。
- `hours`：小时数。

示例 1

确定当前系统时间前的那一小时是否处于前两个小时内。

```
DateTime.IsInPreviousNHours(DateTime.FixedLocalNow() - #duration(0, 2, 0, 0), 2)
```

`true`

DateTime.IsInPreviousNMinutes

2020/4/30 •

语法

```
DateTime.IsInPreviousNMinutes(dateTime as any, minutes as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于之前的分钟数内。注意，如果传递的值处于当前分钟内，此函数将返回 false。

- `dateTime`：要计算的 `datetime` 或 `datetimezone` 值。
- `minutes`：分钟数。

示例 1

确定当前系统时间前的分钟是否处于前两分钟内。

```
DateTime.IsInPreviousNMinutes(DateTime.FixedLocalNow() - #duration(0, 0, 2, 0), 2)
```

`true`

DateTime.IsInPreviousNSeconds

2020/4/30 •

语法

```
DateTime.IsInPreviousNSeconds(dateTime as any, seconds as number) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否按系统当前日期和时间所确定的那样处于之前的秒数内。注意，如果传递的值为当前这一秒的时间，此函数将返回 false。

- `dateTime`：要计算的 `datetime` 或 `datetimezone` 值。
- `seconds`：秒数。

示例 1

确定当前系统时间之前的那一秒是否处于前两秒内。

```
DateTime.IsInPreviousNSeconds(DateTime.FixedLocalNow() - #duration(0, 0, 0, 2), 2)
```

`true`

DateTime.IsInPreviousSecond

2020/4/30 •

语法

```
DateTime.IsInPreviousSecond(dateTime as any) as nullable logical
```

关于

指示给定的日期/时间值 `dateTime` 是否处于前一秒内(由系统上的当前日期和时间确定)。注意，如果传递的值为当前这一秒的时间，此函数将返回 false。

- `dateTime` :要计算的 `datetime` 或 `datetimezone` 值。

示例 1

确定当前系统时间前的一秒是否处于前一秒内。

```
DateTime.IsInPreviousSecond(DateTime.FixedLocalNow() - #duration(0, 0, 0, 1))
```

`true`

DateTime.LocalNow

2019/12/3 •

语法

```
DateTime.LocalNow() as datetime
```

关于

返回设置为系统上的当前日期和时间的 `datetime` 值。

DateTime.Time

2019/12/3 •

语法

```
DateTime.Time(dateTime as any) as nullable time
```

关于

返回给定日期/时间值 `dateTime` 的时间部分。

示例 1

计算 #datetime(2010, 12, 31, 11, 56, 02) 的时间值。

```
DateTime.Time(#datetime(2010, 12, 31, 11, 56, 02))
```

`#time(11, 56, 2)`

DateTime.ToRecord

2019/12/3 •

语法

```
DateTime.ToRecord(dateTime as datetime) as record
```

关于

返回包含给定日期/时间值 `dateTime` 的各个部分的记录。

- `dateTime`：用于计算其部分的记录的 `datetime` 值。

示例 1

将 `#datetime(2011, 12, 31, 11, 56, 2)` 值转换为包含日期和时间值的记录。

```
DateTime.ToRecord(#datetime(2011, 12, 31, 11, 56, 2))
```

"	2011
"	12
"	31
"	11
"	56
"	2

DateTime.ToText

2019/12/4 •

语法

```
DateTime.ToText(dateTime as nullable datetime, optional format as nullable text, optional culture
as nullable text) as nullable text
```

关于

返回 `dateTime`（即日期/时间值 `dateTime`）的文本表示形式。此函数采用可选格式参数 `format`。有关所支持格式的完整列表，请参阅库规范文档。

示例 1

获取 #datetime(2011, 12, 31, 11, 56, 2) 的文本表示形式。

```
DateTime.ToText(#datetime(2010, 12, 31, 11, 56, 2))
```

```
"12/31/2010 11:56:02 AM"
```

示例 2

使用格式选项获取 #datetime(2011, 12, 31, 11, 56, 2) 的文本表示形式。

```
DateTime.ToText(#datetime(2010, 12, 31, 11, 56, 2), "yyyy/MM/ddThh:mm:ss")
```

```
"2010/12/31T11:56:02"
```

#datetime

2019/12/4 •

语法

```
#datetime(year as number, month as number, day as number, hour as number, minute as number, second as number) as any
```

关于

从整数年 `year`、月 `month`、日 `day`、小时 `hour`、分钟 `minute` 和(小数)秒 `second` 创建日期/时间值。如果不满足以下条件，则会引发错误：

- 1 ≤ year ≤ 9999
- 1 ≤ month ≤ 12
- 1 ≤ day ≤ 31
- 0 ≤ hour ≤ 23
- 0 ≤ minute ≤ 59
- 0 ≤ second ≤ 59

日期/时间/时区函数

2020/4/26 •

这些函数创建并操纵 datetimezone 值。

DateTimeZone

"	"
DateTimeZone.FixedLocalNow	返回一个 DateTimeZone 值，该值设置为系统上的当前日期、时间和时区偏移量。
DateTimeZone.FixedUtcNow	返回采用 UTC（GMT 时区）表示的当前日期和时间。
DateTimeZone.From	从给定值返回 datetimezone 值。
DateTimeZone.FromFileTime	从数值返回 DateTimeZone。
DateTimeZone.FromText	从一组日期格式和区域性值返回 DateTimeZone 值。
DateTimeZone.LocalNow	返回设置为当前系统日期和时间的 DateTime 值。
DateTimeZone.RemoveZone	返回从输入的 datetimezone 值中删除时区信息后得到的日期时间值。
DateTimeZone.SwitchZone	更改输入 DateTimeZone 的时区信息。
DateTimeZone.ToLocal	从本地时区返回 DateTime 值。
DateTimeZone.ToRecord	返回包含 DateTime 值的各个部分的记录。
DateTimeZone.ToText	从 DateTime 值返回文本值。
DateTimeZone.ToUtc	返回将 DateTime 值转换到 UTC 时区后所得的值。
DateTimeZone.UtcNow	返回设置当前系统日期和 Utc 时区的时间的 DateTime 值。
DateTimeZone.ZoneHours	从 DateTime 值返回时区小时值。
DateTimeZone.ZoneMinutes	从 DateTime 值返回时区分钟值。
#datetimezone	从年、月、日、小时、分钟、秒、偏移小时和偏移分钟创建一个 datetimezone 值。

DateTimeZone.FixedLocalNow

2019/12/3 •

语法

```
DateTimeZone.FixedLocalNow() as datetimezone
```

关于

返回设置为系统上的当前日期和时间的 `datetime` 值。返回的值包含表示当地时区的时区信息。此值是固定的，因此将不会随着连续调用而更改，这与 DateTimeZone.LocalNow 不同，后者可能会在表达式的执行过程中返回不同值。

DateTimeZone.FixedUtcNow

2019/12/3 •

语法

```
DateTimeZone.FixedUtcNow() as datetimezone
```

关于

返回采用 UTC（GMT 时区）表示的当前日期和时间。此值为固定值，不会随连续调用而更改。

DateTimeZone.From

2019/12/3 •

语法

```
DateTimeZone.From(value as any, optional culture as nullable text) as nullable datetimezone
```

关于

从给定的 `value` 返回 `datetimezone` 值。如果给定的 `value` 为 `null`，则 `DateTimeZone.From` 返回 `null`。如果给定的 `value` 为 `datetimezone`，则返回 `value`。可以将以下类型的值转换为 `datetimezone` 值：

- `text`：文本表示形式的 `datetimezone` 值。有关详细信息，请参阅 `DateTimeZone.FromText`。
- `date`：一个 `datetimezone`，日期部分为 `value`，时间部分为 `12:00:00 AM`，其偏移量对应于本地时区。
- `datetime`：以 `value` 为日期时间的 `datetimezone`，其偏移量对应于本地时区。
- `time`：一个 `datetimezone`，其日期等同于以 `0` 作为日期部分，以 `value` 作为时间部分的 OLE 自动化日期，其偏移量对应于本地时区。
- `number`：一个 `datetimezone`，其日期时间等同于用 `value` 表示的 OLE 自动化日期，其偏移量对应于本地时区。

如果 `value` 为任何其他类型，则返回错误。

示例 1

将 `"2020-10-30T01:30:00-08:00"` 转换为 `datetimezone` 值。

```
DateTimeZone.From("2020-10-30T01:30:00-08:00")
```

```
#datetimezone(2020, 10, 30, 01, 30, 00, -8, 00)
```

DateTimeZone.FromFileTime

2019/12/4 •

语法

```
DateTimeZone.FromFileTime(fileTime as nullable number) as nullable datetimezone
```

关于

根据 `fileTime` 值创建 `datetimezone` 值，并将其转换为本地时区。fileTime 是一个 Windows 文件时间值，表示自公元 1601 年 1 月 1 日午夜 12:00 开始经过的 100 纳秒间隔数 。协调世界时 (UTC)。

示例 1

将 `129876402529842245` 转换为 datetimezone 值。

```
DateTimeZone.FromFileTime(129876402529842245)
```

```
#datetimezone(2012, 7, 24, 14, 50, 52.9842245, -7, 0)
```

DateTimeZone.FromText

2019/12/4 •

语法

```
DateTimeZone.FromText(text as nullable text, optional culture as nullable text) as nullable
datetimezone
```

关于

根据 ISO 8601 格式标准，从文本表示形式 `text` 创建 `datetimezone` 值。

- `DateTimeZone.FromText("2010-12-31T01:30:00-08:00")` // yyyy-MM-ddThh:mm:ssZ

示例 1

将 `"2010-12-31T01:30:00-08:00"` 转换为 datetimezone 值。

```
DateTimeZone.FromText("2010-12-31T01:30:00-08:00")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0, -8, 0)
```

示例 2

将 `"2010-12-31T01:30:00.121212-08:00"` 转换为 datetimezone 值。

```
DateTimeZone.FromText("2010-12-31T01:30:00.121212-08:00")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0.121212, -8, 0)
```

示例 3

将 `"2010-12-31T01:30:00Z"` 转换为 datetimezone 值。

```
DateTimeZone.FromText("2010-12-31T01:30:00Z")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0, 0, 0)
```

示例 4

将 `"20101231T013000+0800"` 转换为 datetimezone 值。

```
DateTimeZone.FromText("20101231T013000+0800")
```

```
#datetimezone(2010, 12, 31, 1, 30, 0, 8, 0)
```

DateTimeZone.LocalNow

2019/12/3 •

语法

```
DateTimeZone.LocalNow() as datetimezone
```

关于

返回设置为系统上的当前日期和时间的 `datetimezone` 值。返回的值包含表示当地时区的时区信息。

DateTimeZone.RemoveZone

2020/4/30 •

语法

```
DateTimeZone.RemoveZone(dateTimeZone as nullable datetimezone) as nullable datetime
```

关于

从 `dateTimeZone` 返回 #datetime 值并删除其中的时区信息。

示例 1

从值 #datetimezone(2011, 12, 31, 9, 15, 36, -7, 0) 中删除时区信息。

```
DateTimeZone.RemoveZone(#datetimezone(2011, 12, 31, 9, 15, 36, -7, 0))
```

```
#datetime(2011, 12, 31, 9, 15, 36)
```

DateTimeZone.SwitchZone

2019/12/3 •

语法

```
DateTimeZone.SwitchZone(dateTimeZone as nullable datetimezone, timezoneHours as number, optional
timezoneMinutes as nullable number) as nullable datetimezone
```

关于

将 datetimezone 值 `dateTimeZone` 的时区信息更改为 `timezoneHours` 和 `timezoneMinutes`（可选）提供的新时区信息。如果 `dateTimeZone` 没有时区部分，则会引发异常。

示例 1

将 #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) 的时区信息更改为 8 小时。

```
DateTimeZone.SwitchZone(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30), 8)
```

```
#datetimezone(2010, 12, 31, 12, 26, 2, 8, 0)
```

示例 2

将 #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) 的时区信息更改为 -30 分钟。

```
DateTimeZone.SwitchZone(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30), 0, -30)
```

```
#datetimezone(2010, 12, 31, 3, 56, 2, 0, -30)
```

DateTimeZone.ToLocal

2019/12/3 •

语法

```
DateTimeZone.ToLocal(dateTimeZone as nullable datetimezone) as nullable datetimezone
```

关于

将 datetimezone 值 `dateTimeZone` 的时区信息更改为本地时区信息。如果 `dateTimeZone` 没有时区部分，则添加本地时区信息。

示例 1

将针对 #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) 的时区信息更改为本地时区（假定为 PST）。

```
DateTimeZone.ToLocal(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30))
```

```
#datetimezone(2010, 12, 31, 12, 26, 2, -8, 0)
```

DateTimeZone.ToRecord

2019/12/3 •

语法

```
DateTimeZone.ToRecord(dateTimeZone as datetimezone) as record
```

关于

返回包含给定 datetimezone 值 `dateTimeZone` 的各个部分的记录。

- `dateTimeZone`：用于计算其部分的记录的 `datetimezone` 值。

示例 1

将 `#datetimezone(2011, 12, 31, 11, 56, 2, 8, 0)` 值转换为包含日期、时间和时区值的记录。

```
DateTimeZone.ToRecord(#datetimezone(2011, 12, 31, 11, 56, 2, 8, 0))
```

[illegible]	2011
[illegible]	12
[illegible]	31
[illegible]	11
[illegible]	56
[illegible]	2
ZONEHOURS	8
ZONEMINUTES	0

DateTimeZone.ToText

2019/12/4 •

语法

```
DateTimeZone.ToText(dateTimeZone as nullable datetimezone, optional format as nullable text,
optional culture as nullable text) as nullable text
```

关于

返回 `dateTimeZone`（即 datetimezone 值 `dateTimeZone`）的文本表示形式。此函数采用可选格式参数 `format`。有关所支持格式的完整列表，请参阅库规范文档。

示例 1

获取 #datetimezone(2011, 12, 31, 11, 56, 2, 8, 0) 的文本表示形式。

```
DateTimeZone.ToText(#datetimezone(2010, 12, 31, 11, 56, 2, 8, 0))
```

```
"12/31/2010 11:56:02 AM +08:00"
```

示例 2

使用格式选项获取 #datetimezone(2010, 12, 31, 11, 56, 2, 10, 12) 的文本表示形式。

```
DateTimeZone.ToText(#datetimezone(2010, 12, 31, 11, 56, 2, 10, 12), "yyyy/MM/ddThh:mm:sszzz")
```

```
"2010/12/31T11:56:02+10:12"
```

DateTimeZone.ToUtc

2019/12/3 •

语法

```
DateTimeZone.ToUtc(dateTimeZone as nullable datetimezone) as nullable datetimezone
```

关于

将日期时间值 `dateTimeZone` 的时区信息更改为 UTC 或通用时间时区信息。如果 `dateTimeZone` 没有时区部分，则添加 UTC 时区信息。

示例 1

将 #datetimezone(2010, 12, 31, 11, 56, 02, 7, 30) 的时区信息更改为 UTC 时区。

```
DateTimeZone.ToUtc(#datetimezone(2010, 12, 31, 11, 56, 02, 7, 30))
```

```
#datetimezone(2010, 12, 31, 4, 26, 2, 0, 0)
```

DateTimeZone.UtcNow

2019/12/4 •

语法

```
DateTimeZone.UtcNow() as datetimezone
```

关于

返回采用 UTC（GMT 时区）表示的当前日期和时间。

示例 1

获取采用 UTC 表示的当前日期和时间。

```
DateTimeZone.UtcNow()
```

```
#datetimezone(2011, 8, 16, 23, 34, 37.745, 0, 0)
```

DateTimeZone.ZoneHours

2019/12/4 •

语法

```
DateTimeZone.ZoneHours(dateTimeZone as nullable datetimezone) as nullable number
```

关于

更改值的时区。

DateTimeZone.ZoneMinutes

2019/12/4 •

语法

```
DateTimeZone.ZoneMinutes(dateTimeZone as nullable datetimezone) as nullable number
```

关于

更改值的时区。

#datetimezone

2019/12/4 •

语法

```
#datetimezone(year as number, month as number, day as number, hour as number, minute as number,
second as number, offsetHours as number, offsetMinutes as number) as any
```

关于

从整数年 `year`、月 `month`、日 `day`、小时 `hour`、分钟 `minute`、(小数)秒 `second`、(小数)偏移小时 `offsetHours`，以及偏移分钟 `offsetMinutes` 创建一个 datetimezone 值。如果不满足以下条件，则会引发错误：

- 1 ≤ year ≤ 9999
- 1 ≤ month ≤ 12
- 1 ≤ day ≤ 31
- 0 ≤ hour ≤ 23
- 0 ≤ minute ≤ 59
- 0 ≤ second ≤ 59
- -14 ≤ 偏移小时 + 偏移分钟 / 60 ≤ 14

持续时间函数

2020/4/26 •

这些函数创建并操纵持续时间值。

持续时间

"	"
Duration.Days	返回持续时间值的日部分。
Duration.From	从值中返回持续时间值。
Duration.FromText	从文本值中返回持续时间值。
Duration.Hours	返回持续时间值的小时部分。
Duration.Minutes	返回持续时间值的分钟部分。
Duration.Seconds	返回持续时间值的秒部分。
Duration.ToRecord	返回带有部分持续时间值的记录。
Duration.TotalDays	返回持续时间值的总天数幅度。
Duration.TotalHours	返回持续时间值的总小时数幅度。
Duration.TotalMinutes	返回持续时间值的总分钟数幅度。
Duration.TotalSeconds	返回持续时间值的总秒数幅度。
Duration.ToText	从持续时间值中返回文本值。
#duration	从日、小时、分钟和秒创建一个持续时间值。

Duration.Days

2019/12/4 •

语法

```
Duration.Days(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值的天数部分，`duration`。

示例 1

查找 #duration(5, 4, 3, 2) 中的天数。

```
Duration.Days(#duration(5, 4, 3, 2))
```

`5`

Duration.From

2019/12/4 •

语法

```
Duration.From(value as any) as nullable duration
```

关于

从给定的 `value` 返回 `duration` 值。如果给定的 `value` 为 `null`，则 `Duration.From` 返回 `null`。如果给定的 `value` 为 `duration`，则返回 `value`。可以将以下类型的值转换为 `duration` 值：

- `text`：文本形式的占用时间格式 (d.h:m:s) 中的 `duration` 值。有关详细信息，请参阅 `Duration.FromText`。
- `number`：与 `value` 表示的整数和小数天数等效的 `duration`。

如果 `value` 为任何其他类型，则返回错误。

示例 1

将 `2.525` 转换为 `duration` 值。

```
Duration.From(2.525)
```

```
#duration(2, 12, 36, 0)
```

Duration.FromText

2019/12/3 •

语法

```
Duration.FromText(text as nullable text) as nullable duration
```

关于

从指定的文本 `text` 返回持续时间值。此函数可以分析以下格式：

- (-)hh:mm(:ss(.ff))
- (-)ddd(.hh:mm(:ss(.ff)))

（包含所有范围）
ddd：天数。
hh：小时数，介于 0 到 23 之间。
mm：分钟数，介于 0 到 59 之间。
ss：秒数，介于 0 到 59 之间。
ff：秒的小数部分，介于 0 到 9999999 之间。

示例 1

将 `"2.05:55:20"` 转换为 `duration` 值。

```
Duration.FromText("2.05:55:20")
```

```
#duration(2, 5, 55, 20)
```

Duration.Hours

2019/12/4 •

语法

```
Duration.Hours(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值的小时部分，`duration`。

示例 1

查找 #duration(5, 4, 3, 2) 中的小时数。

```
Duration.Hours(#duration(5, 4, 3, 2))
```

`4`

Duration.Minutes

2019/12/4 •

语法

```
Duration.Minutes(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值的分钟部分，`duration`。

示例 1

查找 #duration(5, 4, 3, 2) 中的分钟数。

```
Duration.Minutes(#duration(5, 4, 3, 2))
```

`3`

Duration.Seconds

2019/12/3 •

语法

```
Duration.Seconds(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值的秒数部分，`duration`。

示例 1

查找 #duration(5, 4, 3, 2) 中的秒数。

```
Duration.Seconds(#duration(5, 4, 3, 2))
```

`2`

Duration.ToRecord

2019/12/3 •

语法

```
Duration.ToRecord(duration as duration) as record
```

关于

返回包含持续时间值 `duration` 的各个部分的记录。

- `duration`：从中创建记录的 `duration`。

示例 1

将 `#duration(2, 5, 55, 20)` 转换为包括天、小时、分钟和秒这些部分的记录（如果适用）。

```
Duration.ToRecord(#duration(2, 5, 55, 20))
```

[redacted]	2
[redacted]	5
[redacted]	55
[redacted]	20

Duration.TotalDays

2019/12/4 •

语法

```
Duration.TotalDays(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值 `duration` 跨越的总天数。

示例 1

计算 #duration(5, 4, 3, 2) 中跨越的总天数。

```
Duration.TotalDays(#duration(5, 4, 3, 2))
```

```
5.1687731481481478
```

Duration.TotalHours

2019/12/4 •

语法

```
Duration.TotalHours(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值 `duration` 跨越的总小时数。

示例 1

查找 #duration(5, 4, 3, 2) 中跨越的总小时数。

```
Duration.TotalHours(#duration(5, 4, 3, 2))
```

```
124.05055555555555
```

Duration.TotalMinutes

2019/12/3 •

语法

```
Duration.TotalMinutes(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值 `duration` 跨越的总分钟数。

示例 1

计算 #duration(5, 4, 3, 2) 中跨越的总分钟数。

```
Duration.TotalMinutes(#duration(5, 4, 3, 2))
```

`7443.0333333333338`

Duration.TotalSeconds

2019/12/3 •

语法

```
Duration.TotalSeconds(duration as nullable duration) as nullable number
```

关于

返回所提供的 `duration` 值 `duration` 跨越的总秒数。

示例 1

计算 #duration(5, 4, 3, 2) 中跨越的总秒数。

```
Duration.TotalSeconds(#duration(5, 4, 3, 2))
```

```
446582
```

Duration.ToText

2019/12/4 •

语法

```
Duration.ToText(duration as nullable duration, optional format as nullable text) as nullable text
```

关于

以 "day.hour:mins:sec" 格式返回给定持续时间值 `duration` 的文本表示形式。指定格式的文本值可作为可选的第二个参数 `format` 提供。

- `duration`：要计算其文本表示形式的 `duration`。
- `format`：[可选] 指定格式的 `text` 值。

示例 1

将 `#duration(2, 5, 55, 20)` 转换为文本值。

```
Duration.ToText(#duration(2, 5, 55, 20))
```

`"2.05:55:20"`

#duration

2019/12/4 •

语法

```
#duration(days as number, hours as number, minutes as number, seconds as number) as duration
```

关于

使用数字天数 `days` 、小时 `hours` 、分钟 `minutes` 和秒 `seconds` 创建一个持续时间值。

错误处理

2020/4/26 •

这些函数返回详细程度不同的诊断跟踪，并引发错误记录。

错误

"	"
Diagnostics.ActivityId	为当前正在运行的计算返回不透明的标识符。
Diagnostics.Trace	写入跟踪消息（如果已启用跟踪）并返回值。
Error.Record	返回一个记录，其中包含设置为所提供值的“原因”、“消息”、“详细信息”字段。该记录可用于引发错误。
TraceLevel.Critical	返回严重跟踪级别值 1。
TraceLevel.Error	返回错误跟踪级别值 2。
TraceLevel.Information	返回信息跟踪级别值 4。
TraceLevel.Verbose	返回详细跟踪级别值 5。
TraceLevel.Warning	返回警告跟踪级别值 3。

Diagnostics.ActivityId

2019/12/4 •

`

语法

```
Diagnostics.ActivityId() as nullable text
```

关于

为当前正在运行的计算返回不透明的标识符。

Diagnostics.Trace

2019/12/3 •

语法

```
Diagnostics.Trace(traceLevel as number, message as anynonnull, value as any, optional delayed as nullable logical) as any
```

关于

写入跟踪 `message`（如果已启用跟踪的话）并返回 `value`。可选参数 `delayed` 指定是否延迟 `value` 的计算，直到跟踪到消息。 `traceLevel` 可以采取以下值之一：

- `TraceLevel.Critical` - `TraceLevel.Error`

- `TraceLevel.Warning`
- `TraceLevel.Information`
- `TraceLevel.Verbose`

示例 1

在调用 Text.From 函数前跟踪消息，并返回结果。

```
Diagnostics.Trace(TraceLevel.Information, "TextValueFromNumber", () => Text.From(123), true)
```

`"123"`

Error.Record

2019/12/4 •

语法

```
Error.Record(reason as text, optional message as nullable text, optional detail as any) as record
```

关于

从为原因、消息和详细信息提供的文本值返回错误记录。

TraceLevel.Critical

2019/12/3 •

关于

返回严重跟踪级别值 1。

TraceLevel.Error

2019/12/3 •

关于

返回错误**跟踪**级别值 2。

TraceLevel.Information

2019/12/3 •

关于

返回信息跟踪级别值 4。

TraceLevel.Verbose

2019/12/4 •

关于

返回详细**跟踪**级别值 5。

TraceLevel.Warning

2019/12/4 •

关于

返回警告跟踪级别值 3。

表达式函数

2020/4/26 •

这些函数允许构造和评估 M 代码。

表达式

"	"
Expression.Constant	返回常数值的 M 源代码表示形式。
Expression.Evaluate	返回 M 表达式的计算结果。
Expression.Identifier	返回标识符的 M 源代码表示形式。

Expression.Constant

2019/12/4 •

语法

```
Expression.Constant(value as any) as text
```

关于

返回常数值的 M 源代码表示形式。

示例 1

获取数字值的 M 源代码表示形式。

```
Expression.Constant(123)
```

```
"123"
```

示例 2

获取日期值的 M 源代码表示形式。

```
Expression.Constant(#date(2035, 01, 02))
```

```
"#date(2035, 1, 2)"
```

示例 3

获取文本值的 M 源代码表示形式。

```
Expression.Constant("abc")
```

```
"""abc"""
```

Expression.Evaluate

2020/4/30 •

语法

```
Expression.Evaluate(document as text, optional environment as nullable record) as any
```

关于

返回 M 表达式 `document` 的计算结果，其中可用的标识符可以由 `environment` 进行引用和定义。

示例 1

计算简单求和。

```
Expression.Evaluate("1 + 1")
```

`2`

示例 2

计算更复杂的求和。

```
Expression.Evaluate("List.Sum({1, 2, 3})", [List.Sum = List.Sum])
```

`6`

示例 3

计算含标识符的文本值的串联。

```
Expression.Evaluate(Expression.Constant("""abc") & " & " & Expression.Identifier("x"), [x = "def"""])
```

`"""abcdef"""`

Expression.Identifier

2019/12/4 •

语法

```
Expression.Identifier(name as text) as text
```

关于

返回标识符 `name` 的 M 源代码表示形式。

示例 1

获取标识符的 M 源代码表示形式。

```
Expression.Identifier("MyIdentifier")
```

`"MyIdentifier"`

示例 2

获取包含空格的标识符的 M 源代码表示形式。

```
Expression.Identifier("My Identifier")
```

`"#""My Identifier"""`

函数值

2020/4/26 •

这些函数创建并调用其他 M 函数。

函数

"	"
Function.From	采用一元函数 `function` 并创建一个类型为 `functionType` 的新函数，用于构造其参数列表，并将其传递给 `function`。
Function.Invoke	使用指定内容调用给定的函数并返回结果。
Function.InvokeAfter	经过一段持续延迟后，返回调用函数的结果。
Function.IsDataSource	返回是否将函数视为数据源。
Function.ScalarVector	返回 scalarFunctionType 类型的标量函数，此函数使用单行参数调用 vectorFunction 并返回其单个输出。

Function.From

2019/12/3 •

语法

```
Function.From(functionType as type, function as function) as function
```

关于

采用一元函数 `function` 并创建一个类型为 `functionType` 的新函数，用于构造其参数列表，并将其传递给 `function`。

示例 1

将 List.Sum 转换为一个包含两个参数的函数，其参数捆绑添加。

```
Function.From(type function (a as number, b as number) as number, List.Sum)(2, 1)
```

`3`

示例 2

将接受列表的函数转换为双参数函数。

```
Function.From(type function (a as text, b as text) as text, (list) => list{0} & list{1})("2", "1")
```

`"21"`

Function.Invoke

2020/4/30 •

语法

```
Function.Invoke(function as function, args as list) as any
```

关于

使用指定的参数列表调用给定的函数并返回结果。

示例 1

使用一个参数 [A=1,B=2] 调用 Record.FieldNames

```
Function.Invoke(Record.FieldNames, {[A = 1, B = 2]}
```

A
B

Function.InvokeAfter

2019/12/4 •

语法

```
Function.InvokeAfter(function as function, delay as duration) as any
```

关于

经过持续时间 `delay` 后，返回调用 `function` 的结果。

Function.IsDataSource

2019/12/4 •

语法

```
Function.IsDataSource(function as function) as logical
```

关于

返回是否将 `function` 视为数据源。

Function.ScalarVector

2019/12/3 •

语法

```
Function.ScalarVector(scalarFunctionType as type, vectorFunction as function) as function
```

关于

返回 `scalarFunctionType` 类型的标量函数，该函数使用单行参数调用 `vectorFunction` 并返回其单个输出。此外，如果对输入表中的每一行重复应用标量函数（如 Table.AddColumn），则将对所有输入应用一次 `vectorFunction`。

会向 `vectorFunction` 传递一个表，该表的列名称和列位置与 `scalarFunctionType` 的参数匹配。此表中的每一行都包含调用一次标量函数的参数，其中的列对应于 `scalarFunctionType` 的参数。

`vectorFunction` 必须返回与输入表长度相同的列表，其中每个位置的项必须是计算同一位置输入行上的标量函数得出的相同结果。

输入表预期采用流式处理方式，`vectorFunction` 将在输入传入时流式传输其输出，并且一次仅处理一个输入区块。具体而言，`vectorFunction` 不得多次枚举其输入表。

行函数

2020/4/26 •

这些函数在二进制和单个 text 值之间来回转换文本列表。

Lines

"	"
Lines.FromBinary	将二进制值转换为在换行符处拆分的文本值列表。
Lines.FromText	将文本值转换为在换行符处拆分的文本值列表。
Lines.ToBinary	使用指定的编码和 lineSeparator 将文本列表转换为二进制值。指定的 lineSeparator 追加到每行之后。如果未指定，则会使用回车和换行字符。
Lines.ToText	将文本列表转换成单个文本。指定的 lineSeparator 将追加到每一行之后。如果未指定，则会使用回车和换行字符。

Lines.FromBinary

2019/12/4 •

语法

```
Lines.FromBinary(binary as binary, optional quoteStyle as nullable number, optional
includeLineSeparators as nullable logical, optional encoding as nullable number) as list
```

关于

将二进制值转换为在换行符处拆分的文本值列表。如果指定了引号样式，则换行符可能出现在引号内。如果 includeLineSeparators 为 true，则文本中包含换行符。

Lines.FromText

2019/12/3 •

语法

```
Lines.FromText(text as text, optional quoteStyle as nullable number, optional
includeLineSeparators as nullable logical) as list
```

关于

将文本值转换为在换行符处拆分的文本值列表。如果 includeLineSeparators 为 true，则文本中包含换行符。

- `QuoteStyle.None:`（默认）不需要引用行为。
- `QuoteStyle.Csv:` 按 Csv 引用。单个双引号字符用于划分此类区域，一对双引号字符用于指示此类区域中的单个双引号字符。

Lines.ToBinary

2019/12/3 •

语法

```
Lines.ToBinary(lines as list, optional lineSeparator as nullable text, optional encoding as
nullable number, optional includeByteOrderMark as nullable logical) as binary
```

关于

使用指定的编码和 lineSeparator 将文本列表转换为二进制值。指定的 lineSeparator 追加到每行之后。如果未指定，则会使用回车和换行字符。

Lines.ToText

2019/12/3 •

语法

```
Lines.ToText(lines as list, optional lineSeparator as nullable text) as text
```

关于

将文本列表转换成单个文本。指定的 lineSeparator 将追加到每一行之后。如果未指定，则会使用回车和换行字符。

列表函数

2020/4/26 •

这些函数创建并操纵列表值。

信息

"	"
List.Count	返回列表中的项数。
List.NonNullCount	返回列表中不包括 null 值的项数
List.IsEmpty	返回列表是否为空。

选择

"	"
List.Alternate	返回列表，其中包含来自原始列表的基于 count、可选的 repeatInterval 和可选的 offset 交替变化的项。
List.Buffer	在内存中缓冲列表。此调用的结果是一个稳定列表，这意味着它将具有确定的项计数和顺序。
List.Distinct	通过删除重复项来向下筛选列表。可以指定可选的相等条件值来控制相等比较。选择每个相等组中的第一个值。
List.FindText	在包括记录字段在内的值列表中搜索文本值。
List.First	返回列表的第一个值；如果为空，则返回指定的默认值。返回列表的第一项；如果列表为空，则返回可选的默认值。如果列表为空且未指定默认值，则函数返回。
List.FirstN	通过指定要返回的项数或 countOrCondition 提供的限定条件来返回列表中的第一组项 。
List.InsertRange	从输入列表中给定索引处的值插入项。
List.IsDistinct	返回列表是否不同。
List.Last	通过指定要返回的项数或 countOrCondition 提供的限定条件来返回列表中的最后一组项 。
List.LastN	通过指定要返回的项数或指定限定条件来返回列表中的最后一组项。
List.MatchesAll	如果列表中的所有项均满足某一条件，则返回 true。
List.MatchesAny	如果列表中有任何项满足某一条件，则返回 true。

List.Positions	返回输入列表的位置列表。
List.Range	返回从偏移量开始的计数项。
List.Select	选择与某一条件匹配的项。
List.Single	返回列表中的单个项，或者如果列表具有多个项，则会引发 Expression.Error。
List.SingleOrDefault	返回列表中的单个项。
List.Skip	跳过列表的第一项。如果给定空列表，则会返回空列表。此函数采用可选参数 countOrCondition 来支持跳过多个值。

转换函数

List.Accumulate	从列表中累积某一结果。从初始值种子开始，此函数应用累加器函数并返回最终结果。
List.Combine	将包含列表的一个列表合并为单个列表。
List.RemoveRange	返回删除以偏移量开头的计数项后的列表。默认计数为 1。
List.RemoveFirstN	返回一个列表，其中，从第一个元素开始指定数量的元素从列表中被删除了。被删除的元素数取决于可选的 countOrCondition 参数。
List.RemoveItems	从 list1 中删除在 list2 中出现的项，并返回新的列表。
List.RemoveLastN	返回一个列表，其中，从最后一个元素开始，指定数量的元素从列表中被删除了。被删除的元素数取决于可选的 countOrCondition 参数。
List.Repeat	返回一个列表，此列表重复输入列表计数次数的内容。
List.ReplaceRange	返回一个列表，此列表从某一索引处开始将列表中的计数值替换为 replaceWith 列表。
List.RemoveMatchingItems	删除列表中出现的所有给定值。
List.RemoveNulls	从列表中删除 null 值。
List.ReplaceMatchingItems	使用所提供的 equationCriteria 将列表中现有值的匹配项替换为新值。旧值和新值由替换参数提供。可以指定可选相等条件值来控制相等比较。有关替换操作和相等条件的详细信息，请参阅参数值。
List.ReplaceValue	在值列表中搜索值，并将每一匹配项替换为替换值。
List.Reverse	返回一个列表，此列表将列表中的项按相反顺序排列。

"	"
List.Split	使用指定页面大小将指定列表拆分为包含列表的一个列表。
List.Transform	对列表中的每一项执行此函数并返回新列表。
List.TransformMany	返回一个列表，其元素是基于输入列表投射而来的。

成员身份函数

由于可以对所有值进行相等性测试，因此这些函数可以对异类列表执行操作。

"	"
List.AllTrue	如果列表中的所有表达式均为 true，则返回 true
List.AnyTrue	如果列表中有任何表达式为 true，则返回 true
List.Contains	如果在列表中找到值，则返回 true。
List.ContainsAll	如果在列表中找到值的所有项，则返回 true。
List.ContainsAny	如果在列表中找到值的任一项，则返回 true。
List.PositionOf	查找某个值在列表中的第一个匹配项，并返回其位置。
List.PositionOfAny	查找任何值在列表中的第一个匹配项，并返回其位置。

设置操作

"	"
List.Difference	返回列表 1 中未出现在列表 2 中的项。支持重复值。
List.Intersect	从包含列表的一个列表中返回一个列表，并与单个列表中的常见项相交。支持重复值。
List.Union	从包含列表的一个列表中返回一个列表，并与单个列表中的项联合。返回的列表包含任意输入列表中的所有项。重复值会匹配为联合的一部分。
List.Zip	返回包含在相同位置组合项的列表的一个列表。

排序

排序函数执行比较操作。所有被比较的值都必须互相进行比较。这意味着这些值必须采用相同的数据类型（或包含始终经比较为最小的 null）。否则，将引发 Expression.Error。

可比较的数据类型

- 数字
- 持续时间
- DateTime
- 文本

- 逻辑
- Null

List.Max	返回列表中的最大项；如果列表为空，则返回可选默认值。
List.MaxN	返回列表中的最大值。对行进行排序之后，可以指定可选参数以进一步筛选结果
List.Median	返回列表中的中位数项。
List.Min	返回列表中的最小项；如果列表为空，则返回可选默认值。
List.MinN	返回列表中的最小值。
List.Sort	使用比较条件返回已排序的列表。

平均值

这些函数对数字、日期/时间和持续时间的同类列表执行操作。

List.Average	以列表中值的数据类型从列表中返回一个平均值。
List.Mode	返回最常出现在列表中的项。
List.Modes	返回以相同的最大频率出现的所有项。
List.StandardDeviation	从值列表中返回标准偏差。List.StandardDeviation 执行基于示例的估计。结果是用于数字的一个数字以及用于日期/时间和持续时间的一个持续时间。

相加

这些函数适用于数字或持续时间的同类列表。

List.Sum	返回列表的总和。

数值

这些函数仅适用于数字。

List.Covariance	以数字形式返回两个列表的协方差。
List.Product	从数字列表返回乘积。

生成器

这些函数会生成值列表。

List.Dates	从起点开始，根据大小计数返回日期值列表，并将每个值加上一个增量。
List.DateTimes	从起点开始，根据大小计数返回日期/时间值列表，并将每个值加上一个增量。
List.DateTimeZones	从起点开始，根据大小计数返回时区值列表，并将每个值加上一个增量。
List.Durations	从起点开始，根据大小计数返回持续时间值列表，并将每个值加上一个增量。
List.Generate	从值函数、条件函数、next 函数和值的可选转换函数生成列表。
List.Numbers	从初始处开始，根据大小计数返回数字列表，并加上一个增量。增量默认为 1。
List.Random	返回计数随机数列表，其中包含可选的种子参数。
List.Times	从起点开始，根据大小计数返回时间值列表。

参数值

匹配项规范

- Occurrence.First = 0;
- Occurrence.Last = 1;
- Occurrence.All = 2;

排序顺序

- Order.Ascending = 0;
- Order.Descending = 1;

相等条件

可将列表值的相等条件指定为以下任一项

- 一个函数值，它可以是
  - 一个键选择器，用于确定列表中要应用相等条件的值，或
  - 一个比较器函数，用于指定要应用的比较类型。可以指定内置的比较器函数，请参阅“比较器函数”一节。
- 一个列表值，此值
  - 正好有两项
  - 第一个元素是上面指定的键选择器
  - 第二个元素是上面指定的比较器。

有关详细信息和示例，请参阅 List.Distinct。

比较条件

可将比较条件提供为以下值之一：

- 用于指定排序顺序的一个数字值。有关详细信息，请参阅参数值的排序顺序。
- 若要计算用于排序的键，可以使用具有 1 个参数的函数。
- 若要选择键并控制顺序，比较条件可以是包含键和顺序的列表。
- 若要完全控制比较，可以使用具有 2 个参数的函数，此函数将根据左输入和右输入之间的关系返回 -1、0 或 1。Value.Compare 是可用于委托此逻辑的方法。

有关详细信息和示例，请参阅 List.Sort。

替换操作

替换操作由列表值指定，此列表中的每一项都必须为

- 仅有两项的一个列表值
- 第一项是列表中要替换的旧值
- 第二项是新值，它应替换列表中出现的所有旧值

List.Accumulate

2019/12/4 •

语法

```
List.Accumulate(list as list, seed as any, accumulator as function) as any
```

关于

使用 `accumulator` 从列表 `list` 中的项累积汇总值。可以设置一个可选的种子参数 `seed` 。

示例 1

使用 ((state, current) => state + current) 从列表 {1, 2, 3, 4, 5} 中的项累积汇总值。

```
List.Accumulate({1, 2, 3, 4, 5}, 0, (state, current) => state + current)
```

```
15
```

List.AllTrue

2019/12/3 •

语法

```
List.AllTrue(list as list) as logical
```

关于

如果列表 `list` 中的所有表达式均为 true，则返回 true。

示例 1

确定列表 {true, true, 2 > 0} 中的所有表达式是否均为 true。

```
List.AllTrue({true, true, 2 > 0})
```

`true`

示例 2

确定列表 {true, true, 2 < 0} 中的所有表达式是否均为 true。

```
List.AllTrue({true, false, 2 < 0})
```

`false`

List.Alternate

2019/12/3 •

语法

```
List.Alternate(list as list, count as number, optional repeatInterval as nullable number, optional offset as nullable number) as list
```

关于

返回由列表中所有奇数编号的偏移元素组成的列表。根据参数，交替执行获取和跳过 `list` 列表中的值的运算。

- `count`：指定每次跳过的值的数目。
- `repeatInterval`：可选的重复间隔，用于指示在跳过的值之间添加的值的数目。
- `offset`：可选的偏移参数，用于在初始偏移处开始跳过值。

示例 1

从 {1..10} 创建跳过第一个数的列表。

```
List.Alternate({1..10}, 1)
```

2
3
4
5
6
7
8
9
10

示例 2

从 {1..10} 创建隔一个数跳一次的列表。

```
List.Alternate({1..10}, 1, 1)
```

2
4
6
8
10

示例 3

从 {1..10} 创建从 1 开始且隔一个数跳一次的列表。

```
List.Alternate({1..10}, 1, 1, 1)
```

1
3
5
7
9

示例 4

从 {1..10} 创建从 1 开始并按跳过一个值、保留两个值的规律交替的列表。

```
List.Alternate({1..10}, 1, 2, 1)
```

1
3
4
6
7
9
10

List.AnyTrue

2019/12/3 •

语法

```
List.AnyTrue(list as list) as logical
```

关于

如果列表 `list` 中的任意表达式为 true，则返回 true。

示例 1

确定列表 {true, false, 2 > 0} 中的任意表达式是否为 true。

```
List.AnyTrue({true, false, 2>0})
```

`true`

示例 2

确定列表 {2 = 0, false, 2 < 0} 中的任意表达式是否为 true。

```
List.AnyTrue({2 = 0, false, 2 < 0})
```

`false`

List.Average

2019/12/3 •

语法

```
List.Average(list as list, optional precision as nullable number) as any
```

关于

返回列表 `list` 中项的平均值。结果以与列表中的值相同的数据类型给出。仅处理 number、date、time、datetime、datetimezone 和 duration 值。如果列表为空，则返回 NULL。

示例 1

计算数的列表 `{3, 4, 6}` 的平均值。

```
List.Average({3, 4, 6})
```

```
4.333333333333333
```

示例 2

计算 date 值 2011 年 1 月 1 日、2011 年 1 月 2 日和 2011 年 1 月 3 日的平均值。

```
List.Average({#date(2011, 1, 1), #date(2011, 1, 2), #date(2011, 1, 3)})
```

```
#date(2011, 1, 2)
```

List.Buffer

2019/12/4 •

语法

```
List.Buffer(list as list) as list
```

关于

在内存中缓冲列表 `list`。此调用的结果是稳定的列表。

示例 1

创建列表 {1..10} 的稳定副本。

```
List.Buffer({1..10})
```

1
2
3
4
5
6
7
8
9
10

List.Combine

2019/12/3 •

语法

```
List.Combine(lists as list) as list
```

关于

获取一系列的列表 `lists` 并将它们合并为一个新列表。

示例 1

合并两个简单的列表 {1, 2} 和 {3, 4}。

```
List.Combine({{1, 2}, {3, 4}})
```

1
2
3
4

示例 2

合并两个列表 {1, 2} 和 {3, {4, 5}}，其中一个包含嵌套的列表。

```
List.Combine({{1, 2}, {3, {4, 5}}})
```

1
2
3
[列表]

List.Contains

2019/12/4 •

语法

```
List.Contains(list as list, value as any, optional equationCriteria as any) as logical
```

关于

指示列表 `list` 是否包含值 `value`。如果在列表中找到值，则返回 true；否则返回 false。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

查找列表 {1, 2, 3, 4, 5} 是否包含 3。

```
List.Contains({1, 2, 3, 4, 5}, 3)
```

`true`

示例 2

查找列表 {1, 2, 3, 4, 5} 是否包含 6。

```
List.Contains({1, 2, 3, 4, 5}, 6)
```

`false`

List.ContainsAll

2019/12/4 •

语法

```
List.ContainsAll(list as list, values as list, optional equationCriteria as any) as logical
```

关于

指示列表 `list` 是否包含另一个列表中的所有值 `values`。如果在列表中找到值，则返回 true；否则返回 false。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

查看列表 {1, 2, 3, 4, 5} 是否包含 3 和 4。

```
List.ContainsAll({1, 2, 3, 4, 5}, {3, 4})
```

`true`

示例 2

查看列表 {1, 2, 3, 4, 5} 是否包含 5 和 6。

```
List.ContainsAll({1, 2, 3, 4, 5}, {5, 6})
```

`false`

List.ContainsAny

2019/12/3 •

语法

```
List.ContainsAny(list as list, values as list, optional equationCriteria as any) as logical
```

关于

指示一个列表 `list` 是否包含另一个列表中的任意值 `values`。如果在列表中找到值，则返回 true；否则返回 false。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

查看列表 {1, 2, 3, 4, 5} 是否包含 3 或 9。

```
List.ContainsAny({1, 2, 3, 4, 5}, {3, 9})
```

`true`

示例 2

查看列表 {1, 2, 3, 4, 5} 是否包含 6 或 7。

```
List.ContainsAny({1, 2, 3, 4, 5}, {6, 7})
```

`false`

List.Count

2019/12/4 •

语法

```
List.Count(list as list) as number
```

关于

返回列表 `list` 中的项数。

示例 1

查找列表 {1, 2, 3} 中的值的数目。

```
List.Count({1, 2, 3})
```

`3`

List.Covariance

2020/4/30 •

语法

```
List.Covariance(numberList1 as list, numberList2 as list) as nullable number
```

关于

返回两个列表（`numberList1` 和 `numberList2`）之间的协方差。`numberList1` 和 `numberList2` 必须包含相同数量的 `number` 个值。

示例 1

计算两个列表之间的协方差。

```
List.Covariance({1, 2, 3}, {1, 2, 3})
```

```
0.66666666666666607
```

List.Dates

2019/12/4 •

语法

```
List.Dates(start as date, count as number, step as duration) as list
```

关于

从 `start` 开始，返回大小为 `count` 的 `date` 值的列表。给定增量 `step` 是与每个值相加的 `duration` 值。

示例 1

创建 5 个值的列表，此列表从新年除夕 (#date(2011, 12, 31)) 开始，以 1 天为增量 (#duration(1, 0, 0, 0))。

```
List.Dates(#date(2011, 12, 31), 5, #duration(1, 0, 0, 0))
```

12/31/2011 12:00:00 AM
1/1/2012 12:00:00 AM
1/2/2012 12:00:00 AM
1/3/2012 12:00:00 AM
1/4/2012 12:00:00 AM

List.DateTimes

2019/12/4 •

语法

```
List.DateTimes(start as datetime, count as number, step as duration) as list
```

关于

从 `start` 开始，返回大小为 `count` 的 `datetime` 值的列表。给定增量 `step` 是与每个值相加的 `duration` 值。

示例

从新年前 5 分钟 (#datetime(2011, 12, 31, 23, 55, 0)) 开始创建 10 个值的列表，以 1 分钟为增量 (#duration(0, 0, 1, 0))。

```
List.DateTimes(#datetime(2011, 12, 31, 23, 55, 0), 10, #duration(0, 0, 1, 0))
```

12/31/2011 11:55:00 PM
12/31/2011 11:56:00 PM
12/31/2011 11:57:00 PM
12/31/2011 11:58:00 PM
12/31/2011 11:59:00 PM
1/1/2012 12:00:00 AM
1/1/2012 12:01:00 AM
1/1/2012 12:02:00 AM
1/1/2012 12:03:00 AM
1/1/2012 12:04:00 AM

List.DateTimeZones

2019/12/3 •

语法

```
List.DateTimeZones(start as datetimezone, count as number, step as duration) as list
```

关于

从 `start` 开始，返回大小为 `count` 的 `datetimezone` 值的列表。给定增量 `step` 是与每个值相加的 `duration` 值。

示例 1

从新年前 5 分钟 (#datetimezone(2011, 12, 31, 23, 55, 0, -8, 0)) 开始创建 10 个值的列表，以 1 分钟为增量 (#duration(0, 0, 1, 0))。

```
List.DateTimeZones(#datetimezone(2011, 12, 31, 23, 55, 0, -8, 0), 10, #duration(0, 0, 1, 0))
```

2011/12/31 下午 11:55:00 - 08:00
2011/12/31 下午 11:56:00 - 08:00
2011/12/31 下午 11:57:00 - 08:00
2011/12/31 下午 11:58:00 - 08:00
2011/12/31 下午 11:59:00 - 08:00
2012/1/1 凌晨 12:00:00 - 08:00
2012/1/1 凌晨 12:01:00 - 08:00
2012/1/1 凌晨 12:02:00 - 08:00
2012/1/1 凌晨 12:03:00 - 08:00
2012/1/1 凌晨 12:04:00 - 08:00

List.Difference

2020/4/30 •

```
List.Difference(list1 as list, list2 as list, optional equationCriteria as any) as list
```

关于

返回列表 `list1` 中未出现在列表 `list2` 中的项。支持重复值。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

查找列表 {1, 2, 3, 4, 5} 中未出现在 {4, 5, 3} 中的项。

```
List.Difference({1, 2, 3, 4, 5}, {4, 5, 3})
```

1
2

示例 2

查找列表 {1, 2} 中未出现在 {1, 2, 3} 中的项。

```
List.Difference({1, 2}, {1, 2, 3})
```

List.Distinct

2019/12/4 •

语法

```
List.Distinct(list as list, optional equationCriteria as any) as list
```

关于

返回一个列表，此列表包含列表 `list` 中的所有值，并且表中重复项已被删除。如果列表为空，结果则为空列表。

示例 1

从列表 {1, 1, 2, 3, 3, 3} 中删除重复项。

```
List.Distinct({1, 1, 2, 3, 3, 3})
```

1
2
3

List.Durations

2019/12/3 •

语法

```
List.Durations(start as duration, count as number, step as duration) as list
```

关于

返回 `count` `duration` 值的列表，从 `start` 开始，并按给定的 `duration` `step` 递增。

示例

创建包含 5 个值的列表，从 1 小时开始，并按 1 个小时递增。

```
List.Durations(#duration(0, 1, 0, 0), 5, #duration(0, 1, 0, 0))
```

01:00:00
02:00:00
03:00:00
04:00:00
05:00:00

List.FindText

2019/12/3 •

语法

```
List.FindText(list as list, text as text) as list
```

关于

从包含值 `text` 的列表 `list` 返回值列表。

示例 1

在列表 {"a", "b", "ab"} 中查找匹配 "a" 的文本值。

```
List.FindText({"a", "b", "ab"}, "a")
```

一个
ab

List.First

2019/12/4 •

语法

```
List.First(list as list, optional defaultValue as any) as any
```

关于

返回 `list` 列表中的第一项；如果列表为空，则返回可选默认值 `defaultValue`。如果列表为空且未指定默认值，则函数返回 `null`。

示例 1

查找列表 {1, 2, 3} 中的第一个值。

```
List.First({1, 2, 3})
```

`1`

示例 2

查找列表 {} 中的第一个值。如果列表为空，则返回 -1.

```
List.First({}, -1)
```

`-1`

List.FirstN

2020/4/30 •

语法

```
List.FirstN(list as list, countOrCondition as any) as any
```

关于

- 如果指定了数字，则最多返回指定数量的项。
- 如果指定了条件，则返回最初满足条件的所有项。一旦某个项目不符合条件，则不再考虑其他项目。

示例 1

在列表 {3, 4, 5, -1, 7, 8, 2} 中查找大于 0 的初始值。

```
List.FirstN({3, 4, 5, -1, 7, 8, 2}, each _ > 0)
```

3
4
5

List.Generate

2020/4/30 •

语法

```
List.Generate(initial as function, condition as function, next as function, optional selector as nullable function) as list
```

关于

给定生成初始值 `initial` 的四个函数，针对条件 `condition` 进行测试，如果成功，则选择结果并生成下一个值 `next`，以此生成值列表。还可以指定可选参数 `selector`。

示例 1

创建从 10 开始，大于 0 且按 1 递减的值的列表。

```
List.Generate(() => 10, each _ > 0, each _ - 1)
```

10
9
8
7
6
5
4
3
2
1

示例 2

生成包含 x 和 y 的记录列表，其中 x 是一个值，y 是一个列表。x 应小于 10，表示列表 y 中的项数。生成列表后，只返回 x 值。

```
List.Generate(
    () => [x = 1, y = {}],
    each [x] < 10,
    each [x = List.Count([y]), y = [y] & {x}],
    each [x]
)
```

1
0
1
2
3
4
5
6
7
8
9

List.InsertRange

2019/12/3 •

语法

```
List.InsertRange(list as list, index as number, values as list) as list
```

关于

返回新列表，该列表是通过将 `values` 中的值插入到 `index` 中的 `list` 而生成的。列表中的第一个位置位于索引 0 处。

- `list`：要插入值的目标列表。
- `index`：要插入值的目标列表 (`list`) 的索引。列表中的第一个位置位于索引 0 处。
- `values`：要插入 `list` 的值的列表。

示例 1

在索引 2 处将列表 ({3, 4}) 插入目标列表 ({1, 2, 5})。

```
List.InsertRange({1, 2, 5}, 2, {3, 4})
```

1
2
3
4
5

示例 2

在索引 0 处将带嵌套列表的列表 ({1, {1.1, 1.2}}) 插入目标列表 ({2, 3, 4})。

```
List.InsertRange({2, 3, 4}, 0, {1, {1.1, 1.2}})
```

1
[列表]
2
3

4

List.Intersect

2019/12/3 •

语法

```
List.Intersect(lists as list, optional equationCriteria as any) as list
```

关于

返回在输入列表 `lists` 中找到的列表值的交集。可以指定一个可选参数 `equationCriteria`。

示例 1

查找列表 {1..5}, {2..6}, {3..7} 的交集。

```
List.Intersect({{1..5}, {2..6}, {3..7}})
```

3
4
5

List.IsDistinct

2019/12/4 •

语法

```
List.IsDistinct(list as list, optional equationCriteria as any) as logical
```

关于

返回一个逻辑值，指示列表 `list` 中是否有重复值；如果列表是非重复的，则为 `true`，否则为 `false`。

示例 1

查找列表 {1, 2, 3} 是否是非重复的（即没有重复值）。

```
List.IsDistinct({1, 2, 3})
```

`true`

示例 2

查找列表 {1, 2, 3, 3} 是否是非重复的（即没有重复值）。

```
List.IsDistinct({1, 2, 3, 3})
```

`false`

List.IsEmpty

2019/12/3 •

语法

```
List.IsEmpty(list as list) as logical
```

关于

如果列表 `list` 不包含任何值（长度为 0），则返回 `true`。如果列表包含值（长度 > 0），则返回 `false`。

示例 1

查看列表 {} 是否为空。

```
List.IsEmpty({})
```

`true`

示例 2

查看列表 {1, 2} 是否为空。

```
List.IsEmpty({1, 2})
```

`false`

List.Last

2019/12/3 •

语法

```
List.Last(list as list, optional defaultValue as any) as any
```

关于

返回 `list` 列表的最后一项，如果列表为空，则返回可选默认值 `defaultValue`。如果列表为空且未指定默认值，则函数返回 `null`。

示例 1

查找列表 {1, 2, 3} 中的最后一个值。

```
List.Last({1, 2, 3})
```

`3`

示例 2

查找列表 {} 中的最后一个值，如果列表为空，则返回 -1。

```
List.Last({}, -1)
```

`-1`

List.LastN

2020/4/30 •

语法

```
List.LastN(list as list, optional countOrCondition as any) as any
```

关于

返回列表 `list` 中的最后一项。如果列表为空，则会引发异常。此函数采用可选参数 `countOrCondition`，以支持收集多个项或筛选项。`countOrCondition` 可以通过以下三种方式指定：

- 如果指定了数字，则最多返回指定数量的项。
- 如果指定了条件，则从列表末尾开始，返回最初满足条件的所有项。一旦某个项目不符合条件，则不再考虑其他项目。
- 如果此参数为 NULL，则返回列表中的最后一项。

示例 1

查找列表 {3, 4, 5, -1, 7, 8, 2} 中的最后一个值。

```
List.LastN({3, 4, 5, -1, 7, 8, 2}, 1)
```

2

示例 2

查找列表 {3, 4, 5, -1, 7, 8, 2} 中大于 0 的最后一个值。

```
List.LastN({3, 4, 5, -1, 7, 8, 2}, each _ > 0)
```

7
8
2

List.MatchesAll

2020/4/30 •

语法

```
List.MatchesAll(list as list, condition as function) as logical
```

关于

如果列表 `list` 中的所有值均满足条件函数 `condition`，则返回 `true`，否则返回 `false`。

示例 1

确定是否列表 {11, 12, 13} 中所有值都大于 10。

```
List.MatchesAll({11, 12, 13}, each _  > 10)
```

`true`

示例 2

确定是否列表 {1, 2, 3} 中所有值都大于 10。

```
List.MatchesAll({1, 2, 3}, each _  > 10)
```

`false`

List.MatchesAny

2020/4/30 •

语法

```
List.MatchesAny(list as list, condition as function) as logical
```

关于

如果列表 `list` 中的任何值满足条件函数 `condition`，则返回 `true` 否则返回 `false`。

示例 1

查看列表 {9, 10, 11} 中的任意值是否大于 10。

```
List.MatchesAny({9, 10, 11}, each _ > 10)
```

`true`

示例 2

查看列表 {1, 2, 3} 中的任意值是否大于 10。

```
List.MatchesAny({1, 2, 3}, each _ > 10)
```

`false`

List.Max

2020/4/30 •

语法

```
List.Max(list as list, optional default as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as any
```

关于

返回 `list` 列表中最大的一项；如果列表为空，则返回可选默认值 `default`。可以指定一个可选的 comparisonCriteria 值 `comparisonCriteria` 以确定如何比较列表中的项。如果此参数为 NULL，则使用默认比较器。

示例 1

在列表 {1, 4, 7, 3, -2, 5} 中查找最大值。

```
List.Max({1, 4, 7, 3, -2, 5}, 1)
```

`7`

示例 2

查找列表 {} 中的最大值；如果列表为空，则返回 -1。

```
List.Max({}, -1)
```

`-1`

List.MaxN

2019/12/3 •

语法

```
List.MaxN(list as list, countOrCondition as any, optional comparisonCriteria as any, optional
includeNulls as nullable logical) as list
```

关于

返回列表 `list` 中的最大值。对行进行排序之后，可以指定可选参数以进一步筛选结果。可选参数，`countOrCondition` 指定要返回的值的数量或筛选条件。可选参数 `comparisonCriteria` 指定如何比较列表中的值。

- `list`：值列表。
- `countOrCondition`：如果指定了一个数字，则返回最多 `countOrCondition` 个项目的升序列表。如果指定了条件，则返回最初满足条件的项目列表。一旦某个项目不符合条件，则不再考虑其他项目。
- `comparisonCriteria`：*[可选]* 可以指定一个可选 `comparisonCriteria` 值以确定如何比较列表中的项。如果此参数为 NULL，则使用默认比较器。

List.Median

2019/12/4 •

语法

```
List.Median(list as list, optional comparisonCriteria as any) as any
```

关于

返回列表 `list` 的中值项。如果列表中不包含非 `null` 值，此函数则返回 `null`。如果项数存在偶数，此函数选择两个中值项中的较小者，除非列表完全由日期/时间、持续时间、数字或时间组成，在这种情况下，函数则返回两个项的平均值。

示例 1

查找列表 `{5, 3, 1, 7, 9}` 的中值。

```
powerquery-mList.Median({5, 3, 1, 7, 9})
```

`5`

List.Min

2019/12/4 •

语法

```
List.Min(list as list, optional default as any, optional comparisonCriteria as any, optional includeNulls as nullable logical) as any
```

关于

返回 `list` 列表中最小的一项；如果列表为空，则返回可选默认值 `default`。可以指定一个可选的 comparisonCriteria 值 `comparisonCriteria` 以确定如何比较列表中的项。如果此参数为 NULL，则使用默认比较器。

示例 1

查找列表 {1, 4, 7, 3, -2, 5} 中的最小值。

```
List.Min({1, 4, 7, 3, -2, 5})
```

`-2`

示例 2

查找列表 {} 中的最小值；如果列表为空，则返回 -1。

```
List.Min({}, -1)
```

`-1`

List.MinN

2019/12/4 •

语法

```
List.MinN(list as list, countOrCondition as any, optional comparisonCriteria as any, optional
includeNulls as nullable logical) as list
```

关于

返回列表 `list` 中的最小值。参数 `countOrCondition`，指定要返回的值的数量或筛选条件。可选参数 `comparisonCriteria` 指定如何比较列表中的值。

- `list`：值列表。
- `countOrCondition`：如果指定了一个数字，则返回最多 `countOrCondition` 个项目的升序列表。如果指定了条件，则返回最初满足条件的项目列表。一旦某个项目不符合条件，则不再考虑其他项目。如果此参数为 NULL，则返回列表中的单个最小值。
- `comparisonCriteria`：*[可选]* 可以指定一个可选 `comparisonCriteria` 值以确定如何比较列表中的项。如果此参数为 NULL，则使用默认比较器。

示例 1

查找列表 `{3, 4, 5, -1, 7, 8, 2}` 中的 5 个最小值。

```
List.MinN({3, 4, 5, -1, 7, 8, 2}, 5)
```

-1
2
3
4
5

List.Mode

2019/12/4 •

语法

```
List.Mode(list as list, optional equationCriteria as any) as any
```

关于

返回列表 `list` 中出现最频繁的项。如果列表为空，则会引发异常。如果多个项以相同的最大频率显示，则选择最后一个。可以指定可选 `comparisonCriteria` 值 `equationCriteria` 来控制相等测试。

示例 1

查找列表 `{"A", 1, 2, 3, 3, 4, 5}` 中出现最频繁的项。

```
List.Mode({"A", 1, 2, 3, 3, 4, 5})
```

`3`

示例 2

查找列表 `{"A", 1, 2, 3, 3, 4, 5, 5}` 中出现最频繁的项。

```
List.Mode({"A", 1, 2, 3, 3, 4, 5, 5})
```

`5`

List.Modes

2019/12/3 •

语法

```
List.Modes(list as list, optional equationCriteria as any) as list
```

关于

返回列表 `list` 中出现最频繁的项。如果列表为空，则会引发异常。如果多个项以相同的最大频率显示，则选择最后一个。可以指定可选 `comparisonCriteria` 值 `equationCriteria` 来控制相等测试。

示例 1

查找列表 `{"A", 1, 2, 3, 3, 4, 5, 5}` 中出现最频繁的项。

```
List.Modes({"A", 1, 2, 3, 3, 4, 5, 5})
```

3
5

List.NonNullCount

2019/12/4 •

语法

```
List.NonNullCount(list as list) as number
```

关于

返回列表 `list` 中的非 NULL 项数。

List.Numbers

2019/12/3 •

语法

```
List.Numbers(start as number, count as number, optional increment as nullable number) as list
```

关于

返回给定了初始值、计数和可选增量值的数值列表。默认增量值为 1。

- `start`：列表中的初始值。
- `count`：要创建的值的数目。
- `increment`：[可选] 按该值递增。如果省略，值按 1 递增。

示例 1

生成从 1 开始的 10 个连续数的列表。

```
List.Numbers(1, 10)
```

1
2
3
4
5
6
7
8
9
10

示例 2

生成从 1 开始的 10 个数的列表，每个后续数按 2 递增。

```
List.Numbers(1, 10, 2)
```

1
3
5
7
9
11
13
15
17
19

List.PositionOf

2019/12/4 •

语法

```
List.PositionOf(list as list, value as any, optional occurrence as nullable number, optional
equationCriteria as any) as any
```

关于

返回 `value` 值在列表 `list` 中显示的值的偏移量。如果值未出现，则返回 -1。可以指定一个可选的出现参数 `occurrence`。

- `occurrence`：要报告的最大出现次数。

示例 1

查找列表 {1, 2, 3} 中出现值 3 的位置。

```
List.PositionOf({1, 2, 3}, 3)
```

`2`

List.PositionOfAny

2019/12/4 •

语法

```
List.PositionOfAny(list as list, values as list, optional occurrence as nullable number, optional equationCriteria as any) as any
```

关于

返回列表 `values` 中的值第一次出现的列表 `list` 中的偏移值。如果未找到匹配项，则返回 -1。可以指定一个可选的出现参数 `occurrence`。

- `occurrence`：可返回的最大出现次数。

示例 1

查找列表 {1, 2, 3} 中第一次出现值 2 或 3 的位置。

```
List.PositionOfAny({1, 2, 3}, {2, 3})
```

`1`

List.Positions

2019/12/3 •

语法

```
List.Positions(list as list) as list
```

关于

返回 `list` 输入列表的偏移量列表。使用 List.Transform 更改列表时，可以使用位置列表来授予对位置的转换权限。

示例 1

查找列表 {1, 2, 3, 4, null, 5} 中值的偏移量。

```
List.Positions({1, 2, 3, 4, null, 5})
```

0
1
2
3
4
5

List.Product

2019/12/4 •

语法

```
List.Product(numbersList as list, optional precision as nullable number) as nullable number
```

关于

返回列表 `numbersList` 中非 NULL 数的乘积。如果列表中没有非 NULL 值，则返回 NULL。

示例 1

查找列表 `{1, 2, 3, 3, 4, 5, 5}` 中的数的乘积。

```
List.Product({1, 2, 3, 3, 4, 5, 5})
```

```
1800
```

List.Random

2019/12/4 •

语法

```
List.Random(count as number, optional seed as nullable number) as list
```

关于

给定要生成的值数量和可选种子值，返回介于 0 到 1 之间的随机数的列表。

- `count`：要生成的随机值数量。
- `seed`：*[可选]* 用于为随机数生成器设定种子的数值。如果省略，则每次调用函数时会生成唯一的随机数列表。如果使用一个数指定种子值，则每次调用函数都会生成相同的随机数列表。

示例 1

创建 3 个随机数的列表。

```
List.Random(3)
```

0.992332
0.132334
0.023592

示例 2

创建 3 个随机数的列表，并指定种子值。

```
List.Random(3, 2)
```

0.883002
0.245344
0.723212

List.Range

2019/12/4 •

语法

```
List.Range(list as list, offset as number, optional count as nullable number) as list
```

关于

返回从偏移量开始的列表 `list` 的子集。可选参数 `offset` 设置子集中的最大项数。

示例 1

查找包含数字 1-10 的列表中从偏移量 6 开始的子集。

```
List.Range({1..10}, 6)
```

7
8
9
10

示例 2

查找包含数字 1-10 的列表中从偏移量 6 开始、长度为 2 的子集。

```
List.Range({1..10}, 6, 2)
```

7
8

List.RemoveFirstN

2019/12/4 •

语法

```
List.RemoveFirstN(list as list, optional countOrCondition as any) as list
```

关于

返回一个列表，该列表删除列表 `list` 的第一个元素。如果 `list` 为空列表，则返回空列表。此函数采用可选参数 `countOrCondition`，以支持删除下列的多个值。

- 如果指定了数字，则最多会删除指定数量的项。
- 如果指定了条件，则返回的列表将以满足条件的 `list` 的第一个元素开头。一旦某个项目不符合条件，则不再考虑其他项目。
- 如果此参数为 NULL，则会观察到默认行为。

示例 1

从 {1, 2, 3, 4, 5} 创建不带前 3 个数的列表。

```
List.RemoveFirstN({1, 2, 3, 4, 5}, 3)
```

4
5

示例 2

从 {5, 4, 2, 6, 1} 创建以小于数字 3 开头的列表。

```
List.RemoveFirstN({5, 4, 2, 6, 1}, each _ > 3)
```

2
6
1

List.RemoveItems

2019/12/3 •

语法

```
List.RemoveItems(list1 as list, list2 as list) as list
```

关于

从 `list1` 中删除在 `list2` 中出现的所有给定值。如果 `list1` 中不存在 `list2` 中的值，则返回原始列表。

示例 1

从列表 {1, 2, 3, 4, 2, 5, 5} 中删除在列表 {2, 4, 6} 中出现的项。

```
List.RemoveItems({1, 2, 3, 4, 2, 5, 5}, {2, 4, 6})
```

1
3
5
5

List.RemoveLastN

2019/12/4 •

语法

```
List.RemoveLastN(list as list, optional countOrCondition as any) as list
```

关于

返回一个列表，它从列表 `list` 末尾删除最后几个 `countOrCondition` 元素。如果 `list` 中少于 `countOrCondition` 个元素，则返回空列表。

- 如果指定了数字，则最多会删除指定数量的项。
- 如果指定了条件，则返回的列表将以从 `list` 底部开始的第一个满足条件的元素结尾。一旦某个项目不符合条件，则不再考虑其他项目。
- 如果此参数为 NULL，则仅删除一项。

示例 1

从 {1, 2, 3, 4, 5} 创建不带后 3 个数的列表。

```
List.RemoveLastN({1, 2, 3, 4, 5}, 3)
```

1
2

示例 2

从 {5, 4, 2, 6, 4} 创建以小于 3 的数结尾的列表。

```
List.RemoveLastN({5, 4, 2, 6, 4}, each _ > 3)
```

5
4
2

List.RemoveMatchingItems

2019/12/3 •

语法

```
List.RemoveMatchingItems(list1 as list, list2 as list, optional equationCriteria as any) as list
```

关于

从列表 `list1` 删除 `list2` 中出现的所有给定值。如果 `list1` 中不存在 `list2` 中的值，则返回原始列表。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

从 {1, 2, 3, 4, 5, 5} 创建一个不包含 {1, 5} 的列表。

```
List.RemoveMatchingItems({1, 2, 3, 4, 5, 5}, {1, 5})
```

2
3
4

List.RemoveNulls

2019/12/3 •

语法

```
List.RemoveNulls(list as list) as list
```

关于

删除 `list` 中出现的所有“NULL”值。如果列表中没有“NULL”值，则返回原始列表。

示例 1

从列表 {1, 2, 3, null, 4, 5, null, 6} 中删除“NULL”值。

```
List.RemoveNulls({1, 2, 3, null, 4, 5, null, 6})
```

1
2
3
4
5
6

List.RemoveRange

2019/12/3 •

语法

```
List.RemoveRange(list as list, index as number, optional count as nullable number) as list
```

关于

在 `list` 中删除从指定位置 `index` 起的 `count` 个值。

示例 1

在列表 {1, 2, 3, 4, -6, -2, -1, 5} 中删除从索引 4 起的 3 个值。

```
List.RemoveRange({1, 2, 3, 4, -6, -2, -1, 5}, 4, 3)
```

1
2
3
4
5

List.Repeat

2019/12/4 •

语法

```
List.Repeat(list as list, count as number) as list
```

关于

返回为原始列表 `list` 的 `count` 次重复的列表。

示例 1

创建将 {1, 2} 重复了 3 次得到的列表。

```
List.Repeat({1, 2}, 3)
```

1
2
1
2
1
2

List.ReplaceMatchingItems

2019/12/3 •

语法

```
List.ReplaceMatchingItems(list as list, replacements as list, optional equationCriteria as any) as list
```

关于

对列表 `list` 执行给定的替换。替换操作 `replacements` 由两个值(旧值和新值)组成，以列表形式提供。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

从 {1, 2, 3, 4, 5} 创建一个列表，将值 5 替换为 -5，将值 1 替换为 -1。

```
List.ReplaceMatchingItems({1, 2, 3, 4, 5}, {{5, -5}, {1, -1}})
```

-1
2
3
4
-5

List.ReplaceRange

2019/12/4 •

语法

```
List.ReplaceRange(list as list, index as number, count as number, replaceWith as list) as list
```

关于

从指定的位置 `index` 开始，使用列表 `replaceWith` 替换 `list` 中的 `count` 个值。

示例 1

使用 {3, 4} 替换列表 {1, 2, 7, 8, 9, 5} 中的 {7, 8, 9}。

```
List.ReplaceRange({1, 2, 7, 8, 9, 5}, 2, 3, {3, 4})
```

1
2
3
4
5

List.ReplaceValue

2019/12/4 •

语法

```
List.ReplaceValue(list as list, oldValue as any, newValue as any, replacer as function) as list
```

关于

在值列表 `list` 中搜索值 `oldValue`，每次找到后使用替换值 `newValue` 将其替换。

示例 1

使用 "A" 替换列表 {"a", "B", "a", "a"} 中的所有 "a" 值。

v List.ReplaceValue({"a", "B", "a", "a"}, "a", "A", Replacer.ReplaceText)

```
<table> <tr><td>A</td></tr> <tr><td>B</td></tr> <tr><td>A</td></tr> <tr><td>A</td></tr> </table>
```

List.Reverse

2019/12/4 •

语法

```
List.Reverse(list as list) as list
```

关于

返回将列表 `list` 中的值反向排序得到的列表。

示例 1

通过将 {1..10} 反向排序来创建一个列表。

```
List.Reverse({1..10})
```

10
9
8
7
6
5
4
3
2
1

List.Select

2019/12/4 •

语法

```
List.Select(list as list, selection as function) as list
```

关于

从列表 `list` 返回匹配选择条件 `selection` 的值的列表。

示例 1

查找列表 {1, -3, 4, 9, -2} 中大于 0 的值。

```
List.Select({1, -3, 4, 9, -2}, each _ > 0)
```

1
4
9

List.Single

2019/12/3 •

语法

```
List.Single(list as list) as any
```

关于

如果列表 `list` 中只有一个项，则返回该项。如果有多个项或列表为空，则该函数将引发异常。

示例 1

在列表 {1} 中查找单个值。

```
List.Single({1})
```

```
1
```

示例 2

在列表 {1, 2, 3} 中查找单个值。

```
List.Single({1, 2, 3})
```

```
[Expression.Error] There were too many elements in the enumeration to complete the operation.
```

List.SingleOrDefault

2019/12/4 •

语法

```
List.SingleOrDefault(list as list, optional default as any) as any
```

关于

如果列表 `list` 中只有一个项，则返回该项。如果列表为空，则该函数将返回 NULL，除非制定了可选 `default`。如果列表中有多个项，则该函数将返回一个错误。

示例 1

在列表 {1} 中查找单个值。

```
List.SingleOrDefault({1})
```

`1`

示例 2

在列表 {} 中查找单个值。

```
List.SingleOrDefault({})
```

`null`

示例 3

在列表 {} 中查找单个值。如果为空，则返回 -1。

```
List.SingleOrDefault({}, -1)
```

`-1`

List.Skip

2019/12/4 •

语法

```
List.Skip(list as list, optional countOrCondition as any) as list
```

关于

返回一个列表，此列表跳过列表 `list` 的第一个元素。如果 `list` 为空列表，则返回空列表。此函数采用可选参数 `countOrCondition` 以支持跳过下列的多个值。

- 如果指定了数字，则最多跳过指定数量的项。
- 如果指定了条件，则返回的列表将以满足条件的 `list` 的第一个元素开头。一旦某个项目不符合条件，则不再考虑其他项目。
- 如果此参数为 NULL，则会观察到默认行为。

示例 1

从 {1, 2, 3, 4, 5} 创建不带前 3 个数的列表。

```
List.Skip({1, 2, 3, 4, 5}, 3)
```

4
5

示例 2

从 {5, 4, 2, 6, 1} 创建以小于数字 3 开头的列表。

```
List.Skip({5, 4, 2, 6, 1}, each _ > 3)
```

2
6
1

List.Sort

2019/12/3 •

语法

```
List.Sort(list as list, optional comparisonCriteria as any) as list
```

关于

根据指定的可选条件对数据列表 `list` 排序。可选参数 `comparisonCriteria` 可以指定为比较条件。这可以采用以下值：

- 为控制顺序，比较条件可以是顺序枚举值。(`Order.Descending` 、 `Order.Ascending`)。
- 若要计算用于排序的键，可以使用具有 1 个参数的函数。
- 若要选择键并控制顺序，比较条件可以是包含键和顺序 (`{each 1 / _, Order.Descending}`) 的列表。
- 若要完全控制比较，可以使用具有 2 个参数的函数，此函数将根据左输入和右输入之间的关系返回 -1、0 或 1。Value.Compare 是可用于委托此逻辑的方法。

示例 1

对列表 {2, 3, 1} 进行排序。

```
List.Sort({2, 3, 1})
```

1
2
3

示例 2

按降序对列表 {2, 3, 1} 进行排序。

```
List.Sort({2, 3, 1}, Order.Descending)
```

3
2
1

示例 3

使用 Value.Compare 方法，按降序对列表 {2, 3, 1} 进行排序。

```
List.Sort({2, 3, 1}, (x, y) => Value.Compare(1/x, 1/y))
```

3
2
1

List.Split

2019/12/3 •

语法

```
List.Split(list as list, pageSize as number) as list
```

关于

将 `list` 拆分为一系列列表，其中输出列表的第一个元素是包含源列表中前 `pageSize` 个元素的列表，输出列表的下一个元素是包含源列表中接下来 `pageSize` 个元素的列表，以此类推。

List.StandardDeviation

2019/12/4 •

语法

```
List.StandardDeviation(numbersList as list) as nullable number
```

关于

返回列表 `numbersList` 中值的标准偏差的基于样本估计值。如果 `numbersList` 是一列数字，则返回数字。如果列表为空或列表中的项不是 `number` 类型，将引发异常。

示例 1

计算数字 1 到 5 的标准偏差。

```
List.StandardDeviation({1..5})
```

```
1.5811388300841898
```

List.Sum

2019/12/3 •

语法

```
List.Sum(list as list, optional precision as nullable number) as any
```

关于

返回列表（`list`）中非 NULL 值的总和。如果列表中没有非 NULL 值，则返回 NULL。

示例 1

计算列表 `{1, 2, 3}` 中的数的总和。

```
List.Sum({1, 2, 3})
```

`6`

List.Times

2019/12/4 •

语法

```
List.Times(start as time, count as number, step as duration) as list
```

关于

从 `start` 开始，返回大小为 `count` 的 `time` 值的列表。给定增量 `step` 是与每个值相加的 `duration` 值。

示例 1

创建从中午 (#time(12, 0, 0)) 开始的 4 个值的列表，以 1 小时为增量 (#duration(0, 1, 0, 0))。

```
List.Times(#time(12, 0, 0), 4, #duration(0, 1, 0, 0))
```

12:00:00
13:00:00
14:00:00
15:00:00

List.Transform

2019/12/4 •

语法

```
List.Transform(list as list, transform as function) as list
```

关于

通过将转换函数 `transform` 应用到列表 `list` 来返回值的新列表。

示例 1

将 1 与列表 {1, 2} 中的每个值相加。

```
List.Transform({1, 2}, each _ + 1)
```

2
3

List.TransformMany

2019/12/4 •

语法

```
List.TransformMany(list as list, collectionTransform as function, resultTransform as function) as
list
```

关于

返回一个列表，其元素是基于输入列表投射而来的。将 `collectionTransform` 函数应用到每个元素，并调用 `resultTransform` 函数来构造结果列表。 `collectionSelector` 具有签名 (x as Any) => ...，其中 x 是列表中的元素。 `resultTransform` 投射结果的形状并具有签名 (x as Any, y as Any) => ...，其中 x 是列表中的元素，y 是通过将 `collectionTransform` 应用到该元素获得的元素。

List.Union

2020/4/30 •

语法

```
List.Union(lists as list, optional equationCriteria as any) as list
```

关于

采用列表 `lists` 的列表，联合各个列表中的项，并在输出列表中返回这些项。因此，返回的列表包含所有输入列表中的所有项。此操作维护传统的包语义，因此，重复值作为 Union 的一部分进行匹配。可以指定可选相等条件值 `equationCriteria` 来控制相等测试。

示例 1

创建列表 {1..5}、{2..6}、{3..7} 的并集。

```
List.Union({{1..5}, {2..6}, {3..7}})
```

1
2
3
4
5
6
7

List.Zip

2019/12/4 •

语法

```
List.Zip(lists as list) as list
```

关于

提取列表 `lists` 的其中一个列表，并返回一个列表显示在同一位置合并项的列表。

示例 1

压缩两个简单列表 {1, 2} 和 {3, 4}。

```
List.Zip({{1, 2}, {3, 4}})
```

[列表]
[列表]

示例 2

压缩两个具有不同长度 {1, 2} 和 {3} 的简单列表。

```
List.Zip({{1, 2}, {3}})
```

[列表]
[列表]

逻辑函数

2020/4/26 •

这些函数创建和操纵逻辑（即，true/false）值。

逻辑

"	"
Logical.From	从某个值返回逻辑值。
Logical.FromText	从文本值返回逻辑值 true 或 false。
Logical.ToText	从逻辑值返回文本值。

Logical.From

2019/12/3 •

语法

```
Logical.From(value as any) as nullable logical
```

关于

从给定的 `value` 返回 `logical` 值。如果给定的 `value` 为 `null`，则 `Logical.From` 返回 `null`。如果给定的 `value` 为 `logical`，则返回 `value`。

可以将以下类型的值转换为 `logical` 值：

- `text`：文本值中的 `logical` 值，`"true"` 或 `"false"`。有关详细信息，请参阅 `Logical.FromText`。
- `number`：如果 `value` 等于 `0`，则为 `false`，否则为 `true`。

如果 `value` 为任何其他类型，则返回错误。

示例 1

将 `2` 转换为 `logical` 值。

```
Logical.From(2)
```

`true`

Logical.FromText

2019/12/4 •

语法

```
Logical.FromText(text as nullable text) as nullable logical
```

关于

从文本值 `text`（“true”或“false”）创建逻辑值。如果 `text` 包含不同的字符串，则会引发异常。`text` 文本值不区分大小写。

示例 1

从文本字符串“true”创建逻辑值。

```
Logical.FromText("true")
```

```
true
```

示例 2

从文本字符串“a”创建逻辑值。

```
Logical.FromText("a")
```

```
[Expression.Error] Could not convert to a logical.
```

Logical.ToText

2019/12/4 •

语法

```
Logical.ToText(logicalValue as nullable logical) as nullable text
```

关于

从逻辑值 `logicalValue`（`true` 或 `false`）创建文本值。如果 `logicalValue` 不是逻辑值，则会引发异常。

示例 1

从逻辑 `true` 创建一个文本值。

```
Logical.ToText(true)
```

`"true"`

数字函数

2020/4/26 •

这些函数创建并操纵数字值。

数字

常数

Number.E	返回 2.7182818284590451，e 的值最多取16 位小数。
Number.Epsilon	返回可能的最小值。
Number.NaN	表示 0/0。
Number.NegativeInfinity	表示 -1/0。
Number.PI	返回 3.1415926535897931，Pi 的值最多取 16 位小数。
Number.PositiveInfinity	表示 1/0。

信息

Number.IsEven	如果值为偶数，则返回 true。
Number.IsNaN	如果值为 Number.NaN，则返回 true。
Number.IsOdd	如果值为奇数，则返回 true。

转换和格式设置

Byte.From	从给定的值返回8 位整数数值。
Currency.From	从给定的值返回一个货币值。
Decimal.From	从给定的值返回一个十进制数值。
Double.From	从给定的值返回一个双精度数值。
Int8.From	从给定的值返回给带符号的一个 8 位整数值。
Int16.From	从给定的值返回一个 16 位整数值。
Int32.From	从给定的值返回一个 32 位整数值。

Int64.From	从给定的值返回一个 64 位整数值。
Number.From	从某个值返回一个数字。
Number.FromText	从文本值返回一个数值。
Number.ToText	从数值返回一个文本值。
Percentage.From	从给定的值返回一个百分比值。
Single.From	从给定的值返回单个数值。

舍入

Number.Round	如果值为整数，则返回可为空的数字 (n)。
Number.RoundAwayFromZero	当值 >= 0 时，返回 Number.RoundUp(value)；当值 < 0 时，则返回 Number.RoundDown(value)。
Number.RoundDown	返回小于或等于数值的最大整数。
Number.RoundTowardZero	当 x >= 0 时，返回 Number.RoundDown(x)；当 x < 0 时，返回 Number.RoundUp(x)。
Number.RoundUp	返回大于或等于数值的最大整数。

运营

Number.Abs	返回某一数字的绝对值。
Number.Combinations	返回可选组合大小的给定项数的组合数。
Number.Exp	返回表示 e 的乘幂的数字 。
Number.Factorial	返回数字的阶乘。
Number.IntegerDivide	使两个数相除，返回生成的数字的所有部分。
Number.Ln	返回某一数字的自然对数。
Number.Log	根据底数返回数字的对数。
Number.Log10	返回某一数字以 10 为底的对数。
Number.Mod	使两个数相除，返回生成的数字的余数。
Number.Permutations	返回可选排列大小的给定项数的总排列数。

Number.Power	返回由幂生成的数字。
Number.Sign	对于正数，返回 1；对于负数，返回 -1；如果为零，则返回 0。
Number.Sqrt	返回某一数字的平方根。

随机

Number.Random	返回介于 0 到 1 之间的随机小数。
Number.RandomBetween	返回两个给定数值之间的一个随机数。

三角函数

Number.Acos	返回某一数字的反余弦值。
Number.Asin	返回某一数字的反正弦值。
Number.Atan	返回某一数字的反正切值。
Number.Atan2	返回两个数相除的反正切值。
Number.Cos	返回某一数字的余弦值。
Number.Cosh	返回某一数字的双曲余弦值。
Number.Sin	返回某一数字的正弦值。
Number.Sinh	返回某一数字的双曲正弦值。
Number.Tan	返回某一数字的正切值。
Number.Tanh	返回某一数字的双曲正切值。

字节数

Number.BitwiseAnd	返回对提供的操作数进行按位 AND 运算的结果。
Number.BitwiseNot	返回对提供的操作数进行按位 NOT 运算的结果。
Number.BitwiseOr	返回对提供的操作数进行按位 OR 运算的结果。
Number.BitwiseShiftLeft	返回对操作数进行按位左移操作的结果。
Number.BitwiseShiftRight	返回对操作数进行按位右移操作的结果。

[illegible]	[illegible]
Number.BitwiseXor	返回对提供的操作数进行按位 XOR 操作的结果。

[illegible]	[illegible]
RoundingMode.AwayFromZero	RoundingMode.AwayFromZero
RoundingMode.Down	RoundingMode.Down
RoundingMode.ToEven	RoundingMode.ToEven
RoundingMode.TowardZero	RoundingMode.TowardZero
RoundingMode.Up	RoundingMode.Up

Byte.From

2019/12/4 •

语法

```
Byte.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

关于

从给定的 `value` 中返回 8 位整数 `number` 值。如果给定的 `value` 为 `null`，则 `Byte.From` 返回 `null`。如果给定的 `value` 为不含小数部分的 8 位整数范围内的 `number`，则返回 `value`。如果它含小数部分，则使用指定的舍入模式对数字进行舍入。默认的舍入模式为 `RoundingMode.ToEven`。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以便将其转换为 `number` 值，然后将 `number` 值转换为 8 位整数 `number` 值的上一条语句即可适用。请参阅 `Number.Round` 以获取可用的舍入模式。

示例 1

获取 `"4"` 的 8 位整数 `number` 值。

```
Byte.From("4")
```

`4`

示例 2

使用 `RoundingMode.AwayFromZero` 获取 `"4.5"` 的 8 位整数 `number` 值。

```
Byte.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

=

Currency.From

2019/12/3 •

语法

```
Currency.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

关于

从给定的 `value` 返回 `currency` 值。如果给定的 `value` 为 `null`，则 `Currency.From` 返回 `null`。如果给定的 `value` 是货币范围内的 `number`，则 `value` 的小数部分会舍入为 4 位小数并返回。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以将其转换为 `number` 值，然后将 `number` 值转换为 `currency` 值的上一条语句即可适用。货币的有效范围为 `-922,337,203,685,477.5808` 至 `922,337,203,685,477.5807`。有关可用的舍入模式，请参阅 `Number.Round`，默认值为 `RoundingMode.ToEven`。

示例 1

获取 `"1.23455"` 的 `currency` 值。

```
Currency.From("1.23455")
```

`1.2346`

示例 2

使用 `RoundingMode.Down` 获取 `"1.23455"` 的 `currency` 值。

```
Currency.From("1.23455", "en-Us", RoundingMode.Down)
```

`1.2345`

Decimal.From

2019/12/9 •

语法

```
Decimal.From(value as any, optional culture as nullable text) as nullable number
```

关于

从给定的 `value` 返回十进制 `number` 值。如果给定的 `value` 为 `null`，则 `Decimal.From` 返回 `null`。如果给定的 `value` 在十进制范围内为 `number`，则返回 `value`，否则返回错误。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以便将其转换为 `number` 值，然后将 `number` 值转换为十进制 `number` 值的上一条语句即可适用。

示例 1

获取 `"4.5"` 的十进制 `number` 值。

```
Decimal.From("4.5")
```

`4.5`

Double.From

2019/12/3 •

语法

```
Double.From(value as any, optional culture as nullable text) as nullable number
```

关于

从给定的 `value` 返回双精度 `number` 值。如果给定的 `value` 为 `null`，则 `Double.From` 返回 `null`。如果给定的 `value` 在双精度范围内为 `number`，则返回 `value`，否则返回错误。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以便将其转换为 `number` 值，然后将 `number` 值转换为双精度 `number` 值的上一条语句即可适用。

示例 1

获取 `"4"` 的双精度 `number` 值。

```
Double.From("4.5")
```

`4.5`

Int8.From

2019/12/4 •

语法

```
Int8.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

关于

从给定的 `value` 返回带符号的 8 位整数 `number` 值。如果给定的 `value` 为 `null`，则 `Int8.From` 返回 `null`。如果给定的 `value` 为不含小数部分的带符号的 8 位整数范围内的 `number`，则返回 `value`。如果它含小数部分，则使用指定的舍入模式对数字进行舍入。默认的舍入模式为 `RoundingMode.ToEven`。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以将其转换为 `number` 值，然后将 `number` 值转换为带符号的 8 位整数 `number` 值的上一条语句即可适用。有关可用的舍入模式，请参阅 `Number.Round`。

示例 1

获取 `"4"` 的带符号的 8 位整数 `number` 值。

```
Int8.From("4")
```

`4`

示例 2

使用 `RoundingMode.AwayFromZero` 获取 `"4.5"` 的带符号的 8 位整数 `number` 值。

```
Int8.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Int16.From

2019/12/3 •

语法

```
Int16.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

关于

从给定的 `value` 返回 16 位整数 `number` 值。如果给定的 `value` 为 `null`，则 `Int16.From` 返回 `null`。如果给定的 `value` 为 16 位整数范围内的 `number` 且不含小数部分的，则返回 `value`。如果它含小数部分，则使用指定的舍入模式对数字进行舍入。默认的舍入模式为 `RoundingMode.ToEven`。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以便将其转换为 `number` 值，然后可将 `number` 值转换为 16 位整数 `number` 值的上一条语句即可适用。有关可用的舍入模式，请参阅 `Number.Round`。

示例 1

获取 `"4"` 的 16 位整数 `number` 值。

```
Int64.From("4")
```

`4`

示例 2

使用 `RoundingMode.AwayFromZero` 获取 `"4.5"` 的 16 位整数 `number` 值。

```
Int16.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Int32.From

2019/12/4 •

语法

```
Int32.From(value as any, optional culture as nullable text, optional roundingMode as nullable number) as nullable number
```

关于

从给定的 `value` 返回 32 位整数 `number` 值。如果给定的 `value` 为 `null`，则 `Int32.From` 返回 `null`。如果给定的 `value` 为不含小数部分的 32 位整数范围内的 `number`，则返回 `value`。如果它含小数部分，则使用指定的舍入模式对数字进行舍入。默认的舍入模式为 `RoundingMode.ToEven`。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以便将其转换为 `number` 值，然后将 `number` 值转换为 32 位整数 `number` 值的上一条语句即可适用。有关可用的舍入模式，请参阅 `Number.Round`。

示例 1

获取 `"4"` 的 32 位整数 `number` 值。

```
Int32.From("4")
```

`4`

示例 2

使用 `RoundingMode.AwayFromZero` 获取 `"4.5"` 的 32 位整数 `number` 值。

```
Int32.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Int64.From

2019/12/4 •

语法

```
Int64.From(value as any, optional culture as nullable text, optional roundingMode as nullable
number) as nullable number
```

关于

从给定的 `value` 返回 64 位整数 `number` 值。如果给定的 `value` 为 `null`，则 `Int64.From` 返回 `null`。如果给定的 `value` 为 64 位整数范围内的 `number` 且不含小数部分，则返回 `value`。如果它含小数部分，则使用指定的舍入模式对数字进行舍入。默认的舍入模式为 `RoundingMode.ToEven`。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以将其转换为 `number` 值，然后将 `number` 值转换为 64 位整数 `number` 值的上一条语句即可适用。有关可用的舍入模式，请参阅 `Number.Round`。

示例 1

获取 `"4"` 的 64 位整数 `number` 值。

```
Int64.From("4")
```

`4`

示例 2

使用 `RoundingMode.AwayFromZero` 获取 `"4.5"` 的 64 位整数 `number` 值。

```
Int64.From("4.5", null, RoundingMode.AwayFromZero)
```

`5`

Number.Abs

2019/12/3 •

语法

```
Number.Abs(number as nullable number) as nullable number
```

关于

返回 `number` 的绝对值。如果 `number` 为 NULL，则 `Number.Abs` 返回 NULL。

- `number`：要计算其绝对值的 `number`。

示例 1

-3 的绝对值。

```
Number.Abs(-3)
```

`3`

Number.Acos

2019/12/3 •

语法

```
Number.Acos(number as nullable number) as nullable number
```

关于

返回 `number` 的反余弦。

Number.Asin

2019/12/4 •

语法

```
Number.Asin(number as nullable number) as nullable number
```

关于

返回 `number` 的反正弦。

Number.Atan

2019/12/3 •

语法

```
Number.Atan(number as nullable number) as nullable number
```

关于

返回 `number` 的反正切。

Number.Atan2

2019/12/4 •

语法

```
Number.Atan2(y as nullable number, x as nullable number) as nullable number
```

关于

返回两个数（`y` 和 `x`）相除的反正切。该除将构造为 `y` / `x`。

Number.BitwiseAnd

2019/12/3 •

语法

```
Number.BitwiseAnd(number1 as nullable number, number2 as nullable number) as nullable number
```

关于

返回对 `number1` 和 `number2` 执行按位"And"运算所得的结果。

Number.BitwiseNot

2019/12/4 •

语法

```
Number.BitwiseNot(number as any) as any
```

关于

返回对 `number` 执行按位“Not”运算所得的结果。

Number.BitwiseOr

2019/12/4 •

语法

```
Number.BitwiseOr(number1 as nullable number, number2 as nullable number) as nullable number
```

关于

返回对 `number1` 和 `number2` 执行按位"Or"所得的结果。

Number.BitwiseShiftLeft

2019/12/4 •

语法

```
Number.BitwiseShiftLeft(number1 as nullable number, number2 as nullable number) as nullable number
```

关于

返回对 `number1` 执行按位左移指定的位数 `number2` 所得的结果。

Number.BitwiseShiftRight

2019/12/3 •

语法

```
Number.BitwiseShiftRight(number1 as nullable number, number2 as nullable number) as nullable number
```

关于

返回对 `number1` 执行按位右移指定的位数 `number2` 所得的结果。

Number.BitwiseXor

2019/12/4 •

语法

```
Number.BitwiseXor(number1 as nullable number, number2 as nullable number) as nullable number
```

关于

返回对 `number1` 和 `number2` 执行按位"XOR"(异或)所得的结果。

Number.Combinations

2019/12/4 •

语法

```
Number.Combinations(setSize as nullable number, combinationSize as nullable number) as nullable number
```

关于

返回项列表 `setSize` 中具有指定组合大小 `combinationSize` 的唯一组合数。

- `setSize` : 列表中的项数。
- `combinationSize` : 每个组合中的项数。

示例 1

当每个组合为 3 个一组时从总共 5 项中计算组合数。

```
Number.Combinations(5, 3)
```

`10`

Number.Cos

2019/12/4 •

语法

```
Number.Cos(number as nullable number) as nullable number
```

关于

返回 `number` 的余弦值。

示例 1

计算角 0 的余弦值。

```
Number.Cos(0)
```

```
1
```

Number.Cosh

2019/12/4 •

语法

```
Number.Cosh(number as nullable number) as nullable number
```

关于

返回 `number` 的双曲余弦。

Number.E

2019/12/3 •

关于

表示 2.7182818284590451 的常数，e 的值最多取 16 位小数。

Number.Epsilon

2019/12/4 •

关于

表示浮点数可容纳的最小正数的一个常量值。

Number.Exp

2019/12/4 •

语法

```
Number.Exp(number as nullable number) as nullable number
```

关于

返回计算 e 的 `number` 次幂(指数函数)所得的结果。

- `number`:要计算其指数函数的 `number`。如果 `number` 为 NULL，则 `Number.Exp` 返回 NULL。

示例 1

计算 e 的 3 次幂。

```
Number.Exp(3)
```

```
20.085536923187668
```

Number.Factorial

2019/12/4 •

语法

```
Number.Factorial(number as nullable number) as nullable number
```

关于

返回数 `number` 的阶乘。

示例 1

计算 10 的阶乘。

```
Number.Factorial(10)
```

```
3628800
```

Number.From

2019/12/4 •

语法

```
Number.From(value as any, optional culture as nullable text) as nullable number
```

关于

从给定的 `value` 返回 `number` 值。如果给定的 `value` 为 `null`，则 `Number.From` 返回 `null`。如果给定的 `value` 为 `number`，则返回 `value`。可以将以下类型的值转换为 `number` 值：

- `text`：文本表示形式的 `number` 值。处理常见的文本格式（"15"、"3,423.10"、"5.0E-10"）。有关详细信息，请参阅 `Number.FromText`。
- `logical`：1 表示 `true`，0 表示 `false`。
- `datetime`：一个双精度浮点数，包含等效的 OLE 自动化日期。
- `datetimezone`：一个双精度浮点数，包含与 `value` 的本地日期和时间等效的 OLE 自动化日期。
- `date`：一个双精度浮点数，包含等效的 OLE 自动化日期。
- `time`：以天的小数部分表示。
- `duration`：以整天数和天的小数部分表示。

如果 `value` 为任何其他类型，则返回错误。

示例 1

获取 `"4"` 的 `number` 值。

```
powerquery-mNumber.From("4")
```

`4`

示例 2

获取 `#datetime(2020, 3, 20, 6, 0, 0)` 的 `number` 值。

```
Number.From(#datetime(2020, 3, 20, 6, 0, 0))
```

`43910.25`

示例 3

获取 `"12.3%"` 的 `number` 值。

```
Number.From("12.3%")
```

`0.123`

Number.FromText

2019/12/4 •

语法

```
Number.FromText(text as nullable text, optional culture as nullable text) as nullable number
```

关于

从给定的文本值 `text` 返回 `number` 值。

- `text` : 数值的文本表示形式。表示形式必须采用通用数字格式 -“15”、“3,423.10”、“5.0E-10”。

示例 1

获取 `"4"` 的数值。

```
Number.FromText("4")
```

`4`

示例 2

获取 `"5.0e-10"` 的数值。

```
Number.FromText("5.0e-10")
```

`5E-10`

Number.IntegerDivide

2019/12/4 •

语法

```
Number.IntegerDivide(number1 as nullable number, number2 as nullable number, optional precision as nullable number) as nullable number
```

关于

返回将数字 `number1` 除以另一个数字 `number2` 的结果的整数部分。如果 `number1` 或 `number2` 为 NULL，则 `Number.IntegerDivide` 返回 NULL。

- `number1`：被除数。
- `number2`：除数。

示例 1

6 除以 4。

```
Number.IntegerDivide(6, 4)
```

```
1
```

示例 2

8.3 除以 3。

```
Number.IntegerDivide(8.3, 3)
```

```
2
```

Number.IsEven

2019/12/3 •

语法

```
Number.IsEven(number as number) as logical
```

关于

通过返回 `true`（如果为偶数）或 `false`（不是偶数），来指示值 `number` 是否为偶数。

示例 1

检查 625 是否为偶数。

```
Number.IsEven(625)
```

`false`

示例 2

检查 82 是否为偶数。

```
Number.IsEven(82)
```

`true`

Number.IsNaN

2019/12/4 •

语法

```
Number.IsNaN(number as number) as logical
```

关于

指示值是否为 NAN（不是数字）。如果 `number` 等效于 `Number.IsNaN`，则返回 `true` 否则返回 `false`。

示例 1

检查 0 除以 0 是否为 NAN。

```
Number.IsNaN(0/0)
```

`true`

示例 2

检查 1 除以 0 是否为 NAN。

```
Number.IsNaN(1/0)
```

`false`

Number.IsOdd

2019/12/3 •

语法

```
Number.IsOdd(number as number) as logical
```

关于

指示值是否为奇数。如果 `number` 为奇数，则返回 `true`；否则返回 `false`。

示例 1

检查 625 是否为奇数。

```
Number.IsOdd(625)
```

`true`

示例 2

检查 82 是否为奇数。

```
Number.IsOdd(82)
```

`false`

Number.Ln

2019/12/4 •

语法

```
Number.Ln(number as nullable number) as nullable number
```

关于

返回某一数字 `number` 的自然对数 。如果 `number` 为 NULL，则 `Number.Ln` 返回 NULL。

####示例 1 获取 15 的自然对数。

```
Number.Ln(15)
```

`2.70805020110221`

Number.Log

2019/12/3 •

语法

```
Number.Log(number as nullable number, optional base as nullable number) as nullable number
```

关于

返回数值 `number` 以指定的 `base` 为底的对数。如果未指定 `base`，则默认值为 Number.E。如果 `number` 为 NULL，则 `Number.Log` 返回 NULL。

示例 1

获取 2 以 10 为底的对数。

```
Number.Log(2, 10)
```

```
0.3010299956639812
```

示例 2

获取 2 以 e 为底的对数。

```
Number.Log(2)
```

```
0.69314718055994529
```

Number.Log10

2019/12/3 •

语法

```
Number.Log10(number as nullable number) as nullable number
```

关于

返回数值 `number` 的以 10 为底的对数。如果 `number` 为 NULL，则 `Number.Log10` 返回 NULL。

示例 1

获取 2 以 10 为底的对数。

```
Number.Log10(2)
```

```
0.3010299956639812
```

Number.Mod

2019/12/4 •

语法

```
Number.Mod(number as nullable number, divisor as nullable number, optional precision as nullable number) as nullable number
```

关于

返回用 `divisor` 整除 `number` 所得的余数。如果 `number` 或 `divisor` 为 NULL，则 `Number.Mod` 返回 NULL。

- `number`：被除数。
- `divisor`：除数。

示例 1

计算 5 除以 3 所得的余数。

```
Number.Mod(5, 3)
```

`2`

Number.NaN

2019/12/3 •

关于

表示 0 除以 0 的常量值。

Number.NegativeInfinity

2019/12/3 •

关于

表示 -1 除以 0 的常数值。

Number.Permutations

2019/12/4 •

语法

```
Number.Permutations(setSize as nullable number, permutationSize as nullable number) as nullable number
```

关于

使用指定的排列大小 `permutationSize` 返回可从项数 `setSize` 生成的排列数。

示例 1

计算 3 个一组、从总共 5 个项得到的排列数。

```
Number.Permutations(5, 3)
```

`60`

Number.PI

2019/12/4 •

关于

表示 3.1415926535897932 的一个常量，pi 的值最多取 16 位小数。

Number.PositiveInfinity

2019/12/4 •

关于

表示 1 除以 0 的常量值。

Number.Power

2019/12/4 •

语法

```
Number.Power(number as nullable number, power as nullable number) as nullable number
```

关于

返回将 `number` 提升为 `power` 的幂的结果。如果 `number` 或 `power` 为 null，则 `Number.Power` 返回 null。

- `number`：基数。
- `power`：指数。

示例 1

计算 5 的 3 次幂（5 的立方）的值。

```
Number.Power(5, 3)
```

```
125
```

Number.Random

2019/12/3 •

语法

```
Number.Random() as number
```

关于

返回介于 0 到 1 之间的随机数。

示例 1

获取随机数。

```
Number.Random()
```

```
0.919303
```

Number.RandomBetween

2019/12/3 •

语法

```
Number.RandomBetween(bottom as number, top as number) as number
```

关于

返回介于 `bottom` 和 `top` 之间的随机数。

示例 1

获取 1 和 5 之间的一个随机数。

```
Number.RandomBetween(1, 5)
```

```
2.546797
```

Number.Round

2019/12/4 •

语法

```
Number.Round(number as nullable number, optional digits as nullable number, optional roundingMode
as nullable number) as nullable number
```

关于

返回将 `number` 舍入为最接近的数字的结果。如果 `number` 为 NULL，则 `Number.Round` 返回 NULL。`number` 四舍五入为最接近的整数，除非指定了可选参数 `digits`。如果指定 `digits`，则将 `number` 舍入到小数点后 `digits` 位。当要舍入的可能数字之间存在等同值时（有关可能的值，请参阅 `RoundingMode.Type`），可使用可选参数 `roundingMode` 指定舍入方向。

示例 1

将 1.234 舍入到最接近的整数。

```
Number.Round(1.234)
```

`1`

示例 2

将 1.56 舍入到最接近的整数。

```
Number.Round(1.56)
```

`2`

示例 3

将 1.2345 舍入到包含两位小数。

```
Number.Round(1.2345, 2)
```

`1.23`

示例 4

将 1.2345 舍入到包含三位小数（向上舍入）。

```
Number.Round(1.2345, 3, RoundingMode.Up)
```

`1.235`

示例 5

将 1.2345 舍入到包含三位小数（向下舍入）。

```
Number.Round(1.2345, 3, RoundingMode.Down)
```

```
1.234
```

Number.RoundAwayFromZero

2019/12/4 •

语法

```
Number.RoundAwayFromZero(number as nullable number, optional digits as nullable number) as
nullable number
```

关于

返回 `number` 基于数的正负的舍入结果。此函数将向上舍入正数，向下舍入负数。如果指定 `digits`，则将 `number` 舍入到小数点后 `digits` 位。

示例 1

向远离零的方向舍入数 -1.2。

```
Number.RoundAwayFromZero(-1.2)
```

`-2`

示例 2

向远离零的方向舍入数 1.2。

```
Number.RoundAwayFromZero(1.2)
```

`2`

示例 3

将数 -1.234 向远离零的方向舍入到小数点后两位。

```
Number.RoundAwayFromZero(-1.234, 2)
```

`-1.24`

Number.RoundDown

2019/12/3 •

语法

```
Number.RoundDown(number as nullable number, optional digits as nullable number) as nullable number
```

关于

返回将 `number` 向下舍入到上一个最大整数的结果。如果 `number` 为 NULL，则 `Number.RoundDown` 返回 NULL。如果指定 `digits`，则将 `number` 舍入到小数点后 `digits` 位。

示例 1

将 1.234 向下舍入为整数。

```
Number.RoundDown(1.234)
```

```
1
```

示例 2

将 1.999 向下舍入为整数。

```
Number.RoundDown(1.999)
```

```
1
```

示例 3

将 1.999 向下舍入为包含两位小数。

```
Number.RoundDown(1.999, 2)
```

```
1.99
```

Number.RoundTowardZero

2019/12/4 •

语法

```
Number.RoundTowardZero(number as nullable number, optional digits as nullable number) as nullable number
```

关于

返回 `number` 基于数的正负的舍入结果。此函数将向下舍入正数，并向上舍入负数。如果指定 `digits`，则将 `number` 舍入到小数点后 `digits` 位。

Number.RoundUp

2019/12/4 •

语法

```
Number.RoundUp(number as nullable number, optional digits as nullable number) as nullable number
```

关于

返回将 `number` 向下舍入到上一个最大整数的结果。如果 `number` 为 NULL，则 `Number.RoundDown` 返回 NULL。如果指定 `digits`，则将 `number` 舍入到小数点后 `digits` 位。

示例 1

将 1.234 向上舍入为整数。

```
Number.RoundUp(1.234)
```

```
2
```

示例 2

将 1.999 向上舍入为整数。

```
Number.RoundUp(1.999)
```

```
2
```

示例 3

将 1.234 向上舍入为包含两位小数。

```
Number.RoundUp(1.234, 2)
```

```
1.24
```

Number.Sign

2019/12/4 •

语法

```
Number.Sign(number as nullable number) as nullable number
```

关于

如果 `number` 为正数，返回 1，如果为负数，返回 -1，如果为零，返回 0。如果 `number` 为 NULL，则 `Number.Sign` 返回 NULL。

示例 1

确定 182 的符号。

```
Number.Sign(182)
```

`1`

示例 2

确定 -182 的符号。

```
Number.Sign(-182)
```

`-1`

示例 3

确定 0 的符号。

```
Number.Sign(0)
```

`0`

Number.Sin

2019/12/4 •

语法

```
Number.Sin(number as nullable number) as nullable number
```

关于

返回 `number` 的正弦值。

示例 1

计算角 0 的正弦值。

```
Number.Sin(0)
```

`0`

Number.Sinh

2019/12/4 •

语法

```
Number.Sinh(number as nullable number) as nullable number
```

关于

返回 `number` 的双曲正弦。

Number.Sqrt

2019/12/4 •

语法

```
Number.Sqrt(number as nullable number) as nullable number
```

关于

返回 `number` 的平方根。如果 `number` 为 NULL，则 `Number.Sqrt` 返回 NULL。如果它是负数，则返回 `Number.NaN`（非数字）。

示例 1

计算 625 的平方根。

```
Number.Sqrt(625)
```

```
25
```

示例 2

计算 85 的平方根。

```
Number.Sqrt(85)
```

```
9.2195444572928871
```

Number.Tan

2020/4/30 •

语法

```
Number.Tan(number as nullable number) as nullable number
```

关于

返回 `number` 的正切。

示例 1

计算角 1 的正切。

```
Number.Tan(1)
```

```
1.5574077246549023
```

Number.Tanh

2019/12/4 •

语法

```
Number.Tanh(number as nullable number) as nullable number
```

关于

返回 `number` 的双曲正切值。

Number.ToText

2019/12/3 •

语法

```
Number.ToText(number as nullable number, optional format as nullable text, optional culture as
nullable text) as nullable text
```

关于

根据 `format` 指定的格式，将数值 `number` 格式化为文本值。格式是单个字符代码，后面可能带一个数字精度说明符。以下字符代码可用于 `format`。

- "D"或"d"：(十进制)将结果格式化为整数。精度说明符控制输出中的位数。
- "E"或"e"：(指数 [科学])指数表示法。精度说明符控制最大小数位数(默认值为 6)。
- "F"或"f"：(固定点)整数和小数位。
- "G"或"g"：(常规)固定点或科学记数法的最简洁形式。
- "N"或"n"：(数字)带组分隔符和小数分隔符的整数和小数位。
- "P"或"p"：(百分比)乘以 100 并显示百分号的数字。
- "R"或"r"：(往返)可往返转换同一数字的文本值。将忽略精度说明符。
- "X"或"x"：(十六进制)十六进制文本值。

示例 1

将数字格式化为不指定格式的文本。

```
Number.ToText(4)
```

`"4"`

示例 2

将数字格式化为指数格式的文本。

```
Number.ToText(4, "e")
```

`"4.000000e+000"`

示例 3

将数字格式化为带有限精确度的十进制格式的文本。

```
Number.ToText(-0.1234, "P1")
```

`"-12.3 %"`

Percentage.From

2019/12/3 •

语法

```
Percentage.From(value as any, optional culture as nullable text) as nullable number
```

关于

从给定的 `value` 返回 `percentage` 值。如果给定的 `value` 为 `null`，则 `Percentage.From` 返回 `null`。如果给定的 `value` 是带有尾随百分比符号的 `text`，则将返回转换的小数。否则，请参阅 `Number.From` 将其转换为 `number` 值。

示例 1

获取 `"12.3%"` 的 `percentage` 值。

```
Percentage.From("12.3%")
```

`0.123`

RoundingMode.AwayFromZero

2019/12/3 •

关于

RoundingMode.AwayFromZero

RoundingMode.Down

2019/12/3 •

关于

RoundingMode.Down

RoundingMode.ToEven

2019/12/4 •

关于

RoundingMode.ToEven

RoundingMode.TowardZero

2019/12/3 •

关于

RoundingMode.TowardZero

RoundingMode.Up

2019/12/4 •

关于

RoundingMode.Up

Single.From

2019/12/3 •

语法

```
Single.From(value as any, optional culture as nullable text) as nullable number
```

关于

从给定的 `value` 返回单精度 `number` 值。如果给定的 `value` 为 `null`，则 `Single.From` 返回 `null`。如果给定的 `value` 在单精度范围内为 `number`，则返回 `value`，否则返回错误。如果给定的 `value` 为其他任何类型，请参阅 `Number.FromText` 以便将其转换为 `number` 值，然后将 `number` 值转换为单精度 `number` 值的上一条语句即可适用。

示例 1

获取 `"1.5"` 的单精度 `number` 值。

```
Single.From("1.5")
```

`1.5`

记录函数

2020/4/26 •

这些函数创建并操纵记录值。

记录

信息

"	"
Record.FieldCount	返回记录中的字段数。
Record.HasFields	如果记录中存在一个或多个字段名称，则返回 true。

转换

"	"
Record.AddField	添加字段名称和值中的字段。
Record.Combine	组合列表中的记录。
Record.RemoveFields	返回一条新记录，此记录对给定字段进行彼此相互重新排序。任何未指定的字段仍保留在其原始位置。
Record.RenameFields	返回重命名指定字段的新记录。结果字段将保留其原始顺序。此函数支持交换和链接字段名称。但是，所有目标名称以及剩余字段名称必须组成唯一的集，否则将发生错误。
Record.ReorderFields	返回一条新记录，此记录对字段进行彼此相互重新排序。任何未指定的字段仍保留在其原始位置。需要两个或两个以上字段。
Record.TransformFields	通过应用 transformOperations 来转换字段。有关 transformOperations 支持的值的详细信息，请参阅“参数值”。

选择

"	"
Record.Field	返回给定字段的值。此函数可用于为给定记录动态创建字段查找语法。如果用于此目的，它就是动态版本的 record[field] 语法。
Record.FieldNames	按记录的字段顺序返回字段名称列表。
Record.FieldOrDefault	返回记录中的字段的值；如果此字段不存在，则返回默认值。
Record.FieldValues	按记录的字段顺序返回字段值列表。

Record.SelectFields	返回一个新记录，其中包含从输入记录中选择的字段。字段的原始顺序会被保留。

序列化

Record.FromList	根据给定的一个字段值列表和一组字段返回一个记录。
Record.FromTable	从包含字段名称和值的记录的表中返回记录。
Record.ToList	返回包含输入记录的字段值的值列表。
Record.ToTable	返回包含输入记录中的字段名称和值的记录表。

参数值

以下类型定义用于描述以上记录函数引用的参数值。

MissingField 选项	MissingField.Error = 0； MissingField.Ignore = 1； MissingField.UseNull = 2；
转换操作	可通过以下任一值指定转换操作： 两个项的一个列表值：第一项为字段名称，第二项为应用于该字段以生成新值的转换函数。 可以通过提供一个列表值来提供一个转换列表，每项作为上述 2 个项的列表值。 有关示例，请参阅 Record.TransformFields 的描述
重命名操作	可以将记录的重命名操作指定为以下任一种： 单个重命名操作，由两个字段名称（旧名称和新名称）的列表表示。 有关示例，请参阅 Record.RenameFields 的描述。

MissingField.Error

2019/12/3 •

关于

记录和表函数中的一个可选参数，指示缺失字段将导致错误。（这是默认的参数值。）

MissingField.Ignore

2019/12/3 •

关于

记录和表函数中的一个可选参数，指示应忽略缺失字段。

MissingField.UseNull

2019/12/4 •

关于

记录和表函数中的一个可选参数，指示应将缺失字段作为 NULL 值包含在内。

Record.AddField

2019/12/3 •

语法

```
Record.AddField(record as record, fieldName as text, value as any, optional delayed as nullable
logical) as record
```

关于

给定字段 `fieldName` 的名称和值 `value`，将字段添加到记录 `record`。

示例 1

将字段地址添加到记录。

```
Record.AddField([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "Address", "123 Main St.")
```

CUSTOMERID	1
"	Bob
"	123-4567
"	123 Main St.

Record.Combine

2020/4/30 •

语法

```
Record.Combine(records as list) as record
```

关于

组合给定 `records` 中的记录。如果 `records` 包含非记录值，则返回错误。

示例 1

从记录创建组合记录。

```
Record.Combine({
    [CustomerID = 1, Name = "Bob"],
    [Phone = "123-4567"]
})
```

CUSTOMERID	1
"	Bob
"	123-4567

Record.Field

2019/12/3 •

语法

```
Record.Field(record as record, field as text) as any
```

关于

返回 `record` 中指定 `field` 的值。如果找不到该字段，则会引发异常。

示例 1

在记录中查找字段“CustomerID”的值。

```
Record.Field([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "CustomerID")
```

`1`

Record.FieldCount

2019/12/4 •

语法

```
Record.FieldCount(record as record) as number
```

关于

返回记录 `record` 中的字段数。

示例 1

查找记录中的字段数。

```
Record.FieldCount([CustomerID = 1, Name = "Bob"])
```

```
2
```

Record.FieldNames

2019/12/3 •

语法

```
Record.FieldNames(record as record) as list
```

关于

将记录 `record` 中的字段名称作为文本返回。

示例 1

查找记录中字段的名称。

```
Record.FieldNames([OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0])
```

OrderID
CustomerID
项
价格

Record.FieldOrDefault

2020/4/30 •

语法

```
Record.FieldOrDefault(record as nullable record, field as text, optional defaultValue as any) as
any
```

关于

返回记录 `record` 中指定字段 `field` 的值。如果找不到字段，则返回可选的 `defaultValue` 。

示例 1

在记录中查找字段“Phone”的值，如果它不存在，则返回 NULL。

```
Record.FieldOrDefault([CustomerID = 1, Name = "Bob"], "Phone")
```

`null`

示例 2

在记录中查找字段“Phone”的值，如果它不存在，则返回默认值。

```
Record.FieldOrDefault([CustomerID = 1, Name = "Bob"], "Phone", "123-4567")
```

`"123-4567"`

Record.FieldValues

2019/12/3 •

语法

```
Record.FieldValues(record as record) as list
```

关于

返回记录 `record` 中的字段值列表。

示例 1

在记录中查找字段值。

```
Record.FieldValues([CustomerID = 1, Name = "Bob", Phone = "123-4567"])
```

1
Bob
123-4567

Record.FromList

2019/12/4 •

语法

```
Record.FromList(list as list, fields as any) as record
```

关于

根据给定的一个字段值 `list` 和一组字段返回一个记录。可以通过文本值列表或记录类型来指定 `fields` 。如果字段不是唯一的，则会引发错误。

示例 1

从一个字段值列表和字段名称列表生成一个记录。

```
Record.FromList({1, "Bob", "123-4567"}, {"CustomerID", "Name", "Phone"})
```

CUSTOMERID	1
"	Bob
"	123-4567

示例 2

从一个字段值列表和记录类型生成一个记录。

```
Record.FromList({1, "Bob", "123-4567"}, type [CustomerID = number, Name = text, Phone = number])
```

CUSTOMERID	1
"	Bob
"	123-4567

Record.FromTable

2020/4/30 •

语法

```
Record.FromTable(table as table) as record
```

关于

返回记录表 `table` 中包含字段名称和值名称 `{[Name = name, Value = value]}` 的记录。如果字段名称不唯一，则会引发异常。

示例 1

从 Table.FromRecords({[Name = "CustomerID", Value = 1], [Name = "Name", Value = "Bob"], [Name = "Phone", Value = "123-4567"]}) 格式的表创建记录。

```
Record.FromTable(
    Table.FromRecords({
        [Name = "CustomerID", Value = 1],
        [Name = "Name", Value = "Bob"],
        [Name = "Phone", Value = "123-4567"]
    })
)
```

CUSTOMERID	1
"	Bob
"	123-4567

Record.HasFields

2020/4/30 •

语法

```
Record.HasFields(record as record, fields as any) as logical
```

关于

通过返回逻辑值(true 或 false)，指示记录 `record` 是否具有 `fields` 中指定的字段。可以使用一个列表指定多个字段值。

示例 1

检查记录是否包含字段“CustomerID”。

```
Record.HasFields([CustomerID = 1, Name = "Bob", Phone = "123-4567"], "CustomerID")
```

`true`

示例 2

检查记录是否包含字段“CustomerID”和“Address”。

```
Record.HasFields([CustomerID = 1, Name = "Bob", Phone = "123-4567"], {"CustomerID", "Address"})
```

`false`

Record.RemoveFields

2020/4/30 •

语法

```
Record.RemoveFields(record as record, fields as any, optional missingField as nullable number) as record
```

关于

返回一个记录，该记录从输入 `record` 中删除在列表 `fields` 中指定的所有字段。如果指定的字段不存在，则会引发异常。

示例 1

从记录中删除字段“Price”。

```
Record.RemoveFields([CustomerID = 1, Item = "Fishing rod", Price = 18.00], "Price")
```

CUSTOMERID	1
Item	钓鱼竿

示例 2

从记录中删除字段“Price”和“Item”。

```
Record.RemoveFields([CustomerID = 1, Item = "Fishing rod", Price = 18.00], {"Price", "Item"})
```

CUSTOMERID	1

Record.RenameFields

2020/4/30 •

语法

```
Record.RenameFields(record as record, renames as list, optional missingField as nullable number)
as record
```

关于

将 `record` 输入中的字段重命名为 `renames` 列表中指定的新字段名称后返回一条记录。对于多个重命名操作，可使用嵌套列表 ({ {old1, new1}, {old2, new2} }。

示例 1

从记录将字段 "UnitPrice" 重命名为 "Price"。

```
Record.RenameFields(
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", UnitPrice = 100.0],
    {"UnitPrice", "Price"}
)
```

ORDERID	1
CUSTOMERID	1
[illegible]	钓鱼竿
[illegible]	100

示例 2

从记录将字段 "UnitPrice" 重命名为 "Price"，将字段 "OrderNum" 重命名为 "OrderID"。

```
Record.RenameFields(
    [OrderNum = 1, CustomerID = 1, Item = "Fishing rod", UnitPrice = 100.0],
    {
        {"UnitPrice", "Price"},
        {"OrderNum", "OrderID"}
    }
)
```

ORDERID	1
CUSTOMERID	1
[illegible]	钓鱼竿

"	100

Record.ReorderFields

2020/4/30 •

语法

```
Record.ReorderFields(record as record, fieldOrder as list, optional missingField as nullable
number) as record
```

关于

按照列表 `fieldOrder` 中指定的字段顺序对 `record` 中的字段重新排序后返回一条记录。保留字段值，并且 `fieldOrder` 中未列出的字段仍保留在原始位置。

示例 1

将记录中的部分字段重新排序。

```
Record.ReorderFields(
    [CustomerID = 1, OrderID = 1, Item = "Fishing rod", Price = 100.0],
    {"OrderID", "CustomerID"}
)
```

ORDERID	1
CUSTOMERID	1
[	钓鱼竿
[[	100

Record.SelectFields

2020/4/30 •

语法

```
Record.SelectFields(record as record, fields as any, optional missingField as nullable number) as record
```

关于

从输入 `record` 返回一条记录，该记录仅包含在列表 `fields` 中指定的字段。

示例 1

选择记录中的字段“Item”和“Price”。

```
Record.SelectFields(
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
    {"Item", "Price"}
)
```

[	钓鱼竿
[[	100

Record.ToList

2019/12/4 •

语法

```
Record.ToList(record as record) as list
```

关于

返回包含输入 `record` 中的字段值的值列表。

示例 1

从记录提取字段值。

```
Record.ToList([A = 1, B = 2, C = 3])
```

1
2
3

Record.ToTable

2019/12/3 •

语法

```
Record.ToTable(record as record) as table
```

关于

返回一个表，其中包含 `Name` 和 `Value` 列，且 `record` 中的每个字段都占一行。

示例 1

从记录返回表。

```
Record.ToTable([OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0])
```

[[	[
OrderID	1
CustomerID	1
项	钓鱼竿
价格	100

Record.TransformFields

2020/4/30 •

语法

```
Record.TransformFields(record as record, transformOperations as list, optional missingField as
nullable number) as record
```

关于

在将列表 `transformOperations` 中指定的转换应用到 `record` 后返回一条记录。可以在给定时间转换一个或多个字段。

如果转换单个字段，`transformOperations` 应是包含两个项的列表。`transformOperations` 中的第一项指定字段名称，而 `transformOperations` 中的第二项指定要用于转换的函数。例如，`{"Quantity", Number.FromText}`
在转换多个字段的情况下，`transformOperations` 应为一列列表，其中每个内部列表是一对字段名称和转换操作。例如，`{{"Quantity",Number.FromText},{"UnitPrice", Number.FromText}}`

示例 1

将"Price"字段转换为数字。

```
Record.TransformFields(
    [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = "100.0"],
    {"Price", Number.FromText}
)
```

ORDERID	1
CUSTOMERID	1
Item	钓鱼竿
Price	100

示例 2

将"OrderID"和"Price"字段转换为数字。

```
Record.TransformFields(
    [OrderID ="1", CustomerID = 1, Item = "Fishing rod", Price = "100.0"],
    {{"OrderID", Number.FromText}, {"Price", Number.FromText}}
)
```

ORDERID	1
CUSTOMERID	1

[	钓鱼竿
[[	100

替换器函数

2020/4/26 •

这些函数由库中的其他函数用于替换给定值。

替换器

"	"
Replacer.ReplaceText	此函数可提供给 List.ReplaceValue 或 Table.ReplaceValue，分别用于替换列表中的文本值和表值。
Replacer.ReplaceValue	此函数可提供给 List.ReplaceValue 或 Table.ReplaceValue，分别用于替换列表中的值和表值。

Replacer.ReplaceText

2020/4/30 •

语法

```
Replacer.ReplaceText(text as nullable text, old as text, new as text) as nullable text
```

关于

将原始 `text` 中的 `old` 文本替换为 `new` 文本。此替换器函数可用于 `List.ReplaceValue` 和 `Table.ReplaceValue` 中。

示例 1

将字符串“hEllo world”中的“hE”文本替换为“He”文本。

```
Replacer.ReplaceText("hEllo world", "hE", "He")
```

```
"Hello world"
```

Replacer.ReplaceValue

2019/12/3 •

语法

```
Replacer.ReplaceValue(value as any, old as any, new as any) as any
```

关于

将原始 `value` 中的 `old` 值替换为 `new` 值。此替换器函数可用于 `List.ReplaceValue` 和 `Table.ReplaceValue` 中。

示例 1

使用值 10 替换值 11。

```
Replacer.ReplaceValue(11, 11, 10)
```

`10`

拆分器函数

2020/4/26 •

这些函数会拆分文本。

拆分器

""	""
Splitter.SplitByNothing	返回不拆分且将其参数作为单元素列表返回的函数。
Splitter.SplitTextByCharacterTransition	返回一个函数，该函数根据从一种字符到另一种字符的转换过程将文本拆分为文本列表。
Splitter.SplitTextByAnyDelimiter	返回一个函数，此函数按任何受支持的分隔符拆分文本。
Splitter.SplitTextByDelimiter	返回一个函数，此函数根据某一分隔符拆分文本。
Splitter.SplitTextByEachDelimiter	返回一个函数，此函数依次按每个分隔符拆分文本。
Splitter.SplitTextByLengths	返回一个函数，此函数根据指定长度拆分文本。
Splitter.SplitTextByPositions	返回一个函数，此函数根据指定位置拆分文本。
Splitter.SplitTextByRanges	返回一个函数，此函数根据指定范围拆分文本。
Splitter.SplitTextByRepeatedLengths	返回一个函数，此函数在指定的长度后反复将文本拆分为文本列表。
Splitter.SplitTextByWhitespace	返回一个函数，此函数根据空格拆分文本。

"""	""
QuoteStyle.Csv	引号字符指示用引号引起来的字符串的开头。嵌套引号由两个引号字符所指示。
QuoteStyle.None	引号字符没有意义。

QuoteStyle.Csv

2019/12/4 •

关于

引号字符指示用引号引起来的字符串的开头。嵌套引号由两个引号字符所指示。

QuoteStyle.None

2019/12/4 •

关于

引号字符没有意义。

Splitter.SplitByNothing

2019/12/3 •

语法

```
Splitter.SplitByNothing() as function
```

关于

返回不拆分且将其参数作为单元素列表返回的函数。

Splitter.SplitTextByAnyDelimiter

2019/12/3 •

语法

```
Splitter.SplitTextByAnyDelimiter(delimiters as list, optional quoteStyle as nullable number,
optional startAtEnd as nullable logical) as function
```

关于

返回一个函数，它在任意指定的分隔符处将文本拆分为文本列表。

Splitter.SplitTextByCharacterTransition

2019/12/3 •

语法

```
Splitter.SplitTextByCharacterTransition(before as anynonnull, after as anynonnull) as function
```

关于

返回一个函数，该函数根据从一种字符到另一种字符的转换过程将文本拆分为文本列表。`before` 和 `after` 参数可以是字符列表，也可以是接受字符并返回 true/false 的函数。

Splitter.SplitTextByDelimiter

2019/12/4 •

语法

```
Splitter.SplitTextByDelimiter(delimiter as text, optional quoteStyle as nullable number) as
function
```

关于

返回一个函数，它根据指定的分隔符将文本拆分为文本列表。

Splitter.SplitTextByEachDelimiter

2019/12/3 •

语法

```
Splitter.SplitTextByEachDelimiter(delimiters as list, optional quoteStyle as nullable number,
optional startAtEnd as nullable logical) as function
```

关于

返回一个函数，它依次在每个指定的分隔符处将文本拆分为文本列表。

Splitter.SplitTextByLengths

2019/12/4 •

语法

```
Splitter.SplitTextByLengths(lengths as list, optional startAtEnd as nullable logical) as function
```

关于

返回一个函数，它按每个指定的长度将文本拆分为文本列表。

Splitter.SplitTextByPositions

2019/12/3 •

语法

```
Splitter.SplitTextByPositions(positions as list, optional startAtEnd as nullable logical) as
function
```

关于

返回一个函数，它在每个指定的位置将文本拆分为文本列表。

Splitter.SplitTextByRanges

2019/12/4 •

语法

```
Splitter.SplitTextByRanges(ranges as list, optional startAtEnd as nullable logical) as function
```

关于

返回一个函数，它根据指定的偏移量和长度将文本拆分为文本列表。

Splitter.SplitTextByRepeatedLengths

2019/12/4 •

语法

```
Splitter.SplitTextByRepeatedLengths(length as number, optional startAtEnd as nullable logical) as function
```

关于

返回一个函数，此函数在指定的长度后反复将文本拆分为文本列表。

Splitter.SplitTextByWhitespace

2019/12/4 •

语法

```
Splitter.SplitTextByWhitespace(optional quoteStyle as nullable number) as function
```

关于

返回一个函数，它在空白处将文本拆分为文本列表。

表函数

2020/4/26 •

这些函数创建并操纵表值。

表函数

"	"
ItemExpression.From	返回某一函数主体的 AST。
ItemExpression.Item	表示项表达式中的项的一个 AST 节点。
RowExpression.Column	返回表示对行表达式内的列的访问权限的 AST。
RowExpression.From	返回某一函数主体的 AST。
RowExpression.Row	表示行表达式中的行的一个 AST 节点。
Table.FromColumns	从包含嵌套列表的列表中返回一个带有列名称和值的表。
Table.FromList	通过将指定的拆分函数应用于列表中的每一项，将列表转换为表。
Table.FromRecords	从记录列表中返回一个表。
Table.FromRows	从列表中创建一个表，其中列表的每个元素都是包含单行列值的列表。
Table.FromValue	返回一个表，此表中的某一列包含所提供的值或值列表。
Table.FuzzyGroup	按每行的指定列中模糊匹配的值对表的行进行分组。
Table.FuzzyJoin	联接两个表中基于给定键模糊匹配的行。
Table.FuzzyNestedJoin	在两个表之间对提供的列执行模糊联接，并在新列中生成联接结果。
Table.Split	使用指定页面大小将指定表拆分为一表列表。
Table.View	使用用户定义的处理程序创建或扩展表以执行查询和操作。
Table.ViewFunction	创建一个可以由在视图上（通过 `Table.View`）定义的处理程序截获的函数。

转换

Table.ToColumns	返回嵌套列表的一个列表，其中每个列表表示输入表中值的列。
Table.ToList	通过将指定的组合函数应用于表中的每一行值，将表返回到列表中。
Table.ToRecords	从输入表中返回记录列表。
Table.ToRows	从输入表中返回行值的嵌套列表。

信息

Table.ColumnCount	返回表中的列数。
Table.IsEmpty	如果表中不包含任何行，则返回 true。
Table.Profile	返回表中列的配置文件。
Table.RowCount	返回表中的行数。
Table.Schema	返回一个表，此表包含指定表的列的描述（即架构）。
Tables.GetRelationships	返回一组表之间的关系。

行操作

Table.AlternateRows	返回一个表，此表包含表中的行的交替模式。
Table.Combine	返回作为合并一个表列表的结果的表。这些表必须具有相同的行类型结构。
Table.FindText	返回一个表，此表仅包含特定行，这些行的其中一个单元格或任何部分包含了指定文本。
Table.First	返回表中的第一行。
Table.FirstN	返回表中的第一行（或前几行），具体因 countOrCondition 参数而异。
Table.FirstValue	返回表的第一行的第一列或指定的默认值。
Table.FromPartitions	将作为组合一组分区表的结果的表返回到新的列。可以选择性地指定列的类型，默认值为 any。
Table.InsertRows	返回一个表，其中包含在索引处插入到表中的行的列表。要插入的每一行都必须与表的行类型相匹配。

Table.Last	返回表的最后一行。
Table.LastN	返回表中的最后一行(或最后几行)，具体因 countOrCondition 参数而异。
Table.MatchesAllRows	如果表中的所有行均满足条件，则返回 true。
Table.MatchesAnyRows	如果表中的任一行均满足条件，则返回 true。
Table.Partition	根据每行的列值和一个哈希函数，将表分区为表的组数的一个列表。哈希函数应用于行的列值以获取此行的哈希值。哈希值的取模组决定要在其中放置行的返回表。
Table.Range	从表中返回从偏移量开始的指定行数。
Table.RemoveFirstN	返回一个表，此表从第一行开始删除了指定行数。删除的行数取决于可选的 countOrCondition 参数。
Table.RemoveLastN	返回一个表，此表从最后一行开始删除了指定行数。删除的行数取决于可选的 countOrCondition 参数。
Table.RemoveRows	返回一个表，此表从偏移量开始删除了指定行数。
Table.RemoveRowsWithErrors	返回一个表，从表中删除了至少一个单元格包含错误的所有行。
Table.Repeat	返回一个表，此表包含表中重复计数次数的行。
Table.ReplaceRows	返回一个表，其中采用了提供的行代替从偏移量开始且继续计数的行。
Table.ReverseRows	返回一个表，其中的行以相反顺序排序。
Table.SelectRows	返回一个表，此表仅包含符合条件的行。
Table.SelectRowsWithErrors	返回一个表，此表中只有至少一个单元格包含错误的行。
Table.SingleRow	从表中返回单个行。
Table.Skip	返回一个表，此表不包含表的第一行或前几行。

列操作

Table.Column	返回表中一列的值。
Table.ColumnNames	从表中返回列的名称。
Table.ColumnsOfType	返回带有与指定类型相匹配的列名称的列表。
Table.DemoteHeaders	将标题行向下降级为表的第一行。

Table.DuplicateColumn	复制具有指定名称的列。值和类型是从源列复制的。
Table.HasColumns	如果表具有指定的一列或多列，则返回 true。
Table.Pivot	给定表和包含透视值的属性列，为每个透视值创建新列并从 valueColumn 中分配它们的值。可以提供可选的 aggregationFunction 来处理属性列中多次出现的相同键值。
Table.PrefixColumns	返回一个表，其中所有列均以文本值为前缀。
Table.PromoteHeaders	将表的第一行升级为其标头或列名称。
Table.RemoveColumns	返回不包含特定一列或多列的表。
Table.ReorderColumns	返回一个表，此表包含采用与另一列相对的顺序排列的特定列。
Table.RenameColumns	返回一个表，此表中按指定内容对列进行了重命名。
Table.SelectColumns	返回仅包含特定列的表。
Table.TransformColumnNames	使用给定的函数转换列名。
Table.Unpivot	给定表列的列表，将这些列转换为属性值对。
Table.UnpivotOtherColumns	将指定集以外的所有列转换为属性值对，与每行中的剩余值相合并。

参数

JoinKind.Inner	`Table.Join` 中可选 `JoinKind` 参数的可能值。从内部联接生成的表包含根据指定键列确定为匹配的指定表中的每个行对所对应的行。
JoinKind.LeftOuter	`Table.Join` 中可选 `JoinKind` 参数的可能值。左外部联接可确保第一个表的所有行都出现在结果中。
JoinKind.RightOuter	`Table.Join` 中可选 `JoinKind` 参数的可能值。右外部联接可确保第二个表的所有行都出现在结果中。
JoinKind.FullOuter	`Table.Join` 中可选 `JoinKind` 参数的可能值。完全外部联接可确保两个表的所有行都出现在结果中。在另一表中没有匹配项的行会与包含所有列的 NULL 值的默认行联接。
JoinKind.LeftAnti	`Table.Join` 中可选 `JoinKind` 参数的可能值。左反返回第一个表中在第二个表中不具有匹配项的所有行。
JoinKind.RightAnti	`Table.Join` 中可选 `JoinKind` 参数的可能值。右反操作返回第二个表中在第一个表中不具有匹配项的所有行。

MissingField.Error	记录和表函数中的一个可选参数，指示缺失字段将导致错误。（这是默认的参数值。）
MissingField.Ignore	记录和表函数中的一个可选参数，指示应忽略缺失字段。
MissingField.UseNull	记录和表函数中的一个可选参数，指示应将缺失字段作为 NULL 值包含在内。
GroupKind.Global	GroupKind.Global
GroupKind.Local	GroupKind.Local
ExtraValues.List	如果拆分器函数返回的列比表所需的列多，应将它们收集到列表中。
ExtraValues.Ignore	如果拆分器函数返回的列比表所需的列多，应忽略它们。
ExtraValues.Error	如果拆分器函数返回的列数多于表的预期值，会引发错误。
JoinAlgorithm.Dynamic	JoinAlgorithm.Dynamic
JoinAlgorithm.PairwiseHash	JoinAlgorithm.PairwiseHash
JoinAlgorithm.SortMerge	JoinAlgorithm.SortMerge
JoinAlgorithm.LeftHash	JoinAlgorithm.LeftHash
JoinAlgorithm.RightHash	JoinAlgorithm.RightHash
JoinAlgorithm.LeftIndex	JoinAlgorithm.LeftIndex
JoinAlgorithm.RightIndex	JoinAlgorithm.RightIndex
JoinSide.Left	指定联接的左表。
JoinSide.Right	指定联接的右表。

转换

组选项的参数

- GroupKind.Global = 0;
- GroupKind.Local = 1;

联接类型的参数

- JoinKind.Inner = 0;
- JoinKind.LeftOuter = 1;
- JoinKind.RightOuter = 2;
- JoinKind.FullOuter = 3;

- JoinKind.LeftAnti = 4;
- JoinKind.RightAnti = 5

联接算法

可以将以下 JoinAlgorithm 值指定为 Table.Join

```
JoinAlgorithm.Dynamic        0,
```

```
JoinAlgorithm.PairwiseHash   1,
```

```
JoinAlgorithm.SortMerge      2,
```

```
JoinAlgorithm.LeftHash       3,
```

```
JoinAlgorithm.RightHash      4,
```

```
JoinAlgorithm.LeftIndex      5,
```

```
JoinAlgorithm.RightIndex     6,
```

[[[	[[
JoinSide.Left	指定联接的左表。
JoinSide.Right	指定联接的右表。

示例数据

本部分中的示例使用了下表。

客户表

```
Customers = Table.FromRecords({
  [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
  [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
  [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
  [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
}
```

订单表

```
Orders = Table.FromRecords({
  [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
  [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
  [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
  [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
  [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
  [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
  [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
  [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
  [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
})
```

"	"
Table.AddColumn	将名为 newColumnName 的列添加到表中。
Table.AddIndexColumn	返回一个表，此表包含带有指定名称的新列，其中每行都包含表中的行的索引。
Table.AddJoinColumn	从特定列执行 table1 和 table2 之间的嵌套联接，并为 table1 的每一行生成作为 newColumnName 列的联接结果。
Table.AddKey	将键添加到表。
Table.AggregateTableColumn	将嵌套在特定列中的表聚合到包含这些表的聚合值的多列中。
Table.CombineColumns	Table.CombineColumns 使用组合程序函数合并列来生成新列。Table.CombineColumns 与 Table.SplitColumns 相反。
Table.ExpandListColumn	给定表中的列表列，为其列表中的每个值创建行的副本。
Table.ExpandRecordColumn	将记录列扩展为具有每个值的列。
Table.ExpandTableColumn	将一个记录列或表列扩展到包含表中的多列。
Table.FillDown	使用列中的最新非 NULL 值替换表中指定一列或多列中的 NULL 值。
Table.FillUp	从指定的表中返回一个表，其中下一个单元格的值会传播到指定的列中上面值为 NULL 的单元格。
Table.FilterWithDataTable	
Table.Group	按每行的键列的值对表行进行分组。
Table.Join	基于 table1、key1 以及 table2、key2 选择的键列的值的相等性，联接 table1 的行和 table2 的行。

Table.Keys	返回表中键列名称的列表。
Table.NestedJoin	基于键的相等性联接表中的行。结果会输入到新列中。
Table.ReplaceErrorValues	使用相应的指定值替换指定列中的错误值。
Table.ReplaceKeys	返回一个新表，此表具有在键参数中设置的新键信息。
Table.ReplaceRelationshipIdentity	
Table.ReplaceValue	通过使用提供的替换器函数(例如 text.Replace 或 Value.Replace)，用 newValue 替换表的特定列中的 oldValue。
Table.SplitColumn	将拆分器函数应用于每个值来从单个列中返回新的一组列。
Table.TransformColumns	使用函数转换表中的列。
Table.TransformColumnTypes	使用类型转换表中的列类型。
Table.TransformRows	使用转换函数转换表中的行。
Table.Transpose	返回一个表，此表将输入表中的列转换为了行，将行转换为了列。

成员身份

用于成员身份检查的参数

匹配项规范

```
Occurrence.First  = 0
```

```
Occurrence.Last   = 1
```

```
Occurrence.All    = 2
```

Table.Contains	确定记录是否显示为表中的一行。
Table.ContainsAll	确定所有指定的记录是否显示为表中的各行。
Table.ContainsAny	确定任何指定的记录是否显示为表中的各行。
Table.Distinct	删除表中的重复行，从而确保留下的所有行都是不同的。
Table.IsDistinct	确定表是否仅包含非重复行。

Table.PositionOf	确定行在表中的一个或多个位置。
Table.PositionOfAny	确定任何指定的行在表中的一个或多个位置。
Table.RemoveMatchingRows	从表中删除行的所有匹配项。
Table.ReplaceMatchingRows	使用新行替换表中的特定行。

排序

示例数据

本部分中的示例使用了下表。

员工表

```
Employees = Table.FromRecords(
    {[Name="Bill",   Level=7,  Salary=100000],
     [Name="Barb",   Level=8,  Salary=150000],
     [Name="Andrew", Level=6,  Salary=85000],
     [Name="Nikki",  Level=5,  Salary=75000],
     [Name="Margo",  Level=3,  Salary=45000],
     [Name="Jeff",   Level=10, Salary=200000]},
type table [
    Name = text,
    Level = number,
    Salary = number
])
```

Table.Max	使用 comparisonCriteria 返回表中的最大的行或最大的几行。
Table.MaxN	返回表中最大的 N 行。对行进行排序后，必须指定 countOrCondition 参数以进一步筛选结果。
Table.Min	使用 comparisonCriteria 返回表中的最小的行或最小的几行。
Table.MinN	返回给定表中最小的 N 行。对行进行排序后，必须指定 countOrCondition 参数以进一步筛选结果。
Table.Sort	使用 comparisonCriteria 或默认顺序（若未指定）对表中的行进行排序。

其他

"	"
Table.Buffer	在内存中缓冲一个表，从而在计算期间将其与外部更改相隔离。

参数值

为输出列命名

此参数为文本值的一个列表，指定生成的表的列名。此参数通常用在 Table 构造函数中，例如 Table.FromRows 和 Table.FromList。

比较条件

可将比较条件提供为以下值之一：

- 用于指定排序顺序的一个数字值。请参阅上述"参数值"一节中的排序顺序。
- 若要计算用于排序的键，可以使用具有 1 个参数的函数。
- 若要选择键并控制顺序，比较条件可以是包含键和顺序的列表。
- 若要完全控制比较，可以使用具有 2 个参数的函数，此函数将根据左输入和右输入之间的关系返回 -1、0 或 1。Value.Compare 是可用于委托此逻辑的方法。

有关示例，请参阅 Table.Sort 的说明。

Count 或 Condition 条件

此条件通常用于排序或行操作。它确定表中返回的行数，并且可以采用数字或条件两种形式：

- 数字指示适当函数中返回的值的数量
- 如果指定了条件，则返回包含的值最初满足条件的行。一旦有一个值不符合条件，则不再考虑其他值。

请参阅 Table.FirstN 或 Table.MaxN。

额外值的处理

这用于指示函数应如何处理行中的额外值。此参数指定为一个数字，此数值会映射到下面的选项。

```
ExtraValues.List = 0

ExtraValues.Error = 1

ExtraValues.Ignore = 2
```

有关详细信息，请参阅 Table.FromList。

缺少列处理

这用于指示函数应如何处理缺少的列。此参数指定为一个数字，此数值会映射到下面的选项。

```
MissingField.Error = 0;

MissingField.Ignore = 1;

MissingField.UseNull = 2;
```

这用于列或转换操作。有关示例，请参阅 Table.TransformColumns。

排序顺序

这用于指示结果的排序方式。此参数指定为一个数字，此数值会映射到下面的选项。

```
Order.Ascending = 0

    Order.Descending = 1
```

相等条件

可将表的相等条件指定为

- 一个函数值，它可以是
  - 一个键选择器，确定表中要应用相等条件的列，或
  - 一个比较器函数，用于指定要应用的比较类型。可以指定内置的比较器函数，请参阅"比较器函数"一节。
- 一个列表，列出了要应用相等条件的列

有关示例，请查看 Table.Distinc 的说明。

ExtraValues.Error

2019/12/3 •

关于

如果拆分器函数返回的列数多于表的预期值，会引发错误。

ExtraValues.Ignore

2019/12/3 •

关于

如果拆分器函数返回的列比表所需的列多，应忽略它们。

ExtraValues.List

2019/12/3 •

关于

如果拆分器函数返回的列比表所需的列多，应将它们收集到列表中。

GroupKind.Global

2019/12/3 •

关于

语法

```
GroupKind.Global
```

GroupKind.Local

2019/12/3 •

关于

语法

```
GroupKind.Local
```

关于

GroupKind.Local

ItemExpression.From

2019/12/3 •

语法

```
ItemExpression.From(function as function) as record
```

关于

返回 `function` 的主体的 AST，规范化为“项表达式”：

- 函数必须是包含 1 个参数的 lambda 函数。
- 对函数参数的所有引用都将替换为 `ItemExpression.Item`。
- AST 将简化为仅包含以下类型的节点：
  - `Constant`
  - `Invocation`
  - `Unary`
  - `Binary`
  - `If`
  - `FieldAccess`
  - `NotImplemented`

如果 `function` 的主体无法返回项表达式 AST，则会引发错误。

示例 1

返回函数 `each _ <> null` 的主体的 AST

```
ItemExpression.From(each _ <> null)
```

II	二进制
III	NotEquals
I	[记录]
RIGHT	[记录]

ItemExpression.Item

2019/12/3 •

关于

表示项表达式中的项的一个 AST 节点。

JoinAlgorithm.Dynamic

2019/12/3 •

关于

JoinAlgorithm.Dynamic

JoinAlgorithm.LeftHash

2019/12/4 •

关于

JoinAlgorithm.LeftHash

JoinAlgorithm.LeftIndex

2019/12/4 •

关于

JoinAlgorithm.LeftIndex

JoinAlgorithm.PairwiseHash

2019/12/4 •

关于

JoinAlgorithm.PairwiseHash

JoinAlgorithm.RightHash

2019/12/3 •

关于

JoinAlgorithm.RightHash

JoinAlgorithm.RightIndex

2019/12/3 •

关于

JoinAlgorithm.RightIndex

JoinAlgorithm.SortMerge

2019/12/4 •

关于

JoinAlgorithm.SortMerge

JoinKind.FullOuter

2019/12/4 •

关于

`Table.Join` 中可选 `JoinKind` 参数的可能值。完全外部联接可确保两个表的所有行都出现在结果中。在另一表中没有匹配项的行会与包含所有列的 NULL 值的默认行联接。

JoinKind.Inner

2019/12/3 •

关于

`Table.Join` 中可选 `JoinKind` 参数的可能值。从内部联接生成的表包含根据指定键列确定为匹配的指定表中的每个行对所对应的行。

JoinKind.LeftAnti

2019/12/4 •

关于

`Table.Join` 中可选 `JoinKind` 参数的可能值。左反返回第一个表中在第二个表中不具有匹配项的所有行。

JoinKind.LeftOuter

2019/12/3 •

关于

`Table.Join` 中可选 `JoinKind` 参数的可能值。左外部联接可确保第一个表的所有行都出现在结果中。

JoinKind.RightAnti

2019/12/3 •

关于

`Table.Join` 中可选 `JoinKind` 参数的可能值。右反操作返回第二个表中在第一个表中不具有匹配项的所有行。

JoinKind.RightOuter

2019/12/4 •

关于

`Table.Join` 中可选 `JoinKind` 参数的可能值。右外部联接可确保第二个表的所有行都出现在结果中。

JoinSide.Left

2019/12/4 •

关于

指定联接的左表。

JoinSide.Right

2019/12/3 •

关于

指定联接的右表。

Occurrence.All

2019/12/3 •

关于

返回找到的值所有实例的位置列表。

Occurrence.First

2019/12/3 •

关于

返回找到的值第一次出现的位置。

Occurrence.Last

2019/12/4 •

关于

返回找到的值最后一次出现的位置。

Order.Ascending

2019/12/3 •

关于

以升序对列表进行排序的函数类型。

Order.Descending

2019/12/3 •

关于

以降序对列表排序的函数类型。

RowExpression.Column

2019/12/4 •

语法

```
RowExpression.Column(columnName as text) as record
```

关于

返回表示对行表达式内的列 columnName 的访问权限的 AST。

示例 1

创建表示对列“CustomerName”的访问权限的 AST。

```
RowExpression.Column("CustomerName")
```

[]	FieldAccess
[[]]	[记录]
MEMBERNAME	CustomerName

RowExpression.From

2019/12/4 •

语法

```
RowExpression.From(function as function) as record
```

关于

返回 `function` 主体的 AST，规范化为*行表达式*：

- 函数必须是包含 1 个参数的 lambda 函数。
- 对函数参数的所有引用都将替换为 `RowExpression.Row`。
- 对列的所有引用都将替换为 `RowExpression.Column(columnName)`。
- AST 将简化为仅包含以下类型的节点：
  - `Constant`
  - `Invocation`
  - `Unary`
  - `Binary`
  - `If`
  - `FieldAccess`
  - `NotImplemented`

如果无法为 `function` 的主体返回行表达式 AST，则会引发错误。

示例 1

返回函数 each [CustomerID] = "ALFKI" 主体的 AST

```
RowExpression.From(each [CustomerName] = "ALFKI")
```

`11`	二进制
`111`	等于
`1`	[记录]
RIGHT	[记录]

RowExpression.Row

2019/12/3 •

关于

表示行表达式中的行的一个 AST 节点。

Table.AddColumn

2020/4/30 •

语法

```
Table.AddColumn(table as table, newColumnName as text, columnGenerator as function, optional
columnType as nullable type) as table
```

关于

将名为 `newColumnName` 的列添加到表 `table`。使用指定的选择函数 `columnGenerator`（它将每行作为输入）来计算列的值。

示例 1

将名为“总价”的列添加到表，其中每个值是列 [价格] 和列 [运费] 的和。

```
Table.AddColumn(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0, Shipping = 10.00],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0, Shipping = 15.00],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0, Shipping = 10.00]
    }),
    "TotalPrice",
    each [Price] + [Shipping]
)
```

ORDERID	CUSTOMERID	[illegible]	[illegible]	[illegible]	[illegible]
1	1	钓鱼竿	100	10	110
2	1	1 磅蠕虫	5	15	20
3	2	渔网	25	10	35

Table.AddIndexColumn

2020/4/30 •

语法

```
Table.AddIndexColumn(table as table, newColumnName as text, optional initialValue as nullable number, optional increment as nullable number) as table
```

关于

使用显式位置值将名为 `newColumnName` 的列追加到 `table` 中。一个初始索引值 `initialValue`（可选值）。一个可选值 `increment`，指定每个索引值的增量。

示例 1

将名为"Index"的索引列添加到表。

```
Table.AddIndexColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Index"
)
```

CUSTOMERID	"	"	"
1	Bob	123-4567	0
2	Jim	987-6543	1
3	Paul	543-7890	2
4	Ringo	232-1550	3

示例 2

从值 10 开始、按 5 递增将名为"Index"的索引列添加到表。

```
Table.AddIndexColumn(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Index",
    10,
    5
)
```

CUSTOMERID	"	"	"
1	Bob	123-4567	10
2	Jim	987-6543	15
3	Paul	543-7890	20
4	Ringo	232-1550	25

Table.AddJoinColumn

2020/4/30 •

语法

```
Table.AddJoinColumn(table1 as table, key1 as any, table2 as function, key2 as any, newColumnName
as text) as table
```

关于

基于 `key1`（对于 `table1`）和 `key2`（对于 `table2`）所选择的键列的值的相等性，联接 `table1` 的行与 `table2` 的行。结果会输入到名为 `newColumnName` 的列中。此函数的行为类似于 TableJoin（具有 LeftOuter 的 JoinKind），不同之处在于联接结果以嵌套而不是平展的方式呈现。

示例 1

从已联接 [saleID] 的表 ({[saleID = 1, price = 20], [saleID = 2, price = 10]}) 将名为“price/stock”的联接列添加到 ({[saleID = 1, item = "Shirt"], [saleID = 2, item = "Hat"]})。

```
Table.AddJoinColumn(
    Table.FromRecords({
        [saleID = 1, item = "Shirt"],
        [saleID = 2, item = "Hat"]
    }),
    "saleID",
    () => Table.FromRecords({
        [saleID = 1, price = 20, stock = 1234],
        [saleID = 2, price = 10, stock = 5643]
    }),
    "saleID",
    "price"
)
```

SALEID	[illegible]	[illegible]
1	T 恤	[表]
2	帽子	[表]

Table.AddKey

2020/4/30 •

语法

```
Table.AddKey(table as table, columns as list, isPrimary as logical) as table
```

关于

将一个键添加到 `table`，给定的 `columns` 是定义该键的 `table` 的列名称的子集，并且 `isPrimary` 指定该键是否为主键。

示例 1

向包含 {"Id"} 的 {[Id = 1, Name = "Hello There"], [Id = 2, Name = "Good Bye"]} 添加键并使其成为主键。

```
let
    tableType = type table [Id = Int32.Type, Name = text],
    table = Table.FromRecords({
        [Id = 1, Name = "Hello There"],
        [Id = 2, Name = "Good Bye"]
    }),
    resultTable = Table.AddKey(table, {"Id"}, true)
in
    resultTable
```

ID	ıı
1	Hello There
2	Good Bye

Table.AggregateTableColumn

2020/4/30 •

语法

```
Table.AggregateTableColumn(table as table, column as text, aggregations as list) as table
```

关于

将 `table` [`column`] 中的表聚合到包含这些表的聚合值的多个列中。 `aggregations` 用于指定包含要聚合的表的列、要应用于表以生成其值的聚合函数以及要创建的聚合列的名称。

示例 1

将表 `{[t = {[a=1, b=2, c=3], [a=2,b=4,c=6]}, b = 2]}` 的 `[t]` 中的表列聚合为 `[t.a]` 的总和、`[t.b]` 的最小值和最大值以及 `[t.a]` 中的值的计数。

```
Table.AggregateTableColumn(
    Table.FromRecords(
        {
            [
                t = Table.FromRecords({
                    [a = 1, b = 2, c = 3],
                    [a = 2, b = 4, c = 6]
                }),
                b = 2
            ]
        },
        type table [t = table [a = number, b = number, c = number], b = number]
    ),
    "t",
    {
        {"a", List.Sum, "sum of t.a"},
        {"b", List.Min, "min of t.b"},
        {"b", List.Max, "max of t.b"},
        {"a", List.Count, "count of t.a"}
    }
)
```

T.A [t]	T.B [t]	T.B [t]	T.A [t]	B
3	2	4	2	2

Table.AlternateRows

2020/4/30 •

语法

```
Table.AlternateRows(table as table, offset as number, skip as number, take as number) as table
```

关于

保留初始偏移量，然后交替选取和跳过下列行。

- `table`：输入表。
- `offset`：要在开始迭代前保留的行数。
- `skip`：要在每次迭代中删除的行数。
- `take`：要在每次迭代中保留的行数。

示例 1

从表中创建一个表，从第一行开始，跳过 1 个值，然后保留 1 个值。

```
Table.AlternateRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }),
    1,
    1,
    1
)
```

CUSTOMERID	ıı	ıı
1	Bob	123-4567
3	Paul	543-7890

Table.Buffer

2019/12/3 •

语法

```
Table.Buffer(table as table) as table
```

关于

在内存中缓冲一个表，同时在计算期间使其与外部更改隔离。

Table.Column

2020/4/30 •

语法

```
Table.Column(table as table, column as text) as list
```

关于

将表 `table` 中由 `column` 指定的数据列返回为列表。

示例 1

返回表中 [Name] 列的值。

```
Table.Column(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Name"
)
```

Bob
Jim
Paul
Ringo

Table.ColumnCount

2020/4/30 •

语法

```
Table.ColumnCount(table as table) as number
```

关于

返回表 `table` 中的列数。

示例 1

查找表中的列数。

```
Table.ColumnCount(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

`3`

Table.ColumnNames

2020/4/30 •

语法

```
Table.ColumnNames(table as table) as list
```

关于

以文本列表形式返回表 `table` 中的列名称。

示例 1

查找表的列名称。

```
Table.ColumnNames(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)
```

CustomerID
名称
电话

Table.ColumnsOfType

2020/4/30 •

语法

```
Table.ColumnsOfType(table as table, listOfTypes as list) as list
```

关于

返回带有表 `table` 中与 `listOfTypes` 中指定的类型相匹配的列名的列表。

示例 1

返回表中 Number.Type 类型的列名。

```
Table.ColumnsOfType(
    Table.FromRecords(
        {[a = 1, b = "hello"]},
        type table[a = Number.Type, b = Text.Type]
    ),
    {type number}
)
```

一个

Table.Combine

2020/4/30 •

语法

```
Table.Combine(tables as list, optional columns as any) as table
```

关于

返回一个表，此表是合并表 `tables` 的列表的结果。生成的表的行类型结构将由 `columns` 定义，或由输入类型的联合定义（如果未指定 `columns`）。

示例 1

将以下三个表合并在一起。

```
Table.Combine({
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    Table.FromRecords({[CustomerID = 2, Name = "Jim", Phone = "987-6543"]}),
    Table.FromRecords({[CustomerID = 3, Name = "Paul", Phone = "543-7890"]})
})
```

CUSTOMERID	"	"
1	Bob	123-4567
2	Jim	987-6543
3	Paul	543-7890

示例 2

合并结构不同的三个表。

```
Table.Combine({
    Table.FromRecords({[Name = "Bob", Phone = "123-4567"]}),
    Table.FromRecords({[Fax = "987-6543", Phone = "838-7171"]}),
    Table.FromRecords({[Cell = "543-7890"]})
})
```

"	"	"	"
Bob	123-4567		
	838-7171	987-6543	
			543-7890

示例 3

合并两个表并投影到给定类型。

```
Table.Combine(
    {
        Table.FromRecords({[Name = "Bob", Phone = "123-4567"]}),
        Table.FromRecords({[Fax = "987-6543", Phone = "838-7171"]}),
        Table.FromRecords({[Cell = "543-7890"]})
    },
    {"CustomerID", "Name"}
)
```

CUSTOMERID	ıı
	Bob

Table.CombineColumns

2019/12/3 •

语法

```
Table.CombineColumns(table as table, sourceColumns as list, combiner as function, column as text)
as table
```

关于

使用指定的组合程序函数将指定的列组合为一个新列。

Table.Contains

2020/4/30 •

语法

```
Table.Contains(table as table, row as record, optional equationCriteria as any) as logical
```

关于

指示指定的记录 `row` 是否显示为 `table` 中的一行。可以指定一个可选参数 `equationCriteria` 以控制表中各行之间的比较。

示例 1

确定表是否包含行。

```
Table.Contains(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    [Name = "Bob"]
)
```

`true`

示例 2

确定表是否包含行。

```
Table.Contains(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    [Name = "Ted"]
)
```

`false`

示例 3

确定表是否包含只比较 [Name] 列的行。

```
Table.Contains(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    [CustomerID = 4, Name = "Bob"],
    "Name"
)
```

`true`

Table.ContainsAll

2020/4/30 •

语法

```
Table.ContainsAll(table as table, rows as list, optional equationCriteria as any) as logical
```

关于

指示记录列表 `rows` 中的所有指定记录是否在 `table` 中显示为行。可以指定一个可选参数 `equationCriteria` 以控制表中各行之间的比较。

示例 1

确定表是否包含所有行，同时只比较列 [CustomerID]。

```
Table.ContainsAll(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    {
        [CustomerID = 1, Name = "Bill"],
        [CustomerID = 2, Name = "Fred"]
    },
    "CustomerID"
)
```

`true`

示例 2

确定表是否包含所有行。

```
Table.ContainsAll(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    {
        [CustomerID = 1, Name = "Bill"],
        [CustomerID = 2, Name = "Fred"]
    }
)
```

`false`

Table.ContainsAny

2020/4/30 •

语法

```
Table.ContainsAny(table as table, rows as list, optional equationCriteria as any) as logical
```

关于

指示记录 `rows` 列表中的任何指定记录是否在 `table` 中显示为行。可以指定一个可选参数 `equationCriteria` 以控制表中各行之间的比较。

示例 1

确定表 `({[a = 1, b = 2], [a = 3, b = 4]})` 是否包含行 `[a = 1, b = 2]` 或 `[a = 3, b = 5]`。

```
Table.ContainsAny(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {
        [a = 1, b = 2],
        [a = 3, b = 5]
    }
)
```

`true`

示例 2

确定表 `({[a = 1, b = 2], [a = 3, b = 4]})` 是否包含行 `[a = 1, b = 3]` 或 `[a = 3, b = 5]`。

```
Table.ContainsAny(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {
        [a = 1, b = 3],
        [a = 3, b = 5]
    }
)
```

`false`

示例 3

确定表 `(Table.FromRecords({[a = 1, b = 2], [a = 3, b = 4]}))` 是否包含只比较列 [a] 的行 `[a = 1, b = 3]` 或 `[a = 3, b = 5]`。

```
Table.ContainsAny(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    {
        [a = 1, b = 3],
        [a = 3, b = 5]
    },
    "a"
)
```

`true`

Table.DemoteHeaders

2020/4/30 •

语法

```
Table.DemoteHeaders(table as table) as table
```

关于

将列标题（即列名称）降级为值的第一行。默认列名称为“Column1”、“Column2”等。

示例 1

降级表中的第一行值。

```
Table.DemoteHeaders(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    })
)
```

[1	[2	COLUMN3
CustomerID	名称	电话
1	Bob	123-4567
2	Jim	987-6543

Table.Distinct

2020/4/30 •

语法

```
Table.Distinct(table as table, optional equationCriteria as any) as table
```

关于

从表 `table` 中删除重复的行。可选参数 `equationCriteria` 指定对表中的哪些列进行测试以确定是否有重复项。如果未指定 `equationCriteria`，则测试所有列。

示例 1

从表中删除重复行。

```
Table.Distinct(
    Table.FromRecords({
        [a = "A", b = "a"],
        [a = "B", b = "b"],
        [a = "A", b = "a"]
    })
)
```

A	B
A	一个
B	b

示例 2

从表 `({[a = "A", b = "a"], [a = "B", b = "a"], [a = "A", b = "b"]})` 的列 [b] 中删除重复行。

```
Table.Distinct(
    Table.FromRecords({
        [a = "A", b = "a"],
        [a = "B", b = "a"],
        [a = "A", b = "b"]
    }),
    "b"
)
```

A	B
A	一个
A	b

Table.DuplicateColumn

2020/4/30 •

语法

```
Table.DuplicateColumn(table as table, columnName as text, newColumnName as text, optional
columnType as nullable type) as table
```

关于

将名为 `columnName` 的列复制到表 `table`。列 `newColumnName` 的值和类型是从列 `columnName` 复制的。

示例

将列"a"复制到表 `({[a = 1, b = 2], [a = 3, b = 4]})` 中名为"copied column"的列。

```
Table.DuplicateColumn(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4]
    }),
    "a",
    "copied column"
)
```

A	B	A
1	2	1
3	4	3

Table.ExpandListColumn

2020/4/30 •

语法

```
Table.ExpandListColumn(table as table, column as text) as table
```

关于

给定一个 `table`，其中 `column` 是一个值列表，将该列表拆分为每个值对应的一行。其他列中的值在创建的每个新行中都是重复的。

示例 1

拆分表中的列表列 [Name]。

```
Table.ExpandListColumn(
    Table.FromRecords({[Name= {"Bob", "Jim", "Paul"}, Discount = .15]}),
    "Name"
)
```

“	“
Bob	0.15
Jim	0.15
Paul	0.15

Table.ExpandRecordColumn

2020/4/30 •

语法

```
Table.ExpandRecordColumn(table as table, column as text, fieldNames as list, optional
newColumnNames as nullable list) as table
```

关于

给定输入 `table` 中的记录 `column`，创建一个表，其中包含对应记录中每个字段的列。还可以指定 `newColumnNames` 以确保新表中的列具有唯一名称。

- `table`：含要展开的记录列的原始表。
- `column`：要展开的列。
- `fieldNames`：要展开为表中列的字段的列表。
- `newColumnNames`：要提供给新列的列名的列表。新列名不能与新表中的任何列重复。

示例 1

将表 `({[a = [aa = 1, bb = 2, cc = 3], b = 2]})` 中的列 [a] 扩展为“aa”、“bb”和“cc”3 列。

```
Table.ExpandRecordColumn(
    Table.FromRecords({
        [
            a = [aa = 1, bb = 2, cc = 3],
            b = 2
        ]
    }),
    "a",
    {"aa", "bb", "cc"}
)
```

AA	BB	CC	B
1	2	3	2

Table.ExpandTableColumn

2020/4/30 •

语法

```
Table.ExpandTableColumn(table as table, column as text, columnNames as list, optional
newColumnNames as nullable list) as table
```

关于

将 `table` [`column`] 中的表展开为多个行和列。 `columnNames` 用于从内部表中选择要扩展的列。指定 `newColumnNames` 以避免现有列与新列之间的冲突。

示例 1

在表 `({[t = {[a=1, b=2, c=3], [a=2,b=4,c=6]}, b = 2]})` 中将 `[a]` 中的表列展开为 3 列，分别为 `[t.a]`、`[t.b]` 和 `[t.c]`。

```
Table.ExpandTableColumn(
    Table.FromRecords({
        [
            t = Table.FromRecords({
                [a = 1, b = 2, c= 3],
                [a = 2, b = 4, c = 6]
            }),
            b = 2
        ]
    }),
    "t",
    {"a", "b", "c"},
    {"t.a", "t.b", "t.c"}
)
```

T.A	T.B	T.C	B
1	2	3	2
2	4	6	2

Table.FillDown

2020/4/30 •

语法

```
Table.FillDown(table as table, columns as list) as table
```

关于

从指定的 `table` 中返回一个表，其中前一个单元格的值会传播到指定的 `columns` 中下方值为 Null 的单元格。

示例 1

从表返回一个表，其中，列 [Place] 中的 NULL 值是使用这些值上方的值填充的。

```
Table.FillDown(
    Table.FromRecords({
        [Place = 1, Name = "Bob"],
        [Place = null, Name = "John"],
        [Place = 2, Name = "Brad"],
        [Place = 3, Name = "Mark"],
        [Place = null, Name = "Tom"],
        [Place = null, Name = "Adam"]
    }),
    {"Place"}
)
```

“	“
1	Bob
1	John
2	Brad
3	标记
3	Tom
3	Adam

Table.FillUp

2020/4/30 •

语法

```
Table.FillUp(table as table, columns as list) as table
```

关于

从指定的 `table` 中返回一个表，其中下一个单元的值传播到指定的 `columns` 上面值为 NULL 的单元。

示例 1

从表返回一个表，其中，列 [Column2] 中的 NULL 值使用这些值下方的值填充。

```
Table.FillUp(
    Table.FromRecords({
        [Column1 = 1, Column2 = 2],
        [Column1 = 3, Column2 = null],
        [Column1 = 5, Column2 = 3]
    }),
    {"Column2"}
)
```

[1	[2
1	2
3	3
5	3

Table.FilterWithDataTable

2019/12/3 •

语法

```
Table.FilterWithDataTable(**table** as table, **dataTableIdentifier** as text) as any
```

关于

Table.FilterWithDataTable

Table.FindText

2020/4/30 •

语法

```
Table.FindText(table as table, text as text) as table
```

关于

返回表 `table` 中包含文本 `text` 的行。如果未找到文本，则返回一个空表。

示例 1

查找表中包含"Bob"的行。

```
Table.FindText(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Bob"
)
```

CUSTOMERID	"	"
1	Bob	123-4567

Table.First

2020/4/30 •

语法

```
Table.First(table as table, optional default as any) as any
```

关于

返回 `table` 的第一行，或如果表为空，则返回可选默认值 `default`。

示例 1

查找表的第一行。

```
Table.First(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

CUSTOMERID	1
II	Bob
II	123-4567

示例 2

查找表 `({})` 的第一行，或如果为空，则返回 [a = 0, b = 0]。

```
Table.First(Table.FromRecords({}), [a = 0, b = 0])
```

II	0
B	0

Table.FirstN

2020/4/30 •

语法

```
Table.FirstN(table as table, countOrCondition as any) as table
```

关于

根据 `countOrCondition` 的值，返回 `table` 表的第一行：

- 如果 `countOrCondition` 是数字，则返回多个行（从顶部开始）。
- 如果 `countOrCondition` 是条件，则返回满足条件的行，直到某一行不满足条件为止。

示例 1

查找表的前两行。

```
Table.FirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }),
    2
)
```

CUSTOMERID	[illegible]	[illegible]
1	Bob	123-4567
2	Jim	987-6543

示例 2

查找表中 [a] > 0 的前几行。

```
Table.FirstN(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4],
        [a = -5, b = -6]
    }),
    each [a] > 0
)
```

[illegible]	B
1	2

3	4

Table.FirstValue

2019/12/4 •

语法

```
Table.FirstValue(table as table, optional default as any) as any
```

关于

返回表 `table` 的第一行的第一列或指定的默认值。

Table.FromColumns

2020/4/30 •

语法

```
Table.FromColumns(lists as list, optional columns as any) as table
```

关于

从包含嵌套列表的列表 `lists` 中创建一个类型为 `columns` 的表，此表中具有列名称和值。如果某些列具有的值比其他列多，将使用默认值“null”填充其他列缺失的值（如果列可为空）。

示例 1

从列表中的客户名称列表返回一个表。客户列表项中的每个值都将成为行值，每个列表都将成为列。

```
Table.FromColumns({
    {1, "Bob", "123-4567"},
    {2, "Jim", "987-6543"},
    {3, "Paul", "543-7890"}
})
```

[1	[2	COLUMN3
1	2	3
Bob	Jim	Paul
123-4567	987-6543	543-7890

示例 2

从给定的列列表和列名列表创建一个表。

```
Table.FromColumns(
    {
        {1, "Bob", "123-4567"},
        {2, "Jim", "987-6543"},
        {3, "Paul", "543-7890"}
    },
    {"CustomerID", "Name", "Phone"}
)
```

CUSTOMERID	[[	[[
1	2	3
Bob	Jim	Paul

123-4567	987-6543	543-7890

示例 3

创建一个表，其中每行的列数不同。缺少的行值为 NULL。

```
Table.FromColumns(
    {
        {1, 2, 3},
        {4, 5},
        {6, 7, 8, 9}
    },
    {"column1", "column2", "column3"}
)
```

COLUMN1	COLUMN2	COLUMN3
1	4	6
2	5	7
3		8
		9

Table.FromList

2020/4/30 •

语法

```
Table.FromList(list as list, optional splitter as nullable function, optional columns as any,
optional default as any, optional extraValues as nullable number) as table
```

关于

通过将可选的拆分函数 `splitter` 应用于列表中的每一项，将列表 `list` 转换为表。默认情况下，该列表假定为以逗号分隔的文本值列表。可选 `columns` 可以为列数、列列表或 TableType。还可以指定可选的 `default` 和 `extraValues`。

示例 1

使用默认拆分器从列表中创建一个列名为“Letters”的表。

```
Table.FromList({"a", "b", "c", "d"}, null, {"Letters"})
```

LETTERS
一个
b
c
d

示例 2

使用 Record.FieldValues 拆分器从列表创建一个表，结果表的列名为“CustomerID”和“Name”。

```
Table.FromList(
    {
        [CustomerID = 1, Name = "Bob"],
        [CustomerID = 2, Name = "Jim"]
    },
    Record.FieldValues,
    {"CustomerID", "Name"}
)
```

CUSTOMERID	[illegible]
1	Bob
2	Jim

Table.FromPartitions

2020/4/30 •

语法

```
Table.FromPartitions(partitionColumn as text, partitions as list, optional partitionColumnType as nullable type) as table
```

关于

返回一个表，该表是组合一组分区表 `partitions` 的结果。 `partitionColumn` 是要添加的列的名称。列的类型默认为 `any`，但可以通过 `partitionColumnType` 指定。

示例 1

从列表 `{number}` 中查找项类型。

```
Table.FromPartitions(
    "Year",
    {
        {
            1994,
            Table.FromPartitions(
                "Month",
                {
                    {
                        "Jan",
                        Table.FromPartitions(
                            "Day",
                            {
                                {1, #table({"Foo"}, {{"Bar"}})},
                                {2, #table({"Foo"}, {{"Bar"}})}
                            }
                        )
                    },
                    {
                        "Feb",
                        Table.FromPartitions(
                            "Day",
                            {
                                {3, #table({"Foo"}, {{"Bar"}})},
                                {4, #table({"Foo"}, {{"Bar"}})}
                            }
                        )
                    }
                }
            )
        }
    }
)
```

FOO	[	[	[[
条形图	1	1 月	1994

条形图	2	1 月	1994
条形图	3	2 月	1994
条形图	4	2 月	1994

Table.FromRecords

2020/4/30 •

语法

```
Table.FromRecords(records as list, optional columns as any, optional missingField as nullable
number) as table
```

关于

将记录列表 `records` 转换为表。

示例 1

使用记录字段名称作为列名称，通过记录创建表。

```
Table.FromRecords({
    [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
    [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
    [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
})
```

CUSTOMERID	"	"
1	Bob	123-4567
2	Jim	987-6543
3	Paul	543-7890

示例 2

使用键入的列通过记录创建表并选择数字列。

```
Table.ColumnsOfType(
    Table.FromRecords(
        {[CustomerID = 1, Name = "Bob"]},
        type table[CustomerID = Number.Type, Name = Text.Type]
    ),
    {type number}
)
```

CustomerID

Table.FromRows

2020/4/30 •

语法

```
Table.FromRows(rows as list, optional columns as any) as table
```

关于

从列表 `rows` 创建一个表，其中列表的每个元素都是包含单行列值的内部列表。可为 `columns` 提供包含列名称、表类型或多列的可选列表。

示例 1

返回一个表，其中包含 [CustomerID] 列以及值 {1, 2}、[Name] 列以及值 {"Bob", "Jim"}、[Phone] 列以及值 {"123-4567", "987-6543"}。

```
Table.FromRows(
    {
        {1, "Bob", "123-4567"},
        {2, "Jim", "987-6543"}
    },
    {"CustomerID", "Name", "Phone"}
```

CUSTOMERID	"	"
1	Bob	123-4567
2	Jim	987-6543

示例 2

返回一个表，其中包含 [CustomerID] 列以及值 {1, 2}、[Name] 列以及值 {"Bob", "Jim"}、[Phone] 列以及值 {"123-4567", "987-6543"}。其中，[CustomerID] 是数值类型，[Name] 和 [Phone] 是文本类型。

```
Table.FromRows(
    {
        {1, "Bob", "123-4567"},
        {2, "Jim", "987-6543"}
    },
    type table [CustomerID = number, Name = text, Phone = text]
)
```

CUSTOMERID	"	"
1	Bob	123-4567
2	Jim	987-6543

Table.FromValue

2019/12/3 •

语法

```
Table.FromValue(value as any, optional options as nullable record) as table
```

关于

创建一个表，该表中的列包含所提供的值或值列表 `value`。可以指定可选记录参数 `options` 来控制以下选项：

- `DefaultColumnName`：从列表或标量值构造表时使用的列名称。

示例 1

从值 1 创建一个表。

```
Table.FromValue(1)
```

1

示例 2

从列表创建一个表。

```
Table.FromValue({1, "Bob", "123-4567"})
```

1
Bob
123-4567

示例 3

使用自定义列名称从值 1 创建一个表。

```
Table.FromValue(1, [DefaultColumnName = "MyValue"])
```

MYVALUE
1

Table.FuzzyGroup

2020/4/30 •

语法

```
Table.FuzzyGroup(table as table, key as any, aggregatedColumns as list, optional options as nullable record) as table
```

关于

按每行的指定列 `key` 中模糊匹配的值对 `table` 的行进行分组。对于每个组，构造一个记录，其中包含键列（及其值）以及 `aggregatedColumns` 指定的任何聚合列。此函数无法保证返回顺序固定的行。

可能包含一组可选的 `options` 以指定如何比较键列。选项包括：

- `Culture`
- `IgnoreCase`
- `IgnoreSpace`
- `Threshold`
- `TransformationTable`

下表提供了有关高级选项的更多详细信息。

区域性	非特定区域性	有效区域性名称	区域性选项允许根据特定于区域性的规则匹配记录。例如，区域性选项“ja-JP”基于日语来匹配记录。
IgnoreCase	True	true 或 false	IgnoreCase 选项允许在不考虑文本大小写的情况下将键分组。例如，如果 IgnoreCase 选项设置为 true，则“Grapes”（首字母大写）将与“grapes”（小写）分为一组。
IgnoreSpace	True	true 或 false	IgnoreSpace 选项允许组合文本部分以便查找匹配项。例如，如果 IgnoreSpace 选项设置为 true，则“Micro soft”将与“Microsoft”分为一组。

[illegible]	[illegible]	[illegible]	[illegible]
阈值	0.80	介于 0.00 到 1.00 之间	通过相似性阈值选项，可以匹配超过给定相似性分数的记录。阈值 1.00 即指定完全匹配标准。 例如，仅当阈值设置为小于 0.90 时，“Grapes”才与“Graes”（缺少“p”）分为一组。
TransformationTable		有效的表，其中至少有 2 个名为“From”和“To”的列。	TransformationTable 选项允许根据自定义值映射匹配记录。 例如，如果转换表的“From”列包含“Grapes”，而“To”列包含“Raisins”，则“Grapes”与“Raisins”匹配。 请注意，转换表中重复出现的所有文本都将应用转换。例如，在上面的转换表中，“Grapes are sweet”也将与“Raisins are sweet”匹配。

示例

对表进行分组，同时添加聚合列 [Count]，其中包含每个位置的员工数（`each Table.RowCount(_)`）。

```
Table.FuzzyGroup(
    Table.FromRecords(
        {
            [EmployeeID = 1, Location = "Seattle"],
            [EmployeeID = 2, Location = "seattl"],
            [EmployeeID = 3, Location = "Vancouver"],
            [EmployeeID = 4, Location = "Seatle"],
            [EmployeeID = 5, Location = "vancover"],
            [EmployeeID = 6, Location = "Seattle"],
            [EmployeeID = 7, Location = "Vancouver"]
        },
        type table [EmployeeID = nullable number, Location = nullable text]
    ),
    "Location",
    {"Count", each Table.RowCount(_)},
    [IgnoreCase = true, IgnoreSpace = true]
)
```

[illegible]	[illegible]
西雅图	4
温哥华	3

Table.FuzzyJoin

2020/4/30 •

语法

```
Table.FuzzyJoin(table1 as table, key1 as any, table2 as table, key2 as any, optional joinKind as nullable number, optional joinOptions as nullable record) as table
```

关于

基于 `key1`(对于 `table1`)和 `key2`(对于 `table2`)所选择的键列的值的模糊匹配，联接 `table1` 的行与 `table2` 的行。

模糊匹配是基于文本相似性而不是文本相等性的比较。

默认情况下，执行内部联接，但可以包含可选的 `joinKind` 以指定联接类型。选项包括：

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`

可能包含一组可选的 `joinOptions` 以指定如何比较键列。选项包括：

- `ConcurrentRequests`
- `Culture`
- `IgnoreCase`
- `IgnoreSpace`
- `NumberOfMatches`
- `Threshold`
- `TransformationTable`

下表提供了有关高级选项的更多详细信息。

ConcurrentRequests	1	介于 1 和 8 之间	ConcurrentRequests 选项支持通过指定要使用的并行线程数来并行联接操作。
区域性	非特定区域性	有效区域性名称	区域性选项允许根据特定于区域性的规则匹配记录。例如，区域性选项“ja-JP”基于日语来匹配记录。

IgnoreCase	True	true 或 false	IgnoreCase 选项允许在不考虑文本大小写的情况下匹配记录。 例如，如果 IgnoreCase 选项设置为 true，则“Grapes”（首字母大写）将与“grapes”（小写）匹配。
IgnoreSpace	True	true 或 false	IgnoreSpace 选项允许组合文本部分以便查找匹配项。 例如，如果 IgnoreSpace 选项设置为 true，则“Micro soft”与“Microsoft”和“Micro soft”匹配。
NumberOfMatches	2147483647	介于 0 到 2147483647 之间	NumberOfMatches 选项指定可返回的匹配行的最大数。
阈值	0.80	介于 0.00 到 1.00 之间	通过相似性阈值选项，可以匹配超过给定相似性分数的记录。阈值 1.00 即指定完全匹配标准。 例如，仅当阈值设置为小于 0.90 时，“Grapes”才与“Graes”匹配（缺少“p”）。
TransformationTable		有效的表，其中至少有 2 个名为“From”和“To”的列。	TransformationTable 选项允许根据自定义值映射匹配记录。 例如，如果转换表的“From”列包含“Grapes”，而“To”列包含“Raisins”， 则“Grapes”与“Raisins”匹配。

示例

基于 [FirstName] 的两个表的左侧内部模糊联接

```
Table.FuzzyJoin(
      Table.FromRecords(
        {
                [CustomerID = 1, FirstName1 = "Bob", Phone = "555-1234"],
                [CustomerID = 2, FirstName1 = "Robert", Phone = "555-4567"]
          },
        type table [CustomerID = nullable number, FirstName1 = nullable text, Phone = nullable text]
    ),
      {"FirstName1"},
      Table.FromRecords(
        {
                [CustomerStateID = 1, FirstName2 = "Bob", State = "TX"],
                [CustomerStateID = 2, FirstName2 = "bOB", State = "CA"]
          },
        type table [CustomerStateID = nullable number, FirstName2 = nullable text, State = nullable text]
    ),
      {"FirstName2"},
      JoinKind.LeftOuter,
      [IgnoreCase = true, IgnoreSpace = false]
)
```

CUSTOMERID	FIRSTNAME1	ıı	CUSTOMERSTATEID	FIRSTNAME2	ıı/ııı/ııı
1	Bob	555-1234	1	Bob	TX
1	Bob	555-1234	2	bOB	CA
2	Robert	555-4567			

Table.FuzzyNestedJoin

2020/4/30 •

语法

```
Table.FuzzyNestedJoin(table1 as table, key1 as any, table2 as table, key2 as any, newColumnName as text, optional joinKind as nullable number, optional joinOptions as nullable record) as table
```

关于

基于 `key1`（对于 `table1`）和 `key2`（对于 `table2`）所选择的键列的值的模糊匹配，联接 `table1` 的行与 `table2` 的行。结果在名为 `newColumnName` 的新列中返回。

模糊匹配是基于文本相似性而不是文本相等性的比较。

可选的 `joinKind` 指定要执行的联接类型。默认情况下，如果未指定 `joinKind`，则执行左外部联接。选项包括：

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`

可能包含一组可选的 `joinOptions` 以指定如何比较键列。选项包括：

- `ConcurrentRequests`
- `Culture`
- `IgnoreCase`
- `IgnoreSpace`
- `NumberOfMatches`
- `Threshold`
- `TransformationTable`

下表提供了有关高级选项的更多详细信息。

ConcurrentRequests	1	介于 1 和 8 之间	ConcurrentRequests 选项支持通过指定要使用的并行线程数来并行联接操作。
区域性	非特定区域性	有效区域性名称	区域性选项允许根据特定于区域性的规则匹配记录。例如，区域性选项“ja-JP”基于日语来匹配记录。

IgnoreCase	True	true 或 false	IgnoreCase 选项允许在不考虑文本大小写的情况下匹配记录。 例如，如果 IgnoreCase 选项设置为 true，则“Grapes”（首字母大写）将与“grapes”（小写）匹配。
IgnoreSpace	True	true 或 false	IgnoreSpace 选项允许组合文本部分以便查找匹配项。 例如，如果 IgnoreSpace 选项设置为 true，则“Micro soft”将与“Microsoft”匹配。
NumberOfMatches	2147483647	介于 0 到 2147483647 之间	NumberOfMatches 选项指定可返回的匹配行的最大数。
阈值	0.80	介于 0.00 到 1.00 之间	通过相似性阈值选项，可以匹配超过给定相似性分数的记录。阈值 1.00 即指定完全匹配标准。 例如，仅当阈值设置为小于 0.90 时，“Grapes”才与“Graes”匹配（缺少“p”）。
TransformationTable		有效的表，其中至少有 2 个名为“From”和“To”的列。	TransformationTable 选项允许根据自定义值映射匹配记录。 例如，如果转换表的“From”列包含“Grapes”，而“To”列包含“Raisins”， 则“Grapes”与“Raisins”匹配。

示例

基于 [FirstName] 的两个表的左侧内部模糊联接

```
Table.FuzzyNestedJoin(
      Table.FromRecords(
        {
                [CustomerID = 1, FirstName1 = "Bob", Phone = "555-1234"],
                [CustomerID = 2, FirstName1 = "Robert", Phone = "555-4567"]
          },
        type table [CustomerID = nullable number, FirstName1 = nullable text, Phone = nullable text]
    ),
      {"FirstName1"},
      Table.FromRecords(
        {
                [CustomerStateID = 1, FirstName2 = "Bob", State = "TX"],
                [CustomerStateID = 2, FirstName2 = "bOB", State = "CA"]
          },
        type table [CustomerStateID = nullable number, FirstName2 = nullable text, State = nullable text]
    ),
      {"FirstName2"},
      "NestedTable",
      JoinKind.LeftOuter,
      [IgnoreCase = true, IgnoreSpace = false]
)
```

CUSTOMERID	FIRSTNAME1	⁗	NESTEDTABLE

1	Bob	555-1234	[表]
2	Robert	555-4567	[表]

Table.Group

2020/4/30 •

语法

```
Table.Group(table as table, key as any, aggregatedColumns as list, optional groupKind as nullable number, optional comparer as nullable function) as table
```

关于

按每行的指定列 `key` 中的值对 `table` 的行进行分组。对于每个组，构造一个记录，其中包含键列（及其值）以及 `aggregatedColumns` 指定的任何聚合列。请注意，如果多个键与比较器匹配，则可能返回不同的键。此函数无法保证返回顺序固定的行。此外，还可以指定 `groupKind` 和 `comparer`。

示例 1

对表进行分组，同时添加一个聚合列 [total]，其中包含价格总和 ("each List.Sum([price])")。

```
Table.Group(
    Table.FromRecords({
        [CustomerID = 1, price = 20],
        [CustomerID = 2, price = 10],
        [CustomerID = 2, price = 20],
        [CustomerID = 1, price = 10],
        [CustomerID = 3, price = 20],
        [CustomerID = 3, price = 5]
    }),
    "CustomerID",
    {"total", each List.Sum([price])}
)
```

CUSTOMERID	ıı
1	30
2	30
3	25

Table.HasColumns

2020/4/30 •

语法

```
Table.HasColumns(table as table, columns as any) as logical
```

关于

指示 `table` 是否包含指定的列 `columns`。如果表包含这些列，则返回 `true` 否则返回 `false`。

示例 1

确定表是否具有列 [Name]。

```
TTable.HasColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Name"
)
```

`true`

示例 2

查找表是否具有列 [Name] 和 [PhoneNumber]。

```
Table.HasColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    {"Name", "PhoneNumber"}
)
```

`false`

Table.InsertRows

2020/4/30 •

语法

```
Table.InsertRows(table as table, offset as number, rows as list) as table
```

关于

返回一个表，此表包含插入到 `table` 中给定位置 `offset` 处的行 `rows` 的列表。行中要插入的每列都与表的列类型匹配。

示例 1

将行插入表中的位置 1 处。

```
Table.InsertRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    }),
    1,
    {[CustomerID = 3, Name = "Paul", Phone = "543-7890"]}
)
```

CUSTOMERID	ıı	ıı
1	Bob	123-4567
3	Paul	543-7890
2	Jim	987-6543

示例 2

将两行插入表中的位置 1 处。

```
Table.InsertRows(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    1,
    {
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }
)
```

CUSTOMERID	ıı	ıı
1	Bob	123-4567

2	Jim	987-6543
3	Paul	543-7890

Table.IsDistinct

2020/4/30 •

语法

```
Table.IsDistinct(table as table, optional comparisonCriteria as any) as logical
```

关于

指示 `table` 是否仅包含非重复行(没有重复项)。如果行是唯一的，则返回 `true`;否则返回 `false`。可选参数 `comparisonCriteria` 指定对表中的哪些列进行测试以确定是否有重复项。如果未指定 `comparisonCriteria`，则测试所有列。

示例 1

确定表是否为非重复表。

```
Table.IsDistinct(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)
```

`true`

示例 2

确定表中的列是否非重复。

```
Table.IsDistinct(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 5, Name = "Bob", Phone = "232-1550"]
    }),
    "Name"
)
```

`false`

Table.IsEmpty

2020/4/30 •

语法

```
Table.IsEmpty(table as table) as logical
```

关于

指示 `table` 是否包含任何行。如果没有行(即表为空)，则返回 `true`，否则返回 `false`。

示例 1

确定表是否为空。

```
Table.IsEmpty(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

`false`

示例 2

确定表 `({})` 是否为空。

```
Table.IsEmpty(Table.FromRecords({}))
```

`true`

Table.Join

2020/4/30 •

语法

```
Table.Join(table1 as table, key1 as any, table2 as table, key2 as any, optional joinKind as nullable number, optional joinAlgorithm as nullable number, optional keyEqualityComparers as nullable list) as table
```

关于

基于 `key1`（对于 `table1`）和 `key2`（对于 `table2`）所选择的键列的值的相等性，联接 `table1` 的行与 `table2` 的行。

默认情况下，执行内部联接，但可以包含可选的 `joinKind` 以指定联接类型。选项包括：

- `JoinKind.Inner`
- `JoinKind.LeftOuter`
- `JoinKind.RightOuter`
- `JoinKind.FullOuter`
- `JoinKind.LeftAnti`
- `JoinKind.RightAnti`

可能包含一组可选的 `keyEqualityComparers` 以指定如何比较键列。

示例 1

根据 [CustomerID] 对两个表执行内部联接

```
Table.Join(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "CustomerID",
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25]
    }),
    "CustomerID"
)
```

CUSTOMERID	[illegible]	[illegible]	ORDERID	[illegible]	[illegible]
1	Bob	123-4567	1	钓鱼竿	100

1	Bob	123-4567	2	1 磅蠕虫	5
2	Jim	987-6543	3	渔网	25
3	Paul	543-7890	4	电捕鱼器	200
3	Paul	543-7890	5	创可贴	2
1	Bob	123-4567	6	钓具盒	20

Table.Keys

2019/12/4 •

语法

```
Table.Keys(table as table) as list
```

关于

Table.Keys

Table.Last

2020/4/30 •

语法

```
Table.Last(table as table, optional default as any) as any
```

关于

返回 `table` 的最后一行，或如果表为空，则返回可选默认值 `default`。

示例 1

查找表的最后一行。

```
Table.Last(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

CUSTOMERID	3
II	Paul
II	543-7890

示例 2

查找表 `({})` 的最后一行，或如果为空，则返回 [a = 0, b = 0]。

```
Table.Last(Table.FromRecords({}), [a = 0, b = 0])
```

II	0
B	0

Table.LastN

2020/4/30 •

语法

```
Table.LastN(table as table, countOrCondition as any) as table
```

关于

根据 `countOrCondition` 的值，返回 `table` 表的最后一行(或几行)：

- 如果 `countOrCondition` 是数字，则将返回从末尾 - `countOrCondition` 位置开始的多行。
- 如果 `countOrCondition` 是条件，则将以升序位置返回满足此条件的行，直到行不满足条件为止。

示例 1

查找表的最后两行。

```
Table.LastN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    }),
    2
)
```

CUSTOMERID	ıı	ıı
2	Jim	987-6543
3	Paul	543-7890

示例 2

查找表中 [a] > 0 的最后几行。

```
Table.LastN(
    Table.FromRecords({
        [a = -1, b = -2],
        [a = 3, b = 4],
        [a = 5, b = 6]
    }),
    each _ [a] > 0
```

ıı	B
3	4

5	6

Table.MatchesAllRows

2020/4/30 •

语法

```
Table.MatchesAllRows(table as table, condition as function) as logical
```

关于

指示 `table` 中的所有行是否与给定的 `condition` 匹配。如果所有行均匹配，则返回 `true`；否则返回 `false`。

示例 1

确定列 [a] 中的所有行值是否在表中。

```
Table.MatchesAllRows(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8]
    }),
    each Number.Mod([a], 2) = 0
)
```

`true`

示例 2

查找在表 `({[a = 1, b = 2], [a = 3, b = 4]})` 中是否所有行值均为 [a = 1, b = 2]。

```
Table.MatchesAllRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = -3, b = 4]
    }),
    each _ = [a = 1, b = 2]
)
```

`false`

Table.MatchesAnyRows

2020/4/30 •

语法

```
Table.MatchesAnyRows(table as table, condition as function) as logical
```

关于

指示 `table` 中的任何行是否与给定的 `condition` 匹配。如果任何行均匹配，则返回 `true`；否则返回 `false`。

示例 1

确定列 [a] 中的任何行值是否在表 `({[a = 2, b = 4], [a = 6, b = 8]})` 中。

```
Table.MatchesAnyRows(
    Table.FromRecords({
        [a = 1, b = 4],
        [a = 3, b = 8]
    }),
    each Number.Mod([a], 2) = 0
)
```

`false`

示例 2

确定表 `({[a = 1, b = 2], [a = 3, b = 4]})` 中的任何行值是否均为 [a = 1, b = 2]。

```
Table.MatchesAnyRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = -3, b = 4]
    }),
    each _ = [a = 1, b = 2]
)
```

`true`

Table.Max

2020/4/30 •

语法

```
Table.Max(table as table, comparisonCriteria as any, optional default as any) as any
```

关于

给定 `comparisonCriteria`，返回 `table` 中最大的行。如果表为空，则返回可选的 `default` 值。

示例 1

查找表 `({[a = 2, b = 4], [a = 6, b = 8]})` 的列 [a] 中具有最大值的行。

```
Table.Max(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8]
    }),
    "a"
)
```

a	6
b	8

示例 2

查找表 `({})` 的列 [a] 中具有最大值的行。如果为空，则返回 -1。

```
Table.Max(#table({"a"}, {}), "a", -1)
```

`-1`

Table.MaxN

2020/4/30 •

语法

```
Table.MaxN(table as table, comparisonCriteria as any, countOrCondition as any) as table
```

关于

如果给定 `comparisonCriteria`，则返回 `table` 中最大的行。对行进行排序之后，必须指定 `countOrCondition` 参数以进一步筛选结果。注意，排序算法不能保证固定的排序结果。`countOrCondition` 参数可以采用多种形式：

- 如果指定了一个数字，则返回最多 `countOrCondition` 个项目的升序列表。
- 如果指定了条件，则返回最初满足条件的项目列表。一旦某个项目不符合条件，则不再考虑其他项目。

示例 1

在表中使用 [a] > 0 条件查找 [a] 列中具有最大值的行。先对行进行排序，然后再应用筛选器。

```
Table.MaxN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 0, b = 0],
        [a = 6, b = 2]
    }),
    "a",
    each [a] > 0
)
```

A	B
6	2
2	4

示例 2

在表中使用 [b] > 0 条件查找 [a] 列中具有最大值的行。先对行进行排序，然后再应用筛选器。

```
Table.MaxN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 8, b = 0],
        [a = 6, b = 2]
    }),
    "a",
    each [b] > 0
)
```

Table.Min

2020/4/30 •

语法

```
Table.Min(table as table, comparisonCriteria as any, optional default as any) as any
```

关于

如果给定 `comparisonCriteria`，则返回 `table` 中最小的行。如果表为空，则返回可选的 `default` 值。

示例 1

查找表的列 [a] 中具有最小值的行。

```
Table.Min(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8]
    }),
    "a"
)
```

a	2
b	4

示例 2

查找表的列 [a] 中具有最小值的行。如果为空，则返回 -1。

```
Table.Min(#table({"a"}, {}), "a", -1)
```

`-1`

Table.MinN

2020/4/30 •

语法

```
Table.MinN(table as table, comparisonCriteria as any, countOrCondition as any) as table
```

关于

如果给定 `comparisonCriteria`，则返回 `table` 中最小的行。对行进行排序之后，必须指定 `countOrCondition` 参数以进一步筛选结果。注意，排序算法不能保证固定的排序结果。`countOrCondition` 参数可以采用多种形式：

- 如果指定了一个数字，则返回最多 `countOrCondition` 个项目的升序列表。
- 如果指定了条件，则返回最初满足条件的项目列表。一旦某个项目不符合条件，则不再考虑其他项目。

示例 1

在表中使用 [a] < 3 条件查找 [a] 列中具有最小值的行。先对行进行排序，然后再应用筛选器。

```
Table.MinN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 0, b = 0],
        [a = 6, b = 4]
    }),
    "a",
    each [a] < 3
)
```

A	B
0	0
2	4

示例 2

在表中使用 [b] < 0 条件查找 [a] 列中具有最小值的行。先对行进行排序，然后再应用筛选器。

```
Table.MinN(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 8, b = 0],
        [a = 6, b = 2]
    }),
    "a",
    each [b] < 0
)
```

Table.NestedJoin

2019/12/4 •

语法

```
Table.NestedJoin(table1 as table, key1 as any, table2 as any, key2 as any, newColumnName as text,
optional joinKind as nullable number, optional keyEqualityComparers as nullable list) as table
```

关于

基于 `key1`（对于 `table1`）和 `key2`（对于 `table2`）所选择的键列的值的相等性，联接 `table1` 的行与 `table2` 的行。结果会输入到名为 `newColumnName` 的列中。

可选的 `joinKind` 指定要执行的联接类型。默认情况下，如果未指定 `joinKind`，则执行左外部联接。

可能包含一组可选的 `keyEqualityComparers` 以指定如何比较键列。

Table.Partition

2020/4/30 •

语法

```
Table.Partition(table as table, column as text, groups as number, hash as function) as list
```

关于

根据 `column` 和 `hash` 函数的值，将 `table` 分区为数量为 `groups` 的一系列表。将 `hash` 函数应用于 `column` 行的值以获得该行的哈希值。哈希值的取模 `groups` 决定要在其中放置行的返回表。

- `table`：要分区的表。
- `column`：要进行哈希处理的列，用于确定行所在的返回表。
- `groups`：将输入表进行分区而得到的表的数量。
- `hash`：用于获取哈希值的函数。

示例

使用列的值作为哈希函数，将表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` 分区为列 [a] 上的 2 个表。

```
Table.Partition(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    "a",
    2,
    each _
)
```

```
{
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 2, b = 4]
    }),
    Table.FromRecords({
        [a = 1, b = 4],
        [a = 1, b = 4]
    })
}
```

Table.PartitionValues

2019/12/3 •

语法

```
Table.Partition(table as table, column as text, groups as number, hash as function) as list
```

关于

根据 `column` 和 `hash` 函数的值，将 `table` 分区为数量为 `groups` 的一系列表。将 `hash` 函数应用于 `column` 行的值以获得该行的哈希值。哈希值的取模 `groups` 决定要在其中放置行的返回表。

- `table`：要分区的表。
- `column`：要进行哈希处理的列，用于确定行所在的返回表。
- `groups`：将输入表进行分区而得到的表的数量。
- `hash`：用于获取哈希值的函数。

示例 1

使用列的值作为哈希函数，将表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` 分区为列 [a] 上的 2 个表。

```
Table.Partition(Table.FromRecords({[a = 2, b = 4], [a = 1, b = 4], [a = 2, b = 4], [a = 1, b = 4]}), "a", 2,
each _)
```

[表]
[表]

Table.Pivot

2020/4/30 •

语法

```
Table.Pivot(table as table, pivotValues as list, attributeColumn as text, valueColumn as text,
optional aggregationFunction as nullable function) as table
```

关于

在给定一对表示属性-值对的列的情况下，将属性列中的数据旋转为列标题。

示例 1

采用

```
({ [ key = "x", attribute = "a", value = 1 ], [ key = "x", attribute = "c", value = 3 ], [ key = "y", attribute
= "a", value = 2 ], [ key = "y", attribute = "b", value = 4 ] })
```

表的属性列中的值 "a"、"b" 和 "c"，并将它们透视到它们自己的列中。

```
Table.Pivot(
    Table.FromRecords({
        [key = "x", attribute = "a", value = 1],
        [key = "x", attribute = "c", value = 3],
        [key = "y", attribute = "a", value = 2],
        [key = "y", attribute = "b", value = 4]
    }),
    {"a", "b", "c"},
    "attribute",
    "value"
)
```

[illegible]	[illegible]	B	C
x	1		3
y	2	4	

示例 2

采用

```
({ [ key = "x", attribute = "a", value = 1 ], [ key = "x", attribute = "c", value = 3 ], [ key = "x", attribute
= "c", value = 5 ], [ key = "y", attribute = "a", value = 2 ], [ key = "y", attribute = "b", value = 4 ] })
```

表的属性列中的值 "a"、"b" 和 "c"，并将它们透视到它们自己的列中。键 "x" 的属性 "c" 具有多个与之关联的值，因此请使用 List.Max 函数来解决冲突。

```
Table.Pivot(
    Table.FromRecords({
        [key = "x", attribute = "a", value = 1],
        [key = "x", attribute = "c", value = 3],
        [key = "x", attribute = "c", value = 5],
        [key = "y", attribute = "a", value = 2],
        [key = "y", attribute = "b", value = 4]
    }),
    {"a", "b", "c"},
    "attribute",
    "value",
    List.Max
)
```

[illegible]	[illegible]	B	C
x	1		5
y	2	4	

Table.PositionOf

2020/4/30 •

语法

```
Table.PositionOf(table as table, row as record, optional occurrence as any, optional
equationCriteria as any) as any
```

关于

返回 `row` 在指定 `table` 中第一个实例的位置。如果未找到匹配项，则返回 -1。

- `table` : 输入表。
- `row` : 表中要查找其位置的行。
- `occurrence` : *[可选]* 指定要返回的行的匹配项。
- `equationCriteria` : *[可选]* 控制表行之间的比较。

示例 1

查找 [a = 2, b = 4] 在表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` 中第一个实例的位置。

```
Table.PositionOf(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    [a = 2, b = 4]
)
```

`0`

示例 2

查找 [a = 2, b = 4] 在表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` 中第二个实例的位置。

```
Table.PositionOf(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    [a = 2, b = 4],
    1
)
```

`2`

示例 3

查找 [a = 2, b = 4] 在表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` 中所有实例的位置。

```
Table.PositionOf(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    [a = 2, b = 4],
    Occurrence.All
)
```

0

2

Table.PositionOfAny

2020/4/30 •

语法

```
Table.PositionOfAny(table as table, rows as list, optional occurrence as nullable number, optional equationCriteria as any) as any
```

关于

返回 `rows` 列表第一个匹配项在 `table` 中的行位置。如果未找到匹配项，则返回 -1。

- `table`：输入表。
- `rows`：表中要查找其位置的行的列表。
- `occurrence`：*[可选]* 指定要返回的行的匹配项。
- `equationCriteria`：*[可选]* 控制表行之间的比较。

示例 1

查找 [a = 2, b = 4] 或 [a = 6, b = 8] 在表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]})` 中第一次出现的位置。

```
Table.PositionOfAny(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 1, b = 4],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    {
        [a = 2, b = 4],
        [a = 6, b = 8]
    }
)
```

`0`

示例 2

查找 [a = 2, b = 4] 或 [a = 6, b = 8] 在表 `({[a = 2, b = 4], [a = 6, b = 8], [a = 2, b = 4], [a = 1, b = 4]}` 中所有出现的位置。

```
Table.PositionOfAny(
    Table.FromRecords({
        [a = 2, b = 4],
        [a = 6, b = 8],
        [a = 2, b = 4],
        [a = 1, b = 4]
    }),
    {
        [a = 2, b = 4],
        [a = 6, b = 8]
    },
    Occurrence.All
)
```

0
1
2

Table.PrefixColumns

2019/12/4 •

语法

```
Table.PrefixColumns(table as table, prefix as text) as table
```

关于

返回一个表，其中来自所提供的 `table` 中的所有列名称均以给定的文本 `prefix` 为前缀，并加入一个句点，形成 `prefix` `.ColumnName` 格式。

示例 1

为表中的列添加“MyTable”前缀。

```
Table.PrefixColumns(Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}), "MyTable")
```

MYTABLE.CUSTOMERID	MYTABLE.NAME	MYTABLE.PHONE
1	Bob	123-4567

Table.Profile

2019/12/4 •

语法

```
Table.Profile(table as table, optional additionalAggregates as nullable list) as table
```

关于

返回列的配置文件 `table`。

会为每列返回以下信息（如果适用）：

- 最小值
- 最大值
- 平均值
- 标准偏差
- 计数
- NULL 计数
- 非重复计数

Table.PromoteHeaders

2020/4/30 •

语法

```
Table.PromoteHeaders(table as table, optional options as nullable record) as table
```

关于

将第一行值升级为新的列标题(即列名)。默认仅文本或数字值会升级为标题。有效选项:

`PromoteAllScalars`:如果设置为 `true`,则会使用 `Culture`(如果指定有,否则会使用当前文档区域设置)将第一行中的所有标量值升级为标题。对于不能转换为文本的值,将使用默认列名。
`Culture`:指定数据区域性的区域性名称。

示例 1

升级表中的第一行值。

```
Table.PromoteHeaders(
    Table.FromRecords({
        [Column1 = "CustomerID", Column2 = "Name", Column3 = #date(1980, 1, 1)],
        [Column1 = 1, Column2 = "Bob", Column3 = #date(1980, 1, 1)]
    })
)
```

CUSTOMERID	ıc	COLUMN3
1	Bob	1/1/1980 12:00:00 AM

示例 2

将表的第一行中的所有标量升级为标题。

```
Table.PromoteHeaders(
    Table.FromRecords({
        [Rank = 1, Name = "Name", Date = #date(1980, 1, 1)],
        [Rank = 1, Name = "Bob", Date = #date(1980, 1, 1)]}
    ),
    [PromoteAllScalars = true, Culture = "en-US"]
)
```

1	ıc	1980/1/1
1	Bob	1/1/1980 12:00:00 AM

Table.Range

2020/4/30 •

语法

```
Table.Range(table as table, offset as number, optional count as nullable number) as table
```

关于

从以指定 `offset` 开始的 `table` 返回行。可选参数 `count` 指定要返回的行数。默认情况下，返回偏移量后的所有行。

示例 1

返回表中以偏移量 1 开始的所有行。

```
Table.Range(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	"	"
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

示例 2

返回表中以偏移量 1 开始的一行。

```
Table.Range(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1,
    1
)
```

CUSTOMERID	"	"

2	Jim	987-6543

Table.RemoveColumns

2020/4/30 •

语法

```
Table.RemoveColumns(table as table, columns as any, optional missingField as nullable number) as table
```

关于

从提供的 `table` 删除指定的 `columns`。如果列不存在，则会引发异常，除非可选参数 `missingField` 指定替换选项（例如， `MissingField.UseNull` 或 `MissingField.Ignore`）。

示例 1

从表中删除列 [Phone]。

```
Table.RemoveColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "Phone"
)
```

CUSTOMERID	II
1	Bob

示例 2

从表中删除列 [Address]。如果它不存在，将引发错误。

```
Table.RemoveColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "Address"
)
```

```
[Expression.Error] The field 'Address' of the record was not found.
```

Table.RemoveFirstN

2020/4/30 •

语法

```
Table.RemoveFirstN(table as table, optional countOrCondition as any) as table
```

关于

返回一个表，该表不包含表 `table` 中前数行 `countOrCondition`（行数为指定数字）。删除的行数取决于可选参数 `countOrCondition`。

- 如果省略 `countOrCondition` 则只删除第一行。
- 如果 `countOrCondition` 是数字，则将删除多个行（从顶部开始）。
- 如果 `countOrCondition` 是条件，则将删除满足条件的行，直到出现不满足条件的行为止。

示例 1

删除表的第一行。

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	"	"
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

示例 2

删除表的前两行。

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    2
)
```

CUSTOMERID	ıı	ıı
3	Paul	543-7890
4	Ringo	232-1550

示例 3

删除表中 [CustomerID] <=2 的前几行。

```
Table.RemoveFirstN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"] ,
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"] ,
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each [CustomerID] <= 2
)
```

CUSTOMERID	ıı	ıı
3	Paul	543-7890
4	Ringo	232-1550

Table.RemoveLastN

2020/4/30 •

语法

```
Table.RemoveLastN(table as table, optional countOrCondition as any) as table
```

关于

返回一个表，该表不包含表 `table` 中最后的 `countOrCondition` 行。删除的行数取决于可选参数 `countOrCondition` 。

- 如果省略 `countOrCondition`，则只删除最后一行。
- 如果 `countOrCondition` 是数字，则将删除多个行（从底部开始）。
- 如果 `countOrCondition` 是条件，则将删除满足条件的行，直到出现不满足条件的行为止。

示例 1

删除表的最后一行。

```
Table.RemoveLastN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	"	"
1	Bob	123-4567
2	Jim	987-6543
3	Paul	543-7890

示例 2

删除表中 [CustomerID] > 2 的最后几行。

```
Table.RemoveLastN(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each [CustomerID] >= 2
)
```

CUSTOMERID	"	"
1	Bob	123-4567

Table.RemoveMatchingRows

2020/4/30 •

语法

```
Table.RemoveMatchingRows(table as table, rows as list, optional equationCriteria as any) as table
```

关于

从 `table` 中删除所有出现的指定 `rows`。可以指定一个可选参数 `equationCriteria` 以控制表中各行之间的比较。

示例 1

从表 `({[a = 1, b = 2], [a = 3, b = 4], [a = 1, b = 6]})` 中删除 [a = 1] 的任何行。

```
Table.RemoveMatchingRows(
    Table.FromRecords({
        [a = 1, b = 2],
        [a = 3, b = 4],
        [a = 1, b = 6]
    }),
    {[a = 1]},
    "a"
)
```

A	B
3	4

Table.RemoveRows

2020/4/30 •

语法

```
Table.RemoveRows(table as table, offset as number, optional count as nullable number) as table
```

关于

从指定的 `offset` 开始，从 `table` 的开头删除 `count` 行。如果未提供 `count` 参数，则使用默认计数 1。

示例 1

从表中删除第一行。

```
Table.RemoveRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    0
)
```

CUSTOMERID	"	"
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

示例 2

删除表中位置 1 的行。

```
Table.RemoveRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	"	"
1	Bob	123-4567

3	Paul	543-7890
4	Ringo	232-1550

示例 3

从位置 1 开始，删除表中的两行。

```
Table.RemoveRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1,
    2
)
```

CUSTOMERID	ıı	ıı
1	Bob	123-4567
4	Ringo	232-1550

Table.RemoveRowsWithErrors

2020/4/30 •

语法

```
Table.RemoveRowsWithErrors(table as table, optional columns as nullable list) as table
```

关于

返回一个表，其中删除了在至少一个单元格中包含错误的输入表中的行。如果指定了列列表，则仅检查指定列中的单元格是否存在错误。

示例 1

从第一行中删除错误值。

```
Table.RemoveRowsWithErrors(
    Table.FromRecords({
        [Column1 = ...],
        [Column1 = 2],
        [Column1 = 3]
    })
)
```

Column1
2
3

Table.RenameColumns

2020/4/30 •

语法

```
Table.RenameColumns(table as table, renames as list, optional missingField as nullable number) as table
```

关于

对表 `table` 中的列执行给定的重命名。一个替换操作 `renames` 由两个值（旧列名和新列名）组成，以列表的形式提供。如果列不存在，则会引发异常，除非可选参数 `missingField` 指定替换选项（例如， `MissingField.UseNull` 或 `MissingField.Ignore`）。

示例 1

在表中将列名“CustomerNum”替换为“CustomerID”。

```
Table.RenameColumns(
    Table.FromRecords({[CustomerNum = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerNum", "CustomerID"}
)
```

CUSTOMERID	ABC	ABC
1	Bob	123-4567

示例 2

在表中将列名“CustomerNum”替换为“CustomerID”，并将“PhoneNum”替换为“Phone”。

```
Table.RenameColumns(
    Table.FromRecords({[CustomerNum = 1, Name = "Bob", PhoneNum = "123-4567"]}),
    {
        {"CustomerNum", "CustomerID"},
        {"PhoneNum", "Phone"}
    }
)
```

CUSTOMERID	ABC	ABC
1	Bob	123-4567

示例 3

在表中将列名“NewCol”替换为“NewColumn”；如果列不存在，则忽略。

```
Table.RenameColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"NewCol", "NewColumn"},
    MissingField.Ignore
)
```

CUSTOMERID	"	"
1	Bob	123-4567

Table.ReorderColumns

2020/4/30 •

语法

```
Table.ReorderColumns(table as table, columnOrder as list, optional missingField as nullable
number) as table
```

关于

返回输入 `table` 中的表，其中，列是按 `columnOrder` 指定的顺序排列的。不会对列表中未指定的列进行重新排序。如果列不存在，则会引发异常，除非可选参数 `missingField` 指定替换选项（例如， `MissingField.UseNull` 或 `MissingField.Ignore`）。

示例 1

切换表中列 [Phone] 和 [Name] 的顺序。

```
Table.ReorderColumns(
    Table.FromRecords({[CustomerID = 1, Phone = "123-4567", Name = "Bob"]}),
    {"Name", "Phone"}
)
```

CUSTOMERID	ıı	ıı
1	Bob	123-4567

示例 2

在表中切换列 [Phone] 和 [Address] 的顺序，或使用“MissingField.Ignore”。它不会更改表，因为列 [Address] 不存在。

```
Table.ReorderColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"Phone", "Address"},
    MissingField.Ignore
)
```

CUSTOMERID	ıı	ıı
1	Bob	123-4567

Table.Repeat

2020/4/30 •

语法

```
Table.Repeat(table as table, count as number) as table
```

关于

返回一个表，表中的行来自输入 `table` 且重复了指定的 `count` 次。

示例 1

重复表中的行两次。

```
Table.Repeat(
    Table.FromRecords({
        [a = 1, b = "hello"],
        [a = 3, b = "world"]
    }),
    2
)
```

A	B
1	hello
3	world
1	hello
3	world

Table.ReplaceErrorValues

2020/4/30 •

语法

```
Table.ReplaceErrorValues(table as table, errorReplacement as list) as table
```

关于

将 `table` 的指定列中的错误值替换为 `errorReplacement` 列表中的新值。列表的格式为 {{column1, value1}, ...}。每列只能有一个替换值，多次指定列将导致错误。

示例 1

在表中将错误值替换为文本“world”。

```
Table.ReplaceErrorValues(
    Table.FromRows({{1, "hello"}, {3, ...}}, {"A", "B"}),
    {"B", "world"}
)
```

A	B
1	hello
3	world

示例 2

在表中将 A 列中的错误值替换为文本“hello”，将 B 列中的错误值替换为文本“world”。

```
Table.ReplaceErrorValues(
    Table.FromRows({{..., ...}, {1, 2}}, {"A", "B"}),
    {{"A", "hello"}, {"B", "world"}}
)
```

A	B
hello	world
1	2

Table.ReplaceKeys

2019/12/3 •

语法

```
Table.ReplaceKeys(table as table, keys as list) as table
```

关于

Table.ReplaceKeys

Table.ReplaceMatchingRows

2020/4/30 •

语法

```
Table.ReplaceMatchingRows(table as table, replacements as list, optional equationCriteria as any)
as table
```

关于

使用提供的行替换 `table` 中的所有指定行。使用 {old, new} 格式在 `replacements` 中指定要替换的行和替换项。可以指定可选参数 `equationCriteria` 以控制表各行之间的比较。

示例 1

将表中的 [a = 1, b = 2] 和 [a = 2, b = 3] 行替换为 [a = -1, b = -2] 和 [a = -2, b = -3]。

```
Table.ReplaceMatchingRows(
    Table.FromRecords({
        [a = 1, b =2],
        [a = 2, b = 3],
        [a = 3, b = 4],
        [a = 1, b = 2]
    }),
    {
        {[a = 1, b = 2], [a = -1, b = -2]},
        {[a = 2, b = 3], [a = -2, b = -3]}
    }
)
```

A	B
-1	-2
-2	-3
3	4
-1	-2

Table.ReplaceRelationshipIdentity

2019/12/3 •

语法

```
Table.ReplaceRelationshipIdentity(value as any, identity as text) as any
```

关于

Table.ReplaceRelationshipIdentity

Table.ReplaceRows

2020/4/30 •

语法

```
Table.ReplaceRows(table as table, offset as number, count as number, rows as list) as table
```

关于

在输入 `table` 中，用指定的 `rows` 替换指定数目的行 `count`，并在 `offset` 后开始。`rows` 参数是记录列表。

- `table`：要在其中执行替换的表。
- `offset`：进行替换之前要跳过的行数。
- `count`：要替换的行数。
- `rows`：要插入到 `offset` 指定的位置的 `table` 中的行记录列表。

示例 1

从位置 1 开始，替换 3 行。

```
Table.ReplaceRows(
    Table.FromRecords({
        [Column1 = 1],
        [Column1 = 2],
        [Column1 = 3],
        [Column1 = 4],
        [Column1 = 5]
    }),
    1,
    3,
    {[Column1 = 6], [Column1 = 7]}
)
```

[1
1
6
7
5

Table.ReplaceValue

2020/4/30 •

语法

```
Table.ReplaceValue(table as table, oldValue as any, newValue as any, replacer as function,
columnsToSearch as list) as table
```

关于

在 `table` 的指定列中将 `oldValue` 替换为 `newValue`。

示例 1

在表中将文本"goodbye"替换为文本"world"。

```
Table.ReplaceValue(
    Table.FromRecords({
        [a = 1, b = "hello"],
        [a = 3, b = "goodbye"]
    }),
    "goodbye",
    "world",
    Replacer.ReplaceText,
    {"b"}
)
```

A	B
1	hello
3	world

示例 2

在表中将文本"ur"替换为文本"or"。

```
Table.ReplaceValue(
    Table.FromRecords({
        [a = 1, b = "hello"],
        [a = 3, b = "wurld"]
    }),
    "ur",
    "or",
    Replacer.ReplaceText,
    {"b"}
)
```

A	B
1	hello

3	world

Table.Reverse

2019/12/3 •

语法

```
Text.Reverse(text as nullable text) as nullable text
```

关于

反写所提供的 `text` 。

示例 1

反写文本“123”。

```
Text.Reverse("123")
```

`"321"`

Table.ReverseRows

2020/4/30 •

语法

```
Table.ReverseRows(table as table) as table
```

关于

从输入 `table` 返回一个表，其中的行按相反的顺序排列。

示例 1

使表中的行按相反顺序排列。

```
Table.ReverseRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)
```

CUSTOMERID	"	"
4	Ringo	232-1550
3	Paul	543-7890
2	Jim	987-6543
1	Bob	123-4567

Table.RowCount

2020/4/30 •

语法

```
Table.RowCount(table as table) as number
```

关于

返回 `table` 中的行数。

示例 1

查找表中的行数。

```
Table.RowCount(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

`3`

Table.Schema

2019/12/4 •

语法

```
Table.Schema(table as table) as table
```

关于

返回描述列的表 `table`。

表中的每一行描述了列的属性 `table`：

■	■
`Name`	列名称。
`Position`	列中从 0 开始的位置 `table`。
`TypeName`	列的类型名称。
`Kind`	列的类型种类。
`IsNullable`	列是否可以包含 `null` 值。
`NumericPrecisionBase`	`NumericPrecision` 和 `NumericScale` 字段的数字基数（如以 2 为底、以 10 为底）。
`NumericPrecision`	`NumericPrecisionBase` 指定的基数中数字列的精准率。这是此类型（包括小数位数）的值可表示的数字的最大数。
`NumericScale`	`NumericPrecisionBase` 指定的基数中数字列的比例。这是此类型的值的小数部分的位数。`0` 值表示一固定比例，没有小数位数。`null` 值表示比例未知（因为它是浮动的或者未定义的）。
`DateTimePrecision`	日期或时间值的秒数部分支持的最大小数位数。
`MaxLength`	`text` 列中允许的最大字符数，或 `binary` 列中允许的最大字节数。
`IsVariableLength`	指示此列在长度上是否可以发生变化（最大长度为 `MaxLength`）或其大小是否固定。
`NativeTypeName`	源的本机类型系统中列的类型名称（例如 SQL Server 的 `nvarchar`）。

`NativeDefaultExpression`	源的本机表达式语言中此列值的默认表达式(例如 `42` 或 SQL Server 的 `newid()`)。
`Description`	列的说明。

Table.SelectColumns

2020/4/30 •

语法

```
Table.SelectColumns(table as table, columns as any, optional missingField as nullable number) as table
```

关于

返回仅具有指定的 `columns` 的 `table`。

- `table`：提供的表。
- `columns`：要返回的表 `table` 中列的列表。返回的表中的列按 `columns` 中列出的顺序排列。
- `missingField`：*(可选)*如果列不存在，应执行的操作。示例：`MissingField.UseNull` 或 `MissingField.Ignore`。

示例 1

只包含列 [Name]。

```
Table.SelectColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    "Name"
)
```

“
Bob
Jim
Paul
Ringo

示例 2

只包含列 [CustomerID] 和列 [Name]。

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerID", "Name"}
)
```

CUSTOMERID	1
1	Bob

示例 3

如果包含的列没有退出，则默认结果是一个错误。

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    "NewColumn"
)
```

```
[Expression.Error] The field 'NewColumn' of the record wasn't found.
```

示例 4

如果包含的列没有退出，选项 `MissingField.UseNull` 将创建包含 null 值的列。

```
Table.SelectColumns(
    Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}),
    {"CustomerID", "NewColumn"},
    MissingField.UseNull
)
```

CUSTOMERID	NEWCOLUMN
1	

Table.SelectRows

2020/4/30 •

语法

```
Table.SelectRows(table as table, condition as function) as table
```

关于

从 `table` 返回与选择 `condition` 匹配的行的表。

示例 1

选择表中的行，其中 [CustomerID] 列中的值大于 2。

```
Table.SelectRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each [CustomerID] > 2
)
```

CUSTOMERID	ıı	ıı
3	Paul	543-7890
4	Ringo	232-1550

示例 2

选择表中的名称不包含"B"的行。

```
Table.SelectRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    each not Text.Contains([Name], "B")
)
```

CUSTOMERID	ıı	ıı
2	Jim	987-6543
3	Paul	543-7890

4	Ringo	232-1550

Table.SelectRowsWithErrors

2020/4/30 •

语法

```
Table.SelectRowsWithErrors(table as table, optional columns as nullable list) as table
```

关于

返回一个表，其中仅包含输入表中至少一个单元格中包含错误的那些行。如果指定了列列表，则仅检查指定列中的单元格是否存在错误。

示例 1

选择其行中有错误的客户名称。

```
Table.SelectRowsWithErrors(
    Table.FromRecords({
        [CustomerID = ..., Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    })
)[Name]
```

Bob

Table.SingleRow

2019/12/4 •

语法

```
Table.SingleRow(table as table) as record
```

关于

返回包含一行 `table` 中的单行。如果 `table` 包含多个行，会引发异常。

示例 1

返回表中的单行。

```
Table.SingleRow(Table.FromRecords({[CustomerID = 1, Name = "Bob", Phone = "123-4567"]}))
```

CUSTOMERID	1
"	Bob
"	123-4567

Table.Skip

2020/4/30 •

语法

```
Table.Skip(table as table, optional countOrCondition as any) as table
```

关于

返回一个表，该表不包含表 `table` 中前数行 `countOrCondition`（行数为指定数字）。跳过的行数取决于可选参数 `countOrCondition`。

- 如果省略 `countOrCondition` 则只跳过第一行。
- 如果 `countOrCondition` 是数字，则将跳过多个行（从顶部开始）。
- 如果 `countOrCondition` 是条件，则将跳过不满足条件的某一行之前的满足条件的行。

示例 1

跳过表中第一行。

```
Table.Skip(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    1
)
```

CUSTOMERID	"	"
2	Jim	987-6543
3	Paul	543-7890
4	Ringo	232-1550

示例 2

跳过表中前两行。

```
Table.Skip(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Ringo", Phone = "232-1550"]
    }),
    2
)
```

CUSTOMERID	[illegible]	[illegible]
3	Paul	543-7890
4	Ringo	232-1550

示例 3

跳过表中 [Price] > 25 的前几行。

```
Table.Skip(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    each [Price] > 25
)
```

ORDERID	CUSTOMERID	[illegible]	[illegible]
2	1	1 磅蠕虫	5
3	2	渔网	25
4	3	电捕鱼器	200
5	3	创可贴	2
6	1	钓具盒	20
7	5	鱼饵	3.25
8	5	钓鱼竿	100
9	6	鱼饵	3.25

Table.Sort

2020/4/30 •

语法

```
Table.Sort(table as table, comparisonCriteria as any) as table
```

关于

使用一个或多个列名的列表和可选的 `comparisonCriteria`（格式为 { { col1, comparisonCriteria }, {col2} }）对 `table` 排序。

示例 1

在列“OrderID”上对表排序。

```
Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {"OrderID"}
)
```

ORDERID	CUSTOMERID	ITEM	PRICE
1	1	钓鱼竿	100
2	1	1 磅蠕虫	5
3	2	渔网	25
4	3	电捕鱼器	200
5	3	创可贴	2
6	1	钓具盒	20
7	5	鱼饵	3.25
8	5	钓鱼竿	100
9	6	鱼饵	3.25

示例 2

在列“OrderID”上以降序顺序对表排序。

```
Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {"OrderID", Order.Descending}
)
```

ORDERID	CUSTOMERID	I	II
9	6	鱼饵	3.25
8	5	钓鱼竿	100
7	5	鱼饵	3.25
6	1	钓具盒	20
5	3	创可贴	2
4	3	电捕鱼器	200
3	2	渔网	25
2	1	1 磅蠕虫	5
1	1	钓鱼竿	100

示例 3

依次在列“CustomerID”、列“OrderID”上对表排序，其中列“CustomerID”以升序顺序排序。

```
Table.Sort(
    Table.FromRecords({
        [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
        [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0],
        [OrderID = 3, CustomerID = 2, Item = "Fishing net", Price = 25.0],
        [OrderID = 4, CustomerID = 3, Item = "Fish tazer", Price = 200.0],
        [OrderID = 5, CustomerID = 3, Item = "Bandaids", Price = 2.0],
        [OrderID = 6, CustomerID = 1, Item = "Tackle box", Price = 20.0],
        [OrderID = 7, CustomerID = 5, Item = "Bait", Price = 3.25],
        [OrderID = 8, CustomerID = 5, Item = "Fishing Rod", Price = 100.0],
        [OrderID = 9, CustomerID = 6, Item = "Bait", Price = 3.25]
    }),
    {
        {"CustomerID", Order.Ascending},
        "OrderID"
    }
)
```

ORDERID	CUSTOMERID	[illegible]	[illegible]
1	1	钓鱼竿	100
2	1	1 磅蠕虫	5
6	1	钓具盒	20
3	2	渔网	25
4	3	电捕鱼器	200
5	3	创可贴	2
7	5	鱼饵	3.25
8	5	钓鱼竿	100
9	6	鱼饵	3.25

Table.Split

2020/4/30 •

语法

```
Table.Split(table as table, pageSize as number) as list
```

关于

将 `table` 拆分为表列表，其中列表的第一个元素是包含源表中前 `pageSize` 行的表，列表的下一个元素是包含源表中接下来 `pageSize` 行的表，以此类推。

示例 1

将包含五条记录的表拆分为各含两条记录的表。

```
let
    Customers = Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Cristina", Phone = "232-1550"],
        [CustomerID = 5, Name = "Anita", Phone = "530-1459"]
    })
in
    Table.Split(Customers, 2)
```

[表]
[表]
[表]

Table.SplitColumn

2020/4/30 •

语法

```
Table.SplitColumn(table as table, sourceColumn as text, splitter as function, optional
columnNamesOrNumber as any, optional default as any, optional extraColumns as any) as table
```

关于

使用指定的拆分器函数将指定的列拆分为一组其他列。

示例 1

将“i”处的 [Name] 列拆分为两列

```
let
    Customers = Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"],
        [CustomerID = 4, Name = "Cristina", Phone = "232-1550"]
    })
in
    Table.SplitColumn(Customers, "Name", Splitter.SplitTextByDelimiter("i"), 2
```

CUSTOMERID	NAME.1	NAME.2	ıı
1	Bob		123-4567
2	J	m	987-6543
3	Paul		543-7890
4	Cr	st	232-1550

Table.ToColumns

2020/4/30 •

语法

```
Table.ToColumns(table as table) as list
```

关于

从表 `table` 中创建嵌套列表的列表。每个列表项都是包含列值的内联列表。

示例

从表中创建列值的列表。

```
Table.ToColumns(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"]
    })
)
```

[列表]
[列表]
[列表]

Table.ToList

2020/4/30 •

语法

```
Table.ToList(table as table, optional combiner as nullable function) as list
```

关于

通过将指定的组合函数应用于表中的每一行值，将表转换为列表。

示例 1

使用逗号合并每行文本。

```
Table.ToList(
    Table.FromRows({
        {Number.ToText(1), "Bob", "123-4567"},
        {Number.ToText(2), "Jim", "987-6543"},
        {Number.ToText(3), "Paul", "543-7890"}
    }),
    Combiner.CombineTextByDelimiter(",")
)
```

1,Bob,123-4567
2,Jim,987-6543
3,Paul,543-7890

Table.ToRecords

2020/4/30 •

语法

```
Table.ToRecords(table as table) as list
```

关于

将表 `table` 转换为记录**列表**。

示例

将表转换为记录**列表**。

```
Table.ToRecords(
    Table.FromRows(
        {
            {1, "Bob", "123-4567"},
            {2, "Jim", "987-6543"},
            {3, "Paul", "543-7890"}
        },
        {"CustomerID", "Name", "Phone"}
    )
)
```

[记录]
[记录]
[记录]

Table.ToRows

2020/4/30 •

语法

```
Table.ToRows(table as table) as list
```

关于

从表 `table` 中创建嵌套列表的列表。每个列表项都是包含行值的内联列表。

示例

从表中创建行值的列表。

```
Table.ToRows(
    Table.FromRecords({
        [CustomerID = 1, Name = "Bob", Phone = "123-4567"],
        [CustomerID = 2, Name = "Jim", Phone = "987-6543"],
        [CustomerID = 3, Name = "Paul", Phone = "543-7890"]
    })
)
```

[列表]
[列表]
[列表]

Table.TransformColumnNames

2020/4/30 •

语法

```
Table.TransformColumnNames(table as table, nameGenerator as function, optional options as nullable record) as table
```

关于

使用给定的 `nameGenerator` 函数转换列名。有效选项:

`MaxLength` 指定新列名的最大长度。如果给定函数使得列名较长，则将对较长的名称进行剪裁。
`Comparer` 用于在生成新列名时控制比较。比较器可用于提供不区分大小写或区分区域性和区域设置的比较。
以下内置比较器支持公式语言:

- `Comparer.Ordinal` :用于执行精确的序号比较
- `Comparer.OrdinalIgnoreCase` :用于执行精确的、不区分大小写的序号比较
- `Comparer.FromCulture` :用于执行区分区域性的比较

示例 1

从列名中删除 `#(tab)` 字符

```
Table.TransformColumnNames(Table.FromRecords({[#"Col#(tab)umn" = 1]}), Text.Clean)
```

[
1

示例 2

转换列名以生成不区分大小写的名称(长度为 6)。

```
Table.TransformColumnNames(
    Table.FromRecords({[ColumnNum = 1, cOlumnnum = 2, coLumnNUM = 3]}),
    Text.Clean,
    [MaxLength = 6, Comparer = Comparer.OrdinalIgnoreCase]
)
```

[	COLUM1	COLUM2
1	2	3

Table.TransformColumns

2020/4/30 •

语法

```
Table.TransformColumns(table as table, transformOperations as list, optional defaultTransformation as nullable function, optional missingField as nullable number) as table
```

关于

通过对在参数 `transformOperations` 中指定的列应用转换操作（其中格式为 { column name, transformation }），从输入 `table` 中返回一个表。如果列不存在，则会引发异常，除非可选参数 `defaultTransformation` 指定替换选项（例如， `MissingField.UseNull` 或 `MissingField.Ignore` ）。

示例 1

将列 [A] 中的数字值转换为数字值。

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"A", Number.FromText}
)
```

A	B
1	2
5	10

示例 2

将缺失列 [X] 中的数字值转换为文本值，同时忽略不存在的列。

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText},
    null,
    MissingField.Ignore
)
```

A	B
1	2

5	10

示例 3

将缺失列 [X] 中的数字值转换为文本值，同时将不存在的列中的值默认设置为 null。

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText},
    null,
    MissingField.UseNull
)
```

A	B	X
1	2	
5	10	

示例 4

将缺失列 [X] 中的数字值转换为文本值，其中不存在的列出现错误。

```
Table.TransformColumns(
    Table.FromRecords({
        [A = "1", B = 2],
        [A = "5", B = 10]
    }),
    {"X", Number.FromText}
)
```

```
[Expression.Error] The column 'X' of the table wasn't found.
```

Table.TransformColumnTypes

2019/12/3 •

语法

```
Table.TransformColumnTypes(table as table, typeTransformations as list, optional culture as nullable text) as table
```

关于

通过对在参数 `typeTransformations` 中指定的列应用转换操作（其中格式为 { column name, type name}），使用参数 `culture` 中的指定区域性，从输入 `table` 中返回一个表。如果该列不存在，则引发异常。

示例 1

在表 `({[a = 1, b = 2], [a = 3, b = 4]})` 中将列 [a] 中的数值转换为文本值。

```
Table.TransformColumnTypes(Table.FromRecords({[a = 1, b = 2], [a = 3, b = 4]}), {"a", type text}, "en-US")
```

A	B
1	2
3	4

Table.TransformRows

2020/4/30 •

语法

```
Table.TransformRows(table as table, transform as function) as list
```

关于

通过对行应用 `transform` 操作，从 `table` 创建一个表。如果指定了 `transform` 函数的返回类型，则结果将是具有该行类型的表。在所有其他情况下，该函数的结果将是一个列表，其中包含转换函数返回类型的项类型。

示例 1

在表 `({[A = 1], [A = 2], [A = 3], [A = 4], [A = 5]})` 中将行转换为数值列表。

```
Table.TransformRows(
    Table.FromRecords({
        [a = 1],
        [a = 2],
        [a = 3],
        [a = 4],
        [a = 5]
    }),
    each [a]
)
```

1
2
3
4
5

示例 2

在表 `({[A = 1], [A = 2], [A = 3], [A = 4], [A = 5])` 中将列 [A] 中的行转换为列 [B] 中的文本值。

```
Table.TransformRows(
    Table.FromRecords({
        [a = 1],
        [a = 2],
        [a = 3],
        [a = 4],
        [a = 5]
    }),
    (row) as record => [B = Number.ToText(row[a])]
)
```

[记录]
[记录]
[记录]
[记录]
[记录]

Table.Transpose

2020/4/30 •

语法

```
Table.Transpose(table as table, optional columns as any) as table
```

关于

使列成为行，并使行成为列。

示例 1

使名称-值对表的行成为列。

```
Table.Transpose(
    Table.FromRecords({
        [Name = "Full Name", Value = "Fred"],
        [Name = "Age", Value = 42],
        [Name = "Country", Value = "UK"]
    })
)
```

[1	[2	COLUMN3
全名	年限	国家/地区
Fred	42	英国

Table.Unpivot

2020/4/30 •

语法

```
Table.Unpivot(table as table, pivotColumns as list, attributeColumn as text, valueColumn as text)
as table
```

关于

将表中的一组列转换为属性-值对，并与每行中的剩余值相结合。

示例 1

选取表 `({[ key = "x", a = 1, b = null, c = 3 ], [ key = "y", a = 2, b = 4, c = null ]})` 中的列“a”、“b”和“c”，并将它们逆透视为属性-值对。

```
Table.Unpivot(
    Table.FromRecords({
        [key = "x", a = 1, b = null, c = 3],
        [key = "y", a = 2, b = 4, c = null]
    }),
    {"a", "b", "c"},
    "attribute",
    "value"
)
```

[	[[	[
x	一个	1
x	C	3
y	一个	2
y	b	4

Table.UnpivotOtherColumns

2020/4/30 •

语法

```
Table.UnpivotOtherColumns(table as table, pivotColumns as list, attributeColumn as text,
valueColumn as text) as table
```

关于

将指定集以外的所有列转换为属性值对，与每行中的剩余值相合并。

示例 1

将指定集以外的所有列转换为属性值对，与每行中的剩余值相合并。

```
Table.UnpivotOtherColumns(
    Table.FromRecords({
        [key = "key1", attribute1 = 1, attribute2 = 2, attribute3 = 3],
        [key = "key2", attribute1 = 4, attribute2 = 5, attribute3 = 6]
    }),
    {"key"},
    "column1",
    "column2"
)
```

[	COLUMN1	COLUMN2
key1	attribute1	1
key1	attribute2	2
key1	attribute3	3
key2	attribute1	4
key2	attribute2	5
key2	attribute3	6

Table.View

2019/12/3 •

语法

```
Table.View(table as nullable table, handlers as record) as table
```

关于

返回 `table` 的视图，向视图应用运算时，会使用 `handlers` 中指定的函数代替运算的默认行为。处理程序函数是可选的。如果未对运算指定处理程序函数，则改为向 `table` 应用运算的默认行为（`GetExpression` 的情况除外）。

处理程序函数返回的值必须在语义上等效于向 `table` 应用运算的结果（在 `GetExpression` 的情况下则为向生成的视图应用）。

如果处理程序函数引发错误，则会向视图应用运算的默认行为。

`Table.View` 可用于实现到数据源的折叠，将 M 查询转换为特定于源的查询（例如，从 M 查询创建 T-SQL 语句）。

有关 `Table.View` 的更完整说明，请参阅已发布的文档。

Table.ViewFunction

2019/12/4 •

语法

```
Table.ViewFunction(function as function) as function
```

关于

基于 `function` 创建视图函数，此函数可以在 `Table.View` 创建的视图中处理。

`Table.View` 的 `OnInvoke` 处理程序可用于为视图函数定义处理程序。

与内置操作的处理程序一样，如果未指定 `OnInvoke` 处理程序，或者如果未处理视图函数，或者如何处理程序引发了错误，`function` 则将应用于视图的顶部。

有关 `Table.View` 和自定义视图函数更完整的说明，请参阅已发布的文档。

Tables.GetRelationships

2019/12/4 •

语法

```
Tables.GetRelationships(tables as table, optional dataColumn as nullable text) as table
```

关于

获取一组表之间的关系。假设集 `tables` 具有类似于导航表的结构。 `dataColumn` 定义的列包含实际数据表。

#table

2020/1/17 •

语法

```
#table(columns as any, rows as any) as any
```

关于

从列 `columns` 和列表 `rows` 创建一个表值，其中列表的每个元素都是包含单行列值的内部列表。`columns` 可能是列名列表、表类型、若干列或 NULL。

文本函数

2020/4/26 •

这些函数创建并操纵文本值。

文本

信息

Text.InferNumberType	使用 `culture` 推断 `text` 的粒度数字类型（Int64.Type、Double.Type 等）。
Text.Length	返回文本值中的字符数。

文本比较

Character.FromNumber	将数字返回到其字符值。
Character.ToNumber	将字符返回到数字值。
Guid.From	从给定的 `value` 返回 `Guid.Type` 值。
Json.FromValue	生成给定值的 JSON 表示形式。
Text.From	返回数字、日期、时间、日期时间、datetimezone、逻辑、持续时间或二进制值的文本表示形式。如果值为 NULL，则 Text.From 返回 NULL。可选的区域性参数用于根据给定区域性格式化文本值。
Text.FromBinary	使用编码将数据从二进制值解码为文本值。
Text.NewGuid	以文本值的形式返回 Guid 值。
Text.ToBinary	使用编码将文本值编码为二进制值。
Text.ToList	从文本值返回字符列表。
Value.FromText	从文本表示形式解码一个值，并将其解释为具有适当类型的值。Value.FromText 采用文本值并返回数字、逻辑值、NULL 值、DateTime 值、Duration 值或文本值。空文本值将被解释为 NULL 值。

提取

Text.At	返回从零开始的偏移处的字符。

Text.Middle	返回最长为某个特定长度的 substring。
Text.Range	返回文本值中由从零开始的偏移起一定数量的字符。
Text.Start	返回从文本值开头的给定数量的字符。

Text.End	返回从文本值末尾的给定数量的字符。

修改

Text.Insert	返回一个文本值，将 newValue 插入到文本值中从零开始的偏移处。
Text.Remove	删除文本值中某字符或字符列表的所有实例。removeChars 参数可以是字符值或字符值列表。
Text.RemoveRange	删除文本值中由从零开始的偏移起一定数量的字符。
Text.Replace	将某 substring 的所有实例替换为新的文本值。
Text.ReplaceRange	将文本值中由从零开始的偏移起一定长度的字符替换为新的文本值。
Text.Select	选中输入文本值中给定字符或字符列表的所有实例。

成员身份

Text.Contains	如果在文本值字符串中找到文本值 substring，则返回 true；否则为 false。
Text.EndsWith	返回一个逻辑值，指示是否在字符串末尾找到某个文本值 substring。
Text.PositionOf	返回字符串中某 substring 的第一个实例，并返回其基于 startOffset 的位置。
Text.PositionOfAny	返回列表中某文本值的第一个实例，并返回其基于 startOffset 的位置。
Text.StartsWith	返回一个逻辑值，指示是否在字符串开头找到某个文本值 substring。

转换

“	“
Text.AfterDelimiter	返回指定分隔符之后的文本部分。
Text.BeforeDelimiter	返回指定分隔符之前的文本部分。
Text.BetweenDelimiters	返回指定的 startDelimiter 和 endDelimiter 之间的文本部分。
Text.Clean	返回去除了非打印字符的原始文本值。
Text.Combine	返回联接所有文本值(用分隔符分隔的每个值)得到的文本值。
Text.Lower	返回文本值的小写形式。
Text.PadEnd	返回末尾用 pad 填充到一定长度的文本值。
Text.PadStart	返回开头用 pad 填充到一定长度的文本值。如果未指定 pad,则将空格用作填充。
Text.Proper	返回一个文本值,其中所有单词的首字母转换为大写。
Text.Repeat	返回由输入文本值重复一定次数后得到的文本值。
Text.Reverse	反写所提供的文本。
Text.Split	返回包含由分隔符文本值分隔的文本值部分的列表。
Text.SplitAny	返回包含由任意分隔符文本值分隔的文本值部分的列表。
Text.Trim	从文本中删除 trimChars 中字符的任何实例。
Text.TrimEnd	从原始文本值的末尾删除 trimChars 中所指定字符的任何实例。
Text.TrimStart	从原始文本值的开头删除 trimChars 中字符的任何实例。
Text.Upper	返回文本值的大写形式。

参数

“‘	“
Occurrence.All	返回找到的值所有实例的位置列表。
Occurrence.First	返回找到的值第一次出现的位置。
Occurrence.Last	返回找到的值最后一次出现的位置。
RelativePosition.FromEnd	指示应从输入结尾编制索引。
RelativePosition.FromStart	指示应从输入开头编制索引。
TextEncoding.Ascii	用于选择 ASCII 二进制格式。

[illegible]	[illegible]
TextEncoding.BigEndianUnicode	用于选择 UTF16 big endian 二进制格式。
TextEncoding.Unicode	用于选择 UTF16 little endian 二进制格式。
TextEncoding.Utf8	用于选择 UTF8 二进制格式。
TextEncoding.Utf16	用于选择 UTF16 little endian 二进制格式。
TextEncoding.Windows	用于选择 Windows 二进制格式。

Character.FromNumber

2019/12/3 •

语法

```
Character.FromNumber(number as nullable number) as nullable text
```

关于

返回与该数值等效的字符。

示例 1

假定提供数值 9，则查找字符值。

```
Character.FromNumber(9)
```

```
"#(tab)"
```

Character.ToNumber

2019/12/3 •

语法

```
Character.ToNumber(character as nullable text) as nullable number
```

关于

返回与字符 `character` 等效的数值。

示例 1

给定字符 "#(tab)" 9，查找等效的数值。

```
Character.ToNumber("#(tab)")
```

```
9
```

Guid.From

2019/12/3 •

语法

```
Guid.From(value as nullable text) as nullable text
```

关于

从给定的 `value` 返回 `Guid.Type` 值。如果给定的 `value` 为 `null`，则 `Guid.From` 返回 `null`。将执行检查以查看给定 `value` 是否为可接受的格式。示例中提供了可接受的格式。

示例 1

可以将 Guid 提供为 32 个连续的十六进制数字。

```
Guid.From("05FE1DADC8C24F3BA4C2D194116B4967")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

示例 2

可以将 Guid 提供为 32 个由连字符分隔成 8-4-4-4-12 块的十六进制数字。

```
Guid.From("05FE1DAD-C8C2-4F3B-A4C2-D194116B4967")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

示例 3

可以将 Guid 提供为 32 个由连字符分隔且括在大括号中的十六进制数字。

```
Guid.From("{05FE1DAD-C8C2-4F3B-A4C2-D194116B4967}")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

示例 4

可以将 Guid 提供为 32 个由连字符分隔且括在括号中的十六进制数字。

```
Guid.From("(05FE1DAD-C8C2-4F3B-A4C2-D194116B4967)")
```

`"05fe1dad-c8c2-4f3b-a4c2-d194116b4967"`

Json.FromValue

2020/4/30 •

语法

```
Json.FromValue(value as any, optional encoding as nullable number) as binary
```

关于

使用 `value` `encoding` 指定的文本编码生成给定值的 JSON 表示形式。如果省略 `encoding`，则使用 UTF8。值按如下方式表示：

- Null、文本和逻辑值表示为相应的 JSON 类型
- 编号表示为 JSON 中的数字，但 `#infinity`、`-#infinity` 和 `#nan` 都转换为 null
- 列表表示为 JSON 数组
- 记录表示为 JSON 对象
- 表格表示为对象数组
- 日期、时间、日期/时间、日期时间时区和持续时间表示为 ISO-8601 文本
- 二进制值表示为 base-64 编码文本
- 类型和函数产生错误

示例 1

将复杂值转换为 JSON。

```
Text.FromBinary(Json.FromValue([A = {1, true, "3"}, B = #date(2012, 3, 25)]))
```

```
"{""A"":[1,true,""3""],""B"":""2012-03-25""}"
```

RelativePosition.FromEnd

2019/12/4 •

关于

指示应从输入结尾编制索引。

RelativePosition.FromStart

2019/12/3 •

关于

指示应从输入开头编制索引。

Text.AfterDelimiter

2019/12/3 •

语法

```
Text.AfterDelimiter(text as nullable text, delimiter as text, optional index as any) as any
```

关于

返回 `text` 中指定 `delimiter` 后的部分。可选的数值 `index` 指示应考虑 `delimiter` 的哪一次出现。可选列表 `index` 指示应考虑 `delimiter` 的哪一次出现，以及是否应从输入的开头或结尾编制索引。

示例 1

获取"111-222-333"(第一个)连字符后的部分。

```
Text.AfterDelimiter("111-222-333", "-")
```

`"222-333"`

示例 2

获取"111-222-333"第二个连字符后的部分。

```
Text.AfterDelimiter("111-222-333", "-", 1)
```

`"333"`

示例 3

获取"111-222-333"倒数第二个连字符后的部分。

```
Text.AfterDelimiter("111-222-333", "-", {1, RelativePosition.FromEnd})
```

`"222-333"`

Text.At

2019/12/4 •

语法

```
Text.At(text as nullable text, index as number) as nullable text
```

关于

返回在文本值 `text` 中位于第 `index` 位的字符。文本中的第一个字符位于位置 0。

示例 1

查找位于字符串“Hello, World”中第 4 位的字符。

```
Text.At("Hello, World", 4)
```

`"o"`

Text.BeforeDelimiter

2019/12/3 •

语法

```
Text.BeforeDelimiter(text as nullable text, delimiter as text, optional index as any) as any
```

关于

返回 `text` 中指定 `delimiter` 前的部分。可选的数值 `index` 指示应考虑 `delimiter` 的哪一次出现。可选列表 `index` 指示应考虑 `delimiter` 的哪一次出现，以及是否应从输入的开头或结尾编制索引。

示例 1

获取"111-222-333"(第一个)连字符前的部分。

```
Text.BeforeDelimiter("111-222-333", "-")
```

`"111"`

示例 2

获取"111-222-333"第二个连字符前的部分。

```
Text.BeforeDelimiter("111-222-333", "-", 1)
```

`"111-222"`

示例 3

获取"111-222-333"倒数第二个连字符前的部分。

```
Text.BeforeDelimiter("111-222-333", "-", {1, RelativePosition.FromEnd})
```

`"111"`

Text.BetweenDelimiters

2019/12/4 •

语法

```
Text.BetweenDelimiters(text as nullable text, startDelimiter as text, endDelimiter as text,
optional startIndex as any, optional endIndex as any) as any
```

关于

返回指定的 `startDelimiter` 和 `endDelimiter` 之间 `text` 的部分。可选的数值 `startIndex` 指示应考虑 `startDelimiter` 的哪一次出现。可选列表 `startIndex` 指示应考虑 `startDelimiter` 的哪一次出现，以及是否应从输入的开头或结尾编制索引。 `endIndex` 与之类似，只不过索引是相对于 `startIndex` 完成的。

示例 1

获取 "111 (222) 333 (444)" (第一个)左括号及其随后(第一个)右括号之间的部分。

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")")
```

`"222"`

示例 2

获取 "111 (222) 333 (444)" 第二个左括号及其随后第一个右括号之间的部分。

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")", 1, 0)
```

`"444"`

示例 3

获取 "111 (222) 333 (444)" 倒数第二个左括号及其随后第二个右括号之间的部分。

```
Text.BetweenDelimiters("111 (222) 333 (444)", "(", ")", {1, RelativePosition.FromEnd}, {1,
RelativePosition.FromStart})
```

`"222) 333 (444"`

Text.Clean

2019/12/4 •

语法

```
Text.Clean(text as nullable text) as nullable text
```

关于

返回 `text` 的所有非打印字符均已删除的文本值。

示例 1

从文本值中删除换行和其他非打印字符。

```
Text.Clean("ABC#(lf)D")
```

`"ABCD"`

Text.Combine

2019/12/3 •

语法

```
Text.Combine(texts as list, optional separator as nullable text) as text
```

关于

返回将文本值列表（`texts`）合并为单个文本值的结果。可以指定在最终合并文本中使用的可选分隔符，`separator`。

示例 1

组合文本值 "Seattle" 和 "WA"。

```
Text.Combine({"Seattle", "WA"})
```

`"SeattleWA"`

示例 2

组合文本值 "Seattle" 和 "WA"，以逗号和空格 ", " 分隔。

```
Text.Combine({"Seattle", "WA"}, ", ")
```

`"Seattle, WA"`

Text.Contains

2019/12/4 •

语法

```
Text.Contains(text as nullable text, substring as text, optional comparer as nullable function) as nullable logical
```

关于

检测文本 `text` 是否包含文本 `substring`。如果找到文本，则返回 true。

`comparer` 是用于控制比较的 `Comparer`。比较器可用于提供不区分大小写或区分区域性和区域设置的比较。
以下内置比较器支持公式语言：

- `Comparer.Ordinal`：用于执行精确的序号比较
- `Comparer.OrdinalIgnoreCase`：用于执行精确的、不区分大小写的序号比较
- `Comparer.FromCulture`：用于执行区分区域性的比较

示例 1

查找文本“Hello World”是否包含“Hello”。

```
Text.Contains("Hello World", "Hello")
```

`true`

示例 2

查找文本“Hello World”是否包含“hello”。

```
Text.Contains("Hello World", "hello")
```

`false`

Text.End

2019/12/3 •

语法

```
Text.End(text as nullable text, count as number) as nullable text
```

关于

返回一个 `text` 值，该值是 `text` 值 `text` 的后 `count` 个字符。

示例 1

获取文本“Hello, World”的后 5 个字符。

```
Text.End("Hello, World", 5)
```

`"World"`

Text.EndsWith

2019/12/3 •

语法

```
Text.EndsWith(text as nullable text, substring as text, optional comparer as nullable function) as nullable logical
```

关于

指示给定的文本 `text` 是否以指定的值 `substring` 结尾。指示文本区分大小写。

`comparer` 是用于控制比较的 `Comparer`。比较器可用于提供不区分大小写或区分区域性和区域设置的比较。
以下内置比较器支持公式语言：

- `Comparer.Ordinal`：用于执行精确的序号比较
- `Comparer.OrdinalIgnoreCase`：用于执行精确的、不区分大小写的序号比较
- `Comparer.FromCulture`：用于执行区分区域性的比较

示例 1

检查“Hello, World”是否以“world”结尾。

```
Text.EndsWith("Hello, World", "world")
```

`false`

示例 2

检查“Hello, World”是否以“World”结尾。

```
Text.EndsWith("Hello, World", "World")
```

`true`

Text.Format

2019/12/3 •

语法

```
Text.Format(formatString as text, arguments as any, optional culture as nullable text) as text
```

关于

返回通过将来自列表或记录的 `arguments` 应用于格式字符串 `formatString` 创建的格式化文本。可以视情况指定区域性。

示例 1

设置数字列表的格式。

```
Text.Format("#{0}, #{1}, and #{2}.", { 17, 7, 22 })
```

```
"17, 7, and 22."
```

示例 2

根据美国英语区域性设置记录中的不同数据类型的格式。

```
Text.Format("The time for the #[distance] km run held in #[city] on #[date] was #[duration].", [city =
"Seattle", date = #date(2015, 3, 10), duration = #duration(0,0,54,40), distance = 10], "en-US")
```

```
"The time for the 10 km run held in Seattle on 3/10/2015 was 00:54:40."
```

Text.From

2019/12/3 •

语法

```
Text.From(value as any, optional culture as nullable text) as nullable text
```

关于

返回 `value` 的文本表示形式。 `value` 可以是 `number` 、 `date` 、 `time` 、 `datetime` 、 `datetimezone` 、 `logical` 、 `duration` 或 `binary` 值。如果给定的值为 null, `Text.From` 将返回 null。还可以提供可选 `culture` 。

示例 1

从数字 3 创建一个文本值。

```
Text.From(3)
```

`"3"`

Text.FromBinary

2019/12/4 •

语法

```
Text.FromBinary(binary as nullable binary, optional encoding as nullable number) as nullable text
```

关于

使用 `encoding` 类型将数据 `binary` 从二进制值解码为文本值。

Text.InferNumberType

2019/12/4 •

语法

```
Text.InferNumberType(text as text, optional culture as nullable text) as type
```

关于

使用 `culture` 推断 `text` 的粒度数字类型(Int64.Type、Double.Type 等)。如果 `text` 不是数字，则会引发异常

Text.Insert

2019/12/3 •

语法

```
Text.Insert(text as nullable text, offset as number, newText as text) as nullable text
```

关于

返回将文本值 `newText` 插入到位置 `offset` 的文本值 `text` 中的结果。位置从数字 0 开始。

示例 1

在 "ABD" 中的 "B" 和 "D" 之间插入 "C"。

```
Text.Insert("ABD", 2, "C")
```

`"ABCD"`

Text.Length

2019/12/4 •

语法

```
Text.Length(text as nullable text) as nullable number
```

关于

返回文本 `text` 中的字符数。

示例 1

查找文本“Hello World”中有多少个字符。

```
Text.Length("Hello World")
```

```
11
```

Text.Lower

2019/12/4 •

语法

```
Text.Lower(text as nullable text, optional culture as nullable text) as nullable text
```

关于

返回将 `text` 中的所有字符转换为小写的结果。

示例 1

获取“AbCd”的小写版本。

```
Text.Lower("AbCd")
```

```
"abcd"
```

Text.Middle

2019/12/4 •

语法

```
Text.Middle(text as nullable text, start as number, optional count as nullable number) as nullable
text
```

关于

返回 `count` 个字符，或返回至 `text` 的结束；偏移量为 `start`。

示例 1

从文本"Hello World"中查找从索引 6 开始、跨 5 个字符的子字符串。

```
Text.Middle("Hello World", 6, 5)
```

`"World"`

示例 2

从文本"Hello World"中查找从索引 6 开始到结束的子字符串。

```
Text.Middle("Hello World", 6, 20)
```

`"World"`

Text.NewGuid

2019/12/4 •

语法

```
Text.NewGuid() as text
```

关于

返回新的、随机的全局唯一标识符 (GUID)。

Text.PadEnd

2020/4/30 •

语法

```
Text.PadEnd(text as nullable text, count as number, optional character as nullable text) as
nullable text
```

关于

通过在文本值 `text` 的末尾插入空格，返回填充到长度 `count` 的 `text` 值。可使用可选字符 `character` 来指定用于填充的字符。默认填充字符为空格。

示例 1

填充文本值的末尾，使其长度为 10 个字符。

```
Text.PadEnd("Name", 10)
```

```
"Name      "
```

示例 2

用"|"填充文本值的末尾，使其长度为 10 个字符。

```
Text.PadEnd("Name", 10, "|")
```

```
"Name||||||"
```

Text.PadStart

2020/4/30 •

语法

```
Text.PadStart(text as nullable text, count as number, optional character as nullable text) as
nullable text
```

关于

通过在文本值 `text` 的开头插入空格，返回填充到长度 `count` 的 `text` 值。可使用可选字符 `character` 来指定用于填充的字符。默认填充字符为空格。

示例 1

填充文本值的开头，使其长度为 10 个字符。

```
Text.PadStart("Name", 10)
```

```
"      Name"
```

示例 2

用 "|" 填充文本值的开头，使其长度为 10 个字符。

```
Text.PadStart("Name", 10, "|")
```

```
"||||||Name"
```

Text.PositionOf

2019/12/4 •

语法

```
Text.PositionOf(text as text, substring as text, optional occurrence as nullable number, optional comparer as nullable function) as any
```

关于

返回在 `text` 中找到的文本值 `substring` 的指定出现位置。可使用可选参数 `occurrence` 来指定要返回的出现位置(默认为第一次出现的位置)。如果找不到 `substring`，则返回 -1。

`comparer` 是用于控制比较的 `Comparer`。比较器可用于提供不区分大小写或区分区域性和区域设置的比较。
以下内置比较器支持公式语言:

- `Comparer.Ordinal`:用于执行精确的序号比较
- `Comparer.OrdinalIgnoreCase`:用于执行精确的、不区分大小写的序号比较
- `Comparer.FromCulture`:用于执行区分区域性的比较

示例 1

获取"World"在"Hello, World! Hello, World!"文本中 第一次出现的位置。

```
Text.PositionOf("Hello, World! Hello, World!", "World")
```

`7`

示例 2

获取"World"在"Hello, World! Hello, World!"文本中 最后一次出现的位置。

```
Text.PositionOf("Hello, World! Hello, World!", "World", Occurrence.Last)
```

`21`

Text.PositionOfAny

2020/4/30 •

语法

```
Text.PositionOfAny(text as text, characters as list, optional occurrence as nullable number) as
any
```

关于

返回在文本值 `characters` 中找到的字符列表 `text` 中第一次出现任何字符的位置。可选参数 `occurrence` 可用于指定要返回的出现位置。

示例 1

查找"W"在文本"Hello, World!"中的位置。

```
Text.PositionOfAny("Hello, World!", {"W"})
```

`7`

示例 2

查找"W"或"H"在文本"Hello, World!"中的位置。

```
Text.PositionOfAny("Hello, World!", {"H", "W"})
```

`0`

Text.Proper

2019/12/3 •

语法

```
Text.Proper(text as nullable text, optional culture as nullable text) as nullable text
```

关于

返回只使文本值 `text` 中每个字词的第一个字母大写的结果。所有其他字母均以小写返回。

示例 1

对简单的句子使用 `Text.Proper`。

```
Text.Proper("the QUICK BrOWn fOx jUmPs oVER tHe LAzy DoG")
```

`"The Quick Brown Fox Jumps Over The Lazy Dog"`

Text.Range

2019/12/4 •

语法

```
Text.Range(text as nullable text, offset as number, optional count as nullable number) as nullable text
```

关于

返回在文本 `text` 中偏移量 `offset` 处找到的 substring。可以包含可选参数 `count`，以指定要返回的字符数。如果没有足够的字符，则会引发错误。

示例 1

从文本“Hello World”中查找从索引 6 开始的 substring。

```
Text.Range("Hello World", 6)
```

`"World"`

示例 2

从文本“Hello World Hello”中查找从索引 6 开始且涵盖 5 个字符的 substring。

```
Text.Range("Hello World Hello", 6, 5)
```

`"World"`

Text.Remove

2020/4/30 •

语法

```
Text.Remove(text as nullable text, removeChars as any) as nullable text
```

关于

返回文本值 `text` 的副本，其中已删除了 `removeChars` 中的所有字符。

示例 1

从文本值中删除字符 , 和 ;。

```
Text.Remove("a,b;c", {",",";"})
```

`"abc"`

Text.RemoveRange

2019/12/3 •

语法

```
Text.RemoveRange(text as nullable text, offset as number, optional count as nullable number) as
nullable text
```

关于

返回文本值 `text` 已删除了 `offset` 位置后所有字符的副本。可选参数 `count` 可以用来指定要删除的字符数。 `count` 的默认值为 1。位置值从 0 开始。

示例 1

删除文本值“ABEFC”中位置 2 的 1 个字符。

```
Text.RemoveRange("ABEFC", 2)
```

`"ABFC"`

示例 2

删除文本值“ABEFC”中从位置 2 开始的 2 个字符。

```
Text.RemoveRange("ABEFC", 2, 2)
```

`"ABC"`

Text.Repeat

2019/12/3 •

语法

```
Text.Repeat(text as nullable text, count as number) as nullable text
```

关于

返回由输入文本 `text` 重复 `count` 次而组成的文本值。

示例 1

重复文本“a”五次。

```
Text.Repeat("a", 5)
```

`"aaaaa"`

示例 2

重复文本“helloworld”三次。

```
Text.Repeat("helloworld.", 3)
```

`"helloworld.helloworld.helloworld."`

Text.Replace

2019/12/4 •

语法

```
Text.Replace(text as nullable text, old as text, new as text) as nullable text
```

关于

返回将文本值 `text` 中所有出现的文本值 `old` 替换为文本值 `new` 的结果。此函数区分大小写。

示例 1

将句子中出现的每个“the”替换为“a”。

```
Text.Replace("the quick brown fox jumps over the lazy dog", "the", "a")
```

```
"a quick brown fox jumps over a lazy dog"
```

Text.ReplaceRange

2019/12/4 •

语法

```
Text.ReplaceRange(text as nullable text, offset as number, count as number, newText as text) as
nullable text
```

关于

返回从文本值 `text` 中的位置 `offset` 开始删除一些字符 `count`，然后在 `text` 中的相同位置插入文本值 `newText` 的结果。

示例 1

使用新文本值“CDE”替换文本值“ABGF”中位置 2 的单个字符。

```
Text.ReplaceRange("ABGF", 2, 1, "CDE")
```

`"ABCDEF"`

Text.Reverse

2019/12/4 •

语法

```
Text.Reverse(text as nullable text) as nullable text
```

关于

反写所提供的 `text`。

示例 1

反写文本“123”。

```
Text.Reverse("123")
```

`"321"`

Text.Select

2019/12/3 •

语法

```
Text.Select(text as nullable text, selectChars as any) as nullable text
```

关于

返回文本值 `text` 的副本，其中已删除 `selectChars` 中不存在的所有字符。

示例 1

从文本值中选择范围从 "a" 到 "z" 的所有字符。

```
Text.Select("a,b;c", {"a".."z"})
```

`"abc"`

Text.Split

2019/12/4 •

语法

```
Text.Split(text as text, separator as text) as list
```

关于

返回根据指定的分隔符 `separator` 拆分文本值 `text` 而得到的文本值列表。

示例 1

从由 "|" 分隔的文本值 "Name|Address|PhoneNumber" 创建列表。

```
Text.Split("Name|Address|PhoneNumber", "|")
```

名称
地址
电话号码

Text.SplitAny

2019/12/4 •

语法

```
Text.SplitAny(text as text, separators as text) as list
```

关于

返回根据指定的分隔符 `separators` 中的任意字符拆分文本值 `text` 而得到的文本值列表。

示例 1

从文本值“Jamie|Campbell|Admin|Adventure Works|www.adventure-works.com”创建列表。

```
Text.SplitAny("Jamie|Campbell|Admin|Adventure Works|www.adventure-works.com", "|")
```

Jamie
Campbell
管理员
Adventure Works
www.adventure-works.com

Text.Start

2019/12/4 •

语法

```
Text.Start(text as nullable text, count as number) as nullable text
```

关于

返回 `text` 的前 `count` 个字符作为文本值。

示例 1

获取“Hello, World”的前 5 个字符。

```
Text.Start("Hello, World", 5)
```

```
"Hello"
```

Text.StartsWith

2019/12/4 •

语法

```
Text.StartsWith(text as nullable text, substring as text, optional comparer as nullable function)
as nullable logical
```

关于

如果文本值 `text` 以文本值 `substring` 开头，则返回 true。

- `text`：要搜索的 `text` 值
- `substring`：一个 `text` 值，它是要在 `substring` 中搜索的子字符串
- `comparer`：[可选] 用于控制比较的 `Comparer`。例如，`Comparer.OrdinalIgnoreCase` 可用于执行不区分大小写的搜索

`comparer` 是用于控制比较的 `Comparer`。比较器可用于提供不区分大小写或区分区域性和区域设置的比较。
以下内置比较器支持公式语言：

- `Comparer.Ordinal`：用于执行精确的序号比较
- `Comparer.OrdinalIgnoreCase`：用于执行精确的、不区分大小写的序号比较
- `Comparer.FromCulture`：用于执行区分区域性的比较

示例 1

检查文本“Hello, World”是否以文本“hello”开头。

```
Text.StartsWith("Hello, World", "hello")
```

`false`

示例 2

检查文本“Hello, World”是否以文本“Hello”开头。

```
Text.StartsWith("Hello, World", "Hello")
```

`true`

Text.ToBinary

2019/12/4 •

语法

```
Text.ToBinary(text as nullable text, optional encoding as nullable number, optional
includeByteOrderMark as nullable logical) as nullable binary
```

关于

使用指定的 `encoding` 将给定的文本值 `text` 编码为二进制值。

Text.ToList

2019/12/3 •

语法

```
Text.ToList(text as text) as list
```

关于

从给定的文本值 `text` 返回字符值列表。

示例 1

从文本“Hello World”创建字符值列表。

```
Text.ToList("Hello World")
```

H
e
l
l
o
W
o
r
l
d

Text.Trim

2020/4/30 •

语法

```
Text.Trim(text as nullable text, optional trim as any) as nullable text
```

关于

返回从文本值 `text` 删除所有前导空格和尾随空格的结果。

示例 1

删除“ a b c d ”中的前导空格和尾随空格。

```
Text.Trim("    a b c d   ")
```

```
"a b c d"
```

Text.TrimEnd

2020/4/30 •

语法

```
Text.TrimEnd(text as nullable text, optional trim as any) as nullable text
```

关于

返回从文本值 `text` 中删除所有尾随空格的结果。

示例 1

删除“ a b c d ”中的尾随空格。

```
Text.TrimEnd("    a b c d   ")
```

```
"    a b c d"
```

Text.TrimStart

2020/4/30 •

语法

```
Text.TrimStart(text as nullable text, optional trim as any) as nullable text
```

关于

返回从文本值 `text` 删除所有前导空格的结果。

示例 1

删除“ a b c d ”中的前导空格。

```
Text.TrimStart("   a b c d   ")
```

```
"a b c d   "
```

Text.Upper

2019/12/4 •

语法

```
Text.Upper(text as nullable text, optional culture as nullable text) as nullable text
```

关于

返回将 `text` 中的所有字符转换为大写的结果。

示例 1

获取“aBcD”的大写版本。

```
Text.Upper("aBcD")
```

`"ABCD"`

TextEncoding.Ascii

2019/12/4 •

关于

用于选择 ASCII 二进制格式。

TextEncoding.BigEndianUnicode

2019/12/4 •

关于

用于选择 UTF16 big endian 二进制格式。

TextEncoding.Unicode

2019/12/3 •

关于

用于选择 UTF16 little endian 二进制格式。

TextEncoding.Utf8

2019/12/4 •

关于

用于选择 UTF8 二进制格式。

TextEncoding.Utf16

2019/12/4 •

关于

用于选择 UTF16 little endian 二进制格式。

TextEncoding.Windows

2019/12/3 •

关于

用于选择 Windows 二进制格式。

时间函数

2020/4/26 •

这些函数创建并操纵 time 值。

时间

"	"
Time.EndOfHour	返回小时结束时的 DateTime 值。
Time.From	返回值中的时间值。
Time.FromText	返回一组日期格式中的时间值。
Time.Hour	返回 DateTime 值的小时值。
Time.Minute	返回 DateTime 值的分钟值。
Time.Second	返回 DateTime 值中的秒值
Time.StartOfHour	返回时间值中的第一个小时值。
Time.ToRecord	返回包含 Date 值的各个部分的记录。
Time.ToText	返回 Time 值的文本值。
#time	从小时、分钟和秒创建一个时间值。

Time.EndOfHour

2019/12/3 •

语法

```
Time.EndOfHour(dateTime as any) as any
```

关于

返回 `time`、`datetime` 或者表示所述小时结束值 `dateTime` 的 `datetimezone`，包括分数秒。保留时区信息。

- `dateTime`：用于计算小时结束值的 `time`、`datetime` 或 `datetimezone` 值。

示例 1

获取 2011/5/14 下午 05:00:00 的小时结束值。

```
Time.EndOfHour(#datetime(2011, 5, 14, 17, 0, 0))
```

```
#datetime(2011, 5, 14, 17, 59, 59.9999999)
```

示例 2

获取 2011/5/17 下午 05:00:00-7:00 的小时结束值。

```
Time.EndOfHour(#datetimezone(2011, 5, 17, 5, 0, 0, -7, 0))
```

```
#datetimezone(2011, 5, 17, 5, 59, 59.9999999, -7, 0)
```

Time.From

2019/12/3 •

语法

```
Time.From(value as any, optional culture as nullable text) as nullable time
```

关于

从给定的 `value` 返回 `time` 值。如果给定的 `value` 为 `null`，则 `Time.From` 返回 `null`。如果给定的 `value` 为 `time`，则返回 `value`。可以将以下类型的值转换为 `time` 值：

- `text`：文本表示形式的 `time` 值。有关详细信息，请参阅 `Time.FromText`。
- `datetime`：`value` 的时间部分。
- `datetimezone`：与 `value` 等效的本地日期/时间的时间部分。
- `number`：与由 `value` 表示的不完整天数等效的 `time`。如果 `value` 为负数或大于或等于 1，则返回错误。

如果 `value` 为任何其他类型，则返回错误。

示例 1

将 `0.7575` 转换为 `time` 值。

```
Time.From(0.7575)
```

`#time(18,10,48)`

示例 2

将 `#datetime(1899, 12, 30, 06, 45, 12)` 转换为 `time` 值。

```
Time.From(#datetime(1899, 12, 30, 06, 45, 12))
```

`#time(06, 45, 12)`

Time.FromText

2019/12/3 •

语法

```
Time.FromText(text as nullable text, optional culture as nullable text) as nullable time
```

关于

根据 ISO 8601 格式标准，从文本表示形式 `text` 创建 `time` 值。

- `Time.FromText("12:34:12")` // Time, hh:mm:ss
- `Time.FromText("12:34:12.1254425")` // hh:mm:ss.nnnnnnn

示例 1

将 `"10:12:31am"` 转换为时间值。

```
Time.FromText("10:12:31am")
```

`#time(10, 12, 31)`

示例 2

将 `"1012"` 转换为时间值。

```
Time.FromText("1012")
```

`#time(10, 12, 00)`

示例 3

将 `"10"` 转换为时间值。

```
Time.FromText("10")
```

`#time(10, 00, 00)`

Time.Hour

2019/12/3 •

语法

```
Time.Hour(dateTime as any) as nullable number
```

关于

返回所提供的 `time`、`datetime` 或 `datetimezone` 值、`dateTime` 的小时部分。

示例 1

查找 #datetime(2011, 12, 31, 9, 15, 36) 中的小时。

```
Time.Hour(#datetime(2011, 12, 31, 9, 15, 36))
```

`9`

Time.Minute

2019/12/4 •

语法

```
Time.Minute(dateTime as any) as nullable number
```

关于

返回所提供的 `time`、`datetime` 或 `datetimezone` 值的分钟部分，`dateTime`。

示例 1

查找 #datetime(2011, 12, 31, 9, 15, 36) 中的分钟。

```
Time.Minute(#datetime(2011, 12, 31, 9, 15, 36))
```

`15`

Time.Second

2019/12/3 •

语法

```
Time.Second(dateTime as any) as nullable number`
```

关于

返回所提供的 `time` 、 `datetime` 或 `datetimezone` 值 `dateTime` 的秒部分。

示例 1

查找日期/时间值中的秒值。

```
Time.Second(#datetime(2011, 12, 31, 9, 15, 36.5))
```

`36.5`

Time.StartOfHour

2019/12/3 •

语法

```
Time.StartOfHour(dateTime as any) as any
```

关于

给定 `time`、`datetime` 或 `datetimezone` 类型，返回小时的第一个值。

示例 1

查找 2011 年 10 月 10 日上午 8:10:32 (`#datetime(2011, 10, 10, 8, 10, 32)`) 的小时开始值。

```
Time.StartOfHour(#datetime(2011, 10, 10, 8, 10, 32))
```

`#datetime(2011, 10, 10, 8, 0, 0)`

Time.ToRecord

2019/12/4 •

语法

```
Time.ToRecord(time as time) as record
```

关于

返回包含给定时间值 `time` 的各个部分的记录。

- `time`：用于计算其部分的记录的 `time` 值。

示例 1

将 `#time(11, 56, 2)` 值转换为包含时间值的记录。

```
Time.ToRecord(#time(11, 56, 2))
```

[Hour]	11
[Minute]	56
[Second]	2

Time.ToText

2019/12/4 •

语法

```
Time.ToText(time as nullable time, optional format as nullable text, optional culture as nullable text) as nullable text
```

关于

返回时间值 `time` 的文本表示形式，`time` 。此函数采用可选格式参数 `format` 。有关所支持格式的完整列表，请参阅库规范文档。

示例 1

获取 #time(11, 56, 2) 的文本表示形式。

```
Time.ToText(#time(11, 56, 2))
```

`"11:56 AM"`

示例 2

使用格式选项获取 #time(11, 56, 2)的文本表示形式。

```
Time.ToText(#time(11, 56, 2), "hh:mm")
```

`"11:56"`

#time

2019/12/3 •

语法

```
#time(hour as number, minute as number, second as number) as time
```

关于

根据整数小时 `hour`、分钟 `minute` 和(小数)秒 `second` 创建时间值。如果不满足以下条件，则会引发错误：

- 0 ≤ hour ≤ 24
- 0 ≤ minute ≤ 59
- 0 ≤ second ≤ 59
- 如果 hour 为 24，则 minute 和 second 必须为 0

类型函数

2020/4/26 •

这些函数创建并操纵类型值。

类型

"	"
Type.AddTableKey	向表类型添加键。
Type.ClosedRecord	给定类型必须是记录类型，返回给定记录类型的已关闭版本（如果已关闭，则返回相同类型）
Type.Facets	返回类型的 Facet。
Type.ForFunction	从给定的类型创建函数类型。
Type.ForRecord	从字段记录返回记录类型。
Type.FunctionParameters	返回一个记录，它的字段值设置为函数类型的参数名称，值设置为相应的类型。
Type.FunctionRequiredParameters	返回一个数字，表明调用函数类型所需的最少参数数目。
Type.FunctionReturn	返回由函数类型返回的类型。
Type.Is	Type.Is
Type.IsNullable	如果类型是可以为 NULL 的类型，则返回 True；否则，返回 False。
Type.IsOpenRecord	返回记录类型是否打开。
Type.ListItem	从列表类型中返回项类型。
Type.NonNullable	从一个类型返回不可为 NULL 的类型。
Type.OpenRecord	返回记录类型的已打开版本（如果已打开，则返回相同类型）
Type.RecordFields	以 [Type = type, Opional = logical] 格式返回一个记录，描述记录类型的字段，并且所返回记录类型的每个字段都有对应的名称和值。
Type.ReplaceFacets	替换类型的 Facet。
Type.ReplaceTableKeys	替换表类型中的键。
Type.TableColumn	返回表中某列的类型。

Type.TableKeys	返回表类型中的键。
Type.TableRow	从表类型返回行类型。
Type.TableSchema	返回一个表，该表包含指定表类型的列的描述（即架构）。
Type.Union	返回类型列表的联合。

Type.AddTableKey

2019/12/3 •

语法

```
Type.AddTableKey(table as type, columns as list, isPrimary as logical) as type
```

关于

向给定表类型添加键。

Type.ClosedRecord

2019/12/3 •

语法

```
Type.ClosedRecord(type as type) as type
```

关于

返回给定 `record` `type` 的已关闭版本(或者如果已关闭，则为同一类型)。

示例 1

创建 `type [ A = number,…]` 的已关闭版本。

```
Type.ClosedRecord(type [ A = number,...])
```

`type [ A = number ]`

Type.Facets

2019/12/3 •

语法

```
Type.Facets(type as type) as record
```

关于

返回一条包含 `type` Facet 的记录

Type.ForFunction

2019/12/3 •

语法

```
Type.ForFunction(signature as record, min as number) as type
```

关于

从 `signature` 创建 `function type`（`ReturnType` 和 `Parameters` 的记录），再创建 `min`（调用函数所需的最少参数数目）。

示例 1

为需要使用 X 数字参数并返回数字的函数创建类型。

```
Type.ForFunction([ReturnType = type number, Parameters = [X = type number]], 1)
```

```
type function (X as number) as number
```

Type.ForRecord

2019/12/3 •

语法

```
Type.ForRecord(fields as record, open as logical) as type
```

关于

返回一个类型，此类型表示对字段具有特定类型约束的记录。

Type.FunctionParameters

2019/12/3 •

语法

```
Type.FunctionParameters(type as type) as record
```

关于

返回一个记录，它的字段值设置为 `type` 的参数名称，值设置为相应的类型。

示例

查找 `(x as number, y as text)` 函数的参数类型。

```
Type.FunctionParameters(type function (x as number, y as text) as any)
```

X	[类型]
Y	[类型]

Type.FunctionRequiredParameters

2019/12/4 •

语法

```
Type.FunctionRequiredParameters(type as type) as number
```

关于

返回一个数字，表明调用函数的输入 `type` 所需参数的最小数量。

示例 1

查找函数所需的参数数量 `(x as number, optional y as text)`。

```
Type.FunctionRequiredParameters(type function (x as number, optional y as text) as any)
```

`1`

Type.FunctionReturn

2019/12/3 •

语法

```
Type.FunctionReturn(type as type) as type
```

关于

返回由函数 `type` 返回的类型。

示例 1

查找 `() as any)` 的返回类型。

```
Type.FunctionReturn(type function () as any)
```

`type any`

Type.Is

2019/12/4 •

语法

```
Type.Is(type1 as type, type2 as type) as logical
```

关于

Type.Is

Type.IsNullable

2019/12/3 •

语法

```
Type.IsNullable(type as type) as logical
```

关于

如果类型是 `nullable` 类型，则返回 `true`；否则返回 `false`。

示例 1

确定 `number` 是否可以为 null。

```
Type.IsNullable(type number)
```

`false`

示例 2

确定 `type nullable number` 是否可以为 null。

```
Type.IsNullable(type nullable number)
```

`true`

Type.IsOpenRecord

2020/4/30 •

语法

```
Type.IsOpenRecord(type as type) as logical
```

关于

返回一个 `logical`，此值指示记录 `type` 是否处于打开状态。

示例 1

确定记录 `type [ A = number, ...]` 是否处于打开状态。

```
Type.IsOpenRecord(type [A = number, ...])
```

`true`

Type.ListItem

2019/12/4 •

语法

```
Type.ListItem(type as type) as type
```

关于

从列表 `type` 中返回项类型。

示例 1

从列表 `{number}` 中查找项类型。

```
Type.ListItem(type {number})
```

`type number`

Type.NonNullable

2019/12/3 •

语法

```
Type.NonNullable(type as type) as type
```

关于

从 `type` 返回非 `nullable` 类型。

示例 1

返回 `type nullable number` 的不可为 null 类型。

```
Type.NonNullable(type nullable number)
```

`type number`

Type.OpenRecord

2019/12/3 •

语法

```
Type.OpenRecord(type as type) as type
```

关于

返回给定 `record` `type`（或同一类型，如果其已打开）的打开版本。

示例 1

创建 `type [ A = number]` 的打开版本。

```
Type.OpenRecord(type [ A = number])
```

`type [ A = number, ... ]`

Type.RecordFields

2020/4/30 •

语法

```
Type.RecordFields(type as type) as record
```

关于

返回描述 `type` 记录字段的记录。返回的记录类型的每个字段都有相应的名称和值，其形式为记录 `[ Type = type, Optional = logical ]`。

示例

查找记录 `[ A = number, optional B = any]` 的名称和值。

```
Type.RecordFields(type [A = number, optional B = any])
```

A	[记录]
B	[记录]

Type.ReplaceFacets

2019/12/3 •

语法

```
Type.ReplaceFacets(type as type, facets as record) as type
```

关于

将 `type` 的 facet 替换为记录 `facets` 中包含的 facet。

Type.ReplaceTableKeys

2019/12/4 •

语法

```
Type.ReplaceTableKeys(tableType as type, keys as list) as type
```

关于

返回一个新的表类型，其中所有键都替换为指定的键列表。

Type.TableColumn

2019/12/4 •

语法

```
Type.TableColumn(tableType as type, column as text) as type
```

关于

返回表类型 `tableType` 中列 `column` 的类型。

Type.TableKeys

2019/12/4 •

语法

```
Type.TableKeys(tableType as type) as list
```

关于

返回给定表类型的可能为空的键列表。

Type.TableRow

2019/12/4 •

语法

```
Type.TableRow(table as type) as type
```

关于

Type.TableRow

Type.TableSchema

2019/12/4 •

语法

```
Type.TableSchema(tableType as type) as table
```

关于

返回描述 `tableType` 列的表。

Type.Union

2019/12/4 •

语法

```
Type.Union(types as list) as type
```

关于

返回 `types` 中类型的联合。

Uri 函数

2020/4/26 •

这些函数创建并操纵 URI 查询字符串。

Uri

"	"
Uri.BuildQueryString	将记录汇编入 URI 查询字符串。
Uri.Combine	根据基部分和相关部分的组合返回 Uri。
Uri.EscapeDataString	根据 RFC 3986 对特殊字符进行编码。
Uri.Parts	返回一个记录值，其字段设置为 Uri 文本值的一部分。

Uri.BuildQueryString

2020/4/30 •

语法

```
Uri.BuildQueryString(query as record) as text
```

关于

将记录 `query` 汇编入 URI 查询字符串，根据需要转义字符。

示例

对包含某些特殊字符的查询字符串进行编码。

```
Uri.BuildQueryString([a = "1", b = "+$"])
```

```
"a=1&b=%2B%24"
```

Uri.Combine

2019/12/3 •

语法

```
Uri.Combine(baseUri as text, relativeUri as text) as text
```

关于

返回一个绝对 URI，这是输入 `baseUri` 和 `relativeUri` 的组合。

Uri.EscapeDataString

2019/12/4 •

语法

```
Uri.EscapeDataString(data as text) as text
```

关于

根据 RFC 3986 的规则对输入 `data` 中的特殊字符进行编码。

示例

对“+money$”中的特殊字符进行编码。

```
Uri.EscapeDataString("+money$")
```

```
"%2Bmoney%24"
```

Uri.Parts

2020/4/30 •

语法

```
Uri.Parts(absoluteUri as text) as record
```

关于

以记录形式返回输入 `absoluteUri` 的组成部分，包含方案、主机、端口、路径、查询、片段、用户名和密码等值。

示例 1

查找绝对 URI“www.adventure-works.com”的组成部分。

```
Uri.Parts("www.adventure-works.com")
```

[illegible]	http
[illegible]	www.adventure-works.com
[illegible]	80
[illegible]	/
[illegible]	[记录]
[illegible]	
[illegible]	
[illegible]	

示例 2

解码百分比编码的字符串。

```
let
    UriUnescapeDataString = (data as text) as text => Uri.Parts("http://contoso?a=" & data)[Query][a]
in
    UriUnescapeDataString("%2Bmoney%24")
```

`"+money$"`

值函数

2020/4/26 •

这些函数对这些值进行评估并执行操作。

值

Value.Compare	返回 1、0 或 -1，具体取决于 value1 是大于、等于还是小于 value2。可提供可选的比较器函数。
Value.Equals	返回两个值是否相等。
Value.NativeQuery	对目标计算查询。
Value.NullableEquals	基于两个值返回一个逻辑值或 NULL。
Value.Type	返回给定值的类型。

算术运算

Value.Add	返回两个值的总和。
Value.Divide	返回将第一个值除以第二个值的结果。
Value.Multiply	返回两个值的乘积。
Value.Subtract	返回两个值的差。

算术参数

Precision.Double	内置算术运算符的可选参数，以指定双精度。
Precision.Decimal	内置算术运算符的可选参数，以指定小数精度。

参数类型

Value.As	Value.As 是与公式语言中的 As 运算符相对应的函数。表达式值 as 类型断言，根据 is 运算符，value 参数的值与类型 as 兼容。如果不兼容，则会出现错误。

Value.Is	Value.Is 是与公式语言中的 Is 运算符相对应的函数。如果值的归属类型与类型兼容，则表达式值 is 类型返回 true；如果值的归属类型与类型不兼容，则返回 false。
Value.ReplaceType	可以使用 Value.ReplaceType 将值归属给类型。Value.ReplaceType 会返回归属类型的新值，如果新类型与该值的本机原始类型不兼容，则会引发错误。特别是，当试图将抽象类型（如 any）归属于某个函数时，该函数会引发错误。替换记录类型时，新类型必须具有相同的字段数，并且新字段将按序号位置而不是按名称替换旧字段。同样，在替换表类型时，新类型必须具有相同的列数，并且新列按序号位置替换旧列。

DirectQueryCapabilities.From	DirectQueryCapabilities.From
Embedded.Value	在嵌入的混合 Web 应用程序中按名称访问值。
Value.Firewall	Value.Firewall
Variable.Value	Variable.Value
SqlExpression.SchemaFrom	SqlExpression.SchemaFrom
SqlExpression.ToExpression	SqlExpression.ToExpression

元数据

Value.Metadata	返回包含输入的元数据的记录。
Value.RemoveMetadata	删除值的元数据，并返回原始值。
Value.ReplaceMetadata	用提供的新元数据记录替换值上的元数据，并返回带有新元数据的原始值。

DirectQueryCapabilities.From

2019/12/3 •

语法

```
DirectQueryCapabilities.From(value as any) as table
```

关于

DirectQueryCapabilities.From

Embedded.Value

2019/12/3 •

语法

```
Embedded.Value(value as any, path as text) as any
```

关于

在嵌入的混合 Web 应用程序中按名称访问值。

Precision.Decimal

2019/12/4 •

关于

内置算术运算符的可选参数，以指定小数精度。

Precision.Double

2019/12/3 •

关于

内置算术运算符的可选参数，以指定双精度。

SqlExpression.SchemaFrom

2019/12/3 •

语法

```
SqlExpression.SchemaFrom(schema as any) as any
```

关于

SqlExpression.SchemaFrom

SqlExpression.ToExpression

2019/12/4 •

语法

```
SqlExpression.ToExpression(sql as text, environment as record) as text
```

关于

SqlExpression.ToExpression

Value.Add

2019/12/4 •

语法

```
Value.Add(value1 as any, value2 as any, optional precision as nullable number) as any
```

关于

返回 `value1` 和 `value2` 的总和。可以指定可选的 `precision` 参数，默认情况下使用 `Precision.Double` 。

Value.As

2019/12/4 •

语法

```
Value.As(value as any, type as type) as any
```

关于

Value.As

Value.Compare

2019/12/3 •

语法

```
Value.Compare(value1 as any, value2 as any, optional precision as nullable number) as number
```

关于

根据第一个值是小于、等于还是大于第二个值，返回 -1、0 或 1。

Value.Divide

2019/12/3 •

语法

```
Value.Divide(value1 as any, value2 as any, optional precision as nullable number) as any
```

关于

返回 `value2` 除以 `value1` 的结果。可以指定可选的 `precision` 参数，默认情况下使用 `Precision.Double` 。

Value.Equals

2019/12/4 •

语法

```
Value.Equals(value1 as any, value2 as any, optional precision as nullable number) as logical
```

关于

如果值 `value1` 等于值 `value2`，则返回 true；否则返回 false。

Value.Firewall

2019/12/3 •

语法

```
Value.Firewall(key as text) as any
```

关于

Value.Firewall

Value.FromText

2019/12/4 •

语法

```
Value.FromText(text as any, optional culture as nullable text) as any
```

关于

从文本表示形式 `text` 解码一个值，并将其解释为具有适当类型的值。 `Value.FromText` 采用文本值并返回数字、逻辑值、NULL 值、时间/日期值、持续时间值或文本值。空文本值将被解释为 NULL 值。

Value.Is

2019/12/3 •

语法

```
Value.Is(value as any, type as type) as logical
```

关于

Value.Is

Value.Metadata

2019/12/3 •

语法

```
Value.Metadata(value as any) as any
```

关于

返回包含输入的元数据的记录。

Value.Multiply

2019/12/4 •

语法

```
Value.Multiply(value1 as any, value2 as any, optional precision as nullable number) as any
```

关于

返回 `value1` 和 `value2` 的乘积。可以指定可选的 `precision` 参数，默认情况下使用 `Precision.Double`。

Value.NativeQuery

2019/12/4 •

语法

```
Value.NativeQuery(target as any, query as text, optional parameters as any, optional options as nullable record) as any
```

关于

使用 `parameters` 中指定的参数和 `options` 中指定的选项，根据 `target` 计算 `query`。

查询的输出由 `target` 定义。

`target` 提供 `query` 描述的操作的上下文。

`query` 描述要根据 `target` 执行的查询。`query` 以特定于 `target` 的方式表示（例如 T-SQL 语句）。

可选的 `parameters` 值可能包含适当的列表或记录，以便提供 `query` 所需的参数值。

可选的 `options` 记录中所包含的选项可能会影响针对 `target` 计算 `query` 的行为。这些选项特定于 `target`。

Value.NullableEquals

2019/12/4 •

语法

```
Value.NullableEquals(value1 as any, value2 as any, optional precision as nullable number) as
nullable logical
```

关于

如果任一参数 `value1` 或 `value2` 为 Null，则返回 Null，否则等同于 Value.Equals。

Value.RemoveMetadata

2019/12/4 •

语法

```
Value.RemoveMetadata(value as any, optional metaValue as any) as any
```

关于

去除元数据的输入。

Value.ReplaceMetadata

2019/12/4 •

语法

```
Value.ReplaceMetadata(value as any, metaValue as any) as any
```

关于

替换输入的元数据信息。

Value.ReplaceType

2019/12/3 •

语法

```
Value.ReplaceType(value as any, type as type) as any
```

关于

Value.ReplaceType

Value.Subtract

2019/12/3 •

语法

```
Value.Subtract(value1 as any, value2 as any, optional precision as nullable number) as any
```

关于

返回 `value1` 和 `value2` 的差值。可以指定可选的 `precision` 参数，默认情况下使用 `Precision.Double`。

Value.Type

2019/12/4 •

语法

```
Value.Type(value as any) as type
```

关于

返回给定值的类型。

Variable.Value

2019/12/3 •

语法

```
Variable.Value(identifier as text) as any
```

关于

Variable.Value