核心语言的 "How to" 主题

使用变量和函数

- 创建变量和函数的定义
- 清除我的定义
- 将函数映射于列表
- 使用纯函数
- 输入函数的范围和选项

使用列表

- 创建列表
- 获取列表中的元素
- 合并与重排列表
- 进行列表运算
- 使用嵌套列表
- 将函数映射于列表
- 制作表格

使用规则

- 创建和使用规则
- 使用规则形式的解
- 将规则多次用于表达式

使用 [illegible] 的语法

正确使用括号和大括号

平衡括号和大括号

使用简写符号

输入函数的范围和选项

使用函数模板

教程专集

- **Core Language**

How to 使用变量和函数

变量和函数是 *Mathematica* 的符号式编程语言中不可或缺的部分. 这些 "How tos" 针对 *Mathematica* 中与变量、函数及函数式编程相关的常见任务给出逐步说明.

创建函数和变量的定义 »

清除我的定义 »

将函数映射于列表 »

使用纯函数 »

输入函数的范围和选项 »

教程

- 符号的值

- 立即定义和延时定义
- 定义变量
- 定义数量值
- 自定义函数
- 定义函数
- 与不同符号相关的定义
- 函数操作
- 模式和变换规则
- 模块和局部变量

参见

Set · **SetDelayed** · **Rule** · **Clear** · **Unset** · **Remove** · **Map** · **Apply** · **Function**

更多关于

- 定义变量和函数
- 函数式编程
- **"How to"** 分类主题

How to 创建变量和函数的定义

Mathematica 具有一组非常全面的函数记号，例如任意变换规则等. 变量也是按这种方式赋值. 一旦您设定了一个变量的值，该变量将变成这个值的符号.

这是一个简单的变换规则. 它指定：只要看到 x，就用**3**代替：

```
In[31]:= x = 3
Out[31]= 3
```

变量 x 的值为**3.**

无论您何时计算一个表达式，都用**3**来代替 x：

```
In[32]:= x^2
Out[32]= 9
```

该规则可以通过定义一个新规则去除：

```
In[33]:= x = y^2
Out[33]= y^2
```

新规则指定：无论何时看到 x，都将它代之以 y^2. 到此为止由于没有与 y 相关的规则，因此它的值就是其自身.

为 y 赋值：

```
In[34]:= y = 4
Out[34]= 4
```

现在如果计算 x，x 的规则指定用 y^2 代替 x，且 y 的规则指定用**4**代替 y，因此结果是 4^2 即**16**：

```
In[35]:= x
Out[35]= 16
```

如果要改变 y 的值，则 x 的值也随之改变：

```
In[36]:= y = 3
Out[36]= 3

In[37]:= x
Out[37]= 9
```

现在为 z 赋上一个值，像这样：

```
In[38]:= z = y^2
Out[38]= 9
```

由于 y 的值已经被赋为**3**，您已定义规则“用**9**代替 z”，而不是“用 y^2 代替 z”. 因此 z 独立于 y：

```
In[39]:= y = 4
Out[39]= 4

In[40]:= z
Out[40]= 9
```

这种情况的出现是因为当一个规则使用 = (Set) 定义时，等号右端在规则定义之前计算.

您也可以使用 := (SetDelayed) 定义规则，像这样：

```
In[41]:= z := y^2
```

当一个规则用 := 定义时，等号右端在规则定义之前不被计算. 因此即使 y 已经有值，这个新规则指定的是：只要看到 z，就将它用 y^2 代替. 因此在这里，z 取决于 y：

```
In[42]:= z
Out[42]= 16

In[43]:= y = 3
Out[43]= 3

In[44]:= z
Out[44]= 9
```

Mathematica 中的函数由行为遵循模式的规则定义. 这是一个简单的模式：

```
In[45]:= f[x_] := x^2
```

f[x_] 是一个模式，x_ 在其中代表任意表达式（在右端通过名称 x 表示）. 规则指定：对于任何表达式的 f ，将其用该表达式的平方代替：

```
In[46]:= f[expr]
Out[46]= expr^2
```

In[58]:= f[●]

Out[58]= ●²

这是一个有两个自变量的函数：

```
In[48]:= g[x_, y_] := f[x] + f[y]

In[49]:= g[3, 4]
Out[49]= 25
```

始终使用 := 定义函数，否则等号右端的变量有可能不代表左端的相关表述，因为它们将在规则定义之前被计算：

```
In[50]:= h[x_, y_] = f[x] + f[y]
Out[50]= 90
```

这种情况发生的原因是 x 为 **9**， y 为 **3.** 规则指定任何与模式 h[x_, y_] 匹配的都用**90**来代替：

In[59]:= h[●, ●]

```
Out[59]= 90
```

教程

- 定义
- 与不同符号相关的定义
- 符号的值
- 定义变量

- 立即定义和延时定义
- 定义数量值

相关链接

- **How to:** 使用变量和函数

参见

Set · **SetDelayed** · **Rule** · **Blank**

更多关于

- 定义变量和函数
- 语言概述
- **"How to"** 分类主题

How to 清除我的定义

当一个符号被赋值，该符号将在整个 *Mathematica* 进程中使用该值. 不再使用的符号在用于新的计算时可能会导致意外的错误，清除您的定义是非常必要的.

为两个符号（x 和 y）赋值并观察它们的和：

```
In[1]:= x = 5;
        y = 7;
        x + y

Out[3]= 12
```

用 Clear 清除 x 和 y 的值：

```
In[4]:= Clear[x, y]
```

可以看到不再有任何值与 x 和 y，它们被当作没有任何定义的符号：

```
In[5]:= Expand[(x + y)^2]

Out[5]= x^2 + 2 x y + y^2
```

这个命令用于清除在当前 ***Mathematica*** 进程中的所有定义：

```
In[6]:= Clear["Global`*"]
```

使用 ClearAll 不仅可以清除符号的值和定义，还可以清除与之关联的属性和信息.

从一个在0 与某个正整数范围内进行输出的函数开始：

```
In[7]:= f[x_] := Table[i, {i, 0, x}]
```

注意当一个列表作为参数给出时，f 将返回一条错误信息：

```
In[8]:= f[{2, 4, 5, 6}]

        Table::iterb : Iterator {i, 0, {2, 4, 5, 6}}
          does not have appropriate bounds. >>

Out[8]= Table[i, {i, 0, {2, 4, 5, 6}}]
```

```
In[1]:= x = 5;
        y = 7;
        x + y

Out[3]= 12
```

用 Clear 清除 x 和 y 的值：

```
In[4]:= Clear[x, y]
```

可以看到不再有任何值与 x 和 y，它们被当作没有任何定义的符号：

```
In[5]:= Expand[(x + y)^2]

Out[5]= x^2 + 2 x y + y^2
```

这个命令用于清除在当前 ***Mathematica*** 进程中的所有定义：

```
In[6]:= Clear["Global`*"]
```

使用 ClearAll 不仅可以清除符号的值和定义，还可以清除与之关联的属性和信息.

从一个在0 与某个正整数范围内进行输出的函数开始：

```
In[7]:= f[x_] := Table[i, {i, 0, x}]
```

注意当一个列表作为参数给出时，f 将返回一条错误信息：

```
In[8]:= f[{2, 4, 5, 6}]

        Table::iterb : Iterator {i, 0, {2, 4, 5, 6}}
          does not have appropriate bounds. >>

Out[8]= Table[i, {i, 0, {2, 4, 5, 6}}]
```

现在设置 f 的属性为 Listable，这样当参数为列表时，f 将映射列表：

```
In[9]:= SetAttributes[f, Listable]
        f[{2, 4, 5, 6}]

Out[10]= {{0, 1, 2}, {0, 1, 2, 3, 4},
          {0, 1, 2, 3, 4, 5}, {0, 1, 2, 3, 4, 5, 6}}
```

注意在清除了 f 之后，属性 Listable 仍然存在：

```
In[11]:= Clear[f]
         ?f
```

```
Global`f
```

```
Attributes[f] = {Listable}
```

使用 ClearAll 清除属性和信息：

```
In[13]:= ClearAll[f]
         ?f
```

```
Global`f
```

使用 ClearAttributes 仅清除函数的属性，它的定义仍然保留.

重新定义 f，并设置其为先前的 Listable：

```
In[15]:= f[x_] := Table[i, {i, 0, x}]
         SetAttributes[f, Listable]
         ?f
```

现在设置 f 的属性为 Listable，这样当参数为列表时，f 将映射列表：

```
In[9]:= SetAttributes[f, Listable]
        f[{2, 4, 5, 6}]

Out[10]= {{0, 1, 2}, {0, 1, 2, 3, 4},
          {0, 1, 2, 3, 4, 5}, {0, 1, 2, 3, 4, 5, 6}}
```

注意在清除了 f 之后，属性 Listable 仍然存在：

```
In[11]:= Clear[f]
         ?f
```

```
Global`f
```

```
Attributes[f] = {Listable}
```

使用 ClearAll 清除属性和信息：

```
In[13]:= ClearAll[f]
         ?f
```

```
Global`f
```

使用 ClearAttributes 仅清除函数的属性，它的定义仍然保留.

重新定义 f，并设置其为先前的 Listable：

```
In[15]:= f[x_] := Table[i, {i, 0, x}]
         SetAttributes[f, Listable]
         ?f
```

```
Global`f
```

```
Attributes[f] = {Listable}
```

```
f[x_] := Table[i, {i, 0, x}]
```

现在使用 ClearAttributes，尽管 f 的定义仍保留着，它已经不具有 Listable 属性：

In[18]:=
```
ClearAttributes[f, Listable]
?f
```

```
Global`f
```

```
f[x_] := Table[i, {i, 0, x}]
```

也可使用 Unset (=.) 清除一个符号的任何定义值：

In[20]:=
```
x = 5
```

Out[20]= 5

In[21]:=
```
x =.
?x
```

```
Global`x
```

Remove 将完全清除一个符号，直到它被再次引用：

In[23]:=
```
Remove[x]
?x
```

Information::notfound : Symbol x not found. ≫

```
Global`f
```

```
Attributes[f] = {Listable}
```

```
f[x_] := Table[i, {i, 0, x}]
```

现在使用 ClearAttributes，尽管 f 的定义仍保留着，它已经不具有 Listable 属性：

In[18]:=
```
ClearAttributes[f, Listable]
?f
```

```
Global`f
```

```
f[x_] := Table[i, {i, 0, x}]
```

也可使用 Unset (=.) 清除一个符号的任何定义值：

In[20]:=
```
x = 5
```

Out[20]= 5

In[21]:=
```
x =.
?x
```

```
Global`x
```

Remove 将完全清除一个符号，直到它被再次引用：

In[23]:=
```
Remove[x]
?x
```

Information::notfound : Symbol x not found. ≫

对 x 的引用使其再次回到该 ***Mathematica*** 进程：

```
In[25]:= x
         ? x

Out[25]= x
```

```
Global`x
```

对于局部定义的符号，其存储的值只在程序的特定部分使用，定义在使用后自动清除. 使用 `Block` 或 `Module` 可以对符号进行局部定义.

教程

- 属性
- 与不同符号相关的定义
- 定义
- 定义变量
- 定义数量值
- 自定义函数
- 定义函数
- 模块和局部变量
- 立即定义和延时定义

相关链接

- **How to:** 创建变量和函数的定义
- **How to:** 使用变量和函数

参见

Clear · **ClearAll** · **ClearAttributes** · **Unset** · **Remove** · **Block** · **Module**

更多关于

- 定义变量和函数
- 赋值
- **"How to"** 分类主题

How to 将函数映射于列表

Mathematica 包含许多功能强大的操作，用于使用列表. 常常需要将函数映射到每个列表的每个单个元素. 对于具有可列表性的函数，在默认情况下就能做到，而对于具有不可列表性的函数，您可以使用 Map 来执行此操作.

首先设置一个由**1**至**5**的整数组成的列表：

```
In[293]:= mylist = Range[5]
Out[293]= {1, 2, 3, 4, 5}
```

您可以使用 Map 将函数映射到列表的每个元素；下面的例子使用的是一个未定义的函数 f：

```
In[294]:= Map[f, mylist]
Out[294]= {f[1], f[2], f[3], f[4], f[5]}
```

可以使用 /@ 作为 Map 的简写形式（这个命令与上例中的相同）：

```
In[295]:= f /@ mylist
Out[295]= {f[1], f[2], f[3], f[4], f[5]}
```

大多数数学函数具有 Listable 属性，表示它们将自动映射于列表：

```
In[296]:= Attributes[Sin]
Out[296]= {Listable, NumericFunction, Protected}

In[297]:= Sin[mylist]
Out[297]= {Sin[1], Sin[2], Sin[3], Sin[4], Sin[5]}
```

如果函数非 Listable，您可以使用 Map. 设置由5个2×2矩阵组成的列表：

```
In[298]:= matrices = RandomInteger[1, {5, 2, 2}]
Out[298]= {{{0, 0}, {0, 1}},
           {{1, 1}, {1, 1}}, {{0, 1}, {1, 0}},
           {{1, 0}, {0, 0}}, {{1, 1}, {0, 1}}}
```

使用 Map 将 MatrixForm 映射于列表，以数学符号形式查看每一个矩阵：

```
In[299]:= MatrixForm /@ matrices
```

Out[299]=

$$\left\{\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\right\}$$

现在用 Map 计算列表中每个矩阵的特征值：

```
In[300]:= Eigenvalues /@ matrices
Out[300]= {{1, 0}, {2, 0}, {-1, 1}, {1, 0}, {1, 1}}
```

Map 不仅可以操作列表. 它也可以用于任何表达式：

```
In[301]:= Map[f, a + b + c + d]
Out[301]= f[a] + f[b] + f[c] + f[d]
```

Apply 是另一个函数编程操作. 它用于替换一个表达式的头：

您可以通过两个未定义的函数 f 和 g 来看这是怎样作用的：

```
In[302]:= Apply[f, g[a, b, c, d]]
Out[302]= f[a, b, c, d]
```

Apply 用 @@ 作为简写形式（这个命令与上例中的相同）：

```
In[303]:= f @@ g[a, b, c, d]
Out[303]= f[a, b, c, d]
```

在 ***Mathematica*** 中公用表达式以 StandardForm 形式显示，而

```
In[299]:= MatrixForm /@ matrices
```

Out[299]=

$$\left\{\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\right\}$$

现在用 Map 计算列表中每个矩阵的特征值：

```
In[300]:= Eigenvalues /@ matrices
Out[300]= {{1, 0}, {2, 0}, {-1, 1}, {1, 0}, {1, 1}}
```

Map 不仅可以操作列表. 它也可以用于任何表达式：

```
In[301]:= Map[f, a + b + c + d]
Out[301]= f[a] + f[b] + f[c] + f[d]
```

Apply 是另一个函数编程操作. 它用于替换一个表达式的头：

您可以通过两个未定义的函数 f 和 g 来看这是怎样作用的：

```
In[302]:= Apply[f, g[a, b, c, d]]
Out[302]= f[a, b, c, d]
```

Apply 用 @@ 作为简写形式（这个命令与上例中的相同）：

```
In[303]:= f @@ g[a, b, c, d]
Out[303]= f[a, b, c, d]
```

在 ***Mathematica*** 中公用表达式以 StandardForm 形式显示，而

```
In[304]:= {Plus[a, b, c], Times[a, b, c], List[a, b, c]}
Out[304]= {a + b + c, a b c, {a, b, c}}
```

例如，这里将加和变成乘积：

```
In[305]:= Apply[Times, a + b + c]
Out[305]= a b c
```

当您想要将列表中的元素转换为函数的参数时，Apply 很有用.

创建一个由5个有序对 {a, b} 组成的列表：

```
In[306]:= pairs = RandomInteger[{1, 10}, {5, 2}]
Out[306]= {{8, 7}, {5, 9}, {6, 8}, {7, 1}, {6, 7}}
```

Mod 求有序对中第一个数除以第二个数得到的余数：

```
In[307]:= Mod[10, 4]
Out[307]= 2
```

将 Mod 应用于全部有序对，需要在列表的第一层作用（由 {1} 指定）：

```
In[308]:= Apply[Mod, pairs, {1}]
Out[308]= {1, 5, 6, 0, 6}
```

可以用 @@@ 作为简写符号应用于第一层：

```
In[304]:= {Plus[a, b, c], Times[a, b, c], List[a, b, c]}
Out[304]= {a + b + c, a b c, {a, b, c}}
```

例如，这里将加和变成乘积：

```
In[305]:= Apply[Times, a + b + c]
Out[305]= a b c
```

当您想要将列表中的元素转换为函数的参数时，Apply 很有用.

创建一个由5个有序对 {a, b} 组成的列表：

```
In[306]:= pairs = RandomInteger[{1, 10}, {5, 2}]
Out[306]= {{8, 7}, {5, 9}, {6, 8}, {7, 1}, {6, 7}}
```

Mod 求有序对中第一个数除以第二个数得到的余数：

```
In[307]:= Mod[10, 4]
Out[307]= 2
```

将 Mod 应用于全部有序对，需要在列表的第一层作用（由 {1} 指定）：

```
In[308]:= Apply[Mod, pairs, {1}]
Out[308]= {1, 5, 6, 0, 6}
```

可以用 @@@ 作为简写符号应用于第一层：

```
In[309]:= Mod @@@ pairs
Out[309]= {1, 5, 6, 0, 6}
```

这是完成同样任务的另一种方法，方法是利用带有 Function 的纯函数：

```
In[310]:= Map[Function[w, Apply[Mod, w]], pairs]
Out[310]= {1, 5, 6, 0, 6}
```

这里使用 Function 的简写形式：

```
In[311]:= Map[Apply[Mod, #] &, pairs]
Out[311]= {1, 5, 6, 0, 6}
```

教程

- 函数作用于表达式的部分项
- 函数操作
- 表达式的层次结构
- 构建列表
- 嵌套列表

相关链接

- **How to:** 使用变量和函数
- **How to:** 使用列表

参见

更多关于

- 函数作用于列表
- 函数式编程
- 大型数组的处理
- **"How to"** 分类主题

How to 使用纯函数

Mathematica 功能如此强大的体现之一是，用户能够自定义并使用自己的函数. 如果必须对任何一种无论多小的运算所用的函数显式命名，往往将很不方便. 在 *Mathematica* 中，您可以通过声明内联函数（称作纯函数）来避开这一问题.

定义纯函数最显见的方法是使用 `Function`. 第一个参数是一个参数列表，第二个参数是一个函数. 该函数将它的两个参数相加：

```
In[1]:= f = Function[{x, y}, x + y]
Out[1]= Function[{x, y}, x + y]

In[2]:= f[3, 4]
Out[2]= 7
```

您不必给函数命名就可使用它：

```
In[3]:= Function[{x, y}, x + y][3, 4]
Out[3]= 7
```

一种常见的简写符号是，使用一个 & 标记在纯函数的末端，参数位置用 #1、#2 等指定：

```
In[4]:= g = (#1 + #2) &
Out[4]= #1 + #2 &

In[5]:= g[3, 4]
Out[5]= 7
```

纯函数的优点是它不要求有单独的定义或名称：

```
In[6]:= (#1 + #2) &[3, 4]
Out[6]= 7
```

如果纯函数只有一个参数，可以使用 # 而不是 #1. 该函数对其参数进行平方：

```
In[7]:= #^2 &[3]
Out[7]= 9
```

纯函数在与 Map 共同使用时功能会变得相当强大.

这里将一个复数列表变成一个有序对的列表：

```
In[3]:= Function[{x, y}, x + y][3, 4]
Out[3]= 7
```

一种常见的简写符号是，使用一个 & 标记在纯函数的末端，参数位置用 #1、#2 等指定：

```
In[4]:= g = (#1 + #2) &
Out[4]= #1 + #2 &

In[5]:= g[3, 4]
Out[5]= 7
```

纯函数的优点是它不要求有单独的定义或名称：

```
In[6]:= (#1 + #2) &[3, 4]
Out[6]= 7
```

如果纯函数只有一个参数，可以使用 # 而不是 #1. 该函数对其参数进行平方：

```
In[7]:= #^2 &[3]
Out[7]= 9
```

纯函数在与 Map 共同使用时功能会变得相当强大.

这里将一个复数列表变成一个有序对的列表：

```
In[8]:= pairs = Map[{Re[#], Im[#]} &, {I, 1 + I, 2 + 3 I}
Out[8]= {{0, 1}, {1, 1}, {2, 3}}
```

这里将有序对重新变成复数：

```
In[9]:= Map[#[[1]] + I #[[2]] &, pairs]
Out[9]= {ⅈ, 1 + ⅈ, 2 + 3 ⅈ}
```

可用 /@ 作为 Map 的简写符号. 使用纯函数创建可单击式按钮的一个列表：

```
In[10]:= Button[#, MessageDialog[StringJoin[
           ToString[#], " clicked!"]]] & /@ {1, 2, 3}
```

Out[10]= {1, 2, 3}

教程

- 纯函数
- 函数操作
- 纯函数和规则中的变量

相关链接

- **How to:** 使用变量和函数

参见

更多关于

- **"How to"** 分类主题

How to 输入函数的范围和选项

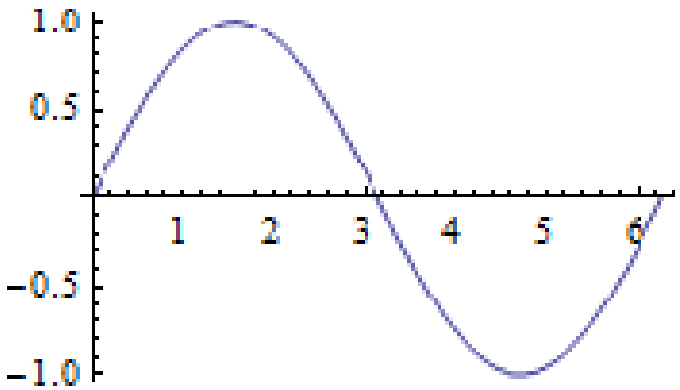
一个 *Mathematica* 内置函数所执行的主要表达式或对象作为函数的第一个参数给出. 作为语法的一部分， *Mathematica* 内置函数可以有多个自变量，这可能要求或可能成为该函数的推广或延伸. 参数之后是选项，用于进一步扩展以控制函数的行为.

Mathematica 中的许多可视化函数要求用户输入绘图界限，作为绘图函数的第二个参数.

这里，`Sin[x]` 是 `Plot` 中的第一个参数，第二个参数 `{x, 0, 2 π}` 给出变量以及图形范围：

```
In[7]:= Plot[Sin[x], {x, 0, 2 π}]
```

Out[7]=



如果使用 Plot 时不指定范围，*Mathematica* 会产生一则错误提示信息：

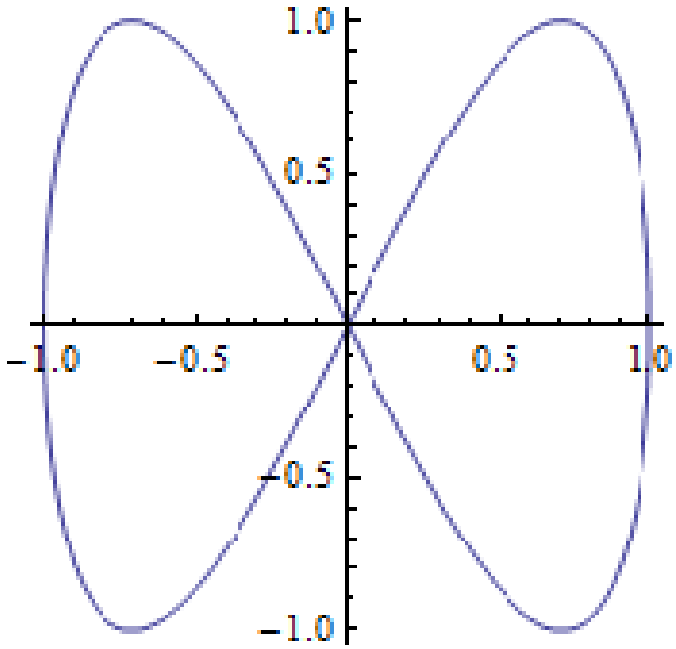
```
In[2]:= Plot[Sin[x], {x}]

        Plot::pllim :
         Range specification {x} is not of the form
             {x, xmin, xmax}. >>

Out[2]= Plot[Sin[x], {x}]
```

其它可视化函数具有类似的关于绘图范围的要求.

这里，一条参数曲线在 0 到 2 π 之间绘制. 这些界限作为 ParametricPlot 的第二参数指定：

```
In[5]:= ParametricPlot[{Sin[u], Sin[2 u]}, {u, 0, 2 π}]
```

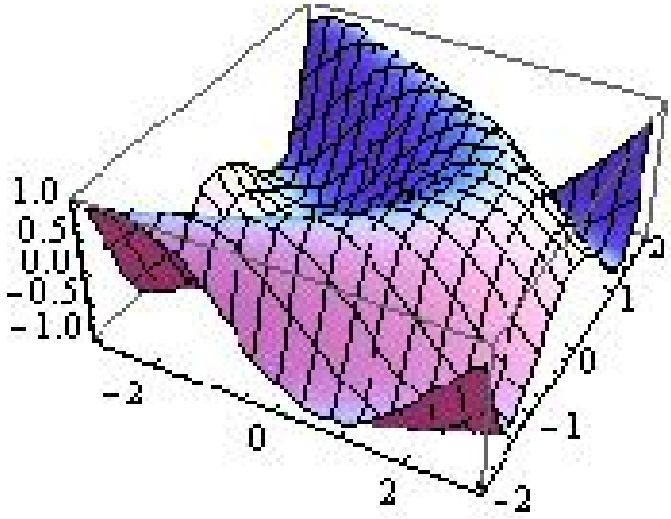
Out[5]=



在绘制两个变量的函数时，每个变量的范围分别作为第二个和第三个参数输入：

```
In[6]:= Plot3D[Sin[x + y^2], {x, -3, 3}, {y, -2, 2}]
```

Out[6]=



要了解一个可视化函数的确切语法，请参见参考资料中心的相关页面. 关于如何查找参考资料中心的函数页面，请参见**How to:** 查找函数的相关信息.

尽管 ***Mathematica*** 内置函数的缺省行为适用于大多数场合，您可以通过函数选项来对其行为进行准确控制.

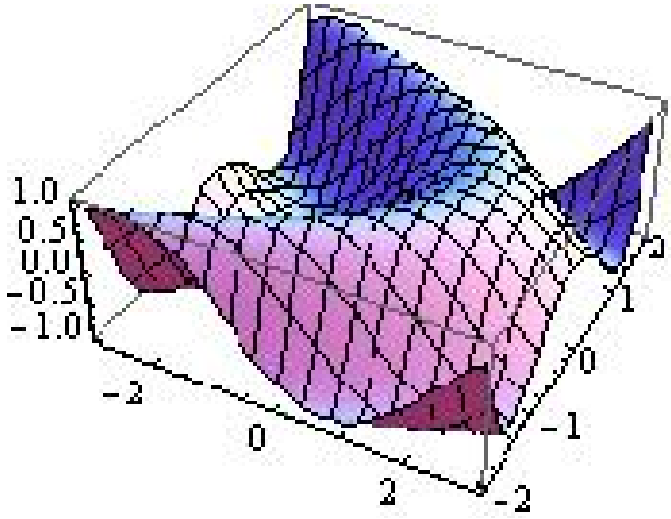
选项使用规则输入. 一个规则的缩写形式使用的是一个右箭头，这可以通过键入 -> （**-** 与**>**之间无空格）得到. 在进一步键入时，***Mathematica*** 前端自动将 -> 转换为 →. 每个符号都是 Rule 的一种缩写形式.

在输入函数选项时，选项在规则的左边，其设置在右边. 选项永远位于函数要求的所有参数之后.

将 Plot 与 Filling 联合使用创建一个填充图形：

```
In[6]:= Plot3D[Sin[x + y^2], {x, -3, 3}, {y, -2, 2}]
```

Out[6]=



要了解一个可视化函数的确切语法，请参见参考资料中心的相关页面. 关于如何查找参考资料中心的函数页面，请参见**How to:** 查找函数的相关信息.

尽管 ***Mathematica*** 内置函数的缺省行为适用于大多数场合，您可以通过函数选项来对其行为进行准确控制.

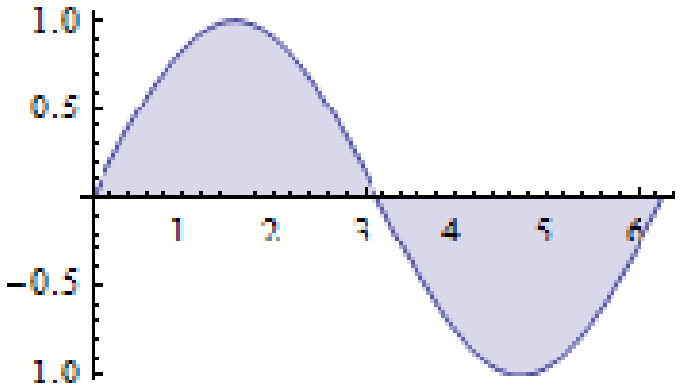
选项使用规则输入. 一个规则的缩写形式使用的是一个右箭头，这可以通过键入 -> （**-** 与**>**之间无空格）得到. 在进一步键入时，***Mathematica*** 前端自动将 -> 转换为 →. 每个符号都是 Rule 的一种缩写形式.

在输入函数选项时，选项在规则的左边，其设置在右边. 选项永远位于函数要求的所有参数之后.

将 Plot 与 Filling 联合使用创建一个填充图形：

In[8]:=
```
Plot[Sin[x], {x, 0, 2 π}, Filling → Axis]
```

Out[8]=



从 ***Mathematica*** 参考资料范例数据中导入某些数据，并将其在一个 `Grid` 中显示：

In[9]:=
```
data =
  Import["ExampleData/classification.tsv"];
```

In[10]:=
```
Grid[data]
```

Out[10]=

Rank	Fruit Fly	Human	Pea	E.coli
Domain	Eukaryota	Eukaryota	Eukaryota	Bacteri
Kingdom	Animalia	Animalia	Plantae	Monera
Phylum	Arthropoda	Chordata	Magnoliophyta	Proteobacteria
Class	Insecta	Mammalia	Magnoliopsida	Proteobacteria
Order	Diptera	Primates	Fabales	Enterobacteriales
Family	Drosophilidae	Hominidae	Fabaceae	Enterobacteriaceae
Genus	Drosophila	Homo	Pisum	Escherichia
Species	D.melanogaster	H.sapiens	P.sativum	E.coli

将 Grid 与 Frame 选项联合使用：

In[11]:=
```
Grid[data, Frame → All]
```

Out[11]=

Rank	Fruit Fly	Human	Pea	E.coli
Domain	Eukaryota	Eukaryota	Eukaryota	Bacteria
Kingdom	Animalia	Animalia	Plantae	Monera
Phylum	Arthropoda	Chordata	Magnoliophyta	Proteobacteria
Class	Insecta	Mammalia	Magnoliopsida	Proteobacteria
Order	Diptera	Primates	Fabales	Enterobacteriales
Family	Drosophilidae	Hominidae	Fabaceae	Enterobacteriaceae
Genus	Drosophila	Homo	Pisum	Escherichia
Species	D.melanogaster	H.sapiens	P.sativum	E.coli

您也可以一起使用多个选项，与参数不同，这些选项可以任意次序排列：

```
Grid[data, Alignment → Left, Spacings → {2, 1},
 Frame → All, ItemStyle → "Text",
 Background → {{Gray, None}, {LightGray, None}}]
```

Rank	Fruit Fly	Human	Pea	E.coli
Domain	Eukaryota	Eukaryota	Eukaryota	Bacteria
Kingdom	Animalia	Animalia	Plantae	Monera
Phylum	Arthropoda	Chordata	Magnoliophyta	Proteobacteria
Class	Insecta	Mammalia	Magnoliopsida	Proteobacteria
Order	Diptera	Primates	Fabales	Enterobacterials
Family	Drosophilidae	Hominidae	Fabaceae	Enterobacteriaceae

```
Grid[data, Alignment → Left, Spacings → {2, 1},
 Frame → All, ItemStyle → "Text",
 Background → {{Gray, None}, {LightGray, None}}]
```

Rank	Fruit Fly	Human	Pea	E.coli
Domain	Eukaryota	Eukaryota	Eukaryota	Bacteria
Kingdom	Animalia	Animalia	Plantae	Monera
Phylum	Arthropoda	Chordata	Magnoliophyta	Proteobacteria
Class	Insecta	Mammalia	Magnoliopsida	Proteobacteria
Order	Diptera	Primates	Fabales	Enterobacterials
Family	Drosophilidae	Hominidae	Fabaceae	Enterobacteriaceae

Genus	Drosophila	Homo	Pisum	Escherichia
Species	D.melanogaster	H.sapiens	P.sativum	E.coli

可用选项取决于所使用的函数. 要知道一个函数可与哪些选项结合使用，请参见 **How to:** 查找可用选项.

教程

- 一些普遍的记号和传统表示
- ***Mathematica*** 语言的语法
- 输入语法
- ***Mathematica*** 的四种括号
- 处理选项
- 从 ***Mathematica*** 中获取信息

相关链接

- **How to:** 查找可用选项
- **How to:** 查找函数的相关信息
- **How to:** 使用 ***Mathematica*** 的语法

参见

更多关于

- 语法
- 选项管理
- 图形选项和样式
- **"How to"** 分类主题

How to 使用列表

列表是 *Mathematica* 符号式语言的核心. 这些 "How tos" 针对 *Mathematica* 中与创建与操作列表相关的常见任务给出逐步说明.

创建列表 »

获取列表中的元素 »

合并与重排列表 »

进行列表运算 »

使用嵌套列表 »

将函数映射于列表 »

制作表格 »

教程

- 列表
- 列表的组合
- 重排列表
- 通过索引操作列表
- 处理列表元素
- 列表元素的分组和合并
- 添加、删除和修改列表元素
- 嵌套列表
- 嵌套列表的重排
- 向量和矩阵
- 标量、向量和矩阵的运算
- 函数操作

相关链接

- **How to:** 进行线性代数计算

参见

List · Table · Range · Part · Sort · Riffle · Partition · Map · Apply · Prepend · Append · ReplacePart

更多关于

- 列表操作
- 列表元素

- 大型数组的处理
- **"How to"** 分类主题

How to 创建列表

列表是 *Mathematica* 中非常重要和常规的结构. 它们允许您将任何类型对象的组合当作一个单一实体来处理. 有多种方式来构建它们.

使用缩写符号 {} 产生一个列表:

```
In[1]:= {2, 3, 5, 6}
Out[1]= {2, 3, 5, 6}
```

或使用 List, 它将自动变为 {}:

```
In[2]:= List[2, 3, 5, 6]
Out[2]= {2, 3, 5, 6}
```

使用一个参数的 Range 来创建一个从**1**开始的整数列表:

```
In[3]:= Range[10]
Out[3]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

或使用两个参数的 Range 来创建从一个从更大的值开始的整数列表:

```
In[4]:= Range[5, 10]
Out[4]= {5, 6, 7, 8, 9, 10}
```

使用三个参数可以得到相邻元素间隔不为**1**的列表：

```
In[5]:= Range[5, 10, 1 / 2]
```

$$\text{Out[5]= } \left\{5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, \frac{17}{2}, 9, \frac{19}{2}, 10\right\}$$

对列表中每个元素进行平方：

```
In[6]:= Range[10]^2
Out[6]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

或使用 Table 创建这头**10**个平方：

```
In[7]:= Table[i^2, {i, 10}]
Out[7]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

正如 Range 一样，Table 可以从更大的值开始，或进行任意数量的跳跃：

```
In[8]:= Table[i^2, {i, 5, 10, 1 / 2}]
```

$$\text{Out[8]= } \left\{25, \frac{121}{4}, 36, \frac{169}{4}, 49, \frac{225}{4}, 64, \frac{289}{4}, 81, \frac{361}{4}, 100\right\}$$

使用 NestList 将 f 应用于 x **0**到**3**次的结果创建一个列表：

```
In[9]:= NestList[f, x, 3]
Out[9]= {x, f[x], f[f[x]], f[f[f[x]]]}
```

用 Array 创建一个长度为4的列表，其中元素为 f[i]：

```
In[10]:= Array[f, 4]
Out[10]= {f[1], f[2], f[3], f[4]}
```

这里给出一个 3×2 数组：

```
In[11]:= Array[f, {3, 2}]
Out[11]= {{f[1, 1], f[1, 2]},
          {f[2, 1], f[2, 2]}, {f[3, 1], f[3, 2]}}
```

用 List 创建字符串列表：

```
In[12]:= List[{"a", "b", "c"}, {"you", "are", "good"}]
Out[12]= {{a, b, c}, {you, are, good}}
```

Mathematica 的矩阵是列表的列表.

使用 RandomInteger 创建一个由0到10之间的随机整数（由 m 存储）组成的 4×4 矩阵：

```
In[13]:= m = RandomInteger[10, {4, 4}]
Out[13]= {{5, 0, 2, 5}, {3, 2, 8, 6},
          {9, 9, 8, 0}, {2, 2, 3, 4}}
```

```
In[9]:= NestList[f, x, 3]
Out[9]= {x, f[x], f[f[x]], f[f[f[x]]]}
```

用 Array 创建一个长度为4的列表，其中元素为 f[i]：

```
In[10]:= Array[f, 4]
Out[10]= {f[1], f[2], f[3], f[4]}
```

这里给出一个 3×2 数组：

```
In[11]:= Array[f, {3, 2}]
Out[11]= {{f[1, 1], f[1, 2]},
          {f[2, 1], f[2, 2]}, {f[3, 1], f[3, 2]}}
```

用 List 创建字符串列表：

```
In[12]:= List[{"a", "b", "c"}, {"you", "are", "good"}]
Out[12]= {{a, b, c}, {you, are, good}}
```

Mathematica 的矩阵是列表的列表.

使用 RandomInteger 创建一个由0到10之间的随机整数（由 m 存储）组成的 4×4 矩阵：

```
In[13]:= m = RandomInteger[10, {4, 4}]
Out[13]= {{5, 0, 2, 5}, {3, 2, 8, 6},
          {9, 9, 8, 0}, {2, 2, 3, 4}}
```

In[14]:=
```
MatrixForm[m]
```

t[14]//MatrixForm=

$$\begin{pmatrix} 5 & 0 & 2 & 5 \\ 3 & 2 & 8 & 6 \\ 9 & 9 & 8 & 0 \\ 2 & 2 & 3 & 4 \end{pmatrix}$$

您可以将函数应用于列表.

您可以直接将数学函数应用于一个列表：

In[15]:=
```
Sqrt[{1, 2, 3, 4}]
```

Out[15]= $\{1, \sqrt{2}, \sqrt{3}, 2\}$

数学函数继续向纵深发展：

In[16]:=
```
1 + {{a}, {a, b}, {a, b, c}}^2
```

Out[16]= $\{\{1 + a^2\}, \{1 + a^2, 1 + b^2\}, \{1 + a^2, 1 + b^2, 1 + c^2\}\}$

一个函数给出数值结果：

In[17]:=
```
Max[{1, 2, 3, 4}]
```

Out[17]= 4

Length 给出列表长度：

In[18]:=
```
Length[{a, b, c}]
```

Out[18]= 3

In[14]:=
```
MatrixForm[m]
```

$$\begin{pmatrix} 5 & 0 & 2 & 5 \\ 3 & 2 & 8 & 6 \\ 9 & 9 & 8 & 0 \\ 2 & 2 & 3 & 4 \end{pmatrix}$$

您可以将函数应用于列表.

您可以直接将数学函数应用于一个列表：

In[15]:=
```
Sqrt[{1, 2, 3, 4}]
```

Out[15]= $\{1, \sqrt{2}, \sqrt{3}, 2\}$

数学函数继续向纵深发展：

In[16]:=
```
1 + {{a}, {a, b}, {a, b, c}}^2
```

Out[16]= $\{\{1 + a^2\}, \{1 + a^2, 1 + b^2\}, \{1 + a^2, 1 + b^2, 1 + c^2\}\}$

一个函数给出数值结果：

In[17]:=
```
Max[{1, 2, 3, 4}]
```

Out[17]= 4

Length 给出列表长度：

In[18]:=
```
Length[{a, b, c}]
```

Out[18]= 3

用 Map 将一个函数应用于一个列表中的元素（不仅限于数学函数）：

```
In[19]:= Map[f, {a, b, c}]
Out[19]= {f[a], f[b], f[c]}
```

这里使用 Map 将 Length 应用于各个子列表：

```
In[20]:= Map[Length, {{a}, {a, b}, {a, b, c}}]
Out[20]= {1, 2, 3}
```

类似地，这里求出每个子列表的最大值：

```
In[21]:= Map[Max, {{a}, {a, b}, {a, b, c}}]
Out[21]= {a, Max[a, b], Max[a, b, c]}
```

教程

- 构建列表
- 处理列表元素
- 列表的组合
- 重排列表
- 通过索引操作列表
- 向量和矩阵

相关链接

- **How to:** 使用列表
- **How to:** 进行线性代数计算

参见

List · Table · Range · NestList · Array

更多关于

- 语言概述
- ***Mathematica*** 语法
- **"How to"** 分类主题

How to 获取列表中的元素

列表是 *Mathematica* 中非常重要的结构. 列表使您可以将任何类型的对象集合作为一个单一实体来处理. 有时您需要从一个列表中择选或提取出个别元素或元素组.

设置一个由前**10**个平方组成的列表（存为 v ）：

```
In[256]:= v = Range[10]^2
Out[256]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

使用 Part 取出列表的第三个元素：

```
In[257]:= Part[v, 3]
Out[257]= 9
```

```
In[258]:= v[[3]]
Out[258]= 9
```

将 ;;（Span 的简写形式）与 Part 合用，取出从**1**到**5**的元素：

```
In[259]:= v[[1 ;; 5]]
Out[259]= {1, 4, 9, 16, 25}
```

取出列表中第**5**至**8**个元素：

```
In[260]:= v[[5 ;; 8]]
Out[260]= {25, 36, 49, 64}
```

取出最后七个元素（负号表示从后向前数）：

```
In[261]:= v[[-7 ;;]]
Out[261]= {16, 25, 36, 49, 64, 81, 100}
```

Mathematica 中的矩阵是等长度列表的列表. 您可以像操作列表一样从矩阵中取出元素.

设置一个**5×5**整数矩阵：

```
In[262]:= m = Partition[Range[25]^2, 5]
Out[262]= {{1, 4, 9, 16, 25}, {36, 49, 64, 81, 100},
           {121, 144, 169, 196, 225},
           {256, 289, 324, 361, 400},
           {441, 484, 529, 576, 625}}
```

```
In[258]:= v[[3]]
Out[258]= 9
```

将 ;;（Span 的简写形式）与 Part 合用，取出从**1**到**5**的元素：

```
In[259]:= v[[1 ;; 5]]
Out[259]= {1, 4, 9, 16, 25}
```

取出列表中第**5**至**8**个元素：

```
In[260]:= v[[5 ;; 8]]
Out[260]= {25, 36, 49, 64}
```

取出最后七个元素（负号表示从后向前数）：

```
In[261]:= v[[-7 ;;]]
Out[261]= {16, 25, 36, 49, 64, 81, 100}
```

Mathematica 中的矩阵是等长度列表的列表. 您可以像操作列表一样从矩阵中取出元素.

设置一个**5×5**整数矩阵：

```
In[262]:= m = Partition[Range[25]^2, 5]
Out[262]= {{1, 4, 9, 16, 25}, {36, 49, 64, 81, 100},
           {121, 144, 169, 196, 225},
           {256, 289, 324, 361, 400},
           {441, 484, 529, 576, 625}}
```

使用 MatrixForm 使之以矩阵形式显示：

```
In[263]:= MatrixForm[m]
```

Out[263]//MatrixForm=

$$\begin{pmatrix} 1 & 4 & 9 & 16 & 25 \\ 36 & 49 & 64 & 81 & 100 \\ 121 & 144 & 169 & 196 & 225 \\ 256 & 289 & 324 & 361 & 400 \\ 441 & 484 & 529 & 576 & 625 \end{pmatrix}$$

矩阵的第一个部分是第一行：

```
In[264]:= m[[1]]
Out[264]= {1, 4, 9, 16, 25}
```

教程

- 获得列表的部分元素
- 添加、删除和修改列表元素
- 处理列表元素
- 检测和搜索列表元素
- 向量和矩阵

相关链接

- **How to:** 使用列表

参见

更多关于

- 列表元素
- 重排列与重构列表
- 表达式的子集
- 子矩阵
- **"How to"** 分类主题

How to 合并与重排列表

Mathematica 提供了一套完整的数据操纵语言，可以极其灵活地重新排列具有任何结构、任何数量元素的列表.

设置一个由六个整数组成的、具有某些重复值的列表（存为 v）：

```
In[1]:= v = {3, 1, 3, 2, 5, 4}
Out[1]= {3, 1, 3, 2, 5, 4}
```

用 Sort 将列表 v 中的元素排序：

```
In[2]:= Sort[v]
Out[2]= {1, 2, 3, 3, 4, 5}
```

使用 Union 对 v 排序，并去除所有重复值

```
In[3]:= Union[v]
Out[3]= {1, 2, 3, 4, 5}
```

应注意除非您重新定义 v，否则它仍保留原值：

```
In[4]:= v
Out[4]= {3, 1, 3, 2, 5, 4}
```

重新定义 v：

```
In[5]:= v = Union[v]
Out[5]= {1, 2, 3, 4, 5}

In[6]:= v
Out[6]= {1, 2, 3, 4, 5}
```

对 v 的元素进行反向排列：

```
In[7]:= Reverse[v]
Out[7]= {5, 4, 3, 2, 1}
```

将 v 的元素向左轮换两位：

```
In[8]:= RotateLeft[v, 2]
Out[8]= {3, 4, 5, 1, 2}
```

将 v 的元素向右轮换两位：

```
In[9]:= RotateRight[v, 2]
Out[9]= {4, 5, 1, 2, 3}
```

用 x 对 v 进行左填充，得到一个长度为**10**的列表：

```
In[10]:= PadLeft[v, 10, x]
Out[10]= {x, x, x, x, x, 1, 2, 3, 4, 5}
```

将列表 v 拆分成长度为**2**的子列表：

```
In[11]:= Partition[v, 2]
Out[11]= {{1, 2}, {3, 4}}
```

将列表 v 拆分成偏移为**1**、长度为**2**的子列表：

```
In[12]:= Partition[v, 2, 1]
Out[12]= {{1, 2}, {2, 3}, {3, 4}, {4, 5}}
```

将列表中的相同元素进行组合：

```
In[13]:= Split[{1, 4, 1, 1, 1, 2, 2, 3, 3}]
Out[13]= {{1}, {4}, {1, 1, 1}, {2, 2}, {3, 3}}
```

Flatten 删除嵌套列表中的内部大括号：

```
In[14]:= Flatten[{{a, b}, {c, {d}, e}, {f, {g, h}}}]
Out[14]= {a, b, c, d, e, f, g, h}
```

```
In[15]:= Join[{a, b}, {c, d, e}, {f, g, h}]
Out[15]= {a, b, c, d, e, f, g, h}
```

教程

- 列表的组合
- 重排列表
- 通过索引操作列表
- 嵌套列表
- 列表元素的分组和合并
- 添加、删除和修改列表元素

相关链接

- **How to:** 使用列表
- **How to:** 进行线性代数计算

参见

Sort ▪ **Join** ▪ **Union** ▪ **Reverse** ▪ **Split** ▪ **Partition** ▪ **Flatten** ▪ **RotateLeft** ▪ **RotateRight** ▪ **PadLeft** ▪ **PadRight**

更多关于

- **"How to"** 分类主题

How to 进行列表运算

列表是 *Mathematica* 的中心结构，用于表示类集、数组、集合以及各种序列. *Mathematica* 中有上千个内置函数可以直接进行列表运算，是实现互操作性的强劲载体.

设置一个列表，由**0**到**10**之间的**5**个随机整数组成（存储为 v）：

```
In[45]:= v = RandomInteger[10, {5}]
Out[45]= {8, 5, 2, 5, 9}
```

用 Max 求 v 的最大元素：

```
In[46]:= Max[v]
Out[46]= 9
```

设置一个**4×3**矩阵，由**0**到**10**之间的随机整数组成：

```
In[47]:= m = RandomInteger[10, {4, 3}]
Out[47]= {{1, 5, 6}, {1, 3, 10}, {3, 8, 4}, {6, 3, 9}}
```

使用 Map 将 Max 应用于 m 顶层的各个元素：

```
In[48]:= Map[Max, m, 2]
Out[48]= {6, 10, 8, 9}
```

您可以将两个等长列表的元素对应相加：

How to 进行列表运算

列表是 *Mathematica* 的中心结构，用于表示类集、数组、集合以及各种序列. *Mathematica* 中有上千个内置函数可以直接进行列表运算，是实现互操作性的强劲载体.

设置一个列表，由**0**到**10**之间的**5**个随机整数组成（存储为 v）：

```
In[45]:= v = RandomInteger[10, {5}]
Out[45]= {8, 5, 2, 5, 9}
```

用 Max 求 v 的最大元素：

```
In[46]:= Max[v]
Out[46]= 9
```

设置一个**4×3**矩阵，由**0**到**10**之间的随机整数组成：

```
In[47]:= m = RandomInteger[10, {4, 3}]
Out[47]= {{1, 5, 6}, {1, 3, 10}, {3, 8, 4}, {6, 3, 9}}
```

使用 Map 将 Max 应用于 m 顶层的各个元素：

```
In[48]:= Map[Max, m, 2]
Out[48]= {6, 10, 8, 9}
```

您可以将两个等长列表的元素对应相加：

```
In[49]:= {a, b} + {x, y}
Out[49]= {a + x, b + y}
```

给列表的各个元素加上标量 c：

```
In[50]:= c + {a, b}
Out[50]= {a + c, b + c}
```

将列表中的每个元素乘以标量 k：

```
In[51]:= k {a, b}
Out[51]= {a k, b k}
```

设置一个由**5**个整数组成的列表：

```
In[43]:= v = Range[5]
Out[43]= {1, 2, 3, 4, 5}
```

将元素 x 追加到列表 v 的开始：

```
In[44]:= Prepend[v, x]
Out[44]= {x, 1, 2, 3, 4, 5}
```

将 x 追加到 v 的尾部：

```
In[45]:= Append[v, x]
Out[45]= {1, 2, 3, 4, 5, x}
```

```
In[49]:= {a, b} + {x, y}
Out[49]= {a + x, b + y}
```

给列表的各个元素加上标量 c：

```
In[50]:= c + {a, b}
Out[50]= {a + c, b + c}
```

将列表中的每个元素乘以标量 k：

```
In[51]:= k {a, b}
Out[51]= {a k, b k}
```

设置一个由**5**个整数组成的列表：

```
In[43]:= v = Range[5]
Out[43]= {1, 2, 3, 4, 5}
```

将元素 x 追加到列表 v 的开始：

```
In[44]:= Prepend[v, x]
Out[44]= {x, 1, 2, 3, 4, 5}
```

将 x 追加到 v 的尾部：

```
In[45]:= Append[v, x]
Out[45]= {1, 2, 3, 4, 5, x}
```

在列表 v 的第三位插入元素 x：

```
In[46]:= Insert[v, x, 3]
Out[46]= {1, 2, x, 3, 4, 5}
```

删除位于列表 v 第三位的元素：

```
In[47]:= Delete[v, 3]
Out[47]= {1, 2, 4, 5}
```

将列表 v 第三位的元素用新元素 x 替换：

```
In[48]:= ReplacePart[v, 3 → x]
Out[48]= {1, 2, x, 4, 5}
```

在 v 的各项之间交错插入 x：

```
In[49]:= Riffle[v, x]
Out[49]= {1, x, 2, x, 3, x, 4, x, 5}
```

教程

- 添加、删除和修改列表元素
- 标量、向量和矩阵的运算
- 通过索引操作列表

相关链接

- **How to:** 使用列表

参见

List · **Prepend** · **Append** · **Map** · **Apply** · **Insert** · **Delete** · **ReplacePart** · **Riffle**

更多关于

- 列表操作
- 列表元素
- **"How to"** 分类主题

How to 使用嵌套列表

嵌套列表是列表内的列表；它们是 *Mathematica* 中数据的主要结构，除了用于矩阵等常规用法外，还允许使用高维数据及参差不齐的数据集合.

创建一个列表的列表以供使用：

```
In[1]:= alist = {{a1, a2, a3, a4}, {b1, b2, b3, b4},
          {c1, c2, c3, c4}, {d1, d2, d3, d4}}

Out[1]= {{a1, a2, a3, a4}, {b1, b2, b3, b4},
         {c1, c2, c3, c4}, {d1, d2, d3, d4}}
```

在 ***Mathematica*** 中，矩阵用嵌套列表表示. 每一行对应于嵌套列表中的一个子列表：

```
In[2]:= MatrixForm[alist]
```

t[2]//MatrixForm=

$$\begin{pmatrix} a1 & a2 & a3 & a4 \\ b1 & b2 & b3 & b4 \\ c1 & c2 & c3 & c4 \\ d1 & d2 & d3 & d4 \end{pmatrix}$$

使用 Part 函数的简写形式 [[]]，得到第二行：

```
In[3]:= alist[[2]]
Out[3]= {b1, b2, b3, b4}
```

从第二行中得到第三个元素：

```
In[2]:= alist[[2, 3]]
Out[2]= b3
```

得到各行的第三个元素：

```
In[3]:= alist[[All, 3]]
Out[3]= {a3, b3, c3, d3}
```

使用 Flatten 解除嵌套：

```
In[4]:= Flatten[alist]
Out[4]= {a1, a2, a3, a4, b1, b2, b3,
         b4, c1, c2, c3, c4, d1, d2, d3, d4}
```

将展平后的数据显示为一列：

```
In[7]:= MatrixForm[Flatten[alist]]
```

t[7]//MatrixForm=

$$\begin{pmatrix} a1 \\ a2 \\ a3 \\ a4 \\ b1 \\ b2 \\ b3 \\ b4 \\ c1 \\ c2 \\ c3 \\ c4 \\ d1 \\ d2 \\ d3 \\ d4 \end{pmatrix}$$

添加 {}，将展平后的数据显示为一行：

```
In[8]:= MatrixForm[{Flatten[alist]}]
```

t[8]//MatrixForm=

$$\begin{pmatrix} a1 & a2 & a3 & a4 & b1 & b2 & b3 & b4 & c1 & c2 & c3 & c4 & d1 & c \end{pmatrix}$$

您可在嵌套列表的独立子列表上进行运算，也可以将嵌套列表作为一个整体进行运算.

设置一个数据集：

```
In[9]:= data = Table[i + j, {i, 1, 5}, {j, 1, 4}]

Out[9]= {{2, 3, 4, 5}, {3, 4, 5, 6},
```

```
In[7]:= MatrixForm[Flatten[alist]]
```

$$\begin{pmatrix} a1 \\ a2 \\ a3 \\ a4 \\ b1 \\ b2 \\ b3 \\ b4 \\ c1 \\ c2 \\ c3 \\ c4 \\ d1 \\ d2 \\ d3 \\ d4 \end{pmatrix}$$

添加 {}，将展平后的数据显示为一行：

```
In[8]:= MatrixForm[{Flatten[alist]}]
```

$$\begin{pmatrix} a1 & a2 & a3 & a4 & b1 & b2 & b3 & b4 & c1 & c2 & c3 & c4 & d1 & d2 & d3 & d4 \end{pmatrix}$$

您可在嵌套列表的独立子列表上进行运算，也可以将嵌套列表作为一个整体进行运算.

设置一个数据集：

```
In[9]:= data = Table[i + j, {i, 1, 5}, {j, 1, 4}]

Out[9]= {{2, 3, 4, 5}, {3, 4, 5, 6},
```

In[10]:=
```
MatrixForm[data]
```

t[10]//MatrixForm=

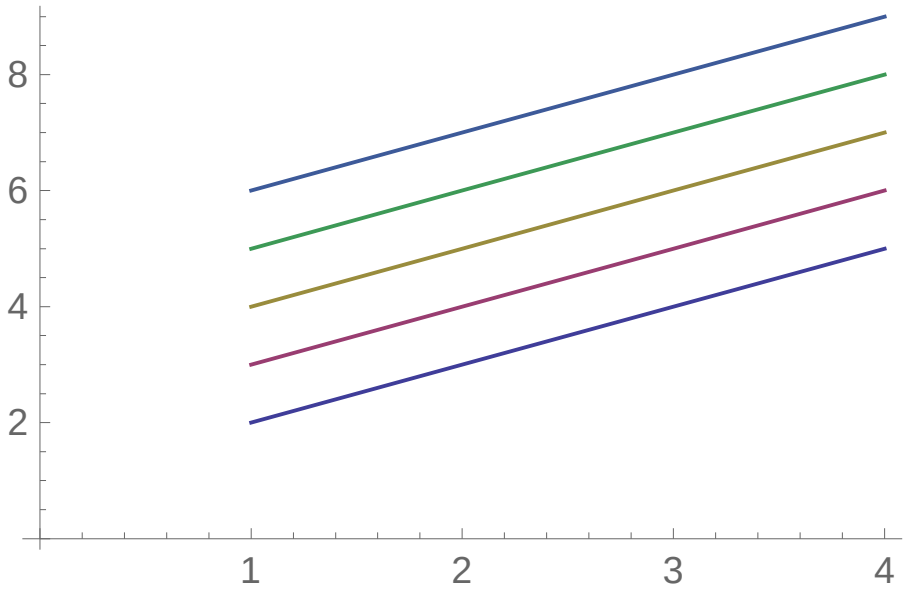
$$\begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \\ 5 & 6 & 7 & 8 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

大多数函数在嵌套列表内的每个子列表上映射.

创建数据集各行的图形:

In[11]:=
```
ListLinePlot[data, AxesOrigin → {0, 0}]
```

Out[11]=



大多数描述统计函数根据列运算.

求各列的均值:

In[12]:=
```
Mean[data]
```

Out[12]= {4, 5, 6, 7}

In[13]:=
```
Mean[Flatten[data]]
```

Out[13]= $\frac{11}{2}$

创建一个双嵌套列表：

In[14]:=
```
dat = Table[i + j + k,
    {i, 1, 4}, {j, 1, 4}, {k, 1, 4}];
MatrixForm[dat]
```

t[15]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} & \begin{pmatrix} 4 \\ 5 \\ 6 \\ 7 \end{pmatrix} & \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} & \begin{pmatrix} 6 \\ 7 \\ 0 \\ 9 \end{pmatrix} \\ \begin{pmatrix} 4 \\ 5 \\ 6 \\ 7 \end{pmatrix} & \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} & \begin{pmatrix} 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} & \begin{pmatrix} 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} \\ \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} & \begin{pmatrix} 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} & \begin{pmatrix} 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} & \begin{pmatrix} 8 \\ 9 \\ 10 \\ 11 \end{pmatrix} \\ \begin{pmatrix} 6 \\ 7 \\ 0 \\ 9 \end{pmatrix} & \begin{pmatrix} 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} & \begin{pmatrix} 8 \\ 9 \\ 10 \\ 11 \end{pmatrix} & \begin{pmatrix} 9 \\ 10 \\ 11 \\ 12 \end{pmatrix} \end{pmatrix}$$

Mean 现在列出的是每个嵌套子列表的均值：

In[13]:=
```
Mean[Flatten[data]]
```

Out[13]=
$$\frac{11}{2}$$

创建一个双嵌套列表：

In[14]:=
```
dat = Table[i + j + k,
    {i, 1, 4}, {j, 1, 4}, {k, 1, 4}];
MatrixForm[dat]
```

$$\begin{pmatrix}
\begin{pmatrix} 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} & \begin{pmatrix} 4 \\ 5 \\ 6 \\ 7 \end{pmatrix} & \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} & \begin{pmatrix} 6 \\ 7 \\ 0 \\ 9 \end{pmatrix} \\
\begin{pmatrix} 4 \\ 5 \\ 6 \\ 7 \end{pmatrix} & \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} & \begin{pmatrix} 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} & \begin{pmatrix} 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} \\
\begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} & \begin{pmatrix} 6 \\ 7 \\ 8 \\ 9 \end{pmatrix} & \begin{pmatrix} 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} & \begin{pmatrix} 8 \\ 9 \\ 10 \\ 11 \end{pmatrix} \\
\begin{pmatrix} 6 \\ 7 \\ 0 \\ 9 \end{pmatrix} & \begin{pmatrix} 7 \\ 8 \\ 9 \\ 10 \end{pmatrix} & \begin{pmatrix} 8 \\ 9 \\ 10 \\ 11 \end{pmatrix} & \begin{pmatrix} 9 \\ 10 \\ 11 \\ 12 \end{pmatrix}
\end{pmatrix}$$

Mean 现在列出的是每个嵌套子列表的均值：

```
In[16]:= MatrixForm[Mean[dat]]
```

Out[16]//MatrixForm=

$$\begin{pmatrix} \frac{9}{2} & \frac{11}{2} & \frac{13}{2} & \frac{15}{2} \\ \frac{11}{2} & \frac{13}{2} & \frac{15}{2} & \frac{17}{2} \\ \frac{13}{2} & \frac{15}{2} & \frac{17}{2} & \frac{19}{2} \\ \frac{15}{2} & \frac{17}{2} & \frac{19}{2} & \frac{21}{2} \end{pmatrix}$$

使用 `Flatten` 得到整个数据集的均值：

```
In[17]:= Mean[Flatten[dat]]
```

Out[17]= $\frac{15}{2}$

教程

- 列表的组合
- 作为集合的列表
- 重排列表
- 嵌套列表
- 嵌套列表的重排

相关链接

- **How to:** 使用列表

参见

更多关于

- 构造列表
- **"How to"** 分类主题

How to 将函数映射于列表

Mathematica 包含许多功能强大的操作，用于使用列表. 常常需要将函数映射到每个列表的每个单个元素. 对于具有可列表性的函数，在默认情况下就能做到，而对于具有不可列表性的函数，您可以使用 Map 来执行此操作.

首先设置一个由**1**至**5**的整数组成的列表：

```
In[293]:= mylist = Range[5]
Out[293]= {1, 2, 3, 4, 5}
```

您可以使用 Map 将函数映射到列表的每个元素；下面的例子使用的是一个未定义的函数 f：

```
In[294]:= Map[f, mylist]
Out[294]= {f[1], f[2], f[3], f[4], f[5]}
```

可以使用 /@ 作为 Map 的简写形式（这个命令与上例中的相同）：

```
In[295]:= f /@ mylist
Out[295]= {f[1], f[2], f[3], f[4], f[5]}
```

大多数数学函数具有 Listable 属性，表示它们将自动映射于列表：

```
In[296]:= Attributes[Sin]
Out[296]= {Listable, NumericFunction, Protected}

In[297]:= Sin[mylist]
Out[297]= {Sin[1], Sin[2], Sin[3], Sin[4], Sin[5]}
```

如果函数非 Listable，您可以使用 Map. 设置由5个2×2矩阵组成的列表：

```
In[298]:= matrices = RandomInteger[1, {5, 2, 2}]
Out[298]= {{{0, 0}, {0, 1}},
           {{1, 1}, {1, 1}}, {{0, 1}, {1, 0}},
           {{1, 0}, {0, 0}}, {{1, 1}, {0, 1}}}
```

使用 Map 将 MatrixForm 映射于列表，以数学符号形式查看每一个矩阵：

```
In[299]:= MatrixForm /@ matrices
```

$$\text{Out[299]}= \left\{\begin{pmatrix}0&0\\0&1\end{pmatrix}, \begin{pmatrix}1&1\\1&1\end{pmatrix}, \begin{pmatrix}0&1\\1&0\end{pmatrix}, \begin{pmatrix}1&0\\0&0\end{pmatrix}, \begin{pmatrix}1&1\\0&1\end{pmatrix}\right\}$$

现在用 Map 计算列表中每个矩阵的特征值：

```
In[300]:= Eigenvalues /@ matrices
Out[300]= {{1, 0}, {2, 0}, {-1, 1}, {1, 0}, {1, 1}}
```

Map 不仅可以操作列表. 它也可以用于任何表达式:

```
In[301]:= Map[f, a + b + c + d]
Out[301]= f[a] + f[b] + f[c] + f[d]
```

Apply 是另一个函数编程操作. 它用于替换一个表达式的头:

您可以通过两个未定义的函数 f 和 g 来看这是怎样作用的:

```
In[302]:= Apply[f, g[a, b, c, d]]
Out[302]= f[a, b, c, d]
```

Apply 用 @@ 作为简写形式（这个命令与上例中的相同）:

```
In[303]:= f @@ g[a, b, c, d]
Out[303]= f[a, b, c, d]
```

在 ***Mathematica*** 中公用表达式以 StandardForm 形式显示，而它们的底层 FullForm 说明可以如何使用 Apply:

```
In[304]:= {Plus[a, b, c], Times[a, b, c], List[a, b, c]}
Out[304]= {a + b + c, a b c, {a, b, c}}
```

例如，这里将加和变成乘积:

```
In[300]:= Eigenvalues /@ matrices
Out[300]= {{1, 0}, {2, 0}, {-1, 1}, {1, 0}, {1, 1}}
```

Map 不仅可以操作列表. 它也可以用于任何表达式:

```
In[301]:= Map[f, a + b + c + d]
Out[301]= f[a] + f[b] + f[c] + f[d]
```

Apply 是另一个函数编程操作. 它用于替换一个表达式的头:

您可以通过两个未定义的函数 f 和 g 来看这是怎样作用的:

```
In[302]:= Apply[f, g[a, b, c, d]]
Out[302]= f[a, b, c, d]
```

Apply 用 @@ 作为简写形式（这个命令与上例中的相同）:

```
In[303]:= f @@ g[a, b, c, d]
Out[303]= f[a, b, c, d]
```

在 ***Mathematica*** 中公用表达式以 StandardForm 形式显示，而它们的底层 FullForm 说明可以如何使用 Apply:

```
In[304]:= {Plus[a, b, c], Times[a, b, c], List[a, b, c]}
Out[304]= {a + b + c, a b c, {a, b, c}}
```

例如，这里将加和变成乘积:

```
In[305]:= Apply[Times, a + b + c]
Out[305]= a b c
```

当您想要将列表中的元素转换为函数的参数时，Apply 很有用.

创建一个由5个有序对 {a, b} 组成的列表：

```
In[306]:= pairs = RandomInteger[{1, 10}, {5, 2}]
Out[306]= {{8, 7}, {5, 9}, {6, 8}, {7, 1}, {6, 7}}
```

Mod 求有序对中第一个数除以第二个数得到的余数：

```
In[307]:= Mod[10, 4]
Out[307]= 2
```

将 Mod 应用于全部有序对，需要在列表的第一层作用（由 {1} 指定）：

```
In[308]:= Apply[Mod, pairs, {1}]
Out[308]= {1, 5, 6, 0, 6}
```

可以用 @@@ 作为简写符号应用于第一层：

```
In[309]:= Mod @@@ pairs
Out[309]= {1, 5, 6, 0, 6}
```

这是完成同样任务的另一种方法，方法是利用带有 Function 的纯

```
In[305]:= Apply[Times, a + b + c]
Out[305]= a b c
```

当您想要将列表中的元素转换为函数的参数时，Apply 很有用.

创建一个由5个有序对 {a, b} 组成的列表：

```
In[306]:= pairs = RandomInteger[{1, 10}, {5, 2}]
Out[306]= {{8, 7}, {5, 9}, {6, 8}, {7, 1}, {6, 7}}
```

Mod 求有序对中第一个数除以第二个数得到的余数：

```
In[307]:= Mod[10, 4]
Out[307]= 2
```

将 Mod 应用于全部有序对，需要在列表的第一层作用（由 {1} 指定）：

```
In[308]:= Apply[Mod, pairs, {1}]
Out[308]= {1, 5, 6, 0, 6}
```

可以用 @@@ 作为简写符号应用于第一层：

```
In[309]:= Mod @@@ pairs
Out[309]= {1, 5, 6, 0, 6}
```

这是完成同样任务的另一种方法，方法是利用带有 Function 的纯

```
In[310]:= Map[Function[w, Apply[Mod, w]], pairs]
Out[310]= {1, 5, 6, 0, 6}
```

这里使用 Function 的简写形式：

```
In[311]:= Map[Apply[Mod, #] &, pairs]
Out[311]= {1, 5, 6, 0, 6}
```

教程

- 函数作用于表达式的部分项
- 函数操作
- 表达式的层次结构
- 构建列表
- 嵌套列表

相关链接

- **How to:** 使用变量和函数
- **How to:** 使用列表

参见

Map · Apply · Listable

更多关于

- 函数作用于列表

- 函数式编程
- 大型数组的处理
- **"How to"** 分类主题

How to 制作表格

在 *Mathematica* 中，很多类型的数据在表格或列表中存储. *Mathematica* 为创建与操作表格提供了多种实用的函数.

使用 `Table` 制作一个函数值表格：

```
In[3]:= Table[i^2, {i, 10}]

Out[3]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

使用两个变量的 `Table` 制作一个二维表格（存储为 `m`）：

```
In[7]:= m = Table[i^2 - j^3, {i, 5}, {j, 6}]

Out[7]= {{0, -7, -26, -63, -124, -215},
         {3, -4, -23, -60, -121, -212},
         {8, 1, -18, -55, -116, -207},
         {15, 8, -11, -48, -109, -200},
         {24, 17, -2, -39, -100, -191}}
```

使用 `Grid` 将值放在一个网格中：

In[8]:=
```
Grid[m]
```

Out[8]=

0	−7	−26	−63	−124	−215
3	−4	−23	−60	−121	−212
8	1	−18	−55	−116	−207
15	8	−11	−48	−109	−200
24	17	−2	−39	−100	−191

使用一个选项加上边框：

In[9]:=
```
Grid[m, Frame → All]
```

Out[9]=

0	−7	−26	−63	−124	−215
3	−4	−23	−60	−121	−212
8	1	−18	−55	−116	−207
15	8	−11	−48	−109	−200
24	17	−2	−39	−100	−191

Mathematica 中的表格可以包含任意元素. 与 `Plot` 合用，制作一个 `Table`，并用 `ImageSize` 控制输出的大小：

In[8]:=
```
Grid[m]
```

Out[8]=

0	−7	−26	−63	−124	−215
3	−4	−23	−60	−121	−212
8	1	−18	−55	−116	−207
15	8	−11	−48	−109	−200
24	17	−2	−39	−100	−191

使用一个选项加上边框：

In[9]:=
```
Grid[m, Frame → All]
```

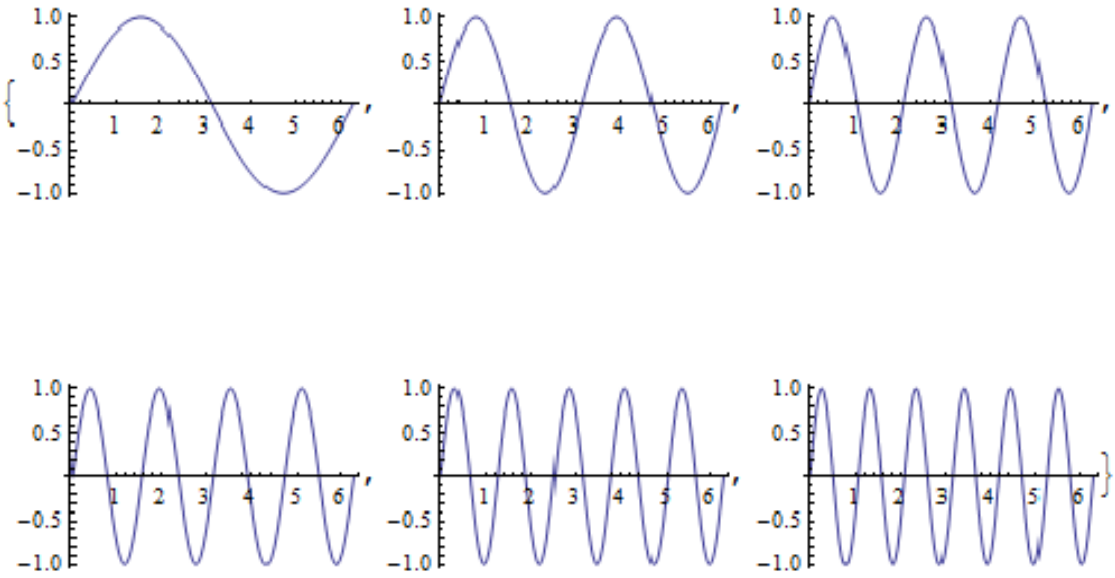
Out[9]=

0	−7	−26	−63	−124	−215
3	−4	−23	−60	−121	−212
8	1	−18	−55	−116	−207
15	8	−11	−48	−109	−200
24	17	−2	−39	−100	−191

Mathematica 中的表格可以包含任意元素. 与 `Plot` 合用，制作一个 `Table`，并用 `ImageSize` 控制输出的大小：

In[9]:=
```
Table[Plot[Sin[n x], {x, 0, 2 Pi},
  ImageSize → {150, 150}], {n, 1, 6}]
```

Out[9]=



也可以用表格迭代量（下例中为 q）的任意值列表创建一个 Table：

In[10]:=
```
Table[Factor[q^5 - 1], {q, {x, y, z}}]
```

Out[10]=

$$\left\{(-1+x)\left(1+x+x^2+x^3+x^4\right),\ (-1+y)\left(1+y+y^2+y^3+y^4\right),\ (-1+z)\left(1+z+z^2+z^3+z^4\right)\right\}$$

教程

- 值表的生成
- 表与矩阵

相关链接

- **How to:** 使用表格

参见

Table · **Grid** · **Frame** · **ImageSize**

更多关于

- 网格和表
- 大型数组的处理
- **"How to"** 分类主题

How to 使用规则

规则是 *Mathematica* 强大的表达式变换语言的关键部分. 这些 "How tos" 为 *Mathematica* 中规则的使用给出逐步说明.

How to: 创建和使用规则

How to: 使用规则形式的解

How to: 将规则多次用于表达式

教程

- 运用变换规则
- 一组变换规则的操作
- 函数的变换规则
- 模式和变换规则
- 变换规则和定义

相关链接

- **How to:** 使用变量和函数

参见

Rule · **RuleDelayed** · **Set** · **ReplaceAll** · **ReplaceRepeated** · **ReplaceList**

更多关于

- 规则
- 赋值
- 选项管理
- 模式
- **"How to"** 分类主题

How to 创建和使用规则

Mathematica 中的变换规则使您可以为符号、函数及所有其它类型的表达式设置局部值. 规则的使用提供了一个强大和可扩展的方法，用您指定的值来取代另一表达式的全部或一部分.

规则的缩写形式使用一个右箭头，它可以通过输入 -> 得到（在 - 和 > 之间无空格）. 在进一步输入时， ***Mathematica*** 前端自动将 -> 转

创建下述变换规则，它可以被认为是“x 取 3”：

```
In[1]:= x → 3

Out[1]= x → 3
```

通过观察 x → 3 的输出，您可以看到该规则并不做任何事情：输出仅仅是规则本身. 这是因为当规则单独存在时，它们是没有任何作用的. 您必须将规则与表达式合用才会有所用处.

规则可以通过使用 /. 应用于表达式（ReplaceAll 的缩写形式）. 一般的语法为 *expr* /. *rules*.

用 /. 将规则用于表达式：

```
In[57]:= 2 x + 1 /. x → 3

Out[57]= 7
```

要将多个规则用于一个表达式，要把它们放到一个列表中 {}：

```
In[58]:= 2 x + y /. {x → 3, y → 4}

Out[58]= 10
```

如果对同一变量给出两个规则，***Mathematica*** 将仅用第一个规则：

```
In[59]:= 2 x + y /. {x → 3, x → 4}

Out[59]= 6 + y
```

可以将变量用任何表达式代替，而不只是单独的值.

用 3 y 替换 x:

```
In[60]:= 2 x + y /. x → 3 y
Out[60]= 7 y
```

可以使用一个规则代替表达式的一大部分:

```
In[62]:= 9 + x^2 - 9 x^3 /. x^2 - 9 x^3 → 3 y
Out[62]= 9 + 3 y
```

事实上，规则可以用于包括函数在内的任何表达式.

用 3 替换 x:

```
In[65]:= 1 + f[x] + f[y] /. x → 3
Out[65]= 1 + f[3] + f[y]
```

对 f[x] 使用一个规则. 注意该规则与 f[x] 完全匹配，并不影响 f[y]:

```
In[64]:= 1 + f[x] + f[y] /. f[x] → p
Out[64]= 1 + p + f[y]
```

要代替函数 f，而不论其自变量如何，您必须在规则中使用一种模式.

规则 f[x_] → x^2 可以被读作 “f[*anything*] 取 *anything*^2”:

```
In[18]:= 1 + f[x] + f[y] /. f[x_] → x^2
Out[18]= 1 + x^2 + y^2
```

关于使用模式的更多信息，请参见"模式引言."

使用 → 设置的规则是即时规则. 也就是说，右端与规则同时计算：

```
In[1]:= x → RandomReal[]
Out[1]= x → 0.107855
```

您有时可能要使用延时规则，这种规则在用于表达式之前保持不被计算. 延时规则通过 RuleDelayed 创建.

延时规则的缩写形式是 :>（: 和 > 之间无空格）. *Mathematica* 前端会自动将 :> 转换成 :↦. 任何一个都代表 RuleDelayed 的缩写形式：

```
In[2]:= x :↦ RandomReal[]
Out[2]= x :↦ RandomReal[]
```

考虑这样一个问题：您希望使用一个规则来生成3个在0和1之间的随机实数. 使用即时规则生成的是3个相同的数：

```
In[3]:= {x, x, x} /. x → RandomReal[]
Out[3]= {0.474858, 0.474858, 0.474858}
```

要得到三个不同数值，使用延时规则：

```
In[18]:= 1 + f[x] + f[y] /. f[x_] → x^2
Out[18]= 1 + x^2 + y^2
```

关于使用模式的更多信息，请参见"模式引言."

使用 → 设置的规则是即时规则. 也就是说，右端与规则同时计算：

```
In[1]:= x → RandomReal[]
Out[1]= x → 0.107855
```

您有时可能要使用延时规则，这种规则在用于表达式之前保持不被计算. 延时规则通过 RuleDelayed 创建.

延时规则的缩写形式是 :>（: 和 > 之间无空格）. *Mathematica* 前端会自动将 :> 转换成 :↦. 任何一个都代表 RuleDelayed 的缩写形式：

```
In[2]:= x :↦ RandomReal[]
Out[2]= x :↦ RandomReal[]
```

考虑这样一个问题：您希望使用一个规则来生成3个在0和1之间的随机实数. 使用即时规则生成的是3个相同的数：

```
In[3]:= {x, x, x} /. x → RandomReal[]
Out[3]= {0.474858, 0.474858, 0.474858}
```

要得到三个不同数值，使用延时规则：

```
In[4]:= {x, x, x} /. x :> RandomReal[]
Out[4]= {0.346312, 0.120139, 0.505519}
```

显式使用 = 的赋值具有全局效应，而规则仅对使用它的表达式产生影响.

用 = 将 x 赋为 3，然后计算 x 来查看它的值：

```
In[19]:= x = 3;
In[2]:= x
Out[2]= 3
```

使用规则将一个值赋给 y：

```
In[20]:= y → 3;
```

计算 y，可以看到赋给它的值并没有保存：

```
In[4]:= y
Out[4]= y
```

规则必须用于表达式才有作用. 然而，您可以将一个规则赋给一个符号，然后像使用规则一样使用这个符号.

使用 = 将规则 p → 2 赋给 n，然后将 n 应用于一个表达式：

```
In[68]:= n = p → 2;
```

```
In[4]:= {x, x, x} /. x :> RandomReal[]
Out[4]= {0.346312, 0.120139, 0.505519}
```

显式使用 = 的赋值具有全局效应，而规则仅对使用它的表达式产生影响.

用 = 将 x 赋为 3，然后计算 x 来查看它的值：

```
In[19]:= x = 3;
In[2]:= x
Out[2]= 3
```

使用规则将一个值赋给 y：

```
In[20]:= y → 3;
```

计算 y，可以看到赋给它的值并没有保存：

```
In[4]:= y
Out[4]= y
```

规则必须用于表达式才有作用. 然而，您可以将一个规则赋给一个符号，然后像使用规则一样使用这个符号.

使用 = 将规则 p → 2 赋给 n，然后将 n 应用于一个表达式：

```
In[68]:= n = p → 2;
```

```
In[69]:= 2 p /. n
Out[69]= 4
```

由于 p → 2 现在作为符号 n 被全局保存，您可以继续在 p → 2 的地方使用 n.

类似地，您可以将一个表达式赋给一个符号，然后对这个符号应用规则：

```
In[22]:= t = 1 + y^2 + y^3 + 3 y;
In[6]:= t /. y → 2
Out[6]= 19
```

当您将表达式用于多个计算时，这样做尤其方便.

教程

- 运用变换规则
- 一组变换规则的操作
- 函数的变换规则
- 引言
- 模式和变换规则
- 变换规则和定义

相关链接

- **How to:** 使用变量和函数
- **How to:** 使用规则

参见

Rule · **RuleDelayed** · **ReplaceAll** · **Set**

更多关于

- 规则
- 赋值
- 选项管理
- 模式
- **"How to"** 分类主题

How to 使用规则形式的解

由于 *Mathematica* 中的许多函数给出的是规则形式的解，您需要能够使用这些规则来探究并解释您的结果. 尽管这类解的多种使用方法取决于所求问题的具体类型，有两个基本步骤是一致的： 从列表中得到规则解，然后将它们应用于一个表达式.

解这个简单的方程 $x + 3 = 5$，得到 x：

```
In[62]:= s = Solve[x + 3 == 5, x]
Out[62]= {{x → 2}}
```

该方程的解包含在一个嵌套列表（即一个列表的列表）中. ***Mathematica*** 中的列表用 { } 表示. 列表中的各项被称作元素，并且可以根据它们的位置来引用

要使用该解，必须首先将它从嵌套列表中取出。方法是使用 [[ ]]（Part 的简写形式），并指定该解在嵌套列表中的位置。该例中，解是嵌套列表中的第一个、也是唯一一个元素：

```
In[63]:= s[[1]]
Out[63]= {x → 2}
```

现在您可以使用解，用 /.（ReplaceAll 的简写形式）将解代入表达式 $x + 4$：

```
In[73]:= x + 4 /. s[[1]]
Out[73]= 6
```

返回一个二次方程的两个解。有两个子列表，每个解一个：

```
In[77]:= t = Solve[x^2 == 9, x]
Out[77]= {{x → -3}, {x → 3}}
```

这里使用第二个解：

```
In[78]:= x + 4 /. t[[2]]
Out[78]= 7
```

求解线性方程组时，解集在一个子列表中返回。

设置简单线性方程组的列表，用于求解：

要使用该解，必须首先将它从嵌套列表中取出。方法是使用 [[ ]]（Part 的简写形式），并指定该解在嵌套列表中的位置。该例中，解是嵌套列表中的第一个、也是唯一一个元素：

```
In[63]:= s[[1]]
Out[63]= {x → 2}
```

现在您可以使用解，用 /.（ReplaceAll 的简写形式）将解代入表达式 $x + 4$：

```
In[73]:= x + 4 /. s[[1]]
Out[73]= 6
```

返回一个二次方程的两个解。有两个子列表，每个解一个：

```
In[77]:= t = Solve[x^2 == 9, x]
Out[77]= {{x → -3}, {x → 3}}
```

这里使用第二个解：

```
In[78]:= x + 4 /. t[[2]]
Out[78]= 7
```

求解线性方程组时，解集在一个子列表中返回。

设置简单线性方程组的列表，用于求解：

```
In[78]:= eqns = {x + 3 == 4, y - 5 == 2, 3 z == 4}
```

$$\text{Out[78]= } \{3 + x == 4,\ -5 + y == 2,\ 3\,z == 4\}$$

解方程组。包含唯一解集的一个嵌套列表被返回：

```
In[79]:= u = Solve[eqns, {x, y, z}]
```

$$\text{Out[79]= } \left\{\left\{x \to 1,\ y \to 7,\ z \to \frac{4}{3}\right\}\right\}$$

含有解集的内列表是嵌套列表中的第一个（也是唯一一个）元素。因此，将 [[1]] 用于嵌套列表将返回解的列表：

```
In[80]:= u[[1]]
```

$$\text{Out[80]= } \left\{x \to 1,\ y \to 7,\ z \to \frac{4}{3}\right\}$$

这里使用该解集：

```
In[81]:= x + y + z /. u[[1]]
```

$$\text{Out[81]= } \frac{28}{3}$$

拓展语法，将解在内列表中的位置加上，可以得到解集的一部分。

得到第一个变量的解：

```
In[83]:= u[[1, 1]]
```

$$\text{Out[83]= } x \to 1$$

```
In[84]:= x + y + z /. u[[1, 1]]
Out[84]= 1 + y + z
```

类似地，代入第二个和第三个变量：

```
In[91]:= x + y + z /. {u[[1, 2]], u[[1, 3]]}
Out[91]= 25/3 + x
```

这些方程含有两个解集：

```
In[128]:= v = Solve[
            {x^2 + y^2 + z^2 == 9, y == x, y == z}, {x, y, z}]
Out[128]= {{x → -√3, z → -√3, y → -√3},
           {x → √3, z → √3, y → √3}}
```

将两个解集都代入：

```
In[129]:= x + y + z /. v
Out[129]= {-3 √3, 3 √3}
```

这里说明的是如何验证一个方程的解（集）：

首先，求解一个含参数的二次方程：

```
In[96]:= eqn = x^2 + 2 a x + 1 == 0;
```

```
In[84]:= x + y + z /. u[[1, 1]]
Out[84]= 1 + y + z
```

类似地，代入第二个和第三个变量：

```
In[91]:= x + y + z /. {u[[1, 2]], u[[1, 3]]}
Out[91]= 25/3 + x
```

这些方程含有两个解集：

```
In[128]:= v = Solve[
            {x^2 + y^2 + z^2 == 9, y == x, y == z}, {x, y, z}]
Out[128]= {{x → -√3, z → -√3, y → -√3},
           {x → √3, z → √3, y → √3}}
```

将两个解集都代入：

```
In[129]:= x + y + z /. v
Out[129]= {-3 √3, 3 √3}
```

这里说明的是如何验证一个方程的解（集）：

首先，求解一个含参数的二次方程：

```
In[96]:= eqn = x^2 + 2 a x + 1 == 0;
```

In[97]:=
```
sol = Solve[eqn, x]
```

Out[97]=
$$\left\{\left\{x \to -a - \sqrt{-1 + a^2}\right\}, \left\{x \to -a + \sqrt{-1 + a^2}\right\}\right\}$$

通过代入法验证结果：

In[98]:=
```
sub = eqn /. sol
```

Out[98]=
$$\left\{1 + 2a\left(-a - \sqrt{-1 + a^2}\right) + \left(-a - \sqrt{-1 + a^2}\right)^2 == 0,\right.$$
$$\left.1 + 2a\left(-a + \sqrt{-1 + a^2}\right) + \left(-a + \sqrt{-1 + a^2}\right)^2 == 0\right\}$$

注意的是，代入后的结果是方程的形式，而不是 True 或 False.

用 Simplify 确定代入结果是否满足方程. 每个代入的运行结果都是 True，表明解使方程成立：

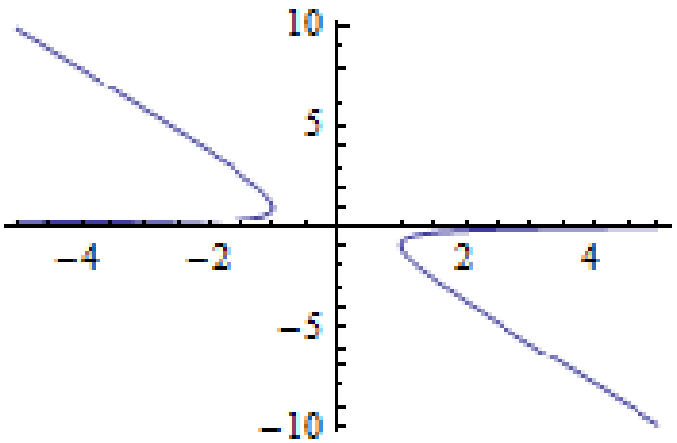
In[99]:=
```
Simplify[sub]
```

Out[99]=
```
{True, True}
```

这里说明的是，如何通过代入对含参数 a 的函数的解进行绘图：

In[70]:=
```
Plot[x /. sol, {a, -5, 5}]
```

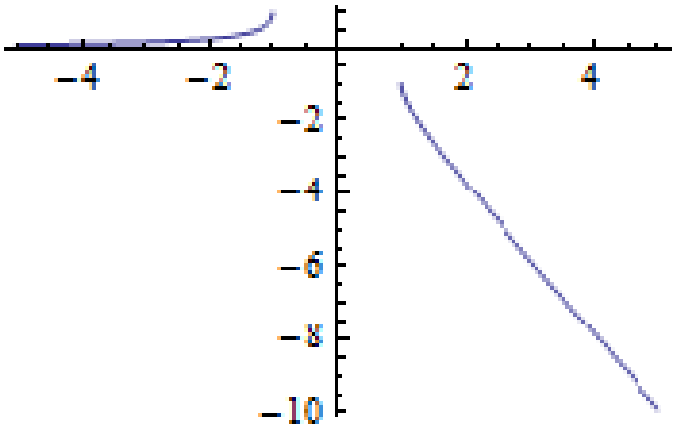
Out[70]=



类似地，只绘出 sol 中第一个解的图形：

```
In[72]:= Plot[x /. sol[[1]], {a, -5, 5}]
```

Out[72]=



教程

- 运用变换规则
- 一组变换规则的操作
- 函数的变换规则
- 方程和不等式
- 函数中的数值运算
- 使用 **DSolve** 求解微分方程

相关链接

- **How to:** 使用列表
- **How to:** 进行代数计算
- **How to:** 使用规则

参见

更多关于

- 规则
- **"How to"** 分类主题

How to 将规则多次用于表达式

您可能会发现，对一个表达式多次使用一个或多个规则很有用. *Mathematica* 提供了将规则循环用于表达式的函数. 当一个规则可以以多种方式用于表达式时，*Mathematica* 还允许您查看所有可能的结果.

尽管 /.（ReplaceAll 的简写形式） 只能单次地将规则用于一个表达式，您将发现重复地将规则应用于表达式，直到它不再发生任何改变为止，非常有用. 这可以通过使用 //.（ReplaceRepeated 的简写形式）实现.

使用 /. 将一个规则或规则列表单次应用于一个表达式的各个部分：

```
In[137]:= a + x^2 + y^6 /. {x → 2 + a, a → 1}
```

$$\text{Out[137]=}\quad 1 + (2 + a)^2 + y^6$$

用 //. 重复应用规则，直到表达式不再变化为止：

```
In[138]:= a + x^2 + y^6 //. {x → 2 + a, a → 1}
Out[138]= 10 + y^6
```

相似地：

```
In[139]:= log[a b c d] /. log[x_ y_] → log[x] + log[y]
Out[139]= log[a] + log[b c d]

In[140]:= log[a b c d] //. log[x_ y_] → log[x] + log[y]
Out[140]= log[a] + log[b] + log[c] + log[d]
```

将 //. 用于一个循环的规则集合时，应当慎用. 用 MaxIterations 选项设置 //. 的最大循环次数：

```
In[64]:= ReplaceRepeated[x,
          x → x + 1, MaxIterations → 1000]

         ReplaceRepeated::rrlim :
          Exiting after x scanned 1000 times. >>

Out[64]= 1000 + x
```

可用 ReplaceList 通过应用一个规则或规则列表对整个表达式进行各种可能方式的变换. 使用 Column 以易于查看的方式显示结果：

```
In[143]:= ReplaceList[log[a b c],
            log[x_ y_] → log[x] + log[y]] // Column

Out[143]= log[a] + log[b c]
          log[b] + log[a c]
          log[a b] + log[c]
          log[a b] + log[c]
          log[b] + log[a c]
          log[a] + log[b c]
```

教程

- 运用变换规则
- 一组变换规则的操作
- 函数的变换规则
- 引言
- 模式和变换规则
- 变换规则和定义
- 函数的重复调用

相关链接

- **How to:** 使用变量和函数
- **How to:** 使用规则

参见

ReplaceAll · **ReplaceRepeated** · **ReplaceList** · **MaxIterations**

更多关于

- 规则
- 赋值
- 选项管理
- 模式
- 函数迭代
- **"How to"** 分类主题

How to 使用 *Mathematica* 的语法

Mathematica 丰富的语法是为了保障 *Mathematica* 多种语言、数学和其它结构的一致性、高效性和可读性而精心设计的. 了解这些结构对于高效使用 *Mathematica* 至关重要，并奠定了广泛使用 *Mathematica* 各种强大功能的基础.

How to: 使用函数模板

How to: 正确使用括号和大括号

How to: 平衡括号和大括号

How to: 使用简写符号

How to: 输入函数的范围和选项

教程

- ***Mathematica*** 语言的语法
- 一些普遍的记号和传统表示
- ***Mathematica*** 的四种括号
- 输入语法
- 笔记本的输入和输出
- 表达式
- 表达式输入的特殊方式

相关链接

- **How to:** 使用列表
- **How to:** 使用变量和函数
- **How to:** 在笔记本中进行内容的选择与键入

更多关于

- ***Mathematica*** 语法
- 笔记本中的选择与输入
- ***Mathematica*** 语法符号
- 特殊字符
- **"How to"** 分类主题

How to 正确使用括号和大括号

Mathematica 丰富的语法使用不同种类的括号和大括号；熟悉这些方面使您能够在 *Mathematica* 中进行有效地阅读与编程.

圆括号**( )**、大括号**{ }**以及方括号**[ ]** 在 ***Mathematica*** 中各代表不同的意义. 前两个有时也分别称作圆括弧和花括号.

圆括号（ ）在 ***Mathematica*** 中用作对表达式编组和确定运算的优先次序：

```
In[53]:= 1 + 2 / 3
```

$$\text{Out[53]=}\ \frac{5}{3}$$

```
In[54]:= (1 + 2) / 3
Out[54]= 1
```

Mathematica 中的列表用大括号 { } 表示，它汇集了被称作元素的各项.

创建一个列表，由前五个正整数组成：

```
In[5]:= {1, 2, 3, 4, 5}
Out[5]= {1, 2, 3, 4, 5}
```

Mathematica 中的任何事物都可在列表中使用，包括数字、变量、排版数学表达式以及字符串等：

$$\text{In[17]:=}\ \left\{1,\ b,\ 2,\ 3,\ 3\,x == 12,\ \sqrt{9 + y},\ \text{"hello"}\right\}$$

$$\text{Out[17]=}\ \left\{1,\ b,\ 2,\ 3,\ 3\,x == 12,\ \sqrt{9 + y},\ \text{hello}\right\}$$

列表可包含其它列表，从而得到嵌套列表：

```
In[18]:= {1, 1, {3, 4, 5}, {3, 2}}
Out[18]= {1, 1, {3, 4, 5}, {3, 2}}
```

方括号在 ***Mathematica*** 中用于插入函数参数.

```
In[53]:= 1 + 2 / 3
```

$$\text{Out[53]=}\ \frac{5}{3}$$

```
In[54]:= (1 + 2) / 3
Out[54]= 1
```

Mathematica 中的列表用大括号 { } 表示，它汇集了被称作元素的各项.

创建一个列表，由前五个正整数组成：

```
In[5]:= {1, 2, 3, 4, 5}
Out[5]= {1, 2, 3, 4, 5}
```

Mathematica 中的任何事物都可在列表中使用，包括数字、变量、排版数学表达式以及字符串等：

$$\text{In[17]:=}\ \left\{1,\ b,\ 2,\ 3,\ 3\,x == 12,\ \sqrt{9 + y},\ \text{"hello"}\right\}$$

$$\text{Out[17]=}\ \left\{1,\ b,\ 2,\ 3,\ 3\,x == 12,\ \sqrt{9 + y},\ \text{hello}\right\}$$

列表可包含其它列表，从而得到嵌套列表：

```
In[18]:= {1, 1, {3, 4, 5}, {3, 2}}
Out[18]= {1, 1, {3, 4, 5}, {3, 2}}
```

方括号在 ***Mathematica*** 中用于插入函数参数.

此处，用方括号将函数 Range、Sin 以及 N 的参数分别括入：

```
In[45]:= Range[10]
Out[45]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[2]:= Sin[2]
Out[2]= Sin[2]

In[46]:= N[Sin[2]]
Out[46]= 0.909297
```

Mathematica 将双方括号 [[ ]] 用作函数 Part 的简写形式，该函数用于得到列表的一部分：

```
In[3]:= v = Range[10]^2
Out[3]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}

In[5]:= v[[3]]
Out[5]= 9
```

不同的括号架构可以一起使用.

绘出一个函数的图形，在列表中指定绘图范围：

此处，用方括号将函数 Range、Sin 以及 N 的参数分别括入：

```
In[45]:= Range[10]
Out[45]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[2]:= Sin[2]
Out[2]= Sin[2]

In[46]:= N[Sin[2]]
Out[46]= 0.909297
```

Mathematica 将双方括号 [[ ]] 用作函数 Part 的简写形式，该函数用于得到列表的一部分：

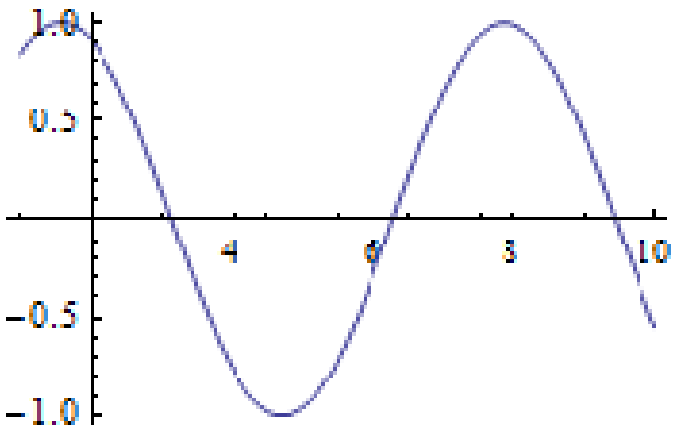
```
In[3]:= v = Range[10]^2
Out[3]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}

In[5]:= v[[3]]
Out[5]= 9
```

不同的括号架构可以一起使用.

绘出一个函数的图形，在列表中指定绘图范围：

```
In[48]:= Plot[Sin[x], {x, 1, 10}]
```

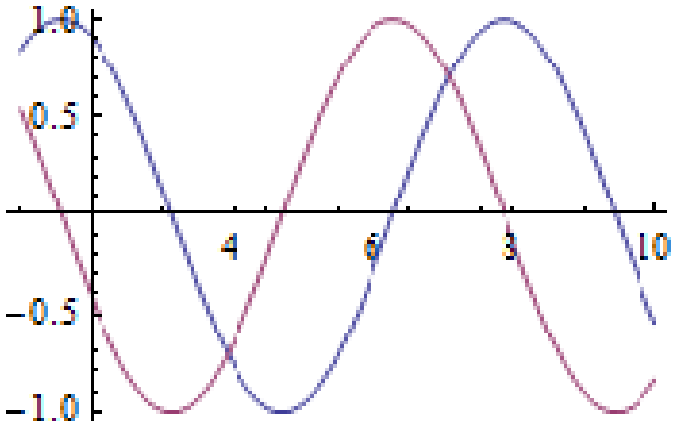
Out[48]=



函数与列表并用的能力无缝地集成在 ***Mathematica*** 中. 将两个函数的图形绘制在一起——这一对函数在一个列表中列出:

```
In[49]:= Plot[{Sin[x], Cos[x]}, {x, 1, 10}]
```

Out[49]=



所有的括号字符必须平衡（即成对出现），以便 ***Mathematica*** 运行一个表达式. 如果括号不平衡，***Mathematica*** 前端将把括号变成紫色:

```
(2 + π
```

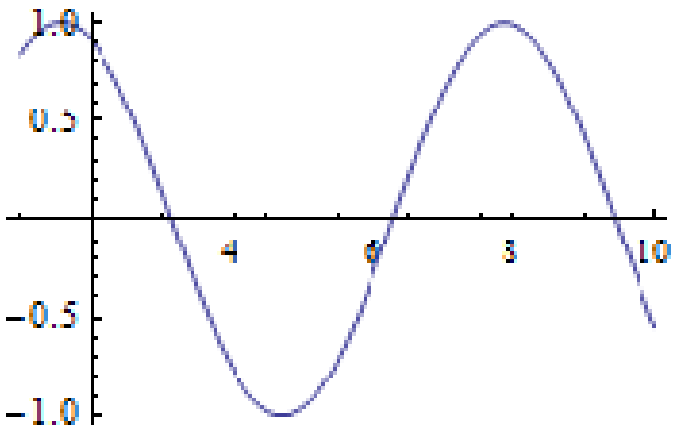
如对该表达式进行运算将生成错误:

```
In[6]:= (2 + π)
```

更多关于括号与大括号平衡的信息，请参阅 **How to**：平衡括号和大括

```
In[48]:= Plot[Sin[x], {x, 1, 10}]
```

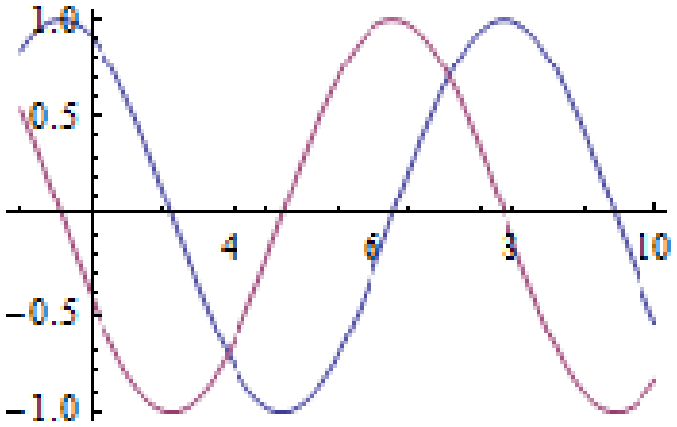
Out[48]=



函数与列表并用的能力无缝地集成在 ***Mathematica*** 中. 将两个函数的图形绘制在一起——这一对函数在一个列表中列出:

```
In[49]:= Plot[{Sin[x], Cos[x]}, {x, 1, 10}]
```

Out[49]=



所有的括号字符必须平衡（即成对出现），以便 ***Mathematica*** 运行一个表达式. 如果括号不平衡，***Mathematica*** 前端将把括号变成紫色:

```
(2 + π
```

如对该表达式进行运算将生成错误:

```
In[6]:= (2 + π)
```

更多关于括号与大括号平衡的信息，请参阅 **How to**：平衡括号和大括

教程

- ***Mathematica*** 的四种括号
- 一些普遍的记号和传统表示
- ***Mathematica*** 语言的语法
- 输入语法
- 运算序列

相关链接

- **How to:** 平衡括号和大括号
- **How to:** 使用 ***Mathematica*** 的语法
- **How to:** 使用列表
- **How to:** 使用变量和函数

更多关于

- ***Mathematica*** 语法
- **"How to"** 分类主题

How to 平衡括号和大括号

Mathematica 中的所有括号必须平衡（即成对出现）. 也就是说，每种类型的开括号必须由相应的右括号平衡. 如果出现不平衡情况，那么 *Mathematica* 不会对单元进行计算. 在 *Mathematica* 前端包含几个方便的工具，让您确保括号和大括号是平衡的.

教程

- ***Mathematica*** 的四种括号
- 一些普遍的记号和传统表示
- ***Mathematica*** 语言的语法
- 输入语法
- 运算序列

相关链接

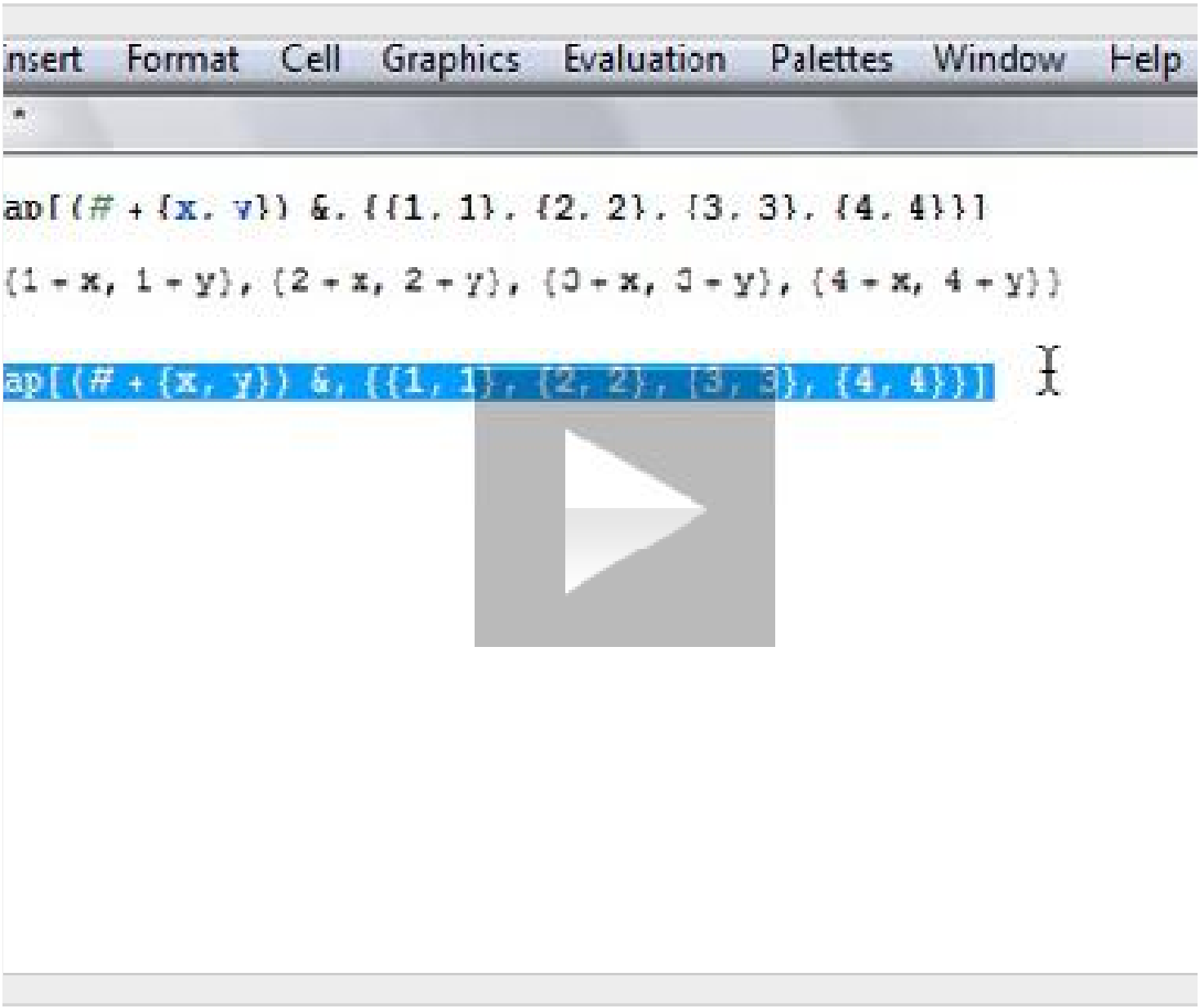
- **How to:** 平衡括号和大括号
- **How to:** 使用 ***Mathematica*** 的语法
- **How to:** 使用列表
- **How to:** 使用变量和函数

更多关于

- ***Mathematica*** 语法
- **"How to"** 分类主题

How to 平衡括号和大括号

Mathematica 中的所有括号必须平衡（即成对出现）. 也就是说，每种类型的开括号必须由相应的右括号平衡. 如果出现不平衡情况，那么 *Mathematica* 不会对单元进行计算. 在 *Mathematica* 前端包含几个方便的工具，让您确保括号和大括号是平衡的.



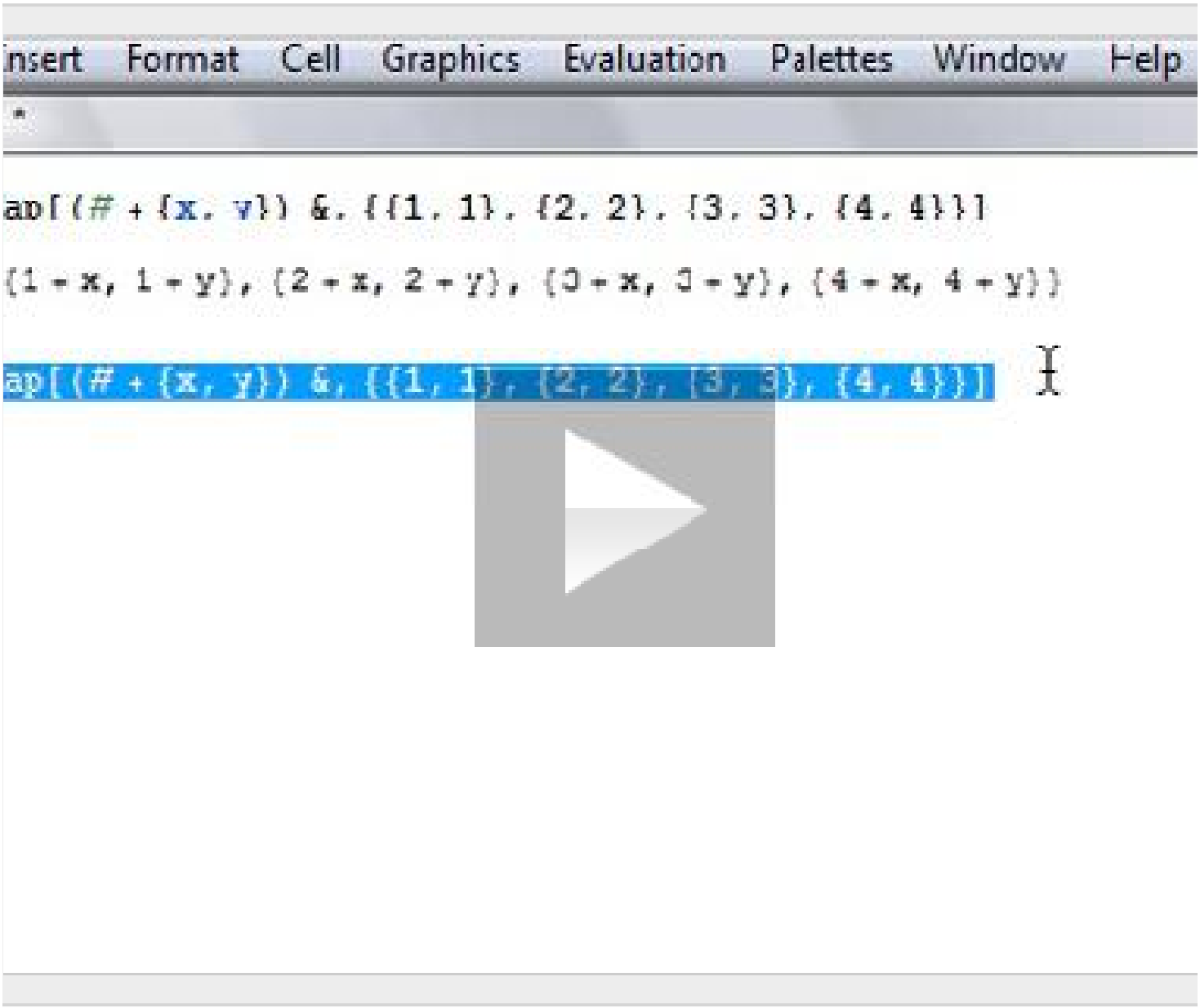
这三种类型的括号是括号 **( )**、大括号 **{ }** 和方括号 **[ ]**. 前两种有时也称作圆括号和花括号.

如果一个括号不平衡，***Mathematica*** 前端将其显示为紫色：

```
{1, 5, 9
```

如果您想要计算一个括号不平衡的表达式，***Mathematica*** 将生成一则错误信息，并高亮显示错误出现的位置. 最右端的单元括号也被高亮显示；并显示一个符号 **+**，对其单击可以得到更多关于错误的信息. 在这里，没有更多的信息可用，因此单击 **+** 没有什么作用：

```
In[1]:= {1, 5, 9}
```



这三种类型的括号是括号 **( )**、大括号 **{ }** 和方括号 **[ ]**. 前两种有时也称作圆括号和花括号.

如果一个括号不平衡，***Mathematica*** 前端将其显示为紫色：

```
{1, 5, 9
```

如果您想要计算一个括号不平衡的表达式，***Mathematica*** 将生成一则错误信息，并高亮显示错误出现的位置. 最右端的单元括号也被高亮显示；并显示一个符号 **+**，对其单击可以得到更多关于错误的信息. 在这里，没有更多的信息可用，因此单击 **+** 没有什么作用：

```
In[1]:= {1, 5, 9}
```

当您输入一个右括号，***Mathematica*** 前端会使与其配对的开括号变为黑色，并使其瞬时高亮显示. 这有助于您看到哪一对括号符号刚刚平衡：

```
{1, 5, 9}
```

这里，方括号和列表括号都不闭合. 闭合的方括号错误地替代了闭合的列表括号：

```
Plot[Sin[x], {x, 1, 10]
```

```
In[1]:= Plot[Sin[x], {x, 1, 10]]
```

这些看似简单的错误很容易发生在您键入长表达式时：

```
In[1]:= Map[(# + {x, y}) &,
          {{1, 1}, {2, 2}, {3, 3}, {4, 4}]
```

Mathematica 前端包含一个有效应对这种情况的菜单项. 将光标放在不平衡的表达式中，进入 **编辑** ▸ **扩展选择**. 最相邻的配对括号的内容现在得以选择. 这也可以通过双击不平衡的表达式实现：





重复使用 **编辑** ▸ **扩展选择** 或重复点击进一步将选择扩展到下一个最相邻的配对括号. 您可以迅速注意到哪个括号是不平衡的：





```
Map[(# + {x, y}) &, {{1, 1}, {2, 2}, {3, 3}, {4, 4}]
```

您也可以使用 **编辑** ▸ **检查括号平衡** 来选择最相邻的一对平衡括号：

```
Map[(# + {x, y}) &, {{1, 1},|{2, 2}, {3, 3}, {4, 4}]
```

```
Map[(# - {x, y}) &, {{1, 1}, {2, 2}, {3, 3}, {4, 4}]
```

使用 **编辑** ▸ **检查括号平衡** 时，光标的位置很重要。当光标位于一个未平衡的括号之前时，它是不会工作的，因为没有平衡的括号集合可供选择：

```
Map[(# + {x, y}) &,|{{1, 1}, {2, 2}, {3, 3}, {4, 4}]
```

三击一个函数头部会将选择扩展从而确定该函数的范围。这也可以通过将光标置于函数头部的某处，按住 Ctrl + . 两次完成：

```
Manipulate[Plbt[Sin[x], {x, 0, r}], {r, 1, 22}]
```

```
Manipulate[Plot[Sin[x], {x, 0, r}], {r, 1, 22}]
```

```
Manipulate[Plot[Sin[x], {x, 0, r}], {r, 1, 22}]
```

类似地，三击任何分隔符，包括引号在内，将选择它的范围：

```
Manipulate[Plot[Sin[x], {x, 0, r}], {r, 1, 22}]
```

```
Manipulate[Plot[Sin[x], {x, 0, r}], {r, 1, 22}]
```

```
{"this is a", "sample string that I am", "typing"}
```

教程

- ***Mathematica*** 的四种括号
- 一些普遍的记号和传统表示
- ***Mathematica*** 语言的语法
- 输入语法
- 运算序列

相关链接

- **How to:** 正确使用括号和大括号
- **How to:** 使用 ***Mathematica*** 的语法
- **How to:** 使用列表
- **How to:** 使用变量和函数

更多关于

- ***Mathematica*** 语法
- **"How to"** 分类主题

How to 使用简写符号

简写符号是 *Mathematica* 丰富的语法体系的一部分，允许用多种方式为函数赋以参数. 使用简写符号不仅能够使代码紧凑，还使您能在 *Mathematica*中自定义工作流程.

Mathematica 提供了多种便捷方法，用于输入函数.

度：

```
In[1]:= Length[{1, 2, 4, 6, 5, 8}]
Out[1]= 6
```

在函数与列表之间使用 @ 符号可以完成同样的任务。使用 @，您不必移到表达式的末端来匹配方括号：

```
In[2]:= Length @ {1, 2, 4, 6, 5, 8}
Out[2]= 6
```

还可以用后缀符号 //：

```
In[3]:= {1, 2, 4, 6, 5, 8} // Length
Out[3]= 6
```

这些符号可扩展至任何函数及任何类型的参数：

```
In[4]:= f@{1, 2, 4, 6, 5, 8}
Out[4]= f[{1, 2, 4, 6, 5, 8}]

In[5]:= 1 // AtomQ
Out[5]= True

In[6]:= x + 1 // g
Out[6]= g[1 + x]
```

对于含两个参数的函数，您可以使用中缀符号：

```
In[7]:= a ~ f ~ b
Out[7]= f[a, b]
```

纯函数在 ***Mathematica*** 中使用得非常频繁. 它们的存在令您无需为一个函数定义明确的名称就可使用它. 您可以使用 `Function` 的简写符号给出一个纯函数.

使用 `Function` 给出一个纯函数，对输入进行立方运算：

```
In[8]:= Function[x, x^3][3]
Out[8]= 27
```

符号 # 和 & 可以组合使用完成同一任务. 符号 # 用作变量的占位符，而符号 & 放在函数的代入值之前：

```
In[9]:= (#^3) &[3]
Out[9]= 27
```

`Map` 和 `Apply` 对于函数式编程非常重要. 使用这些函数的简写符号会带来很大方便.

使用 `Map` 将一个函数映射到一个列表中的各元素：

```
In[10]:= Map[f, {1, 2, 4, 6, 5, 8}]
Out[10]= {f[1], f[2], f[4], f[6], f[5], f[8]}
```

/@ 可实现同一目的.

```
In[7]:= a ~ f ~ b
Out[7]= f[a, b]
```

纯函数在 ***Mathematica*** 中使用得非常频繁. 它们的存在令您无需为一个函数定义明确的名称就可使用它. 您可以使用 `Function` 的简写符号给出一个纯函数.

使用 `Function` 给出一个纯函数，对输入进行立方运算：

```
In[8]:= Function[x, x^3][3]
Out[8]= 27
```

符号 # 和 & 可以组合使用完成同一任务. 符号 # 用作变量的占位符，而符号 & 放在函数的代入值之前：

```
In[9]:= (#^3) &[3]
Out[9]= 27
```

`Map` 和 `Apply` 对于函数式编程非常重要. 使用这些函数的简写符号会带来很大方便.

使用 `Map` 将一个函数映射到一个列表中的各元素：

```
In[10]:= Map[f, {1, 2, 4, 6, 5, 8}]
Out[10]= {f[1], f[2], f[4], f[6], f[5], f[8]}
```

/@ 可实现同一目的.

```
In[11]:= f /@ {1, 2, 4, 6, 5, 8}
Out[11]= {f[1], f[2], f[4], f[6], f[5], f[8]}
```

Apply 也有一个简写符号（@@）：

```
In[12]:= Apply[f, {1, 2, 4, 6, 5, 8}]
Out[12]= f[1, 2, 4, 6, 5, 8]

In[13]:= f @@ {1, 2, 4, 6, 5, 8}
Out[13]= f[1, 2, 4, 6, 5, 8]
```

Mathematica 中的简写符号可以组合使用，生成高效代码.

使用 Map 与 Function 可将列表中的每个元素进行幂乘，并给结果加上一个符号：

```
In[14]:= Map[Function[x, x^3 + a], {1, 2, 4, 6, 5, 8}]
Out[14]= {1 + a, 8 + a, 64 + a, 216 + a, 125 + a, 512 + a}
```

使用 Function 的简写符号执行同一运算：

```
In[15]:= Map[#^3 + a &, {1, 2, 4, 6, 5, 8}]
Out[15]= {1 + a, 8 + a, 64 + a, 216 + a, 125 + a, 512 + a}
```

对此扩展，将 Map 的简写符号也包括进来：

```
In[16]:= (#^3 + a) & /@ {1, 2, 4, 6, 5, 8}
Out[16]= {1 + a, 8 + a, 64 + a, 216 + a, 125 + a, 512 + a}
```

```
In[11]:= f /@ {1, 2, 4, 6, 5, 8}
Out[11]= {f[1], f[2], f[4], f[6], f[5], f[8]}
```

Apply 也有一个简写符号（@@）：

```
In[12]:= Apply[f, {1, 2, 4, 6, 5, 8}]
Out[12]= f[1, 2, 4, 6, 5, 8]

In[13]:= f @@ {1, 2, 4, 6, 5, 8}
Out[13]= f[1, 2, 4, 6, 5, 8]
```

Mathematica 中的简写符号可以组合使用，生成高效代码.

使用 Map 与 Function 可将列表中的每个元素进行幂乘，并给结果加上一个符号：

```
In[14]:= Map[Function[x, x^3 + a], {1, 2, 4, 6, 5, 8}]
Out[14]= {1 + a, 8 + a, 64 + a, 216 + a, 125 + a, 512 + a}
```

使用 Function 的简写符号执行同一运算：

```
In[15]:= Map[#^3 + a &, {1, 2, 4, 6, 5, 8}]
Out[15]= {1 + a, 8 + a, 64 + a, 216 + a, 125 + a, 512 + a}
```

对此扩展，将 Map 的简写符号也包括进来：

```
In[16]:= (#^3 + a) & /@ {1, 2, 4, 6, 5, 8}
Out[16]= {1 + a, 8 + a, 64 + a, 216 + a, 125 + a, 512 + a}
```

您往往需要在一个新的计算中使用前面的输出结果. 这可以用 Out 的简写符号 % 实现.

设置一个计算:

```
In[17]:= {RandomReal[], RandomReal[], RandomReal[]}
Out[17]= {0.19069, 0.785019, 0.99944}
```

使用 Out 的简写符号 % 指定最新的输出:

```
In[18]:= %
Out[18]= {0.19069, 0.785019, 0.99944}
```

联合使用 % 与 Part 的简写符号 [[...]], 取列表中的第一个元素:

```
In[19]:= %[[1]]
Out[19]= 0.19069
```

可以使用 %%...% 表示先前的输出. 得到两个计算生成之前的输出:

```
In[20]:= %%
Out[20]= {0.19069, 0.785019, 0.99944}
```

如果想要得到的输出并非来自最近的计算, 则需要使用多个 % 符号, 这可能会比较繁琐.

这种情况下可以将特定的输出单元号与 Out 合用. 如果您运行该单

您往往需要在一个新的计算中使用前面的输出结果. 这可以用 Out 的简写符号 % 实现.

设置一个计算:

```
In[17]:= {RandomReal[], RandomReal[], RandomReal[]}
Out[17]= {0.19069, 0.785019, 0.99944}
```

使用 Out 的简写符号 % 指定最新的输出:

```
In[18]:= %
Out[18]= {0.19069, 0.785019, 0.99944}
```

联合使用 % 与 Part 的简写符号 [[...]], 取列表中的第一个元素:

```
In[19]:= %[[1]]
Out[19]= 0.19069
```

可以使用 %%...% 表示先前的输出. 得到两个计算生成之前的输出:

```
In[20]:= %%
Out[20]= {0.19069, 0.785019, 0.99944}
```

如果想要得到的输出并非来自最近的计算, 则需要使用多个 % 符号, 这可能会比较繁琐.

这种情况下可以将特定的输出单元号与 Out 合用. 如果您运行该单

元，您将从当前的笔记本进程得到 Out[17]，并不一定与下面的所示相同：

```
In[28]:= Out[17]
Out[28]= {0.19069, 0.785019, 0.99944}
```

这是另一个简写符号：

```
In[29]:= %17
Out[29]= {0.19069, 0.785019, 0.99944}
```

通过标签或简写符号表示先前的输出虽然方便，但可能会很快失控，因为当前的运算始终受约于较早的输出. 因此，您必须确保您要使用的输出可用于您当前的计算. 应谨慎使用此表示法.

通过字符串操作函数的简写符号，可以简化字符串的使用.

连接字符串是一种常用的字符串运算. 用 StringJoin 实现：

```
In[30]:= StringJoin["This is ", "Mathematica."]
Out[30]= This is Mathematica.
```

使用 StringJoin 的简写符号 <>，可将同一运算写作：

```
In[31]:= "This is " <> "Mathematica."
Out[31]= This is Mathematica.
```

StringExpression 是用于表示字符串的一个非常重要的函数.

元，您将从当前的笔记本进程得到 Out[17]，并不一定与下面的所示相同：

```
In[28]:= Out[17]
Out[28]= {0.19069, 0.785019, 0.99944}
```

这是另一个简写符号：

```
In[29]:= %17
Out[29]= {0.19069, 0.785019, 0.99944}
```

通过标签或简写符号表示先前的输出虽然方便，但可能会很快失控，因为当前的运算始终受约于较早的输出. 因此，您必须确保您要使用的输出可用于您当前的计算. 应谨慎使用此表示法.

通过字符串操作函数的简写符号，可以简化字符串的使用.

连接字符串是一种常用的字符串运算. 用 StringJoin 实现：

```
In[30]:= StringJoin["This is ", "Mathematica."]
Out[30]= This is Mathematica.
```

使用 StringJoin 的简写符号 <>，可将同一运算写作：

```
In[31]:= "This is " <> "Mathematica."
Out[31]= This is Mathematica.
```

StringExpression 是用于表示字符串的一个非常重要的函数.

StringCases、StringSplit 和 StringMatchQ 等.

使用 StringExpression 创建一个字符串表达式对象：

```
In[25]:= StringExpression["ab", _]
Out[25]= ab ~~ _
```

或直接使用 StringExpression 的简写符号 ~~：

```
In[26]:= "ab" ~~ _
Out[26]= ab ~~ _
```

教程

- 表达式
- 表达式输入的特殊方式
- ***Mathematica*** 语言的语法
- 函数操作
- 字符串和字符

相关链接

- **How to:** 使用 ***Mathematica*** 的语法
- **How to:** 使用变量和函数

参见

Function · Map · Apply · Out · Part · StringJoin · StringExpression

更多关于

- ***Mathematica*** 语法
- ***Mathematica*** 语法符号
- 函数式编程
- 字符串操作
- **"How to"** 分类主题

How to 输入函数的范围和选项

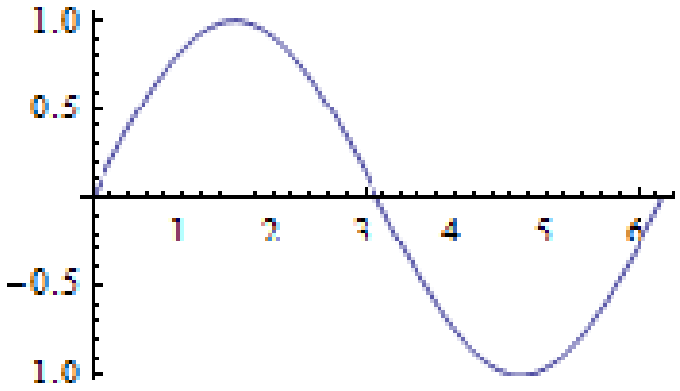
一个 *Mathematica* 内置函数所执行的主要表达式或对象作为函数的第一个参数给出. 作为语法的一部分， *Mathematica* 内置函数可以有多个自变量，这可能要求或可能成为该函数的推广或延伸. 参数之后是选项，用于进一步扩展以控制函数的行为.

Mathematica 中的许多可视化函数要求用户输入绘图界限，作为绘图函数的第二个参数.

这里，`Sin[x]` 是 `Plot` 中的第一个参数，第二个参数 `{x, 0, 2π}` 给出变量以及图形范围：

```
In[7]:= Plot[Sin[x], {x, 0, 2 π}]
```

Out[7]=



如果使用 Plot 时不指定范围，***Mathematica*** 会产生一则错误提示信息：

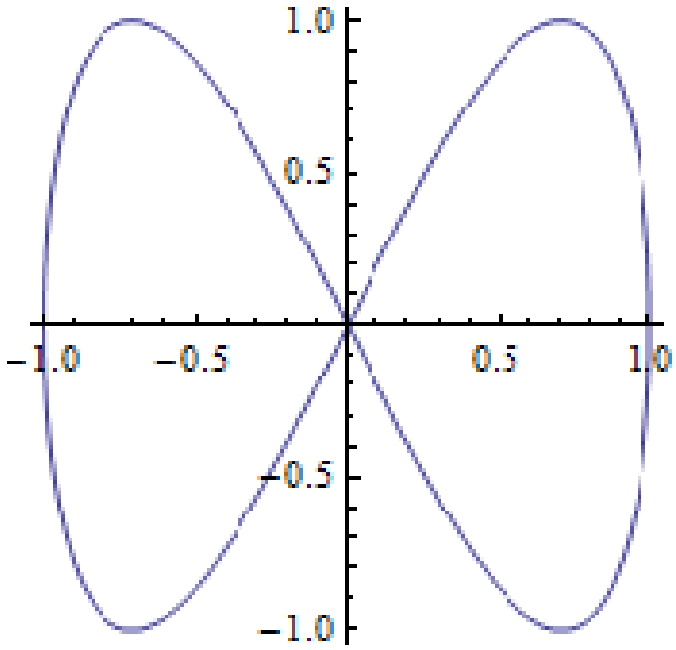
```
In[2]:= Plot[Sin[x], {x}]

          Plot::pllim :
           Range specification {x} is not of the form
               {x, xmin, xmax}. ≫

Out[2]= Plot[Sin[x], {x}]
```

其它可视化函数具有类似的关于绘图范围的要求.

这里，一条参数曲线在 0 到 2 π 之间绘制. 这些界限作为 ParametricPlot 的第二参数指定:

```
In[5]:= ParametricPlot[{Sin[u], Sin[2 u]}, {u, 0, 2 π}]
```

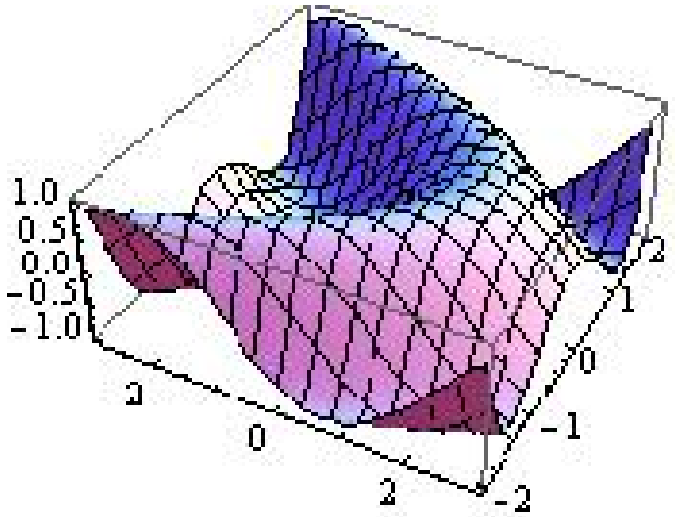
Out[5]=



在绘制两个变量的函数时，每个变量的范围分别作为第二个和第三个参数输入：

```
In[6]:= Plot3D[Sin[x + y^2], {x, -3, 3}, {y, -2, 2}]
```

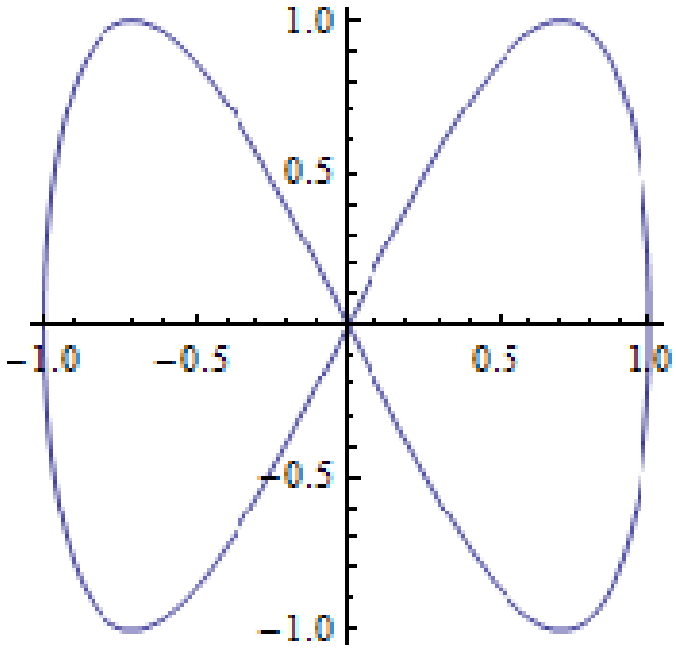
Out[6]=



要了解一个可视化函数的确切语法，请参见参考资料中心的相关页面。关于如何查找参考资料中心的函数页面，请参见**How to:** 查找函数的相关信息。

```
In[5]:= ParametricPlot[{Sin[u], Sin[2 u]}, {u, 0, 2 π}]
```

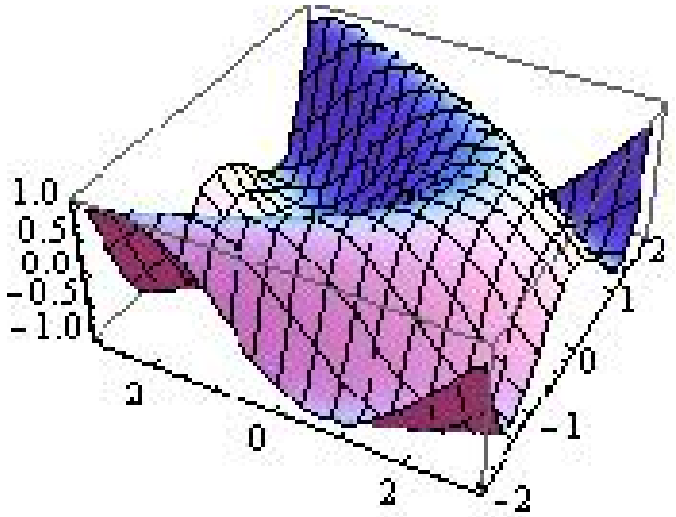
Out[5]=



在绘制两个变量的函数时，每个变量的范围分别作为第二个和第三个参数输入：

```
In[6]:= Plot3D[Sin[x + y^2], {x, -3, 3}, {y, -2, 2}]
```

Out[6]=



要了解一个可视化函数的确切语法，请参见参考资料中心的相关页面。关于如何查找参考资料中心的函数页面，请参见**How to:** 查找函数的相关信息。

过函数选项来对其行为进行准确控制.

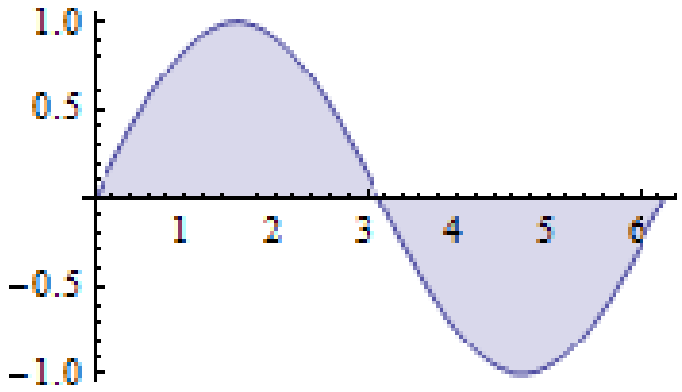
选项使用规则输入. 一个规则的缩写形式使用的是一个右箭头，这可以通过键入 -> （- 与 > 之间无空格）得到. 在进一步键入时，*Mathematica* 前端自动将 -> 转换为 →. 每个符号都是 Rule 的一种缩写形式.

在输入函数选项时，选项在规则的左边，其设置在右边. 选项永远位于函数要求的所有参数之后.

将 Plot 与 Filling 联合使用创建一个填充图形：

```
In[8]:= Plot[Sin[x], {x, 0, 2 π}, Filling → Axis]
```

Out[8]=



从 *Mathematica* 参考资料范例数据中导入某些数据，并将其在一个 Grid 中显示：

```
In[9]:= data =
          Import["ExampleData/classification.tsv"];
```

In[10]:=
```
Grid[data]
```

Out[10]=

Rank	Fruit Fly	Human	Pea	E.coli
Domain	Eukaryota	Eukaryota	Eukaryota	Bacteri
Kingdom	Animalia	Animalia	Plantae	Monera
Phylum	Arthropoda	Chordata	Magnoliophyta	Proteobacteria
Class	Insecta	Mammalia	Magnoliopsida	Proteobacteria
Order	Diptera	Primates	Fabales	Enterobacteriales
Family	Drosophilidae	Hominidae	Fabaceae	Enterobacteriaceae
Genus	Drosophila	Homo	Pisum	Escherichia
Species	D.melanogaster	H.sapiens	P.sativum	E.coli

将 Grid 与 Frame 选项联合使用：

In[11]:=
```
Grid[data, Frame → All]
```

Out[11]=

Rank	Fruit Fly	Human	Pea	E.coli
Domain	Eukaryota	Eukaryota	Eukaryota	Bacteria
Kingdom	Animalia	Animalia	Plantae	Monera
Phylum	Arthropoda	Chordata	Magnoliophyta	Proteobacteria
Class	Insecta	Mammalia	Magnoliopsida	Proteobacteria
Order	Diptera	Primates	Fabales	Enterobacteriales
Family	Drosophilidae	Hominidae	Fabaceae	Enterobacteriaceae
Genus	Drosophila	Homo	Pisum	Escherichia
Species	D.melanogaster	H.sapiens	P.sativum	E.coli

您也可以一起使用多个选项，与参数不同，这些选项可以任意次序排列：

```
Grid[data, Alignment → Left, Spacings → {2, 1},
 Frame → All, ItemStyle → "Text",
 Background → {{Gray, None}, {LightGray, None}}]
```

"Rank"	"Fruit Fly"	"Human"	"Pea"	"E.coli"
"Domain"	"Eukaryota"	"Eukaryota"	"Eukaryota"	"Bacteria"
"Kingdom"	"Animalia"	"Animalia"	"Plantae"	"Monera"
"Phylum"	"Arthropoda"	"Chordata"	"Magnoliophyta"	"Proteobacteria"
"Class"	"Insecta"	"Mammalia"	"Magnoliopsida"	"Proteobacteria"
"Order"	"Diptera"	"Primates"	"Fabales"	"Enterobacteriales"
"Family"	"Drosophilidae"	"Hominidae"	"Fabaceae"	"Enterobacteriace"

```
Grid[data, Alignment → Left, Spacings → {2, 1},
 Frame → All, ItemStyle → "Text",
 Background → {{Gray, None}, {LightGray, None}}]
```

"Rank"	"Fruit Fly"	"Human"	"Pea"	"E.coli"
"Domain"	"Eukaryota"	"Eukaryota"	"Eukaryota"	"Bacteria"
"Kingdom"	"Animalia"	"Animalia"	"Plantae"	"Monera"
"Phylum"	"Arthropoda"	"Chordata"	"Magnoliophyta"	"Proteobacteria"
"Class"	"Insecta"	"Mammalia"	"Magnoliopsida"	"Proteobacteria"
"Order"	"Diptera"	"Primates"	"Fabales"	"Enterobacteriales"
"Family"	"Drosophilidae"	"Hominidae"	"Fabaceae"	"Enterobacteriace"

"Genus"	"Drosophila"	"Homo"	"Pisum"	"Escherichia"
"Species"	"D.melanogaster"	"H.sapiens"	"P.sativum"	"E.coli"

可用选项取决于所使用的函数. 要知道一个函数可与哪些选项结合使用，请参见 **How to:** 查找可用选项.

教程

- 一些普遍的记号和传统表示
- ***Mathematica*** 语言的语法
- 输入语法
- ***Mathematica*** 的四种括号
- 处理选项
- 从 ***Mathematica*** 中获取信息

相关链接

- **How to:** 查找可用选项
- **How to:** 查找函数的相关信息
- **How to:** 使用 ***Mathematica*** 的语法

参见

Rule ▪ **Options** ▪ **AbsoluteOptions** ▪ **SetOptions** ▪

更多关于

- 语法
- 选项管理
- 图形选项和样式
- **"How to"** 分类主题

How to 使用函数模板

您可以在 *Mathematica* 中插入函数模板，模板中包含函数参数的占位符. 这可以直接通过键盘输入，也可以通过 *Mathematica* 的几个内置面板实现. 使用模板输入函数的参数有助于确保所输入的语法正确. 模板还减少了您离开笔记本去参考资料中心查看有关函数信息的次数.

如果在 **PC** 上通过键盘插入函数模板，要先键入函数，然后键入 **Ctrl** + **Shift** + **K**. 如果使用的是 **Macintosh**，则键入 **Cmd** + **Shift** + **K**. 它们是菜单项 **编辑** ▸ **制作模板** 的快捷键.

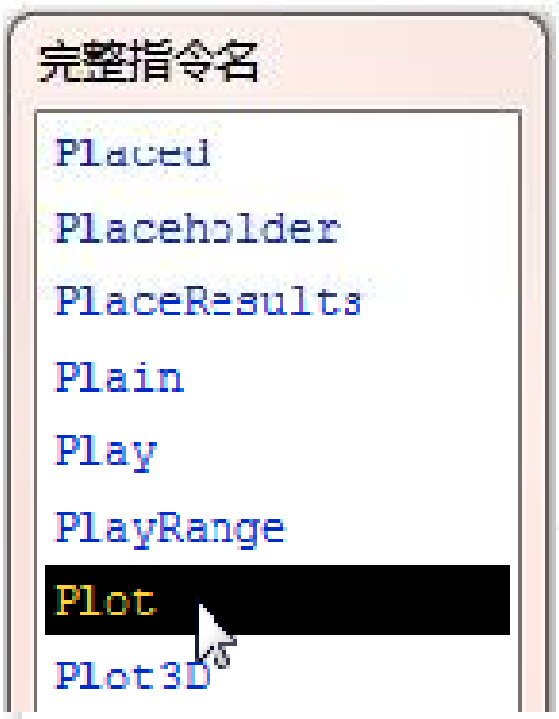
此处，用键盘插入 `Plot` 的模板. 请注意 `Plot` 的第一个参数的占位符被自动选定：

$$\texttt{Plot}[f, \{x, x_{min}, x_{max}\}]$$

插入函数模板还用于当键入了函数名的一部分，然后从使用快捷键弹出的菜单中选择想要的函数时.

要在键入位置插入一个函数模板，双击想要输入的函数（**Mac OS X** 上要单击），或用箭头键在列表上定位，当它突出显示时，按 **Enter** 键：

```
Pl
```



您所得到的内置函数的模板通常是多个可能参数形式中最简单的一种. 如要查看一个函数的所有参数形式，请参阅其在 **参考资料中心** 的页面. 关于如何查看有关函数参考资料的详细说明，请参阅 **How to**：查找函数的相关信息.

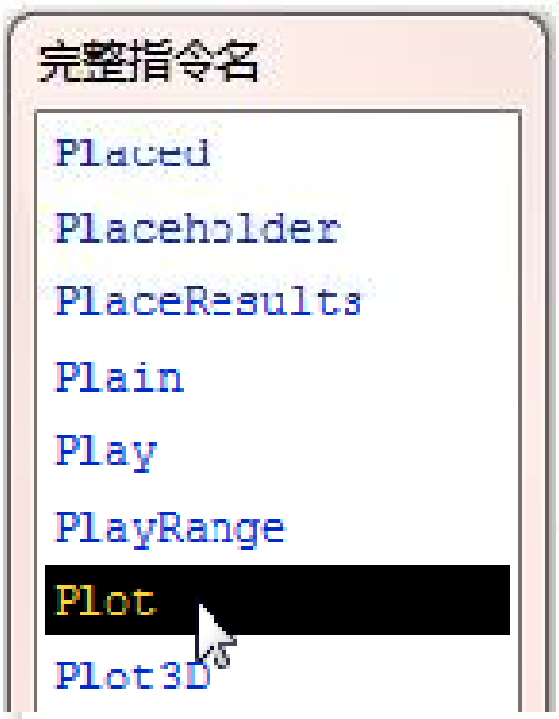
还可以通过 ***Mathematica*** 的许多内置面板插入函数模板. 您可以通过 **面板**菜单来访问这些面板.

从 **面板**菜单打开 **数学助手** 面板，打开该面板的 **基本指令** 部分，然后

插入函数模板还用于当键入了函数名的一部分，然后从使用快捷键弹出的菜单中选择想要的函数时.

要在键入位置插入一个函数模板，双击想要输入的函数（**Mac OS X** 上要单击），或用箭头键在列表上定位，当它突出显示时，按 **Enter** 键：

```
Pl
```

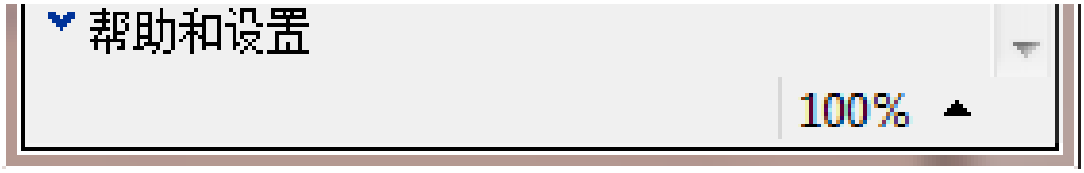


您所得到的内置函数的模板通常是多个可能参数形式中最简单的一种. 如要查看一个函数的所有参数形式，请参阅其在 **参考资料中心** 的页面. 关于如何查看有关函数参考资料的详细说明，请参阅 **How to**：查找函数的相关信息.

还可以通过 ***Mathematica*** 的许多内置面板插入函数模板. 您可以通过 **面板**菜单来访问这些面板.

从 **面板**菜单打开 **数学助手** 面板，打开该面板的 **基本指令** 部分，然后


数学助手
计算器
基本指令
列表
2D
3D
可视化函数
Plot
更多
可视化数据
ListPlot
更多
图表
BarChart
更多
图表元素 ..
图表选项
可视化矢量场
VectorPlot
更多
选项：自动指定
范围
样式
刻度
网格
坐标轴 & 边框
其它
单位
特有的
组合图形对象
Graphics
Show
指令
基元
颜色名称
互动式工具
Manipulate
←选项
Manipulator 控制
←选项
绘图工具...



这里用 **数学助手** 面板输入 `Plot` 的模板：

```
In[4]:= Plot[function, {var, min, max}]
```

您还可以访问 **数学助手** 面板的其它部分的函数模板.

教程

- 键盘快捷键列表
- 字母和类似字母的形式
- 二维表达式的输入
- 笔记本的输入和输出

相关链接

- **How to:** 查找函数的相关信息
- **How to:** 使用 ***Mathematica*** 的语法
- **How to:** 在 ***Mathematica*** 中获取帮助
- **How to:** 在笔记本中进行内容的选择与键入

更多关于

- 编辑菜单
- 面板菜单
- 笔记本中的选择与输入
- 笔记本快捷键

- **"How to"** 分类主题

建立计算

使用前面的结果

定义变量

符号的值

Mathematica 的四种括号

运算序列

教程专集

- **Core Language**

使用前面的结果

在计算中，常常要用前面已得到的结果，在 ***Mathematica*** 中，使用 %（百分号）代表前面的最后一个结果.

%	前面的最后一个结果
%%	前面的倒数第二个结果
%% … % $(k$ 个$)$	前面的倒数第 k 个结果
% n	第n个输出行 Out $[n]$ 的结果（需要小心使用）

使用前面结果的方法.

这里是第一个结果.

```
In[1]:= 77^2
Out[1]= 5929
```

上一结果加1.

```
In[2]:= % + 1
Out[2]= 5930
```

使用前面两个结果.

```
In[3]:= 3 % + %^2 + %%
Out[3]= 35 188 619
```

Mathematica 中输入输出行都被标以数字号码. 可以使用标号引用前面的结果.

第 2 输出行和第 3 输出行的结果之和.

```
In[4]:= %2 + %3
Out[4]= 35 194 549
```

顺序出现. 然而，如果使用 "使用笔记本界面" 讨论的 ***Mathematica*** 笔记本界面，那么输入输出行则不必按顺序出现. 例如，您可以“回滚”到任一地方插入一个算式. 要注意，% 总是代表 ***Mathematica*** 生成的前面最后一个结果. 它不一定是您当前所处位置的上一个结果. 在笔记本界面中，使用特定结果的方法是利用 `Out[n]` 标号. 因此，在笔记本中可以在任意位置上进行插入和删除. 故而，结果的文本顺序与结果的顺序没有关系.

相关指南

- 管理笔记本中的计算
- 笔记本基础

相关教程

- 建立计算

教程专集

- **Core Language**

定义变量

在做较长的计算时，对中间的结果赋以一个名字会带来方便. 如同标准数学或其它计算机语言一样，引入已命名的变量来实现这一点.

这里设置 变量 x 的值为 5.

```
In[1]:= x = 5
Out[1]= 5
```

不论 x 何时出现， ***Mathematica*** 都给其赋值为 5.

```
In[2]:= x^2
Out[2]= 25
```

这里重新设置 x 的值.

```
In[3]:= x = 7 + 4
Out[3]= 11
```

设置 pi 为 **40** 位有效数字的 π.

```
In[4]:= pi = N[Pi, 40]
Out[4]= 3.141592653589793238462643383279502884197
```

显示定义的 pi 值.

```
In[5]:= pi
Out[5]= 3.141592653589793238462643383279502884197
```

计算 π^2 的数值解，精度与 pi 一样.

```
In[6]:= pi^2
Out[6]= 9.869604401089358618834490999876151135 31
```

x=*value*	给 *x* 赋值
x=*y*=*value*	给 *x* 和 *y* 赋值
x=. 或 Clear[*x*]	清除 *x* 的值

给变量赋值.

注意，赋给变量的值具有*永久性*. 一旦给一个变量赋了值，这个值将一 直保持到明确清除它为止. 当然，如果启动一个新的 ***Mathematica*** ，这个值将不存在.

在使用 ***Mathematica*** 时，忘记你前面已设置的变量是产生错误的一个常见原因. 如果设置 x = 5， ***Mathematica*** 认为 x 的值总是 5, 直到或除非您明确告诉它不是. 为了避免错误，当使用完设置的变量后，应立即清除该变量的值.

- 用完所设置的变量后，就立即清除该变量的值.

使用 ***Mathematica*** 的一个原则.

可以用任何名称定义变量，名称长度没有限制. 但变量名不能以数字开头. 例如： x2 可以作为变量， 但 2 x 的意思是 2 * x.

Mathematica 使用大写和小写字母. 有一个规则： ***Mathematica*** 的内置对象总是以大写字母开头. 为了避免混淆，当选择自定义的变量名时应以小写字母开头.

aaaaa	只包含小写字母的变量名
Aaaaa	大写字母开头的内置对象名

命名规则.

在 ***Mathematica*** 可以像在数学中一样，输入包含变量的式子， 但有几个要点须注意：

- x y 表示 x 乘 y.
- 没有空格的 xy 表示变量 xy.
- 5 x 表示 5 乘 x.
- x^2 y 表示（x^2）y， 不是 x^（2 y）.

在 ***Mathematica*** 中， 使用变量应注意的事项.

相关指南

- 管理笔记本中的计算

相关教程

- 建立计算
- 使用变量和函数

教程专集

- **Core Language**

符号的值

当 ***Mathematica*** 把表达式 x + x 变换成 2 x 时，它是把 x 作为纯符号或形式的式子进行处理. 在这种情形下，x 是一个可以代表任何表达式的符号.

然而，常常需要给类似 x 的符号赋一个确定的“值”. 这个值有时是一个数；更多的是另一个表达式.

要对表达式 1 + 2 x 中的符号 x 赋一个确定的值，用户可以建立一个 ***Mathematica*** 变换规则，然后把这个规则用于该表达式. 例如，把 x 替换成 3，用户应建立变换规则 x -> 3. 您必须把 -> 作为一对字符输入，中间没有任何空格. x -> 3 这条规则的含义是“x 取值为 3”.

要对一个特定的 ***Mathematica*** 表达式进行变换，使用 *expr* /. *rule*. “替换符号” /. 是由两个字符组成，中间没有空格.

在表达式 1 + 2 x 中使用变换规则 x -> 3.

```
In[1]:= 1 + 2 x /. x -> 3
Out[1]= 7
```

可以用任何表达式替换 x. 这里用 2 - y 替换 x.

```
In[2]:= 1 + x + x^2 /. x -> 2 - y
Out[2]= 3 + (2 - y)^2 - y
```

这里给出一个变换规则，***Mathematica*** 把它作为符号表达式处理.

```
In[3]:= x -> 3 + y
Out[3]= x → 3 + y
```

这里使用上述规则，对表达式 x^2 - 9 进行变换.

```
In[4]:= x^2 - 9 /. %
Out[4]= -9 + (3 + y)^2
```

expr /. *x*->*value*	用 *value* 替换表达式 *expr* 中的 *x*
expr /. {*x*->*xval*, *y*->*yval*}	进行多个替换

用值替换表达式中的符号.

可以把多个替换规则写成列表一起使用.

```
In[5]:= (x + y) (x - y)^2 /. {x -> 3, y -> 1 - a}
```

Out[5]= $(4 - a)(2 + a)^2$

替换算符 /. 使用户可以对一个特定表达式进行变换. 然而，有时用户想要定义一个*总是* 被使用的变换规则. 例如，无论 x 何时出现，x 总是被 3 替换.

如同在 "定义变量" 讨论的那样，这可以通过 x = 3 给 x *赋* 以值 3 来实现. 一旦用户做了赋值 x = 3，无论 x 何时出现，x 总是被 3 替换.

这里给 x 赋以值 3.

```
In[6]:= x = 3
```

Out[6]= 3

现在 x 自动被 3 替换.

```
In[7]:= x^2 - 1
```

Out[7]= 8

这里给 x 赋以表达式 1 + a.

```
In[8]:= x = 1 + a
Out[8]= 1 + a
```

现在 x 被 1 + a 替换.

```
In[9]:= x^2 - 1
Out[9]= -1 + (1 + a)^2
```

用户可以把符号的值定义成任何表达式，而不仅是能定义成数. 应该注意，一旦给出了定义，这个定义将一直被使用，直到用户明确改变或消除该定义. 在使用 ***Mathematica*** 时，忘记清除已经赋给符号的值是常见的出错原因.

x=*value*	定义 *x* 的值，值将一直被使用
x=.	清除定义给 *x* 的值

符号赋值.

符号 x 仍然有前边已赋给它的值.

```
In[10]:= x + 5 - 2 x
Out[10]= 6 + a - 2 (1 + a)
```

这里清楚赋给 x 的值.

```
In[11]:= x =.
```

现在 x 没有被定义值，因此它被用作纯符号变量.

```
In[12]:= x + 5 - 2 x
Out[12]= 5 - x
```

在 ***Mathematica*** 中一个符号如 x 可以用作多种不同目的. 事实上，***Mathematica*** 的伸缩性主要来自于能够按意愿混合这些目的. 然而，为了避免出错，用户需要直接使用 x 的某种用途. 最重要的区别在于把 x 作为另一个表达式的名称和只代表本身的符号变量.

传统的程序语言并不支持符号计算. 只允许变量作为已经赋值的对象的名称. 然而，在 ***Mathematica*** 中，x 能被作为一个纯形式变量进行处理，可以对x使用各种变换规则. 当然如果用户明确给出一个定义，如 x = 3，那么 x 将总是被 3 代替，而不再作为形式变量.

应当明白，显示定义如 x = 3 有一个全局效应. 而替换如 *expr* /. x -> 3 仅对指定的表达式 *expr* 有效. 除非绝对有必要，一般应避免使用显式定义，这样，保持所做的事情是正确的就容易多了.

用户可以将替换和赋值混合使用. 运用赋值，可以对想要作替换的表达式或替换规则给出一个名称.

这里赋给符号 t 一个值.

```
In[13]:= t = 1 + x^2
Out[13]= 1 + x^2
```

这里用 2 替换 t 中的 x.

```
In[14]:= t /. x -> 2
Out[14]= 5
```

这里用不同的值替换 t 中 x 的值.

```
In[15]:= t /. x -> 5 a
Out[15]= 1 + 25 a^2
```

这里用 Pi 替换 t 中 x 的值，然后计算近似结果.

```
In[16]:= t /. x -> Pi // N
Out[16]= 10.8696
```

相关指南

- 管理笔记本中的计算

相关教程

- 代数计算

教程专集

- **Core Language**
- **Mathematics and Algorithms**

Mathematica 的四种括号

在 ***Mathematica*** 中使用四种括号. 每一种括号都有完全不同的含意. 记住这些括号的含意是很重要的.

(*term*)	表示组合的圆括号
f[*x*]	表示函数的方括号
{*a*, *b*, *c*}	表示列表的花括号
v[[*i*]]	表示索引的双括号 （Part[*v*, *i*]）

Mathematica 的四种括号.

当输入的表达式较为复杂时，在括号之间留出一些间隔是比较好的. 这可以更容易分清相匹配的括号对. 例如， *v*[[{*a*, *b*}]] 比 *v*[[{*a*, *b*}]] 更容易辨认.

相关指南

- 管理笔记本中的计算

相关教程

- 建立计算

教程专集

- **Core Language**

运算序列

使用 ***Mathematica*** 进行运算时，通常要进行一系列运算步骤. 如果您愿意，可分行输入每一步骤. 然而，把几个步骤放在同一行往往是

$expr_1; expr_2; expr_3$	进行几个运算，并输出最后一个的结果
$expr_1; expr_2;$	进行几个运算，但不输出结果

Mathematica 中做运算序列的方法.

此处做三个运算，并输出最后一个运算结果.

```
In[1]:= x = 4; y = 6; z = y + 6
Out[1]= 12
```

如果输入以分号结尾（这表明用户可能给出一个运算序列，结尾为“空”），其效果是：***Mathematica*** 执行该运算，但并不显示输出.

$expr;$	进行一个运算，但不显示输出

禁止输出.

在输入的末尾使用分号，告诉 ***Mathematica*** 不显示输出.

```
In[2]:= x = 67 - 5;
```

仍然可以用 % 得到输出.

```
In[3]:= %
Out[3]= 62
```

相关指南

- 管理笔记本中的计算

相关教程

- 建立计算

教程专集

- **Core Language**

列表

产生对象列表

将一些对象放在一起

值表的生成

处理列表元素

向量和矩阵

获得列表的部分元素

检测和搜索列表元素

添加、删除和修改列表元素

列表的组合

作为集合的列表

重排列表

列表元素的分组和合并

产生对象列表

在做计算时，将一些对象放在一起作为单个实体处理是很方便的. 而列表 （***Lists***） 是 ***Mathematica*** 产生对象集合的一种方法. 后面将看到列表是 ***Mathematica*** 的非常重要和普遍的结构.

例如列表 {3, 5, 1} 是三个对象的集合，但在许多情况下，可以把它作为单一对象处理. 例如，可以一次对整个列表进行运算，或把整个列表赋为某个变量的值.

这里是三个数的列表.

```
In[1]:= {3, 5, 1}
Out[1]= {3, 5, 1}
```

这是列表中每个数的平方再加 1.

```
In[2]:= {3, 5, 1}^2 + 1
Out[2]= {10, 26, 2}
```

这是两个列表相应元素之差，此时两列表长度必须相同.

```
In[3]:= {6, 7, 8} - {3.5, 4, 2.5}
Out[3]= {2.5, 3, 5.5}
```

% 的值是整个列表.

```
In[4]:= %
Out[4]= {2.5, 3, 5.5}
```

"一些数学函数" 中的数学函数都可以使用到列表.

```
In[5]:= Exp[%] // N
Out[5]= {12.1825, 20.0855, 244.692}
```

如同把数赋给变量一样也可以把列表赋给某变量.

这里把列表赋给变量 v.

```
In[6]:= v = {2, 4, 3.1}
Out[6]= {2, 4, 3.1}
```

无论 v 出现在何处，它都用这个列表代替.

```
In[7]:= v / (v - 1)
Out[7]= {2, 4/3, 1.47619}
```

相关教程

- 建立计算

教程专集

- **Core Language**

将一些对象放在一起

在 "产生对象的列表" 节中我们已经讨论过列表，在那里是作为一种把多个数放在一起的方式. 在这一节中，将看到使用列表的许多不同方式. 用户将发现列表是 ***Mathematica*** 中最灵活和最强有力的对象之一. 将看到 ***Mathematica*** 列表是数学和计算机科学中的一些标准概念的一般化.

一般地，***Mathematica*** 列表本质上是提供一种把任何类型的表达式收集在一起的方法.

这是一个数的列表.

```
In[1]:= {2, 3, 4}
```

$$\text{Out[1]}= \{2, 3, 4\}$$

这里给出一个符号表达式的列表.

```
In[2]:= x^% - 1
```

$$\text{Out[2]}= \left\{-1 + x^2, -1 + x^3, -1 + x^4\right\}$$

可以对这些表达式求导数.

```
In[3]:= D[%, x]
```

$$\text{Out[3]}= \left\{2\,x, 3\,x^2, 4\,x^3\right\}$$

然后可以求出用 3 替换 x 所得的值.

```
In[4]:= % /. x -> 3
```

$$\text{Out[4]}= \{6, 27, 108\}$$

能分别作用于列表的每一个元素. 然而，并不一定***Mathematica*** 的所有函数都能做到这一点. 除非用户专门设置它. 用户引入新函数 f 把列表作为单个对象进行处理. "函数作用于表达式的部分项" 和 "结构的操作" 将介绍如何用 Map 和 Thread 使一个函数分别地作用于列表的每个元素.

相关教程

- 列表

教程专集

- **Core Language**

值表的生成

可以使用列表作为值的表. 用户可以生成这些表. 例如，通过对不同参数值的序列计算一个表达式来生成表.

这里给出 i 从**1**到**6**的 i^2 的值表.

```
In[1]:= Table[i^2, {i, 6}]
Out[1]= {1, 4, 9, 16, 25, 36}
```

这里是一个 n 从 0 到 4 的 $\sin(n/5)$ 的表.

```
In[2]:= Table[Sin[n / 5], {n, 0, 4}]
```

$$\text{Out[2]= } \left\{0, \text{Sin}\left[\frac{1}{5}\right], \text{Sin}\left[\frac{2}{5}\right], \text{Sin}\left[\frac{3}{5}\right], \text{Sin}\left[\frac{4}{5}\right]\right\}$$

这里给出近似值.

```
In[3]:= N[%]
Out[3]= {0., 0.198669, 0.389418, 0.564642, 0.717356}
```

也可以生成一个公式列表.

```
In[4]:= Table[x^i + 2 i, {i, 5}]
```

$$\text{Out[4]}= \left\{2 + x,\ 4 + x^2,\ 6 + x^3,\ 8 + x^4,\ 10 + x^5\right\}$$

表 Table 使用像函数 Sum 和 Product 一样的循环表示法. 该表示法在 "求和与求积" 中讨论过.

```
In[5]:= Product[x^i + 2 i, {i, 5}]
```

$$\text{Out[5]}= (2 + x)\left(4 + x^2\right)\left(6 + x^3\right)\left(8 + x^4\right)\left(10 + x^5\right)$$

这是一个 x 从 0 到 1，步长为 0.25 所产生的表.

```
In[6]:= Table[Sqrt[x], {x, 0, 1, 0.25}]
Out[6]= {0, 0.5, 0.707107, 0.866025, 1.}
```

可以在从 Table 中得到的列表上进行其它运算.

```
In[7]:= %^2 + 3
Out[7]= {3, 3.25, 3.5, 3.75, 4.}
```

TableForm 把列表显示为“表格”的形式. 注意，TableForm 中有两个大写字母.

```
In[8]:= % // TableForm
```

```
t[8]//TableForm=
3
3.25
3.5
3.75
4.
```

到现在为止所有表的例子都是通过变化单个参数而得到的. 用户也可以使用多个参数产生表. 在 "求和与求积" 节讨论使用标准的 ***Mathematica*** 循环表示法确定的多维表.

这里生成一个 i 从 **1** 到 **3**，j 从 **1** 到 **2** 的 $x^i + y^j$ 的表.

```
In[9]:= Table[x^i + y^j, {i, 3}, {j, 2}]
```

$$\text{Out[9]=}\ \{\{x + y,\ x + y^2\},\ \{x^2 + y,\ x^2 + y^2\},\ \{x^3 + y,\ x^3 + y^2\}\}$$

这个例子是一个 *列表 的 列表*. 其中外层列表的元素相应于 i 的值. 内层列表的元素相应于 i 值固定的情况下， j 的值.

有时用户可能想通过计算一个特定表达式多次来生成一个表，其中没有任何变量的增加.

这里生成一个包含符号 x 的**4**个拷贝的列表.

```
In[10]:= Table[x, {4}]
```

$$\text{Out[10]=}\ \{x, x, x, x\}$$

这里从 {1, 2, 3, 4} 给出**4**个伪随机数的表. Table 对列表中的

4次，得到了不同的伪随机数.

```
In[11]:= Table[RandomSample[{1, 2, 3, 4}, 2], {4}]
Out[11]= {{3, 2}, {4, 2}, {4, 3}, {2, 1}}
```

这里对列表 {1, 4, 9, 16} 中的每个 i 值计算 $\sqrt{i}$.

```
In[12]:= Table[√i, {i, {1, 4, 9, 16}}]
Out[12]= {1, 2, 3, 4}
```

这里产生一个3×2的表格.

```
In[13]:= Table[i + 2 j, {i, 3}, {j, 2}]
Out[13]= {{3, 5}, {4, 6}, {5, 7}}
```

在这个表格中，行的长度取决于较慢变化的迭代变量 i.

```
In[14]:= Table[i + 2 j, {i, 3}, {j, i}]
Out[14]= {{3}, {4, 6}, {5, 7, 9}}
```

可以使用 Table 生成具有任意维度的数组.

这里生成一个三维 2×2×2 数组. 这是一个列表的列表的列表.

```
In[15]:= Table[i j^2 k^3, {i, 2}, {j, 2}, {k, 2}]
Out[15]= {{{1, 8}, {4, 32}}, {{2, 16}, {8, 64}}}
```

`Table[f, {imax}]`	给出 f 的 i_{max} 个值的列表
`Table[f, {i, imax}]`	给出 f 的 i 从 1 到 i_{max} 的列表
`Table[f, {i, imin, imax}]`	给出 i 从 i_{min} 到 i_{max} 的列表
`Table[f, {i, imin, imax, di}]`	使用 di 为步长
`Table[f, {i, imin, imax}, {j, jmin, jmax}, ...]`	生成多维表
`Table[f, {i, {i1, i2, ...}]`	给出当 i 连续取值 $i_1, i_2, \ldots$ 时 f 的值的列表
`TableForm[list]`	把列表显示为表格形式

生成表的函数.

可以使用 "处理列表元素" 节讨论的方法提取表的元素.

这里生成一个2×2的表，起名为 sq.

```
In[16]:= sq = Table[j^2, {j, 7}]
Out[16]= {1, 4, 9, 16, 25, 36, 49}
```

这里给出表格的第三部分.

```
In[17]:= sq[[3]]
Out[17]= 9
```

这里给出第三到第五部分的列表.

```
In[18]:= Sq[[3 ;; 5]]
Out[18]= {9, 16, 25}
```

这里生成一个 2×2 的表，起名为 m.

```
In[19]:= m = Table[i - j, {i, 2}, {j, 2}]
Out[19]= {{0, -1}, {1, 0}}
```

这里从列表的列表中提取第一个子列表来产生一个表.

```
In[20]:= m[[1]]
Out[20]= {0, -1}
```

这里提取该子列表的第二个元素.

```
In[21]:= %[[2]]
Out[21]= -1
```

此处把前面两个运算合在一起进行.

```
In[22]:= m[[1, 2]]
Out[22]= -1
```

这里把 m 显示成“表格”形式.

```
In[23]:= TableForm[m]
t[23]//TableForm=
        0  -1
        1  0
```

$t[[i]]$ 或 `Part[t,i]`	给出 t 的 i 个子列表 （也作为 $t[\![i]\!]$ 输入）
$t[[i;;j]]$ 或 `Part[t,i;;j]`	给出从 i 到 j 的部分列表
$t[[\{i_1,i_2,\ldots\}]]$ 或 `Part[t,{i1,i2,...}]`	给出 t 的 i_1，i_2，… 部分构成的列表
$t[[i,j,\ldots]]$ 或 `Part[t,i,j,...]`	给出 t 的相应于 $t[[i]][[j]]$ … 的部分元素列表

提取列表的部分元素的方法.

如 "处理列表元素" 节中提到的那样，用户可以把 ***Mathematica*** 列表看作“数组”，那么列表的列表则可看作二维数组. 当把列表显示为表格形式时，每个元素的两个指标如同 x 和 y 坐标一样.

相关教程

- 列表

教程专集

- **Core Language**

处理列表元素

在 ***Mathematica*** 中，列表运算大多是将列表作为单个对象进行处理，然而，有时也需要取出或设置列表中的某个元素.

通过给出 ***Mathematica*** 列表元素的“索引”来涉及其个元素. 列表元

$\{a, b, c\}$	一个列表
`Part[list, i]` 或 `list[[i]]`	列表 *list* 的第 *i* 个元素 （第一个元素为 `list[[1]]`）
`Part[list, {i, j, ...}]` 或 `list[[{i, j, ...}]]`	列表 *list* 的第 *i*，第 *j*，... 个元素组成的列表
`Part[list, i;;j]`	列表 *list* 的第 *i* 到第 *j* 个元素组成的列表

列表元素的运算.

这里取出列表的第二个元素.

```
In[1]:= {5, 8, 6, 9}[[2]]
Out[1]= 8
```

这里取出一组元素构成列表.

```
In[2]:= {5, 8, 6, 9}[[{3, 1, 3, 2, 4}]]
Out[2]= {6, 5, 6, 8, 9}
```

把列表赋给 v.

```
In[3]:= v = {2, 4, 7}
Out[3]= {2, 4, 7}
```

这里提取 v 的元素.

```
In[4]:= v[[2]]
Out[4]= 4
```

通过把列表赋给变量，可以像其它计算机语言中的数组一样使用 ***Mathematica*** 列表. 因此. 例如，通过给 $v[[i]]$ 赋值来重新设置元素列表.

`Part[v,i]` 或 $v[[i]]$	取出列表的第 i 个元素
`Part[v,i]=` *value* 或 $v[[i]]=$ *value*	重新给列表的第 i 个元素赋值

类似数组的列表运算.

给出一个列表.

```
In[5]:= v = {4, -1, 8, 7}
Out[5]= {4, -1, 8, 7}
```

对列表的第三个元素重新赋值.

```
In[6]:= v[[3]] = 0
Out[6]= 0
```

现在赋给 v 的列表被修改了.

```
In[7]:= v
Out[7]= {4, -1, 0, 7}
```

相关教程

- 建立计算

向量和矩阵

在 ***Mathematica*** 中，向量和矩阵简单地分别用列表和列表的列表来表示.

$\{a, b, c\}$	向量 (a, b, c)
$\{\{a, b\}, \{c, d\}\}$	矩阵 $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

用列表表示向量和矩阵.

这是 **2×2** 矩阵.

```
In[1]:= m = {{a, b}, {c, d}}
Out[1]= {{a, b}, {c, d}}
```

这是该矩阵的第一行.

```
In[2]:= m[[1]]
Out[2]= {a, b}
```

这是矩阵中的元素 m_{12}.

```
In[3]:= m[[1, 2]]
Out[3]= b
```

这是一个二元向量.

```
In[4]:= v = {x, y}
Out[4]= {x, y}
```

对象 p 和 q 被作为标量处理.

```
In[5]:= p v + q
Out[5]= {q + p x, q + p y}
```

向量的加法是对应的分量加法.

```
In[6]:= v + {xp, yp} + {xpp, ypp}
Out[6]= {x + xp + xpp, y + yp + ypp}
```

这里是两个向量的点（数量）积.

```
In[7]:= {x, y}.{xp, yp}
Out[7]= x xp + y yp
```

矩阵乘以向量.

```
In[8]:= m.v
Out[8]= {a x + b y, c x + d y}
```

矩阵乘以矩阵.

```
In[9]:= m.m
Out[9]= {{a^2 + b c, a b + b d}, {a c + c d, b c + d^2}}
```

向量乘以矩阵.

```
In[10]:= v.m
Out[10]= {a x + c y, b x + d y}
```

这个组合结果是一个数.

```
In[11]:= v.m.v
Out[11]= x (a x + c y) + y (b x + d y)
```

Mathematica 使用列表表示向量和矩阵的方式，使用户完全不必区分“行”和“列”向量.

`Table[f,{i,n}]`	通过计算 $i = 1, 2, \ldots, n$ 时 f 的值，构造一个 n 维向量
`Array[a,n]`	构造形如 $\{a[1], a[2], \ldots\}$ 的 n 维向量
`Range[n]`	建立列表 $\{1, 2, 3, \ldots, n\}$
`Range[n1,n2]`	建立列表 $\{n_1, n_1 + 1, \ldots, n_2\}$
`Range[n1,n2,dn]`	建立列表 $\{n_1, n_1 + dn, \ldots, n_2\}$
list`[[i]]` or `Part[list,i]`	取出向量 *list* 的第 i 个元素
`Length[list]`	给出 *list* 的元素个数
$c\ v$	数乘向量
$a.b$	向量的点积
`Cross[a,b]`	向量的叉积 （也输入为 $a \times b$）
`Norm[v]`	向量的欧式范数（**norm**）

关于向量的函数.

`Table[f, {i,m},{j,n}]`	通过计算 i 从 1 到 m，j 从 1 到 n 的 f 的值，构造一个 $m\times n$ 矩阵
`Array[a,{m,n}]`	构造一个 $m\times n$ 矩阵，其第 i，j 个元素为 $a[i, j]$
`IdentityMatrix[n]`	生成一个 $n\times n$ 单位矩阵
`DiagonalMatrix[list]`	生成一个对角线上的元素为 *list* 的对角阵
list`[[i]]` 或 `Part[list,i]`	给出矩阵 *list* 的第 i 行
list`[[All, j]]` 或 `Part[list,All, j]`	给出矩阵 *list* 的第 j 列
list`[[i, j]]` 或 `Part[list, i, j]`	给出矩阵 *list* 的第 i，j 个元素
`Dimensions[list]`	给出矩阵 *list* 的维数

`Table[f, {i,m},{j,n}]`	通过计算 i 从 1 到 m，j 从 1 到 n 的 f 的值，构造一个 $m\times n$ 矩阵
`Array[a,{m,n}]`	构造一个 $m\times n$ 矩阵，其第 i，j 个元素为 $a[i, j]$
`IdentityMatrix[n]`	生成一个 $n\times n$ 单位矩阵
`DiagonalMatrix[list]`	生成一个对角线上的元素为 *list* 的对角阵
list`[[i]]` 或 `Part[list,i]`	给出矩阵 *list* 的第 i 行
list`[[All,j]]` 或 `Part[list,All,j]`	给出矩阵 *list* 的第 j 列
list`[[i,j]]` 或 `Part[list,i,j]`	给出矩阵 *list* 的第 i，j 个元素
`Dimensions[list]`	给出矩阵 *list* 的维数

`Table[f, {i,m},{j,n}]`	通过计算 i 从 1 到 m，j 从 1 到 n 的 f 的值，构造一个 $m\times n$ 矩阵
`Array[a,{m,n}]`	构造一个 $m\times n$ 矩阵，其第 i，j 个元素为 $a[i, j]$
`IdentityMatrix[n]`	生成一个 $n\times n$ 单位矩阵
`DiagonalMatrix[list]`	生成一个对角线上的元素为 *list* 的对角阵
list`[[i]]` 或 `Part[list,i]`	给出矩阵 *list* 的第 i 行
list`[[All, j]]` 或 `Part[list,All, j]`	给出矩阵 *list* 的第 j 列
list`[[i, j]]` 或 `Part[list,i, j]`	给出矩阵 *list* 的第 i，j 个元素
`Dimensions[list]`	给出矩阵 *list* 的维数

关于矩阵的函数.

`Column[list]`	在一个列中显示 *list* 的元素
`MatrixForm[list]`	用矩阵形式显示 *list*

向量和矩阵的格式构造.

这里构造一个 **3×3** 矩阵 s，其元素为 $s_{ij} = i + j$.

```
In[12]:= s = Table[i + j, {i, 3}, {j, 3}]
Out[12]= {{2, 3, 4}, {3, 4, 5}, {4, 5, 6}}
```

这里用标准的二维矩阵形式显示 s.

```
In[13]:= MatrixForm[s]
```

t[13]//MatrixForm=

$$\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{pmatrix}$$

这里给出具有符号元素的一个向量. 可以用其推导对任何向量组成部分有效的一般公式.

```
In[14]:= Array[a, 4]

Out[14]= {a[1], a[2], a[3], a[4]}
```

这里给出具有符号元素的 3×2 矩阵. "由函数产生列表" 节将讨论如何用Array 生成具有其它种类元素的矩阵.

```
In[15]:= Array[p, {3, 2}]

Out[15]= {{p[1, 1], p[1, 2]},
          {p[2, 1], p[2, 2]}, {p[3, 1], p[3, 2]}}
```

这是上一个矩阵的维数.

```
In[16]:= Dimensions[%]

Out[16]= {3, 2}
```

这里生成一个 3×3 对角阵.

```
In[17]:= DiagonalMatrix[{a, b, c}]

Out[17]= {{a, 0, 0}, {0, b, 0}, {0, 0, c}}
```

$c\ m$	用数乘以矩阵
$a.b$	两个矩阵的点积
Inverse[m]	矩阵的逆
MatrixPower[m,n]	矩阵的 n 次幂
Det[m]	矩阵的行列式
Tr[m]	矩阵的迹
Transpose[m]	矩阵的转置
Eigenvalues[m]	矩阵的特征值
Eigenvectors[m]	矩阵的特征向量

矩阵的一些运算.

这是前面定义的 2×2 符号变量矩阵.

```
In[18]:= m
Out[18]= {{a, b}, {c, d}}
```

这里求出它的行列式.

```
In[19]:= Det[m]
Out[19]= -b c + a d
```

这是 m 的转置阵.

```
In[20]:= Transpose[m]
Out[20]= {{a, c}, {b, d}}
```

这里给出符号形式的 m 的逆矩阵.

In[21]:=
```
Inverse[m]
```

Out[21]=
$$\left\{\left\{\frac{d}{-b\,c+a\,d},\ -\frac{b}{-b\,c+a\,d}\right\},\ \left\{-\frac{c}{-b\,c+a\,d},\ \frac{a}{-b\,c+a\,d}\right\}\right\}$$

这里是 3×3 的有理矩阵.

In[22]:=
```
h = Table[1 / (i + j - 1), {i, 3}, {j, 3}]
```

Out[22]=
$$\left\{\left\{1,\ \frac{1}{2},\ \frac{1}{3}\right\},\ \left\{\frac{1}{2},\ \frac{1}{3},\ \frac{1}{4}\right\},\ \left\{\frac{1}{3},\ \frac{1}{4},\ \frac{1}{5}\right\}\right\}$$

这里给出它的逆矩阵.

In[23]:=
```
Inverse[h]
```

Out[23]=
```
{{9, -36, 30}, {-36, 192, -180}, {30, -180, 180
```

逆矩阵与原矩阵进行点积给出单位阵.

In[24]:=
```
%.h
```

Out[24]=
```
{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

这是一个 3×3 矩阵.

In[25]:=
```
r = Table[i + j + 1, {i, 3}, {j, 3}]
```

Out[25]=
```
{{3, 4, 5}, {4, 5, 6}, {5, 6, 7}}
```

`Eigenvalues` 给出矩阵的特征值.

```
In[26]:= Eigenvalues[r]
```

$$\text{Out[26]=}\ \left\{\frac{1}{2}\left(15+\sqrt{249}\right),\ \frac{1}{2}\left(15-\sqrt{249}\right),\ 0\right\}$$

这里给出矩阵的数值近似值.

```
In[27]:= rn = N[r]
Out[27]= {{3., 4., 5.}, {4., 5., 6.}, {5., 6., 7.}}
```

这是特征值的近似值.

```
In[28]:= Eigenvalues[rn]
```

$$\text{Out[28]=}\ \left\{15.3899,\ -0.389867,\ -1.49955\times10^{-16}\right\}$$

"*Mathematica* 中的线性代数" 讨论 ***Mathematica*** 内部其它许多矩阵操作.

相关指南

- 向量操作
- 矩阵运算

相关教程

- 列表

教程专集

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**

获得列表的部分元素

`First[`*list*`]`	*list* 的第一个元素
`Last[`*list*`]`	*list* 的最后一个元素
`Part[`*list*`,`*n*`]` 或 *list*`[[`*n*`]]`	*list* 的第 *n* 个元素
`Part[`*list*`,-`*n*`]` 或 *list*`[[-`*n*`]]`	倒数第 *n* 个元素
`Part[`*list*`,`*m*`;;`*n*`]`	从第 *m* 到第 *n* 个元素
`Part[`*list*`,{`n_1`,`n_2`,...}]` 或 *list*`[[{`n_1`,`n_2`,...}]]`	由第 n_1, n_2, ... 个元素组成的列表

取出列表的元素.

使用该列表作为例子.

```
In[1]:= t = {a, b, c, d, e, f, g}
Out[1]= {a, b, c, d, e, f, g}
```

这里是 t 的最后一个元素.

```
In[2]:= Last[t]
Out[2]= g
```

这里给出第3个元素.

```
In[3]:= t[[3]]
Out[3]= c
```

这里给出列表的第 3 到第 6 个元素.

```
In[4]:= t[[3 ;; 6]]
Out[4]= {c, d, e, f}
```

这里给出列表的第1个和第4个元素.

```
In[5]:= t[[{1, 4}]]
Out[5]= {a, d}
```

Take[*list*, *n*]	*list* 的前 *n* 个元素
Take[*list*, −*n*]	*list* 的后 *n* 个元素
Take[*list*, {*m*, *n*}]	*list* 中从 *m* 到 *n* 的元素
Rest[*list*]	去掉 *list* 的第一个元素
Drop[*list*, *n*]	去掉 *list* 的前 *n* 个元素
Most[*list*]	去掉 *list* 的最后一个元素
Drop[*list*, −*n*]	去掉 *list* 的后 *n* 个元素
Drop[*list*, {*m*, *n*}]	去掉 *list* 的从第 *m* 到第 *n* 个间的元素

取出列表元素的一些序列.

这里给出前面定义的列表 t 的前3个元素.

```
In[6]:= Take[t, 3]
Out[6]= {a, b, c}
```

这里给出后3个元素.

```
In[7]:= Take[t, -3]
Out[7]= {e, f, g}
```

这里给出从 2 到 5 的元素.

```
In[8]:= Take[t, {2, 5}]
Out[8]= {b, c, d, e}
```

这里给出从 3 到 7，步长为 2 的元素.

```
In[9]:= Take[t, {3, 7, 2}]
Out[9]= {c, e, g}
```

这里去掉 t 的第1个元素.

```
In[10]:= Rest[t]
Out[10]= {b, c, d, e, f, g}
```

这里去掉 t 的前3个元素.

```
In[11]:= Drop[t, 3]
Out[11]= {d, e, f, g}
```

这里仅去掉 t 的第3个元素.

```
In[12]:= Drop[t, {3, 3}]
Out[12]= {a, b, d, e, f, g}
```

"同类列表的操作" 节将介绍如何将本节的所有函数推广到不仅用于列表，而且用于各种***Mathematica*** 表达式.

本节的函数允许用户提取列表中特定位置上的元素."寻找与模式匹配的表达式" 节将介绍如何使用像 `Select` 和 `Cases` 这样的函数提取列表元素.它们不是根据元素的位置，而是根据元素的特征进行提取.

相关教程

- 列表

- **Core Language**

检测和搜索列表元素

`Position`[*list*, *form*]	*form* 在 *list* 中的位置
`Count`[*list*, *form*]	*form* 作为 *list* 的元素所给出的次数
`MemberQ`[*list*, *form*]	检测 *form* 是否为 *list* 的元素
`FreeQ`[*list*, *form*]	检测 *form* 是否不在 *list* 中

检测和搜索列表的元素.

"获得列表的部分元素" 讨论如何根据元素的位置或标号提取列表的元素. ***Mathematica*** 也有根据元素的值搜索和检测列表元素的函数.

这里给出元素 a 在列表中的位置的列表.

```
In[1]:= Position[{a, b, c, a, b}, a]
Out[1]= {{1}, {4}}
```

`Count` 计算 a 出现的次数.

```
In[2]:= Count[{a, b, c, a, b}, a]
Out[2]= 2
```

这里表明 a 是 {a, b, c} 的一个元素.

```
In[3]:= MemberQ[{a, b, c}, a]
Out[3]= True
```

然而，d 不是其中的元素.

```
In[4]:= MemberQ[{a, b, c}, d]
Out[4]= False
```

定义 m 为一个 **3×3** 单位矩阵.

```
In[5]:= m = IdentityMatrix[3]
Out[5]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

这里表明 0 是 m 的元素.

```
In[6]:= FreeQ[m, 0]
Out[6]= False
```

这里给出 0 出现在 m 中的位置的列表.

```
In[7]:= Position[m, 0]
Out[7]= {{1, 2}, {1, 3}, {2, 1}, {2, 3}, {3, 1}, {3, 2}}
```

如同 "寻找与模式匹配的表达式" 节讨论的那样，函数 Count 和 Position，以及 MemberQ 和 FreeQ，不仅能用于搜索“特定”的列表元素，而且能搜索与特定“模式”相匹配的元素类.

相关指南

- 规则与模式

相关教程

- 列表

教程专集

- **Core Language**

添加、删除和修改列表元素

`Prepend`[*list*, *element*]	在 *list* 的开头添加元素 *element*
`Append`[*list*, *element*]	在 *list* 的末尾添加元素 *element*
`Insert`[*list*, *element*, *i*]	在 *list* 的第 *i* 个位置上插入 *element*
`Insert`[*list*, *element*, −*i*]	在 *list* 的倒数第 *i* 个位置上插入 *element*
`Riffle`[*list*, *element*]	在 *list* 的位置上交错放置 *element*
`Delete`[*list*, *i*]	去掉 *list* 的第 *i* 个位置上的元素
`ReplacePart`[*list*, *i*−>*new*]	用 *new* 替换 *list* 的第 *i* 个位置上的元素
`ReplacePart`[*list*, {*i*, *j*}−>*new*]	用 *new* 替换 *list*[[*i*, *j*]]

在清晰的列表中的元素操作函数.

这里将 x 添加到列表的开头.

```
In[1]:= Prepend[{a, b, c}, x]
Out[1]= {x, a, b, c}
```

这里将 x 插入到第2个位置上.

```
In[2]:= Insert[{a, b, c}, x, 2]
Out[2]= {a, x, b, c}
```

在列表的位置上交错放置 x.

```
In[3]:= Riffle[{a, b, c}, x]
Out[3]= {a, x, b, x, c}
```

用 x 替换列表的第3个元素.

```
In[4]:= ReplacePart[{a, b, c, d}, 3 -> x]
Out[4]= {a, b, x, d}
```

替换 2×2 矩阵中的第1，2个元素.

```
In[5]:= ReplacePart[{{a, b}, {c, d}}, {1, 2} -> x]
Out[5]= {{a, x}, {c, d}}
```

函数如 ReplacePart 用明确的列表产生新的列表. 然而，有时候，用户可能想要在合适的位置修改一个列表，而不明确产生一个新的列表.

$v=\{e_1, e_2, \ldots\}$	将一个列表赋给变量
$v[[i]]=new$	对第 i 个元素赋以新值

重设列表元素.

这里定义 v 为一个列表.

```
In[6]:= v = {a, b, c, d}
Out[6]= {a, b, c, d}
```

这里设置第3个元素为 x.

```
In[7]:= v[[3]] = x
Out[7]= x
```

现在 v 被修改了.

```
In[8]:= v
Out[8]= {a, b, x, d}
```

$m[[i, j]]=new$	替换矩阵的第（i, j）个元素
$m[[i]]=new$	替换第 i 行
$m[[\mathtt{All}, i]]=new$	替换第 i 列

重设矩阵的块.

这里定义 m 为一个矩阵.

```
In[9]:= m = {{a, b}, {c, d}}
Out[9]= {{a, b}, {c, d}}
```

这里设置矩阵的第一列.

```
In[10]:= m[[All, 1]] = {x, y}; m
Out[10]= {{x, b}, {y, d}}
```

这里对第一列的每个元素赋以值 0.

```
In[11]:= m[[All, 1]] = 0; m
Out[11]= {{0, b}, {0, d}}
```

相关指南

- 列表元素

相关教程

- 列表
- 使用列表

教程专集

- **Core Language**

Join[$list_1$, $list_2$, ...]	把列表连接在一起
Union[$list_1$, $list_2$, ...]	列表的并（合并列表，删除重复的元素，并整理结果）
Riffle[$list_1$, $list_2$]	交错放置 $list_1$ 和 $list_2$ 的元素

组合列表函数.

Join 可以连接任意多个列表.

```
In[1]:= Join[{a, b, c}, {x, y}, {t, u}]
Out[1]= {a, b, c, x, y, t, u}
```

Union 合并列表，且保持列表中只有不同元素.

```
In[2]:= Union[{a, b, c}, {c, a, d}, {a, d}]
Out[2]= {a, b, c, d}
```

Riffle 通过交错放置列表的元素合并列表.

```
In[3]:= Riffle[{a, b, c}, {x, y, z}]
Out[3]= {a, x, b, y, c, z}
```

相关指南

- 列表操作

相关教程

- 列表

作为集合的列表

Mathematica 通常按用户最初输入的元素顺序，保持列表元素. 但是，如果用户希望像数学*集合* 一样处理 ***Mathematica*** 列表，可能希望不考虑元素在集合中的顺序.

`Union[`*list*$_1$`,` *list*$_2$`, ...]`	给出 *list*$_i$ 中不同元素的列表
`Intersection[`*list*$_1$`,` *list*$_2$`, ...]`	给出 *list*$_i$ 中共有的元素的列表
`Complement[`*universal*`,` *list*$_1$`, ...]`	给出在 *universal* 中，但不在 *list*$_i$ 中的元素的列表
`Subsets[`*list*`]`	给出 *list* 中元素的所有子集的列表
`DeleteDuplicates[`*list*`]`	从 *list* 中删除所有重复元素

集合论的函数.

`Union` 给出在*任一个* 列表中出现的元素.

```
In[1]:= Union[{c, a, b}, {d, a, c}, {a, e}]
Out[1]= {a, b, c, d, e}
```

Intersection 给出在*所有* 集合中都出现的元素.

```
In[2]:= Intersection[{a, c, b}, {b, a, d, a}]
Out[2]= {a, b}
```

Complement 给出在第1个集合出现，而不在其它集合中出现的元素.

```
In[3]:= Complement[{a, b, c, d}, {a, d}]
Out[3]= {b, c}
```

这里给出该列表的所有子集.

```
In[4]:= Subsets[{a, b, c}]
Out[4]= {{}, {a}, {b}, {c}, {a, b},
          {a, c}, {b, c}, {a, b, c}}
```

DeleteDuplicates 从列表中删除所有重复元素.

```
In[5]:= DeleteDuplicates[{a, b, c, a}]
Out[5]= {a, b, c}
```

相关指南

- 重排列与重构列表

相关教程

- 列表

重排列表

`Sort`[*list*]	把列表 *list* 元素整理成标准顺序
`Union`[*list*]	整理元素，删除重复元素
`Reverse`[*list*]	对 *list* 的元素进行反向排序
`RotateLeft`[*list*, *n*]	把列表 *list* 元素向左轮换 *n* 个位置
`RotateRight`[*list*, *n*]	把列表元素向右轮换 *n* 个位置

重排列表函数.

把列表元素整理成标准顺序，此处，顺序是字母或数字.

```
In[1]:= Sort[{b, a, c, a, b}]
Out[1]= {a, a, b, b, c}
```

整理元素，删除重复的元素.

```
In[2]:= Union[{b, a, c, a, b}]
Out[2]= {a, b, c}
```

把列表中的元素向左轮换（“移动”）两个位置.

```
In[3]:= RotateLeft[{a, b, c, d, e}, 2]
Out[3]= {c, d, e, a, b}
```

可以通过给出负位移或使用 RotateRight 进行向右轮换.

```
In[4]:= RotateLeft[{a, b, c, d, e}, -2]
Out[4]= {d, e, a, b, c}
```

PadLeft[*list*, *len*, *x*]	在 *list* 的左边用 *x* 进行填充，产生一个长度为 *len* 的列表
PadRight[*list*, *len*, *x*]	在 *list* 的右边进行填充

填充列表.

这里用 x 在列表的左边填充生成一个长度为10的列表.

```
In[5]:= PadLeft[{a, b, c}, 10, x]
Out[5]= {x, x, x, x, x, x, x, a, b, c}
```

相关指南

- 重排列与重构列表

相关教程

- 列表
- 嵌套列表
- 嵌套列表的重排

教程专集

- **Core Language**

列表元素的分组和合并

`Partition[list, n]`	把 *list* 分成 *n* 个元素一组
`Partition[list, n, d]`	使用偏移 *d* 进行逐次分组
`Split[list]`	把 *list* 按邻接的相同元素进行分组

列表元素的分组函数.

这是一个列表.

```
In[1]:= t = {a, b, c, d, e, f, g}
Out[1]= {a, b, c, d, e, f, g}
```

把列表元素按对分组，此时，去掉末尾余下的单个元素.

```
In[2]:= Partition[t, 2]
Out[2]= {{a, b}, {c, d}, {e, f}}
```

这里将元素分为3个一组，不同组之间没有重叠.

```
In[3]:= Partition[t, 3]
Out[3]= {{a, b, c}, {d, e, f}}
```

这里分成3个元素一组，相邻的三元素组之间仅差一个元素.

```
In[4]:= Partition[t, 3, 1]

Out[4]= {{a, b, c}, {b, c, d},
         {c, d, e}, {d, e, f}, {e, f, g}}
```

把列表按邻接的相同元素进行分组.

```
In[5]:= Split[{a, a, b, b, b, a, a, a, b}]

Out[5]= {{a, a}, {b, b, b}, {a, a, a}, {b}}
```

Tuples[*list*, *n*]	生成所有元素是由 *list* 得来的 *n* 元组
Tuples[{*list*$_1$, *list*$_2$, ...}]	生成所有第 *i* 个元素是由 *list*$_i$ 得来的元组

寻找列表中元素的可能元组.

这里给出从列表中提取两个元素的所有可能性.

```
In[6]:= Tuples[{a, b}, 2]

Out[6]= {{a, a}, {a, b}, {b, a}, {b, b}}
```

这里给出从每个列表中提取一个元素的所有可能性.

```
In[7]:= Tuples[{{a, b}, {1, 2, 3}}]

Out[7]= {{a, 1}, {a, 2}, {a, 3}, {b, 1}, {b, 2}, {b, 3}}
```

相关指南

- 列表操作

相关教程

- 列表

教程专集

- **Core Language**

列表中的排序

`Sort`[*list*]	把 *list* 中的元素按顺序排列
`Ordering`[*list*]	`Sort`[*list*] 中 *list* 元素的位置
`Ordering`[*list*, *n*]	`Ordering`[*list*] 的前 *n* 个元素
`Ordering`[*list*, -*n*]	`Ordering`[*list*] 的最后 *n* 个元素
`Permutations`[*list*]	*list* 的所有可能排序
`Min`[*list*]	*list* 中的最小元素
`Max`[*list*]	*list* 中的最大元素

列表中的排序.

这是一个由数构成的列表.

```
In[1]:= t = {17, 21, 14, 9, 18}
Out[1]= {17, 21, 14, 9, 18}
```

这里按排好的顺序给出 t 的元素.

```
In[2]:= Sort[t]
Out[2]= {9, 14, 17, 18, 21}
```

这里给出 t 的元素位置，从最小到最大的位置.

```
In[3]:= Ordering[t]
Out[3]= {4, 3, 1, 5, 2}
```

这里与 Sort[t] 产生相同的结果.

```
In[4]:= t[[%]]
Out[4]= {9, 14, 17, 18, 21}
```

这里给出列表中的最小元素.

```
In[5]:= Min[t]
Out[5]= 9
```

相关教程

- 列表

教程专集

- **Core Language**

嵌套列表的重排

如果使用矩阵或生成多维数组的表，将会碰到嵌套列表.
Mathematica 提供了许多函数来处理这样的列表.

`Flatten[`*list*`]`	压平 *list* 的所有层
`Flatten[`*list*`,`*n*`]`	压平 *list* 的前 *n* 层
`Partition[`*list*`,{`n_1`,`n_2`,...}]`	分成大小为 $n_1 \times n_2 \times \ldots$ 的块
`Transpose[`*list*`]`	交换列表的前两层
`RotateLeft[`*list*`,{`n_1`,`n_2`,...}]`	将层逐次轮换 n_i 个位置
`PadLeft[`*list*`,{`n_1`,`n_2`,...}]`	把第 i 层填充为长为 n_i 的子列表

嵌套列表重排函数.

这里“压平”子列表. 用户可以把它看作是去掉了内层集合的括号.

```
In[1]:= Flatten[{{a}, {b, {c}}, {d}}]
Out[1]= {a, b, c, d}
```

这里只压平了子列表的一层.

```
In[2]:= Flatten[{{a}, {b, {c}}, {d}}, 1]
Out[2]= {a, b, {c}, d}
```

对嵌套列表还有许多其它运算. 在 "操作列表" 中将讨论更多操作.

相关教程

- 列表

- **Core Language**

操作列表

构建列表
通过索引操作列表
嵌套列表
列表的分组和填充
稀疏数组：列表的处理

教程专集

- **Core Language**

构建列表

列表在 ***Mathematica*** 中被广泛使用，并且对于构建列表，有许多不同的方式.

`Range[n]`	列表 {1, 2, 3, …, n}
`Table[expr, {i, n}]`	i 从 1 到 n 的 *expr* 的值
`Array[f, n]`	列表 {f[1], f[2], …, f[n]}
`NestList[f, x, n]`	具有 n 个嵌套的 {x, f[x], f[f[x]], …}
`Normal[SparseArray[{`i_1`->`v_1`, …}, n]]`	长度为 n 的列表，元素 i_k 为 v_k
`Apply[List, f[`e_1`, `e_2`, …]]`	列表 {e_1, e_2, …}

用以构建列表的一些明确方式.

这里给出以2为幂的前5个值的列表.

```
In[1]:= Table[2^i, {i, 5}]
Out[1]= {2, 4, 8, 16, 32}
```

以下是取得相同结果的另一种方式.

```
In[2]:= Array[2^# &, 5]
Out[2]= {2, 4, 8, 16, 32}
```

这里给出相似的列表.

```
In[3]:= NestList[2 # &, 1, 5]
Out[3]= {1, 2, 4, 8, 16, 32}
```

SparseArray 帮助用户在特定位置指定值.

```
In[4]:= Normal[SparseArray[{3 -> x, 4 -> y}, 5]]
Out[4]= {0, 0, x, y, 0}
```

也可以使用模式来指定值.

```
In[5]:= Normal[SparseArray[{i_ -> 2^i}, 5]]
Out[5]= {2, 4, 8, 16, 32}
```

通常情况下，用户可以提前知道一个列表应该有多长，并且每个元素应该如何生成. 而且，用户可以从一个列表得到另一个列表.

Table[*expr*, {*i*, *list*}]	当 *i* 从 *list* 中取值时， *expr* 的值
Map[*f*, *list*]	对 *list* 的每个元素应用 *f*
MapIndexed[*f*, *list*]	对于第 *i* 个元素，给出 *f*[*elem*, {*i*}]
Cases[*list*, *form*]	给出与 *form* 匹配的 *list* 的元素
Select[*list*, *test*]	选出 *test*[*elem*] 为 True 的元素
Pick[*list*, *sel*, *form*]	选出 *list* 的元素，*sel* 相应的元素与 *form* 匹配
TakeWhile[*list*, *test*]	只要 *test*[e_i] 是 True，从 *list* 的开头给出元素 e_i
list[[{i_1, i_2, ...}]] 或 Part[*list*, {i_1, i_2, ...}]	给出 *list* 指定部分的列表

从其它列表构建列表.

这里选出小于5的元素.

```
In[6]:= Select[{1, 3, 7, 4, 10, 2}, # < 5 &]
Out[6]= {1, 3, 4, 2}
```

这里选出不小于5的第一个元素.

```
In[7]:= TakeWhile[{1, 3, 7, 4, 10, 2}, # < 5 &]
Out[7]= {1, 3}
```

这里明确给出编号部分.

```
In[8]:= {a, b, c, d}[[{2, 1, 4}]]
Out[8]= {b, a, d}
```

这里选出第二个列表中 1 所对应的元素.

```
In[9]:= Pick[{a, b, c, d}, {1, 0, 1, 1}, 1]
Out[9]= {a, c, d}
```

有时候，用户可能想要在程序执行过程中积累结果列表. 这可以使用 Sow 和 Reap 实现.

Sow[*val*]	对于最接近的 Reap，进行值 *val* 的 **sow**操作
Reap[*expr*]	计算 *expr*，并返回 Sow 产生的值的列表

使用 Sow 和 Reap.

该程序对一个数进行迭代求平方.

```
In[10]:= Nest[#^2 &, 2, 6]

Out[10]= 18 446 744 073 709 551 616
```

这里进行同样的计算，但这里把大于**1000**的中间结果累积成列表.

```
In[11]:= Reap[Nest[(If[# > 1000, Sow[#]];
           #^2) &, 2, 6]]

Out[11]= {18 446 744 073 709 551 616,
          {{65 536, 4 294 967 296}}}
```

另一个效率比较低的方法涉及引入一个临时变量，并以 $t = \{\}$ 开始，然后连续使用 AppendTo[t, $elem$].

相关指南

- 构造列表

相关教程

- 操作列表
- 使用列表

教程专集

- **Core Language**

通过索引操作列表

`Part[`*list*`,`*spec*`]` 或 *list*`[[`*spec*`]]`	列表的一个或多个部分
`Part[`*list*`,`$spec_1$`,` $spec_2$`,...]` 或 *list*`[[`$spec_1$`,` $spec_2$`,...]]`	嵌套列表的一个或多个部分
n	开头的第 *n* 个部分
$-n$	末尾的第 *n* 个部分
$\{i_1, i_2, \ldots\}$	由部分组成的列表
m`;;`*n*	第 *m* 部分到第 *n* 部分
`All`	所有部分

获取列表的部分.

这里给出列表的第一部分和第三部分.

```
In[1]:= {a, b, c, d}[[{1, 3}]]
Out[1]= {a, c}
```

以下是一个嵌套列表.

```
In[2]:= m = {{a, b, c}, {d, e}, {f, g, h}};
```

这里给出第一部分和第三部分的列表.

```
In[3]:= m[[{1, 3}]]
Out[3]= {{a, b, c}, {f, g, h}}
```

这里给出每个部分第一子部的列表.

```
In[4]:= m[[{1, 3}, 1]]
Out[4]= {a, f}
```

这里给出前两个部分的列表.

```
In[5]:= m[[{1, 3}, {1, 2}]]
Out[5]= {{a, b}, {f, g}}
```

这里给出 m 的前两个部分.

```
In[6]:= m[[1 ;; 2]]
Out[6]= {{a, b, c}, {d, e}}
```

这里给出相应的最后部分.

```
In[7]:= m[[1 ;; 2, -1]]
Out[7]= {c, e}
```

这里给出所有子列表的第二部分.

```
In[8]:= m[[All, 2]]
Out[8]= {b, e, g}
```

这里给出所有子列表的最后两个部分.

```
In[9]:= m[[All, -2 ;; -1]]
Out[9]= {{b, c}, {d, e}, {g, h}}
```

用户总数通过类似 *m*[[...]] = *value* 这样的赋值来重设列表的一个

这里重设 m 的第一部分和第二部分.

```
In[10]:= m[[1, 2]] = x
Out[10]= x
```

这是 m 的形式.

```
In[11]:= m
Out[11]= {{a, x, c}, {d, e}, {f, g, h}}
```

这里把第一部分重设为 x，并且把第三部分重设为 y.

```
In[12]:= m[[{1, 3}]] = {x, y}; m
Out[12]= {x, {d, e}, y}
```

这里把第一部分和第三部分都重设为 p.

```
In[13]:= m[[{1, 3}]] = p; m
Out[13]= {p, {d, e}, p}
```

这里重新存储 m 原先的形式.

```
In[14]:= m = {{a, b, c}, {d, e}, {f, g, h}};
```

这里重设使用 m[[{1, 3}, {1, 2}]] 指定的部分.

```
In[15]:= m[[{1, 3}, {1, 2}]] = x; m
Out[15]= {{x, x, c}, {d, e}, {x, x, h}}
```

用户可以使用 ;; 来指定给定范围内所有索引.

```
In[16]:= m[[1 ;; 3, 2]] = y; m
Out[16]= {{x, y, c}, {d, y}, {x, y, h}}
```

有时，把一个嵌套列表想象为在空间的展开是有益的，这时，每个元素都是在由其索引所提供的坐标位置上. 对于 $list[[spec_1, spec_2, \ldots]]$，存在直接的几何解释. 如果一个给定的 $spec_k$ 是一个单一的整数，那么它表示在第 k 维提取单一的分块，而如果它是一个列表的话，它表示提取平行分块的列表. 那么 $list[[spec_1, spec_2, \ldots]]$ 的最终结果是在每个连续纬度上分块所得到的元素集合.

这是一个显示为二维数组的嵌套列表.

```
In[17]:= (m = {{a, b, c}, {d, e, f}, {g, h, i}}) //
          TableForm
```

```
t[17]//TableForm=
        a  b  c
        d  e  f
        g  h  i
```

这里选出行1和行3，接着是列1和列2.

```
In[18]:= m[[{1, 3}, {1, 2}]] // TableForm
```

```
t[18]//TableForm=
        a  b
        g  h
```

Part 被设置使得提取嵌套列表的结构分片更为简单. 然而，有时用户可能想要选出由单一部分组成的任意集合. 也可以使用 Extract 方便地实现.

Part[*list*, {$i_1, i_2, \ldots$}]	列表 {*list*[[i_1]], *list*[[i_2]], ...}
Extract[*list*, {$i_1, i_2, \ldots$}]	元素 *list*[[i_1, i_2, ...]]
Part[*list*, $spec_1, spec_2, \ldots$]	由连续分片指定的部分
Extract[*list*, {{$i_1, i_2, \ldots$}, {$j_1, j_2, \ldots$}, ...}]	单一部分{*list*[[i_1, i_2, ...]], *list*[[j_1, j_2, ...]], ...} 的列表

获取分片与获取单一部分组成的列表的比较.

这里提取单一部分**1**，**3**和**1**，**2**.

```
In[19]:= Extract[m, {{1, 3}, {1, 2}}]
Out[19]= {c, b}
```

Extract 的重要特征是它对部分位置的列表采取与函数如 Position 返回的形式相同的形式.

这里建立一个嵌套列表.

```
In[20]:= m = {{a[1], a[2], b[1]}, {b[2], c[1]}, {{b[3]}}}
```

这里给出 m 中位置的列表.

```
In[21]:= Position[m, b[_]]
Out[21]= {{1, 3}, {2, 1}, {3, 1, 1}}
```

这里从这些位置提取元素.

```
In[22]:= Extract[m, %]
Out[22]= {b[1], b[2], b[3]}
```

`Take[list, spec]`	选出列表的指定部分
`Drop[list, spec]`	丢弃列表的指定部分
`Take[list, spec1, spec2, ...], Drop[list, spec1, spec2, ...]`	在嵌套列表的每一层选出或丢弃指定部分
n	前 n 个元素
$-n$	最后 n 个元素
$\{n\}$	仅含元素 n
$\{m, n\}$	元素 m 到 n （闭区间）
$\{m, n, s\}$	元素 m 到 n，步长为 s
`All`	所有部分
`None`	没有部分

在列表中选出和丢弃元素序列.

这里选出所有从位置2开始的第二个元素.

```
In[23]:= Take[{a, b, c, d, e, f, g}, {2, -1, 2}]
Out[23]= {b, d, f}
```

这里丢弃所有的第二个元素.

```
In[24]:= Drop[{a, b, c, d, e, f, g}, {2, -1, 2}]

Out[24]= {a, c, e, g}
```

与 Part 很相似，Take 和 Drop 可以视为在嵌套列表的连续层中选出块的序列，用户可以使用 Take 和 Drop 来对数组中的元素块进行操作.

这是一个3×3数组.

```
In[25]:= (m = {{a, b, c}, {d, e, f}, {g, h, i}}) //
          TableForm

t[25]//TableForm=
          a  b  c
          d  e  f
          g  h  i
```

这是前 2×2 个子数组.

```
In[26]:= Take[m, 2, 2] // TableForm

t[26]//TableForm=
          a  b
          d  e
```

这里取出前两列的所有元素.

```
In[27]:= Take[m, All, 2] // TableForm

t[27]//TableForm=
          a  b
          d  e
          g  h
```

这里没有保留前两列元素.

```
In[28]:= Drop[m, None, 2] // TableForm
```

```
t[28]//TableForm=
        c
        f
        i
```

`Prepend[list, elem]`	在 *list* 的开头加入 *element*
`Append[list, elem]`	在 *list* 的末尾加入 *element*
`Insert[list, elem, i]`	在位置 *i* 插入 *element*
`Insert[list, elem, {i, j, ...}]`	在位置 {*i*, *j*, ...} 插入
`Delete[list, i]`	在位置 *i* 删除元素
`Delete[list, {i, j, ...}]`	在位置 {*i*, *j*, ...} 删除

在列表中加入和删除元素.

这里令列表的第2，1位置上的元素为 x.

```
In[29]:= Insert[{{a, b, c}, {d, e}}, x, {2, 1}]
Out[29]= {{a, b, c}, {x, d, e}}
```

这里再次删除元素.

```
In[30]:= Delete[%, {2, 1}]
Out[30]= {{a, b, c}, {d, e}}
```

ReplacePart[*list*, *i*->*new*]	用 *new* 替换 *list* 中位置 *i* 上的元素
ReplacePart[*list*, {*i*, *j*, ...}->*new*]	用 *new* 替换 *list*[[*i*, *j*, ...]]
ReplacePart[*list*, {i_1->new_1, i_2->new_2, ...}]	用 new_n 替换位置 i_n 上的部分
ReplacePart[*list*, {{i_1, j_1, ...}->new_1, ...}]	用 new_n 替换位置 {i_n, j_n, ...} 上的部分
ReplacePart[*list*, {{i_1, j_1, ...}, ...}-> *new*]	用 *new* 替换所有部分 *list*[[i_k, j_k, ...]]

替换列表的部分.

这里使用 x 取代矩阵的第3个元素.

```
In[31]:= ReplacePart[{a, b, c, d}, 3 -> x]
Out[31]= {a, b, x, d}
```

这里取代列表的第一和第四部分. 注意在指定多个部分来取代的情况下，需要双列表.

```
In[32]:= ReplacePart[{a, b, c, d}, {{1}, {4}} -> x]
Out[32]= {x, b, c, x}
```

这是一个3×3单位矩阵.

```
In[33]:= IdentityMatrix[3]
Out[33]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

这里使用 x 取代矩阵的2，2部分.

```
In[34]:= ReplacePart[%, {2, 2} -> x]
Out[34]= {{1, 0, 0}, {0, x, 0}, {0, 0, 1}}
```

理解 ReplacePart 总是创建一个新的列表是重要的. 它并不像 *m*[[...]] = *val* 那样改变已经赋给一个符号的列表.

这里把值的列表赋于 alist.

```
In[35]:= alist = {a, b, c, d}
Out[35]= {a, b, c, d}
```

这里给出列表的拷贝，其中第三个元素由 x 取代.

```
In[36]:= ReplacePart[alist, 3 -> x]
Out[36]= {a, b, x, d}
```

alist 的值没有被更改.

```
In[37]:= alist
Out[37]= {a, b, c, d}
```

相关指南

- 表达式的子集

相关教程

- 表达式的项
- 操作列表

- 添加、删除和修改列表元素
- 函数作用于表达式的部分项

教程专集

- **Core Language**

嵌套列表

$\{list_1, list_2, \ldots\}$	列表的列表
`Table[`*expr*`, {`*i*`, `*m*`}, {`*j*`, `*n*`}, ...]`	由 *expr* 的值构成的 *m*×*n*×... 表格
`Array[`*f*`, {`*m*`, `*n*`, ...}]`	由 $f[i, j, \ldots]$ 的值构成的 *m*×*n*×... 数组
`Normal[SparseArray[{{`$i_1, j_1, \ldots$`}->`$v_1, \ldots$`}, {`*m*`, `*n*`, ...}]]`	*m*×*n*×... 数组，元素 $\{i_s, j_s, \ldots\}$ 为 v_s
`Outer[`*f*`, `$list_1, list_2, \ldots$`]`	对与 *f* 结合的元素进行的一般外层求积操作
`Tuples[`*list*`, {`*m*`, `*n*`, ...}]`	由 *list* 得到的元素组成的所有 *m*×*n*×... 数组

构建嵌套列表的方式.

这里产生一个表格，对应于一个**2×3**嵌套列表.

```
In[1]:= Table[x^i + j, {i, 2}, {j, 3}]
```

$$\text{Out[1]= } \left\{\{1 + x,\ 2 + x,\ 3 + x\},\ \left\{1 + x^2,\ 2 + x^2,\ 3 + x^2\right\}\right\}$$

这里产生对应于相同嵌套列表的数组.

```
In[2]:= Array[x^#1 + #2 &, {2, 3}]
Out[2]= {{1 + x, 2 + x, 3 + x}, {1 + x^2, 2 + x^2, 3 + x^2}}
```

没有在稀疏数组中明确指定的元素为 0.

```
In[3]:= Normal[SparseArray[{{1, 3} -> 3 + x}, {2, 3}]]
Out[3]= {{0, 0, 3 + x}, {0, 0, 0}}
```

最终列表的每个元素包含从每个输入列表的元素.

```
In[4]:= Outer[f, {a, b}, {c, d}]
Out[4]= {{f[a, c], f[a, d]}, {f[b, c], f[b, d]}}
```

函数如 Array，SparseArray 和 Outer 总是产生完全数组，其中在特定层的所有子列表都具有相同的长度.

Dimensions[*list*]	完全数组的维度
ArrayQ[*list*]	测试在给定层上的所有子列表是否具有相同长度
ArrayDepth[*list*]	所有子列表具有相同长度

完全数组的函数.

Mathematica 可以处理任意嵌套列表. 列表没有必要形成一个完全数组. 使用 Table 可以很容易地形成不规则的数组.

这里产生一个三角数组.

```
In[5]:= Table[x^i + j, {i, 3}, {j, i}]
```

$$\text{Out[5]}= \left\{\{1+x\},\ \left\{1+x^2,\ 2+x^2\right\},\ \left\{1+x^3,\ 2+x^3,\ 3+x^3\right\}\right\}$$

`Flatten[`*list*`]`	压平 *list* 的所有层
`Flatten[`*list*`,`*n*`]`	压平前 *n* 层
`ArrayFlatten[`*list*`,`*rank*`]`	从由数组构成的数组构建压平的数组

对子列表和子数组进行压平操作.

这里产生一个**2×3**数组.

```
In[6]:= Array[a, {2, 3}]

Out[6]= {{a[1, 1], a[1, 2], a[1, 3]},
         {a[2, 1], a[2, 2], a[2, 3]}}
```

`Flatten` 根据索引的字典顺序放置元素.

```
In[7]:= Flatten[%]

Out[7]= {a[1, 1], a[1, 2], a[1, 3],
         a[2, 1], a[2, 2], a[2, 3]}
```

这里从一个块矩阵生成矩阵.

```
In[8]:= ArrayFlatten[{{{{1}}, {{2, 3}}},
                      {{{4}}, {{7}}, {{5, 6}, {8, 9}}}}]

Out[8]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
```

`Transpose[`*list*`]`	对 *list* 的前两层进行转置
`Transpose[`*list*`,{`n_1`,`n_2`,...}]`	在第 n_k 层放入 *list* 中的第 k 层

在嵌套列表上对不同层进行转置操作.

这里产生一个2×2×2数组.

```
In[9]:= Array[a, {2, 2, 2}]

Out[9]= {{{a[1, 1, 1], a[1, 1, 2]},
          {a[1, 2, 1], a[1, 2, 2]}},
         {{a[2, 1, 1], a[2, 1, 2]},
          {a[2, 2, 1], a[2, 2, 2]}}}
```

这里对不同层进行重排，以使得第3层出现在第1层.

```
In[10]:= Transpose[%, {3, 1, 2}]

Out[10]= {{{a[1, 1, 1], a[2, 1, 1]},
           {a[1, 1, 2], a[2, 1, 2]}},
          {{a[1, 2, 1], a[2, 2, 1]},
           {a[1, 2, 2], a[2, 2, 2]}}}
```

这里恢复原来的数组.

```
In[11]:= Transpose[%, {2, 3, 1}]

Out[11]= {{{a[1, 1, 1], a[1, 1, 2]},
           {a[1, 2, 1], a[1, 2, 2]}},
          {{a[2, 1, 1], a[2, 1, 2]},
           {a[2, 2, 1], a[2, 2, 2]}}}
```

Map[*f*, *list*, {*n*}]	在第 *n* 层上对所有元素进行 *f* 映射
Apply[*f*, *list*, {*n*}]	对第 *n* 层上的所有元素应用 *f*
MapIndexed[*f*, *list*, {*n*}]	在第 *n* 层上的部分元素以及索引上映射 *f*

在嵌套函数上应用函数.

这是一个嵌套列表.

```
In[12]:= m = {{{a, b}, {c, d}}, {{e, f}, {g, h}, {i}}};
```

在第2层对函数 f 进行映射.

```
In[13]:= Map[f, m, {2}]

Out[13]= {{f[{a, b}], f[{c, d}]},
          {f[{e, f}], f[{g, h}], f[{i}]}}
```

这里在第二层应用函数.

```
In[14]:= Apply[f, m, {2}]

Out[14]= {{f[a, b], f[c, d]}, {f[e, f], f[g, h], f[i]}}
```

这里对列表的部分元素和它们的索引应用 f.

```
In[15]:= MapIndexed[f, m, {2}]

Out[15]= {{f[{a, b}, {1, 1}], f[{c, d}, {1, 2}]},
          {f[{e, f}, {2, 1}],
           f[{g, h}, {2, 2}], f[{i}, {2, 3}]}}
```

`Partition[list, {n1, n2, ...}]`	分成 $n_1 \times n_1 \times \ldots$ 的块
`PadLeft[list, {n1, n2, ...}]`	在左边填充以生成一个 $n_1 \times n_1 \times \ldots$ 数组
`PadRight[list, {n1, n2, ...}]`	在右边填充以生成一个 $n_1 \times n_1 \times \ldots$ 数组
`RotateLeft[list, {n1, n2, ...}]`	在第 k 层向左轮换 n_k 个位置
`RotateRight[list, {n1, n2, ...}]`	在第 k 层向右轮换 n_k 个位置

在嵌套列表上的操作.

这是一个嵌套列表.

```
In[16]:= m = {{{a, b, c}, {d, e}}, {{f, g}, {h}, {i}}};
```

这里在每一层旋转不同的量.

```
In[17]:= RotateLeft[m, {0, 1, -1}]
Out[17]= {{{e, d}, {c, a, b}}, {{h}, {i}, {g, f}}}
```

这里用零进行填充，以生成一个2×3×3数组.

```
In[18]:= PadRight[%, {2, 3, 3}]
Out[18]= {{{e, d, 0}, {c, a, b}, {0, 0, 0}},
          {{h, 0, 0}, {i, 0, 0}, {g, f, 0}}}
```

相关指南

- 张量

相关教程

- 操作列表

教程专集

- **Core Language**

列表的分组和填充

`Partition[list, n]`	把 *list* 分为长度为 *n* 的子列表
`Partition[list, n, d]`	使用偏移量 *d* 划分子列表
`Split[list]`	把 *list* 分成相同元素的子列表
`Split[list, test]`	分成相邻元素满足 *test* 的子列表

对列表中的元素进行分组.

这里把列表划分为大小为**3**的块.

```
In[1]:= Partition[{a, b, c, d, e, f}, 3]
Out[1]= {{a, b, c}, {d, e, f}}
```

这里以偏移量**1**把列表分为大小为**3**的块.

```
In[2]:= Partition[{a, b, c, d, e, f}, 3, 1]
Out[2]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e, f}}
```

该偏移量可以大于块大小.

```
In[3]:= Partition[{a, b, c, d, e, f}, 2, 3]
Out[3]= {{a, b}, {d, e}}
```

这里分成具有相同元素的子列表.

```
In[4]:= Split[{1, 4, 1, 1, 1, 2, 2, 3, 3}]
Out[4]= {{1}, {4}, {1, 1, 1}, {2, 2}, {3, 3}}
```

这里分成相邻元素不相等的子列表.

```
In[5]:= Split[{1, 4, 1, 1, 1, 2, 2, 3, 3}, Unequal]
Out[5]= {{1, 4, 1}, {1}, {1, 2}, {2, 3}, {3}}
```

Partition 遍历一个列表，并把相邻的元素分成子列表. 默认情况下，它没有包含任何令原列表悬空的子列表.

这里在出现任何悬空之前停止操作.

```
In[6]:= Partition[{a, b, c, d, e}, 2]
Out[6]= {{a, b}, {c, d}}
```

以下例子也同样适用.

```
In[7]:= Partition[{a, b, c, d, e}, 3, 1]
Out[7]= {{a, b, c}, {b, c, d}, {c, d, e}}
```

用户可以告诉 Partition 包含令原列表末尾悬空的子列表. 默认情况下，通过周期性地处理原来的列表，它可以填入其他额外的元素. 它也可以根据用户指定的元素进行填充.

这里包含了另外的子列表，并认为原先的列表是周期性的.

```
In[8]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}]

Out[8]= {{a, b, c}, {b, c, d},
         {c, d, e}, {d, e, a}, {e, a, b}}
```

这里原列表使用元素 x 填充.

```
In[9]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}, x]

Out[9]= {{a, b, c}, {b, c, d},
         {c, d, e}, {d, e, x}, {e, x, x}}
```

这里使用元素 x 和 y 周期性地填充.

```
In[10]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}, {x, y}]

Out[10]= {{a, b, c}, {b, c, d},
          {c, d, e}, {d, e, y}, {e, y, x}}
```

这里没有引入填充的方法，产生了具有不同长度的子列表.

```
In[11]:= Partition[{a, b, c, d, e}, 3, 1, {1, 1}, {}]

Out[11]= {{a, b, c}, {b, c, d}, {c, d, e}, {d, e}, {e}}
```

用户可以认为 Partition 是通过在原列表使用模版并选出元素来提取子列表. 用户可以告诉 Partition 在什么位置开始和停止该过程.

这里给出所有与原来的列表重叠的子列表.

```
In[12]:= Partition[{a, b, c, d}, 3, 1, {-1, 1}, x]

Out[12]= {{x, x, a}, {x, a, b}, {a, b, c},
          {b, c, d}, {c, d, x}, {d, x, x}}
```

这里只允许开头的重叠.

```
In[13]:= Partition[{a, b, c, d}, 3, 1, {-1, -1}, x]
Out[13]= {{x, x, a}, {x, a, b}, {a, b, c}, {b, c, d}}
```

`Partition[list, n, d]` 或 `Partition[list, n, d, {1, -1}]`	只保留没有悬空的子列表
`Partition[list, n, d, {1, 1}]`	允许在末尾有悬空
`Partition[list, n, d, {-1, -1}]`	允许在开头有悬空
`Partition[list, n, d, {-1, 1}]`	允许在开头和末尾都有悬空
`Partition[list, n, d, {`k_L`, `k_R`}]`	指定第一个和最后一个子列表的对齐
`Partition[list, n, d, spec]`	通过在 *list* 中周期性地重复元素进行填充
`Partition[list, n, d, spec, x]`	通过重复元素 x 填充
`Partition[list, n, d, spec, {`x_1`, `x_2`, ...}]`	通过周期性地重复 x_i 填充
`Partition[list, n, d, spec, {}]`	不使用填充

指定对齐和填充.

对齐规格 {k_L, k_R} 告诉 `Partition` 给出子列表的序列，其中原列表的第一个元素在第一个子列表的位置 k_L 出现，而原列表的最后一个元素在最后一个子列表的位置 k_R 出现.

在第一个子列表中，令 a 出现在位置1.

```
In[14]:= Partition[{a, b, c, d}, 3, 1, {1, 1}, x]

Out[14]= {{a, b, c}, {b, c, d}, {c, d, x}, {d, x, x}}
```

这里令 a 出现在第一个子列表的位置2.

```
In[15]:= Partition[{a, b, c, d}, 3, 1, {2, 1}, x]

Out[15]= {{x, a, b}, {a, b, c},
          {b, c, d}, {c, d, x}, {d, x, x}}
```

这里令 a 首先出现在位置4.

```
In[16]:= Partition[{a, b, c, d}, 3, 1, {4, 1}, x]

Out[16]= {{x, x, x}, {x, x, a}, {x, a, b},
          {a, b, c}, {b, c, d}, {c, d, x}, {d, x, x}}
```

从给定的列表周期性地进行填充.

```
In[17]:= Partition[{a, b, c, d}, 3, 1, {4, 1}, {x, y}]

Out[17]= {{y, x, y}, {x, y, a}, {y, a, b},
          {a, b, c}, {b, c, d}, {c, d, x}, {d, x, y}}
```

函数如 ListConvolve 使用与 Partition 相同的对齐和填充规格.

在一些情况下，对列表进行明确的填充插入可能是方便的. 这可以通过使用 PadLeft 和 PadRight 实现.

PadLeft[*list*, *n*]	通过在左边插入零进行填充，以达到长度 *n*
PadLeft[*list*, *n*, *x*]	通过重复元素 *x* 进行填充
PadLeft[*list*, *n*, {x_1, x_2, ...}]	通过周期性地重复 x_i 进行填充
PadLeft[*list*, *n*, *list*]	通过周期性地重复 *list* 进行填充
PadLeft[*list*, *n*, *padding*, *m*]	在右边留出 *m* 个元素大小的空白
PadRight[*list*, *n*]	在右边插入零进行填充

填充列表.

这里对列表进行填充，使其长度为6.

```
In[18]:= PadLeft[{a, b, c}, 6]
Out[18]= {0, 0, 0, a, b, c}
```

这里以填充方式插入 {x, y}.

```
In[19]:= PadLeft[{a, b, c}, 6, {x, y}]
Out[19]= {x, y, x, a, b, c}
```

这里在右边留出大小为3的空白.

```
In[20]:= PadLeft[{a, b, c}, 10, {x, y}, 3]
Out[20]= {y, x, y, x, a, b, c, x, y, x}
```

PadLeft、PadRight 和 Partition 都可以用于嵌套列表.

这里生成一个**3x3**数组.

```
In[21]:= PadLeft[{{a, b}, {e}, {f}}, {3, 3}, x]

Out[21]= {{x, a, b}, {x, x, e}, {x, x, f}}
```

这里使用偏移量**1**把数组分成**2x2**的块.

```
In[22]:= Partition[%, {2, 2}, {1, 1}]

Out[22]= {{{{x, a}, {x, x}}, {{a, b}, {x, e}}},
          {{{x, x}, {x, x}}, {{x, e}, {x, f}}}}
```

如果用户给出一个嵌套列表作为填充规格，在每一层周期性地选出它的元素.

这里周期性地填入填充列表的拷贝.

```
In[23]:= PadLeft[{{a, b}, {e}, {f}},
          {4, 4}, {{x, y}, {z, w}}]

Out[23]= {{x, y, x, y}, {z, w, a, b},
          {x, y, x, e}, {z, w, z, f}}
```

以下是一个仅包含填充的列表.

```
In[24]:= PadLeft[{{}}, {4, 4}, {{x, y}, {z, w}}]

Out[24]= {{x, y, x, y}, {z, w, z, w},
          {x, y, x, y}, {z, w, z, w}}
```

相关教程

- 操作列表

教程专集

稀疏数组：列表的处理

在 ***Mathematica*** 中，列表通过给出元素的明确集合来指定. 但是当处理大型数组时，能够指明特定位置处的元素值，并将其余元素指定为默认值，常常是有用的. 在 ***Mathematica*** 中可以使用 SparseArray 对象实现.

$\{e_1, e_2, \ldots\}$,
$\{\{e_{11}, e_{12}, \ldots\}, \ldots\}$
, ...
普通列表

SparseArray[{pos_1 -> val_1, pos_2 -> val_2, ...}]
稀疏数组

普通列表和稀疏数组.

这里指定了一个稀疏数组.

```
In[1]:= SparseArray[{2 -> a, 5 -> b}]
Out[1]= SparseArray[<2>, {5}]
```

这是普通列表的形式.

```
In[2]:= Normal[%]
Out[2]= {0, a, 0, 0, b}
```

这里指定了一个二维稀疏数组.

```
In[3]:= SparseArray[
          {{1, 2} -> a, {3, 2} -> b, {3, 3} -> c}]
Out[3]= SparseArray[<3>, {3, 3}]
```

这是由列表组成的普通列表.

```
In[4]:= Normal[%]
Out[4]= {{0, a, 0}, {0, 0, 0}, {0, b, c}}
```

SparseArray[*list*]	*list* 的稀疏数组版本
SparseArray[{pos_1->val_1, pos_2->val_2, ...}]	在位置 pos_i 上值为 val_i 的稀疏数组
SparseArray[{pos_1, pos_2, ...}->{val_1, val_2, ...}]	同样的稀疏数组
SparseArray[Band[{*i*, *j*}]->*val*]	值为 *val* 的带状稀疏矩阵
SparseArray[*data*, {d_1, d_2, ...}]	$d_1 \times d_2 \times \ldots$ 稀疏数组
SparseArray[*data*, *dims*, *val*]	默认值为 *val* 的稀疏数组
Normal[*array*]	*array* 的普通列表版本
ArrayRules[*array*]	*array* 的位置值规则

创建和转换稀疏数组.

这里产生列表的稀疏数组版本.

```
In[5]:= SparseArray[{a, b, c, d}]
Out[5]= SparseArray[<4>, {4}]
```

将其转换为一个普通列表.

```
In[6]:= Normal[%]
Out[6]= {a, b, c, d}
```

这里生产一个默认值为 x，长度为7的稀疏数组.

```
In[7]:= SparseArray[{3 -> a, 5 -> b}, 7, x]
Out[7]= SparseArray[<2>, {7}, x]
```

这是相应的普通列表.

```
In[8]:= Normal[%]
Out[8]= {x, x, a, x, b, x, x}
```

这里显示了稀疏数组中采用的规则.

```
In[9]:= ArrayRules[%%]
Out[9]= {{3} → a, {5} → b, {_} → x}
```

这里创建一个带状矩阵.

```
In[10]:= SparseArray[
           {Band[{1, 1}] → x, Band[{2, 1}] → y},
           {5, 5}] // MatrixForm
```

t[10]//MatrixForm=

$$\begin{pmatrix} x & 0 & 0 & 0 & 0 \\ y & x & 0 & 0 & 0 \\ 0 & y & x & 0 & 0 \\ 0 & 0 & y & x & 0 \\ 0 & 0 & 0 & y & x \end{pmatrix}$$

SparseArray 的重要特征是用户指定的位置可以是模式.

这里指定了一个4×4稀疏数组，与 {i_, i_} 匹配的每个位置为 1.

```
In[11]:= SparseArray[{i_, i_} -> 1, {4, 4}]

Out[11]= SparseArray[<4>, {4, 4}]
```

该结果是一个4×4的单位矩阵.

```
In[12]:= Normal[%]

Out[12]= {{1, 0, 0, 0}, {0, 1, 0, 0},
          {0, 0, 1, 0}, {0, 0, 0, 1}}
```

这是一个具有额外元素的单位矩阵.

```
In[13]:= Normal[SparseArray[
           {{1, 3} -> a, {i_, i_} -> 1}, {4, 4}]]

Out[13]= {{1, 0, a, 0}, {0, 1, 0, 0},
          {0, 0, 1, 0}, {0, 0, 0, 1}}
```

整个第3列为 a.

```
In[14]:= Normal[SparseArray[
           {{_, 3} -> a, {i_, i_} -> 1}, {4, 4}]]

Out[14]= {{1, 0, a, 0}, {0, 1, a, 0},
          {0, 0, a, 0}, {0, 0, a, 1}}
```

可以认为 SparseArray[*rules*] 是采用所有可能位置的规格，然后应用 *rules* 来确定每个情况下的值. 通常情况下，列表中先给出的规则将被首先尝试.

这里产生一个随机对角线矩阵.

```
In[15]:= Normal[SparseArray[
           {{i_, i_} :> RandomReal[]}, {3, 3}]]

Out[15]= {{0.0560708, 0, 0},
          {0, 0.6303, 0}, {0, 0, 0.359894}}
```

用户可以具有值取决于索引的规则.

```
In[16]:= Normal[SparseArray[i_ -> i^2, 10]]

Out[16]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

这里使用 p 对每个偶数位进行填充.

```
In[17]:= Normal[
          SparseArray[{_?EvenQ -> p, i_ -> i^2}, 10]]

Out[17]= {1, p, 9, p, 25, p, 49, p, 81, p}
```

用户可以使用涉及另一种选择的模式.

```
In[18]:= Normal[SparseArray[{1 | 3, 2 | 4} -> a, {4, 4}]]

Out[18]= {{0, a, 0, a}, {0, 0, 0, 0},
          {0, a, 0, a}, {0, 0, 0, 0}}
```

用户也可以对模式给出条件.

```
In[19]:= Normal[SparseArray[i_ /; 3 < i < 7 -> p, 10]]

Out[19]= {0, 0, 0, p, p, p, 0, 0, 0, 0}
```

这里生成一个带状对角矩阵.

```
In[20]:= Normal[SparseArray[
           {{i_, j_} /; Abs[i - j] < 2 -> i + j}, {5, 5}]]

Out[20]= {{2, 3, 0, 0, 0}, {3, 4, 5, 0, 0}, {0, 5, 6, 7, 0}
          {0, 0, 7, 8, 9}, {0, 0, 0, 9, 10}}
```

以下是另一种方式.

```
In[21]:= Normal[
           SparseArray[{Band[{1, 1}] → {2, 4, 6, 8, 10},
             Band[{1, 2}] → {3, 5, 7, 9},
             Band[{2, 1}] → {3, 5, 7, 9}}, {5, 5}]]

Out[21]= {{2, 3, 0, 0, 0}, {3, 4, 5, 0, 0}, {0, 5, 6, 7, 0}
          {0, 0, 7, 8, 9}, {0, 0, 0, 9, 10}}
```

为了许多不同的目的，***Mathematica*** 用类似它们所对应的普通列表的处理方式处理 SparseArray 对象. 因此，例如，如果用户要求一个稀疏数组对象的部分元素，***Mathematica*** 将用类似在对应的普通列表中获取部分元素的方式进行.

这里产生一个稀疏数组对象.

```
In[22]:= s = SparseArray[{2 -> a, 4 -> b, 5 -> c}, 10]
Out[22]= SparseArray[<3>, {10}]
```

以下是对应的普通列表.

```
In[23]:= Normal[s]
Out[23]= {0, a, 0, b, c, 0, 0, 0, 0, 0}
```

稀疏数组的部分元素类似于对应的普通列表的部分元素.

```
In[24]:= s[[2]]
Out[24]= a
```

该部分元素具有默认值0.

```
In[25]:= s[[3]]
Out[25]= 0
```

许多操作用类似普通列表的方式处理 SparseArray 对象. 在可能的情况下，它们给出稀疏矩阵作为结果.

这里给出一个稀疏数组.

```
In[26]:= 3 s + x
Out[26]= SparseArray[<3>, {10}, x]
```

以下是对应的普通列表.

```
In[27]:= Normal[%]
Out[27]= {x, 3 a + x, x, 3 b + x, 3 c + x, x, x, x, x, x}
```

Dot 直接对稀疏数组对象进行操作.

```
In[28]:= s.s
Out[28]= a^2 + b^2 + c^2
```

用户可以混合稀疏数组和普通列表.

```
In[29]:= s.Range[10]
Out[29]= 2 a + 4 b + 5 c
```

Mathematica 用具有头部 SparseArray 的表达式表示稀疏数组. 当计算一个稀疏数组的时候，它就自动转换为具有结构 SparseArray[Automatic, *dims*, *val*, ...] 的优化标准形式.

然而，该结构很少是明显的，由于建立类似 Length 的操作可以对相应的普通列表给出结果，而不是对原始SparseArray 表达式结构给出结果.

这里产生一个稀疏数组.

```
In[30]:= t = SparseArray[{1 -> a, 5 -> b}, 10]
Out[30]= SparseArray[<2>, {10}]
```

以下是内部优化表达式结构.

```
In[31]:= InputForm[%]

Out[31]//InputForm=
SparseArray[Automatic, {10}, 0,    {1, {{0, 2
```

Length 给出对应的普通列表的长度.

```
In[32]:= Length[t]

Out[32]= 10
```

Map 也对单个数值进行操作.

```
In[33]:= Normal[Map[f, t]]

Out[33]= {f[a], f[0], f[0], f[0],
          f[b], f[0], f[0], f[0], f[0], f[0]}
```

相关教程

- 操作列表

教程专集

- **Core Language**

表达式

以下是内部优化表达式结构.

```
In[31]:= InputForm[%]
```

```
SparseArray[Automatic, {10}, 0,    {1, {{0, 2}, {{1
```

Length 给出对应的普通列表的长度.

```
In[32]:= Length[t]
Out[32]= 10
```

Map 也对单个数值进行操作.

```
In[33]:= Normal[Map[f, t]]
Out[33]= {f[a], f[0], f[0], f[0],
          f[b], f[0], f[0], f[0], f[0], f[0]}
```

相关教程

- 操作列表

教程专集

- **Core Language**

表达式

表达式的含义
表达式输入的特殊方式
表达式的项
同类列表的操作
表达式的树结构
表达式的层次结构

表达式

Mathematica 处理多种不同形式的对象：数学公式、列表及图形等. 尽管它们在形式上看起来有所不同，但是***Mathematica*** 以统一的方式来表达它们. 它们都是*表达式*.

Mathematica 的表达式的一个典型例子是 `f[x, y]`. 你可以用 `f[x, y]` 取表示一个数学函数 $f(x, y)$. 这个函数名是 `f`，这个函数有两个自变量 `x` 和 `y`.

不需要经常将表达式写为 $f[x, y, \ldots]$ 的形式. 例如，`x + y` 也是一个表达式. 当输入 `x + y` 时，***Mathematica*** 将它转换成标准形式 `Plus[x, y]`. 但当输出时，***Mathematica*** 仍然给出 `x + y`.

对 `^`（`Power`） 和 `/`（`Divide`）等运算也是一样的.

事实上，对任何输入， ***Mathematica*** 总是把它当作表达式处理.

`x+y+z`	`Plus[x,y,z]`
`x y z`	`Times[x,y,z]`
`x^n`	`Power[x,n]`
`{a,b,c}`	`List[a,b,c]`
`a->b`	`Rule[a,b]`
`a=b`	`Set[a,b]`

Mathematica 中的一些表达式.

可以通过 `FullForm[`*expr*`]` 函数来得到任何表达式的完全形式.

这是一个表达式.

```
In[1]:= x + y + z
Out[1]= x + y + z
```

这是该表达式的完全形式.

```
In[2]:= FullForm[%]
t[2]//FullForm=
        Plus[x, y, z]
```

这是另一个表达式.

```
In[3]:= 1 + x^2 + (y + z)^2
Out[3]= 1 + x^2 + (y + z)^2
```

它的完全形式由几个表达式组合而成.

```
In[4]:= FullForm[%]
```

```
t[4]//FullForm=
        Plus[1, Power[x, 2], Power[Plus[y, z], 2]]
```

对象 f 是表达式 $f[x, y, \ldots]$ 的头. 可以用 Head[*expr*] 去分离此表达式的头部. 特别当用 ***Mathematica*** 编程时，经常要测试一个表达式的头以确定这个表达式是什么.

Head 给出了函数名 f.

```
In[5]:= Head[f[x, y]]
Out[5]= f
```

这里 Head 给出了算符名.

```
In[6]:= Head[a + b + c]
Out[6]= Plus
```

任何表达式都有头部.

```
In[7]:= Head[{a, b, c}]
Out[7]= List
```

数字也具有头部.

```
In[8]:= Head[23 432]
Out[8]= Integer
```

可以通过头部来区别不同类型的数.

```
In[9]:= Head[345.6]
Out[9]= Real
```

Head[*expr*]	给出表达式的头部： $f[x, y]$ 的 f
FullForm[*expr*]	显示 ***Mathematica*** 中使用的表达式的完全形式

处理表达式的函数.

相关教程

- 表达式

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**

表达式的含义

Mathematica 中表达式的概念是按照严格统一的原则给出的. 在 ***Mathematica*** 中每一个对象都有同样的内在结构，这使得

尽管所有的表达式都有相同的基本结构，但仍然可以用多种不同的方式使用这些表达式. 下面就是对一些表达式的说明.

f 的含义	*x*, *y*, ... 的含义	举例
Function （函数）	自变量或参数	`Sin[x], f[x,y]`
命令	自变量或参数	`Expand[(x+1)^2]`
操作符	操作数	`x+y, a=b`
Head （头部）	元素	`{a,b,c}`
对象类型	内容	`RGBColor[r,g,b]`

一些表达式的说明.

在 ***Mathematica*** 中表达式常用来进行某一种运算. 因此，例如，`2 + 3` 表示 2 和 3 相加，而 `Factor[x^6 - 1]` 则进行因式分解.

然而在 ***Mathematica*** 中，表达式更重要的用途是保持一种结构，使得该结构能被其它函数调用. 形如 `{a, b, c}` 的表达式不表示运算，它仅仅保持含有三个元素的集合结构. 像 `Reverse` 或 `Dot` 这样的函数可以对它进行操作.

表达式 `{a, b, c}` 的完全形式是 `List[a, b, c]`. 其头部 `List` 不进行任何运算. 其功能是作为一种标记以说明表达式的结构“类型”.

在 ***Mathematica*** 中，可以用表达式来产生所需要的结构. 例如，可以用 `point[x, y, z]` 来表示三维空间中的点. 这里“函数” `point` 不进行任何运算. 它只代表三个坐标值的集合，并将其标为 `point`.

包”. 即使所有的表达式都具有相同的基本结构，也可以通过不同的头部来区分不同“类型”的表达式. 然后，就可以对不同类型的表达式进行各种变换和编程.

相关教程

- 表达式

教程专集

- **Core Language**

表达式输入的特殊方式

Mathematica 允许用特定的记号表示普通运算符. 例如，尽管在 ***Mathematica*** 内部用 `Plus[x, y]` 表示两个数的和，但可以用更方便的方式 $x + y$.

Mathematica 语言有确定的语法规则，按照这些规则将你的输入转化为内部格式. 语法的一个方面就是怎样将单个输入进行分组. 加入输入了表达式 `a + b^c`，***Mathematica*** 语法就指出这一表达式将按照标准的数学运算 `a + (b^c)` 进行，而不是 `(a + b)^c`. ***Mathematica*** 选择这一的分组是因为 `^` 比 `+` 有更高的*优先级*. 一般说来，具有较高优先级的变量比优先级低的变量先进行分组.

应当认识到，***Mathematica*** 的每一个输入都给出了一个确定的优先次序. 不仅通常的数学运算，而且用来在 ***Mathematica*** 程序中分隔表达式的 `->`、`:=`、或者分号也有优先次序.

"运算符的输入形式" 中的表格给出了按照递减的优先级排列的

行排列，目的是在应用中尽量减少圆括号.

例如，你会发现关系操作符如 < 比算术操作符如 + 的优先级低. 这样你就可以不用括号来写出表达式 x + y > 7.

但还有些情况必须使用括号. 例如，由于 ; 比 = 的优先级低，**Mathematica** 把 x = a; b 当作 (x = a); b. 为了表示 x = (a; b)，必须使用括号. 一般说来，应该尽量使用括号，当漏掉了括号时，**Mathematica** 会按另一种方式执行，其后果大不一样.

$f[x, y]$	$f[x, y]$ 的标准格式
$f@x$	$f[x]$ 的前缀格式
$x//f$	$f[x]$ 的后缀格式
$x \sim f \sim y$	$f[x, y]$ 的嵌入格式

Mathematica 中表达式的四种书写方式.

Mathematica 中具有多种常用的运算. $x + y$ 中的 + 是“中缀”运算. $-p$ 中的 - 是“前缀”运算. 当你要输入表达式，如 $f[x, y, \ldots]$ 时，**Mathematica** 允许你可以用中缀、前缀和后缀格式.

这一“后缀”形式与 f[x + y] 等价.

```
In[1]:= x + y // f
Out[1]= f[x + y]
```

当输入一表达式后想加入一个函数 N 时，可以用后缀形式.

```
In[2]:= 3^(1/4) + 1 // N
Out[2]= 2.31607
```

有时用中缀形式更容易理解.

```
In[3]:= {a, b, c}~Join~{d, e}
Out[3]= {a, b, c, d, e}
```

应该注意 // 的优先级非常低. 当把 // *f* 放在任何包含算术或逻辑操作符的表达式末尾时，则 *f* 是相对*整个表达式* 的. 因此，例如，x + y // f 是 f[x + y]，而不是 x + f[y].

前缀形式 @ 有相当高的优先级. 如 f@x + y 等价于 f[x] + y，而不是 f[x + y]. f[x + y] 的前缀形式为 f@(x + y).

相关指南

- 语法

相关教程

- 表达式

教程专集

- **Core Language**

表达式的项

列表是一个特殊的表达式，可以像给出表达式中的项一样给出列表中的项.

得到列表 {a, b, c} 中的第2个元素.

```
In[1]:= {a, b, c}[[2]]
Out[1]= b
```

用同样的方式得到 x + y + z 中的第2个元素.

```
In[2]:= (x + y + z)[[2]]
Out[2]= y
```

给出最后一个元素.

```
In[3]:= (x + y + z)[[-1]]
Out[3]= z
```

第 0 项表示求表达式的头部.

```
In[4]:= (x + y + z)[[0]]
Out[4]= Plus
```

可以像求嵌套列表中的元素一样，求表达式如 $f[g[a], g[b]]$ 中的项.

求表达式中的第 1 项.

```
In[5]:= f[g[a], g[b]][[1]]
Out[5]= g[a]
```

求表达式中的 {1, 1} 项.

```
In[6]:= f[g[a], g[b]][[1, 1]]
Out[6]= a
```

求表达式 1 + x^2 中的 {2, 1} 项.

```
In[7]:= (1 + x^2)[[2, 1]]
Out[7]= x
```

为了看 {2, 1} 是什么，可以观察该表达式的完全形式.

```
In[8]:= FullForm[1 + x^2]
t[8]//FullForm=
        Plus[1, Power[x, 2]]
```

从 FullForm 函数可以看出表达式中各项的标号是按照 ***Mathematica*** 的内部规则给出的，这些形式不一定和显示形式一致，对一些用 ***Mathematica*** 标准形式的代数表达式尤其是这样.

x / y 的内部形式为.

```
In[9]:= FullForm[x / y]
t[9]//FullForm=
        Times[x, Power[y, -1]]
```

规定项的内部形式.

```
In[10]:= (x / y)[[2]]
```

$$\text{Out[10]}= \frac{1}{y}$$

可以像列表中的项一样对表达式中的项进行操作.

用 x^2 来代替 a + b + c + d 中的第3项. 并自动重新进行排列.

```
In[11]:= ReplacePart[a + b + c + d, 3 -> x^2]
```

$$\text{Out[11]}= a + b + d + x^2$$

定义表达式.

```
In[12]:= t = 1 + (3 + x)^2 / y
```

$$\text{Out[12]}= 1 + \frac{(3 + x)^2}{y}$$

表达式 t 的完全形式.

```
In[13]:= FullForm[t]
```

```
t[13]//FullForm=
Plus[1,
 Times[Power[Plus[3, x], 2], Power[y, -1]]]
```

重新设置表达式 t 的项.

```
In[14]:= t[[2, 1, 1]] = x
```

$$\text{Out[14]}= x$$

现在 t 的形式改变了.

```
In[15]:= t
```

$$\text{Out[15]=}\quad 1+\frac{x^2}{y}$$

`Part[expr,n]` 或 `expr[[n]]`	表达式 *expr* 中的第 *n* 项
`Part[expr,{n1,n2,...}]` 或 `expr[[{n1,n2,...}]]`	表达式中项的组合
`Part[expr,n1;;n2]`	表达式的第 n_1 到第 n_2 项
`ReplacePart[expr,n->elem]`	用 *elem* 替换表达式 *expr* 中的第 *n* 项

处理表达式中项的函数.

"处理列表元素" 讨论怎样用列表标号来选择列表中的元素，用同样的方法可以选择表达式的项.

选择列表中第2、4个元素，并且给出这些元素的列表.

```
In[16]:= {a, b, c, d, e}[[{2, 4}]]
Out[16]= {b, d}
```

选择和式中的第2、4个元素，并给出其*和*.

```
In[17]:= (a + b + c + d + e)[[{2, 4}]]
Out[17]= b + d
```

表达式中的任一项都可以看作某一函数的一个自变量. 当用标号选择

这里选出列表中的第2到第4项.

```
In[18]:= {a, b, c, d, e}[[2 ;; 4]]
Out[18]= {b, c, d}
```

相关教程

- 表达式

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**

同类列表的操作

对 ***Mathematica*** 的任何表达式可以进行 **"列表"** 节中的大部分运算. 使用这些运算，使得我们能对表达式的结构进行各种处理.

这是一个对各项求和的表达式.

```
In[1]:= t = 1 + x + x^2 + y^2
Out[1]= 1 + x + x^2 + y^2
```

Take[t, 2] 选择 t 的前2项，正如 t 是一个列表一般.

```
In[2]:= Take[t, 2]
Out[2]= 1 + x
```

Length 给出 t 中元素的个数.

```
In[3]:= Length[t]
Out[3]= 4
```

用 FreeQ[*expr*, *form*] 测试 *form* 是否出现在 *expr* 中.

```
In[4]:= FreeQ[t, x]
Out[4]= False
```

这里给出 x 在 t 中的位置.

```
In[5]:= Position[t, x]
Out[5]= {{2}, {3, 1}}
```

处理表达式结构的所有函数都是作用于表达式的内部格式上. 可以用 FullForm[*expr*] 来观察这些格式. 它们可能与期望的显示形式不一样.

这是一个具有4个自变量的函数.

```
In[6]:= f[a, b, c, d]
Out[6]= f[a, b, c, d]
```

可以用 Append 添加一个变量.

```
In[7]:= Append[%, e]
Out[7]= f[a, b, c, d, e]
```

颠倒自变量的次序.

```
In[8]:= Reverse[%]
Out[8]= f[e, d, c, b, a]
```

"结构的操作" 节中讨论了适用于表达式的另外一些函数.

相关教程

- 表达式

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**

表达式的树结构

完全形式的表达式.

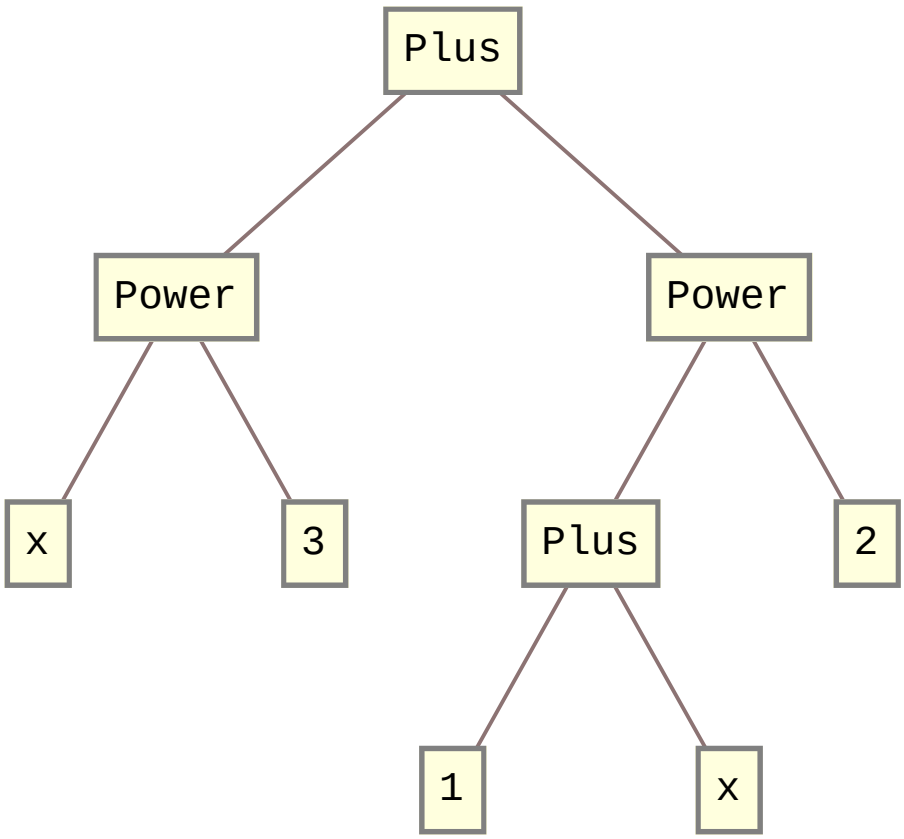
```
In[1]:= FullForm[x^3 + (1 + x)^2]
```

```
t[1]//FullForm=
Plus[Power[x, 3], Power[Plus[1, x], 2]]
```

TreeForm 函数以“树”结构的形式显示表达式.

```
In[2]:= TreeForm[x^3 + (1 + x)^2]
```

```
t[2]//TreeForm=
```

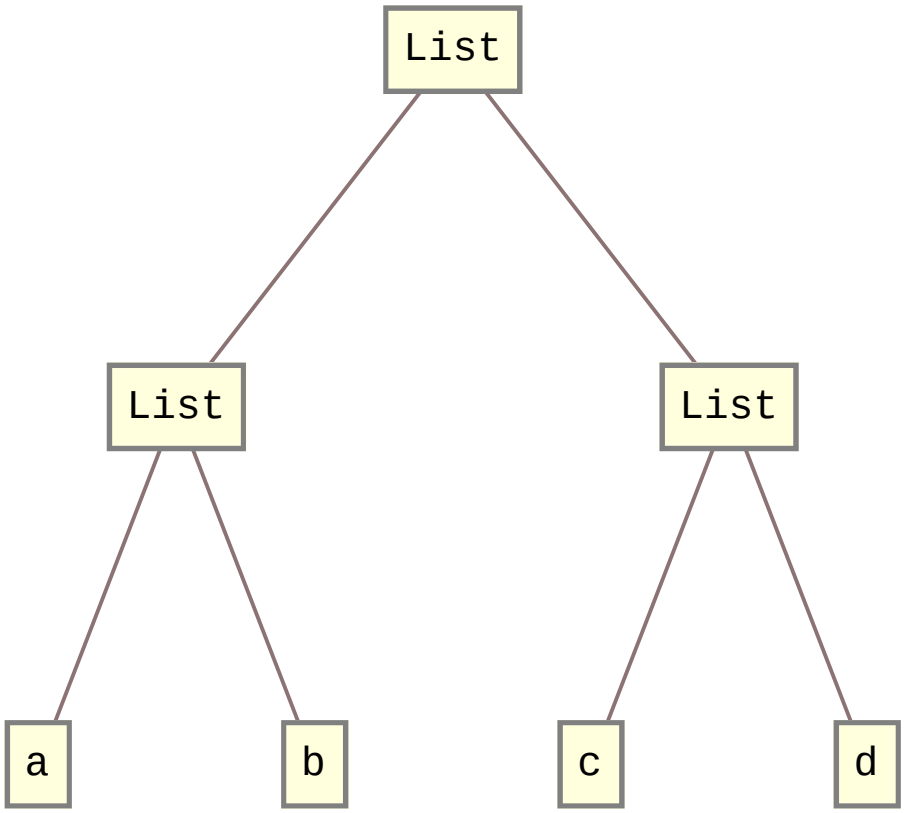


可以用 ***Mathematica*** 将任何一个表达式看作一个树. 在前面这个表达式中，树的最上层结点是 Plus. 这个结点有两个“分支”，x^3 和 (1 + x)^2. 从 x^3 结点，又有两个可以看作“树叶”的分支，x 和 3.

这个矩阵是一个有两级结构的树.

```
In[3]:= TreeForm[{{a, b}, {c, d}}]
```

t[3]//TreeForm=



更复杂的例子为.

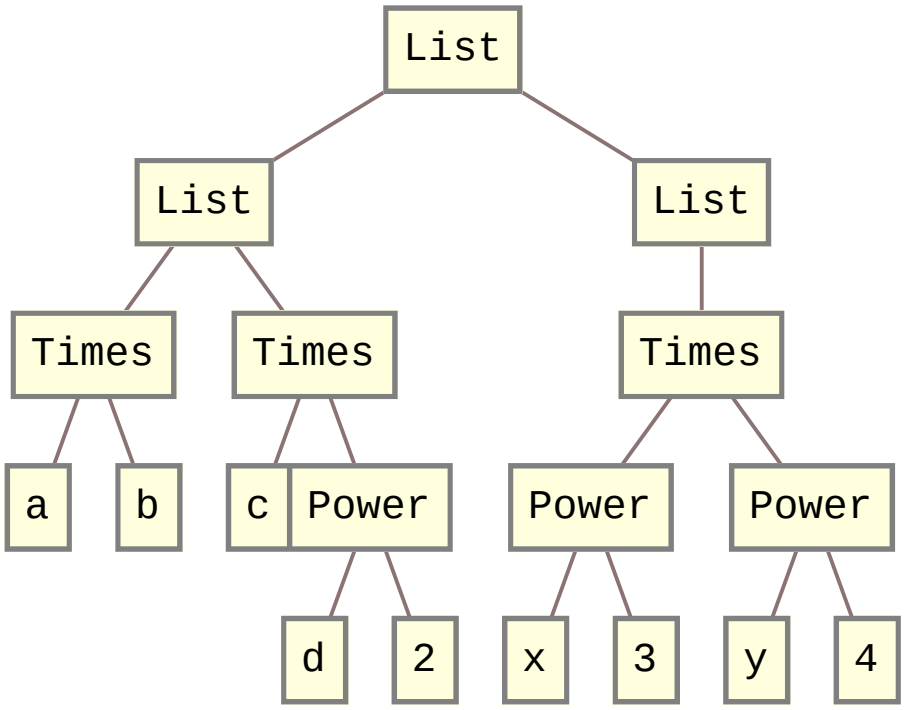
```
In[4]:= {{a b, c d^2}, {x^3 y^4}}
```

Out[4]= $\{\{a\,b,\ c\,d^2\},\ \{x^3\,y^4\}\}$

这个表达式的树结构有几级.

```
In[5]:= TreeForm[%]
```

Out[5]//TreeForm=



表达式每一项的标号在树结构中都有确定的意义. 从树顶层的结点往下，每个标号指出应该从那一个分支到要去的位置.

相关教程

- 表达式

教程专集

- **Core Language**

表达式的层次结构

Part 函数给出 *Mathematica* 表达式中确定的项. 但当某表达式有

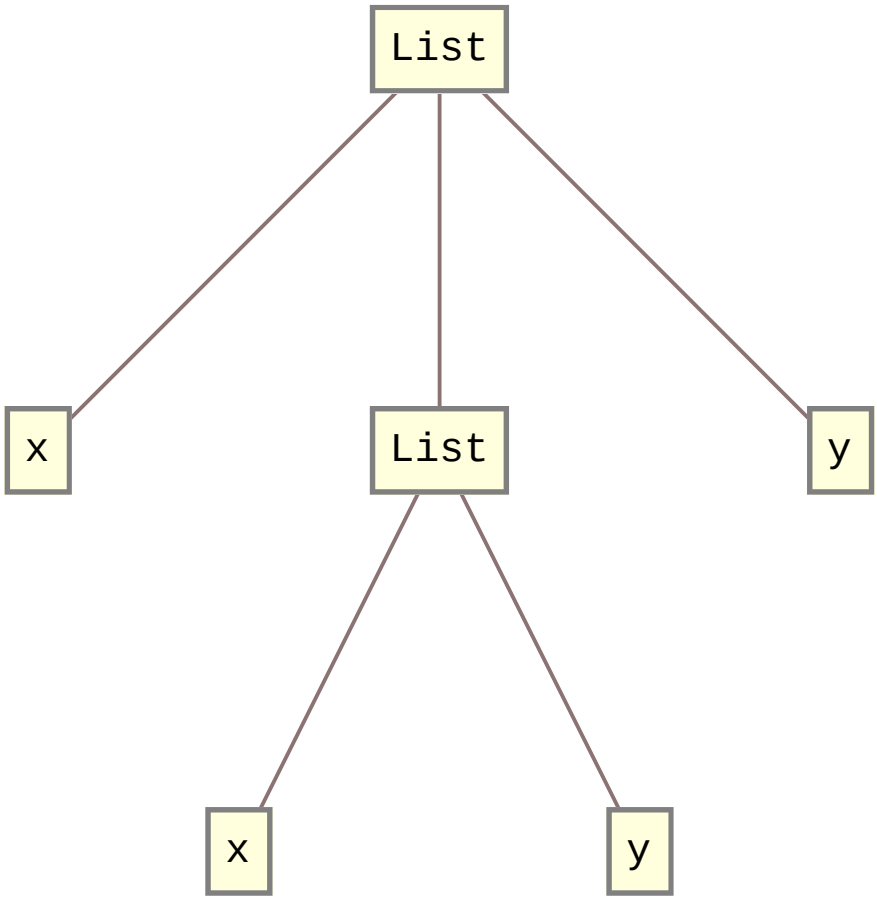
完全一致的结构时，能同时将表达式中的所有项表示出来是很方便并且最令人满意的.

层次结构 （**Levels**） 提供了指定 ***Mathematica*** 表达式中某一类项的途径. ***Mathematica*** 中的许多函数允许对表达式中某一层进行操作.

用树状结构给出的一个简单表达式.

```
In[1]:= (t = {x, {x, y}, y}) // TreeForm
```

t[1]//TreeForm=



在表达式 t 的第1层寻找 x，仅发现了1处.

```
In[2]:= Position[t, x, 1]
Out[2]= {{1}}
```

在表达式的前**2**层寻找 x，发现了**2**处.

```
In[3]:= Position[t, x, 2]
Out[3]= {{1}, {2, 1}}
```

在表达式的第**2**层寻找 x，仅发现了**1**处.

```
In[4]:= Position[t, x, {2}]
Out[4]= {{2, 1}}
```

Position[*expr*, *form*, *n*]	给出在表达式 *expr* 的前 *n* 层内 *form* 的位置
Position[*expr*, *form*, {*n*}]	仅给出第 *n* 层上 *form* 的位置

用层次结构控制位置的函数 Position.

从树结构的观点看层次时，将树的顶部看成第**0**层，一个表达式中的某一项所在的层次简单地说就是它到顶层的距离.

等价地说，第 *n* 层中的项就可以用 *n* 个标号来表示.

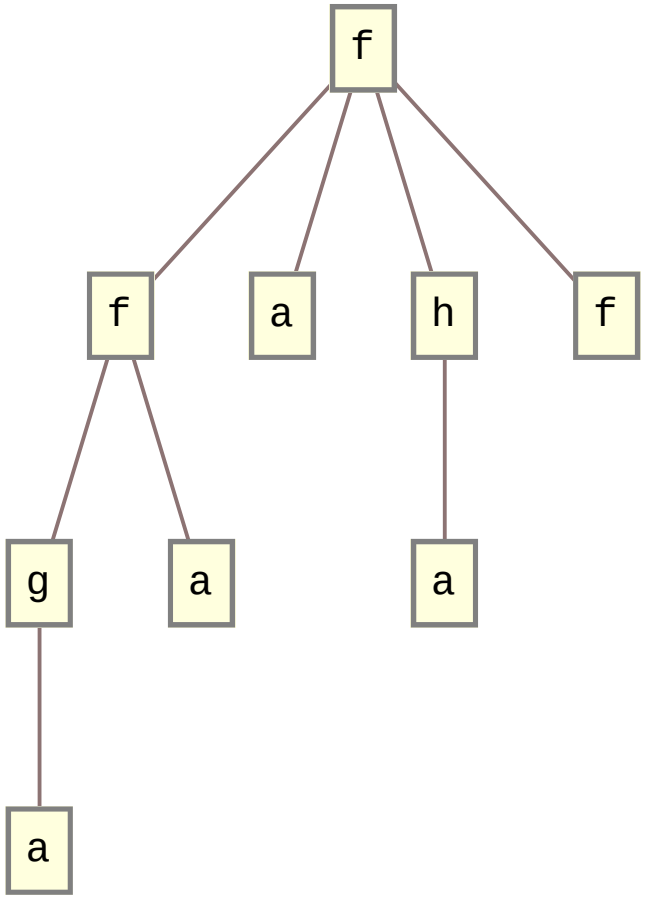
n	从第**1**层到第 n 层
`Infinity`	除**0**以外的所有层
$\{n\}$	仅第 n 层
$\{n_1, n_2\}$	从 n_1 到 n_2 层
`Heads->True`	包括头部
`Heads->False`	不包括头部

层次的标记方式.

这是用树结构显示的表达式.

```
In[5]:= (u = f[f[g[a], a], a, h[a], f]) // TreeForm
```

t[5]//TreeForm=



从第2层往下搜索 a.

```
In[6]:= Position[u, a, {2, Infinity}]
Out[6]= {{1, 1, 1}, {1, 2}, {3, 1}}
```

显示除头部以外 f 出现的位置.

```
In[7]:= Position[u, f, Heads -> False]
Out[7]= {{4}}
```

包括头部在内 f 出现的位置.

```
In[8]:= Position[u, f, Heads -> True]
Out[8]= {{0}, {1, 0}, {4}}
```

Level[*expr*, *lev*]	表达式 *expr* 中第 *lev* 层的项的列表
Depth[*expr*]	表达式 *expr* 中的总层数

测试和取出某些层.

列出 u 中前2层的项.

```
In[9]:= Level[u, 2]
Out[9]= {g[a], a, f[g[a], a], a, a, h[a], f}
```

给出 u 中第2层的项.

```
In[10]:= Level[u, {2}]
Out[10]= {g[a], a, a}
```

部 算起的项. -1层包含像数字和记号这样的对象的所有的树叶.

u 中 -1 层的项.

```
In[11]:= Level[u, {-1}]
Out[11]= {a, a, a, a, f}
```

表达式的"深度"指它的最大层数，可以用 TreeForm 来显示深度. 一般说来，一个表达式的 $-n$ 层由该表达式中所有深度为n 的子表达式组成.

g[a] 的深度是2.

```
In[12]:= Depth[g[a]]
Out[12]= 2
```

在 u 中 -2 层的项深度为2.

```
In[13]:= Level[u, {-2}]
Out[13]= {g[a], h[a]}
```

相关教程

- 表达式

教程专集

- **Core Language**

模式

引言
寻找与模式匹配的表达式
模式块的命名
模式中的表达式类型限制
限制模式
有多种供选方案的模式
模式序列
有交换性和结合性的函数
变量个数不确定的函数
可选变量与默认变量
定义具有可选变量的函数
重复模式
逐字模式
常用表达式的模式
举例：定义积分函数

引言

Mathematica 中用大量的模式来代表各类表达式. 例如，模式 `f[x_]` 表示形如 `f[`*anything*`]` 的一族表达式.

模式的主要功能在于 ***Mathematica*** 中的许多操作不仅可以对单个表达式实现，也可以对代表某类表达式的模式进行操作.

用模式给出一类表达式的变换规则.

```
In[1]:= f[a] + f[b] /. f[x_] -> x^2
```

$$\text{Out[1]=}\quad a^2 + b^2$$

用模式在指定类中找出表达式的位置.

```
In[2]:= Position[{f[a], g[b], f[c]}, f[x_]]
Out[2]= {{1}, {3}}
```

Mathematica 模式的标志是 _ （传统上，***Mathematica*** 程序员称之为空位）. 其基本规则是代表任意表达式. 对大多数键盘，下划线 _ 字符作为 - 破折号字符的**shift** 版本出现.

因此，例如，模式 `f[_]` 表示形如 `f[`*anything*`]` 的表达式. 并且模式 `f[x_]` 将表达式 *anything* 命名为 `x` 以便在变换规则中引用.

可以把空位放在表达式中的任何位置，得到与能以任何方式填充的表达式匹配的模式.

f[n_]	变量名为 n 的 f
f[n_,m_]	变量名为 n 和 m 的 f
x^n_	指数为 n 的 x 的幂
x_^n_	任何次幂的表达式
a_+b_	两个表达式的和
{a1_,a2_}	两个表达式组成的列表
f[n_,n_]	有两个*相同* 变量的 f

一些模式的例子.

构造具有任何结构的模式.

```
In[3]:= f[{a, b}] + f[c] /. f[{x_, y_}] -> p[x + y]
Out[3]= f[c] + p[a + b]
```

模式通常用来“取消”函数的变量. 定义 f[list_] 时，就需要用类似于 Part 的函数来选出列表的元素. 但是若知道集合中总是两个元素时，给出函数定义就比 f[{x_, y_}] 方便. 这就可以用 x 和 y 指代元素. 另外，当 f 的变量不是以两个表达式列表的形式出现时，***Mathematica*** 中刚定义的函数就不能使用.

定义求第一个元素对第二个元素的乘幂值的函数.

```
In[4]:= g[list_] := Part[list, 1]^Part[list, 2]
```

用模式定义更方便地定义上面的函数.

```
In[5]:= h[{x_, y_}] := x^y
```

Mathematica 的模式代表一类给定*结构* 的表达式. 当一个表达式的结构与模式的结构相同时，这个表达式就可以用在模式中. 即使数学

式来表示.

因此，例如，模式 (1 + x_)^2 可以表示 (1 + a)^2 或 (1 + b^3)^2，但不能表示 1 + 2 a + a^2. 尽管该表达式与 (1 + a)^2 相等，但它与模式 (1 + x_)^2 有不同的结构.

在不改变表达式值的情况下，利用模式指代具有某种结构的表达式这一点可以使我们建立表达式结构的变换规则.

应该意识到 ***Mathematica*** 是以结构相等来匹配表达式的，而不是以数学上的相等来匹配表达式的. 可以用 Expand 和 Factor 使 (1 + a)^2 和 1 + 2 a + a^2 具有相同的结构. 但正如 "将表达式化为标准形式" 中讨论的，没有一般的方法去判断任意两个数学表达式相等.

另一个例子是模式 x^_ 与表达式 x^2 匹配. 但它与 1 不匹配，尽管 1 可以写为 x^0. "可选变量与默认变量" 将讨论如何给出模式使这种情形匹配. 应该记住在任何情况下，***Mathematica*** 将按结构相等进行模式匹配.

x^n_ 仅与 x^2 和 x^3 匹配. 1 和 x 在数学上都可以写为 x^n，但不具有相同的结构.

```
In[6]:= {1, x, x^2, x^3} /. x^n_ -> r[n]
Out[6]= {1, x, r[2], r[3]}
```

另外，***Mathematica*** 中的模式只与由 FullForm 给出的表达式的完全形式进行匹配. 因此，例如，1 / x 的完全形式是 Power[x, -1]，它与模式 x_^n_ 匹配，但不能与模式 x_ / y_ 匹配，因为 x_ / y_ 的完全形式是 Times[x_, Power[y_, -1]]. 另外，"可选变量与默认变量" 将讨论用户如何构造对所有情况都匹配的模式

列表中的表达式含有 b 的幂，故可以使用变换规则.

```
In[7]:= {a / b, 1 / b^2, 2 / b^2} /. b^n_ -> d[n]
Out[7]= {a d[-1], d[-2], 2 d[-2]}
```

列表的完全形式.

```
In[8]:= FullForm[{a / b, 1 / b^2, 2 / b^2}]
Out[8]//FullForm=
List[Times[a, Power[b, -1]],
 Power[b, -2], Times[2, Power[b, -2]]]
```

尽管在模式匹配时，***Mathematica*** 认为 $x^1 = x$ 并不相当，但在匹配时，它考虑了交换律、结合律等性质.

在进行变换时，***Mathematica*** 使用了交换律和结合律.

```
In[9]:= f[a + b] + f[a + c] + f[b + d] /.
         f[a + x_] + f[c + y_] -> p[x, y]
Out[9]= f[b + d] + p[b, a]
```

到此为止，我们仅讨论了表示任何单独的表达式的模式对象，如 x_. 接下来，我们将讨论表示一类结构的模式.

相关指南

- 模式
- 规则与模式

相关教程

- 模式

- **Core Language**

寻找与模式匹配的表达式

`Cases[`*list*`,` *form*`]`	给出与 *form* 匹配的 *list* 中的元素
`Count[`*list*`,` *form*`]`	给出与 *form* 匹配的 *list* 中元素的个数
`Position[`*list*`,` *form*`,{1}]`	给出与 *form* 匹配的 *list* 中的元素的位置
`Select[`*list*`,`*test*`]`	给出使 *test* 的值为 `True` 的 *list* 中的元素
`Pick[`*list*`,`*sel*`,` *form*`]`	给出使 *sel* 的相应元素与 *form* 匹配的 *list* 中的元素

寻找与模式匹配元素.

给出与模式 x^_ 匹配的 **list** 中的元素.

```
In[1]:= Cases[{3, 4, x, x^2, x^3}, x^_]
```

$$\text{Out[1]=}\ \{x^2,\ x^3\}$$

此处给出了与模式匹配的元素的总数.

```
In[2]:= Count[{3, 4, x, x^2, x^3}, x^_]
```

```
Out[2]= 2
```

像 Cases 这样的函数不仅可以用于列表，而且可以用于任何表达式，可以指定项所在的层.

Cases[*expr*, *lhs*->*rhs*]	在 *expr* 中寻找与 *lhs* 匹配的元素，并对其进行变换
Cases[*expr*, *lhs*->*rhs*, *lev*]	测试指定层 *lev* 上表达式 *expr* 的项
Count[*expr*, *form*, *lev*]	给出指定层 *lev* 上与模式 *form* 匹配的项数
Position[*expr*, *form*, *lev*]	给出指定层 *lev* 上与模式 *form* 匹配的项的位置

寻找与模式相匹配的表达式的项.

输出指数 n 的集合.

```
In[3]:= Cases[{3, 4, x, x^2, x^3}, x^n_ -> n]
Out[3]= {2, 3}
```

模式 _Integer 可与任何整数匹配. 此处给出了各个层中出现的整数.

```
In[4]:= Cases[{3, 4, x, x^2, x^3}, _Integer, Infinity]
Out[4]= {3, 4, 2, 3}
```

Cases[*expr*, *form*, *lev*, *n*]	给出与模式 *form* 匹配的前 *n* 项
Position[*expr*, *form*, *lev*, *n*]	给出与 *form* 匹配的前 *n* 项的位置

限制所寻找项的数目.

给出任何层上出现的 x 的前2个幂的位置.

```
In[5]:= Position[{4, 4 + x^a, x^b, 6 + x^5},
          x^_, Infinity, 2]

Out[5]= {{2, 2}, {3}}
```

精确地给出位置，这用于 Extract 和 ReplacePart 等函数（见 "列表"）.

```
In[6]:= ReplacePart[{4, 4 + x^a, x^b, 6 + x^5}, zzz, %]

Out[6]= {4, 4 + zzz, zzz, 6 + x^5}
```

DeleteCases[*expr*, *form*]	删除表达式 *expr* 中与 *form* 匹配的元素
DeleteCases[*expr*, *form*, *lev*]	删除表达式 *expr* 的指定层 *lev* 中与 *form* 匹配的项

删除表达式中与模式匹配的项.

删除与 x^n_ 匹配的元素.

```
In[7]:= DeleteCases[{3, 4, x, x^2, x^3}, x^n_]

Out[7]= {3, 4, x}
```

删除所有层中出现的整数.

```
In[8]:= DeleteCases[{3, 4, x, 2 + x, 3 + x},
         _Integer, Infinity]

Out[8]= {x, x, x}
```

`ReplaceList[expr, lhs->rhs]` 寻找所有 *expr* 可以与 *lhs* 匹配的方式

在表达式中寻找与一个模式匹配的排列.

寻找能够写为两项之和的所有方式.

```
In[9]:= ReplaceList[a + b + c, x_ + y_ -> g[x, y]]

Out[9]= {g[a, b + c], g[b, a + c], g[c, a + b],
          g[a + b, c], g[a + c, b], g[b + c, a]}
```

寻找所有相当的元素对. 其中, 模式 ___ 表示元素的任意序列.

```
In[10]:= ReplaceList[{a, b, b, b, c, c, a},
          {___, x_, x_, ___} -> x]

Out[10]= {b, b, c}
```

相关教程

- 模式

教程专集

- **Core Language**

模式块的命名

当进行变换时，常常需要对模式块命名. 对象如 *x_* 表示任何表达式，并将此表达式命名为 *x*. 随后，就可以在变换规则的右端引用它.

一个要点是，当使用 *x_* 时，***Mathematica*** 要求在特定表达式中具有 *x* 名的表达式是相同的.

因此，`f[x_, x_]` 表示 `f` 的两个变量完全相同. 而 `f[_, _]` 可以表示形如 `f[x, y]` 的表达式，其中该表达式中的变量 *x* 和 *y* 不必相同.

这里只有当 `f` 的两个变量完全相同时才能进行变换.

```
In[1]:= {f[a, a], f[a, b]} /. f[x_, x_] -> p[x]
Out[1]= {p[a], f[a, b]}
```

在 ***Mathematica*** 中，不仅可以对一个空位命名，而且可以对模式中的任何部分命名，一般，*x* : *pattern* 表示命名为 *x* 的模式. 在变换规则中，可以将这种机制用到模式的任何部分以便在变换规则的右端使用.

_	任何表达式
x _	命名为 *x* 的任何表达式
x : *pattern*	与 *pattern* 匹配的名为 *x* 的表达式

命名的模式.

对 _^_ 进行命名，可以将其作为一个整体在右端使用.

```
In[2]:= f[a^b] /. f[x : _^_] -> p[x]
```

```
Out[2]= p[a^b]
```

指数为 n，整体为 x.

```
In[3]:= f[a^b] /. f[x : _^n_] -> p[x, n]
```

```
Out[3]= p[a^b, b]
```

当模式中的两部分具有相同的名称时，这就使模式仅与对应部分相同的表达式匹配.

模式匹配两种情形.

```
In[4]:= {f[h[4], h[4]], f[h[4], h[5]]} /.
         f[h[_], h[_]] -> q
```

```
Out[4]= {q, q}
```

这里限制 f 的两个变量相同，故仅第一种情形与模式匹配.

```
In[5]:= {f[h[4], h[4]], f[h[4], h[5]]} /.
         f[x : h[_], x_] -> r[x]
```

```
Out[5]= {r[h[4]], f[h[4], h[5]]}
```

相关教程

- 模式

教程专集

模式中的表达式类型限制

可以通过头部来区分不同“类型”的表达式，如整数的头部为 `Integer`，而列表的头部为 `List`.

模式中，`_h` 和 `x_h` 表示具有头部 `h` 的表达式. 例如，`_Integer` 表示任何整数，而 `_List` 表示任何列表.

`x_h`	具有头部 `h` 的表达式
`x_Integer`	整数型
`x_Real`	实数型
`x_Complex`	复数型
`x_List`	列表型
`x_Symbol`	符号型

指定了对象头部的模式.

仅替换整型元素.

```
In[1]:= {a, 4, 5, b} /. x_Integer -> p[x]
Out[1]= {a, p[4], p[5], b}
```

定义 `f[x_Integer]` 和定义一个具有整型 `Integer` 变量的函数 `f` 一样.

此处定义一个自变量为整数的函数 gamma.

```
In[2]:= gamma[n_Integer] := (n - 1)!
```

当自变量是整数时，才能计算 gamma 的值.

```
In[3]:= gamma[4] + gamma[x]
Out[3]= 6 + gamma[x]
```

由于对象 4. 具有头部 Real，故无法计算.

```
In[4]:= gamma[4.]
Out[4]= gamma[4.]
```

定义一个指数为整数型的表达式的值.

```
In[5]:= d[x_^n_Integer] := n x^(n - 1)
```

仅当指数为整数时，此定义有效.

```
In[6]:= d[x^4] + d[(a + b)^3] + d[x^(1 / 2)]
Out[6]= 3 (a + b)^2 + 4 x^3 + d[Sqrt[x]]
```

相关教程

- 模式

教程专集

- **Core Language**

限制模式

Mathematica 中提供了对模式进行限制的一般方法. 这可以通过在模式后面加 /; *condition* 来实现，此运算符 /; 可读作“斜杠分号”、“每当”或“只要”，其作用是当所指定的 **condition** 值为 True 时模式才能使用.

pattern /; *condition*	当条件满足时，模式才匹配
lhs :> *rhs* /; *condition*	当条件满足时，才使用规则
lhs := *rhs* /; *condition*	当条件满足时，才使用定义

对模式和变换规则进行限制.

定义 fac，其自变量 n 必为正.

```
In[1]:= fac[n_ /; n > 0] := n!
```

当 fac 的自变量为正时，才计算.

```
In[2]:= fac[6] + fac[-4]
Out[2]= 720 + fac[-4]
```

给出列表中的负值元素.

```
In[3]:= Cases[{3, -4, 5, -2}, x_ /; x < 0]
Out[3]= {-4, -2}
```

可以在变换规则中用 /;，也可以在单个模式中用 /;. 可以将 /; *condition* 放在 := 定义域或 :> 规则后告诉 ***Mathematica*** 只有当指定的条件满足时才能使用此定义或规则. 但要注意 /; 不能放

定义"）.

要求自变量 n 为正的另一种定义方式.

```
In[4]:= fac2[n_] := n! /; n > 0
```

自变量为正的阶乘函数.

```
In[5]:= fac2[6] + fac2[-4]
Out[5]= 720 + fac2[-4]
```

运算符 /; 可以用来限制运算规则的使用范围. 典型的情况是，先定义与较广范围的表达式匹配的模式，然后在后面给出一些数学上的限制，使其最终匹配成功的表达式缩减到一个较小的范围.

这一规则仅适用于具有 v[x_, 1 - x_] 结构的表达式.

```
In[6]:= v[x_, 1 - x_] := p[x]
```

此表达式的结构可以用上面的规则.

```
In[7]:= v[a^2, 1 - a^2]
Out[7]= p[a^2]
```

此表达式的数学形式正确，但结构不一致，故无法使用前面的规则.

```
In[8]:= v[4, -3]
Out[8]= v[4, -3]
```

此规则适用于任何形如 w[x_, y_] 的表达式，但要求 y == 1 - x.

```
In[9]:= w[x_, y_] := p[x] /; y == 1 - x
```

新规则可以用于这一表达式.

```
In[10]:= w[4, -3]

Out[10]= p[4]
```

在定义模式或规则时，可以把 /; 放在不同的位置.例如，可以将 /; 放在规则的右端，其形式为 *lhs* :> *rhs* /; *condition*，也可以将其放在左端，其形式为 *lhs* /; *condition* -> *rhs*.还可以把它插在表达式 *lhs* 的中间.但要注意在指定条件中使用的模式名称必须在该条件涉及的模式中出现，否则在判断条件是否成立的过程中所使用的名称就不一定限制在与模式匹配的表达式值之中，这时 ***Mathematica*** 会使用一些全局变量，而不是取决于模式匹配的值.

例如，在 f[x_, y_] /; (x + y < 2) 的条件中将使用与 f[x_, y_] 匹配的 x 和 y 的值，而在f[x_ /; x + y < 2, y_] 的条件中将用一个全局变量 y 的值，而不是与模式匹配的 y 值.

当确信适当的名称定义了之后，通常将条件 /; 放在一个模式可能最小的项上是最有效的，因为 ***Mathematica*** 逐块对模式进行匹配，一旦发现 /; 条件不成立，就不再进行匹配.

将 /; 放在 x_ 处比放在整个模式后更有效.

```
In[11]:= Cases[{z[1, 1], z[-1, 1], z[-2, 2]},
           z[x_ /; x < 0, y_]]

Out[11]= {z[-1, 1], z[-2, 2]}
```

在这种情况下，需要在 /; 处加括号.

```
In[12]:= {1 + a, 2 + a, -3 + a} /. (x_ /; x < 0) + a -> p[x]

Out[12]= {1 + a, 2 + a, p[-3]}
```

通常用 /; 使模式和变换规则使用到具有某一性质的表达式上，在 ***Mathematica*** 中有一类函数去测试表达式的性质. 这类函数后有一个 Q，表明它们在“提问”.

IntegerQ[*expr*]	整数
EvenQ[*expr*]	偶数
OddQ[*expr*]	奇数
PrimeQ[*expr*]	素数
NumberQ[*expr*]	任何数
NumericQ[*expr*]	数字型
PolynomialQ[*expr*, {x_1, x_2, ...}]	关于 x_1, x_2, ... 的多项式
VectorQ[*expr*]	表示向量的列表
MatrixQ[*expr*]	表示矩阵的集合的列表
VectorQ[*expr*, NumericQ], MatrixQ[*expr*, NumericQ]	所有元素都是数字的向量和矩阵
VectorQ[*expr*, *test*], MatrixQ[*expr*, *test*]	对所有元素 *test* 的函数值都为 True 的向量和矩阵
ArrayQ[*expr*, *d*]	深度与 *d* 匹配的完全数组

测试表达式数学特性的一些函数.

对集合中的数字用变换规则.

```
In[13]:= {2.3, 4, 7 / 8, a, b} /. (x_ /; NumberQ[x]) -> x^2
```

$$\text{Out[13]= } \left\{5.29,\ 16,\ \frac{49}{64},\ a,\ b\right\}$$

此定义仅适用于整数向量.

```
In[14]:= mi[list_] := list^2 /; VectorQ[list, IntegerQ]
```

上面的定义仅能用于第一个元素.

```
In[15]:= {mi[{2, 3}], mi[{2.1, 2.2}], mi[{a, b}]}
Out[15]= {{4, 9}, mi[{2.1, 2.2}], mi[{a, b}]}
```

以 Q 结尾的 ***Mathematica*** 测试函数的一个重要特性是：当它无法确定一个表达式具有所指出的特性时，测试函数的返回值为 False.

4561 是个整数，所有其返回值为 True.

```
In[16]:= IntegerQ[4561]
Out[16]= True
```

因为 x 不是一个整数，故返回值为 False.

```
In[17]:= IntegerQ[x]
Out[17]= False
```

函数如 IntegerQ[x] 测试 x 是否是一个整数. 在 Null 的假定下，可以使用 Refine、Simplify 及相关函数对符号变量 x 进行推导.

SameQ[x, y] 或 x===y	x 与 y 相等
UnsameQ[x, y] 或 x=!=y	x 与 y 不等
OrderedQ[{a, b, ...}]	a、b ... 按标准顺序排列
MemberQ[expr, form]	form 与表达式 expr 中的一个元素匹配
FreeQ[expr, form]	form 与表达式 expr 中的任何元素不匹配
MatchQ[expr, form]	expr 与模式 form 匹配
ValueQ[expr]	定义了 expr 的一个值
AtomQ[expr]	expr 无任何子表达式

测试表达式结构特性的一些函数.

“==”意味着方程保持着符号形式；当表达式不同时，“ === ”输出 False.

```
In[18]:= {x == y, x === y}
Out[18]= {x == y, False}
```

n 不在集合 {x, x^n} 中.

```
In[19]:= MemberQ[{x, x^n}, n]
Out[19]= False
```

然而，{x, x^n} 不是完全不含 n.

```
In[20]:= FreeQ[{x, x^n}, n]
Out[20]= False
```

可以使用 FreeQ 对 h 定义线性规则.

```
In[21]:= h[a_ b_, x_] := a h[b, x] /; FreeQ[a, x]
```

从 h 中移出不含 x 的项.

```
In[22]:= h[a b x, x] + h[2 (1 + x) x^2, x]
Out[22]= a b h[x, x] + 2 h[x^2 (1 + x), x]
```

---

pattern?*test*	模式与 *test* 的结果为 True 的表达式匹配

限制模式的另一种方法.

pattern /; *condition* 通过所涉及模式名满足的条件确定是否可以匹配. *pattern*?*test* 通过检查函数 *test* 在表达式的值来确定是否可以匹配. 用 ? 比 /; 更方便.

用函数 NumberQ 测试该定义对 x_ 是否匹配.

```
In[23]:= p[x_?NumberQ] := x^2
```

当 p 的变量为数值时才进行运算.

```
In[24]:= p[4.5] + p[3 / 2] + p[u]
Out[24]= 22.5 + p[u]
```

更复杂的定义不要忘了函数两边的括号.

```
In[25]:= q[{x_Integer, y_Integer}?
            (Function[v, v.v > 4])] := qp[x + y]
```

该定义仅在一些情况下有效.

```
In[26]:= {q[{3, 4}], q[{1, 1}], q[{-5, -7}]}
Out[26]= {qp[7], q[{1, 1}], qp[-12]}
```

Except[*c*]	与任何非 *c* 表达式匹配的模式
Except[*c*, *patt*]	与 *patt* 匹配但非 *c* 的模式

具有例外情况的模式.

这里给出除0以外的所有元素.

```
In[27]:= Cases[{1, 0, 2, 0, 3}, Except[0]]
Out[27]= {1, 2, 3}
```

Except 可以把模式作为一个变量.

```
In[28]:= Cases[{a, b, 0, 1, 2, x, y}, Except[_Integer]]
Out[28]= {a, b, x, y}
```

这里选出所有非0整数.

```
In[29]:= Cases[{a, b, 0, 1, 2, x, y}, Except[0, _Integer]]
Out[29]= {1, 2}
```

Except[*c*] 在一定意义上是个非常普遍的模式：它与除了 *c* 以外的 *任意表达式* 匹配. 在许多情况下，我们需要使用 Except[*c*, *patt*]，其工作原理是从匹配 *patt* 的表达式开始，并且排除与 *c* 匹配的表达式.

- 模式
- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

有多种供选方案的模式

$patt_1 \mid patt_2 \mid \ldots$　　有多种形式的模式

给定有几种选择的模式.

当 h 的变量是 a 或 b 时产生 p.

```
In[1]:= h[a | b] := p
```

前两种情形下给出了 p.

```
In[2]:= {h[a], h[b], h[c], h[d]}
Out[2]= {p, p, h[c], h[d]}
```

可以在变换规则中用此方法.

```
In[3]:= {a, b, c, d} /. (a | b) -> p
Out[3]= {p, p, c, d}
```

可选项之一本身也是模式的一个例子.

```
In[4]:= {1, x, x^2, x^3, y^2} /. (x | x^_) -> q
```

Out[4]= $\{1, q, q, q, y^2\}$

在使用有可选项的模式时，在每个可选项中应该用同名.当形如（a[x_] | b[x_]）的模式与表达式匹配时，则必须有一个与 x 对应的表达式.当（a[x_] | b[y_]）匹配时，必须有与 x 或 y 对应的表达式，而且不匹配的必须是Sequence[ ].

用 f 将头部命名为 a 或 b.

```
In[5]:= {a[2], b[3], c[4], a[5]} /.
         (f : (a | b))[x_] -> r[f, x]
```

Out[5]= {r[a, 2], r[b, 3], c[4], r[a, 5]}

相关教程

- 模式

教程专集

- **Core Language**

模式序列

在有些情况下，可能需要指定比 $x__$ 或 $x..$ 更复杂的模式序列；对于这样的情况，您可以使用 PatternSequence[p_1, p_2, ...].

PatternSequence[$p_1, p_2, \ldots$] 与 p_1, p_2, … 匹配的变量序列

模式序列.

这里定义具有两个或多个变量的函数，并把前两个变量分为一组.

```
In[1]:= f[x : PatternSequence[_, _], y___] := p[{x}, {y}]
```

关于不同的变量数目，计算函数.

```
In[2]:= {f[1], f[1, 2], f[1, 2, 3, 4, 5]}
Out[2]= {f[1], p[{1, 2}, {}], p[{1, 2}, {3, 4, 5}]}
```

这里在列表中选出序列 a, b 的最长段.

```
In[3]:= {a, b, b, a, b, a, b, a, a, b} /.
          {___, x : Longest[PatternSequence[a, b] ..],
            ___} :> {x}
Out[3]= {a, b, a, b}
```

空序列，PatternSequence[]，有时对于指定一个可选变量是有用的.

这里选出恰好具有一个或两个变量的表达式.

```
In[4]:= {g[], g[1], g[1, 2], g[1, 2, 3]} /.
          x : g[_, _ | PatternSequence[]] :> p[x]
Out[4]= {g[], p[g[1]], p[g[1, 2]], g[1, 2, 3]}
```

相关指南

- 规则与模式

相关教程

- 模式

教程专集

- **Core Language**

有交换性和结合性的函数

在 ***Mathematica*** 中模式是按结构形式匹配的，结构的等价是一个相当复杂的问题. 在如 `Plus` 和 `Times` 的函数中，它需要考虑可交换性和结合性.

在模式匹配时，表达式 $x + y$ 和 $y + x$ 是相同的，故模式 `g[x_ + y_, x_]` 不仅可以与 `g[a + b, a]` 匹配，而且还可以与 `g[a + b, b]` 匹配.

这一表达式与模式有完全相同的形式.

```
In[1]:= g[a + b, a] /. g[x_ + y_, x_] -> p[x, y]
Out[1]= p[a, b]
```

此处，必须先将表达式写成 g[b + a, b] 的形式，以便它与模式有相同的结构.

```
In[2]:= g[a + b, b] /. g[x_ + y_, x_] -> p[x, y]
Out[2]= p[b, a]
```

在模式匹配中，涉及到有可结合性和可交换性的函数如 Plus 或 Times 时，***Mathematica*** 就测试变量的各种顺序来进行匹配. 有时会有几种形式，***Mathematica*** 就与先找到的形式进行匹配. 例如，h[x_ + y_, x_ + z_] 可以与 h[a + b, a + b] 按 x→a、y→b、z→b 或 x→b、y→a、z→a 进行匹配. ***Mathematica*** 先找到了情形 x→a、y→b、z→b，故用这种匹配形式.

这里可以按 x→a 或 x→b 匹配. 但 ***Mathematica*** 先找到了 x→a，故选用这一形式匹配.

```
In[3]:= h[a + b, a + b] /. h[x_ + y_, x_ + z_] -> p[x, y, z]
Out[3]= p[a, b, b]
```

ReplaceList 显示两种可能的匹配.

```
In[4]:= ReplaceList[h[a + b, a + b],
          h[x_ + y_, x_ + z_] -> p[x, y, z]]
Out[4]= {p[a, b, b], p[b, a, a]}
```

正如 "属性" 中讨论的，***Mathematica*** 可以给函数赋予一些属性以表明该函数在计算和匹配过程中是被怎样处理的. 例如，对函数赋予了 Orderless 属性后，它就有可交换性和对称性，允许在模式匹配中对变量重新组合.

Orderless	交换性：如 $f[b, c, a]$ 等价于 $f[a, b, c]$
Flat	结合性：如 $f[f[a], b]$ 等价于 $f[a, b]$
OneIdentity	恒等：如 $f[f[a]]$ 等价于 a
Attributes[f]	给出赋予 f 的属性
SetAttributes[f, *attr*]	给 f 赋予属性 *attr*
ClearAttributes[f, *attr*]	清除 f 的属性 *attr*

能赋给函数的属性.

Plus 具有 Orderless 和 Flat 等属性.

```
In[5]:= Attributes[Plus]
Out[5]= {Flat, Listable, NumericFunction,
         OneIdentity, Orderless, Protected}
```

定义 q 为可交换性的函数.

```
In[6]:= SetAttributes[q, Orderless]
```

q 的变量自动按顺序排列.

```
In[7]:= q[b, a, c]
Out[7]= q[a, b, c]
```

Mathematica 重新排列 q 函数的变量以进行匹配.

```
In[8]:= f[q[a, b], q[b, c]] /.
         f[q[x_, y_], q[x_, z_]] -> p[x, y, z]

Out[8]= p[b, a, c]
```

函数如 Plus 和 Times 具有交换性和结合性，这就可以对其变量任意加括号，如 x + (y + z) 与 x + y + z 等价，等等.

Mathematica 在进行匹配时，也考虑到了可结合性，如模式 g[x_ + y_] 可以和 g[a + b + c] 匹配，此时 x→a 并且 y→(b + c).

将 g 的变量写为 a + (b + c) 以便和模式匹配.

```
In[9]:= g[a + b + c] /. g[x_ + y_] -> p[x, y]

Out[9]= p[a, b + c]
```

如果没有其他限制时，***Mathematica*** 把 x_ 与和式中的第一个元素匹配.

```
In[10]:= g[a + b + c + d] /. g[x_ + y_] -> p[x, y]

Out[10]= p[a, b + c + d]
```

这里给出了所有的匹配形式.

```
In[11]:= ReplaceList[g[a + b + c], g[x_ + y_] -> p[x, y]]

Out[11]= {p[a, b + c], p[b, a + c], p[c, a + b],
          p[a + b, c], p[a + c, b], p[b + c, a]}
```

这里要求 x_ 与 b + d 匹配.

```
In[12]:= g[a + b + c + d, b + d] /. g[x_ + y_, x_] -> p[x, y]
Out[12]= p[b + d, a + c]
```

一般情况下，当一个规则中的模式覆盖了函数的所有变量时才能按此规则进行变换，但对具有结合属性的函数而言，没有覆盖其全部变量时，有时也可以进行变换.

没有覆盖全部变量，也可以使用这一规则.

```
In[13]:= a + b + c /. a + c -> p
Out[13]= b + p
```

合并和式中的前两项.

```
In[14]:= u[a] + u[b] + v[c] + v[d] /. u[x_] + u[y_] -> u[x + y
Out[14]= u[a + b] + v[c] + v[d]
```

像 Plus 和 Times 这样的函数既有结合性也有交换性，而像 Dot 这样的函数只有结合性却没有交换性.

x_ 和 y_ 能匹配点积中的所有项.

```
In[15]:= a.b.c.d.a.b /. x_.y_.x_ -> p[x, y]
Out[15]= p[a.b, c.d]
```

给函数 r 赋予属性 Flat.

```
In[16]:= SetAttributes[r, Flat]
```

Mathematica 将表达式写成 r[r[a, b], r[a, b]] 的形式，以

```
In[17]:= r[a, b, a, b] /. r[x_, x_] -> rp[x]
Out[17]= rp[r[a, b]]
```

Mathematica 将表达式写为 r[a, r[r[b], r[b]], c] 的形式以便与模式匹配.

```
In[18]:= r[a, b, b, c] /. r[x_, x_] -> rp[x]
Out[18]= r[a, rp[r[b]], c]
```

在不具有结合性的函数中，模式 x_ 仅与函数的一个变量匹配. 而在具有结合性的函数 $f[a, b, c, \ldots]$ 中，x_ 可以与多个变量的对象如 $f[b, c]$ 匹配. 在 x_ 与具有结合性的函数的一个变量匹配时，就需要弄清楚它是与变量 a 匹配，还是与 $f[a]$ 匹配. ***Mathematica*** 规定当函数具有属性 OneIdentity 时，为第一种，其余为第二种.

给函数 r 赋予属性 OneIdentity.

```
In[19]:= SetAttributes[r, OneIdentity]
```

这里 x_ 与单个变量匹配.

```
In[20]:= r[a, b, b, c] /. r[x_, x_] -> rp[x]
Out[20]= r[a, rp[b], c]
```

函数 Plus，Times 和 Dot 都具有属性 OneIdentity，这反映了 Plus[x] 与 x 等价. 但在表示数学对象时，处理没有 OneIdentity 属性的可结合的函数常常是方便的.

相关教程

- **Core Language**

变量个数不确定的函数

当 f 不可结合时，模式 $f[x_, y_]$ 仅代表恰有两个变量的函数. 有时还需要建立具有任意数目的自变量的函数.

这可以通过多重空位来实现. 一个空位 x_ 表示一个 ***Mathematica*** 表达式，两个空位 x__ 表示多个表达式.

这里 x__ 表示一系列表达式 (a, b, c).

```
In[1]:= f[a, b, c] /. f[x__] -> p[x, x, x]
Out[1]= p[a, b, c, a, b, c, a, b, c]
```

从 h 中挑选重复元素的更复杂的定义.

```
In[2]:= h[a___, x_, b___, x_, c___] := hh[x] h[a, b, c]
```

应用这一定义找出两对元素.

```
In[3]:= h[2, 3, 2, 4, 5, 3]
Out[3]= h[4, 5] hh[2] hh[3]
```

双空位 __ 表示一个或多个表达式的序列. 三空位 ___ 表示零个或多个表达式序列. 在使用三空位时，很容易导致死循环这类错误. 例如，定义 p[x , v] := p[x] q[v]，接下来输入 p[a] 将进

尽量地少用三空位.

`_`	单一表达式
`x_`	名为 `x` 的表达式
`__`	一个或多个表达式序列
`x__`	名为 `x` 的表达式列
`x__h`	头部为 `h` 的表达式列
`___`	零个或多个表达式序列
`x___`	名为 `x` 的零个或多个表达式序列
`x___h`	头部为 `h` 的零个或多个表达式序列

不同类型的模式.

像 `Plus` 和 `Times` 这样具有结合性的函数，***Mathematica*** 自动处理变量的数目，不需要使用双空位和三空位，如 "有交换性和结合性的函数" 中讨论的.

在使用多重空位时，对特定的表达式有不同的匹配方式. 默认情况下，***Mathematica*** 总是先将模式中的第一个多空位与变量的最短序列匹配. 可以通过在模式的项周围使用 `Longest` 或者 `Shortest` 括起来来改变顺序.

`Longest[p]`	匹配与模式 `p` 一致的最长序列
`Shortest[p]`	匹配与模式 `p` 一致的最短序列

控制匹配的顺序.

这里给出了 ***Mathematica*** 所有可能的匹配.

```
In[4]:= ReplaceList[f[a, b, c, d],
         f[x__, y__] -> g[{x}, {y}]]

Out[4]= {g[{a}, {b, c, d}],
         g[{a, b}, {c, d}], g[{a, b, c}, {d}]}
```

这里命令 ***Mathematica*** 对 $x__$ 先尝试最长的匹配.

```
In[5]:= ReplaceList[f[a, b, c, d],
         f[Longest[x__], y__] -> g[{x}, {y}]]

Out[5]= {g[{a, b, c}, {d}],
         g[{a, b}, {c, d}], g[{a}, {b, c, d}]}
```

许多枚举类型可以使用具有不同模式类型的 `ReplaceList` 来实现.

```
In[6]:= ReplaceList[f[a, b, c, d], f[___, x__] -> g[x]]

Out[6]= {g[a, b, c, d], g[b, c, d], g[c, d], g[d]}
```

这里列举出所有至少具有一个元素的子列表.

```
In[7]:= ReplaceList[f[a, b, c, d],
         f[___, x__, ___] -> g[x]]

Out[7]= {g[a], g[a, b], g[a, b, c], g[a, b, c, d], g[b],
         g[b, c], g[b, c, d], g[c], g[c, d], g[d]}
```

这里先对 $x__$ 尝试最短匹配.

```
In[8]:= ReplaceList[f[a, b, c, d],
         f[___, Shortest[x__], ___] -> g[x]]

Out[8]= {g[a], g[b], g[c], g[d], g[a, b], g[b, c],
         g[c, d], g[a, b, c], g[b, c, d], g[a, b, c, d]}
```

相关指南

- 规则与模式

相关教程

- 模式

教程专集

- **Core Language**

可选变量与默认变量

有时需要定义具有默认值的函数. 即省略某些变量时，其值就用设定的默认值代替. 模式 *x*_:*v* 就表示省略时值为 *v* 表示的变量.

这里定义了一个具有变量 x 和可选变量 y 和 z 的函数 j，其值分别用 1 和 2 代替.

```
In[1]:= j[x_, y_: 1, z_: 2] := jp[x, y, z]
```

使用 z 的默认值.

```
In[2]:= j[a, b]
Out[2]= jp[a, b, 2]
```

用 y 和 z 的默认值.

```
In[3]:= j[a]
Out[3]= jp[a, 1, 2]
```

$x_:v$	省略时值用 v 代替的表达式
$x_h:v$	具有头部 h 和默认值 v 的表达式
$x_.$	具有设定默认值的表达式

具有默认值的模式元素.

一些 ***Mathematica*** 常用函数的变量具有系统设定的默认值，此时不能明确给出 $x_:v$ 中的默认值，而是可用 $x_.$ 来使用其系统设定的默认值.

x_+y_.	y 的默认值为 0
x_ y_.	y 的默认值为 1
x_^y_.	y 的默认值为 1

一些具有可选变量的模式.

此处 a 与模式 x_ + y_. 匹配，其中 y 取系统的默认值 0.

```
In[4]:= {f[a], f[a + b]} /. f[x_ + y_.] -> p[x, y]
Out[4]= {p[a, 0], p[b, a]}
```

由于 Plus 是一个可结合的函数，模式 x_ + y_ 可以与任意项的和匹配. 然而，该模式不能与单项 a 匹配. 但由于模式x_ + y_. 中含有一个可选项，故它既可以与任意有限项如 x_ 与 y_ 的和，也可以

与单项 x_ 匹配，此时 y 为默认值 0 .

利用 x_. 可以使一个模式与几个不同的表达式匹配，当需要与多个结构不同但数学形式相同的表达式匹配时这种方式特别方便.

此模式与 g[a^2] 匹配，但与 g[a + b] 不匹配.

```
In[5]:= {g[a^2], g[a + b]} /. g[x_^n_] -> p[x, n]
Out[5]= {p[a, 2], g[a + b]}
```

用指数为可选项的模式可使这一模式与两种情况都匹配.

```
In[6]:= {g[a^2], g[a + b]} /. g[x_^n_.] -> p[x, n]
Out[6]= {p[a, 2], p[a + b, 1]}
```

模式 a_. + b_. x_ 与任何 x_ 的线性函数匹配.

```
In[7]:= lin[a_. + b_. x_, x_] := p[a, b]
```

此时，b → 1.

```
In[8]:= lin[1 + x, x]
Out[8]= p[1, 1]
```

这里 b → 1 并且 a → 0.

```
In[9]:= lin[y, y]
Out[9]= p[0, 1]
```

标准的 ***Mathematica*** 函数（如 Plus 和 Times ）有系统设定的默认值. 正如 "模式" 节中讨论的，也可以给已有的函数设定默认值.

有时不对一个可选变量分配默认值是方便的. 这样的变量可以使用

PatternSequence[] 来指定.

p|PatternSequence[]	不具有默认值的可选模式 p

不具有默认值的可选变量.

该模式与第二个可选变量 2 匹配，而没有默认值.

```
In[10]:= {g[1], g[1, 1], g[1, 2]} /.
          g[x_, 2 | PatternSequence[]] :> p[x]

Out[10]= {p[1], g[1, 1], p[1]}
```

相关指南

- 模式
- 规则与模式

相关教程

- 模式

教程专集

- **Core Language**

定义具有可选变量的函数

在定义复杂的函数时，常常需要用一些“可选”变量，当没有明确指出这些变量时，就需要用系统设定的“默认”值.

Mathematica 提供了两种定义具有可选变量函数的方法，在 ***Mathematica*** 中可以根据需要选用.

第一种方法就是利用变量的位置，忽略这些变量后就用默认值来代替. 几乎所有用这种方法定义的系统函数都可以忽略相应变量. 例如，函数 `Flatten[list, n]` 中的第二个变量忽略时用默认值 `Infinity` 来代替.

位置变量可以用模式 `_:` 来实现.

$f[x_, k_: kdef] := value$	第二个位置为可选变量，默认值为 *kdef* 的函数

定义一个“有位置变量”的函数.

定义第二个变量为可选的函数，当该函数被忽略时，其默认值为 `Infinity`.

```
In[1]:= f[list_, n_: Infinity] := f0[list, n]
```

具有两个可选变量的函数.

```
In[2]:= fx[list_, n1_: 1, n2_: 2] := fx0[list, n1, n2]
```

Mathematica 忽略变量时从最后面开始，故这里 m 是 n1 的值，2 是 n2 的默认值.

```
In[3]:= fx[k, m]
Out[3]= fx0[k, m, 2]
```

变量的名称，然后可用变量规则对其赋值，对像 Plot 等可选变量很多的函数，这种方法特别方便.

一个函数中可选变量的值可以通过将适当的变换规则放在变量的后面给出. 例如，可用 Joined -> True 来指定 ListPlot[*list*, Joined -> True] 中可选变量 Joined 的值.

当定义了具有可选变量的函数 f 后，习惯上是将这些变量的默认值以一些变量规则集合的形式用 Options[f] 保存起来.

---

f[x_, OptionsPattern[]] := *value*	具有零个或多个命名的可选变量的函数
OptionValue[*name*]	函数中命名可选变量值

命名的变量.

在函数 fn 中定义了 opt1 和 opt2 两个命名的可选变量.

```
In[4]:= Options[fn] = {opt1 -> 1, opt2 -> 2}
Out[4]= {opt1 → 1, opt2 → 2}
```

有零个或多个命名可选变量的函数 fn.

```
In[5]:= fn[x_, OptionsPattern[]] :=
          k[x, OptionValue[opt2]]
```

没有指定可选变量，因此用了 opt2 的默认规则.

```
In[6]:= fn[4]
Out[6]= k[4, 2]
```

当明确给出了 opt2 的规则时，它将先于 Options[fn] 中的默认规则而使用.

```
In[7]:= fn[4, opt2 -> 7]
Out[7]= k[4, 7]
```

FilterRules[*opts*, Options[*name*]]	*opts* 中的规则被用作函数 *name* 的选项
FilterRules[*opts*, Except[Options[*name*]]]	*opts* 中的规则不能用作函数 *name* 的选项

过滤器选项.

有时，当写函数的时候，可能需要将选项传给它所调用的其它函数.

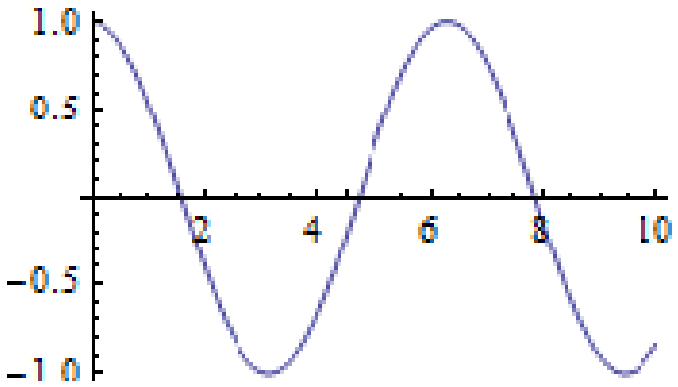
下面是一个简单的函数，求解数值差分方程，并且把结果画出来.

```
In[8]:= odeplot[de_, y_, {x_, x0_, x1_},
          opts : OptionsPattern[]] :=
         Module[{sol},
          sol = NDSolve[de, y, {x, x0, x1},
              FilterRules[{opts}, Options[NDSolve]]]
           If[Head[sol] === NDSolve,
            $Failed,
            Plot[Evaluate[y /. sol],
             {x, x0, x1}, Evaluate[
              FilterRules[{opts}, Options[Plot]]]]
           ]
         ]
```

没有给定选项时，使用 NDSolve 和 Plot 的默认选项.

```
In[9]:= odeplot[{y''[x] + y[x] == 0,
          y[0] == 1, y'[0] == 0}, y[x], {x, 0, 10}]
```

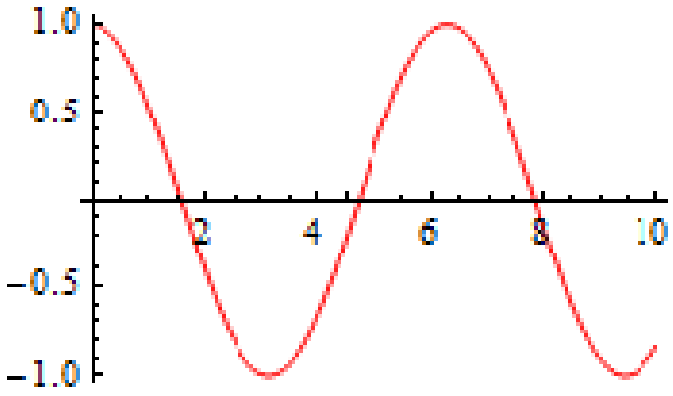
Out[9]=



这里改变了 NDSolve 所用的方法和图中的颜色.

```
In[10]:= odeplot[{y''[x] + y[x] == 0,
    y[0] == 1, y'[0] == 0}, y[x],
   {x, 0, 10}, Method → "ExplicitRungeKutta",
   PlotStyle → Red]
```

Out[10]=



相关教程

- 模式

教程专集

- **Core Language**

重复模式

expr..	重复一次或多次的模式或表达式
expr...	重复零次或多次的模式或表达式

重复模式.

Mathematica 模式重复运算 `..` 和 `...` 可以给出某些形式重复任意次的模式. 例如，`f[a..]` 表示任意形如 `f[a]`，`f[a, a]`，`f[a, a, a]` 的表达式.

模式 `f[a..]` 允许变量 a 重复多次.

```
In[1]:= Cases[{f[a], f[a, b, a], f[a, a, a]}, f[a..]]
Out[1]= {f[a], f[a, a, a]}
```

这一模式允许任何数目的变量 a 后跟任何数目的变量 b.

```
In[2]:= Cases[{f[a], f[a, a, b], f[a, b, a], f[a, b, b]}
         f[a.., b..]]
Out[2]= {f[a, a, b], f[a, b, b]}
```

每一个变量既可以是 a 又可以是 b.

```
In[3]:= Cases[{f[a], f[a, b, a], f[a, c, a]},
         f[(a | b)..]]
Out[3]= {f[a], f[a, b, a]}
```

也可以用 `..` 和 `...` 表示任意模式的重复，当该模式中包含有命名的项时，则这些项在任一次重复中是相同的.

有一列变量对的函数.

```
In[4]:= v[x : {{_, _}..}] := Transpose[x]
```

应用上面的定义.

```
In[5]:= v[{{a1, b1}, {a2, b2}, {a3, b3}}]
Out[5]= {{a1, a2, a3}, {b1, b2, b3}}
```

在这一定义中，所有对的第二个元素必须相等.

```
In[6]:= vn[x : {{_, n_} ..}] := Transpose[x]
```

应用上面的定义.

```
In[7]:= vn[{{a, 2}, {b, 2}, {c, 2}}]
Out[7]= {{a, b, c}, {2, 2, 2}}
```

模式 x .. 可以被扩展为两个变量，以便更准确地控制重复次数.

p .. 或 Repeated[p]	重复一次或多次的模式或表达式
Repeated[p, max]	重复至多 max 次的模式
Repeated[p, {min, max}]	重复次数在 min 和 max 之间的模式
Repeated[p, {n}]	刚好重复 n 次的模式

控制重复的次数.

这里寻找变量 a 的两到三次重复.

```
In[8]:= Cases[{f[a], f[a, a], f[a, a, a], f[a, a, a, a]}
         f[Repeated[a, {2, 3}]]]
Out[8]= {f[a, a], f[a, a, a]}
```

相关教程

- 模式

教程专集

逐字模式

`Verbatim[`*expr*`]`	必须逐字匹配的表达式

逐字模式.

x_ 与任何表达式匹配.

```
In[1]:= {f[2], f[a], f[x_], f[y_]} /. f[x_] -> x^2
```

Out[1]= $\{4, a^2, x_^2, y_^2\}$

`Verbatim` 要求 ***Mathematica*** 仅完全一致的表达式 x_ 才匹配.

```
In[2]:= {f[2], f[a], f[x_], f[y_]} /.
         f[Verbatim[x_]] -> x^2
```

Out[2]= $\{f[2], f[a], x^2, f[y_]\}$

相关指南

- 模式
- 规则与模式
- 表达式测试

相关教程

- 引言

- 模式
- 模式和变换规则
- 模式序列
- 逐字模式
- 可选变量与默认变量

教程专集

- **Core Language**

常用表达式的模式

在“模式引言”中介绍了建立许多类表达式的模式的方法，在所有情况下，应该记住模式表示以 `FullForm` 给出的表达式的结构.

对一些常用表达式，***Mathematica*** 的标准输出格式与内部的完全形式不一致，但在模式中使用的是内部形式.

n_Integer	整数 n
x_Real	实数 x
z_Complex	复数 z
Complex[x_, y_]	复数 $x + i\,y$
Complex[x_Integer, y_Integer]	实部和虚部均为整数的复数
(r_Rational | r_Integer)	有理数或整数 r
Rational[n_, d_]	有理数 $\frac{n}{d}$
(x_ /; NumberQ[x] && Im[x] == 0)	任何形式的实数
(x_ /; NumberQ[x])	任何数

数的一些典型模式.

一些数的完全形式.

```
In[1]:= {2, 2.5, 2.5 + I, 2 / 7} // FullForm
```

```
t[1]//FullForm=
List[2, 2.5`,
 Complex[2.5`, 1.`], Rational[2, 7]]
```

取出复数的每一部分.

```
In[2]:= {2.5 - I, 3 + I} /. Complex[x_, y_] -> p[x, y]
Out[2]= {p[2.5, -1.], p[3, 1]}
```

这些表达式有不同的完全形式表明不能用 $x_ + \mathrm{I}\ y_$ 与复数匹配.

```
In[3]:= {2.5 - I, x + I y} // FullForm
```

```
t[3]//FullForm=
List[Complex[2.5`, -1.`],
 Plus[x, Times[Complex[0, 1], y]]]
```

该模式与实整数和实、虚部均为整数的复数匹配.

```
In[4]:= Cases[{2.5 - I, 2, 3 + I, 2 - 0.5 I, 2 + 2 I},
          _Integer | Complex[_Integer, _Integer]]
```

```
Out[4]= {2, 3 + ⅈ, 2 + 2 ⅈ}
```

在 "符号计算" 中讨论过，***Mathematica*** 将所有的代数表达式表示为标准形式，标准形式的基础是幂乘积的和. 在商中分母转化为负指数，差被转化为负项的和. 定义代数表达式的模式中必须用标准形式，标准形式往往与 ***Mathematica*** 的输出形式不一致. 但在任何情况下，，可以用 FullForm[*expr*] 去得到表达式的内部形式.

这是一个典型的代数表达式.

```
In[5]:= -1 / z^2 - z / y + 2 (x z)^2 y
```

$$\mathrm{Out[5]=}\quad -\frac{1}{z^2} - \frac{z}{y} + 2\,x^2\,y\,z^2$$

这一表达式完整的内部形式.

```
In[6]:= FullForm[%]
```

```
t[6]//FullForm=
Plus[Times[-1, Power[z, -2]],
 Times[-1, Power[y, -1], z],
 Times[2, Power[x, 2], y, Power[z, 2]]]
```

将变换规则用到这一表达式的幂上得到的结果.

```
In[7]:= % /. x_^n_ -> e[x, n]
Out[7]= -z e[y, -1] - e[z, -2] + 2 y e[x, 2] e[z, 2]
```

$x_+y_$	两项或多项的和
$x_+y_.$	单项或多项的和
n_Integer x_	有一个整数因子的表达式
$a_.+b_.\ x_$	线性表达式 $a + b\,x$
$x_\text{\^{}}n_$	x^n 其中 $n \neq 0, 1$
$x_\text{\^{}}n_.$	x^n 其中 $n \neq 0$
$a_.+b_.\ x_+$ $c_.\ x_\text{\^{}}2$	线性项系数非零的二次表达式

代数表达式的一些典型模式.

取出关于 x 的线性函数.

```
In[8]:= {1, a, x, 2 x, 1 + 2 x} /. a_. + b_. x -> p[a, b]
Out[8]= {1, a, p[0, 1], p[0, 2], p[1, 2]}
```

x_List 或 x:{___}	一个列表
x_List/;VectorQ[x]	没有子列的向量
x_List/;VectorQ[x, NumberQ]	数值向量
x:{___List} 或 x:{{___}...}	集合的列表
x_List/;MatrixQ[x]	没有子列的矩阵
x_List/;MatrixQ[x, NumberQ]	常数矩阵
x:{{_,_}...}	元素对的集合

关于列表的模式.

这里定义一个函数变量为含有一个或两个元素集合的列表.

```
In[9]:= h[x : {({_} | {_, _}) ...}] := q
```

这一定义用于第二和第三种情形.

```
In[10]:= {h[{a, b}], h[{{a}, {b}}], h[{{a}, {b, c}}]}
Out[10]= {h[{a, b}], q, q}
```

相关教程

- 模式

教程专集

- **Core Language**

举例：定义积分函数

在介绍了 ***Mathematica*** 模式的基本功能以后，我们给出一些例子. 下面说明如何在 ***Mathematica*** 中定义积分函数.

从数学的观点看，积分函数就是由一系列数学的关系决定的，在 ***Mathematica*** 中通过对模式建立一些变换就可以来实现这些关系.

数学形式	***Mathematica*** *的定义*
$\int (y+z)\,dx = \int y\,dx + \int z\,dx$	`integrate[y_+z_,x_]:= integrate[y,x]+integrate[z,x]`
$\int cy\,dx = c\int y\,dx$ (c 独立于 x)	`integrate[c_ y_,x_]:= c integrate[y,x]/;FreeQ[c,x]`
$\int c\,dx = cx$	`integrate[c_,x_]:=cx/;FreeQ[c,x]`
$\int x^n\,dx = \frac{x^{(n+1)}}{n+1},\ n\neq -1$	`integrate[x_^n_.,x_]:= x^(n+1)/(n+1)/; FreeQ[n,x]&&n!=-1`
$\int \frac{1}{ax+b}\,dx = \frac{\log(ax+b)}{a}$	`integrate[1/(a_.x_+b_.),x_]:= Log[ax+b]/a/; FreeQ[{a,b},x]`
$\int e^{ax+b}\,dx = \frac{1}{a}e^{ax+b}$	`integrate[Exp[a_.x_+b_.],x_]:= Exp[ax+b]/a/; FreeQ[{a,b},x]`

定义积分函数.

这里实现了积分的线性性质：$\int (y+z)\,dx = \int y\,dx + \int z\,dx$.

```
In[1]:= integrate[y_ + z_, x_] :=
          integrate[y, x] + integrate[z, x]
```

Plus 的结合性使线性关系对任意项的和成立.

```
In[2]:= integrate[a x + b x^2 + 3, x]

Out[2]= integrate[3, x] +
         integrate[a x, x] + integrate[b x^2, x]
```

利用 integrate 将与积分变量 x 无关的因子提到积分外去.

```
In[3]:= integrate[c_ y_, x_] :=
         c integrate[y, x] /; FreeQ[c, x]
```

Mathematica 检查乘积的每一项是否满足 FreeQ 条件，从而将其提到积分外.

```
In[4]:= integrate[a x + b x^2 + 3, x]

Out[4]= integrate[3, x] +
         a integrate[x, x] + b integrate[x^2, x]
```

此处给出常数的积分：$\int c \, dx = c\, x$.

```
In[5]:= integrate[c_, x_] := c x /; FreeQ[c, x]
```

于是和式中的常数项就可以积分.

```
In[6]:= integrate[a x + b x^2 + 3, x]

Out[6]= 3 x + a integrate[x, x] + b integrate[x^2, x]
```

这里给出了 x^n 的标准积分公式. 通过使用模式 x_^n_.，而不是 x_^n_，我们包含了 $x^1 = x$ 的情形.

```
In[7]:= integrate[x_^n_., x_] :=
         x^(n + 1) / (n + 1) /; FreeQ[n, x] && n != -1
```

这一积分完全可以做出来.

```
In[8]:= integrate[a x + b x^2 + 3, x]
```

$$\text{Out[8]}= 3\,x + \frac{a\,x^2}{2} + \frac{b\,x^3}{3}$$

内部函数 Integrate （开头为大写的 I） 肯定能做这个积分.

```
In[9]:= Integrate[a x + b x^2 + 3, x]
```

$$\text{Out[9]}= 3\,x + \frac{a\,x^2}{2} + \frac{b\,x^3}{3}$$

对线性函数的倒数进行积分的规则. 模式 a_. x_ + b_. 表示 x 的任何线性函数.

```
In[10]:= integrate[1 / (a_. x_ + b_.), x_] :=
           Log[a x + b] / a /; FreeQ[{a, b}, x]
```

这里 a 和 b 取其缺省值.

```
In[11]:= integrate[1 / x, x]
```

$$\text{Out[11]}= \text{Log}[x]$$

这是较复杂的一个例子. 其中，a 与 2 p 匹配.

```
In[12]:= integrate[1 / (2 p x - 1), x]
```

$$\text{Out[12]}= \frac{\text{Log}[-1 + 2\,p\,x]}{2\,p}$$

可以添加更多的积分规则，这里给出了指数函数的积分.

```
In[13]:= integrate[Exp[a_. x_ + b_.], x_] :=
           Exp[a x + b] / a /; FreeQ[{a, b}, x]
```

相关教程

- 模式

教程专集

- **Core Language**

变换规则和定义

运用变换规则
一组变换规则的操作
定义
特殊形式的赋值
定义带标号的对象
定义函数
定义的顺序
立即定义和延时定义
保留已有值的函数
与不同符号相关的定义
定义数量值
修改内部函数

- **Core Language**

运用变换规则

expr /. *lhs*->*rhs*	对 *expr* 运用变换规则
expr /. {*lhs*$_1$->*rhs*$_1$, *lhs*$_2$->*rhs*$_2$, ...}	将一列变换规则用于 *expr* 的每一项

运用变换规则.

在表达式上，替换运算符 /. 的使用.

```
In[1]:= x + y /. x -> 3
Out[1]= 3 + y
```

可给出一列变换规则，将每个规则分别用到表达式的每一项.

```
In[2]:= x + y /. {x -> a, y -> b}
Out[2]= a + b
```

expr /. {*rules*$_1$, *rules*$_2$, ...}	将规则 *rules*$_i$ 中的每个用于表达式 *expr*

运用一列变换规则.

通过一列规则就可以得到一列结果.

```
In[3]:= x + y /. {{x -> 1, y -> 2}, {x -> 4, y -> 2}}
Out[3]= {3, 6}
```

Solve 和 NSolve 等函数的返回值是一列规则，每个规则代表一个解.

```
In[4]:= Solve[x^3 - 5 x^2 + 2 x + 8 == 0, x]
Out[4]= {{x → -1}, {x → 2}, {x → 4}}
```

运用这些规则可以得到一列结果，每个结果都对应一个解.

```
In[5]:= x^2 + 6 /. %
Out[5]= {7, 10, 22}
```

expr /. *rules* 将每一个规则逐次用到 *expr* 的每一项. 在此过程中进行相应的变换以得到结果.

先用规则 x^3，当它无法运用时用规则 x^n_.

```
In[6]:= {x^2, x^3, x^4} /. {x^3 -> u, x^n_ -> p[n]}
Out[6]= {p[2], u, p[4]}
```

规则一旦使用就立即产生结果，故里面的 h 还没有变.

```
In[7]:= h[x + h[y]] /. h[u_] -> u^2
Out[7]= (x + h[y])^2
```

替换 *expr* /. *rules* 中的每个法则对 *expr* 中的每一项只使用一次.

由于每个规则正好用一次，所以这里是交换 x 和 y.

```
In[8]:= {x^2, y^3} /. {x -> y, y -> x}
```

$$\text{Out[8]=}\ \{y^2,\ x^3\}$$

可以用这种形式一个接一个地运用一组规则.

```
In[9]:= x^2 /. x -> (1 + y) /. y -> b
```

$$\text{Out[9]=}\ (1+b)^2$$

有时需要反复使用规则，直到表达式不再变换为止，这可通过反复替换运算 *expr* `//.` *rules* （或 `ReplaceRepeated[`*expr*, *rules*`]`）来实现.

expr `/.` *rules*	在 *expr* 的每一项中用变换规则一次
expr `//.` *rules*	重复使用规则直到结果不再变化为止

一次或多次重复使用规则.

单一替代运算 `/.` 使规则在表达式中各项上仅用一次.

```
In[10]:= x^2 + y^6 /. {x -> 2 + a, a -> 3}
```

$$\text{Out[10]=}\ (2+a)^2 + y^6$$

重复替代运算 `//.` 使得规则被反复使用直到表达式不再变化为止.

```
In[11]:= x^2 + y^6 //. {x -> 2 + a, a -> 3}
```

$$\text{Out[11]=}\ 25 + y^6$$

此处规则仅使用一次.

```
In[12]:= log[a b c d] /. log[x_ y_] -> log[x] + log[y]
Out[12]= log[a] + log[b c d]
```

用重复替换运算反复使用规则直到结果不再变化为止.

```
In[13]:= log[a b c d] //. log[x_ y_] -> log[x] + log[y]
Out[13]= log[a] + log[b] + log[c] + log[d]
```

用 //. 时，***Mathematica*** 将给定的规则反复使用到表达式上，直到相继的两个结果相同为止.

当使用一系列循环规则时，//. 能一直得到不同的结果. 在实际操作中，对一个特定表达式 //. 所能进行的最大循环次数取决于选项的设置. 用可选项 MaxIterations 可以确定对一个表达式的迭代次数. 当需要一直替换下去时，可以用 ReplaceRepeated[*expr*, *rules*, MaxIterations -> Infinity] 来实现. 通过中断 ***Mathematica*** 总可以停止迭代.

通过可选项 MaxIterations，可以明确指出 ReplaceRepeated 在运用给定规则时的重复次数.

```
In[14]:= ReplaceRepeated[x,
          x -> x + 1, MaxIterations -> 1000]

         ReplaceRepeated::rrlim :
          Exiting after x scanned 1000 times. >>

Out[14]= 1000 + x
```

替换运算 /. 和 //. 的特点是将每个规则用在表达式的各项上，而 Replace[*expr*, *rules*] 将规则用在整个表达式 *expr* 上，不能用

于表达式的一部分.

可以把 Replace 和 Map 以及 MapAt 等共同来指定规则用在表达式的哪一部分上. 另外， ReplacePart[*expr*, *new*, *pos*] 可以用给定的目标来代替表达式的项.

操作符 /. 将规则运用到表达式的每一项.

```
In[15]:= x^2 /. x -> a
```

$$\text{Out[15]= } a^2$$

没有指定层时，Replace 仅将规则用于整个表达式.

```
In[16]:= Replace[x^2, x^2 -> b]
```

$$\text{Out[16]= } b$$

这里没有进行任何替换.

```
In[17]:= Replace[x^2, x -> a]
```

$$\text{Out[17]= } x^2$$

将规则运用到第2层从而替换了 x.

```
In[18]:= Replace[x^2, x -> a, 2]
```

$$\text{Out[18]= } a^2$$

expr /. *rules*	将规则运用于表达式 *expr* 的子项
Replace[*expr*, *rules*]	仅将规则运用于整个表达式 *expr*
Replace[*expr*, *rules*, *levspec*]	将规则用到由 *levspec* 指定层表达式 *expr* 的项上

对整个表达式运用规则.

Replace 给出使用第一个规则后的结果.

```
In[19]:= Replace[f[u], {f[x_] -> x^2, f[x_] -> x^3}]
```

$$\text{Out[19]= } u^2$$

ReplaceList 给出一列规则使用后的结果.

```
In[20]:= ReplaceList[f[u], {f[x_] -> x^2, f[x_] -> x^3}]
```

$$\text{Out[20]= } \{u^2, u^3\}$$

当一个规则有多种使用方式时，ReplaceList 给出所有的结果.

```
In[21]:= ReplaceList[a + b + c, x_ + y_ -> g[x, y]]
Out[21]= {g[a, b + c], g[b, a + c], g[c, a + b],
          g[a + b, c], g[a + c, b], g[b + c, a]}
```

此处给出了将原集合分解为两个集合的方式.

```
In[22]:= ReplaceList[{a, b, c, d},
           {x__, y__} -> g[{x}, {y}]]
Out[22]= {g[{a}, {b, c, d}],
          g[{a, b}, {c, d}], g[{a, b, c}, {d}]}
```

找出由同样元素包在侧面的所有子集.

```
In[23]:= ReplaceList[{a, b, c, a, d, b, d},
           {___, x_, y__, x_, ___} -> g[x, {y}]]
Out[23]= {g[a, {b, c}], g[b, {c, a, d}], g[d, {b}]}
```

Replace[*expr*, *rules*]	仅以一种方式用 *rules*
ReplaceList[*expr*, *rules*]	以所有可能的方式使用 *rules*

用一种方式或所有可能方式运用规则.

相关教程

- 变换规则和定义

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

一组变换规则的操作

对一组规则进行操作.

对三角函数的展开式规则命名为 sinexp.

```
In[1]:= sinexp = Sin[2 x_] -> 2 Sin[x] Cos[x]
Out[1]= Sin[2 x_] → 2 Cos[x] Sin[x]
```

用名称 sinexp 来调用规则.

```
In[2]:= Sin[2 (1 + x)^2] /. sinexp
Out[2]= 2 Cos[(1 + x)^2] Sin[(1 + x)^2]
```

一组变换规则可以用来描述数学或其它关系，通过命名就可以方便地调用一组规则.

大多数情况下，一组规则中仅有一个作用于一个表达式. 然而，/. 操作符依次测试表中的所有规则. 当表中规则很多时，这需要很长时间.

Mathematica 为了提高 /. 的运行速度，可以先对一组规则进行处理，其方法是让 Dispatch 作用在这组规则上. Dispatch 函数作用的结果还是这一组规则，但它产生了一个分派表. 该分派表使 /. 不再逐个测试每个规则，而立即转到可使用的规则上去.

此处给出了前**5**个阶段的规则.

```
In[3]:= facs = Table[f[i] -> i!, {i, 5}]
Out[3]= {f[1] → 1, f[2] → 2,
         f[3] → 6, f[4] → 24, f[5] → 120}
```

建立分派表使规则运用得更快.

```
In[4]:= dfacs = Dispatch[facs]

Out[4]= Dispatch[{f[1] → 1, f[2] → 2, f[3] → 6,
    f[4] → 24, f[5] → 120}, -DispatchTables-]
```

通过 /. 操作符来运用这些规则.

```
In[5]:= f[4] /. dfacs

Out[5]= 24
```

Dispatch[*rules*]	产生一列具有分派表的规则
expr /. *drules*	产生有分派表的规则

产生和使用分派表.

对于较长的规则列表，用了分派表之后可以使一列规则的替换运行得很快，当这些规则不是模式，而是一些单个符号或表达式时更显示出优势，用了分派表时就会发现 /. 操作符所花的时间几乎与规则的数量无关，而没有分派表时，/. 所花的时间与规则的数量成比例.

相关教程

- 变换规则和定义

教程专集

- **Core Language**

定义

替换运算 /. 将规则作用于一个表达式. 但经常需要在可能的情况下自动使用变换规则.

这可以通过对 ***Mathematica*** 表达式和模式赋值来实现. 赋值表明适当形式的表达式出现时就使用规则.

expr /. *lhs*->*rhs*	将规则用于一个表达式
lhs=*rhs*	赋值使规则可能时立即使用

人工和自动应用变换规则.

对指定的表达式使用关于 x 的变换规则.

```
In[1]:= (1 + x)^6 /. x -> 3 - a
```

$$\text{Out[1]}= (4 - a)^6$$

通过对 x 赋值，可以告诉 ***Mathematica*** 对后面的任何 x 使用变换规则.

```
In[2]:= x = 3 - a
```

$$\text{Out[2]}= 3 - a$$

现在 x 自动地进行变换.

```
In[3]:= (1 + x)^7
```

$$\text{Out[3]}= (4 - a)^7$$

除 `Module` 和 `Block` 等一些内部结构外，在 ***Mathematica*** 中的

同一个进程中所赋值保持不变.

赋值的永久性意味着使用时要特别慎重. 一个在使用 ***Mathematica*** 时，常犯的错误是在后面使用 x 时忘记或误用了前面 x 的赋值.

为了减少这一错误，可能时尽量避免赋值而用替换运算 /. 等，也可以在任务完成后立即用 =. 或函数 Clear 去清除所赋的值.

另外一种避免这一错误的途径是对常用或简单的变量名赋值时要仔细考虑. 例如，经常使用变量名 x 作为符号参数. 但是如果用 x = 3 赋值后，以后出现的 x 都用 3 代替，且以后也再不能将 x 当作一个符号参数使用.

一般来说，不要对有几种用途的变量赋值. 例如，若在某处用 c 变量表示光速 3. * 10^8. 则以后就不能将 c 用作一个待定参数. 避免这种情况的一种途径是对光速用更加明确的变量名，如 SpeedOfLight.

---

x=.	清除对 x 的赋值
Clear[x, y, ...]	清除变量 x，y，... 的所有值

清除赋值.

当 x 在前面赋过值时，就不一定能给出所期望的结果.

```
In[4]:= Factor[x^2 - 1]
Out[4]= (-4 + a) (-2 + a)
```

清除 x 以前的值.

```
In[5]:= Clear[x]
```

现在给出了正确的结果.

```
In[6]:= Factor[x^2 - 1]
Out[6]= (-1 + x) (1 + x)
```

相关教程

- 变换规则和定义

教程专集

- **Core Language**

特殊形式的赋值

在 ***Mathematica*** 编程中，经常需要用 *x = value* 等语句不断改变一些变量的值. ***Mathematica*** 在一些常用情况提供了通过增量修改变量的方法.

$i++$	i 加 1
$i--$	i 减 **1**
$++i$	先给 i 加 **1**
$--i$	先给 i 减 **1**
$i+=di$	i 加 di
$i-=di$	i 减 di
$x*=c$	x 乘以 c
$x/=c$	x 除以 c

修改变量的值.

变量 t 赋值为 7 x.

```
In[1]:= t = 7 x
Out[1]= 7 x
```

t 的值增加 18 x.

```
In[2]:= t += 18 x
Out[2]= 25 x
```

观察 t 值的变化.

```
In[3]:= t
Out[3]= 25 x
```

先将 t 的值设为 8，再乘以 7，给出最后 t 的结果.

```
In[4]:= t = 8; t *= 7; t
Out[4]= 56
```

i++ 的值是 i 增加以前的值.

```
In[5]:= i = 5; Print[i++]; Print[i]
        5
        6
```

++i 是 i 增加以后的值.

```
In[6]:= i = 5; Print[++i]; Print[i]
        6
        6
```

$x=y=value$	对 x 和 y 赋同一值
$\{x,y\}=\{value_1,value_2\}$	对 x 和 y 赋不同的值
$\{x,y\}=\{y,x\}$	交换 x 和 y 的值

同时对几个变量赋值.

给 x 赋值 5，y 赋值 8.

```
In[7]:= {x, y} = {5, 8}
Out[7]= {5, 8}
```

交换 x 和 y 的值.

```
In[8]:= {x, y} = {y, x}
Out[8]= {8, 5}
```

现在 x 的值为 8.

```
In[9]:= x
Out[9]= 8
```

y 的值为 5.

```
In[10]:= y
Out[10]= 5
```

可以用赋值语句任意交换变量的值.

```
In[11]:= {a, b, c} = {1, 2, 3};
         {b, a, c} = {a, c, b}; {a, b, c}
Out[11]= {3, 1, 2}
```

在 ***Mathematica*** 编程时，通过逐步增加元素的方法来构造一个集合是十分方便的，这可以用函数 PrependTo 和 AppendTo 来实现.

PrependTo[*v*, *elem*]	*v* 前加元素 *elem*
AppendTo[*v*, *elem*]	*v* 后加元素 *elem*
v={*v*, *elem*}	构造一个包含 *elem* 的嵌套列表

修改列表.

定义 v 的值为集合 {5, 7, 9}.

```
In[12]:= v = {5, 7, 9}
Out[12]= {5, 7, 9}
```

将 11 加到 v 中.

```
In[13]:= AppendTo[v, 11]
Out[13]= {5, 7, 9, 11}
```

v 的值被改变.

```
In[14]:= v
Out[14]= {5, 7, 9, 11}
```

AppendTo[*v*, *elem*] 是与 *v* = Append[*v*, *elem*] 等价的. 由于 ***Mathematica*** 中集合的存储方式，建立像每层长度为**2**的集合所组成的嵌套结构比增添一列元素更有效，当建立了这种嵌套结构后就可以用 Flatten 将其简化为一维列表.

建立一个嵌套结构 w.

```
In[15]:= w = {1}; Do[w = {w, k^2}, {k, 1, 4}]; w
Out[15]= {{{{1}, 1}, 4}, 9}, 16}
```

用 Flatten 取消这种结构.

```
In[16]:= Flatten[w]
Out[16]= {1, 1, 4, 9, 16}
```

相关教程

- 变换规则和定义

教程专集

- **Core Language**

定义带标号的对象

在多种计算中，需要建立包含一列带标号的表达式的阵列. 在 ***Mathematica*** 中得到阵列的一种途径是定义列表. 你可以定义一个列表，如 $a = \{x, y, z, \ldots\}$，然后就可以用 $a[[i]]$ 来调用它的元素或用 $a[[i]] = value$ 来修改它. 这一方式的缺陷是必须给出它的所有元素.

定义阵列的一个简便方法是在需要时给出它的一些元素，这可以通过定义表达式如 $a[i]$ 来实现.

定义 a[1] 的值.

```
In[1]:= a[1] = 9
Out[1]= 9
```

定义 a[2] 的值.

```
In[2]:= a[2] = 7
Out[2]= 7
```

显示到此为止所定义的 a 值.

```
Global`a
```

```
a[1] = 9
```

```
a[2] = 7
```

即使没有给出 a[3] 和 a[4] 的值，仍然可定义 a[5] 的值.

```
In[4]:= a[5] = 0
Out[4]= 0
```

产生 a[*i*] 的列表.

```
In[5]:= Table[a[i], {i, 5}]
Out[5]= {9, 7, a[3], a[4], 0}
```

可以把表达式 a[*i*] 当作一个“带索引”或“带下标”的变量.

$a[i]=value$	增加变量或改动变量的值
$a[i]$	调用变量
$a[i]=.$	删除变量
$?a$	显示定义过的值
`Clear[a]`	清除定义过的值
`Table[a[i],{i,1,n}]` 或 `Array[a,n]`	转换为一个列表 `List`

操作带标号的变量.

在表达式 $a[i]$ 中，i 并不一定要是整数. 事实上，***Mathematica*** 允许它是任意表达式. 通过使用符号型的标号，可以在 ***Mathematica*** 中构造一些简单的数据库等

具有标号 square 的对象的面积 area 为 1.

```
In[6]:= area[square] = 1
Out[6]= 1
```

在面积 area 库中添加另一个元素.

```
In[7]:= area[triangle] = 1 / 2
Out[7]= 1/2
```

显示到此为止的面积库 area 的元素.

```
In[8]:= ?area
```

```
Global`area

area[square] = 1

area[triangle] = 1/2
```

在任何需要的地方都可以使用这种定义. 此处还没有定义 area[pentagon].

```
In[9]:= 4 area[square] + area[pentagon]
Out[9]= 4 + area[pentagon]
```

相关教程

- 变换规则和定义

教程专集

定义函数

"自定义函数" 节中讨论了在 ***Mathematica*** 中函数的定义. 在典型情况下，可以用 f[x_] = x^2 定义一个函数 f. （事实上，在 "自定义函数" 节中用 := 而不是 = 来定义. 在 "立即定义和延时定义" 节中将解释何时使用 := 和 = ）.

定义 f[x_] = x^2 表明当 ***Mathematica*** 遇到与模式 f[x_] 匹配的表达式时，它将用 x^2 来代替该表达式. 由于模式f[x_] 与所有形如 f[*anything*] 的表达式匹配，这一定义可用于具有任何变量的函数 f.

函数定义如 f[x_] = x^2 可以与 "定义带标号的对象" 节讨论的有标号变量 f[a] = b 进行比较. 定义 f[a] = b 表明当*特定* 的表达式 f[a] 出现时用 b 代替. 但这定义对 f[y] 等表达式不起作用，因为这时 f 有另外的标号.

为了定义一个函数，必须给可以是任何值的变量 *x* 的表达式 f[*x*] 指定对应值，这可以通过定义模式 f[x_] 来实现，其中模式对象 x_ 代表任何表达式.

f[*x*] =*value*	对指定表达式 *x* 的定义
f[*x* _] =*value*	对任何涉及到 *x* 的表达式的定义

定有标号的变量和定义函数的区别.

定义 f[2] 或 f[a] 可以看作对阵列 f 的元素赋值. 定义一个函数

上，可以把函数看作具有任意变动标号的元素的阵列.

从数学的观点看，f 是一个*映射*. 当定义 f[1] 和 f[2] 等时，就是给出其定义域中离散点的象，而 f[x_] 是给出 f 在一连续流点集上的像.

对*确定* 的表达式 f[x] 定义变换规则.

```
In[1]:= f[x] = u
```

```
Out[1]= u
```

当这一确定的表达式 f[x] 出现时，用 u 代替它. 其它表达式 f[*argument*] 则不变.

```
In[2]:= f[x] + f[y]
```

```
Out[2]= u + f[y]
```

对任意变量定义 f 的值.

```
In[3]:= f[x_] = x^2
```

```
Out[3]= x^2
```

最初特定形式的 f[x] 的定义仍然有效，一般定义 f[x_] 用来求出 f[y] 的值.

```
In[4]:= f[x] + f[y]
```

```
Out[4]= u + y^2
```

清除 f 的所有定义.

```
In[5]:= Clear[f]
```

f[1] 或 f[a] 等这些具体表达式的定义与像 f[x_] 等的定义相结合.

许多数学函数可以通过将特定和一般定义相结合的方式给出. 例如，阶乘函数，在 ***Mathematica*** 中已经给出了这一函数*n*!，但可以用 ***Mathematica*** 的定义自行建立这个函数.

阶乘函数的数学定义几乎可以直接在 ***Mathematica*** 中使用，其形式为 f[n_] := n f[n - 1]; f[1] = 1. 此定义表明对 *n* 等于 1 的情况，f[1] 等于 1；对其余 *n*，f[*n*] 等于 *n* f[*n* - 1].

变量为 1 时阶乘函数的值.

```
In[6]:= f[1] = 1
Out[6]= 1
```

阶乘函数一般的递推过程.

```
In[7]:= f[n_] := n f[n - 1]
```

现在就可以用这个定义区得到阶乘的值乘的值.

```
In[8]:= f[10]
Out[8]= 3 628 800
```

此结果与内部函数 *n*! 的值相同.

```
In[9]:= 10!
Out[9]= 3 628 800
```

相关教程

- 变换规则和定义

定义的顺序

在 ***Mathematica*** 中给出一系列定义时，一部分总是比另一部分更一般一些. ***Mathematica*** 中的原则是：一般定义在特殊定义之后使用. 这意味着，规则的特例总是在一般情况之前试用.

这种行为对于 "定义函数" 节中给出的阶乘函数特别重要. 不管输入方式如何，***Mathematica*** 总是把特殊规则 `f[1]` 放在 `f[n_]` 之前. 这意味着，当 ***Mathematica*** 在计算形如 `f[n]` 的表达式时，先测试 `f[1]`，当它不能用时，再使用一般情况 `f[n_]`. 所以，在计算 `f[5]` 时，***Mathematica*** 将一直使用一般规则直到“终止条件” `f[1]` 使用了为止.

- ***Mathematica*** 总是将特殊规则放在一般规则之前.

Mathematica 中定义的处理.

如果 ***Mathematica*** 不遵循上述原则，则特殊规则将会被一般规则所“屏蔽”. 在阶乘的定义中，如果规则 `f[n_]` 在规则`f[1]` 之前使用，则 ***Mathematica*** 即使在求 `f[1]` 时也会用 `f[n_]` 规则，而 `f[1]` 规则永远不会被使用.

`f[n_]` 的一般定义.

```
In[1]:= f[n_] := n f[n - 1]
```

特殊情况 f[1] 的定义.

```
In[2]:= f[1] = 1
Out[2]= 1
```

Mathematica 将特殊情形放在一般情形之前.

```
In[3]:= ? f
```

```
Global`f

f[1] = 1

f[n_] := n f[n - 1]
```

在像上面给出的阶乘一类的定义中，可以很明显地看出哪一个更一般一些. 但通常给出的规则没有确定的顺序，这时， ***Mathematica*** 就按给出的顺序使用它们.

这些规则没有明确的顺序.

```
In[4]:= log[x_ y_] := log[x] + log[y];
        log[x_^n_] := n log[x]
```

Mathematica 按顺序保存这些规则.

```
In[5]:= ? log
```

```
Global`log

log[x_ y_] := log[x] + log[y]

log[x_^n_] := n log[x]
```

log[x_ y_] 的一个特例.

Mathematica 将特殊规则放在一般规则之前.

```
In[7]:= ? log
```

```
Global`log

log[2 x_] := log[x] + log2

log[x_ y_] := log[x] + log[y]

log[x_^n_] := n log[x]
```

尽管在许多情况下，***Mathematica*** 能判断哪一个规则更一般一些，但总有些例外. 例如，在两个含有 `/;` 的规则中，就无法确定哪一个更一般，事实上也没有明确的顺序. 在顺序不清楚时，***Mathematica*** 总是按输入顺序保存规则.

相关教程

- 变换规则和定义

教程专集

- **Core Language**

立即定义和延时定义

Mathematica 中有两种赋值形式：*lhs* = *rhs* 和 *lhs* := *rhs*. 其主要区别是*什么时候* 计算 *rhs* 的值. *lhs* = *rhs* 是*立即赋值*，即 *rhs* 是在赋值时立即计算. 而 *lhs* := *rhs* 则是*延时赋值*，即赋值时并不计算 *rhs*，而是在需要 *lhs* 的值时才进行计算.

lhs=*rhs*（立即赋值）	赋值时立即计算 *rhs*
lhs:=*rhs*（延时赋值）	每次需要 *lhs* 时计算 *rhs*

Mathematica 中的两种赋值形式.

用 := 定义函数 ex.

```
In[1]:= ex[x_] := Expand[(1 + x)^2]
```

由于使用了 :=，该定义中仍然保持没有计算之前的形式.

```
In[2]:= ?ex

        Global`ex

        ex[x_] := Expand[(1 + x)^2]
```

使用 = 赋值时，右端立即被计算出来.

```
In[3]:= iex[x_] = Expand[(1 + x)^2]

Out[3]= 1 + 2 x + x^2
```

现在保存的定义是 Expand 命令的结果.

```
In[4]:= ?iex

        Global`iex

        iex[x_] = 1 + 2 x + x^2
```

当执行 ex 时，就调用 Expand.

```
In[5]:= ex[y + 2]

Out[5]= 9 + 6 y + y^2
```

iex 将它的变量替换到已展开的形式中去，给出不同形式的结果.

```
In[6]:= iex[y + 2]

Out[6]= 1 + 2 (2 + y) + (2 + y)^2
```

从上面的例子看出， = 和 := 都可以用来定义函数，要特别注意它们的差异.

一个常用的原则是：当一个表达式的值再不改变时用 =，而通过赋值求表达式的一个值时用 :=. 在无法确定时用 := 总比用= 好一些.

lhs=*rhs*	*rhs* 是 *lhs* 的永久值 （如 *f*[*x*_] = 1 - *x*^2）
lhs:=*rhs*	*rhs* 给出了需要 *lhs* 的值时执行的一个指令，（如 *f*[*x*_] := Expand[1 - *x*^2]）

用 = 和 := 赋值的含意.

尽管 := 比 = 用得多一些，但还有必须用 = 定义函数的一个重要情形. 当进行一个运算得到具有符号参数 x 的结果时，还需要进一步得到对应于不同 x 的结果. 一种方式是用 /. 将适当的规则用于 x. 通常用 = 去定义变量 x 的函数就较方便一些.

涉及 x 的一个表达式.

```
In[7]:= D[Log[Sin[x]]^2, x]

Out[7]= 2 Cot[x] Log[Sin[x]]
```

定义一个自变量可用 x 的值代入的函数.

```
In[8]:= dlog[x_] = %
Out[8]= 2 Cot[x] Log[Sin[x]]
```

x 为 1 + a 时的结果.

```
In[9]:= dlog[1 + a]
Out[9]= 2 Cot[1 + a] Log[Sin[1 + a]]
```

上面例子中值得注意的一点是在模式 x_ 中出现的名称 x 没有任何特殊支持，它仅是一个符号，与其它表达式中出现的 x 没有区别.

---

f[*x*_]=*expr*	对任何指定的 *x* 值，函数的值为 *expr*

定义表达式值的函数.

= 和 := 不仅用来定义函数，还用来给变量赋值. *x* = *value* 立即求 *value* 的值，并将其赋于 *x*. 另一方面，*x* := *value* 不立即求值 *value*. 而是在每次使用 *x* 时才计算其值.

计算 RandomReal[] 求出一个伪随机数，赋给 r1.

```
In[10]:= r1 = RandomReal[]
Out[10]= 0.0560708
```

这里 RandomReal[] 先不计算，在每次使用 r2 时进行计算.

```
In[11]:= r2 := RandomReal[]
```

此处给出 r1 和 r2 的值.

```
In[12]:= {r1, r2}
Out[12]= {0.0560708, 0.6303}
```

r1 的值维持不动，每次使用 r2 时，就产生一个伪随机数.

```
In[13]:= {r1, r2}
Out[13]= {0.0560708, 0.359894}
```

在一系列值语句中，要特别注意立即赋值和延时赋值的区别.

令 a 等于 1.

```
In[14]:= a = 1
Out[14]= 1
```

这里计算 a + 2 得到 3，将结果赋于 ri.

```
In[15]:= ri = a + 2
Out[15]= 3
```

这里 a + 2 先不计算，等到 rd 使用时求其值.

```
In[16]:= rd := a + 2
```

此处 ri 和 rd 值相同.

```
In[17]:= {ri, rd}
Out[17]= {3, 3}
```

此时 a 的值被改变.

```
In[18]:= a = 2
```

```
Out[18]= 2
```

现在 rd 使用 a 的新值，而 ri 保持不动.

```
In[19]:= {ri, rd}
```

```
Out[19]= {3, 4}
```

可以用延时赋值 *t* := *rhs* 来设置在不同环境下有不同值的变量. 当每次需要 *t* 的值时，就用与 *rhs* 有关变量的当前值来计算它的值.

延时赋值的右端暂不计算.

```
In[20]:= t := {a, Factor[x^a - 1]}
```

令 a 等于 4，再求出 t 的值.

```
In[21]:= a = 4; t
```

$$\text{Out[21]= } \left\{4, (-1+x)(1+x)\left(1+x^2\right)\right\}$$

这里令 a 等于6.

```
In[22]:= a = 6; t
```

$$\text{Out[22]= } \left\{6, (-1+x)(1+x)\left(1-x+x^2\right)\left(1+x+x^2\right)\right\}$$

在上面的例子中，符号 a 是一个全局变量，它的值决定了 t 的大小. 当参数中大部分偶尔才变化时，用这种方式是方便的. 但必须认识到明显的或隐含的变量之间的依赖关系是容易混淆的，应该尽可能地使用函数明确地反映依赖关系.

lhs->*rhs*	给出规则后就计算 *rhs*
lhs:>*rhs*	使用规则时计算 *rhs*

Mathematica 中的两种变换规则.

与 ***Mathematica*** 中的立即或延时赋值类似，可以建立立即或延时的变换规则.

给出规则时其左端立即计算.

```
In[23]:= f[x_] -> Expand[(1 + x)^2]
Out[23]= f[x_] → 1 + 2 x + x^2
```

这类规则不一定十分有用.

```
In[24]:= f[x_] -> Expand[x]
Out[24]= f[x_] → x
```

规则的右端暂不计算，每次使用规则时才求它的值.

```
In[25]:= f[x_] :> Expand[x]
Out[25]= f[x_] :⇢ Expand[x]
```

使用规则时就进行展开.

```
In[26]:= f[(1 + p)^2] /. f[x_] :> Expand[x]
Out[26]= 1 + 2 p + p^2
```

与赋值的情形类似，当用确定的值代替表达式的值时用 ->，而当给出一个求值的命令时用 :>.

相关教程

- 变换规则和定义

教程专集

- **Core Language**

保留已有值的函数

用 := 定义的函数中，调用是函数的值就会被反复计算. 但一些计算中，需要将同一组函数使用多次，这时就可以让***Mathematica*** 记住这些函数值以节省时间. 接下来就给出定义这种函数的方法.

---

f[x_]:=f[x]=*rhs*	定义一个保存已有值的函数

定义保存已有值的函数.

定义一个函数 f，它保存所有已得到的值.

```
In[1]:= f[x_] := f[x] = f[x - 1] + f[x - 2]
```

此处给出 f 递推结束的条件.

```
In[2]:= f[0] = f[1] = 1
Out[2]= 1
```

f 已有的定义.

```
Global`f
```

```
f[0] = 1
```

```
f[1] = 1
```

```
f[x_] := f[x] = f[x - 1] + f[x - 2]
```

求 f[5] 时，需要计算 f[5]，f[4]， ... f[2].

In[4]:=
```
f[5]
```

Out[4]= 8

到此为止的 f 值全部保存着.

In[5]:=
```
?f
```

```
Global`f
```

```
f[0] = 1
```

```
f[1] = 1
```

```
f[2] = 2
```

```
f[3] = 3
```

```
f[4] = 5
```

```
f[5] = 8
```

```
f[x_] := f[x] = f[x - 1] + f[x - 2]
```

此时再求 f[5] 时，***Mathematica*** 可立即找出，不需要重新计算.

```
In[6]:= f[5]
Out[6]= 8
```

用户可以看到定义如 f[x_] := f[x] = f[x - 1] + f[x - 2] 是如何工作的. 函数 f[x_] 被定义为这个“程序” f[x] = f[x - 1] + f[x - 2]. 当调用函数 f 的值时，执行该“程序”. 该程序首先计算 f[x - 1] + f[x - 2]，再将结果保存到 f[x] 中.

在 ***Mathematica*** 中进行递推运算时，使用能保存已有值的函数是一个好方法. 递推的一个典型情况是函数 f 在整数 x 的值通过它在 $x-1$、$x-2$ 等值给出. **Fibonacci** 函数定义 $f(x) = f(x-1) + f(x-2)$ 就是这样的一种递推，在计算 $f(10)$ 时，需反复递推，如需要反复计算 $f(5)$ 多次，故此时记住 $f(5)$ 的值下次直接找出就比反复计算要好.

关于记住函数值，这里存在一个权衡. 在记住了一些函数值时，查找就比计算快，但这需要占用内存. 通常当重复计算的代价很大时保存函数值，且需要保存的函数不宜太多.

相关教程

- 变换规则和定义

教程专集

- **Core Language**

与不同符号相关的定义

$f[args] = rhs$ 或 $f[args] := rhs$ 等将 f 和定义联系起来. 当输入 $?f$ 时，就会显示 f 的定义. 一般我们把 f 作为头部的表达式称为 f 的下值.

Mathematica 中也有上值，上值可以把表达式与不直接出现在其头部的符号联系起来.

在定义 `Exp[g[x_]] := rhs` 中，可以把定义看作为符号 `Exp` 的下值，但是从组织和效率的角度来看未必是最好的方式.

较好的方式是将 `Exp[g[x_]] := rhs` 与 `g` 联系起来，即定义为 `g` 的上值.

$f[args]:=rhs$	定义 f 的下值
$f[g[args],\ldots]\text{^}:=rhs$	定义 g 的上值

定义与不同符号的联系.

定义 `f` 的下值.

```
In[1]:= f[g[x_]] := fg[x]
```

查看 `f` 的定义.

```
In[2]:= ?f
```

```
Global`f
```

```
f[g[x_]] := fg[x]
```

定义 g 的上值.

```
In[3]:= Exp[g[x_]] ^:= expg[x]
```

此定义与 g 相关.

```
In[4]:= ?g
```

```
Global`g
```

```
e^g[x_] ^:= expg[x]
```

此定义与 Exp 无关.

```
In[5]:= ?? Exp
```

```
Exp[z] 给出
    z                    的指数函数. >>
```

```
Attributes[Exp] = {Listable, NumericFunction,
```

用定义求表达式的值.

```
In[6]:= Exp[g[5]]
Out[6]= expg[5]
```

在简单的情况下，将 $f[g[x]]$ 作为 f 的下值或 g 的上值. 然而，其中之一比另一个更自然或有效一些.

一般的原则是 f 比 g 更常用的时候，应该将 $f[g[x]]$ 定义为 g 的上值. 因此，例如，在 $\text{Exp}[g[x]]$ 中，Exp 是***Mathematica*** 中的内部函数，g 是引入的一个函数，此时将 $\text{Exp}[g[x]]$ 作为 g 的上值比较合理和自然.

定义 g 的上值.

```
In[3]:= Exp[g[x_]] ^:= expg[x]
```

此定义与 g 相关.

```
In[4]:= ?g
```

```
Global`g

e^g[x_] ^:= expg[x]
```

此定义与 Exp 无关.

```
In[5]:= ?? Exp
```

```
Exp[z] 给出
    z                    的指数函数. >>

= {Listable, NumericFunction, Protected, ReadProtecte
```

用定义求表达式的值.

```
In[6]:= Exp[g[5]]
Out[6]= expg[5]
```

在简单的情况下，将 $f[g[x]]$ 作为 f 的下值或 g 的上值. 然而，其中之一比另一个更自然或有效一些.

一般的原则是 f 比 g 更常用的时候，应该将 $f[g[x]]$ 定义为 g 的上值. 因此，例如，在 $\text{Exp}[g[x]]$ 中，Exp 是***Mathematica*** 中的内部函数，g 是引入的一个函数，此时将 $\text{Exp}[g[x]]$ 作为 g 的上值比较合理和自然.

定义 g 的上值.

```
In[7]:= g /: g[x_] + g[y_] := gplus[x, y]
```

到此为止 g 的定义.

```
In[8]:= ?g
```

```
Global`g

e^g[x_] ^:= expg[x]

g[x_] + g[y_] ^:= gplus[x, y]
```

定义 g 的和.

```
In[9]:= g[5] + g[7]
Out[9]= gplus[5, 7]
```

由于模式 g[x_] + g[y_] 的完全形式是 Plus[g[x_], g[y_]]，这可以定义为 Plus 的下值，然而将它看作 g 的上值更好一些.

一般来说，当 ***Mathematica*** 遇到一个函数时，它测试所有已给出的该函数的定义. 把 g[x_] + g[y_] 定义为 Plus 的下值时，则当遇到 Plus 时就试用这个定义. 由于每次 ***Mathematica*** 遇到加法时都测试这个定义，故运行时速度较慢.

而把 g[x_] + g[y_] 看作 g 的上值时，仅当 g 出现在 Plus 内时才测试这个定义. 因 g 出现的频率比 Plus 出现的少，故这个更有效一些.

$f[g]\text{^=}value$ 或 $f[g[args]]\text{^=}value$	与 g 相关而与 f 无关的赋值
$f[g]\text{^:=}value$ 或 $f[g[args]]\text{^:=}value$	与 g 有关的延时赋值
$f[arg_1, arg_2, \ldots]\text{^=}value$	与所有 arg_i 的头部有关的赋值

定义上值的有效方法.

上值的典型用法是在建立一些对象性质的数据库之中，使用上值可以把所涉及对象的定义与对象相联系，而不是与其给出的性质相联系.

定义 square 的上值给出其面积.

```
In[10]:= area[square] ^= 1
Out[10]= 1
```

加入 **perimeter** 的定义.

```
In[11]:= perimeter[square] ^= 4
Out[11]= 4
```

这两个定义都与 square 对象相关联.

```
In[12]:= ?square
```

```
Global`square

area[square] ^= 1

perimeter[square] ^= 4
```

一般可以将一个表达式的定义和出现在表达式高层次中的符号联系起

来. 在形如 *f*[*args*] 的表达式中，当 *g* 本身或以 *g* 为头部的对象出现在 *args* 中时，就可以定义 *g* 的上值. 但当 *g* 出现在较低的层次中时，就无法将这些定义与它相关联.

g 是变量的头部，可以使其与该定义相关.

```
In[13]:= g /: h[w[x_], g[y_]] := hwg[x, y]
```

g 在左端出现的层次太低，故无法将它与定义相关联.

```
In[14]:= g /: h[w[g[x_]], y_] := hw[x, y]
                TagSetDelayed::tagpos :
                 Tag g in h[w[g[x_]], y_] is too deep
                   for an assigned rule to be found. >>
Out[14]= $Failed
```

f[...]:=*rhs*	*f* 的下值
f/: *f*[*g*[...]][...]:=*rhs*	*f* 的下值
g/:*f*[...,*g*,...]:=*rhs*	*g* 的上值
g/:*f*[...,*g*[...],...]:= *rhs*	*g* 的上值

定义中符号的位置.

如 "表达式的含义" 中描述的，在 ***Mathematica*** 中，符号可作为标记来表明表达式的“类型”. 例如，复数在 ***Mathematica*** 中的内部形式为 `Complex[x, y]`，这里 `Complex` 表明此对象是一个复数.

上值提供了一个运算和标记的简便方法. 例如，当需要引入类型为 `quat` 的一组对象时，就可以用 `quat[data]` 来表示每一个对象.

通过定义 quat 对 Plus 和 Times 的上值的途径来实现.

定义 quat 关于 Plus 的上值.

```
In[15]:= quat[x_] + quat[y_] ^:= quat[x + y]
```

刚定义的上值用来简化这一表达式.

```
In[16]:= quat[a] + quat[b] + quat[c]
Out[16]= quat[a + b + c]
```

定义 quat 关于 Plus 运算的上值，就是将 Plus 的定义域扩大到 quat 上，即告诉 ***Mathematica*** 当 quat 的元素相加时就使用加法的规则.

在定义 quat 中的加法时，可以用形如 quatPlus 的特定加法运算符，但用 ***Mathematica*** 中的标准运算 Plus，再增添一些对 quat 元素的特殊性质更方便一些.

可以将上值看作实现一些面向对象编程功能的一种途径. 符号 quat 表示一类对象，quat 的各种上值指出 quat 的元素在某些运算或情况下的性质.

相关教程

- 变换规则和定义
- **Core Language**

定义数量值

在定义如 f[x_] := value 中，***Mathematica*** 的值将被赋给所有的

函数 f. 但有时需要定义一个求值时才使用的值.

expr=*value*	定义一个任何情况下都用的量
N[*expr*]=*value*	定义一个用作近似值的量

定义普通值和数量值.

定义 f 的一个数量值.

```
In[1]:= N[f[x_]] := Sum[x^-i / i^2, {i, 20}]
```

定义数量值不影响 f 的其它值.

```
In[2]:= f[2] + f[5]
Out[2]= f[2] + f[5]
```

进行近似计算时使用刚才的数量值.

```
In[3]:= N[%]
Out[3]= 0.793244
```

对函数和符号都可以定义数量值，NIntegrate 和 FindRoot 等数值函数都可以使用数量值.

N[*expr*]=*value*	定义默认精度的数量值
N[*expr*, {*n*, Infinity}]=*value*	定义有 *n* 位精度的数量值

定义与精度有关的数量值.

定义符号 const 的数量值，在求积中使用 4 n + 5 项，并具有 n 位精度.

```
In[4]:= N[const, {n_, Infinity}] :=
         Product[1 - 2^-i, {i, 2, 4 n + 5}]
```

有30位精度的 const 的值.

```
In[5]:= N[const, 30]
Out[5]= 0.577576190173204842557799443858
```

在 *Mathematica* 中，数量值的处理与上值类似，当定义了 f 的数量值以后，*Mathematica* 就象求值运算 N 中 f 的上值一样来输入这一定义.

相关教程

- 变换规则和定义

教程专集

- **Core Language**

修改内部函数

在 *Mathematica* 中可以对任何表达式定义变换规则. 不仅可以定义添加到 *Mathematica* 中去的函数，而且还可以对 *Mathematica* 的内部函数进行变换，于是就可以增强或修改内部函数的功能.

这一功能是强大的，同时也具有潜在的危险. *Mathematica* 永远遵

为了避免在修改内部函数过程中的错误，***Mathematica*** 限制对内部函数重新定义. 当要定义内部函数时，首先要去掉这种限制. 完成定义以后要恢复这一限制以免以后犯错误.

`Unprotect[f]`	去掉保护限制
`Protect[f]`	加上保护限制

函数的保护.

内部函数通常被“保护”，不能重新定义它们.

```
In[1]:= Log[7] = 2
        Set::write: Tag Log in Log[7] is Protected. >>
Out[1]= 2
```

去掉 Log 的保护.

```
In[2]:= Unprotect[Log]
Out[2]= {Log}
```

现在可以自行定义 Log 函数. 即使这个定义不正确，***Mathematica*** 也允许你这样定义.

```
In[3]:= Log[7] = 2
Out[3]= 2
```

不论这个定义正确与否，***Mathematica*** 将在所有可能的情况下使用用户的定义.

```
In[4]:= Log[7] + Log[3]
Out[4]= 2 + Log[3]
```

清除 Log 的不正确定义.

```
In[5]:= Log[7] =.
```

恢复对 Log 的保护.

```
In[6]:= Protect[Log]
Out[6]= {Log}
```

用户定义的函数高于 ***Mathematica*** 内部函数. 一般说来，***Mathematica*** 总是先使用用户定义的函数.

Mathematica 的内部规则是希望能进行宽广的各种运算. 在有些不愿意用内部规则的情况下，可以用可以定义的规则来超越内部的规则.

简化 Exp[Log[*expr*]] 时有一个内部规则.

```
In[7]:= Exp[Log[y]]
Out[7]= y
```

可以自行定义 Exp[Log[*expr*]] 来代替内部规则.

```
In[8]:= (Unprotect[Exp];
         Exp[Log[expr_]] := explog[expr];
         Protect[Exp];)
```

现在使用了自己的定义，而不是内部规则.

```
In[9]:= Exp[Log[y]]
Out[9]= explog[y]
```

相关指南

- 属性

相关教程

- 变换规则和定义

教程专集

- **Core Language**

值集的操作

DownValues[*f*]	给出 *f* 的下值集
UpValues[*f*]	给出 *f* 的上值集
DownValues[*f*]=*rules*	建立 *f* 的下值
UpValues[*f*]=*rules*	建立 *f* 的上值

建立和寻找符号的值.

Mathematica 按变换规则列表的方式有效地保存给出的定义，当调

到一个符号时，就调用与它有关的规则列表.

在绝大部分情况下，不需要涉及与定义有关的变换规则，只需要用 *lhs* = *rhs* 和 *lhs* =. 来添加或删除规则. 然而，在有些情况下，直接进入这些规则是十分有用的.

定义 f.

```
In[1]:= f[x_] := x^2
```

定义 f 的规则.

```
In[2]:= DownValues[f]
Out[2]= {HoldPattern[f[x_]] :> x^2}
```

注意，建立 DownValues 和 UpValues 的返回值规则是为了让左、右两边都不进行计算. 左端包含在 HoldPattern 中，规则被延时，所以右边不立即计算.

正如在 "定义函数" 中讨论的一样，***Mathematica*** 按特殊定义出现在一般定义之前的原则对定义排序. 但事实上，没有唯一确定的方式去排序，可以用与 ***Mathematica*** 默认次序不同的方式去排序，对用 DownValues 和 UpValues 得到的规则表重新排序就达到目的.

下面是对象 g 的一些定义.

```
In[3]:= g[x_ + y_] := gp[x, y]; g[x_ y_] := gm[x, y]
```

这里显示了定义的默认顺序.

```
In[4]:= DownValues[g]
Out[4]= {HoldPattern[g[x_ + y_]] :> gp[x, y],
         HoldPattern[g[x_ y_]] :> gm[x, y]}
```

这里对 g 定义的顺序进行反向.

```
In[5]:= DownValues[g] = Reverse[DownValues[g]]

Out[5]= {HoldPattern[g[x_ y_]] :> gm[x, y],
         HoldPattern[g[x_ + y_]] :> gp[x, y]}
```

相关教程

- 变换规则和定义

教程专集

- **Core Language**

函数和编程

定义函数

过程函数

处理选项

循环运算

函数的变换规则

教程专集

- **Core Language**

自定义函数

Mathematica 已有许多内置函数. 在这一节， 我们讨论如何把用户的函数添加到 ***Mathematica*** 中.

例如， 将自变量的平方作为函数 f 添加到 ***Mathematica*** 中. 定义这个函数的命令是 f[x_] := x^2. 等式左端的下划线 _ （被称为“空白”） 是非常重要的； 它的含义将在后面讨论. 现在只需记住：把下划线 _ 放在定义的左端，而不是在右端.

这里定义函数 f， 注意下划线 _ 在左端：

```
In[1]:= f[x_] := x^2
```

f 将自变量进行平方：

```
In[2]:= f[a + 1]
```

$$\text{Out[2]= } (1 + a)^2$$

自变量可以是一个数：

```
In[3]:= f[4]
```

```
Out[3]= 16
```

自变量也可以是一个更复杂的表达式：

```
In[4]:= f[3 x + x^2]
```

$$\text{Out[4]= } \left(3\,x + x^2\right)^2$$

在运算中可以使用 f：

```
In[5]:= Expand[f[(x + 1 + y)]]
```

$$\text{Out[5]}= 1 + 2x + x^2 + 2y + 2xy + y^2$$

此处显示 f 的定义：

```
In[6]:= ?f
```

```
Global`f

f[x_] := x^2
```

`f[x_]:=x^2`	定义函数 f
`?f`	显示 f 的定义
`Clear[f]`	清除所有 f 的定义

在 ***Mathematica*** 中定义函数.

在 ***Mathematica*** 中， 用户定义像 f 一样的函数时使用的函数名仅仅是一个符号. 因此，应该确保使用的名称不以大写字母开头，以避免与 ***Mathematica*** 的内部函数混淆. 用户还应当在同一进程当中，不使用前面已用过的名称.

Mathematica 函数能够有任意多个自变量：

```
In[7]:= hump[x_, xmax_] := (x - xmax)^2 / xmax
```

可以像使用内部函数一 样，使用自定义的 hump 函数：

```
In[8]:= 2 + hump[x, 3.5]
```

$$\text{Out[8]}= 2 + 0.285714\,(-3.5 + x)^2$$

这个重新定义 hump 函数，它覆盖了前面的定义：

```
In[9]:= hump[x_, xmax_] := (x - xmax)^4
```

显示新的定义：

```
In[10]:= ?hump
```

```
Global`hump

hump[x_, xmax_] := (x - xmax)^4
```

清除所有 hump 的定义：

```
In[11]:= Clear[hump]
```

当用户使用完一个定义函数时， 最好清除该函数定义. 否则，当在同一 ***Mathematica*** 进程的后面使用同名函数， 但用于不同的目的时，将会遇到麻烦. 用户可以用 `Clear[f]` 清除 f 函数或符号的所有定义.

相关教程

- 函数和编程
- 使用变量和函数

教程专集

- **Core Language**

过程函数

的命令. 这时，可以通过定义一个包含输入命令的*函数* 来节省许多输入.

这里构造一个三项的连乘，并展开这个结果.

```
In[1]:= Expand[Product[x + i, {i, 3}]]
```

$$\text{Out[1]}= 6 + 11\,x + 6\,x^2 + x^3$$

这里对四项的连乘做上面同样的事情.

```
In[2]:= Expand[Product[x + i, {i, 4}]]
```

$$\text{Out[2]}= 24 + 50\,x + 35\,x^2 + 10\,x^3 + x^4$$

这里定义函数 exprod，它构造 *n* 个项的连乘，然后展开.

```
In[3]:= exprod[n_] := Expand[Product[x + i, {i, 1, n}]]
```

使用该函数，即可进行任意项的 Product （连乘）和 Expand （展开）运算.

```
In[4]:= exprod[5]
```

$$\text{Out[4]}= 120 + 274\,x + 225\,x^2 + 85\,x^3 + 15\,x^4 + x^5$$

在 ***Mathematica*** 中定义的函数本质上是一个过程，它执行用户给定的命令. 在过程中，可以有若干步，通过分号将其分开.

用户从整个函数中得到的结果是过程的最后表达式. 注意，在这样的定义中，必须用圆括号将过程括起来.

```
In[5]:= cex[n_, i_] := (t = exprod[n];
         Coefficient[t, x^i])
```

此处“运行”这个过程.

```
In[6]:= cex[5, 3]
Out[6]= 85
```

$expr_1; expr_2; \ldots$	要进行运算的一列表达式
`Module[{a, b, ...}, proc]`	带有局部变量 a，b，… 的过程

构造过程.

在 ***Mathematica*** 中定义过程时，把过程中的变量定义成*局部变量*通常是个好主意. 这样它不会干扰过程外面的事情. 用户可以通过把过程建立模块（**Module**）来实现这一点，在其中用户给出要作为局部变量的变量列表.

上面定义的函数 `cex` 不是一个模块，所以 `t` 的值“逃出”，并且甚至在函数返回的时候仍然存在.

```
In[7]:= t
Out[7]= 120 + 274 x + 225 x^2 + 85 x^3 + 15 x^4 + x^5
```

这个函数被定义成带有局部变量 `u` 的模块.

```
In[8]:= ncex[n_, i_] := Module[{u}, u = exprod[n];
          Coefficient[u, x^i]]
```

此函数给出与前面的函数相同的结果.

```
In[9]:= ncex[5, 3]
Out[9]= 85
```

然而，现在 u 的值不再从函数中逃出.

```
In[10]:= u
Out[10]= u
```

相关指南

- 过程式编程

相关教程

- 函数和编程

教程专集

- **Core Language**

处理选项

Mathematica 有许多像 `Plot` 这样的内部函数，它们有许多用户能设置的选项. ***Mathematica*** 提供了处理这些选项的一般方法.

如果对函数如 `Plot` 的某个选项不明确设置，那么 ***Mathematica*** 将自动使用该选项的缺省值. 函数`Options`[*function*, *option*] 让用户找出特定选项的缺省值. 用户可以使用 `SetOptions`[*function*, *option* -> *value*] 重新设置缺省值. 注意，如果你重设了缺省值，它将一直保留到你明确改变它为止.

Options[*function*]	给出所有选项当前缺省值的列表
Options[*function*, *option*]	给出特定选项的缺省值
SetOptions[*function*, *option*->*value*, …]	重新设置缺省值

处理选项的缺省值.

这是函数 Plot 的选项 PlotRange 的缺省设置.

```
In[1]:= Options[Plot, PlotRange]
Out[1]= {PlotRange → {Full, Automatic}}
```

这里重新设置 PlotRange 选项. 分号阻止 ***Mathematica*** 显示 Plot 的相当长的选项列表.

```
In[2]:= SetOptions[Plot, PlotRange -> All];
```

除非重新设置它，PlotRange 选项的缺省值将一直是 All.

```
In[3]:= Options[Plot, PlotRange]
Out[3]= {PlotRange → All}
```

从 Plot 或 Show 中得到的图形对象存储了所选项的信息. 用户可以使用函数 Options 得到这些信息.

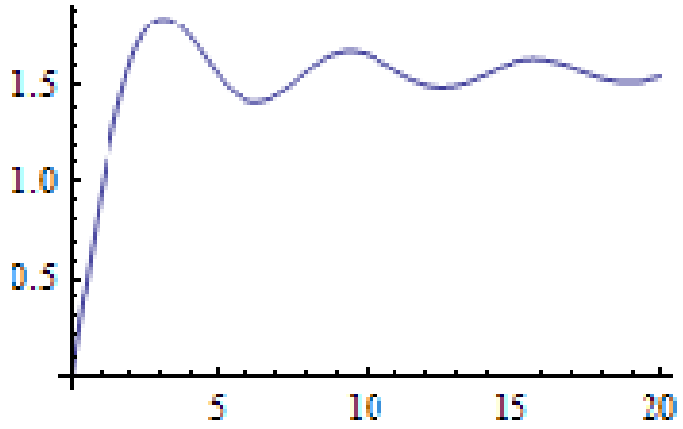
`Options`[*plot*]	给出特定图形所使用的全部选项
`Options`[*plot*, *option*]	显示特定选项的设置
`AbsoluteOptions`[*plot*, *option*]	显示特定选项的绝对形式，即使该选项的设置是 `Automatic` 或 `All`

获得绘图中使用的选项信息.

这是所有选项都为缺省值的图形.

```
In[4]:= g = Plot[SinIntegral[x], {x, 0, 20}]
```

Out[4]=



该图形的 `PlotRange` 选项值为 `All`.

```
In[5]:= Options[g, PlotRange]
Out[5]= {PlotRange → {All, All}}
```

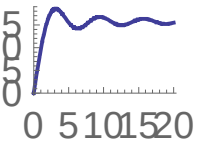
`AbsoluteOptions` 给出 `PlotRange` 选项的绝对的自动选择值.

```
In[6]:= AbsoluteOptions[g, PlotRange]
Out[6]= {PlotRange → {{4.08163 × 10^-7, 20.},
        {4.08163 × 10^-7, 1.85194}}}
```

的，这样的话，图形本身可以直接参与计算. 在笔记本界面上实现该功能的典型方法是，拷贝粘贴该图形，或者简单地在图形输出单元中输入以使得输出单元被转换成一个新的输入单元.

当一个不使用明确的 `ImageSize` 创建的图形被放入一个输入单元时，它将自动地根据空间缩小为一个更容易放置的输入大小.

下面的输入单元通过拷贝粘贴前面的例子中产生的输出图形来创建.

```
In[7]:= AbsoluteOptions[ 

, PlotRange]

Out[7]= {PlotRange → {{4.08163 × 10^-7, 20.},
    {4.08163 × 10^-7, 1.85194}}}
```

相关指南

- 选项管理

相关教程

- 图形和声音

教程专集

- **Core Language**
- **Visualization and Graphics**

循环运算

使用 ***Mathematica*** 时，用户有时需要将一个运算重复许多次，有许多方法实现这一点. 通常最自然的方法是建立一个结构，例如，包含许多元素的列表，然后把运算运用到每个元素上.

另一个方法是使用 ***Mathematica*** 函数 Do，它的功能与 **C** 或者 **Fortran** 语言中的循环结构非常像. 与"求和与求积"中介绍的函数 Sum 与 Product 相同，Do 使用的是 ***Mathematica*** 迭代器.

Do[*expr*, {*i*, i_{max}}]	*i* 从 1 到 i_{max} 计算 *expr* 的值
Do[*expr*, {*i*, i_{min}, i_{max}, *di*}]	*i* 从 i_{min} 到 i_{max}、步长 *di*，计算 *expr* 的值
Print[*expr*]	打印 *expr*
Table[*expr*, {*i*, i_{max}}]	生成 *i* 从 1 到 i_{max} 的 *expr* 的值的列表

实现循环运算.

这里打印出前5个阶乘的值.

```
In[1]:= Do[Print[i!], {i, 5}]
        1
        2
        6
        24
        120
```

得到一个结果的列表常常是很有用的，这样可以作进一步的处理.

```
In[2]:= Table[i!, {i, 5}]
Out[2]= {1, 2, 6, 24, 120}
```

运算.

```
In[3]:= r = 1; Do[r = 1 / (1 + r), {100}]; r
```

$$\text{Out[3]}= \frac{573\,147\,844\,013\,817\,084\,101}{927\,372\,692\,193\,078\,999\,176}$$

相关指南

- 过程式编程

相关教程

- 函数和编程

教程专集

- **Core Language**

函数的变换规则

"符号的值" 节讨论过如何使用形如 *x* –> *value* 的变换规则来用一个值替换符号. 然而，在 ***Mathematica*** 中，变换规则的表示法是相当一般的. 用户不仅能对符号，也能对任何 ***Mathematica*** 表达式建立变换规则.

使用变换规则 x –> 3 用 3 替换 x.

```
In[1]:= 1 + f[x] + f[y] /. x -> 3
```

```
Out[1]= 1 + f[3] + f[y]
```

也可以对 f[x] 使用变换规则. 这个规则并不影响 f[y].

```
In[2]:= 1 + f[x] + f[y] /. f[x] -> p
Out[2]= 1 + p + f[y]
```

f[t_] 是一个*模式*，它代表具有任何自变量的 f.

```
In[3]:= 1 + f[x] + f[y] /. f[t_] -> t^2
Out[3]= 1 + x^2 + y^2
```

Mathematica 变换规则最强有力的方面或许是它们不仅能用于表达式，而且能用于模式（***patterns***）. 模式是一个诸如f[t_] 这样的包含下划线的表达式. 下划线可以代表任何表达式. 这样，对 f[t_] 的变换规则将变换具有*任何* 自变量的函数 f. 注意，作为对照，对不带有下划线的 f[x] 的变换规则只变换表达式 f[x]，而对诸如 f[y] 等表达式没有任何作用.

当用户定义一个函数，例如 f[t_] := t^2 时，所做的事情是告诉 ***Mathematica***，在任何可能的时候，自动使用变换规则 f[t_] -> t^2.

可以对任何形式的表达式建立变换规则.

```
In[4]:= f[a b] + f[c d] /. f[x_ y_] -> f[x] + f[y]
Out[4]= f[a] + f[b] + f[c] + f[d]
```

这里对 x^p_ 使用变换规则.

```
In[5]:= 1 + x^2 + x^4 /. x^p_ -> f[p]
Out[5]= 1 + f[2] + f[4]
```

模式和变换规则，我们可以说 ***Mathematica*** 的所有表达式都有一个确定的符号结构；而变换规则允许用户变换结构的一部分.

相关教程

- 函数和编程

教程专集

- **Core Language**

函数操作

函数名是表达式

函数的重复调用

函数作用于列表和其它表达式

函数作用于表达式的部分项

纯函数

由函数产生列表

用函数选择表达式的项

具有非符号头部的表达式

算子运算

结构的操作

序列

- Core Language

函数名是表达式

在表达式 $f[x]$ 中，“函数名” f 本身就是一个表达式，可以像其他表达式一样来处理它.

用变换规则修改一个函数名.

```
In[1]:= f[x] + f[1 - x] /. f -> g
Out[1]= g[1 - x] + g[x]
```

任何赋值可作为函数名.

```
In[2]:= p1 = p2; p1[x, y]
Out[2]= p2[x, y]
```

用函数名作变量定义一个函数.

```
In[3]:= pf[f_, x_] := f[x] + f[1 - x]
```

将 Log 作为函数名.

```
In[4]:= pf[Log, q]
Out[4]= Log[1 - q] + Log[q]
```

像其它表达式一样，对函数名进行操作是 ***Mathematica*** 语言符号运算的重要特性，这特性使得下面讨论的各种对函数的操作能顺利进行

表达式，但功能运算函数不仅能处理一般的数据，而且也能处理函数本身. 例如，InverseFunction 将 ***Mathematica*** 函数名作为自变量，并求出它的反函数.

InverseFunction 是一个功能运算，它将函数名作为自变量，其返回值是一个反函数.

```
In[5]:= InverseFunction[ArcSin]
Out[5]= Sin
```

由 InverseFunction 得到的是一个函数，它可作用于数据.

```
In[6]:= %[x]
Out[6]= Sin[x]
```

InverseFunction 也可以用来进行符号运算.

```
In[7]:= InverseFunction[f][x]
Out[7]= f^(-1)[x]
```

Mathematica 中有许多功能运算. 有些用于运算，有些用于各种过程和编程.

对高级符号运算不熟悉的人可能无法理解这一节讨论的大部分函数运算. 这些运算比较难懂，但值得研究，功能运算提供了运用 ***Mathematica*** 的一个非常有效的途径.

相关教程

- 函数操作

函数的重复调用

许多程序中涉及迭代过程. Nest 和 NestList 可以进行有效的迭代.

Nest[f,x,n]	f 对 x 复合 n 次
NestList[f,x,n]	产生列表 {x, f[x], f[f[x]], ...}, 其中 f 最多复合 n 次

重复调用一元函数.

Nest[f, x, n] 将 f 对 x 复合 n 次.

```
In[1]:= Nest[f, x, 4]
Out[1]= f[f[f[f[x]]]]
```

给出函数的复合序列.

```
In[2]:= NestList[f, x, 4]
Out[2]= {x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

定义一个简单函数.

```
In[3]:= recip[x_] := 1 / (1 + x)
```

用 Nest 对函数进行迭代.

```
In[4]:= Nest[recip, x, 3]
```

$$\text{Out[4]}= \frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$$

Nest 和 NestList 对函数进行确定数目的复合. 有时需要对函数进行多次复合直到它不再变化为止. FixedPoint 和 FixedPointList 可以实现这一功能.

FixedPoint[*f*, *x*]	将 *f* 复合到结果不变为止
FixedPointList[*f*, *x*]	产生 *f* 的复合序列 {*x*, *f*[*x*], *f*[*f*[*x*]], ...} 直到结果不变为止

将函数复合到结果不变为止.

用牛顿迭代法求 $\sqrt{3}$ 的近似值.

```
In[5]:= newton3[x_] := N[1 / 2 (x + 3 / x)]
```

从 $x = 1$ 开始迭代5次.

```
In[6]:= NestList[newton3, 1.0, 5]
Out[6]= {1., 2., 1.75, 1.73214, 1.73205, 1.73205}
```

使用函数 FixedPoint，自动继续运用 newton3 直到结果不再变为止.

```
In[7]:= FixedPoint[newton3, 1.0]
Out[7]= 1.73205
```

给出迭代序列.

```
In[8]:= FixedPointList[newton3, 1.0]
Out[8]= {1., 2., 1.75, 1.73214,
         1.73205, 1.73205, 1.73205}
```

NestWhile[*f*, *x*, *test*]	重复运用函数 *f*，不断迭代直到 *test* 不为 True 时结束
NestWhileList[*f*, *x*, *test*]	产生迭代序列 {*x*, *f*[*x*], *f*[*f*[*x*]], ...}，直到 *test* 不为 True 时结束
NestWhile[*f*, *x*, *test*, *m*], NestWhileList[*f*, *x*, *test*, *m*]	每一步迭代中将 *m* 个最新的结果作为 *test* 的自变量进行判断
NestWhile[*f*, *x*, *test*, All], NestWhileList[*f*, *x*, *test*, All]	将迭代产生的所有结果代入 *test* 自变量中进行判断

重复迭代直到某一条件不成立时停止.

产生一个函数，其功能为对一个数除以2.

```
In[9]:= divide2[n_] := n / 2
```

重复作用函数 divide2 直到结果不是偶数时停止.

```
In[10]:= NestWhileList[divide2, 123 456, EvenQ]

Out[10]= {123 456, 61 728, 30 864, 15 432, 7716, 3858, 1929
```

重复使用 newton3 直到相邻两结果相同为止，正如 FixedPointList 中的一样.

```
In[11]:= NestWhileList[newton3, 1.0, Unequal, 2]

Out[11]= {1., 2., 1.75, 1.73214,
           1.73205, 1.73205, 1.73205}
```

重复迭代到结果与前面某一次结果相同时结束.

```
In[12]:= NestWhileList[Mod[5 #, 7] &, 1, Unequal, All]

Out[12]= {1, 5, 4, 6, 2, 3, 1}
```

Nest 等重复作用一元函数 f，每次作用中都是将前一次的结果作为 f 的自变量.

将这一过程推广到二元函数是非常重要的，碰到的问题是每次迭代只产生一个新结果，而二元函数有两个自变量，解决这一问题的一个途径就是作用迭代列中相邻的两个结果.

FoldList[f, x, {a, b, ...}]	产生迭代序列 {x, $f[x, a]$, $f[f[x, a], b]$, ...}
Fold[f, x, {a, b, ...}]	用 FoldList[f, x, {a, b, ...}] 产生迭代序列中的最后一项

对二元函数进行迭代的方法.

FoldList 应用举例.

```
In[13]:= FoldList[f, x, {a, b, c}]
Out[13]= {x, f[x, a], f[f[x, a], b], f[f[f[x, a], b], c]}
```

Fold 给出由 FoldList 产生的最后一项.

```
In[14]:= Fold[f, x, {a, b, c}]
Out[14]= f[f[f[x, a], b], c]
```

产生求和序列.

```
In[15]:= FoldList[Plus, 0, {a, b, c}]
Out[15]= {0, a, a + b, a + b + c}
```

用 Fold 和 FoldList 可以在 ***Mathematica*** 中写出高效的程序. 在某些情况下，把 Fold 和 FoldList 当作由第二个变量作为索引用来产生一个函数系列的简单嵌套是有益的.

定义函数 nextdigit.

```
In[16]:= nextdigit[a_, b_] := 10 a + b
```

定义另一个内置函数 FromDigits.

```
In[17]:= fromdigits[digits_] :=
          Fold[nextdigit, 0, digits]
```

下面是具体函数操作的例子.

```
In[18]:= fromdigits[{1, 3, 7, 2, 9, 1}]
Out[18]= 137 291
```

相关教程

- 函数操作

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

函数作用于列表和其它表达式

函数 `f[{a, b, c}]` 有许多自变量. 有时仅需要将这些变量作为一个整体作用，可以用 `Apply` 来实现这一功能.

列表中的每一个元素是 `f` 的自变量.

```
In[1]:= Apply[f, {a, b, c}]
Out[1]= f[a, b, c]
```

`Times[a, b, c]` 给出列表中这些元素的积.

```
In[2]:= Apply[Times, {a, b, c}]
Out[2]= a b c
```

下面是一个函数的定义，它与内置函数 GeometricMean 工作方式相近，表示为 Apply.

```
In[3]:= geom[list_] :=
          Apply[Times, list]^(1 / Length[list])
```

Apply[f, {a, b, ...}]	f 作用于列表，得出 f[a, b, ...]
Apply[f, expr] 或 f@@expr	f 作用于表达式的顶层
Apply[f, expr, {1}] 或 f@@@expr	f 作用于表达式的第一层
Apply[f, expr, lev]	f 作用于表达式的某些指定层

将函数作用于一个列表或其它表达式.

Apply 一般用指定的函数代替表达式的头部，本例中是用 List 来代替 Plus.

```
In[4]:= Apply[List, a + b + c]
Out[4]= {a, b, c}
```

定义矩阵.

```
In[5]:= m = {{a, b, c}, {b, c, d}}
Out[5]= {{a, b, c}, {b, c, d}}
```

没有明确指定层时，Apply 用 f 代替顶层.

```
In[6]:= Apply[f, m]
Out[6]= f[{a, b, c}, {b, c, d}]
```

f 作用于 m 的第一层的部分项上.

```
In[7]:= Apply[f, m, {1}]
Out[7]= {f[a, b, c], f[b, c, d]}
```

f 作用于第0层、第一层.

```
In[8]:= Apply[f, m, {0, 1}]
Out[8]= f[f[a, b, c], f[b, c, d]]
```

相关教程

- 函数操作

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

函数作用于表达式的部分项

当有一系列元素时，经常需要将一个函数分别作用于每一项中，这可以在 ***Mathematica*** 中用 Map 来实现.

f 作用于列表中的每一个元素.

```
In[1]:= Map[f, {a, b, c}]
Out[1]= {f[a], f[b], f[c]}
```

定义作用于一个列表中前两个元素上的函数.

```
In[2]:= take2[list_] := Take[list, 2]
```

用 Map 将 take2 作用于每一个列表上.

```
In[3]:= Map[take2, {{1, 3, 4}, {5, 6, 7}, {2, 1, 6, 6}}]
Out[3]= {{1, 3}, {5, 6}, {2, 1}}
```

Map[*f*, {*a*, *b*, ...}]	将 *f* 作用于列表的每一个元素上得到 {*f*[*a*], *f*[*b*], ...}

将函数作用于列表中的每一个元素上.

Map[*f*, *expr*] 的作用是将函数 *f* 作用于表达式 *expr* 的每一个元素和组成部分上. 可以把 Map 用于任何表达式，不仅仅在列表上.

将 f 作用于和式的每个元素.

```
In[4]:= Map[f, a + b + c]
Out[4]= f[a] + f[b] + f[c]
```

将 Sqrt 作用于 g 的每个变量.

```
In[5]:= Map[Sqrt, g[x^2, x^3]]
```

Out[5]= $g\left[\sqrt{x^2}, \sqrt{x^3}\right]$

Map[*f*, *expr*] 将 *f* 作用于表达式 *expr* 第一层的项. 用 MapAll[*f*, *expr*] 可以将 *f* 作用于表达式 *expr* 的*所有* 项.

定义**2x2**矩阵 m.

```
In[6]:= m = {{a, b}, {c, d}}
```

```
Out[6]= {{a, b}, {c, d}}
```

Map 将 f 作用于 m 的第一层，即矩阵各行.

```
In[7]:= Map[f, m]
```

```
Out[7]= {f[{a, b}], f[{c, d}]}
```

MapAll 将 f 作用于 m 的所有元素上. 如果仔细查看该表达式，将看到 f 在每一项展开.

```
In[8]:= MapAll[f, m]
```

```
Out[8]= f[{f[{f[a], f[b]}], f[{f[c], f[d]}]}]
```

一般说来，如 "表达式的层次结构" 中讨论的，可以通过指定层的方法用 Map 将函数作用于表达式的一些项上.

将 f 作用于 m 的第2层元素上.

```
In[9]:= Map[f, m, {2}]
Out[9]= {{f[a], f[b]}, {f[c], f[d]}}
```

设定 Heads -> True 可将 f 作用于每一项的头部及其元素上.

```
In[10]:= Map[f, m, Heads -> True]
Out[10]= f[List][f[{a, b}], f[{c, d}]]
```

Map[*f*, *expr*] 或 *f* /@ *expr*	将 *f* 作用于表达式 *expr* 的第一层
MapAll[*f*, *expr*] 或 *f* //@ *expr*	将 *f* 作用于表达式 *expr* 的所有项
Map[*f*, *expr*, *lev*]	将 *f* 作用于表达式 *expr* 的第 *lev* 层的每项

将函数作用于表达式不同部分的方法.

可以通过指定层的方法用 Map 将函数作用于表达式的一些项上. 通过指定层次可以使函数作用于表达式的某些项上，用MapAt 可以通过列出元素标号使函数作用于表达式的某些项上，请参见 "表达式的项".

定义2x3矩阵.

```
In[11]:= mm = {{a, b, c}, {b, c, d}}
Out[11]= {{a, b, c}, {b, c, d}}
```

将 f 作用于 {1, 2} 和 {2, 3} 的项上.

```
In[12]:= MapAt[f, mm, {{1, 2}, {2, 3}}]
Out[12]= {{a, f[b], c}, {b, c, f[d]}}
```

给出 b 在 mm 中出现的位置.

```
In[13]:= Position[mm, b]
Out[13]= {{1, 2}, {2, 1}}
```

可以将 Position 得到的值代入 MapAt 中去.

```
In[14]:= MapAt[f, mm, %]
Out[14]= {{a, f[b], c}, {f[b], c, d}}
```

为了避免混淆，即使在仅有一个下标时，也应该以列表的形式指定每一项.

```
In[15]:= MapAt[f, {a, b, c, d}, {{2}, {3}}]
Out[15]= {a, f[b], f[c], d}
```

MapAt[*f*, *expr*, {*part*$_1$, *part*$_2$, …}] 将 *f* 作用于 *expr* 中指定的项

将函数作用于表达式中指定的项.

定义表达式.

```
In[16]:= t = 1 + (3 + x)^2 / x
```

$$\text{Out[16]= } 1 + \frac{(3 + x)^2}{x}$$

给出 t 的完全格式.

```
In[17]:= FullForm[t]
```

```
t[17]//FullForm=
Plus[1,
 Times[Power[x, -1], Power[Plus[3, x], 2]]]
```

MapAt 可用在任何表达式中. 注意，其中指定项的索引是基于表达式的完全格式.

```
In[18]:= MapAt[f, t, {{2, 1, 1}, {2, 2}}]
```

$$\text{Out[18]=}\quad 1+\frac{f\left[(3+x)^2\right]}{f[x]}$$

MapIndexed[*f*, *expr*]	将 *f* 作用于表达式的元素，*f* 的第二个变量给出表达式的每个元素项的位置
MapIndexed[*f*, *expr*, *lev*]	将 *f* 作用于表达式指定层的元素，*f* 的第二个变量给出表达式中每项的指标号

函数作用于表达式中的项和其标号上.

f 作用于一个列表的每个元素上，其第2个自变量给出元素的标号 f.

```
In[19]:= MapIndexed[f, {a, b, c}]
Out[19]= {f[a, {1}], f[b, {2}], f[c, {3}]}
```

f 作用于矩阵的所有层.

```
In[20]:= MapIndexed[f, {{a, b}, {c, d}}, 2]

Out[20]= {f[{f[a, {1, 1}], f[b, {1, 2}]}, {1}],
          f[{f[c, {2, 1}], f[d, {2, 2}]}, {2}]}
```

Map 把一元函数作用于一个表达式的项上；MapThread 将多元函数作用于多个表达式.

MapThread[*f*, {$expr_1, expr_2, \ldots$}]	将 *f* 作用于每个表达式 $expr_i$ 中对应的项
MapThread[*f*, {$expr_1, expr_2, \ldots$}, *lev*]	将 *f* 作用于 $expr_i$ 的指定层

同时将函数作用于多个表达式.

f 作用于相应的元素.

```
In[21]:= MapThread[f, {{a, b, c}, {ap, bp, cp}}]

Out[21]= {f[a, ap], f[b, bp], f[c, cp]}
```

MapThread 可对结构相同的任意长度的表达式进行操作.

```
In[22]:= MapThread[f, {{a, b}, {ap, bp}, {app, bpp}}]

Out[22]= {f[a, ap, app], f[b, bp, bpp]}
```

Map 等函数可以通过项的修改产生表达式，但有时不需要产生新的表达式，仅需要查看某些表达式，或者仅对表达式中的某些项进行运算. 一个典型的情况是当你所运用的函数具有某些“副作用”时，如进行赋值或产生输出时.

Scan[*f*, *expr*]	求出 *f* 作用于 *expr* 中各项时的函数值
Scan[*f*, *expr*, *lev*]	求出 *f* 作用中 *expr* 中第 *lev* 层项时的函数值

计算表达式中各项的函数值.

用 Map 将 f 作用于一个列表而产生新的列表.

```
In[23]:= Map[f, {a, b, c}]
Out[23]= {f[a], f[b], f[c]}
```

Scan 计算将函数作用于每一个元素的结果，但不产生新的表达式.

```
In[24]:= Scan[Print, {a, b, c}]
a
b
c
```

Scan 从最低层开始访问表达式中的所有项.

```
In[25]:= Scan[Print, 1 + x^2, Infinity]
1
x
2
x^2
```

相关教程

- 函数操作
- **Core Language**

纯函数

`Function[x, body]`	纯函数中的 x 可用任何变量代替
`Function[{x1, x2, ...}, body]`	多变量的纯函数
body&	自变量为 ⌗ 或 ⌗1、⌗2、⌗3 等的纯函数.

纯函数.

使用 `Nest` 和 `Map` 等函数运算时，总需要有一个确定函数名的函数去作用. 纯函数可直接作用于变量，不需要函数名.

定义函数 h.

```
In[1]:= h[x_] := f[x] + g[x]
```

`Map` 中使用刚定义的函数名 h .

```
In[2]:= Map[h, {a, b, c}]
Out[2]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

用纯函数得到相同的结果.

```
In[3]:= Map[f[#] + g[#] &, {a, b, c}]
Out[3]= {f[a] + g[a], f[b] + g[b], f[c] + g[c]}
```

Mathematica 中的纯函数有多种形式. 最理想的情况是定义一个目

函数 *fun* 在 *a* 的值.

求平方的纯函数.

```
In[4]:= Function[x, x^2]
```

$$\text{Out[4]= Function}\left[x, x^2\right]$$

计算 n 平方的值.

```
In[5]:= %[n]
```

$$\text{Out[5]= } n^2$$

可以用纯函数名调用纯函数.

在 Map 中使用纯函数.

```
In[6]:= Map[Function[x, x^2], a + b + c]
```

$$\text{Out[6]= } a^2 + b^2 + c^2$$

在嵌套 Nest 中用纯函数.

```
In[7]:= Nest[Function[q, 1 / (1 + q)], x, 3]
```

$$\text{Out[7]= } \frac{1}{1 + \frac{1}{1 + \frac{1}{1+x}}}$$

定义一个纯函数，并求出作用于 a 和 b 的值.

```
In[8]:= Function[{x, y}, x^2 + y^3][a, b]
```

$$\text{Out[8]= } a^2 + b^3$$

当需要重复使用一个函数时，可以先用 f[...] = ... 定义函

数，然后再使用名字 f 调用它. 而当仅使用函数一次时，用纯函数就比较方便.

熟悉形式逻辑和**LISP**编程语言的人会体会到 ***Mathematica*** 中的纯函数与 λ 表达式或无名函数相似，纯函数也类似于数学中的运算符.

⌗	纯函数中的第一个变量
⌗ *n*	纯函数中的第 *n* 个变量
⌗⌗	纯函数中的所有变量列
⌗⌗ *n*	纯函数中从 *n* 个变量开始的变量列

纯函数的简单形式.

正如如果你不想再次提到该函数的话，函数名是无关的，因此同样地，一个纯函数中的变量名也是无关的. ***Mathematica*** 允许用户不使用纯函数变量的显式名字，另一方面，我们可以通过给出“位置数字” ⌗ *n* 来指明变量. 在一个 ***Mathematica*** 纯函数中，⌗ *n* 表示所提供的第 *n* 个变量. ⌗ 表示第一个变量.

⌗^2 & 是求变量平方的纯函数的简化形式.

```
In[9]:= Map[#^2 &, a + b + c]
```

$$\text{Out[9]=}\quad a^2 + b^2 + c^2$$

不另外定义函数选择出每个列表中的前**2**个元素.

```
In[10]:= Map[Take[#, 2] &,
           {{2, 1, 7}, {4, 1, 5}, {3, 1, 2}}]

Out[10]= {{2, 1}, {4, 1}, {3, 1}}
```

利用纯函数的简化形式简化前面 "函数的重复调用" 中

```
In[11]:= fromdigits[digits_] :=
          Fold[(10 #1 + #2) &, 0, digits]
```

当使用纯函数的简化形式时，千万不要忘记 **&** 号，否则 ***Mathematica*** 就无法理解和执行这一输入.

在纯函数中使用 **&** 符号时，要注意 **&** 的优先级很低，必要时要用括号，如 "运算符的输入形式" 中讨论的. 这意味着，你可以不用括号输入形如 ⌗1 + ⌗2 & 的表达式. 另一方面，如果你愿意对纯函数设置选项话，必须使用括号. 例如， *option* -> (*fun* &).

在 ***Mathematica*** 中，纯函数可以选择任何数目的变量. ⌗⌗ 表示给定的任何变量，⌗⌗ *n* 表示从第 *n* 项开始的所有变量.

⌗⌗ 代表所有变量.

```
In[12]:= f[##, ##] &[x, y]
Out[12]= f[x, y, x, y]
```

⌗⌗2 表示除第一项以外的所有变量.

```
In[13]:= Apply[f[##2, #1] &, {{a, b, c}, {ap, bp}}, {1}]
Out[13]= {f[b, c, a], f[bp, ap]}
```

相关教程

- 函数操作

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

由函数产生列表

`Array[f,n]`	产生形如 {f[1], f[2], ...} 的长度为 n 的列表
`Array[f,{n1,n2,...}]`	产生 $n_1 \times n_2 \times \ldots$ 的嵌套列表，每个元素是 f 作用于它的标号上得到的
`NestList[f,x,n]`	产生形如 {x, f[x], f[f[x]], ...} 的列表，其中 f 迭代 n 次为止
`FoldList[f, x,{a,b,...}]`	产生列表 {x, f[x, a], f[f[x, a], b], ...}
`ComposeList[ {f1,f2,...},x]`	产生列表 {x, f1[x], f2[f1[x]], ...}

由函数产生列表.

构造**5**个元素，每个元素为 p[*i*].

```
In[1]:= Array[p, 5]
Out[1]= {p[1], p[2], p[3], p[4], p[5]}
```

产生同一列表的又一种方法.

```
In[2]:= Table[p[i], {i, 5}]
Out[2]= {p[1], p[2], p[3], p[4], p[5]}
```

产生元素为 $i+i^2$ 的表列.

```
In[3]:= Array[# + #^2 &, 5]
Out[3]= {2, 6, 12, 20, 30}
```

产生元素为 m[*i*, *j*] 的 **2×3** 矩阵.

```
In[4]:= Array[m, {2, 3}]
Out[4]= {{m[1, 1], m[1, 2], m[1, 3]},
         {m[2, 1], m[2, 2], m[2, 3]}}
```

产生元素为下标和平方的 **3×3** 矩阵.

```
In[5]:= Array[Plus[##]^2 &, {3, 3}]
Out[5]= {{4, 9, 16}, {9, 16, 25}, {16, 25, 36}}
```

NestList 和 FoldList 曾经在 "函数的重复调用" 中描述过. 尤其在将它们与纯函数结合的时候，可以构造一些非常有效的 ***Mathematica*** 程序.

对 x^n 关于 x 逐次求导得到表列.

```
In[6]:= NestList[D[#, x] &, x^n, 3]
Out[6]= {x^n, n x^(-1+n), (-1 + n) n x^(-2+n),
         (-2 + n) (-1 + n) n x^(-3+n)}
```

用函数选择表达式的项

"处理列表元素" 介绍了如何根据*位置* 在列表中选取元素. 经常也需要根据内容来选取元素，而不是根据位置选取.

`Select`[*list*, *f*] 选择 *list* 的元素以 *f* 为判据. `Select` 将 *f* 依次用于 *list* 的每个元素，并且只保留结果为 `True` 的元素.

这里选取令纯函数生成 `True` 的列表元素，即数值大于**4**.

```
In[1]:= Select[{2, 15, 1, a, 16, 17}, # > 4 &]
Out[1]= {15, 16, 17}
```

可以使用 `Select` 选取任意表达式的项，不仅仅局限于列表的元素.

给出包含 x，y 和 z 的项的和.

```
In[2]:= t = Expand[(x + y + z)^2]
Out[2]= x^2 + 2 x y + y^2 + 2 x z + 2 y z + z^2
```

可以使用 Select 从和式中选取不包含符号 x 的项.

```
In[3]:= Select[t, FreeQ[#, x] &]
Out[3]= y^2 + 2 y z + z^2
```

Select[*expr*, *f*]	选择 *expr* 中 *f* 值为 True 的项
Select[*expr*, *f*, *n*]	选择 *expr* 中前 *n* 个 *f* 值为 True 的项

选择表达式中的项.

"限制模式" 节介绍的一些“判断词”常常用来作为 Select 的判据.

此语句给出了满足选择条件的第一个元素.

```
In[4]:= Select[{-1, 3, 10, 12, 14}, # > 3 &, 1]
Out[4]= {10}
```

相关教程

- 函数操作

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

具有非符号头部的表达式

大部分情况下，我们希望像 $f[x]$ 这样的表达式的头部 f 是一个简单符号，但具有非符号头部表达式也有一些重要应用.

f[3] 是表达式的头部. 可以用这样的头部表示指定的函数.

```
In[1]:= f[3][x, y]
Out[1]= f[3][x, y]
```

可以用任何表达式作头部. 但必须放在括号内.

```
In[2]:= (a + b)[x]
Out[2]= (a + b)[x]
```

将一个复杂的表达式用作头部的情况在 "纯函数" 节已遇到. 将 Function[*vars*, *body*] 作为表达式的头部时就定义了对变量求值的函数.

头部为 Function[x, x^2] 的表达式的值是其变量的平方.

```
In[3]:= Function[x, x^2][a + b]
Out[3]= (a + b)^2
```

在 ***Mathematica*** 中有一些类似于纯函数的结构，它们用来表示数值函数等一些特殊的函数. 在所有情况下，基本机制是给出能够包含所需要的函数完整信息的头部.

Function[*vars*, *body*][*args*]	纯函数
InterpolatingFunction[*data*][*args*]	由 Interpolation 和 NDSolve 产生的近似数值函数
CompiledFunction[*data*][*args*]	由 Compile 产生的编译数值函数
LinearSolveFunction[*data*][*vec*]	由 LinearSolve 产生的矩阵解函数

一些有非符号头部的表达式.

NDSolve 返回一系列规则，这些规则将 y 作为 InterpolatingFunction 的目标.

```
In[4]:= NDSolve[{y''[x] == y[x], y[0] == y'[0] == 1},
          y, {x, 0, 5}]
Out[4]= {{y → InterpolatingFunction[{{0., 5.}}, <>]}}
```

这是 InterpolatingFunction 的目标.

```
In[5]:= y /. First[%]
Out[5]= InterpolatingFunction[{{0., 5.}}, <>]
```

将 InterpolatingFunction 目标作为头部去得到函数 y 的数值近似.

```
In[6]:= %[3.8]
Out[6]= 44.7012
```

复杂表达式作为头部的另一个重要应用是进行泛函运算.

考虑一个求导运算的例子，在 "导数的表示" 节将要讨论的表达式 f' 求函数 f 的导数，在 ***Mathematica*** 中 f' 用 Derivative[1][f] 表示，“函数操作符” Derivative[1] 作用于 f 后给出另一个函数 f'.

这个表达式的头部表示将功能运算 Derivative[1] 作用于函数 f.

```
In[7]:= f'[x] // FullForm
Out[7]//FullForm=
Derivative[1][f][x]
```

可以用 fp 替代头部 f' 表示求导运算.

```
In[8]:= % /. f' -> fp
Out[8]= fp[x]
```

相关教程

- 函数操作

教程专集

算子运算

可以将表达式 $f[x]$ 理解为算子 f 作用于表达式 x 上. $f[g[x]]$ 可以理解为算子 f 和 g 复合作用于 x 上的结果.

`Composition[f,g,…]`	函数 $f, g, \ldots$ 的复合
`InverseFunction[f]`	f 的反函数
`Identity`	恒等函数

一些函数运算.

f、g 和 h 的复合.

```
In[1]:= Composition[f, g, h]
Out[1]= Composition[f, g, h]
```

可以对复合函数进行符号运算.

```
In[2]:= InverseFunction[Composition[%, q]]
Out[2]= Composition[q^(-1), h^(-1), g^(-1), f^(-1)]
```

当指定了一个变量时，就可以明确地给出复合的值.

```
In[3]:= %[x]
Out[3]= q^(-1)[h^(-1)[g^(-1)[f^(-1)[x]]]]
```

子运算中也可以考虑算子的和.

算子 f 和 g 的和.

```
In[4]:= (f + g)[x]
Out[4]= (f + g)[x]
```

Through 可以使上面的表达式更加明确.

```
In[5]:= Through[%, Plus]
Out[5]= f[x] + g[x]
```

算子 $1+\frac{\partial}{\partial x}$.

```
In[6]:= Identity + (D[#, x] &)
Out[6]= Identity + (∂x#1 &)
```

Mathematica 不会主动将算子的几项作用于一个表达式.

```
In[7]:= %[x^2]
Out[7]= (Identity + (∂x#1 &))[x^2]
```

用 Through 去作用这一算子.

```
In[8]:= Through[%, Plus]
Out[8]= 2 x + x^2
```

`Identity[expr]`	恒等函数
`Through[p[f1,f2][x],q]`	当 p 与 q 相同时给出 $p[f_1[x], f_2[x]]$
`Operate[p,f[x]]`	得到 $p[f][x]$
`Operate[p,f[x],n]`	在 f 的第 n 层运用 p
`MapAll[p,expr, Heads->True]`	将 p 作用于 *expr* 的头部和所有项

算子运算

该表达式有一个复杂的头部.

```
In[9]:= t = ((1 + a) (1 + b))[x]
Out[9]= ((1 + a) (1 + b))[x]
```

像 Expand 这样的函数不会自动作用于表达式的头部.

```
In[10]:= Expand[%]
Out[10]= ((1 + a) (1 + b))[x]
```

当可选项 Heads 设定为 True 时, MapAll 作用于头部之内.

```
In[11]:= MapAll[Expand, t, Heads -> True]
Out[11]= (1 + a + b + a b)[x]
```

替代运算 /. 进入表达式的头部.

```
In[12]:= t /. a -> 1
Out[12]= (2 (1 + b))[x]
```

可以用 Operate 将某一函数作用于表达式的头部.

```
In[13]:= Operate[p, t]
Out[13]= p[(1 + a)(1 + b)][x]
```

相关教程

- 函数操作

教程专集

- **Core Language**

结构的操作

Mathematica 有改变表达式结构的有效功能. 这可以使我们实现合并、分配等数学特性，并对一些简洁和有效的程序提供基础.

这里我们描述各种对表达式进行的操作. **"属性"** 介绍一些操作如何自动在所有具有特定头部并且该头部被赋以适合的属性的表达式上进行.

Mathematica 函数 Sort[*expr*] 不仅排列列表元素，也对具有任意头部的表达式排序. 用这种方式，我们可以对任意函数实现结合特性或对称性.

Sort 将函数的变量按序排列.

```
In[1]:= Sort[f[c, a, b]]
Out[1]= f[a, b, c]
```

Sort[*expr*]	按照一种标准顺序对列表或表达式的元素分类
Sort[*expr*, *pred*]	用 *pred* 函数决定是否按顺序排列
Ordering[*expr*]	当排好序时，给出元素的顺序
Ordering[*expr*, *n*]	当排好序时，给出前 *n* 个元素的顺序
Ordering[*expr*, *n*, *pred*]	使用函数 *pred* 来决定是否每对都是排好序的
OrderedQ[*expr*]	当表达式 *expr* 的元素按标准顺序排列时为 True，否则为 False
Order[$expr_1$, $expr_2$]	当 $expr_1$ 按标准顺序出现在 $expr_2$ 前时取值 1，相反时取值 –1

按顺序分类.

Sort 的第二个变量是用于决定是否每对都是排好序的函数. 排序按递减顺序进行.

```
In[2]:= Sort[{5, 1, 8, 2}, (#2 < #1) &]
Out[2]= {8, 5, 2, 1}
```

根据不含 x 的元素比含 x 的元素靠前的次序排列.

```
In[3]:= Sort[{x^2, y, x + y, y - 2}, FreeQ[#1, x] &]
Out[3]= {y, -2 + y, x + y, x^2}
```

`Flatten[expr]`	压平与 *expr* 具有相同头部的嵌套函数
`Flatten[expr, n]`	最多取消 *n* 次嵌套
`Flatten[expr, n, h]`	撤销以 *h* 为头部的函数的结构
`FlattenAt[expr, i]`	撤销表达式 *expr* 中的第 *i* 层结构

压平表达式的结构.

`Flatten` 压平函数嵌套.

```
In[4]:= Flatten[f[a, f[b, c], f[f[d]]]]
Out[4]= f[a, b, c, d]
```

用 `Flatten` 将元素拼接为列表或其它表达式.

```
In[5]:= Flatten[{a, f[b, c], f[a, b, d]}, 1, f]
Out[5]= {a, b, c, a, b, d}
```

可以使用 `Flatten` 进行组合. 函数 `Distribute` 允许你实施特性, 例如, 分配特性和线性.

`Distribute[f[a+b+...,...]]`	将 f 分配到和式去得到 $f[a, \ldots] + f[b, \ldots] + \ldots$
`Distribute[f[args],g]`	将 f 分配到头部为 g 的变量
`Distribute[expr,g,f]`	当头部为 f 时分配
`Distribute[expr,g,f,gp,fp]`	将 f 分配到 g，并分别用 fp 和 gp 代替它们

分配律.

f 分配到 a + b.

```
In[6]:= Distribute[f[a + b]]
Out[6]= f[a] + f[b]
```

更复杂的一些例子.

```
In[7]:= Distribute[f[a + b, c + d]]
Out[7]= f[a, c] + f[a, d] + f[b, c] + f[b, d]
```

一般地，f 分配到和式 `Plus` 时，是将如 $f[a + b]$ 的表达式展开为 $f[a] + f[b]$. 函数 `Expand` 对标准的代数乘法如`Times` 进行类似的分配展开. `Distribute` 可对任意运算进行类似的分配展开.

`Expand` 实施乘法 `Times` 对加法 `Plus` 的分配.

```
In[8]:= Expand[(a + b) (c + d)]
Out[8]= a c + b c + a d + b d
```

对一个列表，而非和式，进行分配. 结果包含所有可能的变量对.

```
In[9]:= Distribute[f[{a, b}, {c, d}], List]
Out[9]= {f[a, c], f[a, d], f[b, c], f[b, d]}
```

当表达式的头部为 f 时，对其进行分配.

```
In[10]:= Distribute[f[{a, b}, {c, d}], List, f]
Out[10]= {f[a, c], f[a, d], f[b, c], f[b, d]}
```

当头部是 f 时，对列表进行分配. 结果，用 gp 代替列表 List，用 fp 代替 f.

```
In[11]:= Distribute[f[{a, b}, {c, d}], List, f, gp, fp]
Out[11]= gp[fp[a, c], fp[a, d], fp[b, c], fp[b, d]]
```

与 Distribute 相关的是函数 Thread. Thread 将函数并行地作用于一个列表或表达式的所有项.

Thread[f[{a_1, a_2}, {b_1, b_2}]]	将 f 线性作用于列表给出 {f[a_1, b_1], f[a_2, b_2]}
Thread[f[$args$], g]	将 f 线性作用于头部为 g 的变量 $args$ 上

线性作用于表达式的函数.

变量是列表的函数.

```
In[12]:= f[{a1, a2}, {b1, b2}]
Out[12]= f[{a1, a2}, {b1, b2}]
```

Thread 线性作用于列表的元素.

```
In[13]:= Thread[%]
Out[13]= {f[a1, b1], f[a2, b2]}
```

不在列表中的元素将重复.

```
In[14]:= Thread[f[{a1, a2}, {b1, b2}, c, d]]
Out[14]= {f[a1, b1, c, d], f[a2, b2, c, d]}
```

在 "将一些对象放在一起" 中提到，并且在 "属性" 中有更详细的讨论，***Mathematica*** 的许多内置函数具有“可列表”的性质，它们可以自动进行线性作用.

内置函数，比如 Log 是“可列表”的，因此它们可以自动逐项作用于列表.

```
In[15]:= Log[{a, b, c}]
Out[15]= {Log[a], Log[b], Log[c]}
```

但 Log 不能自动对方程进行线性作用.

```
In[16]:= Log[x == y]
Out[16]= Log[x == y]
```

可以用 Thread 使函数作用于方程两端.

```
In[17]:= Thread[%, Equal]
Out[17]= Log[x] == Log[y]
```

Outer[f, $list_1$, $list_2$]	广义外积
Inner[f, $list_1$, $list_2$, g]	广义内积

广义内积和外积.

Outer[f, $list_1$, $list_2$] 给出 $list_1$ 和 $list_2$ 中元素所有可能的组合，并将 f 作用于这些组合. Outer 可以看作是张量的积的推广，如 "张量" 中讨论的.

Outer 给出了元素的所有组合，并用 f 作用.

```
In[18]:= Outer[f, {a, b}, {1, 2, 3}]

Out[18]= {{f[a, 1], f[a, 2], f[a, 3]},
          {f[b, 1], f[b, 2], f[b, 3]}}
```

这里 Outer 产生一个下三角 **Boolean** 矩阵.

```
In[19]:= Outer[Greater, {1, 2, 3}, {1, 2, 3}]

Out[19]= {{False, False, False},
          {True, False, False}, {True, True, False}}
```

Outer 可用于任何头部相同的表达式.

```
In[20]:= Outer[g, f[a, b], f[c, d]]

Out[20]= f[f[g[a, c], g[a, d]], f[g[b, c], g[b, d]]]
```

像 Distribute 一样，Outer 给出元素所有可能的组合，而 Inner 像 Thread 一样仅给出表达式中相应位置元素的组合.

由 Inner 形成的结构.

```
In[21]:= Inner[f, {a, b}, {c, d}, g]
Out[21]= g[f[a, c], f[b, d]]
```

Inner 是内积 Dot 的推广.

```
In[22]:= Inner[Times, {a, b}, {c, d}, Plus]
Out[22]= a c + b d
```

相关教程

- 函数操作

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: Programming in *Mathematica***

序列

函数 Flatten 撤销所有的子列表.

```
In[1]:= Flatten[{a, {b, c}, {d, e}}]
Out[1]= {a, b, c, d, e}
```

FlattenAt 撤销指定的子列表结构.

```
In[2]:= FlattenAt[{a, {b, c}, {d, e}}, 2]
Out[2]= {a, b, c, {d, e}}
```

Sequence 序列对象中的元素自动进行拼接，而不要求任何显式的压平操作.

```
In[3]:= {a, Sequence[b, c], Sequence[d, e]}
Out[3]= {a, b, c, d, e}
```

Sequence[e_1, e_2, ...]　　序列的元素自动拼接到函数

函数变元中序列的表示.

Sequence 用于各种函数.

```
In[4]:= f[Sequence[a, b], c]
Out[4]= f[a, b, c]
```

用于有特殊输入格式的函数.

```
In[5]:= a == Sequence[b, c]
Out[5]= a == b == c
```

Sequence 的常用方式.

```
In[6]:= {a, b, f[x, y], g[w], f[z, y]} /. f -> Sequence
Out[6]= {a, b, x, y, g[w], z, y}
```

模块化和事物的命名

模块和局部变量

局部常量

模块工作方式

纯函数和规则中的变量

数学中的哑元

块和局部值

块与模块的比较

上下文

上下文和程序包

Mathematica 程序包

建立 ***Mathematica*** 程序包

程序包中的文件

包的自动调入

模块和局部变量

Mathematica 一般假设变量是*全局* 变量. 即每次使用 x 等名字时，***Mathematica*** 总认为在调用*同*一对象.

然而在编程时，不需要将所有变量都作为全局变量. 例如，在两个不同的程序中，x 可用来指代两个不同的变量. 此时，每个程序中的 x 都必须作为*局部* 变量.

在 ***Mathematica*** 中可以用 ***modules*** 定义局部变量. 在每个模块中，可以给出模块中涉及的局部变量列表.

`Module[{x, y, ...}, body]`	具有局部变量 x，y，… 的模块

在 ***Mathematica*** 中产生模块

定义全局变量 t，其值为**17.**

```
In[1]:= t = 17
Out[1]= 17
```

这里 t 在模块内，所以它的处理与全局变量 t 无关.

```
In[2]:= Module[{t}, t = 8; Print[t]]
        8
```

全局变量 t 的值还是17.

```
In[3]:= t
Out[3]= 17
```

模块的最常用方法是在自定义函数中建立临时或中间变量. 一定要保证这些变量只是局部的，否则当这些变量名与其它变量名重合成就会引起麻烦.

中间变量 t 是模块中的局部变量.

```
In[4]:= f[v_] := Module[{t}, t = (1 + v)^2;
          t = Expand[t]]
```

运行函数 f.

```
In[5]:= f[a + b]
Out[5]= 1 + 2 a + a^2 + 2 b + 2 a b + b^2
```

全局变量 t 的值还是17.

```
In[6]:= t
Out[6]= 17
```

在模块中可以像处理其它符号一样处理局部变量. 例如，可以把它们作为局部函数的名字来用，可以对它们赋于属性，等等.

这里构造了一个建立局部函数 f 的模块.

```
In[7]:= gfac10[k_] := Module[{f, n}, f[1] = 1;
          f[n_] := k + n f[n - 1];
          f[10]]
```

此处局部函数 f 是一般的阶乘.

```
In[8]:= gfac10[0]

Out[8]= 3 628 800
```

这里，f 是一个推广了的阶乘.

```
In[9]:= gfac10[2]

Out[9]= 8 841 802
```

在一个模块中定义局部变量时，***Mathematica*** 开始并不对它赋值. 即使在模块外定义了该变量的全局值，都能以纯符号的方式使用该变量.

这里使用了 t 在前面定义的全局值，故结果是一个数.

```
In[10]:= Expand[(1 + t)^3]

Out[10]= 5832
```

这里 Length 得到了变量的个数.

```
In[11]:= Length[Expand[(1 + t)^3]]

Out[11]= 0
```

由于局部变量 t 没有赋值，因此被当作符号处理，Expand 产生了一个预期的代数结果.

```
In[12]:= Module[{t}, Length[Expand[(1 + t)^3]]]
Out[12]= 4
```

Module[{x=x0, y=y0, ...}, body]　　局部变量有初始值的模块

对局部变量赋初值

这里指局部变量 t 的初始值为 u.

```
In[13]:= g[u_] := Module[{t = u}, t += t / (1 + u)]
```

这里使用了 g 的定义.

```
In[14]:= g[a]
Out[14]= a + a/(1 + a)
```

对模块中任何一个局部变量都可以定义初始值. 这些初始值总是在模块执行之前进行计算. 结果，即使定义了 x 是模块的局部变量，在赋初始值时可以用全局变量 x 的值.

u 的初始值为全局变量 t 的值.

```
In[15]:= Module[{t = 6, u = t}, u^2]
Out[15]= 289
```

lhs := Module [*vars*, *rhs* /; *cond*]	在 *rhs* 和 *cond* 中共用局部变量

在条件定义中使用局部变量

在定义 /; 时经常需要引入临时变量，并且定义的右端也需要使用这些临时变量. ***Mathematica*** 允许将定义的右端和条件包含在模块之中.

定义具有条件的函数.

```
In[16]:= h[x_] := Module[{t}, t^2 - 1 /; (t = x - 4) > 1]
```

Mathematica 在条件和右端项中共用局部变量 t 的值.

```
In[17]:= h[10]
Out[17]= 35
```

相关教程

- 模块化和事物的命名

教程专集

- **Core Language**

局部常量

```
With[                          定义局部常量 x，y，  …
  {x=x0,y=y0,…},
  body]
```

定义局部常量.

在 Module 中可以定义局部变量，这样我们可以对其赋值并且改变其值. 然而，通常我们所需要的是局部常量，对其我们仅需要赋值一次. ***Mathematica*** 中的 With 结构可以建立局部常量.

定义 t 的全局值.

```
In[1]:= t = 17
Out[1]= 17
```

定义把 t 作为局部常量的函数.

```
In[2]:= w[x_] := With[{t = x + 1}, t + t^3]
```

使用定义 w.

```
In[3]:= w[a]
Out[3]= 1 + a + (1 + a)^3
```

t 仍然具有它的全局值.

```
In[4]:= t
Out[4]= 17
```

与模块 Module 中的情况相似，用 With 定义的初始值在 With 执行之前进行计算.

给局部常量 t 赋值的表达式 t + 1 用 t 的全局值进行计算.

```
In[5]:= With[{t = t + 1}, t^2]
Out[5]= 324
```

With[{$x = x_0$, ...}, *body*] 的工作方式是取 *body*，并用 x_0 代替 x，等等. 可以将 With 理解为 /. 运算的推广，可适用于 ***Mathematica*** 的代码.

使用 a 替换 x.

```
In[6]:= With[{x = a}, x = 5]
Out[6]= 5
```

替代以后，With 的内容是 a = 5，故 a 得到了全局值 5.

```
In[7]:= a
Out[7]= 5
```

清除 a 的值.

```
In[8]:= Clear[a]
```

在某种意义上，With 类似于局部变量仅赋值一次的 Module 的特殊形式.

用 With 而不用 Module 的主要原因之一是使 ***Mathematica*** 程序容易理解. 在模块中的某一处遇到局部变量 x 时很可能需要跟踪整个模块的代码来得到 x 在该处的值. 而在 With 结构中，只需要观

有几个 With 结构时，总是某一个变量最里面的一个起作用. 可以将 Module 和 With 混用. 一般原则是某一变量最里面的一个起作用 .

在嵌套 With 结构中，总是最里面的一个起作用.

```
In[9]:= With[{t = 8}, With[{t = 9}, t^2]]
Out[9]= 81
```

可以将 Module 和 With 结构混用.

```
In[10]:= Module[{t = 8}, With[{t = 9}, t^2]]
Out[10]= 81
```

当名字不重合时，结构内的局部变量不屏蔽结构外的变量.

```
In[11]:= With[{t = a}, With[{u = b}, t + u]]
Out[11]= a + b
```

除了 x 和 *body* 在何时计算外，With[{$x = x_0$, ...}, *body*] 和 *body* /. $x \to x_0$ 的做法基本类似. 然而，当表达式 *body* 内部都包含 With 或 Module 结构时， With 显示特殊的状态. 主要的问题是防止不同 With 结构中的局部常量相互冲突，或与全局目标冲突. 有关细节将在 "模块工作方式" 节讨论.

在 With 内部的 y 被重新命名以防止它与全局变量 y 冲突.

```
In[12]:= With[{x = 2 + y}, Hold[With[{y = 4}, x + y]]]
Out[12]= Hold[With[{y$ = 4}, (2 + y) + y$]]
```

- 过程式编程

相关教程

- 模块化和事物的命名
- 在 **Dynamic** 或 **Manipulate** 内部计算表达式

教程专集

- **Core Language**

模块工作方式

Mathematica 中模块的基本工作方式非常简单. 任何模块每一次使用时，就产生一个新符号去代表它的每一个局部变量. 新符号的名字被唯一地给定，它不能跟任何其它名字冲突. 命名的方法是在给定的局部变量后加 $，并给出唯一的序号.

从全局变量 `$ModuleNumber` 的值可以找到序列号. 该变量计算 `Module` 的任何形式所使用的总次数.

> `Module` 中产生形如 *x$nnn* 的符号去代表每个局部变量.

Mathematica 中模块的基本原理.

这里说明了模块内所产生 `t` 的符号.

```
In[1]:= Module[{t}, Print[t]]
```

任何模块每一次运行时产生的符号不同.

```
In[2]:= Module[{t, u}, Print[t]; Print[u]]
        t$2
        u$2
```

在绝大部分情况下，不需要直接涉及模块内产生的实际符号.但在一个模块的执行过程中打开对话时就会看到这些符号.同样，用函数如 Trace 等也能观察模块的计算.

用 Trace 观察模块内部产生的符号.

```
In[3]:= Trace[Module[{t}, t = 3]]
Out[3]= {Module[{t}, t = 3], {t$3 = 3, 3}, 3}
```

在模块中打开对话.

```
In[4]:= Module[{t}, t = 6; Dialog[]]
```

在对话内看到为 t 等局部变量产生的符号.

```
In[5]:= Stack[_]
Out[5]= {Module[{t}, t = 6;
          Dialog[]], t$4 = 6;
         Dialog[], Dialog[]}
```

可以像其它符号一样来处理这些符号.

```
In[6]:= t$4 + 1
Out[6]= 7
```

从对话返回.

```
In[7]:= Return[t$4^2]
Out[7]= 36
```

在某些情况下，明确返回在模块中产生的符号是很方便的.

可以明确地返回在模块中产生的符号.

```
In[8]:= Module[{t}, t]
Out[8]= t$6
```

可以把这些符号像其它符号一样处理.

```
In[9]:= %^2 + 1
Out[9]= 1 + t$6^2
```

Unique[*x*]	产生形如 *x*$*nnn* 唯一名称的新符号
Unique[{*x*, *y*, ...}]	产生一个新符号表

通过唯一的名称产生新符号.

函数 Unique 与 Module 一样产生新符号. 每次调用 Unique 时，$ModuleNumber 增加，故可保证新符号的名称不重复.

产生唯一的名称以 x 开头的新符号.

```
In[10]:= Unique[x]
Out[10]= x$7
```

每调用一次 Unique 就得到一个序列号更大的符号.

```
In[11]:= {Unique[x], Unique[x], Unique[x]}
Out[11]= {x$8, x$9, x$10}
```

对一个集合调用 Unique 时，得到每个符号有相同的序列号.

```
In[12]:= Unique[{x, xa, xb}]
Out[12]= {x$11, xa$11, xb$11}
```

可以用标准 ***Mathematica*** ？*name* 机制得到在模块中或用函数 Unique 产生符号的信息.

执行这一模块产生符号 q$*nnn*.

```
In[13]:= Module[{q}, q^2 + 1]
Out[13]= 1 + q$12^2
```

这里可以看到所产生的符号.

```
In[14]:= ?q*
         q     q$12
```

模块 Module 所产生的符号与计算中的符号性能相同. 但这些符号具有 Temporary 属性，当不再使用时它们会被系统删除，所以在模块内产生的符号当模块执行完时就被删除. 那些明确返回的符号才能继续存在.

这里显示了在模块内产生的新变量 q.

```
In[15]:= Module[{q}, Print[q]]
         q$13
```

这个新变量在模块执行结束时被删除，所以这里不再出现.

```
In[16]:= ?q*

         q      q$12
```

应该意识到对所产生的符号用 *x*$*nnn* 等符号名完全是一种约定. 一般地，可以对任何符号用这类符号，但这样做时可能会与模式 Module 所产生的符号重合.

重要的一点是由模式 Module 所产生的符号的唯一性仅仅在 ***Mathematica*** 的一个进程中是唯一的. 决定符合序列数的变量 $ModuleNumber 在每个进程的开始时总是重新设置.

这意味着将含有所产生符号的表达式存在一个文件中，然后在另一个进程中打开该文件时无法保证不冲突.

避免这种冲突的一个途径是在每个进程的开始时直接设置不同的 $ModuleNumber，特别，当设置 $ModuleNumber = 10^10 $SessionID 时就可以避免任何冲突. 全局变量 $SessionID 给出刻划特定***Mathematica*** 进程的唯一数. 这个变量的值由日期和时间、所用计算的**ID**、以及特定 ***Mathematica*** 过程的**ID**等决定.

$ModuleNumber	由 Module 和 Unique 所产生符号的序列号
$SessionID	在每个 ***Mathematica*** 进程中都不重复的数

决定所产生符号序列号的变量.

在产生了决定局部变量的符号后，Module[*vars*, *body*] 就用这些

With 将每个局部变量的名称用所产生的符号代替，然后模块 Module 来计算结果中表达式的值.

重要的是 Module[*vars*, *body*] 仅将所产生的符号代入 *body* 的实际表达式中. 但它不把这些符号代入 *body* 所调用的但不直接出现在 *body* 中的代码中.

"块和局部值" 节将讨论如何用 Block 去建立以不同方式工作的"局部值".

由于 x 没有直接出现在模块的**body**中，故没有它的局部值.

```
In[17]:= tmp = x^2 + 1; Module[{x = 4}, tmp]
Out[17]= 1 + x^2
```

大部分情况下，通过 Module[*vars*, *body*] 的直接输入来在 ***Mathematica*** 中建立模块. 由于 Module 函数具有属性 HoldAll，所以在模块执行之前 *body* 将维持不计算的状态.

在 ***Mathematica*** 中还可以建立动态模块. 新符号的产生以及向 *body* 中的代入总是在模块执行时进行，而不是在模块作为 ***Mathematica*** 输入建立时进行.

这里立即计算模块的内部，直接给出 x 的值.

```
In[18]:= tmp = x^2 + 1; Module[{x = 4}, Evaluate[tmp]]
Out[18]= 17
```

相关教程

- 模块化和事物的命名
- 基本的内部结构

纯函数和规则中的变量

用 Module 和 With 可以给出作为局部处理的符号名序列. 但有时需要直接将某些变量名进行局部处理.

例如，在使用 Function[{x}, x + a] 等纯函数时，x 是一个“形式参数”，这个名称是局部的. 在规则 f[x_] -> x^2 或定义 f[x_] := x^2 中出现的 x 也是这样.

Mathematica 用统一的方法去保证在纯函数和规则中出现的形式参数是局部的，且不与全局变量混淆. 其基本的思想是必要时用形如 *x*$ 的符号去代替形式参数. 作为一个约定，*x*$ 从不用作全局变量名.

这里是一个纯函数的嵌套.

```
In[1]:= Function[{x}, Function[{y}, x + y]]
Out[1]= Function[{x}, Function[{y}, x + y]]
```

Mathematica 重新命名函数内的形式参数 y 以避免与全局对象 y 冲突.

```
In[2]:= %[2 y]
Out[2]= Function[{y$}, 2 y + y$]
```

这样得到的纯函数与所期望的一样.

```
In[3]:= %[a]
Out[3]= a + 2 y
```

一般来说，在对象如 `Function[vars, body]` 中的形式参数当另一个纯函数修改了 *body* 时 ***Mathematica*** 就要重新命名.

由于纯函数内发生了变化，形式参数 y 就被重新命名.

```
In[4]:= Function[{x}, Function[{y}, x + y]][a]
Out[4]= Function[{y$}, a + y$]
```

当函数内部没有变化时，形式参数就不重新命名.

```
In[5]:= Function[{x}, x + Function[{y}, y^2]][a]
Out[5]= a + Function[{y}, y^2]
```

Mathematica 在对纯函数中的形式参数重命名时比较自由. 原则上，函数中的形式参数与代换到纯函数中表达式的项不冲突时可以不用重新命名. 但为了一致起见，在这种情况下，***Mathematica*** 还是对形式参数重新命名.

这时，在函数内的形式参数 x 屏蔽了函数体，故不需要进行重命名.

```
In[6]:= Function[{x}, Function[{x}, x + y]][a]
Out[6]= Function[{x}, x + y]
```

这里是三重函数嵌套.

```
In[7]:= Function[{x},
         Function[{y}, Function[{z}, x + y + z]]]

Out[7]= Function[{x},
         Function[{y}, Function[{z}, x + y + z]]]
```

这种情况下两个内层函数都重新命名.

```
In[8]:= %[a]

Out[8]= Function[{y$}, Function[{z$}, a + y$ + z$]]
```

正如 "纯函数" 所述，***Mathematica*** 中纯函数类似于形式逻辑中的 λ 表达式. 对形式参数重新命名使 ***Mathematica*** 纯函数再次产生标准 λ 表达式的所有语法.

`Function[{x, ...}, body]`	局部参数
lhs`:>`*rhs* 和 *lhs*`:=`*rhs*	局部模式名
lhs`=`*rhs* 和 *lhs*`:=`*rhs*	局部模式名
`With[{`$x=x_0, \ldots$`}, body]`	局部常数
`Module[{x, ...}, body]`	局部变量

Mathematica 中的定界结构.

Mathematica 中有一些“定界结构”，其中某些名称作为局部变量处理，当这些结构混合时，***Mathematica*** 进行适当的重命名以避免冲突.

Mathematica 重新命名纯函数中的形式参数以避免冲突.

```
In[9]:= With[{x = a}, Function[{a}, a + x]]
Out[9]= Function[{a$}, a$ + a]
```

这里 `With` 内的局部参数被重新命名以避免冲突.

```
In[10]:= With[{x = y}, Hold[With[{y = 4}, x + y]]]
Out[10]= Hold[With[{y$ = 4}, y + y$]]
```

这种情况下变量名之间没有冲突，故没有进行重新命名.

```
In[11]:= With[{x = y}, Hold[With[{z = x + 2}, z + 2]]]
Out[11]= Hold[With[{z = y + 2}, z + 2]]
```

在模块中局部变量 y 重新命名以避免冲突.

```
In[12]:= With[{x = y}, Hold[Module[{y}, x + y]]]
Out[12]= Hold[Module[{y$}, y + y$]]
```

执行模块时，局部变量又一次重新进行命名，以使得名称唯一.

```
In[13]:= ReleaseHold[%]
Out[13]= y + y$1
```

Mathematica 将变换规则当定界结构处理，其中模式的名称是局部的. 可以用 *x*_、*x*__ 等或 *x*：*patt* 建立命名的模式.

h 中的 x 与 x_ 相同，在规则中作为局部量.

```
In[14]:= With[{x = 5}, g[x_, x] -> h[x]]
Out[14]= g[x_, 5] → h[x]
```

在 f[x_] :> x + y 等规则右端出现的 x 与名为 x_ 的模式匹配. 于是，x 被当作规则的局部量处理，不能用其它定界结构去修改.

另一方面，y 在规则中不是局部量，可以用其它定界结构去修改. 当这种情况发生时，***Mathematica*** 重新对规则中的模式命名以防止冲突.

Mathematica 对规则中的 x 重新命名以防止冲突.

```
In[15]:= With[{w = x}, f[x_] -> w + x]
Out[15]= f[x$_] → x + x$
```

在一个定界结构中使用 With 时，***Mathematica*** 就自动地进行适当的重新命名. 但有时需要在定界结构中进行代换以免重命名. 这可以用 /. 运算实现.

当用 With 代替 y 时，纯函数中的 x 被重新命名以防止冲突.

```
In[16]:= With[{y = x + a}, Function[{x}, x + y]]
Out[16]= Function[{x$}, x$ + (a + x)]
```

当使用 /. 而不是 With 时，没有进行这类重新命名.

```
In[17]:= Function[{x}, x + y] /. y -> a + x
Out[17]= Function[{x}, x + (a + x)]
```

时，***Mathematica*** 必须间接地替换出现在表达式 *rhs* 中的 *x*，它用 /. 运算有效地完成此项工作. 于是，这些替换不遵循定界结构. 然而，当定界结构的内部由替换修改后时，在定界结构中的其它变量被重新命名.

这里定义了一个产生纯函数的函数.

```
In[18]:= mkfun[var_, body_] := Function[{var}, body]
```

x 和 x^2 通过使用 /. 运算被简洁地插入纯函数之中.

```
In[19]:= mkfun[x, x^2]
```

```
Out[19]= Function[{x}, x^2]
```

这里定义产生一对嵌套函数的函数.

```
In[20]:= mkfun2[var_, body_] :=
           Function[{x}, Function[{var}, body + x]]
```

此时，纯函数外的 x 被重新命名.

```
In[21]:= mkfun2[x, x^2]
```

```
Out[21]= Function[{x$}, Function[{x}, x^2 + x$]]
```

相关教程

- 模块化和事物的命名

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

数学中的哑元

当建立数学公式时，需要引入各种局部的对象或“哑元”. 可以用模块或其它 ***Mathematica*** 定界结构处理这样的哑元.

积分变量是数学中哑元的常用例子. 当给出一个形式的积分时，约定的记号需要引入一个具有确定名的积分变量. 这个变量对积分而言是局部的，它的任何名称不能与数学表达式中的其它名称冲突.

定义计算积分的一个函数.

```
In[1]:= p[n_] := Integrate[f[s] s^n, {s, 0, 1}]
```

这里的 s 与积分变量冲突.

```
In[2]:= p[s + 1]
```

$$\text{Out[2]=} \int_0^1 s^{1+s} f[s] \, ds$$

这里是一个定义，其中积分变量是模块中的局部变量.

```
In[3]:= pm[n_] :=
          Module[{s}, Integrate[f[s] s^n, {s, 0, 1}]]
```

由于使用了模块，*Mathematica* 自动重新给积分变量命名以避免冲突.

```
In[4]:= pm[s + 1]
```

$$\text{Out[4]}= \int_0^1 s\$20^{1+s}\, f[s\$20]\, \mathrm{d}s\$20$$

在许多情况下，最重要的是将哑元作为局部变量，并不干扰表达式中的其它变量. 有时，同一哑元的不同方式的使用也不应该冲突.

重复的哑元经常出现在向量和张量积之中. 根据“加法约定”，恰好出现两次的向量或张量的下标应该将所有可能的值相加. 重复的下标的实际名称不起作用，但当有两个相分离的重复下标时，必须保证它们的名称不冲突.

这里将重复下标 j 作为哑元.

```
In[5]:= q[i_] := Module[{j}, a[i, j] b[j]]
```

模块对不同位置出现的哑元赋以不同的名称.

```
In[6]:= q[i1] q[i2]
Out[6]= a[i1, j$29] a[i2, j$30] b[j$29] b[j$30]
```

数学中许多情况下均需要变量有唯一的名称. 例如方程的解. 方程如 $\cos(x) = 1$ 有无穷多解 $x = 2\pi n$，其中 n 是任何整数的哑元.

当 *Mathematica* 求解以下方程时，创建一个哑元.

```
In[7]:= Reduce[Cos[x] == 1, x]
Out[7]= C[1] ∈ Integers && x == 2 π C[1]
```

这是使哑元唯一的方法.

```
In[8]:= Reduce[Cos[x] == 1, x,
         GeneratedParameters :> Unique[C]]

Out[8]= C$489[1] ∈ Integers && x == 2 π C$489[1]
```

另一个需要目标具有唯一性的地方是“积分常数” 的表示. 积分时是解一个微分方程. 一般地，这有无限个解，每个仅差一个“积分常数”. ***Mathematica*** 的标准函数 `Integrate` 总是得到一个没有积分常数的解. 但是，要引入积分常数时，必须用模块保证积分常数总是唯一的.

相关教程

- 模块化和事物的命名

教程专集

- **Core Language**

块和局部值

Mathematica 中的模块使变量名具有局部性，但有时需要名是全局的，值是局部的，这可以用 ***Mathematica*** 中的`Block` 来实现.

`Block[{x, y, ...}, body]`	用 x, y, ... 的局部值计算 $body$
`Block[{x=x0, y=y0, ...}, body]`	给 x, y, ... 赋初值

设置局部值.

涉及 x 的一个表达式.

```
In[1]:= x^2 + 3
Out[1]= 3 + x^2
```

用 x 的局部值计算以前的表达式.

```
In[2]:= Block[{x = a + 1}, %]
Out[2]= 3 + (1 + a)^2
```

x 没有全局值.

```
In[3]:= x
Out[3]= x
```

如前一节 "模块和局部变量" 所述，在模块如 `Module[{x}, body]` 中的变量 x 总是有一个唯一符号，模块每次调用这符号不同，且与全局符号 x 也有区别. `Block[{x}, body]` 中的 x 是一个全局的符号 x. 这个块的作用是让 x 有局部值. 进入这个块时，x 的值在退出该块时恢复. 而在块执行时，x 可以取任意值.

给符号 t 设置值 17.

```
In[4]:= t = 17
Out[4]= 17
```

模块中的变量有唯一的局部名.

```
In[5]:= Module[{t}, Print[t]]
        t$1
```

在块中，变量有全局名，但有局部值.

```
In[6]:= Block[{t}, Print[t]]
        t
```

t 在块中给出了一个局部值.

```
In[7]:= Block[{t}, t = 6; t^4 + 1]
Out[7]= 1297
```

当块执行结束时，t 恢复了以前的值.

```
In[8]:= t
Out[8]= 17
```

Mathematica 中的块设置了可以暂时改变值的一种环境. 在块执行过程中，用在块中所定义的变量的当前值计算表达式，不论表达式是该块的一部分或是在计算中某一处产生的情况都是如此.

这里定义了符号 u 的一个延时值.

```
In[9]:= u := x^2 + t^2
```

在块之外计算 u 时，t 的全局值被使用.

```
In[10]:= u
```

$$\text{Out[10]=}\quad 289 + x^2$$

可以指定 t 的临时值在块中使用.

```
In[11]:= Block[{t = 5}, u + 7]
```

$$\text{Out[11]=}\quad 32 + x^2$$

Mathematica 中的 Block 间接使用在 Do、Sum 和 Table 等的递推结构中.在所有这些结构中，***Mathematica*** 利用 Block 建立了迭代递推变量的局部值.

Sum 自动使递推变量 t 是局部值.

```
In[12]:= Sum[t^2, {t, 10}]
```

$$\text{Out[12]=}\quad 385$$

在递推结构中的局部变量比在 Block 中的更一般一些. 它们处理 a[1]、纯符号等变量.

```
In[13]:= Sum[a[1]^2, {a[1], 10}]
```

$$\text{Out[13]=}\quad 385$$

在 ***Mathematica*** 中定义函数时，用不直接给出变量，但能影响函数的全局变量是方便的. 例如，***Mathematica*** 有一个全局变量 $RecursionLimit，它影响所有函数的计算但又不直接是函数的变量.

Mathematica 通常将所定义的全局变量的值保持到明显的改变了它

可以通过把它设置为 ***Mathematica*** 块的局部值来实现.

定义依赖于全局变量 t 的函数.

```
In[14]:= f[x_] := x^2 + t
```

这里，使用了 t 的全局值.

```
In[15]:= f[a]
Out[15]= 17 + a^2
```

在块内，可以设置 t 的局部值.

```
In[16]:= Block[{t = 2}, f[b]]
Out[16]= 2 + b^2
```

全局变量不仅可以用来设置函数的参数，还可以积累从函数得到的结果. 在块中设置这样的局部变量，可以积累在这个块执行过程中从所调用函数得到的结果.

此函数增加了全局变量 t，返回的是它的当前值.

```
In[17]:= h[x_] := (t += x^2)
```

如果不用块，计算 h[a] 时改变全局变量 t 的值.

```
In[18]:= h[a]
Out[18]= 17 + a^2
```

使用了块后，仅局部变量 t 被影响.

```
In[19]:= Block[{t = 0}, h[c]]
Out[19]= c^2
```

全局变量 t 没有改变.

```
In[20]:= t
Out[20]= 17 + a^2
```

当输入块如 Block[{x}, body] 时，x 的所有值被删除，这意味着在这个块内，可以把 x 当“符号变量”来处理. 然而，从块明确地返回了 x 以后，它就被在块外计算时产生的值所代替.

当输入块时，t 的值被删除.

```
In[21]:= Block[{t}, Print[Expand[(t + 1)^2]]]
         1 + 2 t + t^2
```

当返回含有 t 的表达式时，它就用 t 的全局值进行计算.

```
In[22]:= Block[{t}, t^2 - 3]
Out[22]= -3 + (17 + a^2)^2
```

相关教程

- 模块化和事物的命名

教程专集

- **Core Language**

块与模块的比较

序就容易理解、维护和扩充.

保证程序中不同相相互不影响的一个重要途径是给它的变量一定的“范围”. ***Mathematica*** 用模块和块这两种机制来限制变量的范围.

在实际编程时，模块远远比块常用，而在相互作用的计算中需要确定范围时，往往是块比较方便.

`Module`[*vars*, *body*]	词汇（**lexical**）定界
`Block`[*vars*, *body*]	动态定界

Mathematica 变量的定界机理.

大部分计算机语言使用与 ***Mathematica*** 模块类似的词汇定界机理. 一些像**LISP**等符号计算语言与 ***Mathematica*** 块类似的动态定界机理.

在使用词汇定界时，变量在一个程序中的一个代码段被作为局部变量. 在动态定界时，在程序执行历史的一部分被作为局部值.

在 **C** 和 **Java** 等编译语言中， 它们的变量在使用之前就要声明类型，因此在编译前就已经确定了变量的类型；“代码”和“执行历史”之间的区分非常明显. 而 ***Mathematica*** 属于动态类型语言，它的符号特性使这个区别不明显，其原因是代码在程序的执行过程中可以动态地生成.

`Module`[*vars*, *body*] 的作用是在模块作为 ***Mathematica*** 的代码被执行时处理表达式 *body* 的形式，当任何 *vars* 明显地出现在代码中时，就被当作局部变量.

`Block`[*vars*, *body*] 不注意表达式 *body* 的形式. 而是，在 *body* 的全局计算过程中使用 *vars* 的局部值.

通过 i 来定义 m.

```
In[1]:= m = i^2
```

$$\text{Out[1]= } i^2$$

在块内 i + m 的计算过程中，i 用了局部值.

```
In[2]:= Block[{i = a}, i + m]
```

$$\text{Out[2]= } a + a^2$$

这里仅明显出现在 i + m 中的 i 被当作局部变量处理.

```
In[3]:= Module[{i = a}, i + m]
```

$$\text{Out[3]= } a + i^2$$

相关指南

- 过程式编程

相关教程

- 模块化和事物的命名

教程专集

- **Core Language**

上下文

总是给变量或者定义选用尽可能清楚的名称是一个好思想. 但这样做有时会导致变量名很长.

在 ***Mathematica*** 中，可以用上下文来组织符合名. 在引入与其它符号不冲突的变量名的 ***Mathematica*** 程序包中上下文特别有用. 在编写或者调用 ***Mathematica*** 程序包时，就需要了解上下文.

其基本的思想是任何符号的全名为两部分：上下文和短名. 全名被写为 *context*`*short*，其中 ` 是倒引号或重音符字符（ **ASCII** 二进制代码 **96**），在 ***Mathematica*** 中称为上下文标记.

这里是具有短名 x 和上下文 aaaa 的符号.

```
In[1]:= aaaa`x
Out[1]= aaaa`x
```

可以像其它符号一样使用这一符号.

```
In[2]:= %^2 - %
Out[2]= -aaaa`x + aaaa`x^2
```

例如，可以定义这个符号的值.

```
In[3]:= aaaa`x = 78
Out[3]= 78
```

Mathematica 将 a`x 和 b`x 当作两个完全不同的符号.

```
In[4]:= a`x == b`x
Out[4]= a`x == b`x
```

典型的情况是让与一个特殊的主题相关的符号有相同的上下文. 例

的全名可能是 PhysicalUnits`Joule 或者 PhysicalUnits`Mole.

尽管总可以用全名来代表一个符号，但是用短名常常很方便.

在 ***Mathematica*** 进程中的任何点，总有一个当前的上下文 $Context. 可以用短名简单地指代这个上下文中的符号，除非该符号被 $ContextPath 中具有相同短名的符号屏蔽. 如果具有给定短名的符号在上下文路径中存在，它将被使用，而不是当前上下文中的符号被使用.

Mathematica 进程的默认上下文是 Global`.

```
In[5]:= $Context
Out[5]= Global`
```

在 $ContextPath 中不存在名为 x 的符号，因此在当前上下文中用短名指代符号是足够的.

```
In[6]:= {x, Global`x}
Out[6]= {x, x}
```

上下文在 ***Mathematica*** 中的作用在某种程度上类似于许多操作系统的文件目录，可以通过路径和全名指定一个文件. 但在任何点，总有一个当前目录，这类似于 ***Mathematica*** 的当前上下文. 在当前目录下的文件就可以仅用它的短名指定.

与许多操作系统中的目录一样，***Mathematica*** 中的上下文具有启发式（分层）的特性. 例如，符号的全名可以涉及到一系列形如 c_1`$c_2$`c_3`*name* 的上下文名.

context `*name* **or** *c*$_1$ `*c*$_2$ … `*name*	在明确指定的上下文中的符号
`*name*	当前上下文中的符号
`*context* `*name* **or** `*c*$_1$ `*c*$_2$ ` … `*name*	与当前上下文相关的在一个指定上下文中的符号
name	在当前上下文或者在上下文搜索路径中的符号

在不同上下文中指定符号.

在上下文 a`b` 中的符号.

```
In[7]:= a`b`x
Out[7]= a`b`x
```

开始了一个 ***Mathematica*** 进程后，默认当前上下文是 Global`. 引入的符号通常就在这个上下文中. 但是，内部符号Pi 等在上下文 System` 中.

为了方便地处理 Global` 和 System` 中的符号，***Mathematica*** 支持上下文搜索路径. 在 ***Mathematica*** 进程中的任一点，有当前上下文 $Context 和当前上下文搜索路径 $ContextPath. 搜索路径的意思是在输入一个符号的短名后，***Mathematica*** 在一系列上下文中搜索去找到有这个短名的符号.

Mathematica 中的上下文搜索路径与操作系统提供的程序文件的“搜索路径”相似. 由于 $Context 是在$ContextPath 之后搜索的，我们可以认为它在文件搜索路径后添加“.”.

默认的上下文路径包括系统定义符号的上下文.

```
In[8]:= $ContextPath
Out[8]= {System`, Global`}
```

当输入 Pi 时，*Mathematica* 将它翻译为具有全名 System`Pi 的符号.

```
In[9]:= Context[Pi]
Out[9]= System`
```

Context[*s*]	一个符号的上下文
$Context	在 *Mathematica* 进程中的当前上下文
$ContextPath	当前上下文搜索路径
Contexts[]	所有上下文组成的集合

找出上下文和上下文搜索路径.

在 *Mathematica* 中使用上下文时，在两个不同的上下文中两个符号可以有相同的短名. 例如，在上下文 PhysicalUnits` 和 BiologicalOrganisms` 中都可以使用短名 Mole 的符号.

于是，在输入短名 Mole 后，就产生了用户实际上调用了哪一个符号的问题. 解决这个问题时要弄清楚在上下文搜索路径中哪一个上下文先出现.

这里引入两个具有 Mole 短名的符号.

```
In[10]:= {PhysicalUnits`Mole, BiologicalOrganisms`Mole
Out[10]= {PhysicalUnits`Mole, BiologicalOrganisms`Mole
```

这里对 $ContextPath 添加了两个上下文. 通常，*Mathematica* 在 $ContextPath 开头添加新的上下文.

```
In[11]:= $ContextPath = Join[{"PhysicalUnits`",
           "BiologicalOrganisms`"}, $ContextPath]

Out[11]= {PhysicalUnits`,
          BiologicalOrganisms`, System`, Global`}
```

现在如果输入 Mole，得到了在上下文 PhysicalUnits` 中的符号.

```
In[12]:= Context[Mole]

Out[12]= PhysicalUnits`
```

一般地，当输入一个符号的短名后，*Mathematica* 认为用户需要在上下文搜索路径中最早出现的上下文中的符号. 结果，上下文出现晚的符号或者在当前上下文中具有相同短名的符号将被屏蔽，为了调用这些符号必须使用全名.

在引入的新符号屏蔽了当前 $ContextPath 中已经存在的符号时，*Mathematica* 就会发出警告. 另外，如果在笔记本前端，*Mathematica* 就提示用户对屏蔽的符号使用红色.

这里在 Global` 上下文中引入了有短名 Mole 的符号，*Mathematica* 警告新符号屏蔽了已经存在的短名为 Mole 的符号.

```
In[13]:= Global`Mole

         Global`Mole::shdw :
          Symbol Mole appears in multiple contexts
            {Global`, PhysicalUnits`, BiologicalOrganisms`};
            definitions in context Global` may shadow
            or be shadowed by other definitions. >>
```

如果在这里输入 Mole，就会得到上下文路径 PhysicalUnits` 的符号.

```
In[14]:= Context[Mole]
Out[14]= PhysicalUnits`
```

如果引入的符号屏蔽了已经存在的符号，则需要重新安排 $ContextPath，或者直接删去这个符号. 用户应该意识到，仅清除这个符号的值是不够的，必须从 ***Mathematica*** 中全部删除这个符号. 这可以用函数 Remove[*s*] 来实现.

Clear[*s*]	清除一个符号的值
Remove[*s*]	从系统中完全删除一个符号

清除或删除 ***Mathematica*** 中的符号.

删除符号 PhysicalUnits`Mole.

```
In[15]:= Remove[Mole]
```

此时输入 Mole 后，就会得到符号 BiologicalOrganisms`Mole.

```
In[16]:= Context[Mole]
Out[16]= BiologicalOrganisms`
```

当 ***Mathematica*** 显示一个符号名时，它必须选择区显示全名或者短名. 它所做的是给出所有用户应该输入的名称以得到特定的符号，给出当前 $Context 和 $ContextPath 的设置.

对第一个符号先显示短名，这就是输入短名后应该得到的符号.

```
In[17]:= {BiologicalOrganisms`Mole, Global`Mole}
Out[17]= {Mole, Global`Mole}
```

当输入了一个短名，它既不在当前上下文内，也不在上下文的搜索路径内，***Mathematica*** 就产生一个具有该短名的新符号，且总是把新符号放在由 `$Context` 指定的当前上下文中.

这里引入具有短名 `tree` 的符号.

```
In[18]:= tree
Out[18]= tree
```

Mathematica 在当前上下文 `Global`` 内增加了 `tree`.

```
In[19]:= Context[tree]
Out[19]= Global`
```

相关教程

- 模块化和事物的命名
- 基本对象

教程专集

- **Core Language**

上下文和程序包

典型的 ***Mathematica*** 程序包引入一些能在这个包之外使用的符号. 这些符号可能与在包中定义的新函数或目标相对应.

约定在一个程序包中引入的新符号放在名与该程序包的相关的上下文内. 在这个包中阅读时，这个上下文将加在上下文搜索路径 `$ContextPath` 的开头.

为了证明 **primality**，这里读入一个程序包.

```
In[1]:= << PrimalityProving`
```

这个包将它的上下文添加在 `$ContextPath` 之前.

```
In[2]:= $ContextPath
Out[2]= {PrimalityProving`, System`, Global`}
```

符号 `ProvablePrimeQ` 在程序包设置的上下文之中.

```
In[3]:= Context[ProvablePrimeQ]
Out[3]= PrimalityProving`
```

可以用短名来指代这个符号.

```
In[4]:= ProvablePrimeQ[2143]
Out[4]= True
```

在一个程序包中定义的变量的全名往往很长，大部分情况下，只需要用它的短名即可，其原因是在读入这个程序包时，它的上下文被加在 `$ContextPath` 之中，当敲入短名后就会自动搜索这个上下文.

当同一个短名出现在两个不同的程序包中时就变得比较复杂，这时当读入第二个程序包时，***Mathematica*** 就会发出警告，它告诉我们哪

在 PrimalityProving` 中的符号 ProvablePrimeQ 被新程序包中的有相同短名的符号所屏蔽.

```
In[5]:= << NewPrimalityProving`

        ProvablePrimeQ::shdw :
         Symbol ProvablePrimeQ appears in multiple contexts
           {NewPrimalityProving`, PrimalityProving`};
           definitions in context NewPrimalityProving` may
           shadow or be shadowed by other definitions. >>
```

可以通过全名调用被屏蔽的符号.

```
In[6]:= PrimalityProving`ProvablePrimeQ[2143]

Out[6]= True
```

冲突不仅在不同程序包的符号间发生，而且也在程序包的符号与 ***Mathematica*** 的进程中引入的符号间发生. 在当前上下文中定义了符号后，这个符号将被从程序包中读入的短名相同的符号所屏蔽，其原因是 ***Mathematica*** 总是先在上下文搜索路径中寻找符号，然后才在当前上下文中寻找符号.

在当前上下文中定义了一个函数.

```
In[7]:= ScalarQ[f_] = Head[f] =!= List

Out[7]= True


In[8]:= ScalarQ[2 I]

Out[8]= True
```

任何其它名为 ScalarQ 的函数将被再当前上下文中的函数屏蔽.

```
In[9]:= << Quaternions`

          ScalarQ::shdw :
           Symbol ScalarQ appears in multiple contexts
             {Quaternions`, Global`}; definitions
             in context Quaternions` may shadow or
             be shadowed by other definitions. >>
```

这里使用了程序包中的 ScalarQ.

```
In[10]:= ScalarQ[2 I]

Out[10]= False
```

当不需要的符号屏蔽了所需的符号时，最好的途径是用 Remove[*s*] 去删除这个不需要的符号. 有时另一个合适的选择是去重新排列 $ContextPath 中的元素，重新设置 $Context 的值，以便包含所需符号的上下文先出现.

$Packages	一个对应于 ***Mathematica*** 进程中加载的所有程序包的上下文集合

给出一个程序包集合.

相关指南

- 程序包开发
- 命名空间的管理

相关教程

- 上下文

- 模块化和事物的命名
- ***Mathematica*** 程序包

教程专集

- **Core Language**

Mathematica 程序包

Mathematica 的一个重要特性在于它是一个可扩充系统. ***Mathematica*** 中已建立了一定数量的数学和其它功能. 然而，使用 ***Mathematica*** 语言，能添加更多的函数.

对许多种运算，***Mathematica*** 标准版中建立的函数已经足够用了. 然而，当用户在一个特殊专业领域中讨论时，会需要使用特定函数，而这在 ***Mathematica*** 中是没有的.

在这种情况下，用户或许能在 ***Mathematica*** 程序包中找到所需的函数. ***Mathematica*** 程序包是用 ***Mathematica*** 语言写成的文件. 这些文件是由 ***Mathematica*** 定义集合组成，使 ***Mathematica*** 能进行特殊应用领域的工作.

<<package	读入一个 ***Mathematica*** 程序包

读入 ***Mathematica*** 程序包.

如果要使用某个程序包中的函数，必须首先把程序包读入 ***Mathematica*** 中. 具体做法将在 "外部程序" 节讨论. 使用软件包时有许多规定.

这个命令读入一个特定的 ***Mathematica*** 程序包.

```
In[1]:= << PrimalityProving`
```

该程序包中定义了 ProvablePrimeQ 函数.

```
In[2]:= ProvablePrimeQ[1093]
Out[2]= True
```

在不同程序包的函数名之间有许多矛盾. 这些将在 "上下文和程序包" 节讨论. 需要注意一点，在读入某个程序包之前不要使用名称与该程序包中的函数名相同的函数. 如果你错误地这样做了，***Mathematica*** 将会发出一条警告信息并且使用最后定义的名称. 这意味着你定义的函数版本将不会被使用；而它将从程序包中得到. 必须执行命令Remove["*name*"] 来取消程序包函数.

Remove["*name*"]	取消用户自定义的函数

确认 ***Mathematica*** 使用程序包中定义的函数.

Mathematica 通过使用程序包能被扩充的事实说明“ ***Mathematica*** 部分”的界限是模糊的. 从用法上来说，程序包中定义的函数与 ***Mathematica*** 的内部函数没有任何区别.

事实上，本书中所介绍的许多函数实际上是 ***Mathematica*** 程序包中的函数. 而在大多数 ***Mathematica*** 系统中，必须的程序包已经被预先装入了. 所以，其中定义的函数总是存在的.

为了进一步淡化 ***Mathematica*** 的边界， "包的自动调入" 节介绍了如何使 ***Mathematica*** 自动装入特定包. 如果用户从未使用过该包中某个函数，那么该函数是不在 ***Mathematica*** 中的，但是一旦用户要使用它，则它就被从程序包中读入 ***Mathematica*** 中.

从实际情况看来，被认为是“***Mathematica*** 部分”的函数或许是那些在所有 ***Mathematica*** 系统中都有的函数. 本节讨论的也正是这些函数.

然而，***Mathematica*** 的大多数版本都具有标准的 ***Mathematica*** 程序包集合，其中包含了更多的函数. 其中一些函数将来都会提到. 要使用这些函数，用户通常需要明确读入必要的程序包.

用户可以使用参考资料中心来获得 ***Mathematica 8*** 标准附加程序包的信息.



当然，能够设置 ***Mathematica*** 系统，使得特定的程序包被预先装入

作为标准函数出现在所使用的 ***Mathematica*** 版本中，不过没有在 ***Mathematica*** 系统参考文档中出现.

有一点要提到的是程序包和笔记本之间的关系. 二者都是作为文件储存在计算机系统中，都能被读入 ***Mathematica*** 中. 但是，笔记本是要被显示的，而程序包只是作为 ***Mathematica*** 的输入. 事实上，许多笔记本包含了被认为是程序包的段落，其中包含着打算作为 ***Mathematica*** 输入的定义序列. 另外，还可以建立程序包来自动维护笔记本.

相关教程

- 模块化和事物的命名
- 符号数学程序包
- 建立 ***Mathematica*** 程序包

教程专集

- **Core Language**

建立 *Mathematica* 程序包

在一个典型的 ***Mathematica*** 程序包中，一般引入两种类型的新符号，一种是输出为外部程序包使用的符号，另一种仅在本程序包中使用的符号. 可以将这两种类型的符号放入不同的上下文以便区分.

通常的约定是将用作输出的符号放在名为 *Package`* 的上下文内，这与程序包的名称相对应. 当这个程序包读入时，它将这个上下文加到上下文搜索路径中去，所以在这个上下文中的符号可以通过短名调用

仅在程序包内使用的符号习惯上放在名为 *Package*`Private` 的上下文内. 这个上下文没有加入到上下文的搜索路径中去. 于是，这些符号只有通过全名才能调用.

Package `	用于输出的符号
Package `Private`	仅在内部使用的符号
System`	***Mathematica*** 中的内部符号
$Needed_1$ `, $Needed_2$ `, …	在程序包中需要的其它上下文

Mathematica 程序包中使用上下文的约定.

有一个标准的 ***Mathematica*** 指令序列用来设置程序包中的上下文. 这些指令设置 $Context 和 $ContextPath 的值以便新引入的符号放在适当的上下文之中.

BeginPackage["*Package*`"]	将 *Package* ` 作为当前上下文，仅将 System` 放在上下文搜索路径中
f::usage="*text*", …	引入作为输出的目标
Begin["`Private`"]	设置当前上下文为 *Package* `Private`
f[*args*]=*value*, …	给出在程序包中定义的主体
End[]	转换到前一个上下文（这里为 *Package* `）
EndPackage[]	上下文结束程序包，将 *Package* ` 放到上下文搜索路径前面

在程序包中的上下文控制指令标准序列.

```
BeginPackage["Collatz`"]

Collatz::usage =
        "Collatz[n] gives a list of the iterates
in the 3n+1 problem,
        starting from n. The conjecture is that
this sequence always
        terminates."

Begin["`Private`"]

Collatz[1] := {1}

Collatz[n_Integer]  := Prepend[Collatz[3 n + 1],
n] /; OddQ[n] && n > 0

Collatz[n_Integer] := Prepend[Collatz[n/2], n]
/; EvenQ[n] && n > 0

End[ ]

EndPackage[ ]
```

举例： Collatz.m 程序包.

定义在一个程序包开头使用 usage 信息的约定是在合适的上下文中产生用作输出的符号的一个基本的技巧. 在定义这些信息时，所涉及的符号都是用作输出的符号，这些符号都在作为当前上下文的 *Package*` 中产生.

在一个程序包中函数的实际定义中，有许多引入参数、局部变量等新符号. 约定将这些符号放在 *Package*`Private` 上下文中，当程序包读入后它不放在上下文的搜索路径之中.

这里读入前面的样本程序包.

```
In[1]:= << ExampleData/Collatz.m
```

在这个包中的 EndPackage 指令将与该包对应的上下文放入上下文搜索路径中.

```
In[2]:= $ContextPath
Out[2]= {Collatz`, Global`, System`}
```

Collatz 函数在 Collatz` 上下文中产生.

```
In[3]:= Context[Collatz]
Out[3]= Collatz`
```

参数 n 放在专用上下文 Collatz`Private` 中.

```
In[4]:= ?Collatz`Private`*
        Collatz`Private`n
```

在 Collatz 包中所定义的函数仅依赖于 ***Mathematica*** 的内部函数，但通常在一个包中定义的函数依赖着另一个包中定义的函数.

为此必须有两个条件，第一是读入另一个程序包以便所需的函数有定义，第二是上下文搜索路径必须包括这些函数所在的上下文.

可以明确告诉 ***Mathematica*** 在任一处读入程序包，使用的指令为 << *context*`. （"程序包中的文件" 讨论了从与系统不相关的上下文名称到与系统相关的文件名之间的转换）. 然后，设置为需要时就读入某一程序包. 指令 Needs["*context*`"] 告诉 ***Mathematica*** 当与程序包相关的上下文不在 $Packages 列表中时就读入这个程序包.

Get["*context*`"] 或 <<*context* `	读入特定上下文对应的程序包
Needs["*context*`"]	当指定的上下文不在 $Packages 中时读入程序包
BeginPackage["*Package*`",{" $Needed_1$ `", … }]	开始一个包，指定除 System` 外所需要的上下文

指定独立包函数.

当使用一个具有一个变量的 BeginPackage["*Package*`"] 时，***Mathematica*** 仅将 *Package*` 上下文和 ***Mathematica*** 内部符号上下文放在上下文的搜索路径之中. 在自定义包中涉及到其它包中的函数时，一定要确认这些包含在上下文的搜索路径之中，这可以在 BeginPackage 的第二个变量中给出另外的上下文列表即可. BeginPackage 就自动调用 Needs，必要时读入与这些上下文对应的程序包，确定这些上下文在上下文的搜索路径之中.

Begin["*context*`"]	转向一个新的当前上下文
End[]	转向前一个上下文

Context 操作函数.

Begin 等上下文操作函数的执行改变了 ***Mathematica*** 翻译所输入名字的方式. 但要意识到这种改变仅对输入的下一个表达式有效，其原因是 ***Mathematica*** 总是先读入一个完整的表达式，在执行表达式的任何项之前翻译它中的名字. 于是，当一个特定的表达式中执行 Begin 时，表达式中的名称已被翻译，使 Begin 无法再产生效果

上下文操作函数对第2个读入的完整表达式不产生影响这一事实意味着当编写 ***Mathematica*** 程序包时必须作为分离的表达式给出这些函数，经常是在不同的行.

x 的名字在表达式执行之前已经翻译，所以 Begin 没有作用.

```
In[5]:= Begin["a`"]; Print[Context[x]]; End[]
        Global`
Out[5]= a`
```

上下文操作函数首先用作程序包的一部分被 ***Mathematica*** 读入. 有时，交互式地应用它们也很方便.

例如，在执行一个包中定义的函数时用 TraceDialog 进行一个对话，函数中的参数和临时变量一般是在与这个有关的上下文中. 由于这个上下文不在上下文的搜索路径之中，***Mathematica*** 将显示这些符号的全名，并且需要用户输入全名去调用这些变量. 但还可以用 Begin[*"Package`Private`"*] 使包专用上下文成为当前上下文. 这就使 ***Mathematica*** 显示符号的短名，也就可以用短名调用这些符号.

相关教程

- 模块化和事物的命名
- ***Mathematica*** 程序包
- ***Mathematica*** 脚本

教程专集

- **Core Language**

程序包中的文件

当产生或使用 ***Mathematica*** 程序包时，常常需要以与系统无关的方式去引用一个文件. 这可以用上下文去进行.

其基本思想是每一个计算机系统有一个约定，它决定怎样根据 ***Mathematica*** 的上下文去命名文件. 当用一个上下文去引用一个文件时，所用的 ***Mathematica*** 版本就将上下文名转换为适应于所用计算机系统的文件名.

<<context`	读入一个对应于指定上下文的文件

使用上下文指定文件.

读入一个 ***Mathematica*** 标准程序包.

```
In[1]:= << Quaternions`
```

name.mx	DumpSave 格式文件
name.mx/$SystemID/*name*.mx	所用计算机系统的 DumpSave 格式文件
name.m	***Mathematica*** 资源格式文件
name/init.m	一个目录的初始化文件
dir/...	在 $Path 指定的其它目录中的文件

<< name` 寻找的文件序列.

Mathematica 的设置使 *<< name`* 自动装载一个文件的适当版本. 它先试装载对所用计算机系统优化过的 *name*.mx 文件，然后如果找

name.m 文件.

当 *name* 是一个目录时，***Mathematica*** 就试装载该目录中的初始化文件 init.m. init.m 文件的目的是为设置含有许多文件的 ***Mathematica*** 程序包提供一个方便的途径. 其思想是先给出一个指令 << *name*`，然后装载 init.m 将整个程序包初始化，需要时读入其它相关文件.

相关教程

- 文件和流
- 文件、流和外部操作

教程专集

- **Core Language**
- **Data Manipulation**

包的自动调入

其它节中，我们已经讨论了用 << *package* 和Needs[*package*]直接调入 ***Mathematica*** 程序包. 有时候，还要对 ***Mathematica*** 进行设置以便在必要时自动调入一个包.

可以用 DeclarePackage 去给出在一个包中定义的符号名，然后当某一符号被调用时，***Mathematica*** 就自动调入包含这个符号的包.

DeclarePackage["context`",{"name$_1$","name$_2$",...}]

指明当名为 $name_i$ 的符号使用时自动调入一个包

自动调入程序包的安排.

这里指明符号 Div，Grad 和 Curl 在 VectorAnalysis` 中定义.

```
In[1]:= DeclarePackage["VectorAnalysis`",
         {"Div", "Grad", "Curl"}]

Out[1]= VectorAnalysis`
```

第一次使用 Grad 时，*Mathematica* 自动调入定义它的包.

```
In[2]:= Grad[x^2 + y^2, Cartesian[x, y, z]]

Out[2]= {2 x, 2 y, 0}
```

当建立大量的 *Mathematica* 程序包时，最好产生一系列 DeclarePackage 命令的名称文件，以便在某一名称被使用时调入一个程序包. 在一个 *Mathematica* 进程中，仅需要调入名称文件，随后其它的程序包在需要时就被自动调入.

DeclarePackage 对指定的名称立即产生符号，但给每个称号一个特定的属性 Stub. 当 *Mathematica* 发现一个符号具有 Stub 属性时，它就自动调入与该符号对应的内容以得到这个符号的定义.

相关指南

- 程序包开发

相关教程

- 模块化和事物的命名
- ***Mathematica*** 程序包

教程专集

- **Core Language**

通过名称操作符号和内容

`Symbol["name"]`	建立一个具有给定名称的符号
`SymbolName[symb]`	找出一个符号的名称

符号及其名称之间的转换.

这是符号 x.

```
In[1]:= x // InputForm
Out[1]//InputForm=
x
```

它的名字是一个字符串.

```
In[2]:= SymbolName[x] // InputForm
Out[2]//InputForm=
"x"
```

这里重新给出符号 x.

```
In[3]:= Symbol["x"] // InputForm
```

```
t[3]//InputForm=
x
```

当用 x = 2 赋值后，计算时 x 就用 2 代替. 有时候，需要继续使用 x 本身，而不用立即获得它的值 x.

这可以通过调用 x 的名字来实现. 符号 x 的名字是一个字符串 "x"，即使 x 本身用值代替，但字符串 "x" 永远用这个名字.

符号 x 和 xp 的名称是字符串 "x" 和 "xp".

```
In[4]:= t = {SymbolName[x], SymbolName[xp]} // InputFor
```

```
t[4]//InputForm=
{"x", "xp"}
```

这里对 x 赋值.

```
In[5]:= x = 2

Out[5]= 2
```

任何时候输入 x，它被 2 代替.

```
In[6]:= {x, xp} // InputForm
```

```
t[6]//InputForm=
{2, xp}
```

但名称 "x" 没有受影响.

```
In[7]:= t // InputForm

t[7]//InputForm=
        InputForm[{"x", "xp"}]
```

NameQ["*form*"]	测试是否有与 *form* 匹配的已命令名符号
Names["*form*"]	给出与 *form* 匹配的符号列表
Contexts["*form*`"]	给出与 *form* 匹配的所有上下文名称的列表

用名称去指代符号和内容.

x 和 xp 是在 ***Mathematica*** 的这一进程中产生的符号，而 xpp 不是.

```
In[8]:= {NameQ["x"], NameQ["xp"], NameQ["xpp"]}
Out[8]= {True, True, False}
```

可以用 "字符串模式" 节讨论的字符串模式来指定符号名的类型，例如 "x*" 表示所有以 x 开头的符号名.

这里给出了在一个 ***Mathematica*** 进程中所有以 x 开头的符号名.

```
In[9]:= Names["x*"] // InputForm

t[9]//InputForm=
        {"x", "xp"}
```

这些名称与 ***Mathematica*** 的内部函数相对应.

```
In[10]:= Names["Qu*"] // InputForm
```

```
t[10]//InputForm=
        {"QuadraticIrrationalQ", "Quantile", "Quartic
         "QuartileSkewness", "Quiet", "Quit", "Quotie
```

这里寻找与 WeierstrssP "接近" 的名称.

```
In[11]:= Names["WeierstrssP",
           SpellingCorrection -> True]
```

```
Out[11]= {WeierstrassP}
```

`Clear["form"]`	删除名与 *form* 匹配的所有符号的值
`Clear["context`*"]`	删除在指定上下文中所有符号的值
`Remove["form"]`	完全删除名称与 *form* 匹配的符号
`Remove["context`*"]`	在指定的上下文中删除所有符号

通过名称删除变量.

删除所有名以 x 开头的符号的值.

```
In[12]:= Clear["x*"]
```

"x" 这一名称还存在.

```
In[13]:= Names["x*"]
```

```
Out[13]= {x, xp}
```

这些名称与 ***Mathematica*** 的内部函数相对应.

```
In[10]:= Names["Qu*"] // InputForm
```

```
rationalQ", "Quantile", "Quartics", "QuartileDeviati
wness", "Quiet", "Quit", "Quotient", "QuotientRemain
```

这里寻找与 WeierstrssP “接近” 的名称.

```
In[11]:= Names["WeierstrssP",
           SpellingCorrection -> True]
```

```
Out[11]= {WeierstrassP}
```

`Clear["form"]`	删除名与 *form* 匹配的所有符号的值
`Clear["context`*"]`	删除在指定上下文中所有符号的值
`Remove["form"]`	完全删除名称与 *form* 匹配的符号
`Remove["context`*"]`	在指定的上下文中删除所有符号

通过名称删除变量.

删除所有名以 x 开头的符号的值.

```
In[12]:= Clear["x*"]
```

"x" 这一名称还存在.

```
In[13]:= Names["x*"]
```

```
Out[13]= {x, xp}
```

但是 x 的值已经被删除.

```
In[14]:= {x, xp}
Out[14]= {x, xp}
```

这里完全删除了名以 x 开始的符号.

```
In[15]:= Remove["x*"]
```

这里 "x" 就不再存在.

```
In[16]:= Names["x*"]
Out[16]= {}
```

Remove["Global`*"]	完全删除在 Global` 的值 **context** 中的符号

删除所有引入的符号.

当没有建立其它上下文时，所有在 ***Mathematica*** 进程中引入的符号都放在 Global` **context** 中. 可以用Remove["Global`*"] 完全删除这些符号. 而 ***Mathematica*** 的内部对象都在 System` **context** 中，所以它们不受影响.

相关教程

- 模块化和事物的命名

教程专集

- Core Language

拦截新符号的产生

当新输入一个名称时，***Mathematica*** 就产生一个新符号，有时候“拦截”新符号的产生是必要的. ***Mathematica*** 提供了几种途径来阻止新符号的产生.

`On[General::newsym]`	任何新符号产生时就显示一个信息
`Off[General::newsym]`	关闭新符号产生时的信息提示

当新符号产生时显示一个信息.

这里告诉 ***Mathematica*** 产生新符号时显示一个信息.

```
In[1]:= On[General::newsym]
```

Mathematica 于是就显示新产生符号的有关信息.

```
In[2]:= sin[k]
                General::newsym : Symbol sin is new. >>
                General::newsym : Symbol k is new. >>
Out[2]= sin[k]
```

这里关闭信息.

```
In[3]:= Off[General::newsym]
```

在 ***Mathematica*** 产生新符号时显示信息是发现输入错误的一个好方

写错误，但通过提示新产生符号的信息， ***Mathematica*** 可以使我们看到是否有错.

`$NewSymbol`	作用于新产生符号的名称和内容上的函数

当新符号产生时进行操作.

当 ***Mathematica*** 产生一个新符号时，有可能不需要显示信息，需要做一些其它的事情. 所指定的全局变量 `$NewSymbol` 函数将自动作用到给出 ***Mathematica*** 所产生的每一个新符号名称的字符串上.

定义产生新符号时使用的函数.

```
In[4]:= $NewSymbol =
         Print["Name: ", #1, " Context: ", #2] &
Out[4]= Print[Name: , #1,  Context: , #2] &
```

该函数分别对 `v` 和 `w` 作用一次.

```
In[5]:= v + w
        Name: v Context: Global`
        Name: w Context: Global`
Out[5]= v + w
```

相关教程

- 模块化和事物的命名

教程专集

字符串和字符

字符串的性质
字符串的运算
字符串中的字符
字符串模式
规则表达式
特殊字符
字符串中的换行和 Tab
字符代码
字符的原始代码
高级字符串模式

教程专集

Core Language

字符串的性质

Mathematica 大部分是有结构的表达式，但它也可以用来作为 ***Mathematica*** 无结构的文本处理系统

"*text*"	包含任意文本的字符串

文本字符串.

在 ***Mathematica*** 中输入的字符串要放在引号之中，但 ***Mathematica*** 输出字符串时没有引号.

可以通过字符串的形式来显示引号. 在 ***Mathematica*** 的笔记本中，引号在编辑字符串时自动出现.

当 ***Mathematica*** 输出一个字符串时，一般没有引号.

```
In[1]:= "This is a string."
Out[1]= This is a string.
```

通过输入形式的调用可以显示引号.

```
In[2]:= InputForm[%]
t[2]//InputForm=
        "This is a string."
```

Mathematica 输出字符串时一般没有引号可以使我们用字符串去代表不同的输出内容.

这里显示的字符串明显没有引号.

```
In[3]:= Print["The value is ", 567, "."]
        The value is 567.
```

应该理解字符串 "x" 在输出时为 x，但它与符号 x 完全不同.

字符 "x" 与符号 x 不同.

```
In[4]:= "x" === x
Out[4]= False
```

可以通过头部测试一个表达式是不是字符. 字符串的头部总是 String.

所有字符串的头部为 String.

```
In[5]:= Head["x"]
Out[5]= String
```

模式 _String 与任何字符匹配.

```
In[6]:= Cases[{"ab", x, "a", y}, _String]
Out[6]= {ab, a}
```

字符串可以和其它表达式一样作为模式或变换的项. 但不能直接对字符串赋值.

给出涉及字符串的表达式的定义.

```
In[7]:= z["gold"] = 79
Out[7]= 79
```

用符号 x 代替所有出现的 "aa" 字符串.

```
In[8]:= {"aaa", "aa", "bb", "aa"} /. "aa" -> x
Out[8]= {aaa, x, bb, x}
```

教程专集

- **Core Language**

字符串的运算

Mathematica 提供了各种字符串运算函数，这些函数的大部分基本出发点是将字符串当作一个字符序列，许多函数都与列表的运算类似.

s_1<><>s_2<>... 或 StringJoin[{s_1, s_2, ...}]	将几个字符串连接在一起
StringLength[s]	给出一个串中的字符数
StringReverse[s]	颠倒串中字符的顺序

完全字符串的运算.

用 <> 可以将任意数目的字符串相连接.

```
In[1]:= "aaaaaaa" <> "bbb" <> "ccccccccccc"
Out[1]= aaaaaaabbbccccccccccc
```

StringLength 给出串中的字符数.

```
In[2]:= StringLength[%]
Out[2]= 20
```

StringReverse 颠倒串中的字符.

```
In[3]:= StringReverse["A string."]
Out[3]= .gnirts A
```

StringTake[s,n]	由 s 的前 n 个字符形成字符串
StringTake[s,{n}]	取出 s 中的第 n 个字符
StringTake[s,{n_1,n_2}]	取出 n_1 到 n_2 的字符
StringDrop[s,n]	在 s 中用截掉前 n 个字符的方式得到字符串
StringDrop[s,{n_1,n_2}]	截掉从 n_1 到 n_2 的字符

取出或截掉子串.

字符串处理中的 StringTake 和 StringDrop 与集合运算中的 Take 和 Drop 类似. 与 Take 和 Drop 相似，它们用 ***Mathematica*** 标准的排序方式，例如，当**n**为负数时是从字符串的最后开始计数. 注意，字符串中的第一个字符的位置是 1.

这是一个简单的字符串.

```
In[4]:= alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Out[4]= ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

从 alpha 中取出前5个字符.

```
In[5]:= StringTake[alpha, 5]
Out[5]= ABCDE
```

alpha 中的第5个字符.

```
In[6]:= StringTake[alpha, {5}]
Out[6]= E
```

去掉倒数第2到倒数第10个字符.

```
In[7]:= StringDrop[alpha, {-10, -2}]
Out[7]= ABCDEFGHIJKLMNOPZ
```

StringInsert[s, snew, n]	将字符串 *snew* 插在 *s* 中的第 *n* 个位置
StringInsert[s, snew, {n_1, n_2, ...}]	将字符串 *snew* 多次插入 *s* 中

在一个字符串中的插入.

StringInsert[*s*, *snew*, *n*] 产生第 *n* 个字符是 *snew* 的第一个字符的字符串.

产生一个新字符串，它的第4个位置是 "XX" 的第一个字符.

```
In[8]:= StringInsert["abcdefgh", "XX", 4]
Out[8]= abcXXdefgh
```

负位置是从字符串的后面计数.

```
In[9]:= StringInsert["abcdefgh", "XXX", -1]
Out[9]= abcdefghXXX
```

"XXX" 被重复插入原字符串的指定位置.

```
In[10]:= StringInsert["abcdefgh", "XXX", {2, 4, -1}]
Out[10]= aXXXbcXXXdefghXXX
```

使用 Riffle 在列表中的词间加入空格.

```
In[11]:= StringJoin[
          Riffle[{"cat", "in", "the", "hat"}, " "]]
Out[11]= cat in the hat
```

StringReplacePart[*s*, *snew*, {*m*, *n*}]	用字符串 *snew* 替换 *s* 中从 *m* 到 *n* 的字符
StringReplacePart[*s*, *snew*, {{m_1, n_1}, {m_2, n_2}, …}]	用 *snew* 多次在 *s* 中进行替换
StringReplacePart[*s*, {$snew_1$, $snew_2$, …}, {{m_1, n_1}, {m_2, n_2}, …}]	用对应的 $snew_i$ 替换 *s* 中的子串

替换字符串的某一部分.

用 "XXX" 替换从2到6的字符.

```
In[12]:= StringReplacePart["abcdefgh", "XXX", {2, 6}]
Out[12]= aXXXgh
```

用字符串 "XXX" 进行了两次替换.

```
In[13]:= StringReplacePart["abcdefgh",
          "XXX", {{2, 3}, {5, -1}}]
Out[13]= aXXXdXXX
```

用不同的字符串进行了两次替换.

```
In[14]:= StringReplacePart["abcdefgh",
          {"XXX", "YYYY"}, {{2, 3}, {5, -1}}]
Out[14]= aXXXdYYYY
```

StringPosition[*s*, *sub*]	给出 *s* 中子串 *sub* 出现的起始和结束位置的列表
StringPosition[*s*, *sub*, *k*]	仅给出 *s* 中 *sub* 前 *k* 次出现的位置列表
StringPosition[*s*, {sub_1, sub_2, ...}]	给出 sub_i 中任一个出现的位置

找出子字符串的位置.

用 StringPosition 能给出一个字符串中一个子串出现的位置. StringPosition 的返回值是一个列表，列表中的每一个元素对应于子串的一次出现，它指出了子串的起始和结束位置. 这些列表的形式与 StringTake、StringDrop 和

这里给出子串 "abc" 出现位置的列表.

```
In[15]:= StringPosition["abcdabcdaabcabcd", "abc"]
Out[15]= {{1, 3}, {5, 7}, {10, 12}, {13, 15}}
```

这里仅给出第一个 "abc" 出现的位置.

```
In[16]:= StringPosition["abcdabcdaabcabcd", "abc", 1]
Out[16]= {{1, 3}}
```

这里给出了字符串 "abc" 和 "cd" 出现的位置. 默认地，它们在位置上有重叠.

```
In[17]:= StringPosition["abcdabcdcd", {"abc", "cd"}]
Out[17]= {{1, 3}, {3, 4}, {5, 7}, {7, 8}, {9, 10}}
```

这里没有包括重叠.

```
In[18]:= StringPosition["abcdabcdcd",
           {"abc", "cd"}, Overlaps -> False]
Out[18]= {{1, 3}, {5, 7}, {9, 10}}
```

StringCount[*s*, *sub*]	计算在 *s* 中 *sub* 的出现次数
StringCount[*s*, {sub_1, sub_2, …}]	计算任意 sub_i 出现的次数
StringFreeQ[*s*, *sub*]	检测 *s* 是否不包括 *sub*
StringFreeQ[*s*, {sub_1, sub_2, …}]	检测 *s* 是否不包括所有 sub_i

子串的检测.

这里计算任一子串的出现次数，默认地，不包括重叠.

```
In[19]:= StringCount["abcdabcdcd", {"abc", "cd"}]
Out[19]= 3
```

StringReplace[*s*, *sb*->*sbnew*]	在 *s* 中用 *sbnew* 替换 *sb*
StringReplace[*s*, {sb_1->$sbnew_1$, sb_2->$sbnew_2$, ...}]	用对应的 $sbnew_i$ 替换 sb_i
StringReplace[*s*, *rules*, *n*]	进行至多 *n* 次替换
StringReplaceList[*s*, *rules*]	给出使用每个单一替换所得到的字符串列表
StringReplaceList[*s*, *rules*, *n*]	给出至多 *n* 结果

根据规则替换子串.

用字符串 XX 替换所有的 a.

```
In[20]:= StringReplace["abcdabcdaabcabcd", "a" -> "XX"]
Out[20]= XXbcdXXbcdXXXXbcXXbcd
```

用 Y 替换 abc，用 XXX 替换 d.

```
In[21]:= StringReplace["abcdabcdaabcabcd",
          {"abc" -> "Y", "d" -> "XXX"}]
Out[21]= YXXXYXXXaYYXXX
```

第一个 cde 没有被替换，因为它与 abc 重复.

```
In[22]:= StringReplace["abcde abacde",
          {"abc" -> "X", "cde" -> "Y"}]
Out[22]= Xde abaY
```

StringReplace 可以对一个字符串从左到右扫描进行可能的替换，并返回得到的新字符串. 然而，有时候，查看所有可能的单一替换的结果是有用的. 用户可以使用 StringReplaceList 获得所有结果的列表.

这里给出替换每个 a 得到的结果列表.

```
In[23]:= StringReplaceList["aaaaa", "a" -> "X"]
Out[23]= {Xaaaa, aXaaa, aaXaa, aaaXa, aaaaX}
```

这里显示了所有可能的单一替换的结果.

```
In[24]:= StringReplaceList["abcde abacde",
          {"abc" -> "X", "cde" -> "Y"}]
Out[24]= {Xde abacde, abY abacde, abcde abaY}
```

StringSplit[*s*]	把 *s* 根据空格符分界分成子串组
StringSplit[*s*, *del*]	在分界符 *del* 的位置上分组
StringSplit[*s*, {del_1, del_2, …}]	在任意 del_i 的位置上分组
StringSplit[*s*, *del*, *n*]	分成至多 *n* 个子串组

字符串分组.

在每个空格处对字符串进行分组.

```
In[25]:= StringSplit["a b::c d::e f g"]
Out[25]= {a, b::c, d::e, f, g}
```

在每个 "::" 位置上分组.

```
In[26]:= StringSplit["a b::c d::e f g", "::"]
Out[26]= {a b, c d, e f g}
```

在每个冒号或空格位置上分组.

```
In[27]:= StringSplit["a b::c d::e f g", {":", " "}]
Out[27]= {a, b, , c, d, , e, f, g}
```

StringSplit[*s*, *del*->*rhs*]	在每个分解符位置上插入 *rhs*
StringSplit[*s*, {del_1->rhs_1, del_2->rhs_2, ...}]	在对应的 del_i 位置上插入 rhs_i

替换分界符来对字符串进行分组.

在每个 :: 分界符位置上插入 {x, y}.

```
In[28]:= StringSplit["a b::c d::e f g", "::" -> {x, y}]
Out[28]= {a b, {x, y}, c d, {x, y}, e f g}
```

Sort[{s_1, s_2, s_3, ...}]	对字符串列表排序

对字符串排序.

Sort 根据标准字典顺序对字符串进行排序.

```
In[29]:= Sort[{"cat", "fish", "catfish", "Cat"}]
Out[29]= {cat, Cat, catfish, fish}
```

StringTrim[s]	从 s 的开头和结尾删除空格
StringTrim[s, *patt*]	从开头和结尾删除与 *patt* 匹配的子串

从每个字符串的末尾删除空格.

```
In[30]:= StringTrim["   abcabc   "] // FullForm
Out[30]//FullForm=
"abcabc"
```

SequenceAlignment[s_1, s_2]	寻找 s_1 和 s_2 的最优对齐法

寻找两个字符串的最优对齐法.

```
In[31]:= SequenceAlignment[
          "abcXabcXabc", "abcYabcYabc"]
Out[31]= {abc, {X, Y}, abc, {X, Y}, abc}
```

相关教程

字符串中的字符

`Characters["string"]`	将字符串转换为一个字符列表
`StringJoin[{"c1","c2",…}]`	将一个字符集合转换为一个字符串

字符串与字符集合的转换.

这里给出字符串中的字符.

```
In[1]:= Characters["A string."]
Out[1]= {A,  , s, t, r, i, n, g, .}
```

对这一字符列表可以进行标准的列表操作.

```
In[2]:= RotateLeft[%, 3]
Out[2]= {t, r, i, n, g, ., A,  , s}
```

`StringJoin` 将给定的字符列表转换为字符串.

```
In[3]:= StringJoin[%]
Out[3]= tring.A s
```

DigitQ[*string*]	测试串中的所有字符是否是数字
LetterQ[*string*]	测试串中的所有字符是否是字母
UpperCaseQ[*string*]	测试串中的所有字符是否是大写字母
LowerCaseQ[*string*]	测试串中的所有字符是否是小写字母

测试字符串中的字符.

串中的所有字符是字母.

```
In[4]:= LetterQ["Mixed"]
Out[4]= True
```

串中的有些字符不是大写，所有测试结果为 False.

```
In[5]:= UpperCaseQ["Mixed"]
Out[5]= False
```

ToUpperCase[*string*]	产生一个所有字母都是大写的字符串
ToLowerCase[*string*]	产生一个所有字母都是小写的字符串

大小写之间的转换.

将所有的字母换成大写.

```
In[6]:= ToUpperCase["Mixed Form"]
Out[6]= MIXED FORM
```

CharacterRange["c_1","c_2"]	产生一个从 c_1 到 c_2 的字符列表

给出字符的范围.

产生一个小写字母顺序排列的集合.

```
In[7]:= CharacterRange["a", "h"]
Out[7]= {a, b, c, d, e, f, g, h}
```

这里给出了一个大写字母的列表.

```
In[8]:= CharacterRange["T", "Z"]
Out[8]= {T, U, V, W, X, Y, Z}
```

这里给出了一些数字.

```
In[9]:= CharacterRange["0", "7"]
Out[9]= {0, 1, 2, 3, 4, 5, 6, 7}
```

CharacterRange 对任何自然顺序的字符范围将给出有意义的结果，CharacterRange 利用 ***Mathematica*** 内部指定的字符代码进行处理.

这里显示了 ***Mathematica*** 代码所定义的字符顺序.

```
In[10]:= CharacterRange["T", "e"]
Out[10]= {T, U, V, W, X, Y, Z, [,
          \, ], ^, _, `, a, b, c, d, e}
```

相关指南

- 字符运算

相关教程

- 字符串和字符

教程专集

- **Core Language**

字符串模式

字符串操作函数如 StringReplace 的一个重要特征是它们不仅能够处理由字母组成的字符串，也能处理由字符串集合组成的模式.

这里使用 X 替代 b 或者 c.

```
In[1]:= StringReplace["abcd abcd", "b" | "c" -> "X"]
Out[1]= aXXd aXXd
```

这里使用 u 替代任何字符.

```
In[2]:= StringReplace["abcd abcd", _ -> "u"]
Out[2]= uuuuuuuuu
```

可以使用包含与 ***Mathematica*** 符号模式对象相混合的普通字符串的 *字符串表达式* 来指明字符串模式.

s_1~~s_2~~... 或 StringExpression[s_1, s_2, ...]

由字符串和模式对象组成的序列

字符串表达式.

这是表示后面带有任意单个字符的字符串 ab 的字符串表达式.

```
In[3]:= "ab" ~~ _
Out[3]= ab ~~ _
```

这里对字符串模式的每次出现进行替换.

```
In[4]:= StringReplace["abc abcb abdc", "ab" ~~ _ -> "X"
Out[4]= X Xb Xc
```

StringMatchQ["s", patt]	测试 "s" 是否与 patt 匹配
StringFreeQ["s", patt]	测试 "s" 是否不含有与 patt 匹配的子串
StringCases["s", patt]	给出与 patt 匹配的子串 "s" 的列表
StringCases["s", lhs->rhs]	用 rhs 替换每个 lhs
StringPosition["s", patt]	给出与 patt 匹配的子串的位置列表
StringCount["s", patt]	计算有多少子串与 patt 匹配
StringReplace["s", lhs->rhs]	替换与 lhs 匹配的每个子串
StringReplaceList["s", lhs->rhs]	给出所有替换 lhs 的方法列表
StringSplit["s", patt]	在每个与 patt 匹配的子串处分解 s
StringSplit["s", lhs->rhs]	在 lhs 处分解，并在该位置插入 rhs

支持字符串模式的函数.

这里给出在字符串中出现的模式的所有可能性.

```
In[5]:= StringCases["abc abcb abdc", "ab" ~~ _]
Out[5]= {abc, abc, abd}
```

这里给出在字符串 "ab" 后出现的每个字符.

```
In[6]:= StringCases["abc abcb abdc", "ab" ~~ x_ -> x]
Out[6]= {c, c, d}
```

这里给出字符串中所有相同字符组成的对.

```
In[7]:= StringCases["abbcbccaabbabccaa", x_ ~~ x_]
Out[7]= {bb, cc, aa, bb, cc, aa}
```

可以在字符串模式中使用所有标准 ***Mathematica*** 模式对象. _ 总是表示单个字符. __ 表示一个或多个字符组成的序列.

_ 表示任意单个字符.

```
In[8]:= StringReplace[
          {"ab", "abc", "abcd"}, "b" ~~ _ -> "X"]
Out[8]= {ab, aX, aXd}
```

__ 表示一个或多个字符组成的任意序列.

```
In[9]:= StringReplace[
          {"ab", "abc", "abcd"}, "b" ~~ __ -> "X"]
Out[9]= {ab, aX, aX}
```

___ 表示0个或多个字符组成的任意序列.

```
In[10]:= StringReplace[
           {"ab", "abc", "abcd"}, "b" ~~ ___ -> "X"]
Out[10]= {aX, aX, aX}
```

"*string*"	由字符组成的字符串
_	任意单个字符
__	由一个或多个字符组成的任意序列
___	由**0**个或多个字符组成的任意序列
*x*_ , *x*__ , *x*___	名称有 *x* 的子串
x:*pattern*	名称为 *x* 的模式
pattern..	重复**1**次或多次的模式
pattern...	重复**0**次或多次的模式
{*patt*$_1$, *patt*$_2$, ...} 或者 *patt*$_1$ | *patt*$_2$ | ...	与至少一个 *patt*$_i$ 匹配的模式
patt/;*cond*	*cond* 等于 True 的模式
pattern?*test*	对每个字符 *test* 等于 True 的模式
Whitespace	空格字符组成的序列
NumberString	数字字符串
charobj	表示字符类的对象（参见下面的讨论）
RegularExpression["*regexp*"]	与规则表达式匹配的子串

在字符串模式中的对象.

在冒号或分号处分解.

```
In[11]:= StringSplit["a:b;c:d", ":" | ";"]
Out[11]= {a, b, c, d}
```

只包含 a 或 b 的子序列.

```
In[12]:= StringCases["aababbcccdbaa", ("a" | "b") ..]
Out[12]= {aababb, baa}
```

在字符串模式中另外的方法可以由列表形式给出.

```
In[13]:= StringCases["aababbcccdbaa", {"a", "b"} ..]
Out[13]= {aababb, baa}
```

可以使用标准 ***Mathematica*** 结构，如 `Characters["c1c2..."]` 和 `CharacterRange["c1", "c2"]` 来产生在字符串模式中应用的其它字符列表.

这里给出字符列表.

```
In[14]:= Characters["aeiou"]
Out[14]= {a, e, i, o, u}
```

这里替换元音字符.

```
In[15]:= StringReplace["abcdefghijklm",
          Characters["aeiou"] -> "X"]
Out[15]= XbcdXfghXjklm
```

这里在 "A" 到 "H" 的范围内给出字符.

```
In[16]:= CharacterRange["A", "H"]
Out[16]= {A, B, C, D, E, F, G, H}
```

能字符的一些通用类的符号规定.

{"c_1","c_2",...}	任意 "c_i"
Characters["c_1c_2..."]	任意 "c_i"
CharacterRange["c_1","c_2"]	在 "c_1" 到 "c_2" 范围内的任意字符
DigitCharacter	数字 0–9
LetterCharacter	字母
WhitespaceCharacter	空白、换行、tab或其它空格字符
WordCharacter	字母或数字
Except[p]	除了与 p 匹配的任意字符

字符类的规定.

这里在一个字符串中选出数字字符.

```
In[17]:= StringCases["a6;b23c456;", DigitCharacter]
Out[17]= {6, 2, 3, 4, 5, 6}
```

这里选出除了数字以外的所有字符.

```
In[18]:= StringCases["a6;b23c456;",
          Except[DigitCharacter]]
Out[18]= {a, ;, b, c, ;}
```

这里选出所有一个或多个数字组成的子序列.

```
In[19]:= StringCases["a6;b23c456", DigitCharacter ..]

Out[19]= {6, 23, 456}
```

所得结果是字符串.

```
In[20]:= InputForm[%]

t[20]//InputForm=
         {"6", "23", "456"}
```

这里把字符串转换为数字.

```
In[21]:= ToExpression[%] + 1

Out[21]= {7, 24, 457}
```

字符串模式经常用作从文本数据的字符串中提取结构的方法. 通常通过使用与所对应结构的不同部分匹配的字符串模式的不同部分来实现.

这里选出每个后面带有一个数字的 =.

```
In[22]:= StringCases["a1=6.7, b2=8.87",
          "=" ~~ NumberString]

Out[22]= {=6.7, =8.87}
```

这里给出单个的数字.

```
In[23]:= StringCases["a1=6.7, b2=8.87",
          "=" ~~ x : NumberString -> x]

Out[23]= {6.7, 8.87}
```

这里从字符串中提取“变量”和“数值”.

```
In[24]:= StringCases["a1=6.7, b2=8.87",
          v : WordCharacter .. ~~
            "=" ~~ x : NumberString -> {v, x}]

Out[24]= {{a1, 6.7}, {b2, 8.87}}
```

ToExpression 把它们转换为普通的符号和数字.

```
In[25]:= ToExpression[%]^2

Out[25]= {{a1^2, 44.89}, {b2^2, 78.6769}}
```

在许多情况下，文本数据可能包含空白、换行或**tab**这些可以被认作“空格”可以忽略的序列. 在 ***Mathematica*** 中，符号Whitespace 表示任何这种类型的序列.

从字符串中删除所有空格.

```
In[26]:= StringReplace["aa  b  cc  d", Whitespace -> ""]

Out[26]= aabccd
```

使用单个逗号替换所有空格序列.

```
In[27]:= StringReplace["aa  b  cc  d", Whitespace -> ","]

Out[27]= aa,b,cc,d
```

字符串模式通常应用于一个给定字符串中任何位置出现的子串. 然而，有时，指定只能用于特定位置子串的模式是方便的. 这可以通过在字符串模式中包含符号如 StartOfString 来实现.

StartOfString	整个字符串的开头
EndOfString	整个字符串的末尾
StartOfLine	行的开头
EndOfLine	行的末尾
WordBoundary	在词字符和其它之间的分界
Except[StartOfString], etc.	除了特定位置如 StartOfString 等的其它任何位置

表示字符串中特定位置的结构.

这里在一个字符串中替换所有出现的 "a".

```
In[28]:= StringReplace[{"abc", "baca"}, "a" -> "XX"]
Out[28]= {XXbc, bXXcXX}
```

这里仅当 "a" 在一个字符串开头处时替换它.

```
In[29]:= StringReplace[{"abc", "baca"},
          StartOfString ~~ "a" -> "XX"]
Out[29]= {XXbc, baca}
```

这里替换所有出现的子串 "the".

```
In[30]:= StringReplace["the others", "the" -> "XX"]
Out[30]= XX oXXrs
```

这里只替换在两边都具有词边界的子串.

```
In[31]:= StringReplace["the others",
          WordBoundary ~~ "the" ~~ WordBoundary -> "XX"]
Out[31]= XX others
```

字符串模式只允许同种 /; 和作为普通 ***Mathematica*** 模式的其它条件.

这里给出字符串中不等连续字符的情况.

```
In[32]:= StringCases["aaabbcaaaabaaa", x_ ~~ y_ /; x != y
Out[32]= {ab, bc, ab}
```

当在一个字符串模式中给出一个对象，如 x__ 或 e .., ***Mathematica*** 自动假设用户想要它匹配最长可能字符序列. 然而，有时，用户可能想要匹配的是最短可能字符序列. 这可以使用 Shortest[p] 指明.

Longest[p]	对 p 的最长一致匹配（默认）
Shortest[p]	对 p 的最短一致匹配

表示最长和最短匹配的对象.

默认情况下，字符串模式与最长可能字符序列匹配.

```
In[33]:= StringCases[
          "-(a)--(bb)--(c)-", "(" ~~ __ ~~ ")"]
Out[33]= {(a)--(bb)--(c)}
```

Shortest 指明用户要寻找的是最短可能匹配.

```
In[34]:= StringCases["-(a)--(bb)--(c)-",
          Shortest["(" ~~ __ ~~ ")"]]

Out[34]= {(a), (bb), (c)}
```

Mathematica 默认情况下认为字符 "X" 和 "x" 是不同的. 但是通过在字符串操作中设置选项IgnoreCase -> True，可以让 ***Mathematica*** 同等对待大小写字母.

IgnoreCase->True　　同等对待大小写字母

指定与大小写无关的字符串操作.

这里替换所有出现的 "the"，与大小写无关.

```
In[35]:= StringReplace["The cat in the hat.",
          "the" -> "a", IgnoreCase -> True]

Out[35]= a cat in a hat.
```

在一些字符串操作中，用户可能要指明是否包含子串之间的重叠. 默认情况下，StringCases 和 StringCount 不包含重叠，但是 StringPosition 包含重叠.

这里选出连续字符的对，默认情况下忽略重叠.

```
In[36]:= StringCases["abcdefg", _ ~~ _]

Out[36]= {ab, cd, ef}
```

这里包含重叠.

```
In[37]:= StringCases["abcdefg", _ ~~ _, Overlaps -> True
Out[37]= {ab, bc, cd, de, ef, fg}
```

StringPosition 默认情况下包含重叠.

```
In[38]:= StringPosition["abcdefg", _ ~~ _]
Out[38]= {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {6, 7}}
```

Overlaps->All	包含所有重叠
Overlaps->True	包含在每个位置开始的至多一个重叠
Overlaps->False	不包含所有重叠

处理字符串中重叠的选项.

这里只产生单个匹配.

```
In[39]:= StringCases["abcd", __, Overlaps -> False]
Out[39]= {abcd}
```

这里产生连续的重叠匹配.

```
In[40]:= StringCases["abcd", __, Overlaps -> True]
Out[40]= {abcd, bcd, cd, d}
```

这里包含所有可能的重叠匹配.

```
In[41]:= StringCases["abcd", __, Overlaps -> All]
Out[41]= {abcd, abc, ab, a, bcd, bc, b, cd, c, d}
```

相关指南

- 字符串模式

相关教程

- 规则表达式
- 字符串模式的使用
- 字符串和字符

教程专集

- **Core Language**

规则表达式

Mathematica 的一般模式提供了进行字符串操作的强大方法. 但是如果对专门的字符串操作语言熟悉的话，有时可能会发现使用*规则表达式* 记号指明字符串模式是方便的. 在 ***Mathematica*** 中可以使用 `RegularExpression` 对象实现.

`RegularExpression[` 由 "*regex*" 指定的规则表达式 "*regex*" `]`

在 ***Mathematica*** 中使用规则表达式记号.

这里取代所有出现的 a 或 b.

```
In[1]:= StringReplace["abcd acbd",
         RegularExpression["[ab]"] -> "XX"]

Out[1]= XXXXcd XXcXXd
```

这里使用 ***Mathematica*** 的一般字符串模式指明同样的操作.

```
In[2]:= StringReplace["abcd acbd", "a" | "b" -> "XX"]

Out[2]= XXXXcd XXcXXd
```

可以把规则表达式与一般模式混合使用.

```
In[3]:= StringReplace["abcd acbd",
         RegularExpression["[ab]"] ~~ _ -> "YY"]

Out[3]= YYcd YYYY
```

RegularExpression 在 ***Mathematica*** 中支持所有标准规则表达式结构.

c	字母字符 c
.	除了新的一行以外的任何字符
$[c_1\ c_2\ \ldots]$	任何 c_i 字符
$[c_1\text{-}c_2]$	在 c_1-c_2 范围内的任何字符
$[\hat{}c_1\ c_2\ \ldots]$	除了 c_i 的任何字符
$p*$	重复**0**次或多次的 p
$p+$	重复**1**次或多次的 p
$p?$	出现**0**次或**1**次的 p
$p\,\{m,n\}$	重复 m 到 n 次之间的 p
$p*?,\ p+?,\ p??$	最短的符合匹配条件的一致字符串
$(p_1\ p_2\ \ldots)$	与序列 $p_1\ p_2\ \ldots$ 匹配的字符串
$p_1 \mid p_2$	与 p_1 或 p_2 匹配的字符串

在 ***Mathematica*** 规则表达式中的基本结构.

这里寻找与特定规则表达式匹配的子串.

```
In[4]:= StringCases["abcddbbbacbbaa",
          RegularExpression["(a|bb)+"]]
Out[4]= {a, bb, a, bbaa}
```

这里使用 ***Mathematica*** 一般字符串模式做同样的操作.

```
In[5]:= StringCases["abcddbbbacbbaa", ("a" | "bb") ..]
Out[5]= {a, bb, a, bbaa}
```

在许多规则表达式结构和基本的 ***Mathematica*** 一般字符串模式结构之间存在紧密的对应关系.

.	_ (严格上， Except["\n"])
[c_1 c_2 ...]	Characters[" c_1 c_2 ..."]
[c_1-c_2]	CharacterRange["c_1","c_2"]
[^c_1 c_2 ...]	Except[Characters[" c_1 c_2 ..."]]
p*	p...
p+	p..
p?	p|""
p*?, p+?, p??	Shortest[p...],...
(p_1 p_2 ...)	(p_1~~p_2~~...)
p_1 | p_2	p_1 | p_2

在规则表达式和一般字符串模式结构之间的对应关系.

正如在 ***Mathematica*** 一般字符串模式中一样，在规则表达式中也有对不同的通用字符类的特殊记号.注意，用户需要使用 \ \ 符号来在 ***Mathematica*** 规则表达式字符串中输入大多数记号.

\\d	数字 0–9 （DigitCharacter ）
\\D	非数字 （Except[DigitCharacter]）
\\s	空格、换行、tab 或其他空格字符 （WhitespaceCharacter）
\\S	非空格字符 （Except[WhitespaceCharacter] ）
\\w	词字符（字母、数字或 _）（WordCharacter）
\\W	非词字符 （Except[WordCharacter]）
[[:*class*:]]	位于命名类中的字符
[^[:*class*:]]	不位于命名类中的字符

字符类的规则表达式记号.

这里给出带有数字字符的每个 a 的子串.

```
In[6]:= StringCases["a10b6a77a3a#",
         RegularExpression["a\\d+"]]

Out[6]= {a10, a77, a3}
```

这里使用 ***Mathematica*** 的一般字符串模式做同样的事情.

```
In[7]:= StringCases["a10b6a77a3a#",
         "a" ~~ DigitCharacter ..]

Out[7]= {a10, a77, a3}
```

Mathematica 支持标准 **POSIX** 字符类 alnum、alpha、

punct、space、upper、word 和 xdigit.

这里寻找子序列中的大写字母.

```
In[8]:= StringCases["AaBBccDDeefG",
         RegularExpression["[[:upper:]]+"]]

Out[8]= {A, BB, DD, G}
```

这里做同样的事情.

```
In[9]:= StringCases["AaBBccDDeefG",
         CharacterRange["A", "Z"] ..]

Out[9]= {A, BB, DD, G}
```

^	字符串的开头（StartOfString）
$	字符串的末尾（EndOfString）
\\b	词边界（WordBoundary）
\\B	除了词边界的任何位置（Except[WordBoundary]）

字符串中位置的规则表达式记号.

在一般的 ***Mathematica*** 模式中，可以使用例如 x_ 和 $x:patt$ 等结构来对匹配的对象给出任意的名字. 在规则表达式中，可以使用如下方法处理序列号：在一个规则表达式中第 n 个带括号的模式对象（p）在模式内为 \\$n$，而在模式外为$$n$.

这里寻找所出现的相同字母对.

```
In[10]:= StringCases["aaabcccabbaacba",
          RegularExpression["(.)\\1"]]

Out[10]= {aa, cc, bb, aa}
```

现在使用 ***Mathematica*** 的一般字符串模式做同样的事情.

```
In[11]:= StringCases["aaabcccabbaacba", x_ ~~ x_]

Out[11]= {aa, cc, bb, aa}
```

这里 $1 指的是与（.）匹配的字母.

```
In[12]:= StringCases["aaabcccabbaacba",
          RegularExpression["(.)\\1"] -> "$1"]

Out[12]= {a, c, b, a}
```

这是 ***Mathematica*** 模式版本.

```
In[13]:= StringCases["aaabcccabbaacba", x_ ~~ x_ -> x]

Out[13]= {a, c, b, a}
```

相关指南

- 字符串模式

相关教程

- 字符串模式
- 字符串模式的使用
- 字符串和字符

特殊字符

除了标准键盘上的字符外，***Mathematica*** 的字符串中还可以包含 ***Mathematica*** 所支持的一些特殊字符.

含有特殊字符的字符串.

```
In[1]:= "α⊕β⊕..."
Out[1]= α⊕β⊕...
```

可以与其它字符串一样对这一字符串操作.

```
In[2]:= StringReplace[%, "⊕" -> " ⊙⊙ "]
Out[2]= α ⊙⊙ β ⊙⊙ ...
```

这是字符串中字符的列表.

```
In[3]:= Characters[%]
Out[3]= {α,  , ⊙, ⊙,  , β,  , ⊙, ⊙,  , ...}
```

在 ***Mathematica*** 的笔记本中，像 α 等特殊字符直接显示. 但是在用文本界面时，容易显示的字符就是在键盘上所出现的字符.

于是，***Mathematica*** 在这种情况下就用与特殊字符相接近的字

Mathematica 笔记本中用 StandardForm，特殊字符可以直接显示.

```
In[4]:= "Lamé ⟶ αβ+"
Out[4]= Lamé ⟶ αβ+
```

在 OutputForm 中，特殊字符在可能时就用相近的一般字符代替.

```
In[5]:= % // OutputForm
t[5]//OutputForm=
        Lamé ⟶ αβ+
```

在 InputForm 中，***Mathematica*** 总使用特殊字符的全名. 这意味着，在文件或外部程序中使用特殊字符时，它们习惯上就用一列一般字符集合来表示.

Mathematica 中特殊字符的统一表示法对它们在不同计算机系统中的使用是非常重要的.

在 InputForm 中，所有特殊字符的全名总被明确给出.

```
In[6]:= "Lamé ⟶ αβ+" // InputForm
t[6]//InputForm=
        "Lamé ⟶ αβ+"
```

a	一个字符
\[*Name*]	使用全名的字符
\"	" 将被包含在字符串中
\\	\ 将被包含在字符串中

在字符串中输入字符的方式.

如果字符串中有 " 或 \，必须在它们之前使用 \.

```
In[7]:= "Strings can contain
          \"quotes\" and \\ characters."
Out[7]= Strings can contain "quotes" and \ characters
```

\\ 按原样产生一个 \，而不是特殊字符 α 的组成部分.

```
In[8]:= "\\[Alpha] is α."
Out[8]= \[Alpha] is α.
```

将字符串分解成字符列表.

```
In[9]:= Characters[%]
Out[9]= {\, [, A, l, p, h, a, ],  , i, s,  , α, .}
```

这里给出了 α 全名的字符列表.

```
In[10]:= Characters[ToString[FullForm["α"]]]
Out[10]= {", \, [, A, l, p, h, a, ], "}
```

这里产生了一个实际包含 α 的字符串.

```
In[11]:= ToExpression["\"\\[" <> "Alpha" <> "]\""]
```

```
Out[11]= α
```

相关教程

- 字符串和字符

教程专集

- **Core Language**

字符串中的换行和 Tab

\n	在字符串中包含新的一行
\t	一个 **tab** 包含在字符串中

换行和 **tab** 的明确表示.

这里显示**2**行.

```
In[1]:= "First line.\nSecond line."
```

```
Out[1]= First line.
        Second line.
```

在 InputForm 中，有一个 \n 来表示新的一行.

```
In[2]:= InputForm[%]

t[2]//InputForm=
        "First line.\nSecond line."
```

Mathematica 在一个字符串中保留输入的换行.

```
In[3]:= "A string on
        two lines."

Out[3]= A string on
        two lines.
```

在该字符串中，有一个新的换行.

```
In[4]:= InputForm[%]

t[4]//InputForm=
        "A string on \ntwo lines."
```

在一个行的末尾使用单个\，*Mathematica* 就忽略换行.

```
In[5]:= "A string on \
        one line."

Out[5]= A string on one line.
```

应该认识到，虽然有可能通过原来的 **tab** 和换行获得 *Mathematica* 输出的一些格式，但这基本上在很少情况下是个好办法. 通常，一个更好的办法是使用高级的 *Mathematica* 原始格式，可参见 "面向字符串的输出格式"，"数字的输出格式"，和 "表与矩阵". 这些原始格式将总是产生一致的输出，不论在一个特定的设备中**tab**设置的位置.

在包含换行符的字符串中，文本总是左对齐.

```
In[6]:= {"Here is\na string\non several lines.",
         "Here is\nanother"}

Out[6]= {Here is
        a string
        on several lines., Here is
        another}
```

前端格式结构 Column 给出了更多的控制. 这里文本是右对齐.

```
In[7]:= Column[
         {"First line", "Second", "Third"}, Right]

        First line
Out[7]=     Second
             Third
```

这里文本居中.

```
In[8]:= Column[
         {"First line", "Second", "Third"}, Center]

        First line
Out[8]=   Second
           Third
```

相关教程

- 字符串和字符

教程专集

- **Core Language**

字符代码

ToCharacterCode["*string*"]	给出一个字符串中字符代码的列表
FromCharacterCode[*n*]	由代码产生字符
FromCharacterCode[{n_1, n_2, ...}]	由一个字符代码列表产生一个字符串

字符与代码的转换.

Mathematica 对字符串中的每个字符指定唯一的代码. 该代码在内部被用来代表这个字符.

给出字符串中字符的代码.

```
In[1]:= ToCharacterCode["ABCD abcd"]
Out[1]= {65, 66, 67, 68, 32, 97, 98, 99, 100}
```

FromCharacterCode 给出原来的字符串.

```
In[2]:= FromCharacterCode[%]
Out[2]= ABCD abcd
```

特殊字符也有代码.

```
In[3]:= ToCharacterCode["α⊕Γ⊖∅"]
Out[3]= {945, 8853, 915, 8854, 8709}
```

CharacterRange["c_1","c_2"]	用相连的字符代码产生一个字符集合

产生字符序列.

这里给出部分英文字母.

```
In[4]:= CharacterRange["a", "k"]
Out[4]= {a, b, c, d, e, f, g, h, i, j, k}
```

这里是希腊字母.

```
In[5]:= CharacterRange["α", "ω"]
Out[5]= {α, β, γ, δ, ε, ζ, η, θ, ι, κ, λ, μ,
          ν, ξ, ο, π, ρ, ς, σ, τ, υ, φ, χ, ψ, ω}
```

Mathematica 给特殊字符指定了形如 \[Alpha] 的名称. 这就可以通过这些字符的名称调用它们，而不需要知道它们的代码.

通过特殊字符的代码产生的字符串.

```
In[6]:= FromCharacterCode[{8706, 8709, 8711, 8712}]
Out[6]= ∂∅∇∈
```

可以用这些字符的名称来调用它们，而不需要知道它们的代码.

```
In[7]:= FullForm[%]
t[7]//FullForm=
        "\[PartialD]\[EmptySet]\[Del]\[Element]"
```

对常用的数学符号和标准的欧洲语言中的符号，***Mathematica*** 都为

符，但没有明确给这些符号命名，仅给出了标准的字符代码.

这里是包含日语字符的字符串.

```
In[8]:= "数学"
Out[8]= 数学
```

在 FullForm 中，这些字符用标准字符代码表示. 这些字符代码是十六进制数字.

```
In[9]:= FullForm[%]
t[9]//FullForm=
        "\:6570\:5b66"
```

Mathematica 中的笔记本中输入一个特定的字符后，***Mathematica*** 将自动找出这个字符的代码.

有时用代码直接输入字符是方便的.

\.*nn*	具有十六进制代码 *nn* 的字符
\:*nnnn*	具有十六进制代码 *nnnn* 的字符

用字符代码直接输入字符的方式.

对于代码在 **256** 之内的字符，可以用 \.*nn* 输入. 对于代码在 **256** 之上的字符，必须用 \:*nnnn* 输入. 注意，在任何情况下必须给出确定位数的八进制或十六进制字符，必要时在前面添**0**.

这里给出几个字符的十六进制代码.

```
In[10]:= BaseForm[ToCharacterCode["Aàα𐎗"], 16]
t[10]//BaseForm=
        {41_16, e0_16, 3b1_16, 2135_16}
```

这里用代码输入字符. 注意，α 的代码的前一位加个0.

```
In[11]:= "Aàα𐎗"
Out[11]= Aàα𐎗
```

在给字符指定代码时，***Mathematica*** 遵循了与3个标准相容的原则：**ASCII**，**ISO Latin-1** 和 **Unicode**. **ASCII** 覆盖了所有美式英语键盘上出现的字符. **ISO Latin-1** 覆盖了在许多欧洲语言中出现的字符. **Unicode** 是一个更一般的标准，它为世界上语言和记号中使用的几万种字符定义了代码.

0–127 (**\.00–\.7f**)	**ASCII** 字符
1–31 (**\.01–\.1f**)	**ASCII** 控制字符
32–126 (**\.20–\.7e**)	可显示的 **ASCII** 字符
97–122 (**\.61–\.7a**)	小写英语字母
129–255 (**\.81–\.ff**)	**ISO Latin-1** 字符
192–255 (**\.c0–\.ff**)	在欧洲语言中出现的字母
0–59391 (**\:0000–\:e7ff**)	**Unicode** 标准共用字符
913–1009 (**\:0391–\:03f1**)	希腊字母
12288–35839 (**\:3000–\:8bff**)	中文、日语和朝鲜文字符
8450–8504 (**\:2102–\:2138**)	修改后在数学记号中使用的字母
8592–8677 (**\:2190–\:21e5**)	箭头
8704–8945 (**\:2200–\:22f1**)	数学符号及运算
64256–64300 (**\:fb00–\:fb2c**)	由 ***Mathematica*** 特殊定义的 **Unicode** 专用字符

Mathematica 中使用的部分字符代码的范围.

这里给出了所有可显示的 **ASCII** 字符.

```
In[12]:= FromCharacterCode[Range[32, 126]]
```

```
Out[12]=
          !"#$%&'()*+,-./0123456789:;<=>?@
          ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^
          _`abcdefghijklmnopqrstuvwxyz{|}~
```

这里是 **ISO Latin-1** 字母.

```
In[13]:= FromCharacterCode[Range[192, 255]]
```

```
Out[13]= ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×
         ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
```

这里是在数学记号中使用的一些特殊字符. 其中，空格块表示在当前字形中还没有的字符.

```
In[14]:= FromCharacterCode[Range[8704, 8750]]
```

```
Out[14]= "∀ ∂∃∄∅\[Laplace]∇∈∉ ∋∌∋ ∏∐∑−∓ /∖ ∘⋅√  ∝∞∟∠∡
          ∣ ∥∧∨∩∪∫  ∮"
```

这是几个日文字符.

```
In[15]:= FromCharacterCode[Range[30000, 30030]]
```

```
Out[15]= 田由甲申甴电甶男甸甹町画甼甽甾甿畀畀畎畋畅
          畝畇畈畎畔畋界畛畴
```

相关指南

- 本地化和国际化

相关教程

- 字符的原始代码
- 字符串和字符

教程专集

字符的原始代码

在 ***Mathematica*** 中可以用 \[Alpha] 等名称或 \:03b1 等十六进制编码来指代特殊字符. 当 ***Mathematica*** 输出一个文件时，其默认方式就是使用这些名称或代码.

但有时对一些特殊字符使用原始码很方便，这意味着不使用特殊字符的名称或十六进制编码，而用与计算机系统或字形相应的原始模式去表示这些特殊字符.

$CharacterEncoding=None	对所有特殊字符使用可显示的 **ASCII** 名称
$CharacterEncoding="*name*"	使用由 *name* 指定的字符的原始代码
$SystemCharacterEncoding	特定计算机系统的默认原始字符代码

设置字符的原始码.

当按下一个键或组合键时，计算机的操作系统就将一定位数的模式输送给 ***Mathematica***. 这个模式在 ***Mathematica*** 中怎样翻译为一个字符将依赖于所设置的字符编码.

Mathematica 的笔记本前端对所使用的字形将自动设置合适的字符编码. 但在文本界面或通过文件等途径使用 ***Mathematica*** 时，就需要明确地设置 $CharacterEncoding.

通过对 $CharacterEncoding 指定一个合适的值，就能使 ***Mathematica*** 处理不同编辑器或操作系统所产生的原始文本.

然而应该意识到，***Mathematica*** 中所使用的特殊字符的标准表示形式可以用于不同的计算机系统. 但涉及到原始字符编码的表示式却不能.

"PrintableASCII"	仅限于可显示的 **ASCII** 字符（默认）
"ASCII"	包括控制字符的所有 **ASCII** 字符
"ISOLatin1"	常用的西欧语言字符
"ISOLatin2"	中欧及东欧语言字符
"ISOLatin3"	另外的欧洲语言字符（如卡泰兰、土尔其）
"ISOLatin4"	其余的欧洲语言字符（如艾托尼亚、南普斯）
"ISOLatinCyrillic"	英语及苏黎士语言字符
"AdobeStandard"	**Adobe** 标准邮电码
"MacintoshRoman"	**Macintosh** 罗马字符编码
"WindowsANSI"	**Windows** 标准字形码
"Symbol"	符号字形编码
"ZapfDingbats"	**Zapf dingbats** 字形编码
"ShiftJIS"	日语 **shift-JIS**（**8** 位与 **16** 位混用）
"EUC"	扩展的 **Unix** 日语编码（**8** 位与 **16** 位混用）
"UTF-8"	**Unicode** 转换格式编码
"Unicode"	原始的 **16** 位 **Unicode** 模式

Mathematica 所支持的一些原始字符编码.

Mathematica 知道适应于不同计算机系统和不同语言的各种原始字符编码. 当指定了一种编码后，***Mathematica*** 就可以以原始形式给

的字符仍按标准的 ***Mathematica*** 全名或十六进制编码给出.

当写入或读入文本文件时，***Mathematica*** 内核可以使用您所指定的任何字符编码. 默认情况下，Put 和 PutAppend 对 ***Mathematica*** 语言文件从一个系统到另一个的可移植性产生 **ASCII** 表示法.

给文件 tmp 中写入一个字符串.

```
In[1]:= "a b c é α π  " >> tmp
```

特殊字符默认地用全名或十六进制数字表示.

```
In[2]:= Read["tmp", String]
Out[2]= "a b c \[EAcute] \[Alpha] \[Pi] \:2766"
```

Mathematica 支持八进制和十六进制原始字符编码. 在 "ISOLatin1" 编码中，所有字符由包含八进制的模式表示，而在 "ShiftJIS" 等编码中，一些字符就涉及到十六进制的编码.

大部分 ***Mathematica*** 支持的字符编码都将基本的 **ASCII** 作为一个子集. 这就可以在这些编码中按照一般方式给出***Mathematica*** 的常用输入，可以用 \[和 \: 序列指代特殊字符.

还有一些原始字符编码没有将基本的 **ASCII** 作为一个子集. 例如，"Symbol" 编码，在这种编码中，a 和 b 的常用编码用来代表 α 和 β.

这里给出了一些英文字母的 **ASCII** 字符码.

```
In[3]:= ToCharacterCode["abcdefgh"]
Out[3]= {97, 98, 99, 100, 101, 102, 103, 104}
```

在 "Symbol" 编码中，这些字符码被用来代表希腊字母.

```
In[4]:= FromCharacterCode[%, "Symbol"]
```

```
Out[4]= αβχδ∈φγη
```

`ToCharacterCode[ "string"]`	用标准 ***Mathematica*** 编码产生字符码
`ToCharacterCode[ "string","encoding"]`	用指定的编码产生字符码
`FromCharacterCode[ {n1,n2,...}]`	从 ***Mathematica*** 的标准代码产生字符
`FromCharacterCode[{n1,n2,...},"encoding"]`	用指定的编码产生字符

不同编码中字符码的处理.

这里给出 ***Mathematica*** 指定给字符的代码.

```
In[5]:= ToCharacterCode["abcéπ"]
```

```
Out[5]= {97, 98, 99, 233, 960}
```

对同样的字符用 **Macintosh** 罗马编码指定的代码.

```
In[6]:= ToCharacterCode["abcéπ", "MacintoshRoman"]
```

```
Out[6]= {97, 98, 99, 142, 185}
```

这里用 **Windows** 的标准编码给出了字符的代码，在这种编码中，\[Pi] 没有代码.

```
In[7]:= ToCharacterCode["abcéπ", "WindowsANSI"]
Out[7]= {97, 98, 99, 233, None}
```

Mathematica 内部使用的字符代码基于**Unicode.** 但是，作为默认状态，***Mathematica*** 外部总使用 \[*Name*] 或 \:*nnnn* 等简单的**ASCII**序列去代表特殊字符. 通过让 ***Mathematica*** 使用原始 "Unicode" 字符编码，就可以使 ***Mathematica*** 用十六进制 **Unicode** 形式去读写字符.

相关教程

- 字符代码
- 字符串和字符
- **Core Language**

字符串模式的使用

引言

一般字符串模式

规则表达式

RegularExpression 与 **StringExpression** 的比较

字符串操作函数

对于**Perl**用户

一些示例
有效匹配的技巧
实现细节

教程专集

- **Advanced String Patterns**
- **Core Language**

表达式的计算

计算原理
将表达式化为标准形式
属性
标准运算过程
非标准计算
模式、规则和定义中的计算
迭代函数的计算
条件
循环控制结构
在计算过程中收集表达式
计算的跟踪
计算堆栈

计算原理

Mathematica 最基本的功能是*计算*. 输入一个表达式后，***Mathematica*** 就计算出其结果，并返回该结果.

Mathematica 中的计算是通过一系列定义来实现的，有些定义是直接输入的，有些是 ***Mathematica*** 内部固有的.

因此，例如，***Mathematica*** 在计算表达式 6 + 7 时就调用内部的整数加法过程. 同样地，***Mathematica*** 计算代数表达式x - 3 x + 1 时就调用内部的化简过程. 当定义 x = 5 时，***Mathematica*** 就计算出 x - 3 x + 1 等于 -9.

Mathematica 中的两个核心的概念是*表达式* 和*运算*. "表达式" 节中讨论了用统一的方式使用表达式去处理不同的对象. 这一节将讨论怎样用统一的方式来看待 ***Mathematica*** 中的运算.

计算	$5+6 \longrightarrow 11$
化简	$x-3x+1 \longrightarrow 1-2x$
执行	$x=5 \longrightarrow 5$

运算的说明.

Mathematica 是一个*无限* 的*计算* 系统. 输入一个表达式后，***Mathematica*** 将一直用已知的定义来计算它，直到无定义可用时才停止.

先用 x2 来定义 x1，再给出 x2 的定义.

```
In[1]:= x1 = x2 + 2; x2 = 7
Out[1]= 7
```

用到 x1 时，***Mathematica*** 将用所有的定义来计算从而给出结果.

```
In[2]:= x1
Out[2]= 9
```

这是阶乘函数的一个递推定义.

```
In[3]:= fac[1] = 1; fac[n_] := n fac[n - 1]
```

求 fac[10] 时，***Mathematica*** 将一直使用所给出的定义，直到结果不再改变为止.

```
In[4]:= fac[10]
Out[4]= 3 628 800
```

Mathematica 使用完已知的定义后给出结果. 有时结果是一个

Mathematica 使用了和的内部定义来化简表达式，但由于 f[3] 无定义，故该项仍以符号形式出现.

```
In[5]:= f[3] + 4 f[3] + 1
Out[5]= 1 + 5 f[3]
```

Mathematica 的原则是将定义使用到结果不再改变为止. 这意味着，把 ***Mathematica*** 的最后结果再输入时，输出不再改变. （在"无穷计算的控制" 中讨论了一些以上情形不会出现的微妙情况.)

输入一个 ***Mathematica*** 的结果后，得到的是同样的表达式.

```
In[6]:= 1 + 5 f[3]
Out[6]= 1 + 5 f[3]
```

在任何时候，***Mathematica*** 仅使用当时已知的定义. 添加的一些定义 ***Mathematica*** 随后就可以使用. 此时，***Mathematica*** 给出的结果可能变化.

函数 f 的新定义.

```
In[7]:= f[x_] = x^2
Out[7]= x^2
```

有了新定义后，得到的结果可能发生变化.

```
In[8]:= 1 + 5 f[3]
Out[8]= 46
```

计算的简单例子包括使用把一个表达式变换为另一个表达式的定义 f[x_] = x^2. 但计算也是执行 ***Mathematica*** 内部程序的一个过

（其中一些可能是条件或循环）的执行就是对应着这些表达式的计算. 有时计算过程会涉及循环计算某一表达式多次.

表达式 Print[zzzz] 在 Do 表达式中执行了三次.

```
In[9]:= Do[Print[zzzz], {3}]
        zzzz
        zzzz
        zzzz
```

相关指南

- 核心语言
- 函数式编程
- 过程式编程
- ***Mathematica*** 语法
- 语言概述

相关教程

- 表达式的计算

教程专集

- **Core Language**

将表达式化为标准形式

Mathematica 的内部函数以多种方式计算. 但许多数学函数有重要

例如，Plus 函数就把求和化简为无括号的标准形式. 加法的结合律意味着（a + b）+ c、a +（b + c）和 a + b + c 等价. 但由于多种原因，将它们化为标准形式 a + b + c 更方便. 内部函数 Plus 就可以完成此项任务.

用内部函数 Plus 将表达式化简为无括号的形式.

```
In[1]:= (a + b) + c
Out[1]= a + b + c
```

当 ***Mathematica*** 知道一个函数有结合律时，它就会去掉括号使其成为标准的“被压平了”的形式.

像加法一样的函数不仅有结合律，而且有交换律，即 a + c + b 和 a + b + c 相等. ***Mathematica*** 将它们化为“标准”形式，即按字母顺序排列.

Mathematica 将和式的各项整理为标准形式.

```
In[2]:= c + a + b
Out[2]= a + b + c
```

flat （结合律）	$f[f[a, b], c]$ 与 $f[a, b, c]$ 等价.
orderless （交换律）	$f[b, a]$ 与 $f[a, b]$ 等价.

Mathematica 所用的化函数为标准形式的两个重要特性.

将一个表达式化为标准形式的理由很多. 最重要的是在标准形式下，很容易看出两个表达式是否相等.

消，结果为 0.

```
In[3]:= f[a + c + b] - f[c + a + b]
Out[3]= 0
```

可以想象通过测试 a + c + b 和 c + a + b 所有的顺序来判断它们是否相等. 显然将其化为标准形式后很容易判断.

可以设想让 ***Mathematica*** 自动地将*所有* 表达式化简为单一的标准形式. 但通过一些简单表达式可以看出，由于各种目的，不需要将它们化为*相同* 的单一形式.

以多项式为例，有两种不同的标准形式，第一种形式是用 Expand 函数产生的和式，它适宜于进行多项式的加减运算.

另一种形式是用 Factor 函数产生的不可约因子的乘积. 这种形式适宜于作除法.

展开形式和因子形式都是多项式非常好的标准式. 可根据需要选择它们. 结果，***Mathematica*** 不自动将多项式化为某一种标准形式. 而是提供了 Expand 和 Factor 函数让用户选用.

数学上相等的两个多项式.

```
In[4]:= t = {x^2 - 1, (x + 1) (x - 1)}
Out[4]= {-1 + x^2, (-1 + x) (1 + x)}
```

用 Expand 将它们化为展开形式就容易看出其相等.

```
In[5]:= Expand[t]
Out[5]= {-1 + x^2, -1 + x^2}
```

也可以通过因子形式看出它们相等.

```
In[6]:= Factor[t]
Out[6]= {(-1 + x) (1 + x), (-1 + x) (1 + x)}
```

尽管不希望总是将表达式化简为*相同* 的标准形式，但会猜想将表达式化为*某些* 形式.

在计算数学理论中有一个基本结论，它表明无法在任何情况下将表达式化为一些标准形式. 无法保证有限变换序列将两个任意表达式化为一个标准形式.

这结论并不奇怪，如果能将所有的数学表达式化为一个标准形式时，就很容易判断两个表达式是否相等. 许多困难的数学问题都可以表述为表达式是否相等这一事实表明这是十分困难的.

相关教程

- 表达式的计算

教程专集

- **Core Language**

属性

f[x_] = x^2 等定义给出了函数值. 有时仅需要规定函数的特性，而不需要明确指定函数的值.

Mathematica 提供了指定函数特性的一系列属性函数. 例

如，Flat 函数规定一个函数有“无层次”特性，所有的嵌套结构都被压平从而具有结合性.

给函数 f 赋以 Flat 属性.

```
In[1]:= SetAttributes[f, Flat]
```

现在函数 f 具有可结合属性，嵌套结构被压平.

```
In[2]:= f[f[a, b], c]
Out[2]= f[a, b, c]
```

Flat 等函数不仅影响计算，而且影响模式匹配运算.当对一个函数进行定义或变换时，必须首先确认这个函数的属性.

函数 f 的定义.

```
In[3]:= f[x_, x_] := f[x]
```

由于 f 具有结合律，因而对自变量序列自动应用这一定义.

```
In[4]:= f[a, a, a, b, b, b, c, c]
Out[4]= f[a, b, c]
```

`Attributes[f]`	给出 f 的属性
`Attributes[f] = {attr1, attr2, ...}`	设置 f 的属性
`Attributes[f] = {}`	令 f 无任何属性
`SetAttributes[f, attr]`	给 f 增添属性 *attr*
`ClearAttributes[f, attr]`	从 f 属性中清除 *attr*

属性的处理.

显示 f 的属性.

```
In[5]:= Attributes[f]
Out[5]= {Flat}
```

清除 f 的属性.

```
In[6]:= Attributes[f] = {}
Out[6]= {}
```

`Orderless`	可交换性（自变量按标准顺序排列）
`Flat`	可结合性（自变量处于“平等地位”）
`OneIdentity`	$f[f[a]]$ 在模式匹配中等价于 a
`Listable`	f 线状插入变量列表中（如 $f[\{a, b\}]$ 成为 $\{f[a], f[b]\}$）
`Constant`	f 的所有导数是零
`NumericFunction`	当自变量是数值时，f 就有一个数量值
`Protected`	f 的值不能改变
`Locked`	f 的属性不能改变
`ReadProtected`	不允许读 f 的值
`HoldFirst`	f 的第一个自变量不计算
`HoldRest`	f 的第二个及以后的自变量不计算
`HoldAll`	f 的所有自变量不计算
`HoldAllComplete`	f 的所有自变量不动
`NHoldFirst`	f 的第一个自变量不受 `N` 的影响
`NHoldRest`	f 除第一个外的自变量不受 `N` 的影响
`NHoldAll`	f 的所有自变量不受 `N` 的影响
`SequenceHold`	f 的自变量中的对象 `Sequence` 不能被压平
`Temporary`	f 是一个局部变量，不再使用时就被删除
`Stub`	当 f 没有明确输入时自动调用 `Needs`

Mathematica 中符号的属性表.

内部函数 Plus 的属性.

```
In[7]:= Attributes[Plus]

Out[7]= {Flat, Listable, NumericFunction,
         OneIdentity, Orderless, Protected}
```

Mathematica 函数属性中的一个重要属性是 Listable. 该属性规定函数必须线状分布到其自变量中去，即该函数分别作用于每一个自变量列中的每一个元素上.

内部函数 Log 具有属性 Listable.

```
In[8]:= Log[{5, 8, 11}]

Out[8]= {Log[5], Log[8], Log[11]}
```

定义函数 p 为 Listable.

```
In[9]:= SetAttributes[p, Listable]
```

现在 p 自动线状作用于作为它的自变量的列表上.

```
In[10]:= p[{a, b, c}, d]

Out[10]= {p[a, d], p[b, d], p[c, d]}
```

Mathematica 中可赋于函数的许多属性直接影响函数的计算，也有些属性仅影响函数处理的其它方面. 例如属性 OneIdentity 仅影响模式的匹配，如 "有交换性和结合性的函数" 中讨论的. 同样地，属性 Constant 仅与求导有关的运算相关.

Protected 属性影响函数的赋值. *Mathematica* 不允许对具有此属性的符号进行定义. "修改内部函数" 节中讨论的 Protect 和 Unprotect 函数可以用来作为设置或清除这一属性的除

SetAttributes 和ClearAttributes 之外的选择. 从 "修改内部函数" 节看出，大部分 ***Mathematica*** 的内部对象都被保护以避免错误的定义.

函数 g 的定义.

```
In[11]:= g[x_] = x + 1
Out[11]= 1 + x
```

给 g 设置属性 Protected.

```
In[12]:= Protect[g]
Out[12]= {g}
```

现在就不能再修改 g 的定义.

```
In[13]:= g[x_] = x
                Set::write : Tag g in g[x_] is Protected. ≫
Out[13]= x
```

通过 ? *f* 或一些内部函数就可以查看对一些符号的定义，但当设置了 ReadProtected 属性后，***Mathematica*** 就不允许再查看这个符号的定义. 不过在计算时仍可以继续使用此定义.

尽管无法修改 g，但还可以查看它的定义.

```
In[14]:= ?g
```

```
Global`g

Attributes[g] = {Protected}
```

给 g 加 ReadProtected 属性.

```
In[15]:= SetAttributes[g, ReadProtected]
```

现在就不能查看的 g 定义.

```
In[16]:= ?g
```

```
Global`g

Attributes[g] = {Protected, ReadProtected}
```

SetAttributes 和 ClearAttributes 等用来修改符号的属性. 但当对一个符号设置了 Locked 属性后， ***Mathematica*** 就不能再修改它的属性了. 用 Locked 、Protected 或 ReadProtected 属性可以防止用户查看或修改定义.

Clear[*f*]	清除 *f* 的值，保留其属性
ClearAll[*f*]	清除 *f* 的值和属性

清除已有值和属性.

清除 p 的值和属性 （ p 在前面设置了属性 Listable ）.

```
In[17]:= ClearAll[p]
```

现在 p 不再是 Listable.

```
In[18]:= p[{a, b, c}, d]
Out[18]= p[{a, b, c}, d]
```

给函数设置属性就是令函数具有一些性质，该函数在任何时候都具有

不要使用属性，而调用一个特定函数来实现与属性有关的变换.

调用函数 Thread 可以实现 Listable 属性的功能.

```
In[19]:= Thread[p[{a, b, c}, d]]
Out[19]= {p[a, d], p[b, d], p[c, d]}
```

Orderless	Sort[*f*[*args*]]
Flat	Flatten[*f*[*args*]]
Listable	Thread[*f*[*args*]]
Constant	Dt[*expr*, Constants -> *f*]

进行与属性有关变换的函数.

Mathematica 中只能给单个符号永久定义属性，但 ***Mathematica*** 也可以建立具有类似于属性特性的纯函数.

Function[*vars*, *body*, {$attr_1$, ...}]	具有属性 $attr_1$, ... 的纯函数

具有属性的纯函数.

这个纯函数将 p 作用于整个集合.

```
In[20]:= Function[{x}, p[x]][{a, b, c}]
Out[20]= p[{a, b, c}]
```

增添属性 Listable，使 p 作用于每个元素上.

```
In[21]:= Function[{x}, p[x], {Listable}][{a, b, c}]
Out[21]= {p[a], p[b], p[c]}
```

相关指南

- 属性

相关教程

- 表达式的计算

教程专集

- **Core Language**

标准运算过程

本节描述 ***Mathematica*** 计算表达式的标准过程，该过程适用于大部分表达式. 但用来表示 ***Mathematica*** 程序和控制结构等一些类型的表达式不按标准方式处理.

在标准计算过程中，***Mathematica*** 先计算表达式的头部，然后再计算表达式的元素. 这些元素本身也是表达式，其计算过程也是这样递推地进行.

3个 Print 函数逐个处理，打印出它们的自变量，返回值为 Null.

```
In[1]:= {Print[1], Print[2], Print[3]}
        1
        2
        3
Out[1]= {Null, Null, Null}
```

把符号 ps 设置为 Plus.

```
In[2]:= ps = Plus
Out[2]= Plus
```

光处理头部 ps，所以这个表达式就是求几项之和.

```
In[3]:= ps[ps[a, b], c]
Out[3]= a + b + c
```

当 ***Mathematica*** 处理完一个表达式的头部以后，它就判断该头部是否是具有一定属性的符号. 如果该符号具有 Orderless、Flat 或 Listable 属性，则在处理完表达式的元素后，***Mathematica*** 立即进行与这些属性有关的变换.

标准计算过程的下一步就是用 ***Mathematica*** 已知的定义对表达式进行计算. ***Mathematica*** 先使用用户给出的定义，如果没有合适的用户定义可用，它就试用内部的定义.

当 ***Mathematica*** 找到一个可用的定义后，它就对表达式进行相应的变换. 其结果仍然是一个表达式，这个表达式再按标准计算过程进行处理

- 计算表达式的头部.
- 依次计算表达式的每个元素.
- 使用与属性 `Orderless`、`Listable` 和 `Flat` 相关的变换规则.
- 使用已经给出的定义.
- 使用内部定义.
- 计算出结果.

标准计算过程.

如 "计算原理" 节所述，***Mathematica*** 遵循的原则是将一个表达式一直计算到没有定义可用为止，这意味着 ***Mathematica*** 反复进行计算直到结果不再变化为止.

下面是一个显示标准计算过程的例子，其中我们取 `a = 7`.

`2 a x+a^2+1`	原来的表达式
`Plus[Times[2,a,x], Power[a,2],1]`	该表达式的内部形式
`Times[2,a,x]`	先计算这一项
`Times[2,7,x]`	a 被取为**7**
`Times[14,x]`	`Times` 的内部定义给出这一结果
`Power[a,2]`	下一步的计算
`Power[7,2]`	计算了 a 以后的结果
`49`	`Power` 的内部定义给出这一结果
`Plus[Times[14,x], 49,1]`	这是 `Plus` 的自变量被计算出以后的结果
`Plus[50, Times[14,x]]`	调用 `Plus` 的内部定义进行计算
`50+14 x`	输出计算结果

Mathematica 计算的一个简单例子.

Mathematica 提供了“跟踪”计算过程的一些方法，正如 "计算的跟踪" 中讨论的. 函数 `Trace[`*expr*`]` 给出了显示计算过程中产生的子表达式的嵌套列表.（注意，标准计算过程采用深度优先方式遍历表达式树，因此表达式最小的子部分首先出现在 `Trace` 的结果中.）

首先令 a 等于 7.

```
In[4]:= a = 7

Out[4]= 7
```

这里给出了计算过程中所产生的子表达式的嵌套列表.

```
In[5]:= Trace[2 a x + a^2 + 1]

Out[5]= {{{a, 7}, 2 × 7 x, 14 x},
         {{a, 7}, 7^2, 49}, 14 x + 49 + 1, 50 + 14 x}
```

Mathematica 中适合不同定义的顺序是十分重要的. ***Mathematica*** 在内部定义之前使用你所给出的定义，这意味着用户给出的定义超越内部定义，在 "修改内部函数" 节中有详细的讨论.

这表达式用内部定义 ArcSin 计算.

```
In[6]:= ArcSin[1]

Out[6]= π/2
```

可以自己给出 ArcSin 的定义，但必须先清除保护属性.

```
In[7]:= Unprotect[ArcSin]; ArcSin[1] = 5 Pi / 2;
```

在内部定义之前使用了刚给出的定义.

```
In[8]:= ArcSin[1]

Out[8]= 5 π/2
```

由 "与不同符号相关的定义" 节的讨论知道，可以把符号与定义通过上值或下值联系起来. ***Mathematica*** 在下值之前试用上值.

对表达式 $f[g[x]]$ 有两种定义方式：f 的下值或 g 的上值. ***Mathematica*** 在与 f 有关的定义之前试用与 g 有关的定义.

这一顺序遵循特殊定义在一般定义之前使用的原则. 通过在与函数有

能超越一般函数的特殊变量.

这里定义了 f[g[x_]] 与 f 有关的一个规则.

```
In[9]:= f /: f[g[x_]] := frule[x]
```

这里定义了 f[g[x_]] 与 g 有关的一个规则.

```
In[10]:= g /: f[g[x_]] := grule[x]
```

与 g 有关的规则先于与 f 有关的规则而使用.

```
In[11]:= f[g[2]]
Out[11]= grule[2]
```

删除了与 g 有关的规则后，就使用与 f 有关的规则.

```
In[12]:= Clear[g]; f[g[1]]
Out[12]= frule[1]
```

> ■ 在表达式 $f[g[x]]$ 中，与 g 有关的规则在与 f 有关的规则之前使用.

定义使用的顺序.

Plus 等大部分 ***Mathematica*** 的内部函数有下值. 然而，一些 ***Mathematica*** 的内部对象也有上值. 例如，表示幂级数的 SeriesData 对象对各种数学运算有内部定义的上值.

$f[g[x]]$ 等表达式在标准运算过程中试用的所有定义的顺序为:

- 用户给出的与 g 有关的定义;
- 与 g 有关的内部定义;

- 用户给出的与 f 有关的定义；
- 与 f 有关的内部定义

上值在下值前使用在许多情况下是很重要的. 在定义一个复合运算时，可以定义复合对象的上值，一旦这些对象出现，上值就被使用. 也可以对这个复合给出一个过程，这个过程在没有出现特定对象的情况下使用. 对这一过程可以定义下值，因为下值在上值之后使用，所以仅当所有对象都没有上值时才使用此过程.

与 q 有关的“q 对象”的复合.

```
In[13]:= q /: comp[q[x_], q[y_]] := qcomp[x, y]
```

与 comp 有关的复合的一般规则.

```
In[14]:= comp[f_[x_], f_[y_]] := gencomp[f, x, y]
```

给出两个对象 q 时，与 q 有关的规则就被使用.

```
In[15]:= comp[q[1], q[2]]
Out[15]= qcomp[1, 2]
```

给出两个 r 对象时，与 comp 有关的一般规则就被使用.

```
In[16]:= comp[r[1], r[2]]
Out[16]= gencomp[r, 1, 2]
```

在一个表达式中一般会有几个对象具有上值. ***Mathematica*** 先检查表达式的头部，并试用与它有关的上值，随后检查表达式的每个元素，试用其上值. ***Mathematica*** 先对用户定义的上值，再对内部定义的上值实施这一过程. 该过程意味着，在一系列元素中，先出现元素的上值优于晚出现元素的上值.

定义 p 关于 c 的上值.

```
In[17]:= p /: c[l___, p[x_], r___] := cp[x, {l, r}]
```

定义 q 的上值.

```
In[18]:= q /: c[l___, q[x_], r___] := cq[x, {l, r}]
```

使用哪一个上值依赖于 c 中哪一个变量先出现.

```
In[19]:= {c[p[1], q[2]], c[q[1], p[2]]}
Out[19]= {cp[1, {q[2]}], cq[1, {p[2]}]}
```

相关教程

- 表达式的计算

教程专集

- **Core Language**

非标准计算

大部分 ***Mathematica*** 内部的函数按标准过程进行计算，但也有些重要的函数不遵循这一原则. 例如， ***Mathematica*** 中与程序的构造和执行有关的函数就使用非标准计算过程. 典型的情况是这些函数或者不计算它们的一些变量的值，或者按一种特殊的方法计算.

$x=y$	不计算左端
If[p,a,b]	p 等于 True 时计算 a，等于 False 时计算 b
Do[$expr$,{n}]	计算 $expr$ n 次
Plot[f,{x,...}]	对 x 的一系列数量值计算 f
Function[{x},$body$]	直到运用函数时才进行计算

使用非标准计算过程的一些函数.

当给出 a = 1 的定义时，***Mathematica*** 并不计算左端 a 的值. 如果计算 a 将会导致麻烦，其原因是在前面已定义了a = 7，又要在 a = 1 中计算 a，这就导致了 7 = 1 的矛盾.

在标准计算过程中，函数的每个自变量是逐个计算的，可以通过属性 HoldFirst、HoldRest 和 HoldAll 来改变这种情况. 这些属性使某些自变量处于不计算的状态.

HoldFirst	不计算第一个自变量
HoldRest	仅计算第一个自变量
HoldAll	不计算任何自变量

保持函数自变量不被计算的属性.

标准计算过程中的所有自变量都计算.

```
In[1]:= f[1 + 1, 2 + 4]
Out[1]= f[2, 6]
```

给 h 设置属性 HoldFirst.

```
In[2]:= SetAttributes[h, HoldFirst]
```

不计算 h 的第一个自变量.

```
In[3]:= h[1 + 1, 2 + 4]
Out[3]= h[1 + 1, 6]
```

用到 h 的第一个变量时进行计算.

```
In[4]:= h[1 + 1, 2 + 4] /. h[x_, y_] -> x^y
Out[4]= 64
```

内部函数如 Set 有 HoldFirst 等属性.

```
In[5]:= Attributes[Set]
Out[5]= {HoldFirst, Protected, SequenceHold}
```

即使一个函数具有使其某些变量保持不计算状态的属性，还可以用 Evaluate[*arg*] 的形式来告诉 ***Mathematica*** 去计算它们.

Evaluate 有效地屏蔽了 HoldFirst 属性，使第一个自变量被计算.

```
In[6]:= h[Evaluate[1 + 1], 2 + 4]
Out[6]= h[2, 6]
```

$f[\texttt{Evaluate}[arg]]$	即使 f 的属性指定 arg 不计算，也立即计算 arg

强行计算函数的变量.

保持变量可以使函数控制其变量在什么时候被计算. 通过使用 Evaluate 可立即计算变量的值，而不再由函数控制这些变量的计算. 这一功能在许多情况下是有用的.

Mathematica Set 函数保持第一个变量，故此时 a 不计算.

```
In[7]:= a = b
Out[7]= b
```

用 Evaluate 使 Set 计算它的第一个变量. 在这种情况下，结果是 a 的值的对象，而 b 设为 6.

```
In[8]:= Evaluate[a] = 6
Out[8]= 6
```

此时，b 等于 6.

```
In[9]:= b
Out[9]= 6
```

大部分情况下，需要计算表达式，有时也需要某些表达式不计算. 例如，对 ***Mathematica*** 的一个程序块进行符号处理时，就要防止在处理过程中它们被计算.

用函数 Hold 和 HoldForm 保持表达式不计算. 这些函数就是设置 HoldAll 属性或提供一种包装，使表达式处于不计算状态.

标准输出格式，Hold 可以直接显示，而 HoldForm 不直接显示，通过查看 ***Mathematica*** 的完全内部形式，就可以看到这两个函数.

Hold 保持表达式不计算.

```
In[10]:= Hold[1 + 1]

Out[10]= Hold[1 + 1]
```

HoldForm 也保持表达式不计算，但在 ***Mathematica*** 标准输出中显示它.

```
In[11]:= HoldForm[1 + 1]

Out[11]= 1 + 1
```

HoldForm 在内部形式中仍出现.

```
In[12]:= FullForm[%]

t[12]//FullForm=
        HoldForm[Plus[1, 1]]
```

ReleaseHold 函数删除 Hold 和 HoldForm，因此对该表达式进行了计算.

```
In[13]:= ReleaseHold[%]

Out[13]= 2
```

Hold[*expr*]	保持 *expr* 不计算
HoldComplete[*expr*]	保持 *expr* 不计算，并防止与 *expr* 有关的上值被使用
HoldForm[*expr*]	保持 *expr* 不计算，不显示 HoldForm
ReleaseHold[*expr*]	在 *expr* 中取消 Hold 和 HoldForm
Extract[*expr*, *index*, Hold]	取出 *expr* 的一部分，并加上 Hold 防止它被计算

处理不计算表达式的函数.

通常选取表达式的项时就计算它.

```
In[14]:= Extract[Hold[1 + 1, 2 + 3], 2]
Out[14]= 5
```

取出表达式的项，并用 Hold 包装，使它不被计算.

```
In[15]:= Extract[Hold[1 + 1, 2 + 3], 2, Hold]
Out[15]= Hold[2 + 3]
```

f[..., Unevaluated[*expr*], ...]	给出 *f* 一个不计算的 *expr* 作为变量

暂时防止变量求值.

1 + 1 得 2，Length[2] 的结果是 0.

```
In[16]:= Length[1 + 1]
Out[16]= 0
```

1 + 1 不计算作为 Length 的变量.

```
In[17]:= Length[Unevaluated[1 + 1]]
Out[17]= 2
```

Unevaluated[*expr*] 暂时给函数设置类似于 HoldFirst 的属性，并将 *expr* 作为函数的变量.

SequenceHold	不要压平作为变量出现的 Sequence 对象
HoldAllComplete	将所有变量看作不变

防止计算其它属性.

HoldAll 属性可以让 ***Mathematica*** 保持函数的变量不被计算. 但 ***Mathematica*** 还可以对变量进行变换，使用 SequenceHold 属性，可以防止变量中出现的 Sequence 对象被压平. 通过设置 HoldAllComplete 属性，可以防止Unevaluated 出现的问题，并且防止 ***Mathematica*** 保持的变量被计算和与变量有关的上值被使用.

相关教程

- 表达式的计算
- 在 **Dynamic** 或 **Manipulate** 内部计算表达式

教程专集

- **Core Language**

模式、规则和定义中的计算

在 ***Mathematica*** 中模式匹配和计算有重要的关系，首先看到模式匹配是在至少有部分计算出的表达式上进行的，所以与这些表达式匹配的模式也应该进行计算.

计算模式表明它与给出的表达式匹配.

```
In[1]:= f[k^2] /. f[x_^(1 + 1)] -> p[x]
Out[1]= p[k]
```

条件 /; 的右端仅在匹配过程中使用时才计算.

```
In[2]:= f[{a, b}] /.
          f[list_ /; Length[list] > 1] -> list^2
Out[2]= {a^2, b^2}
```

有时需要保持模式或模式的一部分不计算，这可以用 HoldPattern 包装不需要计算的这一部分. 在模式匹配中，HoldPattern[*patt*] 等价于 *patt*，但表达式 *patt* 保持被计算的形式.

HoldPattern[*patt*]	在模式匹配中等价于 *patt*，*patt* 不计算

防止模式被计算.

HoldPattern 的一个应用在于对不需要计算的表达式或保持未被计算的形式的表达式，规定其模式.

HoldPattern 使 1 + 1 不计算，并将它与 /. 操作符左端的 1 + 1 匹配.

```
In[3]:= Hold[u[1 + 1]] /. HoldPattern[1 + 1] -> x
Out[3]= Hold[u[x]]
```

注意，Hold 等函数禁止对表达式进行计算，但它们不影响对有 /. 和其它一些运算的表达式的项的操作.

当其变量不是原子形式时定义 r 的值.

```
In[4]:= r[x_] := x^2 /; ! AtomQ[x]
```

由这一定义，表达式 r[3] 等不变.

```
In[5]:= r[3]
Out[5]= r[3]
```

而根据 r 的定义模式 r[x_] 变化.

```
In[6]:= r[x_]
Out[6]= x_^2
```

需要用 HoldPattern 包装 r[x_] 去禁止它被计算.

```
In[7]:= {r[3], r[5]} /. HoldPattern[r[x_]] -> x
Out[7]= {3, 5}
```

lhs -> *rhs* 等变换规则的左端通常是立即计算的，其原因是这些规则作用的表达式已经被计算出来了. *lhs* -> *rhs* 的右端也是立即计算出来的，但使用延迟规则 *lhs* :> *rhs* 可以使表达式 *rhs* 不计算.

在 -> 中，右端立即计算，而在 :> 中，右端不立即计算.

```
In[8]:= {{x -> 1 + 1}, {x :> 1 + 1}}
Out[8]= {{x → 2}, {x :→ 1 + 1}}
```

使用规则的结果是：:> 的右端放在 Hold 中，没有被计算.

```
In[9]:= {x^2, Hold[x]} /. %
Out[9]= {{4, Hold[2]}, {4, Hold[1 + 1]}}
```

lhs->*rhs*	计算出 *lhs* 和 *rhs*
lhs:>*rhs*	计算 *lhs*，但不计算 *rhs*

变换规则的计算.

变换规则的左端常被计算，而定义的左端通常不计算，这个差异的原因在于变换规则一般是将 /. 用到已经计算出来的表达式上，而定义用在计算过程中，且作用于没有完全计算出来的表达式上. 要作用于这些表达式上，定义的左端至少是部分地未被计算.

符号的定义是一个最简单的情况，如 "非标准计算" 中讨论的，*x* = *value* 等定义左端的符号未被计算. 如果 *x* 在前面已经给赋值 *y*，且 *x* = *value* 要计算出的话，就会出现 *y* = *value* 的情形.

此定义中左端的符号没有计算.

```
In[10]:= k = w[3]
Out[10]= w[3]
```

重新定义这一符号.

```
In[11]:= k = w[4]
Out[11]= w[4]
```

如果计算了左端，定义的不是 k， 而是 k 的 w[4] *值*.

```
In[12]:= Evaluate[k] = w[5]
Out[12]= w[5]
```

现在看到 w[4] 的值为 w[5].

```
In[13]:= w[4]
Out[13]= w[5]
```

尽管在定义左端出现的单个符号不计算，但复杂的表达式则部分地被计算，在定义左端的表达式 *f*[*args*] 中， *args* 已被计算.

已计算出 1 + 1，故定义的是 g[2].

```
In[14]:= g[1 + 1] = 5
Out[14]= 5
```

这里显示 g 的值.

```
In[15]:= ?g
```

```
Global`g

g[2] = 5
```

通过考察表达式计算过程中定义的使用，就会理解为什么出现在定义左端函数的变量被计算. 如 "计算原理" 节所述， ***Mathematica*** 求一

个函数的值时，它先计算它的每个变量，然后再查找函数的定义. 于是，当 ***Mathematica*** 使用一个函数的定义时，它的变量在此以前计算出来. 这里的例外是有些函数具有其变量不被计算的属性.

symbol=*value*	计算 *value*； 不计算 *symbol*
symbol:=*value*	*symbol* 和 *value* 都不计算
f[*args*]=*value*	*args* 计算，但左端作为整体不计算
f[HoldPattern[*arg*]]=*value*	给 *f*[*arg*] 赋值，但不计算 *arg*
Evaluate[*lhs*]=*value*	左端被全部计算出来

定义中的计算.

大部分情况下需要求定义左端函数的变量，但有时也不需要计算出它们，此时可以用 HoldPattern 封装不需要计算的那些项即可.

相关教程

- 表达式的计算

教程专集

- **Core Language**

迭代函数的计算

Table 和 Sum 等 ***Mathematica*** 的内部迭代函数计算其变量值时稍有不同

所述，第一步把 i 作为局部变量，接下来，计算迭代的最大值 i_{max}. 表达式 f 保持在不被计算的状态，但随着 i 下一个值的引入反复进行计算. 当这个过程完成后，i 的全局值就恢复原值.

函数 RandomReal[] 计算了4次故有4个伪随机数.

```
In[1]:= Table[RandomReal[], {4}]
Out[1]= {0.300949, 0.450179, 0.831238, 0.161379}
```

在放入 Table 之前计算 RandomReal[]. 所以这4个数相同.

```
In[2]:= Table[Evaluate[RandomReal[]], {4}]
Out[2]= {0.653098, 0.653098, 0.653098, 0.653098}
```

大部分情况下，将 Table[f, {i, i_{max}}] 等表达式中的 f 在一个特定的值赋给 i 之前保持为不计算的状态是方便的，特别是没有一个对所有的 i 都适用的 f 时.

这里定义 fac 在变量为整数时给出阶乘，其余给出 NaN （表示“**Not a Number**”）.

```
In[3]:= fac[n_Integer] := n!; fac[x_] := NaN
```

这一形式中，fac[i] 直到一个特定的整数赋给 i 前不计算.

```
In[4]:= Table[fac[i], {i, 5}]
Out[4]= {1, 2, 6, 24, 120}
```

用 Evaluate 强行计算 fac[i]，i 是一个符号.

```
In[5]:= Table[Evaluate[fac[i]], {i, 5}]
Out[5]= {NaN, NaN, NaN, NaN, NaN}
```

在Table[f, {i, i_{max}}]等表达式中，如果对每个 i 能找出 f 完整的符号形式，则先计算这个形式，再把它存入 Table 更好. 这可以通过 Table[Evaluate[f], {i, i_{max}}] 来实现.

对每个 i 值计算 Sum.

```
In[6]:= Table[Sum[i^k, {k, 4}], {i, 8}]
Out[6]= {4, 30, 120, 340, 780, 1554, 2800, 4680}
```

对任意 i 值，可得到Sum的计算公式.

```
In[7]:= Sum[i^k, {k, 4}]
Out[7]= i + i^2 + i^3 + i^4
```

用 Evaluate 可以让 ***Mathematica*** 形式地求和，再对每个 i 迭代.

```
In[8]:= Table[Evaluate[Sum[i^k, {k, 4}]], {i, 8}]
Out[8]= {4, 30, 120, 340, 780, 1554, 2800, 4680}
```

Table[f, {i, i_{max}}]	在特定的 i 之前不计算 f
Table[Evaluate[f], {i, i_{max}}]	先对符号 i 求出 f

迭代函数的使用.

相关教程

- 表达式的计算
- **Core Language**

条件

Mathematica 中有一系列设置条件的方法，这可以使表达式在一定的条件满足时才进行计算.

lhs:=*rhs*/;*test*	当 *test* 的值为 `True` 时用定义
`If`[*test*, *then*, *else*]	当 *test* 的值为 `True` 时计算 *then*，值为 `False` 时计算 *else*
`Which`[$test_1$, $value_1$, $test_2$, …]	逐个计算 $test_i$，给出第一个 $test_i$ 为 `True` 的 **value**
`Switch`[*expr*, $form_1$, $value_1$, $form_2$, …]	将 *expr* 与每个 $form_i$ 比较，给出第一个匹配的 **value**
`Switch`[*expr*, $form_1$, $value_1$, $form_2$, …, _, *def*]	将 *def* 作为默认值
`Piecewise`[{{$value_1$, $test_1$}, …}, *def*]	给出第一个 $test_i$ 为 `True` 的 **value**

条件的结构.

test 的值为 `False`，故 *else* 表达式的返回值为 `y`.

```
In[1]:= If[7 > 8, x, y]

Out[1]= y
```

此处仅计算 “*else*” 表达式的值.

```
In[2]:= If[7 > 8, Print[x], Print[y]]
```

```
y
```

在使用 ***Mathematica*** 编程时，可以用 `If` 函数控制的多个分支来给出一个定义，也可以用 `/;` 条件控制来给出几个定义. 通常用多个定义可以使程序清楚和容易修改.

定义阶梯函数：`x > 0` 时值为 `1`，其余值为 `-1`.

```
In[3]:= f[x_] := If[x > 0, 1, -1]
```

用 `/;` 定义阶梯函数正的那一部分.

```
In[4]:= g[x_] := 1 /; x > 0
```

这里定义阶梯函数负的那一部分.

```
In[5]:= g[x_] := -1 /; x <= 0
```

这里显示了用 `/;` 给出的全部定义.

```
In[6]:= ?g
```

```
Global`g
```

```
g[x_] := 1 /; x > 0
```

```
g[x_] := -1 /; x ≤ 0
```

`If` 函数提供了两种情况下选择方法. 用多个选择时，就可以使用 `If` 函数的嵌套形式. 但通常用 `Which` 和 `Switch` 更好一些.

这里定义了一个分三段的函数. 用 True 作第三个测试以使用默认值 0.

```
In[7]:= h[x_] := Which[x < 0, x^2, x > 5, x^3, True, 0]
```

用了 Which 的第一种情况.

```
In[8]:= h[-5]
Out[8]= 25
```

这里用了第三种情况.

```
In[9]:= h[2]
Out[9]= 0
```

定义一个依赖于自变量在模 3 下的值.

```
In[10]:= r[x_] := Switch[Mod[x, 3], 0, a, 1, b, 2, c]
```

Mod[7, 3] 等于 1，故用了 Switch 的第二种情况.

```
In[11]:= r[7]
Out[11]= b
```

17 与 0 和 1 不匹配，但与 _ 匹配.

```
In[12]:= Switch[17, 0, a, 1, b, _, q]
Out[12]= q
```

Mathematica 等符号系统中的条件可能产生既不是 True，也不是 False 的结果. 例如 x == y 不给出 True 或 False 的结果，除非对 x 和 y 都指定了确定的值.

由于测试的结果既不是 True，也不是 False，故 If 没有被计算.

```
In[13]:= If[x == y, a, b]
Out[13]= If[x == y, a, b]
```

可以在 If 中添入第4个变量，当测试结果既非 True，也非 False 时使用它.

```
In[14]:= If[x == y, a, b, c]
Out[14]= c
```

If[*test*, *then*, *else*, *unknown*]	当 *test* 既不是 True 又不是 False 时 If 的一种形式
TrueQ[*expr*]	当 *expr* 为 True 时给出 True，否则给出 False
lhs===*rhs* **or** SameQ[*lhs*, *rhs*]	当 *lhs* 和 *rhs* 相等时给出 True，否则给出 False
lhs=!=*rhs* **or** UnsameQ[*lhs*, *rhs*]	当 *lhs* 和 *rhs* 不相等时给出 True，否则给出 False
MatchQ[*expr*, *form*]	当模式 *form* 与 *expr* 匹配时给出 True，否则给出 False

处理符号条件的函数.

Mathematica 仍将它作为一个符号方程.

```
In[15]:= x == y
Out[15]= x == y
```

当 *expr* 的值不是 True 时，TrueQ[*expr*] 都认为 *expr* 是 False.

```
In[16]:= TrueQ[x == y]
Out[16]= False
```

==，=== 测试两个表达式是否恒等，此处它们不恒等.

```
In[17]:= x === y
Out[17]= False
```

lhs === *rhs* 和 *lhs* == *rhs* 的区别是 === 的返回值是 True 或 False，而 == 可以以符号等式的形式出现 （见 "方程" ）. 测试两个表达式的结构时用 ===，测试数学上相等时用 ==. 在模式匹配中，***Mathematica*** 用 === 检查两个表达式是否匹配.

用 === 去测试表达式的结构.

```
In[18]:= Head[a + b + c] === Times
Out[18]= False
```

== 给出了一个无用的结果.

```
In[19]:= Head[a + b + c] == Times
Out[19]= Plus == Times
```

在设置条件时，常常用到 $test_1$ && $test_2$ && ... 等组合测试. 重要的一点是当这些测试中任一个测试的结果为 False，组合测试 $test_i$ 的结果就是 False 时. ***Mathematica*** 逐次进行每个测试 $test_i$，当某一个测试 $test_i$ 的结果为 False 时就停止.

$expr_1 \&\& expr_2 \&\& expr_3$	计算到某一个 $expr_i$ 为 `False` 为止
$expr_1 \mid\mid expr_2 \mid\mid expr_3$	计算到某一个 $expr_i$ 为 `True` 时停止

逻辑表达式的计算.

涉及两个测试的函数.

```
In[20]:= t[x_] := (x != 0 && 1 / x < 3)
```

两个测试都进行了计算.

```
In[21]:= t[2]
Out[21]= True
```

第一个测试的结果是 `False`，故第二个不再进行.第二个测试将包含 `1 / 0`，并且将产生错误.

```
In[22]:= t[0]
Out[22]= False
```

Mathematica 处理逻辑表达式的方式允许进行组合测试，在组合测试中只有前面的条件满足时，后面的测试才起作用.这种特性与C语言等类似，可以使我们很方便地编写各种 ***Mathematica*** 程序.

相关指南

- 条件

相关教程

- 表达式的计算

循环控制结构

Mathematica 程序的执行涉及到一系列表达式的计算. 在简单程序中，由;分开的表达式将逐个计算，但在一些循环中，某些表达式可能被计算多次.

Do[*expr*, {*i*, i_{max}}]	*i* 从 1 到 i_{max}，增量为 1，重复计算 *expr*
Do[*expr*, {*i*, i_{min}, i_{max}, *di*}]	*i* 从 i_{min} 到 i_{max}，增量为 *di*，重复计算 *expr*
Do[*expr*, {*i*, *list*}]	*i* 取值于 *list*，计算 *expr*
Do[*expr*, {*n*}]	计算 *expr n* 次

简单循环结构.

i 从 1 到 4 计算 Print[i^2].

```
In[1]:= Do[Print[i^2], {i, 4}]
        1
        4
        9
        16
```

k 从 2 到 6，增量为 2，对 t 进行赋值.

```
In[2]:= t = x; Do[t = 1 / (1 + k t), {k, 2, 6, 2}]; t
```

$$\text{Out[2]}= \cfrac{1}{1+\cfrac{6}{1+\cfrac{4}{1+2\,x}}}$$

Do 中定义的循环方式和函数 Table 和 Sum 中的相同. 与这些函数类似，可以使用 Do 循环嵌套.

i 从 1 到 4 循环，对每一个 i，j 从 1 到 i - 1 循环.

```
In[3]:= Do[Print[{i, j}], {i, 4}, {j, i - 1}]
{2, 1}
{3, 1}
{3, 2}
{4, 1}
{4, 2}
{4, 3}
```

有时需要进行某一个运算多次，但还不希望改变迭代变量的值. 可以用如 Table 和其它迭代函数中使用的类似方法在 Do 循环中实现这一要求.

重复赋值 t = 1 / (1 + t) 三次.

```
In[4]:= t = x; Do[t = 1 / (1 + t), {3}]; t
```

$$\text{Out[4]}= \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+x}}}$$

可以在 Do 循环中使用过程.

```
In[5]:= t = 67; Do[Print[t]; t = Floor[t / 2], {3}]
        67
        33
        16
```

Nest[*f*, *expr*, *n*]	*f* 作用于 *expr* *n* 次
FixedPoint[*f*, *expr*]	从 *expr* 开始反复使用 *f* 直到结果不再改变
NestWhile[*f*, *expr*, *test*]	从 *expr* 开始反复使用 *f* 直到 *test* 的值不为 True

重复使用函数.

Do 对迭代变量的不同值反复计算一个表达式，使用户能进行重复运算. 但用 "函数的重复调用" 节介绍的函数编程结构能编写出更有效的程序. Nest[*f*, *x*, *n*] 等可以对一个表达式反复运用函数.

这里函数 f 迭代三次.

```
In[6]:= Nest[f, x, 3]
Out[6]= f[f[f[x]]]
```

对纯函数迭代，也能得出前面使用 Do 的例子中的结果.

```
In[7]:= Nest[Function[t, 1 / (1 + t)], x, 3]
```

$$\text{Out[7]=}\quad \frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$$

Nest 可以重复运用函数若干次，用时要反复使用函数直到结果不再改变为止，这可以用 FixedPoint[*f*, *x*] 来实现.

FixedPoint 重复使用函数直到结果不再改变为止.

```
In[8]:= FixedPoint[Function[t, Print[t];
          Floor[t / 2]], 67]

        67
        33
        16
        8
        4
        2
        1
        0
Out[8]= 0
```

FixedPoint 可用来模仿 ***Mathematica*** 的计算过程或 *expr* //. *rules* 等函数运算. FixedPoint 一直重复执行到相继两个结果不再变化为止. 而 NestWhile 一直重复执行到任一个函数不再产生 True 为止.

Catch[*expr*]	计算 *expr* 直到遇见 Throw[*value*]，返回值为 *value*
Catch[*expr*, *form*]	计算 *expr* 直到 Throw[*value*, *tag*] 出现，其中 *form* 与 *tag* 匹配
Catch[*expr*, *form*, *f*]	返回值为 *f*[*value*, *tag*] 而不是 *value*

遇到 Throw 时停止计算，Catch 的返回值是 i 的当前值.

```
In[9]:= Catch[Do[Print[i];
          If[i > 3, Throw[i]], {i, 10}]]

        1

        2

        3

        4

Out[9]= 4
```

Throw 和 Catch 提供了在 ***Mathematica*** 中计算控制的又一方式. 当遇到 Throw 后计算过程停止，***Mathematica*** 就返回到最近的一个 Catch.

Scan 使用 Print 函数在每个元素上，最后返回值为 Null.

```
In[10]:= Scan[Print, {7, 6, 5, 4}]

         7

         6

         5

         4
```

遇到 Throw 后 Scan 的计算结束，所含的 Catch 返回值是 Throw 变量的值.

```
In[11]:= Catch[Scan[(Print[#];
             If[# < 6, Throw[#]]) &, {7, 6, 5, 4}]]
```

```
7
6
5
Out[11]= 5
```

用 Map 也可以得到同样的结果，当计算过程不是遇到 Throw 时结束，Map 的返回值是一个集合.

```
In[12]:= Catch[Map[(Print[#];
          If[# < 6, Throw[#]]) &, {7, 6, 5, 4}]]
7
6
5
Out[12]= 5
```

用 Throw 和 Catch 可以改变函数的编程结构，使得计算过程到一定条件出现时结束.注意，用 Throw 结束的计算的结果可能和计算全部完成得到的结果不一样.

反复使用一个函数得到的集合.

```
In[13]:= NestList[1 / (# + 1) &, -2.5, 6]
Out[13]= {-2.5, -0.666667, 3.,
          0.25, 0.8, 0.555556, 0.642857}
```

由于没有遇到 Throw，所有结果与前面相同.

```
In[14]:= Catch[NestList[1 / (# + 1) &, -2.5, 6]]
Out[14]= {-2.5, -0.666667, 3.,
          0.25, 0.8, 0.555556, 0.642857}
```

现在 NestList 的计算被改变，显示的是 Throw 变量的返回值.

```
In[15]:= Catch[NestList[
           If[# > 1, Throw[#], 1 / (# + 1)] &, -2.5, 6]]
Out[15]= 3.
```

Throw 和 Catch 完全以一种全局的方式运行，不管哪儿或怎样出现 Throw，它都使计算停止，回到所包含的 Catch.

Throw 停止 f 计算，使 Catch 的返回值为 a，f 没有任何信息留下.

```
In[16]:= Catch[f[Throw[a]]]
Out[16]= a
```

这里定义一个当其自变量的值大于10时就产生 Throw 函数.

```
In[17]:= g[x_] := If[x > 10, Throw[overflow], x!]
```

此处没有出现 Throw.

```
In[18]:= Catch[g[4]]
Out[18]= 24
```

但这里在 g 的计算过程中产生的 Throw 返回到了所包含的 Catch.

```
In[19]:= Catch[g[40]]
Out[19]= overflow
```

在小的程序中，用 Throw[*value*] 和 Catch[*expr*] 的最简单形

Throw[*value*, *tag*] 和 Catch[*expr*, *form*]. 通过保持表达式 *tag* 和 *form* 在程序的某一块中，就可以保证 Throw 和 Catch 也在这一块中运行.

这里内部的 Catch 用到 Throw.

```
In[20]:= Catch[f[Catch[Throw[x, a], a]], b]
Out[20]= f[x]
```

但这里仅外面的 Catch 用到了 Throw.

```
In[21]:= Catch[f[Catch[Throw[x, b], a]], b]
Out[21]= x
```

可以用模式去指定哪个 Catch 应该捕获标志.

```
In[22]:= Catch[Throw[x, a], a | b]
Out[22]= x
```

这里保证标志 a 是局部的.

```
In[23]:= Module[{a}, Catch[Throw[x, a], a]]
Out[23]= x
```

应该认识到当 Throw 是常数时没有必要用标志; 一般来说它可以是任何表达式.

这里内部的 Catch 用标志小于4的所有 **throw**，继续进行 Do 循环. 档标志达到4时，就用外面的 Catch.

```
In[24]:= Catch[Do[
           Catch[Throw[i^2, i], n_ /; n < 4], {i, 10}], _

Out[24]= 16
```

使用 Catch[*expr*, *form*] 和 Throw[*value*, *tag*] 时，Catch 的返回值是 Throw 中的 *value*. 使用Catch[*expr*, *form*, *f*] 时，Catch 的返回值是 *f*[*value*, *tag*].

f 作用于 Throw 中的 **value** 和 **tag** 上.

```
In[25]:= Catch[Throw[x, a], a, f]

Out[25]= f[x, a]
```

无 Throw 时，就不使用 f.

```
In[26]:= Catch[x, a, f]

Out[26]= x
```

While[*test*, *body*]	*test* 值为 True 是反复计算 *body*
For[*start*, *test*, *incr*, *body*]	计算 *start*，接下来反复计算 *body* 和 *incr*，直到 *test* 的值为 **False** 时停止

一般的循环结构.

Do、Nest 和 FixedPoint 等函数在 ***Mathematica*** 中用来进行循环，而 Throw 和 Catch 可以用来修改这种结构. 有时需要建立没有结构的循环，用 While 和 For 就能方便地做到这一点. 它

们反复进行运算，直到某条件不成立时停止.

While 循环一直进行到条件不成立时结束.

```
In[27]:= n = 17; While[(n = Floor[n / 2]) != 0, Print[n]]
         8
         4
         2
         1
```

Mathematica 的 While 和 For 函数与**C**语言中的 while 和 for 控制结构类似. 但也有些重要的区别. 例如， ***Mathematica*** For 循环和**C**语言中的逗号和分号的位置正好相反.

这是一个常用的 For 循环. i++ 使 i 增加.

```
In[28]:= For[i = 1, i < 4, i++, Print[i]]
         1
         2
         3
```

这是一个更复杂的 For 循环. 当条件 i^2 < 10 不成立时结束循环.

```
In[29]:= For[i = 1;
          t = x, i^2 < 10, i++, t = t^2 + i;
          Print[t]]
         1 + x^2
         2 + (1 + x^2)^2
         3 + (2 + (1 + x^2)^2)^2
```

在 ***Mathematica*** 中的 While 和 For 循环中，总是先测试条件是

时，While 和 For 循环结束. 所以，循环体仅在测试结果为 True 时计算.

循环条件不成立，故循环体没有进行计算.

```
In[30]:= While[False, Print[x]]
```

在 While 循环、For 循环以及 ***Mathematica*** 的过程中，***Mathematica*** 表达式按确定的顺序进行计算. 可以将这种顺序看作 ***Mathematica*** 程序执行中的控制流.

大部分情况下应该是 ***Mathematica*** 程序的控制流尽可能的简单. 控制流对于执行过程中产生的值依赖越多，理解该程序运行结构就越困难.

功能编程结构涉及非常简单的控制流. 由于 While 和 For 循环中的控制流依赖于测试条件的值，它们是较复杂的. 即使在这样的循环中，控制流一般不依赖于循环体中表达式的值.

在一些情况中，***Mathematica*** 程序中的控制流受一个过程执行中所产生结果的影响，或受循环体所产生结果的影响. 处理这种情况的一个方法是用 Throw 和 Catch. ***Mathematica*** 也提供了修改控制流的其它函数，这与**C**语言中的情况类似.

`Break[]`	退出最接近的内循环
`Continue[]`	进行本循环的下一步
`Return[`*expr*`]`	返回函数中的值 *expr*
`Goto[`*name*`]`	转入当前过程中的 `Label[`*name*`]` 元素
`Throw[`*value*`]`	将 *value* 作为最接近的 `Catch` 的返回值 （非局部返回）

流程控制函数.

当 t 大于 19 时， `Break[]` 导致循环结束.

```
In[31]:= t = 1; Do[t *= k;
          Print[t];
          If[t > 19, Break[]], {k, 10}]

         1

         2

         6

         24
```

当 k < 3 时，`Continue[]` 使循环跳过 t += 2 继续执行.

```
In[32]:= t = 1; Do[t *= k;
          Print[t];
          If[k < 3, Continue[]];
          t += 2, {k, 10}]
```

```
1
2
6
32
170
1032
7238
57 920
521 298
5 213 000
```

Return[*expr*] 使程序退出一个函数，得到一个返回值. 可以将 Throw 看作一个非局部返回，因为它退出了整个嵌套函数列，这种特性在处理一些错误条件时非常方便.

这里提供了一个使用 Return 的例子. 在以下这个特例中，如果不使用 Return，过程也一样正常运行.

```
In[33]:= f[x_] := (If[x > 5, Return[big]];
                   t = x^3;
                   Return[t - 7])
```

当变量的值大于**5**时，使用第一个 Return.

```
In[34]:= f[10]

Out[34]= big
```

当变量值为负时，这一函数给出 error 错误提示.

```
In[35]:= h[x_] := If[x < 0, Throw[error], Sqrt[x]]
```

这里没有 Throw 错误.

```
In[36]:= Catch[h[6] + 2]
```

Out[36]= $2 + \sqrt{6}$

这里产生了 Throw，Catch 的返回值是 error.

```
In[37]:= Catch[h[-6] + 2]
```

```
Out[37]= error
```

Continue[] 和 Break[] 函数等可以使 ***Mathematica*** 程序转到循环开始或结束循环. 有时，还需要将程序转到过程的某一处. 如果在过程中给出作为元素的 Label，就可以使用 Goto 语句转到该处去.

循环进行到 q 超过 6 时结束.

```
In[38]:= (q = 2;
          Label[begin];
          Print[q];
          q += 3;
          If[q < 6, Goto[begin]])
```

```
2
5
```

只有在 Label 作为元素出现的同一 ***Mathematica*** 过程中，才能使用对这个特定 ***Mathematica*** 过程使用 Goto. 一般来说，使用 Goto 语句可以简化程序结构，但程序的运行过程却复杂了一些.

相关教程

- 表达式的计算

- **Core Language**

在计算过程中收集表达式

在许多计算中，我们只关注作为输入的表达式通过计算产生的最终结果. 但有时，我们也想收集在计算过程中产生的表达式. 这可以通过使用 Sow 和 Reap 来实现.

Sow[*val*]	对最接近的 Reap 所包围的值用 *val* 进行散布
Reap[*expr*]	计算 *expr*，只返回使用 Sow 散布的值的列表

使用 Sow 和 Reap.

这里输出仅包含最终结果.

```
In[1]:= a = 3; a += a^2 + 1; a = Sqrt[a + a^2]
```

Out[1]= $\sqrt{182}$

这里给出两个中间结果.

```
In[2]:= Reap[Sow[a = 3];
          a += Sow[a^2 + 1];
          a = Sqrt[a + a^2]]

Out[2]= {√182 , {{3, 10}}}
```

这里计算和式，并收集所有偶数项.

```
In[3]:= Reap[Sum[
            If[EvenQ[#], Sow[#], #] &[i^2 + 1], {i, 10}]]

Out[3]= {395, {{2, 10, 26, 50, 82}}}
```

与 Throw 和 Catch 类似，Sow 和 Reap 可以在计算中任何位置使用.

这里定义一个可以使用 Sow 的函数.

```
In[4]:= f[x_] := (If[x < 1 / 2, Sow[x]];
           3.5 x (1 - x))
```

这里对函数进行嵌套，并获取所有小于1/2的情况.

```
In[5]:= Reap[Nest[f, 0.8, 10]]

Out[5]= {0.868312,
          {{0.415332, 0.446472, 0.408785, 0.456285}}}
```

Sow[*val*, *tag*]	使用标记散布 *val*，以表明应该在什么时间提取
Sow[*val*, {tag_1, tag_2, …}]	对每一个 tag_i 散布 *val*
Reap[*expr*, *form*]	提取所有标记与 *form* 匹配的值
Reap[*expr*, {$form_1$, $form_2$, …}]	对每个 $form_i$ 生成不同的列表
Reap[*expr*, {$form_1$, …}, *f*]	对每个不同的标记和列表，应用 *f*

使用标记（**tag**）散布（**sow**）和提取（**reap**）.

这里只获取使用了标记 x 散布的值.

```
In[6]:= Reap[Sow[1, x]; Sow[2, y]; Sow[3, x], x]
Out[6]= {3, {{1, 3}}}
```

这里 1 使用标记 x 散布了两次.

```
In[7]:= Reap[Sow[1, {x, x}];
          Sow[2, y];
          Sow[3, x], x]
Out[7]= {3, {{1, 1, 3}}}
```

使用不同标记散布的值总是出现在不同的子列表中.

```
In[8]:= Reap[Sow[1, {x, x}];
          Sow[2, y];
          Sow[3, x]]
Out[8]= {3, {{1, 1, 3}, {2}}}
```

这里对提取的标记的每个形式生成子列表.

```
In[9]:= Reap[Sow[1, {x, x}];
         Sow[2, y];
         Sow[3, x], {x, x, y}]

Out[9]= {3, {{{1, 1, 3}}, {{1, 1, 3}}, {{2}}}}
```

这里对每个不同的标记和列表应用 f.

```
In[10]:= Reap[Sow[1, {x, x}];
          Sow[2, y];
          Sow[3, x], _, f]

Out[10]= {3, {f[x, {1, 1, 3}], f[y, {2}]}}
```

标记可以成为计算的一部分.

```
In[11]:= Reap[Do[Sow[i / j, GCD[i, j]], {i, 4}, {j, i}]]

Out[11]= {Null, {{1, 2, 3, 3/2, 4, 4/3}, {1, 2}, {1}, {1}}}
```

相关教程

- 表达式的计算

教程专集

- **Core Language**

计算的跟踪

Mathematica 运行的标准方式是将任何表达式看作输入，计算它们的值并给出结果. 为了观察 ***Mathematica*** 的运行过程，不仅需要知道运行结果，还要知道中间的计算步骤.

`Trace[expr]`	产生在 *expr* 计算过程中使用的表达式列表
`Trace[expr, form]`	仅包括与模式 *form* 匹配的表达式

跟踪表达式的计算.

立即求出 `1 + 1` 的结果 2.

```
In[1]:= Trace[1 + 1]
Out[1]= {1 + 1, 2}
```

在加法前计算 `2^3`.

```
In[2]:= Trace[2^3 + 4]
Out[2]= {{2^3, 8}, 8 + 4, 12}
```

每个子表达式的计算显示在不同的子列中.

```
In[3]:= Trace[2^3 + 4^2 + 1]
Out[3]= {{2^3, 8}, {4^2, 16}, 8 + 16 + 1, 25}
```

`Trace[expr]` 给出 *expr* 计算过程中涉及的*所有* 中间表达式. 除了特别简单的情形，所产生的这些中间表达式非常多，这使`Trace` 的返回值不容易理解.

`Trace[expr, form]` 可以滤掉一部分 `Trace` 所记录的表达

阶乘函数的递推定义.

```
In[4]:= fac[n_] := n fac[n - 1]; fac[1] = 1

Out[4]= 1
```

这里给出了计算 fac[3] 过程中的*所有* 中间表达式. 结果相当复杂.

```
In[5]:= Trace[fac[3]]

Out[5]= {fac[3], 3 fac[3 - 1],
          {{3 - 1, 2}, fac[2], 2 fac[2 - 1],
            {{2 - 1, 1}, fac[1], 1}, 2 × 1, 2}, 3 × 2, 6}
```

此处仅显示了形如 fac[n_] 的中间表达式.

```
In[6]:= Trace[fac[3], fac[n_]]

Out[6]= {fac[3], {fac[2], {fac[1]}}}
```

在 Trace 中可以任意指定模式.

```
In[7]:= Trace[fac[10], fac[n_ /; n > 5]]

Out[7]= {fac[10],
          {fac[9], {fac[8], {fac[7], {fac[6]}}}}}
```

Trace[*expr*, *form*] 仅选择出 *expr* 计算过程中与模式 *form* 匹配的那些表达式.

需要跟踪 fac 等函数的调用情况时，就可以用 Trace 去挑选出形如 fac[n_] 的表达式. 也可以用模式 f[n_, 2] 去选出具有某一自变量的调用.

典型的 Mathematica 程序不仅在"函数调用"，如 fac[3]，而且在

变量赋值、控制结构等语句. 这些语句都被看作为表达式，所以，能在 Trace 中选出任何 ***Mathematica*** 程序元素. 例如，模式 k = _ 可以选出对符号 k 的所有赋值.

显示了对 k 的赋值序列.

```
In[8]:= Trace[(k = 2;
          For[i = 1, i < 4, i++, k = i / k];
          k), k = _]
```

$$\text{Out[8]= } \left\{\{k = 2\}, \left\{\left\{k = \frac{1}{2}\right\}, \{k = 4\}, \left\{k = \frac{3}{4}\right\}\right\}\right\}$$

Trace[*expr*, *form*] 能选出 *expr* 计算过程中出现的表达式. 这些表达式不必直接出现在给出的 *expr* 形式中. 它们也有可能在 *expr* 计算的部分过程中被调用的时候出现.

定义一个函数.

```
In[9]:= h[n_] := (k = n / 2; Do[k = i / k, {i, n}]; k)
```

查看 h 计算过程中产生的表达式.

```
In[10]:= Trace[h[3], k = _]
```

$$\text{Out[10]= } \left\{\left\{k = \frac{3}{2}\right\}, \left\{\left\{k = \frac{2}{3}\right\}, \{k = 3\}, \{k = 1\}\right\}\right\}$$

用 Trace 可以监视自定义函数和 ***Mathematica*** 内部函数计算的中间步骤. 而且，***Mathematica*** 内部函数的中间步骤序列与 ***Mathematica*** 具体版本的实现和优化细节有关.

`Trace[expr, f[___]]`	显示 f 的调用
`Trace[expr, i=_]`	显示 i 的赋值
`Trace[expr, _=_]`	显示所有的赋值
`Trace[expr, Message[___]]`	显示所有产生的信息

使用 Trace 的一些方式.

Trace 函数的返回值是一个列表，它反映了 ***Mathematica*** 的计算过程. 在这个列表中的表达式按照计算过程中产生的顺序排列. 大部分情况下，Trace 返回的列表具有嵌套结构，这反映了计算过程中的“结构”.

其基本思想是，在 Trace 返回值列表中的子列表代表一个 ***Mathematica*** 表达式的计算序列. 这个序列的不同元素代表同一表达式的不同形式. 通常一个表达式的计算涉及到几个其它表达式的计算. 每个计算对应于由 Trace 返回的结构中的一个子列.

这里给出一个赋值序列.

```
In[11]:= a[1] = a[2]; a[2] = a[3]; a[3] = a[4]
Out[11]= a[4]
```

这里产生一个反映 a[i] 变换序列的计算链.

```
In[12]:= Trace[a[1]]
Out[12]= {a[1], a[2], a[3], a[4]}
```

在 y + x + y 化简过程中产生的序列反映这个计算链中的元素.

```
In[13]:= Trace[y + x + y]
Out[13]= {y + x + y, x + y + y, x + 2 y}
```

函数 f 的每个变量有一个计算链，这些链由子列给出.

```
In[14]:= Trace[f[1 + 1, 2 + 3, 4 + 5]]
Out[14]= {{1 + 1, 2}, {2 + 3, 5}, {4 + 5, 9}, f[2, 5, 9]}
```

每个子表达式的计算链在不同的子列中给出.

```
In[15]:= Trace[x x + y y]
Out[15]= {{x x, x^2}, {y y, y^2}, x^2 + y^2}
```

跟踪一个嵌套表达式给出了一个嵌套列表.

```
In[16]:= Trace[f[f[f[1 + 1]]]]
Out[16]= {{{{1 + 1, 2}, f[2]}, f[f[2]]}, f[f[f[2]]]}
```

在 ***Mathematica*** 表达式的计算中有两种情况需辅助计算. 第一种情况是要计算该表达式所包含的一些子表达式，第二种情况在表达式计算中有一些规则，这些规则又要求计算其它一些表达式. 这两种辅助计算都反映在 Trace 返回值的子列中.

这是由 f 和 g 的变量的计算所产生的子计算.

```
In[17]:= Trace[f[g[1 + 1], 2 + 3]]
Out[17]= {{{1 + 1, 2}, g[2]}, {2 + 3, 5}, f[g[2], 5]}
```

附加了一个条件的函数.

```
In[18]:= fe[n_] := n + 1 /; EvenQ[n]
```

fe[6] 的计算涉及到与该条件有关的子计算.

```
In[19]:= Trace[fe[6]]

Out[19]= {fe[6], {{EvenQ[6], True}, RuleCondition[
           $ConditionHold[$ConditionHold[6 + 1]],
           True], $ConditionHold[
           $ConditionHold[6 + 1]]}, 6 + 1, 7}
```

在跟踪由递推定义的函数计算时，就会得到嵌套列表. 其原因是函数的形式在递推过程中反复出现.

例如，当定义 fac[n_] := n fac[n - 1] 时，计算 fac[6] 时产生表达式 6 fac[5]，其中 fac[5] 是一个子表达式.

这是计算 fac 时产生的嵌套子列.

```
In[20]:= Trace[fac[6], fac[_]]

Out[20]= {fac[6], {fac[5],
           {fac[4], {fac[3], {fac[2], {fac[1]}}}}}}
```

由定义 fp[n - 1] 可作为 fp[n] 的值来得到.

```
In[21]:= fp[n_] := fp[n - 1] /; n > 1
```

子表达式中不出现 fp[n]，因此没有子列表生成.

```
In[22]:= Trace[fp[6], fp[_]]

Out[22]= {fp[6], fp[6 - 1], fp[5],
          fp[5 - 1], fp[4], fp[4 - 1], fp[3],
          fp[3 - 1], fp[2], fp[2 - 1], fp[1]}
```

这里给出 **Fibonacci** 数的递推定义.

```
In[23]:= fib[n_] := fib[n - 1] + fib[n - 2]
```

这里给出递推结束的条件.

```
In[24]:= fib[0] = fib[1] = 1

Out[24]= 1
```

显示在 `fib[5]` 递推计算中的所有步骤.

```
In[25]:= Trace[fib[5], fib[_]]

Out[25]= {fib[5], {fib[4], {fib[3],
     {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}},
    {fib[2], {fib[1]}, {fib[0]}}}, {fib[3],
    {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}}}
```

Mathematica 表达式计算的每一步可以看作一个变换规则的使用. 正如 "与不同符号相关的定义" 中所述，***Mathematica*** 中的规则与特定符号或标志相联系. 可以使用 `Trace[expr, f]` 去观察与符号 *f* 相联系的变换规则作用于 *expr* 的计算步骤. 在这种情况下，`Trace` 不仅给出了使用每个规则的表达式序列，而且给出了使用这些规则的结果.

一般来说，`Trace[expr, form]` 选出计算 *expr* 的所有步骤，其中 *form* 与要计算的表达式匹配，或与所使用的规则相联系的标志匹配.

`Trace[expr, f]`	显示使用与 *f* 有关的变换规则的计算
`Trace[expr, f | g]`	显示所有与 *f* 或 *g* 有关的计算

跟踪与特定标志联系的计算.

仅显示与 fac[_] 匹配的中间表达式.

```
In[26]:= Trace[fac[3], fac[_]]

Out[26]= {fac[3], {fac[2], {fac[1]}}}
```

显示使用所有与符号 fac 有关的变换规则的计算.

```
In[27]:= Trace[fac[3], fac]

Out[27]= {fac[3], 3 fac[3 - 1],
           {fac[2], 2 fac[2 - 1], {fac[1], 1}}}
```

这是 log 函数的规则.

```
In[28]:= log[x_ y_] := log[x] + log[y]
```

跟踪 log[a b c d] 的计算过程，显示与 log 有关的所有变量.

```
In[29]:= Trace[log[a b c d], log]

Out[29]= {log[a b c d], log[a] + log[b c d],
           {log[b c d], log[b] + log[c d],
             {log[c d], log[c] + log[d]}}}
```

Trace[*expr*, *form*, TraceOn->*oform*]	转向仅跟踪与 *oform* 匹配的形式
Trace[*expr*, *form*, TraceOff->*oform*]	停止跟踪与 *oform* 匹配的形式

关闭某些形式内部的跟踪.

Trace[*expr*, *form*] 跟踪计算 *expr* 过程任何点产生的与 *form* 匹配的表达式. 有时候，也需要跟踪在计算 *expr*.

通过设置 TraceOn -> *oform*，就可以仅跟踪与 *oform* 匹配的表达式计算. 同样，通过设置 TraceOff -> *oform*，就可以不用跟踪与 *oform* 匹配的表达式计算.

显示计算的所有步骤.

```
In[30]:= Trace[log[fac[2] x]]

Out[30]= {{{fac[2], 2 fac[2 - 1], {{2 - 1, 1}, fac[1], 1},
     2 × 1, 2}, 2 x}, log[2 x], log[2] + log[x]}
```

仅显示 fac 计算过程中的步骤.

```
In[31]:= Trace[log[fac[2] x], TraceOn -> fac]

Out[31]= {{{fac[2], 2 fac[2 - 1],
     {{2 - 1, 1}, fac[1], 1}, 2 × 1, 2}}}
```

仅显示不在 fac 计算过程中的步骤.

```
In[32]:= Trace[log[fac[2] x], TraceOff -> fac]

Out[32]= {{{fac[2], 2}, 2 x}, log[2 x], log[2] + log[x]}
```

Trace[*expr*, *lhs*->*rhs*]	找出在 *expr* 计算过程中遇到的与 *lhs* 匹配的所有表达式，并用 *rhs* 替换它们

将规则运用到计算过程中遇到的表达式.

令 Trace 仅给出计算 fib[5] 过程中所用到的 fib 的变量.

```
In[33]:= Trace[fib[5], fib[n_] -> n]

Out[33]= {5, {4, {3, {2, {1}, {0}}, {1}}, {2, {1}, {0}}},
     {3, {2, {1}, {0}}, {1}}}
```

Mathematica 表达式. 它可以由其它函数调用，这是 Trace 函数的一个强大功能. 但要注意，Trace 将列表中产生的所有表达式用 HoldForm 封装起来使其不计算. HoldForm 不在 ***Mathematica*** 标准输出中显示，但它仍在表达式的内部结构中存在.

这里显示了计算过程中中间步骤的表达式.

```
In[34]:= Trace[1 + 3^2]
Out[34]= {{3^2, 9}, 1 + 9, 10}
```

这些表达式用 HoldForm 封装，使得它们不被计算.

```
In[35]:= Trace[1 + 3^2] // InputForm
t[35]//InputForm=
        {{HoldForm[3^2], HoldForm[9]}, HoldForm[1 + 9
```

在 ***Mathematica*** 的标准输出中，有些难以区别哪些列表是与 Trace 的返回结构相关，哪些是正在计算的表达式.

```
In[36]:= Trace[{1 + 1, 2 + 3}]
Out[36]= {{1 + 1, 2}, {2 + 3, 5}, {2, 5}}
```

这里的输入解决了上面的二义性问题.

```
In[37]:= InputForm[%]
t[37]//InputForm=
        {{HoldForm[1 + 1], HoldForm[2]},   {HoldForm
```

Mathematica 表达式. 它可以由其它函数调用，这是 Trace 函数的一个强大功能. 但要注意，Trace 将列表中产生的所有表达式用 HoldForm 封装起来使其不计算. HoldForm 不在 ***Mathematica*** 标准输出中显示，但它仍在表达式的内部结构中存在.

这里显示了计算过程中中间步骤的表达式.

```
In[34]:= Trace[1 + 3^2]
Out[34]= {{3^2, 9}, 1 + 9, 10}
```

这些表达式用 HoldForm 封装，使得它们不被计算.

```
In[35]:= Trace[1 + 3^2] // InputForm

{{HoldForm[3^2], HoldForm[9]}, HoldForm[1 + 9
```

在 ***Mathematica*** 的标准输出中，有些难以区别哪些列表是与 Trace 的返回结构相关，哪些是正在计算的表达式.

```
In[36]:= Trace[{1 + 1, 2 + 3}]
Out[36]= {{1 + 1, 2}, {2 + 3, 5}, {2, 5}}
```

这里的输入解决了上面的二义性问题.

```
In[37]:= InputForm[%]

1], HoldForm[2]},   {HoldForm[2 + 3], HoldForm[5]},
```

在 Trace 中使用变换规则时，在用 HoldForm 封装之前先计算结果.

```
In[38]:= Trace[fac[4], fac[n_] -> n + 1]
Out[38]= {5, {4, {3, {2}}}}
```

在复杂的计算中，由 Trace 返回的列表结构可能是相当复杂的. 当使用 Trace[*expr*, *form*] 时，Trace 给出仅与模式 *form* 匹配的表达式列表. 不管给出的是什么模式，列表的结构是相同的.

这里显示在 fib[3] 计算中出现的所有 fib[_].

```
In[39]:= Trace[fib[3], fib[_]]
Out[39]= {fib[3],
           {fib[2], {fib[1]}, {fib[0]}}, {fib[1]}}
```

这里仅显示了 fib[1]，但列表的结构与 fib[_] 的相同.

```
In[40]:= Trace[fib[3], fib[1]]
Out[40]= {{{fib[1]}}, {fib[1]}}
```

可选项 TraceDepth -> *n* 使 Trace 仅跟踪到嵌套列表中的第 *n* 层. 用这种方式，能观察到计算过程中的主要步骤，而忽略一些细节. 用 TraceDepth 或 TraceOff 可以不跟踪计算过程中的一些步骤，使 Trace 的运行加快.

这里仅显示了嵌套列表中前2层.

```
In[41]:= Trace[fib[3], fib[_], TraceDepth -> 2]
Out[41]= {fib[3], {fib[1]}}
```

`Trace[expr, form, TraceDepth->n]`	不跟踪 *expr* 计算过程中第 *n* 层以后的步骤

跟踪层次的限制.

用 `Trace[expr, form]` 得到计算 *expr* 过程中与 *form* 匹配的所有表达式列表. 有时，不仅需要观察这些表达式，也需要通过这些表达式计算得到的结果. 这可以在 `Trace` 中设置可选项 `TraceForward -> True` 来实现.

这里不仅显示了与 `fac[_]` 匹配的表达式，而且给出了它们的计算结果.

```
In[42]:= Trace[fac[4], fac[_], TraceForward -> True]

Out[42]= {fac[4],
           {fac[3], {fac[2], {fac[1], 1}, 2}, 6}, 24}
```

用 `Trace[expr, form]` 能得到计算链中间的表达式，设置 `TraceForward -> True` 可以令 `Trace` 包含计算链末尾的表达式. 设置 `TraceForward -> All` 可使 `Trace` 显示计算链中出现与 *form* 匹配的表达式之后的所有表达式.

用 `TraceForward -> All` 来包含计算链中与 `fac[_]` 匹配后的所有元素.

```
In[43]:= Trace[fac[4], fac[_], TraceForward -> All]

Out[43]= {fac[4], 4 fac[4 - 1],
           {fac[3], 3 fac[3 - 1], {fac[2], 2 fac[2 - 1],
             {fac[1], 1}, 2 × 1, 2}, 3 × 2, 6}, 4 × 6, 24}
```

通过设置 `TraceForward`，就可以在计算过程中有效地观察到某

它的生成过程. 这可以用可选项 TraceBackward 来实现. TraceBackward 显示一个计算链中某特定表达式之前的情况.

这里表示 120 来源于 fac[10] 的计算中对 fac[5] 的计算.

```
In[44]:= Trace[fac[10], 120, TraceBackward -> True]
Out[44]= {{{{{{fac[5], 120}}}}}}
```

这里显示了与产生 120 有关的计算链.

```
In[45]:= Trace[fac[10], 120, TraceBackward -> All]
Out[45]= {{{{{{fac[5], 5 fac[5 - 1], 5 × 24, 120}}}}}}
```

TraceForward 和 TraceBackward 可以使我们观察计算链中一个表达式前后的情形，有时还需要观察包含某一特定表达式的计算链，可以用 TraceAbove 来完成这一任务. 可选项 TraceAbove -> True 令 Trace 包含相关计算链的首尾表达式. 使用 TraceAbove -> All 令 Trace 包含这个计算链中的所有表达式.

这里包含了 120 计算链中的首尾表达式.

```
In[46]:= Trace[fac[7], 120, TraceAbove -> True]
Out[46]= {fac[7], {fac[6], {fac[5], 120}, 720}, 5040}
```

显示计算 fib[5] 过程中生成 fib[2] 的所有方式.

```
In[47]:= Trace[fib[5], fib[2], TraceAbove -> True]
Out[47]= {fib[5], {fib[4], {fib[3], {fib[2], 2}, 3},
           {fib[2], 2}, 5}, {fib[3], {fib[2], 2}, 3}, 8}
```

Trace[*expr*, *form*, *opts*]	使用指定选项跟踪 *expr* 的计算
TraceForward->True	包括含有 *form* 的计算链中的最后一个表达式
TraceForward->All	包括计算链中 *form* 以后的所有表达式
TraceBackward->True	包括含有 *form* 计算链的第一个表达式
TraceBackward->All	包括计算链中 *form* 以前的所有表达式
TraceAbove->True	包括所有含 *form* 的计算链中的首尾表达式
TraceAbove->All	包括所有含 *form* 的计算链中所有表达式

跟踪表中可选项的设置.

Trace[*expr*, ...] 工作的基本方式是截取 *expr* 计算过程中所遇到的每一个表达式，然后用各种判据去确定是否记录该表达式. 然后，Trace 一般在函数变量计算完成 *后* 截取表达式. 通过使用可选项 TraceOriginal -> True 能使 Trace 跟踪函数变量计算 *之前* 的表达式.

这里包括了变量计算前后与 fac[_] 匹配的表达式.

```
In[48]:= Trace[fac[3], fac[_], TraceOriginal -> True]

Out[48]= {fac[3],
           {fac[3 - 1], fac[2], {fac[2 - 1], fac[1]}}}
```

Trace 得到的列表结构中一般仅包含非平凡计算链中的表达式，例如，单个符号自身的计算不包括在内. 但设置了 TraceOriginal -> True 之后，Trace 就跟踪计算过程中的每一个表达式，包括平凡计算链中的表达式.

这里 Trace 跟踪所有表达式，包括平凡计算链内的表达式.

```
In[49]:= Trace[fac[1], TraceOriginal -> True]
Out[49]= {fac[1], {fac}, {1}, fac[1], 1}
```

可选项名	默认值	
TraceForward	False	是否显示计算链中 *form* 后的表达式
TraceBackward	False	是否显示计算链中 *form* 前的表达式
TraceAbove	False	是否显示引出含有 *form* 计算链的计算链
TraceOriginal	False	是否在表达式的头部或变量计算之前跟踪表达式

Trace 的附加可选项.

当使用 Trace 去考察程序的执行时必然涉及到局部变量的处理. 正如 "模块工作方式" 所述，***Mathematica*** 中的Module 等产生一些新符号来表示局部变量. 于是，即使在原代码中变量名为 x，在执行过程中会被重命名为 x$*nnn* 等.

跟踪 Trace[*expr*, *form*] 的设置规律，出现在 *form* 中的 *x* 将与 *expr* 执行中出现的所有形如 *x*$*nnn* 的符号相匹配. 例如，使用 Trace[*expr*, *x* = _] 可以跟踪原程序中所有变量（全局或局部）*x* 的赋值过程.

Trace[*expr*, *form*, MatchLocalNames->False]

包含了与 *form* 匹配的 *expr* 执行过程中所有步骤，但不能用局部变量名替换

禁止与局部变量匹配.

有时，仅需要跟踪全局变量 *x*，不跟踪名为 *x* 的局部变量，这可以通过设置可选项 MatchLocalNames -> False 来实现.

跟踪所有名为 x$*nnn* 变量的赋值.

```
In[50]:= Trace[Module[{x}, x = 5], x = _]
Out[50]= {{x$1 = 5}}
```

仅对全局变量 *x* 的赋值进行跟踪.

```
In[51]:= Trace[Module[{x}, x = 5],
          x = _, MatchLocalNames -> False]
Out[51]= {}
```

Trace 函数进行彻底的计算，返回值是代表这个计算过程的结构. 在很长的计算中，有必要跟踪计算进程. 函数TracePrint 与 Trace 相似，但它显示所遇到的表达式，而不像 Trace 那样保存这些表达式去产生一个结构.

显示在 fib[3] 计算中出现的表达式.

```
In[52]:= TracePrint[fib[3], fib[_]]
```

```
fib[3]
 fib[3 - 1]
 fib[2]
  fib[2 - 1]
  fib[1]
  fib[2 - 2]
  fib[0]
 fib[3 - 2]
 fib[1]
```

Out[52]= 3

TracePrint 显示的表达式序列与 Trace 返回的列表结构中的表达式序列相对应. TracePrint 输出中的缩进形式与Trace 列表结构中的嵌套相对应. 在 TracePrint 中也可以使用 Trace 的可选项 TraceOn、TraceOff 和TraceForward. 但由于 TracePrint 随着进程产生输出，所以它不能用可选项 TraceBackward. 另外，总是把 TracePrint 设置成 TraceOriginal 值为 True.

Trace[*expr*, ...]	跟踪 *expr* 的计算过程，返回值是包含所出现表达式的一个列表结构
TracePrint[*expr*, ...]	跟踪 *expr* 的计算，显示所遇到的表达式
TraceDialog[*expr*, ...]	跟踪 *expr* 的计算，当每个指定的表达式出现时产生一个对话
TraceScan[*f*, *expr*, ...]	跟踪 *expr* 的计算，对遇到表达式的 HoldForm 运用 *f*

跟踪计算函数.

在 fac[10] 的计算过程中出现 fac[5] 时显示一个对话.

```
In[53]:= TraceDialog[fac[10], fac[5]]

          TraceDialog::dgbgn :
           Entering Dialog; use Return[] to exit.

Out[53]= fac[5]
```

在对话中通过查看“**stack**”就可以发现当前在何处.

```
In[54]:= Stack[]

Out[54]= {TraceDialog, Times,
          Times, Times, Times, Times, fac}
```

从对话中返回，给出 fac[10] 的计算结果.

```
In[55]:= Return[]

          TraceDialog::dgend : Exiting Dialog.

Out[55]= 3 628 800
```

函数 TraceDialog 可以暂停计算，并对 ***Mathematica*** 当时环境进行处理. 例如，得到计算过程中中间变量的值，甚至重新设置它们的值. 当然还有许多辅助作用，大部分都与模式或模块有关.

TraceDialog 的功能是在一个表达式列中调用 Dialog 函数. Dialog 函数将在 "对话" 节仔细讨论. 在调用Dialog 时就开始了一个具有自己输入和输出列的 ***Mathematica*** 辅助进程.

跟踪计算时，一般需要对所得到的表达式运用一个任意函数. TraceScan[*f*, *expr*, ...] 将 *f* 作用于所出现的每个表达式上，用 HoldForm 封装的表达式不计算.

在 TraceScan[*f*, *expr*, ...] 中，函数 *f* 在表达式被计算之前

作用于该表达式，而 TraceScan[*f*, *expr*, *patt*, *fp*] 在计算之前作用于 *f*，而在计算之后作用于 *fp*.

相关教程

- 表达式的计算

教程专集

- **Core Language**

计算堆栈

在计算过程中，***Mathematica*** 保持一个包含当前计算的表达式的计算堆栈. 可以用 Stack 函数查看堆栈的内容. 这意味着，例如，在 ***Mathematica*** 中途中断了计算之后，就能用 Stack 函数来看 ***Mathematica*** 做什么.

Mathematica 先计算的总是堆栈中的最后一个元素，前面的元素正在计算之中.

因此，正在计算 *x* 时，与 *f*[*g*[*x*]] 相关的堆栈为 {*f*[*g*[*x*]], *g*[*x*], *x*}.

Stack[_] 给出了调用时正在被计算的表达式，这里包括了 Print 函数.

```
In[1]:= f[g[Print[Stack[_]]]];

        {f[g[Print[Stack[_]]]];,
         f[g[Print[Stack[_]]]],
```

Stack[] 给出了调用时与计算有关的标识符.

```
In[2]:= f[g[Print[Stack[]]]];

        {CompoundExpression, f, g, Print}
```

一般地，可以把计算堆栈看作显示在计算过程中 ***Mathematica*** 调用的函数所在的位置. 用 TraceAbove 设置为 True 的选项，在 Trace 返回的嵌套列中的第一个元素与这个表达式序列相对应.

Stack[]	给出一个与正在进行的计算有关的标识符列表
Stack[_]	给出当前正在计算的表达式列表
Stack[*form*]	仅包括与 *form* 匹配的表达式

观察计算堆栈.

在 ***Mathematica*** 主进程中很少直接调用 Stack. 通常是在计算过程的中间调用 Stack. 典型的情况是在对话中、在辅助进程中调用，这将在 "对话" 节中讨论.

阶乘函数的递推定义.

```
In[3]:= fac[1] = 1; fac[n_] := n fac[n - 1]
```

计算 fac[10]，当遇到 fac[4] 开始对话.

```
In[4]:= TraceDialog[fac[10], fac[4]]

          TraceDialog::dgbgn :
           Entering Dialog; use Return[] to exit.

Out[4]= fac[4]
```

这里显示了对话开始时，正在计算的对象.

```
In[5]:= Stack[]

Out[5]= {TraceDialog, Times, Times,
         Times, Times, Times, Times, fac}
```

退出对话，并给出最终结果.

```
In[6]:= Return[]
                TraceDialog::dgend : Exiting Dialog.

Out[6]= 3 628 800
```

在简单情况下，***Mathematica*** 计算堆栈用来记录当前正在计算的所有表达式，但有时这样不方便. 例如，执行`Print[Stack[]]` 总是显示堆栈，其中 `Print` 是最后一个函数.

函数 `StackInhibit` 可以避免这种问题，`StackInhibit[`*expr*`]` 计算表达式 *expr*，但不改变堆栈的内容.

`StackInhibit` 禁止 `Print` 被包含在堆栈之中.

```
In[7]:= f[g[StackInhibit[Print[Stack[]]]]];

Out[7]= {CompoundExpression, f, g}
```

`TraceDialog` 等函数在开始一个对话时，自动调用 `StackInhibit`，即 `Stack` 不显示在对话过程中调用的函数，只显示外面的函数.

StackInhibit[*expr*]	计算 *expr*，不修改**stack**
StackBegin[*expr*]	计算 *expr*，刷新堆栈
StackComplete[*expr*]	计算 *expr*，将计算链中的表达式包含在堆栈之中

计算堆栈的控制.

使用 StackInhibit 和 StackBegin 时，可以控制将计算过程中某一部分记录在堆栈之中. StackBegin[*expr*] 计算 *expr*，刷新堆栈意味着在计算 *expr* 的过程中，堆栈不包含 StackBegin 外的其它内容. TraceDialog[*expr*, ...] 等函数在计算 *expr* 之前调用 StackBegin，所以堆栈显示 *expr* 的计算过程，而不显示 TraceDialog 的调用过程.

在计算 *expr* 时，StackBegin[*expr*] 使用新堆栈.

```
In[8]:= f[StackBegin[
          g[h[StackInhibit[Print[Stack[]]]]]]]

        {g, h}

Out[8]= f[g[h[Null]]]
```

Stack 一般只显示正在计算的表达式，所以它包括每个表达式的最新形式，有时也需要查看表达式以前的形式，这可以用 StackComplete 实现.

StackComplete[*expr*] 可以记录当前正在计算的表达式的完整计算链，此时堆栈与设置了TraceBackward -> All 和 TraceAbove -> True 后 Trace 得到的表达式序列相对应.

- 符号执行的历史记录

相关教程

- 表达式的计算

教程专集

- **Core Language**

无穷计算的控制

在计算表达式时，***Mathematica*** 遵循的一般原则是将变换规则一直使用到表达式不再变化为止. 这意味着，当***Mathematica*** 进行 `x = x + 1` 的赋值时将进入无穷循环. 但实际上，***Mathematica*** 在有限步后就停止，这个步数由全局变量 `$RecursionLimit` 决定. 但总可以通过明确终止 ***Mathematica*** 提前结束它.

这个赋值可能导致无穷循环. ***Mathematica*** 在由 `$RecursionLimit` 决定的有限步后停止.

```
In[1]:= x = x + 1

          $RecursionLimit::reclim :
           Recursion depth of 256 exceeded. >>

Out[1]= 255 + Hold[1 + x]
```

当 ***Mathematica*** 没有计算完而停止时，它返回一个保持的结果，通过调用 ReleaseHold 可继续计算.

```
In[2]:= ReleaseHold[%]

        $RecursionLimit::reclim :
         Recursion depth of 256 exceeded. >>

Out[2]= 510 + Hold[1 + x]
```

$RecursionLimit	计算堆栈的最大深度
$IterationLimit	计算链的最大长度

限制无穷计算的全局变量.

这是一个循环定义，它的计算由 $IterationLimit 停止.

```
In[3]:= {a, b} = {b, a}

        $IterationLimit::itlim :
         Iteration limit of 4096 exceeded. >>

        $IterationLimit::itlim :
         Iteration limit of 4096 exceeded. >>

Out[3]= {Hold[b], Hold[a]}
```

变量 $RecursionLimit 和 $IterationLimit 以两种方式控制 ***Mathematica*** 中的无穷计算. $RecursionLimit 限制计算链的最大长度，或由 Trace 产生的列表结构中最大嵌套的深度. $IterationLimit 限制任何计算链的最大长度，或由 Trace 产生的结构列表的最大长度.

$RecursionLimit 和 $IterationLimit 的默认值适合大部分计算和计算机系统. 可以重新设置它们为一个整数值或

$RecursionLimit = Infinity，正如 "内存管理" 所述.

这里将 $RecursionLimit 和 $IterationLimit 设置为 20.

```
In[4]:= $RecursionLimit = $IterationLimit = 20

Out[4]= 20
```

这里的无穷定义20步以后结束.

```
In[5]:= t = {t}

            $RecursionLimit::reclim :
             Recursion depth of 20 exceeded. >>

Out[5]= {{{{{{{{{{{{{{{{{{{{Hold[
                                {t}]}}}}}}}}}}}}}}}}}}}}
```

无结束条件时，这个递推定义将导致无穷计算.

```
In[6]:= fn[n_] := {fn[n - 1], n}
```

计算结束之前已形成了一个相当大的结构.

```
In[7]:= fn[10]

            $RecursionLimit::reclim :
             Recursion depth of 20 exceeded. >>

Out[7]= {{{{{{{{{{{{{{{{{{{{Hold[fn[-8 - 1]], -8}, -7},
                                -6}, -5}, -4}, -3}, -2},
                           -1}, 0}, 1}, 2}, 3}, 4},
                     5}, 6}, 7}, 8}, 9}, 10}
```

另一个递推定义.

```
In[8]:= fm[n_] := fm[n - 1]
```

.

```
In[9]:= fm[0]
        $IterationLimit::itlim:
         Iteration limit of 20 exceeded. >>
Out[9]= Hold[fm[-19 - 1]]
```

无穷循环不仅花费时间而且占用内存. 限制的计算 `$IterationLimit` 一般不产生大的中间结构. 但 `$RecursionLimit` 常常产生大的中间结构. 一般情况下，结构的大小是 `$RecursionLimit` 值的线性函数. 但有时，也会随着 `$RecursionLimit` 指数级增长.

赋值如 `x = x + 1` 显然是一个循环. 但在使用复杂的循环时，很难确认循环结束，或者不进入无限循环. 检查的主要方向是变换规则的左右两端总不相等. 这就保证了计算过程不断前进，***Mathematica*** 就不停止变换规则的反复使用.

在使用复杂的 `/;` 条件时会产生奇怪的现象（见 "限制模式" ）. 特别糟糕的是涉及全局变量的改变. ***Mathematica*** 可能会认为由于表达式没有改变，所以计算已经结束. 然而，一些其它操作的副作用可能会改变全局变量的值，所以将导致计算中的新结果. 避免这个问题的最好方式是在 `/;` 条件中不使用全局变量. 这也失败时，可用 `Update[s]` 去令 ***Mathematica*** 更新涉及 *s* 的所有表达式. `Update[]` 令 ***Mathematica*** 更新所有的表达式.

相关指南

- 计算控制
- 定时计算

- 表达式的计算
- 日期和时间函数

教程专集

- **Core Language**

中断与退出

"中断计算" 节中叙述了如何按某些键来中断 ***Mathematica*** 的计算.

有时候，需要在 ***Mathematica*** 程序中模拟使用这一方法. 一般说来，执行 `Interrupt[]` 与按中断键有相同的使用. 如 "中断计算" 节所述，在某些系统中会在屏幕上出现一个选择菜单.

`Interrupt[]`	中断计算
`Abort[]`	退出计算
`CheckAbort[`*expr*`,` *failexpr*`]`	计算 *expr* 并给出结果，如果退出就返回 *failexpr*
`AbortProtect[`*expr*`]`	直到计算 *expr* 完成时一直屏蔽计算结果

中断与退出.

函数 `Abort[]` 在中断计算时与在中断菜单中选退出项 `abort` 的效果相同.

可以用 `Abort[]` 在程序中“立即中断”，但在几乎所有的情况下，最好使用函数如 `Return` 和 `Throw`，它们产生的结果容易控制.

Abort 终止了计算，仅第一个 Print 被执行.

```
In[1]:= Print[a]; Abort[]; Print[b]

        a

Out[1]= $Aborted
```

在计算 ***Mathematica*** 表达式的过程的任何点中断了计算以后，***Mathematica*** 将放弃对整个表达式的计算，返回值为 $Aborted.

但可以使用 CheckAbort 函数去捕获中断. 如果在 CheckAbort[*expr*, *failexpr*] 的 *expr* 计算过程中中断，函数 CheckAbort 的返回值为 *failexpr*，且中断不再扩大. Dialog 等函数这样使用 CheckAbort 以包含中断的影响.

CheckAbort 捕获中断，显示 c，返回值为 aborted.

```
In[2]:= CheckAbort[Print[a];
         Abort[];
         Print[b], Print[c];
         aborted]

        a

        c

Out[2]= aborted
```

Abort 的效果包含在 CheckAbort 中，所以显示 b.

```
In[3]:= CheckAbort[Print[a];
         Abort[], Print[c];
         aborted]; Print[b]
```

```
a
c
b
```

在 ***Mathematica*** 的复杂程序中，有时需程序中的某一部分不发生中断，可以通过交互或是调用 Abort 实现. 函数AbortProtect 可以使计算一直进行下去，而将计算过程中的中断保存到表达式计算结束.

Abort 一直保存到 AbortProtect 完成.

```
In[4]:= AbortProtect[Abort[];
         Print[a]]; Print[b]
a
Out[4]= $Aborted
```

CheckAbort 看见中断，但使中断不再继续扩大.

```
In[5]:= AbortProtect[Abort[];
         CheckAbort[Print[a], x]]; Print[b]
b
```

即使在 AbortProtect 之内，CheckAbort 将发现任何中断，并返回到一个合适的 *failexpr*. 当这个 *failexpr* 本身不包含 Abort[] 时，中断就会被 CheckAbort 所“吸收”.

相关指南

- 控制流

相关教程

Mathematica 表达式的编译

在定义 f[x_] := x Sin[x] 函数时，***Mathematica*** 将表达式 x Sin[x] 按能对所有的 x 求值的方式保存. 然后，当你给出了 x 的一个值时，***Mathematica*** 就将这个值代入表达式 x Sin[x] 中计算出结果. 不管 x 的指定值是一个数、一个列表、一个代数元素，或者是其它形式的表达式，用来进行计算的 ***Mathematica*** 内部代码使其工作过程一样有效.

要考虑到所有这些可能性时计算过程就会变慢. 但是，如果 ***Mathematica*** 能将 x 看作一个机器数时，这将省掉许多步骤，从而使 x Sin[x] 的计算相当快.

使用 Compile 可以在 ***Mathematica*** 中产生编译后的函数，这个函数在计算 ***Mathematica*** 表达式时假设所有出现的参数都是数（或逻辑变量）. Compile[{x_1, x_2, ...}, *expr*] 使用表达式 *expr* 并产生一个“编译过的函数”，当给出了变量 x_1, x_2, ... 的值后，所编译的函数就计算表达式的值.

一般来说，Compile 产生一个 CompiledFunction 目标，该目标包含一系列计算所编译函数的一系列简单指令. 这些指令与典型计算机中的机器代码很接近，所以执行得很快.

Compile[{$x_1, x_2, \ldots$}, $expr$]	产生编译后的函数，对 x_i 的值计算 $expr$

产生编译后的函数.

定义 f 是计算 x Sin[x] 关于 x 的纯函数.

```
In[1]:= f = Function[{x}, x Sin[x]]
Out[1]= Function[{x}, x Sin[x]]
```

产生计算 x Sin[x] 的编译函数.

```
In[2]:= fc = Compile[{x}, x Sin[x]]
Out[2]= CompiledFunction[{x}, x Sin[x], -CompiledCode-
```

f 和 fc 的结果相同，但当变量是数时，fc 运行很快.

```
In[3]:= {f[2.5], fc[2.5]}
Out[3]= {1.49618, 1.49618}
```

在计算数量逻辑表达式许多次时，Compile 是非常有效的. 花一点时间调用 Compile 后，所得到的编译函数比任何普通的 ***Mathematica*** 函数执行得更快.

对形如 x Sin[x] 这样的简单表达式，普通函数和编译函数的执行速度几乎没有什么差别. 当所涉及的表达式的数量增加时，编译函数就显示出了其优越性. 对大型的表达式编译可以提高速度**20**倍.

当表达式含有大量简单的算术运算和函数时，编译就有很大的差异. 对 BesselK 和 Eigenvalues 等复杂函数，大部分时间花在执

这里产生一个计算10阶勒让德多项式. Evaluate 令 *Mathematica* 在编译之前构造多项式.

```
In[4]:= pc = Compile[{x}, Evaluate[LegendreP[10, x]]]
```

$$\text{Out[4]= CompiledFunction}\left[\{x\}, \frac{1}{256}\left(-63 + 3465\,x^2 - 30\,030\,x^4 + 90\,090\,x^6 - 109\,395\,x^8 + 46\,189\,x^{10}\right), -\text{CompiledCode}-\right]$$

当自变量为 0.4 时，计算勒让德多项式的值.

```
In[5]:= pc[0.4]
Out[5]= 0.0968391
```

这里使用了内部代码.

```
In[6]:= LegendreP[10, 0.4]
Out[6]= 0.0968391
```

即使用编译可以加快数值函数的计算速度，但应该尽可能地使用 *Mathematica* 的内部函数. 通常内部函数比用户能产生的任何编译过的 *Mathematica* 程序执行得更快，另外它们使用许多算法，容易控制且计算精度高.

应该知道 *Mathematica* 的内部函数经常使用 Compile. 例如，NIntegrate 自动对要积分的表达式使用 Compile. 函数如 Plot 和 Plot3D 对要绘图的表达式也自动使用 Compile. 使用 Compile 的内部函数一般都有可选项Compiled. 设置 Compiled -> False 让函数不使用 Compile.

Compile[{{x_1, t_1}, {x_2, t_2}, ...}, expr]	编译 expr，假设 x_i 的类型为 t_i
Compile[{{x_1, t_1, n_1}, {x_2, t_2, n_2}, ...}, expr]	编译 expr，假设 x_i 是秩为 n_i、元素都是 t_i 类型的阵列
Compile[vars, expr, {{p_1, pt_1}, ...}]	编译 expr，假设与 p_i 匹配的子表达式的类型为 pt_i
_Integer	机器长度的整型数
_Real	近似实数的机器精度
_Complex	近似复数的机器精度
True | False	逻辑变量

指定编译类型.

Compile 工作时要假设在所给表达式计算过程中出现的对象的类型. 默认值是假设表达式的所有变量是近似实数.

Compile 使用整数、复数（True 或 False）及阵列. 可以通过与某一类型的值匹配的方法来指定某一个变量. 例如，用模式 _Integer 指定整型数，用 True | False 指定逻辑变量是 True 或者 False.

在 i 和 j 都是整型的假设下，编译表达式 5 i + j.

```
In[7]:= Compile[{{i, _Integer}, {j, _Integer}}, 5 i + j]

Out[7]= CompiledFunction[
          {i, j}, 5 i + j, -CompiledCode-]
```

这里给出整数结果.

```
In[8]:= %[8, 7]

Out[8]= 47
```

这里编译一个元素是整数的矩阵运算.

```
In[9]:= Compile[{{m, _Integer, 2}},
          Apply[Plus, Flatten[m]]]

Out[9]= CompiledFunction[{m},
          Plus @@ Flatten[m], -CompiledCode-]
```

按照编译后的方式进行一系列运算，其结果是一个整数.

```
In[10]:= %[{{1, 2, 3}, {7, 8, 9}}]

Out[10]= 30
```

Compile 处理的类型基本上与机器代码水平上计算机处理的类型相对应. 例如，Compile 能处理机器精度内的近似实数，但不能处理任意精度的实数. 另外，如果指定某变量是整数，Compile 产生的代码仅能处理机器长度之内的证书，该长度通常在 $\pm 2^{31}$ 之间.

当编译的表达式仅涉及标准算术运算和逻辑运算时，Compile 能够简单地从输入变量的类型推断每一步产生的对象的类型. 但如果，调用其它函数时，Compile 一般就无法推断其返回值的类型. 当用户不指定时，Compile 假定其它函数产生一个近似实数值. 当然可以直接给出一个模式列去确定与一个模式匹配的表达式的类型.

定义一个函数，当自变量是整数时，其值也为整数.

```
In[11]:= com[i_] := Binomial[2 i, i]
```

假设 com[_] 总是整数，编译 x^com[i].

```
In[12]:= Compile[{x, {i, _Integer}},
          x^com[i], {{com[_], _Integer}}]
```

```
Out[12]= CompiledFunction[
          {x, i}, x^com[i], -CompiledCode-]
```

计算所编译函数的值.

```
In[13]:= %[5.6, 1]
```

```
Out[13]= 31.36
```

Compile 的思路是对自变量的某一类型产生一个优化后的函数. 但 Compile 的构造使得它所产生的函数对任何类型的自变量都能使用. 当无法优化时，就通过计算一个标准的 ***Mathematica*** 表达式来得到函数的值.

求变量平方根的编译函数.

```
In[14]:= sq = Compile[{x}, Sqrt[x]]
```

```
Out[14]= CompiledFunction[{x}, √x, -CompiledCode-]
```

给出实数自变量时，就使用优化代码.

```
In[15]:= sq[4.5]
```

```
Out[15]= 2.12132
```

Mathematica 给出一个警告，然后计算原来的符号表达式.

```
In[16]:= sq[1 + u]

          CompiledFunction::cfsa :
           Argument 1 + u at position 1 should
             be a machine-size real number. >>

Out[16]= √(1 + u)
```

由 Compile 产生的编译代码不仅要对所使用的自变量类型进行假设，而且也要对执行过程中产生目标的类型进行假设. 有时候，这些类型依赖于所给自变量的实际值. 因此，例如，Sqrt[*x*] 当 *x* 为非负数时的值为实数，而当 *x* 不是负数时，*x* 为复数.

Compile 总是对一个函数的返回值类型进行确定的假设. 当这假设在 Compile 产生的代码执行过程中不成立时，***Mathematica*** 就放弃编译过的代码，并计算普通的 ***Mathematica*** 表达式去得到结果.

编译的代码不能得到复数结果，所以 ***Mathematica*** 必须直接计算原来的符号表达式.

```
In[17]:= sq[-4.5]

          CompiledFunction::cfn :
           Numerical error encountered at instruction 2;
             proceeding with uncompiled evaluation. >>

Out[17]= 0. + 2.12132 i
```

Compile 的一个重要特点是它不但可以处理数学表达式，还可以处理各种简单的 ***Mathematica*** 程序. 例如，Compile 可以处理条件和控制流结构.

在所有情况下，Compile[*vars*, *expr*] 保持其变量不计算，这意

这里产生了一个牛顿法求平方根的 ***Mathematica*** 程序的编译形式.

```
In[18]:= newt =
          Compile[{x, {n, _Integer}}, Module[{t}, t = x;
            Do[t = (t + x / t) / 2, {n}];
            t]]
```

```
Out[18]= CompiledFunction[{x, n}, Module[{t}, t = x;
           Do[t = 1/2 (t + x/t), {n}];
           t], -CompiledCode-]
```

执行编译的代码.

```
In[19]:= newt[2.4, 6]
```

```
Out[19]= 1.54919
```

相关指南

- 核心语言
- 调用外部程序
- 时间测量和优化
- 调整和调试
- **Wolfram *LibraryLink***

相关教程

- **The *Mathematica* Compiler User Guide**
- 表达式的计算
- 计算原理
- **C Code Generation User Guide**

教程专集

- **Core Language**

Wolfram|Alpha 中的数据格式

简介

交互式探索数据格式

- **Wolfram|Alpha** 查询
  - 基本工作流程
  - 公开的数据不需要有类似图形的结果
  - 第二个参变量的结构
- 自由格式输入

编程式获取数据格式

- 作为子 **pod** 属性的数据格式
- 决定可用的格式
- 直接索取数据

公开的数据例子

- 一般事项
- 量与数
- 时间序列
- 公式
- 声音

Mathematica 语言结构

基本对象

- 表达式
- 符号
- 上下文
- 原子对象
- 数字
- 字符字符串

输入语法

- 输入字符
- 输入语法的类别
- 由字符组成的字符串
- 符号名和上下文
- 数
- 括号内的对象
- 运算符的输入形式
  - 特殊字符
  - 输入形式的优先级和排序
  - 输入形式的分组
  - 积分算子的优先级
  - 空格与乘法
  - 需要避免的空格
  - 分隔符

文件名
二维输入形式
框的输入
控制键
从文本构建的框
基于字符串的输入
输入表达式的范围
特殊输入
前端文件

一些普遍的记号和传统表示

函数名
函数变量
选项
元素编号
序列规格
层规格
迭代器
作用域结构
表达式的顺序
数学函数
数学常数
保护
简略字符串模式

计算

- 标准计算序列
- 非标准变量计算
  - 逻辑运算
  - 迭代函数
  - 赋值
- 重载非标准变量计算
- 避免计算
- 计算的全局控制
- 退出

模式和变换规则

- 模式
- 赋值
- 值的类型
- 清除和删除对象
- 变换规则

文件和流

- 文件名
- 流

教程专集

- **Core Language**

- 表达式
- 符号
- 上下文
- 原子对象
- 数字
- 字符字符串

表达式

表达式（***Expressions***）是 ***Mathematica*** 中数据的主要类型.

表达式可以写成 $h[e_1, e_2, \ldots]$ 的形式. 一般说来，对象 h 被认为是表达式的*头部*. e_i 被称为表达式的*元素*. 头部和元素本身可能也是表达式.

表达式的各组成部分可以使用数字索引来指代. 头部索引为**0**；元素 e_i 索引为 i. `Part`[*expr*, i] 或 *expr*[[i]] 使用索引 i 给出 *expr* 的部. 负索引从尾部计算.

`Part`[*expr*, i_1, i_2, ...]、*expr*[[i_1, i_2, ...]] 或者 `Extract`[*expr*, {i_1, i_2, ...}] 通过连续提取索引为 i_1, i_2, ... 的子表达式的各组成部分，找到 *expr* 的相应的块. 如果用户把表达式当作树来考虑，索引就指明了当用户从树的根节点开始往下搜索的时候，在每个节点应该选择哪个分支.

通过给出由刚好 n 个索引组成的序列来指定的表达式的组成部分被定义为位于表达式中的第 n 层. 用户可以使用层来决定函数如 `Map` 的应用域. 第**0**层对应于整个表达式.

定义表达式的深度为用来指定表达式各部的最大索引数，并加上**1**. 负的层数 $-n$ 指的是表达式的各部分都具有深度 n.

- 表达式
- 符号
- 上下文
- 原子对象
- 数字
- 字符字符串

表达式

表达式（***Expressions***）是 ***Mathematica*** 中数据的主要类型.

表达式可以写成 $h[e_1, e_2, \ldots]$ 的形式. 一般说来，对象 h 被认为是表达式的*头部*. e_i 被称为表达式的*元素*. 头部和元素本身可能也是表达式.

表达式的各组成部分可以使用数字索引来指代. 头部索引为**0**；元素 e_i 索引为 i. `Part`[*expr*, i] 或 *expr*[[i]] 使用索引 i 给出 *expr* 的部. 负索引从尾部计算.

`Part`[*expr*, i_1, i_2, ...]、*expr*[[i_1, i_2, ...]] 或者 `Extract`[*expr*, {i_1, i_2, ...}] 通过连续提取索引为 i_1, i_2, ... 的子表达式的各组成部分，找到 *expr* 的相应的块. 如果用户把表达式当作树来考虑，索引就指明了当用户从树的根节点开始往下搜索的时候，在每个节点应该选择哪个分支.

通过给出由刚好 n 个索引组成的序列来指定的表达式的组成部分被定义为位于表达式中的第 n 层. 用户可以使用层来决定函数如 `Map` 的应用域. 第**0**层对应于整个表达式.

定义表达式的深度为用来指定表达式各部的最大索引数，并加上**1**. 负的层数 $-n$ 指的是表达式的各部分都具有深度 n.

符号

符号（***Symbols***）是 ***Mathematica*** 中的基本已命名对象.

符号名必须是由字母、类字母形式和数字组成的序列，但开头不可以使用数字. 在 ***Mathematica*** 中，总是区分大写字母和小写字母的.

aaaaa	用户定义的符号
Aaaaa	系统定义的符号
$*Aaaa*	全局或内部的系统定义的符号
aaaa$	在作用域结构中重命名的符号
aa$*nn*	在一个模块（**module**）中产生的唯一局部符号

符号名称的使用惯例.

重要的是所有系统定义的符号名称只包含英文字母、数字和 $. 例外情况是 π、∞、e、i 和 j.

系统定义的符号名称按照惯例包含一个或更多个完整英文单词. 每个单词的第一个字母大写，并且这些单词被组合在一起.

一旦创建，***Mathematica*** 中的普通字符便持续存在，除非使用 `Remove` 明确删除它. 然而，在作用域结构如 `Module` 中自动创建的符号具有属性 `Temporary`，该属性指明一旦它们不在任何表达式中出现时，将自动删除它们.

当创建一个新符号的时候，***Mathematica*** 首先把任何赋给 `$NewSymbol` 的值应用于作为符号名的字符串，以及创建符号所在的上下文.

如果打开消息 General::newsym，那么 ***Mathematica*** 报告创建的新符号. 默认情况下，该消息被关闭. 不报告在作用域结构中自动创建的符号.

上下文

在 ***Mathematica*** 中的任何符号的全名包括两个部分：上下文和短名. 全名以 *context*`*name* 的形式给出. 上下文 *context*` 可以包括与短名相同的字符. 它可能包含任何数目的上下文标记字符 `，并且必须使用上下文标记结束.

在一个 ***Mathematica*** 会话的任何位置，有一个现有上下文 $Context 和包含上下文列表的上下文搜索路径 $ContextPath. 在现有上下文或上下文搜索路径的上下文中的符号，可以通过只给出它们的短名来指定，如果它们没有被具有相同短名的其它符号所屏蔽的话.

name	搜索 $ContextPath，接着是 $Context；如果必要的话，在 $Context 中创建
` *name*	只搜索 $Context ；如果必要的话，在该位置创建
context ` *name*	只搜索 *context*；如果必要的话，在该位置创建
` *context* ` *name*	只搜索 $Context` *context*；如果必要的话，在该位置创建

用于不同符号规格的上下文.

上下文相联系. 程序包通常使用 BeginPackage 和 EndPackage 在合适的上下文中定义对象，并且把上下文加入到全局 $ContextPath 中. 对于任何在程序包中创建、但是在上下文搜索路径中被现存符号所“屏蔽”的符号，EndPackage 显示警告信息.

上下文仅当必须在符号显示的时候用来指定该符号时，才被包含在符号的显示形式中.

原子对象

Mathematica 中的所有表达式最终是由小数目的基本或原子对象类型组成.

这些对象具有可以用来作为类型“标签”的符号的头部. 这些对象包含“原始数据”，通常只能由面向特定对象类型的函数访问. 可以使用 Head 提取对象的头部，但是用户不可以直接提取它的其它各组成部分.

`Symbol`	符号（使用 `SymbolName` 获取名称）
`String`	字符字符串 "*cccc*" （使用 `Characters` 获取字符）
`Integer`	整数（使用 `IntegerDigits` 获取数字）
`Real`	近似实数（使用 `RealDigits` 获取数字）
`Rational`	有理数（使用 `Numerator` 和 `Denominator` 获取各部）
`Complex`	复数（使用 `Re` 和 `Im` 获取各部）

原子对象.

Mathematica 中的原子对象具有深度**0**，并且在使用 `AtomQ` 测试时，产生 `True`.

数字

`Integer`	整数 *nnnn*
`Real`	近似实数 *nnn*.*nnn*
`Rational`	有理数 *nnn* / *nnn*
`Complex`	复数 *nnn* + *nnn* `I`

数字的基本类型.

Mathematica 中的所有数字可以包含任意数目的数位. 当可能的情

有理数的复数，进行计算.

在 ***Mathematica*** 中有两种近似实数的类型：任意精度（***arbitrary precision***）和机器精度（***machine precision***）. 在处理任意精度数时，***Mathematica*** 试图修改精度，以确保所有数位都是准确的.

使用机器精度数，所有的计算在相同的固定精度下实现，所以一些数位可能不是准确的.

除非特别指明，***Mathematica*** 把所有位于 `$MinMachineNumber` 和 `$MaxMachineNumber` 之间、并且输入小于 `$MachinePrecision` 数位的近似实数作为机器精度数处理.

在 `InputForm` 中，***Mathematica*** 使用 `$MachinePrecision` 数位显示机器精度数，除了当尾随数字为零的情况.

在 ***Mathematica*** 的任意实现中，数的幅度（除了**0**）必须位于 `$MinNumber` 和 `$MaxNumber` 之间. 幅度在此之外的数字由 `Underflow[]` 和 `Overflow[]` 表示.

字符字符串

Mathematica 中的字符字符串可以包含任意字符序列. 它们以形式 "*ccccc*" 输入.

单个字符可以是可显示的**ASCII** （字符码在**32**和**126**之间），或者一般说来，任意**8**位或**16**位字符. ***Mathematica*** 对**16**位字符使用**Unicode**字符编码.

在输入形式中，在可能的情况下，16位字符可以表示为形式

\[*name*]，否则表示为 \：*nnnn*.

零（**Null**）字节可以出现在 ***Mathematica*** 字符串的任何位置.

教程专集

- **Core Language**

输入语法

- 输入字符
- 输入语法的类别
- 由字符组成的字符串
- 符号名和上下文
- 数
- 括号内的对象
- 二维输入形式
- 框的输入
- 输入表达式的范围
- 特殊输入
- 前端文件

输入字符

- 直接输入（如 +）
- 通过全名输入（如 \[Alpha]）
- 通过别名输入 （如 Esc a Esc ） （仅限于笔记本前端）
- 通过从面板选择输入（仅限于笔记本前端）
- 通过字符码输入（如 \：03 b1）

输入字符的典型方式.

Mathematica 中被赋予明确的名称，使它们即使在没有明确显示的位置也能够在键盘直接输入.

```
     \[RawSpace]              ;   \[RawSemicolon]
!    \[RawExclamation]        <   \[RawLess]
"    \[RawDoubleQuote]        =   \[RawEqual]
#    \[RawNumberSign]         >   \[RawGreater]
$    \[RawDollar]             ?   \[RawQuestion]
%    \[RawPercent]            @   \[RawAt]
&    \[RawAmpersand]          [   \[RawLeftBracket]
'    \[RawQuote]              \   \[RawBackslash]
(    \[RawLeftParenthesis]    ]   \[RawRightBracket]
)    \[RawRightParenthesis]   ^   \[RawWedge]
*    \[RawStar]               _   \[RawUnderscore]
+    \[RawPlus]               `   \[RawBackquote]
,    \[RawComma]              {   \[RawLeftBrace]
-    \[RawDash]               |   \[RawVerticalBar]
.    \[RawDot]                }   \[RawRightBrace]
/    \[RawSlash]              ~   \[RawTilde]
:    \[RawColon]
```

非字母数字**ASCII**字符的全名.

输入 ***Mathematica*** 内核的所有字符都是根据源流的 `CharacterEncoding` 选项的设置进行解释的.

\[*Name*]	具有特定全名的字符
nnn	具有八进制代码 *nnn* 的字符
\.*nn*	具有十六进制代码 *nn* 的字符
\:*nnnn*	具有十六进制代码 *nnnn* 的字符

输入字符的方式.

字符码可以使用 `ToCharacterCode` 产生. 这里遵循具有各种扩展的 **Unicode** 标准

8位字符的编码小于256；16位字符的编码在256和65535之间. 大约有900个字符在 ***Mathematica*** 中被指定明确的名称. 其它字符必须使用各自的字符码输入.

\\	单反斜线（十进制代码92）
\	单一空间（十进制代码32）
\"	双引号（十进制代码34）
\b	退格键或 Ctrl + H （十进制代码8）
\t	tab 或者 Ctrl + I （十进制代码9）
\n	换行符或者 Ctrl + J （十进制代码10；全名 \[NewLine]）
\f	换页或者 Ctrl + L （十进制代码12）
\r	回车或者 Ctrl + M （十进制代码13）
\000	null字节（代码0）

一些特别的8位字符.

输入语法的类别

Mathematica 所采用的标准输入语法是 InputForm 和 StandardForm 中默认使用的. 可以通过定义 MakeExpression[*expr*, *form*] 来修改语法.

可以设置选项来指定什么输入形式应该被一个笔记本中或者从特定流得到的特定单元接受.

例如，在 TraditionalForm 中的输入语法与 InputForm 和

一般情况下，输入语法用来确定一个特定的字符串或框集合应该如何解释为一个表达式. 当使用笔记本前端创建框的时候，可以有隐藏的 InterpretationBox 或者 TagBox 对象修改框的解释.

由字符组成的字符串

"*characters*"	一个由字符组成的字符串
\"	在字符组成的字符串中的 "
\ \	在字符组成的字符串中的 \
\ （在行末尾）	忽略接下来的换行
\! \(...\)	表示二维框的子串

输入由字符组成的字符串.

字符字符串可以包含**8**位或**16**位字符的任意序列. 由名称或字符码输入的字符如同字符直接输入一般存储.

在笔记本前端，默认粘贴到一个字符串的文本自动插入合适的 \ 字符，以使得在 ***Mathematica*** 中存储的字符串重新生成被粘贴的文本.

在 \! \(...\) 中，任何使用**/**序列表示的框结构都可以被采用.

StringExpression 对象可以用来表示包括符号结构（如模式元素等）的字符串.

符号名和上下文

name	符号名
`*name*	在当前上下文中的符号名
context`*name*	在特定上下文中的符号名
context`	上下文名
context1`*context2*`	复合上下文名
`*context*`	与当前上下文相关的上下文

符号名和上下文.

符号名和上下文可以包含由 ***Mathematica*** 作为字母或类字母形式处理的任意字符. 它们可以包含数字，但不使用数字开头. 上下文必须使用 ` 结束.

数

digits	整数
digits.*digits*	近似数
base^^*digits*	指定基数的整数
base^^*digits*.*digits*	指定基数的近似数
*mantissa**^*n*	科学表示法（$mantissa \times 10^n$）
base^^*mantissa**^*n*	指定基数的科学表示法（$mantissa \times base^n$）
number`	机器精度近似数
number`*s*	精度为 *s* 的任意精度数
number``*s*	准确度为 *s* 的任意精度数

数的输入形式.

数字可以使用基数在 2 到 36 之间的表示法 *base*^^*digits* 输入. 基数本身以十进制形式给出. 对于大于 10 的基数，其余的数字从字母 a-z 或 A-Z 中选择. 对于这些用途来说，大写字母和小写字母是等价的. 浮点数字可以通过在 *digits* 序列中包括 . 来指定.

在科学表示法中，*mantissa* 可以包含 ` 记号. 指数 *n* 必须始终是一个整数，用十进制数指定.

精确度或准确度 *s* 可以是任意实数；不必是整数.

在形式 *base*^^*number*`*s* 中，精确度 *s* 以十进制形式给出，但是它给出在特定基数，而非基数**10**上的精确度的数字的有效数目.

`Ceiling[$MachinePrecision] + 1` 或更少，那么它采取机器精度. 如果给出更多的数字，那么 x 作为一个任意精度数采用. x 的准确度为在小数点右边出现的数字数目，而它的精度为 `Log[10, Abs[x]] + Accuracy[x]`.

以 `0``s` 形式输入的数字采用精度**0**和准确度 s.

括号内的对象

括号内的对象使用明确的左右分隔符来表明它们的范围. 它们可以在 ***Mathematica*** 输入的任何位置出现，并且可以被以任何方式嵌套.

在括号内的对象中的分隔符是匹配操作符（***matchfix operators***）. 但是由于这些分隔符明确封装了所有操作数，对这样的操作符，不需要分配任何优先级.

(*any text*)	注释
(*expr*)	括号（**parenthesization**）：输入的分组

元素不使用逗号分隔的括号内的对象.

注释可以嵌套，也可以持续任意行. 它们可以包含**8**位或**16**位字符.

括号必须对单个完整表达式进行封装；既不允许使用 (*e*, *e*)，也不允许 ().

$\{e_1, e_2, \ldots\}$	`List[`$e_1, e_2, \ldots$`]`
$\langle e_1, e_2, \ldots \rangle$	`AngleBracket[`$e_1, e_2, \ldots$`]`
$\lfloor expr \rfloor$	`Floor[`$expr$`]`
$\lceil expr \rceil$	`Ceiling[`$expr$`]`
$\vert e_1, e_2, \ldots \vert$	`BracketingBar[`$e_1, e_2, \ldots$`]`
$\Vert e_1, e_2, \ldots \Vert$	`DoubleBracketingBar[`$e_1, e_2, \ldots$`]`
\(*input*\)	输入或者框的分组

允许使用逗号分隔元素的括号内的对象.

记号 ... 用来表示任何表达式序列.

{e_1, e_2, ...} 可以包含任意数目的元素，使用逗号分隔连续的元素.

{} 是 `List[]`，具有零个元素的列表.

⟨e_1, e_2, ...⟩ 可以作为 \[LeftAngleBracket] e_1, e_2, ... \[RightAngleBracket] 输入.

字符 \[InvisibleComma] 可以和普通的逗号交替使用；唯一的不同是 \[InvisibleComma] 不被显示.

当分隔符是特殊的字符时，传统上，它们被命名为 \[Left*Name*] 和 \[Right*Name*].

\(... \) 用来使用一维字符串输入框. 注意，在所用输入块的最外层的 \(... \) 中的语法与外层稍有不同，如"框的输入" 所述.

$h[e_1, e_2, \ldots]$	标准表达式
$e[[i_1, i_2, \ldots]]$	`Part[e, i1, i2, ...]`
$e[\![i_1, i_2, \ldots]\!]$	`Part[e, i1, i2, ...]`

具有头部的括号内的对象.

具有头部的括号内的对象明确分隔所有操作数，除了头部. 优先级必须分配用来定义头部的范围.

$h[e]$ 的优先级足够高，使得 $!h[e]$ 解释为 `Not[h[e]]`. 然而，$h_s[e]$ 解释为 $(h_s)[e]$.

二维输入形式

		x^y	`Power[x, y]`	$\int_{x_{min}}^{x_{max}} y \, \mathbb{d}x$	I
		$\frac{x}{y}$	`Divide[x, y]`	$\int_{x_{min}}^{x_{max}} \frac{y\,w}{z} \, \mathbb{d}x$	I
		$\sqrt{x}$	`Sqrt[x]`	$\sum_{x=x_{min}}^{x_{max}} y$	S
		$\sqrt[n]{x}$	`Power[x, 1 / n]`		
a_{11}	a_{12}	$\cdots$	`{{a11, a12, ...}, {a21, a22, ...}}`	$\prod_{x=x_{min}}^{x_{max}} y$	P
a_{21}	a_{22}	$\cdots$			
		$\partial_x y$	`D[y, x]`		
		$\partial_{x,\ldots} y$	`D[y, x, ...]`		

采用内置计算规则的二维输入形式.

由 `GridBox` 表示的表达式的任何阵列可以解释为列表的列表. 即使 `GridBox` 只有一行，解释仍然是 $\{\{a_1, a_2, \ldots\}\}$.

在 $\int_{x_{min}}^{x_{max}} y\, w \, \frac{dx}{z}$ 形式中，可以忽略极限 x_{min} 和 x_{max}，正如 y 和 w

$h[e_1, e_2, \ldots]$	标准表达式
$e[[i_1, i_2, \ldots]]$	`Part[e, i1, i2, ...]`
$e\llbracket i_1, i_2, \ldots \rrbracket$	`Part[e, i1, i2, ...]`

具有头部的括号内的对象.

具有头部的括号内的对象明确分隔所有操作数，除了头部. 优先级必须分配用来定义头部的范围.

$h[e]$ 的优先级足够高，使得 $!h[e]$ 解释为 `Not[h[e]]`. 然而，$h_s[e]$ 解释为 $(h_s)[e]$.

二维输入形式

$\int_{x_{min}}^{x_{max}} y \, \mathrm{d}x$	`Integrate[y, {x, xmin, xmax}]`
$\int_{x_{min}}^{x_{max}} \frac{y\,w}{z} \, \mathrm{d}x$	`Integrate[y w / z, {x, xmin, xmax}]`
$\sum_{x=x_{min}}^{x_{max}} y$	`Sum[y, {x, xmin, xmax}]`
$\{a_{21}, a_{22}, \ldots\}\}$ $\prod_{x=x_{min}}^{x_{max}} y$	`Product[y, {x, xmin, xmax}]`

采用内置计算规则的二维输入形式.

由 `GridBox` 表示的表达式的任何阵列可以解释为列表的列表. 即使 `GridBox` 只有一行，解释仍然是 $\{\{a_1, a_2, \ldots\}\}$.

在 $\int_{x_{min}}^{x_{max}} y\,w\,\frac{dx}{z}$ 形式中，可以忽略极限 x_{min} 和 x_{max}，正如 y 和 w

x_{+}	SubPlus[x]	$\underset{y}{x}$	Underscript[x, y]
x_{-}	SubMinus[x]	$\overline{x}$	OverBar[x]
x_{*}	SubStar[x]	$\vec{x}$	OverVector[x]
x^{+}	SuperPlus[x]	$\tilde{x}$	OverTilde[x]
x^{-}	SuperMinus[x]	$\hat{x}$	OverHat[x]
x^{*}	SuperStar[x]	$\dot{x}$	OverDot[x]
$x^{\dagger}$	SuperDagger[x]	$\underline{x}$	UnderBar[x]

不采用内置计算规则的二维输入形式.

对于如 $\sqrt{x}$ 和 $\hat{x}$ 这样的操作数由操作符展开的形式，不存在优先级问题. 对于如 x_y 和 $x^{\dagger}$ 这样左边的优先级不必指定的形式，这样的形式在上述优先级表中包含.

框的输入

- 采用面板
- 采用控制键

输入框的方式.

控制键

Ctrl + 2 或 Ctrl + @	平方根
Ctrl + 5 或 Ctrl + %	转换到另一个位置（如下标到上标）
Ctrl + 6 或 Ctrl + ^	上标
Ctrl + 7 或 Ctrl + &	顶标
Ctrl + 9 或 Ctrl + (	在一个存在的单元中开始新的单元
Ctrl + 0 或 Ctrl +)	在一个存在的单元中结束新的单元
Ctrl + – 或 Ctrl + _	下标
Ctrl + 4 或 Ctrl + $	底标
Ctrl + Enter	在表格中创建一个新行
Ctrl + ,	在表格中创建一个新列
Ctrl + .	扩大当前的选择
Ctrl + /	分数
Ctrl + Space	从当前位置或状态返回
Ctrl + ←, Ctrl + →, Ctrl + ↑, Ctrl + ↓	通过屏幕上最小的增量移动对象

标准控制键.

在英文键盘中，在给出另一种方法的时候，两种形式都可以适用. 在其他键盘上，第一种形式可以使用，但第二种可能不行.

从文本构建的框

当用户给出的文本输入用来构建框，如笔记本中的 `StandardForm` 或者 `TraditionalForm` 单元，对于输入的处理与当它直接输给内核的情况稍有不同.

输入分成不同的令牌（***tokens***），并且每个令牌作为单独的字符字符串包括在框结构中. 因此，例如，`xx + yyy` 被分成令牌`"xx"`，`"+"`，`"yyy"`.

- 符号名（如 `x123`）
- 数字（如 `12.345`）
- 操作符 （如 `+=`）
- 空白（如 `␣`）
- 字符字符串（如 `"text"`）

用来构建框的文本中的令牌类型.

构建 `RowBox` 用来保持每个操作符和操作数. `RowBox` 对象的嵌套由标准 ***Mathematica*** 语法中操作符的优先级决定.

注意，空间字符不自动被丢弃. 相反地，每个这样的字符的序列被作为单独的令牌.

基于字符串的输入

\(...\)	输入原始框
\!\(...\)	输入和解释框

输入原始和解释框.

用户给出的任何位于 \(和 \) 之间的文本输入用来指定构建的框. 只有在用户指定 \! 应该被完成的情况下，才对框进行解释. 否则，例如，x \^ y 被保留作为 SuperscriptBox[x, y]，而没有转换为Power[x, y].

在最外层的 \(...\) 之内，进一步的 \(...\) 指定分组，并且导致 RowBox 对象的插入.

\(*box*$_1$,*box*$_2$,...\)	RowBox[*box*$_1$,*box*$_2$,...]
box$_1$\^*box*$_2$	SuperscriptBox[*box*$_1$,*box*$_2$]
box$_1$_ *box*$_2$	SubscriptBox[*box*$_1$,*box*$_2$]
box$_1$_ *box*$_2$\% *box*$_3$	SubsuperscriptBox[*box*$_1$,*box*$_2$,*box*$_3$]
box$_1$\& *box*$_2$	OverscriptBox[*box*$_1$,*box*$_2$]
box$_1$\+*box*$_2$	UnderscriptBox[*box*$_1$,*box*$_2$]
box$_1$\+*box*$_2$\% *box*$_3$	UnderoverscriptBox[*box*$_1$,*box*$_2$,*box*$_3$]
box$_1$\/*box*$_2$	FractionBox[*box*$_1$,*box*$_2$]
\@*box*	SqrtBox[*box*]
form\` *box*	FormBox[*box*, *form*]
**input*	通过解释 *input* 构建框
\␣	插入一个空格
\n	插入一个新行
\t	在一个行的开头缩进

构建原始框的基于字符串的方式.

在位于 \(和 \) 之间的空间中的字符串输入，制表符和换行符被丢弃. \␣ 可以用来插入一个单一的空格. 特殊间隔字符如 \[ThinSpace]，\[ThickSpace] 或者 \[NegativeThinSpace] 不被忽略.

当用户把排版形式输入为一个字符串，字符串的内部表示使用上述的形式. 前端显示排版形式，但是当把内容保存为一个文件或者当把字符串发送给内核来计算的时候，使用 \(...\) 表示法.

输入表达式的范围

Mathematica 将把用户在单个行的所有输入作为相同表达式的一部分.

Mathematica 允许单个表达式持续多个行. 一般情况下，它把用户在连续的行的输入作为属于同一表达式处理，不论在不这样做的情况下，有没有形成完整的表达式.

因此，例如，如果一个行以 = 结尾，那么 ***Mathematica*** 将假定表达式将在下一行继续. 如果例如括号或其它用以匹配的操作符在行结尾是开放的，那么 ***Mathematica*** 将做同样的假定.

如果用户给出的特定行的末尾对应于一个完整的表达式，那么 ***Mathematica*** 将立即开始执行该表达式.

然而，如果用户通过在行的末尾放置一个 \ 或者一个 ∴ (`\[Continuation]`) 明确告诉 ***Mathematica*** 某个特定的表达式不是完整的. ***Mathematica*** 将在同样的表达式里包含下一行，而放弃在行开头出现的任何空格或制表符.

特殊输入

?symbol	获取信息
??symbol	获取更多信息
$?s_1\ s_2\ \ldots$	在一些对象上获取信息
!command	执行外部命令（仅限于基于文本的界面）
!! file	显示一个外部文件的内容（仅限于基于文本的界面）

特殊输入行.

在 ***Mathematica*** 的绝大多数实现中，用户可以在输入的任何位置中给出特殊输入行. 唯一的限制是特殊输入必须始于一个行的开头.

一些 ***Mathematica*** 实现可能不允许用户使用 ！*command* 执行外部命令.

前端文件

笔记本文件和前端初始文件可以包含标准 ***Mathematica*** 语言语法的一个子集. 该语法包括：

- 形式为 `FullForm` 的任意 ***Mathematica*** 表达式.
- 以 {...} 形式表示的列表. 操作符 `->`，`:>` 和 **&.** 表示函数**slot**的 ⌗ 形式.
- 各种 ***Mathematica*** 操作符，如 `+`，`*`，`;`，等等.

- 以 \[*Name*]，\：*nnnn* 或者 \.*xx* 形式表示的特殊字符.
- 涉及 \(，\）和其它操作符的框（**box**）的字符串表示.
- 由 (* 和 *) 分隔的 ***Mathematica*** 注释.

相关指南

- 底层笔记本结构
- 语法

相关教程

- 框符的字符串表示
- 运算符的输入形式

教程专集

- **Core Language**

运算符的输入形式

非字母形式或结构元素的字母的字符在 ***Mathematica*** 中被当作运算符. ***Mathematica*** 具有内置的规则来解释所有运算符. 与这些运算符对应的函数可能具有或不具有内嵌赋值或其它规则. 内置意义缺省被定义的情形由下表中 ◃ 的指示给出.

创建二维盒框的运算符（所有盒框的名称都以反斜线符号开头）只能被用在 \(...\) 内部. 下表给出这些在 \！\(...\) 内的运算符的

expr 和 $expr_i$	任意表达式
symb	任意符号
patt	任意模式对象
string 和 $string_i$	"*cccc*" 或字母、字母型及数字序列
filename	与 *string* 类似，但可以额外包括下面的字符
◁	有内部定义

在运算符输入形式表中所使用的对象.

运算符优先次序

运算符形式	完整形式	分组
数的表示形式 (见数)		◁
符号的表示形式 (见符号名和上下文)		◁
字符串的表示形式 (见字符串)		◁
e_{11} e_{12} … e_{21} e_{22} … …	{{e_{11}, e_{12}, …}, {e_{21}, e_{22}, …}, …}	◁
$\begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ \dots \end{bmatrix}$	Piecewise[{{e_{11}, e_{12}}, {e_{21}, e_{22}}, …}]	◁
expr::*string*	MessageName[*expr*, "*string*"]	◁
expr::$string_1$::$string_2$	MessageName[*expr*, "$string_1$", "$string_2$"]	◁
包含 ⌗ 的形式(见额外的输入形式)		◁

包含 _ 的形式(见额外的输入形式)			◁
<< *filename*	`Get[" filename"]`		◁
$expr_1^{expr_2}$	`Overscript[` *expr*$_1$`,` *expr*$_2$`]`	$\binom{e}{e}$ e	
$expr_{1\backslash\&expr_2}$	`Overscript[` *expr*$_1$`,` *expr*$_2$`]`	$e\backslash\&(e\backslash\&e)$	
$expr_1$ $expr_2$	`Underscript[` *expr*$_1$`,` *expr*$_2$`]`	e $\binom{e}{e}$	
$expr_{1\backslash+expr_2}$	`Underscript[` *expr*$_1$`,` *expr*$_2$`]`	$e\backslash+(e\backslash+e)$	
$expr_1^{expr_3}$ $expr_2$	`Underoverscript[` *expr*$_1$`,` *expr*$_2$`,` *expr*$_3$`]`		
$expr_{1\backslash+expr_2\backslash\%expr_3}$	`Underoverscript[` *expr*$_1$`,` *expr*$_2$`,` *expr*$_3$`]`		
$expr_{1\backslash\&expr_2\backslash\%expr_3}$	`Underoverscript[` *expr*$_1$`,` *expr*$_3$`,` *expr*$_2$`]`		
$expr_{1expr_2}$	`Subscript[`*expr*$_1$`,` *expr*$_2$`]`	$e_{(e_e)}$	
$expr_{1\backslash_expr_2}$	`Subscript[`*expr*$_1$`,` *expr*$_2$`]`	$e\backslash_(e\backslash_e)$	
$expr_{1\backslash_expr_2\backslash\%expr_3}$	`Power[Subscript[` *expr*$_1$`,` *expr*$_2$`],` *expr*$_3$`]`		◁
\!*boxes*	(*boxes* 的解释版本)		
$expr_1?expr_2$	`PatternTest[` *expr*$_1$`,` *expr*$_2$`]`		◁
$expr_1[expr_2,\ldots]$	$expr_1[expr_2,\ldots]$	$(e[e])[e]$	◁
$expr_1[[expr_2,\ldots]]$	`Part[`*expr*$_1$`,` *expr*$_2$`,` …`]`	$(e[[e]])[[e]]$	◁
$expr_1[\![expr_2,\ldots]\!]$	`Part[`*expr*$_1$`,` *expr*$_2$`,` …`]`	$(e[\![e]\!])[\![e]\!]$	◁
$expr_{1[\![}$	`Part[`*expr*$_1$ *expr*$_2$	$(e_{[\![e]\!]})_{[\![}$	◁

包含 _ 的形式(见额外的输入形式)			◁
<< *filename*	`Get[" filename"]`		◁
$expr_2$ $expr_1$	`Overscript[` $expr_1$`,`$expr_2$`]`	$\binom{e}{e}$ e	
$expr_{1\backslash\&expr_2}$	`Overscript[` $expr_1$`,`$expr_2$`]`	$e\backslash\&(e\backslash\&e)$	
$expr_1$ $expr_2$	`Underscript[` $expr_1$`,`$expr_2$`]`	e $\binom{e}{e}$	
$expr_{1\backslash+expr_2}$	`Underscript[` $expr_1$`,`$expr_2$`]`	$e\backslash+(e\backslash+e)$	
$expr_3$ $expr_1$ $expr_2$	`Underoverscript[` $expr_1$`,`$expr_2$`,`$expr_3$`]`		
$expr_{1\backslash+expr_2\backslash\%expr_3}$	`Underoverscript[` $expr_1$`,`$expr_2$`,`$expr_3$`]`		
$expr_{1\backslash\&expr_2\backslash\%expr_3}$	`Underoverscript[` $expr_1$`,`$expr_3$`,`$expr_2$`]`		
$expr_{1expr_2}$	`Subscript[`$expr_1$`,`$expr_2$`]`	$e_{(e_e)}$	
$expr_{1\backslash_expr_2}$	`Subscript[`$expr_1$`,`$expr_2$`]`	$e\backslash_(e\backslash_e)$	
$expr_{1\backslash_expr_2\backslash\%expr_3}$	`Power[Subscript[` $expr_1$`,`$expr_2$`],`$expr_3$`]`		◁
\!*boxes*	(*boxes* 的解释版本)		
$expr_1?expr_2$	`PatternTest[` $expr_1$`,`$expr_2$`]`		◁
$expr_1[expr_2,\ldots]$	$expr_1[expr_2,\ldots]$	$(e[e])[e]$	◁
$expr_1[[$ $expr_2,\ldots]]$	`Part[`$expr_1$`,`$expr_2$`,`$\ldots$`]`	$(e[[$ $e]])$ $[[$ $e]]$	◁
$expr_1[\![expr_2,\ldots]\!]$	`Part[`$expr_1$`,`$expr_2$`,`$\ldots$`]`	$(e[\![e]\!])[\![e]\!]$	◁
$expr_1[\![$	`Part[`$expr_1$`,`$expr_2$	$(e[\![$	◁

*expr	(由 expr 构建的盒框)		
expr++	Increment[expr]		◁
expr−−	Decrement[expr]		◁
++expr	PreIncrement[expr]		◁
−−expr	PreDecrement[expr]		◁
$expr_1$@$expr_2$	$expr_1$[$expr_2$]	e@(e@e)	◁
$expr_1$ $expr_2$	(不可见应用，输入如 $expr_1$ Esc @ Esc $expr_2$) $expr_1$[$expr_2$]		◁
$expr_1$~$expr_2$~$expr_3$	$expr_2$[$expr_1$, $expr_3$]	(e~e~e)~e~e	◁
$expr_1$/@$expr_2$	Map[$expr_1$, $expr_2$]	e/@(e/@e)	◁
$expr_1$//@$expr_2$	MapAll[$expr_1$, $expr_2$]	e//@(e//@e)	◁
$expr_1$@@$expr_2$	Apply[$expr_1$, $expr_2$]	e@@(e@@e)	◁
$expr_1$@@@$expr_2$	Apply[$expr_1$, $expr_2$, {1}]	e@@@(e@@@e)	◁
expr!	Factorial[expr]		◁
expr!!	Factorial2[expr]		◁
expr*	Conjugate[expr]		◁
$expr^{T}$	Transpose[expr]		◁
$expr^{\dagger}$	ConjugateTranspose[expr]		◁
$expr^{H}$	ConjugateTranspose[expr]		◁
expr'	Derivative[1][expr]		◁
expr'' …' (n 倍)	Derivative[n][expr]		◁
$expr_1$<> $expr_2$<><>$expr_3$	StringJoin[$expr_1$, $expr_2$, $expr_3$]	e<><>e<><>e	◁

*expr	(由 *expr* 构建的盒框)		
expr++	Increment[*expr*]		◁
expr−−	Decrement[*expr*]		◁
++expr	PreIncrement[*expr*]		◁
−−expr	PreDecrement[*expr*]		◁
$expr_1$@$expr_2$	$expr_1$[$expr_2$]	e@(e@e)	◁
$expr_1$ $expr_2$	(不可见应用，输入如 $expr_1$ Esc @ Esc $expr_2$) $expr_1$[$expr_2$]		◁
$expr_1$~$expr_2$~$expr_3$	$expr_2$[$expr_1$, $expr_3$]	(e~e~e)~e~e	◁
$expr_1$/@$expr_2$	Map[$expr_1$, $expr_2$]	e/@(e/@e)	◁
$expr_1$//@$expr_2$	MapAll[$expr_1$, $expr_2$]	e//@(e//@e)	◁
$expr_1$@@$expr_2$	Apply[$expr_1$, $expr_2$]	e@@(e@@e)	◁
$expr_1$@@@$expr_2$	Apply[$expr_1$, $expr_2$, {1}]	e@@@(e@@@e)	◁
expr!	Factorial[*expr*]		◁
expr!!	Factorial2[*expr*]		◁
expr*	Conjugate[*expr*]		◁
$expr^{\mathsf{T}}$	Transpose[*expr*]		◁
$expr^{\dagger}$	ConjugateTranspose[*expr*]		◁
$expr^{\mathsf{H}}$	ConjugateTranspose[*expr*]		◁
expr'	Derivative[1][*expr*]		◁
expr'' …' (*n* 倍)	Derivative[*n*][*expr*]		◁
$expr_1$<> $expr_2$<><$expr_3$	StringJoin[$expr_1$, $expr_2$, $expr_3$]	e<><e<><e	◁

$expr_1^{expr_2}$	`Power[`$expr_1$`,`$expr_2$`]`	$e^{(e^e)}$	◁
$expr_{1\,expr_2}^{expr_3}$	`Power[Subscript[`$expr_1$`,`$expr_2$`],`$expr_3$`]`		◁
$expr_1$ `\^` $expr_{2}$ `\%`$expr_3$	`Power[Subscript[`$expr_1$`,`$expr_3$`],`$expr_2$`]`		◁
竖直箭头和向量算符			
$\sqrt{expr}$	`Sqrt[`$expr$`]`	$\sqrt{\left(\sqrt{e}\right)}$	◁
`\@` $expr$	`Sqrt[`$expr$`]`	`\@(\@` e`)`	◁
`\@` $expr$`\%`n	`Power[`$expr$`,1/`n`]`		◁
$\int expr_1 \,\mathbb{d}\, expr_2$	`Integrate[`$expr_1$`,`$expr_2$`]`	$\int \left(\int e \,\mathbb{d}\, e\right) \mathbb{d}\, e$	◁
$\int_{e_1}^{e_2} e_3 \,\mathbb{d}e_4$	`Integrate[`e_3`,{`e_4`,`e_1`,`e_2`}]`	$\int \left(\int e \,\mathbb{d}\, e\right) \mathbb{d}\, e$	◁
其它积分算符			
$\partial_{expr_1} expr_2$	`D[`$expr_2$`,`$expr_1$`]`	$\partial_e(\partial_e e)$	◁
∇ $expr$	`Del[`$expr$`]`	$\nabla(\nabla e)$	
$\mathbb{E}_{expr_1} expr_2$	`DiscreteShift[`$expr_2$`,`$expr_1$`]`	$\mathbb{E}_e(\mathbb{E}_e e)$	◁
$\Theta_{expr_1} expr_2$	`DiscreteRatio[`$expr_2$`,`$expr_1$`]`	$\Theta_e(\Theta_e e)$	◁
$\Delta_{expr_1} expr_2$	`DifferenceDelta[`$expr_2$`,`$expr_1$`]`	$\Delta_e(\Delta_e e)$	◁
□ $expr$	`Square[`$expr$`]`	□(□ e)	
$expr_1$ ∘ $expr_2$ ∘ $expr_3$	`SmallCircle[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \circ e \circ e$	
$expr_1$ ⊙ $expr_2$ ⊙ $expr_3$	`CircleDot[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \odot e \odot e$	

$expr_1^{expr_2}$	`Power[expr1, expr2]`	$e^{(e^e)}$	◁
$expr_{1\,expr_2}^{expr_3}$	`Power[Subscript[expr1, expr2], expr3]`		◁
$expr_1 \backslash\hat{}\ expr_{2 \backslash\% expr_3}$	`Power[Subscript[expr1, expr3], expr2]`		◁
竖直箭头和向量算符			
$\sqrt{expr}$	`Sqrt[expr]`	$\sqrt{(\sqrt{e})}$	◁
$\backslash@\ expr$	`Sqrt[expr]`	$\backslash@(\backslash@\ e)$	◁
$\backslash@\ expr \backslash\% n$	`Power[expr, 1/n]`		◁
$\int expr_1 \,\mathbb{d}\, expr_2$	`Integrate[expr1, expr2]`	$\int \left(\int e \,\mathbb{d}\, e\right) \mathbb{d}\, e$	◁
$\int_{e_1}^{e_2} e_3 \,\mathbb{d} e_4$	`Integrate[e3, {e4, e1, e2}]`	$\int \left(\int e \,\mathbb{d}\, e\right) \mathbb{d}\, e$	◁
其它积分算符			
$\partial_{expr_1} expr_2$	`D[expr2, expr1]`	$\partial_e(\partial_e e)$	◁
$\nabla\ expr$	`Del[expr]`	$\nabla(\nabla e)$	
$\Delta_{expr_1} expr_2$	`DiscreteShift[expr2, expr1]`	$\mathbb{E}_e(\mathbb{E}_e e)$	◁
$\Theta_{expr_1} expr_2$	`DiscreteRatio[expr2, expr1]`	$\Theta_e(\Theta_e e)$	◁
$\Delta_{expr_1} expr_2$	`DifferenceDelta[expr2, expr1]`	$\Delta_e(\Delta_e e)$	◁
$\square\ expr$	`Square[expr]`	$\square(\square\ e)$	
$expr_1 \circ expr_2 \circ expr_3$	`SmallCircle[expr1, expr2, expr3]`	$e \circ e \circ e$	
$expr_1 \odot expr_2 \odot expr_3$	`CircleDot[expr1, expr2, expr3]`	$e \odot e \odot e$	

$expr_1$ ** $expr_2$ ** $expr_3$	NonCommutativeMultiply[$expr_1$, $expr_2$, $expr_3$]	e ** e ** e	
$expr_1 \times expr_2 \times expr_3$	Cross[$expr_1$, $expr_2$, $expr_3$]	$e \times e \times e$	◁
$expr_1$. $expr_2$. $expr_3$	Dot[$expr_1$, $expr_2$, $expr_3$]	e . e . e	◁
$-expr$	Times[-1, $expr$]		◁
$+expr$	$expr$		◁
$\pm expr$	PlusMinus[$expr$]		
$\mp expr$	MinusPlus[$expr$]		
$expr_1 / expr_2$	$expr_1$ ($expr_2$)^-1	(e / e) / e	◁
$expr_1 \div expr_2$	Divide[$expr_1$, $expr_2$]	(e ÷ e) ÷ e	◁
$expr_1$ \/ $expr_2$	Divide[$expr_1$, $expr_2$]	(e \/ e) \/ e	◁
$expr_1$ \ $expr_2$ \ $expr_3$	Backslash[$expr_1$, $expr_2$, $expr_3$]	e \ e \ e	
$expr_1 \diamond expr_2 \diamond expr_3$	Diamond[$expr_1$, $expr_2$, $expr_3$]	$e \diamond e \diamond e$	
$expr_1 \wedge expr_2 \wedge expr_3$	Wedge[$expr_1$, $expr_2$, $expr_3$]	$e \wedge e \wedge e$	
$expr_1 \vee expr_2 \vee expr_3$	Vee[$expr_1$, $expr_2$, $expr_3$]	$e \vee e \vee e$	
$expr_1 \otimes expr_2 \otimes expr_3$	CircleTimes[$expr_1$, $expr_2$, $expr_3$]	$e \otimes e \otimes e$	
$expr_1 \cdot expr_2 \cdot expr_3$	CenterDot[$expr_1$, $expr_2$, $expr_3$]	$e \cdot e \cdot e$	
$expr_1$ $expr_2$ $expr_3$	Times[$expr_1$, $expr_2$, $expr_3$]	e e e	◁
$expr_1 * expr_2 * expr_3$	Times[$expr_1$, $expr_2$, $expr_3$]	$e * e * e$	◁
$expr_1 \times expr_2 \times expr_3$	Times[$expr_1$, $expr_2$, $expr_3$]	$e \times e \times e$	◁

$expr_1$ ** $expr_2$ ** $expr_3$	NonCommutativeMultiply [$expr_1$, $expr_2$, $expr_3$]	e ** e ** e	
$expr_1$ × $expr_2$ × $expr_3$	Cross [$expr_1$, $expr_2$, $expr_3$]	e × e × e	◁
$expr_1$. $expr_2$. $expr_3$	Dot [$expr_1$, $expr_2$, $expr_3$]	e . e . e	◁
- $expr$	Times [-1, $expr$]		◁
+ $expr$	$expr$		◁
± $expr$	PlusMinus [$expr$]		
∓ $expr$	MinusPlus [$expr$]		
$expr_1$ / $expr_2$	$expr_1$ ($expr_2$)^-1	(e / e) / e	◁
$expr_1$ ÷ $expr_2$	Divide [$expr_1$, $expr_2$]	(e ÷ e) ÷ e	◁
$expr_1$ \/ $expr_2$	Divide [$expr_1$, $expr_2$]	(e \/ e) \/ e	◁
$expr_1$ \ $expr_2$ \ $expr_3$	Backslash [$expr_1$, $expr_2$, $expr_3$]	e \ e \ e	
$expr_1$ ⋄ $expr_2$ ⋄ $expr_3$	Diamond [$expr_1$, $expr_2$, $expr_3$]	e ⋄ e ⋄ e	
$expr_1$ ∧ $expr_2$ ∧ $expr_3$	Wedge [$expr_1$, $expr_2$, $expr_3$]	e ∧ e ∧ e	
$expr_1$ ∨ $expr_2$ ∨ $expr_3$	Vee [$expr_1$, $expr_2$, $expr_3$]	e ∨ e ∨ e	
$expr_1$ ⊗ $expr_2$ ⊗ $expr_3$	CircleTimes [$expr_1$, $expr_2$, $expr_3$]	e ⊗ e ⊗ e	
$expr_1$ · $expr_2$ · $expr_3$	CenterDot [$expr_1$, $expr_2$, $expr_3$]	e · e · e	
$expr_1$ $expr_2$ $expr_3$	Times [$expr_1$, $expr_2$, $expr_3$]	e e e	◁
$expr_1$ * $expr_2$ * $expr_3$	Times [$expr_1$, $expr_2$, $expr_3$]	e * e * e	◁
$expr_1$ × $expr_2$ × $expr_3$	Times [$expr_1$, $expr_2$, $expr_3$]	e × e × e	◁

$\prod_{e_1=e_2}^{e_3} e_4$	`Product[`e_4`, {`e_1, e_2, e_3`}]`	$\prod(\prod\ e)$	◁
$expr_1 \wr expr_2 \wr expr_3$	`VerticalTilde[`$expr_1, expr_2, expr_3$`]`	$e \wr e \wr e$	
$expr_1 \amalg expr_2 \amalg expr_3$	`Coproduct[`$expr_1, expr_2, expr_3$`]`	$e \amalg e \amalg e$	
$expr_1 \frown expr_2 \frown expr_3$	`Cap[`$expr_1, expr_2, expr_3$`]`	$e \frown e \frown e$	
$expr_1 \smile expr_2 \smile expr_3$	`Cup[`$expr_1, expr_2, expr_3$`]`	$e \smile e \smile e$	
$expr_1 \oplus expr_2 \oplus expr_3$	`CirclePlus[`$expr_1, expr_2, expr_3$`]`	$e \oplus e \oplus e$	
$expr_1 \ominus expr_2$	`CircleMinus[`$expr_1, expr_2$`]`	$(e \ominus e) \ominus e$	
$\sum_{e_1=e_2}^{e_3} e_4$	`Sum[`e_4`, {`e_1, e_2, e_3`}]`	$\sum(\sum\ e)$	◁
$expr_1 + expr_2 + expr_3$	`Plus[`$expr_1, expr_2, expr_3$`]`	$e+e+e$	◁
$expr_1 - expr_2$	$expr_1 + (-1\ expr_2)$	$(e-e)-e$	◁
$expr_1 \pm expr_2$	`PlusMinus[`$expr_1, expr_2$`]`	$(e \pm e) \pm e$	
$expr_1 \mp expr_2$	`MinusPlus[`$expr_1, expr_2$`]`	$(e \mp e) \mp e$	
$expr_1 \cap expr_2$	`Intersection[`$expr_1, expr_2$`]`	$e \cap e \cap e$	◁
其它交集算符			
$expr_1 \cup expr_2$	`Union[`$expr_1, expr_2$`]`	$e \cup e \cup e$	◁
其它并集算符			
$i;;j;;k$	`Span[`i, j, k`]`	$e;;e;;e$	◁
$expr_1 == expr_2$	`Equal[`$expr_1, expr_2$`]`	$e==e==e$	◁
$expr_1 == expr_2$	`Equal[`$expr_1, expr_2$`]`	$e==e==e$	◁
$expr_1 = expr_2$	`Equal[`$expr_1, expr_2$`]`	$e=e=e$	◁
$expr_1 != expr_2$	`Unequal[`$expr_1, expr_2$`]`	$e!=e!=e$	◁
$expr_1 != expr_2$	`Unequal[`$expr_1, expr_2$`]`	$e!=e!=e$	◁
其它相等和类似算符			

$\prod_{e_1=e_2}^{e_3} e_4$	`Product[`e_4`, {`e_1`,`e_2`,`e_3`}]`	$\prod(\prod\ e)$	◁
$expr_1 \wr expr_2 \wr expr_3$	`VerticalTilde[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \wr e \wr e$	
$expr_1 \amalg expr_2 \amalg expr_3$	`Coproduct[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \amalg e \amalg e$	
$expr_1 \frown expr_2 \frown expr_3$	`Cap[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \frown e \frown e$	
$expr_1 \smile expr_2 \smile expr_3$	`Cup[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \smile e \smile e$	
$expr_1 \oplus expr_2 \oplus expr_3$	`CirclePlus[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e \oplus e \oplus e$	
$expr_1 \ominus expr_2$	`CircleMinus[`$expr_1$`,`$expr_2$`]`	$(e \ominus e) \ominus e$	
$\sum_{e_1=e_2}^{e_3} e_4$	`Sum[`e_4`, {`e_1`,`e_2`,`e_3`}]`	$\sum(\sum\ e)$	◁
$expr_1 + expr_2 + expr_3$	`Plus[`$expr_1$`,`$expr_2$`,`$expr_3$`]`	$e+e+e$	◁
$expr_1 - expr_2$	$expr_1 + (-1\ expr_2)$	$(e-e)-e$	◁
$expr_1 \pm expr_2$	`PlusMinus[`$expr_1$`,`$expr_2$`]`	$(e \pm e) \pm e$	
$expr_1 \mp expr_2$	`MinusPlus[`$expr_1$`,`$expr_2$`]`	$(e \mp e) \mp e$	
$expr_1 \cap expr_2$	`Intersection[`$expr_1$`,`$expr_2$`]`	$e \cap e \cap e$	◁
其它交集算符			
$expr_1 \cup expr_2$	`Union[`$expr_1$`,`$expr_2$`]`	$e \cup e \cup e$	◁
其它并集算符			
$i;;j;;k$	`Span[`i`,`j`,`k`]`	$e;;e;;e$	◁
$expr_1 == expr_2$	`Equal[`$expr_1$`,`$expr_2$`]`	$e==e==e$	◁
$expr_1 == expr_2$	`Equal[`$expr_1$`,`$expr_2$`]`	$e==e==e$	◁
$expr_1 = expr_2$	`Equal[`$expr_1$`,`$expr_2$`]`	$e=e=e$	◁
$expr_1 != expr_2$	`Unequal[`$expr_1$`,`$expr_2$`]`	$e!=e!=e$	◁
$expr_1 != expr_2$	`Unequal[`$expr_1$`,`$expr_2$`]`	$e!=e!=e$	◁
其它相等和类似算符			

$expr_1>=expr_2$	GreaterEqual[$expr_1, expr_2$]	$e>=e>=e$	◁
$expr_1\geq expr_2$	GreaterEqual[$expr_1, expr_2$]	$e\geq e\geq e$	◁
$expr_1\geqslant expr_2$	GreaterEqual[$expr_1, expr_2$]	$e\geqslant e\geqslant e$	◁
$expr_1<expr_2$	Less[$expr_1, expr_2$]	$e<e<e$	◁
$expr_1<=expr_2$	LessEqual[$expr_1, expr_2$]	$e<=e<=e$	◁
$expr_1\leq expr_2$	LessEqual[$expr_1, expr_2$]	$e\leq e\leq e$	◁
$expr_1\leqslant expr_2$	LessEqual[$expr_1, expr_2$]	$e\leqslant e\leqslant e$	◁
其它排序算符			
$expr_1 \shortmid expr_2$	VerticalBar[$expr_1, expr_2$]	$e\shortmid e\shortmid e$	
$expr_1 \nmid expr_2$	NotVerticalBar[$expr_1, expr_2$]	$e\nmid e\nmid e$	
$expr_1 \shortparallel expr_2$	DoubleVerticalBar[$expr_1, expr_2$]	$e\shortparallel e\shortparallel e$	
$expr_1 \nshortparallel expr_2$	NotDoubleVerticalBar [$expr_1, expr_2$]	$e\nshortparallel e\nshortparallel e$	
水平箭头和向量算符			
对角箭头算符			
$expr_1===expr_2$	SameQ[$expr_1, expr_2$]	$e===e===e$	◁
$expr_1=!=expr_2$	UnsameQ[$expr_1, expr_2$]	$e=!=e=!=e$	◁
$expr_1\in expr_2$	Element[$expr_1, expr_2$]	$e\in e\in e$	◁
$expr_1\notin expr_2$	NotElement[$expr_1, expr_2$]	$e\notin e\notin e$	◁
$expr_1\subset expr_2$	Subset[$expr_1, expr_2$]	$e\subset e\subset e$	
$expr_1\supset expr_2$	Superset[$expr_1, expr_2$]	$e\supset e\supset e$	
其它集合关系算符			

$expr_1 >= expr_2$	GreaterEqual[$expr_1$, $expr_2$]	$e>=e>=e$	◁
$expr_1 \geq expr_2$	GreaterEqual[$expr_1$, $expr_2$]	$e\geq e\geq e$	◁
$expr_1 \geqslant expr_2$	GreaterEqual[$expr_1$, $expr_2$]	$e\geqslant e\geqslant e$	◁
$expr_1 < expr_2$	Less[$expr_1$, $expr_2$]	$e<e<e$	◁
$expr_1 <= expr_2$	LessEqual[$expr_1$, $expr_2$]	$e<=e<=e$	◁
$expr_1 \leq expr_2$	LessEqual[$expr_1$, $expr_2$]	$e\leq e\leq e$	◁
$expr_1 \leqslant expr_2$	LessEqual[$expr_1$, $expr_2$]	$e\leqslant e\leqslant e$	◁

其它排序算符

$expr_1 \shortmid expr_2$	VerticalBar[$expr_1$, $expr_2$]	$e\shortmid e\shortmid e$	
$expr_1 \nmid expr_2$	NotVerticalBar[$expr_1$, $expr_2$]	$e\nmid e\nmid e$	
$expr_1 \shortmid\shortmid expr_2$	DoubleVerticalBar[$expr_1$, $expr_2$]	$e\shortmid\shortmid e\shortmid\shortmid e$	
$expr_1 \nshortmid expr_2$	NotDoubleVerticalBar[$expr_1$, $expr_2$]	$e\nshortmid e\nshortmid e$	

水平箭头和向量算符

对角箭头算符

$expr_1 === expr_2$	SameQ[$expr_1$, $expr_2$]	$e===e===e$	◁
$expr_1 =!= expr_2$	UnsameQ[$expr_1$, $expr_2$]	$e=!=e=!=e$	◁
$expr_1 \in expr_2$	Element[$expr_1$, $expr_2$]	$e\in e\in e$	◁
$expr_1 \notin expr_2$	NotElement[$expr_1$, $expr_2$]	$e\notin e\notin e$	◁
$expr_1 \subset expr_2$	Subset[$expr_1$, $expr_2$]	$e\subset e\subset e$	
$expr_1 \supset expr_2$	Superset[$expr_1$, $expr_2$]	$e\supset e\supset e$	

其它集合关系算符

$\exists_{expr_1} expr_2$	`Exists`[$expr_1$, $expr_2$]	$\exists_e (\exists_e e)$	◁
$\nexists_{expr_1} expr_2$	`NotExists`[$expr_1$, $expr_2$]	$\nexists_e (\nexists_e e)$	
$!expr$	`Not`[$expr$]	$!(!e)$	◁
$\neg expr$	`Not`[$expr$]	$\neg(\neg e)$	◁
$expr_1 \&\& expr_2 \&\& expr_3$	`And`[$expr_1$, $expr_2$, $expr_3$]	$e \&\& e \&\& e$	◁
$expr_1 \wedge expr_2 \wedge expr_3$	`And`[$expr_1$, $expr_2$, $expr_3$]	$e \wedge e \wedge e$	◁
$expr_1 \barwedge expr_2 \barwedge expr_3$	`Nand`[$expr_1$, $expr_2$, $expr_3$]	$e \barwedge e \barwedge e$	◁
$expr_1 \veebar expr_2 \veebar expr_3$	`Xor`[$expr_1$, $expr_2$, $expr_3$]	$e \veebar e \veebar e$	◁
$expr_1 \barvee expr_2 \barvee expr_3$	`Xnor`[$expr_1$, $expr_2$, $expr_3$]	$e \barvee e \barvee e$	◁
$expr_1 \mid\mid expr_2 \mid\mid expr_3$	`Or`[$expr_1$, $expr_2$, $expr_3$]	$e \mid\mid e \mid\mid e$	◁
$expr_1 \vee expr_2 \vee expr_3$	`Or`[$expr_1$, $expr_2$, $expr_3$]	$e \vee e \vee e$	◁
$expr_1 \barvee expr_2 \barvee expr_3$	`Nor`[$expr_1$, $expr_2$, $expr_3$]	$e \barvee e \barvee e$	◁
$expr_1 \Leftrightarrow expr_2 \Leftrightarrow expr_3$	`Equivalent`[$expr_1$, $expr_2$, $expr_3$]	$e \Leftrightarrow e \Leftrightarrow e$	◁
$expr_1 \Rightarrow expr_2$	`Implies`[$expr_1$, $expr_2$]	$e \Rightarrow (e \Rightarrow e)$	◁
$expr_1 \supset expr_2$	`Implies`[$expr_1$, $expr_2$]	$e \supset e \supset e$	◁
$expr_1 \vdash expr_2$	`RightTee`[$expr_1$, $expr_2$]	$e \vdash (e \vdash e)$	
$expr_1 \models expr_2$	`DoubleRightTee`[$expr_1$, $expr_2$]	$e \models (e \models e)$	
$expr_1 \dashv expr_2$	`LeftTee`[$expr_1$, $expr_2$]	$(e \dashv e) \dashv e$	
$expr_1 \dashv expr_2$	`DoubleLeftTee`[$expr_1$, $expr_2$]	$(e \dashv e) \dashv e$	
$expr_1 \ni expr_2$	`SuchThat`[$expr_1$, $expr_2$]	$e \ni (e \ni e)$	

$\exists_{expr_1} expr_2$	`Exists`[$expr_1$, $expr_2$]	$\exists_e (\exists_e e)$	◁
$\nexists_{expr_1} expr_2$	`NotExists`[$expr_1$, $expr_2$]	$\nexists_e (\nexists_e e)$	
$!expr$	`Not`[$expr$]	$!(!e)$	◁
$\neg expr$	`Not`[$expr$]	$\neg(\neg e)$	◁
$expr_1 \&\& expr_2 \&\& expr_3$	`And`[$expr_1$, $expr_2$, $expr_3$]	$e \&\& e \&\& e$	◁
$expr_1 \wedge expr_2 \wedge expr_3$	`And`[$expr_1$, $expr_2$, $expr_3$]	$e \wedge e \wedge e$	◁
$expr_1 \barwedge expr_2 \barwedge expr_3$	`Nand`[$expr_1$, $expr_2$, $expr_3$]	$e \barwedge e \barwedge e$	◁
$expr_1 \veebar expr_2 \veebar expr_3$	`Xor`[$expr_1$, $expr_2$, $expr_3$]	$e \veebar e \veebar e$	◁
$expr_1 \barvee expr_2 \barvee expr_3$	`Xnor`[$expr_1$, $expr_2$, $expr_3$]	$e \barvee e \barvee e$	◁
$expr_1 \mid\mid expr_2 \mid\mid expr_3$	`Or`[$expr_1$, $expr_2$, $expr_3$]	$e \mid\mid e \mid\mid e$	◁
$expr_1 \vee expr_2 \vee expr_3$	`Or`[$expr_1$, $expr_2$, $expr_3$]	$e \vee e \vee e$	◁
$expr_1 \barvee expr_2 \barvee expr_3$	`Nor`[$expr_1$, $expr_2$, $expr_3$]	$e \barvee e \barvee e$	◁
$expr_1 \Leftrightarrow expr_2 \Leftrightarrow expr_3$	`Equivalent`[$expr_1$, $expr_2$, $expr_3$]	$e \Leftrightarrow e \Leftrightarrow e$	◁
$expr_1 \Rightarrow expr_2$	`Implies`[$expr_1$, $expr_2$]	$e \Rightarrow (e \Rightarrow e)$	◁
$expr_1 \supset expr_2$	`Implies`[$expr_1$, $expr_2$]	$e \supset e \supset e$	◁
$expr_1 \vdash expr_2$	`RightTee`[$expr_1$, $expr_2$]	$e \vdash (e \vdash e)$	
$expr_1 \models expr_2$	`DoubleRightTee`[$expr_1$, $expr_2$]	$e \models (e \models e)$	
$expr_1 \dashv expr_2$	`LeftTee`[$expr_1$, $expr_2$]	$(e \dashv e) \dashv e$	
$expr_1 \dashv\!\dashv expr_2$	`DoubleLeftTee`[$expr_1$, $expr_2$]	$(e \dashv\!\dashv e) \dashv\!\dashv e$	
$expr_1 \ni expr_2$	`SuchThat`[$expr_1$, $expr_2$]	$e \ni (e \ni e)$	

expr..	Repeated[*expr*]		◁
expr...	RepeatedNull[*expr*]		◁
$expr_1$|$expr_2$	Alternatives[$expr_1$, $expr_2$]	e|e|e	◁
symb:*expr*	Pattern[*symb*, *expr*]		◁
patt:*expr*	Optional[*patt*, *expr*]		◁
$expr_1$~~$expr_2$~~$expr_3$	StringExpression[$expr_1$, $expr_2$, $expr_3$]	e~~e~~e	◁
$expr_1$/;$expr_2$	Condition[$expr_1$, $expr_2$]	(e/;e)/;e	◁
$expr_1$->$expr_2$	Rule[$expr_1$, $expr_2$]	e->(e->e)	◁
$expr_1$→$expr_2$	Rule[$expr_1$, $expr_2$]	e→(e→e)	◁
$expr_1$:>$expr_2$	RuleDelayed[$expr_1$, $expr_2$]	e:>(e:>e)	◁
$expr_1$:→ $expr_2$	RuleDelayed[$expr_1$, $expr_2$]	e:→(e:→e)	◁
$expr_1$/.$expr_2$	ReplaceAll[$expr_1$, $expr_2$]	(e/.e)/.e	◁
$expr_1$//.$expr_2$	ReplaceRepeated[$expr_1$, $expr_2$]	(e//. e)//. e	◁
$expr_1$+=$expr_2$	AddTo[$expr_1$, $expr_2$]	e+=(e+=e)	◁
$expr_1$-=$expr_2$	SubtractFrom[$expr_1$, $expr_2$]	e-=(e-=e)	◁
$expr_1$*=$expr_2$	TimesBy[$expr_1$, $expr_2$]	e*=(e*=e)	◁
$expr_1$/=$expr_2$	DivideBy[$expr_1$, $expr_2$]	e/=(e/=e)	◁
expr&	Function[*expr*]		◁
$expr_1$:expr$_2$	Colon[$expr_1$:$expr_2$]	e:e:e	
$expr_1$//$expr_2$	$expr_2$[$expr_1$]	(e//e)//e	
$expr_1$|$expr_2$	VerticalSeparator[	e|e|e	

expr..	Repeated[*expr*]		◁
expr...	RepeatedNull[*expr*]		◁
$expr_1$ | $expr_2$	Alternatives[$expr_1$, $expr_2$]	e|e|e	◁
symb:*expr*	Pattern[*symb*, *expr*]		◁
patt:*expr*	Optional[*patt*, *expr*]		◁
$expr_1$~~$expr_2$~~$expr_3$	StringExpression[$expr_1$, $expr_2$, $expr_3$]	e~~e~~e	◁
$expr_1$ /; $expr_2$	Condition[$expr_1$, $expr_2$]	(e /; e) /; e	◁
$expr_1$ -> $expr_2$	Rule[$expr_1$, $expr_2$]	e -> (e -> e)	◁
$expr_1$ → $expr_2$	Rule[$expr_1$, $expr_2$]	e → (e → e)	◁
$expr_1$:> $expr_2$	RuleDelayed[$expr_1$, $expr_2$]	e :> (e :> e)	◁
$expr_1$:→ $expr_2$	RuleDelayed[$expr_1$, $expr_2$]	e :→ (e :→ e)	◁
$expr_1$ /. $expr_2$	ReplaceAll[$expr_1$, $expr_2$]	(e /. e) /. e	◁
$expr_1$ //. $expr_2$	ReplaceRepeated[$expr_1$, $expr_2$]	(e //. e) //. e	◁
$expr_1$ += $expr_2$	AddTo[$expr_1$, $expr_2$]	e += (e += e)	◁
$expr_1$ -= $expr_2$	SubtractFrom[$expr_1$, $expr_2$]	e -= (e -= e)	◁
$expr_1$ *= $expr_2$	TimesBy[$expr_1$, $expr_2$]	e *= (e *= e)	◁
$expr_1$ /= $expr_2$	DivideBy[$expr_1$, $expr_2$]	e /= (e /= e)	◁
expr&	Function[*expr*]		◁
$expr_1$:expr$_2$	Colon[$expr_1$: $expr_2$]	e:e:e	
$expr_1$ // $expr_2$	$expr_2$[$expr_1$]	(e // e) // e	
$expr_1$ | $expr_2$	VerticalSeparator[	e|e|e	

$expr_1 \therefore expr_2$	`Therefore[`$expr_1$`,`$expr_2$`]`	$e \therefore (e \therefore e)$	
$expr_1 \because expr_2$	`Because[`$expr_1$`,`$expr_2$`]`	$(e \because e) \because e$	
$expr_1 = expr_2$	`Set[`$expr_1$`,`$expr_2$`]`	$e = (e = e)$	◁
$expr_1 := expr_2$	`SetDelayed[`$expr_1$`,`$expr_2$`]`	$e := (e := e)$	◁
$expr_1 \wedge = expr_2$	`UpSet[`$expr_1$`,`$expr_2$`]`	$e \wedge = (e \wedge = e)$	◁
$expr_1 \wedge := expr_2$	`UpSetDelayed[`$expr_1$`,`$expr_2$`]`	$e \wedge := (e \wedge := e)$	◁
$symb$`/:` $expr_1 = expr_2$	`TagSet[`$symb$`,`$expr_1$`,`$expr_2$`]`		◁
$symb$`/:` $expr_1 := expr_2$	`TagSetDelayed[`$symb$`,`$expr_1$`,`$expr_2$`]`		◁
$expr$`=.`	`Unset[`$expr$`]`		◁
$symb$`/:`$expr$`=.`	`TagUnset[`$symb$`,`$expr$`]`		◁
$expr_1 \mapsto expr_2$	`Function[{`$expr_1$`},`$expr_2$`]`	$e \mapsto (e \mapsto e)$	◁
$expr$`>>` $filename$	`Put[`$expr$`,"`$filename$`"]`		◁
$expr$`>>>` $filename$	`PutAppend[`$expr$`,"`$filename$`"]`		◁
$expr_1$`;` $expr_2$`;` $expr_3$	`CompoundExpression[`$expr_1$`,`$expr_2$`,`$expr_3$`]`		◁
$expr_1$`;`$expr_2$`;`	`CompoundExpression[`$expr_1$`,`$expr_2$`,Null]`		◁
$expr_1$`\``$expr_2$	`FormBox[`$expr_2$`,`$expr_1$`]`	e`\``$(e$`\``$e)$	◁

运算符输入形式，按递减优先排序.

特殊输入形式	完整形式
⌗	`Slot[1]`

##	SlotSequence[1]
## *n*	SlotSequence[*n*]
%	Out[ ]
%%	Out[-2]
%% … % (*n* 倍)	Out[-*n*]
% *n*	Out[*n*]
_	Blank[ ]
_*expr*	Blank[*expr*]
__	BlankSequence[ ]
__*expr*	BlankSequence[*expr*]
___	BlankNullSequence[ ]
___*expr*	BlankNullSequence[*expr*]
_.	Optional[Blank[ ]]
*symb*_	Pattern[*symb*,Blank[ ]]
*symb*_*expr*	Pattern[*symb*,Blank[*expr*]]
*symb*__	Pattern[*symb*, BlankSequence[ ]]
*symb*__*expr*	Pattern[*symb*, BlankSequence[*expr*]]
*symb*___	Pattern[*symb*, BlankNullSequence[ ]]
*symb*___*expr*	Pattern[*symb*, BlankNullSequence[*expr*]]
*symb*_.	Optional[Pattern[*symb*,Blank[ ]]]

额外输入形式，按递减优先排序.

特殊字符

运算符中特殊字符通常具有与所代表函数的名称相对应的名称. 因此字符 ⊕ 具有名称 \[CirclePlus]，并得到函数 CirclePlus. 例外的是 \[GreaterSlantEqual]、\[LessSlantEqual] 和 \[RoundImplies].

固定匹配运算符中的分隔符具有名称 \[Left*Name*] 和 \[Right*Name*].

"字符名称列表" 一节给出出现在运算符中特殊字符的一个完整列表.

键盘字符	特殊字符	键盘字符	特殊字符
->	\[Rule] →	>=	\[GreaterEqual
:>	\[RuleDelayed] :→	>=	\[GreaterSlant
==	\[Equal] ==	<=	\[LessEqual] ≤
!=	\[NotEqual] ≠	<=	\[LessSlantEqu

具有相同解释的键盘字符与特殊字符.

键盘字符	特殊字符	
\[RawColon] :	\[Colon] :	\[Raw
\[RawTilde] ~	\[Tilde] ~	\[RawVertical
\[RawWedge] ^	\[Wedge] ∧	\[RawVertical
\[RawWedge] ^	\[And] ∧	\[RawVertical
\[RawStar] *	\[Star] *	\[RawD
\[RawBackslash] \	\[Backslash] \	

具有不同解释的一些键盘字符与特殊字符 .

特殊字符

运算符中特殊字符通常具有与所代表函数的名称相对应的名称. 因此字符 ⊕ 具有名称 \[CirclePlus]，并得到函数 CirclePlus. 例外的是 \[GreaterSlantEqual]、\[LessSlantEqual] 和 \[RoundImplies].

固定匹配运算符中的分隔符具有名称 \[Left*Name*] 和 \[Right*Name*].

"字符名称列表" 一节给出出现在运算符中特殊字符的一个完整列表.

特殊字符		键盘字符	特殊字符
\[Rule]	→	>=	\[GreaterEqual] ≥
\[RuleDelayed]	:→	>=	\[GreaterSlantEqual] ⩾
\[Equal]	==	<=	\[LessEqual] ≤
\[NotEqual]	≠	<=	\[LessSlantEqual] ⩽

具有相同解释的键盘字符与特殊字符.

		键盘字符	特殊字符
:		\[RawDot] .	\[CenterDot] ·
~	\[RawVerticalBar]	|	\[VerticalBar] ∣
^	\[RawVerticalBar]	|	\[VerticalSeparator] ❘
	\[RawVerticalBar]	|	\[LeftBracketingBar] ⎸
	\[RawDash]	-	\[Dash] –
ash] \		...	\[Ellipsis] …

具有不同解释的一些键盘字符与特殊字符 .

输入形式的优先级和排序

输入形式表按优先级降序来组织. 同一盒框中的输入形式具有相同的优先级. 表中每一页以新盒框开始. 如 "表达式输入的特殊方式" 一节中所讨论的，优先级确定在输入表达式中 ***Mathematica*** 的分组方式. 一般的规则是，如果 ⊗ 具有比 ⊕ 高的优先级，则 $a \oplus b \otimes c$ 可以被解释为 $a \oplus (b \otimes c)$，$a \otimes b \oplus c$ 可以被解释为 $(a \otimes b) \oplus c$.

输入形式的分组

表中的第三列显示单个输入形式或具有相同优先级的几个输入形式多处出现时的分组方式. 例如，a / b / c 被分组为 (a / b) / c ("左相关")，而 a^b^c 被分组为 a^(b^c) ("右相关"). 在象 a + b + c 一样的表达式中不需要分组，因为 Plus 是完全相关的，这由属性 Flat 表示.

积分算子的优先级

诸如 $\int \text{expr}_1 \, d \, \text{expr}_2$ 之类的形式具有仅低于 Power 的"外"优先级，这正如上表所示，但"内"优先级仅仅高出 ∑. 外优先级确定 expr_2 何时需要被加上括号；内优先级确定 expr_1 何时需要被加上括号.

\[ContourIntegral]、\[ClockwiseContourIntegral] 和 \[DoubleContourIntegral] 与 \[Integral] 具有相同功能.

对与积分运算符相关的二维输入形式的讨论，参见"二维输入形式"一

空格与乘法

Mathematica 中的空格代表乘法，这与它们在标准的数学记号中所起的作用一样. 此外，***Mathematica*** 将相邻且不由空格分离的完整表达式一起相乘.

- `x y z ⟶ x*y*z`
- `2x ⟶ 2*x`
- `2(x+1) ⟶ 2*(x+1)`
- `c(x+1) ⟶ c*(x+1)`
- `(x+1)(y+2) ⟶ (x+1)*(y+2)`
- `x! y ⟶ x!*y`
- `x!y ⟶ x!*y`

乘法的替代形式.

一个表达式可能有多个含义，如 `x ! y` 既可能是 `(x !) * y`，也可能是 `x * (! y)`. 选择前一个解释的原因是 `Factorial` 比 `Not` 的优先级更高.

单个输入形式内的空格将被忽略. 因而，举例来说，`a + b` 等价于 `a + b`. 往往需要在低优先级运算符周围添加空格以增加可读性.

也可以为符号给出一个“系数”. 它优于任何数字序列.（当使用非**10**进制时，在系数的末尾和符号名称的开头必须留有空格.)

- `x^2 y`，同 `x^2 y`，表示 `(x^2) y`
- `x / 2 y`，同 `x / 2 y`，表示 `(x / 2) y`
- `xy` 是一个符号，而不是 `x * y`

需要注意的情况

需要避免的空格

在一个复合算子如 /.、=. 和 >= 中，不同字符之间应注意避免插入任何空格. 尽管在某些情况下，这样的空格是允许的，但这些空格很容易产生混淆.

另一种必须避免插入空格的情形是在模式对象 x_ 的字符之间. 如果输入 x _，则 ***Mathematica*** 将把它解释为 x * _， 而不是单一命名的模式对象 x_.

类似的，也不应在象 *x*_: *value* 的模式对象内部插入任何空格.

分隔符

- 普通键盘空格 (\[RawSpace])
- \[VeryThinSpace], \[ThinSpace], …, \[ThickSpace]
- \[NegativeVeryThinSpace], \[NegativeThinSpace], …, \[NegativeThickSpace]
- ␣ (\[SpaceIndicator])

分隔符等价于一个普通键盘空格.

关系运算符

相关运算符可以被混合使用. 如 a > b >= c 的一个表达式可以转换为Inequality[a, Greater, b, GreaterEqual, c]，它有效实现计算 (a > b) && (b >= c). (使用中间 Inequality 形式的原因是为了避免在类似 a > b >= c 的输入被处理时对象需要

文件名

在引用任何文件名的时候，都应该放到符号 <<、>> 和 >>> 后. 如果文件名只包含数字字符、特殊字符和 `、/、.、\、!、-、_、:、$、*、~ 和 ?，文件名也可不需要引号. 如果在配对的方括号内没有空格、制表符或换行符，文件名也可不需要引号. 但注意没有用引号给出的文件名后只能是空格、制表符、换行符或字符)、]、}，以及分号和逗号.

相关指南

- 语法

相关教程

- 输入语法

教程专集

- **Core Language**

常用记号和表示惯例

- 函数名
- 函数变量
- 选项
- 元素编号
- 序列规格
- 层规格
- 迭代器
- 作用域结构
- 表达式的顺序
- 数学函数
- 数学常数
- 保护
- 缩写的字符串模式

函数名

内置函数的名称遵循一些规则指南.

- 名称包括完整的英文单词，或者数学简写. 这里使用的是美式拼写法.
- 每个单词的首字母大写.
- 名称以 `Q` 结尾的函数通常表示“提问”，返回值或者是 `True` 或者是 `False`.
- 以人名命名的数学函数在 ***Mathematica*** 中的函数名形式为 *PersonSymbol*.

函数变量

内置函数所作用的主表达式或对象经常作为函数的第一个变量. 辅助参数作为接下来的变量出现.

下面是一些例外情况：

常用记号和表示惯例

- 函数名
- 函数变量
- 选项
- 元素编号
- 序列规格
- 层规格
- 迭代器
- 作用域结构
- 表达式的顺序
- 数学函数
- 数学常数
- 保护
- 缩写的字符串模式

函数名

内置函数的名称遵循一些规则指南.

- 名称包括完整的英文单词，或者数学简写. 这里使用的是美式拼写法.
- 每个单词的首字母大写.
- 名称以 `Q` 结尾的函数通常表示“提问”，返回值或者是 `True` 或者是 `False`.
- 以人名命名的数学函数在 ***Mathematica*** 中的函数名形式为 *PersonSymbol*.

函数变量

内置函数所作用的主表达式或对象经常作为函数的第一个变量. 辅助参数作为接下来的变量出现.

下面是一些例外情况：

- 在函数如 `Map` 和 `Apply` 中，所要应用的函数在它应用的表达式之前出现.
- 在作用域结构如 `Module` 和 `Function` 中，局部变量和参数名在函数体之前出现.
- 在函数如 `Write` 和 `Export` 中，文件名在写入的对象之前给出.

对于数学函数，在标准数学记号中作为下标的变量在上标之前给出.

选项

一些内置函数可以采用选项（***options***）. 每个选项具有一个名称，可以表示为一个符号，或者在一些情况下可以表示为一个字符串. 通过给出形如 *name* -> *value* 或者 *name* :> *value* 的规则，设置选项. 这样的规则必须出现在函数的其它变量之后. 可以按任何顺序给出不同选项的规则. 如果用户不能对特定选项明确地给出规则，则使用该选项的默认设置.

`Options[`*f*`]`	给出所有与 *f* 相关联的选项的默认值
`Options[`*expr*`]`	给出在特定表达式中设置的选项
`Options[`*expr*`,`*name*`]`	给出表达式中选项 *name* 的设置
`AbsoluteOptions[`*expr*`,`*name*`]`	给出 *name* 的绝对设置，即使实际设置为 `Automatic`
`SetOptions[`*f*`,`*name*`->`*value*`,...]`	对与 *f* 相关联的选项设置默认规则
`CurrentValue[`*name*`]`	给出前端选项 *name* 的选项设置；可以在赋值操作的左端使用来设置选项

选项上的操作.

元素编号

n	元素 n（从 1 开始）
$-n$	从尾部开始的元素 n
0	头部

元素编号.

序列规格

All	所有元素
None	没有元素
n	元素 1 到 n
$-n$	最后 n 个元素
$\{n\}$	只包含元素 n
$\{m,n\}$	元素 m 到 n（闭区间）
$\{m,n,s\}$	元素 m 到 n，步长为 s

子序列.

序列规格 $\{m, n, s\}$ 对应于元素 m、$m+s$、$m+2s$、…，直至不大于 n 的最大元素.

序列规格在函数 Drop、Ordering、StringDrop、StringTake、Take 和 Thread 中使用.

层规格

n	层 1 到 n
Infinity	层 1 到 Infinity
$\{n\}$	只包含层 n
$\{n_1, n_2\}$	层 n_1 到 n_2
Heads->True	包含表达式的头部
Heads->False	不包含表达式的头部

层规格.

在表达式中，对应于非负整数 n 的层被定义为由 n 个索引指定的部分组成. 一个负的层数 $-n$ 表示深度为 n 的表达式的所有部分. 表达式的深度，Depth[*expr*]，是用来指明任意部分的索引的最大值，再加**1**. 除非设置选项 Heads -> True，否则层的计数不包括表达式的头部. 层 0 是整个表达式. 层 -1 包含所有没有子部分的符号和其它对象.

由 $\{n_1, n_2\}$ 指定的层的范围包括所有树中的既不在层 n_1 之上，也不在层 n_2 之下的部分. n_i 不必具有相同的符号. 因此，例如，{2, -2} 指明了表达式树中出现在最高层以下，但叶子层以上的子表达式.

函数如 Apply、Cases、Count、FreeQ、Level、Map、MapIndexed、Position、Replace 和 Scan 使用层规格. 然而，注意，对于所有这些函数，默认层规格并不相同.

迭代器

$\{i_{max}\}$	迭代 i_{max} 次
$\{i, i_{max}\}$	i 从 **1** 到 i_{max}，步长为**1**
$\{i, i_{min}, i_{max}\}$	i 从 i_{min} 到 i_{max}，步长为**1**
$\{i, i_{min}, i_{max}, di\}$	i 从 i_{min} 到 i_{max}，步长为 di
$\{i, list\}$	i 采取 $list$ 中的连续值
$\{i, i_{min}, i_{max}\}$, $\{j, j_{min}, j_{max}\}, \cdots$	i 从 i_{min} 到 i_{max}，并且对每个 i，j 从 j_{min} 到 j_{max}，等等.

迭代器表示法.

迭代器在函数如 Sum、Table、Do 和 Range 中使用.

迭代参数 i_{min}，i_{max} 和 di 不必是整数. 变量 i 被赋于一系列值，这些值从 i_{min} 开始，以步长 di 递增，当 i 的下一个值比 i_{max} 大时停止. 迭代参数可以是任意的符号表达式，只要 $(i_{max} - i_{min}) / di$ 是一个数.

当使用一些迭代变量时，后面的迭代变量的极限可以依赖于前面的迭代变量的值.

变量 i 可以是任意符号表达式；它不必是单个字符. i 的值自动设为迭代函数的局部值. 这可以通过对迭代函数中包含 i 的 Block 结构进行封装来实现.

关于计算迭代函数的过程，请参见 "计算".

作用域结构

`Function[{x,...},body]`	局部参数
lhs`:>`*rhs* 和 *lhs*`:>`*rhs*	局部模式名
lhs`=`*rhs* 和 *lhs*`:=`*rhs*	局部模式名
`With[{x=`x_0`,...},body]`	局部常数
`Module[{x,...},body]`	局部变量
`Block[{x,...},body]`	全局变量的局部值
`DynamicModule[{x,...},body]`	在 `Dynamic` 接口中的局部变量

Mathematica 中的作用域结构. 语法上第一组作用域变量上的函数.

作用域结构允许特定符号的名称或值是局部的.

一些作用域在语法范围内缩小作用域，这意味着特定变量或模式的字母实例被合适的值替代. 当要求局部变量名时，名称形如 *xxx* 的符号被重命名为 *xxx*$. 当计算嵌套作用域结构时，在内层作用域结构中自动产生新的符号，以避免与外层作用域结构的符号冲突.

当采用一个变换规则或定义的时候，`ReplaceAll`（`/.`）实际上用来代替出现在右边的模式名. 然而，当必要的时候，将产生新符号用来表示在右边出现的作用域结构的其它对象.

每次计算的时候，`Module` 产生形如 *xxx*$*nnn* 具有唯一名称的符号，用来替代所有在函数体中出现的局部变量.

`Block` 局部化全局变量的值. 即使当变量没有在函数体内明确出现

的值. `Block` 体可能也会对全局变量进行修改，但是任何修改都只持续到 `Block` 结束执行的时候.

在一个笔记本中，`DynamicModule` 对 `DynamicModule` 输出的每个实例局部化它的变量. 这意味着每个使用复制和粘贴创建的 `DynamicModule` 输出的每个拷贝都将使用它自己的局部变量.

表达式的顺序

表达式的正则排序自动使用属性 `Orderless`，并且在函数如 `Sort` 中符合以下规则:

- 整数、有理数和近似实数根据它们的数值值排序.
- 复数根据它们的实部排序，在两个复数的实部相等的情况下，可以根据虚部的绝对值排序.
- 符号根据名称排序，在两个符号相等的情况下，可以根据上下文排序.
- 表达式经常是以深度优先方式比较它们的各部分来排序的. 较短的表达式排在前面.
- 幂和积使用特殊方法处理，它们是根据多项式中的项进行排序的.
- 字符串根据它们在字典中的顺序排序，其中大写字母排在小写字母之后.

普通字母排在最前面，接着依次是脚本（**script**）、哥特式（**Gothic**）、双式（**double-struck**）、希腊文和希伯来文. 数学操作符以优先级的递减顺序出现.

数学函数

Mathematica 内置的数学函数如 `Log[z]` 和 `Besselj[n, z]`

具有大量共同的特点.

- 它们具有属性 `Listable`，因此它们自动逐项作用于任何作为变量出现的列表.
- 它们具有属性 `NumericFunction`，因此当它们的变量是数值的时候，假定它们给出数值值.
- 在特殊情况下，它们给出精确的关于整数、有理数和代数表达式的结果.
- 除了变量总是整数的函数，在 ***Mathematica*** 中的数学函数计算可以达到任意数值精度，可以以任意复数作为变量. 如果一个函数对特定变量集合未定义，在这种情况下它以符号形式返回.
- 数值计算的结果精度不比在变量精度的基础上判断的结果高. 因此 `N[Gamma[27/10], 100]` 产生高精度的结果，但是 `N[Gamma[2.7], 100]` 不能.
- 在可能的情况下，可以关于其它内置函数，计算内置数学函数的符号式导数、积分和级数展开.

数学常数

Mathematica 内置的数学常数如 `E` 和 `Pi` 具有下面特性：

- 它们本身不具有值.
- 它们可表示为任意精度的数值值.
- 在 `NumericQ` 中和其它地方作为数值量处理.
- 它们具有属性 `Constant`，因此在导数中作为常数处理.

保护

Mathematica 允许用户进行赋值，来重载 ***Mathematica*** 内置对象的标准操作和含意.

为了尽量避免出现错误的赋值，大多数 ***Mathematica*** 内置对象具有属性 `Protected`. 如果用户想要对一个内置对象进行赋值，必须首先删除该属性. 可以调用函数 `Unprotect` 来实现该功能.

存在一些基本的 ***Mathematica*** 对象，用户绝对不可以自己对其赋值. 这些对象具有属性 `Locked`，以及`Protected`. `Locked` 属性防止用户对任何属性进行修改，并且也防止删除 `Protected` 属性.

缩写的字符串模式

函数如 `StringMatchQ`、`Names` 和 `Remove` 允许用户给出缩写的字符串模式（***abbreviated string patterns***），以及由 `StringExpression` 指定的完全字符串模式. 缩写的字符串模式可以包含某些元字符（**metacharacters**），元字符可以表示普通字符组成的序列.

*	零个或多个字符
@	除了大写字母外的，一个或多个字符
*，等等.	字母 *，等等

在缩写字符串模式中使用的元字符.

教程专集

- **Core Language**

相关的 Wolfram Training 课程

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

计算

- 标准计算序列
- 非标准变量计算
- 重载非标准变量计算
- 避免计算
- 计算的全局控制
- 退出

标准计算序列

下面是在计算例如 $h[e_1, e_2 \ldots]$ 的表达式的时候，***Mathematica*** 所采取的步骤. 每当表达式发生改变时，***Mathematica*** 重新开始计算的过程序列.

- 如果表达式是一个原始对象（如 `Integer`、`String` 等等），则不改变它.
- 计算表达式的头部 h.
- 依次计算表达式的每个元素 e_i. 如果 h 是具有属性 `HoldFirst`、`HoldRest`、`HoldAll` 或者 `HoldAllComplete` 的符号时，则跳过某些元素的计算过程.
- 除非 h 具有属性 `HoldAllComplete`，否则除去在 e_i 中出现的任何 `Unevaluated` 封装的最外层.
- 除非 h 具有属性 `SequenceHold`，否则压平 e_i 中出现的所有 `Sequence` 对象.

- ***Mathematica*: An Introduction**
- ***Mathematica*: Programming in *Mathematica***

计算

- 标准计算序列
- 非标准变量计算
- 重载非标准变量计算
- 避免计算
- 计算的全局控制
- 退出

标准计算序列

下面是在计算例如 $h[e_1, e_2 \ldots]$ 的表达式的时候，***Mathematica*** 所采取的步骤. 每当表达式发生改变时，***Mathematica*** 重新开始计算的过程序列.

- 如果表达式是一个原始对象（如 `Integer`、`String` 等等），则不改变它.
- 计算表达式的头部 h.
- 依次计算表达式的每个元素 e_i. 如果 h 是具有属性 `HoldFirst`、`HoldRest`、`HoldAll` 或者 `HoldAllComplete` 的符号时，则跳过某些元素的计算过程.
- 除非 h 具有属性 `HoldAllComplete`，否则除去在 e_i 中出现的任何 `Unevaluated` 封装的最外层.
- 除非 h 具有属性 `SequenceHold`，否则压平 e_i 中出现的所有 `Sequence` 对象.

- 如果 h 具有属性 `Flat`，那么压平所有具有头部 h 的嵌套表达式.
- 如果 h 具有属性 `Listable`，那么对任何列表 e_i 进行线性操作.
- 如果 h 具有属性 `Orderless`，则按顺序对 e_i 进行排列.
- 除非 h 具有属性 `HoldAllComplete`，否则使用与 f 相关联的适当变换规则，其中 f 是对形如 $h[f[e_1, \ldots], \ldots]$ 的对象定义的.
- 对形如 $h[f[e_1, \ldots], \ldots]$ 的对象采用与 f 相关联的内置变换规则.
- 采用对 $h[f[e_1, e_2, \ldots], \ldots]$ 或者 $h[\ldots][\ldots]$ 定义的适当的变换规则.
- 采用 $h[e_1, e_2, \ldots]$ 或者 $h[\ldots][\ldots]$ 的内置变换规则.

非标准变量计算

有大量 ***Mathematica*** 内置函数用特殊方式计算变量. 控制结构 `While` 就是一个例子. 符号 `While` 具有属性 `HoldAll`. 于是，`While` 的变量不作为标准计算过程的一部分进行计算. 相反地，`While` 的内部代码采用特殊方式计算变量. 在 `While` 的情况下，代码重复计算变量，以完成一个循环.

控制结构（**Control structures**）	在由控制流决定的序列中计算的变量（如 `CompoundExpression`）
条件（**Conditionals**）	只有当采用该变量对应的分支时，才计算的变量（如 `If`、`Which`）
逻辑运算（**Logical operations**）	只有当确定逻辑结果需要时，才计算的变量 （如 `And`、`Or`）
迭代函数（**Iteration functions**）	在迭代的每个步骤中计算的第一个变量（如 `Do`、`Sum`、`Plot`）
跟踪函数（**Tracing functions**）	从未计算的形式 （如 `Trace`）
赋值	第一个变量只是部分计算 （如 `Set`、`AddTo`）
纯函数	不计算函数体（如 `Function`）
作用域结构（**Scoping constructs**）	不计算变量规格（如 `Module`、`Block`）
保持函数（**Holding functions**）	变量保持不计算的形式（如 `Hold`、`HoldPattern`）

用特殊方式计算变量的内置函数.

逻辑操作

在形如 e_1 && e_2 && e_3 的表达式中，e_i 依次计算. 一旦发现任何 e_i 等于 `False`，计算就停止，并且返回结果 `False`. 这意味着用户可以使用 e_i 来表示程序中的不同“分支”，只有当满足某些条件的时候，才计算一个特定的分支.

`Or` 函数与 `And` 相似；一旦它找到值为 `True` 的变量，就返回 `True`. 另一方面，`Xor` 总是计算所有的变量.

迭代函数

迭代函数如 `Do[f, {i, imin, imax}]` 的计算过程如下：

- 计算 i_{min} 和 i_{max} 的极值.
- 通过使用 `Block`，把迭代变量 i 的值设为局部值.
- i_{min} 和 i_{max} 用来决定对迭代变量 i 所赋的值的序列.
- 迭代变量被连续设为每个值，并且在每个情况下计算 f.
- 清除赋给 i 的局部值.

如果有若干个迭代变量，对每个变量和所有之前的变量值依次使用相同的过程处理.

除非特别指明，否则直到对 i 赋以特定的值才计算 f，并且对每个选定的 i 值进行计算. 使用 `Evaluate[f]` 可以立即计算 f，而不用在一个特定的值被赋给 i 以后才计算.

赋值

赋值的左边只能被部分计算.

- 如果左边是一个符号，则不进行计算.
- 如果左边是一个不具有保持不计算属性的函数，则计算函数的变量，但不计算函数本身.

右边被立即计算（=），但是没有延时赋值（:=）.

出现在赋值左边的形如 `HoldPattern[expr]` 的任意子表达式没

HoldPattern，但是它仍然是匹配的.

重载非标准变量计算

$f[\ldots, \texttt{Evaluate}[expr], \ldots]$	计算变量 *expr*，不论 f 是否具有 HoldFirst、HoldRest 或者 HoldAll 等保持不计算的属性

重载变量的保持不计算功能.

通过使用 Evaluate，可以立即计算函数的任意变量，即使该变量通常在之后的函数的控制下计算.一个异常情况是当函数具有 HoldComplete 属性的时候；在这种情况下，函数的内容不会被修改.

避免计算

Mathematica 提供了各种不同的函数作为封装（“**wrappers**”）来避免计算它们所包含的表达式.

`Hold`[*expr*]	在所有情况下作为 `Hold`[*expr*] 处理
`HoldComplete`[*expr*]	禁用上值（**upvalue**），并作为 `HoldComplete`[*expr*] 处理
`HoldForm`[*expr*]	显示时作为 *expr* 处理
`HoldPattern`[*expr*]	在规则、定义和模式中，作为 *expr* 处理
`Unevaluated`[*expr*]	当变量传递给函数时，作为 *expr* 处理

避免表达式进行计算的封装（**Wrapper**）.

计算的全局控制

在我们目前讨论过的计算过程中，涉及了两种基本步骤：

- 迭代：计算一个特定的表达式直到其不再改变.
- 递归：计算所需的辅助表达式，以找到特定表达式的值.

迭代产生了这样的计算链：其中通过使用各种不同的变换规则，得到连续的表达式.

`Trace` 把计算链显示为列表，并且显示对应于子列表中的递归的辅助计算.

与辅助计算序列相关联的表达式由 `Stack`[] 返回的列表给出，这些辅助计算产生当前正在计算的表达式.

$RecursionLimit	最大迭代深度
$IterationLimit	最大迭代次数

控制表达式计算的全局变量.

退出

通过调用函数 Abort[]，或者通过输入合适的中断键，用户可以让 ***Mathematica*** 在计算中的任意点退出.

当被要求退出时，***Mathematica*** 将尽快终止计算. 如果获得的答案是不正确的或者不完整的，那么 ***Mathematica*** 返回的不是答案而是 $Aborted.

可以使用 CheckAbort 来捕获退出，并且使用 AbortProtect 获得延迟.

相关指南

- 计算控制

教程专集

- **Core Language**

模式和变换规则

- 模式
- 赋值
- 值的类型
- 清除和删除对象
- 变换规则

模式

模式（***Patterns***）表示表达式的不同类别. 它们包含*模式对象*，这些模式对象表示可能的表达式集合.

_	任意表达式
*x*_	名称为 *x* 的任意表达式
x:*pattern*	名称为 *x* 的模式
pattern?*test*	当使用 *test* 时，产生 `True` 的模式
_ *h*	头部为 *h* 的任意表达式
*x*_ *h*	头部为 *h*、名称为 *x* 的任意表达式
__	一个或多个表达式序列
___	零个或多个表达式序列
*x*__ 和 *x*___	名称为 *x* 的表达式序列
__ *h* 和 ___ *h*	由每个头部为 *h* 的表达式组成的序列
*x*__ *h* 和 *x*___ *h*	头部为 *h*、名称为 *x* 的表达式序列
`PatternSequence`[*p*1, *p*2, …]	模式序列
*x*_:*v*	默认值为 *v* 的表达式
*x*_ *h*:*v*	头部为 *h* 并且默认值为 *v* 的表达式

模式和变换规则

- 模式
- 赋值
- 值的类型
- 清除和删除对象
- 变换规则

模式

模式（***Patterns***）表示表达式的不同类别. 它们包含*模式对象*，这些模式对象表示可能的表达式集合.

_	任意表达式
x_	名称为 x 的任意表达式
x:pattern	名称为 x 的模式
pattern?test	当使用 test 时，产生 True 的模式
_h	头部为 h 的任意表达式
x_h	头部为 h、名称为 x 的任意表达式
__	一个或多个表达式序列
___	零个或多个表达式序列
x__ 和 x___	名称为 x 的表达式序列
__h 和 ___h	由每个头部为 h 的表达式组成的序列
x__h 和 x___h	头部为 h、名称为 x 的表达式序列
PatternSequence[p1, p2, …]	模式序列
x_:v	默认值为 v 的表达式
x_h:v	头部为 h 并且默认值为 v 的表达式

`Optional[`*x* `_` *h*`]`	具有头部 *h* 以及全局定义的默认值的表达式
`Except[`*c*`]`	除了与 *c* 匹配的任意表达式
`Except[`*c*`,` *pattern*`]`	与 *pattern* 匹配，但不与 *c* 匹配的任意表达式
pattern`..`	重复一次或多次的模式
pattern`...`	重复零次或多次的模式
`Repeated[` *pattern*`,` *spec*`]`	根据 *spec* 重复的模式
$pattern_1 \mid pattern_2 \mid \ldots$	与 $pattern_i$ 中的至少一个匹配的模式
pattern`/;`*cond*	*cond* 等于 `True` 的模式
`HoldPattern[`*pattern*`]`	未被计算的模式
`Verbatim[`*expr*`]`	逐字匹配的表达式
`OptionsPattern[]`	选项序列
`Longest[`*pattern*`]`	与 *pattern* 一致的最长序列
`Shortest[`*pattern*`]`	与 *pattern* 一致的最短序列

`Pattern` 对象.

当一些具有相同名称的模式对象在单个模式中出现时，所有的对象必须表示相同的表达式. 因此 `f[x_, x_]` 可以表示`f[2, 2]` 但不可以表示 `f[2, 3]`.

在一个模式对象比如 *_h* 中，头部 *h* 可以是任意表达式，但本身不可以是模式.

一个模式对象比如 *x__* 表示一个表达式*序列*. 所以，例如，`f[x__]` 可以表示 `f[a, b, c]`，其中 `x` 为 `Sequence[a, b, c]`. 如果用户使用 `x`，比如在一个变换规则的结果中，那么该序列将与 `x` 所在的函数相拼接. 因

此，g[u, x, u] 将变成 g[u, a, b, c, u].

当模式对象比如 x_: v 和 x_. 作为函数变量出现时，它们表示可能忽略的变量. 当忽略与 x_: v 相对应的变量时，x 将采用值 v. 当忽略与 x_. 相对应的变量时，x 将采用与其所在的函数相关联的*默认值*. 用户可以通过对 Default[f] 等进行赋值指定这些默认值.

Default[f]	当 x_. 作为函数 f 的任意变量出现时，它的默认值
Default[f, n]	当 x_. 作为第 n 个变量 （从尾端算起的第负 n 个数）时，它的默认值
Default[f, n, tot]	当我们一共有 tot 个变量时，第 n 个变量的默认值

Default 值.

模式如 f[x__, y__, z__] 可以匹配具有 x、y 和 z 的不同可能值的表达式，比如 f[a, b, c, d, e]. 首先将尝试 x 和 y 的具有最小长度的选择. 一般说来，当单个函数中有多个 __ 或者 ___ 时，首先尝试的情况是让所有 __ 和 ___ 表示具有最小长度的序列，除了最后一个模式表示“剩余”的变量.

当出现 x_: v 或者 x_. 时，首先尝试的是不与被忽略变量相对应的情况. 接下来尝试的是去掉后来的变量的情况.

可以使用 Shortest 和 Longest 来改变不同情况中的顺序.

`Orderless`	$f[x, y]$ 和 $f[y, x]$ 是等价的
`Flat`	$f[f[x], y]$ 和 $f[x, y]$ 是等价的
`OneIdentity`	$f[x]$ 和 x 是等价的

在模式匹配中使用的属性.

模式对象比如 $x_$ 可以表示具有属性 `Flat` 的函数 f 中的任意变量序列. 在这种情况下，x 的值是应用于变量序列的 f. 如果 f 具有属性 `OneIdentity`，当 x 对应于仅有一个变量的序列时，则使用 e 而不是 $f[e]$.

赋值

lhs=*rhs*	立即赋值：赋值时计算 *rhs*
lhs:=*rhs*	延时赋值：每次需要 *lhs* 值时，计算 *rhs*

Mathematica 中的两种基本赋值类型.

Mathematica 中的赋值指定表达式的变换规则. 用户进行的每次赋值必须与特定的 ***Mathematica*** 符号相关联.

$f[args]=rhs$	与 f 相关联的赋值（下值 **downvalue**）
$t/:f[args]=rhs$	与 t 相关联的赋值（上值 **upvalue**）
$f[g[args]]\wedge=rhs$	与 g 相关联的赋值（上值 **upvalue**）

与不同符号相关联的赋值.

在赋值如 $f[args] = rhs$ 中，***Mathematica*** 查看 f，然后是 f 的头部，然后是头部的头部，以此类推，直到它找到与赋值相关联的符号.

当采用赋值如 *lhs* ^= *rhs* 时，***Mathematica*** 将建立与每个以 *lhs* 的参数，或者以 *lhs* 参数的头部出现的不同符号相关联的变换规则.

与特定符号 *s* 相关联的变换规则总是以特定的顺序存储，并且利用它们所采用的顺序进行测试. 每次进行赋值的时候，相应的变换规则就在与 *s* 相关联的变换规则列表末尾插入，除了以下情况：

- 变换规则的左端与已经存储的变换规则相同，并且右端的任意 /; 条件也相同. 在这种情况下，将把新的变换规则在旧的变换规则的位置上插入.
- ***Mathematica*** 确定新的变换规则比已经存在的变换规则更为具体，并且如果它放置于该规则之后，将不会被使用. 在这种情况下，新规则放置于旧规则之前. 注意，在许多情况下，不可能判断一个规则是否比另一个规则更为具体；在这样的情况下，新规则总是在末尾插入.