MiniScript 手册

学习阅读、编写与使用
世界上最容易的计算机语言

Joe Strout

MiniScript 制作者

由 WaterRun 翻译，使用 GPT-5.2-CodeX

仓库：https://github.com/Water-Run/llm-translate-documents
原文：https://miniscript.org/files/MiniScript-Manual.pdf

版本 1.6.2

2024 年 7 月 22 日 星期一

欢迎使用 MiniScript

一种高级、面向对象的语言
易于阅读与书写

MiniScript 是一门现代脚本语言，旨在保持整洁、简单且易于学习。它从零开始设计，只借鉴其他语言中最好的理念，比如 Python、Lua、Basic 和 C#。如果你几乎了解任何其他编程语言，你会几乎立刻上手 MiniScript。

如果你一生中从未写过一行代码，也别慌！MiniScript 是最友好、最有趣的入门方式。它比你可能想象的要容易得多。

重要：MiniScript 被设计为一门*嵌入式*编程语言。这意味着你通常会在其他程序中使用它，比如一款电子游戏。你应该寻找另一份文档，说明如何在那个其他程序中访问和使用 MiniScript。本文件仅描述 MiniScript 语言本身，以及在多数 MiniScript 应用中通用的内置函数。

简洁、清晰的语法

让我们直接看一个例子，看看 MiniScript 代码是什么样子的。

```
s = “Spam”
while s.len < 50
s = s + “, spam”
end while
print s + “ and spam!”
```

MiniScript 中每条语句通常独占一行。注意行末没有分号、花括号或其他标记。

不过有一个例外：如果你想把多条语句放在同一行上，仅为了让代码更紧凑，可以用分号来分隔语句。以下代码很丑，但合法。

```
s = “Spam”; while s.len < 50; s = s + “, spam”; end while

print s + “ and spam!”
```

在实践中这个特性很少使用，但需要时它就在那里。

代码块

如果你习惯了 C 系语言（如 C、C++、C# 等），那么你已经习惯看到
用花括号包围代码块。MiniScript 并不是这样；代码块总是
以一个关键字开始（ `if`、`for`、`while` 或 `function` ），并以匹配的 end
语句结束（`end if`、`end for`、`end while` 或 `end function` ）。

空白与缩进

你几乎可以在代码中任何想要的地方插入空格和制表符。你不能
把标识符或关键字拆开（ `pr int` 不等同于 `print` ），也不能省略两个标识符或关键
`end if` 是正确的，但 `endif` 将无法工作）。当然，引号内的空格
在数字、运算符等之间，你可以随意加入额外空格。以下两行
在 MiniScript 看来完全一样。

```
x=4*10+2
x = 4 * 10 + 2
```

为了让代码结构更易读，传统做法是将代码块内的行缩进一个制表符或两个空格。
但这并非必需。MiniScript 不关心你是否以及如何缩进代码，所以按你觉得最合适的方式来。

长行拆分

不同于 C 系语言，每行末尾没有分号或其它奇怪的标点来让计算机知道语句结束。
相反，单靠换行就足以表示这一点。但如果你需要输入一条超过一行的语句呢？

如果最后一个记号（在任何注释之前——见下文）是左圆括号、左方括号或左花括号；或逗号，或任何二
（如

`+`、`*`，等等），MiniScript 就会识别该语句尚未完成

```
speech = [“Four score and seven years ago our fathers”,
“brought forth on this continent, a new nation, conceived”,
“in Liberty, and dedicated to the proposition that all”,
“men are created equal.”]
```

这有四行，但就 MiniScript 而言只是一个语句。这是因为前面三行每行末尾都有逗号，告诉 MiniScrip
内容。

注释

注释是你留给自己或阅读你代码的其他人类的小便笺。它们会被 MiniScript 完全忽略。注释以两个斜杠开始，并延伸到行末。因此你可以把注释放在单独一行，或放在一条语句之后。

```
x = 6*7
```

就像缩进一样，注释从不强制需要……但它们大概是个好主意！

括号的使用

MiniScript 里的括号只有三种用途：

1. 用来按你想要的顺序对数学运算分组，就像代数中那样。

```
x = (2+4)*7
```

2. 在函数调用的参数周围使用它们，除非函数调用是整个语句。

```
print cos(0)
```

3. 在声明带参数的函数时使用它们（见函数章节）。

由于其他语言常常在许多其他地方要求括号，值得指出在 MiniScript 中哪些地方不 使用括号。首先，不要在条件周围加括号 `if` 或 `while` 语句（后面会详述）。其次，在调用没有任何参数的函数时，不需要（并且应
函数 `time` 会获取程序开始以来的秒数。它不需要任何参数，所以你可以不加括号地调用它
它不需要任何参数，所以你可以不加括号地调用它。

```
x = time
```

最后，如上所述，当一个函数是语句中的第一个内容时，你不需要在它的参数周围加括号。下面的例子打字，每次等待一秒，然后打印一条消息。请注意我们在调用 `print` 和 `wait` 时没有任何括号。但 `range` 的调用，因为它有参数并且作为较大语句的一部分使用，作为较大语句的一部分使用，就需要括号。

```
for i in range(10, 1)
print i
wait
end for
print “砰！”
```

局部变量与全局变量

A 变量 是一个与某个值关联的单词（也称为标识符）。把变量想象成
可以存放数据的小盒子。你只需给它赋值就能创建一个变量，
就像我们已经见过的许多示例一样。

```
x = 42
```

这一行创建了一个名为 x的变量（如果它之前不存在），并在其中存入 42。它会
替换 x的先前值（如果有的话）。

MiniScript 中的变量是 动态类型 ；也就是说，你可以把任何类型的数据（见
“数据类型”一章）赋给任何变量。

变量的作用域始终是 局部 的。也就是说，一个函数内部名为“x”的变量与另一个函数中
名为“x”的变量毫无关系；每个变量的作用域（限制）都
限定在赋值时正在执行的当前函数内。

不过，MiniScript 也支持任何函数之外的代码，就像我们到目前为止
看到的所有示例一样。在这种上下文中，局部变量与全局变量是相同的。也就是说，
在函数之外给 x赋值 42 会创建一个名为 x的全局变量。这样的全局
变量可在任何上下文中访问。

注意，当某个上下文中有一个与全局变量同名的局部变量时，标识符
总会优先解析为局部变量。同样，函数内的简单赋值语句总会创建一个局部变量，
而不是全局变量。若你确实需要访问全局变量，还有一个
globals 对象可以提供这种
访问。（关于 globals 的更多细节，见“内置函数”一

```
demo = function
print x
x = 2
print x
print globals.x
globals.x = 42
print x
print globals.x
end function

x = 40
demo
```

过度使用全局变量有时会导致棘手的 bug，因此最好谨慎使用，尽可能依赖局部变量。MiniScript 的设计
良好实践自然发生。

数学赋值运算符

作为一种方便的简写，数学运算符（+、-、*、/、%、以及^）可以用于数学赋值形式。这会对变量执行一次数学运算，并将结果再赋值回该变量。例如，数学赋值形式：

```
x += 1
```

与下面的写法完全等价：

```
x = x + 1
```

之前关于局部变量和全局变量的规则仍然适用。因此，要用数学赋值形式更新一个全局变量，你可以这样写：

```
globals.x *= 5
```

这不仅适用于数字，也适用于任何定义了所用运算符的数据类型。例如，如果上面示例中的全局变量x在执行前是“ha”，那么执行该行之后，x的值会变为“hahahahaha”。

MiniScript 区分大小写

在 MiniScript 中，大写和小写是有区别的。内置函数print必须输入为完全写成print，而不是 Print，PRINT，或其它任何变体。同样的规则也适用于你定义的任何变量、函数或类。

至于你在自定义标识符中如何使用大小写由你决定，不过常见的约定是类名首字母大写（例如 Shape），但变量使用小写。因此下面将会是一个完全合理的代码片段。

```
shape = new Shape
```

顺便说一句，关于命名约定，大多数情况下你应避免让任何全局变量或函数名以下划线开头。以下划线开头的标识符常被宿主环境用于特殊的“底层”代码，若发生命名冲突可能会带来问题。

控制流程

循环与分支

控制流程 是让代码多次执行，或仅在特定条件下执行的方式。没有它，你的脚本将只能从第一行开始，按顺序把每一行恰好执行一次，并在最后一行之后结束。

MiniScript 包含一种分支（条件）结构，以及两种循环。

使用 if 进行分支

使用 if…then 语句来指定在什么条件下应执行后续语句。基本语法是：

```
if 条件 then
…
end if
```

当条件不成立时，MiniScript 会直接跳到 end if 语句。

```
if x == 42 then
print “我已经找到了终极答案！”
end if
```

从 if…then 到 end if 的整组行，称为 if 块 。

有时当指定条件不成立时你想做点别的。你可以用 else 块 来指定，放在 end if 之前。

```
if x == 42 then
print “我已经找到了终极答案！”
else
print “我仍在寻找。”
end if
```

最后，你可以按需添加 else-if 块 来检查额外条件。这里有一个稍微更实用的例子，把数字转换成文字。

```
if apples == 0 then
print “你没有苹果。”
else if apples == 1 then
print “你有一个苹果。”
else if apples > 10 then
print “你有很多苹果！”
else
print “你有 “ + apples + “ 个苹果。”
end if
```

在这种情况下，第一个匹配的条件将执行它的代码块。如果没有任何一个
条件匹配，那么 else 代码块将会执行。

注意对于所有这些形式，if、else if 、else，以及 end if 语句都必须各占一行。
不过，也有一种“短形式”的 if 语句，允许你把
一个 if 或 if/else 写在一行里，前提是 then
块只有一条语句，并且 else 块也只有一条语句（如果有的话）。一个短形式的—
形式 if 看起来是这样的：

```
if x == null then x = 1
```

……而短形式的 if/else 看起来是这样的：

```
if x >= 0 then print “positive” else print “negative”
```

注意 end if 不用于短形式的 if 或 if/else。此外，也没有办法把多于一条语句
放进 then 或 else 块。若你需要多于一条语句，
就使用标准的多行形式。

使用 for 进行循环

一个 for…in 语句会将一段代码块循环执行零次或多次。语法是：

```
for 变量 in 列表
…
end for
```

整个代码块被称为 for 循环。在每次循环迭代中，变量
会被赋为指定列表中的一个值。你将在“数据类型”章节中了解更多关于列表的内容，
但目前只需知道，你可以使用
range 函数轻松创建一个数字列表。

这个示例从 10 倒数到 1，然后发射升空。

```
for i in range(10, 1)
print i + “…”
end for
print “Liftoff!”
```

参见“内置函数”章节中的 range，了解更多相关选项。

除了列表外，你也可以遍历一个文本字符串。在这种情况下，循环变量会
按顺序被赋为该字符串的每个字符。

最后，也可以遍历映射。同样，映射会在“数据类型”章节中解释，
但只需记住，当你对一个映射使用 for 语句时，那么在
每次循环迭代中，你的循环变量都是一个包含 key 和
value 的小型迷你映射。例如：

```
m = {1:“one”, 2:“two”, 3:“three”}
for kv in m
print “Key “ + kv.key + “ has value “ + kv.value
end for
```

这会打印出映射中的每一对键值对。

使用 while 循环

在 MiniScript 中遍历代码的另一种方式是使用 while 循环。语法如下：

```
while 条件
…
end while
```

只要 条件 为真，就会执行其中的代码。更具体地说，它先计算条件，如果不为真，就直接跳到 end while 。如果为真，则执行循环中的各行，然后跳回到 while 语句。该过程会一直重复，直到条件变为假，或者永远持续。

下面以本手册的第一个例子来说明，这里再次给出。

```
s = “Spam”
while s.len < 50
s = s + “, spam”
end while
print s + “ and spam!”
```

这段代码通过不断追加 spam 来构建字符串（s），只要字符串长度小于 50。一旦不再小于 50，循环就会退出，并打印结果。

break 与 continue

还有两个关键字可以让你提前跳出 while 或 for 循环。首先，break 语句会直接跳出循环，到 end for 或 end while 之后的下一行。请看下面的例子。

```
while true
if time > 100 then break
end while
```

当你看到 while true（或 while 1 ，两者等价）时，它就是一个无限循环——除非 在循环体里有 break 语句。一旦该 break 语句执行，我们就会直接跳出循环。对 for 循环也是完全一样的。在嵌套循环的情况下， break 只会跳出最内层循环。

该 continue 语句会跳过循环体的其余部分，并继续下一次迭代。 这通常用于大型循环中的“提前退出”情况，在某些条件下你想跳过一次迭代并直接进入下一次。

```
for i in range(1,100)
if i == 42 then continue
print “Considering “ + i + “…”
end for
```

这将输出 1 到 100 的数字，除了 42，因为它被跳过了。 注意，如果你只是把你只是把 continue 改为 break 在这个例子中，循环将输出数字 1 到 41，然后停止。

真值的本质

我们已经谈过把条件求值为真或假，却没有解释这到底意味着什么。 通常你不需要担心它，但这里还是给出细节。

MiniScript 中的布尔（真/假）值用数字表示。 当条件在为 if 和 while 语句中求值时，值为 0（零）被视为假；任何其他值都被视为真。 实际上，内建关键字 true 和 false 分别完全等同于数字 1 和 0。

当你使用比较运算符如 ==（等于）、!=（不等于）、 >（大于），以及 <=（小于或等于）时，它们比较操作数并求值为 1（若为真）或 0（若为假）。

更多可用于数字的布尔运算，请参见“数据类型”一章中的“数字”节（包括 and、or，以及 not）。

最后，在需要真值的上下文中——也就是在 if 和 while 语句中，或作为 and、or，以及 not 的操作数时——其它数据类型若为空则被视为假，若不空则被视为真。 因此，空字符串、列表或映射等同于 0（零），而任何非空的字符串、列表或映射在这些上下文中等同于 1。 特殊值 null 始终被视为假。

数据类型

你可以存储和操作的事物

MiniScript 中的变量是动态类型的；你可以在任何变量中存储任何类型的数据。但有哪些数据类型呢？在 MiniScript 中，有四种主要数据类型：数字，字符串，列表，以及映射。还有一些更冷门的类型，比如函数和 null。其它所有东西，包括类和对象，实际上都是映射的一种特殊情况。

数字

MiniScript 中的所有数值都以标准全精度格式存储（也称为 C 系语言中的“double”）。数字也用于表示 true（1）和 false（0）。

数值字面量写作普通数字，例如 `42`，`3.1415`，或 `-0.24`。

你可以对数字使用以下运算符（其中 a 和 b 是数字）。

a + b	加法	a 与 b 的数值和
a - b	减法	a 与 b 的数值差
a * b	乘法	a 乘以 b
a / b	除法	a 除以 b
a % b	取模	a 除以 b 后的余数
a ^ b	幂	a 的 b 次幂
a and b	逻辑与	a * b，限制在区间 [0,1] 内
a or b	逻辑或	a + b - a*b，限制在区间 [0,1] 内
not a	取反	1 - abs(a)，限制在区间 [0,1] 内
a == b	相等	若 a 等于 b 则为 1，否则为 0
a != b	不等	若 a 不等于 b 则为 1，否则为 0
a > b	大于	若 a 大于 b 则为 1，否则为 0
a >= b	大于或等于	若 a 大于或等于 b 则为 1，否则为 0
a < b	小于	若 a 小于 b 则为 1，否则为 0
a <= b	小于或等于	若 a 小于或等于 b 则为 1，否则为 0

注意 `and`、`or`，以及 `not` 不是函数；它们是运算符，放在两个操作数之间（或者在 `not` 的情况下，在之前），与其他运算符一样。

你可以用 isa运算符来检查一个变量是否包含数字
内建类名为 number，而 x isa number 在 true（1）当 x确实是数字时，会返回
一个数字。

字符串

MiniScript 中的文本值以 Unicode 字符串存储。代码中的字符串字面量由双引号（
“）包围。务必使用普通的直引号，而不是某些文字处理器执意替换

如果你的字符串字面量需要包含引号，可以通过把引号输入两次来实现。比如：

```
s = “如果你不帮助我们，我们就对你说 “”Ni””。”
```

字符串可以用 +运算符连接；如果你尝试把数字和字符串相加，数字会自动转换为
字符串也可以通过与数字相乘或相除来复制（重复）或缩短为其原来的一部分。

```
s = “Spam” * 5
s = s / 2
```

完整的字符串运算符如下； s和 t是字符串，而 n和 m 是数字。

s + t	连接	将 t 连接到 s 后形成的字符串
s - t	减法（截去）	如果 s 以 t 结尾，则返回去掉 t 的 s；否则直接返回 s
s * n	复制	s 重复 n 次（包含 s 的某个分数字段）
s / n	除法	等价于 s * (1/n)
s[n]	索引	s 的第 n 个字符(所有索引都从 0 开始；负索引从末尾计数)
s[:n]	左切片	s 从开头到但不包含第 n 个字符的子串
s[n:]	右切片	s 从第 n 个字符到末尾的子串
s[n:m]	切片	s 从第 n 个字符到但不包含第 m 个字符的子串
s == t	相等	若 s 等于 t 则为 1，否则为 0（所有字符串比较都区分大小写）
s != t	不等	若 s 不等于 t 则为 1，否则为 0
s > t	大于	若 s 大于（排序在后）t 则为 1，否则为 0
s >= t	大于或等于	若 s 大于或等于 t 则为 1，否则为 0
s < t	小于	若 s 小于（排序在前）t 则为 1，否则为 0
s <= t	小于或等于	若 s 小于或等于 t 则为 1，否则为 0

上表不包括 and、or，以及 not，但这些运算符通过布尔强制转换在字符串上完全可(参见上一章中的“真理的本质”)。在
任何布尔语境中，若 s 含有任意字符则视为真，若它是
空字符串则视为假。

此外未列出的是 isa 运算符在字符串上的行为。有一种内建类型叫 string，且 s isa string 对任意字符串 s。true(1) 都会返回

切片运算符值得稍作解释。基本语法是 s[n:m]，它获取一个
从字符 n 开始、一直到（但不包括）字符 m 的 s 的子串，其中
我们从 0 开始给字符编号。但这一基本语法还扩展了几条
巧妙的用法：

1. 你可以只指定一个索引，省略冒号，以获得单个字符。因此
   s[0] 是第一个字符， s[1] 是第二个，等等。
2. 你可以使用负索引，它将从末尾计数。所以 s[-1] 是最后一个
   字符， s[-2] 是倒数第二个，等等。这对任何切片索引都适用。
3. 你可以在双索引形式中省略第一个索引，它将默认为 0。这是一个
   获取字符串前 n 个字符的便捷方法。所以 s[:3] 返回 s 的前 3 个字符
   ； s[:-3] 返回 s 除最后三个字符以外的所有字符。
4. 你可以在双索引形式中省略最后一个索引，它将一直延续到字符串末尾。
   因此， s[3:] 跳过前三个字符并返回字符串其余部分。

这些索引的工作方式带来了许多非常便利的性质。例如，s[:n] + s[n:] == s 对从 0 到 s.len 的任意 n 值都成立；换句话说，有一种非常
自然的语法可以把字符串分成两部分，而这是一件相当常见的事情。

最后，请注意字符串是 不可变的 ；就像数字一样，你永远不能改变一个字符串，
但你可以创建一个 新的字符串并将其赋给一个已有变量。下面的例子
展示了将“spin”变为“spun”的一种正确方式和一种错误方式。

```
s = “spin”
s = s[:2] + “u” + s[3:]
s[3] = “u”
```

列表

MiniScript 中第三种基本数据类型是 list。这是一个有序的元素集合，
可通过从零开始的索引访问。列表中的每个元素都可以是任意类型，包括
另一个列表。

你可以在元素外加方括号来定义一个列表，元素之间应当用
逗号分隔。

```
x = [2, 4, 6, 8]
```

上面的代码创建了一个包含四个元素的列表，并将其赋给 x。 但同样，列表元素
不必是数字；也可以是字符串、列表或映射。 下面是另一个例子。

```
x = [2, “four”, [1, 2, 3], {8:“eight”}]
```

对列表的操作与对字符串的操作非常相似。 你可以用
+连接两个列表，用 *和 /复制或裁剪列表，并用相同的
切片语法来访问元素或子列表。 下面是对列表有效的运算符，其中 p 和 q 是列表，而 n 和 m 是
数字。

p + q	连接	将 q 连接到 p 所形成的列表
p * n	复制	p 重复 n 次（包括 p 的某个分数部分）
p / n	除法	等价于 p * (1/n)
p[n]	索引	p 的第 n 个元素（所有索引从 0 开始；负索引从末尾计数）
p[:n]	左切片	p 的子列表，直到但不包括第 n 个元素
p[n:]	右切片	p 从第 n 个元素到末尾的子列表
p[n:m]	切片	p 从第 n 个元素到但不包括第 m 个元素的子列表

此外，你还可以用 x isa list 来检查某个变量 x是否包含列表。

切片运算符的工作方式与字符串完全相同。 因此 p[-1] 是列表 p 的最后一个元素；
p[3:] 会跳过前三个元素并返回列表的其余部分，等等。

不过，有一个重要差别：列表是 可变的 。 你可以改变
列表的内容（通过给 p[n] 赋值，或使用诸如 p.push 等列表方法），并且无论
有多少不同的变量引用该列表，它们都会看到变化。 下面的例子说明了这一点。

```
a = [1, 2, 3]
b = a
a[-1] = 5
print b
```

因为 a 和 b 都引用同一个列表，对该列表所做的任何改动（无论
）都能从任一变量看到。

如果你希望确保得到列表的新副本而不是共享引用，一个常见的技巧是使用
[:] 来生成一个包含整个列表的切片。 这会把元素复制到一个新列表中。 将下

```
a = [1, 2, 3]
b = a[:]
a[-1] = 5
print b
```

映射

MiniScript 中最后一种基本数据类型是 映射。映射是一组键值对，其中每个唯一的键都映射到某个值。在一些编程环境中，这个相同的概念被称为 字典 。

用花括号包住逗号分隔的键值对列表来创建映射。通过冒号分隔键和值来指定每一对，如下所示。

```
m = {1:“one”, 2:“two”, 3:“three”}
```

这里创建的映射包含三对键值，每一对把一个数字映射到一个字符串(在这个例子中恰好是该数字的英文单词)。

映射的键应为数字或字符串，并且必须唯一；如果你复用一个键，之前的值会被替换。(从技术上讲，键也可以是列表或另一个映射，但在这种情况下，重要的是你不要修改该键，否则行为未定义。)值可以是任何类型，包括列表或映射。映射中的顺序不会被保留；for 循环会以任意顺序遍历一个映射。

映射只支持少数几个运算符（ d和 e是映射， k是一个键，且v是一个值):

d + e	连接	通过对 e 中每个 k,v 赋值 d[k] = v 而形成的映射
d[k]	索引	d 中与键 k 关联的值
d.k	点索引	d 中与（字符串）k 关联的值

获取和设置映射成员有两种方式。第一种是使用方括号索引运算符，就像字符串或列表一样，不同之处在于映射的情况下，键可以是字符串也可以是数字（甚至可以是列表或另一个映射，只要你非常小心)。

```
d = {“yes”:“hai”, “no”:“iie”, “maybe”:“tabun”}
print d[“maybe”]
d[“maybe”] = “kamo”
print d[“maybe”]
```

第二种方式是使用 点索引器 。这只在键是有效标识符的字符串这种特殊情况下生效：它以字母开头，并且只包含字母、数字，和下划线。在这种情况下，你可以把键写在点号后面，而不是把它放在方括号和引号中——该键本质上成为语言中的一个标识符。下面的写法在功能上等价于前一个例子。

```
d = {“yes”:“hai”, “no”:“iie”, “maybe”:“tabun”}
print d.maybe
d.maybe = “kamo”
print d[“maybe”]
```

这个点索引器大多只是语法糖，使得读写 map 的元素更容易。
但当 map 表示一个类或对象时，会有一些微妙的差别，
下一章会描述这些差别。

最后，像其他基本类型一样，这里有一个表示 map 的内建类——map。因此
在这种情况下。x于是isa map 对任何 map（包括任何类或对象，如你将在下一节看到的那样）都会返回 tru
你将在下一节看到）。

类型检查

上面多次提到isa运算符。它让你能在运行时检查你手头的数据类型。很多情况下你不会在意，得益于 MiniScrip
自动类型转换。但有时你会在意。

比如说，你想做一个方法，把它的参数用括号包起来打印……但如果调用者传入一个列表，
你就想把该列表的元素用逗号连接。你可以用
isa来实现这一点。

```
spew = function(x)
if x isa list then x = x.join(“, “)
print “(“ + x + “)”
end function

spew 42
spew [18, 42, “hike!”]
```

扩展内建类型

四种内建类型——number、string、list以及 map——都只是普通的 map，就像你自己的类（
然后用点语法在普通的数字、字符串、列表和 map 上调用这些方法。（唯一的限制是你不能在数值字面量
法。）如果这些听起来像天书，别担心——这是一个高级特性，而且大多数用户永远用不到。

完整运算符列表

下表显示 MiniScript 语言中的所有运算符，以及它们的
优先级。表达式链中的操作数将总是先按更高优先级
运算符分组，再按更低优先级分组；例如，`x + y * z` 会被处理为 `x + (y * z)`，
因为 `*`运算符的优先级高于 `+`运算符。

A = B	赋值	0
A or B	逻辑或：任一操作数为真则为真	1
A and B	逻辑与：两个操作数都为真则为真	2
not A	逻辑非：其操作数为假则为真，反之亦然	3
A isa B	类型检查	4
A == B	相等比较：操作数相等则为真	5
A != B	不等比较：操作数不相等则为真	5
A > B	大于比较	5
A < B	小于比较	5
A >= B	大于或等于比较	5
A <= B	小于或等于比较	5
A + B	加法或拼接	6
A - B	减法或字符串裁剪	6
A * B	乘法或复制	7
A / B	除法或缩减	7
A % B	取模（余数）	7
-A	一元负号（数值取负）	8
new A	实例化	9
@A	取地址（引用函数而不调用它）	10
A ^ B	幂：A 的 B 次方	11
A[B]	索引	12
A[B:C]	切片	12
A(B, C…)	调用函数	12
A.B	点运算符	12

函数与类

复杂软件的构建基石

一函数 本质上是执行某项特定任务的子程序。我们已经见过 MiniScript 内置的一些函数，例如 time 和 range，甚至 print。还有更多此类函数，将在下一章中记录。但编程语言的真正力量来自于定义你自己的函数。

此外，随着程序规模和复杂度的增长，开始将其组织为 类 会变得很有用。类基本上是函数与数据的集合，其中某个类的 对象 共享相同的函数，但可能拥有各自独特的数据。

函数

MiniScript 中的函数是一种特殊的数据类型，与数字、字符串、列表和映射处于同一层级。你可以使用 function 关键字定义函数，将其赋给一个变量，然后通过该变量来调用它，就像内置函数一样。下面是一个例子。

```
triple = function(n=1)
return n*3
end function
print triple
print triple(5)
```

这声明了一个函数，它会将传入的任意值乘以三，并把该函数赋给名为 triple 的变量。随后调用 triple 函数，分别带参数与不带参数。

声明函数的语法是：

```
function( 参数 )
…
end function
```

其中 参数 是由零个或多个参数组成的逗号分隔列表，每个参数的形式为 名称 或 名称 =默认值 。当调用函数时，实参会按位置与参数匹配。若给出的实参少于已定义的参数，剩余的参数将使用其默认值——如果该参数没有定义默认值，则它会被设为 null。

请注意，function 关键字后面的括号只有在存在参数时才需要。在没有参数的函数中，括号不是必需的（并且按标准惯例，应当省略）。

理解函数本身就是一段数据，这一点很重要。只是当你
查找某个变量的值时，MiniScript 会检查这种特殊的函数数据类型；若
找到，就会调用该函数，而不是返回函数本身。

通常这正是所需的，如上面的例子。但偶尔你可能
想要复制函数引用，而不是调用该函数。你可以通过
在标识符前加上一个 @（读作“取地址”）来做到这一点。示例：

```
triple = function(n=1)
return n*3
end function
x = @triple
print x(5)
```

这里我们再次声明了一个函数，并将其存储在名为 triple 的变量中。然后我们
将该函数的地址复制到另一个名为 x 的变量中。此时我们可以通过
任一种方式调用该函数，即通过 triple 或通过 x，两者的效果完全相同。如果我们
在赋值中省略了 @，MiniScript 就会改为求值
triple 所指代的函数，并将结果（3）赋给 x。

这里有一个更现实的例子。我们将定义一个名为 apply 的函数，它可以对
列表中的每个元素应用给定的函数。然后我们可以用任何函数在一个列表上调用它，
只需使用 @ 来引用我们想要应用的函数即可。

```
apply = function(lst, func)
result = lst[:]
for i in indexes(result)
result[i] = func(result[i])
end for
return result
end function

print apply([1, 2, 3], @triple)
```

总之，你只需使用任何引用该函数的标识符即可调用它。你
要避免这种调用、转而引用函数本身，则应在 @ 放在该
标识符之前。

嵌套函数

MiniScript 允许你在函数内部定义函数。这是一个高级特性，
大多数用户可能永远用不上，但偶尔会派上用场，尤其是与
上面的 “apply” 方法之类的东西配合时。就像任何其他局部值一样，你可能
不想为了一个只在某一处使用的函数而
弄乱全局命名空间。这里有一个简单示例，假设我们已经定义了上面的 apply 方法：

```
doubleAll = function(lst)
```

```
f = function(x)
return x + x
end function
return apply(lst, @f)
end function
```

因此，在由（全局变量）所引用的函数内部 doubleAll ，我们定义另一个函数，并把它赋值给（局部变量） f。然后我们把该函数作为第二个参数传给 apply 函数（或者更准确地说，传给由 apply 全局变量所引用的函数）。

当你有这样一个嵌套函数时，它可以访问包含它的函数的局部变量。就像全局变量一样，它可以不加任何前缀地做到这一点（只要没有某个同名的局部变量妨碍）。但要给外层函数的变量赋值，你必须使用特殊标识符 outer 。这里有个例子。

```
makeList = function(sep)
counter = 0
makeItem = function(item)
outer.counter = counter + 1
return counter + sep + item
end function
return [makeItem(“a”), makeItem(“b”), makeItem(“c”)]
end function

print makeList(“. “)
```

这里，makeList 指的是外层函数，而 makeItem 是内层（嵌套）函数。注意 makeList 有一个名为 counter 的局部变量，初始化为 0。但内层函数既读取该值，又使用 outer.counter 来更新它。仔细推敲这段代码，看看你能否推断它会打印什么……然后试一试，看看你是否猜对了！

再次强调，这个嵌套函数的内容是一个高级特性，初学者完全可以先不用理会。但对高级用户来说，这是一个值得理解的语言特性。

类与对象

MiniScript 支持面向对象编程（ OOP ），通过基于原型的继承。
也就是说，在 MiniScript 中类与对象本质上没有区别；差异，若确实存在，完全
只体现于程序员的意图。

类或对象是一个带有特殊 __isa 条目的映射，该条目指向父级（原型）。
这是一个实现细节，你很少需要担心，因为它由以下规则自动处理：
自动处理：

1. 当你使用特殊的 new 运算符创建一个映射时，__isa 成员会为你设置。
2. 当你在一个映射中查找标识符时，MiniScript 会沿着 __isa 链查找
   包含该标识符的映射。返回的值是找到的第一个值。
3. 最后， isa 运算符也会沿着 __isa 链查找，并在该链中有任何映射与右侧操作数匹配时返
   换句话说， x isa y返回 true 若x是 y，或
   是 y的任何子类。

这些简单的规则几乎提供了面向对象编程所需的一切。可以将一系列“类”定义为包含函数和默认数据的
“对象”，并根据需要进行继承或重写。继承自某个类，通常只包含自定义数据。
让我们用一个例子来说明。我们将定义一个名为 Shape 的类，以及一个名为 Square 的子类。
Square。

```
Shape = {}
Shape.sides = 0

Square = new Shape
Square.sides = 4
```

基类只是一个普通映射；在这个例子中，我们添加了一个 sides 条目，值为 0，
表示“sides”是我们期望每个 Shape 都具备的一点数据。接着我们通过使用
创建了一个子类 new Shape ，并将其赋给 Square。在 Square 中，我们重写了
的值 sides（因为所有正方形都应有 4 条边）。

现在让我们创建 Square 类的一个实例，同样使用 new 。

```
x = new Square
print x.sides
```

注意我们为了方便使用了传统 OOP 术语“类”和“实例”，但实际上只有三个映射 —— Shape 是 Squar
而 Square 是 x 的原型。每个映射的
__isa 成员指向原型，因为我们用
创建它们时使用了 new 。

现在让我们给 Shape 类添加一个函数，它应当适用于任何形状子类或对象。

```
Shape.degrees = function
return 180 * (self.sides - 2)
end function

print x.degrees
```

这个例子说明了面向对象编程中另一个重要规则：

- 当通过点索引调用函数时，它会接收一个特殊的 self 变量，引用被调用的那个对象。

因此在上面的例子中，我们以 degrees 函数的形式 x.degrees 来调用它，这会在 x 中查找名为“degrees”的成员（以及通过 __isa 链的原型）。而当该函数被调用时，一个名为 self 的特殊局部变量会被赋予 x 对象，也就是搜索链中的第一张映射。这使得类函数能够访问对象数据。

面向对象编程还有一个特别的支持点，那就是关键字 super。这是另一个内建变量（类似于 self），当你用点语法调用方法时会super 调用另一个方法时，它会在基类上调用该 super 方法，同时保持 self 绑定为与当前函数相同的值。换句话说，super 让你调用超类方法，即便你正在重写它。继续上面的例子，假设我们要定义 Square 的一个子类，它总是比非魔法形状多 42 度：

```
MagicSquare = new Square
MagicSquare.degrees = function
return super.degrees + 42
end function

y = new MagicSquare
print y.degrees
```

请注意 MagicSquare.degrees 函数如何调用 super.degrees。这会使 MiniScript 沿着 __isa 链查找第一个对 degrees 的实现并找到它。那会是 Shape.degrees，于是它调用它，同时 self 仍绑定到 y。

扩展内置类型

有一些映射代表每一种基本数据类型：number、string、list，以及 map。这些包含了这些类型的内置方法。通过向这些映射中的某一个添加新方法，你可以为该类型的值添加可用点语法调用的新方法。

例如，虽然有内置的字符串方法 .upper 和 .lower 用于把字符串转换为全大写或全小写，但没有把字符串首字母大写的方法——也就是只把第一个字母转换为大写。不过你可以在程序中如下添加这样的方法。

```
string.capitalized = function
if self.len < 2 then return self.upper
return self[0].upper + self[1:]
end function
```

这个函数本身相当简单：如果我们的字符串（self）长度少于 2 个字符，就把整个字符串转为大写并追加其余部分。但因为我们已将这个函数赋值给 string.capitalized，也就是把它添加到 string 映射中，我们就能在任何字符串上用点语法调用它。

```
print “miniScript”.capitalized
```

这个技巧有一个限制。数字与其他数据类型略有不同；MiniScript 不支持在数字字面量上使用点语法。所以

```
x = 42
x.someMethod
```

可以正常工作（假设你已定义了合适的 number.someMethod 函数），但是

```
42.someMethod
```

则不行。

固有函数

你可以依赖的内置函数

MiniScript 自带一套标准的内置（或 固有 ）函数。 其中许多是
全局的（即由全局空间中的变量引用）。 其他（尤其是用于字符串、列表和映射的函数）通常通过点语法
标识符之后调用。
标识符。

事实上，所有使用点语法的固有函数都以这样的方式编写，使它们
也可以 可以作为全局函数调用。 所以，例如，你可以通过输入 s
来获取字符串的长度 s.len，但你也可以用 len(s) 做到同样的事。

下表列出了标准的固有函数，按其所操作的数据类型划分。
请记住，MiniScript 旨在嵌入到某个宿主
环境中，例如游戏或应用程序。 宿主通常会添加该环境特有的额外固有
函数。 请查阅你的宿主环境的文档或帮助
资料以了解这些额外函数的信息。

数值函数

MiniScript 包含一组三角函数，全部使用弧度（而
不是度数），以及其他数学函数，还有随机数与将
数字转换为字符串的功能。

在下表中， x是任意数字， i是整数，而 r是弧度数。

abs(x)	x 的绝对值
acos(x)	x 的反余弦，单位为弧度
asin(x)	x 的反正弦，单位为弧度
atan(y, x=1)	y/x 的反正切，单位为弧度（若使用可选参数 x，则返回正确象限）
bitAnd(x, y)	将 x 与 y 视为整数，并返回 a 与 b 的按位“与”
bitOr(x, y)	将 x 与 y 视为整数，并返回 a 与 b 的按位“或”
bitXor(x, y)	将 x 与 y 视为整数，并返回 a 与 b 的按位“异或”
ceil(x)	大于或等于 x 的下一个整数
char(i)	返回码点为 i 的 Unicode 字符（其逆函数参见字符串 .code）
cos(r)	r 弧度的余弦
floor(x)	小于或等于 x 的下一个整数

log(x, base=10)	x 的对数（以给定的底为底），即使得 base^y == x 的 y 值
pi	3.14159265358979
range(x, y=0, step=null)	返回一个列表，包含从 x 到 y 的值，按 step 递增；若 step == null，当 y > x 时视为步长 1，否则视为 -1
round(x, d=0)	将 x 四舍五入到 d 位小数
rnd(seed=null)	若 seed=null，返回区间 [0,1) 内的随机数； 若 seed != null，用给定的整数值为随机数生成器设定种子
sign(x)	x 的符号：x < 0 时为 -1；x == 0 时为 0；x > 0 时为 1
sin(r)	r 弧度的正弦值
sqrt(x)	x 的平方根
str(x)	将 x 转换为字符串
tan(r)	r 弧度的正切值

字符串函数

除 slice 外，所有字符串函数都设计为通过点语法在字符串上调用，但也可作为全局函数调用，并将字符串作为第一个参数传入。注意字符串是不可变的；所有字符串函数都会返回一个 新的字符串，使原字符串保持不变。在下表中，sel指代该字符串，s是另一个字符串参数，而i是一个整数。

.code	self 的首字符的 Unicode 码点（反向见数值 char函数）
.hasIndex(i)	若 i 在 0 到 self.len-1 的范围内则为 1；否则为 0
.indexes	range(0, self.len-1)
.indexOf(s, after=null)	self 中子串 s 第一次出现的基于 0 的位置，若未找到则为 null；可选地在给定位置之后开始搜索
.insert(index, s)	返回在位置 0 插入 s 后的新字符串
.len	self 的长度（字符数）
.lower	self 的小写版本
.remove(s)	self，但移除子串 s 的首次出现（若有）
.replace(oldval, newval, maxCount=null)	返回一个新字符串，将子串 oldval 的出现最多替换 maxCount 次为 newval（若 maxCount 未指定，则替换所有出现）
.upper	self 的大写版本
.val	将 self 转换为数字（若 self 不是有效数字，则返回 0）
.values	self 中各个字符的列表（例如 “spam”.values = [“s”, “p”, “a”, “m”]

slice(s, from, to)	等同于 s[from:to]
.split(delimiter=" ",maxCount=null)	按给定分隔符将字符串拆分为列表，最多 maxCount 个条目（如果未指定 maxCount，则拆分为任意大小的列表）

列表函数

除 slice 外，所有列表函数都设计为使用点语法在列表上调用，但也可以作为全局函数调用，将列表作为第一个参数传递。the 列表是可变的；

pop、pull、push、shuffle，以及 remove 等函数会就地修改列表。要把列表当作栈使用，用 like a stack, add items with push 添加条目，并用 pop 移除它们。要把列表当作队列使用，用 items with push 添加条目，并用 pull 移除它们。

在下表中，sel是列表，i是整数，而 x是任意值。

.hasIndex(i)	如果 i 在 0 到 self.len-1 的范围内则为 1；否则为 0
.indexes	range(0, self.len-1)
.indexOf(x, after=null)	self 中第一个匹配 x 的元素的 0 基位置，若未找到则为 null；可选地从给定位置之后开始搜索
.insert(index, value)	将 value 插入到 self 的给定索引处（就地）
.join(delimiter=" ")	用给定分隔符连接元素来构建字符串
.len	self 的长度（元素数量）
.pop	移除并返回 self 的最后一个元素（像栈一样）
.pull	移除并返回 self 的第一个元素（像队列一样）
.push(x)	将给定值追加到 self 末尾；常与 pop 或 pull 一起使用
.shuffle	随机重排 self 的元素（就地）
.sort(key=null)	就地排序列表，可选地按给定键的值排序（例如在由映射组成的列表中）
.sum	self 中所有数值元素的总和
.remove(i)	从 self 中移除索引 i 处的元素（就地）
.replace(oldval, newval, maxCount=null)	将列表中最多 maxCount 个 oldval 的出现（就地）替换为 newval（如果未指定 maxCount，则替换所有出现）
slice(list, from, to)	等同于 list[from:to]

映射函数

映射上的函数与列表上的函数非常相似。映射（像列表一样）是可变的；**push**，**pop**，**remove**，以及**shuffle**方法会就地修改映射。你可以把映射当作集合使用，通过使用 **push**，它会为给定键插入值 1（true），以及 **pop**，它返回一个键并将其（以及其值）从映射中移除。请记住，映射中键的顺序是未定义的。键在映射中的顺序是未定义的。

在下表中，sel是一个映射，x是一个整数，并且x是任意值。

.hasIndex(x)	若 x 是 self 中包含的键则为 1；否则为 0
.indexes	包含 self 所有键的列表，顺序任意
.indexOf(x, after=null)	self 中第一个映射到 x 的键；若不存在则为 null；可选地从给定键之后开始搜索给定的键
.len	self 的长度（键值对数量）
.pop	移除并返回 self 中任意一个键
.push(x)	等价于 self[x] = 1
.remove(x)	从 self 中移除键= x 的键值对（就地）
.replace(oldval, newval, maxCount=null)	在映射中就地将值 oldval 的出现最多 maxCount 次替换为 newval（若未指定 maxCount，则替换所有出现）
.shuffle	随机重新映射键的值
.sum	self 中所有数值的总和
.values	包含 self 所有值的列表，顺序任意

系统函数

以下函数与 MiniScript 自身的运行相关，或与宿主环境交互。
其中后者（print、time 和 wait）仅为准标准，因为是否支持它们取决于
宿主应用，因此在某些环境中可能无法工作。

globals	指向全局变量映射的引用
intrinsics	包含所有全局内建函数的映射
locals	指向当前调用帧的局部变量映射的引用
print(x, delim)	将 x 转换为字符串并打印到某个文本输出流，可选地在其后输出 delim; 若未指定 delim，在大多数环境中输出后会跟一个换行符
refEquals(a,b)	若 a 与 b 引用同一实例（不仅是值相等）则返回 1
stackTrace	返回当前调用栈，形式为字符串列表
time	自程序开始执行以来的秒数
wait(x=1)	等待 x 秒后再执行下一条 MiniScript 指令
yield	等待主引擎循环的下一次调用（例如游戏中的下一帧）

示例

做有趣事情的小程序

在本手册中我们已经给出许多简短的 MiniScript 代码示例，而这一章展示了几个更长、更有趣的例子。这里演示的许多任务取自 RosettaCode，这是一个在线的编程挑战数据库，提供多种语言的解答。你可以去那里对比 MiniScript 的解法与任何其他语言的解法；你或许会惊讶于 MiniScript 比其它选择可读得多。

FizzBuzz

FizzBuzz是一个标准的入门级编程挑战。任务很简单：打印 1 到 100 的数字，但：对于 3 的倍数，用“Fizz”代替该数字；对于 5 的倍数，用“Buzz”代替该数字，而对于任何同时是 3 的倍数且 5 的倍数的数字，打印“FizzBuzz”。[1]

显然有很多方法可以完成这个任务；这里给出一种。

```
1.fizzBuzz = function(n)
2.for i in range(1, n)
3.s = “Fizz” * (i%3==0) + “Buzz” * (i%5==0)
4.if s == “” then s = str(i)
5.print s
6.end for
7.end function
8.fizzBuzz 100
```

我们没有只是硬编码一个从 1 到 100 的循环，而是做了一个函数，可以对任意数字执行 FizzBuzz。在该函数中，唯一巧妙的地方是第 3 行，它利用了 MiniScript 的几个特性。首先，比较（例如 `i%3==0` ——读作“i mod 3 等于 0”）在为真时求值为 1，为假时求值为 0。其次，你可以将字符串乘以一个数字来重复该字符串相应次数。这意味着，如果你用一个条件去乘字符串，就会得到原字符串（条件为真）或空字符串（条件为假）。

这样我们就能轻松生成“Fizz”、“Buzz”和“FizzBuzz”，取决于循环计数器可被哪些数整除。第 4 行只是当我们没得到这些字符串时填入数字。（小测验：你能把这一行改写为使用与第 3 行相同的“按条件相乘”技巧吗?）

[1] http://rosettacode.org/wiki/FizzBuzz

过滤

这里是另一个 RosettaCode 任务[2]：以一种通用的方式从数组中选择某些元素到一个新数组中。作为演示，从数组中选出所有偶数。

```
filter = function(seq, f)
result = []
for i in seq
if f(i) then result = result + [i]
end for
return result
end function

isEven = function(x)
return x % 2 == 0
end function

list = [2,3,5,6,8,9]
print filter(list, @isEven)
```

这是将任务描述直接转换为 MiniScript 代码的一个相当直观的实现。我们的 `filter` 函数接受一个列表和一个函数，并通过把每个元素依次追加到新列表来构建结果，条件是该元素应用函数后为真。

我们用一个 `isEven` 函数来演示，它只有在参数对 2 取模为零时才返回真（也就是说，该参数能被 2 整除）。随后我们传入 `@isEven` 来找出给定列表中所有的偶数元素。

最大公约数

这里有一个函数，用来找出能整除给定两个数的最大数。到处的中学生很快就要失业了。[3]

```
gcd = function(a, b)
if a == 0 then return b
while b != 0
newA = b
b = a % b
a = newA
end while
return abs(a)
end function
print gcd(-21, 35)
```

这里的算法称为“用于求最大公约数的欧几里得算法”，很巧妙。实际的 MiniScript 代码很简单。

[2]http://rosettacode.org/wiki/Filter

[3]http://rosettacode.org/wiki/Greatest_common_divisor

最大元素

MiniScript 没有用于查找列表最大元素的标准内建函数。但
你可以用下面的代码轻松自己添加。

```
1.max = function(seq)
2.if seq.len == 0 then return null
3.max = seq[0]
4.for item in seq
5.          if item > max then max = item
6.end for
7.return max
8.end function
9.print max([5, -2, 12, 7, 0])
```

很简单。第 2 行检查以确保狡猾的用户没有给我们一个空列表;
如果给了，我们就返回 null，因为在这种情况下没有合理的最大值。否则，
我们就假设第一个元素为最大值，然后遍历列表中的每个元素，保持当前
最大的那个。

注意，max 在第 1 行被赋值的变量位于全局变量空间，而
max 在第 3 行和第 5 行（并在第 7 行返回）被赋值的是函数的局部变量。这两个
恰好同名，但彼此毫无关系。就风格而言，可能把局部变量命名为
result 而不是 max 会更好。但这似乎是一个很
即使它们同名也一样。
它们同名。

标题式大小写

MiniScript 有将字符串转换为全大写或全小写的内建函数。但如果
你想只把每个单词的首字母大写，而其余字母小写呢?

```
1.titlecase = function(s)
2.result = “”
3.for i in s.indexes
4.          if i == 0 or s[i-1] == “ “ then
5.                result = result + s[i].upper
6.          else
7.                result = result + s[i].lower
8.          end if
9.end for
10.return result
11.end function
12.print titlecase(“SO LONG and thanks for all the fish”)
```

我们只是遍历字符串，把每个要么是字符串的第一个字符、要么前面是空格的字母大写，其余字母小写。
其余字母小写。

首字母大写（版本 2）

之前版本的首字母大写能正常工作，但有些不够理想，因为它通过逐字符添加来增长字符串。这会多次复制字符串中较早的字符。下面的代码展示了更好的方法。

```
titlecase = function(s)
result = s.split(“”)
for i in s.indexes
        if i == 0 or s[i-1] == “ “ then
              result[i] = s[i].upper
        else
              result[i] = s[i].lower
        end if
end for
return result.join(“”)
end function
print titlecase(“SO LONG and thanks for all the fish”)
```

这里我们先把字符串拆分为字符（使用空字符串作为分隔符来拆分）。然后遍历字符串，更新列表中的每个
最后再把它们连接回去。