Nim basics

TABLE OF CONTENTS

Nim is a relatively new programming language which allows users to write easy-to-read high-performance code. But if you are reading this Nim tutorial, the chances are that you already know about Nim.

The tutorial is available both online and as a book in epub format.

This is a work-in-progress: if you spot any errors and/or you have an idea how to make this tutorial better, please report it to the issue tracker.

Nim 是一门相对较新的编程语言，它能让用户编写出既易于阅读又高性能的代码。但如果你正在阅读这篇Nim教程，很可能你已经对 Nim 有所了解。 本教程提供在线阅读和 epub 格式的电子书版本。

这是一项进行中的工作：如果您发现任何错误，或者有改进本教程的建议，请到 issue tracker 报告。

本教程适合谁

- 没有或仅有少量编程经验的人士
- 有其他编程语言基础的人士
- 想要从零开始首次探索 Nim 语言的人们

这不适合谁?

- 拥有丰富编程经验的人士：其他更高级的教程可能更适合您。可参阅 Official Tutorial 或 Nim by Example。
- 已掌握 Nim 语言的人士（欢迎协助完善本教程）

如何使用本教程?

本教程旨在向你介绍编程基础及Nim语法，以便你能更轻松地跟进其他教程和/或自行深入探索。

与其仅阅读文本内容，最佳方式是亲自尝试、修改示例、构思自己的例子并保持好奇心。某些章节末尾的练习不容忽视——切勿跳过。

若需进一步理解教程内容或完成练习，你随时可在 Nim 论坛、 Nim Gitter 频道 , Discord server 、或 Nim's IRC channel on freenode #nim 寻求帮助。

译者注

本文档归属于 nim-basics-chs项目，文档的所有内容默认均为机翻，如有勘误及补充，欢迎在 issues 提出！

在线浏览： Nim basics 中文文档

nim是一门正在快速迭代的新兴程序语言，因此本文档中的内容不免存在过时情况，请根据自己的实际情况按需阅读和学习。

安装指南

安装 NIM

Nim为三大主流操作系统提供了现成的发行版，安装 Nim 时有多种选择方案。

您可以按照 官方安装流程 来安装最新的稳定版本，或者使用名为 choosenim 的工具——如果您对最新功能和修复感兴趣，它能帮助您轻松切换稳定版和开发版。

无论选择哪种方式，只需按照对应链接中的安装说明操作，Nim 即可完成安装。我们将在后续章节中验证安装是否成功。

若您使用 Linux 系统，您的发行版很可能已通过包管理器收录 Nim。

若采用以上方式安装，请确保获取的是最新版本（可参照官网确认最新版本号），否则请通过上述两种方法之一进行安装。

在本教程中，我们将使用稳定版本。 最初，本教程是为 Nim 0.19（2018 年 9 月发布）编写的，它应该适用于任何更新的版本，包括 Nim 1.x 和 Nim 2.x。 （译者注：截止到2025-06-13，nim官网展示的安装版本是2.2.4。有消息称nim3作为实验分支正在开发中）

安装其他工具

你可以在任何文本编辑器中编写Nim代码，然后通过终端进行编译和运行。如果需要语法高亮和代码补全功能，主流代码编辑器都有提供这些特性的插件。

大多数 Nim 用户偏好使用 VS Code ,搭配 Nim extension 扩展插件实现语法高亮与代码补全，并安装 Code Runner extension 扩展以实现快速编译运行。

The author personally uses NeoVim editor, with this plugin which provides additional features like syntax highlighting and code completion.

如果您使用的是其他代码编辑器，请参阅 the wiki 以获取可用的编辑器支持。

测试安装

为了检查安装是否成功，我们将编写一个传统上用作入门示例的程序： Hello World.

在 Nim 中打印（即在屏幕上显示；而非通过打印机打印在纸上）短语 `Hello World!` 非常简单，且不需要任何模板代码。

In a new text file called e.g. `helloworld.nim` we need to write just one line of code:

Listing 1. helloworld.nim

```
echo "Hello World!"
```

要打印的短语必须跟在 `echo` 命令之后，并且必须用双引号（ " ）括起来。

首先我们需要编译程序，然后运行它看看是否按预期工作。

在文件所在的目录中打开终端（在 Linux 系统中，右键点击文件管理器中的目录，可以选择“在此处打开终端”；在 Windows 系统中，需按住 Shift 键并右键点击，才能获取打开命令行的菜单选项）。

我们在终端中输入以下命令来编译程序:

```
nim c helloworld.nim
```

编译成功后，我们就可以运行程序了。 在Linux系统中，我们可以在终端输入./helloworld来运行程序，而在Windows系统上则需输入helloworld.exe。

还有一种可能，只需一个命令即可同时编译和运行程序。 我们需要输入：

```
nim c -r helloworld.nim
```

c 告诉 Nim 编译该文件，而 -r 则指示它立即运行该文件。 要查看所有编译器选项，请在终端中输入 nim --help 。

如果你使用的是之前提到的带有 Code Runner 扩展的 VSCode，只需按下 Ctrl+Alt+N ，你的文件就会被编译并运行。

无论你选择哪种方式运行程序，稍等片刻后，在输出窗口（或终端）中你应该会看到：

```
Hello World!
```

恭喜，你已经成功运行了你的第一个Nim程序!

现在你已经学会了如何在屏幕上显示内容（使用 echo 命令）、编译程序（在终端输入 nim c programName.nim ）以及运行程序（有多种方式）。

现在我们可以开始探索基本元素，这些元素将帮助我们编写简单的 Nim 程序。

命名值(NAMING VALUES)

在编程中，为程序中的值赋予名称通常有助于我们追踪和管理数据。例如，当向用户询问其姓名时，我们希望存储该姓名以便后续使用，避免每次需要进行相关计算时反复询问。

在示例 `pi = 3.14` 中，名称 `pi` 与值 `3.14` 相关联。 根据经验可知，变量 `pi` 的类型为（十进制）数字。

Another example would be `firstName = Alice`, where `firstName` is the name of a variable with the value `Alice`. We would say that the type of this variable is a word.

编程语言中的机制与此类似。 这些名称分配包含它们的名称(*name*)、值(*value*)和类型(*type*)三个要素。

变量声明(VARIABLE DECLARATION)

Nim 是一种静态类型(statically typed)编程语言，这意味着在使用值之前需要声明其赋值类型。（译者注：其实nim也可以自动推理，有的情况下无需声明。）

在 Nim 中，我们还会区分可以改变或可变的值与不可变的值(mutate)，但这一点稍后再详述。

我们可以使用 var 关键字声明一个变量（即可变赋值），只需通过以下语法指定其名称和类型（值可以稍后添加）：

```
var <name>: <type>
```

如果我们已经知道它的值，可以立即声明一个变量并为其赋值：

```
var <name>: <type> = <value>
```

尖括号（ <> ）用于表示可以更改的内容.
所以 <name> 并非字面意义上的尖括号内的单词 name ，而是指代任意名称。

Nim 还具备类型推断(type inference)能力：编译器能根据赋值自动推断变量类型，无需显式声明。 我们将在下一章深入探讨各种类型。

因此我们可以像这样不指定显式类型来声明变量:

```
var <name> = <value>
```

在 Nim 中，一个示例如下:

```
var a: int  ①
var b = 7   ②
```

① 变量 a 的类型为 int （整数），未显式设置值。(with no value explicitly set.)

② 变量 b 的值为 7 ，其类型被自动推断为整数。

在命名时，选择对程序有意义的名称非常重要。 简单地命名为 a 、 b 、 c 等很快就会变得混乱。 名称中不能使用空格，因为这会将其分割成两部分。 因此，如果你选择的名称由多个单词组成，通常的做法是使用 camelCase 风格书写（注意名称的首字母应为小写）。

但请注意，Nim对大小写和下划线不敏感，这意味着 helloWorld 和 hello_world 会被视为相同的名称。 唯一的例外是首字符，它对大小写敏感。 名称中还可以包含数字和其他 UTF-8 字符，甚至可以使用表情符号（如果你愿意的话），但请记住，你和其他人可能需要输入这些字符。

无需为每个变量单独输入 var，可以在同一个 var 块中声明多个变量（类型不必相同）。 在 Nim 中，块是指具有相同缩进（首字符前空

格数相同）的代码部分，默认缩进级别为两个空格。 在 Nim 程序中，此类块随处可见，不仅用于命名赋值。

```
var
  c = -11
  d = "Hello"
  e = '!'
```

在 Nim 中不允许使用制表符进行缩进.
你可以设置代码编辑器将按下 Tab 转换为任意数量的空格.
在 VS Code 中，默认设置是将 Tab 转换为四个空格。 这可以通过在设置中（Ctrl+,）配置 "editor.tabSize": 2 轻松覆盖。

如前所述，变量是可变的，即:它们的值可以（多次）改变，但变量的类型必须与声明时保持一致。

```
var f = 7           ①

f = -3              ②
f = 19
f = "Hello" # error ③ ④
```

① 变量 f 的初始值为 7 ，其类型被推断为 int 。

② 首先将 f 的值更改为 -3 ，然后再改为 19 。这两个值都是整数，与原值类型相同。

③ 尝试将 f 的值更改为 "Hello" 时会产生错误，因为 Hello 不是数字，这将导致 f 的类型从整数变为字符串。

④ # error 是注释。Nim代码中的注释写在 # 字符之后。同一行中该字符后的所有内容都会被忽略。

不可变赋值(IMMUTABLE ASSIGNMENT)

与使用 var 关键字声明的变量不同，Nim中还存在另外两种赋值类型，其值不可更改，一种是用 const 关键字声明的，另一种则是用 let 关键字声明的。

常量(Const)

使用 const 关键字声明的不可变赋值，其值必须在编译时（程序运行前）已知。

例如，我们可以将重力加速度声明为 const g = 9.81 或将圆周率声明为 const pi = 3.14 ，因为我们预先知道它们的值，并且这些值在程序执行过程中不会改变。

```
const g = 35
g = -27         # error ①

var h = -5
const i = h + 7 # error ②
```

① 常量的值不可更改。

② 变量 h 在编译时不会被求值（它是一个变量，其值在程序执行过程中可能改变），因此常量 i 的值在编译时无法确定，这将引发错误。

在某些编程语言中，常量的名称通常以 ALL_CAPS 书写。 而在Nim中，常量的书写方式与其他变量无异。

Let

使用 let 声明的不可变赋值不需要在编译时已知，它们的值可以在程序执行的任何时间设置，但一旦设置，它们的值就不能更改。

```
let j = 35
j = -27 # error ①

var k = -5
let l = k + 7   ②
```

① 不可变对象的值不能被改变。

② 与上面的 const 示例不同，这个可以工作。

在实际应用中，你会更频繁地见到或使用 let 而非 const 。

虽然你可以用 var 处理所有情况，但默认应选择 let 。 仅对那些需要被修改的变量使用 var 。

基本数据类型(BASIC DATA TYPES)

整数 INTEGERS

如前一章所述，整数是没有小数部分且不带小数点的数字。

例如: `32` , `-174` , `0` , `10_000_000` 都是整数。注意，我们可以使用 `_` 作为千位分隔符，使较大的数字更易读（比如写成 `10_000_000` 比 `10000000` 更容易看出我们指的是1000万）。

常见的数学运算符——加法（`+`）、减法（`-`）、乘法（`*`）和除法（`/`）——其行为与预期一致。前三种运算总是产生整数结果，而两个整数相除总会得到一个浮点数（带小数点的数），即使两数能够整除也不例外。

整数除法（丢弃小数部分的除法）可以通过 `div` 运算符实现。如果对整数除法的余数（模数）感兴趣，可以使用 `mod` 运算符。 这两个运算的结果始终是整数。

Listing 2. integers.nim

```
let
  a = 11
  b = 4

echo "a + b = ", a + b ①
echo "a - b = ", a - b
echo "a * b = ", a * b
echo "a / b = ", a / b
```

```
echo "a div b = ", a div b
echo "a mod b = ", a mod b
```

① echo 命令会将紧随其后由逗号分隔的所有内容打印到屏幕上。在本例中，它首先打印字符串 a + b =，然后紧接着在同一行打印表达式 a + b 的结果。

我们可以编译并运行上述代码，输出结果应为:

```
a + b = 15
a - b = 7
a * b = 44
a / b = 2.75
a div b = 2
a mod b = 3
```

浮点数 FLOATS

浮点数，简称 floats，是实数的一种 近似表示 。

例如： `2.73` , `-3.14` , `5.0` , `4e7` 是浮点数。

注意，对于较大的浮点数，我们可以使用科学计数法，其中 e 后面的数字是指数。在这个例子中， `4e7` 是一种表示 `4*10^7` 的记法。

我们也可以在两个浮点数之间使用四种基本数学运算。运算符 div 和 mod 未针对浮点数定义。

Listing 3. floats.nim

```
let
  c = 6.75
  d = 2.25

echo "c + d = ", c + d
echo "c - d = ", c - d
echo "c * d = ", c * d
echo "c / d = ", c / d
```

```
c + d = 9.0  ①
c - d = 4.5
c * d = 15.1875
c / d = 3.0  ①
```

① 请注意，在加法和除法的例子中，即使我们得到的数字没有小数部分，结果仍然是浮点类型。

数学运算的优先级符合预期：乘法和除法的优先级高于加法和减法。

```
echo 2 + 3 * 4
echo 24 - 8 / 4
```

```
14
22.0
```

浮点数和整数的转换

在Nim中，不同数值类型的变量之间无法进行数学运算，否则会产生错误：

```
let
  e = 5
  f = 23.456

echo e + f   # error
```

变量的值需要转换为同一类型。 转换过程很简单：要转换为整数，我们使用 `int` 函数；要转换为浮点数，则使用 `float` 函数。

```
let
  e = 5
  f = 23.987

echo float(e)      ①
echo int(f)        ②

echo float(e) + f  ③
echo e + int(f)    ④
```

① 打印一个整数的 `float` 形式。（e 仍为整数类型）

② 打印浮点数f的 `int` 形式。

③ 两个操作数均为浮点数，可以进行加法运算。

④ 两个操作数都是整数，可以相加。

```
5.0
23
28.987
28
```

使用 `int` 函数将浮点数转换为整数时不会执行四舍五入操作，数字会直接舍弃小数部分。
要进行四舍五入运算，我们必须调用另一个函数，但在此之前需要先了解一些关于如何使用 Nim 的基础知识。

字符 CHARACTERS

char 类型用于表示单个 ASCII 字符。

字符写在两个单引号之间（ ' ）。 字符可以是字母、符号或单个数字。 多个数字或多个字母会导致错误。

```
let
  h = 'z'
  i = '+'
  j = '2'
  k = '35' # error
  l = 'xy' # error
```

字符串 STRINGS

字符串可以被描述为一系列字符。 其内容写在两个双引号之间（ " ）。

我们可能认为字符串就是单词，但它们可以包含多个单词、一些符号(译者注：可以是utf-8字符)或数字。

Listing 4. strings.nim

```
let
  m = "word"
  n = "A sentence with interpunction."
  o = ""    ①
  p = "32"  ②
  q = "!"   ③
```

① 一个空字符串

② 这不是数字（整数）。它在双引号内，因此是一个字符串。

③ 尽管这只是一个字符，但它不是 char 类型，因为它被包含在双引号内。

特殊字符 Special characters

如果我们尝试打印以下字符串：

```
echo "some\nim\tips"
```

结果可能会让我们感到惊讶：

```
some
im      ips
```

这是因为有几个字符具有特殊含义。 它们通过在字符前添加转义字符 \ 来使用。

- \n 是换行符
- \t 是一个制表符
- \\ 是一个反斜杠（因为一个 \ 被用作转义字符）

如果我们想按照原样打印上面的示例，那么有两种可能选择：

- 使用 \\ 而不是 \ 来打印反斜杠，或者:
- 使用原始字符串，其语法为 r"..." （在第一个引号前紧接字母 r ），其中没有转义字符，也没有特殊含义：所有内容都会原样输出。

```
echo "some\\nim\\tips"
echo r"some\nim\tips"
```

```
some\nim\tips
some\nim\tips
```

除了上面列出的特殊字符外，还有更多特殊字符，它们都可以在 Nim 手册中找到. (Nim manual)

字符串连接 String concatenation

Nim中的字符串是可变的，意味着它们的内容可以改变。 通过 add 函数，我们可以向现有字符串添加（或追加）另一个字符串或字符。如果我们不想改变原始字符串，也可以使用 & 运算符来连接（拼接）字符串，这会返回一个新的字符串。

Listing 5. stringConcat.nim

```
var                       ①
  p = "abc"
  q = "xy"
  r = 'z'

p.add("def")              ②
echo "p is now: ", p

q.add(r)                  ③
echo "q is now: ", q

echo "concat: ", p & q    ④

echo "p is still: ", p
echo "q is still: ", q
```

① 如果我们计划修改字符串，它们应该被声明为 var 。

② 添加另一个字符串会就地修改现有字符串 p ， 改变其值。

③ 我们也可以向字符串添加一个 char 。

④ 连接两个字符串会产生一个新的字符串，而不会修改原始字符串。

```
p is now: abcdef
q is now: xyz
concat: abcdefxyz
p is still: abcdef
q is still: xyz
```

布尔型 BOOLEAN

一个布尔型（Boolean）（或简称 bool ）数据类型只能有两个值：`true` 或 `false` 。布尔值通常用于控制流程（参见下一章），并且常常是关系运算符的运算结果。

The usual naming convention for boolean variables is to write them as a simple yes/no (true/false) question, e.g. `isEmpty`, `isFinished`, `isMoving`, etc. 布尔变量的命名惯例是将其写成简单的 yes/no（真/假）问题形式，例如 `isEmpty` 、 `isFinished` 、 `isMoving` 等。

关系运算符 Relational operators

关系运算符测试两个实体之间的关系，这两个实体必须是可比较的。

要比较两个值是否相同，使用 `==` （两个等号）。 不要将其与 `=` 混淆，后者(即“=”)如我们之前所见用于赋值。

以下是针对整数定义的所有关系运算符:

Listing 6. relationalOperators.nim

```
let
  g = 31
  h = 99
```

```
echo "g is greater than h: ", g > h
echo "g is smaller than h: ", g < h
echo "g is equal to h: ", g == h
echo "g is not equal to h: ", g != h
echo "g is greater or equal to h: ", g >= h
echo "g is smaller or equal to h: ", g <= h
```

```
g is greater than h: false
g is smaller than h: true
g is equal to h: false
g is not equal to h: true
g is greater or equal to h: false
g is smaller or equal to h: true
```

我们也可以比较字符和字符串：

Listing 7. relationalOperators.nim

```
let
  i = 'a'
  j = 'd'
  k = 'Z'

echo i < j
echo i < k  ①

let
  m = "axyb"
  n = "axyz"
  o = "ba"
  p = "ba "

echo m < n  ②
echo n < o  ③
echo o < p  ④
```

① 所有大写（译者注：应当是ASCII中的英文字符）字母都排在小写字母之前。

② 字符串比较是按字符逐个进行的。前三个字符相同，字符 `b` 小于字符 `z` 。

③ 如果字符串的字符不完全相同，比较时长度并不重要。

④ 较短的字符串比长的字符串小。

```
true
false
true
true
true
```

逻辑运算符 Logical operators

Logical operators are used to test the truthiness of an expression consisting of one or more boolean values.

逻辑运算符用于测试由一个或多个布尔值组成的表达式的真值。

- 逻辑 `and` 仅在两个成员都为 `true` 时返回 `true`
- 逻辑 `or` 在至少有一个成员为 `true` 时返回 `true`
- 逻辑 `xor` 在一个成员为真而另一个为假时返回 `true`
- 逻辑运算符 `not` 会反转其操作数的真值：将 `true` 变为 `false`，反之亦然（它是唯一一个只需要一个操作数的逻辑运算符）

Listing 8. logicalOperators.nim

```
echo "T and T: ", true and true
echo "T and F: ", true and false
echo "F and F: ", false and false
echo "---"
echo "T or T: ", true or true
echo "T or F: ", true or false
echo "F or F: ", false or false
echo "---"
echo "T xor T: ", true xor true
echo "T xor F: ", true xor false
echo "F xor F: ", false xor false
```

```
echo "---"
echo "not T: ", not true
echo "not F: ", not false
```

```
T and T: true
T and F: false
F and F: false
---
T or T: true
T or F: true
F or F: false
---
T xor T: false
T xor F: true
F xor F: false
---
not T: false
not F: true
```

关系运算符和逻辑运算符可以组合在一起形成更复杂的表达式.

例如: `(5 < 7) and (11 + 9 == 32 - 2*6)` 将变为 `true and (20 == 20)`, 接着变为 `true and true`, 最终得到的结果是 `true` 。

回顾

这是本教程中最长的一章，我们涵盖了大量内容。请花时间逐一了解每种数据类型，并尝试用它们进行各种操作实验。

这些类型(types)初看可能像是一种限制，但它们能让 Nim 编译器既提升代码运行速度，又确保你不会无意中犯错——这对大型代码库尤为有益。

现在你已经了解了基本数据类型及其相关操作，这些知识足以在Nim中进行一些简单的计算。 通过完成以下练习来测试你的掌握程度。

练习

1. 创建一个不可变变量，包含你的年龄（以年为单位）。打印你的年龄对应的天数。（1 年 = 365 天）
2. 检查你的年龄是否能被 3 整除。（提示：使用 `mod` ）
3. 创建一个包含你身高（厘米）的不可变变量，并以英寸为单位打印你的身高。（1 英寸=2.54 厘米）
4. 一根管子的直径为 3/8 英寸。请用厘米表示该直径。
5. 创建一个不可变变量存储你的名字，再创建另一个存储你的姓氏。通过拼接前两个变量生成变量 `fullName` ，注意中间要加一个空格。打印你的全名。
6. 爱丽丝(Alice)每 15 天赚 400 美元。鲍勃每小时赚 3.14 美元，每天工作 8 小时，每周工作 7 天。30 天后，爱丽丝赚的钱比鲍勃多吗？（提示：使用关系运算符）

控制流 CONTROL FLOW

到目前为止，在我们的程序中，每一行代码都会在某个时刻执行。控制流语句允许我们编写仅在满足某些布尔条件时才会执行的代码部分。

If we think of our program as a road we can think of control flow as various branches, and we pick our path depending on some condition. For example, we will buy eggs only *if* their price is less than some value. 如果我们将程序想象成一条道路，那么控制流就像是各种分支，我们根据某些条件来选择路径。例如，只有当鸡蛋的价格低于某个值时，我们才会购买。*或者*，如果下雨了，我们会带伞，否则（else）我们会带太阳镜。

用伪代码(pseudocode)表示，这两个例子看起来是这样的：

```
if eggPrice < wantedPrice:
  buyEggs

if isRaining:
  bring umbrella
else:
  bring sunglasses
```

Nim的语法非常相似，如下所示。

IF 语句

如上所示的 if 语句是分支程序的最简单方式。

编写 if 语句的 Nim 语法是：

```
if <condition>:     ①
  <indented block>  ②
```

① condition（即“某种情况或情形”） 必须是布尔类型：可以是布尔变量，也可以是关系和/或逻辑表达式。

② 在 `if` 行之后所有缩进两个空格的代码行构成同一个代码块，只有当条件为 `true` 时才会执行。

if 语句可以嵌套，即在一个 if 代码块内部可以包含另一个 if 语句。

Listing 9. if.nim

```
let
  a = 11
  b = 22
  c = 999

if a < b:
  echo "a is smaller than b"
  if 10*a < b:  ①
    echo "not only that, a is *much* smaller than b"

if b < c:
  echo "b is smaller than c"
  if 10*b < c:  ②
```

```
    echo "not only that, b is *much* smaller than c"

if a+b > c:     ③
  echo "a and b are larger than c"
  if 1 < 100 and 321 > 123:  ④
    echo "did you know that 1 is smaller than 100?"
    echo "and 321 is larger than 123! wow!"
```

① 第一个条件为真，第二个为假——内部 echo 未执行。

② 两个条件均为真，两行内容都会被打印。

③ 第一个条件为假——其代码块内的所有行都将被跳过，不会打印任何内容。

④ 在 if 语句中使用逻辑 and 。

```
a is smaller than b
b is smaller than c
not only that, b is *much* smaller than c
```

Else (否则)

Else 跟在 if 代码块之后，允许我们编写一段在 if 语句条件不成立时执行的代码分支。

Listing 10. else.nim

```
let
  d = 63
  e = 2.718

if d < 10:
  echo "d is a small number"
else:
  echo "d is a large number"

if e < 10:
  echo "e is a small number"
else:
  echo "e is a large number"
```

```
d is a large number
e is a small number
```

如果你只想在语句为 `false` 时执行一个代码块，可以简单地用 `not` 运算符对条件取反。

Elif

Elif 是"else if"的缩写，它使我们能够将多个 if 语句串联在一起。

程序测试每一个语句，直到找到一个为真的语句。之后，所有后续语句都会被忽略。

Listing 11. elif.nim

```
let
  f = 3456
  g = 7

if f < 10:
  echo "f is smaller than 10"
elif f < 100:
  echo "f is between 10 and 100"
elif f < 1000:
  echo "f is between 100 and 1000"
else:
  echo "f is larger than 1000"

if g < 1000:
  echo "g is smaller than 1000"
elif g < 100:
  echo "g is smaller than 100"
elif g < 10:
  echo "g is smaller than 10"
```

```
f is larger than 1000
g is smaller than 1000
```

在 g 的情况下，即使 g 满足所有三个条件，也只会执行第一个分支，自动跳过所有其他分支。

CASE（情况）

case 语句是另一种仅选择多个可能路径之一的方式，类似于带有多个 elif 的 if 语句。 然而， case 语句不接受多个布尔条件，而是接受具有不同状态的任何值，并为每个可能的值提供一个路径。

使用 if-elif 块编写的代码看起来像这样：

```
if x == 5:
  echo "Five!"
elif x == 7:
  echo "Seven!"
elif x == 10:
  echo "Ten!"
else:
  echo "unknown number"
```

可以使用 case 语句这样写:

```
case x
of 5:
  echo "Five!"
of 7:
  echo "Seven!"
of 10:
  echo "Ten!"
else:
  echo "unknown number"
```

与 if 语句不同，case 语句必须覆盖所有可能的情况。如果对某些情况不感兴趣，可以使用 `else: discard`.

Listing 12. case.nim

```
let h = 'y'

case h
of 'x':
  echo "You've chosen x"
of 'y':
  echo "You've chosen y"
of 'z':
  echo "You've chosen z"
else: discard  ①
```

① 尽管我们只对 h 的三个值感兴趣，但必须包含这一行以涵盖所有其他可能的情况（所有其他字符）。没有它，代码将无法编译。

```
You've chosen y
```

如果多个值需要执行相同的操作，我们也可以为每个分支使用多个值。

Listing 13. multipleCase.nim

```
let i = 7

case i
  of 0:
    echo "i is zero"
  of 1, 3, 5, 7, 9:
    echo "i is odd"
  of 2, 4, 6, 8:
    echo "i is even"
  else:
    echo "i is too large"
```

```
i is odd
```

循环 LOOPS

循环是另一种控制流结构，它允许我们多次运行某些代码部分。

在本章中，我们将介绍两种循环类型：

- for循环(for-loop)：运行已知次数
- while-loop: 只要满足某些条件就会一直运行

FOR循环(FOR LOOP)

for 循环的语法是:

```
for <loopVariable> in <iterable>:
  <loop body>
```

传统上， `i` 常被用作 `loopVariable` 名称，但也可以使用其他任何名称。该变量仅在循环内部可用。一旦循环完成，变量的值就会被丢弃。

`iterable` 是我们可以遍历的任何对象。在已经提到的类型中，字符串是可迭代对象。（更多可迭代类型将在下一章介绍。）

循环体(loop body) 中的所有行在每次循环时都会执行，这使我们能够高效地编写重复的代码部分。

如果要在Nim中遍历一个（整数）数字范围，语法是 `start .. finish` ，其中 `start` 和 `finish` 是数字。 这将遍历 `start` 和 `finish` 之间的所有数字，包括 `start` 和 `finish` 。对于默认的范围可迭代对象， `start` 需要小于 `finish` 。

如果我们想迭代到一个数字（不包括该数字），可以使用 `..<` :

Listing 14. for1.nim

```
for n in 5 .. 9:  ①
  echo n

echo ""

for n in 5 ..< 9: ②
  echo n
```

① 使用 `..` 遍历数字范围——范围的两端都包含在内。

② 使用 `..<` 遍历相同的范围——它会迭代到上限但不包括上限。

```
5
6
7
8
9

5
6
7
8
```

如果我们想以不同于1的步长(step size)遍历数字范围，可以使用 `countup` 。通过 `countup` 我们可以定义起始值、终止值（包含在范围内）以及步长。

Listing 15. for2.nim

```
for n in countup(0, 16, 4):  ①
  echo n
```

① 从0开始以步长4向上计数至16（包含16）。

```
0
4
8
```

```
12
16
```

要遍历一个范围，其中 `start` 大于 `finish` ，可以使用一个名为 `countdown` 的类似函数。 即使我们在倒计数，步长也必须是正数。

Listing 16. for2.nim

```
for n in countdown(4, 0):       ①
  echo n

echo ""

for n in countdown(-3, -9, 2):  ②
  echo n
```

① To iterate from a higher to a lower number, we must use `countdown` (The `..` operator can only be used when the starting value is smaller than the end value). 要从高到低进行迭代，我们必须使用 `countdown` （ `..` 运算符只能在起始值小于结束值时使用）。

② 即使在"倒计时"(counting down)时，步长也必须是一个正数。

```
4
3
2
1
0

-3
-5
-7
-9
```

由于字符串是可迭代的，我们可以使用 `for` 循环来遍历字符串中的每个字符（这种迭代有时称为 `for-each` 循环）。

Listing 17. for3.nim

```
let word = "alphabet"

for letter in word:
  echo letter
```

```
a
l
p
h
a
b
e
t
```

如果还需要一个从零开始的迭代计数器，我们可以使用 `for <counterVariable>, <loopVariable> in <iterator>:` 语法来实现。这在需要遍历一个可迭代对象，同时按相同偏移量访问另一个可迭代对象时非常实用。

Listing 18. for3.nim

```
for i, letter in word:
  echo "letter ", i, " is: ", letter
```

```
letter 0 is: a
letter 1 is: l
letter 2 is: p
letter 3 is: h
letter 4 is: a
letter 5 is: b
letter 6 is: e
letter 7 is: t
```

WHILE循环 WHILE LOOP

while循环与if语句类似，但只要条件保持为真，它们就会持续执行其代码块。当我们无法预先知道循环将运行多少次时，就会使用它们。

我们必须确保循环在某个时刻终止，而不会变成无限循环(infinite loop.)

Listing 19. while.nim

```
var a = 1

while a*a < 10: ①
  echo "a is: ", a
  inc a         ②

echo "final value of a: ", a
```

① 每次进入新循环并执行其中的代码之前，都会检查此条件。

② `inc` 用于将 `a` 加一。这与写成 `a = a + 1` 或 `a += 1` 是相同的。

```
a is: 1
a is: 2
a is: 3
final value of a: 4
```

中断（BREAK）与 继续（CONTINUE）

break 语句用于提前退出循环，通常在满足某些条件时使用。

在下面的示例中，如果没有包含 break 的 if 语句，循环将持续运行并打印直到 i 达到 1000 。而通过 break 语句，当 i 变为 3 时，我们会立即退出循环（此时尚未打印 i 的值）。

Listing 20. break.nim

```
var i = 1

while i < 1000:
  if i == 3:
    break
  echo i
  inc i
```

```
1
2
```

continue 语句会立即开始循环的下一次迭代，而不会执行当前迭代的剩余代码。注意以下代码的输出中缺少了 3 和 6 ：

Listing 21. continue.nim

```
for i in 1 .. 8:
  if (i == 3) or (i == 6):
    continue
  echo i
```

```
1
2
4
5
7
8
```

练习

---

1. 科拉兹猜想（Collatz conjecture） 是一个规则简单却广受欢迎的数学问题。首先选择一个数字，若为奇数则乘以三加一；若为偶数则除以二。重复此过程直至结果为1。例如：5 → 奇数 → 3*5+1=16 → 偶数 → 16/2=8 → 偶数 → 4 → 2 → 1 → 终止!
   选取一个整数（作为可变变量），编写循环打印科拉兹猜想的每一步骤。（提示:使用 `div` 表示除法）
2. 创建一个包含你全名的不可变变量。编写一个 `for` 循环，遍历该字符串并仅打印元音字母（a、e、i、o、u）。（提示：使用 `case` 语句，每个分支包含多个值）
3. Fizz buzz 是一种儿童游戏，有时用于测试基础编程能力。我们从1开始向上计数。如果一个数字能被3整除，就用fizz替代；能被5整除则用buzz替代；若同时能被15（即3和5）整除，则替换为fizzbuzz。前几轮的游戏序列如下:1,2,fizz,4,buzz,fizz, 7,...
   编写一个程序，输出 Fizz buzz 游戏的前 30 轮结果。（提示：注意整除测试的顺序）
4. 在之前的练习中，你已经将英寸转换为厘米，反之亦然。创建一个包含多个值的转换表。例如，表格可能如下所示：

```
in      | cm
----------------
```

```
1       | 2.54
4       | 10.16
7       | 17.78
10      | 25.4
13      | 33.02
16      | 40.64
19      | 48.26
```

容器 CONTAINERS

容器是一种数据类型，它包含一组元素并允许我们访问这些元素。通常容器也是可迭代的，这意味着我们可以像在循环 Loops中使用字符串那样使用它们。

例如，购物清单是我们想购买的物品的容器，而质数列表则是数字的容器。用伪代码表示：

```
groceryList = [ham, eggs, bread, apples]
primes = [1, 2, 3, 5, 7]
```

数组 ARRAYS

数组是最简单的容器类型。数组是同质的，即数组中的所有元素必须具有相同的类型。数组的大小也是固定的，意味着元素的数量（或者更准确地说：可能的元素数量）必须在编译时已知。这意味着我们称数组为“固定长度的同质容器”。

数组类型使用 `array[<length>, <type>]` 声明，其中 `length` 是数组的总容量（可容纳的元素数量）， `type` 是其所有元素的类型。如果长度和类型可以从传递的元素中推断出来，则可以省略声明。

数组的元素被包含在方括号内。

```
var
  a: array[3, int] = [5, 7, 9]
  b = [5, 7, 9]         ①
  c = []  # error       ②
  d: array[7, string]   ③
```

① 如果我们提供了值，数组 b 的长度和类型在编译时就是已知的。虽然正确，但无需像数组 a 那样特别声明。

② 这种声明方式既无法推断元素的长度，也无法推断其类型——这会导致错误。

③ 正确声明空数组（稍后将填充）的方法是给出其长度和类型，而不提供元素值——数组 d 可以包含七个字符串。

由于数组的长度必须在编译时已知，因此以下代码将无法工作：

```
const m = 3
let n = 5

var a: array[m, char]
var b: array[n, char] # error ①
```

① 这会产生错误，因为 n 是用 let 声明的——它的值在编译时是未知的。我们只能使用用 const 声明的值作为数组初始化的 length 参数。

序列 SEQUENCES

序列（Sequences）是一种类似于数组的容器，但其长度无需在编译时已知，且可在运行时动态变化：我们仅需通过 `seq[<type>]` 声明所包含元素的类型。序列也是同质的（homogeneous），即序列中的每个元素必须具有相同的类型。

序列的元素被包含在 `@[` 和 `]` 之间。

```
var
  e1: seq[int] = @[]   ①
  f = @["abc", "def"]  ②
```

① 空序列的类型必须被声明。

② 非空序列的类型可以被推断出来。在这个例子中，它是一个包含字符串的序列。

另一种初始化空序列的方法是调用 `newSeq` 过程。我们将在下一章节更详细地介绍过程调用，但现在只需知道这也是一种可行方式：

```
var
  e = newSeq[int]() ①
```

① 在方括号内提供类型参数可以让过程知道它应该返回某种特定类型的序列。
一个常见的错误是遗漏了在末尾的 `()`，这是必需的。

我们可以使用 `add` 函数向序列中添加新元素，这与处理字符串的方式类似。为此，序列必须是可变的（用 `var` 定义），并且我们添加的元

素必须与序列中的元素类型相同。

Listing 22. seq.nim

```
var
  g = @['x', 'y']
  h = @['1', '2', '3']

g.add('z')  ①
echo g

h.add(g)    ②
echo h
```

① 添加一个相同类型（char）的新元素。

② 添加另一个包含相同类型的序列。

```
@['x', 'y', 'z']
@['1', '2', '3', 'x', 'y', 'z']
```

尝试将不同类型传递给现有序列会产生错误：

```
var i = @[9, 8, 7]

i.add(9.81) # error ①
g.add(i)    # error ②
```

① 尝试将 `float` 添加到 `int` 的序列中。

② 尝试将序列 `int` 添加到序列 `char` 中

由于序列长度可变，我们需要一种获取其长度的方法，为此可以使用 `len` 函数。

```
var i = @[9, 8, 7]
echo i.len
```

```
i.add(6)
echo i.len
```

```
3
4
```

索引 INDEXING 与 切片 SLICING

索引允许我们通过下标从容器中获取特定元素。 可以将索引视为容器内部的位置标识。

Nim与许多编程语言一样采用零基索引，即容器中第一个元素的索引为零，第二个元素的索引为一，以此类推。

若要从后向前索引，需使用 ^ 前缀。最后一个元素（即倒数第一个）的索引是 ^1 。

索引的语法是 `<container>[<index>]` 。

Listing 23. indexing.nim

```
let j = ['a', 'b', 'c', 'd', 'e']

echo j[1]   ①
echo j[^1]  ②
```

① 零基索引：索引 1 对应的元素是 `b` 。

② 获取最后一个元素。

```
b
e
```

切片操作允许我们通过一次调用获取一系列元素。它使用的语法与范围（在 for循环(For loop) 中介绍过）相同。

若使用 `start .. stop` 语法，切片范围包含两端。使用 `start ..< stop` 语法时， `stop` 索引不包含在切片内。

The syntax for slicing is `<container>[<start> .. <stop>]`.切片的语法是 `<container>[<start> .. <stop>]` 。

Listing 24. indexing.nim

```
echo j[0 .. 3]
echo j[0 ..< 3]
```

```
@[a, b, c, d]
@[a, b, c]
```

索引和切片均可用于为现有的可变容器和字符串赋予新值。

Listing 25. assign.nim

```
var
  k: array[5, int]
  l = @['p', 'w', 'r']
  m = "Tom and Jerry"

for i in 0 .. 4:  ①
  k[i] = 7 * i
echo k

l[1] = 'q'        ②
echo l

m[8 .. 9] = "Ba"  ③
echo m
```

① 长度为 5 的数组索引从零到四。我们将为数组的每个元素赋值。

② 修改序列中第二个元素（索引为 1）的值。

③ 修改字符串中索引为 8 和 9 位置的字符。

```
[0, 7, 14, 21, 28]
@['p', 'q', 'r']
Tom and Barry
```

元组 TUPLES

我们目前所见到的容器都是同质的。 而元组则包含异质数据，即元组中的元素可以是不同类型。与数组类似，元组具有固定大小。

元组的元素被包裹在圆括号内。

Listing 26. tuples.nim

```
let n = ("Banana", 2, 'c')  ①
echo n
```

① 元组可以包含不同类型的字段。在这个例子中： `string` 、 `int` 和 `char` 。

```
(Field0: "Banana", Field1: 2, Field2: 'c')
```

我们还可以为元组中的每个字段命名以便区分。这种方式可用于访问元组元素，而无需使用索引。

Listing 27. tuples.nim

```
var o = (name: "Banana", weight: 2, rating: 'c')

o[1] = 7          ①
o.name = "Apple"  ②
echo o
```

① 通过字段索引修改字段值

② 通过字段名称修改字段值

```
(name: "Apple", weight: 7, rating: 'c')
```

练习

1. 创建一个能容纳十个整数的空数组。
   - 用数字10、20、……、100填充该数组。（提示：使用循环）
   - 仅打印该数组中位于奇数索引的元素（即数值20、40、……）。
   - 将偶数索引位置的元素乘以 5。打印修改后的数组。
2. 重新实现考拉兹猜想（Collatz conjecture exercise）练习，但这次不打印每个步骤，而是将其添加到序列中。
   - 选择一个起始数字。有趣的选择包括9、19、25和27等。
   - 创建一个仅包含该起始数字的序列
   - 沿用之前的逻辑，持续向序列中添加元素，直到数值变为 1
   - 输出序列的长度及序列本身
3. 在2到100的范围内找出产生最长考拉兹序列的数字。
   - 对于给定范围内的每个数字，计算其考拉兹序列
   - 如果当前序列长度超过之前的记录，则将当前长度和起始数字保存为新记录（可以使用元组（`longestLength, startingNumber`）或两个单独的变量）

- 输出起始数字及其对应序列的最大长度

过程 PROCEDURES

过程（在其他编程语言中常称为函数）是将执行特定任务的代码封装为一个独立单元。 这种代码组织方式的优势在于，我们可以直接调用这些过程，而无需在每次需要时重复编写相同的代码。

在前面的某些章节中，我们探讨了不同场景下的考拉兹猜想。通过将考拉兹猜想逻辑封装成过程，我们就能在所有练习中调用相同的代码。

到目前为止，我们已经使用了许多内置过程，例如用于打印的 `echo` 、向序列添加元素的 `add` 、增加整数值的 `inc` 、获取容器长度的 `len` 等。现在我们将学习如何创建和使用自己的过程。

使用过程（procedure）的一些优势包括：

- 减少代码重复
- 通过为代码片段赋予功能描述性名称，使代码更易阅读
- 将复杂任务分解为更简单的步骤

正如本节开头所述，在其他语言中过程(procedure)常被称为函数(function)。若从数学函数的定义来看，这其实是个不太准确的称

谓。数学函数接收一组参数（如 f(x, y)，其中 f 是函数， x 和 y 是其参数），对于相同的输入总是返回相同的结果。

另一方面，程序化过程对于给定输入并不总是返回相同输出。有时它们根本不返回任何内容。这是因为我们的计算机程序可以在之前提到的变量中存储状态，这些过程可以读取和更改状态。在Nim中，当前保留使用 func 这个词来表示数学上更准确的函数类型，强制要求无副作用。

声明过程 DECLARING A PROCEDURE

在使用（调用）过程之前，我们需要创建它并定义其功能。

过程通过使用 proc 关键字和过程名称来声明，后跟括号内的输入参数及其类型，最后部分是一个冒号和过程返回值的类型，如下所示：

```
proc <name>(<p1>: <type1>, <p2>: <type2>, ...): <returnType>
```

过程体以缩进块的形式编写，紧跟在带有 = 符号的声明之后。

Listing 28. callProcs.nim

```
proc findMax(x: int, y: int): int =  ①
  if x > y:
    return x  ②
  else:
    return y
  # this is inside of the procedure
# this is outside of the procedure
```

① 声明名为 findMax 的过程，它有两个参数 x 和 y ，并返回一个 int 类型。

② 要从过程中返回值，我们使用 return 关键字。

```
proc echoLanguageRating(language: string) = ①
  case language
```

```
  of "Nim", "nim", "NIM":
    echo language, " is the best language!"
  else:
    echo language, " might be a second-best language."
```

① echoLanguageRating 过程只是回显给定的名称，它不返回任何内容，因此没有声明返回类型。

通常情况下，我们不允许修改传入的任何参数。类似这样的操作会引发错误：

```
proc changeArgument(argument: int) =
  argument += 5

var ourVariable = 10
changeArgument(ourVariable)
```

为了让这能够正常工作，我们需要允许Nim以及使用我们过程的程序员通过将参数声明为变量来修改它：

```
proc changeArgument(argument: var int) = ①
  argument += 5

var ourVariable = 10
changeArgument(ourVariable)
echo ourVariable
changeArgument(ourVariable)
echo ourVariable
```

① 注意 argument 现在被声明为 var int ，而不仅仅是 int 。

```
15
20
```

这当然意味着我们传入的名称也必须声明为变量，传入用 `const` 或 `let` 赋值的内容会引发错误。

虽然将内容作为参数传递是良好的实践，但也可以使用过程外部声明的名称，包括变量和常量：

```
var x = 100

proc echoX() =
  echo x  ①
  x += 1  ②

echoX()
echoX()
```

① 这里我们访问外部变量 `x` 。

② 我们可以更新它的值，因为它被声明为变量。

```
100
101
```

调用程序 CALLING THE PROCEDURES

声明过程后，我们可以调用它。在许多编程语言中，调用过程/函数的常规方式是声明其名称并在括号内提供参数，如下所示：

```
<procName>(<arg1>, <arg2>, ...)
```

调用过程的结果可以存储在变量中。

如果要从上面的示例中调用我们的 findMax 过程，并将返回值保存在变量中，我们可以这样做：

Listing 29. callProcs.nim

```
let
  a = findMax(987, 789)
  b = findMax(123, 321)
  c = findMax(a, b)  ①

echo a
echo b
echo c
```

① 函数 `findMax` 的结果在此处命名为 `c` ，并使用前两次调用(`findMax(987, 321)`)的结果进行调用。

```
987
321
```

```
987
```

与其他许多语言不同，Nim 还支持统一函数调用(Uniform Function Call Syntax)语法，这允许以多种不同方式调用过程。

这是一种调用方式，其中第一个参数写在函数名之前，其余参数则在括号内声明：

```
<arg1>.<procName>(<arg2>, ...)
```

我们在向现有序列添加元素时使用了这种语法（ `<seq>.add(<element>)` ），因为它比书写 `add(<seq>, <element>)` 更具可读性且能更清晰地表达意图。参数周围的括号也可以省略：

```
<procName> <arg1>, <arg2>, ...
```

我们在调用 `echo` 过程时，以及无参数调用 `len` 过程时，已经见过这种风格的用法。这两种情况也可以像这样组合使用，不过这种语法并不常见：

```
<arg1>.<procName> <arg2>, <arg3>, ...
```

统一调用语法使得多个过程的链式调用更具可读性：

Listing 30. ufcs.nim

```
proc plus(x, y: int): int =  ①
  return x + y
```

```
proc multi(x, y: int): int =
  return x * y

let
  a = 2
  b = 3
  c = 4

echo a.plus(b) == plus(a, b)
echo c.multi(a) == multi(c, a)


echo a.plus(b).multi(c)  ②
echo c.multi(b).plus(a)  ③
```

① 若多个参数属于同一类型，我们可以采用这种简洁的方式进行类型声明。

② 首先将 a 与 b 相加，然后将运算结果（2 + 3 = 5）作为第一个参数传递给 multi 过程，在该过程中与 c 相乘（5 * 4 = 20）。

③ 首先将 c 与 b 相乘，然后将运算结果（4*3=12）作为第一个参数传递给 plus 过程，在该过程中与 a 相加（12+2=14）。

```
true
true
20
14
```

结果变量 RESULT VARIABLE

在Nim语言中，每个返回值的过程都会隐式声明并初始化（使用默认值）一个 `result` 变量。当过程执行到其缩进代码块末尾时，即使没有 `return` 语句，也会返回这个 `result` 变量的值。

Listing 31. result.nim

```
proc findBiggest(a: seq[int]): int =  ①
  for number in a:
    if number > result:
      result = number
  # the end of proc                   ②

let d = @[3, -5, 11, 33, 7, -15]
echo findBiggest(d)
```

① 返回类型是 `int` 。 `result` 变量会被初始化为 `int` 的默认值： `0` 。

② 当过程执行结束时，将返回 `result` 的值。

```
33
```

请注意，此处展示的过程是为了演示 `result` 变量，它并非完全正确：如果传入一个仅包含负数的序列，该过程将返回 `0` （而该值并不存在于序列中）。

注意！ 在旧版 Nim（0.19.0 之前）中，字符串和序列的默认值为 `nil`，当我们将它们用作返回类型时， `result` 变量需要初始化为空字符串（ `""` ）或空序列（ `@[]` ）。

Listing 32. result.nim

```
proc keepOdds(a: seq[int]): seq[int] =
  # result = @[]          ①
  for number in a:
    if number mod 2 == 1:
      result.add(number)


let f = @[1, 6, 4, 43, 57, 34, 98]
echo keepOdds(f)
```

① 在Nim-0.19.0及更新版本中，无需这行代码 —— 序列会自动初始化为空序列。
在旧版Nim中，序列必须手动初始化，缺少这行代码会导致编译器报错。（注意不应使用 `var` ，因为 `result` 已被隐式声明。）

```
@[1, 43, 57]
```

在过程内部我们也可以调用其他过程。

Listing 33. filterOdds.nim

```
proc isDivisibleBy3(x: int): bool =
  return x mod 3 == 0

proc filterMultiplesOf3(a: seq[int]): seq[int] =
  # result = @[]            ①
  for i in a:
    if i.isDivisibleBy3():  ②
      result.add(i)
```

```
let
  g = @[2, 6, 5, 7, 9, 0, 5, 3]
  h = @[5, 4, 3, 2, 1]
  i = @[626, 45390, 3219, 4210, 4126]

echo filterMultiplesOf3(g)
echo h.filterMultiplesOf3()
echo filterMultiplesOf3 i   ③
```

① 新版本的 Nim 中不再需要这行代码。

② 调用先前声明的过程。其返回类型为 `bool`，可在 `if` 语句中使用。

③ 第三种调用过程的方式，如上文所示。

```
@[6, 9, 0, 3]
@[3]
@[45390, 3219]
```

前置声明 FORWARD DECLARATION

正如本节开头所述，我们可以声明不带代码块的过程。 原因在于必须先声明过程才能调用它们，以下做法是无效的：

```
echo 5.plus(10) # error      ①

proc plus(x, y: int): int =  ②
  return x + y
```

① 这会抛出错误，因为 `plus` 尚未定义。

② 这里我们定义了 `plus` ，但由于它出现在使用之后，Nim此时还不知道它的存在。

解决这个问题的方法称为前置声明：

```
proc plus(x, y: int): int    ①

echo 5.plus(10)              ②

proc plus(x, y: int): int =  ③
  return x + y
```

① 这里我们告诉Nim它应该认为 `plus` 过程是以这个定义存在的。

② 现在我们可以在代码中自由使用它，这将正常工作。

③ 这里实际实现了 `plus` ，当然必须与我们之前的定义相匹配。

练习

1. 创建一个过程（procedure），用于根据提供的姓名向某人问候（打印"Hello <姓名>"）。创建一个姓名序列。使用已创建的过程向每个人问候。
2. 创建一个过程 `findMax3`，该过程将返回三个值中的最大值。
3. 二维平面中的点可以表示为 `tuple[x, y: float]`。编写一个过程，该过程接收两个点并返回一个新点，新点是这两个点的和（分别相加 x 坐标和 y 坐标）。
4. 创建两个过程 `tick` 和 `tock`，分别输出单词"tick"和"tock"。使用一个全局变量来记录它们运行的次数，并让它们相互调用直到计数器达到 20。预期输出是交替出现 20 行"tick"和"tock"。（提示：使用前向声明。）

如果程序进入无限循环，可以按 Ctrl+C 停止执行。

通过调用不同参数来测试所有过程。

模块 MODULES

到目前为止，我们使用的都是每次新建 Nim 文件时默认提供的功能。这些功能可以通过模块进行扩展，从而获得针对特定主题的更多功能。

最常用的Nim模块包括：

- `strutils`: 处理字符串时的附加功能
- `sequtils`: 处理序列时的附加功能
- `math`: 数学函数（对数、平方根等）、三角函数（正弦、余弦等）
- `times`: 测量和处理时间

但还有更多模块，无论是在所谓的标准库（standard library）中，还是在nimble包管理器（nimble package manager）中。

导入模块

如果我们想导入一个模块及其所有功能，只需在文件中添加 `import <moduleName>` 。通常这会在文件顶部完成，以便我们轻松查看代码使用了哪些内容。

Listing 34. stringutils.nim

```
import strutils       ①

let
  a = "My string with whitespace."
  b = '!'

echo a.split()        ②
echo a.toUpperAscii() ③
echo b.repeat(5)      ④
```

① 导入 strutils 模块.

② 使用 `strutils` 模块中的 `split` 功能。它可以将字符串分割成单词序列。

③ `toUpperAscii` 将所有 ASCII 字母转换为大写形式。

④ `repeat` 同样来自 strutils 模块，它能将单个字符或整个字符串重复指定次数。

```
@["My", "string", "with", "whitespace."]
MY STRING WITH WHITESPACE.
!!!!!
```

对于来自其他编程语言（尤其是 Python）的用户来说，Nim 中的导入方式可能看起来"不太对"。如果是这种情况，建议阅读 以下内容 。

Listing 35. maths.nim

```
import math                  ①

let
  c = 30.0 # degrees
  cRadians = c.degToRad()    ②

echo cRadians
echo sin(cRadians).round(2) ③

echo 2^5                     ④
```

① Importing math.

② 使用 degToRad 将度数转换为弧度。

③ sin 接收弧度值。我们将结果（同样来自 math 模块）四舍五入至最多两位小数。（否则结果会是：0.4999999999999999）

④ 数学模块还包含用于计算数字幂的 ^ 运算符。

```
0.5235987755982988
0.5
32
```

创建我们自己的

在项目中，我们经常会有大量代码，这时将其拆分成各自负责特定功能的模块就很有意义。 如果你在同一个文件夹中创建两个文件（假设命名为 `firstFile.nim` 和 `secondFile.nim` ），就可以将一个文件作为模块导入另一个文件中：

Listing 36. firstFile.nim

```
proc plus*(a, b: int): int = ①
  return a + b

proc minus(a, b: int): int = ②
  return a - b
```

① 注意 `plus` 过程现在名称后带有星号（ `*` ），这告诉 Nim 其他导入此文件的模块将能够使用该过程。

② 相比之下，导入此文件时将不会显示该内容。

Listing 37. secondFile.nim

```
import firstFile          ①

echo plus(5, 10)          ②
echo minus(10, 5) # error ③
```

① 这里我们导入 `firstFile.nim` ，无需在此处添加 `.nim` 扩展名。

② 这段代码能正常运行并输出 15 ，因为它在 `firstFile` 中声明且对我们可见。

③ 然而这会抛出错误，因为 `minus` 过程不可见，其名称后未带星号

如果你有超过这两个文件，可能需要将它们组织到一个子目录（或多个子目录）中。 采用如下目录结构：

```
.
├── myOtherSubdir
│   ├── fifthFile.nim
│   └── fourthFile.nim
├── mySubdir
│   └── thirdFile.nim
├── firstFile.nim
└── secondFile.nim
```

如果你想导入 `secondFile.nim` 中的所有其他文件，可以这样操作：

Listing 38. secondFile.nim

```
import firstFile
import mySubdir/thirdFile
import myOtherSubdir / [fourthFile, fifthFile]
```

与用户输入交互

利用目前介绍的内容（基本数据类型和容器、控制流、循环），我们已经能够编写不少简单程序了。

在本章中，我们将学习如何让程序更具交互性。为此，我们需要实现从文件读取数据或向用户请求输入的功能。

从文件读取

假设我们有一个名为 people.txt 的文本文件，它与 Nim 代码位于同一目录下。该文件内容如下：

Listing 39. people.txt

```
Alice A.
Bob B.
Carol C.
```

我们希望将该文件内容作为名称列表（序列）用于程序中。

Listing 40. readFromFile.nim

```
import strutils

let contents = readFile("people.txt") ①
echo contents

let people = contents.splitLines()    ②
echo people
```

① 要读取文件内容，我们使用 readFile 过程，并提供一个文件路径以供读取（如果文件与我们的 Nim 程序位于同一目录，仅提供文件名即可）。结果将是一个多行字符串。

② 要将多行字符串拆分为字符串序列（每个字符串包含单行的全部内容），我们使用来自 strutils 模块的 splitLines 。

```
Alice A.
Bob B.
Carol C.
```

```
                ①
@["Alice A.", "Bob B.", "Carol C.", ""] ②
```

① 原文件中存在一个最后的空行（末尾空行），此处也保留了该空行。

② 由于末尾的换行符，我们的序列比预期/想要的更长。

为了解决末尾换行符的问题，我们可以在读取文件后使用 strutils 中的 strip 过程。该过程的作用就是移除字符串首尾的所有所谓空白字符。空白字符泛指任何产生间隔的字符，包括换行符、空格、制表符等。

Listing 41. readFromFile2.nim

```
import strutils

let contents = readFile("people.txt").strip() ①
echo contents

let people = contents.splitLines()
echo people
```

① 使用 `strip` 可得到预期结果。

```
Alice A.
Bob B.
Carol C.
@["Alice A.", "Bob B.", "Carol C."]
```

读取用户输入

若要与用户交互，我们必须能够向其询问输入内容，随后处理并运用这些信息。我们需要通过向 `readLine` 过程传递 `stdin` 参数来从标准输入 standard input (stdin) 读取数据。

Listing 42. interaction1.nim

```
echo "Please enter your name:"
let name = readLine(stdin)  ①

echo "Hello ", name, ", nice to meet you!"
```

① `name` 的类型被推断为字符串。

```
Please enter your name:
        ①
```

① 等待用户输入。当我们输入姓名并按下 `Enter` 后，程序将继续执行。

```
Please enter your name:
Alice
Hello Alice, nice to meet you!
```

如果您使用的是旧版VS Code，则无法以常规方式运行（使用Ctrl+Alt+N），因为输出窗口不允许用户输入——您需要在终端中运行这些示例。
较新版本的VS Code不存在此类限制。

处理数字

从文件或用户输入读取的内容始终以字符串形式返回。若需使用数字，需将字符串转换为数字：我们再次使用 `strutils` 模块，通过 `parseInt` 转换为整数，或使用 `parseFloat` 转换为浮点数。

Listing 43. interaction2.nim

```
import strutils

echo "Please enter your year of birth:"
let yearOfBirth = readLine(stdin).parseInt() ①

let age = 2018 - yearOfBirth

echo "You are ", age, " years old."
```

① 将字符串转换为整数。这样编写时，我们默认用户会提供有效的整数。如果用户输入 `'79` 或 `ninety-three` 会发生什么？你可以自己试试看。

```
Please enter your year of birth:
1934
You are 84 years old.
```

如果我们有一个文件 `numbers.txt` 与Nim代码位于同一目录下，其内容如下：

Listing 44. numbers.txt

```
27.3
98.24
11.93
33.67
55.01
```

我们想要读取该文件并计算所提供数字的总和与平均值，可以这样做：

Listing 45. interaction3.nim

```
import strutils, sequtils, math        ①

let
  strNums = readFile("numbers.txt").strip().splitLines()  ②
  nums = strNums.map(parseFloat)       ③

let
  sumNums = sum(nums)                  ④
  average = sumNums / float(nums.len)  ⑤

echo sumNums
echo average
```

① 我们导入多个模块。 `strutils` 提供 `strip` 和 `splitLines` ， `sequtils` 提供 `map` ， `math` 提供 `sum` 。

② 我们移除最后的换行符，并将文本分割成字符串序列。

③ `map` 的工作原理是对容器中的每个元素应用一个过程（本例中是 `parseFloat` ）。换句话说，我们将每个字符串转换为浮点数，返回一个新的浮点数序列。

④ 使用 `math` 模块中的 `sum` 函数来计算序列中所有元素的总和。

⑤ 我们需要将序列长度转换为浮点数，因为 `sumNums` 是浮点类型。

```
226.15
45.23
```

练习

1. 向用户询问身高和体重，计算其 BMI 指数。反馈 BMI 数值及对应等级。
2. 重复考拉兹猜想（ Collatz conjecture exercise ）练习：编写程序让用户输入起始数字，打印生成的数列。
3. 让用户输入需要反转的字符串。创建一个接收字符串并返回反转版本的过程。例如用户输入 `Nim-lang` 时，该过程应返回 `gnal-miN` 。（提示：使用索引和 `countdown` ）

结语

是时候结束这篇教程了。希望它对您有所帮助，并让您在编程和/或 Nim 编程语言的学习上迈出了第一步。

这些只是基础知识，我们仅仅触及了皮毛，但应该足以让你编写简单程序并解决一些简单任务或谜题。Nim 还有更多强大功能，希望你能继续探索它的可能性。

下一步

如果你想继续学习Nim：

- Official Nim tutorial
- Nim by example

如果你想解决一些编程难题：

- Advent of Code: 每年12月发布的一系列趣味谜题。提供自 2015 年起的往期谜题存档。
- Project Euler: 主要是数学相关的任务。

编程愉快！

The source files are available on Github.