一、Hello World

这是第一个nim程序：

```
echo "What's your name?"
var name : string  = readLine (stdin)
echo "Hi,", name , "!"
```

保存为文件：helloWorld.nim

那么我们可以编译运行它：

```
nim compile --run helloWorld.nim
```

使用 --run 可以让nim在编译后自动执行文件，这可以缩写：

```
nim c -r helloWorld.nim
```

也可以带命令行参数：

```
nim c -r helloWorld.nim arg1 arg2
```

上面编译的是调试版本，我们可以编译发布版本：

```
nim c -d:release helloWorld.nim
```

默认情况下，nim编译器会产生大量运行时检查，使用-d:release可以关闭一些检查并打开优化。生产代码也请使用-d:realease。如果要与c语言等不安全代码比较性能，请用-d:danger 否则可能nim编译器会阻碍代码通过检查。

对于helloworld.nim里的语法：

```
var name = readLine(stdin)  #自动推断类型
```

以上可以实现类型自动推断，另外，helloWorld里包含几个编译器已知的标识符：echo、readLine等。这些内置函数由system模块所声明，system模块由任何其他模块隐式导入。

二、词汇元素

1. 字符和字符串

字符用单引号包裹：``char`

字符串用双引号包裹："string"

转义字符：\n \t

原始字符串文字（无视转义）： r"c:\program\t"

长字符串文字（无视转义）： """ long string contents """

2. 注释

单行注释：

```
#这是单行注释
```

多行注释：

```
#[这是

多行

注释]#
```

3. 数字

```
123     #正常
1_000_000 #下划线分割，编译时自动忽略
1.0e9   #十亿
0x123   #十六进制
0b0101001 #二进制
0o1234567 #八进制
```

三、var语句

```
var x , y : int
var
  x , y : int
  #这里也可以注释
  a , b , c : string
```

四、const常量

编译时就已知的值为常量：

```
const  x  =  "abc"

const
  x  =  1
  # 这里也可以出现注释
  y  =  2
  z  =  y  +  5  # 可以进行计算,但结果必须在编译时已知
```

五、let单赋值变量

let定义的变量是单赋值，也就是它只能赋值一次，赋值完就不能更改：

```
let  x  =  "abc"  # 引入一个新变量 `x` 并为其绑定一个值
x  =  "xyz"       # 非法：赋值给 `x`
```

let和const的区别是：

let引入了一个不能被重新赋值的变量

const 的意思是"强制编译时必须知道具体值，并直接替换"

```
const  input  =  readLine ( stdin )  # 错误：需要常量表达式
let  input  =  readLine ( stdin )    # 有效
```

六、赋值语句

赋值语句将新值分配给变量或更一般地分配给存储位置：

```
var  x  =  "abc"  # 引入一个新变量 `x` 并为其分配一个值
x  =  "xyz"      # 为 `x` 分配一个新值
```

七、控制流语句

1. if语句

```
let name = readLine(stdin)
if name == "":
  echo "Poor soul, you lost your name?"
elif name == "name":
  echo "Very funny, your name is name."
else:
  echo "Hi, ", name, "!"
```

2. case语句

示例1：

```
case x
of 5:
  echo "Five!"
of 7:
  echo "Seven!"
of 10:
  echo "Ten!"
else:
  echo "unknown number"  #其他情况执行这一句
```

示例2：

```
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    #case语句必须包含所有可能的情况，如果我们不关心其他情况，则可以用
                   #这一句代替，没有它则代码无法编译
```

每个分支多种情况：

```
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"
```

还有另外一种case：

```
case n
of 0..2, 4..7: echo "The number is in the set: {0, 1, 2, 4, 5, 6, 7}"
of 3, 8: echo "The number is 3 or 8"
else: discard
```

3. while语句

while 语句是一个简单的循环结构：

```
echo  "What's your name? "
var  name  =  readLine ( stdin )
while  name  ==  "" :
  echo  "Please tell me your name: "
  name  =  readLine ( stdin )  # 没有`var`，因为我们没有声明一个新变量这里
```

该示例使用 while 循环不断询问用户的姓名，只要用户不输入任何内容（仅按 RETURN）。

4. for语句

for 语句用于循环遍历：

```
echo  "Counting to ten: "
for  i  in  countup ( 1 ,  10 ) :
  echo  i
# --> 在不同的行上输出 1 2 3 4 5 6 7 8 9 10
```

还有第二种表达方式带来同样的结果：

```
for i in 1 .. 10:
  ...
```

如果不包括10则：

```
for i in 0 ..< 10:
  ...  # the same as 0 .. 9
```

倒数的实现如下：

```
echo "Counting down from 10 to 1: "
for i in countdown(10, 1):
  echo i
# --> Outputs 10 9 8 7 6 5 4 3 2 1 on different lines
```

还有其他有用的集合迭代器：

（1）items和mitems，分别提供不可变和可变元素

（2）pairs和mpairs提供元素和索引号（分别为不可变和可变）

```
for index, item in ["a","b"].pairs:
  echo item, " at index ", index
# => a at index 0
# => b at index 1
```

5. 作用域和块

控制流语句隐式开启了一个新的块作用域，作用域内的值外部不可访问：

```
while  false :
  var  x  =  "hi"
echo  x  # 不起作用
```

块语句可用于显式打开一个新块：

```
block  myblock :
  var  x  =  "hi"
echo  x  # 也不起作用
```

块标签：myblock是可选的。

6. break语句

break用于跳出一个块作用域，如for、while、block。一般它跳出最近的块，除非指定跳出的块标签：

```
block  myblock :
  echo  "entering block"
  while  true :
    echo  "looping"
    break  # 离开循环，但不离开块
  echo  "still in block"
echo  "outside the block"

block  myblock2 :
  echo  "entering block"
  while  true :
    echo  "looping"
    break  myblock2  # 离开块（和循环）
  echo  "still in block"  # 它不会被打印出来
echo  "outside the block"
```

7.continue语句

跳过当前轮迭代，进入下一轮：

```
for  i  in  1  ..  5 :
  if  i  <=  3 :  continue
  echo  i  # 只会打印 4 和 5
```

8.when语句

```
when system.hostOS == "windows":
  echo "running on Windows!"
elif system.hostOS == "linux":
  echo "running on Linux!"
elif system.hostOS == "macosx":
  echo "running on Mac OS X!"
else:
  echo "unknown operating system"
```

和if语句差不是很多，但主要有三种区别：

（1）每个条件都必须是一个常量表达式，因为它是由编译器计算的。

（2）分支中的语句不会打开新的作用域。

（3）编译器检查语义并仅为属于第一个计算结果为true 的条件的语句生成代码。

when语句对于编写特定于平台的代码很有用，类似于C 编程语言中的#ifdef结构。

八、语句和缩进

```
# 单赋值语句不需要缩进：
if  x :  x  =  false

# 嵌套 if 语句需要缩进：
if  x :
  if  y :
    y  =  false
  else :
    y  =  true

# 需要缩进，因为条件后面有两个语句：
if  x :
  x  =  false
  y  =  false
```

如果表达式是语句的一部分，表达式可以在某些地方进行缩进来提高可读性：

```
if thisIsaLongCondition() and
    thisIsAnotherLongCondition(1,
       2, 3, 4):
  x = true
```

九、过程

过程的概念和函数、方法差不多：

```
proc yes(question: string): bool =
  echo question, " (y/n)"
  while true:
    case readLine(stdin)
    of "y", "Y", "yes", "Yes": return true
    of "n", "N", "no", "No": return false
    else: echo "Please be clear: yes or no"

if yes("Should I delete all your important files?"):
  echo "I'm sorry Dave, I'm afraid I can't do that."
else:
  echo "I think you know what the problem is just as well as I do."
```

（1）以上定义了一个 yes 过程

（2）传入一个名为question的string类型参数，并返回bool类型值

（3）通过return返回结果

1. 结果变量

proc有一个隐式声明的result变量，代表返回值。

单一个 return 返回语句就是 return result 的简写。

如果没有return则proc会自动返回 result 结果。

```
proc  sumTillNegative ( x :  varargs [ int ] ) :  int  =
  for  i  in  x :
    if  i  <  0 :
      return
    result  =  result  +  i

echo  sumTillNegative ( )  # 回显 0
echo  sumTillNegative ( 3 ,  4 ,  5 )  # 回显 12
echo  sumTillNegative ( 3 ,  4  ,  - 1  ,  6 )  # 回显 7
```

result变量在函数的开头隐式声明，所以可以用 var result 来重新声明一个同名变量来覆盖默认result变量。result的初始值会根据返回类型进行初始化，如果返回的是引用，那么初始化的值为

nil，所以需要手动赋值初始化。

没有return语句且不使用result结果变量的proc，返回最后一个表达式的值，如：

```
proc helloWorld(): string =
  "Hello, World!"
```

输出："Hello, World!

2. 参数

参数默认在proc过程中是不可变的，这让编译器可以以最有效的方式实现参数传递。当然，如果过程里面需要可变变量，就用var来声明：

```
proc printSeq(s: seq, nprinted: int = -1) =
  var nprinted = if nprinted == -1: s.len else: min(nprinted, s.len)
  for i in 0 ..< nprinted:
    echo s[i]
```

如果需要把调用者传入的参数修改了，且在proc外有效的话，则需要用var修饰传参：

```
proc divmod(a, b: int; res, remainder: var int) =
  res = a div b        # integer division
  remainder = a mod b  # integer modulo operation

var
  x, y: int
divmod(8, 5, x, y) # modifies x and y
echo x
echo y
```

如上 res 和 remainder 为调用者传入的x y，divmod调用后，x y的值也被修改。

3. discard

用discard来忽略proc的返回值：

```
discard yes("May I ask a pointless question?")  #yes过程的返回值将被忽略
```

当然，如果proc觉得自己的返回值可以被忽略，那么它可以给自己{.discardable.}标签：

```
proc p(x, y: int): int {.discardable.} =
  return x + y

p(3, 4) # 不用在意返回值
```

4. 带名字传参

参数太多可以带着参数名传参：

```
proc createWindow(x, y, width, height: int; title: string;
                  show: bool): Window =
   ...

var w = createWindow(show = true, title = "My Application",
                     x = 0, y = 0, height = 600, width = 800)
```

编译器只检查每个参数是否只接收一个值。

5. 默认值

可以给形参带默认值：

```
proc createWindow(x = 0, y = 0, width = 500, height = 700,
                  title = "unknown",
                  show = true): Window =
   ...

var w = createWindow(title = "My Application", height = 600, width = 800)
```

其中title和show就有自动推断类型的功能。

6.重载

```
proc toString(x: int): string =
  result =
    if x < 0: "negative"
    elif x > 0: "positive"
    else: "zero"

proc toString(x: bool): string =
  result =
    if x: "yep"
    else: "nope"

assert toString(13) == "positive" # calls the toString(x: int) proc
assert toString(true) == "yep"    # calls the toString(x: bool) proc
```

（toString一般是nim中的 $ 运算符）

编译器选择最合适的调用过程。

7. 操作符

操作符总是可以重载。

甚至我们可以自己去重载现有操作符，但这不太明智。

要定义新的操作符，可以在反引号中定义：`

```
proc `$` (x:myDataType): string = ...   #重载了 $ ，使其支持myDataType
```

反引号 ` 也可以用来调用 操作符：

```
if `==`(`+`(3,4),7): echo "true"
```

8.前向声明

```
# forward declaration:
proc even(n: int): bool
```

```
proc odd(n: int): bool =
  assert(n >= 0) # makes sure we don't run into negative recursion
  if n == 0: false           #隐式返回语句结果
  else:
    n == 1 or even(n-1)

proc even(n: int): bool =  #由于odd中使用even，所以需要前向声明even
  assert(n >= 0) # makes sure we don't run into negative recursion
  if n == 1: false
  else:
    n == 0 or odd(n-1)
```

十、迭代器

iterators迭代器。

以下是一个简单的报数proc：

```
echo "Counting to ten: "
for i in countup(1, 10):
  echo i
```

其中countup就是迭代器，可以实现如下：

```
iterator countup(a,b:int):int =
  var res = a
  while res<=b:
    yield res
    inc(res)
```

这有点像proc过程，但还是有很大的不同的：

（1）迭代器只能从 for 循环中调用

（2）迭代器不能包含 return 语句（proc不能包含yield）

（3）迭代器没有隐式result变量

（4）迭代器不支持递归

（5）迭代器不能前向声明，因为编译器必须能够内联编译迭代器。（此限制将在编译器的未来版本中消失）

十一、基本类型

1. 布尔值 bool

true、false

有一点注意的是， and 和 or 操作符会有短路效果：

```
while p != nil and p.name != "xyz":
  # p.name is not evaluated if p == nil
  p = p.next
```

2. 字符 char

char的大小始终是一个字节，多个char可以构成UTF-8字符，字符文字用单引号括起来。

字符可以用 == 、 < 、 <= 、 > 、 >= 运算符进行比较操作，

$ 运算符将 char 转换为 string，

用 ord 过程来获得 char 的序数，

用 chr 过程来获得 int 对应的 char。

3. 字符串 string

可以用内置的len检索string的长度，长度不计算终止零，不能访问终止零，它的存在只是为了让nim字符串可以在不进行复制的情况下转换为cstring。

字符串赋值是采用赋值字符串的形式，可以用&连接字符串，add插入字符串。

字符串变量用空字符串""初始化。

4. 整数 int uint

nim 内置了整数类型： int int8 int16 int32 int64 uint uint8 uint16 uint32 uint64。

默认整数类型是int。

可以用一个类型后缀来指定一个非默认的整数类型：

```
let
  x  =  0      # x 的类型为 `int`
  y  =  0'i8   # y 的类型为 `int8`
  z  =  0'i32  # z 的类型为 `int32`
  u  =  0' u    # u 的类型为 `uint `
```

位运算：and not xor

位移运算：shr右移、shl左移，位移运算符始终将其参数视为unsigned。

5. 浮点数 float

Nim 内置了这些浮点类型：float float32 float64。

默认的浮点类型是float。

在当前的实现中，float始终为64位。

可以用类型后缀来指定非默认的浮点类型：

```
var
  x  =  0.0       # x 的类型为 `float`
  y  =  0.0'f32   # y 的类型为 `float32`
  z  =  0.0'f64   # z 的类型为 `float64`
```

执行不同浮点类型之间的浮点数计算，会产生类型自动转换：较小的类型转换为较大的类型。整数类型不会自动转换为浮点类型，反之亦然。可使用toInt 或 toFloat进行转换。

6. 类型转换

```
var
  x: int32 = 1.int32   # same as calling int32(1)
  y: int8  = int8('a') # 'a' == 97'i8
  z: float = 2.5       # int(2.5) rounds down to 2
  sum: int = int(x) + int(y) + int(z) # sum == 100
```

十二、类型解析

内置的 $ (stringify)运算符将任何基本类型转换为字符串，然后可以用echo将其打印到控制台。

但是高级类型和自定义类型将用不了 $ 运算符，除非同时自定义 $。

这时候我们可以使用 repr，它使用于任何类型，甚至是带循环的复杂数据图。

以下示例显示，即使是基本类型，使用 $ 和 repr 输出之间也存在差异：

```
var
  myBool = true
  myCharacter = 'n'
  myString = "nim"
  myInteger = 42
  myFloat = 3.14
echo myBool, ":", repr(myBool)
# --> true:true
echo myCharacter, ":", repr(myCharacter)
# --> n:'n'
echo myString, ":", repr(myString)
# --> nim:0x10fa8c050"nim"
echo myInteger, ":", repr(myInteger)
# --> 42:42
echo myFloat, ":", repr(myFloat)
# --> 3.14:3.14
```

十三、高级类型

在nim中，可以在type中定义类型：

```
type
  biggestInt = int64      # biggest integer type that is available
  biggestFloat = float64  # biggest float type that is available
```

枚举类型和object类型只能在type内定义。

1. 枚举 enum

枚举类型变量只能赋值为枚举的指定值之一，这些值是一组有序符号。每个符号在内部映射到一个整数值。第一个符号用0表示，第二个符号用1表示，以此类推：

```
type
  Direction = enum
    north, east, south, west

var x =south
echo x   # print south
```

枚举的符号可以被限定以避免歧义：Direction.north

$ 运算符可以将任何枚举值转换成为其名称，而ord 过程可以将枚举名称转为整数值

枚举可以分配显示序数值，但一定要按升序排列

2. 序数类型 ordinal

枚举、整数、char、bool （subrange）被称为序数类型，序数类型支持以下操作：

```
ord(x)  #返回用于表示x值的整数值
inc(x)  #x自增1
inc(x,n) #x自增n，n为整数
dec(x)  #x自减1
dec(x,n)#x自减n，n为整数
succ(x)  #x的后继
succ(x,n) #x的第n个后继
pred(x)  #x的前驱
pred(x,n) #x的第n个前驱
```

inc、dec、succ、pred可能导致RangeDefect和OverflowDefect错误。

3. 子范围 subrange

subrange是整数或枚举类型的子范围：

```
type
  mySub = range[0..5]
```

mySub是int的子范围，只能包含0到5的值。

system模块将Natural类型定义为range[0..high(int)]，high返回最大值。

4. 集合 sets

集合的元素只能是一定大小的序数类型：

int8 到 int16

uint8/byte-uint16

char

enum

对于有符号整数，集合的基本类型定义在0 .. MaxSetElements-1范围内，其中MaxSetElements当前始终为2^16。

由于集合是作为高性能位向量来实现的，尝试声明具有更大类型的集合将导致错误。

```
var s: set[int64] #错误，设置太大
```

注意，nim提供了 hash sets（需要通过import sets导入），它则没有这样的大小限制。

集合可以通过构造函数构造：{}是空集，空集与任何具体集类型都是兼容的，构造函数也可以用于包含元素或元素范围：

```
type
  CharSet = set [ char ]
var
  x: CharSet
x = {'a','b','c'}
```

集合支持以下操作：

```
A + B ：集合合并

A * B ：集合交集

A - B ：集合差异

A == B ：集合相等

A <= B ：A是B的子集，或A等于B

A < B ：A是B的子集

e in A ：e 是 A 中的元素

e notin A ：A不包含元素 e

contains（A，e）：A不包含e

card（A）：A集合元素个数

incl（A，elem）：A = A+{elem}

excl（A，elem）：A = A-{elem}
```

集合通常用于定义位操作，这很干净也很安全：

```
type
  MyFlag* {.size: sizeof(cint).} = enum
    A
    B
    C
    D
  MyFlags = set[MyFlag]


proc toNum(f: MyFlags): int = cast[cint](f)
proc toFlags(v: int): MyFlags = cast[MyFlags](v)

assert toNum({}) == 0
assert toNum({A}) == 1
assert toNum({D}) == 8
assert toNum({A, C}) == 5
assert toFlags(0) == {}
assert toFlags(7) == {A, B, C}
```

5. 数组 array

数组是简单的定长容器，数组内的元素应该具有同样的类型，序号一般为序数类型。

通过 [] 构造数组：

```
type
  IntArray = array[0..5, int] #序号为0-5，值为int类型
var
  x: IntArray
  x = [1,2,3,6,7,8]
for i in low(x).. high(x): #low为最低序号，high为最高序号
  echo x[i]
```

x[i]代表获取x的第i个元素，这通常会进行边界检查，除非在编译程序时带上：

--bound_checks: off

赋值语句复制整个数组

len返回数组长度

low返回合法的最低序号

high返回合法的最高序号

```
type
  Direction = enum
    north, east, south, west
  BlinkLights = enum
    off, on, slowBlink, mediumBlink, fastBlink
  LevelSetting = array[north..west, BlinkLights]
var
  level: LevelSetting
level[north] = on
level[south] = slowBlink
level[east] = fastBlink
echo level        # --> [on, fastBlink, slowBlink, off]
###################level[west]初始化为第一个 off
echo low(level)   # --> north
echo len(level)   # --> 4
echo high(level)  # --> west
```

多维数组有所不同：

```
type
  LightTower = array[1..5, LevelSetting] #嵌套两个一维数组
  #或这么定义:  LightTower = array[1..5, array[north..west, BlinkLights]
var
  tower: LightTower
tower[1][north] = slowBlink #二个维度的index可以不同
tower[1][east] = mediumBlink
echo len(tower)     # --> 10 #最高维度的len只显示当前维度的长度
echo len(tower[1])  # --> 4 #其他维度长度进去
echo tower          # --> [[slowBlink, mediumBlink, ...more output..
# The following lines don't compile due to type mismatch errors
#tower[north][east] = on
#tower[0][1] = on
```

int类型下标的定义可以简化如下:

```
type
  IntArray = array[0..5, int] # an array that is indexed with 0..5
  QuickArray = array[6, int]  # an array that is indexed with 0..5
var
  x: IntArray
  y: QuickArray
x = [1, 2, 3, 4, 5, 6]
y = x
for i in low(x) .. high(x):
  echo x[i], y[i]
```

6. 列表 sequence

seq类似于array，但是变长的，常被分配在堆和垃圾收集器。

seq常用int类型做index，len、low、high也有用

```
var
  x: seq[int]
x = @[1,2,3] # @符号是array转seq的符号
```

seq用@[]初始化

迭代器:

```
for value in @[3, 4, 5]:
  echo value
# --> 3
# --> 4
# --> 5

for i, value in @[3, 4, 5]:
  echo "index: ", $i, ", value:", $value
# --> index: 0, value:3
# --> index: 1, value:4
# --> index: 2, value:5
```

7. 开放数组 openarray

一个可以变长的数组，始终使用0开始的int类型索引，len、low、high都能用。

openarray只能用于参数，将可变长度的数组或列表参数传入proc，任何具有兼容基类型的数据都能传入openarray。

```
var
  fruits:   seq[string]       # reference to a sequence of strings that is initialized with '@[]'
  capitals: array[3, string]  # array of strings with a fixed size

capitals = ["New York", "London", "Berlin"]   # array 'capitals' allows assignment of only three elements
fruits.add("Banana")          # sequence 'fruits' is dynamically expandable during runtime
fruits.add("Mango")

proc openArraySize(oa: openArray[string]): int =
  oa.len

assert openArraySize(fruits) == 2     # procedure accepts a sequence as parameter
assert openArraySize(capitals) == 3   # but also an array type
```

openarray不能嵌套，不支持多维，因为这样的需求比较少，并且不能有效地完成。

8. 可变参数 varargs

varargs类似于openarray，将可变数量参数传递给proc，编译器自动将参数列表转换为数组：

```
proc myWriteln(f: File, a: varargs[string]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, "abc", "def", "xyz")
# is transformed by the compiler to:
myWriteln(stdout, ["abc", "def", "xyz"])
```

这样的转换只当varargs是最后一个参数时才会发生，在转换时还能进行类型转换：

```
proc myWriteln(f: File, a: varargs[string, `$`]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, 123, "abc", 4.0)
# is transformed by the compiler to:
myWriteln(stdout, [$123, $"abc", $4.0])
```

在上述示例中，$ 作用于任何传递给varargs的参数，$ 是一个 nop?

9. 切片 slice

取范围：

```
var
  a = "Nim is a programming language"
  b = "Slices are useless."

echo a[7 .. 12] # --> 'a prog'
b[11 .. ^2] = "useful"
echo b # --> 'Slices are useful.'
```

前后索引如下：

```
"Slices are useless."
 |          |     |
 0         11    17   using indices
^19        ^8    ^2   using ^ syntax
```

10. 对象 Object

默认将不同类型的值打包到单一类型的结构是object。

object赋值也是复制赋值。

object内置构造函数，它内部任何field可初始化为默认值。

```
type
  Person = object
    name: string
    age: int

var person1 = Person(name: "Peter", age: 30)

echo person1.name # "Peter"
echo person1.age  # 30

var person2 = person1 # copy of person 1

person2.age += 14

echo person1.age # 30
echo person2.age # 44


# the order may be changed
let person3 = Person(age: 12, name: "Quentin")

# not every member needs to be specified
let person4 = Person(age: 3)
# unspecified members will be initialized with their default
# values. In this case it is the empty string.
doAssert person4.name == ""
```

标记为*，让object类型和其内部field可见：

```
type
  Person* = object # the type is visible from other modules
    name*: string  # the field of this type is visible from other modules
    age*: int
```

11. 元组 tuple

与对象类型不同的是，元组类型在结构上是类型化的，这意味着如果不同的元组类型以相同的顺序指定相同类型和相同名称的字段，则它们是等价的。

构造函数（）用于构造元组，构造函数中的字段顺序要和元组定义中的顺序一致。与对象不同的是，此处可能不使用元组类型的名称。

t.field和t[i]用于访问元组的元素

```
type
  # type representing a person:
  # A person consists of a name and an age.
  Person = tuple
    name: string
    age: int

  # Alternative syntax for an equivalent type.
  PersonX = tuple[name: string, age: int]

  # anonymous field syntax
  PersonY = (string, int)

var
  person: Person
  personX: PersonX
  personY: PersonY

person = (name: "Peter", age: 30)
# Person and PersonX are equivalent
personX = person

# Create a tuple with anonymous fields:
personY = ("Peter", 30)

# A tuple with anonymous fields is compatible with a tuple that has
# field names.
person = personY
personY = person

# Usually used for short tuple initialization syntax
person = ("Peter", 30)

echo person.name # "Peter"
echo person.age  # 30

echo person[0] # "Peter"
echo person[1] # 30

# You don't need to declare tuples in a separate type section.
var building: tuple[street: string, number: int]
building = ("Rue del Percebe", 13)
echo building.street

# The following line does not compile, they are different tuples!
#person = building
# --> Error: type mismatch: got (tuple[street: string, number: int])
#     but expected 'Person'
```

使用不同字段名称创建的元组将被视为不同的对象，尽管它们具有相同的字段类型。

元组可以在变量赋值期间解包，要使元组解包工作，您必须在要分配给解包的值周围使用括号，否则，您将为所有单个变量分配相同的值！

```
import std/os

let
  path = "usr/local/nimc.html"
  (dir, name, ext) = splitFile(path)
  baddir, badname, badext = splitFile(path)
echo dir      # outputs "usr/local"
echo name     # outputs "nimc"
echo ext      # outputs ".html"
# All the following output the same line:
# "(dir: usr/local, name: nimc, ext: .html)"
echo baddir
echo badname
echo badext
```

解包同样支持for循环：

```
let a = [(10, 'a'), (20, 'b'), (30, 'c')]

for (x, c) in a:
  echo x
# This will output: 10; 20; 30

# Accessing the index is also possible:
for i, (x, c) in a:
  echo i, c
# This will output: 0a; 1b; 2c
```

元组的字段始终是公共的，它们不需要明确标记为*导出，这与对象类型中的字段不同

12. 引用和指针 reference and pointer

引用是引入多对一关系的一种方式，这意味着不同的引用可以指向和修改内存中的相同位置。

nim中区分跟踪和未跟踪引用，未跟踪引用也成为指针。跟踪引用指向垃圾收集堆中的对象，未跟踪引用指向手动分配的对象或内存中其他位置的对象。因此未追踪的引用是不安全的。然而，对于某些低级操作（例如访问硬件），未跟踪的引用是必要的。

跟踪引用使用ref声明，未跟踪引用使用ptr声明。

[]用于取消引用，.符号用于隐式取消引用

```
type
  Node = ref object
    le, ri: Node
    data: int

var n = Node(data: 9)
echo n.data
# no need to write n[].data; in fact n[].data is highly discouraged!
```

要分配一个新的跟踪对象，可以使用内置过程new：

```
var n: Node
new(n)
```

要处理未跟踪内存，可以使用：alloc、dealloc、realloc。

如果引用没有指向任何内容，它的值为 nil 。

13. 程序类型 proc

过程类型，是指向过程的指针。nil 是过程类型变量的允许值。

```
proc greet(name: string): string =
  "Hello, " & name & "!"

proc bye(name: string): string =
  "Goodbye, " & name & "."

proc communicate(greeting: proc (x: string): string, name: string) =
  echo greeting(name)

communicate(greet, "John")
communicate(bye, "Mary")
```

14. 不同类型 distinct

Distinct 类型允许创建“不暗示它与其基类型之间存在子类型关系”的新类型。

十四、模块

nim使用模块概念将程序拆分成多个部分，每个模块在自己的文件中。

模块启用信息隐藏和单独编译，使用import语句访问其他模块，仅*标的参数会被其他模块访问到。

```
# Module A
var
  x*, y: int

proc `*` *(a, b: seq[int]): seq[int] =
  # allocate a new sequence:
  newSeq(result, len(a))
  # multiply two int sequences:
  for i in 0 ..< len(a): result[i] = a[i] * b[i]

when isMainModule:
  # test the new `*` operator for sequences:
  assert(@[1, 2, 3] * @[1, 2, 3] == @[1, 4, 9])
```

以上模块导出 x 和 * proc，模块的顶级语句在程序开始时执行，这可以用于初始化复杂的数据结构。

每个模块都有一个魔法向量 isMainModule ，如果模块被编译为主文件，则该常量为真。如上所示，这在模块中嵌入测试很有用。

模块符号可以用module.symbol来进行命名空间限定，如果symbol有歧义，那么则必须限定：

```
# Module A
var x*: string
```

```
# Module B
var x*: int
```

```
# Module C
import A, B
write(stdout, x) # error: x is ambiguous
write(stdout, A.x) # okay: qualifier used

var x = 4
write(stdout, x) # not ambiguous: uses the module C's x
```

这个规则不适用于proc和迭代器的导出。

这里适用重载：

```
# Module A
proc x*(a: int): string = $a
```

```
# Module B
proc x*(a: string): string = $a
```

```
# Module C
import A, B
write(stdout, x(3))   # no error: A.x is called
write(stdout, x(""))  # no error: B.x is called

proc x*(a: int): string = discard
write(stdout, x(3))   # ambiguous: which `x` is to call?
```

1. 排除 excluding symbol

用以下语法排除特定symbol的引入：

```
import mymodule except y
```

2. From声明

可以用from声明只引入指定symbol：

```
from mymodule import x, y, z
```

from语句还可强制对符号进行命名空间限定：

```
from mymodule import x, y, z

x()           # use x without any qualification
```

```
from mymodule import nil

mymodule.x()  # must qualify x with the module name as prefix

x()           # using x here without qualification is a compile error
```

可以对模块名称进行重命名

```
from mymodule as m import nil
m.x()
```

3. 包含 include

include语句和import有所不同，include常用于将一个大文件拆分成多个小文件，只包含文件内容：

```
include fileA, fileB, fileC
```