提示：本文档可能部分细节有些过时，具体可见最新的手册。

一、编译指示

pragmas 是nim在不引入大量新关键字的情况下为编译器提供额外信息和命令的方法。

pragmas的语法是这样的：{. #这里是添加的地方# .}

本教程没有pragmas的具体信息，详见用户手册。

二、面向对象编程

nim支持面向对象oop，但面向过程其实也不错，这里自由选择。

1. 继承 Inheritance

要启用继承，对象要继承自RootObj，这可以直接继承，也可以从继承了RootObj的对象里间接继承。通常具有继承的类型也被标记为 ref 类型，即使这不是严格限定的。要在运行时检查对象是否属于某种类型，可以使用 of 运算符。

```
type
  Person = ref object of RootObj
    name*: string
    age : int

  Student = ref object of Person
    id * : int

var
  student: Student
  person: Person
assert(student of Student) # is true
# object construction:
student = Student(name: "Anton", age: 5, id: 2)
echo student[]
```

继承使用 object of 语法完成的，目前不支持多重继承。如果一个对象类型没有合适的祖先，可以使用RootObj作为它的祖先。没有祖先的对象默认是最终的，可以使用inheritable来引入除system.RootObj外的object root。

只要使用继承，最好使用ref对象。

对于简单的代码重用，组合通常由于继承。这两者在nim中的效率差不多。

2. 相互递归类型

对象、元组和引用可以对相互依赖的相当复杂的数据结构进行建模；它们是相互递归的。在 Nim 中，这些类型只能在单个类型部分中声明。（任何其他事情都需要任意符号前瞻，这会减慢编译速度。）

```
type
  Node = ref object  # a reference to an object with the following field:
    le, ri: Node     # left and right subtrees
    sym: ref Sym     # leaves contain a reference to a Sym

  Sym = object       # a symbol
    name: string     # the symbol's name
    line: int        # the line the symbol was declared in
    code: Node       # the symbol's abstract syntax tree
```

3. 类型转换

nim区分cast和type conversion

cast属于强制类型转换，直接粗暴地将位模式转为另一种类型

type conversion属于比较有礼貌的转换，它保留抽象值而非位模式。如果type conversion是非法地，也会报错：

```
proc getID(x: Person): int =
  Student(x).id #type conversion模式是:   目标类型(现在类型)
```

如果x不是Student ， 则会引发InvalidObjectConversionDefect异常。

4. 对象变种

需要对象有一个简单地变形地情况下：

```
# This is an example how an abstract syntax tree could be modelled in Nim
type
  NodeKind = enum  # the different node types
    nkInt,          # a leaf with an integer value
    nkFloat,        # a leaf with a float value
    nkString,       # a leaf with a string value
    nkAdd,          # an addition
    nkSub,          # a subtraction
    nkIf            # an if statement
  Node = ref object
    case kind: NodeKind  # the `kind` field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

var n = Node(kind: nkFloat, floatVal: 1.0)
# the following statement raises an `FieldDefect` exception, because
# n.kind's value does not fit:
n.strVal = ""
```

从示例中可以看出，对象层次结构的一个优点是不需要在不同对象类型之间进行转换。然而，访问无效的对象字段会引发异常。

5. 方法调用

方法调用有个语法糖：

用obj.methodName(args)来代替methodName(obj, args)。

例如用obj.len代替len(obj)

这种语法糖并不限制对象，可以用于任何类型：

```
import std/strutils

echo "abc".len # is the same as echo len("abc")
echo "abc".toUpperAscii()
echo({'a', 'b', 'c'}.card)
stdout.writeLine("Hallo") # the same as writeLine(stdout, "Hallo")
```

6. 特性

对于get和set

nim不需要get，设置值则不一样，需要特殊地setter语法：

```
type
  Socket* = ref object of RootObj
    h: int # cannot be accessed from the outside of the module due to missing star

proc `host=`*(s: var Socket, value: int) {.inline.} =
  ## setter of host address
  s.h = value

proc host*(s: Socket): int {.inline.} =
  ## getter of host address
  s.h

var s: Socket
new s
s.host = 34  # same as `host=`(s, 34)
```

这个案例同样显示 inline 的用法

如下重载 [] 来提供访问值的方法：

```
type
  Vector* = object
    x, y, z: float

proc `[]=`* (v: var Vector, i: int, value: float) =
  # setter
  case i
  of 0: v.x = value
  of 1: v.y = value
  of 2: v.z = value
  else: assert(false)

proc `[]`* (v: Vector, i: int): float =
  # getter
  case i
  of 0: result = v.x
  of 1: result = v.y
  of 2: result = v.z
  else: assert(false)
```

7. 动态调度

proc通常是静态调度的，要动态调度需要将proc替换为method

```
type
  Expression = ref object of RootObj ## abstract base class for an expression
  Literal = ref object of Expression
    x: int
  PlusExpr = ref object of Expression
    a, b: Expression

# watch out: 'eval' relies on dynamic binding
method eval(e: Expression): int {.base.} =
  # override this base method
  quit "to override!"

method eval(e: Literal): int = e.x
method eval(e: PlusExpr): int = eval(e.a) + eval(e.b)

proc newLit(x: int): Literal = Literal(x: x)
proc newPlus(a, b: Expression): PlusExpr = PlusExpr(a: a, b: b)

echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
```

选择proc 和 method需要按实际情况确定。

上例中，method 根据不同类型的输入动态绑定方法类型。

从nim0.20开始，要使用多方法，必须在编译时显式传递 -- multimethods : on

在多方法中，所有具有对象类型的参数都用于调度：

```
type
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

method collide(a, b: Thing) {.inline.} =
  quit "to override!"

method collide(a: Thing, b: Unit) {.inline.} =
  echo "1"

method collide(a: Unit, b: Thing) {.inline.} =
  echo "2"

var a, b: Unit
new a
new b
collide(a, b) # output: 2
```

如示例所示，多重方法的调用不能有歧义：Collide 2 优于 collide 1，因为分辨率是从左到右进行的。因此Unit , Thing优于Thing , Unit。

性能说明：Nim 不生成虚方法表，而是生成调度树。这避免了方法调用的昂贵间接分支并启用内联。但是，编译时评估或死代码消除等其他优化不适用于方法。

三、报错机制

在nim中，异常是对象，异常类型常以"Error"作为后缀，异常来自system.Exception，提供通用接口。

必须在堆上分配异常，因为它们的生命周期未知，编译器将阻止您引发在栈上创建的异常。所有引发的异常应至少在msg字段中指定引发的原因。

最好不要把异常机制作为控制流的替代方法。

1. raise

使用raise引发异常：

```
var
  e: ref OSError
new(e)
e.msg = "the request to the OS failed"
raise e
```

如果raise关键字后面没有跟表达式，则最后一个异常将被再次调用。可以使用以下填充模式：

```
raise newException(OSError, "the request to the OS failed")
```

2. try

```
from std/strutils import parseInt

# read the first two lines of a text file that should contain numbers
# and tries to add them
var
  f: File
if open(f, "numbers.txt"):
  try:
    let a = readLine(f)
    let b = readLine(f)
    echo "sum: ", parseInt(a) + parseInt(b)
  except OverflowDefect:
    echo "overflow!"
  except ValueError:
    echo "could not convert string to integer"
  except IOError:
    echo "IO error!"
  except CatchableError:
    echo "Unknown exception!"
    # reraise the unknown exception:
    raise
  finally:
    close(f)
```

除非引发异常，否则执行try之后的语句。

如果存在未明确列出的异常，则执行空的except部分。类似于if语句中的else部分。

如果有finally部分，它总是在异常处理程序之后执行。

异常在except部分被消耗，如果未处理异常，他将通过调用堆栈传播。这意味着过程的其余部分（不在finally子句中）通常不会执行。

如果需要访问except分支中的实际异常对象或消息，可以使用系统模块中的getCurrentException()和getCurrentExceptionMsg()。

```
try:
  doSomethingHere()
except CatchableError:
  let
    e = getCurrentException()
    msg = getCurrentExceptionMsg()
  echo "Got exception ", repr(e), " with message ", msg
```

3. 使用引发的异常注释过程

通过可选的{. raises .}pragma，你可以指定一个proc引发一组特定异常。

```
proc complexProc() {.raises: [IOError, ArithmeticDefect].} =
  ...

proc simpleProc() {.raises: [].} =
  ...
```

可以添加 {.effect.}到pragma到proc，编译器将输出所有推断的效果到那个点。doc命令为整个模块生成文档并用引发的异常列表装饰所有 proc。

四、泛型

泛型是 Nim 使用类型参数对过程、迭代器或类型进行参数化的方法。通用参数写在方括号内，例如Foo [T]。它们对于高效类型安全容器最有用：

```
type
  BinaryTree*[T] = ref object # BinaryTree is a generic type with
                              # generic param `T`
    le, ri: BinaryTree[T]     # left and right subtrees; may be nil
    data: T                   # the data stored in a node

proc newNode*[T](data: T): BinaryTree[T] =
  # constructor for a node
  new(result)
  result.data = data

proc add*[T](root: var BinaryTree[T], n: BinaryTree[T]) =
  # insert a node into the tree
  if root == nil:
    root = n
  else:
    var it = root
    while it != nil:
      # compare the data items; uses the generic `cmp` proc
      # that works for any type that has a `==` and `<` operator
      var c = cmp(it.data, n.data)
      if c < 0:
        if it.le == nil:
          it.le = n
          return
        it = it.le
      else:
        if it.ri == nil:
          it.ri = n
          return
        it = it.ri

proc add*[T](root: var BinaryTree[T], data: T) =
  # convenience proc:
  add(root, newNode(data))

iterator preorder*[T](root: BinaryTree[T]): T =
  # Preorder traversal of a binary tree.
  # This uses an explicit stack (which is more efficient than
  # a recursive iterator factory).
  var stack: seq[BinaryTree[T]] = @[root]
  while stack.len > 0:
    var n = stack.pop()
    while n != nil:
      yield n.data
      add(stack, n.ri)  # push right subtree onto the stack
      n = n.le          # and follow the left pointer

var
  root: BinaryTree[string] # instantiate a BinaryTree with `string`
```

```
add(root, newNode("hello")) # instantiates `newNode` and `add`
add(root, "world")          # instantiates the second `add` proc
for str in preorder(root):
  stdout.writeLine(str)
```

该示例显示了一个通用的二叉树。根据上下文，括号用于引入类型参数或实例化通用过程、迭代器或类型。

方法调用语法中使用泛型，有特殊的[:T]语法：

```
proc foo[T](i: T) =
  discard

var i: int

# i.foo[int]() # Error: expression 'foo(i)' has no type (or is ambiguous)

i.foo[:int]() # Success
```

五、模板

模板是一种在 Nim 的抽象语法树上运行的简单替换机制。模板在编译器的语义通道中进行处理。它们与语言的其余部分很好地集成并且没有 C 的预处理器宏缺陷。

调用模板请像调用过程一样调用它：

```
template `!=` (a, b: untyped): untyped =
  # this definition exists in the System module
  not (a == b)

assert(5 != 6) # the compiler rewrites that to: assert(not (5 == 6))
```

! =、>、>=、in、notin、isnot运算符实际上是模板：这样做的好处是，如果您重载==运算符，!=运算符将自动可用并做正确的事情。（IEEE 浮点数除外——NaN 破坏了基本的布尔逻辑。）

模板对于惰性评估目的特别有用。考虑一个简单的记录过程：

```
const
  debug = true

proc log(msg: string) {.inline.} =
  if debug: stdout.writeLine(msg)

var
  x = 4
log("x has the value: " & $x)
```

这段代码有一个缺点：如果某天将debug设置为 false，仍然会执行相当昂贵的$和&操作！（过程的参数评估是急切的）。

将日志过程变成模板解决了这个问题：

```
const
  debug = true

template log(msg: string) =
  if debug: stdout.writeLine(msg)

var
  x = 4
log("x has the value: " & $x)
```

参数的类型可以是普通类型或元类型untyped、typed或type。type表示只能将类型符号作为参数给出，而untyped表示在将表达式传递给模板之前不执行符号查找和类型解析。

如果模板没有明确的返回类型，则使用void来与过程和方法保持一致。

要将语句块传递给模板，请对最后一个参数使用untyped ：

```
template withFile(f: untyped, filename: string, mode: FileMode,
                  body: untyped) =
  let fn = filename
  var f: File
  if open(f, fn, mode):
    try:
      body
    finally:
      close(f)
  else:
    quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

在示例中，两个writeLine语句绑定到body参数。

如下示例：

```
import std/math

template liftScalarProc(fname) =
  ## Lift a proc taking one scalar parameter and returning a
  ## scalar value (eg `proc sssss[T](x: T): float`),
  ## to provide templated procs that can handle a single
  ## parameter of seq[T] or nested seq[seq[]] or the same type
  ##
  ## .. code-block:: Nim
  ##  liftScalarProc(abs)
  ##  # now abs(@[@[1,-2], @[-2,-3]]) == @[@[1,2], @[2,3]]
  proc fname[T](x: openarray[T]): auto =
    var temp: T
    type outType = typeof(fname(temp))
    result = newSeq[outType](x.len)
    for i in 0..<x.len:
      result[i] = fname(x[i])

liftScalarProc(sqrt)   # make sqrt() work for sequences
echo sqrt(@[4.0, 16.0, 25.0, 36.0])   # => @[2.0, 4.0, 5.0, 6.0]
```

六、编译成javaScript

Nim 代码可以编译成 JavaScript。但是，为了编写与 JavaScript 兼容的代码，您应该记住以下几点：

- addr和ptr在 JavaScript 中的语义略有不同。如果您不确定它们是如何转换为 JavaScript，建议避免使用它们。
- JavaScript 中的cast [T] (x)被翻译为(x)，但有符号/无符号整数之间的转换除外，在这种情况下，它的行为类似于 C 语言中的静态转换。
- JavaScript 中的cstring表示 JavaScript 字符串。只有在语义上合适时才使用cstring是一种很好的做法。例如，不要将cstring用作二进制数据缓冲区。