R 函数

shixiangzhou

创建函数

```
addnum<- function(x, y){
  s <- x+y
  return(s)
}

addnum (3,7)

addnum2<- function(x, y){
  x+y
}

addnum2(3,7)
addnum2 #输入函数名查看函数定义
body(addnum2) #检查函数体
formals(addnum2) # 检查形参
args(addnum2)
help(sum)
?sum
```

运行原理

R 函数是组织良好且可重用的代码块。**函数本身就是对象**。函数有三个组成部分：body() 函数内部代码，formals() 控制如何调用函数的参数列表，environment() 函数变量位置的 “地图”。

```
f<- function(x) x^2
f
```

```
## function(x) x^2
```

```
formals(f)
```

```
## $x
```

```
body(f)
```

```
## x^2
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

可以用这三个函数的赋值形式对函数进行修改。

与 R 中的所有对象一样，函数也可以有任意多的附加属性，基础包中的属性 “srcref” 是源参考，指向用来创建函数的源代码，与 body() 不同，包含代码的注释和其他格式。

原函数

是另外，如 sum()，它使用 Primitive() 直接调用 c 代码且不包含 R 的代码，因此它的 formals(),body()，environment() 都是 NULL。原函数只存在于 base(基础包) 中，低层运算更高效。

```
formals(sum)
```

```
## NULL
```

匹配参数

```
defaultarg <- function(x, y = 5){
  y <- y * 2
  s <- x+y
  return(s)
}

defaultarg(3) #传递3作为参数
defaultarg(1:3) #传递其他类型参数
defaultarg(3,6) #传两个参数
defaultarg(y = 6, x = 3) #一组命名参数

funcarg <- function(x, y, type= "sum"){
  if (type == "sum"){ #使用命名参数的同时，还可以使用if-else条件语句
    sum(x,y)
  }else if (type == "mean"){
```

```
    mean(x,y)
  }else{
    x * y
  }
}

#测试
funcarg(3,5)
funcarg(3,5, type = 'mean')
funcarg(3,5, type = 'unknown')

unspecarg <- function(x, y, ...){ #传递不确定数量的参数
  x <- x + 2
  y <- y * 2
  sum(x,y, ...)
}

unspecarg(3,5)
unspecarg(3,5,7,9,11)
funcarg(3,5, t = 'unknown') # 参数缩写
```

灵活的参数绑定机制，如果不带参数名就通过位置来绑定传递的值。

理解环境

除了函数名，函数体，环境也是函数的另一个基本组成部分。

环境是 R 管理和存储各种类型变量的地方。除了全局环境外，每一个函数会在创建之初激活自己的环境。

```
environment() #查看当前的环境
.GlobalEnv # 查看全局环境
globalenv()

identical(globalenv(), environment()) #比较环境

myenv <- new.env() #创建一个新环境
myenv

myenv$x <- 3 # 不同环境中的变量
```

```
ls(myenv)

ls()

x
addnum <- function(x, y){
  x+y
}

environment(addnum) #得到函数的环境
environment(lm) #判断函数的环境是否属于程序包

addnum2 <- function(x, y){
  print(environment()) #在函数中打印环境
  x+y
}
addnum2(2,3)

addnum3 <- function(x, y){
  func1 <- function(x){
    print(environment()) #比较函数内部和外部环境
  }
  func1(x) #嵌套的函数
  print(environment())
  x + y
}

addnum3(2,5)
parentenv <- function(){
  e <- environment()
  print(e)
  print(parent.env(e))# parent.env函数获取父环境
}
parentenv()
```

运行原理，

可以把 R 环境看做存储和管理变量的地方，也就是说，只要我们创建了 R 的一个函数或对象，我们就开辟了一个环境。顶层环境默认为是全局环境 R_GlobalEnv，我们可以用函数 environment 确定当前的环境。

我们也可以创建自己的环境，并把变量分配到其中。

创建了一个 addnum 函数之后，可以用 environment 来获取它的环境，由于是在全局环境下创建的，函数显然是属于全局环境的。

当我们获取函数 lm 环境时，却得到了相应的程序包。这意味着函数 lm 位于程序包 stat 的命名空间中。

还可以在函数内部打印出当前环境，通过调用 addnum2，可以确定，函数 environment 输出的环境名与全局环境不同。也就是说我们创建函数时，我们也创建了一个新的环境，以及指向父环境的指针。

静态绑定

有两种绑定变量的方法，动态绑定和静态绑定，静态绑定是函数式编程语言的特征，他的每一个绑定域都会管理变量名和词法环境中的取值，如果一个变量被词法约束了，它会搜索最近的词法环境中的绑定关系。

```
x <- 5
tmpfunc <- function(){
  x + 3
}
tmpfunc()

# 第二个例子，带有嵌套的函数childfunc
x <- 5
parentfunc <- function(){
  x<- 3
  childfunc <- function(){
    x #会使用parentfunc中的x，而不是定义在其外部的x
  }
  childfunc()
}
parentfunc()

#第三个例子
x <- 'string'
localassign<- function(x){
  x <- 5 #在函数内部修改x
  x
}
localassign(x)
x
```

```
x <- 'string'
gobalassign<- function(x){
  x <<- 5  # 重新指派x
  x
}
gobalassign (x)
x
search() #查看R的搜索路径
```

面向对象编程指南

R 有 3 个面向对象系统，类通过描述对象的属性以及对象与其他类的关系来定义对象的行为，在选择方法同时要使用类，函数行为上的不同取决于它们输入类的不同，类通常是分层结构：如果子类中不存在某个方法，它就会使用父类中的方法。

R 的 3 个面向对象系统在类和方法的定义上有所不同：

S3 使用一种称为泛型函数 OO 的面向对象编程方法，与大多数实现消息传递的 OO 编程语言 (java,c++,c#) 不同。

消息被传递给对象，然后对象决定调用哪个函数。

canvas.drawRect(“blue”)

对象在函数调用中出现在方法名之前。

而 S3 就不同，由一种称为泛型函数的特殊函数来决定调用哪个方法。

drawRect(canvas,“blue”)

S3 没有类的正式定义。

S4 的工作方式与 S3 类似，但更正式。S4 有类的正式定义。

参考类 (RC)：RC 实现消息传递 OO，所以方法属于类而不属于函数，$ 用来分隔类和方法，canvas$drawRect(“blue”)

基础类型：是构成其他 OO 系统的内部 C 语言级别的类型；

工具：install.packages(“pryr”)# 检测 OO 性质

基础类型，所有 R 对象底层都是一个用来描述这个对象如何在内存中存储的 C 结构体，其中包含这个对象的内容，内存分配信息及类型。

基础类型不是真正的面向对象系统，因为只有 R 的核心团队才可以创建新类型，可以使用 typeof() 检测对象的基础类型。

可以使用 is.object(x) 返回值是不是 FALSE 检测一个对象是不是一个纯基础类。

即使你永远都不会写 C 代码，但理解基础类型非常重要，因为所有的其他系统都是建立在它们之上，可使用任意基础类型构建 S3 对象，使用特殊的基础类型构建 S4 对象，RC 对象是 S4 与环境对象的结合体。

S3 系统

R 的第一个最简单的 OO 系统，是 R 基础包和统计包中唯一使用的 OO 系统，也是 CRAN 软件包中最常用的系统，大多数对象都是 S3 对象，但不幸的是，在 R 基础包中没有一个简单方法可以检查一个对象是不是 S3 对象。

is.object(x) & ! isS4(x) 确认 x 是对象但不是 S4

另一个简单的方法是使用 prgr::otype()

```
library(pryr)
```

```
## Warning: package 'pryr' was built under R version 4.0.5
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
##
## Attaching package: 'pryr'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     f
```

```
df <- data.frame(x=1:10,y=letters[1:10])
otype(df)
```

```
## [1] "S3"
```

```
otype(df$x) # $ 属性选取运算符
```

```
## [1] "base"
```

```
otype(df$y)
```

```
## [1] "base"
```

在 S3 中，方法属于函数，这个函数称为泛型函数，S3 方法不属于类或对象，与大多数编程语言不同。

为了知道一个函数是不是一个 S3 泛型函数，可以查看它的源代码，找到函数调用 useMethod()：这个函数指出调用的正确方法，也称为方法分派过程。

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000015722428>
## <environment: namespace:base>
```

源代码中：UseMethod(“mean”)

```
ftype(mean)
```

```
## [1] "s3"      "generic"
```

与 sum(),cbind() 类似，有些 S3 泛型函数不调用 UseMethod() 因为它们用 c 语言实现，方法分派称为内部泛型。

大多数现代风格的编程指南不支持在函数名中使用 “.”，使得它们看上去像 S3 的方法，例如，t.test() 是 t 对象的 test 方法吗？

在类名中使用 “.” 也会造成迷糊：print.data.frame() 是 data.frame 的 print 方法吗？或 print.data() 是 frame 的方法？

pryr::ftype() 查看

```
ftype(t.data.frame)
```

```
## [1] "s3"     "method"
```

```
ftype(t.test)
```

```
## [1] "s3"      "generic"
```

可以用 methods() 来查看属于一个泛型函数的所有方法：

```
methods("mean")
```

```
## [1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
## [6] mean.quosure*
## see '?methods' for accessing help and source code
```

```
methods("t.test")
```

```
## [1] t.test.default* t.test.formula*
## see '?methods' for accessing help and source code
```

除基础包中定义的方法外，大部分 S3 方法是不可见的，使用 getS3method() 阅读它们的源代码。

```
require(stats)
exists("predict.ppr") # false
```

```
## [1] FALSE
```

```
getS3method("predict", "ppr")
```

```
## function (object, newdata, ...)
## {
##     if (missing(newdata))
##         return(fitted(object))
##     if (!is.null(object$terms)) {
##         newdata <- as.data.frame(newdata)
##         rn <- row.names(newdata)
##         Terms <- delete.response(object$terms)
##         m <- model.frame(Terms, newdata, na.action = na.omit,
##             xlev = object$xlevels)
##         if (!is.null(cl <- attr(Terms, "dataClasses")))
##             .checkMFClasses(cl, m)
##         keep <- match(row.names(m), rn)
##         x <- model.matrix(Terms, m, contrasts.arg = object$contrasts)
##     }
##     else {
##         x <- as.matrix(newdata)
##         keep <- seq_len(nrow(x))
##         rn <- dimnames(x)[[1L]]
##     }
##     if (ncol(x) != object$p)
##         stop("wrong number of columns in 'x'")
##     res <- matrix(NA, length(keep), object$q, dimnames = list(rn,
##         object$ynames))
##     res[keep, ] <- matrix(.Fortran(C_pppred, as.integer(nrow(x)),
##         as.double(x), as.double(object$smod), y = double(nrow(x) *
##             object$q), double(2 * object$smod[4L]))$y, ncol = object$q)
##     drop(res)
## }
## <bytecode: 0x000000001d5c5250>
## <environment: namespace:stats>
```

对一个类，可列出包含该列的方法的所有泛型函数。

```
methods(class="ts")
```

```
##  [1] [             [<-           aggregate     as.data.frame cbind
##  [6] coerce        cycle         diff          diffinv       initialize
## [11] kernapply     lines         Math          Math2         monthplot
## [16] na.omit       Ops           plot          print         show
## [21] slotsFromS3   t             time          window        window<-
## see '?methods' for accessing help and source code
```

定义类和创建对象

为给一个类创建一个对象实例，只需使用已有的基础对象并设置类属性，在创建时可使用 structure() 或者最后使用 class<-():

```
foo <- structure(list(),class="foo")
# 或者 foo <- list()
 # class(foo)  <- "foo"
```

S3 对象建立在列表或带有属性的原子向量之上。

可使用 class(x) 检查任意对象的属性。

```
class(foo)
```

```
## [1] "foo"
```

```
inherits(foo,"foo") # 查看一个对象是否继承与一个特殊类
```

```
## [1] TRUE
```

一个 S3 对象的类可以是向量，这个向量描述从最具体到最一般的行为。

例如 glm() 对象的类是 c(“glm”,“lm”)，表明广义线性模型是继承线性模型的行为。

类名通常小写，尽量避免使用 “.”。

大多数 S3 类都提供一个构造函数，名字通常与类的名字相同。

```
foo <- function(x){
   if (!is.numeric(x)) stop("x 必须是数字型的")
  structure(list(x),class="foo")}
```

如果没有方法泛型函数就没有用，调用 UseMethod() 函数创建新方法。

跟以前的 OO 编程语言不同，我们甚至可以改变已有对象的类，当然不建议这么做。

选择一个系统

对一个语言来说，3 个 OO 系统实在太多了，对于大多数 R 编程任务来讲，S3 就已经足够了。

如果为相互关联的对象创建更加复杂系统，S4 可能更合适，如 Matrix 包，更有效地存储和计算多种不同类型的稀疏矩阵，定义 102 个类和 20 个泛型函数，Bioconductor 包中也大量使用 S4，对生物对象之间复杂关系建模。

函数式编程

R 语言的核心其实是一门函数式的编程 (FP) 语言，为我们提供了大量的创建和操作函数的工具。

R 有所谓的一级函数，适用于向量的所有操作也都适用于函数：可将函数赋值给变量，将函数存储在列表中，将函数作为参数传递给其他函数，在函数内再创建一个函数，甚至可以把函数作为一个函数的结果返回。

例：

```
set.seed(1014)
df <-data.frame(replicate(6,sample(c(1:10,-99),6,rep=TRUE)))
names(df)<-letters[1:6]
df
```

```
##   a   b   c   d  e f
## 1 7   5 -99   2  5 2
## 2 5   5   5   3  6 1
## 3 6   8   5   9  9 4
## 4 4   2   2   6  6 8
## 5 6   7   6 -99 10 6
## 6 9 -99   4   7  5 1
```

```
# 把-99 用缺失值 NA 代替
df$a[df$a == -99]<- NA
df$b[df$b == -99]<- NA
df$c[df$c == -98]<- NA
df$d[df$d == -99]<- NA
df$e[df$e == -99]<- NA
df$f[df$f == -99]<- NA
#df$g[df$g == -99]<- NA
```

使用赋值和粘贴很容易出错，重复代码很容易错而且使代码变得很难修改，若缺失值从-99 改变为 9999，就必须在多个地方修改。

为避免产生这类漏洞并使代码更灵活，采用 “不要自我重复”– do not repeat yourself, 即 DRY 原则，在系统中每一条知识都必须有一条明确的正式表达。

```
fix_missing <- function(x){
  x[x == -99] <- NA
  x}

df$a< fix_missing(df$a)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
df$b< fix_missing(df$b)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE    NA
```

```
df$c< fix_missing(df$c)
```

```
## [1]    NA FALSE FALSE FALSE FALSE FALSE
```

```
df$d< fix_missing(df$d)
```

```
## [1] FALSE FALSE FALSE FALSE    NA FALSE
```

```
df$e< fix_missing(df$e)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
df$f< fix_missing(df$f)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

这减少了出错的范围，但不能完全消除，仍可能将变量名弄错。下一步就将两个函数结合到一起来避免这种错误发生。

lapply() 可将这种操作应用到数据框中的每一列。

接受 3 个输入，x 为一个列表，f 为一个函数，...传递 f() 的其他参数。

真实的 lapply() 相当复杂，为提高效率，用 c 语言实现的，由于 lapply() 以函数作为输入，所以它是一个泛函，泛函是函数式编程的一个非常重要的部分。

```
fix_missing <- function(x){
  x[x == -99] <- NA
  x}
df[] <- lapply(df,fix_missing)

df[1:5] <- lapply(df[1:5],fix_missing)
```

无论有多少列都可以使用它，不会丢掉任何一列，所有列的操作都是相同的，关键思想是函数组合，将两个简单函数组合起来，一个函数修复缺失值，另一个函数对每一列做同样的操作。

如果每一列使用不同的值来替换缺失值又该怎么处理？

```
fix_missing_99 <- function(x){
  x[x == -99] <- NA
  x}

fix_missing_999 <- function(x){
  x[x == -999] <- NA
  x}
```

与前面一样，它容易出错，然后我们可以使用闭包，它是创建并返回函数的函数，闭包允许我们基于模板来创建函数。

```
missing_fixer <- function(na_value){
     function(x){
       x[x == na_value] <- NA
       x
     }
}
fix_missing_99 <-  (missing_fixer(-99))
fix_missing_999 <- (missing_fixer(-999) )

fix_missing_99(c(-99,-999))
```

```
## [1]   NA -999
```

```
fix_missing_999(c(-99,-999))
```

```
## [1] -99  NA
```

泛函

高阶函数就是以函数作为输入并以函数作为输出的函数。前面已经学习了一种类型的高阶函数：闭包，由另一个函数返回的函数。闭包的一个补充就是泛函，以函数作为输入并返回一个向量的函数，

例：1000 个随机数。

```
randomise <- function(f)
  f(runif(1e3))

randomise(mean) #mean 是一个函数
```

```
## [1] 0.5088265
```

```
randomise(mean)
```

```
## [1] 0.4954888
```

```
randomise(sum)
```

```
## [1] 492.6025
```

3 个常用的泛函为：lapply(),apply(),tapply() 都可以接收一个函数作为输入，并返回一个向量作为输出。

泛函的常用功能就是替代循环，循环的最大缺点是表达不够清晰，for 是对某事进行迭代，但不能清晰地表达更高层次的目的。

lapply() 接收一个函数，并将这个函数应用到列表中的每一个元素，最后再将结果以列表的形式返回，lapply 以 C 语言实现。

lapply 是对 for 循环模式的包装器：为输出创建一个容器，将 f() 应用到列表中的每一个元素，将结果填充到容器中，其他 for 循环泛函都是这一思路的变体。

```
l <- replicate(20,runif(sample(1:10,1)),simplify=FALSE)

out <- vector("list",length(l))
for (i in seq_along(l)){
  out[[i]] <- length(l[[i]])}
unlist(out) # 从列表转换为向量
```

```
##  [1]  6  5  3 10  7  7  6  7  7  4  9  7  9  8  3 10  7  5  5  6
```

```
unlist(lapply(l,length))
```

```
##  [1]  6  5  3 10  7  7  6  7  7  4  9  7  9  8  3 10  7  5  5  6
```

数据框也是列表，所以当我们想对数据框中的每一列进行处理时，lapply 也可以用。

数学泛函

泛函在数学中非常常见，极限，最大值，求根以及定积分都是泛函，给定一个函数，它们返回一个向量，实现它们的算法中都包含迭代。

R 中内置数学泛函：

```r
integrate(sin,0,pi)# 计算曲线面积
```

```
## 2 with absolute error < 2.2e-14
```

```r
# uniroot() # 计算方程 f(x)=0 的根
uniroot(sin,pi*c(1/2,3/2))
```

```
## $root
## [1] 3.141593
##
## $f.root
## [1] 1.224606e-16
##
## $iter
## [1] 2
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

```r
optimise(sin,c(0,2*pi)) # 极值
```

```
## $minimum
## [1] 4.712391
##
## $objective
## [1] -1
```

```r
optimise(sin,c(0, pi),maximum = TRUE) # 极值
```

```
## $maximum
## [1] 1.570796
##
## $objective
## [1] 1
```

R 版的函数是完全向量化的，所以它已经很快了，对一个包含 100 万个元素的向量 y 进行计算，需要 8 毫秒，c++ 函数比它快 2 倍，4 毫秒，但假设编写一个 c++ 函数花费 10 分钟，那么这个函数至少使用大约 15000 次才值得重写它。c++ 函数之所以快的原因是因为与内存管理有关。

R 语言的设计限制了它的最大理论性能，慢的方面，不是因为他们的定义，而是因为他们的实现。

R 有 20 多年的历史，有近 80 万行代码，大约 45% 是 c 代码，19% 为 R 代码，17% 为 FORTRAN 代码，只有 R 的核心组成员才能修改基础的 R，目前 R 核心有 20 位，但只有 6 人活跃在日常开发中，没有一位 R 核心成员全职工作在 R 上，大多数人是统计学教授，花费较少的时间在 R 上。由于必须特别小心，避免破坏已有的代码，所以 R 核心在接受新代码上倾向于十分保守。

理解闭包

函数是 R 的一级成员，你可以给函数传递另一个函数，前面的实例创建了被命名的函数，也可以创建一个不带名字的函数，即闭包，也就是匿名函数。

```
addnum <- function(a,b){
  a + b
}   # 命名函数
addnum(2,3)
(function(a,b){
  a + b
})(2,3) #匿名函数
maxval<- function(a,b){
  (function(a,b){
    return(max(a,b))
  }
  )(a, b) #在一个函数中调用闭包
}
maxval(c(1,10,5),c(2,11))
x <- c(1,10,100)
y <- c(2,4,6)
z <- c(30,60,90)
a <- list(x,y,z)
lapply(a, function(e){e[1] * 10}) #与apply函数族类似，你也可以使用向量化计算
x <- c(1,10,100)
func <- list(min1 = function(e){min(e)}, max1 = function(e){max(e)} )
func$min1(x)
lapply(func, function(f){f(x)})#高阶函数中，把闭包当作参数来使用

x <- c(1,10,100)
y <- c(2,4,6)
z <- c(30,60,90)
a <- list(x,y,z)
sapply(a, function(e){e[1] * 10})
```

执行延迟计算 (Lazy evaluation 惰性求值)

```
test0 <- function(x,y){
if (x>0) x else y
}

test0(1)
```

```
## [1] 1
```

```
# test0(-1) # 报错

system.time(rnorm(10000000))
```

```
##    user  system elapsed
##    0.59    0.00    0.59
```

```
system.time(1)
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(test0(1,rnorm(10000000)))
```

```
##    user  system elapsed
##       0       0       0
```

惰性函数的优点是：节省时间并且避免了不必要的计算，还允许对函数的参数默认值进行更灵活的说明。

双刃剑，在调用函数时，其参数只被解析不被计算，所以我们只能确定参数表达式在语法上是正确的，但逻辑上不一定。

参数只是在某些需要的时候才会被评估。延迟计算会减少计算所需的时间。

```
lazyfunc <- function(x, y){
  x
}
lazyfunc(3)
lazyfunc2 <- function(x, y){
  x + y
}
lazyfunc2(3) #报错
lazyfunc4 <- function(x, y=2){
```

```
  x + y
}
lazyfunc4(3)

fibonacci <- function(n){
  if (n==0)
    return(0)
  if (n==1)
    return(1)
  return(fibonacci(n-1) + fibonacci(n-2))
}
# 延迟计算不使用无限循环就可以创建无限数据结构
fibonacci(10)

#当需要一个表达式的值时候，R会执行延迟计算，优点是通过避免重复计算来提升性能，以递归的方式构建无限

lazyfunc3 <- function(x, y){
  force(y) #使用force检查y是否存在
  x
}
lazyfunc3(3)
input_function <- function(x, func){
  func(x)
}
input_function(1:10, sum)
```

处理函数中的错误

在 R 中添加错误处理机制，来使程序变得更加健壮。防御性编程

```
'hello world' + 3# 错误提示

addnum <- function(a,b){
  if(!is.numeric(a) | !is.numeric(b)){
    stop("Either a or b is not numeric")
  }
  a + b
}
addnum(2,3)
addnum("hello world",3)
```

```
addnum2 <- function(a,b){
  if(!is.numeric(a) | !is.numeric(b)){
    warning("Either a or b is not numeric") #把stop换成警告看看，调用函数内部的错误消息打印

  }
  a + b
}
addnum2("hello world",3)

#把warning去掉看看发生什么
#仅仅使用warning，函数并不会中断，会继续返回a+b
options(warn=2) #抑制警告信息
addnum2("hello world", 3)

把函数封装在suppressWarnings中抑制警告，屏蔽警告信息
suppressWarnings(addnum2("hello world",3))

#用try函数捕捉错误信息
errormsg <- try(addnum("hello world",3))
errormsg

# 设定静默选项，可以抑制错误信息在控制台的展示
errormsg <- try(addnum("hello world",3), silent=TRUE)

# 使用函数try来避免
iter <- c(1,2,3,'O',5)
res <- rep(NA, length(iter))
for (i in 1:length(iter)) {
  res[i] = as.integer(iter[i]) #会有错误提示，强制类型转换
}
res

iter <- c(1,2,3,'O',5)
res <- rep(NA, length(iter))
for (i in 1:length(iter)) {
  res[i] = try(as.integer(iter[i]), silent=TRUE)
} # 加入try处理，不会打断循环
res
```

```
#使用stopinnot函数来检查参数
addnum3 <- function(a,b){
  stopifnot(is.numeric(a), !is.numeric(b))
  a + b
}
addnum3("hello", "world")



#为处理各种错误，使用更高级的函数trycatch函数来检查
dividenum <- function(a,b){
  result <- tryCatch({
    print(a/b)
  }, error = function(e) { #处理错误
    if(!is.numeric(a) | !is.numeric(b)){
      print("Either a or b is not numeric")
    }
  }, finally = {
    rm(a)
    rm(b)
    print("clean variable")
  }
  )
}
dividenum(2,4)
dividenum("hello", "world")
dividenum(1)
```

#R中的错误处理机制是通过函数实现的，而不是通过纯代码块实现的。所有的操作都是通过纯函数调用。

R中有三种基本的错误处理消息，error,warning,interrupt。

调试函数

最简单的调试方法是在期望的位置插入一条打印语句，方法有点低效。

```
debugfunc <- function(x, y){
  x <- y + 2
```

```
  x
}

debugfunc(2)#只传参数2
debug(debugfunc)#开始调试此函数
debugfunc(2) #步进，步入

undebug(debugfunc) #离开调试模式
debugfunc2(2) # 再把2传入函数，进入调试模式，键入help查
看帮助信息，按n下一步，用objects()或ls()列出所有的变量,Q 退出调试模式，也可以使用函数undebug(debug


trace(debugfunc2, quote(if(missing(y)){browser()}), at=4) # 给函数debug插入代码

debugfunc2(3)
debugfunc3 <- function(x, y){
  x <- 3
  sum(x)
  x <- y + 2
  sum(x,y)
  x
}

trace(sum) #跟踪某个函数的使用
debugfunc3(2,3)

lm(y~x)
traceback() #打印函数调用的堆栈
```

也可以用 Rstudio 来调试代码：

参考文献

1、丘祐玮著，魏博译，《数据科学：R 语言实现》，机械工业出版社，2017 年 6 月

2、哈德利. 威克汉姆，《高级 R 语言编程指南》，机械工业出版社，2016 年。