

Chapter 6 OO field guide

class就是一个object的描述，可以通过methods包中的setClass()定义

object就是一个class的实例，可以使用new()创建

generic function就是R分配在methods中的函数，generic function包含“generic”概念(plot, mean, predict,...)

method是generic function针对某一特殊class的对象的处理应用(implementation)

S3称为generic-function OO(object-oriented programming), 无明确的class定义

S4类似S3，但是拥有准确的class定义

Reference classes, 又称RC，不同于S3/4；RC implements message passing OO, so methods belong to classes, not functions。

还有一种不同与以上OO的其他系统，base types: the internal C-level types that underlie the other OO systems。

使用typeof()函数查看对象类型:

```
> f <- function(){}  
> typeof(f)  
[1] "closure"  
> typeof(sum)  
[1] "builtin"
```

function is "builtin"

S3

S3是第一个也是最简单的OO系统。仅用于base和stats包的OO系统，也是包最常用的系统。

- 识别objects, generic functions和methods:

使用 pryr::otype(), 返回对象的OO系统:

```
> df <- data.frame(x=1:10,y=letters[1:10])  
> pryr::otype(df)  
[1] "S3"  
> pryr::otype(df$x)  
[1] "base"  
> pryr::otype(df$y)  
[1] "S3"
```

同样适用 pryr::ftype(), 查看函数作用的OO系统:

```
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x10b22a4a0>
<environment: namespace:base>
> pryr::ftype(mean)
[1] "s3" "generic"
> pryr::ftype(is.numeric)
[1] "primitive" "generic"
```

- 查看generic function的所有methods:

```
> methods("mean")
[1] mean,ANY-method mean.Date      mean.default    mean.difftime   mean.POSIXct
[6] mean.POSIXlt    mean.quosure*
see '?methods' for accessing help and source code
> methods(class="ts")
[1] [      [<-      aggregate    as.data.frame cbind      coerce
[7] cycle      diff      diffinv     initialize    kernapply   lines
[13] Math      Math2     monthplot   na.omit       Ops         plot
[19] print     show      slotsFromS3 t             time        window
[25] window<-
see '?methods' for accessing help and source code
```

- 定义类和创建对象(defining classes and creating objects)

```
> foo <- structure(list(),class="foo")
> foo <- list()
> class(foo) <- "foo"
> inherits(foo,"foo")
[1] TRUE
```

直接构建, 或构建后指定, 或使用 `inherits()`

- 创建新的methods和generics

增加新的generic, 使用 `UseMethod()` 创建function。该函数拥有两个参数, generic function的名称和method dispatch的argument。若没有第二个参数, 那么将会dispatch到function的第一个参数。

没有methods的generic是没有意义的, 使用正确的(generic.class)名称来增加创建的method, 或向现存generic增加method:

```
> f <- function(x)UseMethod("f")
> f.a <- function(x)"Class a"
> a <- structure(list(),class="a")
> class(a)
[1] "a"
> f(a)
[1] "Class a"
> mean.a <- function(x)"a"
> mean(a)
[1] "a"
```

S3的Method dispatch相对简单的, 使用函数 `UseMethod()` 创建一个函数名称向量, 类似 `paste0(generic,".",c(class(x),"default"))`, 然后generic会根据class返回:

```

> f <- function(x)UseMethod("f")
> f.a <- function(x)"Class a"
> f.default <- function(x)"Unknown class"
> f(structure(list(),class="a"))
[1] "Class a"
> f(structure(list(),class=c("b","a")))
[1] "Class a"
> f(structure(list(),class="c"))
[1] "Unknown class"

```

S4

通过 `str()` 识别S4对象, `isS4()`, `pryr::otype()` 用于判断类型

S3可以通过简单改变class属性来改变任何对象的类型, S4需要严格定义: must define the representation of the class using `setClass()`, and create a new object with `new()`.

- a name: 字符数字类别识别符, S4类别名称应该使用大写
- a named list of slots(fields), 提供slot名称和所允许的类别。例如: 人的类别应该由字符名称和数字年龄来代表: `list(name="character",age="numeric")`
- a string giving the class it inherits from, or in S4 terminology, that it contains.

此外, S4类别可以设置validity method来检测object的有效性, 和prototype设置slot的默认值。

```

> setClass("Person",
+         slots=list(name="character",age="numeric"))
> setClass("Employee",
+         slots=list(boss="Person"),contains="Person")
> alice <- new('Person',name="Alice",age=40)
> john <- new("Employee",name="John",age=20,boss=alice)
> alice@age
[1] 40
> slot(john,"boss")
An object of class "Person"
Slot "name":
[1] "Alice"

Slot "age":
[1] 40

```

假如S4 object包含(继承)S3类或一个base type, 它将拥有一个特殊的 `.Data` slot, 包含潜在的base type或S3 object:

```

> setClass("RangedNumeric",
+         contains="numeric",
+         slots=list(min="numeric",max="numeric"))
> rn <- new("RangedNumeric",1:10,min=1,max=10)
> rn@min
[1] 1
> rn@.Data
[1] 1 2 3 4 5 6 7 8 9 10

```

S4使用特殊函数构建generics和methods: `setGeneric()` 构建新的generic或将现存函数转换进入一generic; `setMethod()` takes the name of the generic, the classes and method should be associated with and a function that implements the method.

例如: 现有union仅用于vectors 类对象, 现将其应用于data.frame:

```
> setGeneric("union")
[1] "union"
> setMethod("union", c(x="data.frame", y="data.frame"),
+           function(x,y){
+             unique(rbind(x,y))
+           })
> |
```

如要构建新的generic，需要提供 `standardGeneric()`：

```
> setGeneric("myGeneric", function(x){
+   standardGeneric("myGeneric")
+ })
[1] "myGeneric"
```

The main difference is how you set up default values: S4 uses the special class 'ANY' to match any class and 'missing' to match a missing argument.

```
> selectMethod("nobs", list("mle"))
Method Definition:

function (object, ...)
if ("nobs" %in% slotNames(object)) object@nobs else NA_integer_
<bytecode: 0x10876fc80>
<environment: namespace:stats4>

Signatures:
      object
target  "mle"
defined "mle"
> prvr::method_from_call(nobs(fit))
Method Definition:

function (object, ...)
if ("nobs" %in% slotNames(object)) object@nobs else NA_integer_
<bytecode: 0x10876fc80>
<environment: namespace:stats4>

Signatures:
      object
target  "mle"
defined "mle"
```

RC

Reference classes(or RC for short) are the newest OO system in R, introduced in 2.12. They are functionally different to S3 and S4 because:

- RC methods belong to objects, not functions
- RC objects are mutable: the usual R copy-on-modify semantics do not apply

略!!!

Chapter 7 Environments

环境(environments)的工作就是关联或者绑定一系列名称到值。

环境就是power scoping的数据结构，环境类似于列表，具有三个重要的例外情况：

- 修改一个环境，同时也会修改其每一个拷贝的环境
- 环境具有父环境，如果一个对象在一个环境中没有找到，那么R就会在其父环境中查看，依次类推
- 每个环境中的对象都必须有一个名称，且名称必须是唯一的

技术上而言，环境就是一个修饰的框架，收集一系列命名了的对象(像一个列表)，和一个reference指向一个父环境(parent environment)

可以通过 `new.env()` 构建环境，查看其内容 `ls()`，检查其父环境 `parent.env()`

```
> e <- new.env()
> parent.env(e)
<environment: R_GlobalEnv>
> identical(e, globalenv())
[1] FALSE
> ls(e)
character(0)
```

可像修改列表一样修改环境：

```
> e$a <- 1
> ls(e)
[1] "a"
> e$a
[1] 1
> e$.b <- 2
> ls(e)
[1] "a"
> ls(e, all=TRUE)
[1] ".b" "a"
```

可使用 `$`，`[[` 或 `get` 获得环境内对象，`$` 和 `[[` 仅在环境中查找，但是 `get` 使用regular scoping rules 并且也在其父环境中查找

```
> e$b <- 2
> e$b
[1] 2
> e[["b"]]
[1] 2
> get("b", e)
[1] 2
```

从环境中删除对象类似于 `list`，不同的是 `list` 是设置为 `NULL`，而环境为 `rm()`

一般而言，当创建自己的环境时，想要手动设置父环境为空环境。这保证不会意外的从其他环境继承一些对象：


```

> x <- 1
> e1 <- new.env()
> get("x",e1)
[1] 1
> e2 <- new.env(parent=emptyenv())
> get("x",e2)
Error in get("x", e2) : object 'x' not found
> ls(envir = new.env())
character(0)

```

查看对象是否存在某一环境，可设置不查看父环境：

```

> exists("b",e)
[1] TRUE
> exists("b",e,inherits = FALSE)
[1] TRUE
> exists("b",e,inherits = FALSE)
[1] TRUE

```

查看特殊环境：

`globalenv()`：用户的工作环境

`baseenv()`：base package的环境

`emptyenv()`：所有环境的最终初始环境，唯一没有父环境的环境

最常见的环境为global environment(`globalenv()`)，对应top-level环境，其父环境就是用户所加载的一个package，其顺序取决于加载包的顺序。然后最终的父环境为base environment，对应'base R'的环境，它的父环境为empty environment。

`search()` 列出所有global和base环境：

```

> search()
[1] ".GlobalEnv"          "package:stats4"      "tools:rstudio"
[4] "package:stats"        "package:graphics"    "package:grDevices"
[7] "package:utils"        "package:datasets"    "package:methods"
[10] "Autoloads"           "package:base"

```

可以访问任何search list中的环境：

```

> as.environment("package:stats")
<environment: package:stats>
attr("name")
[1] "package:stats"
attr("path")
[1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/stats"
>

```

借助 `pryr` 包的 `where()` 函数查看相关信息(the environment where the function lives)：

```

> where("where")
<environment: package:pryr>
attr(,"name")
[1] "package:pryr"
attr(,"path")
[1] "/Users/carlos/Library/R/3.5/library/pryr"
> where("mean")
<environment: base>
> where("t.test")
<environment: package:stats>
attr(,"name")
[1] "package:stats"
attr(,"path")
[1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/stats"

```

递归查询 `globalenv()` 的 `ancestor` 环境包含 `baseenv()` 和 `emptyenv()` :

```

F <- function(env=parent.frame()){
  if(identical(env,emptyenv())){
    return(emptyenv())
  }else{
    tmp=1
  }
  if(tmp){
    print(env)
    F(env=parent.env(env))
  }
}
F()

```

函数环境(function environments)

一个函数往往关联的多个环境:

函数function被创建时的环境environment, 当一个function被创建时, 获得function创建时的环境, 使用 `environment()` 访问function:

```

> library(pryr)
> y <- 1
> f <- function(x)x+y
> environment(f)
<environment: R_GlobalEnv>
> where("f")
<environment: R_GlobalEnv>
> environment(plot)
<environment: namespace:graphics>
> environment(t.test)
<environment: namespace:stats>
> where("plot")
<environment: package:graphics>
attr(,"name")
[1] "package:graphics"
attr(,"path")
[1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/graphics"

```

闭包(closure):闭包就是能够读取其他函数内部变量的函数。例如在javascript中, 只有函数内部的子函数才能读取局部变量, 所以闭包可以理解成“定义在一个函数内部的函数”。在本质上, 闭包是将函数内部和函数外部连接起来的桥梁。

```

> plus <- function(x){
+   function(y)x+y
+ }
> plus_one <- plus(1)
> plus_one(10)
[1] 11
> plus_two <- plus(2)
> plus_two(10)
[1] 12
> environment(plus)
<environment: R_GlobalEnv>
> environment(plus_one)
<environment: 0x11041ec48>
> parent.env(environment(plus_one))
<environment: R_GlobalEnv>

```

同样可以修改一个函数function的环境，使用 `environment()`，借以阐述R的基本作用域：

```

> f <- function(x)x+y
> environment(f) <- emptyenv()
> f(1)
Error in x + y : could not find function "+"

```

函数每次运行时都会创建一个新环境供host执行：

```

> f <- function(){
+   list(
+     e <- environment(),
+     p <- parent.env(environment())
+   )
+ }
> str(f())
List of 2
 $ :<environment: 0x110f1f2b8>
 $ :<environment: R_GlobalEnv>
> str(f())
List of 2
 $ :<environment: 0x110f67db8>
 $ :<environment: R_GlobalEnv>
> funenv("f")
<environment: R_GlobalEnv>

```

函数调用时的环境，可通过 `parent.frame()` 来返回

```

> f2 <- function(){
+   x <- 10
+   function(){
+     def <- get("x",environment())
+     cll <- get("x",parent.frame())
+     list(defined=def,called=cll)
+   }
+ }
> g2 <- f2()
> x <- 20
> str(g2())
List of 2
 $ defined: num 10
 $ called : num 20

```


使用 `local()` 明确作用域，创建一个作用域而不是通过植入到一个函数内 `function`，若需要临时变量，然后用后舍弃，这样不会污染环境内的变量：

```
df <- local({  
  x <- 1:10  
  y <- runif(10)  
  data.frame(x=x, y=y)  
})  
##equivalent to  
df <- (function(){  
  x <- 1:10  
  y <- runif(10)  
  data.frame(x=x, y=y)  
})()
```

`local()` 函数相对有限的用处，但是和 `<-` 联合使用时，实现私有变量的共享：

```
> a <- 10  
> my_get <- NULL  
> my_set <- NULL  
> local({  
+   a <- 1  
+   my_get <- function()a  
+   my_set <- function(value)a <- value  
+ })  
> my_get()  
[1] 1  
> my_set(13)  
> my_get()  
[1] 13
```

赋值(assignment: binding names to values)

常规赋值(regular binding)，实现当前环境名称和值的关联。语法性赋值(syntactic)和非语法性赋值(no-syntactic)。前者必须以字母或 `.` 开头(不能后接数字 `._1` wrong)：`a <- 1`，`._ <- 2`，`a_b <- 3`；非语法性赋值时，名称可以是任意字符串：`a+b <- 3`；`:) <- "simle"`；`<- "space"`

```
> ` ` <- "space"  
> `a+b` <- 3  
> `:)` <- "simle"  
> ls()  
[1] " " " :)" "a+b"
```

这里赋值都是发生在当前环境，可以通过三种方式向其他环境赋值：

使用列表方式向环境赋值

```
e <- new.env()
```

```
e$a <- 1
```

使用 `assign()` 函数，三个参数：名称，值，环境：

```
e <- new.env()
```

```
assign("a", 1, envir=e)
```

使用 `eval()` 或 `evalq()` 在环境内赋值:

```
e <- new.env()
```

`eval(quote(a<-1),e)` 等同于 `evalq(a <- 1,e)`

常量(constants), 是不能被改变的变量, 只能被赋值一次, 不能再次赋值, `lockBinding()` 用于在 package 内修改对象:

```
> x <- 10
> lockBinding(as.name("x"),globalenv())
> x <- 15
Error: cannot change value of locked binding for 'x'
```

```
> x %<c-% 20
> x <- 30
Error: cannot change value of locked binding for 'x'
```

`mean <- 4` 可行, 这里是 syntactic, 若 non-syntactic:

```
> assign("mean", function(x) sum(x)/length(x), env=baseenv())
Error in assign("mean", function(x) sum(x)/length(x), env = baseenv()) :
  cannot change value of locked binding for 'mean'
```

`<<-`, 另一种修改 name 和 value 的 binding 的方式. 常规命名值, `<-` 会在当前环境创建一个新变量. `<<-` 不会在当前环境创建新变量, 而是向上在启父环境中修改已经存在的变量, 若 `<<-` 没有发现存在的变量, 将在 global 环境创建一个:

```
> x <- 0
> f <- function(){
+   g <- function(){
+     x <<- 2
+   }
+ }
> x <- 1
> g()
+ x
+ }
> f()
[1] 2
> x
[1] 0
```

因此: `name <<- value` 等价于 `assign("name",value,inherits=TRUE)`

Delayed bindings

构建并存储一个 promise 用于评估表达式, 而不是立即赋给表达式一个结果, 使用 `pryr` 包中的 `%<d-%` 实现:

```
> a %<d-% 1
> system.time(a)
  user  system elapsed
    0      0      0
> b %<d-% {Sys.sleep(1);1}
> system.time(b)
  user  system elapsed
0.003  0.003  1.000
```

Active bindings

每次访问该值名称是重新计算该值 `%<a-%`:

```
> x %<a-% runif(1)
> x
[1] 0.4632384
> x
[1] 0.6902952
```

Chapter 8 Debugging, condition handling and defensive programming

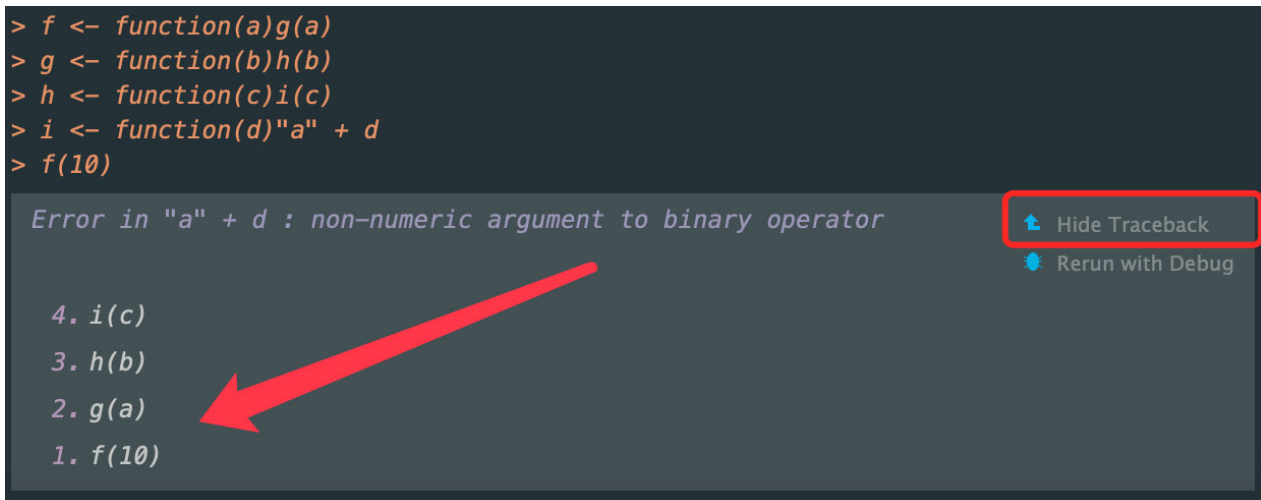
- Fatal errors are raised by `stop()` and force all execution to terminate
- Warnings are generated by `warning()` and are used to display potential problems
- Messages are generated by `message()` and are used to give informative output in a way that can easily be suppressed by the user(`suppressMessages()`)

Debugging tools

- The Rstudio error inspector and `traceback()` which list the sequence of calls that lead to the error
- Rstudio's 'Return on debug' tool and `options(error=browser)` which enter an interactive session where the error occurred
- Rstudio's breakpoints and `browser()` which enter an interactive session at an arbitrary code location

```
> f <- function(a)g(a)
> g <- function(b)h(b)
> h <- function(c)i(c)
> i <- function(d)"a" + d
> f(10)

Error in "a" + d : non-numeric argument to binary operator
```



4. i(c)
3. h(b)
2. g(a)
1. f(10)

Hide Traceback
Rerun with Debug

若不是使用Rstudio, 可使用`traceback()`以获得同样信息

```
traceback()
# 4: i(c) at error.R#3
# 3: h(b) at error.R#2
# 2: g(a) at error.R#1
# 1: f(10)
```

Browsing arbitrary code

