R 语言面向对象编程

徐静

2018-08-06

Contents

写在书前

为了理解 R 中的计算，下面的两个口号是有用的

> 一切皆是对象
>
> 一切皆是函数调用
>
> from John Chambers

1 年半之前就有计划整理一份 R 语言面向对象编程的详细文档，今天终于有了一点眉目。从 2016 年初开始研究 R 语言中的一些黑魔法，一直在尝试 R 语言的面向对象的编程，本人经历了 Java 和 Python 面向对象编程的洗礼，加深了对 R 语言面向对象编程的深入理解。本电子书是我在应用 Java 和 Python 面向对象编程后，对 R 语言面向对象编程的进一步理解的学习文档，希望对 R 社区的小伙伴有帮助。

所有代码本人均在 Ubuntu 16.04 LTS 和 CentOs6.5 下尝试正常通过。

声明

- 关于我

徐静：

硕士研究生, 目前的研究兴趣主要包括：数理统计，统计机器学习，深度学习，网络爬虫，前端可视化，R 语言和 Python 语言的超级粉丝，多个 R 包和 Python 模块的作者，现在正逐步向 Java 迁移。

Graduate students,the current research interests include: mathematical statistics, statistical machine learning, deep learning, web crawler, front-end visualization. He is a super fan of R and Python, and the author of several R packages and Python modules, and now gradually migrating to Java.

- 声明

本书内容并非笔者原创，而是对参考文献 [1-3] 学习之后的整理再现，特此说明。本电子书完全免费，但转载或用于其他商业用途请说明来源：https://dataxujing.github.io/R_oop/

> 学 R 不思则罔，思 R 不学则殆。

Chapter 1

R 语言编程风格指南

好的编程风格 (程序编写格式) 就像正确使用标点符号。没有他也可以，但是他会让你编写的程序易于理解。建议参考 Google R 语言编程风格指南 (https://nanx.me/rstyle/)。虽然你的代码只有一个作者，但通常有许多读者，所以好的风格是很重要的。当你和他人合作编写代码时，尤其要注意编程风格，但是要注意没有一种风格说是最优的，但是你们需要统一。YiHuiXie 编写的 formatR 添加包使得整理格式混乱的代码变得容易，但是不能解决一切问题。

1.1 符号和名字

1.1.1 文件名

文件名应该有一定的意义，并且以.R 结尾

```
#good

fit-model.R
utility-function.R

#bad

foo.r
stuff.r
```

如果文件需要按照顺序执行，那么做好在命名时加上数字前缀

```
0-download.R
1-parse.R
2-explore.R
```

1.1.2 对象名

在计算机科学中有两件非常难的事情，缓存失效和对象命名。变量和函数名应该是小写字母，使用下划线 (_) 将名字中的单词分开。通常变量名应该是名词，函数名应该是动词。追求简洁而有意义的名字。

```
#good

day_one
day_1
```

```
#bad

first_day_of_month
DayOne
dayone
djm1
```

1.2 语法

1.2.1 空格

所有中缀运算符 (=,+,-,<-等) 的两边使用空格。要在逗号的后面而非前面加上空格

```
#good

average <- mean(feet / 12 + inches, na.rm = TRUE)

#bad

average<-mean(feet/12+inches,na.rm=TRUE)
```

这个规则有个小小的例外，:,::,::: 的两边不需要空格

```
#good

x <- 1:10
base::get

#bad

x <- 1:10
base :: get
```

在小括号的左边放置一个空格，函数调用例外

```
#good

if (debug) do(x)
plot(x, y)

#bad

if(debug)do(x)
plot (x, y)
```

如果是为了对齐等号或赋值符号，也可以使用额外的空格

```
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

小括号或中括号内代码的两侧不需要放置空格 (除非有逗号，要在逗号的后边放置空格)

```
#good

if (debug) do(x)
diamonds[5, ]

#bad

if ( debug ) do(x)
x[1,]
x[1, ]
```

1.2.2 大括号

大括号的左半边不能独占一行，它后边也应该新起一行，右半边应该独占一行或它的后边是 else。对大括号中的代码要缩进

```
#good

if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y^x
}


#bad

if (y <= 0 && debug)
message("Y is negative")
if (y == 0) {
  log(x)
}
else {
  y^x
}
```

可以在同一行上留下非常短的语句：

```
if (y < 0 && debug) message("Y is message")
```

1.2.3 行的长度

努力使每行代码不超过 80 个字符，如果你的代码超过了这个长度，这说明你应该将一些功能打包成独立的函数。

1.2.4 缩进

缩进代码使用两个空格，不要使用制表符 (tab) 或者制表符加空格。唯一的例外是，如果一个函数定义跨越了很多行。这种情况下，第二行要缩进到定义开始的地方:

```
long_function_name <- function (a = "a long argument",
                                b = "another argument",
                                c = "another long argument") {
  # As usual code is idented by two spaces.
}
```

1.2.5 赋值

赋值时使用 <- 而不是 =。

```
#good

x <- 5

#bad

x = 5
```

1.3 结构

注释指南

对代码进行注释，每行的注释都应该以注释符和一个空格开始: #。注释应该解释为什么。而不是是什么。

使用由 -和 = 构成的注释行将文件分割成容易理解的段落 (块)

```
# Load Data --------------------------------------------

# Plot Data --------------------------------------------
```

好的程序猿一定有自己的编程风格，并且在团队中能够快速的调整使得编程风格达成一致，所以与其他人合作意味着你可能需要对自己喜好的某些风格做出让步。

Chapter 2

R 语言面向对象编程指南

面向对象是一种对世界理解和抽象的方法，当代码复杂度增加难以维护的时候，面向对象就会显得很重要，我经历过 Java 和 Python 两种语言从面向过程到面向对象的改造，对 R 的面向对象的编程也早有些研究但开始的时候并不是那么的透彻。随着大数据时代 AI 时代的来临，R 将走向大规模的企业级应用，因此面向对象的编程方式也会将成为 R 语言的一种非常重要的趋势，并且多位 R 语言大神，像 Hadley Wickham 等在 R 包开发中早就引入了面向对象的编程方式，R 语言的发展也势必会面向对象。

2.1 什么是面向对象

学过计算机编程基础的人都知道，计算机是通过接受一些逻辑指令，然后翻译成机器码，进而控制 CPU 的电路，从而实现我们能看到的所有操作。无论是何种计算机编程语言，都可以认为是计算机和人类沟通的翻译，将一般人能懂的计算机语言翻译成计算机能懂的机器语言。

简单的理解，有的语言比较接近机器的习惯，机器执行起来会更有效率；有的语言比较接近人的习惯，人类设计起来会更容易。面向对象的思想就属于第二种情况。

人的思维方式和计算机是不同的，计算机习惯按照顺序执行不同的指令，依据严格的逻辑进行不同的行为，而人类处理问题的方式通常是先对问题进行分析，然后调动不同的资源做不同的事情。

喜欢历史故事的朋友都知道诸葛亮打仗和刘邦打仗的区别。诸葛亮会命令某人埋伏，某人放火，某人举旗，某人不战而退，某人斜刺里杀出，每个人听到命令都不知道最后会发生什么。直到最后敌军被一条龙的歼灭后，得胜归来的将军们对诸葛亮佩服的五体投地。而刘邦打仗的故事远没有那么精彩，事情来了该让韩信做的交给韩信，该让萧何做的交给萧何。

如何像刘邦一样的写程序，这就是面向对象的程序设计。韩信，萧何这些人都可以认为是对象，我们只需要知道他们有什么特点，根据问题的不同派不同的人去做就可以了，而诸葛亮关注的是具体流程，至于是关羽去放火还是张飞去放火反而不重要了，这就是过程式的编程思想。

面向对象的程序设计在运行效率上可能没有优势，但是节约了开发者和设计者的时间，在 R 语言和 S 语言上这一点完全一致，S 语言很重要的设计理念是 “人的时间远比机器的时间宝贵”。对各种模型和算法的封装及重用与面向对象的编程目的是相同的。

在面向对象的程序设计中，对象 (object) 是最基本的元素，不过对象指的是具体的实例，在对象之上还有一个类 (class) 的概念。这里的类和 R 中的类的概念没有任何不同，都是指某一种抽象对象的类型 (和 R 中的 type 不同,type 指的是在内存存储方面的类型)。

比如说 “马” 就是一个类，随便牵来一匹白马或红马都属于马这个范畴，但都和马这个东西不一样，如果牵来的白马是刘备的的卢马，红马是关羽的赤兔马，那么这两匹马就是对象。所以类是抽象的概念，对象是类的具体实例。我们直接操作的是对象，但是需要定义的是类。

用面向对象的专业术语来说，马就是一个 “类”，白马和红马是马的 “子类”，的卢马是白马实例化的对象，也是马实例化的对象。

一般来说类包含属性和方法，属性指的是类具有的某些信息，在计算机程序中通常是变量，方法指的是类进行的操作，在计算机程序中相当于函数。

并不是具有了类和对象的概念后就成了面向对象的程序设计，一般来说还得具备三个特性：封装、继承和多态。

封装指的是隐藏对象的实现细节，仅对外公开接口，每个对象都可以独立的完成一定的功能，不需要和其他对象有过多的交互，所有的数据交换都通过接口来处理，专业术语是降低系统的“耦合度”。

比如说马这种交通工具就是一个封装的很好的类，具有颜色、体重等属性，具有载人、奔跑等方法。用的时候把马牵出来，通过缰绳和马鞭这几个接口来控制动作。

继承是一个类可以继承另一个类的各种属性及方法，重写或增加某些属性和方法，被继承的类称为“父类”，继承了父类的类的称为“子类”。通过继承可以重写父类方法或增加额外功能，注意 R 语言像 Python 一样支持多重继承，Java 不支持多继承，但是有其他办法实现。当然除了继承也可以组合类。

多态可以说是面向对象的程序设计中最关键的特性，如果某种语言支持以上所有特性但是不支持多态，我们称其为“基于对象”而不是面向对象，所谓多态简单来说就是希望能用相同的命令作用于不同的类，根据类的不同产生不同的结果。

```
# 作用在数值数据
summary(rnorm(10))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.0652 -0.7036 -0.2353  0.2621  1.2058  2.2533
```

```
# 作用在 Model 上
summary(lm(rnorm(10)~rnorm(10)))
```

```
##
## Call:
## lm(formula = rnorm(10) ~ rnorm(10))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.4959 -0.5513 -0.3392  0.6401  1.7993
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.1801     0.3618  -0.498    0.631
##
## Residual standard error: 1.144 on 9 degrees of freedom
```

2.2 R 为什么要进行面向对象的编程

R 主要面向统计计算，而且代码量一般不会很大，几十行，几百行，使用面向过程的编程方式就可以很好的完成编程任务。在 R 中流传着一个深入人心的说法“万物皆对象”，另一方面，R 是一种函数式的语言，与面向对象的程序设计存在着天然的差异，实际上这两个对象的描述的含义是不同的。“万物皆对象”是指 R 的基本数据结构的地位都是相同的，任何东西包括函数都可以认为是对象，都能作为参数传入到函数中。而“面向对象”指的是一种编程泛型，已经成为一个专有的名词。

但是随着 R 语言在工业界的火热，伴随着越来越多的工程背景的人的加入，R 语言开始向更多领域发展，会有越来越难以维护的海量代码项目，所以必须使用面向对象的编程思想，此外如果你开发一些 R 包需要对特殊的对象重定义 S3(下一章会讲) 的方法，需要对大量程序代码最大化的重用和封装 (S4(后面会讲到))，那么这时候你同样需要用到 R 语言的面向对象的编程。

一句话，随着 R 语言的发展，R 面向对象编程一定是一个大的趋势。

2.3 R 的面向对象编程

R 的面向对象是基于泛型函数 (generic function) 的，而不是基于类层次结构，接下来我们从面向对象的 3 个特征入手，分别用 R 语言进行实现。

- 封装

```
# 定义 teacher 对象和行为
teacher <- function(x,...) UseMethod("teacher")
teacher.lecture <- function(x,...) print(" 上课")
teacher.assignment <- function(x,...) print(" 布置作业")
teacher.correcting <- function(x,...) print(" 批改作业")
teacher.default <- function(x,...) print(" 你不是 teacher")

# 定义同学对象和行为

student <- function(x,...) UseMethod("student")
student.attend <- function(x,...) print(" 听课")
student.homework <- function(x,...) print(" 写作业")
student.exam <- function(x,...) print(" 考试")
student.default <- function(x,...) print(" 你不是 student")

# 定义两个变量，a 老师和 b 同学

a <- 'teacher'
b <- 'student'

# 给老师变量设置行为

attr(a,"class") <- 'lecture'
# 执行老师的行为

teacher(a)
```

```
## [1] "上课"
```

```
attr(b,'class') <- 'attend'
student(b)
```

```
## [1] "听课"
```

```
attr(a,'class') <- 'assignment'
teacher(a)
```

```
## [1] "布置作业"
```

- 继承

```
# 给同学对象增加新的行为
student.correcting <- function(x) print(" 帮助老师批改作业")

# 辅助变量用于设置初始值
char0 = character(0)
# 实现继承关系
create <- function(classes=char0,parents=char0){
  mro <- c(classes)
  for(name in parents){
    mro <- c(mro,name)
```

```
    ancestors <- attr(get(name),'type')
    mro <- c(mro,ancestors(ancestors != name))

  }

  return(mro)
}

# 定义构造函数，创建对象
NewInstance <- function(value=0,classes=char0,parents=char0) {
  obj <- value
  attr(obj,'type') <- create(classes,parents)
  attr(obj,"class") <- c('homework','correcting','exam')
  return(obj)
}

# 创建对象实例
StudentObj <- NewInstance()
# 创建子对象实例
s1 <- NewInstance(" 普通同学",classes = 'normal',parents = "StudentObj")
s2 <- NewInstance(' 课代表',classes='leader',parents='StudentObj')

# 给课代表，增加批改作业行为
attr(s2,'class') <- c(attr(s2,'class'),'correcting')
s1
s2
```

- 多态

```
# 创建优等生和次等生，两个实例
e1 <- NewInstance(" 优等生",classes='excellent',parents='StudentObj')

e2 <- NewInstance(" 次等生",classes='poor',parents='StudentObj')

student.exam <- function(x,score){
  p <- ' 考试'
  if(score>85) print(paste(p," 优秀"))
  if(score<70) print(paste(p," 及格"))

}

# 执行优等生的考试行为，并输入分数为 90
attr(e1,'class') <- 'exam'
stuent(e1,90)
[1] "考试优秀"

attr(e2,'class') <- 'exam'

student(e2,66)
[1] " 考试及格"
```

通过 R 语言的面向对象的反省函数，我们就可以实现面向对象的编程。

Chapter 3

基于 S3 的面向对象编程

对于 R 语言的面向对象编程，不同于其他编程语言，R 语言提供了 3 种底层对象类型，一种是 S3 类型，一种是 S4 类型，还有一种是 RC 类型。

S3 对象简答，具有动态性，结构化特征不明显，S4 对象结构化。功能强大，RC 对象是 R2.12 版本后使用的新类型，用于解决 S3,S4 很难是想的对象。

本章主要介绍 S3 的面向对象编程的细节

3.1 S3 对象的介绍

在 R 语言中，基于 S3 对象的面向对象编程，是一种基于泛型函数的实现方式。泛型函数是一种特殊的函数，根据传入对象的类型决定调用那个具体的方法。基于 S3 对象实现面向对象编程，不同其他语言的面型对象编程，是一种动态函数调用的模拟实现。S3 对象被广泛应用于 R 的早期的开发包中。

3.2 创建 S3 对象

注意：本文会用到 pryr, 为了方便我们检查对象的类型，引入 pryr 包作为辅助工具。

```
library(pryr)

# 通过变量创建 S3 对象

x <- 1
attr(x,'class') <- 'foo'
x
```

```
## [1] 1
## attr(,"class")
## [1] "foo"
```

```
attr(x,"class")
```

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

```
class(x)
```

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

```
# 用 pryr 包的 otype 函数, 检查 x 的类型
otype(x)
```

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

通过 structure() 函数创建 S3 对象

```
y <- structure(2,class="foo")

y
```

```
## [1] 2
## attr(,"class")
## [1] "foo"
```

```
attr(y,"class")
```

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

```
class(y)
```

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

```
otype(y)
```

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

创建一个多类型的 S3 对象，S3 独享没有明确结构关系，一个 S3 对象可以有多个类型，S3 对象的 class 属性可以是一个响亮，包括多种类型

```
x <- 1
attr(x,"class") <- c("foo","bar")
class(x)
```

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

```
otype(x)
```

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

3.3 泛型函数和方法调用

对于 S3 对象的使用，通常用 UseMethod() 函数来定义一个泛型函数的名称，通过传入参数的 class 属性，来确定方法调用。

定义一个 teacher 的泛型函数

- 用 UseMethod() 定义 teacher 泛型函数
- 用 teacher.xxx 的语法格式定义 teacher 对象的行为
- 其中 teacher.default 是默认行为

```
# 用 UseMethod() 定义 teacher 泛型函数
teacher <- function(x,...) UseMethod("teacher")
# 用 pryr 包中 ftype() 函数，检查 teacher 类型
ftype(teacher)
[1] "s3" "generic"

# 定义 teacher 内部函数

teacher.lecture <- function(x,...) print(" 讲课")
teacher.assignment <- function(x,...) print(" 布置作业")
```

```
teacher.correcting <- function(x,...) print(" 批改作业")
teacher.default <- function(x,...) print(" 你不是 teacher")
```

方法调用通过传入参数的 class 属性，来确定不同方法调用

- 定义一个变量 a，并设置 a 的 class 属性为 lecture
- 把变量 a 传入到 teacher 泛型函数中
- 函数 teacher.lecture() 函数的行为被调用

```
a <- "teacher"
# 给老师变量设置行为
attr(a,"class") <- 'lecture'
# 执行老师的行为
teacher(a)
[1] "讲课"
```

当然我们可以直接调用 teacher 中定义的行为，如果这样做就失去了面向对象封装的意义

```
teacher.lecture()
[1] " 讲课"
teacher.lecture(a)
[1] " 讲课"
teacher()
[1] " 你不是 teacher"
```

3.4 查看 S3 对象的函数

当我们使用 S3 队形进行面向对象封装后，可以使用 methods() 函数来查看 S3 对象中的定义的内部行为函数。

```
# 查看 teacher 对象
> teacher
function(x,...) Usemethod("teacher")

# 查看 teacher 对象的内部函数
> methods(teacher)
[1] teacher.assignment teacher.correcting teacher.default teacher.lecture

# 通过 methods() 的 generic.function 参数，来匹配泛型函数名字
> methods(generic.function = predict)
[1] predict.ar* ......
```

通过 methods() 的 class 参数，来匹配类的名字

```
> methods(class=lm)
[1]add1.lm* ......
```

用 getAnywhere() 函数, 查看所有函数

```
# 查看 teacher.lecture 函数

>getAnywhere(teacher.lecture)
```

使用 getS3method() 函数，也同样可以查看不可见的函数

```
# getS3method() 函数查找 predict.ppr

get 时method("predict","ppr")
```

3.5 S3 对象的继承关系

S3 独享有一种非常简单的继承方式，用 NextMethod() 函数来实现。

定义一个 node 泛型函数

```
> node <- function(x) UseMethod("node",x)
> node.default <- function(x) "Default node"

#father 函数
> node.father <- function(x) c("father")

# son 函数，通过 NextMethod() 函数只想 father 函数
> node.son <- function(x) c('son',NextMethod())

# 定义 n1
> n1 <- structure(1,class=c("father"))
# 在 node 函数中传入 n1, 执行 node.father() 函数
> node(n1)
[1] "father"

# 定义 n2，设置 class 属性为两个
> n2 <- structure(1,class=c("son","father"))
# 在 node 函数中传入 n2, 执行 node.son() 函数和 node.father() 函数
> node(n2)
[1] "son" "father"
```

通过对 node() 函数传入 n2 的参数，node.son() 先被执行，然后通过 NextMethod() 函数继续执行了 node.father() 函数。这样其实就模拟了，子函数调用父函数的过程，实现了面向对象编程中的继承。

3.6 S3 对象的缺点

从上面 S3 对象的介绍上来看，S3 对象并不是完全的面向对象实现，而是一种通过泛型函数模拟的面向对象的实现。

- S3 用起来简单，但在实际的面向对象编程的过程中，当对象关系有一定的复杂度，S3 对象所表达的意义就变得不太清楚
- S3 封装的内部函数，可以绕过泛型函数的检查，以直接被调用
- S3 参数的 class 属性，可以被任意设置，没有预处理的检查
- S3 参数，只能通过调用 class 属性进行函数调用，其他属性则不会被 class() 函数执行
- S3 参数的 class 属性有多个值时，调用时会被按照程序赋值顺序来调用第一个合法的函数

所以，S3 只是 R 语言面向对象的一种简单的实现。

3.7 S3 对象的使用

S3 对象系统，被广泛的应用于 R 语言早期的开发中。在 base 包中，就有很多 S3 对象

base 包的 S3 对象

```
# mean 函数
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x000000001b953ed8>
## <environment: namespace:base>
```

```
ftype(mean)
```

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

```
# t 函数
ftype(t)
```

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

```
# plot 函数
ftype(plot)
```

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

自定义 S3 对象

```
# 定义数字变量 a

a <- 1
# 变量 a 的 class 为 number
class(a)
```

```
## [1] "numeric"
```

```
# 定义泛型函数 f1
f1 <- function(x){
  a <-2
  UseMethod("f1")
}
# 定义 f1 的内部函数
f1.numeric <- function(x) a

# 给 f1() 传入变量 a
f1(a)
```

```
## [1] 2
```

```
# 给 f1() 传入 99
f1(99)
```

```
## [1] 2
```

```
# 定义 f1 内部函数
f1.character <- function(x) paste("char",x)

# 给 f1() 传入字符 a
f1("a")
```

```
## [1] "char a"
```

这样，我们就对 S3 对象系统有了一个全面认识，开始 R 语言的面向对象编程之路。

Chapter 4

基于 S4 的面向对象编程

S4 对象系统具有明显的结构化特征，更适合面向对象的程序设计。Bioconductor 社区以 S4 对象作为基础框架，只接受 S4 定义的 R 包。

4.1 S4 对象介绍

S4 对象系统是一种标准的 R 语言面向对象实现方式，S4 对象有明确的类定义，参数定义，参数检查，继承关系，实例化等的面向对象系统的特征。

4.2 创建 S4 对象

使用辅助的 pryr 包

```
# 加载 pryr 包
library(pryr)
```

4.2.1 如何创建 S4 对象?

由于 S4 对象是标准的面向对象实现方式，有专门的类定义函数 setClass() 和类的实例化函数 new()，我们看一下 setClass() 和 new() 是如何工作的。

- setClass()

```
setClass(class,representation,prototype,contains=character(),
  validity,access,where,version,sealed,package,
  S3methods=FALSE,slots)
```

参数列表：

- Class: 定义类名
- slots: 定义属性和属性类型
- prototype: 定义属性的默认值
- contains=character(): 定义父类，继承关系
- validity: 定义属性的类型检查
- where: 定义存储空间

- sealed: 如果设置 TRUE, 则同名类不能再次定义
- package: 定义所属的包
- S3methods:R3.0.0 后不建议使用
- representation:R3.0.0 后不建议使用
- access:R3.0.0 后不建议使用
- version:R3.0.0 后不建议使用

4.2.2 创建一个 S4 对象实例

```
# 定义一个 S4 对象

setClass("Person",slots=list(name="character",age="numeric"))

# 实例化一个 Person 对象

father <- new("Person",name="F",age=44)

# 查看 father 对象，有两个属性 name 和 age
father
```

```
## An object of class "Person"
## Slot "name":
## [1] "F"
##
## Slot "age":
## [1] 44
```

```
# 查看 father 对象类型为 Person
class(father)
```

```
## [1] "Person"
## attr(,"package")
## [1] ".GlobalEnv"
```

```
# 查看 father 对象为 S4 的对象
otype(father)
```

```
## [1] "S4"
```

4.2.3 创建一个有继承关系的 S4 对象

```
# 创建一个 S4 对象 Person
setClass("Person",slots=list(name="character",age="numeric"))

# 创建 Person 的子类
setClass("Son",slots=list(father="Person",mother="Person"),contains = "Person")

# 实例化 Person 对象

father <- new("Person",name="F",age=44)
mother <- new("Person",name="M",age=39)
```

```
# 实例化一个 Son 对象

son <- new("Son",name="S",age=16,father=father,mother=mother)

# 查看 son 对象的 name 属性
son@name
```

```
## [1] "S"
```

```
# 查看 son 对象的 age 属性
son@age
```

```
## [1] 16
```

```
# 查看 son 对象的 father 属性
son@father
```

```
## An object of class "Person"
## Slot "name":
## [1] "F"
##
## Slot "age":
## [1] 44
```

```
# 查看 son 对象的 mother 属性
son@mother
```

```
## An object of class "Person"
## Slot "name":
## [1] "M"
##
## Slot "age":
## [1] 39
```

```
# 查看 son 类型
otype(son)
```

```
## [1] "S4"
```

```
# 查看 son@name 的属性
otype(son@name)
```

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

```
#  查看 son@mother 的属性
otype(son@mother)
```

```
## [1] "S4"
```

```
# 用 isS4() 检查 S4 对象的类型
isS4(son)
```

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

```
isS4(son@name)
```

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

```
isS4(son@mother)
```

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

4.2.4 S4 对象的默认值

```
setClass("Person",slots=list(name="character",age="numeric"))

# 属性 age 为空
a <- new("Person",name="a")
a
```

```
## An object of class "Person"
## Slot "name":
## [1] "a"
##
## Slot "age":
## numeric(0)
```

```
# 设置属性 age 的默认值为 20

setClass("Person",slots=list(name="character",age="numeric"),prototype=list(age=20))

# 初始化 b 对象
b <- new("Person",name="b")

# 属性 age 的默认值是 20
b
```

```
## An object of class "Person"
## Slot "name":
## [1] "b"
##
## Slot "age":
## [1] 20
```

4.2.5 S4 对象的类型检查

```
setClass("Person",slots=list(name="character",age="numeric"))
# 传入错误 age 类型
bad <- new("Person",name="bad",age="abc")

###
Error in validObject(.Object) : 类别为 "Person" 的对象不对: invalid object for slot "age" in c
```

```
# 设置 age 的非负检查
setValidity("Person",function(object){
  if(object@age <= 0) stop("Age is negative.")
})

# 传入小于 0 的年龄
bad2 <- new("Person",name="bad",age=-1)

###
Error in validityMethod(object) : Age is negative.
```

4.2.6 从一个已经实例化的对象中创建新对象

S4 对象, 还支持从一个已经实例化的对象中创建新对象，创建时可以覆盖旧对象的值

```
setClass("Person",slots=list(name="character",age="numeric"))

# 创建一个对象实例 n1

n1 <- new("Person",name="n1",age=19)
n1
```

```
## An object of class "Person"
## Slot "name":
## [1] "n1"
##
## Slot "age":
## [1] 19
```

```
# 从实例 n1 中，创建实例 n2，并修改 name 的属性值

n2 <- initialize(n1,name="n2")
n2
```

```
## An object of class "Person"
## Slot "name":
## [1] "n2"
##
## Slot "age":
## [1] 19
```

4.3 访问对象的属性

在 S3 对象中，一般我使用 $ 来访问一个对象的属性，但在 S4 对象中，□□□□□@□□□□□□□□□□。

```
setClass("Person",slots=list(name="character",age="numeric"))
a <- new("Person",name="a")

# 访问 S4 对象的属性

a@name
```

```
## [1] "a"
```

```
slot(a,"name")
```

```
## [1] "a"
```

```
# 错误的访问
#a$name
#a[1]
```

4.4 S4 的泛型函数

S4 的泛型函数实现有别于 S3 的实现，S4 分离了方法的定义和实现，如在其他语言中我们常说的接口和实现分离。通过 setGeneric() 来定义接口，通过 setMethod() 来定义现实类。这样可以让 S4 对象系统，更符合面向对象

的特征。

普通函数的定义和调用

```
work <- function(x) cat(x,"is working")
work("Conan")
```

```
## Conan is working
```

让我们看看如何用 R 分离接口和实现

```
# 定义 Person 对象

setClass("Person",slots=list(name="character",age="numeric"))

# 定义泛型函数 work 即接口
setGeneric("work",function(object) standardGeneric("work"))
```

```
## [1] "work"
```

```
# 定义 work 的实现，并指定参数类型为 Person 对象
setMethod("work",signature(object="Person"),function(object) cat(object@name,"is working")
```

```
## [1] "work"
```

```
# 创建一个 Person 对象 a
a <- new("Person",name="Conan",age=16)

# 把对象 a 传入 work 函数
work(a)
```

```
## Conan is working
```

通过 S4 对象系统，把原来的函数定义和调用 2 步完成的分成 4 步。

- 定义数据对象类型
- 定义接口函数
- 定义实现函数
- 把数据对象以参数传入到接口函数，执行实现函数

通过 S4 对象系统，是一个结构化的，完整的面向对象的实现。

4.5 查看 S4 对象的函数

当我们使用 S4 对象进行面向对象封装后，我们还需要能查看到 S4 对象的定义和函数定义，还是以上街中 Person 和 work 的例子

```
library(pryr)
# 检查 work 的类型

ftype(work)

# 直接查看 work 函数
work

# 查看 work 函数的显示定义
showMethod(work)
```

```
# 查看 Person 对象的 work 函数现实
getMethod("work","Person")

# 检查 Person 对象有没有 work 函数

existMethod("work","Person")
hasMethod("work","Person")
```

4.6 S4 对象的使用

下面我们用 S4 对象那个实现一个具体的例子。

4.6.1 任务 1：定义一个图形库的数据结构和计算函数

假设 Shape 为图形的基类，包括圆形 (Circle) 和椭圆形 (Ellipse)，并计算出他们的面积 (area) 和周长 (circum)

- 定义图形库的数据结构
- 定义圆形的数据结构，并计算面积和周长
- 定义椭圆形的数据结构，并计算面积和周长

定义基类 Shape 和圆形类 Circle

```
# 定义基类 Shape

setClass("Shape",slots=list(name="character"))

# 定义圆形类，并继承 shape, 属性 radius 默认为 1

setClass("Circle",contains = "Shape",slots=list(radius="numeric"),prototype=list(radius=1)

# 验正 radius 属性值要大于等于 0
setValidity("Circle",function(object){
  if(object@radius <= 0) stop("Radius is negative")
})
```

```
## Class "Circle" [in ".GlobalEnv"]
##
## Slots:
##
## Name:      radius       name
## Class:    numeric character
##
## Extends: "Shape"
```

```
# 创建两个圆形实例

c1 <- new("Circle",name="c1")
c2 <- new ("Circle",name="c2",radius=5)
```

定义计算面积的接口和实现

```
setGeneric("area",function(obj,...){
  standardGeneric("area")
```

```
})
```

```
## [1] "area"
```

```
# 计算面积的函数实现

setMethod("area","Circle",function(obj,...){
  print("Area Circle Method")
  pi*obj@radius^2
})
```

```
## [1] "area"
```

```
# 分别计算 c1 和 c2 的两个圆形的面积
area(c1)
```

```
## [1] "Area Circle Method"
## [1] 3.141593
```

```
area(c2)
```

```
## [1] "Area Circle Method"
## [1] 78.53982
```

定义计算周长的接口和实现

```
# 计算周长泛型函数接口

setGeneric("circum",function(obj,...){
  standardGeneric("circum")
})
```

```
## [1] "circum"
```

```
# 计算周长的函数实现

setMethod("circum","Circle",function(obj,...){
  2*pi*obj@radius
})
```

```
## [1] "circum"
```

```
# 分别计算 c1 和 c2 的周长

circum(c1)
```

```
## [1] 6.283185
```

```
circum(c2)
```

```
## [1] 31.41593
```

上面代码，我们实现了圆形的定义，下面我们实现椭圆形

```
# 定义椭圆形的类，继承 Shape,radius 参数默认值为 c(1,1)# 分别表示椭圆形的常半径和短半径

setClass("Ellipse",contains = "Shape",slots=list(radius="numeric"),prototype = list(radius

# 验证 radius 参数
```

```
setValidity("Ellipse",function(object){
  if(length(object@radius)!=2) stop("It's note Ellipse")
  if(length(which(object@radius<=0))>0) stop("Radius is negative")
})
```

```
## Class "Ellipse" [in ".GlobalEnv"]
##
## Slots:
##
## Name:      radius       name
## Class:   numeric character
##
## Extends: "Shape"
```

```
# 创建两个椭圆形实例 e1,e2

e1 <- new("Ellipse",name="e1")
e2 <- new("Ellipse",name="e2",radius=c(5,1))

# 计算椭圆形面积的函数的实现

setMethod("area",'Ellipse',function(obj,...){
  print("Area Ellipse Method")
  pi*prod(obj@radius)
})
```

```
## [1] "area"
```

```
# 计算 e1,e2 的面积

area(e1)
```

```
## [1] "Area Ellipse Method"
```

```
## [1] 3.141593
```

```
area(e2)
```

```
## [1] "Area Ellipse Method"
```

```
## [1] 15.70796
```

```
# 计算椭圆形周长的函数实现

setMethod("circum","Ellipse",function(obj,...){
  cat("Ellipse Circum:\n")
  2*pi*sqrt((obj@radius[1]^2+obj@radius[2]^2)/2)
})
```

```
## [1] "circum"
```

```
# 计算 e1,e2 周长

circum(e1)
```

```
## Ellipse Circum:
```

```
## [1] 6.283185
```

```
circum(e2)
```

```
## Ellipse Circum:
## [1] 22.65435
```

4.6.2 任务 2：重构圆形和椭圆形的设计

上一步，我们已经完成了圆形和椭圆形的数据结构定义，以及计算面积和周长的方法实现。不知大家有没有发现，圆形是椭圆的一个特例吗？当椭圆形的长轴和短轴相等时，形成的图形为圆形。椭圆是圆形的父类，而圆形是椭圆形的子类。

```
# 基类 Shape

setClass("Shape",slots=list(name="character",shape="character"))

# Ellipse 继承 Shape

setClass("Ellipse",contains = "Shape",slots=list(radius="numeric"),prototype = list(radius

# Circle 继承 Ellipse

setClass("Circle",contains = "Ellipse",slots=list(radius="numeric"),prototype=list(radius=

# 定义 area 接口

setGeneric("area",function(obj,...) standardGeneric("area"))
```

```
## [1] "area"
```

```
# 定义 area 的 Ellipse 实现

setMethod("area","Ellipse",function(obj,...){
  cat("Ellipse Area: \n")
  pi*prod(obj@radius)
})
```

```
## [1] "area"
```

```
# 定义 area 的 Circle 实现

setMethod("area","Circle",function(obj,...){
  cat("Circle Area:\n")
  pi*obj@radius^2
})
```

```
## [1] "area"
```

```
# 定义 circum 接口

setGeneric("circum",function(obj,...) standardGeneric("circum"))
```

```
## [1] "circum"
```

```
# 定义 circum 的 ellipse 实现

setMethod("circum","Ellipse",function(obj,...){
```

```
  cat("Ellipse circum:\n")
  2*pi*sqrt((obj@radius[1]^2+obj@radius[2]^2)/2)
})
```

```
## [1] "circum"
```

```
# 定义 circum 的 circle 实现

setMethod("circum","Circle",function(obj,...){
  cat("Ellipse circum:\n")
  2*pi*obj@radius
})
```

```
## [1] "circum"
```

```
# 创建实例

e1 <- new("Ellipse",name="e1",radius=c(2,5))
e2 <- new("Circle",name="e2",radius=2)

# 计算面积和周长

area(e1)
```

```
## Ellipse Area:
## [1] 31.41593
```

```
circum(e1)
```

```
## Ellipse circum:
## [1] 23.92566
```

```
area(e2)
```

```
## Circle Area:
## [1] 12.56637
```

```
circum(e2)
```

```
## Ellipse circum:
## [1] 12.56637
```

这样是不是显得更合理？

4.6.3 任务 3：增加矩形的图形处理

进一步扩充图形库需要加入矩形和正方形

- 定义矩形的数据结构，计算面积和周长
- 定义长方形的数据结构，计算面积和周长
- 矩形是正方形的父类，正方形是矩形的子类

```
# 定义矩形 Rectangle，继承 Shape

setClass("Rectangle",contains = "Shape",slots=list(edges="numeric"),prototype=list(edges=c

# 定义正方形 Square 继承 Rectangle
```

```
setClass("Square",contains = "Rectangle",slots=list(edges="numeric"),prototype = list(edge

# 定义 area 的 Rectangle 实现

setMethod("area","Rectangle",function(obj,...){
  cat("Rectangle Area:\n")
  prod(obj@edges)
})
```

```
## [1] "area"
```

```
# 定义 area 的 Square

setMethod("area","Square",function(obj,...){
  cat("Square Area: \n")
  obj@edges^2
})
```

```
## [1] "area"
```

```
# 定义 circum 的 Rectangle 实现

setMethod("circum","Rectangle",function(obj,...){
  cat("Rectangle Circum:\n")
  2*sum(obj@edges)
})
```

```
## [1] "circum"
```

```
# 定义 circum 的 Square 实现

setMethod("circum","Square",function(obj,...){
  cat("Square circum:\n")
  4*obj@edges
})
```

```
## [1] "circum"
```

```
# 创建实例

r1 <- new("Rectangle",name="r1",edges=c(2,5))
s1 <- new("Square",name='s1',edges=2)

# 计算矩形的面积和周长

area(r1)
```

```
## Rectangle Area:
## [1] 10
```

```
circum(r1)
```

```
## Rectangle Circum:
## [1] 14
```

```
area(s1)
```

```
## Square Area:
```

```
## [1] 4
```

```
circum(s1)
```

```
## Square circum:
## [1] 8
```

这样，图形库就支持 4 种图形了，用面向对象的结构去设计，就会非常清晰。

4.6.4 任务 4：在基类 Shape 中增加 shape 属性和 getShape 方法

对图形库的所有图形定义图形类型的变量 shape, 然后在提供一个 getShape 函数来检查实例中的 shape 变量。

```
# 重新定义基类 Shape，增加 shape 属性

setClass("Shape",slots=list(name="character",shape="character"))

# 定义 getShape 接口

setGeneric("getShape",function(obj,...){
  standardGeneric("getShape")
})
```

```
## [1] "getShape"
```

```
# 定义 getShape 实现

setMethod("getShape","Shape",function(obj,...){
  cat(obj@shape,"\n")
})
```

```
## [1] "getShape"
```

其实，这样改动一下就 OK 了，我们只需要重新实例化每个图形的对象就好了。

```
# 实例化一个 Square 对象，并给 shape 属性赋值

s1 <- new("Square",name='s1',edges=2,shape='Square')

# 调用基类的 getShape() 函数

getShape(s1)
```

```
## Square
```

如果再多做一步，可以修改每个对象的定义，增加 shape 属性的默认值。

```
setClass("Ellipse",contains = 'Shape',slots=list(radius="numeric"),prototype = list(radius
```

通过这节的例子，我们全面的了解了 R 语言的面向的使用和 S4 对象系统的面向对象的程序设计。

在程序猿的世界里，世间万物都可以抽象成对象！

Chapter 5

基于 RC 的面向对象编程

RC(Reference Class) 对象系统从底层上改变了原有 S3 和 S4 对象系统的设计，去掉了泛型函数，其真正的以类为基础实现面向对象的特征。在 R 中常备简记为 R5http://adv-r.had.co.nz/R5.html(但这不是官方的名称)，而是为了从形式上和 S3 和 S4 相匹配。

5.1 RC 对象系统的介绍

RC 是 Reference Classes 的简称，又称为 R5, 在 R 语言的 2.12 版本被引入，是最新一代的面向对象系统。

RC 不同于原来的 S3 和 S4 独享系统，RC 队形系统得的方法是在类中自定的，而不是泛型函数。RC 对象的行为更相似于其他的编程语言，实例化队形的语法也有所改变。

从面向对象的角度来说，我们下面重定义几个名词。

- 类：面向对象系统的基本类型，类是静态结构定义。
- 对象：类实例化之后，在内存中生成结构体
- 方法：是类中的函数定义，不通过泛型函数实现。

5.2 如何创建 RC 类？

RC 对象是以类为基本类型，有专门的类的定义函数 setRefClass(), 实例化则通过类的方法生成。

5.2.1 setRefClass()

```
setRefClass(Class,fields=,contains=,methods=,where=,...)
```

参数列表：

- Class: 定义类名
- fields: 定义属性和属性类型
- contains: 定义父类，继承关系
- methods: 定义类中的方法
- where: 定义存储空间

从 setRefClass() 函数的定义来看，参数比 S4 的 setClass() 函数变少了

5.2.2 创建 RC 类和实例

```
# 定义一个 RC 类

User <- setRefClass("User",fields=list(name="character"))

# 查看 User 的定义
User
```

```
## Generator for class "User":
##
## Class fields:
##
## Name:       name
## Class: character
##
## Class Methods:
##      "field", "trace", "getRefClass", "initFields", "copy", "callSuper",
##      ".objectPackage", "export", "untrace", "getClass", "show",
##      "usingMethods", ".objectParent", "import"
##
## Reference Superclasses:
##      "envRefClass"
```

```
# 实例化一个 User 对象 u1

u1 <- User$new(name='u1')

# 查看 u1 对对象

u1
```

```
## Reference class object of class "User"
## Field "name":
## [1] "u1"
```

```
# 检查 User 类的类型
library(pryr)
class(User)
```

```
## [1] "refObjectGenerator"
## attr(,"package")
## [1] "methods"
```

```
is.object(User)
```

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

```
otype(User)
```

```
## [1] "RC"
```

```
# 检查 u1 的类型

class(u1)
```

```
## [1] "User"
## attr(,"package")
## [1] ".GlobalEnv"
```

```
is.object(u1)
```

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

```
otype(u1)
```

```
## [1] "RC"
```

5.2.3 创建一个有继承关系的 RC 类

```
# 创建 RC 类 User

User <- setRefClass("User",fields=list(name="character"))

# 创建 User 的子类 Member

Member <- setRefClass("Member",contains="User",fields=list(manager="User"))

# 实例化 User
manager <- User$new(name="manager")

# 实例化一个 son 对象

member <- Member$new(name="member",manager=manager)

# 查看 Mebmber 对象

member
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "member"
## Field "manager":
## Reference class object of class "User"
## Field "name":
## [1] "manager"
```

```
# 查看 member 对象的 name 属性

member$name
```

```
## [1] "member"
```

```
# 查看 member 对象的 manager 属性

member$manager
```

```
## Reference class object of class "User"
## Field "name":
## [1] "manager"
```

```
# 查看对象的属性类型

otype(member$name)
```

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

```
otype(member$manager)
```

```
## [1] "RC"
```

5.2.4 RC 对象的默认值

RC 类有一个指定构造器方法 $initialize(), 这个构造器方法在实例化对象时，会自动被运行一次，通过这个构造方法可以设置属性的默认值。

```
# 定义一个 RC 类

User <- setRefClass("User",
                    # 定义两个属性
                    fields=list(name="character",level="numeric"),
                    methods = list(initialize=function(name,level){
                      print("User::initialize")
                      # 给属性增加默认值
                      name <<- "conan"
                      level <<- 1
                    }))

# 实例化 u1
u1 <- User$new()
```

```
## [1] "User::initialize"
```

```
# 查看对象 u1 属性被增加了默认值

u1
```

```
## Reference class object of class "User"
## Field "name":
## [1] "conan"
## Field "level":
## [1] 1
```

5.3 对象赋值

```
# 定义 User 类

User <- setRefClass("User",fields=list(name='character',age='numeric',gender='factor'))

# 定义一个 factor 类型
genderFactor <- factor(c('F','M'))

# 实例化 u1

u1 <- User$new(name="u1",age=44,gender=genderFactor[1])

# 查看 age 属性值
u1$age
```

```
## [1] 44
```

给 u1 的 age 属性赋值

```
# 重新赋值

u1$age <- 10

# age 属性改变
u1$age
```

```
## [1] 10
```

把 u1 对象赋给 u2 对象

```
# 把 u1 赋值给 u2 对象
u2 <- u1

# 查看 u2 的 age 属性
u2$age
```

```
## [1] 10
```

```
# 重新赋值
u1$age <- 20

# 查看 u1,u2 的 age 属性都发生了改变

u1$age
```

```
## [1] 20
```

```
u2$age
```

```
## [1] 20
```

这是由于把 u1 赋值给 u2，传递的是 u1 的实例化对象的引用，而不是值本身，这一点与其他语言中对象赋值时一样的。

如果想进行赋值而不是引入传递，可以进行下面的操作

```
# 调用 u1 的内置方法 copy(), 赋值给 u3

u3 <- u1$copy()

# 查看 u3 的 age 属性
u3$age
```

```
## [1] 20
```

```
# 重新赋值,u3 的 age 属性值并未变化

u1$age <- 30
```

对引入关系把我，可以减少值传递过程中的内存复制过程，可以让我们的程序运行效率更高。

5.4 定义对象的方法

在 S3,S4 的对象系统中，我们实现对象行为时，都是借助于泛型函数实现的。这种实现方法的最大问题是：在定义函数和对象的代码是分离的，需要在运行时，通过判断对象的类型完成方法的调用。而 RC 对象系统中，方法可以定义在类的内部，通过实例化的对象完成方法的调用。

```
# 定义一个 RC 类包括方法

User <- setRefClass("User",fields=list(name="character",favorite='vector'),

                    # 方法属性
                    methods= list(

                      # 增加一个兴趣
                      addFavorite = function(x){
                        favorite <<- c(favorite,x)
                      },
                      # 删除一个兴趣
                      delFavorite = function(x){
                        favorite <<- favorite[-which(favorite==x)]
                      },
                      # 重新定义兴趣列表
                      setFavorite = function(x){
                        favorite <<- x
                      }
                    ))

# 实例化对象 u1

u1 <- User$new(name="u1",favorite=c('movie','football'))

# 查看 u1 对象

u1
```

```
## Reference class object of class "User"
## Field "name":
## [1] "u1"
## Field "favorite":
## [1] "movie"    "football"
```

接下来进行方法操作

```
# 删除一个兴趣
u1$delFavorite("football")
# 查看兴趣属性
u1$favorite
```

```
## [1] "movie"
```

直接到方法定义到类的内部，通过实例化的对象进行访问。这样就做到了，在定义时就能保证了方法的作用域，减少运行时检查的系统开销。

5.5 RC 对象内置方法和内置属性

对于 RC 的实例化对象，除了我们自己定义的方法函数，还有几个内置的方法。之前属性赋值赋值时使用的 copy() 方法，就是其中之一

5.5.1 内置方法：

- initialize 类的初始化函数，用于设置属性的默认值，只有在类定义的方法中使用。
- callSuper 调用父类的同名方法，只能在类定义的方法中使用
- copy 复制实例化对象的所有属性
- initFields 给对象的属性赋值
- field 查看属性或给属性赋值
- getClass 查看对象的类定义
- getRefClass() 同 getClass()
- show 查看当前对象
- export 查看属性值以类为作用域
- import 把一个对象的属性值赋值给另一个对象
- trace 跟踪对象中方法调用，用于程序 debug
- untrace 取消跟踪
- usingMethods 用于实现方法调用，只能在类定义的方法中使用，这个方法不利于程序的健壮性，所以不建议使用。

接下来我们使用这些内置方法。

```
# 类 User

User <- setRefClass("User",
                    fields=list(name="character",level="numeric"),
                    methods = list(
                      initialize=function(name,level){
                        print("User::initialize")
                        name <<- "conan"
                        level <<- 1
                      },
                      addLevel = function(x){
                        print("User::addlevel")
                        level<<-level+x
                      },
                      addHighLevel = function(){
                        print("user::addHighLevel")
                        addLevel(2)
                      }
                    ))
```

定义子类 Member 继承父类 User, 并包括同名方法 addLevel 覆盖父类的方法，在 addLevel 方法中，会调用父类的同名方法。

```
# 子类 Member

Member <- setRefClass("Member",contains="User",
                      # 子类中的属性
                      fields = list(age='numeric'),
                      methods=list(

                        # 覆盖父类的同名方法
                        addLevel = function(x){
```

```
                            print("Member::addLevel")
                                                    callSuper(x)
                          level <<- level+1

                          }

                        )
                        )
```

分别实例化对象 u1,m1

```
# 实例化 u1

u1 <- User$new(name='u1',level=10)
```

```
## [1] "User::initialize"
```

```
# 查看 u1 对象，$new() 不能实现赋值操作

u1
```

```
## Reference class object of class "User"
## Field "name":
## [1] "conan"
## Field "level":
## [1] 1
```

```
# 通过 $initFields() 向属性赋值

u1$initFields(name='u1',level=10)
```

```
## Reference class object of class "User"
## Field "name":
## [1] "u1"
## Field "level":
## [1] 10
```

```
# 实例化 m1

m1 <- Member$new()
```

```
## [1] "User::initialize"
```

```
m1$initFields(name='m1',level=100,age=12)
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 100
## Field "age":
## [1] 12
```

执行 $copy() 方法，赋值对象属性并传值。

```
# 属性赋值到 u2
u2 <- u1$copy()
```

```
## [1] "User::initialize"
```

使用方法 field()，查看并给 level 属性赋值

```
# 查看 level 属性值

u1$field('level')
```

```
## [1] 10
```

```
# 给 level 属性值为 1

u1$field('level',1)

# 查看 level 属性值

u1$level
```

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

使用 getRefClass() 和 getClass() 方法查看 u1 对象的类定义。

```
# 类引入的定义

m1$getRefClass()
```

```
## Generator for class "Member":
##
## Class fields:
##
## Name:       name     level       age
## Class: character   numeric   numeric
##
## Class Methods:
##      "addLevel#User", "import", ".objectParent", "usingMethods", "show",
##      "getClass", "untrace", "export", ".objectPackage", "callSuper",
##      "copy", "initFields", "getRefClass", "trace", "field", "initialize",
##      "addLevel", "addHighLevel"
##
## Reference Superclasses:
##      "User", "envRefClass"
```

```
# 类定义

m1$getClass()
```

```
## Reference Class "Member":
##
## Class fields:
##
## Name:       name     level       age
## Class: character   numeric   numeric
##
## Class Methods:
##      "addLevel#User", "import", ".objectParent", "usingMethods", "show",
##      "getClass", "untrace", "export", ".objectPackage", "callSuper",
##      "copy", "initFields", "getRefClass", "trace", "field", "initialize",
##      "addLevel", "addHighLevel"
##
## Reference Superclasses:
```

```
##       "User", "envRefClass"
```

```
# 通过 otype 查看类型的不同

otype(m1$getRefClass())
```

```
## [1] "RC"
```

```
otype(m1$getClass())
```

```
## [1] "S4"
```

使用 $show() 方法查看属性值，$show()，同 show() 函数，对象直接输出时就是调用了 $show() 方法

```
m1$show()
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 100
## Field "age":
## [1] 12
```

```
show(m1)
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 100
## Field "age":
## [1] 12
```

```
m1
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 100
## Field "age":
## [1] 12
```

使用 $trace() 跟踪方法调用, 再用 $untrace() 方法取消跟踪绑定

```
# 对我 addLevel() 方法跟踪

m1$trace('addLevel')
```

```
## Tracing reference method "addLevel" for object from class
## "Member"
```

```
## [1] "addLevel"
```

```
# 调用 addlevel() 方法,tracing m1$addLevel(1) 被打印跟踪生效

m1$addLevel(1)
```

```
## Tracing m1$addLevel(1) on entry
## [1] "Member::addLevel"
## [1] "User::addlevel"
```

```
# 取消对 addLevel() 方法跟踪

m1$untrace("addLevel")
```

```
## Untracing reference method "addLevel" for object from class
## "Member"
```

```
## [1] "addLevel"
```

使用 $export() 方法, 以类作为作用域查看属性

```
# 查看在 member 类中的属性
m1$export("Member")
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 102
## Field "age":
## [1] 12
```

```
# 查看在 User 类中的属性，当前作用域不包括 age 属性

m1$export("User")
```

```
## [1] "User::initialize"
```

```
## Reference class object of class "User"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 102
```

使用 $import() 方法, 把一个对象的属性值赋值给另一个对象

```
# 实例化 m2

m2 <- Member$new()
```

```
## [1] "User::initialize"
```

```
m2
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "conan"
## Field "level":
## [1] 1
## Field "age":
## numeric(0)
```

```
# 把 m1 对象的值赋值给 m2 对象

m2$import(m1)
```

5.5.2 内置属性

RC 对象实例化后，有两个内置属性

- .self 实例化对象自身
- .refClassDef 类的定义类型

```
# $.self 属性
m1$.self
```

```
## Reference class object of class "Member"
## Field "name":
## [1] "m1"
## Field "level":
## [1] 102
## Field "age":
## [1] 12
```

```
# m1$.self 和 m1 完全相同
identical(m1$.self,m1)
```

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

```
# 查看类型
otype(m1$.self)
```

```
## [1] "RC"
```

```
# $.refClassDef 属性
m1$.refClassDef
```

```
## Reference Class "Member":
##
## Class fields:
##
## Name:       name     level       age
## Class: character   numeric   numeric
##
## Class Methods:
##      "addLevel#User", "import", ".objectParent", "usingMethods", "show",
##      "getClass", "untrace", "export", ".objectPackage", "callSuper",
##      "copy", "initFields", "getRefClass", "trace", "field", "initialize",
##      "addLevel", "addHighLevel"
##
## Reference Superclasses:
##      "User", "envRefClass"
```

```
# 与 getClass() 相同

identical(m1$.refClassDef,m1$getClass())
```

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

```
# 查看类型

otype(m1$.refClassDef)
```

```
## [1] "S4"
```

5.6 RC 类的辅助函数

当定义好了 RC 类的结构，有一些辅助函数可以帮助我们查看类型的属性和方法，上面用于创建实例化的对象的 $new() 函数，也属于这类辅助函数

- new 用于实例化对象
- help 用于查询类中定义的所有方法
- methods 列出类中定义的所有方法
- fields 列出类中定义的所有属性
- lock 给属性加锁，实例化的对象的属性只允许赋值依次，即赋值变量不可修改
- trace 跟踪方法
- accessors 给属性生成 get/set 方法

接下来，使用辅助函数，继续定义之前的 USER 类的结构

```
# 定义 User 类

User <- setRefClass("User",
                    fields=list(name="character",level='numeric'),
                    methods=list(
                      initialize=function(name,level){
                        print("User::initialize")
                        name <<- 'conan'

level <<- 1                    },

addLevel = function(x){
  print("User::addLevel")
  level <<- level+x
},
addHighLevel = function(){
  print("User::addhighLevel")
  addLevel(2)
}
                    ))

# 实例化对象 u1

u1 <- User$new()
```

```
## [1] "User::initialize"
```

```
# 列出 User 类中的属性

User$fields()
```

```
##        name       level
## "character"   "numeric"
```

```
# 列出 User 类中的方法
User$methods()
```

```
##  [1] ".objectPackage" ".objectParent"  "addHighLevel"   "addLevel"
##  [5] "callSuper"      "copy"            "export"         "field"
##  [9] "getClass"       "getRefClass"     "import"         "initFields"
```

```
## [13] "initialize"     "show"           "trace"          "untrace"
## [17] "usingMethods"
```

```
# 查看 User 类中的方法调用

User$help("addLevel")
```

```
## Call:
## $addLevel(x)
```

```
User$help("show")
```

```
## Call:
## $show()
```

给 User 类中的属性, 增加 get/set 方法

```
# 给 level 属性增加 get/set 方法

User$accessors("level")

# 列出所有方法

User$methods()
```

```
##  [1] ".objectPackage" ".objectParent"  "addHighLevel"   "addLevel"
##  [5] "callSuper"      "copy"           "export"         "field"
##  [9] "getClass"       "getLevel"       "getRefClass"    "import"
## [13] "initFields"     "initialize"     "setLevel"       "show"
## [17] "trace"          "untrace"        "usingMethods"
```

使用 $trace() 函数, 跟踪 addLevel 方法

使用 $lock() 函数，把 level 属性设置为常量

```
# 锁定 level 属性

User$lock("level")

# 查看 User 类中被锁定的属性
User$lock()
```

```
## [1] "level"
```

```
# 实例化 u3，level 就被初始化依次
u3 <- User$new()
```

```
## [1] "User::initialize"
```

```
# 给 level 属性再次赋值出错

#u3$level = 1
```

5.7 RC 对象实例

我们用 RC 面向对象的系统做一个例子，定义一套动物叫声模型

5.7.1 任务 1：定义动物的数据结构和发声方法

定义 animal 为动物的基类，包括：猫，狗，鸭

```
# 创建 Animal 类，包括 name 属性，构造方法 initialize()，叫声方法 bark()

Animal <- setRefClass("Animal",
                      fields=list(name="character"),
                      methods=list(
                        initialize = function(name){
                          name <<- "Animal"
                        },
                        bark = function()print("Animal::bark")
                      ))

# 创建 Cat 类，继承 Animal 类，并重写（overwrite
#）了 initialize() 和 bark()

Cat <- setRefClass("Cat",contains="Animal",
                   methods=list(
                     initialize = function(name) name <<- 'cat',
                     bark = function() print(paste(name,"is miao miao"))
                   ))

# 创建 Dog 类


Dog <- setRefClass("Dog",contains="Animal",
                   methods=list(
                     initialize = function(name) name <<- 'Dog',
                     bark = function() print(paste(name,"is wang wang"))
                   ))

# 创建 Duck 类
Duck<- setRefClass("Duck",contains="Animal",
                   methods=list(
                     initialize = function(name) name <<- 'Duck',
                     bark = function() print(paste(name,"is ga ga"))
                   ))
```

接下来，我们实例化对象，然后研究他们的叫声

```
# 创建 cat 实例

cat <- Cat$new()
cat$name
```

```
## [1] "cat"
```

```
# cat 叫声
cat$bark()
```

```
## [1] "cat is miao miao"
```

5.7.2 任务 2：定义动物的体貌特征

动物的体貌特征包括头，身体，肢，翅膀, 我们只增加肢，只是在 Animal 中添加 limbs 属性

5.7.3 任务 3：定义动物的行动方式

Animal 中定义 action 方法，每一个子类使用 callSuper() 重写该方法。

通过这个例子，我们应该可以全面的理解 R 语言中基于 RC 对象系统的面向对象的程序设计，我本人推荐大家使用 RC 面向对象，因为这更像是传统语言的面向对象方式。

Chapter 6

基于 R6 的面向对象

R6 是什么？听说过 S3,S4,RC(R5),R6 难道是一种新的类型吗？其实 R6 是 R 语言的一个面向对象的 R 包，R6 类型非常接近 RC 类型，但是比 RC 类型更轻，由于 R6 不依赖于 S4 的对象系统，所以用 R6 的构建面向对象系统更加有效率。

6.1 初识 R6

R6 是一个单独的 R 包，与我们熟悉的原生的面向对象系统 S3,S4,RC 类型不一样。在 R 语言的面向对象系统中，R6 类型与 RC 类型是比较相似的，但 R6 并不急于 S4 的对象系统，因此我们在使用 R6 类型开发 R 包的时候，不依赖于 methods 包，而用 RC 类开发 R 包时必须设置 methods 包的依赖，可以参见：发布 gridgame 游戏包

RC 类型比 RC 类型更符合其他编程对于现象对象的设置，支持类的公有成员和私有成员，支持函数的主动绑定，并支持跨包的继承关系，由于 RC 类型的面向对象系统设计并不彻底，所以才会有 R6 这样的包出现。

6.2 创建 R6 类和实例化对象

```
# install.packages("R6")
library(R6)
library(pryr)
```

6.2.1 如何创建 R6 类?

R6 对象系统是以类为基本类型，有专门的类的定义函数 R6Class() 和实例化对象的生成方法，下面我们用 R6 对象创建一个类

先查看 R6 的类创建函数 R6Class() 函数定义

```
R6Class
function(classname=NULL,public=list(),private=NULL,activate=NULL,inherit=NULL,lock=TRUE,c
```

参数列表：

- classname 定义类名
- public 定义共有成员，包括公有方法和属性
- private 定义私有成员，包括私有方法和属性
- active 主动绑定的函数列表

- inherit 定义父类，继承关系
- lock 是否上锁，如果上锁则用于变量存储的环境空间被锁定，不能修改
- class 是否把属性封装成对象，默认是封装，如果选择不封装，类中属性存在一个环境空间中
- portable 是否可移植类型，默认是可移植型类，类中成员访问需要调用 self 和 private 对象
- parent_env 定义对象的父环境空间

从 R6Class() 函数的定义来看，参数比 RC 类定义的 setRefClass() 函数有更多的对象特征。

6.2.2 创建 R6 的类和实例化对象

首先创建一个最简单的 R6 的类，只包括一个公有方法。

```
Person <- R6Class("Person",# 定义一个 R6 类
                  public=list(
                    hello = function(){ # 定义公有方法 hello
                      print(paste("hello"))
                    }
                  ))

Person # 查看 Person 的定义
```

```
## <Person> object generator
##   Public:
##     hello: function ()
##     clone: function (deep = FALSE)
##   Parent env: <environment: R_GlobalEnv>
##   Locked objects: TRUE
##   Locked class: FALSE
##   Portable: TRUE
```

```
class(Person) # 检查 Person 的类型
```

```
## [1] "R6ClassGenerator"
```

接下来，实例化 Person 对象，使用 $new() 函数完成。

```
u1 <- Person$new() # 实例化一个 Person 对象 u1

u1
```

```
## <Person>
##   Public:
##     clone: function (deep = FALSE)
##     hello: function ()
```

```
class(u1)
```

```
## [1] "Person" "R6"
```

通过 pryr 包的 otype 检查 Person 类的类型和 u1 对象的实例化类型

```
otype(Person) # 查看 Person 类型
```

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

```
otype(u1) # 查看 u1 类型
```

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

完全没有想到，Person 和 u1 都是 S3 类型的，如果 R6 是基于 S3 系统构建的，那么其实就可以解释 R6 类型与 RC 类型的不同，并且 R6 在传值和继承上会更有效率。

6.2.3 公有成员和私有成员

类的成员，包括属性和方法 2 部分。R6 类定义中，可以分为设置公有成员和私有成员。我们设置类的共有成员，修改 Person 类的定义，在 public 参数中增加公有属性 name, 并通过 help() 方法打印 name 的属性值，让这个 R6 的类更像是 Java 语言. 在类中访问公有成员时，需要使用 self 对象进行调用。

```
Person <- R6Class("Person",
                  public=list(
                    name=NA, # 公有属性
                    initialize = function(name){ # 构造函数
                      self$name <- name
                    },
                    hello = function(){ #public 方法
                      print(paste("hello",self$name))
                    }

                  ))

connan <- Person$new("Connan") # 实例化对象
connan$hello()  # 调用 hello 方法
```

```
## [1] "hello Connan"
```

接下来设置私有成员，给 person 类中增加 private 参数，并在公有函数有调用私有成员变量，调用私有成员变量时，通过 private 对象进行访问。

```
Person <- R6Class("Person",
                  public = list(
                    name=NA,
                    initialize = function(name,gender){
                      self$name <- name
                      private$gender <- gender # 给私有属性赋值

                    },
                    hello = function(){
                      print(paste("hello",self$name))
                      private$myGender() # 调用私有方法
                    }
                  ),
                  private = list( # 私有成员
                    gender = NA,
                    myGender = function(){
                      print(paste(self$name,"is",private$gender))
                    }

                  ))

conan <- Person$new("Connan","Male") # 实例化对象

connan$hello() # 调用 hello() 方法
```

```
## [1] "hello Connan"
```

在给 Person 类中增加私有成员时, 通过 private 参数定义 gender 的私有属性和 mygender() 的私有方法。值得注意的是在类的内部，需要访问私有成员时，需要使用 private 对象进行调用。

那我直接访问公有属性和私有属性时，公有属性返回正确，而私有属性就是 NULL 值，并且访问私有方法不可见

```
connan$name   # 公有属性
```

```
## [1] "Connan"
```

```
connan$gender # 私有属性
```

```
## NULL
```

```
# connan$myGender() # 私有方法
```

进一步的，我们看看 self 对象和 private 对象, 具体是什么。在 Person 类中，增加公有方法 member(), 在 member 方法中分别打印 self 和 private 对象

```
Person <- R6Class("Person",
                  public = list(
                    name = NA,
                    initialize = function(name,gender){
                      self$name <- name
                      private$gender <- gender
                    },
                    hello = function(){
                      print(paste("Hello",self$name))
                      private$myGender()
                    },
                    member = function(){
                      print(self)
                      print(private)
                      print(ls(envir = private))
                    }
                  ),
                  private = list(
                    gender = NA,
                    myGender = function(){
                      print(paste(self$name,"is",private$gender))
                    }
                  ))

conan <- Person$new("Conan","Male")

conan$member()
```

```
## <Person>
##   Public:
##     clone: function (deep = FALSE)
##     hello: function ()
##     initialize: function (name, gender)
##     member: function ()
##     name: Conan
##   Private:
##     gender: Male
##     myGender: function ()
## <environment: 0x000000001bbbf410>
## [1] "gender"   "myGender"
```

从测试结果看，我们可以看出 self 对象，就像实例化的对象本身。private 对象则是一个环境空间，是 self 对象所在环境空间中的一个子空间，所以私有成员只能在当前类中被调用，外部访问私有成员时，就会找不到。在环境中保存私有成员的属性和方法，通过环境控件的访问控制让外部调用无法使用私有属性和方法，这种方式经常被用在 R 包开发上的技巧。关于 R 的环境请详细阅读本书第七章。

6.3 R6 类的主动绑定

主动绑定 (Active binding) 是 R6 中一种特殊的函数调用方式，把对函数的访问表现为对属性的访问，主动绑定属于公有成员。在类的定义中，通过设置 activate 参数实现主动绑定的功能，给 Person 类增加两个主动绑定的函数 activate 和 rand

```
Person <- R6Class("Person",
                  public = list(
                    num=100
                  ),
                  active = list( # 主动绑定
                    active= function(value){
                      if(missing(value))
                        return (self$num+10)
else self$num <- value/2                    },

rand = function() rnorm(1)


                  )
                  )

conan <- Person$new()

conan$num # 查看公有属性
```

```
## [1] 100
```

```
conan$active # 调用主动绑定的 active() 函数, 结果为 num +10 = 100+10
```

```
## [1] 110
```

```
# 给主动绑定额 active 函数传参书，用赋值符号"<-", 而不是方法调用"()"
conan$active <- 20

conan$num
```

```
## [1] 10
```

```
conan$active
```

```
## [1] 20
```

通过主动绑定，可以把函数的行为转换成属性的行为，让类中额函数操作更加灵活。

6.4 R6 类的继承关系

继承是函数面向对象的基本特征，R6 的面向对象系统也是支持继承的。当创建一个类时，可以继承另一个类作为父类存在。

先创建一个父类 Person, 包括共有尘缘和私有成员

```
Person <- R6Class("Person",
                  public = list(
                    name=NA,
                    initialize = function(name,gender){
                      self$name <- name
                      private$gender <- gender

                    },
                    hello = function(){
                      print(paste("hello",self$name))
                      private$myGender()
                    }
                  ),
                  private=list(
                    gender = NA,
                    myGender = function(){
                      print(paste(self$name,"is",private$gender))
                    }
                  ))
```

创建子类 Worker 继承父类 Person，并在子类增加 bye() 公有方法

```
Worker <- R6Class("Worker",
                   inherit = Person, # 继承，指向父类
                   public = list(
                     bye = function(){
                       print(paste("bye",self$name))
                     }
                   )
                   )
```

实例化父类和子类，看看继承关系是不是生效

```
u1 <- Person$new("Conan","Male") # 实例化父类

u1$hello()
```

```
## [1] "hello Conan"
## [1] "Conan is Male"
```

```
u2 <- Worker$new("Conan","Male") # 实例化子类
u2$hello()
```

```
## [1] "hello Conan"
## [1] "Conan is Male"
```

```
u2$bye()
```

```
## [1] "bye Conan"
```

我们看到继承确实生效了，在子类中我们并没有定义 hello() 方法，子类实例 u2 可以直接使用 hello() 方法。同时，子类 u2 的 bye() 方法，用到了再付类中定义的 name 属性，输出的结果完全正确。

接下来我们在子类中定义父类的同名方法，然后再查看方法的调用，看看是否会出现继承中函数重写的特征。修改 Worker 类, 在子类中定义 private 的属性和方法。

```
Worker <- R6Class("Worker",
                  inherit = Person,
                  public = list(
```

```
    bye = function(){
      print(paste("bye",self$name))
    }
  ),
  private = list(
    gender = NA,
    myGender = function(){
      print(paste("worker",self$name,"is",private$gender))
    }
  ))
```

实例化子类，调用 hello 方法

```
u2 <- Worker$new("Conan","Male")
u2$hello() # 调用 hello() 方法
```

```
## [1] "hello Conan"
## [1] "worker Conan is Male"
```

由于子类中的 myGender() 私有方法，覆盖了父类同名私有方法，所以在调用的时候，hello() 会调用子类中的 myGender() 方法实现，而忽略父类中的方法。

如果在子类中像调用父类的方法，有一个办法是使用 super 对象，通过 super$xx() 的语法进行调用。

```
Worker <- R6Class("Worker",
  inherit = Person,
  public = list(
    bye = function(){
      print(paste("bye",self$name))
    }
  ),
  private = list(
    gender = NA,
    myGender = function(){
      super$myGender() # 调用父类的方法
      print(paste("worker",self$name,"is",private$gender))
    }
  ))
```

```
u2 <- Worker$new("Conan","Male")
u2$hello()
```

```
## [1] "hello Conan"
## [1] "Conan is Male"
## [1] "worker Conan is Male"
```

在子类 myGender() 方法中，用 super 对象调用父类的 myGender() 方法，从输出可以看出，父类的同名方法也同时被调用了。

6.5 R6 类的对象的静态属性

用面向对象的方法进行编程，那么所有变量其实都是对象，我们可以把一个实例化的对象定义成另一个类的属性，这样就形成了对象的引用关系链。

需要注意的是，当属性赋值给另一个 R6 的对象时，属性的值保存了对象的引用，而非对象实例本身。利用这个规则就可以实现对象的静态属性，也就是可以在多种不同的实例中是共享对象属性，类似于 Java 中的 static 属性一样。

下面用代码描述一下，就能很容易的理解。定义两个类 A 和 B,A 类中有一个公有属性 x，B 类中有一个公有属性 a,a 为 A 类的实例化对象

```
A <- R6Class("A",
             public=list(
               x = NULL
             ))

B <- R6Class("B",
             public=list(
               a = A$new()
             ))
```

运行程序，实现 B 实例化对象 A 实例化对象的调用，并给 x 变量赋值。

```
b <- B$new() # 实例化 B 对象

b$a$x <- 1 # 给 x 变量赋值
b$a$x
```

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

```
b2 <- B$new()
b2$a$x <-2
b2$a$x
```

```
## [1] 2
```

```
b$a$x
```

```
## [1] 2
```

从输出结果上来看，a 对象实现了在多个 b 实例的共享，当 b2 实例修改 a 对象 x 值得时候，b 实例的 a 对象的 x 值也发生了变化。

这里有一种写法，我们是应该避免的，就是通过 initialize() 方法赋值

```
C <- R6Class("C",
             public = list(
               a = NULL,
               initialize = function(){
                 a <<- A$new()
               }
             ))

cc <- C$new()

cc$a$x <- 1

cc$a$x
```

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

```
cc2 <-C$new()
cc2$a$x <- 2
cc2$a$x
```

```
## [1] 2
```

```
cc$a$x # x 值未发生改变
```

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

通过 initialize() 构建 a 对象，是对单独的环境空间中的引用，所以不能实现引用对象的共享。

6.6 R6 类的可移植类型

在 R6 类的定义中，portable 参数可以设置 R6 类的类型为可移植类型和不可移植类型。可移植类型和不可移植类型主要有两个明显的特征。

- 可移植类型支持跨 R 包的继承；不可移植类型，在跨 R 包的继承的时候，兼容性不太好
- 可移植类型必须用 self 和 private 对象来访问类中的成员，如 self$x.private$y。不可移植类型，可以直接使用变量 x,y，并通过 “<<-”(超赋值) 实现赋值。

本文使用的是 R6 的 2.2.2 版本，所以默认创建的是可移植类型。所以，当我们要考虑是否有跨包继承的需要时，可以再可移植类型和不可移植类型之间进行选择。

我们比较一下 RC 类型，R6 的可移植类型和 R6 的不可移植类型三者的区别，定义一个简单的类，包括一个属性 x 和两个方法 getx()，setx()

```
RC <- setRefClass("RC",
                  fields = list(x="numeric"),
                  methods = list(
                    getx = function() x,
                    setx = function(value) x <<- value
                  ))

rc <- RC$new()
rc$setx(10)
rc$getx()
```

```
## [1] 10
```

创建一个行为完全一样的不可移植类型的 R6 类

```
NR6 <- R6Class("NR6",# R6 不可移植类型
               portable= FALSE,
               public = list(
                 x = NA,
                 getx = function() x,
                 setx = function(value) x <<- value
               ))

np6 <- NR6$new()
np6$setx(10)
np6$getx()
```

```
## [1] 10
```

再创建一个行为完全一样的可移植类型的 R6 类

```
PR6 <- R6Class("PR6",
               portable = TRUE,
               public= list(
                 x = NA,
                 getx = function() self$x,
                 setx = function(value) self$x <- value
               ))
```

```
pr6 <- PR6$new()

pr6$setx(10)

pr6$getx()
```

```
## [1] 10
```

从这个例子中，可移植类型的 R6 类和不可移植类型的区别在于 self 对象的使用。

6.7 R6 类的动态绑定

对于静态类型的编程语言来说，一旦类定义后，就不能修改类中的属性和方法。对于动态类型的编程语言来说，通常不存在这样的限制，可以任意修改其类的结构或者已经实例化的对象结构。R 作为动态语言来说，同样支持动态变量修改的，基于 S3,S4 可以通过泛型函数动态的增加函数定义，但 RC 类型是不支持的，再次感觉到 R 语言的面向对象系统设计的奇葩了。

R6 包已经考虑这种情况，提供了一种动态设置成员变量的方法用 $get() 函数。

```
A <- R6Class("A",
             public = list(
               x = 1,
               getx = function() x
             ))

A$set("public","getx2",function() self$x*2) # 动态增加 getx2() 方法

s <- A$new()
s$getx2()
```

```
## [1] 2
```

同样的，属性也可以动态的修改，动态改变 x 属性的值

```
A$set("public","x",10,overwrite=TRUE) # 动态改变 x 属性

s <- A$new()
s$x
```

```
## [1] 10
```

```
s$getx2()
```

```
## [1] 20
```

6.8 R6 类的打印函数

R6 提供了用于打印的默认方法 print()，每当要打印实例化对象时，都会调用这个默认的 print() 方法，有点类似于 Java 类中默认的 toString() 方法

我们可以覆盖 print() 方法，使用自定义的打印提示

```
A <- R6Class("A",
             public = list(
               x = 1,
               getx = function() self$x
```

```
              ))

a <- A$new()
print(a) # 使用默认的打印方法
```

```
## <A>
##   Public:
##     clone: function (deep = FALSE)
##     getx: function ()
##     x: 1
```

自定义打印方法，覆盖 print() 方法

```
A <- R6Class("A",
             public = list(
               x = 1,
               getx = function() self$x,
               print = function(...){
                 cat("Class <A> of public",ls(self),":",sep="")
                 cat(ls(self),sep=",")
                 invisible(self)
               }
             ))

a <- A$new()
print(a)
```

```
## Class <A> of publicclonegetxprintx:clone,getx,print,x
```

通过自定义方法，就可以覆盖系统默认的方法，从而输出我们想显示的文字。

6.9 实例化对象的存储

R6 是基于 S3 面向对象系统的构建，而 S3 类型又是一种比较松散的类型，会造成用户环境空间的变量泛滥的问题。R6 提供了一种方式，设置 R6Class() 的 class 参数，把类中定义的属性和方法统一存储到一个 S3 对象中，这种方式是默认的。另一种方式为，把类中定义的属性和方法统一存储到一个单独的环境空间中。

class=TRUE, 实例化 a 对象，就是一个 S3 类

```
A <- R6Class("A",
             class=TRUE,
             public=list(
               x = 1,
               getx = function() self$x
             ))

a <- A$new()
class(a)
```

```
## [1] "A"  "R6"
```

class=FALSE，实例化 a 对象，是一个环境空间，在环境空间中存储了类的变量数据

```
B <- R6Class("B",
             class=TRUE,
             public=list(
               x = 1,
```

```
              getx = function() self$x
            ))

b <- B$new()
class(b)
```

```
## [1] "B"  "R6"
```

```
b
```

```
## <B>
##   Public:
##     clone: function (deep = FALSE)
##     getx: function ()
##     x: 1
```

```
ls(envir = b)
```

```
## [1] "clone" "getx"  "x"
```

实例化对象的存储还有另外一个方面的考虑，由于类中的变量都存在于一个环境空间中，我们也可以通过手动的方式找到这个环境空间，从而进行变量的增加和修改。如果对于环境空间的变量进行修改，我们的程序将会变得非常不安全，所以为了预防安全上的问题，乐意通过 R6Class() 的 lock 参数锁定环境空间，不允许动态修改，默认值为锁定不能修改。

```
A <- R6Class("A",
             lock = TRUE,# 锁定环境空间
             public= list(
             x = 1
             ))
```

```
## R6Class A: 'lock' argument has been renamed to 'lock_objects' as of version 2.1.This code w
```

```
s <- A$new()
ls(s)
```

```
## [1] "clone" "x"
```

```
# s$aa <- 11 # 增加新变量 Error
# rm("x",envir=s) # Error
```

如果不锁定环境空间，让 lock = FALSE，则环境完全处于开放状态，可以对变量任意修改。

通过上面对 R6 的介绍，我们基本掌握了 R6 面向对象系统的只是，我们最后介绍一个基于 R6 的例子

6.10 R6 面向对象案例

用 R6 面向对象系统，构建一个图书分类的使用案例

任务 1：定义图书的静态结构

```
Book <- R6Class("Book",
                private = list(
                  title = NA,
                  price= NA,
                  category = NA
                ),
                public = list(
                  initialize = function(title,price,category){
```

```
                    private$title <- title
                    private$price <- price
                    private$category <- category
                  },
                  getPrice = function(){
                    private$price
                  }
                ))

R <- R6Class("R",inherit=Book)
Java <- R6Class("Java",inherit=Book)
Php <- R6Class("Php",inherit=Book)


r1 <- R$new("R 的极客思想",59,"R")
r1$getPrice()
```

```
## [1] 59
```

```
j1 <- Java$new("Java 编程思想",108,"Java")
j1$getPrice()
```

```
## [1] 108
```

```
p1 <- Php$new("head First PHP & MySQL",98,"PHP")
p1$getPrice()
```

```
## [1] 98
```

任务 2：双 11 图书打折

- 所有图书 9 折
- Java 图书 7 折，不支持重复打折
- R 打 7 折，支持重复打折
- PHP 图书无特别优惠

这个我们可以自己去实现了，本书就不再赘述了。

通过这个例子，我们用 R6 实现了面向对象编程的封装，继承和多态的 3 个特性，证明 R6 是一个完全的面向对象的实现，由于 R6 底层基于 S3 实现，所以比 RC 的类更加有效果，因此除了推荐大家使用 RC 外，也极力推荐大家使用 R6.

截止到现在，我们介绍了 4 种 R 语言的面向对象体系结构，选择自己理解的，总有一种适合你。

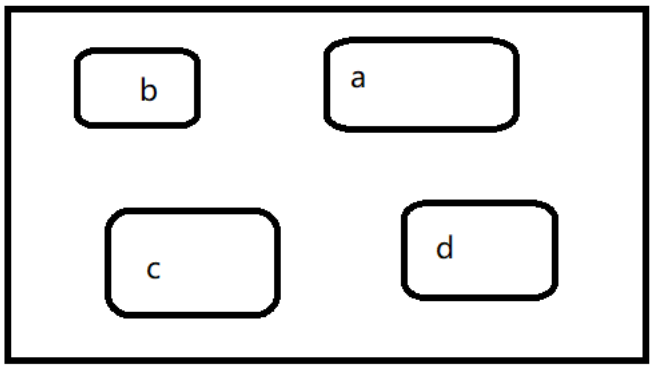
Chapter 7

环境

环境就是作用域发挥作用的数据结构，本章将深入学习环境的概念，由于环境具有引用语义，所以本身他们也是一种很有用的数据结构，当在一个环境中对其绑定的元素进行修改时，环境不会被复制，修改会在原地进行。虽然不会经常使用引用语义，但他还是非常有用的。

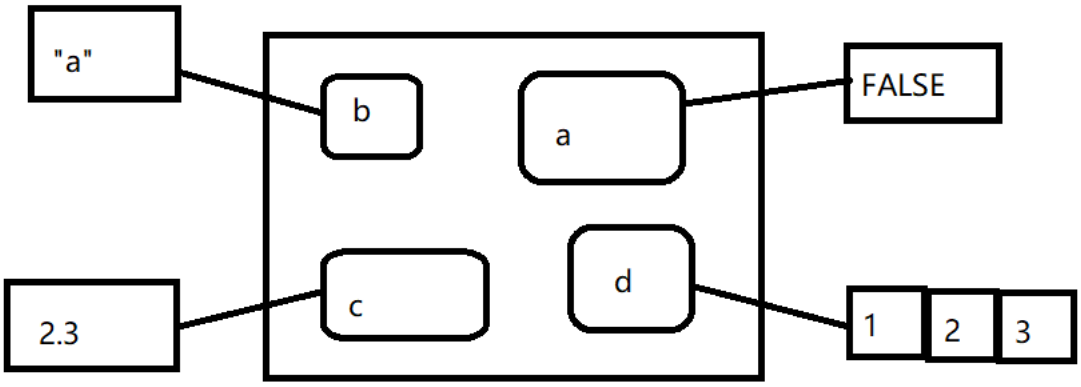
7.1 环境基础

环境的作用就是将一些名字与一些值进行关联，或者绑定 (bind), 可以把环境看做一个装满名字的口袋



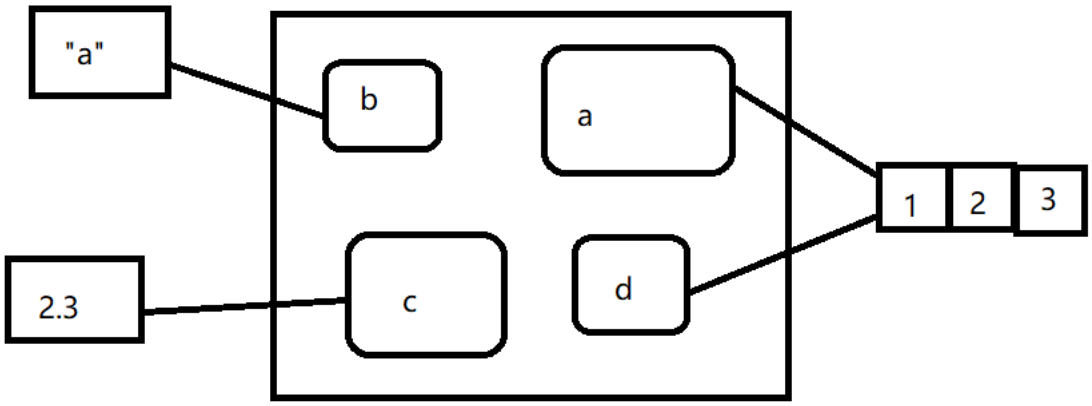
每个名字都指向存储在内存中的一个对象

```
e <- new.env()
e$a <- FALSE
e$b <- "a"
e$c <- 2.3
e$d <- 1:3
```



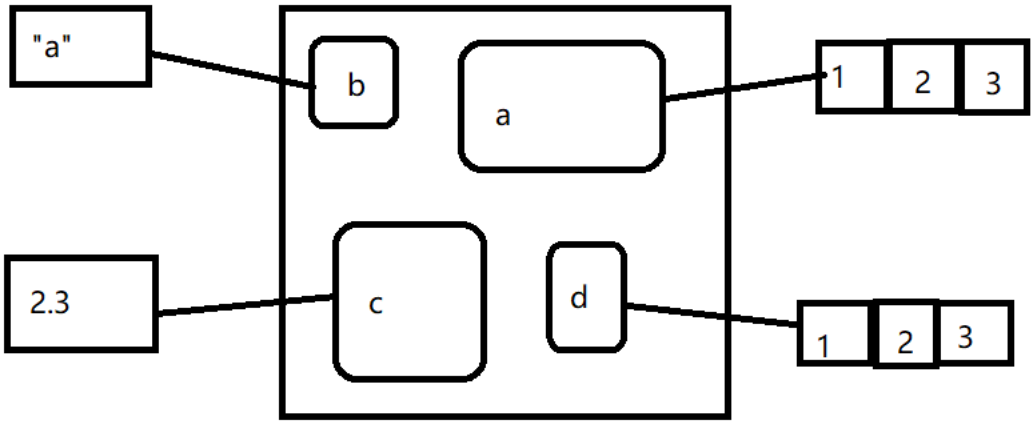
对象并不生存在环境中，所以多个名字可指向同一个对象

```
e$a <- e$d
```



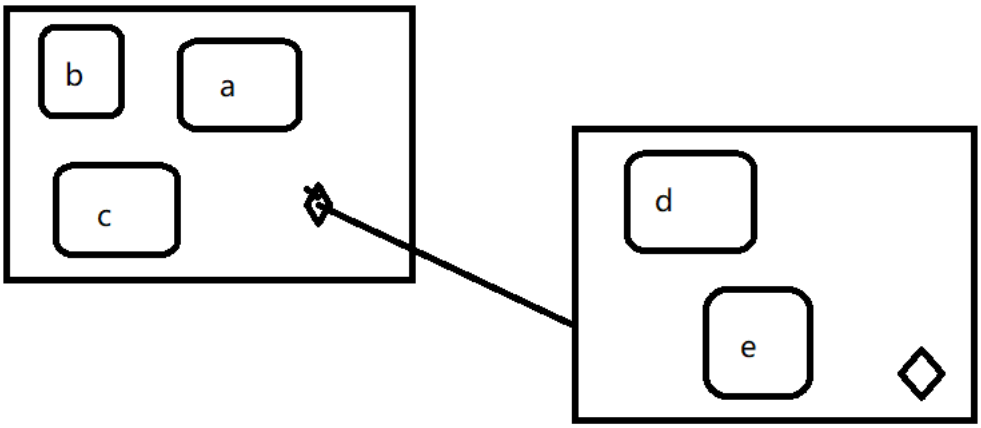
令人困惑的是，他们还可以指向具有相同值得不同对象

```
e$a <- 1:3
```



如果对象没有指向它的名字，那么这个对象就会被垃圾回收器自动删除。

每个环境都有父环境，他是另外一个环境。在下图中，黑色小圆球代表指向父环境的指针，父环境用于实现词法作用域: 如果一个名字在一个环境中没有找到，R 就会到他的父环境去找（直到找到或遍历所有环境），只有空（empty）环境没有父环境



我们可以将环境之间的关系比作家庭中成员的关系。一个环境的爷爷就是他父亲的父亲，它的祖先就包括直到空环境的所有父环境。我们基本上不会说一个环境的子环境，因为他们之间没有反向链接，给定一个环境我们没有办法找到它的子环境。

通常环境与列表相似，除一下 4 点外：

- 环境中的每个对象都有唯一的名字
- 环境中的对象没有顺序
- 环境有父环境
- 环境具有引用语义

更专业一点，环境是有两部分构成：对象框，它包含名称 -对象的绑定关系（行为商更像一个命名列表); 它的父环境。

还有 4 个特殊的环境：

- globaenv() 或者全局环境，他是一个交互式的工作环境，通常情况下我们就是在这个环境工作。全局环境的父环境就是 libtary() 或 require() 添加的最后一个包

- baseenv(), 基础环境，他是 R 基础软件包的环境，他的父环境是空环境
- emptyenv(), 空环境，他是所有环境的祖先，也是唯一一个没有父环境的环境
- environment(), 他是当前环境

下面看一些环境方面的方法：

- search()

列出全局环境的所有父环境

```
search()
```

```
##  [1] ".GlobalEnv"        "package:R6"        "package:pryr"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "package:methods"
## [10] "Autoloads"         "package:base"
```

- as.environment()

访问搜索列表中的任何环境

```
as.environment("package:stats")
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "D:/R-3.4.3/library/stats"
```

- new.env()

手动创建一个环境，ls() 可以列出将此环境的对象框中的所有绑定关系列出来，可以使用 parent.env() 查看他的父环境。

```
e <- new.env()

parent.env(e)
```

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

```
ls(e)
```

```
## character(0)
```

对一个环境中的绑定关系进行修改的最简单的方法就是将其看做列表

```
e$a <- 1
e$b <- 2
ls(e)
```

```
## [1] "a" "b"
```

```
e$a
```

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

默认情况下，ls() 只能列出不是一 "." 开始的名字，可以通过设置参数 all.names=TRUE 来显示一个环境中的所有绑定关系：

```
e$.a <- 2
ls(e)
```

```
## [1] "a" "b"
```

```
ls(e,all.names=TRUE)
```

```
## [1] ".a" "a"  "b"
```

- ls.str()

它可以将环境中的所有对象都显示出来，比 str() 更有用

```
str(e)
```

```
## <environment: 0x000000002126a890>
```

```
ls.str(e)
```

```
## a :  num 1
## b :  num 2
```

- 获取绑定值

```
e$c <- 3
e$c
```

```
## [1] 3
```

```
e["c"]]
```

```
## [1] 3
```

```
get("c",envir=e)
```

```
## [1] 3
```

- 从环境中删除对象

```
e <- new.env()
e$a <- 1
e$a <- NULL # 这样相当于创建了一个新的对象
ls(e)
```

```
## [1] "a"
```

```
rm("a",envir=e)
ls(e)
```

```
## character(0)
```

- exists()

确定一个绑定是否存在

```
x <- 10
exists("x",envir=e) # 查找父环境
```

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

```
exists("x",envir=e,inherits=FALSE) # 不希望在父环境中查找
```

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

- identical()

identical() 与 == 是不同的，是对两个环境进行比较

```
identical(globalenv(),environment())
```

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

```
#globalenv()==environment()
```

7.2 环境递归

环境可以构成一棵树，因此我们非常方便的写出一个递归函数，pryr::where() 会使用 R 的作用域法则找到定义这个名字的空间

```
library(pryr)
x <- 5
# where 函数由两个参数，一个是查找的名字 (字符串)，一个是开始查找的环境
where("x")
```

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

```
where <- function(name,env=parent.frame()){
if(identical(env,emptyenv())){
  # Base case
  stop("Can't find",name,call.=FALSE)
}else if(exists(name,envir=env,inherits=FALSE)){
  #Success case
  env
}else{
  #Recursive case
  where(name,parent.env(env))
}
}
```

有三种情况：

- 基本情况：已经到达空环境，但没有找到绑定，抛出一个错误
- 成功情况：在这个环境中存在该对象，返回该环境
- 递归情况：在这个环境中没有找到该环境的对象，所以尝试在父环境中继续查找

7.3 函数环境

大对数环境并不是通过 new.env() 函数创建的，而是使用函数的结果，本节将讨论 4 种和函数相关的环境：封闭，绑定，执行和调用。

- 封闭环境：创建函数的环境，每个函数有且仅有一个封闭环境
- 使用 <-讲一个函数和一个名字进行绑定，就可以定义一个绑定环境
- 调用函数创建一个临时的执行环境，用来存储执行期间创建的各种变量
- 每个执行环境都与一个调用环境关联，他说明函数在哪调用

7.3.1 封闭环境

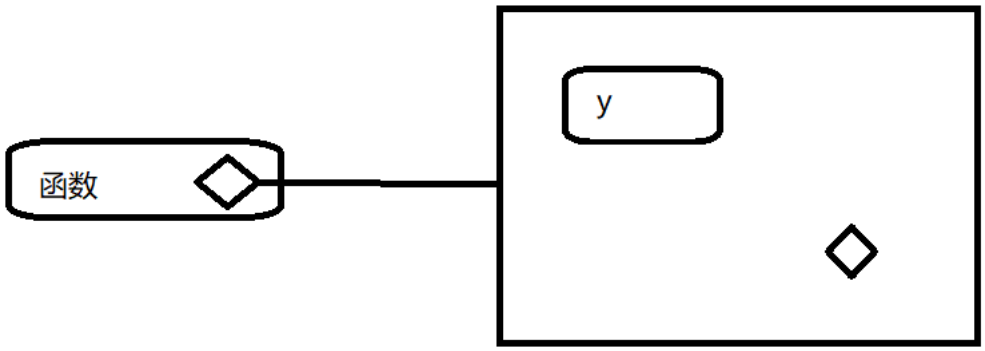
当创建一个函数，他就获得对创建他的函数的引用，这就是封闭环境，他用作此法作用域。为了确定一个函数的封闭空间，只需调用 environment() 并将函数名作为第一个参数

```
y <- 1
f <- function(x) x+y
```

```
environment(f)
```

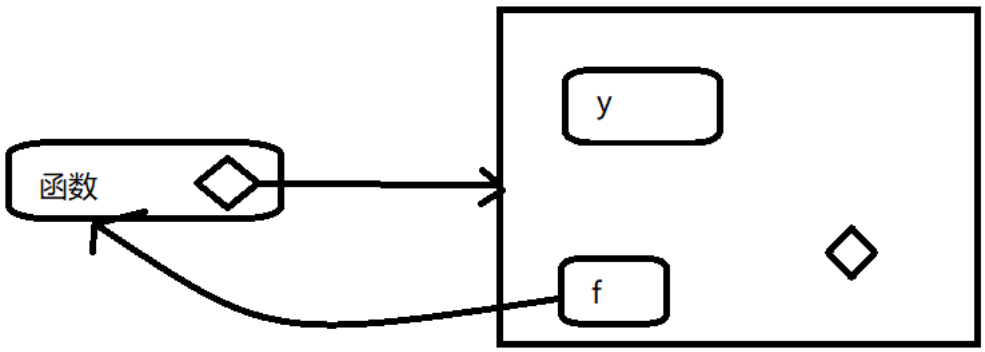
```
## <environment: R_GlobalEnv>
```

下图中，圆角矩形代表函数，黑色菱形代表一个函数的封闭环境



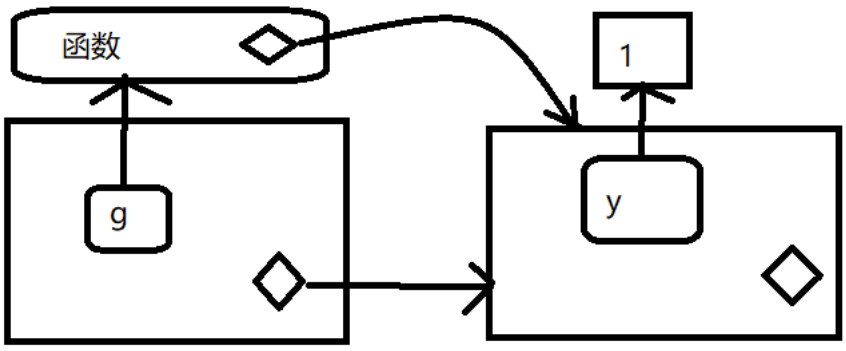
7.3.2 绑定环境

上图太简单，因为函数没有名字，函数的名字可以通过绑定来定义，一个函数的绑定环境就是与其绑定的所有环境，下图更好的反映这种关系，因为封闭环境包含一个从 f 到环境的绑定



这种情况封闭环境和绑定环境是相同的，当将一个函数分配给另一个不同的环境，那么他们就同了

```
e <- new.env()
e$g <- function() 1
```



封闭函数属于该环境，永远也不会发生改变，甚至将该函数移动到其他环境中，封闭环境决定了这个函数该如何找到值，而绑定环境空间决定如何找到该函数。

绑定空间与封闭空间的区别在于软件包命名空间是非常重要的，例如如果软件包中 A 使用基础包中的 mean() 函数，那么如果软件包 B 也创建了自己的 mean() 函数会有什么后果呢？命名空间确保软件包 A 使用基础包中的 mean(), 而不受软件包 B 的影响（除非显式的调用）

命名空间使用环境来实现，利用函数不一定存在于他们的封闭环境中的事实，例如基础包中的 sd() 函数，它的封闭环境与绑定环境是不同的

```
environment(sd)
```

```
## <environment: namespace:stats>
```

```
where("sd")
```

```
## <environment: package:stats>
## attr(,"name")
## [1] "package:stats"
## attr(,"path")
## [1] "D:/R-3.4.3/library/stats"
```

函数 sd() 的定义使用 var()，但是如果创建自己的 var() 函数，那么他也不会影响 sd()

```
x <- 1:10
sd(x)
```

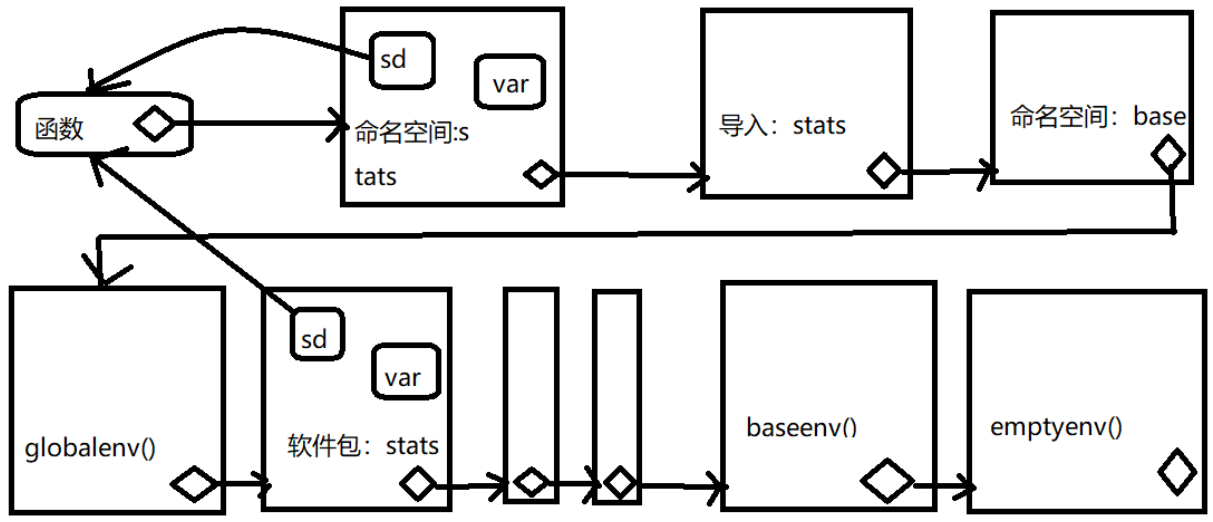
```
## [1] 3.02765
```

```
var <- function(x,na.rm=TRUE) 100

sd(x)
```

```
## [1] 3.02765
```

这是可执行的，因为每个软件包有两个与他相关的环境：软件包环境和命名空间环境。软件包环境包含所有可以访问的公共函数，并且存放在搜索路径上，命名空间环境包含所有函数，并且它的父环境也是比较重要的环境，其中包含了这个软件包需要的所有函数的绑定。软件包中的每一个到处函数都绑定到软件包环境，但都在命名空间环境中，如下图展示杉树关系：



当输入 var 时，R 首先会到全局环境中进行查找，当 sd() 查找 var() 时，他首先到命名空间中查找，而永远不会到 globalenv() 中查找。

7.3.3 执行环境

第一次执行下面的函数他返回什么?，第二次呢?

```
g <- function(x) {
  if(!exists("a",inherits=FALSE)){
    message("Default a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g(10)
```

```
## Default a
```

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

```
g(10)
```

```
## Default a
```

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

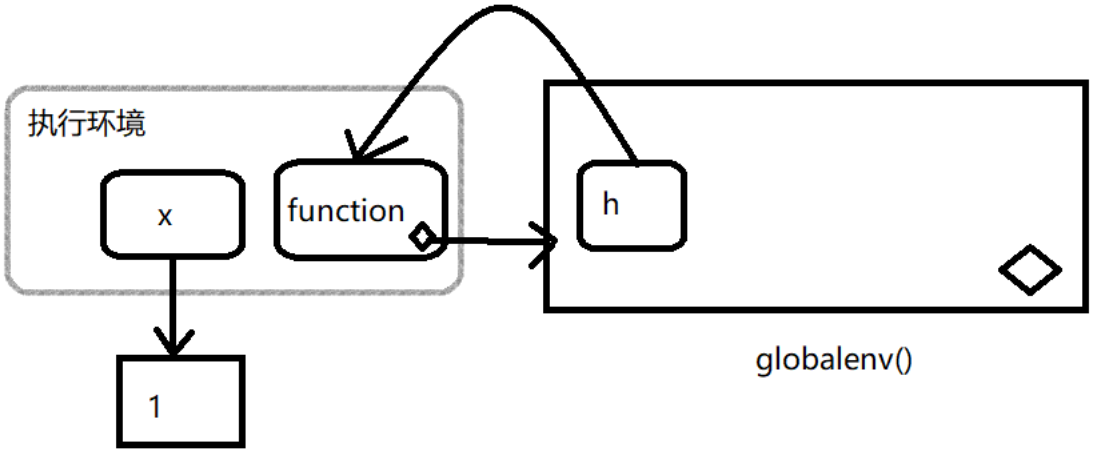
每次调用这个函数都返回相同的值，每次调用函数时都创建一个新的宿主执行环境，执行环境的父环境就是函数的封闭环境，一旦函数执行结束，这个环境就会被销毁。

通过图形简单了解这个过程：虚线包围的就是执行环境

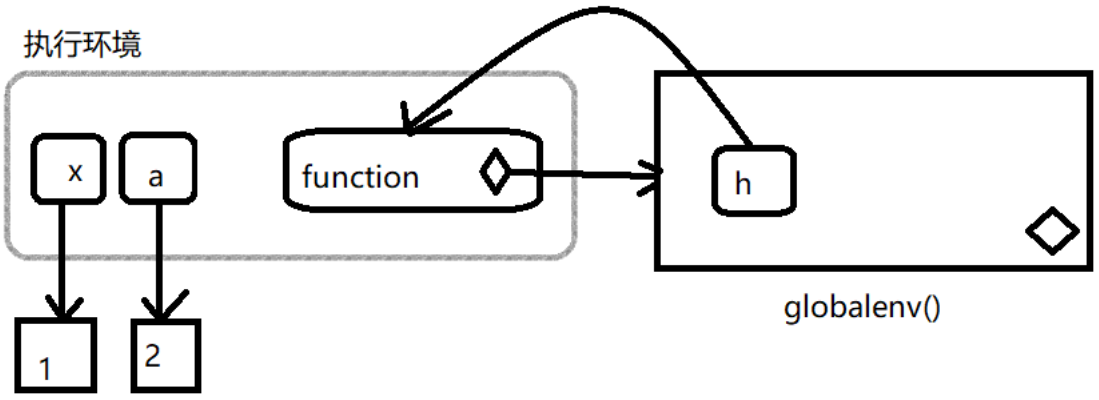
```
h <- function(x) {
  a <- 2
  x + a
}

y <- h(1)
```

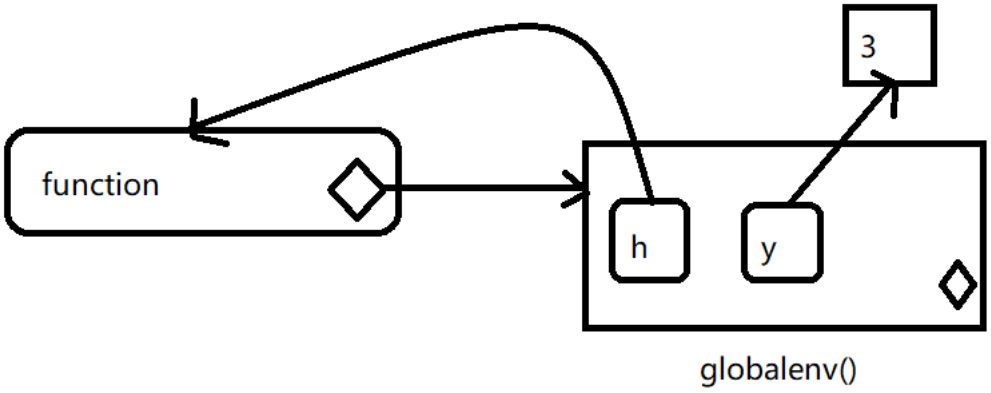
1. 用 $x = 1$ 调用函数



2.a 被赋值为 2



3. 函数完成后返回值为 3



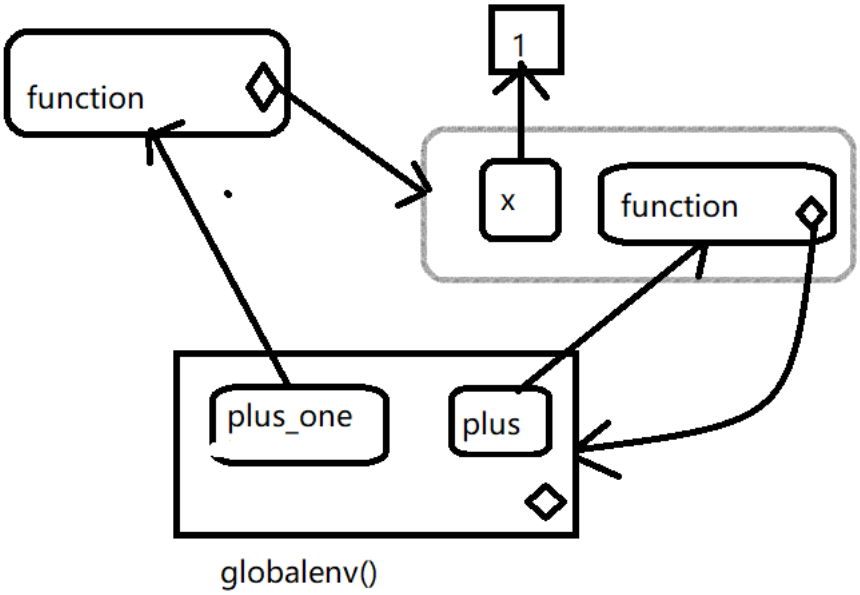
当在另一个函数中创建一个函数的时候，子函数的封闭函数就是父函数的执行环境，而且执行环境也不再是临时的了。下面的栗子用一个函数工厂 (相当于 python 闭包，装饰器)plus() 来说明这个想法。我们使用这个工厂创建一个 plus_one() 函数，plus_one() 的封闭环境是 plus() 的执行环境，其中 x 与数值 1 绑定

```
plus <- function(x){
  function(y) x+y
  message("Note: ","plus_one also a function")

}
plus_one <- plus(1)
```

```
## Note: plus_one also a function
```

```
# 注意 plus_one 也是一个函数
```



7.3.4 调用环境

查看下面代码。当代码运行时，你期望 i() 返回什么？

```
h <- function(){
  x <- 10
  function(){
    def <- get("x",environment())
    # 返回函数被调用的环境
    call <- get("x",parent.frame())
    list(defined=def,called=call,x=x)
    #x
  }
}

i <- h()
x <- 20
i()
```

```
## $defined
## [1] 10
##
## $called
## [1] 20
##
## $x
## [1] 10
```

```
#[1] 10
```

最外层的 x（绑定到 20）是为了分散你的注意力：使用普通作用域法则，h() 首先找到自己是在哪里定义的，然后再寻找与 x 相关联的值 10，但是在调用 i() 的环境中与 x 相关的值是多少？在定义 h() 的环境中 x 为 10，在调用 h() 的环境中 x 为 20.

注意每个执行环境都有两个父环境：一个调用环境和一个封闭环境。R 普通作用域法则只使用封闭的父环境，parent.frame() 允许你访问调用父环境。在调用环境而不是封闭环境中查找变量称为动态作用域 (注：很少的编程语言使用动态作用域，这是因为动态调用使我们更难理解函数式如何运行的，动态调用主要用于开发交互式数据分析的函数)

7.4 绑定名字和数值

赋值操作其实就是就是将一个名字和一个值进行绑定，它对应于作用域，这个规则决定如何找到与一个名字相关联的值。可能你已经使用过上千次的 R 赋值语句，赋值操作为当前环境中的名称和对象建立一种绑定关系。名字通常包括字母，数字，. 和 _, 但是不能以 _ 开头，如果不遵守这些规则，则会出错。

```
_abc <- 1
#Error: unexpected input in "_"
```

使用?Reserved 可以获取完整的保留字列表，这些通常的规则也可以被重写。在一个由任何字符构成的名称的两遍加上反引号，就可以应用该名称了。

```
`a=b` <- 3
`:)` <- "smile"
ls()
```

```
##  [1] ":)"                "a"                 "A"
##  [4] "a=b"               "Animal"            "area"
##  [7] "b"                 "B"                 "b2"
## [10] "Book"              "C"                 "c1"
## [13] "c2"                "cat"               "Cat"
## [16] "cc"                "cc2"               "circum"
## [19] "conan"             "connan"            "Dog"
## [22] "Duck"              "e"                 "e1"
## [25] "e2"                "f"                 "f1"
## [28] "f1.character"      "f1.numeric"        "father"
## [31] "g"                 "genderFactor"      "getShape"
## [34] "h"                 "i"                 "j1"
## [37] "Java"              "m1"                "m2"
## [40] "manager"           "member"            "Member"
## [43] "mother"            "n1"                "n2"
## [46] "np6"               "NR6"               "p1"
## [49] "Person"            "Php"               "plus"
## [52] "plus_one"          "pr6"               "PR6"
## [55] "R"                 "r1"                "rc"
## [58] "RC"                "s"                 "s1"
## [61] "son"               "student"           "student.attend"
## [64] "student.default"   "student.exam"      "student.homework"
## [67] "teacher"           "teacher.assignment" "teacher.correcting"
## [70] "teacher.default"   "teacher.lecture"   "u1"
## [73] "u2"                "u3"                "User"
## [76] "var"               "work"              "Worker"
## [79] "x"                 "y"
```

除了使用反引号外，还可以使用单引号或双引号来创建非语法的绑定，但是不推荐这样做 (在赋值箭头左边使用字符串属于历史问题，在 R 开始支持反引号之前就已经开始使用了)

普通的赋值箭头 <-总是在当前环境中创建一个变量。强制赋值箭头 <<-不会再当前环境中创建变量，但是他修改父环境中已有的变量，也可以使用 assign 来进行深度绑定：name <<- value 就等价于 assign("name",value,inherits=TRUE)

```
x <- 0
f <- function(){
  x <<- 1
}

f()
x
```

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

还有另外两个绑定：延时绑定和主动绑定

- 延时绑定：不是立即把结果赋给一个表达式，它创建和存储一个约定，在需要时对约定中的表达式进行求值，用特殊的赋值运算符号%<d-% 来创建延迟绑定, 他是对 delayedAssign() 函数的封装，如果需要更多的控制可以使用该函数。

```
library(pryr)
system.time(b %<d-% {Sys.sleep(1); 1})
```

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

```
system.time(b)
```

```
##    user  system elapsed
##       0       0       1
```

- 主动绑定：不是绑定到常量对象，相反，每次对其进行访问时都要重新计算。%<a-% 是对基础函数 makeActiveBinding() 的封装。

```
x %<a-% runif(1)
x
```

```
## [1] 0.6131557
```

```
x
```

```
## [1] 0.6850947
```

```
rm(x)
```

7.5 显式环境

除了服务于作用域之外，环境也是一种很有用的数据结构，因此他们有引用语义，与 R 中的大多数对象不同，当你对环境进行修改时，R 不会对其进行复制，例如：

```
modify <- function(x){
  x$a <- 2
  invisible()
}
```

如果将这个函数应用于列表，原始列表不会被改变，因为修改列表实际上是创建和修改副本

```
x_1 <- list()

x_1$a <- 1

modify(x_1)
x_1$a
```

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

```
#[1] 1
```

但是如果将这个函数应用于环境，那么原始环境就会被修改

```
x_e <- new.env()
x_e$a <- 1
modify(x_e)
x_e$a
```

```
## [1] 2
```

```
#[1] 2
```

就像可以使用列表传递数据一样，也可以使用环境，当你创建自己的环境时，应该将父环境设置为空环境，这样确保不会从其他地方继承对象：

```
x <- 1

e1 <- new.env()

get("x",envir=e1)
```

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

```
# e2 <- new.env(parent=emptyenv())
# get("x",envir = e2)

#Error in get("x", envir = e2) : object 'x' not found
```

环境时解决下面 3 类常见问题的有效数据结构

- 避免大数据的复制
- 管理一个软件包的内部状态
- 根据名字高效的查找与其绑定的值

7.5.1 避免复制

由于环境具有引用语义，所有绝不会无意识的创建一个副本。Biocinductor 包中经常使用这种技术，因为它经常需要对非常大的基因对象进行管理。在 R3.1.0 版本后，这种技术已经不像以前那么重要了，因为修改列表不再是深度复制了。以前修改列表的一个元素也要复制整个列表，如果有些元素非常大，就会造成昂贵的操作。

7.5.2 软件包状态

显式环境在软件包中很有用，因为他们允许你在函数调用之间保持软件包的状态，正常情况下软件包的对象是被锁定的，所以你不能直接修改他们，但是可以这样做：

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1

get_a <- function(){
  my_env$a
}

set_a <- function(value){
  old <- my_env$a
```

```
  my_env$a <- value
  invisible(old) # 不返回 old
}
```

7.5.3 模拟 hashmap

hashmap 是一种非常有用的数据结构，它根据名字查找对象的时间复杂度为 O(1)。环境默认提供这种行为，所以可以用它来模拟 hashmap。CRAN 的 hash 包就是用这种思想开发的

7.6 总结

截止到这，我们的 R 语言面型对象的内容介绍完了，回顾起来，首先我们介绍了 R 语言编程的一些规范，告诉读者要参考一些标准的规范构建自己的编程规范并在团队中做出调整，增加自己代码的可读性和运行效率，紧接着我们分了 5 章系统全面的介绍了 R 语言的面向对象编程:S3,S4,R5,R6; 最后本章我们介绍了 R 语言环境的相关知识，环境不仅是作用域的描述，更是一种非常好用的数据结构。

最后希望阅读本电子书的 R 语言用户, 阅读完本书后对你有一点点的帮助。

Chapter 8

参考文献

[1]. 高级 R 语言编程指南 [M]. hadley Wickham 著，李洪成等译.
[2].R 语言面向对象编程
[3]. 数据科学中的 R 语言 [M]. 李舰，肖凯著.