面向对象的 R 程序设计

Object Oriented R Programming

Humoon

2019-08-29

目录


```
library(R6)
```

1 R6 类和对象语法

1.1 创建 R6 类和实例化对象语法

1.1.1 创建 R6 类：

R6Class(classname = NULL, public = list(), private = NULL, active = NULL, inherit = NULL, lock = TRUE, class = TRUE, portable = TRUE, parent_env = parent.frame())

- classname, 类名
- public, 定义公有成员，包括属性和方法
- private, 定义私有成员，包括属性和方法
- active, 主动绑定的函数列表
- inherit, 定义父类，继承关系
- lock, 是否上锁，上锁则用于类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
- portable, 是否为可移植类型，默认是，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

1.1.2 实例化 R6 类对象：

R6 类名 $new()

1.2 公有成员和私有成员

```
Person <- R6Class("Person",
  # 公有成员
  public = list(
    # 公有属性name
    name = NA,
    # 公有方法 1: 实例化对象的赋值函数
    # initialize()是一个初始化函数
    # 实例化对象时 类名$new() 的参数，就传递给了这个initialize()
    initialize = function(str_name, str_gender){
```

```
      # 在类的内部访问公有成员，需要借助self对象来调用
      self$name <- str_name
      # 调用私有属性和私有方法，都需要借助private对象来调用
      private$gender <- str_gender
    },
    # 公有方法 2: hello函数打印name
    hello = function(){
      print(paste("Hello,", self$name))
      private$myGender()
    },

    # 公有方法 3: 测试self和private对象到底是什么
    member = function(){
      print(self)
      print(private)
      print(ls(envir = private))
    }
  ),

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

```
# 查看Person的定义
Person
```

```
<Person> object generator
  Public:
    name: NA
```

```
    initialize: function (str_name, str_gender)
    hello: function ()
    member: function ()
    clone: function (deep = FALSE)
  Private:
    gender: NA
    myGender: function ()
  Parent env: <environment: R_GlobalEnv>
  Locked objects: TRUE
  Locked class: FALSE
  Portable: TRUE
```

```
# 检查Person的类型
class(Person)
```

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

```
# 实例化对象
p1 <- Person$new('Conan', "Male")
p1
```

```
<Person>
  Public:
    clone: function (deep = FALSE)
    hello: function ()
    initialize: function (str_name, str_gender)
    member: function ()
    name: Conan
  Private:
    gender: Male
    myGender: function ()
```

```
class(p1)
```

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

```
p1$hello()
```

```
[1] "Hello, Conan"
[1] "Conan is Male"
```

```
# 公有成员可以直接访问；私有成员一旦赋值完成后，对外界是不可见的
p1$name
```

```
[1] "Conan"
```

```
p1$gender # 访问私有属性，返回NULL
```

```
NULL
```

```
# p1$myGender() # 访问私有方法，直接报Error
```

```
p1$member()
```

```
<Person>
  Public:
    clone: function (deep = FALSE)
    hello: function ()
    initialize: function (str_name, str_gender)
    member: function ()
    name: Conan
  Private:
    gender: Male
    myGender: function ()
<environment: 0x00000000318529d8>
[1] "gender"   "myGender"
```

可见，self 对象，就是实例化的对象本身。private 对象则是一个的环境空间，是 self 对象所在环境空间的中的一个子环境空间，所以私有成员只能在当前的类中被调用，外部访问私有成员时，就会找不到。在环境空间中保存私有成员的属性和方法，通过环境空间的访问控制让外部调用无法使用私有属性和方法，这种方式是经常被用在 R 包开发上的技巧。

1.3 主动绑定的函数

```
Calculate <- R6Class("Calculate",
   public = list(
     num = 100
   ),
   active = list(
     # 主动绑定的第一个函数，有参数
     alter = function(value) {
       # 若参数缺失
       if (missing(value)) return(self$num + 10)
       # 若参数得到了定义
       else self$num <- value/2
     },
     # 主动绑定的第二个函数，无参数
     rand = function() rnorm(1)
   )
)

exam <- Calculate$new()
exam$num
```

```
[1] 100
```

```
# 调用主动绑定的alter()函数
# 由于没有参数，结果为 num  10 = 100  10 = 110
exam$alter
```

```
[1] 110
```

注意，因为 alter 在 active 中被绑定了，虽然它本来是一个函数，在这里却像一个属性一样被引用[1] 。

```
exam$alter <- 100 # exam$alter又被当做了元素，但所赋的值100实际传给了alter()的参数value
exam$num # 查看公有属性num，显然上一句alter()就被调用运行了
```

```
[1] 50
```

```
exam$alter # 调用主动绑定的active()函数，结果为 num10=5010=60
```

```
[1] 60
```

```
# exam$alter(100) # 如果用函数方式来调用，会提示没有这个函数的

exam$rand # 调用rand函数
```

```
[1] 0.9588
```

```
exam$rand
```

```
[1] -0.05427
```

```
# exam$rand <- 99 # 传参出错，因为rand()不接受参数
```

主动绑定的主要作用似乎是把函数的行为转换成属性的行为，让类中的函数操作更加灵活。

1.4 继承（子类对父类）

```
Worker <- R6Class(
  "Worker",
  inherit = Person, # 继承，指向父类
```

[1] 像例 1 中的 hello()，被 p1 引用时就是函数形式。

```
  public = list(
    # 子类中的新函数
    bye = function(){
      print(paste("Bye bye",self$name))
    }
  ),
  private = list(
    gender = NA,
    # 子类中重写、覆盖旧函数
    myGender = function(){
      super$myGender() # 子类中调用父类的方法，通过super对象
      print(paste("Worker",self$name,"is",private$gender))
    }
  )
)

p1 <- Person$new("Conan","Male") # 实例化父类
p1$hello()
```

```
[1] "Hello, Conan"
[1] "Conan is Male"
```

```
p2 <- Worker$new("Conan","Male") # 实例化子类
p2$hello()
```

```
[1] "Hello, Conan"
[1] "Conan is Male"
[1] "Worker Conan is Male"
```

```
p2$bye()
```

```
[1] "Bye bye Conan"
```

1.5 R6 类对象的静态属性

即可以在多个实例中通过共同链接到一个对象的方式共享该对象的属性，且实时联动，不会产生数据冗余。

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

# B类，其属性是一个A类对象
B <- R6Class("B",
  public = list(
    a = A$new()
  )
)

b1 <- B$new() # 实例化B对象
b1$a$x <- 1 # 给x变量赋值
b1$a$x # 查看x变量的值
```

```
[1] 1
```

```
b2 <- B$new() # 实例化b2对象
b2$a$x <- 2 # 给x变量赋值
b2$a$x # 查看x变量的值
```

```
[1] 2
```

```
b1$a$x # b1实例的a对象的x值也发生改变
```

```
[1] 2
```

从输出结果可以看到，a 对象实现了在多个 b 实例的共享。即**不同的 B 类对象，其属性是同一个 A 类对象**。

需要注意的是，不能这样写：

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

cc1 <- C$new()
cc1$a$x <- 1
cc1$a$x
```

```
[1] 1
```

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

```
[1] 2
```

```
cc1$a$x # x值未发生改变
```

```
[1] 1
```

通过 initialize() 构建的 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。

1.6 R6 类的可移植类型

- 可移植类型支持跨 R 包的继承；不可移植类型，在跨 R 包继承的时候，兼容性不太好。

- 可移植类型必须要用 self 对象和 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 <<-实现赋值。

1.7 R6 类结构的修改

定义一个 R6 类后，后续还可以通过类名 `$set()` 追加定义属性和方法。如：

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

# 动态修改
A$set("public", "x", 10, overwrite = T)
A$set("public", "getx2", function() self$x*10, overwrite = T)

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

```
<A>
  Public:
    clone: function (deep = FALSE)
    getx: function ()
    getx2: function ()
    x: 10
```

```
s$x
```

```
[1] 10
```

```
s$getx()
```

```
[1] 10
```

```
s$getx2()
```

```
[1] 100
```

2 综合案例：构建一个图书分类的使用案例

2.1 任务一：定义图书的静态结构

以图书 (book) 为父类，包括 R，Java，PHP 的 3 个分类，在 book 类中定义私有属性及公有方法，继承关系如下：

定义图书系统的数据结构，包括父类的结构和 3 种型类的图书:

```
# 父类
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
    }
  )
)
```

```
# 子类R图书
R <- R6Class("R",
  inherit = Book
)
# 子类JAVA图书
Java <- R6Class("JAVA",
  inherit = Book
)
# 子类PHP图书
Php <- R6Class("PHP",
  inherit = Book
)

# 创建3个实例
r1 <- R$new("R的极客理想-工具篇", 59, "R")
r1$getPrice()
```

```
[1] 59
```

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

```
[1] 108
```

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

```
[1] 98
```

2.2 任务二：结合问题实际修改类的结构

正逢双 11 对各类图书打折促销，假设有如下的打折规则

- 所有图书至少 9 折起，再此基础上：

- JAVA 图书 7 折，但不支持重复打折
- 为了推动 R 图书的销售，R 语言图书享受折上折，在现有基础上再打 7 折
- PHP 图书无更多优惠

根据打折规则，图书都可以被打折，那么打折就可以作为图书对象的一个行为，R, Java, PHP 的 3 类图书，分别还有自己的打折规则，所以是一种多态的表现。

我们修改父类的定义，增加打折的方法 discount()，默认设置为 9 折，满足第一条规则。

```
## 为修改父类，添加属性和方法
# 添加新方法：折扣
Book$set("public", "discount", function() 0.9, overwrite = T)
# 覆盖就方法：输出
Book$set(
  "public",
  "getPrice",
  function() {
    p <- private$price*self$discount()
    print(paste("Price:", private$price, ", Sell out:", p, sep = ""))
    },
  overwrite = T
)

# Java类覆盖父类的discount()
Java <- R6Class("JAVA",
  inherit = Book,
  public = list(
    discount=function() 0.7
  )
)

# R类通过super$调用父类的discount()
```

```
R <- R6Class("R",
  inherit = Book,
  public = list(
    discount=function(){
      super$discount() * 0.7
    }
  )
)

# PHP类直接继承父类
Php <- R6Class("PHP",
  inherit = Book
)

# 查看打折结果
r1<-R$new("R的极客理想-工具篇", 59, "R")
r1$getPrice()
```

```
[1] "Price:59, Sell out:37.17"
```

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

```
[1] "Price:108, Sell out:75.6"
```

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

```
[1] "Price:98, Sell out:88.2"
```