R 学习笔记

R 中的 S4 方法

胡荣兴 hurongxing@126.com

R 主要面向统计计算，似乎很少会用到面向对象的编程方法。但在统计计算中，在下列情形中使用面向对象的编程方法可以编程更有效率。

1）当需要用一种新的方式来表示数据，该方式与已有的数据类型有区别的时候。

2）当需要一个新的函数，该函数可以根据不同的参数类型做出不同的反应的时候。

在 R 中，经常需要定义一个新的函数，并且定义一个新的函数也是一项繁重的工作。相反，较少去定义一个新的类。但有时候定义一个类是一个很关键的步骤。一个类通常决定了如何对对象进行处理，决定了对象中应当包含什么样的信息。甚至有时候，类的定义决定你的项目的成败。

1. 旧的实现方法

R 的*面向对象(OOP)*是基于*泛型函数(generic function)*的，而不是基于类层次结构。什么是 R 中的泛型函数？R 中的面向对象是怎么样的呢？我们首先看下面来自使用 R 编写统计程序,第 3 部分:可重用和面向对象中的一个实例。

下面我们首先来创建一个泛型函数 whoAmI( )

```
# 创建泛型函数
> whoAmI <- function(x, ...) UseMethod("whoAmI")
> whoAmI.foo <- function(x) print("I am a foo")
> whoAmI.bar <- function(x) print("I am a bar")
> whoAmI.default <- function(x) print("I don't know who I am")
```

在 R 中，每个对象都属于 0 个或 1 个或多个类，我们可以用 class( )函数查看对象所属的类。R 中的类可参考附录 1：R 中的基本类

```
> a = 1:10
> b = 2:20
> class(a)
[1] "integer"
> class(b)
[1] "integer"
```

有了泛型函数和类之后，我们就可以看到 R 中的面泛型函数是如何工作的

```
> whoAmI(a)                    # a 的类为“integer”，没有为该类定义方法，
                               #  就执行 whoAmI.default 方法
[1] "I don't know who I am"
```

```
> attr(a,'class') <- 'foo'              # 用 attr( )函数将 a 的类指定为“foo”
> class(a)
[1] "foo"
> attr(b,'class')<-c('baz','bam','bar') #指定 b 的属于三个类'baz','bam','bar'
> class(b)
[1] "baz" "bam" "bar"
> whoAmI(a)                           # 现在 a 属于’foo’类，执行 whoAmI.foo( )
[1] "I am a foo"
> whoAmI(b)                           # b 虽然属于三个类，但只对’bar’类定义了
                                      # 方法，所以就执行 whoAmI.bar()
[1] "I am a bar"
> attr(a,'class') <- 'bar'            # 改变 a 的类
> class(a)
[1] "bar"
> whoAmI(a)
[1] "I am a bar"
```

上面的例子中提到了一个对象同时属于多个类的情况，再看下面的一个补充实例。

```
> meth1 <- function(x) UseMethod("meth1")
> meth1.Mom <- function(x) print("Mom's meth1")# meth1 对类’Mom’
> meth1.Dad <- function(x) print("Dad's meth1")   # 和类’Dad’都定义了方法
> meth2 <- function(x) UseMethod("meth2")         # meth2 只对类
> meth2.Dad <- function(x) print("Dad's meth2")   # ‘Dad’定义了方法
> attr(a,'class') <- c('Mom','Dad')
> meth1(a)                          # 执行了 meth1.Mom( ),因为 Mom 在前。
[1] "Mom's meth1"
> meth2(a)
[1] "Dad's meth2"
```

包含祖先的一个例子。

```
#包含祖先
char0 = character(0)
makeMRO <- function(classes=char0, parents=char0) {
    # Create a method resolution order from an optional
    # explicit list and an optional list of parents
    mro <- c(classes)
    for (name in parents) {
        mro <- c(mro, name)
        ancestors <- attr(get(name),'class')
        mro <- c(mro, ancestors[ancestors != name])
    }
    return(mro)
}
NewInstance <- function(value=0, classes=char0, parents=char0) {
    # Create a new object based on initial value,
```

```
    # explicit classes and parents (all optional)
    obj <- value
    attr(obj,'class') <- makeMRO(classes, parents)
    return(obj)
}
MaternalGrandma <- NewInstance()
PaternalGrandma <- NewInstance()
Mom <- NewInstance(classes='Mom', parents='MaternalGrandma')
Dad <- NewInstance(0, classes=c('Dad','Uncle'), 'PaternalGrandma')
Me <- NewInstance(value='Hello World', 'Me', c('Mom','Dad'))

> print(Me)
[1] "Hello World"
attr(,"class")
[1] 'Me"                  "Mom"                  "MaternalGrandma"  'Dad"
[5] 'Uncle"               "PaternalGrandma"
```

从上面的例子，我们可以看到，R 中的“面向对象”是以类和泛型函数为基础，类可以多重继承类，泛型函数可以分派到任意的参数（签名,signature）集合，泛型函数将不同的方法聚集到一起，由 R 根据函数的对象的类型来决定选择执行哪个方法。在某些情况下，类和方法是不同的概念，我样要区别对待。

2.类

用 setClass 来定义一个类。

```
setClass(Class, representation, prototype, contains=character(),
         validity, access, where=1, version=FALSE)
```

其中：

Class	字符串，类的名称
representation	新的类的接口，或者该类扩展的其它类。通常是对 representation 函数的一个调用。
prototype	为接口提供的默认的数据对象。如果在构造一个新的类的时候，不明确地改写，那么新的实例将采用这些数据。
contains	该类所扩展的类，所有扩展的类的新的实例都会继承其所有父类的接口
validity, access, version	与 S-Plus 兼容的控制参数
where	存储或移除定义的元数据的环境

定义了一个类后，就可以用函数 new()来生成一个类的实例(instances)。类定义了一个对象的结构。类的实例代表了对象本身。一个子类是对其父类的扩展，一个新的类将包含其父类的所有接口。在创建一个新的类时，接口的名字不能重复（包括其直接或间接的父类）。下面的代码将创建两个类 foo 和 bar，再创建它们的一个子类 baz。

```
> library(methods)
> setClass("foo",representation(a="character",b="numeric"))
```

```
[1] "foo"
> setClass("bar", representation(d = "numeric", c = "numeric"))
[1] "bar"
> setClass("baz", contains = c("foo", "bar"))
[1] "baz"
> getClass("baz")
Class “baz” [in ".GlobalEnv"]

Slots:

Name:          a          b          d          c
Class:  character    numeric    numeric    numeric

Extends: "foo", "bar"
```

现在我们可以生成类 baz 的一个实例。

```
> x <- new("baz", a = "xxx", b = 5, c = 10)
> x
An object of class “baz”
Slot "a":
[1] "xxx"

Slot "b":
[1] 5

Slot "d":
numeric(0)

Slot "c":
[1] 10
```

可以通过操作符@访问接口中的数据。但最好的办法是不直接访问接口的数据，而是定义一些特殊的方法来访问这些数据。

```
> x@a
[1] "xxx"
```

3. 虚类

虚类就是不会用来生成实例的类。它们是用来将拥有不同的 representations 的类（它们不能互相继承）链接起来，可以通过虚类为这些不同的 representations 提供相似的功能。一个标准的做法就是建立一个虚类，然后用其它类来扩展它。

在实际中，可以通过以下几种不同的方式来使用该虚类。

1. 虚类的方法将用到其所有的扩展类中。
2. 新的类的接口将与虚类的接口的数据类型一致，这样可以使接口的多样

化。

3. 虚类的接口将会出现在它的所有扩展类中。

下面我们用类来表示，一个树状图。一个树状图的结点有三个值：高度、左结点、右结点。终端结点与之不同，有一个高度和一个值（也有可能是另一个对象的实例。）我们可以用一个虚类 dendNode 来表示该结构。我们可以再定义两个该类的扩展类：终端和非终端结点。

```
> setClass("dendNode")
[1] "dendNode"
> setClass("dnode", representation(left = "dendNode", right = "dendNode",
+ height = "numeric"), contains = "dendNode")
[1] "dnode"
> setClass("tnode", representation(height = "numeric", value = "numeric",
+ label = "character"), contains = "dendNode")
[1] "tnode"
```

现在我们就可以创建树了，它的结点或者是终端结点或者是非终端节点。在类 dnode 中，它的左节点和右节点都是 dendNode 类的实例，而 dnode 类本身又是 dendNode 的扩展。在此例中虚类被用来允许两个不同的类：左节点和右节点。这种设计使得可以重复使用树变得简单。

在实际中，我们经常会出现这样的问题，希望一个接口或者是一个列表或者是一个空对象（NULL( )）。因为一个 NULL( )本身不是一个列表，所以它们不能共享同一个接口。我们可以创建一个虚拟类来扩展列表类和 NULL( )类。

有两种方法可以建立虚类。第一种方法，与上面一样，用 setClass( )建立一个没有 representation 的类。第二种方法，在参数 representation 中包括类“VIRTUAL”

```
> setClass("myVclass", representation(a = "character", "VIRTUAL"))
[1] "myVclass"
> getClass("myVclass")
Virtual Class “myVclass” [in ".GlobalEnv"]

Slots:

Name:          a
Class: character
```

4. 初始化和原型

有两种办法可以用来控制类的实例生成的对象的初始值。

第一种方法：可以用参数 prototype 来控制生成的实例的值。使用该参数，可以将任何接口赋初值。

```
> setClass("xx", representation(a = "numeric", b = "character"),
+ prototype(a = 3, b = "hi there"))
[1] "xx"
```

```
> new("xx")
An object of class “xx”
Slot "a":
[1] 3

Slot "b":
[1] "hi there"
```

第二种方法是为该类指定一个 initialize 方法，用该方法为类的实例赋值。

```
> setMethod("initialize", "xx", function(.Object, b)
+ {
+ .Object@b <- b
+ .Object@a <- nchar(b)
+ .Object
+ })
[1] "initialize"
> new("xx", b = "yowser")
An object of class “xx”
Slot "a":
[1] 6

Slot "b":
[1] "yowser"
```

注意在第二种方法中，应当在 initialize 方法中返回该对象。

5. 泛型函数和方法

面向对象编程的一个重要方面就是泛型函数。泛型函数实际上就是一个分派机制。方法就是一种特殊的函数，它能根据特定的输入对象执行需要执行的任务。泛型函数的工作就是针对不同的输入参数决定什么样的方法被执行。一旦对泛型函数进行了签名，那么泛型函数就不能添加新的参数了。在设计泛型函数的参数时，要将可能用到的参数都包括进来。

我们来看下面的一个例子。在该例中，我们将定义一个名为 whatIs( )的函数，该函数返回对象的类型。

```
> whatIs <- function(object) data.class(object)
> whatIs(1:10)
[1] "numeric"
> whatIs(matrix())
[1] "matrix"
> whatIs(whatIs)
[1] "function"
```

上面的程序好像太简单，我们还可以打印出对象的长度。

```
> whatIs <- function(object) cat("类：",
+ data.class(object),"\n 长度：",length(object),"\n")
> whatIs(1:10)
```

```
类： numeric
长度： 10
> whatIs(matrix())
类： matrix
长度： 1
> whatIs(whatIs)
类： function
长度： 1
> A<-matrix(1:6,c(2,3))
> A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> whatIs(A)
类： matrix
长度： 6
```

现在看来，又改进了一步，但是还是有点不尽如人意的地方，这里给出了函数的长度，意义不大。对于矩阵，我们更想知道它的行数和列数。所以我们需要为 function 类和 matrix 类单独定义函数，才能得到我们期望的信息：

```
> whatIs.function <- function(object)  cat("类：",
+ data.class(object),"\n")
> whatIs.function(whatIs)
类： function
> whatIs.matrix <- function(object)  cat("类：",
+ data.class(object),"\n",nrow(object),"行",ncol(object),"列\n")
> whatIs.matrix(A)
类： matrix
 2 行 3 列
```

定义方法

在上面的例子中，为了得到我们想要的信息，我们定义了三个功能类似的函数，这样程序比较散乱，使用起来也比较繁琐。我们希望通过一个统一的函数名 whatIs( )来代表上述三个函数，然后根据输入的参数的类型(function, matrix 或其它)来决定使用哪一个函数(whatIs.function( ), whatIs.function( )或 whatIs( ))。要完成该工作，我们可以使用函数 setMethod( )。

```
> setMethod("whatIs","function", whatIs.function)
用函数"whatIs"来建立新的同屬函数
[1] "whatIs"
> setMethod("whatIs","matrix", whatIs.matrix)
[1] "whatIs"
> whatIs(1:10)
类： numeric
长度： 10
> whatIs(whatIs)
```

```
类：  standardGeneric
> whatIs(A)
类：  matrix
 2 行 3 列
```

现在我们看到，我们只用一个函数 whatIs( )就可以根据参数的类型执行不同的程序代码。这里的函数 whatIs( )我们称之为泛型函数（上面的输出显示为”standardGeneric”），分别针对不同类型的对象的三个不同的程序功能称为方法（method）。

setMethod( )函数的三个参数分别告诉 setMethod 对什么泛型函数指定方法，执行该方法对应的参数的类型，执行的方法。其中第二项称为签名（signature）。

showMethods( )　查看为函数函数定义了哪些方法。

dumpMethod( )　提取为泛型函数的某个方法。

6. 访问子函数

我们可以通过@运算符来访问接口，但在实际编程中并不建议这么做。我样可以通过访问子函数来访问接口。

```
> setClass("foo", representation(a = "ANY"))   #新建一个类 foo
[1] "foo"
#申明一个标准的泛型函数 a
> if (!isGeneric("a"))
+ {
+ if (is.function("a")) fun <- a
+ else fun <- function(object) standardGeneric("a")
+ setGeneric("a", fun)
+ }
> setMethod("a", "foo", function(object) object@a)
[1] "a"
> b <- new("foo", a = 10)
> a(b)
[1] 10
```

附录 1 R 中的基本类

```
### The following are all basic vector classes.
### They can appear as class names in method signatures,
### in calls to as(), is(), and new().
"character"
"complex"
"double"
"expression"
"integer"
"list"
"logical"
"numeric"
"single"
"raw"

### the class
"vector"
### is a virtual class, extended by all the above

### The following are additional basic classes
"NULL" # NULL objects
"function" # function objects, including primitives
"externalptr" # raw external pointers for use in C code

"ANY" # virtual classes used by the methods package itself
"VIRTUAL"
"missing"

## The following class names can appear in method signatures,
## as the class in as() and is() expressions, and, except for
## the classes commented as VIRTUAL, in calls to new()

"matrix"
"array"
"ts"

"structure" ## VIRTUAL
The specific classes all extend class "structure", directly, and class
"vector", by class "structure".
```