R 与面向对象统计分析

S3、S4 对象的创建和使用

王雨晨

华东师范大学

2012 年 11 月 4 日

面向对象简介

使用已有的对象：S3 对象系统

创建新的对象：S4 及其他对象系统

S3 和 S4 对比示例

面向对象简介

使用已有的对象：S3 对象系统

创建新的对象：S4 及其他对象系统

S3 和 S4 对比示例

There is an important difference in philosophy between S (and hence R) and the other main statistical systems. In S a statistical analysis is normally done as a series of steps, with intermediate results being stored in objects.

— An Introduction to R

现实问题和计算机间的矛盾

- 计算机本质上是用来计算的, 各种高级语言提供的基本功能只能处理整数, 浮点数或者字符等.

现实问题和计算机间的矛盾

- 计算机本质上是用来计算的, 各种高级语言提供的基本功能只能处理整数, 浮点数或者字符等.
- 统计分析往往需要处理相对计算机而言非常复杂的实际问题.

现实问题和计算机间的矛盾

- 计算机本质上是用来计算的, 各种高级语言提供的基本功能只能处理整数, 浮点数或者字符等.
- 统计分析往往需要处理相对计算机而言非常复杂的实际问题.
- 面向对象编程利用对象来描述实际问题, 从而形成更直观的程序.

R 与面向对象

- 面向对象编程 (object-oriented programming) 是一种编程范式. 它将对象作为程序的基本单元, 将程序和数据封装 (encapsulate) 其中, 以提高软件的重用性, 灵活性和扩展性.

R 与面向对象

- 面向对象编程 (object-oriented programming) 是一种编程范式. 它将对象作为程序的基本单元, 将程序和数据封装 (encapsulate) 其中, 以提高软件的重用性, 灵活性和扩展性.
- R 是一种用于统计计算和绘图的程序设计语言和软件环境, 被广泛用于开发统计软件和数据分析.

R 与面向对象

- 面向对象编程 (object-oriented programming) 是一种编程范式. 它将对象作为程序的基本单元, 将程序和数据封装 (encapsulate) 其中, 以提高软件的重用性, 灵活性和扩展性.
- R 是一种用于统计计算和绘图的程序设计语言和软件环境, 被广泛用于开发统计软件和数据分析.
- R 虽然被认为是一种函数式语言, 但同样支持面向对象编程.

R 与面向对象

- 面向对象编程 (object-oriented programming) 是一种编程范式. 它将对象作为程序的基本单元, 将程序和数据封装 (encapsulate) 其中, 以提高软件的重用性, 灵活性和扩展性.
- R 是一种用于统计计算和绘图的程序设计语言和软件环境, 被广泛用于开发统计软件和数据分析.
- R 虽然被认为是一种函数式语言, 但同样支持面向对象编程.
- R 及其拓展大多以面向对象的方式实现了统计模型和算法.

类和方法

一个面向对象系统的核心是其实现的类 (class) 和方法 (method).

- 类定义了对象共同拥有的某些属性及其与其他类的关系.

类和方法

一个面向对象系统的核心是其实现的类 (class) 和方法 (method).

- 类定义了对象共同拥有的某些属性及其与其他类的关系.
- 如果一个对象属于某个类, 则称该对象是这个类的实例 (instance).

类和方法

一个面向对象系统的核心是其实现的类 (class) 和方法 (method).

- 类定义了对象共同拥有的某些属性及其与其他类的关系.
- 如果一个对象属于某个类, 则称该对象是这个类的实例 (instance).
- 方法是一种与特定类的对象关联的函数.

面向对象简介

使用已有的对象：S3 对象系统

创建新的对象：S4 及其他对象系统

S3 和 S4 对比示例

S3 简介

- S3 实现了一种基于泛型函数的面向对象方式.

S3 简介

- S3 实现了一种基于泛型函数的面向对象方式.
- 泛型函数是一种特殊的函数, 其根据传入对象的类型决定调用哪个具体的方法.

S3 简介

- S3 实现了一种基于泛型函数的面向对象方式.
- 泛型函数是一种特殊的函数, 其根据传入对象的类型决定调用哪个具体的方法.
- 面向对象在 R 中的主要用途是 print, summary 和 plot 的方法. 这些方法允许我们仅仅使用一个泛型函数名, 比如 *summary()*, 就能根据一个对象不同的类型来显示它.

```r
x <- rep(0:1, c(10, 20))
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   1.000   0.667   1.000   1.000
```

```r
y <- as.factor(x)
summary(y)
```

```
##  0  1
## 10 20
```

对象的类

S3 对象往往是一个 *list* 并有一个名为 *class* 的属性, 下面的例子创建了一个 *foo* 类的对象并用 *class()* 查看它的类.

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

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

```
class(x)
```

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

```
x <- structure(1, class = "foo")
x
```

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

```
class(x)
```

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

S3 没有正式的类型间关系的定义. 一个对象可以有多个类型, 表现为其 *class* 属性是一个向量.

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

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

方法分派 (method dispatch)

方法分派是指由泛型函数 (generic function) 来决定对某个对象使用的方法. 所有泛型函数都有类似的形式: 一个广义的函数名, 并调用 *UseMethod()* 来决定为对象分派哪个方法. 这也使得泛型函数的形式都很简单, 比如 *mean()*

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x102f0eab0>
## <environment: namespace:base>
```

方法的命名

UseMethod() 会根据对象的 *class* 属性来决定分派什么方法, 所以方法必须以 *generic.class* 的方式命名才能被 *UseMethod()* 找到, 比如:

```
mean.numeric <- function(x, ...) sum(x)/length(x)
mean.data.frame <- function(x, ...) sapply(x, mean, ...)
mean.matrix <- function(x, ...) apply(x, 2, mean)
```

泛型函数示例

如果 *class* 属性是一个向量 *c("foo", "bar")*, 则优先寻找 *mean.foo*, 然后 *mean.bar*, 最后 *mean.default*.

```
bar <- function(x) UseMethod("bar", x)
bar.default <- function(x) "default"
bar.y <- function(x) "y"
bar.z <- function(x) "z"
foo <- structure(1, class = "nonsense")
bar(foo)
```

```
## [1] "default"
```

```
foo <- structure(1, class = c("y", "z"))
bar(foo)
```

```
## [1] "y"
```

继承 (Inheritance)

由于 *class* 属性可以是向量, 所以 S3 中的继承关系自然地表现为 *class* 属性的前一个分量是后一个的子类.*NextMethod()* 函数可以使得一系列的方法被依次应用于对象上.

```
bar.son <- function(x) c("I am son.", NextMethod())
bar.father <- function(x) c("I am father.")
foo <- structure(1, class = c("son", "father"))
bar(foo)
```

```
## [1] "I am son."    "I am father."
```

S3 对象的缺点

S3 对象系统与其称作面向对象系统不如说是一系列命名规则.*class* 属性,*generic.class* 等的命名方式决定了其行为. 所以使用不当时会引起一些问题. 比如方法可以被直接调用而绕过泛型函数的检查.

```
foo <- structure(1, class = "nonsense")
bar(foo)
```

```
## [1] "default"
```

```
bar.z(foo)
```

```
## [1] "z"
```

class 属性可以被直接修改而不管其是不是正确. 同时一个不被叫做 *class* 但表示对象类型的属性不会被 *class()* 函数发现.

```
x <- 1
attr(x, "my_cool_class") <- "foo"
x
```

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

```
class(x)
```

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

class 属性有多个值时, 依次检查每个值是否有合适的方法并分派. 所以 *class* 属性的顺序决定了被分派到的方法.

```
class(foo) <- c("y", "z")
bar(foo)
```

```
## [1] "y"
```

```
class(foo) <- c("z", "y")
bar(foo)
```

```
## [1] "z"
```

有效利用 S3 对象

如果想要计算一个回归模型系数的估计量$\hat{\beta}_0, \hat{\beta}_1$ 的协方差, 可以根据公式

$$cov(\hat{\beta}_0, \hat{\beta}_1) = -\frac{\bar{x}}{\sum(x_i - \bar{x})^2}\sigma^2$$

另外, 各个估计量的方差也可以根据公式计算

$$Var(\hat{\beta}_0) = (1/n + \frac{\bar{x}^2}{\sum(x_i - \bar{x})^2})\sigma^2$$

$$Var(\hat{\beta}_1) = \frac{\sigma^2}{\sum(x_i - \bar{x})^2}$$

```
fit <- lm(dist ~ speed, data = cars)
sigma2 <- sd(fit$residuals)^2
cov <- -mean(cars$speed)/sum((cars$speed - mean(cars$speed))^2) *
    sigma2
var_beta0 <- (1/nrow(cars) + mean(cars$speed)^2/sum((cars$speed -
    mean(cars$speed))^2)) * sigma2
var_beta1 <- sigma2/sum((cars$speed - mean(cars$speed))^2)
```

	1	2
1	44.74	-2.60
2	-2.60	0.17

- 但是回归模型残差方差的估计并不同于一般样本方差的估计, 其自由度与模型的形式和参数有关. 直接计算其标准差而忽略自由度的不同会使得结果出现误差. 但是我们可以利用 *summary()* 对求得的 lm 对象做进一步分析, 然后提取 *summary()* 返回的对象中的方差.

```
sigma2 <- summary(fit)$sigma^2
```

- 但是回归模型残差方差的估计并不同于一般样本方差的估计, 其自由度与模型的形式和参数有关. 直接计算其标准差而忽略自由度的不同会使得结果出现误差. 但是我们可以利用 *summary()* 对求得的 lm 对象做进一步分析, 然后提取 *summary()* 返回的对象中的方差.

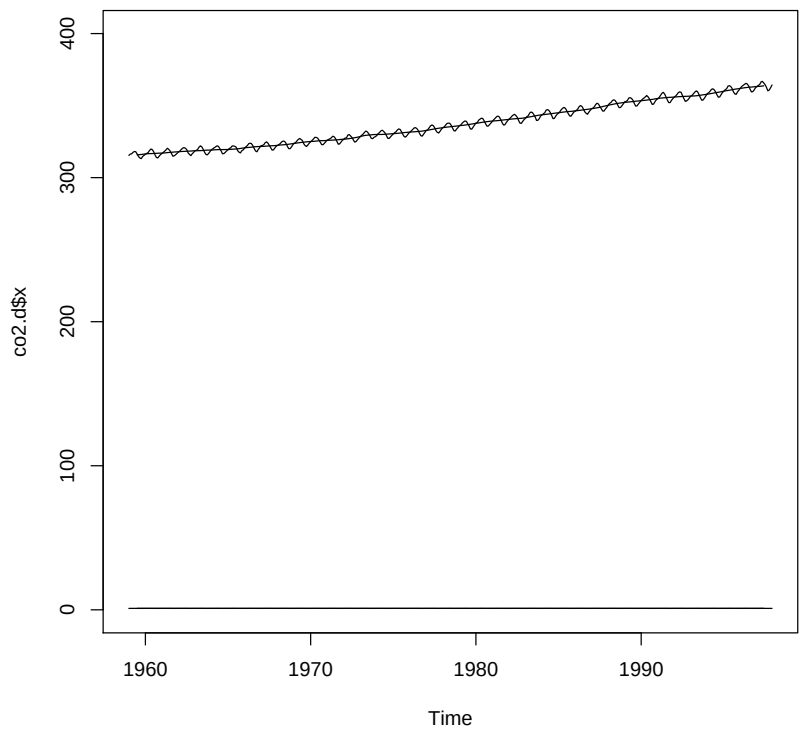
```
sigma2 <- summary(fit)$sigma^2
```

- 或者用 *vcov()* 直接处理拟合后返回的 lm 对象.

```
vcov(fit)
```

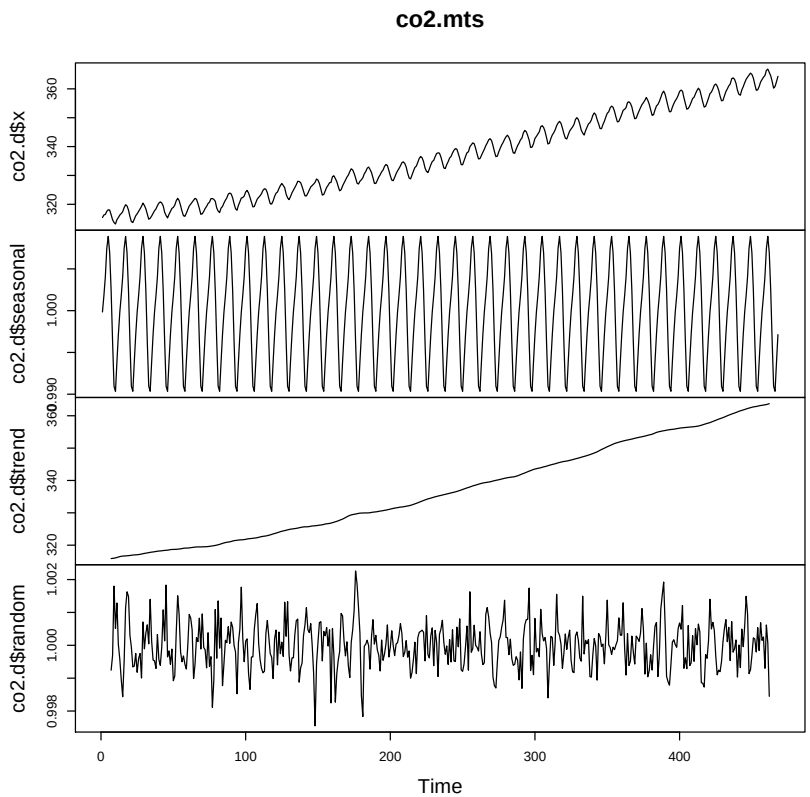
```
##             (Intercept)   speed
## (Intercept)      45.677 -2.6588
## speed            -2.659  0.1727
```

```
co2.d <- decompose(co2, type = "multiplicative")
plot(co2.d$x, ylim = c(0, 400), type = "l")
lines(co2.d$seasonal)
lines(co2.d$trend)
lines(co2.d$random)
```



decompose() 函数提供了时间序列类数据 (ts) 的分解方法. 我们首先将分解后的序列取出并分别绘图.

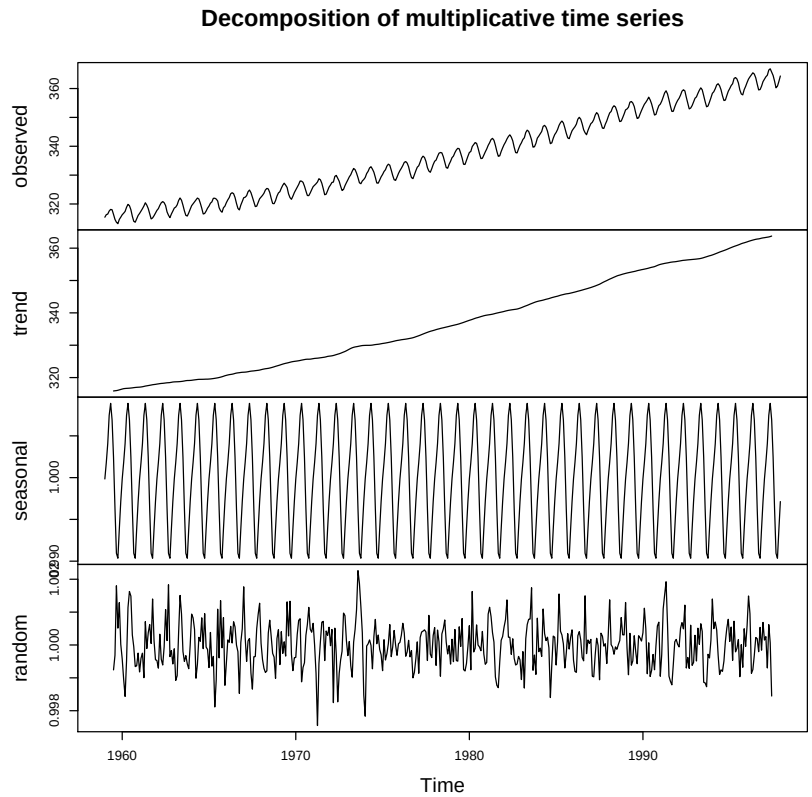
```
m <- cbind(co2.d$x, co2.d$seasonal, co2.d$trend, co2.d$random)
co2.mts <- ts(m)
plot(co2.mts)
```



ts() 函数可以用数据生成一元或多元的时间序列. 将分解后的序列组织成矩阵后, 生成一个多元时间序列, 并调用 *plot()* 泛型函数绘制多元时间序列图.

或者也可以利用对 *decomposed.ts* 类的对象已有的方法 *plot.decomposed.ts()*.

```
plot(co2.d)
```



S3 对象使用小结

- *methods()* 函数可以用来查找 S3 类和方法. 如 *methods(generic.function=predict)* 或 *methods(class=lm)*.

S3 对象使用小结

- *methods()* 函数可以用来查找 S3 类和方法. 如 *methods(generic.function=predict)* 或 *methods(class=lm)*.
- *getS3method(f, class)* 显示泛型函数对特定对象方法的实现.

S3 对象使用小结

- *methods()* 函数可以用来查找 S3 类和方法. 如 *methods(generic.function=predict)* 或 *methods(class=lm)*.
- *getS3method(f, class)* 显示泛型函数对特定对象方法的实现.
- 对于不可见的函数, 可以试试 *getAnywhere()*.

面向对象简介

使用已有的对象：S3 对象系统

创建新的对象：S4 及其他对象系统

S3 和 S4 对比示例

定义 S4 类型

我们可以使用 *setClass()* 函数来定义新的 S4 类型. 如新建一个类来表示本次会议的所有参与者.*representation* 参数用于定义类的属性 (slot) 及其类型.

```
setClass(Class = "Person", representation(name = "character",
    age = "numeric"))
```

继承关系

S4 有比 S3 更为严格的继承关系, 用 *contains* 参数表示. 比如新建一个类表示会议的演讲者, 则演讲者类是参与者类的子类. 子类自动继承父类所有的属性, 并可以定义新的属性.

```
setClass(Class = "Reporter", representation(title = "character"),
    contains = "Person")
```

新建某个类的对象

用 *new()* 函数新建某个类的对象. 此时 S4 会检查每个属性初值的类型是否符合定义类时所给的类型, 如果不符则不能创建对象.

```
yuchen <- new("Reporter", name = "yuchen", age = 22, title = 22)
```

```
## Error: invalid class "Reporter" object: invalid object for
slot "title" in class "Reporter": got class "numeric", should be
or extend class "character"
```

```
yuchen <- new("Reporter", name = "yuchen", age = 22, title = "R and OOP")
```

访问对象的属性

在 S3 中我们通常使用 $ 来访问一个对象的属性, 但是在 S4 对象中我们使用@.

```
yuchen@name
```

```
## [1] "yuchen"
```

或者当你已知属性名的时候, 可以使用 *slot()* 来查看.

```
slot(yuchen, "age")
```

```
## [1] 22
```

设置属性的默认值

当我们不为某个属性赋值时, 其默认值是属性默认类型的默认值. 比如对于类型为 *numeric* 的属性, 其默认值为 *numeric(0)*. 这个默认值显然没有意义, 这时可以用 *prototype* 参数设置其默认值.

```
setClass("Person", representation(name = "character", age = "numeric"),
    prototype(name = NA_character_, age = NA_real_))
new("Person", name = "yuchen")@age
```

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

检验属性的合法性

你可以自定义属性的合法性检查函数, 并通过 *validity* 参数定义给类型.

```
CheckAge <- function(object) {
    if (object@age <= 0) {
        stop("Age is negative.")
    }
}
setClass("Person", representation(name = "character", age = "numeric"),
    validity = CheckAge)
new("Person", age = -5)

## Error: Age is negative.
```

编写泛型函数

编写泛型函数的方式和 S3 类似, 但是使用 *setGeneric()* 函数. 该函数的第二个参数是一个定义了所有需要的参数的函数, 且必须调用 *standardGeneric()* 函数.

```
setGeneric("prepare", function(object) {
    standardGeneric("prepare")
})
```

编写方法

方法的定义使用 *setMethod()* 函数, 并用 *signature()* 函数定义其所面向的类型.

```
setMethod("prepare", signature(object = "Person"), function(object) {
    cat("Got Materials.\n")
})
```

```
prepare(new("Person"))
```

```
## Got Materials.
```

继承

使用 *callNextMethod()* 来为拥有继承关系的对象寻找合适的方法.

```
setMethod("prepare", signature(object = "Reporter"), function(object) {
    callNextMethod()
    cat("Slides are ready.\n")
})


prepare(new("Reporter"))

## Got Materials.
## Slides are ready.
```

S4 对象小结

- *is()* 函数可以用来查看对象的类型, 而 *getSlots()* 可以查看类所有属性的定义.

S4 对象小结

- *is()* 函数可以用来查看对象的类型, 而 *getSlots()* 可以查看类所有属性的定义.
- *showMethods()* 用来查看泛型函数已经定义的方法.

S4 对象小结

- *is()* 函数可以用来查看对象的类型, 而 *getSlots()* 可以查看类所有属性的定义.
- *showMethods()* 用来查看泛型函数已经定义的方法.
- *Bioconductor* 和 *Matrix* 包都基于 S4 对象且遵循良好的编程方式, 可以作为进一步研究的材料.

其他对象系统

- Reference Class 也叫 R5, 是 2.12 版本后新出现的对象系统,R5 完全用 R 实现, 其机制类似 Java 和 C#. 输入*?ReferenceClasses* 查看其帮助.

其他对象系统

- Reference Class 也叫 R5, 是 2.12 版本后新出现的对象系统,R5 完全用 R 实现, 其机制类似 Java 和 C#. 输入*?ReferenceClasses* 查看其帮助.
- 第三方包提供的对象系统: *R.oo*, *proto* 和 mutatr.

面向对象简介

使用已有的对象：S3 对象系统

创建新的对象：S4 及其他对象系统

S3 和 S4 对比示例

定义类型

```
# S3
a1 <- a2 <- a3 <- a4 <- a5 <- 0
class(a1) <- "Instrument"
class(a2) <- c("Stringed", "Instrument")
class(a3) <- c("Wind", "Instrument")
class(a4) <- c("Brass", "Wind", "Instrument")
class(a5) <- c("Woodwind")
# S4
setClass("Instrument", representation("VIRTUAL", tune = "character"))
setClass("Stringed", representation("Instrument"))
setClass("Wind", representation("Instrument"))
setClass("Brass", contains = "Wind")
setClass("Woodwind", representation(tune = "character"))
```

定义泛型函数

```
# S3
play3 <- function(x, ...) UseMethod("play3")
play3.Instrument <- function(x) print("I am a Instrument")
play3.Stringed <- function(x) print("I am a Stringed")
play3.default <- function(x) print("I don't know who I am")

# S4
setGeneric("play4", function(object, ...) standardGeneric("play4"))
setMethod("play4", "Instrument", function(object) print(paste("Play:",
    object@tune)))
```

```
play3(a1)
```

```
## [1] "I am a Instrument"
```

```
play3(a2)
```

```
## [1] "I am a Stringed"
```

```
play3(a3)
```

```
## [1] "I am a Instrument"
```

```
play3(a4)
```

```
## [1] "I am a Instrument"
```

```
play3(a5)
```

```
## [1] "I don't know who I am"
```

```
play4(new("Stringed", tune = "I am a Stringed"))
```

```
## [1] "Play: I am a Stringed"
```

```
play4(new("Wind", tune = "I am a Wind"))
```

```
## [1] "Play: I am a Wind"
```

```
play4(new("Brass", tune = "I am a Brass"))
```

```
## [1] "Play: I am a Brass"
```

```
play4(new("Woodwind", tune = "I am a Woodwind"))
```

```
## Error: unable to find an inherited method for function 'play4'
for signature '"Woodwind"'
```