

R 语言编程：基于 tidyverse

第 28 讲（附录） R6 面向对象, 错误与调试

张敬信

2022 年 12 月 6 日

哈尔滨商业大学

附录 A R6 类面向对象编程简单实例

R6 包为 R 提供了一个封装的面向对象编程的实现，相较于 S3，S4 类，R6 是更新的面向对象的类，支持引用语义。

面向对象的类语法都是类似的，只是语法外形不同，都有构造函数、属性（公共和私有）、方法。

以定义一个银行账户的类为例。

```
library(R6)
```

```
BankAccount = R6Class(  
  classname = "BankAccount",  
  public = list(  
    name = NULL,  
    age = NA,  
    initialize = function(name, age, balance) {  
      self$name = name  
      self$age = age  
      private$balance = balance  
    },  
    printInfo = function() {  
      cat(" 姓名: ", self$name, "\n", sep = "")  
      cat(" 年龄: ", self$age, " 岁\n", sep = "")  
    }  
  )  
)
```

```
    cat(" 存款: ", private$balance, " 元\n", sep = "")
    invisible(self)
},
deposit = function(dep = 0) {
    private$balance = private$balance + dep
    invisible(self)
},
withdraw = function(draw) {
    private$balance = private$balance - draw
    invisible(self)
}),
private = list(balance = 0)
)
```

- 用银行账户的类创建对象，并简单使用：

```
account = BankAccount$new(" 张三", age = 40,  
                           balance = 10000)
```

```
account$printInfo()
```

```
#> 姓名：张三
```

```
#> 年龄：40 岁
```

```
#> 存款：10000 元
```

```
account$balance
```

```
#> NULL
```

```
account$age
```

```
#> [1] 40
```

```
account$  
  deposit(5000)$  
  withdraw(7000)$  
  printInfo()  
#> 姓名: 张三  
#> 年龄: 40 岁  
#> 存款: 8000 元
```

- 定义**继承类**：可透支银行账户

```
BanckAccountCharge = R6Class(  
  classname = "BankAccount",  
  inherit = BankAccount,  
  public = list(  
    withdraw = function(draw = 0) {  
      if (private$balance - draw < 0) {  
        draw = draw + 100  
      }  
      super$withdraw(draw = draw)  
    })  
  ))
```

- 用可透支银行账户创建对象，并简单使用：

```
charge_account = BanckAccountCharge$new(" 李四", age = 35,  
                                          balance = 1000)  
charge_account$withdraw(2000)$  
  printInfo()  
#> 姓名：李四  
#> 年龄：35 岁  
#> 存款：-1100 元
```


调试代码和改正代码中的错误可能是费时且令人头疼的问题。下面介绍一些 R 中有效的处理错误和调试代码的便捷工具。

解决报错的一般策略：

首先，安装 R 时建议将报错消息设置为英文，方便在 Bing 或 Google 搜索到答案。然后，解决问题的一般策略如下：

(1) 搜索

- 通过 Bing 或 Google 搜索错误消息；
- 通过 RStudio community 搜索错误关键词；
- 通过 Stackoverflow 搜索错误关键词 +R 标签；

- (2) 重启 R/RStudio、设置启动时从不保存历史数据：依次打开 Tools -> Global Options -> Workspace, 完成如下设置：

Workspace

☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never ▼

- (3) reprex 包创建最小的可重复的实例向别人提问，在创建过程中自己就能解决问题的概率大概有 80%。

B.1 错误调试技术

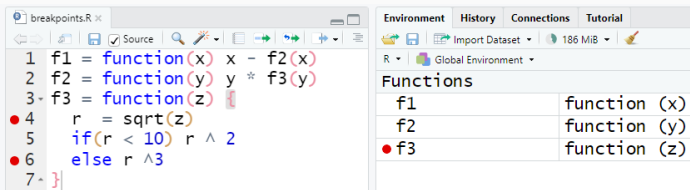
通常，我们不知道错误来自哪一行代码，那么解决问题的一般步骤：

- 开始运行代码
- 在你怀疑有错误或问题的地方停止
- 一步一步地执行代码并查看结果

一种原始的调试方法是在代码中增加 `print()` 或 `cat()` 语句，查看中间结果以便于调试。但更推荐使用 `traceback()` 回溯或 RStudio 提供的交互调试功能。

1. 设置断点

在代码编辑窗口，单击行号左侧则为该行代码设置断点：



注意需要勾选 Source 并单击“保存”才能激活断点。断点的作用是在 R 函数对象中注入一些跟踪代码。包含断点的代码，执行到断点处时将启动调试模式。

2. traceback()

若运行代码出现报错，可以用 `traceback()` 函数来尝试定位错误发生的位置，将打印在错误发生之前的函数调用序列（“调用栈”）。

注意：阅读调用栈是按照从下到上的顺序。

```
f1 = function(x) x - f2(x)
f2 = function(y) y * f3(y)
f3 = function(z) {
  r = sqrt(z)
  if(r < 10) r ^ 2
  else r ^ 3
}
f1(-10)
```

上述代码先定义了 3 个函数，然后执行 `f1(10)` 调用函数，得到如下报错：

```
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In sqrt(z) :
```

```
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE
needed
```



接着执行 `traceback()` 或者直接单击 Show Traceback 按钮，则得到错误回溯：

```
traceback()
```

```
## 3: f3(y) at #1
```

```
## 2: f2(x) at #1
```

```
## 1: f1(-10)
```

错误回溯结果表明，错误发生在 `f3(y)` 的计算过程中。

若想要更详细、可读性更好的报错和错误回溯，可以借助 `rlang` 包：

```
options(error = rlang::entrace)
```

```
f1(-10)
```

```
rlang::last_error()           # 或直接点击该语句
```

```
rlang::last_trace()          # 或直接点击该语句
```

```
<error/rlang_error>
```

```
Error:
```

```
! missing value where TRUE/FALSE needed
```

```
---
```

```
Backtrace:
```

1. `global f1(-10)`
2. `global f2(x)`
3. `global f3(y)`

```
Run rlang::last_trace() to see the full context.
```

```
<error/rlang_error>
```

```
Error:
```

```
! missing value where TRUE/FALSE needed
```

```
---
```

```
Backtrace:
```

1. `global f1(-10)`
2. `global f2(x)`
3. `global f3(y)`

3. browser()

虽然 `traceback()` 确实有用，但它并不能显示一个函数中错误发生的确切位置。为此，需要调试模式。

你可以在代码的任何位置插入 `browser()` 函数，则执行到该步时将从这里开始启动调试模式。

```
f1 = function(x) x - f2(x)
f2 = function(y) y * f3(y)
f3 = function(z) {
  r = sqrt(z)
  browser()                # 插入 browser()
  if(r < 10) r ^ 2
  else r ^3
}
f1(-10)
```


Function: `f3` (GlobalEnv) (Read-only)

Debug location is approximate because the source is not available.

```
1. function(z) {  
2.   r = sqrt(z)  
3.   browser()  
4.   if(r < 10) r ^ 2  
5.   else r ^ 3  
6. }
```

当前函数的源代码
以及正在调试的行

Values

Variable	Value
r	NaN
z	-10

当前函数环境中的
对象

Traceback

```
f3(y)  
f2(x)  
f1(-10)
```

调用栈

Console

```
> f1 = function(x) x - f2(x)  
> f2 = function(y) y * f3(y)  
> f3 = function(z) {  
+   r = sqrt(z)  
+   browser()  
+   if(r < 10) r ^ 2  
+   else r ^ 3  
+ }  
>  
> f1(-10)  
called from: f3(y)  
Warning message:  
In sqrt(z) : NaNs produced  
Browse[1]> r  
[1] NaN  
Browse[1]> z  
[1] -10  
Browse[1]> n  
debug at #4: if (r < 10) r^2 else r^3  
Browse[2]> |
```

辅助调试按钮栏

命令行提示
具体调试窗口

Files

Name	Size	Modified
..		
.._Rhistory	2.4 KB	Aug 29, 2022, 10:42 PM
地址.txt	85 B	Aug 18, 2022, 10:09 PM
debug.Rproj	217 B	Aug 30, 2022, 8:53 AM
my_functions.R	63 B	Aug 29, 2022, 9:36 PM
new_debug.R	1.4 KB	Aug 30, 2022, 2:53 PM

调试模式下控制台的提示符会变成 `Browse[1]>`，此时可以输入想要查看的变量名字，或者执行一些帮助调试的小代码，或者输入一些控制命令（或单击辅助调试按钮）

- `n[Next]`：下一条语句
- `c[Continue]`：继续执行直到下一处断点
- `s[Step into]`：步进函数调用
- `f[Finish]`：完成循环/函数
- `where`：显示之前的调用
- `Q[Stop]`：退出调试器

4. debug()

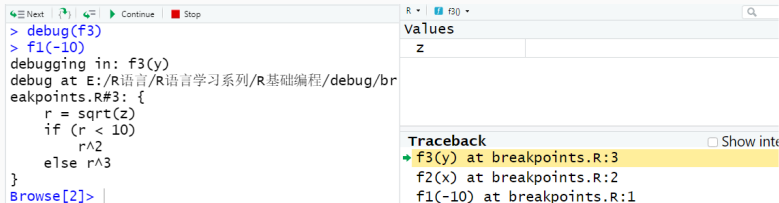
前文方法能够进入函数里面修改代码，适合调试自己编写的函数代码。

对于来自一些包里的函数，不方便修改别人的代码又想进入函数里面去调试，这就需要使用 `debug(函数名)`

```
debug(f3)
```

```
f1(-10)
```

执行上述代码，自动进入函数 `f3` 的调试模式：



The screenshot shows the R Studio interface during a debug session. The console on the left displays the following commands and output:

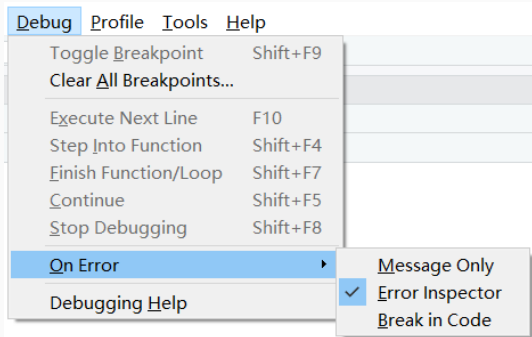
```
> debug(f3)
> f1(-10)
debugging in: f3(y)
debug at E:/R语言/R语言学习系列/R基础编程/debug/breakpoints.R#3: {
  r = sqrt(z)
  if (r < 10)
    r^2
  else r^3
}
Browse[2]> |
```

The right-hand pane is divided into two sections. The top section, titled "Values", shows a table with one column labeled "z". The bottom section, titled "Traceback", shows the call stack with the following entries:

- f3(y) at breakpoints.R:3 (highlighted with a green arrow)
- f2(x) at breakpoints.R:2
- f1(-10) at breakpoints.R:1

运行 `debug()` 之后，需要调用 `undebg()`，否则每次调用该函数时都会进入调试模式。另一种方法是使用 `debugonce()`。

除了上述进入调试模式的方法，还可以设置 RStudio 选项，使得遇到错误后程序自动进入调试模式。依次打开 `Debug->On Error`，并将 “Error Inspector” 改为 “Break in Code”：



另外，有时运行代码会提示警告，若想调试警告可以设置

```
options(warning = 2)
```

这样就能将警告转化为错误，就可以使用上述调试方法了。

B.2 异常处理

在自定义函数特别是开发包中定义函数时，一个好的习惯是让函数尽可能地稳健，或者能够更清晰地将异常信息传达到用户。

异常会导致程序在执行过程中不自然地终止，异常处理也称为条件处理，R 中主要有三类异常：

- 错误 (Errors): 由 `stop()` 或 `rlang::abort()` 引发，强制终止所有执行
- 警告 (Warnings): 由 `warning()` 或 `rlang::warning()` 引发，显示潜在问题
- 消息 (Messages): 由 `message()` 或 `rlang::inform()` 引发，给出信息性输出

其中，错误和警告是需要专门处理的异常，`tryCatch()` 常用来检查 R 代码是否会导致错误或警告信息，以及如何处理它们。

`tryCatch()` 函数通常可设置 4 个参数:

- `expr`: 指定要评估的表达式, 若表达式有多行, 需要用 `{}` 括起来;
- `error`: 接收一个函数, 把捕获的错误信息作为输入, 原样输出或再拼接其他信息作为输出, 即你希望在出现错误时返回的信息;
- `warning`: 接收一个函数, 把捕获的警告信息作为输入, 原样输出或再拼接其他信息作为输出, 即你希望在出现警告时返回的信息;
- `finally`: 最终返回或退出时, 要执行的表达式。

其中, `expr` 是必须指定的, `tryCatch` 首先评估 `expr` 表达式, 若遇到异常 (错误或警告) 则会抛出异常, 此时错误将被 `error` 捕获, 警告将被 `warning` 捕获。

先看一个自定义除法函数的示例：

```
div = function(m, n) {  
  if (!is.numeric(m) | !is.numeric(n)) {  
    stop(" 错误： 分子或分母不是数值！ ")  
  }  
  else if (n == 0) {  
    warning(" 警告： 分母是 0！ ")  
    Inf  
  }  
  else {  
    m / n  
  }  
}
```


注意，直接调用函数，遇到报错就会停止继续向下执行（需要分别执行）：

```
div(3, 2)
```

```
## [0] 0.5
```

```
div("3", 2)
```

```
## Error in div("3", 2) : 错误：分子或分母不是数值！
```

```
div(3, 0)
```

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

```
## Warning message:
```

```
## In div(3, 0) : 警告：分母是 0!
```

改用 tryCatch() 按如下方式包装一下就能全部顺利执行 (不再报错), 并返回你想要的异常 (错误或警告) 信息:

```
tryCatch(div(3, 2),
  error = \(err) err,
  warning = \(warn) cat(paste0(warn, " 小心结果是 Inf! ")))
#> [1] 1.5

tryCatch(div("3", 2),
  error = \(err) err,
  warning = \(warn) cat(paste0(warn, " 小心结果是 Inf! ")))
#> <simpleError in div("3", 2): 错误: 分子或分母不是数值! >

tryCatch(div(3, 0),
  error = \(err) err,
  warning = \(warn) cat(paste0(warn, " 小心结果是 Inf! ")))
#> simpleWarning in div(3, 0): 警告: 分母是 0!
#> 小心结果是 Inf!
```

注意, `\(x)` 是 R 4.1 以来开始支持的 `function(x)` 的简写。

这里的 `error` 实参 (函数) 是将捕获的错误信息原样返回 (输出),
`warning` 实参 (函数) 是对捕获的警告信息, 再拼接上 “小心结果是 Inf!”
”, 并用 `cat()` 输出。可以看到, 若函数正常执行, 相当于 `tryCatch` 机制不起作用。

最后，再看一个利用“`rlang` 包 + `tryCatch()`”处理异常的示例——生成随机数，并计算其对数值。

先定义生成随机数的函数，随机生成一个标准正态分布的随机数，若该随机数不是正数，则用 `rlang::abort()` 抛出一个错误消息，其参数 `class` 用来标记错误的子类：

```
sim_value = function() {  
  val = rnorm(1)  
  if (val <= 0){  
    rlang::abort(message = " 返回值不是正数!",  
                  class = " 模拟值错误", val = val)  
  } else { val }  
}
```

再定义 error 函数, 从捕获的错误信息, 创建想要显示的更具体的错误信息:

```
sim_value_handler = function(err) {  
  msg = " 无法计算值! "  
  if (inherits(err, " 模拟值错误")) {  
    msg = paste0(msg, "`sim_value()`函数生成的数值为",  
                  err$val)  
  }  
  rlang::abort(msg, " 模拟值错误")  
}
```

最后定义计算对数值的函数，对生成随机数的过程使用 `tryCatch` 捕获异常，`error` 采用上面自定义的错误信息处理函数，然后再求对数：

```
log_value = function(){  
  x = tryCatch(sim_value(), error = sim_value_handler)  
  log(x)  
}
```

测试函数设置随机种子保证能够生成负的随机数以触发异常：

```
set.seed(123)
```

```
log_value()
```

```
## Error in `value[[3L]]()`:
```

```
## ! 无法计算值! `sim_value()`函数生成的数值为-0.56047564655
```

```
## Run `rlang::last_error()` to see where the error occurred
```

该例展示了用 rlang 包而不是 base 函数进行异常处理的几个优点，例如能够存储元数据（可自定义处理器做检查），未处理的错误由 abort() 自动保存，可以给用户输出更为详细的信息等。

最后介绍 purrr 循环迭代中的错误或异常处理。purrr 包的 `map_*()`、`walk_*()` 等函数在循环迭代时，若在中间的某步迭代出现错误或异常会直接导致整个循环迭代的失败。因此，有必要提前做好处理预案。

`purrr` 包提供了 3 个“副词”函数，它们会对函数做包装以处理报错或异常，能避免循环迭代的异常中断，并生成增强型的输出：

- `safely(.f, otherwise, quiet)`：包装后的函数反而会返回一个包含组件 `result` 和 `error` 的列表，若发生错误，则 `error` 是错误对象，`result` 使用参数 `otherwise` 提供的值；否则，`error` 为 `NULL`；
- `quietly(.f)`：包装后的函数会返回一个包含结果、输出、消息和警告的列表；
- `possibly(.f, otherwise, quiet)`：包装后的函数在发生错误时使用参数 `otherwise` 提供的值。

其中，`.f` 为要包装的函数，支持 `purrr` 风格的匿名函数写法；`otherwise` 为遇到错误或异常时，用以赋值结果的代替值；`quiet` 为是否隐藏错误（默认 `TRUE`，即不输出）。

这些“副词”函数的结果不便于直接使用，可以配合 `transpose()` 和 `simplify_all()` 一起使用，以使结果更加整洁：

```
safe_log = safely(log, otherwise = NA_real_)
list("a", 10, 100) %>%
  map(safe_log) %>%
  transpose() %>%
  simplify_all()
```

```
#> $result
#> [1]    NA 2.30 4.61
#>
#> $error
#> $error[[1]]
#> <simpleError in .Primitive("log")(x, base): non-numeric
#>
#> $error[[2]]
#> NULL
#>
#> $error[[3]]
#> NULL
```

本篇主要参阅 (张敬信, 2022), R6 包文档, 以及博客文章, 模板感谢 (黄湘云, 2021), (谢益辉, 2021).

参考文献

张敬信 (2022). *R 语言编程：基于 tidyverse*. 人民邮电出版社, 北京.

谢益辉 (2021). *rmarkdown: Dynamic Documents for R*.

黄湘云 (2021). *Github: R-Markdown-Template*.