R for bioinformatics, data iteration & parallel computing

HUST Bioinformatics course series

Wei-Hua Chen (CC BY-NC 4.0)

23 October, 2023

section 1: TOC

前情提要

`stringr`, `stringi` and other string packages ...

1. basics

   - length
   - uppercase, lowercase
   - unite, separate
   - string comparisons, sub string

2. regular expression

本次提要

- for loop
- `apply` functions
- `dplyr` 的本质是遍历
- `map` functions in `purrr` package
- 遍历与并行计算

section 2: iteration basics

for loop , get data ready

```
library(tidyverse);
## create a tibble
df <- tibble( a = rnorm(100), b = rnorm(100), c = rnorm(100), d = rnorm(100) );

head(df, n = 3);
```

```
## # A tibble: 3 x 4
##         a      b       c       d
##     <dbl>  <dbl>   <dbl>   <dbl>
## 1 -0.867   1.68   0.0919  0.0872
## 2  0.470  -1.23  -0.603   1.29
## 3 -0.514  -0.878  0.402  -0.454
```

see for loop in action

```r
## 计算 row means
res1 <- vector( "double", nrow(df) );
for(  row_idx in 1:nrow( df ) ){
  res1[row_idx] <- mean( as.numeric( df[row_idx , ] ) );
}

res2 <- c();
for(  row_idx in 1:nrow( df ) ){
  res2[ length(res2) + 1 ] <- mean( as.numeric( df[row_idx , ] ) );
}

## 计算 column means
res2 <- vector( "double", ncol(df) );
for(  col_idx in 1:ncol( df ) ){
  res2[col_idx] <- mean( df[[col_idx]] );
}
```

for loop 的替代

由于运行效率可能比较低，尽量使用 for loop 的替代

```
rowMeans( df );
```

```
##   [1]  0.247020994 -0.020149461 -0.360947281  0.186008655  0.638557814
##   [6]  0.353293815  0.252250750 -1.439965853 -0.395040085  0.508427240
##  [11] -0.282685516 -0.059651303  0.445753331  0.170417179 -0.333273349
##  [16] -0.053997903  0.621367990  0.319304019 -0.129807504 -0.895528650
##  [21] -0.136260591  0.341992837  0.338033581 -0.393740268  0.167036193
##  [26] -0.831137496 -0.196456889 -0.852351646  0.415013957  0.580096836
##  [31] -0.324702783  1.241690171  0.027231287 -0.281009365  0.612671819
##  [36] -0.199128250 -0.501094948  0.486972679  0.307169826  0.277763028
##  [41]  0.053930322  0.581247524  0.012662938 -0.200267409  0.008958001
##  [46] -0.934607057  0.115323686  0.270584948 -0.613588317  0.719515606
##  [51]  0.086364352  0.346572370  0.552661260 -0.131332625  0.485710815
##  [56] -1.276178577  0.582390433 -0.670922496 -0.179164357 -0.305117110
##  [61]  0.155015154  0.056298132 -0.080184294  0.291757792 -0.924383762
##  [66] -0.289377720  0.093088577  0.233315378  0.380758407  0.134659138
##  [71] -0.365106247  0.660210749  0.636461143 -0.364395222 -0.151247391
##  [76]  0.280192648 -0.405830349 -0.444190745  0.764578528  0.410676940
##  [81]  0.159043715  0.638166429  0.347593752 -0.025144437  0.150655608
##  [86]  0.316654812  0.256029839 -0.101602882  0.190154054  0.627069585
##  [91]  0.778344524  0.138236442  0.049230387  0.754981913 -0.286040085
##  [96] -0.413424211  0.220878909 -0.197827803 -0.538519835 -0.025085757
```

```
colMeans( df );
```

apply 相关函数

Usage:

apply(X, MARGIN, FUN, ...);

MARGIN：1 = 行，2 = 列；c(1,2) = 行 & 列

FUN：函数，可以是系统自带，也可以自己写

```
df %>% apply( ., 1, median ); ## 取行的 median
```

```
##   [1]  0.089535893 -0.066629641 -0.484144970  0.281578753  0.278278463
##   [6]  0.196299105  0.351913534 -1.512594322 -0.048824552  0.362270854
##  [11] -0.072758475 -0.106855828  0.216874541  0.464738163 -0.460509385
##  [16]  0.148839178  0.504382336  0.579397241 -0.144633255 -0.700896607
##  [21] -0.162615694  0.581573223  0.366864323 -0.744630691  0.148883032
##  [26] -0.761328212 -0.216673749 -0.920260680  0.301860440  0.437382660
##  [31] -0.379503319  1.284617928  0.601350970 -0.305187357  0.594016744
##  [36] -0.277972334 -0.603398448  0.149082073  0.444483853  0.108814642
##  [41]  0.204770361  0.485248929 -0.141700346 -0.349087322  0.086473399
##  [46] -1.191693754  0.440336447  0.544448117 -0.030824663  0.763526685
##  [51]  0.069011932  0.460979797  0.682626170 -0.401226671  0.506668287
##  [56] -1.170481325  0.424009612 -0.557588170 -0.047640967 -0.788000848
##  [61] -0.088925386  0.086639342 -0.370042999  0.157290868 -0.866884297
##  [66] -0.208790329  0.171064552  0.409067289  0.496458983  0.102829706
##  [71] -0.171852480  0.765658443  0.650467385 -0.554935716 -0.262533422
##  [76]  0.319868364 -0.553199875 -0.405694300  0.615130193  0.338501610
##  [81] -0.088463584  0.754805731 -0.274548162 -0.238234607  0.212191745
##  [86] -0.007310496  0.053102366 -0.100002458  0.284304394  0.577886824
```

apply 与自定义函数配合

```
df %>% apply( ., 2, function(x) {
    return( c( n = length(x), mean = mean(x), median = median(x) ) );
  } ); ## 列的一些统计结果
```

```
##                    a           b            c           d
## n        100.00000000 100.0000000 100.00000000 100.00000000
## mean       0.03629555   0.1243794  -0.03928564   0.05731394
## median     0.08050399   0.1133232  -0.06403283   0.05596479
```

注意行操作大部分可以被 dplyr 代替

tapply 的使用

以行为基础的操作，用法：

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

用 index 将 x 分组后，用 fun 进行计算 -> 用 姓名将 成绩分组后，计算平均值用 汽缸数将 油耗分组后，计算 平均值

```
library(magrittr);
## 注意 pipe 操作符的使用
mtcars %$% tapply( mpg, cyl, mean ); ## 汽缸数 与 每加仑汽油行驶里程 的关系
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

tapply versus dplyr

然而，使用 dplyr 思路会更清晰

```
mtcars %>% group_by( cyl ) %>% summarise( mean = mean( mpg ) );
```

注意 tapply 和 dplyr 都是基于行的操作!!

lapply 和 sapply

基于列的操作

输入：

- vector ： 每次取一个 element
- data.frame, tibble, matrix ： 每次取一列
- list ： 每次取一个成员

lapply 和 sapply , cont.

输入是 tibble

```
df %>% lapply( mean );
```

```
## $a
## [1] 0.03629555
##
## $b
## [1] 0.1243794
##
## $c
## [1] -0.03928564
##
## $d
## [1] 0.05731394
```

```
df %>% sapply( mean );
```

```
##           a           b           c           d
##  0.03629555  0.12437938 -0.03928564  0.05731394
```

lapply 和 sapply，cont.

输入是 list，使用自定义函数

```
list( a = 1:10, b = letters[1:5], c = LETTERS[1:8] ) %>%
  sapply( function(x) { length(x) } );
```

```
##  a  b  c
## 10  5  8
```

强调

- lapply 是针对列的操作
- 输入是 tibble, matrix, data.frame 时，功能与 apply(x, 2, FUN) 类似 ...

section 3: iteration 进阶：the purrr package

map ，RStudio 提供的 lappy 替代

Figure 1: 来自 purrr package

- part of tidyverse

purrr 的基本函数

map(FUN) : 1. 遍历每列（tibble）或 slot （list），2. 运行 FUN 函数，3. 将计算结果返回至 list

对应：lapply

```
df %>% map( summary );
```

```
## $a
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.1750 -0.6352  0.0805  0.0363  0.7679  1.8472
##
## $b
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.7497 -0.7002  0.1133  0.1244  0.8007  2.9749
##
## $c
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.71961 -0.67434 -0.06403 -0.03929  0.60513  2.81747
##
## $d
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.13920 -0.52907  0.05596  0.05731  0.69860  2.46612
```

对应 sapply 的 map_ 函数

- map_lgl() makes a logical vector.
- map_int() makes an integer vector.
- map_dbl() makes a double vector.
- map_chr() makes a character vector.

```
df %>% map_dbl( mean ); ## 注：返回值只能是单个 double 值
```

```
##           a           b           c           d
##  0.03629555  0.12437938 -0.03928564  0.05731394
```

?? 以下代码运行结果会是什么 ???

```
df %>% map_dbl( summary );
df %>% sapply( summary );
```

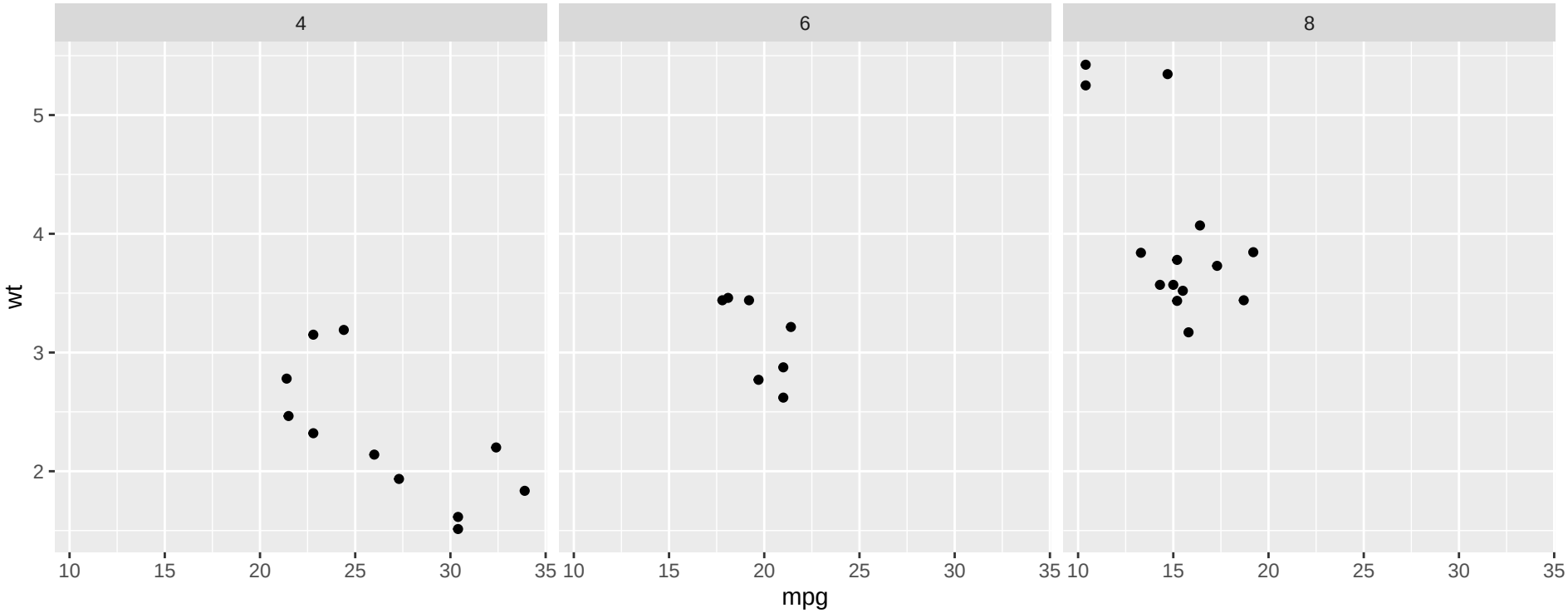
map 的高阶应用

为每一个汽缸分类计算：燃油效率与吨位的关系

```
plt1 <-
  mtcars %>%
  ggplot( aes( mpg, wt ) ) +
  geom_point(  ) + facet_wrap( ~ cyl );
```

取得线性关联关系

```
plt1;
```




```
mtcars %>% split( .$cyl ) %>% map( ~ cor.test( .$wt, .$mpg ) ) %>% map_dbl( ~.$estimate );
```

```
##         4         6         8
## -0.7131848 -0.6815498 -0.6503580
```

命令详解

```
mtcars %>% split( .$cyl ) %>% map( ~ cor.test( .$wt, .$mpg ) ) %>% map_dbl( ~.$estimate );
```

1. split(.$cyl)：由 purrr 提供的函数，将 mtcars 按 cyl 列分为三个 tibble，返回值存入 list

注意：. 在 pipe 中代表从上游传递而来的数据；在某些函数中，比如 cor.test()，必须指定输入数据，可以用 . 代替。

请测试以下代码，查看 split 与 group_by 的区别

```
mtcars %>% split( .$cyl );
mtcars %>% group_by( cyl );
```

命令详解, cont.

```
mtcars %>% split( .$cyl ) %>% map( ~ cor.test( .$wt, .$mpg ) ) %>% map_dbl( ~.$estimate );
```

map：遍历上游传来的数据（list），对每个成分（list 或列）运行函数：~ cor.test(.$wt, .$mpg)

注意

1. 这里的 cor.test 应该有两种写法：

```
## 正规写法:
map( function(df) { cor.test( df$wt, df$mpg ) } )
## 简写:
map( ~ cor.test( .$wt, .$mpg ) )
```

2. ~ 的用法：用于取代 function(df)

命令详解, cont.

map 也可以进行数值提取操作: map_dbl(~.$estimate)

上述命令同样有两种写法:

```
## 完整版
map_dbl( function(eq) { eq$estimate} );
## 简写版
map_dbl( ~.$estimate )
```

more to read & exercise

- `map`: apply a function to **each element** of **a list**, return **a list**
- `map2`: apply a function to **a pair of elements**, return **a list**
- `pmap`: apply a function to groups of elements from **a list of lists or vectors**, return **a list**
- `imap`: ...
- more to read and exercise about iterations: `https://r4ds.had.co.nz/iteration.html`
  - filter
  - index
  - Modify
  - reshape
  - combine
  - reduce
- find more exercise at the end of the slides

reduce

```
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join)
```

```
## Joining with `by = join_by(name)`
## Joining with `by = join_by(name)`
```

```
## # A tibble: 2 x 4
##   name    age sex   treatment
##   <chr> <dbl> <chr> <chr>
## 1 John     30 M     <NA>
## 2 Mary     NA F     A
```

reduce, cont.

```
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
)

vs %>% reduce(intersect)
```

```
## [1]  1  3 10
```

accumulate

```
( x <- sample(10) );
```

```
##  [1]  5  2  4  6  8  9  7 10  1  3
```

```
x %>% accumulate(`+`);
```

```
##  [1]  5  7 11 17 25 34 41 51 52 55
```

section 4: 并行计算

并行计算介绍

并行计算一般需要 3 个步骤：

1. 分解并发放任务
2. 分别计算
3. 回收结果并保存

相关的包

parallel 包：检测 CPU 数量；

doParallel 包：将全部或部分分配给任务

foreach 包：提供 %do% 和 %dopar% 操作符，以提交任务，进行顺序或并行计算

辅助包：

iterators 包：将 data.frame, tibble, matrix 分割为行/列用于提交并行任务。

注意任务完成后，要回收分配的 CPU core。

首先安装相关包（一次完成）。

```
install.packages("doParallel");
install.packages( "foreach" ); ## 会自动安装 iterators
```

简单示例

```
library(doParallel); ##
```

```
## Loading required package: foreach
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
library(foreach);
library(iterators);

## 检测有多少个 CPU --
( cpus <- parallel::detectCores() );
```

```
## [1] 8
```

```
registerDoParallel( cpus - 1 );

## Basic loop
```

简单示例，cont.

%do% loop - foreach notation, but not parallel

```
start <- proc.time()
r <- foreach(icount(trials), .combine=rbind) %do% {
  ind <- sample(100, 100, replace=TRUE)
  result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
  coefficients(result1)
}
do_loop <- proc.time()-start
```

简单示例，cont.

%dopar% adds parallelization

```
start <- proc.time()
r <- foreach(icount(trials), .combine=rbind) %dopar% {
  ind <- sample(100, 100, replace=TRUE)
  result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
  coefficients(result1)
}
dopar_loop <- proc.time()-start
```

简单示例，cont.

结果比较：

```
print(rbind(base_loop,do_loop,dopar_loop)[,1:3])
```

```
##             user.self sys.self elapsed
## base_loop       6.451    0.272   6.738
## do_loop         5.981    0.161   6.142
## dopar_loop      0.366    0.045   1.720
```

命令详解

`.combine = 'c'` 参数的可能值:

- 'c' : 将返回值合并为 vector ; 当返回值是单个数字或字符串的时候使用
- 'cbind' : 将返回值按列合并
- 'rbind' : 将返回值按行合并
- 默认情况下返回 `list`

数据分发练习

将下面的计算转为并行计算

```
mtcars %>% split( .$cyl ) %>% map( ~ cor.test( .$wt, .$mpg ) ) %>% map_dbl( ~.$estimate );
```

```
## make a cluster --
cl2 <- makeCluster( cpus - 1 );
registerDoParallel(cl2);

## 分配任务 ...
res2 <- foreach( df = iter(  mtcars %>% split( .$cyl )  ), .combine = 'rbind' ) %dopar% {
  cor.res <- cor.test( df$wt, df$mpg );
  return ( c( cor = cor.res$estimate, p = cor.res$p.value )  ); ## 注意这里的返回值是
}

res3 <- foreach( df = iter(  mtcars %>% split( .$cyl )  ), .packages = c("ggplot2") ) %dopar% {
  p <- ggplot(df, aes( x = wt, y = mpg )) + geom_point();
  return ( p ); ## 注意这里的返回值是
}

## 注意在最后关闭创建的 cluster
stopCluster( cl2 );

res2;
```

练习详解

1. `df = iter( mtcars %>% split( .$cyl ) )`：mtcars 按汽缸数分割为 3 个 list，依次赋予 df；
2. `cor.res <- cor.test( df$wt, df$mpg );`：计算每个 df 中 wt 与 mpg 的关联，将结果保存在 `cor.res` 变量中；
3. `.combine = 'rbind'`：由于返回值是 vector，用此命令按行合并；

foreach 的其它参数

.packages=NULL：将需要的包传递给任务。如果每个任务需要提前装入某些包，可以此方法。比如：

```
.packages=c("tidyverse")
```

嵌套 (nested) foreach

有些情况下需要用到嵌套循环，使用以下语法：

```
foreach( ... ) %:% {
  foreach( ... ) %dopar% {

  }
}
```

即：外层的循环部分用%:% 操作符

其它并行计算函数

parallel 包本身也提供了 lapply 等函数的并行计算版本，包括：

- parLapply
- parSapply
- parRapply
- parCapply

parLapply 举例

任务：计算 2 的 N 次方：

```
cl<-makeCluster(3);
parLapply(cl,
          2:4,
          function(exponent)
            2^exponent);
stopCluster(cl);
```

其它的函数这里就不一一介绍了

section 5: 小结及作业!

本次小结

iterations 与并行计算

- for loop
- `apply` functions
- `dplyr` 的本质是遍历
- `map` functions in `purrr` package
- 遍历与并行计算

相关包

- purrr
- parallel
- foreach
- iterators

下次预告

data visualizations

- basic plot functions
- basic ggplot2
- special letters
- equations

作业

- `Exercises and homework` 目录下 `talk08-homework.Rmd` 文件;
- 完成时间：见钉群的要求