R for bioinformatics, data iteration & parallel computing

HUST Bioinformatics course series

Wei-Hua Chen (CC BY-NC 4.0)

05 August, 2021

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 -1.18   -2.09  -0.138 -0.0844
## 2 -0.826  -0.151 -0.844  0.145
## 3  1.35   -0.730 -2.05   -0.771
```

see for loop in action

```
## 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 , ] ) );
}

## 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] -0.873035459 -0.418982072 -0.550380533 -0.064820843  0.015886246
##   [6]  0.680985167 -0.539894707 -0.107998568 -0.787421272 -0.128382880
##  [11]  0.871207033 -0.543234882 -1.113125121  0.379533020  0.547306558
##  [16] -0.784066288  0.555569528  0.513703621 -0.122693336 -0.435185802
##  [21] -0.333892354  0.377697294 -0.307493482  0.726564932  0.274047372
##  [26]  0.384318580 -0.404647653  0.252956263 -0.671014161  1.323017629
##  [31] -0.254530427 -0.692574318 -0.920101638 -0.736748651  0.536277965
##  [36] -0.169840870  0.589531702 -0.414052232  0.563622320 -0.397414999
##  [41]  0.004178446 -0.384867911 -0.266331752  0.835490694 -0.590907816
##  [46]  0.126082135  0.435548050  0.378189550  0.193353592 -0.427357560
##  [51]  0.163723282 -0.576241880  0.331744519 -0.075298197  0.160560406
##  [56]  0.561840148  0.942759112 -0.738703123 -0.352537335  0.104740000
##  [61]  0.296770232 -0.894197689 -0.184920318 -0.080449821  0.176735240
##  [66] -0.570605351 -0.258680505  0.647089869  0.917358911  0.879003188
##  [71]  1.145993119 -0.068919585 -0.444509357 -0.057233094 -1.390566767
##  [76] -0.052797172  0.016141793 -0.210057966  0.536214519 -0.274917162
##  [81] -0.275073212 -0.020875603 -1.029712218 -0.114393096  0.487833663
##  [86]  0.389715228 -0.315423927 -0.129295437  0.514293342  0.319728992
##  [91]  0.417082436 -0.551157391  0.226305902 -0.390100392 -0.220885775
##  [96] -0.083694291  0.075556653  0.114807144  1.014896993  0.824098774
```

```
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] -0.65988116 -0.48854148 -0.75046617 -0.04269861  0.42420627  0.90665948
##   [7] -0.47188617  0.32716907 -0.90671940  0.15641561  1.06436647 -0.94320808
##  [13] -1.12807719  0.33137549  0.36938871 -0.74010729  0.79263842  0.45495395
##  [19]  0.04073745 -0.59619295 -0.25276843  0.60064562 -0.74528490  1.35134171
##  [25] -0.02458719  0.76841733 -0.42599456  0.19025918 -0.42499822  1.34140780
##  [31] -0.28877165 -0.97613348 -1.18642319 -0.62374956  0.30585319 -0.04315312
##  [37]  0.72060466 -0.45292849  0.88265986 -0.61500552  0.12272882 -0.39911518
##  [43] -0.15969886  0.55517291 -0.67839611 -0.19053407  0.32052553  0.46303599
##  [49]  0.16189442 -0.04425589  0.12793695 -0.71796038  0.84694629  0.28834850
##  [55]  0.26699240  0.10289292  0.98084808 -0.92807334 -0.37787558  0.07854236
##  [61]  0.51486352 -0.57914390 -0.06314877 -0.09623248  0.01939125 -0.50141319
##  [67] -0.35669474  0.60822256  0.47785338  0.90234008  1.16653782 -0.46042895
##  [73] -0.35672132  0.15441323 -1.61380263 -0.02561484 -0.04785824 -0.35719186
##  [79]  0.28832641 -0.27161410 -0.22498323 -0.31708036 -1.00289676 -0.48272613
##  [85]  0.68785843  0.05505628 -0.40784034 -0.28481204  0.82667388  0.36585740
##  [91]  0.26256447 -0.58627916  0.63138962 -0.96915719 -0.25743517 -0.02724774
##  [97]  0.18702925  0.09943942  1.00809036  1.22178965
```

```
df %>% apply( ., 2, median ); ## 取列的 median
```

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       100.0000000 100.0000000 100.00000000  1.000000e+02
## mean     -0.1326878   0.1274982  -0.03280848 -8.892773e-04
## median   -0.1663087   0.0337408  -0.06725263 -8.493790e-02
```

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] -0.1326878
##
## $b
## [1] 0.1274982
##
## $c
## [1] -0.03280848
##
## $d
## [1] -0.0008892773
```

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

```
##             a             b             c             d
## -0.1326877635  0.1274981994 -0.0328084821 -0.0008892773
```

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.
## -2.9341 -0.7935 -0.1663 -0.1327  0.4997  2.4553
##
## $b
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.98099 -0.70297  0.03374  0.12750  0.80335  2.78176
##
## $c
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.44117 -0.70683 -0.06725 -0.03281  0.67000  2.68315
##
## $d
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -2.9928901 -0.5300190 -0.0849379 -0.0008893  0.7492759  2.2872325
```

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
## -0.1326877635  0.1274981994 -0.0328084821 -0.0008892773
```

?? 以下代码运行结果会是什么 ???

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

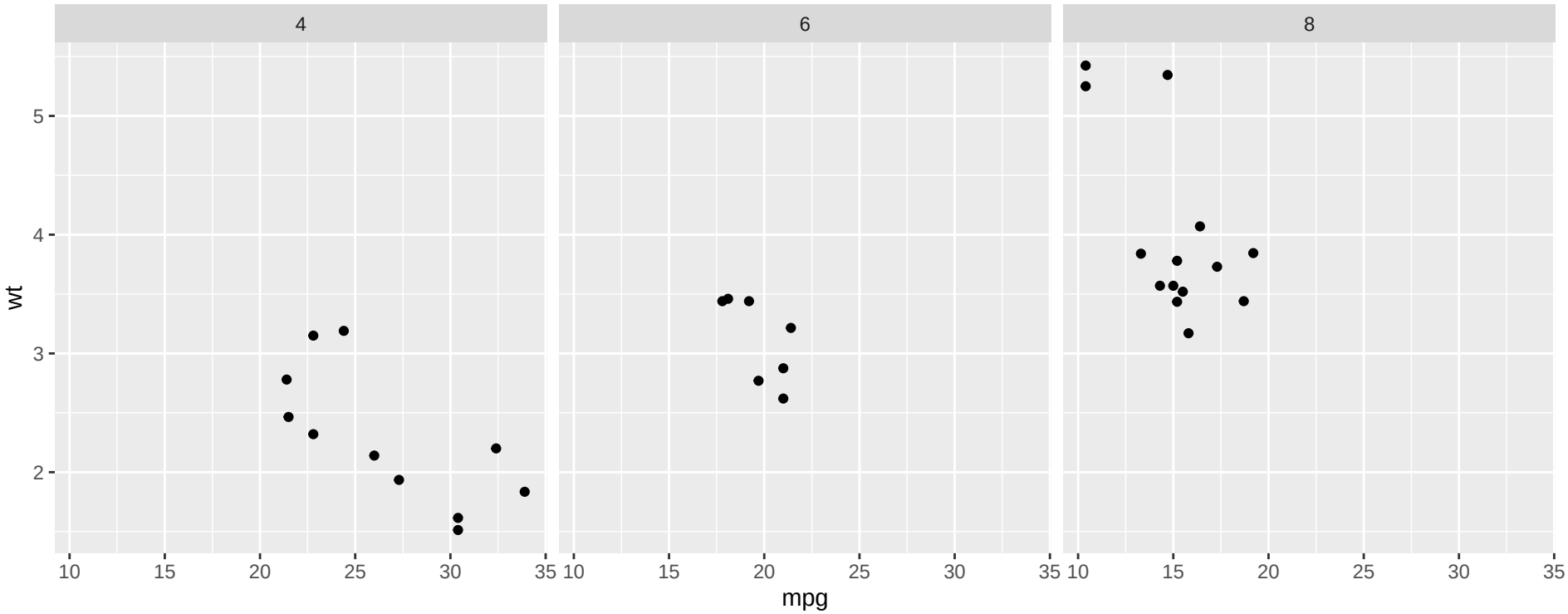
map 的高阶应用

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, by = "name"
## Joining, 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]  9  8  1  6 10  3  5  2  4  7
```

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

```
##  [1]  9 17 18 24 34 37 42 44 48 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：显示 CPU core 数量，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( "parallel" );
install.packages( "foreach" ); ## 会自动安装 iterators
```

简单示例

```
library(parallel); ##
library(foreach);
library(iterators);

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

## 创建一个 data.frame
d <- data.frame(x=1:10000, y=rnorm(10000));

## make a cluster --
cl <- makeCluster( cpus - 1 );

## 分配任务 ...
res <- foreach( row = iter( d, by = "row" ) ) %dopar% {
  return ( row$x * row$y  );
}

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

res;
```

命令详解

1. 获得 CPU 数量
2. 将 CPU - 1 赋给变量 cl

```
( cpus <- parallel::detectCores() );

## 创建一个 data.frame
d <- data.frame(x=1:10, y=rnorm(10));

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

命令详解，cont.

```
res <- foreach( row = iter( d, by = "row" ), .combine = 'c' ) %dopar% {
  return ( row$x * row$y  );
}
```

3. `row = iter( d, by = "row" )`：将输入数据 d (data.frame) 按行 (row) 或列 (col) 遍历，每次取出一行或列，赋予 `row` 这个变量（可随意取名）；
4. `foreach` 将数据 `row` 分发给 `cl` （这里没有体现出来），进行计算 `row$x * row$y`，并返回结果
5. `.combine = 'c'` 参数规定将返回结果合并为 `vector` 。

命令详解，cont.

`.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`

6. `stopCluster( cl );`：释放资源

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 );

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

res2 <- 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