R for bioinformatics, data iteration & parallel computing

HUST Bioinformatics course series

Wei-Hua Chen (CC BY-NC 2.0)

13 August, 2019

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(10), b = rnorm(10), c = rnorm(10), d = rnorm(10) );

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

```
## # A tibble: 3 x 4
##         a       b      c      d
##     <dbl>   <dbl>  <dbl>  <dbl>
## 1 0.0800  1.64    1.30   1.04
## 2 0.270  -0.393  -0.551 -0.169
## 3 1.84   -0.0192 -1.40   0.704
```

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 , ] ) );
}

## 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]  1.0128548 -0.2108880  0.2818320 -1.1128401  0.5197360 -0.8741270
##  [7] -0.3321474  0.5153916  0.2127645 -0.3205796
```

```
colMeans( df );
```

```
##           a           b           c           d
##  0.22892536  0.32702133 -0.66097007 -0.01817792
```

类似的函数还有：

```
rowSums( df );
colSums( 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]  1.16791298 -0.28095776  0.34243673 -0.64486128  0.27517268
##  [6] -0.95231576 -0.25621257  1.07757402 -0.01258584 -0.31915915
```

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

```
##           a           b           c           d
##  0.13957898  0.23198202 -0.61569119  0.07240441
```

问题 df %>% apply(., c(1,2), median); ## 取 both 的 median 会发生什么 ??

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     10.0000000 10.0000000 10.0000000 10.00000000
## mean   0.2289254  0.3270213 -0.6609701 -0.01817792
## median 0.1395790  0.2319820 -0.6156912  0.07240441
```

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] 0.2289254
##
## $b
## [1] 0.3270213
##
## $c
## [1] -0.6609701
##
## $d
## [1] -0.01817792
```

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

```
##           a           b           c           d
##  0.22892536  0.32702133 -0.66097007 -0.01817792
```

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

对应: lapply

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

```
## $a
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.95814  0.01723  0.13958  0.22893  0.26915  1.83878
##
## $b
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.7911 -0.4065  0.2320  0.3270  1.2096  1.9734
##
## $c
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.6911 -1.3172 -0.6157 -0.6610 -0.3311  1.2969
##
## $d
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.73664 -0.31002  0.07240 -0.01818  0.66262  1.22962
```

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
##  0.22892536  0.32702133 -0.66097007 -0.01817792
```

?? 以下代码运行结果会是什么 ???

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

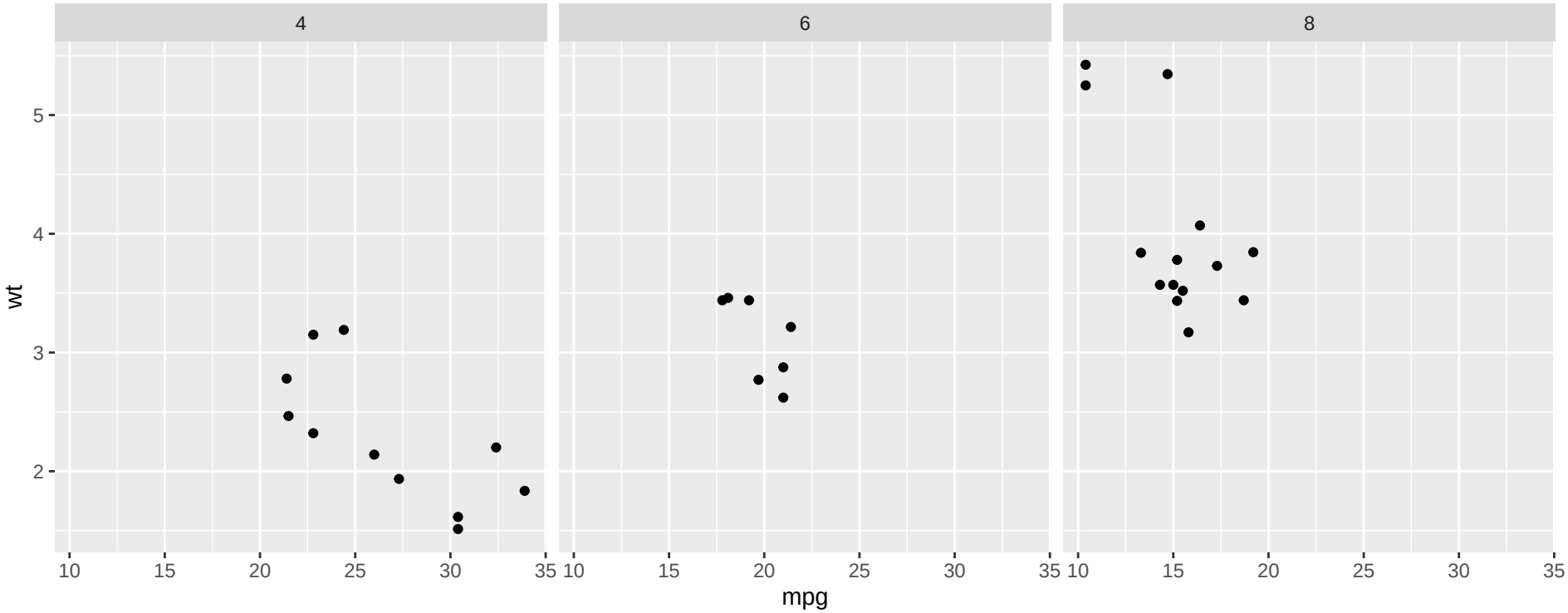
map 的高阶应用

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

- more to read and exercise about iterations:
  `https://r4ds.had.co.nz/iteration.html`
  - walk, walk2 & pwalk：和 map 相似，但只用于运行命令，不返回值
  - keep, discard：用于保留或过滤符合条件的 elements
- purrr 练习 1
- purrr 练习 2

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);
```

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

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

```
library(iterators);

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

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

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

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

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

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
}

## 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);
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 8
##
## [[3]]
## [1] 16
```

```
stopCluster(cl);
```

其它的函数这里就不一一介绍了

more to read & exercise

more to read

- parallel 包的更多示例：https://www.r-bloggers.com/how-to-go-parallel-in-r-basics-tips/

练习

- 练习

section 5: 小结及预告

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 visualisations

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