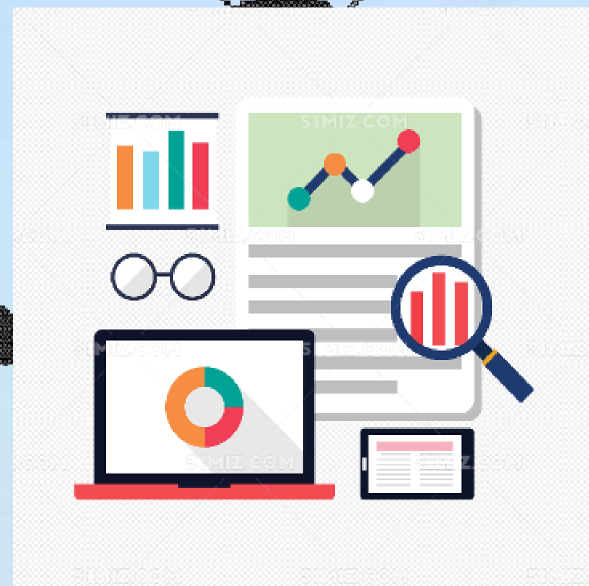
第8章 R语言大数据分析

8.1 R语言效率编程

8.2 内存管理

8.3 R并行处理包

8.4 R高效读取大数据



R语言大数据分析

R语言效率编程

system.time　向量化

C++编程　apply族函数

内存管理

R并行处理包

parallel软件包　foreach软件包

R高效读取大数据

fread()　数据库处理　dplyr

R将所有的对象都存储在虚拟内存中。当分析大数据集时，这种设计会降低程序运行速度，有时还会产生跟内存相关的错误。

“cannot allocate vector of size 或无法分配大小向量”

“cannot allocate vector of length 或无法分配长度向量”

解决方案：

- ✓ 提高程序执行效率
- ✓ 采用并行计算
- ✓ 改变数据结构或把数据储存在外部

8.1 效率编程

检查代码运行时间

用system.time()衡量运行时间，包括user,system和elapsed三个时间。

```
X <- rnorm(100000)
Y <- rnorm(100000)
Z <- rep(NA, 100000)
system.time({
for (i in 1:100000) {
Z[i] <- X[i] + Y[i]
}
})
user system elapsed
0.54 0.00 0.55
```

用户时间。执行用户命令所需CPU时间。

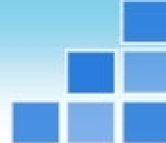
实际运算时间。源程序实际所需的总运算时间，包括操作系统调度的时间。通常是user 与 system 时间和。

系统时间是打开或关闭文件、分配和释放内存、执行系统命令所需CPU时间

向量化运算

向量化运算是R的特点之一。避免使用for这样的显式循环语句，最大化地实现R计算过程的向量化，R的计算速度绝对不容鄙视。

```
X <- rnorm(100000)
Y <- rnorm(100000)
 Z <- c()
for (i in 1:100000) {
    Z <- c(Z, X[i] + Y[i]) # this takes about 54.09 seconds
 }
 Z <- rep(NA, 100000)
 for (i in 1:100000) {
  Z[i] <- X[i] + Y[i] # this takes about 0.54 seconds
   }
Z <- X + Y # 0.002 seconds (approx)
```

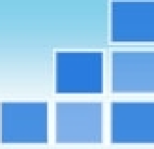


优先使用base包提供的命令

base包的命令是一般经过优化的，速度有保障。

```
 X <- rnorm(10000000)
mean_r = function(x) {
        m = 0
        n = length(x)
        for(i in seq_len(n))
                m = m + x[i] / n
        m
}
mean_r(X) #0.41
mean(X)   #0.02
```

R调用C, C++或Fortran



可以通过Rcpp、RcppEigen、RcppArmadillo 进行R与C++混合编程，用C++分配内存或计算。

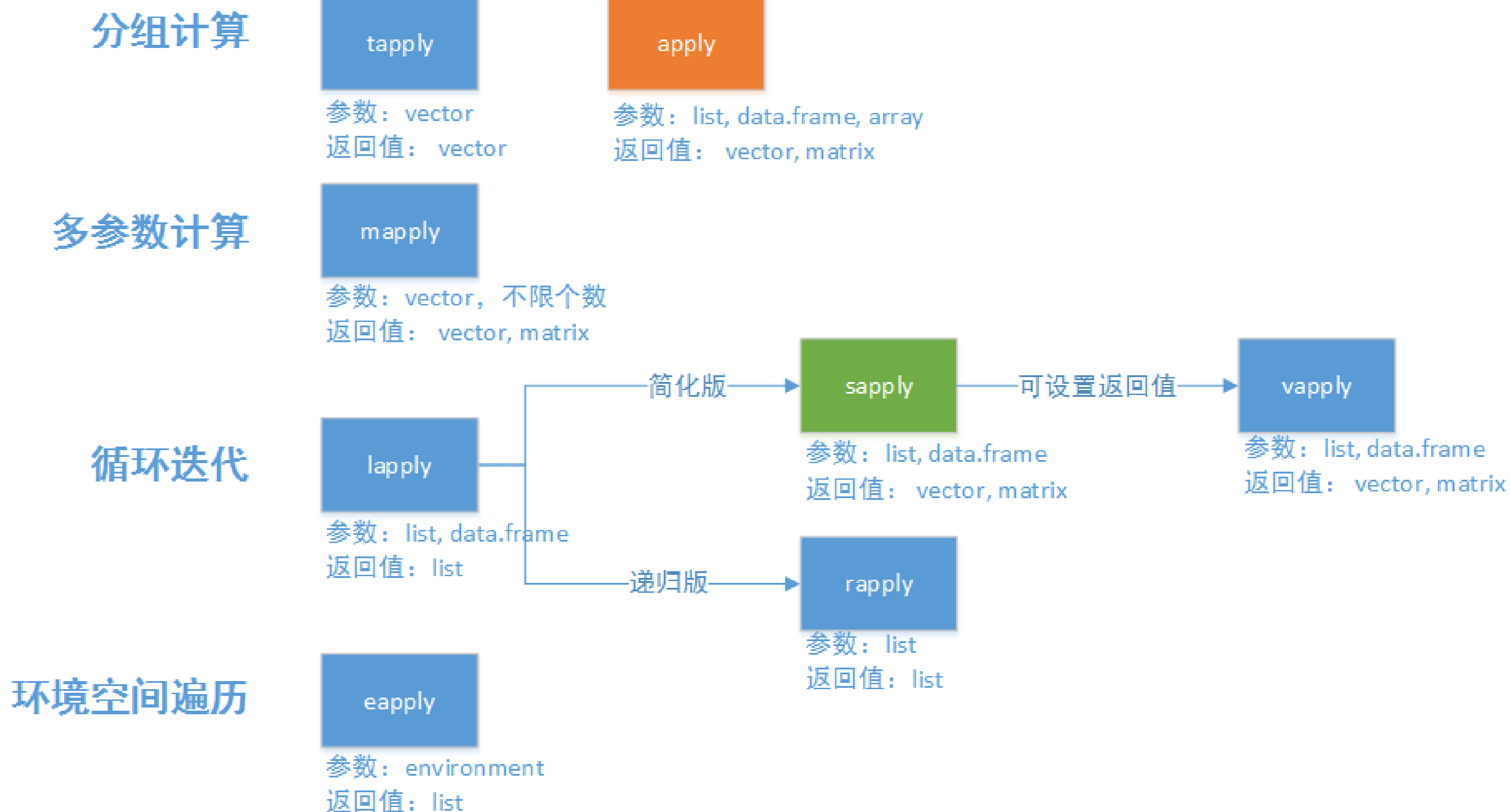
```
Library(Rcpp)
cppFunction( 'int fib_cpp_0(int n){
      if(n==1||n==2) return 1;
      return(fib_cpp_0(n-1)+fib_cpp_0(n-2)); }'
)

fib_r <- function(n){
      if(n==1||n==2)     return(1)
      return(fib_r(n-1)+fib_r(n-2))
}
```

```
> system.time(fib_r(30))
   user  system elapsed
  3.080   0.000   3.083
> system.time(fib_cpp_0(30))
   user  system elapsed
  0.004   0.000   0.004
```

apply族函数

apply族函数： lapply sapply apply tapply mapply。
函数底层是通过C来实现、运算向量化，效率高



apply函数

- 对于向量，我们有sum、mean等函数对其进行计算。对于数组，如果我们想对其中一维（或若干维）进行某种计算，可以用行或列遍历操作函数apply 。其一般形式为：
- apply(X, MARGIN, FUN, ...)

  X: an array, including a matrix.

  MARGIN: 1：行操作； 2：列操作

  FUN:函数名

- 用apply可以很方便地按行列求和/平均，其结果与colMeans,colSums,rowMeans,rowSums是一样的。

```
> a<-matrix(1:12,c(3,4))
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(a,1,sum)
[1] 22 26 30
> apply(a,2,sum)
[1]  6 15 24 33
> apply(a,1,function(x) sum(x)+2)
[1] 24 28 32
```

如果函数FUN的结果是一个标量，MARGIN只有一个元素，则apply的结果是一个向量，其长度等于MARGIN指定维的长度，相当于固定MARGIN指定的那一维的每一个值而把其它维取出作为子数组或向量送入FUN中进行运算。

```
➢a<-matrix(1:12,c(3,4))
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

如果函数FUN的结果是一个长度为N的向量，则结果是一个维数向量等于c(N, dim(X)[MARGIN])的数组，注意这时不论是对哪一维计算，结果都放在了第一维。

```
> apply(a,1,function(x) x^2)
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
[3,]   49   64   81
[4,]  100  121  144
```

```
> apply(a,2,function(x) x^2)
     [,1] [,2] [,3] [,4]
[1,]    1   16   49  100
[2,]    4   25   64  121
[3,]    9   36   81  144
```

8.2 内存管理

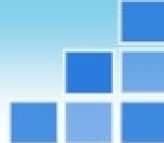
第8章 R语言大数据分析

object.size(): 了解 R 中的内存使用,给出一个R语言对象占用了多少字节的内存.

```
y <- rnorm(10000)
object.size(y)
object.size("R Hello world!")
```

```
# 80048 bytes

# 120 bytes
```



在windows平台下，memory.size()则报告内存中所有对象的总大小(MB 为单位).

```
memory.size()
# [1] 473.4
```

使用函数memory.limit()获取或设置系统上的内存限制

```
memory.limit()
memory.limit(size = 1080)
```

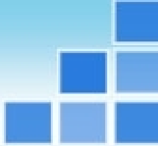
内存清理

使用 gc() 函数强制垃圾收集器运行，释放内存

```
gc()
```

使用rm()函数从环境中删除一个对象

```
# 主动删除一个较大的R内存对象
rm(alargedat)
```

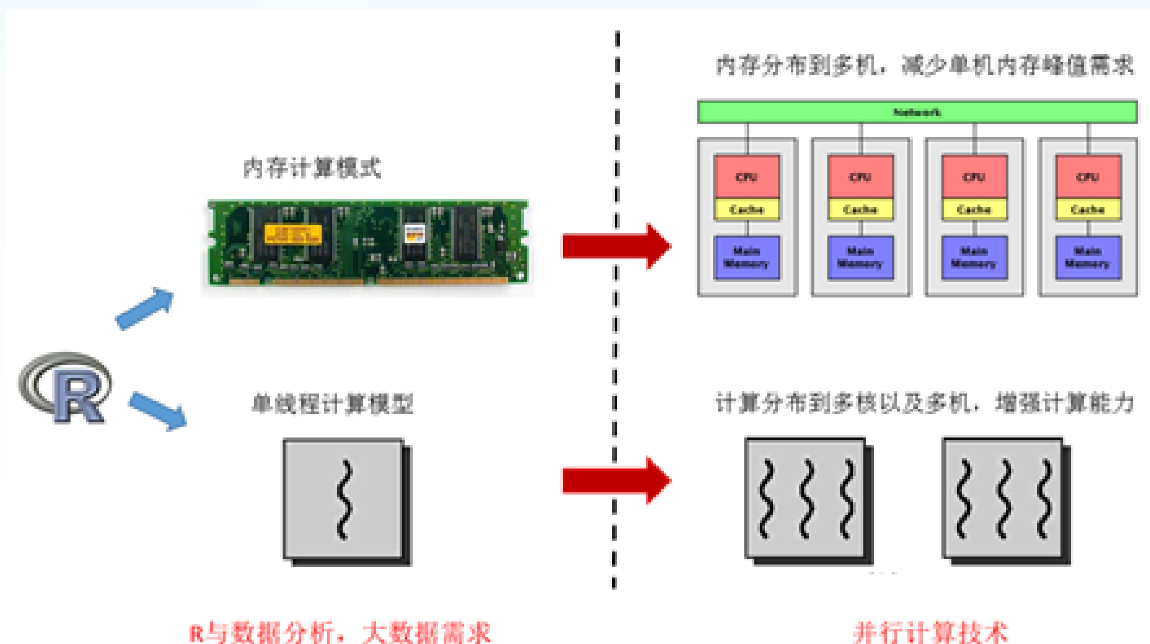


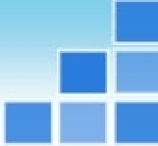
8.3 R并行计算

- 随着数据量的日渐增大，R 的内存使用方式和计算模式限制了 R 处理大规模数据的能力。
- 从内存角度来看，R 采用的是内存计算模式（In-Memory），被处理的数据需要预取到主存（RAM）中。其优点是计算效率高、速度快，但缺点是这样一来能处理的问题规模就非常有限（小于 RAM 的大小）。
- R 的核心（R core）是一个单线程的程序。因此，在现代的多核处理器上，R 无法有效地利用所有的计算内核。因此，在现代的多核处理器上，R 无法有效地利用所有的计算内核。

怎么破？并行计算！

- 并行计算技术正是为了在实际应用中解决单机内存容量和单核计算能力无法满足计算需求的问题而提出的。因此，并行计算技术常有力地扩充 R 的使用范围和场景。
- 最新版 R 已经将 parallel 包设为了默认安装包。可见 R 核心开发组也对并行计算非常重视了。





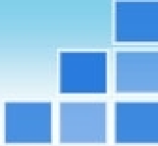
8.3 R并行计算

- 所有的商用计算机都使用并行处理来完成多任务!
- 你的电脑有多少个CPU（核心）?

```
library(parallel)   # Comes with R!
detectCores(logical = F)      # Num. of physical cores
detectCores(logical = T)      # Num. of functional cores
```

超线程CPU允许每个CPU芯片有两个功能CPU

- 您可以可视化计算机上的CPU（和网络）活动:
- Windows: Task Manager
- Mac: Activity Monitor



parallel包简介

- parallel包的思路和lapply函数很相似，都是将输入数据分割、计算、整合结果。只不过并行计算是用到了不同的cpu内核来运算。
- 在parallel包里，对应上述两种并行化方式有如下两个核心函数（针对于lapply函数的并行化，mclapply在windows上不能使用）：
  - parLapply(cl, x, FUN, ...)
  - mclapply(X, FUN, ..., mc.cores)

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/138017012123006143