

# 老少咸宜 Rcpp

Masaki E. Tsuda 著 jywang 译

2019-08-19



# 目录

第一章 啥情况该用 Rcpp	1
第二章 安装	3
2.1 安装 C++ 编译器 . . . . .	3
2.1.1 Windows . . . . .	3
2.1.2 Mac . . . . .	3
2.1.3 Linux . . . . .	3
2.1.4 使用其他编译器 . . . . .	3
2.2 安装 Rcpp . . . . .	4
第三章 基本用法	5
3.1 写 Rcpp 代码 . . . . .	5
3.1.1 在 Rcpp 中定义一个函数的格式 . . . . .	6
3.2 编译代码 . . . . .	6
3.3 执行函数 . . . . .	7
第四章 将 Rcpp 嵌入 R 代码	9
4.1 sourceCpp() . . . . .	9
4.2 cppFunction() . . . . .	10
4.3 evalCpp() . . . . .	10

<b>第五章 C++11</b>	<b>11</b>
5.1 开启 C++11	11
5.2 推荐的 C++11 特性	11
5.2.1 初始化	11
5.2.2 auto	12
5.2.3 decltype	12
5.2.4 基于范围的 for 循环	12
5.2.5 Lambda 表达式	13
<b>第六章 信息打印</b>	<b>15</b>
6.1 Rcout, Rcerr	15
6.2 Rprintf(), Rfprintf()	15
<b>第七章 数据类型</b>	<b>17</b>
7.1 向量和矩阵	17
7.2 data.frame, list, S3, S4	18
<b>第八章 Vector 类</b>	<b>19</b>
8.1 创建向量对象	19
8.2 获取向量元素	20
8.3 成员函数	21
8.3.1 length(), size()	21
8.3.2 names()	22
8.3.3 offset( name ), findName( name )	23
8.3.4 offset( i )	23
8.3.5 fill( x )	24
8.3.6 sort()	25

8.3.7	assign( first_it, last_it ) . . . . .	25
8.3.8	push_back( x ) . . . . .	26
8.3.9	push_back( x, name ) . . . . .	27
8.3.10	push_front( x ) . . . . .	27
8.3.11	push_front( x, name ) . . . . .	28
8.3.12	begin() . . . . .	29
8.3.13	end() . . . . .	29
8.3.14	cbegin() . . . . .	30
8.3.15	cend() . . . . .	30
8.3.16	insert( i, x ) . . . . .	32
8.3.17	insert( it, x ) . . . . .	33
8.3.18	erase(i) . . . . .	33
8.3.19	erase(it) . . . . .	34
8.3.20	erase( first_i, last_i ) . . . . .	35
8.3.21	erase( first_it, last_it ) . . . . .	35
8.3.22	containsElementNamed(name) . . . . .	36
8.4	静态成员函数 . . . . .	37
8.4.1	get_na() . . . . .	37
8.4.2	is_na(x) . . . . .	38
8.4.3	create( x1, x2, ... ) . . . . .	38
8.4.4	import( first_it , last_it ) . . . . .	39
8.4.5	import_transform( first_it, last_it, func ) . . . . .	40
<b>第九章</b>	<b>Matrix 类</b>	<b>43</b>
9.1	创建矩阵对象 . . . . .	43
9.2	访问矩阵元素 . . . . .	44

9.2.1	访问行, 列与子矩阵 . . . . .	45
9.3	成员函数 . . . . .	47
9.3.1	nrow() rows() . . . . .	47
9.3.2	ncol() cols() . . . . .	47
9.3.3	row( i ) . . . . .	48
9.3.4	column( i ) . . . . .	49
9.3.5	fill_diag( x ) . . . . .	49
9.3.6	offset( i, j ) . . . . .	50
9.4	静态成员函数 . . . . .	50
9.4.1	Matrix::diag( size, x ) . . . . .	50
9.5	与 Matrix 相关的其他函数 . . . . .	51
9.5.1	rownames( m ) . . . . .	51
9.5.2	colnames( m ) . . . . .	53
9.5.3	transpose( m ) . . . . .	53
<b>第十章</b>	<b>向量运算</b>	<b>55</b>
10.1	数学运算 . . . . .	55
10.2	比较运算 . . . . .	56
<b>第十一章</b>	<b>逻辑运算符</b>	<b>59</b>
11.1	LogicalVector . . . . .	59
11.1.1	LogicalVector 元素的数据类型 . . . . .	59
11.2	逻辑运算符 . . . . .	59
11.3	接收 LogicalVector 的函数 . . . . .	60
11.3.1	all(), any() . . . . .	60
11.3.2	ifelse() . . . . .	62
11.4	LogicalVector 元素的估值 . . . . .	64

# 表格





# 插图



# 欢迎

Rcpp 能够让你在 R 中使用 C++。即便你对 C++ 没有很深刻的了解，也可以轻松用 R 的风格来写 C++。此外，Rcpp 在易用的同时并不会牺牲执行速度，任何人都可以以此获得高性能的结果。

本文档旨在给那些不熟悉 C++ 的用户提供（使用 Rcpp）必要的信息。因此，在某些情况，作者会从 Rcpp 的角度来解释其用法，让读者容易理解，而不是从 C++ 的视角来力求描述准确。

如果你能为本文档来提供反馈，我会十分感激。

项目原地址<sup>1</sup>

---

<sup>1</sup>[https://github.com/teuder/rcpp4everyone\\_en](https://github.com/teuder/rcpp4everyone_en)



# 本书编译环境

按照惯例，感谢 yihui 大佬，以及他的 **knitr** (Xie, 2015) 和 **bookdown** (Xie, 2019)。以下为本书的 R 进程信息：

```
sessionInfo()
```

```
## R version 3.6.0 (2019-04-26)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 10586)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=Chinese (Simplified)_China.936
## [2] LC_CTYPE=Chinese (Simplified)_China.936
## [3] LC_MONETARY=Chinese (Simplified)_China.936
## [4] LC_NUMERIC=C
## [5] LC_TIME=Chinese (Simplified)_China.936
##
## attached base packages:
## [1] stats      graphics  grDevices  utils      datasets
## [6] methods    base
##
## loaded via a namespace (and not attached):
## [1] compiler_3.6.0  magrittr_1.5    bookdown_0.12
## [4] tools_3.6.0     htmltools_0.3.6 yaml_2.2.0
```

```
## [7] Rcpp_1.0.1      stringi_1.4.3    rmarkdown_1.14
## [10] knitr_1.24       stringr_1.4.0    xfun_0.8
## [13] digest_0.6.20    evaluate_0.14
```

# 翻译的初衷

由于研究需要,我在尝试使用 C++ 来实现一些数值算法,求解优化问题。在倒腾 Rmarkdown 主题美化的时候,关注了 prettydoc<sup>2</sup>。巧的是,在作者 yihuan(后面发现也是给 Rcpp:R 与 C++ 的无缝整合写序的大佬)的 github<sup>3</sup>上,看到了很多相关的工作,比如优化和数值积分,Readme 上我看到的都是 fast 和 c++ 俩词。因此,开始学习 Rcpp<sup>4</sup>。

Rcpp:R 与 C++ 的无缝整合是我见到的第一份完备的 Rcpp 资料。拿到中文书籍的那一刻,爱不释手,然而读了一遍,大半内容我都是云里雾里。因此只能搁置。于是,算法只能用 R 写,速度慢也只能认了。后来在一些嵌入式系统上写了一段时间 c,回头再看书,内容渐渐明了起来。我这才意识到,此前是我没有达到看书的门槛。

Hadley Wickham 在 Advanced R<sup>5</sup>中的前言谈到,很多 R 用户并不是程序员,且 R 用户追求的是解决问题,而不在意该过程。相信同我一样,很多 R 用户,其他语言背景(尤其是 C/C++)很薄弱。这也意味着,以 C++ 的角度来学 Rcpp,会阻挡一部分的 R 用户迈入 Rcpp。

Rcpp for everyone<sup>6</sup>一书对这个问题给出了自己的答案。**This document focuses on providing necessary information to users who are not familiar with C++. Therefore, in some cases, I explain usage of Rcpp conceptually rather than describing accurately from the viewpoint of C++, so that I hope readers can easily understand it.** 与其设想的描述方式一致,本书的内容偏重于从 Rcpp

---

<sup>2</sup><https://github.com/yixuan/prettydoc>

<sup>3</sup><https://github.com/yixuan>

<sup>4</sup>RcppCore/Rcpp

<sup>5</sup><https://adv-r.hadley.nz/>

<sup>6</sup>rcpp4everyone\_en

的角度来讲问题，较少涉及到 C++ 的知识。这在保证用户理解的情况下，又能让用户以一种 R 语言的风格，来写 Rcpp 代码。在我看来，这可能是对 C/C++ 了解不多的 R 用户，最简明的 Rcpp 入门教材。

最后, Rcpp for everyone ==>> Advanced R Rcpp 部分 ==>> Rcpp:R 与 C++ 的无缝整合应该是我目前能发现的最平滑的 Rcpp 学习路线。希望本书的中文翻译能对更进一步地降低 Rcpp 的学习门槛，对大家的学习科研有所帮助。

jywang

二零一九年八月



# 作者简介

Masaki E.Tsuda, github 主页为 <https://github.com/teuder> , 著有 Rcpp for everyone<sup>7</sup>。

---

<sup>7</sup>[https://teuder.github.io/rcpp4everyone\\_en/](https://teuder.github.io/rcpp4everyone_en/)



# 第一章 啥情况该用 Rcpp

R 在做某些操作的时候是弱的。如果你需要做下面列出来的一些运算/操作，是时候考虑使用 Rcpp 了。

- 循环，下一次循环依赖此前的循环结果 (猜测应该是无法直接用向量化来加速循环，所以需要 Rcpp)
- 遍历一个向量或者矩阵中的每一个元素
- 有循环的递归函数
- 向量大小动态变化
- 需要更高端的数据结构和算法的操作



## 第二章 安装

在使用 Rcpp 开发之前，你需要安装一个 c++ 的编译器。

### 2.1 安装 C++ 编译器

#### 2.1.1 Windows

安装 Rtools。

参考 Rstan 的教程可能会有帮助（为 Windows 安装 Rtools）。

#### 2.1.2 Mac

安装 Xcode 命令行工具。在终端执行 `xcode-select --install` 命令。

#### 2.1.3 Linux

安装 gcc 和其他相关的包。在 Ubuntu Linux 中，终端执行 `sudo apt-get install r-base-dev` 命令。

#### 2.1.4 使用其他编译器

如果你安装有与上述不同的其他编译器 (g++, clang++)，在用户的根目录下创建如下的文件。然后在这个文件中设置环境变量。

**Linux, Mac**

- .R/Makevars

## Windows

- .R/Makevars.win

## 环境变量设定范例

```
CC=/opt/local/bin/gcc-mp-4.7
CXX=/opt/local/bin/g++-mp-4.7
CPLUS_INCLUDE_PATH=/opt/local/include:$CPLUS_INCLUDE_PATH
LD_LIBRARY_PATH=/opt/local/lib:$LD_LIBRARY_PATH
CXXFLAGS= -g0 -O2 -Wall
MAKE=make -j4
```

## 2.2 安装 Rcpp

用户能通过执行下面代码安装 Rcpp。

```
install.packages("Rcpp")
```

## 第三章 基本用法

按照如下三步，即可使用你的 Rcpp 函数。

1. 写（你的函数的）Rcpp 代码
2. 编译上述代码
3. 执行函数

### 3.1 写 Rcpp 代码

下面的代码定义了一个名为 `rcpp_sum()` 的函数，来计算一个向量的元素之和。保存代码块内容至 `sum.cpp` 文件。

**sum.cpp**

```
//sum.cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double rcpp_sum(NumericVector v){
    double sum = 0;
    for(int i=0; i<v.length(); ++i){
        sum += v[i];
    }
    return(sum);
}
```

### 3.1.1 在 Rcpp 中定义一个函数的格式

下面的代码展示了如何定义一个 Rcpp 函数的基本格式。

```
#include<Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
RETURN_TYPE FUNCTION_NAME(ARGUMENT_TYPE ARGUMENT){
    //do something
    return RETURN_VALUE;
}
```

- `#include<Rcpp.h>` : 保证你能使用 Rcpp 包中定义的类和函数
- `// [[Rcpp::export]]`: 在这行代码下定义的函数, 才能(在后面的步骤中)被 R 获取.
- `using namespace Rcpp;` : 这行代码是可选的。如果你不写, 那么你需要在特定的类和函数前面加上前缀 `Rcpp::` (比如, `Rcpp::NumericVector`)
- `RETURN_TYPE FUNCTION_NAME(ARGUMENT_TYPE ARGUMENT){}`: 需要指定函数的返回值和参数的类型, 以及函数名.
- `return RETURN_VALUE;`: `return` 声明强制返回一个值, 但是如果你不返回值(例如, `RETURN_TYPE` 是 `void` 型, 即空), 那么可以忽略 `return`。

## 3.2 编译代码

函数 `Rcpp::sourceCpp()` 会编译上述源代码并在 R 中加载。

```
library(Rcpp)
sourceCpp('sum.cpp')
```



## 3.3 执行函数

你可以像使用其他 R 函数一样，来使用加载好的 Rcpp 函数。

```
> rcpp_sum(1:10)
[1] 55
> sum(1:10)
[1] 55
```



## 第四章 将 Rcpp 嵌入 R 代码

三种方式可以让你在 R 代码中写 Rcpp 代码，即使用 `sourceCpp()`, `cppFunction()`, `evalCpp()` 函数。

### 4.1 `sourceCpp()`

不同于3.2中需要加载外部的 Rcpp 文件，你可以直接在 R 中，写 Rcpp 代码，并且将其保存为一个字符串对象，然后利用 `sourceCpp()` 加载这个对象。

```
src <-  
"#include <Rcpp.h>  
using namespace Rcpp;  
// [[Rcpp::export]]  
double rcpp_sum(NumericVector v){  
  double sum = 0;  
  for(int i=0; i<v.length(); ++i){  
    sum += v[i];  
  }  
  return(sum);  
}"  
sourceCpp(code = src)  
rcpp_sum(1:10)
```

## 4.2 cppFunction()

`cppFunction()` 提供了一种更加便利的方式来构建单一的 Rcpp 函数 (采用 `sourceCpp` 的方式可以写多个 Rcpp 函数被 R 调用)。使用 `cppFunction()` 时, 可以忽略 `#include <Rcpp.h>` 和 `using namespace Rcpp;`。

```
src <-  
  "double rcpp_sum(NumericVector v){  
    double sum = 0;  
    for(int i=0; i<v.length(); ++i){  
      sum += v[i];  
    }  
    return(sum);  
  }  
  "  
Rcpp::cppFunction(src)  
rcpp_sum(1:10)
```

## 4.3 evalCpp()

可以使用 `evalCpp()` 直接执行单个的 C++ 声明。

```
# Showing maximum value of double.  
evalCpp('std::numeric_limits<double>::max()')
```

## 第五章 C++11

C++11 是在 2011 年新建立的 C++ 标准，该标准引入了新的函数 (functionalities) 和符号 (notations)。相较于之前的标准，很多新的特性使得 C++ 对于初学者而言更为简单。本文档会对 C++11 的新特性做很多探索。

**重要：**代码示例是在默认 C++11 可得的情况下写的。

### 5.1 开启 C++11

为了开启 C++11，在你的 Rcpp 代码中任意一处加入下面的代码。

```
// [[Rcpp::plugins("cpp11")]]
```

### 5.2 推荐的 C++11 特性

#### 5.2.1 初始化

使用 {} 来初始化变量。

```
// 初始化向量  
// 下面三行代码等同于R中的 c(1, 2, 3).  
NumericVector v1 = NumericVector::create(1.0, 2.0, 3.0);
```

```
NumericVector v2 = {1.0, 2.0, 3.0};  
NumericVector v3 {1.0, 2.0, 3.0}; // 你可以忽略 "=".
```

### 5.2.2 auto

使用 `auto` 关键字，根据赋值，来推断变量的类型

```
// 变量 "i" 会是 int 型  
auto i = 4;  
NumericVector v;  
// 变量 "it" 会是 NumericVector::iterator  
auto it = v.begin();
```

### 5.2.3 decltype

通过使用 `decltype`，你能声明变量的类型与已存在的变量相同。

```
int i;  
decltype(i) x; // 变量 "x" 会是 int 型
```

### 5.2.4 基于范围的 for 循环

能用一种比较 R 风格的方式来写 for 循环。

```
IntegerVector v{1,2,3};  
int sum=0;  
for(auto& x : v) {  
    sum += x;  
}
```

### 5.2.5 Lambda 表达式

你能使用 lambda 表达式来创建一个函数对象。函数对象常用于作为未命名函数传递给其他函数。

Lambda 表达式的形式为 `[](){}.`

在 `[]` 中，写你希望在函数对象中使用的局部变量的列表。

- `[]` 不允许函数对象获取所有的局部变量。
- `[=]` 传值，将所有局部变量的值传递给函数对象。
- `[&]` 引用，函数直接引用所有的局部变量的值。
- `[=x, &y]` “x” 传值给函数对象，而 “y” 可以被函数对象直接引用。

在 `()` 中，写传递给函数的参数列表。

在 `{}` 中，写想做的操作。

#### lambda 表达式的返回类型

函数对象的返回类型会被自动地设定为 `{}` 操作中返回值的类型。如果你希望显式定义返回类型，可以按照 `[]()->int{}` 的方式。

例子下面的例子展示了如何使用 lambda 表达式。可以按照 R 的风格来写某些类型的 C++ 代码。

*R* 例子

```
v <- c(1,2,3,4,5)
A <- 2.0
res <-
  sapply(v, function(x){A*x})
```

译者：上面的 `sapply` 函数，将 `function(x){A*x}` 作用于 `v` 中的每个元素。这是很典型的向量化编程，可以看 `apply` 函数族来对此有更深入的了解。

*Rcpp* 例子

```
// [[Rcpp::plugins("cpp11")]]  
// [[Rcpp::export]]  
NumericVector rcpp_lambda_1(){  
  NumericVector v = {1,2,3,4,5};  
  double A = 2.0;  
  NumericVector res =  
    sapply(v, [&](double x){return A*x;});  
  return res;  
}
```

译者：上面代码中也使用 `sapply` 函数，将 R 中的 `function(x)(A*x)` 用 `lambda` 表达式的方式替代。而 `sapply`，则是吃到了 Rcpp 给我们提供的语法糖 (sugar)。如果要对语法糖有更深入的了解，可以阅读 Rcpp:R 与 C++ 的无缝整合中的相关章节。



## 第六章 信息打印

你可以通过 `Rprintf()` 和 `Rcout` 在 R 的控制台上来打印对象的信息和值。

`REprintf()` 和 `Rcerr` 可用于打印报错信息。

### 6.1 `Rcout`, `Rcerr`

使用 `Rcout` 和 `Rcerr` 的方式与使用 `std::cout` 和 `std::cerr` 方式相同。将你想要输出的信息，按照特定的顺序，使用 `<<` 输出符号链接。当在 `<<` 前是向量对象时，程序会打印向量的所有元素。

```
// [[Rcpp::export]]
void rcpp_rcout(NumericVector v){
    // 打印向量的所有元素值
    Rcout << "The value of v : " << v << "\n";
    // 打印报错信息
    Rcerr << "Error message\n";
}
```

### 6.2 `Rprintf()`, `REprintf()`

使用 `Rprintf()` 和 `REprintf()` 的方式和 `std::printf()` 相同，函数会按照指定的格式打印信息。

```
Rprintf( format, variables)
```

在 `format` 字符串中, 你可以使用下面的格式指示符来打印变量的值。如果要打印多个变量, 你需要按照对应的格式字符串的顺序, 来排列你的变量。

下面只列举了一部分的格式指示符, 可以参考其他的文档以深入探究(如, [cplusplus.com](http://www.cplusplus.com)<sup>1</sup>)。

specifier	explanation
<code>%i</code>	打印 signed integer ( <code>int</code> )
<code>%u</code>	打印 unsigned integer ( <code>unsigned int</code> )
<code>%f</code>	打印 floating point number ( <code>double</code> )
<code>%e</code>	打印 floating point number ( <code>double</code> ) in exponential style
<code>%s</code>	打印 C string ( <code>char*</code> )

此外, `Rprintf()` 和 `REprintf()` 只能打印在标准的 C 语言中已有的数据类型, 因此, 用户并不能直接传递 `Rcpp` 包中定义的数据类型, 如 `NumericVector` 给 `Rprintf()` 打印。如果你想进行此类操作, 那么你可能需要逐个元素进行传递打印, 代码如下。

```
// [[Rcpp::export]]
void rcpp_rprintf(NumericVector v){
    // printing values of all the elements of Rcpp vector
    for(int i=0; i<v.length(); ++i){
        Rprintf("the value of v[%i] : %f \n", i, v[i]);
    }
}
```

<sup>1</sup><http://www.cplusplus.com/reference/cstdio/printf/>

# 第七章 数据类型

Rcpp 提供了 R 中所有的基本数据类型。通过使用这些数据类型，你能够直接使用在 R 中的对象。

## 7.1 向量和矩阵

下面 7 种数据类型在 R 中被经常使用。

logical integer numeric complex character Date POSIXct

上面的 7 种类型与 Rcpp 中的向量 (vector) 类型和矩阵 (matrix) 类型是对应的（比如，有 logicalVector, integerVector 等类型，上面 7 种基本的数据类型都可以在后面加上 Vector 或者 Matrix）。

本文档中，Vector 和 Matrix 用于特指 Rcpp 中所有的向量和矩阵类型。

下表中展示了 R/Rcpp/C++ 中对应的数据类型。

R	Rcpp	Rcpp	Rcpp	C++
Value	vec- tor	vec- tor	ma- trix	scalar scalar
Logical	logical	Logical	Logical	Matrix bool
Integer	integer	Integer	Integer	Matrix int
Real	numeric	Numeric	Numeric	Matrix double
Complex	complex	Complex	Complex	Matrix complex
String	character	Character	Character	Matrix string (StringVector Matrix)
Date	Date	Date	Vector	Date -

R		Rcpp		Rcpp	
vec-	vec-	vec-	ma-	Rcpp	C++
Value	tor	tor	trix	scalar	scalar
DatetimeROSIXct		DatetimeVector		Datetime_t	

7.2 data.frame, list, S3, S4

除了向量和矩阵，在 R 中海油一些数据结构，比如 data.frame，list，S3 和 S4 类。所有这些数据结构同样也可以在 Rcpp 中处理。

R	Rcpp
data.frame	DataFrame
list	List
S3 class	List
S4 class	S4

在 Rcpp 中，Vector，DataFrame，List 都以向量的方式实现。即，Vector 是一个元素全部为标量的向量，DataFrame 是元素全部为向量的向量，List 是元素为各种各样数据类型的向量。因此，在 Rcpp 中 Vector，DataFrame，List 有很多共同的成员函数。

## 第八章 Vector 类

### 8.1 创建向量对象

你可以使用下面的几种方法来创建向量对象。

```
// 等价于  $v \leftarrow rep(0, 3)$ 
NumericVector v (3);

// 等价于  $v \leftarrow rep(1, 3)$ 
NumericVector v (3,1);

// 等价于  $v \leftarrow c(1,2,3)$ 
// C++11 初始化列表
NumericVector v = {1,2,3};

// 等价于  $v \leftarrow c(1,2,3)$ 
NumericVector v = NumericVector::create(1,2,3);

// 命名向量 等价于  $v \leftarrow c(x=1, y=2, z=3)$ 
NumericVector v =
    NumericVector::create(Named("x",1), Named("y")=2 , _["z"]=3);
```

## 8.2 获取向量元素

你可以使用 `[]` 或 `()` 运算符来获取一个向量的个别元素。两种操作符都接受数值向量/整型向量 (`NumericVector`/`IntegerVector`) 的数值索引, 字符向量的元素名索引和逻辑向量。`[]` 运算符会忽略边界溢出, 而 `()` 运算符会抛出 `index_out_of_bounds` 错误。

需要注意的是 C++ 中的向量索引开始于 0

```
// [[Rcpp::export]]
void rcpp_vector_access(){

    // 创建向量
    NumericVector v {10,20,30,40,50};
    // 设置元素名称
    v.names() = CharacterVector({"A","B","C","D","E"});

    // 准备向量索引
    NumericVector    numeric = {1,3};
    IntegerVector    integer = {1,3};
    CharacterVector  character = {"B","D"};
    LogicalVector    logical = {false, true, false, true, false};

    // 根据向量索引获取向量元素值
    double x1 = v[0];
    double x2 = v["A"];
    NumericVector res1 = v[numeric];
    NumericVector res2 = v[integer];
    NumericVector res3 = v[character];
    NumericVector res4 = v[logical];

    // 向量元素赋值
    v[0]    = 100;
    v["A"] = 100;
```

```
NumericVector v2 {100,200};  
v[numeric]    = v2;  
v[integer]    = v2;  
v[character]  = v2;  
v[logical]    = v2;  
}
```

## 8.3 成员函数

成员函数（也被称作方法）是某个对象中的函数。你可以以 `v.f()` 的形式来调用对象 `v` 中的成员函数 `f()`。

```
NumericVector v = {1,2,3,4,5};  
  
// 调用成员函数 length()，求对象 v 的长度  
int n = v.length(); // 5
```

Rcpp 中，向量对象的成员函数列举如下。

### 8.3.1 `length()`, `size()`

返回该向量对象中元素的个数。

```
//test.cpp 文件  
#include <Rcpp.h>  
using namespace Rcpp;  
  
// [[Rcpp::export]]  
int test(NumericVector v) {  
  
    Rcout << v.length() << '\n';  
}
```

```
Rcout << v.size() << '\n';  
return 0;  
}
```

在 R 中，运行结果为：

```
> sourceCpp("test.cpp")  
  
> test(c(1:10))  
10  
10  
[1] 0
```

### 8.3.2 names()

以字符向量的形式，返回该向量的元素名称。

```
//test.cpp 文件  
#include <Rcpp.h>  
using namespace Rcpp;  
  
// [[Rcpp::export]]  
CharacterVector test(NumericVector v) {  
  
    return v.names();  
}
```

在 R 中，运行结果为：

```
> sourceCpp("test.cpp")
```



```
> test(c(a = 1,b = 2, c = 3))  
[1] "a" "b" "c"
```

### 8.3.3 offset( name ), findName( name )

按照指定字符串 `name` 的方式，返回对应元素的数值索引。

```
//test.cpp 文件  
#include <Rcpp.h>  
using namespace Rcpp;  
  
// [[Rcpp::export]]  
int test(NumericVector v, std::string name) {  
  
    return v.offset(name);  
}
```

在 R 中，运行结果为：

```
> sourceCpp("test.cpp")  
  
> test(c(a = 1,b = 2, c = 3),"a")  
[1] 0
```

### 8.3.4 offset( i )

函数在检查数值索引 `i` 没有超过边界后，返回该索引。

举例说明，在 `test.cpp` 文件中键入以下代码，

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int test(NumericVector v, int i) {
    return v.offset(i);
}
```

在 R 中，运行结果为：

```
> sourceCpp('test.cpp')

> test(c(11,22,33,44,55),2)
[1] 2
```

### 8.3.5 fill( x )

将该向量的所有元素用标量  $x$  填充。

举例说明，在 `test.cpp` 文件中键入以下代码，

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v, double i) {
    v.fill(i);
    return v;
}
```

```
> sourceCpp("test.cpp")

> test(c(1,2,3,4,5),6)
[1] 6 6 6 6 6
```

### 8.3.6 sort()

将该向量对象中的元素升序排列。

举例说明，在 `test.cpp` 文件中键入以下代码，

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v) {
  v.sort();
  return v;
}
```

```
> sourceCpp("test.cpp")

> test(c(5,4,7,6,8))
[1] 4 5 6 7 8
```

### 8.3.7 assign( first\_it, last\_it )

assign values specified by the iterator `first_it` and `last_it` to this vector object. 将迭代器 `first_it` 至 `last_it` 所指向的元素赋给向量对象。

举例说明，在 `test.cpp` 文件中键入以下代码，

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v) {
    NumericVector v1;
    v1.assign(v.begin(),v.end());
    return v1;
}
```

`begin` 和 `end` 也是成员函数，下面8.3.12也有对应的例子

```
> sourceCpp("test.cpp")

> test(1:10)
[1] 1 2 3 4 5 6 7 8 9 10
```

### 8.3.8 push\_back( x )

在向量对象的最后加入新的标量值 `x`。

在 `test.cpp` 文件中键入以下代码，

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v, double var_PB) {

    v.push_back(var_PB);
    return v;
}
```

```
> sourceCpp("test.cpp")

> test(c(1,2,3,4,5),6)
[1] 1 2 3 4 5 6
```

### 8.3.9 push\_back( x, name )

在向量后加入标量元素  $x$  时，指定其元素名称。

在 `test.cpp` 文件中键入以下代码，

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v,
                   double var_PB,
                   std::string var_Name) {

    //v.names() = CharacterVector::create("a","b","c","d","e");
    v.push_back(var_PB,var_Name);
    return v;
}
```

```
> sourceCpp('test.cpp')

> test(c(a = 1,b = 2,c = 3,d = 4,e = 5),6,"f")
a b c d e f
1 2 3 4 5 6
```

### 8.3.10 push\_front( x )

在向量前面加入一个标量  $x$ 。

在 `test.cpp` 文件中键入以下代码,

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v,
                   double var_f) {
    v.push_front(var_f);
    return v;
}
```

```
> sourceCpp('test.cpp')

> test(c(1,2,3,4,5),6)
6 1 2 3 4 5
```

### 8.3.11 push\_front( x, name )

在向量前加入标量元素 `x` 时, 指定其元素名称。

在 `test.cpp` 文件中键入以下代码,

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v,
                   double var_f,
                   std::string var_Name) {

    //v.names() = CharacterVector::create("a","b","c","d","e");
```

```

    v.push_front(var_f,var_Name);
    return v;
}

```

```

> sourceCpp('test.cpp')

> test(c(a = 1,b = 2,c = 3,d = 4,e = 5),6,"f")
f a b c d e
6 1 2 3 4 5

```

### 8.3.12 begin()

返回一个指向向量第一个元素的迭代器。

### 8.3.13 end()

返回一个指向向量最后一个元素的迭代器。(one past the last element of this vector).

以求和函数说明 begin() 和 end() 的作用。

```

#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double test(NumericVector v) {

    double mysum = 0;
    NumericVector::iterator it;
    for(it = v.begin();it!=v.end();it++){
        mysum += *it;
    }
}

```

```
    return mysum;
}
```

```
> sourceCpp('test.cpp')

> test(1:10)
[1] 55
```

在循环体内，我们用 `*it` 获取向量 `v` 中的元素，在指明循环范围的时候，也并不是我们熟悉的 `int i = 0; i < n; i++`

该例子来源于 Advanced R 中案例，请点击传送门<sup>1</sup>。

### 8.3.14 `cbegin()`

返回一个指向向量第一个元素的具有 `const` 属性的迭代器。

无法用于元素的修改

### 8.3.15 `cend()`

返回一个指向向量最后一个元素的具有 `const` 属性的迭代器。**(one past the last element of this vector)**.

以求和函数说明 `cbegin()` 和 `cend()` 的作用。下面的例子只在声明迭代器 `it` 的时候，将 `iterator` 改为 `const_iterator`，因为 `cbegin()` 和 `cend()` 得到是 `const_iterator`。

---

<sup>1</sup><http://adv-r.had.co.nz/Rcpp.html#rcpp-classes>



```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double test(NumericVector v) {

    double mysum = 0;
    NumericVector::const_iterator it;
    for(it = v.begin(); it!=v.end(); it++){
        mysum += *it;
    }
    return mysum;
}
```

当然,对于 c++ 不熟悉的用户,完全可以忽视 `const_iterator`。不声明 `it`, 而是采用 `auto`, 如下。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::plugins(cpp11)]]
// [[Rcpp::export]]
double test(NumericVector v) {

    double mysum = 0;
    for(auto it = v.cbegin(); it!=v.cend(); it++){
        mysum += *it;
    }
    return mysum;
}
```

上面的这段代码, 在循环体中, 使用 `auto`, 来自动判别 `it` 的类型。对于

不熟悉 C++ 的用户而言（也包括我），是十分便捷的。但需要注意的是，一定要加上 `// [[Rcpp::plugins(cpp11)]]`，表明你希望使用 c++11 的新特性，否则程序会报错。

```
> sourceCpp('test.cpp')  
  
> test(1:10)  
[1] 55
```

### 8.3.16 insert( i, x )

在数值索引 `i` 指定的位置插入标量 `x`。返回一个指向插入元素的迭代器。

test.cpp 文件如下：

```
#include <Rcpp.h>  
using namespace Rcpp;  
  
// [[Rcpp::export]]  
NumericVector test(NumericVector v) {  
  v.insert(1,6);  
  return v;  
}
```

R 运行结果如下：

```
> sourceCpp("test.cpp")  
  
> test(2:5)  
[1] 2 6 3 4 5
```

### 8.3.17 insert( it, x )

在迭代器 `it` 指定的位置插入标量 `x`。返回迭代器指向的元素。

test.cpp 文件如下：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v) {
  v.insert(v.begin()+1,6);
  return v;
}
```

R 运行结果如下：

```
> sourceCpp("test.cpp")

> test(2:5)
[1] 2 6 3 4 5
```

### 8.3.18 erase(i)

擦除数值索引 `i` 指定的标量元素 `x`。返回指向擦除元素之后一个元素的迭代器。

test.cpp 文件如下：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double test(NumericVector v) {
```

```
NumericVector::iterator it = v.erase(0);  
return *it;  
}
```

R 运行结果如下:

```
> sourceCpp("test.cpp")  
  
> test(1:5)  
[1] 2
```

### 8.3.19 erase(it)

擦除迭代器 `it` 指向的元素。返回指向擦除元素之后一个元素的迭代器。

`test.cpp` 文件如下:

```
#include <Rcpp.h>  
using namespace Rcpp;  
  
// [[Rcpp::export]]  
double test(NumericVector v) {  
  NumericVector::iterator it = v.erase(v.begin());  
  return *it;  
}
```

R 运行结果如下:

```
> sourceCpp("test.cpp")  
  
> test(1:5)  
[1] 2
```

### 8.3.20 erase( first\_i, last\_i )

擦除数值索引 first\_i 至 last\_i - 1 之间的所有元素。返回指向擦除元素之后一个元素的迭代器。

test.cpp 文件如下：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double test(NumericVector v) {
    NumericVector::iterator it = v.erase(0,3);
    return *it;
}
```

R 运行结果如下：

```
> sourceCpp("test.cpp")

> test(1:5)
[1] 4
```

由于擦除的是索引 0 和 3-1，即，第 1 个元素至第 3 个元素被擦除，返回的是对应原本第四个元素的迭代器，\*it 为 4，也印证了结果。

### 8.3.21 erase( first\_it, last\_it )

擦除迭代器 first\_it 至 last\_it - 1 之间的所有元素。返回指向擦除元素之后一个元素的迭代器。

test.cpp 文件如下：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double test(NumericVector v) {
    NumericVector::iterator it = v.erase(v.begin(),v.end());
    return *it;
}
```

R 运行结果如下:

```
> sourceCpp("test.cpp")

> test(1:5)
[1] 5
```

### 8.3.22 containsElementNamed(name)

如果向量包含有某一个元素，其名称与字符串 `name` 相同，那么返回 `true`。

`test.cpp` 文件如下:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
bool test(NumericVector v) {
    if(v.containsElementNamed("b")){
        Rcout <<"name is included" << '\n';
    }else{
        Rcout <<"name is not included" << '\n';
    }
}
```

```
    return v.containsElementNamed("b");  
}
```

R 运行结果如下:

```
> sourceCpp("test.cpp")  
  
> test(c(a = 1, b = 2, c = 3))  
name is included  
[1] TRUE
```

## 8.4 静态成员函数

静态成员函数是对象所在类的函数。k 可以按照 `NumericVector::create()` 的方式来调用该静态成员函数。

### 8.4.1 `get_na()`

返回 `Vector` 类中的 NA 值。

test.cpp 文件如下:

```
#include <Rcpp.h>  
using namespace Rcpp;  
  
// [[Rcpp::export]]  
NumericVector test(NumericVector v) {  
    v.fill(NumericVector::get_na());  
    return v;  
}
```

R 运行结果如下:

```
> sourceCpp("test.cpp")

> test(1:5)
[1] NA NA NA NA NA
```

该例子有参考 [stackoverflow](#) 上的答案，详情点击传送门<sup>2</sup>。

### 8.4.2 is\_na(x)

如果  $x$  为 NA，则返回 true。

test.cpp 文件如下：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
bool test(NumericVector v,int id) {
  return NumericVector::is_na(v(id));
}
```

R 运行结果如下：

```
> sourceCpp("test.cpp")

> test(c(1:3,NA,5),3)
[1] TRUE
```

### 8.4.3 create( x1, x2, ...)

创建一个 Vector 对象，其包含的元素由标量  $x_1, x_2$  指定。参数最大个数为 20。

---

<sup>2</sup><https://stackoverflow.com/questions/23748572/initializing-a-matrix-to-na-in-rcpp>



可以命名元素或不命名，例子如下：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test() {

    NumericVector v = NumericVector::create(_["a"] = 1, _["b"] = 2, _["c"] = 3, _["d"] = 4);
    //NumericVector v = NumericVector::create(1,2,3,4);
    return v;
}
```

```
> sourceCpp("test.cpp")

> test()
a b c d
1 2 3 4
>
```

#### 8.4.4 import( first\_it , last\_it )

创建一个 Vector 对象，其元素由迭代器 first\_it 至 last\_it - 1 指定。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector test(NumericVector v) {
    NumericVector v1 = NumericVector::import(v.begin(),v.end());
}
```

```
    return v1;
}
```

```
> sourceCpp("test.cpp")

> test(1:5)
[1] 1 2 3 4 5
```

#### 8.4.5 import\_transform( first\_it, last\_it, func)

在 `import( first_it , last_it )` 的基础上, 对于每一个迭代器范围内的元素, 进行 `func` 函数的操作。类似于 `apply()` 函数族。

```
#include <Rcpp.h>
using namespace Rcpp;

//构建mypow2函数, 求取元素平方

// [[Rcpp::export]]
double mypow2(double x) {
    return x*x;
}

// [[Rcpp::export]]
NumericVector test(NumericVector v) {
    NumericVector v1 = NumericVector::import_transform(v.begin(),v.end(),mypow2);
    return v1;
}
```

```
> sourceCpp("test.cpp")
```

```
> test(1:5)
```

```
[1]  1  4  9 16 25
```



## 第九章 Matrix 类

### 9.1 创建矩阵对象

矩阵对象可由如下几种方式创建。

```
// 创建一个矩阵对象，等价于在R语句
// m <- matrix(0, nrow=2, ncol=2)
NumericMatrix m1( 2 );
// m <- matrix(0, nrow=2, ncol=3)
NumericMatrix m2( 2 , 3 );
// m <- matrix(v, nrow=2, ncol=3)
NumericMatrix m3( 2 , 3 , v.begin() );
```

此外，R 中的矩阵对象，实际上是行数和列数在属性 `dim` 中设定好的向量。因此，如果你在 Rcpp 中，创建一个有 `dim` 属性的向量，并且将其作为返回值传递给 R，那么该向量在 R 中会被作为矩阵对待。

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
NumericVector rcpp_matrix(){
    // 创建一个向量对象
```

```

NumericVector v = {1,2,3,4};
// 设置该对象的`dim`属性
v.attr("dim") = Dimension(2, 2);
// 返回该对象给R
return v;
}

```

需要注意的是, c++98 是不允许直接使用 `v = {1,2,3,4}` 来赋值的, 因此, 需要加上 `// [[Rcpp::plugins(cpp11)]]`, 确保能使用 C++11 的新特性。

执行结果:

```

> rcpp_matrix()
      [,1] [,2]
[1,]     1     3
[2,]     2     4

```

然而, 即便你给某个向量对象的 `dim` 属性赋值, 在 Rcpp 中对象的类型还是为向量类。因此, 如果你希望在 Rcpp 中, 将其转化为矩阵类, 你需要使用 `as<T>()` 函数。

```

// 设定维度属性
v.attr("dim") = Dimension(2, 2);
// 转为矩阵类
NumericMatrix m = as<NumericMatrix>(v);

```

## 9.2 访问矩阵元素

通过使用 `()` 符号, 你可以指定行, 列号来获取, 分配矩阵对象的元素值。和在向量中的索引类似, 矩阵中的行与列号也是从 0 开始。如果你

希望获取某一行或者一列，使用 `_` 符号。也可以使用 `[]` 操作符，来访问矩阵元素（将矩阵理解为按列连接的向量）。

```
// 创建一个5*5的矩阵
NumericMatrix m( 5, 5 );
// 检索0行，2列的元素，即第一行，第三列的元素
double x = m( 0 , 2 );
// 将0行（向量）赋给v
NumericVector v = m( 0 , _ );
// 将2列赋给v
NumericVector v = m( _ , 2 );
// 将矩阵m的0~1行，2~3列赋值给矩阵m2
NumericMatrix m2 = m( Range(0,1) , Range(2,3) );
// 按照向量的方式来检索矩阵元素
m[5]; // 指向m(0,1)的位置，即第6个元素（矩阵按列连接），为第1行，第2列元素
```

### 9.2.1 访问行，列与子矩阵

Rcpp 也提供了类型来进行矩阵特定部分的“引用”(references)。

```
NumericMatrix::Column col = m( _ , 1); // 对于列1的引用
NumericMatrix::Row row = m( 1 , _ ); // 对于行1的引用
NumericMatrix::Sub sub = m( Range(0,1) , Range(2,3) ); //对于子矩阵的引用
```

对这些“引用”对象的赋值，等效于直接修改其原始矩阵的对应部分。比如，对于上面例子中的 `col` 对象进行赋值，会直接把 `m` 的列 1 的值进行对应的修改。

在 `test.cpp` 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;
```

```
// [[Rcpp::export]]
NumericMatrix test(NumericMatrix v, int idx){
  NumericMatrix::Column col = v( _, idx);
  //将idx列的所有元素乘以2, v也会被修改
  col = col * 2;

  //上行代码等效于
  //v( _, idx) = 2 * v( _, idx);
  return v;
}
```

在 R 中的执行结果为:

```
> sourceCpp('test.cpp')

> a <- matrix(1:16,4,4)

> a
      [,1] [,2] [,3] [,4]
[1,]     1     5     9    13
[2,]     2     6    10    14
[3,]     3     7    11    15
[4,]     4     8    12    16

> test(a,1)
      [,1] [,2] [,3] [,4]
[1,]     1    10     9    13
[2,]     2    12    10    14
[3,]     3    14    11    15
[4,]     4    16    12    16
```

可以看到, 矩阵的第 2 列 (Rcpp 索引为 1) 已经变为原来的 2 倍。



## 9.3 成员函数

Since `Matrix` is actually `Vector`, `Matrix` basically has the same member functions as `Vector`. Thus, member functions unique to `Matrix` are only presented below.

此前提到, `Matrix` 实际上也是 `Vector`, 所以, `Matrix` 基本上与 `Vector` 有着相同的成员函数。因此, 在此列出 `Matrix` 自身独特的成员函数。

### 9.3.1 `nrow()` `rows()`

返回行数。

在 `test.cpp` 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int test(NumericMatrix v){

    return v.nrow();
}
```

在 R 中的执行结果为:

```
> sourceCpp('test.cpp')

> test(matrix(1:16,2,8))
[1] 2
```

### 9.3.2 `ncol()` `cols()`

返回矩阵列数。

在 `test.cpp` 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int test(NumericMatrix v){

    return v.ncol();
}
```

在 R 中的执行结果为：

```
> sourceCpp('test.cpp')

> test(matrix(1:16,2,8))
[1] 8
```

### 9.3.3 row( i )

返回矩阵行 `i` 的“引用”，关于“引用”的具体信息，可参考[9.2.1](#)。

在 `test.cpp` 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix test(NumericMatrix v){
    v.row(1) = 2 * v.row(1);
    return v;
}
```

在 R 中的执行结果为：

```
> sourceCpp('test.cpp')

> (a <- matrix(1:8,2,4))
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

> test(a)
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    4    8   12   16
```

#### 9.3.4 column( i )

返回矩阵行 i 列的“引用”

#### 9.3.5 fill\_diag( x )

使用 x 填充矩阵对角线元素。

在 test.cpp 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix test(NumericMatrix v, double x){
  v.fill_diag(x);
  return v;
}
```

在 R 中的执行结果为：

```
> sourceCpp('test.cpp')

> (a <- matrix(0,4,4))
      [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0

> test(a,1)
      [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

### 9.3.6 offset( i, j )

返回  $i$  行和  $j$  列对应的元素，在将矩阵作为列向量连接时其对应的索引。

`matrix` 中的 `offset` 函数是私有的，似乎因此导致调用失败。

## 9.4 静态成员函数

`Matrix` 基本上有着和 `Vector` 相同的成员函数。其独特的成员函数在此处列出。

### 9.4.1 Matrix::diag( size, x )

返回一个矩阵，行列数均为 `size`，对角元素为 `x`。

在 `test.cpp` 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix test(int size, double x){
  NumericMatrix v = NumericMatrix::diag(size,x);
  return v;
}
```

在 R 中的执行结果为:

```
> sourceCpp('test.cpp')

> test(3L,1)
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

## 9.5 与 Matrix 相关的其他函数

此部分展示一些其他与矩阵相关的函数。

### 9.5.1 rownames( m )

获取和设定矩阵行名。

```
CharacterVector ch = rownames(m);
rownames(m) = ch;
```

在 test.cpp 文件中键入下面的代码。

```

#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix test(NumericMatrix v1, NumericMatrix v2){
  CharacterVector v1_rname = rownames(v1);
  rownames(v2) = v1_rname;
  return v2;
}

```

在 R 中的执行结果为:

```

> sourceCpp('test.cpp')

> a <- matrix(0,3,3)

> b <- matrix(1,3,3)

> rownames(a) <- c('a','b','c')

> a;b
  [,1] [,2] [,3]
a     0     0     0
b     0     0     0
c     0     0     0
  [,1] [,2] [,3]
[1,]   1   1   1
[2,]   1   1   1
[3,]   1   1   1

> test(a,b)
  [,1] [,2] [,3]
a     1     1     1

```

```
b    1    1    1
c    1    1    1
```

### 9.5.2 colnames( m )

获取和设定矩阵列名, 方法同上。

```
CharacterVector ch = colnames(m);
colnames(m) = ch;
```

### 9.5.3 transpose( m )

返回矩阵 `m` 的转置。

在 `test.cpp` 文件中键入下面的代码。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix test(NumericMatrix v){
    return transpose(v);
}
```

在 R 中的执行结果为:

```
> sourceCpp('test.cpp')

> (a <- matrix(1:9,3,3))
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

```
[3,]    3    6    9
```

```
> test(a)
```

```
    [,1] [,2] [,3]
```

```
[1,]    1    2    3
```

```
[2,]    4    5    6
```

```
[3,]    7    8    9
```



# 第十章 向量运算

## 10.1 数学运算

使用  $+$   $-$   $*$   $/$  运算符, 用户可以对相同长度的向量进行元素级别的运算。

```
NumericVector x ;
NumericVector y ;
// 向量与向量运算
NumericVector res = x + y ;
NumericVector res = x - y ;
NumericVector res = x * y ;
NumericVector res = x / y ;
// 向量与标量运算
NumericVector res = x + 2.0 ;
NumericVector res = 2.0 - x;
NumericVector res = y * 2.0 ;
NumericVector res = 2.0 / y;
// expression and expression operation
NumericVector res = x * y + y / 2.0 ;
NumericVector res = x * ( y - 2.0 ) ;
NumericVector res = x / ( y * y ) ;
```

-号逆转了符号。

```
NumericVector res = -x ;
```

## 10.2 比较运算

使用 `==` `!=` `<` `>` `=` `<=` 运算符进行向量比较，会产生逻辑向量。用户可以通过逻辑向量来访问向量元素。

```
NumericVector x ;
NumericVector y ;
// Comparison of vector and vector
LogicalVector res = x < y ;
LogicalVector res = x > y ;
LogicalVector res = x <= y ;
LogicalVector res = x >= y ;
LogicalVector res = x == y ;
LogicalVector res = x != y ;
// Comparison of vector and scalar
LogicalVector res = x < 2 ;
LogicalVector res = 2 > x ;
LogicalVector res = y <= 2 ;
LogicalVector res = 2 != y ;
// Comparison of expression and expression
LogicalVector res = ( x + y ) < ( x*x ) ;
LogicalVector res = ( x + y ) >= ( x*x ) ;
LogicalVector res = ( x + y ) == ( x*x ) ;
```

! 表示对逻辑值的否运算。

```
LogicalVector res = ! ( x < y ) ;
```

通过逻辑向量来访问向量元素。

```
NumericVector res = x[x < 2];
```



# 第十一章 逻辑运算符

## 11.1 LogicalVector

### 11.1.1 LogicalVector 元素的数据类型

LogicalVector 的元素类型不是 `bool`。这是因为，`bool` 只能表示 `true` 和 `false`，但是在 R 中，逻辑向量有三种可能的取值，即 `TRUE`，`FALSE` 以及 `NA`。因此，LogicalVector 元素的数据类型为 `int`，而非 `bool`。

在 Rcpp 中，`TRUE` 用 1 表示，`FALSE` 用 0 表示，而 `NA` 由 `NA_LOGICAL` 表示 (整型的最小值: -2147483648)。

R	Rcpp	int	bool
TRUE	TRUE	1 (除却-2147483648 至 0 之间的 int)	true
FALSE	FALSE	0	false
NA	NA_LOGICAL	-2147483648	true

## 11.2 逻辑运算符

使用运算符 `&` (与) `|` (或) `!` (非) 来对 LogicalVector 中的每个元素进行逻辑运算。

```
LogicalVector v1 = {1,1,0,0};
LogicalVector v2 = {1,0,1,0};
LogicalVector res1 = v1 & v2;
```

```
LogicalVector res2 = v1 | v2;
LogicalVector res3 = !(v1 | v2);
Rcout << res1 << "\n"; // 1 0 0 0
Rcout << res2 << "\n"; // 1 1 1 0
Rcout << res3 << "\n"; // 0 0 0 1
```

## 11.3 接收 LogicalVector 的函数

接收 LogicalVector 的函数有 `all()`, `any()` 及 `ifelse()` 等。

### 11.3.1 `all()`, `any()`

对于 LogicalVector `v`, 当所有元素都为 TRUE 时, `all (v)` 返回 TRUE, 当任意一个元素为 TRUE 时, `any (v)` 返回 TRUE,

然而, 用户并不能在 `if` 语句的条件表达式中, 使用 `all()` 或者 `any()` 的返回值。这是因为这两者的返回值并不是 `bool` 型, 而是 `SingleLogicalResult` 型。如果要在 `if` 条件语句中使用这两个函数, 可以考虑使用 `is_true()`, `is_false()` 和 `is_na()`。这些函数会把 `SingleLogicalResult` 类型转为 `bool` 型。

下面的代码展示了, 如何在 `if` 语句的条件表达式中使用 `all()` 和 `any()`。在这个例子中, 条件表达式的值为 `true`, `all()` 和 `any()` 的返回值也会被打印显示。

在 `test.cpp` 文件中输入下面代码。

```
// [[Rcpp::export]]
List rcpp_logical_03(){
    LogicalVector v1 = LogicalVector::create(1,1,1,NA_LOGICAL);
    LogicalVector v2 = LogicalVector::create(0,1,0,NA_LOGICAL);
    // 对于包含有NA的Logical向量, all (), any () 函数的结果与R一致
    LogicalVector lv1 = all( v1 );    // NA
    LogicalVector lv2 = all( v2 );    // FALSE
```

```

LogicalVector lv3 = any( v2 ); // TRUE
// 将 `SingleLogicalResult` 类型转为 `bool` 型, 然后赋值
bool b1 = is_true ( all(v1) ); // false
bool b2 = is_false( all(v1) ); // false
bool b3 = is_na    ( all(v1) ); // true
// 在 if 语句条件判别式中的情况
if(is_na(all( v1 ))) { // OK
    Rcout << "all( v1 ) is NA\n";
}
//打印所有信息
Rcout << "lv1" << lv1 << '\n';
Rcout << "lv2" << lv2 << '\n';
Rcout << "lv3" <<lv3 << '\n';
Rcout << "b1: " << b1 << '\n';
Rcout << "b2: " << b2 << '\n';
Rcout << "b3: " << b3 << '\n';
return List::create(lv1, lv2, lv3, b1, b2, b3);
}

```

在 R 中的运行结果为:

```

> sourceCpp('test.cpp')

> test_list <- rcpp_logical_03()
all( v1 ) is NA
lv1: -2147483648
lv2: 0
lv3: 1
b1: 0
b2: 0
b3: 1

```

需要注意的是, 在 @ref(#LogicalVector-elements) 中提到过, NA 的值为-2147483648, 与打印的 lv1 信息一致。

### 11.3.2 ifelse()

`ifelse(v, x1, x2)` 接收逻辑向量 `v`，如果 `v` 中的某元素为 `TRUE`，那么返回 `x1` 中对应位置的元素，如果为 `FALSE`，那么返回 `x2` 中对应位置的元素。尽管 `x1` 和 `x2` 可以是标量或者向量，但如果是向量，两者的长度必须与 `v` 的长度一致。

```
// [[Rcpp::export]]
int rcpp_logical_02(NumericVector v1, NumericVector v2){

    //向量元素个数
    int n = v1.length();
    // 情况1: x1 和 x2是标量的情况
    IntegerVector res1      = ifelse( v1>v2, 1, 0);
    NumericVector res2      = ifelse( v1>v2, 1.0, 0.0);

    //CharacterVector res3 = ifelse( v1>v2, "T", "F"); // 不支持此种写法
    //ifelse() 不支持字符串标量，为了得到和R一样的结果
    // 我们需要使用字符串向量，该向量所有元素相同
    CharacterVector chr_v1 = rep(CharacterVector("T"), n);
    CharacterVector chr_v2 = rep(CharacterVector("F"), n);
    CharacterVector res3   = ifelse( v1>v2, chr_v1, chr_v2);
    Rcout << "case1: both x1 and x2 are scalar"<<'\n';
    Rcout << "\t v1 > v2 " << res1 <<'\n';
    Rcout << "\t v1 > v2 " << res2 <<'\n';
    Rcout << "\t v1 > v2 " << res3 <<'\n';

    //情况2, x1是向量, x2是标量
    IntegerVector int_v1, int_v2;
    int_v1 = rep(1,n);
    int_v2 = rep(0,n);
    NumericVector num_v1, num_v2;
    num_v1 = rep(1.,n);
    num_v2 = rep(0.,n);
```



```

IntegerVector  res4 = ifelse( v1>v2, int_v1, 0);
NumericVector  res5 = ifelse( v1>v2, num_v1, 0.0);
CharacterVector res6 = ifelse( v1>v2, chr_v1, Rf_mkChar("F")); // Note

Rcout <<"case2: x1 and x2 are vector and scalar"<<'\n';
Rcout << "\t v1 > v2 " << res4 <<'\n';
Rcout << "\t v1 > v2 " << res5 <<'\n';
Rcout << "\t v1 > v2 " << res6 <<'\n';

//情况3, x1和x2均为向量
IntegerVector  res7 = ifelse( v1>v2, int_v1, int_v2);
NumericVector  res8 = ifelse( v1>v2, num_v1, num_v2);
CharacterVector res9 = ifelse( v1>v2, chr_v1, chr_v2);

Rcout <<"case3: both x1 and x2 are vector"<<'\n';
Rcout << "\t v1 > v2 " << res7 <<'\n';
Rcout << "\t v1 > v2 " << res8 <<'\n';
Rcout << "\t v1 > v2 " << res9 <<'\n';

return 0;
}

```

Note: Rf\_mkChar () 函数作用为将 C 语言中的字符串 (char\*) 转为 CHARSXP(CharacterVector 中的元素类型)。

在 R 中运行结果为:

```

> sourceCpp('test.cpp')

> tmp <- rcpp_logical_02(1:4,4:1)
case1: both x1 and x2 are scalar
      v1 > v2 0 0 1 1
      v1 > v2 0 0 1 1

```

```

v1 > v2 "F" "F" "T" "T"
case2: x1 and x2 are vector and scalar
v1 > v2 0 0 1 1
v1 > v2 0 0 1 1
v1 > v2 "F" "F" "T" "T"
case3: both x1 and x2 are vector
v1 > v2 0 0 1 1
v1 > v2 0 0 1 1
v1 > v2 "F" "F" "T" "T"

```

## 11.4 LogicalVector 元素的估值

LogicalVector 的元素值不应当被用作 if 语句的条件表达式。因为, C++ 中 if 语句将条件表达式评估为 bool 型。而 bool 型把所有非零的值均评估为 true, 因此, LogicalVector 中的 NA(NA\_LOGICAL) 也会被认为是 true。

下面的代码示例展示了 if 语句是如何评估 LogicalVector 的元素值。

```

// [[Rcpp::plugins(cpp11)]]
// [[Rcpp::export]]
LogicalVector rcpp_logical(){
  // 构建一个包含NA值得整型向量
  IntegerVector x = {1,2,3,4,NA_INTEGER};
  // 比较运算的结果是逻辑向量
  LogicalVector v = (x >= 3);
  //如果将逻辑向量的元素直接用于if语句中, NA_LOGICAL会被认为是TRUE
  for(int i=0; i<v.size();++i) {
    if(v[i]) Rprintf("v[%i]:%i is evaluated as true.\n",i,v[i]);
    else Rprintf("v[%i]:%i is evaluated as false.\n",i,v[i]);
  }
  // 评估逻辑向量的元素
  for(int i=0; i<v.size();++i) {

```

```

    if(v[i]==TRUE) Rprintf("v[%i] is TRUE.\n",i);
    else if (v[i]==FALSE) Rprintf("v[%i] is FALSE.\n",i);
    else if (v[i]==NA_LOGICAL) Rprintf("v[%i] is NA.\n",i);
    else Rcout << "v[" << i << "] is not 1\n";
}
// 打印 TRUE, FALSE 和 NA_LOGICAL 的值
Rcout << "TRUE " << TRUE << "\n";
Rcout << "FALSE " << FALSE << "\n";
Rcout << "NA_LOGICAL " << NA_LOGICAL << "\n";
return v;
}

```

需要注意的是,在原始代码上,需要加入// [[Rcpp::plugins(cpp11)]] 语句启动 C++11 的特性,否则代码会因为初始化语句而报错。

执行结果为:

```

> sourceCpp('test.cpp')

> rcpp_logical()
v[0]:0 is evaluated as false.
v[1]:0 is evaluated as false.
v[2]:1 is evaluated as true.
v[3]:1 is evaluated as true.
v[4]:-2147483648 is evaluated as true.
v[0] is FALSE.
v[1] is FALSE.
v[2] is TRUE.
v[3] is TRUE.
v[4] is NA.
TRUE 1
FALSE 0
NA_LOGICAL -2147483648
[1] FALSE FALSE TRUE TRUE NA

```



## 参考文献

- Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.
- Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.12.



# 索引

bookdown, [xi](#)

knitr, [xi](#)