Seamless R and C++ Integation with Rcpp

Wang, Liuying

2017-05-27

1. 引言

1.1.Rcpp 介绍

Rcpp 几乎可以认为是 R 语言的一个里程碑。R 本身自带了 C 语言接口，但并不是那么好用，尤其是涉及内存管理的时候，而 Rcpp 成功的解决了这个问题，实现了 “无缝链接”。

1.1.1Rcpp 作者们

我们先看看 Rcpp 的作者们，几乎是现有最强大的一个 package 团队。

- **Dirk Eddelbuettel**：Rcpp 现在的主要维护者之一，就是 Springer 那本书的作者，Ketchum Trading 公司高级量化分析师，资深 quant。Dirk 是 Debian/Ubuntu 下 R 的维护者，R/Finance 会议发起人之一，其个人博客上有历次会议上报告的 slide，R 社区重要技术网站之一。
- **Romain Francois**：Rcpp 现在的主要维护者之一，独立的 R 开发者和咨询师，自我认定是 “Professional R Enthusiast”，其博客也是圈内重要技术博客之一。
- **Douglas Bates**：Rcpp 作者之一，Wisconsin-Madison 大学统计系荣休教授，bioconductor 创始人。
- **John Chambers**：S 语言的创造者，现为 Stanford 大学顾问教授，美国统计学会院士，R 语言核心团队成员之一。Chambers 于 1998 因 S 语言获得 ACM Software System Award，这是软件界最高奖项之一，1999 年授予 Apache Group，1995 年授予 World Wide Web。
- **JJ Allaire**：RStudio 创始人，著名的 ColdFusion 工具和 Windows Live Writer 也是其作品之一。

1.1.2.Rcpp 大事记

2005 年，Rcpp 作为 RQuantlib 的一部分出现，作者为 Dominick Samperi；

2006 年，Rcpp 在 CRAN 上发布，后更名为 RcppTemplate；

2008 年，Dirk 决定重写 Rcpp，并陆续发布新版本，老版本 API 作为 RcppClassic 继续开发维护；

2009 年，RcppTemplate 正式放弃维护，进入 CRAN 存档；

2009 年，Dirk 和 Franois 重新设计 Rcpp，并发布新版本；

2013 年，CRAN 中基于 Rcpp 开发的 package 已超过 100 个。

以上引文自*gossip R*

1.2. 我们的目标

Rcpp 的被大量应用在 R 组件包的开发中，但我们的目标是利用 Rcpp 对我们的代码进行局部优化，改进递归和循环的性能。故本节课主要会讲：

- Motivation：一个例子比较 R 与 Rcpp 的效率区别
- C++ 基本概念，简单程序撰写
- 如何实现 R 与 C++ 的桥接
- 一个实例

2. 环境配置

2.1.Rcpp 相关的扩展包：

```
install.packages("Rcpp")
install.packages("inline")# 简短代码的直接编译、链接和载入
install.packages("rbenchmark")# 运行时间比较和测试，非必须
install.packages("microbenchmark")#rbenchmark 的微秒版，精度更高
```

inline 包与 Rcpp 包的功能是相近的（至少对我们来说），由于参考书目 *Seamless R and C++ Integration with Rcpp*中示例主要使用 inline，我们将延用它，未来你可以依个人喜好选择。

2.2. 编译器

- Windows：需安装 Rtools 套件，可从 CRAN 网站获取。安装时需要勾选 “配置环境变量”（画重点）选项；
- OS X：从 app store 下载安装 Xcode
- Linux：sudo apt-get install r-base-dev

3. 动机：一个例子 -斐波那契数列

斐波那契数列：0,1,1,2,3,5,8,13,21,34……

3.1.R 与 Cpp 的效率比较

使用递归办法计算：

```
fibR <- function(n)
{
  if(n==0) return(0)
  if(n==1) return(1)
  else return(fibR(n-1)+fibR(n-2))
}
x <- 1:10
sapply(x,FUN = fibR)
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

```
# 斐波那契数列计算的 C 版本
library(Rcpp)
```

```
## Warning: package 'Rcpp' was built under R version 3.3.3
```

```
fibcode <- '
int fibonacci(const int x) {
  if(x==0) return(0);
  if(x==1) return(1);
  return( fibonacci(x-1)+fibonacci(x-2));
}'
cppFunction(fibcode)

# 经过字节编译的 Rcode
fibR.comp <- compiler::cmpfun(fibR)
```

```
library(rbenchmark)
benchmark(fibR(20),fibR.comp(20),fibonacci(20))
```

```
##           test replications elapsed relative user.self sys.self
## 3 fibonacci(20)          100    0.00       NA      0.00     0.00
## 1      fibR(20)          100   14.15       NA     13.89     0.03
## 2 fibR.comp(20)          100   14.54       NA     14.14     0.01
##   user.child sys.child
## 3         NA        NA
## 1         NA        NA
## 2         NA        NA
```

在这里先不必纠结 cppFunction 的具体用法，稍后详解。

另外，为了凸显 Rcpp 的强大效率，这里的函数利用了大量的递归。实际上，这段代码可以通过以下更优雅的方案实现:

3.2.Bonus：其它解决方案

1. 利用 “保存” 避免对相同值的重复运算；

```
mfibR <- local({
  memo <- c(1, 1, rep(NA, 1000))
  f <- function(x) {
    if (x == 0)
      return(0)
    if (x < 0)
      return(NA)
    if (x > length(memo))
      stop("x too big for implementation")
    if (!is.na(memo[x]))
      return(memo[x])
    ans <- f(x - 2) + f(x - 1)
    memo[x] <<- ans  # 通过环境变量实现递归函数中的信息记录
    ans
  }
})
```

2. 线性迭代（循环）

```
fibRiter <- function(n) {
  first <- 0
  second <- 1
  third <- 0
  for (i in seq_len(n)) {
    third <- first + second
    first <- second
    second <- third
  }
  return(first)
}
```

感兴趣的同学可以在本课程后自行练习这两种方案的 C++ 实现。

3.C++ 语法基础

该部分参考菜鸟教程，有兴趣的同学可自行前往钻研。

限于篇幅，这里不可能详尽所有内容，仅摘选我们 “可能” 用到的部分。

3.1. 程序结构

```
#include <iostream>
using namespace std;//命名空间，有点像 R 中 library 功能，无需写明来源

// main() 是程序开始执行的地方

int main()
{
  cout << ”Hello WiseRclub!”; // 输出 Hello WiseRclub!
  return 0;
}
```

C++ 语言定义了一些头文件，这些头文件包含了程序中必需的或有用的信息。上面这段程序中，包含了头文件。

using namespace std; 告诉编译器使用 std 命名空间。命名空间是 C++ 中一个相对新的概念。

下一行 // main() 是程序开始执行的地方是一个单行注释。单行注释以// 开头，在行末结束。

下一行 int main() 是主函数，程序从这里开始执行。

下一行 cout << “Hello WiseRclub!”; 会在屏幕上显示消息 “Hello WiseRclub!”。

下一行 return 0; 终止 main( ) 函数，并向调用进程返回值 0。

画重点：

- 在 C++ 中，分号是语句结束符。也就是说，每个语句必须以**分号**结束。它表明一个逻辑实体的结束。
- 块是一组使用大括号括起来的按逻辑连接的语句。
- C++ 标识符是用来标识变量、函数、类、模块，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。
- C++ 支持单行注释和多行注释。注释中的所有字符会被 C++ 编译器忽略。C++ 注释以/* 开始，以 */ 终止。

3.2. 数据类型

3.2.1.C++ 中的基本数据类型

类型	关键字	描述
布尔型	bool	存储值 true 或 false
字符型	char	通常是一个八位字节（一个字节）。这是一个整数类型
整型	int	对机器而言，整数的最自然的大小
浮点型	float	单精度浮点值
双浮点型	double	双精度浮点值
无类型	void	表示类型的缺失
宽字符型	wchar_t	宽字符类型

基本类型可以使用一个或多个类型修饰符进行修饰：signed，unsigned，short，long。

例如：

int 为 4 个字节，范围 -2147483648 到 2147483647；

long int 为 8 个字节，范围 -9,223,372,036,854,775,808 到 9,223,372,036,854,775,80；

unsigned long int 为 8 个字节，范围 0 到 18,446,744,073,709,551,615。

注意：变量的大小会根据编译器和所使用的电脑而有所不同。

3.2.2. 变量

除了基于 3.2.1. 中的基本类型，C++ 也允许定义各种其他类型的变量，比如枚举、指针、数组、引用、数据结构、类。

定义一个变量的方法是：

```
//type variable_list;
//type variable_name = value;

int    i, j, k;
char   c, ch;
float  f, salary;
double d;

extern int d = 3, f = 5;    // d 和 f 的声明
int d = 3, f = 5;           // 定义并初始化 d 和 f
byte z = 22;                // 定义并初始化 z
char x = 'x';               // 变量 x 的值为 'x'
```

变量声明和初始化可以同时进行也可以分开进行。

例：

```
#include <iostream>
using namespace std;

// 变量声明
extern int a, b;
extern int c;
extern float f;

int main ()
{
 // 变量定义
 int a, b;
 int c;
 float f;

 // 实际初始化
 a = 10;
 b = 20;
 c = a + b;

 cout << c << endl ;

 f = 70.0/3.0;
 cout << f << endl ;
```

```
 return 0;
}
```

C++ 变量作用域：

作用域是程序的一个区域，一般来说有三个地方可以声明变量：

- 在函数或一个代码块内部声明的变量，称为局部变量。
- 在函数参数的定义中声明的变量，称为形式参数。
- 在所有函数外部声明的变量，称为全局变量。

3.2.3. 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量。

常量可以是任何的基本数据类型，可分为整型数字、浮点数字、字符、字符串和布尔值。

常量的定义方法如下：

```
//使用关键字 const
//const type variable = value;

const int  LENGTH = 10;
const int  WIDTH  = 5;
const char NEWLINE = '\n';

//或使用预处理器 #define
//#define identifier value

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
```

3.2.4. 数组

数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

```
//一维数组
//type arrayName [ arraySize ];

double balance[10];//仅声明

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

注意 C++ 的计数习惯从 0 开始，上例中 balance[2] 应为 3.4。

```
//多维数组
//type name[size1][size2]...[sizeN];

int a[3][4] = {
 {0, 1, 2, 3} ,   /*  初始化索引号为 0 的行 */
 {4, 5, 6, 7} ,   /*  初始化索引号为 1 的行 */
 {8, 9, 10, 11}   /*  初始化索引号为 2 的行 */
};
```

3.3.5. 指针与引用

每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址

```
#include <iostream>

using namespace std;

int main ()
{
  int  var1;
  char var2[10];

  cout << "var1 变量的地址： ";
  cout << &var1 << endl;

  cout << "var2 变量的地址： ";
  cout << &var2 << endl;

  return 0;
}

/* 上面的例子应输出：
var1 变量的地址： 0xbfebd5c0
var2 变量的地址： 0xbfebd5b6
*/
```

指针是一个变量，其值为另一个变量的地址。

```
//type *var-name;

int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch;    /* 一个字符型的指针 */

//e.g.

#include <iostream>

using namespace std;

int main ()
{
  int  var = 20;   // 实际变量的声明
  int  *ip;        // 指针变量的声明

  ip = &var;       // 在指针变量中存储 var 的地址

  cout << "Value of var variable: ";
  cout << var << endl;

  // 输出在指针变量中存储的地址
  cout << "Address stored in ip variable: ";
  cout << ip << endl;
```

```
  // 访问指针中地址的值
  cout << "Value of *ip variable: ";
  cout << *ip << endl;

  return 0;
}
```

指针是可以进行加减运算的。前面讲过的数组与指针有着密切联系：数组的名字其实是一个指针，如：

```
int array[5]={1,2,3,4,5};

//array 是 1 的地址；*array 相当于 array[0]
//array[2] 则相当于 *（array+2），访问到 3
```

引用与指针非常相像，你可以看作是“受保护的指针”。一旦确定初始化引用使其指向一个对象，则不能再更改使其指向别的对象。

```
int&    r = i;
//创建一个整型引用 r 指向 i，r 可看作指向 i 的固定指针
```

3.3.6. 结构体

结构是 C++ 中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

```
struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

3.3.C++ 运算符

包括**算数运算符**、**关系运算符**、**逻辑运算符**、**位运算符**、**赋值运算符**和其他运算符。

常见且与 R 中较为不同的有：

运算符	描述	实例
++	自增运算符，整数值增加 1	A++ 将得到 11
–	自减运算符，整数值减少 1	A– 将得到 9
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A

3.4. 循环体与判断

循环即条件判断结构基本与 R 一致, 注意 for 中分号的使用。

```
for ( init; condition; increment )
 //注意分号的使用
{
  statement(s);
}

//for 也可以实现死循环，利用条件判断 +break 跳出
for ( ; ;  )
{
  statement(s);
}
```

3.5 函数

```
/*

return_type function_name( parameter list )
{
  body of the function
}
 */
```

3.5.1. 函数定义的一个实例

```
//e.g. 一个实例
#include <iostream>
using namespace std;

// 函数声明
int max(int num1, int num2);

int main ()
{
  // 局部变量声明
  int a = 100;
  int b = 200;
  int ret;

  // 调用函数来获取最大值
  ret = max(a, b);

  cout << ”Max value is : ” << ret << endl;

  return 0;
}

// 函数返回两个数中较大的那个数
int max(int num1, int num2)
{
  // 局部变量声明
  int result;
```

```
  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

与声明 & 初始化变量类似的，函数的声明与定义也是可以分离的。

3.5.2. 传值、传指针、传引用

调用函数时有三种传递参数的办法:

1. 传值调用，将参数的实际值复制给函数的形式参数。修改函数内的形式参数对实际参数没有影响。

2. 指针调用，把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

3. 引用调用，该方法把参数的引用复制给形式参数。在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

注意，C++ 不允许传递完整的数组给函数（因为当数组维度很大时，这么做是相当低效的），当你试图这么做时，总是向函数传递了一个指针。

3.6. 面向对象编程

面向对象编程的概念有助于我们理解 Rcpp 的机制，所以这里简要介绍一下 OOP 理念。

3.6.1. 类与对象

```
#include <iostream>

using namespace std;

class Box
//使用关键字 class 定义新的类
{
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );

  private:
    double width;
};
/* 访问修饰符：
public：公有成员，可被外部访问
protected：被保护成员，可被派生类访问
private：只可被内部成员访问 */
```

```
// 成员函数定义 使用范围解析运算符 ::
double Box::getWidth(void)
{
   return width ;
}

void Box::setWidth( double wid )
{
   width = wid;
}

// 程序的主函数
int main( )
{
  Box box;

  // 不使用成员函数设置长度
  box.length = 10.0; // OK: 因为 length 是公有的
  cout << "Length of box : " << box.length <<endl;

  // 不使用成员函数设置宽度
  // box.width = 10.0; // Error: 因为 width 是私有的
  box.setWidth(10.0);  // 使用成员函数设置宽度
  cout << "Width of box : " << box.getWidth() <<endl;

  return 0;
}
```

3.6.2. 继承

一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。

子类会继承 public 和 protected 成员，并且可以定义新的成员

```
#include <iostream>

using namespace std;

// 基类
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
```

```
    int height;
};

// 派生类 使用：指定基类，public 是继承类型（public、protected、private，具体不表）
class Rectangle: public Shape
{
  public:
    int getArea()
    {
      return (width * height);
    }
};//无需另外说明，继承基类的 width 和 height

int main(void)
{
  Rectangle Rect;

  Rect.setWidth(5);//setWidth 是我们在基类中定义的，被继承了
  Rect.setHeight(7);

  // 输出对象的面积
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
//实际上可以指定派生自多个基类
```

3.6.3. 多态

```
#include <iostream>
using namespace std;

//定义基类

class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    virtual int area() //注意这里的关键字 virtual，基类中我们定义了一个虚函数 area
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};

//定义两个派生类
class Rectangle: public Shape{
```

```
  public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};
class Triangle: public Shape{
  public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
//注意两个派生类中各有一个 area 函数，当主程序调用 area 这个函数时，会根据当前指针指向的派生类运行类中函数 area

// 程序的主函数
int main( )
{
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);

  // 存储矩形的地址
  shape = &rec;
  // 调用矩形的求面积函数 area
  shape->area();

  // 存储三角形的地址
  shape = &tri;
  // 调用三角形的求面积函数 area
  shape->area();

  return 0;
}

/*
Rectangle class area
Triangle class area
*/
```

3.6.4. 重载函数与重载运算符

在一个作用域内，可以定义多个同名函数，但其接受的参数类型（或至少是顺序）不同，称为重载函数。在调用重载函数时，会根据传入参数的数据类型运行相应代码。

```
#include <iostream>
using namespace std;

class printData
```

```
{
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }

    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }

    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};//在 printData 类中定义了三个重载函数

int main(void)
{
  printData pd;

  // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);
  // Call print to print character
  pd.print("Hello C++");

  return 0;
}
```

3.7. 抽象类

拥有纯虚函数的类称为抽象类，设计抽象类（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为接口使用。可用于实例化对象的类被称为具体类。

面向对象的系统可能会使用一个抽象基类为所有的外部应用程序提供一个适当的、通用的、标准化的接口。然后，派生类通过继承抽象基类，就把所有类似的操作都继承下来。

外部应用程序提供的功能（即公有函数）在抽象基类中是以纯虚函数的形式存在的。这些纯虚函数在相应的派生类中被实现。

重点：数据抽象的设计思路为外部程序提供了接口！

3.8. 模板函数与模板类

模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码。

```
//模板函数
/*
template <class type> ret-type func-name(parameter list)
{
  // 函数的主体
}*/
```

```
#include <iostream>
#include <string>

using namespace std;


//定义一个模板函数 T 指代的是数据类型
template <class T>
inline T const& Max (T const& a, T const& b)
{
   return a < b ? b:a;
}


int main ()
{

   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;
}


//模板类
/*
template <class type> class class-name {
.
.//定义类成员
.
}*/

#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;     // 元素
```

```
 public:
   void push(T const&);  // 入栈
   void pop();               // 出栈
   T top() const;            // 返回栈顶元素
   bool empty() const{       // 如果为空则返回真。
      return elems.empty();
   }
};

template <class T>
void Stack<T>::push (T const& elem)
{
   // 追加传入元素的副本
   elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop(): empty stack");
   }
   // 删除最后一个元素
   elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
   if (elems.empty()) {
      throw out_of_range("Stack<>::top(): empty stack");
   }
   // 返回最后一个元素的副本
   return elems.back();
}

int main()
{
   try {
      Stack<int>         intStack;  // int 类型的栈
      Stack<string> stringStack;    // string 类型的栈

      // 操作 int 类型的栈
      intStack.push(7);
      cout << intStack.top() <<endl;

      // 操作 string 类型的栈
      stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   }
   catch (exception const& ex) {
```

```
    cerr << "Exception: " << ex.what() <<endl;
    return -1;
  }
}
```

所谓的 C++，其实就是 C 语言 + 面向对象编程 +STL

STL：标准模版库，里面存放了很多的模版函数和模版类

所以 C++ 的标准库应该包括：C 的标准结构和标准库 + C++ 标准库 + STL

4.Rcpp

4.1. 如何在 R 中调用 C++ 脚本

运行一个 C++ 脚本（假设名为 Wiserclub.cpp），只需在该脚本所在文件夹打开命令提示符 CMD，键入 g++ Wiserclub.cpp

在这个过程中其实完成了两个动作，我们也可以使用 CMD 分开来做：

1. **g++ Wiserclub.cpp -o Wiserclub** 将源文件进行编译，生成可执行文件；
2. **./Wiserclub** 运行可执行文件。

要在 R 中调用，你必须：

1. 设置环境变量说明头文件和库文件位置
2. 使用编译器 g++ 进行编译 (*.o 文件) 并将其链接为一个共享库 (*.so 文件)，so 文件可供我们载入 R
3. 接着我们就可以加载共享库中的目标文件，在 R 中使用.Call 调用

同时，你还应注意，只有符合规范的输入输出对象（i.e. 通过适合 R 的 API 接口输入输出，使 C 能够识别你的输入，R 能够识别你的输出），才能在 R 中正常调用 Cpp 程序。

这一切，贴心的 Rcpp 都会帮你完成！

4.2. 认识 Rcpp

4.2.1.Rcpp 指什么

通常一个扩展包的主体部分，就是由许多 R 脚本（许多函数）组成的。

但我们提到 Rcpp 时，有两层语义：

- R 的扩展包 Rcpp 包含的 R 函数
- Rcpp 扩展包提供的大量预编译的库，e.g.C++ 库 <Rcpp.h>，内含大量的类、模版、重载函数等。该库为 R User 书写 C++ 代码提供大量便利。

4.2.2.Rcpp 和 inline 的使用

现阶段我们会用到的函数：

```
library(Rcpp)

#?evalCpp# 直接编译、运行简短 C 代码并返回结果

c_code <- 'std::sqrt(2.0)'
evalCpp(code=c_code)
```

```
## [1] 1.414214
```

```
evalCpp( "std::numeric_limits<double>::max()" )
```

```
## [1] 1.797693e+308
```

```
evalCpp( "std::numeric_limits<int>::max()" )
```

```
## [1] 2147483647
```

```
# 用于单句程序，无需添加分号
```

```
#?cppFunction# 用于编译一个 C++ 函数并自动完成链接工作
cppFunction('
int add(int x,int y,int z){
  int sum=x+y+z;
  return sum;
}')
add
```

```
## function (x, y, z)
## .Primitive(".Call")(<pointer: 0x0000000065041580>, x, y, z)
```

```
#.Call 和.C 是 R 标准的 API 接口，add 函数使用调用.Call(某指针)
# 指针指向 C++ 程序所在位置。
add(1,2,3)
```

```
## [1] 6
```

```
#?sourceCpp# 编译链接一个 Cpp 脚本
setwd("C:/Users/River/Desktop/2016 研一下/club 课程/Rcpp learning")
sourceCpp("rcpptest.cpp")
```

```
##
## > library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 3.3.3
```

```
##
## > x = runif(1e+05)
##
## > microbenchmark(mean(x), meanC(x))
## Unit: microseconds
##      expr     min       lq     mean  median      uq     max neval cld
##   mean(x) 518.088 520.9615 536.5908 525.888 536.357 857.185   100   b
##  meanC(x) 254.118 256.9920 263.6300 259.045 262.739 414.224   100  a
```

```
library(inline)
```

```
## Warning: package 'inline' was built under R version 3.3.3
```

```
##
## Attaching package: 'inline'
```

```
## The following object is masked from 'package:Rcpp':
##
##     registerPlugin
```

```
#?cxxfunction# 功能、参数、用法都基本与上面的 cppFunction 很像，但仍有略微区别


src <- '
```

```
  Rcpp::NumericVector xa(a);
  Rcpp::NumericVector xb(b);
  int n_xa=xa.size(), n_xb=xb.size();
  Rcpp::NumericVector xab(n_xa+n_xb-1);
  for(int i=0;i<n_xa;i++)
    for(int j=0;j<n_xb;j++)
      xab[i+j]+=xa[i]*xb[j];
  return xab;
'
myfun <- cxxfunction(signature(a="numeric",b="numeric"),body=src,plugin="Rcpp",verbose=T)

##  >> setting environment variables:
## PKG_LIBS =
##
##  >> LinkingTo : Rcpp
## CLINK_CPPFLAGS =  -I"C:/Users/River/Documents/R/win-library/3.3/Rcpp/include"
##
##  >> Program source :
##
##    1 :
##    2 : // includes from the plugin
##    3 :
##    4 : #include <Rcpp.h>
##    5 :
##    6 :
##    7 : #ifndef BEGIN_RCPP
##    8 : #define BEGIN_RCPP
##    9 : #endif
##   10 :
##   11 : #ifndef END_RCPP
##   12 : #define END_RCPP
##   13 : #endif
##   14 :
##   15 : using namespace Rcpp;
##   16 :
##   17 :
##   18 : // user includes
##   19 :
##   20 :
##   21 : // declarations
##   22 : extern "C" {
##   23 : SEXP file268c2fe85347( SEXP a, SEXP b) ;
##   24 : }
##   25 :
##   26 : // definition
##   27 :
##   28 : SEXP file268c2fe85347( SEXP a, SEXP b ){
##   29 : BEGIN_RCPP
##   30 :
##   31 :   Rcpp::NumericVector xa(a);
##   32 :   Rcpp::NumericVector xb(b);
##   33 :   int n_xa=xa.size(), n_xb=xb.size();
##   34 :   Rcpp::NumericVector xab(n_xa+n_xb-1);
##   35 :   for(int i=0;i<n_xa;i++)
```

```
##   36 :     for(int j=0;j<n_xb;j++)
##   37 :       xab[i+j]+=xa[i]*xb[j];
##   38 :   return xab;
##   39 :
##   40 : END_RCPP
##   41 : }
##   42 :
##   43 :
## Compilation argument:
##  C:/PROGRA~1/R/R-33~1.2/bin/x64/R CMD SHLIB file268c2fe85347.cpp 2> file268c2fe85347.cpp.err.txt
```

```
myfun(1:4,2:5)
```

```
## [1]  2  7 16 30 34 31 20
```

```
#cxxfunction 分开构建函数头 signature 和函数主体 body,
#plugin 设置头文件
#verbose=T 输出 cpp 临时文件细节
# 如果你需要防止 main 函数之外的函数（不需要 export 的部分），使用参数 include
# 使用 inline::cxxfunction 需要对输入项进行转换
```

```
cppFunction('
int add(int x,int y,int z){
  int sum=x+y+z;
  return sum;
}',verbose=T,rebuild=T)
```

```
##
## Generated code for function definition:
## --------------------------------------------------------
##
## #include <Rcpp.h>
##
## using namespace Rcpp;
##
## // [[Rcpp::export]]
##
## int add(int x,int y,int z){
##   int sum=x+y+z;
##   return sum;
## }
##
## Generated extern "C" functions
## --------------------------------------------------------
##
##
## #include <Rcpp.h>
## // add
## int add(int x, int y, int z);
## RcppExport SEXP sourceCpp_9_add(SEXP xSEXP, SEXP ySEXP, SEXP zSEXP) {
## BEGIN_RCPP
##     Rcpp::RObject rcpp_result_gen;
##     Rcpp::RNGScope rcpp_rngScope_gen;
##     Rcpp::traits::input_parameter< int >::type x(xSEXP);
##     Rcpp::traits::input_parameter< int >::type y(ySEXP);
##     Rcpp::traits::input_parameter< int >::type z(zSEXP);
```

```
##     rcpp_result_gen = Rcpp::wrap(add(x, y, z));
##     return rcpp_result_gen;
## END_RCPP
## }
##
## Generated R functions
## -------------------------------------------------------
##
## '.sourceCpp_9_DLLInfo' <- dyn.load('C:/Users/River/AppData/Local/Temp/Rtmpk5n2VO/sourceCpp-x86_64-w64-mingw32-0.12.10/so
##
## add <- Rcpp:::sourceCppFunction(function(x, y, z) {}, FALSE, '.sourceCpp_9_DLLInfo', 'sourceCpp_9_add')
##
## rm('.sourceCpp_9_DLLInfo')
##
## Building shared library
## -------------------------------------------------------
##
## DIR: C:/Users/River/AppData/Local/Temp/Rtmpk5n2VO/sourceCpp-x86_64-w64-mingw32-0.12.10/sourcecpp_268c76bd16b5
##
## C:/PROGRA~1/R/R-33~1.2/bin/x64/R CMD SHLIB -o "sourceCpp_13.dll" --preclean "" "file268c717762cd.cpp"
```

```
#rebuild=T 强制重建函数与链接，默认为 F 会在链接存在时提示"无需重建"
```

Rcpp::cppFunction 比 cxxfunction 多做了一层封装（Rcpp::traits::input_parameter::type x(xSEXP)，使用起来更方便。

而 inline::cxxfunction 能让我们更清楚的了解 R 与 C++ 如何进行数据交换。

其次：BEGIN_RCPP 与 END_RCPP 是两个宏，它们的作用保证了 Rcpp 的异常处理机制，使你无需担心无从监控 C++ 程序运行时可能发生的错误。

try{code chunk}；Catch(error_1){solution_1};Catch(error_2){solution_2};

BEGIN_RCPP：try{
END_RCPP: }；Catch(error_1){.....

它的作用类似于

- Python 中的 try:…;except:…
- R 中的 tryCatch();try() 函数

Rcpp 的两个宏，当程序中间发生错误时，C++ 丢出（throw）一个错误信息，随后使用 Catch，如果接住的是 C++ 标准错误类型，将其转换为 R 可识别的信息回传给 R；如果接住的是 C++ 非标准错误类型，则返回信息提示 "未知错误"。有关 C++ 标准错误类型参考菜鸟教程，内有介绍。

```
add(1,2,"it will throw an error")
```

```
## Error in add(1, 2, "it will throw an error"): not compatible with requested type
```

4.3.Rcpp 核心数据类型

哪一些 R 对象可以输入到 C++ 中呢？我们前面已经提过，抽象类和其派生的具体类是 C++ 与外部应用程序交互的关键，C++ 中提供什么样的类（用以包装 R 对象），我们就能使用什么样的 R 对象。

我将他们总结为：

- 两个模板函数：as<> 和 wrap<>
- 一个基类：RObject
- 若干派生类：

派生类	对应 R 中对象
IntegerVector	integer 型向量
NumericVector	numeric 型向量
LogicalVector	logical 型向量
CharacterVector	character 型向量
GenericVector	List 类型泛型向量
ExpressionVector	expression 型向量
RawVector	raw 字节数据
NumericMatrix	Numeric 型矩阵
Named（辅助类）	为向量或 list 创建标识标签
DataFrame	dataframe 泛型向量
Function	R 中的 function

其它还有 environment 类、S4 类、referenceClasses 类

4.3.1.R 中的对象

每个 RObject 类实例都封装了一个 R 对象，每个 R 对象都是一个 SEXP：指向 “S 表达式” 的**指针**。

这个类包含了通用的几个成员函数，适用于所有 RObject 及其派生类：isNULL、isObject、isS4 等

但我们通常不会直接使用到基类 RObject。

4.3.2.as<> 和 wrap<>

as<> 函数用于将 SEXP 转化为 C++ 中类型，例如 as(欲转化变量):sexp->int

wrap() 用于将 C++ 类型包装为 sexp，使用时只需：wrap(欲转化的变量)

```
incltxt <- '
int fibonacci(const int x)
{
 if(x==0) return(0);
 if(x==1) return(1);
 return fibonacci(x-1)+fibonacci(x-2);
}'
body <- '
 int x = Rcpp::as<int>(xs);
 return Rcpp::wrap(fibonacci(x));
'
fibRcpp <- cxxfunction(signature(xs="int"),plugin = "Rcpp",
                       body=body,includes = incltxt)
fibRcpp(10)
```

```
## [1] 55
```

实际上在 Rcpp 包中这两个函数会被 “隐式” 调用（inline 包仍需我们手动处理），通常不需费心。

但这两个函数也适用于从 NumericVector 等派生类向 C++Vector 的转化，后面仍有可能遇到。

4.3.3. 派生类

IntegerVector

```
# 这个例子用来求连乘积
src <- '
  Rcpp::IntegerVector x(vx);
  int prod=std::accumulate(x.begin(),x.end(),1,std::multiplies<int>());
  return Rcpp::wrap(prod);'
fun <- cxxfunction(signature(vx="integer"),body=src,plugin = "Rcpp")
fun(1:10)
```

```
## [1] 3628800
```

NumericVector

```
# 这个例子用来对向量求 log

src <- '
Rcpp::NumericVector invec(vx);
Rcpp::NumericVector outvec(vx);
  for (int i=0;i<invec.size();i++)
    {outvec[i]=log(invec[i]);}
return outvec;'

fun <- cxxfunction(signature(vx="Numeric"),body=src,plugin = "Rcpp")
x <- seq(1.0,10.0,by=1.0)
fun(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

```
x
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

```
# 为什么 x 也改变了？因为通过接口传出的 sexp 是指向变量 x 的"指针"！
#invec 和 outvec 都对 sexp 进行了封装，但本质上它们指向同一个内存地址，无论是对 outvec 还是 invec 进行修改，x 都发生了变化

# 我们对上面的代码做一点修改使它能够正常工作

src <- '
Rcpp::NumericVector invec(vx);
Rcpp::NumericVector outvec=Rcpp::clone(vx);
  for (int i=0;i<invec.size();i++)
    {outvec[i]=log(invec[i]);}
return outvec;'

fun <- cxxfunction(signature(vx="Numeric"),body=src,plugin = "Rcpp")
x <- seq(1.0,10.0,by=1.0)
fun(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

```
x
```

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

```
#Rcpp::clone 拷贝了 vx 指向的内容。
```

NumericMatrix

```
# 这个例子对矩阵中每一个元素做开方
# 声明矩阵的两种方式
#Rcpp::NumericVector vec3=
#  Rcpp::NumericVector(Rcpp::Dimension(3,3));

# 或者
src <- '
Rcpp::NumericMatrix mat=Rcpp::clone<NumericMatrix>(mx);
std::transform(mat.begin(),mat.end(),mat.begin(),::sqrt);
return mat;'
fun <- cxxfunction(signature(mx="Numeric"),body=src,plugin = "Rcpp")
mx <- matrix(1.3:9.3,3,3)
fun(mx)
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.140175 2.073644 2.701851
## [2,] 1.516575 2.302173 2.880972
## [3,] 1.816590 2.509980 3.049590
```

```
# 补充：对于多维数组，需要用第一种方式进行定义
```

LogicalVector

```
src <- '
LogicalVector fun()
{
  LogicalVector v(6);
  v[0]=v[1]=false;
  v[2]=true;
  v[3]=R_NaN;
  v[4]=R_PosInf;
  v[5]=NA_REAL;
  return v;
}'
fun <- cppFunction(src)
fun()
```

```
## [1] FALSE FALSE  TRUE    NA    NA    NA
```

```
#NaN,Inf,NA 在逻辑向量中被当作 NA 处理
```

CharacterVector

```
src <- '
CharacterVector fun()
{
  Rcpp::CharacterVector v(3);
  v[0]=" 听困了吗?";
  v[1]=" 快醒醒";
  v[2]=R_NaString;
  return v;
}'
fun <- cppFunction(src)
```

```
fun()
```

```
## [1] ”听困了吗?” ”快醒醒”  NA
```

Named

Named 类是一个辅助类，用来设定键值对的键名。比如说：

```
NamedVector <- c(these=1,elements=2,have=3,names=4)
NamedVector
```

```
##    these elements     have    names
##       1        2        3        4
```

这个命名的工作放到 C++ 中就要依靠 Named 类来完成

```
src <- ’
NumericVector fun(double x,double y,double z,double anyway)
{
 NumericVector xx=NumericVector::create(
  Named(”these”)=x,
  Named(”elements”)=y,
  Named(”have”)=z,
  Named(”names”)=anyway
 );
 return xx;
}’
# 上面的书写中省略了 ”Rcpp::”, 因为 Rcpp 包函数会自动添加
#using Namespace Rcpp;
fun <- cppFunction(src)
fun(2,3,4,5)
```

```
##    these elements     have    names
##       2        3        4        5
```

```
# 上面的书写太繁琐了，其实可以简化为 _[”names”] 形式
src <- ’
NumericVector fun(double x,double y,double z,double anyway)
{
 NumericVector xx=NumericVector::create(
  _[”these”]=x,
  _[”elements”]=y,
  _[”have”]=z,
  _[”names”]=anyway
 );
 return xx;
}’
```

GenericVector(List)

注意，如果你想要用 list 作为输入，而对其中的数据进行操作，你需要使用 as<> 将其一一转化为合适的类型。

```
alist <- list(list1=2,list2=c(3,4,5),list3=c(6,7,8))
alist
```

```
## $list1
## [1] 2
```

```
##
## $list2
## [1] 3 4 5
##
## $list3
## [1] 6 7 8
```

```
# 假设一个 list[3]
# 对 list[1] 开方，对 list[2] 求平均，对 list[3] 逐项平方，返回 list
src <- '
List fun(List alist)
{
  double list1=as<double>(alist["list1"]);
  NumericVector list2=as<NumericVector>(alist["list2"]);
  NumericVector list3=clone(as<NumericVector>(alist["list3"]));
  list1=std::sqrt(list1);
  double mean_list2=mean(list2);
  list3=pow(list3,2);
  List list_re=List::create(
  _["sqrt"]=list1,_["mean"]=mean_list2,_["power2"]=list3
  );
  return list_re;
}'
cppFunction(src)
fun(alist)
```

```
## $sqrt
## [1] 1.414214
##
## $mean
## [1] 4
##
## $power2
## [1] 36 49 64
```

DataFrame 类

DataFrame 的处理方式几乎与 List 相同

```
src <- '
DataFrame fun()
{
  NumericVector x=NumericVector::create(1,2,3,4,5);
  std::vector<std::string> s(5);
  s[0]=" 这节课真的好多内容";
  s[1]=" 我有什么办法";
  s[2]=" 我也很绝望啊";
  s[3]=" 希望大家下学期";
  s[4]=" 继续支持 Wiserclub";
  DataFrame df=DataFrame::create(_["a"]=x,_["b"]=s);
  return df;
}'
cppFunction(src)
fun()
```

```
##   a                b
```

```
## 1 1 这节课真的好多内容
## 2 2      我有什么办法
## 3 3      我也很绝望啊
## 4 4    希望大家下学期
## 5 5  继续支持Wiserclub
```

Function

```
#Rcpp 允许我们将 function 作为一个参数输入
src <- '
Function sort(x);
return sort(y,Named("decreasing",true));
'
fun <- cxxfunction(sig=signature(x="function",y="ANY"),
                   src,plugin = "Rcpp")
fun(sort,sample(1:5,10,TRUE))
```

```
##  [1] 5 5 4 4 4 3 1 1 1 1
```

```
fun(sort,sample(LETTERS[1:5],10,TRUE))
```

```
##  [1] "E" "E" "D" "D" "D" "D" "C" "B" "A" "A"
```

```
#Rcpp 也允许我们直接访问 R 函数
src <- '
RNGScope scp;
Rcpp::Function rt("rt");
return rt(5,3);'
#Function 类封装了 R 中名为 rt 的函数（的指针）
#RNGScope scp 初始化随机种子
fun <- cxxfunction(sig=signature(),src,plugin = "Rcpp")
fun()
```

```
## [1] -0.3740201  0.7616664 -0.9409260  0.2675390 -2.5029594
```

其他类

除了上面的派生类以外，还有 Environment,S4,ReferenceClasses,rawVector，望各位自行探索。寡人写课件要吐血了 orz。

4.4. 语法糖

语法糖 (Syntactic sugar), 它意指那些没有给计算机语言添加新功能，而只是对人类来说更 “甜蜜” 的语法。语法糖往往给程序员提供了更实用的编码方式，有益于更好的编码风格，更易读。不过其并没有给语言添加什么新东西。

举个例子：在 C 语言里用 a[i] 表示 *(a+i), 用 a[i][j] 表示 *(*(a+i)+j)

Rcpp 为我们提供了大量的语法糖，如：+、-、*、/，其本质是重载函数，可直接用于向量操作，完美贴合你在 R 中的编程习惯，有了这些语法糖，Rcpp 才可称名副其实的 “无缝链接”！

- **二元运算符**：+,-,*,/,<,>,<=,>=,!=,==;
- **一元运算符**：-（取负）,!;
- **函数**：all,any,is_na,seq_along,seq_len,pmax,pmin,ifelse,sapply（居然连这个也有！惊不惊喜意不意外）,lapply,mapply,sign,diff,setdiff,union_（下划线 "_" 为与 C++ 关键字 union 区分）,intersect,clamp,unique,sort_unique,table,duplicated
- **数学函数**：abs,exp,floor,ceil,pow,log,log10,sqrt,sin,cos,tan,sinh,cosh,tanh,asin,acos,atan,gamma…

- **dqpr 概率分布统计函数**：dnorm,pnorm,qnorm,rnorm…

其实前面讲解 list 派生类的例子中我已经偷偷使用了……

5. 一个实例

例自 RcppArmadillo 扩展包下的 fastLM.cpp，用于操作大规模运算的线性回归。

```
src <- '
  Rcpp::NumericVector yr(ys);
  Rcpp::NumericMatrix Xr(Xs);
  int n=Xr.nrow(),k=Xr.ncol();
  arma::mat X(Xr.begin(),n,k,false);
  arma::colvec y(yr.begin(),yr.size(),false);
  arma::colvec coef=arma::solve(X,y);// fit y~X
  arma::colvec res=y-X*coef;//residuals
  double s2=std::inner_product(res.begin(),res.end(),res.begin(),double())/(n-k);
  arma::colvec se=arma::sqrt(s2*
              arma::diagvec(arma::inv(arma::trans(X)*X)));
  return Rcpp::List::create(Rcpp::Named("coef")=coef,
                        Rcpp::Named("se")=se,
                        Rcpp::Named("df")=n-k);
'
fLm <- cxxfunction(sig=signature(ys="numeric",Xs="numeric"),
                 body=src,plugin="RcppArmadillo")

# 测试性能
y <- log(trees$Volume)
x <- cbind(1,log(trees$Girth))

microbenchmark(fLm(y,x),lm.fit(x,y))
```

```
## Unit: microseconds
##         expr     min      lq      mean  median       uq     max neval cld
##    fLm(y, x)  27.095  28.327  39.27181  40.643  41.4640 378.097   100  a
##  lm.fit(x, y) 122.338 125.622 141.01725 135.475 139.3755 310.771   100   b
```

注意：不要用这个例子替代 lm 函数，除非你保证数据适用于 OLS。

Armadillo 对 lm 的重现依赖于 QR 分解，传统数值计算库通过列主元方法来处理不满秩矩阵的 QR 分解；而 R 中选择删除不满秩矩阵中的若干列以保障结果的统计意义。lm 尽管性能较低，但其通过链接至 Linpack 库定制解决各种如共线性等问题的方案，而 Armadillo 的传统线性代数库中不含有这些方法。

我们学习的 Rcpp 仍是浅层面的，更适用于你的自定义函数，在必须使用递归或循环时能够有效提升代码性能。但不要总是尝试重写 R 扩展包中的函数，尤其是统计学方法的函数，这些扩展包背后通常都有论文理论的支持，对各种异常情况提供了合适的解决方案。记住：R 是统计学家的创造物！

6. 参考

- C++ 教程：www.runoob.com/cplusplus/cpp-tutorial.html
- STL 简短教程：http://net.pku.edu.cn/~yhf/UsingSTL.htm
- STL 模版库查询：https://www.sgi.com/tech/stl/index.html
- Rcpp 书籍：advanced R

- Rcpp 书籍：Seamless R and C++ Integation with Rcpp