CS205　C/C++ Program Design　Assignment4

Name：任艺伟　ID NUM：11912714

Code Part ：https://github.com/MVP-D77/Assignment4-Operator

Part1 Description：

GitHub 上共包含 3 部分代码，Matrix.cpp Matrix.hpp(头文件) main.cpp 和 CMakeLists.txt 文件。第一部分：实现了对于矩阵 Class 的定义以及 constructor，copy constructor，析构函数，运算符重写(+,-,*,~(矩阵求逆),<<,>>,==,!=以及 friend 函数 *的多种情况)；同时给出了矩阵求逆，矩阵行列式, 矩阵转置的函数接口可以使用；第二部分：对于核心关键问题, Matrix class 中含有指针(用一维向量模拟二维矩阵存储 float 元素), 在 constructor 中需要 new 开内存空间，在析构函数中需要 delete 回收空间，防止内存泄漏，在后面部分研究了单线程, 多线程时对于该问题的处理, 主要采用 mutex 互斥对象, 加 lock 及 atomic_int 指针原子操作，实现公用同一指针，以防止 Matrix 元素过多时，memoryCopy 占用过多内存问题；体会 Matrix Class 继承，针对有时需要矩阵*向量，继承 Matrix 实现了行/列向量，对比正常新建 Vector Class 节省了很多逻辑和代码量；最后在 arm 板上运行代码测试，结果正确!

Part2 Result && Verification：

1、CmakeLists.txt 文件：

```
cmake_minimum_required(VERSION 3.17)
project(Assignment4)
find_package (Threads)

set(CMAKE_CXX_STANDARD 14)
aux_source_directory(. DIR_SRCS)

add_executable(Assignment4 ${DIR_SRCS})
#add_executable(Assignment4 main.cpp Matrix.cpp)
target_link_libraries (Assignment4 ${CMAKE_THREAD_LIBS_INIT})
```

```
renyiweideMacBook-Pro:Assignment4 evelynryw$ cmake .
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/evelynryw/CLionProjects/Assignment4
renyiweideMacBook-Pro:Assignment4 evelynryw$ make
[100%] Built target Assignment4
renyiweideMacBook-Pro:Assignment4 evelynryw$ ./Assignment4
Please input the row number of  matrix:
```

第一张为手写的 CMake 文件，定义了最低使用该文件的版本，C++版本，以及项目名称，由于后面用到 thread 库，在这里导入了该库，链接到 Project 中，并指定 make 之后生成的可执行文件即为 Assignment4。下图为执行，cmake .执行当前目录下的 CMakeList.txt，其会自动生成 MakeFile。make 命令执行，生成可执行文件运行，进入程序。

2、Matrix Class 定义以及各种运算符重写测试。

```
class Matrix {
private:
    int rowNumber;
    int columnNumber;
    float * valueItem;
    atomic_int * counter;

public:
    Matrix();
```

Matrix 中定义四个成员变量，均为 private，分别对应矩阵的行数，列数，float 数据，int 原子指针（后面详细讲述）。其中元素为一维数组模拟存储，运算时，通过行数列数找到真正对应的位置，如 Part1 中介绍，实现了诸多方法和操作符重写，以使得其可以通过+，-，*等运算符，<<,>>等输入输出符进行操作。

2.1、Constructor

```
Matrix::Matrix() :rowNumber(0),columnNumber(0),valueItem(nullptr){
//    counter = new int;
    counter = new atomic_int;
    *(counter) = 0;
}

Matrix::Matrix(int rowNumber, int columnNumber,float * value) {
    this->columnNumber = columnNumber;
    this->rowNumber = rowNumber;
    this->valueItem = new float [rowNumber*columnNumber]();
    if(value!=NULL){
        for(int i=0;i<rowNumber*columnNumber;i++){
            *(this->valueItem+i) = *(value+i);
        }
    }
//    counter = new int;

    counter = new atomic_int;
    *(counter) = 1;
}
```

提供有参数和无参数两个构造器，无参数的直接默认为 0 矩阵，有参数的进行赋值，在这里要求新建对象时传入行，列，value 数组，为了便于统一，在 constructor 里赋值时进行了 memoryCopy，也可以改进为全部都公用传入的数组。counter 为指针，对应地址的数据记录当前有多少 Matrix 对象公用同一个 valueItem 指针存储数据。

```
float matrixValue[10] = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0};
Matrix matrix1( rowNumber: 2, columnNumber: 5,matrixValue);
cout<<"Matrix constructor : "<<endl;
cout<<matrix1<<endl;
```

```
Matrix constructor :
1.000 1.000 1.000 1.000 1.000
1.000 1.000 1.000 1.000 1.000
```

2.2 Copy Constructor && “=” operator ：

```
Matrix::Matrix(const Matrix &matrix) {
    this->columnNumber = matrix.columnNumber;
    this->rowNumber = matrix.rowNumber;
    this->valueItem = matrix.valueItem;
    this->counter = matrix.counter;
//    mutex m;
//    m.lock();
    (*counter) +=1;
//    m.unlock();
}
```

```
Matrix & Matrix::operator=(const Matrix &matrix) {
    if(this == &matrix) return *this;

    if(*(counter) == 1)  {
        delete counter;
        delete [] valueItem ;
    }else if(*(counter)==0){
        delete counter;
    }
    else (*counter)--;

    this->counter = matrix.counter;
    this->rowNumber = matrix.rowNumber;
    this->columnNumber = matrix.columnNumber;
    this->valueItem = matrix.valueItem;
    (*counter)++;
    return *this;
}
```

在此两个方法里，既要防止内存泄漏，规避 memoryCopy 造成的内存消耗，同时维护好动态数组的个数，防止在析构时，多次释放统一数组引起错误。在 copy constructor 中，直接进行指针传递，将 counter 地址下的值+1。而重写等号复杂些，当 this 对应的 counter 地址下数值为 1 时需要删除对应的 valueItem，以及 counter，然后再进行赋值，并+1。

```
Matrix matrix7(matrix1);
cout<<"copy constructor :"<<endl;
cout<<matrix7<<endl;

Matrix matrix8;
matrix8 = matrix1;
cout<<"Assignment = override : "<<endl;
cout<<matrix8<<endl;
```

```
copy constructor :
1.000 1.000 1.000 1.000 1.000
1.000 1.000 1.000 1.000 1.000

Assignment = override :
1.000 1.000 1.000 1.000 1.000
1.000 1.000 1.000 1.000 1.000
```

2.3 析构函数

```
Matrix::~Matrix() {
//    cout<<*counter<<endl;
    if(*(counter) == 1) {
//        cout<<"You really delete the pointer of object Matrix"<<endl;
        delete [] valueItem ;
        delete counter;
    }
    else {
//        mutex m;
//        m.lock();
        (*counter) -=1;
//        m.unlock();
    }
}
```

当 counter 地址下的值为 1 时真正删除 new 申请的内存空间，大于 1 时只需要将值--即可，不再进行输出信息验证

2.4 >> && << 输入输出

和 lecture/lab 课上讲的相似，按照矩阵行列输出，前面其实已经有所展示，而在>>输入时，提示用户输入行数、列数、以及里面要存储的 value，并进行了违法输入判断以及警告，重点不在于此部分，就不再进行防违法输入演示。

```
Matrix matrix;
cin>>matrix;
cout<<matrix<<endl;
```

```
Please input the row number of  matrix: 3
Please input the column number of matrix: 2
2.1e2 1.7
-9.8 3.4
1.0 2.0e-1
210.000 1.700
-9.800 3.400
1.000 0.200
```

2.5 +/- operator

```
float matrixValue[10] = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}
Matrix matrix1( rowNumber: 2, columnNumber: 5,matrixValue);
  cout<<"Matrix constructor : "<<endl;
  cout<<matrix1<<endl;
Matrix matrix2( rowNumber: 5, columnNumber: 2,matrixValue);

Matrix matrix3 = matrix1+matrix1;
cout<<matrix3<<endl;
Matrix matrix10 = matrix1-matrix1;
cout<<matrix10<<endl;
Matrix matrix9 = matrix1+matrix2;
```

```
type A+B :
2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000

type A-B :
0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000

type A+B :
You are do wrong things matrix addition must two matrix with same rowNumber and columnNumber
```

代码不再专门贴上，实现较为简单，如建立两个矩阵分别 2*5，5*2，同规格的可以直接用+/-运算符进行加减法并且运算结果正确，不同规格会被杀死，不符合矩阵运算原则。

2.6 * 乘法运算符

2.6.1 Matrix 对象与 Matrix 对象乘法 A*B, 或矩阵与数相乘 A*b b 为标量 重载了两个运算符 直接*运算：

```
Matrix matrix4 = matrix1*matrix2;
cout<<matrix4<<endl;

Matrix matrix5 = matrix1*3;
cout<<matrix5<<endl;
```

```
type A*B :
5.000 5.000
5.000 5.000

type A*b :
3.000 3.000 3.000 3.000 3.000
3.000 3.000 3.000 3.000 3.000
```

其中矩阵的乘法套用了之前 Project 中较为暴力的算法，没有将指令集等应用，没将重点放在测速度方面。

2.6.2 友元函数 实现 b*A b 为标量

```
Matrix matrix6 = 4*matrix1;
cout<<matrix6<<endl;
```

```
type b*A :
type A*b :
4.000 4.000 4.000 4.000 4.000
4.000 4.000 4.000 4.000 4.000
```

```
Matrix operator *(const float multiply ,const Matrix & matrix){
    cout<<"type b*A :"<<endl;

    return matrix*multiply;
}
```

实现其实还是先进入友元 b*A，然后正常调用 A*b 进行运算。

2.7 ==/!= operator

```
float matrixValue[10] = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0};
Matrix matrix1( rowNumber: 2, columnNumber: 5,matrixValue);
  cout<<"Matrix constructor : "<<endl;
  cout<<matrix1<<endl;
Matrix matrix2( rowNumber: 5, columnNumber: 2,matrixValue);
cout<<(matrix1==matrix2)<<endl;
cout<<(matrix1!=matrix2)<<endl;
```

```
bool Matrix::operator==(const Matrix & matrix) const {
    cout<<"type A==B :"<<endl;

    if(matrix.rowNumber == this->rowNumber&&matrix.columnNumber==this->columnNumber){
        bool judge = true;
        for(int i=0;i<rowNumber*columnNumber;i++){
            if(*(matrix.valueItem+i)!= *(this->valueItem+i)){
                judge = false;
                break;
            }
        }
        return judge;
    } else return false;
}
```

两者较为类似，以==为例，先判断规格是否相同，之后为里面的元素，只要有不同即 break 判断。

```
type A==B :
0
type A!=B :
1
```

2.8 ~ operator

所有运算符重写中最麻烦的一个，首先 ~ 作为一元运算符 ~A 形式出现，需要用友元函数去重写，求逆在这部分用了伴随矩阵除以行列式的思路，前提条件是方阵，倘若不是方阵后面所有的都无需计算，直接终止程序即可，不可能存在逆矩阵。先求一个矩阵的行列式，同时可以去判断行列式是否为 0 和矩阵是否可逆联系起来。如果有逆矩阵存在，利用求行

列式的方法，将每个位置的余子式求出来，构成伴随矩阵，然后转置，除以行列式即为逆矩阵，故在此过程中，同时实现了转置，行列式，求逆的接口。

```
void Matrix:: transposition(){
    for(int i=0;i<rowNumber;i++){
        for(int j=i+1;j<columnNumber;j++){
            float temp = *(valueItem+i*rowNumber+j);
            *(valueItem+i*rowNumber+j) = *(valueItem+j*rowNum
            *(valueItem+j*rowNumber+i) = temp;
        }
    }
    int temp = rowNumber;
    columnNumber = rowNumber;
    rowNumber = temp;
}
```

```
float getA(int num,float * value) {
    if(num==1) return value[0];
    float ans = 0;
    float * subValue = new float [(num-1)*(num-1)]();

    for(int i=0;i<num;i++){
        int m = 0;
        for(int k=num;k<num*num;k++){
            if(k%num!=i){
                subValue[m]= value[k];
                m++;
            }
        }
        double t = getA( num: num-1,subValue);
        if(i%2==0) ans += value[i]*t;
        else ans -= value[i]*t;
    }
    return ans;
    delete []  subValue;
}
```

此部分为求转置和 det 行列式，转置较为好理解，由于一维向量模拟，在转置时，其实即为(1,2)和(2,1)位置互换，找到对应位置进行交换即可。而在求行列式过程中用了递归的思想，都是从第一行展开，去计算每一个对应的值，不断递归，最后逐个返回得到最终的行列式值。下面的求逆矩阵建立在上面两个模块基础上，求伴随矩阵时，即先去掉所在行所在列，拿出来当作一个新的矩阵传入行列式函数计算并乘（-1）的 i+j 方得到代数余子式，之后即可构成伴随矩阵，除先前行列式得到结果。

```
Matrix operator ~(const Matrix &matrix){
    cout<<"type ~A :"<<endl;

    if(matrix.rowNumber!=matrix.columnNumber){
        cout<<"Matrix must be square can inverse!"<<endl;
        exit(0);
    }
    int num = matrix.rowNumber;
    float A = getA(num,matrix.valueItem);
    if(A==0){
        cout<<"Matrix is not full rank,it has no inverse!!"<<endl;
        exit(0);
    }
    float * resultValue = new float [num*num]();
    float * tempValue = new float [(num-1)*(num-1)]();
    for(int i=0;i<num;i++){
        for(int j=0;j<num;j++){
            int m=0;
            for(int k=0;k<num*num;k++){
                if(k%num!=j&&!(k>=i*num&&k<(i+1)*num)){
                    *(tempValue+m) = matrix.valueItem[k];
                    m++;
                }
            }
            resultValue [i*num+j] = (i+j)%2==0?1/A*getA( num: num-1,tempValue):-1/A*getA( num: num-1,tempValue);
        }
    }
```

```
delete [] tempValue;
Matrix resultMatrix =  Matrix (num,num,resultValue);
resultMatrix.transposition();
delete [] resultValue;
return resultMatrix;
```

```
float matrixValue1 [16]= {1,2,3,4,5,6,0,8,9,3,11,12,12,14,18,19};
Matrix matrixI( rowNumber: 4, columnNumber: 4,matrixValue1);
cout<<~matrixI;
```

```
type ~A :
-0.570 0.055 0.093 0.038
-0.139 0.022 -0.148 0.114
0.002 -0.132 -0.003 0.057
0.460 0.074 0.053 -0.109
```

如测试数据对该 4*4 矩阵求逆，得到左图结果，经过手算得到结果正确。

3、原子操作，mutex 解决多线程访问问题。

就像最开始在开头部分提到的，当多线程访问时，如果不对 count 数据进行保护，此时多个线程同时进行更改，会造成数据线程之间并不能同步，在析构函数时容易出现多次释放同一动态数组内存，造成错误。下面为此部分的主要测试代码，构造了多线程同时进行 copy constructor，并进行析构函数，验证此问题。

```
void testCorrect(Matrix & matrix){
    for(int i=0;i<10000;i++){
        Matrix a (matrix);
    }
}

void threadTest(){
    Matrix matrix ( rowNumber: 1, columnNumber: 1);

    auto start = std::chrono::steady_clock::now();
    thread th1 = thread(testCorrect,ref( &: matrix));
    thread th2 = thread(testCorrect,ref( &: matrix));
    thread th3 = thread(testCorrect,ref( &: matrix));
    thread th4 = thread(testCorrect,ref( &: matrix));
    testCorrect( &: matrix);
    th1.join();
    th2.join();
    th3.join();
    th4.join();

    auto end = std::chrono::steady_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
    cout << "matrixProduct 1 , duration = " << duration <<" us"<< std::endl;
}
```

如果操作是 int * 类型，其在多线程访问同一地址数据时，并不能保证数据同步，出现多次释放同一动态数组错误。

```
Assignment4(56938,0x70000a90e000) malloc: *** error for object 0x7fc696402ab0: pointer being freed was not allocated
Assignment4(56938,0x70000a90e000) malloc: *** set a breakpoint in malloc_error_break to debug
Assignment4(56938,0x70000aa14000) malloc: *** error for object 0x7fc696402ab0: pointer being freed was not allocated
```

想到的解决方案是，在 copy constructor 和 析构函数中 添加 mutex 互斥对象，锁定 counter 地址下的数据，当一个线程进行改变时，另一个线程并不能对它进行更改，保证数据完整性。

```
Matrix::Matrix(const Matrix &matrix) {
    this->columnNumber = matrix.columnNumber;
    this->rowNumber = matrix.rowNumber;
    this->valueItem = matrix.valueItem;
    this->counter = matrix.counter;
    mutex m;
    m.lock();
    (*counter) +=1;
    m.unlock();
}
```

```
Matrix::~Matrix() {
//    cout<<*counter<<endl;
    if(*(counter) == 1) {
//        cout<<"You really delete the pointer of object Matrix"<<endl;
        delete [] valueItem ;
        delete counter;
    }
    else {
        mutex m;
        m.lock();
        (*counter) -=1;
        m.unlock();
    }
}
```

```
matrixProduct 1 , duration = 4043 us

Process finished with exit code 0
```

多次实验发现可以正常运行，并且 0 正常结束程序，耗时约为 4000us。

对于 int 类型，需要先取出来，+1，再放回去，并不是原子操作，所以需要 lock 锁定，于是采用另外一种方式 atomic_int 原子，写法几乎相同，但是直接对于不可再分的原子操作，即使是多线程同时运行也不可以打断当前的改变，可以做到结果稳定。

```
matrixProduct 1 , duration = 2611 us

Process finished with exit code 0
```

而且此情况下对应时间约为 2600ms 明显快于 lock, lock 会一定程度上破坏多线程的加速。

4、胡乱设计部分：

一开始由于读题不细，将第 3 问中的实现 A*b b 标量误以为成了 b 列向量，去写了相应的* operator 重载以及友元函数，在此过程中，一开始认为列向量用一个数组表示，传入一个指针即可，在编写过程中发现，CPP 不同于 java，直接一个数组等价于指针，并不知道该数组的长度，还需要传入一个长度来记录数组位数，但 operator 重载 A*b 显然只能识别一个参数，就只能想到了 struct 结构体或者 class，一开始写了一个 class 去实现该问题，重写了不少东西，可以进行计算。

```
class Vector{
private:
    int length;
    float * valueItem;
    atomic_int * use_counter;

public:
    Vector();
    Vector(int length,float * value = NULL);
    ~Vector();
    Vector(const Vector& vector);
    Vector& operator =(const Vector & vector);
    int getLength() const;
    float * getValue() const;
};
```

```
Matrix operator *(const Vector &vector) const;
friend Matrix operator *(const Vector &vector, const Matrix & matrix);

float vectorValue[5] = {1.0,1.0,1.0,1.0,1.0};
Vector vector( length: 5,vectorValue);

cout<<matrix1*vector<<endl;
cout<<vector*matrix1<<endl;
```

```
5.000
5.000

5.000
5.000
```

如上面几张图所示，创建 Vector 对象，重载 Matrix 的 operator *方法及其友元，可以当作列向量进行运算得到结果。

进行完这个操作后，感到有些奇怪，向量也是一个矩阵，有行向量或者列向量，联系这周 lecture 讲的知识，又写另一个 class，这个 class 继承于 Matrix，多了一个 ifTransition bool 类型，从而表示列向量还是行向量，由于此类用处相比于 Matrix 用处较小，只是处理行列向量情况，只重写了其 constructor，copy constructor。

```
class VectorInherit: public Matrix{
private:
    bool ifTransition = false;
public:
    VectorInherit();
    VectorInherit(int length,bool ifTransition,float * value = NULL);
//    ~VectorInherit();
    VectorInherit(const VectorInherit& vector);
```

```
VectorInherit::VectorInherit(int length,bool ifTransition,float *value) :Matrix(length, columnNumber: 1,value){
    this->ifTransition = ifTransition;
    if(ifTransition){
        setColumeColumn(length);
        setRowColumn( row: 1);
    }
}

VectorInherit::VectorInherit(const VectorInherit &vector):Matrix(vector) {}
```

在继承之后相比于之前的 Vector 类，可以节省大代码，而且无需重载*的 vector 参数方法，由于其为 Matrix 的子类，又重写了 copy constructor，可以直接相当于在调用之前重载的 A*B 矩阵相乘的运算符进行计算。

```
float vectorValue[5] = {1.0,1.0,1.0,1.0,1.0};
  Vector vector(5,vectorValue);

  cout<<matrix1*vector<<endl;
  cout<<vector*matrix1<<endl;

VectorInherit vectorInherit( length: 5, ifTransition: false,vectorValue);
cout<<matrix1*vectorInherit<<endl;
VectorInherit vectorInherit1( length: 5, ifTransition: true,vectorValue);
cout<<vectorInherit1*matrix2<<endl;
```

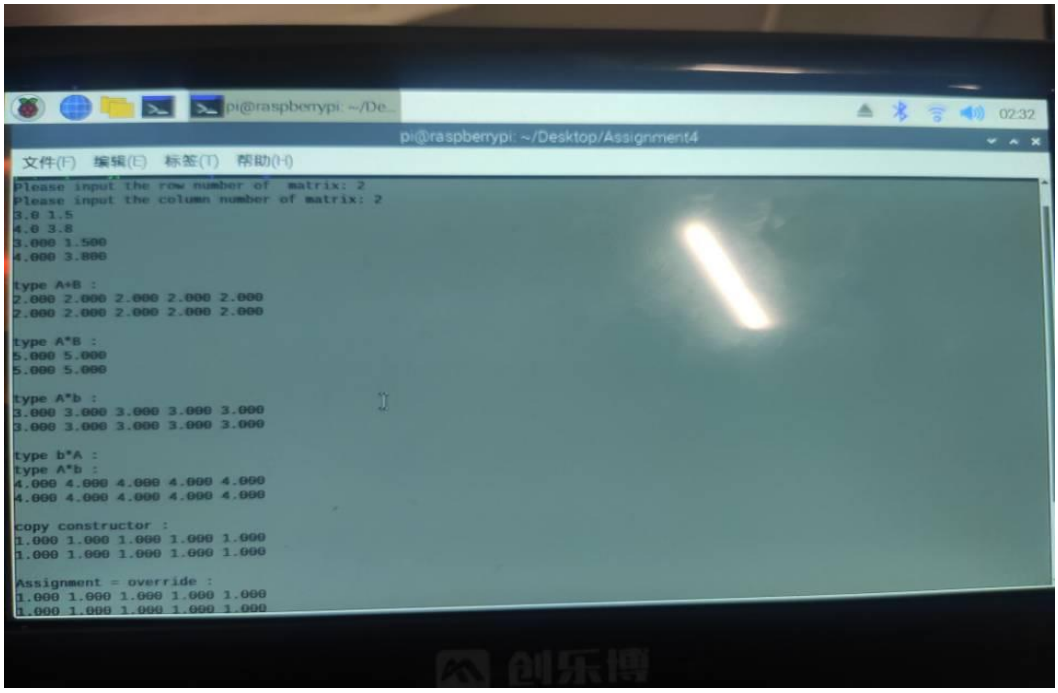
```
type A*B :
5.000
5.000

type A*B :
5.000 5.000
```

借用之前的数据，matrix1 是 2*5 全为 1 的矩阵，matrix2 为 5*2 全为 1 的矩阵，vectorInherit 是列向量可以乘在 2*5 的矩阵右边，得到一个行为 2 的列向量，而 vectorInherit 为行向量可以乘在 5*2 的矩阵前面，得到一个行向量，如结果所示，也证明了继承的子类可以直接去进行*运算符计算，当然如果调换顺序 5*2 的矩阵右乘 5*1 的列向量会一如既往报错终止，不满足矩阵相乘。

当写到这里的时候，十分开心，去看作业要求发现是标量....顿时心里十分崩溃，但也体会了继承的优势，既节省了已有的元素，还可以是逻辑清晰，减少方法构建。

5、ARM 树莓派测试

将所写代码拷贝到 ARM 开发板，运行 CMakeLists.txt 可以正常运行，结果正确。



将最终代码通过 CLion 链接 github 直接 commit ， push 提交，完成!

Part3 Difficulties && Solution or other

在此次 Assignment 中，原子操作，atomic_int 的使用，求逆矩阵（~ operator 的重载），写 cmake 这些比较新颖的上面花费另一些时间，通过查阅手册，知乎，CSDN 找到相关用法，逐个解决问题，除此之外，之间了解到对于智能指针在内存管理多线程方面也可以实现上述要求，在日后的学习中更加深入的去研究下。