WHY

资源申请释放自动化完成，防止内存泄漏

WHAT

智能指针是一个模板类，在构造函数中会对资源进行申请，析构函数中自动进行释放，因此超出变量作用域时会自动析构释放资源。

HOW to use

unique_ptr

```
    unique_ptr<int> u1 = make_unique<int>(25);
    // unique_ptr<int> u2 = u1; // ptr是唯一的，只能使用 移动语义 move()
    auto u2 = move(u1);

    // cout << *u1 << endl;  //u1已经为空指针，会报错
    cout << *u2 << endl;
```

当资源唯一且无法复制时使用unique_ptr

shared_ptr

```
class MyClass
{
public:
    MyClass();
    ~MyClass();
};

MyClass::MyClass()
{
    cout << "constructor invoked" << endl;
}

MyClass::~MyClass()
{
    cout << "destructor invoked" << endl;
}
```

```
int main()
{
    shared_ptr<MyClass> s1 = make_shared<MyClass>();
    cout << "shared count: " << s1.use_count() << endl;

    {
        shared_ptr<MyClass> s2 = s1;
        cout << "shared count: " << s2.use_count() << endl;
    }

    cout << "shared count: " << s1.use_count() << endl;

    return 0;
}
```

```
constructor invoked
shared count: 1
shared count: 2
shared count: 1
destructor invoked
```

shared_ptr的失效：交叉引用

```
class CLeader;
class CMember;

class CLeader
{
public:
    CLeader() { cout << "--CLeader::CLeader()" << endl; }
    ~CLeader() { cout << "--CLeader:;~CLeader() " << endl; }

    std::shared_ptr<CMember> member;
};

class CMember
{
public:
    CMember() { cout << "--CMember::CMember()" << endl; }
    ~CMember() { cout << "--CMember::~CMember() " << endl; }

    std::shared_ptr<CLeader> leader;
};

void TestSharedPtrCrossReference()
{
    cout << "TestCrossReference Start ... "<< endl;
    shared_ptr<CLeader> ptrleader(new CLeader);
    shared_ptr<CMember> ptrmember(new CMember);

    ptrleader->member = ptrmember;
    ptrmember->leader = ptrleader;
```

```
    cout << "--ptrleader.use_count: " << ptrleader.use_count() << endl;
    cout << "--ptrmember.use_count: " << ptrmember.use_count() << endl;

    cout << "TestCrossReference End! "<< endl;
}
```

```
TestCrossReference Start ...
--CLeader::CLeader()
--CMember::CMember()
--ptrleader.use_count: 2
--ptrmember.use_count: 2 //这里并没有析构
TestCrossReference End!
```

解决交叉引用问题！ weak_ptr

weak_ptr 与 shared_ptr 协同使用

weak_ptr，是一种弱引用，需要使用数据时使用weak_ptr.lock()返回shared_ptr对象进行解引用，weak_ptr作用起到打断shared_ptr的作用，通常不出现循环引用时不使用

```
class CLeader
{
public:
    CLeader() { cout << "--CLeader::CLeader()" << endl; }
    ~CLeader() { cout << "--CLeader:;~CLeader() " << endl; }

    std::shared_ptr<CMember> member;
};

class CMember
{
public:
    CMember() { cout << "--CMember::CMember()" << endl; }
    ~CMember() { cout << "--CMember::~CMember() " << endl; }

    std::weak_ptr<CLeader> leader; //将可能出现循环引用的地方改为weak_ptr打断循环引用链
};

void TestSharedPtrCrossReference()
{
    cout << "TestCrossReference Start ... " << endl;
    shared_ptr<CLeader> ptrleader(new CLeader);
    shared_ptr<CMember> ptrmember(new CMember);

    ptrleader->member = ptrmember;
    ptrmember->leader = ptrleader; //两个shared_ptr会循环引用，造成失败

    cout << "--ptrleader.use_count: " << ptrleader.use_count() << endl;
    cout << "--ptrmember.use_count: " << ptrmember.use_count() << endl;

    cout << "TestCrossReference End! " << endl;
}

int main()
```

```
{
    TestSharedPtrCrossReference();
}
```

```
TestCrossReference Start ...
--CLeader::CLeader()
--CMember::CMember()
--ptrleader.use_count: 1
--ptrmember.use_count: 2
TestCrossReference End!
--CLeader::;~CLeader()
--CMember::~CMember()
```