

第一节课主要内容

- 智能指针
- 右值引用
- 匿名函数
- STL容器
- 正则表达式

第二节课主要内容

- thread、condition、mutex
- atomic
- function、bind
- 使用新特性实现线程池（支持可变参数列表）
- 异常
- 协程
- 其他

C++ 参考手册 <https://zh.cppreference.com/w/cpp>

编译方法

主要是要加上 `-std=c++11` 或者更高的版本，比如 `-std=c++14`

编译范例： `g++ -o test test.cpp -std=c++11`

一 为什么需要智能指针

智能指针主要解决以下问题：

1. 内存泄漏：内存手动释放，使用智能指针可以自动释放
`malloc free; new delete`
2. 共享所有权指针的传播和释放，比如多线程使用同一个对象时析构问题

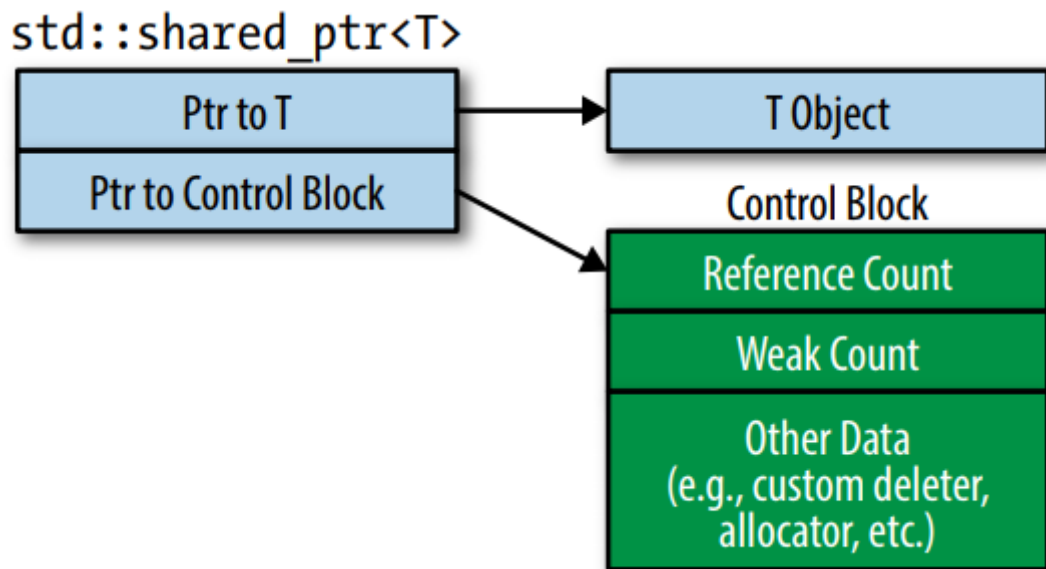
C++里面的四个智能指针: `auto_ptr`, `shared_ptr`, `unique_ptr`, `weak_ptr` 其中后三个是C++11支持，并且第一个已经被C++11弃用。

几个指针的特点：

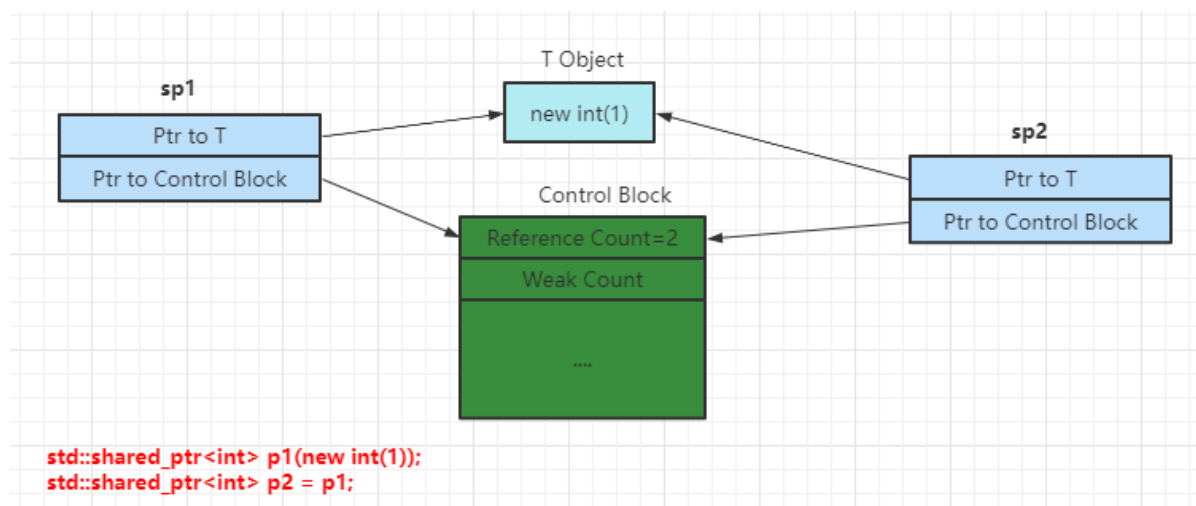
- `unique_ptr`独占对象的所有权，由于没有引用计数，因此性能较好。
- `shared_ptr`共享对象的所有权，但性能略差。
- `weak_ptr`配合`shared_ptr`，解决循环引用的问题。

三者如何选择呢？

1.1 `shared_ptr`内存模型



`shared_ptr` 内部包含两个指针，一个指向对象，另一个指向控制块(control block)，控制块中包含一个引用计数(reference count)，一个弱计数(weak count)和其它一些数据。



1.2 智能指针使用场景案例

范例1-0-shared_ptr_app

1. 使用智能指针可以自动释放占用的内存

```
{
    shared_ptr<Buffer> buf = make_shared<Buffer>("auto free memory"); // Buffer
对象分配在堆上，但能自动释放
    对比
    Buffer *buf = new Buffer("auto free memory");// Buffer对象分配在堆上，但需要手动
delete释放
}
```

2. 共享所有权指针的传播和释放

范例代码：1-2-shared_ptr-app



同样的数据，但不同的业务处理不一样：

比如一帧 视频数据：
 业务A要将该数据保存到本地；
 业务B要将该数据推送到云服务器。

1.3 shared_ptr共享的智能指针

`std::shared_ptr`使用引用计数，每一个`shared_ptr`的拷贝都指向相同的内存。再最后一个`shared_ptr`析构的时候，内存才会被释放。

`shared_ptr`共享被管理对象，同一时刻可以有多个`shared_ptr`拥有对象的所有权，当最后一个`shared_ptr`对象销毁时，被管理对象自动销毁。

简单来说，`shared_ptr`实现包含了两部分，

- 一个指向堆上创建的对象裸指针，`raw_ptr`
- 一个指向内部隐藏的、共享的管理对象。`share_count_object`

第一部分没什么好说的，第二部分是需要注意的重点：

- `use_count`，当前这个堆上对象被多少对象引用了，简单来说就是引用计数。

1.3.1 shared_ptr的基本用法和常用函数

`s.get()`：返回`shared_ptr`中保存的裸指针；

`s.reset(...)`：重置`shared_ptr`；

- `reset()`不带参数时，若智能指针`s`是唯一指向该对象的指针，则释放，并置空。若智能指针`P`不是唯一指向该对象的指针，则引用计数减少1，同时将`P`置空。
- `reset()`带参数时，若智能指针`s`是唯一指向对象的指针，则释放并指向新的对象。若`P`不是唯一的指针，则只减少引用计数，并指向新的对象。如：

```
auto s = make_shared<int>(100);
s.reset(new int(200));
```

`s.use_count()`：返回`shared_ptr`的强引用计数；

`s.unique()`：若`use_count()`为1，返回`true`，否则返回`false`。

1 初始化`make_shared/reset`

通过构造函数、`std::shared_ptr`辅助函数和`reset`方法来初始化`shared_ptr`，代码如下：

```
// 智能指针初始化
std::shared_ptr<int> p1(new int(1));
std::shared_ptr<int> p2 = p1;
std::shared_ptr<int> p3;
p3.reset(new int(1));

if(p3) {
    cout << "p3 is not null";
}
```

我们应该优先使用`make_shared`来构造智能指针，因为他更高效。

```
auto sp1 = make_shared<int>(100);
或
shared_ptr<int> sp1 = make_shared<int>(100);

//相当于
shared_ptr<int> sp1(new int(100));
```

不能将一个原始指针直接赋值给一个智能指针，例如，下面这种方法是错误的：

```
std::shared_ptr<int> p = new int(1);
```

`shared_ptr`不能通过“直接将原始这种赋值”来初始化，需要通过构造函数和辅助方法来初始化。

- 对于一个未初始化的智能指针，可以通过`reset`方法来初始化；
- 当智能指针有值的时候调用`reset`会引起引用计数减1。

另外智能指针可以通过重载的`bool`类型操作符来判断。

```
// 1-3-1-reset-count
#include <iostream>
#include <memory>
using namespace std;
```

```

int main()
{

    std::shared_ptr<int> p1;
    p1.reset(new int(1));
    std::shared_ptr<int> p2 = p1;

    // 引用计数此时应该是2
    cout << "p2.use_count() = " << p2.use_count() << endl;
    p1.reset();
    cout << "p1.reset()\n";
    // 引用计数此时应该是2
    cout << "p2.use_count()= " << p2.use_count() << endl;
    if(!p1) {
        cout << "p1 is empty\n";
    }
    if(!p2) {
        cout << "p2 is empty\n";
    }
    p2.reset();
    cout << "p2.reset()\n";
    cout << "p2.use_count()= " << p2.use_count() << endl;
    if(!p2) {
        cout << "p2 is empty\n";
    }
    return 0;
}

```

2 获取原始指针get

当需要获取原始指针时，可以通过get方法来返回原始指针，代码如下所示：

```

std::shared_ptr<int> ptr(new int(1));
int *p = ptr.get(); //
不小心 delete p;

```

谨慎使用p.get()的返回值，如果你不知道其危险性则永远不要调用get()函数。

p.get()的返回值就相当于一个裸指针的值，不合适的使用这个值，上述陷阱的所有错误都有可能发生，遵守以下几个约定：

- 不要保存p.get()的返回值，无论是保存为裸指针还是shared_ptr都是错误的
- 保存为裸指针不知什么时候就会变成空悬指针，保存为shared_ptr则产生了独立指针
- 不要delete p.get()的返回值，会导致对一块内存delete两次的错误

3 指定删除器

如果用shared_ptr管理非new对象或是没有析构函数的类时，应当为其传递合适的删除器。

示例代码如下：

```
//1-3-1-delete
#include <iostream>
#include <memory>
using namespace std;

void DeleteIntPtr(int *p) {
    cout << "call DeleteIntPtr" << endl;
    delete p;
}

int main()
{
    std::shared_ptr<int> p(new int(1), DeleteIntPtr);
    return 0;
}
```

当p的引用计数为0时，自动调用删除器DeleteIntPtr来释放对象的内存。删除器可以是一个lambda表达式，上面的写法可以改为：

```
std::shared_ptr<int> p(new int(1), [](int *p) {
    cout << "call lambda delete p" << endl;
    delete p;});
```

当我们用shared_ptr管理动态数组时，需要指定删除器，**因为shared_ptr的默认删除器不支持数组对象**，代码如下所示：

```
std::shared_ptr<int> p3(new int[10], [](int *p) { delete [] p;});
```

1.3.2 使用shared_ptr要注意的问题

1 不要用一个原始指针初始化多个shared_ptr

例如下面错误范例：

```
int *ptr = new int;
shared_ptr<int> p1(ptr);
shared_ptr<int> p2(ptr); // 逻辑错误
```

2 不要在函数实参中创建shared_ptr

对于下面的写法：

```
function(shared_ptr<int>(new int), g()); //有缺陷
```

因为C++的函数参数的计算顺序在不同的编译器不同的约定下可能是不一样的，一般是从右到左，但也可能从左到右，所以，可能的过程是先new int，然后调用g()，如果恰好g()发生异常，而shared_ptr还没有创建，则int内存泄漏了，正确的写法应该是先创建智能指针，代码如下：

```
shared_ptr<int> p(new int);
function(p, g());
```

3 通过shared_from_this()返回this指针

不要将this指针作为shared_ptr返回出来，因为this指针本质上是一个裸指针，因此，这样可能会导致重复析构，看下面的例子。

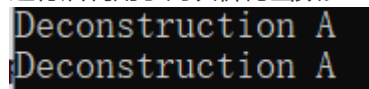
```
//1-3-2-shared_from_this
#include <iostream>
#include <memory>

using namespace std;

class A
{
public:
    shared_ptr<A> GetSelf()
    {
        return shared_ptr<A>(this); // 不要这么做
    }
    ~A()
    {
        cout << "Destructor A" << endl;
    }
};

int main()
{
    shared_ptr<A> sp1(new A);
    shared_ptr<A> sp2 = sp1->GetSelf();
    return 0;
}
```

运行后调用了两次析构函数。



在这个例子中，由于用同一个指针（this）构造了两个智能指针sp1和sp2，而他们之间是没有任何关系的，在离开作用域之后this将会被构造的两个智能指针各自析构，导致重复析构的错误。

正确返回this的shared_ptr的做法是：让目标类通过std::enable_shared_from_this类，然后使用基类的成员函数shared_from_this()来返回this的shared_ptr，如下所示。

```
//1-3-2-shared_from_this2
#include <iostream>
#include <memory>

using namespace std;
```

```

class A: public std::enable_shared_from_this<A>
{
public:
    shared_ptr<A> GetSelf()
    {
        return shared_from_this(); //
    }
    ~A()
    {
        cout << "Destructor A" << endl;
    }
};

int main()
{
    shared_ptr<A> sp1(new A);
    shared_ptr<A> sp2 = sp1->GetSelf(); // ok

    return 0;
}

```

在weak_ptr章节我们继续讲解使用shared_from_this()的原因。

4 避免循环引用

循环引用会导致内存泄漏，比如：

```

// 1-3-2-cycle-shared-ptr
#include <iostream>
#include <memory>
using namespace std;

class A;
class B;

class A {
public:
    std::shared_ptr<B> bptr;
    ~A() {
        cout << "A is deleted" << endl;
    }
};

class B {
public:
    std::shared_ptr<A> aptr;
    ~B() {
        cout << "B is deleted" << endl;
    }
};

int main()
{
    {

```



```

        std::shared_ptr<A> ap(new A);
        std::shared_ptr<B> bp(new B);
        ap->bp_ptr = bp;
        bp->ap_ptr = ap;
    }
    cout<< "main leave" << endl; // 循环引用导致ap bp退出了作用域都没有析构
    return 0;
}

```

循环引用导致ap和bp的引用计数为2，在离开作用域之后，ap和bp的引用计数减为1，并不回减为0，导致两个指针都不会被析构，产生内存泄漏。

解决的办法是把A和B任何一个成员变量改为weak_ptr，具体方法见weak_ptr章节。

1.4 unique_ptr独占的智能指针

1. unique_ptr是一个独占型的智能指针，不能将其赋值给另一个unique_ptr
2. unique_ptr可以指向一个数组
3. unique_ptr需要确定删除器的类型

范例代码：1-4-unique_ptr

unique_ptr是一个独占型的智能指针，它不允许其他的智能指针共享其内部的指针，不允许通过赋值将一个unique_ptr赋值给另一个unique_ptr。下面的错误示例。

```

unique_ptr<T> my_ptr(new T);
unique_ptr<T> my_other_ptr = my_ptr; // 报错，不能复制

```

unique_ptr不允许复制，但可以通过函数返回给其他的unique_ptr，还可以通过std::move来转移到其他的unique_ptr，这样它本身就不再拥有原来指针的所有权了。例如

```

unique_ptr<T> my_ptr(new T); // 正确
unique_ptr<T> my_other_ptr = std::move(my_ptr); // 正确
unique_ptr<T> ptr = my_ptr; // 报错，不能复制

```

std::make_shared是c++11的一部分，但std::make_unique不是。它是在c++14里加入标准库的。

```

auto upw1(std::make_unique<widget>()); // with make func
std::unique_ptr<widget> upw2(new widget); // without make func

```

使用new的版本重复了被创建对象的键入，但是make_unique函数则没有。重复类型违背了软件工程的一个重要原则：应该避免代码重复，代码中的重复会引起编译次数增加，导致目标代码膨胀。

除了unique_ptr的独占性，unique_ptr和shared_ptr还有一些区别，比如

- unique_ptr可以指向一个数组，代码如下所示

```
std::unique_ptr<int []> ptr(new int[10]);
ptr[9] = 9;
std::shared_ptr<int []> ptr2(new int[10]); // 这个是不合法的
```

- unique_ptr指定删除器和shared_ptr有区别

```
std::shared_ptr<int> ptr3(new int(1), [](int *p){delete p;}); // 正确
std::unique_ptr<int> ptr4(new int(1), [](int *p){delete p;}); // 错误
```

unique_ptr需要确定删除器的类型，所以不能像shared_ptr那样直接指定删除器，可以这样写：

```
std::unique_ptr<int, void(*)(int*)> ptr5(new int(1), [](int *p){delete p;}); //
正确
```

关于shared_ptr和unique_ptr的使用场景是要根据实际应用需求来选择。

如果希望只有一个智能指针管理资源或者管理数组就用unique_ptr，如果希望多个智能指针管理同一个资源就用shared_ptr。

范例代码：1-4-unique_ptr-delete

1.5 weak_ptr弱引用的智能指针

- 什么是weak_ptr
- weak_ptr解决了什么问题；
- weak_ptr为什么能解决问题。

share_ptr虽然已经很好用了，但是有一点share_ptr智能指针还是有内存泄露的情况，当两个对象相互使用一个shared_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。

weak_ptr 是一种**不控制对象生命周期**的智能指针，它指向一个 shared_ptr 管理的对象。进行该对象的内存管理的是那个强引用的shared_ptr，weak_ptr只是提供了对管理对象的一个访问手段。

weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作，它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造，它的构造和析构不会引起引用计数的增加或减少。

1.5.1 weak_ptr的基本用法

1. 通过use_count()方法获取当前观察资源的引用计数，如下所示：

```
shared_ptr<int> sp(new int(10));
weak_ptr<int> wp(sp);
cout << wp.use_count() << endl; //结果讲输出1
```

2. 通过expired()方法判断所观察资源是否已经释放，如下所示：

```

shared_ptr<int> sp(new int(10));
weak_ptr<int> wp(sp);
if(wp.expired())
    cout << "weak_ptr无效,资源已释放";
else
    cout << "weak_ptr有效";

```

3. 通过lock方法获取监视的shared_ptr, 如下所示:

lock有什么用处?

范例: 1-5-1-weak_ptr

```

std::weak_ptr<int> gw;
void f()
{
    auto spt = gw.lock();
    if(gw.expired()) {
        cout << "gw无效,资源已释放";
    }
    else {
        cout << "gw有效, *spt = " << *spt << endl;
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        f();
    }
    f();
    return 0;
}

```

1.5.2 weak_ptr返回this指针

shared_ptr章节中提到不能直接将this指针返回shared_ptr, 需要通过派生std::enable_shared_from_this类, 并通过其方法shared_from_this来返回指针, 原因是std::enable_shared_from_this类中有一个weak_ptr, 这个weak_ptr用来观察this智能指针, 调用shared_from_this()方法是, 会调用内部这个weak_ptr的lock()方法, 将所观察的shared_ptr返回, 再看前面的范例

```

#include <iostream>
#include <memory>

using namespace std;

class A: public std::enable_shared_from_this<A>
{
public:

```

```

shared_ptr<A>GetSelf()
{
    return shared_from_this(); //
}
~A()
{
    cout << "Destructor A" << endl;
}
};

int main()
{
    shared_ptr<A> sp1(new A);
    shared_ptr<A> sp2 = sp1->GetSelf(); // ok

    return 0;
}

```

输出结果如下：

Destructor A

在外面创建A对象的智能指针和通过对象返回this的智能指针都是安全的，因为shared_from_this()是内部的weak_ptr调用lock()方法之后返回的智能指针，在离开作用域之后，spy的引用计数减为0，A对象会被析构，不会出现A对象被析构两次的问题。

需要注意的是，获取自身智能指针的函数尽在shared_ptr的构造函数被调用之后才能使用，因为enable_shared_from_this内部的weak_ptr只有通过shared_ptr才能构造。

1.5.3 weak_ptr解决循环引用问题

在shared_ptr章节提到智能指针循环引用的问题，因为智能指针的循环引用会导致内存泄漏，可以通过weak_ptr解决该问题，只要将A或B的任意一个成员变量改为weak_ptr

```

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

class A;
class B;

class A {
public:
    std::weak_ptr<B> bptr; // 修改为weak_ptr
    ~A() {
        cout << "A is deleted" << endl;
    }
};

class B {
public:
    std::shared_ptr<A> aptr;
}

```

```

~B() {
    cout << "B is deleted" << endl;
}

};

int main()
{
    {
        std::shared_ptr<A> ap(new A);
        std::shared_ptr<B> bp(new B);
        ap->bp_ptr = bp;
        bp->ap_ptr = ap;
    }
    cout<< "main leave" << endl;
    return 0;
}

```

这样在对B的成员赋值时，即执行bp->ap_ptr=ap;时，由于ap_ptr是weak_ptr，它并不会增加引用计数，所以ap的引用计数仍然会是1，在离开作用域之后，ap的引用计数减为0，A指针会被析构，析构后其内部的bp_ptr的引用计数会被减为1，然后在离开作用域后bp引用计数又从1减为0，B对象也被析构，不会发生内存泄漏。

1.5.4 weak_ptr使用注意事项

1. weak_ptr在使用前需要检查合法性。

```

weak_ptr<int> wp;
{
    shared_ptr<int> sp(new int(1)); //sp.use_count()==1
    wp = sp; //wp不会改变引用计数，所以sp.use_count()==1
    shared_ptr<int> sp_ok = wp.lock(); //wp没有重载->操作符。只能这样取所指向的对象
}
shared_ptr<int> sp_null = wp.lock(); //sp_null .use_count()==0;

```

因为上述代码中sp和sp_ok离开了作用域，其容纳的K对象已经被释放了。

得到了一个容纳NULL指针的sp_null对象。**在使用wp前需要调用wp.expired()函数判断一下。**

因为wp还仍旧存在，虽然引用计数等于0，仍有某处“全局”性的存储块保存着这个计数信息。直到最后一个weak_ptr对象被析构，这块“堆”存储块才能被回收。否则weak_ptr无法直到自己所容纳的那个指针资源的当前状态。

如果shared_ptr sp_ok和weak_ptr wp;属于同一个作用域呢？如下所示：

```

weak_ptr<int> wp;
shared_ptr<int> sp_ok;
{
    shared_ptr<int> sp(new int(1)); //sp.use_count()==1
    wp = sp; //wp不会改变引用计数，所以sp.use_count()==1
    sp_ok = wp.lock(); //wp没有重载->操作符。只能这样取所指向的对象
}

if(wp.expired()) {
    cout << "shared_ptr is destroy" << endl;
} else {
    cout << "shared_ptr no destroy" << endl;
}

```

范例见：1-5-4-misuse-weak_ptr

1.6 智能指针安全性问题

引用计数本身是安全的，至于智能指针是否安全需要结合实际使用分情况讨论：

情况1：多线程代码操作的是同一个shared_ptr的对象，此时是不安全的。

比如std::thread的回调函数，是一个lambda表达式，其中引用捕获了一个shared_ptr

```
std::thread td([&sp1]()){...};
```

又或者通过回调函数的参数传入的shared_ptr对象，参数类型引用

```

void fn(shared_ptr<A>&sp) {
    ...
}
..
std::thread td(fn, sp1);

```

这时候必然不是线程安全的。

情况2：多线程代码操作的不是同一个shared_ptr的对象

这里指的是管理的数据是同一份，而shared_ptr不是同一个对象。比如多线程回调的lambda的是按值捕获的对象。

```
std::thread td([sp1]()){...};
```

另个线程传递的shared_ptr是值传递，而非引用：

```

void fn(shared_ptr<A>sp) {
    ...
}
..
std::thread td(fn, sp1);

```

这时候每个线程内看到的sp，他们所管理的是同一份数据，用的是同一个引用计数。但是各自是不同的对象，当发生多线程中修改sp指向的操作的时候，是不会出现非预期的异常行为的。
也就是说，如下操作是安全的。

```
void fn(shared_ptr<A>sp) {  
    ...  
    if(..){  
        sp = other_sp;  
    } else {  
        sp = other_sp2;  
    }  
}
```

需要注意：所管理数据的线程安全性问题。显而易见，所管理的对象必然不是线程安全的，必然 sp1、sp2、sp3智能指针实际都是指向对象A，三个线程同时操作对象A，那对象的数据安全必然是需要对象A自己去保证。

1.7 智能指针实践

比如直播场景，一个主播对应多个观众，此时在流媒体服务器中需要将主播的音视频数据帧转发给上千个观众，我们该如何设计数据模型。

二 右值引用和移动语义

作用：C++11中引用了右值引用和移动语义，可以避免无谓的复制，提高了程序性能。

1. 什么是左值、右值

可以从2个角度判断：

- 左值可以取地址、位于等号左边；
- 而右值没法取地址，位于等号右边。

```
int a = 6;
```

- a可以通过 & 取地址，位于等号左边，所以a是左值。
- 6位于等号右边，6没法通过 & 取地址，所以6是个右值。

再举个复杂点的例子：

```

struct A {
    A(int a = 0) {
        a_ = a;
    }

    int a_;
};

A a = A();

```

- 同样的，a可以通过 & 取地址，位于等号左边，**所以a是左值**。
- A()是个临时值，没法通过 & 取地址，位于等号右边，**所以A()是个右值**。

可见左右值的概念很清晰，有地址的变量就是左值，没有地址的字面值、临时值就是右值。

2. 什么是左值引用、右值引用

引用本质是别名，可以通过引用修改变量的值，传参时传引用可以避免拷贝。

范例：2-2-3-1-r-move

2.1 左值引用

左值引用：**能指向左值，不能指向右值的就是左值引用**：

```

int a = 5;
int &ref_a = a; // 左值引用指向左值，编译通过
int &ref_a = 5; // 左值引用指向了右值，会编译失败

```

引用是变量的别名，由于右值没有地址，没法被修改，所以左值引用无法指向右值。

但是，const左值引用是可以指向右值的：

```

const int &ref_a = 5; // 编译通过

```

const左值引用**不会修改指向值**，因此可以指向右值，这也是为什么要使用 const & 作为函数参数的原因之一，如 std::vector 的 push_back：

```

void push_back (const value_type& val);

```

如果没有 const，vec.push_back(5) 这样的代码就无法编译通过。

2.2 右值引用

再看下右值引用，右值引用的标志是 &&，顾名思义，右值引用专门为右值而生，**可以指向右值，不能指向左值**：


```
int &&ref_a_right = 5; // ok

int a = 5;
int &&ref_a_left = a; // 编译不过，右值引用不可以指向左值

ref_a_right = 6; // 右值引用的用途：可以修改右值
```

2.3 对左右值引用本质的讨论

左右值引用的本质。

2.3.1 右值引用有办法指向左值吗？

有办法，使用 `std::move`：

```
int a = 5; // a是个左值
int &ref_a_left = a; // 左值引用指向左值
int &&ref_a_right = std::move(a); // 通过std::move将左值转化为右值，可以被右值引用指向

cout << a; // 打印结果：5
```

在上边的代码里，看上去是左值a通过std::move移动到了右值ref_a_right中，那是不是a里边就没有值了？并不是，打印出a的值仍然是5。

`std::move` 是一个非常有迷惑性的函数：

- 不理解左右值概念的人们往往以为它能把一个变量里的内容移动到另一个变量；
- 但事实上std::move移动不了什么，唯一的功能是把左值强制转化为右值，让右值引用可以指向左值。其实现等同于一个类型转换：`static_cast<T&&>(lvalue)`。所以，单纯的std::move(xxx)不会有性能提升。

同样的，右值引用能指向右值，本质上也是把右值提升为一个左值，并定义一个右值引用通过std::move指向该左值：

```
int &&ref_a = 5;
ref_a = 6;

等同于以下代码：
int temp = 5;
int &&ref_a = std::move(temp);
ref_a = 6;
// 此时temp等于？
```

2.3.2 左值引用、右值引用本身是左值还是右值？

被声明出来的左、右值引用都是左值。因为被声明出的左右值引用是有地址的，也位于等号左边。仔细看下边代码：

范例：2-2-3-2-r-move

```
// 形参是个右值引用
void change(int&& right_value) {
    right_value = 8;
}
```

```

int main() {
    int a = 5; // a是个左值
    int &ref_a_left = a; // ref_a_left是个左值引用
    int &&ref_a_right = std::move(a); // ref_a_right是个右值引用

    change(a); // 编译不过, a是左值, change参数要求右值
    change(ref_a_left); // 编译不过, 左值引用ref_a_left本身也是个左值
    change(ref_a_right); // 编译不过, 右值引用ref_a_right本身也是个左值

    change(std::move(a)); // 编译通过
    change(std::move(ref_a_right)); // 编译通过
    change(std::move(ref_a_left)); // 编译通过

    change(5); // 当然可以直接接右值, 编译通过

    cout << &a << ' ';
    cout << &ref_a_left << ' ';
    cout << &ref_a_right;
    // 打印这三个左值的地址, 都是一样的
}

```

看完后你可能有个问题, `std::move`会返回一个右值引用 `int &&`, 它是左值还是右值呢? 从表达式 `int &&ref = std::move(a)` 来看, 右值引用 `ref` 指向的必须是右值, 所以`move`返回的 `int &&` 是个右值。所以右值引用既可能是左值, 又可能是右值吗? 确实如此: **右值引用既可以是左值也可以是右值, 如果有名称则为左值, 否则是右值。**

或者说: **作为函数返回值的 && 是右值, 直接声明出来的 && 是左值。** 这同样也符合前面章节对左值, 右值的判定方式: 其实引用和普通变量是一样的, `int &&ref = std::move(a)` 和 `int a = 5` 没有什么区别, 等号左边就是左值, 右边就是右值。

最后, 从上述分析中我们得到如下结论:

1. 从性能上讲, 左右值引用没有区别, 传参使用左右值引用都可以避免拷贝。
2. 右值引用可以直接指向右值, 也可以通过`std::move`指向左值; 而左值引用只能指向左值(`const`左值引用也能指向右值)。
3. 作为函数形参时, 右值引用更灵活。虽然`const`左值引用也可以做到左右值都接受, 但它无法修改, 有一定局限性。

```

void f(const int& n) {
    n += 1; // 编译失败, const左值引用不能修改指向变量
}

void f2(int && n) {
    n += 1; // ok
}

int main() {
    f(5);
    f2(5);
}

```

3 右值引用和std::move使用场景

`std::move` 只是类型转换工具, 不会对性能有好处;

右值引用在作为函数形参时更具灵活性。他们有什么实际应用场景吗？

3.1 右值引用优化性能，避免深拷贝

浅拷贝重复释放

对于含有堆内存的类，我们需要提供深拷贝的拷贝构造函数，如果使用默认构造函数，会导致堆内存的重复删除，比如下面的代码：

```
//2-3-1-memory
#include <iostream>

using namespace std;

class A
{
public:
    A() :m_ptr(new int(0)) {
        cout << "constructor A" << endl;
    }

    ~A(){
        cout << "destructor A, m_ptr:" << m_ptr << endl;
        delete m_ptr;
        m_ptr = nullptr;
    }

private:
    int* m_ptr;
};

// 为了避免返回值优化，此函数故意这样写
A Get(bool flag)
{
    A a;
    A b;
    cout << "ready return" << endl;
    if (flag)
        return a;
    else
        return b;
}

int main()
{
    {
        A a = Get(false); // 运行报错
    }
    cout << "main finish" << endl;
    return 0;
}
```

打印

constructor A

constructor A

ready return

destructor A, m_ptr:0xf87af8

destructor A, m_ptr:0xf87ae8

destructor A, m_ptr:0xf87af8

main finish

深拷贝构造函数

在上面的代码中，默认构造函数是浅拷贝，main函数的 a 和Get函数的 b 会指向同一个指针 m_ptr，在析构的时候会导致重复删除该指针。正确的做法是提供深拷贝的拷贝构造函数，比如下面的代码：

```
//2-3-1-memory2
#include <iostream>

using namespace std;

class A
{
public:
    A() :m_ptr(new int(0)) {
        cout << "constructor A" << endl;
    }

    A(const A& a) :m_ptr(new int(*a.m_ptr)) {
        cout << "copy constructor A" << endl;
    }

    ~A(){
        cout << "destructor A, m_ptr:" << m_ptr << endl;
        delete m_ptr;
        m_ptr = nullptr;
    }

private:
    int* m_ptr;
};

// 为了避免返回值优化，此函数故意这样写
A Get(bool flag)
{
    A a;
    A b;
    cout << "ready return" << endl;
    if (flag)
        return a;
    else
        return b;
}

int main()
{
    {
```

```

        A a = Get(false); // 正确运行
    }
    cout << "main finish" << endl;
    return 0;
}

```

运行结果

constructor A

constructor A

ready return

copy constructor A

destructor A, m_ptr:0xea7af8

destructor A, m_ptr:0xea7ae8

destructor A, m_ptr:0xea7b08

main finish

移动构造函数

这样就可以保证拷贝构造时的安全性，但有时这种拷贝构造却是不必要的，比如上面代码中的拷贝构造就是不必要的。上面代码中的 Get 函数会返回临时变量，然后通过这个临时变量拷贝构造了一个新的对象 b，临时变量在拷贝构造完成之后就销毁了，**如果堆内存很大，那么，这个拷贝构造的代价会很大，带来了额外的性能损耗。**有没有办法避免临时对象的拷贝构造呢？答案是肯定的。看下面的代码：

```

//2-3-1-memory3
#include <iostream>

using namespace std;

class A
{
public:
    A() :m_ptr(new int(0)) {
        cout << "constructor A" << endl;
    }

    A(const A& a) :m_ptr(new int(*a.m_ptr)) {
        cout << "copy constructor A" << endl;
    }
    // 移动构造函数，可以浅拷贝
    A(A&& a) :m_ptr(a.m_ptr) {
        a.m_ptr = nullptr; // 为防止a析构时delete data，提前置空其m_ptr
        cout << "move constructor A" << endl;
    }

    ~A(){
        cout << "destructor A, m_ptr:" << m_ptr << endl;
        if(m_ptr)
            delete m_ptr;
    }

private:
    int* m_ptr;
};

```

```
// 为了避免返回值优化，此函数故意这样写
A Get(bool flag)
{
    A a;
    A b;
    cout << "ready return" << endl;
    if (flag)
        return a;
    else
        return b;
}

int main()
{
    {
        A a = Get(false); // 正确运行
    }
    cout << "main finish" << endl;
    return 0;
}
```

运行结果

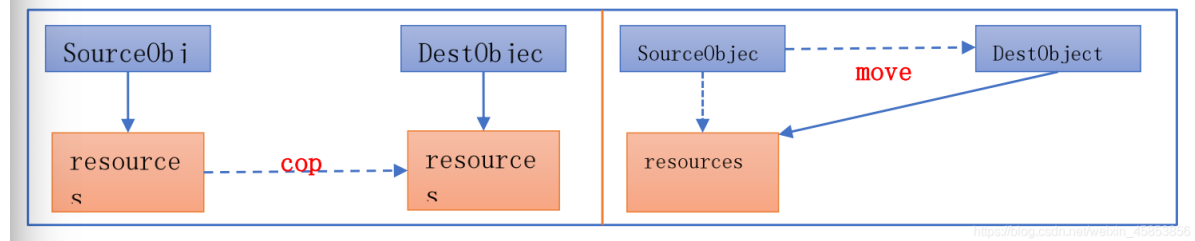
```
constructor A
constructor A
ready return
move constructor A
destructor A, m_ptr:0
destructor A, m_ptr:0xfa7ae8
destructor A, m_ptr:0xfa7af8
main finish
```

上面的代码中没有了拷贝构造，取而代之的是移动构造（Move Construct）。从移动构造函数的实现中可以看到，它的参数是一个右值引用类型的参数 A&&，这里没有深拷贝，只有浅拷贝，这样就避免了对临时对象的深拷贝，提高了性能。这里的 A&& 用来根据参数是左值还是右值来建立分支，如果是临时值，则会选择移动构造函数。移动构造函数只是将临时对象的资源做了浅拷贝，不需要对其进行深拷贝，从而避免了额外的拷贝，提高性能。这也就是所谓的移动语义（move 语义），**右值引用的一个重要目的是用来支持移动语义的。**

移动语义可以将资源（堆、系统对象等）**通过浅拷贝方式从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝以及销毁**，可以大幅度提高 C++ 应用程序的性能，消除临时对象的维护（创建和销毁）对性能的影响。

3.2 移动(move)语义

move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转义，没有内存拷贝。要move语义起作用，核心在于需要对应类型的构造函数支持。



```
//2-3-2-move
#include <iostream>
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <string.h>
using namespace std;

class MyString {
private:
    char* m_data;
    size_t m_len;
    void copy_data(const char *s) {
        m_data = new char[m_len+1];
        memcpy(m_data, s, m_len);
        m_data[m_len] = '\0';
    }
public:
    MyString() {
        m_data = NULL;
        m_len = 0;
    }

    MyString(const char* p) {
        m_len = strlen(p);
        copy_data(p);
    }

    MyString(const MyString& str) {
        m_len = str.m_len;
        copy_data(str.m_data);
        std::cout << "Copy Constructor is called! source: " << str.m_data <<
std::endl;
    }

    MyString& operator=(const MyString& str) {
        if (this != &str) {
            m_len = str.m_len;
            copy_data(str.m_data);
        }
        std::cout << "Copy Assignment is called! source: " << str.m_data <<
std::endl;
        return *this;
    }

    // 用c++11的右值引用来定义这两个函数
    MyString(MyString&& str) {
```

```

        std::cout << "Move Constructor is called! source: " << str.m_data <<
std::endl;
        m_len = str.m_len;
        m_data = str.m_data; //避免了不必要的拷贝
        str.m_len = 0;
        str.m_data = NULL;
    }

    MyString& operator=(MyString&& str) {
        std::cout << "Move Assignment is called! source: " << str.m_data <<
std::endl;
        if (this != &str) {
            m_len = str.m_len;
            m_data = str.m_data; //避免了不必要的拷贝
            str.m_len = 0;
            str.m_data = NULL;
        }
        return *this;
    }

    virtual ~MyString() {
        if (m_data) free(m_data);
    }
};

int main()
{
    MyString a;
    a = MyString("Hello"); // Move Assignment
    MyString b = a; // Copy Constructor
    MyString c = std::move(a); // Move Constructor is called! 将左值转为右值

    std::vector<MyString> vec;
    vec.push_back(MyString("World")); // Move Constructor is called!
    return 0;
}

```

有了右值引用和转移语义，我们在设计和实现类时，对于需要动态申请大量资源的类，应该设计右值引用的拷贝构造函数和赋值函数，以提高应用程序的效率。

3.3 forward 完美转发

forward 完美转发实现了**参数在传递过程中保持其值属性的功能**，即若是左值，则传递之后仍然是左值，若是右值，则传递之后仍然是右值。

现存在一个函数

```

Template<class T>
void func(T &&val);

```

根据前面所描述的，这种引用类型既可以对左值引用，亦可以对右值引用。

但要注意，引用以后，**这个val值它本质上是一个左值！**

看下面例子


```
int &&a = 10;
int &&b = a; //错误
```

注意这里，a是一个右值引用，但其本身a也有内存名字，所以a本身是一个左值，再用右值引用引用a这是不对的。

因此我们有了std::forward()完美转发，这种T &&val中的val是左值，但如果我们用std::forward (val)，就会按照参数原来的类型转发；

```
int &&a = 10;
int &&b = std::forward<int>(a);
```

这样是正确的！

通过**范例2-3-2-forward1**巩固下知识：

```
// 2-3-2-forward1
#include <iostream>

using namespace std;

template <class T>
void Print(T &t)
{
    cout << "L" << t << endl;
}
template <class T>
void Print(T &&t)
{
    cout << "R" << t << endl;
}
template <class T>
void func(T &&t)
{
    Print(t);
    Print(std::move(t));
    Print(std::forward<T>(t));
}
int main()
{
    cout << "-- func(1)" << endl;
    func(1);
    int x = 10;
    int y = 20;
    cout << "\n-- func(x)" << endl;
    func(x); // x本身是左值
    cout << "\n-- func(std::forward<int>(y))" << endl;
    func(std::forward<int>(y)); //T为int，以右值方式转发y
    cout << "\n-- func(std::forward<int&>(y))" << endl;
    func(std::forward<int&>(y));
    cout << "\n-- func(std::forward<int&&(y))" << endl;
    func(std::forward<int&&(y));
    return 0;
}
```

运行结果：

```

-- func(1)
L1
R1
R1

-- func(x)
L10
R10
L10

-- func(std::forward(y))
L20
R20
R20

-- func(std::forward<int&>(y))
L20
R20
L20

```

解释：

func(1)：由于1是右值，所以未定的引用类型T&&v被一个右值初始化后变成了一个右值引用，但是在func()函数体内部，调用PrintT(v)时，v又变成了一个左值（因为在std::forward里它已经变成了一个具名的变量，所以它是一个左值），因此，示例测试结果第一个PrintT被调用，打印出“L1”

调用PrintT (std::forward(v)) 时，由于std::forward会按参数原来的类型转发，因此，它还是一个右值（这里已经发生了类型推导，所以这里的T&&不是一个未定的引用类型，会调用void PrintT (T&&t) 函数打印 “R1”。调用PrintT(std::move(v))是将v变成一个右值（v本身也是右值），因此，它将输出“R1”

func(x)未定的引用类型T&&v被一个左值初始化后变成了一个左值引用，因此，在调用PrintT(std::forward(v))时它会被转发到void PrintT (T&t) 。

forward将左值转换为右值：

```

MyString str1 = "hello";
MyString str2(str1);
MyString str3 = Fun();
MyString str4 = move(str2);
MyString str5(forward<MyString>(str3));

```

```

template <class T>
void Print(T &t)
{
    cout << "L" << t << endl;
}
template <class T>
void Print(T &&t)
{
    cout << "R" << t << endl;
}
template <class T>
void func(T &&t)
{
    Print(t);
    Print(std::move(t));
    Print(std::forward<T>(t));
}
int main()
{
    cout << "-- func(1)" << endl;
    func(1);
    int x = 10;
    int y = 20;
    cout << "-- func(x)" << endl;
    func(x); // x本身是左值
    cout << "-- func(std::forward<int>(y))" << endl;
    func(std::forward<int>(y)); //
    return 0;
}

```

```

-- func(1)
L1 参数本身是左值
R1 move转为右值
R1 1本身是右值
-- func(x)
L10
R10
L10 x本身是左值
-- func(std::forward<int>(y))
L20
R20 转发为右值

```

综合示例

```

//2-3-3-forward2
#include "stdio.h"
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;
class A
{
public:
    A() : m_ptr(NULL), m_nSize(0) {}
    A(int *ptr, int nSize)
    {
        m_nSize = nSize;
        m_ptr = new int[nSize];
        printf("A(int *ptr, int nSize) m_ptr:%p\n", m_ptr);
        if (m_ptr)
        {
            memcpy(m_ptr, ptr, sizeof(int) * nSize);
        }
    }
    A(const A &other) // 拷贝构造函数实现深拷贝
    {
        m_nSize = other.m_nSize;
        if (other.m_ptr)
        {
            printf("A(const A &other) m_ptr:%p\n", m_ptr);
            if(m_ptr)
                delete[] m_ptr;
            printf("delete[] m_ptr\n");
            m_ptr = new int[m_nSize];
            memcpy(m_ptr, other.m_ptr, sizeof(int) * m_nSize);
        }
    }
}

```

```

        else
        {
            if(m_ptr)
                delete[] m_ptr;
            m_ptr = NULL;
        }
        cout << "A(const int &i)" << endl;
    }
    // 右值引用移动构造函数
    A(A &&other)
    {
        m_ptr = NULL;
        m_nSize = other.m_nSize;
        if (other.m_ptr)
        {
            m_ptr = move(other.m_ptr); // 移动语义
            other.m_ptr = NULL;
        }
    }
    ~A()
    {
        if (m_ptr)
        {
            delete[] m_ptr;
            m_ptr = NULL;
        }
    }
    void deleteptr()
    {
        if (m_ptr)
        {
            delete[] m_ptr;
            m_ptr = NULL;
        }
    }
    int *m_ptr = NULL; // 增加初始化
    int m_nSize = 0;
};

int main()
{
    int arr[] = {1, 2, 3};
    A a(arr, sizeof(arr) / sizeof(arr[0]));
    cout << "m_ptr in a Addr: 0x" << a.m_ptr << endl;
    A b(a);
    cout << "m_ptr in b Addr: 0x" << b.m_ptr << endl;

    b.deleteptr();
    A c(std::forward<A>(a)); // 完美转换
    cout << "m_ptr in c Addr: 0x" << c.m_ptr << endl;
    c.deleteptr();
    vector<int> vect{1, 2, 3, 4, 5};
    cout << "before move vect size: " << vect.size() << endl;
    vector<int> vect1 = move(vect);
    cout << "after move vect size: " << vect.size() << endl;
    cout << "new vect1 size: " << vect1.size() << endl;
    return 0;
}

```

3.4 `emplace_back` 减少内存拷贝和移动

范例：//2-5-`emplace_back`

对于STL容器，C++11后引入了`emplace_back`接口。

`emplace_back`是就地构造，不用构造后再次复制到容器中。因此效率更高。

考虑这样的语句：

```
vector<string> testVec;  
testVec.push_back(string(16, 'a'));
```

上述语句足够简单易懂，将一个string对象添加到testVec中。底层实现：

- 首先，`string(16, 'a')`会创建一个string类型的临时对象，这涉及到一次string构造过程。
- 其次，vector内会创建一个新的string对象，这是第二次构造。
- 最后在`push_back`结束时，最开始的临时对象会被析构。加在一起，这两行代码会涉及到两次string构造和一次析构。

c++11可以用`emplace_back`代替`push_back`，`emplace_back`可以直接在vector中构建一个对象，而非创建一个临时对象，再放进vector，再销毁。`emplace_back`可以省略一次构建和一次析构，从而达到优化的目的。

测试范例2-3-4-`emplace_back`

```
//2-3-4-emplace_back  
//time_interval.h  
#ifndef TIME_INTERVAL_H  
#define TIME_INTERVAL_H  
#include <iostream>  
#include <memory>  
#include <string>  
#ifdef GCC  
#include <sys/time.h>  
#else  
#include <ctime>  
#endif // GCC  
  
class TimeInterval  
{  
public:  
    TimeInterval(const std::string& d) : detail(d)  
    {  
        init();  
    }  
  
    TimeInterval()  
    {  
        init();  
    }  
  
    ~TimeInterval()
```

```

    {
#ifdef GCC
        gettimeofday(&end, NULL);
        std::cout << detail
            << 1000 * (end.tv_sec - start.tv_sec) + (end.tv_usec -
start.tv_usec) / 1000
            << " ms" << endl;
#else
        end = clock();
        std::cout << detail
            << (double)(end - start) << " ms" << std::endl;
#endif // GCC
    }

protected:
    void init() {
#ifdef GCC
        gettimeofday(&start, NULL);
#else
        start = clock();
#endif // GCC
    }
private:
    std::string detail;
#ifdef GCC
        timeval start, end;
#else
        clock_t start, end;
#endif // GCC
    };
#define TIME_INTERVAL_SCOPE(d)    std::shared_ptr<TimeInterval>
time_interval_scope_begin = std::make_shared<TimeInterval>(d)
#endif // TIME_INTERVAL_H

```

```

//2-3-4-emplace_back
#include <vector>
#include <string>
#include "time_interval.h"

int main() {

    std::vector<std::string> v;
    int count = 10000000;
    v.reserve(count);           //预分配十万大小，排除掉分配内存的时间
    {
        TIME_INTERVAL_SCOPE("push_back string:");
        for (int i = 0; i < count; i++)
        {
            std::string temp("ceshi");
            v.push_back(temp);// push_back(const string&), 参数是左值引用，内部做深拷
            贝
        }
    }

    v.clear();
}

```

```

{
    TIME_INTERVAL_SCOPE("push_back move(string):");
    for (int i = 0; i < count; i++)
    {
        std::string temp("ceshi");
        v.push_back(std::move(temp)); // push_back(string &&), 参数是右值引用
    }
}

v.clear();
{
    TIME_INTERVAL_SCOPE("push_back(string):");
    for (int i = 0; i < count; i++)
    {
        v.push_back(std::string("ceshi")); // push_back(string &&), 参数是右值引用
    }
}

v.clear();
{
    TIME_INTERVAL_SCOPE("push_back(c string):");
    for (int i = 0; i < count; i++)
    {
        v.push_back("ceshi"); // push_back(string &&), 参数是右值引用
    }
}

v.clear();
{
    TIME_INTERVAL_SCOPE("emplace_back(c string):");
    for (int i = 0; i < count; i++)
    {
        v.emplace_back("ceshi"); // 只有一次构造函数, 不调用拷贝构造函数, 速度最快
    }
}
}

```

测试结果

```

push_back string:335 ms
push_back move(string):307 ms
push_back(string):285 ms
push_back(c string):295 ms
emplace_back(c string):234 ms

```

第1中方法耗时最长, 原因显而易见, 将调用左值引用的push_back, 且将会调用一次string的拷贝构造函数, 比较耗时, 这里的string还算很短的, 如果很长的话, 差异会更大

第2、3、4中方法耗时基本一样, 参数为右值, 将调用右值引用的push_back, 故调用string的移动构造函数, 移动构造函数耗时比拷贝构造函数少, 因为不需要重新分配内存空间。

第5中方法耗时最少, 因为emplace_back只调用构造函数, 没有移动构造函数, 也没有拷贝构造函数。

为了证实上述论断，我们自定义一个类，并在普通构造函数、拷贝构造函数、移动构造函数中打印相应描述：

范例：2-3-4-emplace_back2

```
//2-3-4-emplace_back2
#include <vector>
#include <string>
#include "time_interval.h"

using namespace std;

class Foo {
public:
    Foo(std::string str) : name(str) {
        std::cout << "constructor" << std::endl;
    }
    Foo(const Foo& f) : name(f.name) {
        std::cout << "copy constructor" << std::endl;
    }
    Foo(Foo&& f) : name(std::move(f.name)){
        std::cout << "move constructor" << std::endl;
    }

private:
    std::string name;
};

int main() {

    std::vector<Foo> v;
    int count = 10000000;
    v.reserve(count);          //预分配十万大小，排除掉分配内存的时间

    {
        TIME_INTERVAL_SCOPE("push_back T:");
        Foo temp("test");
        v.push_back(temp);// push_back(const T&), 参数是左值引用
        //打印结果:
        //constructor
        //copy constructor
    }
    cout << " -----\n" << endl;
    v.clear();
    {
        TIME_INTERVAL_SCOPE("push_back move(T):");
        Foo temp("test");
        v.push_back(std::move(temp));// push_back(T &&), 参数是右值引用
        //打印结果:
        //constructor
        //move constructor
    }
    cout << " -----\n" << endl;
    v.clear();
    {
        TIME_INTERVAL_SCOPE("push_back(T&&):");
```



```

        v.push_back(Foo("test")); // push_back(T &&), 参数是右值引用
        //打印结果:
        //constructor
        //move constructor
    }
    cout << " -----\n" << endl;
    v.clear();
    {
        std::string temp = "test";
        TIME_INTERVAL_SCOPE("push_back(string):");
        v.push_back(temp); // push_back(T &&), 参数是右值引用
        //打印结果:
        //constructor
        //move constructor
    }
    cout << " -----\n" << endl;
    v.clear();
    {
        std::string temp = "test";
        TIME_INTERVAL_SCOPE("emplace_back(string):");
        v.emplace_back(temp); // 只有一次构造函数, 不调用拷贝构造函数, 速度最快
        //打印结果:
        //constructor
    }
}

```

测试结果

```

constructor
copy constructor

push_back T:2 ms
-----

constructor
move constructor

push_back move(T):0 ms
-----

constructor
move constructor

push_back(T&&):0 ms
-----

constructor
move constructor

push_back(string):0 ms
-----

constructor
emplace_back(string):0 ms

```

3.5 小结

C++11 在性能上做了很大的改进，最大程度减少了内存移动和复制，通过右值引用、forward、emplace 和一些无序容器我们可以大幅度改进程序性能。

- 右值引用仅仅是通过改变资源的所有者（剪切方式，而不是拷贝方式）来避免内存的拷贝，能大幅度提高性能。
- forward 能根据参数的实际类型转发给正确的函数（参数用 &&的方式）。
- emplace 系列函数通过直接构造对象的方式避免了内存的拷贝和移动。

三 匿名函数lambda

重点：

- 怎么传递参数。
- 传引用还是传值

3.1 匿名函数的基本语法

```
[捕获列表](参数列表) mutable(可选) 异常属性 -> 返回类型 {  
    // 函数体  
}
```

语法规则：lambda表达式可以看成是一般函数的函数名被略去，返回值使用了一个 -> 的形式表示。唯一与普通函数不同的是增加了“捕获列表”。

```
//[捕获列表](参数列表)->返回类型{函数体}  
int main()  
{  
    auto Add = [](int a, int b)->int {  
        return a + b;  
    };  
    std::cout << Add(1, 2) << std::endl;    //输出3  
    return 0;  
}
```

一般情况下，编译器可以自动推断出lambda表达式的返回类型，所以我们可以不指定返回类型，即：

```
//[捕获列表](参数列表){函数体}  
int main()  
{  
    auto Add = [](int a, int b) {  
        return a + b;  
    };  
    std::cout << Add(1, 2) << std::endl;    //输出3  
    return 0;  
}
```

但是如果函数体内有多个return语句时，编译器无法自动推断出返回类型，此时必须指定返回类型。

3.2 捕获列表

有时候，需要在匿名函数内使用外部变量，所以用捕获列表来传递参数。根据传递参数的行为，捕获列表可分为以下几种：

1 值捕获

与参数传值类似，值捕获的前提是变量可以拷贝，不同之处则在于，**被捕获的变量在 lambda表达式被创建时拷贝**，而非调用时才拷贝：

```
void test3()
{
    cout << "test3" << endl;
    int c = 12;
    int d = 30;
    auto Add = [c, d](int a, int b)->int {
        cout << "d = " << d << endl;
        return c;
    };
    d = 20;
    std::cout << Add(1, 2) << std::endl;
}
```

2 引用捕获

与引用传参类似，引用捕获保存的是**引用**，值会发生变化。

如果Add中加入一句：c = a;

```
void test5()
{
    cout << "test5" << endl;
    int c = 12;
    int d = 30;
    auto Add = [&c, &d](int a, int b)->int {
        c = a; // 编译对的
        cout << "d = " << d << endl;
        return c;
    };
    d = 20;
    std::cout << Add(1, 2) << std::endl;
}
```

3 隐式捕获

手动书写捕获列表有时候是非常复杂的，这种机械性的工作可以交给编译器来处理，这时候可以在捕获列表中写一个 & 或 = 向编译器声明采用引用捕获或者值捕获。

```
void test7()
{
    cout << "test7" << endl;
}
```

```

int c = 12;
int d = 30;

// 把捕获列表的&改成=再测试
auto Add = [=](int a, int b)->int {
    c = a; // 编译对的
    cout << "d = " << d << endl;
    return c;
};
d = 20;
std::cout << Add(1, 2) << std::endl;
std::cout << "c:" << c << std::endl;
}

```

4 空捕获列表

捕获列表'[]'中为空，表示Lambda不能使用所在函数中的变量。

```

void test8()
{
    cout << "test7" << endl;
    int c = 12;
    int d = 30;

    // 把捕获列表的&改成=再测试
    // [] 空值，不能使用外面的变量
    // [=] 传值，lambda外部的变量都能使用
    // [&] 传引用值，lambda外部的变量都能使用
    auto Add = [&](int a, int b)->int {
        cout << "d = " << d << endl; // 编译报错
        return c; // 编译报错
    };
    d = 20;
    std::cout << Add(1, 2) << std::endl;
    std::cout << "c:" << c << std::endl;
}

```

5 表达式捕获

上面提到的值捕获、引用捕获都是已经在外层作用域声明的变量，因此这些捕获方式捕获的均为左值，而不能捕获右值。

C++14之后支持捕获右值，允许捕获的成员用任意的表达式进行初始化，被声明的捕获变量类型会根据表达式进行判断，判断方式与使用 auto 本质上是相同的：

```

void test9()
{
    cout << "test9" << endl;
    auto important = std::make_unique<int>(1);

    auto add = [v1 = 1, v2 = std::move(important)](int x, int y) -> int {
        return x + y + v1 + (*v2);
    };

    std::cout << add(3,4) << std::endl;
}

```

6 泛型 Lambda

在C++14之前，lambda表示的形参只能指定具体的类型，没法泛型化。从 C++14 开始，Lambda 函数的形式参数可以使用 auto关键字来产生意义上的泛型：

```

//泛型 Lambda C++14
void test10()
{
    cout << "test10" << endl;
    auto add = [](auto x, auto y) {
        return x+y;
    };

    std::cout << add(1, 2) << std::endl;
    std::cout << add(1.1, 1.2) << std::endl;
}

```

7 可变lambda

- 采用值捕获的方式，lambda不能修改其值，如果想要修改，使用mutable修饰
- 采用引用捕获的方式，lambda可以直接修改其值

```

void test12() {
    cout << "test12" << endl;
    int v = 5;
    // 值捕获方式，使用mutable修饰，可以改变捕获的变量值
    auto ff = [v]() mutable {return ++v;};

    v = 0;
    auto j = ff(); // j为6
}

void test13() {
    cout << "test13" << endl;
    int v = 5;
    // 采用引用捕获方式，可以直接修改变量值
    auto ff = [&v] {return ++v;};

    v = 0;
    auto j = ff(); // v引用已修改，j为1
}

```

```
}

```

3.3 总结

- 1. 如果捕获列表为[&]，则表示所有的外部变量都按引用传递给lambda使用；
- 2. 如果捕获列表为[=]，则表示所有的外部变量都按值传递给lambda使用；
- 3. 匿名函数构建的时候对于按值传递的捕获列表，**会立即将当前可以取到的值拷贝一份作为常数**，然后将该常数作为参数传递。

Lambda捕获列表总结

[]	空捕获列表，Lambda不能使用所在函数中的变量。
[names]	names是一个逗号分隔的名字列表，这些名字都是Lambda所在函数的局部变量。默认情况下，这些变量会被拷贝，然后按值传递，名字前面如果使用了&，则按引用传递
[&]	隐式捕获列表，Lambda体内使用的局部变量都按引用方式传递
[=]	隐式捕获列表，Lambda体内使用的局部变量都按值传递
[&,identifier_list]	identifier_list是一个逗号分隔的列表，包含0个或多个来自所在函数的变量，这些变量采用值捕获的方式，其他变量则被隐式捕获，采用引用方式传递，identifier_list中的名字前面不能使用&。
[=,identifier_list]	identifier_list中的变量采用引用方式捕获，而被隐式捕获的变量都采用按值传递的方式捕获。identifier_list中的名字不能包含this，且这些名字面前必须使用&。

四 C++11标准库(STL)

STL定义了强大的、基于模板的、可复用的组件，实现了许多通用的数据结构及处理这些数据结构的算法。其中包含三个关键组件——容器（container，流行的模板数据结构）、迭代器（iterator）和算法（algorithm）。

组件	描述
容器	容器是用来管理某一类对象的集合。C++ 提供了各种不同类型的容器，比如 deque、list、vector、map 等。
迭代器	

| 迭代器用于遍历对象集合的元素。这些集合可能是容器，也可能是容器的子集。 |
| 算法 | 算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作。 |

4.1 容器简介

STL容器，可将其分为四类：序列容器、有序关联容器、**无序关联容器**、**容器适配器**
序列容器：

标准库容器类	描述
array	固定大小，直接访问任意元素
deque	从前部或后部进行快速插入和删除操作，直接访问任何元素
forward_list	单链表，在任意位置快速插入和删除
list	双向链表，在任意位置进行快速插入和删除操作
vector	从后部进行快速插入和删除操作，直接访问任意元素

有序关联容器（键按顺序保存）：

标准库容器类	描述
set	快速查找，无重复元素
multiset	快速查找，可有重复元素
map	一对一映射，无重复元素，基于键快速查找
multimap	一对一映射，可有重复元素，基于键快速查找

无序关联容器：

标准库容器类	描述
unordered_set	快速查找，无重复元素
unordered_multiset	快速查找，可有重复元素
unordered_map	一对一映射，无重复元素，基于键快速查找
unordered_multimap	一对一映射，可有重复元素，基于键快速查找

容器适配器：

标准库容器类	描述
stack	后进先出（LIFO）
queue	先进先出（FIFO）
priority_queue	优先级最高的元素先出

序列容器描述了线性的数据结构（也就是说，其中的元素在概念上“排成一行”），例如数组、向量和链表。
关联容器描述非线性的容器，它们通常可以快速锁定其中的元素。这种容器可以**存储值的集合或者键 - 值对**。

栈和队列都是在序列容器的基础上加以约束条件得到的，因此STL把stack和queue作为**容器适配器**来实现，这样就可以使程序以一种约束方式来处理线性容器。类型string支持的功能跟线性容器一样，但是它只能存储字符数据。

4.2 迭代器简介

迭代器在很多方面与指针类似，也是用于指向首类容器中的元素（还有一些其他用途，后面将会提到）。迭代器存有它们所指的特定容器的状态信息，即迭代器对每种类型的容器都有一个实现。有些迭代器的操作在不同容器间是统一的。例如，* 运算符间接引用一个迭代器，这样就可以使用它所指向的元素。++运算符使得迭代器指向容器中的下一个元素（和数组中指针递增后指向数组的下一个元素类似）。

STL 首类容器提供了成员函数 begin 和 end。函数 begin 返回一个指向容器中第一个元素的迭代器，函数 end 返回一个指向容器中最后一个元素的下一个元素（这个元素并不存在，常用于判断是否到达了容器的结束位仅）的迭代器。如果迭代器 i 指向一个特定的元素，那么 ++i 指向这个元素的下一个元素。* i 指代的是 i 指向的元素。从函数 end 中返回的迭代器只在相等或不等的比较中使用，来判断这个“移动的迭代器”（在这里指 i）是否到达了容器的末端。

使用一个 iterator 对象来指向一个可以修改的容器元素，使用一个 const_iterator 对象来指向一个不能修改的容器元素。

类型	描述
随机访问迭代器(random access)	在双向迭代器基础上增加了直接访问容器中任意元素的功能，即可以向前或向后跳转任意个元素
双向迭代器(bidirectional)	在前向迭代器基础上增加了向后移动的功能。支持多遍扫描算法
前向迭代器(forward)	综合输入和输出迭代器的功能，并能保持它们在容器中的位置（作为状态信息），可以使用同一个迭代器两次遍历一个容器（称为多遍扫描算法）
输出迭代器(output)	用于将元素写入容器。输出迭代器每次只能向前移动一个元素。输出迭代器只支持一遍扫描算法，不能使用相同的输出迭代器两次遍历一个序列容器
输入迭代器(input)	用于从容器读取元素。输入迭代器每次只能向前移动一个元素。输入迭代器只支持一遍扫描算法，不能使用相同的输入迭代器两次遍历一个序列容器

每种容器所支持的迭代器类型决定了这种容器是否可以在指定的 STL 算法中使用。支持随机访问迭代器的容器可用于所有的 STL 算法（除了那些需要改变容器大小的算法，这样的算法不能在数组和 array 对象中使用）。指向数组的指针可以代替迭代器用于几乎所有的 STL 算法中，包括那些要求随机访问迭代器的算法。下表显示了每种 STL 容器所支持的迭代器类型。注意，vector、deque、list、set、multiset、map、multimap以及 string 和数组都可以使用迭代器遍历。

容器	支持的迭代器类型	容器	支持的迭代器类型
vector	随机访问迭代器	set	双向迭代器
array	随机访问迭代器	multiset	双向迭代器
deque	随机访问迭代器	map	双向迭代器
list	双向迭代器	multimap	双向迭代器
forward_list	前向迭代器	unordered_set	双向迭代器
stack	不支持迭代器	unordered_multiset	双向迭代器
queue	不支持迭代器	unordered_map	双向迭代器
priority_queue	不支持迭代器	unordered_multimap	双向迭代器

下表显示了在 STL 容器的类定义中出现的几种预定义的迭代器 typedef。不是每种 typedef 都出现在每个容器中。我们使用常量版本的迭代器来访问只读容器或不应该被更改的非只读容器，使用反向迭代器来以相反的方向访问容器。

为迭代器预先定义的typedef	++的方向	读写能力
iterator	向前	读/写
const_iterator	向前	读
reverse_iterator	向后	读/写
const_reverse_iterator	向后	读

下表显示了可作用在每种迭代器上的操作。除了给出的对于所有迭代器都有的运算符，迭代器还必须提供默认构造函数、拷贝构造函数和拷贝赋值操作符。前向迭代器支持 ++ 和所有的输入和输出迭代器的功能。双向迭代器支持--操作和前向迭代器的功能。随机访问迭代器支持所有在表中给出的操作。另外，对于输入迭代器和输出迭代器，不能在保存迭代器之后再使用保存的值。

迭代器操作	描述
适用所有迭代器的操作	
++p	前置自增迭代器
p++	后置自增迭代器
p=p1	将一个迭代器赋值给另一个迭代器
输入迭代器	
*p	间接引用一个迭代器
p->m	使用迭代器读取元素m
p==p1	比较两个迭代器是否相等
p!=p1	比较两个迭代器是否不相等
输出迭代器	
*p	间接引用一个迭代器
p=p1	把一个迭代器赋值给另一个
前向迭代器	前向迭代器提供了输入和输出迭代器的所有功能
双向迭代器	
-p	q
p-	后置自减迭代器
随机访问迭代器	
p+=i	迭代器p前进i个位置
p-=i	迭代器p后退i个位置
p+i	在迭代器p 的位置上前进i个位置
p-i	在迭代器p的位置上后退i个位置
p-p1	表达式的值是一个整数，它代表同一个容器中两个元素间的距离
p[i]	返回与迭代器p的位置相距i的元素
p<p1	若迭代器p小于p1(即容器中p在p1前) 则返回 true, 否则返回 false
p<=p1	若迭代器p小于或等于p1 (即容器中p 在p1前或位置相同) 则返回 true, 否则返回 false
p>p1	若迭代器p 大于p1(即容器中p在p1后) 则返回true, 否则返回false
p>=p1	若迭代器p大于或等于p1(即容器中p在p1后或位置相同) 则返回 true, 否则返回 false

迭代器操作	描述

4.3 map与unordered_map（红黑树VS哈希表）

C++11 增加了无序容器 unordered_map/unordered_multimap 和 unordered_set/unordered_multiset，由于这些容器中的元素是不排序的，因此，比有序容器 map/multimap 和 set/multiset 效率更高。map 和 set 内部是红黑树，在插入元素时会自动排序，而无序容器内部是散列表（Hash Table），通过哈希（Hash），而不是排序来快速操作元素，使得效率更高。由于无序容器内部是散列表，因此无序容器的 key 需要提供 hash_value 函数，其他用法和 map/set 的用法是一样的。不过对于自定义的 key，需要提供 Hash 函数和比较函数。

4.3.1 map和unordered_map的差别

需要引入的头文件不同

```
map: #include < map >
unordered_map: #include < unordered_map >
```

内部实现机理不同

- **map**：map内部实现了一个红黑树（红黑树是非严格平衡二叉搜索树，而AVL是严格平衡二叉搜索树），红黑树具有自动排序的功能，因此map内部的所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。
- **unordered_map**：unordered_map内部实现了一个哈希表（也叫散列表，通过把关键码值映射到Hash表中一个位置来访问记录，查找的时间复杂度可达到O(1)，其在海量数据处理中有着广泛应用）。因此，其元素的排列顺序是无序的。

4.3.2 优缺点以及适用处

map:

1. 优点:

- 有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作
- 红黑树，内部实现一个红黑书使得map的很多操作在lg n的时间复杂度下就可以实现，因此效率非常的高

2. 缺点:

- 空间占用率高，因为map内部实现了红黑树，虽然提高了运行效率，但是因为每一个节点都需要额外保存父节点、孩子节点和红/黑性质，使得每一个节点都占用大量的空间

3. 适用处：
对于那些有顺序要求的问题，用map会更高效一些

unordered_map:

1. 优点： 因为内部实现了哈希表，因此其查找速度非常的快
2. 缺点： 哈希表的建立比较耗费时间
3. 适用处： 对于查找问题， unordered_map会更加高效一些， 因此遇到查找问题， 常会考虑一下用 unordered_map

4.3.3 总结

1. 内存占有率的问题就转化成红黑树 VS hash表， 还是unordered_map占用的内存要高。
2. 但是unordered_map执行效率要比map高很多
3. 对于unordered_map或unordered_set容器， 其遍历顺序与创建该容器时输入的顺序不一定相同， 因为遍历是按照哈希表从前往后依次遍历的

4.4 更多STL使用范例

C++ 参考手册 <https://zh.cppreference.com/w/cpp>

重点C++11起的容器

C++ 参考手册

C++98, C++03, C++11, C++14, C++17, C++20, C++23 编译器支持 C++11, C++14, C++17, C++20, C++23		
<div>编译器支持 自立实现</div> <div>语言</div> <div>基本概念</div> <div>关键词</div> <div>预处理器</div> <div>表达式</div> <div>声明</div> <div>初始化</div> <div>函数</div> <div>语句</div> <div>类</div> <div>重载</div> <div>模板</div> <div>异常</div> <div>头文件</div> <div>具名要求</div> <div>功能特性测试宏 (C++20)</div> <div>语言支持库</div> <div>类型支持 - 特性 (C++11)</div> <div>程序工具</div> <div>线程支持 (C++20)</div> <div>三路比较 (C++20)</div> <div>numeric_limits - type info</div> <div>initializer_list (C++11)</div>	<div>概念库 (C++20)</div> <div>诊断库</div> <div>通用工具库</div> <div>智能指针与分配器</div> <div>unique_ptr (C++11)</div> <div>shared_ptr (C++11)</div> <div>日期和时间</div> <div>函数对象 - hash (C++11)</div> <div>字符串转换 (C++17)</div> <div>工具函数</div> <div>pair - tuple (C++11)</div> <div>optional (C++17) - any (C++17)</div> <div>variant (C++17) - format (C++20)</div> <div>字符串库</div> <div>basic_string</div> <div>basic_string_view (C++17)</div> <div>空终止字符串:</div> <div>章节 - 多字节 - 宽</div> <div>容器库</div> <div>array (C++11) - vector - deque</div> <div>map - unordered_map (C++11)</div> <div>set - unordered_set (C++11)</div> <div>priority_queue - span (C++20)</div> <div>其他容器:</div> <div>顺序 - 关联</div> <div>无序关联 - 适配器</div>	<div>迭代器库</div> <div>范围库 (C++20)</div> <div>算法库</div> <div>数值库</div> <div>常用数学函数</div> <div>数学特殊函数 (C++17)</div> <div>数值算法</div> <div>伪随机数生成</div> <div>浮点环境 (C++11)</div> <div>complex - valarray</div> <div>输入/输出库</div> <div>基于流的 I/O</div> <div>同步输出 (C++20)</div> <div>I/O 操纵符</div> <div>文件系统库 (C++17)</div> <div>本地化库</div> <div>正则表达式库 (C++11)</div> <div>basic_regex - 算法</div> <div>原子操作库 (C++11)</div> <div>atomic - atomic_flag</div> <div>atomic_ref (C++20)</div> <div>线程支持库 (C++11)</div> <div>thread - mutex</div> <div>condition_variable</div>

五 正则表达式

还有正则表达式，也有范例讲解：<https://zh.cppreference.com/w/cpp/regex>

具体范例见代码。

参考

深入应用C++11：代码优化与工程级应用

[\(147条消息\) 为什么多线程读写 shared_ptr 要加锁? 陈硕的博客-CSDN博客多线程读写](#)

现代C++教程

<https://mp.weixin.qq.com/s/5hy-wkqi0lxFBynTl4Bi6g>