02讲自己动手，实现C++的智能指针

你好，我是吴咏炜。

上一讲，我们描述了一个某种程度上可以当成智能指针用的类 `shape_wrapper`。使用那个智能指针，可以简化资源的管理，从根本上消除资源（包括内存）泄漏的可能性。这一讲我们就来进一步讲解，如何将 `shape_wrapper` 改造成一个完整的智能指针。你会看到，智能指针本质上并不神秘，其实就是 RAII 资源管理功能的自然展现而已。

在学完这一讲之后，你应该会对 C++ 的 `unique_ptr` 和 `shared_ptr` 的功能非常熟悉了。同时，如果你今后要创建类似的资源管理类，也不会是一件难事。

回顾

我们上一讲给出了下面这个类：

```
class shape_wrapper {
public:
  explicit shape_wrapper(
    shape* ptr = nullptr)
    : ptr_(ptr) {}
  ~shape_wrapper()
  {
    delete ptr_;
  }
  shape* get() const { return ptr_; }

private:
  shape* ptr_;
};
```

这个类可以完成智能指针的最基本的功能：对超出作用域的对象进行释放。**但它缺了点东西：**

1. 这个类只适用于 shape 类
2. 该类对象的行为不够像指针
3. 拷贝该类对象会引发程序行为异常

下面我们来逐一看一下怎么弥补这些问题。

模板化和易用性

要让这个类能够包装任意类型的指针，我们需要把它变成一个类模板。这实际上相当容易：

```
template <typename T>
class smart_ptr {
public:
  explicit smart_ptr(T* ptr = nullptr)
    : ptr_(ptr) {}
  ~smart_ptr()
  {
    delete ptr_;
  }
  T* get() const { return ptr_; }
private:
  T* ptr_;
};
```

和 `shape_wrapper` 比较一下，我们就是在开头增加模板声明 `template <typename T>`，然后把代码中的 `shape` 替换成模板参数 `T` 而已。这些修改非常简单自然吧？模板本质上并不是一个很复杂的概念。这个模板使用也很简单，把原来的 `shape_wrapper` 改成 `smart_ptr<shape>` 就行。

目前这个 smart_ptr 的行为还是和指针有点差异的：

- 它不能用 * 运算符解引用
- 它不能用 -> 运算符指向对象成员
- 它不能像指针一样用在布尔表达式里

不过，这些问题也相当容易解决，加几个成员函数就可以：

```
template <typename T>
class smart_ptr {
public:
  …
  T& operator*() const { return *ptr_; }
  T* operator->() const { return ptr_; }
  operator bool() const { return ptr_; }
}
```

拷贝构造和赋值

拷贝构造和赋值，我们暂且简称为拷贝，这是个比较复杂的问题了。关键还不是实现问题，而是我们该如何定义其行为。假设有下面的代码：

```
smart_ptr<shape> ptr1{create_shape(shape_type::circle)};
smart_ptr<shape> ptr2{ptr1};
```

对于第二行，究竟应当让编译时发生错误，还是可以有一个更合理的行为？我们来逐一检查一下各种可能性。

最简单的情况显然是禁止拷贝。我们可以使用下面的代码：

```
template <typename T>
class smart_ptr {
  …
  smart_ptr(const smart_ptr&)
    = delete;
  smart_ptr& operator=(const smart_ptr&)
    = delete;
  …
};
```

禁用这两个函数非常简单，但却解决了一种可能出错的情况。否则，`smart_ptr<shape> ptr2{ptr1};` 在编译时不会出错，但在运行时却会有未定义行为——由于会对同一内存释放两次，通常情况下会导致程序崩溃。

我们是不是可以考虑在拷贝智能指针时把对象拷贝一份？不行，通常人们不会这么用，因为使用智能指针的目的就是要减少对象的拷贝啊。何况，虽然我们的指针类型是 shape，但实际指向的却应该是 circle 或 triangle 之类的对象。在 C++ 里没有像 Java 的 clone 方法这样的约定；一般而言，并没有通用的方法可以通过基类的指针来构造出一个子类的对象来。

我们要么试试在拷贝时转移指针的所有权？大致实现如下：

```
template <typename T>
class smart_ptr {
  …
  smart_ptr(smart_ptr& other)
  {
    ptr_ = other.release();
  }
  smart_ptr& operator=(smart_ptr& rhs)
  {
    smart_ptr(rhs).swap(*this);
    return *this;
  }
  …
  T* release()
  {
    T* ptr = ptr_;
    ptr_ = nullptr;
    return ptr;
  }
  void swap(smart_ptr& rhs)
  {
    using std::swap;
    swap(ptr_, rhs.ptr_);
  }
  …
};
```

在拷贝构造函数中，通过调用 `other` 的 `release` 方法来释放它对指针的所有权。在赋值函数中，则通过拷贝构造产生一个临时对象并调用 `swap` 来交换对指针的所有权。实现上是不复杂的。

如果你学到的赋值函数还有一个类似于 `if (this != &rhs)` 的判断的话，那种用法更啰嗦，而且异常安全性不够好——如果在赋值过程中发生异常的话，this 对象的内容可能已经被部分破坏了，对象不再处于一个完整的状态。

目前这种惯用法（见参考资料 [1]）则保证了强异常安全性：赋值分为拷贝构造和交换两步，异常只可能在第一步发生；而第一步如果发生异常的话，this 对象完全不受任何影响。无论拷贝构造成功与否，结果只有赋值成功和赋值没有效果两种状态，而不会发生因为赋值破坏了当前对象这种场景。

如果你觉得这个实现还不错的话，那恭喜你，你达到了 C++ 委员会在 1998 年时的水平：上面给出的语义本质上就是 C++98 的 `auto_ptr` 的定义。如果你觉得这个实现很别扭的话，也恭喜你，因为 C++ 委员会也是这么觉得的：`auto_ptr` 在 C++17 时已经被正式从 C++ 标准里删除了。

上面实现的最大问题是，它的行为会让程序员非常容易犯错。一不小心把它传递给另外一个 `smart_ptr`，你就不再拥有这个对象了……

“移动”指针?

在下一讲我们将完整介绍一下移动语义。这一讲，我们先简单看一下 smart_ptr 可以如何使用“移动”来改善其行为。

我们需要对代码做两处小修改：

```
template <typename T>
class smart_ptr {
  …
  smart_ptr(smart_ptr&& other)
  {
    ptr_ = other.release();
  }
  smart_ptr& operator=(smart_ptr rhs)
  {
    rhs.swap(*this);
    return *this;
  }
  …
};
```

看到修改的地方了吗？我改了两个地方：

- 把拷贝构造函数中的参数类型 smart_ptr& 改成了 smart_ptr&&；现在它成了移动构造函数。
- 把赋值函数中的参数类型 smart_ptr& 改成了 smart_ptr，在构造参数时直接生成新的智能指针，从而不再需要在函数体中构造临时对象。现在赋值函数的行为是移动还是拷贝，完全依赖于构造参数时走的是移动构造还是拷贝构造。

根据 C++ 的规则，如果我提供了移动构造函数而没有手动提供拷贝构造函数，那后者自动被禁用（记住，C++ 里那些复杂的规则也是为方便编程而设立的）。于是，我们自然地得到了以下结果：

```
smart_ptr<shape> ptr1{create_shape(shape_type::circle)};
smart_ptr<shape> ptr2{ptr1};              // 编译出错
smart_ptr<shape> ptr3;
ptr3 = ptr1;                              // 编译出错
ptr3 = std::move(ptr1);                   // OK，可以
smart_ptr<shape> ptr4{std::move(ptr3)};   // OK，可以
```

这个就自然多了。

这也是 C++11 的 unique_ptr 的基本行为。

子类指针向基类指针的转换

哦，我撒了一个小谎。不知道你注意到没有，一个 circle* 是可以隐式转换成 shape* 的，但上面的 smart_ptr<circle> 却无法自动转换成 smart_ptr<shape>。这个行为显然还是不够“自然”。

不过，只需要额外加一点模板代码，就能实现这一行为。在我们目前给出的实现里，只需要修改我们的移动构造函数一处即可

——这也算是我们让赋值函数使用拷贝/移动构造函数的好处了。

```
template <typename U>
smart_ptr(smart_ptr<U>&& other)
{
  ptr_ = other.release();
}
```

这样，我们自然而然利用了指针的转换特性：现在 `smart_ptr<circle>` 可以移动给 `smart_ptr<shape>`，但不能移动给 `smart_ptr<triangle>`。不正确的转换会在代码编译时直接报错。

至于非隐式的转换，因为本来就是要写特殊的转换函数的，我们留到这一讲的最后再讨论。

引用计数

`unique_ptr` 算是一种较为安全的智能指针了。但是，一个对象只能被单个 `unique_ptr` 所拥有，这显然不能满足所有使用场合的需求。一种常见的情况是，多个智能指针同时拥有一个对象；当它们全部都失效时，这个对象也同时会被删除。这也就是 `shared_ptr` 了。

`unique_ptr` 和 `shared_ptr` 的主要区别如下图所示：

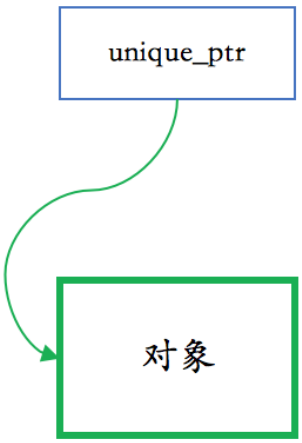

图1a：unique_ptr

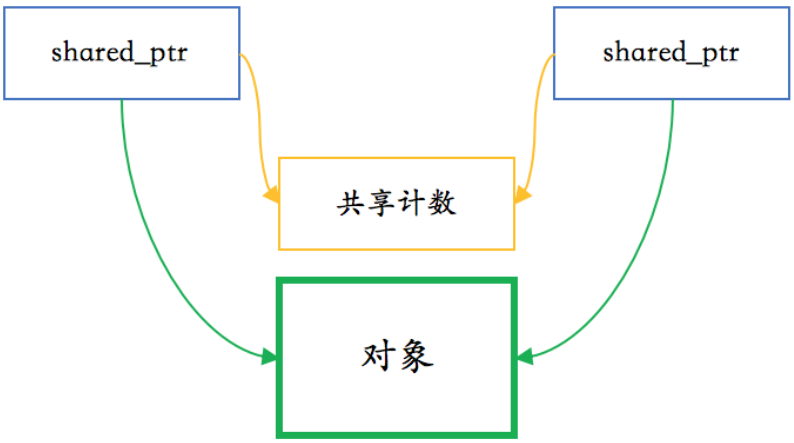

图1b：shared_ptr

多个不同的 `shared_ptr` 不仅可以共享一个对象，在共享同一对象时也需要同时共享同一个计数。当最后一个指向对象（和共享计数）的 `shared_ptr` 析构时，它需要删除对象和共享计数。我们下面就来实现一下。

我们先来写出共享计数的接口：

```
class shared_count {
public:
  shared_count();
  void add_count();
  long reduce_count();
  long get_count() const;
};
```

这个 shared_count 类除构造函数之外有三个方法：一个增加计数，一个减少计数，一个获取计数。注意上面的接口增加计数不需要返回计数值；但减少计数时需要返回计数值，以供调用者判断是否它已经是最后一个指向共享计数的 shared_ptr 了。由于真正多线程安全的版本需要用到我们目前还没学到的知识，我们目前先实现一个简单化的版本：

```
class shared_count {
public:
  shared_count() : count_(1) {}
  void add_count()
  {
    ++count_;
  }
  long reduce_count()
  {
    return --count_;
  }
  long get_count() const
  {
    return count_;
  }

private:
  long count_;
};
```

现在我们可以实现我们的引用计数智能指针了。首先是构造函数、析构函数和私有成员变量：

```
template <typename T>
class smart_ptr {
public:
  explicit smart_ptr(T* ptr = nullptr)
    : ptr_(ptr)
  {
    if (ptr) {
      shared_count_ =
        new shared_count();
    }
  }
  ~smart_ptr()
  {
    if (ptr_ &&
      !shared_count_
         ->reduce_count()) {
      delete ptr_;
      delete shared_count_;
    }
  }

private:
  T* ptr_;
  shared_count* shared_count_;
};
```

构造函数跟之前的主要不同点是会构造一个 shared_count 出来。析构函数在看到 ptr_ 非空时（此时根据代码逻辑，shared_count 也必然非空），需要对引用数减一，并在引用数降到零时彻底删除对象和共享计数。原理就是这样，不复杂。

当然，我们还有些细节要处理。为了方便实现赋值（及其他一些惯用法），我们需要一个新的 swap 成员函数：

```
  void swap(smart_ptr& rhs)
  {
    using std::swap;
    swap(ptr_, rhs.ptr_);
    swap(shared_count_,
         rhs.shared_count_);
  }
```

赋值函数可以跟前面一样，保持不变，但拷贝构造和移动构造函数是需要更新一下的：

```
template <typename U>
smart_ptr(const smart_ptr<U>& other)
{
  ptr_ = other.ptr_;
  if (ptr_) {
    other.shared_count_
      ->add_count();
    shared_count_ =
      other.shared_count_;
  }
}
template <typename U>
smart_ptr(smart_ptr<U>&& other)
{
  ptr_ = other.ptr_;
  if (ptr_) {
    shared_count_ =
      other.shared_count_;
    other.ptr_ = nullptr;
  }
}
```

除复制指针之外，对于拷贝构造的情况，我们需要在指针非空时把引用数加一，并复制共享计数的指针。对于移动构造的情况，我们不需要调整引用数，直接把 `other.ptr_` 置为空，认为 `other` 不再指向该共享对象即可。

不过，上面的代码有个问题：它不能正确编译。编译器会报错，像：

> fatal error: ‘ptr_’ is a private member of ‘smart_ptr<circle>’

错误原因是模板的各个实例间并不天然就有 friend 关系，因而不能互访私有成员 `ptr_` 和 `shared_count_`。我们需要在 `smart_ptr` 的定义中显式声明：

```
template <typename U>
friend class smart_ptr;
```

此外，我们之前的实现（类似于单一所有权的 `unique_ptr` ）中用 `release` 来手工释放所有权。在目前的引用计数实现中，它就不太合适了，应当删除。但我们要加一个对调试非常有用的函数，返回引用计数值。定义如下：

```
  long use_count() const
  {
    if (ptr_) {
      return shared_count_
        ->get_count();
    } else {
      return 0;
    }
  }
```

这就差不多是一个比较完整的引用计数智能指针的实现了。我们可以用下面的代码来验证一下它的功能正常：

```
class shape {
public:
  virtual ~shape() {}
};

class circle : public shape {
public:
  ~circle() { puts("~circle()"); }
};

int main()
{
  smart_ptr<circle> ptr1(new circle());
  printf("use count of ptr1 is %ld\n",
         ptr1.use_count());
  smart_ptr<shape> ptr2;
  printf("use count of ptr2 was %ld\n",
         ptr2.use_count());
  ptr2 = ptr1;
  printf("use count of ptr2 is now %ld\n",
         ptr2.use_count());
  if (ptr1) {
    puts("ptr1 is not empty");
  }
}
```

这段代码的运行结果是：

```
use count of ptr1 is 1
use count of ptr2 was 0
use count of ptr2 is now 2
```

```
ptr1 is not empty
~circle()
```

上面我们可以看到引用计数的变化，以及最后对象被成功删除。

指针类型转换

对应于 C++ 里的不同的类型强制转换：

- static_cast
- reinterpret_cast
- const_cast
- dynamic_cast

智能指针需要实现类似的函数模板。实现本身并不复杂，但为了实现这些转换，我们需要添加构造函数，允许在对智能指针内部的指针对象赋值时，使用一个现有的智能指针的共享计数。如下所示：

```
  template <typename U>
  smart_ptr(const smart_ptr<U>& other,
            T* ptr)
  {
    ptr_ = ptr;
    if (ptr_) {
      other.shared_count_
        ->add_count();
      shared_count_ =
        other.shared_count_;
    }
  }
```

这样我们就可以实现转换所需的函数模板了。下面实现一个 `dynamic_pointer_cast` 来示例一下：

```
template <typename T, typename U>
smart_ptr<T> dynamic_pointer_cast(
  const smart_ptr<U>& other)
{
  T* ptr =
    dynamic_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}
```

在前面的验证代码后面我们可以加上：

```
  smart_ptr<circle> ptr3 =
    dynamic_pointer_cast<circle>(ptr2);
  printf("use count of ptr3 is %ld\n",
         ptr3.use_count());
```

编译会正常通过，同时能在输出里看到下面的结果：

> use count of ptr3 is 3

最后，对象仍然能够被正确删除。这说明我们的实现是正确的。

代码列表

为了方便你参考，下面我给出了一个完整的 `smart_ptr` 代码列表：

```
#include <utility>  // std::swap

class shared_count {
public:
  shared_count() noexcept
    : count_(1) {}
  void add_count() noexcept
  {
    ++count_;
  }
  long reduce_count() noexcept
  {
    return --count_;
  }
  long get_count() const noexcept
  {
    return count_;
  }

private:
  long count_;
};

template <typename T>
class smart_ptr {
public:
  template <typename U>
  friend class smart_ptr;

  explicit smart_ptr(T* ptr = nullptr)
```

```
  : ptr_(ptr)
{
  if (ptr) {
    shared_count_ =
      new shared_count();
  }
}
~smart_ptr()
{
  printf("~smart_ptr(): %p\n", this);
  if (ptr_ &&
    !shared_count_
       ->reduce_count()) {
    delete ptr_;
    delete shared_count_;
  }
}

template <typename U>
smart_ptr(const smart_ptr<U>& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    other.shared_count_->add_count();
    shared_count_ = other.shared_count_;
  }
}
template <typename U>
smart_ptr(smart_ptr<U>&& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    shared_count_ =
      other.shared_count_;
    other.ptr_ = nullptr;
  }
}
template <typename U>
smart_ptr(const smart_ptr<U>& other,
          T* ptr) noexcept
{
  ptr_ = ptr;
  if (ptr_) {
    other.shared_count
```

```
    other.shared_count_
      ->add_count();
    shared_count_ =
      other.shared_count_;
  }
}
smart_ptr&
operator=(smart_ptr rhs) noexcept
{
  rhs.swap(*this);
  return *this;
}

T* get() const noexcept
{
  return ptr_;
}
long use_count() const noexcept
{
  if (ptr_) {
    return shared_count_
      ->get_count();
  } else {
    return 0;
  }
}
void swap(smart_ptr& rhs) noexcept
{
  using std::swap;
  swap(ptr_, rhs.ptr_);
  swap(shared_count_,
       rhs.shared_count_);
}

T& operator*() const noexcept
{
  return *ptr_;
}
T* operator->() const noexcept
{
  return ptr_;
}
operator bool() const noexcept
{
```

```
    return ptr_;
  }

private:
  T* ptr_;
  shared_count* shared_count_;
};

template <typename T>
void swap(smart_ptr<T>& lhs,
          smart_ptr<T>& rhs) noexcept
{
  lhs.swap(rhs);
}

template <typename T, typename U>
smart_ptr<T> static_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = static_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> reinterpret_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = reinterpret_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> const_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = const_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> dynamic_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = dynamic_cast<T*>(other.get());
```

```
    return smart_ptr<T>(other, ptr);
}
```

如果你足够细心的话，你会发现我在代码里加了不少 `noexcept`。这对这个智能指针在它的目标场景能正确使用是十分必要的。我们会在下面的几讲里回到这个话题。

内容小结

这一讲我们从 `shape_wrapper` 出发，实现了一个基本完整的带引用计数的智能指针。这个智能指针跟标准的 `shared_ptr` 比，还缺了一些东西（见参考资料 [2]），但日常用到的智能指针功能已经包含在内。现在，你应当已经对智能指针有一个较为深入的理解了。

课后思考

这里留几个问题，你可以思考一下：

1. 不查阅 `shared_ptr` 的文档，你觉得目前 `smart_ptr` 应当添加什么功能吗？
2. 你想到的功能在标准的 `shared_ptr` 里吗？
3. 你觉得智能指针应该满足什么样的线程安全性？

欢迎留言和我交流你的看法。

参考资料

[1] Stack Overflow, GManNickG's answer to "What is the copy-and-swap idiom?". https://stackoverflow.com/a/3279550/816999

[2] cppreference.com, "std::shared_ptr". https://en.cppreference.com/w/cpp/memory/shared_ptr

精选留言

frazer

有点吃力了，得反复看几遍

2019-11-26 16:45

作者回复

没关系。我打赌你看的时间肯定没我写稿的时间长。

2019-11-26 18:50

W.jyao

对不熟悉C++ 11的程序员来看，有的地方不是很懂

2019-11-26 14:27

作者回复

特别有困难的点可以提出。大部分概念的引入我应该都是有解释的。

2019-11-26 18:32

流浪地球

老师您好，问一个比较基础的问题，我理解这个语句

smart_ptr<shape> ptr1{create_shape(shape_type::circle)}; 是调用ptr1的拷贝构造函数。

为什么{create_shape(shape_type::circle)}是使用大括号，不应该是小括号吗？

谢谢

2019-11-26 10:19

作者回复

嘻嘻，我在偷偷地塞进C++11的语法。对象初始化可以统一用大括号。（小括号这儿也行。）

2019-11-26 18:01

yuchen

有深度的专栏，不错。市面上讲解C++的课程一般太基础了。这一章推荐读者可以看看《Professional C＋＋ 4th edition》第九章。

2019-11-26 22:07

作者回复

谢谢。

《Professional C++》之前没看过，扫了两眼，觉得内容不错，推荐。内容还挺多挺深的，适合决心在 C++ 上深入的同学。

2019-11-27 08:08

小美

计数器线程安全是不是更好点

2019-11-26 10:08

作者回复

应该的。但我们还没学到通用的C++并发编程呢。

2019-11-26 18:07

Lilin

才第二节，就有点吃力了。这篇专栏真是满满的干货

2019-11-26 09:03

作者回复

希望是难，但还能看得下去。哈哈。

2019-11-26 18:09

hdongdong123

真的好难啊，呜呜呜

2019-11-27 01:48

作者回复

一遍看不懂，就再看一遍。所有的代码自己试验一下。

学习无捷径。掌握 C++ 不是 30 个课时能解决的事情。一万小时理论对于任何复杂领域都是基本适用的。

2019-11-27 08:26

小林coding

见一些多线程使用计数器的时候，会用std::atomic<int> num的方式来定义计数器，但是没有理解到为啥std::atomic是如何保证线程安全的?

2019-11-26 20:24

作者回复

后面会讲到。简单来说，atomic变量的读取、写入、增、减都会翻译成目标系统上的原子操作指令。

2019-11-26 21:03

nullptr

我觉得effective c++也有相关说明

2019-11-27 18:36

作者回复

不太明白你的评论。不过呢，如果谁读过Meyers的四本Effective系列，这个专栏能让他学习的新知识可能就不太多了。

2019-11-27 20:37

虫 二

应该还要添加下标访问，比较的重载，看得有些吃力，还是得坚持下

2019-11-27 16:18

作者回复

下标？现在只管理一个对象，下标有什么用?

2019-11-27 20:32

糖

吴老师好，smart_ptr的拷贝构造函数和赋值运算符是否可以同种方法解决，比如，上面赋值运算符中swap部分的代码改成release：

```
smart_ptr& operator=(smart_ptr& other) {
data = other.release();
```

return *this;

}

这段代码是正确的吗？对于单纯实现smart_ptr有缺陷的地方吗？

2019-11-27 11:23

作者回复

不正确，this == &other 时会自我销毁。

2019-11-27 20:25

中年男子

智能指针目前看有两个成员，一个是指向对象的指针，一个是引用计数，多线程读的话只需考虑引用计数的同步即可，可以加锁或者atomic原子操作，涉及写话还要给指针加锁

2019-11-26 23:55

作者回复

一般实现中，只考虑对引用计数的原子操作，那是必须的。并发修改智能指针不是主流用法，需要自行实现或外部加锁。

2019-11-27 08:21

qinsi

不知道是不是因为Rust刚出的时候C++11还没出，所以Rust里默认赋个值就要move ownership

2019-11-26 22:14

作者回复

抱歉，Rust不熟……

2019-11-27 08:08

夜空中最亮的星（华仔）

老师您真有近30年的编程经验吗？我咋看你就二十几岁的样子啊？

2019-11-26 21:38

作者回复

哈哈，照骗嘛。谢谢。

2019-11-27 07:47

姜姜

这个smart_ptr目前已经实现了:获取所有权，共享所有权，拷贝赋值语意，计数器，自动增减等功能。

还缺少: 转移所有权reset的操作(我记得reset有三种重载方式); 获取原始指针方法，重写解引用*，重写指向操作符->，重写比较操作符，获取引用计数器值的方法，判断当前引用是否为1的方法，以及线程安全的设计。

最后，shared_ptr本身还需要解决“循环引用”，“自引用”的问题，一般需要搭配弱指针weak_ptr使用。

老师，不知我回答的是否全面，有无差错?

2019-11-26 21:22

作者回复

很全。有几个小问题哈（像reset有几个重载不算 ）：

获取原始指针的 get 方法我提供了；

* 和 -> 代码里是提供的；

引用计数器值也是有的。

实践层面，最大的区别，除了不支持弱引用外，可能是不支持自定义删除器了。

2019-11-27 07:47

微秒

看了这两章，对C++的智能指针和内存管理有了较清晰的认识了，完全不同于java和python啊

2019-11-26 16:54

作者回复

缺省值语义和只有引用语义，区别确实是很大的。

2019-11-26 18:43

bearlu

老师，这边有源码？可以拿来调试一下。

2019-11-26 14:00

作者回复

完整代码在结尾处啊。拷贝粘贴大法就可以了。

2019-11-26 18:31

Txz

一遍下来，还是觉得理解的不够，只能反复阅读。

2019-11-25 23:14

作者回复

没关系的。学习就是要多读多写多练。

2019-11-26 09:43

天天

面试高频题目

2019-11-25 22:14

作者回复

2019-11-25 22:49