### shared\_ptr实现剖析

###### shared\_ptr简介：

几乎每一个稍有分量的程序都需要“在相同时间的多处地点处理或使用对象”的能力。为此，你必须在程序的多个地点指向（refer to）同一对象。虽然C++语言提供了引用（reference）和指针（pointer），但还是不够，因为你往往必须确保当“指向对象”的最末一个引用（reference）被删除时该对象本身也被删除，毕竟对象被删除时析构函数可以要求某些操作，例如释放内存或是归还资源等等。

所以我们需要“当对象再也不被使用时就被清理”的语义。Class shared\_ptr提供了这样的共享式拥有语义。也就是说，多个shared\_ptr可以共享（或说拥有）同一对象。对象的最末一个拥有者有责任销毁对象，并清理与该对象相关的所有资源。

如果对象以new产生，默认情况下清理工作就由delete完成。但你可以（并且往往必须）定义其他清理办法。你可以定义你自己的析构策略。举个例子，如果你的对象是个以new[]分配而得的array，你必须定义自己的delete[]加以清理。其他例子还包括删除相应资源，如handle、lock、associated temporary file（相关临时文件）等。

总而言之，shared\_ptr的目标就是，在其所指向的对象不再被需要之后（而非之前），自动释放与对象相关的资源。

###### shared\_ptr实现方案？

一种被多次实现过的思想就是所谓的引用计数：对于每个被指向的对象，都保存一个计数，用于代表指向该对象的指针的个数，当计数值减少到0时，就删除此对象。如图1.



图1

###### 计数器在什么地方？

由于我们的想法是计算指向对象的指针的个数，所以把计数器放在对象中是完全合理的。遗憾的是，对于被指向的对象的类型，如果在早期设计的时候，完全未考虑引用计数，那么我们就无法再计数器放入对象中：因为如果对象是封装起来或者不可改变的话，要加入计数器是不可行的。

若被引用计数的对象不能包含计数器，那么就必须将计数器存放在单独的存储区：而且，该存储区的生命期不能比被指向对象的生命期短；也就是说，我们必须动态分配这块存储区。最简单直白的方式，如图2。



图2

###### shared\_ptr的内存对象布局：

shared\_ptr将计数器和指向对象的指针放在一起，组成一个struct（结构体），然后每个shared\_ptr存放一个指向该struct（结构体）的指针，如图3。



图3

###### shared\_ptr设计、接口和说明：

为了支持不同析构策略，shared\_ptr采用了策略模式（strategy pattern），基类sp\_counted\_base管理引用计数器，子类管理指针以及相应的析构策略。如图4。

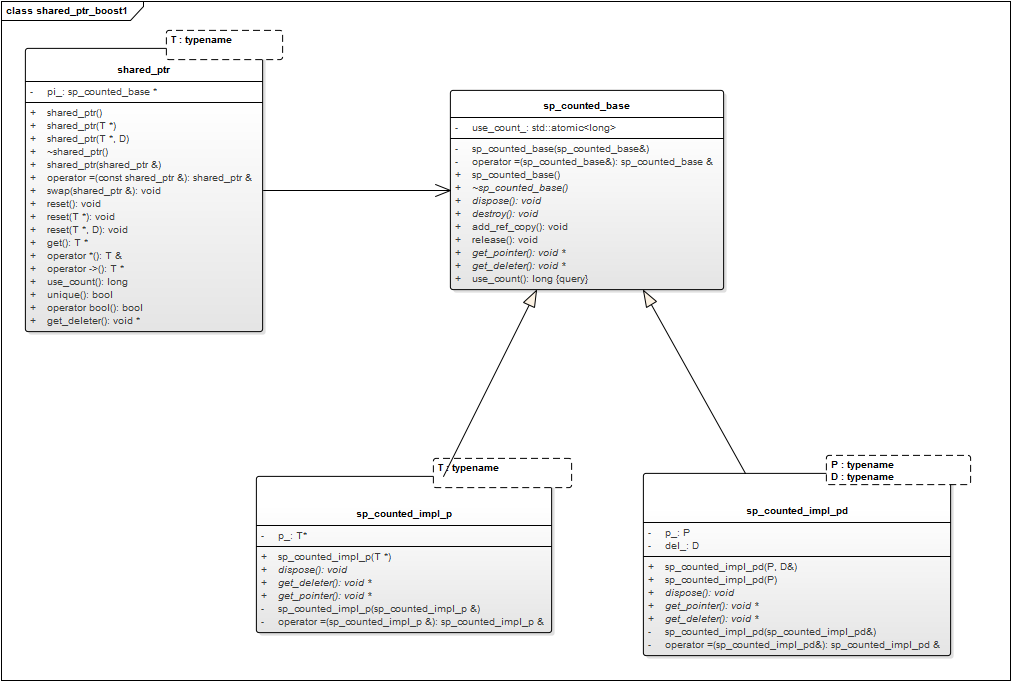


图4

sp\_counted\_base：

引用计数基类，管理引用计数器，不支持拷贝构造和赋值。

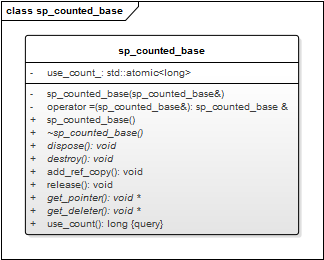


图5

成员变量：

1. use\_count\_：引用计数，原子变量。

构造函数：

1. sp\_counted\_base()：默认构造函数，将引用计数初始化为1。
2. ~sp\_counted\_base()：虚析构函数，do nothing。

成员函数：

1. add\_ref\_copy()：增加共享持有者，将引用计数加1：

++use\_count\_;

1. release()：持有者释放控制，将引用计数减1，然后判断引用计数是否等于0，如果引用计数等于0，则先调用dispose，然后调用destroy：

if(--use\_count\_ == 0) {

dispose();

destroy();

}

1. dispose()：纯虚函数，必须由子类覆盖，语义为释放所指向的对象（及相关的资源）。
2. destroy()：虚函数，默认操作时delete this，释放引用计数本身（及占用的内存资源）：

delete this;

1. get\_pointer()：纯虚函数，获取所指向对象的指针。
2. get\_deleter()：纯虚函数，获取指向的自定义的deleter子对象的指针，如果采用默认的delete来释放所指向的对象，应该返回nullptr。
3. use\_count()：返回当前引用计数的值：

return user\_count\_;

sp\_counted\_impl\_p<T>：

sp\_counted\_base的子类，采用delete表达式释放所持有的对象。

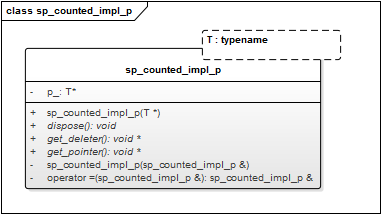


图6

成员变量：

1. p\_：保存所持有对象的指针。

构造函数：

1. sp\_counted\_imp\_p(T \*p)：保存所持有对象的指针。

成员函数：

1. dispose()：用delete释放所持有的对象：

delete p\_;

1. get\_pointer()：返回所持有对象的指针：

return p\_;

1. get\_deleter()：返回nullptr，因为没有自定义的deleter子对象：

return nullptr;

sp\_counted\_impl\_pd<P, D>：

sp\_counted\_base的子类，采用自定义的deleter子对象释放所持有的对象。

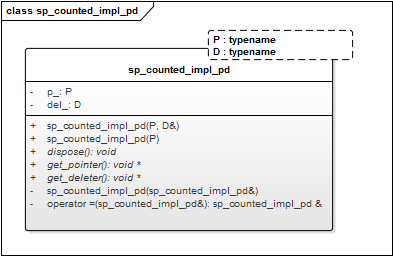


图7

成员变量：

1. p\_：保存所持有对象的指针。
2. del\_：保存自定义的deleter子对象。

构造函数：

1. sp\_counted\_impl\_pd(P p, const D &d)：保存所持有对象的指针和自定义的deleter子对象。

成员函数：

1. dispose()：用自定义的deleter子对象释放所持有的对象（及相关的资源）：

del\_(p\_);

1. get\_pointer()：返回所持有对象的指针：

return p\_;

1. get\_deleter()：返回指向的自定义的deleter子对象的指针：

return &del\_;

shared\_ptr<T>：

拥有共享对象所有权语义的智能指针。

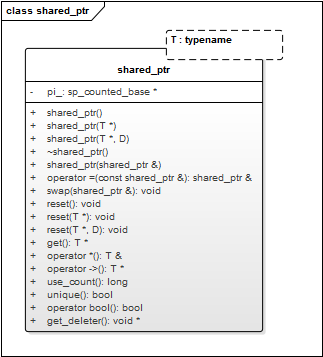


图8

成员变量：

1. pi\_：指向sp\_counted\_base的指针。

构造函数：

1. shared\_ptr()：默认构造函数，构造一个空的shared\_ptr对象，不持有任何对象：

pi\_ = nullptr;

1. shared\_ptr(T \*ptr)：使用一个指针来构造一个shared\_ptr对象，该shared\_ptr对象持有ptr所指向的对象，当最后一个持有者（shared\_ptr对象）释放控制时，使用delete释放所指向的对象，即delete ptr。（对标准的template <typename Y> shared\_ptr(Y \*ptr)的简化。）：

pi\_ = new sp\_counted\_impl\_p<T>(p);

1. template <typename D> shared\_ptr(T \*ptr, D del)：使用一个指针（ptr）和指定的deleter子对象（del）来构造一个shared\_ptr对象，该shared\_ptr持有ptr所指向的对象，当最后一个持有者（shared\_ptr对象）释放控制时，使用指定的deleter子对象释放所持有的对象（等价于del(ptr)），例如deleter可以为定义有void operator ()(T \*ptr)的仿函数。 deleter必须支持复制构造。（对标准的template <typename Y, typename D> shared\_ptr(Y \*ptr, D del)的简化。）：

pi\_ = new sp\_counted\_impl\_pd<T \*, D>(p, del);

1. ~shared\_ptr()：如果当前shared\_ptr对象是最后一个持有者，则释放所持有的对象，否则只是将引用计数减1：

if (pi\_ != nullptr) pi\_->release();

1. shared\_ptr(const shared\_ptr &x)：复制构造函数，使用一个shared\_ptr对象（x）构造一个新的shared\_ptr对象，使得\*this（当前shared\_ptr对象）共享x所持有对象的持有权，直接结果是所持有的对象的引用计数加1：

pi\_ = x.pi\_;

if (pi\_ != nullptr) pi\_->add\_ref\_copy();

成员函数：

1. get()：返回所持有的对象的指针。如果\*this（当前shared\_ptr对象）并不持有任何对象，返回nullptr：

if (pi\_ != nullptr) {

return static\_cast<element\_type \*>(pi\_->get\_pointer());

} else {

return nullptr;

}

1. operator \*()：返回所持有的对象的引用。如果\*this（当前shared\_ptr对象）并不持有任何对象，这个函数的行为将是未定义的：

return \*get();

1. operator ->()：选择运算符（selection operator）会返回get()，因此，如果sp是一个shared\_ptr<T>类的对象，那么表达式sp->member的行为和(sp.get())->member一致。这里，shared\_ptr对象的get()返回不能为空，T必须是一个类、结构或联合（union）类型，并且具有一个名为member的成员：

return get();

1. explicit operator bool()：等价于get() != nullptr。
2. use\_count()：返回当前有多少个shared\_ptr对象持有\*this（当前shared\_ptr对象）所持有的对象。对于一个空的shared\_ptr对象，返回0：

return pi\_ != nullptr ? pi\_->use\_count() : 0;

1. unique()：如果没有其他任何shared\_ptr对象持有\*this（当前shared\_ptr对象）所持有的对象，那么返回true，否则返回false：等价于use\_count() == 1。
2. swap(shared\_ptr &x)：在\*this（当前shared\_ptr对象）和x之间交换所指有的对象：

using std::swap;

swap(this->pi\_, x.pi\_);

1. operator =(const shared\_ptr<T> &x)：赋值运算符，如果\*this（当前shared\_ptr对象）持有了某个对象，那么先释放其控制，然后让\*this（当前shared\_ptr对象）共享x（赋值运算符右边shared\_ptr）持有的对象的持有权：

// typedef shared\_ptr<T> this\_type;

this\_type(x).swap(\*this);

return \*this;

/\*\*

if (this != &x) {

pi\_->release();

pi\_ = x.pi\_;

pi\_->add\_ref\_copy();

}

\*/

1. reset()：释放对所持有的对象的控制。在这个函数返回后，\*this为空：

this\_type().swap(\*this);

/\*\*

if (pi\_) {

pi\_->release();

pi\_ = nullptr;

}

\*/

1. reset(T \*ptr)：释放对所持有的对象的控制，并且让该shared\_ptr对象持有指针ptr所指向的对象：

this\_type(p).swap(\*this);

1. reset(T \*ptr, D del)：释放对所持有的对象的控制，并且让\*this（当前shared\_ptr对象）持有指针ptr所指向的对象，并且使用指定的deleter子对象：

this\_type(p, del).swap(\*this);

1. get\_deleter()：返回指向的自定义的deleter子对象的指针（非shared\_ptr标准对外接口）。

自由函数：

1. template <typename T> swap(shared\_ptr<T> &a, shared\_ptr<T> &b)：等价于a.swap(b)。
2. template <typename D, typename T> D \*get\_deleter(const shared\_ptr<T> &sp)：等价于static\_cast<D \*>(sp.get\_deleter())。
3. operator <<(ostream &out, const shared\_ptr<T> &sp)：等价于out << sp.get()。
4. 关系运算符：简单的比较get()值。

###### shared\_ptr使用和示例：

1. 你可以像使用任何其他指针（pointer）一样地使用shared\_ptr。你可以赋值、拷贝、比较它们，也可以使用操作符\*和->访问其所指向的对象。
2. 我们也可以使用一个空(null)指针来调用shared\_ptr的2）、3）构造函数。这会创建一个比较特殊的对象；从技术上说，它并不是一个空的shared\_ptr对象，虽然它并不持有任何对象（及相关的资源）。
3. 各种构造函数生成的shared\_ptr的内存布局：
4. **shared\_ptr<int> sp0;**



1. **shared\_ptr<int> sp1(new int(3));**



1. **shared\_ptr<int> sp2(nullptr);**



1. **shared\_ptr<int> sp3(new int(3), deleter());**



1. 拷贝构造和reset对引用技术和持有对象生命周期的影响：
2. **shared\_ptr<int> sp0(new int(3));**



1. **shared\_ptr<int> sp1(sp0);**



1. **sp0.reset();**



1. **sp1.reset();**



1. 赋值运算符对持有对象生命周期的影响：
2. **shared\_ptr<int> sp0(new int(3));**



1. **shared\_ptr<int> sp1(new int(4));**



1. **sp0 = sp1;**



1. 不要使用同一个指针来创建两个shared\_ptr对象，如果我们这样做了，那么当这些shared\_ptr对象被销毁时，会对被持有的对象的析构函数调用两次。我们应该对第一个shared\_ptr对象进行拷贝来创建同一资源的第二个shared\_ptr对象。
2. 当两个或更多个被持有对象彼此包含对方的指针时，这些指针就会形成循环，从而导致循环引用。例如，如果节点head包含了一个指向另一个节点N1的shared\_ptr对象，并且节点N1包含了一个指向head的shared\_ptr对象，这两个节点就形成了循环引用。因为每个节点都包含了一个指向另一个节点的指针，这两个引用计数中的任何一个都永远不会为0。这两个节点将不会被删除，即使没有任何其他shared\_ptr对象指向它们中的任何一个。为了打破这个循环引用，N1应该包含指向head的weak\_ptr对象，而不是一个shared\_ptr对象。通过weak\_ptr对象从N1回到head（不增加head的引用计数），这样一旦指向head的最后一个shared\_ptr对象被销毁了，head的引用计数就会变为0，于是head将会被删除；它的析构函数也会销毁它所包含的指向N1的shared\_ptr对象，因此N1的引用计数也会变为0，从而N1也会被销毁：
3. **shared\_ptr<node> head(new node(3));**



1. **shared\_ptr<node> N1(new node(4));**



1. **head->next = N1;N1->next = head;**



###### 未介绍和实现的接口列表：

1. 构造函数：

template <typename U, typename D> shared\_ptr(U \*p, D dek);

template <typename U, typename D, typename Alloc> shared\_ptr(U \*p, D dek, Alloc alloc);

template <typename U > shared\_ptr(const shared\_ptr<U> &x);

template <typename U > shared\_ptr(const shared\_ptr<U> &x, element\_type \*p);

1. make\_shared、allocate\_shared
2. static\_pointer\_cast、dynamic\_pointer\_cast、const\_pointer\_cast
3. enable\_shared\_from\_this
4. weak\_ptr

###### 参考书目和网站：

*《C++标准库 第2版》*，英文原版为：*The C++ standard library : a tutorial and reference / Nicolai M. Josuttis.—2nd ed*

*《C++标准库扩展权威指南》*，英文原版为：*The C++ standard library extensions : a tutorial and refence / Pete Bdcker*

*《C++ Templates 中文版》*，英文原版为：*C++ Templates : The Complete Guide / David Vandevoorde, Nicolai M. Josuttis*

<http://www.boost.org/libs/smart_ptr/shared_ptr.htm>

<http://www.cplusplus.com/reference/memory/shared_ptr/>

<http://en.cppreference.com/w/cpp/memory/shared_ptr>