智能指针

描述各种类型智能指针的使用场景和使用方法，以《Boost程序库完全开发指南》为资料。Boost、C++11、Qt中的智能指针大同小异。

# smart\_ptr库概述

程序申请资源（内存、文件描述符、socket、操作系统handle、数据库连接等）后必须及时归还系统，否则就会产生难以预料的后果。

## RAII机制

为了管理内存等资源，C++通常采用RAII机制（资源获取即初始化，Resource Acquisition Is Initialization），在使用资源的类的构造函数中申请资源，然后使用，最后在析构函数中释放资源。

## scoped\_ptr

scoped\_ptr是一个类似auto\_ptr的智能指针，它包装了new操作符在堆上分配的动态对象，能够保证动态创建的对象在任何时候都可以被正确地删除。但scoped\_ptr的所有权更加严格，不得转让，一旦scoped\_ptr获取了对象的管理权，你就无法再从它那里取回来。

scoped\_ptr的名字向使用者传递了一个信息：这个智能指针只能在本作用域里使用，不希望被转让。

### 操作函数

scoped\_ptr的构造函数接受一个类型为T\*的指针p，创建出一个scoped\_ptr对象，并在内部保存指针参数p。p必须是一个new表达式动态分配的结果，或者是个空指针（0）。当scoped\_ptr对象的生命期结束时，析构函数~scoped\_ptr()会使用delete操作符自动销毁所保存的指针对象，从而正确的回收资源。

scoped\_ptr同时把拷贝构造函数和赋值操作符都声明为私有的，禁止对智能指针的复制操作，保证被它管理的指针不能被转让所有权。

reset()的功能是重置scoped\_ptr：它删除原来保存的指针，在保存新的指针值p。如果p是空指针，那么scoped\_ptr将不持有任何指针。一般情况下reset()不应该被调用，因为它违背了scoped\_ptr的本意—资源应该一直由scoped\_ptr自己自动管理。

scoped\_ptr用operator\*()和operator->()重载了解引用操作符\*和箭头操作符->，以模仿被代理的原始指针的行为，因此可以把scoped\_ptr对象如同指针一样使用。如果scoped\_ptr保存空指针，那么这两个操作未定义。

scoped\_ptr不支持比较操作，不能在两个scoped\_ptr之间、scoped\_ptr和原始指针或空指针之间进行相等和不相等测试，也无法为它编写额外的比较函数，因为它已经将operator==和operator!=两个操作符的重载都声明为私有的。但是scoped\_ptr提供了一个可以在bool语境中自动转换成bool值的功能，用来测试scoped\_ptr是否持有一个有效的指针。它可以代替与空指针的比较操作。

swap()可以交换两个scoped\_ptr保存的原始指针。它是高效的操作，被用于实现reset()函数，也可以被boost::swap所利用。

get()返回scoped\_ptr内部保存的原始指针，可以用在某些要求必须是原始指针的场景。但使用时必须小心，这将使原始指针脱离scoped\_ptr的控制！不能对这个指针做delete操作，否则scoped\_ptr析构时会对已经删除的指针再进行删除操作，发生未定义行为。

### 用法

scoped\_ptr只能在scoped\_ptr被声明的作用域内使用。

使用scoped\_ptr会带来两个好处：一是使代码变得清晰简单，而简单意味着更少的错误；二是它没有增加多余的操作，安全的同时保证了效率，可以获得与原始指针同样的速度。

scoped\_ptr不能用作容器的元素。因为scoped\_ptr不支持拷贝和赋值，不符合容器对元素类型的要求。

## shared\_ptr

shared\_ptr和scoped\_ptr一样包装了new操作符在堆上分配的动态对象，但它实现的是引用计数型的智能指针，可以被自由地拷贝和赋值，在任意的地方共享它，当没有代码使用（引用计数为0）它时才删除被包装的动态分配的对象。shared\_ptr也可以安全的放到标准容器中。

### 操作函数

shared\_ptr与scoped\_ptr的相似之处：都重载了\*和->操作符以模仿原始指针的行为，提供隐式bool类型转换以判断指针的有效性，get()可以得到原始指针，并且没有提供指针算术操作。

shared\_ptr有多种形式的构造函数，应用于各种可能的情形：

* 无参的shared\_ptr()创建一个持有空指针的shared\_ptr；
* shared\_ptr(Y \* p)获得指向类型T的指针p的管理权，同时引用计数置为1。这个构造函数要求Y类型必须能够转换为T类型；
* shared\_ptr(shared\_ptr const & r)从另外一个shared\_ptr获得指针的管理权，同时引用计数加1，结果是两个shared\_ptr共享一个指针的管理权；
* shared\_ptr(std::auto\_ptr<Y> & r)从一个auto\_ptr获得指针的管理权，引用计数置为1，同时auto\_ptr自动失去管理权；
* operator==赋值操作符可以从另外一个shared\_ptr或auto\_ptr获得指针的管理权，其行为同构造函数；
* shared\_ptr(Y \* p, D d)行为类似shared\_ptr(Y \* p)，但使用参数d指定了析构时的定制删除器，而不是简单的delete。

reset()是将引用计数减1，停止对指针的共享，除非引用计数为0，否则不会发生删除操作。带参数的reset()则类似相同形式的构造函数，原指针引用计数减1的同时改为管理另一个指针。

unique()在shared\_ptr是指针的唯一所有者时返回true，这时shared\_ptr的行为类似auto\_ptr或scoped\_ptr。

use\_count()返回当前指针的引用计数。use\_count()仅用于测试或者调试，它不提供高效率的操作，而且有时候可能是不可用的。而unique()则是可靠的，任何时候都可用，而且比use\_count() == 1速度更快。

shared\_ptr还支持比较运算，可以测试两个shared\_ptr的相等或不相等，比较基于内部保存的指针。shared\_ptr还可以用operator<比较大小，同样基于内部保存的指针，但不提供除operator<之外的比较操作符，这使得shared\_ptr可以被用于标准关联容器。

shared\_ptr不能使用诸如static\_cast<T\*>(p.get())的形式，这将导致转型后的指针无法再被shared\_ptr正确管理。为了支持这样的用法，shared\_ptr提供了类似的转型函数static\_pointer\_cast<T>()、const\_pointer\_cast<T>()、dynamic\_pointer\_cast<T>()，它们与标准的转换操作符类似，但返回的是转型后的shared\_ptr。

shared\_ptr还支持流输出操作符operator<<，输出内部的指针值，方便调试。

### 工厂函数

shared\_ptr在头文件<boost/make\_shared.hpp>中提供了一个自由工厂函数make\_shared<T>()，来消除显示的new调用，声明如下：

template<class T, class… Args>

shared\_ptr<T> make\_shared(Args && … args);

make\_shared()函数可以接受最多10个参数，然后把它们传递给类型T的构造函数，创建一个shared\_ptr<T>的对象并返回。make\_shared()函数要比直接创建shared\_ptr对象的方式快且高效，因为它内部仅分配一次内存，消除了shared\_ptr构造时的开销。

除了make\_shared()，smart\_ptr库还提供了一个allocate\_shared()，它比make\_shared()多接受一个定制的内存分配器类型参数，其他方面都相同。

### 定制删除器

shared\_ptr(Y \* p, D d)的第一个参数是要被管理的指针，它的含义与其他构造函数的参数相同。第二删除器参数d则告诉shared\_ptr在析构时不是使用delete来操作指针p，而要用d来操作，即把delete p换成d(p)。

删除器d可以是一个函数对象，也可以是一个函数指针，只要它能够像函数那样被调用，使得d(p)成立即可。对删除器的要求是它必须是可拷贝的，行为必须像delete那样，不能抛出异常。

为了配合删除器的工作，shared\_ptr提供一个自由函数get\_deleter(shared\_ptr<T> const & p)，它能够返回删除器的指针。

基于shared\_ptr<void>和定制删除器，shared\_ptr<void>可以实现退出作用域时调用任意函数。例如：

void any\_func(void\* p)

{

cout << “some operate” << endl;

}

int main()

{

shared\_ptr<void> p((void\*)0, any\_func);

}

shared\_ptr<void>存储了一个空指针，并指定了删除器是操作void\*的一个函数，因此当它析构时会自动调用函数any\_func()，从而执行任意我们想做的工作。

### shared\_ptr<void>

shared\_ptr<void>能够存储void\*型的指针，而void\*型指针可以指向任何类型，因此shared\_ptr<void>就像是一个泛型的指针容器，拥有容纳任意类型的能力。

但将指针存储为void\*同时也丧失了原来的类型信息，为了在需要的时候正确使用，可以用static\_pointer\_cast<T>等转型函数重新转为原来的指针。但这涉及到运行时动态类型转换，它会使代码不够安全，建议最好不要这样使用。

## weak\_ptr

weak\_ptr是为配合shared\_ptr而引入的一种智能指针，它更像是shared\_ptr的一个助手而不是智能指针，因为它不具有普通指针的行为，没有重载operator\*和->。它的最大作用在于协助shared\_ptr工作，像旁观者那样观测资源的使用情况。

### 用法

weak\_ptr被设计为与shared\_ptr共同工作，可以从一个shared\_ptr或者另一个weak\_ptr对象构造，获得资源的观测权。但shared\_ptr没有共享资源，它的构造不会引起指针引用计数的增加。同样，在weak\_ptr析构时也不会导致引用计数减少，它只是一个静静的观测者。

use\_count()可以观测资源的引用计数。

expired()的功能等价于use\_count() == 0，但更快，表示被观测的资源已经不复存在。

weak\_ptr没有重载operator\*和->，因为它不共享指针，不能操作资源，这正是它“弱”的原因。但它可以用一个成员函数lock()从被观测的shared\_ptr获得一个可用的shared\_ptr对象，从而操作资源。但当expired() == true的时候，lock()将返回一个存储空指针的shared\_ptr。

### 获得this的shared\_ptr

weak\_ptr的一个重要用途是获得this指针的shared\_ptr，使对象自己能够生产shared\_ptr管理自己：对象使用weak\_ptr观测this指针，这并影响引用计数，在需要的时候调用lock()函数，返回一个符合要求的shared\_ptr供外界使用。

这个解决方案被实现为一个惯用法，在头文件<boost/enable\_shared\_from\_this.hpp>定义了一个助手类enable\_shared\_from\_this<T>，它的生命摘要如下：

template<class T>

class enable\_shared\_from\_this

{

public:

shared\_ptr<T> shared\_from\_this();

}

使用的时候只要让想被shared\_ptr管理的类从它继承即可，成员函数shared\_from\_this()会返回this的shared\_ptr。

需要注意的是千万不能从一个普通对象（非shared\_ptr）使用shared\_from\_this()获取shared\_ptr。因为这样虽然语法上正确，编译也无问题，但在运行时会导致shared\_ptr析构时企图删除一个栈上分配的对象，发生未定义行为。