Item 19:对于共享资源使用std::shared_ptr

条款十九:对于共享资源使用std::shared_ptr

程序员使用带垃圾回收的语言指着C++笑看他们如何防止资源泄露。“真是原始啊！”他们嘲笑着说。“你们没有从1960年的Lisp那里得到启发吗，机器应该自己管理资源的生命周期而不应该依赖人类。”C++程序员翻白眼。“你得到的启发就是只有内存算资源，其他资源释放都是非确定性的你知道吗？我们更喜欢通用，可预料的销毁，谢谢你。”但我们的虚张声势可能底气不足。因为垃圾回收真的很方便，而且手动管理生命周期真的就像是使用石头小刀和兽皮制作RAM电路。为什么我们不能同时有两个完美的世界：一个自动工作的世界（垃圾回收），一个销毁可预测的世界（析构）？

C++11中的`std::shared_ptr`将两者组合了起来。一个通过`std::shared_ptr`访问的对象其生命周期由指向它的指针们共享所有权（shared ownership）。没有特定的`std::shared_ptr`拥有该对象。相反，所有指向它的`std::shared_ptr`都能相互合作确保在它不再使用的那个点进行析构。当最后一个`std::shared_ptr`到达那个点，`std::shared_ptr`会销毁它所指向的对象。就垃圾回收来说，客户端不需要关心指向对象的生命周期，而对象的析构是确定性的。

`std::shared_ptr`通过引用计数来确保它是否是最后一个指向某种资源的指针，引用计数关联资源并跟踪有多少`std::shared_ptr`指向该资源。`std::shared_ptr`构造函数递增引用计数值（注意是通常——原因参见下面），析构函数递减值，拷贝赋值运算符可能递增也可能递减值。（如果sp1和sp2是`std::shared_ptr`并且指向不同对象，赋值运算符`sp1=sp2`会使sp1指向sp2指向的对象。直接效果就是sp1引用计数减一，sp2引用计数加一。）如果`std::shared_ptr`发现引用计数值为零，没有其他`std::shared_ptr`指向该资源，它就会销毁资源。

引用计数暗示着性能问题：

- **`std::shared_ptr`大小是原始指针的两倍**，因为它内部包含一个指向资源的原始指针，还包含一个资源的引用计数值。
- **引用计数必须动态分配**。 理论上，引用计数与所指对象关联起来，但是被指向的对象不知道这件事情（译注：不知道有指向自己的指针）。因此它们没有办法存放一个引用计数值。Item21会解释使用`std::make_shared`创建`std::shared_ptr`可以避免引用计数的动态分配，但是还存在一些`std::make_shared`不能使用的场景，这时候引用计数就会动态分配。
- **递增递减引用计数必须是原子性的**，因为多个reader、writer可能在不同的线程。比如，指向某种资源的`std::shared_ptr`可能在一个线程执行析构，在另一个不同的线程，`std::shared_ptr`指向相同的对象，但是执行的确是拷贝操作。原子操作通常比非原子操作要慢，所以即使是引用计数，你也应该假定读写它们是存在开销的。

我写道`std::shared_ptr`构造函数只是“通常”递增指向对象的引用计数会不会让你有点好奇？创建一个指向对象的`std::shared_ptr`至少产生了一个指向对象的智能指针，为什么我没说**总是**增加引用计数值？

原因是移动构造函数的存在。从另一个`std::shared_ptr`移动构造新`std::shared_ptr`会将原来的`std::shared_ptr`设置为null，那意味着老的`std::shared_ptr`不再指向资源，同时新的`std::shared_ptr`指向资源。这样的结果就是不需要修改引用计数值。因此移动`std::shared_ptr`会比拷贝它要快：拷贝要求递增引用计数值，移动不需要。移动赋值运算符同理，所以移动赋值运算符也比拷贝赋值运算符快。

类似`std::unique_ptr`（参加Item18），`std::shared_ptr`使用**delete**作为资源的默认销毁器，但是它也支持自定义的销毁器。这种支持有别于`std::unique_ptr`。对于`std::unique_ptr`来说，销毁器类型是智能指针类型的一部分。对于`std::shared_ptr`则不是：

```
auto loggingDel = [](Widget *pw)     //自定义销毁器
                  {                  // (和Item 18一样)
                      makeLogEntry(pw);
                      delete pw;
                  };

std::unique_ptr<                     // 销毁器类型是
    Widget, decltype(loggingDel)     // ptr类型的一部分
    > upw(new Widget, loggingDel);
std::shared_ptr<Widget>              // 销毁器类型不是
spw(new Widget, loggingDel);         // ptr类型的一部分
```

`std::shared_ptr`的设计更为灵活。考虑有两个`std::shared_ptr`，每个自带不同的销毁器（比如通过lambda表达式自定义销毁器）：

```
auto customDeleter1 = [](Widget *pw) { … };
auto customDeleter2 = [](Widget *pw) { … };
std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

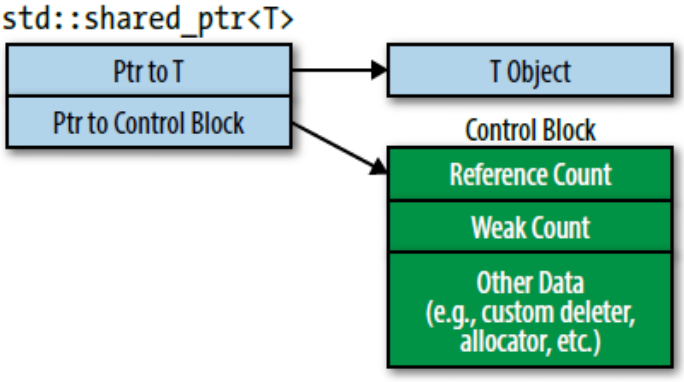
因为**pw1**和**pw2**有相同的类型，所以它们都可以放到存放那个类型的对象的容器中：

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

它们也能相互赋值，也可以传入形参为`std::shared_ptr<Widget>`的函数。但是`std::unique_ptr`就不行，因为`std::unique_ptr`把销毁器视作类型的一部分。

另一个不同于`std::unique_ptr`的地方是，指定自定义销毁器不会改变`std::shared_ptr`对象的大小。不管销毁器是什么，一个`std::shared_ptr`对象都是两个指针大小。这是个好消息，但是它应该让你隐隐约约不安。自定义销毁器可以是函数对象，函数对象可以包含任意多的数据。它意味着函数对象是任意大的。`std::shared_ptr`怎么能引用一个任意大的销毁器而不使用更多的内存？

它不能。它必须使用更多的内存。然而，那部分内存不是`std::shared_ptr`对象的一部分。那部分在堆上面，只要`std::shared_ptr`自定义了分配器，那部分内存随便在哪都行。我前面提到了`std::shared_ptr`对象包含了所指对象的引用计数。没错，但是有点误导人。因为引用计数是另一个更大的数据结构的一部分，那个数据结构通常叫做**控制块**（control block）。控制块包含除了引用计数值外的一个自定义销毁器的拷贝，当然前提是存在自定义销毁器。如果用户还指定了自定义分配器，控制器也会包含一个分配器的拷贝。控制块可能还包含一些额外的数据，正如Item21提到的，一个次级引用计数weak count，但是目前我们先忽略它。我们可以想象`std::shared_ptr`对象在内存中是这样：



当`std::shared_ptr`对象一创建，对象控制块就建立了。至少我们期望是如此。通常，对于一个创建指向对象的`std::shared_ptr`的函数来说不可能知道是否有其他`std::shared_ptr`早已指向那个对象，所以控制块的创建会遵循下面几条规则：

- **`std::make_shared`总是创建一个控制块**(参见Item21)。它创建一个指向新对象的指针，所以可以肯定`std::make_shared`调用时对象不存在其他控制块。
- **当从独占指针上构造出`std::shared_ptr`时会创建控制块（即`std::unique_ptr`或者`std::auto_ptr`）**。独占指针没有使用控制块，所以指针指向的对象没有关联其他控制块。（作为构造的一部分，`std::shared_ptr`侵占独占指针所指向的对象的独占权，所以`std::unique_ptr`被设置为null）
- **当从原始指针上构造出`std::shared_ptr`时会创建控制块**。如果你想从一个早已存在控制块的对象上创建`std::shared_ptr`，你将假定传递一个`std::shared_ptr`或者`std::weak_ptr`作为构造函数实参，而不是原始指针。用`std::shared_ptr`或者`std::weak_ptr`作为构造函数实参创建`std::shared_ptr`不会创建新控制块，因为它可以依赖传递来的智能指针指向控制块。

这些规则造成的后果就是从原始指针上构造超过一个`std::shared_ptr`就会让你走上未定义行为的快车道，因为指向的对象有多个控制块关联。多个控制块意味着多个引用计数值，多个引用计数值意味着对象将会被销毁多次（每个引用计数一次）。那意味着下面的代码是有问题的，很有问题，问题很大：

```
auto pw = new Widget; // pw是原始指针
…
std::shared_ptr<Widget> spw1(pw, loggingDel); // 为*pw创建控制块
…
std::shared_ptr<Widget> spw2(pw, loggingDel); // 为*pw创建第二个控制块
```

创建原始指针指向动态分配的对象很糟糕，因为它完全背离了这章的建议：对于共享资源使用`std::shared_ptr`而不是原始指针。（如果你忘记了该建议的动机，请翻到115页）。撇开那个不说，创建**pw**那一行代码虽然让人厌恶，但是至少不会造成未定义程序行为。

现在，传给**spw1**的构造函数一个原始指针，它会为指向的对象创建一个控制块（引用计数值在里面）。这种情况下，指向的对象是`*pw`。就其本身而言没什么问题，但是将同样的原始指针传递给**spw2**的构造函数会再次为`*pw`创建一个控制块。因此`*pw`有两个引用计数值，每一个最后都会变成零，然后最终导致`*pw`销毁两次。第二个销毁会产生未定义行为。

`std::shared_ptr`给我们上了两堂课。第一，避免传给`std::shared_ptr`构造函数原始指针。通常替代方案是使用`std::make_shared`(参见Item21)，不过上面例子中，我们使用了自定义销毁器，用`std::make_shared`就没办法做到。第二，如果你必须传给`std::shared_ptr`构造函数原始指针，直接传new出来的结果，不要传指针变量。如果上面代码第一部分这样重写：

```
std::shared_ptr<Widget> spw1(new Widget, // 直接使用new的结果
                             loggingDel);
```

会少了很多创建第二个从原始指针上构造`std::shared_ptr`的诱惑。相应的，创建spw2也会很自然的用spw1作为初始化参数（即用`std::shared_ptr`拷贝构造），那就没什么问题了：

```
std::shared_ptr<Widget> spw2(spw1);  // spw2使用spw1一样的控制块
```

一个尤其令人意外的地方是使用**this**原始指针作为`std::shared_ptr`构造函数实参的时候可能导致创建多个控制块。假设我们的程序使用`std::shared_ptr`管理**Widget**对象，我们有一个数据结构用于跟踪已经处理过的**Widget**对象：

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

继续，假设**Widget**有一个用于处理的成员函数：

```
class Widget {
public:
    …
    void process();
    …
};
```

对于**Widget::process**看起来合理的代码如下：

```
void Widget::process()
{
    …                                           // 处理Widget
    processedWidgets.emplace_back(this); // 然后将他加到已处理过的Widget的列表中
                                                // 这是错的
}
```

评论已经说了这是错的——或者至少大部分是错的。（错误的部分是传递this，而不是使用了**emplace_back**。如果你不熟悉**emplace_back**，参见Item42）。上面的代码可以通过编译，但是向容器传递一个原始指针（this），`std::shared_ptr`会由此为指向的对象（`*this`）创建一个控制块。那看起来没什么问题，直到你意识到如果成员函数外面早已存在指向**Widget**对象的指针，它是未定义行为的Game, Set, and Match（译注：一部电影，但是译者没看过。。。）。

`std::shared_ptr`API已有处理这种情况的设施。它的名字可能是C++标准库中最奇怪的一个：`std::enable_shared_from_this`。它是一个用做基类的模板类，模板类型参数是某个想被`std::shared_ptr`管理且能从该类型的**this**对象上安全创建`std::shared_ptr`指针的存在。在我们的例子中，**Widget**将会继承自`std::enable_shared_from_this`：

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
    …
    void process();
    …
};
```

正如我所说，`std::enable_shared_from_this`是一个用作基类的模板类。它的模板参数总是某个继承自它的类，所以**Widget**继承自`std::enable_shared_from_this<Widget>`。如果某类型继承自一个由该类型（译注：作为模板类型参数）进行模板化得到的基类这个东西让你心脏有点遭不住，别去想它就好了。代码完全合法，而且它背后的设计模式也是没问题的，并且这种设计模式还有个标准名字，尽管该名字和`std::enable_shared_from_this`一样怪异。这个标准名字就是奇异递归模板模式（The Curiously Recurring Template Pattern(CRTP)）。如果你想学更多关于它的内容，请搜索引擎一展身手，现在我们要回到`std::enable_shared_from_this`上。

`std::enable_shared_from_this`定义了一个成员函数，成员函数会创建指向当前对象的`std::shared_ptr`却不创建多余控制块。这个成员函数就是`shared_from_this`，无论在哪当你想使用`std::shared_ptr`指向this所指对象时都请使用它。这里有个`Widget::process`的安全实现：

```
void Widget::process()
{
    // 和之前一样，处理Widget
    …
    // 把指向当前对象的shared_ptr加入processedWidgets
    processedWidgets.emplace_back(shared_from_this());
}
```

从内部来说，`shared_from_this`查找当前对象控制块，然后创建一个新的`std::shared_ptr`指向这个控制块。设计的依据是当前对象已经存在一个关联的控制块。要想符合设计依据的情况，必须已经存在一个指向当前对象的`std::shared_ptr`(即调用shared_from_this的成员函数外面已经存在一个`std::shared_ptr`)。如果没有`std::shared_ptr`指向当前对象（即当前对象没有关联控制块），行为是未定义的，shared_from_this通常抛出一个异常。

要想防止客户端在调用`std::shared_ptr`前先调用`shared_from_this`，继承自`std::enable_shared_from_this`的类通常将它们的构造函数声明为private，并且让客户端通过工厂方法创建`std::shared_ptr`。以**Widget**为例，代码可以是这样：

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
    // 完美转发参数的工厂方法
    template<typename... Ts>
    static std::shared_ptr<Widget> create(Ts&&... params);
    …
    void process(); // 和前面一样
    …
private:
    …
};
```

现在，你可能隐约记得我们讨论控制块的动机是想了解`std::shared_ptr`关联一个控制块的成本。既然我们已经知道了怎么避免创建过多控制块，就让我们回到原来的主题。

控制块通常只占几个word大小，自定义销毁器和分配器可能会让它变大一点。通常控制块的实现比你想的更复杂一些。它使用继承，甚至里面还有一个虚函数（用来确保指向的对象被正确销毁）。这意味着使用`std::shared_ptr`还会招致控制块使用虚函数带来的成本。

了解了动态分配控制块，任意大小的销毁器和分配器，虚函数机制，原子引用计数修改，你对于`std::shared_ptr`的热情可能有点消退。可以理解，对每个资源管理问题来说都没有最佳的解决方案。但就它提供的功能来说，`std::shared_ptr`的开销是非常合理的。在通常情况下，`std::shared_ptr`创建控制块会使用默认销毁器和默认分配器，控制块只需三个word大小。它的分配基本上是无开销的。（开销被并入了指向的对象的分配成本里。细节参见Item21）。对`std::shared_ptr`解引用的开销不会比原始指针高。执行原子引用计数修改操作需要承担一两个原子操作开销，这些操作通常都会一一映射到机器指令上，所以即使对比非原子指令来说，原子指令开销较大，但是它们仍然只是单个指令。对于每个被`std::shared_ptr`指向的对象来说，控制块中的虚函数机制产生的开销通常只需要承受一次，即对象销毁的时候。

作为这些轻微开销的交换，你得到了动态分配的资源的生命周期自动管理的好处。大多数时候，比起手动管理，使用`std::shared_ptr`管理共享性资源都是非常合适的。如果你还在犹豫是否能承受`std::shared_ptr`带来的开销，那就再想想你是否需要共享资源。如果独占资源可行或者可能可行，用`std::unique_ptr`是一个更好的选择。它的性能profile更接近于原始指针，并且从`std::unique_ptr`升级到`std::shared_ptr`也很容易，因为`std::shared_ptr`可以从`std::unique_ptr`上创建。

反之不行。当你的资源由`std::shared_ptr`管理，现在又想修改资源生命周期管理方式是没有办法的。即使引用计数为一，你也不能重新修改资源所有权，改用`std::unique_ptr`管理它。所有权和`std::shared_ptr`指向的资源之前签订的协议是"除非死亡否则永不分离"。不能离婚，不能废除，没有特许。

`std::shared_ptr`不能处理的另一个东西是数组。和`std::unique_ptr`不同的是，`std::shared_ptr`的API设计之初就是针对单个对象的，没有办法`std::shared_ptr<T[]>`。一次又一次，"聪明"的程序员踌躇于是否该使用`std::shared_ptr<T>`指向数组，然后传入自定义数组销毁器。（即`delete []`）。这可以通过编译，但是是一个糟糕的注意。一方面，`std::shared_ptr`没有提供`operator[]`重载，所以数组索引操作需要借助怪异的指针算术。另一方面，`std::shared_ptr`支持转换为指向基类的指针，这对于单个对象来说有效，但是当用于数组类型时相当于在类型系统上开洞。（出于这个原因，`std::unique_ptr`禁止这种转换。）。更重要的是，C++11已经提供了很多内置数组的候选方案（比如`std::array`,`std::vector`,`std::string`）。声明一个指向傻瓜数组的智能指针几乎总是标识着糟糕的设计。

记住:

- `std::shared_ptr`为任意共享所有权的资源一种自动垃圾回收的便捷方式。
- 较之于`std::unique_ptr`，`std::shared_ptr`对象通常大两倍，控制块会产生开销，需要原子引用计数修改操作。
- 默认资源销毁是通过**delete**，但是也支持自定义销毁器。销毁器的类型是什么对于`std::shared_ptr`的类型没有影响。
- 避免从原始指针变量上创建`std::shared_ptr`。