7、std::shared_ptr

Chapter	M
Name	xingzp

std::unique_ptr被设计为单独拥有和管理一个资源，与之不同的是，std::shared_ptr用于解决需要多个智能指针共同拥有一个资源的情况。

这意味着可以让多个std::shared_ptr指向相同的资源。在内部，std::shared_ptr跟踪有多少std::shared_ptr正在共享资源。只要至少有一个std::shared_ptr指向资源，资源将不会被释放。一旦管理资源的最后一个std::shared_ptr超出作用域(或被重新分配到指向其他东西)，资源将被释放。

像std::unique_ptr一样, std::shared_ptr存在于<memory>头文件中。

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  // allocate a Resource object and have it owned by std::shared_ptr
  Resource* res { new Resource };
  std::shared_ptr<Resource> ptr1{ res };
```

```
  {
    std::shared_ptr<Resource> ptr2 { ptr1 }; // make another std::shared_ptr pointing
 to the same thing

    std::cout << "Killing one shared pointer\n";
  } // ptr2 goes out of scope here, but nothing happens

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
```

上述程序打印：

```
Resource acquired
Killing one shared pointer
Killing another shared pointer
Resource destroyed
```

在上面的代码中，我们创建了一个动态Resource对象，并设置了一个名为ptr1的std::shared_ptr来管理它。在嵌套块内部，我们使用复制构造函数创建第二个std::shared_ptr (ptr2)，它指向同一个Resource。当ptr2超出作用域时，资源不会被释放，因为ptr1仍然指向资源。当ptr1超出作用域时，ptr1注意到不再有std::shared_ptr管理资源，因此它释放资源。

注意，**我们从第一个共享指针创建了第二个共享指针。这是很重要的**。考虑下面的类似程序:

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Resource* res { new Resource };
  std::shared_ptr<Resource> ptr1 { res };
  {
    std::shared_ptr<Resource> ptr2 { res }; // create ptr2 directly from res (instead
 of ptr1)

    std::cout << "Killing one shared pointer\n";
  } // ptr2 goes out of scope here, and the allocated Resource is destroyed

  std::cout << "Killing another shared pointer\n";
```

```
  return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed again
```

上述程序打印：

```
Resource acquired
Killing one shared pointer
Resource destroyed
Killing another shared pointer
Resource destroyed
```

然后崩溃(至少在作者的机器上)。

不同之处在于，我们创建了两个std::shared_ptr，彼此独立。因此，即使它们都指向相同的资源，它们也不知道彼此。当ptr2超出作用域时，它认为自己是资源的唯一所有者，并将其释放。当ptr1稍后超出作用域时，它会想同样的事情，并尝试再次删除资源。然后不好的事情发生了。

幸运的是，这很容易避免:如果您需要多个std::shared_ptr到一个给定的资源，复制一个现有的std::shared_ptr。

🙂 **Best practice**
如果需要多个std::shared_ptr指向相同的资源，请始终复制现有的std::shared_ptr。

---

std::make_shared

就像std::make_unique()可以在c++ 14中用来创建std::unique_ptr一样，std::make_shared()可以(而且应该)用来创建std::shared_ptr。std::make_shared()在c++ 11中可用。

这是我们最初的例子，使用std::make_shared():

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
```

```
};

int main()
{
  // allocate a Resource object and have it owned by std::shared_ptr
  auto ptr1 { std::make_shared<Resource>() };
  {
    auto ptr2 { ptr1 }; // create ptr2 using copy of ptr1

    std::cout << "Killing one shared pointer\n";
  } // ptr2 goes out of scope here, but nothing happens

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
```

使用std::make_shared()的原因与std::make_unique()相同——std::make_shared()更简单、更安全(无法使用此方法直接创建指向相同资源的两个std::shared_ptr)、性能也会更好。

挖掘一下std::shared_ptr

与std::unique_ptr内部使用一个指针不同，std::shared_ptr内部使用两个指针。一个指针指向被管理的资源。另一个指向一个“控制块”，这是一个动态分配的对象，它跟踪一堆东西，包括有多少std::shared_ptr指向资源。当std::shared_ptr通过std::shared_ptr构造函数创建时，管理对象(通常被传入)的内存和控制块(由构造函数创建)的内存是分别分配的。但是，当使用std::make_shared()时，可以将其优化为单个内存分配，从而获得更好的性能。

这也解释了为什么独立创建两个指向相同资源的std::shared_ptr会给我们带来麻烦。每个std::shared_ptr将有一个指向资源的指针。但是，每个std::shared_ptr将独立分配它自己的控制块，这将表明它是拥有该资源的唯一指针。因此，当std::shared_ptr超出作用域时，它将释放该资源，而没有意识到还有其他std::shared_ptr也试图管理该资源。

但是，当使用拷贝赋值克隆std::shared_ptr时，可以适当地更新控制块中的数据，以指示现在有额外的std::shared_ptr共同管理资源。

---

std::shared_ptr可以从std::unique_ptr创建

std::unique_ptr可以通过一个特殊的std::shared_ptr构造函数（接受一个std::unique_ptr右值）转换为std::shared_ptr。std::unique_ptr的内容将被移动到std::shared_ptr。

但是，std::shared_ptr不能安全地转换为std::unique_ptr。这意味着，如果要创建一个返回智能指针的函数，最好返回std::unique_ptr，并在适当的时候可以将其赋值给std::shared_ptr。

---

std::shared_ptr的风险

std::shared_ptr有一些与std::unique_ptr相同的挑战——如果std::shared_ptr没有被正确地释放(要么因为它是动态分配的，从来没有删除，要么它是动态分配的对象的一部分，且从来没有删除的)，那么它所管理的资源也不会被释放。使用std::unique_ptr，您只需担心一个智能指针是否被正确处理。对于std::shared_ptr，您必须考虑所有这些问题。如果管理资源的std::shared_ptr中的任何一个没有被正确地销毁，该资源将不会被正确地释放。

---

std::shared_ptr和数组

在c++ 17和更早的版本中，std::shared_ptr对管理数组没有适当的支持，不应该用来管理C风格的数组。从c++ 20开始，std::shared_ptr确实支持数组。

结论

std::shared_ptr是为需要多个智能指针共同管理同一资源而设计的。当管理该资源的最后一个std::shared_ptr被销毁时，该资源将被释放。