3、移动构造函数和移动赋值

Chapter	M
Name	xingzp

在本章第一课——智能指针和移动语义的介绍中，我们研究了std::auto_ptr，讨论了移动语义的需求，并研究了在为实现移动语义而修改复制语义的函数(复制构造函数和复制赋值操作符)时出现的一些缺点。

在这一课中，我们将深入了解c++ 11如何通过移动构造函数和移动赋值来解决这些问题。

复制（拷贝）构造函数和复制（拷贝）赋值

首先，让我们花点时间回顾一下复制语义。

复制构造函数用于通过复制同一类的对象来初始化类。复制赋值用于将一个类对象复制到另一个现有类对象。默认情况下，如果没有显式提供复制构造函数和复制赋值操作符，c++将提供它们。这些编译器提供的函数执行浅拷贝，这可能会给分配动态内存的类带来问题。因此，处理动态内存的类应该重写这些函数来进行深度复制。

回到本章第一课中的Auto_ptr智能指针类示例，让我们看看实现了执行深度复制的复制构造函数和复制赋值操作符的版本，以及练习它们的示例程序:

```
#include <iostream>

template<typename T>
```

```
class Auto_ptr3
{
  T* m_ptr;
public:
  Auto_ptr3(T* ptr = nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr3()
  {
    delete m_ptr;
  }

  // Copy constructor
  // Do deep copy of a.m_ptr to m_ptr
  Auto_ptr3(const Auto_ptr3& a)
  {
    m_ptr = new T;
    *m_ptr = *a.m_ptr;
  }

  // Copy assignment
  // Do deep copy of a.m_ptr to m_ptr
  Auto_ptr3& operator=(const Auto_ptr3& a)
  {
    // Self-assignment detection
    if (&a == this)
      return *this;

    // Release any resource we're holding
    delete m_ptr;

    // Copy the resource
    m_ptr = new T;
    *m_ptr = *a.m_ptr;

    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};

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

Auto_ptr3<Resource> generateResource()
{
  Auto_ptr3<Resource> res{new Resource};
  return res; // this return value will invoke the copy constructor
}
```

```
int main()
{
  Auto_ptr3<Resource> mainres;
  mainres = generateResource(); // this assignment will invoke the copy assignment

  return 0;
}
```

在这个程序中，我们使用了一个名为generateResource()的函数来创建一个智能指针封装的资源，然后将其传递回main()函数。函数main()然后将其赋值给一个现有的Auto_ptr3对象。

当这个程序运行时，它输出:

```
Resource acquired
Resource acquired
Resource destroyed
Resource acquired
Resource destroyed
Resource destroyed
```

(注意:如果你的编译器省略了generateResource()函数的返回值，你可能只会得到4个输出)

对于这样一个简单的程序，创建和销毁的资源太多了!这是怎么回事?

让我们仔细看看。在这个程序中有6个关键步骤(每个打印信息一个步骤):

1. 在generateResource()内部，使用动态分配的Resource创建和初始化局部变量res，这将导致第一个“获取的资源”。
2. res按值返回给main()。我们在这里通过值返回，因为res是一个局部变量——它不能通过地址或引用返回，因为res将在generateResource()结束时被销毁。所以res被复制构造到一个临时对象中。由于复制构造函数执行深度复制，所以在这里分配了一个新的Resource，这将导致第二个“获取资源”。
3. res超出作用域，破坏最初创建的资源，这导致第一个“资源被破坏”。
4. 通过拷贝赋值将临时对象赋值给主对象。因为我们的拷贝赋值也进行深度复制，所以会分配一个新的资源，从而导致另一个“获取资源”。
5. 赋值表达式结束，临时对象超出表达式范围并被销毁，导致“资源销毁”。
6. 在main()的末尾，mainres超出了作用域，显示最后的“Resource destroyed”。

因此，简而言之，因为我们调用一次复制构造函数将构造res复制到一个临时对象，并调用一次复制赋值将临时对象复制到主对象，所以我们最终总共分配和销毁了3个单独的对象。

效率很低，但至少不会崩溃!

但是，使用move语义，我们可以做得更好。

移动构造函数和移动赋值

c++ 11为move语义定义了两个新函数:一个是move构造函数，一个是move赋值操作符。复制构造函数和复制赋值的目标是将一个对象复制到另一个对象，而移动构造函数和移动赋值的目标是将资源的所有权从一个对象转移到另一个对象(这通常比复制成本低得多)。

定义移动构造函数和移动赋值与复制函数的工作方式类似。然而，**这些函数的复制类型采用const的l值引用形参，而移动类型使用非const的r值引用形参**。

这里是与上面相同的Auto_ptr3类，添加了移动构造函数和移动赋值操作符。为了进行比较，我们保留了深度复制复制构造函数和复制赋值操作符。

```
#include <iostream>

template<typename T>
class Auto_ptr4
{
  T* m_ptr;
public:
  Auto_ptr4(T* ptr = nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr4()
  {
    delete m_ptr;
  }

  // Copy constructor
  // Do deep copy of a.m_ptr to m_ptr
  Auto_ptr4(const Auto_ptr4& a)
  {
    m_ptr = new T;
    *m_ptr = *a.m_ptr;
  }

  // Move constructor
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr4(Auto_ptr4&& a) noexcept
    : m_ptr(a.m_ptr)
  {
    a.m_ptr = nullptr; // we'll talk more about this line below
  }
```

```
  // Copy assignment
  // Do deep copy of a.m_ptr to m_ptr
  Auto_ptr4& operator=(const Auto_ptr4& a)
  {
    // Self-assignment detection
    if (&a == this)
      return *this;

    // Release any resource we're holding
    delete m_ptr;

    // Copy the resource
    m_ptr = new T;
    *m_ptr = *a.m_ptr;

    return *this;
  }

  // Move assignment
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr4& operator=(Auto_ptr4&& a) noexcept
  {
    // Self-assignment detection
    if (&a == this)
      return *this;

    // Release any resource we're holding
    delete m_ptr;

    // Transfer ownership of a.m_ptr to m_ptr
    m_ptr = a.m_ptr;
    a.m_ptr = nullptr; // we'll talk more about this line below

    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};

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

Auto_ptr4<Resource> generateResource()
{
  Auto_ptr4<Resource> res{new Resource};
  return res; // this return value will invoke the move constructor
}

int main()
{
  Auto_ptr4<Resource> mainres;
  mainres = generateResource(); // this assignment will invoke the move assignment
```

```
  return 0;
}
```

移动构造函数和移动赋值操作符很简单。我们不是将源对象(a)深入复制到隐式对象中，而是简单地移动(窃取)源对象的资源。这涉及到将源指针浅层复制到隐式对象中，然后将源指针设置为空。

当运行时，这个程序输出:

```
Resource acquired
Resource destroyed
```

这是更好的!

程序的流程和以前完全一样。但是，这个程序调用的不是复制构造函数和复制赋值操作符，而是移动构造函数和移动赋值操作符。再深入一点看:

1. 在generateResource()内部，使用动态分配的Resource创建和初始化局部变量res，这将导致第一个“获取的资源”。
2. res按值返回给main()。res被移动构造到一个临时对象中，将动态创建的存储在res中的对象（资源）转移到临时对象。我们将在下面讨论为什么会发生这种情况(🙂**需要结合C++编译器的RVO机制，不然很可能理解起来有点儿困惑**)。
3. res超出作用域。因为res不再管理指针(它被移动到临时对象中)，所以这里没有发生什么有趣的事情。
4. 临时对象被移动赋值给mainres。这将把存储在临时对象中的动态创建的对象（资源）转移到mainres中。
5. 赋值表达式结束，临时对象超出表达式范围并被销毁。但是，因为临时对象不再管理指针(它被移到了主进程中)，所以这里也没有发生什么有趣的事情。
6. 在main()的末尾，mainres超出了作用域，显示最后的“Resource destroyed”。

因此，我们不是复制Resource两次(一次用于复制构造函数，一次用于复制赋值)，而是将其传输两次。这是更有效的，因为资源只构建和摧毁一次，而不是三次。

什么时候调用移动构造函数和移动赋值?

移动构造函数和移动赋值函数在定义了这些函数后调用，用于构造或赋值的参数是一个r值（🙂**涉及到函数返回值时请结合C++编译器的RVO机制理解**）。最典型的是，

这个r值将是一个字面值或临时值。

在大多数情况下，默认情况下不会提供移动构造函数和移动赋值操作符，除非类没有任何已定义的复制构造函数、复制赋值、移动赋值或析构函数。

移动语义背后的关键

现在您已经有了足够的内容支撑来理解移动语义背后的关键。

如果我们构造一个对象或进行赋值，其中实参是一个l值，我们唯一能做的合理的事情就是复制这个l值。我们不能假设改变l值是安全的，因为它可能在稍后的程序中再次被使用。如果我们有一个表达式" a = b "，我们就不会合理地期望b以任何方式改变。

然而，如果我们构造一个对象或做一个赋值，其中参数是r-value，那么我们知道r-value只是某种类型的临时对象。我们不需要复制它(这可能很昂贵)，而是可以简单地将它的资源(这很便宜)转移到我们正在构造或分配的对象。这样做是安全的，因为临时变量无论如何都会在表达式结束时被销毁，所以我们知道它永远不会再被使用!

c++ 11通过r值引用，使我们能够在实参是r值和l值时提供不同的行为，使我们能够对对象的行为做出更聪明、更有效的决定。

移动构造和移动赋值函数应该始终将两个对象保持在定义良好的状态

在上面的例子中，移动构造函数和移动赋值函数都将a.m_ptr设置为nullptr。这可能看起来无关紧要——毕竟，如果“a”是一个临时的r值，那么如果参数“a”无论如何都将被销毁，为什么还要费心做“清理”呢?

答案很简单:当" a "超出作用域时，将调用a的析构函数，并删除a.m_ptr。如果此时，a.m_ptr仍然指向与m_ptr相同的对象，那么m_ptr将被保留为悬浮指针。当包含m_ptr的对象最终被使用(或销毁)时，我们将得到未定义的行为。

此外，在下一课中，我们将看到“a”可以是l值的情况。在这种情况下，“a”不会立即被销毁，可以在其生命周期结束之前进一步查询。

RVO机制

在上面Auto_ptr4示例的generateResource()函数中，当变量res通过value返回时，它将被移动而不是复制，即使res是一个l值。c++规范有一个特殊的规则，即函数按值返回的自动对象可以移动，即使它们是l值。这是有道理的，因为res无论如何都会在函数结束时被销毁!我们还不如窃取它的资源，而不是制作昂贵和不必要的拷贝。

虽然编译器可以移动l值的返回值，但在某些情况下，如果简单地完全省略复制(这完全避免了复制或移动的需要)，它甚至可以做得更好。在这种情况下，复制构造函数和移动构造函数都不会被调用。

禁用复制

在上面的Auto_ptr4类中，为了进行比较，我们保留了复制构造函数和赋值操作符。但在启用移动的类中，有时需要删除复制构造函数和复制赋值函数，以确保不会生成复制。在Auto_ptr类中，我们不想复制模板化对象T——因为它的开销很大，它甚至可能不支持复制!

下面是Auto_ptr的一个版本，它支持移动语义但不支持复制语义:

```
#include <iostream>

template<typename T>
class Auto_ptr5
{
  T* m_ptr;
public:
  Auto_ptr5(T* ptr = nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr5()
  {
    delete m_ptr;
  }

  // Copy constructor -- no copying allowed!
  Auto_ptr5(const Auto_ptr5& a) = delete;

  // Move constructor
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr5(Auto_ptr5&& a) noexcept
    : m_ptr(a.m_ptr)
  {
    a.m_ptr = nullptr;
  }

  // Copy assignment -- no copying allowed!
  Auto_ptr5& operator=(const Auto_ptr5& a) = delete;
```

```
  // Move assignment
  // Transfer ownership of a.m_ptr to m_ptr
  Auto_ptr5& operator=(Auto_ptr5&& a) noexcept
  {
    // Self-assignment detection
    if (&a == this)
      return *this;

    // Release any resource we're holding
    delete m_ptr;

    // Transfer ownership of a.m_ptr to m_ptr
    m_ptr = a.m_ptr;
    a.m_ptr = nullptr;

    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};
```

如果试图按值将Auto_ptr5 l-value传递给函数，编译器会抱怨初始化函数实参所需的复制构造函数已被删除。这很好，因为我们可能应该通过const l-value引用来传递Auto_ptr5！

Auto_ptr5(终于)是一个很好的智能指针类。而且，实际上标准库包含一个非常类似于这个类的类(您应该使用它)，名为std::unique_ptr。我们将在本章后面讨论更多关于std::unique_ptr的内容。