5、std::move_if_noexcept

Chapter	M
Name	xingzp

在第20.9课——异常规范和noexcept中，我们介绍了noexcept异常说明符和操作符，这是本课构建的基础。

我们还介绍了**强异常保证**，它保证如果函数被异常中断，不会泄露任何内存，也不会改变程序状态。特别是，所有的构造函数都应该支持强异常保证，这样，如果对象的构造失败，程序的其余部分就不会处于更改的状态。

移动构造函数异常问题

考虑这样一种情况:我们正在复制某个对象，由于某种原因复制失败了(例如，机器内存不足)。在这种情况下，被复制的对象不会受到任何损害，因为不需要修改源对象来创建副本。我们可以丢弃失败的副本，继续前进。支持**强异常保证**。

现在考虑一下我们移动一个物体的情况。移动操作将给定资源的所有权从源对象转移到目标对象。如果在所有权转移发生后，移动操作被异常中断，那么源对象将处于修改状态。如果源对象是一个临时对象，并且在移动后将被丢弃，这就不是问题——但对于非临时对象，我们现在已经破坏了源对象。为了遵守**强异常保证**，我们需要将资源移回源对象，但如果一旦移动失败，也不能保证移回会成功。

如何给移动构造函数**强异常保证**?避免在移动构造函数体中抛出异常非常简单，但是移动构造函数可能调用其他可能抛出异常的构造函数。以std::pair的移动构造函数为例，它必须尝试将源pair中的每个子对象移动到新的pair对象中。

```
// Example move constructor definition for std::pair
// Take in an 'old' pair, and then move construct the new pair's 'first' and 'second'
 subobjects from the 'old' ones
template <typename T1, typename T2>
pair<T1,T2>::pair(pair&& old)
  : first(std::move(old.first)),
    second(std::move(old.second))
{}
```

现在让我们使用两个类，MoveClass和CopyClass，我们将把它们组合在一起来演示使用移动构造函数的强异常保证问题:

```
#include <iostream>
#include <utility> // For std::pair, std::make_pair, std::move, std::move_if_noexcept
#include <stdexcept> // std::runtime_error

class MoveClass
{
private:
  int* m_resource{};

public:
  MoveClass() = default;

  MoveClass(int resource)
    : m_resource{ new int{ resource } }
  {}

  // Copy constructor
  MoveClass(const MoveClass& that)
  {
    // deep copy
    if (that.m_resource != nullptr)
    {
      m_resource = new int{ *that.m_resource };
    }
  }

  // Move constructor
  MoveClass(MoveClass&& that) noexcept
    : m_resource{ that.m_resource }
  {
    that.m_resource = nullptr;
  }

  ~MoveClass()
  {
    std::cout << "destroying " << *this << '\n';
```

```
    delete m_resource;
  }

  friend std::ostream& operator<<(std::ostream& out, const MoveClass& moveClass)
  {
    out << "MoveClass(";

    if (moveClass.m_resource == nullptr)
    {
      out << "empty";
    }
    else
    {
      out << *moveClass.m_resource;
    }

    out << ')';

    return out;
  }
};


class CopyClass
{
public:
  bool m_throw{};

  CopyClass() = default;

  // Copy constructor throws an exception when copying from a CopyClass object where i
ts m_throw is 'true'
  CopyClass(const CopyClass& that)
    : m_throw{ that.m_throw }
  {
    if (m_throw)
    {
      throw std::runtime_error{ "abort!" };
    }
  }
};

int main()
{
  // We can make a std::pair without any problems:
  std::pair my_pair{ MoveClass{ 13 }, CopyClass{} };

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  // But the problem arises when we try to move that pair into another pair.
  try
  {
    my_pair.second.m_throw = true; // To trigger copy constructor exception

    // The following line will throw an exception
    std::pair moved_pair{ std::move(my_pair) }; // We'll comment out this line later
    // std::pair moved_pair{ std::move_if_noexcept(my_pair) }; // We'll uncomment this
```

```
line later

    std::cout << "moved pair exists\n"; // Never prints
  }
  catch (const std::exception& ex)
  {
      std::cerr << "Error found: " << ex.what() << '\n';
  }

  std::cout << "my_pair.first: " << my_pair.first << '\n';

  return 0;
}
```

上述程序打印：

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(empty)
destroying MoveClass(empty)
```

让我们来探究一下发生了什么。第一行显示了用于初始化my_pair的临时MoveClass对象，一旦my_pair实例化语句被执行，它就会被销毁。它是空的，因为my_pair中的MoveClass子对象是从它移动构造的，所以下一行显示my_pair.first包含值为13的MoveClass对象。

第三行变得很有趣。我们通过复制构造它的CopyClass子对象(CopyClass没有移动构造函数)来创建moved_pair，但是由于我们更改了布尔标志，复制构造抛出了一个异常。moved_pair的构造被异常终止，它已经构造的成员被销毁。在本例中，MoveClass成员被销毁，输出销毁的MoveClass(13)变量。接下来我们看到main()中打印的Error found: abort!消息。

当我们试图打印my_pair。首先，它再次显示MoveClass成员为空。因为moved_pair是用std::move初始化的，所以MoveClass成员(MoveClass有一个移动构造函数)被移动构造，导致my_pair.first是空的。

最后，my_pair在main()的末尾被销毁。

总结一下上面的结果 : 创建moved_pair时，使用std::move(my_pair)尝试移动构造second时使用了CopyClass的复制构造函数（因为CopyClass没有移动构造函数）。此复制构造函数抛出异常，导致moved_pair的创建终止，moved_pair终止时销毁了已经构造的成员first，从而导致my_pair.first永久丢失。没有保留强异常保证。

---

std::move_if_noexcept救场

请注意，如果std::pair尝试进行复制而不是移动，那么上述问题是可以避免的。在这种情况下，moved_pair将无法构造，但my_pair将不会被修改。

但复制而不是移动有一个性能成本，我们不想为所有对象付出代价——理想情况下，我们希望在安全的情况下进行移动，否则进行复制。

幸运的是，c++有两种机制，在组合使用时，可以让我们做到这一点。首先，因为noexcept函数是不会抛出异常/不会失败的（no-throw/no-fail），所以它们隐式地满足强异常保证的标准。因此，noexcept的移动构造函数保证会成功。

其次，我们可以使用标准库函数std::move_if_noexcept()来确定应该执行移动还是复制。std::move_if_noexcept与std::move对应，使用方式相同。

如果编译器能够判断作为std::move_if_noexcept参数传递给它的对象在移动构造时不会抛出异常(或者如果对象是move-only且没有复制构造函数)，那么std::move_if_noexcept将执行与std::move()相同的操作(并返回转换为r值的对象)。否则，std::move_if_noexcept将返回对对象的正常的l值引用。

> 🙂 **Key insight**
> **如果对象具有noexcept移动构造函数，std::move_if_noexcept则返回可移动的r值，否则将返回可复制的l值。我们可以将noexcept说明符与std::move_if_noexcept一起使用，以便仅在存在强异常保证时使用移动语义(否则使用复制语义)。**

让我们像下面这样更新上一个例子中的代码:

```
//std::pair moved_pair{std::move(my_pair)}; // comment out this line now
std::pair moved_pair{std::move_if_noexcept(my_pair)}; // and uncomment this line
```

再次运行上述程序打印如下：

```
destroying MoveClass(empty)
my_pair.first: MoveClass(13)
destroying MoveClass(13)
Error found: abort!
my_pair.first: MoveClass(13)
destroying MoveClass(13)
```

如您所见，在抛出异常之后，子对象my_pair。第一个仍然指向值13。

std::pair的移动构造函数不是noexcept(从c++ 20开始)的，所以std::move_if_noexcept返回my_pair作为l值引用。这会导致通过复制构造函数(而不是移动构造函数)创建moved_pair。复制构造函数可以安全地抛出，因为它不修改源对象。

标准库经常使用std::move_if_noexcept来优化noexcept函数。例如，如果元素类型有noexcept修饰的移动构造函数，则std::vector::resize将使用移动语义，否则使用copy语义。这意味着std::vector对于具有noexcept修饰的移动构造函数的对象通常操作更快。

Warning
如果一个类型同时具有潜在抛出异常的移动语义和删除的复制语义(复制构造函数和复制赋值操作符不可用)，则std::move_if_noexcept将放弃强保证并调用移动语义。这种放弃强保证的条件在标准库容器类中非常普遍，因为它们经常使用std::move_if_noexcept。