使用初始化捕获来移动对象到闭包中

在某些场景下，按值捕获和按引用捕获都不是你所想要的。如果你有一个只能被移动的对象（例如**std::unique_ptr**或**std::future**）要进入到闭包里，使用C++11是无法实现的。如果你要复制的对象复制开销非常高，但移动的成本却不高（例如标准库中的大多数容器），并且你希望的是宁愿移动该对象到闭包而不是复制它。然而C++11却无法实现这一目标。

但如果你的编译器支持C++14，那又是另一回事了，它能支持将对象移动道闭包中。如果你的兼容支持C++14，那么请愉快地阅读下去。如果你仍然在使用仅支持C++11的编译器，也请愉快阅读，因为在C++11中有很多方法可以实现近似的移动捕获。

缺少移动捕获被认为是C++11的一个缺点，直接的补救措施是将该特性添加到C++14中，但标准化委员会选择了另一种方法。他们引入了一种新的捕获机制，该机制非常灵活，移动捕获是它执行的技术之一。新功能被称作初始化捕获，它几乎可以完成C++11捕获形式的所有工作，甚至能完成更多功能。默认的捕获模式使得你无法使用初始化捕获表示，但第31项说明提醒了你无论如何都应该远离这些捕获模式。（在C++11捕获模式所能覆盖的场景里，初始化捕获的语法有点不大方便。因此在C++11的捕获模式能完成所需功能的情况下，使用它是完全合理的）。

使用初始化捕获可以让你指定：

1. 从lambda生成的闭包类中的数据成员名称；
2. 初始化该成员的表达式；

这是使用初始化捕获将**std::unique_ptr**移动到闭包中的方法：

```
class Widget { // some useful type
public:
...
  bool isValidated() const;
  bool isProcessed() const;
  bool isArchived() const;
private: ...
};

auto pw = std::make_unique<Widget>(); // create Widget; see Item 21 for info on
std::make_unique configure *pw

auto func = [pw = std::move(pw)] // init data mbr in closure w/ std::move(pw)
                        { return pw->isValidated()
                                             && pw->isArchived(); };
```

上面的文本包含了初始化捕获的使用，"="的左侧是指定的闭包类中数据成员的名称，右侧则是初始化表达式。有趣的是，"="左侧的作用范围不同于右侧的作用范围。在上面的示例中，'='左侧的名称`pw`表示闭包类中的数据成员，而右侧的名称`pw`表示在lambda上方声明的对象，即由调用初始化的变量到调用`std::make_unique`。因此，`pw = std :: move(pw)`的意思是“在闭包中创建一个数据成员pw，并通过将`std::move`应用于局部变量pw的方法来初始化该数据成员。

一般中，lambda主体中的代码在闭包类的作用范围内，因此pw的使用指的是闭包类的数据成员。

在此示例中，注释`configure * pw`表示在由`std::make_unique`创建窗口小部件之后，再由lambda捕获到该窗口小部件的`std::unique_ptr`之前，该窗口小部件即pw对象以某种方式进行了修改。如果不需要这样的配置，即如果`std::make_unique`创建的`Widget`处于适合被lambda捕获的状态，则不需要局部变量`pw`，因为闭包类的数据成员可以通过直接初始化`std::make_unique`来实现：

```
auto func = [pw = std::make_unique<Widget>()] // init data mbr
                        { return pw->isValidated()              // in closure w/
                                            && pw->isArchived(); };    // result of
call // to make_unique
```

这清楚地表明了，这个C ++ 14的捕获概念是从C ++11发展出来的的，在C ++11中，无法捕获表达式的结果。 因此，初始化捕获的另一个名称是广义lambda捕获。
但是，如果您使用的一个或多个编译器不支持C ++ 14的初始捕获怎么办？ 如何使用不支持移动捕获的语言完成移动捕获？

请记住，lambda表达式只是生成类和创建该类型对象的一种方式而已。如果对于lambda，你觉得无能为力。 那么我们刚刚看到的C++ 14的示例代码可以用C ++11重新编写，如下所示：

```
class IsValAndArch {
public:
    using DataType = std::unique_ptr<Widget>; // "is validated and archived"
  explicit IsValAndArch(DataType&& ptr) // Item 25 explains
                    : pw(std::move(ptr)) {}         // use of std::move
  bool operator()() const
  { return pw->isValidated() && pw->isArchived(); }
private:
  DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

这个代码量比lambda表达式要多，但这并不难改变这样一个事实，即如果你希望使用一个C++11的类来支持其数据成员的移动初始化，那么你唯一要做的就是在键盘上多花点时间。

如果你坚持要使用lambda（并且考虑到它们的便利性，你可能会这样做），可以在C++11中这样使用：

1. 将要捕获的对象移动到由`std::bind`；
2. 将被捕获的对象赋予一个引用给lambda；

如果你熟悉std::bind，那么代码其实非常简单。如果你不熟悉std::bind，那可能需要花费一些时间来习惯改代码，但这无疑是值得的。

假设你要创建一个本地的`std::vector`，在其中放入一组适当的值，然后将其移动到闭包中。在C ++14中，这很容易实现：

```
std::vector<double> data; // object to be moved
                          // into closure
                          // populate data
auto func = [data = std::move(data)] { /* uses of data */ }; // C++14 init
capture
```

我已经对该代码的关键部分进行了高亮：要移动的对象的类型（`std::vector\<double>`），该对象的名称（数据）以及用于初始化捕获的初始化表达式（`std::move(data)`）。C++11的等效代码如下，其中我强调了相同的关键事项：

```
std::vector<double> data; // as above
auto func =
  std::bind(
// C++11 emulation
        [](const std::vector<double>& data) { /* uses of data */ }, // of init
capture
    std::move(data)
);
```

如lambda表达式一样，`std::bind`生产了函数对象。我将它称呼为由std::bind所绑定对象返回的函数对象。`std::bind`的第一个参数是可调用对象，后续参数表示要传递给该对象的值。

一个绑定的对象包含了传递给`std::bind`的所有参数副本。对于每个左值参数，绑定对象中的对应对象都是复制构造的。对于每个右值，它都是移动构造的。在此示例中，第二个参数是一个右值（`std::move`的结果，请参见第23项），因此将数据移动到绑定对象中。这种移动构造是模仿移动捕获的关键，因为将右值移动到绑定对象是我们解决无法将右值移动到C++11闭包中的方法。

当“调用”绑定对象（即调用其函数调用运算符）时，其存储的参数将传递到最初传递给`std::bind`的可调用对象。在此示例中，这意味着当调用func（绑定对象）时，func中所移动构造的数据副本将作为参数传递给传递给`std::bind`中的lambda。

该lambda与我们在C++14中使用的lambda相同，只是添加了一个参数data来对应我们的伪移动捕获对象。此参数是对绑定对象中数据副本的左值引用。（这不是右值引用，因尽管用于初始化数据副本的表达式（`std::move(data)`)为右值，但数据副本本身为左值。）因此，lambda将对绑定在对象内部的移动构造数据副本进行操作。

默认情况下，从lambda生成的闭包类中的`operator()`成员函数为`const`的。这具有在lambda主体内呈现闭包中的所有数据成员为`const`的效果。但是，绑定对象内部的移动构造数据副本不一定是`const`的，因此，为了防止在lambda内修改该数据副本，lambda的参数应声明为`const`引用。 如果将`lambda`声明为可变的，则不会在其闭包类中将`operator()`声明为const，并且在lambda的参数声明中省略`const`也是合适的：

```
auto func =
        std::bind(                                  // C++11 emulation
        [](std::vector<double>& data) mutable       // of init capture
        { /* uses of data */ },                                 // for
mutable lambda std::move(data)
);
```

因为该绑定对象存储着传递给`std::bind`的所有参数副本，所以在我们的示例中，绑定对象包含由lambda生成的闭包副本，这是它的第一个参数。 因此闭包的生命周期与绑定对象的生命周期相同。 这很重要，因为这意味着只要存在闭包，包含伪移动捕获对象的绑定对象也将存在。

如果这是您第一次接触`std::bind`，则可能需要先阅读您最喜欢的C ++11参考资料，然后再进行讨论所有详细信息。 即使是这样，这些基本要点也应该清楚：

- 无法将移动构造一个对象到C ++11闭包，但是可以将对象移动构造为C++11的绑定对象。
- 在C++11中模拟移动捕获包括将对象移动构造为绑定对象，然后通过引用将对象移动构造传递给lambda。
- 由于绑定对象的生命周期与闭包对象的生命周期相同，因此可以将绑定对象中的对象视为闭包中的对象。

作为使用`std::bind`模仿移动捕获的第二个示例，这是我们之前看到的在闭包中创建`std::unique_ptr`的C++14代码：

```
auto func = [pw = std::make_unique<Widget>()]   // as before,
                        { return pw->isValidated()                  // create pw
                        && pw->isArchived(); };                         // in
closure
```

这是C++11的模拟实现：

```
auto func = std::bind(
              [](const std::unique_ptr<Widget>& pw)
              { return pw->isValidated()
                              && pw->isArchived(); },
              std::make_unique<Widget>()
);
```

具备讽刺意味的是，这里我展示了如何使用`std::bind`解决C++11 lambda中的限制，但在条款34中，我却主张在`std::bind`上使用lambda。

但是，该条目解释的是在C++11中有些情况下`std::bind`可能有用，这就是其中一种。（在C++14中，初始化捕获和自动参数等功能使得这些情况不再存在。）

要谨记的是：

- 使用C ++14的初始化捕获将对象移动到闭包中。
- 在C ++11中，通过手写类或`std::bind`的方式来模拟初始化捕获。