Item25: 对右值引用使用`std::move`，对通用引用使用`std::forward`

右值引用仅绑定可以移动的对象。如果你有一个右值引用参数，你就知道这个对象可能会被移动：

```
class Widget {
  Widget(Widget&& rhs); //rhs definitely refers to an object eligible for moving
  ...
};
```

这是个例子，你将希望通过可以利用该对象右值性的方式传递给其他使用对象的函数。这样做的方法是将绑定次类对象的参数转换为右值。如Item23中所述，这不仅是`std::move`所做，而且是为它创建：

```
class Widget {
public:
  Widget(Widget&& rhs) :name(std::move(rhs.name)), p(std::move(rhs.p)) {...}
  ...
private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

另一方面（查看Item24），通用引用可能绑定到有资格移动的对象上。通用引用使用右值初始化时，才将其强制转换为右值。Item23阐释了这正是`std::forward`所做的：

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName) {      //newName is universal reference
    name = std::forward<T>(newName);
  }
  ...
}
```

总而言之，当传递给函数时右值引用应该无条件转换为右值（通过`std::move`），通用引用应该有条件转换为右值（通过`std::forward`）。

Item23 解释说，可以在右值引用上使用`std::forward`表现出适当的行为，但是代码较长，容易出错，所以应该避免在右值引用上使用`std::forward`。更糟的是在通用引用上使用`std::move`，这可能会意外改变左值。

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName) {
    name = std::move(newName);  //universal reference compiles, but is bad ! bad ! bad !
  }
  ...

private:
```

```
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName(); // factory function

Widget w;
auto n = getWidgetName(); // n is local variiable
w.setName(n); // move n into w! n's value now unkown
```

上面的例子，局部变量n被传递给`w.setName`，可以调用方对n只有只读操作。但是因为`setName`内部使用`std::move`无条件将传递的参数转换为右值，`n`的值被移动给w，n最终变为未定义的值。这种行为使得调用者蒙圈了。

你可能争辩说`setName`不应该将其参数声明为通用引用。此类引用不能使用`const`（Item 24），但是`setName`肯定不应该修改其参数。你可能会指出，如果const左值和右值分别进行重载可以避免整个问题，比如这样：

```
class Widget {
public:
  void setName(const std::string& newName) { // set from const lvalue
    name = newName;
  }
  void setName(std::string&& newName) { // set from rvalue
    name = std::move(newName);
  }
};
```

这样的话，当然可以工作，但是有缺点。首先编写和维护的代码更多；其次，效率下降。比如，考虑如下场景：

```
w.setName("Adela Novak");
```

使用通用引用的版本，字面字符串"Adela Novak"可以被传递给setName，在w内部使用了`std::string`的赋值运算符。w的name的数据成员直接通过字面字符串直接赋值，没有中间对象被创建。但是，重载版本，会有一个中间对象被创建。一次setName的调用会包括`std::string`的构造器调用（中间对象），`std::string`的赋值运算调用，`std::string`的析构调用（中间对象）。这比直接通过const char*赋值给`std::string`开销昂贵许多。实际的开销可能因为库的实现而有所不同，但是事实上，将通用引用模板替换成多个函数重载在某些情况下会导致运行时的开销。如果例子中的`Widget`数据成员是任意类型（不一定是`std::string`），性能差距可能会变得更大，因为不是所有类型的移动操作都像`std::string`开销较小（参看Item29）。

但是，关于重载函数最重要的问题不是源代码的数量，也不是代码的运行时性能。而是设计的可扩展性差。`Widget::setName`接受一个参数，可以是左值或者右值，因此需要两种重载实现，`n`个参数的话，就要实现2^n种重载。这还不是最坏的。有的函数---函数模板----接受无限制参数，每个参数都可以是左值或者右值。此类函数的例子比如`std::make_unique`或者`std::make_shared`。查看他们的的重载声明：

```
template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args);

template<class T, class... Args>
unique_ptr<T> make_unique(Args&&... args);
```

对于这种函数，对于左值和右值分别重载就不能考虑了：通用引用是仅有的实现方案。对这种函数，我向你保证，肯定使用`std::forward`传递通用引用给其他函数。

好吧，通常，最终。但是不一定最开始就是如此。在某些情况，你可能需要在一个函数中多次使用绑定到右值引用或者通用引用的对象，并且确保在完成其他操作前，这个对象不会被移动。这时，你只想在最后一次使用时，使用`std::move`或者`std::forward`。比如：

```
template<typename T>
void setSignText(T&& text)
{
  sign.setText(text);

  auto now = std::chrono::system_clock::now();

  signHistory.add(now, std::forward<T>(text));
}
```

这里，我们想要确保`text`的值不会被`sign.setText`改变，因为我们想要在`signHistory.add`中继续使用。因此`std::forward`只在最后使用。

对于`std::move`，同样的思路，但是需要注意，在有些稀少的情况下，你需要调用`std::move_if_noexcept`代替`std::move`。要了解何时以及为什么，参考Item 14。

如果你使用的按值返回的函数，并且返回值绑定到右值引用或者通用引用上，需要对返回的引用使用`std::move`或者`std::forward`。要了解原因，考虑`+`操作两个矩阵的函数，左侧的矩阵参数为右值（可以被用来保存求值之后的和）

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs){
  lhs += rhs;
  return std::move(lhs); // move lhs into return value
}
```

通过在返回语句中将lhs转换为右值，lhs可以移动到返回值的内存位置。如果`std::move`省略了

```
Matrix operator+(Matrix&& lhs, const Matrix& rhs){
  lhs += rhs;
  return lhs; // copy lhs into return value
}
```

事实上，lhs作为左值，会被编译器拷贝到返回值的内存空间。假定Matrix支持移动操作，并且比拷贝操作效率更高，使用`std::move`的代码效率更高。

如果Matrix不支持移动操作，将其转换为左值不会变差，因为右值可以直接被Matrix的拷贝构造器使用。如果Matrix随后支持了移动操作，`+`操作符的定义将在下一次编译时受益。就是这种情况，通过将`std::move`应用到返回语句中，不会损失什么，还可能获得收益。

使用通用引用和`std::forward`的情况类似。考虑函数模板`reduceAndCopy`收到一个未规约对象`Fraction`，将其规约，并返回一个副本。如果原始对象是右值，可以将其移动到返回值中，避免拷贝开销，但是如果原始对象是左值，必须创建副本，因此如下代码：

```
template<typename T>
Fraction reduceAndCopy(T&& frac) {
  frac.reduce();
  return std::forward<T>(frac); // move rvalue into return value, copy lvalue
}
```

如果`std::forward`被忽略，frac就是无条件复制到返回值内存空间。

有些开发者获取到上面的知识后，并尝试将其扩展到不适用的情况。

```
Widget makeWidget() {
  Widget w; //local variable
  ...   // configure w
  return w; // "copy" w into return value
}
```

想要优化copy的动作为如下代码:

```
Widget makeWidget() {
  Widget w; //local variable
  ...   // configure w
  return std::move(w); // move w into return value(don't do this!)
}
```

这种用法是有问题的，但是问题在哪?

在进行优化时，标准化委员会远领先于开发者，第一个版本的makeWidget可以在分配给函数返回值的内存中构造局部变量w来避免复制局部变量w的需要。这就是所谓的返回值优化（RVO），这在C++标准中已经实现了。

所以"copy"版本的makeWidget在编译时都避免了拷贝局部变量w，进行了返回值优化。（返回值优化的条件：1. 局部变量与返回值的类型相同；2. 局部变量就是返回值）。

移动版本的makeWidget行为与其名称一样，将w的内容移动到makeWidget的返回值位置。但是为什么编译器不使用RVO消除这种移动，而是在分配给函数返回值的内存中再次构造w呢？条件2中规定，仅当返回值为局部对象时，才进行RVO，但是move版本不满足这条件，再次看一下返回语句:

```
return std::move(w);
```

返回的已经不是局部对象w，而是局部对象w的引用。返回局部对象的引用不满足RVO的第二个条件，所以编译器必须移动w到函数返回值的位置。开发者试图帮助编译器优化反而限制了编译器的优化选项。

（译者注：本段即绕又长，大意为即使开发者非常熟悉编译器，坚持要在局部变量上使用`std::move`返回）

这仍然是一个坏主意。C++标准关于RVO的部分表明，如果满足RVO的条件，但是编译器选择不执行复制忽略，则必须将返回的对象视为右值。实际上，标准要求RVO，忽略复制或者将`sdt::move`隐式应用于返回的本地对象。因此，在makeWidget的"copy"版本中，编译器要不执行复制忽略的优化，要不自动将`std::move`隐式执行。

按值传递参数的情形与此类似。他们没有资格进行RVO，但是如果作为返回值的话编译器会将其视作右值。结果就是，如果代码如下:

```
Widget makeWidget(Widget w) {
  ...
  return w;
}
```

实际上，编译器的代码如下:

```
Widget makeWidget(Widget w){
  ...
  return std::move(w);
}
```

这意味着，如果对从按值返回局部对象的函数使用`std::move`，你并不能帮助编译器，而是阻碍其执行优化选项。在某些情况下，将`std::move`应用于局部变量可能是一件合理的事，但是不要阻碍编译器RVO。

需要记住的点

- 在右值引用上使用`std::move`，在通用引用上使用`std::forward`
- 对按值返回的函数返回值，无论返回右值引用还是通用引用，执行相同的操作
- 当局部变量就是返回值是，不要使用`std::move`或者`std::forward`