# CHAPTER 5 RValue References, Move Semantics and Perfect Forwarding

当你第一次了解到**移动语义**和**完美转发**的时候，它们看起来非常直观:

- **移动语义**使编译器有可能用廉价的移动操作来代替昂贵的复制操作。正如复制构造函数和复制赋值操作符给了你赋值对象的权利一样，移动构造函数和移动赋值操作符也给了控制移动语义的权利。移动语义也允许创建**只可移动**(move-only)的类型，例如`std::unique_ptr`, `std::future` 和 `std::thread`。
- **完美转发**使接收任意数量参数的函数模板成为可能，它可以将参数转发到其他的函数，使目标函数接收到的参数与被传递给转发函数的参数保持一致。

**右值引用**是连接这两个截然不同的概念的胶合剂。它隐藏在语言机制之下，使移动语义和完美转发变得可能。

你对这些特点(features)越熟悉，你就越会发现，你的初印象只不过是冰山一角。移动语义、完美转发和右值引用的世界比它所呈现的更加微妙。
举个例子，`std::move`并不移动任何东西，完美转发也并不完美。移动操作并不永远比复制操作更廉价；即便如此，它也并不总是像你期望的那么廉价。而且，它也并不总是被调用，即使在当移动操作可用的时候。构造`type&&`也并非总是代表一个右值引用。

无论你挖掘这些特性有多深，它们看起来总是还有更多隐藏起来的部分。幸运的是，它们的深度总是有限的。本章将会带你到最基础的部分。一旦到达，`C++11`的这部分特性将会具有非常大的意义。比如，你会掌握`std::move`和`sd::forward`的惯用法。你能够对`type&&`的歧义性质感到舒服。你会理解移动操作的令人惊奇的不同代价的背后真相。这些片段都会豁然开朗。在这一点上，你会重新回到一开始的状态，因为移动语义、完美转发和右值引用都会又一次显得直截了当。但是这一次，它们不再使人困惑。

在本章的这些小节中，非常重要的一点是要牢记**参数**(parameter)永远是**左值**(lValue)，即使它的类型是一个右值引用。比如，假设

```
void f(Widget&& w);
```

参数`w`是一个左值，即使它的类型是一个**Widget**的右值引用(如果这里震惊到你了，请重新回顾从本书第二页开始的关于左值和右值的总览。)

## Item 23: 理解std::move和std::forward

为了了解`std::move`和`std::forward`，一种有用的方式是从*它们不做什么*这个角度来了解它们。`std::move`不移动(move)任何东西，`std::forward`也不转发(forward)任何东西。在运行期间(run-time)，它们不做任何事情。它们不产生任何可执行代码，一字节也没有。

`std::move`和`std::forward`仅仅是执行转换(cast)的函数（事实上是函数模板）。`std::move`无条件的将它的参数转换为右值，而`std::forward`只在特定情况满足时下进行转换。
它们就是如此。这样的解释带来了一些新的问题，但是从根本上而言，这就是全部内容。

为了使这个故事更加的具体，这里是一个C++11的`std::move`的示例实现。它并不完全满足标准细则，但是它已经非常接近了。

```
template <typename T>                           //in namespace std
typename remove_reference<T>::type&&
move(T&& param)
{
    using ReturnType =                          // alias declaration;
    typename remove_reference<T>::type&&;       // 见 Item 9

    return static_cast<ReturnType>(param);
}
```

我为你们高亮了这段代码的两部分（译者注：markdown不支持代码段内高亮。高亮的部分为`move`和`static_cast`）。一个是函数名字，因为函数的返回值非常具有干扰性。而且我不想你们被它搞得晕头转向。另外一个高亮的部分是包含这段函数的本质的转换。正如你所见，`std::move`接受一个对象的引用（准确的说，一个通用引用(universal reference)，后见Item 24），返回一个指向同对象的引用。

该函数返回类型的`&&`部分表明`std::move`函数返回的是一个右值引用，但是，正如Item 28所解释的那样，如果类型`T`恰好是一个左值引用，那么`T&&`将会成为一个左值引用。为了避免如此，类型萃取器（type trait，见Item 9）`std::remove_reference`应用到了类型`T`上，因此确保了`&&`被正确的应用到了一个不是引用的类型上。这保证了`std::move`返回的真的是右值引用，这很重要，因为函数返回的右值引用是右值（rvalues）。因此，`std::move`将它的参数转换为一个右值，这就是它的全部作用。

此外，`std::move`在C++14中可以被更简单地实现。多亏了函数返回值类型推导（见Item 3)和标准库的模板别名`std::remove_reference_t`（见Item 9），`std::move`可以这样写：

```
template <typename T>
decltype(auto) move(T&& param)          //C++14;still in namesapce std
{
    using ReturnType = remove_referece_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

看起来更简单，不是吗？

因为`std::move`除了转换它的参数到右值以外什么也不做，有一些提议说它的名字叫`rvalue_cast`可能会更好。虽然可能确实是这样，但是它的名字已经是`std::move`，所以记住`std::move`做什么和不做什么很重要。它其实并不移动任何东西。

当然，右值本来就是移动操作的侯选者，所以对一个对象使用`std::move`就是告诉编译器，这个对象很适合被移动。所以这就是为什么`std::move`叫现在的名字：更容易指定可以被移动的对象。

事实上，右值只不过**经常**是移动操作的候选者。假设你有一个类，它用来表示一段注解。这个类的构造函数接受一个包含有注解的`std::string`作为参数，然后它复制该参数到类的数据成员（data member)。假设你了解Item 41,你声明一个值传递(by value)的参数：

```
class Annotation {
public:
    explicit Annotation(std::string text);  //将会被复制的参数
    ...                                     //如同 Item 41,
};                                          //值传递
```

但是`Annotation`类的构造函数仅仅是需要读取参数`text`的值，它并不需要修改它。为了和历史悠久的传统：能使用`const`就使用`const`保持一致，你修订了你的声明以使`text`变成`const`，

```
class Annotation {
public:
    explicit Annotation(const std::string text);
    ...
};
```

当复制参数`text`到一个数据成员的时候，为了避免一次复制操作的代价，你仍然记得来自Item 41的建议，把`std::move`应用到参数`text`上，因此产生一个右值，

```
class Annotation {
public:
    explicit Annotation(const std::string text)
    : value(std::move(text))    //"move" text到value上；这段代码执行起来
                                //并不如看起来那样
    {...}
    ...

    private:
        std::string value;
};
```

这段代码可以编译，可以链接，可以运行。这段代码将数据成员`value`设置为`text`的值。这段代码与你期望中的完美实现的唯一区别，是`text`并不是被移动到`value`，而是被**复制**。诚然，`text`通过`std::move`被转换到右值，但是`text`被声明为`const std::string`，所以在转换之前，`text`是一个左值的`const std::string`，而转换的结果是一个右值的`const std::string`，但是纵观全程，`const`属性一直保留。

当编译器决定哪一个`std::string`的构造函数被构造时，考虑它的作用，将会有两种可能性。

```
class string {                  //std::string事实上是
    public:                     //std::basic_string<char>的类型别名
    ...
    string(const string& rhs); //复制构造函数
    string(string&& rhs);       //移动构造函数
}
```

在类`Annotation`的构造函数的成员初始化列表(member initialization list)中，`std::move(text)`的结构是一个`const std::string`的右值。这个右值不能被传递给`std::string`的移动构造函数，因为移动构造函数只接受一个指向非常量(non-const)`std::string`的右值引用。然而，该右值却可以被传递给`std::string`的复制构造函数，因为指向常量的左值引用允许被绑定到一个常量右值上。因此，`std::string`在成员初始化的过程中调用了**复制构造函数**，即使`text`已经被转换成了右值。这样是为了确保维持常量属性的正确性。从一个对象中移动（Moving)出某个值通常代表着修改该对象，所以语言不允许常量对象被传递给可以修改他们的函数（例如移动构造函数）。

从这个例子中，可以总结出两点。第一，不要在你希望能移动对象的时候，声明他们为常量。对常量对象的移动请求会悄无声息的被转化为复制操作。第二点，`std::move`不仅不移动任何东西，而且它也不保证它执行转换的对象可以被移动。关于`std::move`，你能确保的唯一一件事就是将它应用到一个对象上，你能够得到一个右值。

关于`std::forward`的故事与`std::move`是相似的，但是与`std::move`总是**无条件**的将它的参数转换为右值不同，`std::forward`只有在满足一定条件的情况下才执行转换。`std::forward`是**有条件**的转换。要明白什么时候它执行转换，什么时候不，想想`std::forward`的典型用法。
最常见的情景是一个模板函数，接收一个通用引用参数(universal reference parameter)，并将它传递

给另外的函数:

```
void process(const Widget& lvalArg);  //左值处理
void process(Widget&& rvalArg);         //右值处理

template <typename T>                  //用以转发参数到process的模板
void logAndProcess(T&& param)
{
    auto now =                         //获取现在时间
        std::chrono::system_clock::now();
    makeLogEntry("calling 'process',now);
    process(std::forward<T>(param));
}
```

考虑两次对`logAndProcess`的调用，一次左值为参数，一次右值为参数，

```
Widget w;

logAndProcess(w);               //call with lvalue
logAndProcess(std::move(w));    //call with rvalue
```

在`logAndProcess`函数的内部，参数`param`被传递给函数`process`。函数`process`分别对左值和右值参数做了重载。当我们使用左值来调用`logAndProcess`时，自然我们期望该左值被当作左值转发给`process`函数，而当我们使用右值来调用`logAndProcess`函数时，我们期望`process`函数的右值重载版本被调用。

但是参数`param`，正如所有的其他函数参数一样，是一个左值。每次在函数`logAndProcess`内部对函数`process`的调用，都会因此调用函数`process`的左值重载版本。为防如此，我们需要一种机制(mechanism)：当且仅当传递给函数`logAndProcess`的用以初始化参数`param`的值是一个右值时，参数`param`会被转换为有一个右值。这就是为什么`std::forward`是一个**有条件**的转换：它只把由右值初始化的参数，转换为右值。

你也许会想知道`std::forward`是怎么知道它的参数是否是被一个右值初始化的。举个例子，在上述代码中，`std::forward`是怎么分辨参数`param`是被一个左值还是右值初始化的？ 简短的说，该信息藏在函数`logAndProcess`的模板参数`T`中。该参数被传递给了函数`std::forward`，它解开了含在其中的信息。该机制工作的细节可以查询 Item 28.

考虑到`std::move`和`std::forward`都可以归结于**转换**，他们唯一的区别就是`std::move`总是执行转换，而`std::forward`偶尔为之。你可能会问是否我们可以免于使用`std::move`而在任何地方只使用`std::forward`。 从纯技术的角度，答案是yes: `std::forward`是可以完全胜任，`std::move`并非必须。当然，其实两者中没有哪一个函数是**真的必须**的，因为我们可以到处直接写转换代码，但是我希望我们能同意：这将相当的，嗯，让人恶心。

`std::move`的吸引力在于它的便利性： 减少了出错的可能性，增加了代码的清晰程度。考虑一个类，我们希望统计有多少次移动构造函数被调用了。我们只需要一个静态的计数器(static counter)，它会在移动构造的时候自增。假设在这个类中，唯一一个非静态的数据成员是`std::string`，一种经典的移动构造函数（例如，使用std::move)可以被实现如下:

```
class Widget{
public:
    Widget(Widget&& rhs)
    : s(std::move(rhs.s))
    {
        ++moveCtorCalls;
    }
private:
    static std::size_t moveCtorCalls;
    std::string s;
}
```

如果要用`std::forward`来达成同样的效果，代码可能会看起来像

```
class Widget{
public:
    Widget(Widget&& rhs)                    //不自然，不合理的实现
    : s(std::forward<std::string>(rhs.s))
    {
        ++moveCtorCalls;
    }
    ...
}
```

注意，第一，`std::move`只需要一个函数参数(rhs.s)，而`std::forward`不但需要一个函数参数(rhs.s)，还需要一个模板类型参数`std::string`。其次，我们转发给`std::forward`的参数类型应当是一个**非引用**(non-reference)，因为传递的参数应该是一个右值（见 Item 28)。 同样，这意味着`std::move`比起`std::forward`来说需要打更少的字，并且免去了传递一个表示我们正在传递一个右值的类型参数。同样，它根绝了我们传递错误类型的可能性，（例如，`std::string&`可能导致数据成员`s`被复制而不是被移动构造）。

更重要的是，`std::move`的使用代表着无条件向右值的转换，而使用`std::forward`只对绑定了右值的引用进行到右值转换。这是两种完全不同的动作。前者是典型地为了移动操作，而后者只是传递（亦作转发）一个对象到另外一个函数，保留它原有的左值属性或右值属性。因为这些动作实在是差异太大，所以我们拥有两个不同的函数（以及函数名）来区分这些动作。

记住:

- `std::move`执行到右值的无条件的转换，但就自身而言，它不移动任何东西。
- `std::forward`只有当它的参数被绑定到一个右值时，才将参数转换为右值。
- `std::move`和`std::forward`在运行期什么也不做。