C++右值

右值引用 (Rvalue Referene) 是 C++11 新标准中引入的新特性，它实现了转移语义 (Move Sementics) 和精确传递 (Perfect Forwarding)。它的主要目的有两个方面：

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
- 能够更简洁明确地定义泛型函数。

右值与左值的定义

C++(包括 C) 中所有的表达式和变量要么是左值，要么是右值。通俗的左值的定义就是非临时对象，可以在多条语句中使用的对象。所有的变量都满足这个定义，在多条代码中都可以使用，都是左值。右值是指临时的对象，它们只在当前的语句中有效。请看示例：

- `int i = 0;` 在这条语句中，i 是左值，0 是临时值，就是右值。在下面的代码中，i 可以被引用，0 就不可以了。立即数都是右值。
- `((i>0) ? i : j) = 1;` 右值也可以出现在赋值表达式的左边，但是不能作为赋值的对象，因为右值只在当前语句有效，赋值没有意义。

  在 C++11 之前，右值是不能被引用的，最大限度就是用常量引用绑定一个右值，如：`const int &a = 1;`

左值和右值的语法符号

左值的声明符号为”&”， 为了和左值区分，右值的声明符号为”&&”，示例程序：

```
void process_value(int& i) {
 std::cout << "LValue processed: " << i << std::endl;
}

void process_value(int&& i) {
 std::cout << "RValue processed: " << i << std::endl;
}

int main() {
 int a = 0;
 process_value(a);
 process_value(1);
}
// result:
// LValue processed: 0
// RValue processed: 1
```

Process_value 函数被重载，分别接受左值和右值。由输出结果可以看出，临时对象是作为右值处理的。

但是如果临时对象通过一个接受右值的函数传递给另一个函数时，就会变成左值，因为这个临时对象在传递过程中，变成了命名对象，示例程序：

```
void process_value(int& i) {
 std::cout << "LValue processed: " << i << std::endl;
}
```

```
void process_value(int&& i) {
 std::cout << "RValue processed: " << i << std::endl;
}

void forward_value(int&& i) {
 process_value(i);
}

int main() {
 int a = 0;
 process_value(a);
 process_value(1);
 forward_value(2);
}

// result
// LValue processed: 0
// RValue processed: 1
// LValue processed: 2
```

转移语义的定义

右值引用是用来支持转移语义的。转移语义可以将资源 (堆，系统对象等) 从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝以及销毁，能够大幅度提高 C++ 应用程序的性能。临时对象的维护 (创建和销毁) 对性能有严重影响。

转移语义是和拷贝语义相对的，可以类比文件的剪切与拷贝，当我们将文件从一个目录拷贝到另一个目录时，速度比剪切慢很多。

通过转移语义，临时对象中的资源能够转移其它的对象里。

在现有的 C++ 机制中，我们可以定义拷贝构造函数和赋值函数。要实现转移语义，需要定义转移构造函数，还可以定义转移赋值操作符。对于右值的拷贝和赋值会调用转移构造函数和转移赋值操作符。如果转移构造函数和转移拷贝操作符没有定义，那么就遵循现有的机制，拷贝构造函数和赋值操作符会被调用。

普通的函数和操作符也可以利用右值引用操作符实现转移语义。

实现转移构造函数和转移赋值函数

```
MyString(MyString&& str) {
   std::cout << "Move Constructor is called! source: " << str._data << std::endl;
   _len = str._len;
   _data = str._data;
   str._len = 0;
   str._data = NULL;
}

MyString& operator=(MyString&& str) {
   std::cout << "Move Assignment is called! source: " << str._data << std::endl;
   if (this != &str) {
     _len = str._len;
     _data = str._data;
     str._len = 0;
```

```
        str._data = NULL;
    }
    return *this;
}
```

- 参数（右值）的符号必须是右值引用符号，即“&&”。
- 参数（右值）不可以是常量，因为我们需要修改右值。
- 参数（右值）的资源链接和标记必须修改。否则，右值的析构函数就会释放资源。转移到新对象的资源也就无效了。

有了右值引用和转移语义，我们在设计和实现类时，对于需要动态申请大量资源的类，应该设计转移构造函数和转移赋值函数，以提高应用程序的效率。

标志库函数 `std::move`

简单实现

```
template<typename T>
decltype(auto) move(T&& param)
{
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

`std::move` 并不是 move，它只是一个cast而已，它可以帮助我们触发move semantics. 既然编译器只对右值引用才能调用转移构造函数和转移赋值函数，而所有命名对象都只能是左值引用，如果已知一个命名对象不再被使用而想对它调用转移构造函数和转移赋值函数，也就是把一个左值引用当做右值引用来使用，标准库提供了函数 std::move，这个函数以非常简单的方式将左值引用转换为右值引用。

精确传递 (Perfect Forwarding)

精确传递适用于这样的场景：需要将一组参数原封不动的传递给另一个函数。

左值 / 右值和 const/non-const。 精确传递就是在参数传递过程中，所有这些属性和参数值都不能改变。在泛型函数中，这样的需求非常普遍。

下面举例说明。函数 forward_value 是一个泛型函数，它将一个参数传递给另一个函数 process_value。

```
template <typename T> void forward_value(const T& val) {
 process_value(val);
}
template <typename T> void forward_value(T& val) {
 process_value(val);
}
```

函数 forward_value 为每一个参数必须重载两种类型，T& 和 const T&，否则，下面四种不同类型参数的调用中就不能同时满足 :

```
int a = 0;
const int &b = 1;
forward_value(a); // int&
forward_value(b); // const int&
forward_value(2); // int&
```

对于一个参数就要重载两次，也就是函数重载的次数和参数的个数是一个正比的关系。这个函数的定义次数对于程序员来说，是非常低效的。我们看看右值引用如何帮助我们解决这个问题：

```
template <typename T> void forward_value(T&& val) {
 process_value(val);
}
```

只需要定义一次，接受一个右值引用的参数，就能够将所有的参数类型原封不动的传递给目标函数。四种不用类型参数的调用都能满足，参数的左右值属性和 const/non-cosnt 属性完全传递给目标函数 process_value。

C++11 中定义的 T&& 的推导规则为：右值实参为右值引用，左值实参仍然为左值引用。一句话，就是参数的属性不变。这样也就完美的实现了参数的完整传递。