

【转】C++11 标准新特性：右值引用与转移语义

VS2013出来了，对于C++来说，最大的改变莫过于对于C++11新特性的支持，在网上搜了一下C++11的介绍，发现这篇文章非常不错，分享给大家同时自己作为存档。

原文地址：[http://www.ibm.com/developerworks/cn/aix/library/1307\\_lisl\\_c11/](http://www.ibm.com/developerworks/cn/aix/library/1307_lisl_c11/)

C++ 的新标准 C++11 已经发布一段时间了。本文介绍了新标准中的一个特性，右值引用和转移语义。这个特性能够使代码更加简洁高效。

新特性的目的

右值引用 (Rvalue Referene) 是 C++ 新标准 (C++11, 11 代表 2011 年 ) 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：

- 1. 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。
- 2. 能够更简洁明确地定义泛型函数。

左值与右值的定义

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，那些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。请看下列示例：

- 1. 简单的赋值语句  
如：`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。

- 2. 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。  
  
如：`((i>0) ? i : j) = 1;`

在这个例子中，`0` 作为右值出现在了“=”的左边。但是赋值对象是 `i` 或者 `j`，都是左值。  
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;
```

在这种情况下，右值不能被修改的。但是实际上右值是可以被修改的，如：

```
T().set().get();
```

`T` 是一个类，`set` 是一个函数为 `T` 中的一个变量赋值，`get` 用来取出这个变量的值。在这句中，`T()` 生成一个临时对象，就是右值，`set()` 修改了变量的值，也就修改了这个右值。

既然右值可以被修改，那么就可以实现右值引用。右值引用能够方便地解决实际工程中的问题，实现非常有吸引力的解决方案。

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：



公告



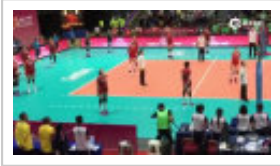
高峰LBJ 北京 大兴区

加关注

哈哈哈哈哈！

新浪视频：【当国歌响起时】（女排大奖赛香港站）中国女排姑娘们正在赛前热身，突然，国歌误播了.....姑娘们中断热身，肃穆，行注目礼。👍在场的观众也都自发站起来行礼。👍👍现场戳

→\_→<http://t.cn/RLM34vh>



7月20日 17:15 转发 | 评论

昵称：天堂大鸟  
园龄：5年2个月  
粉丝：20  
关注：1  
+加关注

<	2015年7月						>
日	一	二	三	四	五	六	
28	29	30	1	2	3	4	
5	6	7	8	9	10	11	
12	13	14	15	16	17	18	
19	20	21	22	23	24	25	
26	27	28	29	30	31	1	
2	3	4	5	6	7	8	

搜索

常用链接

我的随笔  
我的评论  
我的参与  
最新评论  
我的标签  
更多链接

```
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);
}
```

运行结果：

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);
}
```

运行结果：

LValue processed: 0  
RValue processed: 1  
LValue processed: 2

虽然 **2** 这个立即数在函数 **forward\_value** 接收时是右值，但到了 **process\_value** 接收时，变成了左值。

## 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。

## 我的标签

- [c++ \(11\)](#) [MFC \(2\)](#)
- [unicode \(2\)](#) [vs \(1\)](#)
- [vs2008 \(1\)](#) [VS2010 \(1\)](#)
- [vs2013 \(1\)](#) [WIN7 \(1\)](#)
- [代码风格 \(1\)](#) [断点续传 \(1\)](#)
- [更多](#)

## 随笔档案

- [2015年3月 \(1\)](#)
- [2015年1月 \(1\)](#)
- [2014年6月 \(2\)](#)
- [2014年5月 \(1\)](#)
- [2014年3月 \(1\)](#)
- [2013年8月 \(2\)](#)
- [2013年7月 \(2\)](#)
- [2013年6月 \(2\)](#)
- [2013年5月 \(3\)](#)
- [2013年4月 \(1\)](#)
- [2013年3月 \(4\)](#)
- [2013年2月 \(1\)](#)
- [2013年1月 \(2\)](#)
- [2012年12月 \(1\)](#)
- [2012年10月 \(2\)](#)
- [2012年7月 \(1\)](#)
- [2012年4月 \(1\)](#)
- [2012年3月 \(1\)](#)
- [2012年2月 \(2\)](#)
- [2011年12月 \(1\)](#)
- [2011年11月 \(1\)](#)
- [2011年9月 \(2\)](#)
- [2011年6月 \(1\)](#)
- [2011年5月 \(3\)](#)
- [2011年4月 \(2\)](#)
- [2011年3月 \(1\)](#)
- [2011年2月 \(3\)](#)
- [2011年1月 \(2\)](#)
- [2010年12月 \(3\)](#)
- [2010年11月 \(4\)](#)
- [2010年10月 \(5\)](#)
- [2010年9月 \(3\)](#)
- [2010年8月 \(10\)](#)
- [2010年7月 \(9\)](#)
- [2010年6月 \(1\)](#)

## 相册

- [james\(1\)](#)
- [聊天+传送文件程序\(4\)](#)
- [普吉岛照片\(1\)](#)
- [图书管理系统Ribbon\(4\)](#)


## 最新评论

- [1. Re:线程中使用UpdateDa...  
谢谢，原来原因是这样的。  
--捡了一棵小白菜](#)
- [2. Re:同步和异步的区别（...  
路过  
--Martin-li](#)
- [3. Re:【转】C++11 标准新...  
修正下：forward\\_value\(2\);  
// int&&, 这句话编译是过了，但是是int&的。  
--清清飞扬](#)
- [4. Re:【转】C++11 标准新...  
forward\\_value\(b\); // const  
int& forward\\_value\(2\); // int&&  
以上2句针对： template void  
forward\\_value\(T&&.....](#)

## 实现转移构造函数和转移赋值函数

以一个简单的 `string` 类为示例，实现拷贝构造函数和拷贝赋值操作符。

示例程序：



```
class MyString {
private:
    char* _data;
    size_t _len;
    void _init_data(const char *s) {
        _data = new char[_len+1];
        memcpy(_data, s, _len);
        _data[_len] = '\0';
    }
public:
    MyString() {
        _data = NULL;
        _len = 0;
    }


    MyString(const char* p) {
        _len = strlen (p);
        _init_data(p);
    }

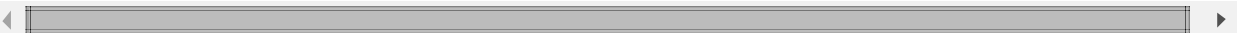
    MyString(const MyString& str) {
        _len = str._len;
        _init_data(str._data);
        std::cout << "Copy Constructor is called! source: " << str._data << std::endl;
    }

    MyString& operator=(const MyString& str) {
        if (this != &str) {
            _len = str._len;
            _init_data(str._data);
        }
        std::cout << "Copy Assignment is called! source: " << str._data << std::endl;
        return *this;
    }

    virtual ~MyString() {
        if (_data) free(_data);
    }
};

int main() {
    MyString a;
    a = MyString("Hello");
    std::vector<MyString> vec;
    vec.push_back(MyString("World"));
}
```






运行结果：

```
Copy Assignment is called! source: Hello
Copy Constructor is called! source: World
```

这个 `string` 类已经基本满足我们演示的需要。在 `main` 函数中，实现了调用拷贝构造函数的操作和拷贝赋值操作符的操作。`MyString("Hello")` 和 `MyString("World")` 都是临时对象，也就是右值。虽然它们是临时的，但程序仍然调用了拷贝构造和拷贝赋值，造成了没有意义的资源申请和释放的操作。如果能够直接使用临时对象已经申请的资源，既能节省资源，有能节省资源申请和释放的时间。这正是定义转移语义的目的。

我们先定义转移构造函数。



```
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;
}
```

--清清飞扬

5. Re:VC++ MFC 多线程及...  
写得挺好

--hn\_xs\_zhongjx

## 阅读排行榜

1. 同步和异步的区别（转）（...
2. 线程同步的几种方式（转...
3. 简单的 C++ SOCKET编...
4. memset用法详解(转)（1...
5. 初涉Ribbon界面简单编程...

## 评论排行榜

1. 初涉Ribbon界面简单编程...
2. 【原创】聊天+传送文件+...
3. 5月35日临近，Google...
4. c++ 请抛弃匈牙利命名法...
5. VC++ MFC 多线程及线...

## 推荐排行榜

1. 同步和异步的区别（转）（...
2. 5月35日临近，Google...
3. c++ 请抛弃匈牙利命名法...
4. 生成随机端口函数(2)
5. 初涉Ribbon界面简单编程...

```
}

```

和拷贝构造函数类似，有几点需要注意：

1. 参数（右值）的符号必须是右值引用符号，即“&&”。
2. 参数（右值）不可以是常量，因为我们需要修改右值。
3. 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。

现在我们定义转移赋值操作符。

```
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;
}

```

这里需要注意的问题和转移构造函数是一样的。

增加了转移构造函数和转移复制操作符后，我们的程序运行结果为：

```
Move Assignment is called! source: Hello
Move Constructor is called! source: World

```

由此看出，编译器区分了左值和右值，对右值调用了转移构造函数和转移赋值操作符。节省了资源，提高了程序运行的效率。

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。

## 标准库函数 `std::move`

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，怎么做呢？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。

示例程序：

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

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

int main() {
    int a = 0;
    ProcessValue(a);
    ProcessValue(std::move(a));
}

```

运行结果：

```
LValue processed: 0
RValue processed: 0

```

`std::move`在提高 `swap` 函数的的性能上非常有帮助，一般来说，`swap`函数的通用定义如下：



```
template <class T> swap(T& a, T& b)
{
    T tmp(a);    // copy a to tmp
    a = b;        // copy b to a
    b = tmp;      // copy tmp to b
}
```

有了 `std::move`，`swap` 函数的定义变为：

```
template <class T> swap(T& a, T& b)
{
    T tmp(std::move(a)); // move a to tmp
    a = std::move(b);    // move b to a
    b = std::move(tmp);  // move tmp to b
}
```

通过 `std::move`，一个简单的 `swap` 函数就避免了 3 次不必要的拷贝操作。

## 精确传递 (Perfect Forwarding)

本文采用精确传递表达这个意思。“Perfect Forwarding”也被翻译成完美转发，精准转发等，说的都是一个意思。

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。

“原封不动”不仅仅是参数的值不变，在 C++ 中，除了参数值之外，还有一下两组属性：

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`。

`forward_value` 的定义为：

```
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`。这个解决方案不是简洁优雅吗？

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

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。

右值引用，表面上看只是增加了一个引用符号，但它对 C++ 软件设计和类库的设计有非常大的影响。它既能简化代码，又能提高程序运行效率。每一个 C++ 软件设计师和程序员都应该理解并能够应用它。我们在设计类的时候如果有动态申请的资源，也应该设计转移构造函数和转移拷

贝函数。在设计类库时，还应该考虑 `std::move` 的使用场景并积极使用它。

## 总结

右值引用和转移语义是 C++ 新标准中的一个重要特性。每一个专业的 C++ 开发人员都应该掌握并应用到实际项目中。在有机会重构代码时，也应该思考是否可以应用新也行。在使用之前，需要检查一下编译器的支持情况。

标签: [c++](#) , [C++11](#) , [vs2013](#)

绿色通道:

[好文要顶](#)

[关注我](#)

[收藏该文](#)

[与我联系](#)



天堂大鸟

关注 - 1

粉丝 - 20

[+加关注](#)

0

0

(请您对文章做出评价)

« 上一篇: [c++ 请抛弃匈牙利命名法 - 变量命名代码风格的建议。](#)

» 下一篇: [5 月 35 日临近, Google 无法访问, 可以使用 Google IP 来解决。](#)

posted @ 2014-03-20 22:12 天堂大鸟 阅读(2441) 评论(2) 编辑 收藏

### 评论列表

#1楼

2015-02-10 11:51 清清飞扬

`forward_value(b); // const int&`  
`forward_value(2); // int&&`

以上2句针对: `template <typename T> void forward_value(T&& val)`编译无法通过呀! 你怎么还说"这个解决方案简洁优雅"呢?

支持(0) 反对(0)

#2楼

2015-02-10 11:57 清清飞扬

修正下: `forward_value(2); // int&&`, 这句话编译是过了, 但是是int&的。

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问网站首页](#)。

【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库

【热门】支持Visual Studio2015的葡萄城控件新品发布暨夏季促销



最新IT新闻:

- 拉勾网模式之殇，颠覆还是骗局
- 开发者自述：在国内做独立游戏很难 心态要平和
- Touch ID遥控一切，苹果获得指纹识别远程控制技术专利
- 世界首个仿生眼植入手术完成 让患者重见光明
- 苹果新专利暗示iPhone 7设计更接近iPhone 4

» 更多新闻...

【教程】Android开发工程师极速养成计划

0基础入门+项目实战，3个月学习，快速开发属于你自己的APP！



最新知识库文章：

- 代码审查的价值——为何做、何时做、如何做？
- 所有程序员都应该遵守的11条规则
- RESTful架构详解
- 优秀程序员眼中的整洁代码
- 怎样看待比自己强的人
- » 更多知识库文章...