

[文章](#) [问答](#) [论坛](#) [东西](#) [休息室](#) [?](#)

Search for articles, questions,

手表



C++11 移动语义，右值参考

BrainlessLabs.com

2014 年 9 月 10 日 [MPL](#)

评价我: 4.08/5 (5 票)

在本文中，我们将讨论 C++ 的移动语义。

在本文中，我们将讨论 C++ 的移动语义。我们将尝试弄清楚究竟什么是移动。

问题陈述

复制并不是所有情况下的最佳解决方案。某些情况需要移动，因为复制可能意味着资源的重复，并且这可能是一项繁重的任务。

另一个问题是临时的。这些临时文件可能会记录内存并减慢 C++ 的执行速度。

解决方案

解决方案是通知移动语义。我们将逐渐发现这是什么。

右值参考

RValues 是 C++11 的新增功能。我们将看到它的目的是什么以及它为什么被实施。

的最初定义 **lvalues** 和 **rvalues** 如下：

根据 C 风格定义，an **lvalue** 是一个可以出现在赋值左侧或右侧 **rvalue** 的表达式，而 an **rvalue** 是一个只能出现在赋值右侧的表达式。

C++

复制代码

```
int a = 42;
int b = 43;

// a and b are both l-values:
a = b; // ok
b = a; // ok
a = a * b; // ok

// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42; // error, rvalue on left hand side of assignment
```

C++ 及其用户定义类型引入了一些关于可修改性和可分配性的微妙之处，导致此定义不正确。所以现在我们要讲的 **lvalue** 是一个表示内存位置的表达式。这 **lvalue** 让我们可以获取位置的地址。什么是 **rvalue**? 很简单，任何不是 **lvalue**。

现在让我们正式地更好地定义这些术语和属性。

首先，什么是表达式？

- 表达式是一系列运算符和操作数。表达式是指定计算的语句。它告诉计算机或说 C++ 做什么。
- 一个表达式可以产生一个像“1+2; // 它的值为3”这样的值
- 表达式也可以像函数调用一样有副作用。
- 表达式可以是简单的，也可以是复杂的。
- 每个表达式具有类型和一个值的类别，即，如果表达式是 **lvalue**, **rvalue** 等

左值

An **lvalue** 是标识非临时对象或非成员函数的表达式。

- **lvalue** 可以取 **a** 的地址。
- 可修改，即 **nonconst lvalue** 可以用在“=”的左边
- **lvalue** 可用于初始化 **lvalue** 引用。
- 有时在允许的情况下，**lvalue** 可以有不完整的类型。
- 指定域的表达式（例如 **s.x** 其中 **s** 是 **type** 的对象 **struct S { int x:3; };**）是一个 **lvalue** 表达式（或者 **xvalue** 如果是一个）：它可以用在赋值运算符的左侧，但它的地址不能被使用，并且一个非-**const lvalue** 引用不能绑定到它。一个 **const lvalue** 参考值可以从一个位字段被初始化 **lvalue**，但位字段的临时副本将被制成：它不会直接绑定到比特字段。

例子

- 作用域中的变量或函数的名称，与类型无关，例如 **std::cin** 或 **std::endl**。即使变量的类型是 **rvalue** 引用，由其名称组成的 **lvalue** 表达式也是一个表达式。
- 如果函数或重载运算符的返回类型是 **lvalue** 引用，则为函数调用或重载运算符表达式，例如 **std::getline(std::cin, str)** 或 **std::cout << 1** 或 **str1 = str2** 或 **++iter**
- 内置预增和预减、解引用、赋值和复合赋值、下标（数组除外 **xvalue**）、成员访问（**static** 非引用成员 **xvalues**、成员枚举器和非 **static** 成员函数除外）、成员如果左侧操作数为 **lvalue**，则通过指向数据成员的指针访问，如果右侧操作数为，则为逗号运算符 **lvalue**，如果第二个和第三个操作数为，则为三元条件 **lvalues**。
- 将表达式转换为 **lvalue** 引用类型。
- 字符串字面量
- 如果函数的返回类型是 **rvalue** 对函数类型的引用，则为函数调用表达式
- 将表达式转换为 **rvalue** 对函数的引用。

右值

纯 **rvalue** (**prvalue**) 是标识临时对象（或其一个）的表达式，**subobject** 或者是与任何对象无关的值。

- 它可以是一个右值
- **prvalue** 不能是多态的：它标识的对象的动态类型始终是表达式的类型
- 非类非数组 **prvalue** 不能被 **const** 限定。
- **prvalue** 不能有不完整的类型（类型除外 **void**，见下文）
- 表达式 **obj.func** 和 **ptr->func**，其中 **func** 是非 **static** 成员函数，而表达式 **obj.*mfp** 和 **ptr->*mfp** 其中 **mfp** 是指向成员函数的指针，被归类为 **prvalue** 表达式，但它们不能用于初始化引用、作为函数参数或用于任何目的，除非作为函数调用表达式的左侧参数，例如 **(pobj->*ptr)(args)**。
- 函数调用表达式返回 **void**、将表达式转换为 **[cpp]void[/cpp]** 和 **[cpp]throw-expressions[/cpp]** 被归类为 **prvalue** 表达式，但它们不能用于初始化引用或作为函数参数。它们可以在某些上下文中使用（例如，在它自己的一行中，作为逗号运算符的左参数等）以及在 **return** 函数返回的语句中使用 **void**

例子

- 文字（**string** 文字除外），例如 **42** 或 **true** 或 **nullptr**。

- 函数调用或重载操作表达式，如果该函数的或重载的操作者的 **return** 类型不是一个参考，例如 **str.substr(1, 2)** 或 **str1 + str2**
- 内置的自增和自减、算术和逻辑运算符、比较运算符、地址运算符、成员枚举器的成员访问、非 **static** 成员函数或 **static** 右值的非非引用数据成员，通过指向数据成员 **rvalue** 或非 **static** 成员函数的指针进行成员访问，右侧操作数为的逗号运算符，**rvalue** 第二个或第三个操作数不是的三元条件 **lvalues**。
- 将表达式转换为引用类型以外的任何类型。
- Lambda 表达式，例如 **[](int x){return x*x;}**

值

An **xvalue** 是标识“**eXpiring**”对象的表达式，即可以从中移动的对象。由 **xvalue** 表达式标识的对象可能是无名临时对象，也可能是作用域中的命名对象，或任何其他类型的对象，但如果用作函数参数，**xvalue** 将始终绑定到 **rvalue** 引用重载（如果可用）。

- 它可以是 **rvalue** 或
- 它也可以是一个 **gvalue**
- 像 **prvalues**，**xvalues** 绑定到 **rvalue** 引用
- 与不同 **prvalues**，an **xvalue** 可能是多态的，而非类 **xvalue** 可能是 cv 限定的。

例子

- 如果函数或重载运算符的返回类型是 **rvalue** 对对象类型的引用，则为函数调用或重载运算符表达式，例如 **std::move(val)**
- **rvalue** 对对象类型的引用的强制转换表达式，例如 **static_cast<T&&>(val)** 或 **(T&&)val**
- 一个非 **static** 类成员访问表达式，其中对象表达式是一个 **xvalue**
- 指向成员的指针表达式，其中第一个操作数是 an **xvalue**，第二个操作数是指向数据成员的指针。

重力值

A **glvalue** (“generalized **lvalue**”) 是一个表达式，它要么是 an 要么是 **lvalue** an **xvalue**。

- 大多数情况下，它的属性适用于 pre-C++11 **lvalues**
- A **glvalue** 可以隐式转换为 **prvalue** with **lvalue-to-rvalue**、数组到指针或函数到指针的隐式转换。
- A **glvalue** 可能是多态的：它标识的对象的动态类型不一定 **static** 是表达式的类型。

右值

An **rvalue** 是一个表达式，它要么是 a 要么是 **prvalue** an **xvalue**。

- 它具有适用于这两种性质 **xvalues** 和 **prvalues**，这意味着它们适用于预C++11 **rvalues** 以及
- 一个地址 **rvalue** 不能被占用：**&int()**、**&i++[3]**、**&42**、和 **&std::move(val)** 是无效的。
- An **rvalue** 可用于初始化 **const lvalue** 引用，在这种情况下，由标识的对象的生命周期 **rvalue** 会延长，直到引用的范围结束。
- An **rvalue** 可用于初始化 **rvalue** 引用，在这种情况下，由标识的对象的生命周期 **rvalue** 会延长，直到引用的范围结束。
- 当用作函数参数并且当函数的两个重载可用时，一个接受 **rvalue** 引用参数，另一个接受参数 **lvalue** 引用 **const**，**rvalues** 绑定到 **rvalue** 引用重载（因此，如果复制和移动构造函数都可用，则 **rvalue** 参数调用移动构造函数，以及同样的复制和移动赋值运算符）。

移动

假设我们有一个 3D 模型类。模型类保存作为图像文件的纹理、可以产生数千个的顶点、每个顶点的颜色信息。像这样说：

C++

缩小▲ 复制代码

```
class Vertex {
    // Members not imp
public:
```

```

void addVertex ( /*vertex type*/ ) {
}
~Vertex ( ) {
    // Destroy verted
}
};

class Texture {
    // Members not imp
public:
    void load (/*info*/ ) {
        // Heavy duty image Loading
    }
    ~Texture ( ) {
        // Destroy
    }
};

class Model3D {
private:
    Vertex* _ver;
    Texture* _tex;
public:
    void initialize ( ) {
        _ver = new Vertex;
        _tex = new Texture;
        for ( int i = 0; i<10000; ++i ) {
            // Some more processing
            _ver->addVertex ( );
        }
        for ( int i = 0; i<500; ++i ) {
            _tex->load ( );
        }
    }
    ~Model3D ( ) {
        delete _ver;
        delete _tex;
    }
};

Model3D retGraphics ( ) {
    Model3D g;
    // Do some operation and return
    return g;
}

Model3D g1 = retGraphics ( );

```

在这里，如您所见，**ThreeD** 模型类执行了一些重型顶点和纹理加载。现在来看看声明 `Model3D g1 = retGraphics ();`。该语句可以转换为以下伪代码。

C++

复制代码

```

Model3D tempG;
Model3D retGraphics ( ) {
    Model3D g;
    // Do some operation and return
    // g will die with the scope, so copy it to a temp object
    tempG = g; // Clone the resources call Model3D::operator =( Model3D& ) on tempG
    g->~Model3D( );
}
Model3D g1 = tempG; // Clone tempG. Call Model3D::operator =( Model3D& ) on g1
tempG->~Model3D ( );

```

正如你所看到的，有一个临时的参与。这意味着顶点和纹理的破坏和加载发生在 2ce。这是一个耗时且不必要的过程。所以现在聪明的程序员只需要编写一些代码来实际进行资源交换，而不是让资源被破坏。这又是一项既费时又无聊的工作，但所有人都必须这样

做，以增加源大小。如果语言做到了，从而减轻程序员的负担不是很好吗？好吧，C++ 正是通过**移动**功能做到了这一点。所以它会做类似的事情：

C++

复制代码

```
Model3D& Model3D::operator = ( <move type> rhs ) {
    //swap _ver
    //swap _tex
}
```

这就是 C++ 使用 move 类型创建重载的原因，这是一种特殊类型，用于告诉编译器移动资源而不是执行删除和构造操作。使用移动类型时，编译器会处理以下选择：

- 移动类型必须是参考
- 当在两个重载之间进行选择时，一个是普通引用，另一个是神秘类型，那么 **rvalues** 必须更喜欢神秘类型
- **lvalues** 必须更喜欢普通参考

那么这个招式究竟是什么呢？这是 **rvalue** 参考，即 **Model3D&&**。

Model3D& 称为 **lvalue** 参考。那么 **rvalue** 现在引用的属性是什么？

- 在函数重载期间，决议 **lvalue** 更喜欢 **lvalue** 引用并且 **rvalue** 更喜欢 **rvalue** 引用。

C++

复制代码

```
void f ( Model3D& m ); // Lvalue reference overload.
void f ( Model3D&& m ); // rvalue reference overload.

f ( g1 ); // Here g1 is lvalue, so call void f ( Model3D& m );
f ( retGraphics ( ) ); // Here rvalue is needed. so void f ( Model3D&& m ); is called.
```

- 我们可以用重载任何函数 **rvalue**。但主要是在实践中，复制构造函数和赋值运算符。

那么如果你实现 **rvalue** 并忘记了 **lvalue** 重载会发生什么？嗯，自己试试吧。我们稍后会介绍。

有关移动和更多信息 **rvalue**，请参阅[博客](#)。

参考书目

- [Thomas Becker](#) 解释的 C++ 右值引用。
- [Mikael Kilpeläinen](#) 的左值和右值



本文最初发表于[http://brainlesslabs.com/cpp-cxx-cxx11-cpp11-move-semantics-rvalue-reference-lvalue?](http://brainlesslabs.com/cpp-cxx-cxx11-cpp11-move-semantics-rvalue-reference-lvalue?utm_source=rss&utm_medium=rss&utm_campaign=cpp-cxx-cxx11-cpp11-move-semantics-右值参考左值)
utm_source=rss&utm_medium=rss&utm_campaign=cpp-cxx-cxx11-cpp11-move-semantics-右值参考左值

执照

本文以及任何相关的源代码和文件均根据[The Mozilla Public License 1.1 \(MPL 1.1\)](#)获得许可

分享

关于作者

**BrainlessLabs.com**建筑师
印度 🇮🇳手表
该会员

我喜欢探索技术的不同方面。尝试新事物，并获得快乐。我的兴趣是编程语言和成像。但在其他事情上工作也不难。算法让我在喝咖啡休息时感到高兴。

我基本上用 C++ 编写代码，但 JAVA 对我来说并不陌生。我也知道很少的脚本语言。基本上我觉得知道一门编程语言只是一个无...

[展示更多](#)

评论和讨论

[添加评论或问题](#)[电子邮件提醒](#)[第一](#) [页上一](#) [下一页](#)

有趣的错字 ✨

Stefan_Lang 15-Sep-14 22:52

感谢 Deeksha 对其进行编辑和完善。 ✨

BrainlessLabs.com 12-Sep-14 21:50

字符串字面量 ✨

szulak 12-Sep-14 19:22

回复：字符串文字 ✨

BrainlessLabs.com 12-Sep-14 21:43

我的投票2 ✨

Austin Mullins 12-Sep-14 2:55

回复：我投了2票 ✨

BrainlessLabs.com 12-Sep-14 11:58[刷新](#)

1

[📄 一般](#) [📰 新闻](#) [💡 建议](#) [❓ 问题](#) [🐛 错误](#) [✅ 答案](#) [😄 笑话](#) [👍 赞美](#) [🗣️ 咆哮](#) [👤 管理员](#)

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。

[永久链接](#)[广告](#)[隐私](#)[Cookie](#)[使用条款](#)布局: [固定](#) | [体液](#)

文章 Copyright 2014 by BrainlessLabs.com

所有其他版权 © CodeProject ,

1999-2021 Web01 2.8.20210930.1