

【Modern C++】深入理解左值、右值

原创 雨乐 高性能架构探索 2022-02-16 12:08

收录于合集

#C/C++系列

28个



高性能架构探索

专注于分享干货，硬货，欢迎关注 😊

58篇原创内容

公众号

你好，我是雨乐！

作为C/C++开发人员，在平时的项目开发过程中，或多或少的听过左值和右值的概念，甚至在编译器报错的时候，遇到过 `lvalue` 和 `rvalue` 等字样；甚至使用过 `std::move()`，但是不知道其含义。作为多年的C++开发人员，一直以来，对左值右值的理解没有一个系统的认识，总感觉似懂非懂。今天，借助本文，详细的介绍下这些知识点，并从代码实例的角度去分析什么是左值或者右值，同时，也算是给自己知识点做一个总结。

背景

作为C++开发人员，相信我们都写过如下代码：

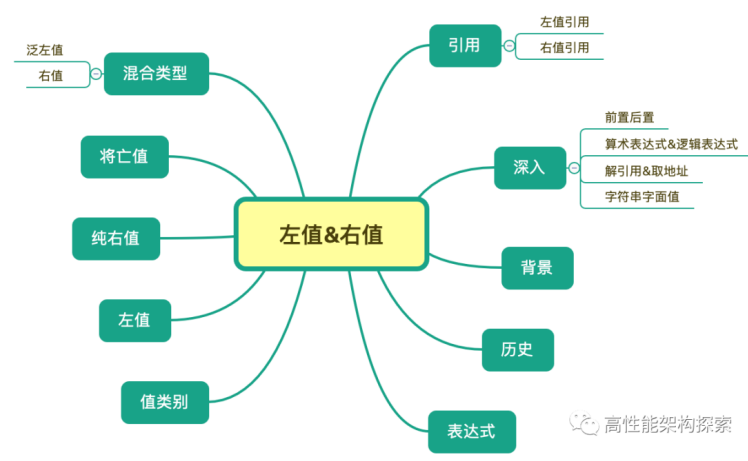
```
void fun(int &x) {  
    //  
}  
  
int main() {  
    fun(10);  
    return 0;  
}
```

在编译的时候，会提示如下：

```
invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
```

其中上述报错中的rvalue就是10，也就是说10就是rvalue，那么到底什么是rvalue，rvalue的意义是什么？这就是本文的目的，通过本文，让你彻底搞清楚什么C++下的值类别，以及如何区分左值、纯右值和将亡值。

本文的主要内容如下图所示：



历史

在正式介绍左值和右值之前，我们先了解下其历史。

编程语言CPL第一次引入了值类别，不过其定义比较简单，即对于赋值运算符，在运算符左边的为左值，在运算符右边的为右值。

C语言遵循与CPL类似的分类法，但是弱化了赋值的作用，C语言中的表达式被分为 **左值** 和其它(函数和非对象值)，其中左值被定义为标识一个对象的表达式。不过，C语言中的左值与CPL中的左值区别是，在C语言中lvalue是 **locator value** 的简写，因此lvalue对应了一块内存地址。

C++11之前，左值遵循了C语言的分类法，但与C不同的是，其将非左值表达式统称为右值，函数为左值，并添加了引用能绑定到左值但唯有const的引用能绑定到右值的规则。几种非左值的C表达式在C++中成为了左值表达式。

自C++11开始，对值类别又进行了详细分类，在原有左值的基础上增加了纯右值和消亡值，并对以上三种类型通过是否具名(identity)和可移动(moveable)，又增加了glvalue和rvalue两种组合类型，在后面的内容中，会对这几种类型进行详细讲解。

表达式

C/C++代码是由标识符、表达式和语句以及一些必要的符号(大括号等)组成。

表达式由按照语言规则排列的运算符，常量和变量组成。一个表达式可以包含一个或多个操作数，零个或多个运算符来计算值。每个表达式都会产生一些值，该值将在赋值运算符的帮助下分配给变量。

在C/C++中，表达式有很多种，我们常见的有前后缀表达式、条件运算符表达式等。字面值(literal)和变量(variable)是最简单的表达式，函数的返回值也被认为是表达式。

表达式是可求值的，对表达式求值可得到一个结果，这个结果有两个属性：

- 类型。这个我们很常见，比如int、string、引用或者我们自定义的类。类型确定了表达式可以进行哪些操作。
- 值类别(在下节中会细讲)。

值类别

在上节中，我们提到表达式是可求值的，而值类别就是求值结果的属性之一。

在C++11之前，表达式的值分为左值和右值两种，其中右值就是我们理解中的字面值1、true、NULL等。

自C++11开始，表达式的值分为 左值(lvalue, left value)、将亡值(xvalue, expiring value)、纯右值(pvalue, pure rvalue) 以及两种混合类别 泛左值(glvalue, generalized lvalue) 和 右值(rvalue, right value) 五种。

这五种类别的分类基于表达式的两个特征：

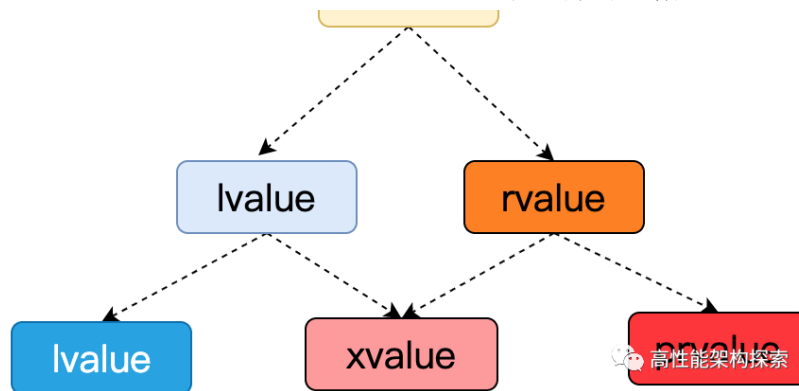
- 具名(identity):可以确定表达式是否与另一表达式指代同一实体，例如通过比较它们所标识的对象或函数的（直接或间接获得的）地址
- 可被移动：移动构造函数、移动赋值运算符或实现了移动语义的其他函数重载能够绑定于这个表达式

结合上述两个特征，对五种表达式值类别进行重新定义：

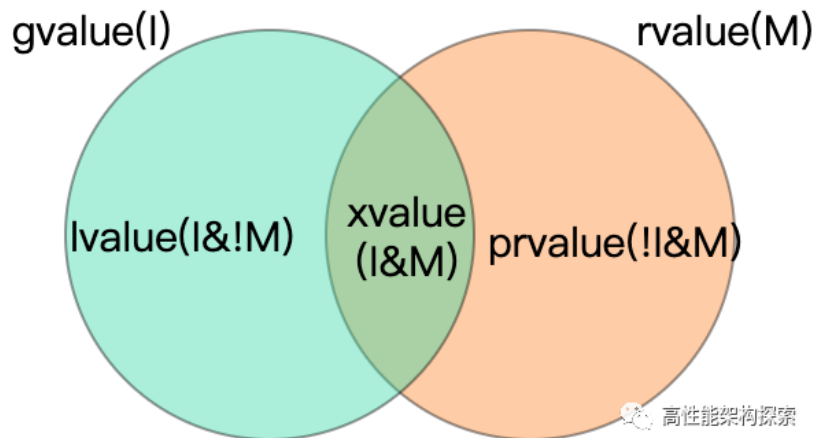
- lvalue:具名且不可被移动
- xvalue:具名且可被移动
- prvalue:不具名且可被移动
- glvalue:具名，lvalue和xvalue都属于glvalue
- rvalue:可被移动的表达式，prvalue和xvalue都属于rvalue

用图表示如下：

value



从glvalue和rvalue出发，将具名(identity)和可移动两个特征结合起来，如下图所示：



在上图中，I代表identity，M代表moveable。以xvalue为例，在上图中xvalue为(I&M)，即代表具名且可移动。

对于identity，有些文章译为 有身份的，有些文章译为 具名的，本文统一称为 具名的。

左值

左值(lvalue,left value)，顾名思义就是赋值符号左边的值。准确来说，左值是表达式结束（不一定是赋值表达式）后依然存在的对象。

可以将左值看作是一个关联了名称的内存位置，允许程序的其他部分来访问它。在这里，我们将“名称”解释为任何可用于访问内存位置的表达式。所以，如果 arr 是一个数组，那么 arr[1] 和 *(arr+1) 都将被视为相同内存位置的“名称”。

左值具有以下特征：

- 可通过取地址运算符获取其地址
- 可修改的左值可用作内建赋值和内建符合赋值运算符的左操作数

- 可以用来初始化左值引用(后面有讲)

那么哪些都是左值呢?查了相关资料,做了些汇总,基本覆盖了所有的类型:

- 变量名、函数名以及数据成员名
- 返回左值引用的函数调用
- 由赋值运算符或复合赋值运算符连接的表达式, 如(a=b, a-=b等)
- 解引用表达式*ptr
- 前置自增和自减表达式(++a, ++b)
- 成员访问 (点) 运算符的结果
- 由指针访问成员 (->) 运算符的结果
- 下标运算符的结果([])
- 字符串面值("abc")

为了能够更加清晰地理解左值, 我们举例:

```
int a = 1; // a是左值
T& f();
f(); //左值
++a; //左值
--a; //左值
int b = a; //a和b都是左值
struct S* ptr = &obj; // ptr为左值
arr[1] = 2; // 左值
int *p = &a; // p为左值
*p = 10; // *p为左值
class MyClass{};
MyClass c; // c为左值
"abc"
```

对于一个表达式, 凡是对其取地址 (&) 操作可以成功的都是左值

纯右值

在前面有提过, 自C++11开始, 纯右值(pvalue, pure rvalue)相当于之前的右值, 那么什么是纯右值呢?

字面值或者函数返回的非引用都是纯右值。

以下表达式的值都是纯右值：

- 字面值(字符串字面值除外)，例如1, 'a', true等
- 返回值为非引用的函数调用或操作符重载，例如：str.substr(1, 2), str1 + str2, or it++
- 后置自增和自减表达式(a++, a--)
- 算术表达式
- 逻辑表达式
- 比较表达式
- 取地址表达式
- lambda表达式

为了加深对右值的理解，下面的例子是常见的纯右值：

```
nullptr;
true;
1;
int fun();
fun();

int a = 1;
int b = 2;
a + b;

a++;
b--;

a > b;
a && b;
```

纯右值特征：

- 等同于C++11之前的右值
- 不会是多态
- 不会是抽象类型或数组
- 不会是不完全类型

将亡值

将亡值(xvalue, expiring value), 顾名思义即将消亡的值, 是C++11新增的跟右值引用相关的表达式, 通常是将要被移动的对象(移为他用), 比如返回右值引用T&&的函数返回值、std::move的返回值, 或者转换为T&&的类型转换函数的返回值。

将亡值可以理解为通过“盗取”其他变量内存空间的方式获取到的值。在确保其他变量不再被使用、或即将被销毁时, 通过“盗取”的方式可以避免内存空间的释放和分配, 能够延长变量值的生命期。(通过右值引用来续命)。

xvalue 只能通过两种方式来获得, 这两种方式都涉及到将一个左值赋给(转化为)一个右值引用:

- 返回右值引用的函数的调用表达式, 如 `static_cast<T&&>(t)`; 该表达式得到一个 xvalue
- 转换为右值引用的转换函数的调用表达式, 如: `std::move(t)`、`static_cast<T&&>(t)`

下面通过几个代码来详细分析什么是将亡值:

```
std::string fun() {  
    std::string str;  
    // ...  
    return str;  
}  
  
std::string s = fun();
```

在函数fun()中, str是一个局部变量, 并在函数结束时候被返回。

在C++11之前, `s = fun();`会调用拷贝构造函数, 会将整个str复制一份, 然后把str销毁。如果str特别大的话, 会造成大量额外开销。在这一行中, s是左值, fun()是右值(纯右值), fun()产生的那个返回值作为一个临时值, 一旦str被s复制后, 将被销毁, 无法获取、也不能修改。

自C++11开始, 引入了move语义, 编译器会将这部分优化成move操作, 即不再是之前的复制操作, 而是move。此时, str会被进行隐式右值转换, 等价于 `static_cast<std::string&&>(str)`, 进而此处的 s 会将 foo 局部返回的值进行移动。

无论是C++11之前的拷贝, 还是C++11的move, str在填充(拷贝或者move)给s之后, 将被销毁, 而被销毁的这个值, 就成为将亡值。

将亡值就定义了这样一种行为：具名的临时值、同时又能够被move。

混合类型

泛左值

泛左值（**glvalue, generalized lvalue**），又称为**广义左值**，是具名表达式，对应了一块内存。glvalue有lvalue和xvalue两种形式。

一个表达式是具名的，则称为glvalue，例子如下：

```
struct S{
    int n;
};

S fun();
S s;
s;
std::move(s);

fun();
S{};
S{}.n;
```

在上述代码中：

- 定义了结构体S和函数fun()
- 第6行声明了类型为S的变量s，因为其是具名的，所以是glvalue
- 第七行同上，因为s具名，所以为glvalue
- 第8行中调用了move函数，将左值s转换成xvalue，所以是glvaue
- 第10行中，fun()是不具名的，是纯右值，所以不是glvalue
- 第11行中，生成一个不具名的临时变量，是纯右值，所以不是glvalue
- 第12行中，n具名，所以是glvalue

glvalue的特征如下：

- 可以自动转换成prvalue
- 可以是多态的
- 可以是不完整类型，如前置声明但未定义的类型

右值

右值(**rvalue**, **right value**)是指可以移动的表达式。prvalue和xvalue都是rvalue, 具体的示例见下文。

rvalue具有以下特征:

- 无法对rvalue进行取地址操作。例如: `&1` , `&(a + b)` , 这些表达式没有意义, 也编译不过。
- rvalue不能放在赋值或者组合赋值符号的左边, 例如: `3 = 5` , `3 += 5` , 这些表达式没有意义, 也编译不过。
- rvalue可以用来初始化const左值引用 (见下文) 。例如: `const int& a = 1` 。
- rvalue可以用来初始化右值引用 (见下文) 。
- rvalue可以影响函数重载: 当被用作函数实参且该函数有两种重载可用, 其中之一接受右值引用的形参而另一个接受 `const` 的左值引用的形参时, 右值将被绑定到右值引用的重载之上。

深入

经过前面的内容, 我们对左值和右值(纯右值和将亡值)有了一个初步的认识, 在本节, 我们借助一些例子, 来加深对左值和右值的理解。

前置自增(减)是左值, 后置自增(减)是纯右值

代码如下:

```
int i = 0;
++i;
--i;
i++;
i--;
```

在上面代码中, 我们定义了一个int类型的变量i, 并初始化为0。

- `++i`的操作是对i加1后再赋值给i, 所以`++i`的结果是具名的, 名称就是i, 所以`++i`是左值
- 对于`i++`而言, 先将i的值进行拷贝(此处假设拷贝到临时变量ii), 然后再对i加1, 最后返回ii(其实不存在的, 为了在此表述方便)。所以`i++`是不具名的, 因此不是glvaue, 所以`i++`是右值, 又因为不具名, 且是右值, 所以`i++`是纯右值

- 同理，`--i`是左值，`i--`是纯右值

算术表达式是纯右值

代码如下：

```
int x = 0;
int y = 0;
x + y;
x && y;
x == y;
```

在上述代码中，`x + y`得到的是一个不具名的临时对象，所以`x+y`是纯右值；而`x && y`和`x == y`得到的是一个bool常量值，要么是true要么是false，所以是纯右值。

解引用是左值，取地址是纯右值

代码如下：

```
int x = 0;
int *y = &x;
*y = 1;
&y;
```

`*y`得到的是`y`指向地址的实际值，所以`&(*y)`是合法的，因此`*y`是左值；对`&y`操作得到的是一个地址，即一个long值，所以是一个字面值，因此`&y`是纯右值。

字符串字面值是左值

字符串字面值为左值，这个比较特殊。在前面提到过字面值都是纯右值(字符串字面值除外)，一个很重要的原因，就是可以字符串字面值可以 **获取地址**，

下面代码在编译器中可正常编译且运行：

```
std::cout << &"abc" << std::endl;
```

这是因为 C++将字符串面值实现为char型数组，实实在在地为每个字符都分配了空间并且允许程序员对其进行操作。如果从存储区的概念来理解，那就是字符串面值存储在 常量区。

引用

既然提到了左值右值，就得提一下引用。

在C++11之前，引用分为左值引用和常量左值引用两种，但是自C++11起，引入了右值引用，也就是说，在C++11中，包含如下3中引用：

- 左值引用
- 常量左值引用(不希望被修改)
- 右值引用

左值引用和常量左值引用，我们很常见，如下代码：

```
std::string str = "abc";  
std::string &s = str;  
  
const int &a = 10;  
  
int &b = 10; // 错
```

在上述代码中，s是一个左值引用，而a是一个const 左值引用。那么，为什么最后一句 `int &b = 10;` 编译器会报错呢？这是因为10是常量，而常量是右值，一个右值怎么能够被左值引用去引用呢。

那么什么是右值引用呢？右值引用就是引用右值的引用，这不废话嘛😁。

在C++11中引入了右值引用，因为右值的生命周期很短，右值引用的引入，使得可以延长右值的生命周期。在C++中规定，右值引用是&&即由2个&表示，而左值引用是一个&表示。右值引用的作用是为了 绑定右值。

为了能区分左值引用和右值引用，代码如下：

```
int a = 1;  
int &rb = a; // b为左值引用  
int &&rbb = a; // 错误, a是左值, 右值引用不能绑定左值
```

```
int &&rb1 = 1; // 正确, 1为右值
int &rb1 = i * 2; // 错误, i * 2是右值, 而rb1位左值引用
int &&rb2 = i * 2; // 正确
const int &c = 1; // 正确
const int &c1 = i * 2; // 正确
```

在这里, 我们需要特别注意的一点就是 右值引用虽然是引用右值, 但是其本身是左值, 以下代码为例:

```
int &&a = 1;
```

在上述代码中, a是一个右值引用, 但是其本身是左值, 合适因为:

- a出现在等号(=)的左边
- 可以对a取地址

我们在前面有提到过, 一个表达式有两个属性, 分别为类型和值类别。本节说的 左值引用和右值引用就属于类型, 而 左值和右值则属于值类别范畴, 这个概念很重要, 千万不能混淆。

可能有人会问, 除了自己根据规则区分左值引用和右值引用, 有没有更快更准确的方式来判断呢? 其实, 系统提供了API, 如下:

```
std::is_lvalue_reference
is_rvalue_reference

int a = 1;
int &ra = a;
int &&b = 1;

std::cout << std::is_lvalue_reference<decltype(ra)>::value << std::endl;

std::cout << std::is_rvalue_reference<decltype(ra)>::value << std::endl;

std::cout << std::is_rvalue_reference<decltype(b)>::value << std::endl;
```

输出结果:

```
1  
0  
1
```

结语

这篇文章是在整理了大量资料，结合自己的理解之后完成的。左值右值这种本身就比较抽象，在写文的过程中，发现有些东西，很难用文字来描述。在写这篇文章的过程中，也纠正了自己长久以来对左值右值的疑惑，因为这块确实比较复杂，所以文章中难免有出错或者不周全的地方，希望您批评指正。

好了，今天的文章就到这里，我们下期见！

参考

https://en.cppreference.com/w/cpp/language/value_category

<https://www.internalpointers.com/post/understanding-meaning-lvalues-and-rvalues-c>

<https://www.fatalerrors.org/a/left-value-reference-and-right-value-reference-of-c-c-class-and-object.html> https://www.bogotobogo.com/cplusplus/C11/4_C11_Rvalue_Lvalue.php

<https://users.soe.ucsc.edu/~pohl/code/lvalue.htm>

如果对本文有疑问可以加笔者**微信**直接交流，笔者也建了C/C++相关的技术群，有兴趣的可以联系笔者加群。



往期**精彩**回顾



[智能指针-使用、避坑和实现](#)

[内存泄漏-原因、避免以及定位](#)


[GDB调试-从入门实践到原理](#)

[【线上问题】P1级公司故障，年终奖不保](#)

[【性能优化】高效内存池的设计与实现](#)

[2万字|30张图带你领略glibc内存管理精髓](#)

点个关注吧!



高性能架构探索

专注于分享干货，硬货，欢迎关注😊

58篇原创内容

公众号

收录于合集 #C/C++系列 28

上一篇

编译器之返回值优化

下一篇

【线上故障】通过系统日志分析和定位

喜欢此内容的人还喜欢

一文弄懂Python中的Map、Filter和Reduce函数

AI算法之道



超详细的Python基础教程.pdf（最新版python3.9）

程序员森芋



Go语言中常见100问题-#86 Sleeping in unit tests

数据小冰

