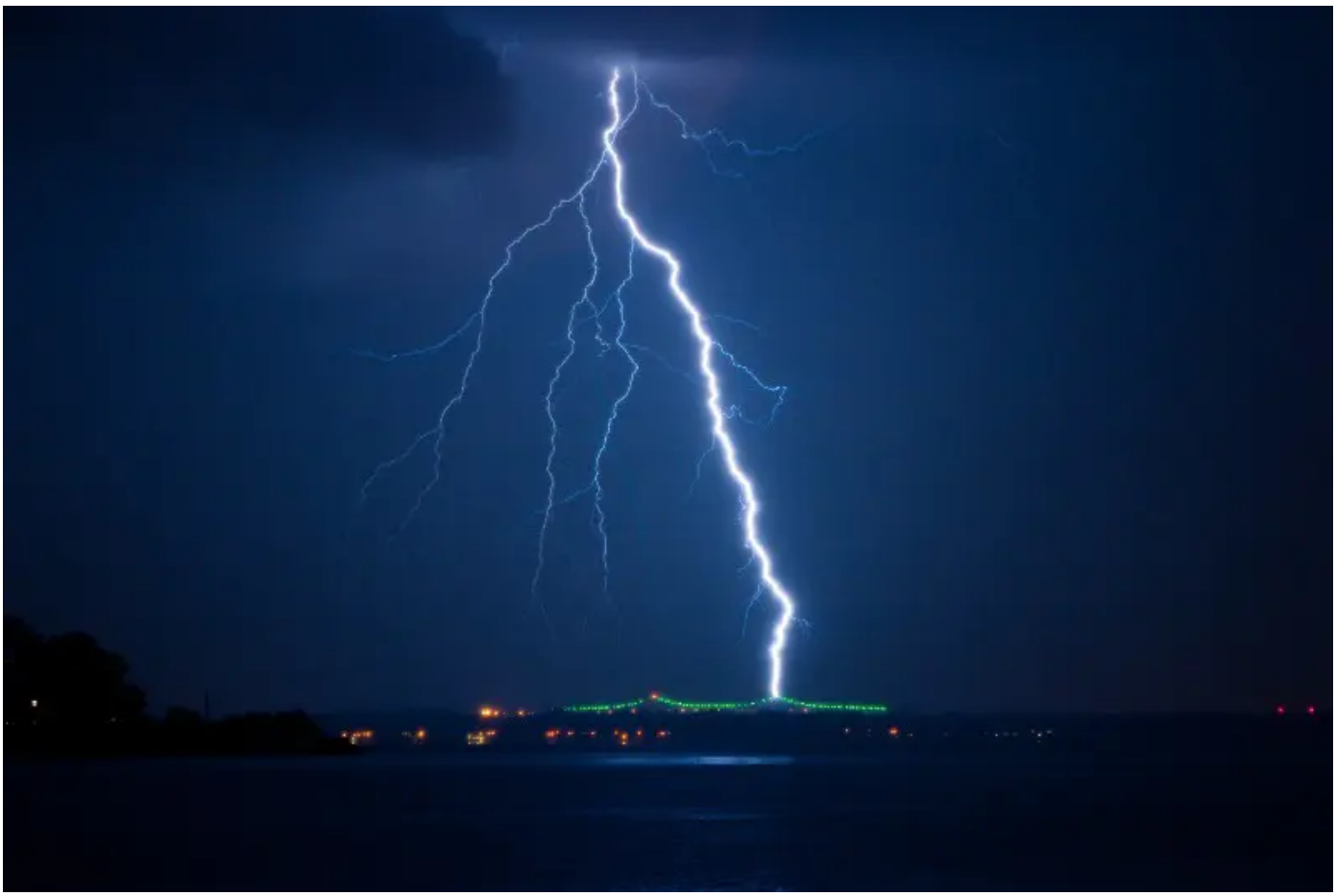
古铁 Lv3
2017年09月22日 阅读 15838

关注

一次性搞定右值，右值引用（&&），和move语义



英文版发表在hackernoon并在我的博客存档。

本文是汉化重制版。

> C++在性能和扩展性上越走越远，结果牺牲了易用性，一版比一版更不易学习。这篇文章主要讨论新版C＋＋的几个相关的知识，右值，右值引用（&&），和move语义，希望能帮助你一次搞定这几个难点。

首先，我们先来看看啥是

右值（r - value）

简单点说，右值就是在等号右边的值。

打上码：

复制代码

```
int var; // too much JavaScript recently:)
var = 8; // OK! l-value (yes, there is a l-value) on the left
8 = var; // ERROR! r-value on the left
(var + 1) = 8; // ERROR! r-value on the left
```

够简单吧。我们看一个更隐晦的情况，函数返回右值。

打上码：

复制代码

```
#include <string>
#include <stdio.h>

int g_var = 8;
int& returnALvalue() {
   return g_var; //here we return a left value
}

int returnARvalue() {
   return g_var; //here we return a r-value
}

int main() {
   printf("%d", returnALvalue()++); // g_var += 1;
   printf("%d", returnARvalue());
}
```

结果：

复制代码

```
8
9
```

注意，我在例子里函数返回左值只是为了做演示，现实生活中请勿模仿。

右值具体有啥用

其实，在右值引用（&&）发明之前，右值就已经可以影响代码逻辑了。

比如这行代码：

复制代码

```
const std::string& name = "rvalue";
```

没有问题，但是下面这行：

复制代码

```
std::string& name = "rvalue"; // use a left reference for a rvalue
```

是编译不过的：

复制代码

```
error: non-const lvalue reference to type 'std::string' (aka 'basic_string<char, char_traits<char>, all
```

说明编译器强制我们用常量引用来指向右值。

再来个更有趣的：

复制代码

```
#include <stdio.h>
#include <string>

void print(const std::string& name) {
    printf("rvalue detected:%s\n", name.c_str());
```

```
void print(std::string& name) {
    printf("lvalue detected:%s\n", name.c_str());
}

int main() {
    std::string name = "lvalue";
    print(name); //compiler can detect the right function for lvalue
    print("rvalue"); // likewise for rvalue
}
```

运行结果：

复制代码

```
lvalue detected:lvalue
rvalue detected:rvalue
```

说明这个差异足以让编译器决定重载函数。

所以右值就是常量咯?

不完全是。这时就轮到 && 出场了。

打上码：

复制代码

```
#include <stdio.h>
#include <string>

void print(const std::string& name) {
  printf("const value detected:%s\n", name.c_str());
}

void print(std::string& name) {
  printf("lvalue detected%s\n", name.c_str());
}

void print(std::string&& name) {
  printf("rvalue detected:%s\n", name.c_str());
}
```

```
    const std::string cname = “cvalue”;

    print(name);
    print(cname);
    print(“rvalue”);
}
```

运行结果：

复制代码

```
lvalue detected:lvalue
const value detected:cvalue
rvalue detected:rvalue
```

说明如果有专门为右值重载的函数的时候，右值的传参会去选择专有函数（接受 && 参数的那个），而不去选更通用的接受常量引用作为参数的函数。所以， && 可以更加细化右值和常量引用。

我总结了函数实参（实际传的那个变量）和形参（括号里声明的那个变量）的适配性，有兴趣的话你也可以通过改上面的验证下：

	Function versions		
	l-value	const value	r-value
l-value	1st choice	2nd choice	not compatible
const value	not compatible	1st choice	not compatible
r-value	not compatible	2nd choice	1st choice

把常量引用细分成常量引用和右值是很好，但是还是没回答具体有啥用。

&&解决了什么问题?

问题是当参数为右值时，不必要的深拷贝。

讲具体点， && 用来区分右值，这样在这个右值 1）是一个构造函数或赋值函数的参数，和2）对应的类包含指针，并指向一个动态分配的资源（内存）时，就可以在函数内避免深拷贝。

用代码的话还可以具体点：

```
#include <algorithm>

using namespace std;

class ResourceOwner {
public:
  ResourceOwner(const char res[]) {
    theResource = new string(res);
  }

  ResourceOwner(const ResourceOwner& other) {
    printf("copy %s\n", other.theResource->c_str());
    theResource = new string(other.theResource->c_str());
  }

  ResourceOwner& operator=(const ResourceOwner& other) {
    ResourceOwner tmp(other);
    swap(theResource, tmp.theResource);
    printf("assign %s\n", other.theResource->c_str());
  }

  ~ResourceOwner() {
    if (theResource) {
      printf("destructor %s\n", theResource->c_str());
      delete theResource;
    }
  }
private:
  string* theResource;
};

void testCopy() {
 // case 1
  printf("=====start testCopy()=====\n");
  ResourceOwner res1("res1");
  ResourceOwner res2 = res1;
  //copy res1
  printf("=====destructors for stack vars, ignore=====\n");
}

void testAssign() {
 // case 2
  printf("=====start testAssign()=====\n");
  ResourceOwner res1("res1");
```

```
 //copy res1, assign res1, destrctor res2
  printf("=====destructors for stack vars, ignore=====\n");
}

void testRValue() {
 // case 3
  printf("=====start testRValue()=====\n");
  ResourceOwner res2("res2");
  res2 = ResourceOwner("res1");
 //copy res1, assign res1, destructor res2, destructor res1
  printf("=====destructors for stack vars, ignore=====\n");
}

int main() {
  testCopy();
  testAssign();
  testRValue();
}
```

运行结果：

复制代码

```
=====start testCopy()=====copy res1=====destructors for stack vars, ignore=====destructor res1destructo
```

前两个例子 `testCopy()` 和 `testAssign()` 里面的结果没问题。这里将 `res1` 里面的的资源拷贝到 `res2` 里是合理的，因为这两个独立的个体都需要有各自的独享资源（string）。

但是在第三个例子就不对了。这次深拷贝的对象res1是个右值（ `ResourceOwner("res1")` 的返回值），其实它马上就要被回收了。所以本身是不需要独享资源的。

我把问题描述再重复一次，这次应该就好理解了：

> `&&` 用来区分右值，这样在这个右值 1）是一个构造函数或赋值函数的参数，和2）对应的类包含指针，并指向一个动态分配的资源（内存）时，就可以在函数内避免深拷贝。

如果深拷贝右值的资源不合理，那什么操作是合理的呢？答案是

继续讨论move语义。解决方法很简单，如果参数是右值，就不拷贝，而是直接"搬"资源。我们先把赋值函数用右值引用重载下：

复制代码

```
ResourceOwner& operator=(ResourceOwner&& other) {
  theResource = other.theResource;
  other.theResource = NULL;
}
```

这个新的赋值函数就叫做**move赋值函数**。**move构造函数**也可以用差不多的办法实现，这里先不赘述了。

如果不太好理解的话，可以这么来：比如你卖了个旧房子搬新家，搬家的时候不一定要把家具都丢掉再买新的对伐（我们在3里面就丢掉了）。你也可以把家具"搬"到新家去。

完美。

那std::move又是啥?

最后我们来解决这个std::move。

我们还是先看看问题：

当1）我们知道一个参数是右值，但是2）编译器不知道的时候，这个参数是调不到move重载函数的。

一个常见的情况是在resource owner上面再加一层类 `ResourceHolder`

复制代码

```
holder
 |
 |----->owner
        |
        |----->resource
```

注意，在下面的代码里，我把**move构造函数**也加上了。

打上码：

复制代码

```
#include <string>
#include <algorithm>


using namespace std;

class ResourceOwner {
public:
  ResourceOwner(const char res[]) {
    theResource = new string(res);
  }


  ResourceOwner(const ResourceOwner& other) {
    printf(“copy %s\n”, other.theResource->c_str());
    theResource = new string(other.theResource->c_str());
  }

++ResourceOwner(ResourceOwner&& other) {
++ printf(“move cons %s\n”, other.theResource->c_str());
++ theResource = other.theResource;
++ other.theResource = NULL;
++}


  ResourceOwner& operator=(const ResourceOwner& other) {
    ResourceOwner tmp(other);
    swap(theResource, tmp.theResource);
    printf(“assign %s\n”, other.theResource->c_str());
  }


++ResourceOwner& operator=(ResourceOwner&& other) {
++ printf(“move assign %s\n”, other.theResource->c_str());
++ theResource = other.theResource;
++ other.theResource = NULL;
++}

  ~ResourceOwner() {
    if (theResource) {
      printf(“destructor %s\n”, theResource->c_str());
      delete theResource;
    }
```

```
  private:
    string* theResource;
  };


  class ResourceHolder {
  ……
  ResourceHolder& operator=(ResourceHolder&& other) {
    printf("move assign %s\n", other.theResource->c_str());
    resOwner = other.resOwner;
  }
  ……
  private:
    ResourceOwner resOwner;
  }
```

在 `ResourceHolder` 的**move赋值函数**中，其实我们想调用的是的**move赋值函数**，因为右值的成员也是右值。但是

```
resOwner = other.resOwner
```

其实是调用了普通赋值函数，还是做了深拷贝。

那再重复一次问题，看看是不是好理解了：

> 当1）我们知道一个参数是右值，但是2）编译器不知道的时候，这个参数是调不到move重载函数的。

解决方法是，我们可以用 `std::move` 把这个变量强制转化成右值，就能调用到正确的重载函数了。

复制代码

```
ResourceHolder& operator=(ResourceHolder&& other) {
  printf("move assign %s\n", other.theResource->c_str());
  resOwner = std::move(other.resOwner);
}
```

还能再深入一点吗？

我们都知道强转除了让编译器闭嘴，其实是会生成对应的机器码的。（在不开O的情况下比较容易观察到）这些机器码会把变量在不同大小的寄存器里面移来移去来真正完成强转操作。

所以 `std::move` 也和强转做了类似的操作吗？我也不知道，一起来试试看。

首先，我们把main函数改一改，（我尽量保持逻辑一致）

打上码：

复制代码

```
int main() {
  ResourceOwner res(“res1”);
  asm(“nop”); // remeber me
  ResourceOwner && rvalue = std::move(res);
  asm(“nop”); // remeber me
}
```

编译它，然后用下面的命令把汇编语言打出来

复制代码

```
clang++ -g -c -std=c++11 -stdlib=libc++ -Weverything move.cc
gobjdump -d -D move.o
```

😯，原来藏在下面的画风是这样的：

复制代码

```
0000000000000000 <_main>:
 0: 55 push %rbp
 1: 48 89 e5 mov %rsp,%rbp
 4: 48 83 ec 20 sub $0x20,%rsp
 8: 48 8d 7d f0 lea -0x10(%rbp),%rdi
 c: 48 8d 35 41 03 00 00 lea 0x341(%rip),%rsi # 354
 <GCC_except_table5+0x18>
 13: e8 00 00 00 00 callq
 18 <_main+0x18> 18: 90 nop // remember me
 19: 48 8d 75 f0 lea -0x10(%rbp),%rsi
 1d: 48 89 75 f8 mov %rsi,-0x8(%rbp)
 21: 48 8b 75 f8 mov -0x8(%rbp),%rsi
 25: 48 89 75 e8 mov %rsi,-0x18(%rbp)
 29: 90 nop // remember me
 2a: 48 8d 7d f0 lea -0x10(%rbp),%rdi
```

```
39: 5d pop %rbp
3a: c3 retq
3b: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)
```

我也看不太懂，还好用 nop 做了染色。看两个 nop 中间那段确实生成了一些机器码，但是这些机器码貌似啥都没做，只是简单的把一个变量的地址赋值给另一个而已。并且，如果我们把O（ - O1就够了）打开，所有的nop中间的机器码就都被干掉了。

复制代码

```
clang++ -g -c -O1 -std=c++11 -stdlib=libc++ -Weverything move.cc
gobjdump -d -D move.o
```

再来，如果把关键行改成

复制代码

```
ResourceOwner & rvalue = res;
```

除了变量的相对偏移有变化，其实生成的机器码是一样一样的。

说明了在这里 std::move 其实是个纯语法糖，而并没有啥实际的操作。

好了，今天先写到这里。如果你喜欢本篇，欢迎点赞和关注。有兴趣也可以去Medium上随意啪啪啪我的其他文章。感谢阅读。

文章分类 后端 文章标签 C++ 编译器 汇编语言

输入评论...

ytlw Golang Developer @ ...

重载等号那里，漏了return *this

3月前 1 回复

用户3186085938441

源码跑不过呀 😓

5月前 回复

郝姬友

说明编译器强制我们用常量引用来指向右值。
--这句说错了吧？应该是编译器强制我们用非常量引用来指向右值

1年前 回复

郝姬友

mark下

1年前 回复

Lysander

theResource 这个东西复制到visual studio 里面说unresolved reference啊。。
能不能把源代码搞得能编译啊

1年前 回复

相关推荐

进击的大葱 · 3小时前 · 编译器

实现JavaScript语言解释器（二）

2

吴尼玛 · 1天前 · C++

Linux命令行参数解析——getopt_long

Lucy_FinTech社区 · 2天前 · C++

手把手教你用C++搭建操作系统

青玉白露 · 3天前 · C++

C/C++杂谈(1)指针常量、常量指针

3

doodlewind · 1月前 · JavaScript / C++

写给前端的手动内存管理基础入门（一）返璞归真：从引用类型到裸指针

137 16

彭丑丑 · 13天前 · C++

NDK | C++ 复习笔记

8 2

God23Bin · 2天前 · C++

C语言-字符串函数的实现（一）之strlen

前端小菜鸟001 · 3月前 · 编译器

代码编译初探 （中）- 前端必看

13 3

vivo互联网技术 · 1月前 · Android / C++

Android NativeCrash 捕获与解析

42 5

理查德森 · 12天前 · 汇编语言

汇编002-函数本质(上)

4