C++右值引用（std::move）

niediao
C++

前段时间和朋友聊天的过程中说到了右值和std::move以及项目代码里面不合时宜的使用std::move。以及前段时间，华为开源了方舟编译器，大神们讨论方舟编译器，有人贴出里面乱用std::move，看了一下它的源码，是有随意使用std::move的现象。因此，对右值引用和std::move做了一下回顾。

说到**右值**，先看一下什么是右值，在c++中，一个值要么是右值，要么是左值，*左值是指表达式结束后依然存在的持久化对象，右值是指表达式结束时就不再存在的临时对象。所有的具名变量或者对象都是左值，而右值不具名。*

比如：

常见的右值："abc",123等都是右值。

右值引用，用以引用一个右值，可以延长右值的生命期，比如：

```
int&& i = 123;
int&& j = std::move(i);
int&& k = i;//编译不过，这里i是一个左值，右值引用只能引用右值
```

可以通过下面的代码，更深入的体会左值引用和右值引用的区别：

```
int i;
int&& j = i++;
int&& k = ++i;
int& m = i++;
int& l = ++i;

move.cpp: In function 'int main()':
move.cpp:72:14: error: cannot bind 'int' lvalue to 'int&&'
  int&& k = ++i;
              ^
move.cpp:73:15: error: invalid initialization of non-const reference of type 'int
     int& m = i++;
```

为什么需要右值引用

C++引入右值引用之后，可以通过右值引用，充分使用临时变量，或者即将不使用的变量即右值的资源，减少不必要的拷贝，提高效率。如下代码，均会产生临时变量：

```
class RValue {
};

RValue get() {
    return RValue();
}

void put(RValue){}
```

为了充分利用右值的资源，减少不必要的拷贝，C++11引入了右值引用(&&)，移动构造函数，移动复制运算符以及std::move。

右值引用(&&)，移动构造函数，移动复制运算符以及std::move

将上面的类定义补充完整：

```
ut<<#include <iostream>
#include <memory>
#include <vector>
#include <string>
using namespace std;

struct RValue {
        RValue():sources("hello!!!"){}
        RValue(RValue&& a) {
                sources = std::move(a.sources);
                cout<<"&& RValue"<<endl;
        }

        RValue(const RValue& a) {
                sources = a.sources;
                cout<<"& RValue"<<endl;
        }

        void operator=(const RValue&& a) {
                sources = std::move(a.sources);
                cout<<"&& =="<<endl;
        }

        void operator=(const RValue& a) {
                sources = a.sources;
                cout<<"& =="<<endl;
        }

        string sources;;
};

RValue get() {
        RValue a;
        return a;
}

void put(RValue){}

int main() {
        RValue a = get();
        cout<<"---------------"<<endl;
        put(RValue());
        return 0;
}
```

不过，当运行的时候却发现没有任何输出

```
g++ move.cpp -std=c++11 -o move
 ./move
```

这是因为，编译器做了优化，编译的时候加上-fno-elide-constructors，去掉优化

```
g++ move.cpp -std=c++11 -fno-elide-constructors -o move
 ./move
&& RValue
&& RValue
---------------
&& RValue
```

通过上面的代码，可以看出，在没有加-fno-elide-constructors选项时，编译器做了优化，没有临时变量的生成。在加了-fno-elide-constructors选项时，get产生了两次临时变量，二put生成了一次临时变量。

将get函数稍微修改一下：

```
RValue get() {
        RValue a;
        return std::move(RValue());
}

g++ move.cpp -std=c++11 -o move
 ./move
&& RValue
---------------

//加编译选项
g++ move.cpp -std=c++11 -fno-elide-constructors -o move
 ./move
&& RValue
&& RValue
---------------
&& RValue
```

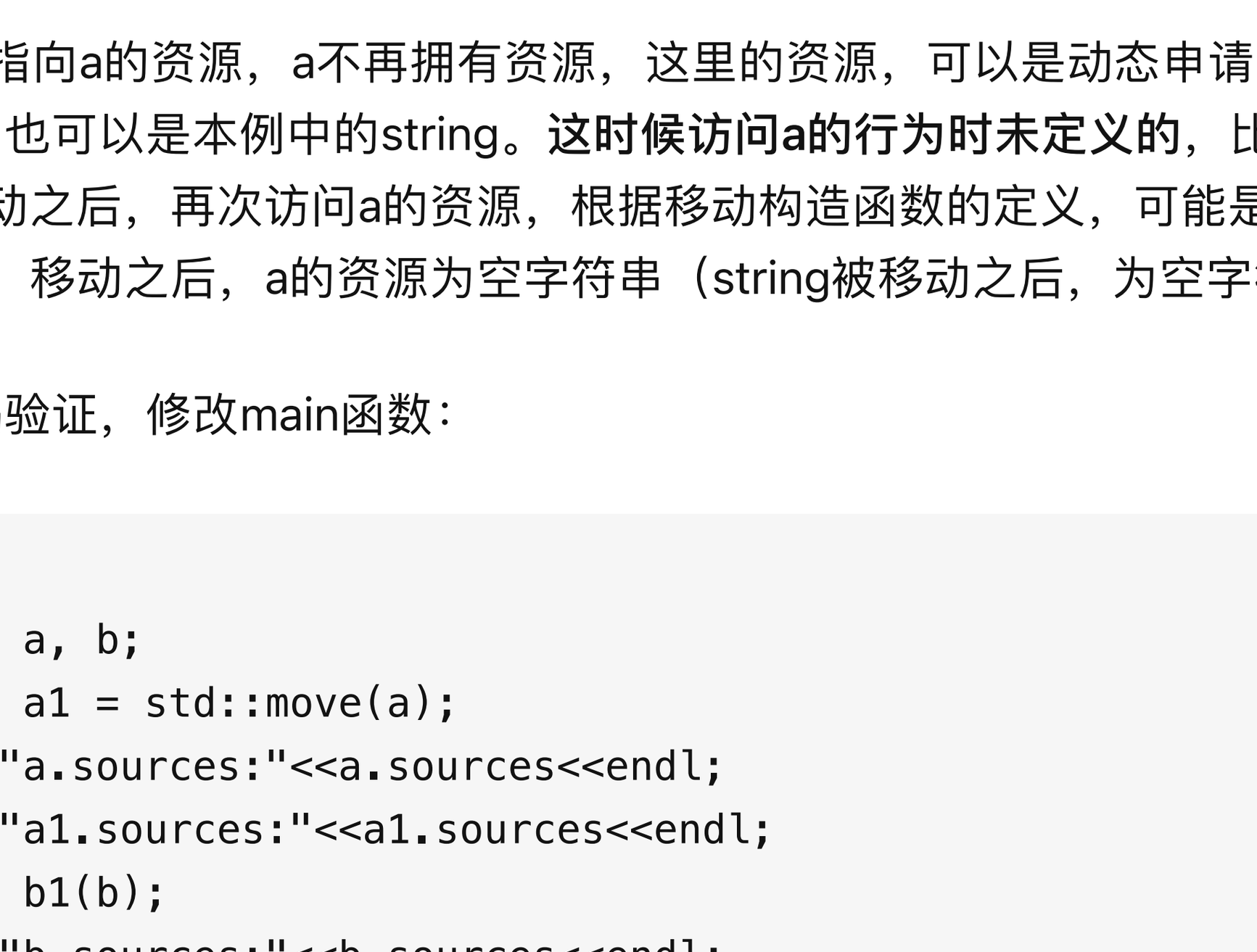
只是简单的修改了一下，std::move(a)，在编译器做了优化的情况下，用了std::move，反而多做了一次拷贝。

其实，RValue如果在没有定义移动构造函数，重复上面的操作，生成临时变量的次数还是一样的，只不过，调用的时拷贝构造函数了而已。

通过get函数可以知道，乱用std::move在编译器开启构造函数优化的场景下反而增加了不必要的拷贝。那么，std::move应该在什么场景下使用？

std::move使用场景

1、移动构造函数的原理



通过移动构造，b指向a的资源，a不再拥有资源，这里的资源，可以是动态申请的内存，网络链接，打开的文件，也可以是本例中的string。**这时候访问a的行为时未定义的**，比如，如果资源是动态内存，a被移动之后，再次访问a的资源，根据移动构造函数的定义，可能是空指针，如果是资源上文的string，移动之后，a的资源为空字符串（string被移动之后，为空字符串）。

可以通过下面代码验证，修改main函数：

```
int main() {
        RValue a, b;
        RValue a1 = std::move(a);
        cout<<"a.sources:"<<a.sources<<endl;
        cout<<"a1.sources:"<<a1.sources<<endl;
        RValue b1(b);
        cout<<"b.sources:"<<b.sources<<endl;
        cout<<"b1.sources:"<<a1.sources<<endl;
        return 0;
}

g++ move.cpp -std=c++11 -o move
 ./move
&& RValue
a.sources:
a1.sources:hello!!!
& RValue
b.sources:hello!!!
b1.sources:hello!!!
```

通过移动构造函数之后，a的资源为空，b指向了a的资源。通过拷贝构造函数，b**复制**了a的资源。

2、std::move的原理

std::move的定义：

```
template<typename _Tp>
  constexpr typename std::remove_reference<_Tp>::type&&
  move(_Tp&& __t) noexcept
  { return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }
```

这里，T&&是通用引用，需要注意和右值引用（比如int&&）区分。通过move定义可以看出，move并没有"移动"什么内容，**只是将传入的值转换为右值**，此外没有其他动作。std::move+移动构造函数或者移动赋值运算符，才能充分起到减少不必要拷贝的意义。

3、std::move的使用场景

在之前的项目中看到有的同事到处使用std::move，好像觉得使用了std::move就能移动资源，提升性能一样，在我看来，std::move主要使用在以下场景：

- 使用前提：1 定义的类使用了资源并定义了移动构造函数和移动赋值运算符，2 该变量即将不再使用
- 使用场景

```
RValue a, b;

//对a,b坐一系列操作之后，不再使用a,b，但需要保存到智能指针或者容器之中
unique_ptr<RValue> up(new RValue(std::move(a)));
vector<RValue*> vr;
vr.push_back(new RValue(std::move(b)));

//临时容器中保存的大量的元素需要复制到目标容器之中
vector<RValue> vrs_temp;
vrs_temp.push_back(RValue());
vrs_temp.push_back(RValue());
vrs_temp.push_back(RValue());
vector<RValue> vrs(std::move(vrs_temp));

RValue c;
put(std::move(c));
```

- 在没有右值引用之前，为了使用临时变量，通常定义const的左值引用，比如const string&，在有了右值引用之后，为了使用右值语义，不要把参数定义为常量左值引用，否则，传递右值时调用的时拷贝构造函数

```
void put(const RValue& c){
        cout<<"----------"<<endl;
        unique_ptr<RValue> up(new RValue(std::move(c)));
        cout<<"----------"<<endl;
}

RValue c;
put(std::move(c));

g++ move.cpp -std=c++11 -o move
 ./move
----------
& RValue
----------
```

不使用左值常量引用：

```
void put(RValue c){
        cout<<"----------"<<endl;
        unique_ptr<RValue> up(new RValue(std::move(c)));
        cout<<"----------"<<endl;
}

RValue c;
put(std::move(c));

g++ move.cpp -std=c++11 -o move
 ./move
&& RValue
----------
&& RValue
----------
```

这是因为，根据通用引用的定义，std::move(c)过程中，模板参数被推倒为const RValue&，因此，调用拷贝构造函数。

总结

通过简绍右值和右值引用以及std::move和移动构造函数，总结右值引用，移动构造函数和移动赋值运算符和std::move的用法和注意事项。

编辑于 2019-12-02 08:39