本次主要讲c++11中的右值引用，后面还会讲到右值引用如何结合std::move优化我们的程序。

c++11增加了一个新的类型，称作右值引用(R-value reference)，标记为T &&，说到右值引用类型之前先要了解什么是左值和右值。  
左值具名，对应指定内存域，可访问；右值不具名，不对应内存域，不可访问。临时对象是右值。左值可处于等号左边，右值只能放在等号右边。区分表达式的左右值属性有一个简便方法：若可对表达式用 & 符取址，则为左值，否则为右值。  
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，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。

int && a = 1; //&&为右值引用

**&&的特性**

　　实际上T&&并不是一定表示右值引用，它的引用类型是未定的，即可能是左值有可能是右值。看看这个例子：

[复制代码](javascript:void(0);)

template<typename T>

void f(T&& param);

f(10); //10是右值

int x = 10;

f(x); //x是左值

[复制代码](javascript:void(0);)

　　从这个例子可以看出，param有时是左值引用，有时是右值引用，它在上面的例子中&&实际上是一个未定的引用类型。这个未定的引用类型被scott meyers称为universal references（可以认为它是种通用的引用类型），它必须被初始化，它是左值应用还是右值引用取决于它的初始化，如果&&被一个左值初始化的话，它就是一个左值引用；如果它被一个右值初始化的话，它就是一个右值引用。

**&&为universal references时的唯一条件是有类型推断发生。**

[复制代码](javascript:void(0);)

template<typename T>

void f(T&& param); //这里T的类型需要推导，所以&&是一个universal references

template<typename T>

class Test {

...

Test(Test&& rhs); // 已经定义了一个特定的类型， 没有类型推断

... // && 是一个右值引用

};

void f(Test&& param); // 已经定义了一个确定的类型， 没有类型推断,&& 是一个右值引用

[复制代码](javascript:void(0);)

再看一个复杂一点的例子

template<typename T>

void f(std::vector<T>&& param);

这里既有推断类型T又有确定类型vector，那么这个param到底是什么类型呢？  
它是右值引用类型，因为在调用这个函数之前，这个vector<T>中的推断类型已经确定了，所以到调用f时没有类型推断了。

再看看这个例子：

template<typename T>

void f(const T&& param);

这个param是universal references吗？错，它是右值引用类型，也许会迷糊，T不是推断类型吗，怎么会是右值引用类型。其实还有一条规则：**universal references仅仅在T&&下发生，任何一点附加条件都会使之失效，而变成一个右值引用。**

**引用折叠（Reference collapsing）规则：**

1. 所有的右值引用叠加到右值引用上变成一个右值引用
2. 所有的其它引用类型叠加都变成一个左值引用
3. 左值或者右值是独立于它的类型的，也就是说一个右值引用类型的左值是合法的。

int&& var1 = x; // var1 is of type int&& (no use of auto here)

auto&& var2 = var1; // var2 is of type int& ，var2的类型是universal references(有类型推导)

var1的类型是一个左值类型，但var1本身是一个左值；  
var1是一个左值，根据引用折叠规则，var2是一个int&

int w1, w2;

auto&& v1 = w1;

decltype(w1)&& v2 = w2;

v1是一个universal reference，它被一个左值初始化，所以它最终一个左值；  
v2是一个右值引用类型，但它被一个左值初始化，一个左值初始化一个右值引用类型是不合法的，所以会编译报错。但是如果我希望把一个左值赋给一个右值引用类型该怎么做呢 ，用std::move，decltype(w1)&& v2 = std::move(w2); std::move可以将一个左值转换成右值，关于std::move将在下一篇博文中介绍。

**&&的总结：**

1. 左值和右值是独立于它们的类型的，一个左值的类型有可能是右值引用类型。
2. T&&是一个未定的引用类型，它可能是左值引用也可能是右值引用类型，取决于初始化的值类型。
3. &&成为未定的引用类型的唯一条件是：T&&且发生类型推断。
4. 所有的右值引用叠加到右值引用上变成一个右值引用，其它引用折叠都为左值引用。

如果想更详细了解&&，可以参考scott-meyers这个文章：<http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers>

**右值引用优化性能，避免深拷贝**

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。消除了临时对象的维护 ( 创建和销毁 ) 对性能的影响。

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。

[复制代码](javascript:void(0);)

class MyString {

private:

char\* m\_data;

size\_t m\_len;

void copy\_data(const char \*s) {

m\_data = new char[m\_len+1];

memcpy(m\_data, s, m\_len);

m\_data[m\_len] = '\0';

}

public:

MyString() {

m\_data = NULL;

m\_len = 0;

}

MyString(const char\* p) {

m\_len = strlen (p);

copy\_data(p);

}

MyString(const MyString& str) {

m\_len = str.m\_len;

copy\_data(str.m\_data);

std::cout << "Copy Constructor is called! source: " << str.m\_data << std::endl;

}

MyString& operator=(const MyString& str) {

if (this != &str) {

m\_len = str.m\_len;

copy\_data(str.m\_data);

}

std::cout << "Copy Assignment is called! source: " << str.m\_data << std::endl;

return \*this;

}

virtual ~MyString() {

if (m\_data) free(m\_data);

}

};

void test() {

MyString a;

a = MyString("Hello");

std::vector<MyString> vec;

vec.push\_back(MyString("World"));

}

[复制代码](javascript:void(0);)

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。

用c++11的右值引用来定义这两个函数

[复制代码](javascript:void(0);)

MyString(MyString&& str) {

std::cout << "Move Constructor is called! source: " << str.\_data << std::endl;

\_len = str.\_len;

\_data = str.\_data; //避免了不必要的拷贝

str.\_len = 0;

str.\_data = NULL;

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

MyString& operator=(MyString&& str) {

std::cout << "Move Assignment is called! source: " << str.\_data << std::endl;

if (this != &str) {

\_len = str.\_len;

\_data = str.\_data; //避免了不必要的拷贝

str.\_len = 0;

str.\_data = NULL;

}

return \*this;

}

[复制代码](javascript:void(0);)

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，应该设计右值引用的拷贝构造函数和赋值函数，以提高应用程序的效率。

c++11 boost技术交流群：296561497，欢迎大家来交流技术。

一点梦想：尽自己一份力，让c++的世界变得更美好！

分类: [C++](https://www.cnblogs.com/qicosmos/category/466797.html),[c++11 使用c++11改进我们的程序系列](https://www.cnblogs.com/qicosmos/category/490693.html)