A.1 右值引用

如果你从事过 C++ 编程，你会对引用比较熟悉， C++ 的引用允许你为已经存在的对象创建一个新的名字。对新引用所做的访问和修改操作，都会影响它的原型。

例如：

```
int var=42;
int& ref=var;  // 创建一个var的引用
ref=99;
assert(var==99);  // 原型的值被改变了，因为引用被赋值了
```

目前为止，我们用过的所有引用都是左值引用——对左值的引用。lvalue这个词来自于C语言，指的是可以放在赋值表达式左边的事物——在栈上或堆上分配的命名对象，或者其他对象成员——有明确的内存地址。rvalue这个词也来源于C语言，指的是可以出现在赋值表达式右侧的对象——例如，文字常量和临时变量。因此，左值引用只能被绑定在左值上，而不是右值。

不能这样写：

```
int& i=42;  // 编译失败
```

例如，因为42是一个右值。好吧，这有些假；你可能通常使用下面的方式讲一个右值绑定到一个const左值引用上：

```
int const& i = 42;
```

这算是钻了标准的一个空子吧。不过，这种情况我们之前也介绍过，我们通过对左值的const引用创建临时性对象，作为参数传递给函数。

其允许隐式转换，所以你可这样写：

```
void print(std::string const& s);
print("hello");  //创建了临时std::string对象
```

C++11标准介绍了*右值引用*(rvalue reference)，这种方式只能绑定右值，不能绑定左值，其通过两个 `&&` 来进行声明：

```
int&& i=42;
int j=42;
int&& k=j;  // 编译失败
```

因此，可以使用函数重载的方式来确定：函数有左值或右值为参数的时候，看是否能被同名且对应参数为左值或有值引用的函数所重载。

其基础就是C++11新添语义——*移动语义*(move semantics)。

A.1.1 移动语义

右值通常都是临时的，所以可以随意修改；如果知道函数的某个参数是一个右值，就可以将其看作为一个临时存储或“窃取”内容，也不影响程序的正确性。这就意味着，比起拷贝右值参数的内容，不如移动其内容。动态数组比较大的时候，这样能节省很多内存分配，提供更多的优化空间。试想，一个函数以 `std::vector<int>` 作为一个参数，就需要将其拷贝进来，而不对原始的数据做任何操作。 C++ 03/98的办法是，将这个参数作为一个左值的const引用传入，然后做内部拷贝：

```
void process_copy(std::vector<int> const& vec_)
{
  std::vector<int> vec(vec_);
  vec.push_back(42);
}
```

这就允许函数能以左值或右值的形式进行传递，不过任何情况下都是通过拷贝来完成的。如果使用右值引用版本的函数来重载这个函数，就能避免在传入右值的时候，函数会进行内部拷贝的过程，因为可以任意的对原始值进行修改：

```
void process_copy(std::vector<int> && vec)
{
  vec.push_back(42);
}
```

如果这个问题存在于类的构造函数中，窃取内部右值在新的实例中使用。可以参考一下清单中的例子(默认构造函数会分配很大一块内存，在析构函数中释放)。

清单A.1 使用移动构造函数的类

```
class X
{
private:
  int* data;

public:
  X():
    data(new int[1000000])
  {}

  ~X()
  {
    delete [] data;
  }

  X(const X& other):  // 1
   data(new int[1000000])
  {
    std::copy(other.data,other.data+1000000,data);
  }

  X(X&& other):  // 2
    data(other.data)
  {
    other.data=nullptr;
  }
};
```

一般情况下，拷贝构造函数①都是这么定义：分配一块新内存，然后将数据拷贝进去。不过，现在有了一个新的构造函数，可以接受右值引用来获取老数据②，就是移动构造函数。在这个例子中，只是将指针拷贝到数据中，将other以空指针的形式留在了新实例中；使用右值里创建变量，就能避免了空间和时间上的多余消耗。

X类(清单A.1)中的移动构造函数，仅作为一次优化；在其他例子中，有些类型的构造函数只支持移动构造函数，而不支持拷贝构造函数。例如，智能指针 `std::unique_ptr<>` 的非空实例中，只允许这个指针指向其对象，所以拷贝函数在这里就不能用了(如果使用拷贝函数，就会有两个 `std::unique_ptr<>` 指向该对象，不满足 `std::unique_ptr<>` 定义)。不过，移动构造函数允许

对指针的所有权，在实例之间进行传递，并且允许 `std::unique_ptr<>` 像一个带有返回值的函数一样使用——指针的转移是通过移动，而非拷贝。

如果你已经知道，某个变量在之后就不会在用到了，这时候可以选择显式的移动，你可以使用 `static_cast<X&&>` 将对应变量转换为右值，或者通过调用 `std::move()` 函数来做这件事：

```
X x1;
X x2=std::move(x1);
X x3=static_cast<X&&>(x2);
```

想要将参数值不通过拷贝，转化为本地变量或成员变量时，就可以使用这个办法；虽然右值引用参数绑定了右值，不过在函数内部，会当做左值来进行处理：

```
void do_stuff(X&& x_)
{
  X a(x_);  // 拷贝
  X b(std::move(x_));  // 移动
}
do_stuff(X());  // ok，右值绑定到右值引用上
X x;
do_stuff(x);  // 错误，左值不能绑定到右值引用上
```

移动语义在线程库中用的比较广泛，无拷贝操作对数据进行转移可以作为一种优化方式，避免对将要被销毁的变量进行额外的拷贝。在2.2节中看到，在线程中使用 `std::move()` 转移 `std::unique_ptr<>` 得到一个新实例；在2.3节中，了解了在 `std:thread` 的实例间使用移动语义，用来转移线程的所有权。

`std::thread` 、 `std::unique_lock<>` 、 `std::future<>` 、 `std::promise<>` 和 `std::packaged_task<>` 都不能拷贝，不过这些类都有移动构造函数，能让相关资源在实例中进行传递，并且支持用一个函数将值进行返回。 `std::string` 和 `std::vector<>` 也可以拷贝，不过它们也有移动构造函数和移动赋值操作符，就是为了避免拷贝拷贝大量数据。

C++标准库不会将一个对象显式的转移到另一个对象中，除非将其销毁的时候或对其赋值的时候(拷贝和移动的操作很相似)。不过，实践中移动能保证类中的所有状态保持不变，表现良好。一个 `std::thread` 实例可以作为移动源，转移到新(以默认构造方式)的 `std::thread` 实例中。还有，`std::string` 可以通过移动原始数据进行构造，并且保留原始数据的状态，不过不能保证的是原始数据中该状态是否正确(根据字符串长度或字符数量决定)。

A.1.2 右值引用和函数模板

在使用右值引用作为函数模板的参数时，与之前的用法有些不同：如果函数模板参数以右值引用作为一个模板参数，当对应位置提供左值的时候，模板会自动将其类型认定为左值引用；当提供右值的时候，会当做普通数据使用。可能有些口语化，来看几个例子吧。

考虑一下下面的函数模板：

```
template<typename T>
void foo(T&& t)
{}
```

随后传入一个右值，T的类型将被推导为：

```
foo(42);  // foo<int>(42)
foo(3.14159);  // foo<double><3.14159>
foo(std::string());  // foo<std::string>(std::string())
```

不过，向foo传入左值的时候，T会被推导为一个左值引用：

```
int i = 42;
foo(i);  // foo<int&>(i)
```

因为函数参数声明为 `T&&` ，所以就是引用的引用，可以视为是原始的引用类型。那么foo()就相当于：

```
foo<int&>(); // void foo<int&>(int& t);
```

这就允许一个函数模板可以即接受左值，又可以接受右值参数；这种方式已经被 `std::thread` 的构造函数所使用(2.1节和2.2节)，所以能够将可调用对象移动到内部存储，而非当参数是右值的时候进行拷贝。