C++11尝鲜：右值引用和转发型引用

右值引用

为了解决移动语义及完美转发问题，C++11标准引入了右值引用（rvalue reference）这一重要的新概念。右值引用采用T&&这一语法形式，比传统的引用T&（如今被称作左值引用 lvalue reference）多一个&。
如果把经由T&&这一语法形式所产生的引用类型都叫做右值引用，那么这种广义的右值引用又可分为以下三种类型：

- 无名右值引用
- 具名右值引用
- 转发型引用

无名右值引用和具名右值引用的引入主要是为了解决移动语义问题。
转发型引用的引入主要是为了解决完美转发问题。

无名右值引用

无名右值引用（unnamed rvalue reference）是指由右值引用相关操作所产生的引用类型。
无名右值引用主要通过返回右值引用的类型转换操作产生， 其语法形式如下：
static_cast<T&&>(t)
标准规定该语法形式将把表达式 t 转换为T类型的无名右值引用。
无名右值引用是右值，标准规定无名右值引用和传统的右值一样具有潜在的可移动性，即它所占有的资源可以被移动（窃取）。

std::move()

由于无名右值引用是右值，借助于类型转换操作产生无名右值引用这一手段，左值表达式就可以被转换成右值表达式。为了便于利用这一重要的转换操作，标准库为我们提供了封装这一操作的函数，这就是std::move()。
假设左值表达式 t 的类型为T&，利用以下函数调用就可以把左值表达式 t 转换为T类型的无名右值引用（右值，类型为T&&）。
std::move(t)

具名右值引用

如果某个变量或参数被声明为T&&类型，并且T无需推导即可确定，那么这个变量或参数就是一个具名右值引用（named rvalue reference）。
具名右值引用是左值，因为具名右值引用有名字，和传统的左值引用一样可以用操作符&取地址。
与广义的右值引用相对应，狭义的右值引用仅限指具名右值引用。
传统的左值引用可以绑定左值，在某些情况下也可绑定右值。与此不同的是，右值引用只能绑定右值。
右值引用和左值引用统称为引用（reference），它们具有引用的共性，比如都必须在初始化时绑定值，都是左值等等。

```
struct X {};
X a;
X&& b = static_cast<X&&>(a);
X&& c = std::move(a);
//static_cast<X&&>(a) 和 std::move(a) 是无名右值引用，是右值
//b 和 c 是具名右值引用，是左值
X& d = a;
X& e = b;
const X& f = c;
const X& g = X();
X&& h = X();
//左值引用a和e只能绑定左值（包括传统左值：变量a以及新型左值：右值引用b）
//const左值引用f和g可以绑定左值（右值引用c），也可以绑定右值（临时对象X()）
//右值引用b，c和h只能绑定右值（包括新型右值：无名右值引用std::move(a)以及传统右值：临时对象X()）
```

左右值重载策略

有时我们需要在函数中区分参数的左右值属性，根据参数左右值属性的不同做出不同的处理。适当地采用左右值重载策略，借助于左右值引用参数不同的绑定特性，我们可以利用函数重载来做到这一点。常见的左右值重载策略如下：

```
struct X {};
//左值版本
void f(const X& param1){/*处理左值参数param1*/}
//右值版本
void f(X&& param2){/*处理右值参数param2*/}

X a;
f(a);            //调用左值版本
f(X());          //调用右值版本
f(std::move(a)); //调用右值版本
```

即在函数重载中分别重载const左值引用和右值引用。
重载const左值引用的为左值版本，这是因为const左值引用参数能绑定左值，而右值引用参数不能绑定左值。
重载右值引用的为右值版本，这是因为虽然const左值引用参数和右值引用参数都能绑定右值，但标准规定右值引用参数的绑定优先度要高于const左值引用参数。

移动构造器和移动赋值运算符

在类的构造器和赋值运算符中运用上述左右值重载策略，就会产生两个新的特殊成员函数：移动构造器（move constructor）和移动赋值运算符（move assignment operator）。

```
struct X
{
    X();                            //缺省构造器
    X(const X& that);               //拷贝构造器
    X(X&& that);                    //移动构造器
    X& operator=(const X& that); //拷贝赋值运算符
    X& operator=(X&& that);      //移动赋值运算符
};

X a;                            //调用缺省构造器
X b = a;                        //调用拷贝构造器
X c = std::move(b);             //调用移动构造器
b = a;                          //调用拷贝赋值运算符
c = std::move(b);               //调用移动赋值运算符
```

移动语义

无名右值引用和具名右值引用的引入主要是为了解决移动语义问题。
移动语义问题是指在某些特定情况下（比如用右值来赋值或构造对象时）如何采用廉价的移动语义替换昂贵的拷贝语义的问题。
移动语义（move semantics）是指某个对象接管另一个对象所拥有的外部资源的所有权。移动语义需要通过移动（窃取）其他对象所拥有的资源来完成。移动语义的具体实现（即一次that对象到this对象的移动（move））通常包含以下若干步骤：

- 如果this对象自身也拥有资源，释放该资源
- 将this对象的指针或句柄指向that对象所拥有的资源
- 将that对象原本指向该资源的指针或句柄设为空值

上述步骤可简单概括为①释放this（this非空时）②移动that
移动语义通常在移动构造器和移动赋值运算符中得以具体实现。两者的区别在于移动构造对象时this对象为空因而①释放this无须进行。

与移动语义相对，传统的拷贝语义（copy semantics）是指某个对象拷贝（复制）另一个对象所拥有的外部资源并获得新生资源的所有权。拷贝语义的具体实现（即一次that对象到this对象的拷贝（copy））通常包含以下若干步骤：

- 如果this对象自身也拥有资源，释放该资源
- 拷贝（复制）that对象所拥有的资源
- 将this对象的指针或句柄指向新生的资源
- 如果that对象为临时对象（右值），那么拷贝完成之后that对象所拥有的资源将会因that对象被销毁而即刻得以释放

上述步骤可简单概括为①释放this（this非空时）②拷贝that③释放that（that为右值时）
拷贝语义通常在拷贝构造器和拷贝赋值运算符中得以具体实现。两者的区别在于拷贝构造对象时this对象为空因而①释放this无须进行。

比较移动语义与拷贝语义的具体步骤可知，在赋值或构造对象时，

- 如果源对象that为左值，由于两者效果不同（移动that ≠ 拷贝that），此时移动语义不能用来替换拷贝语义。
- 如果源对象that为右值，由于两者效果相同（移动that = 拷贝that + 释放that），此时廉价的移动语义（通过指针操作来移动资源）便可以用来替换昂贵的拷贝语义（生成，拷贝然后释放资源）。

由此可知，只要在进行相关操作（比如赋值或构造）时，采取适当的左右值重载策略区分源对象的左右值属性，根据其左右值属性分别采用拷贝语义和移动语义，移动语义问题便可以得到解决。

下面用MemoryBlock这个自我管理内存块的类来具体说明移动语义问题。

```
#include <iostream>

class MemoryBlock
{
public:

    // 构造器（初始化资源）
    explicit MemoryBlock(size_t length)
        : _length(length)
        , _data(new int[length])
    {
    }

    // 析构器（释放资源）
    ~MemoryBlock()
    {
        if (_data != nullptr)
        {
            delete[] _data;
        }
    }

    // 拷贝构造器（实现拷贝语义：拷贝that）
    MemoryBlock(const MemoryBlock& that)
        // 拷贝that对象所拥有的资源
        : _length(that._length)
        , _data(new int[that._length])
    {
        std::copy(that._data, that._data + _length, _data);
    }

    // 拷贝赋值运算符（实现拷贝语义：释放this 并 拷贝that）
    MemoryBlock& operator=(const MemoryBlock& that)
    {
        if (this != &that)
        {
            // 释放自身的资源
            delete[] _data;

            // 拷贝that对象所拥有的资源
            _length = that._length;
            _data = new int[_length];
            std::copy(that._data, that._data + _length, _data);
        }
        return *this;
    }

    // 移动构造器（实现移动语义：移动that）
    MemoryBlock(MemoryBlock&& that)
        // 将自身的资源指针指向that对象所拥有的资源
        : _length(that._length)
        , _data(that._data)
    {
        // 将that对象原本指向该资源的指针设为空值
        that._data = nullptr;
        that._length = 0;
    }

    // 移动赋值运算符（实现移动语义：释放this 并 移动that）
    MemoryBlock& operator=(MemoryBlock&& that)
    {
        if (this != &that)
        {
            // 释放自身的资源
            delete[] _data;

            // 将自身的资源指针指向that对象所拥有的资源
            _data = that._data;
            _length = that._length;

            // 将that对象原本指向该资源的指针设为空值
            that._data = nullptr;
            that._length = 0;
        }
        return *this;
    }
private:
    size_t _length; // 资源的长度
    int* _data; // 指向资源的指针，代表资源本身
};

MemoryBlock f() { return MemoryBlock(50); }

int main()
{
    MemoryBlock a = f();            // 调用移动构造器，移动语义
    MemoryBlock b = a;              // 调用拷贝构造器，拷贝语义
    MemoryBlock c = std::move(a);   // 调用移动构造器，移动语义
    a = f();                        // 调用移动赋值运算符，移动语义
    b = a;                          // 调用拷贝赋值运算符，拷贝语义
    c = std::move(a);               // 调用移动赋值运算符，移动语义
}
```

转发型引用

如果某个变量或参数被声明为T&&类型，并且T需要经过推导才可确定，那么这个变量或参数就是一个转发型引用（forwarding reference）。
转发型引用由以下两种语法形式产生

- 如果某个变量被声明为auto&&类型，那么这个变量就是一个转发型引用
- 在函数模板中，如果某个参数被声明为T&&类型，并且T是一个需要经过推导才可确定的模板参数类型，那么这个参数就是一个转发型引用

转发型引用是不稳定的，它的实际类型由它所绑定的值来确定。转发型引用既可以绑定左值，也可以绑定右值。
如果绑定左值，转发型引用就成了左值引用。如果绑定右值，转发型引用就成了右值引用。
转发型引用在被C++标准所承认之前曾经被称作万能引用（universal reference）。万能引用这一术语的发明者，Effective C++系列的作者Scott Meyers认为，如此异常灵活的引用类型不属于右值引用，它应该拥有自己的名字。

对于某个转发型引用类型的变量（auto&&类型）来说

- 如果初始化表达式为左值（类型为U&），该变量将成为左值引用（类型为U&）。
- 如果初始化表达式为右值（类型为U&&），该变量将成为右值引用（类型为U&&）。

对于函数模板中的某个转发型引用类型的形参（T&&类型）来说

- 如果对应的实参为左值（类型为U&），模板参数T将被推导为引用类型U&，该形参将成为左值引用（类型为U&）。
- 如果对应的实参为右值（类型为U&&），模板参数T将被推导为非引用类型U，该形参将成为右值引用（类型为U&&）。

```
struct X {};
X&& var1 = X();                             // var1是右值引用，只能绑定右值X()
auto&& var2 = var1;                         // var2是转发型引用，可以绑定左值var1
                                            // var2的实际类型等同于左值var1，即X&
auto&& var3 = X();                          // var3是转发型引用，可以绑定右值X()
                                            // var3的实际类型等同于右值X()，即X&&

template<typename T>
void g(std::vector<typename T>&& param1);  // param1是右值引用
template<typename T>
void f(T&& param2);                         // param2是转发型引用

X a;
f(a);               // 模板函数f()的形参param2是转发型引用，可以绑定左值a
                    // 在此次调用中模板参数T将被推导为引用类型X&
                    // 而形参param2的实际类型将等同于左值a，即X&
f(X());             // 模板函数f()的形参param2是转发型引用，可以绑定右值X()
                    // 在此次调用中模板参数T将被推导为非引用类型X
                    // 而形参param2的实际类型将等同于右值X()，即X&&

// 更多右值引用和转发型引用
const auto&& var4 = 10;                             // 右值引用
template<typename T>
void h(const T&& param1);                           // 右值引用
template <typename T/*, class Allocator = allocator*/>
class vector
{
public:
    void push_back( T&& t );                        // 右值引用
    template <typename Args...>
    void emplace_back( Args&&... args );            // 转发型引用
};
```

完美转发

完美转发（perfect forwarding）问题是指函数模板在向其他函数转发（传递）自身参数（形参）时该如何保留该参数（实参）的左右值属性的问题。也就是说函数模板在向其他函数转发（传递）自身形参时，如果相应实参是左值，它就应该被转发为左值；同样如果相应实参是右值，它就应该被转发为右值。这样做是为了保留在其他函数针对转发而来的参数的左右值属性进行不同处理（比如参数为左值时实施拷贝语义；参数为右值时实施移动语义）的可能性。如果将自身参数不分左右值一律转发为左值，其他函数就只能将转发而来的参数视为左值，从而失去针对该参数的左右值属性进行不同处理的可能性。

转发型引用的引入主要是为了解决完美转发问题。在函数模板中需要保留左右值属性的参数，也就是要被完美转发的参数须被声明为转发型引用类型，即参数必须被声明为T&&类型，而T必须被包含在函数模板的模板参数列表之中。按照转发型引用类型形参的特点，该形参将根据所对应的实参的左右值属性而分别蜕变成左右值引用。但无论该形参成为左值引用还是右值引用，该形参在函数模板内都将成为左值。这是因为该形参有名字，左值引用是左值，具名右值引用也同样是左值。如果在函数模板内照原样转发该形参，其他函数就只能将转发而来的参数视为左值，完美转发任务将会失败。

```
#include<iostream>
using namespace std;

struct X {};
void inner(const X&) {cout << "inner(const X&)" << endl;}
void inner(X&&) {cout << "inner(X&&)" << endl;}
template<typename T>
void outer(T&& t) {inner(t);}

int main()
{
    X a;
    outer(a);
    outer(X());
}
//inner(const X&)
//inner(const X&)
```

std::forward()

要在函数模板中完成完美转发转发型引用类型形参的任务，我们必须在相应实参为左值，该形参成为左值引用时把它转发成左值，在相应实参为右值，该形参成为右值引用时把它转发成右值。此时我们需要标准库函数std::forward()。
标准库函数 std::forward<T>(t) 有两个参数：模板参数 T 与 函数参数 t。函数功能如下：

- 当T为左值引用类型U&时，t 将被转换为无名左值引用（左值，类型为U&）。
- 当T为非引用类型U或右值引用类型U&&时，t 将被转换为无名右值引用（右值，类型为U&&）。

使用此函数，我们在函数模板中转发类型为T&&的转发型引用参数 t 时，只需将参数 t 替换为std::forward<T>(t)即可完成完美转发任务。这是因为

- 如果 t 对应的实参为左值（类型为U&），模板参数T将被推导为引用类型U&，t 成为具名左值引用（类型为U&），std::forward<T>(t)就会把 t 转换成无名左值引用（左值，类型为U&）。
- 如果 t 对应的实参为右值（类型为U&&），模板参数T将被推导为非引用类型U，t 成为具名右值引用（类型为U&&），std::forward<T>(t)就会把 t 转换成无名右值引用（右值，类型为U&&）。

```
#include<iostream>
using namespace std;

struct X {};
void inner(const X&) {cout << "inner(const X&)" << endl;}
void inner(X&&) {cout << "inner(X&&)" << endl;}
template<typename T>
void outer(T&& t) {inner(forward<T>(t));}

int main()
{
    X a;
    outer(a);
    outer(X());
}
//inner(const X&)
//inner(X&&)
```