1. 非常量左值引用：能绑定到非常量左值，不能绑定常量左值、非常量右值、常量右值

常量左值引用能绑定非常量左值、常量左值、非常量右值、常量右值

非常量右值引用能绑定非常量右值，不能绑定非常量左值、常量左值和常量右值

常量右值能绑定非常量右值和常量右值，不能绑定常量左值和非常量左值

1. **右值引用特点：**
2. 通过右值引用的声明，右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一样长，只要该变量还活着，该右值临时量将会一直存活下去。
3. 右值引用独立于左值和右值。意思是右值引用类型的变量可能是左值也可能是右值
4. T&& t在发生自动类型推断的时候，它是未定的引用类型（universal references），如果被一个左值初始化，它就是一个左值；如果它被一个右值初始化，它就是一个右值，它是左值还是右值取决于它的初始化。
5. **模板类型推导：**

template<typename T>  
void f(ParamType param);

f(expr); // call f with some expression

在编译过程中，编译器会用expr去推导两个类型：一个是T，一个是ParamType。这两个类型通常不一样的，因为ParamType经常包括一些修饰，比如const或者引用。

分三种情况：

1. **ParamType是一个指针或引用 ，但不是统一引用。**

推导原理：（1）如果expr类型是一个引用，忽略引用部分；（2）expr的类型和ParamType模式匹配的部分决定了T

比如这样的模板，

template<typename T>  
void f(T& param); // param is a reference

我们的变量声明是这样的，

int x = 27;            // x is an int

const int cx = x;   // cx is a const int

const int& rx = x; // rx is a reference to x as a const int

各种变量对ParamType和T的推导出的类型如下，

f(x);                     // T is int, param's type is int&  
f(cx);                    // T is const int,  
                         // param's type is const int&  
f(rx);                    // T is const int,  
                         // param's type is const int&

在第二和第三个调用中，注意到因为cx和rx是常量值，所以T被推导为const int，于是产生了一个const int&的参数类型，对调用者来说这很重要，当他们传递一个常量对象给一个引用参数时，他们希望对象仍然是不可改变的，也就是说参数是个指向const的引用。这就是为什么我们传递一个常量对象给一个T&参数的模板时是安全的：对象的常量性变成了T推导出的类型的一部分。

在第三个例子中，注意到即使rx的类型是引用，T依然被推导为非引用的。这是因为在类型推导中rx的引用特性被忽略了。

1. **ParamType是个统一引用**

模板使用统一引用的话，事情就不是那么显然了。这样的参数被称作右值参数（就是说在函数模板用T做参数时，统一引用的声明参数为T&&），但当左值参数传入时，它们的表现不一样

（1）假如expr是个左值，T和ParamType都被推导为左值引用。这里有两个地方不同寻常，第一，这是唯一一种场景下模板类型推导把T推导成一个引用。第二，尽管ParamType是使用了右值引用的语法来声明的，它的推导类型却是一个左值引用。

（2）假如expr是个右值，通用规则（就是场景1）会适用。

template<typename T>  
void f(T&& param);              // param is now a universal reference

int x = 27;                          // as before  
const int cx = x;                   // as before  
const int& rx = x;                 // as before

f(x);                                     // x is lvalue, so T is int&,  
                                           // param's type is also int&

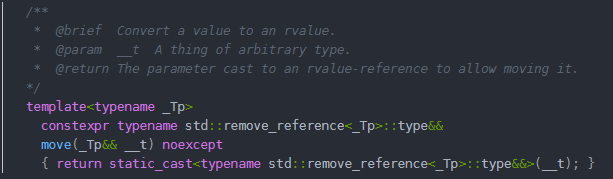
f(cx);                                  // cx is lvalue, so T is const int&,  
                                          // param's type is also const int&

f(rx);                                  // rx is lvalue, so T is const int&,  
                                          // param's type is also const int&  
 f(27);                                // 27 is rvalue, so T is int,  
                                         // param's type is therefore int&&

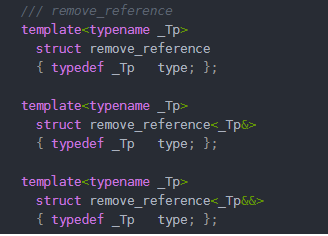
**4、move**

（1）函数功能：功能将一个左值/右值， 转换为右值引用。 主要是将左值强制转为右值引用，因为右值引用无法直接绑定到左值上， 为了能让右值引用绑定到左值上， 必须将左值转为右值引用，std::move提供做的就是这个。 对于传入右值， 那么std::move将什么都不做， 直接返回对应的右值引用.

（2）实现原理：



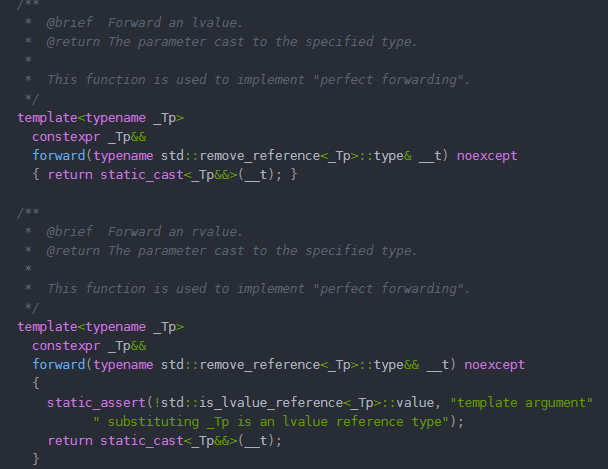
接着看看std::remove\_reference<\_Tp>的实现，可以看出，其主要是将去除类型的引用。 对于模板参数T, T&, T&&. 其type类型都为T, **而且T如果有const属性，则type也会保留**.



std::move函数类型为Tp\_&&, 一个指向模板类型参数的右值引用， 通过引用折叠，此参数可以与任意类型匹配。函数的返回值为std::remove\_reference<\_Tp>::type&&，std::remove\_reference<*Tp>::type类型为Tp, 因此，返回类型Tp*&&.

5、forward

std::forward实现, forward提供两个重载版本， 一个针对左值， 一个针对右值。



6、move与forward的区别：

函数返回值的不同：move 返回std::remove\_reference<\_Tp>::type&&，此处不涉及引用折叠，所以只返回右值引用；forward返回\_Tp&&，这就涉及了引用折叠，并且通过引用折叠实现完美转发