右值引用是 C++11 引入的与 Lambda 表达式齐名的重要特性之一。它的引入解决了 C++ 中大量的历史遗留问题，消除了诸如 `std::vector`、`std::string` 之类的额外开销，也才使得函数对象容器 `std::function` 成为了可能。

左值、右值的纯右值、将亡值、右值

要弄明白右值引用到底是怎么一回事，必须要对左值和右值做一个明确的理解。

左值(lvalue, left value)，顾名思义就是赋值符号左边的值。准确来说，左值是表达式（不一定是赋值表达式）后依然存在的持久对象。

右值(rvalue, right value)，右边的值，是指表达式结束后就不再存在的临时对象。

而 C++11 中为了引入强大的右值引用，将右值的概念进行了进一步的划分，分为：纯右值、将亡值。

纯右值(prvalue, pure rvalue)，纯粹的右值，要么是纯粹的字面量，例如 `10`, `true`；要么是求值结果相当于字面量或匿名临时对象，例如 `1+2`。非引用返回的临时变量、运算表达式产生的临时变量、原始字面量、Lambda 表达式都属于纯右值。

将亡值(xvalue, expiring value)，是 C++11 为了引入右值引用而提出的概念（因此在传统 C++中，纯右值和右值是统一个概念），也就是即将被销毁、却能够被移动的值。

将亡值可能稍有些难以理解，我们来看这样的代码：

```
std::vector<int> foo() {
 std::vector<int> temp = {1, 2, 3, 4};
 return temp;
}

std::vector<int> v = foo();
```

在这样的代码中，函数 `foo` 的返回值 `temp` 在内部创建然后被赋值给 `v`，然而 `v` 获得这个对象时，会将整个 temp 拷贝一份，然后把 `temp` 销毁，如果这个 `temp` 非常大，这将造成大量额外的开销（这也就是传统 C++ 一直被诟病的问题）。在最后一行中，`v` 是左值、`foo()` 返回的值就是右值（也是纯右值）。但是，`v` 可以被别的变量捕获到，而 `foo()` 产生的那个返回值作为一个临时值，一旦被 `v` 复制后，将立即被销毁，无法获取、也不能修改。

将亡值就定义了这样一种行为：临时的值能够被识别、同时又能够被移动。

右值引用和左值引用

需要拿到一个将亡值，就需要用到右值引用的申明：`T &&`，其中 `T` 是类型。右值引用的声明让这个临时值的生命周期得以延长、只要变量还活着，那么将亡值

将继续存活。

C++11 提供了 `std::move` 这个方法将左值参数无条件的转换为右值，有了它我们就能够方便的获得一个右值临时对象，例如：

```
#include <iostream>
#include <string>

void reference(std::string& str) {
 std::cout << "左值" << std::endl;
}
void reference(std::string&& str) {
 std::cout << "右值" << std::endl;
}

int main()
{
 std::string lv1 = "string,"; // lv1 是一个左值
 // std::string&& r1 = s1; // 非法, s1 在全局上下文中没有声明
 std::string&& rv1 = std::move(lv1); // 合法, std::move 可以将左值转移为右值
 std::cout << "rv1 = " << rv1 << std::endl; // string,

 const std::string& lv2 = lv1 + lv1; // 合法, 常量左值引用能够延长临时变量的生命周期
 // lv2 += "Test"; // 非法, 引用的右值无法被修改
 std::cout << "lv2 = "<<lv2 << std::endl; // string,string

 std::string&& rv2 = lv1 + lv2; // 合法, 右值引用延长临时对象的生命周期
 rv2 += "string"; // 合法, 非常量引用能够修改临时变量
 std::cout << "rv2 = " << rv2 << std::endl; // string,string,string,

 reference(rv2); // 输出左值
}
```

注意：`rv2` 虽然引用了一个右值，但由于它是一个引用，所以 `rv2` 依然是一个左值。

移动语义

传统 C++ 通过拷贝构造函数和赋值操作符为类对象设计了拷贝/复制的概念，但为了实现对资源的移动操作，调用者必须使用先复制、再析构的方式，否则就需要自己实现移动对象的接口。试想，搬家的时候是把家里的东西直接搬到新家去，而不是将所有东西复制一份（重买）再放到新家、再把原来的东西全部销毁，这是非常反人类的一件事情。

传统的 C++ 没有区分『移动』和『拷贝』的概念，造成了大量的数据移动，浪费时间和空间。右值引用的出现恰好就解决了这两个概念的混淆问题，例如：

```cpp
#include <iostream>
class A {
public:
 int *pointer;
 A() :pointer(new int(1)) {
 std::cout << "构造" << pointer << std::endl;
 }
 // 无意义的对象拷贝
 A(A& a) :pointer(new int(*a.pointer)) {
  std::cout << "拷贝" << pointer << std::endl;
  }

  A(A&& a) :pointer(a.pointer) {
  a.pointer = nullptr;
  std::cout << "移动" << pointer << std::endl;
  }

  ~A() {
  std::cout << "析构" << pointer << std::endl;
  delete pointer;
  }
};
// 防止编译器优化
A return_rvalue(bool test) {
 A a,b;
 if(test) return a;
 else return b;
}
int main() {
 A obj = return_rvalue(false);
 std::cout << "obj:" << std::endl;
 std::cout << obj.pointer << std::endl;
 std::cout << *obj.pointer << std::endl;

 return 0;
}
```

在上面的代码中：

1. 首先会在 `return_rvalue` 内部构造两个 `A` 对象，于是获得两个构造函数的输出；
2. 函数返回后，产生一个将亡值，被 `A` 的移动构造（`A(A&&)`）引用，从而延长生命周期，并将这个右值中的指针拿到，保存到了 `obj` 中，而将亡值的指针被设置为 `nullptr`，防止了这块内存区域被销毁。

从而避免了无意义的拷贝构造，加强了性能。再来看看涉及标准库的例子：

```
#include <iostream> // std::cout
#include <utility> // std::move
#include <vector> // std::vector
#include <string> // std::string

int main() {

 std::string str = "Hello world.";
 std::vector<std::string> v;

 // 将使用 push_back(const T&), 即产生拷贝行为
 v.push_back(str);
 // 将输出 "str: Hello world."
 std::cout << "str: " << str << std::endl;

 // 将使用 push_back(const T&&), 不会出现拷贝行为
 // 而整个字符串会被移动到 vector 中，所以有时候 std::move 会用来减少拷贝出现的开销
 // 这步操作后，str 中的值会变为空
 v.push_back(std::move(str));
 // 将输出 "str: "
 std::cout << "str: " << str << std::endl;

 return 0;
}
```

完美转发

前面我们提到了，一个声明的右值引用其实是一个左值。这就为我们进行参数转发（传递）造成了问题：

```
void reference(int& v) {
 std::cout << "左值" << std::endl;
}
void reference(int&& v) {
 std::cout << "右值" << std::endl;
}
```

```cpp
template <typename T>
void pass(T&& v) {
 std::cout << "普通传参:";
 reference(v); // 始终调用 reference(int& )
}
int main() {
 std::cout << "传递右值:" << std::endl;
 pass(1); // 1是右值, 但输出左值

 std::cout << "传递左值:" << std::endl;
 int v = 1;
 pass(v); // v是左引用, 输出左值

 return 0;
}
```

对于 `pass(1)` 来说，虽然传递的是右值，但由于 `v` 是一个引用，所以同时也是左值。因此 `reference(v)` 会调用 `reference(int&)`，输出『左值』。而对于 `pass(v)` 而言，`v`是一个左值，为什么会成功传递给 `pass(T&&)` 呢?

这是基于引用坍缩规则的：在传统 C++ 中，我们不能够对一个引用类型继续进行引用，但 C++ 由于右值引用的出现而放宽了这一做法，从而产生了引用坍缩规则，允许我们对引用进行引用，既能左引用，又能右引用。但是却遵循如下规则：

函数形参类型	实参参数类型	推导后函数形参类型
T&	左引用	T&
T&	右引用	T&
T&&	左引用	T&
T&&	右引用	T&&

因此，模板函数中使用 `T&&` 不一定能进行右值引用，当传入左值时，此函数的引用将被推导为左值。更准确的讲，无论模板参数是什么类型的引用，当且仅当实参类型为右引用时，模板参数才能被推导为右引用类型。这才使得 `v` 作为左值的成功传递。

完美转发就是基于上述规律产生的。所谓完美转发，就是为了让我们在传递参数的时候，保持原来的参数类型（左引用保持左引用，右引用保持右引用）。为了解决这个问题，我们应该使用 `std::forward` 来进行参数的转发（传递）：

```cpp
#include <iostream>
```

```
#include <utility>
void reference(int& v) {
 std::cout << "左值引用" << std::endl;
}
void reference(int&& v) {
 std::cout << "右值引用" << std::endl;
}
template <typename T>
 void pass(T&& v) {
  std::cout << "普通传参:";
  reference(v);
  std::cout << "std::move 传参:";
  reference(std::move(v));
  std::cout << "std::forward 传参:";
  reference(std::forward<T>(v));

 }
 int main() {
  std::cout << "传递右值:" << std::endl;
  pass(1);

  std::cout << "传递左值:" << std::endl;
  int v = 1;
  pass(v);

  return 0;
 }
```

输出结果为：

```
传递右值:
普通传参:左值引用
std::move 传参:右值引用
std::forward 传参:右值引用
传递左值:
普通传参:左值引用
std::move 传参:右值引用
std::forward 传参:左值引用
```

无论传递参数为左值还是右值，普通传参都会将参数作为左值进行转发，所以 `std::move` 总会接受到一个左值，从而转发调用了`reference(int&&)` 输出右值引用。

唯独 `std::forward` 即没有造成任何多余的拷贝，同时完美转发(传递)了函数的实参给了内部调用的其他函数。

> `std::forward` 和 `std::move` 一样，没有做任何事情，`std::move` 单纯的将左值转化为右值，`std::forward` 也只是单纯的将参数做了一个类型的转换，从是实现来看，`std::forward<T>(v)` 和 `static_cast<T&&>(v)` 是完全一样的。