完美转发

前置信息

万能引用与引用折叠

```
template <typenam T>
void func(T&& value) {}
```

在前面我们已经知道了 func 中的形参其实是一个万能引用，当实参为左值时，T 和 value 的类型均为左值引用；当实参为右值时，T 为实参类型，value 为右值引用

```
int i = 1024;
int &&k = i;

func(i);
func(1024);
func(k);
```

- func(i)
  - 实参为左值，故此时 T 将被推断为 int &，value 则为 int &
  - 如果我们把 int & 带入到函数原型中去的话，得到的将是 func(int& &&value)，在编译器内部将会对其进行"引用折叠"，最终结果为 int &
- func(1024);
  - 实参为右值，故此时 T 将被推断为 int，value 则为 int &&，也就是一个右值引用
- func(k);
  - **需要注意的是，虽然 k 是一个右值引用，但是 k 本身是一个左值，还记得吗，变量都是左值**
  - 所以，此时 T 将被推断为 int &，value 也为 int &

本节使用的函数

这里用一个简单的 reverse_string 函数作为下面使用的基本函数用例

```
void reverse_string(string& s) {
    int i = 0, j = s.size() - 1;
    while (i < j) {
        std::swap(s[i], s[j]);
        i++, j--;
    }
}
```

- 该函数做的事情非常简单，接收一个 string 引用，并将其反转
-

```
string s = "Hello";
reverse_string(s);
cout << s << endl;
```

  - 输出结果为 "olleH"

① 普通函数模板转发

```
template <typename F, typename T>
void func_template(F f, T t) {
    f(t);
}
```

- 定义这样的一个函数模板，F 应为可调用对象，T 则是该可调用对象的参数
- 理论上来说，func_template 这个函数模板能够在内部调用任何形参数量为 1 的可调用对象

```
string s = "Hello";
func_template(reverse_string, s);
cout << s << endl;
```

- 现在我们把提前写好的 reverse_string 函数丢进去试试，编译正常，运行也不会报错
- 但是 stdout 的结果却是 "Hello"，并没有像我们期望的那样反转字符串 s
- **原因就在于 func_template 函数模板中在推断 T 的类型时，被推断成了值类型，修改无法传播到外部**

② 引用函数模板转发

```
template <typename F, typename T>
void func_template(F f, T& t) {
    f(t);
}
```

- 既然前面的 func_template 不能正常工作是因为 T 被推断成值类型，那么我们加一个引用不就好了？
-

```
string s = "Hello";
func_template(reverse_string, s);
cout << s << endl;
```

  - 再重新编译运行，诶，这次结果是正确的了

但是，此时 func_template 只能接收左值，如果我们的 f 可调用对象需要接收一个右值的话，func_template 将不能正常工作

③ 万能引用函数模板转发

```
template <typename F, typename T>
void func_template(F f, T&& t) {
    f(t);
}
```

- 既然引用函数模板不能接收右值，那么我们上万能引用不就好了？这个时候左值右值都能接收
-

```
string s = "Hello";
string rs = s;

func_template(reverse_string, s);
func_template(reverse_string, std::move(rs));
```

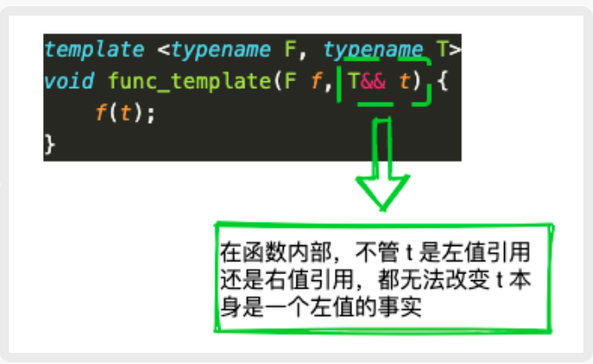
  - 现在 func_template 既能处理左值，又能处理右值

那么，如果我们把 reverse_string 这个函数改成接收右值引用呢？

- **void reverse_string(string&& s) { /* ..... */ }**

这个时候编译器就不干了，不管我们在调用 func_template 是实参为左值，还是实参为右值，编译器都会抛出错误

这是因为尽管 T&& t 是一个万能引用，既能接收左值又能接收右值，但是形参 t 却永远都是左值，那么我们把一个左值往接收右值引用的函数里面儿传递，当然会报错



④ 使用 std::forward

std::forward 的作用就是还原一个类型参数的左值、右值属性，我们传入一个左值，它将返回左值，传入一个右值，则返回一个右值

我们知道，实参的左值属性和右值属性实际上是保存在类型参数 T 中的。当实参为左值时，T 为左值引用；当实参为右值时，T 为非引用类型

```
template <typename F, typename T>
void func_template(F f, T&& t) {
    f(std::forward<T>(t));
}
```

- **当形参 t 的实参为左值时，std::forward 还原实参原有属性，那么 std::forward<T>(t) 的结果仍然为左值**
- **当形参 t 的实参为右值时，std::forward 还原实参原有属性，那么 std::forward<T>(t) 的结果就是一个右值**

因此，此时不管我们的 reverse_string 函数接收一个左值引用还是右值引用，func_template 调用时的实参我左值还是右值，该函数模板均能正常工作

另外需要注意的是，std::forward 的工作场景必需在万能引用的函数模板中，并且其作用是进行参数转发

一个小实验

```
void buz(int& value) {
    cout << "左值引用 buz 被调用" <<endl;
}

void buz(int&& value) {
    cout << "右值引用 buz 被调用" <<endl;
}

template <typename T>
void func(T&& t) {
    buz(t);                         // 不管实参是左值还是右值，t 反正是个左值，所以会调用左值引用 buz
    buz(std::forward<T>(t));   //  当实参为左值时，调用左值引用 buz；实参为右值时调用右值引用 buz
    buz(std::move(t));           //  不管实参是左值还是右值，t 是一个左值，std::move(t) 强制转换成右值
}
```

- buz 是一个重载函数，对左值引用和右值引用的形参进行重载

- 左值调用
  -

```
int buffer = 1024;
func(buffer);
```

    - 左值引用 buz 被调用
    - 左值引用 buz 被调用
    - 右值引用 buz 被调用
- 右值调用
  - func(1024);
    - 左值引用 buz 被调用
    - 右值引用 buz 被调用
    - 右值引用 buz 被调用

可以看到，只有 buz(std::forward<T>(t)); 会根据实参是左值还是右值来动态地选择调用函数

一个小应用

C++ 中的线程要么调用 join 进行连接，要么调用 detach 进行分离，在没有 C++20 的 jthread 之前，我们自行对其封装

```
class thread_wrapper {
private:
    thread m_thread;
public:
    template <typename F, typename... Args>           // 可变类型参数模板
    explicit thread_wrapper(F func, Args&&... args)
    : m_thread(func, std::forward<Args>(args)...) {}    // 完美转发相关参数

    thread_wrapper(const thread& other) = delete;

    ~thread_wrapper() {        // 利用 RAII 在析构函数中对未分离的线程进行连接
        if (m_thread.joinable()) m_thread.join();
    }
};
```

```
int main() {
    thread_wrapper t1{buz, 1024};  // 注意，这里的 buz 函数不能重载
    return 0;
}
```