实现function

在模板中如何正确地传递引用？

需求：不区分左值引用和右值引用。左值就是左值引用，右值就是右值引用。

在模板中参数有 && 两个符号即可。

```
void func(T &&a) {
    // T &&a并不是就是说他是右值引用，而是都可以，引用
}
```

引用折叠：在模板当中，奇数个&就是左值引用。偶数个&就是右值引用。

```
typename remove_reference<T>::type c;
// 将T如果是引用地话，则去掉引用。
```

1.remove_reference的实现：

```
template<typename T>
struct remove_reference {
    typedef T type;
};
// 对于左值的特殊处理。
template<typename T>
struct remove_reference<T &> {
    typedef T type;
};
// 对于右值的特殊处理
template<typename T>
struct remove_reference<T &&> {
    typedef T type;
};

template<typename T>
void func(T &&a) {
    typename remove_reference<T>::type c;
    cout << "function T& a = " << a << endl;
    return ;
}
int main() {
    int n = 123;
    func(n);
    func(123);
    typename remove_reference<int &&>::type a;
    typename remove_reference<int &>::type b;
    typename remove_reference<int>::type c;
    return 0;
}
```

2.add_const 的实现

```
template<typename T>
struct add_const {
    typedef const T type;
};
template<typename T>
struct add_const<const T> {
    typedef const T type;
};
```

3.add_lvalue_reference的实现

```
template<typename T>
struct add_lvalue_reference {
    // 先把T当成普通类型来看
    typedef T & type;
};
// 如果T类型原本就是左值引用呢？上述代码不就变成右值引用了嘛
template<typename T>
struct add_lvalue_reference<T &> {
    typedef T & type;
};
// 当然还有右值引用的情况啦
template<typename T>
struct add_lvalue_reference<T &&>{
    typedef T & type;
};
```

4. remove_pointer的实现

```
template<typename T>
struct remove_pointer {
    typedef T type;
};
template<typename T>
struct remove_pointer<T *> {
    // 有没有可能是一个二维指针：int **？——当然有可能
    typedef typename remove_pointer<T>::type type;
};
```

5. ref()关键字 / bind()用法

表示我这里要传递的是一个引用，而不是单纯的参数n

```
BEGINS(bind_usage)

int add(int a, int b) {
    cout << "add(a, b) = ";
    return a + b;
}
void add_one(int &n) {
    n += 1;
    return ;
}
void func(int n, const char *msg) {
    cout << n << " " << msg << endl;
    return ;
}
int main() {
    // 1. bind基础用法
    auto t1 = bind(add, 3, 4); // 返回一个可调用对象
    cout << t1() << endl;

    // 2. ref关键字传递引用
    int n = 1;
    cout << "n = " << n << endl;
    auto t2 = bind(add_one, ref(n));
    t2(), t2(), t2();
    cout << "after 3 times t2 function call, n = " << n << endl;

    // 3. 交换参数传递顺序
    func(3, "hello world");
    auto t3 = bind(func, std::placeholders::_4, std::placeholders::_1);
    // 4. placeholders::_n 表示把调用时候的第n个参数传给当前顺位的函数参数
    t3("hello world", 3, 4, 5);
    return 0;
}
ENDS(bind_usage)
```

最终代码：

```
BEGINS(function_impl)
int add(int a, int b) {
    cout << "normal function : ";
    return a + b;
}
class ADD_MULT {
public:
    ADD_MULT(int x) : x(x) {}
    int operator()(int a, int b) {
        cout << "functor : ";
        return (a + b) * x;
    }
private:
    int x;
};
template<typename T, typename ...ARGS>
class Base {
public:
    virtual T run(ARGS...) = 0;
    virtual Base<T, ARGS...> *getCopy() = 0;
    virtual ~Base() {}
};
template<typename T, typename ...ARGS>
class normal_function : public Base<T, ARGS...> {
public:
    normal_function(T (*ptr)(ARGS...)) : func(ptr) {}
    T run(ARGS ...args) override {
        // 如果直接传，会出现所有的都会变成左值的情况
        return func(forward<ARGS>(args)...);// ...表明这是一个变参列表
    }
    Base<T, ARGS...> *getCopy() override {
        return new normal_function(*this);
    }

private:
    T (*func)(ARGS...);
};

template<typename CLASS_T, typename T, typename ...ARGS>
class functor : public Base<T, ARGS...> {
public:
    functor(CLASS_T obj) : obj(obj) {}
    T run(ARGS ...args) override {
        return obj(forward<ARGS>(args)...);
    }
    Base<T, ARGS...> *getCopy() override {
```

```
        return new functor(*this);
    }
private:
    CLASS_T obj;
};

template<typename T, typename ...ARGS> class function;
template<typename T, typename ...ARGS>
class function<T(ARGS...)> {
public:
    // 1. 构造函数一：普通函数
    function(T (*ptr)(ARGS...)) : ptr(new normal_function<T, ARGS...>(ptr)) {}

    // 2. 构造函数二：任意类型的函数对象
    template<typename CLASS_T>
    function(CLASS_T obj) : ptr(new functor<CLASS_T, T, ARGS...>(obj)) {}

    // ptr类型既能存储普通函数，又能存储函数对象，ptr肯定有他自己的类型，而这种类型又有两种不同的应用场景，
    T operator()(ARGS ...args) {
        return ptr->run(forward<ARGS>(args)...);
    }
    function &operator=(const function<T(ARGS...)> &func) {
        delete this->ptr;
        this->ptr = func.ptr->getCopy();//ptr是一个虚拟类，怎么获得子类地方法呢？—设定一个纯虚函数，让
        return *this;
    }
    ~function() {
        delete this->ptr;
    }
private:
    Base<T, ARGS...> *ptr;
};

int main() {
    ADD_MULT add_mult(2);
    function<int(int, int)> f1 = add;
    function<int(int, int)> f2 = add_mult;
    cout << f1(3, 4) << endl;
    cout << f2(3, 4) << endl;
    f1 = f2;
    cout << f1(3, 4) << endl;
    return 0;
}
ENDS(function_impl)

int main() {
```

```
    // reference_param::main();
    // bind_usage::main();
    function_impl::main();
    return 0;
}
```