可能会疑惑,为什么写了一个void(void)函数签名,却可以传递lambda,普通函数,仿函数这些不同类型?

std::function借助了**类型擦除**的方法进行实现,类型擦除是一种借助多态来完成的技术,function类管理base类,传递进来的对象为derived类,通过把derived传递给base的方式实现了std::function可以管理不同类型的"函数"简单实现如下

```
template<typename func>
struct function;//function的声明func为func为函数签名

template<typename Rtype, typename...Args>//Rtype为函数返回类型, 形参包为传递给函数的参数
struct function<Rtype(Args...)> {//函数签名,也就是function<void(void)>,function<int(int,double)>这种形式

    template<typename T>
    function(T f):ptr(new eraseDerived<T>(f)) {}//注意,ptr指针是eraseBase类型,但是我们这里要创建一个Derived类型来完成多态
    //T 会识别出传进来的是函数指针还是lambda,或者是仿函数...,

    Rtype operator()(Args&&...args) {//这个仿函数是从function角度上来看,比如创建了function<void(void)> a,此时a()调用的就是该函数
        if (ptr)
            return ptr->invoke(std::forward<Args>(args)...);//invoke是调用对象本身自己的函数,通过完美转发传递参数.
        else throw std::bad_function_call();//空指针抛出异常
    }

    function(const function &other) {
        if (other.ptr)
            ptr = other.ptr->copy();//这里使用copy防止浅拷贝
        else ptr = nullptr;
    }

    function(function &&other) noexcept {
        ptr = other.ptr;
        other.ptr = nullptr;
    }

    function &operator=(const function &other) {
        if (this == &other) {
            return *this;
        }
        delete ptr;
        ptr = nullptr;

        if (other.ptr) {
            ptr = other.ptr->copy();//防止浅拷贝
        }
        return *this;
    }

    function &operator=(function &&other) noexcept {
```

```
        if (this == other) {
            return *this;
        }

        delete ptr;
        ptr = nullptr;
        ptr = other.ptr;
        other.ptr = nullptr;
        return *this;
    }

    ~function() {
        delete ptr;
    }

private:
    //实现类型擦除,
    //需要管理的对象需要实现invoke操作,和copy操作所以需要设置为虚函数
    struct eraseBase {
        virtual ~eraseBase() = default;
        virtual Rtype invoke(Args...args) const noexcept = 0;
        virtual eraseBase *copy() = 0;
    };

    template<typename T>
    struct eraseDerived : eraseBase {

        explicit eraseDerived(T f) : func(std::move(f)) {}

        Rtype invoke(Args...args) const noexcept override {//调用自己
            return func(std::forward<Args>(args)...);
        }

        eraseBase *copy() override {
            return new eraseDerived(func);
        }

    private:
        T func;
    };

public:

    eraseBase *ptr;
};
```

```
void foo() {
    std::cout << __func__ << std::endl;
}

void test(const function<void(void)> x) {
    std::cout << "YES" << std::endl;
}
```

```
int main() {
    //demo:
    function<int(int, int)> a = [](int x, int y) { return x + y; };
    std::cout << a(1, 2) << std::endl;//OK

    function<void(void)> b = foo;
    b();//OK

    struct A {
        void operator()() const {//const成员函数不能调用非const函数
            std::cout << "operator" << std::endl;
        }
    };

    function<void(void)> func = A{};
    func();
}
```