# 可调用对象 function

## 简介

C++ 中的可调用对象

1. 普通函数
   - `void foo(int size) { /* ..... */ }`
     `foo(1024);`
   - foo 为一个函数对象，函数对象当然是可调用对象，函数生来就是被调用的
2. 重载了 operator() 的 class
   - ```
     class Buz {
     public:
         void operator() (int size) { /* ..... */ }
     };
     Buz buz;
     buz(1024);
     ```
   - 此时 Buz 类的对象可调用，因为我们重载了 operator() 函数，或者说，重载了"函数调用运算符"
3. lambda 表达式
   - C++11 引入，本质上是一个匿名函数
   - `auto f = [](int size) { /* ..... */ };`
     `f(1024);`

这 3 个可调用对象的形参和返回值均相同，我们可以说它们的调用形式相同

## std::function

既然前面有 3 中可调用对象，那么，我们是否可以使用一个函数指针来保存这 3 个可调用对象呢?

```
void (*ptr)(int);

ptr = foo;    // right

ptr = f;      // right

ptr = buz;    // wrong
```

**可以看到，尽管 buz 是一个可调用对象，但是编译器并不认为它能够赋值给函数指针，这样的"差别对待"将会影响到我们使用 C++ 来进行函数式编程**

比如说我们有一个 handler，接收一个可调用对象和其参数，那么我们就需要单独的对可调用类对象进行函数重载

此时我们就可以使用 std::fuction 类模板，其参数类型为对象的调用形式，例如 int(int， double) 表示返回值为 int，形参分别为 int 、double 的调用形式

这时候我们就可以把 buz 丢进去了

```
#include <functional>

std::function<void(int)> f1 = foo;
std::function<void(int)> f2 = f;
std::function<void(int)> f3 = buz;
```

handler 就可以这样写

```
void handler(std::function<void(int)>& func, int arg) {
    func(arg);
}
```

std::function 中定义了完整的拷贝控制函数，也就是说，我们既可以拷贝和移动该类模板的实例对象

## std::bind

std::bind 非常类似于 Python 中的 functools.partial()，目的就是为了减少可调用对象的参数个数

作用

Python

```
def func(a, b, c, d):
    print(a, b, c, d)


if __name__ == "__main__":
    s1 = partial(func, 10, 20)              # fill a 、b
    s2 = partial(func, 10, 20, d=40)    # fill a、b、d
    s1(30, 40)
```

func() 函数本身接收 4 个参数，但是我们可以使用 partial 方法给该函数填充一部分的参数，函数返回另一个可调用对象。此时我们只需要再提供 c、d 两个参数即可

C++

```
void foo(int a, int b, int c, int d) {
    printf("%d-%d-%d-%d\n", a, b, c, d);
}

int main () {
    using std::placeholders::_1, std::placeholders::_2;
    auto f = std::bind(foo, 10, 20, _1, _2);
    f(30, 40);
}
```

C++ 中的 std::bind 就没有 partial 那么智能了，我们必须要使用诸如 _1, _2 的占位符对形参进行占位

std::bind(foo, 10, 20); 这样的写法编译器并不会自动地去推断"还剩两个形参"，也就是说，这么写是不对的

**即虽然 std::bind 允许我们预先为一个可调用对象添加部分参数，但是剩余的参数必须使用诸如 std::placeholders::_1、std::placeholders::_2 等占位符进行占位，必须保证形参个数保持一致**

这么做也带来了另外一个好处: 灵活性大大增强

我们可以任意的调整 _1、_2 等占位符的位置来表示缺省的函数形参

```
std::bind(foo, _1, 10, 20, _2);    // 此时 b = 10, c = 20
std::bind(foo, 10, _1, 20, _2);    // 此时 a = 10, c = 20
```

通常来说我们会使用 std::function<> 来代替使用 auto，使得代码更加清晰

```
std::function<void(int, int)> f = std::bind(foo, 10, 20, _1, _2);
f(30, 40);
```

1. 绑定普通函数
   - 当普通函数作为实参时，在调用 std::bind() 时将会默认的转换成函数指针。也就是说，std::bind(&foo, _1, 10, 20, _2); 这么写也是可以的
2. 绑定成员函数
   - **和 std::thread 一样，当我们绑定成员函数时，首先需要显式地使用诸如 &Buz::func 的方式将成员函数转换成函数指针**
   - 其次，因为在调用成员函数时需要知道该对象的归属，所以我们还需要把对象传入，也就是 "this pointer"
   - ```
     class Buz {
     public:
         void foo(const string& s) {/* ..... */}
     };

     Buz buz;
     std::function<void()> f = std::bind(&Buz::foo, &buz, "hello");
     ```
     如果要绑定引用类型参数的话，请使用 std::ref