C++单元测试、一种简单打桩方式的实现

此篇文章涉及大量代码示例更适合作为参考手册，需要时查查用法。对于技术人员代码是最好的解释。

桩，或称桩代码，是指用来代替关联代码或者未实现代码的代码。如果用函数 B1 来代替 B，那么，B 称为原函数，B1 称为桩函数。打桩就是调用 B 的地方变成调用 B1。

打桩主要涉及两点：

- 第一如何获取原函数地址
- 第二如何用桩函数替换原函数

桩函数替换原函数的原理介绍

主要用到 inline hook 技术，核心思想，通过替换目标函数头部指令，实现在函数执行之前跳转到其他的指令区域，执行完毕跳转回到原来的函数，跳转到的指令区域通常是我们自己编写的函数。

图 1 所示，如果原函数和桩函数同在 32 地址空间里，则采用 JMP 指令实现，

占 5 字节。

图 2 所示，如果原函数和桩函数不同在 32 地址空间里，则采用 MOV、PUSH、RET 指令实现，占 12 字节。

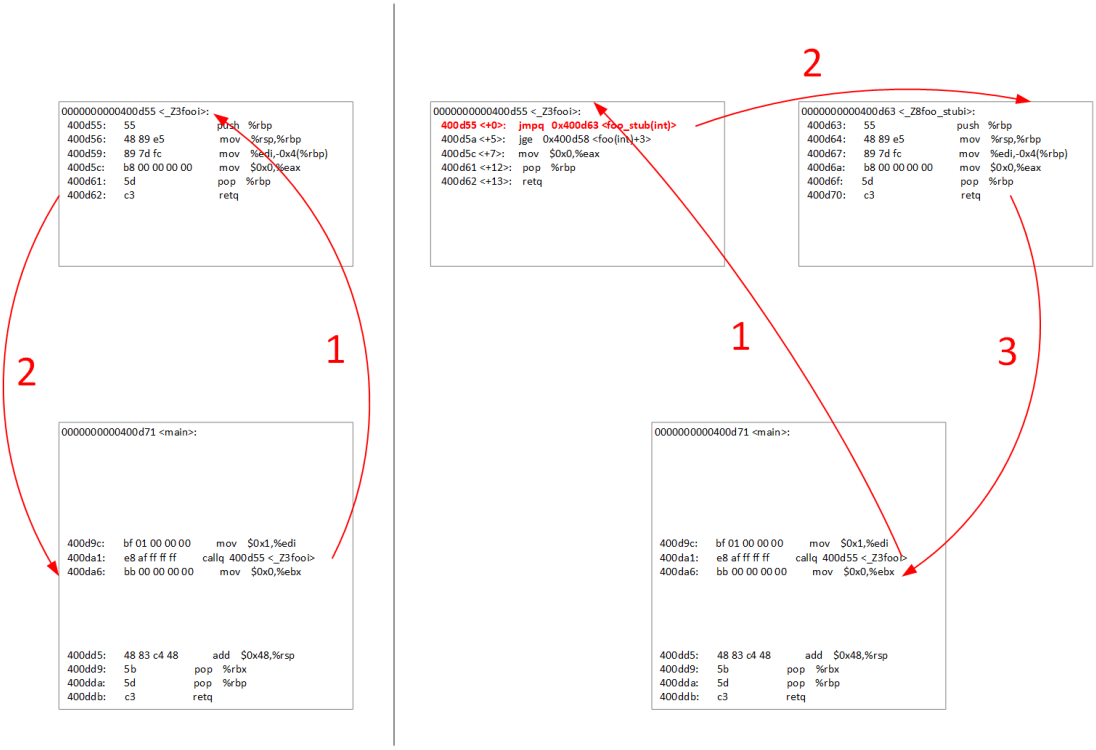

图 1

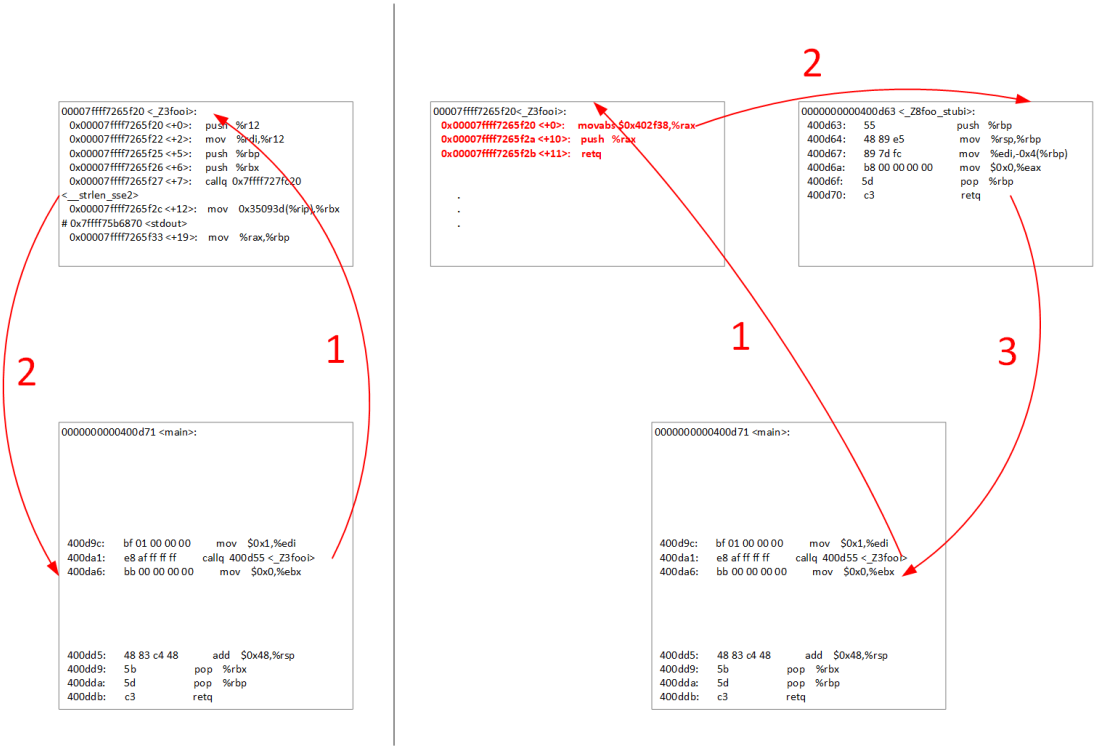

图 2

关键替换源码解释：

```
template<typename T,typename S>
void set(T addr, S addr_stub)    //addr 原函数地址，addr_stub 桩函数地址
{
        void * fn;
        void * fn_stub;
        fn = addrof(addr);               //强转地址
        fn_stub = addrof(addr_stub);     //强转地址
        struct func_stub *pstub;
        pstub = new func_stub;
        //start
        pstub->fn = fn;
#ifdef __x86_64__
        if(judge_far_jmp(fn, fn_stub))   //判断远跳还是近跳
        {
                pstub->far_jmp = true;
                memcpy(pstub->code_buf, fn, CODESIZE_MAX);//保留原函数现场
        }
        else
        {
                pstub->far_jmp = false;
                memcpy(pstub->code_buf, fn, CODESIZE_MIN); //保留原函数现场
        }
#else
        memcpy(pstub->code_buf, fn, CODESIZE);
#endif
#ifdef _WIN32
        DWORD lpflOldProtect;
        if(0 == VirtualProtect(pageof(pstub->fn), m_pagesize * 2,
PAGE_EXECUTE_READWRITE, &lpflOldProtect))
#else
        if (-1 == mprotect(pageof(pstub->fn), m_pagesize * 2, PROT_READ |
PROT_WRITE | PROT_EXEC))
#endif
        {
                throw("stub set mprotect to w+r+x faild");
        }

#ifdef __x86_64__
        if(pstub->far_jmp)
        {
```

```
                //12 byte 替换，MOV、PUSH、RET
                *(unsigned char*)fn = 0x48;
                *((unsigned char*)fn + 1) = 0xb8;
                *(unsigned long long *)((unsigned char *)fn + 2) =
(unsigned long long)fn_stub;
                *(unsigned char *)((unsigned char *)fn + 10) = 0x50;
                *(unsigned char *)((unsigned char *)fn + 11) = 0xc3;

        }
        else
        {
                //5 byte 替换，JMP 指令
                *(unsigned char *)fn = (unsigned char)0xE9;
                *(unsigned int *)((unsigned char *)fn + 1) = (unsigned
char *)fn_stub - (unsigned char *)fn - CODESIZE_MIN;
        }
#else
         //5 byte 替换，JMP 指令
         *(unsigned char *)fn = (unsigned char)0xE9;
         *(unsigned int *)((unsigned char *)fn + 1) = (unsigned char
*)fn_stub - (unsigned char *)fn - CODESIZE;
#endif


#ifdef _WIN32
        if(0 == VirtualProtect(pageof(pstub->fn), m_pagesize * 2,
PAGE_EXECUTE_READ, &lpflOldProtect))
#else
        if (-1 == mprotect(pageof(pstub->fn), m_pagesize * 2, PROT_READ |
PROT_EXEC))
#endif
        {
                throw("stub set mprotect to r+x failed");
        }
        m_result.insert(std::pair<void*,func_stub*>(fn,pstub));
        return;
}
```

获取原函数地址，各种函数的打桩用法

各种类函数地址的获取方式各不相同，不同平台同种类的获取方式也不同，下面将会列举一些常见类型函数的地址获取方式。

桩函数写法基于调用约定，C++中常见的调用约定有 stdcall、cdecl、fastcall 和 thiscall

- **普通函数打桩(非 static)**

```
//for linux and windows
#include<iostream>
#include "stub.h"
using namespace std;
int foo(int a)
{
    cout<<"I am foo"<<endl;
    return 0;
}
int foo_stub(int a)
{
    cout<<"I am foo_stub"<<endl;
    return 0;
}


int main()
{
    Stub stub;
    stub.set(foo, foo_stub);
    foo(1);
    return 0;
}
```

- **静态成员函数打桩**

```
//for linux and windows
#include<iostream>
#include "stub.h"
using namespace std;
class A{
    int i;
public:
    static int foo(int a){
        cout<<"I am A_foo"<<endl;
        return 0;
    }
};

int foo_stub(int a)
{
    cout<<"I am foo_stub"<<endl;
    return 0;
}


int main()
{
    Stub stub;
    stub.set(ADDR(A,foo), foo_stub);

    A::foo(1);
    return 0;
}
```

- **实例成员函数打桩**

```
//for linux, __cdecl
#include<iostream>
#include "stub.h"
using namespace std;
class A{
    int i;
public:
    int foo(int a){
        cout<<"I am A_foo"<<endl;
        return 0;
    }
```

```
};

int foo_stub(void* obj, int a)
{
    A* o= (A*)obj;
    cout<<"I am foo_stub"<<endl;
    return 0;
}


int main()
{
    Stub stub;
    stub.set(ADDR(A,foo), foo_stub);
    A a;
    a.foo(1);
    return 0;
}

//for windows, __thiscall
#include<iostream>
#include "stub.h"
using namespace std;
class A{
    int i;
public:
    int foo(int a){
        cout<<"I am A_foo"<<endl;
        return 0;
    }
};


class B{
public:
    int foo_stub(int a){
        cout<<"I am foo_stub"<<endl;
        return 0;
    }
};

int main()
{
    Stub stub;
```

```
    stub.set(ADDR(A,foo), ADDR(B,foo_stub));
    A a;
    a.foo(1);
    return 0;
}
```

- **模板函数打桩(实例成员函数)**

```
//for linux，__cdecl
#include<iostream>
#include "stub.h"
using namespace std;
class A{
public:
   template<typename T>
   int foo(T a)
   {
        cout<<"I am A_foo"<<endl;
        return 0;
   }
};

int foo_stub(void* obj, int x)
{
    A* o= (A*)obj;
    cout<<"I am foo_stub"<<endl;
    return 0;
}


int main()
{
    Stub stub;
    stub.set((int(A::*)(int))ADDR(A,foo), foo_stub);
    A a;
    a.foo(5);
    return 0;
}

//for windows，__thiscall
#include<iostream>
#include "stub.h"
using namespace std;
```

```
class A{
public:
   template<typename T>
   int foo(T a)
   {
        cout<<"I am A_foo"<<endl;
        return 0;
   }
};


class B {
public:
        int foo_stub(int a) {
                cout << "I am foo_stub" << endl;
                return 0;
        }
};


int main()
{
    Stub stub;
    stub.set((int(A::*)(int))ADDR(A,foo), ADDR(B, foo_stub));
    A a;
    a.foo(5);
    return 0;
}
```

- **重载函数打桩(实例成员函数)**

```
//for linux, __cdecl
#include<iostream>
#include "stub.h"
using namespace std;
class A{
    int i;
public:
    int foo(int a){
        cout<<"I am A_foo_int"<<endl;
        return 0;
    }
    int foo(double a){
        cout<<"I am A_foo-double"<<endl;
```

```
        return 0;
    }
};

int foo_stub_int(void* obj,int a)
{
    A* o= (A*)obj;
    cout<<"I am foo_stub_int"<< a << endl;
    return 0;
}
int foo_stub_double(void* obj,double a)
{
    A* o= (A*)obj;
    cout<<"I am foo_stub_double"<< a << endl;
    return 0;
}

int main()
{
    Stub stub;
    stub.set((int(A::*)(int))ADDR(A,foo), foo_stub_int);
    stub.set((int(A::*)(double))ADDR(A,foo), foo_stub_double);
    A a;
    a.foo(5);
    a.foo(1.1);
    return 0;
}

//for windows，__thiscall
#include<iostream>
#include "stub.h"
using namespace std;
class A{
    int i;
public:
    int foo(int a){
        cout<<"I am A_foo_int"<<endl;
        return 0;
    }
    int foo(double a){
        cout<<"I am A_foo-double"<<endl;
        return 0;
    }
};
```

```
class B{
        int i;
public:
        int foo_stub_int(int a)
        {
                cout << "I am foo_stub_int" << a << endl;
                return 0;
        }
        int foo_stub_double(double a)
        {
                cout << "I am foo_stub_double" << a << endl;
                return 0;
        }
};
int main()
{
    Stub stub;
    stub.set((int(A::*)(int))ADDR(A,foo), ADDR(B, foo_stub_int));
    stub.set((int(A::*)(double))ADDR(A,foo), ADDR(B, foo_stub_double));
    A a;
    a.foo(5);
    a.foo(1.1);
    return 0;
}
```

- **虚函数打桩**

```
//for linux
#include<iostream>
#include "stub.h"
using namespace std;
class A{
public:
    virtual int foo(int a){
        cout<<"I am A_foo"<<endl;
        return 0;
    }
};

int foo_stub(void* obj,int a)
{
    A* o= (A*)obj;
    cout<<"I am foo_stub"<<endl;
    return 0;
```

```
}


int main()
{
    typedef int (*fptr)(A*,int);
    fptr A_foo = (fptr)(&A::foo);   //获取虚函数地址
    Stub stub;
    stub.set(A_foo, foo_stub);
    A a;
    a.foo();
    return 0;
}

//for windows x86(32 位)
#include<iostream>
#include "stub.h"
using namespace std;
class A {
public:
        virtual int foo(int a) {
                cout << "I am A_foo" << endl;
                return 0;
        }
};

class B {
public:
        int foo_stub(int a)
        {
                cout << "I am foo_stub" << endl;
                return 0;
        }
};



int main()
{
        unsigned long addr;
        _asm {mov eax, A::foo}
        _asm {mov addr, eax}
        Stub stub;
        stub.set(addr, ADDR(B, foo_stub));
```

```
        A a;
        a.foo(1);
        return 0;
}
//for windows x64(64 位)，VS 编译器不支持内嵌汇编。可以把汇编代码独立成一个文件。
```

- **内联函数打桩**

```
//for linux
//添加-fno-inline 编译选项，禁止内联，能获取到函数地址，打桩参考上面。
//for windows
//添加/Ob0 禁用内联展开。
```

- **第三方库私有成员函数打桩**

```
//for linux
//被测代码添加-fno-access-private 编译选项，禁用访问权限控制，成员函数都为公有的
//无源码的动态库或静态库无法自己编译，需要特殊技巧获取函数地址
#include<iostream>
#include "stub.h"
#include "addr_pri.h"    //只适用 c++11
using namespace std;
class A{
    int a;
    int foo(int x){
        cout<<"I am A_foo "<< a << endl;
        return 0;
    }
    static int b;
    static int bar(int x){
        cout<<"I am A_bar "<< b << endl;
        return 0;
    }
};


ACCESS_PRIVATE_FIELD(A, int, a);
ACCESS_PRIVATE_FUN(A, int(int), foo);
ACCESS_PRIVATE_STATIC_FIELD(A, int, b);
ACCESS_PRIVATE_STATIC_FUN(A, int(int), bar);

int foo_stub(void* obj, int x)
{
    A* o= (A*)obj;
```

```
    cout<<"I am foo_stub"<<endl;
    return 0;
}
int bar_stub(int x)
{
    cout<<"I am bar_stub"<<endl;
    return 0;
}
int main()
{
    A a;

    auto &A_a = access_private_field::Aa(a);
    auto &A_b = access_private_static_field::A::Ab();
    A_a = 1;
    A_b = 10;

    call_private_fun::Afoo(a,1);
    call_private_static_fun::A::Abar(1);

    auto A_foo= get_private_fun::Afoo();
    auto A_bar = get_private_static_fun::A::Abar();

    Stub stub;
    stub.set(A_foo, foo_stub);
    stub.set(A_bar, bar_stub);

    call_private_fun::Afoo(a,1);
    call_private_static_fun::A::Abar(1);
    return 0;
}
//for windows，__thiscall
#include<iostream>
#include "stub.h"
using namespace std;
class A{
    int a;
    int foo(int x){
        cout<<"I am A_foo "<< a << endl;
        return 0;
    }
    static int b;
    static int bar(int x){
        cout<<"I am A_bar "<< b << endl;
```

```
        return 0;
    }
};


ACCESS_PRIVATE_FIELD(A, int, a);
ACCESS_PRIVATE_FUN(A, int(int), foo);
ACCESS_PRIVATE_STATIC_FIELD(A, int, b);
ACCESS_PRIVATE_STATIC_FUN(A, int(int), bar);
class B {
public:
        int foo_stub(int x)
        {
                cout << "I am foo_stub" << endl;
                return 0;
        }
};
int bar_stub(int x)
{
    cout<<"I am bar_stub"<<endl;
    return 0;
}


int main()
{
    A a;

    auto &A_a = access_private_field::Aa(a);
    auto &A_b = access_private_static_field::A::Ab();
    A_a = 1;
    A_b = 10;

    call_private_fun::Afoo(a,1);
    call_private_static_fun::A::Abar(1);

    auto A_foo= get_private_fun::Afoo();
    auto A_bar = get_private_static_fun::A::Abar();

    Stub stub;
    stub.set(A_foo, ADDR(B,foo_stub));
    stub.set(A_bar, bar_stub);

    call_private_fun::Afoo(a,1);
```

```
    call_private_static_fun::A::Abar(1);
    return 0;
}
```

- **static 函数打桩**

```
//for linux
#include <iostream>
#include <string>
#include <stdio.h>

#include "addr.h"
#include "stub.h"

// g++ -g test_addr.cpp -std=c++11 -I../ -o test_addr

static int test_test()
{
    printf("test_test\n");
    return 0;
}

static int xxx_stub()
{
    std::cout << "xxx_stub" << std::endl;
    return 0;
}
int main(int argc, char **argv)
{
    std::string res;
    get_exe_pathname(res);
    std::cout << res << std::endl;
    unsigned long base_addr;
    get_lib_pathname_and_baseaddr("libc-2.17.so", res, base_addr);
    std::cout << res << base_addr << std::endl;
    std::map<std::string,ELFIO::Elf64_Addr> result;
    get_weak_func_addr(res, "^puts$", result);

    test_test();


    Stub stub;
    std::map<std::string,ELFIO::Elf64_Addr>::iterator it;
    for (it=result.begin(); it!=result.end(); ++it)
    {
```

```
        stub.set(it->second + base_addr ,xxx_stub);
        std::cout << it->first << " => " << it->second +
base_addr<<std::endl;
    }

    test_test();

    return 0;
}
```

说明：

- 只适用 linux，和 windows 的 x86、x64 架构

不可以打桩的情况：

- 不可以对 exit 函数打桩，编译器做了特殊优化
- 不可以对纯虚函数打桩，纯虚函数没有地址