链滴

关于 C/C++ 的 Thunk 技术

作者：Private

原文链接：https://ld246.com/article/1545390403832

来源网站：链滴

Thunk

从一次偶然的发现说起

环境： Windows 10 ， VS 2015

有一次编写一个动态库(.dll)，很简单很标准的导出了几个API

像这样：

DLL.h

```
#pragma once
#define C_STYLE extern "C"
#ifdef __EXPORT__
#define API __declspec(dllexport)
#else
#define API __declspec(dllimport)
#endif // __EXPORT__

C_STYLE int API DLL_Func1();

C_STYLE int API DLL_Func2();
```

DLL.cpp

```
#define __EXPORT__
#include "DLL.h"

int DLL_Func1()
{
    return 1;
}

int DLL_Func2()
{
    return 2;
}
```

编译链接后得到 DLL.dll，然后使用dumpbin /EXPORTS DLL.dll 查看导出函数

结果是这样的：

```
Dump of file DLL.dll


File Type: DLL

  Section contains the following exports for DLL.dll

    00000000 characteristics
    5C1CC40C time date stamp Fri Dec 21 18:44:28 2018
        0.00 version
           1 ordinal base
```

```
        2 number of functions
        2 number of names

   ordinal hint RVA      name

         1    0 00011041 DLL_Func1 = @ILT+60(_DLL_Func1)
         2    1 00011046 DLL_Func2 = @ILT+65(_DLL_Func2)
```

不知道读者有没有发现什么问题?

细心的读者可能注意到了

```
1    0 00011041 DLL_Func1 = @ILT+60(_DLL_Func1)
```

_DLL_Func1 这个符号，看起来没问题，又很有问题。
没问题是因为 它就是我们导出的第一个函数名。
很有问题是因为，它和我们导出的函数名不完全一致，多了一个下划线。
为什么会有这个下划线呢？当时心中只是有这个疑问，并没有深入研究。

再次发现奇怪现象

直到我再次发现一个奇怪现象的时候，我才决定一探究竟
代码如下：

```
void Test()
{
}
typedef void(*Func)();
int main(int argc,const char * argv[],const char *envp[])
{
   Func pFunc = &Test;
   return 0;
}
```

代码很简单，就是用一个函数指针保存了函数 Test 的地址
奇怪现象如下图：

这个现象描述起来就是：函数指针的值，和函数的实际地址**不一致**

然后我就懵逼了，连刚入门的c语言新手都知道，函数指针就是函数的首地址。那为什么这个理论和看到的现象不一样呢?

函数指针的本质

为了弄清楚为什么会出现上面的奇怪现象，我决定通过反汇编看看编译器都做了些什么处理。

修改代码，通过函数名调用函数和通过函数指针调用函数

```
void Test(){
}

typedef void(*Func)();

int main(int argc,const char * argv[],const char *envp[])
{
   Func pFunc = &Test;
   Test();
   (*pFunc)();
   return 0;
}
```

设置断点，查看反汇编。

```
int main(int argc,const char * argv[]
{
    Func pFunc = &Test;
    Test(); pFunc 0x00f512ee {Test.exe!Test(void)}
    (*pFunc)();
    return 0;
}
```

```
    Test();
00F520C5  call        Test (0F512EEh)
    (*pFunc)();
00F520CA  mov         esi,esp
00F520CC  call        dword ptr [pFunc]
00F520CF  cmp         esi,esp   pFunc 0x00f512ee {Test.exe!Test(void)}
```

通过反汇编可以看到，不管是通过函数名调用还是通过函数指针调用

对应的汇编指令都是 **call xxxx** ，**xxxx**代表某一个地址。

对于两种不同的调用方式，这个地址是一样的，都是函数指针的值

也就是图中的 **0x00F512EE**

PS.两次截图，函数指针不一样，是因为是两个不同的进程

接下来继续查看反汇编，看看 call 命令之后都执行了什么指令

```
Test:
00F512EE  jmp         Test (0F51B90h)
```

F11之后，反汇编结果如图

jmp 0F51B90h

jmp指令为无条件跳转，直到这里，我们基本上可以大概知道了函数调用的过程了

call 调用某个地址，然后 jmp 跳转到另一个地址去执行
那么 jmp 跳转的这个地址，是不是就是函数的实际地址呢?

```
void Test(){
}
```
Test 0x00f51b90 {Test.exe!Test(void)}

```
void Test(){
00F51B90  push        ebp     已用时间 <= 1ms
00F51B91  mov         ebp,esp
00F51B93  sub         esp,0C0h
00F51B99  push        ebx
00F51B9A  push        esi
00F51B9B  push        edi
00F51B9C  lea         edi,[ebp-0C0h]
00F51BA2  mov         ecx,30h
00F51BA7  mov         eax,0CCCCCCCCh
00F51BAC  rep stos    dword ptr es:[edi]
}
```
果然，jmp 跳转的地址，和函数实际地址一样
第二张图是 jmp xxx F11 之后的反汇编结果，更加证明了上面的结论

总结:
函数调用或者函数调用，其实并没有直接调用函数，而是通过一次跳转（jmp）之后才执行真正的函数
这也就解释了为什么函数指针的值和函数实际地址不一致

Thunk技术的实现

上面说到的 jmp xxxx 这一段程序，被微软称之为 Thunk ，翻译为 形实转换程序
所谓的**形**,我个人理解的就是 call xxxx 中的xxxx这个地址
对应的，**实**就是 jmp xxxx 中的xxxx这个地址
本帖先不讨论为什么要使用Thunk技术，只讨论Thunk的实现

CPU执行流程

为了搞清楚Thunk的实现原理，需要先弄清楚cpu是如何执行汇编代码的
cpu执行汇编指令的大概流程如下:

1. 读取 EIP 地址中的指令（EIP是指令寄存器，存放下一条要执行的命令地址）
2. EIP 增加 读取的指令的大小，即 指向下一条指令
3. 执行读取的指令
4. 返回第 1 步

所以，当执行到 jmp xxxx 时（第 2 步 与第 3 步 之间），此时 EIP 已经指向 jmp xxxx 的下一条指令
例如， jmp xxxx 指令在内存中的地址为 0x10
当执行到 jmp xxxx 时，此时 EIP = 0x15

因为 jmp xxxx 这条指令本身占 5 个字节，其中，jmp 指令占1字节，xxxx 相对地址占 4 字节（x64 是 4 字节）

将 jmp xxxx 实体化

将 jmp xxxx 看成一整个Thunk对象

Thunk对象的地址(this)就是jmp xxxx这条指令的地址

执行到Thunk时，下一条指令地址EIP = (int)this + sizeof(Thunk) = (int)this + 5

那么有一个问题，我们如何保证让 jmp 跳转的地址，就是我们需要跳转的实际地址呢?

jmp跳转，是相对跳转，相对于 EIP 跳转

也就是说，jmp指令执行之后，会修改 EIP 的值，修改之后的值 就是 EIP = EIP + xxxx

上面已经说到 执行 jmp 之前 **EIP = (int)this + sizeof(Thunk) = (int)this + 5**

那么执行 jmp 之后 **EIP = EIP + xxxx = (int)this + sizeof(Thunk) = (int)this + 5 + xxxx**

执行完jmp执行后，此时 EIP 的值就是我们需要跳转的实际地址

即：

要跳转的实际地址 = EIP(原) + xxxx = (int)this + sizeof(Thunk) = (int)this + 5 + xxxx

移项得到：

xxxx = 实际地址 - ((int)this + 5 + addr)

也就是说，只要保证 xxxx 这个跳转的相对地址，满足上述公式，就可以达到我们预期的需求

Thunk 实现

第一步，设计数据结构

我们已经知道 jmp xxxx 这条指令占 5 字节，我们需要设计出一个占5字节的结构。

但是在一般情况下，类成员都是按4字节对齐的，为了使我们设计的结构占 5 字节，我们需要修改类员的对齐方式。

通过 **#pragma pack(push,1)**可以强制编译器，使数据按字节边界对齐

```
#pragma pack(push,1) //强制编译器，使数据按字节边界对齐
class Thunk
{
public:
    char JMP;
    int JMP_RA;//跳转的地址 相对于寄存器 IP 的相对地址，x86 x64 都是四字节
}
#pragma pack(pop)//撤销数据按字节对齐，数据按双字对齐的主要目的是优化执行速度
```

因为 成员 JMP 其实就是 jmp 指令对应的 机器码 0xE9，是不会变的

要跳转的相对地址才是关键，所以我们接下来需要提供设置相对跳转地址的成员函数

```
const void * SetRealAddress(const void * addr)
{
    char * old = (char *)GetRealAddress();
        JMP_RA = (int)((char *)addr - (char *)this - sizeof(Thunk));
    return old;
```

```
}
```

通过上面推导出来的公式，这个函数很容易

有了 Setter,自然要有 Getter

```
//获取Thunk跳转的实际地址
const void * GetRealAddress() const
{
    return (void *)((char *)this + sizeof(Thunk) + this->JMP_RA);
}
```

然后我们需要提供这个类的构造函数，

第一种构造函数，通过任意的实际地址创建 Thunk 对象

实际上就是调用了**SetRealAddress**

```
//通过需要跳转的实际地址创建 Thunk 对象
Thunk(void * addr) :JMP((char)0xE9)
{
    SetRealAddress(addr);
}
```

第二种就是通过函数指针创建 Thunk 对象

上面已经提到了，call xxxx 中的 xxxx 其实就是 jmp xxxx 这条指令的地址

将jmp xxxx看成一个 Thunk 对象

而 call xxxx 中的 xxxx 就是函数指针的值，所以我们可以理解为 函数指针的值，就是一个Thunk 对的地址

```
//通过函数指针创建 Thunk 对象
template<class RET, class ...Args>
Thunk(RET(*func)(Args...))
{
    Thunk * that = (Thunk *)func;
    this->JMP = that->JMP;
    SetRealAddress(that->GetRealAddress());
}
```

成员函数指针也是类似，只不过在强制类型转换时会编译失败

这是需要借助 union

```
//通过成员函数指针创建 Thunk 对象
template<class RET, class THIS, class ...Args>
Thunk(RET(THIS::*func)(Args...))
{
    union
    {
        RET(THIS::*func)(Args...);
        Thunk * pThunk;
    } u;//成员函数指针直接转Thunk指针编译不通过，所以通过联合来转换
    u.func = func;

    Thunk * that = u.pThunk;
```

```
    this->JMP = that->JMP;
    SetRealAddress(that->GetRealAddress());
}
```

验证

接下来就可以验证代码的准确性了

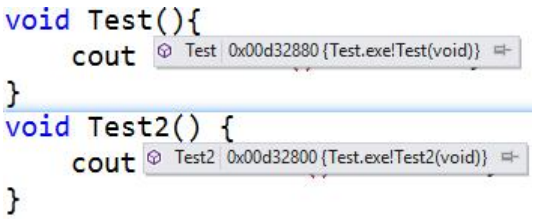
```
#include "Thunk.h"
#include <iostream>
using namespace std;
void Test(){
    cout << "Test()" << endl;
}
void Test2() {
    cout << "Test2()" << endl;
}
typedef void(*Func)();
int main(int argc,const char * argv[],const char *envp[])
{
    Func pFunc = &Test;
    cout << "pFunc : " << pFunc << endl;

    Thunk thunk1 = &Test;
    Thunk thunk2 = &Test2;
    Thunk * pThunk = (Thunk *)&Test;
    cout << "thunk1.GetRealAddress() : " << thunk1.GetRealAddress() << endl;
    cout << "thunk2.GetRealAddress() : " << thunk2.GetRealAddress() << endl;
    cout << "pThunk->GetRealAddress() : " << pThunk->GetRealAddress() << endl;

    thunk1.SetRealAddress(thunk2.GetRealAddress());
    cout << "thunk1.GetRealAddress() : " << thunk1.GetRealAddress() << endl;

    return 0;
}
```

结果如下：



最后贴上完整的代码供读者参考

Thunk.h

```
#pragma once
```

```
#pragma pack(push,1) //强制编译器，使数据按字节边界对齐
class Thunk
{
public:
    char JMP;
    int JMP_RA;//跳转的地址 相对于寄存器 IP 的相对地址，x86 x64 都是四字节
public:
    //通过需要跳转的实际地址创建 Thunk 对象
    Thunk(void * addr) :JMP((char)0xE9)
    {
        SetRealAddress(addr);
    }

    //通过函数指针创建 Thunk 对象
    template<class RET, class ...Args>
    Thunk(RET(*func)(Args...))
    {
        Thunk * that = (Thunk *)func;
        this->JMP = that->JMP;
        SetRealAddress(that->GetRealAddress());
    }

    //通过成员函数指针创建 Thunk 对象
    template<class RET, class THIS, class ...Args>
    Thunk(RET(THIS::*func)(Args...))
    {
        union
        {
            RET(THIS::*func)(Args...);
            Thunk * pThunk;
        } u;//成员函数指针直接转Thunk指针编译不通过，所以通过联合来转换
        u.func = func;

        Thunk * that = u.pThunk;
        this->JMP = that->JMP;
        SetRealAddress(that->GetRealAddress());
    }

    //获取Thunk跳转的实际地址
    const void * GetRealAddress() const
    {
        return (void *)((char *)this + sizeof(Thunk) + this->JMP_RA);
    }

    const void * SetRealAddress(const void * addr)
    {
        char * old = (char *)GetRealAddress();

        JMP_RA = (int)((char *)addr - (char *)this - sizeof(Thunk));

        return old;
    }
};
#pragma pack(pop)//撤销数据按字节对齐，数据按双字对齐的主要目的是优化执行速度
```

Thunk 的小应用

修改函数的实际地址

```
#include "Thunk.h"

#include <Windows.h>

void Func1()
{
    MessageBox(NULL, "Func1", "", 0);
}

void Func2()
{
    MessageBox(NULL, "Func2", "", 0);
}

typedef void(*FUNC)();

int main(int argc,const char * argv[],const char *envp[])
{
    FUNC f1 = &Func1;
    FUNC f2 = &Func2;
    Thunk * pThunk1 = (Thunk *)f1;
    Thunk * pThunk2 = (Thunk *)f2;

    Func1();

    //更改内存保护属性
    DWORD old;
    VirtualProtect(pThunk1, sizeof(Thunk), PAGE_EXECUTE_READWRITE, &old);

    //修改Thunk的跳转地址
    pThunk1->SetRealAddress(pThunk2->GetRealAddress());

    //回复原来的保护属性
    VirtualProtect(pThunk1, sizeof(Thunk), old, nullptr);


    Func1();


    return 0;
}
```

效果是:

第一次调用Func1，弹出 Func1 的MessageBox .

第二次调用 Func1 ,会弹出 Func2 的MessageBox.

说明修改函数实际地址成功

结束语

至此，Thunk 的实现原理已经结束，本帖主要是带读者入门，让读者对Thunk有一个大概的了解，至微软为什么要使用这个技术，笔者目前还不是很清楚，读者可自行查阅资料，最后，感谢读者坚持读，有问题欢迎留言。