看雪学院课程《汇编语言详解与二进制漏洞初阶》笔记

原创

猫咪钓鱼 于 2020-01-14 23:20:35 发布 489 收藏 7

分类专栏： 网络空间安全

本文链接：https://blog.csdn.net/weixin_43655282/article/details/103981157

网络空间安全 专栏收录该内容

16 篇文章 2 订阅

订阅专栏

前言和声明

安全工程师这条路任重道远。如今国际形势复杂，网络战一旦爆发，安全势力弱的一方很快会处于竞争的下风，加上国家的安全人才缺口过大，我辈则应当肩挑重担，为祖国安全尽一份力。

本博客是博主在学习看雪学院《汇编语言详解与二进制漏洞初阶》课程的笔记，笔记中大部分是对课程作业的解答，少部分是博主补充的知识。
如若转载，请声明出处。谢谢。
与广大有心朝安全方向前进的网友们共勉之。

另注：本博客会持续更新到课程学习完毕。

汇编语言部分

学习汇编语言的意义

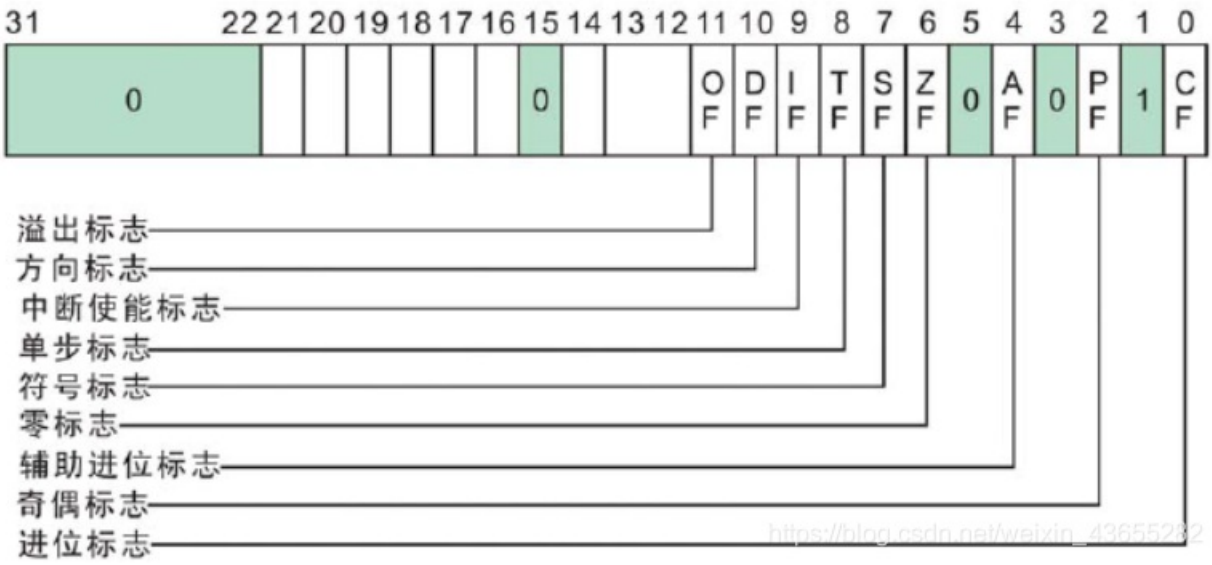
- 开发遇到bug时调试更加便利。在用高级语言进行调试时往往会遇到一些非常难以调试出来的bug，此时程序员若能将高级语言代码反汇编成汇编代码来进行调试则可以提高调试效率。
- 逆向分析时的代码阅读。逆向分析时，所分析的软件对于分析者来说其实是一个“黑盒”，此时若不懂汇编语言则将寸步难行。
- 对某些特殊技术的使用。用高级语言编写的程序往往占较大的内存，当程序员编写shellcode、壳等代码时，要尽量压缩其大小，此时汇编语言则大有用处，因为使用汇编语言编写程序时可以精确到字节。

  > shellcode: 能在一个完整程序中的任意位置运行的代码。

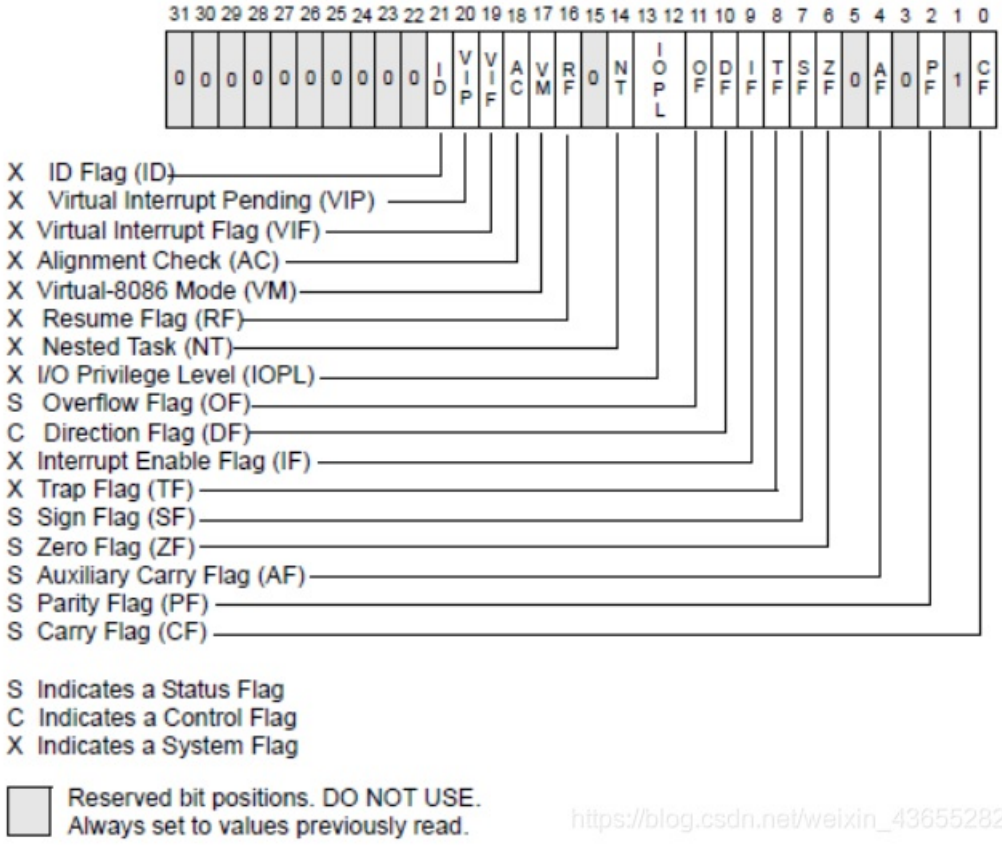
EFLAGS寄存器

EFLAGS寄存器包含了独立的二进制位，用于控制CPU操作，或是反应一些CPU操作的结果。有些指令可以测试和控制这些单独的处理器标志位。

- 中文图



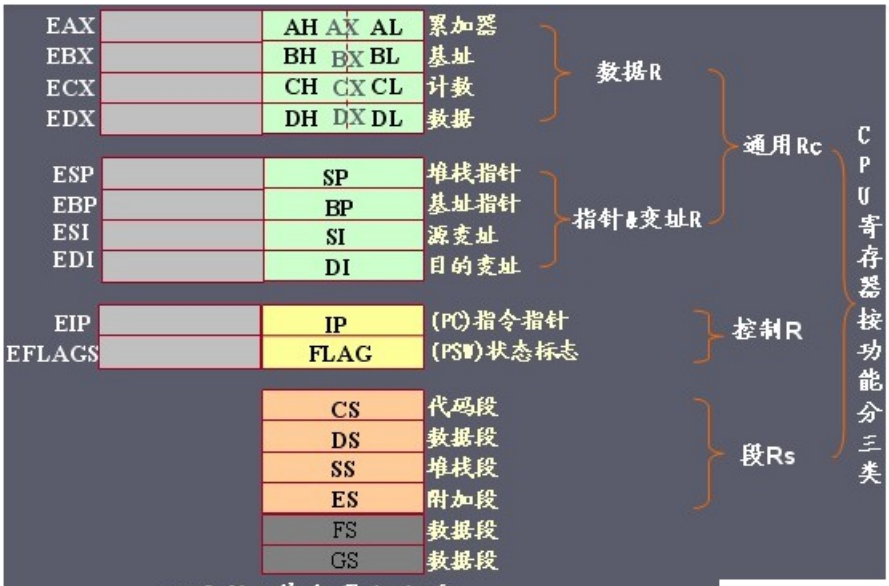
- 英文图



这篇文章对EFLAGS寄存器的讲解非常详细，请点击这里□

各寄存器的名称及其主要用途

寄存器	累加器	基地址寄存器	计数器	数据寄存器	源变址寄存器	源目标变址寄存器	基地址指针	栈顶指针
代号	EAX	EBX	ECX	EDX	ESI	EDI	EBP	ESP
主要用途	算术运算、存储中间结果、函数返回值	基地址指针	循环计数、移位操作计数、重复操作计数	乘除运算、存储中间结果	存储指针、字符串指令的源操作数指针	存储指针、字符串指令的目的操作数指针	基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部	栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。



16位段+偏移寻址组合

段	偏 移	主要用途
CS	IP	指令地址
DS	BX，SI，DI，Disp8，Disp16	数据地址
SS	SP，BP	堆栈地址
ES	DI	串操作目标地址

32位段+偏移寻址组合

段	偏 移	主要用途
CS	EIP	指令地址
DS	EAX，EBX，ECX，EDX，ESI，EDI，Disp8，Disp32	数据地址
SS	ESP，EBP	堆栈地址
ES	EDI	串操作目标地址
FS	无默认	一般地址
GS	无默认	一般地址

内存寻址范围

给出几个计算例子。

1、某计算机字长32位，存储容量8MB。按字编址，其寻址范围为(0～2M-1) 计算步骤：8MB字节=8*1024*1024*8位。所以8MB/32位=2M.

2、某计算机字长32位，其存储容量为4MB，若按半字编址，它的寻址范围是（0-2M-1）计算步骤：若按半字就是16位了4MB=4*1024*1024*8位，所以4MB/16 = 2M；

3、字长为32位.存储器容量为64KB.按字编址的寻址范围是多少计算步骤：64K字节=64＊1024＊8位. 所以64KB/32位=(64＊1024＊8)/32=16＊1024=16K 故寻址范围为: 0-16K-1

4、某机字长32位，存储容量1MB，若按字编址，它的寻址范围是什么？

解释：容量1M=2*1024*1024 位 一个字长是32 位
所以，寻址范围是二者相除=256K

内存的五种表现形式

内存地址的5种表达形式

1、 [立即数] (全局变量)

eg 读取内存的值:

```
mov eax,dword ptr ds:[0x34ddfe11]
```

向内存中写入数据:

```
mov dword ptr ds:[0x34ddfe11],eax
```

2、 [reg] reg代表寄存器，只能是8个32位通用寄存器中的任意一个。 (局部变量)

eg 读取内存的值:

```
mov ecx,0x14eed2
mov eax,dword ptr ds:[ecx]
```

向内存中写入数据:

```
mov edx,0x14eed2
mov dword ptr ds:[edx],0x12345678
```

3、 [reg+立即数]

eg 读取内存的值:

```
mov ecx,0x14eed2
mov eax,dword ptr ds:[ecx+0x3]
```

向内存中写入数据:

```
mov edx,0x14eed2
mov dword ptr ds:[edx+0xe],0x12345678
```

4、 [reg+reg*{1,2,4,8}] (数组)

eg 读取内存的值:

```
mov eax,0x14eed2
mov ecx,2
mov edx,dword ptr ds:[eax+ecx*4]
```

向内存中写入数据:

```
mov eax,0x14eed2
mov ecx,2
mov dword ptr ds:[eax+ecx*4],0x12345678
```

4、 [reg+reg*{1,2,4,8}+立即数]

eg 读取内存的值:

```
mov eax,0x14eed2
mov ecx,2
mov edx,dword ptr ds:[eax+ecx*4+4]
```

向内存中写入数据:

```
mov eax,0x14eed2
mov ecx,2
mov dword ptr ds:[eax+ecx*4+4],0x12345678
```


声明：图片来源于

若想加深对内存表现形式的理解，请点击这里

数据存储模式（大小端模式）

下面以unsigned int value = 0x12345678为例，分别看看在两种字节序下其存储情况，我们可以用unsigned char buf[4]来表示value

- **Big-Endian**: 低地址存放高位，如下：
  高地址
  ---
  buf[3] (0x78) – 低位
  buf[2] (0x56)
  buf[1] (0x34)
  buf[0] (0x12) – 高位

  低地址
- **Little-Endian**: 低地址存放低位，如下：
  高地址
  ---
  buf[3] (0x12) – 高位
  buf[2] (0x34)
  buf[1] (0x56)
  buf[0] (0x78) – 低位

  低地址

	内存地址	小端模式存放内容	大端模式存放内容
低地址	0x4000	0x78	0x12
↓	0x4001	0x56	0x34
↓	0x4002	0x34	0x56
高地址	0x4003	0x12	0x78

使用VS编写汇编程序

- VS环境配置

> 1.建一个空项目
>
> 2.选中项目右键 “生成自定义”,选择MASM生成规则
>
> 3.新建一个.asm后缀的新文件
>
> 4.选中项目右键->属性->链接器->系统->子系统选<控制台(SUBSYSTEM:CONSOLE)>
>
> 5.选中项目右键->属性->链接器->高级->入口点 填 "main"

本课程中教学用VS2015，具体操作请看□。
若使用VS2019则与VS2015操作有所不同，VS2019可以直接在搜索框中搜索“生成自定义”、“属性”等关键词，操作也很方便。

- 一个汇编程序的大致结构

```
.586
.MODEL flat, stdcall
includelib user32.lib
includelib kernel32.lib
ExitProcess PROTO, dwExitCode : DWORD
MessageBoxA PROTO hWnd : DWORD, lpText : BYTE, lpCaption : BYTE, uType : DWORD
.data
Number DWORD 0
text db "shellcode",0
.code
main proc
       mov eax,5
       mov ebx,6
       add eax,ebx
       add eax,Number
       push 0
       push offset text
       push offset text
       push 0
       call MessageBoxA
       sub esp,16
       call ExitProcess
main ENDP
END main
```


汇编语言中的数学运算

在汇编语言中，有 **加 减 乘 除 自增 自减** 六种运算。

- 加法

加法

> 加法指令 ADD(Addition)
>
> 格式: ADD OPRD1,OPRD2
>
> 功能: 两数相加
>
> 加法指令运算的结果对CF、SF、OF、PF、ZF、AF都会有影响

加法指令运算的结果对CF、SF、OF、PF、ZF、AF都会有影响

不允许OPRD1与OPRD2同时为存储器

带进位加法指令 ADC(Addition Carry)

格式: ADC OPRD1,OPRD2

功能: OPRD1 = OPRD1+OPRD2+CF

- 减法

减法

减法指令SUB(SUBtract)

格式: SUB OPRD1,OPRD2

功能: 两个操作数的相减,即从OPRD1中减去OPRD2,其结果放在OPDR1中. 指令的类型及对标志位的影响与ADD指令相同,注意立即数不能用于目的操作数,两个存储器操作数之间不能直接相减.操作数可为8位或16位的无符号数或带符号数.

带借位减法指令 SBB(SuBtraction with Borrow)

格式: SBB OPRD1,OPRD2

功能: 进行两个操作数的相减再减去CF进位标志位,即从OPRD1 = OPRD1-OPRD2-CF,其结果放在OPDR1中.

- 乘法

乘法

无符号数乘法指令 MUL(MULtiply)

格式: MUL OPRD

带符号数乘法指令 IMUL(Integer MULtiply)

格式: IMUL OPRD

功能: 乘法操作.

OPRD为通用寄存器或存储器操作数.

本指令影响标志位CF及OF.

- 除法

除法

无符号数除法指令 DIV(DIVision)

格式: DIV OPRD

功能: 实现两个无符号二进制数除法运算.

带符号数除法指定 IDIV(Interger DIVision)

格式: IDIV OPRD

功能: 这实现两个带符号数的二进制除法运算.

比如16bit的被除数，分存在2个8bit寄存器AH:AL，商放在AL，余数在AH

比如32bit的被除数，分存在2个16bit寄存器DX:AX，商放在AX，余数在DX

比如64bit的被除数，分存在2个32bit寄存器EDX:EAX，商放在EAX，余数在EDX
比如128bit的被除数，分存在2个64bit寄存器RDX:RAX，商放在RAX，余数在RDX

- 自增

自增

加1指令 INC(INCrement by 1)
格式: INC OPRD
功能: OPRD = OPRD+1

- 自减

自减

减一指令 DEC(Decrement by 1)
格式: DEC OPRD
功能: OPRD = OPRD-1

堆栈操作

理论

- 什么是栈？

  学过数据结构一般都知道，后进先出数据结构者为栈。

  千万要牢记，栈的基本特点是：**后进先出**。

  就跟你把羽毛球装在球筒里面一样，最后放进去的肯定是第一个拿出来的。

  1.栈是一个后进先出的存储区域，位于堆栈段中，SS段寄存器描述的就是堆栈段的段地址。

  2.栈的数据出口位于栈顶，也就是esp寄存器所指向的位置。

  3.栈顶是低位，也就是地址较小的一侧，由ebp寄存器指向的栈底，并不会改变。

- 栈操作指令

  最基本的无非就是 PUSH 和 POP 。

  PUSH：压栈指令，32位汇编首先ESP-4，留出一个空间，然后把要压入栈中的内容压入。

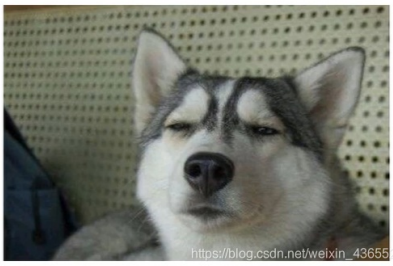
  POP：出栈指令，32位汇编首先将栈顶的数据弹出给指定的目标，然后ESP+4，清掉空间。

- 栈的作用

  1.存储少量数据

  2.保存寄存器环境

  3.传递参数



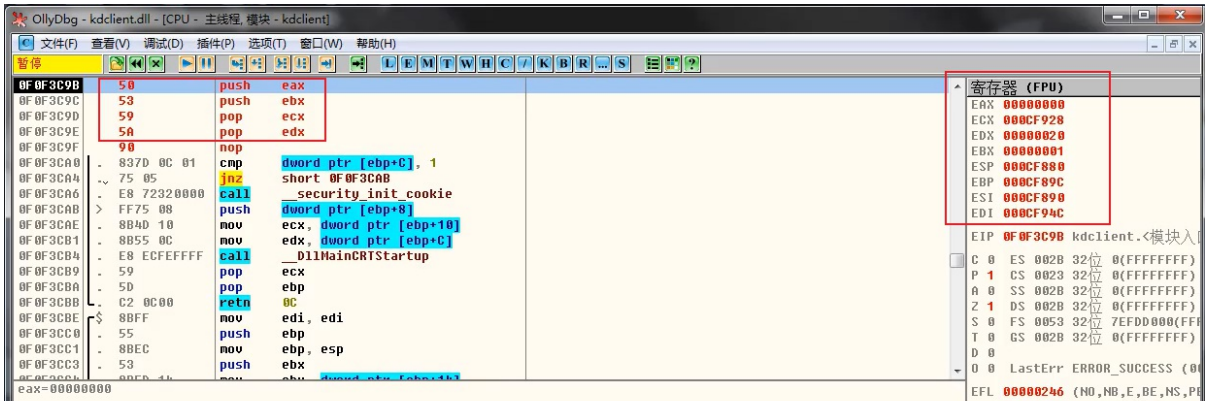
  栈的空间有限，所以只能存储少量数据；

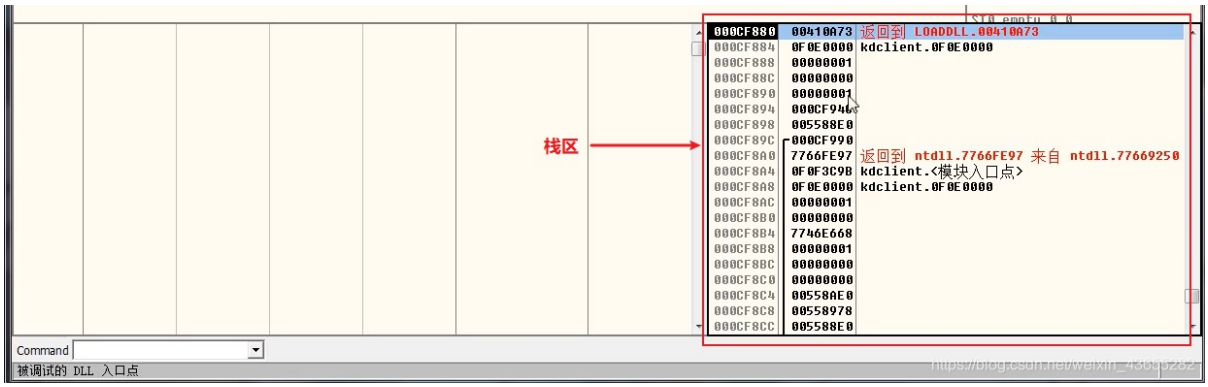
  所谓保存寄存器环境，也就是说我们对寄存器进行了一些操作，但我想在操作完成之后恢复程序的全部功能，此时寄存器的内容应该也要跟原来一样，故称之为保存寄存器环境。

实践

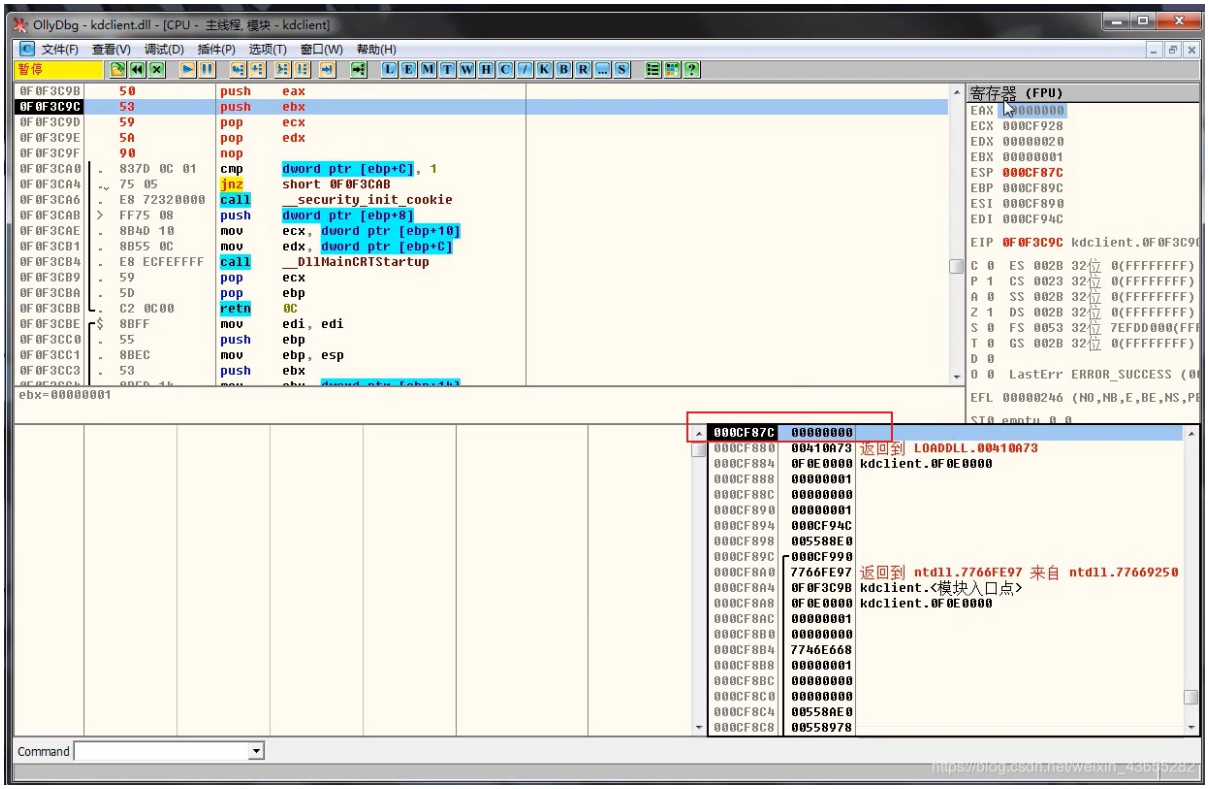
使用OllyDbg实现栈操作。

将任意PE文件拖入OllyDbg，如图修改前四行代码：

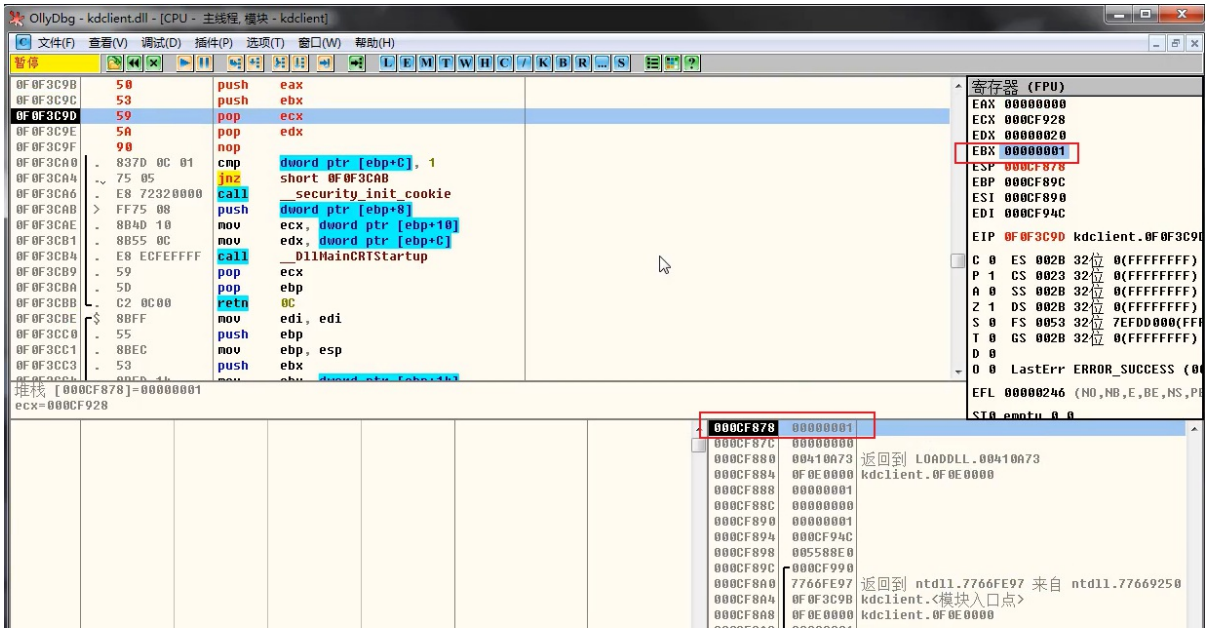


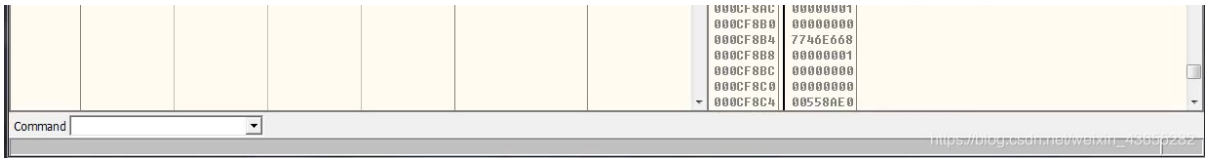


按F8执行，将**eax**的内容push进栈

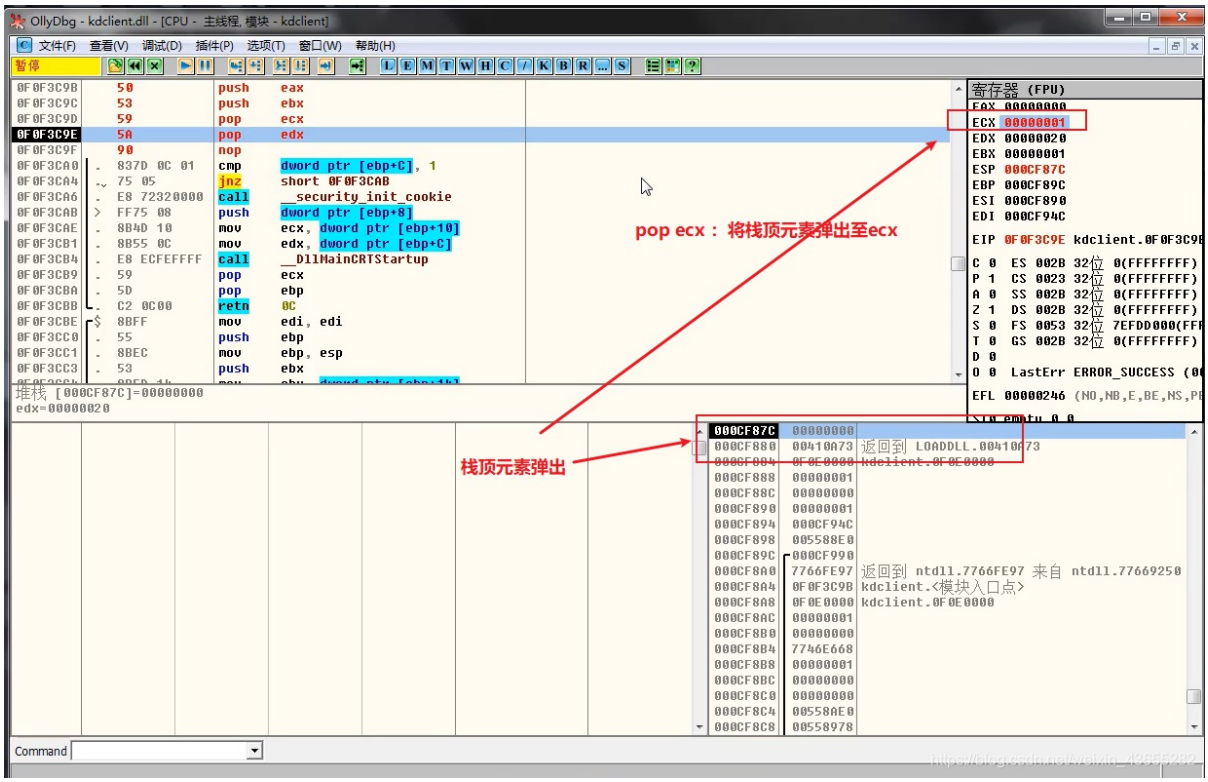


再按F8，将**ebx**的内容push进栈

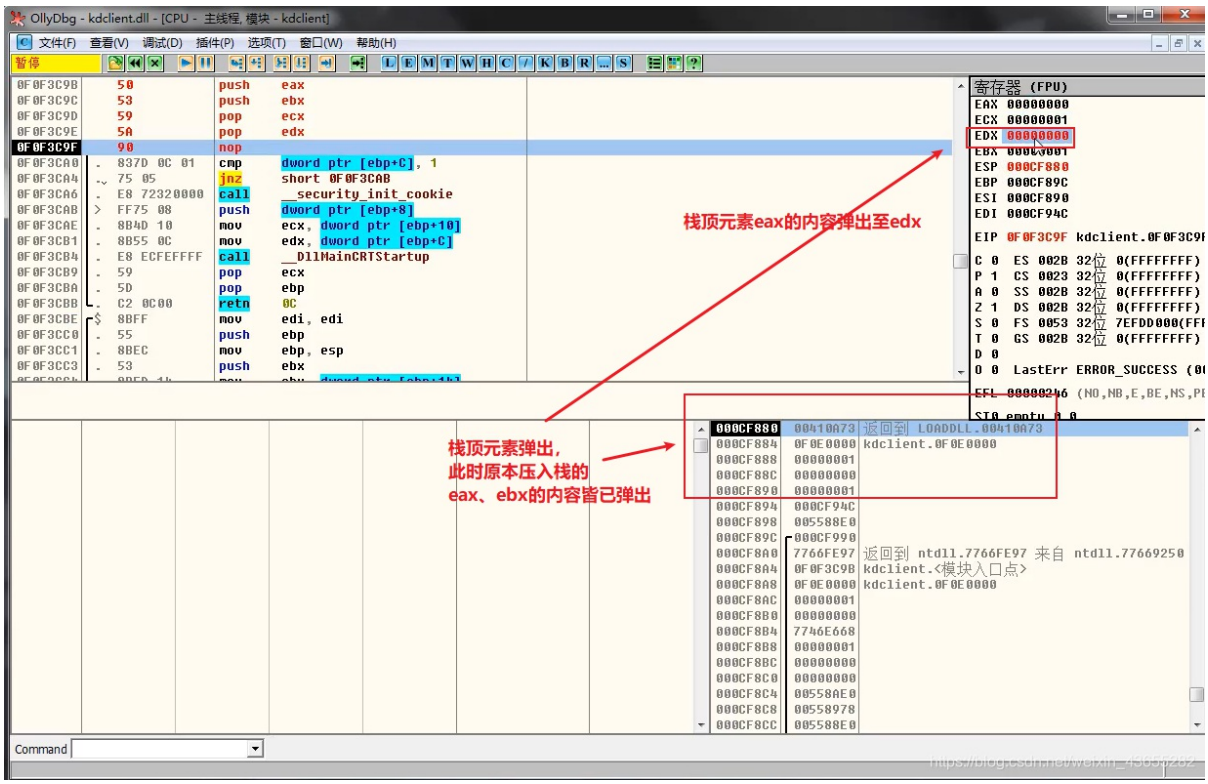




再按F8，将栈顶元素弹出到**ecx**



再按F8，栈顶元素弹出至**edx**



再强调一次，栈的基本特点是：**后进先出**。

数据移动指令

- **MOV** 指令

> 数据传送指令 MOV
> 格式: MOV OPRD1,OPRD2
> 功能: 本指令将一个源操作数送到目的操作数中,即OPRD1<--OPRD2.
> 说明:
> OPRD1 为目的操作数,可以是寄存器、存储器、累加器.
> OPRD2 为源操作数,可以是寄存器、存储器、累加器和立即数.

- **LEA** 指令

> 有效地址传送指令 LEA
> 格式: LEA OPRD1,OPRD2
> 功能: 将源操作数给出的有效地址传送到指定的的寄存器中.
> OPRD1必须是寄存器

- **XCHG** 指令

> 数据交换指令 XCHG
> 格式: XCHG OPRD1,OPRD2 其中的OPRD1为目的操作数,OPRD2为源操作数
> 功能: 将两个操作数相互交换位置,该指令把源操作数OPRD2与目的操数OPRD1交换.

作业：用VS和OllyDbg复现课程中代码，熟练掌握上述三个指令的使用方法。

比较指令

- **CMP 指令**

CMP指令

> 比效指令 CMP(CoMPare)
> 格式: CMP OPRD1,OPRD2
> 功能: 对两数进行相减,进行比较.

- **TEST 指令**

TEST

> 测试指令 TEST
> 格式: TEST OPRD1,OPRD2
> 功能: 其中OPRD1、OPRD2的含义同AND指令一样,也是对两个操作数进行按位的'与'运算, ---- 唯一不同之处是不将'与'的结果送目的操作数,即本指令对两个操作数 的内容均不进行修改,仅是在逻辑与操作后,对标志位重新置位.

JCC条件转移指令

JCC 表

JCC指令	中文含义	英文原意	检查符号位
JZ/JE	若为0则跳转；若相等则跳转	jump if zero;jump if equal	ZF=1
JNZ/JNE	若不为0则跳转；若不相等则跳转	jump if not zero;jump if not equal	ZF=0
JS	若为负则跳转	jump if sign	SF=1
JNS	若为正则跳转	jump if not sign	SF=0
JP/JPE	若1出现次数为偶数则跳转	jump if Parity (Even)	PF=1
JNP/JPO	若1出现次数为奇数则跳转	jump if not parity (odd)	PF=0
JO	若溢出则跳转	jump if overflow	OF=1
JNO	若无溢出则跳转	jump if not overflow	OF=0
JC/JB/JNAE	若进位则跳转；若低于则跳转；若不高于等于则跳转	jump if carry;jump if below;jump if not above equal	CF=1
JNC/JNB/JAE	若无进位则跳转；若不低于则跳转；若高于等于则跳转；	jump if not carry;jump if not below;jump if above equal	CF=0
JBE/JNA	若低于等于则跳转；若不高于则跳转	jump if below equal;jump if not above	ZF=1或CF=1
JNBE/JA	若不低于等于则跳转；若高于则跳转	jump if not below equal;jump if above	ZF=0或CF=0
JL/JNGE	若小于则跳转；若不大于等于则跳转	jump if less;jump if not greater equal	SF != OF
JNL/JGE	若不小于则跳转；若大于等于则跳转；	jump if not less;jump if greater equal	SF = OF
JLE/JNG	若小于等于则跳转；若不大于则跳转	jump if less equal;jump if not greater	ZF != OF 或 ZF=1
JNLE/JG	若不小于等于则跳转；若大于则跳转	jump if not less equal;jump if greater	SF=0F 且 ZF=0

常用的JCC指令

> JMP：无条件跳转
> JZ/JE：ZF = 1 等于0或相等跳转
> JNZ/JNE：ZF = 0 不等于0或者不相等跳转
> JBE/JNA：CF=1/ZF=1 低于等于/不高于跳转
> JNBE/JA：CF=0/ZF=0 不低于等于/高于跳转
> JL/JNGE：SF！=OF 小于/不大于等于跳转
> JNL/JGE：SF=OF 不小于/大于等于跳转

JMP 指令
本例子可供总结 jmp 指令的使用方法。

```
.586
.MODEL flat, stdcall
.code
main proc
    mov eax,0
jmpflag:
    add eax,1
    jmp jmpflag
main ENDP
END main
```

这段汇编代码给寄存器eax赋初值为0，
然后进入jmpflag段并执行*add eax,1*语句，
对eax加1之后再跳转(jmp)回jmpflag段起始语句*add eax,1*，
又对eax加1。
这样子就形成了一个无限循环。

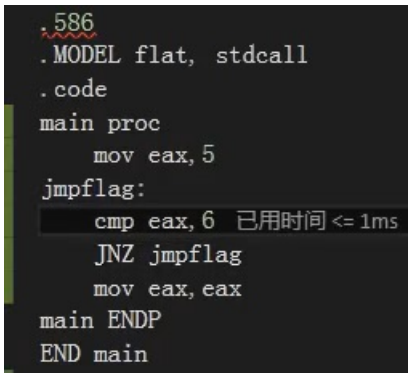
JZ/JE 指令
本例子可供总结 jz/je 指令的使用方法。

```
.586
.MODEL flat, stdcall
.code
main proc
    mov eax,5
jmpflag:
    cmp eax,6
    JZ jmpflag
    mov eax,eax  已用时间 <= 1ms
main ENDP
END main
```

代码解释：
给eax赋初值为5，
之后使用 cmp 指令(上文有提到其作用)与6比较，
不等，
则**不执行** jz 指令。

JNE 指令

本例子可供总结 jne 指令的使用方法。



代码解释：

给eax赋初值为5，

之后使用 cmp 指令(上文有提到其作用)与6比较，

不等，

则**执行** jz 指令。

串操作指令

汇编中的串应理解为字节串、字串等，应该属于数组的范畴，可以进行扫描查找、比较、传送(填充)等操作。

- **MOVS 指令**

> 字符串传送指令 MOVS
> 格式: MOVS OPRD1,OPRD2
> ---- MOVSB
> MOVSW
> 功能: OPRD1<--OPRD2.
> 说明: 1. 其中OPRD2为源串符号地址,OPRD1为目的串符号地址.

- **STOS 指令**

> 字符串存储指令 STOS
> 格式: STOS OPRD
> 功能: 把AL(字节)或AX(字)中的数据存储到DI为目的串地址指针所寻址的存储器单元中去指针DI将根据DF的值进行自动调整.

- **REP 指令**
  重复操作字符串指令。

> 重复前缀的说明
> 格式: REP ;CX<>0 重复执行字符串指令
> ---- REPZ/REPE ;CX<>0 且ZF=1重复执行字符串指令
> REPNZ/REPNE ;CX<>0 且ZF=0重复执行字符串指令
>
> 功能: 在串操作指令前加上重复前缀,可以对字符串进重复处理.由于加上重复前缀后,对应的指令代码是不同的,所以指令的功能便具有重复处理的功能,重复的次数存放在CX寄存器中.

CALL和RETN指令

CALL指令和RETN指令是配合起来写函数的指令。

理论

- **CALL** 指令

call指令

> 过程调用指令 CALL
> 格式: CALL OPRD
> 功能: 过程调用指令
>
> 相当于：
>
> push eip
> jmp OPRD

可以调用任意地址，并且将call指令的下一条指令的地址压入栈中。

- **RETN** 指令

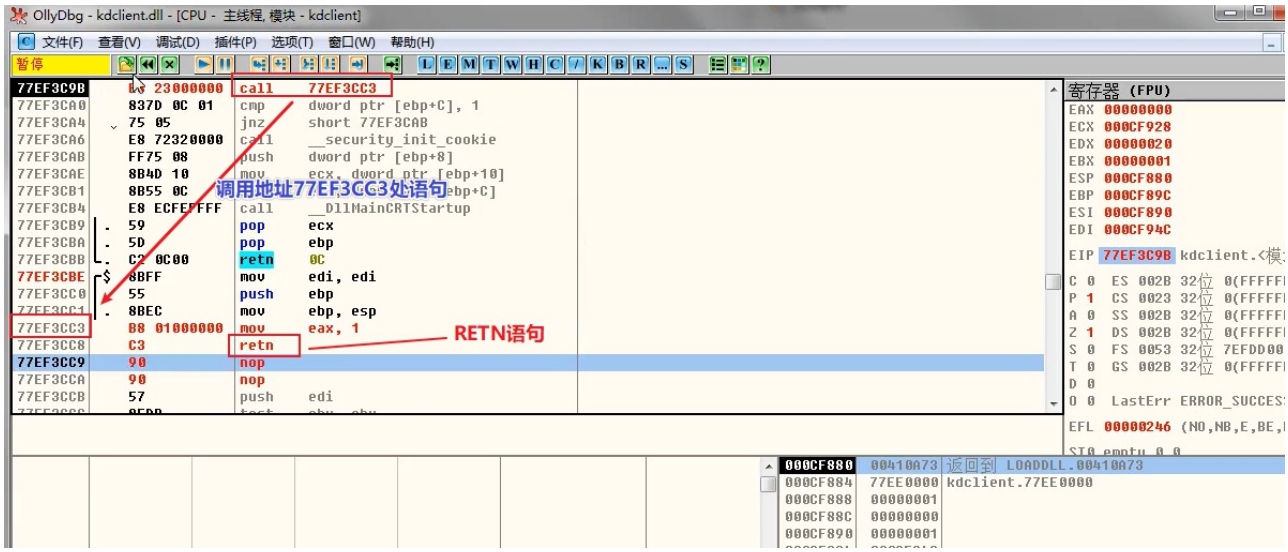
RETN指令

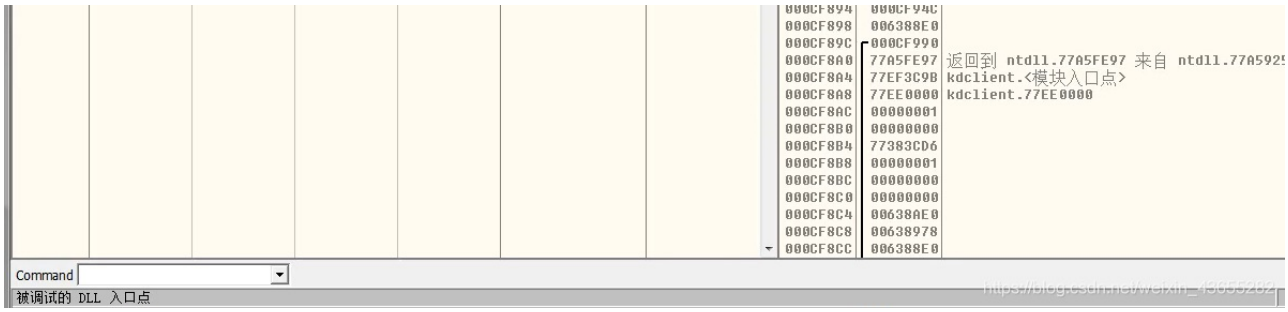
> 返回指令，相当于：
>
> pop eip
> jmp eip

将栈顶元素(该元素是执行call语句时压入的call语句下一条语句的地址)弹出,返回该地址，所以它的功能是使程序执行完call指令继续往下执行。

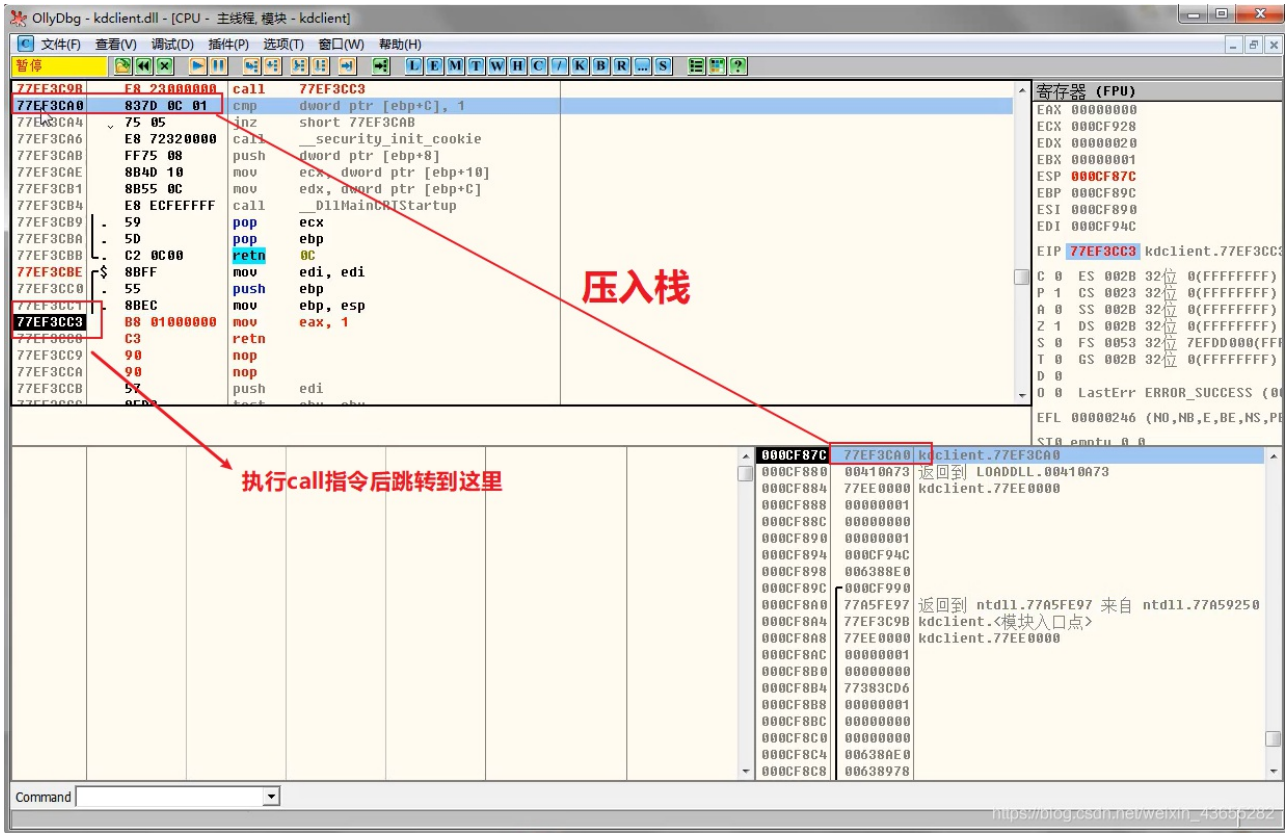
实践

- 修改程序第1条语句为 **call** 指令语句，修改**77EF3CC3**处语句为mov eax，1(修改为其他语句也可以，无所谓的)，修改**77EF3CC8**处语句为 **retn**



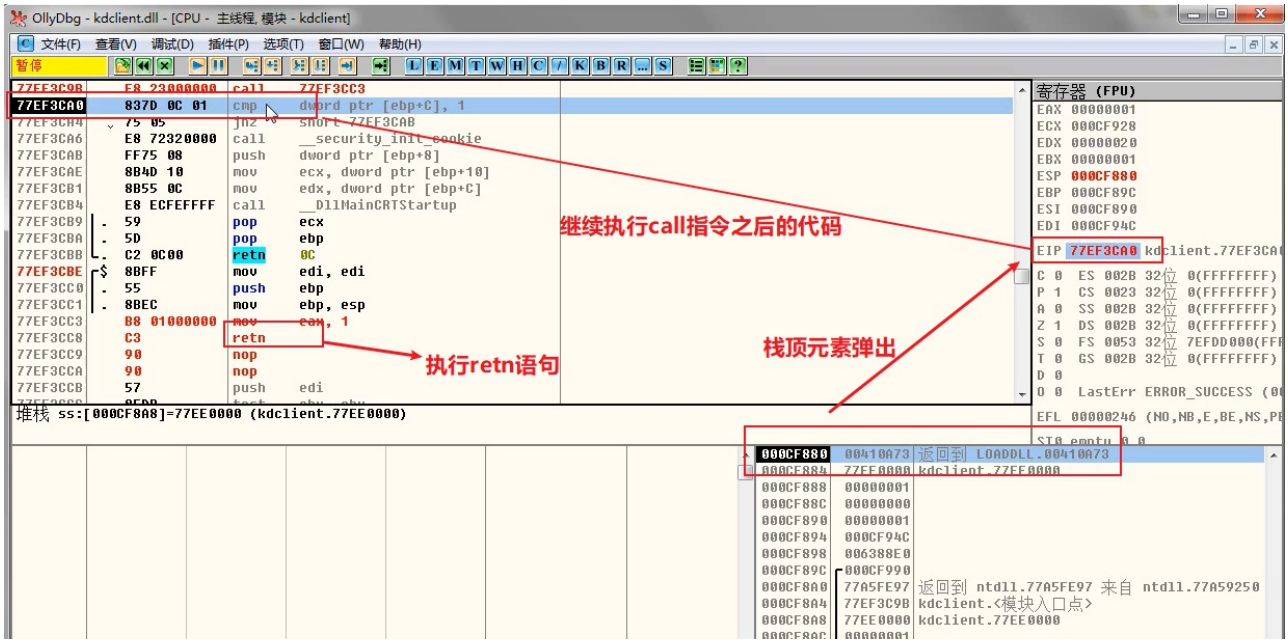


- 执行**call** 指令



是跳转到红色框框那里，不是跳转到箭头的位置。。。

- 执行**retn**指令



右边第一个框框中的77EF3CA0已经弹出了，所以在栈中看不到它。

汇编中的函数

过程调用的方式要根据编译器而定，下面图片中的只是一个例子。

过程调用的方式：

```
function proc
    code
function endp
```

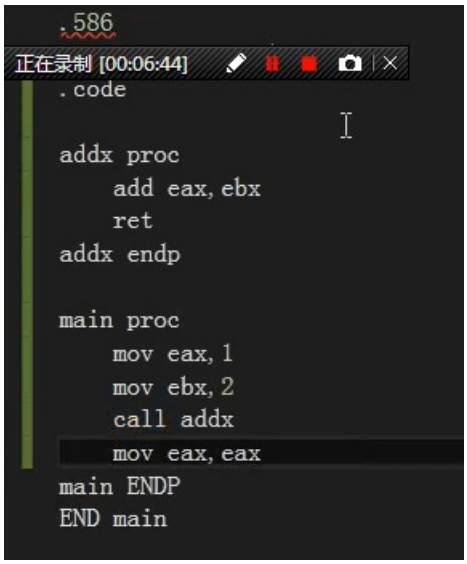
参数传递方式：

1.寄存器传参

2.堆栈传参

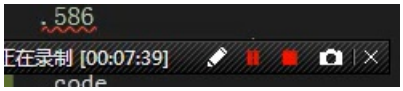
寄存器数量有限，为防止出现寄存器不够用的情况，所以出现了栈传参。

- 寄存器传参



可根据此程序代码动手实践，观察各相关寄存器的内容的变化情况。

- 堆栈传参



```
addx proc
    mov eax,[esp+4]
    mov ebx,[esp+8]
    add eax,ebx
    ret
addx endp

main proc
    push 1
    push 2
    call addx
    mov eax,eax
main ENDP
END main
```

[esp+4]是第二参数，[esp+8]是第一参数。
为什么呢？因为**栈的特点是先进后出，后进先出**!所以越靠近栈底的元素(地址越高)的元素**越先进栈**！
那么为什么要+4和+8呢？因为一个整数占4个字节，所以栈顶元素的首地址应该为esp+4。

- 作业

> ① 熟练掌握过程调用的方式
>
> ② 编写加减乘除四种运算函数

Win32汇编入门

理论

什么是API

> API（Application Programming Interface，应用程序接口）是一些预先定义的函数，或指软件系统不同组成部分衔接的约定。目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无需访问原码，或理解内部工作机制的细节。

操作系统的用户接口API函数包含在Windows系统目录下的动态连接库文件中。Windows API是一套用来控制Windows的各个部件的外观和行为的预先定义的Windows函数。用户的每个动作都会引发一个或几个函数的运行以告诉Windows发生了什么。这在某种程度上很像Windows的天然代码。而其他的语言只是提供一种能自动而且更容易的访问API的方法。当你点击窗体上的一个按钮时，Windows会发送一个消息给窗体，VB获取这个调用并经过分析后生成一个特定事件。

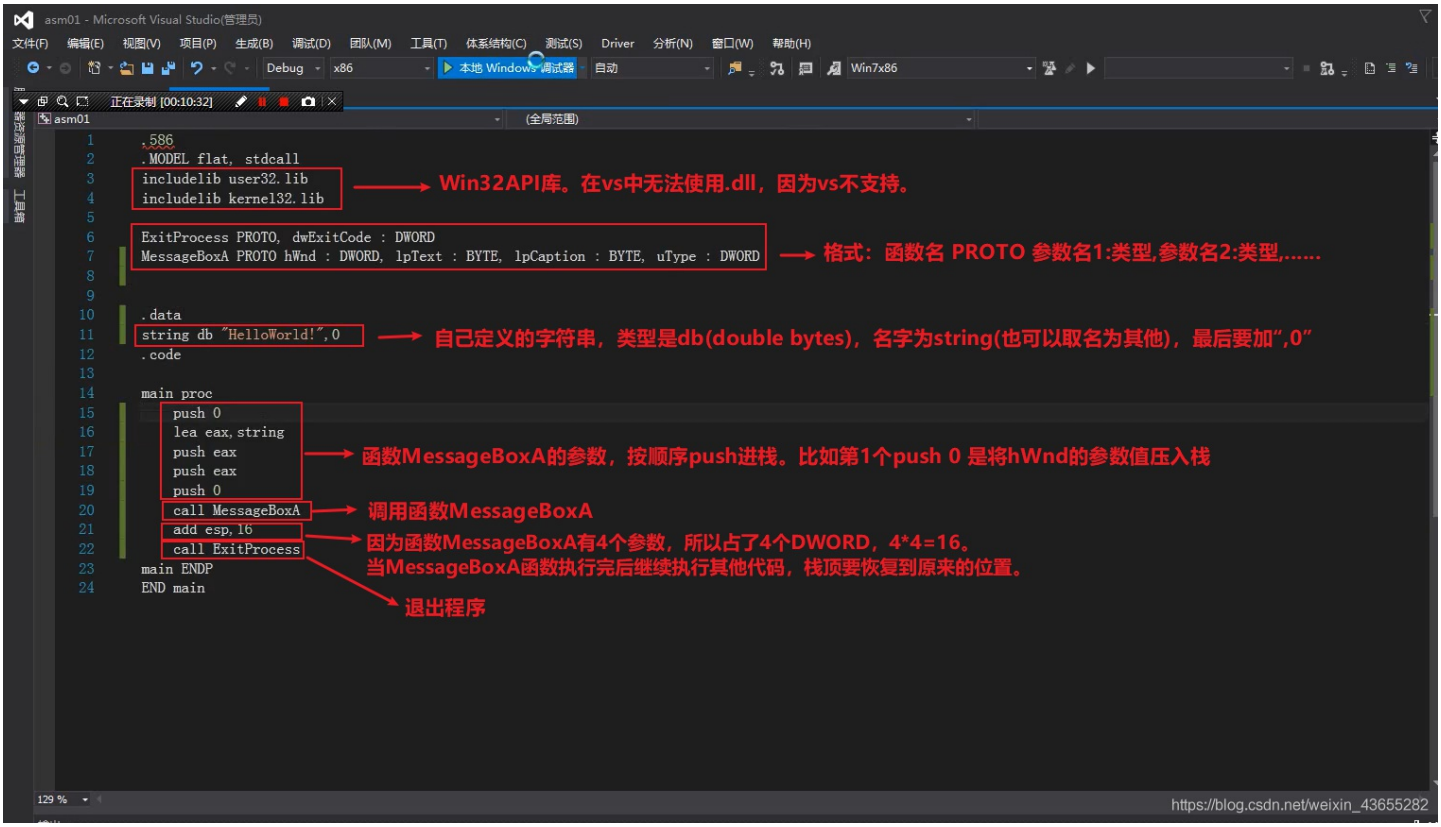
更易理解来说：Windows系统除了协调应用程序的执行、内存的分配、系统资源的管理外，同时他也是一个很大的服务中心。调用这个服务中心的各种服务(每一种服务就是一个函数)可以帮助应用程序达到开启视窗、描绘图形和使用周边设备等目的，由于这些函数服务的对象是应用程序，所以称之为Application Programming Interface，简称API 函数。WIN32 API也就是MicrosoftWindows 32位平台的应用程序编程接口。

凡是在 Windows工作环境底下执行的应用程序，都可以调用Windows API。

实践

注意！！！

更改： 函数MessageBoxA参数部分，**第1个push 0是将uType的参数值压入栈**



作业：

作业

① 熟练掌握API的概念

② 编写一个弹出消息框且正常退出的程序

C语言部分

C语言的优点

1.设计特性

2.高效性

3.可移植性

4.强大的功能和灵活性

5.面向编程人员

C语言概述

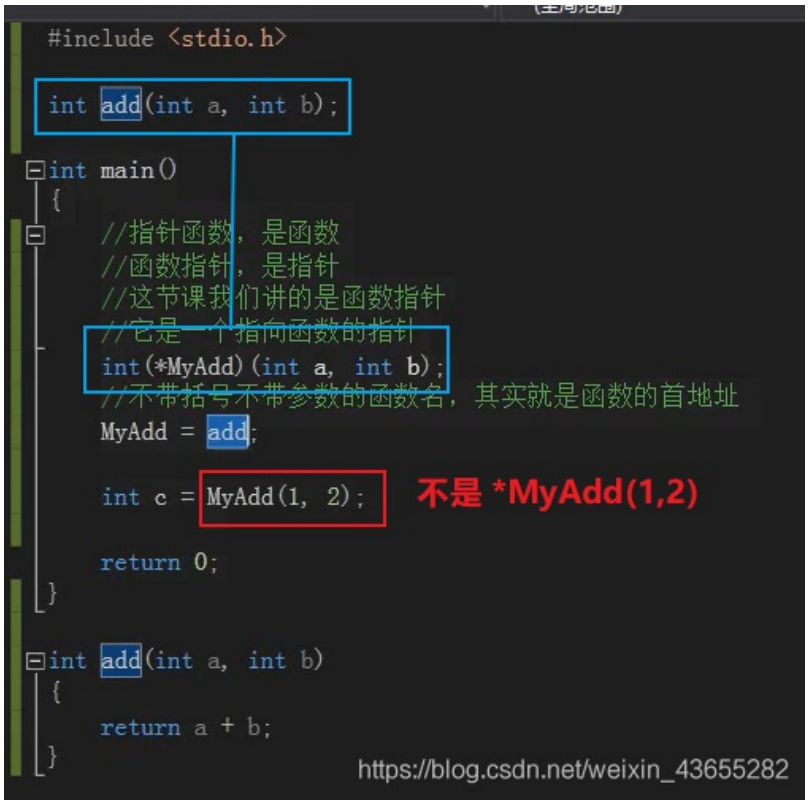
写程序的过程：

> 定义程序目标
> 设计程序
> 编写代码
> 编译
> 运行程序
> 调试程序
> 维护和修改程序

函数指针

即指向函数的指针。

- 代码
  直观展示什么是指向函数的指针。



在上图中，下面两行代码是相等的。
MyAdd是函数add的指针。

```
int add(inta, int b)
int (*MyAdd)(int a, int b)
```

- 反汇编
  从汇编层面认识什么是指向函数的指针。

补充：

`dword ptr [myadd]是什么意思？`

dword 双字 就是四个字节

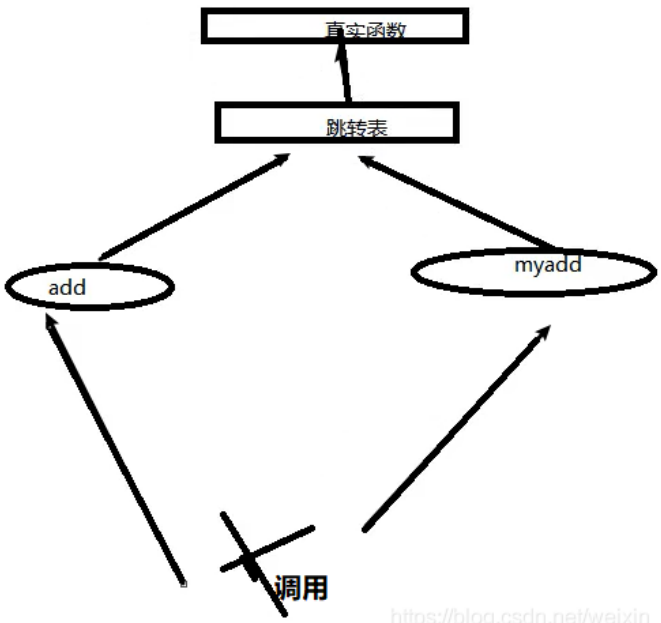
ptr pointer缩写 即指针

[]里的数据是一个地址值，这个地址指向一个双字型数据

比如mov eax, dword ptr [12345678] 把内存地址12345678中的双字型（32位）数据赋给eax

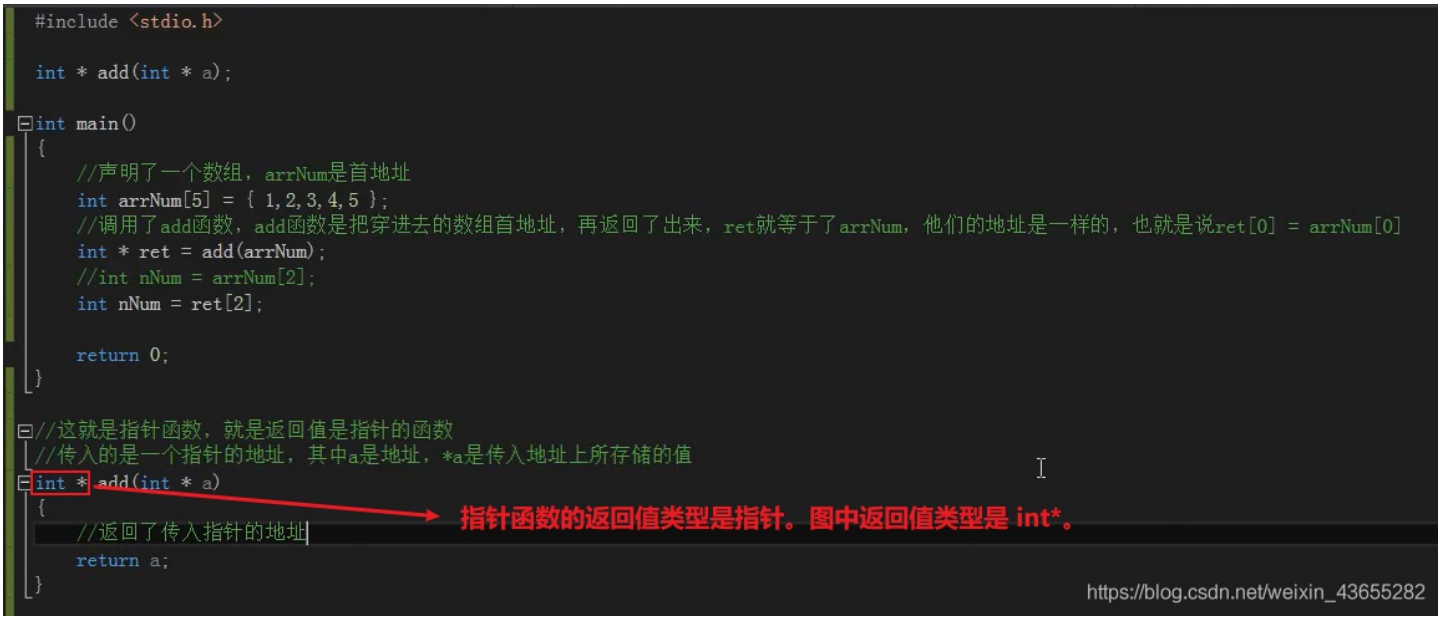
```
    MyAdd = add;
00911D1E  mov         dword ptr [MyAdd],offset  add (0911352h)
dword ptr [myadd]
叫做myadd的四字节指针，指向一段空间地址，这块地址应该存储的是一个函数的首地址，但是因为myadd这个标签是我们自己什么的指针变量名，所以[myadd]是一个空值或者垃圾值

offset  add(0911352h)
是一段跳转代码，在跳转表中

跳转表内容如下：
00911352  jmp         add (0914110h)
跳转表的作用就是跳转到函数的真实首地址
0914110h就是函数的真实首地址

其实就是把__add跳转表的地址给[myadd]

myadd = 00911352（跳转表地址）
add = 00911352（跳转表地址）

0914110h真实函数首地址


    int c = MyAdd(1, 2);
00911D25  mov         esi,esp
00911D27  push        2
00911D29  push        1
00911D2B  call        dword ptr [MyAdd]

call   dword ptr [MyAdd]

push eip
jmp 00911352（跳转表地址）
jmp 0914110h真实函数首地址
执行add或者myadd真实的函数代码
然后返回



00911D2E  add         esp,8
00911D31  cmp         esi,esp
00911D33  call        __RTC_CheckEsp (0911109h)
00911D38  mov         dword ptr [c],eax
```

框框里面的是汇编代码的说明



指针函数

即返回值是指针的函数。



函数分析

"Hello world!"程序分析

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

> 建立新栈时将新栈刷为CCCCCC…h，很多程序的汇编代码中都能看到CCCCCC…h。

建立新栈是程序健壮性的体现。

作业

记住函数分析的内容。

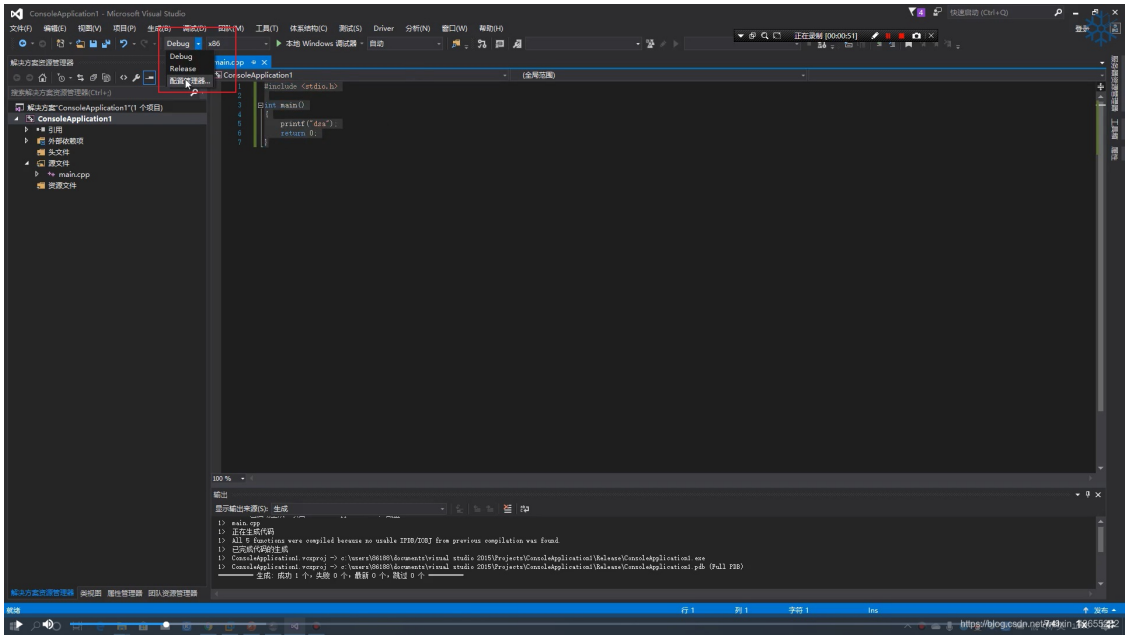
C语言命名规则

```c
#include <stdio.h>


int myGetProcAddress()
{

    return 0;
}

int main()
{
    //最基本的命名规则
    //字母，数字，下划线
    //变量要以字母或者下划线开头，不得使用数字开头

    //匈牙利命名法
    //在变量名前，加上属性，类型

    //下划线分割，忘掉，不怎么使用

    //驼峰命名法
    //大驼峰命名法,所有单词全部首字母大写，例如GetProcAddress
    //小驼峰命名法，除了第一个单词小写外，其他的单词首字母大写，例如myGetProcAddress

    int nProcessFlag;
    return 0;
}
```


逆向入门

寻找main函数

- 程序版本



- Release
  编译器会对程序进行优化，程序占存较小，但调试相对困难。发布程序时一般为此版本。
- Debug
  少量优化，程序占存较大，方便调试。

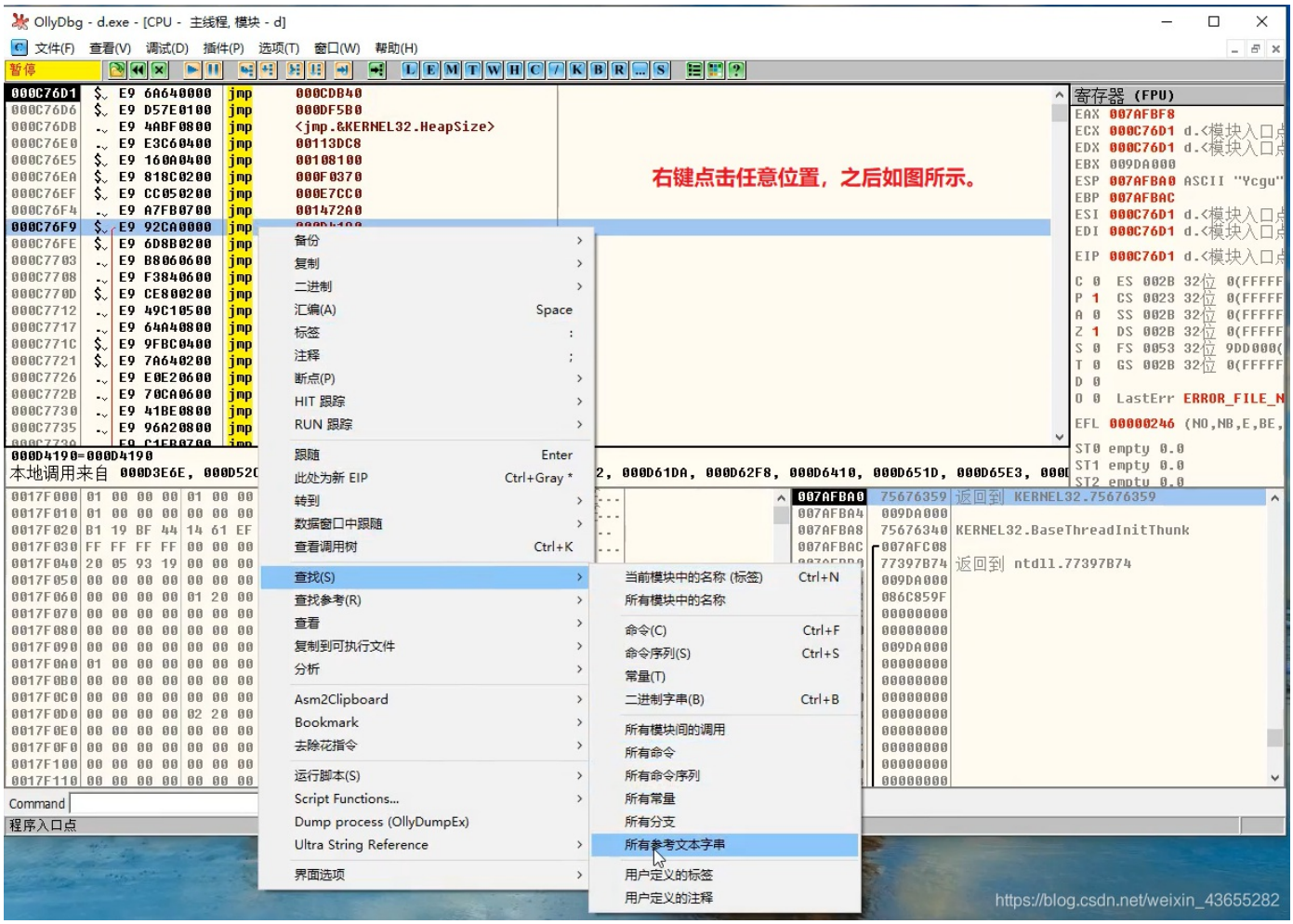
实践

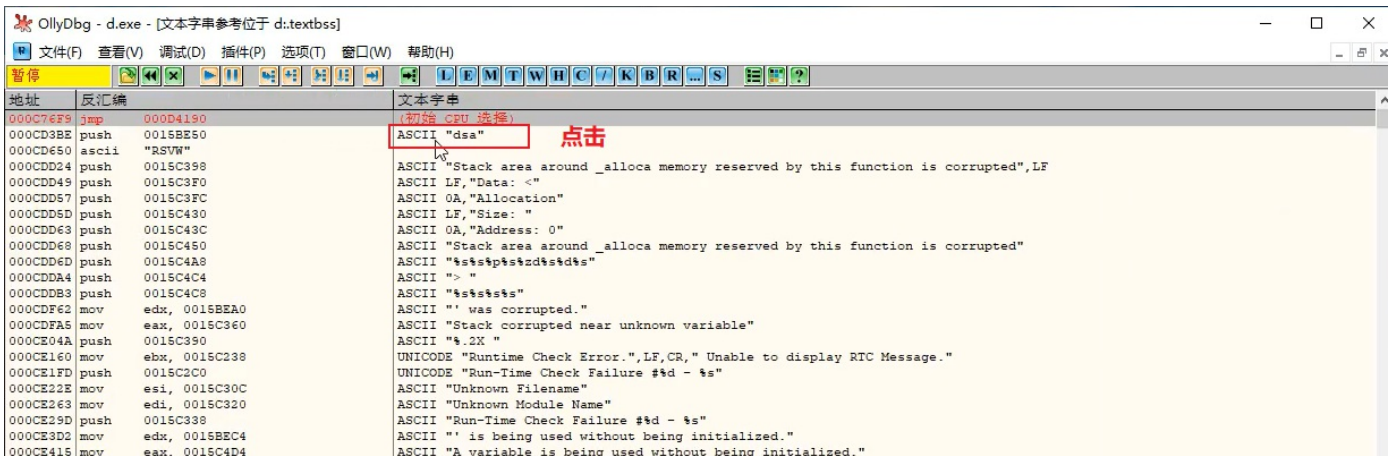
OllyDebug

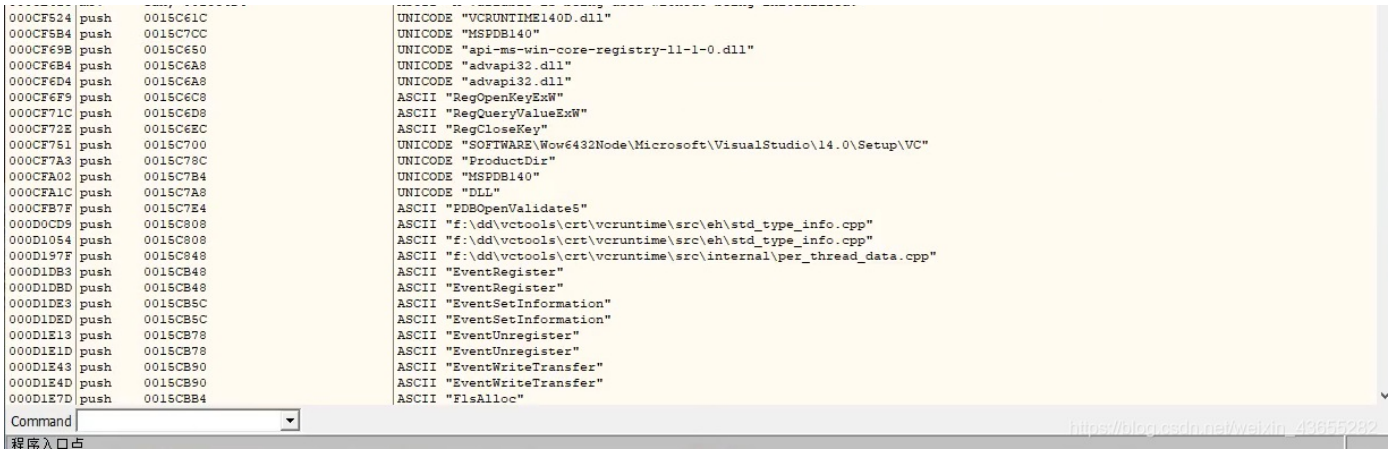
Debug版本程序

```
#include <stdio.h>

int main()
{
 printf("das");
 return 0;
}
```

- 以管理员身份运行OllyDebug





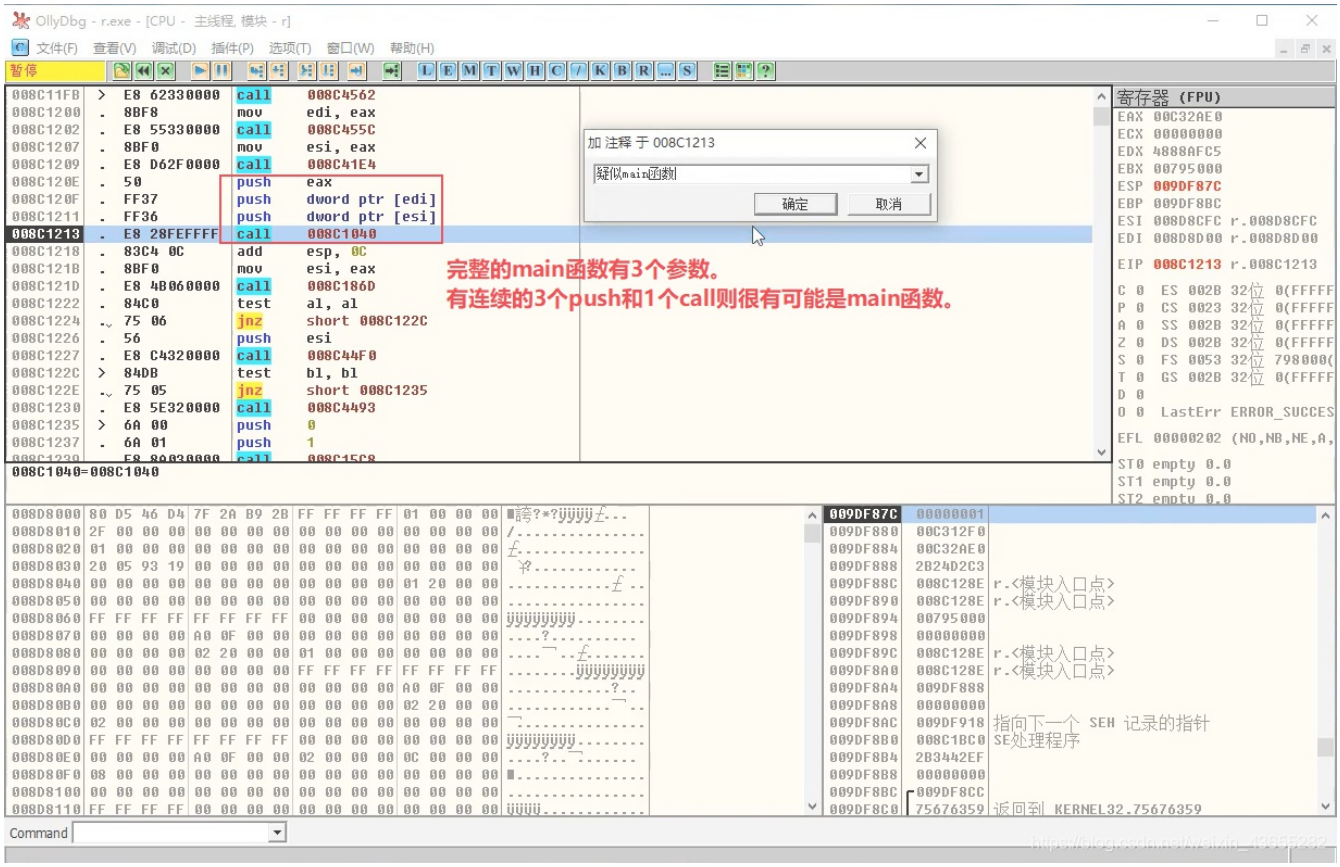


Release版本

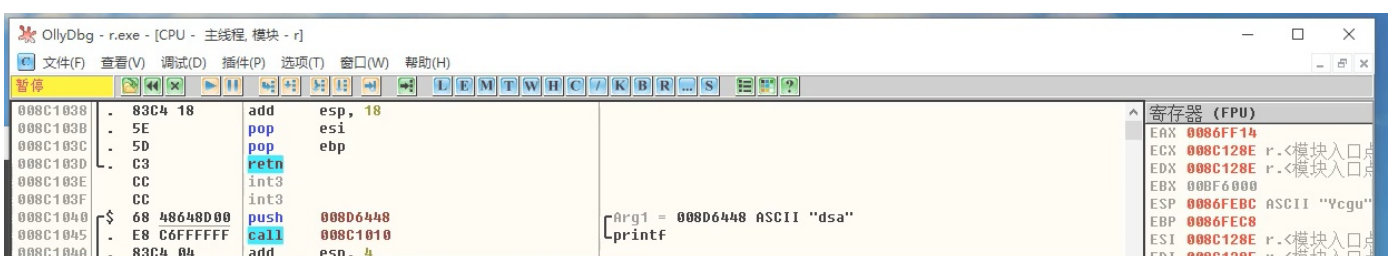
看视频如何操作

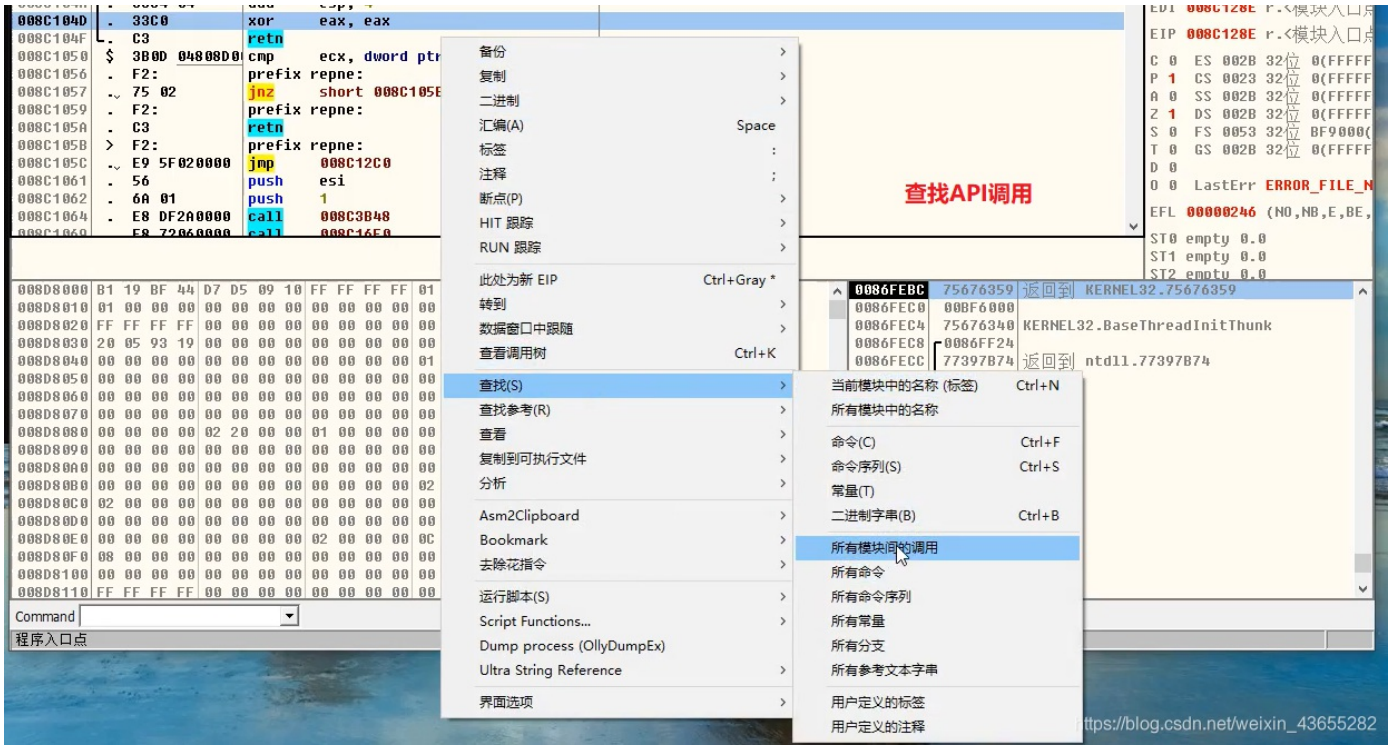
视频 3:14~ □

- OllyDebug
  - 第一种方法



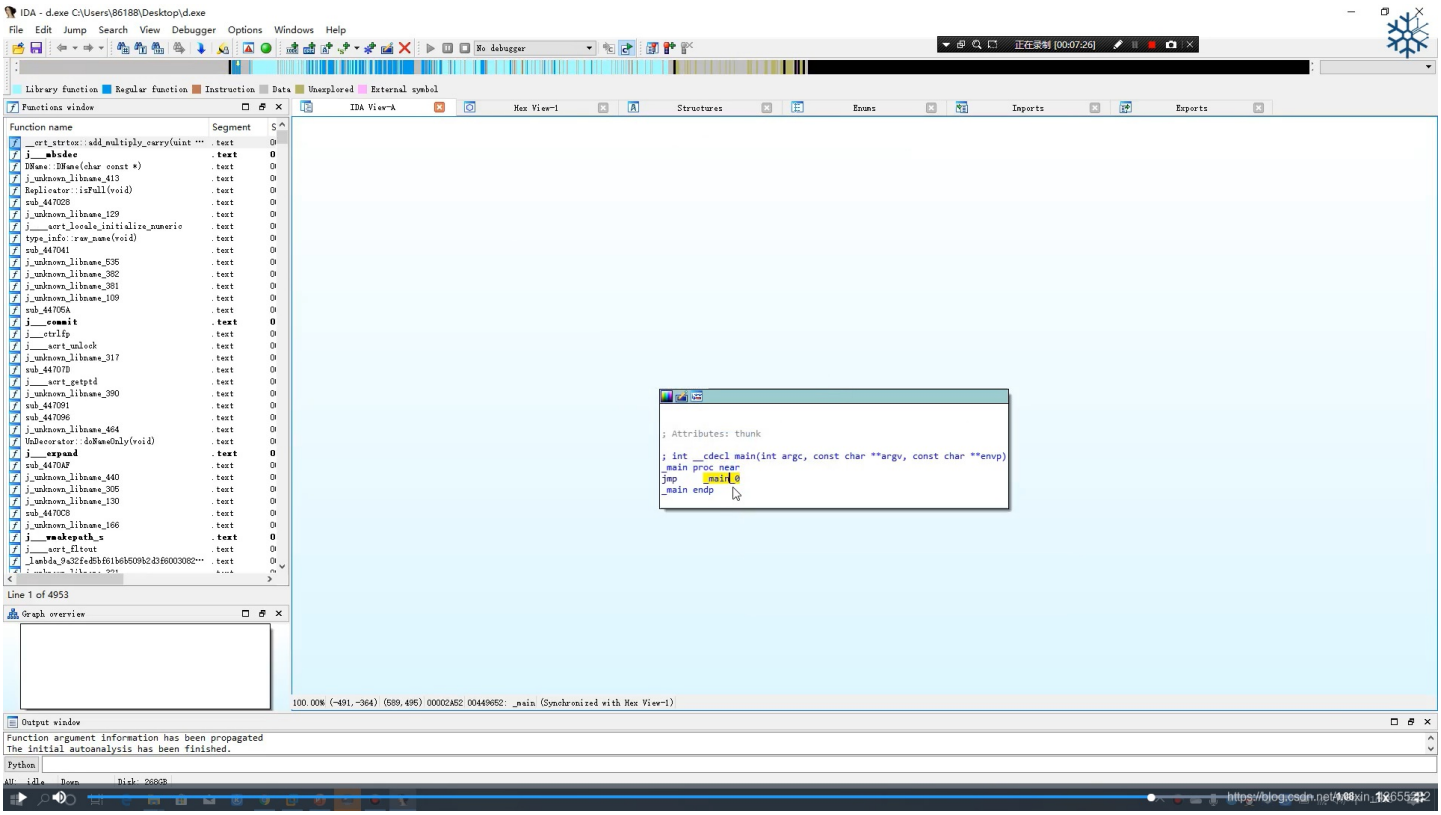
- 第二种方法



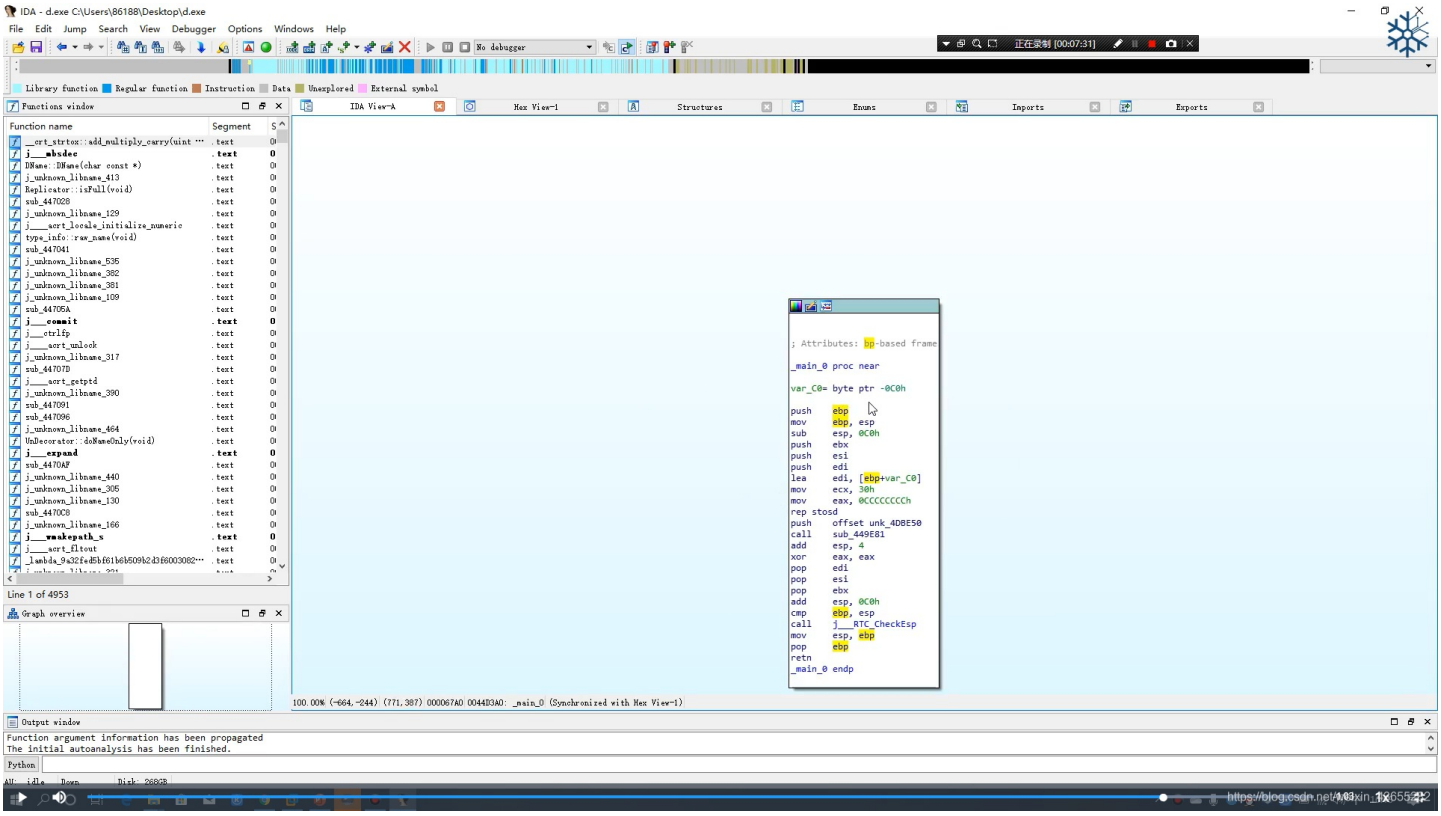

查找API调用
备份
复制
二进制
汇编(A) Space
标签 :
注释 ;
断点(P)
HIT 跟踪
RUN 跟踪
此处为新 EIP Ctrl+Gray *
转到
数据窗口中跟随
查看调用树 Ctrl+K
查找(S)
查找参考(R)
查看
复制到可执行文件
分析
Asm2Clipboard
Bookmark
去除花指令
运行脚本(S)
Script Functions...
Dump process (OllyDumpEx)
Ultra String Reference
界面选项
当前模块中的名称 (标签) Ctrl+N
所有模块中的名称
命令(C) Ctrl+F
命令序列(S) Ctrl+S
常量(T)
二进制字串(B) Ctrl+B
所有模块间的调用
所有命令
所有命令序列
所有常量
所有分支
所有参考文本字串
用户定义的标签
用户定义的注释
Command
程序入口点
https://blog.csdn.net/weixin_43655282

IDA

使用IDA打开，IDA会自动分析出main函数，按F5进入main函数。

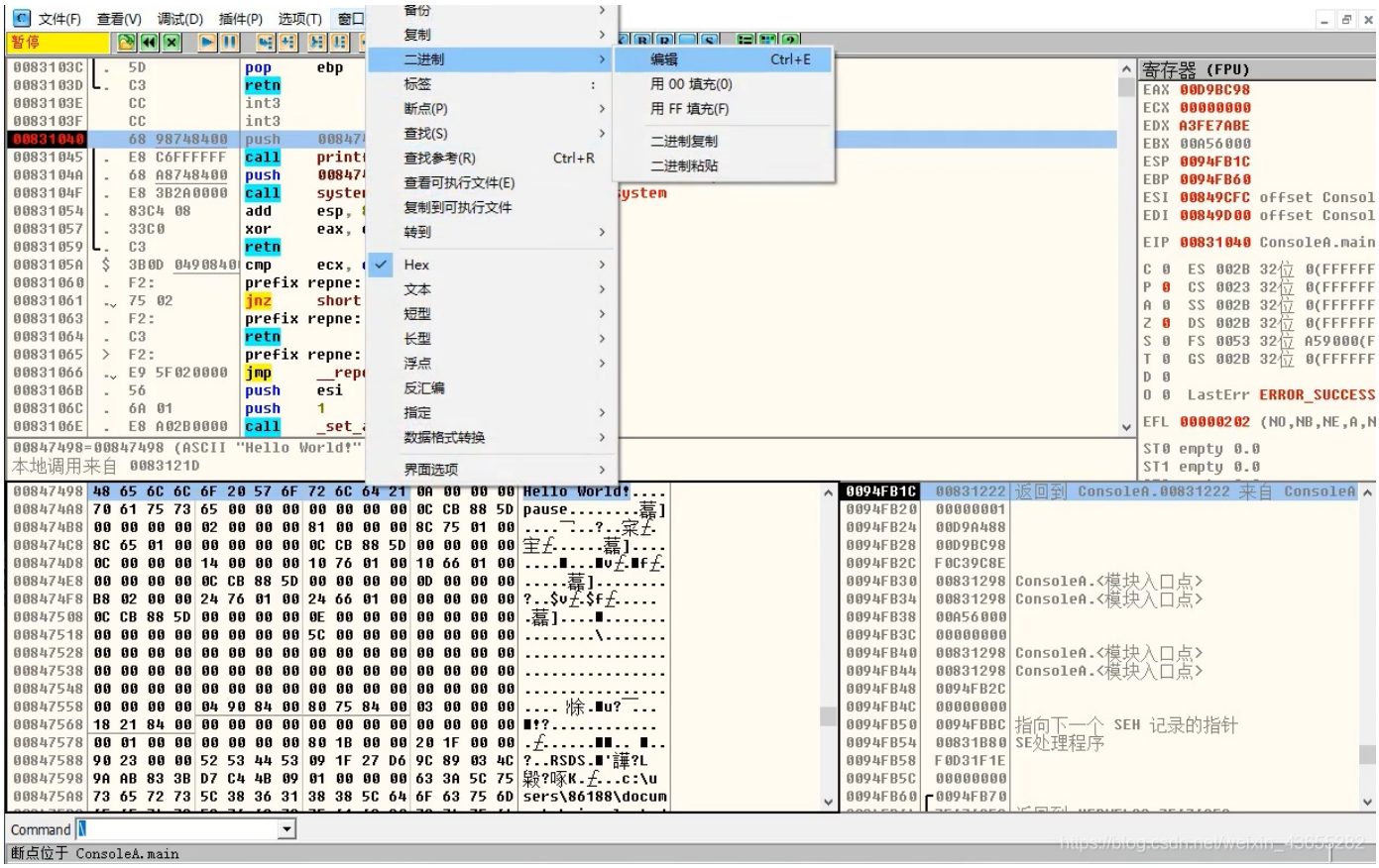


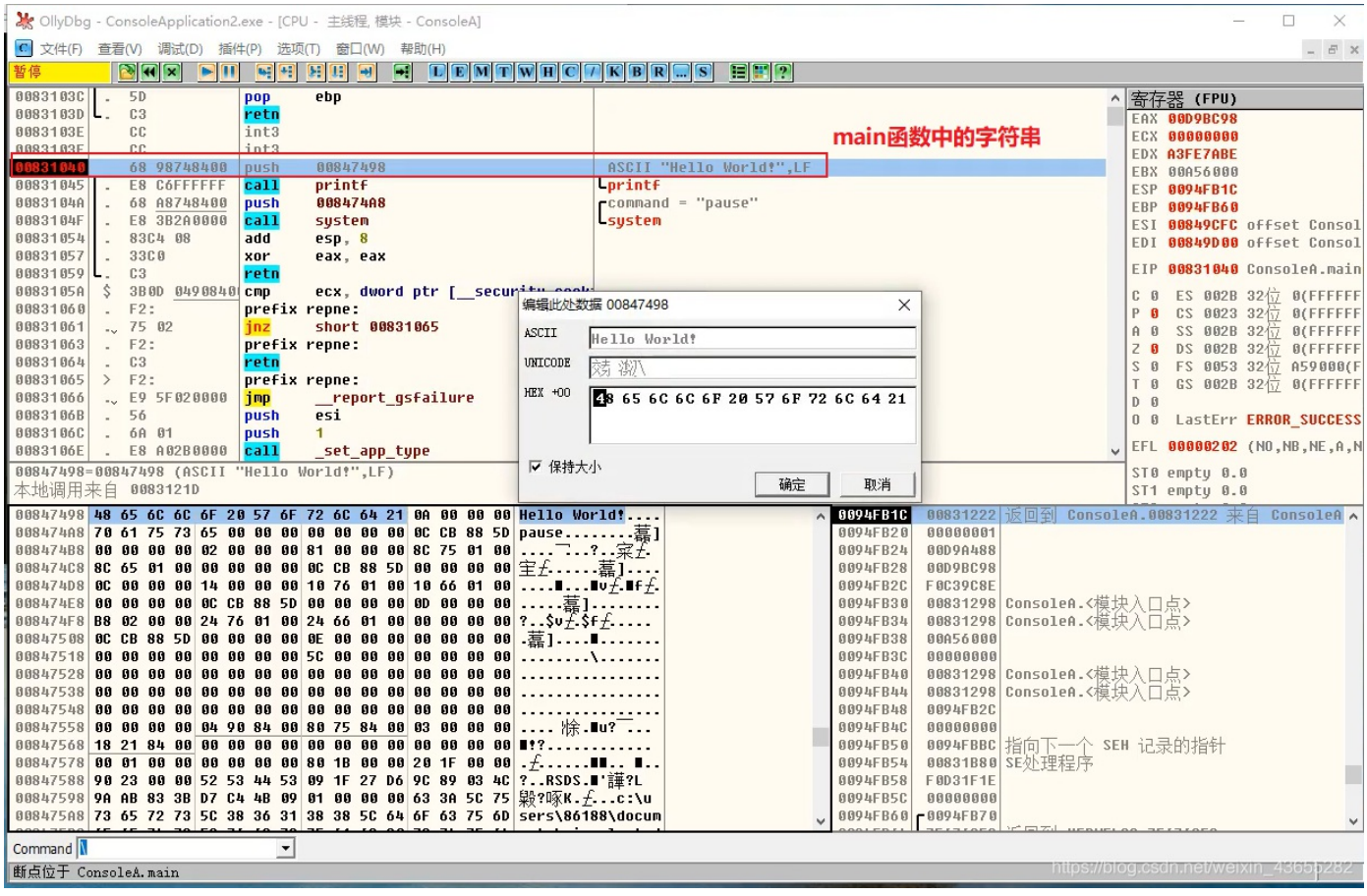
双击上图高亮部分，即可查看main函数汇编代码。



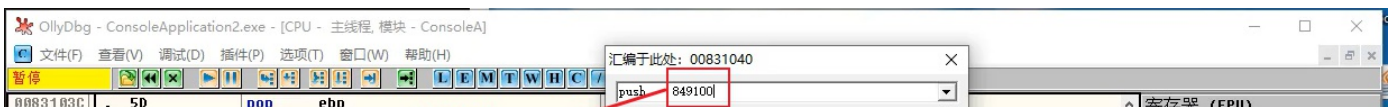
修改内存中的数据

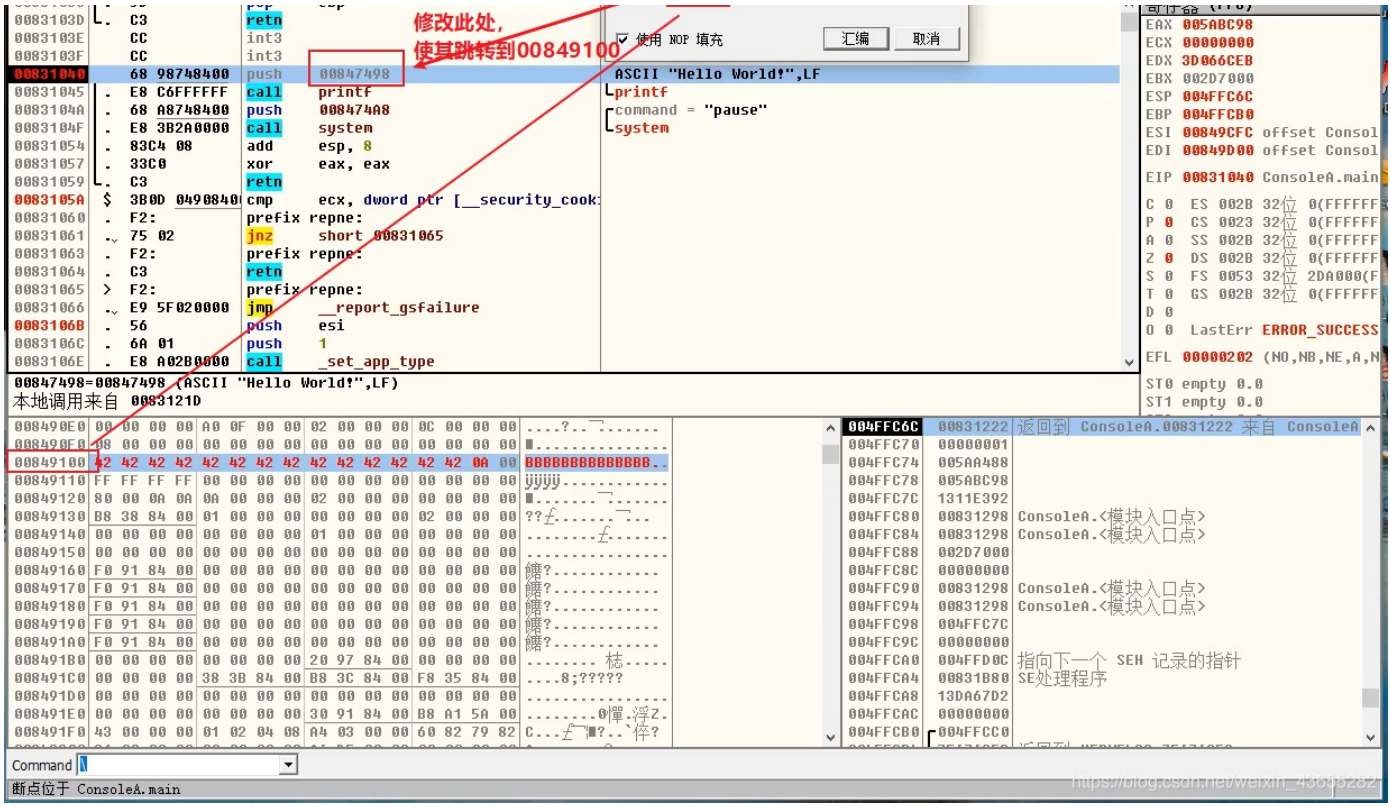
- 直接修改待修改内存中的数据

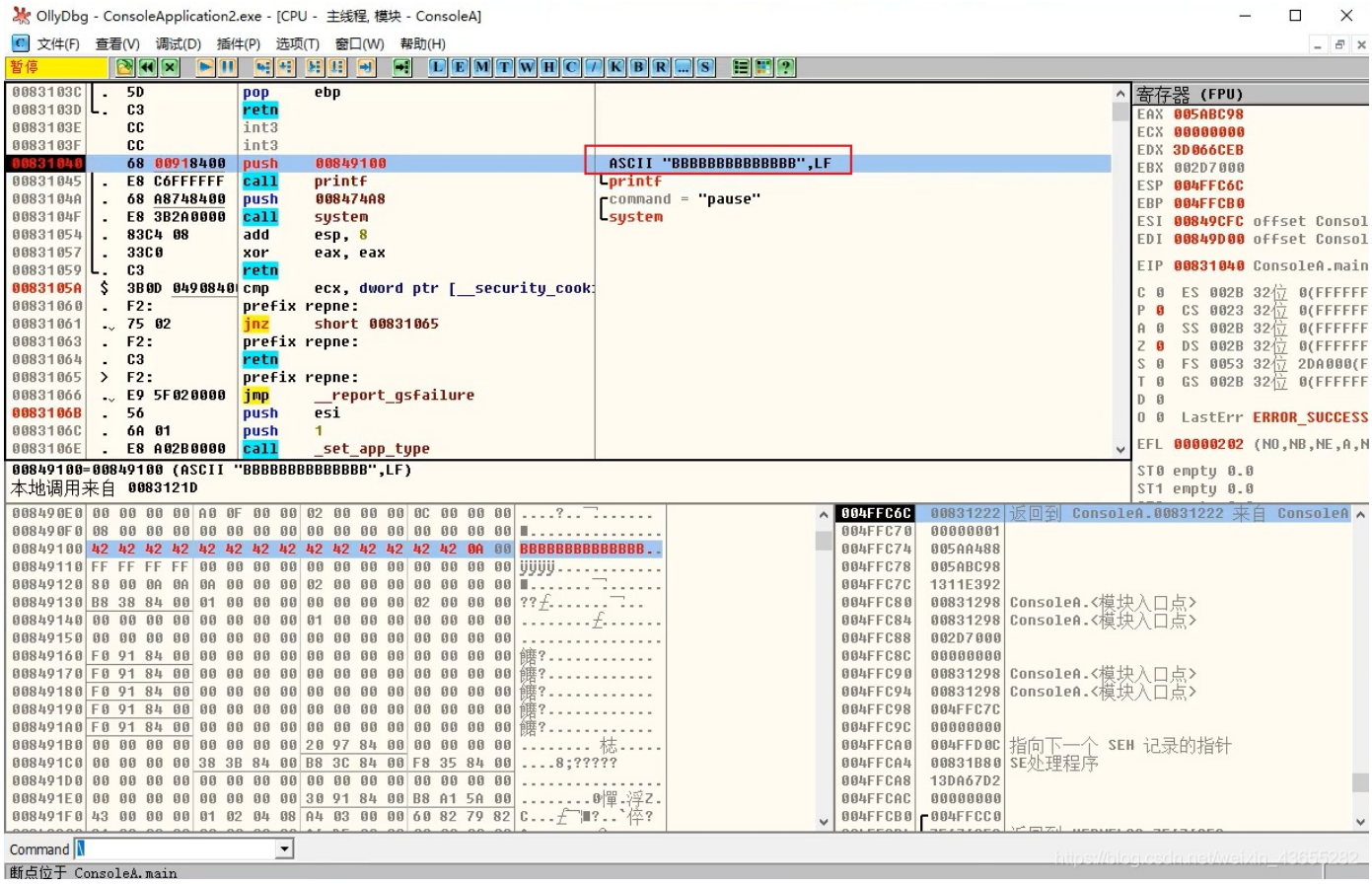




- 修改任意地址处的数据，再跳转到该地址，从而间接修改待修改内存中的数据







> 思考：
> 某些地址中的数据不能随便修改，因为可能会导致程序运行时所需的必要数据缺失而损坏程序。

修改跳转

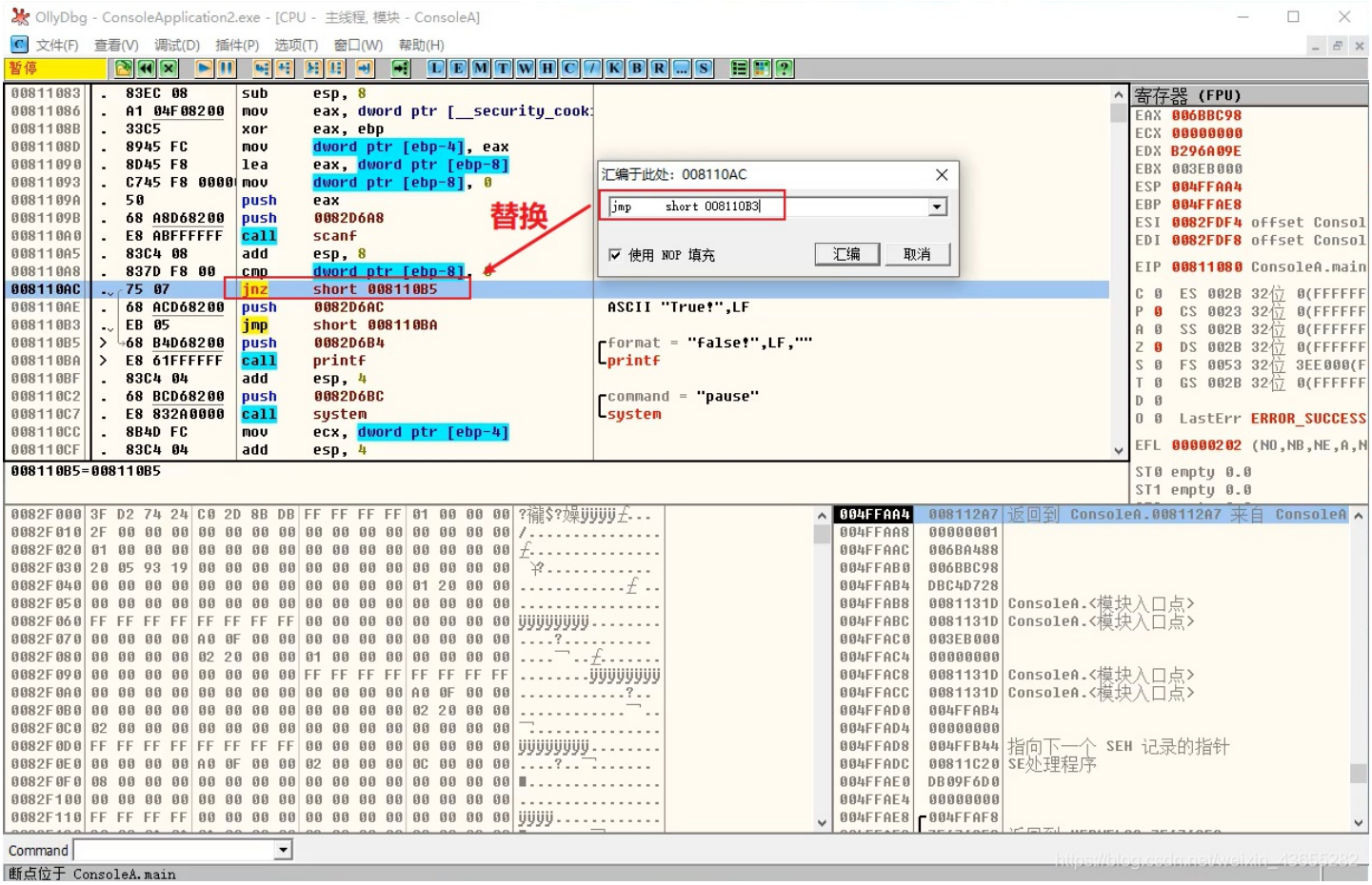
修改跳转进而改变程序执行流程。

直接上实践例子

问题：如何通过修改跳转达到就算键入0也打印"True!"
程序为Release x86版本。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int nNum = 0;
    scanf("%d", &nNum);
    if (nNum == 0)
    {
        printf("True!\n");
    }
    else
    {
        printf("false!\n");
    }
    system("pause");
    return 0;
}
```

注意！！图中修改汇编代码应该修改为jmp short 008110AE



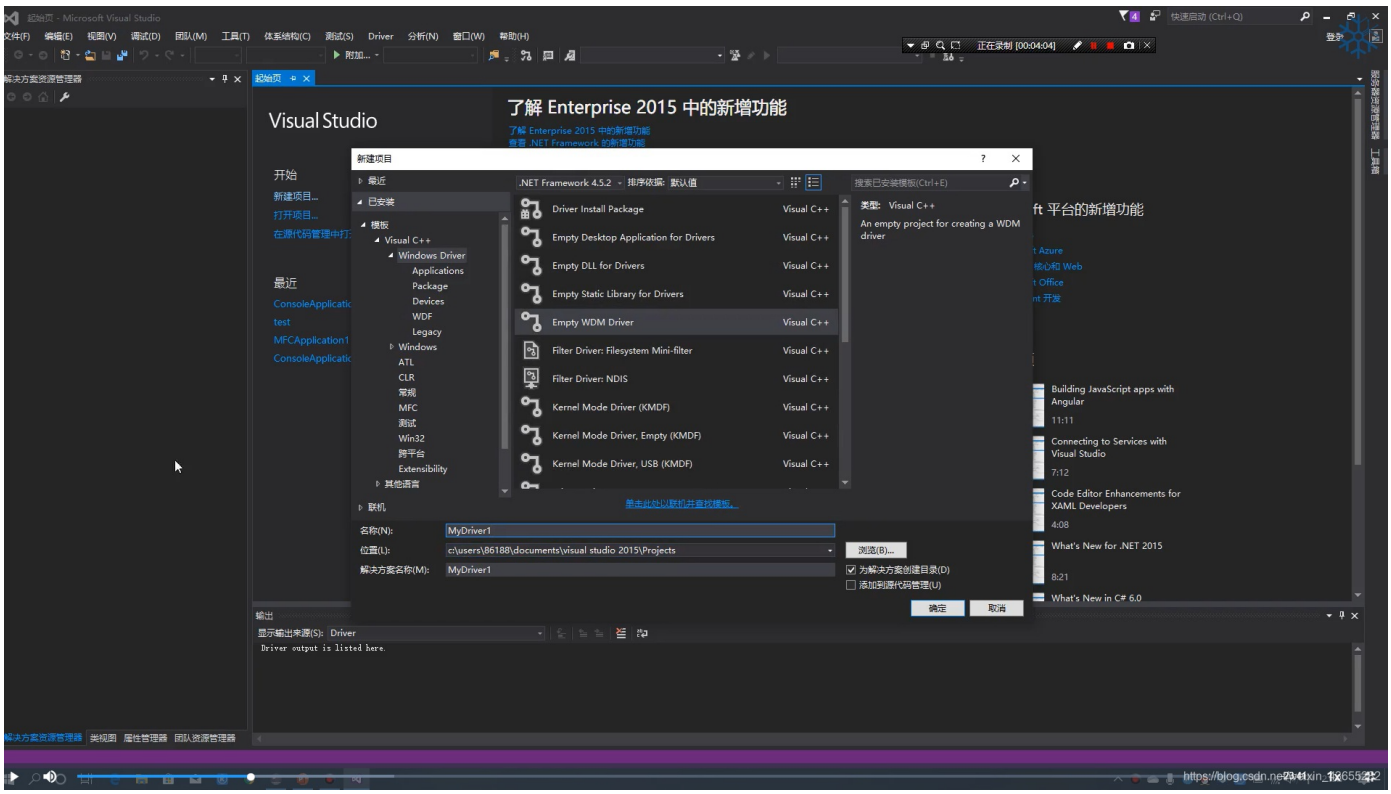
进行上述操作之后，无论是否键入0，都回跳转到**008110AE**处，这样就只会打印"True!"了。

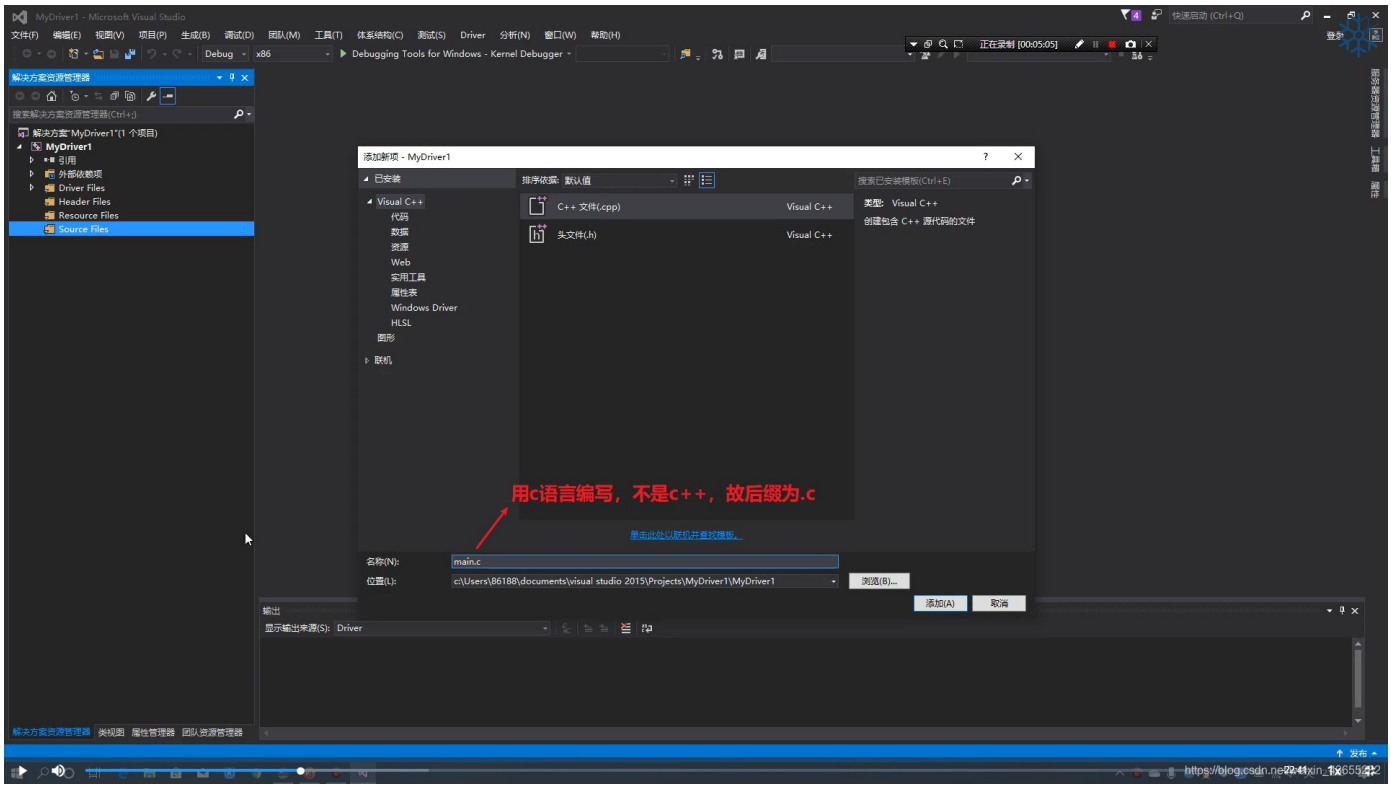
有个小问题，我们要做的第一步其实是找到main函数，但在这节课里并没用到上面笔记中的方法，以后补充。

驱动入门

第一个驱动程序

- 创建驱动项目

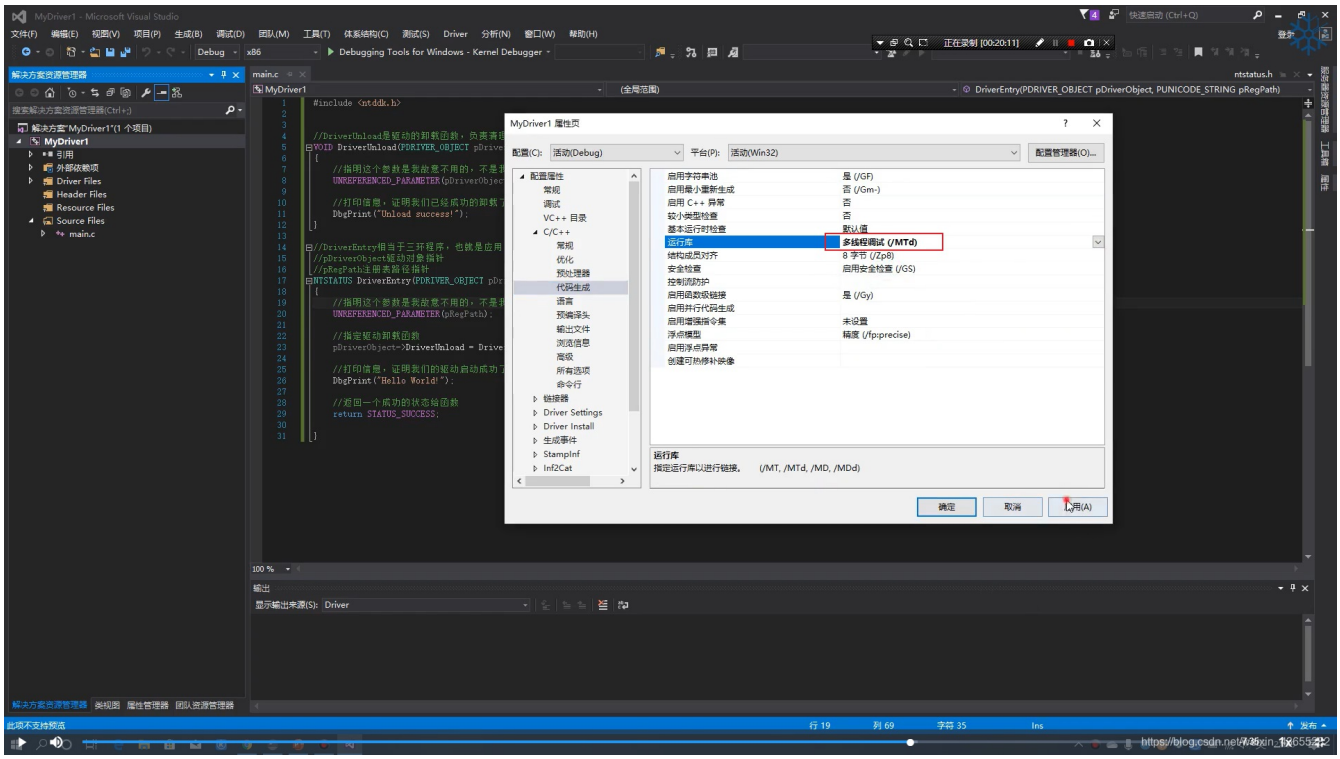


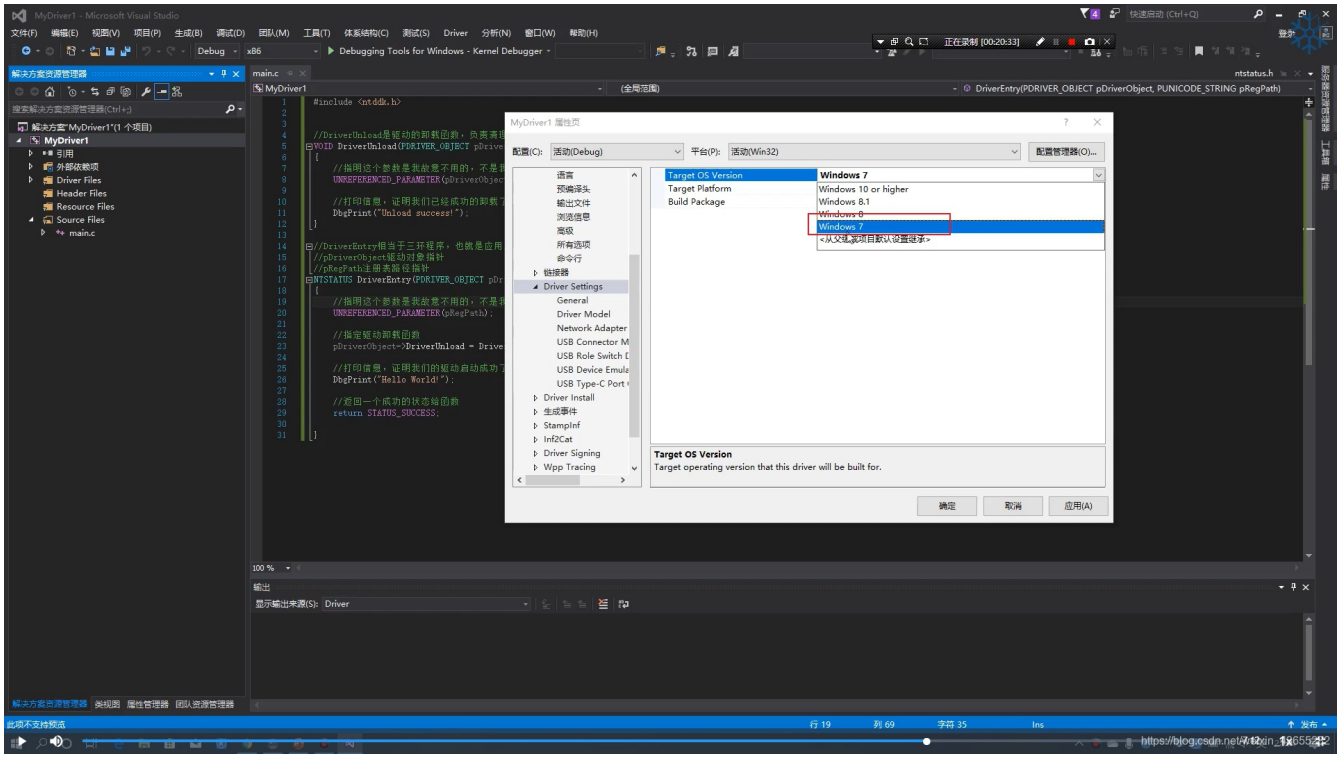


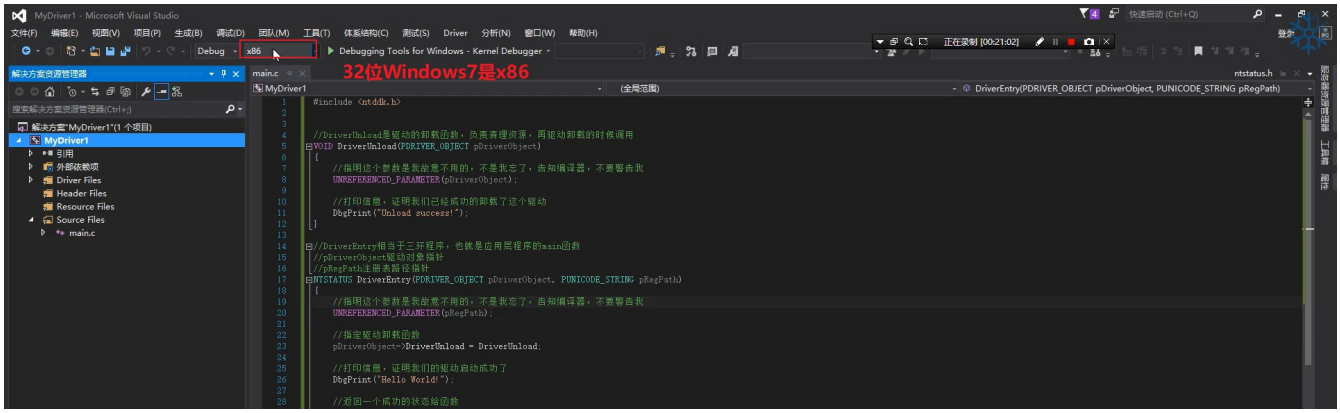
- 程序编写
  - 程序

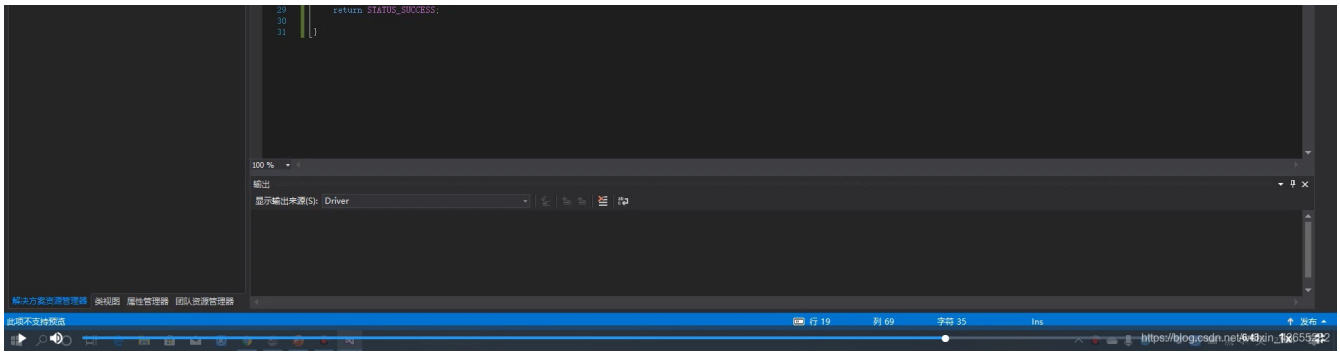
```
```

  - 程序的配置

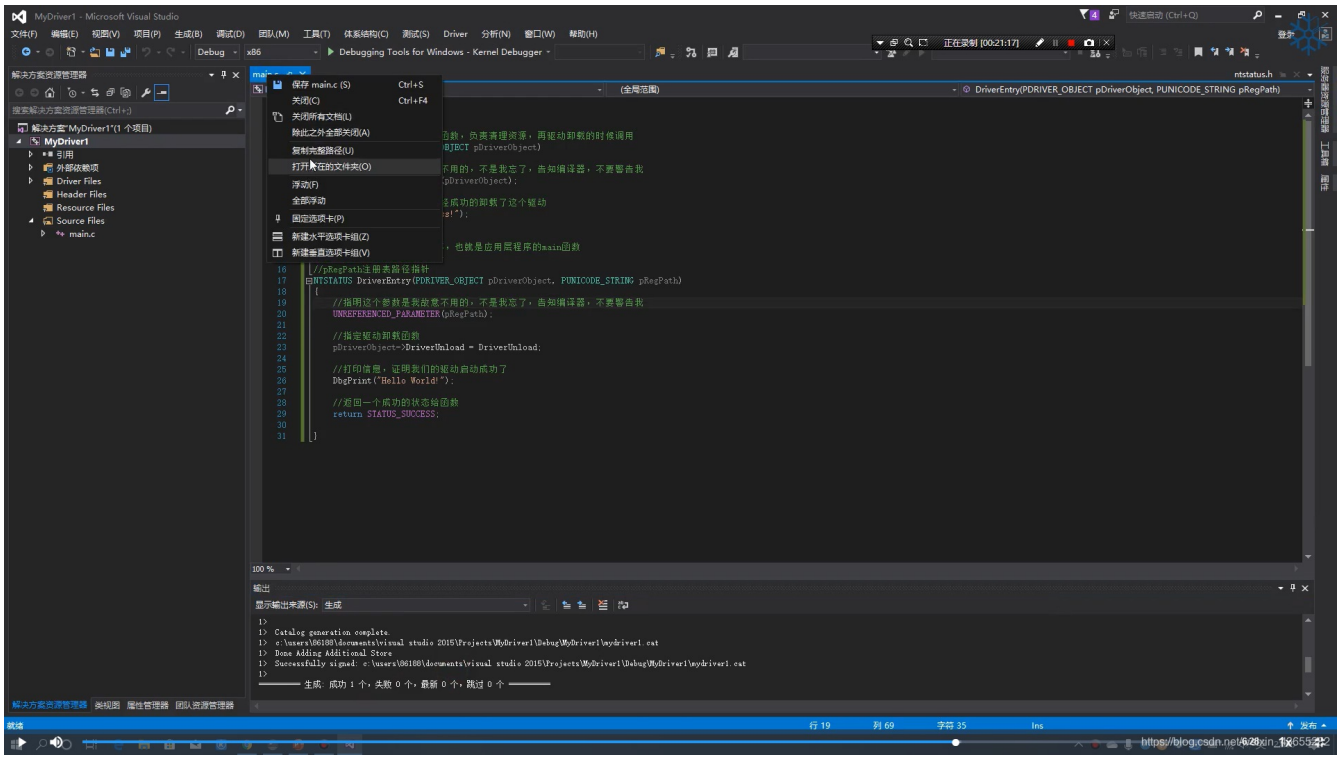

MyDriver1 - Microsoft Visual Studio
MyDriver1 属性页
配置(C): 活动(Debug)
平台(P): 活动(Win32)
配置管理器(O)...
配置属性
常规
调试
VC++ 目录
C/C++
常规
优化
预处理器
代码生成
语言
预编译头
输出文件
浏览信息
高级
所有选项
命令行
链接器
Driver Settings
Driver Install
生成事件
StampInf
Inf2Cat
启用字符串池 是 (/GF)
启用最小重新生成 否 (/Gm-)
启用 C++ 异常 否
较小类型检查 否
基本运行时检查 默认值
运行库 多线程调试 (/MTd)
结构成员对齐 8 字节 (/Zp8)
安全检查 启用安全检查 (/GS)
控制流防护
启用函数级链接 是 (/Gy)
启用并行代码生成
启用增强指令集 未设置
浮点模型 精度 (/fp:precise)
启用浮点异常
创建可热修补映像
运行库
指定运行库以进行链接。 (/MT, /MTd, /MD, /MDd)
确定
取消
应用(A)

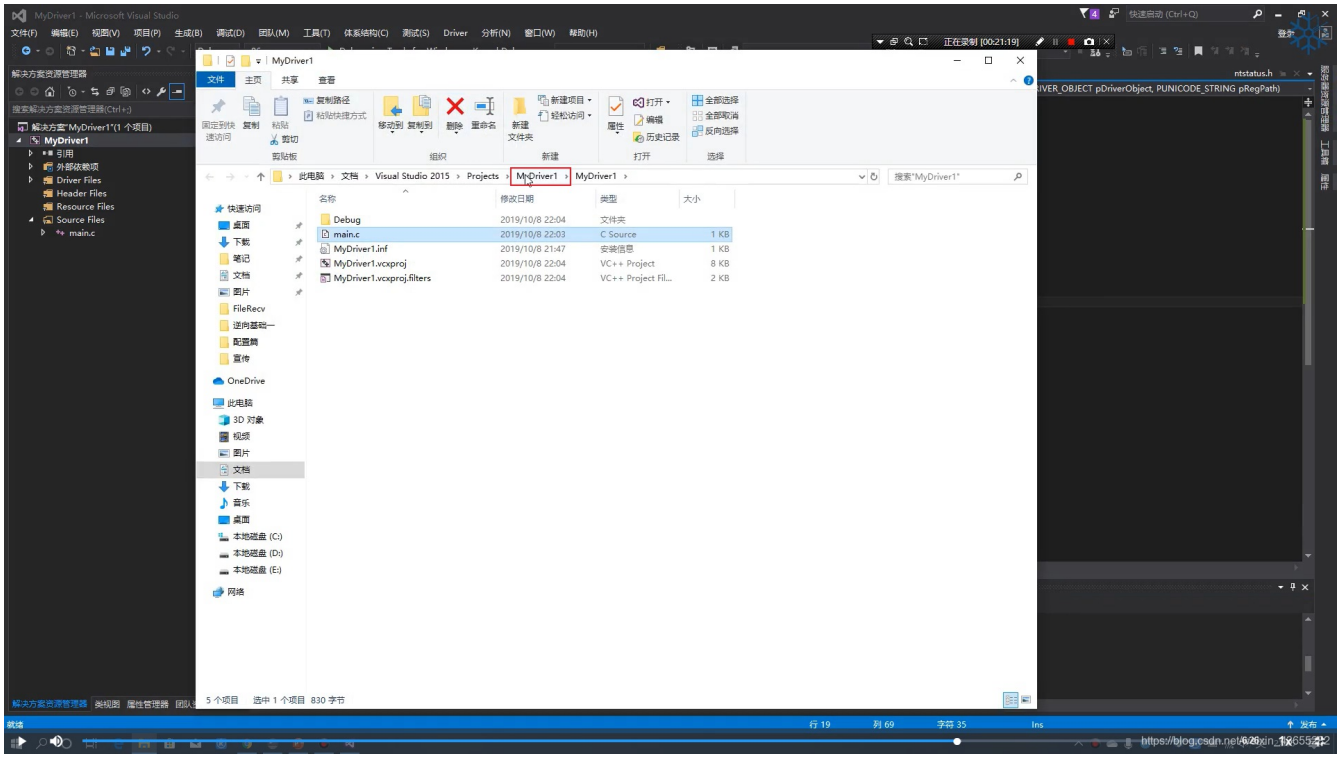

MyDriver1 - Microsoft Visual Studio
MyDriver1 属性页
Target OS Version
Target Platform
Build Package
Windows 7
Windows 10 or higher
Windows 8.1
Windows 8
Windows 7
<从父级、项目默认设置继承>
Target OS Version
Target operating version that this driver will be built for.
确定
取消
应用(A)

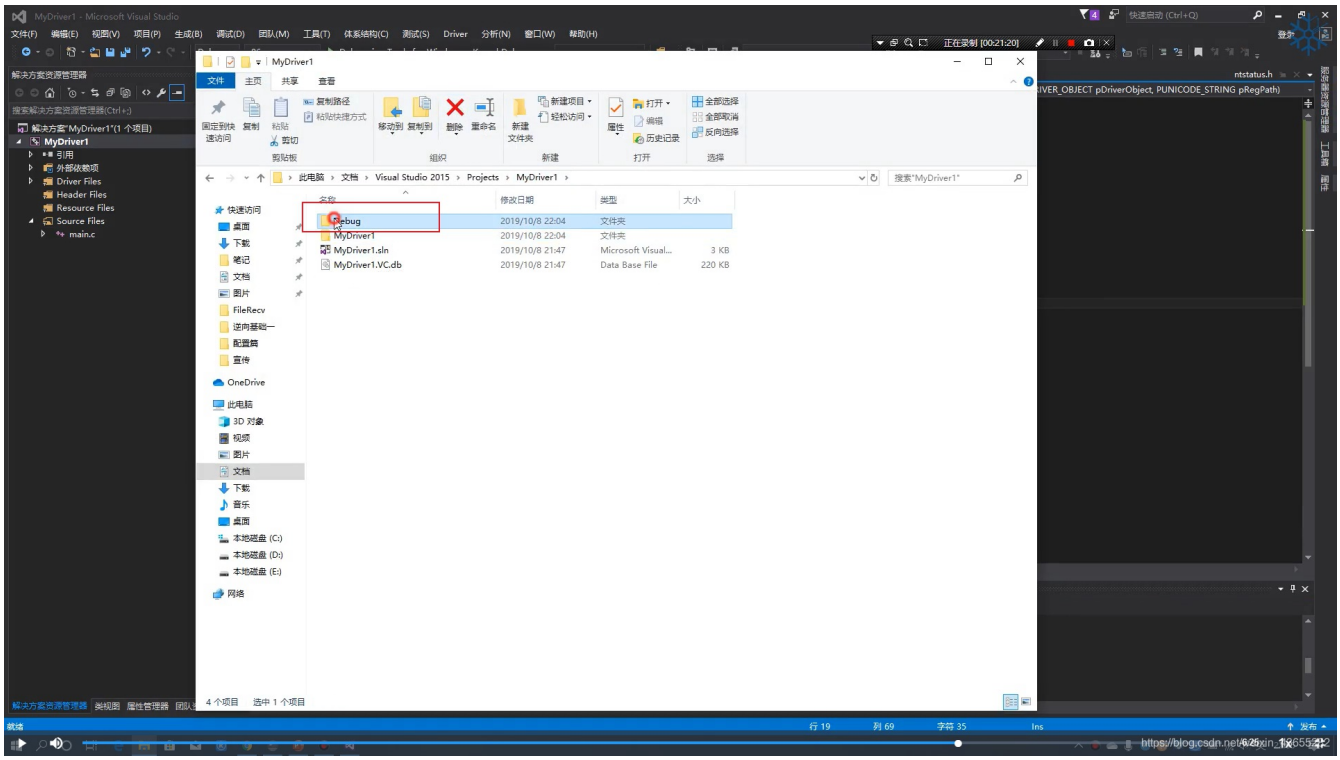

MyDriver1 - Microsoft Visual Studio
32位Windows7是x86

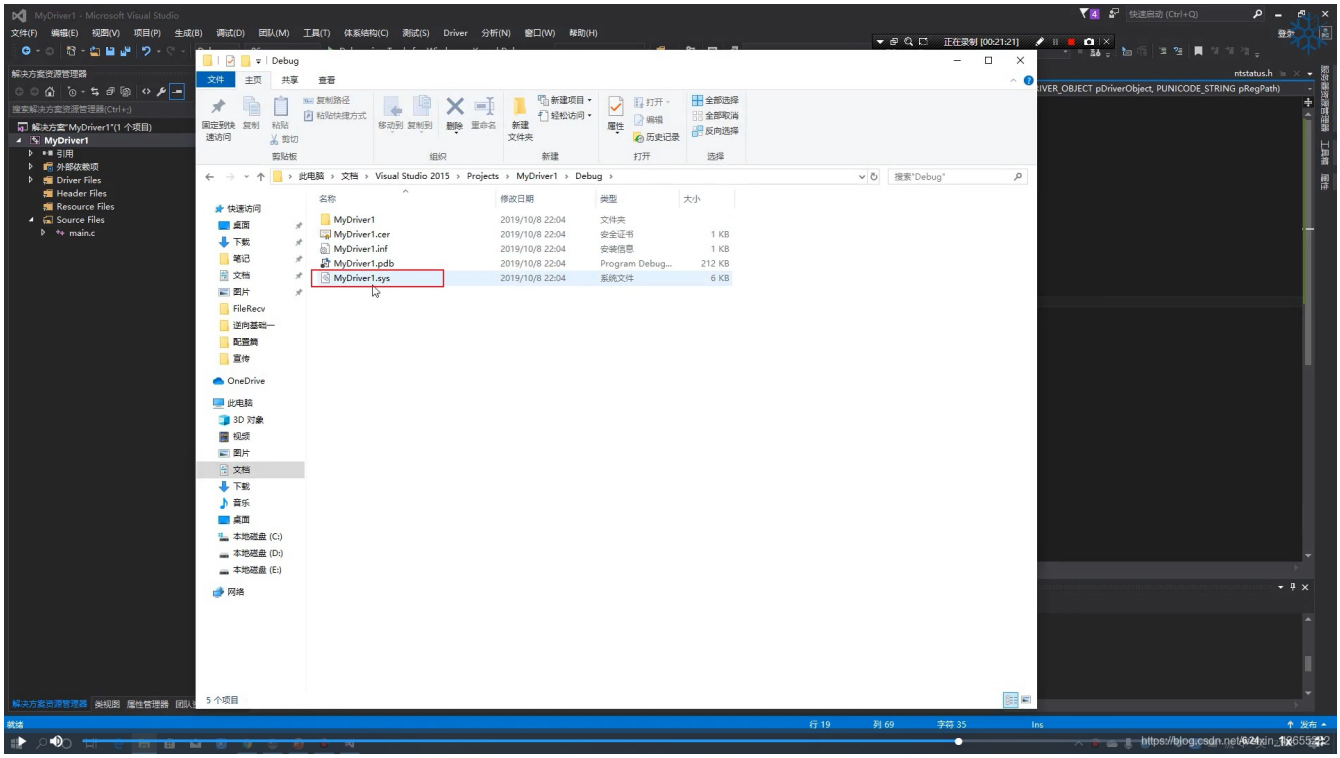


- 寻找驱动程序的存放位置

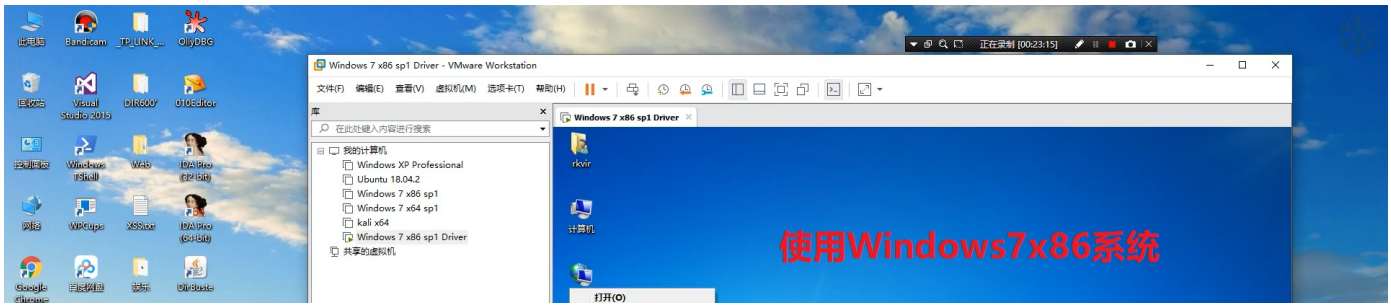


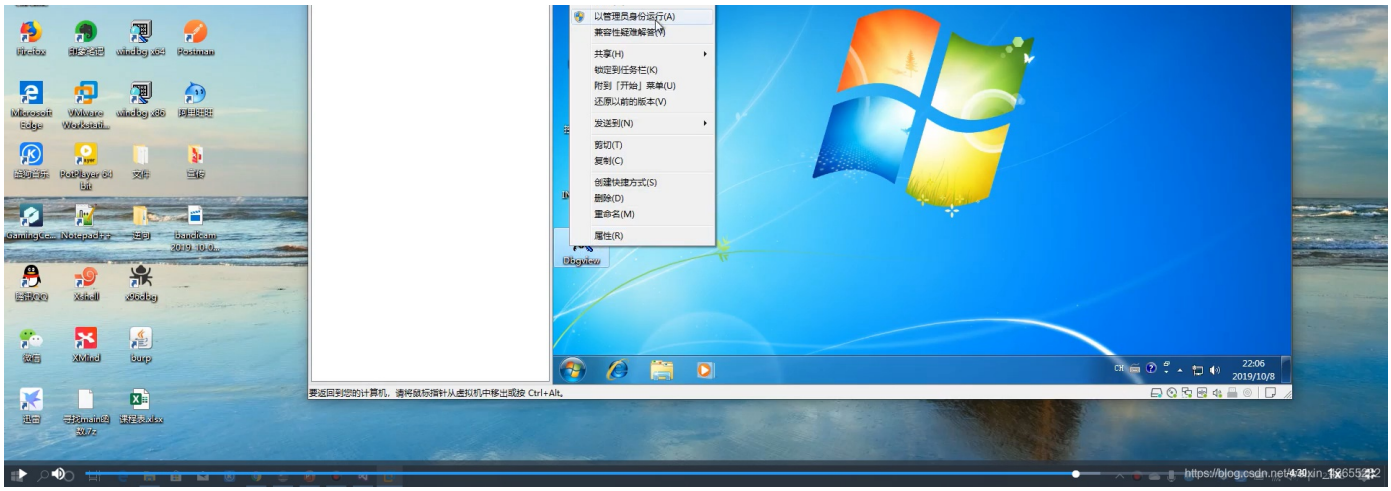


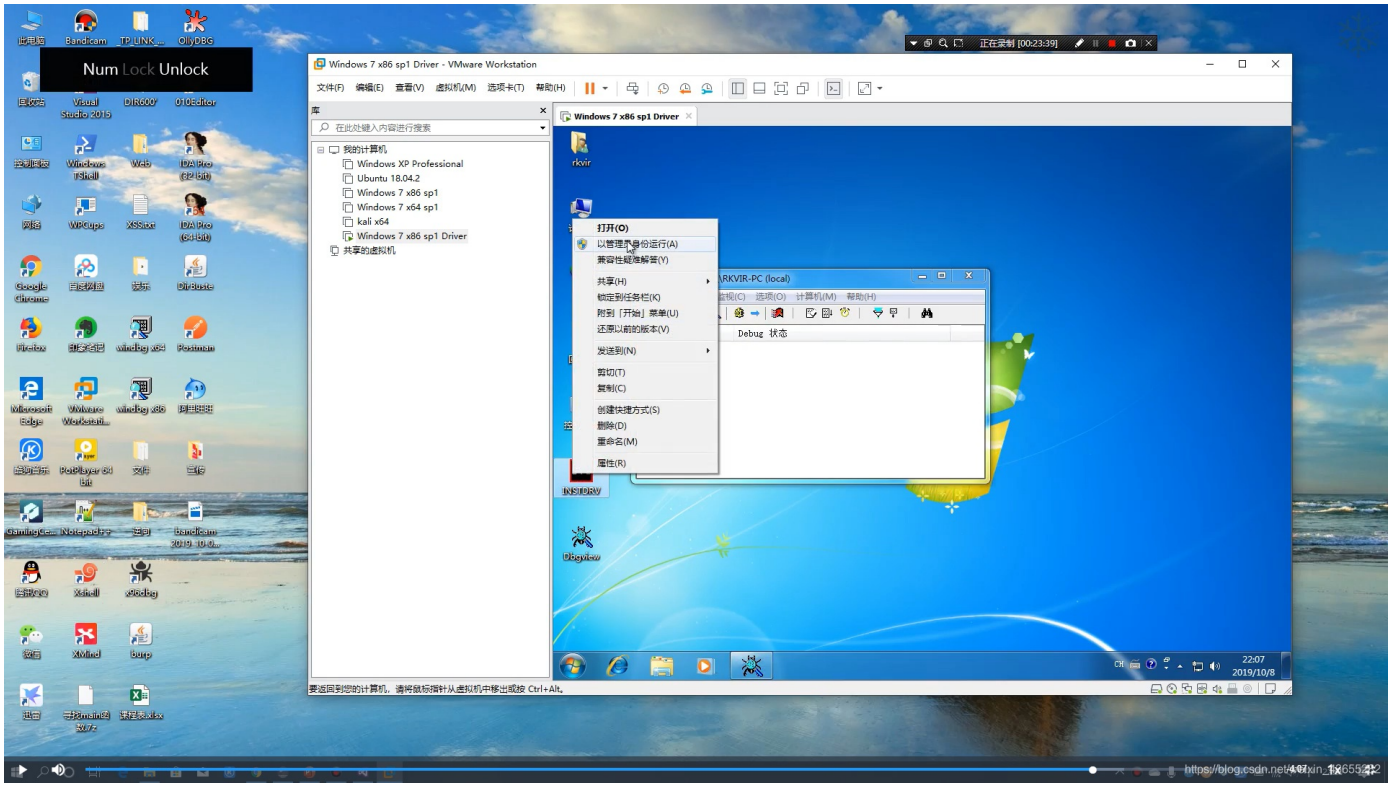

MyDriver1 - Microsoft Visual Studio
MyDriver1
Debug
MyDriver1.sln
MyDriver1.VC.db
4 个项目
选中 1 个项目

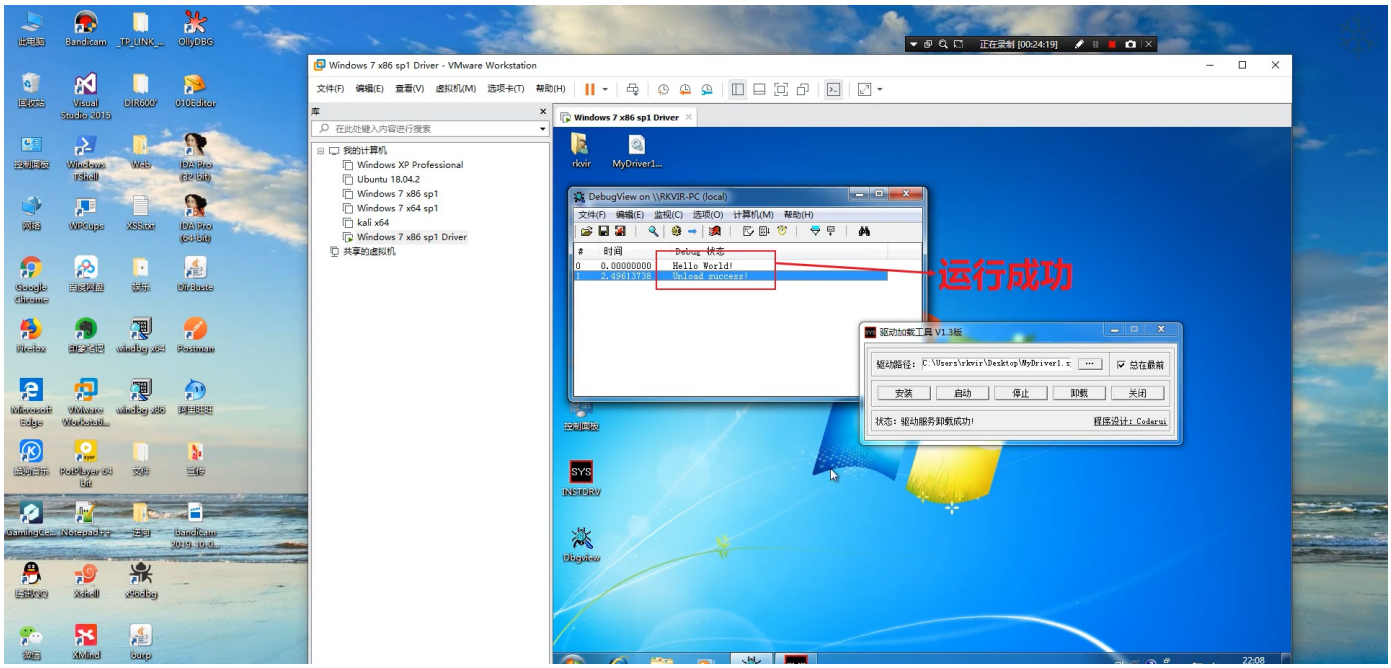

MyDriver1 - Microsoft Visual Studio
Debug
MyDriver1
MyDriver1.cer
MyDriver1.inf
MyDriver1.pdb
MyDriver1.sys
5 个项目

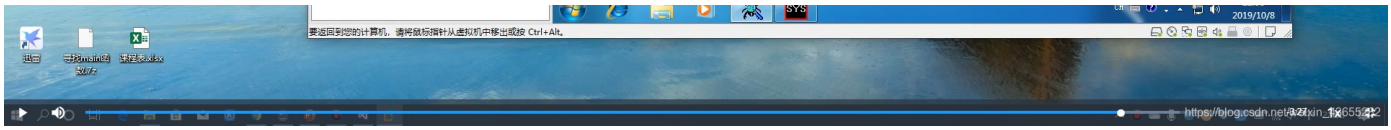
- 程序运行


Windows 7 x86 sp1 Driver - VMware Workstation
Windows 7 x86 sp1 Driver
使用Windows7x86系统


以管理员身份运行(A)
兼容性疑难解答(Y)
共享(H)
锁定到任务栏(K)
附到「开始」菜单(U)
还原以前的版本(V)
发送到(N)
剪切(T)
复制(C)
创建快捷方式(S)
删除(D)
重命名(M)
属性(R)
要返回到您的计算机，请将鼠标指针从虚拟机中移出或按 Ctrl+Alt。


Num Lock Unlock
Windows 7 x86 sp1 Driver - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟机(M) 选项卡(T) 帮助(H)
Windows 7 x86 sp1 Driver
我的计算机
Windows XP Professional
Ubuntu 18.04.2
Windows 7 x86 sp1
Windows 7 x64 sp1
kali x64
Windows 7 x86 sp1 Driver
共享的虚拟机
打开(O)
以管理员身份运行(A)
兼容性疑难解答(Y)
共享(H)
锁定到任务栏(K)
附到「开始」菜单(U)
还原以前的版本(V)
发送到(N)
剪切(T)
复制(C)
创建快捷方式(S)
删除(D)
重命名(M)
属性(R)
RKVIR-PC (local)
Debug 状态
要返回到您的计算机，请将鼠标指针从虚拟机中移出或按 Ctrl+Alt。


Windows 7 x86 sp1 Driver - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟机(M) 选项卡(T) 帮助(H)
Windows 7 x86 sp1 Driver
我的计算机
Windows XP Professional
Ubuntu 18.04.2
Windows 7 x86 sp1
Windows 7 x64 sp1
kali x64
Windows 7 x86 sp1 Driver
共享的虚拟机
DebugView on \\RKVIR-PC (local)
时间 Debug 状态
0 0.00000000 Hello World!
1 2.49613738 Unload success!
运行成功
驱动加载工具 V1.3版
驱动路径：C:\Users\rkvir\Desktop\MyDriver1.s
总在最前
安装 启动 停止 卸载 关闭
状态：驱动服务卸载成功!
程序设计：Coderui
要返回到您的计算机，请将鼠标指针从虚拟机中移出或按 Ctrl+Alt。



第一个驱动程序

代码如下：

```
#include <ntddk.h>

//DriverUnload是驱动的卸载函数，负责清理资源，再驱动卸载的时候调用
VOID myDriverUnload(PDRIVER_OBJECT pDriverObject)
{
 //指明这个参数是我故意不用的，不是我忘了，告知编译器不要警告我
 UNREFERENCED_PARAMETER(pDriverObject);

 //打印信息，证明我们已经成功地卸载了这个驱动
 DbgPrint("Unload success!");
}


//DriverEntry相当于三环程序，也就是应用层程序的main函数
//pDriverObject驱动对象指针
//pRegPath注册表路径指针
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegPath)
{
 //指明这个参数是我故意不用的，不是我忘了，告知编译器不要警告我
 UNREFERENCED_PARAMETER(pRegPath);

 //指定驱动卸载函数
 pDriverObject->DriverUnload = myDriverUnload;

 //打印信息，证明我们的驱动启动成功了
 DbgPrint("Hello World!");

 //返回一个成功的状态函数
 return STATUS_SUCCESS;
}
```