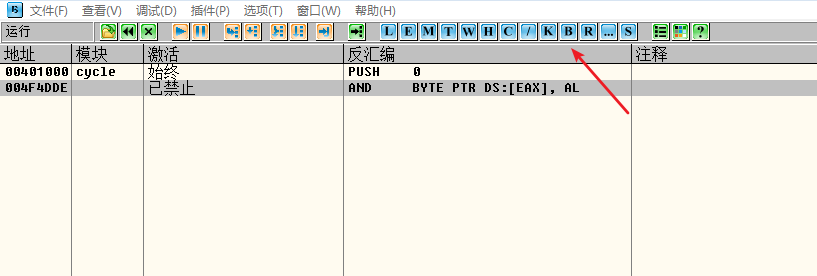
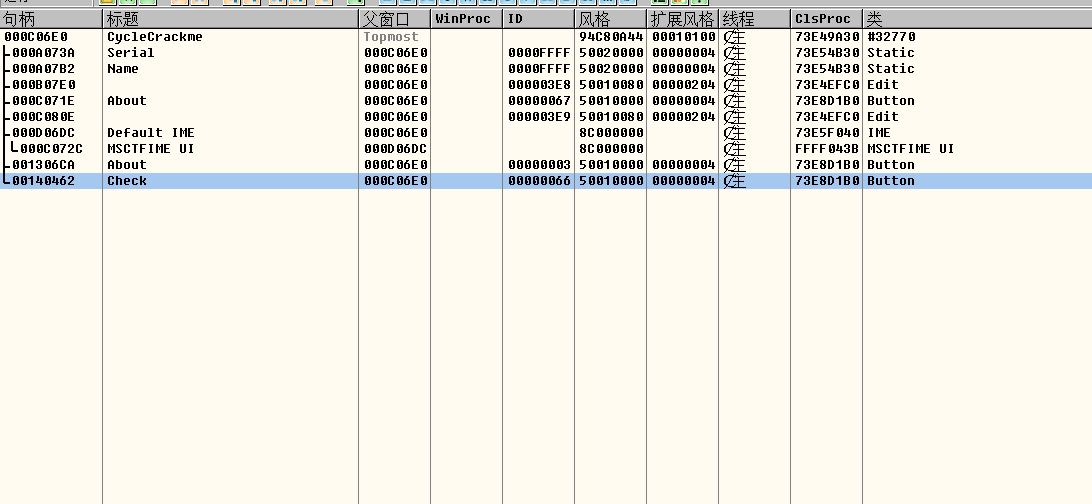
1. 被调试的程序如果之前就被加过断点，重新调试的时候就会出现 一个提示框，

大概的意思就是出现不安全断点，解决方法是

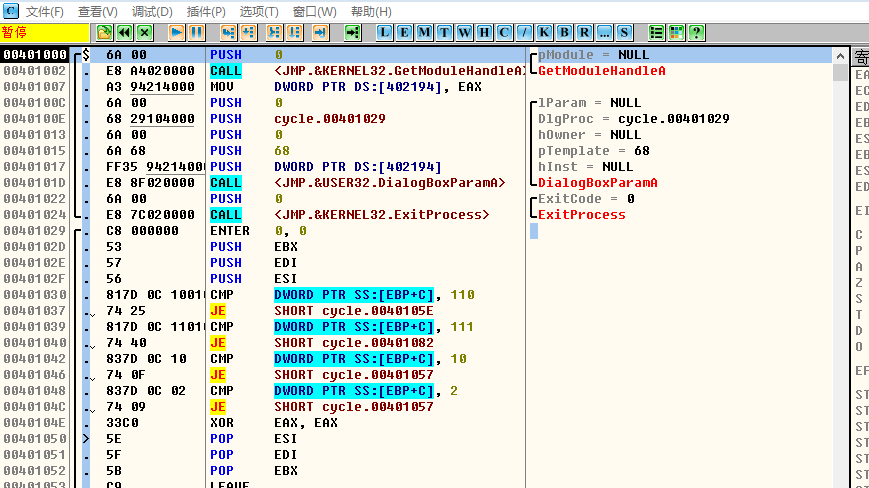


删除断点即可

1. 如果点击窗口按钮 W，出现了一片空白，只要右键刷新一下即可，出现当前程序的窗口。



1. 如果调试程序的时候，出现在系统领空，只要alt+F9 即可回到用户领空，领空的概念就是内存的区域。

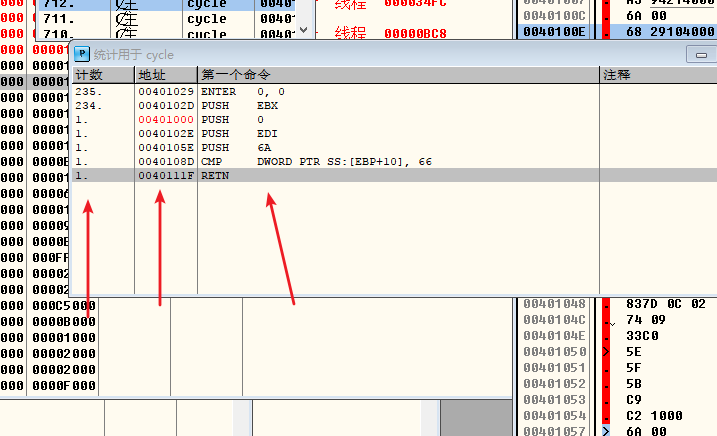


1. 这个程序alt+M在code段上设置访问终端断点的作用是可以直接回到用户代码。



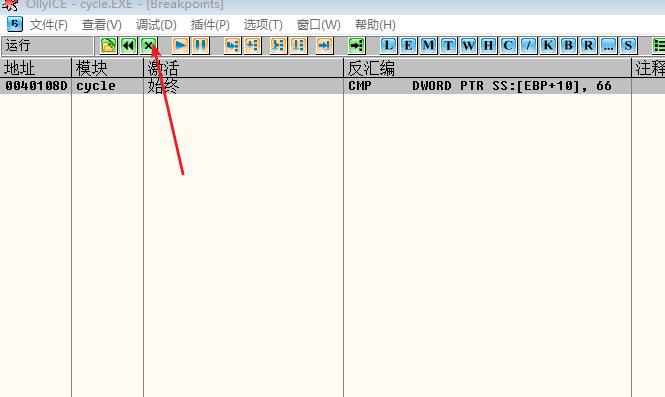
1. RUN 跟踪就是把被调试程序执行过的指令保存下来，让你可以查看被调试程序运行期间干了哪些事。RUN 跟踪会把地址、寄存器的内容、消息以及已知的操作数记录到 RUN 跟踪缓冲区中，你可以通过查看 RUN 跟踪的记录来了解程序执行了那些指令。在这还要注意一个缓冲区大小的问题，如果执行的指令太多，缓冲区满了的话，就会自动丢弃前面老的记录。我们可以在调试选项->跟踪中设置：

6.



在进行模块统计的时候，第一列可以看到模块的执行次数

1. 在菜单栏的调试选项中可以关闭run跟踪



8.LoadPE的作用

1.可以查看分区表

2.可以修改分区分区表的访问属性

3.可以动态添加dll和相应函数分区表。

9.在调试中，animate step over/in 这两个命令非常方便，尤其我觉得在对付nag程序的时候，几乎不必设断，比如，先analyze程序，然后选animate   
step over或in，程序开始运行，光标在命令行上滑过（要的就是这种效果），然后nag出现，仔细看看反汇编窗格，是不是光标停在一个call上，它到底call什么我们不关心，反正call出了nag对话框，然后往上看看，极有可能有个jz,jnz，直接改了跳过即可（一般程序而言）

10. 然后有两个命令是非常好的东西，hit trace和run trace,hit trace可以用来查看在你选定的命令范围内命令被执行与否，执行了前面就有个红竖线，而run trace可以为你设定的命令范围作一个运行的profile,有了它，你就可以完全掌握程序的跳转，在爆破的时候不用为了一个个跳转反复的找。

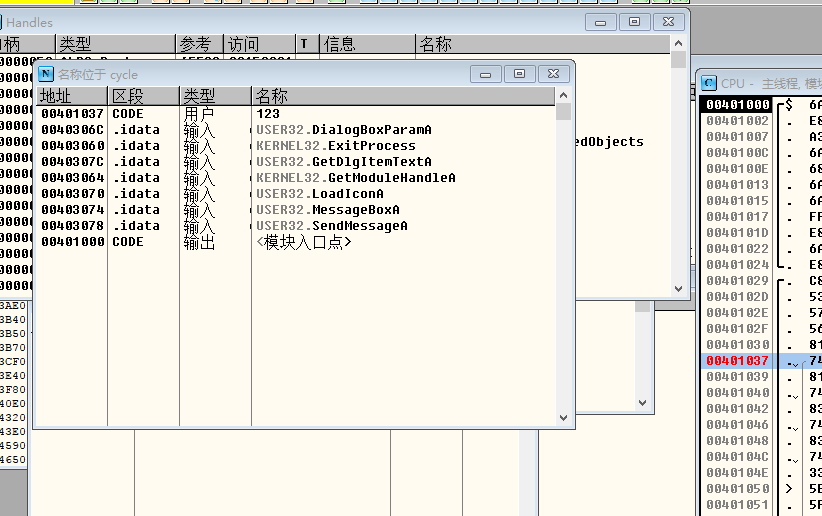
73ec9929 user32中messageboxA地址

Ollydbg中加标签和加注释的方法

加注释的方法是 ; 分号

加标签的方法是： 冒号 具体的操作室 shift+；分号。

Ctrl+N 查看标签列表



在数据窗口中，也可以设置标签，但是不能使用冒号，进行设置，正确的操作是

对应的数据右键标签

Alt+e 查看调用的动态链接库 dll，可以通过 窗口—》其他窗口—》executable modules进行查看

Ollydbg 中如果想要调到特定的地址，具体的操作是右键---》转到---》表达式—》00401024

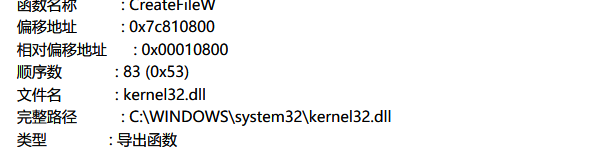
想要在数据窗口中查看反汇编窗口的命令的方法是 选择一条命令—》右键—》在数据窗口中跟随—》选择，这样在数据窗口中即可看到

Ollydbg中如果还没有运行到某个call，但是想要查看这个call的代码，可以使用+进入这个call。使用方向键进行移动，使用-号脱离这个call，\*号相对于－号更近了一部，直接返回当前指令。

当在数据窗口中，找到某个字符串的时候，如果想要在反汇编窗口中查看它的引用，操作步骤是选中参考字符串，右键—》查找参考/ ctrl+r，即可。

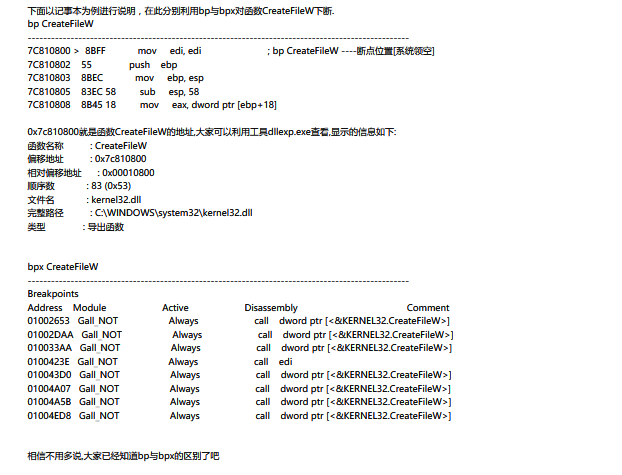
在IDA中 查看字符串，后边的；data text 04012014+o👆即可查看

可以使用dllexp.exe查看dll相关信息，如dll导出表的函数的个数，



Bp和bpx的联系和区别

Bp和bpx都是下断点的作用，但是bp是给一个语句下断点。Bpx是给调用这个函数的语句下断点。



Ollydbg是的花指令 使用ctrl+↑/↓进行设置开始的程序入口点。原理是将设置之前的入口点进行整合。返回变成新的指令。

Ollydbg反汇编窗口中，右键—》查找---》在当前模块中查找，可以看到在当前的窗口中所调用的函数名称，和函数类型，值得注意的是 使用标签标记的位置也会出现在这个窗口中。

关于ollydbg中断点的总结

1. ollydbg中断点分为 硬件断点， 内存断点 ，普通断点

硬件断点的原理是利用了cpu内部的 调试寄存器 dr0-dr3

内存断点的原理是假如你用int 3断点对数据区下断，OD会提示你断点可能不会实现，其实也是必然，程序不可能执行数据区，然而我们却可以当数据被读取或写入时进行下断，这种原理主要基于内存属性，当下读写断点是，OD会修改断点处读写属性，如果程序对此数据读写的话，会产生读写异常，OD捕捉此异常并分析，其可以知道运行到何处，对代码段也可以下此断点，机理相似。

普通断点的原理是替换语句为cc异常，当访问异常捕获异常进行处理

1. 值得说明的是

1使用ollydbg进行调试的时候，调试器会在读取程序的时候，创建一个odd类型的文件，用于调试此程序的记录，在调试器关闭的时候，调试器会根据这个odd还原这个程序的文件

思考，如果这是真的，那么在调试程序的时候突然断电，那么程序可能无法使用，因为一些断点没有被替换过来

2ollydbg首先根据pe结构来确定程序入口点，然后根据 首指令和指令的参数类型确定第一条指令，之后依次将指令换成cc，这个循环下去反编译完整个程序。

1. 也就是说，可能存在着一些校验函数，如果程序的某些位置被下断点，那么说明程序正在被调试。普通的cc断点可以被校验程序检测出来，但是内存断点和硬件断点利用的是操作系统的调试和异常机制，是无法被校验程序检测出来的。
2. 关于ollydbg中导入的lib文件的作用，因为为了优化程序的运行，将程序中使用dll方法的地方编译的时候使用相应的符号，也就是说dll中存在着一张符号和dll地址的对应表。但是，编程的时候使用函数的时候使用的函数的名字。这其中就需要一个名字和地址和符号相关联的数据结构，IAT 输入地址表。

现在来梳理一下，程序中用的是符号， lib文件是 在编译的时候，将名字编程符号，并且帮助dll根据符号建立符号地址表。

实际操作系统会将输入表保存在程序中，使用的时候二次解析，（和dll文件相配合）。

关于反调试的一些思考

1. 在程序中保留一张黑名单，程序运行壳程序程序，枚举系统进程，进行环境检查
2. 在某些位置进行cc校验，如果在这些位置上下断点，校验值出现异常，说明正在被调试
3. 花指令，

花指令的作用是对付静态分析，以下面一段程序说明一下花指令的原理

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

#include <iostream.h>  
#include <windows.h>  
void main()  
{  
    \_asm  
    {  
            jmp l2  
        \_EMIT 0x1//这里就是花指令  
        \_EMIT 0x2//这里就是花指令  
        \_EMIT 0x3//这里就是花指令  
        \_EMIT 0x4//这里就是花指令  
l2:  
        mov eax,0x11111111  
    }  
}

[复制代码](javascript:void(0);)

可以看到，程序直接跳转到标签l2了，在jmp指令和l2之间的就是花指令，花指令为什么能起作用呢？那是因为反编译器在反编译的时候不会像我们人一样去理解jmp和l2之间的指令是永远不会执行到的，所以在反编译的时候就把这段乱七八糟的代码作为正常的指令了，而这样的反编译会影响mov eax,0x11111111这个指令的正确识别，所以导致在OD中以上代码不会正确显示。

了解了原理我们就可以自如地设计花指令了，比如再加一段\_EMIT 0x5等等。

花指令不光是能够用jmp指令来设计，还可以用call指令配合ret指令来进行设计，原理是这样的：我们知道call指令等于这样两条指令，一是把自身所在位置的下一条指令的地址压入堆栈，二是jmp到call的地址处，而ret指令可以理解为jmp到call指令压入堆栈的地址，因此，可以用call指令这样来写花指令：

1.call一个地址，在call下面随便写一点花指令，但是要注意一点与jmp版花指令不同的，我们要记得自己写的花指令占了多少个字节，比如，占了2字节，至于为什么要记得，往下看

2.在call里面，也就是函数里面，首先pop出压入的地址，然后把这个地址减去花指令占用的字节数，这里是2字节，再重新push进堆栈，然后就ret

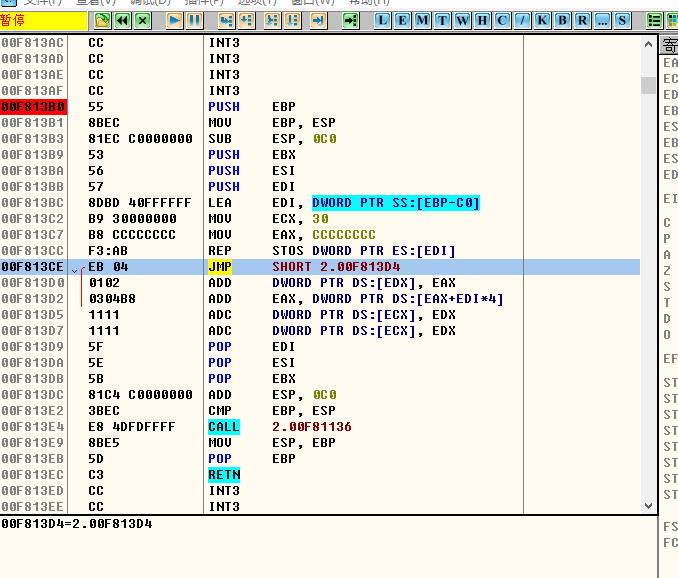
这样，call结束以后执行的下一条指令就是我们想要去的位置了，也就是花指令下面的正常的指令了

其实用call来做的话起到的也就是jmp的作用，道理是一样的，只不过手法不同

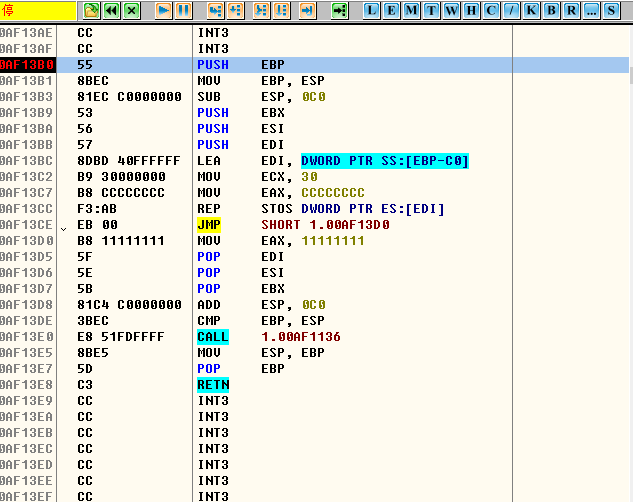
这个call ret配合的方法是我分析aspack看到的，不知道还有没有其他的花指令的方法，有的话再加上：）

这里有个找花指令的小技巧，当发现jmp或者call的地址在OD中没显示的话，就把jmp或者call指令的下一个指令nop掉吧，肯定是花指令，如果还没有显示的话就继续nop下一条，直到显示的指令和执行的指令一致，所以说，去花指令是个体力活，哈哈

补充一点，如果这个call不跟进去的话，貌似程序就没办法调试了，不知道怎么回事，不清楚原因



使用花指令



不使用花指令

测试代码

#include <windows.h>

void main()

{

\_asm

{

jmp l2

\_EMIT 0x1//这里就是花指令

\_EMIT 0x2//这里就是花指令

\_EMIT 0x3//这里就是花指令

\_EMIT 0x4//这里就是花指令

l2:

mov eax, 0x11111111

}

}

1.编译器不认识的指令，拆成机器码来写。

2.插入垃圾字节来反跟踪，又称花指令。

用emit就是在当前位置直接插入数据（实际上是指令），一般是用来直接  
  
插入汇编里面没有的特殊指令，多数指令可以用asm内嵌汇编来做，没有  
  
必要用emit来做，除非你不想让其它人看懂你的代码。

直接插入数据，而不是汇编的助记符

和[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)编程的db,dw效果相同

\_emit是伪指令

一个简单的例子

\_\_asm

{

-emit 0xEA

-emit 0xAA

-emit 0xAA

-emit 0xAA

-emit 0xAA

-emit 0x08

-emit 0x00

}

相当于

\_\_asm

{

jmp FAR 0x08:0xAAAAAAAA

}

花指令的原理就是

由于程序编译的时候，会优化asm代码，所以可以直接跳转，不受花指令的影响，但是花指令的代码仍然保存在程序中，导致了，编译器在反编译的时候，由于将垃圾数据当作是指令来解析，导致了接下来的解析出现了错误，导致了调试程序的时候，调试器会爆出异常。

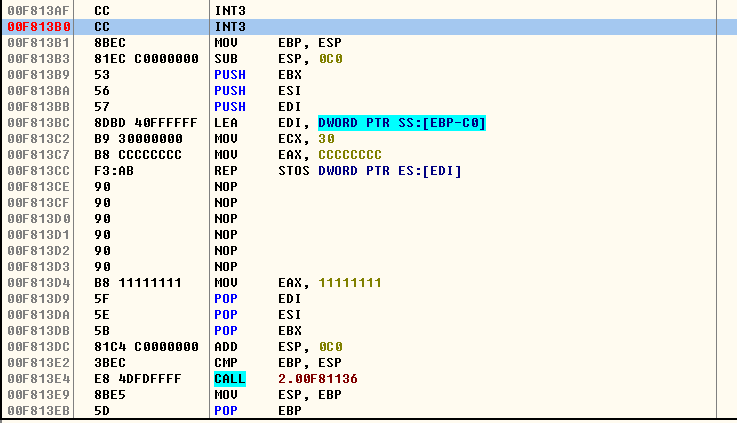
值得注意的是ida可以自动去除一些简单的花指令，ollydbg需要通过插件来去除。

花指令的原理是，如果当前指令调试器不能识别，那么调试器会继续往寻找和联合直到可以翻译为止，

去除花指令的猜想

正常编译程序，如果反汇编时，出现无效指令，那么这将其替换为nop。

去除花指令之后的截图



关于中断调试的一些思考

中断调试利用的是call方法在被调用时，call所在的位置会被压入栈中，压入栈中的作用是为了子程序完成后，可以继续执行主程序。此外，在栈中还会记录压入栈中的主程序的位置。

在主程序的首部设置断点，如果程序运行时，如果在当前这段程序之中进行校验。并作相应的逻辑判断，是否弹出对话框，那么在程序首部的断点应该会中终端，否则就说明逻辑判断程序一定在这个程序的父程序中，依次执行这个方法，直到出现中断为止。中断的程序就是值得去分析的程序了，可以爆破，分析程序的验证逻辑框架。

F2可以使用普通断点 shitf+F2 可以设置条件断点，只有满足了一定的条件才会中断，

条件断点的使用 edx==00401000

说明 1.比较的时候使用的是双等号

2.这个条件的含义是edx的值等于00401000才会中断，16进制。

Call中压栈的参数说明，由于参数是从右向左的顺序压栈，

到了函数的call有

Push ebp

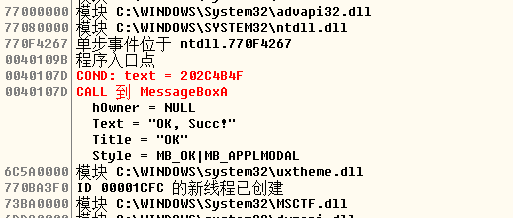
Mov ebp,esp

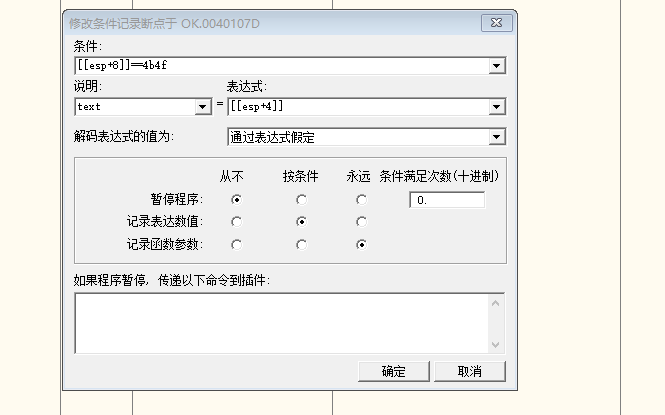
所以函数从左往右看 第一个参数在栈中的位置是esp+4，也可以是ebp+4，但是ebp一般指向栈底不动，所以标准的使用是 使用esp

条件断点在窗口中的使用，可以在窗口中的某个控件上设置，只有满足某个条件才能断下。

例如 [esp+4]=0x00401000,或者

STRING[[ESP+4]]==”abcdefg”



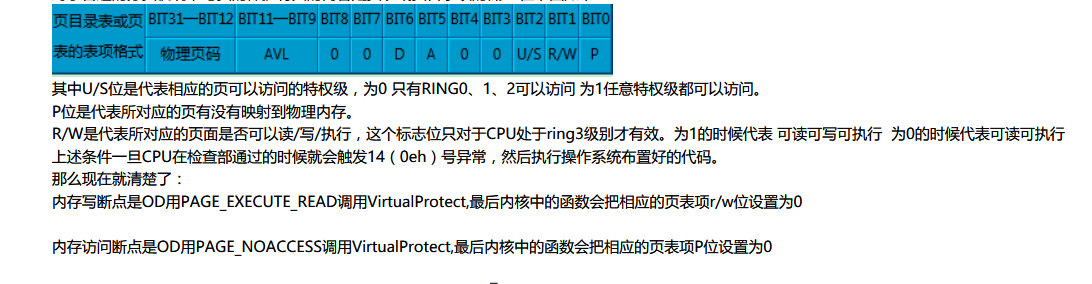


说明， 使用条件断点的时候，可以在满足条件的时候记录一些信息

条件记录的快捷键是shift+f4 ，也可以反汇编窗口右键—》断点—》条件记录

条件断点的快捷点是shift+f2.也可以反汇编窗口右键—》断点—》条件断点

使用快捷键alt+l查看条件记录断点的输出和其他断点的输出。



内存分页