

OllyDbg 完全教程

目录

第一章 概述.....	1
第二章 组件.....	5
一、一般原理 [General principles]	5
二、反汇编器 [Disassembler]	8
三、分析器 [Analysis]	9
四、Object 扫描器 [Object scanner]	12
五、Implib 扫描器 [Implib scanner]	12
第三章 OllyDbg 的使用.....	13
一、如何开始调试 [How to start debugging session]	13
二、CPU 窗口 [CPU window]	14
三、断点 [Breakpoints]	14
四、数据窗口 [Dump]	15
五、可执行模块窗口 [Executable modules window]	16
六、内存映射窗口 [Memory map window]	17
七、监视与监察器 [Watches and inspectors]	19
八、线程 [Threads]	19
九、调用栈 [Call stack]	20
十、调用树 [Call tree]	21
十一、选项 [Options]	21
十二、搜索 [Search]	22
十三、自解压文件 [Self-extracting (SFX) files]	22
十四、单步执行与自动执行 [Step-by-step execution and animation]	23
十五、Hit 跟踪 [Hit trace]	23
十六、Run 跟踪 [Run trace]	24
十七、快捷键.....	26
十八、插件 [Plugins]	29
十九、技巧提示 [Tips and tricks]	29
第四章 其他功能.....	30
一、调试独立的 DLL [Debugging of stand-alone DLLs]	30
二、解码提示 [Decoding hints]	32
三、表达式赋值 [Evaluation of expressions]	32
四、自定义函数描述 [Custom function descriptions]	34

第一章 概述

OllyDbg 是一种具有可视化界面的 32 位汇编—分析调试器。它的特别之处在于可以在没有源代码时解决问题，并且可以处理其它编译器无法解决的难题。

Version 1.10 是最终的发布版本。这个工程已经停止，我不再继续支持这个软件了。但不用担心：全新打造的 OllyDbg2.00 不久就会面世！

运行环境：OllyDbg 可以在任何采用奔腾处理器的 Windows 95、98、ME、NT 或是 XP（未经完全测试）操作系统中工作，但我们强烈建议你采用 300—MHz 以上的奔腾处理器以达到最佳效果。还有，OllyDbg 是极占内存的，因此如果你需要使用诸如追踪调试 [Trace] 之类的扩展功能话，建议你最好使用 128MB 以上的内存。

支持的处理器：OllyDbg 支持所有 80x86、奔腾、MMX、3DNOW!、Athlon 扩展指令集、SSE 指令集以及相关的数据格式，但是不支持 SSE2 指令集。

配置：有多达百余个选项用来设置 OllyDbg 的外观和运行。

数据格式：OllyDbg 的数据窗口能够显示的所有数据格式：HEX、ASCII、UNICODE、16/32 位有/无符号/HEX 整数、32/64/80 位浮点数、地址、反汇编 (MASM、IDEAL 或是 HLA)、PE 文件头或线程数据块。

帮助：此文件中包含了关于理解和使用 OllyDbg 的必要的信息。如果你还有 Windows API 帮助文件的话（由于版权的问题 **win32.hlp** 没有包括在内），你可以将它挂在 **OllyDbg** 中，这样就可以快速获得系统函数的相关帮助。

启动：你可以采用命令行的形式指定可执行文件、也可以从菜单中选择，或直接拖放到 OllyDbg 中，或者重新启动上一个被调试程序，或是挂接 [Attach] 一个正在运行的程序。OllyDbg 支持即时调试。OllyDbg 根本不需要安装，可直接在软盘中运行！

调试 DLLs：你可以利用 OllyDbg 调试标准动态链接库 (DLLs)。OllyDbg 会自动运行一个可执行程序。这个程序会加载链接库，并允许你调用链接库的输出函数。

源码级调试：OllyDbg 可以识别所有 Borland 和 Microsoft 格式的调试信息。这些信息包括源代码、函数名、标签、全局变量、静态变量。有限度的支持动态（栈）变量和结构。

代码高亮：OllyDbg 的反汇编器可以高亮不同类型的指令（如：跳转、条件跳转、入栈、出栈、调用、返回、特殊的或是无效的指令）和不同的操作数（常规 [general]、FPU/SSE、段/系统寄存器、在栈或内存中的操作数，常量）。你可以定制个性化高亮方案。

线程：OllyDbg 可以调试多线程程序。因此你可以在多个线程之间转换，挂起、恢复、终止线程或是改变线程优先级。并且线程窗口将会显示每个线程的错误（就像调用 GETLASTERROR 返回一样）。

分析：OllyDbg 的最大特点之一就是分析。它会分析函数过程、循环语句、选择语句、表 [tables]、常量、代码中的字符串、欺骗性指令 [tricky constructs]、API 调用、函数中参数的数目，import 表等等。这些分析增加了二进制代码的可读性，减少了出错的可能性，使得我们的调试工作更加容易。

Object 扫描: OllyDbg 可以扫描 Object 文件/库 (包括 OMF 和 COFF 格式), 解压代码段 [code segments] 并且对其位置进行定向。

Implib 扫描: 由于一些 DLL 文件的输出函数使用的索引号, 对于人来说, 这些索引号没有实际含义。如果你有与 DLL 相应的输入库 [import library], OllyDbg 就可以将序号转换成符号名称。

完全支持 Unicode: 几乎所有支持 ASCII 的操作同时也支持 UNICODE, 反之亦然。

名称: OllyDbg 可以根据 Borland 和 Microsoft 格式的调试信息, 显示输入/输出符号及名称。Object 扫描器可以识别库函数。其中的名称和注释你可任意添加。如果 DLL 中的某些函数是通过索引号输出的, 则你可通过挂接输入库 [import library] 来恢复原来的函数名称。不仅如此, OllyDbg 还能识别大量的常量符号名 (如: 窗口消息、错误代码、位域 [bit fields] ...) 并能够解码为已知的函数调用。

已知函数: OllyDbg 可以识别 2300 多个 C 和 Windows API 中的常用函数及其使用的参数。你可以添加描述信息、预定义解码。你还可以在已知函数设定 Log 断点并可以对参数进行记录。

函数调用: OllyDbg 可以在没有调试信息或函数过程使用非标准的开始部分 [prolog] 和结尾部分 [epilog] 的情况下, 对递归调用进行回溯。

译者注:

```
004010D0  push  ebp                \
004010D1  mov   ebp,esp            |
004010D3  sub   esp,10h           |>prolog
004010D6  push  ebx                |
004010D7  push  esi                |
004010D8  push  edi                /
.....
004010C5  pop   edi                \
004010C6  pop   esi                |
004010C7  pop   ebx                |>epilog
004010C8  mov   esp,ebp           |
004010CA  pop   ebp                |
004010CB  ret                               /
```

栈: 在栈窗口中, OllyDbg 能智能识别返回地址和栈框架 [Stack Frames]。并会留下一些先前的调用。如果程序停在已知函数上, 堆栈窗口将会对其参数进行分析解码。

译者注: 栈框架 [Stack Frames] 是指一个内存区域, 用于存放函数参数和局部变量。

SEH 链: 跟踪栈并显示结构化异常句柄链。全部链会显示在一个单独的窗口中。

搜索: 方法真是太多了! 可精确、模糊搜索命令或命令序列, 搜索常数, 搜索二进制、文本字符串, 搜索全部命令地址, 搜索全部常量或地址域 [address range], 搜索所有能跳到选定地址的跳转, 搜索所有调用和被调用的函数, 搜索所有参考字符串, 在不同模块中搜索所有调用、搜索函数名称, 在全部已分配的内存中搜索二进制序列。如果搜索到多个结果, 你可以对其进行快速操作。

窗口: OllyDbg 能够列出关于调试程序中的各种窗口, 并且可以在窗口、类甚至选定的消息上设置断点。

资源: 如果 Windows API 函数使用了参考资源串, OllyDbg 可以显示它。其支持显示的类型仅限于附带资源 [attached resources] 的列表、数据显示及二进制编辑。

断点: OllyDbg 支持各种断点: 一般断点、条件断点、记录断点 (比如记录函数参数到记录窗口)、内存读写断点、硬件断点 (只适用于 ME/NT/2000) 等。在 Hit 跟踪情况下, 可以在模块的每条命令上都设置 INT3 断点。在使用 500—MHZ 处理器的 Windows NT 中, OllyDbg 每秒可以处理高达 5000 个中断。

监视与监察器: 每个监视都是一个表达式并能实时显示表达式的值。你可以使用寄存器、常数、地址表达式、布尔值以及任何复杂代数运算, 你还可以比较 ASCII 和 UNICODE 字符串。监察器 [inspectors] 是一种包含了两个的索引序列的监视 [Watches], 它以二维表的形式呈现, 可以对数组和结构进行解码分析。

Heap walk.: 在基于 Win95 的系统中, OllyDbg 可以列出所有的已分配的堆。

句柄: 在基于 NT 的系统中, OllyDbg 可列出被调试程序的所有系统句柄。

执行: 你可以单步执行、步入子程序或者步过子程序。你也可以执行程序直到函数返回时、执行到指定地址处, 还可以自动执行。当程序运行时, 你仍然可以操纵程序并能够查看内存、设置断点甚至修改代码。你也可以任意的暂停或重启被调试的程序。

Hit 跟踪: Hit 跟踪可以显示出目前已执行的指令或函数过程, 帮助你检验代码的各个分支。Hit 跟踪会在指定指令到达之前设置断点, 而在这个指令执行后, 会把这个断点清除掉。译者注: Hit 在英文中是“击中”的意思, 指令如果运行了就表示这个指令被“击中”了, 没有执行的指令就是“未击中”, 这样我们就很容易看出被调试程序哪些部分运行了, 而哪些没有运行。

Run 跟踪: Run 跟踪可以单步执行程序, 它会在一个很大的循环缓冲区中模拟运行程序。这个模拟器包含了除了 SSE 指令集以外的所有寄存器、标志、线程错误、消息、已经函数的参数。你可以保存命令, 这样可以非常方便地调试自修改代码 (译者注: 比如加壳程序)。你可以设置条件中断, 条件包括地址范围、表达式、命令。你可以将 Run 跟踪信息保存到一个文件中, 这样就可以对比两次运行的差别。Run 跟踪可以回溯分析已执行过的上百万条命令的各种细节。

统计: 统计 [Profiler] 可以在跟踪时计算某些指令出现的次数。因此你就能了解代码的哪一部分被频繁执行。

补丁: 内置汇编器能够自动找到修改过的代码段。二进制编辑器则会以 ASCII、UNICODE 或者十六进制的形式同步显示修改后的数据。修改后的数据同其它数据一样, 能够进行复制—粘贴操作。原来的数据会自动备份, 以便数据恢复时使用。你可以把修改的部分直接复制到执行文件中, OllyDbg 会自动修正。OllyDbg 还会记录以前调试过程中使用的所有补丁。你可以通过空格键实现补丁的激活或者禁止。

自解压文件: 当调试自解压文件时, 你往往希望跳过解压部分, 直接停在程序的原始入口点。OllyDbg 的自解压跟踪将会使你实现这一目的。如果是加保护的自解压段, 自解压跟踪往往会失败。而一旦 OllyDbg 找到了入口点, 它将会跳过解压部分, 并准确的到达入口点。

插件：你可以把自己的插件添加到 OllyDbg 中，以增加新的功能。OllyDbg 的插件能够访问几乎所有重要的数据的数据的结构、能够在 OllyDbg 的窗口中添加菜单和快捷键，能够使用 100 个以上的插件 API 函数。插件 API 函数有详细的说明文档。默认安装已经包含了两个插件：命令行插件和书签插件。

UDD：OllyDbg 把所有程序或模块相关的信息保存至单独的文件中，并在模块重新加载时继续使用。这些信息包括了标签、注释、断点、监视、分析数据、条件等等

更多：这里介绍的功能，仅仅是 OllyDbg 的部分功能。因为其具有如此丰富的功能，以至于 OllyDbg 能成为非常方便的调试器！

第二章 组件

一、一般原理 [General principles]

我希望你能对 80x86 系列处理器的内部结构有所了解,同时具有一定的编写汇编程序的能力。对于 Microsoft Windows 方面的知识,你也要熟悉。

OllyDbg 是运行在 Windows 95、Windows 98、Windows ME、Windows NT 和 Windows 2000 系统下的一个单进程、多线程的分析代码级调试工具。它可以调试 PE 格式的执行文件及动态链接库,并可以对其打补丁。“代码级”意味着你可以直接与比特、字节或处理器指令打交道。OllyDbg 仅使用已公开的 Win32 API 函数,因此它可以在所有 Windows 操作系统及后继版本中使用。但是由于我没有对 XP 系统进行彻底测试,因此不能保证 OllyDbg 功能的充分发挥。注意:OllyDbg 不支持对 .NET 程序的调试。

OllyDbg 不是面向编译器的。它没有特别的规则规定必须是哪一个编译器产生的代码。因此,OllyDbg 可以非常好的处理通过编译器生成的代码,或是直接用汇编写入的代码。

OllyDbg 可以并行调试程序。你无须暂停执行程序,就可以浏览代码和数据,设置断点、停止或恢复线程,甚至直接修改内存。(这可以视为一种软件调试的模式,与之相对的硬件模式则是当进程在运行时调试器被阻滞,反之亦然)。假使所需的操作比较复杂,OllyDbg 会让进程终止一小段时间,但是这种暂停对于用户来说是透明的。有时进程会发生非法操作。你可以把 OllyDbg 设置成即时 [just-in-time] 调试器,它会挂接出错程序,并停在程序产生异常的地方。

通过 OllyDbg,你可以调试单独的 DLL [standalone DLLs] 文件。操作系统不能直接运行 DLL 文件,因此 OllyDbg 将一个可以加载 DLL 的小程序压缩到资源里,这个程序允许你调用最多 10 个参数的输出函数。

OllyDbg 是完全面向模块 [module-oriented] 的。模块 [Module] 包括可执行文件 (扩展名通常为 .EXE) 和在启动时加载或需要时动态加载的动态链接库 (扩展名通常为 .DLL)。在调试期间,你可以设置断点 [breakpoints]、定义新的标签 [labels]、注释 [comment] 汇编指令,当某个模块从内存中卸载 [unload] 时,调试器会把这些信息保存在文件中,文件名就是模块的名称,扩展名为 .UDD (表示 用户自定义文件 [User-Defined Data]) 当 **OllyDbg** 下一次加载该模块时,它会自动恢复所有的调试信息,而不管是哪一个程序使用这个模块。假设你正在调试程序 Myprog1,这个程序使用了 Mydll。你在 Mydll 中设置了一些断点,然后你开始调试 Myprog2,这个程序同样使用了 Mydll。这时你会发现,所有 Mydll 中的断点依然存在,即使 Mydll 加载到不同的位置!

一些调试器把被调试进程的内存当作一个单一的 (并且大部分是空的) 大小为 2^{32} 字节的区域。OllyDbg 采用了与之不同的技术:在这里,内存由许多独立的块组成,任何对内存内容的操作都被限制在各自的块内。在大多数情况下,这种方式工作得很好并且方便了调试。但是,如果模块包含好几个可执行段 [executable sections],你将不能一次看到全部代码,然而这种情况是非常少见的。

OllyDbg 是一个很占用内存的程序 [memory-hungry application]。它在启动时就需要 3 MB,并且当你第一次装载被调试的程序时还需要一到两兆的内存。每一次的分析、备份、

跟踪或者文件数据显示都需要占用一定的内存。因此当你调试一个很大的项目，发现程序管理器显示有 40 或 60 兆内存被占用时，请不要惊慌。

为了有效地调试一些不带源码的程序，你必须首先理解它是如何工作的。OllyDbg 包含的大量特性可以使这种理解变得非常容易。

首先，OllyDbg 包含一个内置的代码分析器。分析器遍历整个代码，分出指令和数据，识别出不同的数据类型和过程，分析出标准 API 函数（最常用的大约有 1900 个）的参数并且试着猜出未知函数的参数数目。你也可以加入自己的函数说明 [your own function descriptions]。它标记出程序入口点和跳转目的地，识别出跳转表 [table-driven switches] 和指向字符串的指针，加入一些注释，甚至标示出跳转的方向等等。在分析结果的基础上，调用树 [call tree] 显示哪些函数被指定过程调用（直接或间接）并且识别出递归调用、系统调用和叶子过程 [leaf procedures]。如果需要的话，你可以设置解码提示 [decoding hints] 来帮助分析器解析那些不明确的代码或数据。

OllyDbg 还包含 Object 扫描器 [Object Scanner]。如果你有库文件 [libraries] 或目标文件 [object files]，扫描器会在被调试的程序中定位这些库函数。在全部函数调用中，对标准函数的调用占很重要的一部分（据我估计可达 70%）。如果你知道正要被调用的函数的功能，你就不必把注意力集中在这个函数上，可以简单地单步步过 [step over] 这个 call。分析器知道 400 多个标准 C 函数，比如 fopen 和 memcpy。然而我必须承认当前版本的 OllyDbg 不能定位很短的函数（比一个 return 命令多不了多少的）或相似的函数（只在重定位上有不同）。Object 扫描器 [Object scanner] 也能够识别输入库 [import libraries]。如果某个 DLL 是按序号输出的，你不会看到函数名，只会发现一堆无意义的神秘数字。这种 DLL 的开发者通常会提供一个输入库来实现函数符号名与序号间的对应。让 OllyDbg 使用这个输入库，它就会恢复原始的函数符号名。

面向对象的语言（如 C++），使用了一种叫做名称修饰 [name mangling] 的技术，把函数类型和参数都加入函数名中。OllyDbg 可以解码 [demangle] 这种函数名，使程序更易读。译者注：C++ 的名称修饰是编译器将函数的名称转变成为一个唯一的字符串的过程，这个字符串会对函数的类、其命名空间、其参数表，以及其他等等进行编码。C++ 的名称修饰适用于静态成员函数，也适用于非静态成员函数。静态函数的名称修饰的一个好处之一，是能够在不同的类里使用同一个名称来声明两个或者更多的静态成员函数——而不会发生名称上的冲突。

OllyDbg 完全支持 UNICODE，几乎所有对 ASCII 字符串的操作都可以同样应用于 UNICODE。

汇编指令都是很相似的。你经常会搞不清自己是不是已经跟踪过某一段代码。在 OllyDbg 中你可以加入自己的标签 [labels] 和注释 [comments]。这些极大地方便了调试。注意一旦你注释了某个 DLL，以后每次加载这个 DLL 时，注释和标签都有效——尽管你在调试不同的程序。

OllyDbg 可以跟踪标准的栈帧 [stack frames]（由 PUSH EBP; MOV EBP,ESP 所创建的）。现代编译器有禁止产生标准栈框架的选项，在这种情况下分配栈 [stack walk] 是不可能的。当程序运行到已知的函数时，栈窗口 [stack window] 解析它的参数，调用栈 [Call stack] 窗口显示到达当前位置所调用函数的序列。

现代的面向对象应用程序广泛地使用了一种叫做结构化异常处理 [Structured Exception Handling, SHE] 的技术。SHE 窗口 [SEH window] 可以显示异常处理链。

多种不同的搜索 [search] 选项可以让你找到二进制代码或数据、命令或命令序列、常量或字符串、符号名或在 Run 跟踪中的一条记录。

对于任何地址或常量，OllyDbg 可以找出参考 [referencing] 到该地址或常量的全部命令的列表。然后你可以在这个列表里找出对你来说是重要的参考。举例来说，某个函数可能被直接调用，或者经过编译器优化后把地址放入寄存器间接调用，或者把地址压入堆栈作为一个参数————没问题，OllyDbg 会找出所有这样的地方。它甚至能找到并列列出所有和某个指定的位置有关的跳转。

OllyDbg 支持所有标准类型的断点 [breakpoints] ————非条件和条件断点、内存断点（写入或访问）、硬件断点或在整个内存块上下断点（后两项功能只在 Window ME, NT, 2000, XP 中有效）。条件表达式可以非常复杂（“当 [ESP+8] 的第 2 位被设置，并且 123456 位置处的字 [word] 小于 10，或者 EAX 指向一个以“ABC”开头的 UNICODE 字符串，但跳过前 10 次断点而在第 11 次中断”）。你可以设定一条或多条指令，当程序暂停时由 OllyDbg 传递给插件 [plugins]。除了暂停，你还可以记录某个表达式的值（可以带有简短的说明），或者记录 OllyDbg 已知的函数的参数。在 Athlon 2600+、Windows 2000 环境下，OllyDbg 可以每秒处理多达 25000 个条件断点。

另一个有用的特性是跟踪。OllyDbg 支持两种方式的跟踪：hit 和 run。

在第一种情况下，它对指定范围内的每条指令上设置断点（比如在全部可执行代码中）。当到达设断的指令后，OllyDbg 清除断点并且把该指令标记为 hit。这种方法可以用来检测某段代码是否被执行。Hit 跟踪速度惊人的快，在一个很短时间的启动后程序几乎达到了全速（译者注：这应该是与不进行调试时速度相比而言）。因为 INT3 断点可能对数据有灾难性的影响，所以我建议不要使用模糊识别过程。**当代码没有被分析时 Hit 跟踪是不可以使用的。**

Run 跟踪 [Run trace] 是一步一步地执行程序，同时记录精确的运行历史和所有寄存器的内容、已知的参数和可选的指令（当代码是自修改时会有帮助）。当然，这需要大量的内存（每个指令需要 15 至 50 个字节，取决于调试的模式）但是可以精确地回溯和分析。你可以只在选定的一段代码甚至是一条指令中进行 Run 跟踪，或者你可以跳过无关紧要的代码。对于每个地址，OllyDbg 能够计算这个地址在 Run 跟踪日志中出现的次数，虽然会导致执行缓慢但是可以得到代码执行的统计。比如说，某命令让你在每个已识别的过程入口处进行 Run 跟踪，那么统计 [profile] 就会给你每个过程被调用的次数。在到达某条指令、某个地址范围或指令计数器达到某一数值时 Run 跟踪可以自动地暂停 [pause]。

在多线程程序里 OllyDbg 可以自动管理线程 [threads]，如果你单步调试或跟踪程序，它会自动恢复当前线程而挂起其它线程。如果你运行程序，OllyDbg 会恢复先前的线程状态。

你可以为内存块建立快照（叫做备份）。OllyDbg 会高亮显示所有的改动。你可以把备份保存到文件或从文件中读取出来，从而发现两次运行的不同之处。你可以查看备份，搜索下一处改动，恢复全部或选定的改动。补丁管理器 [Patch manager] 记录了上次应用到程序中的所有补丁，在下次调试时可以再次应用它们。你可以很容易地把你的补丁加在可执行文件上。OllyDbg 会自动进行修正。

你不能在带有 Win32 的 16 位 Windows 下使用 OllyDbg。这种 32 位扩展操作系统无

法实现某些必需的调试功能。你既不能调试 **DOS** 程序也不能调试 **16 位 NE (New Executable)** 格式文件，我也没有打算在未来的版本中支持这些。

二、反汇编器 [Disassembler]

反汇编器识别所有的标准 80x86、保护、FPU、MMX 和 3DNow!指令集（包括 Athlon 扩展的 MMX 指令集）。但它不识别 ISSI 命令，尽管计划要在下个版本中支持这种命令。某些过时或者未公开的命令，像 LOADALL，也不支持。

反汇编器可以正确解码 16 位地址。但它假设所有的段都是 32 位的(段属性使用 32 位)。这对于 PE [Portable Executable] 格式文件总是正确的。OllyDbg 不支持 16 位的 NE [New Executables] 格式。

如果你熟悉 MASM 或者 TASM，那么反汇编的代码对于你没有任何问题。但是，一些特例也是存在的。以下命令的解码与 Intel 的标准不同：

AAD (ASCII Adjust AX Before Division)—该命令的解码后的一般形式为：AAD imm8

AAM (ASCII Adjust AX After Multiply)—该命令(非十进制数)的一般解码形式为：AAM imm8

SLDT (Store Local Descriptor Table register)—操作数总被解码为 16 位。这个命令的 32 位形式会在目的操作数的低 16 位中存储段选择器，并保留高 16 位不变。

SALC (Sign—extend Carry bit to AL, undocumented)—OllyDbg 支持这个未公开指令。

PINSRW (Insert Word From Integer Register, Athlon extension to MMX)—在 AMD 的官方文档中，这个命令的内存形式使用了 16 位内存操作数；然而寄存器形式需要 32 位寄存器，但只使用了低 16 位。为了方便处理，反汇编器解码寄存器为 16 位形式。而汇编器两种形式都支持。

CVTTPS2PI and CVTTTPI (Convert Packed Single — Precision Floating to Packed Doubleword, Convert with Truncation Packed Single—Precision Floating to Packed Doubleword)—在这些命令中，第一个操作数是 MMX 寄存器，第二个或者是 128 位 XMM 寄存器或者是 64 位内存区域。为了方便处理，内存操作数也被解码为 128 位。

有些指令的助记符要依赖操作数的大小：

不分大小的形式	明确的 16 位形式	明确的 32 位形式
PUSHA	PUSHAW	PUSHAD
POPA	POPAW	POPAD
LOOP	LOOPW	LOOPD
LOOPE	LOOPWE	LOOPDE
LOOPNE	LOOPWNE	LOOPDNE
PUSHF	PUSHFW	PUSHFD
POPF	POPFW	POPFD
IRET	IRETW	IRETD

你可以改变解码大小敏感助记符 [decoding of size—sensitive mnemonics]。根据选项，

反汇编器从三种可能中选择之一进行解码。这个选项也会影响汇编器的默认处理方式。解码 MMX 和 3DNow! 指令总是开启的，尽管你的处理器并不支持这些指令。

三、分析器 [Analysis]

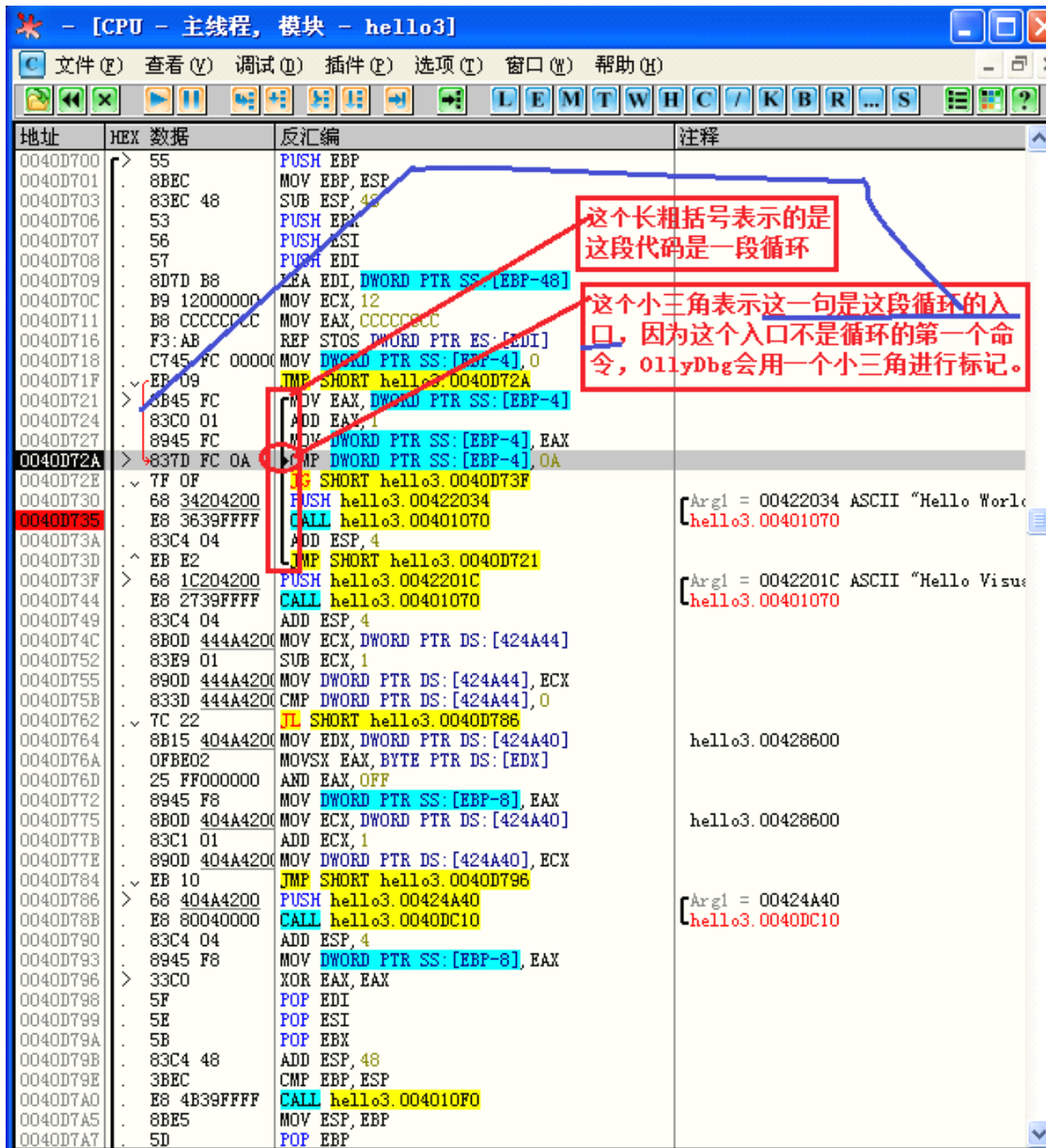
OllyDbg 整合了一个快速而强大的代码分析器。你可以从快捷菜单，或者在 CPU 窗口的反汇编面板中按 **Ctrl+A**，或者在可执行模块中选择“分析全部模块 [Analyze all modules]”，来使用它。

分析器有很高的启发性。它能区分代码和数据，标记入口和跳转目的地址，识别转换表 [switch tables]，ASCII 和 UNICODE 串，定位函数过程，循环，高阶转换 [high-level switches] 并且能解码标准 API 函数的参数（示例 [example]）。OllyDbg 的其他部分也广泛的使用了分析后的数据。

这是如何实现的？我将为你揭开这一神秘面纱。第一遍，OllyDbg 反汇编代码段中所有可能的地址，并计算调用的每个目的地址的个数。当然，很多调用是假的，但不可能两个错误的调用都指向了相同的命令，当然如果有三个的话，就更不可能了。因此如果有三个或者更多的调用指向了相同的地址，我可以肯定的说这个地址是某个频繁使用的子程序的入口。从定位的入口出发，我继续跟踪所有的跳转和函数调用，等等。按这种方法，我可能准确定位 99.9% 的命令。但是，某些字节并不在这个链条上。我再用 20 多种高效的启发方法（最简单的方法，比如“直接访问前 64K 内存是不允许的，像在 MOV [0],EAX 中”）来探测他们。有时，分析器在你感兴趣的地方分析错误。有两种解决方法：或者从选中的部分移除分析（**快捷键退格键**），这样 OllyDbg 将使用默认的解码（反汇编）方式；或者设置解码提示 [decoding hints] 并重新分析。注意：在某些情况下，当分析器认为你的提示是不合适的，或者有冲突，则可能忽略你的设置。

探测程序的函数过程也很简单。在分析器眼中看来，程序只是一个连绵不断的代码，从一个入口开始，可能达到（至少从理论上）所有的命令（除了 NOP 以及类似的用于填充间隙的命令）。你可能指定三个识别级别。严格的函数过程要求有准确的一个入口，并且至少有一个返回。在启发级别下，分析器只要求过程有一个入口。而如果你选择模糊模式，差不多连贯的代码都会被识别为单独的过程。现代编译器进行全局代码优化，有可能把一个过程分成几个部份。在这种情况下，模糊模式非常有用。但是也会误识别的机率也就更高。

同样地，循环是一个封闭的连续的命令序列，并有一个到开始处的跳转作为一个入口，还有若干个出口。循环与高级操作命令 do, while 和 for 相对应。**OllyDbg 能够识别任何复杂的嵌套循环。他们会在反汇编栏 [Disassembly] 中用长而粗括号标记。如果入口不是循环的第一个命令，OllyDbg 会用一个小三角进行标记。**



为了实现一个转换 [switch]，许多编译器，读取转换变量 [switch variable] 到寄存器中，然后减它，像如下的代码序列：

```

MOV EDX,<switch variable>
SUB EDX,100
JB DEFAULTCASE
JE CASE100          ; Case 100
DEC EDX
JNE DEFAULTCASE
...                ; Case 101

```

这个序列可能还包含一到两阶的转换表、直接比较、优化和其他元素。如果在比较或跳转的很深处，这就很难知道哪是一个分支 [Case]。OllyDbg 会帮助你，它会标记所有的分支，包括默认的，甚至尝试分析每个分支的含义，如 'A'、WM_PAINT 或者

EXCEPTION_ACCESS_VIOLATION。如果命令序列没有修改寄存器（也就是仅仅由比较组成），那么这可能不是转换，而很有可能是选择嵌套：

```
if (i==0) {...}
else if (i==5) {...}
else if (i==10) {...}
```

如果需要 OllyDbg 将选择嵌套解码成选择语句，请在分析 1 [Analysis1] 中设置相关选项。

OllyDbg 包含多达 1900 条常用 API 函数，这些都作为内部预处理资源。这个列表包含了 KERNEL32, GDI32, USER32, ADVAPI32, COMDLG32, SHELL32, VERSION, SHLWAPI, COMCTL32, WINSOCKET, WS2_32 和 MSVCRT。你可以添加自己的函数描述 [add your own descriptions]。如果分析器遇到的调用，使用了已知的函数名（或者跳转到这样的函数），它将在调用之前立即解码 PUSH 命令。因此，你只需略微一看就能明白函数调用的含义。OllyDbg 还包含了大约 400 多种的标准 C 函数。如果你有原始的库文件，我推荐你在分析前扫描目标文件。这样 OllyDbg 将能解码这些 C 函数的参数。

如果选项“猜测未知函数的参数个数”开启，分析器将会决定这个调用函数过程使用的长度为双字的参数个数。并且标记他们为参数 1 [Arg1]，参数 2 [Arg2]，等等。注意：无论如何，寄存器参数是无法识别的，所以不会增加参数的数目。分析器使用了一种比较安全的方法。例如，它不能识别的没有参数的函数过程，或者该过程 POP 命令直接做返回前的寄存器恢复，而不销毁参数。然而，识别出来的函数参数数目通常非常高，这大大加大了代码的可读性。

分析器能够跟踪整型寄存器的内容。现代优化编译器，特别是奔腾系列，频繁地使用寄存器读取常量和地址，或使用尽量少的使用内存。如果某个常量读取到寄存器中，分析器会注意它，并尝试解码函数和其参数。分析器还能完成简单的算术计算，甚至可以跟踪压栈和出栈。

分析器不能区分不同类的名称 [different kinds of names]。如果你将某些函数指定为已知的名称，OllyDbg 将会解码所有到该地址的调用。这是几个预定义的特殊名称 WinMain, DllEntryPoint and WinProc。你可能使用这些标签标记主程序、DLL 的入口以及窗口过程（注意：OllyDbg 不检查用户自定义的标签是否唯一）。另外，假定预定义参数 assume predefined arguments 是一种更好的方法，不幸的是，没有一般规则能够做到 100% 的准确分析。在某些情况下，例如当模块包含了 P-Code 或代码段中包换了大量的数据，分析器可能将一些数据解释成代码。如果统计分析显示代码部分很可能是压缩包或者经过加密了，分析器会发出警告。**如果你想使用 Hit 跟踪 [Hit trace]，我建议你不要使用模糊分析 [fuzzy analysis]，因为设置断点的地方可能正是数据部分。**

自解压文件 [Self-extractable files] 通常有一个自提取器，在“正式”代码段之外。如果你选择自解压选项 [SFX option] 中的“扩展代码段，包含提取器 [Extend code section to include self-extractor]”，OllyDbg 将会扩展代码段，形式上允许分析它，并可以使用 Hit 跟踪 [Hit trace] 和 Run 跟踪 [Run trace]。

四、Object 扫描器 [Object scanner]

扫描器将特定的目标文件或者目标库（包括 OMF 和 COFF 两种格式），提取出代码段，然后将这些段定位在当前模块的代码节 [Code section] 中。如果段定位好了，扫描器将从目标文件中的调试信息提取名称（也就是所谓的库标签 [library labels]）。这极大的增加了代码与数据的可读性。扫描器并不会对已识别的目标文件进行标签匹配，所以它不能识别非常小或相似的函数（比如：两个函数只是在重定位有区别）。因此要经常检查扫描器发送到登陆窗口的警告列表！

五、Implib 扫描器 [Implib scanner]

某些 DLL 的输出符号仅仅是一个序号。许多符号都是井号加数字（比如：MFC42.#1003），这非常不便于理解。幸运的是，软件零售商提供了输入连接库（implibs），它与序号符号名相关。

使用 implib 扫描器的方法：从主菜单中选择**调试** [Debug] 一>选择**输入链接库** [Select import libraries]。当你加载应用程序时，OllyDbg 会读取链接库并从内置表格 [internal tables] 中提取符号名。每次遇到序号符号，而对应的链接库已经注册到 OllyDbg 中时，这个序号符号会被替换。

第三章 OllyDbg 的使用

一、如何开始调试 [How to start debugging session]

最简单的方法是：运行 OllyDbg，点击菜单上的文件 [File] → 打开 [Open]，选择你想调试的程序。如果程序需要命令行参数，你可以在对话框底部的输入栏中，输入参数或者选择以前调试时输入过的一条参数。

OllyDbg 能够调试独立的 DLL [stand-alone DLLs]。在这种情况下，OllyDbg 会创建并运行一个小的应用程序来加载链接库并根据你的需要调用输出函数。

如果你想重新启动上一次调试的程序，只要按一下 Ctrl+F2（这是重启程序的快捷键（???）），这样 OllyDbg 会以同样的参数运行这个程序。另一种做法是在菜单中选择文件 [File]，从历史列表中选择程序。你也可以在 Windows 资源管理器中将可执行文件或 DLL 文件拖拽到 OllyDbg 中。

当然，你可以在 OllyDbg 启动时，运行指定带有运行参数的被调试程序。例如：你可以在桌面创建一个 OllyDbg 的快捷方式，右击并选择“属性”，在“快捷方式”中的“目标”中添加调试的程序的全路径。这样，你每次双击快捷方式时，OllyDbg 将自动运行被调试程序。注意：DLL 文件不支持这种方式。

你可以把正在运行的进程挂接到 OllyDbg 中。在菜单中打开文件 [File] → 挂接 [Attach]，从进程列表中选择要挂接的进程。注意：在你关闭 OllyDbg 的同时，这个进程也会被关闭。不要挂接系统进程，否则可能会导致整个操作系统的崩溃。（事实上在大多数情况下，操作系统禁止你挂接敏感进程）。

OllyDbg 可以作为即时 [just-in-time] 调试器。这需要在系统注册表中注册。在菜单中选择选项 [Options] → 即时调试 [Just-in-time debugging] 并在弹出的对话框中单击按钮“设置 OllyDbg 为即时调试器” [Make OllyDbg just-in-time debugger]。今后，如果某个应用程序发生了非法操作，系统将提示你是否用 OllyDbg 调试这个程序。操作系统会启动 OllyDbg 并直接停在发生异常的地方。如果你选择了“挂接时不询问” [attaching without confirmation]，则在即时调试时 OllyDbg 不会弹出询问对话框。如果想恢复成以前的即时调试器 [Restore old just-in-time debugger]，按相应的按钮即可。

另一种方法是把 OllyDbg 添加到与可执行文件关联的快捷菜单中（这个想法是 Jochen Gerster 提出的）。在主菜单中，选择选项 [Options] → 添加到资源管理器中 [Add to Explorer]。以后你可以在所有的文件列表中，右击可执行文件或 DLL，在快捷菜单中选择 OllyDbg。这个功能会创建四个注册表键值：

```
HKEY_CLASSES_ROOT\exefile\shell\Open with OllyDbg
HKEY_CLASSES_ROOT\exefile\shell\Open with OllyDbg\command
HKEY_CLASSES_ROOT\dllfile\shell\Open with OllyDbg
HKEY_CLASSES_ROOT\dllfile\shell\Open with OllyDbg\command
```

OllyDbg 能够调试控制台程序（基于文字的）。

OllyDbg 不能调试 .NET 应用程序。.NET 程序是由微软的中间语言这种伪指令组成的，或是 on-the-fly to native ?6 commands 编译的。

注意：如果你运行的是 Windows NT、2000 或 XP 操作系统，你应该拥有管理员权限以便能够调试程序。。

二、CPU 窗口 [CPU window]

对于用户来说，CPU 窗口在 OllyDbg 中是最重要的窗口。你调试自己程序的绝大部分操作都要在这个窗口中进行。它包括以下五个面板（这五个面板的大小都是可以调节的）：

- 反汇编 [Disassembler]
- 信息 [Information]
- 数据 [Dump]
- 寄存器 [Registers]
- 栈 [Stack]

按 TAB 键，可以切换到下一个 CPU 面板中（顺时针方向）。

按 Shift+TAB，可以切换到前一个 CPU 面板（逆时针方向）。

三、断点 [Breakpoints]

OllyDbg 支持数种不同类型的断点：

一般断点 [Ordinary breakpoint]，将你想中断的命令的第一个字节，用一个特殊命令 INT3（调试器陷阱）来替代。你可以在反汇编窗口中选中要设断点的指令行并按下 **F2** 键就可以设定一个此类型的断点。也可以在快捷菜单中设置。再次按下 F2 键时，断点将被删除。注意，**程序将在设断指令被执行之前中断下来**。INT3 断点的**设置数量是没有限制的**。当你关闭被调试程序或者调试器的时候，OllyDbg 将自动把这些断点保存到硬盘中，**永远不要试图在数据段或者指令的中间设置这种断点**，如果你试图在代码段以外设置断点，OllyDbg 将会警告。你可以在安全选项 [Security options] 中永远关闭这个提示，在某些情况下调试器会插入自带的临时 INT3 断点。

条件断点 [Conditional breakpoint]（快捷键 **Shift+F2**）是一个带有条件表达式的普通 INT3 断点。当调试器遇到这类断点时，它将计算表达式的值，如果结果非零或者表达式无效，将暂停被调试程序，当然，由条件为假的断点引起的开销是非常高的（主要归因于操作系统的反应时间）。在 Windows NT、奔腾 II/450 处理器环境下 OllyDbg 每秒最多处理 2500 个条件为假的断点。**条件断点的一个典型使用情况**就是在 Windows 消息上设置断点（比如 WM_PAINT）。为此，你可以将伪变量 MSG 同适当的参数说明联合使用。如果窗口被激活，参考一下后面的消息断点描述。

条件记录断点 Conditional logging breakpoint（**Shift+F4**）是一种条件断点，每当遇到此类断点或者满足条件时，它将记录已知函数表达式或参数的值。例如，你可以在一些窗口过程函数上设置记录断点并列出对该函数的所有调用。或者只对接收到的 WM_COMMAND 消息标识符设断，或者对创建文件的函数（CreateFile）设断，并且记录以只读方式打开的文件名等，记录断点和条件断点速度相当，并且从记录窗口中浏览上百条消息要比按上百次 F9 轻松的多，你可以为表达式选择一个预先定义好的解释说明。你可以设置通过的次数一

每次符合暂停条件时，计数器就会减一。如果通过计数在减一前，不等于零，OllyDbg 就会继续执行。如果一个循环执行 100 次（十进制），在循环体内设置一个断点，并设置通过次数为 99（十进制）。OllyDbg 将会在最后一次执行循环体时暂停。

另外，条件记录断点允许你传递一个或多个命令给插件 [plugins]。例如，你需要使用命令行插件改变一个寄存器的内容，然后继续执行程序。

消息断点 [Message breakpoint] 和条件记录断点基本相同，除了 OllyDbg 会自动产生一个条件，这个条件允许在窗口过程的入口处设置某些消息（比如 WM_PSINT）断点，你可以在窗口 [Windows] 中设置它。

跟踪断点 [Trace breakpoint] 是在每个选中命令上设置的一种特殊的 INT3 断点。如果你设置了 Hit 跟踪 [hit trace]，断点会在命令执行后移除，并在该地址处做一个标记。如果你使用的是 Run 跟踪 [run trace]，OllyDbg 会添加跟踪数据记录并且断点仍然是保持激活状态。

内存断点 [Memory breakpoint] OllyDbg 每一时刻只允许有一个内存断点。你可以在反汇编窗口、CPU 窗口、数据窗口中选择一部分内存，然后使用快捷菜单设置内存断点。如果有以前的内存断点，将被自动删除。你有两个选择：在内存访问（读，写，执行）时中断，或内存写入时中断。设置此类断点时，OllyDbg 将会改变所选部分的内存块的属性。在与 80x86 兼容的处理器上将会有 4096 字节的内存被分配并保护起来。即使你仅仅选择了一个字节，OllyDbg 也会将整个内存块都保护起来。这将会引起大量的错误警告，请小心使用此类断点。某些系统函数（特别是在 Windows95/98 下）在访问受保护的内存时不但不会产生调试事件反而会造成被调试程序的崩溃。

硬断点 [Hardware breakpoint]（仅在 Windows ME，NT 或 2000 下可用）在 80x86 兼容的处理器上，允许你设置 4 个硬件断点。和内存断点不同，硬件断点并不会降低执行速度，但是最多只能覆盖四个字节。在单步执行或者跟踪代码时，OllyDbg 能够使用硬断点代替 INT3 断点。

内存访问一次性断点 [Single-shot break on memory access]（仅在 Windows NT 或 2000 下可用）。你可以通过内存窗口的快捷菜单（或按 **F2**），对整个内存块设置该类断点。当你想捕捉调用或返回到某个模块时，该类断点就显得特别有用。中断发生以后，断点将被删除。

暂停 Run 跟踪 [Run trace pause]（快捷键：**Ctrl+T**）是在每一步 Run 跟踪 [run trace] 时都要检查的一个条件集。你可以在 EIP 进入某个范围或超出某个范围时暂停，某个条件为真时暂停，或者命令与指定的模式匹配时暂停，或者当命令可疑的时候暂停。注意，这一选择会极大的（高达 20%）降低 Run 跟踪的速度。OllyDbg 也可以在一些调试事件 [debugging events] 上暂停程序执行。比如加载或卸载 DLL，启动或终止线程，或者程序发出调试字符串的时候。

四、数据窗口 [Dump]

数据窗口用于显示内存或文件的内容。你可以从以下预处理格式 [predefined formats] 中选择一种显示方式：字节 [byte]、文本 [text]、整数 [integer]、浮点数 [float]、地址 [address]、反汇编 [disassembly]、PE 头 [PE Header]。

所有的 dump 窗口支持备份 [backup]、搜索和编辑操作。CPU 窗口 [CPU window] 的 Dump 面板允许你对可执行代码的数据和可执行文件 (.exe, 或.dll) 的内存映射做如下操作: 定义标签 [labels]、设置内存断点 [memory breakpoints]、查找参考 [references]。数据菜单 [Dump menu] 只显示与选中部分相关的命令。

如果 备份 [backup] 可用, 则单击第一个列标题栏, 会在地址 [Address]/备份 [Backup] 两种显示模式之间切换。点击其他列标题栏, 会改变 Dump 模式。

像反汇编窗口一样, 数据窗口也保存了大量查看内存地址的历史记录。你可以通过“+”和“-”键来访问过去查看过的数据地址空间。要翻动一字节的数据, 可以按住 **Ctrl+↓** 或 **Ctrl+↑**。

五、可执行模块窗口 [Executable modules window]

可执行模块窗口 (快捷键: **Alt+E**) 列出了当前被调试进程加载的所有可执行模块。它也显示了很多有用的信息, 比如模块大小、入口地址、模块版本、以及可执行文件路径等。一些信息, 如以十进制显示的模块大小、入口地址的符号名、是否为系统模块等, 通常是被隐藏的。如果想看, 可以增加相应栏的宽度。快捷菜单支持以下操作:

刷新 [Actualize] —重新扫描模块并去除对新加载模块的高亮显示。在大多数情况下, OllyDbg 会自动完成该操作。

查看内存 [View memory] —打开内存窗口, 并定位到属于该模块镜像的第一个内存块处。

在 CPU 窗口中查看代码 [View code in CPU] (快捷键: **回车键**) —在反汇编窗口中显示模块的可执行代码。

跟进到入口 [Follow entry] —在反汇编窗口中跟进到模块的入口处。

在 CPU 窗口中查看数据 [Dump data in CPU] —在 CPU 窗口的数据面板中显示模块的数据段。块代码段。

显示名称 [View names] (快捷键: **Ctrl+N**) —显示当前模块定义或使用的全部名称 [names] (包括输出表、引入表、链接库、用户自定义)。

标记为系统 DLL [Mark as system DLL], 标记为非系统 DLL [Mark as non-system DLL] —将选中模块标记为系统或非系统属性。如果设置为系统属性, 则在 **Run 跟踪** [Run trace] 时会直接执行 (不进行跟踪) 这个模块, 从而大大加快跟踪速度。默认情况下, 所有驻留在系统目录 (通常在 Windows 95/98 下为 c:\windows\system, 在 WinNT/2000/XP 下为 c:\winnt\system32) 的模块都认为是系统模块。

立即更新.udd 文件 [Update .udd file now] —向文件“<模块名>.udd”写入模块相关的全部数据, udd 文件保存了在调试期间设置的断点、标签、注释、监视、分析等信息。当模块卸载时 OllyDbg 会自动创建.udd 文件。

查看可执行文件 [View executable file] —显示可执行文件的全部内容。

查看全部资源 [View all resources] —以列表形式显示模块定义的全部资源, 并带有一

个简短信息。OllyDbg 并不把资源当作单独实体来支持。你可以提取 [Dump] 并以二进制的形式进行编辑。

查看资源字符串 [View resource strings] —以列表形式显示资源字符串及其标识符。

查看 Run 跟踪的统计 [View run trace profile] —在此模块中计算统计 [profile]。

分析全部模块 [Analyze all modules] —允许同时分析全部模块。分析将从代码中提取大量的有用信息；代码经过分析后再进行调试，通常会非常快并且可靠。

鼠标双击某一行，将会在反汇编窗口中显示模块的执行代码。

六、内存映射窗口 [Memory map window]

内存映射窗口显示了被调试程序分配的所有内存块。因为没有标准的方法来完成这项任务，所以 OllyDbg 可能会把一个大的内存块分成几个部分。然而，在大多数情况下，并非一定要精确处理。如果想查看由应用程序通过调用 GlobalAlloc ()和 LocalAlloc()等申请的内存块列表，请使用堆列表 [Heap list]。

如果内存块是可执行模块的一个节，OllyDbg 则会报告这个内存块所包含的数据类型：代码、数据、资源等。

Windows95/98 是和 WindowsNT/2000 是有一些区别的。在 Windows95/98 下，OllyDbg 是不能显示被映射文件的名称的。另外，Windows95/98 不允许的访存类型为读和写，然而，在 WindowsNT/2000 下，OllyDbg 却有拥有更多功能，包括执行访问，写复制 [copy—on—write] 以及监视标志位。OllyDbg 忽略写复制 [copy—on—write] 属性。

如果 OllyDbg 发现程序分配了新内存或者重新分配了已经存在的内存块，它将在内存映射窗口中高亮显示相应的记录，去掉高亮度显示，可以选择快捷菜单中的刷新 [Actualize] 项。你可以按 Alt+M 来调用内存窗口。

以下是快捷菜单中可以选择的菜单项：

刷新 [Actualize] —更新已分配内存的列表并去除对新内存块的高亮显示。

在反汇编窗口中查看 [View in Disassembler] —在反汇编窗口中查看：在反汇编窗口中打开内存块，这一选项仅在某些模块的内存块中包含可执行代码或者自解压器时可用。

在 CPU 数据窗口中查看 [Dump in CPU] —在 CPU 的数据窗口中显示内存块的内容。

数据窗口 [Dump] —在单独窗口中显示内存块内容。如果内存块的类型已知，则 OllyDbg 会自动选择显示格式。

查看全部资源 [View all resources] —如果内存块包含资源数据，则列出所有资源及相关数据。OllyDbg 并不把资源当作单独实体来支持。你可以显示其数据并以二进制的形式进行编辑。

查看资源字符串 [View resource strings] —如果内存块包含资源数据，则列出全部资源字符串及其标识符。

搜索 [Search] —允许搜索所有的内存块，从选择处开始，搜索匹配的二进制串。如果找到，则 OllyDbg 将显示该内存块。内存映像窗口和数据窗口共享同一种搜索模式，所以你可以在弹出的数据窗口中立即继续搜索该二进制串出现的下一位置。按 Esc 键可以关闭数据窗口。

搜索下一个 [Search next] (快捷键: Ctrl+L) —继续上次搜索。

设置访问中断 [Set break—on—access] (快捷键: F2, 仅在 WindowsNT/2000 下可用) —保护整个内存块。当中断发生后 OllyDbg 暂停被调试程序并清除断点。这类断点在你想捕捉调用或返回到某个模块的时候特别有用。

清除访问中断 [Remove break—on—access] (快捷键: F2) —从内存块中清除访问中断保护。

设置内存访问断点 [Set memory breakpoint on access] —在整个内存块上设置断点，每当该内存块被访问时程序都将中断。OllyDbg 只支持一个内存访问断点。在 Windows95/98 下，当系统程序访问含有内存断点的内存块时，可能会导致所被调试程序崩溃，因此，不到万不得已，请不要设置这种断点。

设置内存写入断点 [Set memory breakpoint on write] —在整个内存块上设置断点，每当该内存块被写入数据时程序都将中断。在 Windows95/98 下，当系统程序访问含有内存断点的内存块时，可能会导致所被调试程序崩溃，因此，不到万不得已，请不要设置这种断点。

清除内存断点 [Remove memory breakpoint] —清除内存断点。

清除自解压内存断点 [Remove SFX memory breakpoint] —停止搜索自解压程序 [self—extractable (SFX) program] 的真实入口。这个搜索使用了特殊类型的内存断点。

访问设置 [Set access] —设置整个内存块的保护属性，可选择的有：

禁止访问 [No access]

只读 [Read only]

读/写 [Read/write]

执行 [Execute]

执行/读 [Execute/read]

完全访问 [Full access]

复制到剪切板 [Copy to clipboard]

整行 [Whole line] —以多行文本（包括解释）的方式把所选记录复制到剪切板，如果复制时想排除某些列，可将该列的宽度置为最小（该栏剩余的边框将变灰）。

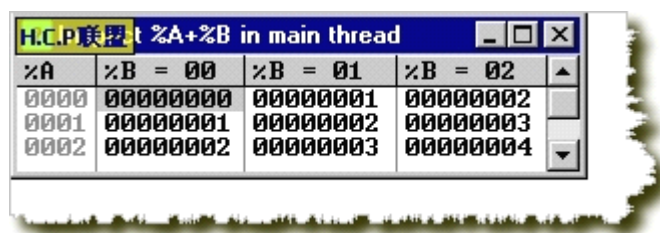
整个表格 [Whole table] —以多行文本的方式将整个内存映像信息复制到剪切板，该文本的第一行为窗口标题（"内存映射 [Memory map]"），第二行为列标题栏，后面几行的内容为内存数据记录。复制将保持列的宽度。如果复制时想排除某些列，可将该列的宽度置为最小（该栏剩余的边框将变灰）。

七、监视与监察器 [Watches and inspectors]

监视 [Watch] 窗口包含若干个表达式 [expressions]。它在第二列里显示这些表达式的值。OllyDbg 会把这些表达式保存到主模块的.UDD 文件中，因此它们在下一一次调试时同样有效。

监察器 [inspector] 是显示若干变量、1/2 维数组或是选定项目结构数组 [selected items of array of structures] 的独立窗口。它的表达式与监视窗口中的基本相同，只是多包含了两个参数：%A 和%B。你可以指定这两个参数的界限，OllyDbg 将会用所有可能的组合代替表达式中的%A 和%B。从 0 开始一直到界限（不包含界限），并在表格中显示结果。参数 %B（列数）的界限不能超过 16。

例如，如果你指定了表达式%A+%B，并且限定%A 和%B 的上限为 3，你将获得如下的表格：



%A	%B = 00	%B = 01	%B = 02
0000	00000000	00000001	00000002
0001	00000001	00000002	00000003
0002	00000002	00000003	00000004

八、线程 [Threads]

OllyDbg 以简单而有效的线程管理为特色。如果你单步调试、跟踪、执行到返回或者执行到所选，则线程管理器将停止除当前线程以外的所有线程。即使当前线程被挂起，它也会将其恢复。在这种情况下，如果你手动挂起或者恢复线程，动作将被延期。如果你运行被调试的应用程序，OllyDbg 将恢复最初的线程状态。（从调试器的角度来看，Hit 跟踪 [hit trace] 和自由运行是等效的）。

依据这种方案，线程窗口可能会有如下五种线程状态：

激活 [Active]	线程运行中，或被调试信息暂停 t
挂起 [Suspended]	线程被挂起
跟踪 [Traced]	线程被挂起，但 OllyDbg 正在单步跟踪此线程
暂停 [Paused]	线程是活动的，但 OllyDbg 临时将其挂起，并在跟踪其它的线程
结束 [Finished]	线程结束

线程窗口同时也显示了最后的线程错误（GetLastError 函数的返回值）并计算该线程以用户模式和系统模式（仅 NT/2000/XP）运行的时间。线程窗口还会高亮主线程的标识符。

以下在快捷菜单中可用：

刷新 [Actualize] — 标记所有线程为旧的。

挂起 [Suspend] — 挂起线程。

恢复 [Resume] — 恢复先前挂起的线程。

设置优先级 [Set priority] — 调整进程中线程的优先级。以下选项可用：

空闲 [Idle] 一进程中线程的最低优先级
 最低 [Lowest]
 低 [Low]
 标准 [Normal]
 高 [High]
 最高 [Highest]
 时间临界 [Time critical] 一最高优先级

在 CPU 窗口打开 [Open in CPU] (双击) 一在 CPU 窗口中显示所选线程的当前状态。

复制到剪切板 [Copy to clipboard]:

整行 [Whole line] 一全部行——以多行文本的形式并带注释将所选记录复制到剪切板。如果在复制时想排除某个栏目，可以将该栏的宽度置为最小（栏目的残留部分将变灰）。

整个表格 [Whole table] 一整个表格——以多行文本的形式将整个内存映象复制到剪切板，该文本的第一行包含窗口标题（“内存映射 [Memory map] ”），第二行是栏目标题，所有后继行是内存数据记录。复制将保持栏目的宽度。如果在复制时想排除某些栏目，可以将该栏的宽度置为最小（栏目的残留部分将变灰）。

九、调用栈 [Call stack]

调用栈窗口（快捷键：**Alt+K**）根据选定线程的栈，尝试反向跟踪函数调用顺序并将其显示出来，同时包含被调用函数的已知的或隐含的参数。如果调用函数创建了标准的堆栈框架（PUSH EBP; MOV EBP,ESP），则这个任务非常容易完成。现代的优化编译器并不会为栈框架而操心，所以 OllyDbg 另辟蹊径，采用了一个变通的办法。例如，跟踪代码到下一个返回处，并计算其中全部的入栈、出栈，及 ESP 的修改。如果不成功，则尝试另外一种办法，这个办法风险更大，速度也更慢：移动栈，搜索所有可能的返回地址，并检查这个地址是否被先前的已分析的命令调用。如果还不行，则会采用启发式搜索。栈移动 [Stack Walk] 可能会非常慢。OllyDbg 仅在调用栈窗口打开时才会使用。

调用栈窗口包含 5 个栏目：地址 [Address]、栈 [Stack]、过程 [Procedure]，调用来自 [Called from]，框架 [Frame]。

地址 [Adress] 栏包含栈地址，栈 [Stack] 栏显示了相应的返回地址或参数值。

函数 [Procedure]（或 函数/参数 [Procedure / arguments]）显示了被调用函数的地址，在某些情况下，OllyDbg 并不能保证该地址是正确的并会添加如下标记之一：

? 找到的入口点不可靠
 可能 [Maybe] OllyDbg 无法找到精确的入口点，报告的地址是用启发式算法猜测的。
 包含 [Includes] OllyDbg 无法找到入口点，仅知道该函数包含显示的地址

通过按例标题栏上的按钮或从菜单中选择“隐藏/显示参数 [Hide/Show arguments] ”，可以在显示或隐藏函数的参数之间切换。

调用来自 [Called from] 用于显示调用该函数的命令地址。

最后一栏是框架 [Frame] 这一栏默认是隐藏的，如果框架指针的值（寄存器 EBP）已

知的话，则该栏用于显示这个值。

当调用函数经过分析 [analyzed] 后，栈移动会更可靠并且迅速。

十、调用树 [Call tree]

调用树（快捷键：在反汇编窗口中 **Ctrl+K**）利用分析 [Analysis] 的结果来找出指定函数过程直接或间接调用的函数列表，同时列出指定函数过程被调用的地址。为了避免由此可能造成的副作用。调用树会判断选定函数是否明确地是递归的。“明确地”意味着它不会跟踪目标未知的调用，比如 CALL EAX。如果函数过程中有未知调用，调用树将会添加标记“未知目标”。

某些函数调用将会添加如下注释之一：

叶子 [Leaf] 不调用其他函数

纯函数 [Pure] 不调用函数，不会产生副作用

单返回 [RETN] 只有一个 RETN 命令

系统 [Sys] 系统动态链接库中的函数。系统动态链接库定义为保存在系统目录下的动态链接库。

如果想在调用树上移动，可以双击“被调用 [Called from]”或“调用/直接调用 [Calls/Calls directly]”两栏中的地址。调用树窗口保存了移动记录（快捷键“-”和“+”）。

如果被调试的程序包含几个模块，推荐你分析所有模块。Call tree 不会试图处理系统函数。

十一、选项 [Options]

外观选项 [Appearance options]

常规 [General]

默认 [Defaults]

对话框 [Dialogs]

目录 [Directories]

字体 [Fonts]

颜色 [Colours]

代码高亮 [Code highlighting]

调试选项 [Debugging options] (Alt+O)

安全 [Security]

调试 [Debug]

事件 [Events]

异常 [Exceptions]

跟踪 [Trace]

自解压 [SFX]

- 字符串 [Strings]
- 地址 [Addresses]
- 命令 [Commands]
- 反汇编 [Disasm]
- CPU
 - 寄存器 [Registers]
 - 栈 [Stack]
 - 分析 1 [Analysis 1]
 - 分析 2 [Analysis 2]
 - 分析 3 [Analysis 3]
 - 即时调试 [Just-in-time debugging]
 - 添加到资源管理器 [Add to Explorer]

十二、搜索 [Search]

OllyDbg 允许你使用以下的搜索方式：

- 符号名（标签） [Symbolic name (label)]
- 二进制串 [binary string]
- 常量 [constant]
- 命令 [command]
- 命令序列 [sequence of commands]
- 模块间调用 [intermodular calls]
- 修改过的命令或数据 [modified command or data]
- 自定义标签 [user-defined label]
- 自定义注释 [user-defined comment]
- 文本字符串 [text string]
- Run 跟踪的记录 [record in run trace]
- 参考命令 [referencing commands]

十三、自解压文件 [Self-extracting (SFX) files]

自解压文件由提取程序和压缩的原程序两部分组成。当遇到自解压文件 (SFX) 文件时，我们通常希望跳过解压部分，而直接跳到原始程序的入口（真正的入口）。

OllyDbg 包含了几个便于完成这一任务的功能。

通常提取程序的加载地址都在执行代码之外。在这种情况下，OllyDbg 将这类文件均视为自解压文件(SFX)。

当自解压选项 [SFX options] 要求跟踪真正入口时，OllyDbg 在整个代码节 [Code section] 设置内存断点，最初这里是空的，或者只包含压缩数据。当程序试图执行某个在这个保护区域的命令，而这些命令不是 RET 和 JMP 时，OllyDbg 会报告真正的入口。这就是提取工作的原理。

上面的方法非常慢。有另外一种比较快的方法。每次读取数据发生异常时，OllyDbg 使这个 4K 内存区域变为可读，而使原先可读的区域变为无效。而每次发生写数据异常时，OllyDbg 使这个区域变为可写，而使原先可写的区域变为无效。当程序执行在保留的保护区域中的指令时，OllyDbg 报告真正的入口。但是，当真正的入口点在可读或可写区域内部时，报告的地址就可能有误。

你可以纠正入口位置，选择新的入口，从反汇编窗口的快捷菜单中选择“断点 [Breakpoint] —>设置真正的自解压入口 [Set real SFX entry here]”。如果相应的 SFX 选项是开启的，OllyDbg 下次可以迅速而可靠的跳过自提取程序。

注意：OllyDbg 在跟踪采取了保护或者反调试技术的解压程序时通常会失败。

十四、单步执行与自动执行 [Step-by-step execution and animation]

你可以通过按 F7（单步步入）或 F8（单步步过），对程序进行单步调试。这两个单步执行操作的主要区别在于：如果当前的命令是一个子函数，按 F7，将会进入子函数，并停在子函数的第一条命令上；而按 F8，将会一次运行完这个子函数。如果你单步步过的子函数中含有断点或其他调试事件，执行将会被暂停，并且 OllyDbg 会在子函数的后一条命令上，自动下一个断点，而这个断点你迟早会碰到。

如果被调试程序停在异常上，你可以跳过它，并转到被调试程序建立的句柄处。只需简单的 **Shift** 键和任何一个单步命令。

如果需要连续按 F7、F8 键上百次，你可以使用自动执行（**Ctrl+F7** 或者 **Ctrl+F8**）功能。在这种情况下，OllyDbg 将自动重复 F7 或者 F8 操作，并且实时更新所有的窗口。这个过程会在下面情况停止：

- 按 Esc 键或发出任何单步命令
- OllyDbg 遇到断点
- 被调试程序发生异常

使用“+”和“-”按键，可以回溯以前的执行历史 [execution history]。

注意：当执行停止时 OllyDbg 将会刷新大部分窗口。如果动态执行过程非常慢，可以尝试关掉或最小化没有用的窗口。

另外，更快捷的找到以前执行指令的办法是 Run 跟踪 [run trace]。它将创建一个执行协议并告知你指定指令的执行时间和次数

十五、Hit 跟踪 [Hit trace]

Hit 跟踪能够让你辨别哪一部分代码执行了，哪一部分没有。OllyDbg 的实现方法相当简单。它将选中区域的每一条命令处均设置一个 INT3 断点。当中断发生的时候，OllyDbg 便把它去除掉，并把该命令标志为命中 [hit]。因为每个跟踪断点只执行一次，所以这种方法速度非常快。

在使用 **Hit** 跟踪的时候，一定要注意不能在数据中设置断点，否则应用程序极有可能崩溃。因此，你必须打开相关的菜单选项，以进行代码分析 [analyze]。我推荐你选择严格或启发式函数识别 [strict or heuristical procedure recognition]。如果选择模糊 [Fuzzy] 的话，可能会产生很多难以容忍的错误，而且经常把本不是函数的代码段识别成函数。

只要你在模块中设置了跟踪断点，哪怕只设了一个，OllyDbg 都会分配两倍于代码段大小的缓冲区。

注意：当你退出 **Hit** 跟踪的时候，**Run** 跟踪也会同时退出。

十六、Run 跟踪 [Run trace]

Run 跟踪是一种反方向跟踪程序执行的方式，可以了解以前发生的事件。你还可以使用 Run 跟踪来了解运行的简单统计 [profile]。基本上，OllyDbg 是一步一步地执行被调试程序的，就像动画 [animation] 演示一样，但不会实时刷新窗口，最重要的是它能将地址、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。如果被调试的代码是自修改的，你就能够保存原始的命令。可以通过按 **Ctrl+F11** (**Run** 跟踪步入，进入子函数) 或者 **Ctrl+F12** (**Run** 跟踪步过，一次执行完子函数) 开始 Run 跟踪，并用 **F12** 或者 **Esc** 键停止跟踪。

你可以指定在 Run 跟踪时执行每一步的条件集 (快捷键: **Ctrl+T**)。如果条件符合，Run 跟踪将暂停。条件包括:

- (1) 当 EIP 在某个地址范围内时暂停 [Pause when EIP is in the address range];
- (2) 当 EIP 在某个地址范围之外时暂停 [Pause when EIP is outside the address range];
- (3) 当某个条件为真时暂停 [Pause when some condition is true];
- (4) 当下一条指令可疑时暂停 [Pause when next command is suspicious]，比如：可能为非法指令 (根据在分析 3 [Analysis 3] 中设定的规则而定)，访问不存在的内存，设置了单步陷阱标志 [single-step trap flag] 或者越 ESP 界访问栈。注意：这个选项会明显地 (大约 20%) 减慢 Run 跟踪的速度；
- (5) 当命令执行达到指定的次数 (更确切的说，是添加到 Run 跟踪的缓冲区里面的命令数量) 时暂停 [Pause after specified number of commands is traced]。注意计数器不能自动归零。也就是说，如果你设置指令次数为 10，则在第 10 次执行到该命令时暂停，并不是该命令每执行 10 次就暂停一次。
- (6) 当下一条命令符合指定的样式之一时暂停 [Pause when next command matches one of the specified patterns]。你可以使用模糊命令和操作数 [imprecise commands and operands] 及匹配 32 位寄存器 RA 和 RB，像 R32 一样，这两个寄存器可以替代任何通用 32 位寄存器，但是在同一条命令中其值是不能变的。而 RA 和 RB 在同一条命令中，则一定是不同的。例如，在程序中含有 XOR EAX,EAX; XOR ESI,EDX 两条命令，两条命令均符合样式 XOR R32,R32; 第一条命令符合样式 XOR RA,RA; 而第二条命令 XOR ESI,EDX 符合样式 XOR RA,RB。

毫无疑问，Run 跟踪需要足够的内存，每条命令平均需要占用 16 到 35 字节，同时速度也非常慢。在 500—MHZ 处理器、Windows NT 环境下，它每秒能跟踪 5000 条指令。Windows95 更慢：每秒钟仅 2200 条指令。但是在许多情况下，例如当一个程序跳转到不存在的地址的时候，这是找到原因的唯一方法。你可以在 Run 跟踪时将准线性命令序列 (即序列尾部只

有唯一出口) 跳过。当 OllyDbg 遇到这些需跳过的命令序列时, 会设置一个临时断点, 然后跟进到序列中, 并一次运行完。当然了, 如果排除命令中返回或跳转的地址在跟踪范围之外, 将可能导致跟踪发生错误; 因此 OllyDbg 会检查你想跳过的代码块, 如果存在上述情况, 会向你询问。

在大多数情况下, 你对跟踪系统 API 代码不感兴趣。跟踪选项总是跟过系统 DLL [Always trace over system DLLs] 允许你在跟踪/自动模式下跟过 API 函数。如果模块在系统目录下, OllyDbg 就假设该模块是系统的。你可以在模块 [Modules] 窗口中标记任意 DLL 是系统的或者非系统的。

为了使执行速度更快, 你可以通过设置 Run 跟踪断点, 先将 Run 跟踪限制在选定的命令或代码块上, 然后再运行程序。我把这种做法称作“**强迫 Run 跟踪**”。一般来说, **删除 Run 跟踪断点不会移除 Hit 跟踪断点。但如果你删除了 hit 跟踪断点, 同时你也移除了 Run 跟踪断点。**

跟踪命令会保存到跟踪缓冲区中, 这个缓冲区在跟踪开始时自动创建。你可以在选项中指定它的大小(最高 64MB)。这个缓冲区是循环队列, 当满了的时候, 会丢弃老的记录。你可以通过从 OllyDbg 主菜单中选择“**调试 [Debug] —> 打开或者清除 Run 跟踪 [Open or clear run trace]**”, 来**打开或者清除 Run 跟踪缓冲区**。在 Run 跟踪缓冲区打开后, OllyDbg 会记录在执行过程中的所有暂停, 甚至那些不是由 Run 跟踪引起的暂停。例如, 你可以通过按 F7 或者 F8 单步执行程序, 然后通过使用+键和一键来反方向跟踪程序的执行。注意: 如果 Run 跟踪缓冲区已经关闭, 则用这些键浏览的是历史 [history] 记录。在你查看 Run 跟踪记录时, 寄存器和信息面板会变灰, 来强调它们所显示的寄存器并不是实际的寄存器。跟踪缓冲区并不保存栈顶或由寄存器所指向的内容。寄存器、信息和栈在 Run 跟踪的时候使用实际的内存状态来解释寄存器的变化。

OllyDbg 能够记下每个指令在 Run 跟踪缓冲区里面出现的次数。在反汇编窗口快捷菜单中, 选择是“**查看 [View] —> 统计作为注释 [Profile as comments]**”。这个命令使用统计取代了注释栏。或者, 如果列标题栏可见, 则可以单击它几次直到它显示统计信息。注意显示出来的数字是动态的, 而且不计算已经从跟踪缓冲区中丢弃的指令。你还可以在单独的统计窗口 [Profile window] 中, 按触发次数排序, 来查看整个模块的统计数据。

在反汇编窗口的快捷菜单中选择“**Run 跟踪 [Run trace] —> 添加到所有函数入口处 [Add entries of all procedures]**”, 这样能够检查每个可识别的函数被调用的次数。另一个命令“**Run 跟踪 [Run trace] —> 添加到函数中所有的分支 [Add branches in procedure]**”会强行跟踪此函数中所有识别的跳转目的地址的内容。在这种情况下, 统计功能能够找到最频繁执行的分支, 你可以优化这部分的代码, 以提高速度。

在反汇编窗口中的某条命令上使用快捷菜单中选择“**搜索 [Search for] —> Run 跟踪的最新记录 [Last record in run trace]**”用于查找该命令是否被执行过, 如果执行过, 最后一次执行在哪里。

Run 跟踪窗口显示跟踪缓冲区的内容。对每个指令来说包括被指令改变的整数寄存器的内容(更准确的说是给定的记录变成下一条记录的变化)。如果你双击某条指令, 窗口会选择在跟踪缓冲区里全部含有该命令的记录, 而且你可以通过按+和-键来快速的浏览; 如果你在调试选项 [Debugging options] 中设置了“**跟踪 [Trace] —> 同步 CPU 和 Run 跟踪 [Synchronize CPU and Run trace]**”, 双击记录则会跟进到对应的反汇编窗口中位置。

注意：当你退出 Hit 跟踪时，你同时也强行退出了 Run 跟踪。

十七、快捷键

1、通用快捷键 [Global shortcuts]

无论当前的 OllyDbg 窗口是什么，这些快捷键均有效：

Ctrl+F2—重启程序，即重新启动被调试程序。如果当前没有调试的程序，OllyDbg 会运行历史列表 [history list] 中的第一个程序。程序重启后，将会删除所有内存断点和硬件断点。译者注：从实际使用效果看，硬件断点在程序重启后并没有移除。

Alt+F5—让 OllyDbg 总在最前面。如果被调试程序在某个断点处发生中断，而这时调试程序弹出一个总在最前面的窗口（一般为模式消息或模式对话框 [modal message or dialog]），它可能会遮住 OllyDbg 的一部分，但是我们又不能移动最小化这个窗口。激活 OllyDbg（比如按任务栏上的标签）并按 Alt+ F5，OllyDbg 将设置成总在最前面，会反过来遮住刚才那个窗口。如果你再按一下 Alt+F5，OllyDbg 会恢复到正常状态。OllyDbg 是否处于总在最前面状态，将会保存，在下次调试时依然有效。当前是否处于总在最前面状态，会显示在状态栏中。

F4—运行到选定位置。作用就是直接运行到光标所在位置处暂停。

F7—单步步入到下一条命令，如果当前命令是一个函数 [Call]，则会停在这个函数体的第一条命令上。如果当前命令是含有 REP 前缀，则只执行一次重复操作。

Shift+F7—与 F7 相同，但是如果被调试程序发生异常而中止，调试器会首先尝试步入被调试程序指定的异常处理（请参考忽略 Kernel32 中的内存非法访问）。

Ctrl+F7—自动步入，在所有的函数调用中一条一条地执行命令（就像你按住 F7 键不放一样，只是更快一些）。当你执行其他一些单步命令，或者程序到达断点，或者发生异常时，自动步入过程都会停止。每次单步步入，OllyDbg 都会更新所有的窗口。所以为了提高自动步入的速度，请你关闭不必要窗口，对于保留的窗口最好尽量的小。按 Esc 键，可以停止自动步入。

F8—单步步过到下一条命令。如果当前命令是一个函数，则一次执行完这个函数（除非这个函数内部包含断点，或发生了异常）。如果当前命令是含有 REP 前缀，则会执行完重复操作，并停在下一条命令上。

Shift+F8—与 F8 相同，但是如果被调试程序发生异常而中止，调试器会首先尝试步过被调试程序指定的异常处理（请参考忽略 Kernel32 中的内存非法访问）。

Ctrl+F8—自动步过，一条一条的执行命令，但并不进入函数调用内部（就像你按住 F8 键不放一样，只是更快一些）。当你执行其他一些单步命令，或者程序到达断点，或者发生异常时，自动步过过程都会停止。每次单步步过，OllyDbg 都会更新所有的窗口。所以为了提高自动步过的速度，请你关闭不必要窗口，对于保留的窗口最好尽量的小。按 Esc 键，可以停止自动步过。

F9—让程序继续执行。

Shift+F9—与 F9 相同，但是如果被调试程序发生异常而中止，调试器会首先尝试执行被调试程序指定的异常处理（请参考忽略 Kernel32 中的内存非法访问）。

Ctrl+F9—执行直到返回，执行程序直到遇到返回，在此期间不进入子函数也不更新 CPU 数据。因为程序是一条一条命令执行的，所以速度可能会慢一些。按 Esc 键，可以停止跟踪。

Alt+F9—执行直到返回到用户代码段，执行程序直到指令所属于的模块不在系统目录中，在此期间不进入子函数也不更新 CPU 数据。因为程序是一条一条执行的，所以速度可能会慢一些。按 Esc 键，可以停止跟踪。

Ctrl+F11—Run 跟踪步入，一条一条执行命令，进入每个子函数调用，并把寄存器的信息加入到 Run 跟踪的存储数据中。Run 跟踪不会同步更新 CPU 窗口。

F12—停止程序执行，同时暂停被调试程序的所有线程。请不要手动恢复线程运行，最好使用继续执行快捷键或菜单选项（像 F9）。

Ctrl+F12—Run 跟踪步过，一条一条执行命令，但是不进入子函数调用，并把寄存器的信息加入到 Run 跟踪的存储数据中。Run 跟踪不会同步更新 CPU 窗口。

Esc—如果当前处于自动运行或跟踪状态，则停止自动运行或跟踪；如果 CPU 显示的是跟踪数据，则显示真实数据。

Alt+B—显示断点窗口。在这个窗口中，你可以编辑、删除、或跟进到断点处。

Alt+C—显示 CPU 窗口。

Alt+E—显示模块列表 [list of modules]。

Alt+K—显示调用栈 [Call stack] 窗口。

Alt+L—显示日志窗口。

Alt+M—显示内存窗口。

Alt+O—显示选项对话框 [Options dialog]

Ctrl+P—显示补丁窗口。

Ctrl+T—打开“暂停 Run 跟踪”对话框

Alt+X—关闭 OllyDbg。

2、大多数窗口都支持以下的键盘命令

Alt+F3—关闭当前窗口。

Ctrl+F4—关闭当前窗口。

F5—最大化当前窗口或将当前窗口大小改为正常化。

F6—切换到下一个窗口。

Shift+F6—切换到前一个窗口。

F10—打开与当前窗口或面板相关的快捷菜单。

左方向键—显示窗口左方一个字节宽度的内容。

Ctrl+左方向键—显示窗口左方一栏的内容。

右方向键—显示窗口右方一个字节宽度的内容

Ctrl+右方向键—显示窗口右方一栏的内容

3、反汇编窗口中的快捷键 [Disassembler shortcuts]

当 CPU 窗口中的反汇编面板 [Disassembler pane] 处于激活状态时，你可以使用以下快捷键：

回车键—将选中的命令添加到命令历史 [command history] 中，如果当前命令是一个跳转、函数或者是转换表的一个部分，则进入到目的地址。

退格键—移除选中部分的自动分析信息。如果分析器将代码误识别为数据，这个快捷键就非常有用。请参考解码提示 [decoding hints]。

Alt+退格键—撤消所选部分的修改，以备份数据的相应内容替换所选部分。仅当备份数据存在且与所选部分不同时可用。

Ctrl+F1—如果 API 帮助文件已经选择，将打开与首个选择行内的符号名相关联的帮助主题。

F2—在首个选择的命令上开关 **INT3** 断点 [Breakpoint]，也可以双击该行第二列。

Shift+F2—在首个选择命令设置条件断点，参见忽略 Kernel32 中内存访问异常 [Ignore memory access violations in Kernel32]。

F4—执行到所选定行，在首个选择的命令上设置一次性断点，然后继续执行调试程序，直到 OllyDbg 捕获到异常或者停止在该断点上。在程序执行到该命令之前，该一次性断点一直有效。如有必要，可在断点窗口 [Breakpoints window] 中删除它。

Shift+F4—设置记录断点（一种条件断点，当条件满足时一些表达式的值会记录下来），详情参见断点 [Breakpoint]。

Ctrl+F5—打开与首个选择的命令相对应的源文件。

Alt+F7—转到上一个找到的参考。

Alt+F8—转到下一个找到参考。

Ctrl+A—分析当前模块的代码段。

Ctrl+B—开始二进制搜索。

Ctrl+C—复制所选内容到剪贴板。复制时会简单地按列宽截断不可见内容，如果希望排除不需要的列，可把这些列的宽度调整到最小。

Ctrl+E—以二进制（十六进制）格式编辑所选内容。

Ctrl+F—开始命令搜索。

Ctrl+G—转到某地址。该命令将弹出输入地址或表达式的窗口。该命令不会修改 **EIP**。

Ctrl+J—列出所有的涉及到该位置的调用和跳转，在你用这个功能之前，你必须使用分析代码功能。

Ctrl+K—查看与当前函数相关的调用树 [Call tree]。在你用这个功能之前，你必须使用分析代码功能。

Ctrl+L—搜索下一个，重复上一次的搜索内容。

Ctrl+N—打开当前模块的名称（标签）列表。

Ctrl+O—扫描 **object** 文件。扫描 Object 文件。该命令会显示扫描 Object 文件对话框，你可以在该对话框中选择 Object 文件或者 lib 文件，并扫描这个文件，试图找到在实际代码段中用到的目标模块。

Ctrl+R—搜索所选命令的参考。该命令扫描激活模块的全部可执行代码，以找到涉及到首个选中的命令的全部相关参考（包括：常量、跳转及调用），你可以在参考中使用快捷键 **Alt+F7** 和 **Alt+F8** 来浏览这些参考。为便于你使用，被参考的命令也包含在该列表中。

Ctrl+S—命令搜索。该命令显示命令查找 [Find command] 对话框供你输入汇编命令，并从当前命令开始搜索。

星号 [Asterisk] (*)—转到原始位置（激活线程的 **EIP** 处）。

Ctrl+星号(*)—指定新的起始位置，设置当前所选线程的 **EIP** 为首个选择字节的地址。你可以在选择 **EIP** 并撤消该操作。

加号 [Plus](+)—如果 run 跟踪 [run trace] 没有激活，则根据命令历史 [command history] 跳到下一条运行过命令的地方；否则跳到 Run 跟踪的下一个记录。

Ctrl+加号—跳到前一个函数的开始处。（注意只是跳到，并不执行）

减号 [Minus] (—)—如果 run 跟踪 [run trace] 没有激活，则根据命令历史 [command history] 跳到前一条运行过命令的地方；否则跳到 Run 跟踪的前一个记录。

Ctrl+减号—跳到下一个函数的开始处。（注意只是跳到，并不执行）

空格 [Space] —修改命令。你可在显示对话框中以汇编语言修改实际指令或输入新指令，这些指令将替换实际代码，你也可以在想要修改的指令处双击鼠标。

冒号 [Colon] (:)—添加标签。显示添加标签窗口 [Add label] 或修改标签窗口 [Change

label], 你可在此输入与首个选择的命令中的第一个字节相关联的标签 (符号名)。注意, 在多种编程语言中, 冒号可以是标签的一部分。

分号 [Semicolon] (;)—添加注释 [comment]。显示添加注释窗口 [Add label] 或修改注释窗口 [Change label], 你可在此输入与首条所选命令的第一个字节相关联的注释 (注释串会显示在最后一列中)。注意, 多种汇编语言使用分号作为注释开始。你也可以在注释列双击需要注释的命令。

十八、插件 [Plugins]

插件是一个 DLL, 存放在 OllyDbg 的目录中, 用于增加 OllyDbg 的功能。你可以从 OllyDbg 的主页上 (<http://home.t-online.de/home/Ollydbg>) 免费下载插件开发工具包 plug110.zip。

插件可以设置断点, 增加标签和注释, 修改寄存器和内存。插件可以添加到主菜单和很多的窗口 (比如反汇编窗口、内存窗口) 的快捷菜单中, 也可以拦截快捷键。插件还可以创建 MDI (多文档界面) 窗口。插件还可以根据模块信息和 OllyDbg.ini 文件, 将自己数据写到.udd 文件中; 并能读取描述被调试程序的各种数据结构。插件 API 包含了多达 170 个函数。

许多第三方插件都可以从 Internet 网上获得, 比如由网友 TBD 创建并维护的 OllyDbg 的论坛 (<http://ollydbg.win32asmcommunity.net>)。

安装插件的方法: 将 DLL 复制到插件目录 [plugin directory] 中, 然后重新启动 Ollydbg。默认情况下, 这个插件目录为 ollydbg.exe 文件所在的目录。

现在的版本中已经包含了两个“原始”插件: 书签 [Bookmark] 和命令行 [Command line]。他们的源代码都保存在 plug110.zip 文件中。这些插件都是免费的, 你可以任意修改或使用它们。

十九、技巧提示 [Tips and tricks]

(1) OllyDbg 可以作为二进制编辑器使用。选择视图 [View] → 文件 [File] 并选定需要查看的文件。文件不能大于剩余内存数量。

(2) 假使你修改了内存中的执行文件, 这时你想恢复修改的部分, 但是你忘记哪里被修改了, 你可以把原始文件当作备份进行加载, 这样你就可以找到修改的部分了。

(3) 分析前, 先扫描 OBJ 文件。这时 OllyDbg 会对已知 C 函数的参数进行解码。

(4) 一些表格中包含了隐藏数据。可以通过增加列宽来显示出来。

(5) 所有数据窗口 (包括反汇编窗口), 可以通过双击显示相对的地址。

(6) 你可以通过 Ctrl+↑ 或 Ctrl+↓ 对数据窗口翻动一个字节。

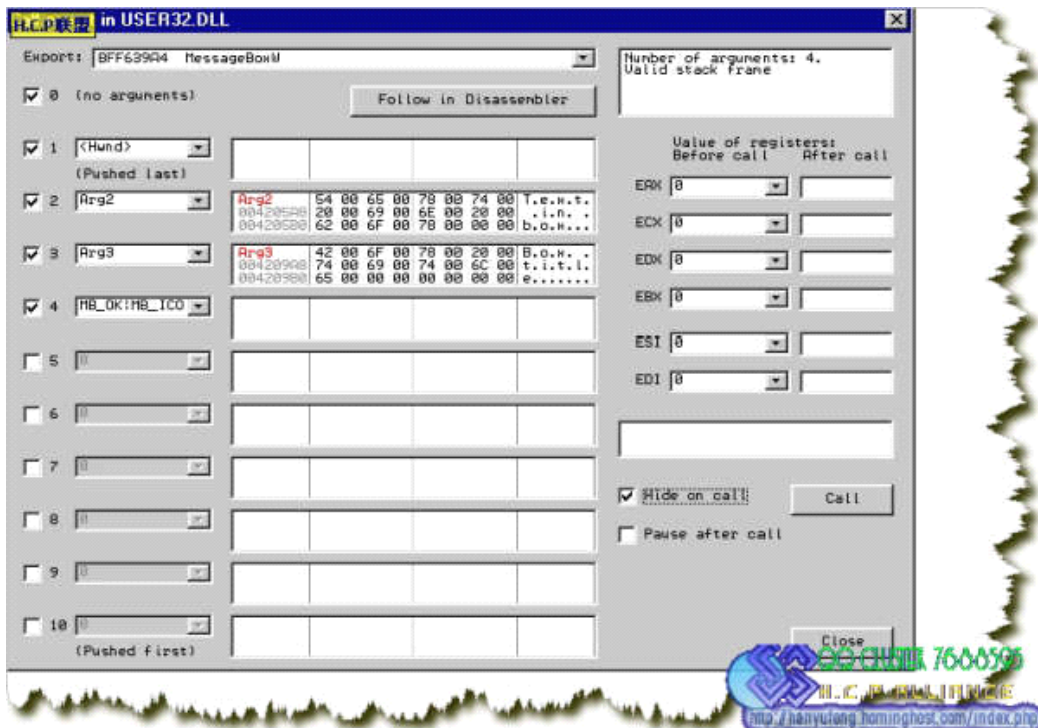
第四章 其他功能

一、调试独立的 DLL [Debugging of stand-alone DLLs]

打开 DLL，也可以直接将其从资源管理器拖放到 OllyDbg 上。OllyDbg 会询问你并将该文件的全路径作为参数传递给 loaddll.exe。然后链接库被加载并停在代码的入口 (<DllEntryPoint>)。你可以设置断点，运行或跟踪启动代码，等等。在初始化完成后，应该程序会再次暂停。这次停在标签名为 Firstbp 的位置，其在立即进入主消息循环之前。

现在，你可以调用 DLL 函数。从主菜单选择“调试 [Debug] →调用 DLL 输出 [Call DLL export]”。这时会弹出一个对话框。由于这个对话框是无模式对话框，因此你仍然能够使用 OllyDbg 的全部功能，比如查看代码、数据，查看断点，修改内存等等。

选择你想调用的函数。例如我们将开始使用 USER32.DLL 里的 MessageBox 函数。注意 loaddll.exe 已经使用了这个链接库，因此会假定这个 DLL 已经初始化而不再调用入口。MessageBox 这个函数名是通用函数名，事实上，这个函数有处理 ASCII 的 MessageBoxA 和处理 Unicode 的 MessageBoxW 两种。我们继续往下看：



在我们选择这个函数后，右边的消息框中会出现 Number of arguments: 4 (有四个参数) 的字样。OllyDbg 会根据函数尾部的 RET 10 语句来正确识别参数的数量。RET nnn 是使用 PASCAL 调用约定的函数的典型特征。(参数被放入栈中，第一个参数会被最后一个压入栈中，函数调用完毕后，参数会被遗弃)。大多数的 Windows API 函数都是 PASCAL 形式的。

下一步，我们要设定栈中参数的个数。在这个例子中，不必做进行这个操作，因为 OllyDbg 已经知道了 MessageBoxW 函数的参数数量。但是，如果你愿意的话，也可以单击左边的复选框，改变成你认为合适的参数数量。

现在填写参数列表。这个对话框中支持至多 10 个参数。参数可以是任何有效的表达式，而不必使用寄存器。如果操作数指向了内存，则参数右边的缓冲区窗口会显示内存中的数据。LoadDll.exe 有 10 个大小为 1K 的缓冲区，这些缓冲区被标记为 Arg1 .. Arg10，你可以方便自由的使用它们。另外，对话框还支持两个伪变量：由 loaddll.exe 创建的父窗口句柄<Hwnd>，以及 loaddll 的实例句柄<Hinst>。为了方便你的使用，在你第一次使用调用输出函数时，OllyDbg 就已经将这两个伪变量加到了历史列表中去了。

MessageBoxW e 函数需要 4 个参数：

- 父窗口句柄。 这里我们选择<Hwnd> ； handle of owner window. Here, we simply select <Hwnd>;
- 在消息框中 UNICODE 文本的地址。选择 Arg2 并按回车。缓冲区窗口会以 16 进制的格式显现内存中的缓冲区。这个缓冲区初始化全是 0。点击第一个字节，并按快捷键 Ctrl+E（另外，也可以从菜单中选择“二进制 [Binary] —>编辑 [Edit] ”）。这时会出现一个对话框，在对话框中键入“Text in box”或者其他希望显示的字符串；
- 消息框标题的 UNICODE 文本的地址。选择 Arg3 并在 Unicode 格式的内存中写上“Box title”；
- 消息框的风格。使用常量 MB_XXX 进行组合。OllyDbg 可以识别这些常量。在这里我们键入：MB_OK—>MB_ICONEXCLAMATION。

这里不需要寄存器参数。

现在我们准备调用输出函数。选项“在调用时隐藏 [Hide on call]”意思是说，当函数运行时对话框将会从屏幕消失。当我们执行一个会运行很长时间的函数，或者设置了断点的时候，这个选项非常的有用。你也可以手动关闭对话框。当函数执行完毕后，OllyDbg 会重新自动打开。“调用输出函数”对话框。选项“在调用后暂停 [Pause after call]”意思是说，在执行完函数后，loaddll 将会被暂停。

按“调用 [Call] 按钮”后，OllyDbg 会自动备份所有的内存、校验、参数、寄存器等信息。并隐藏对话框，然后调用 MessageBoxW 函数。和期望的一样，消息框在屏幕中出现了：



函数 MessageBoxW 不会修改参数。如果你调用的函数更新了内存，比如函数 GetWindowName，修改的字节将会在数据区里高亮。注意：EAX 返回值为 1，表示成功。

其他的例子请访问我的网站：<http://home.t-online.de/home/Ollydbg/LoadDll.htm>

不幸的是，你不能通过这种方式调试 OllyDbg 的插件，插件关联到 ollydbg.exe 文件，Windows 系统不能在同一个应用程序里加载并运行两个可执行文件。

二、解码提示 [Decoding hints]

在某些情况下，分析器不能区分代码和数据。让我们看看下面的例子：

```
const char s[11] = "0123456789";
...for (i=0x30; i<0x3a; i++) t[i-0x30]=s[i-0x30];
```

好的编译器将会将上面的代码优化成如下样子： e

```
for (i=0x30; i<0x3a; i++) (t-0x30)=(s-0x30);
```

这里 `t-0x30` 和 `s-0x30` 都是常量，并编译成如下形式：

```
MOV AL,[BYTE s_minus_30+EBX]
MOV [BYTE t_minus_30+EBX],AL
```

编译器也可能将常量字符串"0123456789"插入到执行代码中。在 1.10 版本中，我打算用寄存器的值来决定是否的数据或代码。当遇到上面的命令，分析器将假定地址 `s_minus_30` 处包含字符数据。但事实上，可能那里是代码。

万一出现上述问题，我们应该怎么办呢？有两种办法：最快最笨的办法是：将分析错误的部分删除（快捷键：退格键），这样 OllyDbg 将使用默认的反汇编器进行解码。

更好的办法是使用解码提示 [decoding hints]。你可以告诉 OllyDbg 如何解释选中的内存内容。这种方法在重新分析 (Ctrl+A) 时，解释依然有效。

设置提示的方法：在反汇编窗口中，选中需要修正提示的代码或数据，然后在快捷菜单中选择 分析 [Analysis] → 在下次分析时，将选择部分视为 [During next analysis, treat selection as]。选择以下选项之一：

命令 [Command] — 第一个被选中的字节开始的有效命令。这条命令，还有所有后面的部分，直到有 Jump 或 Return 命令出现，以及含有 Jump 或 Call 命令所到达位置的部分，都会被视为命令；

字节 [Byte]，字 [Word]，双字 [Doubleword] — 选中的前 1、2、4 字节视为对应大小的数据；

所有选中命令 [Commands] — 全部选中部分（直到第一个无效命令）和可以到达由有效命令集组成的目的地址；

字节 [Bytes]，字 [Words]，双字 [Doublewords]，— 全部选中部分以 1、2、或 4 字节分组；

ASCII 字符串 [ASCII text]，UNICODE 字符串 [UNICODE text] — 全部选中部分为 ASCII 或 UNICODE 字符串；

默认（移除提示）[Default (remove hints)] — 从选中部分中移除全面提示；

移除全部提示 [Remove all hints] — 从全部模块中移除解码提示。

OllyDbg 保存提示到.udd 文件中。

三、表达式赋值 [Evaluation of expressions]

OllyDbg 能够支持非常复杂的表达式。表达式的语法格式将在这个主题的后面进行介绍，但我想你对此不一定真的感兴趣。那么我先举几个实例来说明：

10—常量 0x10（无符号）。所有整数常量都认为是十六进制的，除非后面跟了点；

10.—十进制常量 10（带符号）；

'A'—字符常量 0x41;

EAX—寄存器 EAX 的内容，解释为无符号数;

EAX—寄存器 EAX 的内容，解释为带符号数;

[123456]—在地址 123456 处的无符号双字内容。默认情况，**OllyDbg** 假定是双字长操作数;

DWORD PTR [123456]—同上。关键字 PTR 可选;

[SIGNED BYTE 123456]—在地址 123456 处带符号单字节。**OllyDbg** 支持类 MASM 和类 IDEAL 两种内存表达式;

STRING [123456]—以地址 123456 作为开始，以零作为结尾的 ASCII 字符串。中括号是必须的，因为你要显示内存的内容;

<123456>—在地址 123456 处存储的双字所指向的地址内的双字内容;

2+3*4—值为 14。**OllyDbg** 按标准 C 语言的优先级进行算术运行;

(2+3)*4—值为 20。使用括号改变运算顺序。

EAX.<0—如果 EAX 在 0 到 0x7FFFFFFF 之间，则值为 0，否则值为 1。注意 0 也是有符号的。当带符号数与无符号数比较时，**OllyDbg** 会将带符号数转成无符号数。

EAX<0—总为 0 (假)，因为无符号数永远是正的。

MSG=111—如果消息为 WM_COMMAND，则为真。0x0111 是命令 WM_COMMAND 的数值。**MSG** 只能用于设置在进程消息函数的条件断点内。

[STRING 123456]="Brown fox"—如果从地址 0x00123456 开始的内存为 ASCII 字符串 "Brown fox"、"BROWN FOX JUMPS"、"brown fox???"，或类似的串，那么其值为 1。比较不区分大小写和文本长度。

EAX="Brown fox"—同上，EAX 按指针对待。

UNICODE [EAX]="Brown fox"—**OllyDbg** 认为 EAX 是一个指向 UNICODE 串的指针，并将其转换为 ASCII，然后与文本常量进行比较。

[ESP+8]=WM_PAINT—i 在表达式中你可以使用上百种 Windows API 符号常量。

(BYTE ESI+DWORD DS:[450000+15*(EAX-1)]> & 0F0)!=0—这绝对是个有效的表达式。

现在我们介绍**语法规则**。在大括号 ({}) 内的每个元素都只能出现一次，括号内的元素顺序可以交换:

表达式 = 内存中间码 | 内存中间码 <二元操作符> | 内存中间码

内存中间码 = 中间码 | { 符号标志 大小标志 前缀 } [表达式]

中间码 = (表达式) | 一元操作符 | 内存中间码 | 带符号寄存器 | 寄存器 | FPU 寄存器 | 段寄存器 | 整型常量 | 浮点常量 | 串常量 | 参数 | 伪变量

一元操作符 = ! | ~ | +

带符号寄存器 = 寄存器.

寄存器 = AL | BL | CL ... | AX | BX | CX ... | EAX | EBX | ECX ...

FPU 寄存器 = ST | ST0 | ST1 ...

段寄存器 = CS | DS | ES | SS | FS | GS

整型常量 = <十进制常量> | <十六进制常量> | <字符常量> | <API 符号常量>

浮点常量 = <符点常量>

串常量 = "<串常量>"

符号标志 = SIGNED | UNSIGNED

大小标志 = BYTE | CHAR | WORD | SHORT | DWORD | LONG | QWORD | FLOAT | DOUBLE |

FLOAT10 | STRING | UNICODE

前缀 = 中间码:

参数 = %A | %B // 仅允许在监视器 [inspector] 中使用

伪变量 = MSG // 窗口消息中的代码

这个语法并不严格。在解释[WORD [EAX]]或类似的表达式时会产生歧义。可以理解为以寄存器 EAX 所指向地址的两字节内容为地址，所指向的双字内容；也可以理解为以寄存器 EAX 所指向地址的四字节内容为地址，所指向的两字节内容。而 OllyDbg 会将修饰符尽可能的放在地址最外面，所以在这种情况下，[WORD [EAX]]等价于 WORD 。

默认情况下，**BYTE**、**WORD** 和 **DWORD** 都是无符号的，而 **CHAR**、**SHORT** 和 **LONG** 都是带符号的。也可以使用明确的修饰符 **SIGNED** 或 **UNSIGNED**。例如在二元操作时，如果一个操作数是浮点的，那么另外一个就要转成浮点数；或者如果一个是无符号的，那么另外一个要转成无符号的。浮点类型不支持 **UNSIGNED**。大小修饰符后面跟 MASM 兼容关键字 **PTR**（如：**BYTE PTR**）也允许的，也可以不要 **PTR**。寄存器名和大小修饰符不区分大小写。

你可以使用下面类 C 的运算符（0 级最高）：

优先级	类型	运算符
0	一元运算符	! ~ + -
1	乘除运算	* / %
2	加减运算	+ -
3	位移动	<< >>
4	比较	< <= > >=
5	比较	== !=
6	按位与	&
7	按位异或	^
8	按位或	
9	逻辑与	&&
10	逻辑或	

在计算时，中间结果以 **DWORD** 或 **FLOAT10** 形式保存。某些类型组合和操作是不允许的。例如：**QWORD** 类型只能显示；**STRING** 和 **UNICODE** 只能进行加减操作（像 C 语言里的指针）以及与 **STRING**、**UNICODE** 类型或串常量进行比较操作；你不能按位移动浮点 [**FLOAT**] 类型，等等。

四、自定义函数描述 [Custom function descriptions]

1、概论 [Introduction]

OllyDbg 包含（做为内部资源）1900 多种标准函数以及 400 多种标准 C 函数的名称和参数。分析器 [Analyzer] 用这些描述使被调试程序更加易懂。比较下面一个例子，分析器的函数 **CreateFont**：

```
PUSH 0x00469F2A          ; ASCII "Times New Roman"
PUSH 12
PUSH 2
PUSH 0
PUSH 0
PUSH 0
PUSH 0
```

```

PUSH 0
MOV EAX,DWORD PTR [49FA70]
PUSH EAX
PUSH 190
PUSH 0
PUSH 0
PUSH 0
PUSH 10
CALL <JMP.&GDI32.CreateFontA>

```

这是分析后的：

```

MOV EAX,DWORD PTR [49FA70]
PUSH OT.00469F2A          ; / FaceName = "Times New Roman"
PUSH 12                  ; | PitchAndFamily = VARIABLE_PITCH—>FF_ROMAN
PUSH 2                   ; | Quality = PROOF_QUALITY
PUSH 0                   ; | ClipPrecision = CLIP_DEFAULT_PRECIS
PUSH 0                   ; | OutputPrecision = OUT_DEFAULT_PRECIS
PUSH 0                   ; | CharSet = ANSI_CHARSET
PUSH 0                   ; | StrikeOut = FALSE
PUSH 0                   ; | Underline = FALSE
PUSH EAX                 ; | Italic => TRUE
PUSH 190                 ; | Weight = FW_NORMAL
PUSH 0                   ; | Orientation = 0
PUSH 0                   ; | Escapement = 0
PUSH 0                   ; | Width = 0
PUSH 10                  ; | Height = 10 (16.)
CALL <JMP.&GDI32.CreateFontA>; \ CreateFontA

```

显然，后面的代码更容易理解。API 函数 `CreateFont` 有 14 个参数。分析器标记所有这些参数的名称并解码他们的值。如果寄存器跟踪开启，那么分析器同时会解码参数 `Italic` 的值为地址 49FA70 处双字长的内容。解码使用参数的真实值，所以如果[49FA70]里的内容改变了，那么参数 `Italic` 的值也会随之改变。当 EIP 指向跳转或调用该函数的命令，或指向入口时，OllyDbg 也会在栈中对已知函数的参数进行解码。

OllyDbg 可以对像 `printf()` 这样参数个数可变的函数进行参数解码：

```

PUSH EAX                 ; / <%. *s>
PUSH E8                  ; | <*> = E8 (232.)
PUSH EBX                 ; | <%08X>
PUSH Mymodule.004801D2   ; | format = "Size %08X (%. *s) bytes"
PUSH ESI                 ; | s
CALL Mymodule.sprintf    ; \sprintf

```

你可以定义自己的函数。每次你打开某个应用程序时，OllyDbg 都会重新设置函数参数表并用内嵌描述添充这个表。然后尝试打开文件“< OllyDbg 目录>\common.arg”和“<OllyDbg 目录>\<应用程序名>.arg”，这里<应用程序名>使用 8.3 格式（DOS）被调试程序文件名（不带路径和扩展名）。

下面看一个简单的.arg 文件实例:

```
INFO Simple .ARG file that decodes CreateHatchBrush
TYPE HS_X
    IF 0 "HS_HORIZONTAL"
    IF 1 "HS_VERTICAL"
    IF 2 "HS_FDIAGONAL"
    IF 3 "HS_BDIAGONAL"
    IF 4 "HS_CROSS"
    IF 5 "HS_DIAGCROSS"
    ELSEINT
END
TYPE COLORREF
    IF 0 "<BLACK>"
    IF 00FFFFFF "<WHITE>"
    OTHERWISE
    TEXT "RGB("
    FIELD 000000FF
    UINT
    TEXT ","
    FIELD 0000FF00
    UINT
    TEXT ","
    FIELD 00FF0000
    UINT
    TEXT ")")
END
STDFUNC CreateHatchBrush
    "style" HS_X
    "colorref" COLORREF
END
```

标准 Windos API 函数 CreateHatchBrush(int style,int colorref) 有两个参数。第一个必须是阴影风格 [hatch style], 第二个是常量由红色、绿色、蓝色组成, 并用一个 32 位整数的低三字节表示。为了解码这些参数, 文件定义了两个新的参数类型: HS_X 和 COLORREF。

阴影风格是一个简单的枚举类型, 如 0 表示 HS_HORIZONTAL (水平风格)、1 表示 HS_VERTICAL (垂直风格)。IF 关键字比较参数与第一个操作数 (注意: 其总是十六进制的), 如果相同则显示第二个操作数里的文本。但万一匹配失败会如何? 关键字 ELSEINT 会然 OllyDbg 会将参数解释为一个整数。

COLORREF 更复杂一些。首先尝试解码两个广泛使用的颜色值: 黑 (全 0 组成) 与白 (全 0xFF 组成)。如果匹配失败, COLORREF 尝试解码颜色为一个结构包含红、绿、蓝的亮度。FIELD 会用第一个操作数与参数进行逻辑与操作。然后转换结果为整数, 并同时按位右移第一个操作及该整数, 直到第一个操作数的二进制个位数字为 1, 这时整数按位右移的结果以无符号 10 进制显示出来。这个例子做了三次这样的操作, 以分离出每个颜色成份。TEXT 关键字用于无条件显示文本。如果参数为 00030201, 那么 COLORREF 将其解码为

RGB(1.,2.,3.)。

大多断 API 函数都会从栈中移除参数并保护寄存器 EBX, EBP, ESI 和 EDI。声明这样的函数为 STDFUNC，以告诉分析器该函数做了这样的事情。否则请其描述为 FUNCTION。

万一某个参数由多个域及比特值组成，比如上面提到的 `fdwPitchAndFamily`，我们该怎么办？请看下面这个例子：

```
TYPE FF_PITCH
  MASK 03
  IF 00 "DEFAULT_PITCH"
  IF 01 "FIXED_PITCH"
  IF 02 "VARIABLE_PITCH"
ELSEHEX
TEXT "—>"
  MASK 0C
  BIT 04 "4—>"
  BIT 08 "8—>"
  MASK FFFFFFF0
  IF 00 "FF_DONTCARE"
  IF 10 "FF_ROMAN"
  IF 20 "FF_SWISS"
  IF 30 "FF_MODERN"
  IF 40 "FF_SCRIPT"
  IF 50 "FF_DECORATIVE"
ELSEHEX
END
```

前两个比特位（第 0 和等 1 位）表示倾斜度，必须一起解码。我们使用 `MASK 03` 来提取这两个比特并通过 `IF` 序列来解码。增加了连接符“—>”，分别提取第 2 和第 3 个比特位，并分别单独解码。最后提取剩余部分并进行解码。

OllyDbg 会移除生成串尾部的连接符“—>”、空格、冒号、逗号、分号和等号。

目前版本的分析仅能够解码 32 位参数。如你不能解码双精度浮点或长双精度浮点的函数参数。

2、格式描述

自定义解码信息由函数描述和类型描述两部分组成。函数描述部分非常的简单：

```
FUNCTION|STDFUNC [模块名]函数名
  <第一个参数的名称> <第一个参数的类型>
  .....
  <最后一个参数的名称> <最后一个参数的类型>
END
```

如果函数从栈中移除参数并保护寄存器 EBX, EBP, ESI 和 EDI，请使用关键字 `STDFUNC`。大多少函数都遵循这样的规则。其他情况则声明为 `FUNCTION`。模块（EXE 或 DLL）名是可选的。如果模块名被忽略，OllyDbg 会对尝试匹配任何模块。模块名不区分大

小写。

函数名称总是区分大小写的。有针对 UNICODE 的函数必须使用后缀 A 或 W 加以区分，比如 SetWindowTextA。

参数的顺序又 C 风格的参数使用惯例一致。而 16 位 Windows 和 32 位 API 函数也是按惯例使用。如果参数名由多个字组成，或者包含特殊字符，那么请将其用两个单引号引起来。与在 C 语言中一样，省略号（）是一个特殊的记录用于表示参数数目可变。它必须在函数描述的最后。OllyDbg 不会尝试解码这样的参数。如果函数的参数为空，则按 functionname(void)对待。

OllyDbg 仅支持 32 位的参数。某些参数已经预定义好了：

INT	以十六进制和带符号整数两种格式显示值
UINT	以十六进制和无符号整数两种格式显示值
HEX	以十六进制格式显示值
BOOL	TRUE 或 FALSE
CHAR	ASCII 字符
WCHAR	UNICODE 字符
FLOAT	32 位浮点数
ERRCODE	系统错误代码（像由函数 GetLastError()报告的）
ADDR, PTR	地址（特殊情况：NULL）
ASCII	ASCII 串指针
UNICODE	UNICODE 串指针
FORMAT	在类似函数 printf()（不包括 wscanfW()!）使用的 ASCII 格式串
WFORMAT	类似函数 wsprintfW()（不包括 scanf()!）使用的 UNICODE 格式串
RECT	RECT（矩形）结构指针
MESSAGE	MSG（ASCII 窗口消息）结构指针
WMESSAGE	MSG（UNICODE 窗口消息）结构指针
HANDLE	句柄（特殊情况：NULL, ERROR_INVALID_HANDLE）
HWND	窗口句柄
HMODULE	模块句柄
RSRC_STRING	带索引的资源串
NULL, DUMMY	有参数，但解码时跳过了

你不能重定义预定义类型。自定义类型允许你将参数分离成几个域并分别解码。类型描述有以下几种格式：

TYPE 类型名

[TEXT "任何文本"]

[<域选择器>]

<域解码>

<域解码>

[TEXT "任何文本"]

[PURGE]

...

<域选择器>

<域解码>

```
<域解码>
[TEXT "任何文本"]
END
```

类型名的程度限制在 16 个字符以内。OllyDbg 会无条件将"任何文本"作为生成的解码。域选择器提取一部分参数用于解码。以下域选择器，可以用于提取域：

MASK 十六进制掩码—域等于参数同十六进制掩码按位与（AND）的结果。

FIELD 十六进制掩码—参数同十六进制掩码按位与（AND）的数值，然后 OllyDbg 同时按位右移掩码和计算的数值直到掩码的二进制个位为 1，这时数值按位右移的结果就是域的值。例如参数 0xC250，FIELD F0，得到的结果是 5。

SIGFIELD 十六进制掩码—参数同十六进制掩码按位与（AND）的数值，然后 OllyDbg 同时按位右移掩码和计算的数值直到掩码的二进制个位为 1，这时数值按位右移的结果转成带符号 32 位数就是域的值。例如参数 0xC250，SIGFIELD FF00，得到的结果是 0xFFFFF2C。

简单域的解码会一次显示整个域的内容：

HEX—以十六进制形式显示域内容；

INT—以带符号十进制形式显示域内容（带小数点）；

UINT—以无符号十进制形式显示域内容（带小数点）；

CHAR—以 ASCII 字符形式显示域内容。

域若是一个枚举类型，则可以使用 IF 序列，如果必要的话还可以在 IF 序列后跟关键字 TRYxxx 与 ELSExxx：

IF 十六进制值 "文本"—如果域等于十六进制值，则将文本作为输出字符串；

TRYASCII—如果域是一个指向 ASCII 串的指针，则显示这个串；

TRYUNICODE—如果域是一个指向 UNICODE 串的指针，则显示这个串；

TRYORDINAL—如果域是一序号（有 16 位均为 0），则会显示为序号（“#”后跟整数）；

OTHERWISE—如果前面 IF 语句为真，则停止解码，否则继续解码；

ELSEINT—如果前面所有的 IF 和 TRYxxx 语句均失败，则以带符号十进制数形式（带小数点）显示这个域；

ELSEHEX—如果前面所有的 IF 和 TRYxxx 语句均为失败，则以十六进制形式显示这个域；

ELSECHAR—如果前面所有的 IF 和 TRYxxx 语句均为失败，则以 ASCII 字符形式显示这个域；

ELSEWCHAR—如果前面所有的 IF 和 TRYxxx 语句均为失败，则以 UNICODE 字符形式显示这个域。

如果域是一个二进制位集，则可以使用 BIT 序列，如果必要的话可以后面跟关键字 BITZ 与 BITHEX：

BIT 十六进制掩码 "文本"—如果值与十六进制掩码按位与（AND）的结果不是 0，则将文本做为输出串；

BITZ 十六进制掩码 "文本"—如果值与十六进制掩码按位与（AND）的结果是 0，则将文本做为输出串；

BITHEX 十六进制掩码—如果值与十六进制掩码按位与（AND）的结果不是 0，则将结果以十六进制形式显示。

特殊关键字 **PURGE** 会从输出串尾部移除以下几种符号：

```
空格      ''
逗号      ','
或        '—>'
```


冒号 ':'
等于 '='

这会让某些解码情况变的简单。关键字 END 是类型定义结尾标记并会自动运行 PURGE 命令。

3、预编译类型

OllyDbg 在预编译资源时，已经包含 150 多种类型描述。以下列出了一部分。你可以在自定义文件中直接使用这些类型：

LANG_X—操作系统语言 ID (0—未知、9—语言、C—法语，等等)
GENERIC_X—访问类型 (GENERIC_READ, GENERIC_WRITE...)
FILE_SHARE_X—共享类型 (FILE_SHARE_READ, FILE_SHARE_WRITE)
CREATEFILE_X—文件创建模式 (CREATE_NEW, OPEN_EXISTING...)
FILE_ATTRIBUTE_X—文件属性 (READONLY, SYSTEM, DELETE_ON_CLOSE...)
RT_AX—资源类型 (RT_CURSOR, RT_GROUP_ICON, ASCII string...)
RT_WXX—资源类型 (RT_CURSOR, RT_GROUP_ICON, UNICODE string...)
COORD—坐标结构 "(X=xxx,Y=yyy)"
STD_IO_X—标准句柄 (STD_INPUT_HANDLE, STD_ERROR_HANDLE...)
GMEM_X—全局内存类型 (GMEM_FIXED, GPTR...)
LMEM_X—局部内存类型 (LMEM_FIXED, LPTR...)
FSEEK_X—文件查找类型 (FILE_BEGIN, FILE_CURRENT...)
OF_X—文件模式 (fOF_READ, OF_SHARE_COMPAT, OF_VERIFY...)
O_X—文件创建模式 (O_RDONLY, O_BINARY, SH_COMPAT...)
SEMAPHORE_X—信号量类型 (SEMAPHORE_ALL_ACCESS, SYNCHRONIZE...)
SLEEP_TIMEOUT—超时 (INFINITE 或时间)
ROP—一些标准栅格运算标志代码 (ROP) (SRCCOPY, MERGEPAIN...)
COLORREF—RGB 颜色值 ("<WHITE>", "RGB(rr,gg,bb)"...)
WS_X—窗口风格 (WS_OVERLAPPED, WS_POPUP...)
WS_EX_X—扩展窗口风格 (WS_EX_DLGMODALFRAME, WS_EX_TOPMOST...)
MF_X—菜单标志 (MF_BYPOSITION, MF_ENABLED...)
WM_X—ASCII 窗口消息类型 (WM_CREATE, WM_KILLFOCUS, CB_SETCURSEL...)
WM_W—UNICODE 窗口消息类型 (WM_CREATE, WM_KILLFOCUS, CB_SETCURSEL...)
VK_X—虚拟键盘代码 (VK_LBUTTON, VK_TAB, VK_F10...)
MB_X—message box style (MB_OK, MB_ICONHAND...)
HKEY_X—预定义注册表句柄 (HKEY_CLASSES_ROOT, HKEY_LOCAL_MACHINE...)

还有更多的预编译类型。如果常量在它文件被定义为 ABC_ xxxxxxxx，那么一般就有 ABC_X 预编译类型。

注意：

1.如果 OllyDbg 是即时调试器，并且在 Windows 95 下挂接执行了 DebugBreak() 的应用程序，则这个应用程序在挂接后，还会运行。在基于 NT 的系统下，应用程序应该会在 DebugBreak() 暂停。

2. 命令 SMSW (保存机器状态字 [Store Machine Status Word])。这个命令仅接受寄存器 AX 作为参数，而程序编译成 EAX 接受参数。