常规 WinDbg 命令（主题分组）

原作：Robert Kuster 写于 2007 年 11 月。版权归属于原作者。 http://www.software.rkuster.com

翻译：不死怨灵

1) 内置帮助命令

命令	变型/参数	描述
?	? ? /D	显示常用命令 显示常用命令和 DML
.help	.help .help /D .help /D a*	显示“.”命令 以 DML 形式显示“.”命令（顶部已给出链接） 以 DML 形式显示 a 开头的“.”命令（通配符）
.chain	.chain .chain /D	列出所有加载的调试器扩展 以 DML 形式列出所有加载的调试器扩展（扩展会链接到.extmatch 命令）
.extmatch	.extmatch /e ExtDLL 函数过滤 .extmatch /D /e ExtDLL 函数过滤	显示一个扩展 DLL 的输出函数。函数过滤 = 通配字符串 以 DML 方式显示。（函数链接指向”!Ext 名称.help 函数名”命令） 例如：**.extmatch /D /e uext *** （显示所有 uext.dll 的输出函数）
.hh	.hh .hh 文本	打开 WinDbg 的帮助文件 文本 = 在帮助文件的索引中要查找的文本 例如： **.hh dt**

<<返回顶部

2) 常用 WinDbg 命令（显示版本、清空屏幕等）

命令	变型/参数	描述
version		转储调试器和所加载的扩展的版本信息
vercommand		转储命令还用于启动调试器
vertaget		目标计算机的版本
Ctrl+Alt+V		切换详细模式开/关 在详细模式下，一些命令（如寄存器转储）会更加详细的输出
n	n [8 | 10 | 16]	设置数基

.formats	.formats 表达式	显示数字格式 = 求一个数值表达式或符号的值，并用多种数值格式表达它（16 进制、10 进制、8 进制、2 进制、时间……） 例 1：.formats 5 例 2：.formats poi(nLocal1) == .formats @@($!nLocal1)
.cls		清空屏幕
.lastevent		显示最近一次发生的例外或事件（为什么调试器要等待？）
.effmach	.effmach .effmach . .effmach # .effmach x86 | amd64 | ia64 | ebc	转储有效机器（x86,amd64, ...）: 使用目标电脑本身的处理器模式 使用执行最近事件代码的处理器模式 使用 x86,amd64,ia64 或 ebc 处理器模式 该设置影响许多调试器的功能： ->这些处理器的辗转开解器用于栈跟踪 ->这些处理器的寄存器是激活的
.time		显示时间（系统启动、进程启动、内核时间、用户时间）

<<返回顶部

3) 调试会话（附加、分离……）		
命令	**变型/参数**	**描述**
.attach	PID	附加到一个进程
.detach		结束调试会话，但保留一切用户模式目标程序的运行
q	q, qq	退出 = 结束调试会话并且终止目标程序 远程调试：q = 无影响；qq = 终止调试服务器
.restart		重启目标程序

<<返回顶部

4) 表达式与命令		
命令	**变型/参数**	**描述**
;		命令分隔符（命令 1; 命令 2; ……）
?	? 表达式 ?? 表达式	求表达式的值（使用默认求值器） 求 c++表达式的值
.expr	.expr .expr /q .expr /s c++ .expr /s masm	选择默认的表达式求值器 显示当前求值器 显示可用的求值器 设置 **c++**为默认求值器 设置**宏汇编**为默认求值器
*	* [任意文本]	注释行说明符 终止于：行末
$$	$$[任意文本]	注释说明符 终止于：行末或分号
.echo	.echo 字符串 .ecoh "字符串"	响应注释 -> 注释文本 + 响应它 终止于：行末或分号 若拥有$$标记或*标记，调试器会忽略输入文本而不响应

<<返回顶部

5) 调试器标识语言（DML）

自 6.6.07 版本的调试器开始，加入了一种新的调试器与扩展的增强输出机制：DML

DML 允许以标签的形式输出正规命令和额外的非显示信息

调试器的用户界面分析出额外的信息用以提供新的行为

DML 主要是为了解决两个问题：

- **链接到相关信息**
- **调试器与扩展功能的可发掘性**

命令	变型/参数	描述
.dml_start		转到其他的 DML 命令
.prefer_dml	.prefer_dml [1 | 0]	全局设置：DML 默认是否为 DML 增强命令? 注释许多命令，诸如：k, lm, ...输出后附加 DML 内容
.help /D		.help 加入了一个新的 DML 模式，顶部给出链接
.chain /D		.chain 加入了一个新的 DML 模式，扩展会链接到.extmatch
.extmatch /D		.extmatch 加入了一个新的 DML 模式，输出函数链接指向"!Ext 名称.help 函数名"命令
lmD		lm 加入了一个新的 DML 模式，模块名指向 lmv 命令
kM		k 加入了一个新的 DML 模式，框体数指向.frame /dv
.dml_flow	.dml_flow 起始地址 目标地址	允许对函数进行代码流程的交互式搜寻 1．为一个开始于给出的起始地址的函数构造一个代码流程图（类似于 uf） 2．显示由当前程序块给出的基本程序块地址同时连接着将要提交的程序块和程序块指向 例如：**.dml_flow CreateRemoteTread CreateRemoteTread+30**

<<返回顶部

6) 主扩展

命令	变型/参数	显示支持的命令
!Ext.help		通常扩展
!Exts.help		-||-
!Uext.help		用户模式扩展（非特定系统）
!Ntsdexts.help		用户模式（特性系统）
!logexts.help		记录器扩展
!clr10\sos.help		调试管理模式
!wow64exts.help		Wow64 调试器扩展
!Wdfkd.help		内核模式驱动构架扩展
!Gdikdx.help		图形驱动扩展
...		
!名称.help	!名称.help 函数	显示关于输出函数的详细帮助 名称 = 扩展 DLL 的占位符 函数 = 输出函数的占位符 例如：**!Ntsdexts.help handle**（显示!Ntsdexts.handle 的详细信息）

<<返回顶部

7) 符号

命令	变型/参数	描述
ld	ld 模块名 ld *	为模块加载符号 为所有模块加载模块
!sym	!sym !sym noisy !sym quiet	获取符号加载情况 设置为**噪声**符号加载模式（调试器显示出它搜寻符号的信息） 设置为**安静**符号加载模式（默认）
x	x *[选项]* 模块!符号 x /t ... x /v ... x /a ... x /n ... x /z ...	**审核符号**：显示与制定样式相匹配的符号 带数据类型 详情（符号类型与大小） 按地址分类 按名称分类 按大小分类（函数符号的大小是指函数在内存中的大小）
ln	ln 地址	**列出最近的符号** = 显示在或接近给出地址的符号。用于： 1. 确定指针指向的位置 2. 当我们看到一个损坏的栈时确定是哪一个程序在调用
.sympath	.sympath .sympath+	显示或设置符号的检索路径 向之前的符号路径追加目录
.sumopt	.symopt .symopt+ *标识* .symopt- *标识*	显示当前符号选项 增加选项 移除选项
.symfix	.symfix .symfix+ DownstreamStore	设置符号存储路径自动指向 http://msdl.microsoft.com/download/symbols + = 附加到现有的路径 DownstreamStore = 目录被用作下载存储。默认是 WinDbg 安装路径\Sym
.reload	.reload .reload [/f | /v] .reload [/f | /v] 模块	从所有模块中重载符号信息** f = 强制立即加载符号（优先于延迟加载）；v = 详细模式 模块 = 仅用于模块 **注意：.reload 命令实际上并不会引起符号信息被读取。它只是让调试器知道符号文件可能已经改变，或者一个新的模块将被加到模块列表中。要强制符号加载的发生需要使用/f 项，或是 ld（加载符号）命令

扩展

x *!	列出所有模块
x ntdll!*	列出 ntdll 的所有符号
x /t /v MyDll!*	列出 MyDll 的所有符号的数据类型、符号类型和大小
x kernel32!*LoadLib*	列出 kernel32 中所有包含 LoadLib 的符号
.sympath+ C:\MoreSymbols	从 C:\MoreSymbols（本地文件夹）中添加符号
.reload /f @"ntdll.dll"	立刻从 ntdll.dll 中重载符号
.reload /f @"C:\WINNT\System32\verifier.dll"	从 verifier 中重载符号。使用给出的路径

同时检查"!lmi"命令

<<返回顶部

8) 源码		
命令	**变型/参数**	**描述**
.srcpath	.srcpath .srcpath+ 目录	显示或设置源码的检索路径 将目录添加到检索到的源码路径
.srcnoisy	*{1|0}*	控制噪声源码的加载
.lines	[-e | -d | -t]	切换源码行支持：启用；禁用；切换
l（小写 L）	l+l, l-l l+o, l-o l+s, l-s l+t, l-t	显示行数 除了[s]隐藏一切 源码和行数 源码模式对汇编模式

<<返回顶部

9) 例外、事件与事故分析		
命令	**变型/参数**	**描述**
g	g gN gH	运行 运行例外已处理 运行而不处理
.lastevent		发生了什么？显示最近的事件或例外
!analyze	!analyze -v !analyze -hang !analyze -f	显示当前例外或故障检查信息；详细 用户模式：分析线程栈以确认是否有哪个线程正阻止其他线程 查看例外分析，即便是调试器没有检测到例外
sx	sx sxe sxd sxn sxi sxr	显示所有中断情况与处理的事件过滤器 中断首次偶然事件 中断第二次偶然事件 通知；不要中断 忽略事件 重置过滤器设置为默认值
.exr	.exr -1 .exr 地址	显示最近的例外记录 显示地址处的例外记录
.ecxr		结合当前例外显示例外环境记录（寄存器）
!cppexr	地址	显示 C++例外的内容与类型

扩展

.exr -1	显示最近的例外
.exr 7c901230	显示地址 7c901230 地址处的例外
!cppexr 7c901230	显示地址 7c901230 地址处的 C++例外

<<返回顶部

10) 加载模块与映像信息		
命令	**变型/参数**	**描述**

lm	lm[v | l | k | u | f] *[m 型]* lmD	列出模块；详细|带加载符号|仅内核符号信息|仅用户符号信息|映像路径；模块名必须匹配。 lm 的 DML 模式；输出中包含 lmv 的链接
!dlls	!dlls !dlls -i !dlls -l !dlls -m !dlls -v !dlls -c 模块地址 !dlls -?	按照**加载序数**列出所有加载的模块 按照初始化顺序 按加载顺序（默认） 按内存顺序 显示详细信息 仅显示模块地址处的模块 简要帮助
!imgreloc	*映像基地址*	偏移映像的信息
!lmi	*模块*	模块的详细信息（包括准确的符号信息）
!dh	!dh *映像基地址* !dh -f *映像基地址* !dh -s *映像基地址* !dh -h	转储映像基地址的头部 f = 仅文件头 s = 仅区块头 h = 简要帮助 !lmi 扩展提取出映像头中最重要的信息并以简单摘要的形式显示出。这往往比!dh 更有用。

扩展

lm	显示所有已加载和未加载的模块
lmv m kernel32	显示 kernel32.dll 的详细（尽可能）信息
lmD	DML 型的 lm
!dlls -v -c kernel32	显示 kernel32.dll 的信息，包括**加载序数**
!lmi kernel32	显示 kernel32.dll 的详细信息，包括**符号信息**
!dh kernel32	显示 kernel32.dll 的头部

<<返回顶部

11）进程相关信息

命令	变型/参数	描述
!dml_proc		(DML)显示当前进程并且允许跟进进程获取更多信息
| (信道)		显示所有正被调试的进程的状态
.tlist		列出系统正在运行的所有进程
!peb		显示进程环境块（PEB）的标准视图

扩展

!peb	转储进程 PEB 标准视图（仅某些信息）
r $peb	转储 PEB.$peb 地址 == 伪寄存器
dt ntdll!_PEB	转储 PEB 结构
dt ntdll!_PEB @$peb -r	递归(-r)转储我们线程的 PEB

<<返回顶部

12) 线程相关信息		
命令	**变型/参数**	**描述**
~	~ ~* *[命令]* ~. *[命令]* ~# *[命令]* ~数目 *[命令]* ~~[TID] *[命令]* ~Ns	列出线程 所有线程 当前线程 当前事件或例外引发的线程 序号为给出数目的线程 ID 为 TID 的线程（需加方括号） 转换到线程 N（新的当前线程） *[命令]：可使用一些标准命令，诸如：k,r*
~e	~* e *命令串* ~. e *命令串* ~# e *命令串* ~数目 e *命令串*	为如下目标执行“线程特定”命令（命令串 = 一个或多个要执行的命令）： 所有线程 当前线程 当前时间引发的线程 对应序号的线程
~f	~线程 f	冻结线程（参见线程的~语法）
~u	~线程 u	解冻线程（参见线程的~语法）
~n	~线程 n	挂起线程（参见线程的~语法）
~m	~线程 m	恢复线程（参见线程的~语法）
!teb		显示线程环境块（TEB）的标准视图
!tls	!tls -1 !tls 空位索引 !tls [-1 | 空位索引] TEB 地址	-1 = 转储当前线程的所有空位 空位索引 = 仅转储指定的空位 TEB 地址 = 特定线程；缺省则使用当前线程
.ttime		显示线程时间（用户+内核模式）
!runaway	[标识：0 | 1 | 2]	显示每个线程消耗时间的信息（0-用户时间，1-内核时间，2-线程创建起持续的时间）。找出哪些线程失控或是消耗过多 CPU 时间的捷径
!gle	!gle !gle -all	转储当前线程的最终错误 转储所有线程的最终错误 兴趣点： **SetLastError(dwErrCode)**检测 kernel32!g_dwLastErrorToBreakOn 的值并尽可能的运行一个 DbgBreakPoint. **if**((g_dwLastErrorToBreakOn != 0) && (**dwErrCode == g_dwLastErrorToBreakOn**)) **DbgBreakPoint();** 不足之处在于 SetLastError 仅能在 KERNEL32.DLL 内部被调用。而其它调用将被传递到一个位于 NTDLL.DLL 的函数——RtlSetLastWin32Error.
!error	!error 错误值 !error 错误值 1	解码并显示一个错误值的信息 将错误值作为 NTSTATUS 代码处理

扩展

~* k	调用所有线程的栈 ~ !uniqstack
~2 f	冻结 TID=2 的线程

~# f	冻结由当前事件发起的线程
~3 u	解冻 TID=3 的线程
~2e r; k; kd	等同于：~2r; ~2k; ~2kd
~*e !gle	会在每一个独立线程被调试时重复扩展命令!gle
!tls -1	为当前线程转储所有 TLS 空位
!runaway 7	1（用户模式） + 2（内核模式） + 4（线程运行时间）
!teb	转储线程 TEB 标准视图（仅部分信息）
dt ntdll!_TEB @$teb	转储当前线程 TEB

<<返回顶部

13) 断点

命令	变型/参数	描述
bl		列出所有断点
bc	bc * bc # [#] [#]	清除所有断点 清除#断点
be	be * be # [#] [#]	启用所有断点 启用#断点
bd	bd * bd # [#] [#]	禁用所有断点 禁用#断点
bp	bp [地址] bp [地址] ["命令串"] [~线程] bp[#] [选项] [地址] [Passes] ["命令串"]	在地指出设置断点 命令串 = 命令 1；命令 2；……每次停在断点处均执行 ~线程 = 线程处断点依然执行 # = 断点 ID Passes = 激活#Passes 之后的断点（之前的忽略）
bu	bu [地址] 参见 bp	设置未实现断点。模块被加载时断点将被设置
bm	bm **符号型** bm 符号型 ["命令串"] [~线程] bm [选项] 符号型 [#Passes] ["命令串"]	设置符号断点。符号型可包含通配符 命令串 = 命令 1；命令 2；……每次停在断点处均执行 ~线程 = 线程处断点依然执行 Passes = 激活#Passes 之后的断点（之前的忽略） **bm 符号型** 等同于使用 **x 符号型** 命令之后对每个结果使用 bu 命令
ba	ba [r|w|e] [大小] 地址 [~线程] ba[#] [r|w|e] [大小] [选项] [地址] [Passes] ["命令串"]	访问时中断：[r=读/写；w=写入；e=执行]，大小=[1|2|4 字节] ~线程 = 线程处断点依然执行 # = 断点 ID Passes = 激活#Passes 之后的断点（之前的忽略）
br	br 旧 ID 新 ID [旧 ID2 新 ID2……]	为一个或多个 ID 重新编号

扩展

使用 bp 命令，断点位置总是指向一个地址。相较之下，bu 和 bm 命令中断点则往往与符号值相关联

简单例子

bp `模块!source.c:12`	在指定的源码处设置断点
bm myprogram!mem*	符号型等同于使用 x 符号型
bu my 模块 ule!func	my 模块 ule 被加载后立即设置断点
ba w4 77a456a8	写入时中断
bp @@(MyClass::MyMethod)	方法处中断（适用于相同方法被重载并因此出现于多处地址）

断点选项

断点仅被触发一次

bp 模块!地址 /1

断点将于k-1 处之后开始触发

bp 模块!地址 k

断点命令：当断点触发时命令开始被执行

每次断点触发则创建一份记录

ba w4 81a578a8 "k;g"

每次断点触发则建立一个转储

bu my 模块 ule!func ".dump c:\dump.dmp; g"

DllMain 调用 MYDLL -> 检查原因

bu MYDLL!DllMain "j (dwo(@esp+8) == 1) '.echo MYDLL!DllMain -> DLL_PROCESS_ATTACH; kn' ; 'g' "

调用 LoadLibraryExW(任意 DLL) -> 显示任意 DLL 名称

bu kernel32!LoadLibraryExW ".echo LoadLibraryExW for ->; du dwo(@esp+4); g"

调用 LoadLibraryExW(MYDLL)? -> 仅在 LoadLibrary 调用 MyDll 时中断

bu kernel32!LoadLibraryExW ";as /mu ${/v:MyAlias} poi(@esp+4); .if ($spat(\"${MyAlias}\", \"*MYDLL*\") != 0) { kn; } .else { g }"

- LoadLibrary 的第一个参数（位于 ESP+4 处）是一个指向问题中 DLL 名的字符串指针。
- 宏汇编的$spat 操作会将该指针与一个预定义的字符串通配符，在我们的例子中是*MYDLL*。
- 遗憾的是$spat 可以使用别名或常量，但不能使内存指针。这就是为什么在我们的问题中要首先将字符串存储为一个别名（MyAlias）。
- 我们的 kernel32!LoadLibraryExW 断点仅在形式与$spat 相匹配时触发。否则程序会继续执行

跳过一个函数的执行

bu sioctl!DriverEntry "r eip = poi(@esp); r esp = @esp + 0xC; .echo sioctl!DriverEntry skipped; g"

- 在一个函数入口点处，值建立于栈顶部建立并包含返回值

  r eip = poi(@esp) -> 设置 EIP（指令指针）指向一个建立在偏移量 0x0 处的值
- DriverEntry 有 2x4 字节参数 = 8 字节 + 4 字节返回地址 = 0xC

  r esp = @esp + 0xC -> ESP（栈指针）加上 0xC，有效的碾转开解对战指针

bu MyApp!WinMain "r eip = poi(@esp); r esp = @esp + 0x14; .echo WinSpy!WinMain entered; g"

- WinMain 有 4x4 字节参数 = 0x10 字节 + 4 字节返回地址 = 0x14

如何在你的编程代码中设置一个断点？

- kernel32!DebugBreak
- ntdll!DbgBreakPoint
- __asm int 3 (x86 only)

<<返回顶部

14) 步入与步过（F10,F11）

无论是单步执行一个单独的汇编指令还是一个单独的源码行，都依赖于调试器是处于汇编模式还是源码模式。

使用 l+t 和 l-t 命令或是 WinDbg 工具栏的按钮在这两个模式之间切换。

命令	变型/参数	描述
g (F5)	g gu	运行（F5） Go up = 执行至当前函数完成 gu ~= g @$ra gu ~= bp /1 /c @$csp @$ra;g -> $csp = 等同于 x86 中的 esp -> $ra = 栈中当前的返回地址
p (F10)	p pr p **计数** p [计数] **"命令"** p =起始地址 [计数] ["命令"] [~线程] p [=起始地址] [计数] ["命令"]	**单步步过**——执行一个单独指令或是一个源码行。子程序按照一个单步来处理。 切换显示寄存器和标识符 计数 = 所要步过的指令或是源码行的数目 命令 = 步过后调试器所要执行的命令 起始地址 = 指定执行的起始地址。默认为当前的 EIP. ~线程 = 指定线程解冻而其他线程冻结
t (F11)	t ...	**单步步入**——执行一个单独指令或是一个源码行。子程序也要每步跟进。
pc	pc ...	**步过至下一个 call**——执行程序直至到达一个 call 指令 若此时 EIP 正指向一个 **call** 指令,则整个 call 函数将被执行。call 返回后，执行将继续进行直至到达另一个 **call** 指令。
tc	tc ...	**步入至下一个 call**——执行程序直至到达一个 call 指令 若此时 EIP 正指向一个 **call** 指令，则调试器将进入 call 并继续执行，直至到达另一个 **call** 指令。
pa	pa 终止地址 par pa 终止地址 **"命令"** pa = **起始地址** 终止地址 ["命令"]	**步过至地址；**终止地址 = 执行结束的地址 调用的函数被视为一个整体。 切换显示寄存器和标识符 命令 = 步过后调试器所要执行的命令 起始地址 = 指定执行的起始地址。默认为当前的 EIP.
ta	ta 终止地址 ...	**步入至地址；**终止地址 = 执行结束的地址 调用的函数同样步入。
wt	wt wt [选项] [= 起始地址] [结束地址] wt -l 深度... wt -m 模块 [-m 模块 2] ... wt -i 模块 [-i 模块 2] ... wt -oa ... wt -or ... wt -oR ... wt -nc ... wt -ns ... wt -nw ...	**步入并查看数据。**到一个函数的起始处执行 **wt.**将会运行全部函数并显示统计数据。 起始地址 = 执行开始；结束地址 = 结束步入处的地址（默认为当前函数的 RET 后） l = 跟进 call 的最大深度 m = 阻止步入到模块 i = 忽略模块中的代码 oa = 转储调用位置的实际地址 or = 转储子函数返回的寄存器值（EAX 值） oR = 以适合的类型转储返回的寄存器值（EAX 值） nc = 无特殊调用信息 ns = 无摘要信息 nw = 无警告
.step_filter	.step_filter	转储当前过滤单 = 步入（t, ta, tc）时函数将被跳过

	.step_filter "过滤单" .step_filter /c	过滤单 = 过滤器 1; 过滤器 2; ...符号连同函数将被步过（跳过） 清除过滤单 .step_filter 在汇编模式下并不很有用，每个函数的调用会在不同的行

扩展

g	运行
g `:123`; ? poi(counter); g	执行当前程序至源码行：123；显示计数器值；恢复运行
p	单步步过
pr	切换为显示寄存器
p 5 "kb"	执行五步步过，之后执行"kb"
pc	步过至下一个 CALL 指令
pa 7c801b0b	步过直至到达 7c801b0b
wt	步入并查看子函数
wt -l 4 -oR	步入子函数，最大至深度 4.显示返回值

<<返回顶部

15) 调用栈

命令	变型/参数	描述
k	k [n] [f] [L] [#帧] kb ... kp ... kP ... kv ...	转储栈；n = 带帧号；f = 相邻帧间距；L = 省略源码行；要显示的栈帧数 前三项 所有项：项的类型+名称+值 所有项格式化（换行） FPO 信息，调用协议
kd	kd [字数]	显示原始栈信息+可能的符号信息 == 完整 esp
kM		DML 变型，链接到.frame #; dv 命令
.kframes		设置栈长度。默认是 20（0x14）
.frame	.frame .frame # .frame /r [#]	显示当前帧 指定帧号 显示寄存器值 .frame 命令会指定用哪个本地环境（处理范围）来解释本地变量，或显示当前本地环境。 当执行一个临近调用时，处理器将 EIP 寄存器（包含 CALL 指令所调用的指令的偏移量）中的值压入栈中（用于此后的返回指令指针）。这是创建一个帧的第一步。每进行一次函数调用就会有一个帧被创建，故此被调用的函数可以访问参数、创建本地变量，并提供一个机制用于返回调用函数。帧的构成依赖于函数调用协议。
!uniqstack	!uniqstack !uniqstack [b|v|p] [n] !uniqstack -?	显示所有线程的栈 [b=前三项, v=FPO+调用协议, p=所有项: 项类型+名称+值], [n=带帧号] 简要帮助
!findstack	!findstack Symbol !findstack Symbol [0|1|2]	指出所有包含符号和模块的栈 [0 = 仅显示 TID，1 = TID + 帧， 2 = 全部线程栈]

	!findstack -?	简要帮助

扩展

k	显示调用栈
kn	调用带帧号的栈
kb	显示调用前三项栈
kb 5	仅显示调用前五帧

从栈中获取多于 3 项的函数参数

dd ChildEBP+8 (Parameters start at ChildEBP+8)

dd ChildEBP+8 (frame X) == dd ESP (frame X-1)

!uniqstack	获取进程的所有栈（每个线程一个）
!findstack kernel32 2	显示所有包含 kernel32 的栈
.frame	显示当前帧
.frame 2	为 2 号帧设置本地环境
.frame /r 0d	显示 0 号帧中的寄存器

<<返回顶部

16) 寄存器

命令	变型/参数	描述
r	r	转储所有寄存器
	r **寄存器 1，寄存器 2**	仅转储指定寄存器（即：**r eax, edx**）
	r 寄存器=**值**	将值分配给寄存器（即：**r eax=5, edx=6**）
	r 寄存器:**类型**	类型 = 用哪种数据格式显示寄存器（即：**r eax:uw**） ib = 有符号字节 ub = 无符号字节 iw = 有符号字（2 字节） uw = 无符号字（2 字节） id = 有符号双字（4 字节） ud = 无符号双字（4 字节） iq = 有符号四字（8 字节） uq = 无符号四字（8 字节） f = 32 位浮点 d = 64 位浮点
	r 寄存器:[数目]类型	数目=要显示的部分的数目（即：**r eax:1uw**） 默认为全寄存器长度，因此当 EAX 作为一个 32 位寄存器时 **r eax:uw** 会显示两个值
	~线程 r [寄存器:[数目]类型]	线程=要从寄存器中读取的线程（即：**~1 r eax**）
rM	rM 掩码	以掩码转储指定类型的寄存器
	rM 掩码 寄存器 1，寄存器 2	仅转储当前掩码中指定的寄存器
	rM 掩码 寄存器=值	分配给寄存器的值
	...	掩码标识

		0x1 = 基本整型寄存器 0x4 =浮点寄存器 == rF 0x8 = 段寄存器 0x10 = 多媒体扩展指令集（MMX）寄存器 0x20 = 调试寄存器 0x40 = 单指令多数据流扩展内存管理（SSE XMM）寄存器 == rX
rF	rF rF 寄存器 1，寄存器 2 rF 寄存器=值 ...	转储所有浮点寄存器 == rM 0x4 仅转储制定的浮点寄存器 分配给寄存器的值
rX	rX rX 寄存器 1，寄存器 2 rX 寄存器=值 ...	转储所有 SSE XMM 寄存器 == rM 0x40 仅转储制定的 SSE XMM 寄存器 分配给寄存器的值
rm	rm rm ? rm 掩码	转储默认寄存器掩码。这个掩码控制着寄存器如何用"r"命令显示 转储可能的掩码位的列表 指定显示寄存器时使用的掩码

扩展

rm ?	显示可能的掩码位
rm 1	仅启用整型寄存器
r	转储所有整型寄存器
r eax, edx	仅转储 eax 和 edx
r eax=5, edx=6	赋给 eax 和 edx 新值
r eax:1ub	仅转储 eax 的第一个字节
rm 0x20	启用调试寄存器掩码
r	转储调试寄存器
rF	转储所有浮点寄存器
rM 0x4	转储所有浮点寄存器
rm 0x4; r	转储所有浮点寄存器

<<返回顶部

17) 变量信息

命令	变型/参数	描述
dt	dt -h	简要帮助
	dt [模块!]**名称**	转储变量信息
	dt [模块!]名称 **字段** [字段]	仅转储“字段名”（结构或集合）
	dt [模块!]名称 [字段] **地址**	被转储的结构的地址
	dt [模块!]名称*	列出符号（通配符）
	dt [**-n|y**] [模块!]名称 [**-n|y**] [字段] [地址]	-n 名称 = 参数是一个名称（当名称肯能背误认为地址时使用） -y 名称 = 部分匹配，而不是默认的精确匹配

	dt [-n|y] [模块!]名称 [-n|y] [字段] [地址] -**abcehioprsv**	**-a** = 换行显示带索引的数组元素 -b = 仅转储结构中相邻的数据块 -c = 压缩输出（一行显示所有字段） -i = 不缩进子类型 -l 列表字段 = 列表中指向下一个元素的字段指针 -o = 省略偏移量（结构字段） -p = 从屋里地址转储 **-r[l]** = 递归转储子类型/字段（最多到 l 级） -s [大小] = 仅为计数，统计给出大小的类型个数。 **-v** = 详细输出。
dv	dv dv **样式** dv [**/i /t /V**] [样式] dv [/i /t /V **/a /n /z**] [样式]	显示本地变量与参数 变量匹配样式 i = 类型（本地，全局，参数），t = 数据类型，V = 内存地址或寄存器位置 a = 地址种类，n = 名称种类，z = 尺寸种类

扩展

dt ntdll!_PEB*	列出所有包含有_PEB 的变量
dt ntdll!_PEB* -v	以详细输出方式列出（包含地址与大小）
dt ntdll!_PEB* -v -s 9	仅列出大小为 9 字节的符号
dt ntdll!_PEB	转储_PEB 信息
dt ntdll!_PEB @$peb	为我们的进程转储_PEB
dt ntdll!_PEB 7efde000	在地址 7efde000 处转储_PEB 你可以用"r @$peb"或"!peb"命令获取我们的进程 PEB 地址
dt ntdll!_PEB Ldr 会话 Id	仅转储 PEB 的 Ldr 和会话 Id 字段
dt ntdll!_PEB Ldr -y OS*	转储 Ldr 字段+所有以 OS 开头的字段
dt 模块!变量 m_cs.	转储 m_cs 并扩其子字段
dt 模块!变量 m_cs..	扩展其子字段至 2 级
dt ntdll!_PEB -r2	递归转储（2 级）
dv /t /i /V	转储本地变量的类型信息（/t）、地址与 EBP 偏移量（/V）并分类（/i） 注意：dv 也会为一个名为“本体调用协议”的方法显示一个本体指针的值 缺陷：你必须先执行些别的命令，dv 才能显示正确的值 处于函数入口点时本体指针就在当前的 ECX 中，你可以轻易的从那里获取它

<<返回顶部

18) 内存		
命令	**变型/参数**	**描述**
d*	d[a| u| b| w| W| d| c| q| f| D] [/c #] [地址]	**显示内存**[#要现实的列] a = ASCII 字符 u = Unicode 字符 b = 字节+ASCII

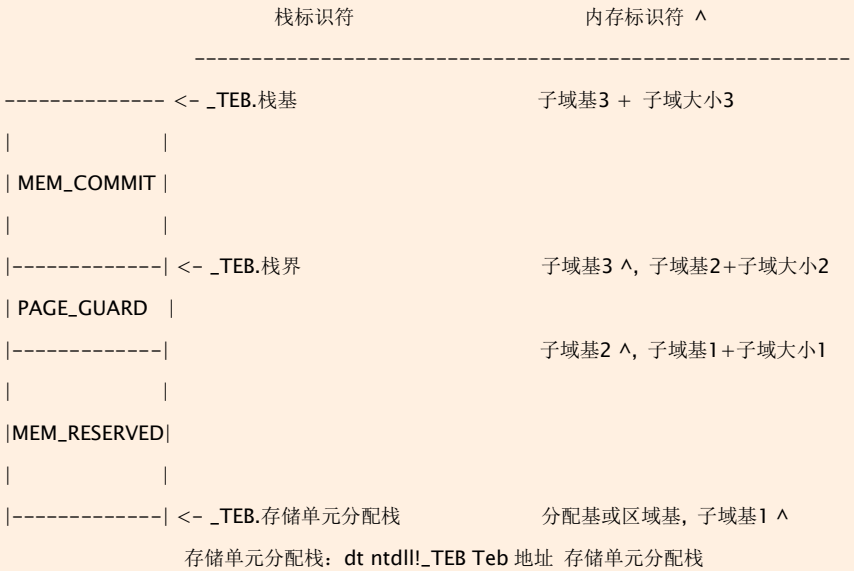
		w = 字（2 字节） W = 字（2 字节）+ASCII d = 双字（4 字节） c = 双字（4 字节）+ASCII q = 四字（8 字节） f = 浮点（单精度——4 字节） D = 浮点（双精度——8 字节）
	dy[b | d] ...	b = 二进制+字节 d = 二进制+双字
e*	e[b | w | d | q | f | D] 地址 值	**编辑内存** b = 字节 w = 字（2 字节） d = 双字（4 字节） q = 四字（8 字节） f = 浮点（单精度——4 字节） D = 浮点（双精度——8 字节）
	e[a | u | za | zu] 地址 "字符串"	a = ASCII 字符串 za = ASCII 字符串（NULL 结尾） u = Unicode 字符串 zu = Unicode 字符串（NULL 结尾）
ds, dS	ds *[/c #] [地址]* dS *[/c #] [地址]*	转储字符串结构（结构！不是无间隔的字符序列） s = 字符串或 ASCII 字符串 S = Unicode 字符串
d*s	dds [/c #] [地址] dqs [/c #] [地址]	**显示字和符号**（地址处的内存被认定为符号表中的一系列地址） dds = 双字（4 字节） dqs = 四字（8 字节）
dd*, dq*, dp*	dd* dq* dp* d*a d*u d*p	**显示引用内存** = 显示指定地址处的指针，解引用它并显示多种格式结果位中的内存 第二个字符决定使用的指针大小 dd* -> 使用 32 位指针 dq* -> 使用 64 位指针 dp* -> 标准大小：32 位或 64 为，取决于 CPU 的结构 第三个字符决定如何显示解引用内存 d*a -> 以 ASCII 字符形式显示解引用内存 d*u -> 以 Unicode 字符形式显示解引用内存 d*p -> 双字或四字显示解引用内存，取决于 CPU 结构 如果值与已知符号匹配，则同时显示对应符号
dl	dl[b] 地址 最大计数 大小	**显示链接单**（LIST_ENTRY 或 SINGLE_LIST_ENTRY） b = 倒叙转储（按照 BLinks 而非 FLinks） 地址 = 列表的起始地址 最大计数 = 转储最多#元素 大小 = 每个元素的大小 使用!list 为列表中的每个元素执行命令

!address	!address -? !address 地址 !address -summary !address -RegionUsageXXX	显示关于目标进程的内存信息 简要帮助 转储地址区域的信息 转储进程的摘要信息 转储指定区域（RegionUsageStack, RegionUsagePageHeap, ...）
!vprot	!vprot -? !vprot 地址	简要帮助 转储虚拟内存保护信息
!mapped_file	!mapped_file -? !mapped_file 地址	简要帮助 转储包含指定地址的文件名

扩展

dd 0046c6b0	显示 0046c6b0 处的双字
dd 0046c6b0 L1	显示 0046c6b0 处的 1 个双字
dd 0046c6b0 L3	显示 0046c6b0 处的 3 个双字
du 0046c6b0	显示 0046c6b0 处的 Unicode 字符
du 0046c6b0 L5	显示 0046c6b0 处的 5 个 Unicode 字符
dds esp == kd	显示栈上的字与符号
!mapped_file 00400000	转储包含地址 00400000 的文件名
!address	显示我们进程的所有内存区域
!address -RegionUsageStack	显示我们进程的所有栈区域
!address esp	显示我们线程栈的固有子区域信息 注意：栈溢出 SubRegionSize（内存大小已分配）将会很大，即： AllocBase : SubRegionBase - **SubRegionSize** ---------- 001e0000 : 002d6000 - 0000a000

线程的确定栈的使用



出自 MSDN CreateThread > dwStackSize > "Thread Stack Size"：

“每一个新线程接收它的栈空间都要组成固有和预留两个内存。默认状态下，每个线程使用 1MB 的预留内存和一页的固有内存。当需要时，程序会从预留内存中划分出一页的程序块。”

<<返回顶部

19) 操作内存范围

命令	变型/参数	描述
c	c 范围 目标地址	比较内存
m	m 范围 目标地址	移动内存
f	f 范围 样式	填充内存。样式 = 一串字节（数字或 ASCII 字符）
s	*s 范围 样式* s -[标识]b *范围 样式* s -[标识]w *范围 '样式'* s -[标识]d *范围 '样式'* s -[标识]q *范围 '样式'* s -[标识]a *范围 "样式"* s -[标识]u *范围 "样式"* s -[标识,l 长度]sa *范围* s -[标识,l 长度]su *范围* s -[标识]v *范围 对象*	搜索内存 b = 字节（默认值） 样式 = 一串字节（数字或 ASCII 字符） w = 字（2 字节） d = 双字（4 字节） q = 四字（8 字节） 样式 = 括在单引号中（例如'Tag7'） a = ASCII 字符串（不能以 NULL 结尾） u = Unicode 字符串（不能以 NULL 结尾） 样式 = 括在双引号中（例如"This string"） 搜索搜有包含有可显示 ASCII 字符串的内存 搜索搜有包含有可显示 Unicode 字符串的内存 长度 = 这些字符串的最小长度；默认是 3 个字符 搜索同样类型的对象 对象 = 一个对象指针的地址或对象本身 标识 ---- w = 仅搜索可写内存 l = 仅输出匹配的搜索地址（使用.foreach 时会很有用） 标识必须用方括号括起来并且无空格 例如：**s -[swl 10]Type Range Pattern**
.holdmem	.holdmem -a 范围 .holdmem -o .holdmem -c 范围 .holdmem -D .holdmem -d { 范围 | 地址 }	**保存并比较内存。**比较方式为字节对字节 存储的内存范围 显示所有已存储的内存范围 对所有已存储的内存范围进行范围比较 删除所有以存储的内存范围 删除指定的内存范围（一切存储的范文均包含范围的地址或重叠）

扩展

c 地址 (地址+100) 目标地址	比较目标地址处 100 字节的地址
c 地址 L100 目标地址	-||-
m 地址 L20 目标地址	从地址处移动 20 字节至目标地址处
f 地址 L20 'A' 'B' 'C'	按照"ABC"的形式填充指定内存位置，反复数次
f 地址 L20 41 42 43	-||-
s 0012ff40 L20 'H' 'e' 'l' 'l' 'o'	通过 0012FF5F 搜索内存位置 0012FF40 处的"Hello"样式
s 0012ff40 L20 48 65 6c 6c 6f	-||-
s -a 0012ff40 L20 "Hello"	-||-
s -[w]a 0012ff40 L20 "Hello"	仅搜索可写内存

<<返回顶部

20) 内存：堆

命令	变型/参数	描述
!heap	!heap **-?**	简要介绍
	!heap	列出带索引和 HeapAddr 的堆
	!heap **-h**	列出带**索引和范围**（=起始地址（=HeapAddr），结束地址）的堆
	!heap -h [HeapAddr | 索引 | 0]	详细的堆信息 [索引 = 堆索引，0 = 所有堆]
	!heap -v [HeapAddr | 索引 | 0]	验证堆 [索引 = 堆索引，0 = 所有堆]
	!heap **-s** [HeapAddr | 0]	**概要信息，即预留和固有内存** [索引 = 堆索引，0 = 所有堆]
	!heap -i [HeapAddr]	给出地址处程序块的详细信息
	!heap -x [-v] 地址	搜索含有地址的堆块（v = 搜索整个进程虚拟空间）
	!heap -l	搜索潜在的泄露堆块
!heap -b, -B	!heap 堆 -b [alloc | realloc | free] [Tag]	在堆管理器中设置条件断点[堆 = HeapAddr | 索引 | 0]
	!heap 堆 -B [alloc | realloc | free]	移除一个条件断点
!heap -flt	!heap -flt s 大小	转储与指定大小匹配的分配信息
	!heap -flt r 最小 最大	范围过滤器
!heap -stat	!heap **-stat**	转储 HeapHandle **列表**
	!heap **-stat -h** [HeapHandle | 0]	转储分配大小的使用率统计[HeapHandle = 给出的堆 | 0 = 所有堆]。 统计包括每个分配大小的分配空间、程序块号、总计内存
!heap -p	!heap -p -?	扩展页堆帮助
	!heap **-p**	NtGlobalFlag 的概要，HeapHandle+标准堆列表**
	!heap -p -h HeapHandle	带句柄的页堆详细信息
	!heap **-p -a UserAddr**	包含 UserAddr 的堆分配详情。可用时显示回溯。
	!heap **-p -all**	进程中所有堆的所有分配详情 输出包括每个分配调用的 UserAddr 和分配大小。

看起来下列情况适用于 Windows XP SP2

a) 常规堆

1. CreateHeap -> 创建一个_HEAP

2. AllocHeap -> 创建一个_HEAP_ENTRY

b) 激活页堆（gflags.exe /i <IMAGE.EXE> +hpa）

1. CreateHeap -> 创建一个_DPH_HEAP_ROOT (+ _HEAP + 2x _HEAP_ENTRY)**

2. AllocHeap -> 创建一个_DPH_HEAP_BLOCK

**随着页堆的激活，每次调用 CreateHeap 仍需一个带有两_HEAP_ENTRY 常数的_HEAP

项	描述	堆类型
HeapHandle	= **HeapCreate** 或 **GetProcessHeap** 的返回值 对于常规堆：HeapHandle == 堆起始地址	常规 & 页
HeapAddr	= 起始地址 = 常规堆	常规 & 页
UserAddr，UserPtr	= 范围中的值 [**HeapAlloc...**HeapAlloc+AllocSize] 对于常规堆，这个范围在堆中更深入[起始地址-结束地址]	常规 & 页
UserSize	= 分配大小（值传递给 HeapAlloc）	常规 & 页
_HEAP	= HeapHandle = 堆起始地址 对于每一个 **HeapCreate**，都将建立一个_HEAP 结构 你可以使用“!heap -p -all”获取这些地址	常规堆
_HEAP_ENTRY	对于每一个 **HeapAlloc**，都将建立一个_HEAP_ENTRY 你可以使用“!heap -p -all”获取这些地址	常规堆
_DPH_HEAP_ROOT	= 通常为 HeapHandle + 0x1000 对于每一个 **HeapCreate**，都将建立一个_DPH_HEAP_ROOT 你可以使用“!heap -p -all”获取这些地址	页堆
_DPH_HEAP_BLOCK	对于每一个 **HeapAlloc**，都将建立一个_DPH_HEAP_BLOCK 你可以使用“!heap -p -all”获取这些地址	页堆

扩展

dt ntdll!_HEAP	转储_HEAP 结构
dt ntdll!_DPH_HEAP_ROOT	转储_DPH_HEAP_ROOT 结构 启用页堆。而后你可以使用“!heap -p -all”来获取你的进程中实际_DPH_HEAP_ROOT 结构的地址。
dt ntdll!_DPH_HEAP_BLOCK	转储_DPH_HEAP_BLOCK 结构 启用页堆。而后你可以使用“!heap -p -all”来获取你的进程中实际_DPH_HEAP_BLOCK 结构的地址 。
!heap	带索引和 HeapAddr 列出所有堆
!heap -h	带范围信息（起始地址，结束地址）列出所有堆
!heap -h 1	索引 1 堆的详细堆信息
!heap -s 0	所有堆的概要（预留和固有内存，……）
!heap -flt s 20	转储大小为 20 字节的堆分配

!heap -stat	转储 HeapHandle 列表。HeapHandle = HeapCreate 或 GetProcessHeap 的返回值
!heap -stat -h 00150000	转储 HeapHandle = 00150000 的使用率统计
!heap 2 -b alloc mtag	索引 2 堆中目标为 mtag 的 HeapAlloc 调用处的断点
!heap -p	转储堆句柄列表
!heap -p -a 014c6fb0	包含地址 014c6fb0 的堆分配详情 + 可用时调用栈
!heap -p -all	转储进程中所有堆的所有分配详情

谁分配内存——谁调用 HeapAlloc?

1.在 GFlags 中为你的映像选择“创建用户模式栈跟踪数据库”（gflags.exe /i < APPNAME.EXE > +ust）

2.使用 WinDbg 命令行执行**!heap -p -a <用户地址>**，<用户地址>就是你分配的地址***。

3.当!heap -p -a <用户地址>转储一个调用堆时，没有包含源码信息。

4.获取源码信息你必须在第一步中启用页堆。

5.执行 **dt ntdll!_DPH_HEAP_BLOCK StackTrace <我的堆块地址>**，<我的堆块地址>就是第三步中 DPH_HEAP_BLOCK 检索到的地址。

6.执行 **dds <StackTrace>**，<StackTrace>就是第五步中检索到的值。

注意：dds 将转储包含源码信息的栈。

谁创建堆——谁调用 HeapCreate?

1.在 GFlags 中为你的映像选择“创建用户模式栈跟踪数据库”和“启用页堆”（gflags.exe /i < APPNAME.EXE > +ust +hpa）

2.

a) 使用 WinDbg 命令行执行**!heap -p -h <堆句柄>**，<堆句柄>就是 HeapCreate 的返回值。你可以执行**!heap -stat** 或**!heap -p** 来获取你的进程的堆句柄。

b) 或者你可以使用**!heap -p -all** 直接获取你的进程的所有_DPH_HEAP_ROOT 的地址。

3.执行 **dt ntdll!_DPH_HEAP_ROOT CreateStackTrace <我的堆跟地址>**，<我的堆跟地址>就是第二步中_DPH_HEAP_ROOT 检索到的地址。

4.执行 **dds <创建栈跟踪>**，<创建栈跟踪>就是第三步中检索到的值。

寻找内存泄露

使用 WinDbg 命令行执行**!address -summary**.

若 **RegionUsageHeap** 或 **RegionUsagePageHeap** 扩增，那么你可能有一个堆上的内存泄露。而后继续以下步骤。

1.在 GFlags 中为你的映像选择“创建用户模式栈跟踪数据库”（gflags.exe /i < APPNAME.EXE > +ust）

2.使用 WinDbg 命令行执行**!heap -stat**，获取所有激活的堆块和它们的句柄。

3.执行**!heap -stat -h 0**.这将为没一个 AllocSize 列出指定分配统计句柄。每一个 AllocSize 都将列出：分配大小、程序块号和内存总计。以最大内存总计获取分配大小。

4.执行**!heap -flt s <大小>**。<大小> = 上一步中我们决定的分配大小。此命令将以特定大小列出所有程序块。

5.执行**!heap -p -a <用户地址>**来从你分配的字节中获取栈跟踪。使用第四步中你获取的<用户地址>。

6.获取源码信息你必须在第一步中启用页堆（gflags.exe /i < APPNAME.EXE > +ust +hpa）

7.执行 **dt ntdll!_DPH_HEAP_ROOT StackTrace <我的堆跟地址>**，<我的堆跟地址>就是第五步中 DPH_HEAP_ROOT 检索到的地址。

8.执行 **dds <StackTrace>**，<StackTrace>就是第七步中检索到的值。

注意：dds 将转储包含源码信息的栈。

***什么是<用户地址>?**

1.<用户地址>一般是 HeapAlloc 的返回地址：

```
int AllocSyze = 0x100000; // == 1 MB
BYTE* pUserAddr = (BYTE*) HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, AllocSyze);
```

2.通常范围内[UserAddr....UserAddr+AlloSize]的一切地址都是有效参数：

!heap -p -a [**UserAddr....UserAddr+AlloSize**]

<<返回顶部

21) 应用程序验证工具

应用程序验证工具概述并跟踪微软 Win32 下的 API（堆、句柄、锁、线程、DLL 加载/卸载等）、例外、内核对象、寄存器、文件系统。我们要是用!avrf 扩展获取跟踪信息。

命令	变型/参数	描述
!avrf		显示应用程序验证工具选项。若一个应用程序验证工具停止，则显示停止类型和原因
	!avrf -?	简要帮助
	!avrf -vs *N*	从存放日志中转储最后 N 个入口（MapViewOfFile, UnmapViewOfFile, ...）。
	!avrf -vs -*a* *地址*	在存放日志中搜索地址
	!avrf -hp *N*	堆分配、堆释放、新建并删除日志
	!avrf -hp -*a* *地址*	搜索堆日志中的地址
	!avrf -cs *N*	DeleteCriticalSection API 日志（最后#个入口）。~CCriticalSection 隐藏调用。
	!avrf -cs -*a* *地址*	在临界区删除日志中搜索地址
	!avrf -dlls *N*	加载库/释放库日志
	!avrf -ex *N*	例外日志
	!avrf -cnt	全局计数器（WaitForSingleObject, HeapAllocation calls, ...）
	!avrf -threads	线程信息 + 子线程启动参数
	!avrf -trm	TerminateThread API 日志
	!avrf -trace *索引*	转储带索引的栈跟踪
	!avrf -brk *[索引]*	转储或设置/复位中断触发器

<<返回顶部

22) 记录扩展（logexts.dll）

你必须在 GFlags 中为你的映像启用下列选项

-> “创建用户模式的栈跟踪数据库”

-> “栈回溯：（兆）” -> 10

-> 看起来有时候你需要在 GFlags 中设置检查和指定“调试器”的域

命令	变型/参数	描述
!logexts.help		显示所有 Logexts.dll 扩展命令
!loge	!loge *[显示文件列表]*	启用日志+在未初始化时尽可能初始化。输出目录选项。
!logi		初始化（=将记录器注入到目标程序中）但不启用日志
!logd		禁用日志
!logo	!logo !logo [e|d] [d|t|v]	列出输出设置 启用/禁用[d-调试器；t-文本文件；v-详细日志]输出。使用 logviewer.exe 查阅详细日志
!logc	!logc !logc p # !logc [e|d] *	列出所有类别 按类别号列出 API 启用/禁用所有类别

	!logc [e|d] # [#] [#]	启用/禁用类别号
!logb	!logb p !logb f	显示调试器缓冲区 刷新缓冲区日志文件
!logm	!logm !logm [i|x] [DLL] [DLL]	显示模块的包含/排除列表 指定模块的包含/排除列表

扩展

启用 19-ProcessesAndThreads 和 22-StringManipulation 日志

!loge	启用日志
!logc d *	启用所有类别
!logc p 19	显示类别19的API
logc e 19 22	启用类别 19 和 22
!logo d v	禁用详细输出
!logo d t	禁用文本输出
!logo e d	启用调试器输出

<<返回顶部