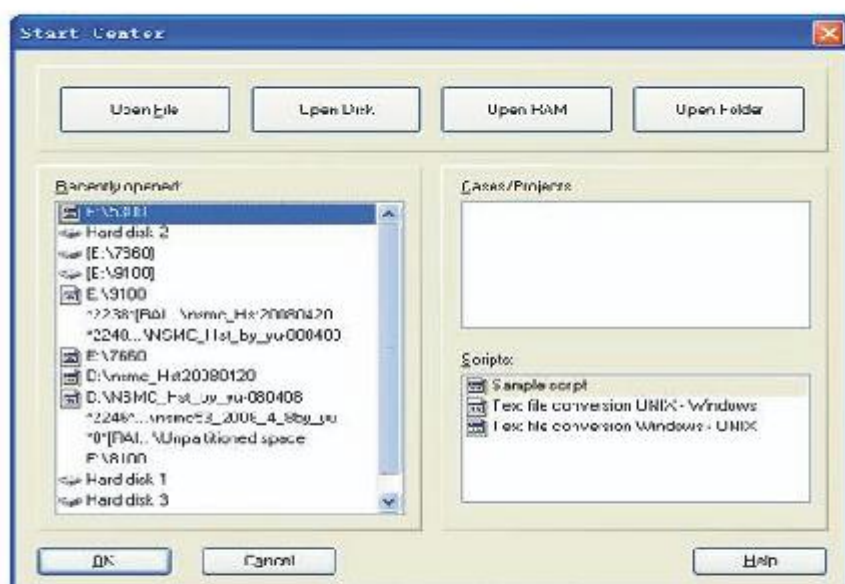
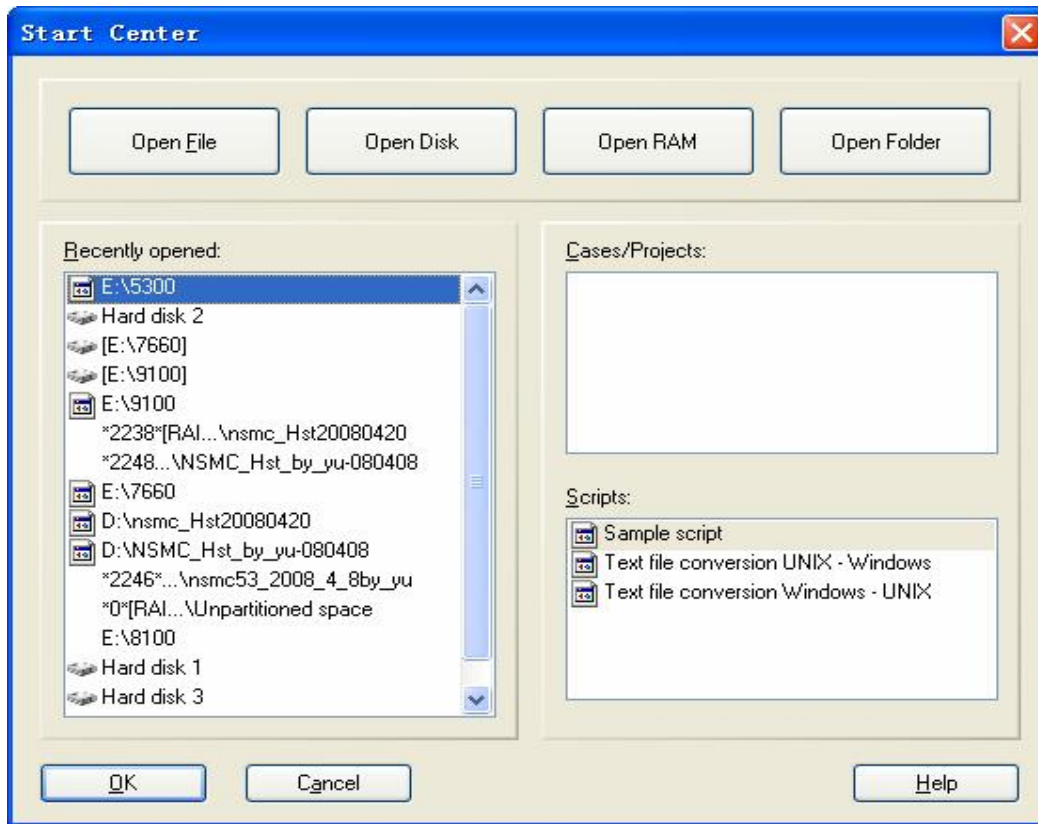


winhex教程



本教程是作者困惑的浪漫(高志鹏)耗费大量心血原创，拥有完全的著作权和知识产权，

整理者:徐旺森 qq:879520041

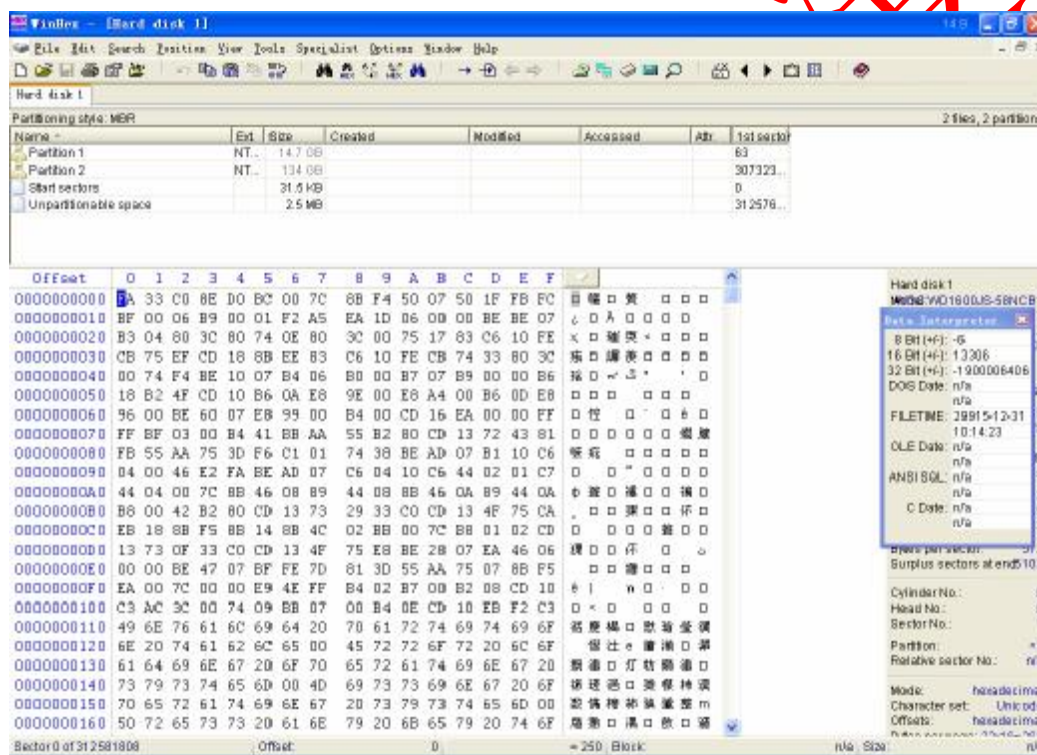


本教程是作者困惑的浪漫(高志鹏)耗费大量心血原创，拥有完全的著作权和知识产权，任何人未经同意不得擅自转载、抄录、或用于营利目的。

QQ: 409308150

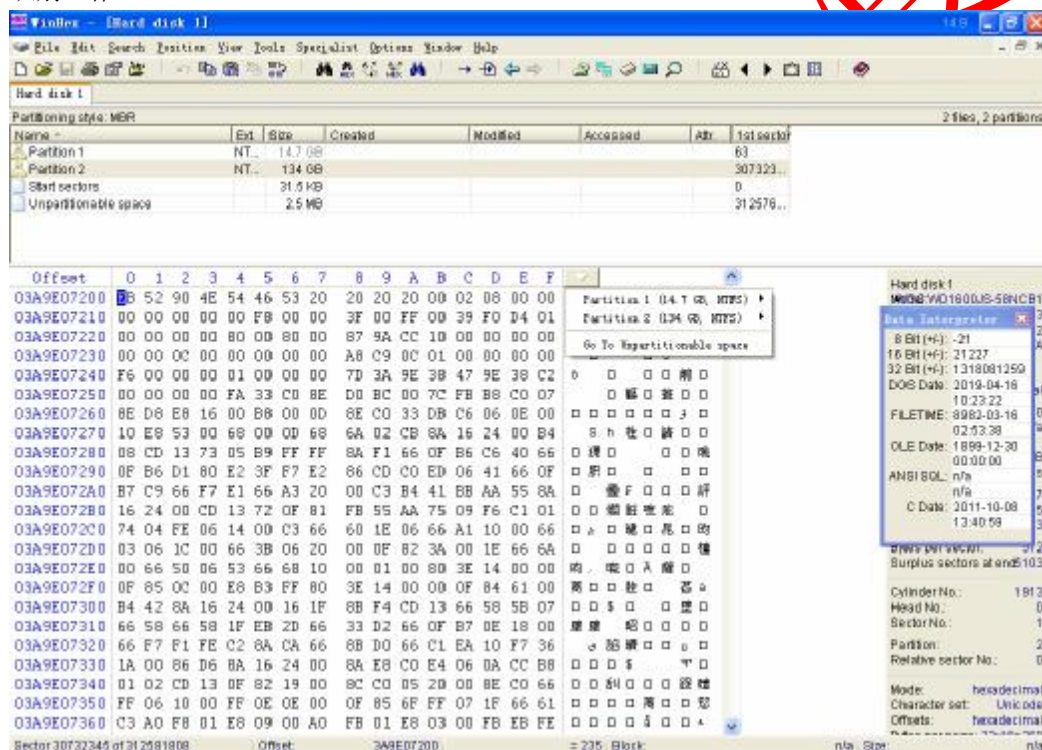
gzp001015@163.com

WINHEX 启动中心已经可以大致反映出它的强大功能，正上方有 Open File、Open Disk、Open RAM、Open Folder 四个控件，可以对文件、磁盘、内存、文件夹进行十六进制或文字编码的编辑工作。WINHEX 支持除加密文件外的所有已知文件格式，可以静态地编辑文件中的任意字节，改变文件结构，跟踪文件各部分在处理器中的协调过程和状态，或是对文件内容进行不可逆转的擦除工作。WINHEX 支持除非标准硬盘（每扇区不是 512 字节的特殊硬盘，一般为光纤接口）外的所有介质，可以静态地改变磁盘数据的排列、状态、属性等。可以改变除 ECC 区外任意扇区任意字节的内容。WINHEX 还提供高访问级别的内存编辑功能，可以对当前操作系统进程进行在线、动态地编辑工作，可以更改内存变量在磁盘保留区的映射值。WINHEX 可以对文件夹内部所有指定文件进行动态、同步的比较和集体修改工作。从 Recently opened 中可以看出 WINHEX 对以往操作工程的智能记忆，Case/Projects 则为用户有选择地保留操作成果提供便利条件。Scripts 是一个类似于流程序模型的批处理脚本编辑系统，可以调用 WINHEX 已经开发并集成的各种函数指令进行编程工作（已经有三个实验脚本），指令将不折不扣地按先后顺序依次执行，以后篇章中我们将详细介绍。



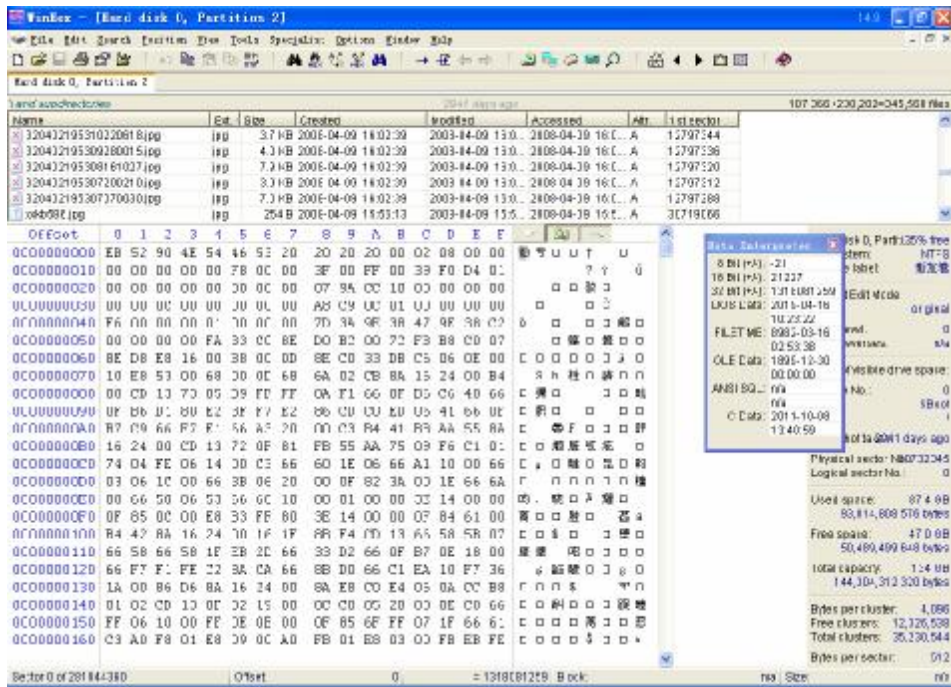
选择任意硬盘双击展开，我们就可以看到 WINHEX 的主编辑窗口，最上方的 WINDOWS 窗体显示当前任务对象为 Hard disk 1，接下来是菜单栏，集成了文件操作、编辑、搜索查找、方位、视图、工具、特殊工具、设置、窗口和帮助等十大主菜单。菜单栏下方为工具栏，提供了 WINHEX 操作中使用率很高的各种功能键，依次是创建新文件、打开文件、保存文件、打印、文件属性、打开文件夹、撤消操作、拷贝扇区、剪切板操作、数据转换、修改数据、查找文本、查找 16 进制编码、替换文本、替换 16 进制编码、同步查找、转到偏移量、转到扇区、光标向回移动、光标向前移动、打开磁盘、复制磁盘、编辑内存、调用计算器、数据图像化分析、HEX 区减少一行、HEX 区增加一行、文件系统快照和目录浏览器选项、帮助调用等。再下方是目录浏览器视图，任务对象、文件筛选结果等都在这里体现。整个左下角的特殊区域都是最直接的操作区（编辑区），直观地反映着磁盘或文件中的 16 进制编码状况，

通常情况下由 HEX 视图和文本视图构成。坐标中的 X 轴表示十六进制高低位，Y 轴表示连续的偏移量行，两轴的任意值相交即可完成一次偏移量的定位，用户可以在此区域直接完成字节的各种操作修改任务。最下方一行是扇区定位及状态模块，用户可以轻松完成范围内任意扇区的瞬间定位；偏移量定位及状态模块，操作同扇区定位相仿；16 进制转换模块，可以方便地显示 16 进制和 10 进制数值的转换结果，而且可以对位数和符号参与进行选择；选块定位及状态模块，可以对选块起始和结尾进行定义；最后是选块对象体积，单位是字节。整个右下方是操作对象状态属性框，可以反映当前操作对象的各种参数，便于用户进行判断。以硬盘为例，从上到下显示的内容有：操作对象名称、硬盘型号、硬盘序列号、固件版本号、接口总线类型、当前编辑模式、状态、撤销级别、修改状况、硬盘总容量、柱面数、磁头数、扇区数、每扇区字节数、结尾剩余扇区、当前柱面编号、当前磁头编号、当前扇区编号、当前分区号、当前分区相对扇区号、模式、字符集、偏移量、编码模式、字符集、偏移量进制、HEX 区域结构、当前窗口号、窗口数量、剪切板状态、临时文件夹状态。给界面的另一默认窗口是数据解释器，可以实现各种函数的翻译调用，大大方便了用户对特殊 HEX 区域的识别工作。



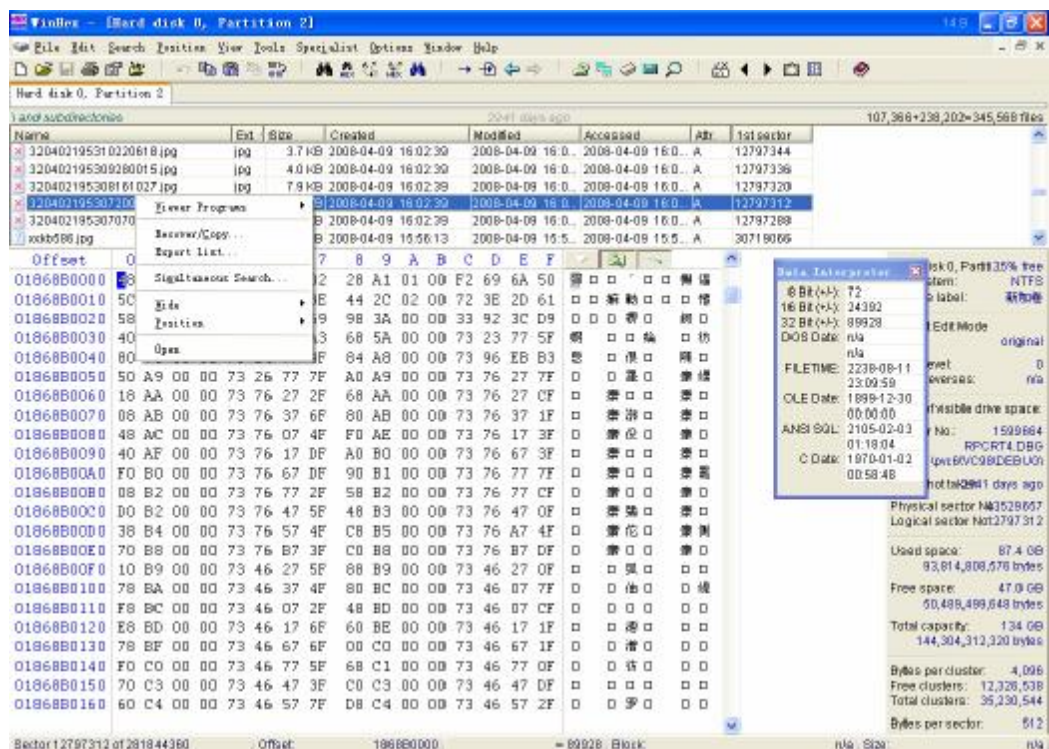
HEX 视图区与文本视图区交接处有一个分区快捷入口，可以在错误模式下展开各种分区，如分区被操作系统认为丢失时、引导扇区或文件系统结构轻微损坏时。



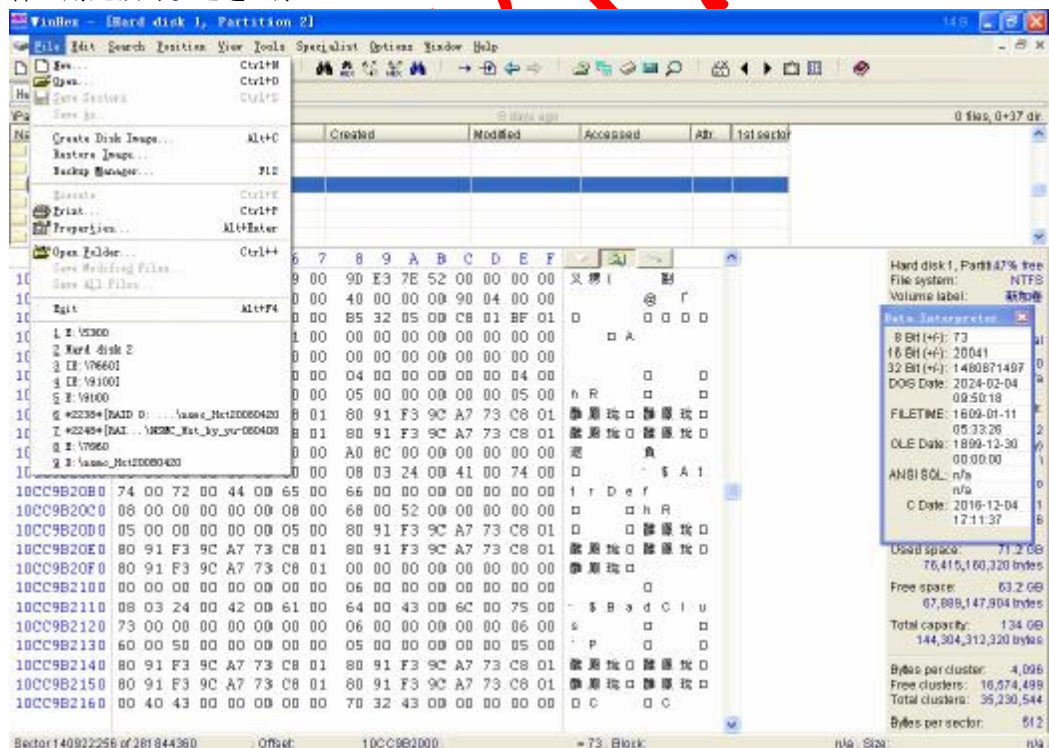


(分类浏览模式)

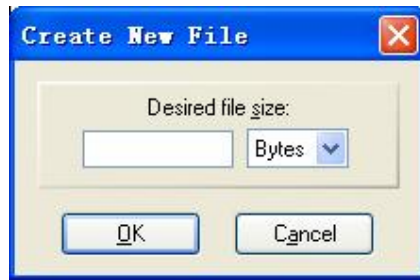
无论用何种方式进入分区，WINHEX 都会提醒你是否获取新的分区视图，如果选 OK 则调用上次的视图，这样新写入的数据将无法显现，故推荐每次进入时都使用 Take new one，使该分区视图保持最新状态。展开后我们可以看到类似于 WINDOWS 资源管理器的目录浏览结构，从左到右显示了该文件的各种属性状态，如扩展名、体积、创建时间，LCN 映射等。我们还可以进行分类浏览，按文件属性进行各种有利于用户操作的排列。在文件列表中我们可以点击鼠标右键进行各种基本操作，如文件提取、预览、隐藏等。文件列表上方一栏对文件数量进行了粗略统计。



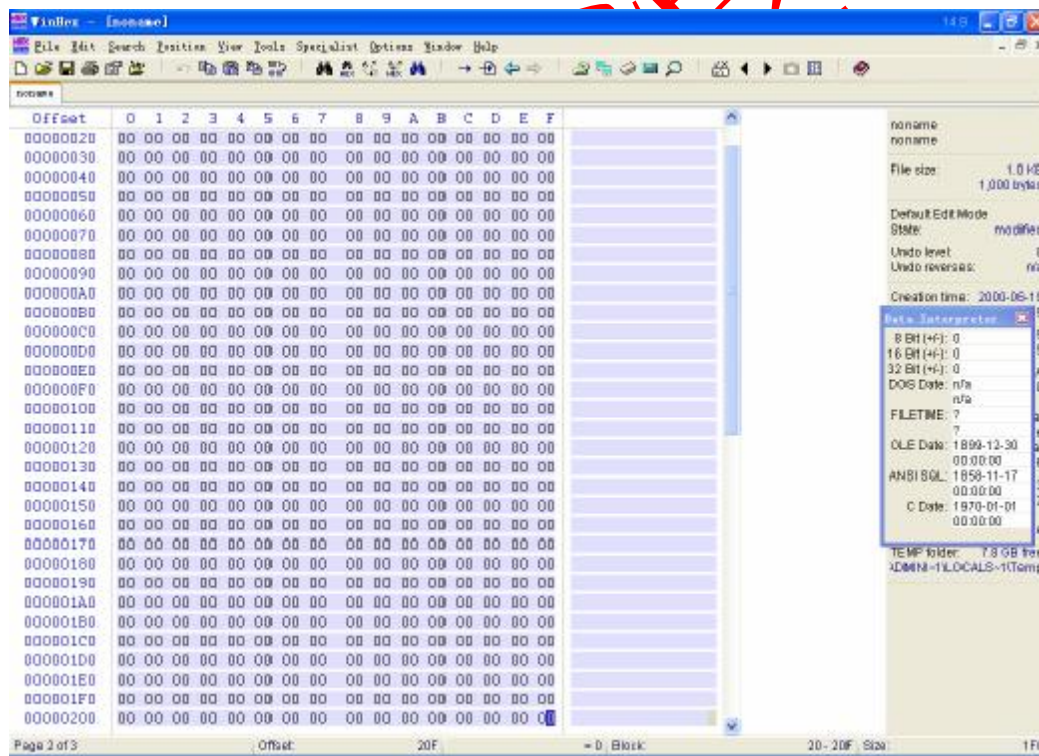
单击 Recover/Copy 就可以完成简单的数据恢复工作，文件图标为红色打叉就是被删除的文件，用此法可以迅速还原。



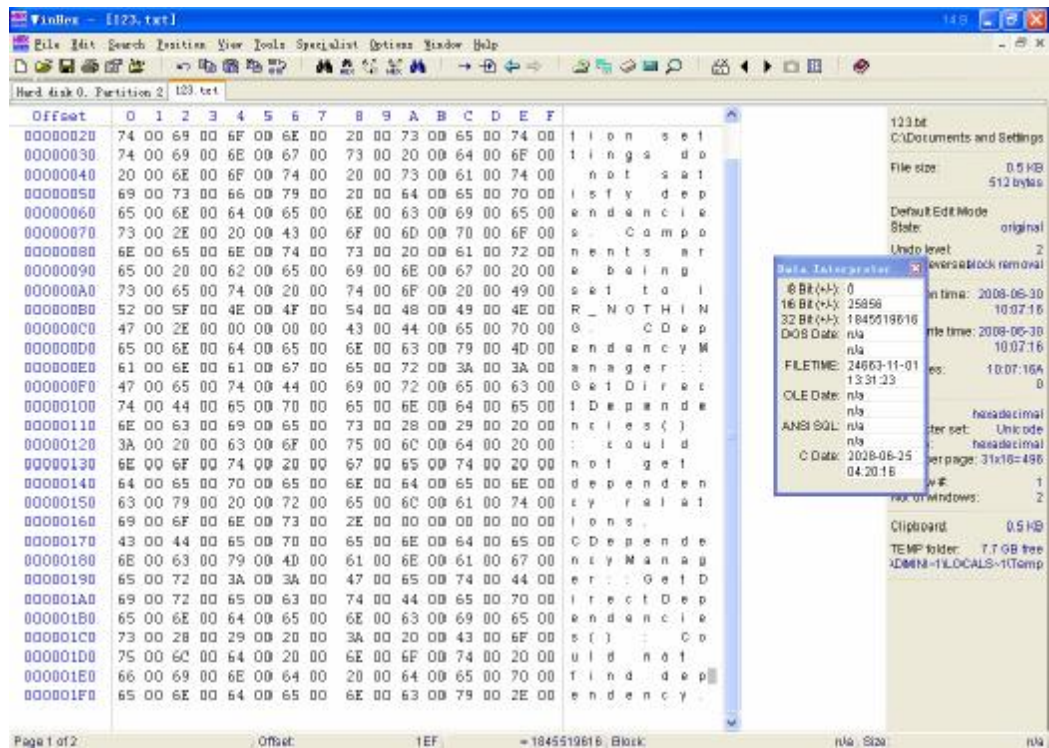
展开主菜单 File，我们可以看到各级子菜单，按上下顺序分别是：新建一个文件、打开一个文件、保存扇区改动、另存为、创建磁盘镜像或备份、还原磁盘镜像或备份、备份管理器、文件预览、打印、属性访问器、打开文件夹、保存改动过的文件、保存所有文件、退出和一些近期项目的快捷访问键。此菜单是 WINHEX 操作的基本菜单，非常重要。



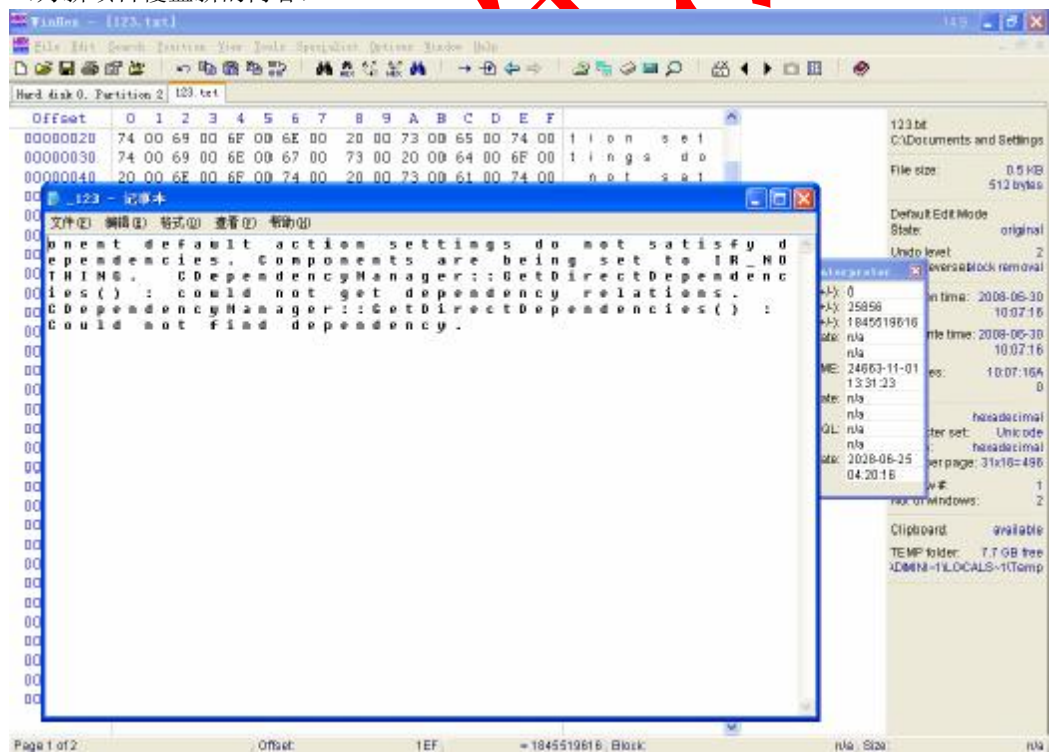
首先我们单击 New 键，出现 Create New File 对话框，提示输入要创建文件的大小，单位可以是字节、KB、MB、GB。如输入 496Bytes，点击 OK 就创建了一个以 noname 命名由 496 个零值字节构成的项目。此时我们就可以为这些零字节赋予有意义的值，如我们将某文本文件的 HEX 值覆盖到这些零字节中，然后另存为（Save As）123.txt。打开该文本文件，就可以看到我们写入的内容。也就是说理论上我们可以复制任意文件的内容到新建项目中，从而使新项目拥有了该文件的灵魂。熟悉汇编语言和文件编码的人甚至可以写文章一般创造任意文件结构。



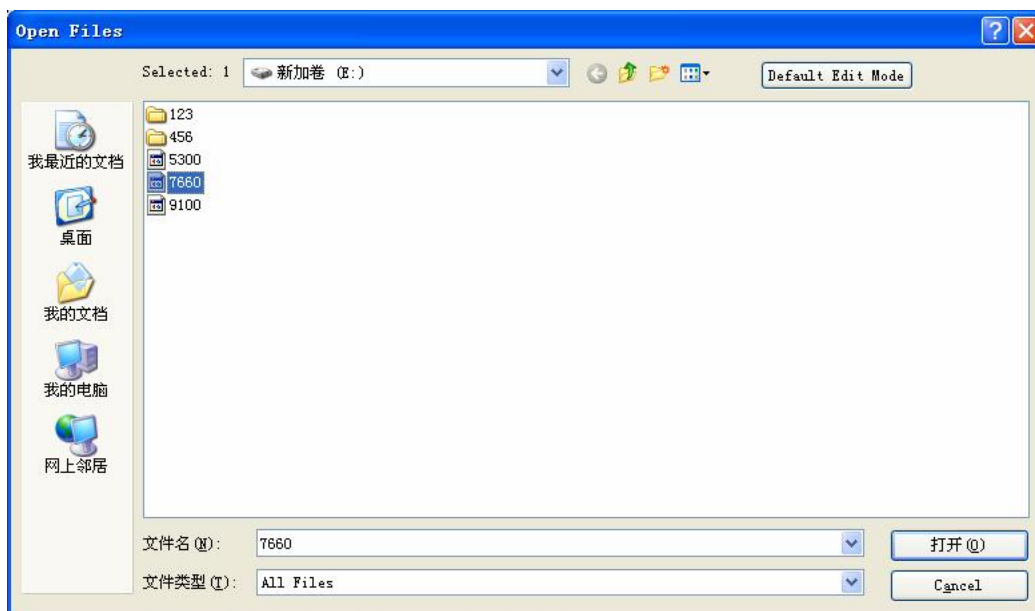
（创建一个新项目）



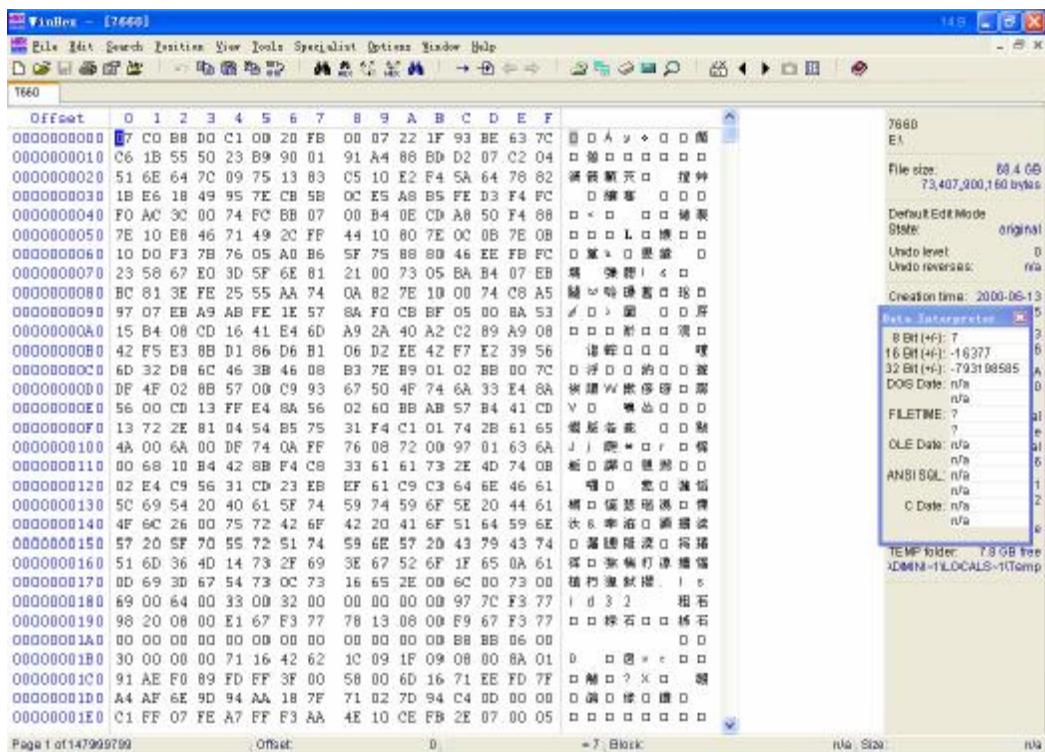
(为新项目覆盖新的内容)



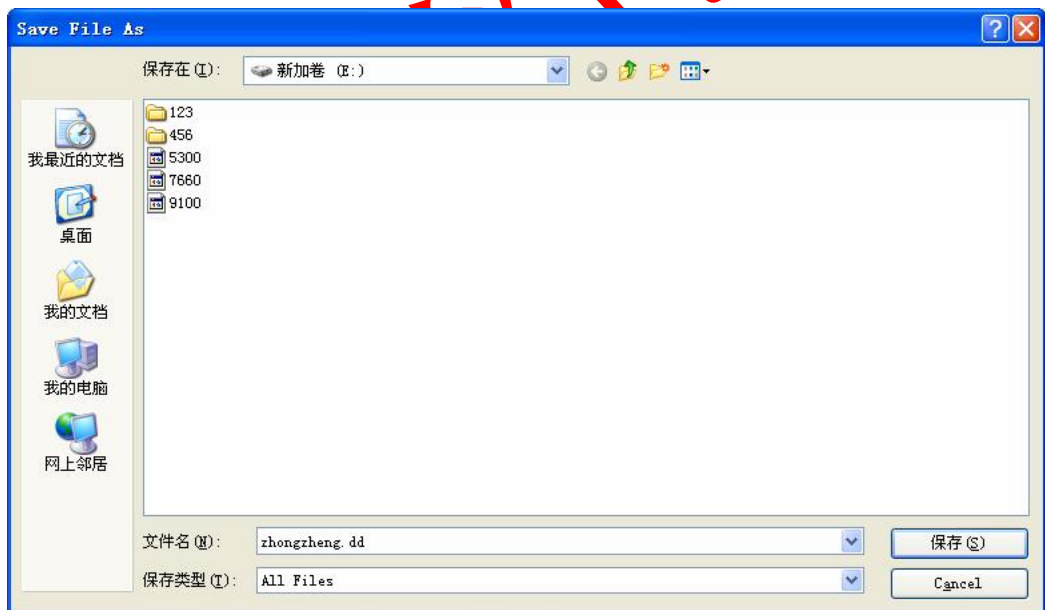
(新项目构建的新文件 123.txt)



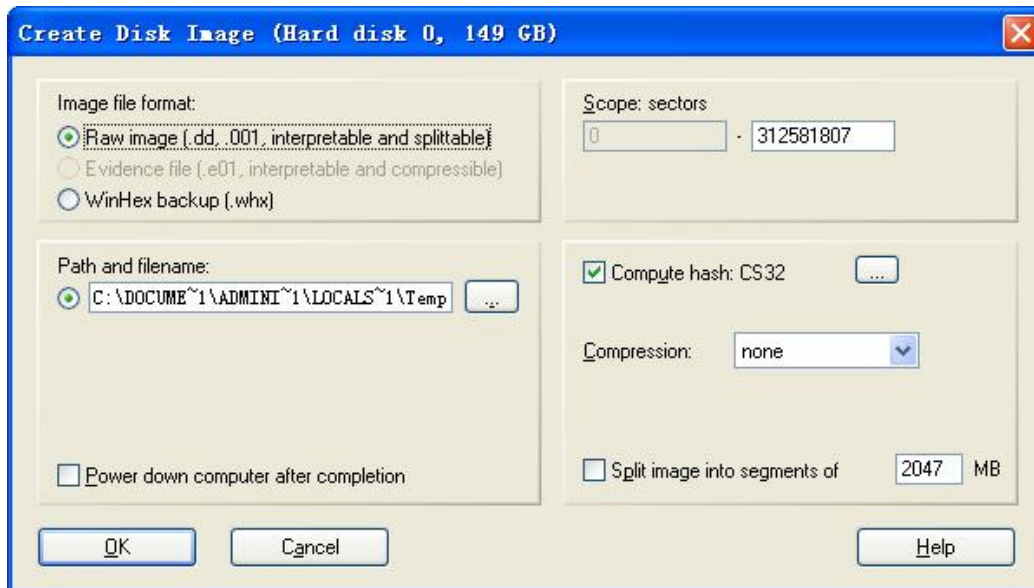
点击 **Open Files**，便可以浏览任意文件的 **HEX** 编码、字符串等甚至连磁盘镜像文件或部分加密文件都可以轻松解析出来。打开文件后我们就可以进行各种修改、裁剪、填补，销毁工作了。此时，主界面右边会显示出该文件的各种属性参数，如大小、创建时间等。注意，普通文件被打开后将不再按照扇区单位进行浏览，而是采用“页面”，我们可以看到原本扇区之间的分割线已经消失。单个页面没有固定大小，纯粹是显示单位。当然，如果遇到特殊情况如打开的是一个原始磁盘镜像文件，按页面浏览就会产生诸多不便，定位扇区，解释文件系统等工作将无法完成，这时我们就需要将此文件强制按照每 512 字节/扇区进行处理，**WINHEX** 介质管理器就会视此文件为一个标准磁盘，从而激活许多针对磁盘的特殊功能。后面我们会重点讲解。



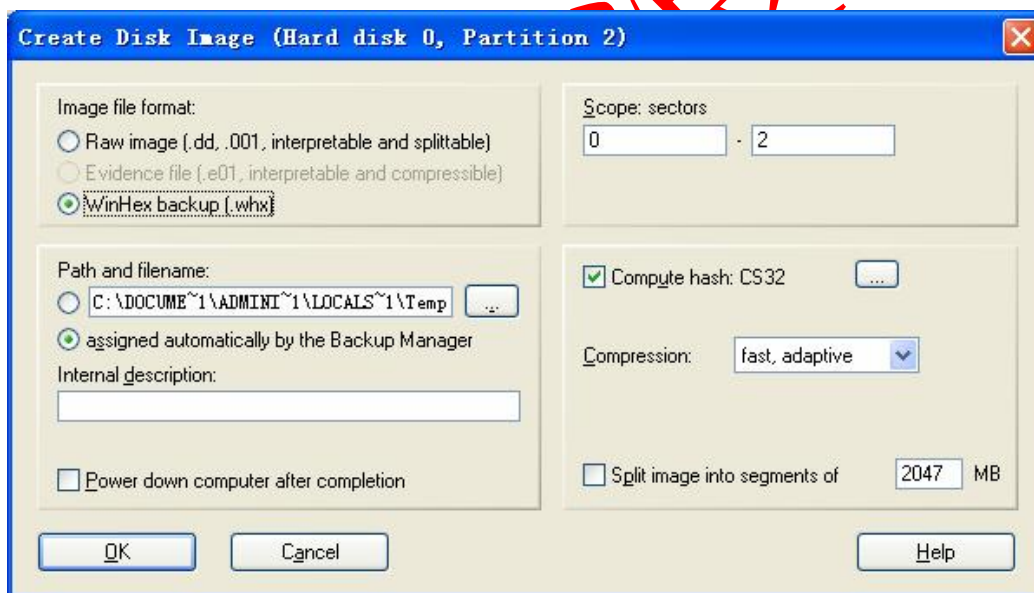
(打开了一个磁盘镜像文件)



如果要将修改、裁剪、填补，销毁等操作最终写入文件，就需要用到 Save、Save As 等功能键，如我们修改了某文件的文件头，直接退出是不行的系统就会提示是否保存修改等，这里的保存等同于菜单里的 Sava，我们还可以用“另存为”来更改文件扩展名等。注意如果修改对象过于巨大，保存工作可能会持续很长时间，甚至因为内存耗尽而中断。



(原始镜像方式)

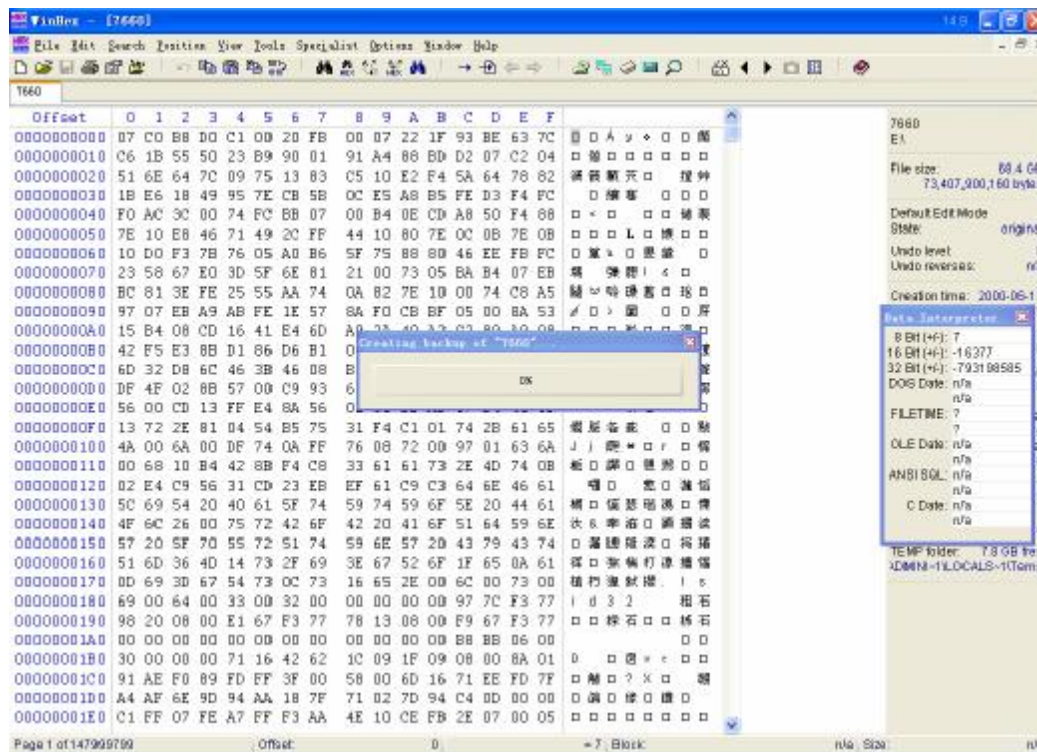


(WINHEX 备份)

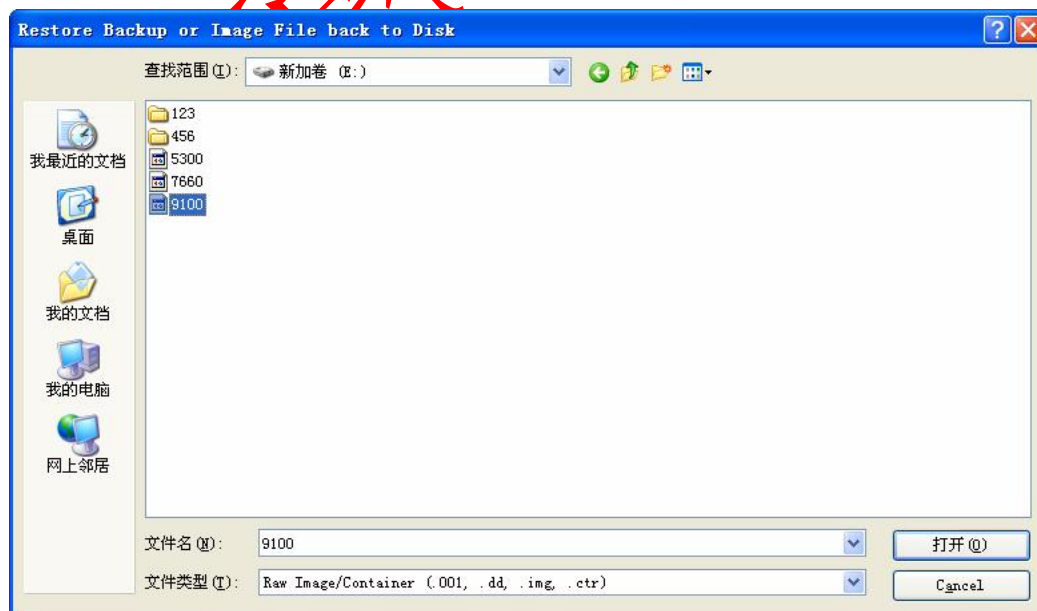
创建磁盘镜像是 WINHEX 最常用最重要的功能之一，被广泛应用于电子取证、磁盘克隆、数据备份领域。点击 Create Disk Image，出现基于目前操作对象信息的操作控制台，比如我们打开的是一个分区，而且显示在最前面的窗口正是此分区的内容，那么此时的操作对象就是该分区，我们创建的镜像也是与该分区相关的。

现在我们分析功能界面：Image Files format 代表了需要创建镜像的类型，分为原始镜像、证据镜像、和 WINHEX 备份 3 种类型。原始镜像也称一对一镜像，RAW 镜像，是指完全不考虑文件系统和未使用空间，按照扇区单位逐一复制而成的镜像。生成的镜像无论是体积还是数据分布都与其来源不会有一丝一毫差距，此种镜像使用最为广泛。证据镜像是指可压缩可解释可加密的特殊镜像方式，一般应用于保密程度很高的电子取证工作中，高密度的压缩却不会影响证据的原始性，令人称奇。严格的讲 WINHEX 备份已经不算是一种镜像方式，

顶多是一种类似于 GHOST 的数据备份方式。接下来，我们可以设定镜像保存的路径和名称，右上角是扇区选择范围，可以指定扇区段进行复制，如某硬盘前端出现大量坏道，如果强行复制既不安全也花费时间，我们就可以利用此功能从坏道较少的地方开始复制。下方是一个 HASH 值计算工具，可以有效记录数据的原始校验（数据有一个字节的改变 HASH 值都会变化）。还有许多功能如压缩比、镜像分割、坏扇区跳过填充等都是因环境而选择是否采用。

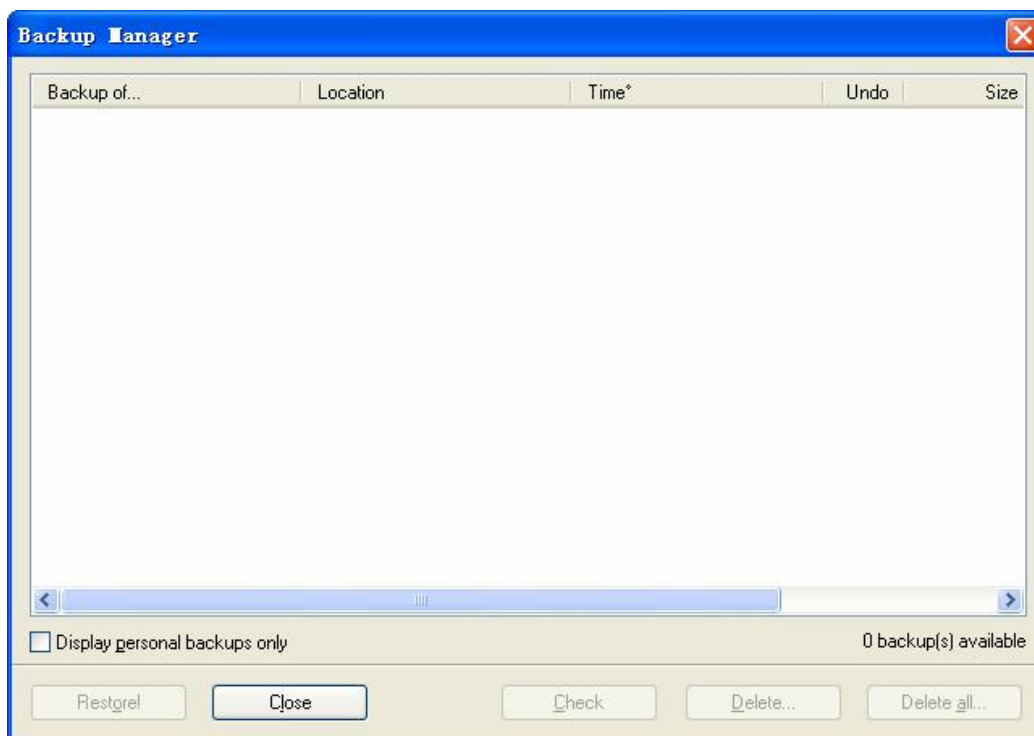


设定好参数后，点击 **OK** 开始复制，当然复制时间的长短和镜像来源物理载体的健康程度息息相关。

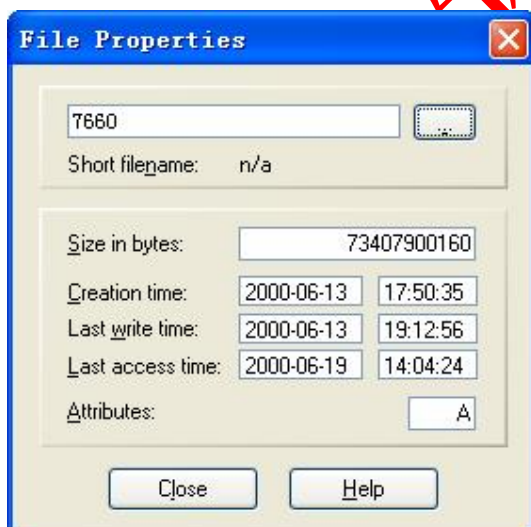


有镜像制作当然就有备份还原，Restore Image 键就是将已成型的镜像文件还原到磁盘中，

注意目标磁盘的环境最好与镜像文件相仿，不然会产生问题，给下一步工作造成困难。



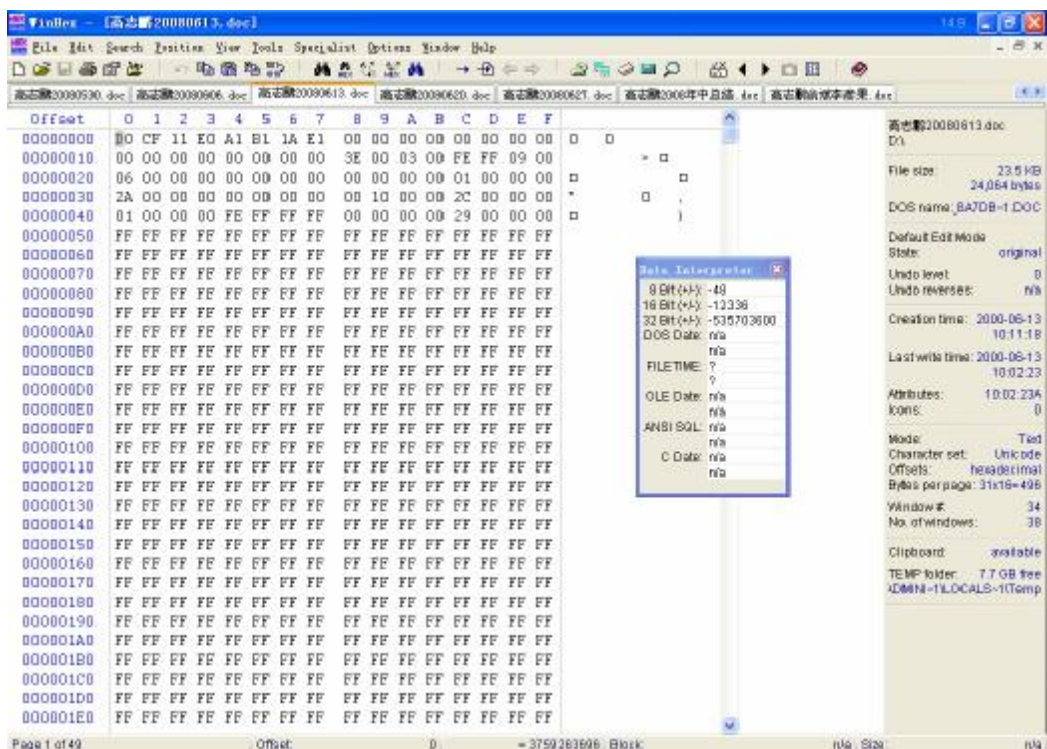
备份管理器可以很好的对备份文件进行管理、归类、错误检查。但是如果一次工程造成的备份文件过多，就会严重侵占磁盘空间，所以应当定期剔除过期或无用的备份。



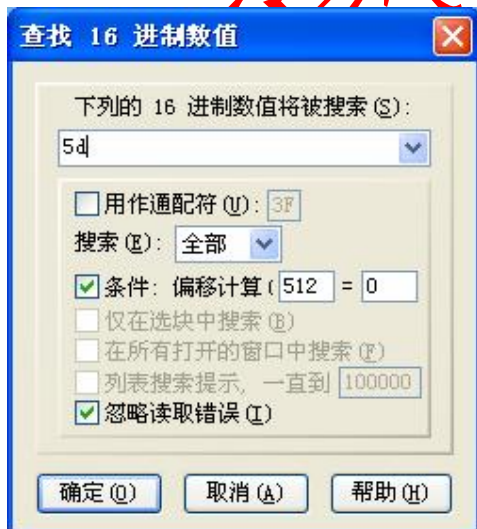
Files Properties 可以显示指定文件的基本属性如体积大小、创建时间、修改时间、访问时间、等。



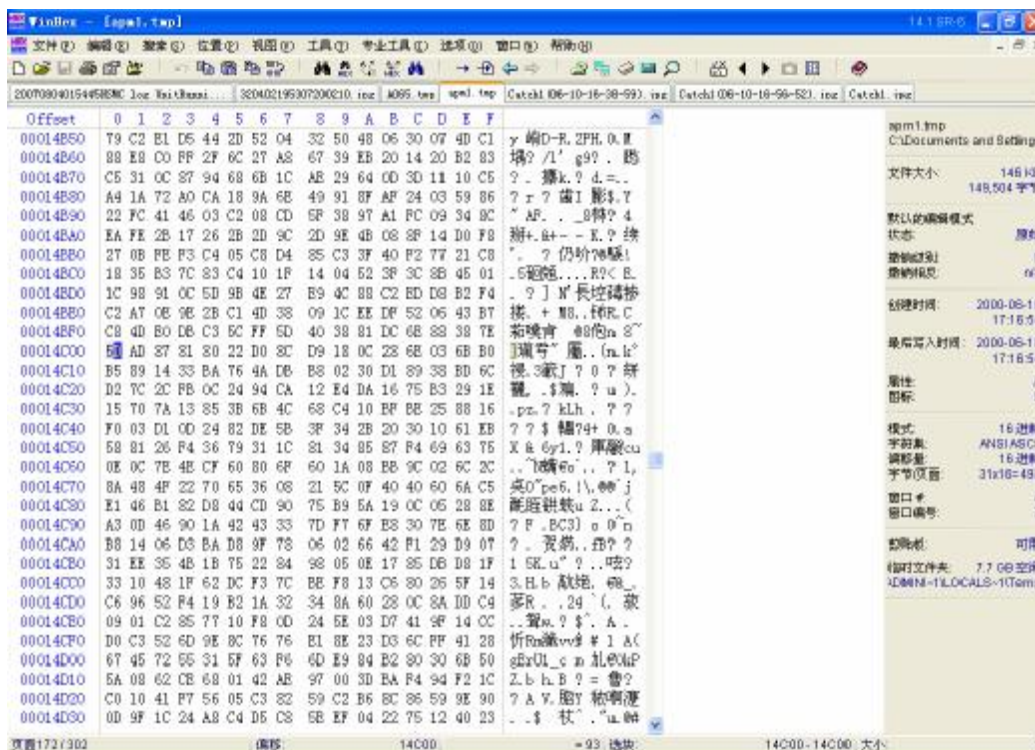
打开文件夹是 WINHEX 的特色功能之一，可以对某文件夹内某类型的文件进行批量展开。以方便后续的同步、对比、批量修改等工作。如图显示的是控制台，上方调用了 WINDOWS 资源管理器界面，查找并选择磁盘内任意目录。Mask 处可以填写一些简单的表达式来完成对文件的筛选如*.doc、*.jpg 等。



如图所示：根目录下所有扩展名为 doc 的文件都被打开了，大大减少工作量。此处需注意的是对象目录中的文件不宜过多，否则会长时间无法完成任务甚至耗尽系统资源造成崩溃。正下方可以改变 WINHEX 对这些文件的访问模式，如默认模式（修改后需要保存）、只读模式、直接修改模式。右边三个选项分别为：包含指定文本、包含指定的 HEX 值、包含文件夹。前两项十分实用，可以利用文件特征进行批量定位，如我们知道某一类文件中都包含字节“5D”和 Unicode 字符“数据”，可是这些文件混杂在文件堆中，用手工查看的方式去定位它们几乎是不可能的，此时 WINHEX 的字节级定位功能就可以大显身手了。

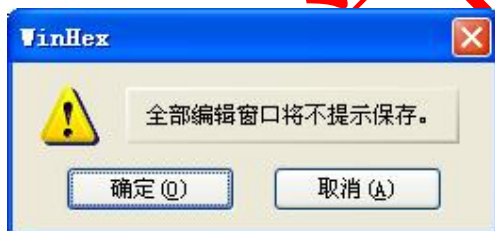


（查找字节“5D”）



(每个包含 5D 的文件都被定位)

如果一组文件被批量修改，那逐一保存显然需要花费很多时间和精力，WINHEX 的批量保存文件功能有效解决了这一问题。一种情况是我只想保存修改过的一类文件，另一种情况是全部打开文件均按照保存处理。这两种保存方式都只提醒一次。



(批量保存提醒)

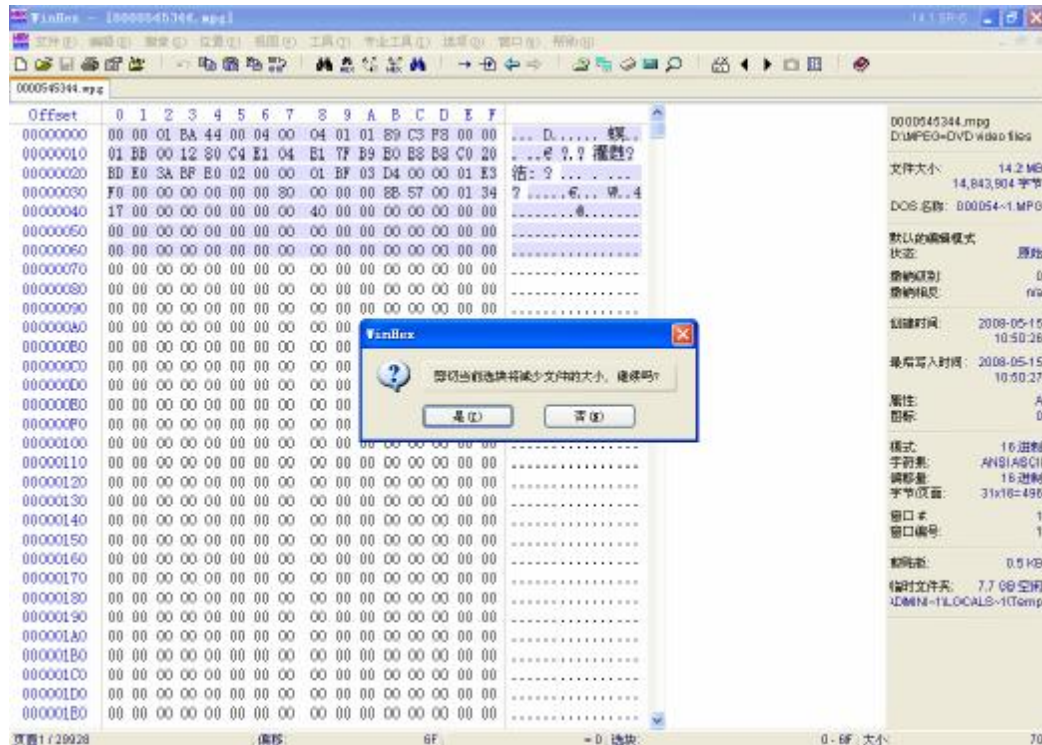
至此，“文件”主菜单的所有子功能就介绍完了，这些操作是 WINHEX 必须用到的，是入门级的，务必熟练掌握。

2.2.2

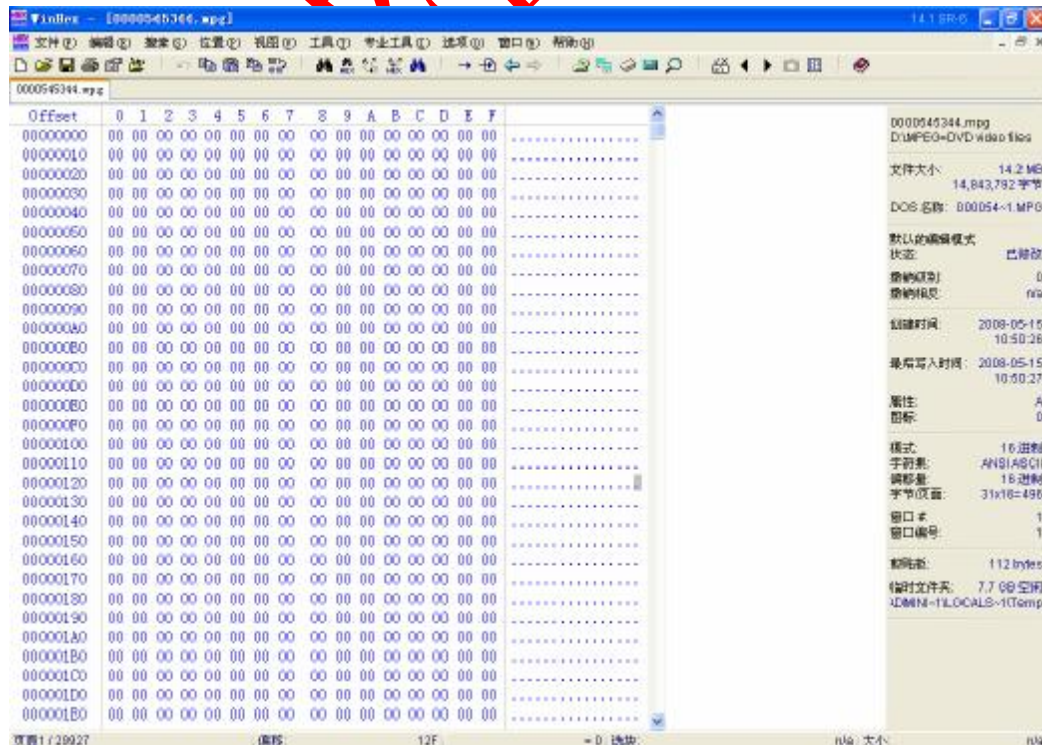
编辑菜单由撤销、剪切、复制、剪贴板、删除、粘贴 0 字节、定义选块、全选、清除选块、数据转换、修改数据、填充数据等子项构成，某些子项还拥有二级子项。从子项名称可以看出，“编辑”主菜单是 WINHEX 中操作性最强的。利用此菜单，我们可以进行字节级别的文件修改工作，如粘贴偏移的数据到正常方位，从磁盘中提取任意数据段写入新文件等。对于已经定义大小的文件项目，甚至还可以采用“补充 0 字节”的方式进行扩容。对于具有保密要求的文件或磁盘，可以利用“修改数据”子项中的各种逻辑代数算法进行简单加密，比如对某硬盘所有数据进行异或修改，要使用时，再利用已知元素逆运算回来。总之，此菜单相关功能是熟练应用 WINHEX 的关键。

“撤销”很好理解，我们做了某些错误的修改，想更正回来，就可以用到它了，这跟 WORD 中的撤销是一个意思，当然也不可能无限制撤销，已经保存的数据不能撤销，这里不再赘述。

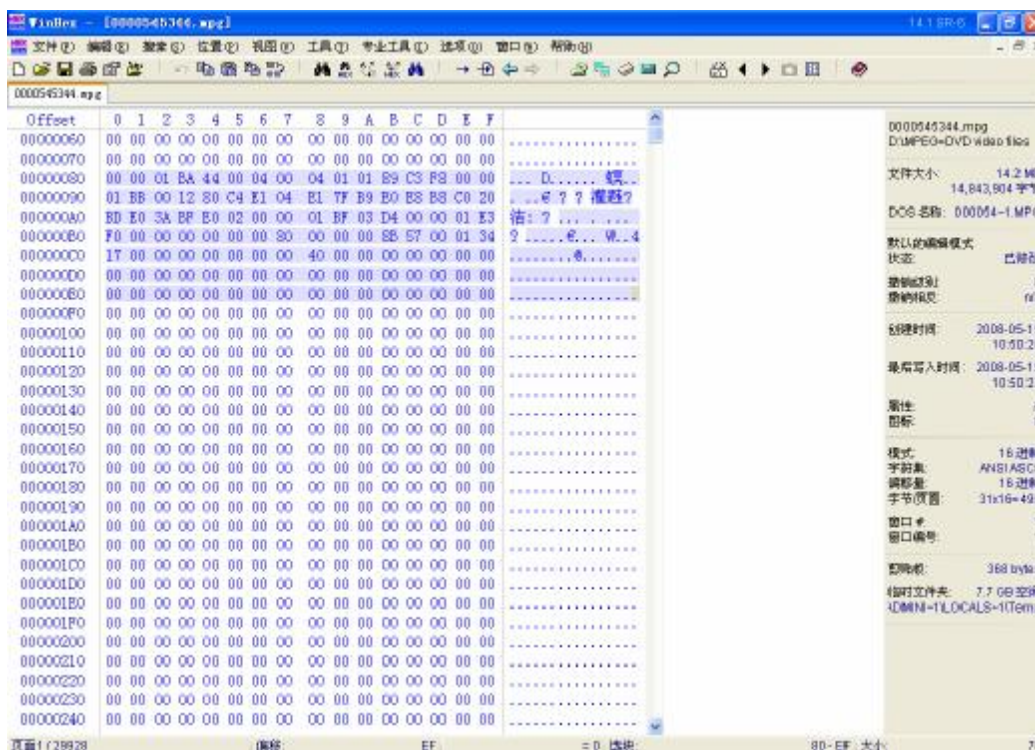
“剪切”就是将一段范围的字节或字符串移动到另一位置，如图所示我们将文件0000545344.mpg中的开始112个字节作剪切处理。完成后这些数据被读入缓存准备写入到另一位置。选择另一位置点击剪贴板>粘贴，数据就被转移到了这里，期间文件大小会产生变化。



✓ (剪切一段字节)



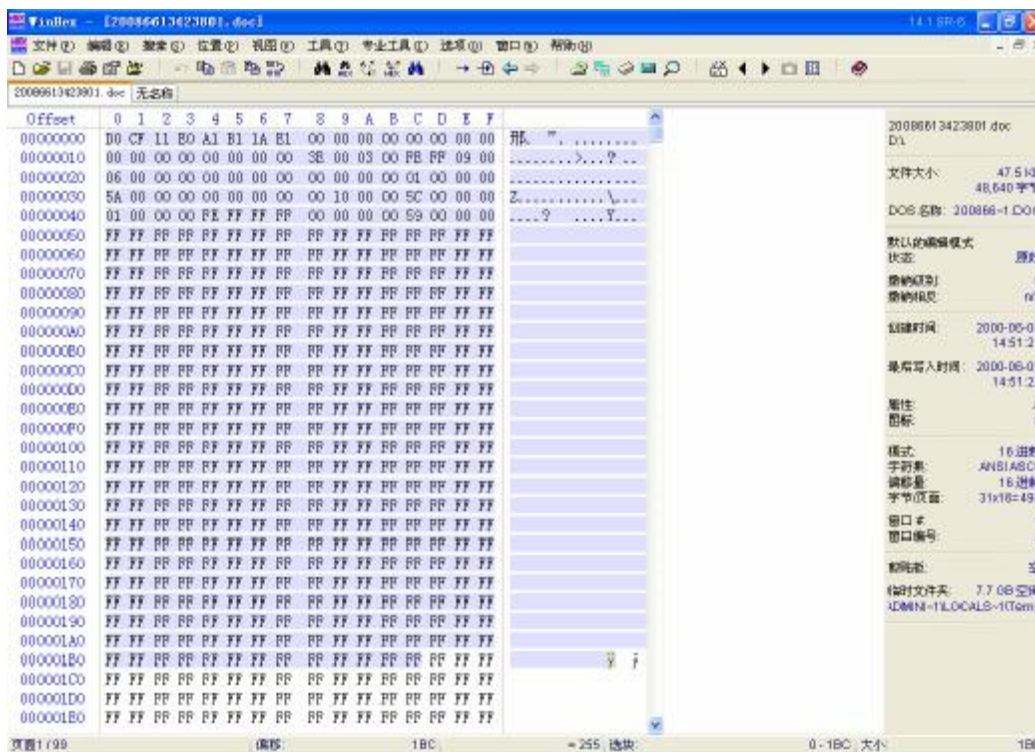
(字节被转走, 文件体积减少)



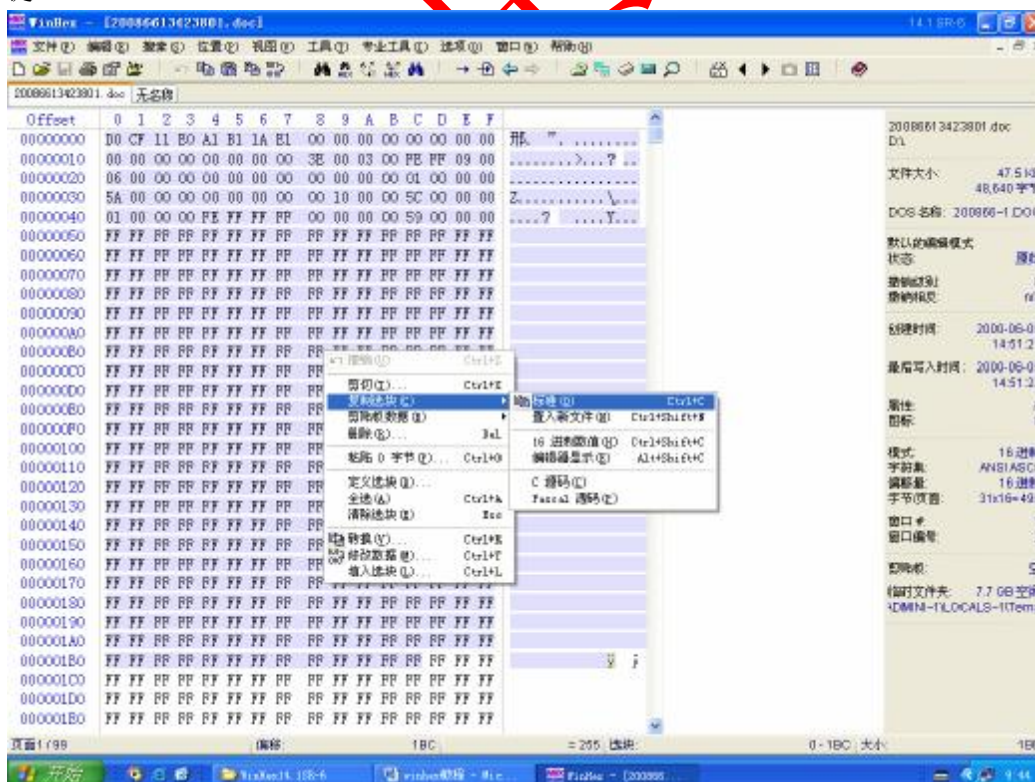
(字节被写入到另一位置)

很多时候我们都用到剪切功能, 如文件头偏移而导致文件无法打开的状况, 我们只需要将文件头粘贴回文件开始的部位即可。我们也可以用此功能将一个文件的有用内容转移到另一个文件中, 或是完成两个文件的拼接, 将其合并为一个大文件。注意最好不要进行超大规模(几百 MB 或几个 GB)的剪切操作, 你的系统资源可能因此被耗尽。

“复制选块”子项是最常用到的选项之一, 在数据恢复中能够准确地判断字节范围并复制到合适的地方, 就已经是至高境界, 当然对字节含义的判断能力是读者日积月累来的。举个例子, 当 DBR 严重损坏时, 磁盘分区会提示“未格式化”字样, 此时我们就需要找到 DBR 备份并复制到该分区的首扇区, 数据恢复工作也随之完成。在手工修复 WORD 文档时, 往往需要复制出其 Unicode 字符区(正文)的内容, 再加以修缮。该子项包含了若干二级子项, 分别是标准复制、复制到一个新文件、复制 HEX 值、复制编辑器视图、复制并构建 C 源码、复制并构建 Pascal 源码。标准复制是我们最常用到的功能, 如图所示:

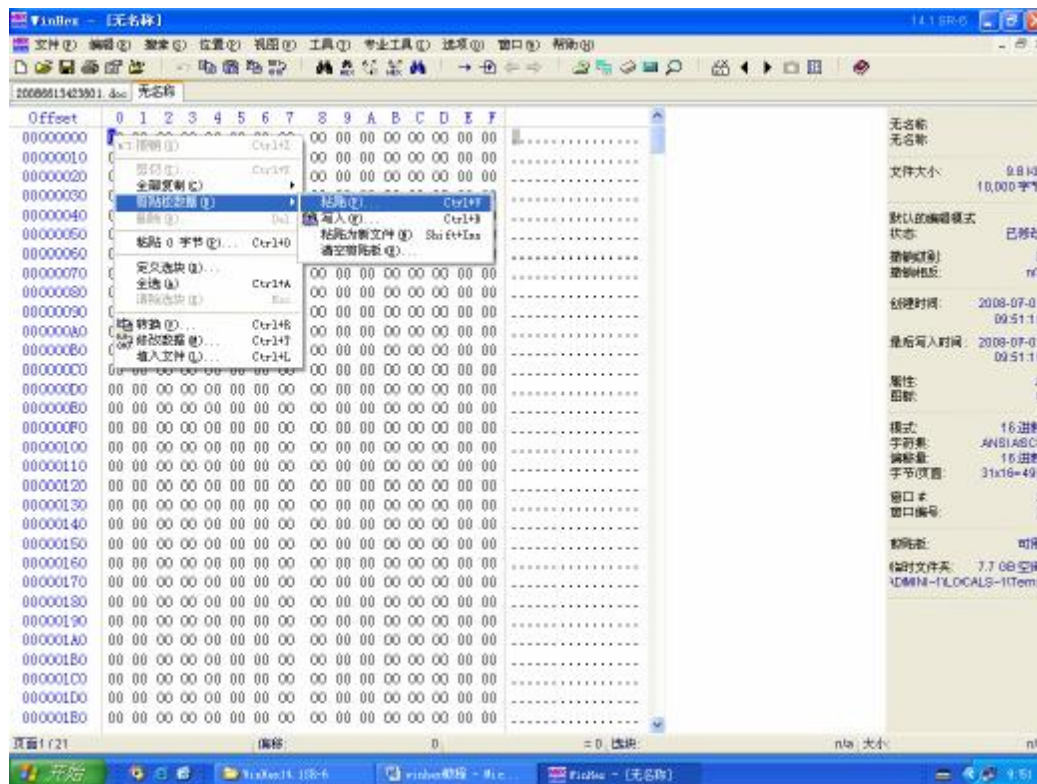


我们先新建一个零字节文件“无名称”，然后打开 WORD 文档 20086613423801.doc，我们直接用鼠标选块然后点击右键（出现的菜单与“编辑”主菜单功能完全相同，只是操作更为便捷）。

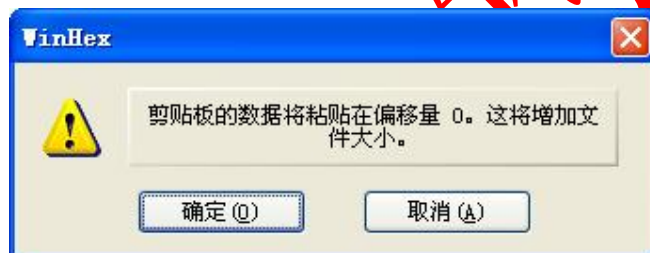


在“复制选块”中选择标准，然后进入“无名称”窗口，选择任意位置（这里选择了首字节）

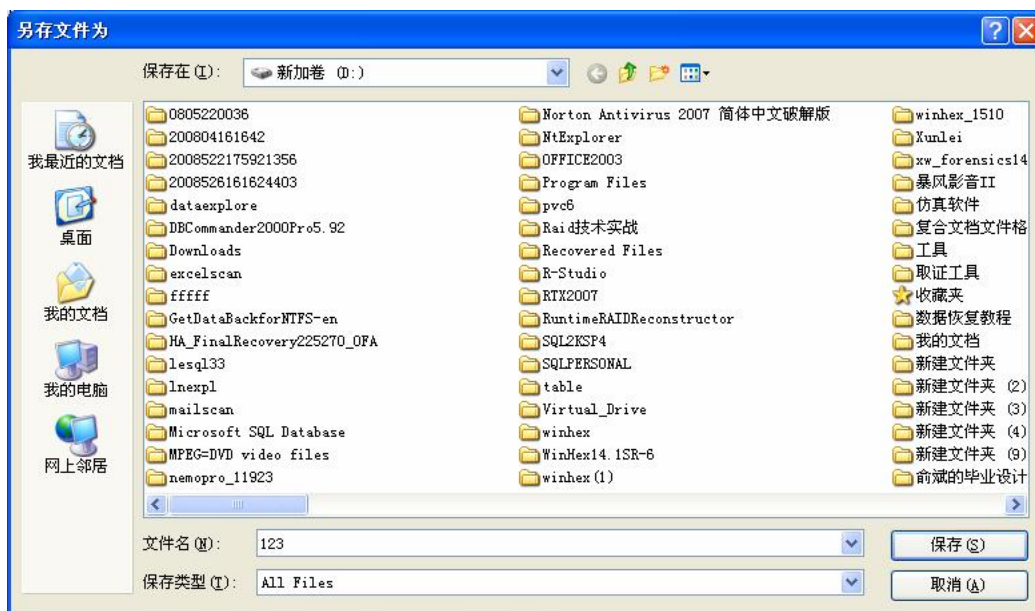
再次点击右键，选择剪贴板数据，在其子项中选择粘贴。



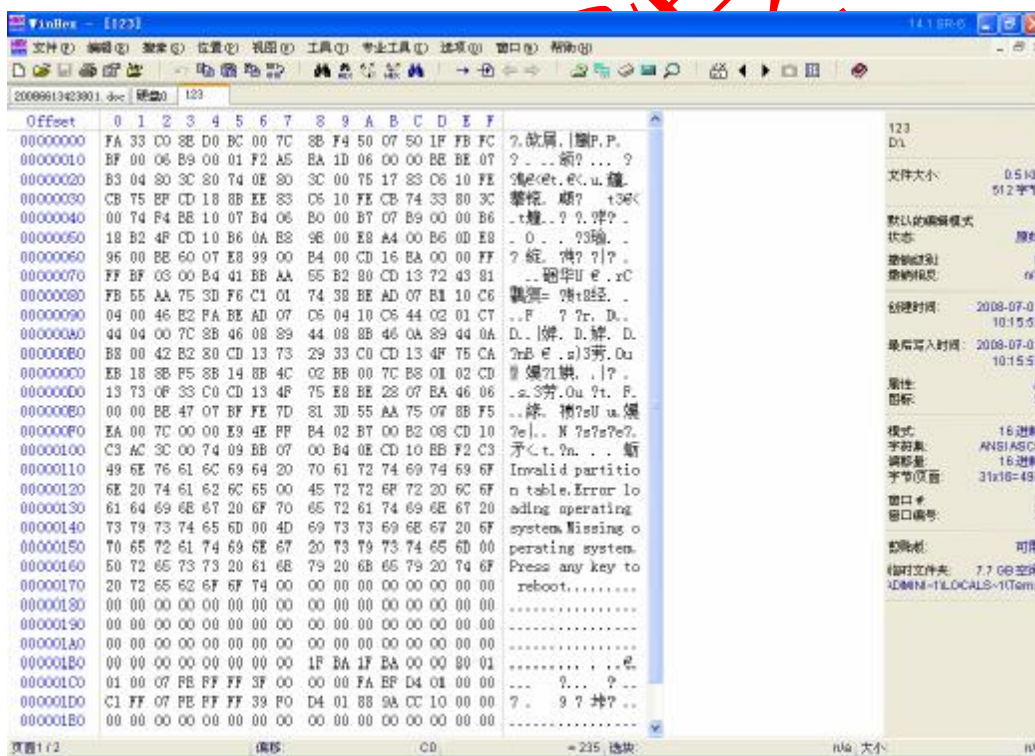
出现以下提示框点击确定



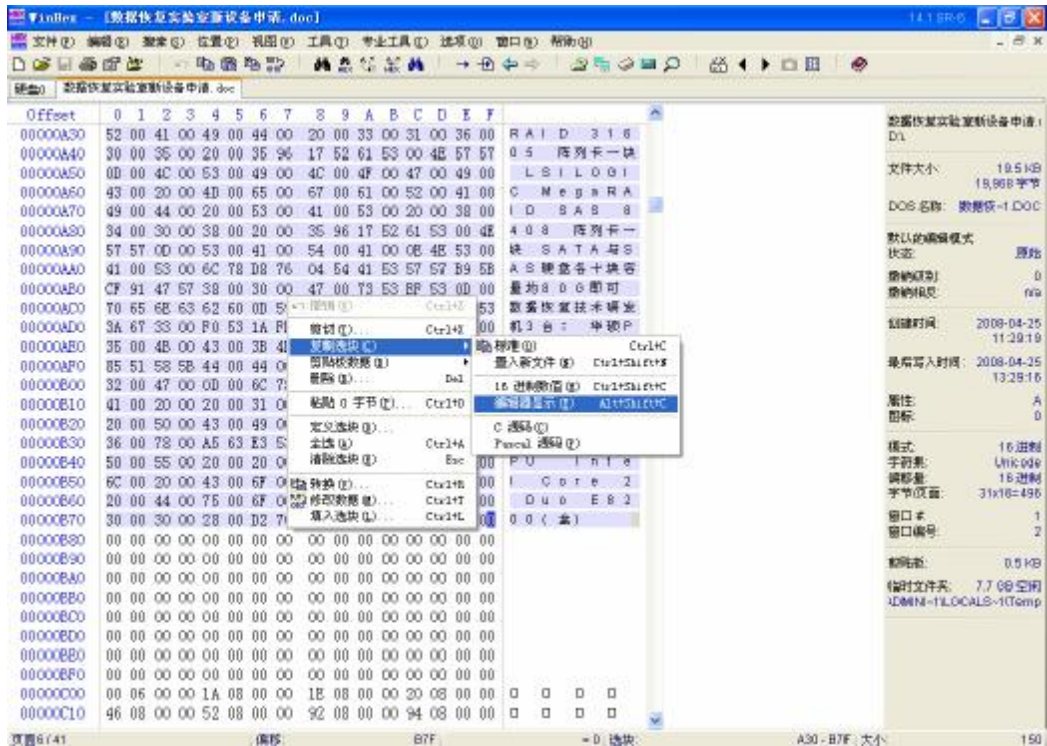
可以看到，选块被完整地复制到“无名称”中。



点击保存，一个大小为 512 字节的新文件成功构建，正是主引导扇区的内容。



复制 HEX 数值，就是说只针对 16 进制字节进行提取，很多情况下可以作为“标准复制”使用，它的优势是可以把 HEX 值复制到 WINHEX 以外的系统中。如我们要将某分区的引导扇区复制到 WORD 文档中，先选块该扇区，点击右键，选择“复制选块”中的“16 进制数值”一项。如图所示：



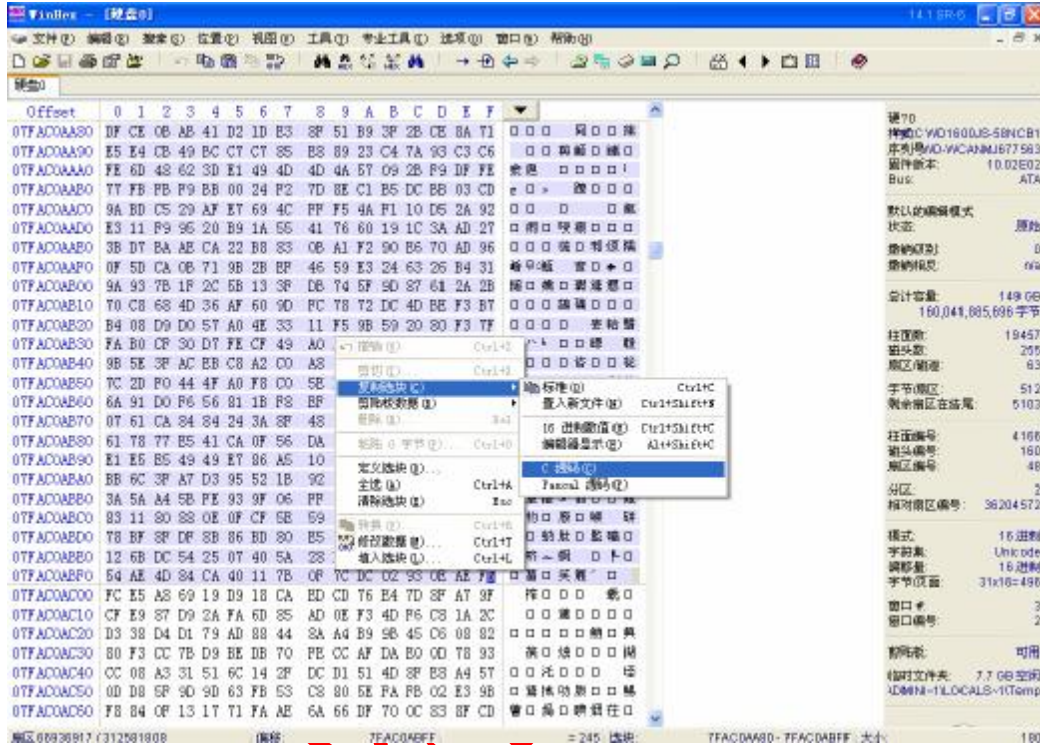
然后在 WORD 文档中进行粘贴：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000A30	52	00	41	00	49	00	44	00	20	00	33	00	31	00	36	00	R.A.I.D. .3.1.6.
00000A40	30	00	35	00	20	00	35	96	17	52	61	53	00	4E	57	57	0.5. .5?RaS.NWW
00000A50	0D	00	4C	00	53	00	49	00	4C	00	4F	00	47	00	49	00	..L.S.I.L.O.G.I.
00000A60	43	00	20	00	4D	00	65	00	67	00	61	00	52	00	41	00	C. .M.e.g.a.R.A.
00000A70	49	00	44	00	20	00	53	00	41	00	53	00	20	00	38	00	I.D. .S.A.S. .8.
00000A80	34	00	30	00	38	00	20	00	35	96	17	52	61	53	00	4E	4.0.8. .5?RaS.N
00000A90	57	57	0D	00	53	00	41	00	54	00	41	00	0E	4E	53	00	WW.S.A.T.A..NS.
00000AA0	41	00	53	00	6C	78	D8	76	04	54	41	53	57	57	B9	5B	A.S.Ix 豹.TASWW 等
00000AB0	CF	91	47	57	38	00	30	00	47	00	73	53	EF	53	0D	00	蟒 GW8.0.GsS 颯..
00000AC0	70	65	6E	63	62	60	0D	59	80	62	2F	67	14	78	D1	53	pench`.Yfb/g.x 祉
00000AD0	3A	67	33	00	F0	53	1A	FF	0D	00	4E	53	55	78	50	00	:g3.鰭. ..NSUxP.
00000AE0	35	00	4B	00	43	00	3B	4E	7F	67	00	4E	57	57	0D	00	5.K.C.;N g.NWW..
00000AF0	85	51	58	5B	44	00	44	00	52	00	32	00	20	00	20	00	匚 X[D.D.R.2. . .
00000B00	32	00	47	00	0D	00	6C	78	D8	76	53	00	41	00	54	00	2.G...Ix 豹 S.A.T.
00000B10	41	00	20	00	20	00	31	00	54	00	0D	00	3E	66	61	53	A. . .1.T...>faS
00000B20	20	00	50	00	43	00	49	00	20	00	2D	00	45	00	31	00	.P.C.I. -.E.1.
00000B30	36	00	78	00	A5	63	E3	53	73	53	EF	53	0D	00	43	00	6.x. 鋳 sS 颯..C.
00000B40	50	00	55	00	20	00	20	00	49	00	6E	00	74	00	65	00	P.U. . .In.te.
00000B50	6C	00	20	00	43	00	6F	00	72	00	65	00	20	00	32	00	l. .C.o.r.e. .2.
00000B60	20	00	44	00	75	00	6F	00	20	00	45	00	38	00	32	00	.D.u.o. .E.8.2.
00000B70	30	00	30	00	28	00	D2	76	29	00	0D	00	00	00	00	00	0.0.(.襪).....

此功能在学术或科研引用时非常便利，它意味着我们可以抽取任意数据区作深入分析。

C 源码（二级子项）是一个新颖而有趣的功能。因为磁盘编辑器往往配合编程语言做程序开发用，所以会不可避免地引入编辑区的内容，但是 WINHEX 的显示方式和编程语言是大不相同的，这就需要将数据提取后依次改变成符合编程语言的格式。但是，如果纯粹靠程序员人力完成，将是一项十分浩大的工程。WINHEX 开发人员充分考虑了其实用性，就在其功能中加入了多种格式转换工具，而“复制 C 源码”就是其中一个。

我们要将选块内的字节转换为 C 程序可以识别格式。如图所示：



复制到 C 程序中显示如下：

```
unsigned char data[384] = {
    0xDF, 0xCE, 0x0B, 0xAB, 0x41, 0xD2, 0x1D, 0xE3, 0x8F, 0x51, 0xB9, 0x3F, 0x2B, 0xCE,
    0x8A, 0x71,
    0xE5, 0xE4, 0xCB, 0x49, 0xBC, 0xC7, 0xC7, 0x85, 0xE8, 0x89, 0x23, 0xC4, 0x7A, 0x93,
    0xC3, 0xC6,
    0xFE, 0x6D, 0x48, 0x62, 0x3D, 0xE1, 0x49, 0x4D, 0x4D, 0x4A, 0x57, 0x09, 0x2B, 0xF9,
    0xDF, 0xFE,
    0x77, 0xFB, 0xFB, 0xF9, 0xBB, 0x00, 0x24, 0xF2, 0x7D, 0x8E, 0xC1, 0xB5, 0xDC, 0xBB,
    0x03, 0xCD,
    0x9A, 0xBD, 0xC5, 0x29, 0xAF, 0xE7, 0x69, 0x4C, 0xFF, 0xF5, 0x4A, 0xF1, 0x10, 0xD5,
    0x2A, 0x92,
    0xE3, 0x11, 0xF9, 0x95, 0x20, 0xB9, 0x1A, 0x55, 0x41, 0x76, 0x60, 0x19, 0x1C, 0x3A,
    0xAD, 0x27,
    0x3B, 0xD7, 0xBA, 0xAE, 0xCA, 0x22, 0xB8, 0x83, 0x0B, 0xA1, 0xF2, 0x90, 0xE6, 0x70,
    0xAD, 0x96,
    0x0F, 0x5D, 0xCA, 0x0B, 0x71, 0x9B, 0x2B, 0xEF, 0x46, 0x59, 0xE3, 0x24, 0x63, 0x26,
    0xB4, 0x31,
    0x9A, 0x93, 0x7B, 0x1F, 0x2C, 0x5B, 0x13, 0x3F, 0xDB, 0x74, 0x5F, 0x9D, 0x87, 0x61,
```

```
0x2A, 0x2B,
    0x70, 0xC8, 0x68, 0x4D, 0x36, 0xAF, 0x60, 0x9D, 0xFC, 0x78, 0x72, 0xDC, 0x4D, 0xBE,
0xF3, 0xB7,
    0xB4, 0x08, 0xD9, 0xD0, 0x57, 0xA0, 0x4E, 0x33, 0x11, 0xF5, 0x9B, 0x59, 0x20, 0x80,
0xF3, 0x7F,
    0xFA, 0xB0, 0xCF, 0x30, 0xD7, 0xFE, 0xCF, 0x49, 0xA0, 0xA4, 0xDA, 0x66, 0x81, 0xED,
0xF7, 0x7A,
    0x9B, 0x5E, 0x3F, 0xAC, 0xEB, 0xC8, 0xA2, 0xC0, 0xA8, 0x54, 0x5A, 0x22, 0x50, 0x43,
0xC7, 0x68,
    0x7C, 0x2D, 0xF0, 0x44, 0x4F, 0xA0, 0xF8, 0xC0, 0x5E, 0xEB, 0x03, 0x05, 0x2E, 0x52,
0x21, 0x8F,
    0x6A, 0x91, 0xD0, 0xF6, 0x56, 0x81, 0x1B, 0xF8, 0xEF, 0x57, 0xC9, 0x8E, 0xF4, 0x15,
0x40, 0x4B,
    0x07, 0x61, 0xCA, 0x84, 0x84, 0x24, 0x3A, 0x8F, 0x48, 0x7F, 0xFA, 0x84, 0x64, 0x6C,
0xE1, 0x76,
    0x61, 0x78, 0x77, 0xE5, 0x41, 0xCA, 0x0F, 0x56, 0xDA, 0x2B, 0x6F, 0x7A, 0x97, 0xBD,
0x69, 0x49,
    0xE1, 0xE5, 0xE5, 0x49, 0x49, 0xE7, 0x86, 0xA5, 0x10, 0x3F, 0xEC, 0xD6, 0xD9, 0x2A,
0x2F, 0x8A,
    0xBB, 0x6C, 0x3F, 0xA7, 0xD3, 0x95, 0x52, 0x1B, 0x92, 0x92, 0xE9, 0xC8, 0x6A, 0x9B,
0x64, 0x14,
    0x3A, 0x5A, 0xA4, 0x5B, 0xFE, 0x93, 0x9F, 0x06, 0xFF, 0x83, 0xC1, 0xFF, 0x30, 0x2E,
0x97, 0x9F,
    0x83, 0x11, 0x80, 0x88, 0x0F, 0x0F, 0xCF, 0x5E, 0x59, 0x38, 0x89, 0x55, 0x38, 0xF3, 0x51,
0x78,
    0x78, 0xBF, 0x8F, 0xDF, 0x8B, 0x86, 0xBD, 0x80, 0xE5, 0x39, 0xE3, 0x76, 0x11, 0x79,
0x0B, 0x45,
    0x12, 0x6B, 0xDC, 0x54, 0x25, 0x07, 0x40, 0x5A, 0x28, 0xEC, 0xD3, 0x03, 0x1E, 0x25,
0x5A, 0x1D,
    0x54, 0xAE, 0x4D, 0x84, 0xCA, 0x40, 0x11, 0x7B, 0x0F, 0x7C, 0xDC, 0x02, 0x93, 0x0E,
0xAE, 0xF5
};
```

很不错，完全符合这一类程序语言的语法标准。

Pascal 语言深受反病毒学家的喜爱，而 WINHEX 作为“逆向工程”的利刃同样不可忽视，复制 Pascal 源码（二级子项）为以上工作带来了福音，具体操作同“C 源码”。如图所示：

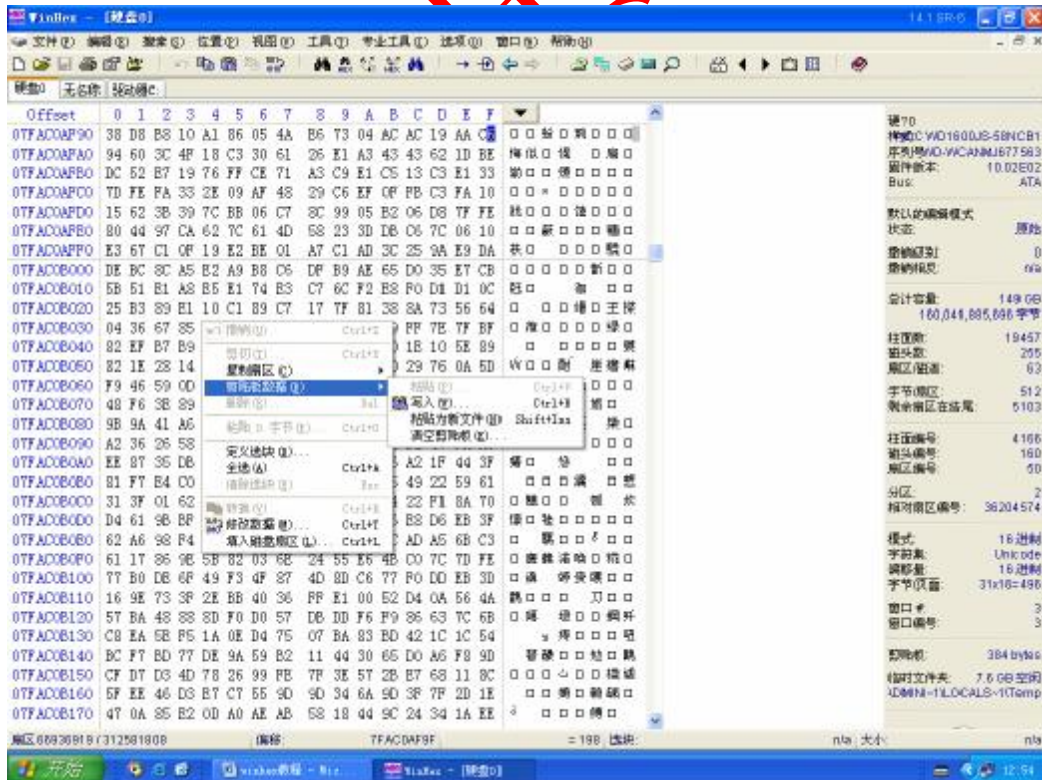
data: array[0..383] of byte = (

```
$DF, $CE, $0B, $AB, $41, $D2, $1D, $E3, $8F, $51, $B9, $3F, $2B, $CE, $8A, $71,
$E5, $E4, $CB, $49, $BC, $C7, $C7, $85, $E8, $89, $23, $C4, $7A, $93, $C3, $C6,
$FE, $6D, $48, $62, $3D, $E1, $49, $4D, $4D, $4A, $57, $09, $2B, $F9, $DF, $FE,
$77, $FB, $FB, $F9, $BB, $00, $24, $F2, $7D, $8E, $C1, $B5, $DC, $BB, $03, $CD,
$9A, $BD, $C5, $29, $AF, $E7, $69, $4C, $FF, $F5, $4A, $F1, $10, $D5, $2A, $92,
$E3, $11, $F9, $95, $20, $B9, $1A, $55, $41, $76, $60, $19, $1C, $3A, $AD, $27,
$3B, $D7, $BA, $AE, $CA, $22, $B8, $83, $0B, $A1, $F2, $90, $E6, $70, $AD, $96,
$0F, $5D, $CA, $0B, $71, $9B, $2B, $EF, $46, $59, $E3, $24, $63, $26, $B4, $31,
```


\$9A, \$93, \$7B, \$1F, \$2C, \$5B, \$13, \$3F, \$DB, \$74, \$5F, \$9D, \$87, \$61, \$2A, \$2B, \$70, \$C8, \$68, \$4D, \$36, \$AF, \$60, \$9D, \$FC, \$78, \$72, \$DC, \$4D, \$BE, \$F3, \$B7, \$B4, \$08, \$D9, \$D0, \$57, \$A0, \$4E, \$33, \$11, \$F5, \$9B, \$59, \$20, \$80, \$F3, \$7F, \$FA, \$B0, \$CF, \$30, \$D7, \$FE, \$CF, \$49, \$A0, \$A4, \$DA, \$66, \$81, \$ED, \$F7, \$7A, \$9B, \$5E, \$3F, \$AC, \$EB, \$C8, \$A2, \$C0, \$A8, \$54, \$5A, \$22, \$50, \$43, \$C7, \$68, \$7C, \$2D, \$F0, \$44, \$4F, \$A0, \$F8, \$C0, \$5E, \$EB, \$03, \$05, \$2E, \$52, \$21, \$8F, \$6A, \$91, \$D0, \$F6, \$56, \$81, \$1B, \$F8, \$EF, \$57, \$C9, \$8E, \$F4, \$15, \$40, \$4B, \$07, \$61, \$CA, \$84, \$84, \$24, \$3A, \$8F, \$48, \$7F, \$FA, \$84, \$64, \$6C, \$E1, \$76, \$61, \$78, \$77, \$E5, \$41, \$CA, \$0F, \$56, \$DA, \$2B, \$6F, \$7A, \$97, \$BD, \$69, \$49, \$E1, \$E5, \$E5, \$49, \$49, \$E7, \$86, \$A5, \$10, \$3F, \$EC, \$D6, \$D9, \$2A, \$2F, \$8A, \$BB, \$6C, \$3F, \$A7, \$D3, \$95, \$52, \$1B, \$92, \$92, \$E9, \$C8, \$6A, \$9B, \$64, \$14, \$3A, \$5A, \$A4, \$5B, \$FE, \$93, \$9F, \$06, \$FF, \$83, \$C1, \$FF, \$30, \$2E, \$97, \$9F, \$83, \$11, \$80, \$88, \$0E, \$0F, \$CF, \$5E, \$59, \$38, \$89, \$55, \$38, \$F3, \$51, \$78, \$78, \$BF, \$8F, \$DF, \$8B, \$86, \$BD, \$80, \$E5, \$39, \$E3, \$76, \$11, \$79, \$0B, \$45, \$12, \$6B, \$DC, \$54, \$25, \$07, \$40, \$5A, \$28, \$EC, \$D3, \$03, \$1E, \$25, \$5A, \$1D, \$54, \$AE, \$4D, \$84, \$CA, \$40, \$11, \$7B, \$0F, \$7C, \$DC, \$02, \$93, \$0E, \$AE, \$F5

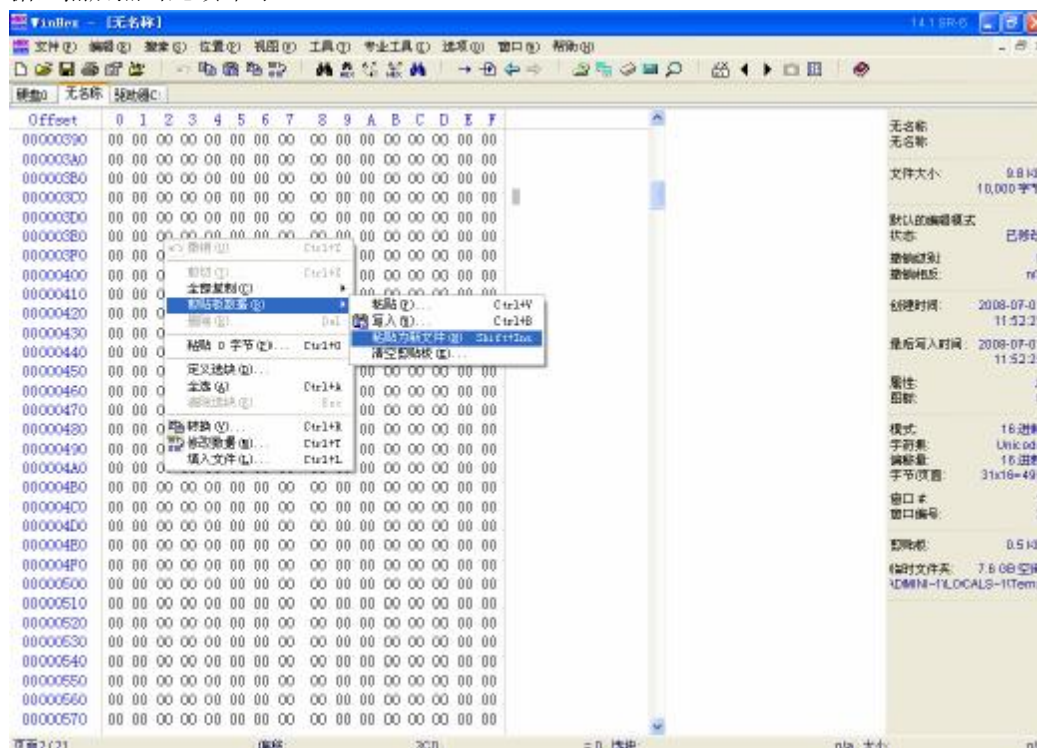
);

剪贴板数据子项与其它子项相辅相成，复制、剪切的东西首先存在这里，每次都覆盖上次的内容。它也有四个二级子项：粘贴、写入、粘贴为新文件、清空剪切板。前两项我们在“复制”中已经反复演示，但要说明一点，粘贴和写入虽然在结果上似乎完全相同，但在使用环境中不同要求，比如要复制一段字节到文件中，就可以随意使用这两项中的一项，但是如果复制目标是严格按照扇区划分的磁盘，就只能用写入，粘贴项会表示为未激活状态灰色。

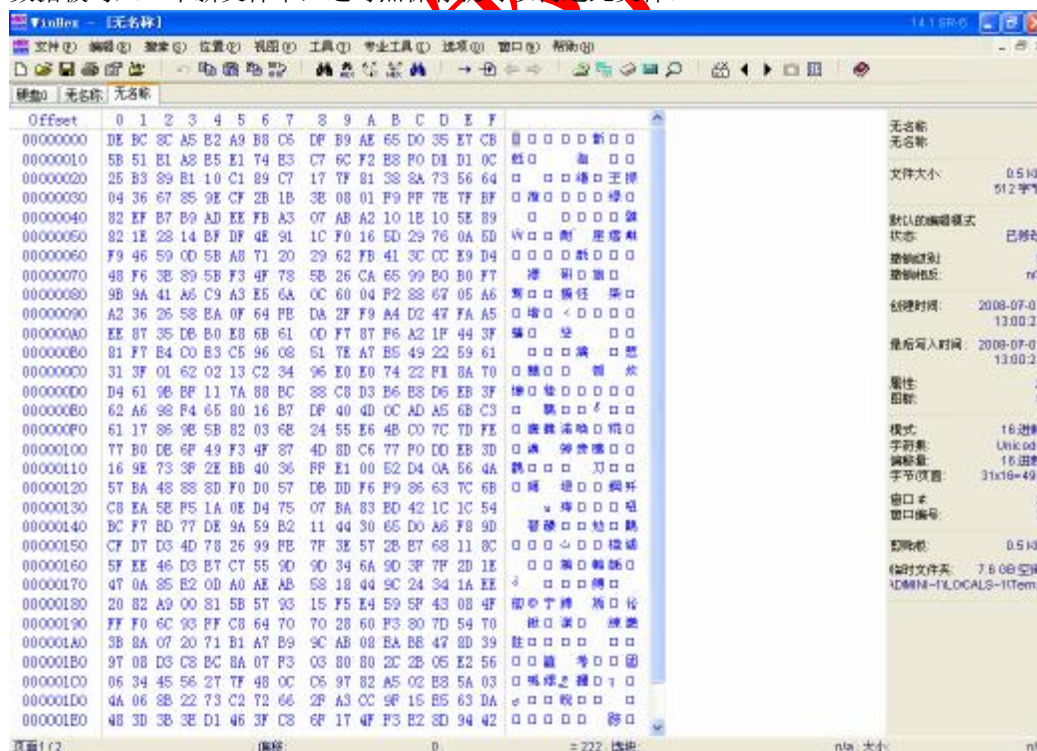


(粘贴键成为灰色)

粘贴为新文件（二级子项）和“复制”中“置入一个新文件”很相似，先在剪切板中存入数据，然后点击此项即可。

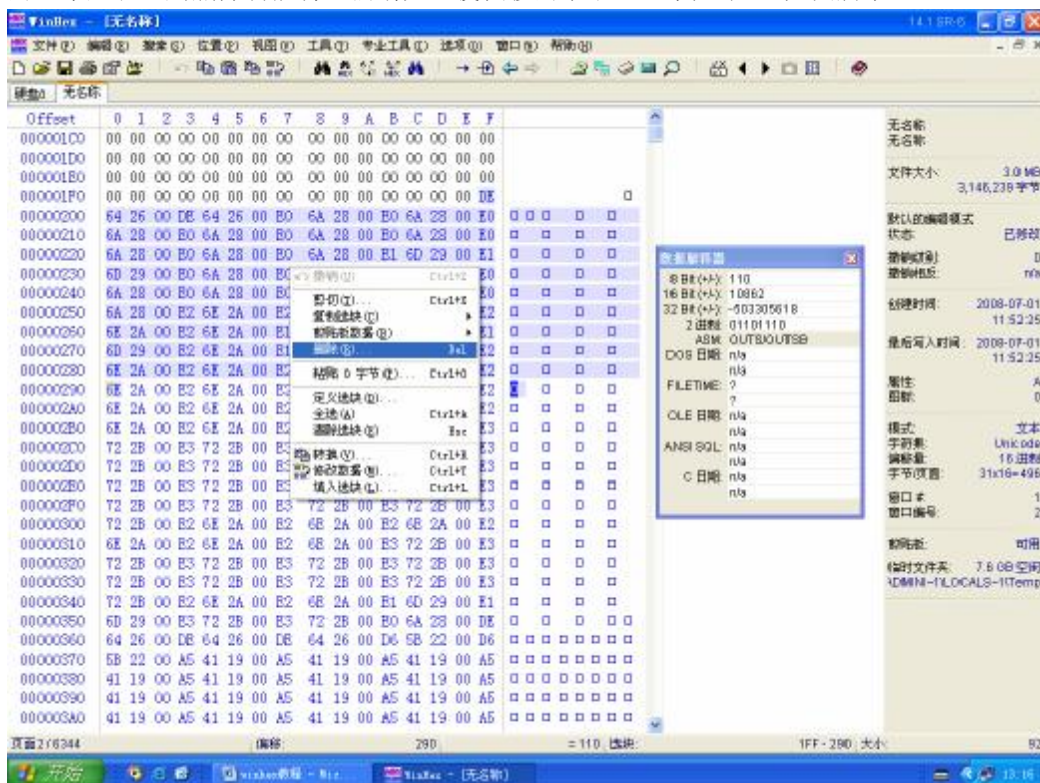


数据被写入一个新文件中，这时点保存就可以构造此文件。

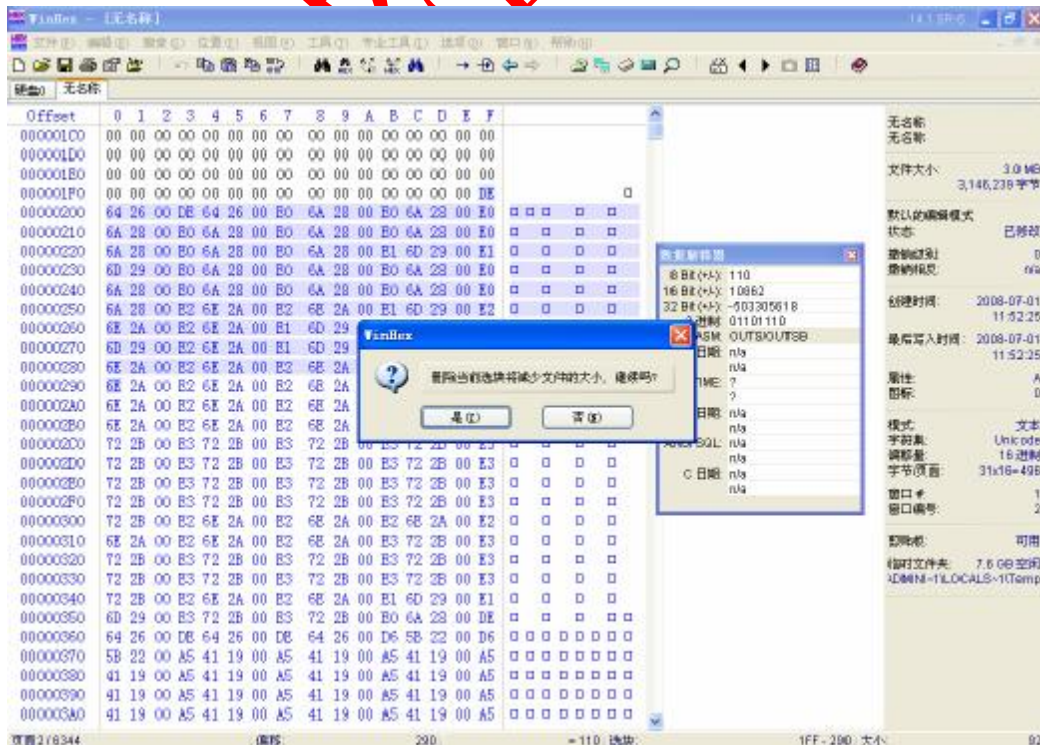


删除子项很好理解，就是把我们认为不需要的数据部分去掉。这里要严格注意的是：删除不同与“填零”，前者会造成文件体积的减少和数据排列方位的改变，从而对文件功能产生实

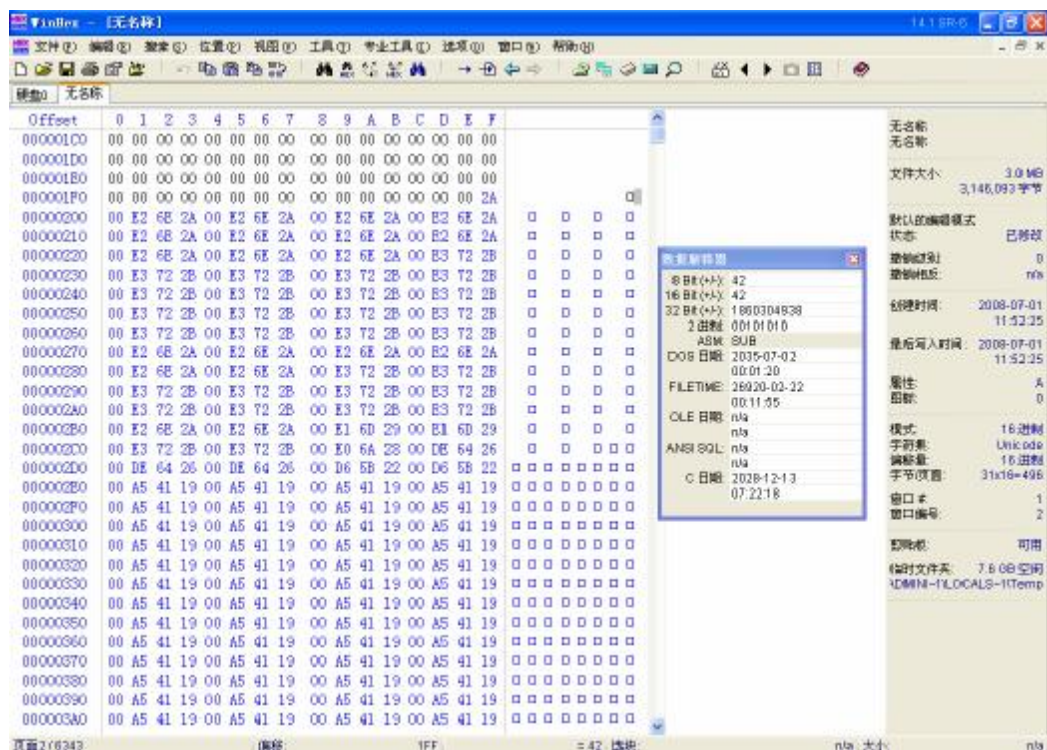
质影响，后者只是在原来的字节位置上将数值变为 0 而已，文件体积不会发生变化，其它字节也不会因此产生位移。我们在数据恢复中经常会碰到某些恢复出的文件体内参杂着不少“冗余”，而导致文件无法正常使用，此时我们就可以计算出“冗余”部分（需要文件结构的基本知识）用删除功能将它们消灭，使偏移的字节“返本归原”。如图所示：



（删除选块部分）

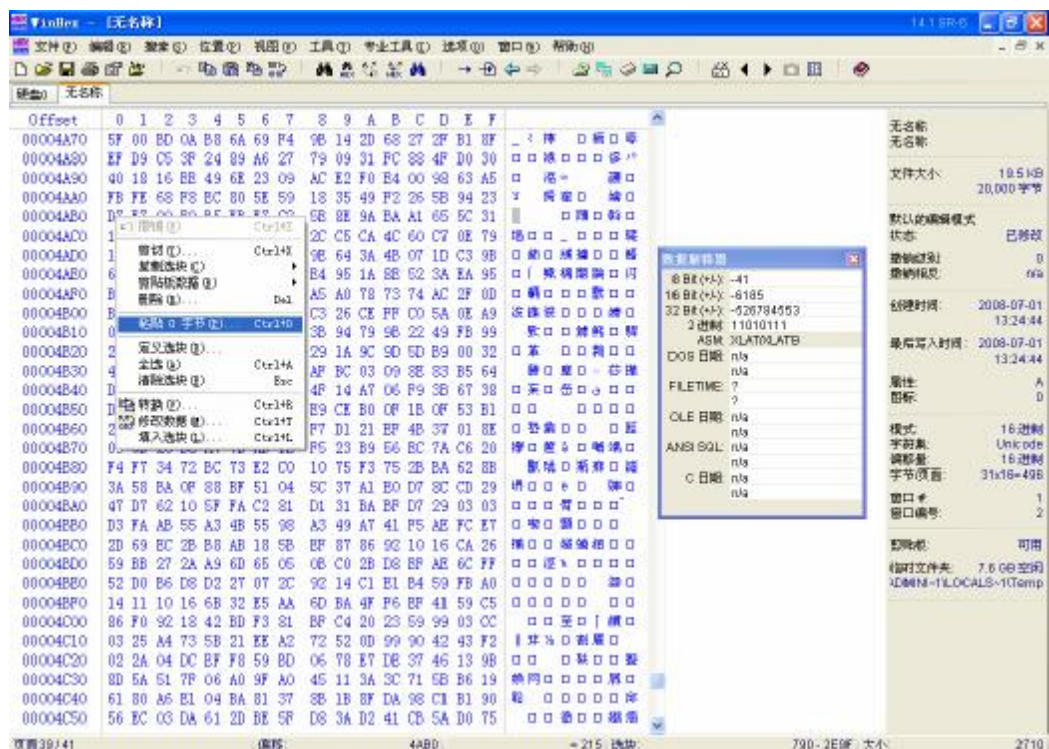


(会有警告提示)



(选块被删除)

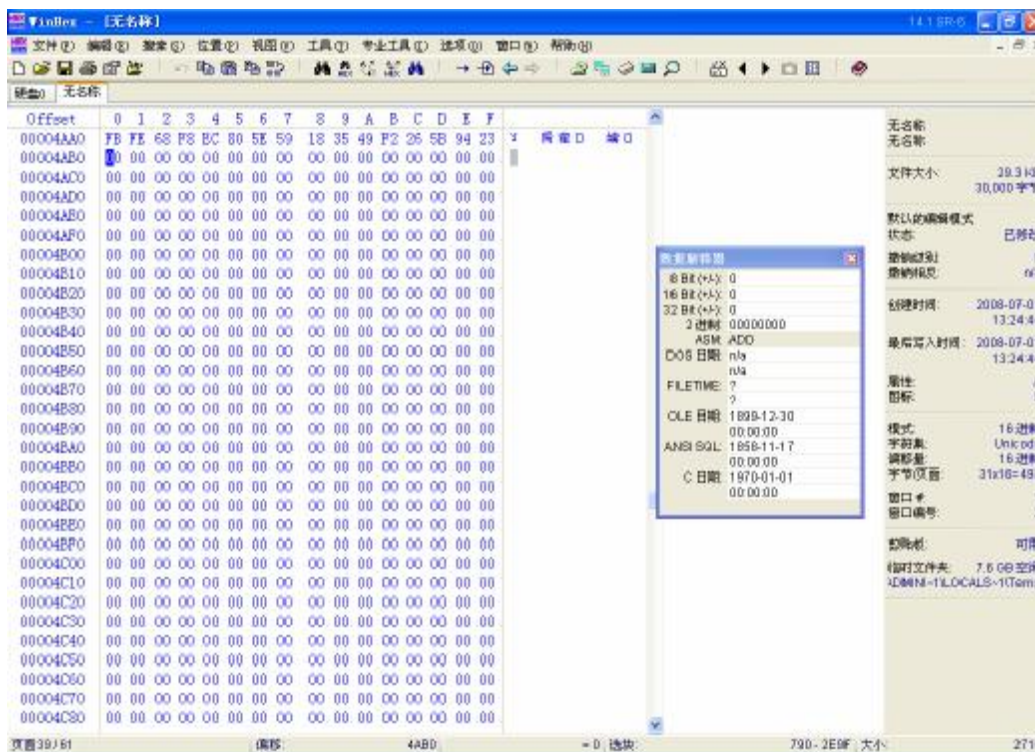
粘贴 0 字节子项和删除子项完全相反，一个减少文件体积，一个为文件扩容。该功能可以在文件中的任何位置实现 0 字节插入，从而凭空增加文件的总字节数，用户也可以为这些新的“0”赋予意义。该功能在数据恢复工作中很少用到。首先在任意位置点击右键该子项，如图所示：



出现对话框，要求输入插入 0 字节的数量。数字最好不要太大，这里我们输入 10000。

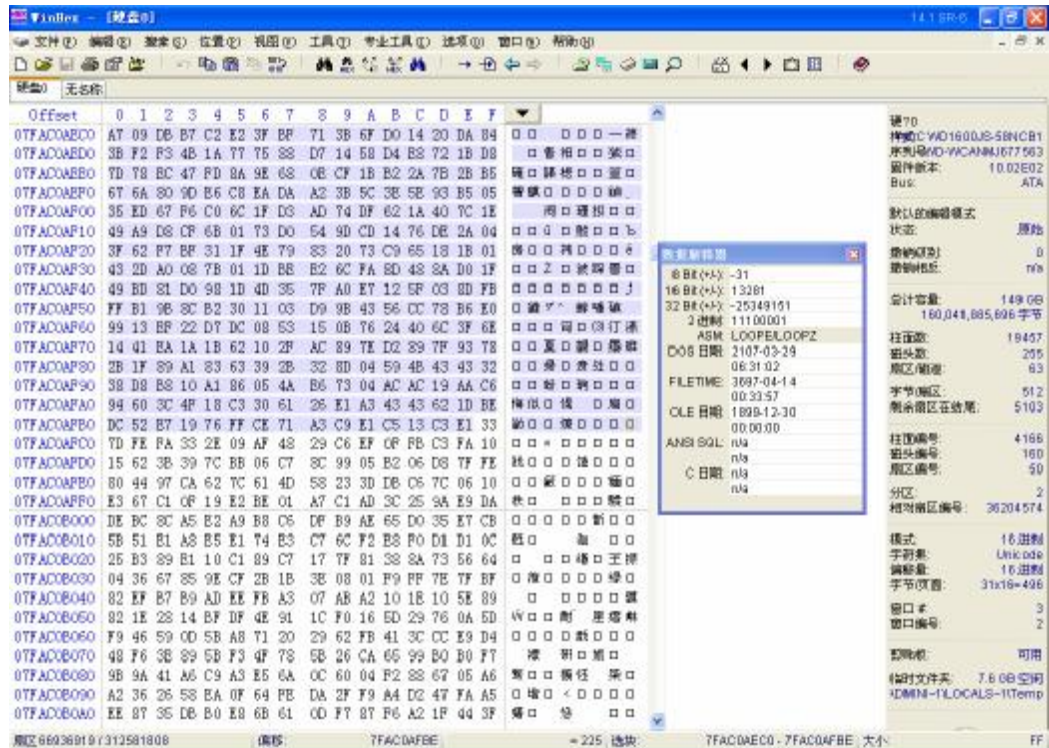


我们看到，文件的页面数增加了很多，出现大量连续的“00”。

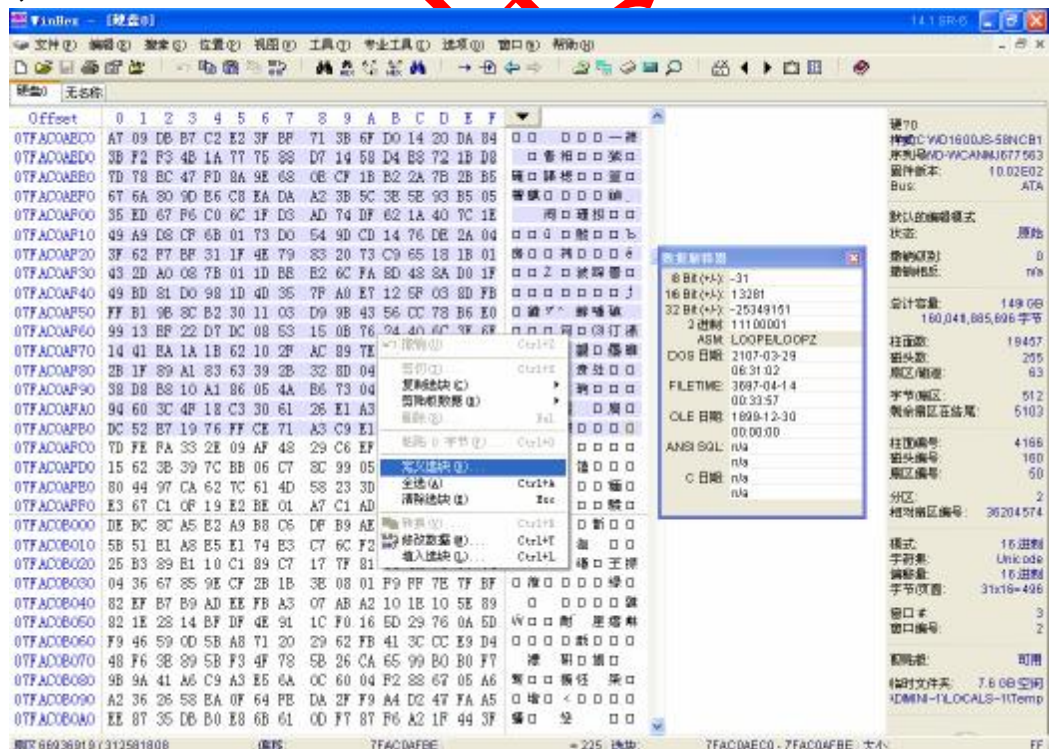


定义选块子项非常考验用户的动手能力，选块是定义操作对象范围的基础，其它一切操作都是建立在“选块准确”的大前提下。选块的方法大体分为三种：指针范围选块、使用定义选块子项、开始结尾法。

顾名思义，指针范围选块就是利用鼠标拖动功能进行标记，就好比在“我的电脑”中选择文件一样。这是最为快捷的选块方式，但是由于鼠标拖动范围有限，用户不可能采用此法大规模选块，而且选块结尾也很难确定。所以只有在选块范围小于一个扇区或一个页面时才建议用此法。



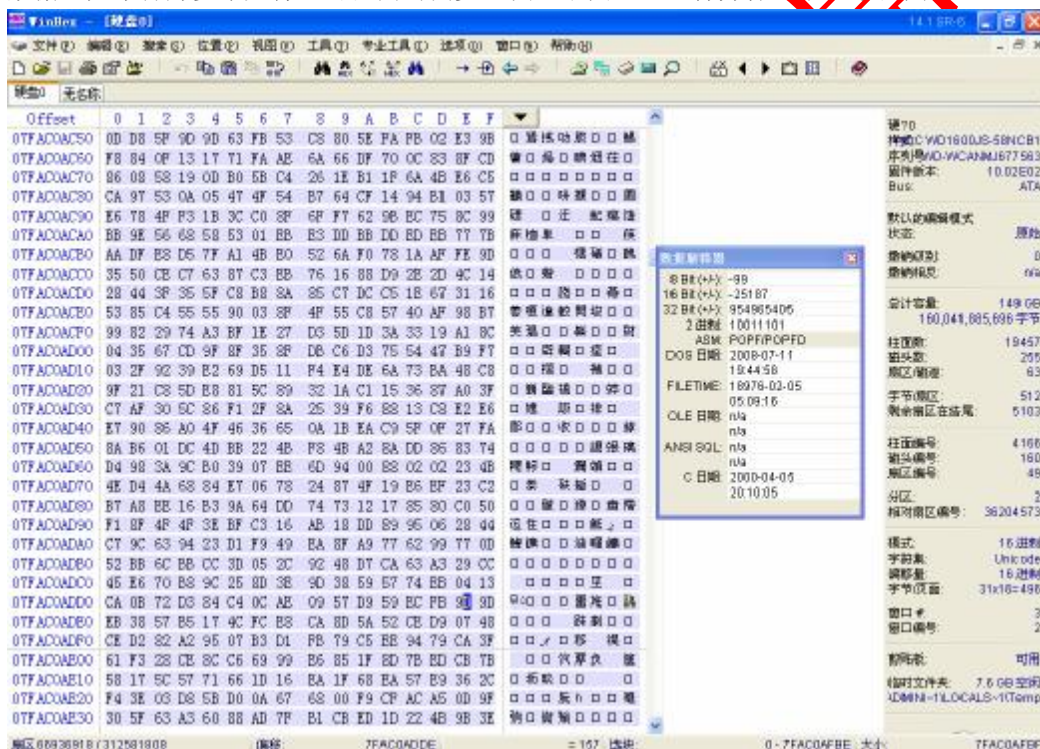
此外我们可以使用一种更为精确的手段实现选块，也就是纯粹用数字进行定位。定义选块子项就可以实现此功能，当我们事先判断好选块范围参数，输入到定义选块控制台中：如图所示：



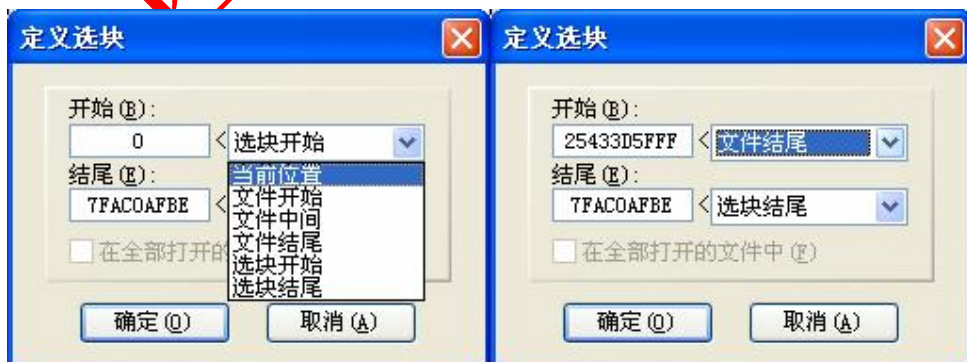
编辑器任意位置点击右键定义选块子项，出现以下对话框；



我们可以看到，只要在“开始”和“结尾”中填入选块偏移量，就可以选择标记一个范围的数据。如果打开多个文件还可以同时同步选块。下图已经选择并标记了一个范围：

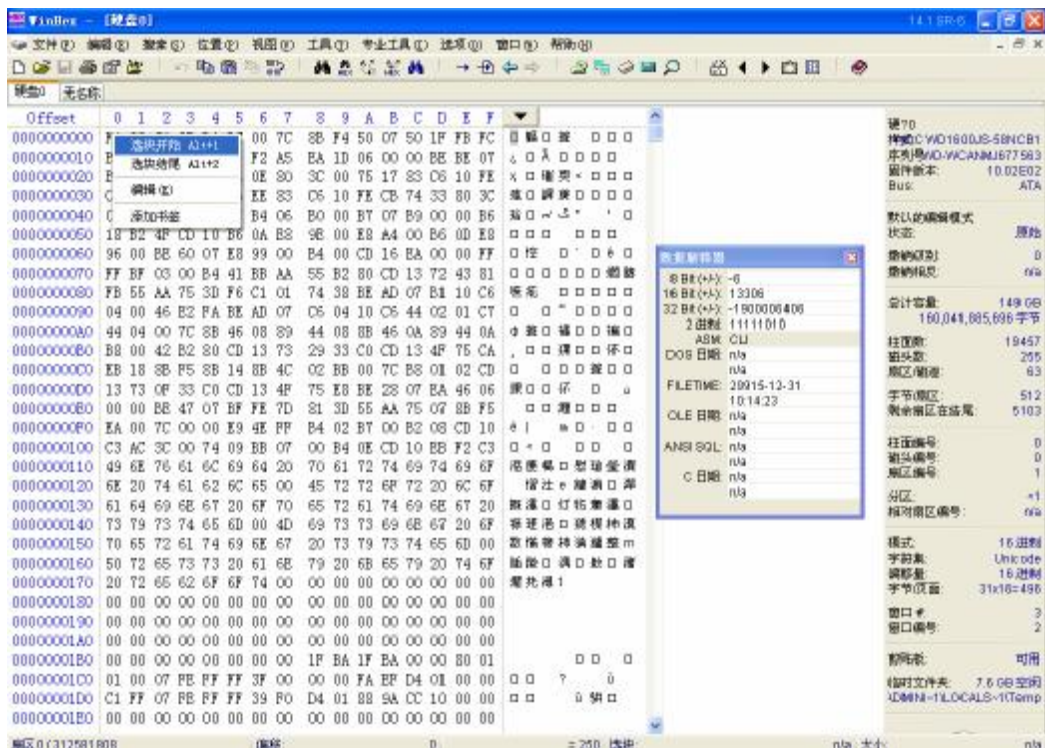


此外定义选块控制台还融入了不少技巧性的功能，比如可以直接定位当前位置、文件开始、文件中间、文件结尾等，不需要用户花费精力获取偏移量值。

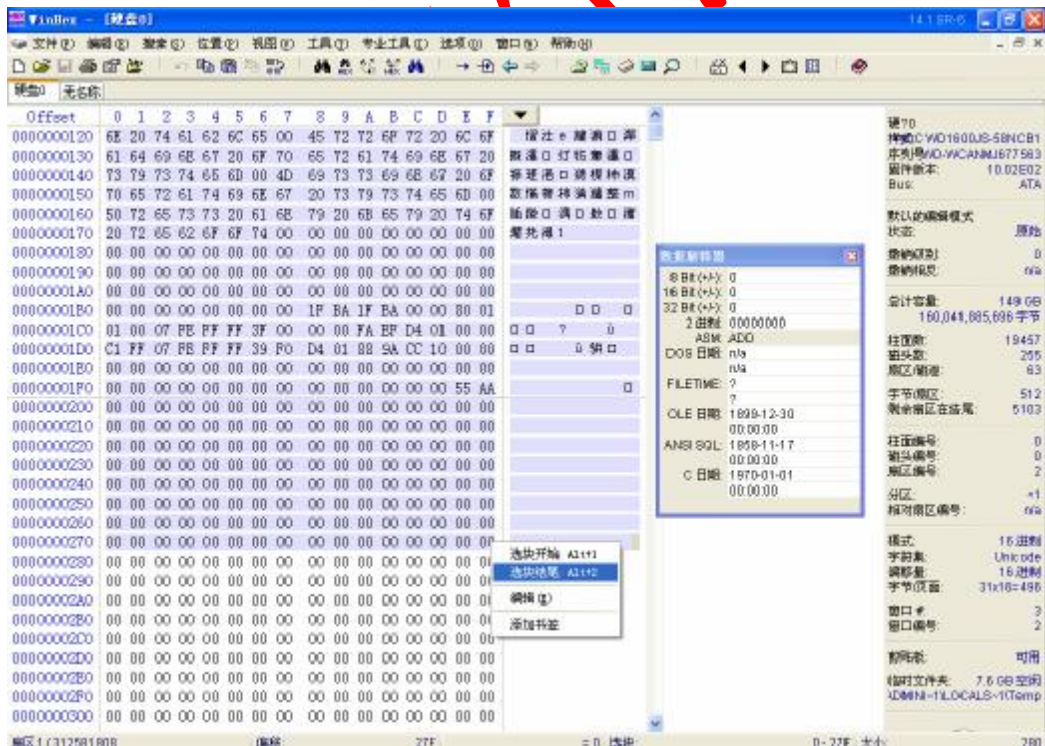


第三种方法是开始结尾法，点击右键直接在选块起始点击“选块开始”项，再跳转到终止处

点击“选块结尾”项，那么之间的范围将被标记。如图所示：

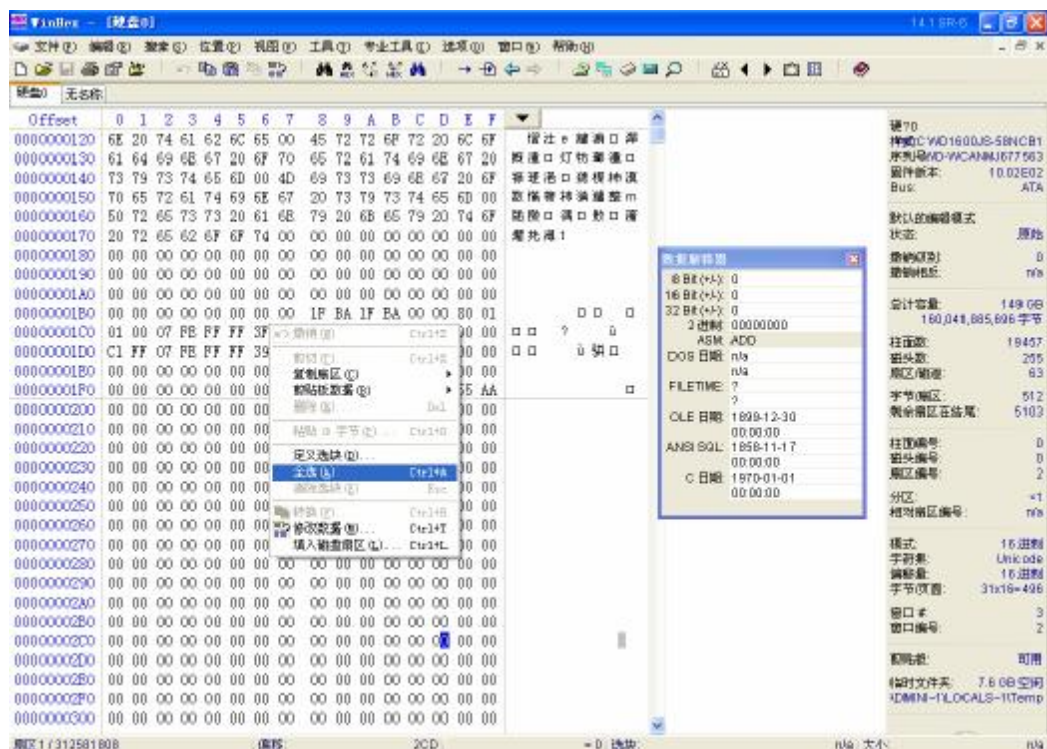


(任意位置点击“选块开始”)

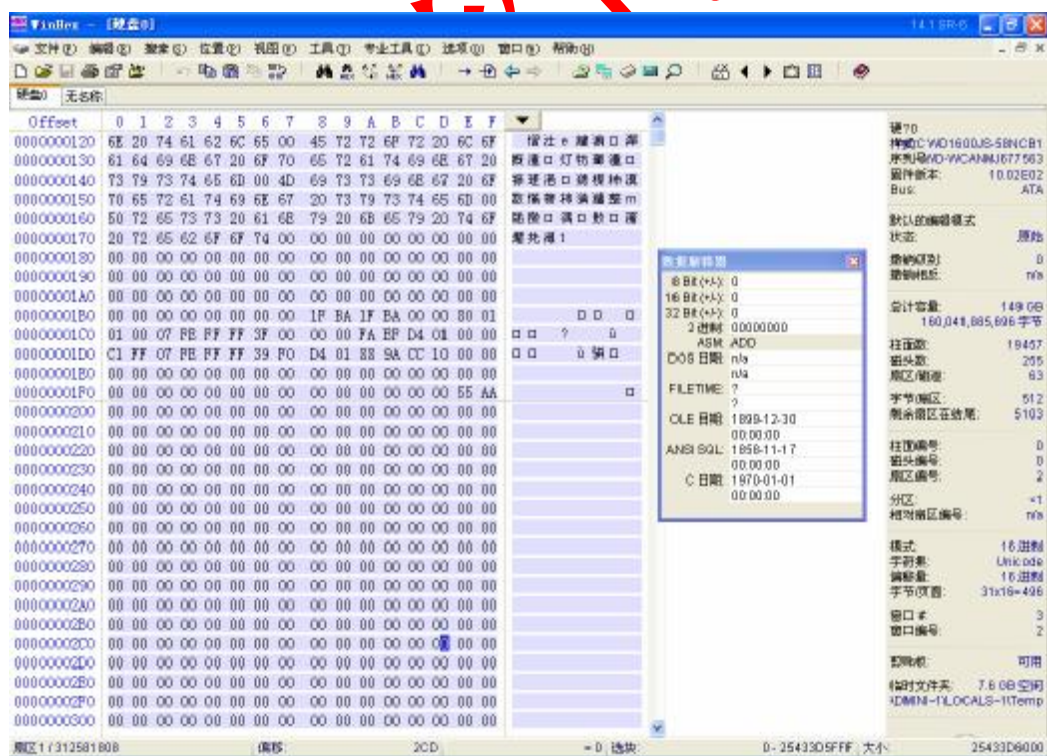


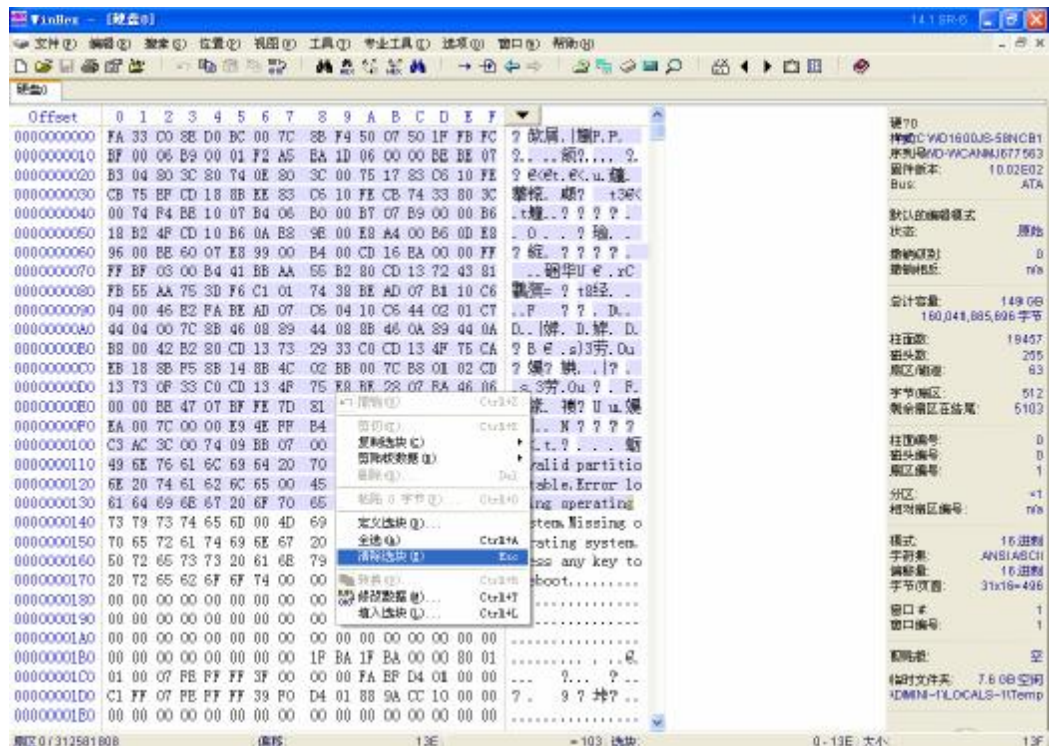
(可以看到选块被标记)

“全选”子项可以标记该操作对象所有数据区，可以用于文件拼接、模块组合等，也可以为大范围搜索定义尺度。

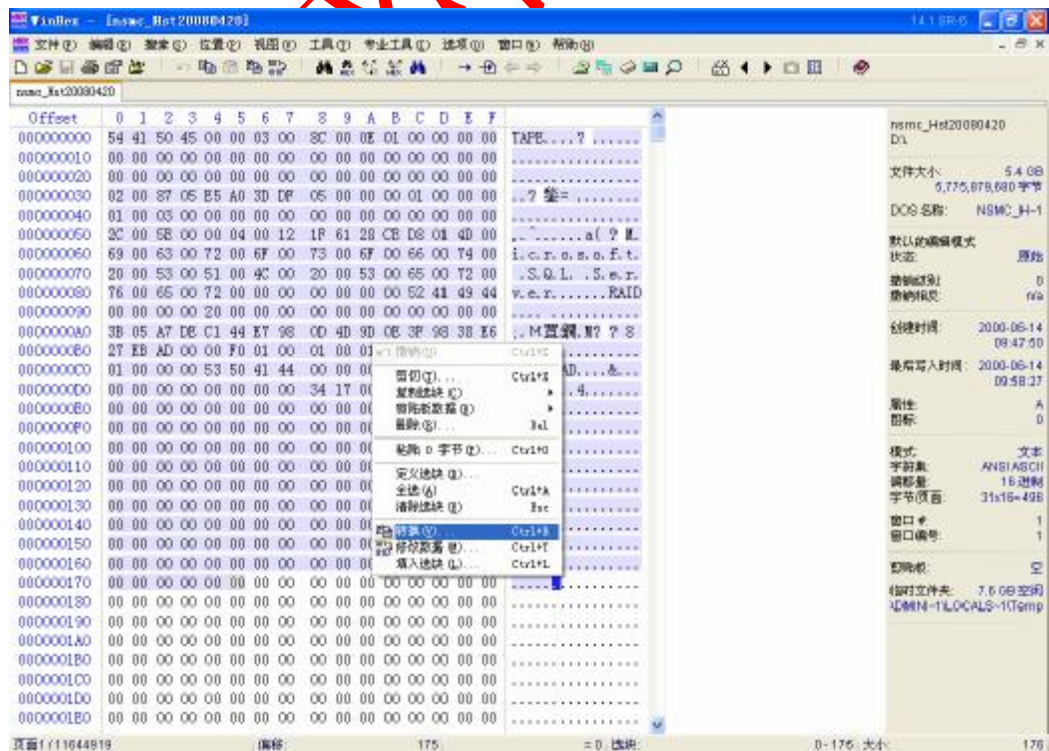


很多时候，“全选”子项可以减少 WINHEX 对用户指令的误解。毕竟“全选”也是指定了范围，比“不选定”要严谨一些。





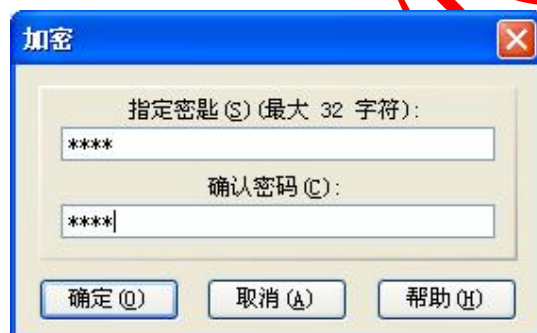
转换子项在数据恢复中应用较少，但其强大功能是不提不提的，它不仅可以实现编码互转、进制互转、文件加密，还可以程序化地设置密钥，可以解析 NTFS 压缩数据流。它牵扯到文件编码学、文件系统学、密码学等众多学科，是 WINHEX 使用中技术含量较高的，当然与用户的知识水平也是密不可分的。比如我们要实现一段数据的加密，以 AES 算法为密钥基础，如图所示：



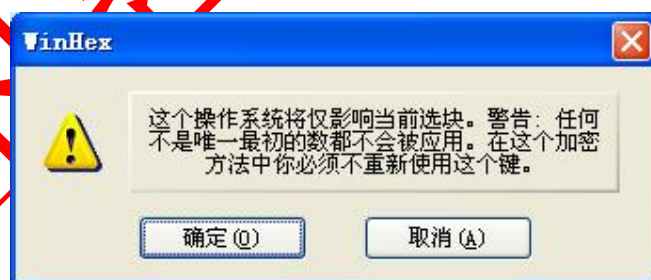
(选块任意范围，点击右键转换子项)



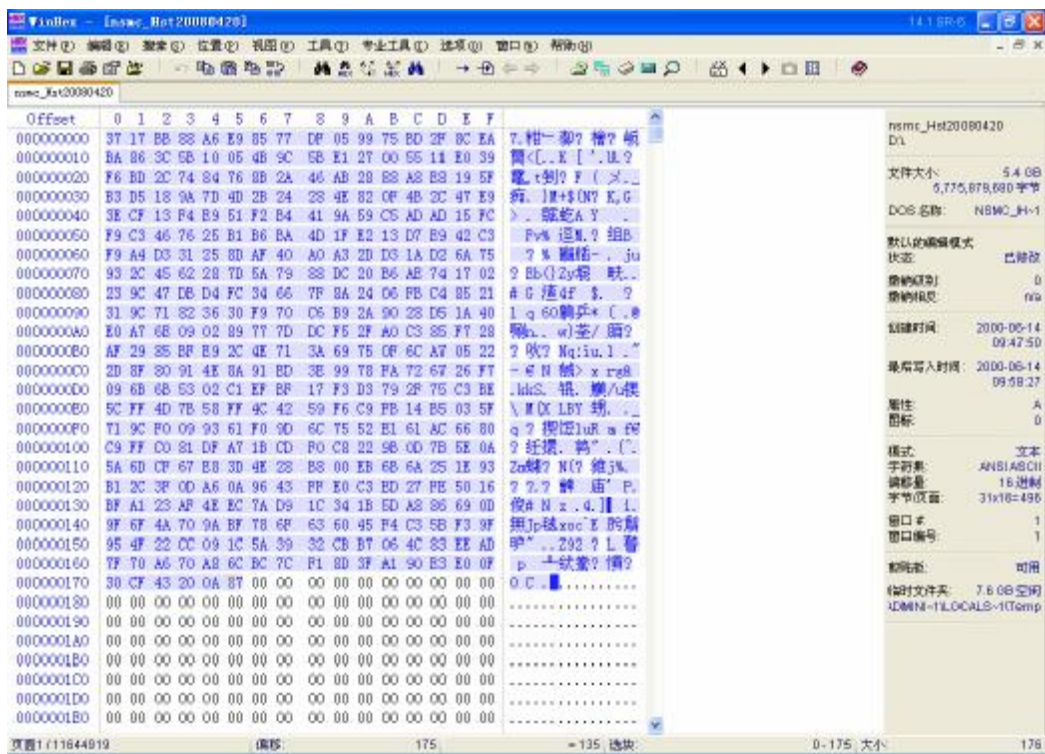
(出现转换选块控制台，选择加密 AES)



(指定一个密钥，然后重复输入加以确认)

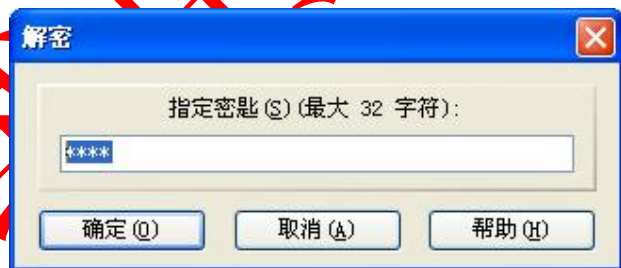


(出现加密警告，点击确定)

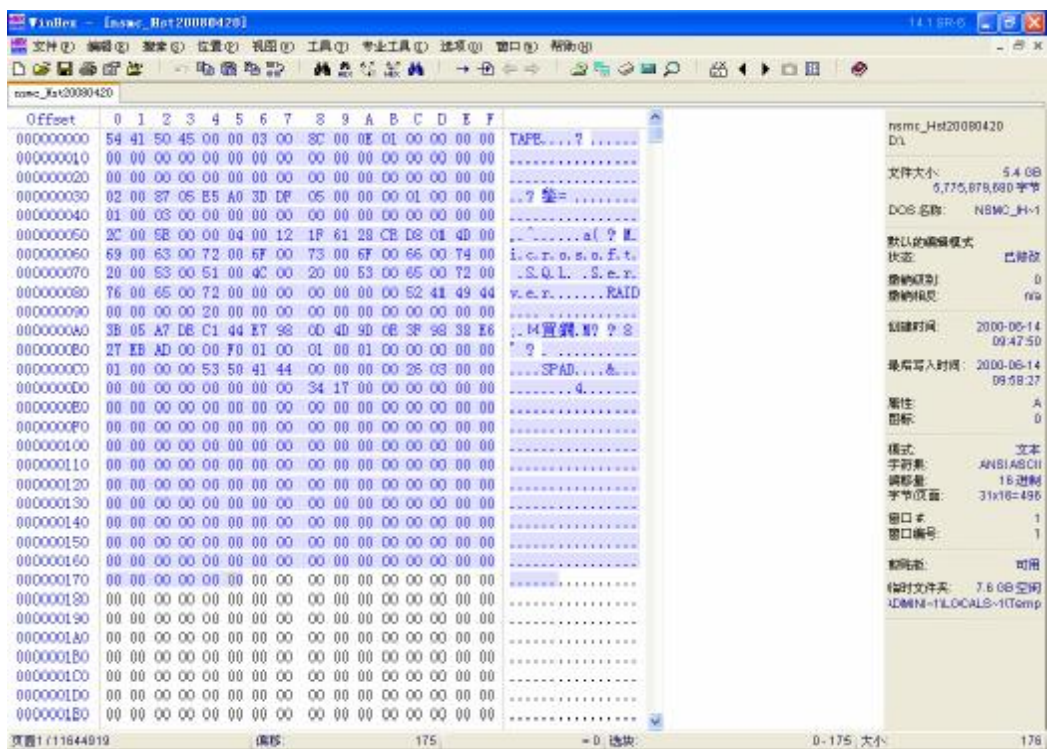


(数据被加密, 可以看到选块区已经完全改变形态)

解密过程也是一样, 第一步选块被加密的区域, 点击右键转换选项, 选择“解密 AES”出现下图:



输入密钥后数据将恢复原貌, 这里注意选块一定要准确, 不要把非加密区选中。



操作上都一样，总之 WINHEX 在不依赖第三方应用程序的前提下，可以按照用户意志，多形态地改变数据原貌，的确难能可贵。

转换子项操作范围：

ANSI ASCII, IBM ASCII (two different ASCII character sets)

EBCDIC (an IBM mainframe character set)

Lowercase/uppercase characters (ANSI ASCII)

Binary* (raw data)

Hex ASCII* (hexadecimal representation of raw data as ASCII text)

Intel Hex* (=Extended Intellec; hex ASCII data in a special format, incl. checksums etc.)

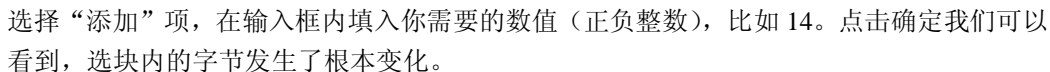
Motorola S* (=Extended Exorcisor; ditto)

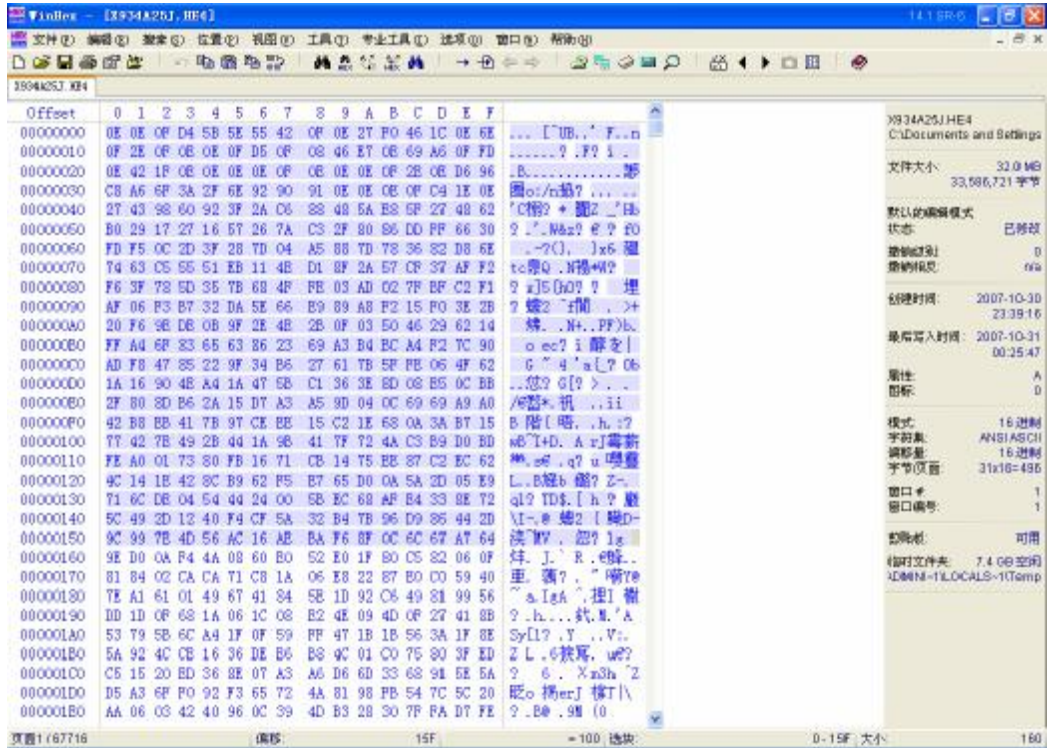
Base64*

UUCode*

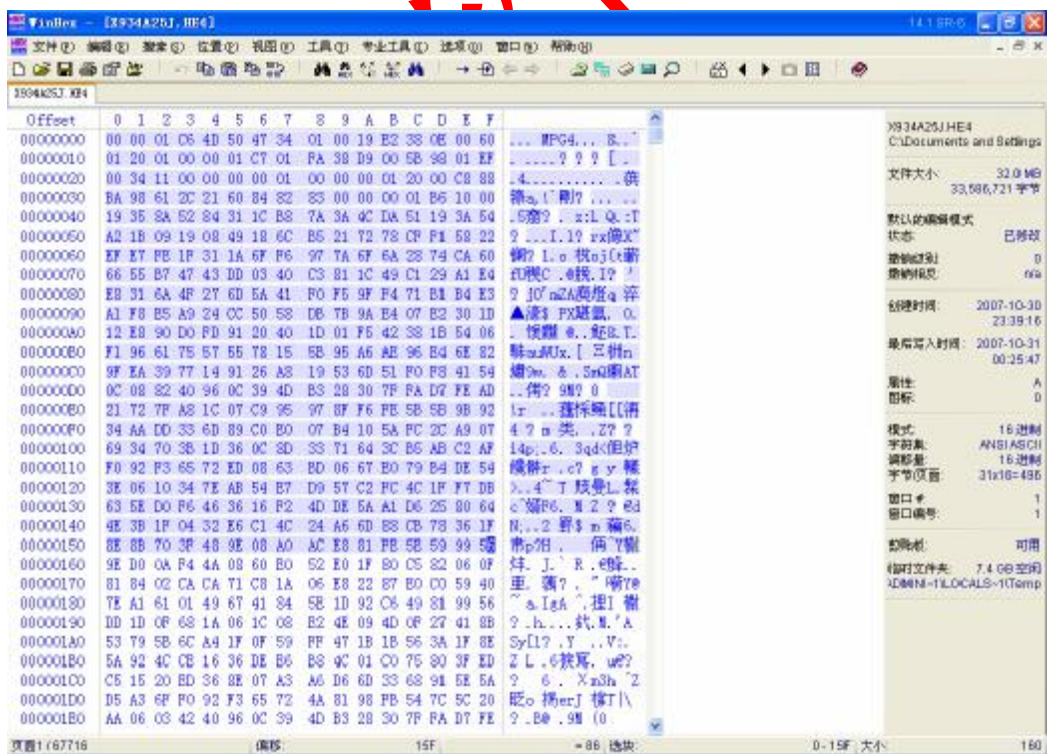
修改数据子项可以改变数据的排列规律，应用到了逻辑代数学的许多知识，具体实现的功能有：给单个或批量字节做指定加数（整数，可以是正负值或是 16 进制数值）的加法，给单个或批量字节做反转位（0-255 元素集合内的补集运算）、16 位字节交换（每两个字节左右交换位置）、32 位字节交换（每四个字节左右交换位置）、XOR 运算、OR 运算、AND 运算、循环左移一位运算、循环右移一位运算、位移运算、ROT13 运算（字母对应位编码）、左旋圆运算。下面我们来详细讲解操作和算法原理。

加数运算是整个数据修改算法中比较简单的，仅仅是给每个字节加上相同的数字，从而彻底改变数据全貌，当然如果再取负值数据还可以恢复原貌。如图所示：





此时我们在控制台中输入-14，看看会有什么变化：

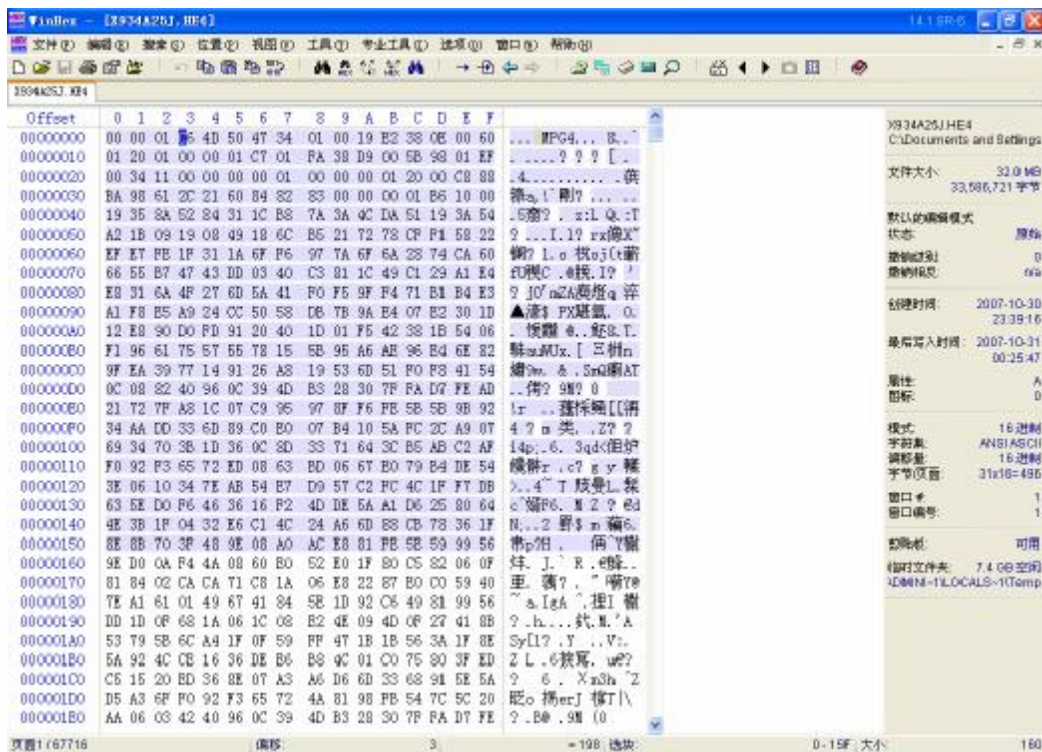


数据恢复了原貌，我们可以手动计算验证：

就拿偏移量为3的字节C6举例，它加上14（16进制为E）正好是十六进制值D4，从图中我们可以看出是正确的。-14相当于为改变后的字节统一减去14，数据当然就恢复原样了。

我们还可以选择添加 16 进制值，在后面选项框打勾就行，操作同上。下面还可以限定整数范围和类型。

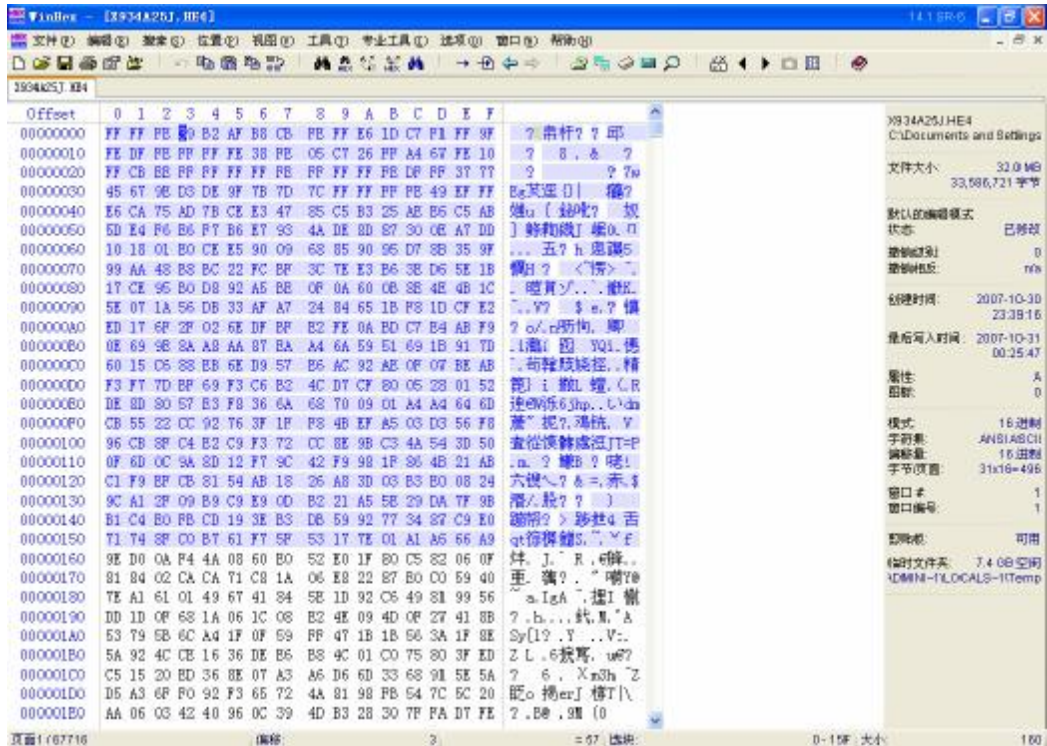
反转位是一种有意思的集合运算，这个集合的元素只能在（00--FF）之间，这里 WINHEX 用了补集法。如图所示：



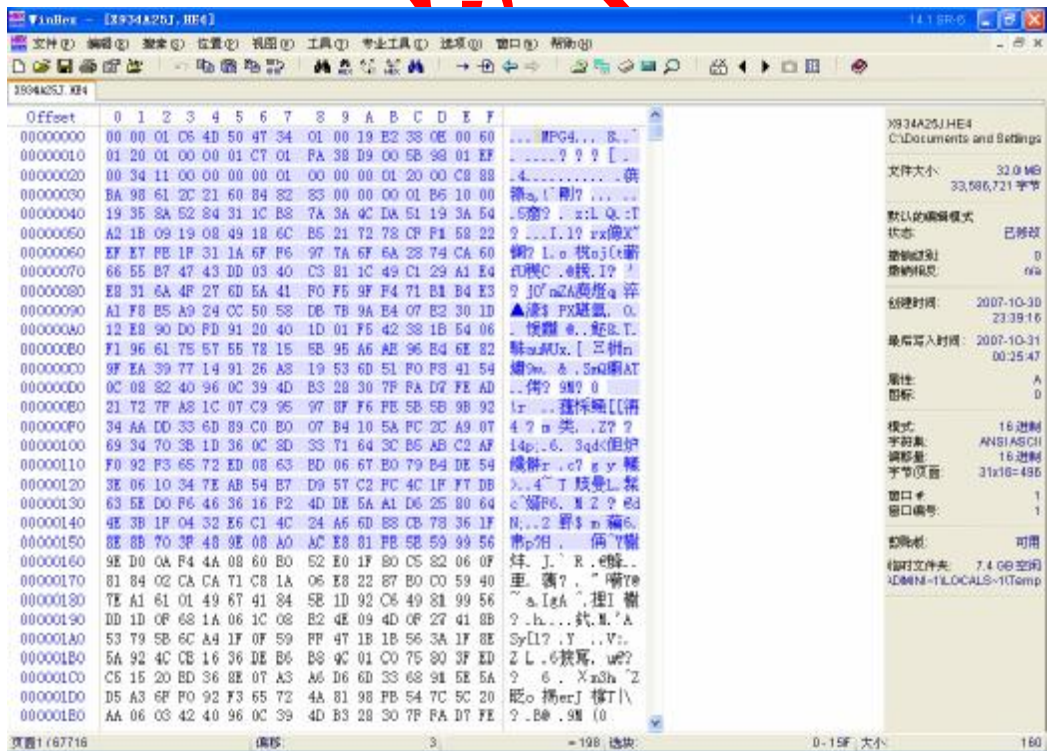
选块后，在控制台中选择反转位一项。



数据发生改变，改变后的值相当于 FF 减去当前字节值：



可以看到 00 变成了 FF 减去 00 为 FF、4D 变成了 FF 减去 4D 为 B2、依此类推。当然我们再次使用“反转位”数据便会恢复。



32 位字节交换仅仅改变了字节的位置，我们可以看出：

000001C64D504734010019E2380E0060012001000001C701FA38D9005B9801EF003411000000
0001000000012000C888BA98612C216084828300000001B6100019358A5284311CB87A3A4C
DA51193A54A21B09190849186CB5217278CFF15822EFE7FE1F311A6FF6977A6F6A2874CA
606655B74743DD0340C3811C49C129A1E4E8316A4F276D5A41F0F59FF471B1B4E3A1F8E5
A924CC5058DB7B9AE407E2301D12E890D0FD9120401D01F542381B5406F19661755755781
55B95A6AE96E46E829FEA3977149126A819536D51F0F841540C088240960C394DB328307F
FAD7FEAD21727FA81C07C995978FF6FE5B5B9B9234AADD336D89C0E007B4105AFC2CA
9076934703B1D360C8D3371643CB5ABC2AFF092F36572ED0863BD0667E079B4DE543E061
0347EAB54E7D957C2FC4C1FF7DB635ED0F6463616F24DDE5AA1D62580644E3B1F0432E6
C14C24A66D88CB78361F8E8B703F489E08A0ACE881FE5E599956

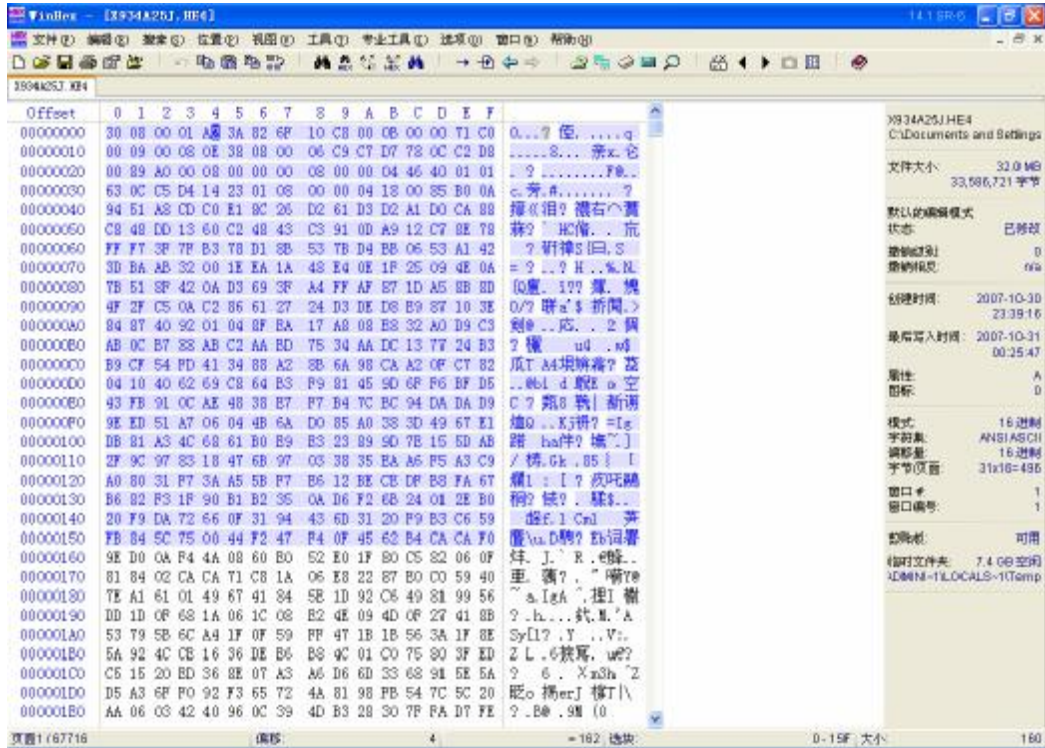
(交换前的 HEX 视图)

C60100003447504DE219000160000E380001200101C7010000D938FAEE01985B001134000100
00000100000088C800202C6198BA82846021000000830010B601528A3519B81C3184DA4C3A
7A543A195119091BA26C184908787221B52258F1CF1FFEE7EFF66F1A316A6F7A9760CA74
2847B755664003DD43491C81C3E4A129C14F6A31E8415A6D27F49FF5F0E3B4B171A9E5F8
A15850CC24E49A7BDB1D30E207D090E812402091FD42F5011D06541B38756196F11578555
7AEA6955B826EE4967739EA9FA8269114516D53195441F8F04082080C4D390C967F3028B3
ADFD7FAA87F722195C9071CFEF68F97929B5B5B3DDAA34E0C0896D5A10B40707A92
CFC3B7034698D0C361D3C647133AFC2ABB565F392F06308ED72E06706BD54DEB4793410
063EE754AB7EFCC257D9DBF71F4CF6D05E63F2163646A15ADE4D648025D6041F3B4E4C
C1E632886DA6241F3678CB3F708B8EA0089E48FE81E8AC5699595E

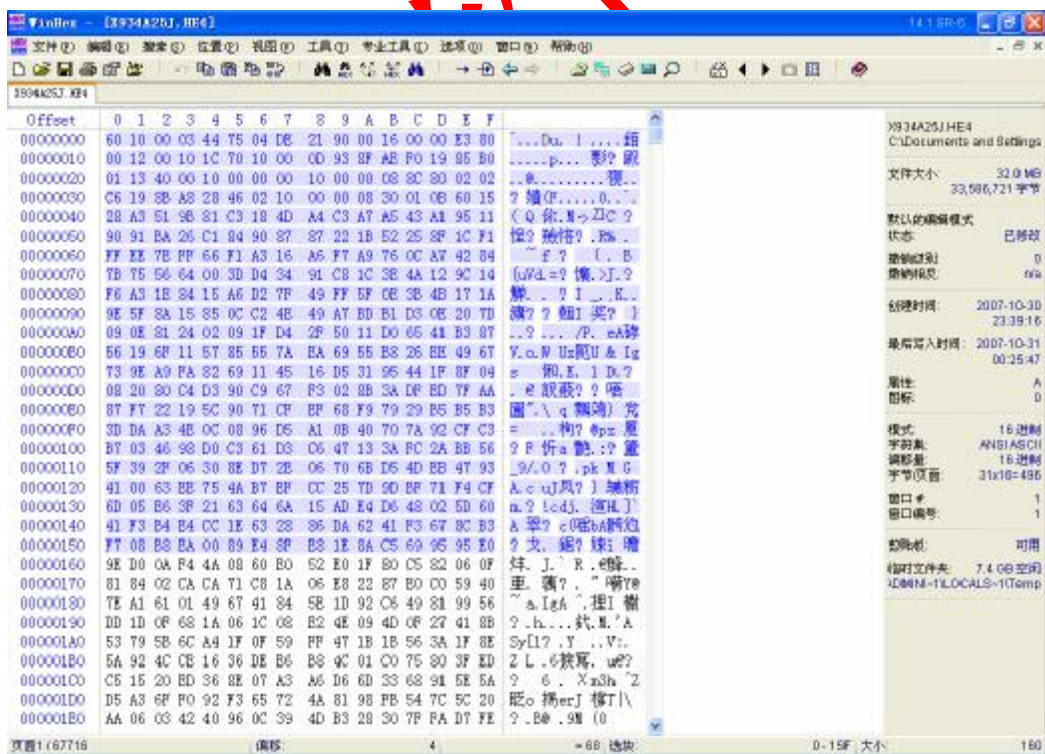
(交换后的 HEX 视图)

000001C6 变成了 C6010000、4D504734 变成了 3447504D、每四个字节左右交换。

循环左右移算法常常应用在电路控制方面，我们先要将 16 进制编码转化为 2 进制，然后将数位移动，高低位补零。如图所示：



选择“按一位左移”数据变为：

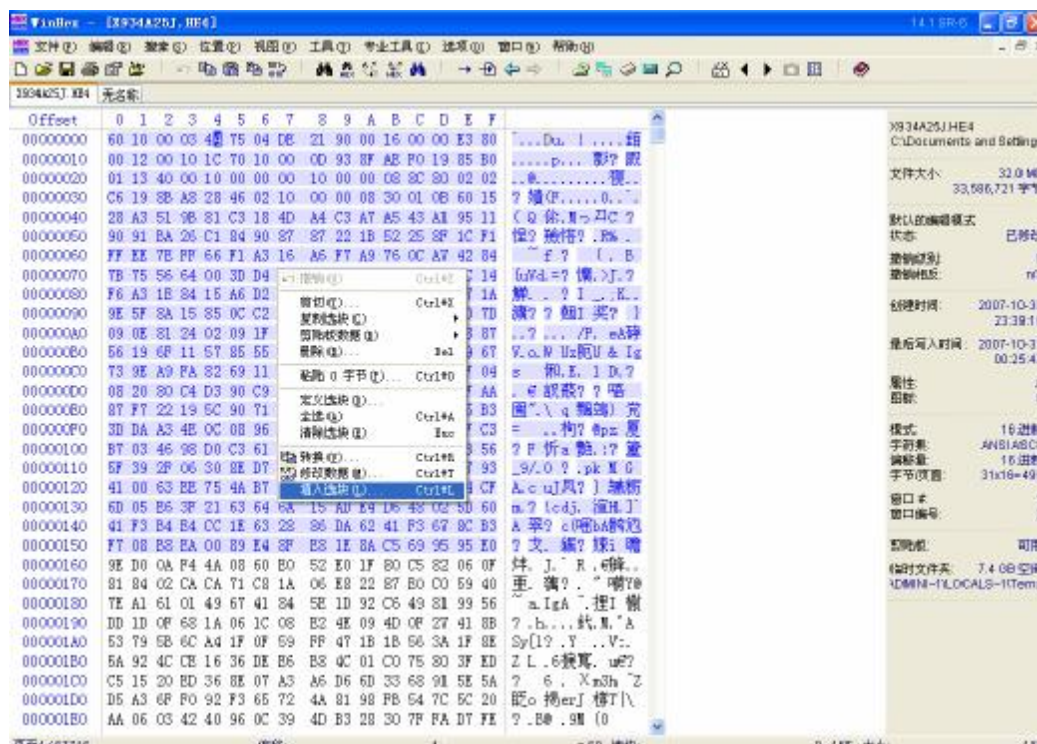


首字节 30 转化为二进制为 110000，左移一位是 1100000，十六进制为 60。

ROT13 编码是一种简单的编码，它把字母分成前后两组，每组 13 个，编码和解码的算法相

同，仅仅交换字母的这两个部分，即：[a..m] --> [n..z] 和 [n..z] --> [a..m] (--> Caesar 13)。ROT13 用简易的手段使得信件不能直接被识别和阅读，也不会被搜索匹配程序用通常的方法直接找到。WINHEX 中 ROT13 仅仅是一种数据隐藏手段。XOR 运算相信大家都很熟悉了，WINHEX 数据修改中要求用户指定一个值分别与字节进行异或运算。此外还有其它许多运算，这里不再详细介绍，有兴趣的用户可以自行研究。

填入选块子项主要应用于数据销毁领域，我们可以整盘、整文件、或指定区域进行无意义字节填充，从而彻底覆盖数据。如图所示：

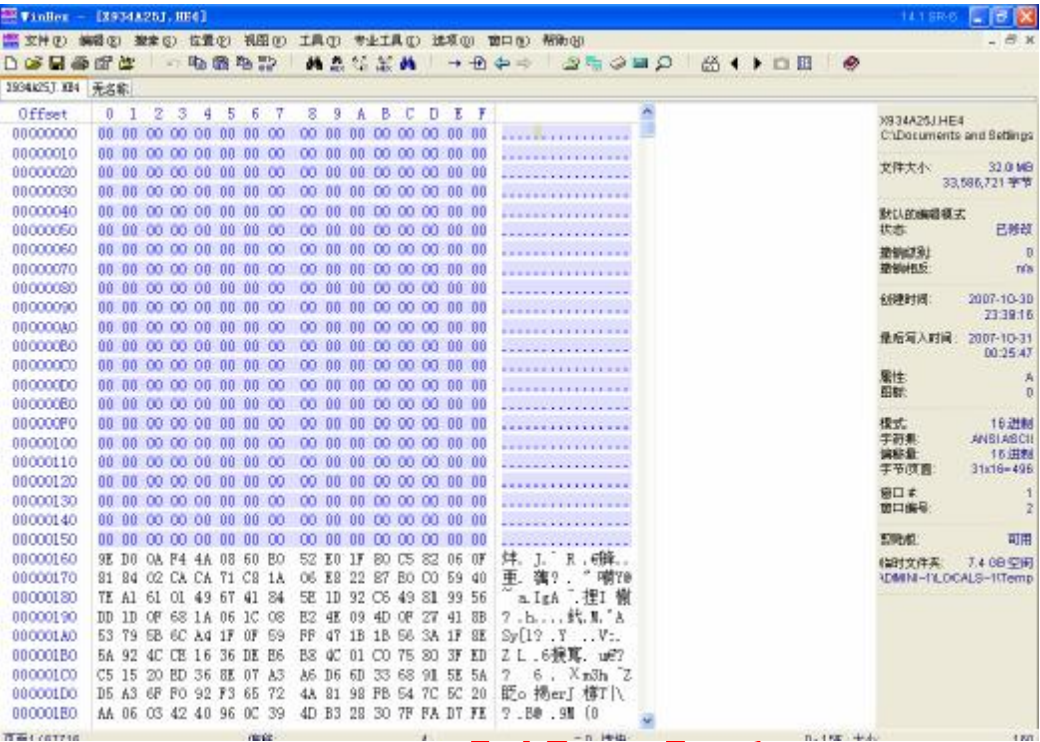


出现“填入选块”控制台，这里有四种选项：用 16 进制数值填充、用随机字节填充、模拟加密数据、加密安全性伪随机。

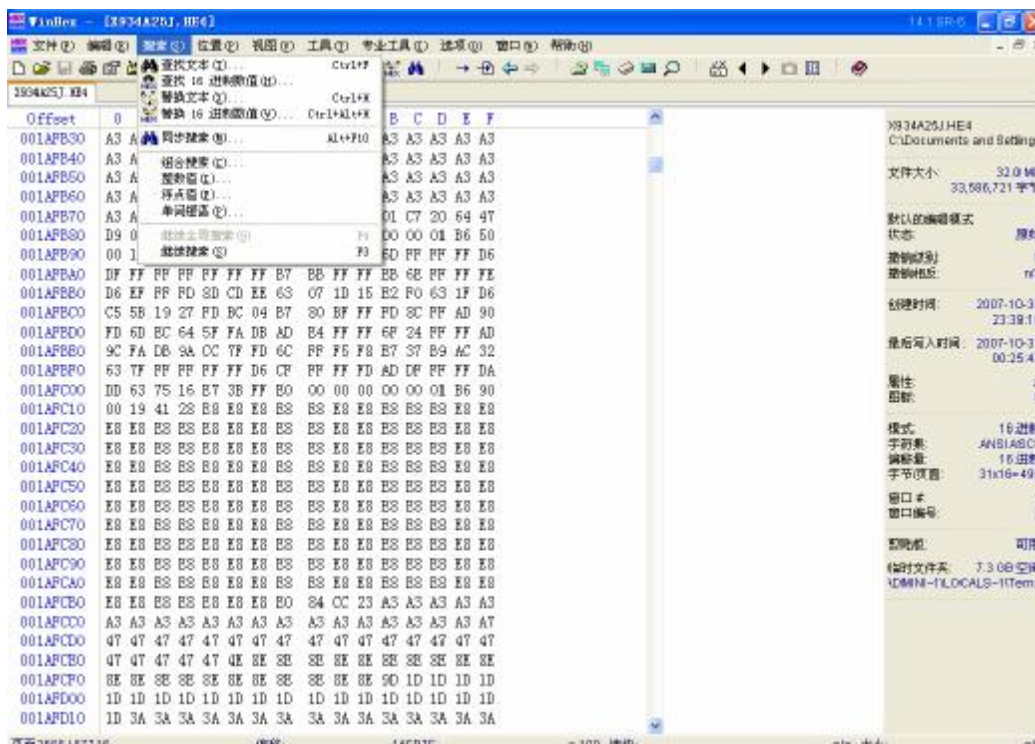


这四种方法效果大同小异，如果要满足数据销毁标准，必须重复多次。这里我们选择“用 16 进制数值 00 填充”。可以看到数据被零字节覆盖。右边点“添加”可以设定多套方案，

然后批处理执行，我们就可以

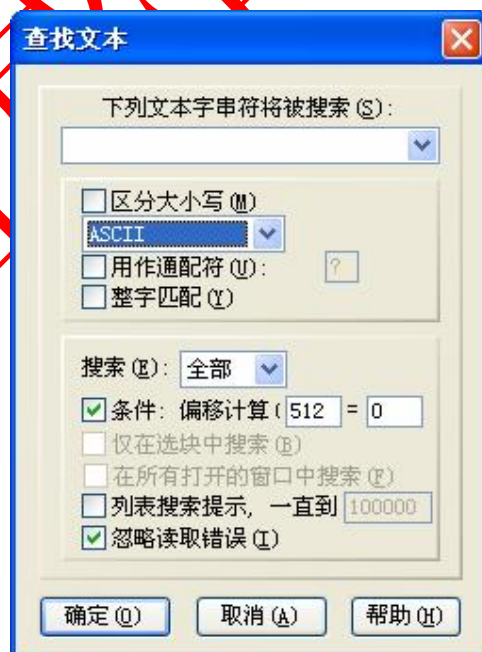


搜索主菜单是数据恢复较为常用的大类之一。我们在工作中，接触到的往往是文件系统底层，没有直观简易的标志去定位各种我们需要的参数，我们只有在牢牢记住特征编码后，利用磁盘编辑器去寻找，而 WINHEX 搜索便是此领域的佼佼者。“搜索”包含的子项有：查找文本、查找 16 进制数值、替换文本、替换 16 进制数值、同步搜索、组合搜索、搜索整数、搜索浮点值、搜索单词短语、继续全局搜索、继续搜索。我们知道，文件系统对内管理是分层分级的，要定位并访问一个文件，需要经过一级一级大量精确的计算，而掉电、病毒、误操作、磁盘物理故障等众多原因都是破坏这种组合计算的罪魁祸首。比如某个分区的文件系统要完成引导，它的最根本前提是与之相关的分区表链完好无损，当分区表链严重缺陷时，甚至连 WINHEX 都无法直接顺利的访问该分区了，此时我们可能采用的方法就是，搜索 DBR 的某些特征值定位该分区起始扇区，然后虚拟加载该分区。我们在重组 RAID 时，需要严格分析文件系统上的很多记录，这也需要我们利用“搜索”去寻找它们。搜索本身就是一门技术、一门学科，相信随着时间的推移，用户会总结出更多的经验技巧，受益终生。下面我们将重点讲解与数据恢复关系紧密的搜索技术。



(搜索主菜单)

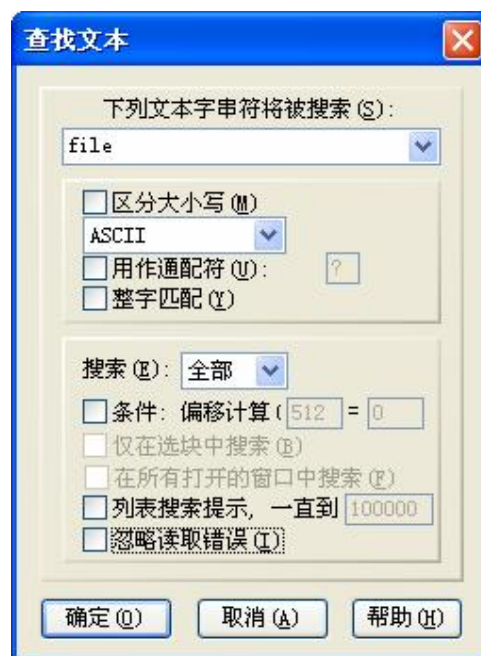
查找文本子项的主要作用就是搜索、定位操作对象中存在的任意字符串。很多视频文件、MSSQL 备份文件，或者 MFT 记录头等，都是以某种字符串作为起始，此时我们只要查找这些字符串就可以在字节的海洋中轻易寻访它们的踪迹。当然字符集有多种，不过不用担心，该子项支持的字符集包括 ASCII 和 UNICODE 大类，涵盖了 90% 以上的应用领域。如图所示：



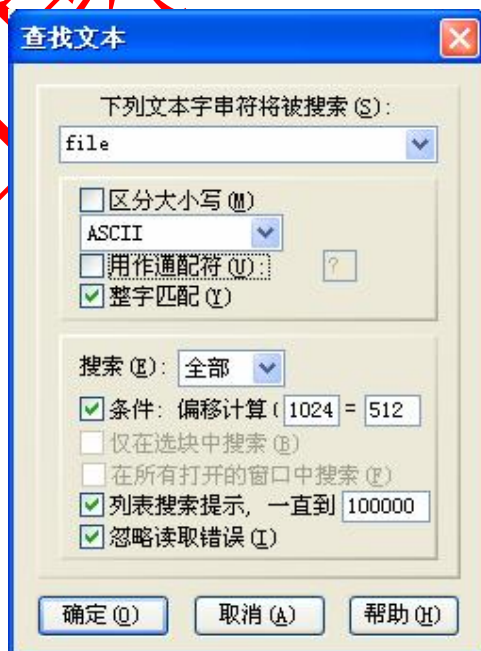
最上方是一个文本填写框，用来输入用户想要搜索的字符串。下方都是为搜索任务量身定做的各种条件，用户可以对字符串字母的大小写提出更为准确和特别的要求，可以自由选择两大大字

符编码类型，可以在搜索表达式中加入一个通配符、可以要求整字匹配（精度搜索）、可以选择搜索方向或全局搜索、可以在指定范围内为搜索对象确定方位（偏移量）、可以选择只在选块区搜索、可以在所有打开的项目中进行搜索、可以给出并保存搜索列表、可以忽略读取错误（比如跳过坏扇区）。举个例子：我们要在当前硬盘搜索 MFT 记录头，我们已知该记录头特征是 ASCII 码 FILE。

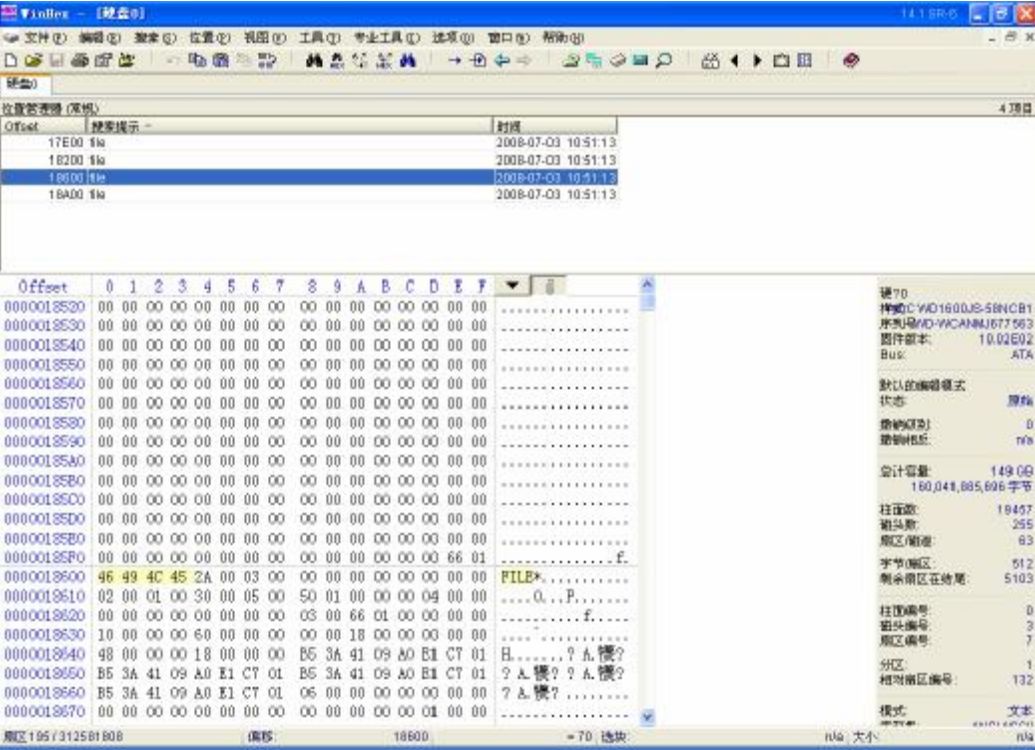
首先在文本填写框内填入 file 这里大小写不作要求：



下面我们设定条件：区分大小写可以忽略，选择 ASCII、由于对象单一且很清晰故不采用通配符，FILE 是固定单词，可以要求整字匹配。这里我们不妨做全局搜索，选择“全部”。该字符串一般在扇区起始，我们将偏移量设定为 1024=512 或 512=0（这里设定的越精确，搜索速度越快）。列表搜索和忽略读取错误都是很实用的功能，建议选上。



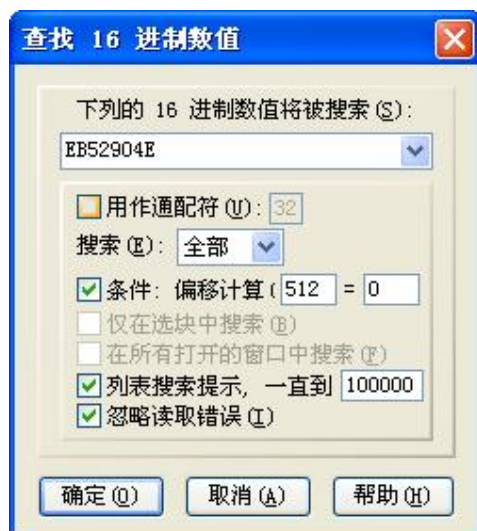
搜索结果会自动列表保存，我们可以看到，定位到的字串和 HEX 值被高亮标记。



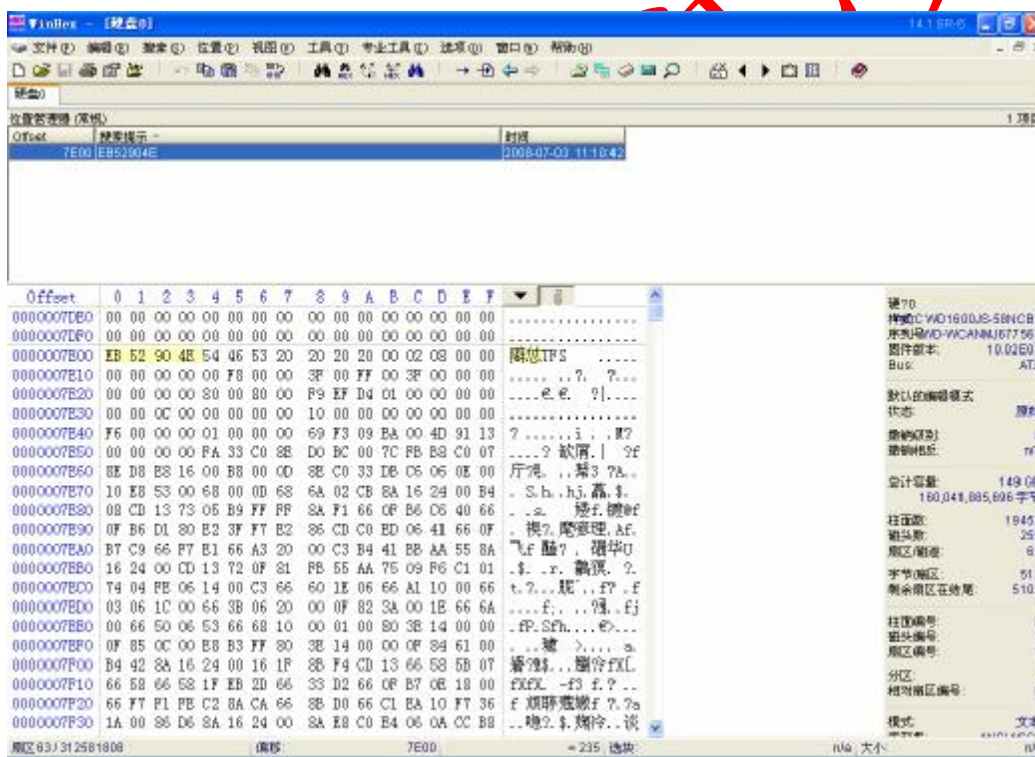
查找 16 进制数值子项与查找文本子项用法非常相似，如图所示：



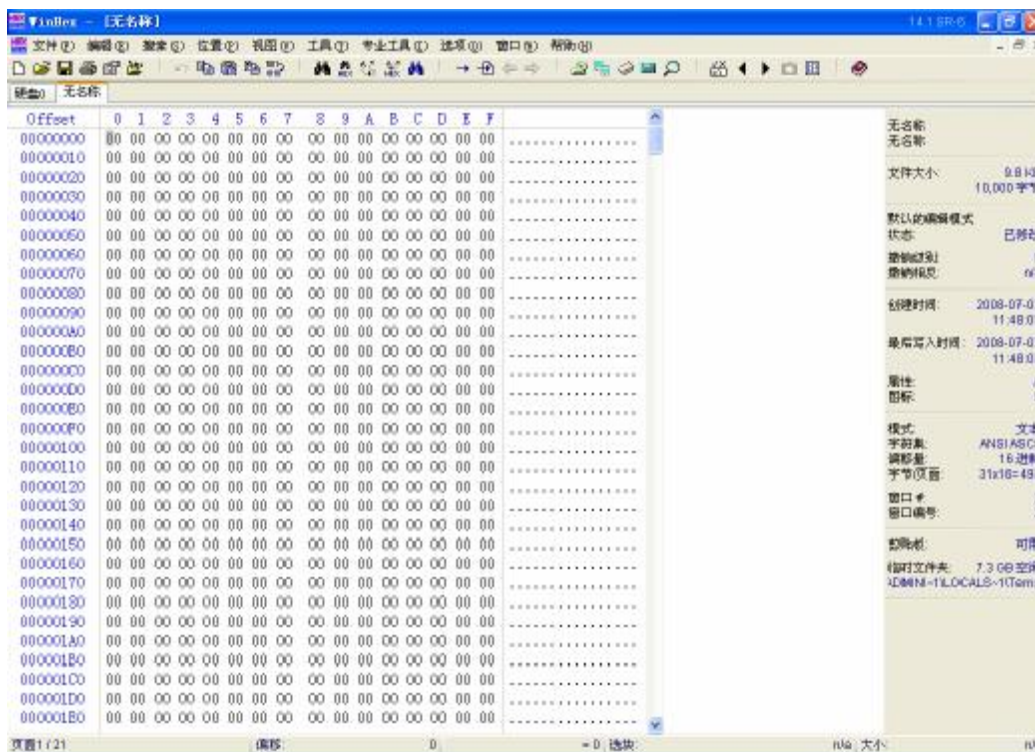
控制台界面几乎完全一样，只是文本框内被要求填入一组十六进制值。这里我们填入 NTFS 分区 DBR 起始 EB52904E。



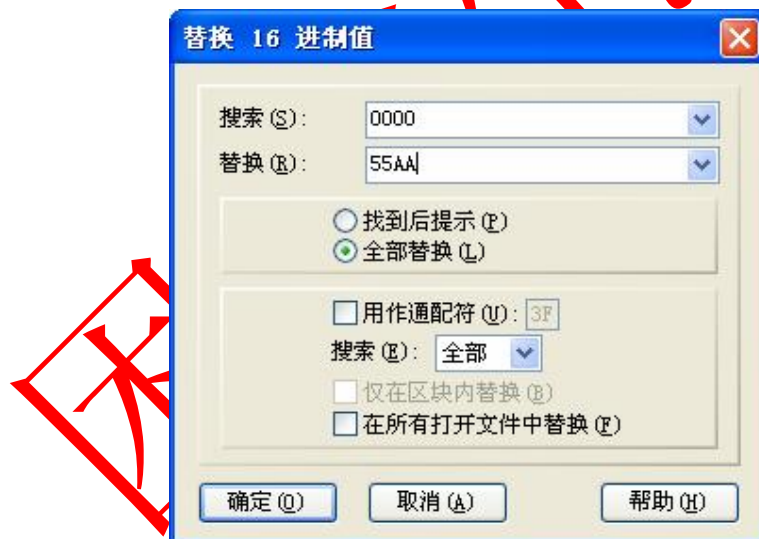
可以看到在 63 扇区，我们成功定位了一个分区的引导扇区。



此外如果需要定位文件系统所有特殊扇区，不妨查找分区有效结束标志 55AA。



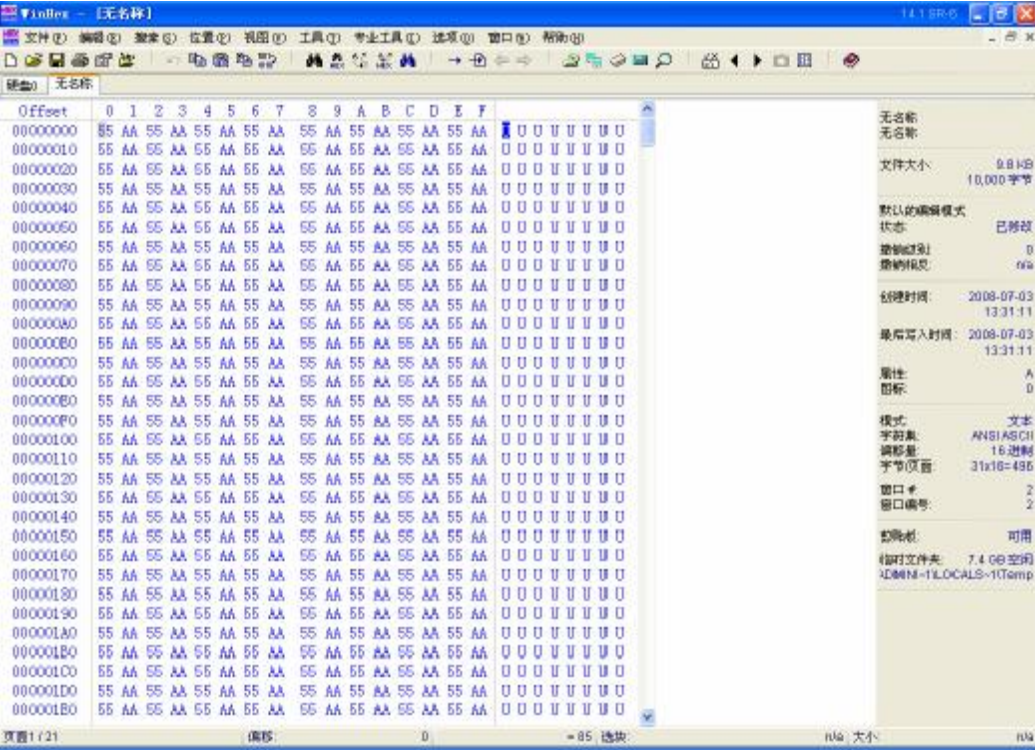
现在我们要将所有 0000 替换为 55AA：点击搜索主菜单替换 16 进制数值，出现以下控制台：



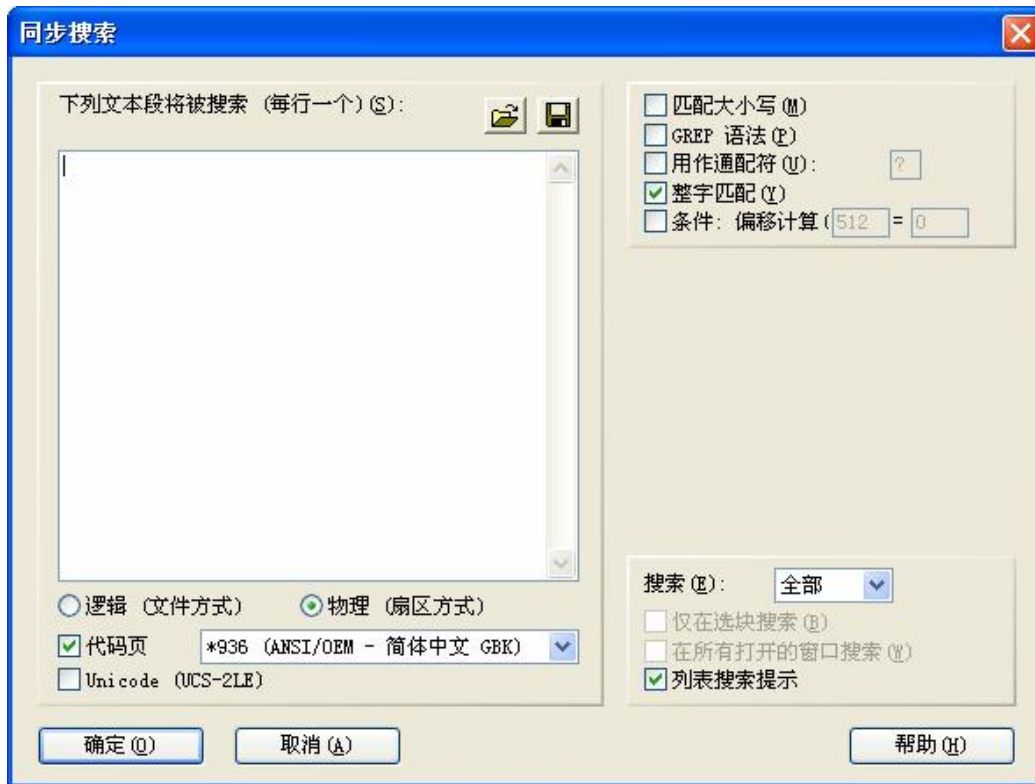
输入搜索值和替换值之后，选择全部替换。如果用户不想草率，还可以要求系统每次进行替换前先提示。我们也可以打开多个操作对象进行批量替换。设置好后点击确定。出现替换次数提示：



我们可以看到，0000 全部改变成 55AA：



同步搜索是集成度及智能化相对较高的子项，可以实现多编码字串的同时搜索，也就是说它可以同时完成多个搜索任务。注意同步搜索的对象目前仅限于文本。

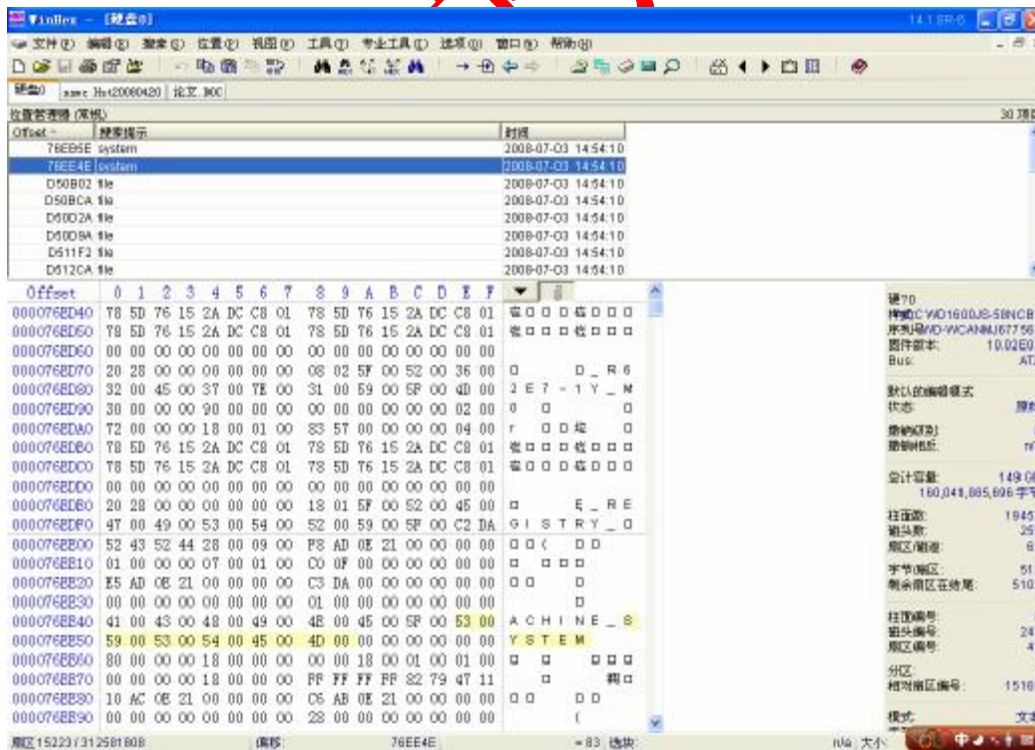


文本框用来输入字串，这里对格式有一定要求，如果要进行多任务搜索，每个任务必须占用独立的一行。此外我们还可以从外部导入文本文件来定义搜索内容，这里有逻辑搜索和物理搜索两种方式，逻辑搜索主要针对操作对象中的文件，搜索范围较小，速度很快。而物理搜索是字节级的逐个检查，主要针对物理磁盘。

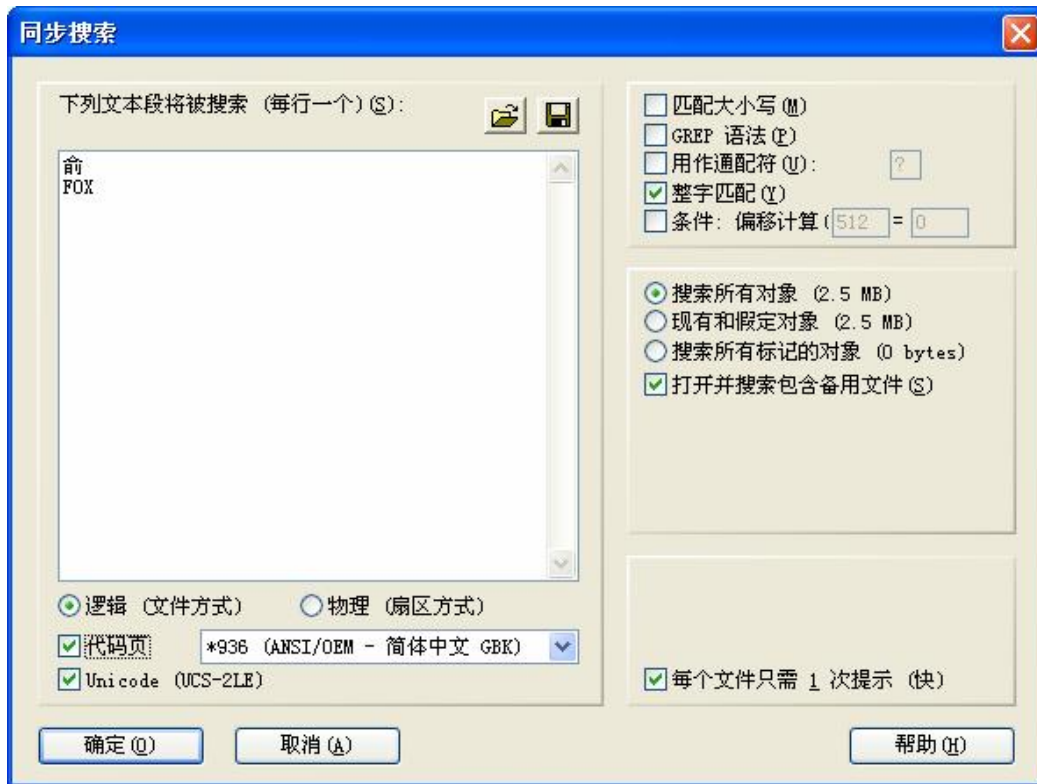
我们先来看物理方式，每行各输入一个单词：FILE 和 SYSTEM。搜索对象是当前硬盘。



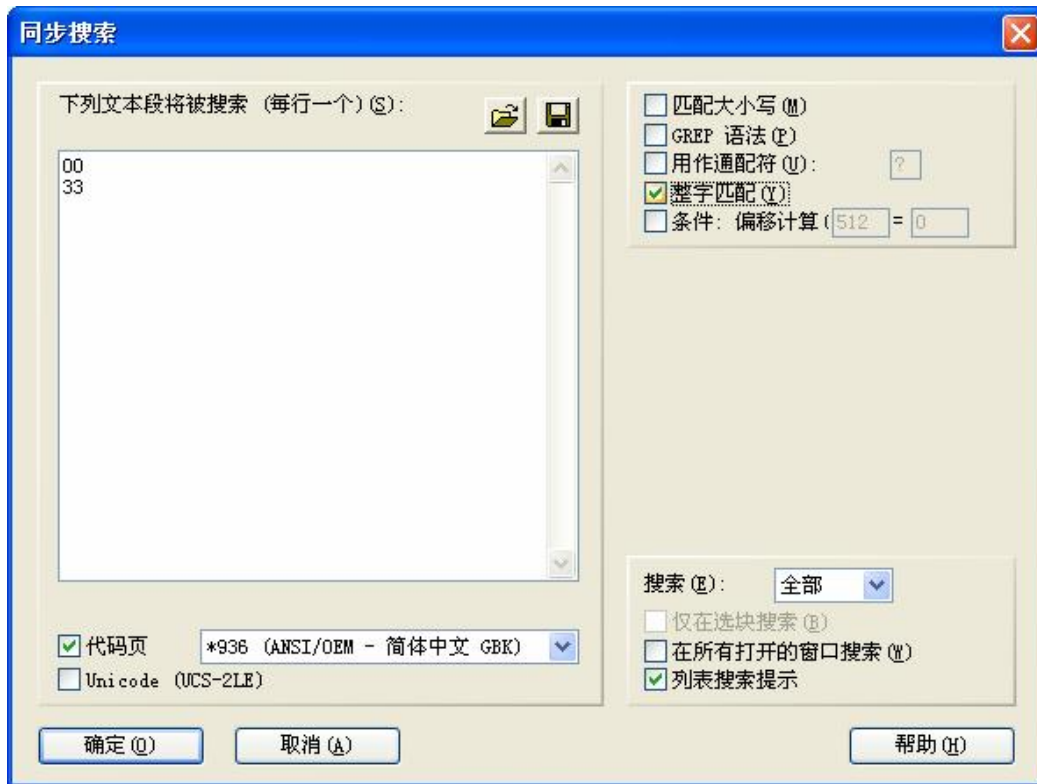
点击确定开始搜索，搜索结果以列表显示：



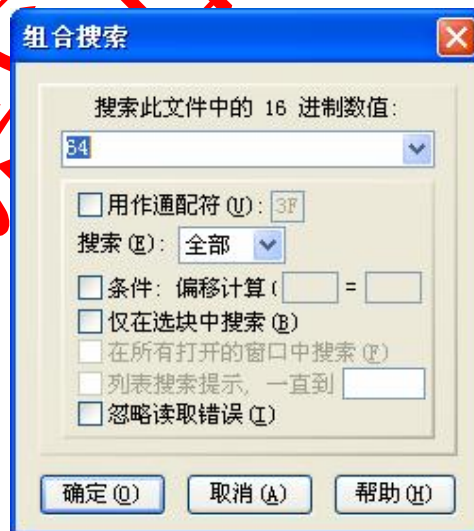
我们选择某分区进行逻辑方式搜索。因为只有分区是一个与文件个体相关的逻辑概念。



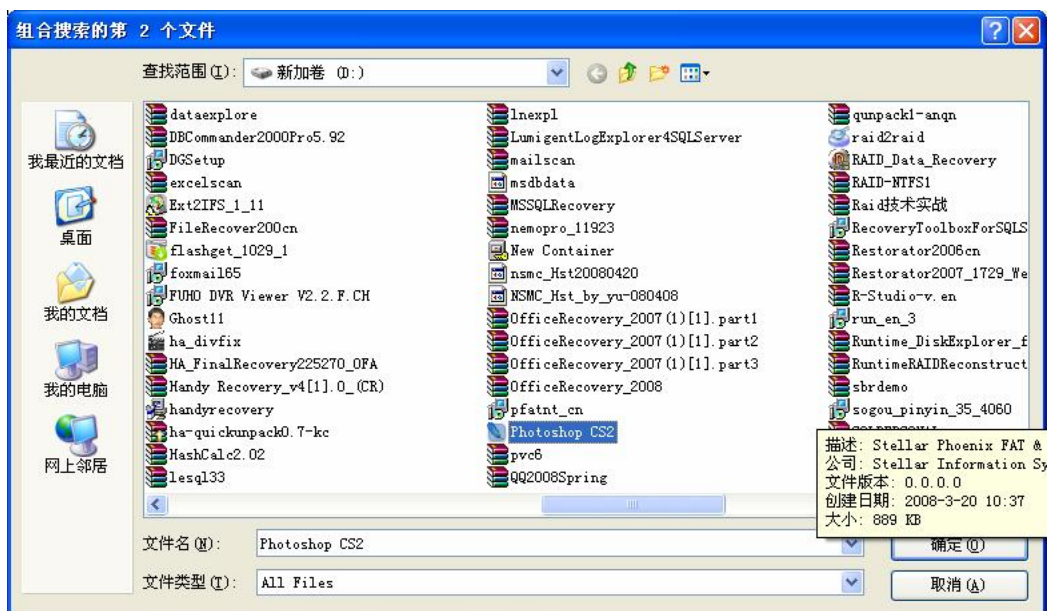
注意，这里提到了 GREP 语法，这是 UNIX 中的一个命令 用以在文件字符串中寻找或比对，或是当没有指定档案时，由标准输入充当参照，具体语法参照 UNIX 相关资料。右边有三个选项，默认是“搜索所有对象”就是对当前 WINHEX 目录浏览器所识别的所有文件进行搜索。“现有和假定对象”会摒弃那些操作系统不能直接调用的文件，如已删除或置疑状态的文件。“搜索所有标记对象”是一种特殊用法，前提是该文件被提前指定。



组合搜索条件非常苛刻，即搜索两个文件中相同位置的特定数据。此功能对分析文件的相似性帮助很大。这里我们先填入一个特定值 64，如图所示：



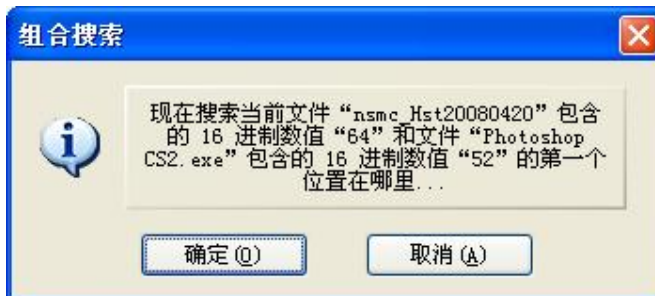
点击确定后要求指定一个“搭伴文件”。



现在大家明白为什么叫“组合搜索”了吧。点击确定后再次出现针对“搭伴文件”的控制台。



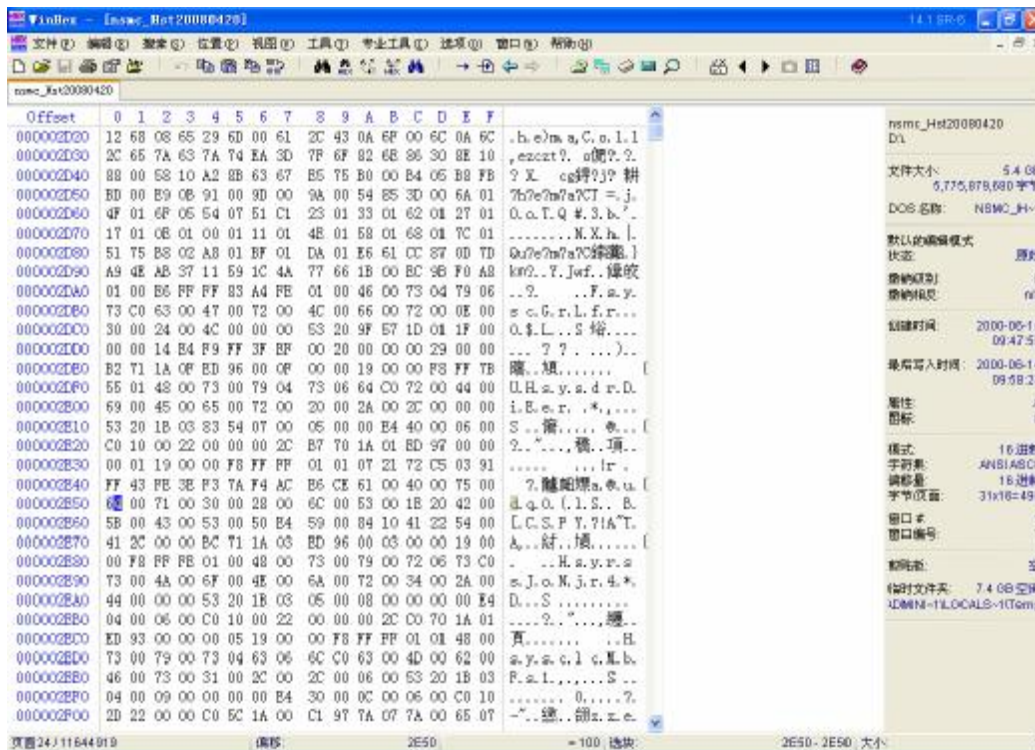
这里我们填入特定值 52，点击确定，出现最后确认界面，如果无误则确认。



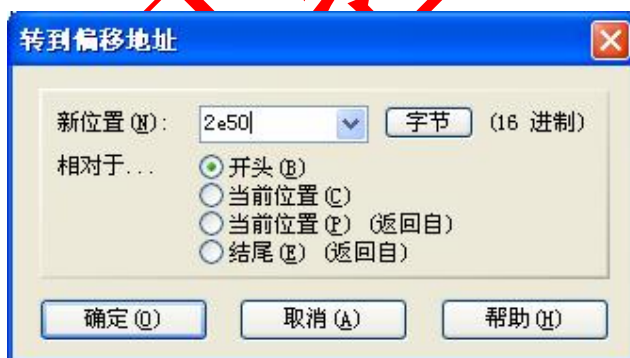
搜索完成，出现结果提示：



可以看到，第一个文件特定值“64”已经被成功定位，我们记下它的偏移量。



打开“搭伴文件”转到同样的偏移量。

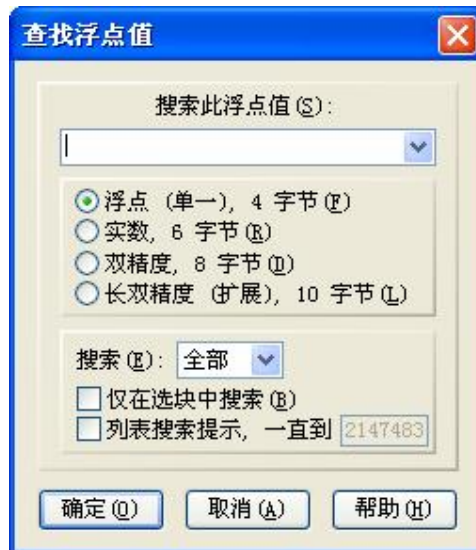


可以看到，“搭伴文件”的特定值“52”出现在了相同位置。

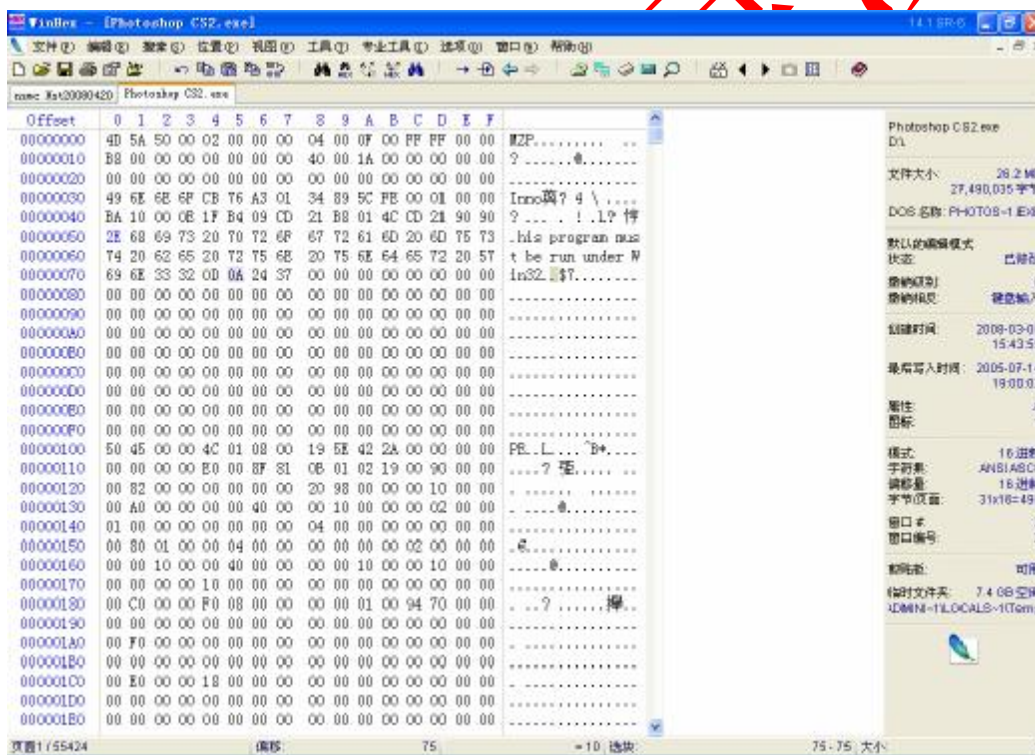
整数值子项和浮点值子项操作同上，其中浮点值搜索支持单一浮点、实数、双精度和扩展双精度，可以配合反汇编功能跟踪处理器运算。但它们在数据恢复中较为少用。



(整数值搜索控制台)



(浮点值搜索控制台)



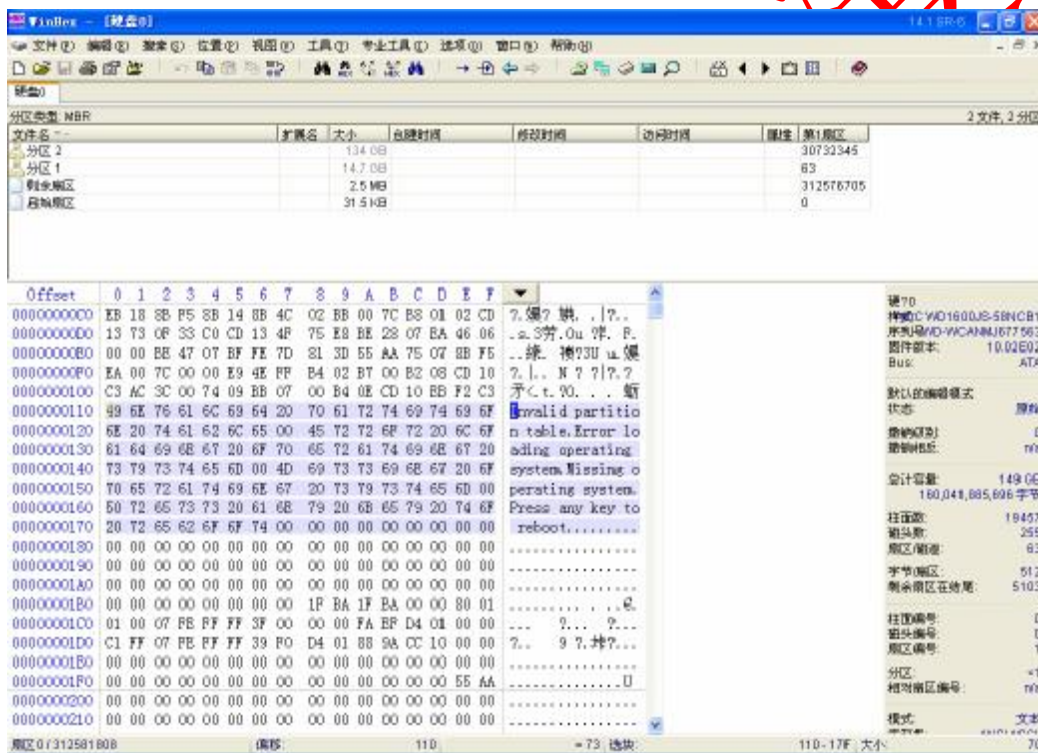
(搜索到的整数值 10, 这里是 16 进制数值 A)

单词短语搜索子项使用比较简单, 可以定位操作对象中的“固定字符串”也就是文字资源。不仅可以搜索包含文字、数字、标点及特殊符号在内的整句, 还能大概限定单词短语的长度(有一定范围)。

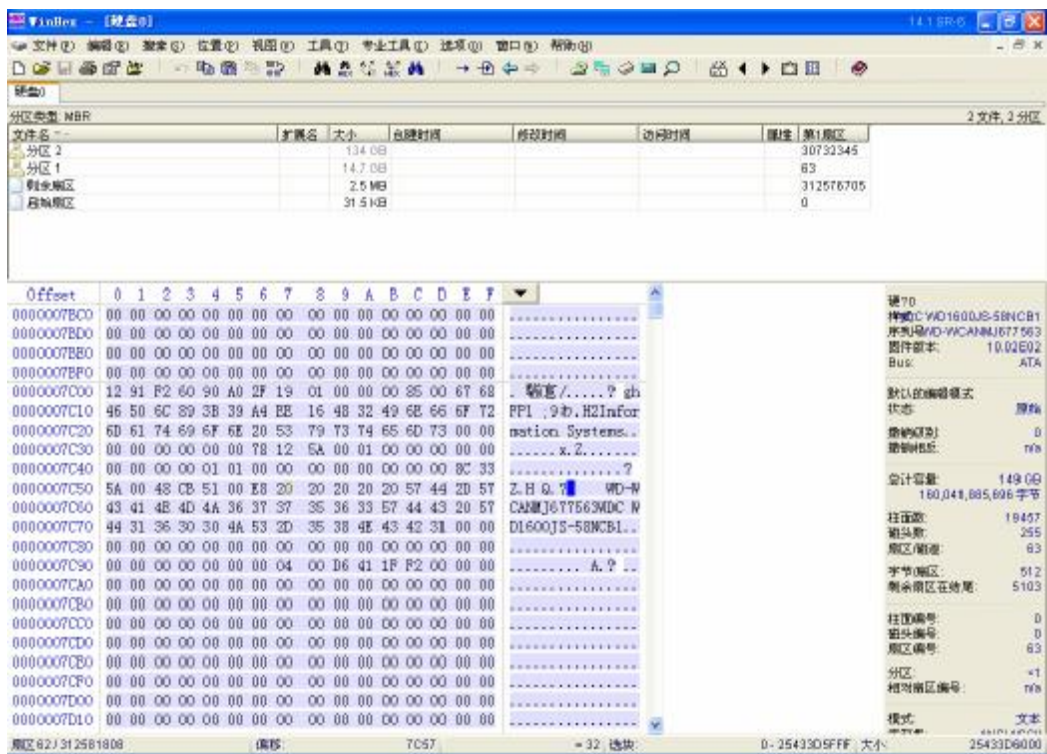
我们在当前硬盘下使用此功能, 直接定位到了 MBR 中的分区表错误提示。



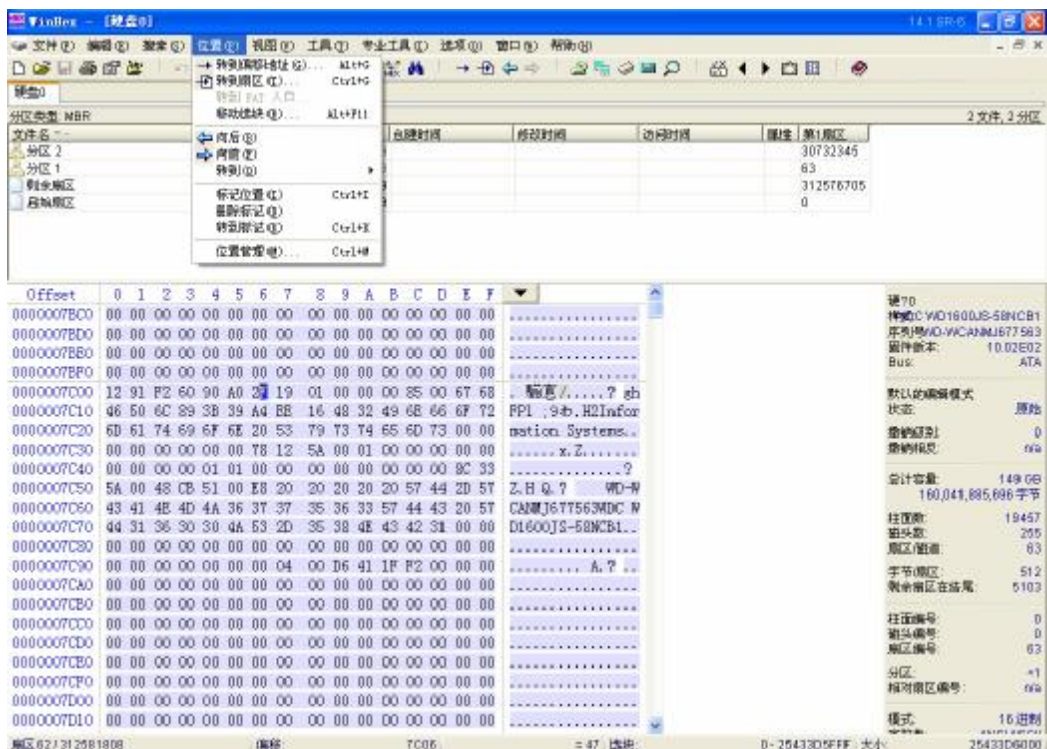
识别文本默认填 10，当前有效范围 1 至 29。



继续搜索子项可以在当前任务意外中断后从中断处继续启动搜索。这里我们继续刚才的“单词短语”搜索。



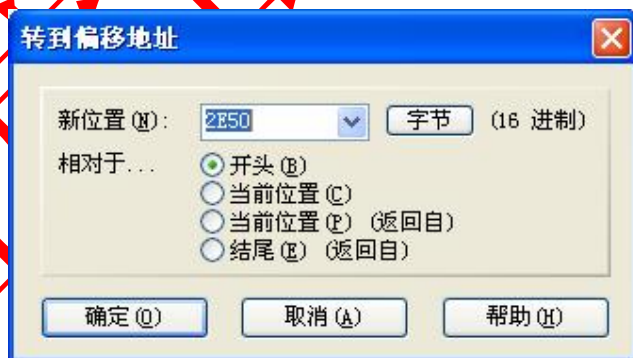
随着存储技术的发展，当前个人存储解决方案都发展到了 TB 级，而数据恢复恰恰是一项与“字节”打交道的微观工作，想凭直觉在数码海洋中打捞自己想要的东西，近乎是一个神话。所以，只有像地球经纬度那样，用数字说话，才是客观而高效的。位置主菜单便给用户提供了线性、坐标性等各种精确定位方法。其中包含若干子项分别为：转到偏移地址、转到扇区、转到 FAT 入口、移动选块、向后、向前、转到特殊位置、标记位置、删除标记、转到标记和一个功能丰富的位置管理器。充斥在磁盘中的特殊扇区、功能扇区、隐藏在扇区中至关重要的字节，是数据恢复的“下刀点”。所以用户要充分运用本菜单的各项功能，使整个工作流畅化。



转到偏移地址子项是字节级的定位方案，可以看出，编辑区的 Offset 列和 16 进制高低位构成了一个完美的坐标仪。利用 Y+X 可以快速跳转到任意字节处。编辑区的最下方一行有该子项的快捷入口，如图所示：

偏移： 7D16

点击该处出现该功能控制台：



一般来说，我们的定位会取一个参照位置，默认的参照位置是操作对象的开头，用户也可以根据自身习惯或技巧设定参照位置，如以当前位置参照、反方向参照等。我们填入任意值 2E50，以开头参照。光标成功跳转到了 2E50 处。

```
0000002E20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

我们以当前位置为参照，继续跳转 2E50 个偏移。



光标转到 5CA0 处，刚好等于 2E50+2E50。

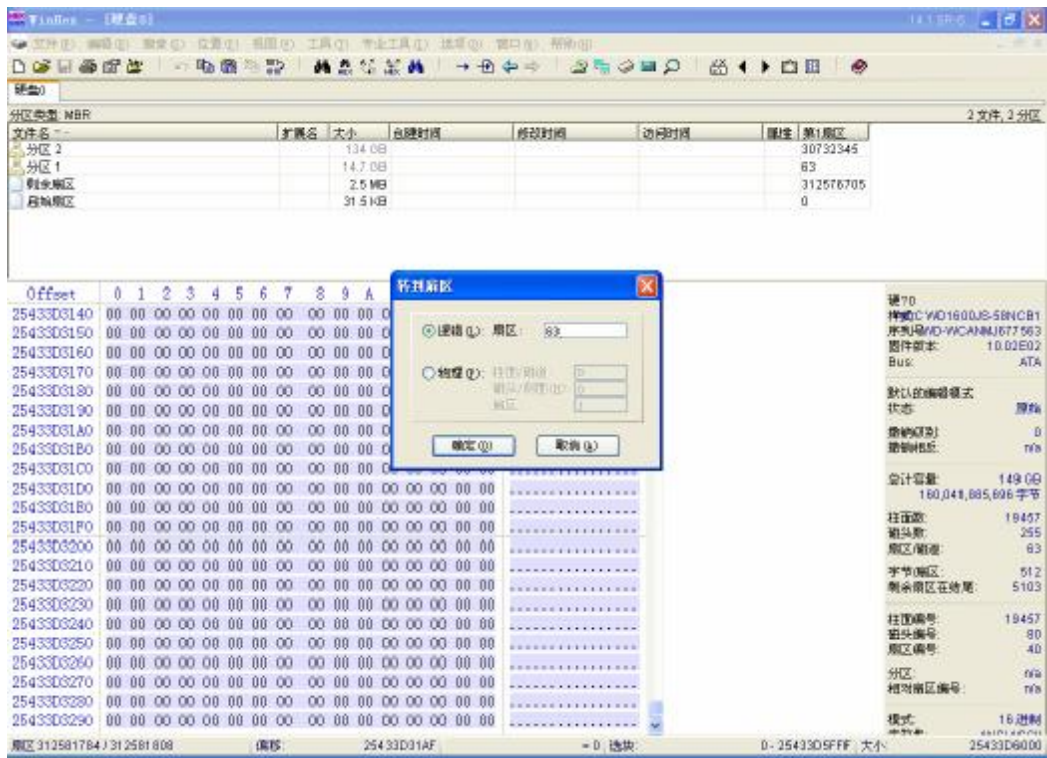
0000005C80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000005C90	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000005CA0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000005CB0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000005CC0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000005CD0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000005CE0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

我们用反向参照，选择“当前位置返回自”再次跳转 2E50 个偏移。光标又转回了 2E50 也就是 5CA0-2E50。

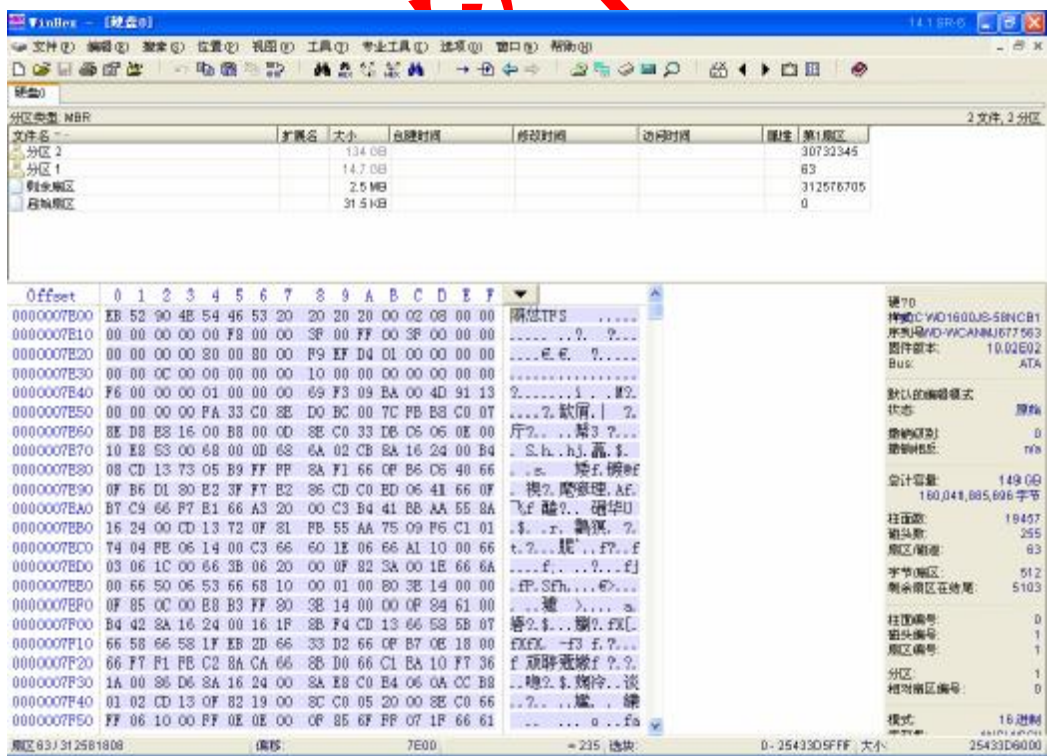
0000002E10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E30	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E40	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E50	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E60	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E70	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000002E90	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

我们以结尾做反向参照，选择“结尾返回自”，依然跳转 2E50 个偏移。光标转到了 25433D31AF 处。用总偏移量 25433D5FFF-25433D31AF 恰好是 2E50。现在大家应该明白“参照位置”的含义了。

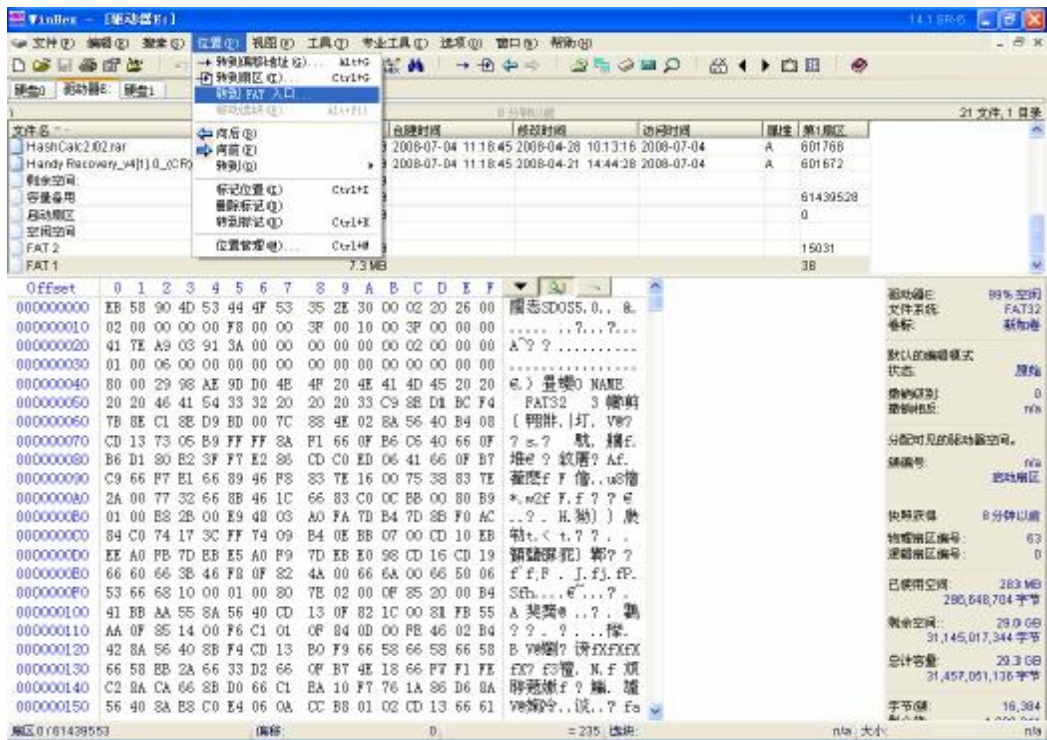
转到扇区子项大家一定非常熟悉，这是数据恢复入门操作之一。我们经常需要查看的扇区如 0 扇区、63 扇区、69 扇区、91 扇区等可以直接跳转。



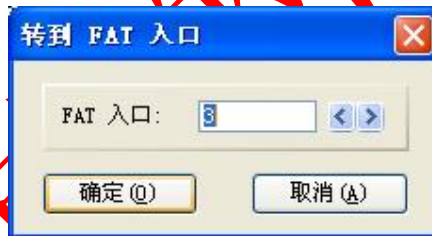
跳转到主分区的引导扇区:



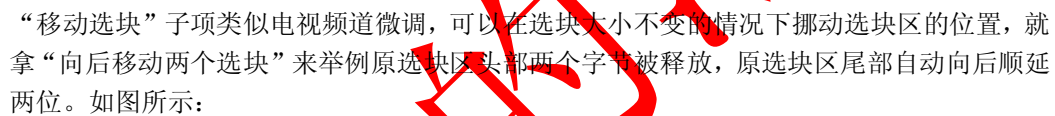
我们知道，FAT 文件系统是靠文件分配表通信数据区的，其重要性不言自明。为了方便用户准确定位 FAT 起始，WINHEX 专门定制了“转到 FAT 入口”子项。

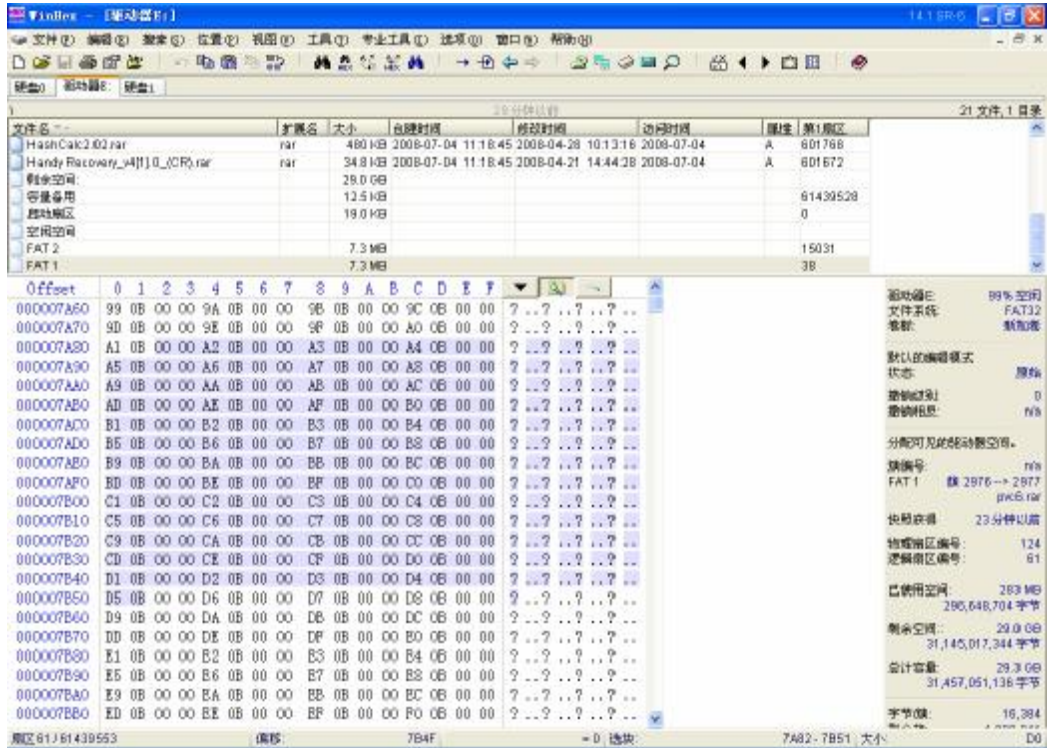


点击位置主菜单“转到 FAT 入口”子项。注意这里填入的数字是 FAT 项编号（FAT32 中每四个字节为一个 FAT 项）



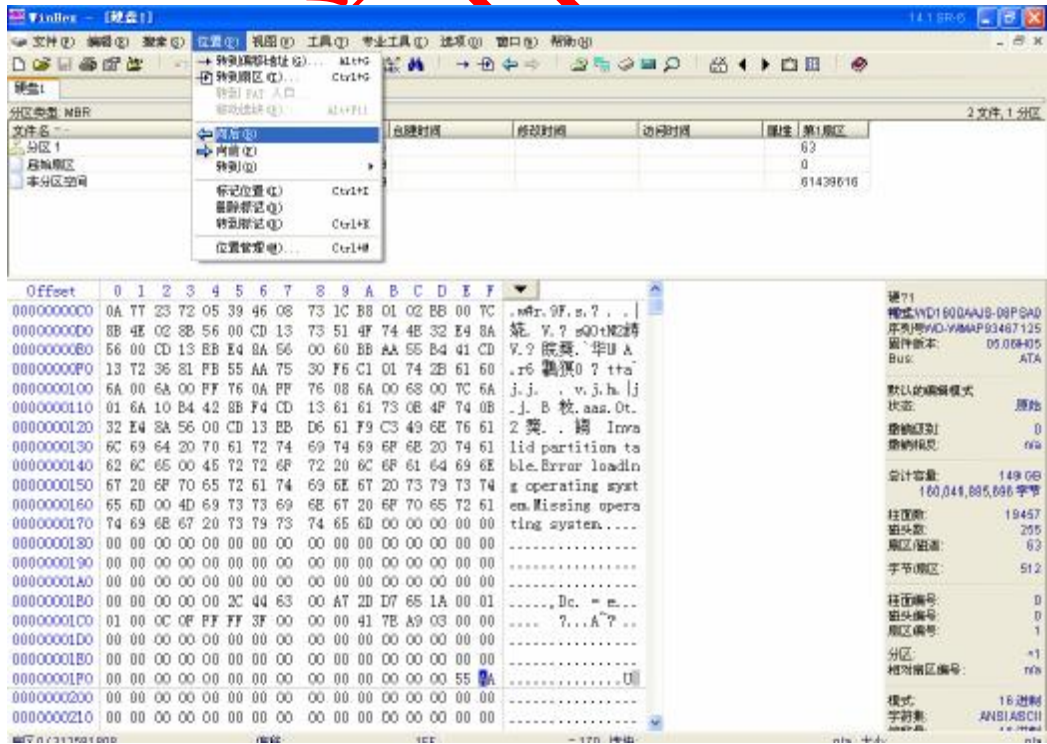
该分区的 FAT 被成功找到，如果我们想定位更深处的 FAT 项，只需输入其编号即可。





(选块移动后)

向前、向后两个子项，仅仅是一个用户操作过程的记忆，我们打开本地硬盘，我先点击 MBR 的第一个字节，再点击此扇区的最后一个字节，然后点击向后子项，光标又回到了第一个字节位置。



数据恢复是很繁琐的工程，有了“向前、向后”功能，我们就可以及时发现并回到被错过的

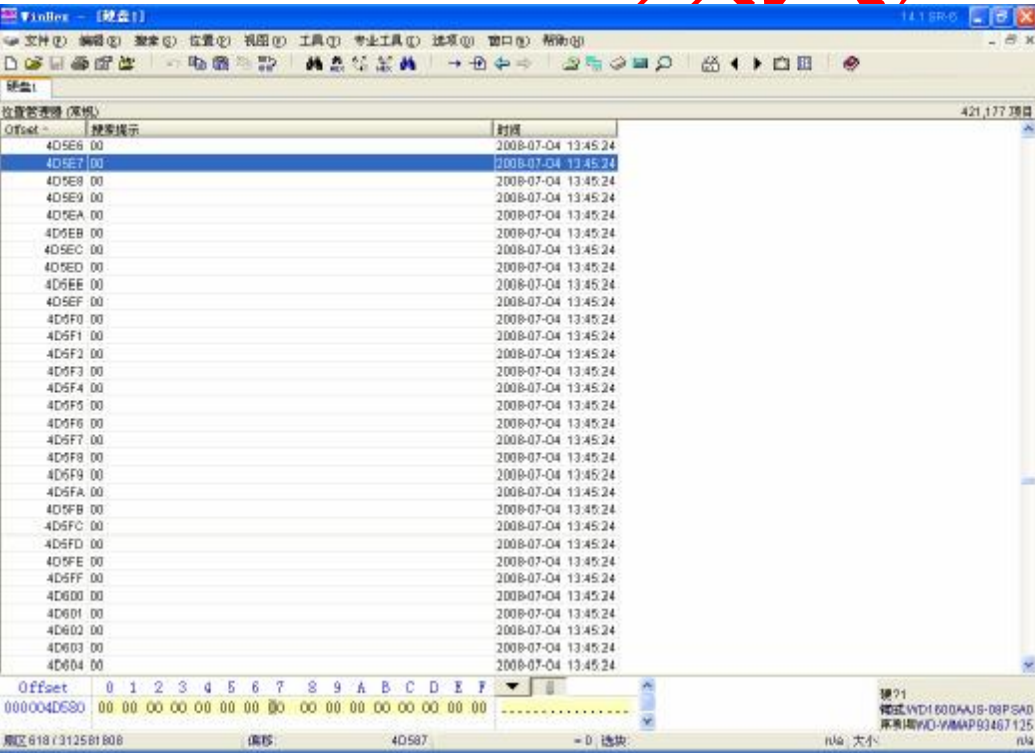
地方。

标记位置子项，是用户对发现的重要位置刻意标示的工具，非常实用。下图中“64”被打上标记。

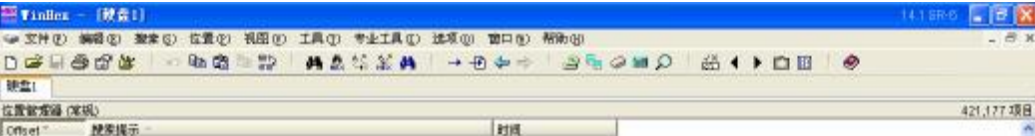
```
00000000100 6A 00 6A 00 FF 76 0A FF 76 08 6A 00 68 00 7C 6A
00000000110 01 6A 10 B4 42 8B F4 CD 13 61 61 73 0E 4F 74 0B
00000000120 32 E4 8A 56 00 CD 13 EB D6 61 F9 C3 49 6E 76 61
00000000130 6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61
00000000140 62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E
00000000150 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74
00000000160 65 6D 00 4D 69 73 73 69 6E 67 20 6F 70 65 72 61
```

与之相关的删除标记子项和转到标记子项都很好理解。

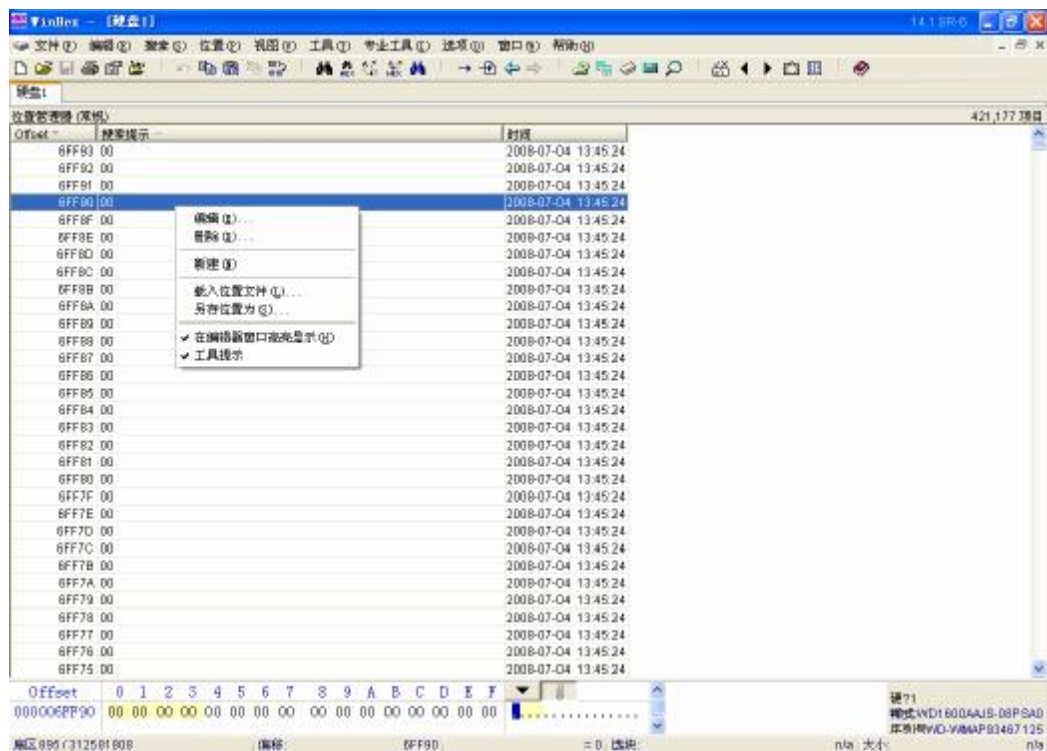
位置管理器是 WINHEX 重要组成部分，所谓的搜索列表其实就是它，可以看出每次搜索结果都被当作特殊位置存入位置管理器。这样对同样的操作对象，只要数据没有发生大的改变，我们只需调用位置管理器就可以找出上次搜索结果，避免重复用功。



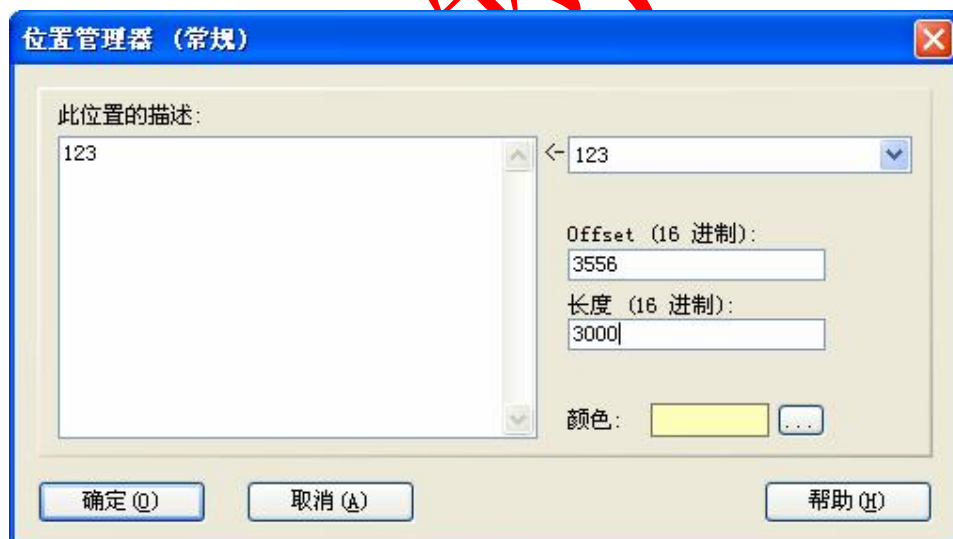
位置管理器可以存储数十万条记录，一般是按搜索先后顺序排列，用户也可以点击列表上方的控件自行排列，如按照偏移量排列、按照搜索提示内容排列、按照时间排列等。



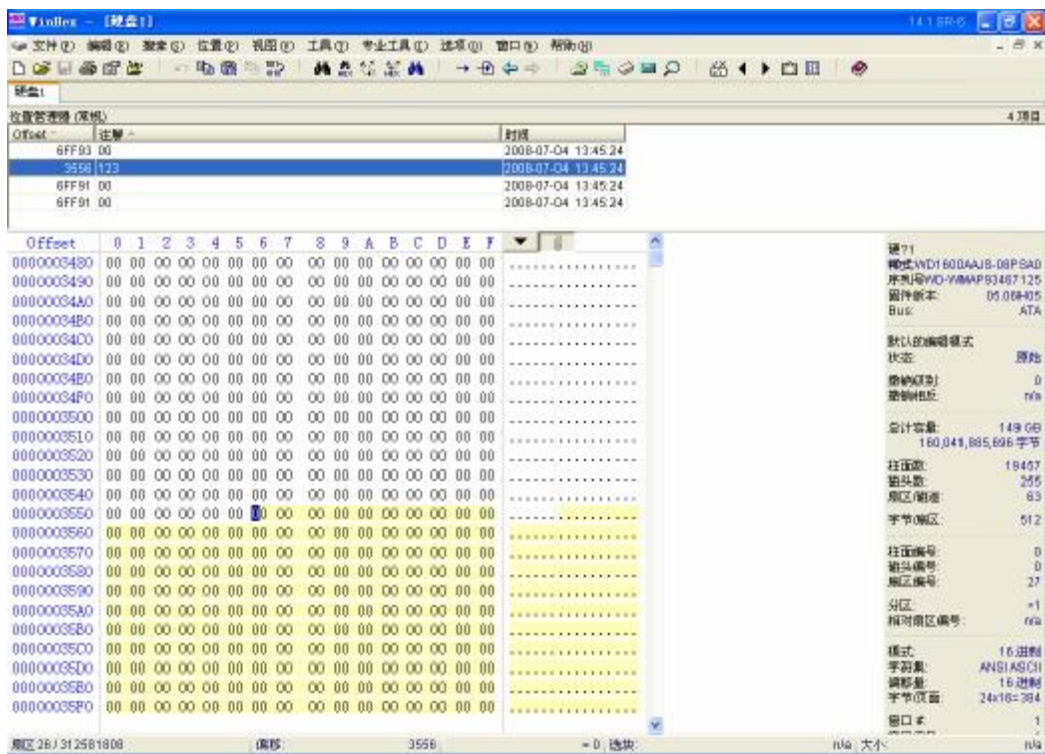
在位置管理器中任意记录上点击右键，会出现其功能菜单。包含二级子项有：编辑、删除、新建、载入位置文件、另存位置为、在编辑窗口高亮显示、工具提示。



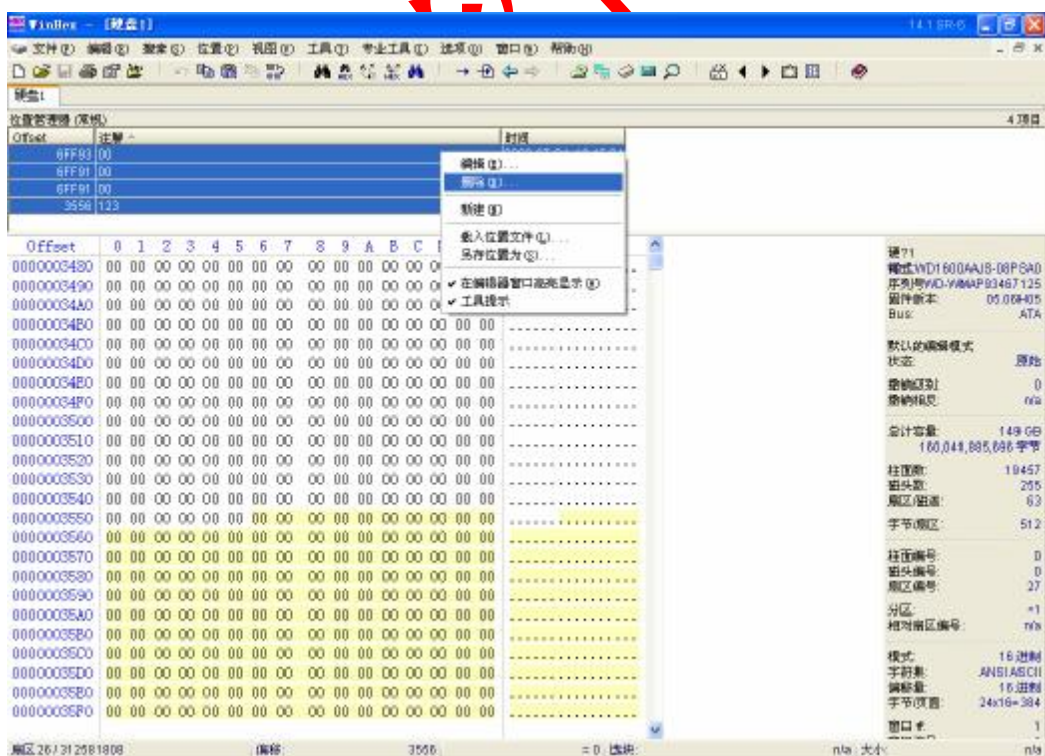
“编辑”就是在系统原先已经记录内容的基础上进行符合用户意愿的修改活动。编辑控制台不仅可以实现记录位置、搜索提示的改变，还可以实现重点着色。比如我们将位置描述区填入 123，将偏移量改为 3556，将标记长度扩大到 3000 并着黄色。



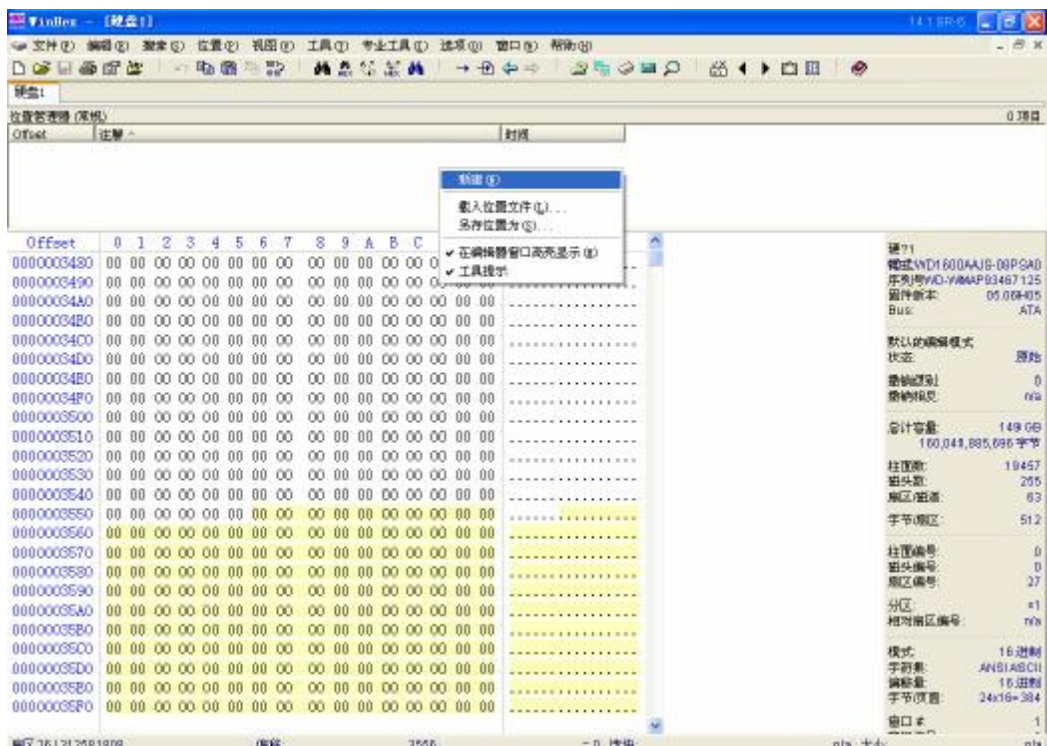
搜索记录已经按照要求改变，位置也应要求发生了偏移：



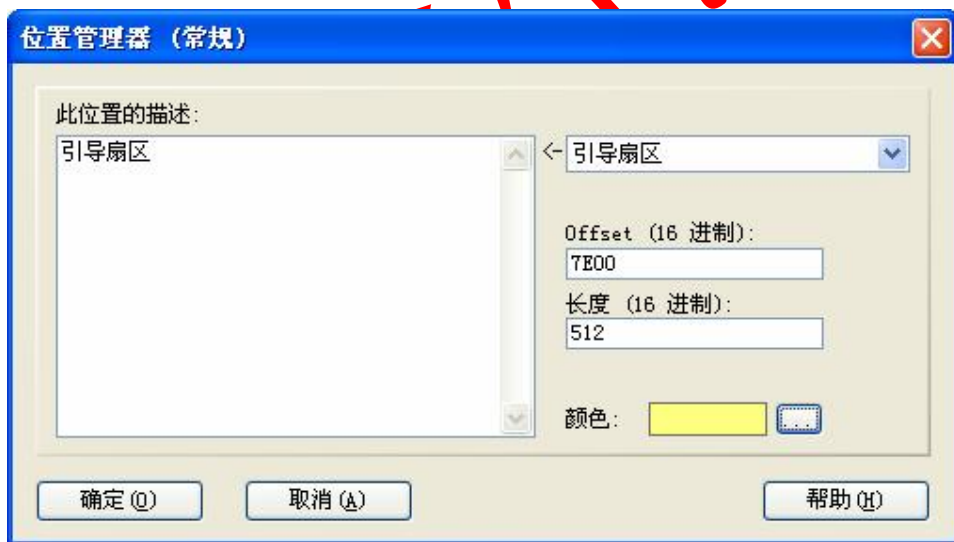
我们还可以将不必要的位置记录清除，也可以利用全选批量删除。



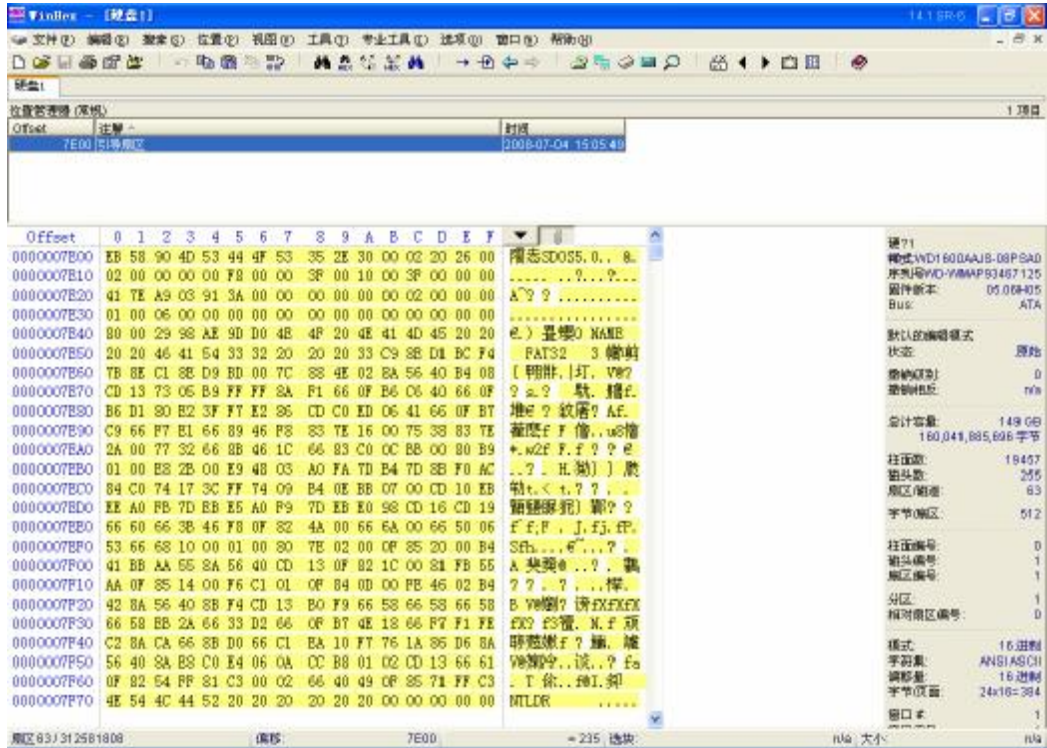
我们还可以利用“新建”把平时比较常用、通用的位置记录手动写入位置管理器中。



会出现和“编辑”相同的操作界面，这里我们为第一个分区的 DBR 新建一个位置记录。

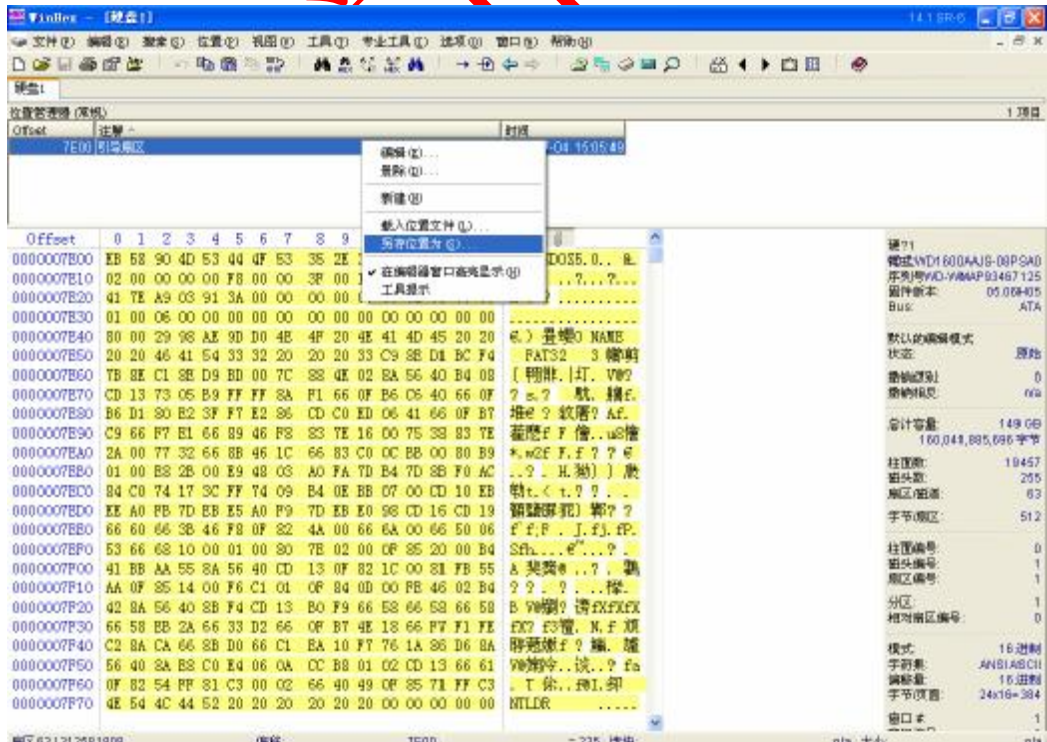


点击确定后该记录就出现在位置管理器中，点击该记录便可以跳转到引导扇区并重点着色。

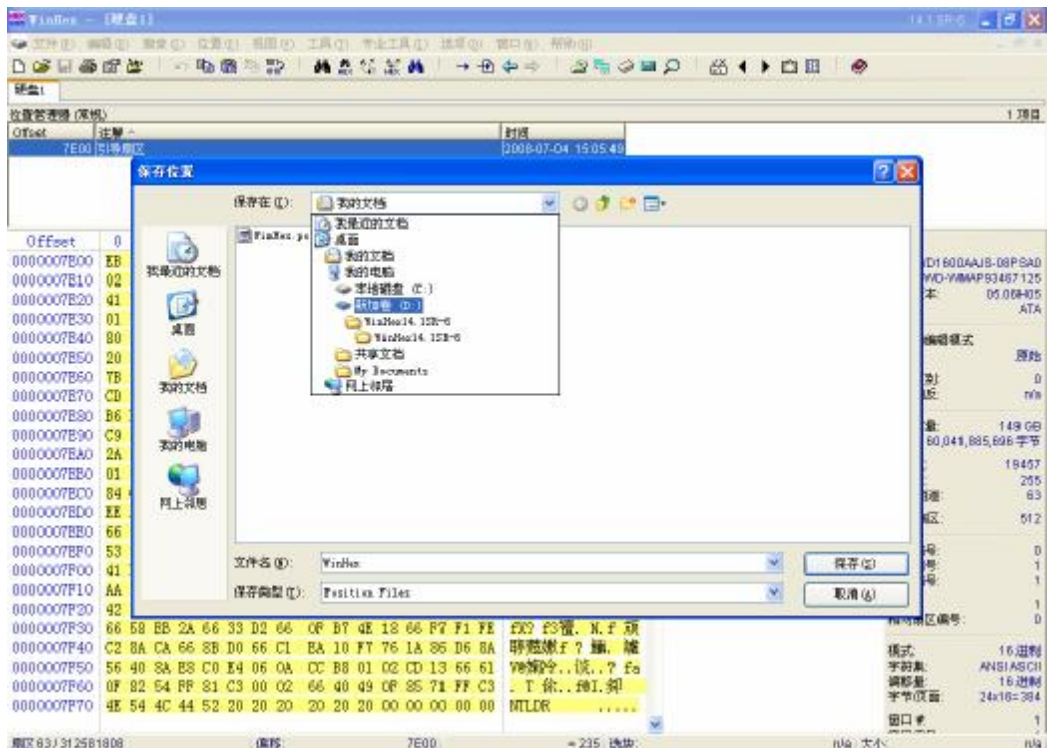


大家在手工填写分区表时需要记录每一级分区表链的重点位置，不妨暂时存入位置管理器中，调用和跳转一气呵成。

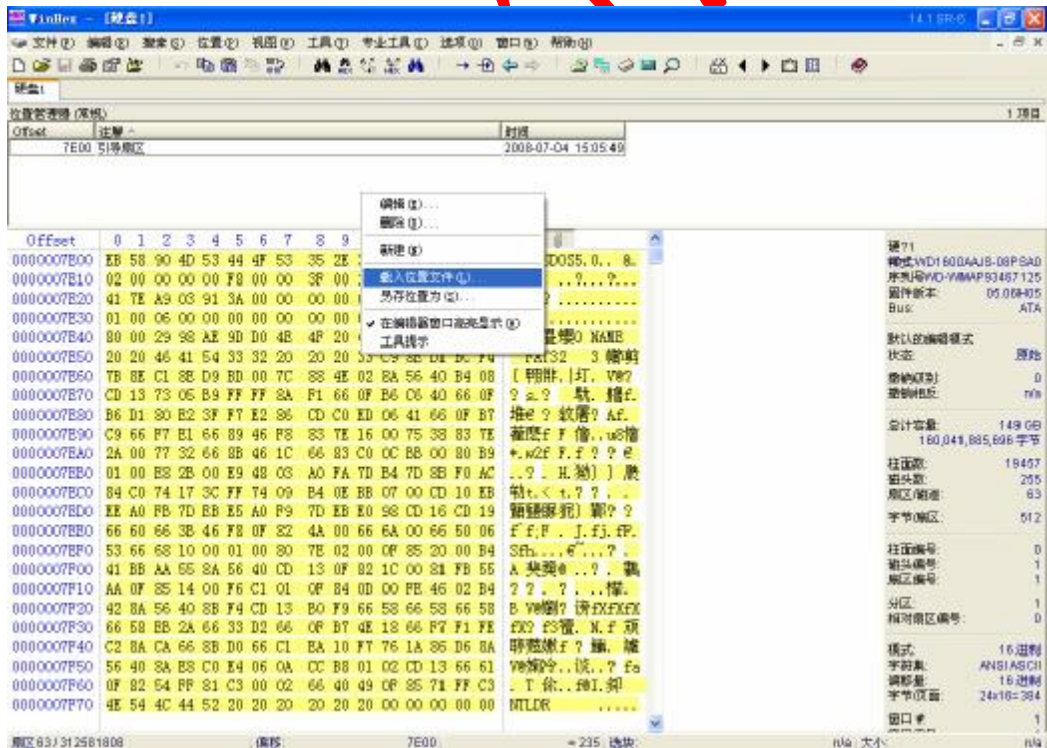
如果位置记录过于臃肿而又不能舍弃，不妨将它们保存为单独文件，需要时再重新载入。



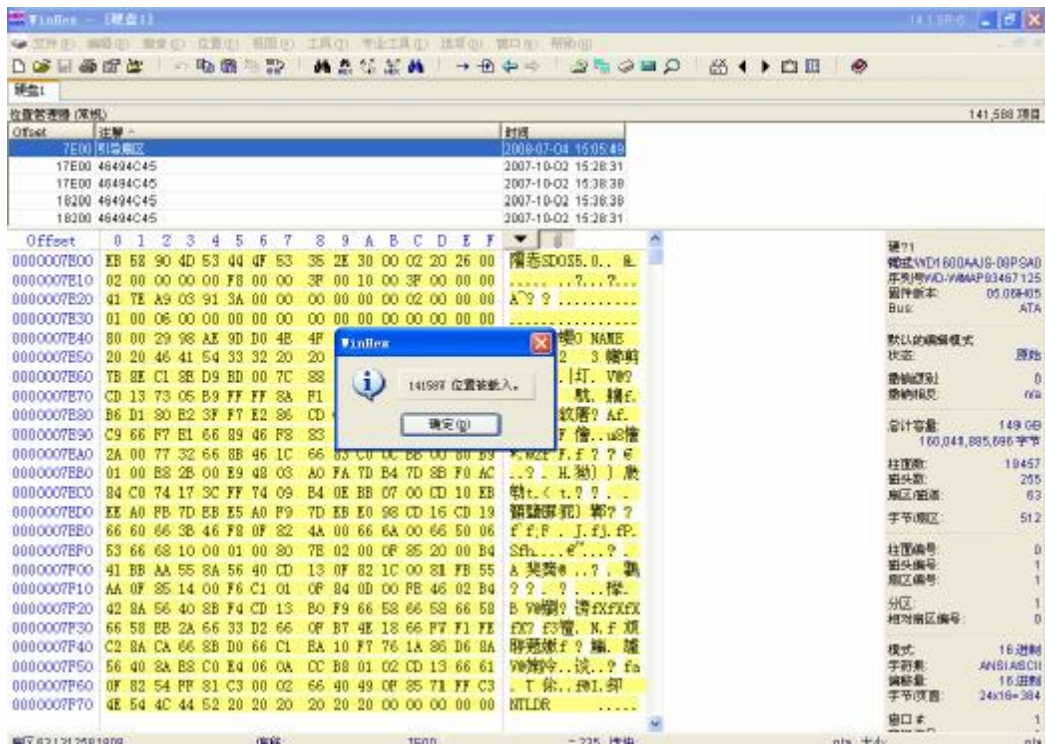
位置管理文件可以保存在磁盘的任何位置：



需要时选择“载入位置文件”导回这些文件:

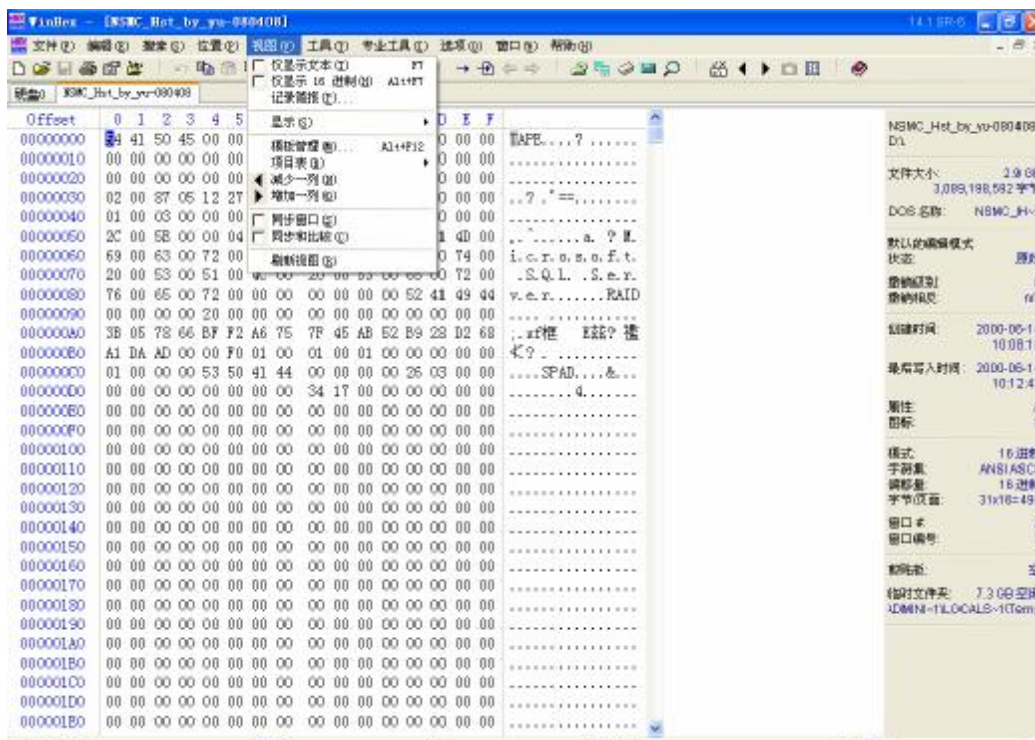


这里我们导入一个保存着大量 MFT 记录头位置的文件。

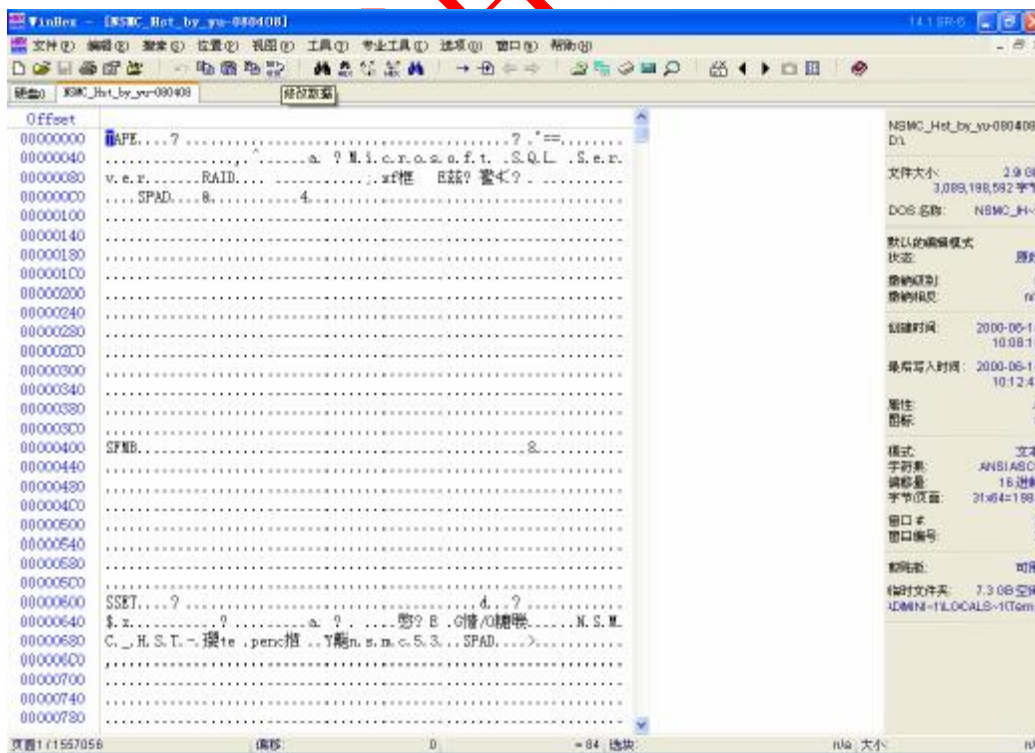


如果想去掉这些着色，可以直接将“在编辑窗口高亮显示”前面的勾去掉。

在磁盘工具中，视图是实时的、动态的、很重要的组成部分，它是软件功能丰富与否的最直观反应。一般情况下，每一种操作就会提供一种视图，以供用户及时掌握操作过程中发生的各种问题。WINHEX 视图是经过程序师尽心设计和巧妙安排的杰作，能够和用户的意图完美结合，该主菜单下属于项有：仅显示文本、仅显示 HEX、记录简报、显示、模板管理、项目表、减少一列、增加一列、同步窗口、同步和比较、刷新视图。从子项可以看出，应用心理学是该模块设计的思路之一。

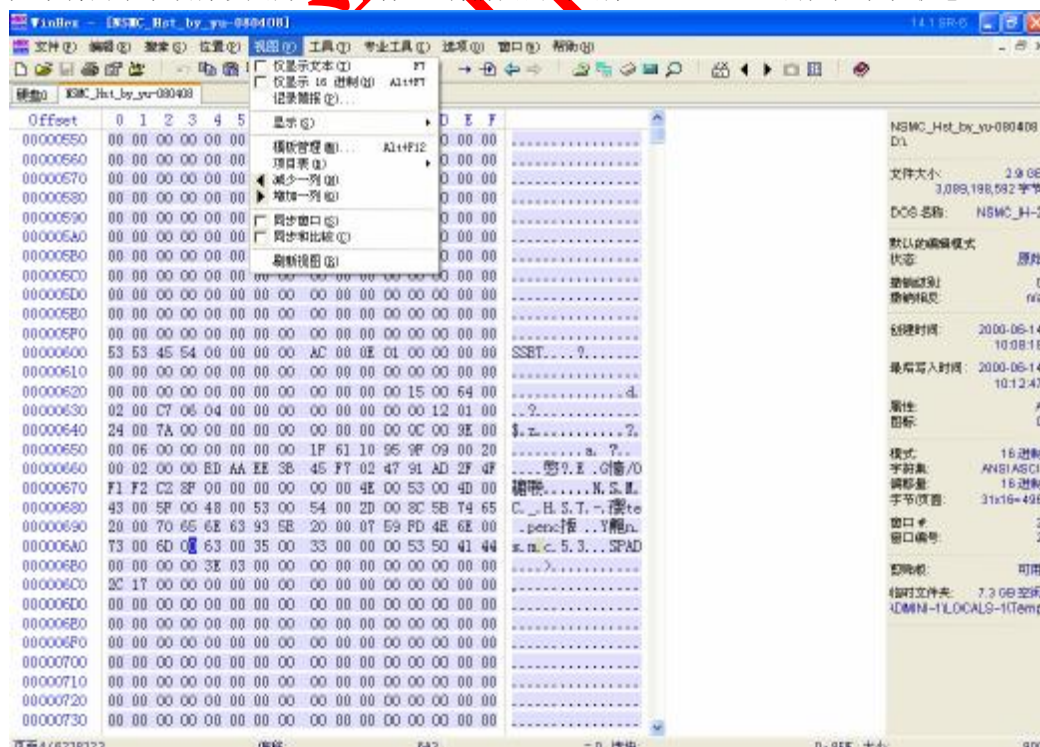


仅显示文本子项很好理解，就是隐藏 HEX 编辑区域但保留文本编辑区域，该功能在字符串识别、编辑、编码转换工作时可以有效排除“数字”带来的干扰。选择视图主菜单第一项，可以看到：

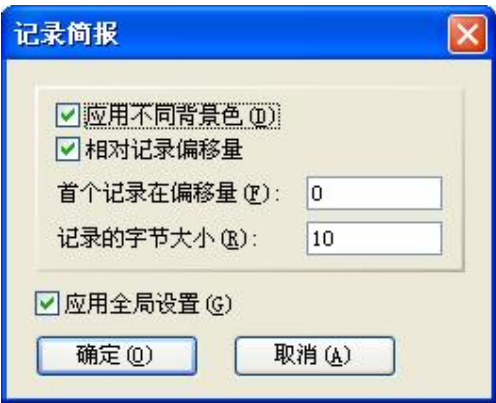


WINHEX 编辑区几乎成为一个文本编辑器，从字符串可以看出当前操作对象的是一个

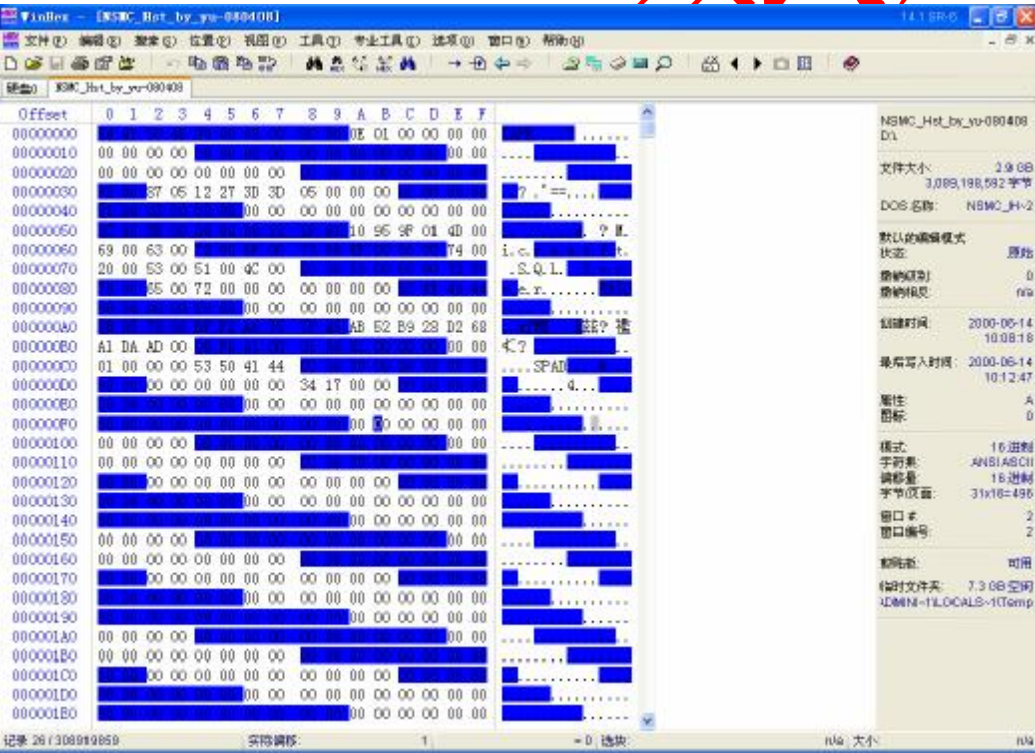
仅显示 HEX 子项同样可以减少“文本”带来的干扰。如图所示:



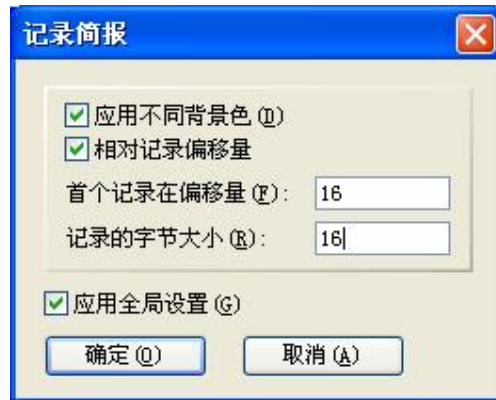
记录简报子项可以从操作对象中随机标记部分数据并加以着色，以描述操作对象的大概特征，点击该子项，出现记录简报控制台，我们看到，用户完全可以根据设定的范围进行标记。如图所示：



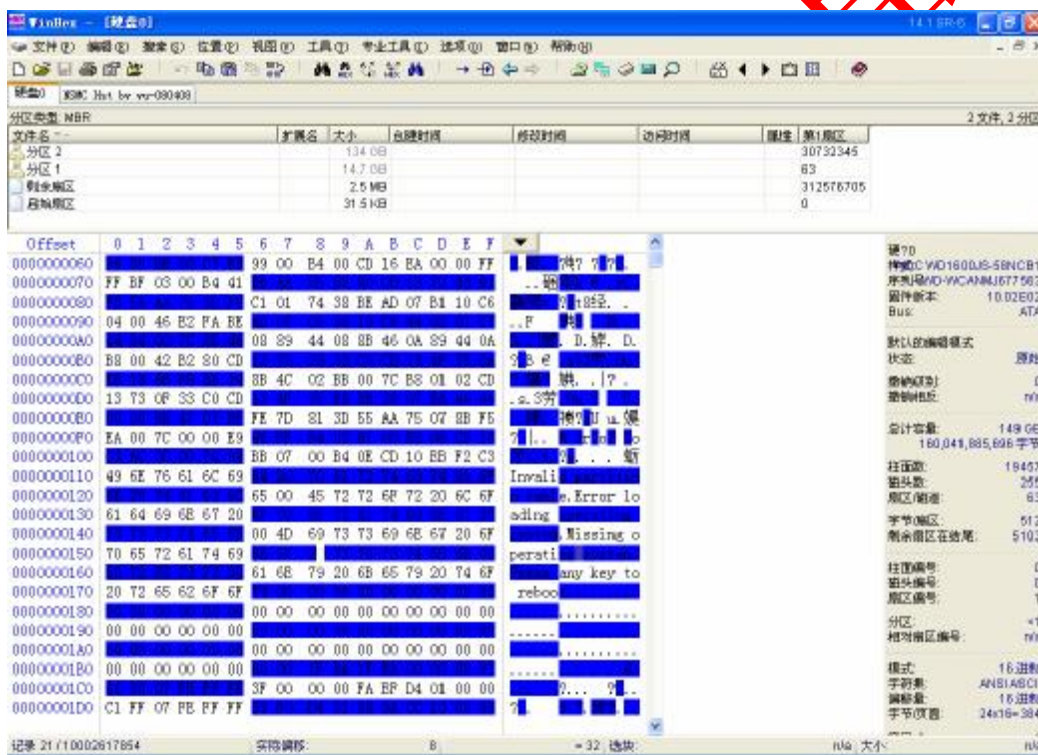
这里我们在“应用不同背景色”和“相对记录偏移量”前打钩，填入“首个记录在偏移量”为 0，“记录字节大小”为 10。如果选中“应用全局设置”则是对所有打开的对象进行操作。



可以看到，从首字节开始，每隔 10 个字节被重点着色，颜色可以修改（以后篇章会介绍），黑色则等于涂抹掉这些字节。我们重新设置如图：



两个参数都改为 16，点击确定：



记录范围发生了改变，注意该功能不对选块范围和其它操作造成任何影响，仅是一种显示方式而已。

显示子项可以选择 WINHEX 某些主要界面的去留，该子项包含的二级子项有：案例数据、目录浏览器、数据解释器、工具栏、制表站控制、详细资料面板。用户可根据自身需要和习惯决定是否在 WINHEX 中显示它们。

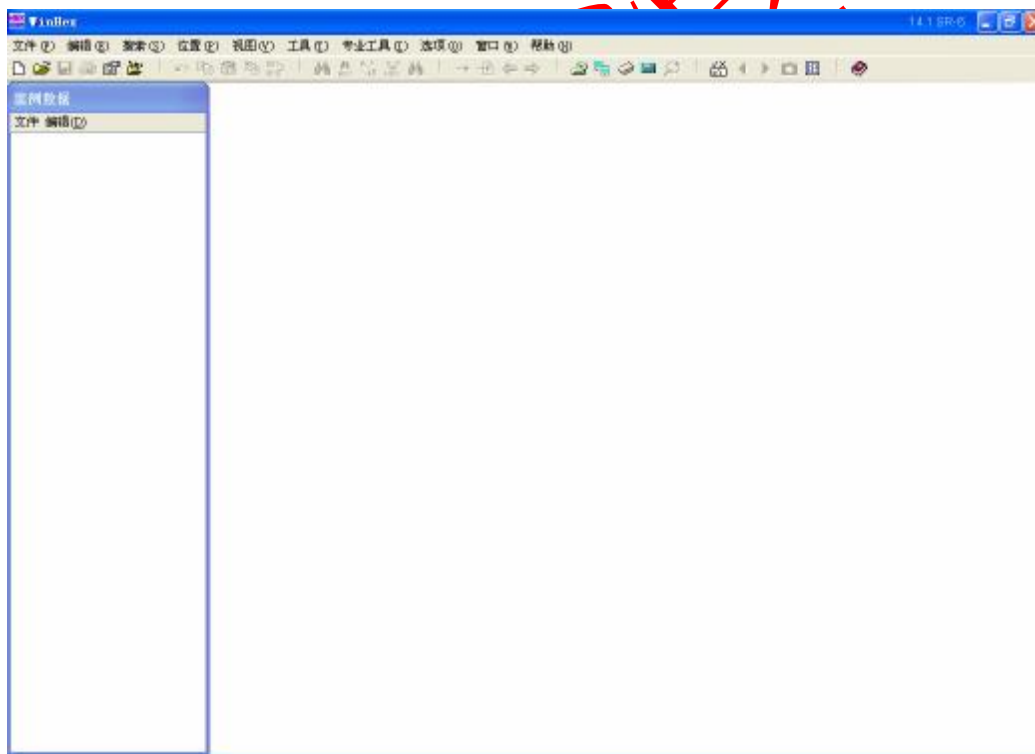
显示的第一个二级子项是“案例”也称“证据容器”，该功能是 WINHEX 法证版 X-WAYS 特有的电子取证专用模块、其它版本里仅仅提供了演示所需的最基本功能。



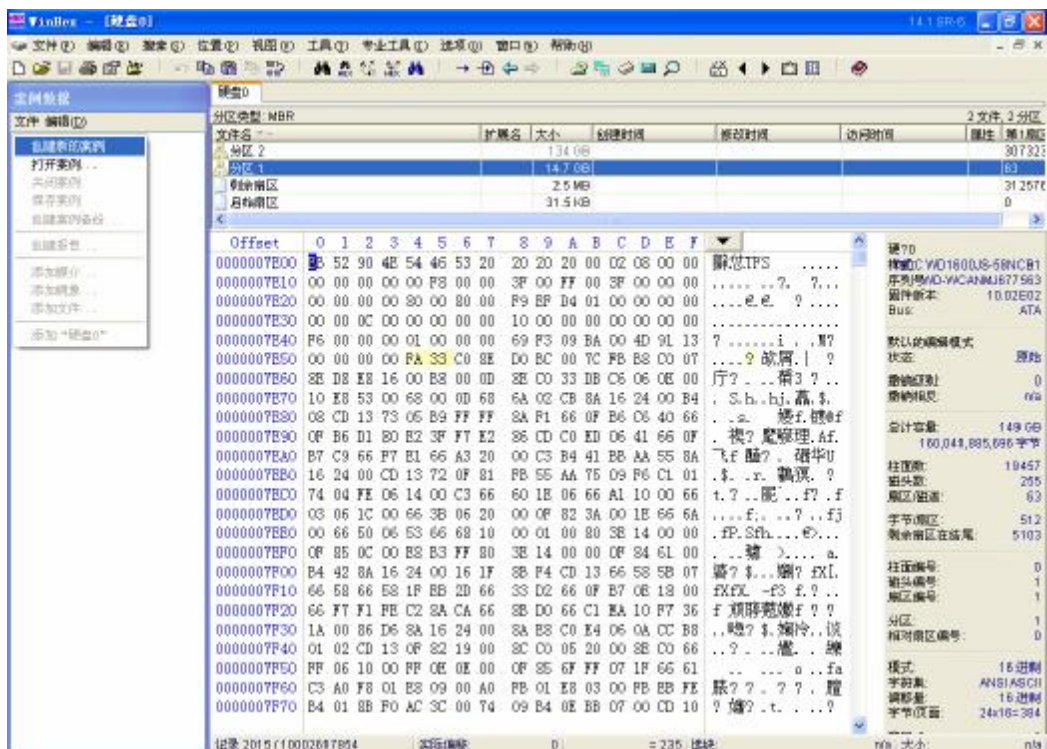
在“案例数据”前打钩，出现使用权限警告：



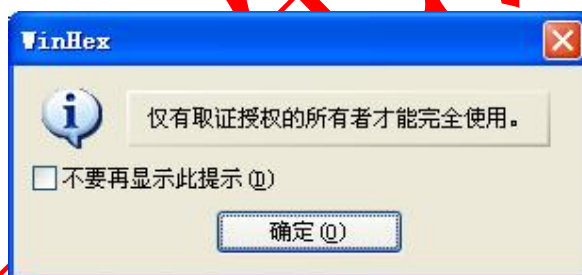
点击确定后可以发现主界面发生了重大改观，左边出现了“案例数据”控制台。包含“文件”和“编辑”两个菜单



点击文件菜单，出现若干二级子项，其中包含：创建新的案例、打开案例、关闭案例、保存案例、创建案例备份、创建报告、添加介质、添加镜像、添加文件等。我们不妨先创建一个新案例：



再次出现权限警告：



点击确定后，出现案例基本信息录入模块，从上到下，我们可以命名案例编号或名称，这里为 00001 号；显示创建日期为 2008-07-07；填写案例描述如“色情网站、裸聊……”；该案例的调查员或负责人签名如“高志鹏”，也可以填入多个调查员或调查机构、注释其各种信息；可以选择创建操作日志而保证操作的公正性、可监督性；可以设定操作日志内是否包含截屏和截屏的色彩；可以为报告简单设计版式和固定信息；可以设定操作日志的默认保存路径；可以设置自动保存时间和时区、可以加密案例文件，自动解释磁盘分区等。

案例数据

案例名称/编号 (I): 00001

创建日期: 2008-07-07

案例文件:

描述 (D): 色情网站、裸聊.....

调查员, 机构, 地址 (X): 高志鹏

☒ 创建操作日志 (L)

☒ 在日志内包含屏幕快照 (L)

☒ 屏幕快照是黑白的 (B) ...

☒ 输出至缺省证据文件夹。

日志: 61 bytes

删除 (D)...

essages.txt...

opylog.html...

报告 (选项 (O))...

显示时区...

☐ 每个取证对象一个时区 (I)

☒ 自动保存周期 (分钟) 10

☐ 设置案例文件打开口令 *

案例文件备份编号: 3

☒ 自动添加磁盘分区到案例

确定 (O)

取消 (A)

帮助 (H)

点击“报告（选项）”出现设置界面，其中包括鉴定机构徽标、特定报告头、报告封面、报告表和报告表包含的字段名称等。

报告 (选项(O))

☒ 打印基本报告

可选徽标(L): ...

☐ 左(L) ☐ 中(C) ☒ 右(R)

可选报告头(H):

☐ 左(L) ☒ 中(C) ☐ 右(R)

可选封面(F):  

☐ 包含操作日志

☒ 包含时间

☒ 包含报告表(T)

新建(N) 删除(D)... 重命名(R)...

1 文件/行

☒ 在报告中生成包含图像的副本

最大图像尺寸(I): 640 × 480

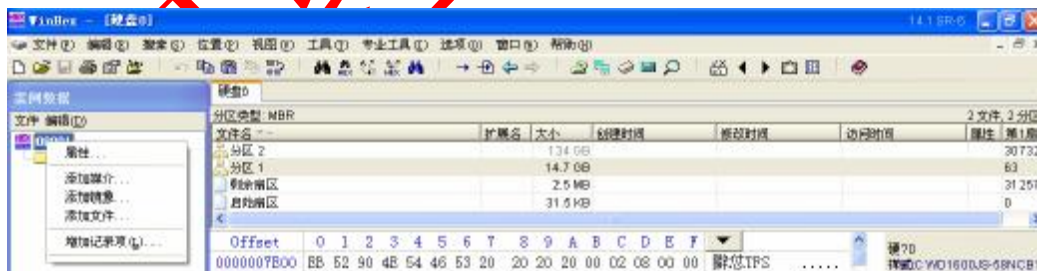
输出字段:

☒ 字段名称

- 文件名
- 描述
- 扩展名
- 类型
- 状态
- 类别
- 取证对象
- 路径
- 发件人
- 收件人
- 大小
- 创建时间
- 修改时间
- 访问时间
- 记录更新
- 删除
- 属性

确定(O) 取消(A)

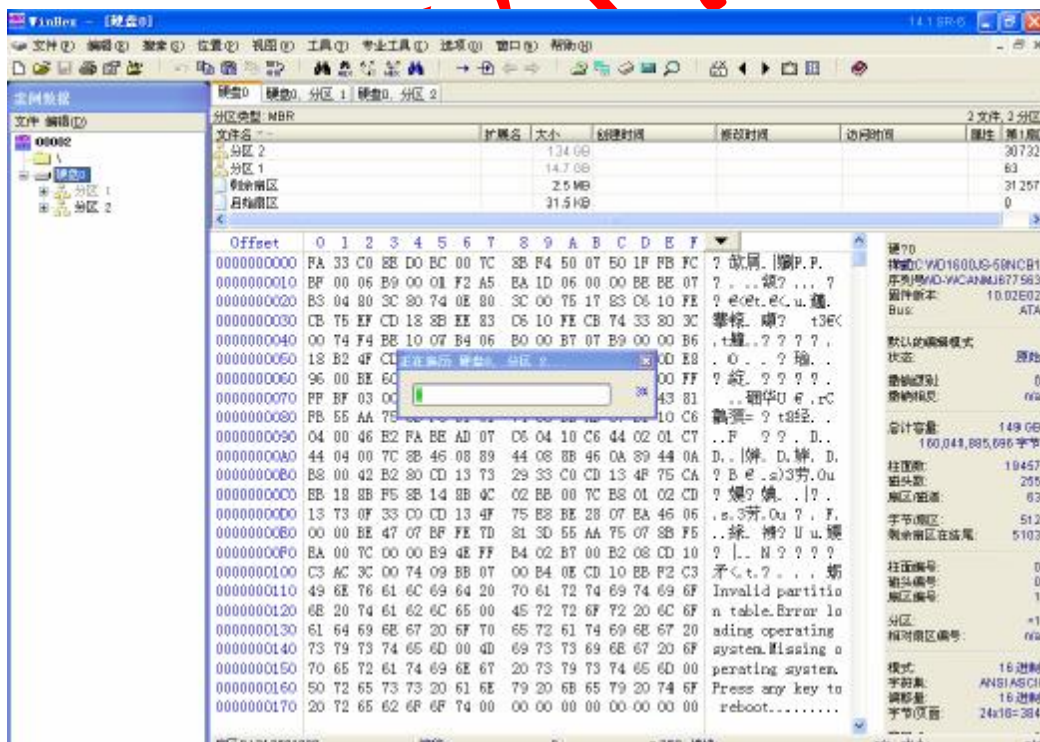
一切设置好以后点确定，出现 00001 案例树形结构，在图标或名称上点击右键，可以看到属性、添加媒介、添加镜像、添加文件、增加记录项等三级子项。



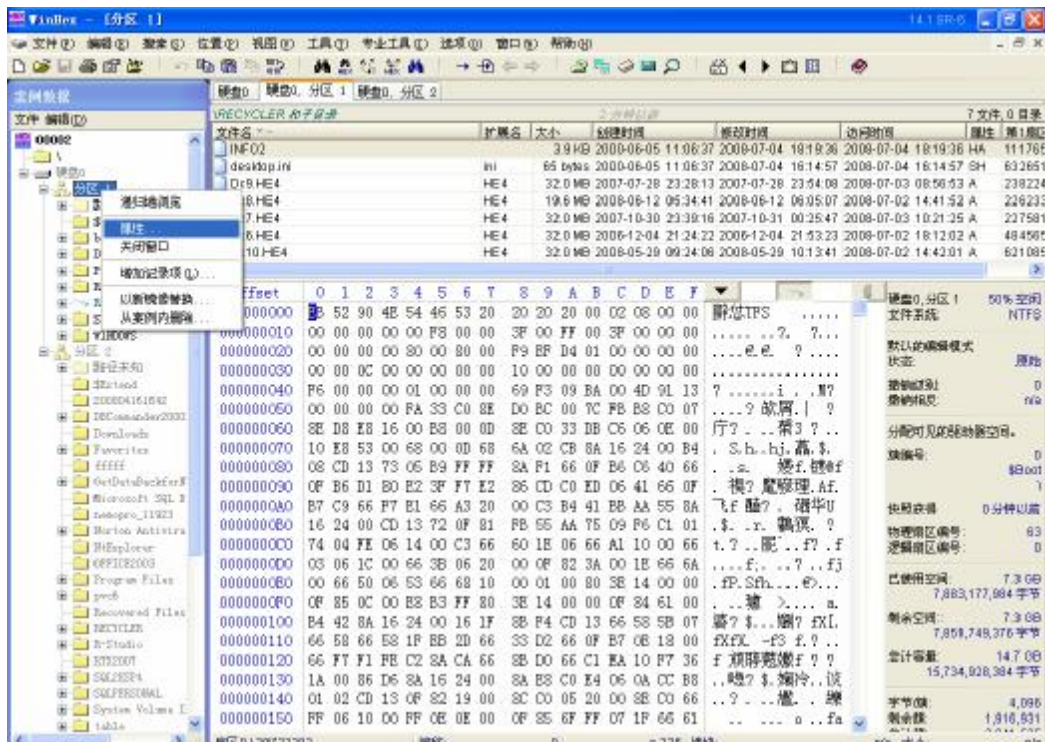
点击属性出现案例基本信息录入模块，可以查询案件的初始信息，点击添加媒介出现磁盘选择模块、这里我们选择本地硬盘：



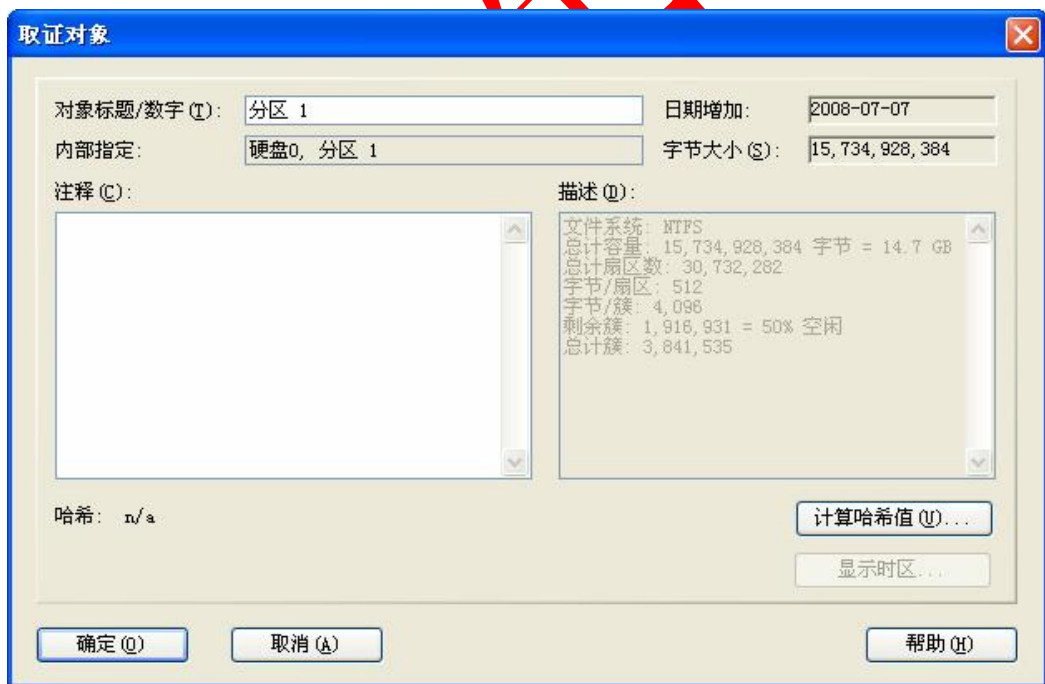
点击确定后，一个分类鲜明的目录树迅速生成。

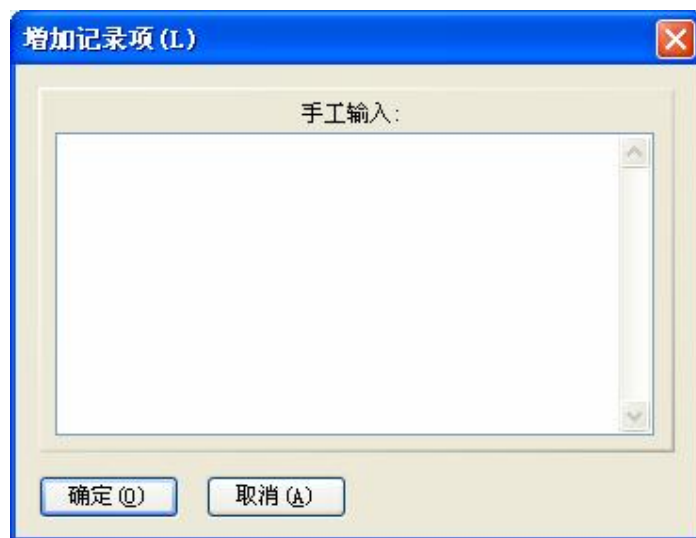


与该物理磁盘的相关分区也随之展开，并获取一个新快照，此时就可以利用 WINHEX 中的其它功能进行电子取证工作。在目录树节点单击右键，会出现具体的记录操作菜单，以实现递归浏览，增加记录项等功能。

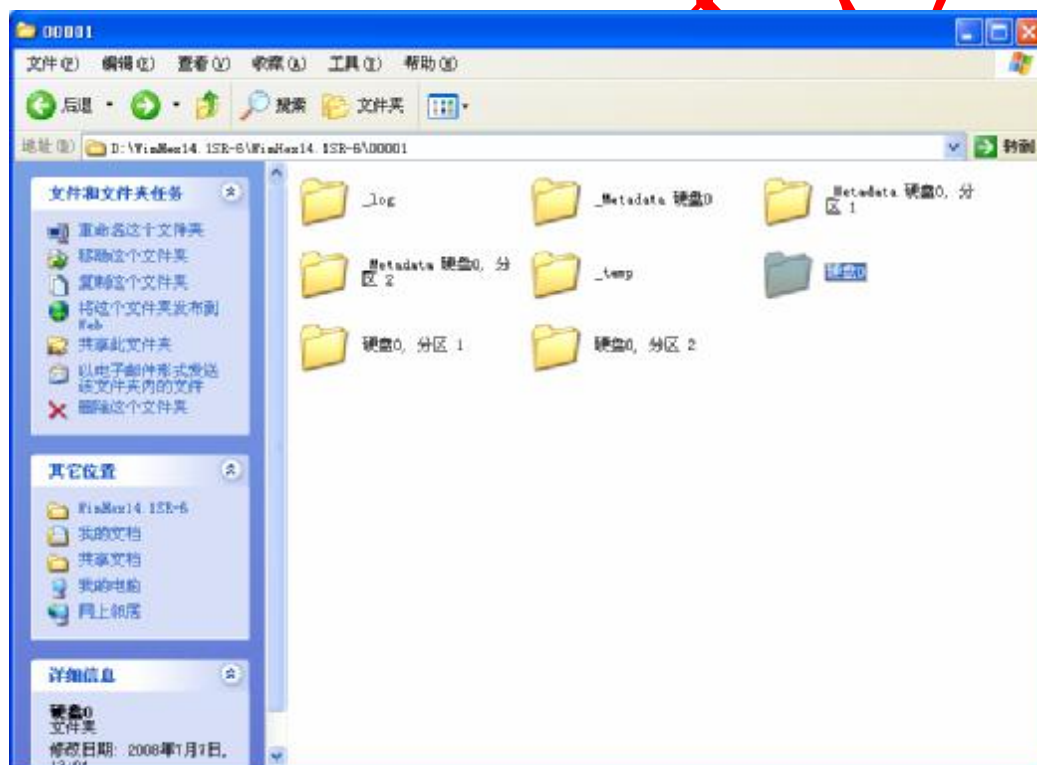


可以随时对该分区分析进展作出注释和说明:



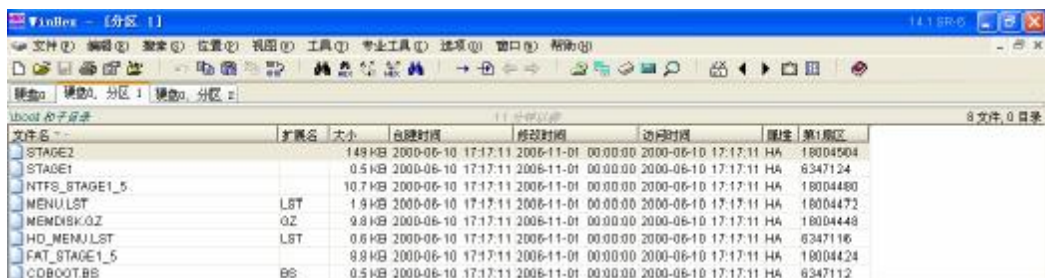


案例相关文件都自动保存在 WINHEX 目录中：

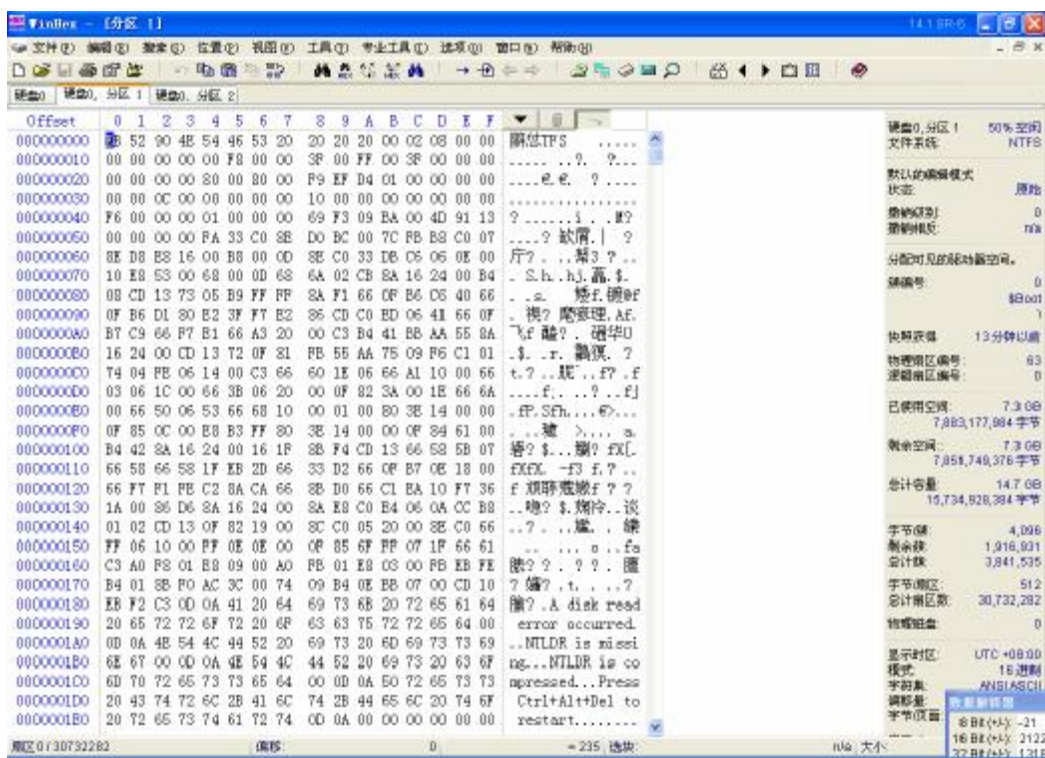


我们也可以添加一个原始镜像或一个文件进行分析，这里不再演示。

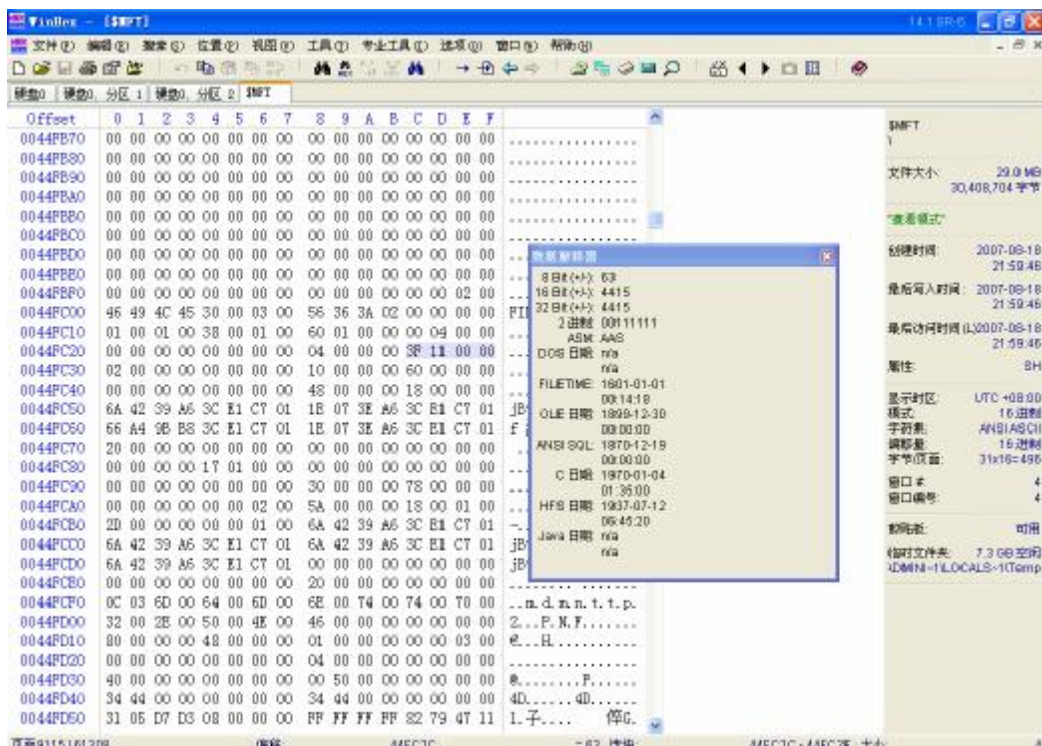
显示的第二个二级子项是目录浏览器，就是我们平时对文件的具体操作区：



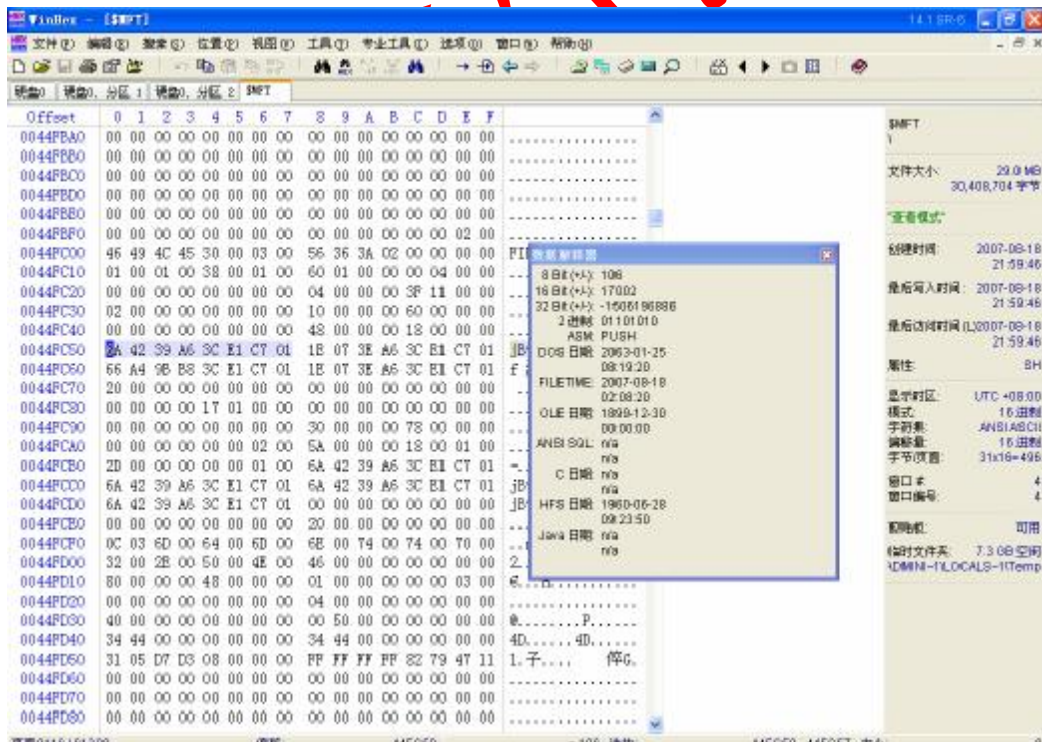
如果将选项框内的勾去掉，它将消失仅留下编辑区：



数据解释器是 WINHEX 体系中非常重要的附加功能模块，可以解析多种编码或完成计算。如我们要进行某段数据的 16 进制转 10 进制工作，直接对其选块即可，注意选块方向最好从右到左，如图所示：

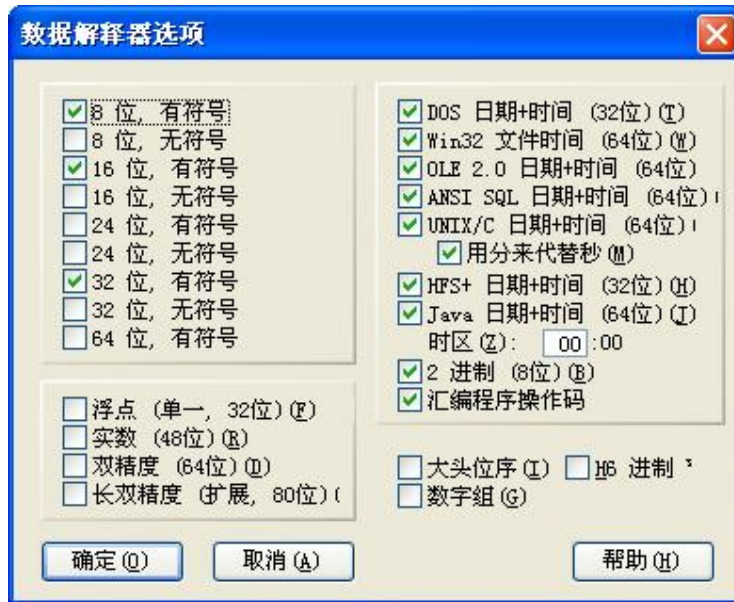


数据解释器已经得出 113F 的 10 进制值为 4415，这样就避免了大量繁琐的手工运算。我们还可以解析时间日期编码：

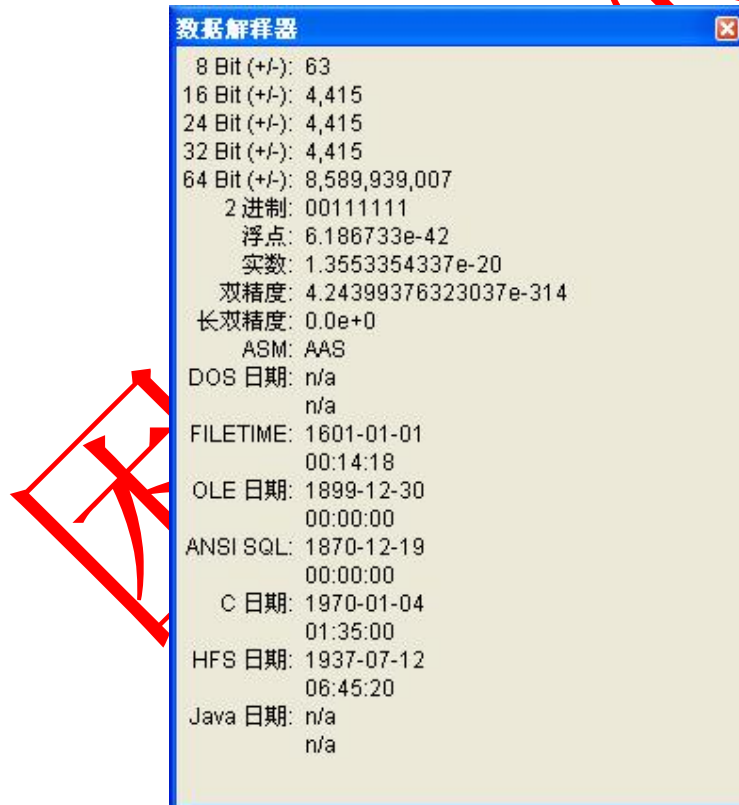


可以看到，选块区在数据解释器 FILETIME 一栏中被识别为 2007-08-18 02:08:20。

其实，数据解释器的功能远不止这些，在数据解释器上右键点击选项，出现配置窗口：



它识别范围之广、支持种类之多令人惊叹：2 至 64 位编码进制转换、各种数据运算、各种文件系统或文件编码时间函数调用和解析、静态反汇编等。



它已经成为数据恢复工作中不可或缺的计算大师。如果要更深层地利用，还需要读者有广泛的编码编程知识。

工具栏一般情况下应该默认显示，不然会造成诸多不便。制表站控制其实就是窗口标识，没有它便不能在操作对象中自由往来。

(制表站控制)

详细资料面板提供很多基础参数，最好予以保留。

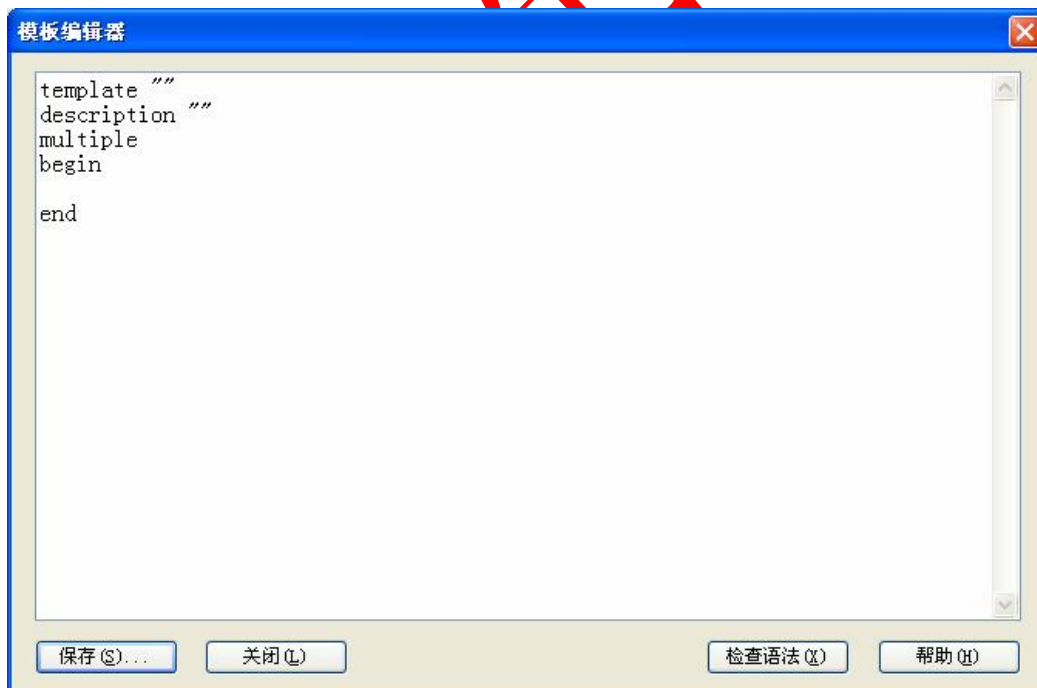
模板管理是 WINHEX 技术特色之一，所谓模板，就是将相同位置相同范围的数据套入一个框架，使用户能明了其广泛而通用的意义。这在文件系统的特殊扇区中最为常见，如 DBR、超级块等。



模板可以自由编辑或者创建，WINHEX 本身就提供了丰富的模板、涉及到多种文件系统。我们选择任意模板点击下方编辑键：



我们发现，WINHEX 对模板编辑的指令语法要求十分严格，其中包括涉及数据范围、字节定位、错误信息等。根据这些语法，我们完全可以根据自身需求新建模板。



检查语法可以帮助用户及时发现并解决错误。

这里我们不妨列出每一个模板的编写过程，供用户学习和参考：

1 Ext2/Ext3 节点

template "Ext2/Ext3 节点"

// By Jens Kirschner

description "包含文件中继信息 (节点大小 128 字节)"

applies_to disk

multiple

begin

section "File mode"

octal uint_flex "8,7,6,5,4,3,2,1,0" "权限"

move -4

uint_flex "15,14,13,12" "File type (8=注册表文件, 4=目录)"

move -4

uint_flex "9" "粘性位"

move -4

uint_flex "10" "SGID"

move -4

uint_flex "11" "SUID"

move -2

endsection

uint16 "所有者 ID"

uint32 "字节大小 (最小 4 字节)"

UNIXDateTime "访问时间"

UNIXDateTime "节点改变 "

UNIXDateTime "修改 "

UNIXDateTime "缺失 (如果不是 1/1/70)"

uint16 "组 ID"

uint16 "硬链接计数"

uint32 "扇区计数"

uint32 "文件标志"

move -4

uint_flex "19" "范围"

uint32 "系统从属"

ifequal Extents 1

section "跳过范围。"

section "请使用不同的外部节点模板。"

endsection

```
        move 60
    else
        numbering 1
        {
            uint32    "直接块 #~"
        } [12]
        uint32    "间接块"
        uint32    "双间接块"
        uint32    "三间接块"
    endif

    uint32    "文件版本"
    uint32    "文件 ACL"
    uint32    "字节大小 (最大 4 字节)"
    uint32    "区段地址 "
    uint8      "区段 #"
    uint8      "区段大小 "
    uint16     "填充"
    hex 4      "保留"
    goto 0
    move 128 // 改变此值定义节大小 (缺省: 128)
end
2 Ext2/Ext3 目录项
template "Ext2/Ext3 目录项"

// Template by Eoghan Casey
// Revised by Jens Kirschner on Sep 29, 2004

// 适用于 Ext2 驱动器的扇区
// 它包含一个目录启动和第一节点项。这个模板仅显示指定文件
// -- 已删除的文件名是不显示的。

description "定位特定文件的信息节点"
applies_to disk
multiple

begin
    uint32    "节点"
    uint16     "项长度"
    uint8      "Name length"
    uint8      "类型 (1=文件 2=目录 7=符号链接)"
    char[Name length] "文件名称"
    goto      0
    move      "项长度"
```

```
end
3 Ext2/Ext3 组描述
template "Ext2/Ext3 组描述"
.....
description "定位选块组"
applies_to disk
sector-aligned
multiple

begin
    uint32    "块位图块"
    uint32    "节点位图块"
    uint32    "节点表块"
    uint16    "空闲块计数"
    uint16    "空闲节点计数"
    uint16    "目录计数"
    uint16    "填充"
    hex 12    "保留"
end
4 Ext2/Ext3/Ext4 超级选块
template "Ext2/Ext3/Ext4 超级选块"

// Created by Jens Kirschner
// X-Ways Software Technology AG, 2004-2007

// 超级选块总是开始定位在扇区或系统选块大小的 1024。
// 它总是在组的第一个选块，除非 "稀疏超级选块功能" 是设置在驱动器。

description "应用到偏移地址的一个 Ext2/3/4 分区的 1024"
applies_to disk
sector-aligned

requires 0x38 "53 EF" // ext2 magic

begin
    uint32    "节点计数"
    uint32    "块计数"
    uint32    "保留块计数"
    uint32    "自由块计数"
    uint32    "自由节点计数"
    uint32    "第一个数据块 "
    uint32    "块大小 (0=1K, 1=2K, 2=4K) "
    int32     "区段大小(相同)"
```

```

uint32  "块 / 组"
uint32  "区段 / 组 "
uint32  "节点 / 组"
UNIXDateTime  "最后安装时间"
UNIXDateTime  "最后写入时间"
uint16  "支持计数"
int16    "最大支持计数"
hex 2    "Magic 特征 (53 EF)"
uint16  "文件系统状态"
uint16  "当检测到错误时的行为"
uint16  "局部修正级别"
UNIXDateTime  "最后一次检查时间"
uint32  "校验时最大时间(秒)"
uint32  "系统 (0: Linux)"
uint32  "校正级别"
uint16  "缺省用户 ID 保留块"
uint16  "缺省组 ID 保留块"

IfEqual "校正级别" 0
    // 未选择扩展超级选块
Else
    section "扩展超级选块扇区"
    uint32  "第一个非保留节点"
    uint16  "节点大小"
    uint16  "这是个超级选块组"

    section "兼容性特性标志"
    uint_flex "2" "日志"
    move -4
    uint_flex

"31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,1,0"  "Others"

    section "不兼容特性标志"
    uint_flex "1" "目录项文件类型"
    move -4
    uint_flex "6" "使用范围"
    move -4
    uint_flex "7" "64 位块编号"
    move -4
    uint_flex

"31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,5,4,3,2,0"  "Others"

    section "RO 兼容特征标志"
    uint_flex "0" "稀疏超级选块"

```

```

        move -4
        uint_flex
"31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1"  "Others"
    endsection

    hex 16  "卷的 UUID"
    char[16] "卷名"
    char[64] "上次载入路径"
    uint32  "位图算法"
    uint8   "块预分配"
    uint8   "目录块预分配"
    move 2
    hex 16  "日志 UUID"
    uint32  "日志节点"
    uint32  "日志设备 #"
    uint32  "最后孤体索引节点"
    numbering 1 {
        uint32  "哈希种子 ~"
    } [4]
    uint8   "缺省哈希版本"
    move 3
    uint32  "缺省载入选项"
    uint32  "第一个 metablock 块组"
    UNIXDateTime  "文件系统创建"

    section "日志索引节点备份" /17x 4 字节
    {
        uint32  "日志块 ~"
    } [12]
    uint32  "日志间接块"
    uint32  "日志双间接块"
    uint32  "日志三间接块"
    uint32  "未知"
    uint32  "日志文件大小"

    section "64 位支持"
    uint32  "块计数 hi DWord"
    uint32  "资源块 hi DWord"
    uint32  "空闲块 hi DWord"
    EndIf
End
5 FAT 长文件名项
template "FAT 长文件名项"
.....

```

description "长项格式"

applies_to disk

requires 11 0F

multiple

begin

hex 1 "次序编号"

char16[5] "文件名 (5 个字符, FF 填补)"

goto 14

char16[6] "文件名 (下次 6 个字符)"

goto 28

char16[2] "文件名 (下次 2 个字符)"

goto 11

hex 1 "0F = LFN 项"

move -1

binary "属性 (- -a-dir-vol-s-h-r)"

read-only byte "(保留)"

hex 1 "SFN 校验和"

goto 26

uint16 "16 位簇 # (总是 0)"

move 4

end

6 FAT 目录项

template "FAT 目录项"

.....

description "标准/短 项格式"

applies_to disk

multiple

begin

char[8] "文件名 (填补空白)"

char[3] "扩展名 (填补空白)"

hex 1 "0F = LFN 项"

move -1

binary "属性 (- -a-dir-vol-s-h-r)"

goto 0

hex 1 "00 = 从未使用, E5 = 擦除"

move 11

read-only byte "(保留)"

move 1

DOSDateTime "创建日期 & 时间"

move -5

byte "创建时间精度: 10 毫秒单位"

move 2

```

DOSDateTime"访问日期 (无时间!)"
move      2
DOSDateTime"更新日期 & 时间"
move      -6
uint16    "(FAT 32) 高字 / 簇 #"
move      4
uint16    "16 位簇 #"
uint32    "文件大小 (0 为目录)
end
7 FAT 引导扇区
template "FAT 引导扇区"
.....
description "BIOS 参数块 (BPB) 和其它"
applies_to disk
sector-aligned

requires 0x0  "EB"    // JMP 指令通常是使用 EB xx 90
requires 0x2  "90"    // (虽然旧的驱动器可能使用 E9 xx xx)
requires 0x1FE "55 AA"

begin
    read-only hex 3 "JMP 指令"
    char[8]  "OEM"

    section  "BIOS 参数块"
    uint16   "字节 / 扇区"
    uint8    "扇区 / 簇"
    uint16   "保留扇区"
    uint8    "FAT 计数"
    uint16   "根项目"
    uint16   "扇区 (32 MB 以下)"
    hex 1    "媒介描述 (16 进制)"
    uint16   "扇区 / FAT"
    uint16   "扇区 / 磁轨"
    uint16   "头"
    uint32   "隐藏扇区"
    uint32   "扇区 (32 MB 以上)"
endsection

hex 1      "BIOS 驱动 (16 进制, HD=8x)"
read-only uint8 "(未使用)"
hex 1      "扩展启动特征 (29h)"
uint32     "卷序列号 (十进制)"
move -4

```

```
hex 4      "卷序列号 (16 进制)"
char[11] "卷标签"
char[8]    "文件系统"

goto      0x1FE
read-only hex 2 "特征 (55 AA)"
end
8 FAT32 引导扇区
template "FAT32 引导扇区"
.....
description "BIOS 参数块 (BPB) 和其它"
applies_to disk
sector-aligned

requires 0x02 "90"
requires 0x52 "46 41 54 33 32" // ="FAT32" 在偏移地址 52
requires 0x1FE "55 AA"

begin
    read-only hex 3 "JMP 指令"
    char[8]    "OEM"

    section "BIOS 参数块"
    uint16   "字节 / 扇区"
    uint8    "扇区 / 簇"
    uint16   "保留扇区"
    uint8    "FAT 计数"
    uint16   "根项目 (未使用)"
    uint16   "扇区 (小容量)"
    hex 1    "媒介描述 (16 进制)"
    uint16   "扇区 / FAT (小容量)"
    uint16   "扇区 / 磁轨"
    uint16   "头"
    uint32   "隐藏扇区"
    uint32   "扇区 (大容量)"

    section "FAT32 扇区"
    uint32   "扇区 / FAT"
    uint16   "延迟"
    uint16   "版本"
    uint32   "根目录第 1 簇"
    uint16   "FSInfo 扇区"
    uint16   "备份引导扇区"
    read-only hex 12 "(保留)"
```

```
endsection

hex 1      "BIOS 驱动 (16 进制, HD=8x)"
read-only uint8 (未使用)
hex 1      "扩展启动特征 (29h)"
uint32     "卷序列号 (10 进制)"
move -4
hex 4      "卷序列号 (16 进制)"
char[11]   "卷标签"
char[8]     "文件系统"
endsection

goto      0x1FE
read-only hex 2 "特征 (55 AA)"
end
9 HFS+ 卷标
template "HFS+ 卷标"
description "从卷标开始定位 1024 字节"
.....
big-endian
sector-aligned
applies_to disk
requires 0x0 "48 2B"

begin
char[2]    signature
UInt16     version
UInt32     attributes
UInt32     lastMountedVersion
UInt32     journalInfoBlock

AppleDateTime createDate
AppleDateTime modifyDate
AppleDateTime backupDate
AppleDateTime checkedDate

UInt32     fileCount
UInt32     folderCount

UInt32     blockSize
UInt32     totalBlocks
UInt32     freeBlocks

UInt32     nextAllocation
```

UInt32 rsrcClumpSize
UInt32 dataClumpSize
UInt32 nextCatalogID

UInt32 writeCount
Int64 encodingsBitmap

UInt32[8]finderInfo

section "HFSPlusForkData allocationFile"

Int64 logicalSize
UInt32 clumpSize
UInt32 totalBlocks

{
UInt32 startBlock
UInt32 blockCount
}[8]

section "HFSPlusForkData extentsFile"

Int64 logicalSize
UInt32 clumpSize
UInt32 totalBlocks

{
UInt32 startBlock
UInt32 blockCount
}[8]

section "HFSPlusForkData catalogFile"

Int64 logicalSize
UInt32 clumpSize
UInt32 totalBlocks

{
UInt32 startBlock
UInt32 blockCount
}[8]

section "HFSPlusForkData attributesFile"

Int64 logicalSize
UInt32 clumpSize
UInt32 totalBlocks

{
UInt32 startBlock
UInt32 blockCount
}[8]

```

    section "HFSPlusForkData startupFile"
    Int64    logicalSize
    UInt32   clumpSize
    UInt32   totalBlocks
    {
    UInt32   startBlock
    UInt32   blockCount
    }[8]
End
10 NTFS 文件记录
template "NTFS 文件记录"
.....
description "适用于在主要文件表的记录"
applies_to disk
sector-aligned
multiple

begin
    char[4]   "特征: 文件"
    uint16    "偏移地址更新次序"
    uint16    "更新次序大小在 words"
    int64      "日志文件次序编号"
    uint16    "次序编号 (重新计算)"
    uint16    "硬链接计数"
    uint16    "到第一个标志偏移地址"
    hex 2      "标记"
    uint32    "文件记录实例大小"
    uint32    "分配记录大小"
    int64      "基础记录 (0: 自身)"
    uint16    "下一个属性 ID"
    IfEqual "到第一个属性偏移地址" 56
        move 2
        uint32 "记录 ID"
    EndIf
    goto "偏移地址更新次序"
    hex 2      "更新次序编号"
    //Update Sequence Array -> disregarded here
    goto "到第一个标志偏移地址"

{
    endsection

    hexadecimal uint32 "Attribute type"

```

```

    IfEqual "Attribute type" 4294967295
        ExitLoop
    EndIf
    uint16 "Length of the attribute"
    move 2
    IfEqual "Attribute type" 16 //属性类型 0x10: 标准信息
        move 16
        FileTime "创建在 UTC"
        FileTime "修改在 UTC"
        FileTime "记录改变在 UTC"
        FileTime "最后访问在 UTC"
        move -48
    EndIf
    IfEqual "Attribute type" 48 //标志类型 0x30: 文件名
        move 16
        uint32 "根源文件记录"
        move 2
        uint16 "根源重新计算"
        move 56
        uint8 "Namelength"
        move 1
        char16[Namelength] "Filename"
        move -82
        move (Namelength*(-2))
    EndIf
    move "Length of the attribute"
    move -8

    IfEqual "Length of the attribute" 0
        ExitLoop
    EndIf
}[20] //arbitrary number to avoid infinite loops

Goto 0
Move 1024

End
11 NTFS 引导扇区
template "NTFS 引导扇区"
.....
description "NTFS 分区引导扇区"
applies_to disk
sector-aligned

requires 0x00 "EB" // 字节 0 到 2 是

```

```
requires 0x02 "90" // JMP 指令
requires 0x03 "4E 54 46 53 20" // ID 必须是 "NTFS"
requires 0x1FE "55 AA" // "magic" 特征
```

```
begin
    read-only hex 3 "JMP 指令" //00
    char[8] "文件系统 ID" //03
    uint16 "字节 / 扇区" //0B
    uint8 "扇区 / 簇" //0D
    uint16 "保留扇区" //0E
    hex 3 "(始终零)" //10
    read-only hex 2 "(未使用)" //13
    hex 1 "媒介描述" //15
    read-only hex 2 "(未使用)" //16
    uint16 "扇区 / 磁轨" //18
    uint16 "头" //1A
    uint32 "隐藏扇区" //1C
    read-only hex 4 "(未使用)" //20
    read-only hex 4 "(总是 80 00 80 00)" //24
    int64 "合计扇区" //28
    int64 "开始 C# $MFT" //30
    int64 "开始 C# $MFTMirr" //38
    int8 "文件记录大小指示器" //40
    read-only uint24 "(未使用)"
    int8 "簇 / 索引块" //44
    read-only uint24 "(未使用)"
    hex 4 "32 位序列号 (16 进制)"
    move -4
    hexadecimal uint32 "32 位 SN (16 进制, 保留)"
    move -4
    hex 8 "64 位序列号 (16 进制)"
    uint32 "校验和" //50
    goto 0x1FE //boot load code follows
    read-only hex 2 "签名 (55 AA)" //1FE
end
```


NTFS 引导扇区, 基础偏移量: 7E00		
Offset	标题	数值
7E00	JMP 指令	EB 52 90
7E03	文件系统 ID	NTFS
7E0B	字节 / 扇区	512
7E0D	扇区 / 簇	8
7E0E	保留扇区	0
7E10	(始终零)	00 00 00
7E13	(未使用)	00 00
7E15	媒介描述	F8
7E16	(未使用)	00 00
7E18	扇区 / 磁轨	63
7E1A	头	255
7E1C	隐藏扇区	63
7E20	(未使用)	00 00 00 00
7E24	(总是 80 00 80 00)	80 00 80 00
7E28	合计扇区	30,732,281
7E30	开始 C# \$MFT	786,432
7E38	开始 C# \$MFTMirr	16
7E40	文件记录大小指示器	-10
7E41	(未使用)	0
7E44	簇 / 索引块	1
7E45	(未使用)	0
7E48	32 位序列号 (16 进制)	69 F3 09 BA

```

12 Reiser4 超级选块
template "Reiser4 超级选块"
.....
description "超级选块适用于 Reiser4 的 Format40"
applies_to disk
//扇区对齐
requires 0x0 "52 65 49 73 45 72 34"

begin
    section "主要 Reiser4 超级选块"
        char[16] "Magic 字符 ReIsEr4"
        int16 "磁盘插件 (0: Format40)"
        int16 "块大小"
        hex 16 "UUID"
        char[16] "标签"
        int64 "Diskmap Block"
    endsection

    IfEqual "磁盘插件 (0: Format40)" 0

        goto "块大小"

```

```
    section "Format40 Superblock"
    int64 "块计数"
    int64 "空闲块计数"
    int64 "根块 #"
    int64 "小量空闲对象 ID"
    int64 "文件计数"
    int64 "超级选块计数刷新计数"
    hex 4 "UID"
    char[16] "Magic 字符 ReIsEr40FoRmAt"
    int16 "树调试"
    int16 "格式化策略"
    int64 "标记"
    char[432] "未使用"
    endsection
endif
end
13 ReiserFS 超级选块
template "ReiserFS 超级选块"
.....
description "适用于 ReiserFS 分区的偏移地址 0x10000"
applies_to disk
sector-aligned
requires 0x34 "52 65 49 73 45 72" // Reiser magic "ReIsEr"

begin
    uint32 "块计数"
    uint32 "空闲块计数"
    uint32 "根块 #"
    uint32 "日志块 #"
    uint32 "日志设备 #"
    uint32 "日志大小"
    uint32 "最大处理块"
    uint32 "日志 Magic"
    uint32 "最大批量处理块"
    uint32 "最大提交寿命(秒)"
    uint32 "最大处理寿命(秒)"
    uint16 "块大小"
    uint16 "最大大小对象 ID 数组"
    uint16 "当前大小对象 ID 数组"
    uint16 "状态 (1=清除)"
    char[10] "Reiser Magic"
    uint16 "状态 (fsck)"
    uint32 "哈希功能代码"
```

```
uint16  "树高度"
uint16  "块位图块编号"
uint16  "版本"
uint16  "保留日志"
```

```
uint32  "节生成"
uint32  "标记"
hex 16  "UUID"
char[16] "卷标"
```

```
end
```

```
14 主引导记录
```

```
template "主引导记录"
```

```
.....
```

```
description "包含分区表"
```

```
applies_to disk
```

```
sector-aligned
```

```
requires 510 "55 AA"
```

```
begin
```

```
read-only hex 440 "主引导程序载入代码"
```

```
// Addition by Daniel B. Sedory:
```

```
big-endian hexadecimal uint32 "Windows disk signature"
```

```
move -4
```

```
hexadecimal uint32 "Same reversed"
```

```
// This SN is created by any NT-type OS (NT, 2000, XP,
```

```
// 2003) and used in the Windows Registry.
```

```
move 2
```

```
numbering 1
```

```
{
```

```
section "分区表项 #~"
```

```
hex 1 "80 = 活动分区"
```

```
uint8 "开始头"
```

```
uint_flex "5,4,3,2,1,0" "开始扇区"
```

```
move -4
```

```
uint_flex "7,6,15,14,13,12,11,10,9,8" "开始柱面"
```

```
move -2
```

```
hex 1 "分区类型指示 (16 进制)"
```

```
uint8 "结束头"
```

```
uint_flex "5,4,3,2,1,0" "结束扇区"
```

```
move -4
```



```

uint_flex "7,6,15,14,13,12,11,10,9,8" "结束柱面"
move -2
uint32    "扇区在前的分区 ~"
uint32    "扇区在分区 ~"
} [4]

endsection
read-only hex 2 "特征 (55 AA)"
end

```

模板编写可不是一件容易的事情，以上模板都是前辈大师里给我们的珍贵资料，是各种文件系统研究的辉煌成果。但是就其语法本身来说，只要掌握了关键指令的含义和格式，还是有我等发挥余地的。比如 `uint` 其实就是整型运算（`int` 是带符号，`uint` 是不带符号）这里的 `uint8` 仅代表 8 位也就是通常 1 个字节的 16 进制转 10 进制运算，`uint16` 通常为两个字节进制转换，`uint32` 通常为双字 8*4 位运算，`int64` 通常带符号四字 8*8 位运算（比如负数）。而常量读取为 `read-only hex` 后面跟字节数，变量读取为 `hex` 后面跟字节数，`move` 是位移等。

数据类型	字节数	取值范围
Int8	1	-128 到 127
Int16	2	-32,768 到 32,767
Int32 或 Integer	4	-2,147,483,648 到 2,147,483,647
Int64	8	-2^63 到 2^63-1
UInt8 或 Byte	1	0 到 255
UInt16	2	0 到 65535
UInt32	4	0 到 4,294,967,295
UInt64	8	0 到 2^64-1

我们不妨选择任意扇区举例，自己动手写一个简单模板。

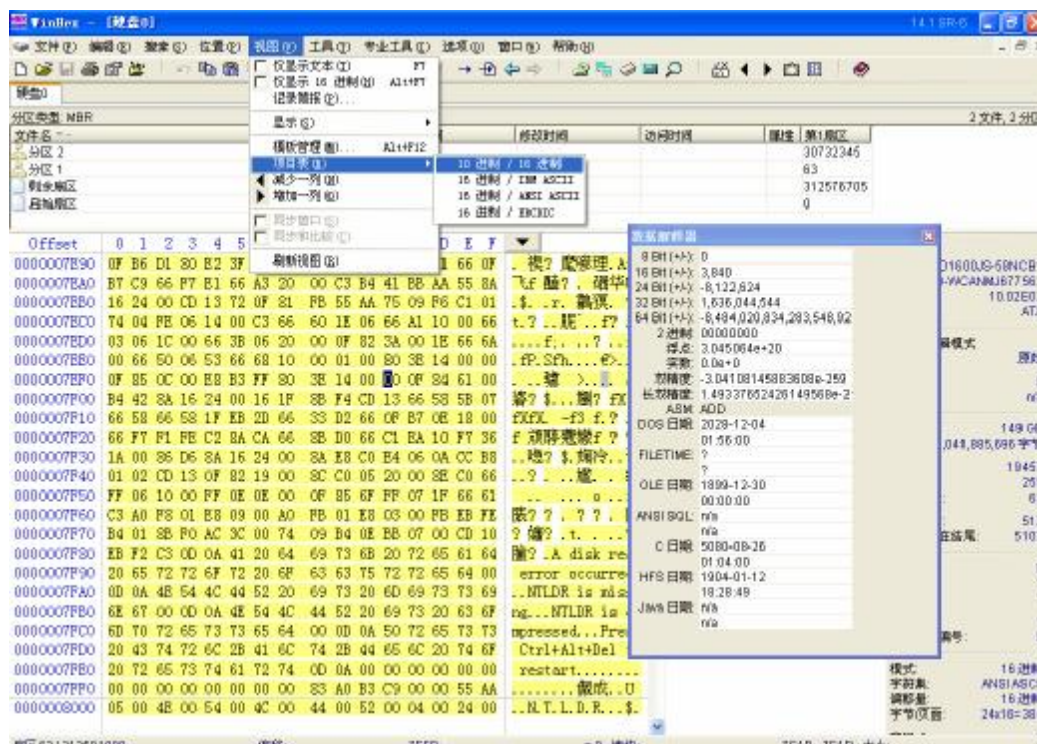


选择任意扇区偏移量，点击应用，如图所示：



项目表子项其实包含了四个编码转换表，明确指出了字符串符号与 16 进制数值相对应的关系。

ASCII 想必大家已经很熟悉而 EBCDIC (广义二进制编码的十进制交换码) (读作 "ehb-suh-dik"或"ehb-kuh-dik") 则作为字母或数字字符的二进制编码。它是为 IBM 的 S/390 上的 IBMOS/390 操作系统上使用的文本文件的编码, 并且数千个公司为它们的遗留应用程序和数据库使用这种编码。在一个 EBCDIC 的文件里, 每个字母或数字字符都被表示为一个 8 位的二进制数 (一个 0、1 字符串)。有 256 个可能的字符被定义 (字母, 数字和一些特殊字符)。



10 进制 / 16 进制																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
16	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
32	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
48	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
64	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
80	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
96	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
112	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
128	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
144	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
160	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
176	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
192	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
208	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
224	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
240	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

(10 进制转 16 进制基本元素表)

16 进制 / IBM ASCII																			
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
00		☺	☹	♥	♦	♣	♠	•	◼	◻	♂	♀	♪	♫	♫	♫			
10	▶	◀	↑	↓	!!	¶	§	_-	↑	↓	→	←	↖	↗	↘	↙			
20		!	"	#	\$	%	&	'	()	*	+	,	-	.	/			
30	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O			
40	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	~				
50		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o			
60	p	q	r	s	t	u	v	w	x	y	z	{		}	~				
70		€	•	,	f	”	...	†	‡	^	‰	Š	<	Œ	+	+			
80	•	•	•	“	”	•	—	—	~	™	š	>	œ	+	+	Ÿ			
90		i	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	–	®	—			
A0	•	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿			
B0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï			
C0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß			
D0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï			
E0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ			
F0																			

(16 进制值与 IBM ASCII 符号之间的对应关系)

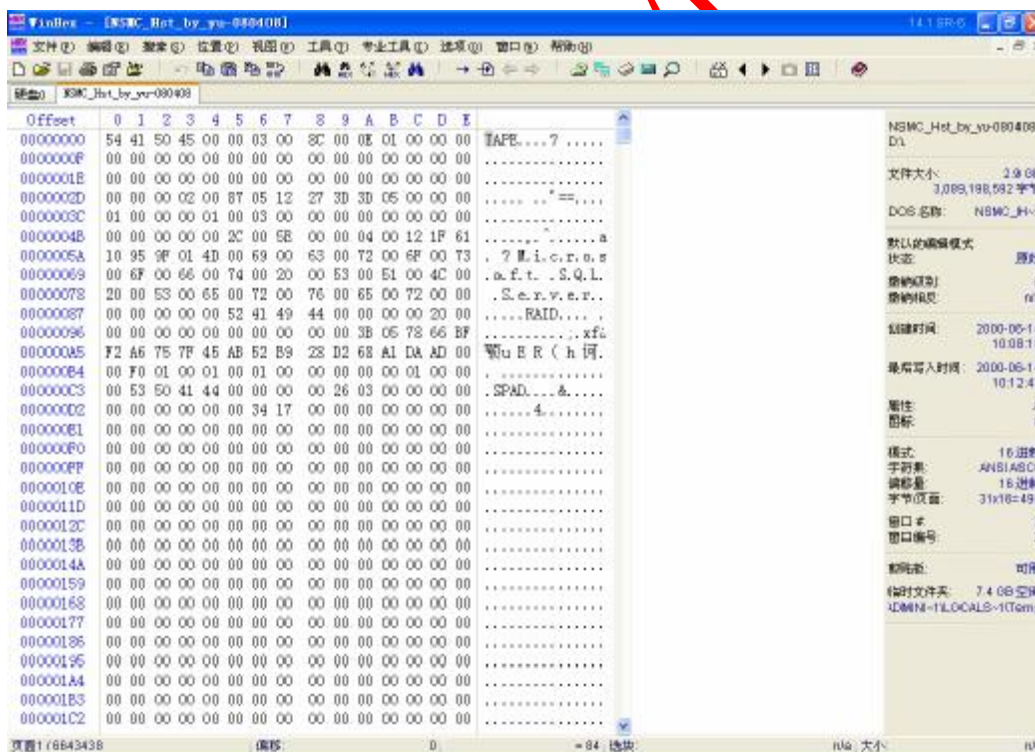
16 进制 / ANSI ASCII																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	+
80	€	•	,	f	”	...	†	‡	^	‰	Š	<	Œ	+	+	+
90	•	•	•	“	”	•	—	—	~	™	š	>	œ	+	+	ÿ
A0		i	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	–	®	—
B0	•	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

(16 进制值与 ANSI ASCII 字串的对应关系)

16 进制 / EBCDIC																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00					œ		†	+	-	+						
10					+	...		±			+					
20	€	+	,	f	”				~	%	Š	<	œ			
30	+	‘	“	”	.	-		~	™	š	>					
40			i	ø	£	¤	¥	¦	§	¨	[.	<	(+	!
50	&	@	³	<<	¬	-	@	-	°	±]	\$	*)	:	^
60	-	/	²	³	´	µ	¶	.	,	¹		.	%	-	>	?
70	º	>>	¼	½	¾	¿	À	Á	Â	Ã	:	#	@	‘	=	"
80	Ä	Å	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	é
90	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×		
A0	Ñ	~	s	t	u	v	w	x	y	z	Ò	Ó	Ô	Õ	Ö	×
B0	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	à	á	â	ã	ä	å	æ	ç
C0	{	À	B	C	D	E	F	G	H	I	è	é	ê	ë	ì	í
D0	}	J	K	L	M	N	O	P	Q	R	î	ï	ð	ñ	ò	ó
E0	\	ÿ	S	T	U	V	W	X	Y	Z	ò	ó	ô	÷	ø	ù
F0	0	1	2	3	4	5	6	7	8	9	ú	û	ü	ý	þ	ÿ

(16 进制值与 EBCDIC 元素符号之间的对应关系)

减少一列/增加一列子项仅仅是在编辑区显示上做更改，点击“减少一列”我们发现“F”列已经隐藏，每行偏移量也随之发生变化。如图所示:



增加一列亦是如此，通常情况 WINHEX 默认排列已经是最佳方案，不需要更改。

同步窗口子项很有用处，可以同时动态浏览多个操作对象相同位置的数据，被广泛应用于数据比对、代码分析、逆向工程中，而我们的 RAID 重组分析工作也是万万离不开它的。

一年来，由于工作繁忙，对数据恢复技术的研究已经是有心无力，更无暇顾及教程的更新和业内诸位兄弟的请教。在此还望大家多多原谅。

高某人会计专业出身，凭着对计算机科学的满腔热忱和一点点天赋，义无反顾地投入其中。刚入行的日子里，干过数据恢复、硬件维修、编程、软件架构、网络攻防、电子取证，如今又回到数据恢复，工作经历可谓五花八门。当时有人曾指责我三心二意，没有抓住数据恢复这个新兴产业一路拼杀下去，没有向大师的巅峰迈进。

今天，回首走过的路，我并不后悔，我更深刻的体会到计算机各分支学科间相辅相成的微妙关系，我原先的公司有人号称数据分析专家，却连一块硬盘都拆不下来；有人号称软件开发高手，承担着国家多重大的研发任务，却因为对 NTFS 一窍不通，而导致竞标答辩失败，毁了整个年度项目；有人号称信息安全系统分析师，却看不出一张拓扑图中显眼的错误。

看来，“术业有专攻”并不适用于计算机行业，计算机各个分支在走向融合，如果你忽视任何一部分，就有可能在追求梦想的脚步上停滞不前。

我要表达的是：不希望大家都成为“专业的数据恢复工程师”，而是展开计算机技术的多面手，让多种技能共同促进。学了驱动编程，你就发现数据恢复软件无非是一堆 API 函数和 ATA 指令的集合；学了操作系统内核，你就会明白，那些苦苦求来的珍贵文件系统资料，竟然是一组组已经开源的网上随处可见的操作系统数据结构的文字表达。当然，一切源于决心和努力。

最近，我在网上看到有人为数据恢复行业的前景担忧，其实这大可不必，只要你不断学习，境界超过大众，自然就跳出了那残酷的市场竞争，笑看凡尘。数据恢复还有很多未开辟的净土等着各位去探索，不要悲观。放弃的人其实就是被淘汰的人。我相信，业内的大部分人都和我一样二十岁出头，要说没有学习的时间和精力，还为时过早。青春就是用来学习的。

言归正传，今天开始的教程还是离不开 WINHEX，但是教学难度有所增加，如果你有编程基础，那学习过程将会非常轻松有趣。因为有研发任务，我不能保证会定期续写这个教程，但有时间就会写。

困惑的浪漫（高志鹏）

第一讲 语言和变量（1）

学过 C 的读者都知道，一段完整的，功能化的 C 代码，起码由一组函数和数据结构组成。如果升华到 C++、C#、JAVA 等就会引入更高级的类型概念。对类型的强制规范和区别，是 C 系列语言的根本特征，所以它们也被称为“强类型语言”。

先从最低级的 C 说起，我们最先学习的往往是编译器自带的基础类型，如 `int`，`long`，`char` 等，更多的人称它们为变量，因为在学习 C 的阶段，很少有人会有明晰的“类型概念”。到了 C++时期，MFC 编程不可避免地用到类，后来 C#作为纯粹的面向对象语言，更是把类的作用发挥到了极致。有人说，面向对象比面向过程简单，这是个误区。只能说，面向对象比面向过程的重复利用率高，出错率低，无形中加快了开发速度，给人们留下了“简单”的印象。其实，面向对象使人们把精力更多放在功能设计上，是一种更高级的思维方式。我可以负责任地说：一个在指针和内存的海洋里游走的人永远摆脱不了程序员的帽子，而一个与类打交道的人则很可能朝架构师的行列迈进。因为，面向过程的目的是让代码编译通过，大量的繁琐工作（调试、垃圾回收）只是为了保证代码的正确率，而面向对象则对功能优劣，开发效率等实用方向关注更多。

随着 WEB 编程的兴起，更多人开始关注脚本语言，脚本的本质我不想分析太多，但有一点是共同的，它们都是依附于特定平台，为完成特定功能应运而生的。

Perl 就是典型，它原本是处理大批量字符所用，但经过几代黑客不懈努力，也具备了和 C 并驾齐驱的强大功能。而我们的 WINHEX 脚本，就是为了数据的批量搜索、定位、修改所用。这里的特定平台就是 WINHEX，特定功能就是服务于底层数据，对我们来说，就是将原本复杂繁琐的手工数据分析过程智能化，

这和开发一个全新的数据恢复软件是一样的。我们只需一次告诉 WINHEX 要干

什么，剩下的重复工作就由它全权代劳。

现在，我们承认 WINHEX 脚本也是编程语言家族的一种，那么它和我们现在熟知的开发语言又有何异同呢？

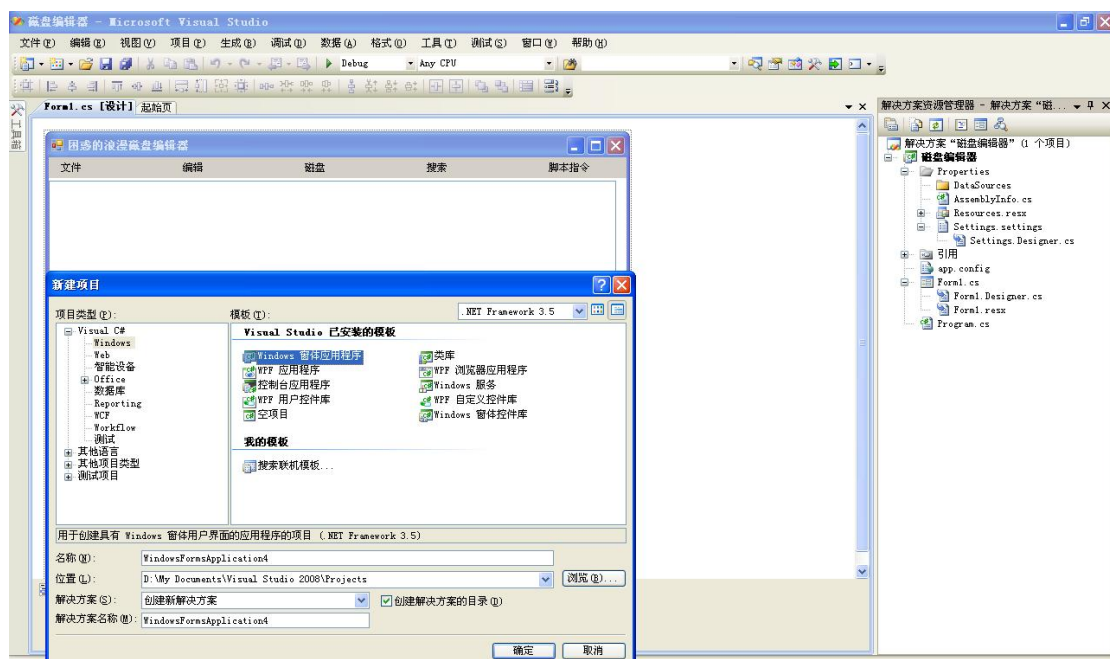
首先，WINHEX 是由高级语言开发的，所以 WINHEX 脚本是对 WINHEX 内部函数功能的高度浓缩。

其次，WINHEX 脚本一旦脱离了底层数据，就丧失其根本目标。也就是说，没有硬盘，没有文件，没有内存，这个语言就失去了运行基础。

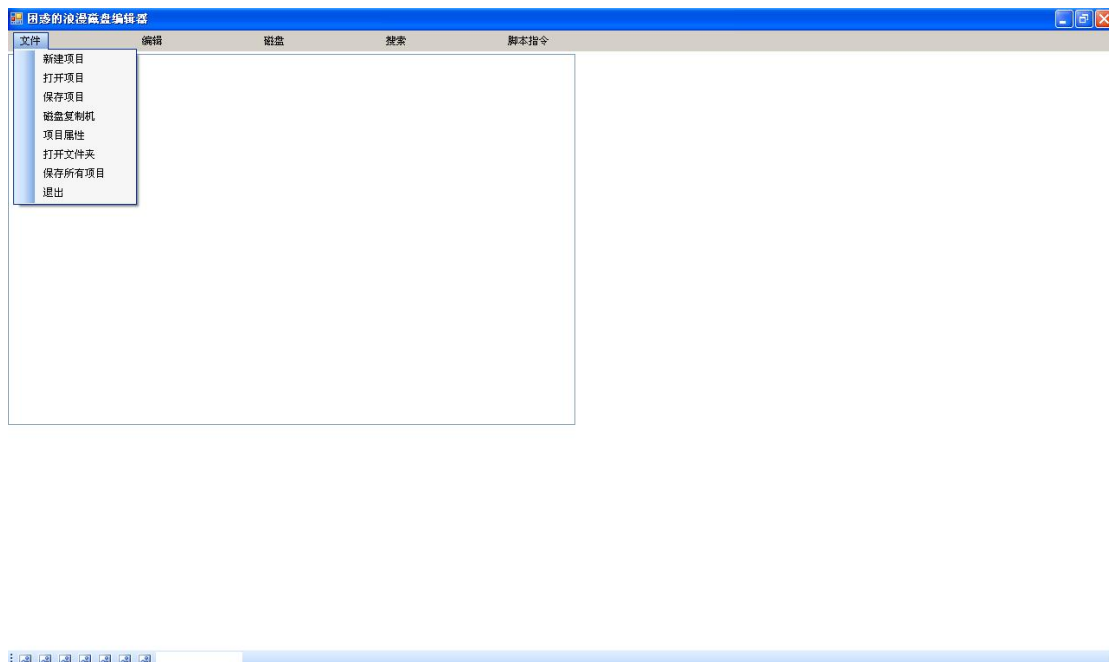
再次，WINHEX 脚本是 WINHEX 操作步骤的指令化和批处理化。这个大家应该深有体会，脚本使我们的工作效率大大提高。

有人拿 ENCASE 脚本和 WINHEX 脚本作比较，鼓吹 ENCASE 是“取证之王”，这点我实在不能苟同，ENCASE 那些花花肠子都是建立在部分函数公开的基础上，功能薄弱，而且，没有 C++编程基础的人是不可能掌握的，已经丧失了脚本语言的方便性。这就是 ENCASE 在中国粉丝较少的根本原因。

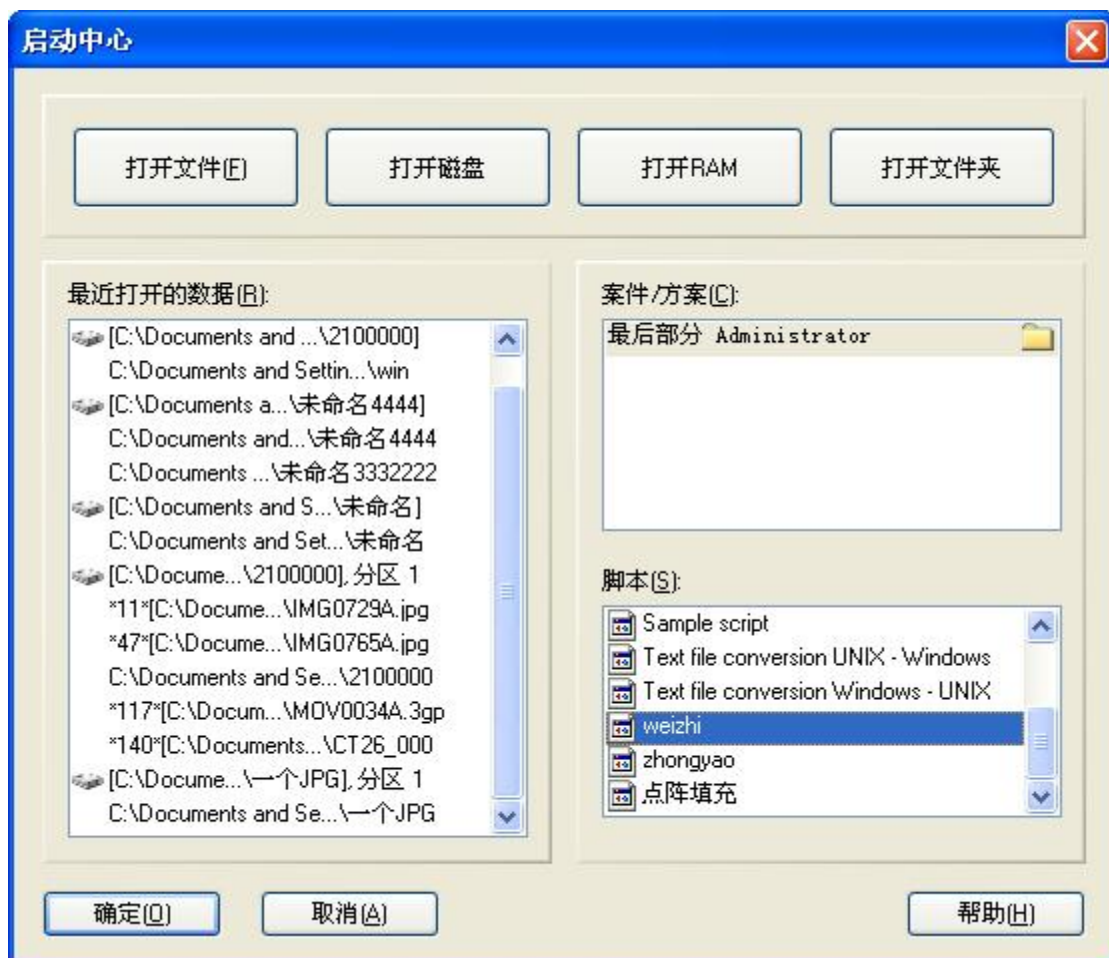
本教程中，我会拿 C#（本想用 C，但 C 的表达力较差，很多人不能忍受而丧失学习兴趣）和 WINHEX 进行交叉讲学，彻底激发您对编程的兴趣和向往。



（C#集成编译开发环境）



（C#可以开发出类 WINHEX 平台）



（WINHEX 脚本编辑环境）

上面几幅图，反映了两种语言的开发环境和平台等级：C#基于.NET 和操作系统，脚本只基于 WINHEX，而 WINHEX 基于操作系统。下面是一段类似于冒泡排序法的科学计算数据处理脚本，我们不关心该代码本身的算法和功能，只分析它的构成。从语句中，我们看到很多语法功能，有变量声明的，有设置跳转标签的，有循环递增的，有字符串搜索的，有地址位移的，更有一个标准的循环体结构和循环常量。脚本中还是脱不开 C 系列的影子，用大括号进行了简单的，也是唯一的层次分类。该脚本既有与 C#相同的基本控制流语句如 `inc love1`、`assign love1 0x00000000` 等，也有 WINHEX特有的，经过多重高级封装的 `find---iffound` 语句。编程语言一般有两个基本功能，处理对象的和处理对象地址的（如数组下标，结构指针）。两种功能都在 WINHEX 中有体现。`assign love1 0x00000000`、`read love2 8` 等都是偏重处理对象的，`goto 0x0000`、`move -12` 都是偏重处理地址的，当然，对象和地址可以应需要而互相转换。对象可以成为另一个对象的地址，地址也可以成为另一个地址所指向的对象。`assign love1 0x00000000`

```

{
//Label ContinueHere
inc love1
find love1 down
iffound
assign acc CurrentPos
move -8
read love2 8
else
//inc love1
goto 0x0000
//JumpTo ContinueHere
endif
ifequal love2 0x88FFFFFFFFFFFFFFFF
move -12

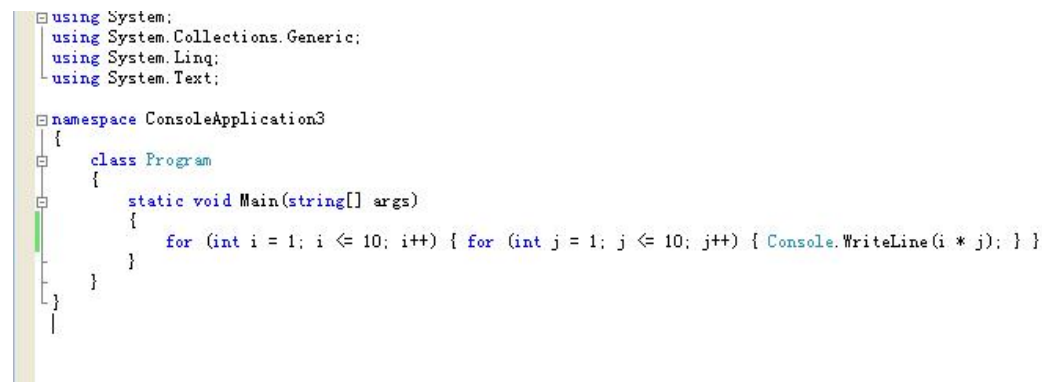
```

```

read love3 528
nextobj
write love3
nextobj
//goto 0x0000
//JumpTo ContinueHere
else
//inc love1
goto (acc+2)
JumpTo ContinueHere
//endif
}[unlimited]

```

所以，我们不难看出，WINHEX 脚本编程就是对“对象和地址”的活用，而且是不加修饰的，最基础的活用。对象和地址都是数学量，我们还可以进一步分为变量和常量。我们知道 `assign love1 0x00000000`、`read love2 8` 都是变量声明的语法，其中 `assign` 是关键字，`love1` 是变量标识符，`0x00000000` 是标识符所代表的实际数据。WINHEX 里很少用到常量的概念，更多的是硬编码如 `goto 0x0000`。令人兴趣盎然的是，脚本中竟然还有 `ENDIF` 这种条件判断语句。循环十分有用，可惜的是 WINHEX 脚本不支持嵌套你不可能创造出：`for (int i = 1;i<=10;i++) { for (int j = 1;j<=10;j++) {Console.WriteLine(i*j)}}`这样的语句来。

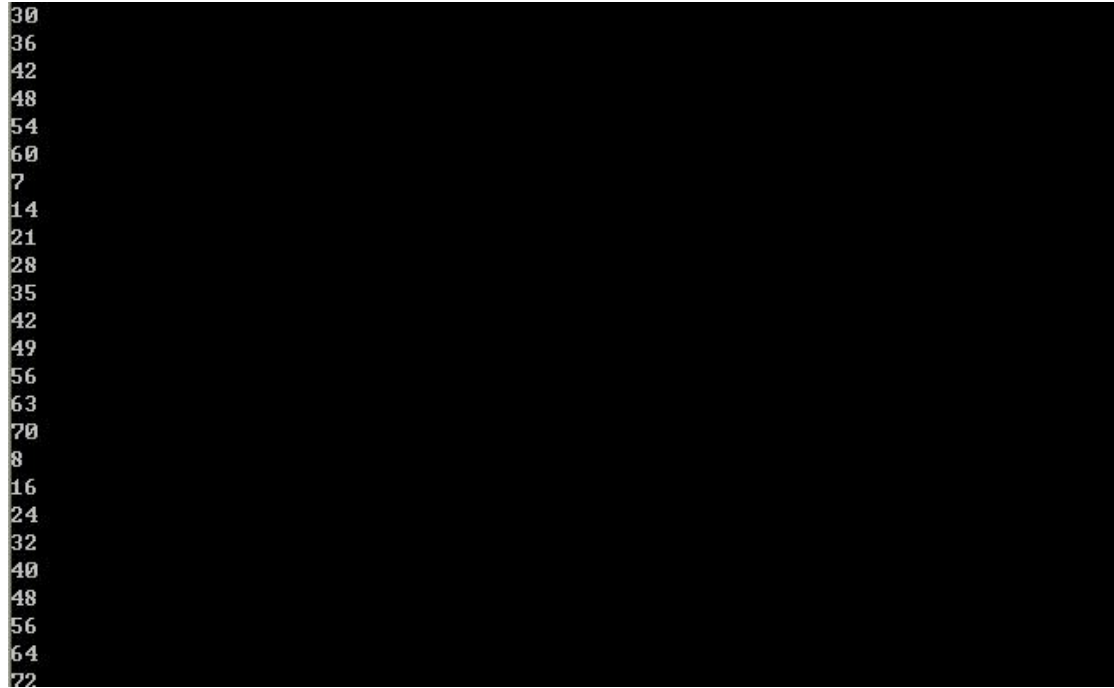


```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 10; i++) { for (int j = 1; j <= 10; j++) { Console.WriteLine(i * j); } }
        }
    }
}

```



也就是说：

。 。 。 。 。 。

Ifound

Find

。 。 。 。 。 。

类似结构是错误语法，逻辑上，找，如果找到再找是 WINHEX 所不允许的。而且大括号只能用于循环，不可滥用。循环自然也不能嵌套。

因为 WINHEX 有偏移量寻址的功能，所以移动地址可以改变当前操作的对象，并以字节为基本单位。

现在回到一开始我们所说的变量的概念。研究 WINHEX 说明书脚本一章，大家一定会发现满篇都会出现 **MyVariable** 这个词组，其实它就代表一个自定义变量。我们可以为变量赋值，可以用变量进行常规运算，可以用它顶替循环次数。下面的示例中，我们先声明一个变量 **love1** 并将 **100** 这个数字存进该变量中。循环时，我们直接把 **love1** 顶替到循环次数的位置上去，和直接填写 **100** 这个数字是一样的，但你一定会感到，变量蕴含着更方便灵活的处理方式和力量。


```
assign love1 100
```

```
{
```

```
write 0x55aa
```

```
}[love1]
```

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000010	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000020	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000030	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000040	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000050	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000060	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000070	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000080	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
00000090	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
000000A0	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
000000B0	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	55	AA	U@U@U@U@U@U@U@
000000C0	55	AA	55	AA	55	AA	55	AA	00	00	00	00	00	00	00	00	U@U@U@U@U@U@U@

Read 也是 **WINHEX** 脚本中声明变量的有效方式，它直接从操作对象数据中获取变量内容，我们可以指定某段连续存储的字节到变量中，这给我们“搬运数据”带来便利，比如你想把 **DBR** 备份移植到 **DBR** 位置，该语句肯定是首选。**C** 系列语言中，要完成 **WINHEX** **Read** 功能，必须用数组类型来创建一个叫缓冲区的存储空间，操作难度自然要大得多。

C#中创建变量就是类型实例化的过程，其中，必须要具备类型名称、数据结构和约束条件三大部分，例如

```
Short 困惑的浪漫 = 24;
```

Short 就是类型名称，代表短整型，16 位占用两个字节空间，是行为和结构。每种类型分管着各自的作用范围，类型间不轻易发生联系，除非必要时使用“转换”法则。**C#**中的类型其实就是一种模板，这和 **WINHEX** 模板的道理如出一辙。**WINHEX** 中，模板可以限定数据的属性，位置和类型，**C#**也一样，创建一个类，就势必要创建它的数据成员和函数成员，实例化的时候，这些成员就会按照已经设定好的特性去完成数据处理。

C#光整型就下面许多种，和 **C**，**C++**高度相似，此外，还有浮点型，字符型，布尔型，字符串型和基类型。**WINHEX** 用一个 **assign** 就可以遇佛杀佛。

sbyte	-128 到 127	有符号 8 位整数
byte	0 到 255	无符号 8 位整数
char	U+0000 到 U+ffff	16 位 Unicode 字符
short	-32,768 到 32,767	有符号 16 位整数
ushort	0 到 65,535	无符号 16 位整数
int	-2,147,483,648 到 2,147,483,647	有符号 32 位整数
uint	0 到 4,294,967,295	无符号 32 位整数
long	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	有符号 64 位整数
ulong	0 到 18,446,744,073,709,551,615	无符号 64 位整数

WINHEX 脚本很大的缺陷就是无法创建完整的自定义类型（或引用类型）。这个问题我们在以后的课程中予以讨论。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            int love1;
            string love2;
            love1 = 24;
            love2 = "困惑的浪漫今年：";
            Console.WriteLine("{0}{1}岁", love2, love1);
            Console.ReadKey();
        }
    }
}
```



我们可以尝试用 **WINHEX** 脚本的形式把上述代码重写：

assign love1 24

assign love2 "困惑的浪漫今年:"

assign love3 "岁"

messagebox love2

messagebox love1

messagebox love3



看上去似乎比汇编还麻烦，没办法，除了运算，WINHEX 脚本似乎无法在一行语句内处理数字和字符串的混合运算。下面的代码更复杂一些但似乎有些进步：

assign love1 24

assign love2 "困惑的浪漫今年:"

assign love3 "岁"

write love2

write love1

write love3

move -18

read love4 18

messagebox love4



字符串的问题解决了但数字问题又陷入僵局，不要怕，功夫不负有心人，我们可以利用 INTtoSTR 语句来实现整型与字符型的互转：

assign love1 24

inttostr love5 love1

```
assign love2 "困惑的浪漫今年:"
```

```
assign love3 "岁"
```

```
write love2
```

```
write love5
```

```
write love3
```

```
move -19
```

```
read love4 19
```

```
love4
```

我们先将整形变量 love1 转换为字符型保存在 love5 中，然后依次写入当前位置，再将它们全部读入变量 love4，然后用 messagebox 显示出来：



成功了，相信通过今天的课程，大家对 WINHEX 变量已经有了深刻的认识。

上面这个小程序和数据恢复无关，甚至和 WINHEX 脚本的初衷无关，但它在极力模仿 C#，以求得出和 C# 同样的运行结果，它证明了只要动动歪脑筋，我们甚至可以用 WINHEX 实现部分与某些高级语言相同的功能。

WINHEX 脚本的功能限制是可以迂回绕开的，为了证明和达到这个目的，以后的探索之路将越发艰难，课程难度自然会日益增加。

第二讲 语言和变量（2）

困惑的浪漫

程序的本质是数据的处理。就拿我们平时接触最多的计算机显示单元来说，其最初形式便是存储在计算机内存中的 0，1 数据流。即使我们轻轻移动鼠标，也会大量改变内存中的数据。

对数据进行处理的前提是让你的计算机得到数据，也就是我们时时挂在嘴边的“存储”。我们都知道，操作系统存储的基本计算单位是字节，而我们热爱的 WINHEX 恰恰是将字节作为处理对象，从最细致的角度来观察计算机存储系统的健康状况。

但是 WINHEX 的眼中，数据是不加修饰的赤裸裸的二进制流，这也为我们呈现出一个光怪陆离，复杂纷繁的数字世界，很多人在这座迷宫中叹而却步。手工数据恢复入门历程之艰辛可想而知。我们必须牢牢记住哪些数字是系统命脉，哪些数字是陷阱伪装，哪些数字是恶意干扰，哪些数字是搜寻目标，它们没有特征，没有标记，只能靠我们多年积攒的经验和丰富的文件系统和文件编码知识去探寻、解析。这就是典型的逆向思维技术，就好比我们要从一个雕塑身上清理出工匠创造时的思路灵感，难度不言自明。

在十几年前，数据恢复还是一种不为人知的“黑客手段”，因为需要分析操作系统内核，动用了大量的静态反汇编工具，WINHEX、W32ASM 都在当时崭露头角，曾经一度成为高手们不可或缺的利器。

既然 WINHEX 代表一种逆向思维，那么学习 WINHEX 就得反其道而行之，不了解它的根本运行机理，就不能掌握 WINHEX 与操作系统互动的过程，学习经历自然会痛苦死板，没有成效。

前一讲已经指出 WINHEX 脚本语言是对 WINHEX 内部主要函数功能的高度凝缩。脚本其实就是一张清晰的 WINHEX 内部结构图。WINHEX 能做什么不能做什么有什么忌讳什么限制在这里都一目了然。脚本做不到的，WINHEX 主界面的菜单里自然也不会有那项功能。所以，学习脚本就是在学习 WINHEX 的根本运行机理。WINHEX 脚本的语法格式，如参数、运算符很大程度接近于 C 系列语言。这些我们从以后的 WINHEX API 学习中就可以知晓。

今天我们继续用脚本与 C#同步对比的方式，来让大家了解 WINHEX 脚本语言的“编程极限”。

我们知道 C#是一种块结构语言，代码块可以任意包含多条语句。层次分明的块将 C#内部划分成结构级别和作用域。从程序集、类、函数到控制流，每一集都离不开“块”。而 WINHEX 脚本则大大不同，除了循环，似乎用不到那么多块来规范书写，这无形中为阅读代码增加了难度。WINHEX 模板则有些不同，对变量的支持细致而丰富，更多的使用了“块”。但是模板的处理面太窄，没有功能扩展的余地，只能静态地给出结果，缺乏“程序气质”，对我们这几讲的学习没什么帮助。

C#中空格、回车、制表符通通称为空白字符，编译时不予考虑，这样设计格式时就具有很大余地。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10;
            int b = 10;
            Console.WriteLine(          a * b);
            Console.WriteLine(a * b);
            Console.WriteLine(a *          b);
        }
    }
}
```

上面代码的结果都一样，空格并没有造成任何影响，而且 C#在多数时候会自动纠正上面的不雅。

```
100
100
100
请按任意键继续. . .
```

C#中分号代表一句代码结束，无论这些代码是否书写在一行中，WINHEX 必须靠换行来区分语句，不需要分号：

```
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10;
            int b = 10;
            Console.WriteLine(a * b); Console.WriteLine(a * b); Console.WriteLine(a * b);

        }
    }
}
```

上面例子中三个控制台输出函数并用分号隔开，不影响任何显示结果。但是影响美观，降低了可读性。

```
Console.WriteLine("困惑的浪漫正在续写国内最强大的 WINHEX 教程!!!");
```

```
困惑的浪漫正在续写国内最强大的WINHEX教程!!!
请按任意键继续. . .
```

但是输出字符串则不同，因为无论哪种字符编码，空格都是有意义的，它会被识别出来。

```
Console.WriteLine("困惑的浪漫正在续写          国内最强大的 WINHEX 教程!!!");
```

```
困惑的浪漫正在续写          国内最强大的WINHEX教程!!!
请按任意键继续. . .
```

我们再回到 WINHEX 中：

`write "困惑的浪漫正在续写国内最强大的 WINHEX 教程！！！”`

按照 Unicode 码占用两个字节来计算此时应该在编辑区输出 46 个字符：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	C0	A7	BB	F3	B5	C4	C0	CB	C2	FE	D5	FD	D4	DA	D0	F8	□ □ □ 上 □ □
00000010	D0	B4	B9	FA	C4	DA	D7	EE	C7	BF	B4	F3	B5	C4	57	49	□ □ □ □ □ □ □
00000020	4E	48	45	58	BD	CC	B3	CC	A3	A1	A3	A1	A3	A1	00	00	□ 壤 □ □ □ □ □
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

如果改写成，下面这样，看看会有什么结果：

`write "困惑的浪漫正在续写国内最强大的 WINHEX 教程！！！”`

C0	A7	BB	F3	B5	C4	C0	CB	C2	FE	20	20	20	20	20	20	20	□ □ □ 上 ↑ ↑ ↑
20	20	20	20	20	D5	FD	D4	DA	D0	F8	D0	B4	B9	FA	C4		↑ ↑ □ □ □ □ □ □
DA	D7	EE	C7	BF	B4	F3	B5	C4	57	49	4E	48	45	58	BD		□ □ □ □ 壤 义 □ □
CC	B3	CC	A3	A1	A3	A1	A3	A1	00	00	00	00	00	00	00	00	□ □ □ □ i

字节中多了许多 `0x20`，这些就是刚刚定义的空格，它们不仅被识别，而且没有用 `0x00` 来表现。学过 C 的读者都明白，在 C 中 `char` 和 `int` 型是可以直接互换互通的，但是在 C# 中则不然，需要经过转换：

namespace ConsoleApplication3

```
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 254;
            Console.WriteLine((char)a);
        }
    }
}
```

请按任意键继续...

熟悉编码的人都能看出，C# 中的 `char` 表示一个 Unicode 字符。

此外，注释是所有程序语言都共有的基本功能，注释能帮程序员理清思路，实

时标注思维成果，在调试代码时还能起到屏蔽的作用。

```
// // class Program
// //{
//     static void Main(string[] args)
//     {
//         byte a = 254;
```



```
//          Console.WriteLine((char)a);

//      }
//  }
```

WINHEX 中的注释和 C#相仿，作用也一至，凡是打了双斜杠的那一行都不在执行之列。

```
//{
/move 16
/read love1 512
/nextobj
/write love1
/nextobj
/}[unlimited]
```

当然 C#还支持另外两种注释方式：/*，*/和///(xml)。

```
namespace ConsoleApplication3
{
    /* class Program
    {
        static void Main(string[] args)
        {
            byte a = 254;
            Console.WriteLine((char)a);

        }
    }*/
}
```



我们看到，若套用在 WINHEX 上则会出现错误。

此外，大小写也是应该注意的文体，C#对大小写很敏感，无论是关键字还是标识符，大小写误用都会导致严重后果：

```
class Program
{
    static void Main(string[] args)
    {
        byte a = 254;
        CONSOLE.WRITELINE((char)a);
    }
}
```

错误 1 当前上下文中不存在名称“CONSOLE” D:\My Documents\Visual

Studio 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs 13

13 ConsoleApplication3

```
class Program
{
    static void Main(string[] args)
    {
        byte a = 254;
        byte A = 245;
        Console.WriteLine("{0} {1}",a,A);
    }
}
```



```
254 245
请按任意键继续。 . .
```

可以看出字母 a 的大小写是两个不同的标识符，代表了不同的变量。

WINHEX 可不同，大写 LOVE 和小写 love 代表一个变量：代码中先将 100 付给 LOVE，再将 130 赋给 love,所以最终是 130 替换了 100，显示为 130；

assign LOVE 100

assign love 130

messagebox love



可以再作演示：

assign LOVE 200

assign love 240

messagebox LOVE



但是 WINHEX 的关键字也没有大小写限制，这样写完全正确：

ASSIGN love 200

ASSIGN love2 180

MESSAGEBOX LOVE



但是要注意一点，命名最好始终按照一种风格进行，主要是为了避免错误，提高可读性。

WINHEX 中的逐字指定是默认的，也就是说遇到路径类的字符串用一个斜杠即可表示：

Create "D:\My File.txt" 1000

C#默认必须双斜杠才可以，除非在字符串前端加@。这是由于转义序列符的缘故，WINHEX 在函数参数设计时就定义好了这些，所以我们不必关注。

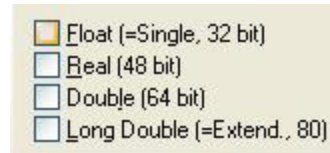
我们已经知道，WINHEX 只有一种变量声明方式，就是将变量内容赋予变量名。

所以 WINHEX 变量不存在转换一说，把数字用作字符串时强加双引号即可。

```
write 123
```

```
write "123"
```

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C
D	E	F											
00000000	7B 31 32 33 00 00 00 00				00 00 00 00 00 00 00 00				{123.....}				



上图是 WINHEX 数据解释器的小数部分，我们甚至看到比 C 系类还要丰富的

浮点类型，当然这些都是用程序定义的。但是它的上限远不如 C# 宽广。

float	±1.5e-45 到 ±3.4e38	7 位
double	±5.0e-324 到 ±1.7e308	15 到 16 位

类型	大致范围	精度	.NET Framework 类型
decimal	±1.0 × 10 ⁻²⁸ 到 ±7.9 × 10 ²⁸	28 到 29 位有效位	System.Decimal

C# 中有高达 128 位的高精度浮点型 **decimal**，用于行业运算，WINHEX 中基本不可能用到这种占 16 个字节的参数。

此外，布尔型变量隐藏在 WINHEX 的条件判断语句中，C# 则表现为 **true** 或 **false**，可以直接运算使用。布尔是逻辑代数的基础，大家一定要好好掌握。

我们用一个较复杂的例子来总结这两讲涉及的变量知识：

```
lass Program
{
    static void Main(string[] args)
    {
        int 工资 = Convert.ToInt32(Console.ReadLine()), 奖金 =
        Convert.ToInt32(Console.ReadLine()), 回扣 = Convert.ToInt32(Console.ReadLine()), 总收入;
        总收入 = 工资 + 奖金 + 回扣;
        Console.WriteLine("困惑的浪漫期望的总收入是每月稳定的工资{0}元。\\n可观的科研奖金
        {1}元。\\n黑心的回扣{2}元！\\n令人惬意的总收入{3}元\\n当然这只是梦想！", 工资, 奖金, 回扣, 总收入);
        if(总收入>300000)
            Console.WriteLine("好像发财了!");
        else
            Console.WriteLine("好像在做梦啊！");
        Console.ReadKey();
    }
}
```



```

    }
}

10000
10000
5000
困惑的浪漫期望的总收入是每月稳定的工资10000元。
可观的科研奖金10000元。
黑心的回扣5000元!
令人惬意的总收入25000元
当然这只是梦想!
好像在做梦啊!

```

这个例子要求手工输入三个值，然后得出困惑的浪漫期望的总收入，然后判断是做梦还是发财了。

程序一开始我们就定义了四个变量，工资、奖金、回扣、总收入。前三个变量需要手工输入赋值，由于输入时是字符串，必须强制转化为整型。总收入当然是子收入之和。分别输出后，我们用布尔运算来做条件判断，如果总收入超过 300000 则发财了，如果没到那就是作富人梦。当然这个例子大家请一笑置之！

我们要做的是用 WINHEX 脚本将其重新组织一遍，难点是输入和输出怎么表达，WINHEX 脚本这两项能力都不强。但我们可以利用 GetUserInputI 指令来完成整数的输入：

```

messagebox "困惑的浪漫期望的总收入是:"
GetUserInputI love1    "每月稳定的工资:"
GetUserInputI love2    "可观的科研奖金:"
GetUserInputI love3    "黑心的回扣:"
assign love4    (love1+love2+love3)
messagebox "令人惬意的总收入:"
messagebox love4
ifGreater love4 300000
messagebox "好像发财了!"
else
messagebox "好像在做梦啊!"
endif

```



Love4 作为总收入变量要注意数学表达式的语法要求（下一讲），ifGreater 完全起到了大于小于符号的作用。看不明白的朋友请尽快补习 WINHEX 脚本基础语法，此教程不会详述的。

上面这个脚本还不完美，尤其是每个金额结尾都缺少一个“元”字让人很不痛

快，很简单，用我们上一讲展示的输出技巧即可。

```
messagebox "困惑的浪漫期望的总收入是:"
```

```
GetUserInputI love1 "每月稳定的工资:"
```

```
write "每月稳定的工资:"
```

```
inttostr love9 love1
```

```
write love9
```

```
write "元"
```

```
move -23
```

```
read love6 23
```

```
messagebox love6
```



剩下部分请读者们自行完成，当然，要和 C# 一样精准顺利是不可能的，这毕竟绕了很大弯子，而且 love6 的范围控制（move -23）不能一次完成，只能不断试验直到最佳。

希望这一讲能让各位出现质的提升！再见！！！！

WINHEX 高级专题第三讲

实战 WINHEX API 函数(1)

困惑的浪漫(高志鹏)

初看这个题目,大家一定感到非常诧异,几个月前,大家跟随我制定的学习路线,了解了语言变量、表达式的初步知识,更学习了如何活用 WINHEX 脚本,使之拥有实现某些高级编程语言特性的方法.当然,这些特性都是最初级最简单不过的了.变量、表达式对程序本身来说是重要的地基,但对编程的学习框架来说,只是微不足道的入门理论.活用变量、表达式,能给 WINHEX 脚本带来脱胎换骨的变化,但是却无法使其提升到结构化编程的更高层级---函数.今天我们就迈上一个更高的台阶,如果各位兄弟感到吃力是可以理解的,但是不能轻言放弃,还是那句老话只要努力,就没有困难.

首先,要明确一点,本讲内容是磁盘函数编程的入门知识,但不是函数编程语法的入门知识,语法大家要背后下苦功.相信学习过程艰苦而漫长,我选择 WINHEX API 为入门素材是因为它是函数体系中结构比较简单的,更容易学习.运用 WINHEX API 只是利用别人的函数完成自己的功能,本人最终的目的是让各位拥有独立改造、编写函数功能的能力,继而拥有设计类型,接口平台的能力,从而带领大家创造一套功能更强大,智能程度更高的全新 WINHEX 来.

WINHEX 有三个指令接受接口:

- 一、 基于界面操作的:也就是我在《WINHEX 困惑的浪漫教程》中所演示的窗体交互、菜单点击、文本修改等“傻瓜操作”。当然，绝大部分 WIN32 程序都有此功能。
- 二、 基于脚本解释的: WINHEX 脚本大家也学习得够多了，如果设计得当，功能是非常强大的，但它毕竟是生存于 WINHEX 大平台之上的，有局限性。
- 三、 基于 API 的: 也就是应用程序接口，往往是软件厂商为二次开发而预留的。WINHEX API 就是一座概括 WINHEX 基本功能的古老函数库。

但是，无论如何，这三种接口最终都会归于一体: WINHEX 元函数。也就是 WINHEX 作者编写 WINHEX 的时设计的功能函数。简单的说，“Open D: ”

和菜单中的“打开逻辑磁盘”实际调用了同一个元函数，而这个元函数的深层实质上又调用了操作系统 API。这些都是我以后要讲解的高深内容，这里不再细谈。

WINHEX 官方公布的 API 有四十个左右，它们的功能和形态与脚本非常相似，使用上又有颇多限制，比如必须是专家及以上版本，或者是个人版额外购买了 API 授权序列号。可惜的是，由于祖国的盗版事业，这些版本我们手头都有，不影响学习。

WINHEX API 国内没有学习资料，但这并不代表它们高深，优秀的程序员只需观察其函数参数就可以判断出它们的使用方法，何况官方还给了一份简介文档。下面我们就按照官方给出的顺序，一一详解，本讲我们主要关注前三个函数，也是最重要的三个函数。

大家先观察下面的代码：

```
.....{  
  
    public partial class Form1 : Form  
    {  
  
        public Form1()  
        {  
            InitializeComponent();  
        }  
  
        private void button1_Click(object sender, EventArgs e)  
        {  
            int a = 1;  
            int WINHEXSTARTNUM = WINHEXAPI.WHX_Init(a);  
            MessageBox.Show(WINHEXSTARTNUM.ToString(), "返回指示信  
号:");  
  
            if (WINHEXSTARTNUM == 1)  
            {
```

```
// WINHEXAPI.WHX_OpenEx("80h", 0x00000001);
if (WINHEXAPI.WHX_Open("80h") == true)
{
    //WINHEXAPI.WHX_Create(@"E:\困惑的浪漫.txt",
1000000);

    WINHEXAPI.WHX_Find("0xEB52904E","Down
SaveAllpose");

    if (WINHEXAPI.WHX_WasFound() == true)
    {
        MessageBox.Show("找到了!!!");
    }

    WINHEXAPI.WHX_Done();
}
else
{
    WINHEXAPI.WHX_Done();
}
}
else if (WINHEXSTARTNUM == 0)
{
    MessageBox.Show("常规错误!!!", "高志鹏警告您:");
}
else if (WINHEXSTARTNUM == -1)
{
    MessageBox.Show("未准备好!!!", "高志鹏警告您:");
}
else if (WINHEXSTARTNUM == -2)
```

```
        {
            MessageBox.Show("版本不正确!!!", "高志鹏警告您:");
        }
        else if (WINHEXSTARTNUM == -3)
        {
            MessageBox.Show("未取得充足许可权限!!!", "高志鹏警告
您:");
        }
        else
        {
            MessageBox.Show("WINHEXAPI异常,请仔细检查!!!", "高志
鹏警告您:");
        }
    }
}

class WINHEXAPI
{

[DllImport("whxapi.dll", SetLastError=true, CharSet=CharSet.Ansi)]
    public static extern int WHX_Init(int APIVersion );

[DllImport("whxapi.dll", SetLastError=true, CharSet=CharSet.Ansi)]
    public static extern bool WHX_Done();

[DllImport("whxapi.dll", SetLastError=true, CharSet=CharSet.Ansi)]
    public static extern bool WHX_Open(string lpResName);

[DllImport("whxapi.dll", SetLastError=true, CharSet=CharSet.Ansi)]
    public static extern bool WHX_Create( string lpPathName,int
```

```
Size);
```

```
[DllImport("whxapi.dll",SetLastError=true,CharSet=CharSet.Ansi)]  
    public static extern bool WHX_OpenEx(string lpResName, int  
Param);
```

```
[DllImport("whxapi.dll",SetLastError=true,CharSet=CharSet.Ansi)]  
    public static extern bool WHX_Find(string lpData,string  
lpOptions);
```

```
[DllImport("whxapi.dll",SetLastError=true,CharSet=CharSet.Ansi)]  
    public static extern bool WHX_WasFound();  
}  
}.....
```

里面牵扯了两个类，一个用于执行函数，一个用以声明DLL中的内容。这里要注意，使用WINHEX API尽量采用安装版的WINHEX，否则可能会有不确定的错误因素，或者无效。API DLL尽量拷贝至SYSTEM32或本程序所在目录下.也可以显示给出DLL路径,但这不利于程序移植和维护,注意使用的DLL有VB版,VC版等几种,这里我们用的是VC DLL.

`int WINHEXSTARTNUM = WINHEXAPI.WHX_Init(a);`这一句是WINHEX API的首次使用,等号右边就是API文档提到的第一个函数:初始化函数:

<code>int WHX_Init(int <i>APIVersion</i>);</code>	<div>2 Success (limited)* 1 Success 0 General error WinHex installation not -1 ready -2 <i>APIVersion</i> incorrect -3 Invalid or insufficient</div>
---	--

		license
--	--	---------

函数声明原型和异常反馈列表.

其中int是函数的返回类型,WHX_Init是函数名, int APIVersion是该函数的形参列表(这里看不懂的赶紧去学C的基本知识).形参是要在函数执行时予以赋值的,否则编译器会显示一个局部变量错误,程序无法通过. APIVersion代表了该DLL的版本号,官方给出的版本号始终为1.所以int a = 1;WINHEXAPI.WHX_Init(a)中我们将赋值为1的变量a带入形参,就是为了将1这个数传入函数中.此时a叫做实参,也就是拥有了实际内容的参数.

但是外部的DLL我们需要额外予以声明

```
[DllImport("whxapi.dll",SetLastError=true,CharSet=CharSet.Ansi)]
```

```
public static extern int WHX_Init(int APIVersion );
```

我们用了平台调用的技术,注意WINHEX的基准字符为Ansi格式,所以我们要以CharSet=CharSet.Ansi来传入.否则函数中的字符串全都会出现缺损.

WHX_Init是所有WINHEX API中最重要的,起到进入准备状态和版本认证的作用,如果不初始化而直接调用其它API,整个程序会出现不可预知错误.

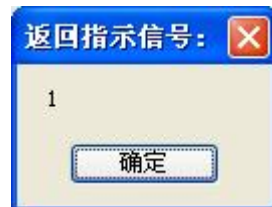
使用WHX_Init也有条件:

WHX_Init必须用于WINHEX彻底关闭的状态,连任务管理器中也不能存在WINHEX.EXE进程.如果因此导致初始化失败,会返回WinHex installation not ready错误,也就是返回值为-1(见文档中的错误列表)的错误.返回值-1也就代表了上面代码中int WINHEXSTARTNUM = WINHEXAPI.WHX_Init(a);等号左边的WINHEXSTARTNUM整型变量得到的值是-1,根据我设计的条件判断结构,应该给出警告并终止程序运行:

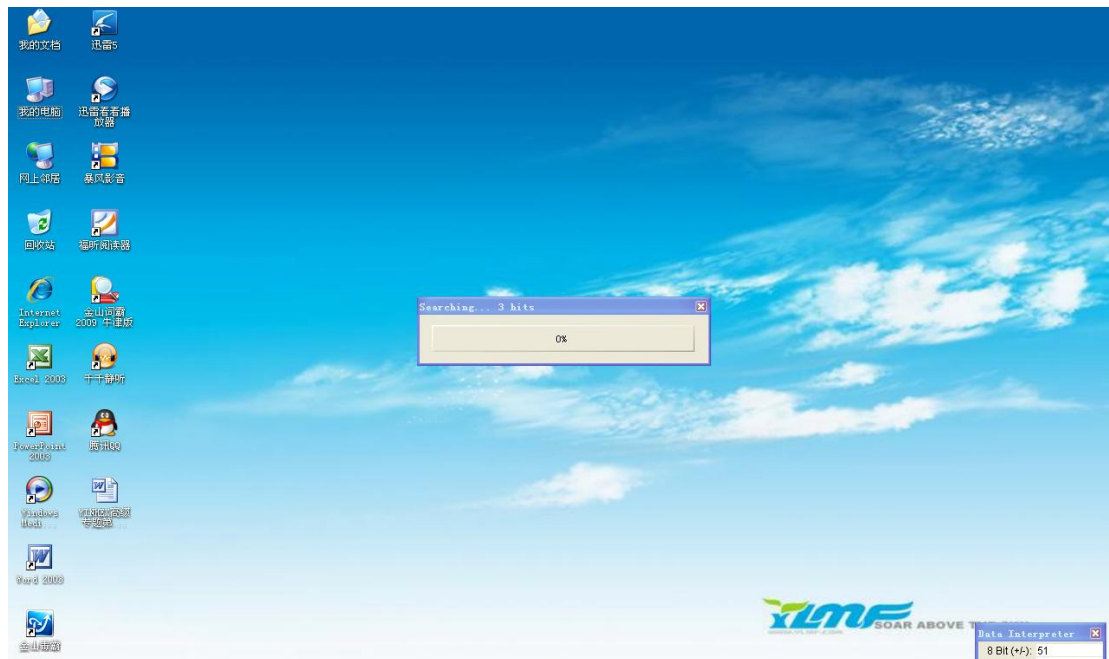




所以,运行WINHEX API前最重要的就是保证WINHEX本身处于关闭状态.如果初始化则返回1并执行下面的指令:



我们看到,下图中在看不到WINHEX主界面的情况下却出现了WINHEX搜索进度条,这是由于执行了程序中给出的搜索API函数的缘故.可见程序已经正常运行:



该部分代码展示了条件判断的全过程:

```
if (WINHEXSTARTNUM == 1)
{
    // WINHEXAPI.WHX_OpenEx("80h", 0x00000001);
    if (WINHEXAPI.WHX_Open("80h") == true)
    {
```

```
//WINHEXAPI.WHX_Create(@"E:\困惑的浪漫.txt",  
1000000);  
  
WINHEXAPI.WHX_Find("0xEB52904E","Down  
SaveAllpose");  
  
if (WINHEXAPI.WHX_WasFound() == true)  
{  
    MessageBox.Show("找到了!!!");  
}  
  
WINHEXAPI.WHX_Done();  
}  
else  
{  
    WINHEXAPI.WHX_Done();  
}
```

首先,当WINHEXSTARTNUM为1时,如果WINHEXAPI.WHX_Open("80h")函数执行成功,则WINHEXAPI.WHX_Find("0xEB52904E","Down SaveAllpose"); WHX_Open("80h")是打开设备或文件的API函数,形参80H是0号硬盘的BIOS简写. WHX_Find是搜索API函数,我们看到其语法和脚本有异曲同工之妙,第一个实参是搜索内容,第二个实参是搜索模式,这里用到了“向下搜索”和“保存搜索结果”两个模式.这两个函数我们以后都会详细演示。这里大家记住,如果你在脚本学习上下了功夫,那么你理解WINHEX API的过程就只是对编程语法的学习、熟悉而已,因为灵魂你已经基本掌握了。

相信大家对WHX_Init已经有了理性了解,说实话WHX_Init是很多高手对WINHEX API得其门而不入的重要原因,因为他们总是不能把程序成功运行起来。

WHX_Init还有不少限制,比如WINHEX权限不足会返回2错误,进入受限模式,版本号没输正确会引发-2错误.WINHEX自身故障会引起0错误.3错误在现有版本中不

常见.这些错误类型的使用非常简单,用条件判断结构即可.

以上代码中可以看到,我们频繁使用了**WINHEXAPI.WHX_Done()**;这个函数,该函数类似于C中的内存回收和C#中的垃圾回收,他能保证WINHEX API在执行过后能杀灭驻留在进程中的WINHEX程序.从而保证下一次WHX_Init能够成功.所以,每一段程序末尾,大家一定要执行这个函数.

BOOL WHX_Done();	程序末尾一定要执行.
-------------------------	------------

该函数是一个布尔函数,返回值只有TRUE或FALSE,C/C++中可以为0或1.该函数声明原型:

```
[DllImport("whxapi.dll",SetLastError=true,CharSet=CharSet.Ansi)]
```

```
public static extern bool WHX_Done();
```

WHX_Open也是WINHEX API中最重要的函数之一,如果打不开设备和文件那么剩下的操作就都没用了.

BOOL WHX_Open(LPCSTR lpResName);	C:\My File.txt <i>file</i>
	D: <i>logical drive</i>
	E: <i>logical drive</i>
	00h <i>floppy disk</i>
	80h <i>hard disk 0*</i>
	81h <i>hard disk 1*</i>
	<i>removable medium</i>
	82h <i>2*</i>
	83h <i>hard disk 3*</i>

	9Eh <i>1st optical disc</i>

	2nd optical disc
	9Fh

该API函数也是一个布尔类型, LPCSTR lpResName是设备或文件路径名称的字符串,官方文档中已经给出了字符表示方法.前面的代码中已经给出示例—和脚本几乎一样.注意, LPCSTR是一个WIN32类型,在WINDOWS API编程中很常见,代表ANSI字符串,进入托管环境后都声明为string.记住设置CharSet=CharSet.Ansi,否则传入的字符串会不完整.该函数声明原型:

```
[DllImport("whxapi.dll", SetLastError=true, CharSet=CharSet.Ansi)]
```

```
public static extern bool WHX_Open(string lpResName);
```

今天就到这里,内容会越来越难,大家一定要不断学习编程的基本知识来配合本教程的学习.如果你对编程一窍不通,我敢说你会比看天书还难受.动起来吧,难道一辈子依赖R-STUDIO混日子?呵呵,我想大家不会.

高志鹏

WINHEX 高级专题第四讲

简单结构

通常在 C 系列（C++、C#、JAVA）语言中，符号占据了语法构成的绝大部分，比如“：”在面向对象编程中既有类之间的继承含义，又有三项表达式中的遴选含义。继承可以从基类的继承，更可以是对接口的实现，对构造函数的调用，对 switch 标签的终结。分号表示语句的结束，花括号表示对执行程序的分块。圆括号包含参数列表，同时又起到强制转换、隔离代码元素、定义表达式求值顺序的更重要的作用。对上面的列举，我们可以总结为：程序是由符号的多重含义组织成的简单结构。

上面的话乍听起来很拗口、很抽象。没关系，程序就是对现实世界事务的抽象，是表达意愿的抽象，是计算机对人类思维进行探究的抽象。如果我们要学习编程，就必须适应这种抽象，必须养成一种以点代面的思维方式。有的朋友会说：“我们恰恰缺少的就是这种思维方式。”其实他们说的这句话本身就已经是对自己思维能力的抽象，“缺少”两个字着实是严重了。

我们不妨先看一段代码：

//声明一个在程序集中访问级别为公共的结构，结构名称为TESTWINHEXCODE

```
public struct TESTWINHEXCODE
```

```
{
```

//为该结构声明一个公有的构造函数，该构造函数有两个形参，

//分别是类型为单精度浮点型的var1和var2

```
public TESTWINHEXCODE(float var1,float var2)
```

```
{
```

//构造中执行一个输出语句，输出的第2实参是一个表达式，

//结果为var1和var2的和。同时，在第1实参字符格式设置参数中，

//占位符为0号，同时设置输出显示精度为两位小数

```
Console.WriteLine("{0:F}",var1+var2);
```

```
}
```

```
}
```

```

//声明一个访问权限默为公共的类类型Program
class Program
{
    //声明一个静态的，返回类型为无的应用程序入口执行函数Main，
    //同时声明一个形参args，它需要从外部获取一个字符串数组。
    static void Main(string[] args)
    {
        /*声明一个单精度浮点型数组，它有两个元素，
        * 一个是2.0，一个是3.0，我么称之为硬编码。
        * 为了使类型更为精确，我们对它们进行了强制转换
        */
        float[] vals = {(float)2.0,(float)3.0};
        //声明一个结构TESTWINHEXCODE的变量ts,
        //并t将一个TESTWINHEXCODE引用返回给ts，同时从外部执行构造
        TESTWINHEXCODE ts = new TESTWINHEXCODE(vals[0],vals[1]);
    }
}

```

上面的代码中，我详细地给出了每一句的注释，这是一段非常简单的代码，程序的第一阶段声明了一个结构，并定义了构造。程序的第二阶段是执行部分，演示了标准实例化构造过程。程序的结果不用执行也能看出是 5.00。

```

5.00
请按任意键继续. . .

```

如果要用 WINHEX 脚本来实现，便困难得多，因为 WINHEX 脚本缺乏符号的多重含义。难道，WINHEX 脚本已经不能算是严格意义上的程序了？

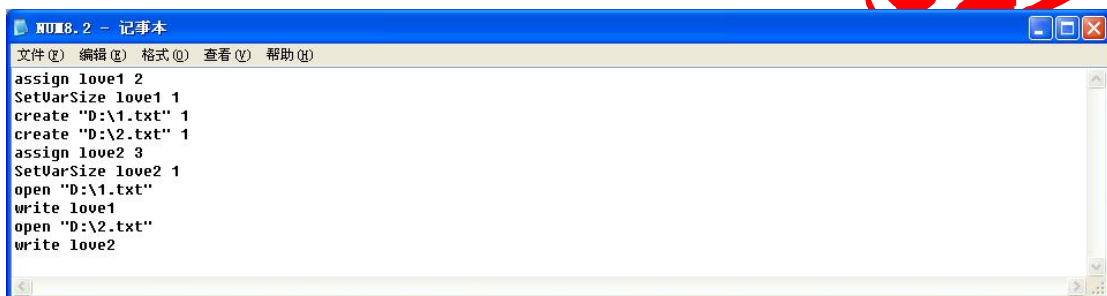
其实，WINHEX 脚本与高级程序语言的区别还体现在 WINHEX 缺乏庞大的预定义系统。在一门语言的设计初期，往往只存在少量的关键字和编译逻辑，后来，随着对语言功能要求的逐步提升，关键字经过复杂的二次组合，形成更为高级的、严密封装的上层结构，封装后的关键字在程序中一句顶替原先很多功能（执行了它封装的低级语句）。这种层次分明的封装，造就了高级程序语言的预定义系统，举几个简单的例子我们经常用到的 int、float 等简单类型，其实质是一个结构，

我们完全可以从它们身上找到结构的所有功能和限制。我们用到的+-*/*d 等数学符号，背后都隐藏着一个执行函数，只是它们没有反应到台面上而已。C#和 C++都支持强大的运算符重载，其实就是对原先执行函数的重定义。

WINHEX 的预定义系统不可能跳出原先作者设计程序时定义的功能，但是我们可以用三种方法来努力打破这种限制：

1. 利用嵌套脚本来完成。
2. 利用 WINHEXAPI 函数来完成。
3. 通过脚本调用其它语言程序来完成。

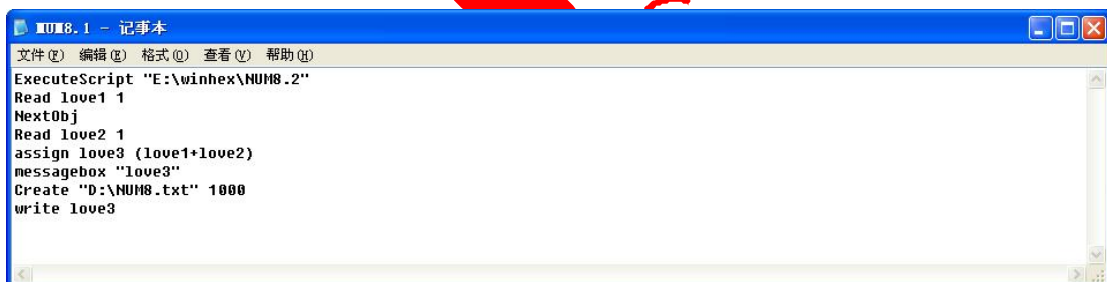
上面的程序，可以这样用脚本来实现：



```

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
assign love1 2
SetVarSize love1 1
create "D:\1.txt" 1
create "D:\2.txt" 1
assign love2 3
SetVarSize love2 1
open "D:\1.txt"
write love1
open "D:\2.txt"
write love2
  
```

脚本 NUM8.1



```

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
ExecuteScript "E:\winhex\NUM8.2"
Read love1 1
NextObj
Read love2 1
assign love3 {love1+love2}
messagebox "love3"
Create "D:\NUM8.txt" 1000
write love3
  
```

脚本 MUM8.2

assign love1 2

我们首先声明一个变量 love1 给它赋值为 2

SetVarSize love1 1

我们将变量 love1 的数据长度限制为 1 个字节

create "D:\1.txt" 1

我们创建一个文本文件，大小为 1 字节

Close

我们关闭该文件

create "D:\2.txt" 1

我们创建一个文本文件，大小为 1 字节

Close

我们关闭该文件

assign love2 3

我们声明一个变量 love2，赋值为 3

SetVarSize love2 1

我们将变量 love2 的数据长度限制为 1 个字节

open "D:\1.txt"

我们打开 D:\1.txt

write2 love1

我们写入 love1，注意这里用的是“write2”，它可以保证光标位置不发生变化。

open "D:\2.txt"

我们打开 D:\1.txt

write2 love2

我们写入 love2

.....

ExecuteScript "E:\winhex\NUM8.2"

我们启动脚本 E:\winhex\NUM8.2，注意要指定详细路径

Read love1 1

我们从当前位置读取一个字节数据到变量 love1 中

NextObj

转移到下一个工程

Read love2 1

我们从当前位置读取一个字节数据到变量 love2 中

assign love3 (love1+love2)

我们将表达式 love1+love2 装入变量 love3

messagebox "love3"

显示 love3 中的数据

Create "D:\NUM8.txt" 1000

创建一个大小为 1000 字节的文本

write love3

将 LOVE3 写入该文本

程序运行结果：



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C
D	E	F											
00000000	05	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00

上面这段脚本实现了程序的功能，但是还不值得庆贺，因为它仅仅是通过一个嵌套脚本来实现的。在这中间我们创建了许多用于缓冲数据的文本文件，注意，建立缓冲文本是 WINHEX 脚本的最绝妙的编程技巧之一。
该程序其实还可以这样实现：

```
create "D:\NUM83.txt" 348
```

创建一个 348 字节的文本

```
assign love1 "assign love7 2 "
```

声明变量 love1 为语句“assign love7 2”

```
assign love2 "SetVarSize love7 1"
```

设定变量 love2 为"SetVarSize love7 1"

```
assign love3 "assign love8 3"
```

声明变量 love3 为语句“assign love7 2”

```
assign love4 "SetVarSize love8 1"
```

```
assign love5 "assign love9 (love7+love8)"
```

声明变量 love5 为语句“assign love9 (love7+love8)”

```
assign love6 "write love9"
```

```
write love1
```

困惑的浪漫（高志鹏 6）

注意大家恐怕不能用 `ExecuteScript "111.whs"` 直接调用这个脚本，我们需要先用记事本打开 `111.whs`，重新点击保存，记事本程序才会自动消除动态写入带来的语法歧义，尽管它看起来与正常脚本没有什么不同。

上面的脚本似乎努力去实现动态语言的某些功能，但是很不完美，毕竟 WINHEX 还没有一个完美的编译器作为支撑，但是，上面脚本的思想是值得学习的。

下面的程序要复杂一些：

```
//引入 System.Collections命名空间;
using System.Collections;

//引入System;
using System;

//声明命名空间test
namespace test
{
    //声明类point
    class point
    {
        //声明一个公有的字段X
        public int X;
        //声明一个公有的字段Y
        public int Y;
        //声明一个构造函数，参数列表包含两个32位整型形参
        public point(int offsetX, int offsetY)
        {
            //在构造中为X和Y赋值
            this.X = offsetX;
            this.Y = offsetY;
        }

        //重写Object类型的ToString()方法
        public override string ToString()
```



```
{  
    //方法返回值为X和Y值的字符串相连  
    return X.ToString() + Y.ToString();  
}  
  
//声明一个公有的无返回类型的函数move，函数两个32位整型形参  
public void move(int pX,int pY)  
{  
    //给X和Y增加传入的参数值  
    X += pX;  
    Y += pY;  
}  
}  
  
//声明一个公有的类类型gzp  
class gzp  
{  
    //静态应用程序入口点  
    static void Main()  
    {  
        //实例化一个动态数组类，大小为3个元素  
        ArrayList arr= new ArrayList(3);  
        //循环将point的对象装入动态数组对象arr  
        for (int i = 0; i < 3; i++)  
        {  
            // ArrayList的实例函数Add  
            arr.Add(new point(10,10));  
        }  
        //循环调用point的实例函数move，更改对象堆中的值  
        for (int i = 0; i < 3; i++)  
        {
```

```
//Object必须强制转换为Point
((point)arr[i]).move(20,30);
}

//循环输出改变后的每一个Point对象的值
for (int i = 0; i < 3; i++)
{
    Console.WriteLine(((point)arr[0]).ToString());
}

}

}
```

以上代码验证了值类型和引用类型的内存关系，值类型在栈上分配，引用类型引用部分在栈上分配而对象数据在堆上分配。如果把Point改成结构类型，那么装箱拆箱将不会改变数据，编程有一定基础的朋友想必都很清楚，我不再多说。

上面程序运行结果：

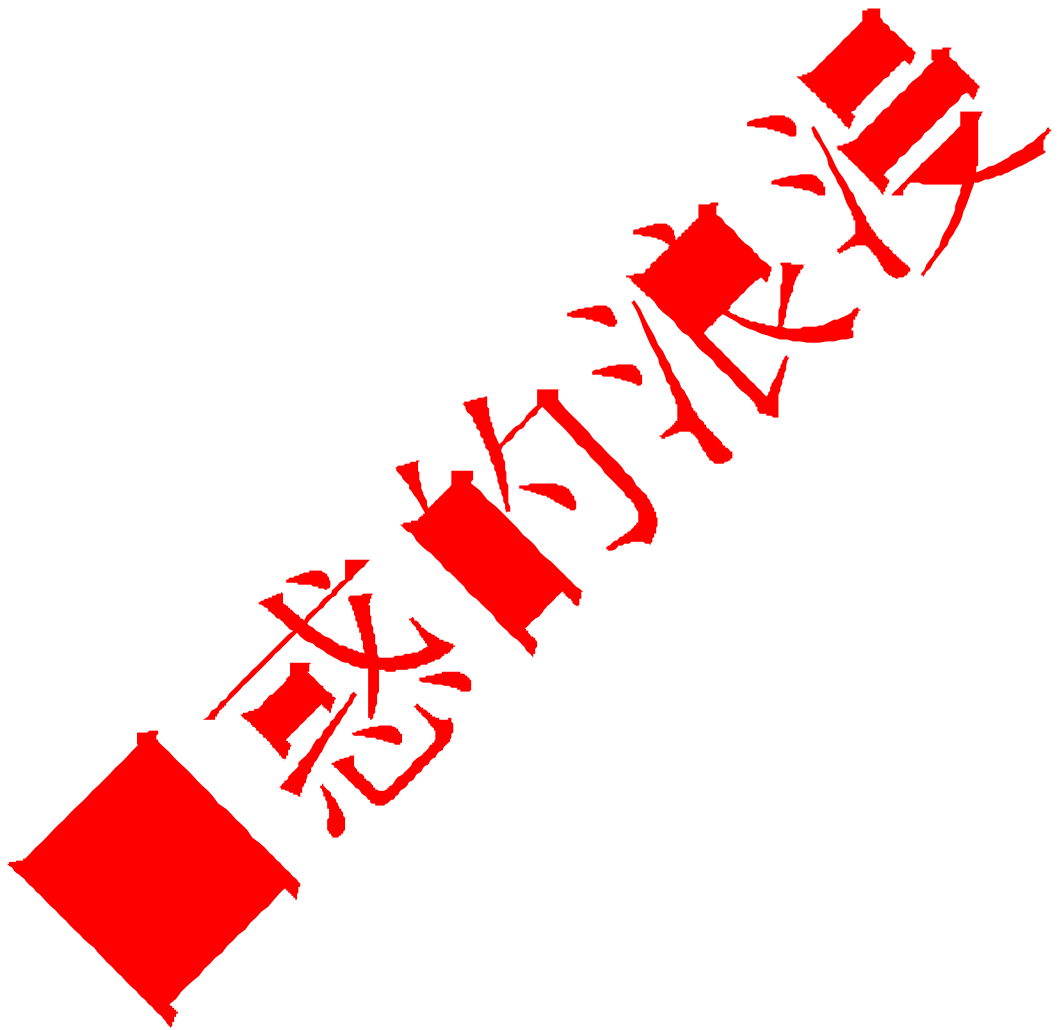


上面的程序如果用 WINHEX 脚本来实现那将比较恐怖的事情，但我把它留作思考题，下一讲但愿我和各位读者都能编出一个绝妙的脚本来共同欣赏。给各位的建议是“穷尽一切办法”。相信各位的实力又将大大提升。

提示：

1. 脚本的循环结构
2. Inc MyVariable 循环递增
3. 缓冲文本
4. 编程实例强的朋友可以考虑将缓冲写入内存，注意可别损坏操作系统内核。
5. IntToStr MyStr MyInt 的数值<>字符转化功能

最近很多朋友向我咨询那本 WINHEX 教程的购买情况。再次说明：该教程在淘宝有售，但不是本人销售，大家在淘宝中搜索 WINHEX 即可找出。对脚本基础语法还不熟悉的朋友建议购买。



WINHEX 高级专题第五讲

类型基础（抽象和继承）

注意：淘宝教程共 416 页，是 WINHEX 脚本的基础知识，和现在的高级专题不同，大家视自身学习情况购买。

前两讲，我们花费大量篇幅介绍了 WINHEX 脚本语言的一些独特技巧，使其实现了简单的数据交换和结构嵌套。今天我们要来研究现代编程语言中最重要的单元：类型。在汇编和 C 的时代，程序员几乎是依靠晦涩难懂的寄存器交换和庞大的变量函数群来堆砌代码。虽然从 C 语言开始，结构化编程思想深入人心，可是由于缺乏对现实事物的高度概括抽象能力，C 语言依然是一种不近人情的“机械语言”。那时候，人们希望利用程序在计算机中模拟生产生活是一件相当困难的事情，人们称其为面向过程时代，人们往往为了保障代码的正确性耗费大量精力，而对程序本身的功能性健壮性有所忽略。面向过程语言开发的作品，往往生产周期长、资源浪费严重、代码 BUG 过多、软件操作复杂等通病。善于创新和进取的人类当然不能满足这种苦难的创造历程，一种简单、集约、平易近人、高度概括的新开发模式呼之欲出，这就是面向对象。

面向对象大体可以概括出三个特征：

抽象：类型是对一类事物特征行为的抽象，而对象是某个类型的实际范例。人们可以先期够造出一个模板，再根据实际情况向模板赋予活力。

继承：人们可以创建一个父类型，再根据父类型的特征衍生出一个拥有同样特征的子类型，同时子类型还拥有自己独特的特性。比如“人”可以有主妇、工人、学生等称号，他们拥有人的基本特征和行为，比如他们需要吃饭睡觉。但是不仅仅是如此，他们还必须拥有区别于其他人的地方。比如主妇有主妇的饭菜需要准备，工人有工人的零件需要组装，学生有学生的作业需要完成。而无论主妇还是工人学生，又可以向下细化具体到某个人，这样一级一级传下去，不仅“人”的基本特性得到继承，又使下一级保持了自己的特征。这种理念，大大提高了开发效率，使得大量代码得以重用。

封装：面向过程时代，人们编写一段代码往往实打实反映编写过程，程序内容

可以被轻易理解和访问，这样对软件安全造成严重隐患。面向对象通过对程序数据的分类控制，隐藏了一些行为细节，大大提高了程序的逻辑性和安全性。所以，我们可以说面向对象是一种：可抽象、可继承、可封装的现代化语言。由于 C# 是面向对象语言中的佼佼者（JAVA 也不错，但微软就是微软.....），我们还是以它为参照语言，使用 WINHEX 脚本来模仿它。有面向对象编程基础的朋友可以把重点放在 WINHEX 脚本上。教程难度从本讲开始会有一个飞跃，大家不但要注意代码的编写方法，更要时时刻刻去参悟面向对象带来的各种困惑。还是那句话：世上无难事只怕有心人。有人问我为什么把网名叫做“困惑的浪漫”，这是由于我时刻都处于一种困惑的状态，而我又认为这种状态对我个人心境的影响是十分浪漫而有趣的。

言归正传：在 C# 中，所有类型都是从 `System.Object` 中派生出来的，所以我们可以用两种方法声明一个类型：

```
namespace WINHEXLAB
//创建一个命名空间
{
    public sealed class test
    //声明一个公共的密封的类
    {
        .....
    }

    public sealed class test1 : System.Object
    //显示声明一个类并继承于System.Object
    {
        .....
    }

    class Program
    //声明一个普通类
    {
        static void Main(string[] args)
```

//声明一个私有的、静态的、无返回类型并且形参列表为字符串数组变量的Main方法

```
{
    .....
}

}
```

以上代码成功通过编译，一共生成了三个类型，test是隐式的声明方法，其IL代码中可以看出其继承于System.Object的直接证据（

```
.class public auto ansi sealed beforefieldinit WINHEXLAB.test
    extends [mscorlib]System.Object
```

```
{
```

} // end of class WINHEXLAB.test)。Test1用了显示声明其继承关系的方法其IL码和test完全一致。而Program是为了存放程序运行入口而存在的基本类型。如果我们再声明一个类型gzp并使其继承于test1

```
public sealed class test {} //声明一个公共的密封的类
```

```
public class test1 : System.Object {} //显示声明一个类并继承于
System.Object
```

```
public class gzp : test1 {} //声明一个类并继承于test1
```

其 IL 反汇编代码如下：

```
.class public auto ansi beforefieldinit WINHEXLAB.gzp
    extends WINHEXLAB.test1
```

```
{
```

```
} // end of class WINHEXLAB.gzp
```

看到奥秘所在了吧

```
extends WINHEXLAB.test1
```

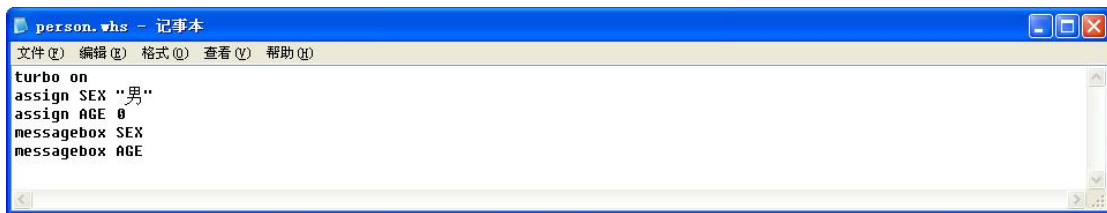
一句的

含义就是扩展自

WINHEXLAB.test1 的意思。

现在我们已经掌握了类型声明的方法和原理了，那么怎样利用 WINHEX 脚本声明一个类似的“类型”出来呢？我只能说，从原理上讲，WINHEX 脚本根本不具备这样的能力，因为底层运行机制的差异太大。但是我们还是可以用照猫画

虎的思想去完成一个“WINHEX 类”，这种手段我们在前面也展示的很多了。首先我们可以定义一个“基类”Person：



我们先声明一个变量 SEX 赋予初始值为“男”，紧接着我们就声明了一个变量 AGE，赋予初始值为 0。这就意味着，我们这个类拥有两个基本特征：性别和年龄。这个类基本抽象了“Person”的一些特征如必须有年龄和性别。



仅仅有抽象是不够的，我们必须使其实现继承的能力，比如高志鹏类继承于 Person 类就是一个不错的选择。但是 WINHEX 脚本本身是一个文本文件，怎么能让它顺利地衍生出另一个文本文件来，又同时包含原有文件的内容呢？这里 COPYFILE 等平时不常用的关键字就可以用上了。



执行脚本后，我们在 E:\winhex\winhex\下发现了一个新文件高志鹏.Whs：



其内容与 person 完全一致：



很简单吧，大家一定要掌握 **COPYFILE** 的用法，会带来很多便利。但是，要完成继承，就必须在新生成脚本中加入新的特性：比如职业、专业、网名、政治面貌等，这些特性是专属于高志鹏个人的。听起来似乎有一定的难度，我们不妨先用 **WINHEX** 编辑一下高志鹏.Whs 看看这个文件是怎样的：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	74	75	72	62	6F	20	6F	6E	0D	0A	61	73	73	69	67	6E	turbo on..assign
00000010	20	53	45	58	20	22	C4	D0	22	0D	0A	61	73	73	69	67	SEX "男"..assign
00000020	6E	20	41	47	45	20	30	0D	0A	6D	65	73	73	61	67	65	n AGE 0..message
00000030	62	6F	78	20	53	45	58	0D	0A	6D	65	73	73	61	67	65	box SEX..message
00000040	62	6F	78	20	41	47	45	0D	0A	63	6F	70	79	66	69	6C	box AGE..copyfil
00000050	65	20	45	3A	5C	77	69	6E	68	65	78	5C	77	69	6E	68	e E:\winhex\winh
00000060	65	78	5C	70	65	72	73	6F	6E	2E	77	68	73	20	45	3A	ex\person.whs E:
00000070	5C	77	69	6E	68	65	78	5C	77	69	6E	68	65	78	5C	B8	\winhex\winhex)?
00000080	DF	D6	BE	C5	F4	2E	77	68	73	20							咧九?whs

可以看出，在脚本文件中，代码内容完全是按照 ASCII 码的，脚本内容一览无余。我们不但要保证特性的继承，还要阻止重复的继承，因为高志鹏.WHS 中依然有 `copyfile E:\winhex\winhex\person.whs E:\winhex\winhex\高志鹏.whs` 这条语句，它会继续复制文件覆盖当前的脚本，这是很危险的。

所以，我们不妨利用 `remove` 将这句从脚本文件中删除。



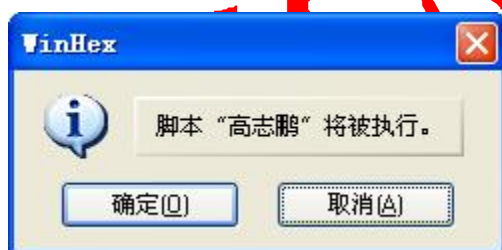
上面的脚本我们催动调用语句 `ExecuteScript` 去引导 `person2`，我们编辑高志鹏.whs 并对其 `copyfile E:\winhex\winhex\person.whs E:\winhex\winhex\高志`

鹏.whs 部分进行选块然后移除。

似乎大功告成了，可是先看看执行效果过吧：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	74	75	72	62	6F	20	6F	6E	0D	0A	61	73	73	69	67	6E	turbo on..assign
00000010	20	53	45	58	20	22	C4	D0	22	0D	0A	61	73	73	69	67	SEX "男"..assign
00000020	6E	20	41	47	45	20	30	0D	0A	6D	65	73	73	61	67	65	n AGE 0..message
00000030	62	6F	78	20	53	45	58	0D	0A	6D	65	73	73	61	67	65	box SEX..message
00000040	62	6F	78	20	41	47	45	0D	0A	45	78	65	63	75	74	65	box AGE..Execute
00000050	53	63	72	69	70	74	20	22	70	65	72	73	6F	6E	32	22	Script "person2"
00000060	0D	0A	0D														
0A																	...

copyfile E:\winhex\winhex\person.whs E:\winhex\winhex\高志鹏.whs 固然被移除了，可多出了 ExecuteScript "person2"让人很不舒服。但是这个新生成的脚本确是可以运行的：



还有一种方法，我们不必非得移除 copyfile E:\winhex\winhex\person.whs E:\winhex\winhex\高志鹏.whs，我们可以保留它只是将我们需要的部分拷贝到一个新脚本文件中。改动 person2 如下：

create E:\winhex\winhex\高志鹏.whs 1000

open E:\winhex\winhex\person.whs

block 0 72

copy

open E:\winhex\winhex\高志鹏.whs

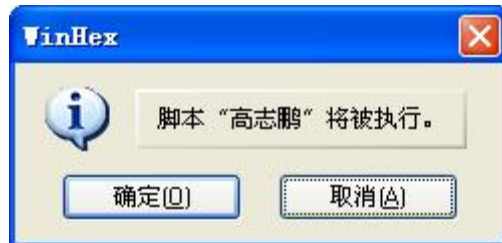
Paste

Save

效果已经出来了，我们不仅成功地实现了预期目标（ExecuteScript "person2"也没有了），而且新生成的脚本也是可以运行的：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
--------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

00000000	74 75 72 62 6F 20 6F 6E	0D 0A 61 73 73 69 67 6E	turbo on..assign
00000010	20 53 45 58 20 22 C4 D0	22 0D 0A 61 73 73 69 67	SEX "男"..assign
00000020	6E 20 41 47 45 20 30 0D	0A 6D 65 73 73 61 67 65	n AGE 0..message
00000030	62 6F 78 20 53 45 58 0D	0A 6D 65 73 73 61 67 65	box SEX..message
00000040	62 6F 78 20 41 47 45 0D	0A 00 00 00 00 00 00 00	box AGE.....



这的确是令人兴奋的结果，我们的目的绝不仅仅是保留父类型的特性而已，我们还要为子类型增加新的特性：

`namespace WINHEXLAB` //创建一个命名空间

```
{
    public sealed class test //声明一个公共的密封的类
    {
        //public int a = 10;
    }

    public class test1: System.Object //显示声明一个类并继承于
System.Object
    {
        public int b = 100;
    }

    public class gzp:test1//声明一个类并继承于test1
    {
        public int c = 1000;
    }

    class Program//声明一个普通类
    {
        static void Main(string[] args)
```

//声明一个私有的、静态的、无返回类型并且形参列表为字符串数组变量的Main方法

```
{
    gzp g = new gzp();
    // 实例化类型gzp
    Console.WriteLine(g.b);
    Console.WriteLine(g.c);
    //输出类型字段的值
    test1 t1 = g;
    //引用类型的隐式转换
    g.b = 4555;
    //更改对象上的值
    Console.WriteLine(t1.b);
}
}
```

上面的程序大家都能看明白，gzp 不仅集成了 test1 里的成员变量 b,更拥有自己的变量 c，子类总是可以隐式转换为父类，而且它们访问的都是同一个内存区域。该程序显而易见地揭示了继承的行为过程，运行结果如下：

```
C:\WINDOWS\system32\cmd.exe
100
1000
4555
请按任意键继续...
```

有了上面的经验，我们就可以轻易给高志鹏.WHS 加上几个特性了：

create E:\winhex\winhex\高志鹏.whs 1000

open E:\winhex\winhex\person.whs

block 0 72

copy

open E:\winhex\winhex\高志鹏.whs

Paste

goto 73

write 0x61737369676E204A4F422022B3CCD0F2D1D0BEBFD4B1220D0A

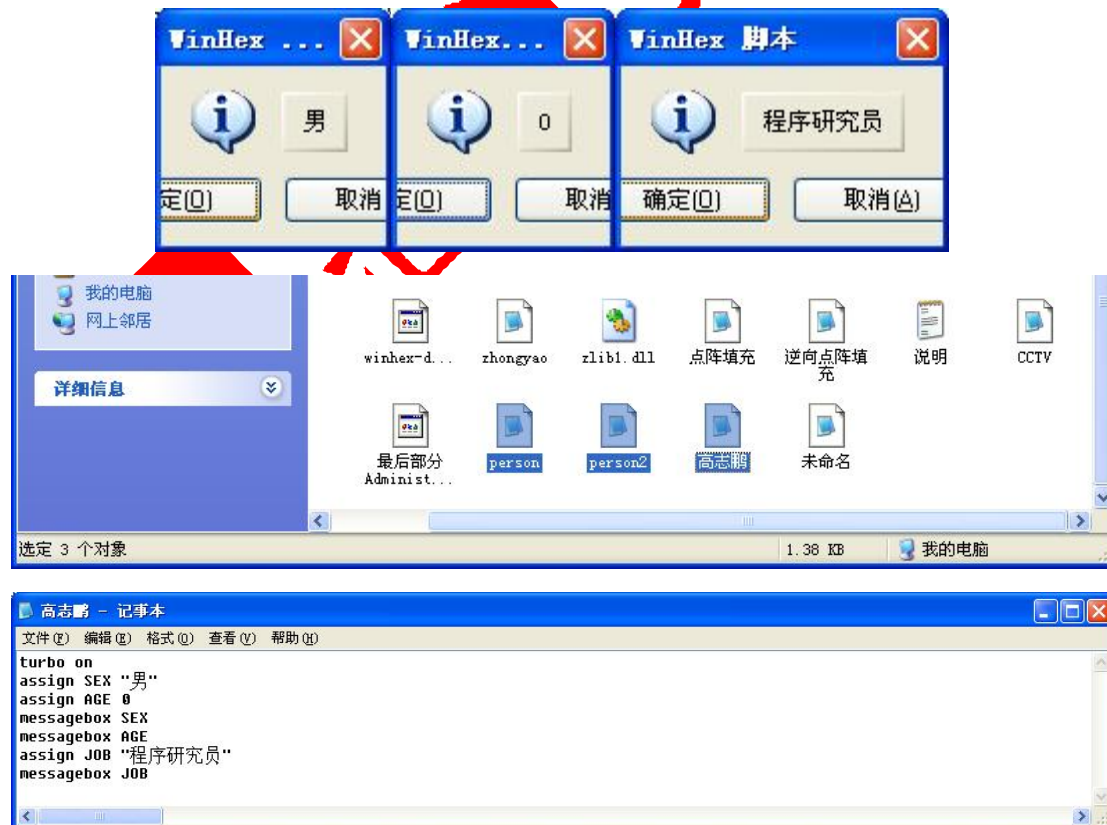
write 0x6D657373616765626F78204A4F420D0A

save

上面的两个 write 就是新写入的语句，这一次的更改是相当成功的：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	74	75	72	62	6F	20	6F	6E	0D	0A	61	73	73	69	67	6E	turbo on..assign
00000010	20	53	45	58	20	22	C4	D0	22	0D	0A	61	73	73	69	67	SEX "男"..assign
00000020	6E	20	41	47	45	20	30	0D	0A	6D	65	73	73	61	67	65	n AGE 0..message
00000030	62	6F	78	20	53	45	58	0D	0A	6D	65	73	73	61	67	65	box SEX..message
00000040	62	6F	78	20	41	47	45	0D	0A	61	73	73	69	67	6E	20	box AGE..assign
00000050	4A	4F	42	20	22	B3	CC	D0	F2	D1	D0	BE	BF	D4	B1	22	JOB "程序研究员"
00000060	0D	0A	6D	65	73	73	61	67	65	62	6F	78	20	4A	4F	42	..messagebox JOB
00000070	0D	0A	00	00	00	00	00	00	00	00	00	00	00	00	00	00

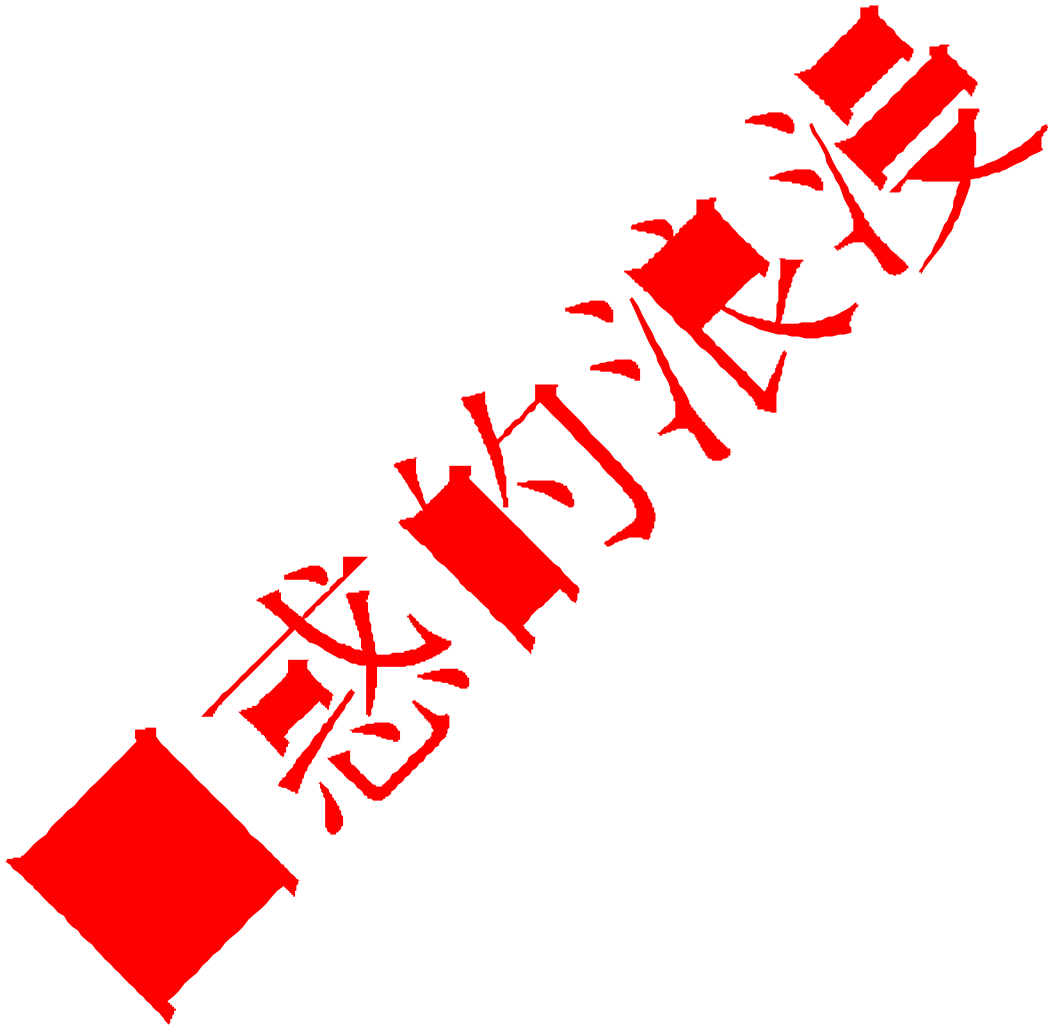
现在我们已经迫不及待地想要验证一下程序执行结果了：



大家可以看到，新生成的脚本高志鹏.whs 中成功增加了 assign JOB "程序研究

员”，这个特性，并在脚本执行中显示了出来。

想必大家都已经领悟了一些玄妙。很多人可能看得云里雾里，但是大家绝对不用担心自己的学习实力。所谓实力，其实就是毅力的坚韧度，有毅力就有实力。本讲解释了面向对象的一些知识，并成功地用 `WINHEX` 脚本模拟了抽象和继承两大特点，下一讲我们会接触封装的相关技巧。



WINHEX 高级专题六

走进本质

前些日子我成功地为 WINHEX 脚本系统增加了一条指令,想到以后 WINHEX 脚本语言将在各位高人手中扩展数百个更强大的功能,就巴不得马上将这些技术传授大家. 但是,出于循序渐进的原则,大家还是得一点一点接受,我也得一篇一篇把它们作为教学资料写出来.从本讲起我们会进入一个新纪元,完成由现象到本质的过渡.如果你的基础仍不牢靠,甚至仍不知脚本为何物,请关注淘宝销售教程,我会在文档末尾附上该教程目录.

让我们先来看一段神奇的汇编语言代码:

```
.386
.model flat,stdcall
option casemap:none
.....
.data
;tbb1 TBBUTTON=0,0,TBSTATE_ENABLED,TBSTYLE_SEP,0,0,0>
disklable Word "D:"
winhex byte "winhex.exe",0h
classname byte "WinXMDI0",0h
.start
words 0 word 38 dup(0)
.code
start:

invoke WinExec,offset winhex,1

invoke FindWindowA,offset classname,0

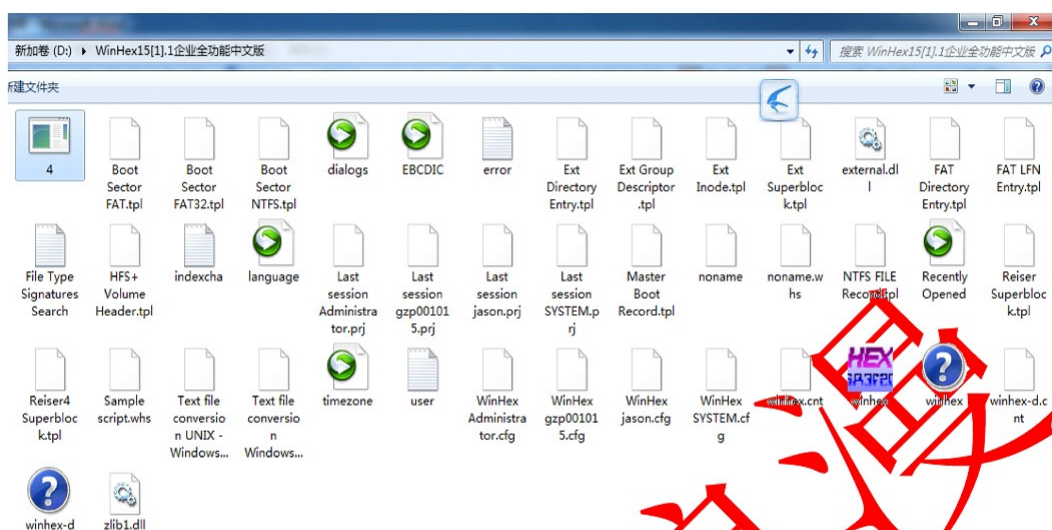
invoke SendMessage,eax,0ch,0,offset disklable

;invoke SendMessage,0,492h,0,0

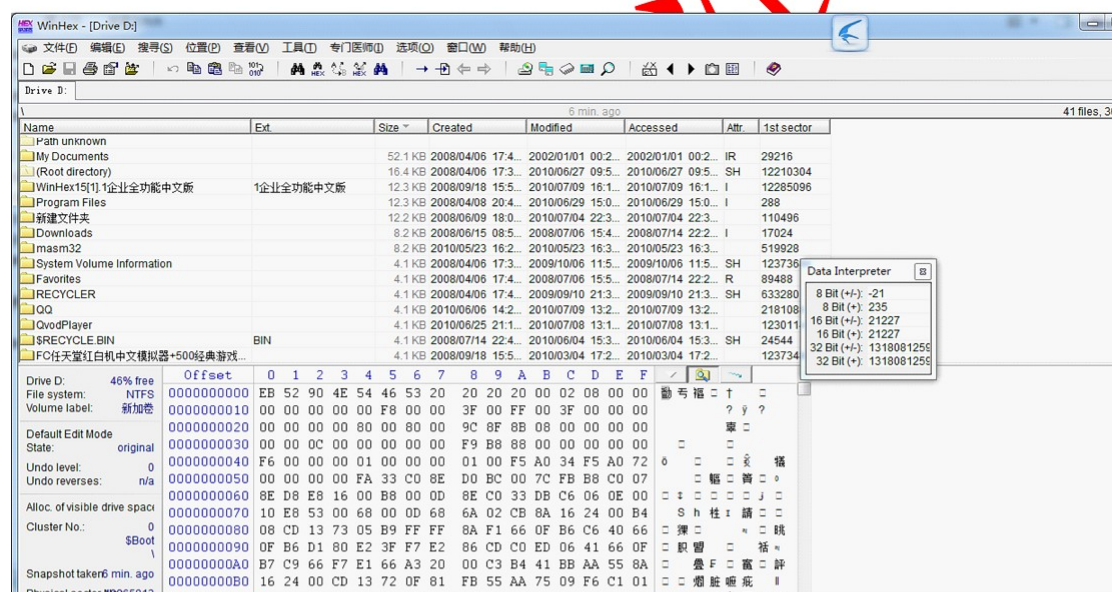
invoke ExitProcess,0

end start
```

大家先不用管这段代码的含义,我直接拿它生成了一个.EXE 的可执行文件,现在我们把生成的 4.exe 复制到 WINHEX 目录下:



我们执行 4.exe,winhex 被自动启动并打开了 D 盘.



不必猜测,能达到这种效果是因为我成功模拟了 WINHEX API 函数库里面的指令,通过反汇编 whxapi.dll 文件,我找到了遍历分区所需的必要参数和过程,成功促使 WINHEX 打开了 D 盘.因为这个实验的成功,我们改进了 WINHEX 以下缺陷:

1. 从今以后 WINHEX 二次开发不再需要 WINHEX 官方提供的 whxapi.dll.我们不再需要 WINHEX API 函数.执行指令前我们也不必先执行 WHX_INIT 指令(参考高级专题三).
2. 可以为 WINHEX 编写外挂,从另一个程序调用 WINHEX 的功能(也可以制

1. 造借 WINHEX 之手销毁数据的病毒).
2. 由于改变了 WINHEX API 中原先的那种初始化方式,WINHEX 在被外界驱动运行时不再隐藏主窗口,执行过程可见.

.....

这样的好处不胜枚举,各位是否蠢蠢欲动了呢?

前面,我们通过一部初级教程和 5 讲高级专题,使大家初步了解了 WINHEX 的基本使用方法。我们的成果是丰硕而令人鼓舞的。我们不仅一步一步摸清了 WINHEX 每个菜单、每个按钮、每个列表、每个复选框的操作步骤和操作结果。更重要的,我们掌握了脚本语言,它是伟大的发明,它让每个基础薄弱的数据恢复梦想家享受了磁盘编程的快乐,它带来的工作上的效率是以往手工恢复所不能比拟的,它解决困难的能力也是不言自明的。我们通过各种途径、各种实例,一步步掌握着脚本编程的各种技巧,我们在向高手迈进。

今天的你是否已经满足了?从 08 年到现在,已经涌现了不少脚本高手,他们的技能全部来自于自学,他们编写出了令人惊诧的代码,有人用区区几行代码就解决了某些大公司鼓吹的“非标”硬盘问题,有人设计的精妙算法甚至解决了 PC3000 的某些缺陷,有人则走上了脚本杀毒的道路,成为令人尊崇的侠士。以前,我为了鼓励大家,反复强调了脚本的优越性,我甚至希望用脚本实现部分高级语言才有的功能,我做到了,相信大家已经从中汲取了养分,但是今天,我们必须来讨论脚本的劣势,我希望今天的讨论激起大家更强烈的钻研兴趣。

脚本的劣势有两点:

第一、功能瓶颈。我们所希望的 WINHEX 脚本,可以实现 WINHEX 的所有功能,可惜的是由于开发成本,WINHEX 作者似乎并没有为每一个 WINHEX 功能都争取一个脚本实现,而且,WINHEX 脚本的功能只有 WINHEX 全部功能的 10% 不到。当然,我们可以利用 WINHEX API 函数,其它的功能让 C++、C# 去实现。可是,如果这样,我们干脆自己开发 WINHEX 好了,还千方百计利用 WINHEX 的开发包何用呢?我们也可以利用大量的嵌套脚本勉强实现一些功能,但是某些功能比如我们想利用脚本创建一个 WINDOWS 窗体,从而实现自己的 WINHEX 插件,还有,我们希望利用脚本操作注册表的键值,从而完

第一、 善杀毒功能。更大胆的，我们希望利用脚本控制驱动程序，从而为 WINHEX 戴上真正的“底层”王冠。很遗憾，对现有的 WINHEX 来说，要实现上述功能无异于痴人说梦。

第二、 速度瓶颈。我曾经对 WINHEX 进行动态调试，并发现一个惊人的现象，**OPEN D:** 这个脚本语句在 CPU 中是一个近千条指令的集合，实现过程更是百转千回。也就是说，达到同样的目的，脚本比本地代码慢数百倍之多。想必有人用脚本组过磁盘阵列，那真是一种煎熬，花费数十天时间且不说，突如其来的内存错误也会令你的艰苦等待付诸东流。

从正常角度讲，这两个缺陷属于“先天不足”，无法用“药物”医治。可是，如果我告诉大家，它们都可以用“手术”的方式来改善，尤其是功能缺陷，我可以利用特殊的逆向技术，为 WINHEX 添加近乎无限多功能的脚本指令时，大家将作何感想。

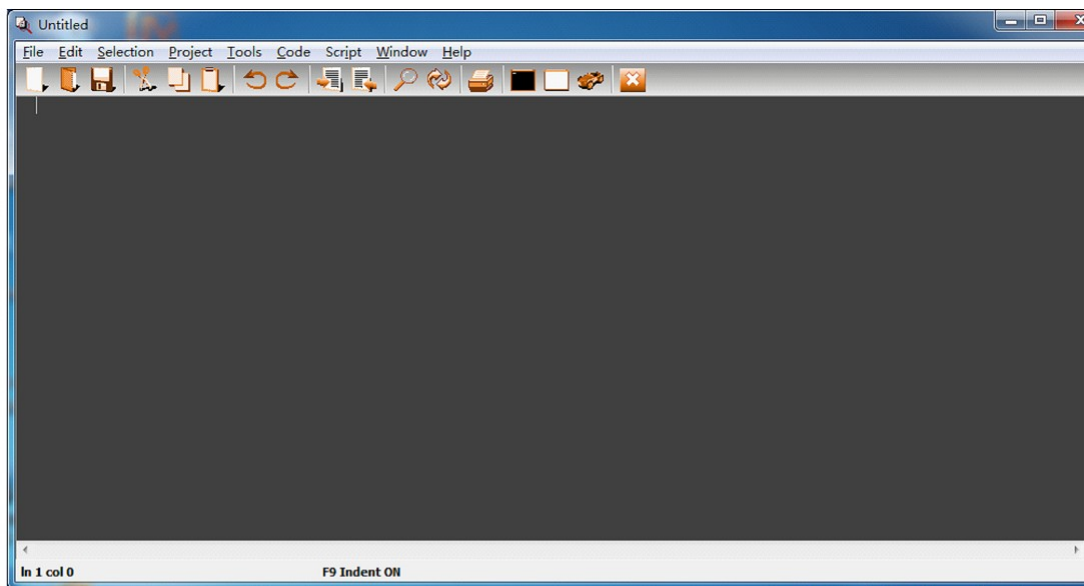
话不多说，开始我们的学习之路吧，记住，我的目的还是让大家掌握编程，WINHEX 只是我的教学工具而已。我要指出，数据恢复不是一个靠混就可以立足的行业，未来市场整合时，数据恢复势必要官方化、规范化，并成为取证行业的重要分支。目前取证和司法鉴定已经开始纳入国家考试，中国大陆甚至引进了 ENCASE 认证体系。如果不会 C++，恐怕用 ENCASE 搜个 55AA 都困难。甘愿沉沦吗？肯定不是，努力！

要改造 WINHEX，首先要了解 WINHEX 各个功能的设计思路。最直接的手段是阅读 WINHEX 的源码，当然，这是不可能的，WINHEX 是 delphi 语言的杰作(有个很大的好处是 delphi 几乎和 WINDOWS API 函数密不可分，这很利于我们后来的分析)，所以我们只有逆向这一条路。逆向工具有很多，WINHEX 本身就是一个静态逆向工具，网上就有教程利用 WINHEX 破解注册码，修改游戏金钱，但是 WINHEX 并不适合所有的逆向工作，WINHEX 的逆向只针对数据，很难针对代码。用 WINHEX 修改某处的数值是可能的，但是要用 WINHEX 分析改变一个程序的流程是相当困难的，这需要使用者熟练掌握二进制编程，也就是直接用机器码编程。虽然二进制编程还不能算是什么不可企及的境界，但是效率毕竟太慢。我曾经用机器码写过一个 C 语言两行就可完成的程序，输入

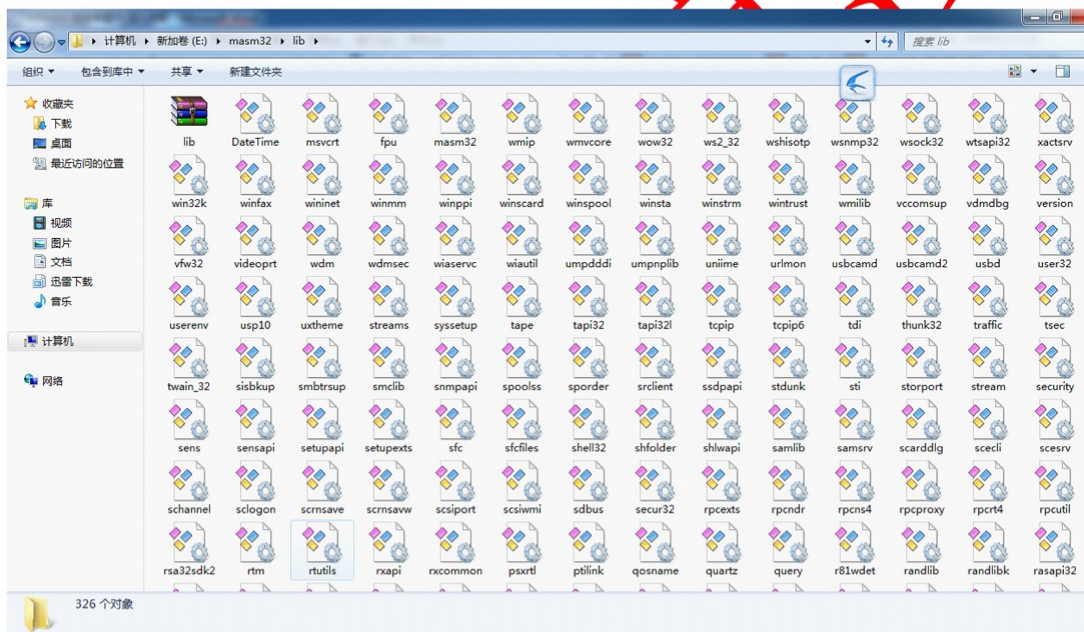
了数百个字符，错误百出，调试一下午才宣告完成，这肯定是不可取的。我们的逆向工具起码是要有反汇编能力的，也就是能显示出汇编语言的。

说起汇编语言，无数程序员的第一反应就是头大，晦涩难懂的助记符，不计其数的伪指令和寻址方式，都足以让人望而生畏。但是，我要指出，在计算机学科中，要看到技术的本质，汇编语言是必备技能。如果你轻视汇编语言，对不起，你离真正高手还有很长的路，甚至没有路。汇编语言和机器码一一对应，也就是说，汇编语言是机器码的一种较易懂的表现形式。正因为如此，汇编语言可以反映出软件执行的最底层现象：直接控制硬件。它是唯一没有包装没有掩饰的编程语言。汇编语言有很多种，可以说有多少处理器就有多少种汇编语言，但是至今在 PC 机上，8086 汇编和 80386 汇编还是最流行的。因为最主要的两大处理器生产商 INTEL 和 AMD 几乎完全统一了它们的指令集，并向下兼容。中途曾经杀出过 64 位安腾处理器，但是现在微软已经放弃了对它的支持，微软认为，当前架构仍然是短时间内无法改变的最成熟的处理模式。针对安腾，AMD 率先推出了兼容 IA-32 的 64 位处理器，INTEL 也不甘落后，推出了高度一致的 64 位处理器。32 位和 64 位汇编我们都必须掌握，国内至今还没有多少公开用 64 位技术开发的高手，但是我相信，64 位的普及应该是 10 年内的事情。64 位开发并没有多大难度，相反 64 位增加了不少寄存器和位宽，可以显著提高程序速度，增加编程乐趣。

MASM 是目前最流行的汇编编译器，因为它是微软出品，所以开发包最全，民间高手关注度也最高。MASM 虽然被很少人诟病，可是我经过长时间的比对，仍然认为它的设计最为合理。我们就用作为我们的教学蓝本。MASM 有个 IDE 环境 MASM32，使用方便简单，操作步骤我们在编程中会体现出来。



MASM32IDE 开发环境



MASM32 开发包

从图中可以看出，MASM32 有个 WINDOWS 下的窗体界面，有近乎所有的开发函数包。可以说。只要你愿意，可以用它开发出任何程序。一开始的学习并不急着去用汇编语言驱动硬件，那对初学者还有很大困难。建议大家先去读一些 DOS 和 WIN32 汇编教材，这样也可与我们的 WINHEX 改造工作相得益彰。我肯定不会一条指令一条指令讲解，和 C#一样，基础语法是靠大家私下努力的。

我们先来看一段代码：

stack segment ;设置栈段

db 1000 dup (?);设置一个 1000 字节大小的未初始化内存

stack ends;栈段结束

data segment;设置数据段

gzp db "你好高志鹏!",0h;设置一个变量 gzp,代表“你好高志鹏”的首地址

data ends;数据段结束

assume cs:code,ds:data,ss:stack;段寄存器关联,代码关联 CS,数据 DS,栈关联

ss

code segment;设置代码段

start: 程序起始

mov ax, data; 将 data 地址传给 AX 寄存器

mov ds,ax;用 AX 初始化 ds 寄存器

mov ah,9h ;功能号 9,显示字符串

mov dx,offset gzp DS:DX 存入 gzp 的首地址

int 21h;调用 21h 中断

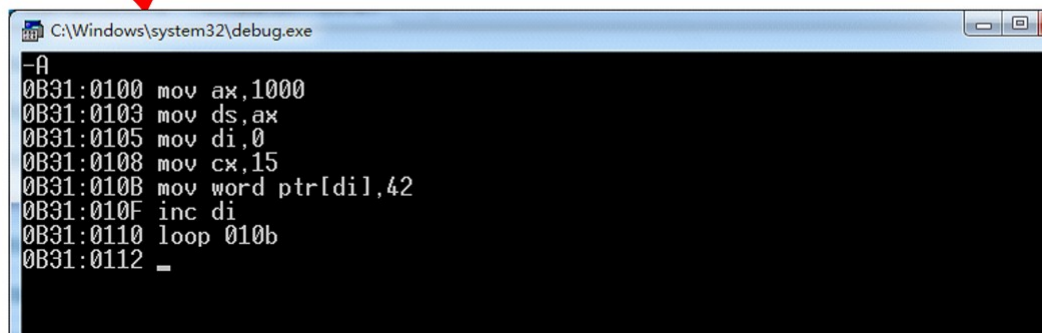
mov ah,4ch 功能号 4c,终止线程

int 21h;调用 21h 中断

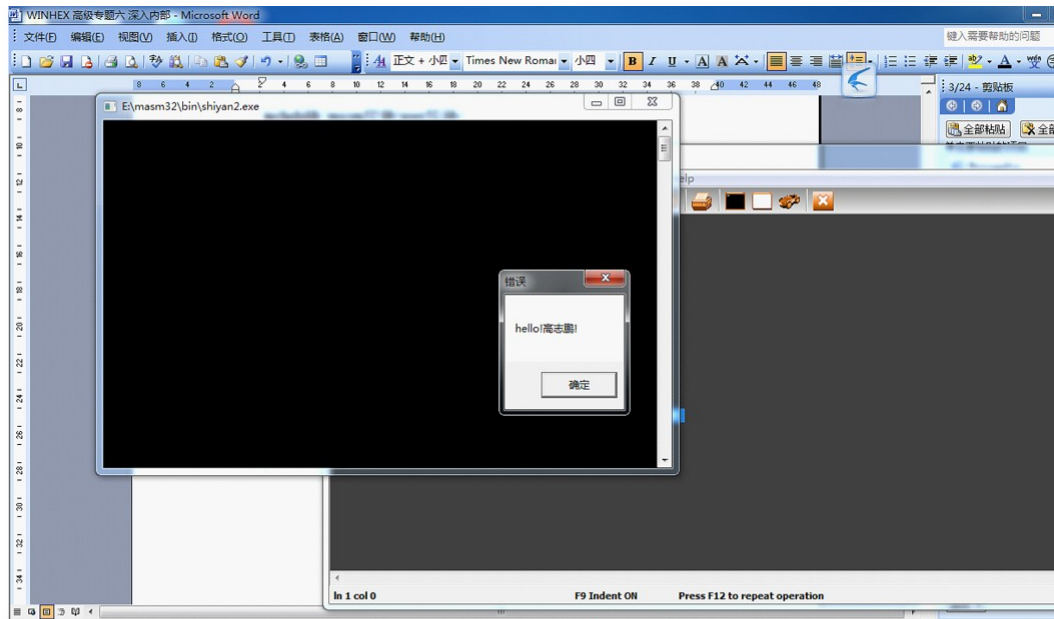
code ends;代码段结束

end start;程序入口为 Start

这是一个典型的调用 MS-DOS INT21 中断的程序,注释我也已经给出.我们可以看到,MASM 的语法非常严谨,并不是想象的那样汇编语言就是一盘散沙.我们再来编一段 debug 码:



```
C:\Windows\system32\debug.exe
-A
0B31:0100 mov ax,1000
0B31:0103 mov ds,ax
0B31:0105 mov di,0
0B31:0108 mov cx,15
0B31:010B mov word ptr[di],42
0B31:010F inc di
0B31:0110 loop 010b
0B31:0112 _
```

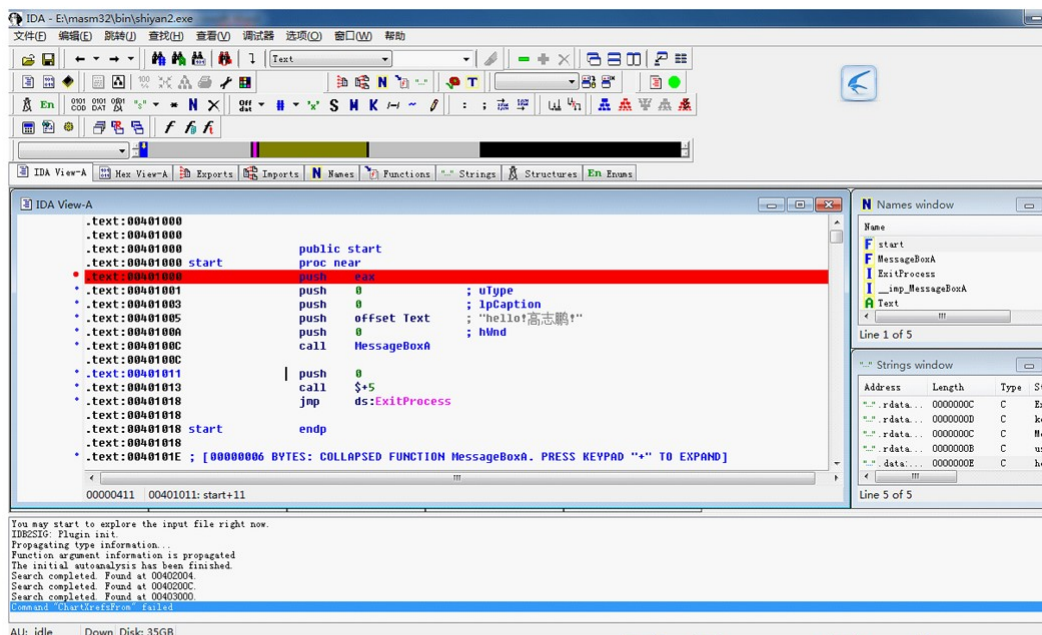



我们可以对比一下这次三种方式有何不同,哪种更简单?答案当然是后者,因为后者利用了 **WINDOWS** 的 **API** 函数,大大缩短了程序开发难度,程序风格也更接近高级语言。

这里有几点必须注意:第一个中断程序只能用于实模式下,也就是 **DOS** 下.第二个 **DEBUG** 是应用于虚拟 **8086** 模式下的,而第三个程序只能在 **32 位 windows** 操作系统下运行,程序开头我们用了伪指令 **.386**,就是说我们要启用 **386** 中的指令和 **32 位** 寄存器.**model flat,stdcall** 指明了内存初始化方式为保护模式,过程调用方式为 **stdcall**,这些都是针对 **WINDOWS** 操作系统而言的,在 **DOS** 下运行势必会出错。

相信大家此刻已经领略了汇编语言的一些特色,在汇编语言下一切似乎都是透明的,什么指针,数组,字串如此直观.但是离我们的逆向工程所要展示的内容还相差甚远,逆向工程最重要的是反汇编,是将机器码还原成汇编的过程.当然,任何工具都无法一丝不差地按照最优方式还原出作者本身的代码,但是逆向工具的智能度越高,我们后期处理话费的精力就越小,所以我们在兵器的选择上一定要慎重.这里有一个工具不得不说,它让所有的静态反汇编工具都相形见绌,那就是 **IDA PRO**。

IDA PRO 相信大家都不会完全没有印象,病毒分析就常常利用它来进行.熟练掌握它意味着你已经迈入黑客的行列。



IDA PRO

通过 IDA PRO 的分析,程序的执行过程已经显而易见:

```
.text:00401000 ; ***** SUBROUTINE *****
.text:00401000
.text:00401000
.text:00401000 public start
.text:00401000 start proc near
.text:00401000 push     eax                ; uType
.text:00401001 push     0                  ; lpCaption
.text:00401003 push     0                  ; "hello!高志鹏!"
.text:00401005 push     offset Text        ; hWnd
.text:0040100A push     0
.text:0040100C call     MessageBoxA
.text:0040100C
.text:0040100C push     0
.text:00401011 call     $+5
.text:00401013 jmp     ds:ExitProcess
.text:00401018
.text:00401018 start      endp
.text:00401018
```

主要代码

上图代码中没有用 INVOKE 伪指令,而是执行了 CALL 指令(两者后续教程会说明白),先用 push 指令将过程参数从右至左压栈(stdcall),然后调用 CALL 将当前 EIP 寄存器所保存的地址压栈并跳转至 MessageBox 函数的地址执行.由于 MessageBox 是 STDCALL 的 WINDOWS API 函数,所以堆栈释放由被调用方执行我们不用考虑.下面三条指令比较奇怪,它先压栈了地址 00401018(\$表示当前地址,\$+5 其实就是指令 jmp ds:ExitProcess 的地址)然后跳转至 \$+5 继续执行,着看起来似乎是绕了个大圈子,还不如直接 call ExitProcess 来得爽快.

.text:00401011

push 0

.text:00401013

call \$+5

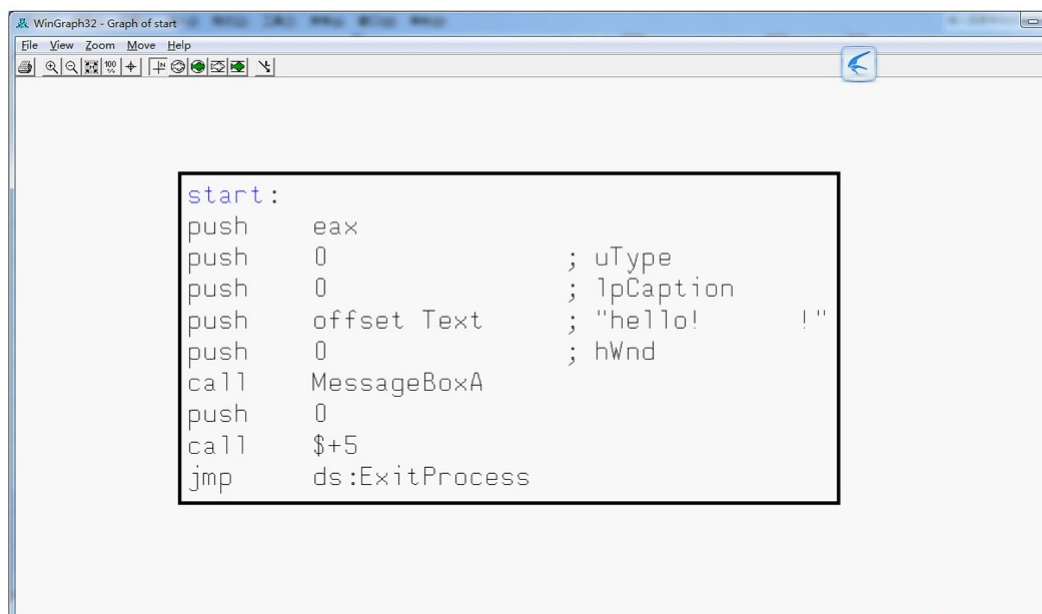
ds:ExitProcess

API 函数说明

```
.data:00403000 ; Section size in file      ; 00000200 (    512.)  
.data:00403000 ; Offset to raw data for section: 00000800  
.data:00403000 ; Flags C0000040: Data Readable Writable  
.data:00403000 ; Alignment       : default  
.data:00403000 ; ccccccccccccccccccccccccccccccccccccccccccccc  
.data:00403000 ; Segment type: Pure data  
.data:00403000 ; Segment permissions: Read/Write  
.data:00403000 _data                segment para public 'DATA' use32  
                                assume cs:_data  
.data:00403000                               org 403000h  
.data:00403000 ; char Text[]  
.data:00403000 Text                 db 'hello!高志鵬!',0          ; DATA XREF: start+5f0  
.data:0040300E align 200h  
.data:0040300E _data                ends  
.data:0040300E  
.data:0040300E  
.data:0040300E end start
```

全局变量声明

核心代码无非是下图所示,这是利用了 **IDA PRO** 的流程图绘制技术,我们的代码只有一个过程,所以图很简单:



```
start:
push    eax
push    0           ; uType
push    0           ; lpCaption
push    offset Text ; "hello!  !"
push    0           ; hWnd
call    MessageBoxA
push    0
call    $+5
jmp     ds:ExitProcess
```

上述反汇编结果与 MASM 所编代码究竟有何不同呢?答案是没有不同,反汇编出的结果与 MASM 中写下的代码所生成的机器码是一致的,只是表现形式有所差异,最重要的是 INVOKE 和 CALL,它们究竟有何不同,我请大家先查查资料,CALL 可是逆向工作中接触到的最重要的指令之一,务必牢牢掌握.今天的课程可以说是一个总论,细节问题会在今后的课程中依次展开.

淘宝教程名称:

WINHEX 教程 困惑的浪漫 完全版

教程共 420 页

目录:

第一章认识 3

1.1 启动中心 3

1.2 主编辑窗口 5

1.3 分区快捷入口 6

1.4 快捷恢复 8

1.5 文件主菜单 9

1.6 新建项目 9

1.7 打开项目 11

1.8 保存项目 13

1.9 创建磁盘镜像和 WINHEX 备份文件 14

1.10 备份还原 16

1.11 文件基础属性 17

1.12 打开文件夹 18

1.13 批量保存 21

第二章编辑 21

2.1 编辑主菜单	21
2.2 撤销	21
2.3 剪切	22
2.4 复制	24
2.5 剪切板	33
2.6 删除	35
2.7 粘贴 0 字节	36
2.8 定义选块	38
2.9 全选	42
2.10 清除选块	44
2.11 转换	44
2.12 修改	48
2.13 填入选块	55
第三章 搜索	57
3.1 搜索主菜单	57
3.2 查找文本	58
3.3 查找 16 进制字节	61
3.4 替换	63
3.5 同步搜索	65
3.6 组合搜索	70
3.7 搜索整型和浮点型	73
3.8 单词短语搜索	74
第四章 位置	75
4.1 位置主菜单	75
4.2 转到偏移地址	76
4.3 转到扇区	78
4.4 转到 FAT 入口	79
4.5 移动选块	81
4.6 向前向后	83
4.7 标记位置	83
4.8 位置管理器	84
第五章 视图	91
5.1 视图主菜单	91
5.2 仅显示文本	91
5.3 仅显示 16 进制数值	92
5.4 记录简报	94
5.5 显示	95
5.6 模板管理器	106
5.7 项目表	133
5.8 增加一列或减少一列	136
5.9 同步窗口	137
5.10 同步并比较	139
5.11 刷新	140
第六章 工具	140

6.1 工具主菜单	140
6.2 打开磁盘	141
6.3 磁盘工具	142
6.4 文件工具	164
6.5 打开内存	178
6.6 连接外部程序	182
6.7 计算器	184
6.8 16 进制快速转换	185
6.9 图形化分析	185
6.10 计算哈希值	191
6.11 哈希数据库	192
6.12 强大的脚本语言	193
6.13 方案工程	232
第七章 专业工具	234
7.1 专业工具主菜单	234
7.2 获取卷快照	234
7.3 详细技术报告	238
7.4 设置镜像文件为磁盘	240
7.5 RAID 重组功能	242
7.6 收集剩余空间	243
7.7 收集占用空间	245
7.8 收集分区间隙	246
7.9 收集文本	247
7.10 文件批量命名	249
7.11 文件内容替换	251
7.12 加亮	253
第八章 选项	253
8.1 选项主菜单	253
8.2 常规选项	253
8.3 目录浏览器	256
8.4 附加程序	258
8.5 撤销管理	258
8.6 安全管理	259
8.7 编辑模式	260
8.8 字符集	260
8.9 目录浏览器补充	263
第九章 常见数据恢复思路	267
9.1 实战导读	267
9.2 大类一：MBR 故障	267
9.3 大类二：DBR 故障	271
9.4 大类三：MFT 记录头破坏	274
9.5 大类四：80 属性故障	276
9.6 大类五：手工提取文件实例	277
第十章 WINHEX 脚本编程高级课堂	282

10.1 脚本导读	282
10.2 再解语法（案例）	283
第十一章 WINHEX 配置详解	373
11.1 汉化	373
11.2 再论常规设置	375
11.3 目录浏览器的深入探讨	385
11.4 详解模板头	

WINHEX 高级专题 七

进入机器世界

作为一个只有不足 2MB 大小的软件,WINHEX 中凝结了太多智慧和奇迹,分析 WINHEX 自然是一项令人兴奋的工作.逆向工程的思路是从扎实的知识积累中锻造而成的,比如下面这段代码,即使你的汇编功底仍不过关,你起码因该看出 WINHEX 的开发语言是 Delphi.

start:

```
CODE:00577238      push     ebp
CODE:00577239      mov      ebp, esp
CODE:0057723B      add      esp, 0FFFFFFFh;其实为 ebp-14
CODE:0057723E      push     ebx
CODE:0057723F      push     esi
CODE:00577240      push     edi
CODE:00577241      xor      eax, eax
CODE:00577243      mov      [ebp-14h], eax ;局部变量赋值
CODE:00577246      mov      eax, offset dword_576EA8
CODE:0057724B      call     @@InitExeSqqrpv
.....
@@InitExeSqqrpv proc near                                ; CODE XREF:
CODE:0057724B      push     ebp
CODE:004062E8      push     ebx
CODE:004062E9      mov      ebx, eax
CODE:004062EB      xor      eax, eax
CODE:004062ED      mov      ds:TlsIndex, eax
CODE:004062F2      push     0                                ; lpModuleName
CODE:004062F4      call     GetModuleHandleA_0;API 函数,获
取模块句柄.
CODE:004062F4
```

CODE:004062F9	mov	ds:hInstance, eax;句柄传入变量.
CODE:004062FE	mov	eax, ds:hInstance
CODE:00406303	mov	ds:dword_5780C4, eax
CODE:00406308	xor	eax, eax
CODE:0040630A	mov	ds:dword_5780C8, eax
CODE:0040630F	xor	eax, eax
CODE:00406311	mov	ds:dword_5780CC, eax
CODE:00406316	call	@SysInit@_16395
CODE:0040631B	mov	edx, (offset s_123456789abcd+0Fh)
CODE:00406320	mov	eax, ebx
CODE:00406322	call	@System@@@StartExe\$qqrp23
CODE:00406327	pop	ebx
CODE:00406328	retn	
CODE:00406328		
CODE:00406328 @@InitExe\$qqrp23 endp		

这种判断从何而来?不妨告诉大家,因为 call InitExe 这句,在 Delphi 中, InitExe 函数用来初始化 Hinstance 和模块信息表.是不是很简单?可是对 Delphi 并不了解的人来说,这些代码势必让他陷入茫然.

积累!还是积累!没有无用的知识,只有侥幸的意识!

今天我们要开始系统地感受机器世界的魅力.

首先我们应该了解 8086 系列处理器的三种工作模式:

保护模式.这是 286 时代就出现的跨越式功能,可直到 386 时才被真正使用,当时着实让 WINDOWS 3.0 风光了一把.保护模式的特点有:采用线性虚拟内存方式 (flat),内存寻址达 4GB.程序之间的内存访问行为受到严格限制;程序的内存空间保持高度独立;采用分页机制;我们所说的 WIN32 下的编程肯定是基于保护模式的编程.此外保护模式把程序权限划分为 4 个优先级,0 级权限最大,也就是我们平时所称的内核级,3 级最小,也就是我们所说的应用级(应用程序).保护模式下有自己的特权指令和系统控制寄存器,如果熟悉了这些指令的用法,我可以说,你已经具备了一个操作系统设计者的基本素质.WINHEX 常常被广大朋友们称为底层,

其实它跟底层完全不沾边,它用到的函数全部来自于用户级的 API 函数,这些函数进过多层跳转,才能调用核心系统函数.WINHEX 是一个典型的 WIN32 应用程序
虚拟 8086 模式:也就是我们在 XP 下用到的命令行,这是为了在保护模式下执行实模式软件而开发的特有的处理器功能,而且实现了多任务(每个 CMD 都有自己的内存空间).虚拟 8086 对我们今后的学习很有帮助,我们有大量的 debug 例程需要掩饰.

实模式:也称实地址模式,由于 8086 地址总线的限制,实地址模式最大只能寻址 1M 的内存空间,除去 BIOS 等所用留给程序员的只剩可怜的 640KB.当然如今的处理器拥有保护模式实模式互相切换的指令.DOS 下的程序都属于实模式程序,安全没有保障且单任务运行.实模式编程是痛苦的梦魇,但是对于我们数据恢复人员来说,非常重要,否则你这辈子都不会明白 PC3000 是怎么控制磁盘的了.还有一种辅助的系统管理模式,主要用于电源管理,也有指令可以操作,我们先不关注.我们初期以 WINHEX 为蓝本的教学任务,是让大家了解保护模式下用户态的编程,是汇编语言中比较容易掌握的,后期我们将充分利用自己设计的驱动程序,感受内核态操作的快乐.实模式下的 8086 作为参照教学我们也会涉及.

首先千篇一律的,我们需要了解汇编语言的基础,寄存器和内存.寄存器的详细知识我没有精力来讲,大家只要明白,能用寄存器存储数据就要用,这是因为寄存器比内存拥有更快的传输速度.安腾处理器用于高端运算就是因为它拥有高达 128 个寄存器.而目前主流的 X86 系列只拥有 32 位寄存器,16 位寄存器,8 位寄存器和段寄存器.我们学习汇编,应当将牢牢记住每个寄存器的默认用途,当然必要时我们可以强行改变这些用途,但会使程序难以维护.指令指针寄存器 EIP/IP 非常重要,大家要多下功夫.汇编另外一个强大的特点是内存寻址,寻址方式灵活多样,中括号代表一个地址所保存的数据,中括号内便是该数据的地址.系统寄存器\浮点寄存器\多媒体寄存器暂时不是我们关注的范围.

有了存储空间,接下来考虑的就是如何使用.CPU 采用指令集的方式为程序提供最基础的保障,其实我们所说的汇编指令就是相应的 CPU 指令集.今天我们只来关注最常用的指令 MOV,我们不妨在 DEBUG 中查看它们的结果:

MOV AX,6

```

C:\Windows\system32\debug.exe
-A
0B31:0100 mov ax,1000
0B31:0103 mov ds,ax
0B31:0105 mov ax,6
0B31:0108
-t
AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B31 ES=0B31 SS=0B31 CS=0B31 IP=0103 NV UP EI PL NZ NA PO NC
0B31:0103 8ED8 MOV DS,AX
-t
AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0105 NV UP EI PL NZ NA PO NC
0B31:0105 B80600 MOV AX,0006
-t
AX=0006 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0108 NV UP EI PL NZ NA PO NC
0B31:0108 3E DS:
0B31:0109 4E DEC SI

```

我们从上图可以看出第一条指令执行后,AX 为 1000, 我们成功把立即数 1000 装入了 AX 中.第二条指令将 AX 中的内容装入 DS 中,这实质上是一种复制,AX 和 DS 都成为 1000.最后将立即数 6 装入 AX,AX 原先的 1000 被覆盖,显示为 0006.

MOV BX,5

```

C:\Windows\system32\debug.exe
-A
0B31:0100 mov bx,2000
0B31:0103 mov es,bx
0B31:0105 mov bx,5
0B31:0108
-t
AX=0000 BX=2000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B31 ES=0B31 SS=0B31 CS=0B31 IP=0103 NV UP EI PL NZ NA PO NC
0B31:0103 8EC3 MOV ES,BX
-t
AX=0000 BX=2000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B31 ES=2000 SS=0B31 CS=0B31 IP=0105 NV UP EI PL NZ NA PO NC
0B31:0105 BB0500 MOV BX,0005
-t
AX=0000 BX=0005 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B31 ES=2000 SS=0B31 CS=0B31 IP=0108 NV UP EI PL NZ NA PO NC
0B31:0108 3E DS:
0B31:0109 4E DEC SI

```

上图中,我们先将立即数 2000 装入 BX,再将 BX 装入 ES,最后为 BX 赋值 5

MOV BX,AX

```

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B31 ES=2000 SS=0B31 CS=0B31 IP=010A NV UP EI PL NZ NA PO NC
0B31:010A 99 CWD

```

执行上述命令后,因为原本 AX 为 0, 所以 BX 也变成 0

MOV BX,[AX]

MOV [DI],AX

MOV WORD PTR [DI],666h

```
AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0105 NV UP EI PL NZ NA PO NC
0B31:0105 BF0005 MOV DI,0500
-t

AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0108 NV UP EI PL NZ NA PO NC
0B31:0108 C7056606 MOV WORD PTR [DI],0666 DS:0500=0000
-t

AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=010C NV UP EI PL NZ NA PO NC
0B31:010C 8905 MOV [DI],AX DS:0500=0666
-t

AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=010E NV UP EI PL NZ NA PO NC
0B31:010E 99 CWD
```

上图中,我们首先给寄存器 **DI** 传入 500,然后以 **DI** 为地址,装入 0666,最后将 **AX** 的值也付给 **[DI]**,也就是地址 500 处

MOV DS:[0004],BX

```
AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0111 NV UP EI PL ZR NA PE NC
0B31:0111 8ED8 MOV DS,AX
-t

AX=1000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0113 NV UP EI PL ZR NA PE NC
0B31:0113 BB4404 MOV BX,0444
-t

AX=1000 BX=0444 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=0116 NV UP EI PL ZR NA PE NC
0B31:0116 891E0400 MOV [0004],BX DS:0004=0000
-t

AX=1000 BX=0444 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0500
DS=1000 ES=0B31 SS=0B31 CS=0B31 IP=011A NV UP EI PL ZR NA PE NC
0B31:011A 99 CWD
```

上面的指令只是显式指定了段寄存器 **DS** 而已,其它都是一样的。

下面这段代码较为复杂,但其中多次用到了 **MOV**,大家可以考虑一下它们的用途。其中有个 **MOVZX** 指令非常显眼,其实它也是 **MOV** 家族的一员,称为零扩展传送指令,它的用途是将比目的操作数位宽小的原操作数用零扩展高位,比如将字节型高位插入 0 扩展成字型。

.386;处理器兼容类型

.model flat,stdcall ;内存模式为平坦,过程调用 STDCALL

option casemap:none ;强制大小写敏感

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\user32.lib

include Irvine32.inc

includelib \masm32\lib\Irvine32.lib

main proto,:byte ;main 函数原型声明

.data; 数据段

var word 1000h ;生命字型变量

var2 db 10 dup (100 dup (0AAh)),0;声明一个 10*100+1 数组变量 初始值 AAh

.code;代码段

start::;代码开始

movzx ecx,var 将 var 送至 ecx 并零扩展成双字

invoke main,0BBh 调用 main 过程 参数传入 0xBB

exit ;终止进程

main proc uses edi esi,ARRAY:byte; main 过程声明保护 edi esi,声明字节形参

mov eax,ecx;将 ecx 的值传入 eax

call WriteInt ;执行 WriteInt 过程

movzx eax,ARRAY 将参数 ARRAY 零扩展传送至 eax

call WriteInt;执行 WriteInt 过程

mov ebx,offset var2;将 var2 的地址传入 ebx

xor ecx,ecx;ecx 清零

push edx;edx 入栈

push ebx;ebx 入栈

mov edx,40h ;edx 传入 40h

.while ecx< sizeof var2 ;当 ecx 小于数组 var2 的长度时

mov [ebx],edx 将 edx 内容传入 ebx 地指处

add ebx,1;ebx 加 1

add edx,1;加 1

add ecx,1;加 1

.endw; 循环结束

pop ebx;出栈

pop edx;出栈

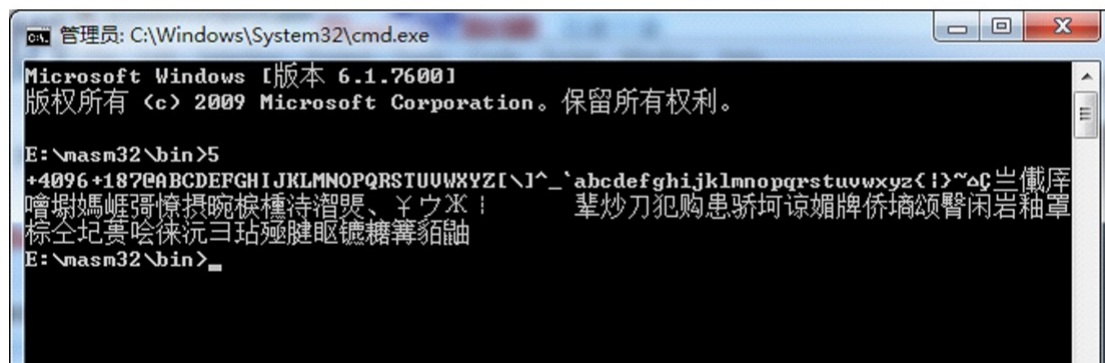
mov edx,ebx;ebx 传入 edx 中

call WriteString;执行 WriteString 过程

ret ;恢复 EIP 的值.

main endp;main 过程结束

end start; 程序从 Start 开始执行



从上图可以看出运行结果,我们立刻用 IDA PRO 反汇编这个文件,首先我们关注启动部分:

```
.text:00401000                public start
.text:00401000 start          proc near
.text:00401000                movzx    ecx, word_403000
.text:00401007                push     0BBh
.text:0040100C                call     sub_401018
.text:0040100C
.text:00401011                push     0                ; uExitCode
.text:00401013                call     ExitProcess
.text:00401013
.text:00401013 start          endp
```

上述代码是程序的开始环节,非常重要,首先 **public start** 指明了 **start** 标号开始的程序是可以被其它模块访问到的.从第二行我们可以看出,**start** 其实在背后生成了一个近过程,也就是段内调用的过程.第三行和我们编写的语句相差无几,将变量零扩展至 ECX 中.第四行和第五行共同调用了一个过程,其实就是我们的 **main**,只是名字由于没有识别而用 IDA PRO 自己的方式予以重命名.第五第六行调用

了 WINDOWS API 函数 `ExitProcess`,最后 `start` 过程结束.怎么样,和我们编写的程序是不是非常相似,我们几乎就是在阅读源码了.可喜的是,IDA PRO 甚至为我们分析出了 `ExitProcess` 的参数名称.

在 IDA PRO 中双击 `sub_401018` 名称,该工具会自动完成向该子过程的跳转:

```
sub_401018      proc near                                ; CODE XREF: start+C↑ p
.text:00401018
.text:00401018 arg_0      = byte ptr 8
.text:00401018
.text:00401018          push    ebp
.text:00401019          mov     ebp, esp
.text:0040101B          push    edi
.text:0040101C          push    esi
.text:0040101D          mov     eax, ecx
.text:0040101F          call   sub_4018CA
.text:0040101F
.text:00401024          movzx   eax, [ebp+arg_0]
.text:00401028          call   sub_4018CA
.text:00401028
.text:0040102D          mov     ebx, offset s_Kkkkkkkkkkkkkk ;
.text:00401032          xor     ecx, ecx
.text:00401034          push    edx
.text:00401035          push    ebx
.text:00401036          mov     edx, 40h
.text:0040103B          jmp     short loc_401048
.text:0040103B
.text:0040103D ; -----
.text:0040103D
.text:0040103D loc_40103D:                                ; CODE XREF:
sub_401018+36↑ j
.text:0040103D          mov     [ebx], edx
```



```

.text:0040103F      add     ebx, 1
.text:00401042      add     edx, 1
.text:00401045      add     ecx, 1
.text:00401045
.text:00401048
.text:00401048 loc_401048:                                ; CODE XREF:
sub_401018+23↑ j
.text:00401048      cmp     ecx, 3E9h
.text:0040104E      jb     short loc_40103D
.text:0040104E
.text:00401050      pop     ebx
.text:00401051      pop     edx
.text:00401052      mov     edx, ebx
.text:00401054      call   sub_40192B
.text:00401054
.text:00401059      pop     esi
.text:0040105A      pop     edi
.text:0040105B      leave
.text:0040105C      retn    4
.text:0040105C
.text:0040105C sub_401018      endp

```

有了上面的经验,该子过程的代码费不了多少精力我们也可以分析出来:

sub_401018 **proc near** 定义了一个段内转移的过程,后面的注释指出了该子过程在 **start** 程序中的偏移位置,非常有助于我们的分析. **arg_0 = byte ptr 8** 一开始会让人费解,可是观察多了就会发现,它只是为数字 8 创建了一个代号 **arg_0** 而已,这样做的目的是避免人们陷入可怕的数字漩涡中,但是不少高手可能会不适应,他们更希望直观地看出其中的数学关系. **push ebp, mov ebp, esp** 是 C/C++/Stdcall 调用约定中都要遵循的环节,也就是利用栈结构传递过程参数.首先将 **ebp** 压栈,然后将栈指针付给 **ebp**,由 **ebp** 的相对位置来间接寻址局部变量和

参数,程序结束时会恢复栈指针和 `ebp`. 以后几行代码与我们编写的程序并无二致 `movzx eax, [ebp+arg_0]`有所不同,我们在程序中是将参数零扩展传送至 `eax`,现在参数不见了,取而代之`[ebp+arg_0]`是怎么回事呢?其实仔细想想就会恍然大悟,参数被保存在占空间中,参数又先于 `EBP` 和 `EIP` 压栈,所以`[ebp+8]`就是我们需要的参数 `ARRAY`.红色标注的代码差别最大,在我们的代码中用了伪指令 `WHILE`,在这里则全部用 `CPU` 指令来实现,请大家对照参考书籍先重点分析这个循环是如何实现的.记住,分析软件流程全靠这些知识,牢牢掌握颇有裨益.最后的 `leave` 指令将 `ebp` 弹出,`ret 4` 释放参数堆栈,在 `win32` 汇编中,即使参数设定为 `byte` 类型,可依然会按照 `Dword` 类型压栈,也就是该参数实质占用了 4 字节空间,这里 `ret 4` 就是先将返回地址出栈,再将参数释放的意思,有 `N` 个参数就是 `ret N`.

反汇编代码中还有其它内容,其实就是我们引入的 `.inc` 文件中的代码,我们下一讲继续分析.

汇编语言的学习一定要与我们的教程同步进行,另外 `C#` 也不可放下,高级语言和低级语言应该相互促进.

WINHEX 高级专题 第八讲

机器码与一次真正的 NTFS DBR 教学

对新技术的渴求一直是每个数据恢复从业者的工作动力,可是我们不得不面对一个残酷的现实:大部分工作都是千篇一律的:10 年不变的文件系统,10 年不变的扫描工具,10 年不变的磁盘阵列,这些陈年八股的技术如同腐朽的思想,一点点蚕食我们年轻活跃的灵魂.的确,很多人在 10 年前本有机会跨入高手的行列,从荣誉和物质的最高点绘制新人生的美丽蓝图,可今天,依然蜷缩在电子城的一角,顶着维修技师的名号,眼睁睁看着市场的汪洋用价格战的巨浪将自己苦心经营的乐土吞噬.

不是没有拼搏的精神,也不是没有一颗富于智慧的心,只是遗憾的错过了学习的最佳时光,不是不学,而是浮光掠影式的学习方法,消极被动的学习态度,换来的只是肤浅的,易学的所谓知识财富.我相信,不少人当初选择这条路是因为它好走!从根本就抱着“容易才学”的心态.

数据恢复本来是一门高深的学科,一个合格的数据恢复工程师应该具备的基本技能,是从软硬件两方面考虑的.软件方面,不懂高级语言编程,怎么快速实践自己的思路?不懂机器语言编程,怎么观察和分析各种存储格式和数据分配细节?没有良好的算法设计功底,面对未来海量的数据存储和有限的工作时间,又将何去何从?硬件方面,对数字逻辑一无所知,面对电路和芯片一脸迷惘,看 VHDL 如品天书,每天还得意洋洋的拿着风枪烙铁,满面春风地对别人说:我是搞硬件的!回到家里,心里是否浮过一丝丝的失落?的确,以这些标准来衡量,合格的并不多,可拥有这些技能的,活跃在其它领域的高人们却不在少数,如果他们有一天突然进入这个领域,瓜分这个市场,很多人恐怕连一口残羹也喝不上.除此之外,我们身边更有许许多多的在向这个目标迈进的勇者,用不了几年,他们将是这个行业的真正能获得利益和荣耀的人物.而停止不前的,那些所谓“啃老本”的,注定要烟消云散.

从宏观来看,今天的数据恢复在中国已经几乎沦为笑柄,,从它诞生的那一天,似乎就在走下坡路,在”高新技术”的光环下,价格一降再降,店面越开越多,可以说尚未迎来辉煌就已跌入深谷.而制造这一切的罪魁祸首,恰恰是我们自己!的确,一种谁学谁会的技能,还有什么利益化可言?大公司抄袭软件,小公司降价揽客,一个图表从这本书转移到另一本书,一个教程从一个论坛转移到另一个论坛,每个人

都在等待别人从别人那获取技术教给自己,分区表,引导扇区学了一遍又一遍乃至倒背如流.至于更低层次的软件扫描,恐怕已经成为电脑桌面上的大众化工具.这种背景下,很多人已经不能再叫做技术人员了,因为手上的技术正在普及,优势正在淡化.

特殊文件系统恢复技术在国外早就普及,从印度的凤凰系列就能看出这一点.曾经听很多人说,国内没有 Ext 的资料,所以无法掌握 Linux 数据恢复技术,那国外又是怎么写出这些资料的呢?很简单,Linux 是开源的,到文件系统源码那部分看一眼就了解了,恐怕随便一个入门级程序员都能做到,而国内的工程师们竟为这么一个简单的,既有的技术等了很多年,直到某些教材的出版.

还是那句话:觉醒吧!要么就睡着死去.每一讲的开头我都要不厌其烦的讲这些道理,如果有一天很多人开始走上正轨了,我就闭嘴了.

言归正传!本讲的标题称为: 机器码与一次真正的 NTFS DBR 教学,难道以前的 NTFS DBR 资料都是虚假的?并不是那样,以前的资料有一个共同点:一个 DBR 的 WINHEX 截图,加上一张按照偏移量逐字解释的表.这样做的好处是大家可以快速的记住这张表,然后对照表中的条条框框来判断 DBR 的故障,与表上不符的,我们就更改到符合为止,我相信任何人只要认真些,都可以在一周内掌握这些知识.这种学习方法叫“对比法”,靠的是死记硬背,不必去理会原理,有多少文件系统,就得掌握多少相关的表,如果没有相应的表资料,就称文件系统特殊.....无法.....

跳转指令	00-02	3
名称“NTFS” OEM	03-0A	8
每扇区字节数	0B-0C	2
每簇扇区数	0D-0D	1
保留扇区数	0E-0F	2
未使用	10-14	5
介质描述符	15-15	1
未使用	16-17	2
每磁道扇区数	18-19	2
每柱面磁头数	1A-1B	2
隐含扇区数	1C-1F	4
未使用	20-23	4
未使用	24-27	4
文件系统扇区总数	28-2F	8
MFT 起始簇号	30-37	8

每 MFT 项大小	40-40	4
未使用	41-43	3
每索引簇数	45-47	4
序列号	48-4F	8
校验和	50-53	4
引导代码	54-1FD	426
扇区有效标志 55AA	1FE-1FF	2

(以前的资料:表)

我不得不指出,对比法有个根本缺陷就是:知其然不知其所以然,也养成了今天依靠资料却难得资料的尴尬局面.在大家眼中,文件系统似乎就是一张张表而已,而忽视了它本身也是一段计算机程序的本质.

通过对程序的分析,我们完全可以得到比对比法更理性的认识,因为我们将了解 DBR 的工作方式,和造成其故障的根本原因.即便针对没有资料的文件系统,我们也可以通过程序分析逆向出资料来.

程序分析法应该也必须是我们的基本技能,否则我们只能做一个技术乞讨者.但是这项基本技能需要背后勤奋的努力,要分析程序,首先要会编写程序.前面几讲,我以 WINHEX 为蓝本,把汇编语言拖入大家的眼帘.我分析 WINHEX,根部目的不是为了“怪用”WINHEX,而是想让大家了解汇编语言的强大和奇妙之处,从而激起大家的学习兴趣,最终“活用”WINHEX.

很多人也许今天还不明白 WINHEX 的含义,WIN 是 WINDOWS 这没什么说的,而 HEX 正是 INTEL 机器语言文本格式的简称,可以说 WINHEX 里面展示的,就是机器码而已,如果我们水平足够高,完全可以用 WINHEX 来进行机器语言的编程.而汇编语言就是机器语言,他们是同一种语言不同的书写形式,汇编语言更容易为人所接受,WINHEX 的数据解释器里面就有机器码到汇编语言的转换功能,可见作者当初开发时的目的并不仅仅局限于满足当前取证工作需要而已.其实掌握了机器指令的人,从理论上讲,已经没有任何计算机软件技术能对其造成障碍,答案就悬挂在时间和毅力的高塔上.而 WINHEX 对掌握机器指令的人来说,会展现出一个全新的世界.试想有一天,我们打开 WINHEX,就读懂了一串串 16 进制的含义,该是多么幸福的事情.今天我就利用 NTFS DBR,向大家展示程序分析法的强大之处.

程序无外乎数据和指令.大家学习数据恢复这么多年,一定能从一个 DBR 中大致判断出数据和指令,本分区参数记录表 BPB 是我们修复引导扇区的关键

部位，里面就存放着许多数据，他们都是描述本分区特征的关键参数，这些参数存在的目的就是为了被指令调用，如果这些参数出现错误，指令调用时就会产生错误的计算结果，那分区引导就可能出现问题。其实这些参数之所以有名有姓，是因为在当初编写程序时，就利用了“结构”，这种数据组织方式。一个结构描述一段存储空间，结构成员再利用数据宽度，保存相应的数据。往往，指令是不变的，而数据有可能在不同的环境下呈现出多种形态，比如“簇大小”这个数据就不是恒定的，它是由创建分区时被人为指定产生的，但 DBR 的引导指令从来都是不变的，我们在修复时可以从其它分区借鉴过来。正因为如此，我们始终不了解 DBR 中的指令究竟怎样执行，怎样和数据交互。

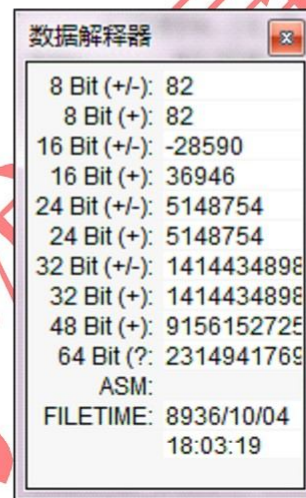
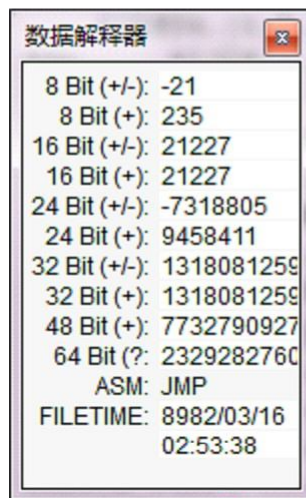
我们还是以实例来说明：**DBR 的前两个字节 EB52**，许多人以为是固定的数据。我们可以细细分析一下：引导扇区作为首先且唯一载入内存的程序，范围已经固定在 512 字节，总不可能将大段的数据无限制的堆在程序开头使 CPU 出现混淆，所以它很可能是一个指令，是什么指令呢？这当然有个机器码表可以查询我们称之为 OPCODE，也就是操作码的意思。那么我们到哪里找这张表呢？既然是 X86 的机器码，那当然要到 INTEL CPU 的开发手册上去找（从 INTEL 官网下载），我们先搜索 EB，果然有：

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
EB cb	JMP rel8	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits

图片中的内容大致告诉我们机器码 **EB+地址** 的形式，是一种直接转移指令，而且是短转移，转移范围在一个 8 位整数内也就是 -128 到 127 之间，既然是直接短转移，那么他的汇编语言模式应当为 **JMP 8 位整数**。这里 8 位整数的位置刚好被 52 占据，那么不就是从当前位置跳转 52 个字节的意思吗？注意，这里是直接转移，是从当前位置转移 52 个字节，而不是跳到地址 52，EB52 本身从零开始又占用 2 个字节，再偏移 52 个字节，此时 CPU 的执行地址被更改到 2+52 为 54。最终的含义为：跳转到偏移 54 去执行：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	EB	52	90	4E	54	46	53	20	20	20	00	02	08	00	00	
00000010	00	00	00	00	00	F8	00	00	3F	00	FF	00	3F	00	00	00
00000020	00	00	00	00	80	00	80	00	9C	8F	8B	08	00	00	00	00
00000030	00	00	0C	00	00	00	00	00	F9	B8	88	00	00	00	00	00
00000040	F6	00	00	00	01	00	00	00	01	00	F5	A0	34	F5	A0	72
00000050	00	00	00	00	FA	33	C0	8E	D0	BC	00	7C	FB	B8	C0	07

从上图可以看出,偏移 54 处为 FA 刚好是 BPB 结束,引导指令开始的位置,这下大家该明白一点了吧,EB52 的作用就是跳过 BPB 去执行之后的指令.即使不了解 BPB,我们也应该猜到跳过的这一段八成是数据.



数据解释器也告诉我们 EB 是 JMP,而 52 解释不出来自然是一个值了.我们也可以用 2 进制编程来解码,例如,EB 转换成 2 进制是 11101011,我们在 CPU 开发手册查找 11101011:

Instruction and Format	Encoding
JMP - Unconditional Jump (to same segment)	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : full displacement
register indirect	1111 1111 : 11 100 reg
memory indirect	1111 1111 : mod 100 r/m

上图中是 JMP 的几种转移方式,其中 short 正好对应 11101011,我们就知道 11101011 是 JMP 短转移的二进制码而,而近转移、远转移、寄存器间接转移和内存间接转移都在表中有其表现形式.

看来,INTEL 不遗余力地在其开发手册中解释机器码和汇编语言之间的关系.但是 EB52 只是很简单的指令而已,有许多较长的复杂指令,要靠反复的摸索来实现,如果有兴趣学习,那么 OPCODE 表就是你每天都要熟悉的工具,那样过个

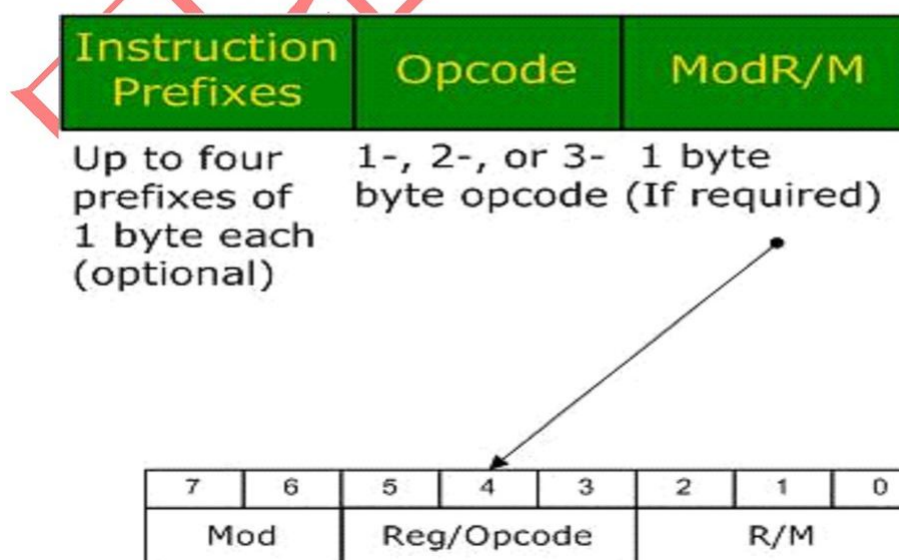
N年后,你就是传说中可以用2进制编程的高手了。那么引导代码开头的FA又是什么呢,其实它是一个中断禁止,这是因为DBR在引导过程中有个向内存映射的过程,BIOS将会在映射地址开始执行,从而激活DBR的引导过程,此刻堆栈指针寄存器SS:SP需要同时装入这个映射地址,但是在对SS和SP操作的时候,如果有中断发生,中断的保存现场的操作把相关寄存器值保存到SS:SP指向的地址,这时候指向的则是中断保护的数据而不是映射地址的数据.会导致系统崩溃.所以FA是必须的,待SS:SP赋值完成后我们可以中断使能,恢复中断功能.仔细想想FA是不是经常看到?某些MBR的第一个字节就是FA,这样的MBR才更健壮.

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FA	CLI	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

上图所示FA的汇编模式为CLI,作用是清除中断标志.紧接着后面的33C0似乎更熟悉,很多MBR的前两个字节都是它,可它为什么又跑到DBR中来了呢?其实它并没有什么特殊的作用,仅仅是对AX中的值清零而已:

33/r XOR r16, r/m16 Valid Valid r16 XOR r/m16.

原来33是XOR指令在16位寄存器为目的操作数,而16位寄存器或内存为源操作数时使用,那么目的操作数肯定是一个寄存器,此时的机器码就没有那么简单了因为牵扯到寄存器,此时我们就不得不讲一下机器指令的格式规范.



如上图所示第一部分是指令前缀,一般占用1个字节,指令前缀只有在调用非

当前模式寄存器时才会出现,比如当前程序运行在 16 位模式,却调用了 32 位寄存器 EAX,此时就会出现指令前缀 66.第二部分是操作码,也就是汇编指令,大多数汇编指令只占用一个字节,第三部分 MOD-REG-R/M 字段规定指令的寻址方式.这里我们只能简单讲解,因为机器语言体系庞大,有近两万中变换方式(一定要牢牢掌握).我们只用 XOR 指令来解释:这里 33 转换成 2 进制为 00110011,其中 001100 是固定操作码,最后的 11 称为 DW 标志,其中 D 为 1 时表示寄存器为目的操作数,W 为 1 时表示是一个 16 或 32 位寄存器,11 自然是两个标志都为 1,第二个字节 C0 转换为 2 进制 11000000,也就是 MOD 为 11,REG 为 000,R/M 为 000. MOD 为 11 表示 R/M 是一个寄存器号,此时 REG 和 R/M 都为 000,按照寄存器符号表 000 为 AX 规定,此指令的汇编原型应该是 XOR AX,AX 也就是 AX 与自己异或,按照异或运算相同为 0 不同为 1 的原则,AX 最终为 0.此指令的目的是为 AX 清零.

上面的指令依然只能算是简单,因为还没有牵扯到内存寻址和立即数操作,更复杂的模式我会找机会给大家演示,但是苦工还是要下的,否则机器语言当真是一座万里长城,只能让人兴叹.

如果机器指令还没学到家,我们完全可以借助反汇编工具来尽心分析:

```
* seg000:0054 FA          cli
* seg000:0055 33 C0       xor     ax, ax
* seg000:0057 8E D0       mov     ss, ax
* seg000:0059 BC 00 7C     mov     sp, 7C00h
* seg000:005C FB       sti
* seg000:005D B8 C0 07     mov     ax, 7C0h
* seg000:0060 8E D8       mov     ds, ax
seg000:0062
* seg000:0062 E8 16 00     call    sub_7B
* seg000:0065 B8 00 0D     mov     ax, 0D00h
* seg000:0068 8E C0       mov     es, ax
seg000:006A
* seg000:006A 33 DB       xor     bx, bx
* seg000:006C C6 06 0E 00 10 mov     byte ptr ds:0Eh, 10h
* seg000:0071 E8 53 00     call    sub_C7
* seg000:0074 68 00 0D     push    0D00h
* seg000:0077 68 6A 02     push    26Ah
* seg000:007A CB       retf
```

上图是 IDA PRO 反汇编的结果,机器码和汇编语言都有显示,我们可以看到刚才我们自己的机器码解析成果还是完全正确的!我再不妨为大家翻译一下汇编语言的含义:

第一行中断禁止.

第二行 ax 清零.

第三行将 ax 传入 ss,也就是将 0 传入 ss

第四行 7c00 传入 sp,也就是堆栈顶设为 7c00

第四行中断使能

第五行将 7c0 传入 ax

第六行将 ax 传入 DS(证明数据段从 7c0 开始)

第七行执行 sun7b 地址处的过程:

```
seg000:007B      sub_7B      proc near
seg000:007B  8A 16 24 00      mov     dl, ds:24h
seg000:007F  B4 08      mov     ah, 8
seg000:0081  CD 13      int     13h
seg000:0081
seg000:0081
seg000:0081
seg000:0083  73 05      jnb     short loc_8A
seg000:0085  B9 FF FF      mov     cx, 0FFFFh
seg000:0088  8A F1      mov     dh, cl
seg000:008A
seg000:008A      loc_8A:      movzx   eax, dh
seg000:008A  66 0F B6 C6      inc     ax
seg000:008E  40      movzx   edx, cl
seg000:008F  66 0F B6 D1      and     dl, 3Fh
seg000:0093  80 E2 3F      mul     dx
seg000:0096  F7 E2      xchg    cl, ch
seg000:0098  86 CD      shr     ch, 6
seg000:009A  C0 ED 06      inc     cx
seg000:009D  41      movzx   ecx, cx
seg000:009E  66 0F B7 C9      mul     ecx
seg000:00A2  66 F7 E1      mov     ds:20h, eax
seg000:00A5  66 A3 20 00
seg000:00A9  C3      retn
```

注意上图 sun7b 地址处的过程就开始用到 BPB 中的参数,将 DS:24 地址处的字节传入 dl 中,由于前面的代码在 DBR 扇区加载进内存后已经将扇区起始装入 DS 所以, DS:24 就是 DBR 的第 24 个字节,所以我们直接看 24 字节有什么:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	EB	52	90	4E	54	46	53	20	20	20	20	00	02	08	00	00
00000010	00	00	00	00	00	F8	00	00	3F	00	FF	00	3F	00	00	00
00000020	00	00	00	00	80	00	80	00	9C	8F	8B	08	00	00	00	00

看到了吗?是 80,这个 80 在以前的资料中注解为总是 80 或未知的,现在大家往下看应当清楚了吧,这是一个 INT13 中断的调用,80 表示驱动器为硬盘,这是告诉中断执行的对象是硬盘,而下面的 MOV AH,8 是决定调用 INT13 的哪项功能,这里用了第 8 号功能:读取驱动器参数.也就是说 DBR 在开始运行时要读取硬盘参数.读取成功后跳转到 008A 这个地方:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	EB	52	90	4E	54	46	53	20	20	20	20	00	02	08	00	00
00000010	00	00	00	00	00	F8	00	00	3F	00	FF	00	3F	00	00	00
00000020	00	00	00	00	80	00	80	00	9C	8F	8B	08	00	00	00	00
00000030	00	00	0C	00	00	00	00	00	F9	B8	88	00	00	00	00	00
00000040	F6	00	00	00	01	00	00	00	01	00	F5	A0	34	F5	A0	72
00000050	00	00	00	00	FA	33	C0	8E	D0	BC	00	7C	FB	B8	C0	07
00000060	8E	D8	E8	16	00	B8	00	0D	8E	C0	33	DB	C6	06	0E	00
00000070	10	E8	53	00	68	00	0D	68	6A	02	CB	8A	16	24	00	B4
00000080	08	CD	13	73	05	B9	FF	FF	8A	F1	66	0F	B6	C6	40	66

我们看 WINHEX 中 8A 位置跟反汇编器显示的机器码一样就是 66 0F B6 C6.

这条指令将 INT13 返回的磁头数扩展传送至 EAX 中,注意此时在 16 位模式下应用 32 位寄存器,所以出现了指令前缀 66.接下来用到了多个 INT13 返回参数,所以这些功能我们必须知道:

CH=柱面数的低 8 位

CL 的位 7-6=柱面数的高 2 位

CL 的位 5-0=扇区数

DH=磁头数

DL=驱动器数

这样,我们来分析一下接下来的程序:

```

inc      ax          ax 加 1
movzx    edx, cl     将 cl 扩展传送至 edx
and      dl, 3Fh     将 dl 中内容与 3f 相与
mul      dx          AX 中的值乘以 DX
xchg     cl, ch      交换 cl 和 ch
shr      ch, 6       将 ch 右移 6 位
inc      cx          cx 加 1
movzx    ecx, cx     cx 扩展传送 ecx
mul      ecx         EAX 中的值乘以 ECX
mov      ds:20h, eax  将乘积的低位写入 ds:20h 处
retn                      过程近返回

```

这段指令,给人的大致感觉是利用柱面扇区磁头计算出了某种结果,而结果保存在 20 处,我们看看 20 处有什么:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	EB	52	90	4E	54	46	53	20	20	20	20	00	02	08	00	00
00000010	00	00	00	00	00	F8	00	00	3F	00	FF	00	3F	00	00	00
00000020	00	00	00	00	80	00	80	00	9C	8F	8B	08	00	00	00	00

是 0，以前的资料中竟然标注未使用,可程序明明使用了这个地方,真是令人汗颜啊,资料出现了严重的误导,这里不是没使用,只是在内存中使用而已.那这个位置到会保存什么呢?下一讲我们会予以动态跟踪,解开这个谜团.

今天的内容难度极大,尤其是机器码的格式和 **OPCODE** 表是最大难点,我是没精力把它们完全讲出来,只能告诉大家一个学习方向和基本方法,剩下的靠大家自己去“挣扎”了。