编程高手箴言

书名: 编程高手箴言
作者: 梁肇新
来源: 电子工业出版社
ISBN: 7-5053-9141-0
页数: 416
开本: 16开
出版时间: 2003年11月
定价: 50元

内容简介:

本书是作者十余年编程生涯中的技术和经验的总结。内容涵盖了从认识CPU、Windows运行机理、编程语言的运行机理，到代码的规范和风格、分析方法、调试方法和内核优化，内有作者对许多问题的认知过程和透彻的分析，以及优秀和精彩的编程经验。

整理说明：

【献给CSDN上的朋友们】

在CSDN论坛上多次见到网友搜寻《编程高手箴言》一书，我本人也常常在书店里站着翻阅此书，
虽然对梁先生的部分观点实在不敢苟同，但里面一些知识点确是讲的非常不错。
所以一直也在寻找电子版。
今天正好看到有朋友帖出地址：http://act.it.sohu.com/book/serialize.php?id=71
虽然只有前三章，但已经相当不错，（个人认为前三章乃是此书精华之所在）
只不过页面在网络上，看起来太麻烦，而且很多广告链接，看得不太舒服。
于是我花了两个多小时整理出来（主要时间花在清理无用链接以及一些脚本错误，还有图片和链接的相对地址转换）。
希望能给大家带来方便，则是本人莫大欣慰。

整理者：Featured (mail: lizhaozhuo@people.com.cn)
2005.4.21 晚

第1章 程序点滴

1.1 程序≠软件(1)

现在很多人以为程序就是软件，软件就是程序。事实上，软件和程序在20世纪80年代时，还可以说是等同的，或者说，在非PC领域里它们可能还会是等同的。比如说某个嵌入式软件领域，软件和程序可能是等同的。但是，在PC这个领域内，现在的程序已不等于软件了。这是什么意思呢？

1. 软件发展简述

在20世纪80年代的时候，PC刚诞生，那时国内还没有几个人会写程序。那么，如果你写个程序，别人就可以拿来用。那时候的程序就能产生价值，那个程序就直接等同于软件。

但软件行业发展到现在，这里以中国的情况为例（美国在20世纪80年代，程序已经不等同于软件了），程序也不等同于软件了。因为现在写程序很容易，但是你的这个程序很难产生什么样的商业意义，也不能产生什么价值，这就很难直接变成软件。要使一个程序直接变成软件，中间就面临着很高的门槛问题。这个门槛问题来自于整个行业的形成。

现在，你写了一个程序以后，要面临商业化的过程。你要宣传，你要让用户知道，你要建立经销渠道，可能你还要花很多的时间去说服别人用你的东西。这是程序到软件的一个过程。这门槛已比较高了。

我们在和国内的大经销商的销售渠道的人聊天时，他们的老板说，这几年做软件的门槛挺高的，如果你没有五六百万元做软件，那是“玩”不起来的。我说：“你们就使门槛很高了。”他说：“那肯定是的。如果你写个“烂”程序，明天你倒闭了，你的东西还占了我的库房，我还不知道找谁退去呢。我的库房是要钱的呀！现在的软件又是那么多！”

所以，如果你没有一定的资产的话，经销商都不理你。实际情况也是这样的，如果你的公司比较小，且没什么名气，你的产品放到经销商库房，那么他最多给你暂收，产品销不动的话，一般两周绝对会退货。因为现在经销商可选择的余地已很多了，所谓的软件也已经很多了。而程序则更多，程序都想变成软件，谁都说自己的是“金子”。但只有经受住用户的检验，才能成为真正的“金子”。

这就是美国为什么在20世纪90年代几乎没有什么新的软件公司产生的原因。只是原来80年代的大的软件公司互相兼并，我吞你，你吃我。但是，写程序的人很多，美国的程序变软件的门槛可能比我们还高，所以很多人写了程序就丢在网上，就形成了共享软件。

2. 共享软件

共享软件是避开商业渠道的一种方法。它避开了商业的门槛，因为这个行业的门槛发展很高以后就轻易进不去了。我写个程序丢在网上，你下载就可以用，这时候程序又等于软件。共享软件是这样产生的，是因为没有办法中的办法。如果说程序直接等于软件的话，谁也不会轻易把程序丢到网上去。

开始做共享软件的人并不认为做它能赚钱，只是后来用的人多了，有人付钱给他了。共享软件使得程序和软件的距离缩短了，但是它与商业软件的距离会进一步拉大。商业软件的功能和所要达到的目标就不是一个人能“玩”得起来的了。这时的软件也已不是几个人、一个小组就能做出来的了。这就是在美国新的软件公司没法产生的原因。比如Netscape网景是在1995～1996年产生的新软件公司，但是，两三年后它就不见了。

1.1.1 商业软件门槛的形成

1. 商业软件门槛的形成

商业软件门槛的形成是整个行业发展的必然结果。任何一个行业初始阶段时的门槛都非常低，但是，只要发展到一定的阶段后，它的门槛就必然抬高。比如，现在国内生产小汽车很困难，但在20世纪50年代～60年代的时候，你装4个轮子，再加上柴油机等就形成汽车。那时的莱特兄弟装个螺旋桨，加两个机翼，就能做飞机。整个行业还没有形成的时候，绝对可以这样做，但是，到整个行业形成时，你就做不了了。所有的行业都是这样的。

为什么网站一出来时那么多人去挤着做？这也是因为一开始的时候，看起来门槛非常低，人人都可以做。只要有一个服务器，架根网线，就能做网站。这个行业处于初始阶段时，情况就是这样的。但这个行业形成后，你就轻易地“玩”不了了。

国内的软件发展也是如此。国内的软件自从软件经销商形成以后，这个行业才真正地形成。有没有一个渠道是判断一个行业是否形成的很重要的环节。任何一个行业都会有一个经销渠道，如果渠道形成了，那么这个行业也就形成了。第一名的经销商是1994年～1995年成立的，也就是说，中国软件行业大概也就是在1995年形成的，至今才经历8年时间的发展。

有一种浮躁的思想认为，中国软件产业应该很快就能赶上美国。美国软件行业是20世纪80年代形成的，到现在已经发展了20多年了。中国软件行业才8年，8岁才是一个懵懂的小孩，20多岁是一个强壮的青年，那么他们的力量是不对等的。但也要看到，当8岁变成15岁的时候，它真正的能量才会反映出来。

2. 软件门槛对程序员的影响

现在中国软件行业正在形成。所以，现在做一个程序员一定要有耐心，因为现在已经不等于以前了。你一定要把所有的问题搞清楚，然后再去做程序。

对于程序员来说，最好的工作环境是在现有的或者初始要成立的公司里面，这是最容易成功的。个人单枪匹马闯天下已经很困难了。即使现在偶尔做两个共享软件放在网上能成名，但是也已经比较困难了。因为现在做软件的人已经很多了。这也说明软件已经不等于程序了，程序也不等于软件。

程序要变成软件，这中间是一个商业化的过程。没有门槛以前，它没有这个商业过程，现在有这个行业了，它中间就有商业化的过程。这个商业化的过程就不是一个人能“玩”的。

如果你开始做某一类软件的时候，别人已经做成了，这时你再决定花力气去做，那么你就要花双倍的力气去赶上别人。

现在的商业软件往往是由很多模块组成的，模块是整个系统的一部分。个人要完整地写一个商业系统几乎是不可能的。软件进入Windows平台后，它已经很复杂了，不像在DOS的时候，你写两行程序就能卖，做个ZIP也能卖。事实上，美国的商业编译器也不是一个人能“玩”的。现在你可能觉得它是很简单的，甚至Linux还带了一个GCC，且源程序还在。你可以把它改一改，做个VC试一试，看它会有人用吗？它能变成软件吗？即使你再做个界面，它也还是一个GCC，绝对不会成为Visual C++那样能商业化的软件。

可见，国外软件行业的门槛要比中国的高很多了。我觉得我们中国即使再去做这样的东西，也没有多大的意义了。这个门槛你是追不过来的。不仅要花双倍的力气，而且在这么短的时间内，你还要完成别人已经完成过的工作，包括别人所做的测试工作。只有这样，才能做到你的软件与别人有竞争力，能与它做比较。

第1章 程序点滴

1.1 程序≠软件(2)

1.1.2 认清自己的发展

如果连以上认识都不清楚，很可能就以为去书店买一本MFC高手速成之类的书，编两个程序就能成为软件高手。就好像这些书是“黄金”，我学两下，学会了VC、MFC，就能做一个软件拿出去卖了。这种想法也不是不行，最后一定能行，但要有耐心，还要有机遇。机遇是从耐心中产生的，越有耐心，就越有机遇。你得非常努力，要花很多的精力，可能还要走很多的弯路。

如果你是从MFC入手的，或是从VB入手的，则如要做出一个真正的能应用个人领域的通用软件，就会走非常多的弯路。直接的捷径绝对不是走这两条路。这两条路看起来很快，而且在很多公司里面确实需要这样的东西，比如说我这家公司就是为另一个家公司做系统集成的，那我就需要这样的东西，我不管你具体怎么实现，我只需要达到这个目标就行了。

任何软件的实现都会有*n*种方法，即使你是用最差的那种方法实现的，也没有问题，最后它还是能运行。即使有问题，再改一改就是。但是，做通用软件就不行了，通用是一对多，你做出来的软件以后要面向全国，如果将来自由贸易通到香港也好，通到国外也好，整个产品能销到全世界的话，这时候，通用软件所有做的工作就不是这么简单了。所以说，正确的入门方法就很关键。

如果你仅仅只是想混口饭吃，找个工作，可能教你成为MFC的高手之类的书对你就足够了。但是，如果你想做一个很好的软件，不仅能满足你谋一碗饭吃，还能使你扬名，最后你的软件还能成为很多人用，甚至你还想把它作为一个事业去经营，那么这第一步就非常关键。这时就绝对不能找一本MFC或找一本VB的书学两下就行，而是要从最低层开始做起，从最基本做起。

第1章 程序点滴

1.2 高手是怎样练成的(1)

1.2.1 高手成长的六个阶段

程序员怎样才能达到编程的最高境界？最高境界绝对不是你去编两行代码，或者是几分钟能写几行代码，或者是用什么所谓的可视化工具产生最少的代码这些工作，这都不是真正的高手境界。即使是这样的高手，那也都是无知者的自封。

我认为，一个程序员的成长可分为如下六个阶段。

- 第一阶段

此阶段主要是能熟练地使用某种语言。这就相当于练武中的套路和架式这些表面的东西。

- 第二阶段

此阶段能精通基于某种平台的接口（例如我们现在常用的Win 32的API函数）以及所对应语言的自身的库函数。到达这个阶段后，也就相当于可以进行真实散打对练了，可以真正地在实践中做些应用。

- 第三阶段

此阶段能深入地了解某个平台系统的底层，已经具有了初级的内功的能力，也就是“手中有剑，心中无剑”。

- 第四阶级

此阶段能直接在平台上进行比较深层次的开发。基本上，能达到这个层次就可以说是进入了高层次。这时进入了高级内功的修炼。比如能进行VxD或操作系统的内核的修改。

这时已经不再有语言的束缚，语言只是一种工具，即使要用自己不会的语言进行开发，也只是简单地熟悉一下，就手到擒来，完全不像是第一阶段的时候学习语言的那种情况。一般来说，从第三阶段过渡到第四阶段是比较困难的。为什么会难呢？这就是因为很多人的思想转变不过来。

- 第五阶级

此阶段就已经不再局限于简单的技术上的问题了，而是能从全局上把握和设计一个比较大的系统体系结构，从内核到外层界面。可以说是“手中无剑，心中有剑”。到了这个阶段以后，能对市面上的任何软件进行剖析，并能按自己的要求进行设计，就算是MS Word这样的大型软件，只要有充足的时间，也一定会设计出来。

- 第六阶级

此阶段也是最高的境界，达到“无招胜有招”。这时候，任何问题就纯粹变成了一个思路的问题，不是用什么代码就能表示的。也就是“手中无剑，心中也无剑”。

此时，对于练功的人来说，他已不用再去学什么少林拳，只是在旁看一下少林拳的对战，就能把此拳拿来就用。这就是真正的大师级的人物。这时，Win 32或Linux在你眼里是没有什么差别的。

每一个阶段再向上发展时都要按一定的方法。第一、第二个阶段通过自学就可以完成，只要多用心去研究，耐心地去学习。

要想从第二个阶段过渡到第三个阶段，就要有一个好的学习环境。例如有一个高手带领或公司里有一个好的练手环境。经过二、三年的积累就能达到第三个阶段。但是，有些人到达第三个阶段后，常常就很难有境界上的突破了。他们这时会产生一种观念，认为软件无非如此，认为自己已无所不能。其实，这时如果遇到大的或难些的软件，他们往往还是无从下手。

现在我们国家大部分程序员都是在第二、三级之间。他们大多都是通过自学成才的，不过这样的程序员一般在软件公司也能独当一面，完成一些软件的模块。

但是，也还有一大堆处在第一阶段的程序员，他们一般就能玩玩VB，做程序时，去找一堆控件集成一个软件。

现在一种流行的说法是，中国软件人才现在是一个橄榄型的人才结构，有大量的中等水平的程序员，而初级和高级程序员比较少。而我认为，现在中国绝大多数都是初级的程序员，中级程序员很少，高级的就更少了。所以，现在的人才结构是“方塔”形，这是一种断层的不良结构。而真正成熟的软件人才结构应该是平滑的三角形结构。这样，初级、中级、高级程序员才能充分地各施所长。三种人才结构对比如图1.1所示。

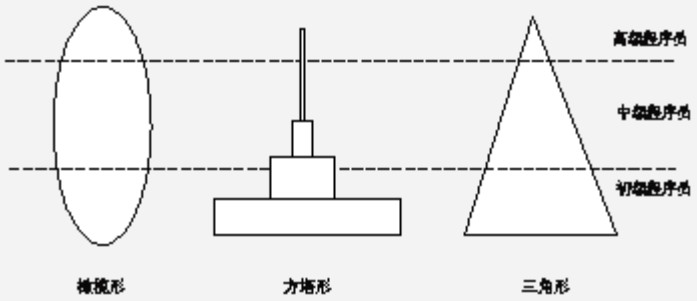


图1.1 三种人才结构对比

第1章 程序点滴

1.2 高手是怎样练成的(2)

1.2.2 初级程序员和高级程序员的区别

一般对于一个问题，初级程序员和高级程序员考虑这个问题的方法绝对是不同的。比如，在初级程序员阶段时，他会觉得VB也能做出应用来，且看起来也不错。

但到了中级程序员时，他可能就不会选择VB了，可能会用MFC，这时，也能做出效果不错的程序。

到高级程序员时，他绝对不是首先选择以上工具，VB也好，VC也好，这些都不是他考虑的问题。这时考虑的绝对是什么才是具有最快效率、最稳定性能的解决问题的方法。

> 软件和别的产品不同。比如，在软件中要达到某个目标，有*n*种方法，但是在*n*种方法中，只有一种方法或两种方法是最好的，其他的都很次。所以，要做一个好的系统，是很需要耐心的。如果没有耐心，就不会有细活，有细活的东西才是好东西。我觉得做软件是这样，做任何事情也是这样的，一定要投入。

程序员到达最高境界的时候，想的就是“我就是程序，程序就是我”。这时候我要做一个软件，不会有自己主观的思路，而是以机器的思路来考虑问题，也就是说，就是以程序的思考方式来思考程序，而不是以我去设计程序的方式去思考程序。这一点如果不到比较高的层次是不能明白的。

你设计程序不就是你思考问题，然后按自己的思路去做程序吗？

其实不是的。在我设计这个程序的时候，相当于我“钻”入这个程序里面去了。这时候没有我自己的任何思维，我的所有思维都是这个程序，它这步该怎么走，下步该怎么走，它可能会出现什么情况。我动这个部分的时候，别的部分是否要干扰，也许会动一发而牵全身，它们之间是怎么相互影响的？

也只有到达这个境界，你的程序才能真正地写好，绝对不是做个什么可视化。可视化本身就是“我去设计这个程序”，而真正的程序高手是“我就是程序”，这两种方法绝对是不同的。比如，我要用VB去设计一个程序，和我本身就是一个程序的思维方式，是不一样的。别人也许觉得操作系统很深奥，很复杂，其实，如果你到达高手状态，你就是操作系统，你就能做任何程序。

对待软件要有一个全面的分析方法，光说理论是没有用的。如果你没有经过第一、第二、第三、第四这四个阶段，则永远到达不了高境界。因为空中楼阁的理论没有用，而这些必须是一步一步地去做出来。

一个高级程序员应该具备开放性思维，从里到外的所有的知识都能了解。然后，看到世界最新技术就能马上掌握，马上了解。实际上，技术到达最高的境界后，是没有分别的。任何东西都是相通的，只要你到达这个境界以后，什么问题一看就能明白，一看就能抓住最核心的问题，最根本的根本，而不会被其他的枝叶或表象所迷惑，做到这一步后才算比较成功。

从程序员本身来说，如果它到达这一步以后，他就已经形成了开阔的思维。他有这种开放性思维的话，他就能做战略决策，这对他将来做任何事情都有好处。事实上，会做程序后，就会有一种分析问题的

方法，学会怎么样把问题的表象剖开，看到它的本质。这时你碰到任何具体的问题，只要给点时间，都能轻而易举地解决。实际上，对开发计算机软件来说，没有什么做不了的软件，所有的软件都能做，只是看你有没有时间，有没有耐心，有没有资金做支撑。

这几年，尤其是这两三年，估计到2005年前，中国软件这个行业里面大的软件公司就能形成。现在就已经在形成，例如用友，它上市后，地位就更加稳固了。其他大的软件企业会在这几年内迅速长大。这时候，包括流通渠道、经销商的渠道也会迅速长大。也就是说，到2005年以后，中国软件这个行业的门槛比现在还要高很多，与美国不会有太大的差别。此时，中国软件才真正体现出它的威力来。如果你是这些威力中的一员，就已经很厉害了。

别人可能知道比尔·盖茨是个谈判的高手，是卖东西的高手，其实，比尔·盖茨从根本上来说是个程序高手，这是他根本中的根本。他对所有的技术都非常敏感，一眼就看到本质，而且他本身也能做程序，时常在看程序。现在他不做董事长，而做首席设计师，这时他就更加接近程序的本质。因为他本身就有很开阔的思维，又深入到技术的本身，所以他就知道技术的方向。这对于一个公司，对他这样的人来说，是非常重要的。

如果他判断错误一步，那公司以后再回头就很难了。计算机的竞争是非常激烈的，不能走错半步。很多公司以前看上去很火，后来就
销声匿迹了，就是因为它走错一步，然后就不行了。为什么它会走错？因为他不了解技术的本质在哪里，技术的发展方向在哪里。

比尔·盖茨因为父母是学法律的，所以他本身就很能“侃”，很有说服力，而他又是做技术的，就非常清楚技术的方向在哪里，所以他才能把方向把握得很准确，公司越来越大。而别的公司只火一阵子，他却火了还会再火。就算微软再庞大，你如果不把握好软件技术的最前沿，一样也会玩完。就像Intel时刻把握着CPU的最新技术，才能保证自己是行业老大。技术决定它的将来。

所以，程序员要能达到这样的目标，就要有非常强的耐心和非常好的机遇才有可能。事实上，现在的机会挺好的，2005年以前机会都非常大，以后机会会比较小。但是，如果有耐心的话，你还是会有机会的，机会都是出在耐心里。我记得有句话说“雄心的一半是耐心”，我认为雄心的三分之二都是耐心。如果你越有野心，你就越要有耐心，你的野心才有可能实现。如果你有野心而没有耐心，那都是胡思乱想，别人一眼就能看穿。最后在竞争中，对手一眼就看到你的意图，那你还有什么可竞争的？

1.2.3 程序员是吃青春饭的吗

很多人都认为程序员是三十岁以前的职业，到了三十岁以后，就不应再做程序员了。现在的很多程序员也有这种想法，我觉得这种想法很不对。

在20世纪80年代末到90年代初，那时软件还没有形成行业，程序员不能以此作为谋生的手段时，你必须转行，因为你年轻的时候不用考虑吃饭的问题，天天“玩”都可以，但是以后就不可能了。

据我了解，微软里面的那些高手，几乎都是四五十岁的，而且都是做底层的。他们是上世纪70年代就开始“玩”程序的，所以对于整个计算机，他们是太清楚了。现在有些人主观臆断地希望微软第二天倒闭就好了，但那可能性太小了。因为那些程序员是从CPU是4004的时候开始，玩到现在奔腾IV，没有哪一代东西他们没有经历过。

你知道他们现在正在玩什么吗？现在正在玩64位的CPU。你说你普通的程序员，有这个耐心吗？没有这个耐心，你绝对做不了，你也绝对当不了高手。他为什么能做？因为他不仅是玩过来的，而且他还非常

有耐心，每一步技术他都跟得上，所以对他来说，没有任何的难度和压力。

第1章 程序点滴

1.2 高手是怎样练成的(3)

因为计算机技术没有任何时候是突变的。它的今年和去年相差不会很大，但是回过头来看三年以前的情况，和现在的距离就很大。所以说，如果你每年都跟着技术进步的话，你的压力就很小，因为你时刻都能掌握最新的技术。但是，如果你落下来，别说十年，就是三年，你就赶不上了。

如果你一旦赶不上，就会觉得非常吃力；如果你赶不上，你就会迷失方向；如果你迷失了方向，你就觉得计算机没有味道，越做越没劲。当你还只是有个思路的时候，别人的产品都做出来了，因为你的水平跟别人相差太远，人家早就想到的问题，你现在才开始认识。水平越高，他就看得越远，那么他的思维就越开阔；水平越低，想的问题就越窄。

> 64位CPU是这个十年和下个十年最重要的技术之一，谁抓住这个机会，谁就能抓住未来赚钱的商机。CPU是英特尔设计的，对这一点他肯定清楚。举例来说，如果从64位的角度来看现在的32位，就像从现在的角度去看DOS。你说DOS很复杂吗？当你在DOS年代的时候，你会觉得DOS很复杂。你说现在的Windows不够复杂吗？Windows太复杂了，但是你到了64位的时候再去看Windows，就如同现在看DOS一样。
>
> 整个64位系统的平台和思维方式、思路都比现在更开阔，打个比方说，现在的Windows里面能开*n*个DOS窗口，每个DOS窗都能运行一个程序。到达64位的时候，操作系统事实上能做到开*n*个X86，开*n*个Windows 98，然后再开*n*个Windows 95都没有问题，系统能做到这一步，甚至你的系统内开*n*个Windows NT都没有关系。这就是64位和32位的差别。所以，微软的那些“老头”，四、五十岁的那几个做核心的人，现在正在玩这些东西。你说微软的技术它能不先进吗？是Linux那几个玩家能搞定的吗？
>
> 微软的技术非常雄厚，世界计算机的最新技术绝对集中在这几个人手里。而且这几个人的思维模式非常开阔，谁都没有意识到的东西他早就开始做了。现在64位的CPU都出来一二年了，你说有什么人去做这些应用吗？没有，有的就是那几个UNIX厂商做好后给自己用的。

所以，追求技术的最高境界的时候，实际上是没有年龄限制的。对我来说，现在都三十三了，我从来没有想过退出这行，我觉得我就能玩下去，一直玩到退休都没有问题。我要时刻保持技术的最前端，这样的话对我来说是不困难的，没有任何累的感觉。

很多人说做程序不是人干的事情，是非人的待遇。这样，他们一旦成立一个公司，做出一点成绩，在辉煌的时候马上就考虑退出。因为他们太苦了，每天晚上熬夜，每天晚上烧了两包烟还不够，屋子里面简直就缺氧了，好像还没有解决问题。

白天睡觉，晚上干活，那当然累死了，这是自己折腾自己。所以，做程序员一定要有一种正常的心态，就是说，你做程序的时候，不要把自己的生活搞得颠三倒四的。如果非得搞得晚上烧好多烟才行，这样你肯定折腾不到三十岁，三十岁以后身体就差了。

事实上，我基本上就没有因为做程序而熬夜的。我只经历过三次熬夜，一次是在学校的时候，1986年刚接触计算机时，一天晚上跟一个同桌在计算机室内玩游戏，研究了半天，搞着搞着就到了天亮，这是第一次。然后在毕业之前，在286上做一个程序。还有一次就是超级解霸上市前，那时公司已吹得很大了，那天晚上没法睡觉。

一般来说，我也是十二点钟睡觉，第二天七点就起了。所以说，只有具有正常的生活、正常的节奏，才有正常的心态来做程序员，这样，你的思路才是正常的，只有正常的东西才能长久。搞疲劳战或者是黑白颠倒，时间长久后就玩不转了，玩着玩着就不想玩了。

只要你不想玩，不了解新技术，你就会落后，一旦落后，你再想追，就很难了。

第1章 程序点滴

1.3 正确的入门方法(1)

在这一节中，主要讲从我的经验来看，一般程序员需要注意的地方。教你怎样去具体学习不是我的责任，你可以去任何一个书店去找一本书回来自己看就可以了。这里只是对这些书做一些补充以及一些平常从来没注意的内容。

入门最基本的方法就是从C语言入手。如果以前学过BASIC语言的话，那么从C语言入手是非常容易的。我就经历了一个过程，根本不觉得这中间有太大的难度。其实，C语言本身和BASIC没有什么两样。BASIC每个所谓的命令在C语言里面都可以做成一个函数来实现，那么你就能用那个命令组合成整个程序。从这个角度来看，BASIC和C语言没有本质的差别。C语言就是入门的正确方法，没有其他。

现在的C语言本身就包含了嵌入汇编，使学习汇编语言的时候更加方便。你可以忽略掉纯汇编里面的很多操作。也许有人觉得这个方法太慢了。但要知道，工欲善其事，必先利其器，要想成功，没有一个艰苦的过程是不可能的，所以一开始的时候就要有耐心。如果你准备花5年的时间成为高手，那我敢说，你根本不用等到5年，你只要有这个耐心就足够了，你可能2年～3年内就能达到目标。但如果你想在一年时间内就成为高手，即使5年后，你还是成不了高手。

> 我们公司1998年招的开发人员都是应届大学毕业生。很明显，有人好像什么都会，又会CorelDraw，又会Photoshop，又会Flash，又会C＋＋，甚至VB也会。可是这样的人到现在还是全都会，但是什么事情也做不好，做的东西“臭”死了。但其中有一个人就不同，他以前甚至连Windows的程序都没有做过，只会在DOS下做几个小程序。但当我们把超级解霸的程序给他看，让他去研究的时候，他只用一周的时间，就迅速掌握。他那个月进步非常快，几乎就是一生中进步最快的阶段，这就是一个质的飞跃。
>
> 从基本入手以后，当你的积累到达一个阶段以后，就会有一个质的飞跃的阶段。事实上，我也有这么一个阶段，这个阶段也是我离开大学以后，真正去公司做事的时候。当我真正拥有一台计算机后，我把所有以前积累的问题在一个月内做了探讨以后，感觉自己的水平迅速提高。

入门和积累是很重要的。事实上，到达高手的境界以后，不管什么语言不语言的，其实都根本不用去学，只要拿过来看两天，就全部精通。如果你没有入门，即使去书店找n本书，天天背它，你也不会成为高手。

所有的语言只是很花哨的表面东西。高手马上就能透过它的表象而看到它的本质。这样才是真正的高手。他不需要再去学什么Java，或者其他什么语言。当他真正要写个Java程序的时候，只要把Java程序拿过来看一看，瞄一瞄书，就全都清楚了。如果这时他学VB就更容易了，我想他不用一天的时间，就能学会。到达高手的境界以后，所有的事物都是触类旁通的。

当你成为C语言的高手，那么就你很容易进入到操作系统的平台里面去；当你进入到操作系统的平台里去实际做程序时，就会懂得进行调试；当你懂得调试的时候，你就会发现能轻而易举地了解整个平台的

架构。这时候，计算机基本上一切都在你的掌握之中了，没有什么东西能逃得出你的手掌心。

上面只是针对程序的角度说明，另外一点也很重要，即好的程序员必须具备开放性思维，也就是思考问题的方法。程序员，尤其现在很多的程序员，都被误导从MFC入手，这就很容易形成一种封闭式的思维模式。这也是微软希望很多人只能学点表面的东西，不致成为高手，所以他大力推荐MFC之类的工具，但也真有很多人愿意去上他的当，最后真正迷失方向。说他做不了程序吧，他也能做程序，但是如果那个程序复杂一点，出现问题时，问题出在哪里就搞不清楚了，反正是不清楚。如果你真正有一种开放性的思维，在你能够成为高级程序员的时候，对MFC这些是不屑一顾的，MFC、VB根本不会在考虑的范围之内。

事实上很多人，包括外面很多公司里面工资挺高的人，可能一个月能拿五、六万的这些人，他们的思维也不一定能达到很高的境界。但是，他确实做了很多的事情，已经有很好的积累了。但要上升到更高的境界上，就要有正确的思维方法。这就是为什么比尔·盖茨说，他招人的时候宁愿招一个学物理，而不是学编程的。学物理的人会有非常非常广的思维，他考虑的小到粒子，大到宇宙，思维空间非常广阔，这样，他思考问题的时候，就会很有深度。

有人研究物理研究得比较深的时候，他能针对某个问题一直深入进去。很多写程序的人只会注意到这行代码或那行代码，则比较起来则显得肤浅。所以，编程的时候也要深入进去，把你的爱好、你的所有思维都放进去，努力做到物我合一的境界。

第1章 程序点滴

1.3 正确的入门方法(2)

1.3.1 规范的格式是入门的基础

以前所有的C语言的书中，不太重视格式的问题，写的程序像一堆堆的垃圾一样。这也导致了现在的很多程序员的程序中有很多是废码、垃圾代码，这和那些入门的书非常有关系。因为这些书从不强调代码规范，而真正的商业程序绝对是规范的。你写的程序和他写的程序应该格式大致相同，否则谁也看不懂。如果写出来的代码大家都看不懂，那绝对是垃圾。如果把那些垃圾“翻”半天，勉强才能把里面“金子”找出来，那这样的程序不如不要，还不如重新写过，这样，思路还会更清楚一点。这是入门首先要注意的事情，即规范的格式是入门的基础。

1. 成对编码

正确的程序设计思路是成对编码，先写上面的大括号，然后马上写下面的大括号。这样一个函数体就已经形成了。它没有任何问题。然后，比如你要写个for循环，这时候先申明一个变量I，再写这个for循环。写上面的大括号，马上写下面的大括号，然后再在中间插一二行代码。插这段代码后，如果你又要用到新变量，则再在头上添加新的变量，然后再让它进行工作。这就是一种成对编码。

这样，当你用到一个内存的时候，写一个分配函数分配一块内存，马上就接着写释放这块内存的代码。然后你再在中间插上你要用这个内存做什么。这是正确的快速的编程方法。否则，你去查或调试代码都无从下手。针对这个程序来说，如果用成对编码，则它任何时候都是可以调试的，不需要你整个程序都写完后才能进行调试。

它是任何时候都可以编译调试的，甚至你写了两个大括号，中间什么也没有，它是空的时，你都可以进行调试。你写了第一个for循环，它也可以进行调试，当你又写了一个分配内存、释放内存以后，它还可以进行调试。它可以编译运行，里面可以放断点，这就是成对编码。

成对编码就涉及到代码规范的问题。为什么我说上面一个大括号，下面一个大括号，而不说成是前面一个大括号，后面一个大括号呢？如果是一般C语言的书，则它绝对说是后面加个大括号，回过头前面加个大括号。事实上，这就是垃圾程序的写法。正确的思路是写完行给它回车，给它大括号独立的一行，下面大括号也是独立的一行，而且这两个大括号跟那个for单词中间错开一个TAB。

> 集成环境的TAB首先要设成8，因为TAB的基本定义就是8，而现在的VC把它设成了4，这样使得你编出的程序放到一个标准的环境里看的时候就是乱的。

代码一定不能乱，一定要格式非常清楚，这点使你写的程序我能读，我写的程序你也能读，不需要再去习惯彼此的不同写法。

而且结合成对编码思维，这时候你去读一个程序的时候，你会发现，你读程序的方法变了。以前读程序的时候，你可以先去读它的变量是什么，然后再读第一行、第二行，读到最后一个大括号，这是一种读程序的方法。现在就不一样了，现在读程序的时候就养成了一种习惯，就是分块阅读程序，很明显两个大括号之间就是一块代码。

那么写出一个程序后，你要读这个程序是干什么的，只要看这个大括号和那个大括号之间的部分就可以了，不需要再去读其他的代码是干什么的。比如，你从Linux中或网上下载了一个“烂”程序后，该怎么去阅读它？最好的方法是先把程序所有的格式都整理好，先别去读它。把所有的格式按照这种规范化的方法，把它的括号全部整理好。这时候你再读那个程序，只要半分钟就读懂了，但是你可能要整理一个小时。但如果不这样做，你可能读两个小时都读不清楚该程序。

这点绝对不会有人告诉你，现在没有人去讲解这方面的技巧。这也是我写了那么多的程序，才总结出来的。一开始的时候，我也像那些教科书所教导那样写，后面放个大括号，前面放个大括号，甚至括号连括号，一连四个括号，每个括号对哪个最后都找不清楚。编译告诉你好像少了一个括号，于是找呀，找呀，上面找，下面找，而这个程序又很大，只有一个函数，上面在上屏，下面在下屏，最后翻来翻去也翻不出。

所以我就想，大括号之间要互相对应，即使不在一个屏幕内，也能很容易地看到它，因为只要光标落在这个大括号里面，往上去找，即能找到它头上的那个与此对正的，而且这些代码是在一起的。这一层代码和下一层代码是互相隔开的，我只要读这层代码，下面那一层代码就不需要了。

比如，它有*n*个for循环的时候，我只想看某一个for循环，这时我只要对正大括号，它的光标往上走，一下就能找到了。如果按照教科书那样写的话，你要读呀，读呀，要把所有的代码，把所有的for
循环都读一遍，才可能找到你要的东西。这就是成对编码和规范化的方法（详细叙述请参考代码规范一章）。

代码中如果不包括正确的思路，那该代码就没有什么用。如果是一个代码爱好者去收集代码，而现在网络上代码成群，Linux本身就带了一大堆的程序，那些程序对你真的有用吗？我看不见得。而且那些程序还在不断地升级，那程序还会有新版，如果你把它拿来看一下，对你来说其实没什么价值。

那怎么样使得它对你有用？就必须用上面所说的方法，经过这么处理以后，你就能真正取到它其中的设计思路，这样才能变废为宝。如果是MFC之类的东西，那你就不用找了，因为即使找，也找不出有价值的东西，全部是VC自动给你生成的一堆堆的垃圾框架，相对于网上Linux程序来说，它可能更“臭”一些。

第1章 程序点滴

1.3 正确的入门方法(3)

在软件没有形成行业，程序等同于软件的时候，那时候程序很容易体现出价值来。只要得到代码，就相当于得到这个软件。但现在就不同了。现在的程序都不是几行，你写出的程序，如果又没有注释，格式又很乱，你拿过来给我，我还得花很长的时间才能读得清楚，那这样的程序的代码有价值吗？

> 我经常听到一些程序员在外面兜销代码，很多是学校的学生，尤其那些素质比较差的研究生，和老师做了一个项目后，他拿出来到外面到处去卖，但是他最后可能卖出去吗？最后可能还是没卖出去，因为那个程序很庞大。如果某个公司买了这个程序以后，该公司还得招一个人去读这个程序，当这个人读懂以后，他又离职了，那公司买这个代码干嘛？

2. 代码的注释

代码本身体现不出价值来，有价值的代码一定是不仅格式非常规范，而且还要有很详细的设计思路和注释，这个是很重要的。首先要养成这种习惯，教科书里面很少讲为什么要做注释，注释应该怎么注。有些人爱在哪儿下注释就在哪儿下注释，甚至在语句中间也加，中间也可弄两个斜杠放两个花括号写点注释。

注释格式是非常重要的，但很少有人去注意它。现在的程序如果没有注释，则基本上是没法用的，也就跟你拿一个可执行程序没什么两样，你拿过来还不能随便改，你改了后编出来的程序绝对不能用。所以，程序如果没有详细的注释，别人就算拿到了代码也没有用，体现不出它的价值来。

Linux是个操作系统，很厉害呀！其实那些程序你拿回来，耐心地去读它，会发现，它里面乱得很，那个内核程序除了作者自己能读懂外，别人可能要花很长的时间才能读懂。Apache的作者对自己Apache那套代码是很清楚，但换一个做浏览器的人去读，也会很困难。一般人只把代码复制下来后，打个BUILD命令看看能不能正确地编译，最后能正确编译的程序就是好的，如果不能正确编译的程序就删掉吧，再下载一个，因为他没有正确的对待代码的那种思维，而只是认为那代码本身才有很大的价值，不用关心有没有注释。

如果代码没有注释和规范，是没有价值的，这也是现在为什么很多的个人跑去卖源程序的时候，很多的公司都不要。我们不是说没有技术，任何程序都能做，只是时间的问题，而且像视频中有的技术，比那些卖代码的技术还要深得多。真正要做一个有价值的程序，开发程序的思维就很重要，这种思维的具体体现就在注释及规范的代码本身。

1.3.2 调试的重要性

调试是很重要的一个部分。所有的程序都是调试出来的，不是写出来的。讲怎么去调试，实际上就是讲一种解决问题的思路。所有的程序写出来后一定是有问题的，既然有问题，就一定会有一个解决问题的思路。解决问题的方法就是调试的方法。

用VB或者是MFC做出来的程序，先运行一遍看看什么地方有问题，如果发现有问题，重新改一改，然后又重新运行。这种方法是还没有入门的调试方法，即是看直接的表象。这种方法既浪费时间，又不能

消除隐患。

调试是很重要的内容，如果要进入高深境界，调试是除了了解设计程序、平台以外，一个非常重要的难关。如果要成为高级程序员，就必须过这一关。如果不懂调试，则永远成不了高手。在学习调试的过程中，对汇编语言、体系结构会有进一步的了解。

你可能觉得我把调试的作用说得言过其实了，举例子说明一下吧。请把以下的C程序改写成汇编代码:

```
int i;
extern int In[],Out[];
for(i=0;i<100;i++)
   {
   Out[i]*=In[i];
   }
```

我发现90%的人写出来的汇编代码可能是不正常的或有错误的。要么是不了解32位汇编，要么是不循环，要么只有循环没有处理等。这是为什么呢？因为就算是一段小小的代码，如果没有经过调试，也可能错误百出。

如果你是初级一点的程序员，则如果程序出了问题，也不知道原因所在。怎么回事呀？我就是搞不清楚。要搞清楚首先要调试，这就涉及到调试的问题。比如说，放到一个文件里面的，它出错了，我查程序看了*n*遍，它就是没有任何问题，这时候该怎么办呢？这时的解决方法就是调试，调试能使得一个程序正常地运转起来。如果对于程序员来说写这个程序可能只用了一天的时间，但是调试可能会花他二三天的时间。一个程序绝对是调试出来的，不是编出来的。如果说哪个系统是编出来的，那它肯定会有很多性能方面的问题，包括可能有不可预测的各种各样的问题。

程序出现问题的话，要能考虑到各种各样可能的情况，绝对没有任何臆测。比如，有可能完全是编译器的错误，也有可能因你程序里面增加了什么，而对程序产生干扰，甚至还有一种可能是你的指针基本就没有给它赋值，指向了别的地方，把别的东西破坏了。这些情况太多了。还有一种常见的错误，即MFC里面很常见的一种设计思维，就是任何一个东西，只管创建，不管释放、销毁。这种思路是现在很多程序员做的程序没用几下就会死机的原因。这绝对是错误的设计思路，而MFC让你这么做，就是让你永远成不了高手，你写的程序永远不可能稳定。

> MFC里面的所有的结构也好，变量也好，只需要你去分配一个，几乎就不需要你去释放它。这绝对是错误的，程序一定要成对编写。成对编码是快速编写程序的一种方法，而教科书里面讲的那些都是从头到尾去编。先把那个什么变量编写上，再写第一行，再写第二行，再写第三行，最后再写个大括号。这种方法绝对是错误的。对于现在的程序来说，它效率很慢，没法即时调试，因为只有最后把所有的程序做完以后，才能进行调试，所以在这中间出现错误的几率就积累得非常大了。

第1章 程序点滴

1.4 开放性思维(1)

要具备开放性思维，就必须了解包括从CPU的执行方法，到Windows平台的运转，到你的程序的调试，最后到你要实现的功能这一整套的内容，只有做到这样，才能真正提高。如果你的知识范围很窄，什么也不了解，纯粹只了解语言，那你的思维就会很狭隘，就会只想到这个语言有这个函数，那个语言没有那个函数，这个C++有这个类，那个语言没有这个类等。而真正要做一个系统，思维一定要是全面的，游离于平台之上的系统和实际的应用软件是不现实的。

这种所谓理想化，已经有很多人提出是不现实的。所以，任何一个软件一定都是跟一个平台相关联的，脱离平台之上的软件几乎都是不能用的。这就必须对平台的本身非常了解。如果你有平台这些方面的知识，这样在思考一个问题的时候，能马上想到操作系统能提供些什么功能，我再需要做些什么，然后就能达到这个目标。这就是一种开放的思维。

在开放的思维下，我要做这个程序的时候，就会考虑怎么把它拆成几个独立的、分开的模块，最简单的，怎么把这个模块尽量能单独调用，而不是我要做个很大的EXE程序。一个很普通的程序员，如果他能够考虑到将程序分成好几个动态库，那么它的思维就已经有点开放性了，就已经不是MFC那些思维方式了。思考问题的时候能把它拆开，就是说，任何一个问题，如果你能把它拆开来思考，这就是简单的开放性思维。

但光会拆还是不够的，尽管有很多人连拆都不会。很多教科书中的程序，要解决问题的时候，就一个main，以后就是一个非常长的函数。这个main函数把所有的事情都解决了。如果连函数都不会分的话，则就是典型的封闭式思维。

这样的人不是没有，我是碰见过的。一些毕业生做的程序就有这种情况。所有的问题都由一个函数来解决。他就不会把它拆成几个模块。我问他，把一件工作拆成几件模块不是更清晰吗？他说，拆出来后的模块执行会更慢些。这就是很明显的封闭式思维和非封闭式思维的区别。

你看MFC的思路，那就是一层套一层的，要把所有的类都实现了，然后继承。它从CWnd以后，把所有的东西都包括进去了，组成一个巨型的类。这个巨型的类连界面到实现统统包括在里面。这时你怎么拆？根本就没有拆的方法，这就是封闭式思维。

如果一个系统、一个程序不能拆的话，则它基本上是做不好的。因为任何一个程序，如果它本身的复杂度越大，它可能出错的几率就越大。比如最简单的，哪个函数越大，则该函数的出错几率就越大。但如果把该函数分成很多小的函数，每个小的函数的出错几率就会很小，那么组合起来的整个程序的出错几率就很小。这就是为什么要把它拆出来的原因。

你用C++来实现的方法也是一样的。你要把它拆成许多的接口，如果能做到这样，你就能把它独立起来，甚至你能把它用动态库的方法去实现。动态库是现在的程序非常重要的一块。

1.4.1 动态库的重要性

有了动态库，当你要改进某一项功能的时候，你可以不动任何其他的地方，只要改其中你拆出来的这

一块。这一块是一个动态库，然后把它改进，只需要把这个动态库调试好后，整个系统就可以进行升级。

但如果不是这样，你的整个程序是独立的文件，然后，另外的功能也是一个独立的文件，把这个程序编译成一个EXE，这就不是动态库的思想。按道理，我只改这个文件，其他系统也不需要进行调试。理论上看起来是一样的，而实际的结果往往就是因为你改动了这个文件，使得原来跑得很好的整个系统，现在不能跑了或者出现了很奇怪的现象。如何解释这个问题？事实上，这就涉及到编译器产生代码的方法，如果不了解这点的话，永远找不出问题来。

不存在没有BUG的编译器，包括VC，它也会产生编译上的问题。就算把这些问题都排除，你的软件也可能因为你加了某些功能，而影响了其他的文件，这个几率甚至非常大。这又得把你以前的测试工作重头再来一遍了。

动态库和EXE有什么不同呢？

动态库，包括它的代码和数据都是独立的，绝对不会跟其他的动态库串在一起。但是，如果你把所有功能放到一个EXE的工程里面，它的数据和代码就都是放到一起的，最后产生可执行程序的时候，就会互相干扰。而动态库就不会，这是由操作系统来保证的。从理论上看，动态库也是一个文件，我做这个工程的时候也是一个独立的文件，但它就会出现这样的问题。

1.4.2 程序设计流程

程序设计流程其实很简单。第一步就是要拆出模块，如果你有开放性思维,则任何软件都非常容易设计。怎么设计呢？首先，拿到问题的时候，一定要明确目标；然后，对操作系统所提供哪些功能，程序怎么跟操作系统接口考虑清楚；接着，就是“砍”，把它分开，要把它拆成一个个的独立的模块；最后，再进一步去实现，从小到大地进行设计。

首先“抓”马上能进行测试的简单的模块，就像刚才说的成对编码那样，写任何一个部分都要进行调试，每个部分最好能独立进行调试。这样，每个部分都是分开的时候，它都有一定的功能。当把所要做的功能都实现后，组合起来，再进行通调就可以了。

决定一个软件的成败还是得看该软件设计的思维是否正确。我们也试过，即使你把那些所谓的软件写得再明白也没有用，如果实现这个软件的思路不对，则下面的工作根本就没有必要。

做软件时，一定要把注释写进去。这样写成的软件如果要改版的话，就很容易，因为你的整个系统是开放性的，那么你要增强某些功能的时候，都是针对其中的某个小项做改进，只要改它就是了。如果那个功能是全新的，则它本身就是一个独立块，只要去做即可。

现在很多开发工具都提供了自动化设计的功能，在生成新的程序的时候，只要设置好一些条件，就能自动产生程序的框架，这是一种趋势吗？

其实，这种方法不太适用通用软件的开发，针对某个公司做个ERP系统，可能会管用，但是那些方法拿不到通用软件里面来。通用软件绝对是一行一行地编码产生出来的，而且每一行编码的结果要达到一种可预测性。

什么叫可预测性？就是你写程序的时候，如果发现某一种症状，马上就能想到该症状是由于哪个地方出了错，而不是别的地方，也就是从症状就能判断出是哪些代码产生了问题，这就是可预测性。

如果你用MFC来“玩”的话，即使它出错了，你也可能不知道错误在哪里，它的可预测性就很差。做软件时，如果它的可预测性越高，解决问题的方法就越快。如果某用户说我出现什么状况了，你马上就可以

断定错误，而不用去搜索源代码，就能想到程序可能是什么地方有问题，则这就是可预测性。

第1章 程序点滴

1.4 开放性思维(2)

1.4.3 保证程序可预测性

设计程序的时候，如何保证可预测性呢？答案就是我们上面所说的，所有的代码必须是经过测试的，必须是一步一步调试过的。只有经过你调试过的代码，你才能知道这个代码做某种运算的时候，它是怎样的执行方法。如果你不知道它的执行方法，你没进行过调试，则你就没有任何预测性。要达到可预测性，代码在汇编级是怎么执行的，你都得非常清楚。代码对哪个部分进行了什么操作，你都得知道。如果达不到这点，你的可预测性就很差。

比如，有些程序，你看它的C或者C++的源代码时，都看不出任何的问题。你看静态的程序时看不出任何问题，动态的程序调试你也看不出任何问题，这时，你必须把它的汇编打开，看一看它具体的操作，才能知道。所以说，开放性思维非常重要，你必须从最低层到最上层都要清楚。VC本身提供了一个汇编的调试环境，但是打开汇编后，如果你都看不懂，那你说怎么调呢？调什么？如果一个程序经过调试出来，则它会出错的地方你马上就会知道，只要看一些表现，就知道它有些什么问题。

比如说，我们做“大眼睛”的时候有个这样的现象。当要显示一个很大的图的时候，屏幕上只能显示其中的一小块，这样就可能需要拖动整个图像，但是拖的时候，如果在Windows 2000或Windows XP系统下就会发现，一旦我将图像拖到右下角时，图像就一下到左上角去了。该图像在右下角没有到底的时候还是显示正确的，但一旦到底，就把右下角转到左上角去了，如图1.2所示。

这是怎么回事？在Windows 98和Windows 95下，从来没有这个问题，而且如果图像不到右下角这一行，只差一点，它也不会出现这样的问题。为什么在Windows 98下没有这样的问题，在Windows 2000下会有呢？难道是我的程序有问题？

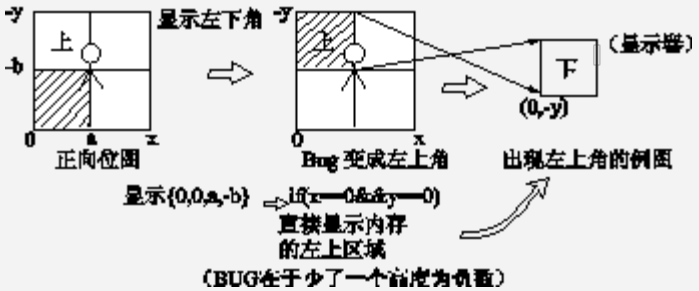


图1.2 图像显示问题示意图

这时，我就做了一个区域的比较，即看这个区域和整个这个图像的区域，是否中间运算有错误。但程序是调用Windows本身的API，我就怀疑是不是这个API出问题了。于是又重新写了一个区域相交部分，一步一步去查它，也没有任何问题，在任何情况下都是好的，但是到达右下角时，图像就会翻过来。经过以上两个步骤后，我就能确定，这是Windows操作系统的问题，Windows 98下没有这个问题，Windows 2000有，Windows XP也没有改过来。这是操作系统的原因，绝对不是软件的问题。

为什么会出现这样的问题？这是因为微软设计系统的那些家伙自以为聪明。只要图像的左上角是0，不管三七二十一，肯定往下面放，但是它的图像是正向位图，所有的位图设计的时候是倒过来的。而一个

正向位图的高度是负的，否则它显示的时候是倒过来的。高度是负的时候，这个0发生了变化，从上向下的，那么他设计操作系统的时候，只看了0而没去看高度，这时他没做条件处理。他的想法是为了加速这个位图的速度，是做优化的结果，但结果就出错了，而到现在他也没有解决这个问题。

所以，可预测性在这里就显得很重要了。当出现这个问题时，能想到要么就是区域合并有问题，要么就是直接显示的这个函数有问题。区域合并的问题可以解决，我写个函数还不行吗？我一步一步地去跟踪，就能肯定这个API有没有问题，最后得出结论是有问题，也的确是它有问题。如果你不会调试的话，这个问题你永远也查不出来；如果你不了解操作系统，你永远不会想到操作系统会出问题；如果你不了解这个平台，你根本就不知道问题所在。所以，要成为一个高手，视角一定要从里到外，从点到面非常开阔。如果你局限在一个封闭的思维里，做系统就很难。

第2章 认识CPU

2.1 8位微处理器回顾/2.2 16位微处理器(1)

2.1 8位微处理器回顾

在20世纪70年代中期，开始出现了8位芯片。8位芯片与以前的4位芯片相比，无论在指令还是译码数据，以及数据处理上都能按8位的方式进行处理，并且它提供了更多的寄存器和更快的寻址方式。

当时形成了以Intel的8080、摩托罗拉的MC6800（设计此芯片的人还设计出6502，后被苹果II采用）和Z80（此芯片在我国当初应用甚广）三足鼎立的局面。

以Intel 8080为例，它由6000多个晶体管构成，每秒能执行约60万次操作。寻址空间达到64KB，指令多达60条以上。

苹果Ⅱ使用的是6502芯片。6502的指令比较少，6502 CPU有256Byte的固定堆栈区，内有一些基本函数的功能。因为6502为8位，所以整个内存只有64KB。6502在苹果II及任天堂游戏机中被广泛地使用，可惜6502没有后续的兼容性的产品。

在没有IBM PC之前，个人电脑就是苹果。其中，苹果II是成功之作，而它没有使用Intel的8080及后来的8086。这令Intel这家CPU厂商倍受压力。为此，Intel加快了技术的研发，从8位机转向16位机；相反，6502的成功没有令它的厂商进一步开发16位的高性能的CPU。由此可见，机会永远是留给有心人的。

2.2 16位微处理器

为了保持在微处理器领域的领先地位，Intel在1978年推出了16位的8086芯片。但当时大部分计算机外部设备都是为8位微处理器而设计的，所以8086并没有引起大的反响。为此，Intel于1979年推出了准16位芯片8088，即它的内部总线为16位，而外部总线为8位。

> 当IBM进入PC市场时，8086/8088成为首选。尽管后来IBM要自己开发新CPU，并且想踢开Intel，但Intel 80286却助Compaq抓住了机会。Compaq迅速推出兼容机并大举成功（Compaq可能是Compatibility Quickly的缩写），IBM自己的CPU也就胎死腹中。
>
> 因为当时人们还没有对计算机产生“代”的概念。当时苹果机选用6502时，开发6502的那家CPU公司认为从此可以稳坐泰山了，就没有投入精力去开发新的或与这一代兼容的16位的下一代CPU。这时，Intel看到了机会，它迅速地研制出比苹果机要好得多的16位CPU 8086。这时，苹果机发现压力很大，所以也做了一个16位的也能兼容6502的CPU。但是，这个CPU比8086差些，所以苹果公司以后也就一直没有用生产6502 CPU的公司的CPU了，这个公司就失去了成为生产CPU的核心公司的一个机会。后来的苹果选用了68000。

2.2.1 组成结构

8086 CPU内部结构如图2.1所示。

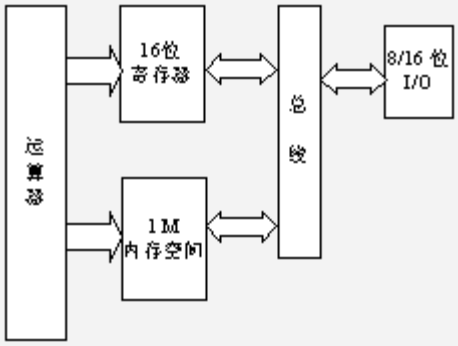


图2.1 计算机执行单元的主要结构

计算机主要是由总线、I/O、内存、寄存器、运算器这几个主要部件组成的。

8086/8088与6502之间最大的不同在于指令的体系结构。当我在使用6502的苹果II时，面临的最大难题是64KB的内存限制。同样的问题从8086（16位）到80386（32位）也出现了，在32位到64位时还将出现。

8086最头痛的问题在于段式结构，1MB的内存被它的段偏移所限制。至今我也不明白Intel当初为何要设计成这么复杂的内存机制，也许是为了与8080兼容的需要。这套笨拙的体系一直延续到IA64为止。

8086的内存机制使得段寄存器IP只要用16位就可以进行工作，否则，IP寄存器就要用20位来工作。

从软件的角度来看，执行指令如一个个小的函数一般，所以CPU中的指令可以通过软件的方法来模拟。也就是有这种思想，计算机界曾经出现过RISC（精简指令体系）和CISC（复杂指令体系）的争论。RISC就是在设计CPU时，只把最常用的指令用硬件来实现，其他的指令都通过微代码用软件的方法模拟实现。CISC是一种指令对应一组执行单元的体系结构。不过，随着CISC工作频率的提高和技术的发展，RISC现在已经黯然失色了。

8086在指令执行的时候引入了流水线的概念。例如，一个运算过程要分为6步来完成，当运算完成第一步后，CPU就会自动地进入第二步继续工作，当第三步完成后再运行第四步，这样一直下去，直到整个过程结束，这个计算过程就宣告完成。

但当CPU开始运行第一条指令的第一步时，第二条指令就可以进来了，这样就可以连续不断地运行。如果把每一步想像成CPU中的一个周期，那么相当于一个周期就运算完一条指令。如果增加流水线的数目，就可以相应地增加每个周期所完成的指令运算。

2.2.2 8086寄存器组成

8086/8088包括4个16位的数据寄存器，两个16位指针寄存器，两个16位变址寄存器，分成四组，它们的名称和分组情况如图2.2所示。

通用寄存器中，这些寄存器除完成规定的专门用途外，均可用于传送和暂存数据，可以保存算术逻辑运算的操作和运算结果。

第2章 认识CPU

2.2 16位微处理器(2)

段寄存器能在8086中实现1MB物理空间寻址，并可与8080 CPU进行兼容。段寄存器都是16位的，分别称为代码段（Code Segment）寄存器CS、数据段（Data Segment）寄存器DS、堆栈段（Stack Segment）寄存器SS和附加段寄存器。

图2.2 8086寄存器的组成

标志寄存器在8086中有一个16位用于反映处理器的状态和运算结果的某些特征。其中，包括9个标志位，如图2.3所示。

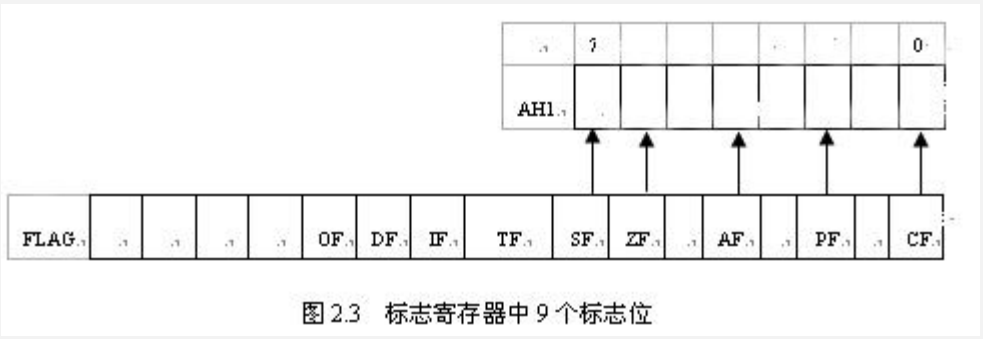


图 2.3 标志寄存器中 9 个标志位

这些标志位分为两类，其一是运算结果标志，主要用于反映处理器的状态和运算结果特征，有进位标志CF（Carry Flags）、零标志ZF（Zero Flag）、符号标志SF（Sign Flag）、溢出标志OF（Over Flag）、奇偶标志PF（Parity Flag）、辅助进位标志AF（Auxiliary Carry Flag）。

其二是状态控制标志。它控制着处理器的操作。要通过专门的指令才能使状态控制标志发生变化。其中有方向标志DF（Direction Flag）、中断允许标志IF（Interrupt Flag）、追踪标志TF（Trap Flag）。

2.2.3 内存的寻址

8086 CPU有20根地址线，可直接寻址的物理地址空间为1MB。系统内存由以字节为单位内存的存储单元组成，存储单元的物理地址长20位，范围是00000H至FFFFFH。尽管8086/8088内部的ALU每次最多进行16位运算，但存放存储单元地址偏移的指针寄存器都是16位的，所

以8080/ 8086通过内存分段和使用段寄存器的方法来有效地实现寻址1MB的空间。

逻辑段要求满足第一逻辑段的开始地址必须是16的整数倍，第二逻辑段最长不超过64KB的空间。段与段可以相互重叠和联接。

存储单元的逻辑地址由段值和偏移两部分组成，用如下的形式表示：

段值：偏移

所以根据逻辑地址可以方便地得到存储单元的物理地址，计算公式如下：

物理地址＝段值×16＋偏移

段值通过逻辑段的段寄存器的值来取得，偏移可由指令指针的IP、堆栈指针SP和其他可作为内存指针使用的寄存器（SI、DI、BX和BP）给出，偏移还可以直接用16位数给出。指令中不使用物理地址，而使用逻辑地址，由总线接口单元BIU按需要根据段值和偏移自动形成20位物理地址。物理地址的形成如图2.4所示。

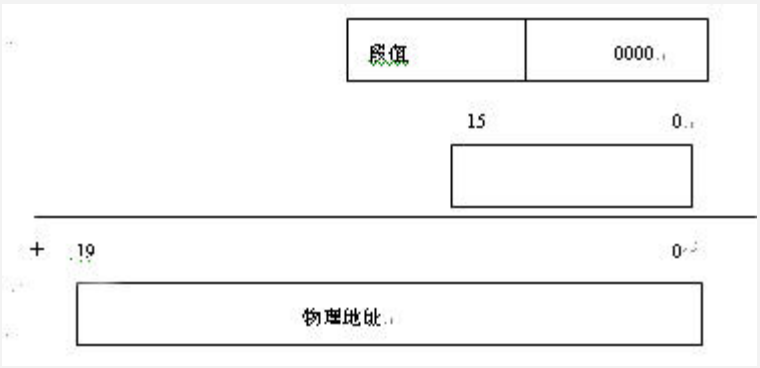


图2.4 物理地址的形成

2.2.4 中断处理

中断使CPU暂停正在运行的事件而转去处理另一事件。其实，中断还可以认为是一种函数的调用，不过，这个函数是随时都可能调用的，这样，中断就很好理解了。我们把引起这种操作的事件就叫中断源。它们可以是外设的输入输出请求，也可是计算机的一些异常事件或者其他的内部原因。

在8086/8088的计算机中，支持256种类型的中断，其中断编号依次为0~0FFH。

每种中断都有一个中断处理程序与之相对应。这些处理程序的段值和偏移量都被安排在内存的最顶端。因为它们占用1KB字节空间（256×4），所以当发生中断时，CPU根据中断向量表就可以很快地查找到对应的处理程序来处理中断事件。中断向量表如图2.5所示。

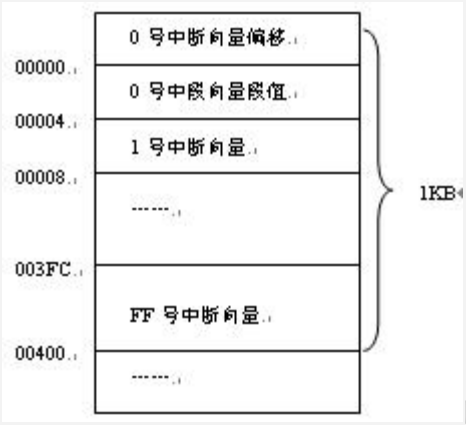


图2.5 中断向量表

我们从图中可以看到，所谓中断号其实就是中断处理的入口地址。

在IBM PC系列兼容计算机中，中断分为两种，一种是可屏蔽中断，另一种是不可屏蔽中断。DOS的部分中断分配情况如表2.1所示。

表2.1 DOS的部分中断分配表

向量号	功能	向量号	功能
0H	除法出错	10H	视频显示
01H	单步调试	11H	设备配置
02H	非屏蔽中断	12H	存储容量
03H	断点	13H	硬盘I/O
04H	溢出	14H	串行I/O
05H	打印屏幕	15H	扩充BIOS
06H	保留	16H	键盘输入
07H	保留	17H	打印输出
08H	定时器	18H	ROM BASIC
09H	键盘	19H	系统自举
0AH	保留（从中断控制器）	1AH	时钟管理
0BH	串行通信端口2	1BH	Ctrl+Break键处理
0CH	串行通信端口1	1CH	定时处理
0DH	硬盘（并行口）	1DH—1FH	参数指针
0EH	软盘	20H～2FH	DOS使用
0FH	打印机	30H～3FH	为DOS保留

第2章 认识CPU

2.3 32位微处理器(1)

按Intel的定义，0～32个中断是CPU出错用的，称为异常。32～255是给系统自己定义使用的。在DOS中，系统使用被分成了两个部分，一个部分是硬件的IRQ，IRQ就是级连的中断控制器。其他的则被分配给软件使用。现在64位的CPU中，中断扩充成16位，则理论上可有64KB个中断。

80286芯片能在实模式和保护模式两种方式下工作。在实模式下，80286与8086芯片一样，与操作系统DOS和绝大部分硬件系统兼容；在保护模式下，每个同时运行的程序都在分开的空间内独自运行。286的保护模式还是有很多不兼容缺陷，到了386才算有真正的改革，操作系统才真正进一步发挥作用，从16位真正跨入32位程序。

2.3 32位微处理器

1985年，真正的32位微处理器80386DX诞生，为32位软件的开发提供了广阔的舞台。1989年，Intel推出80486芯片，把387的浮点运算器合于486之中，并且采用流水线技术，令CPU每个周期可以执行一条指令，速度上突破100 MHz，超过了RISC的CPU。1992年，Intel发布奔腾芯片，采用多流水线技术及并行执行的能力，从此，CPU可以每个周期执行多个指令。1995年的奔腾Pro能力上再进了一步，产生动态执行技术，使CPU可以乱序执行。我们知道，从80386开始到现在的P4的CPU，它们的体系结构一直都是相同的，增加的只是内部的实现方式，所以，这些体系结构对大多数程序员来说就是透明的。

2.3.1 寄存器组成

80386寄存器的宽度大多是32位，可分为如下几组：通用寄存器、段寄存器、指令指针及标志寄存器、系统地址寄存器、调试寄存器、控制寄存器和测试寄存器。应用程序主要使用前面三组寄存器，只有系统才会使用其他寄存器。这些寄存器是8080、8086、80286寄存器的超集，所以，80386包含了先前处理器的全部16位寄存器。80386的部分寄存器如图2.6所示。

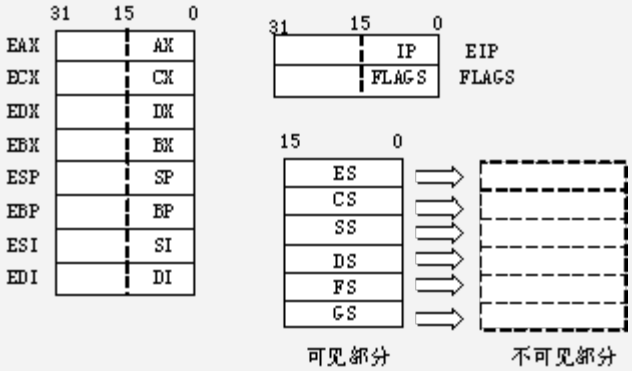


图2.6 80386的部分寄存器

1. 通用寄存器

80386有8个通用寄存器，这8个寄存器分别定名为EAX、EBX、ECX、EDX、ESP、EBP、ESI和EDI。它们都由原先的16位寄存器扩展而成。这些通用寄存器的低16位还是可以作为16位寄存器存取，并不受影响。以前的AX、BX、CX、DX这4个寄存器还可以单独使用这16位中的高8位和低8位，即分别是AH、AL、BH、BL、CH、CL、DH和DL。

在80386中，8个32位通用寄存器都可以作为指针寄存器使用，所以32位通用寄存器更加通用。

2. 段寄存器

80386中有6个16位的段寄存器，分别命名为CS、SS、DS、ES、FS和GS。其中，FS和GS是80386新增加的寄存器。

在实模式下，内存的逻辑地址仍是“段值：偏移”形式，而在保护模式下，情况就复杂很多了。它总体上是通过可见部分寄存器指向不可见的内存部分。有关内容将在2.3.2节中介绍。

所有这些寄存器的可见的部分和不可见的部分在IA64中可以直接处理IA 32位的一切，就像80386中的VM86一样，即如在Windows上执行DOS窗一样。

3. 指令指针和标志寄存器

80386的指令指针寄存器扩展到了32位，记为EIP。EIP的低16位是16位的指令指针IP，与以前的X86系统相同。

由于在实模式下，段的最大范围是64KB，所以EIP的高16位必须全是0，仍相当于16位的IP作用。

80386中，标志寄存器也扩展到了32位，记为EFLAG，如图2.7所示。

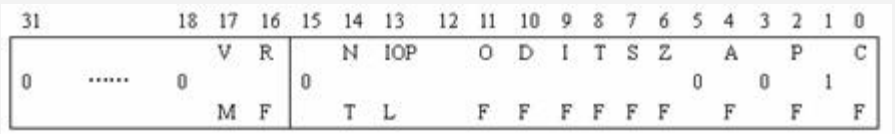


图2.7 80386的标志寄存器

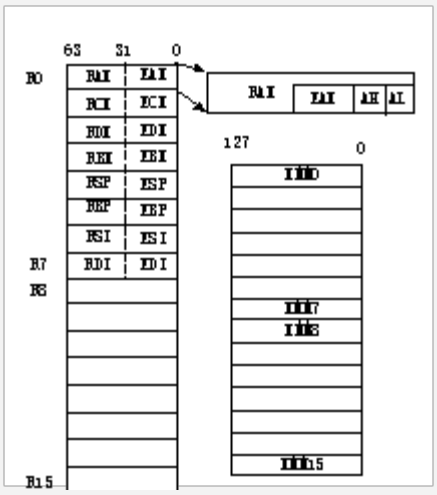
其中，增加了IO特权标志IOPL（I/O Privilege Level）、嵌套任务标志NT（Nest Task）、重启动标志RF（Reset Flag）、虚拟8086方式标志VM（Virtual 8086 Mode）。

AMD采用了X86架构并将之扩展至64位，开创了X86-64架构。

（1）处理器在32位的X86位纯模式下工作，可以运行现在的32位操作系统和应用软件。

（2）处理器在“长模式”下工作，运行64位的操作系统，既能执行32位应用程序，又能执行64位应用程序。

（3）只有在“64位模式”下，才能进行64位寻址和访问64位　寄存器。



（4）扩展是简单并且兼容的，所以处理器可以以最高的速度和性能支持X86和X86-64。

所有的用户都能获得32位的性能和32位的兼容性。在需要时，客户可以在不放弃32位兼容性的情况下迁移至64位的寻址和数据类型，沿用主流PC架构的发展而不是重新创作。AMD-64寄存器如上图所示。

第2章 认识CPU

2.3 32位微处理器(2)

2.3.2 保护模式

80386提供了两种工作模式。其一为实模式，在此模式下，80386可以和8086、8088完全地兼容。其二为保护模式，它是80386提供的一种全新的强的工作模式。在保护模式下，不仅可寻址4GB的内存空间，扩充了内存的分段管理机制，并可对内存进行分页管理，而且还可实现虚拟内存，支持多任务。

保护模式最重要的是完善了多任务保护机制。其实在80286开始，就具备了保护工作方式，但当时还不是很完善，80386才得到真正的完善。有两种保护模式任务方式。

（1）不同任务之间的保护：通过把每个不同的任务放在不同的虚拟地址空间中，来实现不同任务间的隔离（即A程序不能访问和修改B程序的代码和数据），以达到程序间的隔离。

（2）同一任务的保护：在每一任务之内定义了4种保护级别。分别为0、1、2、3，按环的方式来表示，如图2.8所示。

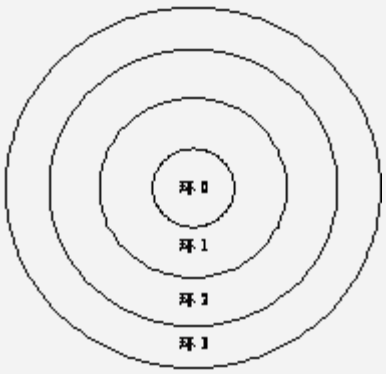


图2.8 同一任务保护模式

其中，0级代表最高的权限级，3级代表最低的权限级。按环的方式来表示，数字小的在“内环”，数字大的在外环。其中，环0、1、2为系统级，环3为用户级。原来的系统都是基于用户和系统来设计的，所以一般的系统只使用环0和环3这两个级。

2.3.3 80386的寻址方式

80386继续采用分段的方法管理主内存。内存的逻辑地址由段基地址（段的起始地址）和段内偏移两部分表示，存储单元的地址由段基地址加上段偏移得到。段寄存器指示段基地址，各种寻址方式决定段内偏移。

实模式下，段基地址仍然是16的倍数，段的最大长度仍然是64KB。段寄存器内所含的仍然是段基地址对应的段值，存储单元的物理地址仍然是段寄存器内的段值乘上16再加上偏移。所以，尽管386有32根地址线，可直接寻址物理地址空间达到4GB字节，但在实模式下，仍然与8086/8088相似。

在保护模式下，段基地址可长32位，并且无需是16的倍数，可以是内存内任意一个开始点，段的最大长度可达4GB。它的寻址就与8086/8088有很大的变化，如图2.9所示。

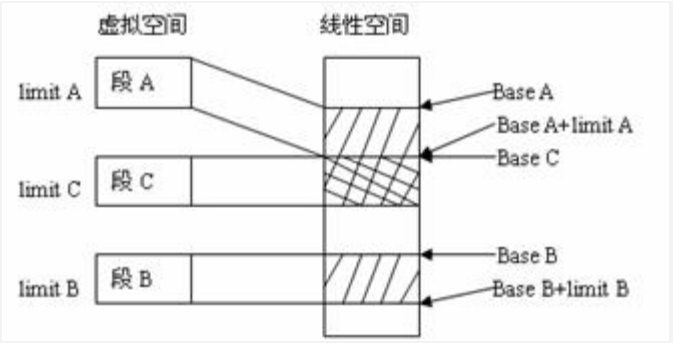


图2.9 保护模式下的寻址

1. 描述符

保护模式下的虚拟器由大小可变的存储块组成，这样的存储块还是称“段”。每个段由如下的三个参数进行定义：基地址、段界限、段属性。在保护下可以建立多个段。

而描述段的属性参数就称为“描述符”。它的格式如图2.10所示。

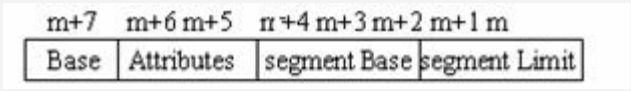


图2.10 描述符的格式

这些描述符会放置在内存的某一块空间内。

2. 选择子

在8086/8088和80386实模式下，段寄存器用来表示段值。而在80386的保护模式下，段寄存器就成为选择子。可以将选择子看做一个句柄。

选择子的作用就是指向对应的描述符。例如，代码选择子的值是02H（也就是CS＝02H），那么它指向的就是02H个描述符。

3. 简单的寻址过程

80386的寻址过程如图2.11所示。

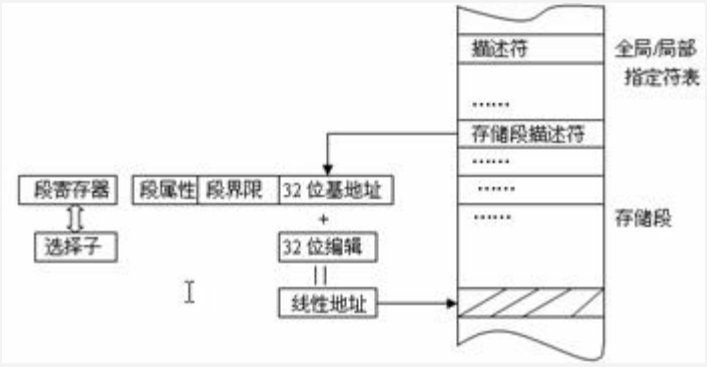


图2.11 80386的寻址过程

当在机器运行如下代码时：

MOV AX，DS：[DX];

假设此时DS＝04H，DX＝2344H，那么CPU怎样才能在内存中找DS：［DS］的值呢？其步骤如下：

（1）从DS选择子中选取04H。

（2）从对应的描述符空间中查找到第04H个描述符。

（3）取出描述符中的三个参数，分别是段基地址、段界限和段属性。假设段的基地址等

于00012345H，段界限等于5678H。

（4）这时，段基地址就是段的开始位置，通过EIP的32位偏移，就可得到物理地址，由：

物理地址＝段基地址＋偏移

可得物理地址就是179BDH（00012345H＋5678H）。

（5）此时就可以从179BDH中取出数据放入AX寄存器中。

这个寻址过程是经过简化后的模型，真实的寻址要比这复杂得多，有兴趣的读者可参考其他的书籍。

第2章 认识CPU

2.3 32位微处理器(3)

4. 中断处理

80386不但保存了8086/8088的所有中断，还增强了很多功能。我们把外部中断称为“中断”，把内部中断称为“异常”。

在实模式下，中断的处理和8086/8088完全一样。但是，在保护模式下，80386不再使用简单的中断向量表来处理中断程序，而是引入了“中断描述符”。中断描述符的结构如图2.12所示。

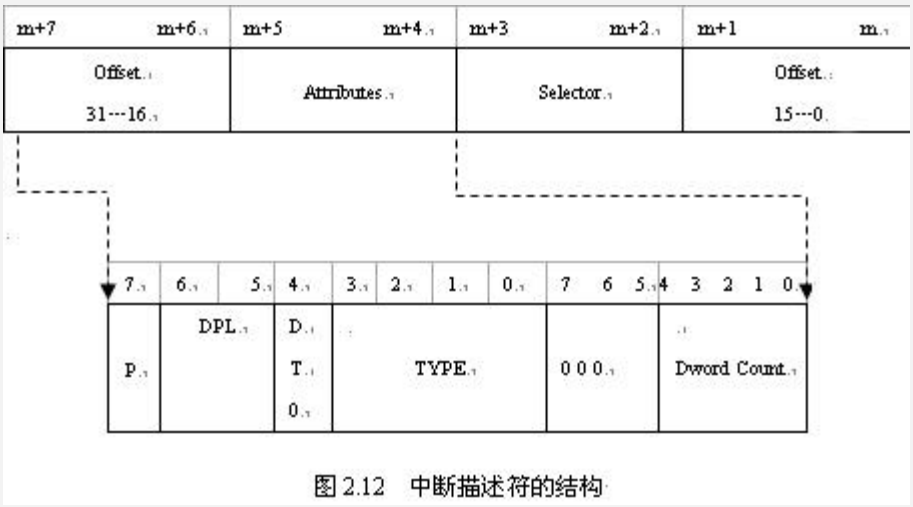


图 2.12 中断描述符的结构

```
GATE        STRUC ;门的数据结构
OFFSETL     DW 0   ;32位偏移的低a16位
SELECTOR        DW  0   ;选择子
DCOUNT      DW  0   ;双字计数字段
GTYPE       DB  0   ;类型
OFFSETH     DW  0   ;32位偏移的高16位
GETE    ENDS
```

中断的简单处理过程如下:

（1）当中断产生时，通过中断号找到对应的中断描述表。

（2）从中断描述表中取出对应的选择子和偏移。

（3）通过选择子从描述符中取出段的基值加上偏移，形成中断处理程序的位置。

（4）转入中断处理程序。

（5）中断处理程序分为以下两种。

- 当程序出现中断时，让中断自己进行处理，程序跳到中断点后继续运行。
- 中断程序可能先在环1进行一些处理，然后再跳环2进行一些处理，还可能跳用户层（环3）进行处理。但是Windows中是没有环1、环2的过程的，所以这种情况一般发生在异常中。这时就会

变成先在系统级进行处理，当处理完后，再返回到用户级继续处理，当用户级完成后，再返回到中断点。

中断处理过程简图如图2.13所示。

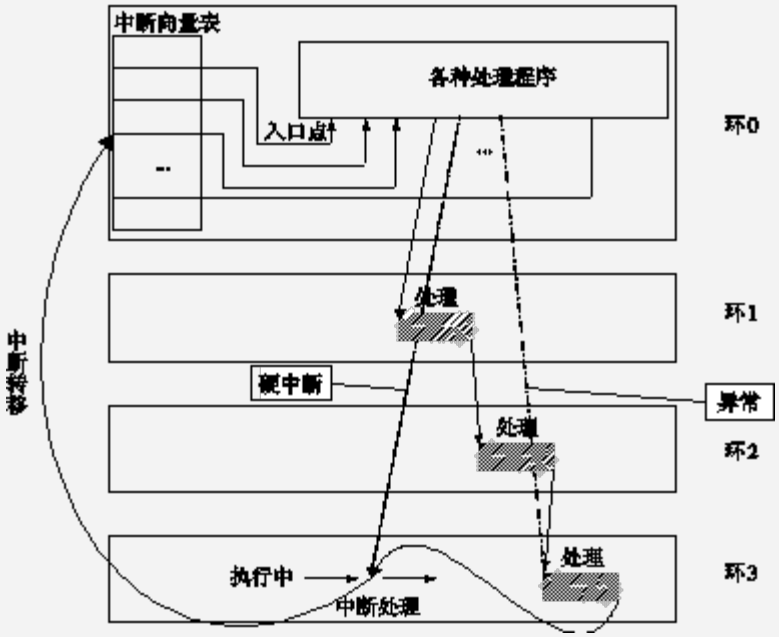


图2.13 中断处理过程简图

第2章 认识CPU

2.4 【实例】：在DOS实模式下读取4GB内存(1)

为了帮助读者实际了解以上所介绍的一些概念，下面我们来分析一段在DOS实模式下直接读取4GB内存的代码。通过该程序来分析CPU 的工作原理，揭开保护模式的神秘面纱，读者将会发现，保护模式其实与实模式一样简单和易于控制。在此基础上用四五十行C 语言程序做到进出保护模式和在实模式之下直接访问整个4GB内存空间。

这个访问4GB内存的程序是在实模式下使用的，它只是让CPU中的不可见部分有4GB大小访问权限。

在进入保护模式（CR0成为1）后，如果段寄存器不发生变化的话，则一切和实模式一样。所以CPU的保护位为1时，后面的代码依然可以执行，而不是死机状态。

同样的方法就不能用于分页，如果分页后的内存与不分页前时对于执行的地方发生不同，如分页的指令在内存0X12345处，分页后这个地方可能变成不存在，则计算机就只有出错重启。对于这个问题，本人做过多次实验，屡试不爽。

2.4.1 程序的意义

此程序具有如下功能：

- 不需要在保护模式状态下就可以直接把386的4GB内存读出来；
- 利用此程序可直接在DOS中做物理设备的检测；
- 理解GDT表的对应关系后，所谓386 32位模式也就很容易理解；
- 在DOS下，可根据此类方法将中断向量表移到任意位置，达到反跟踪或其他等目的。

2.4.2 程序代码

程序代码如下所示。

```
#include <dos.h>
//////////////////////////////////////////////////////////////
//          4G Memory Access
//  This Program Can Access 4G Bytes in DOS Real
//Mode,Needn't in Protection Mode It Works.
//  The Program Enter 32 Bit Flat Mode a moment and
//Only Load FS a 32 Bit Flat Mode Selector,Then Return
//Real Mode.
//  Used The FS Can Access All 4G Memory till It be
//reloaded.
//
//////////////////////////////////////////////////////////////

unsigned long   GDT_Table[]=
```

```
{   0,  0,                          //NULL   - 00H
    0x0000FFFF, 0x00CF9A00,     //Code32 - 08H Base=0
                            //Limit=4G-1 Size=4G
     0x0000FFFF, 0x00CF9200      //Data32 - 10H Base=0
                            //Limit=4G-1 Size=4G
};

//Save The IDTR before Enter Protect Mode.
unsigned char   OldIDT[6]={0};

//NULL The IDTR,IDTR's Limit=0 will disable all
//Interrupts,include NMI.
unsigned char   pdescr_tmp[6]={0};

#define KeyWait()   {while(inportb(0x64)&2);}
void        A20Enable(void)
{
        KeyWait();
        outportb(0x64,0xD1);
        KeyWait();
        outportb(0x60,0xDF);    //Enable A20 with 8042.
        KeyWait();
        outportb(0x64,0xFF);
        KeyWait();
}



void        LoadFSLimit4G(void)
{
    A20Enable();                //Enable A20

    //*************************************
     //*       Disable ints & Null IDT      *
     //*************************************
     asm {
             CLI                //Disable inerrupts
             SIDT    OldIDT      //Save OLD IDTR
             LIDT    pdescr_tmp  //Set up empty IDT.Disable any
                         //interrupts,
```

```
    }                   //Include NMI.
//*************************************
//*     Load GDTR       *
//*************************************
asm {

      //The right Code is Real,But BC++'s Linker NOT Work
   //with 32-bits Code.
    db  0x66        //32 bit Operation Prefix in 16 Bit DOS.
    MOV CX,DS   //MOV   ECX,DS
    db  0x66        //Get Data segment physical Address
    SHL CX,4        //SHL   ECX,4
    MOV word ptr pdescr_tmp[0],(3*8-1)
    //MOV   word ptr pdescr_tmp[0],(3*8-1)
    db  0x66
    XOR AX,AX   //XOR   EAX,EAX
    MOV AX,offset GDT_Table
   //MOV   AX,offset GDT_Table
    db  0x66
    ADD AX,CX   //ADD   EAX,ECX
    MOV word ptr pdescr_tmp[2],AX
                //GDTR Base high16 bits
    db  0x66
    SHR AX,16   //SHR   EAX,16
    MOV word ptr pdescr_tmp[4],AX
                //GDTR Base high16 bits
    LGDT    pdescr_tmp  //Load GDTR
      }
//*************************************
//*  Enter 32 bit Flat Protected Mode  *
//*************************************
//  Set CR0 Bit-0 to 1 Enter 32 Bit Protection
//Mode,And NOT Clear machine perform cache,It Meaning
//the after Code HAD Ready To RUN in 32 Bit Flat Mode,
//Then Load Flat Selector to FS and Description into it's
//Shadow register,After that,ShutDown Protection Mode
//And ReEnter Real Mode immediately.
//  The FS holds Base=0 Size=4G Description and
//it can Work in Real Mode as same as Pretect Mode,
```

```
    //untill FS be reloaded.
    //  In that time All the other Segment Registers are
    //Not Changed,except FS.(They are ERROR Value holded in CPU).
    asm {
        MOV DX,0x10                 //The Data32 Selector
        db  0x66,0x0F,0x20,0xC0     //MOV   EAX,CR0
        db  0x66
        MOV BX,AX                   //MOV   EBX,EAX
        OR  AX,1
        db  0x66,0x0F,0x22,0xC0     //MOV   CR0,EAX
       //Set Protection enable bit
       JMP Flush
        }               //Clear machine perform cache.
    Flush:              //Now In Flat Mode,But The
   //CS is Real Mode Value.
    asm {                    //And it's attrib is 16-Bit Code
                   //Segment.
        db  0x66
        MOV AX,BX           //MOV   EAX,EBX
        db  0x8E,0xE2           //MOV   FS,DX   //Load FS now
        db  0x66,0x0F,0x22,0xC0
                        //MOV   CR0,EAX
        //Return Real Mode.Now FS's Base=0 Size=4G
        LIDT        OldIDT
        //LIDT  OldIDT Restore IDTR
        STI                 //STI       Enable INTR
        }
}
//With FS can Access All 4G Memory Now.But if FS be reloaded
//in Real Mode It's Limit will Be Set to FFFFh(Size=64K),
//then Can not used it
// to Access 4G bytes Memory Again,Because FS is Segment:Offset
//Memory type after that.
//If Use it to Access large than 64K will generate Execption 0D.
//unsigned char ReadByte(unsigned long Address)
{
    asm db  0x66
    asm mov di,word ptr Address //MOV   EDI,Address
    asm db  0x67                    //32 bit Address Prefix
```

```
    asm db  0x64                    //FS:
    asm mov al,byte ptr [BX]        //=MOV AL,FS:[EDI]
    return  _AL;
}
unsigned char   WriteByte(unsigned long Address)
{
    asm db  0x66
    asm mov di,word ptr Address //MOV   EDI,Address
    asm db  0x67                    //32 bit Address Prefix
    asm db  0x64                    //FS:
    asm mov byte ptr [BX],al        //=MOV FS:[EDI],AL
    return  _AL;
}
///////////////// Don't Touch Above Code ////////////
#include <stdio.h>
//////////////////////////////////////////////////////////
//打印出Address指向的内存中的数据
//////////////////////////////////////////////////////////
void    Dump4G(unsigned long Address)
{
        int i;
        int j;
        for(i=0;i<20;i++)
          {
            printf("%08lX: ",(Address+i*16));
            for(j=0;j<16;j++)
                printf("%02X ",ReadByte(Address+i*16+j));
            printf("    ");
            for(j=0;j<16;j++)
              {
                if(ReadByte(Address+i*16+j)<0x20) printf(".");
                else printf("%c",ReadByte(Address+i*16+j));
                }
            printf("\n");
            }
}
main()
{
        char        KeyBuffer[256];
```

```
        unsigned long   Address=0;
        unsigned long   tmp;

        LoadFSLimit4G();
        printf("====Designed By Southern.1995.7.17====\n");
        printf("Now you can Access The Machine All 4G Memory.\n");
        printf("Input the Start Memory Physical to DUMP.\n");
        printf("Press D to Cuntinue DUMP,0 to End & Quit.\n");
        do  {
            printf("-");
            gets(KeyBuffer);
            sscanf(KeyBuffer,"%lX",&tmp);
            if(KeyBuffer[0]=='q') break;
            if(KeyBuffer[0]=='d') Address+=(20*16);
            else Address=tmp;
            Dump4G(Address);
            }while(Address!=0);
        return 0;
}
```

程序运行后，等用户从键盘输入一个字符。当输入“Q”字符时，整个程序将退出，当输入“D”时，将在屏幕上显示一屏内存的数据，最左边为绝对地址，其后一列显示的是以十六进制位表示的内存的数据，后一列是数据所对应的ASCII码。

第2章 认识CPU

2.4 【实例】：在DOS实模式下读取4GB内存(2)

2.4.3 程序原理

我们知道，CPU上电后，从ROM 中的BIOS开始运行，而Intel 文档却说80x86 CPU上电总是从最高内存下16字节开始执行，那么，BIOS是处在内存的最顶端64KB（FFFF0000H），还是1MB之下的64KB（F0000H）处呢？事实上，BIOS在这两个地方都同时出现（可用后面存取4GB 内存的程序验证）。

为了弄清楚以上问题，首先要了解CPU 是如何处理物理地址的。真的是在实模式下用段寄存器左移4位与偏移量相加，还是在保护模式下用段描述符中的基地址加偏移量，难道两者是毫无关联的吗？

答案是两者其实是一样的。当Intel把80286推出时，其地址空间变成了24位，则从8086的20位到24位，十分自然地要加大段寄存器才行。实际上，段寄存器和指针都被加大了，只是由于保护的原因，加大的部分没有被程序看见，到了80386之后，地址又从24位加大到32位（80386 SX是24位）。

在8086中，CPU只有“看得见部分”，但在80286之后，在“看不见部分”中已经包含了地址值，“看得见部分”就退化为只是一个标号，再也不用参与地址形成运算了。地址的形成总是从“不可看见部分”取出基址值与偏移相加形成地址。也就是说，在实模式下，当一个段寄存器被装入一个值时，“看不见部分”的界限被设成FFFFH，基址部分将装入值左移4位，属性部分设成16位0特权级。这个过程与保护模式时装入一个段寄存器是同理的，只是保护模式的“不可见部分”是从描述表中取值，而实模式是一套固定的过程。

对于CPU在形成地址时，是没有实模式与保护模式之分的，它只管用基址（“不可见部分”）去加上偏移量。实模式与保护模式的差别实际上只是保护处理部件是否工作得更精确而已，比如不允许代码段的写入。实模式下的段寄存装入有固定的形成办法，从而也就不需要保护模式的“描述符”了，因此，保持了与8086/8088的兼容性。而“描述符”也只是为了装入段寄存器的“不可见部分”而设的。

从上面的“整个段寄存器”可见，CPU的地址形成与“看得见部分”的当前值毫无关系。这也解释了为什么在刚进入保护模式时，后面的代码依然被正确地运行，而这时代码段寄存器CS的值却还是进入保护模式前的实模式值，或者从保护模式回到实模式时，代码段CS被改变之前程序是正常地工作，而不会“突变”到CS左移4位的地址上去。比如在保护模式时，CS是08H的选择子，到了实模式时，CS还是08H，但地址不会突然变成80H加上偏移量。因为地址的形成不理会段寄存器“看得见部分”的当前值，这一个值只是在被装入时对CPU有用。

地址的形成与CPU的工作模式无关，也就是说，实模式与0特权级保护模式不分页时是一模一样的。明白了这一机理后，在实模式下一样可以处理通常被认为只有在保护模式才能做的事，比如访问整个机器的内存。不必理会保护模式下的众多术语或许会更易于理解，如选择子就是“看得见部分”，描述符是为了装入“不可见部分”而设的。

有一些书籍也介绍有同样功能的汇编程序，但它们都错误地认为是利用80386芯片的设计疏漏。实际上，Intel本身就在使用这种办法，使得CPU上电时能从FFFFFFF0H处开始第一条指令，这种技术

在　之后的每一台机器每一次冷启动时都使用，只是我们不知道罢了。

2.4.4 程序中的一些解释

下面对程序做几点说明。

（1）IP=0000FFF0H

通过这样设置，CS：EIP就形成了FFFFFFF0H的物理地址，当CPU进行一次远跳转重新装入CS时，基址就变了。

（2）为了访问4GB内存空间，必须有一个段寄存器的“不可见部分”的界限为4G-1，基址为0，这样就包含了4GB内存，不必理会“可见部分”的值。显然要让段寄存器在实模式下直接装入这些值是不可能的。惟一的办法是让CPU进入一会儿保护模式，在装入了段寄存器之后马上回到实模式。

进入保护模式十分简单，只要建好GDT，把CR0寄存器的位0置上1，CPU就在保护模式了。从前面分析CPU地址形成机理可知，这时不必理会寄存器的“看得见部分”值是否合法，各种段寄存器是一样可用的，就像没进保护模式一样。在把一个包含有4GB地址空间的值装入某个段寄存器之后，就可返回实模式。

（3）预先可建好GDT如下：

```
unsigned long GDT-Table[]=
        {   0,0,                          //空描述符，必须为零
           0x0000FFFF,0xCF9A00,         //32位平面式代码段
           0x0000FFFF,0xCF9200      //32位平面式数据段
        }
```

这只是为了访问数据只要2个GDT就足够了，因为并没有重装代码段，所以这里只是为了完整性而给出3个GDT。

（4）通常，在进入保护模式时要关闭所有的中断，把IDTR的界限设置为0，CPU自动关闭所有中断，包括NMI，返回实模式后恢复IDTR并开中断。

（5）A20地址线的控制对于正确访问整个内存也很重要，在进入保护模式前，要让8042打开A20地址线，否则会出现4GB内存中的混乱。

在这个例子里，FS段寄存器设成可访问4GB内存的基址和界限，由于在DOS中很少有程序会用到GS、FS这两个386增加的段寄存器，所以当要读写4GB范围中的任一个地方时，都可通过FS段来达到，直到FS在实模式下被重装入冲掉为止。

这个例子在386SX、386DX、486上都运行通过。例子里加有十分详细的注释，由于这一程序是用BC 3.1编译连接的，而其连接器不能为DOS程序处理32位寄存器，所以直接在代码中加入操作码前缀0x66和地址前缀0x67，以便让DOS实模式下的16位程序可用32位寄存器和地址。程序的右边以注释形式给出等效的32位指令。

要注意，16位的指令中，mov al, byte ptr [BX]的指令码正好是32位的指令mov al, byte ptr[EDI]。

读者可用这个程序验证BIOS是否同时在两个区域出现。如果有线性定址能力的VESA显示卡（如TNT2），还可进一步验证线性显示缓冲区在1MB之上的工作情况。

第3章 Windows运行机理

3.1 内核分析(1)

3.1.1 运行机理

1. 概述

我们知道，DOS是一个开放的操作系统，应用程序和操作系统在同一个级别上，所以应用程序能控制整个机器的所有资源。这在DOS的早期还没什么问题，但是，后来随着应用程序的增加，系统就出现了一个很严重的问题—资源冲突。

当Windows 3.x推出时，市场上已有很多优秀的DOS软件。为了不失去巨大的市场，微软公司引入了全新的方法，让每个DOS程序和Windows程序都认为自己拥有所有的硬件资源。它们对系统硬件的操作是通过一些虚拟设备（VxD）来实现的，这就是所谓的虚拟机（VM）。之所以称为虚拟机，是因为它有完整的内存空间、I/O端口，以及中断向量。每个DOS都是一个VM，而所有的Win32的进程都运行在一个叫System VM中。其中，VxD中的“x”代表任意的设备。例如，VDD表示虚拟显示设备，VDMAD表示虚拟DMA设备。对于熟悉DOS的人而言，可以把VxD看做是32位的DOS。

Windows是怎么实现一个多任务的操作系统呢？原理很简单，就是CPU把运算时间轮流地分给每个虚拟机。这样，在Windows 3.x里，Windows程序之间用的是合作多任务，虚拟机之间用的是优先级多任务。而管理所有VxD和时间调试策略的程序就是虚拟机管理器（VMM）。虚拟机管理器是Windows的核心，它控制着计算机的主存、CPU的执行时间和外围设备功能。VMM结构图如图3.1所示。

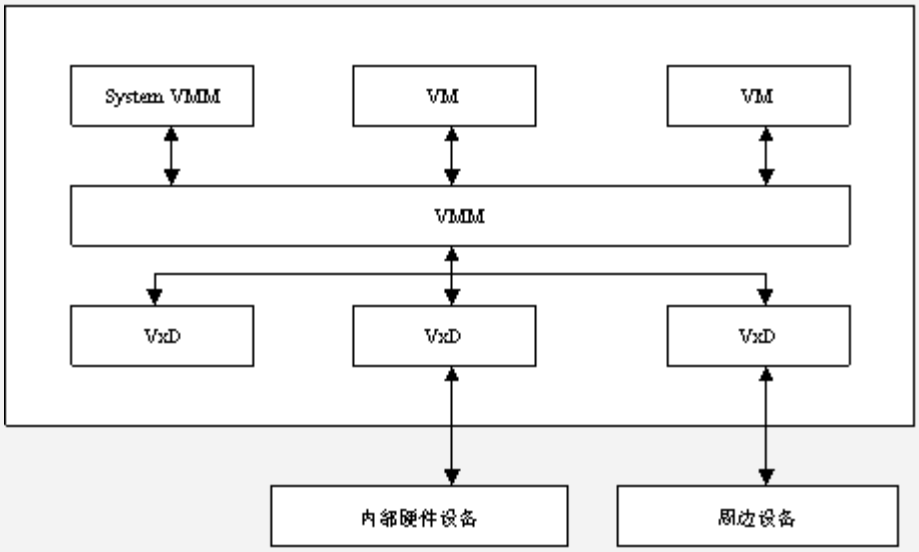


图3.1 VMM结构图

- 虚拟机管理器

VMM是一个32位的保护模式程序。它的主要任务是建立和维护一个支持虚拟机的框架，并对每个VM提供服务。例如，它要创建、运行和结束一个虚拟机。VMM是众多的系统VxD程序之一，放在系统目录下的VMM32.VxD文件中。VMM是第一个被加载到内存的VxD程序。它创建系统虚拟机并初始化其他的VxD程序，也为这些VxD程序提供许多服务。

VMM和VxD的操作模式和真正的程序不同。在大多数时候，它们是潜伏的。当应用程序在系统中运行时，这些VxD程序没有被激活。 当某些需要它们处理的中断/错误/事件发生时，它们才被唤醒。

- 虚拟设备驱动程序

在DOS程序中，虚拟设备驱动程序能控制系统的一切资源。当它们在虚拟机中运行时，Windows需要为每一个设备建立一种虚拟的设备来模拟DOS对硬件的操作。例如，在DOS程序中按下键盘时，这个事件消息首先会通知VMM，VMM接到它感兴趣的消息后，会向所有的VxD发送这个消息。当键盘VxD接收到后，会把中断发送给VMs。一个VxD程序通常控制真正的硬件设备，并对该设备在各个虚拟机之间的共享进行管理。

尽管如此，并不是说每个VxD程序必须和一个硬件设备相联。虽然VxD程序是用来虚拟硬件设备的，但是我们也可以把VxD程序看做是在第0级别的DLL。如果需要编写一个在第0级别才能工作的程序，就可以编一个VxD程序来为你完成这个工作。这样，由于此VxD程序并没有虚拟任何设备，就可以把它仅仅看做是你的程序的扩展。

CIH病毒就是一个VxD，所以能对硬件直接设置修改。

VxD是系统中权力最大的程序。由于它们可以对系统做任何事情，所以它们是极度危险的。一个恶意的或错误的VxD程序可以毁掉整个系统。操作系统对于恶意的、错误的VxD程序没有任何的保护措施。

VxD程序是Windows 3.1和Windows 9x特有的，在Windows NT下不能运行。现在Windows NT下的驱动程序已经改为WDM，它比VxD更规范，标准对系统的控制也有更严格的限制。

Windows 95下有两种VxD，静态VxD和动态VxD。静态VxD是那些从系统启动就被加载，在系统关闭之前一直存在于内存中的VxD程序。这种VxD是在Windows 3.x时产生的。动态VxD是在Windows 9x下才有的。动态VxD程序可以在需要的时候，通过程序本身加载或卸载。这些程序大多数都是用来控制设置管理器和输入输出监视器加载的即插即用设备的。

2. 虚拟机管理器

虚拟机管理器(VMM)是Windows 9x操作系统的真正内核。它建立并维护起所有的虚拟机，同时为其他VxD程序提供许多重要的服务。VMM处在VM和VxD之间。所有在VM上运行的软件和VxD之间通过VMM接口连接起来。

VMM提供了一组服务例程，它们可以创建、撤销、运行、同步以及改变所有VM的状态。VMM还提供了调试服务例程、内存管理及I/O管理和截取软中断服务。

第3章 Windows运行机理

3.1 内核分析(2)

① 内存的物理地址空间

VMM使用80386的保护模式管理内存。从认识CPU一章中，我们知道，在80386以后，系统能提供4GB的32位的虚拟空间。VMM在使用空间上把它们分为4个区域，如图3.2所示。

- 私有区
- 共享区
- 系统区
- DOS区

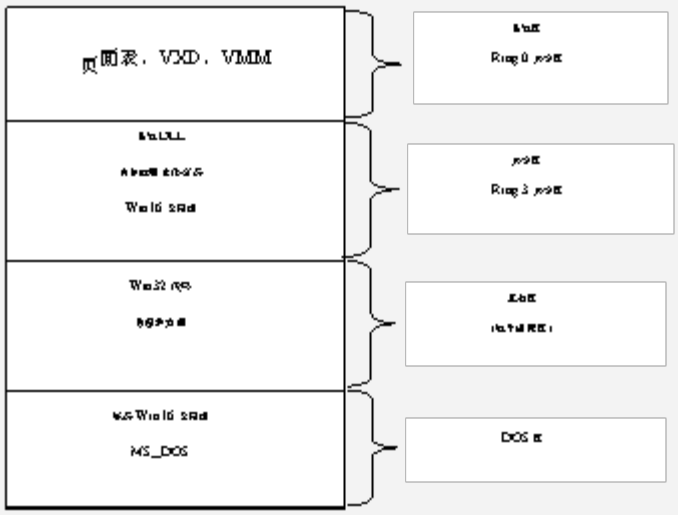


图3.2 VMM对使用空间的划分

私有区地址是从4MB到2GB。这是Win32应用程序运行的空间。每个Win32的进程都有它自己的2GB（要减去4MB）的空间，被Win32应用系统用来存放自己的代码和资源。这块区域是私有的，因为每个Win32程序映射到不同的物理空间上。当一个Win32程序访问4MB空间内时，它其实访问的是映射的某物理空间。

共享区地址是从2GB到3GB。这个区域是被虚拟机内的所有应用程序共享的。系统DLL（user32，kernel32，gid32）和Win16进程（由于Win16要求在共享空间运行）都驻存在这里。

系统区地址是从3GB到4GB的线性空间的顶端。这里是Win9x为第0级的超级进程VMM和VxD专门开辟的区域，并且此空间也是共享的。

DOS区地址是从0到4MB的空间内，这个空间是专为DOS的应用程序留下的，另外，Win16应用程序堆栈的一小部分也放在这里。

② 内存的服务程序

VMM中使用了虚拟存储的技术，能够克服物理内存的限制。尽管在物理上不存在，但理论上4GB的空间是能被访问的。通过从RAM和次级存储器设备上交换（分页）代码和数据以及将代码和数据交换到RAM和次级存储器设备上以实现虚拟技术。因为VxD驻留在32位的保护模式部分，所以它应该可以直接访问所有的内存空间，但内存的管理是通过VMM来完成的，所以它只能通过存储器管理服务获得的内存空间。

Windows决定实际有效的虚拟存储器的数量和有效的磁盘空间的数量。实际有效的虚拟存储器的数量基于系统物理上的总量，可以手工指定。

存储器管理程序在外部程序需要时，会一直分配物理空间，直到物理存储器已经用尽。然后，它会从物理存储器移动4KB的代码或数据页到磁盘上，以使附加的物理存储器有效。Windows中是按4KB的大小来对内存空间进行分页的。这种分页对程序来说是透明的。如果程序企图访问某部分已交换到磁盘上的数据，则会产生一个页错中断。然后存储器管理程序将其页换出存储器，并恢复该程序所需要的那些页。

下面列出了Windows存储器管理服务。列出的服务构成了公共使用子集。

- 系统目标管理

Alloacte_Device_Cb_Area

- 设备虚拟V86页管理

Assign_Device_V86_Pages

- 系统页分配程序

HeapAllocate

HeapFree

- 系统页分配程序

CopyPageTable

MapIntoV86

ModifyPage bits

PageAllocate

PageLock

PageUnlock

PageGetAllocInfo

PhyIntoV86

- 查看保护方式中的物理设备存储器

MapehysToLiner

DataAccessServices

GetFristV86Page

- 对保护方式API的专用服务

实例数据管理

查看V86空间

中断处理程序

线程调度程序

3. 虚拟设备

VxD的功能十分强大，它不但能“虚拟”某种设备，还能给别的VxD或应用程序提供服务。

VxD可以和VMM一起被静态地装入系统，也可以由应用程序主动地装入系统。因为VxD就在第0级工作，并且有极高的权限，所以VxD能访问任何的硬件，不仅可以访问任何的物理空间，还可以捕获软件中断和I/O端口以及其他程序对内存的访问，就连硬件中断也可以被它捕获。

① VxD的组成

安装一个VxD的过程有下面几个部分，如图3.3所示。

- 实模式的初始化代码和数据在完成以下4部分后，被系统销毁。
- 保护模式（PM）初始化代码部分，完成后销毁。
- 保护模式（PM）初始化代码数据，完成后销毁。
- PM代码，包括设备过程、API和回调过程，以及服务例程。
- PM数据，包括设备描述符块、服务表，以及全局数据。

② VxD的加载过程

Windows 9x支持静态加载和动态加载两种加载方式。静态加载的VxD是在Windows初始化时被自动加载的，只有当Windows结束运行后，它才会卸载。Windows 9x中可以通过两种方法来加载静态的VxD。

- 直接在SYSTEM.INI中加入如下一行代码:

  Device =VxD_NAME
- 可以在Windows 9x注册表中的HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\key\StaticVxD子键下加入如下的VxD的路径和名字:

  VxD__NAME＝PATHNAME

这种动态加载的VxD不是和VMM一起在Windows启动时一起装入内存的，而由应用程序或另外的VxD装入，并且也可以通过VxD或其他应用程序动态地删除，所以，动态的VxD就有很大的灵活性。

第3章 Windows运行机理

3.1 内核分析(3)

如果一个VxD只是为了某个应用提供某种服务，选择动态加载就比较好，因为VxD能在需要时加载，在用完后就立即卸载。这两种加载方式所响应的VMM消息有一点不同，有的只能响应静态VxD，有的则只能响应动态VxD。但大多数消息对这两种方式都能响应。

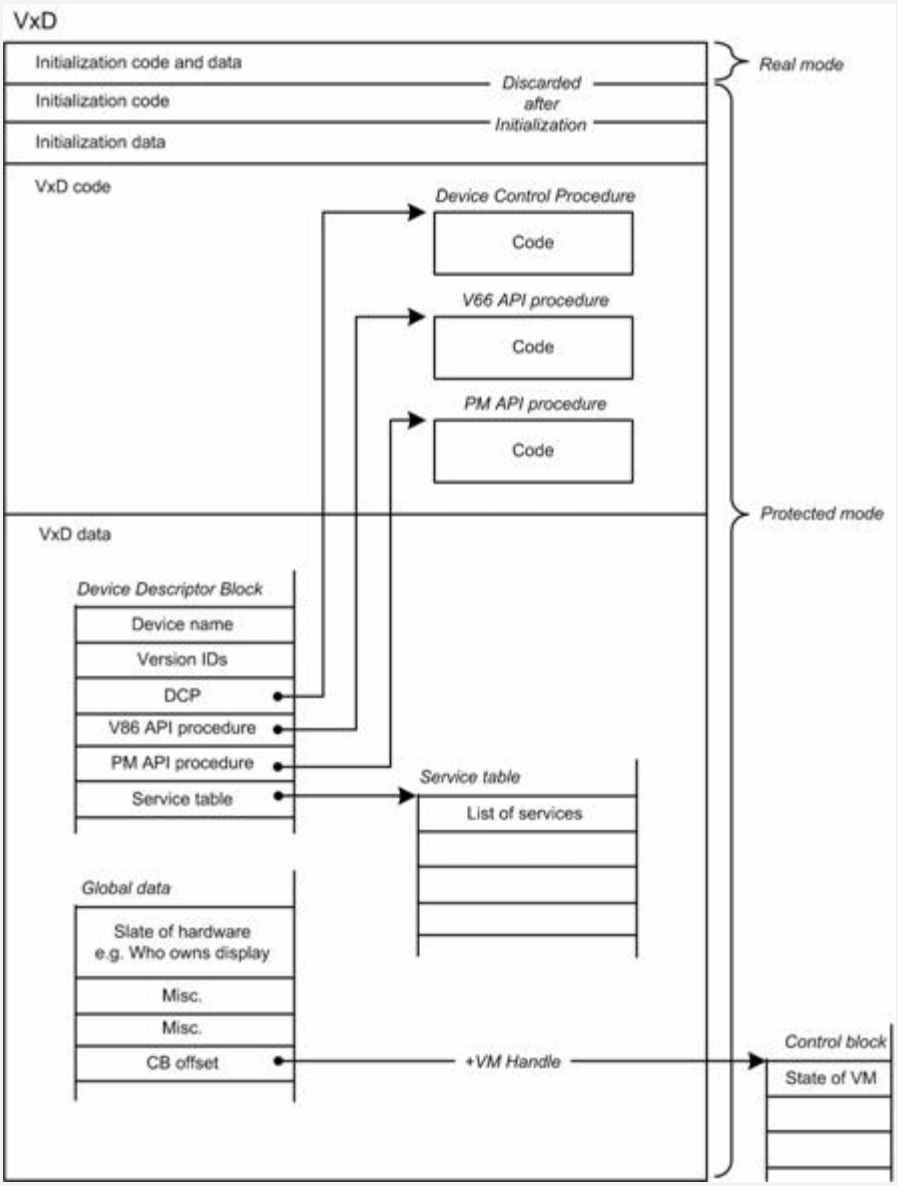


图3.3 VxD安装过程

③ DDB的结构

设备描述块（The Device Descriptor Block）简称DDB，是VMM联系VxD的句柄。DDB中包括

了VxD的信息和指向VxD主要的入口指针。当然，为了给其他的应用程序使用，也可以包括指向其他入口的指针。表3.1是DDB的数据结构。

表3.1 DDB的数据结构

字段区域	描述
Name	8个字节的VxD名称
Major Version	VxD的主版号，与Windows的版本号无关
Minor Version	VxD的从版号，与Windows的版本号无关
Device Control Procedure	设备控制过程的地址
Device ID	Microsoft分配的惟一的ID号
Initialization Order	通常是Undefine_Init_Order 。如果要强制在某个指定的VxD初始化之前或结束之后进行初始化，那就在VMM.INC中找到相应的Init_Order加1或减1
Service Table	服务表的地址
V86 API Procedure	V86 API函数的地址
PM API Procedure	PM API函数的地址

VxD源程序中的标号是不区分大小写的，大写、小写或者混合起来用，都可以。

下面对这些字段做些说明

- Name：VxD的名字，最多8个字符。它必须是大写！在系统中的所有VxD程序里，它们的名字不能重复，每个VxD的名字应该是惟一的。这个宏同时也会根据这个名字产生DDB的名字，产生的办法就是在这个名字的后面加上_DDB。
- MajorVer和MinorVer：VxD的主要的和次要的版本。
- CtrlProc：VxD程序的设备控制函数的名字。设备控制函数是一个接受和处理VxD程序的控制消息的函数。你可以把设备控制函数看做Windows函数的等价物。
- DeviceID：VxD程序的16位惟一标识符，当且仅当VxD程序需要处理以下情况时，需要用到这个ID：

  • VxD程序导出一些供其他VxD程序使用的VxD服务。因为20H中断接口用设备ID来定位/区分VxD程序，所以一个惟一的ID对你的VxD程序是必要的。

  • VxD程序要在初始化中断2FH、1607H时通知实模式程序它的存在。

  • 有一些实模式软件（TSR）要用中断2FH、1605H来加载VxD程序。

如果VxD程序不需要一个惟一的设备ID，则可以把这一项设为UNDEFINED_DEVICE_ID；如果需要它，则可以向Microsoft申请一个。

- InitOrder：初始化的顺序。简单地说，就是加载的顺序。VMM就按照这个次序来加载VxD程序。每个VxD程序都有一个加载次序号，例如：

```
VMM_INIT_ORDER          EQU 000000000H
DEBUG_INIT_ORDER        EQU 000000000H
DEBUGCMD_INIT_ORDER     EQU 000000000H
PERF_INIT_ORDER         EQU 000900000H
APM_INIT_ORDER          EQU 001000000H
```

可以看到，VMM, DEBUG和DEBUGCMD是首先加载的VxD程序，然后是PERF和APM。初始化顺序值越低的VxD程序越先被加载。如果VxD程序在初始化时需要用到其他VxD程序提供的服务，那么必须把初始化顺序的值设得比你所要调用的那个VxD程序的值大。这样，当VxD程序加载时，所要的VxD就已经在内存中为你准备好了。如果不想去管VxD的初始化顺序，就把这个参数填写为UNDEFINED_INIT_ORDER 。

- V86Proc和PMProc：程序可以导出供V86和保护模式程序使用的API，这两个参数就是用来填写这些API的地址。记住，VxD程序除了监控系统虚拟机外，还要监控一个或多个运行在DOS或者保护模式下的虚拟机程序。VxD程序理所当然要为DOS和保护模式程序提供API支持。如果你不导出这些API，则可以不填这两个参数。
- RefData：这是输入输出监视器（IOS）要用到的参考数据。只有在一种情况下要用到这个参数，即当在为IOS编写一个层驱动程序时。否则，可以不填这个参数。

④ VxD的事件处理

当实模式初始化完成后，VMM将通过专门的消息方法来通知所有的VxD发生了什么。VxD的消息处理就像Windows的窗口消息处理一样，能通过如下一组切换函数：

```
switch   (事件){
case   系统初始化事件
    处理此消息代码
case   VM初始化
    处理此消息代码
    …
case   其他
    …
}
```

为了给VxD发送消息，VMM就会从VxD的DDB中取得设备控制函数的地址，在EAX中放置的是消息的值，EBX中放入当前VM的句柄，接着调用对应的函数。

第3章 Windows运行机理

3.1 内核分析(4)

3.1.2 LE文件的格式

VxD采用线性可执行文件格式（LE）。这种文件格式是为OS/2 2.0版设计的。它同时包含16位和32位代码，这也是VxD程序的需要。回想VxD在Windows 3.x的时代，从DOS启动Windows，Windows在把机器转到保护模式之前，需要在实模式下做一些初始化。实模式的16位代码必须和32位代码一起放在可执行文件中。所以，LE文件格式成为理所当然的选择。Windows NT驱动程序不必在实模式下初始化，所以它们不必使用LE文件格式。它们用的是PE文件格式。

在LE文件中，代码和数据被存放在几类运行属性不同的段中。以下是一些可用的段类。

- LCODE：页面锁定的代码和数据段。这种段被锁定在内存里。换句话说，它永远不会被放在硬盘上，所以一定要谨慎地使用这种段类，以免浪费宝贵的内存。但那些每时每刻都必须放在内存中的代码和数据应该放在这个段里。尤其是那些硬件中断处理程序。
- PCODE：可调页代码段。VMM可以对这种段实行调页处理，在这种段里的代码不必时刻放在内存里，当VMM需要物理内存的时候，它就会把这段放到硬盘上去。
- PDATA：可调页数据段。
- ICODE：仅用于初始化段。这种段里的代码仅仅用来进行VxD的初始化。当初始化完成后，VMM就把这段从内存中释放。
- DBOCODE：仅用于调试的代码数据段。当你要调试VxD程序时，就要用到这种段里的代码和数据，例如，它包含要调试的消息的处理代码。
- SCODE：静态代码和数据段。这种段时刻存在于内存中，即使VxD已经卸载，这种段对某些动态的VxD程序也很有用。这些VxD程序需要在某一Windows进程里不停地加载/卸载，而又要记录上次的环境和状态。
- RCODE：实模式初始化代码数据段。这种段包含实模式初始化需要的16位代码和数据。
- 16ICODE：16ICODE USE16保护模式初始化数据段。这是一个16位的段，它包含VxD要从保护模式拷贝到V86模式的代码。例如，如果要把一些V86的代码拷贝到一个虚拟机上时，想拷贝的代码就要放在这里。如果你把它放在其他的段里，编译程序就会产生错误的代码，例如，它会产生32位代码而不是16位代码。
- MCODE：锁定的消息字串。这种段包含了由VMM消息宏帮助编译的消息字串，这有助于构造驱动程序的国际版本。

VxD程序并不意味着必须包含以上所有的段，可以选择VxD程序需要的段。例如，如果VxD程序不进行实模式初始化，那么就不必包含RCODE段。

大多数时候，要用到LCODE，PCODE和PDATA段。作为一个VxD程序编写者，为代码和数据选择合适的段取决于自己的判断。总的来说，应该尽可能多地使用PCODE和PDATA。因为这样，VMM就可以

LCODE

在需要的时候把段调入调出内存。另外，硬件中断程序及其所用到的服务必须放在　　段里。

注意，不能直接地使用这些段类，你要用这些段类来定义段，这些段的定义被存放在模块定义文件（.def）中。下面是一个标准的模块定义文件：

```
VxD SthVxD DYNAMIC

DESCRIPTION
    'SthVxD (C) Beijing Herosoft Computer Technology Ltd.1996-2002'

SEGMENTS
    _LPTEXT CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _LTEXT  CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _LDATA  CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _TEXT   CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _DATA   CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    CONST   CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _TLS        CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _BSS        CLASS 'LCODE'   PRELOAD NONDISCARDABLE
    _ITEXT  CLASS 'ICODE'   DISCARDABLE
    _IDATA  CLASS 'ICODE'   DISCARDABLE
    _PTEXT  CLASS 'PCODE'     NONDISCARDABLE
    _PDATA  CLASS 'PDATA'   NONDISCARDABLE SHARED
    _STEXT  CLASS 'SCODE'   RESIDENT
    _SDATA  CLASS 'SCODE'    RESIDENT
    _DBOSTART   CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
    _DBOCODE    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
    _DBODATA    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
    _16ICODE    CLASS '16ICODE' PRELOAD DISCARDABLE
    _RCODE  CLASS 'RCODE'
EXPORTS
    SthVxD_DDB @1
```

第一个声明定义了VxD的名称，一个VxD的名称必须是全部大写的。

接下来是段的定义，段的定义包括三个部分：段的名称、段类和要求的段的运行属性。可以看到，很多段都基于相同的段类，例如，_LPTEXT，_LTEXT，_LDATA都是基于LCODE段类，而且属性也完全一样。这样定义段有利于让代码更容易被理解。如，LCODE可以包含代码和数据，对于一个程序员来说，如果他能把数据放到_LDATA段里，把代码放到_LTEXT 段里，代码就会显得很容易理解。最后，这两个段都会被编译到最后的可执行程序的同一个段内。

一个VxD程序导出且仅导出一个标记：它的设备描述块（DDB）。DDB实际上是一个结构，它包含了VMM需要知道的所有的VxD信息。必须在模块定义文件中导出DDB。

在大多数时候，可以把上面的.DEF文件用到新建的VxD项目中去。只要把.DEF文件里第一行和最后一行的VxD名字改掉就可以了。在一个汇编的VxD项目中，段的定义是不必要的，段的定义主要用于C的VxD项目编写，但用在汇编里也是可以的。你会得到一大堆警告的信息，但是它能汇编成功。也可以删掉你项目里没有用到的段定义，从而去掉这些讨厌的警告信息。

vmm.inc包含了许多用于定义源文件中的段的宏：

```
_LTEXT      VxD_LOCKED_CODE_SEG
_PTEXT      VxD_PAGEABLE_CODE_SEG
_DBOCODE        VxD_DEBUG_ONLY_CODE_SEG
_ITEXT      VxD_INIT_CODE_SEG
_LDATA      VxD_LOCKED_DATA_SEG
_IDATA      VxD_IDATA_SEG
_PDATA      VxD_PAGEABLE_DATA_SEG
_STEXT      VxD_STATIC_CODE_SEG
_SDATA      VxD_STATIC_DATA_SEG
_DBODATA        VxD_DEBUG_ONLY_DATA_SEG
_16ICODE        VxD_16BIT_INIT_SEG
_RCODE      VxD_REAL_INIT_SEG
```

第3章 Windows运行机理

3.1 内核分析(7)

（2）用Win32应用程序里的 CreateFile API。你在调用CreateFile时，动态VxD要以下面的格式填写：

\\.\VxD完整路径名

例如，如果要加载一个在当前目录下名为SthVxD的动态VxD，则需要做如下的工作，一般可以直接用C来编写主功能，然后和汇编进行连接：

```
hCVxD = CreateFile("\\\\.\\SthVxD", 0,0,0, CREATE_NEW,
                        FILE_FLAG_DELETE_ON_CLOSE, 0);
```

FILE_FLAG_DELETE_ON_CLOSE 这个标志用来说明该VxD在CreateFile返回的句柄关闭时被卸载。

如果用CreateFile来加载一个动态VxD，那么这个动态VxD必须处理w32_DeviceIoControl 消息。当动态VxD第一次被CreateFile函数加载的时候，WIN32向VxD发出这个消息。VxD响应这个消息，返回时，eax中的值必须为零。当应用程序调用DeviceIoControl API来与一个动态VxD通信时，w32_DeviceIoControl消息也被发送。

（3）当一个动态VxD在初始化时收到一个消息：

Sys_Dynamic_Device_Init

在结束时也收到一个控制消息：

Sys_Dynamic_Device_Exit

但动态VxD不会收到Sys_Critical_Init, Device_Init和Init_Complete控制消息，因为这些消息是在系统虚拟机初始化时发送的。除了这三个消息，动态VxD能收到所有的控制消息，只要它还在内存里。它可以做静态VxD可以做的所有事情。简单地说，动态VxD除了加载机制和接收到的初始化/结束消息跟静态VxD不同以外，它能做静态VxD所能做的一切。

当VxD在内存里的时候，除了接收和初始化及结束相关的消息外，它还要收到许多别的控制消息。这些消息有的是关于虚拟机管理器的，有的是关于各种事件的。例如，关于虚拟机的消息如下：

```
Create_VM
VM_Critical_Init
VM_Suspend
VM_Resume
Close_VM_Notify
Destroy_VM
```

选择地响应你所感兴趣的消息是你自己的责任。

（4）在VxD内创建函数

要在一个段里面定义函数，应该首先定义一个段，然后把函数放进去。例如，如果要把函数放到一个可调页段中，应该先定义一个可调页段：

```
VxD_PAGEABLE_CODE_SEG
（你的函数写在这里）
VxD_PAGEABLE_CODE_ENDS
```

可以在一个段里面插入多个的函数。作为一个VxD编写者，必须决定每一个函数应该放到哪个段里面去。如果函数必须时刻存在于内存中，如某些硬件中断处理程序，就把它们放到锁定页面段里面，否则，应该把它们放到可调页段。

（5）要用BeginProc和EndProc 宏来定义函数：

```
BeginProc 函数名
EndProc 函数名
```

使用BeginProc 宏还可以加上一些参数，想了解这些细节，你可以看看Win95 DDK的文档。大多数时候，你只用填写函数的名字就够了。

因为BeginProc-EndProc 宏比proc-endp 指令的功能要强，所以你应该用BeginProc-EndProc宏来代替proc-endp指令

3. VxD编程约定

① 寄存器的使用

VxD程序可以使用所有的寄存器，FS和GS。但是在改动段寄存器的时候一定要小心。尤其是，一定不要改动CS和SS的内容，除非你对将发生的事情有绝对的把握。你可以使用DS和ES，但一定要记住在返回时恢复它们的初值。有两个特征位尤其重要：方向和中断特征位。不要长时间地屏蔽中断。还有，如果你要改动方向特征位，不要忘了在返回之前恢复它的初值。

② 数传递约定

VxD服务函数有两种调用约定：寄存器法和堆栈法。调用寄存器法服务函数时，通过各种寄存器来传递服务函数的参数。并且，在调用完成后，检查寄存器的值来看操作是否成功。不要总是以为在调用服务函数后，主要寄存器的值还和以前一样。当调用堆栈法服务函数时，你把要传递的参数压栈，在eax得到返回值。堆栈调用法的服务函数保存ebx，esi，edi和ebp的值。许多寄存器调用法服务函数都源于Windows 3.x的时代。

在大多数时候，可以通过名字来区分这两种服务函数。如果一个函数的名字以下划线开头，如_HeapAllocate，它就是一个堆栈法的服务函数（除了少数从VWIN32.VxD导出的函数）。如果函数名不是以下划线开头，它就是一个寄存器法的服务函数。

③ 调用VxD服务函数

可以通过VMMCall和VxDCall 宏来调用VMM和VxD服务。这两个宏的语法是一样的。当你要调用VMM导出的VxD服务函数时，用VMMCall。当要用其他VxD程序导出的VxD服务函数时，用VxDCall。

```
VMMCall service                          ; 调用寄存器法服务函数
VMMCall  _service, <argument list>   ; 调用堆栈法服务函数
```

当调用堆栈法服务时，必须用角括号把你的参数列括起来。

```
VMMCall  _HeapAllocate, <<size mybuffer>, HeapLockedIfDP>
```

_HeapAllocate是一个堆栈法服务函数。它有两个参数，我们必须用角括号把它们括起来。由于第一个参数是一个宏，这个宏不能正确解释表达式，所以我们要再用一个角括号把它括起来。

4. VxD函数的调用方法

我们知道，VxD程序都有一个VxD的DDB列表，当VxD被加载时，DDB就会被装到Windows 95的系统内存里，Windows 95就是通过这个表把所有的VxD作为一个链表来进行管理的。Windows 95使用INT 20H来进行功能调用，凡是新VxD文件被装入内存的时候，都会产生一个INT 20H，在其后会紧跟着DDB的ID号码和服务函数号码。系统处理INT 20H时，就会去查找INT 20H的服务函数链表，当查找到函数ID和地址相同，就替换掉程序本身的指令。

VxD程序，包括VMM在内，通常要导出一系列的被别的VxD程序调用的公共函数，这些函数被称为VxD服务。调用这些服务的机制和在第三层级别运行的应用程序有很大的不同：每个导出VxD服务的VxD程序必须有一个惟一的ID，你可以从Microsoft得到一个这样的ID。这个ID是一个包含了一个VxD惟一的身份验证的16位的数字，例如：

```
UNDEFINED_DEVICE_ID EQU 00000H
VMM_DEVICE_ID           EQU 00001H
DEBUG_DEVICE_ID     EQU 00002H
VPICD_DEVICE_ID     EQU 00003H
VDMAD_DEVICE_ID     EQU 00004H
VTD_DEVICE_ID           EQU 00005H
```

第3章 Windows运行机理

3.1 内核分析(5)

每个宏都有与它相对应的结束宏，例如，如果要在源文件中定义一个_LTEXT段，应该写成如下：

```
VxD_LOCKED_CODE_SEG
(把你的代码写在这里)

VxD_LOCKED_CODE_ENDS
```

我们可以用VC++ Dump工具提供的DUMPBIN工具来分析以下VxD的文件结构和组织机理。可以进入MS_DOS输入如下的命令行（在光碟上的第三章\cpu降温\COOLCPU\BIN路径下的STHVxD）：

DUMPBIN /ALL STHVxD.VxD

就可以看到如下的信息：

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.


Dump of file sthvxd.vxd

File Type: VXD

    454C magic number
       0 byte order
       0 word order
       0 executable format level
       2 CPU type (**)
       4 operating system (**)
       0 module version
   38000 module flags
       4 number of memory pages
       2 object number of entry point
       0 offset of entry point
       0 object number of stack
       0 offset of stack
     200 memory page size
      2C bytes on last page
```

```
         61 fixup section size
          0 fixup section checksum
         6C loader section size
          0 loader section checksum
         C4 object table
          3 object table entries
        10C object map
          0 iterated data map
          0 resource table
          0 resource table entries
        11C resident names table
        126 entry table
          0 module directives table
          0 module directives entries
        130 fixup page table
        144 fixup record table
        191 imported modules name table
          0 imported modules
        191 imported procedures name table
          0 page checksum table
       1000 enumerated data pages
          2 preload page count
       162C non-resident name table
         4E non-resident name table size
          0 non-resident name checksum
          0 automatic data object
          0 debug information
          0 debug information size
          0 preload instance page count
          0 demand instance page count
          0 extra heap allocation
          0 offset of Windows resources
          0 size of Windows resources
        ABC device id
        400 DDK version

   OBJECT HEADER #1
        23C virtual size
          0 virtual address
```

```
    2045 flags
         Execute Read
         Has preload pages
         32-bit
       1 map index
       2 map size
444F434C reserved

OBJECT PAGE MAP #1

    Logical  Physical File     Flags
    Page      Page     Offset   Flags
    -------- -------- -------- --------
    00000001 00000001 00001000 Valid
    00000002 00000002 00001200 Valid

RAW DATA #1
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 04 BC 0A 05 00 00 00 53 74 68 56  ............SthV
00000020: 58 44 20 20 00 00 00 80 00 00 00 00 00 00 00 00  XD  ...  ........
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 76 65 72 50  ............verP
00000050: 50 00 00 00 31 76 73 52 32 76 73 52 33 76 73 52  P...1vsR2vsR3vsR
00000060: 8B 4C 24 08 85 C9 75 05 33 C0 C2 14 00 83 F9 FF  .L$...u.3.......
00000070: 75 08 E8 69 00 00 00 C2 14 00 83 F9 03 76 08 B8  u..i.........v..
00000080: 32 00 00 00 C2 14 00 8B 44 24 14 8B 54 24 10 50  2.......D$..T$.P
00000090: 52 8B 44 24 14 8B 54 24 0C 50 52 FF 14 8D FC FF  R.D$..T$.PR.....

000000A0: FF FF C2 14 00 CC CC CC CC CC CC CC CC CC CC CC  ................
000000B0: A1 00 00 00 00 8B 4C 24 10 8B 49 18 85 C9 74 08  ......L$..I...t.
000000C0: 8B 09 89 0D 00 00 00 00 C2 10 00 CC CC CC CC CC  ................
000000D0: E8 2A 01 00 00 B8 01 00 00 00 C3 CC CC CC CC CC  .*..............
000000E0: B8 01 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC  ................
000000F0: 8B 44 24 04 83 F8 01 74 0A 83 F8 02 74 15 33 C0  .D$....t....t.3.
00000100: C2 0C 00 8B 44 24 08 50 E8 63 00 00 00 83 C4 04  ....D$.P.c......
00000110: C2 0C 00 8B 44 24 0C 8B 4C 24 08 50 51 E8 5E 00  ....D$..L$.PQ.^.
00000120: 00 00 83 C4 08 C2 0C 00 CC CC CC CC CC CC CC CC  ................
00000130: 8B 44 24 04 83 F8 01 74 0A 83 F8 02 74 15 33 C0  .D$....t....t.3.
00000140: C2 0C 00 8B 44 24 08 50 E8 43 00 00 00 83 C4 04  ....D$.P.C......
```

```
00000150: C2 0C 00 8B 44 24 0C 8B 4C 24 08 50 51 E8 3E 00  ....D$..L$.PQ.>.
00000160: 00 00 83 C4 08 C2 0C 00 CC CC CC CC CC CC CC CC  ................
00000170: B8 01 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC  ................
00000180: B8 02 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC  ................
00000190: B8 01 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC  ................
000001A0: B8 02 00 00 00 C3 CC CC 83 F8 1B 75 09 E8 00 00  ...........u....
000001B0: 00 00 83 F8 01 C3 83 F8 1C 75 09 E8 10 FF FF FF  .........u......
000001C0: 83 F8 01 C3 83 F8 23 75 0E 56 52 53 51 55 E8 8D  ......#u.VRSQU..
000001D0: FE FF FF 83 F8 01 C3 F8 C3 A1 00 00 00 00 85 C0  ................
000001E0: 75 02 F9 C3 FB F4 F9 C3 56 8D 35 00 00 00 00 CD  u.......V.5.....
000001F0: 20 3A 00 01 00 5E B8 00 00 00 00 0F 93 C0 C3 56   :...^.........V
00000200: 8D 35 00 00 00 00 CD 20 2B 01 01 00 5E B8 00 00  .5..... +...^...
00000210: 00 00 0F 93 C0 C3 FF 75 18 FF 75 10 FF 75 1C E8  .......u..u..u..
00000220: CC FE FF FF 89 45 1C C3 FF 75 18 FF 75 10 FF 75  .....E...u..u..u
00000230: 1C E8 FA FE FF FF 89 45 1C C3 CC CC              .......E....
OBJECT HEADER #2
       B virtual size
       0 virtual address
    1005 flags
         Execute Read
         16:16 alias
          3 map index
          1 map size
444F4352 reserved

OBJECT PAGE MAP #2

    Logical  Physical File     Flags
    Page      Page     Offset   Flags
    -------- -------- -------- --------
    00000001 00000003 00001400 Valid

RAW DATA #2
  00020000: 33 DB 33 F6 66 33 D2 B8 00 00 C3 3.3.f3.....

OBJECT HEADER #3
      2C virtual size
       0 virtual address
    2015 flags
```

```
          Execute Read
          Discardable
          32-bit
        4 map index
        1 map size
444F4349 reserved

OBJECT PAGE MAP #3

    Logical  Physical File         Flags
    Page     Page     Offset         Flags
    -------- -------- -------- --------
    00000001 00000004 00001600 Valid

RAW DATA #3
00000000: 0D 0A 44 5F 45 5F 42 5F 55 5F 47 3D 3D 3D 3E 53  ..D_E_B_U_G===>S
00000010: 74 68 56 58 44 3C 3D 3D 3D 0D 0A CC CC CC CC CC  thVXD<===.......
00000020: E8 00 00 00 00 B8 01 00 00 00 C3 CC              ............

  Summary
```

第3章 Windows运行机理

3.1 内核分析(6)

3.1.3 VxD的设计实现

VxD的设计并不是通常我们所讲的调用API的Windows程序，而是通过对DDK的调用来工作。DDK可以从微软的网站上下载。在DDK中有很多VxD的例子，我们在设计时可以作为参照样板。VxD的设计一般要直接用汇编编程，并且要直接地操作硬件，所以设计比较困难。不过，用汇编写VxD的框架结构、用C来完成具体的工作实现就会大大地提高开发的效率，后面的例子中就是使用了这种方法。

1. 静态**VxD**

在下列情况下，VMM加载一个静态VxD:

（1）此VxD在注册表中的如下位置有定义:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\key\StaticVxD=VxD带路径文件名

（2）此VxD在system.ini中的[386enh]行下有定义:

[386enh] section:

device=VxD带路径文件名

在开发的时候，建议从system.ini载入VxD程序，因为这样如果VxD程序有错而将导致Windows不能启动，可以在DOS下修改system.ini，而如果使用注册表载入的办法，就无法修改了。

当VMM加载静态VxD程序时，VxD程序会按以下顺序接收到3个系统控制消息。

（1）Sys_Critical_Init：VMM在转入到保护模式后，开放中断前发出这个控制消息。大多数VxD程序不要用这个消息，除非VxD程序要接管一些其他VxD程序或者保护模式程序要用到的中断。既然处理这个消息的时候，这个中断还没有打开，就可以确定在你接管这个中断的时候，此中断不会被调用。VxD程序为其他的VxD程序提供了一些VxD服务。

（2）Device_Init：控制消息时需要调用一些VxD服务，既然Sys_Critical_Init 控制消息在Device_Init消息之前被发送，所以你应该在Sys_Critical_Init 消息发送时初始化你的程序。

如果要对这消息进行处理，则应该尽可能快地做完初始化工作，以免太长的执行时间导致硬中断丢失(记住，中断还没打开)。Device_Init VMM在开放中断后发送此信息。大多数VxD程序都在得到这个消息时初始化。因为中断都开放了，所以耗时的操作也可以在这里执行，而不怕会导致硬中断的丢失。你可以在这时进行初始化(如果你需要的话)。

（3）Init_Complete：在所有的VxD程序处理完Device_Init 消息之后,VMM释放初始化段（ICODE和RCODE段类）之前，VMM发出这个控制消息。只有少数几个VxD要处理这个消息。

VxD程序在成功地初始化后，必须将返回标志清零，反之，必须在返回之前把返回标志设为出错信息。如果VxD不需要初始化，就不必对这些消息进行处理。

当要结束静态VxD的时候，VMM发送如下的控制消息。

（1）System_Exit2：当VxD程序收到这个消息，Windows 9x正在关闭系统，除了系统虚拟机外，所有其他虚拟机都已经退出了。尽管如此，CPU仍然处于保护模式下，在系统虚拟机上执行实模式编码也是安全的。这时，Kernel32.dll也已经被卸载了。

（2）Sys_Critical_Exit2：当所有的VxD完成对System_Exit2的响应处理并且中断都被关闭后，VxD收到这个消息。

许多VxD程序并不要响应这两个消息，除非你要为系统做转换到实模式的准备。要知道，当Windows 95关闭时，它进入到实模式。所以，如果VxD程序对实模式影像做了一些会导致它不稳定的操作，它就需要在这时进行恢复。

你也许会感到奇怪：为什么这两个消息后面都跟着个“2”？这是因为在VMM加载VxD程序的时候，它是按照初始化顺序值小的VxD先加载的顺序加载的，这样，VxD程序就可以使用那些在它们之前加载的VxD程序提供的服务。例如，VxD2要用到VxD1中的服务，它就必须把它的初始化顺序值定义得比VxD1小。加载的顺序是：

..... VxD1 => VxD2 => VxD3 .....

那么卸载的时候，理所当然地是初始化顺序值大的VxD程序先被卸载，这样它们仍然可以使用比它们后加载的那些VxD程序提供的服务。如上面的例子，次序是：

.... VxD3 => VxD2 => VxD1.....

在上边的例子中，如果VxD2在初始化时调用了VxD1中的某些服务，那么卸载时它可能也要再次用到一些VxD1中的服务。System_Exit2和Sys_Critical_Exit2是按反初始化顺序发送的。这表示，当VxD2接受到这些消息时，VxD1还没有被卸载，它仍可以调用VxD1的服务，而System_Exit和Sys_Critical_Exit消息不是按照反初始化顺序发送的。这意味着，你不能肯定你是否仍能调用在你之前加载的VxD提供的VxD服务。

现在的VxD程序不应该使用这些消息，而应该使用以下两种退出消息。

（1）Device_Reboot_Notify2 告诉VxD程序VMM正在准备重新启动系统。这时候，不管是中断还是开放的Crit_Reboot_Notify2，都会告诉VxD程序VMM正在准备重新启动系统，并把中断关闭。

（2）Device_Reboot_Notify和Crit_Reboot_Notify 消息一样，但它们并不是像“2”版本的消息那样，按反初始化顺序发送。其他就和Device_Reboot_Notify2一样了。

2. 动态**VxD**

动态VxD在Windows 9x里可以动态地被加载和卸载。这个特点在Windows 3.x下是没有的。动态VxD程序的主要作用是用来支持某些动态的硬件设备的重装，比如即插即用设备。尽管如此，可以从Win32程序中加载/卸载它，也可以把它看做是程序的一个到ring0的扩展。

上一节我们提到的例子是一个静态的VxD，你可以把它转换成一个动态的VxD，只要在.def文件中VxD标记的后面加上关键字DYNAMIC：

VxD STHVxD DYNAMIC

这就是把一个静态VxD转换成一个动态的VxD所要做的一切。

一个动态的VxD可以按以下的方法被加载。

（1）把它放到Windows目录下的\SYSTEM\IOSUBSYS目录中。在这个目录里的VxD会被输入输出监视器(IOS)加载。这些VxD必须支持层设备驱动。所以用这种方法加载动态VxD并不是一个好办法。

用VxD加载服务。VxDLDR是一个可以加载动态VxD的静态VxD。你可以在其他VxD里面或者在16位代码里面调用它的服务。

第3章 Windows运行机理

3.1 内核分析(8)

可以看到，VMM的ID是1，VPICD的ID是3等。VMM用这些ID来找到导出所需VxD服务的VxD程序。当一个VxD程序导出VxD服务时，它把所有服务的地址存在一个表里面。所以，你还需要通过服务分支表里面服务的索引来找到你所要的服务。例如，如果你要调用第一个服务，GetVersion服务，就要指定0（这个索引是从0开始的）。调用VxD服务实际上包括中断20H，你的代码产生一个中断20h，并带有一个双字的值，这个值包含了设备ID和服务索引。例如，如果你要调用一个VxD程序导出的VxD服务，假设VxD程序设备ID是000DH，服务号码是1，那么代码应该是：

```
int  20h
dd  000D0001h
```

跟在中断20H后的双字的高字包含设备ID。低字是在服务列表中的索引。

当20H中断执行时，VMM就得到了控制权，并马上检测跟着的双字。然后它提出设备ID用来找到VxD程序，用服务索引来定位在那个VxD程序中所要求的服务的地址。

可以看到，这个操作是很费时的。VMM必须浪费很多时间来定位VxD程序和所要服务的地址，所以VMM作了个小小的弊 。当中断20H操作成功后，VMM抓取链接。这就是说，VMM用直接的服务调用来替代20H中断和它后面的双字。所以，上面的20H中断代码片断就被改变成:

```
call dword ptr [VxD_Service_Address]
```

这个方法很不错，因为int 20h+dword加一个双字用6个字节，正好和call dword ptr结构相等。所以，接下来的服务调用是快速而有效的。这个方法具有直接性、简洁性。一方面，它减轻了VMM和VxD载入器的工作量，因为它们不用定位VxD中所有的服务，那些没有执行过的服务将会保持原样。另一方面，一旦一个静态VxD程序导出的服务被调用，那么就不可能把这个静态的VxD程序卸载了。由于VMM把调用锁定到VxD服务的实际地址上，如果提供这个服务的VxD程序从内存中被卸载了，其他VxD程序调用这个服务时，就会很快地因为调用无效的内存地址而导致系统崩溃。没有办法来消除抓取的链接。这个问题的结论是动态VxD不适合作为服务提供者。

3.1.4 【实例】：CPU降温程序代码分析

有人可能认为VxD很高深，其实不然。下面介绍一个简单的CPU降温的程序，来加深大家的理解。

1. 程序的组成

这个程序由两个部分组成。

其一，VxD模块。它是一个动态VxD，可以用以下的处理过程来分析CPU降温的基本原理：

（1）被加载时，就对VMM注册空闲的消息（idle）处理函数。

（2）当VMM空闲时，就会自动地调用VxD的注册的消息函数。处理函数通过一条HLT指令使CPU暂停，当CPU暂停时，很多器件就会停止工作，这样就可以降温。

（3）CPU接收到新指令时，中止HLT命令，即退出函数。

（4）当VMM空闲时，继续调用（2）。这样循环进行，直到程序退出。

其二，主程序模块。它负责装载和卸载VxD，并处理用户的界面及响应用户事件。基本的流程如下：

（1）如果装载降温VxD程序，则成功转入（2），否则就退出系统。

（2）设置VxD的处理函数，即VxD的主处理函数。

（3）生成一个托盘（即在Windows 9x任务条右下角上的可控制的小图标）以控制VxD的运行状态。

2. 程序的编译

我们现在就编译程序，先看一下运行和效果。

① 编译动态的VxD文件

在本书配套的光碟上，可以在 \ COOLCPU \STHVxD目录下找到该程序，我们可以看到有5个文件。

- CVxDctrl.asm：动态VxD的运行框架，这段必须用汇编编写。
- SthVxD.c：VxD工作代码（由汇编调用）。它是一个C的模块，主要负责应用程序和VxD的内核进行接口。通常VxD可以用汇编来编程，但这样会使开发效率很低。为了更方便地开发，一般用汇编生成框架，然后用C语言编写程序的主要部件，这样就会大大地提高开发的效率。
- SthVxD.def：VxD结构的定义文件。它是每个VxD必需的，是标准的一部分。
- Makefile：编译的参数设置文件。
- SETPATH.BAT：设置编译的路径的文件。

当然，要编译以上的文件，您还需要注意以下几点

- 安装VC++ 5.0以上的版本。
- 必须具备Windows 9x Device Driver Development Kit。可以从 http://download.microsoft.com/download/win98SE/Install/Gold/W98/EN-US/98DDK.EXE下载Windows 98 DDK。不过，因为Windows 98 DDK的很多库和函数都发生了改变，所以此程序不能直接使用下载的DDK。我们可以用windows 95的DDK，它们在碟的COOLCPU的win95DDK目录中。Windows 95 DDK中包括Inc32和Lib，其中，Inc32目录中包括了32位的头文件，Lib目录中包括所有的库文件上。
- 还有一点最重要的是，您的操作系统一定要是Windows 9x系列的，因为VxD只能用在Windows 9x系列的操作系统中。
- 设置编译器的文件和头文件的查找路径，需要修改SETPATH.BAT文件中的路径设置，其原始内容如下：

```
@ECHO OFF
SET
LIB=    D:\win95DDK\LIB;D:\98DDK\LIB;C:\MSDEV\LIB;
```

```
        D:\98DDK\lib\i386\free;%LIB%
SET     PATH=D:\98DDK\BIN;D:\98DDK\bin\win98;%PATH%
SET     INCLUDE=D:\WIN95\INC32;
        D:\Microsoft Visual Studio\VC98\Include;D:\98DDK\ inc\win98;
@ECHO ON
```

D:\98DDK是DDK的安装目录，您可把这个目录改成自己机器的DDK安装的目录。

D:\win95DDK是光盘中Windows 95的DDK的目录。当您把此目录复制到硬盘后，需要修改成对应的目录。

- 设置Makefile，来设置编译路径，在Makefile下可以找到如下的一行语句：

```
CFLAGS= -DWIN32 -DCON -Di386 -D_X86_ -D_NTWIN -W3 -Gs -D_DEBUG
        -Zi -O2 -IC:\MSDEV\INCLUDE -ID:\WIN95DDK\INC32
```

第3章 Windows运行机理

3.1 内核分析(9)

把C:\MSDEV\INCLUDE这个路径（其中，C:\MSDEV是VC++的头文件.h文件的路径）修改为您机器安装的VC++的路径即可。

例如，如VC++安装在C:\Program Files\Microsoft Visual Studio\VC98中，就可以将之改为：

```
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
```

把D:\WIN95DDK\INC32修改为DDK的安装路径的头文件的路径。

例如，如果DDK的目录为C:\WIN98DDK，就可以将之修改为E:\98DDK\inc\win98。

接下来，就可以编译。可按如下步骤进行：

（1）进入MS DOS方式。

进入STHVxD文件的路径，例如：

```
CD  D:\COOLCPU\STHVxD
```

（2）运行nmake.exe程序，对整个程序进行编译。当BIN目录下生成SthVxD.VxD的文件时，该VxD就编译完成了。在编译完成后，会出现一些警告，这是正常的，没有什么问题。

注意：一定要把光碟上的程序复制到硬盘上才能进行编译！

② 编译主程序（CoolCpu）

VxD文件编译好后，主程序就很容易编译。只需打开VC＋＋的open workspace文件CoolCpu.dsp或CoolCPU.mak。

③ 运行程序

直接编译，就可以看到在BIN目录中生成了一个CoolCPU.EXE文件。当在编译环境中，BulidExecute系统将弹出一个写有“can’t execute program”的信息提示框。这是为什么呢？

其实，这是CoolCPU.exe在当前的编译目录中找SthVxD.VxD的文件，因为当前路径下没有这个VxD文件，所以就弹出错误的对话框。直接到BIN文件下运行CoolCPU.exe,就可以看见在Windows的任务栏的右下角出现了一个小云雨的图标。当单击此图标时，弹出菜单，“空闲时让CPU节能”的小钩被打上时，表示允许CPU使用降温功能。去掉小钩时，表示不用此降温功能。

3. 程序的分析

下面我们来分析一下这个程序。

首先看一下VxD的基本框架。从CVxDctrl.asm文件中，可以看到如下的程序结构。

```
PAGE 58,132
;******************************************************
TITLE CONTROL - ControlDispatch for VxD in C
;******************************************************
```

```
;
.586p
;*********************************************************
;                     包含头
;*********************************************************

    .xlist
    include vmm.inc
    include debug.inc
    .list

;编译成动态VxD，动态的VxD为1
SthVxD_DYNAMIC EQU 1

;VxD的ID号
CVxD_DEVICE_ID EQU 0ABCH

ifdef _VxD_SERVICES
;定义可以被其他VxD调用的接口函数
Create_CVxD_Service_Table = 1

;可以被其他VxD调用的接口函数表
Begin_Service_Table CVxD
CVxD_Service     _CVxD_Get_Version, VxD_LOCKED_CODE
End_Service_Table CVxD
Endif
```

很多人可能对汇编不是很熟悉，但这不要紧。在这段汇编中用了很多的宏汇编语句，使整个代码很像高级语言。

在VxD的这段汇编的代码中，很多东西是必须的，下面我们来分别介绍。

```
.586p
```

告诉编译器要使用CPU特权指令的80586指令系统，还可以使用.386p或者.486p等。

```
include vmm.inc
```

每个VxD源代码都必须包含imm.inc。它包含了代码中宏的定义。可以根据需要包含其他的库文件，如Pci.inc（PCI设备）的宏。

```
SthVxD_DYNAMIC EQU 1
```

表示SthVxD_DYNAMIC等于1，相当于C语言中的 # define语句的作用。

```
CVxD_DEVICE_ID EQU 0ABCH
```

每个VxD程序的16位惟一标识符，有了这个ID就可以导出一些供其他VxD程序使用的VxD服务。

如果VxD程序不需要一个惟一的设备ID，可以把这一项设为UNDEFINED_DEVICE_ID ，还可以由微软分配一个固定的ID号。也可以任意设置，只要以前没有使用过，此处设置为0ABCH。

```
ifdef _VxD_SERVICES
;定义可以被其他VxD调用的接口函数
Create_CVxD_Service_Table = 1


;可以被其他VxD调用的接口函数表
Begin_Service_Table CVxD
CVxD_Service     _CVxD_Get_Version, VxD_LOCKED_CODE
End_Service_Table CVxD
Endif
```

定义被其他函数调用的接口的申明，很多VxD中的函数可以给其他的VxD进行调用，有些VxD提供了一此功能能让别的VxD使用，就可以像动态连接库一样引出一些函数。

```
DECLARE_VIRTUAL_DEVICE  SthVxD, 5, 0, CVxD_Control, CVxD_DEVICE_ID,\
    UNDEFINED_INIT_ORDER, CVxD_V86, CVxD_PM
VxD_LOCKED_CODE_SEG
```

这是一个标准的VxD的说明部分，VMM通过VxD程序的设备描述块（DDB）来获取VxD的有关的信息。一个设备描述块是一个结构，它包含了许多关于VxD的重要信息，查阅DDB表可以知道。

SthVxD为VxD的名称，5为主版本号，0为副版本号，CVxD_Control为指向VxD的消息处理函数的指针，CVxD_DEVICE_ID为设备ID号。UNDEFINED_INIT_ORDER是初始化的序列，可以通过这个序列号让VMM来决定是装入时初始化还是随机地初始化或者是一定要在某事件前初始化，但一般的VxD是不必要设置这个参数的，它一般用在系统的某些固定的设备的VxD上。CVxD_V86为V86的处理函数，CVxD_PM为保护模式程序使用的API地址。

在上面这段代码中，我们还可以看到VxD_LOCKED_CODE_SEG这个语句，它是什么呢？

其实，它是vmm.inc中的宏语句，表示代码在内存中的一种存储方式，请参考介绍LE的文件模式的一节。

第3章 Windows运行机理

3.1 内核分析(10)

我们可以从SthVxD.c中看到CVxD_Dynamic_Init和CVxD_Dynamic_Exit这两个函数的定义。

```
ifdef       _VxD_SERVICES
extrn       _CVxD_Get_Version:near
endif

extrn       _CVxD_V86API@12:near
extrn       _CVxD_PMAPI@12:near
extrn   C   EnableHlt:  dword

BeginProc CVxD_Control
   Control_Dispatch SYS_DYNAMIC_DEVICE_INIT,
                        CVxD_Dynamic_Init, sCall
    Control_Dispatch SYS_DYNAMIC_DEVICE_EXIT,
                        CVxD_Dynamic_Exit, sCall
       Control_Dispatch W32_DEVICEIOCONTROL,
                            CVxD_W32_DeviceIOControl,
                            sCall, <ebp, ecx, ebx, edx, esi>
    clc
    ret
EndProc CVxD_Contro
```

这是一段消息处理函数。

当VxD初始化时，发生SYS_DYNAMIC_DEVICE_INIT的消息而转入CVxD_Dynamic_Init函数中进行处理。

W32_DEVICEIOCONTROL是宏来定义的，设备控制程序CVxD_W32_DeviceIOControl，<ebp, ecx, ebx, edx, esi>是调用函数时用来传递参数的寄存器的名字。CVxD_W32_DeviceIOControl是留给应用程序的接口函数。当VxD的应用运行后，它和VxD进行数据交换就通过此函数来实现。因为它是用汇编编写的，所以，所有的参数都使用寄存器来传递。

- 接下来可以看见三个函数

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;      内核空闲(IDLE)时调用此函数
;;
```

```
;;      使用HLT指令可以降低CPU的温度
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BeginProc _IDLEHandleProc
mov eax,[EnableHlt]
test    eax,eax
jnz NEXT       ;查看EnableHlt是否为真，当EnableHlt为真时
               ;（即允许用降温功能）转入NEXT
               ;当EnableHlt为假时，清去零标志位，返回系统中。
stc
ret
NEXT:
sti               ;必须打开中断， 如果没有打开中断，CPU就不能响应中断，
                  ;就会死机。当中断打开时，当运行HLT指令后，
                  ;机器一直停止直到外部有一个中断。
                  ;例如敲键或移动鼠标时，当中断处理写成后，
                  ;会从下一条指令开始执行,就不会死在HLT指令处。
hlt               ;CPU停机,停机就是使得CPU不工作，
;当CPU不工作时，CPU的功耗就很小，所以能降低温度
stc                  ;当有事件产生时，清去零标志位，
;返回系统中去处理其程序;
ret
EndProc   _IDLEHandleProc
```

内核空闲（IDLE）时，调用_IDLEHandleProc函数，通过HLT指令达到降温的目的。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  安装内核空闲(IDLE)时调用的函数
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BeginProc _InstallIDLEProc
push    esi
lea esi,[_IDLEHandleProc]
VMMCall Call_When_Idle
pop esi
mov eax,0
setnc   al
ret
```

```
EndProc   _InstallIDLEProc
```

_InstallIDLEProc函数是在SthVxD.c的CVxD_Dynamic_Init（void）的函数中被调用的，当它被装入后，VMM空闲时，就自动地调用函数_IDLEHandleProc。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  取消安装内核空闲(IDLE)时调用的函数
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
BeginProc _UnInstallIDLEProc
push    esi
lea esi,[_IDLEHandleProc]
VMMCall Cancel_Call_When_Idle
pop esi
mov eax,0
setnc   al
ret
EndProc   _UnInstallIDLEProc
```

_UnInstallIDLEProc函数和_InstallIDLEProc函数一样，也是在SthVxD.c中被CVxD_Dynamic_Exit(void)函数调用的。它的作用是去掉VMM注册的空闲函数。

```
;************************************************
;               V86模式调用的入口
;
;   把调用转变成对C函数的调用, 以便于开发功能强大的处理能力
;
;************************************************
BeginProc CVxD_V86
    scall   CVxD_V86API,
       <[ebp].Client_EAX, [ebp].Client_EBX, [ebp].Client_ECX>
    mov    [ebp].Client_EAX,eax    ; put return code
    ret
EndProc   CVxD_V86
;************************************************
;           保护模式调用的入口
;
;   把调用转变成对C函数的调用, 以便于开发功能强大的处理能力
;************************************************
```

```
BeginProc CVxD_PM
    scall   CVxD_PMAPI,
       <[ebp].Client_EAX, [ebp].Client_EBX, [ebp].Client_ECX>
    mov     [ebp].Client_EAX,eax
    ret
EndProc   CVxD_PM

VxD_LOCKED_CODE_ENDS
```

以上CVxD_V86和CVxD_PM两个函数都在SthVxD.c中定义。它们是V86模式和保护模式调用的入口，其实，它们的代码分别为CVxD_V86API和CVxD_PMAPI。此处只有函数定义，没有具体的实现功能。

VxD不光可以给WIN32位程序调用，WIN16和DOS程序都可以调用VxD提供的功能，它们都通过中断2FH来调用，并设置相应的参数。保护模式与此也是一样的，不过要用到VxD的号码。

```
;************************************************;
; 不是动态的VxD时，在Window启动时会被实模式调用
;
;************************************************
VxD_REAL_INIT_SEG

BeginProc CVxD_Real_Init

    xor    bx, bx
    xor    si, si
    xor    edx, edx
    mov   ax, Device_Load_Ok
    ret

EndProc CVxD_Real_Init

VxD_REAL_INIT_ENDS

END CVxD_Real_Init
```

以上是一段实模式的初始化调用的函数，主要是用在Windows 9x启动时。当Windows 9x启动时，就会调用到这个函数。这是一个实模式代码，可以看到代码段和寄存器是不同的。这不过是一个框架程序，它会将对汇编的调用全部转变成对C的调用。这样就很方便开发程序。

第3章 Windows运行机理

3.1 内核分析(11)

在相应的SthVxD.c的代码中，可以看见其实现方法：

```
int _stdcall CVxD_V86API(unsigned int function,
            unsigned int parm1,
            unsigned int parm2)
{
    int retcode;

    switch (function)
        {
        case CVxD_V86_FUNCTION1:
            retcode = V86Func1(parm1);
            break;

        case CVxD_V86_FUNCTION2:
            retcode = V86Func2(parm1, parm2);
            break;

        default:
            retcode = FALSE;
            break;
        }

        return (retcode);
}
```

function参数是传入的功能号。

parm1和parm2是传入的两个参数。

程序会根据功能号转入对应的段运行，很像GUI中的消息处理部分。在此处，V86Func1(parm1)和V86Func2(parm1, parm2)没有功能代码，不过是向大家展示实现框架。

我们还可看到这段的说明，因为Windows 9x中为了省内存，段内的有些代码和数据在系统启动完以后会释放，也就是说，启动完成后，这部分代码就没有了。

在汇编段中还定义了DeviceIOControl函数。DeviceIOControl这个函数只定义了一些必要传递的一批参数，例如调用的服务号。具体的实现都是在应用程序中完成的，步骤如下：

应用程序->DeviceIoControl->内核->由汇编调用-> CVxD_W32_ DeviceIOControl

```
DWORD _stdcall CVxD_W32_DeviceIOControl(    CRS * lpClient,
                                    DWORD  dwService,
                                    DWORD  dwDDB,
                                    DWORD  hDevice,
                          LPDIOC lpDIOCParms)
{
DWORD dwRetVal = 0;

// DIOC_OPEN is sent when VxD is loaded w/ CreateFile
//  (this happens just after SYS_DYNAMIC_INIT)
if( dwService == DIOC_OPEN ){
           //Out_Debug_String("SthVxD: WIN32 DEVIOCTL
           //supported here!\n\r");
           // Must return 0 to tell WIN32 that this VxD
           //supports DEVIOCTL
             dwRetVal = 0;
             }
// DIOC_CLOSEHANDLE is sent when VxD is unloaded  w/  CloseHandle
// (this happens just before SYS_DYNAMIC_EXIT)
else    if( dwService == DIOC_CLOSEHANDLE ){
                    // Dispatch to cleanup proc
                    dwRetVal = CVxD_CleanUp();
                    }
else    if( dwService > MAX_CVxD_W32_API )
           {
           // Returning a positive value will cause the
           //WIN32 DeviceIOControl
           // call to return FALSE, the error code can then
           //be retrieved
             // via the WIN32 GetLastError
             dwRetVal = ERROR_NOT_SUPPORTED;
             }
        else    {
             //调用功能函数功能号从1开始
           dwRetVal=(CVxD_W32_Proc[dwService-1])
                     (lpClient,dwDDB,hDevice,lpDIOCParms);
             }
```

```
    return(dwRetVal);
}
```

通常，为了方便，一般的VxD的做法是把函数的指针放在某个结构中，然后通过功能号直接去调用这个函数就行了。

```
//DeviceIoControl功能号表
DWORD (_stdcall *CVxD_W32_Proc[])(CRS *,DWORD,DWORD,LPDIOC)=
{
    0,                  //1(未使用)
    0,                  //2(未使用)
    CVxD_W32_EnableHalt //3(开关降温功能)
};
```

从CVxD_W32_Proc这个函数代码可以看到，功能一、二是没有用的，功能三是用来降温的，功能三调用CVxD_W32_EnableHalt函数，这个函数用来开关降温功能。

```
////////////////////////////////////////////////////////////
//
//          开关降温功能(功能号3)
//
////////////////////////////////////////////////////////////
DWORD _stdcall CVxD_W32_EnableHalt(CRS * lpClient,DWORD dwDDB,
                        DWORD hDevice, LPDIOC lpDIOCParms)
{
    LPDWORD lpEnablePtr;
    DWORD   OldEnable;

    OldEnable   =EnableHlt;
    lpEnablePtr=(LPDWORD)lpDIOCParms->lpvOutBuffer;
    if(lpEnablePtr) EnableHlt=*lpEnablePtr;

    return(OldEnable);
}
```

什么地方调用CVxD_W32_EnableHalt函数呢？我们可以从上面的CVxD_W32_DeviceIOControl函数中看到以下的一段语句：

```
//调用功能函数功能号从1开始
dwRetVal=(CVxD_W32_Proc[dwService-1])(lpClient,dwDDB,hDevice,lpDIOCParms);
```

其中，dwService是传入的服务号，lpClient，dwDDB，hDevice，lpDIOCParms是功能传入的固定的参数。

第3章 Windows运行机理

3.1 内核分析(12)

我们可从CpuCool.c程序中看到VxD的装入内存、安装功能和设置VxD功能号3的全过程。在代码中都有详细的注释。其实，VxD的装入方法与一个通常的文件一样，也是通过Win API的函数CreateFile来完成，大家一定不要被Create这个词误解了，其实，CreateFile可以用来打开和创建新文件。

```
int     APIENTRY WinMain( HANDLE  hInstance,
                     HANDLE hPrevInstance,
                        LPSTR   lpszCmdLine,int nCmdShow)
{
    static char szAppName[]="CoolCPU";
    char            Buffer[64];
    HMENU       hPopupMenu;
    WNDCLASS    wndclass;
    HWND        hwnd;
    MSG     msg;
   int     Ret;

    hResInstance=hInstance;

    //是否是中文
   LoadString(hResInstance,IDS_CODEPAGE,Text,sizeof(Text));
    Ret=StrToInt(Text);

if(GetSystemMetrics(SM_DBCSENABLED) && GetACP()==(DWORD)Ret)
   China=1;
else
   China=0;

//取得操作系统的版本
WinNT=GetVersion();
WinNT=(WinNT&0x80000000)==0 ? 1:0;

//如果不是WinNT就打开VxD，因为VxD只能在WIN9X下工作
if(WinNT==0)
```

```
    {//打开VxD
 lstrcpy(SthVxDName,"\\\\.\\");
 GetStartPath(Buffer,sizeof(Buffer));
 //VxD只认短路径

    GetShortPathName(Buffer,&SthVxDName[4],
sizeof(SthVxDName));
lstrcat(SthVxDName,"SthVxD.VxD");

    //尝试打开默认的VxD
hCVxD = CreateFile("\\\\.\\SthVxD", 0,0,0, CREATE_NEW,
           FILE_FLAG_DELETE_ON_CLOSE, 0);
if(hCVxD==INVALID_HANDLE_VALUE)
    {
    //直接打开全路径的VxD
     hCVxD = CreateFile(SthVxDName, 0,0,0, CREATE_NEW,
                     FILE_FLAG_DELETE_ON_CLOSE, 0);
    }
if(hCVxD==INVALID_HANDLE_VALUE)
    {//直接打开默认的.VxD
     hCVxD = CreateFile("\\\\.\\SthVxD.VxD", 0,0,0, CREATE_NEW,
                  FILE_FLAG_DELETE_ON_CLOSE, 0);
     }

//成功否
if(hCVxD!=INVALID_HANDLE_VALUE)
    {
     EnableHlt=TRUE;
     //设置VxD功能号3
     DeviceIoControl(hCVxD,3,(LPVOID)NULL,0,
               (LPVOID)&EnableHlt,sizeof(EnableHlt),
               &cbBytesReturned,NULL);
     }
else    {
     if(China)
       {
       LoadString(hResInstance,IDS_MAYBEERROR,
              Cap,sizeof(Cap));
         LoadString(hResInstance,IDS_NOTLOADVxD,
```

```
                        Text,sizeof(Text));
                MessageBox(NULL,Text,Cap,MB_OK);
    }
else    MessageBox(NULL,"Can't load STHVxD.VxD,STHVCD maybe failure !",
    "Maybe Error",MB_OK);
}
        }

hIcon=LoadIcon(hResInstance,MAKEINTRESOURCE(IDI_ICON));

if(China) hPopupMenu=LoadMenu(hResInstance,MAKEINTRESOURCE(IDR_CMENU));
else    hPopupMenu=LoadMenu(hResInstance,MAKEINTRESOURCE(IDR_MENU));

hPopMenu=GetSubMenu(hPopupMenu,0);

if(!hPrevInstance)
    {
    wndclass.style      =CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc    =(WNDPROC)WndProc;
    wndclass.cbClsExtra =0;
    wndclass.cbWndExtra =0;
    wndclass.hInstance      =hInstance;
    wndclass.hIcon      =hIcon;
    wndclass.hCursor        =LoadCursor(NULL,IDC_ARROW);
    wndclass.hbrBackground  =(HBRUSH)COLOR_WINDOW;
    wndclass.lpszMenuName   =NULL;
    wndclass.lpszClassName  =szAppName;

    RegisterClass(&wndclass);
    }

MainWin=hwnd=CreateWindow(  szAppName,"CoolCPU",
                      WS_OVERLAPPED|WS_CAPTION|WS_SYSMENU|
                      WS_MINIMIZEBOX,
                     0,0,
                     240,160,
                     NULL,NULL,hResInstance,NULL);

ShowWindow(hwnd,SW_HIDE);
```

```
UpdateWindow(hwnd);

AddShellIcon();

while(GetMessage(&msg,NULL,0,0))
    {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }

DelShellIcon();

DestroyIcon(hIcon);
DestroyMenu(hPopupMenu);

//关闭降温
        if(WinNT==0)
           {
            if(hCVxD!=INVALID_HANDLE_VALUE)
               {
                EnableHlt=0;
                //设置VxD功能号3
                DeviceIoControl(hCVxD,3,(LPVOID)NULL,0,
                       (LPVOID)&EnableHlt, sizeof(EnableHlt),
                    &cbBytesReturned,NULL);
                }
            }
        //关闭VxD
        if( hCVxD != INVALID_HANDLE_VALUE )
            CloseHandle(hCVxD);

        return msg.wParam;
    }
```

第3章 Windows运行机理

3.1 内核分析(13)

在程序的运行中，可以看见，当程序启动的时候，并没有出现窗口，而是在Windows的任务栏的右下角出现了一个下雨一样的小图标，这叫做托盘方法。实现起来也是很简单，很多资料中都介绍了，这里就不赘述。

```
#define WM_ICONCALLBACK (WM_USER+0x1234)

////////////////////////////////////////////////
//
//          添加任务条Icon
//
////////////////////////////////////////////////
int AddShellIcon(void)
{
    LPBYTE          lpszTip;
    NOTIFYICONDATA  tnid;
    BOOL                res;

    if(China)
       {
        LoadString(hResInstance,IDS_COOLCPUNAME,
              Text,sizeof(Text));
       lpszTip=Text;
       }
    else    lpszTip="CoolCPU";

    tnid.cbSize         = sizeof(NOTIFYICONDATA);
    tnid.hWnd               = MainWin;
    tnid.uID                = 1;
    tnid.uFlags         = NIF_MESSAGE | NIF_ICON | NIF_TIP;
    tnid.uCallbackMessage   = WM_ICONCALLBACK;
    tnid.hIcon          = hIcon;
   lstrcpyn(tnid.szTip,lpszTip,sizeof(tnid.szTip));
```

```
    res = Shell_NotifyIcon(NIM_ADD, &tnid);
    return res;
}
////////////////////////////////////////////////
//
//          删除任务条Icon
//
////////////////////////////////////////////////
int DelShellIcon(void)
{
    NOTIFYICONDATA  tnid;
    BOOL                res;

    tnid.cbSize = sizeof(NOTIFYICONDATA);
    tnid.hWnd       = MainWin;
    tnid.uID        = 1;

    res = Shell_NotifyIcon(NIM_DELETE, &tnid);
    return res;
}
////////////////////////////////////////////////
//
//          窗口处理函数
//
////////////////////////////////////////////////
long    APIENTRY WndProc(   HWND hwnd,UINT message,UINT wParam,
                  LONG lParam)
{
    POINT       ptCurrent;
    PAINTSTRUCT ps;

        switch(message)
        {
        case WM_PAINT:
            BeginPaint(hwnd,&ps);
            EndPaint(hwnd,&ps);
            return 0;
        case WM_ICONCALLBACK:   //任务条Icon回调消息
            switch(lParam)
```

```
                {
                case WM_LBUTTONDBLCLK:
                case WM_LBUTTONDOWN:
                case WM_RBUTTONDOWN:
                    GetCursorPos(&ptCurrent);
                    SetForegroundWindow(hwnd);
                    //显示菜单
                    TrackPopupMenu( hPopMenu,
                            TPM_RIGHTBUTTON,
                            ptCurrent.x,
                            ptCurrent.y,
                            0,
                            hwnd,
                            NULL);
                    break;
                }
            return 0;
        case WM_INITMENUPOPUP:
            if(lParam==0)
                {
                if(WinNT==0 && hCVxD!=INVALID_HANDLE_VALUE)
                    {
                    if(EnableHlt)
                             CheckMenuItem((HMENU)wParam,
                        ID_COOLCPU,
                        MF_BYCOMMAND|MF_CHECKED);
                    else
                     CheckMenuItem((HMENU)wParam,
                        ID_COOLCPU,
                        MF_BYCOMMAND|MF_UNCHECKED);
                    }
                else
                  EnableMenuItem((HMENU)wParam,ID_COOLCPU,
                              MF_BYCOMMAND|MF_GRAYED);
                }
            return  0;
        case    WM_COMMAND:
            switch(wParam)
                {
```

```
                case BN_CLICKED:
                    break;

                case ID_EXIT:
                    PostMessage(hwnd,WM_CLOSE,0,0);
                    break;
                    //Cool Cpu
                    case ID_COOLCPU:
                        if(WinNT==0)
                            {
                            if(hCVxD!=INVALID_HANDLE_VALUE)
                                {
                                EnableHlt^=1;
                            DeviceIoControl(hCVxD,3,
                                (LPVOID)NULL,0,
                                (LPVOID)&EnableHlt,
                                sizeof(EnableHlt),
                                &cbBytesReturned,NULL);
                                }
                            }
                        break;
                }
            break;
        case WM_DESTROY:
              PostQuitMessage(0);
            break;
        case WM_CLOSE:
            break;
        }

    return (DefWindowProc(hwnd,message,wParam,lParam));
}
```

我们可以通过Windows的系统资源监视器看到，当降温程序打开时，CPU的占用率会马上提高，当降温程序关闭时，CPU的占用率又马上恢复原值。这是因为系统资源监视器也是通过空闲时调用的方法实现的，所以当降温程序工作时，CPU就会暂停了，就好像是占用了很多的资源。

第3章 Windows运行机理

3.2 消息的运行方式(1)

3.2.1 认识消息

我们首先从16位的Windows来认识消息。在16位时代，Windows的整个内核是32位的、分时的、抢占的。可以从Windows的内核模型得知，有两种VM，一种是 SYSTEM VM,另一种是DOS的VM。一个系统中可以运行很多的DOS窗口，因为在16位的时代，能运行DOS的程序是很重要的，所以在当时，Windows的主要任务之一，就是能同时运行很多DOS窗口。Windows的内核实现上用了很多微内核，而微内核的工作很多都是靠消息来完成的。

在系统内部我们可以看到Windows的消息内核原理，消息结构如图3.4所示。

可以看到，所有的功能还是通过中断来实现的，只不过是在保护模式内调用中断。在Windows的16位时代，大部分的工作都是基于各种中断的基础上，而且应用程序可以直接调用DOS中断。其实，Windows的内核和DOS是平等的，包括设备驱动，也是和DOS应用程序是同一级别的。这样，这个系统的VM中的DLL就直接调用中断来进行管理，当调用中断时，就会用VxD或.386文件，对中断或IO进行截取，来模拟直接操作硬件的工作。其实，它的驱动也是一个DLL，和USER.DLL、GDI.DLL是一样的。

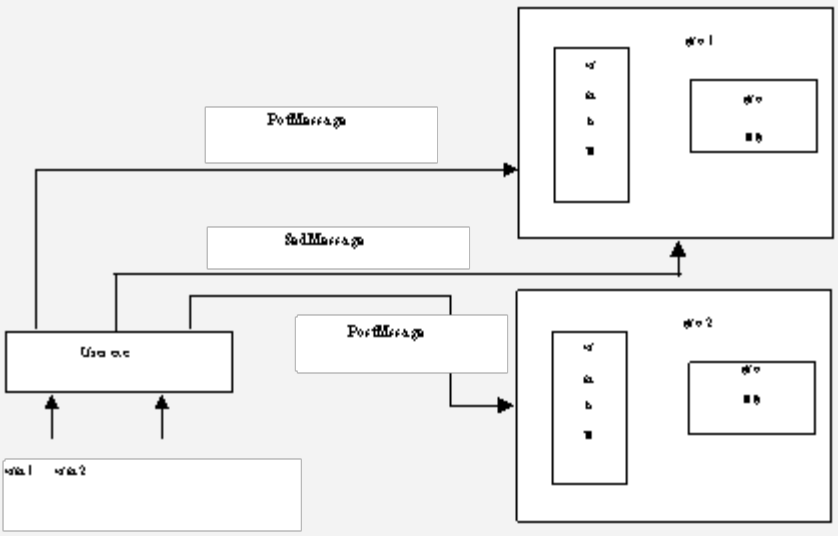


图3.4 消息结构

```
//一直等待消息，直到有消息发生时
while (GetMessage(&msg, NULL, 0, 0))
        {
           //翻译消息
        TranslateMessage(&msg);
```

```
        …
    //分配消息到对应的窗口
    DispatchMessage(&msg);
     }
```

通过以上代码可以看出，在Windows 16位时代中，实现消息调度的函数是GetMessage。还有一种函数是PeekMessage，它会从消息中取出一条消息，但它和GetMessage不同的是，当消息队列中有消息时，它会返回函数；没有消息时就会返回0。而GetMessage就会一直停止在这些函数上，直到有消息为止。

到Windows 32位时，消息的运行机理就不相同了。从内核中可以看出，有一个Win32的VxD，把DOS的抢占分时都放在这个VM中完成，系统VM就进一步和系统底层融合。然后在这个基础上分出时间片。这样，每个应用程序就自己有自己的消息队列。

所有的消息队列看上去是放在USER32的模块内，但每个应用程序自己有一个USER32，因为每个应用程序在内存内都是从4000000B（也就是4MB的位置开始的），这样，每个GetMessage和PeekMessage都在处理事件。实际上，每个GetMessage就会成为一个WaitsingleMessage,当有事件来后，就直接进行处理，也不用做什么调度。因为自己完成自己的消息处理，每个程序都是独立的，所以要用底层内核来实现页面的切换。它某一程序切入时，其他程序就会被切出。当切换出去时，整个消息队列也就被切换出去了。所以，整个消息的处理就很简单了。

Windows 32位时的消息机理如图3.5所示。

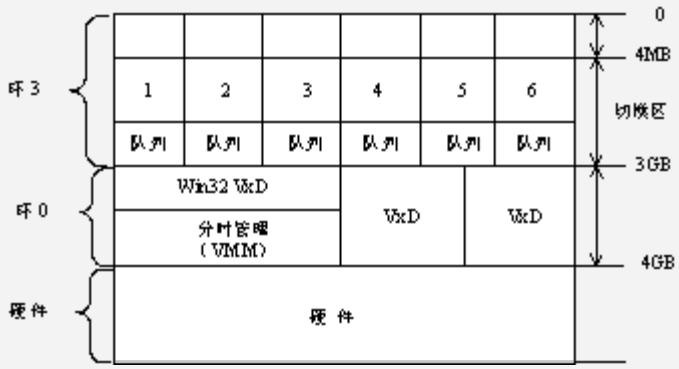


图3.5 消息的机理

第3章 Windows运行机理

3.2 消息的运行方式(2)

3.2.2 Windows系统中消息的运作方式

1. 消息循环

在Windows程序的经典设计程序中，可以看到如下程序：

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC     hdc;
    TCHAR szHello[MAX_LOADSTRING];
    LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);

    switch (message)
    {
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
                {
                case IDM_ABOUT:
                    DialogBox(hInst,(LPCTSTR) IDD_ABOUTBOX,
                        hWnd, (DLGPROC)About);
                  break;
                case IDM_EXIT:
                     DestroyWindow(hWnd);
                     break;
                default:
                   return DefWindowProc(hWnd, message,
                               wParam, lParam);
                  }
```

```
                break;
            case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                    // TODO: Add any drawing code here...
                    RECT rt;
                    GetClientRect(hWnd, &rt);
                    DrawText(hdc, szHello, strlen(szHello),
                            &rt, DT_CENTER);
                    EndPaint(hWnd, &ps);
                    break;
                case WM_DESTROY:
                    PostQuitMessage(0);
                    break;
                default:
                     return DefWindowProc(hWnd, message, wParam,
                                    lParam);
         }
       return 0;
   }
```

首先有一个GetMessage，只要这个消息不为0，就可以一直循环，当有消息来时，就通过跳转到消息处理函数来完成相应的功能。这样做有一个好处，例如，在DOS中，按键、鼠标消息都是放在键盘缓存区和鼠标缓存区中的，现在就可以直接将输入放入消息队列中。消息的结构如下：

```
typedef struct tagMSG {     // msg
    HWND      hwnd;              //发送给的对应窗口句柄
    UINT      message;             //消息的类型
    WPARAM   wParam;          //消息传送第一个32位参数，
    LPARAM   lParam;          //消息传送第二个32位参数
    DWORD     time;            //发送消息的时间
    POINT     pt;              //发送消息时鼠标所在的位置
} MSG;
```

从以上结构中可以看到，每个消息都对应着一个窗口。USER模块是管理窗口的，一般每个窗口自己有一个消息队列。当键盘或鼠标有消息时，就会发给激活的窗口，当在程序设计中用SendMessage来发送消息时，就会明确指定窗口句柄，当运行此函数后，就会把消息放到此窗口的消息队列中。

所有的程序都通过系统消息队列来调用USER的DLL来完成工作，所以，这个DLL就有机会轮循，来查看什么程序有消息。如果某程序有消息，就会去调用这个程序的窗口函数，而这个窗口在生成时，必须注册在这个窗口类中。在窗口类中就有窗口的处理消息函数的地址指针。当程序有消息时，USER就调用这个函数的地址，去完成消息处理。

在键盘或鼠标这类设备中，消息一般只是发给当前激活的窗口，当然其他窗口也可以得到消息，可以通过程序直接发送。还有一些情况也可以得到消息，例如时钟消息TIMER是底层驱动的，当TIMER产生一个消息时，它会查找当前窗口中定义了时钟消息的时间是否来到，当时间到了，就会在对应的窗口函数中放入一时间消息事件。

其实，明白了消息的处理过程，消息也就很简单了。消息不过是定义一个结构，定义一堆ID，在程序运行中调用switch和case去完成相应的功能。

2. 消息处理函数

有两种消息的发送函数，一种是立即发送消息，另一种是队列调用。

```
LRESULT SendMessage(
          HWND        Hwnd
       UINT    uMsg,
       WPARAM  wParam,
       LPARAM  lParam );
   LRESULT PostMessage(
        HWND        Hwnd
       UINT    uMsg,
       WPARAM     wParam,
       LPARAM  lParam );
```

这两种函数的接口参数基本上是一样的。

- HWnd：将要发送给消息的对应的窗口句柄。它实际指向消息发给谁。
- UMsg：消息的类型，说明被发送的消息是什么消息。
- WParam：第一个32位的参数。
- lParam：第二个32位的参数。

可不能小看这两个参数，它们可是很有用的。

- SendMessage：当用它向一个窗口（也可以是本身窗口）发送消息时，它不会把消息放入消息队列中，而是直接发送给窗口。窗口接到消息后就立刻处理，处理完成后，把结果作为返回值传送回来。这样的处理过程就像是操作函数一样。
- PostMessage：当用它向一个窗口（也可以是本身窗口）发送消息时，它把消息放入消息队列中，自己什么也不干就会返回，到底消息什么时候处理，有没有被处理它是不知道的。

第3章 Windows运行机理

3.2 消息的运行方式(3)

3.2.3 消息处理过程实例

我们已经对消息有了些了解，那到底消息是什么呢？其实，消息不过是定义了一个结构（在微软中定义的是MSG结构，自己也可以定义不同的结构），然后定义一堆ID号，例如：

```
#define  WM_MSG01    0X0001
…..
#define  WM_MSG**    0X*****
```

调到函数中，用swicth 和case语句对每一种ID进行相应的处理。

在Windows的编程中，有一个很经典的程序“Hello World”。我们也用这个最简单的程序来说消息的处理过程。这个程序可以直接在VC中用向导生成。首先，任何一个Windows程序都是从WinMain开始的。

```
int APIENTRY WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR        lpCmdLine,
int           nCmdShow)
{
    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_AA, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
        {
        return FALSE;
       }

   hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_AA);
```

```
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
        {
       if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    return msg.wParam;
}
```

大家可能对这段代码已很熟悉了，在以前用API进行Windows编程时，几乎所有的程序都会去套用这个框架。

从一个WinMain中注册一个窗口，其他再用GetMessage取得窗口的消息，翻译后分给对应的窗口。

其实，很多程序可以完全不用注册窗口。它只要做一些事件，当有事件来时，就处理相应的事件。例如，以下就是一个Windows程序，其中没有用到任何消息循环，只是在运行中弹出一个对话框：

```
//------------------------------------------------
    //   HelloMsg.c -- Displays "Hello, Windows 98!" in a
    //message box
//   ------------------------------------------------

#include <windows.h>

int WINAPI WinMain (    HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                          PSTR szCmdLine, int iCmdShow)
{
     MessageBox (NULL, TEXT ("Hello, Windows 98!"),
          TEXT ("HelloMsg"), 0) ;

     return 0 ;
}
```

可以看到，这个WIN32的程序就没用到微软的框架，它完全是自己做自己的事，直到被中止。

实际上，很多应用可以不用窗口，例如，Windows NT的服务程序就不需要窗口。

第3章 Windows运行机理

3.3 GDI的结构和组成(1)

3.3.1 GDI的组成

GDI有一些基本的函数。GDI内只有和HDC有关的几个做图的函数，更多的功能其实是在USER32内实现的。所以我们说的GUI是GDI和USER接合起来的。

GDI在Windows内只是划一些点、线。点线填充时，USER不但要管理窗口和字体等许多资源（大多数GDI函数都和HDC有关），USER还要管理很多窗口，并且管理窗口的裁减和输出。在每个程序中，它所管理的屏幕就好像USER完全占用了所有的窗口，和其他程序在显示屏上不冲突。在这个程序内部，USER好像自己拥有一个显示屏一样。在HDC中是通过裁剪来对窗口进行管理的。

HDC是一个很大的结构，一般系统内没有很多的HDC。在Windows 3.1中，只有5个系统的HDC，在Windows 98下又扩充了几个，它们组成了一个HDC的池。当系统要使用HDC的资源时，系统会随机地从这5个中选取其中一个没有被占用的分配给用户来使用，所以，当用户使用HDC后，一定要释放HDC资源，要不就有可能导致系统的资源不足。这一点在Windows 3.1中很明显，但Windows 95可以自己用CreateDC动态地创建，当完成使用后，用DeleteDC函数来删除HDC的系统。

下面是DC的数据结构，这个结构微软是保密的，我是从开放源码中获得的，大家可以研究一下。

```
typedef struct tagDC
{
    GDIOBJHDR        header;
    HDC            hSelf;                /* Handle to this DC */
    const struct    tagDC_FUNCS *funcs; /* DC function table */
    PHYSDEV      physDev;                 /* Physical device */
   /*(driver - specific) */
    INT            saveLevel;
    DWORD        dwHookData;
    FARPROC16   hookProc;            /* the original SEGPTR  */
    DCHOOKPROC  hookThunk;          /* and the thunk to call it */

    INT            wndOrgX;                  /* Window origin */
    INT            wndOrgY;
    INT            wndExtX;                  /* Window extent */
    INT            wndExtY;
    INT        vportOrgX;      /* Viewport origin */
    INT        vportOrgY;
```

```
    INT         vportExtX;      /* Viewport extent */
    INT         vportExtY;

    int         flags;
    HRGN        hClipRgn;       /* Clip region (may be 0) */
    HRGN        hVisRgn;        /* Visible region (must never be 0) */
    HRGN        hGCClipRgn;     /* GC clip region(ClipRgn AND VisRgn) */
    HPEN        hPen;
    HBRUSH       hBrush;
    HFONT        hFont;
    HBITMAP   hBitmap;
    HANDLE       hDevice;
    HPALETTE    hPalette;

    GdiFont   gdiFont;
    GdiPath   path;

    WORD        ROPmode;
    WORD        polyFillMode;
    WORD        stretchBltMode;
    WORD        relAbsMode;
    WORD        backgroundMode;
    COLORREF  backgroundColor;
    COLORREF  textColor;
        short      brushOrgX;
    short      brushOrgY;
    WORD        textAlign;     /* Text alignment from
                              SetTextAlign() */
    short      charExtra;     /* Spacing from
                                 SetTextCharacterExtra()*/
    short      breakTotalExtra;   /* Total extra space
                              for justification */
    short      breakCount;         /* Break char. count */
    short      breakExtra;         /* breakTotalExtra breakCount */
    short      breakRem;       /* breakTotalExtra % breakCount */

    RECT        totalExtent;
    BYTE        bitsPerPixel;
```

```
    INT         MapMode;
    INT         GraphicsMode;         /* Graphics mode */
    ABORTPROC     pAbortProc;     /* AbortProc for Printing */
    ABORTPROC16   pAbortProc16;
    INT         CursPosX;             /* Current position */
    INT       CursPosY;
    INT         ArcDirection;
    /* World - to - window transformation */
    XFORM         xformWorld2Wnd;
    /* World - to - viewport transformation */
     XFORM          xformWorld2Vport;
    /* Inverse of the above transformation */
    XFORM         xformVport2World;
    /* Is xformVport2World valid? */
    BOOL          vport2WorldValid;
} DC;
```

所以可以看到，以上这些结构和与之相关的函数就是一个“类”。在MFC中就是CDC类。CDC也就是把所有的与DC有关的函数进行了封装。

在Windows的应用程序中，当创建一个窗口时，如果用OwnDC属性，就是使用窗口自己的DC。窗口创建时，自己创建和管理自己的静态DC，这样就不使用系统的资源。

在Windows 95中，HDC的线、字体、刷子都是一种共享的属性。也就是说，两个应用程序中，当一个程序改变了HDC的属性，例如背景颜色时，另一个程序的背景颜色也会发生改变，这就是为什么使用HDC的资源前，一定要保存原始的值，当用完后就立刻恢复的原因。

但在Windows NT中就不是这样了。此时，每一个DC都是私有的，所以很多程序在Windows 95中能正常运行，但在Windows NT中就不能正常运行了。

第3章 Windows运行机理

3.3 GDI的结构和组成(2)

3.3.2 GDI和DirectDraw的关系

屏幕上的显示在内存中是以下这样的结构。

当向显示缓存区中写入数据时，就会显示相应的图像。DirectDraw的作用是创建，其实就是取得缓存区的地址，并且还能创建一个虚拟的缓存区内存。例如，A区域内存可以在主内存中创建一块叫offscreen的缓存区。

如果显示卡的内存比较大，如图3.6所示，有一块区域是映像到屏幕上的可见区域，还有的显存区域是屏幕上看不见的，这个区域被称为offscreen。也就是说，A区域为主显存，B区域也可以称为次显存。B区域实际上是被隐藏在后面的，就像DOS的游戏一样，先在次显存绘制好图形，当需要显示时，马上就可以切换过来。DirectDraw中有一个这种操作函数，这个命令如果能切换，就直接切换，如果不能直接地切换，就直接通过显示卡，从次缓存复制到主缓存，这种在显卡内的复制要比软件的memcpy命令快很多。

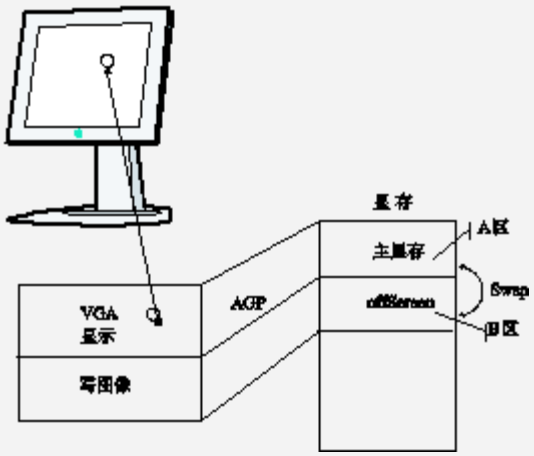


图3.6 显示内存图

把两个缓存区域结合起来用就可以做出高速的动画。例如，游戏可以先在次显存上绘制好下一帧画，一切换就能立刻显示出来。这样，画面的速度就很快了。

如图3.6所示，当向A地址写入一个数据时，对应的屏幕上就会出现一个点。

如果需要快速地显示图像，就不能用GDI，而应直接使用DirectDraw。它的缺点就是你必须对显示卡有充分的了解。显示卡可以分为很多种模式，如表3.2所示。

表3.2 显示卡的模式

颜色数	内存位数	颜色位数分配	字节数
16色	4位	Index索引	1/2
256色	8位	Index索引	1
15位色	16位	5,5,5	2
16位	16位	5,6,5	2
24位	24位	8,8,8	3
32位	32位	8,8,8,8	4

当用GDI显示一个图像时，就不用管显示卡是什么模式，只要设置好颜色，发送一个绘制命令即可。如果一个图是15位色，当把图形数据直接复制到对应的显存区域时，此时图形就被显示出来了。如果用GDI来显示图形时，它会将相应的色彩进行转换，把它转换成显示所支持的，这个过程需要用一点时间。

DirectDraw只是提供了一种方法，直接地向显存写入数据。在写数据进入显存比较慢时，可能会出现裂缝的图像显示。这是因为当上帧已显示完成了，此时次显存向主显存复制数据。

当把一个24位的图像用DirectDraw直接向显存中写入时是不正常的，但GDI就会没问题。

要想在16位模式中显示24位的图形，就需要通过程序进行转化。下面是转化的程序。

```
//24 位 R G B （8 8 8） 16位 R G B（6 5 6）
 void Convert24To16 (LPBYTE lpInDate,LPBYTE lpOutDate,
                  const int nSize)
{
    int i ;
    int nData;
    BYTE    R,G,B;
    for (i = 0; i < nSize; i++){
        nData = *((int *)lpInDate);
        R = nData >> 3;
        G = nData >> 10;
        B = nData >> 19;
        nData = B|(G<<5)|(B<<11);
        *((int *)lpOutDate) = nData;
        }
}
```

第3章 Windows运行机理

3.4 线程的机制(1)

3.4.1 线程的工作方式

线程是Windows 95的新特征，一个线程就是一个执行程序的事例。线程允许一个程序同时在多于一个以上的地方运行，这有些像多个CPU，每一个CPU执行程序的一部分。在单处理器系统中（Window 95只支持单处理器系统），只有同时处理时才出现线程。Windows 95系统中，线程之间切换CPU的间隔称为时间片（timeslicing）。因为硬件内部的计时器是以有规律的时间间隔通知操作系统的，所以操作系统可以选择不同的线程。另外，尽管16位的程序作为一个线程出现在系统线程表中，但只有Windows 32应用程序中能产生附加的线程。

一个线程被切换有两个原因，原因之一是本线程需要另一个线程先执行，此时，当前线程则把CPU让给另一个线程。另一个原因是当一个线程执行了足够长的时间后，需要把线程给另一个程序。Windows 95线程调度使用的是这样的一种算法，即把大部分时间给那些急需的线程。CPU时间间隔用硬件时钟中断，操作系统内部计时器中断处理调度决定另一个程序是否需要运行，如果运行，则切换到另一个线程上。Windows 95的时间片是20毫秒，也就是说，一秒钟内，理论上可在50个线程之间进行强制切换，但如果所有的线程都主动放弃CPU或等待系统，则切换的频率就会很高，每秒切换4、5千次也不奇怪。

每一个线程被分配到一个进程中，当操作系统产生一个新的进程时，也要设置一个初始线程。一个进程中的所有线程共享该进程的资源（下面要用“资源”一词来表示操作系统提供的内容），进程资源包括内存文本、文本柄和当前目录。

一般来讲，进程不交换，也不使用其他进程的资源。然而，一个进程中的多线程可能在进程资源的使用上发生冲突，这样，资源共享可能是一个混合物。例如，程序有一段代码改变了几个全局变量的代码序列，如果一个线程正好在这个序列中间被切换掉，那么下一个线程将作用这些全局变量，而且与状态不一致。成功地执行多线程程序要求你标记出一个进程中的所有的资源，这些资源需要由同步机进行监视，保证它们不会被不适宜的线程侵害。临界段（CriticalSection）和其他的线程同步机在下面进行讨论。

尽管线程共享进程资源，但每一个线程还有一定的资源提供自身，那么最重要的是栈吗？

不，每一个线程本身没有SS寄存器和相互依存，实际上，每一个线程在本身所在进程的地址空间内部有一个地址空间区。每一线程被分配的栈区隐含值是1MB，这个容量要么在可执行文件的.DEF文件栈中，要么在调用CreateThread产生线程规定一个非零栈区。Windows 95对每一个线程栈不使用MB，而是用“guardpage（保护页）”。

3.4.2 线程与GDI的冲突：死机的主要原因

很多人使用线程的时候，都喜欢在线程内画图。如果在线程内作画，程序就会很容易出错，而且还是

那种没有任何响应和提示的错误问题。

例如，如下是一个文件复制的程序，这个程序由两个线程组成，一个是复制文件的线程，另一个是显示文件复制进度的过程。当文件复制一部分后，进度条就向前移动一点。理论上，这个程序没什么问题。但是，这个程序有一个很大的隐患，即主程序也可能某一时刻要更新这个进度条。例如，进度被其他窗口挡住后或者整个窗口放大缩小时，整个窗口就要刷新，这时，线程的那个部分也要刷新它，操作系统也要刷新它。这样，三个部分都要去刷新它，程序就很容易死锁。程序运行界面如图3.7所示。

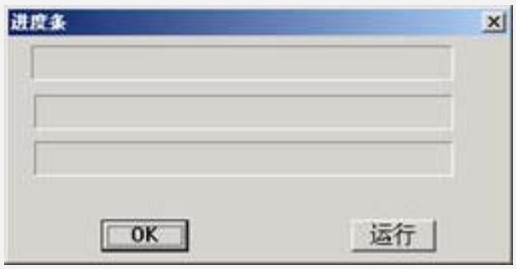


图3.7 程序运行界面图

这时会什么响应也没有了。这种问题在多线程中是很常见的。那怎么处理这个问题呢？

有一条原则，即程序中的线程一概不直接操作线程部分中的GDI。它只要发一个消息给主程序，让主程序来绘制图形，就不会出现任何的问题了。

发送消息的方法就是用PostMessage的函数。但一定不能用SendMessage。因为用PostMessage可以让主程序去调度绘图，而SendMesage会立即去绘制图形。所以在线程中要避免画图，因为当作画时，程序会取得一个DC，内存中的DC表示的是一块显存。DC代表的是一个窗口，因为一个程序得到此DC时，其他程序是不能再取得DC的。以后，如果继续再取，就会进入死锁的循环内。死锁结构如图3.8所示。

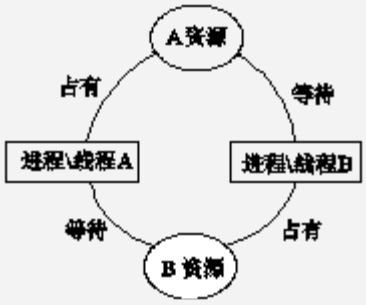


图 3.8 死锁的结构图

第3章 Windows运行机理

3.4 线程的机制(2)

3.4.3 线程的内存泄漏的主要原因

在很多参考书上，都说不要用CreateThread 创建线程、并用CloseHandle来关闭这个线程，因为这样做会导致内存泄漏，而应该用_beginthread来创建线程，_endthread来销毁线程。其实，真正的原因并非如此。看如下一段代码：

```
HANDLE CreateThread(
       // 线程安全属性
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
       // 堆栈大小
        DWORD dwStackSize,
       // 线程函数
         LPTHREAD_START_ROUTINE lpStartAddress,
          //线程参数
         LPVOID lpParameter,
       // 线程创建属性
        DWORD dwCreationFlags,
       // 线程ID
       LPDWORD lpThreadId
    );
```

线程中止运行后，线程对象仍然在系统中，必须通过CloseHandle函数来关闭该线程对象。CloseHandle函数的原型是：

```
BOOL CloseHandle(
    HANDLE hObject   // 对象句柄
);
```

CloseHandle可以关闭多种类型的对象，比如文件对象等，这里使用这个函数来关闭线程对象。调用时，hObject为待关闭的线程对象的句柄。

说用这种方法时内存在泄漏，其实不完全正确。那为什么会引起内存的泄漏呢？因为当线程的函数用到了C的标准库的时候，很容易导致冲突，所以在创建VC的工程时，系统提示是用单线程还是用多线程的库，因为在C的内部有很多的全局变量。例如，出错号、文件句柄等全局变量。

因为在C的库中有全局变量，这样用C的库时，如果程序中使用了标准的C的库时，就很容易导致运行不正常，会引起很多的冲突。所以，微软和Borland都对C的库进行了一些改进。但是这个改进的一个条件

就是，如果一个线程已经开始创建了，就应该创建一个结构来包含这些全局变量，接着把这些全局变量放入线程的上下文中和这个线程相关起来。这样，全局变量就会依赖于这个线程，不会引起冲突。

这样做就会有一个问题，什么时候这个线程开始创建呢？标准的Windows的API是不知道的，因为它是静态的库。这些库都是放在VC的LIB的目录内的，而线程函数是操作系统的函数。所以，VC和BC在创建线程时，都会用_beginThread来创建线程，再用_endThread来结束线程。这样，它们在创建线程的时候，就会知道什么时候创建了线程，并把全局变量放入某一结构中，让它和线程能关联起来。这样就不会发生冲突了。

很显然，要完成这个功能，首先需要分配结构表把全局变量包含起来。这个过程是在_beginThread时做的，而释放在_endTread内完成。

所以，当用_beginThread来创建，而用CloseHandle来关闭线程时，这时复制的全局结构就不会被释放了，这就有了内存的泄漏。这就是很多资料所说的内存泄漏问题的真正的原因。

其实，可以不用_beginThread和_endThread这一对函数。如果用CreateThread函数创建，用CloseHandle关闭，那么，与C有关的库就会用全局的，它们会引起冲突。所以，比较好的方法就是在线程内不用标准的C的库（可以使用Windows API的库函数）。这样就不会有什么问题，也就不会引起冲突。例如，字符串的操作函数、文件操作等。

当某个程序创建一个线程后，会产生一个线程的句柄，线程的句柄主要用来控制整个线程的运行，例如停止、挂起或设置线程的优先级等操作。一般来说，当线程启用后，就会用线程的CloseHandle来关闭线程。但在微软的示例程序中，有一个例子创建以后，就马上调用CloseHandle关闭线程的运行。这样做在Windows 98下没什么问题，但在Windows NT下，内核就会出现错误。这是为什么呢？

这是因为虽然线程有关的结构已经释放了，但线程还在运行中，所以程序就会出现错误。那怎么做才能确保正常运行呢？

其实，要正常运行，可以让线程完全结束以后，再调用CloseHandle来释放资源。

怎样知道线程完全结束呢？在Windows 的API中有一类等待线程的命令：

```
DWORD WaitForSingleObject(
  HANDLE hHandle,              // handle to object to wait for
  DWORD dwMilliseconds   // time-out interval in milliseconds
);


DWORD WaitForMultipleObjects(
  DWORD nCount,           // number of handles in the handle array
  CONST HANDLE *lpHandles,  // pointer to the object-handle array
  BOOL fWaitAll,                // wait flag
  DWORD dwMilliseconds   // time-out interval in milliseconds
);
```

可以用以上两函数，等待线程的结束。如果线程结束，函数就会返回。否则就一直等待，直到指定的时间结束。

还有一种线程根本不会退出，它一直运行着循环的线程。我们就要用中止线程的方法来结束线程的运

行，强制把它关闭。强制关闭后，再用CloseHandle来释放结构。

3.4.4 进程管理

Win16中，一个正在运行的程序被称为一个任务（task），16位的KERNEL把每一个Win16任务的信息保持在一个叫任务数据库（TDB）的段内，任务数据库的选择器被认为是一个HTASK，通过它可获知正在执行任务的API。

Windows 95中，针对32位程序做了什么改进呢？它把一个运行的程序称为一个进程而不是一个任务，每一个进程运行在自己的地址空间内。它们可以看到自己的内存和操作系统，而看不到其他的进程或其他进程的空间。使进程相互之间保持分离的基本原因是防止有问题的进程影响其他进程。

在Win32程序中，给WinMain的hPrevInstance参数总是为0。不管其他程序是否运行，一般情况下，一个进程自认为系统中只有该程序在运行。当然，如果你确实需要与另外的进程通信（或是去操作另一个进程），也是很容易的，这在编写代码之前就要考虑到。

每一个Windows 95进程在系统中被分配一个单一值。这个值为进程ID，一个程序可以通过GetCurrentProcessID函数获取自己的进程ID。这个进程ID非常近似于一个Win16 HTASK。NT中的进程ID分配给系统数据结构，因为典型的进程ID值是数字的，所以Windows 95中的进程ID的值比较高，并且是随机的。一个进程ID可以通过转换获取一个指示器，该指示器指向KERNEL32.DLL，用于跟踪进程的进程数据库结构。

第3章 Windows运行机理

3.4 线程的机制(5)

e_lfanew是相对实际PE头标的相对偏移量（或RVA）。要得到内存中一个指向PE头标的指针，只需将该域的值与映像的基相加：

```
//Ignoring typecasts and pointer conversion issues for clarity…
pNTHeader= dosHeader + dosHeader->e_lfanew;
```

其他字段的意义是和DOS头有关的字节，这里没有什么大的作用，就不做介绍了。

2. IMAGE_NT_HEADERS

主PE头标是一个IMAGE_NT_HEADERS类型的结构，该类型在WINNT.H中定义。

在内存中，Windows中把IMAGE_NT_HEADERS结构作为它内存中的模块数据库。在Windows中，每个被装入的EXE或DLL都用一个IMAGE_NT_HEADERS结构来说明。其结构如下：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Signature表示此文件所表示的类型，其意义定义如下：

```
#define IMAGE_DOS_SIGNATURE     0x4D5A        // MZ
#define IMAGE_OS2_SIGNATURE     0x4E45        // NE
#define IMAGE_OS2_SIGNATURE_LE  0x4C45        // LE
#define IMAGE_NT_SIGNATURE        0x50450000  // PE00
```

如果是PE格式，则Signature为PE\0\0（PE后跟两个0）。

3. IMAGE_FILE_HEADER

PE头标中紧随PE的WORD记号的是一个IMAGE_FILE_HEADER类型的结构，如下所示：

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
```

```
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

这个结构的域只包含了关于文件的最基本的信息。

Machine表示该文件运行所要求的CPU，有如下的CPU ID定义：

```
#define IMAGE_FILE_MACHINE_UNKNOWN           0
#define IMAGE_FILE_MACHINE_I386                0x014c
// Intel 386.
#define IMAGE_FILE_MACHINE_R3000             0x0162
// MIPS little-endian, 0x160 big-endian
#define IMAGE_FILE_MACHINE_R4000           0x0166
// MIPS little-endian
#define IMAGE_FILE_MACHINE_R10000        0x0168
// MIPS little-endian
#define IMAGE_FILE_MACHINE_WCEMIPSV2    0x0169
// MIPS little-endian WCE v2
#define IMAGE_FILE_MACHINE_ALPHA          0x0184
// Alpha_AXP
#define IMAGE_FILE_MACHINE_POWERPC       0x01F0
// IBM PowerPC Little-Endian
#define IMAGE_FILE_MACHINE_SH3             0x01a2
// SH3 little-endian
#define IMAGE_FILE_MACHINE_SH3E           0x01a4
// SH3E little-endian
#define IMAGE_FILE_MACHINE_SH4             0x01a6
// SH4 little-endian
#define IMAGE_FILE_MACHINE_ARM           0x01c0
// ARM Little-Endian
#define IMAGE_FILE_MACHINE_THUMB          0x01c2
#define IMAGE_FILE_MACHINE_IA64            0x0200
// Intel 64
#define IMAGE_FILE_MACHINE_MIPS16         0x0266
// MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU       0x0366
// MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU16     0x0466
```

```
// MIPS
#define IMAGE_FILE_MACHINE_ALPHA64       0x0284
// ALPHA64
#define IMAGE_FILE_MACHINE_AXP64
//IMAGE_FILE_MACHINE_ALPHA64
```

NumberOfSection表示在EXE或OBJ中的节数。这个很重要，因为它直接表示节表数组的大小。

TimeDateStamp表示连接器生成该文件的时间。该值是指从1969年12月31日下午4点整开始至文件生成时之间的秒数。

PointerToSymbolTable表示文件的COFF符号表的偏移量。该域只用在OBJ文件和带有COFF调试信息的PE文件中，此信息只在调试文件中有用。

NumberOfSymbols表示在COFF符号表中的符号数目，参见前一个域，此信息只在调试文件中有用。

SizeOfOptionalHeader表示紧跟该结构之后的一个可选头标的大小。在可执行文件中，它是紧随该结构的image_file_header结构的大小。这个值必须有效。

Characteristics表示文件的信息化标记。一些重要的域描述如下：

```
// Relocation info stripped from file.
#define IMAGE_FILE_RELOCS_STRIPPED             0x0001
// File is executable  (i.e. no unresolved external references).
#define IMAGE_FILE_EXECUTABLE_IMAGE           0x0002
// Line nunbers stripped from file.
#define IMAGE_FILE_LINE_NUMS_STRIPPED         0x0004
// Local symbols stripped from file.
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED       0x0008
// Agressively trim working set
#define IMAGE_FILE_AGGRESIVE_WS_TRIM          0x0010
// App can handle >2gb addresses
#define IMAGE_FILE_LARGE_ADDRESS_AWARE           0x0020
// Bytes of machine word are reversed.
#define IMAGE_FILE_BYTES_REVERSED_LO          0x0080
// 32 bit word machine.
#define IMAGE_FILE_32BIT_MACHINE                0x0100
// Debugging info stripped from file in .DBG file
#define IMAGE_FILE_DEBUG_STRIPPED              0x0200
// If Image is on removable media, copy and run from the swap file.
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP   0x0400
// If Image is on Net, copy and run from the swap file.
#define IMAGE_FILE_NET_RUN_FROM_SWAP          0x0800
// System File.
#define IMAGE_FILE_SYSTEM                        0x1000
```

```
// File is a DLL.
#define IMAGE_FILE_DLL                          0x2000
// File should only be run on a UP machine
#define IMAGE_FILE_UP_SYSTEM_ONLY               0x4000
// Bytes of machine word are reversed.
#define IMAGE_FILE_BYTES_REVERSED_HI            0x8000
```

我们常见的意义如下。

- 0x0001：该文件中没有重定位。
- 0x0002：文件是一个可执行的映像（即不是一个OBJ或LIB）。
- 0x2000：文件是一个动态连接库，不是一个程序。

第3章 Windows运行机理

3.4 线程的机制(3)

当Windows 95进程工作时，不用跟踪进程ID。实际上，大部分相关进程API函数期望一个HANDLE参数，通常称做hProcess。hProcess与某些事情（Win16任务数据库）没有直接的关联，与进程ID不一样，可有多重独特的hProcess值，但都属于同一个进程。

KERNEL32对象句柄

句柄渗透着Win32 API。一个句柄就是当需做某件事情时，从操作系统返回给API函数的一个魔数（Magic Value）。理论上讲一句柄值对应用程序是无意义的，只有操作系统知道如何去解释它（几乎所有Win16程序的句柄值可被解释为选择器值或指针）。

当用KERNEL32 API工作时，大部分句柄属于调用KERNEL32的句柄。KERNEL32句柄有专门属性，比如可传递给对象WaitforSingleObject这样的函数。KERNEL32的对象句柄包括进程柄、线程柄、文件柄、Mutex柄等。

一个KERNEL32句柄只有在进程自身内部有效，企图将一个进程柄用于另一个进程是没有意义的。尽管句柄在理论上是透明的，但对一应用程序而言，将一句柄转换成有用的对象指针是可能的。

Windows 95中最基本的进程函数是CreateProcess,这是模拟Win16 WinExec和LoadModule函数，且这两个函数仍存在于Windows 95中，但其内部有些改变。如果需要查询或操作后来的进程，则应使用CreateProcess，即可反馈给你一个hProcess HANLE。

因为WinExec和LoadModule没有hProcess和HANDLE的概念，所以不能返回hProcess。实际上，这两个函数调用CreateProcess以后，立即关闭了CreateProcess返回的hProcess，这样做的目的是防止为那些联系紧密且无必要的进程分配系统资源。

请记住，关闭一个处理并不意味着结束这个进程，相反你可通过特殊处理到该进程进行访问，当进程结束和所有的处理被关闭时，操作系统仔细地清除相关进程资源。

除了产生一进程获取一个hProcess外，另一个方法是有效的进程ID去调用OpenProcess。用hProcess可以做一些基本的进程查询和操作。在进程控制的范围，一个程序可以用TerminateProcess中止另一个进程，用SetPriorityClass影响另一个进程的执行优先权。

学习一下Windows mirror KERNEL是很有趣的，在进程的任务区，每一个Win32进程有16位任务数据库（TDB），并把TDB连接到TDB链上。如果你用TOOLHELP浏览这个任务表，则会看到除了这个16位任务外，每一个正在运行着的Win32程序也有一个TDB，TDB有8个字节的文件名，可重新调用。

除了TDB以外，对16位或32位进程而言，Windows中的所有TDB（包括Win32进程的TDB）还有一个PSP。和Windows 3.x不一样，Window 95 TDB中的PSP没有必要跟着TDB立即进入内存，

在TDB和PSP之间的100h字节存放当前目录区，这个区可有效地保存Window 95支持的足够大的长文件名和路径名目录，Windows 3.x中当前目录存放在TDB内一个只有65字长的区域内。

3.4.5 同步机制

1. 进程与线程同步

同步的意思是一个程序保证在不适宜地被切换时，不会出问题，虽然Windows 3.1有多任务，但没有真正的同步基础，因为这些多任务是协作多过调用API函数（如GetMessage和PeekMessage）。如果一个程序调用了GetMessage或Peekmessage，则意思是说“现在我处在可中断状态”。

Win32程序没有这样的协作多任务。它们必须做好随时被CPU切换掉的准备，一个真正的Win32程序不会耗尽CPU时间等待某些事件发生，Win32 API有四个主要的同步对象：

- Event 事件
- Seqmaphore 信号器
- Mutexes 互斥
- Critical Section 临界段

除Critical Section外，其余是系统全局对象，并且与不同进程及相同进程中的线程一起工作，这样，同步机也可以用于分离进程的同步活动（同一进程内部的线程除外）。

2. 事件（**Event**）

这是同步对象的一种类型，正如其名字的含义，在这个中心周围是一些发生在另一个进程或线程中的特殊活动。当你希望线程暂时挂起时，不会消耗CPU的工作周期。事件很类似于我们常用的消息的概念。如果我们剖析消息的内核肯定会发现，它就是用事件来实现的。

程序可用CreateEvent或OpenEvent对事件获得一个句柄：

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
                          // pointer to security attributes
  BOOL bManualReset,     // flag for manual-reset event
  BOOL bInitialState,   // flag for initial state
  LPCTSTR lpName          // pointer to event-object name
);
HANDLE OpenEvent(
  DWORD dwDesiredAccess,     // access flag
  BOOL bInheritHandle,        // inherit flag
  LPCTSTR lpName               // pointer to event-object name
);
```

然后，该程序再调用 WaitForSingleObject,选定事件柄和暂停周期，那么线程就被挂起，一直到其他线程给出事件有关信号后才被再次激活。其他线程指调用SetEvent或PulseEvent所需活动的线程，事件获得这个信号，被挂起的线程即被唤醒并继续执行。

例如，当一个线程要使用另一个线程的排序结果时，你或许希望去使用一个事件。比较糟的方法是执行这个线程并在结束时设置全局变量标志，另一个线程循环检查这个标志是否已设置，这将浪费许多CPU的时间。用事件（Event）做同样的事情则很简单，排序线程在结束时产生一个事件（Event）,其他线程调用WaitForSingleObject。这就使得线程被挂起，不浪费CPU周期，当排序线程完成排序时，调用SetEvent唤醒另一个线程继续执行，有效地利用了CPU。

除了WaitForSingleObject外，还有WaitForMultipleObject允许一个线程被挂起，一直到要么满足Event条件，要么有一个等待视窗信息时能恢复，其他挂起的函数一直等到挂起的被满足或I/O操作已经完成时才能，无疑这里体现了灵活性。

3. 信号器（**Semaphores**）

当你需限制访问特殊资源或限制一段代码到某些线程时，Semaphores非常有用。打一个比喻，就像是餐厅用的餐桌一样，假设这个餐厅有二十个餐桌，当你去时，二十个餐桌都有人在用餐，你就只好等二十个餐桌中有人吃完后才能去用餐，否则你必须等待。在Win32编程中获得Semaphores，就好像得到餐桌的一次控制。

为了利用Semaphores，一个线程调用 CreatSemaphore去获得一个HANDLE给Semaphores。该调用包括同时有多少线程使用资源或代码，如果其他线程在另一个进程中，可调用OpenSemaphore去获得一个可利用的HANDLE，当一个线程需要访问共享资源时，要把资源传递给WaitForSingleObject，如果这个Semaphore没有被等待的所有线程请求，等待功能将简单处理Semaphore的使用数，且线程继续执行。换句话说，如果Semaphore已经超出最大值，则调用等待功能的线程将被挂起。一个线程的含义就是使用一个Semaphore来执行，并用ReleaseSemaphore来释放资源。

第3章 Windows运行机理

3.4 线程的机制(4)

4. 互斥（**Mutexes**）

这是同步对象的第三种类型，Mutex（互斥）是“mutual exclusion”的缩略语。一个程序或一组程序希望一次只有一个线程去访问一个资源或一段代码时可使用一次互斥。如果一个线程正在使用这个资源，则另一个线程被排斥在同一资源之外。互斥的用法非常类似于信号器，产生、打开和释放信号器函数都有与互斥类似的内容。当一个线程有互斥要求时，可调用WaitForSingleObject/ WaitForMultipleObjects系列中的函数。

用餐桌来比喻的话，就是整个餐厅只有一个餐桌，当有一个人在用餐时，另一个人只能等待用餐。

5. 临界段（**Critical Sections**）

临界段相当于一个微型的互斥，只能被同一进程中的线程使用。临界段是为了防止多线程同时执行同一段代码。相对其他同步机而言，临界段相对简单和易用，一个临界段可以被认为是仅在单一进程中有效的轻量级互斥。为了使用临界段，一个程序要么分配，要么声明一个CRITICAL_SECTION类型的全局变量。在临界段首次使用之前，其场地需要通过调用InitiazeCriticalSection进行初始化，之后调用EnterCriticalSection将一线程进入临界段了。

临界段使用起来很简单，在Windows 95中，当没有其他线程时，如果一个线程线程调用EnterCriticalSection,则只需在CRITICAL__SECTION结构中调整和设置一些场地即可。只有已经存在临界段的另一个线程把EnterCriticalSection调入VMIN 32 VxD时，才能使该线程挂起。

6. WaitForSingleObject/ WaitForMultipleObjects函数

至此，已经概述了线程同步的四种基本方法，我想谈论一下同步线程的其他方法。除了事情、信号器和互斥外，WaitForSingleObject/ WaitForMultipleObjects系列函数可接受几种其他的句柄，把一个进程HANDLE传到一个WaitForSingleObject/ WaitForMultipleObjects函数，则会引起调用线程挂起。如果这个进程已经中止，则Wait函数立即返回。同样，把一个线程的HANDLE传到WaitForSingleObject/ WaitFor Multiple Objects，调用线程也将被挂起。

WaitForSingleObject/ WaitForMultipleObjects函数可以挂起的另一个HANDLE是这个文件的变更，之间的变更可以限定一个给定的目录及有选择的子目录。WaitForSingleObject/ WaitForMultipleObjects函数的另外一个HANDLE是一个针对输入装置的HANDLE文件，一旦有未经使用的输入进入输入缓存，Wait函数则返回，并告诉线程继续执行。

3.5 PE结构分析

因为PE结构是一个很复杂的结构，所以下面我们在讨论PE时把它分为PE头标、表节、文件导入/导出、资源分别介绍。如果你只对某部分内容感兴趣，可以直接跳到此节阅读。

3.5.1 PE头标

PE 的意思就是 Portable Executable（可移植的执行体）。它是 Win32环境自身所带的执行体文件格

式。它的一些特性继承自 Unix的 Coff (common object file format)文件格式。“Portable Executable”（可移植的执行体）意味着此文件格式是跨Win32平台的：即使Windows运行在非Intel的CPU上，任何win32平台的PE装载器都能识别和使用该文件格式。当然，移植到不同的CPU上的PE执行体必然得有一些改变。所有Win32执行体（除了VxD和16位的DLL）都使用PE文件格式，包括NT的内核模式驱动程序（Kernel Mode Drivers）。

我们在PE结构中最先看见的PE格式中的是PE结构的头标。像所有其他微软可执行文件格式一样，PE文件在一个已知（或容易找到的）位置上，有一系列域来定义该文件其余部分看起来像什么。PE头标包含了至关重要的一些信息，诸如代码和数据区的位置和大小、该文件要用什么操作系统以及初始的堆栈大小。我们在学习PE结构时最好用PEDUMP来DUMP一个EXE或DLL文件比较好学习点（PEDUMP可以在X:Msvc\COMMON\TOOLS找到，X为VC的安装目录）。

1. DOS头

与其他微软的可执行格式相似的是，在PE头标前面还有一个百多个字节的DOS头。这个DOS区域是一小段DOS程序。这一段程序只有几行简单的汇编程序，在Windows 3.1中可以自己定义。把一个很大的DOS程序当成PE结构的头也是可以的，例如说做一个从DOS下启动的游戏，就可以把DOS启动的内容放在前面。到了Windows 9x中的PE结构，在VC 4.0以后，DOS头就不可定义了。

现在，它的作用是如果此程序在DOS平台运行时，它将打印出“该程序不能在DOS模式下运行”之类的信息。这样就能提示程序的用户到Windows平台去运行此程序。图3.9是PE结构图。

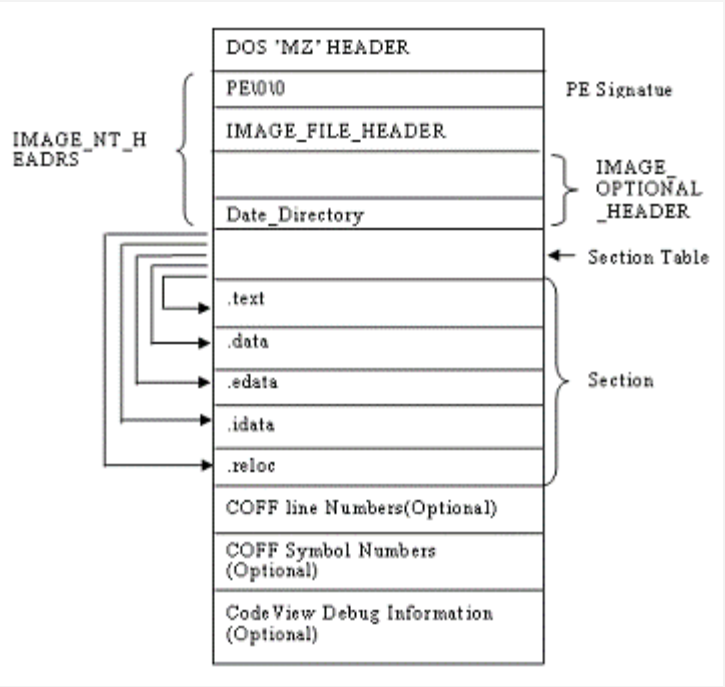


图3.9 PE结构图

PE文件的所有结构都能在WINNT.H文件中找到，其结构如下：

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD   e_magic;              // Magic number
    WORD   e_cblp;               // Bytes on last page of file
    WORD   e_cp;                  // Pages in file
    WORD   e_crlc;               // Relocations
    WORD   e_cparhdr;           // Size of header in paragraphs
```

```
    WORD   e_minalloc;          // Minimum extra
   //paragraphs needed
    WORD   e_maxalloc;        // Maximum extra
   //paragraphs needed
    WORD   e_ss;                  // Initial (relative) SS value
    WORD   e_sp;             // Initial SP value
    WORD   e_csum;          // Checksum
    WORD   e_ip;              // Initial IP value
    WORD   e_cs;              // Initial (relative) CS value
    WORD   e_lfarlc;            // File address of relocation table
    WORD   e_ovno;           // Overlay number
    WORD   e_res[4];            // Reserved words
    WORD   e_oemid;        // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;      // OEM information;
   //e_oemid specific
    WORD   e_res2[10];         // Reserved words
    LONG   e_lfanew;            // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

第3章 Windows运行机理

3.4 线程的机制(6)

4. IMAGE_OPTIONAL_HEADER

PE头标的第三部分是一个IMAGE_OPTIONAL_HEADER类型结构。对于PE文件，这部分是必要的。除了标准的IMAGE_FILE_HEADER外，COFF格式还允许单独定义一个附加信息结构。

IMAGE_OPTIONAL_HEADER分为两种，一种是32位的，一种是64位的，我们可以在WINNT.H中找到对应的结构，其名分别为：IMAGE_OPTIONAL_HEADER32各IMAGE_OPTIONAL_HEADER64。我们在这里只对32位进行介绍，其结构如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD     Magic;
    BYTE      MajorLinkerVersion;
    BYTE      MinorLinkerVersion;
    DWORD     SizeOfCode;
    DWORD     SizeOfInitializedData;
    DWORD     SizeOfUninitializedData;
    DWORD     AddressOfEntryPoint;
    DWORD     BaseOfCode;
    DWORD     BaseOfData;

    //
    // NT additional fields.
    //

    DWORD     ImageBase;
    DWORD     SectionAlignment;
    DWORD     FileAlignment;
    WORD      MajorOperatingSystemVersion;
    WORD      MinorOperatingSystemVersion;
```

```
    WORD      MajorImageVersion;
    WORD      MinorImageVersion;
    WORD      MajorSubsystemVersion;
    WORD      MinorSubsystemVersion;
    DWORD     Win32VersionValue;
    DWORD     SizeOfImage;
    DWORD     SizeOfHeaders;
    DWORD     CheckSum;
    WORD      Subsystem;
    WORD      DllCharacteristics;
    DWORD     SizeOfStackReserve;
    DWORD     SizeOfStackCommit;
    DWORD     SizeOfHeapReserve;
    DWORD     SizeOfHeapCommit;
    DWORD     LoaderFlags;
    DWORD     NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
          DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Magic表示标志映像文件状态的一个WORD记号。值定义如下:

```
#define IMAGE_NT_OPTIONAL_HDR32_MAGIC        0x10b
#define IMAGE_NT_OPTIONAL_HDR64_MAGIC        0x20b
#define IMAGE_ROM_OPTIONAL_HDR_MAGIC          0x107
```

- 0x0107:一个ROM映像。
- 0x010B:一个普通的可执行映像(大多数文件含此值)。

MajorLinkerVersion和MinorLinkerVersion表示生成该文件的连接器版本号。该数字以十进制形式显示,而不是十六进制,一个典型的连接器版本号是2.23。

SizeOfCode表示所有代码段组合聚集在一起的尺寸大小,内存中整个PE映像体的尺寸。它是所有头和节经过节对齐处理后的大小。

SizeOfInitializedData表示由初始化的数据(不包括代码段)组成的所有节的总尺寸。

SizeOfUninitializedData表示初始化的数据的大小。未初始化的数据通常被归入称为.bss的一节中。

AddressOfEntryPoint表示映像开始执行位置的地址。PE装载器准备运行的PE文件的第一个指令的RVA。若您要改变整个执行的流程,可以将该值指定到新的RVA,这样,新RVA处的指令首先被执行。

BaseOfCode表示文件代码节开始处的RVA。典型情况下,代码节在PE头标之后,并在数据节之前进入内存。在微软生成的EXE文件中,该RVA通常是0x1000。

BaseOfData表示文件的数据节开始处的RVA。典型情况下,数据节最后进入内存,排在PE头标和代码节后面。

ImageBase表示当连接器创建一个可执行文件时，它假设该文件将被内存映射到内存中的一个指定位置上。也就是PE文件的优先装载程序的地址。因为在Windows操作系统中，总是把可执行程序安装到虚拟空间中去，每个虚拟空间在逻辑上都是相对独立的，不相干的。此值就是表示程序装在虚拟空间的什么地方开始。

SectionAlignment表示内存中节对齐的粒度。例如，如果该值是4096 (1000h)，那么每节的起始地址必须是4096的倍数。若第一节从401000h开始且大小是10个字节，则下一节必定从402000h开始，即使401000h和402000h之间还有很多空间没被使用。

FileAlignment表示文件中节对齐的粒度。例如，如果该值是(200h),，那么每节的起始地址必须是512的倍数。若第一节从文件偏移量200h开始且大小是10个字节，则下一节必定位于偏移量400h: 即使偏移量512和1024之间还有很多空间没被使用/定义。

MajorOperatingSystemVersion和MinorOperatingSystemVersion表示使用该可执行文件所要求的操作系统最小版本。该域含义有点模棱两可，因为subsystem域（后面的一些域）页体现类似的目的。在大多数Win32文件中，该域为版本1.0。

MajorImageVersion和MinorImageVersion表示一个用户自定义域。该域允许你具有一个EXE或一个DLL的不同版本。可用连接器的/VERSION开关来置该域的值，如LINK/VERSION：2.0 myobj.obj。

MajorSuvsystemVersion和MinorSubsystemVersion表示运行该可执行文件所要求的最小子系统版本。该域的一个典型值是4.0（意为Windows 4.0，即Windows 95）。

Reserved1一般总为0。

SizeOfImage一般是装载器不得不关心的映像部分的总尺寸。它是从映像基地址开始直到最后一节的尾端这个范围的长度。最后一节的尾端是被调整为最接近节对齐值的倍数的。

SizeOfHeaders表示PE头标和节（对象）表的尺寸。这些节的生数据直接跟在所有头标部分之后。

SizeOfHeaders ＝所有头+节表的大小
也就等于文件尺寸减去文件中所有节的尺寸。

CheckSum总是值0。

Subsystem表示该可执行文件为它用户接口而使用的子系统类型。WINNT.H定义了如下值:

```
// Unknown subsystem.
#define IMAGE_SUBSYSTEM_UNKNOWN              0
// Image doesn't require a subsystem.
#define IMAGE_SUBSYSTEM_NATIVE                   1
// Image runs in the Windows GUI subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_GUI          2
// Image runs in the Windows character subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_CUI          3
// image runs in the OS/2 character subsystem.
#define IMAGE_SUBSYSTEM_OS2_CUI                5
// image runs in the Posix character subsystem.
#define IMAGE_SUBSYSTEM_POSIX_CUI                7
// image is a native Win9x driver.
```

```
#define IMAGE_SUBSYSTEM_NATIVE_WINDOWS        8
// Image runs in the Windows CE subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_CE_GUI       9
```

第3章 Windows运行机理

3.5 PE结构分析(1)

3.5.2 表节

PE文件的真正内容划分成块，称之为sections（节）。每节是一块拥有共同属性的数据，比如代码/数据、读/写、导入/导出等。我们可以把PE文件想像成一逻辑磁盘，PE header 是磁盘的boot扇区，而sections就是各种文件，每种文件自然就有不同属性，如只读、系统、隐藏、文档等。节的划分是基于各组数据的共同属性，而不是逻辑概念。重要的不是数据/代码是如何使用的，如果PE文件中的数据/代码拥有相同属性，它们就能被归入同一节中。

不必关心节中类似于“data”、“code”或其他的逻辑概念：如果数据和代码拥有相同属性，它们就可以被归入同一个节中。节名称仅仅是个区别不同节的符号而已，自己也可以定义一些不同的名字的字。

节表位于PE头标和映像节的生数据中间，节表包含了关于映像中的每节的信息。映像的节是以它们的地址而不是其字母来排序的。

在此处是值得弄清楚一个节到底是什么的时候了。

然而与NE文件的段表又不同，一个PE节表并不为每个代码或数据块保存一个选择器的值。取而代之的是，节表的每一项存储一个地址，该地址是文件的生数据被影射入内存所在位置的地址。尽管节类似于32位段，但它们确实不是单独的段。实际上，一个节简单地对应一个进程的虚拟地址空间中的一片内存区域。

PE文件不同于NE文件的另一个方面，体现在它们是如何管理支撑数据方面，你的应用程序不使用这些支撑数据，但操作系统要用。可执行模块用到的DLL列表和安置表的位置是支撑数据的两个例子。

用PE文件就不同了。任何被认为是相关的代码和数据被存储在一个节中。因此，关于引入函数的信息存储在它自己的节中，它被作为模块引出的函数表。对重定位数据也是如此。任何可能被程序或操作系统需要的代码或数据同样也是获得它们自己的节。

我先描述操作系统管理这些节所用的数据，在内存中，紧跟在PE头标之后的是一个IMAGE_SECTION_HEADER数组。这个数组中的元素个数在PE头标中（的IMAGE_NT_HEADER.FileHeader.NumberOfSection域）给出。

IMAGE_SECTION_HEADER的结构如下：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD       PhysicalAddress;
            DWORD       VirtualSize;
         } Misc;
    DWORD     VirtualAddress;
    DWORD     SizeOfRawData;
```

```
    DWORD     PointerToRawData;
    DWORD     PointerToRelocations;
    DWORD     PointerToLinenumbers;
    WORD     NumberOfRelocations;
    WORD      NumberOfLinenumbers;
    DWORD     Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

用PEDUMP程序可输出节表和所有节的域和属性。下面分别显示了一个典型的EXE文件PEDUMP输出的节表以及一个OBJ文件的节表输出。

```
Section Table
  01 .text     VirtSize: 00002C2A  VirtAddr:  00001000
    raw data offs:   00000400  raw data size: 00002E00
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:         00000000
    characteristics: 60000020
      CODE  MEM_EXECUTE  MEM_READ

  02 .rdata    VirtSize: 0000038F  VirtAddr:  00004000
    raw data offs:   00003200  raw data size: 00000400
    relocation offs: 00000000  relocations:    00000000
    line # offs:     00000000  line #'s:         00000000
    characteristics: 40000040
      INITIALIZED_DATA  MEM_READ

  03 .data     VirtSize: 00001334  VirtAddr:  00005000
    raw data offs:   00003600  raw data size: 00001000
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:         00000000
    characteristics: C0000040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE

  04 .idata    VirtSize: 000006E2  VirtAddr:  00007000
    raw data offs:   00004600  raw data size: 00000800
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:         00000000
    characteristics: C0000040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE
```

```
05 .rsrc     VirtSize: 00000550  VirtAddr:  00008000
  raw data offs:   00004E00  raw data size: 00000600
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:         00000000
  characteristics: 40000040
    INITIALIZED_DATA  MEM_READ

06 .reloc    VirtSize: 0000041E  VirtAddr:  00009000
  raw data offs:   00005400  raw data size: 00000600
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:         00000000
  characteristics: 42000040
    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

每个IMAGE_SECTION_HEADER是关于EXE或OBJ文件中一节信息的一个完整数据库，它具有如下格式。

Name[IMAGE_SIZEOF_SHORT_NAME]：这是给本节命名的一个8字节长的ANSI名，多数节名以一个小数点作为开始（例如：.text），你也可以在微软C/C++编译器中用#pragma data_seg和#pragma code_seg来命名。重要的是，注意如果节名占满了8个字节，则没有NULL中止字节。如果你爱用printf()，可使用“%.8s”，以避免拷贝名字串到另一缓冲区时误用空白符中止了它。

Misc：根据是出现在EXE文件中还是在OBJ文件中，该域具有不同的含义。在一个EXE中，它保存代码或数据节的虚拟尺寸，这是调整到最接近文件对齐值倍数的尺寸，本结构中后面的SizeOfRawData域保存这个对齐值。对于OBJ文件，该域指示本节的物理地址。第一节在地址0上开始。要找到其下一节的物理地址，只需要将SizeOfRawData值与当前的此物理地址相加即可。

VirtualAddress节：的RVA（相对虚拟地址）。PE装载器将节映射至内存时会读取本值，因此如果域值是1000h，而PE文件装在地址400000h处，那么本节就被载到401000h。

SizeOfRawData：在EXE文件中，该域含本节被对齐到文件对齐尺寸后的尺寸。

PointerToRawData：这是基于文件的偏移量，用它可以找到本节的生数据所在位置。如果你自己内存映射一个PE或COFF文件（而不是让操作系统装载它），则该域比VirtualAddress更为重要。那是因为在这种情况下，你将有整个文件的一个完全线性映射，因此你将在该偏移处找到本节的数据，而不是用VirtualAddress域指定的RVA。

PointerToRelocations：在OBJ文件中，这是一个该节重定位信息基于文件的偏移量。OBJ每节的重定位信息直接跟在该节数据之后。在EXE文件中，这个域（和后一个域）无意义，并总被置为0。当连接器创建EXE时，它已解决了大多数的地址分配和安排问题，只有基地址重定位和引入函数才在装载时解决。有关基地址和引入函数的信息存储在基地址和引入函数节中。因此，对一个EXE，不需要在节的生数据之后还要有每节重定位的数据。

第3章 Windows运行机理

3.4 线程的机制(7)

表示的意义如下。

- native=1：不需要子系统（例如，一个设备驱动器）
- WINDOWS_GUI=2：在Windows GUI子系统中运行
- WINDOWS_GUI=3：在Windows字符子系统中运行（一个控制台应用程序）
- OS2_GUI=5：在OS/2字符子系统中运行（只对OS/2 1.x的应用程序）
- POSIX_CUI=7：在Posix字符子系统中运行

DllCharacteristics （在NT 3.5中标为obsolete）指示什么情况下一个DLL的初始化函数，例如DllMain（）要被调用的标志集合。该值看起来总被置为0，然而操作系统仍为4个事件调用了DLL初始化函数。

被定义的值如下。

- 1：当DLL第一次被装入一个进程的地址空间时调用；
- 2：当一个线程中止时调用；
- 4：当一个线程启动时调用；
- 8：当DLL退出时调用。

SizeOfStakeReserve表示为初始线程栈保留的虚拟内存量。然而，这些内存不是都要交付的（见后一个域）。该域默认为0x100000（1MB）。如果你对CreateThread（）指定一个0作为栈的大小，结果线程仍是得到一个域默认值相同的栈。

SizeOfStackCommit表示为初始线程栈首先交付的内存量。在微软连接器中，该域默认值是0x1000字节（1页），而TLINK默认为0x2000字节（2页）。

SizeOfHeapReserve表示为初始进程堆保留的虚拟内存量。该堆句柄可通过调用GetProcessHeap()来获得。这些内存也不是都要交付的（见下一个域）。

SizeOfHeapCommit表示在进程堆中初始交付的内存量。连接器在该域的默认值是0x1000字节。

Loaderflags（在NT 3.5中标记为obsolete）它们一般是与调试支持有关的域。

NumberOfRvaAndSizes表示在DataDiretory数组中项的数目。目前的工具总把该域的值置为16。

DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]是一个IMAGE_DATA_DIRECTORY结构数组。数组中前面的元素包含了该可执行文件重要部分的起始RVA和尺寸。数组尾端的元素目前还未用到。数组的第一个元素总是引出函数表（如果有的话）的地址和尺寸。第二个数组项是引入函数表的地址和尺寸，如此等等。对于一个完整的数组项的定义列表，在WINNT.H中的IMAGE_DIRECTORY_ENTRY_xxx #defin's中有如下的几项：

```
// Export Directory
#define IMAGE_DIRECTORY_ENTRY_EXPORT              0
```

```
// Import Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT               1
// Resource Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE             2
// Exception Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION           3
// Security Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY            4
// Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_BASERELOC           5
// Debug Directory
#define IMAGE_DIRECTORY_ENTRY_DEBUG                 6
// Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE       7
// RVA of GP
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR           8
// TLS Directory
#define IMAGE_DIRECTORY_ENTRY_TLS                   9
// Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG        10
// Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT     11
// Import Address Table
#define IMAGE_DIRECTORY_ENTRY_IAT                  12
// Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT       13
// COM Runtime descriptor
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR    14
```

该数组的目的是允许装载器可迅速地找到一个映像的特定节（例如引入函数表），而不必遍历映像的每一个节并逐一比较它们的名字。数组的大多数项描述了一个完整的节的数据。然而，IMAGE_DIRECTORY_ENTRY_ DEBUG元素只含了.rdata节中一小部分字节。

第3章 Windows运行机理

3.5 PE结构分析(2)

PointerToLinenumbers：表示行号表的基于文件的偏移量。一个行号表把源文件行号和一个地址对应起来，在该地址上可找到给定行产生的代码。主要用于调试中。

NumberOfRelocations：表示本节重定向表中重定向的数目（PointerTorRelocations域已在前面列出）。该域只用于OBJ文件。

WORD NumberOfLinenumbers：表示本节行号表中行号的数目（PointerRoLinenumbers域已在前面列出）。

Characteristics：含标记以指示节属性，比如节是否含有可执行代码、初始化数据、未初始数据，是否可写、可读等。对所有可能的节属性的列表请见WINNT.H中的IMAGE_SCN_XXX_XXX #defines。

下面我们列出了一些常见的节名。

1. .text节

.text节又叫代码节，它是编译器或汇编器产生的所有通用码。在.text节中，除了我用编译器创建的和从运行时间库中用到的外，还有另外附加的代码时，我感到很惊奇。在PE文件中，当你调用另一个模块中的函数（例如USER32.DLL的GetMessage()）时，编译器产生的CALL指令并不是把控制直接传给DLL中的该函数。取而代之的是，该调用指令把控制传给也是在该.text节中的一个JMP DWORD PTR[XXXXXXXX]指令。该JMP指令跳转到以一个DWORD存于.idata节中的一个地址上。这个.idata节的DWORD含该操作系统函数入口点的真正地址，如图3.10所示。

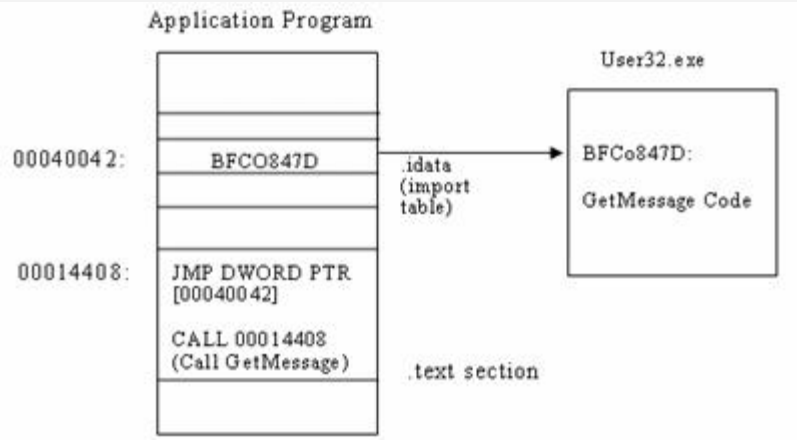


图3.10 .text节表

通过一个位置把所有对一个给定的DLL函数的调用进行归结后，装载器就没有必要对每个调用DLL的指令进行拼凑了。PE装载器必须要做的只是把目标函数的正确地址放入.idata节中的该DWORD中就行了。没有任何的CALL指令需要拼凑。这与NE文件有明显的不同，后者中每个段含有一个用在该段上的安置表。如果该段调了某个DLL函数20次，则装载器必须将该函数的地址拷贝到该段中20次。在PE方法

下，你不能用一个DLL函数的真正地址来初始化一个变量，你会以为如下句子：

```
FARPROC pfnGetMessage=GetMessage;
```

将会把GetMessage的地址放入变量pfnGetMessage中。在Win16中，这确实如此，但在Win32中则行不通。在Win32中，变量pfnGetMessage结果存的是在.text节中转换了的JMP DWORD PTR[XXXXXXXX]的地址。如果你通过函数指针来调用，结果会像你期望的那样出现。然而，如果要在GetMessage()的开始处读这些字节，则就不那么幸运了（除非你自己做一些附加的工作来跟随.idata的“指针”）。

2. .data节

正像.text是代码的默认节一样，.data节就是初始化了的数据所在的地方。初始化了的数据由全局变量和静态变量组成，他们在编译时被初始化。它还包括字符串文字（例如，在一个C/C++程序中的字符串“HELLO WORLD”）。连接器把来自于OBJ和LIB文件的所有.data节组合成EXE中的一个.data节。局部变量是被定位在一个线程栈上的，并且在.data或.bss节中不占空间。

3. .bss节

.bss节是未初始化的静态和全局变量存储的地方。连接器把来自于OBJ和LIB文件的所有.bss节组合成为EXE中的一个.bss节。在节表中，为.ss节所用的RawDataOffset域被置为0，表示这一节在文件中未占任何空间。TLINK32不产生.bss节，代之的是它扩展DATA节的虚拟尺寸以说明未被初始化的数据。

4. .CRT节

.CRT节是另一个初始化了的数据节，它被微软C/C++运行时间库（因而称为.CRT）所使用。该节中的数据被用于这样一些事情，如在Main或WinMain被执行之前调用静态C++类的构造函数。

5. .rsrc节

.rsrc节包含了本模块所用的资源。在NT出现后的较早时期，16位RC.EXE产生的.RES文件输出格式不能被微软连接器所识别。CVTRES程序把这些.RES文件转换成COFF格式的OBJ，并把数据放入OBJ内部的.rsrc节中。连接器然后才能把资源OBJ当做另一个OBJ连接进来。这意味着连接器并不必知道关于资源的任何特殊的东西。微软较新的连接器似乎能够直接处理.RES文件。

6. .idata节

.idata节包含模块从其他DLL引入的函数（和数据）的信息。该节等价于NE文件的一个模块访问表。不同的关键点在于：PE文件引入的每个函数特别地要在这一节列出。要在一个NE文件中找等价的信息，你不得不深入到每段所用的生数据尾端处的重定位。

7. .edata节

.edata是被其他模块使用的PE文件引出的函数和数据的一个列表。NE文件与此等价的是项表、驻留名字表和非驻留名字表这三个表的结合。不像在Win16中那样，几乎没有理由从一个EXE文件中输出任何东西，因此，通常只能在DLL文件中看到.edata节。例外的是Borland C++产生的EXE文件，它似乎总是引出一个函数（_ _GetExceptDLLinfo），该函数为运行时间库内部使用。

当使用微软工具时，.edata节中的数据通过.EXP文件到PE文件中。另一方面，连接器本身不产生这些信息，而是依靠库管理器（LIB32）来扫描OBJ文件，并创建.EXP文件，然后连接器把该.EXP文件加到模块列表中以便连接。那些麻烦的.EXP文件确实正是具有一个不同扩展名的OBJ文件。通过用/S（显示符号表）选项来运行PEDUMP程序，可以看到从一个.EXP中引出的函数。

第3章 Windows运行机理

3.5 PE结构分析(3)

8. .reloc节

.reloc节容纳了一个基址重定位的表。基址重定位是对指令或初始化过的变量值的一个调整；如果装载器不能把EXE或DLL文件装到连接器假定它应该放置的地址上时，则该文件需要做这个调整。如果装载器能把映像装到连接器预先确定的基地址上，则装载器将忽略该节中的重定位信息。

如果你想要进行一个选择，并且希望装载器能够总把映像装到假定的基地址上，可使用/FIXED选项来告诉连接器除去这个信息。尽管这样可在可执行文件中节省空间，但它可能使该可执行文件不能在别的Win32平台上运行。例如，假设你建立了一个NT下的EXE文件，并把该EXE基址定到0x10000处。如果你告诉连接器除去重定位信息，则该EXE将不能在95下运行，因为在95下，地址0x10000不是有效的（在95中，最小的装载地址是0x400000，即4MB）。

注意被编辑器产生的JMP和CALL指令用的是相对于其指令的偏移量，而不是在32位段中的实际偏移量。假如映像需要装载到与连接器所指定的基地址不同的位置上，这些指令也不需改变，因为他们用的是相对地址。如果需要重定位的并没有像你想像得那么多，通常只有使用了对某些数据的32位偏移量的指令才需要重定位。例如，假如有如下的全局变量声明：

```
int         Addr;
int         *ptr=& Addr;
```

如果连接器给映像指定的基址是0x10000，则变量Addr的地址结构是含像0x12004之类的值。在存指针ptr的内存处，连接器将写值0x12004，因为那是变量Addr的地址。如果装载器（不管是什么原因）决定在0x70000为基地址的地方装载该文件，Addr的地址则将为0x72004。然而，这样该预先初始化的ptr变量的值就不正确了，因为Addr现在在内存中比原来高了0x60000字节。

这正是重定位信息发挥作用的地方。.reloc节实际上是记录了映像中一些位置的一张表，在这些位置上，连接器所假定的装载地址和实际装载地址之间的差别需要考虑。

3.5.3 PE文件引入

我们知道函数是如何对外部DLL文件调用的。它并不是直接调DLL的函数地址，而通过CALL指令转向该可执行文件中的.text节中其他地方上的一个JMP DWORD PTR[XXXXXXXX]。作为一种选择，如果在VC中用了__declspec(dllimprot)，则函数调用变成CALL DWORD PTR[XXXXXXXX]。在这两种情况下，JMP或CALL要查的地址存于.idata节中。JMP或CALL指令把控制传给该地址，该地址是所要求的目的地址。

在被装入内存之前，PE文件的.idata节包含了一些信息。这些信息对于装载器确定目标函数的地址并把它们拼入可执行的映像中，是必不可少的。在.idata节被装入后，它包含了一个指针，该指针指

向EXE/DLL引入的函数。注意，在本节我所讨论的所有数组和结构都被包含在.idata节中。

.idata节（从文件来说可以是idata节，如果内存映射就是import table，即引入表）用一个IMAGE_IMPORT_DESCRIPTOR的数组作为开始。对于PE文件隐含连接的每个DLL，都有一个IMAGE_IMPORT _DESCRIPTOR。对于该数组，没有任何计数来指示该数组中结构的数目，数组的最后一个元素是通过在最后域中填入NULL的一个IMAGE_IMPORT_DESCRIPTOR来表示的。一个IMAGE_IMPORT_ DESCRIPTOR的格式如下：

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
       // 0 for terminating null import descriptor
          DWORD   Characteristics;
       // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
          DWORD   OriginalFirstThunk;
    };
    // 0 if not bound,
   // -1 if bound, and real date\time stamp
   // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
   //(new BIND) O.W. date/time stamp of DLL bound to (Old BIND)
    DWORD   TimeDateStamp;
   // -1 if no forwarders
   DWORD   ForwarderChain;
   DWORD   Name;
   // RVA to IAT (if bound this IAT has actual addresses)
   DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED
```

- Characteristics/OriginalFirstThunk：该域是相对的一个偏移量（RVA）。它指向一个IMAGE_THUNK_DATA类型的数据。其中，每个IMAGE_THUNK_DATA DWORD对应一个被该EXE/DLL引入的函数。
- TimeDataStamp：表示该时间/日期印记指示文件是什么时间建立的。该域一般为0。
- ForwarderChain：该域与传递相关联，包括一个DLL把对它的一个函数的访问传递给另一个DLL。例如，在Windows中，KERNEL32.DLL把它的一些引出函数传递给NTDLL.DLL，一个应用程序或许认为它调用了KERNEL32.DLL中的一个函数，但实际上它的调用进入到了NTDLL.DLL中。该域把一个索引含到FirstThunk数组中，被该域索引过的函数将被传递给另一个DLL。
- Name：这是相对一个用null作为结束符的ASCII字符串的一个RVA，该字符串含的是该引入DLL文件的名字（例如，KERNEL32.DLL或者USER32.DLL）。
- PIMAGE_THUNK_DATA FirstThunk：　该域是相对一个PIMAGE_THUNK_DATA

DWORD数组的一个偏移量（RVA）。大多数情况下，该DWORD被解释成指向一个IMAGE_IMPORT_ BYNAME结构的一个指针。然而，也有可能用顺序值引入一个函数。

一个IMAGE_IMPORT_DESCRIPTOR的重要部分是引入DLL的名字和两个IMAGE_THUNK_DATA DWORD数组。每个IMAGE_THUNK_ DATA DWORD对应一个引入函数。在EXE文件中，这两个数组（被Characteristics和FirstThunk域所指向）并行地运行，并且都是在尾端处以一个NULL指针项作为中止。

为什么会有两个并行的指向IMAGE_THUNK_DATA结构的指针数组呢？

第一个数组（被Characteristics指向的那一个）被单独留下，并且绝不会被修改，它有时也被称做提示名称表（hint-name table）。

第二个数组（被IMAGE_IMPORT_DESCRIPTOR的FirstThunk域所指向）被PE装载器改写，然后用该引入函数的地址来改写IMAGE_ THUNK_DATA DWORD的值。

第3章 Windows运行机理

3.5 PE结构分析(4)

对DLL函数的CALL调用要通过一个“JMP DWORD PTR[XXXX XXXX]”转换，该转换的[XXXXXXXX]部分要根据FirstThunk数组中的某一项而定。因为被装载器实际地改写了的这个IMAGE_THUNK_DATA数组，保存了所有引入函数的地址，因此它被称为“引入地址表”（Import Address Table）。图3.11显示了这两个数组。

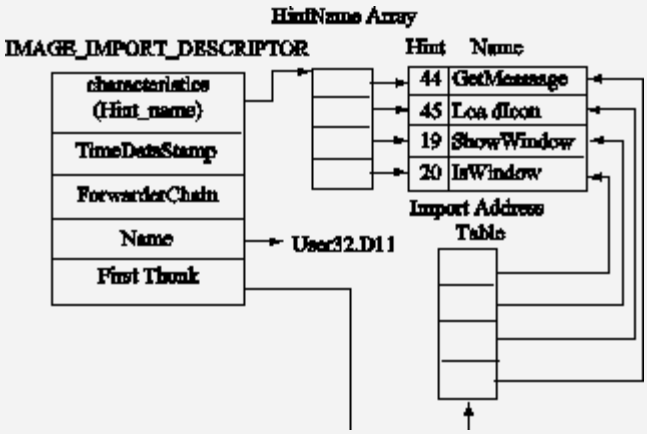


图3.11 两个指针数组

因为该引入地址表通常是在一个可写的节中，因此可相对比较容易地截取一个EXE或DLL文件对另一个DLL调用。可简单地把恰当的引入地址表项指向希望截取的函数，这不需要修改调用者中的任何代码。这个功能是非常有用的。

在引入库中的一个.idata节包含了替换要反查的DWORD。另一个.idata节有一个空间用于“提示序数”，而引入函数名紧跟其后。这两个域构成一个IMAGE_IMPORT_BY_NAME结构。当你稍后连接一个用了该引入库的PE文件时，给引入库中的替换具有和被引入的函数相同的名字。连接器认为这个替换真正就是引入函数，并且把对引入函数的调用安置到替换点上。在引入库中的替换基本上可以看成是引入函数。

除了提供一个引入函数替换的代码部分外，引入库提供了PE文件的.idata节（或称为引入表）的部分东西。这些部分来自于库管理器放入引入库中的各个.idata间的差别。它只不过是遵循为建立和组合节而预先设置好的规则，并且每件事都自然而然地到了位。

每个IMAGE_THUNK_DATA DWORD对应一个引入函数。该DWORD的解释根据该文件是否已被装入内存和该函数是否已通过名字或序数来引入了（通过名字更常用一些）而变化。

当一个函数是通过其序数值引入的时（少见的情形），EXE文件的IMAGE_THUNK_DATA DWORD中置最高一个二进位为1（0X80000000）。例如，考虑GDI32.DLL数组中一个具有值为0x80000112的IMAGE_ THUNK_DATA，该IMAGE_THUNK_DATA是引入来自于GDI32.DLL中的第0x112个引出函数。用序数来引入的问题是：微软不能在Windows NT、Windows 95和Win32之间使Win32 API函数的引出序数保持不变。

如果一个函数用名字来引入，则它的IMAGE_THUNK_DATA DWORD含一个RVA，该RVA是IMAGE_IMPORT_BY_NAME结构所要用到的。一个IMAGE_IMPORT_BY_NAME结构非常简单，看起来如下：

```
typedef struct _IMAGE_IMPORT_BY_NAME
{    WORD    Hint;
     BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Hint猜测是引入函数所用的引出序数之类的一个值。

BYTE[1]具有该引入函数名字的一个以NULL结尾的ASCII字符串。IMAGE_THUNK_DATA DWORD的最终解释是在PE文件被Win32装载器装入之后。Win32装载器使用IMAGE_THUNK_DATA DWORD中的原始信息来查阅引入函数（不管是用名字还是用序数引入的）的地址。装载器然后用引入函数的地址再改写该IMAGE_THUNK_DATA DWORD。

有IMAGE_IMPORT_DESCRIPTOR和IMAGE_THUNK_DATA结构了，现在很容易就可构造关于一个EXE或DLL使用的所有引入函数的报表。简单地在该IMAGE_IMPORT_DESCRIPTOR数组（它的每一元素对应一个引入的DLL）上反复进行如下操作就可达到目的：对每个IMAGE_IMPORT_DESCRIPTOR，找出IMAGE_THUNK_DATA DWORD数组的位置并适当地解释它们。下面显示了运行PEDUMP输出的结果（无名字的函数是用序数来引入的）。

```
  KERNEL32.dll
  Hint/Name Table: 00007050
  TimeDateStamp:     00000000
  ForwarderChain:    00000000
  First thunk RVA:  00007164
  Ordn  Name
   665  lstrcpynA
    23  CloseHandle
    79  DeviceIoControl
    48  CreateFileA
   653  lstrcatA
   293  GetShortPathNameA
   662  lstrcpyA
   331  GetVersion
   156  GetACP
   668  lstrlenA
   251  GetModuleFileNameA
   617  WideCharToMultiByte
   277  GetProcAddress
   598  VirtualAlloc
```

```
 359  HeapAlloc
 365  HeapFree
 630  WriteFile
 601  VirtualFree
 361  HeapCreate
 363  HeapDestroy
 297  GetStdHandle
 238  GetFileType
 534  SetHandleCount
 264  GetOEMCP
 162  GetCPInfo
 253  GetModuleHandleA
 226  GetEnvironmentStringsW
 150  FreeEnvironmentStringsW
 224  GetEnvironmentStrings
 425  MultiByteToWideChar
 149  FreeEnvironmentStringsA
 587  UnhandledExceptionFilter
 481  RtlUnwind
 398  LoadLibraryA
 210  GetCurrentProcess
 577  TerminateProcess
 106  ExitProcess
 169  GetCommandLineA
 295  GetStartupInfoA

USER32.dll
Hint/Name Table: 000070F8
TimeDateStamp:     00000000
ForwarderChain:    00000000
First thunk RVA:  0000720C
Ordn  Name
 374  LoadIconA
 435  PostQuitMessage
   9  BeginPaint
 182  EndPaint
  48  CheckMenuItem
 176  EnableMenuItem
 237  GetCursorPos
```

```
  501  SetForegroundWindow
  573  TrackPopupMenu
  128  DefWindowProcA
  300  GetSystemMetrics
  405  MessageBoxA
  387  LoadStringA
  382  LoadMenuA
  296  GetSubMenu
  370  LoadCursorA
  445  RegisterClassA
   85  CreateWindowExA
  556  ShowWindow
  591  UpdateWindow
  277  GetMessageA
  579  TranslateMessage
  144  DispatchMessageA
  136  DestroyIcon
  137  DestroyMenu
  433  PostMessageA

SHELL32.dll
Hint/Name Table:  000070F0
TimeDateStamp:     00000000
ForwarderChain:    00000000
First thunk RVA:  00007204
Ordn  Name
 101  Shell_NotifyIconA
```

这是本书中CoolCPU.exe的例子的导出。

第3章 Windows运行机理

3.5 PE结构分析(5)

3.5.4 PE文件引出

引入是引出的一个反过程，PE文件在.edata节中存储它引出函数的信息。

DLL/EXE要引出一个函数给其他DLL/EXE使用，有两种实现方法：通过函数名引出或者仅仅通过序数引出。比如某个DLL要引出名为“TextOut”的函数，如果它以函数名引出，那么其他DLLs/EXEs若要调用这个函数，必须通过函数名，就是TextOut。另外一个办法就是通过序数引出。什么是序数呢？序数是惟一指定DLL中某个函数的16位数字，在所指向的DLL里是独一无二的。例如在上例中，DLL可以选择通过序数引出，假设是16，那么其他DLLs/EXEs若要调用这个函数必须以该值作为GetProcAddress调用参数。这就是所谓的仅仅靠序数引出。

我们一般不提倡仅仅通过序数引出函数这种方法，这会带来DLL维护上的问题。一旦DLL升级/修改，程序员就无法改变函数的序数，否则调用该DLL的其他程序都将无法工作。

我们从前面知道，导出的数据位于.edata节中，在此节的开始处是一个IMAGE_EXPORT_DIRECTORY结构。紧随该结构的是由一个IMAGE_ EXPORT_DIRECTORY结构中的域指向的数据。一个IMAGE_EXPORT_ DIRECTORY看起来如下：

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;       // RVA from base of image
    DWORD   AddressOfNames;            // RVA from base of image
    DWORD   AddressOfNameOrdinals;    // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

- Characteristics：该域似乎没有被用到，并且总是被置为0。
- TimeDataStamp：该时间/日期印记指示该文件建立的时间。
- MajorVersion和MinorVersion：该域看起来也没有用，并且被置为0。
- Name：具有该DLL名的一个ASCII字符串的RVA。
- Base：被本模块引出的函数的起始引出序号。例如，如果文件用序数值20，21和22来引出其

函数，则本域的值是20。

- NumberOfFunctions：数组中元素个数。该值也是被本模块引出的函数个数。这个值通常和NumberOfNames域（见下一个描述）的相同，但它们也可以不同。
- NumberOfNames：在AddressOfFunctions数组中的元素个数。这个值包含了用名字来引出的函数个数，它通常（但不总是）和引出函数的总数相匹配。
- AddressOfFunctions：该域是一个RVA，并且指向一个函数地址数组。该函数地址是本模块中每个引出函数的RVA。
- AddressOfNames：该域是一个RVA，并且指向一个字符串指针数组。该串含的是从这个模块中通过名字来引出的函数的名字。
- AddressOfNamesOrdinals：该域是一个RVA，并且指向一个WORD数组。这些WORD基本上是从本模块中所有通过名字来引出的函数的引出序数。然而不要忘记加上在Base域中给出的起始引出序号。

引出一个函数所需要的是一个地址和一个引出序数。如果你用名字来引出函数，则应存在一个函数名。你应该想到PE格式的设计者会把这三个项放入一个结构中，并且再具有这些结构的一个数组。否则，你不得不在三个分离的数组中查询各个部分。

被IMAGE_EXPORT_DIRECTORY指向的数组的最重要部分，是由AddressOfFunctions域所指向的数组。它是一个DWORD数组，每个DWORD含一个引入函数的地址（RVA）。每个引出函数的引出序数对应于数组中它的位置。例如（假定序数起始值为1），具有引出序数为1的函数的地址将在该数组的第一个元素中，则引出序数是2的函数的地址存在该数组的第二个元素中，依次类推。

关于AddressOfFunctions数组,有以下两点要注意：

第一，引出序数需要把一个IMAGE_EXPORT_DIRECTORY中的Base域的值作为基准值。如果Base域值0，则AddressOfFunctions数组中的第一个DWORD对应引出序数10，第二项对应引出序数11，并且依次类推。

第二，引出序数可能有空白。让我们假定你明确地引出一个DLL中的两个函数，用序数值1和3。即使你只引出两个函数，但AddressOfFunctions数组不得不含三个元素。在该数组中任何不与一个引出函数相对应的项，其值都为0。

Win32 EXE和DLL更经常用名字而不是序数来引入函数。这正是另外两个数组要发挥作用的地方，这两个数组由一个IMAGE_EXPORT_ DIRECTORY结构中的值指向。AddressOfNames和AddressOfNames Ordinals数组是为了让装载器更快地找到与给定函数名相对应的引出序数。这两个数组都包含相同数目的元素（该数目由一个IMAGE_EXPORT_ DIRECTORY的NumberOfNames域给出）。该AddressOfFunctions数组是一个索引值的数组，该索引将用在AddressOfFunctions数组中。

引出表的设计是为了方便PE装载器工作。首先，模块必须保存所有引出函数的地址以供PE装载器查询。模块将这些信息保存在Address OfFunctions域指向的数组中，而数组元素数目存放在NumberOfFunctions域中。 因此，如果模块引出40个函数，则AddressOfFunctions指向的数组必定有40个元素，而NumberOfFunctions值为40。现在如果有一些函数是通过名字引出的，那么模块必定也在

文件中保留了这些信息。这些名字的RVAs存放在一数组中以供PE装载器查询。该数组由AddressOfNames指向，NumberOfNames包含名字数目。

考虑一下PE装载器的工作机制，它知道函数名，并想以此获取这些函数的地址。至今为止，模块已有两个部分：名字数组和地址数组，但两者之间还没有联系的纽带。因此，我们还需要一些联系函数名及其地址的内容。PE参考指出使用到地址数组的索引作为联接，因此，PE装载器在名字数组中找到匹配名字的同时，它也获取了指向地址表中对应元素的索引。 而这些索引保存在由AddressOfNameOrdinals域指向的另一个数组(最后一个)中。由于该数组起了联系名字和地址的作用，所以其元素数目必定和名字数组相同。比如，每个名字有且仅有一个相关地址，反过来则不一定: 每个地址可以有好几个名字来对应。因此我们给同一个地址取“别名”。为了起到连接作用，名字数组和索引数组必须并行地成对使用，譬如，索引数组的第一个元素必定含有第一个名字的索引，以此类推。

第3章 Windows运行机理

3.5 PE结构分析(6)

下面显示了对WS2_32.DLL引出节的PEDUMP输出。

```
  Name:            WS2_32.dll
  Characteristics: 00000000
  TimeDateStamp:     3A1B81FA
  Version:             0.00
  Ordinal base:       00000001
  # of functions:  000001F4
  # of Names:         0000006D

  Entry Pt  Ordn  Name
  0000CC51     1  accept
  00001E77     2  bind
  000013B6     3  closesocket
  0000C453     4  connect
  0000C553     5  getpeername
  0000C5FA     6  getsockname
  00001ABC     7  getsockopt
  00001E2E     8  htonl
  000012B0     9  htons
00007FFE     10  ioctlsocket
……
0000DB55    116  WSACleanup
00001BF5      151  __WSAFDIsSet
0000E180      500  WEP
```

我们知道导出有两种方法，一种是代名导出，一种是序号导出，那它们到底是一个怎样的过程呢？当用名字导出时，导出过程如下。

（1）定位到PE header。

（2）从数据目录读取引出表的虚拟地址。

（3）定位引出表获取名字数目(NumberOfNames)。

（4）并行遍历AddressOfNames和AddressOfNameOrdinals指向的数组匹配名字。如果在AddressOfNames 指向的数组中找到匹配名字，则从AddressOfNameOrdinals 指向的数组中提取索引值。例如，若发现匹配名字的RVA存放在AddressOfNames 数组的第54个元素，那就提

取AddressOfNameOrdinals数组的第54个元素作为索引值。如果遍历完NumberOfNames 个元素，则说明当前模块没有所要的名字。

（5）从AddressOfNameOrdinals 数组提取的数值作为AddressOfFunctions 数组的索引。也就是说，如果值是5，就必须读取AddressOfFunctions 数组的第5个元素，此值就是所要函数的RVA。

序号导出方法的过程如下。

（1）定位到PE header。

（2）从数据目录读取引出表的虚拟地址。

（3）定位引出表获取nBase值。

（4）减掉nBase值得到指向AddressOfFunctions 数组的索引。

（5）将该值与NumberOfFunctions作比较，大于等于后者则序数无效。

通过上面的索引就可以获取AddressOfFunctions数组中的RVA了。

可以看出，从序数获取函数地址比函数名快捷容易。不需要遍历AddressOfNames 和 AddressOfNameOrdinals 这两个数组。然而，综合性能必须与模块维护的简易程度作一平衡。

总之，如果想通过名字获取函数地址，则需要遍历AddressOfNames 和 AddressOfNameOrdinals 这两个数组。如果使用函数序数，减掉nBase值后就可直接索引AddressOfFunctions 数组。

3.5.5 PE文件资源

在PE文件中寻找资源比较复杂，当寻找它们时，需要遍历一个复杂的层次结构才能找到。

资源目录结构很像磁盘的目录。它有一个主目录（根目录），主目录含有子目录，子目录还可有它自己的子目录。在这些子目录中，可以找到文件。资源目录的数据结构格式如下：

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    WORD    NumberOfNamedEntries;
    WORD    NumberOfIdEntries;
   //  IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[];
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

- Characteristics：理论上讲，这个域应该保存该资源的标记，但看起来它总是为0。
- TimeDataStamp：这个时间/日期印记描述该资源创建时间。
- MajorVersiont和MinorVersion：理论上讲，这些域保存该资源的版本号。但这些域似乎总被置为0。
- NumberOfNameEntries：使用名字并且跟在本结构之后的数组元素（稍后描述）的数目。
- NumberOfIdEntries：使用整数ID并且跟在结构和任何有命名的项之后的数组元素的数目。
- IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]：该域形式上并不是IMAGE_RESOURCE_DIRECTORY_ENTRY结构的部分，而是紧跟其后的一

个IMAGE_RESOURCE_DIRECTORY_ ENTRY结构数组。该数组中元素个数是NumberOfNameEntries和NumberOfIdEntries域之和。含有名称标志符（而不是整数ID）的目录项元素位于数组的最前面。

在资源中，一个目录项既可指向一个子目录（即另一个IMAGE_ RESOURCE_DIRECTORY_ENTRY），也可指向一个IMAGE_RESOURCE_ DIRECTORY_ENTRY。当目录是指向一个目录数据时，它将描述在文件中什么地方可以找到资源的生数据。

一般情况下，在你到达一个给定资源的IMAGE_RESOURCE_DATA_ ENTRY之前，至少有三个目录层。最顶层目录（只有一个）总位于资源节（.rsrc）的开始处。最顶层目录的子目录对应于文件中能找到的资源的各种类型。

例如，如果一个PE文件包括对话框、字符串表和菜单，则这三个子目录将会是一个对话目录、一个字符串表目录和一个菜单目录。每一个这样的“类型”子目录将轮流具有“ID”子目录。对一种给定资源类型的一个实例，将有一个ID子目录。图3.12中我们以一个更易理解的可视形式显示了资源目录层次结构。

第3章 Windows运行机理

3.5 PE结构分析(7)

下面显示的是用PEDUMP来DUMP出的CPUCOOL.EXE中的资源的输出。见交叉状结构的第二层，可以看到其中有图表、菜单、对话、字符串表、组图标和版本资源。在第三层上，有一个图标、一个图标组、一个菜单以及其他等资源。

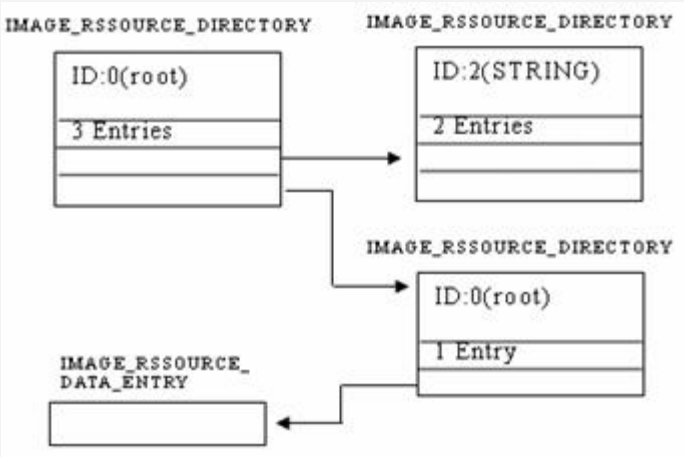


图3.12 资源目录层次结构

```
ResDir (0) Named:00 ID:04 TimeDate:00000000 Vers:0.00 Char:0
   ResDir (ICON) Named:00 ID:01 TimeDate:00000000
Vers:0.00  Char:0
        ResDir (1) Named:00 ID:01 TimeDate:00000000
Vers:0.00 Char:0
            ID: 00000804  DataEntryOffs: 00000110
            Offset: 172B0  Size: 002E8  CodePage: 0
    ResDir (MENU) Named:00 ID:02 TimeDate:00000000
Vers:0.00 Char:0
        ResDir (68) Named:00 ID:01 TimeDate:00000000
Vers:0.00 Char:0
            ID: 00000804  DataEntryOffs: 00000120
            Offset: 175B0  Size: 00054  CodePage: 0
        ResDir (69) Named:00 ID:01 TimeDate:00000000
Vers:0.00 Char:0
            ID: 00000804  DataEntryOffs: 00000130
            Offset: 17608  Size: 00036  CodePage: 0
    ResDir (STRING) Named:00 ID:01 TimeDate:00000000
```

```
Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:00000000
Vers:0.00 Char:0
            ID: 00000804  DataEntryOffs: 00000140
            Offset: 17640  Size: 00068  CodePage: 0
    ResDir (GROUP_ICON) Named:00 ID:01 TimeDate:00000000
Vers:0.00 Char:0
        ResDir (66) Named:00 ID:01 TimeDate:00000000
Vers:0.00 Char:0
            ID: 00000804  DataEntryOffs: 00000150
            Offset: 17598  Size: 00014  CodePage: 0
```

每个资源目录项是一个IMAGE_RESOURCE_DIRECTORY_ENTRY类型的结构。每个IMAGE_RESOURCE_DIRECTORY_ENTRY具有如下格式:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
        struct {
            DWORD NameOffset:31;
            DWORD NameIsString:1;
        };
        DWORD   Name;
        WORD    Id;
    };
    union {
        DWORD   OffsetToData;
        struct {
            DWORD   OffsetToDirectory:31;
            DWORD   DataIsDirectory:1;
        };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

- Name：该域包含的既可以是一个整数ID，也可是指向含一个字符串名字的结构的指针。如果高位（0x80000000）是0，则该域被解释为一个整数ID。如果高位非零，则低的31个二进制数是相对于一个IMAGE_RESOURCE_DIR_STRING_U结构的偏移量（相对资源节的开始处）。这个结构含一个WORD字符计数，后跟一个具有资源名称的单一码字符串。使得即使是为非单一码Win32而设计的PE文件，也在这里使用单一码。要把该单一码字符串转换为一个ANSIC字符串，请见WideCharToMultiByte（）函数。
- OffsetToData：该域既可是相对于另一个资源目录的一个偏移量，也可是指向关于一个特定

资源实例的信息的一个指针。如果高位（0x80000000）被置为1，则该目录项对应一个子目录，低31个二进制数是一个相对于另一个IMAGE_RESOURCE_DIRECTORY结构的偏移量（相对于资源的开始处）。如果高位置为0，则低31位是一个相对于一个IMAGE_RESOURCE_DATA_ENTRY结构的偏移量（相对于该资源节）。IMAGE_RESOURCE_DATA_ENTRY结构包含了资源的生数据的位置、它的尺寸和它的代码页。