CD-ROM INCLUDED

如何开发 Windows 95 应用程序

[美] Kevin J. Goodman 著

郭 勇 译

学习如何将现有的程序转换到 Windows 95 环境

给出有关 32 位程序设计的高效指南

学习如何充分利用网络支持

学习用多个应用程序进行内存管理

清华大学出版社

如何开发 Windows 95 应用程序

[美] Kevin J. Goodman 著
郭 勇 沈 龙
赵 军 赵 华 译
陈 隆 审校

清华大学出版社

(京)新登字 158 号

如何开发 Windows 95 应用程序
Building Windows 95 Applications

图书在版编目(CIP)数据

如何开发 Windows 95 应用程序/(美)Goodman, K. J. 著；郭勇等译. —北京：清华大学出版社，1996. 5
书名原文：Building Windows 95 Applications
ISBN 7-302-02261-5

Ⅰ. 如… Ⅱ. ①G… ②郭… Ⅲ. 软件开发 Ⅳ. TP311. 52

中国版本图书馆 CIP 数据核字(96)第 12887 号

出版者：清华大学出版社(北京清华大学校内，邮编 100084)
印刷者：清华大学印刷厂印刷
发行者：新华书店总店北京科技发行所
开 本：787×1092 1/16 印张：25. 75 字数：640 千字
版 次：1996 年 10 月第 1 版 1996 年 10 月第 1 次印刷
书 号：ISBN 7-302-02261-5/TP・1105
印 数：0001—8000
定 价：38. 00 元 附光盘书定价：63.00元

目　录

第一章 32位Windows介绍

对一些人来说，Windows 95的介入恰恰是在广大的程序员(和无数的用户)对Windows 3.1日渐得心应手的时候。但是，对于软件产业来说，创新就意味着要在用户尚未意识到他们需要它之前开发出能够满足这种需求的新产品。如果读者是众多的Windows 3.1的开发人员之一，那么一定知道Windows 95的推出已经有些迟了。如果不是一名Windows 3.1的开发人员，那也许是因为已经意识到了Windows 3.1的一些缺陷。不论怎样，本章将详细介绍16位计算的缺陷以及32位计算是如何克服这些缺陷的，从而向读者说明32位操作系统并非仅仅是计算机科学领域的一个尝试。本章还将概要介绍32位操作系统在编程上与以往的操作系统的细微差别，并将对Windows 95和Windows NT这两种操作系统做专门介绍。

由于编写能在32位操作系统上运行的代码只是整体工作中的一小部分，本章同时将对Intel和RISC两种平台做一些解释，以便读者在程序出现问题的时候能够对其进行调试。

1.1 32位计算的优点

32位处理器及操作系统较之16位处理器有多方面的优越性，显而易见的优点可以归纳为以下四种：

- 线性编程模式取代了分段编程模式；
- 更大的数据和可用地址空间；
- 不再必需复杂的编译器支持；
- 处理器被优化为32位模式。

我们将逐一介绍这些优点。

1.1.1 线性编程模式

32位处理器的一个主要优点是它采用了线性编程模式。与线性编程模式关联的内存模式有时被称作平面模式(Flat model)。平面模式与分段模式中的微模式(Tiny model)类似。平面模式对Intel系列处理器来说并不陌生，它只是由16位模式转换到32位模式所获得的一种益处。Windows 3.1在编写设备驱动程序时提供了平面模式。为了访问平面模式中的一个内存位置，只需使用偏移量即可。因为寄存器是32位的，所以平面模式下的内存寻址范围是4GB。在平面模式中，段寄存器对于开发人员不再那么重要了。我们常可以听到，在Intel系统的平面模式中，SS、DS和CS寄存器总是相等的(即SS==DS==CS)。无论对于何种使用目的，这一点均成立，因为三个段寄存器都指向某一个相同的物理地址空间。不过，在调试过程中，会发现SS、DS和ES寄存器的内容总是相同的，而CS寄存器却总有一个不同的值。基于RISC的机器只支持32位模式，有些可以支持4位模式。

使这些段寄存器不再重要的原因是，在用户模式下运行时，应用程序无法改变这些寄存

器的值(在 NT 中,有两类进程:无优先权的用户模式和有优先权的内核模式。本书的实例均是用户模式进程)。虽然 Windows 95 不是真正的客户服务器类型的操作系统,但用户模式这个术语同样可以适用。CS、DS、ES 和 SS 的值是由内核(它在内核模式下运行)在创建用户模式进程时决定的。这就是 Windows 95 和 NT 分割地址空间的方法。同样的偏移量在进程 A 和进程 B 中可以得到不同的物理内存位置。作为一名开发人员,可以完全忘记段寄存器。这一点更适用于 MIPS、Power PC 和 Alpha 处理器,因为它们根本没有段寄存器!

1.1.2 更大的数据和可用地址空间

平面模式有另一个优点——它不需要编译器支持内存模式。多数 16 位编译器一般支持四种或五种不同的内存模式:

- 微模式:在这种模式下,代码和静态数据(包括堆栈)都必须可以存放在一个 64KB 的段里。多数保护模式程序不能支持这种模式。因此,许多编译器不对这种模式提供支持。如果一个编译器支持微模式,那么所得程序为 COM 程序(由.COM 扩展名得名)。
- 小模式:小模式程序有一个代码段和一个独立的数据段,它们均必须小于 64KB。
- 中模式:中模式程序可以包括多个代码段(它们中的每一个都必须小于 64KB)和一个数据段。
- 紧凑模式:与中模式相反,紧凑模式的程序只有一个代码段而有多个数据段。
- 大模式:大模式程序可有多个代码段和多个数据段。
- 巨模式:巨模式程序基本与大模式相同,但是它有创建大于 64KB 数据对象(如数组)的能力。

在 Win32 中,这些内存模式不复存在。不过,在 Win32 中仍存在多种模式编程的痕迹,其中之一就是 FAR 和 NEAR 关键字(FAR 和 NEAR 实际上是类型定义关键字。ANSI 标准需要_near 和_far)。FAR 和 NEAR 可以帮助用户越过内存模式的障碍。例如,在中模式程序中,可以通过使用关键字 FAR 来访问缺省数据段以外的数据。关键字 FAR 指示编译器通过选择器和偏移量来引用一个变量而不是只使用 16 位偏移量。

在大模式、紧凑模式和巨模式中,数据的缺省类型是 FAR。为了访问大模式、紧凑模式和巨模式中缺省数据段中的数据,16 位的程序员需使用 NEAR 关键字。同样的道理也适用于代码段:微模式、小模式和紧凑模式中的缺省类型是 NEAR;而中模式、大模式和巨模式中的缺省类型是 FAR。为了具有兼容性,Win32 仍支持 FAR 和 NEAR,不过它们已没有任何意义了。所以,如果你看到别人的代码中包含 FAR 指针和 FAR 函数,不要惊慌。该段代码的编写者可能是在设法与 Win16 保持源代码级兼容;或者,他可能是一位尚未改变分段操作系统编程习惯的老的 Win16 程序员。

Windows 95 和 NT 中所有类型的缺省值均为 NEAR(与微模式中相同),因为在引用对象和调用函数时只需使用一个偏移量。

消除了 16 位计算中 64KB 的障碍是 32 位系统的另一个显著优势。由于每个进程中代码和数据的存储空间为 2GB,NT 和 Windows 95 可以更高效率地支持大于 64KB 的数组等数据对象,为大于 64KB 的图形和代码等分配空间。在巨模式程序中,16 位的编译器也可以

支持这些方式。不过,速度要慢得多,下面一节将讲述其原因。

1.1.3 不需要复杂的编译器支持

在 Microsoft Systems Journal 1992 年 7 月刊中的"The Case for 32 Bits"一文中,Charles Petzold 说明了为什么一个程序的 16 位版本在运行速度上几乎比同一程序的 32 位版本慢了五倍的原因。执行速度缓慢的原因是,16 位编译器需要做许多额外的工作才能执行 32 位的操作。在 32 位模式下,编译器(或用户)不需要产生额外的代码去管理分散在多个段中的对象。请看下面这段代码范例,它执行的操作是将两个数求和,然后调用一个虚构的用于求平均值的名为 avg 的函数:

```
void avg(int *, int *, int *);
int a,b,c;
void main(void)

c=a+b;
avg(&a,&b,&c);
```

在按小模式程序编译时,用 Microsoft C 8.00 产生的 16 位汇编语言如下:

```
_main    PROC NEAR
    mov  ax,WORD PTR _b
    add  ax,WORD PTR _a
    mov  WORD PTR _c,ax
    push OFFSET DGROUP:_c
    push OFFSET DGROUP:_b
    push OFFSET DGROUP:_a
    call _avg
    add  sp,6
    ret

_main    ENDP
```

然而,如按 Large 模式编译,16 位的汇编语言代码膨胀成了这样:

```
_main    PROC FAR
; Line 4
; Line 5
    mov  es,WORD PTR $T106
    mov  ax,WORD PTR es:_b
    mov  es,WORD PTR $T107
    add  ax,WORD PTR es:_a
    mov  es,WORD PTR $T108
    mov  WORD PTR es:_c,ax
; Line 6
    push es
```

```
        push  OFFSET _c
        push  SEG _b
        push  OFFSET _b
        push  SEG _a
        push  OFFSET _a
        call  FAR PTR _avg
        add   sp,12       ;000cH
; Line 7
        ret
```

如果从来不看汇编语言的输出，那么也许永远也不会知道这个程序的大模式版本竟比小模式版本多了约 15%。因为变量 a,b,c 分别定位在不同的数据段中，所以 16 位编译器必须产生附加代码来操纵这些段。这个道理对于 avg 函数调用也适用。就编译器所知，avg 是一个独立的代码段，因此编译器就必须为此做些调整。如果实际上是我们编译并链接了这个简单的程序，那么我们也可以看出这两个程序在磁盘上所占空间的不同，如清单 1-1 所示。在 Win32 中由于只有一种模式，因此编译器不需要做额外的支持。

清单 1-1　16 位程序由于内存模式不同而占用不同的空间

```
TEST        EXE        2875 9-29-94 9:44a
TESTFAR     EXE        3257 9-29-94 9:44a
             2 file(s)  6132 bytes
```

1.1.4　优化为 32 位模式的处理器

OS/2 最初的带有挖苦性的口号是“比 Windows 更好的窗口系统”。IBM 设计这条营销标语是为了将操作系统购买者的视线从 Windows 3.1 上吸引走。这也许误导了某些开发人员，因为对现实更好的描绘是，32 位的解决办法比 16 位的好。换句话说，不要劝说开发人员为 32 位操作系统去开发 16 位的应用程序，干脆劝说他们去直接开发 32 位的应用程序。

同样的方案也适用于 Windows 95 和 Windows NT。它们所支持的处理器在 32 位模式下运行比在 16 位模式下运行的速度快。不仅如此，在诸如 MIPS R4x00 和 Alpha AXP 等的处理器上，相应的 32 位应用程序比起 16 位的应用程序有令人难以置信的巨大优势，因为处理器必须通过一种仿真模式才能运行 16 位的软件。如果注重的是程序的性能，那么一个 32 位版本的 Windows 应用程序在 NT 上的运行速度将比 16 位版本更快。

这一点在 16 位版本需要使用大模式时尤为正确。特别使我感到不安的一件事便是看到有人在不必要的情况下，在 16 位 Windows 上使用大模式。很多时候，一个缺乏经验的程序员是意识不到这样使用大模式所带来的损害的，毕竟其 C 源代码与其它模式看起来没有什么不同。如果静态数据不到 64KB，那么在 Windows 3.1 上使用大模式是不必要的。在 NT 中，开发人员则可以不必为此多费心思。

不幸的是，对 Windows 3.1 的程序来说，有些情况下大模式的使用是不可避免的。在 Windows 3.1 中驻留在 DLL 中的所有 Microsoft Foundation Class(MFC)类必须用大模式编译，这又是由于 16 位系统的复杂编程模式造成的，因为它规定在 DLL 中堆栈段与数据段不

等(即SS! =DS)。

即使开发人员设法避免在Windows 3.1中使用大模式,FAR调用和FAR数据仍然会使性能降低,选择器加载的开销尤其大。当在其它段中调用例程或者在多个段中操纵数据时,发生选择器加载。可由下表比较一下16位NEAR调用和16位FAR调用在Intel x86系统上所用时间的不同。

调用	时间
NEAR调用	7个时钟周期
对内存中段的FAR调用	28个时钟周期

这在Windows 3.1中是非常普遍的,因为多数程序使用中模式。为了避免在Windows 3.1上使用不必要的选择器调入,开发人员有时人工优化代码,以便把那些经常互相调用的函数放在同一个段中,形成一个准NEAR调用。

在Windows 3.1的标准模式中,程序员还要确保各段不要有太大的责任。不像386增强模式那样可以将内存分页到在4KB的簇中的一个交换设备上去,标准模式只能将整个段交换出去。于是产生了Windows 3.1的第三方程序,它统一了代码段的大小。它的思想是:如果所有段是同样大小的,那么标准模式中的Memory Manager(内存管理器)则可以随意地控制各段的交换,而无需寻找一个足够大的空间来放置该段。

64KB的障碍不仅限制了代码段和数据段,算术运算同样受到了16位模式的限制。变成32位模式以后,整数的算术运算性能将得到提高,因为编译器不再需要使用临时的16位寄存器来仿真32位模式了。

不过,在Win32中不再有这些麻烦。在Win32中,如果是在编写全新的代码,那么可以完全忘记NEAR和FAR以及64KB的约束。如果是在从16位Windows移植代码,那么那些NEAR和FAR也无碍于事。

值得一提的是,大模式有一点优越性。因为大模式程序中的指针缺省值是32位的,所以将大模式代码移植到32位模式的工作量便不像移植中模式程序那么大(因为必须保证所有指针都从16位变到了32位)。

1.2 了解目标平台

许多16位计算固有的缺陷程序员是看不出来的。如果不花点时间来看看16位编译器的编译结果,那么也许会开发出一段执行效果很不理想的程序,而又不知其中的原因。这一问题在Win32中同样存在。因为虽然Windows 95和Windows NT使用32位模式,并且32位模式可以解决16位计算的一些问题,但这并不能作为可以忽视其底层结构的借口。尤其是现在每一种新的开发工具的问世,似乎都使开发人员和他们正在使用的机器离得愈来愈远了。有了诸如Windbg或Visual C++ 2.0(见图1-1和图1-2)等程序,可以很容易地与所使用的机器相互独立开来。

这一点当在一个图形环境(如Windows)编程时尤为正确。可以设想这样一种方案,一个比较聪明但又不熟悉Windows 95的人学习了"Power Objects"(一个假想的编程环境,它可

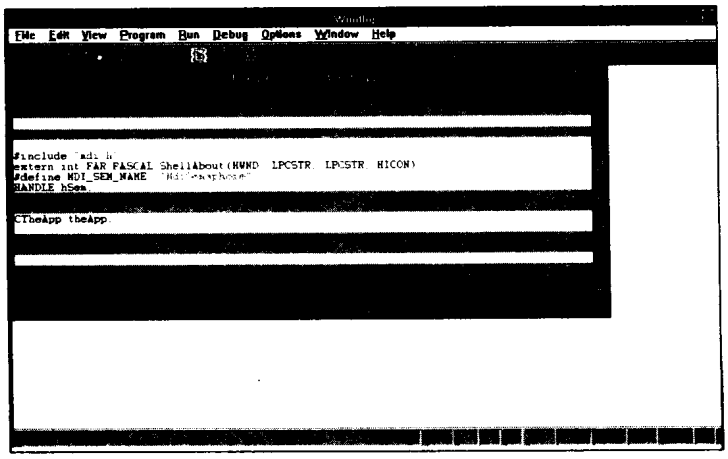

图 1-1 Windbg，一种 GUI 源代码级调试程序

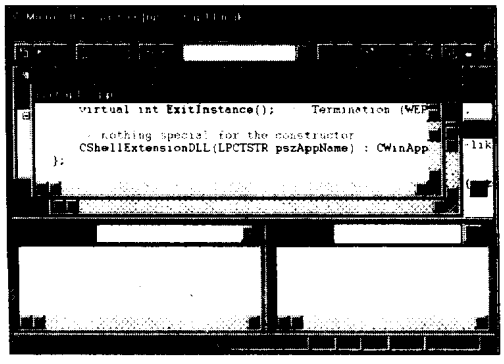

图 1-2 Visual C ++ 2.0≠GENERIC

以为开发人员生成代码）之后就不再学习其它任何关于该图形环境的知识了，那么几天之后，这个人就可以交付能够产生一些可爱的简单图形的程序原型。不幸的是，几周之后，这个人所能交付的仍旧是这些程序原型。为什么呢？这当然不是 Power Objects 的问题。在一个有经验的 DOS/Windows 程序员手中，Power Objects 可以产生的远不止这些。但是一种高级工具只能为用户服务到这种程度。当碰到这种工具的限制时，除非懂得它是如何工作的，否则便不能更好地利用它。换句话说，在一个初学者手中，每项工具都有其不可超越的界限。

许多成功的 Windows 3.1 的开发人员都会告诉你，他们成功的部分原因是他们对基础硬件了如指掌。对于 Win32 的开发人员来说，这一点同样适用。

不过，请不要误解我的意思。我不是在抨击“Power”之类的语言。事实上，本书中的实例是用 Visual C ++ 创建的。然而，这有很大的不同。一旦懂得了 Win32 API 并且熟悉了 Windows 95 和 NT 的基础体系结构，就可以对存在问题的地方进行调试或者可以超越 Visual C ++ 的极限。

如果不了解编译器对代码做了哪些工作，那么便不可能写出高效率的程序。如果不知道机器希望看到怎样的程序，那就只能听由编译器摆布了。观察编译器对源代码做了哪些工作

有两种办法，一种是检查 MAP 文件(参阅本章 1.4“如何阅读 MAP 文件”)，另一种办法是学会看编译器生成的汇编语言。

对了——就是汇编语言。人们已经讨论了这么多关于 NT 本身是如何几乎全部用 C/C++写成的，你可能因此已不再想使用汇编器了(Windows 95 是用 C 语言和汇编语言写的)。因为大多数编译器可以产生并读懂汇编语言(利用_inline 关键字)，把编译器丢在一边是一个好主意，不过可不要丢弃汇编语言知识。我可以理解，忘掉汇编技能对开发人员有很大的诱惑。除非是在 Win32 环境下编程，否则你的代码对其它平台没有可移植性。而这种可移植性又是通过写 C/C++代码获得的，而不是汇编语言。有了 Visual C++中可作源代码级调试的调试程序，更增加了忘记汇编语言的诱惑力。但我几乎从来不在源代码级调试程序。因为如果是我写了源代码，那么我应该知道它的功能。我想知道的是，在编译器看来，我的源代码的功能是怎样的。所以，最好的协调办法是用 C/C++编写源代码，同时懂得基础硬件的汇编语言。

Windows NT 使编程人员得到的程序是那些源代码与 MIPS，Intel，Power PC 和 Alpha 处理器平台(或者其它可用的平台)有兼容性的程序。在我写本书的时候，还没有可用的交叉编译器。这也就是说，如果想提供一个 Alpha 版本的软件，那么就必须先有 Alpha 处理器，并准备好在 Alpha 处理器上进行调试。如果想提供一个 MIPS 版本的软件，就必须最终在一个有 MIPS 处理器的机器上调试。如果想提供一个 Intel 版本的软件——那么我想你一定知道应如何去做。

1.2.1　判断基本类型的长度

在任何一个想做开发工作的操作系统上，了解基本数据类型的长度都是很重要的，因为这些基本数据类型的长度是独立实现的。表 1-1 列出了 Win32 中的基本类型及其长度。

表 1-1　Win32 平台上的基本类型长度

类型	长度
char(字符型)	8 位(一个字节)
short(短字)	16 位
int(整型)	32 位
long(长字)	32 位
float(浮点型)	32 位(IEEE)
double(双字)	64 位(IEEE)
ULONG；	32 位
USHORT；	16 位
UCHAR；	8 位
DWORD；	32 位
BOOL；	32 位