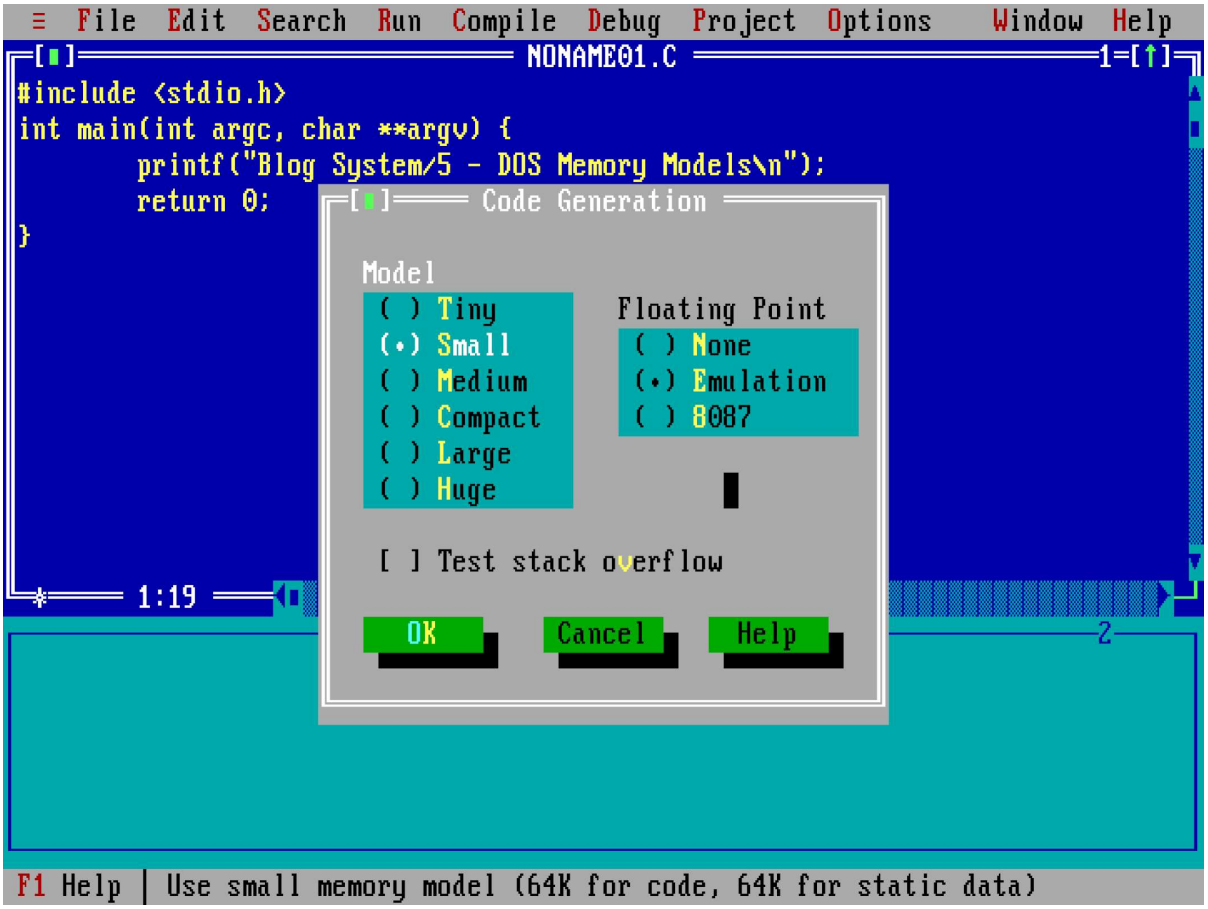
> 原文链接：https://blogsystem5.substack.com/p/dos-memory-models
>
> 作者：Julio Merino
>
> 发表日期：Sep 30, 2024

重新审视 DOS 内存模型

年初，我写了一些关于DOS 如何克服 x86 实模式内存限制的文章，介绍 DOS 如何克服 x86 实模式内存限制的技巧的文章。有一个问题出现了，但一直没有得到解答：当时的编译器提供了哪些不同的“模型”？看一下 Borland Turbo C++ 的代码生成菜单：

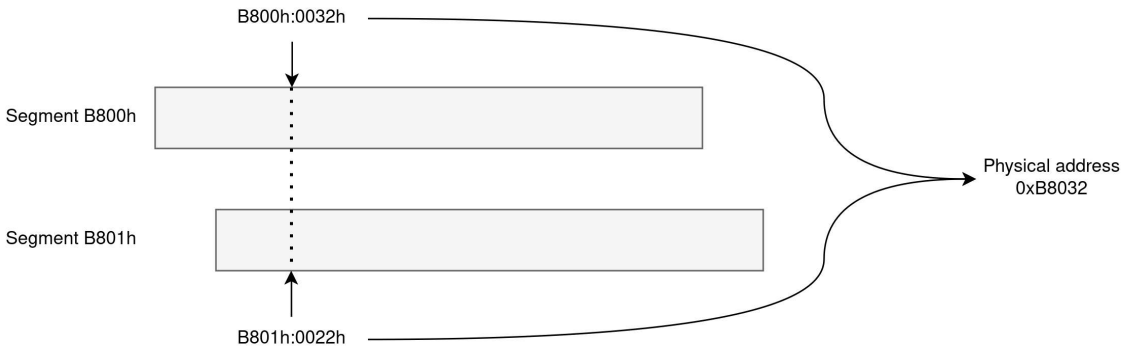


Tiny，small，medium，compact，large，huge……这些选项是什么意思？它们的作用是什么？更重要的是……在当今 64 位机器和数 GB 内存的世界中，这些遗留物还有意义吗？为了回答这些问题，我们必须先简要回顾一下 8086 架构和 DOS 支持的二进制格式。

8086 分段

在 8086 架构中（DOS 的目标架构），内存引用由两部分组成：一个 **2 字节段** “标识符”和一个 **2 字节偏移量** 在段内。这些对通常表示为 。

段是连续的 64KB 内存块，并由其基地址标识。为了能够寻址 8086 支持的全部 1MB 内存，段之间偏移 16 字节。由此可以推断，段是重叠的，这意味着一个特定的物理内存位置可以由多个段/偏移量对引用。



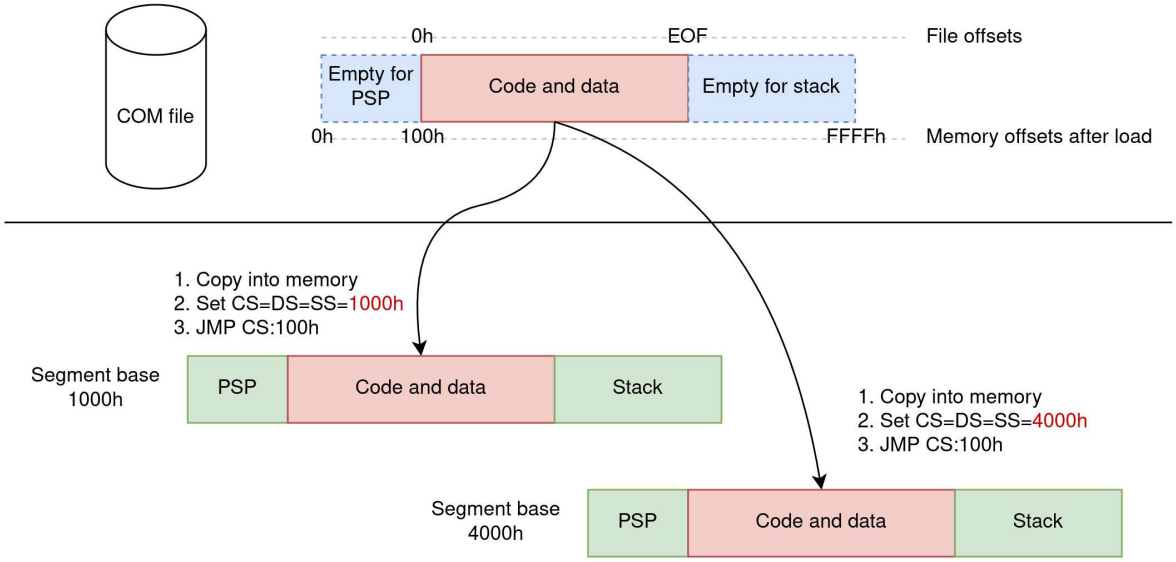
例如，分段地址 `B800h:0032h` 对应于物理地址 `B8032h`，通过 `B800h * 10h + 0032h` 计算得出。虽然这对人类来说是可读的，但这并不是机器级指令对其进行编码的方式。相反，指令依赖于 **段寄存器** 来指定要访问的段，8086 支持其中的四个：CS（代码段）、DS（数据段）、ES（额外数据段）和 SS（堆栈段）。了解这一点后，访问此示例内存位置需要首先将 `B800h` 加载到 `DS` 中，然后引用 `DS:0032h`。

指令依赖于段寄存器而不是每次内存访问都使用段标识符的原因之一是效率：编码要使用的段寄存器只需要 2 位（我们总共有 4 个段寄存器），而存储段基地址则需要 2 字节。稍后会详细介绍。

COM 文件

COM 文件是您可以想到的最简单的可执行文件格式：它们包含原始机器代码，可以放置在几乎任何内存位置，并且无需任何后处理即可执行。没有重定位，没有共享库，没有任何需要担心的：您可以将二进制文件直接复制到内存中并运行它。

它的工作方式是利用 8086 分段架构：将 COM 映像加载到*任何*段中，始终位于该段内的偏移量 100h 处。COM 映像中的所有内存地址都必须相对于此偏移量（这解释了您可能在过去看到的 `ORG 100h`），但映像不需要知道它被加载到哪个段中：加载器（在本例中是 DOS，但 COM 文件实际上来自 CP/M）将 CS、DS、ES 和 SS 设置为完全相同的段，并将控制权转移到 `CS:100h`。



神奇！COM 文件本质上是 PIE（位置无关的可执行文件），无需内核进行任何 MMU 或花哨的内存管理。

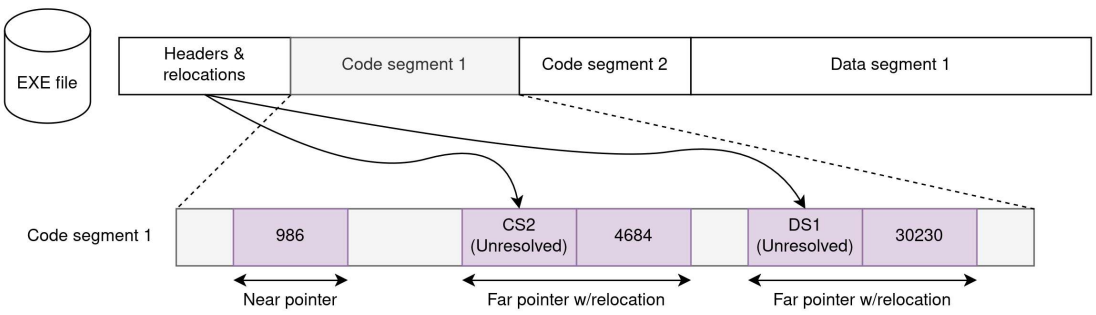
不幸的是，并非一切都那么美好。COM 文件的问题在于它们的大小有限：因为它们被加载到一个段中，而段最多为 64KB 长，所以 COM 文件最大只能是 64KB（减去前面为 PSP 保留的 256 字节）。这包括代码*和*数据，而 64KB 对两者来说都不多。显然，当 COM 程序运行时，它可以自由支配处理器，并且可以通过重置 CS、DS、ES 和/或 SS 来访问其单个段之外的任何内存，但所有内存管理都留给了程序员。

EXE 文件

为了解决 DOS 中 COM 文件的限制，微软为 DOS 设计了一种不同的可执行文件格式：EXE 文件，也称为 MZ 可执行文件。

与 COM 文件相比，EXE 文件具有一些内部结构，并且不受 64KB 限制的约束：它们可以包含更大的代码和数据块。但是……鉴于 8086 段最多仍然是 64KB 长，这怎么可能呢？答案很简单：EXE 文件包含*多个*段，并将代码和数据分布在它们上面。

为了在运行时支持多个段，EXE 文件在其标头中包含重定位信息。从概念上讲，重定位告诉加载器二进制映像中的哪些位置包含“不完整”的指针，这些指针需要在将段加载到内存后用段的基地址进行修复。DOS 作为加载器，负责进行此修补。



那么，EXE 中有多少个段呢？这取决于具体情况，因为并非所有程序都有相同的需求。有些程序整体很小，可以放在 COM 文件中。其他程序包含大量数据，但代码很少。还有一组程序包含大量代码和数据。等等。

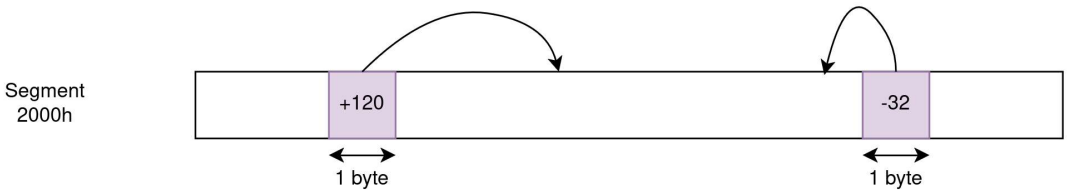
所以问题就变成了：通用的 EXE 格式如何以有效的方式支持这些选项？这就是内存模型变得重要的地方，但要谈论这些，我们必须绕道讨论指针类型。

指针类型

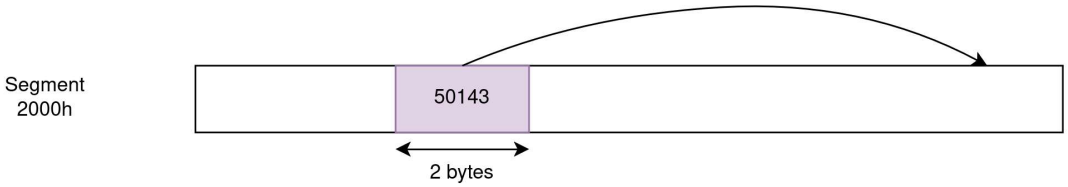
局部性原理指出“处理器倾向于在短时间内重复访问同一组内存位置”。这很容易理解：代码通常几乎按顺序运行，数据通常打包在连续的内存块中，如数组或结构体。

因此，将*所有*内存地址表示为 4 字节　　对将是一种浪费，这就是 8086 的分段再次对我们有利的地方。我们可以首先将一个段寄存器加载“我们所有数据”的基地址，然后我们只需要将地址记录为该段内的偏移量即可。我们重新加载段寄存器的次数越少越好，因为我们在每条指令和每个内存引用中携带的信息就越少。

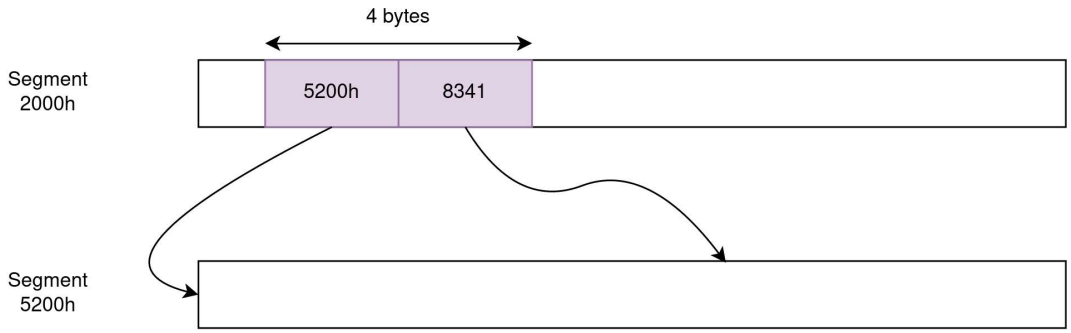
但是我们不能总是使用单个段内的偏移量，因为我们可能要处理多个段。而且偏移量有各种大小，所以对它们都使用唯一的大小也将是一种浪费。这意味着内存地址，或者*指针*，需要有不同的形状和形式，每种都最适合特定的用例。



短指针只占用 1 字节，表示一个*相对*地址，相对于正在执行的指令。它们在跳转指令中特别有用，以保持其二进制表示紧凑：跳转出现在每个条件或循环中，并且在许多情况下，条件分支和循环体非常短，以至于最小化表达这些分支点所需的代码量是值得的。



近指针可以引用“上下文”隐含的 64KB 段内的地址，并且长度为 2 字节。例如，像 `JMP 12829h` 这样的指令通常不需要携带有关此地址引用的段的信息，因为代码跳转几乎总是在发出跳转的代码的同一 CS 内。类似地，像 `MOV AX, [5610h]` 这样的指令假定给定地址引用当前选择的 DS，这样它就不必每次都表达段。近指针编码的偏移量可以是相对的或绝对的。



远指针可以通过编码段和偏移量来引用任何内存地址。它们长度为 4 字节。当用于指针算术时，段保持固定，只有偏移量变化。例如，这在迭代数组时很重要，因为我们可以将基地址加载到 DS 或 ES 中一次，然后操作段内的偏移量。但是，这意味着此类迭代的最大范围为 64KB。

巨指针类似于远指针，因为它们也是 4 字节长，并且可以引用任何内存地址，但它们消除了指针算术周围的 64KB 限制。它们通过在每次内存访问时重新计算段和偏移量部分来做到这一点（请记住，段是重叠的，因此我们可以为任何物理地址想出多个段/偏移量对）。可以想象，这需要在每次内存访问时进行额外的代码，因此巨指针会对运行时造成明显的负担。

内存模型

现在我们已经了解了 8086 分段、EXE 文件和指针类型……我们终于可以将所有这些概念联系在一起，揭开我们在 DOS 的旧编译器中看到的内存模型的神秘面纱。

以下是细分：

- **微型**：这是 COM 映像的内存模型。整个程序适合一个 64KB 段，并且所有段寄存器在启动时都设置为这一个段。这意味着程序中的所有指针都是短指针或近指针，因为它们始终引用这同一个 64KB 段。
- **小型**：到处使用近指针，但数据段和堆栈段与代码段不同。这意味着这些程序有 64KB 用于代码，64KB 用于数据。
- **紧凑型**：对代码使用短指针，但对数据使用远指针。这意味着这些程序可以使用全部 1 MB 内存空间用于数据，因此，它对于代码尽可能紧凑同时能够加载和引用内存中所有资源的游戏特别有用。
- **中等型**：与紧凑型相反。对代码使用远指针，但对数据使用短指针。这种模型很奇怪，因为如果您有一个代码量很大的程序，它很可能也是处理大量数据的程序。
- **大型**：到处使用远指针，因此代码和数据都可以引用完整的 1 MB 地址空间。但是，由于远指针的性质，所有内存*偏移量*最多为 64 KB，这意味着数据结构和数组的大小受到限制。
- **巨型**：到处使用巨指针。这通过发出代码来计算每次内存访问的绝对地址来克服大型模型的限制，并允许结构体和数组跨越 64 KB 的内存。显然，这是有代价的：程序代码现在更大，运行时成本也更大。

就是这样！

值得强调的是，这些模型都是旧的 C 编译器用来生成代码的*约定*。如果您用手写汇编，您可以混合和匹配指针类型来做任何您想做的事情，因为这些概念对操作系统没有特殊意义。

演变到当今世界

到目前为止，我告诉您的所有内容都是遗留的东西，您可以很容易地将其视为无用的知识。或者你能吗？

有一件事我没有谈及的是*代码密度*以及它与性能的关系。我们在代码中选择表达指针的方式对代码密度有直接影响，因此当我们将计算从 8086 之类的 16 位机器发展到当代 64 位机器时，指针表示会增长*很多*，我们面临一些艰难的选择。

但是要解释所有这些并回答性能问题，您必须等待下一篇文章。所以现在就订阅，不要错过它！

更多意味着“下一篇文章”，因为有很多内容需要解释这与当今 64 位世界之间的联系。订阅 Blog System/5 以免错过！