<http://blog.sina.com.cn/s/blog_bf9b5aa60101dp8v.html>

开始进行 64 位 Windows 系统编程之前需要了解的所有信息

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif (2013-03-24 15:39:25)

[[http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif](javascript:;)转载▼](javascript:;)

|  |  |
| --- | --- |
|  | 分类： [Ring3](http://blog.sina.com.cn/s/articlelist_3214629542_1_1.html) |

本文讨论：

|  |  |
| --- | --- |
| • | 64 位版本 Windows 的背景信息 |
| • | 适当地利用 x64 体系结构 |
| • | 使用 Visual C++ 2005 进行 x64 开发 |
| • | 针对 x64 版本的调试技术 |

**本文使用以下技术：**  
Windows、Win64、Visual Studio 2005

*

**本页内容**

|  |  |
| --- | --- |
| [x64 操作系统](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EDC) | [x64 操作系统](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EDC) |
| [适当利用 x64](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#ECE) | [适当利用 x64](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#ECE) |
| [使用 Visual C++ 进行 x64 开发](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#ERF) | [使用 Visual C++ 进行 x64 开发](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#ERF) |
| [使代码与 Win64 兼容](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EQG) | [使代码与 Win64 兼容](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EQG) |
| [调试](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EFH) | [调试](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EFH) |
| [关于托管代码](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#E2H) | [关于托管代码](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#E2H) |
| [小结](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EJAAC) | [小结](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#EJAAC) |

使用 Windows® 先锋产品的乐趣之一是能够探究新技术以了解它的工作方式。实际上，我不太喜欢使用操作系统，直到对其内部结构有了一点深入了解之后。因此，当 Windows XP 64 位版本和 Windows Server® 2003 出现时，我简直快完蛋了。

Win64 和 x64 CPU 体系结构的优点是：它们与其前任完全不同，但不需要很长的学习过程。尽管开发人员认为迁移到 x64 只是一个重新编译的过程，但事实是我们仍然要在调试器中花费很多时间。拥有 OS 和 CPU 的应用知识十分宝贵。

本文，我将本人在 Win64 和 x64 体系结构方面的经验归结为一个高手 Win32® 程序员迁移到 x64 必备的几个要点。我假设您了解基本的 Win32 概念、基本的 x86 概念以及为什么代码应该在 Win64 上运行。这使我可以将关注的重点放在更重要的内容上。通过本概述，您可以在已经理解的 Win32 和 x86 体系结构基础上了解到一些重要差异。

有关 x64 系统的一个优点是：与基于 Itanium 的系统不同，您可以在同一台计算机上使用 Win32 或 Win64，而不会导致严重的性能损失。此外，除了 Intel 和 AMD x64 实现之间的几个模糊差异，与 x64 兼容的同一个 Windows 版本应该能够在这两个系统上运行。您不需要在 AMD x64 系统上使用一个 Windows 版本，在 Intel x64 系统上使用另一个版本。

我将讨论分为三大领域：OS 实现细节、适当地利用 x64 CPU 体系结构以及使用 Visual C++® 进行 x64 开发。

**x64 操作系统**

在 Windows 体系结构的所有概述中，我一般喜欢从内存和地址空间开始。尽管 64 位处理器在理论上寻址 16 EB 的内存 (264)，但 Win64 目前支持 16 TB（由 44 位表示）。为什么不能在计算机中加载到 16 EB 以使用全部 64 位呢？原因有很多。

对初级用户而言，当前的 x64 CPU 通常只允许访问 40 位（1 TB）的物理内存。体系结构（不包括当前硬件）可以将其扩展到 52 位（4 PB）。即使没有该限制，映射如此大内存的页表大小也是巨大的。

与 Win32 中一样，可寻址范围分为用户模式区和内核模式区。每个进程都在底部获得其唯一的 8 TB，而内核模式的代码存在于顶部的 8 TB 中，并由所有进程共享。不同版本的 64 位 Windows 具有不同的物理内存限制，如[图 1](http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx?fig=true#fig1) 和[图 2](http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx?fig=true#fig2) 所示。

同样，与 Win32 中一样，x64 页大小为 4 KB。前 64 KB 的地址空间始终不映射，因此您看到的最低有效地址应该是 0x10000。与在 Win32 中不同，系统 DLL 在用户模式的地址范围顶部附近没有默认的加载地址。相反，它们在 4 GB 内存以上加载，通常在 0x7FF00000000 附近的地址上加载。

许多较新的 x64 处理器的一个出色功能是：支持 Windows 用于实现硬件数据执行保护 (DEP) 的 CPU No Execute 位。x86 平台上存在许多错误和病毒，这是因为 CPU 可以将数据当作合法代码字节执行。CPU 在供数据存储使用的内存中执行从而可终止缓冲区溢出（有意或无意）。通过 DEP，OS 可以在有效代码区域周围设置更清晰的边界，从而使 CPU 在执行超出这些预期边界时捕获到该事件。这推动着为使 Windows 减少受到的攻击而付出的不懈努力。

在为捕获错误而设计的活动中，x64 链接器将可执行文件默认的加载地址指定为在 32 位 (4 GB) 之上。这可以帮助在代码迁移到 Win64 之后能够在现有代码中快速找到这些区域。具体说，如果将指针存储为一个 32 位大小的值（如 DWORD），那么在 Win64 版本中运行时，它将被有效地截断，从而导致指针无效，进而触发访问冲突。该技巧使查找这些令人讨厌的指针错误变得非常简单。

有关指针和 DWORD 的主题将在 Win64 类型系统中继续讨论。指针有多大？LONG 怎么样？那么句柄（如 HWND）呢？幸好，Microsoft 在进行从 Win16 到 Win32 的复杂转换时，使新的类型模型能够轻松地进一步扩展到 64 位。一般地，除了个别几种情况外，新的 64 位环境中的所有类型（除了指针和 size\_t）均与 Win32 中的完全相同。也就是说，64 位指针是 8 字节，而 int、long、DWORD 和 HANDLE 仍然是 4 字节。在随后讨论进行 Win64 开发时，我将讨论更多有关类型的内容。

Win64 的文件格式称为 PE32+。几乎从每个角度看，该格式在结构上都与 Win32 PE 文件完全相同。只是扩展了少数几个字段（例如，头结构中的 ImageBase），删除了一个字段，并更改了一个字段以反映不同的 CPU 类型。[图 3](http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx?fig=true#fig3) 显示已更改的字段。

除 PE 头之外，没有太多的更改。有几个结构（例如，IMAGE\_LOAD\_CONFIG 和 IMAGE\_THUNK\_DATA）只是将某些字段扩展到 64 位。添加的 PDATA 区段很有趣，因为它突出了 Win32 和 Win64 实现之间的一个主要差异：异常处理。

在 x86 环境中，异常处理是基于堆栈的。如果 Win32 函数包含 try/catch 或 try/finally 代码，则编译器将发出在堆栈上创建小型数据块的指令。此外，每个 try 数据块指向先前的 try 数据结构，从而形成了一个链表，其中最新添加的结构位于表头。随着函数的调用和退出，该链表头会不断更新。如果发生异常，OS 将遍历堆栈上的数据块链表，以查找相应的处理程序。我在 1997 年 1 月的 [*MSJ* 文章](http://microsoft.com/msj/0197/Exception/Exception.aspx)中非常详细地描述了该过程，因此这里只做简要说明。

与 Win32 异常处理相比，Win64（包括 x64 和 Itanium 版本）使用了基于表的异常处理。它不会在堆栈上生成任何 try 数据块链表。相反，每个 Win64 可执行文件都包含一个运行时函数表。每个函数表项都包含函数的起始和终结地址，以及一组丰富数据（有关函数中异常处理代码）的位置和函数的堆栈帧布局。请参见 WINNT.H 和 x64 SDK 中的 IMAGE\_RUNTIME\_FUNCTION\_ENTRY 结构，了解这些结构的实质。

当异常发生时，OS 会遍历常规的线程堆栈。当堆栈审核遇到每个帧和保存的指令指针时，OS 会确定该指令指针属于哪一个可执行的模块。随后，OS 会在该模块中搜索运行时函数表，查找相应的运行时函数项，并根据这些数据制定适当的异常处理决策。

如果您是一位火箭科学家，并直接在内存中生成了代码而没有使用基本的 PE32+ 模块，该怎么办呢？这种情况也包含在内。Win64 有一个 RtlAddFunctionTable API，它可让您告诉 OS 有关动态生成的代码的信息。

基于表的异常处理的缺点（相对于基于堆栈的 x86 模型）是：在代码地址中查找函数表项所需的时间比遍历链表的时间要长。但优点是：函数没有在每次执行时设置 try 数据块的开销。

请记住，这只是一个简要介绍，而不是 x64 异常处理的完整描述，但是很令人激动，不是吗？有关 x64 异常模型的进一步概述，请参阅 Kevin Frei 的[网络日记项](http://blogs.msdn.com/509372.aspx)。

与 x64 兼容的 Windows 版本不包含最新 API 的具体数量；大部分新的 Win64 API 都添加到针对 Itanium 处理器的 Windows 版本。简言之，现有的两个重要的 API，分别是 IsWow64Process 和 GetNativeSystemInfo。它们允许 Win32 应用程序确定是否在 Win64 上运行，如果是，则可以看到系统的真正功能。否则，调用 GetSystemInfo 的 32 位进程只能看到 32 位系统的系统功能。例如，GetSystemInfo 只会报告 32 位进程的地址范围。[图 4](http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx?fig=true#fig4) 显示的 API 以前在 x86 上不可用，但可用于 x64。

尽管运行完全的 64 位 Windows 系统听起来很不错，但事实是，在某些情况下，您很可能需要运行 Win32 代码。为此，x64 版本的 Windows 包含 WOW64 子系统，以允许 Win32 和 Win64 进程在同一个系统上并行运行。但是，将 32 位 DLL 载入 64 位进程（反之亦然）则不受支持。（相信我，这是件好事。）您终于可以向 16 位旧式代码吻别了！

在 x64 版本的 Windows 中，从 64 位可执行文件启动的进程（如 Explorer.exe）只能加载 Win64 DLL，而从 32 位可执行文件启动的进程只能加载 Win32 DLL。当 Win32 进程调用内核模式（例如，读取文件）时，WOW64 代码会安静地截断该调用，并在适当的位置调用正确的 x64 等效代码。

当然，不同系统（32 位与 64 位）的进程需要能够互相通信。幸运的是，Win32 中您知道并喜爱的所有常规进程间通信机制也可以在 Win64 中工作，包括共享内存、命名管道以及命名同步对象。

您可能在想，"那么系统目录呢？同一个目录不能同时保存 32 位和 64 位版本的系统 DLL（例如，KERNEL32 或 USER32），不是吗"？通过执行可选择的文件系统重定向，WOW64 魔法般地为您解决了这个问题。来自 Win32 进程的文件活动通常转到 System32 目录，而不是在名为 SysWow64 的目录中。在内部，WOW64 会默默地更改这些请求以指向 SysWow64 目录。Win64 系统实际上有两个 \Windows\System32 目录 - 一个用于 x64 二进制文件，另一个用于 Win32 等效文件。

这看上去没什么，但会令人混淆。例如，我在某一点上使用了 32 位命令行提示（我自己并不知道）。当我针对 System32 目录中的 Kernel32.dll 运行 DIR 时，所得到的结果与我在 SysWow64 目录中执行相同操作后所得到的结果完全相同。我绞尽脑汁后才发现，文件系统重定向的工作方式就是这样。也就是说，即使我认为是在 \Windows\System32 目录中工作，但 WOW64 实际上已将调用重定向到 SysWow64 目录。顺便说一下，如果您确实希望从 x64 应用程序访问 32 位 \Windows\System32 目录，则 GetSystemWow64Directory API 会提供正确的路径。请一定阅读 MSDN® 文档，了解完整的信息。

除了文件系统重定向之外，WOW64 施加的另一个小魔法是注册表重定向。请考虑我前面提到的 Win32 DLL 不能载入 Win64 进程的内容，然后再考虑一下 COM 及其使用注册表加载进程内服务器 DLL 的情况。如果 64 位应用程序要使用 CoCreateInstance 创建一个在 Win32 DLL 中实现的对象，该怎么办呢？该 DLL 不能加载，对吗？WOW64 通过将来自 32 位应用程序的访问重定向到 \Software\Classes（以及相关的）注册节点，再一次节省了时间。实际结果是，Win32 应用程序的注册表视图与 x64 应用程序的不同（但大部分是相同的）。如您所料，OS 通过在调用 RegOpenKey 及友元时指定新的标记值，为 32 位应用程序提供了一个读取实际 64 位注册表值的应急方法。

更进一步说，后几个正中我下怀的 OS 差异涉及线程的局部数据。在 x86 版本的 Windows 中，FS 寄存器用于指向每个线程的内存区域，包括"最后一个错误"和线程的本地存储（分别是 GetLastError 和 TlsGetValue）。在 x64 版本的 Windows 中，FS 寄存器由 GS 寄存器取代。另外，它们的工作方式几乎完全相同。

虽然本文主要从用户模式角度讨论 x64，但有一项重要的内核模式体系结构附加内容需要说明。针对 x64 的 Windows 中有一项称为 PatchGuard 的新技术，该技术主要针对安全性和健壮性。简言之，能够更改关键内核数据结构（例如，系统调用表和中断调度表 (IDT)）的用户模式程序或驱动程序会导致安全漏洞和潜在的稳定性问题。对于 x64 体系结构而言，Windows 家族决定不允许以不受支持的方式修改核心内存。强制该操作的技术是 PatchGuard。它使用内核模式线程监视对关键核心内存位置的更改。如果该内存被更改，则错误检测时系统将停止。

总之，如果您熟悉 Win32 体系结构，并且了解如何编写在它上面运行的本机代码，那么在迁移到 Win64 的过程中您就不会感到很惊奇了。您可以在很大程度上将其视为一个更广阔的环境。

[返回页首返回页首](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#top)

**适当利用 x64**

现在，我们看一下 CPU 体系结构本身，因为对 CPU 指令集有一个基本的了解可以使开发（特别是调试）工作更轻松。在编译器生成的 x64 代码中，您将注意到的第一件事是，它与您了解并喜爱的 x86 代码是多么地相似。这对于了解 Intel IA64 编码的人们则完全不同。

随后您将注意到的第二件事是，注册名称与您所熟悉的略有不同，并且有很多名称。通用 x64 寄存器的名称以 R 开头，如 RAX、RBX 等等。这是针对 32 位 x86 寄存器的基于 E 的旧命名方案的发展演化。就像过去一样，16 位 AX 寄存器变为 32 位 EAX，16 位 BX 变为 32 位 EBX，以此类推。如果从 32 位版本转换，所有 E 寄存器都会变为其 64 位形态的 R 寄存器。因此，RAX 是 EAX 的继承者，RBX 超越 EBX，RSI 取代 ESI，以此类推。

此外，还添加了 8 个新的通用寄存器 (R8-R15)。主要的 64 位通用寄存器清单如[图 5](http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx?fig=true#fig5) 所示。

此外，32 位 EIP 寄存器也会变为 RIP 寄存器。当然，32 位指令必须继续执行，以便这些寄存器（EAX、AX、AL、AH 等）的原始、较小类型的版本仍然可用。

为了照顾到图形和科学编程人员，x64 CPU 还有 16 个 128 位 SSE2 寄存器，分别以 XMM0 到 XMM15 命名。由 Windows 保存的 x64 寄存器的完整集合位于 WINNT.H 中定义的相应 #ifdef'ed \_CONTEXT 结构中。

在任何时候，x64 CPU 不是以旧式的 32 位模式操作，就是以 64 位模式操作。在 32 位模式中，CPU 与任何其他 x86 类别的 CPU 一样对指令进行解码和操作。在 64 位模式中，CPU 对某些指令编码进行了少量调整，以支持新的寄存器和指令。

如果您熟悉 CPU 操作码编码模型，就会记得为新的指令编码提供的空间会很快消失，并且在 8 个新寄存器中挤出空间也不是一项轻松的任务。为此，一种方法是删除一些极少使用的指令。到目前为止，我留下的指令只有 64 位版本的 PUSHAD 和 POPAD，它们用于在堆栈上保存和恢复所有通用寄存器。释放指令编码空间的另一种方法是，在 64 位模式中完全消除区段。这样，CS、DS、ES、SS、FS 和 GS 的生命周期就结束了。没有太多人会想念它们的。

由于地址是 64 位的，您可能会担心代码大小。例如，下面是一个常见的 32 位指令：

CALL DWORD PTR [XXXXXXXX]

这里，用 X 表示的部分是一个 32 位地址。在 64 位模式中，这会变为 64 位地址，从而将 5 字节的指令变为 9 字节吗？幸运的是，答案是"否"。指令大小保持不变。在 64 位模式中，指令的 32 位操作数部分被视为相对于当前指令的数据偏移。一个示例可以更清楚地说明这一点。在 32 位模式中，以下是调用地址 00020000h 中存储的 32 位指针值的指令：

00401000: CALL DWORD PTR [00020000h]

在 64 位模式中，相同的操作码字节调用地址 00421000h (4010000h + 20000h) 中存储的 64 位指针值。这可以使您联想到，如果是自己生成代码，则这种相对寻址模式会造成重大分歧。您不能仅在指令中指定 8 字节的指针值，而是需要为实际 64 位目标地址驻留的内存位置指定一个 32 位相对地址。因而，有一个未提出的假设是：64 位目标指针必须在使用它的指令的 2GB 空间中。对大多数人而言，这并不是一个大问题，但如果您要生成动态代码或者修改内存中的现有代码，就会出现问题！

所有 x64 寄存器的一个主要优势是，编译器能够最终生成在寄存器中（而非堆栈上）传递大部分参数的代码。将参数推入堆栈会引发内存访问。我们都需要牢记，在 CPU 缓存中找不到的内存访问会导致 CPU 延迟许多个周期，以等待可用的常规 RAM 内存。

在设计调用约定时，x64 体系结构利用机会清除了现有 Win32 调用约定（如 \_\_stdcall、\_\_cdecl、\_\_fastcall、\_thiscall 等）的混乱。在 Win64 中，只有一个本机调用约定和 \_\_cdecl 之类的修饰符被编译器忽略。除此之外，减少调用约定行为还为可调试性带来了好处。

您需要了解的有关 x64 调用约定的主要内容是：它与 x86 fastcall 约定的相似之处。使用 x64 约定，会将前 4 个整数参数（从左至右）传入指定的 64 位寄存器：

RCX: 1st integer argument RDX: 2nd integer argument R8: 3rd integer argument R9: 4th integer argument

前 4 个以外的整数参数将传递到堆栈。该指针被视为整数参数，因此始终位于 RCX 寄存器内。对于浮点参数，前 4 个参数将传入 XMM0 到 XMM3 的寄存器，后续的浮点参数将放置到线程堆栈上。

更进一步探究调用约定，即使参数可以传入寄存器，编译器仍然可以通过消耗 RSP 寄存器在堆栈上为其预留空间。至少，每个函数必须在堆栈上预留 32 个字节（4 个 64 位值）。该空间允许将传入函数的寄存器轻松地复制到已知的堆栈位置。不要求被调用函数将输入寄存器参数溢出至堆栈，但需要时，堆栈空间预留确保它可以这样做。当然，如果要传递 4 个以上的整数参数，则必须预留相应的额外堆栈空间。

让我们看一个示例。请考虑一个将两个整数参数传递给子函数的函数。编译器不仅会将值赋给 RCX 和 RDX，还会从 RSP 堆栈指针寄存器中减去 32 个字节。在被调用函数中，可以在寄存器（RCX 和 RDX）中访问参数。如果被调用代码因其他目的而需要寄存器，可将寄存器复制到预留的 32 字节堆栈区域中。图 6 显示在传递 6 个整数参数之后的寄存器和堆栈。

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif

图 6 传递整数

x64 系统上的参数堆栈清除比较有趣。从技术上说，调用方（而非被调用方）负责清除堆栈。但是，您很少看到在起始代码和结束代码之外的位置调整 RSP。与通过 PUSH 和 POP 指令在堆栈中显式添加和移除参数的 x86 编译器不同，x64 代码生成器会预留足够的堆栈空间，以调用最大目标函数（参数方法）所使用的任何内容。随后，在调用子函数时，它重复使用相同的堆栈区域来设置参数。

另一方面，RSP 很少更改。这与 x86 代码大不相同，在 x86 代码中，ESP 值随着参数在堆栈中的添加和移除而不断变化。

有一个示例可帮助说明这一点。请考虑一个调用三个其他函数的 x64 函数。第一个函数接受 4 个参数（0x20 个字节），第二个接受 12 个参数（0x60 个字节），第三个接受 8 个参数（0x40 个字节）。在起始代码中，生成的代码只需在堆栈上预留 0x60 个字节，并将参数值复制到 0x60 字节中的适当位置，以便目标函数能够找到它们。

您可以在 [Raymond Chen 的网络日记](http://blogs.msdn.com/58579.aspx)中看到一个有关 x64 调用约定更详细的描述。我不会过多地讨论所有细节，仅在这里强调一些要点。首先，小于 64 位的整数参数进行了符号扩展，然后仍然通过相应的寄存器传递（如果在前 4 个整数参数内）。其次，任何参数所处的堆栈位置都应该是 8 字节的倍数，从而保持 64 位对齐。不是 1、2、4 或 8 字节的任何参数（包括结构）都是通过引用传递的。最后，8、16、32 或 64 位的结构和联合作为相同长度的整数传递。

函数的返回值存储在 RAX 寄存器中。如果返回到 XMM0 中的是浮点类型，就会引发异常。在所有调用中，以下寄存器必须保留：RBX、RBP、RDI、RSI、R12、R13、R14 和 R15。以下寄存器不稳定，可能会被毁坏：RAX、RCX、RDX、R8、R9、R10 和 R11。

我在前面提到过，作为异常处理机制的一部分，OS 会遍历堆栈帧。如果您曾经编写过堆栈遍历代码，就会知道 Win32 帧布局的这一特性可巧妙处理该过程。这种情况在 x64 系统上要好得多。如果某个函数需要分配堆栈空间，调用其他函数，保留任何寄存器或者使用异常处理，则该函数必须使用一组定义良好的指令来生成标准的起始代码和结束代码。

实行创建函数堆栈帧的标准方法是 OS 确保（在理论上）能够始终遍历堆栈的一种方法。除了一致、标准的起始代码，编译器和链接器还必须创建关联的函数表数据项。奇怪的是，所有这些函数项都在 IMAGE\_FUNCTION\_ENTRY64 的数组表（在 winnt.h 中定义）中结束。如何找到这个表呢？它由 PE 头的 DataDirectory 字段中的 IMAGE\_DIRECTORY\_ENTRY\_EXCEPTION 项指出。

我在短短的一段中讨论了许多体系结构内容。但是，通过大体了解这些概念以及 32 位程序集语言的现有知识，您应该能够在一段较短的时间内了解调试器中的 x64 指令。总是实践出真知。

[返回页首返回页首](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#top)

**使用 Visual C++ 进行 x64 开发**

尽管可以使用 Visual Studio® 2005 之前的 Microsoft® C++ 编译器编写 x64 代码，但这在 IDE 中是一项沉闷的体验。因此，在本文中，我假定您使用的是 Visual Studio 2005，并选择在默认安装中未启用的 x64 工具。我还假定您在 C++ 中拥有要为 x86 和 x64 平台构建的现有 Win32 用户模式项目。

针对 x64 构建的第一步是创建 64 位生成配置。作为一个优秀的 Visual Studio 用户，您应该已经知道项目在默认情况下有两种配置：Debug 和 Retail。这里，您只需创建另外两个配置：x64 形态下的 Debug 和 Retail。

首先，加载现有项目/解决方案。在 Build 菜单上，选择 Configuration Manager。在 Configuration Manager 对话框中，从 Active solution platform 下拉菜单中选择 New（参见图 7）。现在，您应该看到另一个标题为 New Solution Platform 的对话框。

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif

图 7 创建新的生成配置

选择 x64 作为您的新平台（参见图 8），并将另一个配置保留为默认状态；然后单击 OK。就这么简单！现在，您应该拥有四个可能的生成配置：Win32 Debug、Win32 Retail、x64 Debug 和 x64 Retail。使用 Configuration Manager，您可以轻松地在它们之间切换。

现在，我们看一下您的代码与 x64 的兼容性。将 x64 Debug 配置设为默认值，然后生成项目。除非代码不重要，否则可能会收到一些不会在 Win32 配置中发生的编译器错误。除非您已经完全摒弃了编写可移植 C++ 代码的所有原则，否则修正这些问题以使代码能够随时用于 Win32 和 x64 相对比较轻松，而无需大量的条件编译代码。

issues_x64_default_fig08

图 8 选择生成平台

[返回页首返回页首](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#top)

**使代码与 Win64 兼容**

将 Win32 代码转换为 x64，所需的最重要的工作可能是确保类型定义正确。还记得先前讨论的 Win64 类型系统吗？通过使用 Windows typedef 类型而非 C++ 编译器的本机类型（int、long 等），Windows 头使得编写干净的 Win32 x64 代码很轻松。您应该在自己的代码中继续保持这一点。例如，如果 Windows 将一个 HWND 传递给您，请不要仅仅为了方便就将其存储在 FARPROC 中。

升级完许多代码之后，我看到的最常见而简单的错误可能就是：假定指针值可以存储或传递到 32 位类型（如 int 和 long）甚至 DWORD 中。Win32 和 Win64 中的指针长度视需要而不同，而整数类型长度保持不变。但是，让编译器不允许指针存储在整数类型中也是不现实的。这是一个根深蒂固的 C++ 习惯。

解救方法是 Windows 头中定义的 \_PTR 类型。DWORD\_PTR、INT\_PTR 和 LONG\_PTR 之类的类型可让您声明整数类型的变量，并且这些变量始终足够长以便在目标平台上存储指针。例如，定义为 DWORD\_PTR 类型的变量在针对 Win32 编译时是 32 位整数，在针对 Win64 编译时是 64 位整数。经过实践，我已经习惯了声明类型以询问"这里是否需要 DWORD 或者实际是指 DWORD\_PTR 吗？"。

正如您期望的，可能有机会明确指定整数类型需要多少字节。定义 DWORD\_PTR 及其友元的同一头文件 (Basetsd.h) 还可以定义特定长度的整数，如 INT32、INT64、INT16、UINT32 和 DWORD64。

与类型大小差异相关的另一个问题是 printf 和 sprintf 格式化。我对于在过去使用 %X 或 X 格式化指针值感到懊悔万分，并且在 x64 系统上运行该代码时还遇到了阻碍。正确的方法是使用 %p，%p 可以在目标平台上自动考虑指针大小。此外，对于与大小相关的类型，printf 和 sprintf 还具有 I 前缀。例如，您可能使用 %Iu 来打印 UINT\_PTR 变量。同样，如果您知道该变量始终是 64 位标记值，则可以使用 %I64d。

在清除了无法用于 Win64 的类型定义所导致的错误之后，可能还有只能在 x86 模式下运行的代码。或者，您可能需要编写函数的两个版本，一个用于 Win32，另一个用于 x64。这就是一组预处理器宏的用武之地：

\_M\_IX86 \_M\_AMD64 \_WIN64

正确使用预处理器宏对于编写正确的跨平台代码而言至关重要。\_M\_IX86 和 \_M\_AMD64 仅在针对特定处理器编译时进行定义。\_WIN64 在针对任何 64 位版本的 Windows（包括 Itanium 版）编译时定义。

在使用预处理器宏时，请仔细考虑您的需要。例如，只需要代码真正特定于 x64 处理器，没有别的需要了吗？然后，使用与以下类似的代码：

#ifdef \_M\_AMD64

另一方面，如果同一代码既可以在 x64 又可以在 Itanium 上工作，则使用如下所示的代码可能更好：

#ifdef \_WIN64

我发现一个有用的习惯是：只要使用其中一个宏，就始终显式创建 #else 情况，以便提前知道是否忘记了某些情况。请考虑以下编写错误的代码：

#ifdef \_M\_AMD64 // My x64 code here #else // My x86 code here #endif

如果现在针对第三个 CPU 体系结构编译该代码，会发生什么情况？系统将无意识地编译我的 x86 代码。上面代码的一个更好的表达方式如下：

#ifdef \_M\_AMD64 // My x64 code here #elif defined (\_M\_IX86) // My x86 code here #else #error !!! Need to write code for this architecture #endif

在我的 Win32 代码中无法轻松移植到 x64 的一部分代码是内联汇编，Visual C++ 不支持它的 x64 目标。不要害怕，汇编有办法。它提供了一个 64 位 MASM (ML64.exe)，这在 MSDN 中有所说明。ML64.exe 和其他 x64 工具（包括 CL.EXE 和 LINK.EXE）可以从命令行调用。您可以只运行 VCVARS64.BAT 文件，该文件可以将它们添加到您的路径中。

[返回页首返回页首](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#top)

**调试**

最后，您需要在 Win32 和 x64 版本上干净地编译代码。最后一个难题是运行和调试代码。无论是否在 x64 盒上生成 x64 版本，您都需要使用 Visual Studio 远程调试功能在 x64 模式下进行调试。幸运的是，如果您在 64 位计算机上运行 Visual Studio IDE，则 IDE 将为您执行以下所有步骤。如果您出于某些原因无法使用远程调试，则另一个选项是使用 [x64 版本的 WinDbg](http://microsoft.com/whdc/devtools/debugging/install_64bit.mspx)。但是，您会失去 Visual Studio 调试器提供的许多调试优势。

如果您从未使用过远程调试，也不需要过于担心。一旦设置好，远程调试就可以像在本地一样无缝使用。

第一步是在目标计算机上安装 64 位 MSVSMON。这通常是通过运行 Visual Studio 随附的 RdbgSetup 程序来完成的。一旦 MSVSMON 运行，请使用 Tools 菜单为 32 位 Visual Studio 和 MSVSMON 实例之间的连接配置适当的安全设置（或者缺失）。

接下来，您需要在 Visual Studio 中将项目配置为针对 x64 代码使用远程调试，而不是尝试进行本地调试。您可以从调试项目的属性开始启动这个过程（参见图 9）。

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif

图 9 调试属性

确定 64 位配置是当前配置，然后选择 Configuration Properties 下面的 Debugging。靠近顶端是标题为 Debugger to launch 的下拉菜单。通常，它设置为 Local Windows Debugger。将其更改为 Remote Windows Debugger。在下面，您可以指定在启动调试时要执行的远程命令（例如，程序名），以及远程计算机名和连接类型。

如果您正确设置了所有内容，就可以使用与启动 Win32 应用程序相同的方式开始调试 x64 目标应用程序。您可以知道是否已经成功连接到 MSVSMON，因为每次调试器成功连接后，MSVSMON 的跟踪窗口都会显示一个"connected"字符串。在这里，通常都是您知道并喜爱的同一个 Visual Studio 调试器。确保屏幕显示寄存器窗口，并查看所有这些出色的 64 位寄存器，然后转到反汇编窗口以查看"非常熟悉但略有不同的"x64 程序集代码。

请注意，不能将 64 位小型转储直接加载到 Visual Studio 之类的 32 位转储中，而是需要使用远程调试。此外，Visual Studio 2005 目前不支持本机 64 位代码和托管 64 位代码之间的互操作调试。

[返回页首返回页首](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#top)

**关于托管代码**

使用 Microsoft .NET Framework 进行编码的一个优势是，大部分基础操作系统都归纳为通用代码。此外，IL 指令格式是 CPU 不可知的。因此，从理论上说，在 Win32 系统上生成的基于 .NET 的程序二进制文件应该无需修改就可以在 x64 系统上运行。但实际情况却有一点复杂。

.NET Framework 2.0 提供了 x64 版本。在 x64 计算机上安装 .NET Framework 2.0 之后，我能够运行先前在 Win32 环境中运行的 .NET 可执行文件。这真棒！当然，虽然不能保证每个基于 .NET 的程序无需重新编译就可以在 Win32 和 x64 上都运行良好，但它确实在一段合理的时间内"很有用"。

如果您的托管代码显式调用本机代码（例如，通过 C# 或 Visual Basic® 中的平台调用），则在尝试针对 64 位 CLR 运行时可能会遇到问题。但是，有一个编译器开关 (/platform) 可让您更清楚地了解代码应该在哪个平台上运行。例如，您可能希望托管代码在 WOW64 中运行，即使可以使用 64 位 CLR。

[返回页首返回页首](http://www.microsoft.com/china/MSDN/library/Windev/64bit/issuesx64.mspx#top)

**小结**

总之，对于我而言，迁移到 x64 版本的 Windows 是一个相对比较轻松的经历。一旦您很好地掌握了 OS 体系结构和工具中相对较小的差异，就可以轻松地使一个代码基在这两个平台上运行。Visual Studio 2005 可从根本上使这些工作更加轻松。此外，由于每天都会出现更多特定于 x64 版本的设备驱动程序和工具（如 SysInternals.com 提供的 Process Explorer），因此没有理由不进行讨论！

**Matt Pietrek** 与他人合著有几本关于 Windows 系统级编程的书籍，以及 *MSDN Magazine* 的 Under the Hood 专栏。他以前曾是 NuMega/Compuware BoundsChecker 系列产品的首席架构师。现在，他是 Microsoft Visual Studio 小组的一员。

[转到原英文页面](http://msdn.microsoft.com/msdnmag/issues/06/05/x64/default.aspx)

<http://www.boxcounter.com/printthread.php?tid=96>

**64位系统的对齐问题** - [boxcounter](http://boxcounter.com/member.php?action=profile&uid=1) - **10-25-2012** **09:46 PM**   
  
Boxcounter  
<2012-10-25 周四>  
  
今天在技术群里有位童鞋问了一个问题：  
“  
On 64-bit Windows, if a data structure is misaligned, routines that manipulate the structure, such as RtlCopyMemory and memcpy, will not fault. Instead, they will raise an exception. For example:  
  
#pragma pack (1) /\* also set by /Zp switch \*/  
struct Buffer {  
ULONG size;  
void \*ptr;  
};  
  
void SetPointer(void \*p) {  
struct Buffer s;  
s.ptr = p; /\* will cause alignment fault \*/  
...  
}  
  
  
You could use the UNALIGNED macro to fix this:  
  
void SetPointer(void \*p) {  
struct Buffer s;  
\*(UNALIGNED void \*)&s.ptr = p;  
}  
”  
  
我搜了一下，这段话来自[Porting Issues Checklist](http://msdn.microsoft.com/en-us/library/windows/hardware/ff559744(v=vs.85).aspx)这篇文章。  
  
他的问题是：  
“我对这句 s.ptr = p 不太明白。为什么会引发对齐错误，什么意思？”  
  
我的回答（当时的原话，貌似有点凌乱，群里讨论嘛，想一句说一句，见谅）：  
1. 因为s这个结构体的首地址是被编译器默认对齐的。由于“#pragma pack (1)”，s.ptr肯定不在对齐的地址上。所以访问s.ptr时会触犯x64 win的对齐规定，于是抛出异常。  
2. 对于x86架构，对于类似这种没有对齐的地址访问，win内核会自动帮你处理好，用户感觉不到这个问题的存在。但实际上会导致系统性能下降。   
到了x64架构，win决定不帮用户擦屁股了。于是就会抛出异常。   
3. x64是按照8字节对齐，&s是对齐的，那s.ptr是处在 （N\*8 + 4）的地址上，于是没有对齐。  
  
这时另一位提了一个问题：  
“#pragma pack (1) 了，对齐地址是按1的倍数来的，为什么会s.ptr不在对齐的地址上啊？”  
  
我的回答：  
4. pragma pack的对齐只是影响编译器对结构体的padding，这个俗称的对齐，跟硬件手册里的对齐，不是一回事。  
  
回答完后觉得心理有点不踏实，主要的不确定点是回答3中的“x64是按照8字节对齐”。翻了下手册，果然没这么简单。  
（x64架构的）cpu的对齐是根据数据类型的不同而不同的，对于第一个问题，s.ptr是个指针，确实是按照8个字节来对齐的，所以我的回答没有问题。  
更具体的对齐在《AMD64 Architecture Programmer's Manual Volume 2 [System Programming]》的“8.2.17 #AC—Alignment-Check Exception (Vector 17)”，如下图：  
[attachment=35]  
  
  
还有个问题，为什么不对齐就会影响性能？  
以x64位下的void\*数据类型为例，void\*是8个字节大小。如果某void\* p没有按照8字节对齐，比如下图所示：

Code:

addr:      N\*8           (N+1)\*8  
            |               |  
offset:     0 1 2 3 4 5 6 7  
           +-+-+-+-+-+-+-+-+-+  
           | |X|X|X|X|X|X|X|X|  
           +-----------------+  
             ^  
             |  
             p

如果按照8字节对齐，即ptr处在N\*8的位置，那么当需要访问ptr的时候，把ptr的值从mem读到cpu cache，只需要一次读取操作（从N\*8到N\*8+7，正好8个字节）。  
但是如果如上图所示，那么就不得不读两次，第一次也是从N\*8到(N+1)\*8，第二次读取(N+1)\*8到(N+1)\*8+7，然后还要把第一次读的后7个字节和第二次​读的第一个字节拼起来。此外，在多核环境下，为了保证对ptr的读取操作的原子性，还必须额外消耗一个bus control。另外，unaligned的ptr还有可能分布在两个cache line中，导致更多额外的处理成本（每次对ptr的修改操作会影响两个cache line，造成更严重的[False Sharing](http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/)）。  
  
再多想想，有人可能会问：对于WORD类型，只要求2字节对齐，那如果具体的内存是这样呢：

Code:

addr:      N\*8  
            |  
offset:     0 1 2 3 4 5 6 7  
           +-+-+-+-+-+-+-+-+  
           | | |X|X| | | | |  
           +---------------+

这样虽然是按照2字节对齐，但是并没有在N\*8的位置，会不会有性能损耗？  
  
理论上是有的，因为读取到cpu cache后，还有个+2的操作（理论上有，但我不确定到电路那一层，+0和+2的耗时是不是就等价了）。但即便有这个+2的操作，也没太大影响，因为对cpu cache的操作是非常非常快的，而mem->cpu cache这个操作相对来说就非常耗时了。  
总结：性能损耗主要是因为多了一次mem->cpu cache操作。  
  
注：  
在整理这篇文章之前，我不确定多一次读操作是不是导致性能损耗的**主要**原因，于是请教sinster师傅，对话如下：  
  
Boxcounter 18:43:11  
贾佳师傅，请教个问题：  
  
访问unaligned的数据会造成性能下降，比如64位系统，某个void\*类型的数据如果是unaligned。那么访问它时会导致2次从内存到cpu cache的读操作，而如果它是aligned，则只需要一次。  
是因为这个原因吗？   
sinister 18:43:55  
是，有些系统直接CORE DUMP了。   
sinister 18:44:44  
一些RISC的，POWER,SPARC,EPIC 都这样   
  
后记：  
提问的童鞋说在x64 WinXP系统里，即使不对齐也没有因为异常而中断下来。  
我试了下，确实如此，x64 win7系统，挂着windbg内核调试，单步步过unaligned操作，并没有任何提示和中断等异样。  
恢复到一个没有任何调试设置的快照，也是如此。应用层也是一样。  
  
继续琢磨，在AMD手册里翻到这么几段资料：  
1. 还是在“8.2.17 #AC—Alignment-Check Exception (Vector 17)”

Quote:After a processor reset, #AC exceptions are disabled. Software enables the #AC exception by setting  
the following register bits:  
• CR0.AM=1.  
• RFLAGS.AC=1.  
When the above register bits are set, an #AC can occur only when CPL=3. #AC never occurs when  
CPL < 3.

2. CR0中AM标志在第18bit，RFLAGS的AC标志也是在18bit  
  
查看了下我的虚拟机：  
0: kd> r efl  
efl=00000246  
  
0: kd> .formats 0x246  
......  
Binary: 00000000 00000000 00000000 00000000 00000000 00000000 00000010 01000110  
......  
  
  
0: kd> r cr0  
cr0=0000000080050031  
  
0: kd> .formats 0x0000000080050031   
......  
Binary: 11111111 11111111 11111111 11111111 10000000 00000101 00000000 00110001  
......  
  
可见两个18bit都是0，并没有启用#AC异常。  
  
为了避免是内核调试器对虚拟系统的干扰造成的，我用livekd抓了个本机系统DMP来分析：  
  
0: kd> r cr0  
cr0=0000000080050033  
  
0: kd> .formats 0000000080050033  
......  
Binary: 11111111 11111111 11111111 11111111 10000000 00000101 00000000 00110011  
......  
  
  
0: kd> r efl  
efl=00000286  
  
0: kd> .formats 00000286  
......  
Binary: 00000000 00000000 00000000 00000000 00000000 00000000 00000010 10000110  
......  
  
结果是一样的，也没有启用#AC异常。  
  
不明白为什么，难道MSDN上那篇文章里的64bit仅指IA64？  
目前暂没什么好的思路，先记到这里吧。  
（如果有朋友知道原因，请不吝赐教。因为垃圾回复太多，所以关闭回帖功能了，请发邮件交流吧，ns.boxcounter[at]gmail.com，感谢！）  
  
再后记：  
发现一篇相关的文章：[How to debug unaligned accesses on amd64 using Visual Studio?](http://stackoverflow.com/questions/5498102/how-to-debug-unaligned-accesses-on-amd64-using-visual-studio)  
其中有一条回复是：  
"How do real world Windows developers track unaligned accesses in their code?". Being a primarily Windows developer myself, I think they just never do it, unless the code has to run on Itanium. And if it has to run on Itanium, the exception will tell the places. What is the real overhead of unaligned access on x86? BTW, you can still use assembly on x64, in .asm files, just not inline assembly.   
  
后记  
2013-01-06  
群里的朋友分享了新资料：[\_\_unaligned](http://msdn.microsoft.com/en-us/library/ms177389.aspx)，里面有这么一段话：

Quote:The \_\_unaligned modifier is valid for the x64 and Itanium compilers but affects only applications that target an IPF computer. This modifier describes the alignment of the addressed data only; the pointer itself is assumed to be aligned.  
The Itanium processor generates an alignment fault when it accesses misaligned data, and the time to process the fault weakens performance. Use the \_\_unaligned modifier to cause the processor to read data one byte at a time and avoid the fault. This modifier is not required for x64 applications because the x64 processor handles misaligned data without faulting.

也就是说，确实只有在IA64环境才会凸显出来。