Solaris（64位）开发者指南

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

文件号码 819–7054–10
2006 年 11 月

061229@15821

目录

前言

Solaris™ 操作环境的功能为满足用户需求而不断扩展。Solaris 操作环境旨在完全支持 32 位和 64 位体系结构。Solaris 操作环境为可以使用大文件和大虚拟地址空间的 64 位应用程序提供生成和运行环境。与此同时，Solaris 操作环境继续为 32 位应用程序提供最大程度的源代码兼容性、最大程度的二进制兼容性和互操作性。实际上，在 Solaris 64 位实现上运行和生成的大多数系统命令都是 32 位程序。

注 – 此 Solaris 发行版支持使用以下 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。支持的系统可以在 `http://www.sun.com/bigadmin/hcl` 上的《Solaris 10 Hardware Compatibility List》中找到。本文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，术语 "x86" 是指使用与 AMD64 或 Intel Xeon/Pentium 产品系列兼容的处理器生产的 64 位和 32 位系统。若想了解本发行版支持哪些系统，请参见《Solaris 10 Hardware Compatibility List》。

---

32 位和 64 位应用程序开发环境之间的主要差异在于 32 位应用程序基于 ILP32 数据模型（其中类型为 `int`、`long` 的数据和指针是 32 位的），而 64 位应用程序则基于 LP64 模型（其中类型为 `long` 的数据和指针是 64 位，其他基本类型的长度与在 ILP32 中的长度相同）。

大多数应用程序都可以保留为 32 位程序，并且无需进行任何更改。仅当应用程序具有以下一个或多个要求时，才需要进行转换：

- 需要 4 GB 以上的虚拟地址空间
- 使用 `libkvm` 库、`/dev/mem` 文件或 `/dev/kmem` 文件来读取和解释内核内存
- 使用 `/proc` 来调试 64 位进程
- 使用仅有 64 位版本的库
- 需要完全 64 位寄存器来执行高效的 64 位运算

对于特定的互操作性问题，还可能需要对代码进行更改。例如，如果应用程序使用大于 2 GB 的文件，则可能需要将应用程序转换为 64 位。

在某些情况下，可能会出于性能方面的原因需要将应用程序转换为 64 位。例如，可能需要 64 位寄存器以执行高效的 64 位运算，或者可能需要利用 64 位指令集所提供的其他提高的性能。

目标读者

本文档是为 C 和 C++ 开发者编写的，用于指导您如何确定应用程序是 32 位还是 64 位。本文档提供以下内容：

- 32 位和 64 位应用程序环境之间相似处和差异的列表
- 有关如何编写可在这两个环境之间移植的代码的说明
- 有关操作环境所提供的用于开发 64 位应用程序的工具的说明

本书的结构

本书由以下各章组成：

- 第 1 章介绍 64 位计算的意图并概述 64 位应用程序的优点。
- 第 2 章说明 Solaris 32 位和 64 位生成环境和运行时环境之间的差异。所编写的信息有助于应用程序开发者确定是否需要将代码转换为对于 64 位安全的代码以及何时进行转换比较适当。
- 第 3 章重点介绍 32 位应用程序和 64 位应用程序之间的相似处，还介绍了 64 位接口。
- 第 4 章介绍如何将当前的 32 位代码转换为对于 64 位安全的代码，还介绍了可用于使转换更容易的工具。本章重点介绍如何编写可移植代码。此信息适用于转换现有的应用程序或编写在 32 位和 64 位环境中均能运行的新应用程序。
- 第 5 章重点介绍生成环境（包括头文件、编译器和库）、打包指南和调试工具。
- 第 6 章概述 64 位系统编程、ABI 和一些性能问题。
- 附录 A 重点介绍许多在 64 位应用程序开发环境中已更改的派生类型。
- 附录 B 提供有关 64 位实现和应用程序开发环境中最常见问题的答案。

相关书籍

建议进一步阅读以下书籍：

- 《American National Standard for Information Systems Programming Language - C》，ANSI X3.159-1989
- 《SPARC Architecture Manual, Version 9》，SPARC International
- 《SPARC Compliance Definition, Version 2.4》，SPARC International
- 《*Large Files in Solaris: A White Paper*》，文件号码：96115-001
- 《*Solaris 10 Reference Manual*》
- 《*Writing Device Drivers*》，文件号码：816－4854
- 《*Sun Studio 10: C User's Guide*》，文件号码：819-0494-10

联机访问 Sun 文档

可以通过 docs.sun.com℠ Web 站点联机访问 Sun 技术文档。您可以浏览 docs.sun.com 文档库或查找某个特定的书名或主题。URL 为 http://docs.sun.com。

印刷约定的含义

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的不同	`machine_name%` **`su`** `Password:`
AaBbCc123	命令行占位符：使用实名或值进行替换	要删除文件，请键入 **`rm`** *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *class* 选项。 注意：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表列出了 C shell、Bourne shell 和 Korn shell 的缺省系统提示符和超级用户提示符。

表 P–2 Shell 提示符

Shell	提示符
C shell	`machine_name%`
C shell 超级用户	`machine_name#`
Bourne shell 和 Korn shell	`$`
Bourne shell 和 Korn shell 超级用户	`#`

注 – 术语 "IA-32" 是指 Intel 32 位处理器体系结构。此体系结构包括 Pentium、Pentium Pro、Pentium II、Pentium II Xeon 和 Pentium III 处理器以及由 AMD 和 Cyrix 制造的兼容微处理器芯片。

◆ ◆ ◆ 第 1 章

1 64 位计算

随着应用程序日渐多功能化和复杂化，同时数据集的大小不断增长，现有应用程序对地址空间的要求也持续增加。目前，某些类的应用程序需要超过 32 位系统的 4 GB 地址空间限制。以下举例说明了超过 4 GB 地址空间的应用程序：

- 各种数据库应用程序，特别是那些执行数据挖掘的应用程序
- Web 高速缓存和 Web 搜索引擎
- CAD（计算机辅助设计）和 CAE（计算机辅助工程）模拟组件以及建模工具
- 科学计算

开发 64 位计算环境的主要目的是为了使这些应用程序和其他大型应用程序能够高效运行。

突破 4 GB 限制

图 1–1 中的图表绘制了在安装了大量物理内存的计算机上，典型的性能与所运行的应用程序的问题大小之间的关系。如果问题非常小，则数据高速缓存 (D$) 或外部高速缓存 (E$) 中可以容纳整个程序。但是，程序的数据区域最终会变得非常大，以至于占满 32 位应用程序的整个 4 GB 虚拟地址空间。

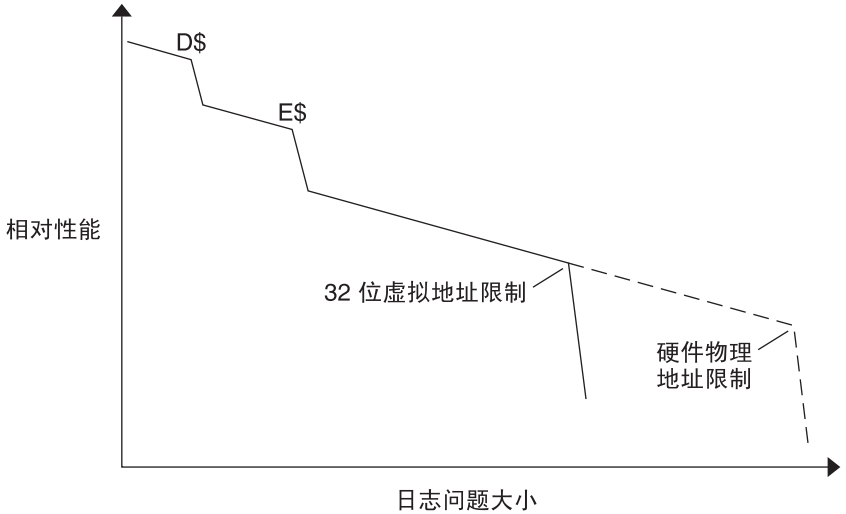


图 1–1 典型性能与问题大小之间的曲线

超出 32 位虚拟地址限制之后，应用程序编程人员仍可以处理大型问题。对于超出 32 位虚拟地址空间限制的应用程序，通常可以在主存储器和辅助存储器（例如，磁盘）之间分割应用程序数据集。遗憾的是，向磁盘驱动器传入或传出数据远比在内存间传输数据慢（相差若干个数量级）。

现在，许多服务器都可以处理 4 GB 以上的物理内存。高端台式机遵循同样的趋势，但是，没有任何 32 位程序可直接处理 4 GB 以上的空间。不过，64 位应用程序可以使用 64 位虚拟地址空间功能来允许直接处理最多为 18 EB（1 EB 大约等于 10^{18} 个字节）的空间进行寻址。这样，较大的问题便可以直接在主存储器中进行处理。对于可伸缩的多线程应用程序，可以向系统中添加更多处理器以进一步加快应用程序的执行速度。此类应用程序仅受计算机中物理内存量的限制。

不过，很显然，对于各类应用程序来说，能够直接在主存储器中处理大型问题无疑**是** 64 位计算机的主要性能优势。

- 主存储器中可驻留更大的数据库。
- 主存储器中可以容纳更大的 CAD/CAE 模型和模拟对象。
- 主存储器中可以容纳更大的科学计算问题。
- Web 高速缓存可在内存中包含更多内容，从而缩短了延迟时间。

突破大地址空间限制

迫切需要创建 64 位应用程序的其他原因还包括：

- 需要针对 64 位整数值执行大量计算，这些值使用 64 位处理器更宽的数据路径来提高性能。
- 几个系统接口已进行了增强或者取消了限制，因为用来增强这些接口的基础数据类型的长度已变大。
- 需要获取 64 位指令集的性能益处（如改进的调用约定和充分利用寄存器集）。

◆ ◆ ◆ 第 2 章

2 何时使用 64 位

对于应用程序开发者，Solaris 64 位和 32 位操作环境之间的主要区别在于所使用的 C 数据类型模型。64 位版本使用 LP64 模型，其中类型为 `long` 的数据和指针是 64 位。其他所有基础数据类型仍然与基于 ILP32 模型的 32 位实现中的数据类型相同。在 ILP32 模型中，类型为 `int`、`long` 的数据和指针是 32 位的。第 3 章中将对这些模型进行更详细地介绍。

只有极少数的应用程序真正**要求**进行转换。大多数应用程序都可以保留为 32 位应用程序，并且仍可以在 64 位操作系统上运行，而无需对代码进行任何更改或重新编译。实际上，不需要 64 位功能的 32 位应用程序应保留为 32 位，以便最大程度地提高可移植性。

对于具有以下特征的应用程序，可能需要进行转换：

- 可以利用 4 GB 以上的虚拟地址空间
- 受 32 位接口的限制
- 可以利用完全 64 位寄存器来执行高效的 64 位运算
- 可以利用 64 位指令集所提供的性能方面的改进

对于具有以下特征的应用程序，可能需要进行转换：

- 使用 `libkvm`、`/dev/mem` 或 `/dev/kmem` 来读取和解释内核内存
- 使用 `/proc` 来调试 64 位进程
- 使用仅有 64 位版本的库

对于某些特定的互操作性问题需要对代码进行更改。同样，如果应用程序使用大于 2 GB 的文件，请考虑将其转换为 64 位应用程序，而不要直接使用大文件 API。

这些项将在以下几节中进一步介绍。

主要功能

为了说明 Solaris 操作环境同时支持 32 位和 64 位，下图并排显示了两个栈。左边的系统仅支持在使用 32 位设备驱动程序的 32 位内核中运行 32 位库和应用程序。右边的系统所支持的 32 位应用程序和库与左边的系统相同。该系统还**同时**支持使用 64 位设备驱动程序的 64 位内核顶部的 64 位库和应用程序。

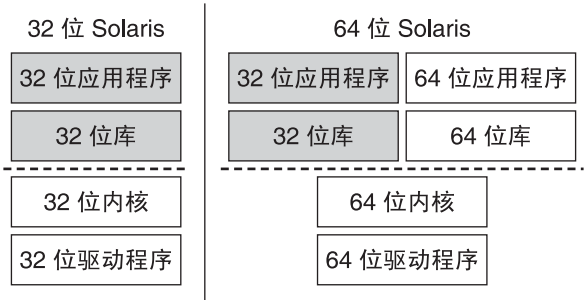


图 2–1 Solaris 操作环境体系结构

64 位环境的主要功能包括支持以下各项：

- 大虚拟地址空间
- 大文件
- 64 位运算
- 取消某些系统限制

大虚拟地址空间

在 64 位环境中，一个进程最多可以使用 64 位（即 18 EB）虚拟地址空间。该虚拟地址空间大约是 32 位进程当前所使用最大空间的 *40* 亿倍。

注 – 由于存在硬件限制，因此某些平台可能不支持完全 64 位的地址空间。

大文件

如果应用程序只需支持大文件，则该应用程序可以保留为 32 位并使用大文件接口。但是，如果可移植性并不是主要问题，请考虑将应用程序转换为 64 位程序。64 位程序可以通过一整套接口来充分利用 64 位功能。

64 位运算

很久以前在早期发行版的 32 位 Solaris 中便已提供了 64 位运算。但是，64 位实现现在使用完全 64 位计算机寄存器来进行整数运算和参数传递。通过 64 位实现，应用程序可充分利用 64 位 CPU 硬件的功能。

取消系统限制

本质上，64 位系统接口的性能要优于一些等效的 32 位接口。对于担心 32 位 `time_t` 用完时间时会出现 2038 年问题的应用程序编程人员来说，可以使用 64 位 `time_t`。尽管 2038 年看起来还很遥远，但是，对于需要执行与将来事件（如抵押）有关的计算的应用程序，则可能需要扩展的时间功能。

互操作性问题

下面列出了一些互操作性问题，关于要求使应用程序对于 64 位安全，或者对应用程序进行更改，使其与 32 位和 64 位应用程序均能够互操作：

- 客户机和服务器传输
- 处理持久性数据的程序
- 共享内存

内核内存读取器

由于内核是在内部使用 64 位数据结构的 LP64 对象，因此，使用 `libkvm`、`/dev/mem` 或 `/dev/kmem` 的现有 32 位应用程序将无法正常工作，必须将其转换为 64 位程序。

`/proc` 限制

使用 `/proc` 的 32 位程序可以查看 32 位进程，但是无法识别 64 位进程的所有属性。描述该进程的现有接口和数据结构不够大，因此无法包含所涉及的 64 位值。此类程序需要重新编译为 64 位程序，以便使其可同时用于 32 位进程和 64 位进程。对于调试器来说，这是一个最为典型的问题。

64 位库

32 位应用程序必须链接到 32 位库，64 位应用程序必须链接到 64 位库。除已过时的库以外，所有系统库均提供了 32 位和 64 位两种版本。

评估转换工作

确定要将应用程序转换为完全 64 位程序之后，一定要注意，对于许多应用程序，只需执行很少的操作即可完成此目标。其余各章将讨论如何评估应用程序以及转换过程中所涉及的工作。

◆ ◆ ◆ 第 3 章

3 比较 32 位接口和 64 位接口

正如第 11 页中的 “突破 4 GB 限制”中所讨论的那样，大多数 32 位应用程序将在 Solaris 64 位操作环境中照常运行。一些应用程序可能只需重新编译为 64 位应用程序，其他应用程序则需要进行转换。本章所面向的开发者应根据第 11 页中的 “突破 4 GB 限制”中所讨论的各项，确定其应用程序是需要重新编译还是转换成 64 位。

应用编程接口

64 位操作环境中支持的 32 位应用编程接口 (application programming interface, API) 与 32 位操作环境中支持的 API 相同。因此，对于 32 位应用程序，无需在 32 位环境和 64 位环境之间进行更改。但是，**重新编译**为 64 位应用程序可能需要执行清理操作。有关如何清理代码以实现 64 位应用程序的指导说明，请参见第 4 章中所定义的规则。

缺省的 64 位API 基本属于 UNIX 98 系列的 API，其规范是使用派生类型的术语编写的。通过将某些派生类型扩展到 64 位值可获取 64 位版本。使用这些 API 的正确编写的应用程序可以在 32 位环境和 64 位环境之间以源代码形式进行移植。Solaris 10 中也提供了 UNIX 2001 API 系列，请参见 `standards`(5)。

应用程序二进制接口

SPARC V8 ABI 是现有的特定于处理器的应用程序二进制接口 (Application Binary Interface, ABI)，32 位 SPARC 版本的 Solaris 实现以该接口为基础。SPARC V9 ABI 对 SPARC V8 ABI 进行了扩展，使其可支持 64 位操作，并为此扩展的体系结构定义了新功能。有关其他信息，请参见第 53 页中的 “SPARC V9 ABI 特征”。

i386 ABI 是特定于处理器的 ABI，32 位版本的 Solaris (Intel Platform Edition) 以该接口为基础。

对于 Solaris 10 发行版，amd64 ABI 是特定于处理器的 ABI，x86 系统中 64 位版本的 Solaris 以该接口为基础。amd64 ABI 支持 64 位操作并为新体系结构定义了新功能。使

用 64 位 ABI 的程序的性能可能会优于对应的 32 位程序。支持 amd64 ABI 的处理器也支持 i386 ABI。有关其他信息，请参见第 56 页中的 “AMD64 ABI 特征”。

32 位应用程序和 64 位应用程序之间的兼容性

以下几节将讨论 32 位应用程序和 64 位应用程序之间不同级别的兼容性。

应用程序二进制对象

现有的 32 位应用程序可以在 32 位或 64 位操作环境中运行，仅有那些使用 `libkvm`、`/dev/mem`、`/dev/kmem` 或 `/proc` 的应用程序例外。有关更多信息，请参见第 11 页中的 “突破 4 GB 限制”。

应用程序源代码

对于 32 位应用程序保留了源代码级别的兼容性。对于 64 位应用程序，进行的主要更改与应用编程接口中所使用的派生类型有关。正确使用派生类型和接口的应用程序，其源代码对于 32 位是兼容的，并且更便于转换到 64 位。

设备驱动程序

由于 32 位设备驱动程序无法用于 64 位操作系统中，因此这些驱动程序必须重新编译为 64 位对象。此外，64 位驱动程序还需要同时支持 32 位应用程序和 64 位应用程序。64 位操作环境附带的所有驱动程序均可同时支持 32 位应用程序和 64 位应用程序。但是，DDI（设备驱动程序接口）支持的基础驱动程序模型和接口没有发生变化。主要工作是清理要在 LP64 环境中进行更正的代码。有关更多信息，请参见《编写设备驱动程序》手册。

运行的是哪种 Solaris 操作环境？

Solaris 操作环境可以同时支持两个一流的 ABI。换句话说，两个完全正常工作的独立系统调用路径可连接到 64 位内核中，这两组库均支持应用程序。

64 位操作系统只能在 64 位 CPU 硬件上运行，而 32 位版本则可以在 32 位 CPU 硬件上或 64 位 CPU 硬件上运行。由于 Solaris 32 位和 64 位操作环境看上去非常相似，因此可能不易分辨特定硬件平台上运行的是哪个版本的操作环境。

要确定系统上运行的是哪个版本的操作环境，最容易的方法就是使用 `isainfo` 命令。此新命令可以列显有关系统上所支持的应用程序环境的信息。

以下举例说明了在运行 64 位操作系统的 UltraSPARC 系统中执行的 `isainfo` 命令：

```
% isainfo -v

64-bit sparcv9 applications

32-bit sparc applications
```

以下是在运行 32 位 Solaris 操作系统的 x86 系统中执行的 `isainfo` 命令：

```
% isainfo -v

32-bit i386 applications
```

以下是在运行 64 位 Solaris 操作系统的 x86 系统上执行的 `isainfo` 命令：

```
% isainfo -v

64-bit amd64 applications

32-bit i386 applications
```

注 – 并非所有 x86 系统都能够运行 64 位内核。在这种情况下，如果系统运行的是 Solaris 操作环境，则内核会在 32 位模式下运行。

`-n` 是 `isainfo`(1) 命令的一个有用选项，用来列显所运行的平台的本机指令集：

```
% isainfo -n

sparcv9
```

`-b` 选项用来列显对应本机应用程序环境的地址空间中的位数：

```
% echo "Welcome to "‘isainfo -b‘"-bit Solaris"

Welcome to 64-bit Solaris
```

必须在早期版本的 Solaris 操作环境中运行的应用程序可以确定 64 位功能是否可用，方法是检查 `uname`(1) 的输出或检查 `/usr/bin/isainfo` 是否存在。

`isalist`(1) 是 `isainfo` 的相关命令，它更适合在 shell 脚本中使用。`isalist` 可用来列显平台所支持的指令集的完整列表。但是，随着指令集扩展数量的增加，所有子集的列表的局限性也更加明显。建议用户将来不要依赖此接口。

如果用户要创建依赖指令集扩展的库，则应使用动态链接程序的硬件功能。使用 isainfo 命令可以确定当前平台上的指令集扩展。

```
% isainfo -x
amd64: sse2 sse fxsr amd_3dnowx amd_3dnow amd_mmx mmx cmov amd_sysc cx8 tsc fpu
i386: sse2 sse fxsr amd_3dnowx amd_3dnow amd_mmx mmx cmov amd_sysc cx8 tsc fpu
```

◆ ◆ ◆ 第 4 章

4 转换应用程序

对于应用程序开发者而言，需要面对以下两个有关转换的基本问题：

- 不同数据模型之间的数据类型一致性
- 使用不同数据模型的应用程序之间的互操作

尝试维护包含尽可能少的 `#ifdefs` 的单一源代码通常比维护多个源代码树更好。本章将指导您编写在 32 位和 64 位环境中均能够正常工作的代码。最理想的情况是，对当前代码的转换只需重新编译以及重新链接到 64 位的库。然而，还有需要更改代码的情况，本章讨论了有助于使这类转换更容易的工具。

数据模型

如前所述，32 位环境和 64 位环境之间最大的区别在于两种基础数据类型的变化。

用于 32 位应用程序的 C 数据类型模型是 ILP32 模型，之所以这样命名，是因为类型 `int`、`long` 和指针均为 32 位。LP64 数据模型是用于 64 位应用程序的 C 数据类型模型，业界公司联盟已就此达成一致。之所以这样命名，是因为 long 和指针类型的数据长度会增加到 64 位。其余的 C 类型 `int`、`short` 和 `char` 均与 ILP32 模型中的相应类型相同。

以下的程序样例 `foo.c` 直接对比说明了 LP64 数据模型与 ILP32 数据模型的不同结果。同一个程序既可以编译为 32 位程序，也可以编译为 64 位程序。

```
#include <stdio.h>

int

main(int argc, char *argv[])

{

        (void) printf("char is \t\t%lu bytes\n", sizeof (char));
```

```
        (void) printf("short is \t%lu bytes\n", sizeof (short));
        (void) printf("int is \t\t%lu bytes\n", sizeof (int));
        (void) printf("long is \t\t%lu bytes\n", sizeof (long));
        (void) printf("long long is \t\t%lu bytes\n", sizeof (long long));
        (void) printf("pointer is \t%lu bytes\n", sizeof (void *));
        return (0);
}
```

32 位编译的结果是 :

```
% cc -O -o foo32 foo.c
% foo32
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          4 bytes
long long is     8 bytes
pointer is       4 bytes
```

64 位编译的结果是 :

```
% cc -xarch=generic64 -O -o foo64 foo.c
% foo64
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          8 bytes
long long is     8 bytes
pointer is       8 bytes
```

注 – 缺省编译环境旨在最大化可移植性，即创建 32 位应用程序。

C 整型之间的标准关系仍然适用。

```
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)
```

表 4–1 列出了基本 C 数据类型及其在 LP32 和 LP64 数据类型模型中的对应长度（以位为单位）。

表 4–1 数据类型的长度（以位为单位）

C 数据类型	ILP32	LP64
`char`	8	不变
`short`	16	不变
`int`	32	不变
`long`	32	**64**
`long long`	64	不变
`pointer`	32	**64**
`enum`	32	不变
`float`	32	不变
`double`	64	不变
`long double`	128	不变

某些旧的 32 位应用程序可互换使用 `int`、`long` 和指针类型。类型为 `long` 的数据和指针长度在 LP64 数据模型中会增加。需要注意的是，单是这种变化就可导致许多 32 位到 64 位的转换问题。

此外，声明和强制类型转换在说明意图时也会变得非常重要。类型更改时，表达式的求值方式会受到影响。标准 C 转换规则的作用受数据类型长度变化的影响。要充分说明意图，可能需要声明常量的类型。为了确保按照预期的方式对表达式求值，可能还需要在表达式中进行强制类型转换。对表达式进行正确的求值在符号扩展时尤其重要，在这种情况下进行显式强制类型转换可能是实现预期效果所必需的。

对于内置的 C 运算符、格式字符串、汇编语言以及兼容性和互操作性，还会出现其他问题。

本章的其余部分将通过介绍以下内容来给出解决这些问题的建议：

- 更详细地解释以上概述的问题

- 介绍某些可用来使代码对于 32 位和 64 位均安全的派生类型和头文件
- 介绍有助于使代码对于 64 位安全的工具
- 提供使代码可在 32 位和 64 位环境之间移植的一般规则

实现单一源代码

以下几节介绍了一些可供应用程序开发者使用的资源，可帮助编写能够同时支持 32 位编译和 64 位编译的单一源代码。

系统头文件 <sys/types.h> 和 <inttypes.h> 中包含有助于使应用程序对于 32 位和 64 位均安全的常量、宏和派生类型。尽管详细讨论这些内容超出了本文档的范围，但是以下各节以及附录 A 中仍讨论了其中的部分内容。

功能测试宏

包含 <sys/types.h> 的应用程序源文件可以通过包括 <sys/isa_defs.h> 使编程模型符号 _LP64 和 _ILP32 的定义可用。

有关预处理程序符号（_LP64 和 _ILP32）和宏（_LITTLE_ENDIAN 和 _BIG_ENDIAN6）的信息，请参见 types(3HEAD)。

派生类型

使用系统派生类型有助于使代码对于 32 位和 64 位均安全，这是由于派生类型本身对于 ILP32 和 LP64 数据模型均安全。通常，使用派生类型以便于更改是良好的编程做法。如果数据模型在将来发生变化，或者在移植到其他平台时，只需更改系统派生类型即可，而无需更改应用程序。

<sys/types.h> 文件

<sys/types.h> 头文件中包含大量应在适当时机使用的基本派生类型。特别是以下几种类型颇受关注：

clock_t	类型 clock_t 表示系统时间（以时钟周期为单位）。
dev_t	类型 dev_t 用于设备号。
off_t	类型 off_t 用于文件大小和偏移量。
ptrdiff_t	类型 ptrdiff_t 是一种带符号整数类型，用于对两个指针执行减法运算后所得的结果。
size_t	类型 size_t 用于内存中对象的大小（以字节为单位）。

`ssize_t`	带符号的大小类型 `ssize_t` 供返回字节计数或错误指示的函数使用。
`time_t`	类型 `time_t` 用于时间（以秒为单位）。

所有这些类型在 ILP32 编译环境中都保持 32 位值，并会在 LP64 编译环境中增加到 64 位值。

其中某些类型的用法将在本章第 32 页中的 “转换为 LP64 的指导原则”中更详细地介绍。

<inttypes.h> 文件

在 Solaris 2.6 发行版中添加了头文件 `<inttypes.h>`，程序员可利用它提供的常量、宏和派生类型使其代码与显式指定大小的数据项兼容，而不管编译环境如何。该文件中包含用来处理 8 位、16 位、32 位和 64 位对象的机制，它是 ANSI C 议案的一部分，可以与 ISO/JTC1/SC22/WG14 C 委员会对当前 ISO C 标准（即 ISO/IEC 9899:1990 编程语言－C）所进行修订的工作草案保持一致。

`<inttypes.h>` 提供的基本功能包括：

- 一组定宽的整数类型
- `uintptr_t` 和其他有用的类型
- 常量宏
- 限制
- 格式字符串宏

以下各节中将对这些功能进行更详细的讨论。

定宽的整数类型

`<inttypes.h>` 提供的定宽整数类型同时包括带符号整数类型（如 `int8_t`、`int16_t`、`int32_t` 和 `int64_t`）以及无符号整数类型（如 `uint8_t`、`uint16_t`、`uint32_t` 和 `uint64_t`）。定义为可具有指定位数的最短整数类型的派生类型包括 `int_least8_t`、`int_least64_t`、`uint_least8_t` 和 `uint_least64_t`。

不应不加选择地使用这些定宽类型。例如，类型 `int` 可以继续用于循环计数器和文件描述符，类型 `long` 可用于数组索引。另一方面，对于以下各项的显式二进制表示形式，则应使用定宽类型：

- 盘上数据
- 线上数据
- 硬件寄存器

- 二进制接口规格（包含显式指定了大小的对象，或者涉及 32 位和 64 位程序之间的共享或通信）
- 二进制数据结构（供 32 位和 64 位程序通过共享内存和文件等进行使用）

uintptr_t 和其他有用的类型

<inttypes.h> 提供的其他有用类型包括大小足以包含一个指针的带符号整数类型和无符号整数类型。这些类型以 intptr_t 和 uintptr_t 形式提供。此外，还会将 intmax_t 和 uintmax_t 分别定义为可用的最长（以位为单位）带符号整数类型和无符号整数类型。

选用 uintptr_t 类型作为指针的整数类型比使用基本类型（如 unsigned long）要好。尽管在 ILP32 和 LP64 数据模型中，类型 unsigned long 与指针的长度相同，但如果使用 uintptr_t，则只需在使用其他数据模型时更改 uintptr_t 的定义即可。这会使其可移植到许多其他系统中，并且还可以在 C 中更为清楚地表达意图。

需要执行地址运算时，intptr_t 和 uintptr_t 类型对于强制转换指针非常有用。因此，应使用这些类型，而不是使用 long 或 unsigned long 类型。

注 – 使用 uintptr_t 进行强制类型转换通常比使用 intptr_t 安全，在进行比较时尤为安全。

常量宏

提供宏的目的在于指定给定常量的大小和符号。这些宏包括 INT8_C(c)、INT64_C(c)、UINT8_C(c) 和 UINT64_C(c) 等。基本上，这些宏会在常量的末尾放置一个 l、ul、ll 或 ull（如有必要）。例如，对于 ILP32，INT64_C(1) 会在常量 1 后面附加 ll；对于 LP64，则附加 l。

用来使常量成为最长类型的宏包括 INTMAX_C(c) 和 UINTMAX_C(c)。这些宏对于指定第 32 页中的 “转换为 LP64 的指导原则”中介绍的常量类型会非常有用。

<inttypes.h> 定义的限制

由 <inttypes.h> 定义的限制是用于为各种整数类型指定最小值和最大值的常量，其中包括每个定宽类型的最小值（INT8_MIN 和 INT64_MIN 等）和最大值（如 INT8_MAX 和 INT64_MAX 等）及其对应的无符号的最小值和最大值。

该文件还提供了每个最短长度类型的最小值和最大值，其中包括 INT_LEAST8_MIN、INT_LEAST64_MIN、INT_LEAST8_MAX 和 INT_LEAST64_MAX 等及其对应的无符号的最小值和最大值。

最后，该文件定义了支持的最长整数类型的最小值和最大值，其中包括 INTMAX_MIN 和 INTMAX_MAX 及其对应的无符号的最小值和最大值。

格式字符串宏

<inttypes.h> 文件还提供了用于指定 printf 和 scanf 格式说明符的宏。实际上，这些宏会根据内置于宏名称中的参数的位数，在格式说明符前面放置一个 l 或 ll，将参数指定为 long 或 long long 类型。

printf(3C) 的宏以十进制、八进制、无符号和十六进制格式列显 8 位、16 位、32 位和 64 位整数以及最短和最长整数类型。例如，以十六进制表示法列显 64 位整数：

```
int64_t i;

printf("i =%" PRIx64 "\n", i);
```

同样，scanf(3C) 的宏也以十进制、八进制、无符号和十六进制格式读取 8 位、16 位、32 位和 64 位整数以及最长整数类型。例如，读取无符号的 64 位十进制整数：

```
uint64_t u;

scanf("%" SCNu64 "\n", &u);
```

请勿毫无限制地使用这些宏，最好将其与定宽类型一起使用。有关更多详细信息，请参阅第 27 页中的 “定宽的整数类型”一节。

工具支持

Sun Studio 10 编译器中提供的 lint 程序可以检测潜在的 64 位问题，并有助于使代码对于 64 位安全。此外，该 C 编译器的 -v 选项也会非常有用。该选项可指示编译器执行更严格的附加语义检查，还会针对指定的文件启用某些与 lint 相似的检查。

有关 C 编译器和 lint 的功能的更多信息，请参见《*Sun Studio 10: C User's Guide*》。

用于 32 位和 64 位环境的 lint

lint 既可用于 32 位代码又可用于 64 位代码。对于要在 32 位和 64 位环境中运行的代码，请使用 -errchk=longptr64 选项。-errchk=longptr64 选项用于检查是否可将代码移植到长整数和指针的长度为 64 位、无格式整数的长度为 32 位的环境中。

对于要在 64 位 SPARC 环境中运行的 lint 代码，应当使用 -Xarch=v9 选项。将 -errchk=longptr64 选项和 -Xarch=v9 选项一起使用，可以针对要在 64 位 SPARC 上运行的代码生成有关潜在 64 位程序问题的警告。

从 Solaris 10 发行版开始，对于要在 64 位 AMD 环境中运行的 lint 代码，应当使用 -Xarch=amd64 选项。

注 – lint 的 -D__sparcv9 选项已不再需要，不应当再使用。

有关 lint 选项的说明，请参见《*Sun Studio 10: C User's Guide*》。

生成警告时，lint(1) 会列显违例代码的行号、描述问题的警告消息以及是否涉及指针的注释。它还会指明所涉及类型的长度。如果确定涉及指针并且知道数据类型的长度，则会便于查找特定的 64 位问题，并避免 32 位和更短类型之间的已有问题。

注 – 尽管 lint 会给出有关潜在的 64 位问题的警告，但也无法检测所有的问题。请务必牢记，在 lint 生成的警告中，并非所有警告都是真正的 64 位问题。在许多情况下，符合应用程序意图并且正确无误的代码也会生成警告。

以下的程序和 lint(1) 输出样例说明了非纯 64 位代码中可能出现的大多数 lint 警告。

```
1   #include <inttypes.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   static char chararray[] = "abcdefghijklmnopqrstuvwxyz";
6
7   static char *myfunc(int i)
8   {
9       return(& chararray[i]);
10  }
11
12  void main(void)
13  {
14      int    intx;
15      long    longx;
```

```
16        char    *ptrx;
17
18        (void) scanf("%d", &longx);
19        intx = longx;
20        ptrx = myfunc(longx);
21        (void) printf("%d\n", longx);
22        intx = ptrx;
23        ptrx = intx;
24        intx = (int)longx;
25        ptrx = (char *)intx;
26        intx = 2147483648L;
27        intx = (int) 2147483648L;
28        ptrx = myfunc(2147483648L);
29    }
```

```
(19) warning: assignment of 64-bit integer to 32-bit integer
(20) warning: passing 64-bit integer arg, expecting 32-bit integer: myfunc(arg 1)
(22) warning: improper pointer/integer combination: op "="
(22) warning: conversion of pointer loses bits
(23) warning: improper pointer/integer combination: op "="
(23) warning: cast to pointer from 32-bit integer
(24) warning: cast from 64-bit integer to 32-bit integer
(25) warning: cast to pointer from 32-bit integer
```

```
(26) warning: 64-bit constant truncated to 32 bits by assignment

(27) warning: cast from 64-bit integer constant expression to 32-bit integer

(28) warning: passing 64-bit integer constant arg, expecting 32-bit integer: myfunc(arg 1)

function argument ( number ) type inconsistent with format

    scanf (arg 2)     long * :: (format) int *    t.c(18)

    printf (arg 2)     long  :: (format) int     t.c(21)
```

（仅当常量表达式不适合它所强制转换到的类型时，才会发出从以上代码样例的第 27 行中生成的 `lint` 警告。）

通过在上一行中加一个 `/*LINTED*/` 注释，可以禁止对给定的源代码行发出警告。如果确实要让代码以特定方式运行（例如强制类型转换和赋值）时，这种做法会很有用。使用 `/*LINTED*/` 注释时请务必谨慎，因为它可能会掩盖真实问题。有关更多信息，请参阅《*Sun Studio 10: C User's Guide*》或 `lint(1)` 手册页。

转换为 LP64 的指导原则

使用 `lint(1)` 时，请记住，并非所有问题都将生成 `lint(1)` 警告，并且并非所有 `lint(1)` 警告都表示要求进行更改。请针对不同意图检查各种可能性。以下示例说明了转换代码时可能遇到的一些常见问题。在适当情况下，会显示对应的 `lint(1)` 警告。

请勿假设 `int` 和指针的长度相同

由于类型为 `int` 的数据和指针在 ILP32 环境中长度相同，因此许多代码会依赖于这一假设。通常会将指针强制转换为 `int` 或 `unsigned int` 类型以进行地址运算。但是，由于类型为 `long` 的数据和指针在 ILP32和 LP64 数据类型模型中长度均相同，因此可以将指针强制转换为 `long` 类型。请使用类型 `uintptr_t` 而不是显式使用 `unsigned long`，因为前者可更贴切地表达意图并使代码具有更强的可移植性，从而使其不会受到将来变化的影响。例如，

```
char *p;

p = (char *) ((int)p & PAGEOFFSET);
```

将生成以下警告：

```
warning: conversion of pointer loses bits
```

使用以下代码将生成不带警告的结果：

```
char *p;

p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

请勿假设 int 和 long 的长度相同

由于类型为 int 和 long 的数据在 ILP32 中从未真正加以区分，因此许多现有的代码会隐式或显式地假设它们可互换使用，并不加区分地进行使用。必须修改做出此假设的任何代码，使其可同时用于 ILP32 和 LP64。类型 int 和 long 在 ILP32 数据类型模型中均为 32 位，而类型 long 在 LP64 数据模型中则为 64 位。例如，

```
int waiting;

long w_io;

long w_swap;

...

waiting = w_io + w_swap;
```

将生成以下警告：

```
warning: assignment of 64-bit integer to 32-bit integer
```

符号扩展

转换到 64 位时，经常会遇到非故意符号扩展问题。该问题很难在实际发生前检测到，因为 lint(1) 不会针对它发出警告。此外，类型转换和提升规则也有些模糊。要解决非故意符号扩展的问题，必须使用显式强制类型转换来实现预期结果。

要了解出现符号扩展的原因，了解 ANSI C 的转换规则会有所帮助。以下是可能会导致 32 位和 64 位整数值之间大多数符号扩展问题的转换规则：

1. 整型提升

   无论有无符号，均可以在任何调用类型为 int 的表达式中使用 char、short、枚举类型或位字段。如果 int 可以支持初始类型的所有可能值，则值会转换为 int 类型。否则，值会转换为 unsigned int 类型。
2. 带符号整数和无符号整数之间的转换

将带负号的整数提升为同一类型或更长类型的无符号整数时，该整数首先提升为更长类型的带符号相同值，然后再转换为无符号值。

有关转换规则的更详细讨论，请参阅 ANSI C 标准。该标准中还包括适用于普通运算转换和整数常量的规则。

以下示例编译为 64 位程序时，即使 `addr` 和 `a.base` 均是 `unsigned` 类型，`addr` 变量仍可成为带符号扩展变量。

示例 4–1 `test.c`

```
struct foo {

        unsigned int    base:19, rehash:13;

};



main(int argc, char *argv[])

{

        struct foo    a;

        unsigned long addr;



        a.base = 0x40000;

        addr = a.base << 13;        /* Sign extension here! */

        printf("addr 0x%lx\n", addr);



        addr = (unsigned int)(a.base << 13);  /* No sign extension here! */

        printf("addr 0x%lx\n", addr);

}
```

进行此符号扩展的原因是按以下方式应用了转换规则：

1. `a.base` 由于整型提升规则而从类型 `unsigned int` 转换为 `int`。因此，表达式 `a.base << 13` 的类型为 `int`，但是未进行符号扩展。

2. 表达式 a.base << 13 的类型为 int，但是在赋值给 addr 之前，由于带符号和无符号整型提升规则，会转换为类型 long，然后再转换为类型 unsigned long。从类型 int 转换为 long 时，会进行符号扩展。

```
% cc -o test64 -xarch=v9 test.c

% ./test64

addr 0xffffffff80000000

addr 0x80000000

%
```

如果将同一示例编译为 32 位程序，则不显示任何符号扩展：

```
% cc -o test32 test.c

% ./test32

addr 0x80000000

addr 0x80000000

%
```

使用指针运算而不是地址运算

通常，由于指针运算是独立于数据模型的，而地址运算则可能不是，因此使用指针运算比地址运算更好。此外，使用指针运算通常还可以简化代码。例如，

```
int *end;

int *p;

p = malloc(4 * NUM_ELEMENTS);

end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);
```

将生成以下警告：

```
warning: conversion of pointer loses bits
```

以下代码将生成不带警告的结果：

```
int *end;

int *p;

p = malloc(sizeof (*p) * NUM_ELEMENTS);

end = p + NUM_ELEMENTS;
```

对结构重新压缩

由于 long 和指针字段在 LP64 中会增加到 64 位，因此编译器可能会向结构中添加额外的填充内容，以满足对齐要求。对于 SPARCV9 ABI 和 amd64 ABI，所有类型的结构均与结构中最长成员的长度对齐。对结构重新压缩时，一个简单的规则就是，将 long 和指针字段移到结构开头，然后重新排列其余的字段，通常（但不总是）按长度的降序排列，具体取决于这些字段可以压缩的程度。例如，

```
struct bar {

        int i;

        long j;

        int k;

        char *p;

};            /* sizeof (struct bar) = 32 */
```

要获取更好的结果，请使用：

```
struct bar {

        char *p;

        long j;

        int i;

        int k;

};            /* sizeof (struct bar) = 24 */
```

注 – 基本类型的对齐方式在 i386 和 amd64 ABI 之间会发生变化。请参见第 58 页中的 “对齐问题”。

检查联合类型

请确保对联合类型进行检查，因为其字段的长度在 ILP32 和 LP64 数据类型模型之间可能会发生变化。例如，

```
typedef union {

       double   _d;

       long _l[2];

} llx_t;
```

应当为：

```
typedef union {

       double _d;

       int _l[2];

} llx_t;
```

指定常量类型

在某些常量表达式中，由于精度不够而会导致数据丢失。这些类型的问题很难发现。请在常量表达式中显式指定数据类型。通过向每个整型常量的末尾增加 {u,U,l,L} 的某些组合，可以指定其类型。另外，也可以使用强制类型转换来指定常量表达式的类型。例如，

```
int i = 32;

long j = 1 << i;        /* j will get 0 because RHS is integer expression */
```

应当为：

```
int i = 32;

long j = 1L << i;
```

注意隐式声明

在某些编译模式下，编译器可能会假设针对在模块中使用却未在外部定义或声明的任何函数或变量，使用 int 类型。编译器的隐式 int 类型声明会将以此方式使用的任何类型为 long 的数据和指针截断。函数或变量的相应 extern 类型声明应置于头文件而非 C 模块中。然后，使用此函数或变量的任何 C 模块中应包含此头文件。如果是系统头文件定义的函数或变量，则仍然需要在代码中包含正确的头文件。

例如，由于未声明 getlogin()，因此以下代码：

```
int

main(int argc, char *argv[])

{

        char *name = getlogin()

        printf("login = %s\n", name);

        return (0);

}
```

将生成以下警告：

```
warning: improper pointer/integer combination: op "="

warning: cast to pointer from 32-bit integer

implicitly declared to return int

getlogin        printf
```

要获取更好的结果，请使用：

```
#include <unistd.h>

#include <stdio.h>



int

main(int argc, char *argv[])

{
```

```
        char *name = getlogin();

        (void) printf("login = %s\n", name);

        return (0);

}
```

sizeof 是 unsigned long

在 LP64 环境中，sizeof 的有效类型为 size_t，该类型会实现为 unsigned long。有时，sizeof 会传递给需要使用类型为 int 的参数的函数，或者赋值给 int 类型的数据或强制转换为 int 类型。在某些情况下，这种截断可能会导致数据丢失。例如，

```
long a[50];

unsigned char size = sizeof (a);
```

将生成以下警告：

```
warning: 64-bit constant truncated to 8 bits by assignment

warning: initializer does not fit or is out of range: 0x190
```

使用强制类型转换说明意图

关系表达式可能会因转换规则而变得错综复杂。您应当通过在必要的地方添加强制类型转换来非常明确地指定表达式的求值方式。

检查格式字符串转换操作

对于类型为 long 或指针的参数，可能需要更改 printf(3C)、sprintf(3C)、scanf(3C) 和 sscanf(3C) 的格式字符串。对于要同时在 32 位和 64 位环境中运行的指针参数，格式字符串中给定的转换操作应为 %p。例如，

```
char *buf;

struct dev_info *devi;

...
```

```
(void) sprintf(buf, "di%x", (void *)devi);
```

将生成以下警告：

```
warning: function argument (number) type inconsistent with format

  sprintf (arg 3)     void *: (format) int
```

使用以下代码将生成不带警告的结果：

```
char *buf;

struct dev_info *devi;

...

(void) sprintf(buf, "di%p", (void *)devi);
```

另外，请检查以确保 buf 指向的存储器大小足以包含 16 个数字。对于类型为 long 的参数，long 类型的长度规范 l 应置于格式字符串中转换操作字符的前面。例如，

```
    size_t nbytes;

    ulong_t align, addr, raddr, alloc;

    printf("kalloca:%d%%%d from heap got %x.%x returns %x\n",

            nbytes, align, (int)raddr, (int)(raddr + alloc), (int)addr);
```

将生成以下警告：

```
warning: cast of 64-bit integer to 32-bit integer

warning: cast of 64-bit integer to 32-bit integer

warning: cast of 64-bit integer to 32-bit integer
```

以下代码将生成不带警告的结果：

```
    size_t nbytes;

    ulong_t align, addr, raddr, alloc;

    printf("kalloca:%lu%%%lu from heap got %lx.%lx returns %lx\n",

            nbytes, align, raddr, raddr + alloc, addr);
```

其他注意事项

其余指导原则将重点说明将应用程序转换为完全 64 位程序时遇到的常见问题。

长度增加的派生类型

许多派生类型已进行了更改，以便在 64 位应用程序环境中表示 64 位值。此更改不会影响 32 位应用程序；但是，使用或导出这些类型所描述的数据的任何 64 位应用程序均需要重新求值以确保正确。关于这一点的一个示例是直接处理 `utmpx`(4) 文件的应用程序。要在 64 位 应用程序环境中正确操作，**请勿**尝试直接访问这些文件，而应当使用 `getutxent`(3C) 及相关的函数系列。

附录 A中包括了更改的派生类型的列表。

检查更改的副作用

需要注意的一个问题是，一个区域中的类型更改可能会导致另一个区域中进行意外的 64 位转换。例如，如果某个函数以前返回类型 `int` 而现在返回 `ssize_t`，则需要检查所有调用方。

检查直接使用 `long` 类型是否仍有意义

由于类型 `long` 在 ILP32 模型中为 32 位，在 LP64 模型中为 64 位，因此可能会出现以前定义为 `long` 类型的数据既不恰当又不必要的情况。在这种情况下，请尽可能使用可移植性更强的派生类型。

与此相关的是，由于上述原因，许多派生类型在 LP64 数据类型模型中可能已更改。例如，`pid_t` 在 32 位环境中仍为 `long` 类型，但是在 64 位环境中，`pid_t` 则为 `int` 类型。有关针对 LP64 编译环境修改的派生类型的列表，请参见附录 A。

对显式 32 位与 64 位原型使用 `#ifdef`

在某些情况下，一个接口存在特定的 32 位和 64 位版本是不可避免的。可以通过在头文件中使用 `_LP64` 或 `_ILP32` 功能测试宏来区分这些版本。同样，要在 32 位和 64 位环境中运行的代码也需要利用相应的 `#ifdefs`，具体取决于编译模式。

算法更改

在确认代码对 64 位环境是安全的之后，请再次检查代码，以验证算法和数据结构是否仍有意义。由于数据类型变大，因此数据结构可能会使用更多空间。代码的性能也可能变化。考虑到以上几点，您可能需要相应地修改代码。

入门清单

假设需要将代码转换为 64 位，以下清单可能会有所帮助：

- 阅读整个文档，重点阅读第 32 页中的 “转换为 LP64 的指导原则” 。
- 检查所有数据结构和接口，验证它们在 64 位环境中仍然有效。
- 在代码中包含 `<sys/types.h>`，以获取 `_ILP32` 或 `_LP64` 定义以及多种基本派生类型。
- 将函数原型以及具有非局部作用域的外部声明移到头文件中，并将这些头文件包含在代码中。
- 使用 `-errchk=longptr64` 选项运行 `lint(1)`，并单独检查每个警告。请注意，并非所有警告都要求对代码进行更改。根据所进行的更改，可能还需要在 32 位和 64 位模式下再次运行 `lint(1)`。
- 除非提供的应用程序仅为 64 位，否则请将代码编译为 32 位和 64 位两种形式。
- 测试应用程序，方法是在 32 位操作系统上执行 32 位版本，在 64 位操作系统上执行 64 位版本。也可以在 64 位操作系统上测试 32 位版本，但这不是必需的。

程序样例

以下程序样例 `foo.c` 直接对比说明了 LP64 数据模型与 ILP32 数据模型的不同结果。同一个程序既可以编译为 32 位程序，也可以编译为 64 位程序。

```
#include <stdio.h>

int

main(int argc, char *argv[])

{

        (void) printf("char is \t\t%lu bytes\n", sizeof (char));

        (void) printf("short is \t%lu bytes\n", sizeof (short));

        (void) printf("int is \t\t%lu bytes\n", sizeof (int));

        (void) printf("long is \t\t%lu bytes\n", sizeof (long));

        (void) printf("long long is \t\t%lu bytes\n", sizeof (long long));

        (void) printf("pointer is \t%lu bytes\n", sizeof (void *));
```

```
        return (0);
}
```

32 位编译的结果是：

```
% cc -O -o foo32 foo.c
% foo32
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          4 bytes
long long is     8 bytes
pointer is       4 bytes
```

64 位编译的结果是：

```
% cc -xarch=generic64 -O -o foo64 foo.c
% foo64
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          8 bytes
long long is     8 bytes
pointer is       8 bytes
```

注 – 缺省编译环境旨在最大化可移植性，即创建 32 位应用程序。

◆ ◆ ◆ 第 5 章

5 开发环境

本章介绍 64 位应用程序开发环境以及生成环境，包括头文件和库问题、编译器选项、链接和调试工具。本章中的信息还将指导如何处理打包问题。

不过，在开始操作之前，必须首先确定所安装的操作系统是 32 位还是 64 位版本。如果不确定所安装的版本，则假设运行的是 64 位版本的操作系统。要进行确认，可以使用第 3 章中介绍的 `isainfo`(1) 命令。即使您使用的是 32 位操作环境，仍可以生成自己的 64 位应用程序，但前提是系统中包含系统头文件和 64 位库。

生成环境

生成环境包括系统头文件、编译系统和库，这些内容将在以下各节中进行介绍。

头文件

一组系统头文件可同时支持 32 位和 64 位编译环境。您无需为 64 位编译环境指定其他头文件路径。

应了解头文件 `<sys/isa_defs.h>` 中的各种定义，以便更好地了解为支持 64 位环境而对头文件进行的更改。该头文件包含一组已知的 `#defines`，并会为每个指令集体系结构设置这些指令。如果包含 `<sys/types.h>`，则会自动包含 `<sys/isa_defs.h>`。

下表中的符号由编译环境定义：

符号	说明
`__sparc`	表示 SPARC 系列处理器体系结构，其中包括 SPARC V7、SPARC V8 和 SPARC V9 体系结构。符号 `sparc` 是 `__sparc` 的前身。

符号	说明
`__sparcv8`	表示 SPARC Architecture Manual 版本 8 中所定义的 32 位 SPARC V8 体系结构。
`__sparcv9`	表示 *SPARC Architecture Manual* 版本 9 中所定义的 64 位 SPARC V9 体系结构。
`__x86`	表示 x86 系列处理器体系结构，其中包括 386、486、Pentium、IA-32、AMD64 和 EM64T 处理器。
`__i386`	表示 32 位 i386 体系结构。
`__amd64`	表示 64 位 amd64 体系结构。

注 – `__i386` 和 `__amd64` 互斥。符号 `__sparcv8` 和 `__sparcv9` 互斥并且仅当定义符号 `__sparc` 时才相关。

以下符号是从上面所定义的符号的一些组合派生而来的：

符号	说明
`_ILP32`	类型 `int`、`long` 和指针的长度均为 32 位的数据模型。
`_LP64`	类型 `long` 和指针的长度均为 64 位的数据模型。

注 – 符号 `_ILP32` 和 `_LP64` 互斥。

如果无法编写完全可移植的代码，并且需要特定的 32 位与 64 位代码，请使用 `_ILP32` 或 `_LP64` 在代码中设置条件。这会使编译环境独立于计算机，并且最大程度地增强应用程序对所有 64 位平台的可移植性。

编译器环境

为了同时支持 32 位应用程序和 64 位应用程序的创建，Sun Studio C、C++ 和 Fortran 编译环境已进行了增强。Sun Studio 中 10.0 发行版的 C 编译器可提供 64 位编译支持。

本机编译和交叉编译模式均受支持。缺省编译环境仍然可以生成 32 位应用程序。尽管这两种模式均受支持，但是它们仍特定于体系结构。使用 Sun 编译器无法在 x86 计算机上创建 SPARC 对象，也无法在 SPARC 计算机上创建 x86 对象。如果缺少体系结构规范或编译模式，则缺省情况下会定义相应的 `__sparcv8` 或 `__i386` 符号，在此过程中还会定义 `_ILP32`。这会最大程度地提高与现有应用程序和硬件库的互操作性。

从 Sun Studio 8 发行版开始，可使用 `cc`(1) `-xarch=generic64` 标志来启用 64 位编译环境。

这会在 ELF64 对象中生成 LP64 代码。ELF64 是支持 64 位处理器和体系结构的 64 位目标文件格式，它与在缺省 32 位模式下进行编译时生成的 ELF32 目标文件相对。

`-xarch=generic64` 标志用来在 32 位或 64 位系统中生成 64 位代码。使用 32 位编译器可以在 32 位系统中生成 64 位对象；但是，不能在 32 位系统中运行 64 位对象。无需为 64 位库指定库路径。如果使用 `-l` 或 `-L` 选项来指定其他库或库路径，并且该路径仅指向 32 位库，则链接程序会检测到这一情况，同时将失败并显示错误。

有关编译器选项的说明，请参见《*Sun Studio 10 C* **用户指南**》。

32 位和 64 位库

Solaris 操作环境为 32 位和 64 位编译环境均提供了共享库。

32 位应用程序必须与 32 位库链接，64 位应用程序必须与 64 位库链接。不能使用 64 位库来创建或执行 32 位应用程序。32 位库仍位于 `/usr/lib` 和 `/usr/ccs/lib` 中。64 位库位于相应的 `lib` 目录的子目录中。由于 32 位库的位置没有变化，因此在早期发行版中生成的 32 位应用程序保持二进制兼容。可移植 makefile 应当使用 `64` 位符号链接来引用任何库目录。

为了生成 64 位应用程序，需要使用 64 位库。可以在本机或交叉编译模式下生成，因为 64 位库对于 32 位和 64 位环境均可用。编译器和其他各种工具（例如 `ld`、`ar` 和 `as`）是能够在 32 位或 64 位系统中生成 64 位程序的 32 位程序。当然，在运行 32 位操作系统的系统中生成的 64 位程序不能在该 32 位环境中执行。

链接目标文件

链接程序仍然是 32 位应用程序，但是该应用程序对于大多数用户应保持透明，因为它通常是由编译器驱动程序（例如 `cc(1)`）间接调用的。如果向链接程序提供 ELF32 目标文件的集合作为输入，则它会创建 ELF32 输出文件；同样，如果向链接程序提供 ELF64 目标文件的集合作为输入，则它会创建 ELF64 输出文件。链接程序不允许尝试混合使用 ELF32 和 ELF64 输入文件。

LD_LIBRARY_PATH 环境变量

SPARC。32 位应用程序和 64 位应用程序的动态链接程序分别是 `/usr/lib/ld.so.1` 和 `/usr/lib/sparcv9/ld.so.1`。

x86。对于 AMD64 体系结构，32 位应用程序和 64 位应用程序的动态链接程序分别是 `/usr/lib/ld.so.1` 和 `/usr/lib/amd64/ld.so.1`。

在运行时，这两个动态链接程序会搜索 `LD_LIBRARY_PATH` 环境变量所指定的用冒号分隔的**相同**目录列表。但是，32 位动态链接程序仅绑定到 32 位库，而 64 位动态链接程序则仅绑定到 64 位库。因此，如有必要，可通过 `LD_LIBRARY_PATH` 来指定同时包含 32 位和 64 位库的目录。

使用 LD_LIBRARY_PATH_64 环境变量可以完全覆盖 64 位动态链接程序的搜索路径。

$ORIGIN 关键字

分发和管理应用程序的一种常用方法就是将相关的应用程序和库放入一个简单的目录分层结构中。通常，应用程序使用的库驻留在 lib 子目录中，而应用程序本身则驻留在基目录的 bin 子目录中。该基目录随后可以使用 NFS™（Sun 的分布式计算文件系统）导出并挂载到客户机上。在某些环境中，自动挂载程序和名称服务可用来分发应用程序，并可确保应用程序分层结构的文件系统名称空间在所有客户机上都相同。在此类环境中，可以在生成应用程序时，使用链接程序的 -R 标志来指定在运行时应当在其中搜索共享库的目录的绝对路径名。

但是在其他环境中，文件系统名称空间的控制并不是很好，并且开发者已转向使用调试工具（LD_LIBRARY_PATH 环境变量）来在包装脚本中指定库搜索路径。这是不必要的，因为可以在路径名（在链接程序的 -R 选项中指定）中使用 $ORIGIN 关键字。$ORIGIN 关键字在运行时会扩展为可执行文件本身所在目录的名称。实际上，这意味着可以使用相对于 $ORIGIN 的路径名来指定库目录的路径名。这允许在完全未设置 LD_LIBRARY_PATH 的情况下重新定位应用程序的基目录。

此功能对于 32 位和 64 位应用程序均可用，并且需要考虑何时创建新的应用程序，以减少正确配置 LD_LIBRARY_PATH 时用户或脚本的依赖项。

有关更多详细信息，请参见《链接程序和库指南》。

对 32 位和 64 位应用程序进行打包

以下几节将讨论对 32 位和 64 位应用程序进行打包时的注意事项。

库和程序的位置

SPARC。新库和新程序的位置遵循第 47 页中的 “32 位和 64 位库”中介绍的标准约定。32 位库仍然位于相同位置，而 64 位库则会放在常规缺省目录下与体系结构有关的特定目录中。特定于 32 位和 64 位的应用程序的位置对于用户应保持透明。

这意味着 32 位库应放在相同的库目录中，64 位库应放在相应的 lib 目录下的 sparcv9 子目录中。

如果程序要求使用特定于 32 位或 64 位环境的版本，则情况会稍有不同，此时，应将这些程序放在其通常所在目录的相应 sparcv7 或 sparcv9 子目录中。

64 位库应放在相应的 lib 目录下的 amd64 子目录中。

如果程序要求使用特定于 32 位或 64 位环境的版本，则应将这些程序放在其通常所在目录的相应 i86 或 amd64 子目录中。

请参见第 49 页中的 “应用程序命名约定” 。

打包原则

打包选项包括为 32 位和 64 位应用程序创建特定的软件包或者将 32 位和 64 位版本合并到一个软件包中。如果创建了单个软件包，则应当针对软件包的内容使用子目录命名约定，如本章中所述。

应用程序命名约定

如第 48 页中的 “库和程序的位置” 中所述，32 位和 64 位应用程序可以放在平台特定的相应子目录中，而不是为同一个应用程序的 32 位和 64 位版本指定特定的名称，如 `foo32` 和 `foo64`。然后，可以使用包装（在下一节中进行介绍）来运行正确版本的应用程序。这样做的一个优点是用户无需了解具体版本（32 位或 64 位），因为系统会根据平台自动执行正确的版本。

Shell 脚本包装

在需要使用特定于 32 位和 64 位的应用程序版本的情况下，可以借助 shell 脚本包装使版本对于用户保持透明。如果 Solaris 操作环境中有许多工具需要使用 32 位和 64 位版本，则可以这样做。包装可以使用 `isalist` 命令来确定可在特定硬件平台上执行的本机指令集，并基于指令集来运行适当版本的工具。

以下是本机指令集包装的一个示例：

```
#! /bin/sh



CMD=‘basename $0‘

DIR=‘dirname $0‘

EXEC=

for isa in ‘/usr/bin/isalist‘; do

    if [-x ${DIR}/${isa}/${CMD}]; then

        EXEC=${DIR}/${isa}/${CMD}

        break
```

```
    fi

done

if [-z "${EXEC}"]; then

        echo 1>&2 "$0: no executable for this architecture"

        exit 1



fi

exec ${EXEC} "${@}"
```

此示例存在一个问题，即它要求 `$0` 参数是其可执行文件的全路径名。为此，创建了一个常规包装 `isaexec()`，以便解决特定于 32 位和 64 位应用程序的问题。下面将说明该包装。

/usr/lib/isaexec 二进制文件

`isaexec`(3C) 是一个 32 位可执行二进制文件，它执行上面刚介绍的 shell 脚本包装中概述的包装函数，但是具有准确完整的参数列表。该可执行文件的全路径名是 `/usr/lib/isaexec`，但是在执行时并不使用该名称，而是将该名称复制或链接（硬链接而不是软链接）到存在多个版本的程序的主名称，这些版本是通过 `isalist`(1) 进行选择的。

例如，在 SPARC 环境中，`truss`(1) 命令作为以下三个可执行文件存在：

```
/usr/bin/truss
/usr/bin/sparcv7/truss
/usr/bin/sparcv9/truss
```

`sparcv7` 和 `sparcv9` 子目录中的可执行文件分别是实际的 32 位和 64 位 `truss`(1) 可执行文件。包装文件 `/usr/bin/truss` 是指向 `/usr/lib/isaexec` 的硬链接。

在 x86 环境中，`truss`(1) 命令作为以下三个可执行文件存在：

```
/usr/bin/truss
/usr/bin/i86/truss
/usr/bin/amd64/truss
```

`isaexec`(3C) 包装使用 `getexecname`(3C) 来确定它自己的经过完全解析的无符号链接路径名（与其 `argv[0]` 参数无关），`sysinfo(SI_ISALIST, ...)` 获取 `isalist`(1)，并针对

第一个可执行文件执行 exec(2)，第一个可执行文件的名称可在为其所在的目录生成的子目录列表中找到。然后，该包装会按原样传递参数向量和环境向量。这样，传递给最后一个程序映像的 argv[0] 将显示为所指定的第一个参数，而不是转换成经修改以包含子目录名称的全路径名。

注 – 因为可能存在包装，所以在将可执行文件移到其他位置时，需要格外小心。您可能会移动包装而不是实际程序。

isaexec(3c) 接口

许多应用程序已经使用启动包装程序来设置环境变量、清除临时文件、启动守护进程等。 libc(3LIB) 中的 isaexec(3C) 接口允许直接从自定义包装程序调用以上基于 shell 的包装示例中所使用的算法。

调试 64 位应用程序

为了处理 64 位应用程序，所有 Solaris 调试工具均已进行了更新。这些工具包括 truss(1) 命令、/proc 工具 (proc(1)) 和 mdb。

dbx 调试器作为 Sun Studio 工具套件的一部分提供，可用来调试 64 位应用程序。其余的工具是 Solaris 发行版附带的。

所有这些工具的选项都保持不变。mdb 中提供了许多增强功能，用来调试 64 位程序。正如所预期的那样，使用 "*" 取消引用指针时，对于 64 位程序将取消引用 8 个字节，对于 32 位程序取消引用 4 个字节。 此外，还提供了以下修饰符：

```
Additional ?, /, = modifiers:


g        (8) Display 8 bytes in unsigned octal

G        (8) Display 8 bytes in signed octal

e        (8) Display 8 bytes in signed decimal

E        (8) Display 8 bytes in unsigned decimal

J        (8) Display 8 bytes in hexadecimal

K        (n) Print pointer or long in hexadecimal

          Display 4 bytes for 32-bit programs
```

```
          and 8 bytes for 64-bit programs.

y        (8) Print 8 bytes in date format


Additional ? and / modifiers:


M <value> <mask>  Apply <mask> and compare for 8-byte value;

          move '.' to matching location.

Z        (8) write 8 bytes
```

◆ ◆ ◆ 第 6 章

6 高级主题

本章所介绍的各种编程主题面向需要详细了解 64 位 Solaris 操作环境的系统程序员。

尽管 64 位环境的几个新增特征是 64 位环境所特有的，但是大多数新增特征都是对普通 32 位接口的扩展。

SPARC V9 ABI 特征

64 位应用程序使用可执行和链接格式 (Executable and Linking Format, ELF64) 进行描述，使用该格式可以全面地描述大型应用程序和较大地址空间。

SPARC V9。《*SPARC Compliance Definition Version 2.4*》中包含 SPARC V9 ABI 的详细信息。此文档描述了 32 位 SPARC V8 ABI 和 64 位 SPARC V9 ABI，可以从 SPARC International 公司的网站 www.sparc.com 上获取。

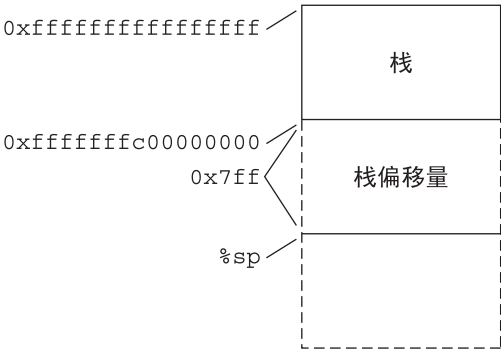
以下列出了 SPARC V9 ABI 的特征：

- SPARC V9 ABI 允许充分利用所有 64 位 SPARC 指令和数据宽度为 64 位的寄存器。许多新的相关指令都是对现有 V8 指令集的扩展。请参见《*The SPARC Architecture Manual, Version 9*》。
- 基本调用约定相同。调用方的前六个参数放在外部寄存器 `%o0`-`%o5` 中。SPARC V9 ABI 对于较大的寄存器文件仍会使用寄存器窗口，以便使函数调用操作“降低成本”。结果将返回到 `%o0` 中。由于所有的寄存器现在都被视为 64 位，因此 64 位值现在可以通过单寄存器而不是寄存器对传送。
- 栈的布局不同。除了基本单元大小从 32 位增加到 64 位外，各种隐藏的参数术语均已删除。返回地址仍是 `%o7 + 8`。
- `%o6` 仍然称作**栈指针**寄存器 `%sp`，`%i6` 称作**帧指针**寄存器 `%fp`。但是，`%sp` 和 `%fp` 寄存器距离栈的实际内存位置的偏移量（称为**栈偏移量**）是常量。栈偏移量的大小为 2047 个字节。

- 指令大小仍为 32 位。因此，生成地址常量需要更多指令。该调用指令无法再使用该调用指令在地址空间中的任何位置建立分支，因为它只能达到离 %pc (2 GB 的位置。
- 现在，整数乘除函数可以完全在硬件中实现。
- 结构的传递和返回以不同的方式实现。小型数据结构和某些浮点参数现在直接在寄存器中传递。
- 通过用户陷阱，用户陷阱处理程序可处理非特权代码中的某些陷阱（而不是传送信号）。
- 所有数据类型现在都与其长度对齐。
- 许多基本派生类型会更长，因此，许多系统调用接口数据结构现在具有不同的长度。
- 系统中存在两组不同的库：用于 32 位 SPARC 应用程序的库和用于 64 位 SPARC 应用程序的库。

栈偏移量

SPARC V9。对于开发者来说，SPARC V9 ABI 的一个重要特征就是栈偏移量。 对于 64 位 SPARC 程序来说，必须向帧指针和栈指针中都添加大小为 2047 个字节的栈偏移量，才能达到栈帧的实际数据。 请参见下图。

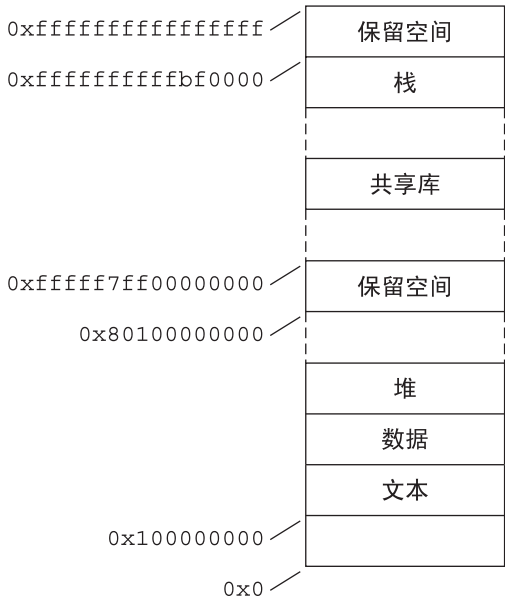


有关栈偏移量的更多信息，请参见 SPARC V9 ABI。

SPARC V9 ABI 的地址空间布局

SPARC V9。对于 64 位应用程序，尽管起始地址和寻址限制大不相同，但地址空间的布局与 32 位应用程序的布局密切相关。与 SPARC V8 一样，SPARC V9 栈从地址空间的顶部开始减小，而堆则从底部开始扩展数据段。

下图说明了为 64 位应用程序提供的缺省地址空间。地址空间中标记为保留空间的区域可能不是由应用程序进行映射。这些限制在将来的系统中可能会有所放松。



上图中的实际地址描述了特定计算机上的特定实现，仅供参考。

SPARC V9 ABI 文本和数据的位置

缺省情况下，64 位程序与起始地址 0x100000000 链接，整个程序（包括其文本、数据、堆、栈和共享库）将超过 4 GB。这有助于确保 64 位程序正确无误，方法是使其在截断其任何指针时在较低的 4 GB 地址空间中出错。

尽管 64 位程序会链接到 4 GB 以上的地址空间，但是仍可以通过使用链接程序映射文件以及编译器或链接程序的 `-M` 选项，将其链接到 4 GB 以下的地址空间。`/usr/lib/ld/sparcv9/map.below4G` 中提供了一个用来将 64 位 SPARC 程序链接到 4 GB 以下地址空间的链接程序映射文件。

有关更多信息，请参见 `ld`(1) 链接程序手册页。

SPARC V9 ABI 的代码模型

SPARC V9。编译器针对不同目的提供了不同的代码模型，旨在提高性能并减少 64 位 SPARC 程序中代码的大小。代码模型由以下因素确定：

- 可放置性（绝对独立代码与位置无关代码）

- 代码大小（< 2 GB）
- 位置（地址空间中的低、中或任意位置）
- 外部对象引用模型（小或大）

下表介绍了可用于 64 位 SPARC 程序中的不同代码模型。

表 6–1 代码模型说明：SPARC V9

代码模型	可放置性	代码大小	位置	外部对象引用模型
`abs32`	绝对	< 2 GB	低（低 32 位地址空间）	无
`abs44`	绝对	< 2 GB	中（低 44 位地址空间）	无
`abs64`	绝对	< 2 GB	任意位置	无
`pic`	PIC（位置无关代码）	< 2 GB	任意位置	小（<= 1024 个外部对象）
`PIC`	PIC	< 2 GB	任意位置	大（<= 2**29 个外部对象）

在某些情况下，可以使用较小的代码模型实现较短的指令序列。在绝对代码中执行静态数据引用所需的指令数在不同的代码模型中各不相同：在 `abs32` 代码模型中最少，在 `abs64` 代码模型中最多，在 `abs44` 代码模型中居于两者之间。同样，在执行静态数据引用时，`pic` 代码模型使用的指令比 `PIC` 代码模型使用的要少。因此，代码模型越小，代码块越小，对于无需利用较大代码模型的更完整功能的程序，还可能会提高其性能。

要指定要使用的代码模型，应使用 `-xcode=<model>` 编译器选项。目前，对于 64 位对象，编译器在缺省情况下使用 `abs64` 模型。通过使用 `abs44` 代码模型可以优化代码，此时将使用较少的指令，并且仍能涵盖当前的 UltraSPARC 平台所支持的 44 位地址空间。

有关代码模型的更多信息，请参见 SPARC V9 ABI 和编译器文档。

注 – 对于使用 `abs32` 代码模型编译的程序，必须使用 `-M /usr/lib/ld/sparcv9/map.below4G` 选项将其链接到 4 GB 以下的地址空间。

AMD64 ABI 特征

64 位应用程序使用可执行和链接格式 (Executable and Linking Format, ELF64) 进行描述，使用该格式可以全面地描述大型应用程序和较大地址空间。

以下列出了 AMD ABI 的特征：

- AMD ABI 允许充分利用所有的 64 位指令和 64 位寄存器。许多新指令都是对现有 i386 指令集的直接扩展。目前，共有十六个通用寄存器：

七个通用寄存器（%rdi、%rsi、%rdx、%rcx、%r8、%r9 和 %rax）在函数调用序列中具有明确定义的角色，该序列现在用于在寄存器中传递参数。
两个寄存器（%rsp 和 %rbp）用于管理栈。
两个寄存器（%r10 和 %r11）是临时寄存器。
五个寄存器（%r12、%r13、%r14、%r15 和 %rbx）由被调用方保存。

- 对于 AMD ABI，基本函数调用的约定不同。参数会放在寄存器中。对于简单的整数参数，前几个参数依次放在 %rdi、%rsi、%rdx、%rcx、%r8 和 %r9 寄存器中。
- 对于 AMD，栈的布局稍有不同。具体来说，栈在紧靠调用指令的前面始终以 16 个字节为边界对齐。
- 指令大小仍为 32 位。因此，生成地址常量需要更多指令。由于调用指令只能到达从 %rip 加/减 2 GB 的范围，因此无法再将其用于在地址空间中的任何位置建立分支。
- 现在，整数乘除函数可以完全在硬件中实现。
- 结构的传递和返回以不同的方式实现。小型数据结构和某些浮点参数现在直接在寄存器中传递。
- 使用新的 PC 相关寻址模式，可以生成更高效的位置无关代码。
- 所有数据类型现在都与其长度对齐。
- 许多基本派生类型会更长，因此，许多系统调用接口数据结构现在具有不同的长度。
- 系统中存在两组不同的库：一种用于 32 位 i386 应用程序的库，另一种用于 64 位 amd64 应用程序的库。
- AMD ABI 极大地增强了浮点功能。

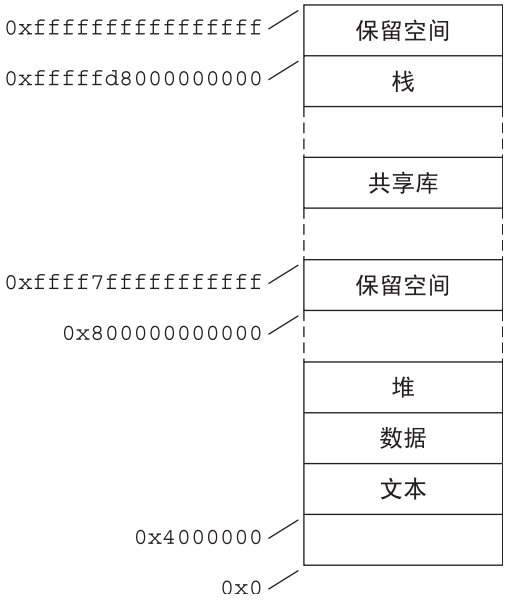
通过 64 位 ABI，可以使用所有使用 x87 浮点寄存器（%fpr0 至 %fpr7，%mm0 至 %mm7）的 x87 和 MMX 指令。
此外，还可以使用那些使用 128 位 XMM 寄存器（%xmm0 至 %xmm15）完整 SSE 和 SSE2 指令集。

请参见 amd64 psABI 草案文档《System V Application Binary Interface, AMD64 Architecture Processor Supplement》（草案版本 0.92）。

amd64 应用程序的地址空间布局

对于 64 位应用程序，尽管起始地址和寻址限制大不相同，但地址空间的布局与 32 位应用程序的布局密切相关。与 SPARC V9 一样，amd64 栈从地址空间的顶部开始减小，而堆则从底部开始扩展数据段。

下图说明了为 64 位应用程序提供的缺省地址空间。地址空间中标记为保留空间的区域可能不是由应用程序进行映射。这些限制在将来的系统中可能会有所放松。



上图中的实际地址描述了特定计算机上的特定实现，仅供参考。

对齐问题

还有一个问题也与数据结构中 32 位 `long long` 元素的对齐相关；i386 应用程序仅以 32 位为边界将 `long long` 元素对齐，而 amd64 ABI 则限制 `long long` 元素以 64 位为边界，这有可能会在数据结构中产生更宽的空白区域。这与 SPARC 中有所不同，在 SPARC 中 32 位或 64 位 `long long` 项均以 64 位为边界对齐。

下表说明了指定体系结构中的数据类型对齐情况。

表 6–2 数据类型对齐

体系结构	long long	double	long double
i386	4	4	4
amd64	8	8	16
sparcv8	8	8	8
sparcv9	8	8	16

针对 SPARC 系统，尽管看起来已经是干净的 64 位代码，但是如果在 32 位和 64 位编程环境之间复制数据结构，则这两个环境中不同的对齐情况可能会产生问题。这些编程环境包括设备驱动程序的 `ioctl` 例程、`doors` 例程或其他 IPC 机制。通过对这些接口仔细进行编码，并且谨慎地使用 `#pragma pack` 或 `_Pack` 指令，可避免对齐问题。

进程间通信

以下进程间通信 (interprocess communication, IPC) 元语仍适用于 64 位和 32 位进程之间的通信。

- System V IPC 元语，如 `shmop`(2)、`semop`(2) 和 `msgsnd`(2)
- `mmap`(2)（针对共享文件调用）
- `pipe`(2)（在进程之间使用）
- `door_call`(3DOOR)（在进程之间使用）
- `rpc`(3NSL)（在使用 `xdr`(3NSL) 中所述的外部数据表示形式的同一台或不同计算机上的进程之间使用）

尽管所有这些元语都允许在 32 位和 64 位进程之间进行通信，但是可能需要明确执行一些步骤来确保在进程间交换的数据可以由所有这些元语正确解释。例如，两个进程共享由 C 数据结构所描述的数据，并且该数据结构中包含类型为 `long` 的变量，如果不了解 32 位进程将该变量视为 4 字节值，而 64 位进程将该变量视为 8 字节值，则在这两个进程之间交换的数据将无法正确解释。

处理此差异的一种方法是确保数据在这两个进程中具有完全相同的长度和含义。构建使用定宽类型（如 `int32_t` 和 `int64_t`）的数据结构。处理对齐问题时仍需要小心。对于共享数据结构，可能需要对其进行填充，或者使用编译器指令（如 `#pragma pack` 或 `_Pack`）对其重新打包。请参见第 58 页中的 “对齐问题” 。

`<sys/types32.h>` 中提供了用于镜像系统派生类型的派生类型系列。这些类型的符号和长度与 32 位系统的基本类型相同，但是按照特定方式进行定义，以便长度在 ILP32 和 LP64 编译环境中保持不变。

在 32 位和 64 位进程之间共享指针极为困难。显然，指针长度不同，但更重要的是，尽管在现有的 C 用法中有一个 64 位整数值 (`long long`)，但是 64 位指针在 32 位环境中没有等效指针。为了使 64 位进程可以与 32 位进程共享数据，必须使 32 位进程一次最多只能查看 4 GB 共享数据。

XDR 例程 `xdr_long`(3NSL) 可能是一个问题，但是，在线上仍然会将其作为 32 位值进行处理，以便与现有协议兼容。如果要求该例程的 64 位版本对不适合 32 位值的 `long` 值进行编码，则编码操作将失败。

ELF 和系统生成工具

64 位二进制对象存储在 ELF64 格式的文件中，此格式与 ELF32 格式非常相似，不同的是大多数字段都已增大，可以适应所有的 64 位应用程序。ELF64 文件可以使用 elf(3ELF) API（例如 elf_getarhdr(3ELF)）进行读取。

32 位和 64 位版本的 ELF 库 elf(3ELF) 可同时支持 ELF32 格式和 ELF64 格式及其对应的 API。这允许应用程序从 32 位或 64 位系统（尽管 64 位系统仍需执行 64 位）生成、读取或修改这两种文件格式。

此外，Solaris 还提供了一组 GELF（常规 ELF）接口，以便允许程序员使用一个单独的公用 API 来处理这两种格式。请参见elf(3ELF)。

所有的系统 ELF 实用程序（包括 ar(1)、nm(1)、ld(1) 和 dump(1)）均已进行更新，可以接受这两种 ELF 格式。

/proc 接口

/proc 接口可供 32 位和 64 位应用程序使用。32 位应用程序可以检查和控制其他 32 位应用程序的状态。因此，可以使用现有的 32 位调试器来调试 32 位应用程序。

64 位应用程序可以检查和控制 32 位或 64 位应用程序。但是，32 位应用程序无法控制 64 位应用程序，因为 32 位 API 不允许描述 64 位进程的完整状态。因此，必须使用 64 位调试器才能调试 64 位应用程序。

sysinfo(2) 的扩展

使用 Solaris S10 操作环境中新的 sysinfo(2) 子代码，应用程序可确定有关可用指令集体系结构的更多信息。

```
SI_ARCHITECTURE_32
SI_ARCHITECTURE_64
SI_ARCHITECTURE_K
SI_ARCHITECTURE_NATIVE
```

例如，对于系统中可用的 64 位 ABI 的名称（如果有），可以通过使用 SI_ARCHITECTURE_64 子代码来对其进行使用。有关详细信息，请参见 sysinfo(2)。

libkvm 和 /dev/ksyms

64 位版本的 Solaris 系统是使用 64 位内核实现的。直接检查或修改内核中内容的应用程序必须转换为 64 位应用程序，并且必须与 64 位版本的库链接。

执行此转换和清理工作之前，应当考虑为什么应用程序需要首先直接查看内核数据结构。可能是由于在此过程中首先会导入或创建程序，因此 Solaris 平台上将启用用来提取系统调用所需数据的其他接口。请参见 sysinfo(2)、kstat(3KSTAT)、sysconf(3C) 和 proc(4)，这些接口是最常见的替换 API。如果可以使用这些接口来替代 kvm_open(3KVM)，请使用它们并使应用程序保持 32 位以实现最大可移植性。另一个益处是其中的大多数 API 可能更快，并且可能不要求访问内核内存所需的相同安全权限。

如果尝试针对 64 位内核或崩溃转储使用 kvm_open(3KVM)，则 32 位版本的 libkvm 会返回失败信息。同样，如果尝试针对 32 位内核崩溃转储使用 kvm_open(3KVM)，则 64 位版本的 libkvm 也会返回失败信息。

因为内核是一个 64 位程序，所以用来打开 /dev/ksyms 以直接检查内核符号表的应用程序需要进行增强，以便可以识别 ELF64 格式。

无法明确判断 kvm_read() 或 kvm_write() 的地址参数应该是内核地址还是用户地址这一问题对于 64 位应用程序和内核更加严重。所有使用 libkvm 并且还使用 kvm_read() 和 kvm_write() 的应用程序应当转换为使用相应的 kvm_read()、kvm_write()、kvm_uread() 和 kvm_uwrite() 例程。（这些例程最初是在 Solaris 2.5 中提供的。）

尽管直接读取 /dev/kmem 或 /dev/mem 的应用程序尝试解释从这些设备中读取的数据可能会出错，但这些应用程序仍可以运行；32 位和 64 位内核之间的数据结构偏移量和长度几乎肯定是不相同的。

libkstat 内核统计信息

许多内核统计信息的大小与内核是 64 位还是 32 位程序完全无关。通过指定的 kstats（请参见 kstat(3KSTAT)）导出的数据类型是自述性类型，可用来导出带符号或无符号并且具有相应标记的 32 位或 64 位计数器数据。因此，使用 libkstat 的应用程序**无需**转换为 64 位应用程序即可成功处理 64 位内核。

注 – 如果要修改的设备驱动程序用来创建和维护指定的 kstats，则应当尝试使用定宽统计信息类型，使所导出的统计信息的大小在 32 位和 64 位内核之间保持不变。

---

stdio 的更改

在 64 位环境中，已经对 stdio 功能进行了扩展，允许同时打开 256 个以上的流。32 位 stdio 功能仍具有最多只能同时打开 256 个流这一限制。

64 位应用程序不应当依赖具有对 FILE 数据结构成员的访问权限。如果尝试直接访问特定于实现的专用结构成员，则可能会导致编译错误。现有的 32 位应用程序不会受到此更改的影响，但是，在任何代码中都不应当再直接使用这些结构成员。

FILE 结构有很长的历史，只有少数应用程序曾经查看过数据结构内部以收集有关流状态的其他信息。由于 64 位版本的 FILE 结构现在是不透明的，因此，32 位 libc 和 64 位 libc 中均已添加了一系列新例程，以允许在不依赖内部实现的情况下检查同一个状态。例如，请参见 __fbufsize(3C)。

性能问题

以下几节将讨论 64 位性能的优缺点。

64 位应用程序的优点

- 针对 64 位值更高效地执行算术和逻辑运算。
- 运算过程使用全寄存器集宽度、全寄存器集和新指令。
- 64 位值的参数传递效率更高。
- 小型数据结构和浮点值的参数传递效率更高。
- 提供了额外的整数寄存器和浮点寄存器。
- 对于 amd64，提供了 PC 相关寻址模式，从而可提高位置无关代码的执行效率。

64 位应用程序的缺点

- 64 位应用程序需要更多的栈空间才能容纳更大的寄存器。
- 由于使用了更大的指针，因此应用程序需要更大的高速缓存。
- 64 位应用程序不能在 32 位平台上运行。

系统调用问题

以下几节将讨论系统调用问题。

EOVERFLOW 的含义

每次用来从内核向外传递信息的数据结构中的一个或多个字段太小而无法容纳该值时，系统调用中便会返回 EOVERFLOW 返回值。

现在，许多 32 位系统调用在遇到 64 位内核中的大对象时都会返回 EOVERFLOW。在处理大文件时会出现上述情况，由于 daddr_t、dev_t、time_t 及其派生类型 struct timeval 和 timespec_t 现在包含 64 位值，因此这可能意味着 32 位应用程序会遇到更多的 EOVERFLOW 返回值。

谨慎使用 ioctl()

过去，指定的一些 ioctl(2) 调用非常不当。遗憾的是，ioctl() 完全不执行编译时类型检查，因此，可能难以找到错误的来源。

请考虑使用两个 ioctl() 调用：一个用来处理 32 位值 (IOP32) 的指针，另一个用来处理长值 (IOPLONG) 的指针。

以下代码样例作为 32 位应用程序的一部分来运行：

```
int a, d;

long b;

...

if (ioctl(d, IOP32, &b) == -1)

        return (errno);

if (ioctl(d, IOPLONG, &a) == -1)

        return (errno);
```

编译此代码段并将其作为 32 位应用程序的一部分运行时，这两个 ioctl(2) 调用均可正常工作。

编译此代码段并将其作为 64 位应用程序的一部分运行时，这两个 ioctl() 调用也将成功返回。但是，这两个 ioctl() 均无法正常工作。第一个 ioctl() 传递的容器太大，并且在大端字节序实现中，内核将从 64 位字的错误部分向外复制数据或向其中复制数据。即使在小端字节序实现中，该容器也可能在高 32 位中包含栈垃圾。第二个 ioctl() 会从字中向外复制或向其中复制过多的数据，从而导致读取错误的值或破坏用户栈上的相邻变量。

◆ ◆ ◆ 附 录 A

派生类型更改

就派生类型及其长度而言，缺省的 32 位编译环境与早期发行版的 Solaris 操作环境相同。在 64 位编译环境中，有必要对派生类型进行一些更改。以下各表中将重点介绍这些经过更改的派生类型。

请注意，尽管 32 位和 64 位编译环境不同，但是它们使用同一组头文件，其相应的定义通过编译选项确定。为了更好地了解可供应用程序开发者使用的选项，首先了解 _ILP32 和 _LP64 功能测试宏会有所帮助。

表 A–1 功能测试宏

功能测试宏	说明
_ILP32	_ILP32 功能测试宏用来指定 ILP32 数据模型，该模型中类型为 int、long 的数据和指针是 32 位的。如果仅使用该宏，则会使与以前 Solaris 实现相同的派生类型和长度可见。此为生成 32 位应用程序时的缺省编译环境，它可确保 C 和 C++ 应用程序保持完全二进制和源代码兼容。
_LP64	_LP64 功能测试宏用来指定 _LP64 数据模型，该模型中类型为 int 的数据是 32 位的，类型为 long 的数据和指针是 64 位的。如果是在 64 位模式下进行编译，则缺省情况下会定义 _LP64。开发者只需确保在源代码中包括 <sys/types.h> 或 <sys/feature_tests.h>，即可使 _LP64 定义可见。

以下几个示例说明如何根据编译环境使用功能测试宏来使正确的定义可见。

示例 A–1 _LP64 中定义的 size_t

```
#if defined(_LP64)

typedef ulong_t size_t;   /* size of something in bytes */

#else

typedef uint_t size_t;    /* (historical version) */
```

示例 A–1 _LP64 中定义的 size_t （续）

```
#endif
```

使用本示例中的定义生成 64 位应用程序时，size_t 是类型为 ulong_t 或 unsigned long 的数据（在 LP64 模型中是 64 位值）。相反，生成 32 位应用程序时，size_t 会定义成类型为 uint_t 或 unsigned int 的数据（在 LP32 或 LP64 模型中是 32 位值）。

示例 A–2 _LP64 中定义的 uid_t

```
#if defined(_LP64)

typedef int     uid_t;          /* UID type             */

#else

typedef long    uid_t;          /* (historical version) */

#endif
```

在以上任一示例中，只要 ILP32 类型表示形式与 LP64 类型表示形式相同，即可获取相同的最终结果。例如，如果在 32 位应用程序环境中将 size_t 更改为 ulong_t，或者将 uid_t 更改为 int，则这些类型仍表示 32 位值。但是，保留以前的类型表示形式可确保在 32 位 C 和 C++ 应用程序中保持一致，并与早期发行版的 Solaris 操作环境保持完全二进制兼容和源代码兼容。

表 A–2 列出了已经更改的派生类型。请注意，在向 Solaris 软件中添加 64 位支持以前，_ILP32 功能测试宏下面列出的类型与 Solaris 2.6 中的类型相匹配。生成 32 位应用程序时，可供开发者使用的派生类型与 _ILP32 列中的类型相匹配。生成 64 位应用程序时，派生类型与 _LP64 列中所列的类型相匹配。除了 wchar_t 和 wint_t 类型在 <wchar.h> 中定义以外，所有这些类型都在 <sys/types.h> 中定义。

检查这些表时，请记住，在 32 位环境中类型为 int、long 的数据和指针是 32 位的；在 64 位环境中，类型为 int 的数据是 32 位的，而类型为 long 的数据和指针则是 64 位的。

表 A–2 经过更改的常规派生类型

派生类型	Solaris 2.6	_ILP32	_LP64
blkcnt_t	longlong_t	longlong_t	long
id_t	long	long	int

表 A–2 经过更改的常规派生类型 （续）

派生类型	Solaris 2.6	_ILP32	_LP64
major_t	ulong_t	ulong_t	uint_t
minor_t	ulong_t	ulong_t	uint_t
mode_t	ulong_t	ulong_t	uint_t
nlink_t	ulong_t	ulong_t	uint_t
paddr_t	ulong_t	ulong_t	未定义
pid_t	long	long	int
ptrdiff_t	int	int	long
size_t	uint_t	uint_t	ulong_t
ssize_t	int	int	long
uid_t	long	long	int
wchar_t	long	long	int
wint_t	long	long	int

表 A–3 列出了特定于大文件编译环境的派生类型。这些类型仅在定义了 _LARGEFILE64_SOURCE 功能测试宏的情况下才会进行定义。请注意，以前的 Solaris 2.6 发行版中保留了 ILP32 编译环境。

表 A–3 经过更改的特定于大文件的派生类型

派生类型	Solaris 2.6	_ILP32	_LP64
blkcnt64_t	longlong_t	longlong_t	blkcnt_t
fsblkcnt64_t	u_longlong_t	u_longlong_t	blkcnt_t
fsfilcnt64_t	u_longlong_t	u_longlong_t	fsfilcnt_t
ino64_t	u_longlong_t	u_longlong_t	ino_t
off64_t	longlong_t	longlong_t	off_t

表 A–4 列出了经过更改的与 _FILE_OFFSET_BITS 值相关的派生类型。对于定义了 _LP64 并且 _FILE_OFFSET_BITS==32 的应用程序，不能进行编译。缺省情况下，如果定义了 _LP64，则 _FILE_OFFSET_BITS==64。如果定义了 _ILP32，但是未定义 _FILE_OFFSET_BITS，则缺省情况下 _FILE_OFFSET_BITS==32。这些规则在 <sys/feature_tests.h> 头文件中定义。

表 A–4 经过更改的 FILE_OFFSET_BITS 值的派生类型

派生类型	_ILP32 _FILE_ OFFSET_BITS ==32	_ILP32 _FILE_ OFFSET_BITS ==64	_LP64 _FILE_ OFFSET_BITS==64
ino_t	ulong_t	u_longlong_t	ulong_t
blkcnt_t	long	longlong_t	long
fsblkcnt_t	ulong_t	u_longlong_t	ulong_t
fsfilcnt_t	ulong_t	u_longlong_t	ulong_t
off_t	long	longlong_t	long

◆ ◆ ◆ 附 录 B

常见问题解答 (Frequently Asked Question, FAQ)

如何确定系统运行的是 32 位还是 64 位版本的操作系统？

可以使用 `isainfo -v` 命令来确定操作系统运行的应用程序。该命令会显示一组操作系统支持的应用程序。有关更多信息，请参见 `isainfo(1)` 手册页。

是否可以在 32 位硬件上运行 64 位版本的操作系统？

不可以。不能在 32 位硬件上运行 64 位操作系统。64 位操作系统要求使用 64 位 MMU（内存管理单元）和 CPU 硬件。

如果要在装有 32 位操作系统的系统上运行 32 位应用程序，是否需要对该应用程序进行更改？

不需要。如果应用程序仅在 32 位操作系统上执行，则不需要对其进行任何更改或重新编译。

如果要在装有 64 位操作系统的系统上运行 32 位应用程序，是否需要对该应用程序进行更改？

大多数应用程序都可以保留为 32 位，并且仍可以在运行 64 位操作系统的系统上执行，而无需对代码进行任何更改或重新编译。对于不需要 64 位功能的 32 位应用程序可以保留为 32 位，以便最大程度地提高可移植性。

如果应用程序使用 `libkvm(3LIB)`，则必须将其重新编译为 64 位 才可以在运行 64 位操作系统的系统上执行。如果应用程序使用 `/proc`，则可能需要将其重新编译为 64 位，否则它无法识别 64 位进程。这是因为描述该进程的现有接口和数据结构不够大，无法包含所涉及的 64 位值。

如果要实现 64 位功能，需要调用什么程序？

没有专门用来调用 64 位功能的程序。要在运行 64 位版本操作系统的系统上利用 64 位功能，需要重新生成应用程序。

是否可以在运行 64 位操作系统的系统上生成 32 位应用程序？

可以。本机编译和交叉编译模式均受支持。无论系统运行的是 32 位版本的操作系统还是 64 位版本的操作系统，缺省编译模式均为 32 位。

是否可以在运行 32 位操作系统的系统上生成 64 位应用程序？

可以，但前提是安装了系统头文件和 64 位库。但是，不能在运行 32 位操作系统的系统上运行 64 位应用程序。

生成和链接应用程序时，是否可以结合使用 32 位库和 64 位库？

不可以。32 位应用程序必须与 32 位库链接，64 位应用程序必须与 64 位库链接。如果尝试使用错误的库版本执行生成和链接操作，则会产生错误。

64 位实现中浮点数据类型的长度是多少？

仅有 `long` 和 `pointer` 类型发生了变化。请参见表 4–1。

`time_t` **有何变化？**

`time_t` 类型仍然是类型为 `long` 的值。在 64 位环境中，此类型会增大到 64 位。因此，64 位应用程序将不会出现 2038 年问题。

在运行 64 位 Solaris 操作环境的计算机上，`uname(1)` **的值是多少？**

`uname -p` 命令的输出没有变化。

是否可以创建 64 位 XView 或 OLIT 应用程序？

不可以。这些库对于 32 位环境已经过时，并且不会在 64 位环境中继续使用。

为什么 `/usr/bin/sparcv9/ls` **中存在 64 位版本的** `ls`**？**

在常规操作中，无需使用 64 位版本的 `ls`。但是，由于在 `/tmp` 和 `/proc` 中可能会创建过大的文件系统对象而导致 32 位 `ls` 无法识别，因此，64 位版本的 `ls` 允许用户对这些对象进行检查。

索引