C 用户指南

Sun™ Studio 8

Sun Microsystems, Inc.
www.sun.com

部件号码 817-5796-10
2004 年 4 月，修订版 A

如对本文档有任何意见，请将电子邮件发送到：http://www.sun.com/hwdocs/feedback

请回收

Adobe PostScript

目录

图

表

开始之前必须了解的事项

本手册描述了 Sun™ Studio 8 的 C 编译器。本手册供有 C 语言和 UNIX® 工作经验的应用程序开发者使用。

本手册提供有关许多编程和编译器相关主题的信息，包括：

- 编译器命令选项的参考附录
- 支持的 ISO/IEC 9899:1999 （本手册中称为 C99）功能的描述
- 特定于 C 标准实现的信息，如 pragma 和声明说明符
- `lint` 代码检查程序的描述和参考
- 用于并行化代码的指令
- 用于转换为 ISO 标准代码的指令
- 递增链接程序 `ild` 的描述和参考

还有几个参考资料附录，如 ISO C 数据表示、实现定义的行为、 Sun C (K & R) 与 Sun ISO C 之间的差异、性能调节以及转换要编译的应用程序以适应 64 位环境。

印刷惯例

表 P-1 字样惯例

字样	含义	示例
AaBbCc123	命令、文件和目录的名称；计算机屏幕输出	编辑您的 .login 文件。 使用 ls -a 列出所有文件。 % You have mail.
AaBbCc123	您键入的内容，与计算机屏幕输出对比时	% **su** Password:
AaBbCc123	书名、新字或术语、要强调的字	阅读 《*用户指南*》第 6 章。 这些称为*类*选项。 您*必须*是超级用户才能这样做。
AaBbCc123	命令行占位符文字；将用实际名称或值替代	要删除文件，请键入 **rm** *filename*。

表 P-2 代码惯例

代码符号	含义	表示法	代码示例
[ ]	方括号包含可选的参数。	O[*n*]	-O4, -O
{ }	花括号包含一个必需选项的一组选项。	d{y|n}	-dy
|	“管道”或“条形”符号分隔参数，只能选择其中一个参数。	B{dynamic|static}	-Bstatic
:	冒号类似逗号，有时用来分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
...	省略号表示数列中的省略。	-xinline=*f1*[,...*fn*]	-xinline=alpha,dos

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 的超级用户	#

访问 Sun Studio 软件和手册页

编译器和工具及其手册页未安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问编译器和工具，必须正确设置 PATH 环境变量（请参见第 xxxiii 页上的“访问编译器和工具”）。要访问手册页，必须正确设置 MANPATH 环境变量 （请参见第 xxxiv 页上的“访问手册页”）。

有关 PATH 变量的详细信息，请参见 csh(1)、 sh(1) 和 ksh(1) 手册页。有关 MANPATH 变量的详细信息，请参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问此发行版的详细信息，请参见安装指南或与系统管理员联系。

注 – 本节所含信息假定您的 Sun Studio 编译器和工具安装在 /opt 目录中。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

访问编译器和工具

按照下面的步骤确定您是否需要更改您的 PATH 变量以访问编译器和工具。

▼ 确定您是否需要设置您的 PATH 环境变量

1. **通过在命令提示符后面键入以下内容，显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **检查输出以查找包含 /opt/SUNWspro/bin/ 的路径的字符串。**

   如果您找到路径，您的 PATH 变量已设置为可访问编译器和工具。如果您没有找到路径，请按照下一过程中的指令设置 PATH 环境变量。

▼ 要设置您的 PATH 环境变量以便能够访问编译器和工具

1. **如果您正在使用 C shell，请编辑您的起始 .cshrc 文件。如果您正在使用 Bourne shell 或 Korn shell，请编辑您的起始 .profile 文件。**

2. **将以下内容增加到您的 PATH 环境变量中。如果安装了 Sun ONE Studio 软件或 Forte Developer 软件，请在其安装路径前面增加以下路径。**

   /opt/SUNWspro/bin

访问手册页

按照下列步骤确定您是否需要更改您的 MANPATH 变量以访问手册页。

▼ 确定您是否需要设置您的 MANPATH 环境变量

1. **通过在命令提示符后面键入以下内容，请求 dbx 手册页。**

```
% man dbx
```

2. **检查输出 （如果有）。**

   如果无法找到 dbx(1) 手册页，或者显示的手册页不适用于所安装软件的当前版本，请按照下一过程中的指令设置您的 MANPATH 环境变量。

▼ 设置您的 MANPATH 环境变量以便能够访问手册页

1. **如果您正在使用 C shell，请编辑您的起始 .cshrc 文件。如果您正在使用 Bourne shell 或 Korn shell，请编辑您的起始 .profile 文件。**

2. **将以下内容增加到您的 MANPATH 环境变量中。**

   /opt/SUNWspro/man

访问集成开发环境

Sun Studio 8 集成开发环境 (IDE) 提供用于创建、编辑、生成、调试和分析 C、 C++ 或 Fortran 应用程序性能的模块。

IDE 需要 Sun Studio 8 的核心平台组件。如果核心平台组件未安装，或者安装到以下某个位置，您必须将 SPRO_NETBEANS_HOME 环境变量设置为核心平台组件所安装的位置 (*installation_directory*/netbeans/3.5R)：

- 缺省安装目录 /opt/netbeans/3.5R
- Sun Studio 8 的编译器和工具组件的同一位置 （例如，编译器和工具组件安装在 /foo/SUNWspro 中，核心平台组件安装在 /foo/netbeans/3.5R 中）

用于启动 IDE 的命令是 sunstudio。有关此命令的详细信息，参见 sunstudio(1) 手册页。

访问编译器和工具文档

您可以从以下位置访问该文档：

- 该文档可通过随软件安装在您的本地系统或网络以下位置的文档索引获得 file:/opt/SUNWspro/docs/index.html。

  如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。
- 大多数手册可从 docs.sun.com[sm] web 站点获得。以下手册只能通过安装的软件获得：
  - *标准 C++ 库类参考*
  - *标准 C++ 库用户指南*
  - *Tools.h++ 类库参考*
  - *Tools.h++ 用户指南*
- 发行说明可从 docs.sun.com web 站点获得。
- 在 IDE 中， IDE 的所有组件的联机帮助可通过 [帮助] 菜单获得，也可通过许多窗口和对话框上的 [帮助] 按钮获得。

docs.sun.com web 站点 (http://docs.sun.com) 使您能够通过国际互联网阅读、打印和购买 Sun Microsystems 手册。如果您无法找到某本手册，请查看随软件安装在您的本地系统或网络中的文档索引。

注 – Sun 对于本文档中提到的第三方 web 站点的可用性概不负责，并且未对此类站点或资源上或从中获得的任何内容、广告、产品或其他资料作任何保证且概不负责。对于因为使用通过任何此类站点或资源获得的任何此类内容、商品或服务而造成或宣称造成的直接或间接损害或损失， Sun 概不负责。

可访问格式的文档

该文档以可访问格式提供，残疾人用户可通过辅助技术进行阅读。您可以找到下表所述文档的可访问版本。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

文档类型	可访问版本的格式和位置
手册 （第三方手册除外）	HTML，在 http://docs.sun.com 上
第三方手册： • *标准 C++ 库类参考* • *标准 C++ 库用户指南* • *Tools.h++ 类库参考* • *Tools.h++ 用户指南*	HTML，在安装的软件中，通过位于 file:/opt/SUNWspro/docs/index.html 的文档索引访问
自述文件和手册页	HTML，在安装的软件中，通过位于 file:/opt/SUNWspro/docs/index.html 的文档索引访问
联机帮助	HTML，可通过 IDE 中的 [帮助] 菜单访问
发行说明	HTML，在 http://docs.sun.com 上

相关的编译器和工具文档

下表描述了可从 file:/opt/SUNWspro/docs/index.html 和 http://docs.sun.com 获得的相关文档。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

文档标题	描述
数值计算指南	描述关于浮点计算的数值准确性的问题。

访问相关的 Solaris 文档

下表描述了可通过 docs.sun.com web 站点获得的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	参见手册页部分的标题。	提供有关 Solaris 操作环境的信息。
Solaris 软件开发者集合	*链接程序和库指南*	描述了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 软件开发者集合	*多线程编程指南*	包括 POSIX 和 Solaris 线程 API、使用同步对象编程、编译多线程程序以及查找多线程程序的工具。

开发者资源

访问 http://developers.sun.com/prodtech/cc 以查找以下经常更新的资源：

- 关于编程技巧和最佳实践的文章
- 包含简短编程提示的知识库
- 编译器和工具组件的文档以及随软件安装的文档的校正
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

您可以在 http://developers.sun.com 上找到适用于开发者的更多资源。

联系 Sun 技术支持

如果您有关于本产品的技术问题，而本文档中未涉及，请访问：

http://www.sun.com/service/contacting

发送您的意见

Sun 乐于改进文档并欢迎您发表意见和建议。请通过以下地址将您的意见 Email 给 Sun：

docfeedback@sun.com

请在您的电子邮件主题行中包含您的文档的部件号码 (817-5796-10)。

第 1 章

介绍 C 编译器

本章提供了有关 C 编译器的信息，包括编译器的操作环境、标准一致性、组织结构以及 C 相关的编程工具。

1.1 标准一致性

该编译器符合以下标准：

- ISO/IEC 9899:1990，编程语言 — C 标准。有关实现特定行为的信息，请参见附录 C。
- FIPS 160 标准。

此发行版还支持以下标准中指定的某些功能：

- ISO/IEC 9899:1999，编程语言 — C 标准。有关支持的功能的详细信息，请参见附录 D。

由于该编译器也支持传统的 K&R C （Kernighan 与 Ritchie，即 ANSI 之前的 C），因此便于迁移到 ISO C。

本书中所用术语 C99 是指 ISO/IEC 9899:1999 C 编程语言。术语 C90 是指 ISO/IEC 9899:1990 C 编程语言。

1.2 C 自述文件

C 编译器的自述文件重点阐述有关编译器的重要信息，包括：

- 手册印制之后发现的信息
- 新功能和更改的功能
- 软件校正
- 问题和回避方法
- 限制和不兼容性

要查看 C 自述文件的文本版本，请在命令提示符下键入以下内容：

```
example% cc -xhelp=readme
```

要访问自述文件的 HTML 版本，请在 Netscape Communicator 4.0 或兼容版本浏览器中打开以下文件：

```
/opt/SUNWspro/docs/index.html
```

（如果 C 编译器软件未安装在 `/opt` 目录中，请询问系统管理员以了解系统中的等价路径。）您的浏览器将显示 HTML 文档的索引。要打开自述文件，请在索引中查找其条目，然后单击标题。

1.3 手册页

联机手册 (`man`) 页提供了关于命令、函数、子例程或诸如此类的集合的立即文档。

您可以通过运行以下命令显示手册页：

```
example% man topic
```

在整个 C 文档中，手册页参考以主题名称和手册节编号显示：`cc(1)` 通过 `man cc` 进行访问。其它节 （例如以 `ieee_flags(3M)` 表示的节）通过在 `man` 命令中使用 `-s` 选项进行访问：

```
example% man -s 3M ieee_flags
```

1.4 新功能

C 编译器在此发行版中引入以下新功能。

1.4.1 一般增强

- 变量作用域不再需要链接程序映射文件：`-xldscope`

  现在您可以使用两种不同的方法来控制动态库中符号的输出。此功能称为链接程序作用域，已获得链接程序映射文件一段时间的支持。首先，现在您可以在代码中嵌入新的声明说明符。

  通过直接在代码中嵌入 `__global`、`__symbolic` 和 `__hidden`，不再需要使用映射文件。其次，您可以通过在命令行中指定 `-xldscope`，覆盖变量作用域的缺省设置。

  有关详情，请参见第 2-3 页上的 “ 链接程序作用域说明符 ” 和第 A-50 页上的 “`-xldscope={v}`” 。

- 实现附加 C99 功能

  此发行版增加了对以下 ISO/IEC 9899:1999 （本文档中称为 C99）功能的支持。下面列表仅详细列出此发行版中实现的 C99 功能，它是所有实现的 C99 功能的子集。有关 C 编译器的过去和当前发行版中实现的所有 C99 功能的完整列表，请参见附录 D 第 D-1 页上的 “ 支持的 C99 功能 ” 。每项均列出 C99 标准的子节编号。

  - 6.2.5 _Bool
  - 6.2.5 _Complex 类型

    此发行版支持部分实现 _Complex。在 Solaris 7 操作环境、 Solaris 8 操作环境和 Solaris 9 操作环境中，您必须链接 `-lcplxsupp`。尚未实现 _Complex 数据的静态初始化。

  - 6.3.2.1 数组到指针的转换不仅限于左值
  - 6.4.4.2 十六进制浮点文字
  - 6.5.2.5 复合文字
  - 6.7.2 类型说明符
  - 6.10.6 STDC pragma
  - 6.10.8 __STDC_IEC_559 和 __STDC_IEC_559_COMPLEX 宏

- 支持 VIS™ 开发者工具包：`-xvis` (SPARC)

  如果您正在使用 VIS 指令集软件开发者工具包 (VSDK) 中定义的汇编语言模板，请使用 `-xvis=[yes|no]` 选项。

  VIS 指令集是 SPARC v9 指令集的扩展。尽管 UltraSPARC 处理器是 64 位，但是在很多情况下数据长度仅限于 8 位或 16 位，特别是在多媒体应用程序中。 VIS 指令可以用一条指令处理 4 个 16 位数据，因此极大地提高了处理图像、线性代数、信号处理、音频、视频及网络等新媒体的应用程序的性能。

  有关 VSDK 的详细信息，请访问 `http://www.sun.com/processors/vis`。参见第 A-85 页上的 “`-xvis`” 。

- 从属线程的缺省栈大小更大

  从属线程的缺省栈大小现在更大。所有从属线程的栈大小都相同。对于 32 位应用程序，栈大小为 4 兆字节；对于 64 位应用程序，栈大小为 8 兆字节。栈大小通过环境变量 STACKSIZE 进行设置。

  参见第 3-4 页上的 “`STACKSIZE`” 。

- 改进的 `-xprofile` (SPARC)

  `-xprofile` 选项提供以下改进：

  - 支持文件配置共享库
  - 使用 -xprofile=collect -mt 的线程安全配置文件收集
  - 改进了对单个配置文件目录中配置多个程序的支持。

  通过设置 `-xprofile=use`，编译器能够在包含具有非唯一基名的多个目标文件的数据的配置文件目录中查找配置文件数据。如果编译器无法找到目标文件的配置文件数据，它将提供一个新选项 `-xprofile_pathmap=`*collect-prefix*`:` *use-prefix*。

  参见第 A-69 页上的 “`-xprofile=p`” 和第 A-72 页上的 “`-xprofile_pathmap`” 。

- 支持 UTF-16 字符串文字：`-xustr`

  如果需要支持使用 ISO10646 UTF-16 字符串文字的国际化应用程序，请指定 `-xustr=ascii_utf16_ushort`。此选项使系统能够将 U"*ASCII_string*" 字符串文字识别为无符号短数组类型。

  参见第 A-84 页上的 “`-xustr={ascii_utf16_ushort|no}`” 。

  参见第 5-16 页上的 “`-Xustr={ascii_utf16_ushort|no}`” ，以了解 `lint` 公用程序的的等价选项。

1.4.2 更快编译

- 更快的文件配置：`-xprofile_ircache` (SPARC)

  使用 `-xprofile_ircache[=`*path*`]` 及 `-xprofile=collect|use`，通过重用收集阶段保存的编译数据，减少 `use` 阶段的编译时间。

  对于大程序，由于保存了中间数据，因此 `use` 阶段的编译时间可显著减少。注意，保存的数据会占用相当大的磁盘空间。

  参见第 A-72 页上的 “`-xprofile_ircache[=path]`” 。

- 预编译的头文件：`-xpch`

  编译器的此发行版引入新的预编译头文件功能。预编译头文件的作用是减少其源文件共享同一组包含文件的应用程序的编译时间，而这些包含文件包含大量源代码。预编译头文件的工作机理是：首先从一个源文件收集关于某个头文件序列的信息，然后在重新编译该源文件或者其它具有相同头文件序列的源文件时使用这些信息。您可以通过 `-xpch` 和 `-xpchstop` 选项以及 `#pragma hdrstop` 指令使用此功能。

  参见第 A-64 页上的 “`-xpch=v`” 、第 A-67 页上的 “`-xpchstop=file`” 和第 2-13 页上的 “hdrstop” 。

- 使用多处理器：`-xjobs=`*n* (SPARC)

  指定 `-xjobs=`*n* 选项，以设置编译器需要创建多少个进程来完成其工作。在多 cpu 机器上，此选项可以减少生成时间。当前， `-xjobs` 只能与 `-xipo` 选项一起使用。当您指定 `-xjobs=`*n* 时，过程间调用优化器使用 *n* 作为它可调用以编译不同文件的最大代码生成器实例数。

  参见第 A-50 页上的 “`-xjobs=n`” 。

1.4.3 改进的性能

- 使用链接程序支持的线程本地存储减少运行时间：`-xthreadvar`

  使用编译器新的链接程序支持的线程本地存储功能完成以下任务：

  - 利用用于分配线程特定数据的 POSIX 接口的快速实现。
  - 将多进程程序转换成多线程程序。
  - 将使用线程本地存储的 Windows 应用程序移植到 Solaris 操作环境。
  - 利用 OpenMP 中线程专用变量的快速实现。

  通过声明线程本地变量，编译器中现在可提供线程本地存储。声明由一个标准变量声明外加变量说明符 `__thread` 以及命令选项 `-xthreadvar` 组成。

  有关详细信息，请参见第 2-4 页上的 “ 线程本地存储说明符 ” 和第 A-82 页上的 “`-xthreadvar[=o]`” 。

- 通过减少缺页减少运行时间：-xF

  使用 -xF 的新功能性使链接程序对变量和函数的重新排序达到最佳。这有助于解决对运行时性能产生负面影响的以下问题：

  - 由内存中相互靠近的无关变量造成的缓存和页争用。
  - 由于内存中相互较远的相关变量而造成工作集大小不必要地较大。
  - 由于未使用可降低有效数据密度的弱变量副本而造成工作集大小不必要地较大。

- 减少运行时间：-xlinkopt (SPARC)

  当您指定 -xlinkopt 命令时， C++ 编译器可以对可重定位目标文件进行链接时优化。

  指定 -xlinkopt 选项时，编译器在链接时进行某些附加优化，而不会修改链接的 .o 文件。这些优化仅在可执行程序中出现。 -xlinkopt 选项在您编译整个程序时十分有效，并提供配置文件反馈。

  参见第 A-52 页上的 “-xlinkopt[=level]” 。

- 减少运行时间：-xpagesize=*n* (SPARC)

  设置栈在内存中的页大小。*n* 可以为 8K、64K、512K、4M、32M、256M、2G、16G 或 default。您必须在目标平台上为 Solaris 操作环境指定有效的页大小，getpagesize(3C) 会返回该值。如果不指定有效的页大小，运行时将忽略该请求而不提示。您可以使用 pmap(1) 或 meminfo(2) 来确定目标平台上的页大小。

注 – 此功能仅适用于 Solaris 软件。在早期的 Solaris 操作环境中，使用此选项编译的程序将不会链接。

  此选项是 -xpagesize_stack 和 -xpagesize_heap. 的宏。

  参见第 A-62 页上的 “-xpagesize=n” ，第 A-63 页上的 “-xpagesize_heap=n” 和第 A-63 页上的 “-xpagesize_stack=n” 。

- 基于硬件计数器的文件配置：-xhwcprof (SPARC)

  使用 -xhwcprof=[enable|disable] 选项启用编译器对基于硬件计数器的文件配置的支持。

  启用 -xhwcprof 时，编译器生成信息，以帮助工具匹配硬件计数器数据引用并忽略具有关联指令的事件。相应的数据类型和结构成员也会与符号信息 （生成时带有 -g）一起标识。此信息在性能分析中很有用，因为根据代码地址、源代码语句或例程从配置文件中识别它并不容易。

  参见第 A-46 页上的 “-xhwcprof” 。

1.4.4 更方便调试

- Dwarf 格式的调试器信息：`-xdebugformat`

  如在 “DWARF 调试信息格式”中指定的那样， C 编译器会将调试信息从 stabs 格式迁移到 dwarf 格式。如果要维护读取调试信息的软件，您可以使用该选项将工具从 stabs 格式转换为 Dwarf 格式。在此发行版中，缺省设置为 `-xdebugformat=stabs`。

  使用 `-xdebugformat=dwarf` 选项作为移植工具时访问新格式的一种方法。除非您要维护读取调试器信息的软件，或者某个特定工具要求使用这些格式之一的调试器信息，否则没有必要使用此选项。

  参见第 A-43 页上的 “`-xdebugformat=[stabs|dwarf]`” 。

- 支持对 OpenMP 程序的调试：`-xopenmp=noopt`

  如果您正在使用 dbx 调试 OpenMP 程序，请使用 `-g` 和 `-xopenmp=noopt` 进行编译，以便您可以在并行区域中设置断点并显示变量的内容。

  参见第 A-61 页上的 “`-xopenmp[=i]`” 。

1.5 编译器的组织结构

C 编译系统由一个编译器、一个汇编程序和一个链接编辑器组成。除非您使用命令行指定，否则 `cc` 命令自动调用每个组件。

附录 A 讨论 `cc` 的所有可用选项。

下图显示 C 编译系统的组织结构。

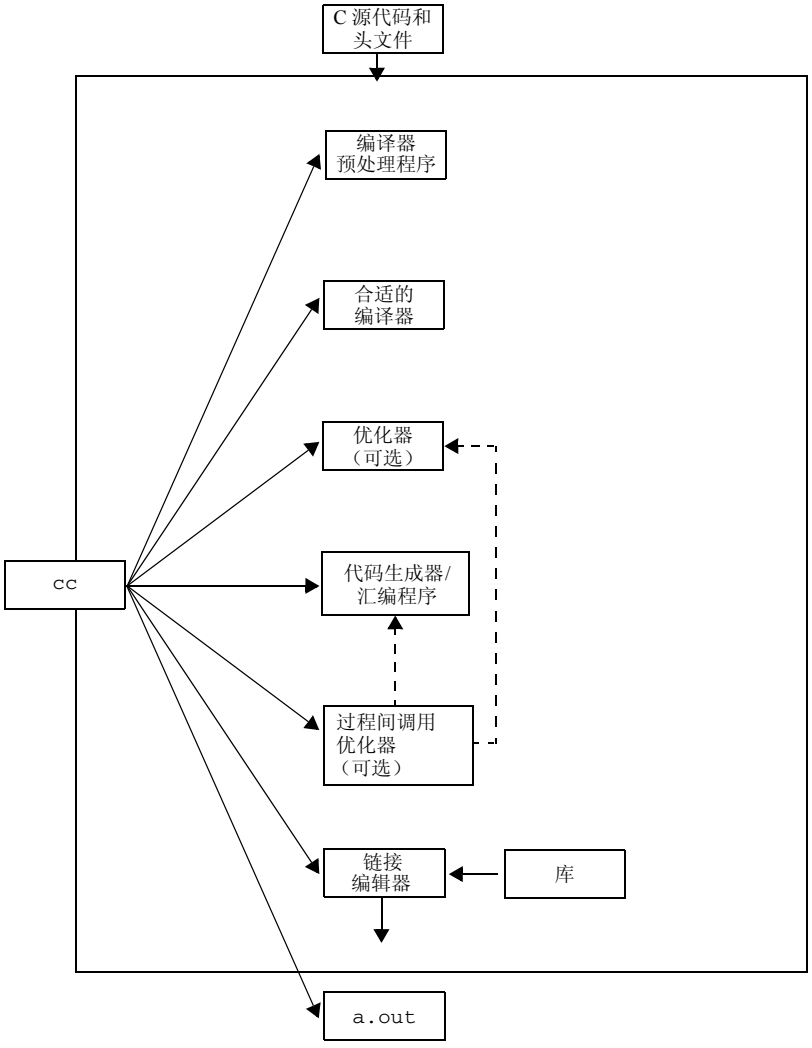

图 1-1 C 编译系统的组织结构

下表汇总了编译系统的组件。

表 1-1 C 编译系统的组件

组件	描述	使用说明
cpp	预处理程序	仅适用于 -Xs
acomp	编译器 （用于非 -Xs 模式的内建预处理程序）	
ssbd	静态同步错误检测	*(SPARC)*
iropt	代码优化器	*(SPARC)* -O, -xO2, -xO3, -xO4, -xO5, -fast
fbe	汇编程序	
cg	代码生成器、内联函数、汇编程序	*(SPARC)*
ipo	过程间调用优化器	*(SPARC)*
postopt	后优化器	*(SPARC)*
ir2hf	中间代码翻译者	(INTEL)
ube	代码生成器	(INTEL)
ube_ipa	过程间调用分析器	(INTEL)
ld	链接程序	
ild	递增链接程序	*(SPARC)* -g, -xildon
mcs	处理注释部分	-mr

1.6 C 相关编程工具

有多种工具可用来帮助您开发、维护和改进 C 程序。本书中介绍了两种与 C 联系最紧密的工具：cscope 和 lint。此外，每个工具均有手册页。

还提供其它用于源代码浏览、调试和性能分析的工具。有关详细信息，请参见第 xxxv 页上的“ 访问编译器和工具文档 ”。

第 2 章

特定于 Sun 实现的 C 编译器信息

本章讨论特定于 C 编译器的方面。该信息已编入语言扩展和环境。

C 编译器与新的 ISO C 标准 (ISO/IEC 9899-1999) 中描述的 C 语言的某些功能兼容。如果您要编译与以前的 C 标准 （ISO/IEC 9889-1990 标准及修订版 1）兼容的代码，请使用 `-xc99=%none`，编译器会忽视 ISO/IEC 9899-1999 标准的增强标准。

2.1 常量

本节包含与特定于 Sun C 编译器的常量相关的信息。

2.1.1 整型常量

十进制、八进制和十六进制的整型常量可加后缀以指示类型，如下表所示。

表 2-1 数据类型后缀

后缀	类型
`u` 或 `U`	`unsigned`
`u` 或 `U`	`long`
`ll` 或 `LL`	`long long`[1]
`lu`、`LU`、`Lu`、`lU`、`ul`、`uL`、`Ul` 或 `UL`	`unsigned long`
`llu`、`LLU`、`LLu`、`llU`、`ull`、`ULL`、`uLL`、`Ull`	`unsigned long long`[1]

1 `long long` 和 `unsigned long long` 在 `-xc99=%none` 和 `-Xc` 模式下不可用。

如果设置 -xc99=%all，编译器按常量大小的要求，使用以下列表中可表示值的第一项。

- int
- long int
- long long int

如果值超过 long long int 可表示的最大值，编译器会发出警告。

如果设置 -xc99=%none，为无后缀常量指定类型时，编译器按常量大小的要求，使用以下列表中可表示值的第一项。

- int
- long int
- unsigned long int
- long long int
- unsigned long long int

2.1.2 字符常量

本身不是换码序列的多字符常量具有从每个字符的数值派生的值。例如，常量 '123' 的值为：

0	'3'	'2'	'1'

或 0x333231。

使用 -Xs 选项并且在 C 的其它非 ISO 版本中，该值为：

0	'1'	'2'	'3'

或 0x333231。

2.2 链接程序作用域说明符

使用以下声明说明符帮助隐藏外部符号的声明和定义。通过使用这些说明符，您不需要再对链接程序作用域使用映射文件。您还可以通过在命令行上指定 -xldscope 以控制变量作用域的缺省设置。有关详细信息，请参见第 A-50 页上的 “-xldscope={v}”。

表 2-2 声明说明符

值	含义
__global	该符号具有全局链接程序作用域，并且是限制最少的链接程序作用域。该符号的所有引用都绑定到定义符号的第一个动态模块中的定义上。该链接程序作用域是外部符号的当前链接程序作用域。
__symbolic	该符号拥有符号链接程序作用域，该作用域的限制比全局链接程序作用域的限制更多。该符号的来自所链接的动态模块内部的所有引用都绑定到模块内部定义的符号上。在模块外部，该符号如同全局符号。此链接程序作用域对应于链接程序选项 -Bsymbolic。有关链接程序的详细信息，请参见 ld(1)。
__hidden	该符号具有隐藏的链接程序作用域。隐藏的链接程序作用域的限制比符号链接程序作用域和全局链接程序作用域的限制更多。一个动态模块内部的所有引用都绑定到该动态模块内部的一个定义上。该符号在模块外部不可见。

对象或函数可以用限制较多的说明符重新声明，但不能用限制较少的说明符重新声明。符号被定义后，就不能用其它说明符来声明。

__global 是限制最少的作用域，__symbolic 限制较多，__hidden 是限制最多的作用域。

2.3 线程本地存储说明符

要使用线程本地存储，需要声明线程本地变量。线程本地变量声明由一个标准变量声明外加变量说明符 `__thread` 组成。有关详细信息，请参见第 A-82 页上的“`-xthreadvar[=o]`”。

您必须在所编译的源文件中的线程变量的第一个声明中包含 `__thread` 说明符。

在具有静态存储持续时间的对象的声明中只能使用 `__thread` 说明符。您可以如初始化任何其它静态存储持续时间的对象一样静态地初始化线程变量。

使用 `__thread` 说明符声明的变量与不使用 `__thread` 说明符声明的变量具有相同的链接程序绑定。这包括临时定义，如无初始化函数的声明。

线程变量的地址不是常量。因此，线程变量的地址操作符 (&) 在运行时求值，并返回当前线程的线程变量的地址。结果，静态存储持续时间的对象被动态地初始化为线程变量的地址。

线程变量的地址在相应线程的生存期内稳定。在变量的存在期内，进程中的任何线程都可以自由地使用线程变量的地址。线程终止后，您不能使用线程变量的地址。线程终止后，该线程的变量的所有地址都无效。

2.4 浮点，非标准模式

IEEE 754 浮点缺省运算是 “不停止”。下溢是 “渐进式”。以下是一个摘要，有关详情请参见 *Numerical dinsstanleyComputation Guide*。

不停止意味着遇到除数为零、浮点下溢或无效操作异常时执行不会停止。例如，考虑以下算式，其中 `x` 为零， `y` 为正数：

```
z = y / x;
```

缺省情况下， `z` 设置为值 `+Inf`，执行不会停止。但是，如果设置 `-fnonstd` 选项，此代码会导致退出，如信息转储。

下面是渐进下溢的工作方式。假设您有下列代码：

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
x = x / 10;
```

第一次执行循环时， x 设置为 1，第二次执行循环时，设置为 0.1，第三次执行循环时，设置为 0.01，依此类推。最后， x 达到机器能力的下限而无法表示其值。下次循环运行时将出现什么情况？

假设可表示的最小的数为 1.234567e-38

下次循环运行时，将通过去掉一位尾数并且指数减去 1 来修改该数，因此新值为 1.23456e-39，然后为 1.2345e-40，依此类推。这称为 “渐进下溢”，它是缺省行为。在非标准模式下，不会发生这种 “去位”情况，通常， x 被简单地设置为零。

2.5 作为值的标签

C 编译器可识别称为程序决定的 goto 的 C 扩展。程序决定的 goto 能够在运行时确定分支目标。通过使用 “&&”运算符可以获取标签的地址，并且可以为标签地址指定 void * 类型的指针：

```
void *ptr;
...
ptr = &&label1;
```

后面的 goto 语句可以通过 ptr 转到 label1：

```
goto *ptr;
```

由于 ptr 在运行时计算，因此 ptr 可以表示作用域内任何标签的地址， goto 语句可以转到该位置。

使用程序决定的 goto 的一种方法是用于转移表的实现：

```
static void *ptrarray[] = { &&label1, &&label2, &&label3 };
```

现在可以通过索引来选择数组元素：

```
goto *ptrarray[i];
```

标签的地址只能通过当前函数作用域计算。尝试在当前函数外部获取标签的地址会产生不可预测的结果。

转移表和开关语句的作用相似 （虽然存在某些主要差异），转移表使跟踪程序流更加困难。一个显著的差异是：开关语句转移目标全都从开关保留字开始正向转移；使用程序决定的 goto 实现转移表可进行正向和反向分支。

```
#include <stdio.h>
void foo()
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return;

  label1:
  printf("Passed!\n");
  return;
}

int main(void)
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return 0;

  label1:
  foo();
  return 0;
}
```

以下示例也使用转移表控制程序流：

```
#include <stdio.h>

int main(void)
{
  int i = 0;
  static void * ptr[3]={&&label1, &&label2, &&label3};

  goto *ptr[i];

  label1:
  printf("label1\n");
  return 0;

  label2:
  printf("label2\n");
  return 0;

  label3:
  printf("label3\n");
  return 0;
}

%example: a.out
%example: label1
```

程序决定的 goto 的另一个应用是作为线程代码的解释程序。解释程序函数内部的标签地址可以存储在线程代码中以便快速分发。

下面是编写以上示例的另一种方法：

```
static const int ptrarray[] = { &&label1 - &&label1,
&&label2 - &&label1, &&label3 - &&label1 };
goto *(&&label1 + ptrarray[i]);
```

这对于共享库代码效率更高，因为它减少了所需的动态重定位数量，因此，允许数据（ptrarray 元素）为只读。

2.6 long long 数据类型

当您使用 -xc99=%none 进行编译时， Sun C 编译器包含数据类型 long long 和 unsigned long long，它们与数据类型 long 相似。 long long 数据类型存储 64 位信息；在 SPARC V8 和 x86 上，long 存储 32 位信息。在 SPARC V9 上，long 数据类型存储 64 位信息。 long long 数据类型在 -Xc 模式下不可用。

2.6.1 打印 long long 数据类型

要打印或扫描 long long 数据类型，请在转换说明符前面加字母 ll。例如，要以带符号十进制格式打印 llvar （long long 数据类型的变量），请使用：

```
printf("%lld\n", llvar);
```

2.6.2 常见算术转换

某些二元运算符将其操作数的类型转换为普通类型，它也是结果的类型。下面这些转换称为常见算术转换：

- 如果一个操作数的类型为 long double，则另一个操作数的类型转换为 long double。
- 另外，如果一个操作数的类型为 double，则另一个操作数的类型转换为 double。
- 另外，如果一个操作数的类型为 float，则另一个操作数的类型转换为 float。
- 另外，对两个操作数执行整型提升。然后，应用以下规则：
  - 如果一个操作数的类型为 unsigned long long int，则另一个操作数的类型转换为 unsigned long long int。
  - 如果一个操作数的类型为 long long int，则另一个操作数的类型转换为 long long int。
  - 如果一个操作数的类型为 unsigned long int，则另一个操作数的类型转换为 unsigned long int。
  - 另外，当您只在 SPARC V9 上进行编译并指定 cc -xc99=none 时，如果一个操作数的类型为 long int 而另一个操作数的类型为 unsigned int，则两个操作数的类型均转换为 unsigned long int。
  - 另外，如果一个操作数的类型为 long int，则另一个操作数的类型转换为 long int。

- 另外，如果一个操作数的类型为 unsigned int，则另一个操作数的类型转换为 unsigned int。
- 另外，两个操作数的类型均为 int。

2.7 断言

以下形式的一行：

```
#assert predicate (token-sequence)
```

将 *token-sequence* 和断言名称空间 （与用于宏定义的空间分开）中的谓词联合。谓词必须为标识符标记。

```
#assert predicate
```

断言 *predicate* 存在，但是未与任何标记序列联合。

在缺省情况下，编译器提供以下预定义谓词 （不在 -Xc 模式下）：

```
#assert system (unix)
#assert machine (sparc)(SPARC)
#assert machine (i386)(Intel)
#assert cpu (sparc)(SPARC)
#assert cpu (i386)(Intel)
```

在缺省情况下， lint 提供以下预定义谓词 （不在 -Xc 模式下）：

```
#assert lint (on)
```

任何断言均可使用 #unassert 进行删除，该命令的语法与 assert 的语法相同。使用不带参数的 #unassert 将删除关于谓词的所有断言；指定一个断言将只删除该断言。

可以使用以下语法在 #if 语句中测试断言：

```
#if #predicate(non-empty token-list)
```

例如，可以使用以下行测试预定义谓词 `system`：

```
#if #system(unix)
```

其结果为真。

2.8 Pragma

以下形式的预处理行：

```
#pragma pp-tokens
```

指定实现定义的操作。

以下 `#pragma` 由编译系统识别。编译器忽略未识别的 pragma。使用 `-v` 选项将为未识别的 pragma 发出警告。

2.8.1 对齐

`#pragma align` *integer* (*variable*[, *variable*])

对齐 pragma 使所有提到的变量内存调整为*整数*字节，覆盖缺省值。具有以下限制：

- *integer* 值必须为 2 的幂，并且介于 1 和 128 之间，有效值为：1、2、4、8、16、32、64 和 128。
- *variable* 是全局或静态变量，它不能为自动变量。
- 如果指定的对齐小于缺省值，则使用缺省值。
- pragma 行必须在它提到的变量的声明前面出现；否则，它将被忽略。
- 提到但未在 pragma 行后面的文本中声明的任何变量将被忽略。例如：

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct astruct{int a; char *b;};
```

2.8.2 does_not_read_global_data

`#pragma does_not_read_global_data (`*funcname* [, *funcname*]`)`

该 pragma 断言指定列表中的例程不直接或间接读取全局数据。这可以更好地优化此类例程的调用周围的代码。特别是可以在此类调用的周围移动赋值语句或存储。

指定的函数必须在该 pragma 之前使用原型或空参数列表进行声明。如果关于全局访问的断言不为真，则程序的行为未定义。

2.8.3 does_not_return

`#pragma does_not_return (`*funcname* [, *funcname*]`)`

该 pragma 向编译器断言：对指定例程的调用将不会返回值。这允许编译器执行符合该假定的优化。例如，寄存器生存期将在调用点终止，从而又允许更多的优化。

如果指定的函数返回值，则程序的行为未定义。该 pragma 只能在使用原型或空参数列表声明指定的函数之后出现，如以下示例所示：

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

2.8.4 does_not_write_global_data

`#pragma does_not_write_global_data (`*funcname [, funcname]*`)`

该 pragma 断言指定列表的例程不直接或间接写全局数据。这可以更好地优化此类例程的调用周围的代码。特别是可以在此类调用的周围移动赋值语句或存储。

指定的函数必须在该 pragma 之前使用原型或空参数列表进行声明。如果关于全局访问的断言不为真，则程序的行为未定义。

2.8.5 error_messages

`#pragma error_messages (on|off|default, tag… tag)`

错误消息 pragma 提供源程序内部对 C 编译器和 lint 发出的消息的控制。对于 C 编译器，pragma 只对警告消息有效。 C 编译器的 `-w` 选项通过禁止所有警告消息覆盖该 pragma。

- `#pragma error_messages (on, tag… tag)`

  `on` 选项结束任何前面的 `#pragma error_messages` 选项 （如 off 选项）的作用域，并且覆盖 `-erroff` 选项的效果。

- `#pragma error_messages (off, tag… tag)`

  `off` 选项防止 C 编译器或 lint 程序发出以 pragma 中指定的标记开头的指定消息。pragma 对任何指定的错误消息的作用域仍有效，直到被另一个 `#pragma error_messages` 覆盖或编译结束时为止。

- `#pragma error_messages (default, tag… tag)`

  `default` 选项结束任何前面的 `#pragma error_messages` 指令对指定标记的作用域。

2.8.6 fini

`#pragma fini (f1[, f2…,fn])`

导致在调用 `main()` 例程之后调用函数 *f1* 至 *fn* （finalization 函数）。此类函数的类型应为 `void`，并且不带任何参数，在程序正常终止或所属共享对象被从内存中删除时调用。和 “初始化函数”一样， finalization 函数按链接编辑器的处理顺序执行。

2.8.7 hdrstop

```
#pragma hdrstop
```

hdrstop pragma 必须放在最后一个头文件之后，以标识要共享相同预编译头文件的每个源文件中活动前缀的结束。例如，考虑下列文件：

```
example% cat a.c
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.h
#include "a.h"
#include "b.h"
#include "c.h"
```

活动源代码前缀在 c.h 结束，因此您需要在每个文件中在 c.h 后面插入 #pragma hdrstop。

#pragma hdrstop 必须只在使用 cc 命令指定的源文件的活动前缀的末尾出现。不要在任何包含文件中指定 #pragma hdrstop。

2.8.8 ident

```
#pragma ident string
```

将 *string* 放在可执行文件的 `.comment` 部分。

2.8.9 init

```
#pragma init (f1[, f2...,fn])
```

导致在调用 `main()` 之前调用函数 *f1* 至 *fn*（初始化函数）。此类函数的类型应为 `void`，并且不带任何参数，在开始执行时构造程序的内存映像时调用。如果初始化程序在共享对象中，则在执行将共享对象放入内存的操作时执行，该操作可以是程序启动，也可以是某些动态加载操作，如 `dlopen()`。调用初始化函数的唯一顺序是链接编辑器处理它们的顺序，静态和动态均可。

2.8.10 inline

`#pragma [no_]inline (`*funcname[, funcname]*`)`

该 pragma 控制 pragma 命令的参数中列出的例程名称的内联。该 pragma 的作用域针对整个文件。该 pragma 只允许全局内联控制，不允许特定于调用点的控制。

如果您使用 `#pragma inline`，它建议编译器内联当前文件中与 pragma 中列出的例程列表匹配的调用。在某些情况下，此建议可能被忽略。例如，当函数的主体在另一个模块并且未使用交叉文件选项时，忽略该建议。

如果您使用 `#pragma no_inline`，它建议编译器内联当前文件中与 pragma 中列出的例程列表匹配的调用。

只有在使用原型或空参数列表声明函数之后，才允许使用 `#pragma inline` 和 `#pragma no_inline`，如以下示例所示：

```
static void foo(int);
static int bar(int, char *);
#pragma inline(foo, bar)
```

参见 `-xldscope`、 `-xinline`、 `-xO` 和 `-xcrossfile`。

2.8.11 int_to_unsigned

`#pragma int_to_unsigned (`*funcname*`)`

对于返回类型 `unsigned` 的函数，在 `-Xt` 或 `-Xs` 模式下，将函数返回值的类型更改为 `int`。

2.8.12 MP serial_loop

(SPARC) `#pragma MP serial_loop`

有关详情，请参见第 3-20 页上的第 3.8.3.1 节 “串行 Pragma”。

2.8.13 MP serial_loop_nested

(SPARC) `#pragma MP serial_loop_nested`

有关详情，请参见第 3-20 页上的第 3.8.3.1 节 “串行 Pragma”。

2.8.14 MP taskloop

(SPARC) `#pragma MP taskloop`

有关详情，请参见第 3-20 页上的第 3.8.3.2 节 “并行 Pragma”。

2.8.15 nomemorydepend

(SPARC) `#pragma nomemorydepend`

该 pragma 指定，对于某个循环的任何迭代，不存在内存依赖性。也就是说，在某个循环的任何迭代内部，不存在对相同内存的引用。该 pragma 将允许编译器（流水线化程序）在某个循环的单次迭代内更有效地调度指令。如果某个循环的任何迭代内部存在任何内存依赖性，则程序的执行结果未定义。该 pragma 应用于当前块内部的下一个 `for` 循环。编译器在优化级别 3 或更高级别上使用此信息。

2.8.16 no_side_effect

(SPARC) `#pragma no_side_effect` (*funcname[, funcname…]*)

funcname 指定当前转换单元内部某个函数的名称。必须在 pragma 之前使用原型或空参数列表声明函数。必须在函数的定义之前指定 pragma。对于命名的函数 *funcname*，该 pragma 声明函数无任何副作用。这意味着， *funcname* 返回一个只依赖传递参数的结果值。另外， *funcname* 和任何已调用的子函数具有以下特性：

- 不读取或写入在调用点可被调用程序识别的程序状态的任何部分。
- 不执行 I/O。
- 不更改在调用点可被识别的程序状态的任何部分。

使用函数进行优化时，编译器使用此信息。如果函数确实有副作用，则调用此函数的程序的执行结果未定义。编译器在优化级别 3 或更高级别上使用此信息。

2.8.17 opt

`#pragma opt` *level* (*funcname[, funcname]*)

funcname 指定当前转换单元内部定义的函数的名称。 *level* 的值指定命名的函数的优化级别。您可以指定优化级别 0、1、2、3、4、5。您可以通过将 *level* 设置为 0 来关闭优化。必须在 pragma 之前使用原型或空参数列表声明函数。该 pragma 必须在要优化的函数的定义之前。

pragma 中列出的任何函数的优化级别被降低为值 `-xmaxopt`。当 `-xmaxopt=off` 时，忽略 pragma。

2.8.18 pack

(SPARC) `#pragma pack(n)`

使用 `#pragma pack(n)` 影响结构或联合的成员压缩。在缺省情况下，结构或联合的成员在其自然边界上对齐；一个 char （字符）型数据占一个字节，一个 short 型数据占两个字节，一个整数占四个字节，等等。如果存在 *n*，它必须为 2 的幂，并且为任何结构或联合成员指定最严格的自然对齐。不接受零。

您可以使用 `#pragma pack(n)` 为结构或联合成员指定对齐边界。例如， `#pragma pack(2)` 在双字节边界而不是其自然边界上对齐 int、 long、 long long、 float、 double、 long double 和指针。

如果 *n* 等于或大于您使用的平台上最严格的对齐 （在 Intel 上为 4，在 SPARC v8 上为 8，在 SPARC v9 上为 16），则该指令具有自然对齐的效果。同样，如果忽略 *n*，成员对齐将恢复为自然对齐边界。

`#pragma pack(n)` 指令应用于它后面的所有结构或联合定义，直到出现下一个 pack 指令时为止。如果使用不同的压缩方式在不同的转换单元中定义相同的结构或联合，程序可能会以不可预测的方式失败。特别是不应在包含定义了预编译库的接口的头文件之前使用 `#pragma pack(n)`。推荐的 `#pragma pack(n)` 用法是将它放在紧挨在要压缩的任何结构或联合前面的程序代码中。在压缩的结构后面紧接着使用 `#pragma pack( )`。

请注意，使用 `#pragma pack` 时，压缩的结构或联合本身的对齐与其更严格对齐的成员相同。因此，该结构或联合的任何声明将使用压缩对齐。例如，一个只包含 char 型数据的结构无对齐限制，然而包含 double 型数据的结构将在 8 字节边界上对齐。

注 – 如果您使用 `#pragma pack` 在不同于其自然边界的边界上对齐结构或联合成员，则访问这些字段会导致 SPARC 上出现总线错误。有关编译此类程序的最佳方法，请参见第 A-56 页上的第 A.3.102 节 “`-xmemalign=ab`”。

2.8.19 pipeloop

(SPARC) `#pragma pipeloop(n)`

对于参数 *n*，该 pragma 接受正整数常量值或 0。该 pragma 指定，循环可流水线化，并且循环携带的依赖性的最小依赖距离为 *n*。如果该距离为 0，则循环实际上是 Fortran 风格的 `doall` 循环，并且应在目标处理程序上流水线化。如果该距离大于 0，则编译器（流水线化程序）将只尝试流水线化 *n* 次连续迭代。该 pragma 应用于当前块内部的下一个 `for` 循环。编译器在优化级别 3 或更高级别上使用此信息。

2.8.20 rarely_called

`#pragma rarely_called(`*funcname[, funcname]*`)`

该 pragma 命令向编译器发出提示，指出指定的函数不经常被调用。这允许编译器在无需配置文件收集阶段的开销的情况下，在此类例程的调用点上执行配置文件反馈式优化。由于该 pragma 是一个建议，因此编译器可能不执行基于该 pragma 的任何优化。

指定的函数必须在该 pragma 之前使用原型或空参数列表进行声明。下面是 `#pragma rarely_called` 的一个示例：

```
extern void error (char *message);
#pragma rarely_called(error)
```

2.8.21 redefine_extname

`#pragma redefine_extname` *old_extname new_extname*

该 pragma 导致对象代码中名称 *old_extname* 的各个外部定义名称被 *new_extname* 替换。结果，链接程序在链接时只看到名称 *new_extname*。如果在第一次使用 *old_extname*（作为函数定义、初始化函数或表达式）之后遇到 `#pragma redefine_extname`，则该作用未定义。（在 `-Xs` 模式下不支持该 pragma。）

如果 `#pragma redefine_extname` 可用，则编译器提供预定义宏 `PRAGMA_REDEFINE_EXTNAME` 的定义，从而使您可以编写在有无 `#pragma redefine_extname` 的条件下均可运行的可移植代码。

#pragma redefine_extname 的目的是提供一种在函数的名称无法更改时重新定义函数接口的有效方法。例如，当某个库中必须保留初始函数定义时，为了与现有程序兼容，创建同一函数的新定义以供新程序使用。这可以通过用新名称将新函数定义增加到库中来完成。因此，声明函数的头文件使用 #pragma redefine_extname，以便对函数的所有使用均与该函数的新定义链接。

```
#if    defined(__STDC__)

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine(const long *, int *);
#pragma redefine_extname myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(const long * arg1, int * arg2)
{
    extern int __myroutine(const long *, int*);
    return (__myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#else /* __STDC__ */

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine();
#pragma redefine_extnmae myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(arg1, arg2)
    long *arg1;
    int *arg2;
{
    extern int __fixed_myroutine();
    return (__fixed_myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#endif /* __STDC__ */
```

2.8.22 returns_new_memory

`#pragma returns_new_memory (`*funcname[, funcname]*`)`

该 pragma 断言指定函数的返回值在调用点上不使用任何内存作为别名。实际上，该调用返回一个新存储单元。该信息使优化器更好地跟踪指针值并澄清存储单元。这将导致循环的调度、流水线化和并行化的改进。然而，如果断言为假，则程序的行为未定义。

只有在使用原型或空参数列表声明指定的函数之后才允许使用该 pragma，如以下示例所示：

```
void *malloc(unsigned);
#pragma returns_new_memory(malloc)
```

2.8.23 unknown_control_flow

`#pragma unknown_control_flow (`*funcname[, funcname]*`)`

为了描述改变调用程序的流程图的过程， C 编译器提供 `#pragma unknown_control_flow` 指令。通常，该指令带有 `setjmp()` 之类的函数声明。在 Sun 系统上，包含文件 `<setjmp.h>` 包含以下内容：

```
extern int setjmp();
#pragma unknown_control_flow(setjmp)
```

同样，必须声明具有 `setjmp()` 类似属性的其它函数。

原则上，识别该属性的优化器应在控制流程图中插入相应的界限，从而在调用 `setjmp()` 的函数中安全地处理函数调用，同时保持优化流程图的未受影响部分的代码的能力。

必须在该 pragma 之前使用原型或空参数列表声明指定的函数。

2.8.24 unroll

(SPARC) `#pragma unroll (`*unroll_factor*`)`

对于参数 *unroll_factor*，该 pragma 接受正整数常量值。该 pragma 应用于当前块内部的下一个 `for` 循环。当解开因子不为 1 时，该指令作为对编译器的建议，指出指定的循环应由给定的因子解开。如果可能，编译器将使用该解开因子。如果解开因子值为 1，该指令作为一个命令，向编译器指出不解开该循环。编译器在优化级别 3 或更高级别上使用此信息。

2.8.25 weak

`#pragma weak` *symbol1 [= symbol2]*

定义弱全局符号。该 pragma 主要用在生成库的源文件中。如果无法解析弱符号，链接程序不会生成错误消息。

```
#pragma weak symbol
```

将 *symbol* 定义为弱符号。如果链接程序找不到 *symbol* 的定义，它不会生成错误消息。

```
#pragma weak symbol1 = symbol2
```

将 *symbol1* 定义为弱符号，它是符号 *symbol2* 的别名。这种形式的 pragma 只能用于其中已定义 *symbol2* 的同一转换单元，并且在源文件中或者在其一个包含的头文件中。否则，将导致编译错误。

如果程序调用但未定义 *symbol1*，并且 *symbol1* 在所链接的库中是弱符号，则链接程序使用该库中的定义。但是，如果程序定义自己的 *symbol1* 版本，则使用该程序的定义，而不使用库中 *symbol1* 的弱全局定义。如果程序直接调用 *symbol2*，则使用库中的定义； *symbol2* 的重复定义会导致错误。

2.9 预定义名称

以下标识符预定义为类似对象的宏：

表 2-3 预定义标识符

标识符	描述
__STDC__	__STDC__ 1 -Xc __STDC__ 0 -Xa, -Xt 未定义 -Xs

如果 __STDC__ 未定义 (#undef __STDC__)，编译器将发出警告。__STDC__ 在 -Xs 模式下未定义。

预定义 （在 -Xc 模式下无效）：

- sun
- unix
- sparc *(SPARC)*
- i386 *(Intel)*

下列预定义在所有模式下均有效：

- __sun
- __unix
- __SUNPRO_C=0x550
- __`name -s`_`name -r` （示例：__SunOS_5_7）
- __sparc *(SPARC)*
- __i386 *(Intel)*
- __BUILTIN_VA_ARG_INCR
- __SVR4
- __sparcv9 (-Xarch=v9, v9a)

编译器还预定义类似对象的宏 __PRAGMA_REDEFINE_EXTNAME，表示将识别该 pragma。以下标识符只在 -Xa 和 -Xt 模式下预定义：

__RESTRICT

2.10 _Restrict 关键字

C 编译器支持 _Restrict 关键字，该关键字与 C99 标准中的 restrict 关键字等价。_Restrict 关键字可与 -xc99=%none 和 -xc99=%all 一起使用，而 restrict 关键字只能与 -xc99=%all 一起使用。

有关支持的 C99 功能的详细信息，请参见附录 D。

2.11 _ _asm 关键字

_ _asm 关键字 （注意开头的两个下划线字符）是 asm 关键字的同义字。如果您使用 asm 而不是 _ _asm，并且在 -Xc 模式下编译，编译器会发出警告。如果您在 -Xc 模式下使用 _ _asm，编译器不会发出警告。 _ _asm 语句为以下形式：

```
_ _asm("string");
```

其中 *string* 是有效的汇编语言语句。 _ _asm 语句必须在函数主体内部出现。

2.12 环境变量

本节列出用于控制编译和运行环境的环境变量。

2.12.1 OMP_DYNAMIC

启用或禁止线程数的动态调整。

2.12.2 OMP_NESTED

启用或禁止嵌套并行性。

2.12.3 OMP_NUM_THREADS

设置执行过程中要使用的线程数。

2.12.4 OMP_SCHEDULE

设置运行时调度类型和块大小。

2.12.5 PARALLEL

(SPARC) 指定可供程序进行多处理器执行的处理器数。如果目标机器具有多个处理器，线程可以映射到独立的处理器。运行程序将创建执行程序的并行化部分的两个线程。

2.12.6 SUN_PROFDATA

控制 `-xprofile=collect` 命令在其中存储执行频率数据的文件的名称。

2.12.7 SUN_PROFDATA_DIR

控制 `-xprofile=collect` 命令将执行频率数据文件放在哪个目录中。

2.12.8 SUNPRO_SB_INIT_FILE_NAME

包含 `.sbinit`(5) 文件的目录的绝对路径名。只有在使用 `-xsb` 或 `-xsbfast` 标记时，才能使用该变量。

2.12.9 SUNW_MP_THR_IDLE

控制每个帮助程序线程的任务结束状态，可设置为 `spin` *n*`s` 或 `sleep` *n*`ms`。缺省值为 `spin`。有关详情，请参见《*OpenMP API 用户指南*》。

2.12.10 TMPDIR

cc 通常在目录 /tmp 中创建临时文件。您可以通过将环境变量 TMPDIR 设置为您选定的目录来指定另一个目录。但是，如果 TMPDIR 不是有效目录， cc 将使用 tmp。-xtemp 选项优先于 TMPDIR 环境变量。

如果您使用 Bourne shell，请键入：

```
$ TMPDIR=dir; export TMPDIR
```

如果您使用 C shell，请键入：

```
% setenv TMPDIR dir
```

2.13 如何指定包含文件

要包含随 C 编译系统提供的任何标准头文件，请使用以下格式：

```
#include <stdio.h>
```

尖括号 (<>) 使预处理程序在系统上头文件的标准位置查找头文件，通常为 /usr/include 目录。

对于您已存储在您自己的目录中的头文件，格式不同：

```
#include "header.h"
```

对于 #include "foo.h" 形式的语句 （其中使用了引号），编译器按以下顺序查找包含文件：

1. 当前目录 （即包含 “包含” 文件的目录）
2. 目录以 -I 选项 （如果有）命名
3. /usr/include 目录

如果头文件所在的目录与包含头文件的源文件所在的目录不同，请指定使用 cc 及 -I 选项存储头文件的目录的路径。例如，假设在源文件 mycode.c 中已包含 stdio.h 和 header.h：

```
#include <stdio.h>
#include "header.h"
```

进一步假设 header.h 存储在目录 ../defs 中。命令：

```
% cc -I../defs mycode.c
```

指示预处理程序首先在包含 mycode.c 的目录中查找 header.h，然后在目录 ../defs 中查找，最后在标准位置查找。它还指示预处理程序首先在 ../defs 中查找 stdio.h，然后在标准位置查找。不同之处在于：仅对于其名称用引号括起的头文件，才查找当前目录。

您可以在 cc 命令行上多次指定 -I 选项。预处理程序按指定目录出现的顺序查找它们。您可以在同一个命令行上为 cc 指定多个选项：

```
% cc -o prog -I../defs mycode.c
```

2.13.1 使用 -I- 选项更改搜索算法

新的 -I- 选项提供对缺省搜索规则的更多控制。当命令行中出现 -I- 时：

- 除非在 -I 指令中显式列出当前目录，否则编译器决不会搜索当前目录。该效果甚至适用于 #include "foo.h" 形式的包含语句。
- 对于 #include "foo.h" 形式的包含语句，编译器按以下顺序查找包含文件：
  a. 目录以 -I 选项 （在 -I- 前后均可）命名。
  b. /usr/include 目录
- 对于 #include <foo.h> 形式的包含语句，编译器按以下顺序查找包含文件：
  a. 目录以 -I 命名，该选项在 -I- 后面出现（也就是说，编译器不搜索在 -I- 之前出现的 -I 目录）
  b. /usr/include 目录

以下示例显示在编译 prog.c 时使用 -I- 的结果。

```
prog.c          #include "a.h"
                #include <b.h>
                #include "c.h"

c.h             #ifndef _C_H_1
                #define _C_H_1
                int c1;
                #endif

int/a.h         #ifndef _A_H
                #define _A_H
                #include "c.h"
                int a;
                #endif

int/b.h         #ifndef _B_H
                #define _B_H
                #include <c.h>
                int b;
                #endif
int/c.h         #ifndef _C_H_2
                #define _C_H_2
                int c2;
                #endif
```

以下命令显示在当前目录（包含文件所在的目录）中查找 #include "foo.h" 形式的包含语句的缺省行为。当处理 inc/a.h 中的 #include "c.h" 语句时，预处理程序包含 inc 子目录中的头文件 c.h。当处理 prog.c 中的 #include "c.h" 语句时，预处理程序包含具有 prog.c 的目录中的 c.h 文件。请注意，-H 选项指示编译器打印被包含文件的路径。

```
example% cc -c -Iinc -H prog.c
inc/a.h
               inc/c.h
inc/b.h
               inc/c.h
c.h
```

下一个命令显示 -I- 选项的作用。当预处理程序处理 #include "foo.h" 形式的语句时，它并不首先在包含目录中查找。相反，它按目录在命令行中的出现顺序，搜索由 -I 选项命名的目录。当处理 inc/a.h 中的 #include "c.h" 语句时，预处理程序包含 ./c.h 头文件而不是 inc/c.h 头文件。

```
example% cc -c -I. -I- -Iinc -H prog.c
inc/a.h
              ./c.h
inc/b.h
              inc/c.h
./c.h
```

有关详细信息，请参见第 A-21 页上的第 A.3.31 节 “-I[-|dir]”。

第 3 章

并行化 Sun C 代码

Sun C 编译器可以优化代码，以便在 SPARC 共享内存多处理器机器上运行。该进程称为*并行化*。编译的代码可以使用系统上的多个处理器并行执行。本章解释如何利用编译器的并行化功能。

3.1 概述

C 编译器为那些它确定可以安全进行并行化的循环生成并行代码。通常，这些循环具有彼此独立的迭代。对于此类循环，迭代以什么顺序执行或者是否并行执行并不重要。虽然不是全部，但是许多向量循环都属于此种类。

由于 C 中使用别名的方式，难以确定并行化的安全。为帮助编译器， Sun C 提供了 pragma 和附加指针限定，以提供程序员知道但编译器无法确定的别名信息。有关详细信息，请参见第 6-1 页上的 “基于类型的别名分析”。

3.1.1 使用示例

以下示例说明了如何启用和控制并行化 C：

```
% cc -fast -xO4 -xautopar example.c -o example
```

这将生成一个称为 `example` 的可正常执行的可执行程序。如果要利用多处理器执行，请参见第 A-34 页上的第 A.3.65 节 “`-xautopar`”。

3.2 OpenMP 并行化

您可以编译代码，以便它使用 OpenMP 规范进行编译。有关针对 C 的 OpenMP 规范的详细信息，请访问 web 站点 `http://www.openmp.org/specs/`。

要利用编译器的 OpenMP 支持，您需要执行编译器的 `-xopenmp` 选项。参见第 A-61 页上的第 A.3.107 节 “-xopenmp[=i]”。

3.2.1 处理 OpenMP 运行时警告

OpenMP 运行时系统对于非致命错误会发出警告。使用以下函数登记一个回调函数以处理这些警告：

```
int sunw_mp_register_warn(void (*func) (void *) )
```

您可以通过对 `<sunw_mp_misc.h>` 发出 `#include` 预处理程序指令来访问该函数的原型。

如果您不想登记函数，请将环境变量 `SUNW_MP_WARN` 设置为 `TRUE`，警告消息将发送给 `stderr`。有关 `SUNW_MP_WARN` 的详细信息，请参见第 3-3 页上的 “SUNW_MP_WARN”。

有关 OpenMP 的该实现的详细信息，请参见附录 G。

3.3 环境变量

与并行化 C 相关的环境变量有四种：

- `PARALLEL`
- `SUNW_MP_THR_IDLE`
- `SUNW_MP_WARN`
- `STACKSIZE`

PARALLEL

如果您可以利用多处理器执行，请设置 `PARALLEL` 环境变量。`PARALLEL` 环境变量指定可供程序使用的处理器数。在以下示例中， `PARALLEL` 设置为 2：

```
% setenv PARALLEL 2
```

如果目标机器具有多个处理器，线程可以映射到独立的处理器。运行该程序将导致创建执行程序的并行化部分的两个线程。

SUNW_MP_THR_IDLE

当前，程序的起始线程创建绑定线程。绑定线程一旦创建，将会参与执行程序的并行部分 （并行循环、并行区域等），并在程序的串行部分运行时保持自旋等待状态。在程序终止之前，这些绑定线程不会休眠或停止。并行化程序在专用系统上运行时，使这些线程保持自旋等待状态通常可达到最佳性能。然而，保持自旋等待的线程会占用系统资源。

使用 SUNW_MP_THR_IDLE 环境变量控制每个线程在完成其并行作业后的状态。

```
% setenv SUNW_MP_THR_IDLE value
```

您可以用 spin 或 sleep[*n* s | *n* ms] 替换 *value*。缺省值为 spin，表示线程在完成并行任务之后应自旋 （或忙等待），直到新的并行任务到来时为止。

另一个选项 sleep[*n* s|*n* ms] 在自旋等待 *n* 个单位之后使线程进入休眠状态。等待单位可以是秒 （s，缺省单位）或毫秒 (ms)，其中 1s 表示 1 秒， 10ms 表示 10 毫秒。不带参数的 sleep 在线程完成并行任务后使线程立即进入休眠状态。 sleep、 sleep0、 sleep0s 以及 sleep0ms 均等价。

如果新作业在未达到 *n* 个单位之前到达，线程将停止自旋等待并开始执行新作业。如果 SUNW_MP_THR_IDLE 包含非法值或未设置，则 spin 用作缺省值。

SUNW_MP_WARN

将此环境变量设置为 TRUE，以便从 OpenMP 或其它并行化运行时系统打印警告消息。

```
% setenv SUNW_MP_WARN TRUE
```

如果您通过使用 sunw_mp_register_warn() 登记某个函数以处理警告消息，则即使将 SUNW_MP_WARN 设置为 TRUE，它也不会打印警告消息。如果未登记函数，并且将 SUNW_MP_WARN 设置为 TRUE，则 SUNW_MP_WARN 将警告消息打印至 stderr。如果您未登记函数，并且未设置 SUNW_MP_WARN，则不会发出警告消息。有关 sunw_mp_register_warn() 的详细信息，请参见第 3-2 页上的第 3.2.1 节 “处理 OpenMP 运行时警告”。

STACKSIZE

正在执行的程序将会为主线程保留一个主内存栈，同时为每个从属线程保留不同栈。栈是临时内存地址空间，用来存储子程序调用中的参数和自动变量。

主栈的缺省大小约为 8 兆字节。使用 `limit` 命令显示当前主栈大小并设置它。

```
% limit
cputime 没有限制
filesize 没有限制
datasize 2097148 千字节
stacksize 8192 千字节 <- 当前主栈大小
coredumpsize 0 千字节
descriptors 256
memorysize 没有限制
% limit stacksize 65536 <- 将主栈设置为 64Mb
```

多线程程序的每个从属线程均具有其自身的线程栈。该栈与主线程的主栈相似，但对该线程是唯一的。该线程的私有数组和变量 （对该线程来说是局部数组和变量）位于线程栈中。

所有从属线程均具有相同的栈大小。缺省情况下，对于 32 位应用程序，栈大小为 4 兆字节；对于 64 位应用程序，栈大小为 8 兆字节。该大小使用环境变量 `STACKSIZE` 进行设置：

```
% setenv STACKSIZE 16483 <- 将线程栈大小设置为 16 Mb
```

对于某些并行化代码，可能有必要将线程栈大小设置为大于缺省值的值。

有时，编译器会生成一条警告消息，指出需要更大的栈大小。然而，除了通过尝试并出错之外，不可能知道应设置多大的栈大小正合适，尤其是涉及私有/局部数组时。如果栈大小太小而导致线程无法运行，程序将异常终止并出现段故障。

3.3.0.1 关键字

关键字 `restrict` 可以与并行化 C 配合使用。正确使用关键字 restrict 有助于优化器了解所需数据的别名，从而确定代码序列是否可以并行化。有关详情，请参见第 D-7 页上的 “C99 关键字”。

3.4 数据依赖性和干扰

C 编译器通过分析程序中的循环来确定并行执行循环的不同迭代是否安全。分析的目的是确定循环的两次迭代之间是否会相互干扰。通常，如果变量的一次迭代读取某个变量而另一次迭代正在写入该变量，会发生干扰。考虑以下程序片段：

编码示例 3-1 带依赖性的循环

```
for (i=1; i < 1000; i++) {
       sum = sum + a[i]; /* S1 */
}
```

在编码示例 3-1 中，任意两次连续迭代 `i` 和 `i+1` 将写入和读取同一变量 `sum`。因此，为了并行执行这两次迭代，需要以某种形式锁定该变量。否则，允许并行执行这两次迭代不安全。

然而，使用锁定会产生可能降低程序运行速度的开销。 C 编译器通常并不会并行化编码示例 3-1 中所示循环。在编码示例 3-1 中，循环的两次迭代之间存在数据依赖性。考虑另一个示例：

编码示例 3-2 不带依赖性的循环

```
for (i=1; i < 1000; i++) {
       a[i] = 2 * a[i]; /* S1 */
}
```

在此情况下，循环的每次迭代均引用不同的数组元素。因此，循环的不同迭代可以按任意顺序执行。由于不同迭代的两个数据元素不可能相互干扰，因此它们可以并行执行而无需任何锁定。

编译器为确定一个循环的两次不同迭代是否引用相同变量而执行的分析称为数据依赖性分析。如果其中一个引用写入变量，数据依赖性阻止循环并行化。编译器执行的数据依赖性分析有三种结果：

- 存在依赖性。在此情况下，并行执行循环不安全。编码示例 3-1 说明了此情况。
- 不存在依赖性。循环可使用任意多个进程安全地并行执行。编码示例 3-2 说明了此情况。
- 无法确定依赖性。为安全起见，编译器假定存在阻止并行执行循环的依赖性，并且不会并行化循环。

在编码示例 3-3 中，循环的两次迭代是否写入数组 a 的同一元素取决于数组 b 是否包含重复元素。除非编译器可以确定实际情况，否则它假定存在依赖性并且不会并行化循环。

编码示例 3-3 可能包含也可能不包含依赖性的循环

```
for (i=1; i < 1000; i++) {
       a[b[i]] = 2 * a[i];
}
```

3.4.1 并行执行模型

循环的并行执行由 Solaris 线程完成。启动程序的初始执行的线程称为主线程。程序启动时，主线程创建多个从属线程，如下图所示。程序结束时，所有从属线程均终止。从属线程的创建只进行一次，以使开销减至最小。

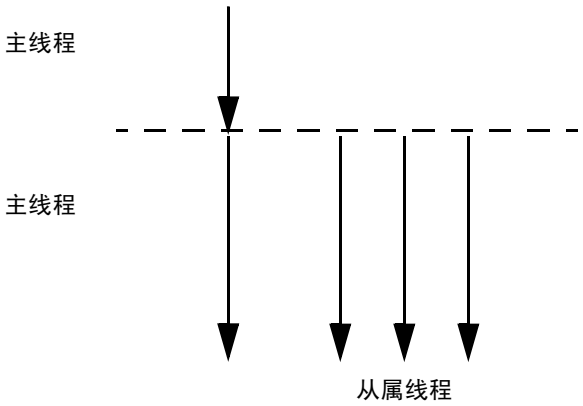

图 3-1 主线程和从属线程

程序启动后，主线程开始执行程序，而从属线程保持空闲等待状态。当主线程遇到并行循环时，循环的不同迭代将会在启动循环执行的从属线程和主线程之间分布。在每个线程完成其块的执行之后，将与剩余线程保持同步。此同步点称为*障碍*。在所有线程完成其工作并到达障碍之前，主线程不能继续执行程序的剩余部分。从属线程在到达障碍之后进入等待状态，等待分配更多的并行工作，而主线程继续执行该程序。

在此期间，可发生多种开销：

- 同步和工作分配的开销
- 障碍同步的开销

通常存在某些特殊的并行循环，为其执行的有用工作量不足以证明开销是值得的。对于此类循环，其运行速度会明显减慢。在下图中，循环是并行化的。然而，障碍（以水平条表示）带来大量开销。如图所示，障碍之间的工作串行或并行执行。并行执行循环所需的时间比主线程和从属线程在障碍处同步所需的时间少得多。

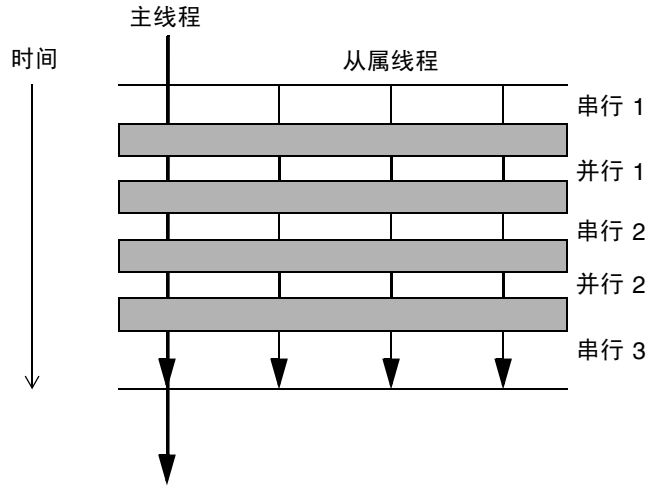

图 3-2 循环的并行执行

3.4.2 私有标量和私有数组

对于某些数据依赖性，编译器仍能够并行化循环。考虑以下示例。

编码示例 3-4 带依赖性的可并行化循环

```
for (i=1; i < 1000; i++) {
       t = 2 * a[i];            /* S1 */
       b[i] = t;                /* S2 */
}
```

在本示例中，假定数组 a 和 b 为非重叠数组，而由于变量 t 的存在而使两次迭代之间存在数据依赖性。在第一次迭代和第二次迭代时执行以下语句。

编码示例 3-5 第一次迭代和第二次迭代

```
t = 2*a[1];  /* 1 */
b[1] = t;    /* 2 */
t = 2*a[2];  /* 3 */
b[2] = t;    /* 4 */
```

由于第一个语句和第三个语句修改变量 t，因此编译器无法并行执行它们。然而， t 的值始终在同一次迭代中计算并使用，因此编译器可以对每次迭代使用 t 的单独一个副本。这消除了不同迭代之间由于此类变量而产生的干扰。事实上，我们已使变量 t 成为执行迭代的每个线程的私有变量。这种情形可以说明如下：

编码示例 3-6 变量 t 作为每个线程的私有变量

```
for (i=1; i < 1000; i++) {
       pt[i] = 2 * a[i];        /* S1 */
       b[i] = pt[i];            /* S2 */
}
```

编码示例 3-6 与编码示例 3-3 基本相同，但每个标量变量引用 t 现在被替换为数组引用 pt。现在，每次迭代使用 pt 的不同元素，因此消除了任意两次迭代之间的数据依赖性。当然，本示例产生的一个问题是可能导致数组非常大。在实际运用中，编译器为参与循环执行的每个线程只分配变量的一个副本。事实上，每个此类变量是线程的私有变量。

编译器还可以私有化数组变量，以便为循环的并行执行创造机会。考虑以下示例：

编码示例 3-7 带数组变量的可并行化循环

```
for (i=1; i < 1000; i++) {
       for (j=1; j < 1000; j++) {
              x[j] = 2 * a[i];         /* S1 */
              b[i][j] = x[j];          /* S2 */
       }
}
```

在编码示例 3-7 中，外部循环的不同迭代修改数组 x 的相同元素，因此外部循环不能并行化。然而，如果执行外部循环迭代的每个线程均具有整个数组 x 的私有副本，则外部循环的任意两次迭代之间不存在干扰。这种情形说明如下：

编码示例 3-8 使用私有化数组的可并行化循环

```
for (i=1; i < 1000; i++) {
       for (j=1; j < 1000; j++) {
              px[i][j] = 2 * a[i];     /* S1 */
              b[i][j] = px[i][j];      /* S2 */
       }
}
```

如私有标量的情形一样，不必要为所有迭代展开数组，而只需要达到系统中执行的线程数。这由编译器自动完成，方式是在每个线程的私有空间中分配初始数组的一个副本。

3.4.3 返回存储

变量私有化对改进程序中的并行性十分有用。然而，如果在循环外部引用私有变量，则编译器需要确保私有变量具有正确的值。考虑以下示例：

编码示例 3-9 使用返回存储的并行化循环

```
for (i=1; i < 1000; i++) {
       t = 2 * a[i];            /* S1 */
       b[i] = t;                /* S2 */
}
x = t;                         /* S3 */
```

在编码示例 3-9 中，在语句 S3 中引用的 t 值是循环计算的最终 t 值。在变量 t 私有化并且循环完成执行之前， t 的正确值需要重新存储到初始变量中。这称为返回存储。这通过将最后一次迭代中的 t 值重新复制到变量 t 的初始位置来完成。在很多情况下，编译器自动执行此操作。但是也存在不易计算最终值的情况：

编码示例 3-10 不能使用返回存储的循环

```
for (i=1; i < 1000; i++) {
       if (c[i] > x[i] ) {          /* C1 */
              t = 2 * a[i];            /* S1 */
              b[i] = t;                /* S2 */
       }
}
x = t*t;                            /* S3 */
```

正确执行后，语句 S3 中的 t 值通常并不是循环最终迭代中的 t 值。事实上，它是条件 C1 为真时的最后一次迭代。通常，计算 t 的最终值十分困难。在类似情况下，编译器不会并行化循环。

3.4.4 约简变量

有时循环的迭代之间存在真正的依赖性，而导致依赖性的变量并不能简单地私有化。例如，从一次迭代到下一次迭代累计值时，会出现这种情况。

编码示例 3-11 可以或不可以并行化的循环

```
for (i=1; i < 1000; i++) {
       sum += a[i]*b[i]; /* S1 */
}
```

在编码示例 3-11 中，循环计算两个数组的向量乘积，并将结果赋给一个称为 sum 的公共变量。该循环不能以简单的方式并行化。编译器可以利用语句 S1 中的计算的关联特性，并为每个线程分配一个称为 psum[i] 的私有变量。变量 psum[i] 的每个副本均初始化为 0。每个线程以自己的变量 psum[i] 副本计算自己的部分和。在穿过障碍之前，所有部分和均加到初始变量 sum 上。在本示例中，变量 sum 称为约简变量，因为它计算和约简。然而，将标量变量提升为约简变量存在的危险是：累计舍入值的方式会更改 sum 的最终值。只有在您专门授权这样做时，编译器才执行该变换。

3.5 加速

如果编译器不并行化所花时间量占主体的程序部分，则不会发生加速。这基本上是 Amdahl 定律的推论。例如，如果并行化一个占用程序执行时间的百分之五的循环，则总加速仅限于百分之五。然而，根据工作量和并行执行开销的大小，可能没有任何改善。

一般说来，并行化的程序执行所占的比例越大，加速的可能性就越大。

每个并行循环均会在启动和关闭期间发生少量开销。启动开销包括工作分配代价，关闭开销包括障碍同步代价。如果循环执行的工作总量不足够大，则不会发生加速。事实上，循环甚至可能减慢。因此，如果大量程序执行工作由许多短并行循环完成，则整个程序可能减慢而不是加速。

编译器执行几个试图增大循环粒度的循环变换。其中某些变换是循环交换和循环合并。因此，一般说来，如果程序中的并行量很小或者分散在小并行区域，则加速很少。

通常，按比例增大问题大小可提高程序中的并行程度。例如，考虑包含以下两部分的问题：按顺序的二次部分，以及可并行化的三次部分。对于此问题，工作量的并行部分的增长速度比顺序部分的增长速度快。因此在某些点，除非达到资源限制，否则问题可以大大加速。

尝试进行某些调节，使用指令、问题大小进行试验，重新构造程序，以便从并行 C 中获益。

3.5.1 Amdahl 定律

固定问题大小加速通常遵守 Amdahl 定律。 Amdahl 定律说明，给定问题中的并行加速量受问题的顺序部分的限制。以下方程式描述问题的加速，其中 F 是在顺序区域花费的时间，剩余时间均匀地分摊在 *P* 个处理器之间。如果方程式的第二项减小到零，则总加速受第一项约束，保持固定。

$$\frac{1}{S} = F + \frac{(1-F)}{P}$$

下图以图表方式说明此概念。深色阴影部分表示程序的顺序部分，并且对于 1、2、4、8 个处理器均保持不变。浅色阴影部分代表程序的并行部分，可均匀地分摊在任意多个处理器之间。

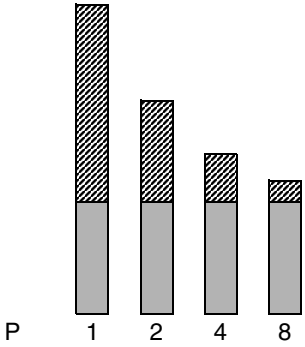

图 3-3 固定问题加速

然而，实际上可能会发生由于通信以及向多个处理器分配工作而产生的开销。对于使用的任意多个处理器，这些开销可能固定，也可能不固定。

图 3-4 说明一个包含 0%、 2%、 5% 和 10% 顺序部分的程序的理想加速。此处假定无开销。

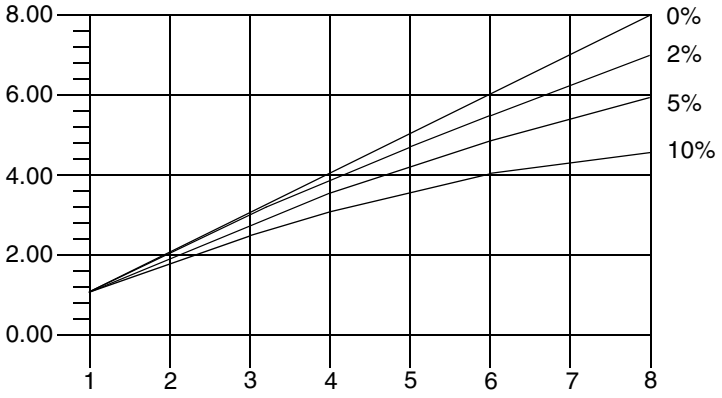

图 3-4 Amdahl 定律加速曲线

3.5.1.1 开销

一旦将开销加入此模型中，加速曲线会发生很大的变化。为了方便说明，我们假定开销包括以下两部分：独立于处理器数的固定部分，以及随使用的处理器数呈二次增长的非固定部分：

$$\frac{1}{S} = \frac{1}{F + \left(1 - \frac{F}{P}\right) + K_1 + K_2 P^2}$$

在此方程式中， K_1 和 K_2 是固定因子。在这些假定下，加速曲线如下图所示。值得注意的是，在此情况下，加速达到峰值。在某个点之后，增加更多处理器会降低性能，如下图所示。

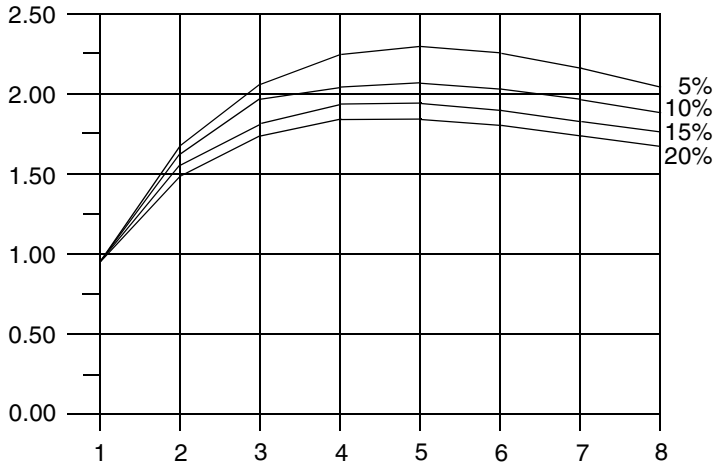

图 3-5 带开销的加速曲线

3.5.1.2 Gustafson 定律

Amdahl 定律可能会在预测真实问题的并行加速时造成误导。花费在程序的顺序部分的时间有时取决于问题大小。也就是说，通过按比例缩放问题大小，您可以获得更多加速机会。以下示例说明了这一点。

编码示例 3-12 按比例缩放问题大小可能会获得更多加速机会

```
/*
* 初始化数组
*/
for (i=0; i < n; i++) {
       for (j=0; j < n; j++) {
              a[i][j] = 0.0;
              b[i][j] = ...
              c[i][j] = ...
       }
}
/*
* 矩阵乘法
*/
for (i=0; i < n; i++) {
       for(j=0; j < n; j++) {
              for (k=0; k < n; k++) {
                  a[i][j] = b[i][k]*c[k][j];
              }
       }
}
```

假定理想的开销为零，并假定只有第二个循环嵌套并行执行。不难发现，对于较小的问题大小 （即 n 的值较小），程序的顺序部分和并行部分所用的时间彼此相差并不大。然而，随着 n 的增大，花费在程序并行部分的时间比花费在顺序部分的时间增长快。对于此问题，随问题大小的增大而增加处理器数很有益。

3.6 负载平衡和循环调度

循环调度是将并行循环的迭代分布到多个线程的进程。为获得最大加速，在线程间均匀地分布工作而不产生太大开销十分重要。编译器针对不同情况提供多种调度类型。

3.6.1 静态调度或块调度

当循环的不同迭代执行的工作相同时，将工作均匀地分摊在系统上不同的线程之间很有益。此方法称为静态调度。

编码示例 3-13 静态调度的良好循环

```
for (i=1; i < 1000; i++) {
       sum += a[i]*b[i];        /* S1 */
}
```

在静态调度或块调度中，每个线程将获取相同次数的迭代。如果有 4 个线程，则在以上示例中，每个线程将获取 250 次迭代。假设不存在中断，并且每个线程的进度相同，则所有线程将同时完成。

3.6.2 自我调度

一般说来，当每次迭代执行的工作不同时，静态调度不会达到良好的负载平衡。在静态调度中，每个线程获取同一迭代块。除主线程之外，每个线程在完成自己的块时，将等待参与下一个并行循环的执行。主线程将继续执行程序。在自我调度中，每个线程获取不同的小迭代块，并且在完成为其分配的块之后，尝试从同一循环中获取更多块。

3.6.3 引导自我调度

在引导自我调度 (GSS) 中，每个线程获取连续变少的块。如果每次迭代的大小不同，GSS 有助于平衡负载。

3.7 循环变换

编译器执行多个循环重构变换，帮助改进程序中循环的并行化。其中某些变换还可以提高循环的单处理器执行性能。下面描述编译器执行的变换。

3.7.1 循环分布

循环通常包含少量无法并行执行的语句以及许多可以并行执行的语句。循环分布旨在将顺序语句移到单独一个循环中，并将可并行化的语句收集到另一个循环中。这在以下示例中得以说明：

编码示例 3-14 循环分布举例

```
for (i=0; i < n; i++) {
       x[i] = y[i] + z[i]*w[i];                 /* S1 */
       a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;    /* S2 */
       y[i] = z[i] - x[i];                      /* S3 */
}
```

假定数组 x、y、w、a 和 z 不重叠，则语句 S1 和 S3 可以并行化，但语句 S2 不能并行化。以下是循环被分割或分布为两个不同循环后的情形：

编码示例 3-15 分布式循环

```
/* L1: 并行循环 */
for (i=0; i < n; i++) {
       x[i] = y[i] + z[i]*w[i];                 /* S1 */
       y[i] = z[i] - x[i];                      /* S3 */
}
/* L2: 顺序循环 */
for (i=0; i < n; i++) {
       a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

在此变换之后，循环 L1 不包含任何阻止循环并行化的语句，因此可并行执行。然而，循环 L2 仍包含来自初始循环的不可并行化的语句。

循环分布并非始终有益或安全。编译器会进行分析，以确定分布的安全性和有益性。

3.7.2 循环合并

如果循环的粒度（或循环执行的工作量）很小，则分布的效果可能并不明显。这是因为与循环工作量相比，并行循环启动的开销太大。在这种情况下，编译器使用循环合并将多个循环合并到单个并行循环中，从而增大循环的粒度。当具有相同行程计数的循环彼此相邻时，循环合并很方便且很安全。考虑以下示例：

编码示例 3-16 工作量小的循环

```
/* L1: 短并行循环 */
for (i=0; i < 100; i++) {
       a[i] = a[i] + b[i];         /* S1 */
}
/* L2: 另一个短并行循环 */
for (i=0; i < 100; i++) {
       b[i] = a[i] * d[i];         /* S2 */
}
```

这两个短并行循环彼此相邻，可以安全地合并，如下所示：

编码示例 3-17 合并的两个循环

```
/* L3: 较大的并行循环 */
for (i=0; i < 100; i++) {
       a[i] = a[i] + b[i];        /* S1 */
       b[i] = a[i] * d[i];        /* S2 */
}
```

新循环产生的开销是并行循环执行产生的开销的一半。循环合并在其它方面也很有帮助。例如，如果在两个循环中引用同一数据，则合并这两个循环可以改善引用环境。

然而，循环合并并非始终安全。如果循环合并创建之前并不存在的数据依赖性，则合并可能会导致错误执行。考虑以下示例：

编码示例 3-18 不安全合并举例

```
/* L1: 短并行循环 */
for (i=0; i < 100; i++) {
       a[i] = a[i] + b[i];       /* S1 */
}
/* L2: 具有数据依赖性的短循环 */
for (i=0; i < 100; i++) {
       a[i+1] = a[i] * d[i];     /* S2 */
}
```

如果合并编码示例 3-18 中的循环，则产生语句 S2 至 S1 的数据依赖性。实际上，语句 S1 右边 a[i] 的值在语句 S2 中计算。如果不合并循环，此情况则不会发生。编译器执行安全性和有益性分析，以确定是否应执行循环合并。通常，编译器可以合并任意多个循环。以这种方式增大粒度有时可以大大改善循环，使其足从并行化中获益。

3.7.3 循环交换

并行化循环嵌套的最外层循环通常更有益，因为发生的开销很小。然而，由于此类循环可能携带依赖性，并行化最外层循环并非始终安全。以下对此进行说明：

编码示例 3-19 不能并行化的嵌套循环

```
for (i=0; i <n; i++) {
       for (j=0; j <n; j++) {
              a[j][i+1] = 2.0*a[j][i-1];
       }
}
```

在本示例中，具有索引变量 i 的循环不能并行化，原因是循环的两次连续迭代之间存在依赖性。这两个循环可以交换，并行循环 （j-循环）变为外部循环：

编码示例 3-20 交换的循环

```
for (j=0; j<n; j++) {
       for (i=0; i<n; i++) {
              a[j][i+1] = 2.0*a[j][i-1];
       }
}
```

交换后的循环只发生一次并行工作分配开销，而先前发生 n 次开销。编译器执行安全性和有益性分析，以确定是否执行循环交换。

3.8 别名和并行化

ISO C 别名通常可防止循环并行化。存在两个对同一存储单元的可能引用时，将产生别名。考虑以下示例：

编码示例 3-21 具有对同一存储单元的两个引用的循环

```
void copy(float a[], float b[], int n) {
       int i;
       for (i=0; i < n; i++) {
              a[i] = b[i]; /* S1 */
       }
}
```

由于变量 a 和 b 为参数，因此 a 和 b 可能指向重叠的内存区域；例如，如果 copy 的调用如下：

```
copy (x[10], x[11], 20);
```

在调用的例程中， copy 循环的两次连续迭代可能读/写数组 x 的同一元素。然而，如果例程 copy 的调用如下，则循环的 20 次迭代中不可能出现重叠：

```
copy (x[10], x[40], 20);
```

通常，在不知道例程如何调用的情况下，编译器不可能正确地分析此情况。编译器提供 ISO C 的关键字扩展，使您可以传达此类别名信息。有关详情，请参见第 3-19 页上的第 3.8.2 节 “限定指针”。

3.8.1 数组引用和指针引用

别名问题的部分原因是：C 语言可以通过指针运算来定义数组引用及定义。为使编译器有效地并行化循环 （自动或显式使用 pragma），所有采用数组布局的数据均必须使用 C 数组引用语法而不是指针进行引用。如果使用指针语法，编译器将无法确定循环的不同迭代之间的关系。因此，它将保守而不会并行化循环。

3.8.2 限定指针

为使编译器有效地执行循环的并行执行任务，需要确定特定左值是否指定不同的存储区域。别名是其存储区域相同的左值。由于需要分析整个程序，因此确定对象的两个指针是否为别名是一个困难而费时的过程。考虑下面的函数 vsq()：

编码示例 3-22 带两个指针的循环

```
void vsq(int n, double * a, double * b) {
       int i;
       for (i=0; i<n; i++) {
              b[i] = a[i] * a[i];
       }
}
```

如果编译器知道指针 a 和 b 访问不同的对象，它可以并行化循环不同迭代的执行。如果通过指针 a 和 b 访问的对象中存在重叠，则编译器并行执行循环将会不安全。在编译时，编译器并不能通过简单地分析 vsq() 来获悉 a 和 b 访问的对象是否重叠；编辑器需要分析整个程序才能获取此信息。

限定指针用来指定哪些指定不同对象的指针，以便编译器可以执行指针别名分析。以下是函数参数声明为限定指针的函数 vsq() 的示例：

```
void vsq(int n, double * restrict a, double * restrict b)
```

指针 a 和 b 声明为限定指针，因此编译器知道 a 和 b 指向不同的存储区域。有了此别名信息，编译器就能够并行化循环。

关键字 restrict 是类型限定符，类似于 volatile，并且仅限定指针类型。当您使用 -xc99=%all （不包括使用 -Xs 的情况）时， restrict 被识别为关键字。在某些情况下，您可能不希望更改源代码。您可以通过使用以下命令行选项，指定将指针赋值的函数参数视为限定指针：

```
-xrestrict=[func1,…,funcn]
```

如果指定函数列表，则指定的函数中的指针参数将被视为限定的；否则，整个 C 文件中的所有指针参数均被视为限定的。例如， -xrestrict=vsq 是限定带关键字 restrict 的函数 vsq() 的第一个示例中给定的指针 a 和 b。

正确使用 restrict 至关重要。如果指针被限定为限定指针而指向不同的对象，编译器会错误地并行化循环而导致未定义的行为。例如，假定函数 vsq() 的指针 a 和 b 指向重叠的对象，如 b[i] 和 a[i+1] 是同一对象。如果 a 和 b 未被声明为限定指针，循环将以串行执行。如果 a 和 b 被错误地限定为限定指针，则编译器会并行化循环的执行，但这不安全，原因是 b[i+1] 应在计算 b[i] 之后才会计算。

3.8.3 显式并行化和 Pragma

通常，编译器没有足够的信息来判断并行化的合法性或有益性。 Sun ISO C 支持 pragma，使程序员能够有效地并行化循环，否则编译器很难或不可能处理。

3.8.3.1 串行 Pragma

有两个串行 pragma，均适用于 `for` 循环：

- `#pragma MP serial_loop`
- `#pragma MP serial_loop_nested`

`#pragma MP serial_loop` pragma 向编译器指示：下一个 `for` 循环不自动并行化。

`#pragma MP serial_loop_nested` pragma 向编译器指示：下一个 `for` 循环以及该 `for` 循环的作用域内嵌套的任何 `for` 循环均不自动并行化。 `serial_loop_nested` pragma 的作用域不会超出它所应用的循环的作用域。

3.8.3.2 并行 Pragma

有一个并行 pragma：`#pragma MP taskloop` [*options*]。

`MP taskloop` pragma 可以根据需要带一个或多个以下参数。

- `maxcpus` *(number_of_processors)*
- `private` *(list_of_private_variables)*
- `shared` *(list_of_shared_variables)*
- `readonly` *(list_of_readonly_variables)*
- `storeback` *(list_of_storeback_variables)*
- `savelast`
- `reduction` *(list_of_reduction_variables)*
- `schedtype` *(scheduling_type)*

每个 `MP taskloop` pragma 只能指定一个选项；但是， pragma 是累积的，并应用于源代码中当前块内部遇到的下一个 `for` 循环：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop shared(a,b)
#pragma MP taskloop storeback(x)
```

这些选项在其所应用的 `for` 循环之前可能会多次出现。如果存在冲突选项，编译器将发出警告消息。

for *循环的嵌套*

MP taskloop 应用于当前块内部的下一个 for 循环。并行化 C 不存在并行化 for 循环的嵌套。

并行化的合格性

除非另有禁止，否则 MP taskloop pragma 建议编译器应并行化指定的 for 循环。

具有不规则控制流和未知循环迭代增量的任何 for 循环均不能进行并行化。例如，包含 setjmp、longjmp、exit、abort、return、goto、labels 和 break 的 for 循环均不能并行化。

特别重要的是，具有迭代间依赖性的 for 循环可以进行显式并行化。这意味着，如果为此类循环指定了一个 MP taskloop pragma，除非 for 循环被取消资格，否则编译器将完全遵循该 pragma。用户有责任确保此类显式并行化不会导致错误结果。

如果为 for 循环指定了 serial_loop 或 serial_loop_nested 以及 taskloop pragma，则将使用最后指定的 pragma。

考虑以下示例：

```
#pragma MP serial_loop_nested
    for (i=0; i<100; i++) {
   # pragma MP taskloop
      for (j=0; j<1000; j++) {
      ...
 }
}
```

i 循环将不并行化，但是 j 循环可能并行化。

处理器数

#pragma MP taskloop maxcpus (*number_of_processors*) 指定要用于此循环的处理器数 （如果可能）。

maxcpus 的值必须为正整数。如果 maxcpus 等于 1，则指定的循环将串行执行。（请注意，将 maxcpus 设置为 1 相当于指定 serial_loop pragma。）将比较 maxcpus 的值与 PARALLEL 环境变量的解释值，使用较小的值。在未指定环境变量 PARALLEL 的情况下，该 pragma 被解释为具有值 1。

如果为一个 for 循环指定了多个 maxcpus pragma，将使用最后指定的 pragma。

变量分类

循环中使用的变量可分类为 private、 shared、 reduction 或 readonly 变量。变量只能属于其中一种分类。通过显式 pragma 只能将变量分类为 reduction 或 readonly 变量。参见 #pragma MP taskloop reduction 和 #pragma MP taskloop readonly。通过显式 pragma 或以下缺省作用域规则可将变量分类为 private 或 shared 变量。

private *和* shared *变量的缺省作用域规则*

private 变量的值是处理 for 循环的某些迭代的每个处理器的私有值。换句话说，在 for 循环的一次迭代中赋给 private 变量的值不会传送给处理该 for 循环的其它迭代的其它处理器。而 shared 变量的当前值可处理 for 循环的迭代的所有处理器访问。处理循环迭代的一个处理器赋给 shared 变量的值可能会被处理循环的其它迭代的其它处理器识别。通过使用 #pragma MP taskloop 指令显式并行化并包含共享变量引用的循环，必须确保值的共享不会导致正确性问题 （如竞争情况）。对显式并行化的循环中的共享变量的更新和访问，编译器不提供同步。

在分析显式并行化的循环时，编译器使用以下 “缺省作用域规则”确定变量是 private 还是 shared：

- 如果变量未通过 pragma 显式分类，声明为指针或数组，并且仅在循环内部使用数组语法进行引用，则该变量缺省分类为 shared 变量。否则，将被分类为 private 变量。
- 循环索引变量始终被视为 private 变量和返回存储变量。

*强烈建议*将显式并行化的 for 循环中使用的所有变量显式分类为 shared、 private, reduction 或 readonly 变量之一，以避免使用 “缺省作用域规则”。

由于编译器对共享变量的访问不执行同步，因此在对包含数组引用等的循环使用 MP taskloop pragma 之前必须格外谨慎。如果此类显式并行化的循环中存在迭代间数据依赖性，则其并行执行会导致错误结果。编译器不一定能够检测到此类潜在问题情况并发出警告消息。无论如何，编译器不会禁止具有潜在共享变量问题的循环的显式并行化。

private *变量*

#pragma MP taskloop private (*list_of_private_variables*)

使用此 pragma 指定所有应视为此循环私有变量的变量。此循环中使用但未显式指定为 shared、 readonly 或 reduction 变量的所有其它变量，按缺省作用域规则的定义，为 shared 或 private。

private 变量的值是处理循环的某些迭代的每个处理器的私有值。换句话说，处理循环迭代的一个处理器赋给 private 变量的值不会传送给处理该循环的其它迭代的其它处理器。private 变量在循环的每次迭代开始时没有初始值，并且在循环的迭代内部第一次使用它之前，必须在该迭代内部设置为一个值。如果某个循环包含一个在设置之前使用其值的显式声明 private 变量，则执行包含该循环的程序将导致未定义的行为。

shared *变量*

#pragma MP taskloop shared (*list_of_shared_variables*)

使用此 pragma 指定所有应视为此循环的 shared 变量的变量。循环中使用的但未显式指定为 private、 readonly、 storeback 或 reduction 变量的所有其它变量，按缺省作用域规则的定义，为 shared 或 private。

shared 变量的当前值可被处理 for 循环的迭代的所有处理器访问。处理循环迭代的一个处理器赋给 shared 变量的值可能会被处理循环的其它迭代的其它处理器识别。

readonly *变量*

#pragma MP taskloop readonly *(list_of_readonly_variables)*

readonly 变量是一类特殊的共享变量，不在循环的任何迭代中进行修改。使用此 pragma 向编译器指示，它可以对处理循环迭代的每个处理器使用该变量值的单独副本。

storeback *变量*

#pragma MP taskloop storeback *(list_of_storeback_variables)*

使用此 pragma 指定所有应视为 storeback 变量的变量。

storeback 变量的值在循环中计算，并且计算的值在循环终止后使用。 storeback 变量的最后一个循环迭代值在循环终止后使用。当变量为私有变量时，此类变量可以很好地通过此指令显式声明为 storeback 变量，通过将变量显式声明为私有变量或通过使用缺省作用域规则均可。

请注意， storeback 变量的返回存储操作发生在显式并行化循环的最后一次迭代中，而无论该迭代是否更新 storeback 变量的值。换句话说，处理循环最后一次迭代的处理器可能并不是当前包含 storeback 变量的最后更新值的同一处理器。考虑以下示例：

```
#pragma MP taskloop private(x)
#pragma MP taskloop storeback(x)
   for (i=1; i <= n; i++) {
      if (...) {
          x=...
      }
}
   printf ("%d", x);
```

在以上示例中，通过 printf() 调用打印出的 storeback 变量的值可能与通过 i 循环的串行版本打印出的值不同，原因是在显式并行化情况下，处理循环最后一次迭代 （当 i==n 时）的处理器，即执行 x 的返回存储操作的处理器可能不是当前包含 x 的最后更新值的同一处理器。编译器将尝试发出警告消息，提醒用户注意此类潜在问题。

在显式并行化的循环中，作为数组引用的变量不视为 storeback 变量。因此，如果需要此类返回存储操作 （例如，如果作为数组引用的变量已声明为私有变量），将作为数组引用的变量包括在 *list_of_storeback_variables* 中很重要。

savelast

#pragma MP taskloop savelast

使用此 pragma 指定要视为返回存储变量的所有私有变量。此 pragma 的语法如下：

#pragma MP taskloop savelast

通常，使用这种形式很方便，无需在将每个变量声明为返回存储变量时列出循环的每个私有变量。

reduction *变量*

#pragma MP taskloop reduction *(list_of_reduction_variables)* 指定出现在约简列表中的所有变量均将视为循环的 reduction 变量。 reduction 变量的部分值可由处理循环迭代的每个处理器单独计算，其最终值可从其所有部分值中计算。 reduction 变量列表的存在便于编译器识别循环是否为约简循环，从而允许为其生成并行约简代码。考虑以下示例：

```
#pragma MP taskloop reduction(x)
    for (i=0; i<n; i++) {
        x = x + a[i];
}
```

变量 *x* 是 (sum) 约简变量， i 循环是 (sum) 约简循环。

调度控制

Sun ISO C 编译器支持多种 pragma，这些 pragma 可与 taskloop pragma 配合使用，以控制给定循环的循环调度策略。此 pragma 的语法是：

#pragma MP taskloop schedtype *(scheduling_type)*

此 pragma 可用来指定要用来调度并行化循环的特定 *scheduling_type*。 *Scheduling_type* 可为以下类型之一：

- static

  在 static 调度中，循环的所有迭代均匀地分布在参与处理的所有处理器之间。考虑以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(static)
    for (i=0; i<1000; i++) {
...
}
```

在以上示例中，四个处理器中的每个处理器将处理循环的 250 次迭代。

- `self` [*(chunk_size)*]

  在 `self` 调度中，每个参与处理的处理器处理固定次数的迭代 （称为 “块大小”），直到循环的所有迭代均已处理时为止。可选的 *chunk_size* 参数指定要使用的 “块大小”。 *Chunk_size* 必须为正整数常量或整型变量。如果指定为变量， *chunk_size* 在循环开始时求出的值必须为正整数值。如果未指定这个可选的参数，或者其值不为正数，编译器将选择要使用的块大小。考虑以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(self(120))
for (i=0; i<1000; i++) {
...
}
```

  在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

  120, 120, 120, 120, 120, 120, 120, 120, 40。

- `gss` [(*min_chunk_size*)]

  在 `guided self` 调度中，每个参与处理的处理器处理可变次数的迭代 （称为 “最小块大小”），直到循环的所有迭代均已处理时为止。可选的 *min_chunk_size* 参数指定使用的每个可变块大小至少必须为 *min_chunk_size*。 *Min_chunk_size* 必须为正整数常量或整型变量。如果指定为变量， *min_chunk_size* 在循环开始时求出的值必须为正整数值。如果未指定这个可选的参数，或者其值不为正数，编译器将选择要使用的块大小。考虑以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(gss(10))
for (i=0; i<1000; i++) {
...
}
```

  在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

  250, 188, 141, 106, 79, 59, 45, 33, 25, 19, 14, 11, 10, 10, 10。

- `factoring` [(*min_chunk_size*)]

  在 `factoring` 调度中，每个参与处理的处理器处理可变次数的迭代（称为“最小块大小”），直到循环的所有迭代均已处理时为止。可选的 *min_chunk_size* 参数指定使用的每个可变块大小至少必须为 *min_chunk_size*。 *Min_chunk_size* 必须为正整数常量或整型变量。如果指定为变量， *min_chunk_size* 在循环开始时求出的值必须为正整数值。如果未指定这个可选的参数，或者其值不为正数，编译器将选择要使用的块大小。考虑以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(factoring(10))
for (i=0; i<1000; i++) {
...
}
```

在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 10, 10, 10, 10, 10, 10

第 4 章

递增链接编辑器 (ild)

本章描述 ild、 ild 特有功能、示例消息以及 ild 选项。

4.1 介绍

ild 是链接编辑器 ld 的递增版本，并取代用于链接程序的 ld。使用 ild 可有效地并更快地完成编辑、编译、链接和调试循环。您可以使用 dbx 的*修复并继续*功能完全避免重新链接，该功能使您可以无需重新链接而继续工作。但是，如果您需要重新链接，使用 ild 时该进程更快。有关修复并继续的详细信息，请参见《*使用 dbx 调试程序*》第 11 章。

ild 递增地进行链接，以便您可以将修改的目标代码插入您早先创建的可执行文件中，而无需重新链接未修改的目标文件。重新链接需要的时间取决于修改的代码量。每次生成时链接应用程序需要的时间并不相同。代码更改得越少，重新链接速度就越快。

在初始链接时， ild 需要的时间与 ld 需要的时间大大致相同，但是后续 ild 链接比 ld 链接快得多。减少链接时间的代价是增大可执行文件的大小。

4.2 递增链接概述

若使用 ild 代替 ld，进行初始链接时，各种文本、数据、 bss、异常表段等将用附加空间予以填充以便于未来的扩展 （参见图 4-1）。此外，所有重定位记录和全局符号表均保存到可执行文件中新的持久性状态区域中。执行后续递增链接时， ild 使用时间标记类来确定已更改的目标文件，并将已更改的目标代码修补为先前生成的可执行代码。也就是说，目标文件的以前版本失效，新目标文件装入到空出的空间，或者需要时装入到可执行文件的填充段。对失效目标文件中的符号的所有引用均修补为指向正确的新目标文件。

ild 并不支持所有 ld 命令选项。如果向 ild 传递它不支持的命令选项， ild 将直接调用 /usr/ccs/bin/ld 以执行链接。有关递增链接程序不支持的命令的详细信息，请参见第 4-16 页上的第 4.9 节 “ild 不支持的 ld 选项”。

4.3 如何使用 ild

在某些条件下，编译系统自动调用 ild 以取代 ld。当您调用编译系统时，您正在调用编译器驱动程序。当您将某些选项传递给驱动程序时，驱动程序使用 ild。编译器驱动程序从命令行读取选项，并按正确的顺序执行各种程序，而且从传递的参数的列表中增加文件。

例如， cc 首先运行 acomp（编译器的前端），然后 acomp 运行优化代码生成器，然后 cc 对命令行上列出的其它源代码执行这些操作。驱动程序即可生成一个对 ild 或 ld （取决于选项）的调用，从而将编译的所有文件以及使程序完成所需的其它文件和库传递给它。

下图显示递增链接的示例。

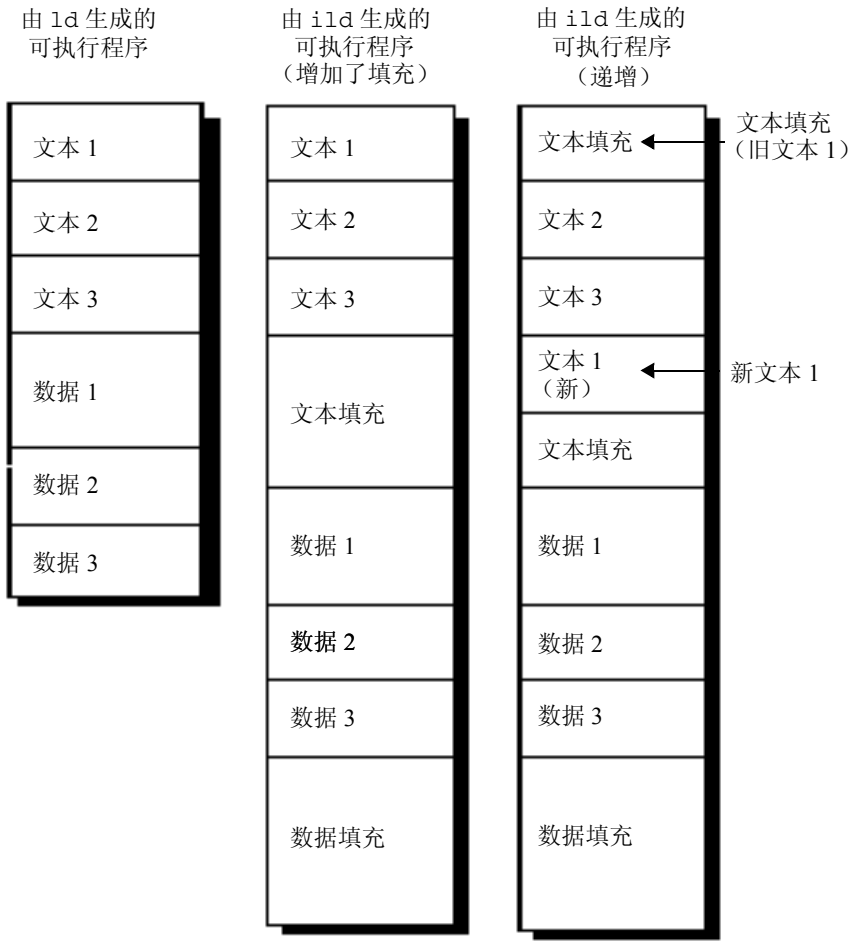

图 4-1 递增链接的示例

下列编译系统选项控制链接步骤是由 `ild` 还是由 `ld` 执行：

- `-xildon` 总是使用 `ild`
- `-xildoff` 总是使用 `ld`

注 – 如果 `-xildon` 和 `-xildoff` 均存在，驱动程序将使用最后列出的命令来选择链接程序。

- `-g` 如果 `-xildoff` 或 `-G` 均未指定，则对仅链接调用 （命令行上无源文件）使用 `ild`。有关 `-g` 的完整说明，请参见第 A-19 页上的第 A.3.28 节 “`-g`”。

- -G 防止 -g 选项影响链接程序选择。有关 -G 的完整说明，请参见第 A-19 页上的第 A.3.27 节 “-G”。

如果您使用 -g 选项调用调试，并且具有缺省 Makefile 结构 （它包含编译时选项，如链接命令行上的 -g），您可以在开发时自动使用 ild。

4.4 ild 的工作方式

执行初始链接时， ild 保存以下各项的信息：

- 查看的所有目标文件。
- 生成的可执行程序的符号表。
- 在编译时未解析的所有符号引用。

初始的 ild 链接所用的时间与 ld 链接所用的时间大体相同。

执行递增连接时， ild 执行以下操作：

- 确定已更改的文件。
- 重新链接修改的目标文件。
- 使用存储的信息修改程序其余部分更改的符号引用。

递增 ild 链接比 ld 链接快得多。

一般说来，只执行一次初始链接，所有后续链接是递增的。

例如， ild 保存代码中引用符号 foo 的所有位置的列表。如果您执行更改 foo 的值的递增链接， ild 必须更改 foo 的所有引用的值。

ild 展开程序的组件，并且可执行程序的每段均增加了填充。填充使可执行模块比 ld 链接它们时大。由于在执行连续递增链接期间目标文件增大，因此填充可能会耗尽。如果出现这种情况， ild 将显示一则消息并对可执行程序执行完全重新链接。

例如，如图 4-1 所示，三列中的每列均显示链接的可执行程序中的文本和数据的序列。左列显示由 ld 链接的可执行程序中的文本和数据。中间列显示由 ild 链接的可执行文件中增加的文本填充和数据填充。假定对文本 1 的源文件的更改导致文本段增长而不影响其它段的大小。右列显示文本 1 的原始位置已被文本填充取代 （文本 1 已失效）。文本 1 已移动，占用一部分文本填充空间。

要生成更小的非递增可执行程序，请使用 -xildoff 选项运行编译器驱动程序 （例如，cc 或 CC），这样将调用 ld 以生成更简洁的可执行程序。

dbx 可以调试从 ild 生成的可执行程序，因为 dbx/调试器理解 ild 在程序之间插入的填充。

对于 ild 不理解的任何命令行选项， ild 调用 ld。 ild 与 ld（在 /usr/ccs/bin/ld 中）兼容。有关详情，请参见第 4-9 页上的第 4.7 节 “ild 选项”。

没有 ild 使用的特殊文件或额外文件。

4.5 ild 的局限性

当调用 ild 以创建共享对象时， ild 调用 ld 以创建链接。

如果您更改的目标文件所占比例较高， ild 的性能会大大降低。当 ild 检测到更改的文件所占比例较高时，自动执行完全重新链接。

不要使用 ild 生成用于发货的最终产品代码。程序的某些部分已由于填充而展开，因此 ild 使文件变大。由于填充以及链接需要的额外时间，建议您不要对产品代码使用 -xildon 选项。（如果存在 -g，请在链接行上使用 -xildoff。）

ild 链接小程序并非快得多，可执行程序的大小增大比大程序大。

处理可执行程序的第三方工具在处理 ild 生成的二进制数据时会出现意外结果。

任何修改可执行程序的程序 （例如， strip 或 mcs）均可能影响 ild 执行递增链接的能力。如果出现了这种情况， ild 将发出一则消息并执行完全重新链接。有关完全重新链接的详细信息，请参见第 4-5 页上的第 4.6 节 “完全重新链接的原因”。

4.6 完全重新链接的原因

以下部分解释在哪种情况下 ild 调用 ld 以完成链接。

4.6.1 ild 延迟链接消息

消息 “ild： **正在调用** ld **以完成链接**” ... 意味着 ild 无法完成链接，并且正在将链接请求交给 ld 完成。在缺省情况下，必要时显示这些消息。您可以通过使用 -z i_quiet 选项禁止这些消息。

如果隐式请求 ild(-g)，则禁止以下消息。但是，如果 -xildon 在命令行上，则显示以下消息。如果您使用 -z i_verbose 选项，则在所有情况下均显示该消息。如果您使用 -z i_quiet 选项，则从不显示该消息。

ild： **正在调用** ld **以完成链接 -- 无法处理归档** *library name* **中的共享库**

下面是 -z i_verbose 消息的更多示例：

ild: **正在调用 ld 以完成链接 -- 无法处理关键字** *Keyword*

ild: **正在调用 ld 以完成 链接 -- 无法处理关键字** -d

ild: **正在调用 ld 以完成链接 -- 无法处理** -z *keyword*

ild: **正在调用 ld 以完成链接 -- 无法处理参数** *keyword*

4.6.2 ild 重新链接消息

消息 “ild: （**执行完全重新链接**）” ... 意味着由于某种原因， ild 无法执行递增链接而必须执行完全重新链接。这并不是错误。它通知您该链接需要的时间将比递增链接需要的时间长 （有关详情，请参见第 4-4 页上的第 4.4 节 “ild 的工作方式”）。 ild 选项 -z i_quiet 和 -z i_verbose 可以控制 ild 消息。某些消息采用具有描述性文本的冗余模式。

您可以通过使用 ild 选项 -z i_quiet 禁止所有这些消息。如果缺省消息采用冗余模式，该消息以省略号 ([...]) 结束，省略号表示还有更多信息。您可以通过使用 -z i_verbose 选项查看附加信息。选择 -z i_verbose 选项时显示示例消息。

4.6.3 示例 1：内部可用空间耗尽

最常见的完全重新链接消息是 “**内部可用空间耗尽**”消息：

```
$ cat test1.c
int main() { return 0; }
$ rm a.out
$ cc -xildon -c -g test1.c
$ cc -xildon -z i_verbose -g test1.o

$ cc -xildon -z i_verbose -g test1.c

ild: （执行完全重新链接）输出文件中的内部可用空间耗尽
（段）
$
```

 # 此命令创建 test1.o
 # 此命令创建具有最少调试信息的 a.out。
 # 单行编译和链接将所有调试信息放入 a.out。

这些命令显示，从单行编译到双行编译会导致可执行程序中的调试信息增长。这种增长导致 ild 用尽空间而执行完全重新链接。

4.6.4 示例 2：运行 strip

当您运行 strip 时，会出现另一个问题。自示例 1 继续：

```
# Strip a.out
# 尝试执行递增链接
$ strip a.out
$ cc -xildon -z i_verbose -g test1.c

ild:（执行完全重新链接） a.out 在上一次递增链接后已被更改 -- 也许您在其上运行了 strip 或 mcs?
$
```

4.6.5 示例 3：ild 版本

当 ild 新版本在由 ild 旧版本创建的可执行程序上运行时，您会看到以下错误消息：

```
# 假定 old_executable 由 ild 的较早版本创建
$ cc -xildon -z i_verbose foo.o -o old_executable

ild:（执行完全重新链接） a.out 在上次链接之后，更新 ild 已安装 (2/16)
```

注 – 数字 (2/16) 仅用于内部报告。

4.6.6 示例 4：被更改的文件过多

有时 ild 确定完全重新链接将比递增链接更快。例如：

```
$ rm a.out
$ cc -xildon -z i_verbose \
     x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
$ touch x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o
$ cc -xildon -z i_verbose \
     x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
ild:（执行完全重新链接）被更改的文件过多
```

这里，使用 touch 命令会导致 ild 确定文件 x0.o 到 x8.o 均已更改，并确定完全重新链接将比递增链接全部九个目标文件更快。

4.6.7 示例 5：完全重新链接

与导致对当前链接进行完全重新链接的前面示例比较，某些条件可导致对下一个链接进行完全重新链接。

下次尝试链接该程序时，您会看到以下消息：

 # ild 检测先前的错误并执行完全重新链接

```
$ cc -xildon -z i_verbose broken.o
ild: (执行完全重新链接）由于先前链接中的问题而无法执行递增重新链接
```

发生完全重新链接。

4.6.8 示例 6：新工作目录

```
% cd /tmp
% cat y.c
    int main(){ return 0;}
% cc -c y.c
% rm -f a.out
% cc -xildon -z i_verbose y.o -o a.out

% mkdir junk
% mv y.o y.c a.out junk
% cd junk
% cc -xildon -z i_verbose y.o -o a.out

ild: (执行完全重新链接）当前目录已从 '/tmp' 更改为 '/tmp/junk'
%
```

 # 初始链接，cwd 等于 to /tmp

 # 递增链接，cwd 现在为 /tmp/junk

4.7 ild 选项

本节描述编译系统直接接受的链接程序控制选项以及可通过编译系统传递到 ild 的链接程序选项。

4.7.1 -a

仅在静态模式下生成可执行的目标文件；对未定义的引用发出错误消息。这是静态模式的缺省行为。

4.7.2 -B dynamic | static

控制库包含的选项。选项 -Bdynamic 仅在动态模式下有效。可以在命令行上任意多次指定这些选项进行切换：如果指定 -Bstatic 选项，则在出现 -Bdynamic 之前，不会接受共享目标。参见第 4-10 页上的第 4.7.9 节 “-lx”选项。

4.7.3 -d y|n

如果指定 -dy （缺省值），则 ild 使用动态链接；如果指定 -dn，则 ild 使用静态链接。参见第 4-9 页上的第 4.7.2 节 “-B dynamic | static”选项。

4.7.4 -e epsym

将输出文件的入口点地址设置为符号 epsym 的入口点地址。

4.7.5 -g

如果指定 -g 选项，编译系统将调用 ild 而不是 ld，除非以下某个条件为真：

- 指定 -G 选项 （生成共享库）
- 存在 -xildoff 选项
- 命令行上指定了任何源文件

4.7.6 -I *name*

生成可执行程序时，将 *name* 用作要写入程序头文件的解释程序路径名。静态模式下，缺省值是无解释程序；在动态模式下，缺省值为运行时链接程序的名称 /usr/lib/ld.so.1。这两种情况下，缺省值均可被 -I*name* 覆盖。 exec 在装入 a.out 时仅装入该解释程序，并将控制直接传递给解释程序而不是 a.out。

4.7.7 -i

忽略 LD_LIBRARY_PATH 设置。如果 LD_LIBRARY_PATH 设置有效地影响运行时库搜索，可能干扰所执行的链接编辑，此时该选项很有用。（这也适用于 LD_LIBRARY_PATH_64 设置）。

4.7.8 -L*path*

将 *path* 增加到库搜索目录中。 ild 首先在 -L 选项指定的任何目录中查找库，然后在标准目录中查找。只有该选项位于命令行上它应用的 -l 选项前面时，该选项才有用。您可以使用环境变量 LD_LIBRARY_PATH 和 LD_LIBRARY_PATH_64 补充库搜索路径（参见第 4-14 页上的 “LD_LIBRARY_PATH”）。

4.7.9 -l*x*

搜索库 lib*x*.so 或 lib*x*.a，它们分别是共享对象和归档库的常规名称。在动态模式下，除非 -Bstatic 选项起作用，否则 ild 在库搜索路径中指定的每个目录中查找文件 lib*x*.so 或 lib*x*.a。目录搜索在包含这两个文件的第一个目录处停止。如果 -l 扩展到文件名的形式为 lib*x*.so 和 lib*x*.a 的两个文件，则 ild 选择以 .so 结尾的文件。如果未找到 lib*x*.so，则 ild 接受 lib*x*.a。在静态模式下，或者当 -Bstatic 选项起作用时，ild 仅选择以 .a 结尾的文件。遇到库名时即找到库，因此 -l 的位置非常重要。

4.7.10 -m

产生标准输出中输入/输出段的内存映射或列表。

4.7.11 -o *outfile*

产生命名为 *outfile* 的输出目标文件。缺省目标文件的名称为 a.out。

4.7.12 -Q y|n

在 -Qy 下，将 ident 字符串增加到输出文件的 .comment 段，以标识用于创建文件的链接编辑器的版本。已存在多个链接步骤时，如使用 ld -r 时，这导致产生多个 ld ident。这与 cc 命令的缺省操作相同。选项 -Qn 禁止版本标识。

4.7.13 -R*path*

该选项提供一个以冒号分隔的列表，该列表指定运行时链接程序的库搜索目录。如果存在并且非空，则 *path* 记录在输出目标文件中并传递给运行时链接程序。该选项的多个实例并置并被以冒号分隔。

4.7.14 -s

从输出文件去除符号信息。删除任何调试信息及关联的重定位条目。另外，除了可重定位文件或共享对象之外，从输出目标文件中删除符号表和字符串表段。

4.7.15 -t

禁止关于大小不同的多重定义符号的警告。

4.7.16 -u *symname*

输入 *symname* 作为符号表中未定义的符号。这有助于完全从归档库中装入，因为符号表最初为空，强制装入第一个例程需要未解析的引用。命令行上该选项的位置非常重要，它必须位于定义符号的库之前。

4.7.17 -V

输出关于所使用的 ild 的版本的消息。

4.7.18 -xildoff

递增链接程序关闭。强制使用捆绑的 ld。如果未在使用 -g，或正在使用 -G，则这是缺省值。您可以使用 -xildon 覆盖该缺省值。

4.7.19 -xildon

递增链接程序打开。在递增模式下强制使用 ild。如果正在使用 -g，则这是缺省值。您可以使用 -xildoff 覆盖该缺省值。

4.7.20 -YP,*dirlist*

（仅适用于 cc）更改用于查找库的缺省目录。选项 *dirlist* 是以冒号分隔的路径列表。

注 – ild 对特殊选项使用 “-z *name*” 形式。-z 选项的 *i_* 前缀表明这些选项特定于 ild。

4.7.21 -z allextract|defaultextract|weakextract

改变从该指令后面的任何归档中提取对象的标准。在缺省情况下，提取归档成员以满足未定义的引用并使用数据定义提升暂定定义。弱符号引用不会触发提取。在 -z allextract 下，从归档中提取所有归档成员。在 -z weakextract 下，弱引用触发归档提取。-z defaultextract 提供一种在使用前面的提取选项之后返回缺省值的方法。

4.7.22 -z defs

强制在链接末尾存在任何未定义的符号时产生致命错误。当生成可执行程序时，这是缺省值。这在生成共享对象以确保对象是自我包含 （即内部解析其所有符号引用）的对象时也很有用。

4.7.23 -z i_dryrun

（仅适用于 ild。）打印将由 ild 链接的文件的列表并退出。

4.7.24 -z i_full

（仅适用于 ild。）在递增模式下执行完全重新链接。

4.7.25 -z i_noincr

（仅适用于 ild。）在非递增模式下运行 ild（建议客户不要使用这种模式 — 仅用于测试）。

4.7.26 -z i_quiet

（仅适用于 ild）禁止所有 ild 重新链接消息。

4.7.27 -z i_verbose

（仅适用于 ild）详述有关某些 ild 重新链接消息的缺省信息。

4.7.28 -z nodefs

允许使用未定义的符号。当生成共享对象时，这是缺省值。用于可执行程序时，未指定对此类 “未定义的符号”的引用的行为。

4.8 从编译系统传递给 ild 的选项

ild 接受以下选项，但是您必须使用以下形式：

-Wl,*arg*,*arg*（对于 cc），通过编译系统将它们传递给 ild。

4.8.1 -a

仅在静态模式下生成可执行的目标文件；对未定义的引用产生错误。这是静态模式的缺省行为。选项 -a 不能与 -r 选项一起使用。

4.8.2 -e *epsym*

将输出文件的入口点地址设置为符号 *epsym* 的入口点地址。

4.8.3 -I name

生成可执行程序时，将 *name* 用作要写入程序头文件的解释程序路径名。静态模式下，缺省值是无解释程序；在动态模式下，缺省值为运行时链接程序的名称 /usr/lib/ld.so.1。这两种情况下，缺省值均可被 -I *name* 覆盖。exec 系统调用在装入 a.out 时装入该解释程序，并控制直接传递给解释程序而不是 a.out。

4.8.4 -m

产生标准输出中输入/输出段的内存映射或列表。

4.8.5 -t

禁止关于多次定义但大小不同的符号的警告。

4.8.6 -u *symname*

输入 *symname* 作为符号表中未定义的符号。这有助于完全从归档库中装入，因为符号表最初为空，强制装入第一个例程需要未解析的引用。命令行上该选项的位置非常重要，它必须位于定义符号的库之前。

4.8.7 环境

LD_LIBRARY_PATH

查找使用 -l 选项指定的库时搜索的目录的列表。多个目录由以冒号分隔。一般情况下，它包含两个由分号分隔的目录列表：

```
dirlist1; dirlist2
```

如果使用任意数目的 -L 具体值调用 ild，如下所示：

```
ild ...-Lpath1 ...  -Lpathn ...
```

则搜索路径顺序为：

```
dirlist1  path1  ...  pathn  dirlist2  LIBPATH
```

当目录列表不包含分号时，它解释如下：

```
dirlist2
```

LD_LIBRARY_PATH 还用于为运行时链接程序指定库搜索目录。也就是说，如果环境中存在 LD_LIBRARY_PATH，则运行时链接程序将在该变量中指定的目录中 （在缺省目录之前）查找在执行时要与程序链接的共享对象。

注 – 当运行 set-user-ID 或 set-group-ID 程序时，运行时链接程序只在 /usr/lib 中查找库。它还查找可执行程序中指定的任何完整路径名。完整路径名是构造可执行程序时指定的运行路径的结果。指定为相对路径名的任何库依赖性会被忽略且无提示。

LD_LIBRARY_PATH_64

对于 Solaris 7 和 Solaris 8 软件，该环境变量与 LD_LIBRARY_PATH 相似，但是在查找 64 位依赖性时覆盖它。

当您在 SPARC 处理器上运行 Solaris 7 或 Solaris 8 软件并且在 32 位模式下链接时，LD_LIBRARY_PATH_64 会被忽略。如果只定义了 LD_LIBRARY_PATH，则它用于 32 位和 64 位链接。如果 LD_LIBRARY_PATH 和 LD_LIBRARY_PATH_64 均已定义，则将使用 LD_LIBRARY_PATH 执行 32 位链接，使用 LD_LIBRARY_PATH_64 执行 64 位链接。

LD_OPTIONS

ild 的一组缺省选项。 LD_OPTIONS 由 ild 解释，仿佛其值已放在命令行上紧跟在用于调用 ild 的名称后面，如下所示：

```
ild $LD_OPTIONS  ...  other-arguments ...
```

LD_PRELOAD

要由运行时链接程序解释的共享对象的列表。指定的共享对象链接在所执行的程序之后，并且在该程序引用的任何其它共享对象之前。

注 – 当运行 set-user-ID 或 set-group-ID 程序时，该选项会被忽略且无提示。

LD_RUN_PATH

另一种为链接编辑器指定运行路径的方法（参见 -R 选项）。如果 LD_RUN_PATH 和 -R 选项均已指定，则使用 -R。

LD_DEBUG

（ild 不支持）提供导致运行时链接程序将调试信息打印到标准错误输出的标记。特殊标记 *help* 表示整个可用标记列表。

注 – 以字符 “LD_”开头的环境变量名保留供将来可能出现的 ld 增强功能使用。以字符 “ILD_”开头的环境变量名保留供将来可能出现的 ild 增强功能使用。

4.9 ild 不支持的 ld 选项

如果 ild 确定某个命令行选项未实现，则 ild 直接调用 /usr/ccs/bin/ld 以执行链接。

ild 不支持以下选项，这些选项可能被提供给编译系统。

4.9.1 -B symbolic

仅在动态模式下，当生成共享对象时，如果定义可用，将全局符号的引用绑定到对象内的定义上。通常，即使定义可用，共享对象内的全局符号的引用也直到运行时才绑定，以便可执行程序中或其它共享对象中相同符号的定义可以覆盖对象自身的定义。除非 -z *defs* 覆盖该选项，否则 ld 对未定义的符号发出警告。

4.9.2 -b

仅在动态模式下，当创建可执行程序时，不对引用共享对象中符号的重定位进行特殊处理。如果未设置 -b 选项，链接编辑器为共享对象中定义的函数的引用创建独立于特殊位置的重定位，并安排由运行时链接程序将共享对象中定义的数据对象复制到可执行程序的内存映射中。如果设置 -b 选项，输出代码的效率可能会提高，但是它的共享性会降低。

4.9.3 -G

仅在动态模式下，生成共享对象。允许使用未定义的符号。

4.9.4 -h name

仅在动态模式下，当生成共享对象时，记录对象的动态段中的 *name*。选项 *name* 记录在与该对象而不是该对象的 UNIX 系统文件名链接的可执行程序中。因此，运行时链接程序将 *name* 用作在运行时查找的共享对象的名称。

4.9.5 -z muldefs

允许使用多个符号定义。在缺省情况下，在可重定位的对象之间出现的多个符号定义会导致致命的错误条件。该选项禁止错误条件，并允许采用第一个符号定义。

4.9.6 -z text

仅在动态模式下，强制在仍存在对不可写的可分配段的任何重定位时产生致命错误。

4.10 其它不支持的命令

此外， ild 不支持可直接传递给 ld 的以下选项：

4.10.1 -D *token,token, …*

将每个标记指定的调试信息打印到标准错误输出。特殊标记 *help* 表示整个可用标记列表。

4.10.2 -F name

仅在生成共享对象时有用。指定共享对象的符号表用作由 *name* 指定的共享对象的符号列表的 “过滤器”。

4.10.3 -M *mapfile*

读取 *mapfile* 作为 ld 指令的文本文件。有关映射文件的描述，请参见 《*SunOS 5.3 Linker and Libraries Manual*》。

4.10.4 -r

合并可重定位目标文件以生成一个可重定位目标文件。ld 不会对无法解析的引用发出错误信息。该选项不能在动态模式下使用，也不能与 -a 一起使用。

4.11 ild 使用的文件

- lib*x*.a 库
- a.out 输出文件

第 5 章

lint 源代码检验器

本章解释如何使用 lint 程序检查您的 C 代码是否存在可能导致编译失败或在运行时出现意外结果的错误。在很多情况下， lint 会向您发出关于编译器未作必要标志的不正确、有错误倾向或不标准的代码的警告。

lint 程序发出 C 编译器生成的错误和警告消息。它还发出关于潜在错误和可移植性问题的警告。由 lint 发出的许多消息可以帮助您提高程序的效率，包括减小其大小和必需的内存。

lint 程序使用与编译器相同的语言环境，并且 lint 的输出传递至 stderr。有关在执行基于类型的别名歧义消除之前如何使用 lint 检查代码的详细信息和示例，请参见第 6 章。

5.1 基本和增强 lint 模式

lint 程序在以下两种模式下运行：

- *基本*，缺省模式
- *增强*，包括由基本 lint 执行的一切以及附加的详细代码分析

在基本模式和增强模式下， lint 通过标记文件 （包括您已使用的任何库）之间定义和使用的不一致，补偿 C 中单独和独立的编译。特别是在一个大的项目环境中，同一个函数可能被不同程序员在数百个单独的代码模块中使用， lint 有助于发现以其它方式很难发现的错误。例如，调用某个函数时使用的参数若比所需的参数少一个，该函数将在栈中查找该调用从未推进栈的值，结果是在一个条件下正确，在另一个条件下不正确，具体取决于内存中该栈位置发生的情况。通过标识类似的依赖性以及对机器体系结构的依赖性， lint 可以提高运行于您的机器或其他人的机器上的代码的可靠性。

在增强模式下， lint 提供比基本模式更详细的报告。在基本模式下， lint 的功能包括：

- 源程序的结构和流分析
- 常量传播和常量表达式求值
- 控制流和数据流的分析
- 数据类型使用的分析

在增强模式下， lint 可以检测以下问题：

- 未使用的 #include 指令、变量和过程
- 内存释放之后内存的使用
- 未使用的赋值
- 初始化之前使用变量值
- 未分配内存的释放
- 写入常量数据段时使用指针
- 非等价宏重定义
- 未执行到的代码
- 符合联合中值类型的用法
- 实际参数的隐式强制类型转换。

5.2 使用 lint

从命令行调用 lint 程序及其选项。要在基本模式下调用 lint，请使用以下命令：

```
% lint file1.c file2.c
```

使用 -Nlevel 或 -Ncheck 选项调用增强 lint。例如，您可以按如下方式调用增强 lint：

```
% lint -Nlevel=3 file1.c file2.c
```

lint 在两次*传递*中检查代码。在第一次传递中， lint 检查 C 源文件中是否有错误条件；在第二次传递中，它检查 C 源文件之间是否有不一致的情况。除非使用 -c 调用 lint，否则该进程对用户是不可见的：

```
% lint -c file1.c file2.c
```

该命令指示 lint 仅执行第一次传递并在中间文件 *file1*.ln 和 *file2*.ln 中收集与第二次传递相关的信息 — 关于 *file1*.c 和 *file2*.c 之间定义和使用的不一致：

```
% ls
file1.c
file1.ln
file2.c
file2.ln
```

这样， lint 的 -c 选项类似于 cc 的 -c 选项，禁止编译的链接编辑阶段。一般说来，lint 的命令行语法严格遵循 cc 的语法。

当 .ln 文件被 lint 时：

```
% lint file1.ln file2.ln
```

执行第二次传递。 lint 按命令行顺序处理任意多个 .c 和 .ln 文件。因此，

```
% lint file1.ln file2.ln file3.c
```

指示 lint 检查 *file3*.c 是否有内部错误以及所有三个文件是否一致。

lint 按 cc 同样的顺序在目录中查找包含的头文件。您可以如使用 cc 的 -I 选项那样使用 lint 的 -I 选项。参见第 2-24 页上的第 2.13 节 “如何指定包含文件”。

您可以在同一命令行中指定 lint 的多个选项。除非其中一个选项带有参数或者选项有多个字母，否则选项可以并置：

```
% lint -cp -Idir1 -Idir2 file1.c file2.c
```

该命令指示 lint 执行以下操作：

- 仅执行第一次传递
- 执行附加的可移植性检查
- 在指定的目录中查找包含的头文件

lint 有许多选项，可用来指示 lint 执行某些任务并报告某些条件。

5.3 lint 选项

lint 程序是一个静态分析器。它不能求出它检测到的依赖性的运行时结果。例如，某些程序可能包含数百个无法执行到的 break 语句，这些语句并不重要，但是 lint 仍然标记它们。这是一个示例，其中包含 lint 命令行选项和指令 — 源代码文本中嵌入的特殊注释：

- 您可以使用 -b 选项调用 lint，以禁止关于无法执行到的 break 语句的所有错误消息。
- 您可以在任何无法执行到的语句前面加上注释 /*NOTREACHED*/ 以禁止对该语句的诊断。

lint 选项在下面按字母顺序列出。几个 lint 选项与禁止 lint 诊断消息有关。这些选项也在表 5-7 中列出，跟在这些按字母顺序排列的选项后面，并显示选项禁止的特定消息。用于调用增强 lint 的选项以 -N 开始。

lint 识别许多 cc 命令行选项，包括 -A、 -D、 -E、 -g、 -H、 -O、-P、 -U、 -Xa、 -Xc、 -Xs、 -Xt 和 -Y，虽然 -g 和 -O 被忽略。未识别的选项被警告并忽略。

5.3.1 -#

打开冗余模式，显示调用的每个组件。

5.3.2 -###

显示调用但实际并不执行的每个组件。

5.3.3 -a

禁止某些消息。参考表 5-7。

5.3.4 -b

禁止某些消息。参考表 5-7。

5.3.5 -C *filename*

使用指定的文件名创建一个 .ln 文件。这些 .ln 文件仅为 lint 的第一次传递产生的文件。*filename* 可以为完整路径名。

5.3.6 -c

为命令行上命名的各个 .c 文件创建一个 .ln 文件，该文件包含与 lint 的第二次传递相关的信息。不执行第二次传递。

5.3.7 -dirout=*dir*

指定目录 *dir*，其中将存放 lint 输出文件 （.ln 文件）。此选项影响 -c 选项。

5.3.8 -err=warn

-err=warn 是 -errwarn=%all 的宏。参见第 5-9 页上的第 5.3.14 节“-errwarn=t”。

5.3.9 -errchk=*l*[, *l*]

执行由 *l* 指定的附加检查。缺省值为 -errchk=%none。指定 -errchk 相当于指定 -errchk=%all。*l* 是以逗号分隔的检查列表，它由下列一项或多项组成。例如， -errchk=longptr64,structarg。

表 5-1 -errchk 值

值	含义
%all	执行 -errchk 的所有检查。
%none	不执行 -errchk 的任何检查。这是缺省值。
[no%]locfmtchk	在 lint 的第一次传递期间检查类 printf 格式字符串。无论您是否使用 -errchk=locfmtchk， lint 始终在第二次传递时检查类 printf 格式字符串。
[no%]longptr64	检查是否可移植到长整数和指针的大小为 64 位并且无格式整数的大小为 32 位的环境。即使使用了显式强制类型转换，也检查指针表达式和长整数表达式是否赋值为无格式整数。

表 5-1 -errchk 值 *（续）*

值	含义
[no%]structarg	检查通过值传递的结构参数并在形式参数类型未知时报告情况。
[no%]parentheses	检查代码中优先级的透明度。使用此选项增强代码的可维护性。如果 -errchk=parentheses 返回一个警告，请考虑使用附加圆括号清楚地表示代码中操作的优先级。
[no%]signext	检查标准的 ISO C 值保留规则允许在不带符号的整型表达式中使用带符号的整数值的符号扩展的情形。只有在同时也指定 -errchk=longptr64 时，此选项才产生错误消息。
[no%]sizematch	检查较长整数到较短整数的赋值并发出警告。对于具有不同符号的相同长度的整数之间的赋值 （不带符号的整数获取带符号的整数），也会发出这些警告。

5.3.10 -errfmt=*f*

指定 lint 输出的格式。*f* 可以为以下格式之一：macro、 simple、 src 或 tab。

表 5-2 -errfmt 值

值	含义
macro	显示错误的源代码、行号和位置，并展开宏
simple	对于一行 （简单）诊断消息，在括号中显示错误的行号和位置号。类似于 -s 选项，但是包含错误的位置信息
src	显示错误的源代码、行号和位置 （不展开宏）
tab	以制表格式显示。这是缺省值。

缺省值为 -errfmt=tab。指定 -errfmt 相当于指定 -errfmt=tab。

如果指定了多种格式，则使用最后指定的格式，并且 lint 发出关于未使用格式的警告。

5.3.11 -errhdr=*h*

与 -Ncheck 一起使用时，允许报告头文件的某些消息。*h* 是以逗号分隔的列表，它由下列一项或多项组成：*dir*、 no%*dir*、 %all、 %none、 %user。

表 5-3 -errhdr 值

值	含义
dir	检查目录 *dir* 中使用的头文件
no%*dir*	不检查目录 *dir* 中使用的头文件
%all	检查使用的所有头文件
%none	不检查头文件。这是缺省值。
%user	检查使用的所有用户头文件，即除 /usr/include 及其子目录中的头文件以及由编译器提供的头文件之外的所有头文件。

缺省值为 -errhdr=%none。指定 -errhdr 相当于指定 -errhdr=%user。

示例：

```
% lint -errhdr=inc1 -errhdr=../inc2
```

检查目录 inc1 和 ../inc2 中使用的头文件。

```
% lint -errhdr=%all,no%../inc
```

检查除目录 ../inc 中的头文件之外的所有已使用的头文件。

5.3.12 -erroff=*tag*(, *tag*)

禁止或启用 lint 错误消息。

t 是以逗号分隔的列表，它由下列一项或多项组成：*tag*、 no%*tag*、 %all、 %none。

表 5-4 -erroff 值

值	含义
tag	禁止由此 *tag* 指定的消息。您可以通过使用 -errtags=yes 选项显示消息的标记。
no%*tag*	启用由此 *tag* 指定的消息
%all	禁止所有消息
%none	启用所有消息。这是缺省值。

缺省值为 -erroff=%none。指定 -erroff 相当于指定 -erroff=%all。

示例：

```
% lint -erroff=%all,no%E_ENUM_NEVER_DEF,no%E_STATIC_UNUSED
```

仅打印消息 “enum 从未被定义”和 “static 未使用”，并禁止其它消息。

```
% lint -erroff=E_ENUM_NEVER_DEF,E_STATIC_UNUSED
```

仅禁止消息 “enum 从未被定义”和 “static 未使用”。

5.3.13 -errtags=*a*

显示每个错误消息的消息标记。 *a* 可以为 yes 或 no。缺省值为 -errtags=no。指定 -errtags 相当于指定 -errtags=yes。

与所有 -errfmt 选项配合使用。

5.3.14 -errwarn=*t*

如果发出指示的警告消息，lint 以失败状态退出。*t* 是以逗号分隔的列表，它由下列一项或多项组成：*tag*、 no%*tag*、 %all、 %none。顺序很重要；例如，如果发出除 *tag* 之外的任何警告， %all,no%*tag* 导致 lint 以致命状态退出。下表列出 -errwarn 值：

表 5-5 -errwarn 值

tag	如果此 *tag* 指定的消息作为警告消息发出，该值导致 lint 以致命状态退出。如果未发出 *tag*，则没有影响。
no%*tag*	如果 *tag* 指定的消息仅作为警告消息发出，该值防止 lint 以致命状态退出。如果未发出 *tag*，则没有影响。使用此选项回复警告消息（该警告消息先前由此选项及 *tag* 或 %all 指定），以使其在作为警告消息发出时，不会导致 lint 以致命状态退出。
%all	如果发出任何警告消息，该值导致 lint 以致命状态退出。 %all 后面可以跟 no%*tag*，以免除特定的警告消息出现此行为。
%none	如果发出任何警告消息，该值防止任何警告消息导致 lint 以致命状态退出。

缺省值为 -errwarn=%none。如果您仅指定 -errwarn，它等价于 -errwarn=%all。

5.3.15 -F

当引用命令行上命名的 .c 文件时，打印命令行上提供的路径名而不仅仅是它们的基名。

5.3.16 -fd

报告关于旧式样函数定义或声明的情况。

5.3.17 -flagsrc=*file*

使用文件 *file* 中包含的选项执行 lint。可在 *file* 中指定多个选项，每行一个。

5.3.18 -h

禁止某些消息。参考表 5-7。

5.3.19 -I*dir*

在目录 *dir* 中查找包含的头文件。

5.3.20 -k

改变 `/* LINTED [message] */` 指令或 `NOTE(LINTED(message))` 注释的行为。通常，`lint` 对这些指令后面的代码禁止警告消息。如果不设置该选项， `lint` 不禁止消息，而是打印包含指令或注释内部的注释的附加消息。

5.3.21 -L*dir*

与 `-l` 配合使用时，在目录 *dir* 中查找 `lint` 库。

5.3.22 -l*x*

访问 `lint` 库 `llib-l`*x*`.ln`。

5.3.23 -m

禁止某些消息。参考表 5-7。

5.3.24 -Ncheck=*c*

检查头文件是否有相应的声明；检查宏。 *c* 是以逗号分隔的检查列表，它由下列一项或多项组成：macro、 extern、 %all、 %none、 no%macro、 no%extern。

表 5-6 -Ncheck 值

值	含义
macro	检查文件之间的宏定义的一致性
extern	检查源文件与关联的头文件（例如， file1.c 与 file1.h）之间的声明的一一对应。确保头文件中既没有无关的也没有缺少的 extern 声明。
%all	执行 -Ncheck 的所有检查
%none	不执行 -Ncheck 的任何检查。这是缺省值。
no%macro	不执行 -Ncheck 的任何 macro 检查
no%extern	不执行 -Ncheck 的 extern 检查

缺省值为 -Ncheck=%none。指定 -Ncheck 相当于指定 -Ncheck=%all。

值可能用逗号结合，例如， -Ncheck=extern,macro。

示例：

```
% lint -Ncheck=%all,no%macro
```

执行除宏检查之外的所有检查。

5.3.25 -Nlevel=*n*

指定报告问题的分析级别。此选项允许您控制检测到的错误量。级别越高，检验时间越长。 *n* 是一个数字：1、 2、 3 或 4。缺省值为 -Nevel=2。指定 -Nlevel 相当于指定 -Nlevel=4。

5.3.25.1 -Nlevel=1

分析单个过程。报告发生在某些程序执行路径上的无条件错误。不执行全局数据和控制流分析。

5.3.25.2 -Nlevel=2

缺省值。分析整个程序，包括全局数据和控制流。报告发生在某些程序执行路径上的无条件错误。

5.3.25.3 -Nlevel=3

分析整个程序，包括常量传播、常量用作实际参数时的情况以及在 `-Nlevel=2` 下执行的分析。

在此分析级别检验 C 程序所用的时间比前一级别长 2 到 4 倍。需要额外的时间是因为 lint 通过为程序变量创建可能值的集合对程序进行部分解释。这些变量集是以常量以及包含程序中的常量操作数的条件语句为基础创建的。这些集合形成创建其它集合的基础（一种常量传播形式）。作为分析结果接收的集合根据以下算法来评估正确性：

如果对象的所有可能值之中存在一个正确值，则该正确值用作进一步传播的基础；否则诊断出一个错误。

5.3.25.4 -Nlevel=4

分析整个程序，并报告使用某些程序执行路径时会发生的条件错误，以及在 `-Nlevel=3` 下执行的分析。

在此分析级别上，存在更多诊断消息。分析算法通常对应于 `-Nlevel=3` 的分析算法，所不同的是任何无效值现在生成一个错误消息。在此级别上进行分析所需的时间量增大两个数量级 （大约慢 20 到 100 倍）。在这种情况下，所需的额外时间与以递归、条件语句等为特征的程序复杂性成正比。因此，对超过 100,000 行的程序使用此级别的分析可能很困难。

5.3.26 -n

禁止检查与缺省 `lint` 标准 C 库的兼容性。

5.3.27 -o*x*

导致 lint 创建一个 lint 库，名称为 llib-l*x*.ln。该库是使用 lint 在第二次传递中使用的所有 .ln 文件创建的。 -c 选项使 -o 选项的任何使用无效。要生成 llib-l*x*.ln 而不会产生无关系的消息，您可以使用 -x 选项。如果 lint 库的源文件仅为外部接口，则 -v 选项很有用。如果使用 -l*x* 调用 lint，则生成的 lint 库可以在以后使用。

缺省情况下，使用 lint 的基本格式创建库。如果您使用 lint 的增强模式，则创建的库将为增强格式，并且只能在增强模式下使用。

5.3.28 -p

启用某些与可移植性相关的消息。

5.3.29 -R*file*

将 .ln 文件写入 *file*，以供 cxref(1) 使用。如果此选项打开，此选项禁止增强模式。

5.3.30 -s

将复合消息转换为简单消息。

5.3.31 -u

禁止某些消息。参考表 5-7。此选项适合对较大程序的文件子集运行 lint。

5.3.32 -V

将产品名及发行版本写入标准错误输出。

5.3.33 -v

禁止某些消息。参考表 5-7。

5.3.34 -W*file*

将 .ln 文件写入 *file*，以供 cflow(1) 使用。如果此选项打开，此选项禁止增强模式。

5.3.35 -x

禁止某些消息。参考表 5-7。

5.3.36 -XCC=*a*

接受 C++ 式样注释。特别是，// 可用来指示注释的开始。*a* 可以为 yes 或 no。缺省值为 -XCC=no。指定 -XCC 相当于指定 -XCC=yes。

注 – 如果您使用 -xc99=%none，则只需使用此选项。在 -xc99=%all （缺省值）情况下， lint 接受由 // 指示的注释。

5.3.37 -Xalias_level[=*l*]

其中，*l* 是 any、basic、weak、layout、strict、std 或 strong 之一。有关不同的歧义消除级别的详细解释，请参见表 A-8。

如果您不指定 -Xalias_level，该标志的缺省值为 -Xalias_level=any。这意味着不存在基于类型的别名分析。如果您指定 -Xalias_level，但不提供级别，缺省值为 -Xalias_level=layout。

确保运行 lint 时所在的歧义消除级别不如运行编译器时所在的级别严格。如果运行 lint 时所在的歧义消除级别比编译时所在的级别更严格，结果将很难解释并且可能令人误解。

有关歧义消除的详细解释以及有助于歧义消除的 pragma 列表，请参见第 6 章。

5.3.38 -Xarch=v9

预定义 __sparcv9 宏并查找 lint 库的 v9 版本。

5.3.39 -Xc99[=*o*]

-Xc99 标志控制编译器对 C99 标准 （ISO/IEC 9899:1999，编程语言 -C）中实现功能的识别。

o 可以为下列之一：%all, %none.

-Xc99=%none 关闭对 C99 功能的识别。-Xc99=%all 打开对支持的 C99 功能的识别。

指定不带任何参数的 -Xc99 与指定 -Xc99=%all 相同。

注 – 虽然编译器支持级别缺省为附录 D 中列出的 C99 的功能，但是 /usr/include 中由 Solaris 软件提供的标准头文件并不符合 1999 ISO/IEC C 标准。
如果遇到错误消息，请尝试使用 -Xc99=%none 获取这些头文件的 1990 ISO/IEC C 标准行为。

5.3.40 -Xexplicitpar=*a*

(SPARC) 指示 lint 识别 #pragma MP 指令。 *a* 可以为 yes 或 no。缺省值为 -Xexplicitpar=no。指定 -Xexplicitpar 相当于指定 -Xexplicitpar=yes。

5.3.41 -Xkeeptmp=*a*

保留 lint 期间创建的临时文件，而不是自动删除它们。*a* 可以为 yes 或 no。缺省值为 -Xkeeptmp=no。指定 -Xkeeptmp 相当于指定 -Xkeeptmp=yes。

5.3.42 -Xtemp=*dir*

将临时文件的目录设置为 *dir*。如果没有此选项，临时文件进入 /tmp。

5.3.43 -Xtime=*a*

报告每次 lint 传递的执行时间。 *a* 可以为 yes 或 no。缺省值为 -Xtime=no。指定 -Xtime 相当于指定 -Xtime=yes。

5.3.44 -Xtransition=*a*

对 K&R C 和 Sun ISO C 之间的差异发出警告。 *a* 可以为 yes 或 no。缺省值为 -Xtransition=no。指定 -Xtransition 相当于指定 -Xtransition=yes。

5.3.45 -Xustr={ascii_utf16_ushort|no}

此选项允许编译器将格式 U"*ASCII_string*" 的字符串文字识别为无符号短整数数组。缺省值为 -Xustr=no，它禁止编译器识别 U"*ASCII_string* 字符串文字。
"-Xustr=ascii_utf16_ushort 允许编译器识别 U"*ASCII_string*" 字符串文字。

5.3.46 -y

将命令行上命名的各个 .c 文件视为以指令 /* LINTLIBRARY */ 或注释 NOTE(LINTLIBRARY) 开始。 lint 库通常是使用 /* LINTLIBRARY */ 指令或 NOTE(LINTLIBRARY) 注释创建的。

5.4 lint 消息

lint 的大多数消息很简单，对于它们诊断的每个问题打印一行语句。包含的文件中检测到的错误由编译器报告多次，但是仅由 lint 报告一次，而无论其它源文件中包含该文件多少次。文件之间存在不一致时会发出复合消息，并且在少数情况下，文件内部存在问题时也会发出复合消息。单个消息描述所检查的文件中的各个问题。如果使用 lint 过滤器 （参见第 5-28 页上的第 5.6.2 节 “lint 库”）要求对每个问题打印消息，可通过使用 -s 选项调用 lint 将复合诊断转换为简单类型。

lint 的消息写入 stderr。

5.4.1 用于禁止消息的选项

您可以使用几个 lint 选项禁止 lint 诊断消息。可使用后跟一个或多个 *tag* 的 -erroff 选项禁止消息。这些助记符标记可使用 -errtags=yes 选项来显示。

下表列出禁止 lint 消息的选项。

表 5-7 用于禁止消息的 lint 选项

选项	禁止的消息
-a	赋值导致隐式范围缩小转换 向较长整型转换可能导致错误地进行符号扩展
-b	未执行到的语句 （无法执行到的中断和空语句）
-h	赋值运算符 "=" 出现在需要 "==" 的位置 运算符的操作数为常量："!" case 语句失败 指针强制类型转换导致不正确的对齐 优先级可能不明确；加括号 语句无结果：if 语句无结果：else
-m	声明为全局，应为静态
-erroff=*tag*	由 *tag* 指定的一个或多个 lint 消息
-u	名称已定义但从未使用 名称已使用但未定义
-v	参数未在函数中使用
-x	名称已声明但从未使用或定义

5.4.2 lint 消息格式

通过使用某些选项， lint 程序可以显示精确的源文件行以及指向发生错误的行位置的指针。启用此功能的选项是 -errfmt=*f*。如果设置此选项， lint 提供以下信息：

- 源代码行和位置
- 宏展开
- 有错误倾向的栈

例如，以下程序 Test1.c 包含一个错误。

```
1 #include <string.h>
2 static void cpv(char *s, char* v, unsigned n)
3 { int i;
4   for (i=0; i<=n; i++){
5        *v++ = *s++;}
6 }
7 void main(int argc, char* argv[])
8 {
9     if (argc != 0){
10        cpv(argv[0], argc, strlen(argv[0]));}
11}
```

使用选项对 Test1.c 使用 lint：

```
% lint -errfmt=src -Nlevel=2 Test1.c
```

产生以下输出：

```
      |static void cpv(char *s, char* v, unsigned n)
      |             ^  line 2, Test1.c
      |
      |         cpv(argv[0], argc, strlen(argv[0]));
      |                          ^  line 10, Test1.c
警告：不正确的指针/整型组合：arg #2
      |
      |static void cpv(char *s, char* v, unsigned n)
      |                                  ^  line 2, Test1.c
      |
      |cpv(argv[0], argc, strlen(argv[0]));
      |                           ^ line 10, Test1.c
      |
      |        *v++ = *s++;
      |         ^  line 5, Test1.c
警告：使用了用不可靠方法产生的指针
       v defined at Test1.c(2)::Test1.c(5)
         call stack:
             main()                ,Test1.c(10)
             cpv()                 ,Test1.c(5)
```

第一个警告指示互相矛盾的两个源代码行。第二个警告显示调用栈以及导致错误的控制流。

另一个程序 Test2.c 包含另一个不同的错误：

```
1 #define AA(b) AR[b+1]
2 #define B(c,d) c+AA(d)
3
4 int x=0;
5
6 int AR[10]={1,2,3,4,5,6,77,88,99,0};
7
8 main()
9  {
10  int y=-5, z=5;
11  return B(y,z);
12 }
```

使用选项对 Test2.c 使用 lint：

```
% lint -errfmt=macro Test2.c
```

产生以下输出，并显示宏替换的步骤：

```
       | return B(y,z);
       |        ^  line 11, Test2.c
       |
       |#define B(c,d) c+AA(d)
       |                  ^  line 2, Test2.c
       |
       |#define AA(b) AR[b+1]
       |                    ^  line 1, Test2.c
错误：未定义的符号：1
|
       |    return B(y,z);
       |           ^  line 11, Test2.c
       |
       |#define B(c,d) c+AA(d)
       |                  ^  line 2, Test2.c
       |
       |#define AA(b) AR[b+1]
       |                    ^  line 1, Test2.c
变量在设置之前使用：1
lint: Test2.c 有错误；未创建输出文件
lint: pass2 未运行 - Test2.c 有错误
```

5.5 lint 指令

5.5.1 预定义值

以下预定义在所有模式下均有效：

- __sun
- __unix
- __lint
- __SUNPRO_C=0x550
- __`name -s`_`name -r` （示例：__SunOS_5_7）
- __RESTRICT （仅适用于 -Xa 和 -Xt 模式）
- __sparc *(SPARC)*
- __i386 *(Intel)*
- __BUILTIN_VA_ARG_INCR
- __SVR4
- __sparcv9 (-Xarch=v9)

以下预定义在 -Xc 模式下无效：

- sun
- unix
- sparc *(SPARC)*
- i386 *(Intel)*
- lint

5.5.2 指令

现有注释支持 /*...*/ 形式的 lint 指令，但将来的注释不支持这些指令。建议对所有注释使用源代码注释形式的指令 NOTE(...)。

通过包含文件 note.h 指定源代码注释形式的 lint 指令，例如：

```
#include <note.h>
```

Lint 与其它多种工具共享源代码注释方案。当您安装 Sun C 编译器时，您也自动安装了文件 /usr/lib/note/SUNW_SPRO-lint，它包含 LockLint 理解的所有注释的名称。然而， Sun C 源代码检验器 lint 也检查 /usr/lib/note 和 /opt/SUNWspro/prod/lib/note 中的所有文件的注释是否全部有效。

通过设置环境变量 NOTEPATH，您可以指定除 /usr/lib/note 之外的位置，如下所示：

```
setenv NOTEPATH $NOTEPATH:other_location
```

下表列出 lint 指令及其操作。

表 5-8 lint 指令

指令	操作
NOTE(ALIGNMENT(*fname,n*)) where *n*=1, 2, 4, 8, 16, 32, 64, 128	使 lint 以 *n* 字节设置后面的函数结果对齐。例如，malloc() 被定义为在实际返回字 （甚至双字）对齐的指针时返回一个 char* 或 void*。 禁止以下消息： • **不正确的对齐**
NOTE(ARGSUSED(n)) /*ARGSUSEDn*/	该指令的作用类似于下一个函数的 -v 选项。 禁止对该指令之后的参数定义中除前 *n* 个参数之外的各个参数发出以下消息。缺省值为 0。对于 NOTE 格式，必须指定 *n*。 • **参数未在函数中使用**
NOTE(ARGUNUSED (*par_name[,par_name...]*))	使 lint 不检查提到的参数的用法 （此选项仅适用于下一个函数）。 禁止对 NOTE 或指令中列出的各个参数发出以下消息。 • **参数未在函数中使用**
NOTE(CONSTCOND) /*CONSTCOND*/	禁止发出关于条件表达式的常量操作数的警告消息。禁止对该指令之后的构造发出以下消息。亦作 NOTE(CONSTANTCONDITION) 或 /* CONSTANTCONDITION */。 **常量在条件上下文中** **运算符的操作数为常量：** "!" **逻辑表达式总是为 false：操作符** "&&" **逻辑表达式总是为 true：操作符** "||"
NOTE(EMPTY) /*EMPTY*/	禁止在 if 语句出现 null 语句结果时发出警告信息。该指令应放在测试表达式之后和分号之前。当空 if 语句后跟有效的 else 语句时，提供该指令以支持空 if 语句。它禁止在出现空 else 结果时发出消息。 禁止在 if 的控制表达式与分号之间插入时发出以下消息。 • **语句无结果：** else 在 else 与分号之间插入时； • **语句无结果：** if

表 **5-8** lint 指令 *（续）*

指令	操作
NOTE(FALLTHRU) /*FALLTHRU*/	禁止在 case 或 default 带标签语句失败时发出警告消息。此指令应放在标签之前并紧挨着标签。 禁止对该指令之后的 case 语句发出以下消息。亦作 NOTE(FALLTHROUGH) 或 /* FALLTHROUGH */。 • **case 语句失败**
NOTE(LINTED (msg)) /*LINTED [msg]*/	禁止除处理未使用的变量或函数的警告之外的任何文件内警告。此指令应放在发生 lint 警告的位置之前并紧挨着该位置的行上。-k 选项改变 lint 处理此指令的方式。lint 不禁止消息，而是打印注释中包含的附加消息（如果有）。此指令与 -s 选项一起用于 post-lint 过滤。 未调用 -k 时，禁止对该指令之后的代码行发出关于文件内问题的各个警告，但不包括以下警告： • **参数未在函数中使用** • **声明未在块中使用** • **已设置但未在函数中使用** • **static 未使用** • **变量未在函数中使用** 忽略 *msg*。
NOTE(LINTLIBRARY) /*LINTLIBRARY*/	调用 -o 时，仅将该指令之后的 .c 文件中的定义写入库 .ln 文件。此指令禁止发出关于此文件中存在未使用的函数和函数参数的警告消息。
NOTE(NOTREACHED) /*NOTREACHED*/	在适当的点，停止关于无法执行到的代码的注释。此注释通常放在 exit(2) 等函数的调用之后。 禁止在函数末尾该指令之后存在右花括号时发出以下消息。 • **语句执行不到** 对于该指令之后的无法执行到的语句； • **case 语句失败** 对于该指令之后无法从前面的 case 执行到的 case； • **函数未返回值即转向底部**
NOTE(PRINTFLIKE(*n*)) NOTE(PRINTFLIKE(*fun_name,n)*) /*PRINTFLIKE*n**/	将它之后的函数定义的第 *n* 个参数被看作 [fs]printf() 格式字符串，并在其余参数与转换规范之间不匹配时发出以下消息。在缺省情况下，如果标准 C 库提供的 [fs]printf() 函数的调用中存在错误，lint 会发出这些警告。 对于 NOTE 格式，必须指定 *n*。 • **格式错误的格式字符串** 对于该参数中的无效转换规范，以及函数参数类型与格式不一致； • **格式的参数过少** • **格式的参数过多**

表 5-8 lint 指令 *（续）*

指令	操作
NOTE(PROTOLIB(*n*)) /*PROTOLIB*n*/	当 *n* 为 1，并且使用 NOTE(LINTLIBRARY) 或 /* LINTLIBRARY */ 时，仅将该指令之后的 .c 文件中的函数原型声明写入库 .ln 文件。缺省值为 0，它取消该进程。 对于 NOTE 格式，必须指定 *n*。
NOTE(SCANFLIKE(*n*)) NOTE(SCANLIKE*(fun_name,n*)) /*SCANFLIKE*n**/	与 NOTE(PRINTFLIKE(*n*)) 或 /* PRINTFLIKE*n* */ 相同，只是函数定义的第 *n* 个参数被看作 [fs]scanf() 格式字符串。缺省情况下，标准 C 库提供的 [fs]scanf() 函数的调用中存在错误时，lint 会发出警告。 对于 NOTE 格式，必须指定 *n*。
NOTE(VARARGS(*n*)) NOTE(VARARGS(*fun_name,n*)) /*VARARGS*n**/	禁止对以下函数声名中可变数量的参数执行常规检查。检查前 *n* 个参数的数据类型；*n* 缺少时则被视为 0。建议在新代码或更新的代码中，在定义中使用省略号 (...) 终结符。 对于其定义在该指令之后的函数，禁止在调用带有 *n* 个或更多参数的函数时发出以下消息。对于 NOTE 格式，必须指定 *n*。 • **用不同的参数数目调用了函数**

5.6 lint 参考和示例

本节提供关于 lint 的参考信息，包括由 lint 执行的检查、 lint 库以及 lint 过滤器。

5.6.1 由 lint 执行的诊断

对以下三种主要条件发出特定于 lint 的诊断消息：不一致的使用、不可移植的代码以及可疑的构造。在本节中，我们将研究在每种条件下 lint 的行为的示例，并针对它们引起的问题提供可能的解决方法建议。

5.6.1.1 一致性检查

在文件内部以及各文件之间检查使用变量、参数和函数的不一致性。一般说来，对原型的使用、声明和参数执行的检查与 lint 对旧式样函数的检查相同。如果您的程序不使用函数原型， lint 将比编译器更严格地检查函数的每个调用中参数的数量和类型。lint 还标识 [fs]printf() 和 [fs]scanf() 控制字符串中的转换规范和参数的不匹配。

示例：

- 在文件内部， lint 标记未向调用函数返回值即转向底部的非 void 函数。以前，程序员通常通过省略返回类型指明某个函数不应返回值：fun() {}。该惯例对编译器没有任何意义，它将 fun() 视为具有返回类型 int。使用返回类型 void 声明函数以消除问题。
- 在文件之间， lint 检测非 void 函数不返回值但由于它在某个表达式中有值而仍被使用的情况 — 相对立的问题，返回值的函数在后续调用中有时或总是被忽略。当值*总是*被忽略时，可能表示函数定义无效率。当它*有时*被忽略时，可能是式样错误（通常，测试时不作为错误条件）。如果您无需检查 strcat()、 strcpy() 和 sprintf() 等字符串函数的返回值，或者无需检查 printf() 和 putchar() 等输出函数，则将不合适的调用强制转换为 void 类型。
- lint 标识已声明但未使用或定义、已使用但未定义或者已定义但未使用的变量或函数 当 lint 应用于某个集合中要一起装入的某些文件但不是全部文件时，它会产生关于出现以下情况的函数和变量的错误消息：
  - 在那些文件中声明，但在其它地方定义或使用
  - 在那些文件中使用，但在其它地方定义
  - 在那些文件中定义，但在其它地方使用

  调用 -x 选项以禁止第一种错误消息，调用 -u 以禁止后两种错误消息。

5.6.1.2 可移植性检查

某些不可移植的代码由 lint 在其缺省行为中标记，并且当使用 -p 或 -Xc 调用 lint 时诊断其它一些情况。后者导致 lint 检查不符合 ISO C 标准的构造。有关 -p 和 -Xc 发出的消息，请参见第 5-28 页上的第 5.6.2 节 “lint 库”。

示例：

- 在某些 C 语言实现中，未显式声明 signed 或 unsigned 的字符变量被视为带符号的数量，范围一般从 -128 到 127。在其它实现中，它们被视为非负数量，范围一般从 0 到 255。因此测试：

```
char c;
c = getchar();
if (c == EOF) ...
```

其中 EOF 的值为 -1，在字符变量取非负值的机器上总是失败。使用 -p 调用的 lint 会检查暗示*无格式* char 可取负值的所有比较。然而，在以上示例中，将 c 声明为 signed char 可避免执行诊断，但不能避免出现该问题。这是因为 getchar() 必须返回所有可能的字符和一个不同的 EOF 值，因此 char 不能存储其值。此示例可能是实现定义的符号扩展最常见的示例，我们引用该示例显示如何巧妙地应用 lint 的可移植性选项帮助您发现与可移植性无关的错误。在任何情况下，将 c 声明为 int。

- 位字段出现类似的问题。将常量值赋给位字段时，该字段太小而无法容纳该值。在将 int 类型的位字段视为无符号数量的机器上， int x:3 允许值的范围从 0 到 7；而在将它们视为带符号数量的机器上，允许值的范围从 -4 到 3。但是，声明为 int 类型的一个三位字段在后一种机器上不能容纳值 4。使用 -p 调用的 lint 会标记除 unsigned int 或 signed int 之外的所有位字段类型。这些只是*可移植的*位字段类型。编译器支持 int、 char、 short 和 long 位字段类型，它们可能为 unsigned、 signed 或*无格式*。编译器还支持 enum 位字段类型。
- 当较长的类型赋值给较短的类型时，会出现错误。如果有效位被截断，则失去准确性：

```
short s;
long l;
s = l;
```

lint 在缺省情况下标记所有此类赋值；可通过调用 -a 选项禁止诊断。请记住，当您使用此选项或任何其它选项调用 lint 时，可能会禁止其它诊断。请查看第 5-28 页上的第 5.6.2 节 “lint 库”中的列表，以了解禁止多项诊断的选项。

- 一个对象类型指针向具有更严格对齐要求的对象类型指针的强制类型转换可能不可移植。 lint 标记以下代码：

```
int *fun(y)
char *y;
{
    return(int *)y;
}
```

这是因为在大多数机器上， int 不能在一个任意字节边界上开始，而 char 则可以。您可以通过调用 lint 和 -h 禁止诊断，尽管再次可能禁止其它消息。但是最好通过使用通用指针 void * 消除该问题。

- ISO C 未定义复杂表达式的求值顺序。也就是说，如果在由于某个表达式的求值而更改某个变量时，函数调用、嵌套赋值语句或增减运算符引起副作用，副作用发生的顺序极大程度上依赖于机器。在缺省情况下， `lint` 会标记由于副作用而更改并且在同一表达式的其它地方使用的任何变量：

```
int a[10];
main()
{
    int i = 1;
    a[i++] = i;
}
```

在此示例中， `a[1]` 的值在使用一个编译器时可能为 1，在使用另一个编译器时为 2。如果在逻辑运算符 `&&` 的位置错误地使用了按位逻辑运算符 `&`，同会引起该诊断：

```
if ((c = getchar()) != EOF & c != '0')
```

5.6.1.3 可疑的构造

`lint` 会标记不能表示程序员意图的各种合法构造。示例：

- `unsigned` 变量总是具有非负值。因此测试：

```
unsigned x;
if (x < 0) ...
```

总是失败。测试：

```
unsigned x;
if (x > 0) ...
```

等价于：

```
if (x != 0) ...
```

这可能不是预期的操作。lint 会标记 unsigned 变量与负常量或 0 的可疑比较。要将 unsigned 变量与负数的位模式比较，请执行强制类型转换，将它转换为 unsigned：

```
if (u == (unsigned) -1) ...
```

或者使用 U 后缀：

```
if (u == -1U) ...
```

- lint 会标记用于预期出现副作用的上下文但没有副作用的表达式 — 也就是说，表达式不能表示程序员的意图。它在应该出现赋值运算符的位置发现等号运算符时发出附加警告 — 也就是说，预期出现副作用：

```
int fun()
{
    int a, b, x, y;
    (a = x) && (b == y);
}
```

- lint 提醒您注意给混合了逻辑运算符和按位运算符（特别是 &、 |、 ^、 <<、 >>）的表达式加上括号，运算符优先级的误解会引起不正确的结果。例如，因为按位运算符 & 的优先级低于逻辑运算符 ==，所以表达式：

```
if (x & a == 0) ...
```

求值如下：

```
if (x & (a == 0)) ...
```

这很有可能不是您的意图。使用 -h 调用的 lint 禁止该诊断。

5.6.2 lint 库

您可以使用 lint 库检查您的程序是否与您已在其中调用的库函数 （函数返回类型的声明、函数预期的参数的数量和类型，等等）兼容。标准 lint 库与 C 编译系统提供的库对应，并且通常存储在系统上的标准位置。按照惯例， lint 库的名称形式为 llib-lx.ln。

lint 标准 C 库 llib-lc.ln 缺省情况下附加至 lint 命令行；可通过调用 -n 选项禁止其兼容性检查。其它 lint 库作为 -l 的参数进行访问。即：

```
% lint -lx file1.c file2.c
```

指示 lint 检查 file1.c 和 file2.c 中的函数和变量的用法是否与 lint 库 llib-l*x*.ln 兼容。仅由定义组成的库文件完全作为普通源文件和普通 .ln 文件处理，只是在库文件中的使用不一致或者在库文件中定义但未在源文件中使用的函数和变量不会发出错误消息。

要创建您自己的 lint 库，请在 C 源文件的最前面插入指令 NOTE(LINTLIBRARY)，然后使用 -o 选项并将库名指定为 -l 为该文件调用 lint：

```
% lint -ox file1.c file2.c
```

导致仅将以 NOTE(LINTLIBRARY) 开头的源文件中的定义写入文件 llib-l*x*.ln。（请注意，lint -o 类似于 cc -o。）可使用包含函数原型声明的文件以同样方式创建库，只是 NOTE(LINTLIBRARY) 和 NOTE(PROTOLIB(*n*)) 必须插入到声明文件的最前面。如果 *n* 为 1，则原型声明写入库 .ln 文件，这与旧式样定义相同。如果 *n* 为 0 （缺省值），则取消该进程。使用 -y 调用 lint 是创建 lint 库的另一种方法。命令行：

```
% lint -y -ox file1.c file2.c
```

导致该行中指定的每个源文件被视为以 NOTE(LINTLIBRARY) 开头，并且只有其定义被写入 llib-l*x*.ln。

缺省情况下，lint 在标准位置查找 lint 库。要指示 lint 在不是标准位置的目录中查找 lint 库，请使用 -L 选项指定目录路径：

```
% lint -Ldir -lx file1.c file2.c
```

在增强模式下，lint 生成 .ln 文件，这些文件存储的信息比在基本模式下生成的 .ln 文件存储的信息多。在增强模式下， lint 可以读取并理解由基本或增强 lint 模式生成的所有 .ln 文件。在基本模式下， lint 只能读取并理解使用基本 lint 模式生成的 .ln 文件。

缺省情况下， lint 使用 /usr/lib 目录中的库。这些库采用基本 lint 格式。您可以运行一次 makefile，并以新格式创建增强 lint 库，从而使增强 lint 更有效地工作。要运行 makefile 并创建新库，请输入命令：

```
% cd /opt/SUNWspro/prod/src/lintlib; make
```

其中 /opt/SUNWspro/prod 为安装目录。运行 makefile 之后， lint 在增强模式下使用新库，而不是使用 /usr/lib 目录中的库。

在搜索标准位置之前搜索指定的目录。

5.6.3 lint 过滤器

lint 过滤器是特定于项目的后处理程序，通常使用一个 awk 脚本或类似程序读取 lint 的输出，并丢弃您的项目认为*没有*标识真正问题的消息 — 例如字符串函数，返回值有时或总是被忽略。当 lint 选项和指令未提供对输出的足够控制时， lint 过滤器生成定制诊断报告。

lint 的两个选项在开发过滤器的过程中特别有用：

- 使用 -s 调用 lint 导致复合诊断转换为简单诊断，对诊断的每个具体问题发出一行消息。这个容易进行语法分析的消息格式适合由 awk 脚本进行分析。
- 使用 -k 调用 lint 导致您在源文件中编写的注释在输出中打印，并且有助于将项目决策形成文档并指定后处理程序的行为。在后一个实例中，如果注释标识预期的 lint 消息，并且报告的消息相同，则会过滤掉该消息。要使用 -k，请在您要注释的代码前面的行中插入 NOTE(LINTED(*msg*)) 指令，其中 *msg* 是指在使用 -k 调用 lint 时将打印的注释。

  请参考表 5-8 中的指令列表，以了解对于包含 NOTE(LINTED(*msg*)) 的文件*未*调用 -k 时， lint 执行的操作的说明。

第 6 章

基于类型的别名分析

本文档解释如何使用 -xalias_level 选项和几个新的 pragma 使编译器能够执行基于类型的别名分析和优化。您可以使用这些扩展功能表示关于 C 程序中使用指针方法的基于类型的信息。 C 编译器又可以使用此信息对程序中基于指针的内存引用更好地进行别名歧义消除并且效果显著。

有关此命令的语法的详细说明，请参见第 A-28 页上的第 A.3.63 节 “-xalias_level[=l]”。另外，有关 lint 程序的基于类型别名分析功能的说明，请参见第 5-14 页上的第 5.3.37 节 “-Xalias_level[=l]”。

6.1 介绍基于类型的分析

您可以使用 -alias_level 选项指定七个别名级别之一。每个级别指定一组关于您在 C 程序中使用指针的方法的属性。

当您使用 -xalias_level 选项的较高级别进行编译时，编译器会对您的代码中的指针进行更广泛的假定。当编译器产生较少假定时，您有更大的编程自由。但是，这些狭义假定产生的优化可能不会导致运行时性能的显著提高。如果您依照 -xalias_level 选项的更高级别的编译器假定进行编码，则更有可能使产生的优化提高运行环境性能。

-xalias_level 选项指定应用于每个转换单元的别名级别。在越详细越有益的情况下，您可以使用新的 pragma 覆盖已生效的别名级别，以便可以明确指定转换单元中个体类型或指针变量之间的别名关系。如果转换单元中指针的使用对应于某个可用别名级别，但是一些特定指针变量的使用方法是某个可用级别不允许的不规则方法，这些 pragma 非常有用。

6.2 使用 Pragma 以便更好地控制

在基于类型的分析由于更详细而受益的情况下，您可以使用以下 pragma 覆盖已生效的别名级别，并指定转换单元中个体类型或指针变量之间的别名关系。如果转换单元中指针的使用与某个可用别名级别一致，但是一些特定指针变量的使用方法是某个可用级别不允许的不规则方法，这些 pragma 非常有益。

注 – 必须在 pragma 之前声明命名的类型或变量，否则会发出警告消息并忽略 pragma。如果 pragma 出现在其含义所适用的第一个内存引用之后，则程序的结果未定义。

下列术语用于 pragma 定义。

术语	含义
级别	第 A-28 页上的第 A.3.63 节 “`-xalias_level[=l]`”下面列出的任何别名级别。
类型	以下任何类型： • `char`、`short`、`int`、`long`、`long long`、`float`、`double`、`long double` • `void`，表示所有指针类型 • `typedef` *name*，它是 `typedef` 声明中定义的类型的名称 • `struct` *name*，它是后面有 *struct tag* 名称的关键字 `struct` • `union`，它是后面有 *union tag* 名称的关键字 `union`
pointer_name	转换单元中指针类型的任何变量的名称。

6.2.0.1 `#pragma alias_level` *level* (*list*)

使用以下七种别名级别之一替换 *level*：`any`、`basic`、`weak`、`layout`、`strict`、`std` 或 `strong`。您可以使用单一类型或以逗号分隔的类型列表替换 *list*，也可以使用单一指针或以逗号分隔的指针列表替换 *list*。例如，您可以按以下方式发出 `#pragma alias_level`：

- `#pragma alias_level` *level* (`type` [, `type`])
- `#pragma alias_level` *level* (`pointer` [, `pointer`])

此 pragma 指定，指示的别名级别应用于所列类型的转换单元的所有内存引用，或者应用于其中某个命名指针变量正在被非关联化的转换单元的所有非关联化。

如果您指定多个要应用于特定非关联化的别名级别，则指针名称 （如果有）应用的级别优先于所有其它级别。类型名称 （如果有）应用的级别优先于选项应用的级别。在以下示例中，如果在编译程序时将 #pragma alias_level 设置得比 any 高，则 std 级别应用于 *p*。

```
typedef int * int_ptr;
int_ptr p;
#pragma alias_level strong (int_ptr)
#pragma alias_level std (p)
```

6.2.0.2 #pragma alias (type, type [, type]…)

此 pragma 指定所有列出的类型互为别名。在以下示例中，编译器假定间接访问 *pt 的别名为间接访问 *pf。

```
#pragma alias (int, float)
int *pt;
float *pf;
```

6.2.0.3 #pragma alias (pointer, pointer [, pointer]…)

此 pragma 指定，在任何命名指针变量的任何非关联化点，正被非关联化的指针值可以指向与任何其它命名指针变量相同的对象。但是，指针并不仅限于命名变量中包含的对象，可以指向列表中未包含的对象。此 pragma 覆盖任何应用别名级别的别名假定。在以下示例中，该 pragma 之后对 p 和 q 的任何间接访问无论是什么类型，均被视为别名。

```
#pragma alias(p, q)
```

6.2.0.4 #pragma may_point_to (pointer, variable [, variable]…)

此 pragma 指定，在命名指针变量的任何非关联化点，正被非关联化的指针值可以指向任何命名变量中包含的对象。但是，指针并不仅限于命名变量中包含的对象，可以指向列表中未包含的对象。此 pragma 覆盖应用的任何别名级别的别名假定。在以下示例中，编译器假定对 *p 的任何间接访问的别名是任何直接访问 a、 b 和 c。

```
#pragma alias may_point_to(p, a, b, c)
```

6.2.0.5 #pragma noalias (type, type [, type]...)

此 pragma 指定列出的类型不互为别名。在以下示例中，编译器假定对 *p 的任何间接访问不将间接访问 *ps 作为别名。

```
struct S {
   float f;
   ...} *ps;

#pragma noalias(int, struct S)
int *p;
```

6.2.0.6 #pragma noalias (pointer, pointer [, pointer]...)

此 pragma 指定，在任何命名指针变量的任何非关联化点，正被非关联化的指针值不指向与任何其它命名指针变量相同的对象。此 pragma 覆盖所有其它应用别名级别。在以下示例中，编译器假定无论两个指针是什么类型，对 *p 的任何间接访问均不将间接访问 *q 作为别名。

```
#pragma noalias(p, q)
```

6.2.0.7 #pragma may_not_point_to (pointer, variable [, variable]...)

此 pragma 指定，在命名指针变量的任何非关联化点，正被非关联化的指针值不指向任何命名变量中包含的对象。此 pragma 覆盖所有其它应用别名级别。在以下示例中，编译器假定对 *p 的任何间接访问不将直接访问 a、 b 或 c 作为别名。

```
#pragma may_not_point_to(p, a, b, c)
```

6.3 使用 lint 检查

lint 程序识别与编译器的 -xalias_level 命令同级别的基于类型的别名歧义消除。lint 程序还识别与本章中说明的基于类型的别名歧义消除相关的 pragma。有关 lint -Xalias_level 命令的详细解释，请参见第 5-14 页上的第 5.3.37 节“-Xalias_level[=l]”。

lint 检测以下四种情况并生成警告：

- 将标量指针强制转换为结构指针
- 将空指针强制转换为结构指针
- 将结构字段强制转换为标量指针
- 将结构指针强制转换为 -Xalias_level=strict 级别上没有显式别名的结构指针。

6.3.1 标量指针向结构指针的强制类型转换

在以下示例中，整型指针 p 强制转换为 struct foo 类型的指针。如果 int -Xalias_level=weak （或更高），这将产生错误。

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
int *p;

void main()
{
    f = (struct foo *)p; /* 标量指针向结构指针的强制类型转换错误 */
}
```

6.3.2 空指针向结构指针的强制类型转换

在以下示例中，空指针 vp 强制转换为结构指针。如果 lint -Xalias_level=weak（或更高），这将产生警告。

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
void *vp;

void main()
{
    f = (struct foo *)vp; /* 空指针向结构指针的强制类型转换错误 */
}
```

6.3.3 结构字段向结构指针的强制类型转换

在以下示例中，结构成员 foo.b 的地址正在被强制转换为结构指针，然后赋值给 p。如果 lint -Xalias_level=weak（或更高），这将产生警告。

```
struct foo p{
    int a;
    int b;
  };

struct foo *f1;
struct foo *f2;

void main()
{
    f2 = (struct foo *)&f1->b; /* 标量指针向结构指针的强制类型转换错误
*/
}
```

6.3.4 要求显式别名

在以下示例中， struct `fooa` 类型的指针 f1 正在被强制转换为 struct `foob` 类型的指针。如果 lint `-Xalias_level=strict` （或更高），则除非结构类型相同（相同类型的相同数目的字段），否则此类强制类型转换要求显式别名。此外，在别名级别 `standard` 和 `strong` 上，假定标记必须匹配才能出现别名。在给 f1 赋值前使用 `#pragma alias` (`struct fooa`, `struct foob`)， lint 停止产生警告。

```
struct fooa {
    int a;
};

struct foob {
    int b;
};

struct fooa *f1;
struct foob *f2;

void main()
{
    f1 = (struct fooa *)f2; /* 需要显式别名警告 */
}
```

6.4 内存引用约束的示例

本节提供可能会在您的源文件中出现的代码的示例。每个示例后面讨论编译器关于基于类型的分析的应用级别所指示的代码的假定。

考虑以下代码。可以使用不同的别名级别对它进行编译，以说明显示类型的别名关系。

编码示例 6-1

```
struct foo {
       int f1;
       short f2;
       short f3;
       int f4;
} *fp;

struct bar {
       int b1;
       int b2;
       int b3;
} *bp;

int *ip;
short *sp;
```

如果使用 -xalias_level=any 选项编译编码示例 6-1，则编译器将认为以下间接访问互为别名：

*ip, *sp, *fp, *bp, fp->f1, fp->f2, fp->f3, fp->f4, bp->b1, bp->b2, bp->b3

如果使用 -xalias_level=basic 选项编译编码示例 6-1，则编译器将认为以下间接访问互为别名：

*ip, *bp, fp->f1, fp->f4, bp->b1, bp->b2, bp->b3

另外， *sp、 fp->f2 和 fp->f3 可以互为别名， *sp 和 *fp 可以互为别名。

但是，在 -xalias_level=basic 条件下，编译器作出以下假定：

- *ip 不将 *sp 作为别名。
- *ip 不将 fp->f2 和 fp->f3 作为别名。
- *sp 不将 fp->f1、 fp->f4、 bp->b1、 bp->b2 和 bp->b3 作为别名。

由于两个间接访问的访问类型是不同的基本类型，因此编译器作出这些假定。

如果使用 -xalias_level=weak 选项编译编码示例 6-1，则编译器假定以下别名信息：

- *ip 可以将 *fp、 fp->f1、 fp->f4、 *bp、 bp->b1、 bp->b2 和 bp->b3 作为别名。
- *sp 可以将 *fp、 fp->f2 和 fp->f3 作为别名。
- fp->f1 可以将 bp->b1 作为别名。
- fp->f4 可以将 bp->b3 作为别名。

由于 f1 是结构中偏移为 0 的字段，而 b2 是结构中偏移为 4 个字节的字段，因此编译器假定 fp->fp1 不将 bp->b2 作为别名。同样，编译器假定 fp->f1 不将 bp->b3 作为别名， fp->f4 不将 bp->b1 或 bp->b2 作为别名。

如果使用 -xalias_level=layout 选项编译编码示例 6-1，则编译器假定以下信息：

- *ip 可以将 *fp、 *bp、 fp->f1、 fp->f4、 bp->b1、 bp->b2 和 bp->b3 作为别名。
- *sp 可以将 *fp、 fp->f2 和 fp->f3 作为别名。
- fp->f1 可以将 bp->b1 和 *bp 作为别名。
- *fp 和 *bp 可以互为别名。

由于 f4 和 b3 不是 foo 和 bar 的公共初始序列中的相应字段，因此 fp->f4 不将 bp->b3 作为别名。

如果使用 -xalias_level=strict 选项编译编码示例 6-1，则编译器假定以下别名信息：

- *ip 可以将 *fp、 fp->f1、 fp->f4、 *bp、 bp->b1、 bp->b2 和 bp->b3 作为别名。
- *sp 可以将 *fp、 fp->f2 和 fp->f3 作为别名。

如果 -xalias_level=strict，则由于当忽略字段名时， foo 和 bar 不相同，因此编译器假定 *fp、*bp、fp->f1、fp->f2、fp->f3、fp->f4、bp->b1、bp->b2 和 bp->b3 并不互为别名。但是， fp 将 fp->f1 作为别名， bp 将 bp->b1 作为别名。

如果使用 -xalias_level=std 选项编译编码示例 6-1，则编译器假定以下别名信息：

- *ip 可以将 *fp、 fp->f1、 fp->f4、 *bp、 bp->b1、 bp->b2 和 bp->b3 作为别名。
- *sp 可以将 *fp、 fp->f2 和 fp->f3 作为别名。

但是，由于当考虑字段名时， foo 和 bar 不相同，因此 fp->f1 不将 bp->b1、 bp->b2 或 bp->b3 作为别名。

如果使用 -xalias_level=strong 选项编译编码示例 6-1，则编译器假定以下别名信息：

- 由于指针 （如 *ip）不应指向结构的内部，因此 *ip 不将 fp->f1、 fp->f4、 bp->b1、 bp->b2 和 bp->b3 作为别名。
- 同样， *sp 不将 fp->f1 或 fp->f3 作为别名。
- 由于类型不同， *ip 不将 *fp、 *bp 和 *sp 作为别名。
- 由于类型不同， *sp 不将 *fp、 *bp 和 *sp 作为别名。

考虑以下源代码示例。当使用不同的别名级别编译时，它说明显示的类型的别名关系。

编码示例 6-2

```
struct foo {
    int f1;
    int f2;
    int f3;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;
```

如果使用 -xalias_level=any 选项编译编码示例 6-2，则编译器假定以下别名信息：

由于在 -xalias_level=any 级别上任何两个内存访问互为别名，因此 *fp、 *bp、 fp->f1、 fp->f2、 fp->f3、 bp->b1、 bp->b2 和 bp->b3 均可以互为别名。

如果使用 -xalias_level=basic 选项编译 编码示例 6-2，则编译器假定以下别名信息：

*fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2 和 bp->b3 均可以互为别名。在本示例中，由于所有结构字段均为基本类型，因此任何两个使用指针 *fp 和 *bp 的字段访问均可以互为别名。

如果使用 -xalias_level=weak 选项编译编码示例 6-2，则编译器假定以下别名信息：

- *fp 和 *fp 可以互为别名。
- fp->f1 可以将 bp->b1、 *bp 和 *fp 作为别名。
- fp->f2 可以将 bp->b2、 *bp 和 *fp 作为别名。
- fp->f3 可以将 bp->b3、 *bp 和 *fp 作为别名。

但是， -xalias_level=weak 强加以下限制：

- 由于 f1 的偏移为零，与 b2 的偏移 （四个字节）和 b3 的偏移 （八个字节）不同，因此 fp->f1 不将 bp->b2 或 bp->b3 作为别名。
- 由于 f2 的偏移为四个字节，与 b1 的偏移 （零字节）和 b3 的偏移 （八个字节）不同，因此 fp->f2 不将 bp->b1 或 bp->b3 作为别名。
- 由于 f3 的偏移为八个字节，与 b1 的偏移 （零字节）和 b2 的偏移 （四个字节）不同，因此 fp->f3 不将 bp->b1 或 bp->b2 作为别名。

如果使用 -xalias_level=layout 选项编译编码示例 6-2，则编译器假定以下别名信息：

- *fp 和 *bp 可以互为别名。
- fp->f1 可以将 bp->b1、 *bp 和 *fp 作为别名。
- fp->f2 可以将 bp->b2、 *bp 和 *fp 作为别名。
- fp->f3 可以将 bp->b3、 *bp 和 *fp 作为别名。

但是， -xalias_level=layout 强加以下限制：

- 由于字段 f1 对应于 foo 和 bar 的公共初始序列中的字段 b1，因此 fp->f1 不将 bp->b2 或 bp->b3 作为别名。
- 由于 f2 对应于 foo 和 bar 的公共初始序列中的字段 b2，因此 fp->f2 不将 bp->b1 或 bp->b3 作为别名。
- 由于 f3 对应于 foo 和 bar 的公共初始序列中的字段 b3，因此 fp->f3 不将 bp->b1 或 bp->b2 作为别名。

如果使用 -xalias_level=strict 选项编译编码示例 6-2，则编译器假定以下别名信息：

- *fp 和 *bp 可以互为别名。
- fp->f1 可以将 bp->b1、 *bp 和 *fp 作为别名。
- fp->f2 可以将 bp->b2、 *bp 和 *fp 作为别名。
- fp->f3 可以将 bp->b3、 *bp 和 *fp 作为别名。

但是， -xalias_level=strict 强加以下限制：

- 由于字段 f1 对应于 foo 和 bar 的公共初始序列中的字段 b1，因此 fp->f1 不将 bp->b2 或 bp->b3 作为别名。
- 由于 f2 对应于 foo 和 bar 的公共初始序列中的字段 b2，因此 fp->f2 不将 bp->b1 或 bp->b3 作为别名。
- 由于 f3 对应于 foo 和 bar 的公共初始序列中的字段 b3，因此 fp->f3 不将 bp->b1 或 bp->b2 作为别名。

如果使用 -xalias_level=std 选项编译编码示例 6-2，则编译器假定以下别名信息：

fp->f1、 fp->f2、 fp->f3、 bp->b1、 bp->b2 和 bp->b3 并不互为别名。

如果使用 -xalias_level=strong 选项编译编码示例 6-2，则编译器假定以下别名信息：

fp->f1、 fp->f2、 fp->f3、 bp->b1、 bp->b2 和 bp->b3 并不互为别名。

考虑下列源代码示例，它说明某些别名级别无法处理内部指针。有关内部指针的定义，请参见表 A-8。

编码示例 6-3

```
struct foo {
        int f1;
        struct bar *f2;
        struct bar *f3;
        int f4;
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

weak、 layout、 strict 或 std 不支持编码示例 6-3 中的非关联化。在指针赋值 bp=(struct bar*)(&fp->f2) 之后，以下各对内存访问将访问相同的存储单元：

- fp->f2 和 bp->b2 访问相同的存储单元
- fp->f3 和 bp->b3 访问相同的存储单元
- fp->f4 和 bp->b4 访问相同的存储单元

但是，使用选项 weak、layout、strict 和 std 时，编译器假定 fp->f2 和 bp->b2 不设定别名。由于 b2 的偏移为零，与 f2 的偏移（四个字节）不同，并且 foo 和 bar 没有公共初始序列，因此编译器作出该假定。同样，编译器还假定 bp->b3 不将 fp->f3 作为别名， bp->b4 不将 fp->f4 作为别名。

因此，指针赋值 bp=(struct bar*)(&fp->f2) 使编译器关于别名信息的假定不正确。这可能会导致不正确的优化。

请在进行以下示例中显示的修改之后尝试编译。

```
struct foo {
        int f1;
        struct bar fb;    /* 修改的行 */
#define f2 fb.b2          /* 修改的行 */
#define f3 fb.b3          /* 修改的行 */
#define f4 fb.b4          /* 修改的行 */
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

在指针赋值 bp=(struct bar*)(&fp->f2) 之后，以下各对内存访问将访问相同的存储单元：

- fp->f2 和 bp->b2
- fp->f3 和 bp->b3
- fp->f4 和 bp->b4

通过检查前面的代码示例中显示的更改，您可以看到表达式 fp->f2 是表达式 fp->fb.b2 的另一种形式。由于 fp->fb 的类型是 bar，因此 fp->f2 访问 bar 类型的 b2 字段。此外， bp->b2 也访问 bar 类型的 b2 字段。因此，编译器假定 fp->f2 将 bp->b2 作为别名。同样，编译器假定 fp->f3 将 bp->b3 作为别名， fp->f4 将 bp->b4 作为别名。结果，编译器假定的别名与指针赋值产生的实际别名匹配。

考虑以下源代码示例。

编码示例 6-4

```
struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

struct cat {
        int c1;
        struct foo cf;
        int c2;
        int c3;
} *cp;

struct dog {
        int d1;
        int d2;
        struct bar db;
        int d3;
} *dp;
```

如果使用 -xalias_level=weak 选项编译编码示例 6-4，则编译器假定以下别名信息：

- fp->f1 可以将 bp->b1、 cp->c1、 dp->d1、 cp->cf.f1 和 df->db.b1 作为别名。
- fp->f2 可以将 bp->b2、 cp->cf.f1、 dp->d2、 cp->cf.f2、 df->db.b2、 cp->c2 作为别名。
- bp->b1 可以将 fp->f1、 cp->c1、 dp->d1、 cp->cf.f1 和 df->db.b1 作为别名。
- bp->b2 可以将 fp->f2、 cp->cf.f1、 dp->d2、 cp->cf.f1 和 df->db.b2 作为别名。

由于 *dp 可以将 *cp 作为别名，*fp 可以将 dp->db 作为别名，因此 fp->f2 可以将 cp->c2 作为别名。

- cp->c1 可以将 fp->f1、 bp->b1、 dp->d1 和 dp->db.b1 作为别名。
- cp->cf.f1 可以将 fp->f1、 fp->f2、 bp->b1、 bp->b2、 dp->d2 和 dp->d1 作为别名。

cp->cf.f1 不将 dp->db.b1 作为别名。

- cp->cf.f2 可以将 fp->f2、 bp->b2、 dp->db.b1 和 dp->d2 作为别名。
- cp->c2 可以将 dp->db.b2 作为别名。

cp->c2 不将 dp->db.b1 作为别名， cp->c2 不将 dp->d3 作为别名。

如果考虑偏移，只有在 *dp 将 cp->cf 作为别名时， cp->c2 才可以将 db->db.b1 作为别名。但是，如果 *dp 将 cp->cf 作为别名，则 dp->db.b1 必须在 foo cf 末尾之后设定别名，这是对象约束所禁止的。因此，编译器假定 cp->c2 不能将 db->db.b1 作为别名。

cp->c3 可以将 dp->d3 作为别名。

请注意， cp->c3 不将 dp->db.b2 作为别名。由于具有非关联化所涉及的类型的字段的偏移不同并且不重叠，因此这些内存引用不设定别名。基于这种情况，编译器假定它们不能设定别名。

- dp->d1 可以将 fp->f1、 bp->b1 和 cp->c1 作为别名。
- dp->d2 可以将 fp->f2、 bp->b2 和 cp->cf.f1 作为别名。
- dp->db.b1 可以将 fp->f1、 bp->b1 和 cp->c1 作为别名。
- dp->db.b2 可以将 fp->f2、 bp->b2、 cp->c2 和 cp->cf.f1 作为别名。
- dp->d3 可以将 cp->c3 作为别名。

请注意， dp->d3 不将 cp->cf.f2 作为别名。由于具有非关联化所涉及的类型的字段的偏移不同并且不重叠，因此这些内存引用不设定别名。基于这种情况，编译器假定它们不能设定别名。

如果使用 -xalias_level=layout 选项编译编码示例 6-4，则编译器仅假定以下别名信息：

- fp->f1、 bp->b1、 cp->c1 和 dp->d1 均可以互为别名。
- fp->f2、 bp->b2 和 dp->d2 均可以互为别名。
- fp->f1 可以将 cp->cf.f1 和 dp->db.b1 作为别名。
- bp->b1 可以将 cp->cf.f1 和 dp->db.b1 作为别名。
- fp->f2 可以将 cp->cf.f2 和 dp->db.b2 作为别名。
- bp->b2 可以将 cp->cf.f2 和 dp->db.b2 作为别名。

如果使用 -xalias_level=strict 选项编译编码示例 6-4，则编译器仅假定以下别名信息：

- fp->f1 和 bp->b1 可以互为别名。
- fp->f2 和 bp->b2 可以互为别名。
- fp->f1 可以将 cp->cf.f1 和 dp->db.b1 作为别名。
- bp->b1 可以将 cp->cf.f1 和 dp->db.b1 作为别名。
- fp->f2 可以将 cp->cf.f2 和 dp->db.b2 作为别名。
- bp->b2 可以将 cp->cf.f2 和 dp->db.b2 作为别名。

如果使用 -xalias_level=std 选项编译编码示例 6-4，则编译器仅假定以下别名信息：

- fp->f1 可以将 cp->cf.f1 作为别名。
- bp->b1 可以将 dp->db.b1 作为别名。
- fp->f2 可以将 cp->cf.f2 作为别名。
- bp->b2 可以将 dp->db.b2 作为别名。

考虑以下源代码示例。

编码示例 6-5

```
struct foo {
        short f1;
        short f2;
        int   f3;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

union moo {
        struct foo u_f;
        struct bar u_b;
} u;
```

下面是编译器根据以下别名级别作出的假定：

- 如果使用 -xalias_level=weak 选项编译编码示例 6-5，则 fp->f3 和 bp->b2 可以互为别名。
- 如果使用 -xalias_level=layout 选项编译编码示例 6-5，则没有字段可以互为别名。
- 如果使用 -xalias_level=strict 选项编译编码示例 6-5，则 fp->f3 和 bp->b2 可以互为别名。
- 如果使用 -xalias_level=std 选项编译编码示例 6-5，则没有字段可以互为别名。

考虑以下源代码示例。

编码示例 6-6

```
struct bar;

struct foo {
        struct foo *ffp;
        struct bar *fbp;
} *fp;

struct bar {
        struct bar *bbp;
        long        b2;
} *bp;
```

下面是编译器根据以下别名级别作出的假定：

- 如果使用 -xalias_level=weak 选项编译编码示例 6-6，则只有 fp->ffp 和 bp->bbp 可以互为别名。
- 如果使用 -xalias_level=layout 选项编译编码示例 6-6，则只有 fp->ffp 和 bp->bbp 可以互为别名。
- 如果使用 -xalias_level=strict 选项编译编码示例 6-6，则没有字段可以互为别名，原因是即使删除两种结构类型的标记，两种结构类型仍不相同。
- 如果使用 -xalias_level=std 选项编译编码示例 6-6，则没有字段可以互为别名，原因是两种类型和标记均不相同。

考虑以下源代码示例：

```
struct foo;
struct bar;
#pragma alias (struct foo, struct bar)

struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        short b1;
        short b2;
        int   b3;
} *bp;
```

此示例中的 pragma 告诉编译器，允许 `foo` 和 `bar` 互为别名。编译器作出关于别名信息的以下假定：

- `fp->f1` 可以将 `bp->b1`、`bp->b2` 和 `bp->b3` 作为别名
- `fp->f2` 可以将 `bp->b1`、`bp->b2` 和 `bp->b3` 作为别名

第 7 章

转换为 ISO C

本章提供的信息可以帮助您移植 K&R 风格 C 应用程序，以符合 9899:1990 ISO/IEC C 标准。本章提供的信息假定您不想符合更新的 9899:1999 ISO/IEC C 标准，因此使用 `-xc99=%none`。 C 编译器缺省为 `-xc99=%all`，支持 9899:1999 ISO/IEC C 标准。

7.1 基本模式

ISO C 编译器允许使用旧风格和新风格 C 代码。如果您使用下列 `-X`（注意大小写）选项并且 `-xc99=%none`，则编译器提供不同的 ISO C 标准一致性级别。 `-Xa` 为缺省模式。请注意，编译器的缺省模式为 `-xc99=%all`，因此在设置每个 `-X` 选项的情况下编译器的行为取决于 `-xc99` 的设置。

7.1.1 `-Xa`

ISO C 以及 K&R C 兼容性扩展，具有 ISO C 要求的语义更改。如果 K&R C 和 ISO C 为相同构造指定不同语义，则编译器发出关于冲突的警告并使用 ISO C 解释。这是缺省模式。

7.1.2 `-Xc`

（`c` = 一致性）在没有 K&R C 兼容性扩展的情况下，在最大程度上与 ISO C 一致。编译器对使用 ISO C 构造的程序发出错误和警告。

7.1.3 -Xs

(s = K&R C) 编译的语言包括与 ISO K&R C 兼容的所有功能。计算机对在 ISO C 和 K&R C 之间具有不同行为的所有语言构造发出警告。

7.1.4 -Xt

（t = 转换）ISO C 以及 K&R C 兼容性扩展，*没有* ISO C 要求的语义更改。如果 K&R C 和 ISO C 为相同构造指定不同语义，则编译器发出关于冲突的警告并使用 K&R C 解释。

7.2 旧风格和新风格函数的混合

1990 ISO C 标准在语言方面的最大变化是借鉴 C++ 语言的函数原型。通过为每个函数指定其参数的数目和类型，不仅使各常规编译获得对每个函数调用的参数检查 （类似于 `lint` 的参数检查）的益处，而且参数自动转换 （与赋值情形相同）为函数预期的类型。由于现有 C 代码中的许多行应转换为使用原型，因此 1990 ISO C 标准包含控制旧风格和新风格函数声明混合的规则。

7.2.1 编写新代码

当您编写全新的程序时，请在头文件中使用新风格函数声明 （函数原型），在其它 C 源文件中使用新风格函数声明和定义。但是，如果将来可能使用预 ISO C 编译器将代码移植到机器上，我们建议您在头文件和源文件中使用宏 `__STDC__` （仅为 ISO C 编译系统定义）。有关示例，请参见第 7-3 页上的第 7.2.3 节 “混合注意事项”。

如果同一对象或函数的两个不兼容声明在同一作用域内，则符合 ISO C 的编译器必须发出诊断消息。如果使用原型来声明和定义所有函数，并且相应的头文件包含在正确的源文件中，则所有调用应与函数的定义一致。此协议消除一种最常见的 C 编程错误。

7.2.2 更新现有代码

如果您有现有的应用程序并且要获取函数原型的益处，则可以进行很多更新，取决于您要更改代码的程度：

1. 重新编译而不进行任何更改。

   如果使用 `-v` 选项进行调用，即使不更改代码，编译器也会对参数类型和数目的不匹配发出警告。

2. 仅在头文件中增加函数原型。

   包括所有全局函数调用。

3. 在头文件中增加函数原型，并使每个源文件以其局部 （静态）函数的函数原型开头。

   包括所有函数调用，但是这样做需要在源文件中为每个局部函数键入两次接口。

4. 更改所有函数声明和定义以使用函数原型。

对于大多数程序员，第 2 种选择和第 3 种选择可能最具成本效益。遗憾的是，这些选项要求程序员详细了解混合新旧风格的规则。

7.2.3 混合注意事项

为使函数原型声明与旧风格函数定义配合工作，二者必须指定功能相同 （或者套用 ISO C 术语，具有*兼容类型*）的接口。

对于具有可变参数的函数，不存在 ISO C 的省略号表示法和旧风格 `varargs()` 函数定义的混合。对于具有固定数目参数的函数，情况相当简单：只需指定在先前实现中传递的参数的类型。

在 K&R C 中，每个参数在传递到被调用函数之前根据缺省参数提升进行转换。这些提升指定，所有比 `int` 窄的整数类型均已提升为 `int` 长度，并且任何 `float` 参数均已提升为 `double`，从而简化编译器和库。函数原型更具有表示性 — 指定的参数类型即为传递给函数的类型。

因此，如果函数原型是针对现有 （旧风格）函数定义编写的，则具有以下任何类型的函数原型中不应存在参数：

`char`	`signed char`	`unsigned char`	`float`
`short`	`signed short`	`unsigned short`	

在编写原型方面仍存在两个复杂因素：`typedef` 名称以及窄无符号类型的提升规则。

如果旧风格函数中的参数是使用 typedef 名称 （如 off_t 和 ino_t）声明的，则要知道 typedef 名称是否指定受缺省参数提升影响的类型，这一点很重要。对于这两个名称， off_t 为 long 类型，因此适合在函数原型中使用； ino_t 过去通常为 unsigned short 类型，因此，如果用于原型，则编译器会发出诊断消息，原因是旧风格定义和原型指定不同的不兼容接口。

考虑究竟应该使用什么来代替 unsigned short 致使问题最终复杂化。K&R C 和 1990 ISO C 编译器之间一个最大的不兼容性是用于将 unsigned char 和 unsigned short 加宽为 int 值的提升规则。（参见第 7-8 页上的第 7.4 节 “提升：无符号保留与值保留”。）与此类旧风格参数匹配的参数类型取决于编译时使用的编译模式：

- -Xs 和 -Xt 应使用 unsigned int
- -Xa 和 -Xc 应使用 int

最佳方法是更改旧风格定义，以指定 int 或 unsigned int 并在函数原型中使用匹配类型。如有必要，在输入函数后，您可以始终将其值赋给具有更窄类型的局部变量。

请注意原型中 id 的使用，它可能受预处理的影响。考虑以下示例：

```
#define status 23
void my_exit(int status);    /* 通常，作用域以原型开始 */
                             /* 并以原型结束 */
```

不要将函数原型与包含窄类型的旧风格函数声明混合。

```
void foo(unsigned char, unsigned short);
void foo(i, j) unsigned char i; unsigned short j; {...}
```

正确使用 __STDC__ 可生成一个可用于新旧编译器的头文件：

```
header.h:
    struct s { /* .  .  .  */ };
    #ifdef __STDC__
       void errmsg(int, ...);
       struct s *f(const char *);
       int g(void);
    #else
      void errmsg();
      struct s *f();
      int g();
    #endif
```

以下函数使用原型，但仍可在较旧的系统中编译：

```
struct s *
#ifdef __STDC__
    f(const char *p)
#else
    f(p) char *p;
#endif
{
    /* .  .  .  */
}
```

下面是更新的源文件 （与上述选项 3 相同）。局部函数仍使用旧风格定义，但包含一个原型以用于较新的编译器：

```
source.c:
   #include "header.h"
      typedef /* .  .  .  */ MyType;
   #ifdef __STDC__
      static void del(MyType *);
      /* .  .  .  */
      static void
      del(p)
      MyType *p;
      {
      /* .  .  .  */
      }
      /* .  .  .  */
```

7.3 带有可变参数的函数

在前面的实现中，您不能指定函数预期的参数类型，但 ISO C 鼓励您使用原型执行该操作。为支持 `printf()` 等函数，原型的语法包含一个特殊的省略号 (...) 终结符。由于一个实现可能需要执行一些特殊操作来处理可变数目的参数，因此 ISO C 要求此类函数的所有声明和定义均包含省略号终结符。

由于参数的 "..." 部分没有名称，因此 `stdarg.h` 中包含的一组特殊宏为函数提供对这些参数的访问权。此类函数的早期版本必须使用 `varargs.h` 中包含的类似宏。

我们假定要编写的函数是一个称为 errmsg() 的错误处理程序，它返回 void，并且函数的唯一固定参数是指定关于错误消息的详细信息的 int。此参数后面可以跟一个文件名、一个行号或二者，在它们之后是格式和参数，与指定错误消息文本的 printf() 类似。

为使示例可以使用较早的编译器进行编译，我们广泛使用了仅针对 ISO C 编译系统定义的宏 __STDC__。因此，该函数在相应头文件中的声明为：

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

在包含 errmsg() 的定义的文件中，新旧风格变得复杂。首先，要包含的头文件取决于编译系统：

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif
#include <stdio.h>
```

包含 stdio.h 是因为我们稍后调用 fprintf() 和 vfprintf()。

其次是函数的定义。标识符 va_alist 和 va_dcl 是旧风格 varargs.h 接口的一部分。

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl /* 注：无分号！ */
#endif
{
   /* 下面是更多详细信息 */
}
```

由于旧风格变量参数机制不允许指定任何固定参数，因此必须安排在可变部分之前访问它们。此外，由于参数的 “…” 部分缺少名称，新的 va_start() 宏具有第二个参数 — “…” 终结符前面的参数的名称。

作为一种扩展， Sun ISO C 允许在没有固定参数的情况下声明和定义函数，如下所示：

```
int f(...);
```

对于此类函数，应在第二个参数为空的情况下调用 va_start()，如下所示：

```
va_start(ap,)
```

以下是函数的主体：

```
{
    va_list ap;
    char *fmt;
#ifdef __STDC__
    va_start(ap, code);
#else
    int code;
    va_start(ap);
    /* 提取固定参数 */
    code = va_arg(ap, int);
#endif
    if (code & FILENAME)
        (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
    if (code & LINENUMBER)
        (void)fprintf(stderr, "%d: ", va_arg(ap, int));
    if (code & WARNING)
        (void)fputs("warning: ", stderr);
    fmt = va_arg(ap, char *);
    (void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

对于旧风格和 ISO C 版本， va_arg() 和 va_end() 宏的执行情况均相同。由于 va_arg() 更改 ap 的值，因此对 vfprintf() 的调用不能为：

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

FILENAME、 LINENUMBER 和 WARNING 宏的定义可能包含在与 errmsg() 的声明相同的头文件中。

对 errmsg() 的调用的样例为：

```
errmsg(FILENAME, "<command line>", "cannot open: %s\n",
argv[optind]);
```

7.4 提升：无符号保留与值保留

1990 ISO C 标准的补充材料 “Rationale”部分出现以下信息： “QUIET CHANGE”。依赖于无符号保留算术转换的程序表现各异，可能没有错误消息。这被认为是委员会对普遍的当前实践的最重大的更改。

本节研究此更改如何影响代码。

7.4.1 背景

根据 K&R 的 《*The C Programming Language*》（第一版）， `unsigned` 准确地指定一种类型；不存在 `unsigned char`、 `unsigned short` 或 `unsigned long`，但是在此之后不久，大多数 C 编译器增加了这些类型。有些编译器未实现 `unsigned long`，但是包含了其它两种类型。自然地，当这些新类型与在表达式中其它类型混合时，实现为类型提升选择不同的规则。

在大多数 C 编译器中，使用比较简单的规则 “无符号保留”：当无符号类型需要加宽时，将被加宽为无符号类型；当无符号类型与带符号类型混合时，结果为无符号类型。

ISO C 指定的另一个规则称为 “值保留”，其中结果类型取决于操作数类型的相对长度。当加宽 `unsigned char` 或 `unsigned short` 类型时，如果 `int` 的长度足以表示较短类型的所有值，则结果类型为 `int`。否则，结果类型为 `unsigned int`。对于大多数表达式，值保留规则产生最常见类型的算术结果。

7.4.2 编译行为

只有在转换模式或 ISO 模式（`-Xt` 或 `-Xs`）下， ISO C 编译器才使用无符号保留提升；在其它两种模式下，即符合标准模式 (`-Xc`) 和 ISO 模式 (`-Xa`)，使用值保留提升规则。

7.4.3 第一个示例：强制类型转换的使用

在以下代码中，假定 unsigned char 比 int 短。

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

   return (i + uc) < 17;
}
```

以上代码导致编译器在您使用 -xtransition 选项时发出以下警告：

第 6 行： 警告： "<" 的语义在 ISO C 中发生变化；应使用显式强制类型转换

加法运算结果的类型为 int （值保留）或 unsigned int （无符号保留），但二者之间的位模式不会更改。在 2 的补码机器上，我们获得：

```
    i:    111...110 (-2)
+   uc:  000...001 ( 1)
====================
        111...111 (-1 或 UINT_MAX)
```

这种位表示对应于 -1 （对于 int）或 UINT_MAX （对于 unsigned int）。因此，如果结果的类型为 int，则使用带符号比较并且小于测试为真；如果结果类型为 unsigned int，则使用无符号比较并且小于测试为假。

强制类型转换的加法用来指定这两种行为之中所期望的行为：

```
value preserving:
    (i + (int)uc) < 17
unsigned preserving:
    (i + (unsigned int)uc) < 17
```

由于不同的编译器对相同的代码选择的不同的含义，因此该表达式存在歧义。强制类型转换的加法帮助阅读器并消除警告消息。

7.4.4 位字段

位字段值的提升存在同样的情况。在 ISO C 中，如果 `int` 或 `unsigned int` 位字段中的位数少于 `int` 的位数，则提升后的类型为 `int`；否则，提升后的类型为 `unsigned int`。在大多数较早的 C 编译器中，对于显式无符号位字段，提升后的类型为 `unsigned int`，否则为 `int`。

强制类型转换的类似使用可以消除存在歧义的情况。

7.4.5 第二个示例：相同结果

在以下代码中，假定 `unsigned short` 和 `unsigned char` 均比 `int` 窄。

```
int f(void)
{
    unsigned short us;
    unsigned char uc;
    return uc < us;
}
```

在此示例中，自动变量被提升为 `int` 或 `unsigned int`，因此比较有时无符号，有时带符号。然而，由于两种选择的结果相同，因此 C 编译器并不向您发出警告。

7.4.6 整型常量

与表达式一样，某些整型常量的类型的规则已更改。在 K&R C 中，只有在无后缀十进制常量的值用 `int` 足以表示时，其类型才为 `int`；只有在无后缀八进制或十六进制常量的值用 `unsigned int` 足以表示时，其类型才为 `int`。否则，整型常量的类型为 `long`。有时，值用结果类型不足以表示。在 1990 ISO/IEC C 标准中，常量类型是以下列表中与值对应的第一个类型：

- 无后缀十进制：`int`, `long`, `unsigned long`
- 无后缀八进制或十六进制：`int`, `unsigned int`, `long`, `unsigned long`
- `U` 后缀：`unsigned int`, `unsigned long`
- `L` 后缀：`long`, `unsigned long`
- `UL` 后缀：`unsigned long`

当您使用 `-xtransition` 选项时，对于其行为可能会根据所涉及常量的类型处理规则而更改的任何表达式， ISO C 编译器向您发出警告。旧整型常量类型处理规则仅在转换模式下使用； ISO 模式和符合标准模式使用新规则。

注 – 无后缀十进制常量的类型处理规则已按照 1999 ISO C 标准更改。参见第 2-1 页上的第 2.1.1 节 “整型常量”。

7.4.7 第三个示例：整型常量

在以下代码中，假定 `int` 为 16 位。

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

由于十六进制常量的类型为 `int`（在 2 的补码机器上值为 `-1`）或 `unsigned int`（值为 65535），因此比较在 `-Xs` 和 `-Xt` 模式下为真，而在 `-Xa` 和 `-Xc` 模式下为假。

同样，相应的强制类型转换澄清代码并禁止警告：

```
-Xt, -Xs modes:
    i > (int)0xffff

-Xa, -Xc modes:
    i > (unsigned int)0xffff
       or
    i > 0xffffU
```

`U` 后缀字符是 ISO C 的新特征，在使用较早的编译器时可能会产生错误消息。

7.5 标记化和预处理

这可能是以前的 C 版本中最少涉及的部分，涉及将每个源文件从一串字符转换为标记序列 （即可进行语法分析）的操作。这些操作包括空白 （包括注释）的识别、将连续指令捆绑为标记、处理预处理指令行以及宏替换。然而，从未对其各自顺序提供保证。

7.5.1 ISO C 转换阶段

这些转换阶段的顺序由 ISO C 指定。

源文件中的各个三字母序列被替换。 ISO C 具有九个三字母序列，创建这些序列只是为了弥补字符集的不足，它们是三字符序列，用来命名 ISO 646-1983 字符集中没有的字符：

表 7-1 三字母序列

三字母序列	转换为
??=	#
??-	~
??(	[
??)	]
??!	|
??<	{
??>	}
??/	\
??'	^

ISO C 编译器必定理解这些序列，但我们建议不要使用它们。当您使用 `-xtransition` 选项时，如果它在转换 (`-Xt`) 模式下甚至在注释中替换一个三字母， ISO C 编译器会向您发出警告。例如，考虑以下情形：

```
/* comment *??/
/* still comment? */
```

??/ 变为反斜杠。该字符和后面的换行符被删除。结果字符为：

```
/* comment */* still comment? */
```

第二行的第一个 / 是注释的结尾。下一个标记是 *。

1. 删除各对反斜杠/换行符。
2. 源文件转换为预处理标记和空白序列。每个注释有效地替换为一个空格字符。
3. 处理各个预处理指令并替换所有宏调用。每个 #include 源文件在其内容替换指令行之前运行较早的阶段。
4. 解释各个换码序列 （形式为字符常量和字符串文字）。
5. 并置相邻字符串文字。
6. 各个预处理标记转换为常规标记，编译器正确分析这些标记并生成代码。
7. 解析所有外部对象和函数引用，形成最终程序。

7.5.2 旧 C 转换阶段

以前的 C 编译器不执行如此简单的阶段序列，也不保证何时应用这些步骤。独立预处理程序识别标记和空白的时间基本上与它替换宏和处理指令行的时间相同。然后输出由适当的编译器完全重新标记化，接着编辑器分析语言并生成代码。

由于预处理程序内的标记化进程是一个时刻执行的操作，并且宏替换作为基于字符而不是基于标记的操作来执行，因此标记和空白在预处理期间有大量变化形式。

这两种方法存在很多差异。本节其余部分讨论代码行为如何因宏替换过程中发生的行拼接、宏替换、字符串化以及标记粘贴而更改。

7.5.3 逻辑源代码行

在 K&R C 中，反斜杠/换行符对仅允许作为将指令、字符串文字或字符常量续到下一行的一种方法。 ISO C 扩展了这个概念，从而使反斜杠/换行符对可以将任何内容续到下一行。结果为逻辑源代码行。因此，依赖于在反斜杠/换行符对的一侧独立识别标记的任何代码均不执行预期的行为。

7.5.4 宏替换

在 ISO C 之前，从未详细描述宏替换进程。这种不明确性产生很多有分歧的实现。依赖于比明显常量替换和简单类函数宏更奇特的事情的任何代码可能并不真正可移植。本手册无法指出旧 C 宏替换实现与 ISO C 版本之间的所有差异。除标记粘贴和字符串化之外的几乎所有宏替换的使用产生的标记系列均与以前完全相同。此外， ISO C 宏替换算法可以完成在旧 C 版本中无法完成的工作。例如：

```
#define name (*name)
```

使 name 的任何使用均替换为通过 name 进行的间接引用。旧 C 预处理程序会产生大量圆括号和星号，并最终产生关于宏递归的错误。

ISO C 对宏替换方法的主要更改是要求宏参数，而不是要求那些本身是宏替换操作符 # 和 ## 的操作数并且在替换标记列表中替换之前要递归扩展的参数。然而，这种更改很少在结果标记中产生实际差异。

7.5.5 使用字符串

注 – 在 ISO C 中，如果您使用 -xtransition 选项，则以下带有 ‡ 标记的示例将生成警告。只有在转换模式 （-Xt 和 -Xs）下，结果才与以前 C 版本中的结果相同。

在 K&R C 中，以下代码生成字符串文字 "x y!"：

```
#define str(a) "a!"    ‡
str(x y)
```

因此，预处理程序在字符串文字和字符常量中查找看起来类似宏参数的字符。 ISO C 认识到此功能的重要性，但不允许对部分标记的操作。在 ISO C 中，以上宏的所有调用均生成字符串文字 "a!"。为在 ISO C 中实现旧效果，我们使用 # 宏替换操作符和字符串文字并置。

```
#define str(a) #a "!"
str(x y)
```

以上代码生成两个字符串文字 "x y" 和 "!" 它们在并置后生成相同的 "x y!"。

不直接替换字符常量的类似操作。此功能的主要用法与以下类似：

```
#define CNTL(ch) (037 & ‘ch’)     ‡
CNTL(L)
```

它生成

```
(037 & 'L')
```

求值为 ASCII control-L 字符。我们知道的最佳解决办法是将此宏的用法更改为：

```
#define CNTL(ch) (037 & (ch))
CNTL(‘L’)
```

此代码的可读性和实用性更强，因为它还可以应用于表达式。

7.5.6 标记粘贴

在 K&R C 中，合并两个标记的方法至少有两种。以下代码中的两个调用均使用 `x` 和 `1` 两个标记生成单个标识符 `x1`。

```
#define self(a) a
#define glue(a,b) a/**/b ‡
self(x)1
glue(x,1)
```

同样，ISO C 不认可这两种方法。在 ISO C 中，以上两个调用均生成两个独立标记 `x` 和 `1`。可以通过使用 `##` 宏替换操作符针对 ISO C 重新编写以上第二种方法：

```
#define glue(a,b) a ## b
glue(x, 1)
```

只有在定义了 `__STDC__` 时，才应将 # 和 ## 用作宏替换操作符。由于 `##` 是实际操作符，因此对于定义和调用中的空白，调用更加自由。

实现第一种旧风格粘贴方案没有直接方法，但是由于该方案在调用中增加了粘贴负担，因此其使用频率比另一种形式低。

7.6 const 和 volatile

关键字 const 是可转换到 ISO C 的 C++ 功能之一。当 ISO C 委员会创建类似关键字 volatile 时，同时创建 “类型限定符” 种类。

7.6.1 类型，仅适用于 lvalue

const 和 volatile 属于标识符的类型，而不属于标识符的存储类。然而，当从表达式求值中获取对象的值时，确切地说是当 lvalue 变为 rvalue 时，经常会将这些类型从类型的最顶端删除。这些术语起源于原型赋值 "L=R";，其中左侧必须仍直接引用对象 （一个 lvalue），右侧只需为一个值 （一个 rvalue）。因此，只有本身是 lvalue 的表达式才可以由 const 或 volatile （或二者）限定。

7.6.2 派生类型中的类型限定符

类型限定符可修改类型名称和派生类型。派生类型是 C 声明的那些可反复应用而生成越来越复杂的类型的部分：指针、数组、函数、结构和联合。除函数之外，可使用一个或两个类型限定符更改派生类型的行为。

例如，

```
const int five = 5;
```

声明并初始化类型为 const int 并且其值未被相应的程序更改的对象。关键字的顺序对于 C 并不重要。例如，声明：

```
int const five = 5;
```

和

```
const five = 5;
```

与以上声明在效果上相同。

声明

```
const int *pci = &five;
```

声明一个类型为指向 const int 的指针的对象，该对象最初指向以前声明的对象。指针本身并不具有限定类型 — 它指向限定类型，并且实际上在程序执行期间可更改为任何 int。除非使用强制类型转换，否则 pci 不能用来修改它指向的对象，如下所示：

```
*(int *)pci = 17;
```

如果 pci 实际上指向 const 对象，则此代码的行为未定义。

声明

```
extern int *const cpi;
```

表明程序中某个位置存在类型为指向 int 的 const 指针的全局对象的定义。在此情况下， cpi 的值将不会被相应的程序更改，但是可用来修改它指向的对象。请注意，在以上声明中， const 位于 * 之后。以下一对声明产生的效果相同：

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

这些声明可以合并为以下声明，其中对象的类型声明为指向 const int 的 const 指针：

```
const int *const cpci;
```

7.6.3 const 意味着 readonly

根据经验，对于关键字， readonly 优于 const。如果以此方式读取 const，则如下声明：

```
char *strcpy(char *, const char *);
```

很容易理解，即第二个参数仅用于读取字符值，而第一个参数覆写它指向的字符。此外，尽管在以上示例中， cpi 的类型是指向 const int 的指针，您仍可以通过某些其它方法更改它指向的对象的值，除非它确实指向被声明为 const int 类型的对象。

7.6.4 const 用法示例

const 的两种主要用法是将大的编译时初始化表声明为不更改，以及指定指针参数不修改它们指向的对象。

第一种用法潜在允许某个程序的部分数据被同一程序的其它并行调用共享。其结果是，如果试图修改此不变数据，将会通过某种内存保护故障立即检测到，因为数据驻留在内存的只读部分。

第二种用法有助于在演示期间生成内存故障之前查找潜在错误。例如，如果将指针传递给无法进行如此修改的字符串，则临时将空字符置入字符串中间的函数在编译时被检测到。

7.6.5 volatile 意味着精确语义

到目前为止，示例全都使用 const，因为其概念比较简单。但是， volatile 到底意味着什么？ 对于编译器编写者，它只有一种含义：访问此类对象时不采用代码生成快捷方式。在 ISO C 中，声明具有相应属性及 volatile 限定类型的各个对象是程序员的责任。

7.6.6 volatile 用法示例

volatile 对象的四个常见示例为：

- 本身是内存映射 I/O 端口的对象
- 多个并行进程之间共享的对象
- 异步信号处理程序修改的对象
- 调用 setjmp 的函数中声明的自动存储持续时间对象，其值在 setjmp 调用和相应的 longjmp 调用之间会更改

前三个示例是具有特殊行为的对象的所有实例：在程序执行期间的任何点均可修改其值。因此，表面上的死循环：

```
flag = 1;
while (flag);
```

实际上有效，只要 flag 具有 volatile 限定类型。某些异步事件将来可能将 flag 设置为零。否则，由于 flag 的值在循环主体中保持不变，编译系统会将以上循环更改为完全忽略 flag 值的真正死循环。

第四个示例涉及调用 setjmp 的函数的局部变量，因此进一步加以讨论。关于 setjmp 和 longjmp 行为的细小字体注释表明对于符合第四种情形的对象的值没有任何保证。对于大多数所期望的行为，有必要让 longjmp 检查调用 setjmp 的函数与调用 longjmp 的函数之间的各个栈帧是否有保存的寄存器值。由于存在异步创建栈帧的可能性，使该作业更加困难。

在将自动对象声明为 volatile 限定类型时，编译系统知道生成的代码必须与程序员编写的代码完全匹配。因此，此类自动对象的最新值始终存储在内存中，而不是仅仅存储在寄存器中，并且保证在调用 longjmp 时是最新的。

7.7 多字节字符和宽字符

最初， ISO C 的国际化只影响库函数。但是，国际化的最终阶段 （多字节字符和宽字符）还影响语言。

7.7.1 亚洲语言需要多字节字符

在亚洲语言计算机环境中，基本困难是 I/O 需要大量表意文字。要在通常的计算机体系结构的约束下工作，需要将这些表意文字编码为字节序列。相关的操作系统、应用程序和终端将这些字节序列理解为单个表意字符。此外，所有这些编码允许将常规单字节字符与表意字符字节序列混合。识别不同表意字符的难度取决于使用的编码方案。

无论使用什么编码方案， ISO C 均定义术语 “多字节字符”来表示为表意字符编码的字节序列。所有多字节字符均为 “扩展字符集”的成员。常规单字节字符只是多字节字符的一个特例。对编码的唯一要求是多字节字符不能将空字符用作它的编码的一部分。

ISO C 指定程序注释、字符串文字、字符常量和头文件名均为多字节字符序列。

7.7.2 编码变种

编码方案分为两种。第一种方案是，每个多字节字符均自我标识，也就是说，可以将任何多字节字符简单地插入任何一对多字节字符之间。

第二种方案是，特殊的移位字节的存在会更改后续字节的解释。例如，某些字符终端使用该方法进入和退出绘线模式。对于使用多字节字符和移位状态相关编码编写的程序， ISO C 要求每个注释、字符串文字、字符常量和头文件名必须在未移位状态下开始和结束。

7.7.3 宽字符

如果所有字符的字节数或位数都相同，则消除了处理多字节字符的一些不便之处。由于此类字符集中可包含成千上万的表意字符，因此应该使用 16 位 或 32 位整数值来表示所有成员。（整个中文字母表包含的表意字符超过 65,000 个！）ISO C 包含 `typedef` 名称 `wchar_t`，作为足以表示扩展字符集的所有成员的实现定义整数类型。

对于每个宽字符，存在一个相应的多字节字符，反之亦然；对应于常规单字节字符的宽字符需要具有与其单字节值相同的值，包括空字符。但是，并不保证宏 `EOF` 的值可以存储在 `wchar_t` 中，因为 `EOF` 可能不能表示为 `char`。

7.7.4 转换函数

1990 ISO/IEC C 标准提供了五个管理多字节字符和宽字符的库函数， 1999 ISO/IEC C 标准提供了更多此类函数。

7.7.5 C 语言特征

为了给亚洲语言环境中的程序员提供更大的灵活性， ISO C 提供宽字符常量和宽字符串文字。其形式与其非宽版本相同，只是前面加上前缀字母 `L`：

- `'x'` 常规字符常量
- `'¥'` 常规字符常量
- `L'x'` 宽字符常量
- `L'¥'` 宽字符常量
- `"abc¥xyz"` 常规字符串文字
- `L"abcxyz"` 宽字符串文字

在常规版本和宽版本中，多字节字符均有效。生成表意字符 `¥` 所必需的字节序列与编码有关，但是如果它由多个字节组成，则字符常量 `'¥'` 的值是实现定义的，正如 `'ab'` 的值是实现定义的一样。除了换码序列之外，常规字符串文字包含引号之间指定的字节，包括每个指定的多字节字符的字节。

当编译系统遇到宽字符常量或宽字符串文字时，每个多字节字符转换为宽字符，如同调用了 `mbtowc()` 函数一样。因此， `L'¥'` 的类型为 `wchar_t`； `abc¥xyz` 的类型为八位数组 `wchar_t`。正如常规字符串文字一样，每个宽字符串文字具有附加的零值元素，但是在这些情况下，它是具有零值的 `wchar_t`。

正如常规字符串文字可用作字符数组初始化的速记方法，宽字符串文字可用于初始化 wchar_t 数组：

```
wchar_t *wp = L"a¥z";
wchar_t x[] = L"a¥z";
wchar_t y[] = {L'a', L'¥', L'z', 0};
wchar_t z[] = {'a', L'¥', 'z', '\0'};
```

在以上示例中，x、y 和 z 这三个数组以及 wp 指向的数组具有相同长度。所有数组均使用相同的值进行初始化。

最后，正如常规字符串文字一样，并置相邻宽字符串文字。但是，对于 1990 ISO/IEC C 标准，相邻常规字符串文字和宽字符串文字会产生未定义的行为。另外， 1990 ISO/IEC C 标准指定，如果编译器不接受此类并置，则无需生成错误消息。

7.8 标准头文件和保留名称

在标准化过程早期， ISO 标准委员会选择包含库函数、宏和头文件作为 ISO C 的一部分。

本节介绍各种保留名称及有关其保留的某些基本原理。最后是一系列规则，遵循这些规则可使程序扫清任何保留名称。

7.8.1 标准头文件

标准头文件为：

表 7-2 标准头文件

assert.h	locale.h	stddef.h
ctype.h	math.h	stdio.h
errno.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h

大多数实现提供更多头文件，但是严格符合 1990 ISO/IEC 标准的 C 程序只能使用这些头文件。

关于其中某些头文件的内容，其它标准稍有不同。例如， POSIX (IEEE 1003.1) 指定 fdopen 在 stdio.h 中声明。为了允许这两种标准共存， POSIX 要求在包含任何头文件之前对宏 _POSIX_SOURCE 进行 #defined，以保证这些附加名称存在。在其《*Portability Guide*》中， X/Open 对其扩展也使用这种宏方案。 X/Open 的宏是 _XOPEN_SOURCE。

ISO C 要求标准头文件自给自足且幂等。标准头文件之前或之后不需要任何其它头文件进行 #included，并且每个标准头文件可多次进行 #included 而不会导致问题。该标准还要求它的头文件只能在安全上下文中进行 #included，以便保证头文件中使用的名称保持不变。

7.8.2 保留供实现使用的名称

该标准对有关其库的实现施加更多约束。以前，大多数程序员不知道在 UNIX 系统上对他们自己的函数使用 read 和 write 等名称。ISO C 要求实现中的引用仅引入该标准保留的名称。

因此，该标准保留所有可能名称的子集供实现使用。此类名称由标识符组成，标识符以下划线开头，后面是其它下划线或大写字母。此类名称包含与以下常规表达式匹配的所有名称：

```
_[_A-Z][0-9_a-zA-Z]*
```

严格地说，如果程序使用此标识符，其行为未定义。因此，使用 _POSIX_SOURCE（或 _XOPEN_SOURCE）的程序具有未定义的行为。

但是，未定义的行为具有不同的程度。如果您在符合 POSIX 的实现中使用 _POSIX_SOURCE，则您知道程序的未定义行为包括某些头文件中的附加名称，并且该程序仍符合公认的标准。 ISO C 标准中的预留漏洞允许实现符合表面上不兼容的规范。另一方面，当遇到 _POSIX_SOURCE 等名称时，不符合 POSIX 标准的实现按任意方式执行。

该标准还保留以下划线开头的所有其它名称，以用于作为常规文件作用域标识符以及作为结构和联合的标记的头文件，但不用于局部作用域。允许以下公共实践：让命名为 _filbuf 和 _doprnt 的函数实现库的隐藏部分。

7.8.3 保留供扩展使用的名称

除了显式保留的所有名称之外， 1990 ISO/IEC C 标准还保留 （供实现和将来标准使用）与某些模式匹配的名称：

表 7-3 保留供扩展使用的名称

文件	保留名称模式
errno.h	E[0-9A-Z].*
ctype.h	(to|is)[a-z].*
locale.h	LC_[A-Z].*
math.h	*当前函数名称* [fl]
signal.h	(SIG|SIG_)[A-Z].*
stdlib.h	str[a-z].*
string.h	(str|mem|wcs)[a-z].*

在以上列表中，只有在包含相关头文件时，以大写字母开头的名称才是宏并被保留。其余名称可指定函数，不能用于为任何全局对象或函数命名。

7.8.4 可安全使用的名称

您可以遵循以下四个简单规则以避免与任何 ISO C 保留名称冲突：

- #include 源文件顶部的所有系统头文件 （除非可能在 _POSIX_SOURCE 和/或 _XOPEN_SOURCE 的 #define 之后）。
- 不要定义或声明以下划线开头的任何名称。
- 在所有文件作用域标记和常规名称的前几个字符内的某位置使用下划线或大写字母。请注意 stdarg.h 或 varargs.h 中的前缀 va_。
- 在所有宏名称的前几个字符内的某位置使用数字或非大写字母。如果 errno.h 被 #included，则保留几乎所有以 E 开头的名称。

这些规则仅仅是要遵循的一般准则，缺省情况下大多数实现将继续向标准头文件增加名称。

7.9 国际化

第 7-19 页上的第 7.7 节 “多字节字符和宽字符”介绍了标准库的国际化。本节讨论受影响的库函数，并提供一些关于应如何编写程序以便利用这些功能的提示。本节只讨论关于 1990 ISO/IEC C 标准的国际化。1999 ISO/IEC C 标准并未进行重大扩展以支持高于此处讨论的国际化。

7.9.1 语言环境

任何时候，C 程序都有当前语言环境 — 描述适合某些国家、文化和语言的惯例的信息集合。语言环境具有字符串名称。唯一的两个标准化语言环境名称为 "C" 和 ""。每个程序在 "C" 语言环境中启动，这导致所有库函数像过去一样工作。 "" 语言环境是实现在选择适合程序的调用的正确惯例集时的首选。 "C" 和 "" 可导致相同的行为。实现可能提供其它语言环境。

为了实用和方便，语言环境被划分为一系列种类。程序可更改整个语言环境，或者只更改一个或多个种类。通常，每个种类影响与受其它种类影响的函数分开的函数集，因此可以临时更改一个种类。

7.9.2 setlocale() 函数

setlocale() 函数是程序的语言环境的接口。通常，使用调用国家的惯例的任何程序在程序执行路径的早期应发出一个调用，如：

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

该调用导致程序的当前语言环境更改为相应的本地版本，因为 LC_ALL 是指定整个语言环境而不是某个种类的宏。以下是标准种类：

LC_COLLATE	排序信息
LC_CTYPE	字符分类信息
LC_MONETARY	货币打印信息
LC_NUMERIC	数值打印信息
LC_TIME	日期和时间打印信息

这些宏中的任何宏均可作为 setlocale() 的第一个参数传递以指定该种类。

setlocale() 函数返回给定种类的当前语言环境的名称 （或 LC_ALL），并且在其第二个参数为空指针时具有仅查询功能。因此，如下代码可用于在有限持续时间内更改语言环境或其中一部分：

```
#include <locale.h>
/*...*/
char *oloc;
/*...*/
oloc = setlocale(LC_category, NULL);
if (setlocale(LC_category, "new") != 0)
{
        /* 使用临时更改的语言环境 */
    (void)setlocale(LC_category, oloc);
}
```

大多数程序不需要此功能。

7.9.3 更改的函数

只要可能并且适当，现有的库函数被扩展为包含依赖语言环境的行为。这些函数分为两组：

- ctype.h 头文件声明的函数 （字符分类和转换），以及
- 转换为和转换自数值的可打印形式和内部形式的函数，如 printf() 和 strtod()。

对于附加字符，如果当前语言环境的 LC_CTYPE 种类不是 "C"，则除 isdigit() 和 isxdigit() 之外的所有 ctype.h 判定函数可返回非零 （真）值。在西班牙语言环境中，isalpha('ñ') 应为真。同样，字符转换函数 tolower() 和 toupper() 应相应地处理 isalpha() 函数标识的任何额外字母字符。ctype.h 函数通常是使用由字符参数索引的查表而实现的宏。通过将表重新设置为新语言环境的值可更改这些函数的行为，因此没有性能影响。

如果当前语言环境的 LC_NUMERIC 种类不是 "C"，则那些写入或解释可打印的浮点值的函数可以更改为使用小数点字符而不是句点 (.)。关于将任何数值转换为具有千分隔符型字符的可打印形式，没有任何规定。从可打印形式转换为内部形式时，允许实现接受此类附加形式，同样是不在 "C" 语言环境中。使用小数点字符的函数是 printf() 和 scanf() 系列、atof() 以及 strtod()。允许实现定义的扩展的函数是 atof()、atoi()、 atol()、 strtod()、 strtol()、 strtoul() 以及 scanf() 系列。

7.9.4 新函数

某些依赖语言环境的功能已作为新的标准函数增加。除了 setlocale()（它允许控制语言环境本身）之外，该标准还包括以下新函数：

localeconv()	数值/货币转换
strcoll()	两个字符串的整理顺序
strxfrm()	转换字符串以便整理
strxfrm()	转换字符串以便整理

此外，还有多字节函数 mblen()、 mbtowc()、 mbstowcs()、 wctomb() 和 wcstombs()。

localeconv() 函数返回一个指针，该指针指向包含对格式化数值有用的信息以及适合当前语言环境的 LC_NUMERIC 和 LC_MONETARY 种类的货币信息的结构。这是唯一的一个其行为依赖于多个种类的函数。对于数值，该结构描述小数点字符、千分隔符以及分隔符的位置。有十五个描述如何格式化货币值的其它结构成员。

strcoll() 函数类似于 strcmp() 函数，只是它根据当前语言环境的 LC_COLLATE 种类比较两个字符串。 strxfrm() 函数还可以用于将一个字符串转换为另一个字符串，以便任何两个此类转换后字符串均可以传递到 strcmp()，并且获得与 strcoll() 在传递两个预转换字符串时返回的排序类似的排序。

strftime() 函数提供与 sprintf() 对 struct tm 中的值使用的格式化类似的格式化，并提供依赖当前语言环境的 LC_TIME 种类的某些日期和时间表示。此函数基于作为 UNIX System V 发行版本 3.2 的一部分发行的 ascftime() 函数。

7.10 表达式中的分组和求值

Dennis Ritchie 在设计 C 时所作的选择之一是为编译器提供一个许可证，以便重新整理包含算术上可交换并且关联的相邻操作符 （甚至出现圆括号）的表达式。这在 Kernighan 和 Ritchie 合著的 《*The C Programming Language*》的附录中明确指出。但是， ISO C 没有给编译器同样的自由。

本节通过考虑以下代码片段中的表达式语句，讨论这两个 C 定义之间的差异，并阐明表达式的副作用、分组以及求值之间的差别。

```
int i, *p, f(void), g(void);
/*...*/
i = *++p + f() + g();
```

7.10.1 定义

表达式的副作用是修改内存并访问 volatile 限定对象。以上表达式的副作用是更新 i 和 p 以及函数 f() 和 g() 内包含的任何副作用。

表达式的分组是将值与其它值和操作符合并的方式。以上表达式的分组主要是加法的执行顺序。

表达式的求值包括生成结果值所必需的一切。要对表达式求值，所有指定的副作用必须在上下两个序列点之间发生，并且使用特定的分组执行指定的操作。对于以上表达式，必须在前一个语句之后和该表达式语句的 ; 之前更新 i 和 p；函数调用可以在前一个语句之后的任何时候，但在使用它们的返回值之前按任何顺序发生。特别地，在使用操作的值之前，导致内存更新的操作符不需要分配新值。

7.10.2 K&R C 重新整理许可证

由于加法在算术上可交换并且关联，因此 K&R C 重新整理许可证适用于以上表达式。为了区别常规圆括号和表达式的实际分组，左、右花括号指定分组。表达式的三种可能的分组为：

```
i = { {*++p + f()} + g() };
i = { *++p + {f() + g()} };
i = { {*++p + g()} + f() };
```

给定 K&R C 规则，所有这些分组均有效。此外，即使表达式是按以下任意方式编写的，所有这些分组仍有效：

```
i = *++p + (f() + g());
i = (g() + *++p) + f();
```

如果在溢出导致异常的体系结构上对该表达式求值，或者加法和减法在溢出时不是互逆运算，则当一个加法运算溢出时，这三种分组表现不同。

对于这些体系结构上的此类表达式，K&R C 中唯一可用的求助措施是分割表达式以强制进行特定的分组。以下是分别强制执行以上三种分组的可能重写：

```
i = *++p; i += f(); i += g()
i = f(); i += g(); i += *++p;
i = *++p; i += g(); i += f();
```

7.10.3 ISO C 规则

对于算术上可交换并且关联但实际上在目标体系结构上并非如此的操作， ISO C 不允许进行重新整理。因此， ISO C 语法的优先级和关联性完整描述了所有表达式的分组；所有表达式在进行语法分析时必须进行分组。所考虑的表达式按以下方式分组：

```
i = { {*++p + f()} + g() };
```

此代码仍不表示必须在 `g()` 之前调用 `f()`，或者必须在调用 `g()` 之前增加 `p`。

在 ISO C 中，不需要为了避免意外的溢出而分割表达式。

7.10.4 圆括号

由于不完全的理解或不准确的表示， ISO C 经常被错误地描述为支持圆括号或根据圆括号求值。

由于 ISO C 表达式仅仅具有语法分析指定的分组，因此圆括号仍然仅用作控制表达式语法分析方式的方法；表达式的自然优先级和关联性与圆括号同等重要。

以上表达式可写为：

```
i = (((*(++p)) + f()) + g());
```

对其分组或求值没有不同影响。

7.10.5 As If 规则

K&R C 重新整理规则的几个理由：

- 重新整理提供更多的优化机会，如编译时常量折叠。
- 在大多数机器上，重新整理不更改整型表达式的结果。
- 在所有机器上，一些操作在算术上和计算上可交换并且关联。

ISO C 委员会最终承认：重新整理规则应用于所描述的目标体系结构时，本来是要作为格式的一个实例。 ISO C 的格式是通用许可证，它允许实现任意偏离抽象机器描述，只要偏离不更改有效 C 程序的行为。

因此，由于无法通知此类重新分组，因此允许在任何机器上重新整理所有二元按位运算符 （移位除外）。同样原因，在充满了溢出的典型的 2 的补码机器上，可以重新整理包含乘法或加法的整数表达式。

因此， C 中的这种更改对大多数 C 程序员不会产生重大影响。

7.11 不完全类型

ISO C 标准引入术语 “不完全类型”使 C 的基本（但容易造成误解）部分形式化，这种类型的开头具有某种暗示。本节描述不完全类型、其允许位置以及它们有用的原因。

7.11.1 类型

ISO 将 C 的类型分为三个不同的集合：函数、对象和不完全。函数类型很明显；对象类型包含其它一切，除非不知道对象的大小。该标准使用术语 “对象类型”指定指派的对象必须具有已知大小，但是除 `void` 之外的不完全类型也称为对象，知道这一点很重要。

不完全类型有三种不同形式：`void`、未指定长度的数组以及具有指定内容的结构和联合。 `void` 类型与其它两种类型不同，它是无法完成的不完全类型，并且作为特殊函数返回和参数类型。

7.11.2 完成不完全类型

通过在表示相同对象的相同作用域中的后面声明中指定数组大小，可完成数组类型。当声明并在相同声明中初始化不具有大小的数组时，仅在其声明符的末尾与其初始化函数的末尾之间，数组才具有不完全类型。

通过在具有相同标记的相同作用域中的后面声明中指定内容，可完成不完全结构或联合类型。

7.11.3 声明

某些声明可使用不完全类型，但是其它声明需要完全对象类型。需要对象类型的声明是数组元素、结构或联合的成员以及函数的局部对象。所有其它声明允许不完全类型。特别地，允许下列构造：

- 指向不完全类型的指针
- 返回不完全类型的函数
- 不完全函数参数类型
- 不完全类型的 `typedef` 名称

函数返回和参数类型特殊。除 void 之外，在定义或调用函数之前，必须完成以这种方式使用的不完全类型。void 的返回类型指定不返回值的函数，void 的单个参数指定不接受参数的函数。

由于数组和函数的参数类型重写为指针类型，因此表面上不完全的数组参数类型实际上并非不完全。 main 的 argv 的典型声明 （即 char *argv[]，一个未指定长度的字符指针数组）重写为指向字符指针的指针。

7.11.4 表达式

大多数表达式运算符需要完全对象类型。仅有的三个例外是一元运算符 &、逗号运算符的第一个操作数以及 ?: 运算符的第二个和第三个操作数。除非需要指针运算，否则接受指针操作数的大多数运算符也允许指向不完全类型的指针。该列表包含一元运算符 *。例如，给定：

```
void *p
```

&*p 是使用该声明的有效子表达式。

7.11.5 正当理由

为什么不完全类型是必要的？ 在忽略 void 的情况下，只有一个由不完全类型提供的功能是 C 无法以其它方式处理的，而必须利用对结构和联合的正向引用。如果两个结构需要相互指向的指针，则唯一的方法是使用不完全类型：

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

具有某种形式的指针以及异构数据类型的所有强类型编程语言提供处理这种情形的某些方法。

7.11.6 示例

为不完全结构和联合定义 typedef 名称通常很有用。如果您有一系列包含许多相互指向的指针的复杂数据结构，结构前面有一个 typedef 列表 （可能在中央头文件中），则可以简化声明。

```
typedef struct item_tag Item;
typedef union note_tag Note;
typedef struct list_tag List;
.  .  .
struct item_tag { .  .  .  };
.  .  .
struct list_tag {
    struct list_tag {
};
```

此外，对于其内容不应该用于程序其余部分的结构和联合，头文件可以声明不带该内容的标记。程序的其它部分可以使用指向不完全结构或联合的指针而不会出现任何问题，除非它们尝试使用它的任何成员。

频繁使用的不完全类型是未指定长度的外部数组。通常，没有必要知道使用其内容的数组的范围。

7.12 兼容类型和复合类型

对于 K&R C，甚至对于 ISO C，引用相同实体的两个声明可能不相同。 ISO C 中使用的术语 “兼容类型”表示 “足够接近”的类型。本节描述兼容类型和 “复合类型” — 合并两种兼容类型而产生的结果。

7.12.1 多个声明

如果只允许 C 程序声明每个对象或函数一次，则不需要兼容类型。链接 （允许两个或更多声明引用相同实体）、函数原型和分别编译全部需要此功能。独立转换单元 （源文件）具有与单个转换单元不同的类型兼容性规则。

7.12.2 分别编译兼容性

由于每个编译可能查看不同的源文件，因此独立编译中的大多数兼容类型规则实质上是结构化的：

- 匹配标量 （整型、浮点和指针）类型必须兼容，如同它们在相同的源文件中一样。
- 匹配结构、联合和枚举必须具有相同数目的成员。每个匹配成员必须具有兼容类型（在分别编译的意义上），包括位字段宽。
- 匹配结构必须具有相同顺序的成员。联合和枚举成员的顺序并不重要。
- 匹配枚举成员必须具有相同的值。

  附加要求是，对于结构、联合和枚举，成员的名称 （包括缺少未命名成员的名称）必须匹配，但是它们各自的标记不必匹配。

7.12.3 单编译兼容性

当相同作用域内的两个声明描述相同的对象或函数时，这两个声明必须指定兼容类型。然后这两种类型合并为与这两种类型兼容的单个复合类型。后面将详细讨论复合类型。

兼容类型是递归定义的。底部为类型说明符关键字。规则规定， `unsigned short` 与 `unsigned short int` 相同，不带类型说明符的类型与带有 `int` 的类型相同。所有其它类型仅当派生它们的类型兼容时才为兼容类型。例如，如果限定符 `const` 和 `volatile` 相同，并且非限定基类型兼容，则两个限定类型兼容。

7.12.4 兼容指针类型

要使两种指针类型兼容，它们指向的类型必须兼容，并且必须对这两个指针进行相同的限定。考虑到指针的限定符在 `*` 之后指定，因此以下两个声明

```
int *const cpi;
int *volatile vpi;
```

声明指向相同类型 `int` 的两个不同限定的指针。

7.12.5 兼容数组类型

要使两个数组类型兼容，它们的元素类型必须兼容。如果两个数组类型均具有指定的长度，则它们必须匹配，换句话说，一个不完全数组类型 （参见第 7-29 页上的第 7.11 节“不完全类型”）与另一个不完全数组类型以及指定长度的数组类型兼容。

7.12.6 兼容函数类型

要使函数兼容，请遵守以下规则：

- 要使两个函数类型兼容，它们的返回类型必须兼容。如果其中一个或两个函数类型均具有原型，则规则更加复杂。
- 要使两个带有原型的函数兼容，它们也必须具有相同数目的参数，包括使用省略号 (...) 表示法，并且相应的参数必须兼容。
- 要使旧风格函数定义与带有原型的函数类型兼容，原型参数*不能*以省略号 (...) 结尾。在应用缺省参数提升之后，每个原型参数必须与相应的旧风格参数兼容。
- 要使旧风格函数声明 （而不是定义）与带有原型的函数类型兼容，原型参数不能以省略号 (...) 结尾。所有原型参数的类型必须不受缺省参数提升的影响。
- 要使两种类型的参数兼容，在删除顶层限定符 （如果有）并且函数或数组已转换为相应的指针类型之后，这两种类型必须兼容。

7.12.7 特殊情况

`signed int` 的行为与 `int` 的行为相同，只是可能位字段除外，在位字段中，无格式 `int` 可表示无符号行为数量。

另一点值得注意的是，每个枚举类型必须与某些整数类型兼容。对于可移植的程序，这意味着枚举类型是独立类型。通常， ISO C 标准将枚举类型视为独立类型。

7.12.8 复合类型

由两个兼容类型构成的复合类型也是递归定义的。由于不完全数组或旧风格函数类型，兼容类型各不相同。因此，复合类型最简单的描述是，它是与两个原始类型均兼容的类型，包括原始类型中的各个可用数组长度和各个可用参数列表。

第 8 章

转换应用程序以适用于 64 位环境

本章提供针对 32 位或 64 位编译环境编写代码所需的信息。

您尝试针对 32 位或 64 位编译环境编写或修改代码时，将面对以下两个基本问题：

- 不同数据类型模型之间的数据类型一致性
- 使用不同数据类型模型的应用程序之间的交互作用

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好。因此，本章提供编写在 32 位和 64 位编译环境中正常工作的代码的指导。在某些情况下，当前代码的转换仅需要重新编译和重新链接 64 位库。但是，对于需要更改代码的情况，本章讨论使转换更容易的工具和策略。

8.1 数据模型差异概述

32 位和 64 位编译环境之间的最大差异是数据类型模型的更改。

32 位应用程序的 C 数据类型模型是 ILP32 模型，如此命名是因为整型、长型和指针均为 32 位数据类型。LP64 数据模型（如此命名是因为长型和指针增长为 64 位）由业界公司联盟创建。其余 C 类型 `int`、 `long long`、 `short` 和 `char` 在两种数据类型模型中相同。

无论是何种数据类型模型， C 整数类型之间的标准关系保持为真：

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

下表列出基本 C 数据类型及其对于 ILP32 和 LP64 数据模型的相应长度 （位数）。

表 8-1 ILP32 和 LP64 的数据类型长度

C 数据类型	LP32	LP64
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64
`long long`	64	64
`pointer`	32	64
`enum`	32	32
`float`	32	32
`double`	64	64
`long double`	128	128

当前 32 位应用程序通常假定整型、指针和长型的长度相同。由于长型和指针的长度在 LP64 数据模型中更改，因此您需要注意，单是这种更改就可以导致许多 ILP32 至 LP64 转换问题。

此外，检查声明和强制类型转换变得很重要；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更改的影响。要充分表示您的意图，您需要显式声明常量的类型。您也可以在表达式中使用强制类型转换，以确保表达式按您想要的方式求值。特别是在符号扩展的情况下，显式强制类型转换对说明意图至关重要，此时这样做更有必要。

8.2 实现单一源代码

以下各节描述您可以用来编写支持 32 位和 64 编译的单一源代码的某些可用资源。

8.2.1 派生类型

使用系统派生类型使代码对于 32 位和 64 位编译环境均安全。通常，使用派生类型以适应更改是良好的编程实践。当您使用派生数据类型时，只有系统派生类型由于数据模型更改或某个端口而需要更改。

系统包含文件 <sys/types.h> 和 <inttypes.h> 包含有助于使 32 位和 64 位应用程序安全的常量、宏和派生类型。

8.2.1.1 <sys/types.h>

将 <sys/types.h> 包含在应用程序源文件中，以获取对 _LP64 和 _ILP32 的定义的访问权。此头文件还包含适当时应使用的多个基本派生类型。尤其是以下类型更为重要：

- clock_t 表示系统时钟时间。
- dev_t 用于设备号。
- off_t 用于文件大小和偏移。
- ptrdiff_t 是减去两个指针所得结果的带符号整数类型。
- size_t 反映内存中对象的大小（字节数）。
- ssize_t 由返回字节数或错误提示的函数使用。
- time_t 以秒计时。

所有这些类型在 ILP32 编译环境中保持为 32 位数量，在 LP64 编译环境中增长为 64 位数量。

8.2.1.2 <inttypes.h>

包含文件 <inttypes.h> 提供有助于使您的代码与显式设定长度并且独立于编译环境的数据项兼容的常量、宏和派生类型。它包含处理 8 位、16 位、32 位和 64 位对象的机制。该文件是新的 1999 ISO/IEC C 标准的一部分，文件内容反映了导致它包含在 1999 ISO/IEC C 标准中的建议。文件即将更新，以便完全与 1999 ISO/IEC C 标准一致。以下是对 <inttypes.h> 提供的基本功能的讨论：

- 固定宽度整数类型。
- 有用类型，如 uintptr_t
- 常量宏
- 限制
- 格式字符串宏

以下部分提供有关 <inttypes.h> 基本功能的详细信息。

固定宽度整数类型

<inttypes.h> 提供的固定宽度整数类型包括带符号整数类型（如 int8_t、int16_t、int32_t 和 int64_t）和无符号整数类型（如 uint8_t、uint16_t、uint32_t 和 uint64_t）。

定义为可容纳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,..., uint_least64_t。

对于循环计数器和文件描述符等操作，使用整数是安全的；对于数组索引，使用 long 也是安全的。然而，不要不加区别地使用这些固定宽度类型。对于以下各项的显式二进制表示法，使用固定宽度类型：

- 磁盘上的数据
- 通过数据线
- 硬件寄存器
- 二进制接口规范
- 二进制数据结构

有用类型，如 unintptr_t

<inttypes.h> 文件包含足以容纳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 的形式给出。此外，<inttypes.h> 提供 intmax_t 和 uintmax_t，它们是最长（位数）的带符号整数类型和无符号整数类型。

使用 uintptr_t 类型作为指针的整数类型而不是基本类型，如无符号 long。即使在 ILP32 和 LP64 数据模型中，无符号 long 与指针的长度相同，使用 uintptr_t 也意味着在数据模型更改时，只有 uintptr_t 的定义受到影响。这使您的代码可移植到许多其它系统中。它也是在 C 中更清楚地表达意图的方式。

当您要执行地址运算时， intptr_t 和 uintptr_t 类型对强制转换指针非常有用。因此，使用 intptr_t 和 uintptr_t 类型而不是 long 或无符号 long。

常量宏

使用宏 INT8_C(c), ..., INT64_C(c), UINT8_C(c),..., UINT64_C(c) 指定给定常量的长度和符号。基本上，如有必要，这些宏在常量的末尾放置一个 l、ul、ll 或 ull。例如，对于 ILP32， INT64_C(1) 在常量 1 后面附加 ll ； 对于 LP64，则附加 l。

使用 INTMAX_C(c) 和 UINTMAX_C(c) 宏使常量成为最长的类型。这些宏对于指定第 8-6 页上的第 8.3 节 “转换为 LP64 数据类型模型”中描述的常量的类型很有用。

限制

<inttypes.h> 定义的限制是指定不同整数类型的最小值和最大值的常量。这包括每个固定宽度类型的最小值和最大值，如 INT8_MIN,..., INT64_MIN, INT8_MAX,..., INT64_MAX 及其无符号对应部分。

<inttypes.h> 文件还提供每个最短长度类型的最小值和最大值。它们包括 INT_LEAST8_MIN,..., INT_LEAST64_MIN, INT_LEAST8_MAX,..., INT_LEAST64_MAX 及其无符号对应部分。

最后， <inttypes.h> 还定义支持的最长整数类型的最小值和最大值。它们包括 INTMAX_MIN 和 INTMAX_MAX 及其相应的无符号版本。

格式字符串宏

<inttypes.h> 文件还包括指定 printf(3S) 和 scanf(3S) 格式说明符的宏。实质上，假设参数的位数已内置于宏的名称中，这些宏在格式说明符前面放置一个 l 或 ll 将参数标识为 long 或 long long。

printf(3S) 的宏以十进制、八进制、无符号和十六进制格式打印最短和最长整数类型，如以下示例所示：

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

同样， scanf(3S) 的宏以十进制、八进制、无符号和十六进制格式读取最短和最长整数类型。

```
uint64_t u;
scanf("%" SCNu64 "\n", &u);
```

不要不加区别地使用这些宏。它们最适用于第 8-4 页上的 “固定宽度整数类型”中讨论的固定宽度类型。

8.2.2 工具

lint 程序的 -errchk 选项检测潜在的 64 位移植问题。此外， C 编译器的 -v 选项执行更严格的附加语义检查。 -v 选项还启用对指定文件的某些类 lint 检查。

当您将代码增强为 64 位安全时，请使用 Solaris 操作环境中的头文件，原因是这些文件具有适用于 64 位编译环境的派生类型和数据结构的正确定义。

8.2.2.1 lint

使用 lint 检查针对 32 位和 64 位编译环境编写的代码。指定 -errchk=longptr64 选项以生成 LP64 警告。另外使用 -errchk=longptr64 标志，该标志用于检查是否可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。即使使用显式强制类型转换， -errchk=longptr64 标志也检查指针表达式和长整数表达式对无格式整数的赋值。

使用 -errchk=longptr64,signext 选项查找符合以下条件的代码：其中标准 ISO C 值保留规则允许在无符号整型表达式中使用带符号整数值的符号扩展。

当您要检查您只想在 64 位编译环境中运行的代码时，请使用 lint 的 -Xarch=v9 选项。

当 lint 生成警告时，它将打印错误代码的行号、描述问题的消息以及是否涉及指针。警告消息还指明涉及的数据类型的长度。当您知道涉及指针并且知道数据类型的长度时，您可以查找特定的 64 位问题并避免 32 位和更短类型之间的先存在问题。

但是请注意，即使 lint 发出关于潜在 64 位问题的警告，它也不能检测到所有问题。另外在许多情况下，符合应用程序意图且正确无误的代码会生成警告。

您可以通过在上一行中放置 "NOTE(LINTED("<optional message">))" 形式的注释，禁止对指定代码行发出警告。当您想让 lint 忽略某些代码行 （如强制类型转换和赋值）时，这样做很有用。使用 "NOTE(LINTED("<optional message">))" 注释时要格外谨慎，因为它可能掩盖真实问题。在使用 NOTE 时，请包含 #include<note.h>。有关详细信息，请参见 lint 手册页。

8.3 转换为 LP64 数据类型模型

以下示例说明在转换代码时可能遇到的某些常见问题。适当时会显示相应的 lint 警告。

8.3.1 整型和指针长度更改

由于整型和指针在 ILP32 编译环境中长度相同，因此某些代码依赖以下假定。指针通常强制转换为 `int` 或 `unsigned int` 以进行地址运算。但是，由于 `long` 和指针在 ILP32 和 LP64 数据类型模型中长度均相同，因此将指针强制转换为 `long`。并不是显式使用 `unsigned long`，而是使用 `uintptr_t`，因为它更贴切地表达您的意图并使代码具有更大的可移植性，从而将它与将来更改隔离。考虑以下示例：

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
%
警告： 指针转换丢失位
```

下面是修改的版本：

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

8.3.2 整型和长型长度更改

由于整型和长型在 ILP32 数据类型模型中从未真正加以区分，因此您的现有代码可能不加区别地使用它们。修改交换使用整型和长型的代码，使其符合 ILP32 和 LP64 数据类型模型的要求。整型和长型在 ILP32 数据类型模型中均为 32 位，而长型在 LP64 数据类型模型中为 64 位。考虑以下示例：

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;

%
警告： 64 位整数向 32 位整数赋值
```

8.3.3 符号扩展

由于类型转换和提升规则有些模糊，因此在转换到 64 位编译环境时，符号扩展是常见问题。为防止出现符号扩展问题，请使用显式强制类型转换以取得预期结果。

要了解发生符号扩展的原因，了解 ISO C 的转换规则会有所帮助。可能导致在 32 位和 64 位编译环境之间发生最多符号扩展问题的转换规则在以下操作过程中生效：

- 整型提升

  无论有无符号，您均可以在任何调用整数的表达式中使用 `char`、 `short`、 `enumerated type` 或位字段。

  如果一个整数可以容纳初始类型的所有可能值，则值转换为整数；否则，值转换为无符号整数。

- 带符号整数和无符号整数之间的转换

  当一个带负号的整数被提升为同一类型或更长类型的无符号整数时，它首先被提升为更长类型的带符号等价值，然后转换为无符号值。

当以下示例作为 64 位程序编译时，尽管 `addr` 和 `a.base` 均为无符号类型， `addr` 变量仍变为符号扩展变量。

```
%cat 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;  /* 此处存在符号扩展！ */
  printf("addr 0x%lx\n", addr);

 addr = (unsigned int)(a.base << 13); /* 此处不存在符号扩展！ */
 printf("addr 0x%lx\n", addr);
}
```

发生此符号扩展的原因是按以下方式应用了转换规则：

- `a.base` 由于整型提升规则而从无符号 `int` 转换为 `int`。这样，表达式 `a.base << 13` 属于类型 `int`，但是未发生符号扩展。

- 表达式 a.base << 13 属于类型 int，但是在赋值给 addr 之前，由于带符号和无符号整数提升规则而转换为 long，然后转换为无符号 long。从 int 转换为 long 时，发生符号扩展。

```
% cc -o test64 -xarch=v9 test.c
% ./test64
addr 0xffffffff80000000
addr 0x80000000
%
```

这个相同示例作为 32 位程序编译时，不显示任何符号扩展：

```
cc -o test test.c
%test

addr 0x80000000
addr 0x80000000
```

有关转换规则的详细讨论，请参见 ISO C 标准。此标准中还包含对普通算术转换和整型常量有用的规则。

8.3.4 指针运算而不是整数

通常，由于指针运算独立于数据模型，而整数不可以，因此使用指针运算比整数好。此外，通常可以使用指针运算简化代码。考虑以下示例：

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

%
警告： 指针转换丢失位
```

下面是修改的版本：

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

8.3.5 结构

检查应用程序中的内部数据结构有无漏洞。在结构中的字段之间使用额外填充，以满足对齐要求。当长型或指针字段增长到适用于 LP64 数据类型模型的 64 位时，会分配此额外填充。在 SPARC 平台上的 64 位编译环境中，所有类型的结构均与结构中的最长成员的长度对齐。当您重组结构时，请遵循将长型和指针字段移到结构开头的简单规则。考虑以下结构定义：

```
struct bar {
   int i;
   long j;
   int k;
   char *p;
};   /* sizeof (struct bar) = 32 */
```

下面是在结构开头定义了长型和指针数据类型的相同结构：

```
struct bar {
  char *p;
  long j;
  int i;
  int k;
};   /* sizeof (struct bar) = 24 */
```

8.3.6 联合

请务必检查联合，因为其字段的长度在 ILP32 和 LP64 数据类型模型之间会更改。

```
typedef union {
   double _d;
   long _l[2];
} llx_t;
```

下面是修改的版本

```
typedef union {
   double _d;
   int _l[2];
} llx_t;
```

8.3.7 类型常量

在某些常量表达式中，缺少精度会导致数据丢失。请在常量表达式中显式指定数据类型。通过增加 {u,U,l,L} 的组合指定每个整型常量的类型。您也可以使用强制类型转换来指定常量表达式的类型。考虑以下示例：

```
int i = 32;
long j = 1 << i; /* j 将获得 0，原因是 RHS 为整型 */
                                    /* 表达式 */
```

下面是修改的版本：

```
int i = 32;
long j = 1L << i;
```

8.3.8 注意隐式声明

如果您使用 -xc99=%none，则 C 编译器假定在模块中使用但未在外部定义或声明的函数或变量为整数。编译器的隐式整数声明会将以此方式使用的任何长型和指针截尾。将函数或变量的相应外部声明置于头文件而非 C 模块中。在使用函数或变量的 C 模块中包含此头文件。如果它是系统头文件定义的函数或变量，您还需要在代码中包含正确的头文件。考虑以下示例：

```
int
main(int argc, char *argv[])
{
  char *name = getlogin()
  printf("login = %s\n", name);
  return (0);
}

%
警告： 不正确的指针/整型组合： op "="
警告： 从 32 位整数转换为指针
隐式声明为返回 int
getlogin        printf
```

修改的版本中现在有正确的头文件

```
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char *name = getlogin();
  (void) printf("login = %s\n", name);
  return (0);
}
```

8.3.9 sizeof( ) 是无符号 long

在 LP64 数据类型模型中， sizeof() 的有效类型为无符号长型。有时， sizeof() 会传递给需要 int 类型的参数的函数，或者赋值给整数或强制转换为整数。有些情况下，这种截尾会导致数据丢失。

```
long a[50];
unsigned char size = sizeof (a);

%
警告： 64 位常量赋值后被截断为 8 位
警告： 初始化函数不适合或溢出： 0x190
```

8.3.10 使用强制类型转换显示您的意图

关系表达式可能会因为转换规则而显得错综复杂。您应该通过在必要的地方增加强制类型转换很明确地指定表达式的求值方式。

8.3.11 检查格式字符串转换操作

确保 printf(3S)、sprintf(3S)、scanf(3S) 和 sscanf(3S) 的格式字符串可以容纳长型或指针参数。对于指针参数，要在 32 位和 64 位编译环境中运行，格式字符串中给定的转换操作应为 %p。

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%x", (void *)devi);

%
警告： 函数参数 （数目）类型与格式不一致
sprintf (arg 3)      void *: (format) int
```

下面是修改的版本

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, ‘di%p", (void *)devi);
```

对于长型参数，长型长度规范 l 应前置于格式字符串中的转换操作字符前面。另外，还要检查以确保 buf 指向的存储器足以包含 16 个数字。

```
size_t nbytes;
u_long 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);

%
警告： 64 位整数向 32 位整数的强制类型转换
警告： 64 位整数向 32 位整数的强制类型转换
警告： 64 位整数向 32 位整数的强制类型转换
```

下面是修改的版本

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%lu%%%lu from heap got%lx.%lx returns%lx\n",
nbytes, align, raddr, raddr + alloc, addr);
```

8.4 其它考虑事项

其余指导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。

8.4.1 长度增长的派生类型

很多派生类型已更改，目前它们在 64 位应用程序编译环境中表示 64 位数量。此更改不影响 32 位应用程序；然而，使用或输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。关于这一点的一个示例是直接处理 `utmp`(4) 或 `utmpx`(4) 文件的应用程序。为了在 64 位应用程序环境中正确操作，不要试图直接访问这些文件。而使用 `getutxent`(3C) 及相关的函数系列。

8.4.2 检查更改的副作用

请注意，一个区域中的类型更改会导致另一个区域中发生意外的 64 位转换。例如，检查以前返回 `int` 而现在返回 `ssize_t` 的函数的所有调用程序。

8.4.3 检查 `long` 的文字使用是否仍有意义

定义为 `long` 的变量在 ILP32 数据类型模型中为 32 位，在 LP64 数据类型模型中为 64 位。如有可能，通过重新定义变量并使用可移植性更强的派生类型避免出现问题。

与此相关，很多派生类型在 LP64 数据类型模型中已更改。例如，`pid_t` 在 32 位环境中仍为 `long`，但是在 64 位环境中， `pid_t` 为 `int`。

8.4.4 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。您可以通过在头文件中指定 `_LP64` 或 `_ILP32` 功能测试宏区分它们。同样，在 32 位和 64 位环境中运行的代码需要利用相应的 `#ifdefs`，取决于编译代码。

8.4.5 调用转换更改

当您通过值传递结构并针对 SPARC V9 编译代码时，若结构足够小，则通过寄存器传递结构而不是将结构作为副本的指针。如果您尝试在 C 代码与手写汇编代码之间传递结构，这会导致问题。

浮点参数的工作方式类似；有些通过值传递的浮点值通过浮点寄存器传递。

8.4.6 算法更改

在代码对 64 位环境是安全的之后，请再次检查代码，以检验算法和数据结构是否仍有意义。数据类型较长，因此数据结构可能占用更多空间。代码的性能也可能变化。出于这些利害关系考虑，您可能需要相应地修改代码。

8.5 入门指南清单

使用以下清单有助于将代码转换为 64 位。

- 检查所有数据结构和接口，以检验它们在 64 位环境中是否仍有效。
- 在代码中包含 `<inttypes.h>`，以获取 `_ILP32` 或 `_LP64` 定义以及多种基本派生类型。系统程序可能需要包含 `<sys/types.h>`（或至少包含 `<sys/isa_defs.h>`），以获取 `_ILP32` 或 `_LP64` 的定义。
- 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,signext` 和 `-D__sparcv9` 标志运行 lint，并分别检查每个警告。请注意，并非所有警告均需要更改代码。根据更改，在 32 位和 64 位模式下再次运行 lint。
- 除非应用程序仅作为 64 位应用程序提供，否则将代码编译为 32 位和 64 位。
- 通过在 32 位操作系统上执行 32 位版本和在 64 位操作系统上执行 64 版本来测试应用程序。您也可以在 64 位操作系统上测试 32 位版本。

第 9 章

cscope: 交互检查 C 程序

cscope 是一个交互式程序，它定位 C、 lex 或 yacc 源文件中代码的指定元素。使用 cscope，您可以比使用典型编辑器时更有效率地搜索和编辑源文件。这是因为 cscope 支持函数调用 — 当某个函数被调用时，当它执行调用时 — 以及 C 语言标识符和关键字。

本章是关于此发行版附带的 cscope 浏览器的教程。

注 – cscope 程序尚未更新，无法理解针对 1999 ISO/IEC C 标准编写的代码。例如，它尚不能识别 1999 ISO/IEC C 标准中引入的新关键字。

9.1 cscope 进程

当您为一组 C、 lex 或 yacc 源文件调用 cscope 时，该应用程序会为这些文件中的函数、函数调用、宏、变量以及预处理程序符号生成一个符号交叉引用表。然后您可以查询该表，了解您指定的符号的位置。首先，它显示一个菜单，要求您选择要执行的搜索的类型。例如，您可能想让 cscope 查找调用某个指定函数的所有函数。

cscope 完成搜索后，将打印一个列表。每个列表条目均包含文件名、行号以及 cscope 在其中找到指定代码的行的文本。在我们的示例中，列表还包括调用指定函数的函数的名称。您可以选择请求执行另一次搜索或使用编辑器检查某个列出的行。如果您选择后者， cscope 将为包含该行的文件调用编辑器，并使光标位于该行。您现在可以在上下文中查看代码，并在需要时将文件作为任何其它文件进行编辑。然后您可以从编辑器返回菜单，请求执行新搜索。

由于执行的过程取决于现行任务，因此没有单独集中说明如何使用 cscope。有关其用法的详细示例，请查阅下一节中描述的 cscope 会话。它显示如何定位程序中的错误而无需了解所有代码。

9.2 基本用法

假设您负责维护程序 prog。您被告知错误消息 “out of storage”有时在程序启动时出现。现在您要使用 cscope 定位生成该消息的代码部分。您可以按以下方式执行。

9.2.1 步骤 1：设置环境

cscope 是一个面向屏幕的工具，只能在终端信息公用程序 (terminfo) 数据库中列出的终端上使用。确保将 TERM 环境变量设置为您的终端类型，以便 cscope 能够检验它是否在 terminfo 数据库中列出。如果您尚未这样做，请为 TERM 赋值并将其输出到 shell，如下所示：

在 Bourne shell 中，键入：

```
$ TERM=term_name; export TERM
```

在 C shell 中，键入：

```
% setenv TERM term_name
```

现在您可能要为 EDITOR 环境变量赋值。缺省情况下，cscope 调用 vi 编辑器。（本章中的示例说明了 vi 用法。）如果您不想使用 vi，请将 EDITOR 环境变量设置为您选择的编辑器并输出 EDITOR，如下所示：

在 Bourne shell 中，键入：

```
$ EDITOR=emacs; export EDITOR
```

在 C shell 中，键入：

```
% setenv EDITOR emacs
```

您可能必须编写 cscope 与您的编辑器之间的接口。有关详情，请参见第 9-18 页上的第 9.2.9 节 “编辑器的命令行语法”。

如果您要将 cscope 仅用于浏览 （不进行编辑），您可以将 VIEWER 环境变量设置为 pg 并输出 VIEWER。 cscope 将调用 pg 而不是 vi。

可以设置一个称为 VPATH 的环境变量，以指定为查找源文件而搜索的目录。参见第 9-13 页上的第 9.2.6 节 “视图路径”。

9.2.2 步骤 2：调用 cscope 程序

缺省情况下， cscope 为当前目录中的所有 C、 lex 和 yacc 源文件以及当前目录中或标准位置的任何包含的头文件生成一个符号交叉引用表。因此，如果要浏览的程序的所有源文件均位于当前目录，并且其头文件也位于此处或标准位置，请不带参数调用 cscope：

```
% cscope
```

要浏览选定的源文件，请将这些文件的名称用作参数调用 cscope：

```
% cscope file1.c file2.c file3.h
```

有关调用 cscope 的其它方法，请参见第 9-11 页上的第 9.2.5 节 “命令行选项”。

cscope 在第一次用于要浏览的程序的源文件时，将生成符号交叉引用表。缺省情况下，该表存储在当前目录中的文件 cscope.out 中。在后续调用中， cscope 仅在源文件被修改或源文件列表不同时重新生成交叉引用。重新生成交叉引用时，未更改的文件的数据从旧的交叉引用中复制，从而使重新生成比初始生成快，并减少后续调用的启动时间。

9.2.3 步骤 3：定位代码

现在让我们返回到本节开始时执行的任务：确定导致打印错误消息 “out of storage” 的问题。您已调用 cscope，交叉引用表已生成。 cscope 任务菜单显示在屏幕上。

cscope 任务菜单：

```
% cscope

cscope     Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

按 [Return] 键使光标在屏幕上向下移动 （显示屏底部有环绕），按 ^p (Control-p) 使光标向上移动；或者使用向上 (ua) 和向下 (da) 箭头键。您可以使用以下单键命令操作菜单以及执行其它任务：

表 9-1 cscope 菜单操作命令

Tab	移至下一个输入字段。
Return	移至下一个输入字段。
^n	移至下一个输入字段。
^p	移至上一个输入字段。
^y	使用上次键入的文本进行搜索。
^b	移至上一个输入字段并搜索模式。
^f	移至下一个输入字段并搜索模式。
^c	搜索时在不区分/区分大小写之间切换。例如，不区分大小写时，查找 FILE 将找到 file 和 File。
^r	重新生成交叉引用。
!	启动交互式 shell。键入 **^d** 以返回到 cscope。

表 9-1 cscope 菜单操作命令 *（续）*

^l	刷新屏幕。
?	显示命令列表。
^d	退出 cscope。

如果您要查找的文本的第一个字符与这些命令之一匹配，您可以通过在该字符之前输入 \（反斜杠）避免执行命令。

现在将光标移至第五个菜单项 “Find this text string”，输入文本 “out of storage”，然后按 [Return] 键。

cscope 函数：请求查找文本字符串：

```
$ cscope

cscope     Press the ? key for help


Find this C symbol
Find this global definition
Find functions called by this function
Find functions calling this function
Find this text string:  out of storage
Change this text string
Find this egrep pattern
Find this file
Find files #including this file
```

注 – 按照相同过程执行菜单中列出的除第六项 “Change this text string” 之外的任何其它任务。由于该任务比其它任务稍微复杂一些，因此要按照另一个过程执行。有关如何更改文本字符串的描述，请参见第 9-14 页上的第 9.2.8 节 “示例”。

cscope 查找指定的文本，找到包含该文本的一行，并报告其查找结果。

cscope 函数：列出包行文本字符串的行：

```
Text string: out of storage

  File Line
1 alloc.c 63 (void) fprintf(stderr, "\n%s:  out of storage\n",
argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

在 cscope 显示成功搜索的结果之后，您有多个选项。您可能要更改其中一行或在编辑器中检查该行周围的代码。或者，如果 cscope 找到很多行，以致这些行的列表不能一次显示在屏幕上，您可能要查看列表的下一部分。下表显示 cscope 找到指定文本之后可用的命令：

表 9-2 初次搜索之后可使用的命令

1 - 9	编辑该行引用的文件。您键入的数字对应于 cscope 打印的行列表中的一项。
空格	显示下一组匹配行。
+	显示下一组匹配行。
^v	显示下一组匹配行。
–	显示上一组匹配行。
^e	按顺序编辑显示的文件。
>	将显示的行列表附加至某个文件。
|	将所有行通过管道传送至某个 shell 命令。

同样，如果您要查找的文本的第一个字符与这些命令之一匹配，您可以通过在该字符之前输入反斜杠以避免执行命令。

现在，检查新找到的行周围的代码。输入 **1**（该行在列表中的编号）。系统通过文件 alloc.c 调用编辑器，并使光标位于 alloc.c 第 63 行的开头。

cscope 函数：检查代码行：

```
{
   return(alloctest(realloc(p, (unsigned) size)));
}

/* 检查内存分配故障 */

static char *
alloctest(p)
char *p;
{
    if (p == NULL) {
        (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);
        exit(1);
    }
    return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

您会发现变量 p 为 NULL 时，生成错误消息。要确定传递给 alloctest() 的变量如何为 NULL，首先必须识别调用 alloctest() 的函数。

使用标准退出惯例退出编辑器。您即返回至任务菜单。现在，请在第四项 “Find functions calling this function”后面键入 **alloctest**。

cscope 函数：请求调用 alloctest() 的函数的列表：

```
Text string: out of storage

  File Line
1 alloc.c 63(void)fprintf(stderr,"\n%s:  out of storage\n",argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

cscope 查找并列出三个此类函数。

cscope 函数：**列出**调用 alloctest() 的函数：

```
Functions calling this function: alloctest
File Function Line
1 alloc.c mymalloc 33 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc 43 return(alloctest(calloc((unsigned) nelem,
(unsigned) size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned)
size)));


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

现在您要知道哪些函数调用 mymalloc()。 cscope 查找十个此类函数。它会在屏幕上列出其中九个，并指示您按空格键以查看列表的其余部分。

cscope 函数：**列出**调用 mymalloc() 的函数：

```
Functions calling this function: mymalloc

File         Function       Line
1 alloc.c    stralloc       24 return(strcpy(mymalloc
                            (strlen(s) + 1), s));
2 crossref.c crossref       47 symbol = (struct symbol *)mymalloc
                            (msymbols * sizeof(struct symbol));
3 dir.c      makevpsrcdirs  63 srcdirs = (char **) mymalloc
                            (nsrcdirs * sizeof(char*));
4 dir.c      addincdir      167 incdirs = (char **)mymalloc
                            (sizeof(char *));
5 dir.c      addincdir      168 incnames = (char **)
                            mymalloc(sizeof(char *));
6 dir.c      addsrcfile     439 p = (struct listitem *) mymalloc
                            (sizeof(struct listitem));
7 display.c  dispinit       87 displine = (int *) mymalloc
                            (mdisprefs * sizeof(int));
8 history.c  addcmd         19  h = (struct cmd *) mymalloc
                            (sizeof(struct cmd));
9 main.c     main           212 s = mymalloc((unsigned )
                            (strlen(reffile) +strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

由于您知道错误消息 “out of storage”在程序开始时生成，因此您可以猜测问题可能发生在函数 dispinit()（显示初始化）中。

要查看 dispinit()（列表中第七个函数），请键入 **7**。

cscope 函数：在编辑器中查看 dispinit()：

```
void
dispinit()
{
         /* 计算显示的参考行最大数目 */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
       mdisprefs = 9;
    }
       /* 分配显示的行数组 */
   displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* 显示一页参考 */

void
display()
{
    char file[PATHLEN + 1]; /* 文件名 */
    char function[PATLEN + 1];/* 函数名 */
    char linenum[NUMLEN + 1]; /* 行号 */
    int screenline; /* 屏幕行号 */
    int width; /* 源代码行显示宽度 */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

mymalloc() 失败，原因是调用它时使用了很大的数或使用了负数。通过检查 FLDLINE 和 REFLINE 的可能值，您会发现 mdisprefs 的值为负的情况，也就是说，您正尝试使用负数调用 mymalloc()。

9.2.4 步骤 4：编辑代码

在开窗口终端上，您可能有多个任意大小的窗口。可能由于在行过少的窗口中运行 prog 而出现错误消息“out of storage”。换句话说，可能属于使用负数调用 mymalloc() 的情况之一。现在您要注意，将来程序在这种情况下异常终止时，在打印更有意义的错误消息“screen too small”之后，它会显示上述消息。按以下方式编辑函数 dispinit()。

cscope 函数：纠正问题：

```
void
dispinit()
{
/* 计算显示的参考行最大数目 */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
       mdisprefs = 9;
    }
/* 分配显示的行数组 */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* 显示一页参考 */

void
display()
{
    char file[PATHLEN + 1]; /* 文件名 */
    char function[PATLEN + 1];/* 函数名 */
    char linenum[NUMLEN + 1]; /* 行号 */
    int screenline; /* 屏幕行号 */
    int width; /* 源代码行显示宽度 */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

您已纠正我们在本节开始时讨论的问题。现在，如果 prog 在行过少的窗口中运行，不会简单地以无启示性的错误消息 “out of storage”而失败。相反，它在退出之前检查窗口大小并生成更有意义的错误消息。

9.2.5 命令行选项

如前所述， cscope 在缺省情况下为当前目录中的 C、 lex 和源文件生成一个符号交叉引用表。也就是说，

```
% cscope
```

等价于：

```
% cscope *.[chly]
```

我们还发现，您可以将选定的源文件的名称作为参数调用 cscope，以便浏览这些文件：

```
% cscope file1.c file2.c file3.h
```

cscope 提供一些命令行选项，这些命令行选项在指定要包含在交叉引用中的源文件方面具有更大的灵活性。当您使用 -s 选项和任意数目的目录名称 （用逗号隔开）调用 cscope 时：

```
% cscope -s dir1,dir2,dir3
```

cscope 为指定的目录中以及当前目录中的所有源文件生成交叉引用。要浏览其名称在*文件*中列出的所有源文件 （文件名用空格、制表符或换行符隔开），请使用 -i 选项以及包含列表的文件的名称调用 cscope：

```
% cscope -i file
```

如果源文件位于目录树中，请使用以下命令进行浏览：

```
% find . -name '*.[chly]' -print | sort > file
% cscope -i file
```

但是，如果选择了此选项， cscope 将忽略命令行中出现的任何其它文件。

对 cscope 使用 -I 选项与对 cc 使用 -I 选项方法相同。参见第 2-24 页上的第 2.13 节“如何指定包含文件”。

您可以通过调用 -f 选项，指定一个除缺省 cscope.out 之外的交叉引用文件。这对在同一目录中保持独立的符号交叉引用文件很有用。如果两个程序在同一目录中，但不共享所有相同文件，您可能需要这样做：

```
% cscope -f admin.ref admin.c common.c aux.c libs.c
% cscope -f delta.ref delta.c common.c aux.c libs.c
```

在本示例中， admin 和 delta 两个程序的源文件在同一目录中，但这两个程序由不同的文件组组成。当您为每组源文件调用 cscope 时，通过指定不同的符号交叉引用文件，使两个程序的交叉引用信息保持独立。

您可以使用 -p*n* 选项指定让 cscope 在列出搜索结果时显示路径名或路径名的一部分。您为 -p 指定的数代表您要显示的路径名的最后 *n* 个元素。缺省值为 1，即文件本身的名称。因此，如果您的当前目录为 home/common，则命令：

```
% cscope -p2
```

使 cscope 在列出搜索结果时显示 common/file1.c、 common/file2.c 等等。

如果您要浏览的程序包含大量源文件，您可以使用 -b 选项，使 cscope 在生成交叉引用之后停止； cscope 不显示任务菜单。当您在流水线中使用 cscope -b 并使用 batch(1) 命令时， cscope 在后台生成交叉引用：

```
% echo 'cscope -b' | batch
```

生成交叉引用之后，只要您未同时更改源文件或源文件列表，您只需指定：

```
% cscope
```

以便按正常方式复制交叉引用和显示任务菜单。如果您希望无需等待 cscope 完成其初始处理即可继续工作，您可以使用此命令序列。

-d 选项指示 cscope 不更新符号交叉引用。如果您确定未进行此类更改，您可以使用该选项以节省时间； cscope 不检查源文件是否更改。

注 – 使用 -d 选项时请小心。如果您错误地认为源文件无任何更改，并据此指定 -d，cscope 在响应您的查询时将引用过期的符号交叉引用。

有关其它命令行选项，请查看 cscope(1) 手册页。

9.2.6 视图路径

我们已经看到， cscope 缺省情况下在当前目录中搜索源文件。如果设置环境变量 VPATH， cscope 将在包含您的视图路径的目录中查找源文件。视图路径是有序的目录列表，每个目录下面具有相同的目录结构。

例如，假设您参与某个软件项目。在 /fs1/ofc 下面的目录中有一组*正式*源文件。每个用户都有一个起始目录 (/usr/you)。如果您更改软件系统，则 /usr/you/src/cmd/prog1 中可能只有您正在更改的文件的副本。完整程序的正式版本可在目录 /fs1/ofc/src/cmd/prog1 中找到。

假设您使用 cscope 浏览包含 prog1 的三个文件，即 f1.c、 f2.c 和 f3.c。您需要将 VPATH 设置为 /usr/you 和 /fs1/ofc 并将其输出，如下所示：

在 Bourne shell 中，键入：

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

在 C shell 中，键入：

```
% setenv VPATH /usr/you:/fs1/ofc
```

然后创建当前目录 /usr/you/src/cmd/prog1，并调用 cscope：

```
% cscope
```

程序将定位视图路径中的所有文件。如果找到重复文件， cscope 使用其父目录在 VPATH 中较早出现的文件。因此，如果 f2.c 在您的目录中，并且所有三个文件在正式目录中， cscope 将从您的目录中检查 f2.c，并从正式目录中检查 f1.c 和 f3.c。

VPATH 中的第一个目录必须为您将在其中工作的目录的前缀，通常为 $HOME。 VPATH 中的每个以冒号分隔的目录均必须为绝对目录：它应该以 / 开头。

9.2.7 cscope 和编辑器调用栈

cscope 和编辑器调用可以形成栈。也就是说，当 cscope 使您进入编辑器查看某个符号的引用时，如果存在另一个相关引用，您可以从编辑器内部再次调用 cscope 以查看第二个引用，而无需退出对 cscope 或编辑器的当前调用。然后，您可以通过使用相应的 cscope 命令和编辑器命令退出，执行倒退操作。

9.2.8 示例

本节给出如何使用 cscope 执行以下三种任务的示例：将常量更改为预处理程序符号、向函数增加参数以及更改变量的值。第一个示例说明了更改文本字符串的过程，这与 cscope 菜单上的其它任务稍有不同。也就是说，在您输入要更改的文本字符串之后，cscope 会提示您输入新文本，显示包含旧文本的行，并等待您指定要更改的行。

9.2.8.1 将常量更改为预处理程序符号

假设您要将常量 100 更改为预处理程序符号 MAXSIZE。选择第六个菜单项 “Change this text string”，然后输入 **\100**。必须使用反斜杠将 1 换码，原因是它对 cscope 具有特殊意义 （菜单上的第 1 项）。现在按 [Return] 键。 cscope 提示您输入新文本字符串。键入 **MAXSIZE**。

cscope 函数：更改文本字符串：

```
cscope        Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

cscope 显示包含指定文本字符串的行，并等待您选择要更改其中文本的行。

cscope 函数：提示要更改的行：

```
cscope        Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

您知道列表中行 1、2 和 3（列出的源文件的行 4、26 和 8）中的常量 100 应更改为 MAXSIZE。您还知道 read.c 中的 0100 和 err.c 中的 100.0（列表中的行 4 和 5）不应更改。您可以使用以下单键命令选择要更改的行：

表 9-3 用于选择要更改的行的命令

1-9	标记要更改的行或去标记。
*	标记要更改的所有显示行或去标记。
空格	显示下一组行。
+	显示下一组行。
-	显示上一组行。
a	标记要更改的所有行。
^d	更改标记的行并退出。
Esc	退出而不更改标记的行。

在这种情况下，输入 **1**、**2** 和 **3**。您键入的数不会打印在屏幕上。相反，cscope 通过在列表中您要更改的每个列表项的行号后面打印 >（大于）符号来标记这些项。

cscope 函数：标记要更改的行：

```
Change "100" to "MAXSIZE"

  File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* 获得百分比 */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

现在键入 **^d** 以更改选定的行。 cscope 显示已更改的行并提示您继续。

cscope 函数：显示已更改的文本行：

```
Changed lines:

    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {

Press the RETURN key to continue:
```

当您响应此提示而按 [Return] 键时， cscope 将刷新屏幕，从而使其恢复到您选择要更改的行之前的状态。

下一步是为新符号 MAXSIZE 增加 #define。由于将要显示 #define 的头文件不在其行得到显示的文件之列，因此您必须通过键入 **!** 退出到 shell。shell 提示符出现在屏幕底部。然后进入编辑器，并增加 #define。

cscope 函数：退出到 Shell：

```
Text string: 100

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;             /* 获得百分比 */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

要恢复 cscope 会话，请退出编辑器并键入 **^d** 以退出 shell。

9.2.8.2 向函数增加参数

向函数增加参数包括两个步骤：编辑函数本身，以及向代码中调用函数的各个位置增加新参数。

首先，通过使用第二个菜单项“Find this global definition”编辑函数。接着，找出调用函数的位置。使用第四个菜单项“Find functions calling this function”获取调用此函数的所有函数的列表。使用此列表，您可以通过分别输入每行在列表中的编号为该行调用编辑器，或通过键入 **^e** 自动为所有行调用编辑器。使用 cscope 进行此类更改可确保您需要编辑的任何函数均不会被忽略。

9.2.8.3 更改变量的值

有时，您可能要了解建议的更改如何影响代码。

假设您要更改变量或预处理程序符号的值。在这样做之前，请使用第一个菜单项“Find this C symbol”获取受到影响的引用的列表。然后使用编辑器检查每个引用。此步骤有助于您预测您建议的更改的总体影响。随后，您可以按同样的方式使用 cscope，以验证已进行的更改。

9.2.9 编辑器的命令行语法

缺省情况下， cscope 调用 vi 编辑器。您可以通过将首选编辑器指定给 EDITOR 环境变量并输出 EDITOR（如第 9-2 页上的第 9.2.1 节“步骤 1：设置环境”中所述）来覆盖缺省设置。但是， cscope 要求它使用的编辑器具有以下形式的命令行语法：

```
% editor +linenum filename
```

与 vi 一样。如果您要使用的编辑器没有这种命令行语法，您必须编写 cscope 与编辑器之间的接口。

假设您要使用 ed。由于 ed 不允许在命令行上指定行号，因此，除非您编写一个包含以下行的 shell 脚本，否则不能通过 cscope 使用该编辑器查看或编辑文件：

```
/usr/bin/ed $2
```

让我们将 shell 脚本命名为 myedit。现在，将 EDITOR 的值设置为您的 shell 脚本并输出 EDITOR：

在 Bourne shell 中，键入：

```
$ EDITOR=myedit; export EDITOR
```

在 C shell 中，键入：

```
% setenv EDITOR myedit
```

当 cscope 为您指定的列表项 （如 main.c 中的行 17）调用编辑器时，它使用以下命令行调用您的 shell 脚本：

```
% myedit +17 main.c
```

myedit 然后丢弃行号 ($1) 并使用文件名 ($2) 正确调用 ed。当然，您不会自动移动到文件的第 17 行，而必须执行相应的 ed 命令以显示和编辑该行。

9.3 未知终端类型错误

如果您看到以下错误消息：

```
Sorry, I don't know how to deal with your "term" terminal
```

您的终端可能未在当前装入的终端信息公用程序 (terminfo) 数据库中列出。请确保将正确的值赋给 TERM。如果该消息再出现，请尝试重新装入终端信息公用程序。

如果显示以下消息：

```
Sorry, I need to know a more specific terminal type than "unknown"
```

请按第 9-2 页上的第 9.2.1 节 “步骤 1：设置环境”中所述设置并输出 TERM 变量。

附录 A

C 编译器选项

本章描述 C 编译器选项。请注意，缺省情况下，C 编译器识别 1999 ISO/IEC C 标准的某些构造。特别是，在第 D-1 页上的 “支持的 C99 功能”中详细说明了所支持的功能。如果要用 1990 ISO/IEC C 标准限制编译器，请使用 `-xc99=%none` 命令。

如果您将 K&R C 程序移植到 ISO C 中，需要特别注意兼容性标志部分，即第 A-27 页上的第 A.3.59 节 “`-x[c|a|t|s]`”。使用它们可更容易地向 ISO C 转换。参见第 7 章中对转换的讨论。

A.1 选项语法

`cc` 命令的语法是：

```
% cc [options] filenames [libraries]...
```

其中：

- *options* 表示在第 A-9 页上的第 A.3 节 “`cc` 选项”中描述的一个或多个选项。
- *filenames* 表示在生成可执行程序过程中使用的一个或多个文件。

  C 编译器接受包含在由 *filenames* 指定的文件列表中的 C 源文件和目标文件的列表。除非使用 `-o` 选项，否则最终可执行代码位于 `a.out` 中。在使用 `-o` 选项的情况下，代码位于由 `-o` 选项指定的文件中。

使用 C 编译器编译和链接以下任何组合：

- C 源文件，带有 .c 后缀
- 内联模板文件，带有 .il 后缀 （仅当使用 .c 文件指定时）
- C 预处理源文件，带有 .i 后缀
- 目标代码文件，带有 .o 后缀
- 汇编程序源文件，带有 .s 后缀

  在链接之后， C 编译器将所链接的文件 （当前在可执行代码中）置于一个名称为 a.out 的文件中，或由 -o 选项指定的文件中。

- *libraries* 表示任意多个标准库或用户提供的库，其中包含函数、宏和常量定义。

参见选项 -Y*P, dir*，以更改用于查找库的缺省目录。 *dir* 是一个以冒号分隔的路径列表。 cc 的缺省库搜索顺序为：

```
/opt/SUNWspro/prod/lib

/usr/ccs/lib

/usr/lib
```

cc 使用 getopt 分析命令行选项。这些选项被视为单个字母或带一个参数的单个字母。参见 getopt(3c)。

A.2 选项汇总

在本节中，编译器选项按功能分组，以便于参考。以下各页的相应部分提供了详细信息。下表按功能对 cc 编译器选项进行了汇总。有些标志具有多种功能，因此在表中多次出现。

表 A-1 按功能分组的编译器选项

许可	选项标志
返回有关许可系统的信息。	-xlicinfo
优化和性能	**选项标志**
为可执行代码的速度选择最佳编译选项组合。	-fast
准备目标代码，以便为进行文件配置而收集数据	-p
针对 80386 处理器进行优化。	-x386
针对 80486 处理器进行优化。	-x486
允许编译器执行基于类型的别名分析和优化。	-xalias_level

表 A-1　按功能分组的编译器选项　*（续）*

优化和性能	选项标志
改善对调用标准库函数的代码的优化。	-xbuiltin
启用跨源文件的优化和内联处理。	-xcrossfile
分析循环以了解迭代间数据依赖性并执行循环重构。	-xdepend
允许由链接程序对数据和函数进行重新排序。	-xF
启用编译器对基于硬件计数器的文件配置的支持。	-xhwcprof
尝试仅内联指定的函数。	-xinline
通过调用过程间调用分析组件，执行整个程序优化。	-xipo
设置编译器创建的进程数。	-xjobs
内联某些库例程以提高执行速度。	-xlibmil
在 Sun 提供的性能库中进行链接。	-xlic_lib=sunperf
在可重定位目标文件中执行链接时优化。	-xlinkopt
该命令将 pragma opt 的级别限制为指定的级别。	-xmaxopt
不内联数学库例程。	-xnolibmil
优化目标代码。	-x0
设置栈和堆的首选页面大小。	-xpagesize
设置栈的首选页面大小。	-xpagesize_stack
设置堆的首选页面大小。	-xpagesize_heap
减少其源文件共享同一组包含文件的应用程序的编译时间。	-xpch
可与 -xpch 配合使用，以指定活动前缀的最后一个包含文件。	-xpchstop
针对 Pentium™ 处理器进行优化。	-xpentium
启用预取指令。	-xprefetch
控制 -xprefetch=auto 设置的预取指令自动插入的主动性。	-xprefetch_level
为配置文件收集数据或使用配置文件进行优化。	-xprofile
通过重新使用在 -xprofile=collect 阶段保存的数据来减少 -xprofile=use 阶段的编译时间。	-xprofile_ircache
支持单个配置文件目录中的多个程序或共享库。	-xprofile_pathmap
将指针值函数参数视为限定指针。	-xrestrict
允许编译器假定不会出现基于内存的陷阱。	-xsafe

表 **A-1** 按功能分组的编译器选项 *（续）*

优化和性能	选项标志
不优化或并行化增加代码大小的循环。	`-xspace`
建议优化器解开循环 *n* 次。	`-xunroll`
数据对齐	**选项标志**
通过按照指定字节顺序排列多字符字符常量的字符来生成整数常量。	`-xchar_byte_order`
分析循环以了解迭代间数据依赖性并执行循环重构。	`-xdepend`
指定最大假定内存对齐和未对齐数据访问的行为。	`-xmemalign`
支持 OpenMP 显式并行化接口，包括一组源代码指令、运行时库例程和环境变量。	`-xopenmp`
数值和浮点	**选项标志**
导致对浮点运算硬件的非标准初始化。	`-fnonstd`
打开 SPARC 非标准浮点模式。	`-fns`
初始化浮点控制字中的舍入精度模式位。	`-fprecision`
设置在程序初始化运行时建立的 IEEE 754 舍入模式。	`-fround`
允许优化器产生关于浮点运算的简化假定。	`-fsimple`
使编译器按单精度而非双精度对 `float` 表达式进行求值。	`-fsingle`
使编译器将浮点表达式或函数的值转换为赋值左侧的类型。	`-fstore`
设置在启动时有效的 IEEE 754 捕获模式。	`-ftrap`
不将浮点表达式或函数的值转换为赋值左侧的类型	`-nofstore`
分析循环以了解迭代间数据依赖性并执行循环重构。	`-xdepend`
强制在异常情况下数学例程具有 IEEE 754 式样返回值。	`-xlibmieee`
支持 OpenMP 显式并行化接口，包括一组源代码指令、运行时库例程和环境变量。	`-xopenmp`
将无后缀的浮点常量表示为单精度。	`-xsfpconst`
启用向量库函数调用的自动生成。	`-xvector`
并行化	**选项标志**
用于扩展为 `-D_REENTRANT -lthread` 的宏选项。	`-mt`
为多个处理器启用自动并行化。	`-xautopar`
对栈溢出增加一个运行时检查。	`-xcheck`
分析循环以了解迭代间数据依赖性并执行循环重构。	`-xdepend`

表 A-1　按功能分组的编译器选项　*（续）*

并行化	选项标志
生成基于 #pragma MP 指令规范的并行化代码。	-xexplicitpar
显示已并行化的循环以及尚未并行化的循环。	-xloopinfo
支持 OpenMP 显式并行化接口，包括一组源代码指令、运行时库例程和环境变量。	-xopenmp
执行编译器自动进行的循环并行化和程序员显式指定的循环并行化。	-xparallel
启用自动并行化期间的约简识别。	-xreduction
将指针值函数参数视为限定指针。	-xrestrict
对指定了 #pragma MP 指令但可能无法正确指定并行化的循环发出警告。	-xvpara
	-xthreadvar
为 lock_lint 创建程序数据库，但不生成可执行代码。	-Zll
源代码	**选项标志**
将 *name* 作为一个谓词与指定的 *tokens* 相关联，这与使用 #assert 预处理指令类似。	-A
防止预处理程序删除在预处理指令行之外的注释。	-C
将 *name* 与指定的 *tokens* 相关联，这与使用 #define 预处理指令类似。	-D
仅通过预处理程序运行源文件并将输出发送到 stdout。	-E
报告 K&R 风格定义和声明。	-fd
将在当前编译过程中包含的每个文件的路径名打印到标准错误输出，每行打印一个。	-H
将目录增加到用来查找带有相对文件名的 #include 文件的列表。	-I
仅通过 C 预处理程序运行源文件。	-P
删除预处理程序符号 *name* 的所有初始定义。	-U
-X 选项指定符合 ISO C 标准的不同级别。	-X
接受 C++ 风格注释。	-xCC
控制编译器对所支持的 C99 功能的识别。	-xc99
帮助从字符被定义为无符号类型的系统中迁移。	-xchar
允许 C 编译器接受在不符合 ISO C 源字符代码要求的语言环境中编写的源代码。	-xcsi

表 **A-1** 按功能分组的编译器选项 *（续）*

源代码	选项标志
对指定的 C 程序仅运行预处理程序，请求生成 makefile 依赖性并将结果发送到标准输出。	`-xM`
收集诸如 `-xM` 的依赖性，但排除 `/usr/include` 文件。	`-xM1`
打印在此模块中定义的所有 K&R C 函数的原型。	`-xP`
准备目标代码，以便为使用 *gprof*(1) 进行文件配置而收集数据。	`-xpg`
为源代码浏览器生成附加符号表信息。	`-xsb`
为源代码浏览器创建数据库。	`-xsbfast`
确定三字母序列的识别。	`-xtrigraphs`
启用由 16 位字符组成的字符串文字的识别。	`-xustr`
编译的代码	**选项标志**
指示编译器禁止链接 *ld*(1) 并为每个源文件生成一个 `.o` 文件。	`-c`
为输出文件命名。	`-o`
指示编译器生成汇编源文件，但不汇编程序。	`-S`
编译模式	**选项标志**
打开冗余模式，它可显示命令选项的扩展过程和调用的每个组件。	`-#`
显示每个可能调用但实际上并不执行的组件。还显示命令选项扩展的过程。	`-###`
保留在编译期间创建的临时文件而非将它们自动删除。	`-keeptmp`
指示 `cc` 打印编译器所执行的每个组件的名称和版本标识。	`-V`
将参数传递到 C 编译系统组件。	`-W`
保留字符的符号。	`-xchar`
显示联机帮助信息。	`-xhelp`
设置编译器创建的进程数。	`-xjobs`
减少其源文件共享同一组包含文件的应用程序的编译时间。	`-xpch`
可与 `-xpch` 配合使用，以指定活动前缀的最后一个包含文件。	`-xpchstop`
将 `cc` 使用的临时文件的目录设置为 *dir*。	`-xtemp`
报告每个编译组件使用的时间和资源。	`-xtime`
为 C 编译系统组件的位置指定一个新目录。	`-Y`
更改查找组件的缺省目录。	`-YA`

表 **A-1** 按功能分组的编译器选项 *（续）*

编译模式	选项标志
更改查找包含文件的缺省目录。	`-YI`
更改查找库文件的缺省目录。	`-YP`
更改启动目标文件的缺省目录。	`-YS`
诊断	**选项标志**
在错误消息前面加上 “error:” 字符串作为前缀，以便与警告消息区别。	`-errfmt`
禁止编译器警告消息。	`-erroff`
控制编译器在发现类型不匹配时所产生错误消息的详细程度。	`-errshort`
显示每条警告消息的消息标记。	`-errtags`
如果发出指示的警告消息， cc 将以故障状态退出。	`-errwarn`
指示编译器执行更严格的语义检查并启用其它类 `lint` 检查。	`-v`
禁止编译器警告消息。	`-w`
对源文件仅执行语法和语义检查，但不产生任何对象或可执行代码。	`-xe`
发出关于 K&R C 和 Sun ISO C 之间存在差异的警告。	`-xtransition`
对指定了 `#pragma MP` 指令但可能无法正确指定并行化的循环发出警告。	`-xvpara`
调试	**选项标志**
对栈溢出增加一个运行时检查。	`-xcheck`
为调试器生成附加符号表信息。	`-g`
从输出目标文件中删除所有符号调试信息。	`-s`
生成 dwarf 格式而非 stabs 格式的调试信息	`-xdebugformat`
设置栈和堆的首选页面大小。	`-xpagesize`
设置栈的首选页面大小。	`-xpagesize_stack`
设置堆的首选页面大小。	`-xpagesize_heap`
禁止 `dbx` 的目标文件自动读取。	`-xs`
允许编译器识别 VIS[tm] 指令集中定义的汇编语言模板。	`-xvis`
链接和库	**选项标志**
指定用于链接的库绑定是 `static` 还是 `dynamic`。	`-B`
指定链接编辑器中的链接类型是动态还是静态。	`-d`

表 **A-1** 按功能分组的编译器选项 *（续）*

链接和库	选项标志
将选项传递给链接编辑器，以产生共享对象而非动态链接的可执行程序。	`-G`
作为一种获得库的不同版本的方法，为共享动态库指定名称。	`-h`
将选项传递给链接程序，以忽略任何 `LD_LIBRARY_PATH` 设置。	`-i`
将目录增加到链接程序查找库的列表。	`-L`
与对象库 `lib`*name*`.so` 或 `lib`*name*`.a` 链接。	`-l`
从目标文件的 `.comment` 部分删除重复的字符串。	`-mc`
从 `.comment` 部分删除所有字符串。也可以在目标文件的这一部分插入一个 *string*。	`-mr`
对输出文件发出或不发出识别信息。	`-Q`
将一个以冒号分隔的、用于指定库搜索目录的目录列表传递给运行时链接程序。	`-R`
将数据段合并到文本段中。	`-xMerge`
指定代码地址空间。	`-xcode`
在文本段 （而非缺省数据段）的只读数据部分插入字符串文字。	`-xstrconst`
关闭递增链接程序并强制使用 `ld`。	`-xildoff`
打开递增链接程序并在递增模式下强制使用 `ild`。	`-xildon`
控制变量和函数定义的缺省作用域，以创建更快、更安全的共享库。	`-xldscope`
在目标文件中和后续共享库中包含接口信息，以使共享库可与使用 Java[tm] 编程语言编写的代码连接。	`-xnativeconnect`
缺省情况下，不链接任何库。	`-xnolib`
不内联数学库例程。	`-xnolibmil`
目标平台	**选项标志**
指定指令集体系结构。	`-xarch`
定义优化器使用的缓存属性。	`-xcache`
指定 `-xarch`、`-xchip` 和 `-xcache` 的值。	`-xcg`
指定优化器使用的目标处理器。	`-xchip`
为生成的代码指定寄存器的使用。	`-xregs`
为指令集和优化指定目标系统。	`-xtarget`

A.3 cc 选项

本节按字母顺序描述 cc 选项。手册页 cc(1) 也提供了这些描述。使用 cc -flags 选项可得到这些描述的单行摘要。

注明特定于一个或多个平台的选项可被接受且不会出现错误，并在其它所有平台上被忽略。有关使用这些选项和参数的印刷表示法的说明，请参见第 xxxii 页上的 “印刷惯例”。

A.3.1 -#

打开冗余模式，显示命令选项扩展的过程。显示调用的每个组件。

A.3.2 -###

显示每个可能调用但实际上并不执行的组件。还显示命令选项扩展的过程。

A.3.3 -A*name*[(*tokens*)]

将 *name* 作为一个谓词与指定的 *tokens* 相关联，这与使用 #assert 预处理指令类似。预断言：

- system(unix)
- machine(sparc) *(SPARC)*
- machine(i386) *(Intel)*
- cpu(sparc) *(SPARC)*
- cpu(i386) *(Intel)*

这些预断言在 -Xc 模式下无效。

A.3.4 -B[static|dynamic]

指定用于链接的库绑定是 static 还是 dynamic，并分别指明库是非共享的还是共享的。

如果给定 -l*x* 选项，则 -Bdynamic 使链接编辑器查找名称为 lib*x*.so 的文件，然后查找名称为 lib*x*.a 的文件。

-Bstatic 可使链接编辑器仅查找名称为 lib*x*.a 的文件。此选项可作为一个切换开关在命令行中多次指定。此选项及其参数将传递给 ld(1)。

注 – 很多系统库 （如 libc）在 Solaris 64 位编译环境中只能用作动态库。因此，请勿将 -Bstatic 用作命令行的最后切换开关。

A.3.5 -C

防止 C 预处理程序删除预处理指令行之外的注释。

A.3.6 -c

指示 cc 禁止链接 ld(1) 并为每个源文件生成一个 .o 文件。您可使用 -o 选项显式指定单一目标文件。当编译器为每个 .i 或 .c 输入文件生成目标代码时，它始终会在当前工作目录中创建一个目标 (.o) 文件。如果禁止链接步骤，将会同时禁止删除目标文件。

A.3.7 -D*name*[=*tokens*]

将 *name* 与指定的 *tokens* 相关联，这与使用 #define 预处理指令类似。如果未指定 =*tokens*，则提供标记 1。

预定义 （在 -Xc 模式下无效）：

- sun
- unix
- sparc *(SPARC)*
- i386 *(Intel)*

以下预定义在所有模式下均有效。

- __sparcv9 (-xarch=v9, v9a, v9b)
- __sun
- __unix
- __SUNPRO_C=0x550
- __`uname -s`_`uname -r` （示例：__SunOS_5_7）
- __sparc *(SPARC)*
- __i386 *(Intel)*
- __BUILTIN_VA_ARG_INCR
- __SVR4

以下预定义仅能在 -Xa 和 -Xt 模式下预定义：

- __RESTRICT

编译器还预定义类对象宏 __PRAGMA_REDEFINE_EXTNAME，表示将识别 pragma。

A.3.8 -d[y|n]

-dy 指定链接编辑器中的动态链接，它是缺省选项。

-dn 指定链接编辑器中的静态链接。

此选项及其参数将传递给 ld(1)。

注 – 很多系统库在 Solaris 64 位编译环境中只能作为动态库使用。因此，如果将该选项与 -xarch=v9 结合使用，将导致致命错误。

A.3.9 -dalign

-dalign 等价于 -xmemalign=8s。参见第 A-56 页上的第 A.3.102 节“-xmemalign=ab”。

A.3.10 -E

仅通过预处理程序运行源文件并将输出发送到 stdout。预处理程序被直接内建到编译器中，除了处于 -Xs 模式之外，因为在该模式中要调用 /usr/ccs/lib/cpp。包含预处理程序行号信息。参见 -P 选项。

A.3.11 -errfmt[=[no%]error]

如果要将字符串 “error:”作为前缀放在错误消息开头，以便与警告消息区别，可使用此选项。此前缀也可附加到通过 -errwarn 转换成错误的警告。

表 A-2 -errfmt 值

值	含义
error	在所有错误消息前面增加前缀 “error:”。
no%error	不在任何错误消息前面增加前缀 “error:”。

如果不使用此选项，则编译器将其设置为 -errfmt=no%error。如果您指定 -errfmt，但不提供值，则编译器将其设置为 -errfmt=error。

A.3.12 -erroff[=*t*]

此命令禁止 C 编译器警告消息，但对错误消息无任何影响。

t 是一个以逗号分隔的列表，该列表包括以下各项中的一个或多个：*tag*、 no%*tag*、 %all、 %none。以上各项的先后顺序至关重要；例如， %all,no%*tag* 禁止除 *tag* 之外的所有警告消息。下表列出了 -erroff 值：

表 A-3 -erroff 值

值	含义
tag	禁止此 *tag* 指定的警告消息。您可以通过使用 -errtags=yes 选项显示消息的标记。
no%*tag*	启用此 *tag* 指定的警告消息
%all	禁止所有警告消息
%none	启用所有警告消息 （缺省值）

缺省值为 -erroff=%none。指定 -erroff 相当于指定 -erroff=%all。

-erroff 选项只能禁止来自 C 编译器前端并在使用 -errtags 选项时显示标记的警告消息。您可以更好地控制错误消息禁止。参见第 2-12 页上的第 2.8.5 节 “error_messages”。

A.3.13 -errshort[=*i*]

使用此选项可以控制编译器在发现类型不匹配时所产生的错误消息的详细程度。当编译器发现涉及到大聚集的类型不匹配时，此选项特别有用。

i 可以是以下各项之一：

表 A-4 -errshort 值

值	含义
short	以短形式打印错误消息，且不会出现类型扩展。不扩展聚集成员、函数参数和返回类型。
full	以完全冗余形式打印错误消息，并显示不匹配类型的完全扩展。
tags	对于具有标记名称的类型，打印错误消息的标记名称。无标记名称的类型将以扩展形式显示。

如果您不使用 -errshort，则编译器将该选项设置为 -errshort=full。如果您指定 -errshort，但不提供值，则编译器将该选项设置为 -errshort=tags。

此选项不会累积，它接受在命令行指定的最后一个值。

A.3.14 -errtags[=*a*]

显示来自 C 编译器前端的每则警告消息的消息标记，使用 -erroff 选项可禁止这些警告消息，而使用 -errwarn 选项则会产生致命错误。来自 C 编译器驱动程序以及 C 编译系统其它组件的消息不带错误标记，使用 -erroff 选项并不能禁止这些消息，而使用 -errwarn 选项也不会产生致命错误。

a 可以是 yes 或 no。缺省值为 -errtags=no。指定 -errtags 相当于指定 -errtags=yes。

A.3.15 -errwarn[=*t*]

对于给定的警告消息，使用 -errwarn 可使 C 编译器以故障状态退出。

t 是一个以逗号分隔的列表，该列表包括以下各项中的一个或多个：*tag*、 no%*tag*、 %all、 %none。以上各项的先后顺序至关重要；例如，如果发出除 *tag* 之外的任何警告， %all,no%*tag* 将会导致 cc 以致命状态退出。

由于编译器错误检查的改善和功能的增加， C 编译器生成的警告消息也会因发行版本而异。使用 -errwarn=%all 编译而无错误的代码在使用编译器的下一个发行版本进行编译时，可能无法编译而出现错误。

只有来自 C 编译器前端并在使用 -errtags 选项时会显示标记的警告消息可以使用 -errwarn 选项进行指定，以使 C 编译器以故障状态退出。

下表详细列出了 -errwarn 值：

表 A-5 -errwarn 值

值	含义
tag	如果 *tag* 指定的消息作为警告消息发出，则导致 cc 以致命状态退出。如果未发出 *tag*，则没有影响。
no%*tag*	如果 *tag* 指定的消息仅作为警告消息发出，则防止 cc 以致命状态退出。如果未发出 *tag* 指定的消息，则没有影响。使用此选项进行回复，以使先前此选项及 *tag* 或 %all 指定的警告消息作为警告消息发出时，不会导致 cc 以致命状态退出。
%all	如果发出任何警告消息，则导致 cc 以致命状态退出。%all 后面可带有 no%*tag*，以免除特定的警告消息出现此行为。
%none	如果发出任何警告消息，则防止警告消息导致 cc 以致命状态退出。

缺省值为 -errwarn=%none。如果仅指定 -errwarn，则等价于 -errwarn=%all。

A.3.16 -fast

为基准测试应用程序的优化选择一组基准线选项。这些优化可能会改变程序中由 ISO C 和 IEEE 标准定义的程序的行为。用 -fast 编译的模块必须也用 -fast 进行链接。

-fast 是一个宏选项，可有效用作调节可执行程序以达到最高运行时性能的起点。-fast 是一个宏，可随编译器发行版本的升级而更改，并扩展为目标平台特定的选项。我们建议您使用 -# 选项检查 -fast 的扩展，并将 -fast 的相应选项合并到现行的可执行程序调节进程中。

-fast 选项不适于原本在编译机器之外的其它目标机器上运行的程序。在这种情况下，请在 -fast 后附带相应的 -xtarget 选项。例如：

```
cc -fast -xtarget=ultra ...
```

对于依赖于 SUID 指定的异常处理的 C 模块，请在 -fast 后附带 -xnolibmil：

```
% cc -fast -xnolibmil
```

使用 -xlibmil 后，就不能通过设置 errno 或调用 matherr(3m) 来注释异常。

-fast 选项不适于要求与 IEEE 754 标准严格一致的程序。

下表列出了 -fast 交叉平台选择的选项集。

表 A-6 -fast 扩展值

选项	SPARC	x86
-fns	X	X
-fsimple=2	X	X
-fsingle	X	X
-ftrap=%none	X	X
-nofstore	-	X
-xalias_level=basic	X	-
-xarch	X	X
-xbuiltin=%all	X	X
-xdepend	X	X
-xlibmil	X	X
-xmemalign=8s	X	-
-xO5	X	X
-xprefetch=auto,explicit	X	-

注 – 有些优化对程序行为进行特定假定。如果程序不符合这些假定，则应用程序将会崩溃或产生错误结果。请参见单个选项的描述，以确定您的程序是否适合用 -fast 编译。

由这些选项执行的优化可能会改变由 ISO C 和 IEEE 标准定义的程序的行为。有关详细信息，请参见特定选项的描述。

-fast 的作用相当于命令行上的一个宏扩展。因此，您可以通过在 -fast 后附带预期的优化级别或代码生成选项来覆盖优化级别和代码生成选项。编译时使用 -fast -xO4 对类似于使用 -xO2 -xO4 对。后者优先。

在以前的发行版本中， -fast 宏选项包含 -fnonstd ；而现在它包含 -fns。

-fast 还定义宏 __MATHERR_ERRNO_DONTCARE。该宏导致 math.h 对一些原型为 <math.h> 的某些数学例程断言性能相关的 pragma，如：

- #pragma does_not_read_global_data
- #pragma does_not_write_global_data
- #pragma no_side_effect

在 matherr(3M) 手册页中说明的异常情况下，如果您的代码依赖于 errno 的返回值，则您必须通过在 -fast 选项之后发出 -U__MATHERR_ERRNO_DONTCARE 宏来关闭宏。

通常情况下，您可以使用此选项提高大多数程序的性能。

请勿对依赖于 IEEE 标准异常处理的程序使用此选项；否则您将获得不同的数值结果、程序提前终止或未预料的 SIGFPE 信号。

参见第 A-9 页上的 “-#” 和第 A-9 页上的 “-###”，以获取如何查看宏扩展选项的详细信息。

A.3.17 -fd

报告 K&R 风格函数定义和声明。

A.3.18 -flags

打印关于每个可用编译器选项的简短摘要。

A.3.19 -fnonstd

导致对浮点运算硬件的非标准初始化。此外， -fnonstd 选项导致为浮点溢出、除数为零和无效操作异常启用硬件陷阱。这些将被转换成 SIGFPE 信号；如果程序没有 SIGFPE 处理程序，该程序将以内存转储终止。

缺省情况下， IEEE 754 浮点运算不间断，并且下溢是渐进的。 （参见第 2-4 页上的第 2.4 节 “浮点，非标准模式”以了解更多的说明。）

(SPARC) 等同于 -fns -ftrap=common。

A.3.20 -fns[={no,yes}]

(SPARC) 打开 SPARC 非标准浮点模式。

缺省值为 -fns=no，表示 SPARC 标准浮点模式。 -fns 与 -fns=yes 相同。

使用 =yes 或 =no （可选）提供了一种切换包含 -fns 的某些其它宏标志 （如 -fast）后面的 -fns 标志的方法。当程序开始执行时，此标志启用非标准浮点模式。缺省情况下，将不会自动启用非标准浮点模式。

在某些 SPARC 系统上，非标准浮点模式禁用 “渐进下溢”，从而使微结果归零而不是产生次正规数。它还导致次正规操作数被替换为零而无提示。在这些不通过硬件支持渐进下溢和次正规数的 SPARC 系统上，使用此选项可以显著提高某些程序的性能。

在启用了非标准模式的情况下，浮点运算可能会产生不符合 IEEE 754 标准要求的结果。有关详细信息，请参见 《数值计算指南》。

此选项仅对 SPARC 系统且仅在编译主程序时使用才有效。在 x86 系统上，忽略此选项。

A.3.21 -fprecision=*p*

(*x86*) `-fprecision={single, double, extended}`

将浮点控制字中的舍入精度模式位分别初始化为单 （24 位）、双 （53 位）或扩展 （64 位）。缺省浮点舍入精度模式为扩展。

请注意，在 Intel 上，浮点舍入精度模式的设置仅对精度范围有影响，而对指数范围无影响。

A.3.22 -fround=*r*

设置在程序初始化运行时建立的 IEEE 754 舍入模式。

r 必须为以下之一：`nearest`、 `tozero`、 `negative`、 `positive`。

缺省值为 `-fround=nearest`。

含义与 `ieee_flags` 子例程的含义相同。

如果 *r* 为 `tozero`、 `negative` 或 `positive`，则此标志在程序开始执行时将舍入方向模式分别设置为：舍入为零、舍入为负无穷或舍入为正无穷。如果 *r* 为 `nearest` 或使用了 `-fround` 标志，则舍入方向模式仍为其初始值 （缺省情况下，舍入为最近值）。

此选项只有在编译主程序时使用才有效。

A.3.23 -fsimple[=*n*]

允许优化器产生关于浮点运算的简化假定。

如果 *n* 存在，则它必须是 0、 1 或 2。缺省值为：

- 如果没有 `-fsimple[=`*n*`]`，则编译器使用 `-fsimple=0`
- 如果只有 `-fsimple` 而无 `=`*n*，则编译器使用 `-fsimple=1`

`-fsimple=0`

允许无简化假定。保持严格的 IEEE 754 一致性。

`-fsimple=1`

允许保守简化。最后所得到的代码并不与 IEEE 754 严格一致，但并不更改大多数程序的数值结果。

如果 `-fsimple=1`，则优化器可以假定以下内容：

- 在进程初始化之后， IEEE 754 缺省舍入/捕获模式不会发生更改。
- 除了产生潜在浮点异常之外而不会产生可见结果的计算可能会被删除。
- 使用无穷大或 NaN 作为操作数的计算不必将 NaN 传送给其结果，例如， `x*0` 可能会被替换为 `0`。
- 计算不依赖于零的符号。

如果 `-fsimple=1`，则*禁止*优化器进行完全优化而不考虑舍入或异常。特别是，浮点计算不能替换为一个在运行时在舍入模式保持不变的情况下产生不同结果的计算。`-fast` 宏标志包含 `-fsimple=1`。

`-fsimple=2`

允许可能导致很多程序因舍入更改而产生不同数值结果的主动浮点优化。例如，`-fsimple=2` 允许优化器将给定循环中 `x/y` 的所有计算替换为 `x*z` ，其中保证在循环中至少对 `x/y` 进行一次求值， `z=1/y`，并且已知 `y` 和 `z` 的值在循环执行期间具有常量值。

即使 `-fsimple=2`，也不允许优化器在否则不产生任何内容的程序中引入浮点异常。

A.3.24 -fsingle

（仅适用于 `-Xt` 和 `-Xs` 模式）使编译器以单精度而非双精度对 `float` 表达式进行求值。由于已按单精度对 `float` 表达式进行求值，因此如果在 `-Xa` 或 `-Xc` 模式下使用编译器，则此选项无效。

A.3.25 -fstore

(Intel) 当表达式或函数被赋值给变量或表达式被强制转换为较短浮点类型时，使编译器将浮点表达式或函数的值转换为赋值左侧的类型，而不是将该值保留在寄存器中。由于舍入和截尾，结果可能会与使用寄存器值生成的结果不同。这是缺省模式。

要关闭此选项，请使用 `-nofstore` 选项。

A.3.26 -ftrap=*t*

在启动时，设置有效的 IEEE 754 捕获模式。

t 是一个以逗号分隔的列表，该列表包括以下各项中的一个或多个：%all、 %none、common、[no%]invalid、[no%]overflow、[no%]underflow、[no%]division、[no%]inexact。

缺省值为 -ftrap=%none。

此选项设置在程序初始化时建立的 IEEE 754 捕获模式。处理顺序是从左至右。common 异常有 （按定义）：无效、除数为零和溢出。

示例：-ftrap=%all,no%inexact 表示设置除 inexact 之外的所有陷阱。

含义与 ieee_flags 子例程的含义相同，只是有以下不同：

- %all 打开所有捕获模式。
- %none（缺省值）关闭所有捕获模式。
- no% 前缀关闭特定的捕获模式。

如果您使用 -ftrap=*t* 编译一个例程，则使用相同的 -ftrap=*t* 选项编译程序的所有例程；否则，您会得到未预料的结果。

A.3.27 -G

将选项传递给链接编辑器，以产生共享对象而非动态链接的可执行程序。此选项将传递给 ld(1)，且不能与 -dn 选项一起使用。

A.3.28 -g

生成附加符号表信息，以使用 dbx(1) 和性能分析器 analyzer(1) 进行调试。

在编译器的仅链接调用中， -g 选项使递增链接程序 (ild) 而不是链接程序 (ld) 成为缺省值。如果使用 -g，则编译器的缺省行为是自动调用 ild，而不是调用 ld，除非在命令行上指定 -G 或任何源文件。使用 -xildoff 可禁用 ild。有关详细信息，请参见 ild(1) 手册页。参见第 A-47 页上的第 A.3.86 节 “-xildoff”和第 A-47 页上的第 A.3.87 节 “-xildon”。

如果指定 -g，并且优化级别为 -xO3 或更低，则编译器提供最佳效果符号信息以及几乎完全优化。禁用尾部调用优化和后端内联。

如果指定 -g，并且优化级别为 -xO4，则编译器提供最佳效果符号信息以及完全优化。

使用 -g 选项进行编译，以使用性能分析器的全部性能。虽然某些性能分析功能不需要 -g，但是您必须使用 -g 进行编译，以查看注释的源代码、部分函数级别信息以及编译器注释消息。有关详细信息，请参见 analyzer(1) 手册页和 《*程序性能分析工具*》中的 “编译您的程序以进行数据收集和分析”。

使用 -g 生成的注释消息描述编译器在编译程序时进行的优化和变换。使用 er_src(1) 命令显示与源代码交错的消息。

有关调试的详细信息，请参见 《*使用* dbx *调试程序*》手册。

A.3.29 -H

将当前编译过程中包含的每个文件的路径名打印到标准错误输出，每行打印一个。这样打印的目的是显示包含在其它文件中的文件。

此处，程序 sample.c 包含 stdio.h 和 math.h 文件； math.h 包含 floatingpoint.h 文件，该文件本身包含使用 sys/ieeefp.h 的函数：

```
% cc -H sample.c
       /usr/include/stdio.h
    /usr/include/math.h
        /usr/include/floatingpoint.h
            /usr/include/sys/ieeefp.h
```

A.3.30 -h *name*

作为一种获得库的不同版本的方法，为共享动态库指定名称。通常， -h 后面的 *name* 应与 -o 选项后指定的文件名相同。 -h 和 *name* 之间的空格是可选的。

链接程序将指定的 *name* 指定给库，并在库文件中将此名称记录为库的 *intrinsic* 名称。如果 -h*name* 选项不存在，则库文件中不记录任何内名称。

当运行时链接程序将库装入可执行文件时，它将内名称从库文件复制到可执行文件中的所需共享库文件列表中。每个可执行文件都有这样的列表。如果共享库的内名称不存在，则链接程序复制共享库文件的路径。

A.3.31 -I[- | *dir*]

-I *dir* 将 *dir* 增加到使用相对文件名 （即不以 / （斜杠）开头的文件名）查找 #include 文件时搜索的目录列表。-I 值将累积。有关对用于查找包含文件的搜索顺序的讨论，请参见第 2-24 页上的第 2.13 节 “如何指定包含文件”。

-I- 使您可以更好地控制编译器在查找包含文件时使用的算法。-I- 值不累积。本节先后描述了缺省搜索算法和 -I- 对这些算法的影响。

有关编译器的搜索模式的详细信息，请参见第 2-24 页上的第 2.13 节 “如何指定包含文件”。

A.3.32 -i

将该选项传递给链接程序，以忽略任何 LD_LIBRARY_PATH 或 LD_LIBRARY_PATH_64 设置。

A.3.33 -KPIC

(SPARC) -KPIC 命令等价于 -xcode=pic32。参见第 A-41 页上的第 A.3.75 节 “-xcode[=v]”。

(Intel) -KPIC 与 -Kpic 相同。

A.3.34 -Kpic

(SPARC) -Kpic 命令等价于 -xcode=pic13。参见第 A-41 页上的第 A.3.75 节 “-xcode[=v]”。

(Intel) 生成共享库 （小模型）中使用的与位置无关的代码。允许最多引用 2**11 个唯一外部符号。

A.3.35 -keeptmp

保留在编译期间创建的临时文件而非自动删除它们。

A.3.36 -L*dir*

将 *dir* 增加到 ld(1) 查找库时搜索的目录的列表。此选项及其参数将传递给 ld(1)。

A.3.37 -l*name*

链接对象库 lib*name*.so 或 lib*name*.a。由于符号是按从左至右的顺序解析的，因此命令行上库的顺序至关重要。

此选项必须在 *sourcefile* 参数之后。

A.3.38 -mc

从目标文件的 .comment 部分删除重复的字符串。当您使用 -mc 标志时，调用 mcs -c。

A.3.39 -misalign

(SPARC) -misalign 等价于 -xmemalign=1i。参见第 A-56 页上的第 A.3.102 节“-xmemalign=ab”。

A.3.40 -misalign2

(SPARC) -misalign2 等价于 -xmemalign=2i。参见第 A-56 页上的第 A.3.102 节“-xmemalign=ab”。

A.3.41 -mr[,*string*]

-mr 从 .comment 部分删除所有字符串。当您使用此标志时，调用 mcs -d -a。

-mr,*string* 删除目标文件 .comment 部分的所有字符串，并在此部分插入 *string*。如果 *string* 包含嵌入空白，则必须将其括入引号。空 *string* 将导致 .comment 部分为空。此选项将作为 -d -*astring* 传递给 mcs。

A.3.42 -mt

用于扩展为 -D_REENTRANT -lthread 的宏选项。如果要编译自己的多线程编码，您必须在编译和链接步骤中使用此选项。要获得更快的执行速度，该选项需要多处理器系统。在单处理器系统中，使用此选项时产生的可执行程序通常运行更慢。

A.3.43 -native

此选项等同于 -xtarget=native。

A.3.44 -nofstore

(Intel) 当表达式或函数被赋值给变量或被强制转换为较短浮点类型时，不将浮点表达式或函数的值转换为赋值左侧的类型；而是将值保留在寄存器中。参见第 A-18 页上的第 A.3.25 节 “-fstore”。

A.3.45 -O

与 -xO2 相同。

A.3.46 -o *filename*

给输出文件 *filename* 命名 （与缺省值 a.out 相反）。*filename* 不能与 *sourcefile* 相同，因为 cc 不覆写源文件。此选项及其参数将传递给 ld(1)。

A.3.47 -P

仅通过 C 预处理程序运行源文件。然后将输出放在带有 .i 后缀的文件中。与 -E 不同，此选项不在输出中包含预处理程序类型行号信息。参见 -E 选项。

A.3.48 -p

准备目标代码，以便为使用 gprof(1) 进行文件配置而收集数据。此选项调用在正常终止情况下产生 mon.out 文件的运行时记录机制。

A.3.49 -Q[y|n]

对输出文件发出或不发出标识信息。-Qy 是缺省值。

如果使用 -Qy，则将关于每个调用的编译工具的标识信息增加到输出文件的 .comment 部分，使用 mcs 可访问这部分文件。此选项对软件管理十分有用。

-Qn 禁止此信息。

A.3.50 -qp

与 -p 相同。

A.3.51 -R*dir*[:*dir*]

将一个以冒号分隔的、用于指定库搜索目录的目录列表传递给运行时链接程序。如果此选项存在且非空，则它记录在输出目标文件中并传递给运行时链接程序。

如果同时指定了 LD_RUN_PATH 和 -R 选项，则 -R 选项优先。

A.3.52 -S

指示 cc 生成汇编源文件但不汇编程序。

A.3.53 -s

从输出目标文件中删除所有符号调试信息。不能使用 -g 指定此选项。

传递给 ld(1)。

A.3.54 -U*name*

取消定义预处理程序符号 *name*。此选项可删除由 -D 在命令行上创建的预处理程序符号 *name* 的任何初始定义，包括那些由命令行驱动程序放置的定义。

-U 对源文件中的预处理程序指令无影响。您可以在命令行指定多个 -U 选项。

如果为命令行上的 -D 和 -U 指定了相同的 *name*，则会取消定义 *name*，而无论这些选项出现的顺序如何。在以下示例中， -U 取消定义 __sun：

```
cc -U__sun text.c
```

由于已取消定义 __sun，因此在 test.c 中采用以下形式的预处理程序语句将不会生效。

```
#ifdef(__sun)
```

有关预定义符号的列表，请参见第 A-10 页上的第 A.3.7 节 “-Dname[=tokens]”。

A.3.55 -V

指示 cc 打印编译器所执行的每个组件的名称和版本标识。

A.3.56 -v

指示编译器执行更严格的语义检查并启用其它类 lint 检查。例如，代码：

```
#include <stdio.h>
main(void)
{
     printf("Hello World.\n");
}
```

编译和执行时不会出现问题。如果使用 -v，该代码仍可编译；但是，编译器会显示以下警告：

```
"hello.c", 第 5 行：警告： 函数无 return 语句：
main
```

-v 不能给出 lint(1) 可给出的所有警告。尝试通过 lint 运行以上示例。

A.3.57 -W*c* , *arg*

将参数 *arg* 传递给指定的组件 *c*。参见表 1-1 以获取组件的列表。

每个参数与前一个参数之间仅以逗号分隔。所有 -W 参数均在常规命令行参数之后进行传递。在逗号之前立即使用换码符 \（反斜杠）可将逗号作为参数的一部分。所有 -W 参数均在常规命令行参数之后进行传递。

例如， -Wa,-o,objfile 按顺序将 -o 和 objfile 传递给汇编程序。此外，-Wl,-I,*name* 将导致链接阶段覆盖动态链接程序的缺省名称 /usr/lib/ld.so.1。

参数传递到工具的顺序可能会因其它指定的命令行选项而更改。

c 可以是以下各项之一：

表 A-7 -W 值

a	汇编程序：(fbe); (gas)
c	C 代码生成器：(cg) *(SPARC)*;
d	cc 驱动程序[1]
h	中间代码翻译者 (ir2hf)(*Intel*)
i	过程间调用优化器 (ube_ipa)(Intel)
l	链接编辑器 (ld)
m	mcs
O	过程间调用优化器 *(SPARC)*
p	预处理程序 (cpp)
u	C 代码生成器 (ube) (*Intel*)
0	编译器 (acomp) (ssbd, *SPARC*)
2	优化器：(iropt) *(SPARC)*

1 您不能使用 -Wd 将本章列出的 cc 选项传递给 C 编译器。

A.3.58 -w

禁止编译器警告消息。

此选项覆盖 error_messages pragma。

A.3.59 -X[c|a|t|s]

-X 选项（注意 X 为大写）指定符合 ISO C 标准的不同程度。-xc99 的值影响 -X 选项所应用的 ISO C 标准的版本。-xc99 选项的缺省值为支持 1999 ISO/IEC C 标准的子集的 -xc99=%all。-xc99=%none 支持 1990 ISO/IEC C 标准。有关对 1999 ISO/IEC 支持功能的讨论，请参见附录 D。有关对 SO/IEC C 和 K&R C 差异的讨论，请参见附录 F。

缺省模式为 -Xa。

-Xc

（c = 一致性）对使用非 ISO C 构造的程序发出错误和警告。此选项与 ISO C 严格一致，没有 K&R C 兼容性扩展。如果设置 -Xc 选项，预定义的宏 _ _STDC_ _ 的为 1。

-Xa

这是缺省编译器模式。ISO C 以及 K&R C 兼容性扩展，具有 ISO C 要求的语义更改。如果 K&R C 和 ISO C 为同一构造指定不同语义，则编译器使用 ISO C 解释。如果 -Xa 选项与 -xtransition 选项配合使用，则编译器发出关于不同语义的警告。如果设置 -Xa 选项，预定义的宏 _ _STDC_ _ 的值为 0。

-Xt

（t = 转换）此选项使用 ISO C 以及 K&R C 兼容性扩展，但*没有* ISO C 要求的语义更改。如果 K&R C 和 ISO C 为同一构造指定了不同语义，则编译器使用 K&R C 解释。如果将 -Xt 选项与 -xtransition 选项配合使用，则编译器发出关于不同语义的警告。如果设置 -Xt 选项，预定义的宏 _ _STDC_ _ 的值为 0。

-Xs

（s = K&R C）试图对在 ISO C 和 K&R C 之间具有不同行为的所有语言构造发出警告。编译器语言包括与 K&R C 兼容的所有功能。此选项调用 cpp 以进行预处理。_ _STDC_ _ 在此模式下未定义。

A.3.60 -x386

(Intel) 针对 80386 处理器进行优化。

A.3.61 -x486

(Intel) 针对 80486 处理器优化。

A.3.62 -xa

此选项目前被认为已作废。而使用 -xprofile=tcov。

A.3.63 -xalias_level[=*l*]

(SPARC) 编译器使用 -xalias_level 选项确定要采用哪种假定，以使用基于类型的别名分析来执行优化。此选项使指定的别名级别对正编译的转换单元生效。

如果您不指定 -xalias_level 命令，则编译器将假定 -xalias_level=any。如果指定不带值的 -xalias_level，则缺省值为 -xalias_level=layout。

-xalias_level 选项要求的优化级别为 -xO3 或更高。如果优化级别设置较低，则发出警告并忽略 -xalias_level 选项。

切记，如果您使用 -xalias_level 选项，但无法坚持为任何别名级别描述的关于别名的所有假定和约束，则程序的行为未定义。

将 *l* 替换为下表中的某一术语。

表 A-8 别名歧义消除级别

术语	含义
any	编译器假定所有内存引用可在此级别互为别名。在级别 -xalias_level=any 上，不存在基于类型的别名分析。
basic	如果您使用 -xalias_level=basic 选项，则编译器假定涉及不同 C 基本类型的内存引用并不互为别名。此外，编译器还假定对其它所有类型的引用互为别名，并可以使用任何 C 基本类型作为别名。编译器假定使用 char * 的引用可以使用涉及任何其它类型的引用作为别名。 例如，在 -xalias_level=basic 级别上，编译器假定类型为 int * 的指针变量不会访问浮点对象。因此，编译器可安全执行优化，该优化假定类型为 float * 的指针不会使用 int * 类型指针引用的相同内存作为别名。
weak	如果您使用 -xalias_level=weak 选项，则编译器假定所有结构指针均可指向任何结构类型。 任何结构或联合类型，只要它包含对编译的源代码的表达式中引用的任何类型的引用，或者包含对从编译的源代码外部引用的任何类型的引用，就必须在编译的源代码中该表达式之前进行声明。 您可以通过包含某个程序的所有头文件来满足此约束，该程序包含的类型引用了在所编译的源代码的表达式中引用的对象的类型。 在级别 -xalias_level=weak 上，编译器假定涉及不同 C 基本类型的内存引用并不互为别名。编译器假定使用 char * 的引用使用涉及任何其它类型的内存引用作为别名。

表 A-8 别名歧义消除级别 *（续）*

术语	含义
layout	如果您使用 -xalias_level=layout 选项，则编译器假定涉及内存中类型序列相同的类型的内存引用可以互为别名。 编译器假定在内存中看起来并不相同的两个引用并不互为别名。如果初始结构成员在内存中看起来相同，则编译器假定任何两个内存均通过不同的结构类型进行访问。然而，在此级别上，您不应使用指向结构的指针来访问不同结构对象的某字段，该字段不属于这两个结构之间在内存中看起来相同的公共初始序列。这是因为编译器假定此类引用并不互为别名。 在 -xalias_level=layout 级别上，编译器假定涉及不同 C 基本类型的内存引用并不互为别名。编译器假定使用 char * 的引用可以使用涉及其它类型的内存引用作为别名。
strict	如果您使用 -xalias_level=strict 选项，则编译器假定涉及在删除标记时相同的类型（如结构或联合）的内存引用可以互为别名。反之，则编译器假定涉及那些即使在删除标记后也不相同的类型的内存引用并不互为别名。 然而，任何结构或联合类型，只要它包含对编译的源代码的表达式中引用的任何类型的引用，或者包含对从编译的源代码外部引用的任何类型的引用，就必须在编译的源代码中该表达式之前进行声明。 您可以通过包含某个程序的所有头文件来满足此约束，该程序要包含的类型引用了在所编译的源代码的表达式中引用的对象的任何类型。在 -xalias_level=strict 级别上，编译器假定涉及不同 C 基本类型的内存引用并不互为别名。编译器假定使用 char * 的引用可以使用涉及任何其它类型的引用作为别名。
std	如果您使用 -xalias_level=std 选项，则编译器假定类型和标记必须相同才能使用别名，但使用 char * 的引用可以使用涉及任何其它类型的引用作为别名。此规则与 1999 ISO C 标准中对指针非关联化的约束相同。正确使用此规则的程序将非常易于移植，而且优化之后性能大大提高。
strong	如果您使用 -xalias_level=strong 选项，则所应用的约束与在 std 级别相同，但除此之外，编译器还假定类型为 char * 的指针仅用于访问字符类型的对象。此外，编译器还假定不存在内部指针。内部指针被定义为指向结构成员的指针。

A.3.64 -xarch=*isa*

指定指令集体系结构 (ISA)。

此选项将编译器生成的代码限制为指令集体系结构指定的指令。此选项并不保证使用特定于任何目标的指令。然而，使用此选项可能影响二进制程序的可移植性。

如果您分别在单独的步骤中进行编译和链接，请确保在这两个步骤中为 -xarch 指定相同的值。

-xarch *isa* 关键字接受的体系结构如表 A-9 所示：

表 A-9 -xarch ISA 关键字

平台	有效 -xarch 关键字
SPARC	generic, native, v7, v8a, v8, v8plus, v8plusa, v8plusb, v9, v9a, v9b
x86	generic, native, 386, pentium_pro

请注意，虽然 -xarch 可单独使用，但它是 -xtarget 选项扩展的一部分并可能会用于覆盖由特定的 -xtarget 选项设置的 -xarch 值。例如：

```
% cc -xtarget=ultra2 -xarch=v8plusb ...
```

覆盖 -xtarget=ultra2 设置的 -xarch=v8。

如果您在优化时使用此选项，则适当的选择可在指定的体系结构中为可执行程序提供良好的性能。选择不当将导致二进制程序在所需的目标平台上不可执行。

A.3.64.1 仅适用于 SPARC

下表详细说明了使用指定的 -xarch 选项进行编译然后由各种 SPARC 处理器执行的可执行程序的性能。此表的目的在于帮助您识别适合您的可执行程序在特定目标机器上运行的最佳 -xarch 选项。首先识别您关心的机器的范围，然后考虑维护多个二进制程序的成本与在较新的机器上获得最高性能的利益。

表 A-10 -xarch 矩阵

		SPARC 机器指令集：					
		V7	V8a	V8	V9（非 Sun 处理器）	V9（Sun 处理器）	V9b
-xarch 编译选项	v7	N	S	S	S	S	S
	v8a	PD	N	S	S	S	S
	v8	PD	PD	N	S	S	S
	v8plus	NE	NE	NE	N	S	S
	v8plusa	NE	NE	NE	**	N	S
	v8plusb	NE	NE	NE	**	NE	N

** 注：使用此指令集编译的可执行程序可能名义上在 V9 非 Sun 处理器芯片上执行或根本就不执行。请咨询硬件供应商，以确保您的可执行程序可在它的目标机器上运行。

表 A-10 `-xarch` 矩阵 *（续）*

	SPARC 机器指令集：					
v9	NE	NE	NE	N	S	S
v9a	NE	NE	NE	**	N	S
v9b	NE	NE	NE	**	NE	N

** 注：使用此指令集编译的可执行程序可能名义上在 V9 非 Sun 处理器芯片上执行或根本就不执行。请咨询硬件供应商，以确保您的可执行程序可在它的目标机器上运行。

- N 反映标称性能。程序执行并充分利用处理器的指令集。
- S 反映满意性能。程序执行，但并不利用所有可用处理器指令。
- PD 反映性能降低。程序执行，但根据使用的指令，可能会导致性能轻微或显著降低。当内核对处理器未实现的指令进行仿真时，出现性能降低。
- NE 表示不可执行。程序因内核不对处理器未实现的指令进行仿真而不可执行。

如果您使用 `v8plus` 或 `v8plusa` 指令集编译可执行程序，请考虑使用 `v9` 或 `v9a` 编译。提供 `v8plus` 和 `v8plusa` 选项的目的是在 Solaris 7 软件及其对 64 位程序的支持出现之前，使程序可以利用 SPARC V9 和 UltraSPARC 的某些功能。使用 `v8plus` 或 `v8plusa` 选项编译的程序不可移植到 SPARC V8 或更旧的机器上。您可以分别使用 `v9` 或 `v9a` 编译此类程序，以充分利用 SPARC V9 和 UltraSPARC 的所有功能。可通过 Sun 代表获得 《*The V8+ Technical Specification*》白皮书，部件号码为 802-7447-10，此书解释了 `v8plus` 和 `v8plusa` 的限制。

- SPARC 指令集体系结构 V7、 V8 和 V8a 均为二进制兼容型。
- 使用 `v8plus` 和 `v8plusa` 编译的目标二进制文件 (.o) 可进行链接并可一起执行，但仅适用于 SPARC V8plusa 兼容平台。
- 使用 `v8plus`、`v8plusa` 和 `v8plusb` 编译的目标二进制文件 (.o) 可进行链接并可一起执行，但仅适用于 SPARC V8plusb 兼容平台。
- `xarch` 值 `v9`、 `v9a` 和 `v9b` 仅在 UltraSPARC 64 位 Solaris 操作环境中可用。
- 使用 `v9` 和 `v9a` 编译的目标二进制文件 (.o) 可进行链接并可一起执行，但仅适用于 SPARC V9a 兼容平台。
- 使用 `v9`、 `v9a` 和 `v9b` 编译的目标二进制文件 (.o) 可进行链接并可一起执行，但仅适用于 SPARC V9b 兼容平台。

无论是哪种选择，生成的可执行程序在早期体系结构上运行可能慢得多。此外，虽然四精度 （`REAL*16` 和 `long double`）浮点指令在很多指令集体系结构中可用，但是编译器并不在其生成的代码中使用这些指令。

下表提供了 SPARC 平台上每个 -xarch 关键字的详细信息。

表 A-11 SPARC 平台上的 -xarch 值

-xarch=	含义
generic	**为在大多数系统上获得良好性能而进行编译。** 这是缺省值。此选项使用最佳指令集，以便在多数处理器上获得良好性能而不会使任何处理器性能明显降低。“最佳”指令集的定义可能会根据新发行版本的不同而适当进行调整。
native	**为在此系统上获得良好性能而进行编译。** 这是 -fast 选项的缺省值。编译器为当前正运行的系统处理器选择适当的设置。
v7	**针对 SPARC-V7 ISA 进行编译。** 使编译器能够生成可在 V7 ISA 上获得良好性能的代码。 这相当于使用最佳指令集以便在 V8 ISA 上获得良好性能，但不需要整型 mul 和 div 指令以及 fsmuld 指令。 示例：SPARCstation 1、 SPARCstation 2
v8a	**针对 SPARC-V8 ISA 的 V8a 版本进行编译。** 根据定义， V8a 表示 V8 ISA，但没有 fsmuld 指令。 此选项使编译器能够生成可在 V8a ISA 上获得良好性能的代码。 示例：任何基于 microSPARC I 芯片体系结构的系统
v8	**针对 SPARC-V8 ISA 进行编译。** 此选项使编译器能够生成可在 V8 体系结构上获得良好性能的代码。 示例：SPARCstation 10
v8plus	**针对 SPARC-V9 ISA 的 V8plus 版本进行编译。** 根据定义， V8plus 表示 V9 ISA，但仅限于 V8plus ISA 规范定义的 32 位子集，并且没有可视化指令集 (VIS) 及其它特定于实现的 ISA 扩展。 • 此选项使编译器能够生成可在 V8plus ISA 上获得良好性能的代码。 • 最终的目标代码采用 SPARC-V8+ ELF32 格式且仅在 Solaris UltraSPARC 环境中执行，即不能在 V7 或 V8 处理器上运行。 示例：任何基于 UltraSPARC 芯片体系结构的系统
v8plusa	**在 SPARC-V9 ISA 的 V8plusa 版本上编译。** 根据定义， V8plusa 表示 V8plus 体系结构以及可视化指令集 (VIS) 1.0 版，并具有 UltraSPARC 扩展。 • 此选项使编译器能够生成可在 UltraSPARC 体系结构上获得良好性能的代码，但仅限于 V8plus 规范定义的 32 位子集。 • 最终的目标代码采用 SPARC-V8+ ELF32 格式且仅在 Solaris UltraSPARC 环境下执行，即不能在 V7 或 V8 处理器上运行。 示例：任何基于 UltraSPARC 芯片体系结构的系统

表 A-11 SPARC 平台上的 -xarch 值 *（续）*

-xarch=	含义
v8plusb	**针对具有 UltraSPARC III 扩展的 SPARC-V8plus ISA 的 V8plusb 版本进行编译。** 此选项允许编译器在具有 UltraSPARC III 扩展的 UltraSPARC 体系结构以及可视化指令集 (VIS) 2.0 版上生成目标代码。 • 最终的目标代码采用 SPARC-V8+ ELF32 格式且仅在 Solaris UltraSPARC III 环境中执行。 • 使用此选项进行编译将使用最佳指令集，以便在 UltraSPARC III 体系结构上获得良好性能。
v9	**针对 SPARC–V9 ISA 进行编译。** 此选项使编译器能够生成可在 V9 SPARC 体系结构上获得良好性能的代码。 • 最终的 .o 目标文件采用 ELF64 格式，而且只能与其它格式相同的 SPARC-V9 目标文件进行链接。 • 最终的可执行程序只能在运行具有 64 位内核的 64 位 Solaris 操作环境的 UltraSPARC 处理器上运行。 • -xarch=v9 仅当在 64 位 Solaris 环境中编译时才可用。
v9a	**针对具有 UltraSPARC 扩展的 SPARC–V9 ISA 进行编译。** 将 UltraSPARC 处理器的特定可视化指令集 (VIS) 和扩展增加到 SPARC-V9 ISA，并使编译器能够生成可在 V9 SPARC 体系结构上获得良好性能的代码。 • 最终的 .o 目标文件采用 ELF64 格式，而且只能与其它格式相同的 SPARC-V9 目标文件进行链接。 • 最终的可执行程序只能在运行具有 64 位内核的 64 位 Solaris 操作环境的 UltraSPARC 处理器上运行。 • -xarch=v9a 仅当在 64 位 Solaris 操作环境中编译时才可用。
v9b	**针对具有 UltraSPARC III 扩展的 SPARC–V9 ISA 进行编译。** 将 UltraSPARC III 扩展和 VIS 2.0 版增加到 SPARC-V9 ISA 的 V9a 版。使用此选项进行编译将使用最佳指令集，以便在 Solaris UltraSPARC III 环境中获得良好性能。 • 最终的目标代码采用 SPARC-V9 ELF64 格式，而且只能与其它格式相同的 SPARC-V9 目标文件进行链接。 • 最终的可执行程序只能在运行具有 64 位内核的 64 位 Solaris 操作环境的 UltraSPARC III 处理器上运行。 • -xarch=v9b 仅当在 64 位 Solaris 操作环境中编译时才可用。

A.3.64.2 仅适用于 X86

表 A-12 x86 上的 -xarch 值

值	含义
generic	将指令集限制于 Intel x86 体系结构，等价于 386 选项。
native	为在此系统上获得良好性能而进行编译。这是 -fast 选项的缺省值。编译器为当前正编译的系统处理器选择适当的设置。
386	将指令集限制于 Intel 386/486 体系结构。
pentium_pro	将指令集限制于 pentium_pro 体系结构。

A.3.65 -xautopar

(SPARC) 为多个处理器打开自动并行化。进行依赖性分析 （分析循环以了解迭代间数据依赖性）和循环重构。如果优化级别不是 -xO3 或更高级别，则将优化级别提高到 -xO3 并发出警告。

如果要执行自己的线程管理，请不要使用 -xautopar。

要使执行速度更快，则该选项需要多处理器系统。在单处理器系统上，最终的二进制程序通常运行较慢。

要确定您使用的处理器数量，请使用 psrinfo 命令：

```
% psrinfo
0      on-line since 01/12/95 10:41:54
1      on-line since 01/12/95 10:41:54
3      on-line since 01/12/95 10:41:54
4      on-line since 01/12/95 10:41:54
```

要请求多个处理器，请设置 PARALLEL 环境变量。缺省值为 1。

- 请求的处理器数量不能超出可用的处理器数量。
- 如果 N 代表机器上的处理器数量，则对于单用户多处理器系统，请尝试使用 PARALLEL=N-1。

如果您使用 -xautopar 并在一个步骤中进行编译和链接，则链接将自动包括微任务库和线程安全 C 运行时库。如果您使用 -xautopar 并分别在*单独的*步骤中进行编译和链接，则您还必须链接 -xautopar。

A.3.66 -xbuiltin[=(%all | %none)]

如果您要改善对调用标准库函数的代码的优化，请使用 -xbuiltin[=(%all|%none)] 命令。很多标准库函数，例如用 math.h 和 stdio.h 定义的函数，通常由多个程序使用。此命令使编译器在对性能有益的地方替换内函数或内联系统函数。

如果您不指定 -xbuiltin，则缺省值为 -xbuiltin=%none，该值表示不替换或内联标准库中的函数。如果您指定 -xbuiltin，但未提供任何参数，则缺省值为 -xbuiltin%all，该值表示编译器在确定优化益处时替换内函数或内联标准库函数。

如果您使用 -fast 进行编译，则 -xbuiltin 设置为 %all。

注 – -xbuiltin 仅内联系统头文件中定义的全局函数，从不内联用户定义的静态函数。

A.3.67 -xCC

如果您指定 -xc99=%none 和 -xCC，则编译器接受 C++ 风格注释。特别地，可使用 // 来指明注释的开始。

A.3.68 -xc99[=*o*]

-xc99 标志控制编译器对 C99 标准 （ISO/IEC 9899:1999，编程语言 - C）中已实现功能的识别。

o 可以是包含以下内容的以逗号分隔的列表：

表 A-13 -xc99 值

值	含义
[no%]lib	[不] 启用出现在 1999 和 1999 C 标准中的 1990 C 标准例程库语义。
%all	打开对支持的 C99 功能的识别。
%none	关闭对 C99 功能的识别。

如果您未指定 -xc99，则编译器的缺省值为 -xc99=%all,no%lib。如果您指定了 -xc99，但没有指定任何值，则该选项设置为 -xc99=%all。

编译器将 %all、 %none 和 no%lib 分别视为 all、 none 和 no_lib 的同义字。

注 – 虽然编译器的支持级别的缺省值为附录 D 中列出的 C99 功能，但是 Solaris 软件在 `/usr/include` 提供的标准头文件仍不符合 1999 ISO/IEC C 标准。如果遇到错误消息，请尝试使用 `-xc99=%none` 获取这些头文件的 1990 ISO/IEC C 标准行为。

A.3.69 -xcache[=*c*]

定义供优化器使用的缓存属性。 *c* 必须是以下各项之一：

- `generic`
- *s1*/*l1*/*a1*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*

si/*li*/*ai* 定义如下：

si 级别为 *i* 的数据缓存的大小，以千字节表示

li 级别为 *i* 的数据缓存的行大小，以千字节表示

ai 级别为 *i* 的数据缓存的关联性

虽然此选项可单独使用，但它是 `-xtarget` 选项扩展的一部分；其*主要*用途是覆盖 `-xtarget` 选项提供的值。

此选项指定优化器可以使用的缓存属性。但不保证可使用任何特定缓存属性。下表列出了 `-xcache` 值：

表 A-14 `-xcache` 值

值	含义
`generic`	这是缺省值，该值指示编译器使用缓存属性以便在大多数 x86 和 SPARC 处理器上获得良好性能，而不会使任何处理器性能明显下降。 这些最佳计时属性可能会根据新发行版本的不同而适当进行调整。
`native`	设置参数以便在主机环境中获得最佳性能。
s1/*l1*/*a1*	定义一级缓存属性。
s1/*l1*/*a1*:*s2*/*l2*/*a2*	定义一级和二级缓存属性。
s1/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*	定义一级、二级和三级缓存属性。

示例：-xcache=16/32/4:1024/32/1 指定以下内容：

一级缓存具有：	二级缓存具有：
16 千字节	1024 千字节
32 字节行大小	32 字节行大小
4 向关联性	直接映射关联性

A.3.70 -xcg[89|92]

(SPARC)

-xcg89 是一个宏，用于：-xarch=v7 -xchip=old -xcache=64/32/1。

-xcg92 是一个宏，用于：-xarch=v8 -xchip=super

-xcache=16/32/4:1024/32/1。

A.3.71 -xchar[=*o*]

提供此选项的只是为了方便从字符类型被定义为无符号类型的系统中迁移代码。如果不是从此类系统中迁移，请勿使用此选项。只有那些依赖于字符类型符号的代码才需要重写，以显式指定带符号或者无符号类型。

您可以将 *o* 替换成以下选项之一：

表 A-15 -xchar 值

值	含义
signed	将声明为字符的字符常量和变量视为带符号类型。这将会影响所编译代码的行为，但不会影响库例程的行为。
s	等价于 signed
unsigned	将声明为字符的字符常量和变量视为无符号类型。这将会影响所编译代码的行为，但不会影响库例程的行为。
u	等价于 unsigned

如果您不指定 -xchar，则编译器假定 -xchar=s。

如果您指定 -xchar，但未指定值，则编译器假定 -xchar=s。

-xchar 选项只对使用 -xchar 编译的代码更改字符类型的值的范围。对于任何系统例程或头文件中的字符类型，此选项不会更改其值的范围。特别地，如果指定了此选项，limits.h 定义的 CHAR_MAX 和 CHAR_MIN 的值不会更改。因此， CHAR_MAX 和 CHAR_MIN 不再表示在无格式字符中可编码的值的范围。

如果您使用 -xchar，由于宏中的值可能带符号，因此在将字符与预定义的系统宏进行比较时要特别谨慎。这对返回可通过宏访问错误代码的例程来说十分常见。错误代码通常为负值，因此将字符与来自此类宏的值进行比较时，结果始终为假。任何无符号类型的值不能为负数。

强烈建议您决不要使用 -xchar 为任何通过库输出的接口编译例程。Solaris ABI 将字符类型指定为带符号，系统库的行为从而做相应调整。将字符指定为无符号的影响尚未使用系统库进行广泛测试。如果不使用此选项，您可以修改代码以使其不依赖于字符类型是带符号还是无符号。字符类型的符号因编译器和操作系统的不同而变化。

A.3.72 -xchar_byte_order[=*o*]

通过按照指定字节顺序排列多字符字符常量的字符来生成整型常量。您可以将 *o* 替换成下列值之一：

- low：按从低至高的字节顺序排列多字符字符常量的字符。
- high：按从高至低的字节顺序排列多字符字符常量的字符。
- default：按编译模式 -X[c|a|t|s] 确定的顺序排列多字符字符常量的字符。有关详细信息，请参见第 2-2 页上的第 2.1.2 节 “字符常量”。

A.3.73 -xcheck[=*o*]

(SPARC)

使用 -xcheck=stkovf 进行编译将增加对单线程程序中的主线程的栈溢出的运行时检查，以及对多线程程序中的从属线程栈的栈溢出的运行时检查。如果检查到栈溢出，则生成一个 SIGSEGV。如果您的应用程序需要以处理其它地址空间违规不同的方式处理栈溢出导致的 SIGSEGV，请参见 sigaltstack(2)。

您可以将 *o* 替换成下列值之一：

表 A-16 -xcheck 值

值	含义
%none	不执行任何 -xcheck 检查。
%all	执行所有 -xcheck 检查。
stkovf	打开栈溢出检查。
no%stkovf	关闭栈溢出检查。

如果您未指定 -xcheck，则编译器的缺省值为 -xcheck=%none。如果您指定了 -xcheck，但未指定任何参数，则编译器的缺省值为 -xcheck=%all，即打开对栈溢出的运行时检查。

-xcheck 选项并不在命令行累积。编译器根据命令的最后出现的值来设置标记。

A.3.74 -xchip[=*c*]

指定供优化器使用的目标处理器。

c 必须是以下各项之一：generic, old, super, super2, micro, micro2, hyper, hyper2, powerup, ultra, ultra2, ultra2e, ultra2i, ultra3, ultra3cu, 386, 486, pentium, pentium_pro。

虽然此选项可单独使用，但它是 -xtarget 选项扩展的一部分；其*主要*用途是覆盖 -xtarget 选项提供的值。

此选项通过指定目标处理器来指定计时属性。

其部分影响表现在以下方面：

- 指令的顺序，即调度
- 编译器使用分支的方法

■ 在具有语义上等效的指令时使用的指令。

表 A-17 `-xchip` 值

值	含义
`generic`	使用计时属性，以便在大多数 x86 和 SPARC 体系结构上获得良好性能。这是缺省值，该值指示编译器使用最佳计时属性以便在多数处理器上获得良好性能，而不会使任何处理器性能明显下降。
`old`	使用 SuperSPARC 以前的处理器的计时属性。
`super`	使用 SuperSPARC 处理器的计时属性。
`super2`	使用 SuperSPARC II 处理器的计时属性。
`micro`	使用 microSPARC 处理器的计时属性。
`micro2`	使用 microSPARC II 处理器的计时属性。
`hyper`	使用 hyperSPARC 处理器的计时属性。
`hyper2`	使用 hyperSPARC II 处理器的计时属性。
`powerup`	使用 Weitek PowerUp 处理器的计时属性。
`ultra`	使用 UltraSPARC 处理器的计时属性。
`ultra2`	使用 UltraSPARC II 处理器的计时属性。
`ultra2e`	使用 UltraSPARC IIe 处理器的计时属性。
`ultra2i`	使用 UltraSPARC IIi 处理器的计时属性。
`ultra3`	使用 UltraSPARC III 处理器的计时属性。
`ultra3cu`	使用 UltraSPARC III Cu 处理器的计时属性。
`386`	使用 Intel 386 体系结构的计时属性。
`486`	使用 Intel 486 体系结构的计时属性。
`pentium`	使用 Intel pentium 体系结构的计时属性。
`pentium_pro`	使用 Intel pentium_pro 体系结构的计时属性。

A.3.75 -xcode[=*v*]

(SPARC) 指定代码地址空间。

注 – 强烈建议您通过指定 `-xcode=pic13` 或 `-xcode=pic32` 生成共享对象。可以使用 `-xarch=v9 -xcode=abs64` 和 `-xarch=v8 -xcode=abs32` 生成可用的共享对象，但这样做效率低。使用 `-xarch=v9 -xcode=abs32` 或 -xarch=v9 -xcode=abs44 生成的共享对象将无法工作。

---

v 必须是以下各项之一：

表 A-18 `-xcode` 值

值	含义
`abs32`	生成 32 位绝对地址。代码 + 数据 + bss 大小被限制为 2**32 字节。这在 32 位体系结构上是缺省值： `-xarch=(generic|v7|v8|v8a|v8plus|v8plusa|v8plusb)`
`abs44`	生成 44 位绝对地址。代码 + 数据 + bss 大小被限制为 2**44 字节。仅在 64 位体系结构上可用：`-xarch=(v9|v9a|v9b)`
`abs64`	生成 64 位绝对地址。仅在 64 位体系结构上可用：`-xarch=(v9|v9a|v9b)`
`pic13`	生成与位置无关的代码，以便在共享库中使用（小模型）。等价于 -Kpic。在 32 位体系结构上允许引用的唯一外部符号最多为 2**11，在 64 位体系结构上为 2**10。 `-xcode=pic13` 命令类似于 `-xcode=pic32`，只有全局偏移表的大小被限制为 8 千字节。
`pic32`	生成与位置无关的代码，以便在共享库中使用 （小模型）。等价于 -KPIC。在 32 位体系结构上允许引用的唯一外部符号最多为 2**30，在 64 位体系结构上为 2**29。 每个对全局数据的引用在全局偏移表中均生成为指针非关联化。在 `pc` 相对寻址模式下通过过程链接表生成每个函数调用。如果设置此选项，在 `-xcode=pic32` 具有太多全局数据对象的稀有情况下，全局偏移表可跨越 32 位地址的范围。

对于 SPARC V7 和 V8，缺省值为 `-xcode=abs32` ；对于 SPARC 和 UltraSPARC V9（带有 `-xarch=v9|v9a`），缺省值为 `-xcode=abs64`。

如果使用 `-xarch=v9`、`v9a` 或 `v9b` 在 64 位 Solaris 操作环境中生成共享动态库，您可以指定 `-xcode=pic13` 或 `-xcode=pic32`，但实际上并不需要这样做。

在 SPARC 上使用 -xcode=pic13 和 -xcode=pic32 的标称性能成本有两种：

- 使用 -xcode=pic13 或 -xcode=pic32 编译的例程在进入时执行一些额外指令，以便将寄存器设置为指向用于访问共享库的全局或静态变量的表 (_GLOBAL_OFFSET_TABLE_)。
- 每次访问全局或静态变量均涉及通过 _GLOBAL_OFFSET_TABLE_ 进行的额外间接内存引用。如果使用 -xcode=pic32 完成编译，则每个全局和静态内存引用均具有两个附加指令。

在考虑以上成本时，请注意，由于库代码共享的影响，使用 -xcode=pic13 和 -xcode=pic32 可以明显降低系统内存需求。使用 -xcode=pic13 或 -xcode=pic32 编译的共享库中的各个代码页可以由使用该库的各个进程共享。如果共享库中的某个代码页包含单个非 pic （即绝对）内存引用，则该页将不可共享，并且每次执行使用该库的程序时必须创建该页的副本。

确定 .o 文件是否已使用 -xcode=pic13 或 -xcode=pic32 进行编译的最简单的方法是使用 nm 命令：

```
% nmfile.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

包含与位置无关的代码的 .o 文件包含一个对 _GLOBAL_OFFSET_TABLE_ 的未解析外部引用，表示为字母 U。

要确定是使用 -xcode=pic13 还是 -xcode=pic32，请使用 nm 来识别在库中使用或定义的不同全局和静态变量的数量。如果 _GLOBAL_OFFSET_TABLE_ 小于 8,192 字节，则您可以使用 -Kpic。否则，您必须使用 -xcode=pic32。

A.3.76 -xcrossfile[=*n*]

(SPARC) 启用跨源文件的优化和内联处理。如果指定该选项，则 *n* 可为 0 或 1。

通常情况下，编译器分析的范围仅限于命令行的每个独立文件。例如，-xO4 的自动内联处理仅限于同一源文件中定义和引用的子程序。

如果设置 -xcrossfile 选项，编译器将分析在命令行指定的所有文件，仿佛这些文件已被并置为单个源文件。-xcrossfile 仅在与 -xO4 或 -xO5 同时使用时才有效。

此编译生成的文件因可能会发生的内联处理而互相依赖，并且在链接到程序时必须作为一个单元。如果某个例程发生更改且文件被重新编译，则必须重新编译所有文件。因此，使用此选项将会影响 make 文件的构成。

缺省值为 -xcrossfile=0，同时不执行任何交叉文件优化。-xcrossfile 等价于 -xcrossfile=1。

参见 -xldscope。

A.3.77 -xcsi

允许 C 编译器接受在不符合 ISO C 源字符代码要求的语言环境中编写的源代码。这些语言环境包括：ja_JP.PCK。

需要处理此类语言环境的编译器转换阶段可能导致编译时间明显增长。仅当您编译的源文件包含此类语言环境中某个语言环境的源代码字符时，您才应该使用此选项。

除非您指定 -xcsi，否则编译器不识别在不符合 ISO C 源字符代码要求的语言环境中编写的源代码。

A.3.78 -xdebugformat=[stabs|dwarf]

如在“DWARF 调试信息格式”中指定的那样，C 编译器正将调试器信息从 stabs 格式迁移到 dwarf 格式。如果要维护读取调试信息的软件，您可以使用此选项将工具从 stabs 格式转换为 dwarf 格式。在此发行版本中，缺省设置为 -xdebugformat=stabs。

使用此选项可作为一种为移植工具而访问新格式的方法。除非您要维护读取调试器信息的软件，或者特定工具要求使用这些格式之一的调试器信息，否则没有必要使用此选项。

-xdebugformat=stabs 生成的调试信息采用 stabs 标准格式。

-xdebugformat=dwarf 生成的调试信息采用 dwarf 标准格式。

如果您未指定 -xdebugformat，则编译器假定 -xdebugformat=stabs。指定此选项而不带参数是非法操作。

此选项影响使用 -g 选项记录的数据的格式。即使在没有使用 -g 的情况下记录少量调试信息，此选项仍可控制其信息格式。因此，即使不使用 -g， -xdebugformat 仍有影响。

dbx 和性能分析软件理解 stabs 和 dwarf 格式，因此使用此选项对任何工具的功能性并无影响。

注 – 这是过渡性接口，因此在发行版本之间会发生更改而不兼容，即使在发行版本更新较少时也是如此。stabs 或 dwarf 格式的任何特定字段或值的详细资料也在不断改进。

A.3.79 -xdepend=[yes|no]

(SPARC) 分析循环以了解迭代间数据依赖性并执行循环重构。

循环重构包括循环交换、循环合并、标量替换和“死”数组赋值消除。如果优化级别不是 -xO3 或更高级别，则编译器将优化级别提高到 -xO3 并发出警告。

如果您未指定 -xdepend，则缺省值为 -xdepend=no，该值表示编译器不分析循环以了解数据依赖性。如果您指定 -xdepend，但未指定参数，则编译器将此选项设置为 -xdepend=yes，该值表示编译器分析循环以了解数据依赖性。

使用 -xautopar 或 -xparallel 时也包括依赖性分析。依赖性分析在编译时完成。

依赖性分析可能有助于单处理器系统。然而，如果要在单处理器系统上使用 -xdepend，则您不应使用 -xautopar 或 -xexplicitpar。如果使用了二者之一，则对多处理器系统执行 -xdepend 优化。

A.3.80 -xdryrun

此选项是用于 -### 的宏。

A.3.81 -xe

对源文件仅执行语法和语义检查，但不生成任何对象或可执行代码。

A.3.82 -xexplicitpar

(SPARC) 生成基于 #pragma MP 指令规范的并行化代码。您可以进行依赖性分析：分析并指定循环以了解迭代间数据依赖性。软件将并行化指定的循环。如果优化级别不是 -xO3 或更高级别，则将优化级别将提高到 -xO3 并发出警告。如果您执行自己的线程管理，则不要使用 -xexplicitpar。

要使代码的运行速度更快，则该选项需要多处理器系统。在单处理器系统上，生成的代码通常运行较慢。

如果您指定某个要并行化的循环且该循环具有依赖性，则您可能会得到错误的结果，有可能每次运行的结果都不相同且不发出警告。请勿将显式并行 pragma 应用于约简循环。执行显式并行化，但不执行循环的约简，因此结果可能不正确。

总之，要显式并行化，请执行以下步骤：

- 分析循环以查找可安全并行化的代码。
- 插入 `#pragma MP` 以并行化循环。有关详细信息，请参见第 3-20 页上的第 3.8.3 节“显式并行化和 Pragma”。
- 使用 `-xexplicitpar` 选项。

以下是一个在紧挨在循环之前插入并行 pragma 的示例：

```
#pragma MP taskloop
  for (j=0; j<1000; j++){
  ...
  }
```

如果您使用 `-xautopar` 并在一个步骤中进行编译和链接，则链接将自动包括微任务库和线程安全 C 运行时库。如果您使用 `-xexplicitpar` 并在*单独的*步骤中进行编译和链接，则您还必须与 `-xautopar` *链接*。

请勿同时指定 `-xexplicitpar` 和 `-xopenmp`。

A.3.83 -xF

允许链接程序对函数和变量进行最佳重新排序。

该选项要求编译器将函数和/或数据变量放入单独的分段中，通过在链接程序的 `-M` 选项指定的映射文件中使用方向来对这些段重新排序以优化程序性能。通常，仅当缺页时间是程序运行时间的重要组成部分时，此优化才有效。

重新排序变量有助于解决以下对运行时性能产生负面影响的问题：

- 内存中相互靠近的无关变量导致的缓存和页争用。
- 内存中相距较远的相关变量导致工作集不必要增大。
- 由于未使用可降低有效数据密度的弱变量副本而导致工作集不必要增大。

为达到最佳性能，变量和函数重新排序需要下列操作：

1. **使用 `-xF` 进行编译和链接。**
2. **按照《程序性能分析工具》手册中关于如何生成函数的映射文件，或“链接程序和库向导”中关于如何生成数据的映射文件的说明进行操作。**
3. **通过使用链接程序的 `-M` 选项重新链接新的映射文件。**
4. **用分析器重新执行以检验提高情况。**

v 可以是下列值中的一个或多个：

表 A-19 `-xF` 值

值	含义
[no%]func	[不] 将函数分割为单独的段。
[no%]gbldata	[不] 将全局数据 （具有外部链接的变量）分割为单独的段。
%all	分割函数和全局数据。
%none	不进行任何分割。

如果未指定 `-xF`，则缺省值为 `-xF=%none`。如果指定不带任何参数的 `-xF`，则缺省值为 `-xF=%none,func`。

参见 `analyzer`(1)、 `debugger`(1)、 `ld`(1) 手册页

A.3.84 `-xhelp=`*f*

显示联机帮助信息。

f 必须是 `flags` 或 `readme`。

`-xhelp=flags` 显示编译器选项汇总信息。

`-xhelp=readme` 显示 `README` 文件。

A.3.85 `-xhwcprof`

(SPARC) 允许编译器支持基于硬件计数器的文件配置。

启用 `-xhwcprof=[enable|disable]` 时，编译器会生成信息来帮助工具匹配硬件计数器的数据引用，并忽略相关指令引发的事件。相关的数据类型和结构成员可能也会随着符号信息 （由 `-g` 生成）一起进行标识。这些信息对性能分析可能有用，但很难区别于基于代码地址的配置文件、源代码语句或例程。

您可使用 `-xhwcprof` 编译一组指定的目标文件。然而， `-xhwcprof` 的最大作用表现在应用于应用程序中的所有目标文件时。它能全面识别并关联分布在应用程序目标文件中的所有内存引用。

如果您分别在单独的步骤中进行编译和链接，最好在链接时使用 `-xhwcprof`。如果将来扩展为 `-xhwcprof`，则在链接时可能会使用它。

-xhwcprof=enable 或 -xhwcprof=disable 的实例将会覆盖同一命令行上的 -xhwcprof 的所有以前的实例。

在缺省情况下，禁用 -xhwcprof。指定不带任何参数的 -xhwcprof 等价于 -xhwcprof=enable。

-xhwcprof 要求应启用优化并且调试数据的格式应设置为 DWARF (-xdebugformat=dwarf)。

-xhwcprof 和 -g 的组合会增加编译器临时文件的存储需求，而且比单独指定 -xhwcprof 和 -g 所引起的增加的总数还多。

下列命令可编译 example.c 并为硬件计数器文件配置，以及针对使用 DWARF 符号的数据类型和结构成员的符号分析指定支持：

```
example% cc -c -O -xhwcprof -g -xdebugformat=dwarf example.c
```

有关基于硬件计数器的文件配置的详细信息，请参见 *《程序性能分析工具》*手册。

A.3.86 -xildoff

关闭递增链接程序并强制使用 ld。如果您未使用 -g 选项、或者使用 -G 选项、或者命令行上存在任何源文件，则该选项为缺省值。通过使用 -xildon 选项覆盖该缺省值。

A.3.87 -xildon

打开递增链接程序并在递增模式下强制使用 ild。如果您使用 -g 选项、且未使用 -G 选项、同时命令行上不存在任何源文件，则该选项是缺省值。使用 -xildoff 选项覆盖该缺省值。

A.3.88 -xinline=*list*

-xinline 的 *list* 格式如下：
[{%auto,*func_name*,no%*func_name*}[,{%auto,*func_name*,no%*func_name*}]...]

-xinline 只内联可选列表中指定的函数。该列表可为空，或者由逗号分开的 func_name、no%func_name 或 %auto 列表组成，其中 *func_name* 是函数名称。仅在 -xO3 或更高级别上， -xinline 才有效。

表 A-20 -xinline 值

值	含义
%auto	指定编译器将自动内联源文件中的所有函数。仅在 -xO4 或更高优化级别上， %auto 才起作用。在 -xO3 或更低优化级别上， %auto 被忽略而无提示。
func_name	指定编译器内联已命名的函数。
no%func_name	指定编译器不内联已命名的函数。
无值	指定编译器不内联源文件中的任何函数。

该列表值从左至右进行累积。因此对于 -xinline=%auto,no%foo 的规范，编译器试图内联除 foo 之外的所有函数。对于 -xinline=%bar,%myfunc,no%bar 的规范，编译器仅尝试内联 myfunc。

当您在优化级别为 -xO4 或更高级别上编译时，编译器通常尝试内联源文件中定义的对函数的所有引用。通过指定 -xinline 选项，您可以限定编译器试图内联的函数集。如果只指定 -xinline=，即不指定任何函数或 %auto，这表示不内联源文件中的任何例程。如果您指定了一系列 func_name 和 no%func_name，而未指定 %auto，则编译器只试图内联列表中指定的函数。如果在优化级别为 -xO4 或更高级别上用 -xinline 选项在值列表中指定了 %auto，则编译器试图内联所有未被 no%func_name 显式排除的函数。

如果以下任何条件适用，则不内联函数。且不发出任何警告。

- 优化级别低于 -xO3。
- 无法找到例程。
- 优化器无法内联例程。
- 正编译的文件中不存在例程的源代码 （请参见 -xcrossfile）。

如果在命令行指定多个 -xinline 选项，则它们不会累积。命令行上的最后一个 -xinline 指定编译器试图内联的函数。

参见 -xldscope。

A.3.89 -xipo[=*a*]

(SPARC) 用 0、1 或 2 替换 *a*。不带任何参数的 -xipo 等价于 -xipo=1。-xipo=0 是缺省值，此值将关闭 -xipo。

通过调用过程间调用分析组件，编译器执行整个程序优化。不同于 -xcrossfile，-xipo 在链接步骤中对所有目标文件执行优化，并且不仅限于编译命令的源文件。在 -xipo=1 时，编译器对所有源文件执行内联处理。在 -xipo=2 时，编译器执行过程间调用别名分析同时优化内存分配和布局，以提高缓存的性能。

由于优化文件时需要附加信息，因此 -xipo 选项会生成更大的目标文件。但是，这些附加信息不会成为最终的可执行二进制文件的一部分。可执行程序在大小上的任何增加均起因于附加优化的执行。在编译步骤中创建的目标文件在其内部编译附加分析信息，以允许链接步骤执行交叉文件优化。

在编译和链接大型多文件应用程序时，-xipo 特别有用。使用此标志编译的目标文件在其内部编译分析信息，从而在源文件和预编译的程序文件间启用过程间调用分析。

但是，分析和优化仅限于使用 -xipo 编译的目标文件，且不会扩展到库中的目标文件。

-xipo 是多相的，如果您分别在单独的步骤中编译和链接，则您需要为每一步指定 -xipo。

在此例中，编译和链接在单独的步骤中进行：

```
cc -xipo -xO4 -o prog part1.c part2.c part3.c
```

优化器在三个源文件之间执行交叉文件内联处理。这是在最终链接步骤里完成的，因此源文件的编译不必全部在单个编译中进行，可以通过多个单独的编译来进行，且每个编译都要指定 -xipo。

在此例中，编译和链接在单独的步骤中进行：

```
cc -xipo -xO4 -c part1.c part2.c
cc -xipo -xO4 -c part3.c
cc -xipo -xO4 -o prog part1.o part2.o part3.o
```

即使用 -xipo 编译，仍存在库不参与交叉文件过程间调用分析的约束，如下例所示：

```
cc -xipo -xO4 one.c two.c three.c
ar -r mylib.a one.o two.o three.o
...
cc -xipo -xO4 -o myprog main.c four.c mylib.a
```

在此例中，过程间调用优化是在以下例程之间执行的：one.c、 two.c 和 three.c 之间、 main.c 和 four.c 之间，但不在 main.c 或 four.c 和 mylib.a 上的例程之间执行。（第一次编译可能产生未定义符号警告，但是由于它是编译与链接步骤，所以会执行过程间调用优化。）

关于 -xipo 的其它重要信息：

- 它需要一个最低为 -xO4 的优化级别。
- 它与 -xcrossfile 冲突。如果它们一起使用，则会产生编译错误。
- 没有使用 -xipo 编译的对象可以自由地与使用 -xipo 编译的对象链接。

参见：-xjobs

A.3.90 -xjobs=*n*

指定 -xjobs 选项，以设置编译器为完成其工作需要创建的进程数。在多 cpu 机器上，此选项可以减少生成时间。当前， -xjobs 只能与 -xipo 选项一起使用。当您指定 -xjobs=*n* 时，过程间调用优化器使用 *n* 作为它可调用以编译不同文件的代码生成器实例最大数目。

通常， *n* 的安全值为可用处理器数的 1.5 倍。使用高于可用处理器数量很多倍的值可能会因所产生任务间的上下文切换开销而降低性能。此外，该值过大也会耗尽系统资源的极限，例如交换空间。

您必须始终为 -xjobs 指定一个值。否则会发出一个错误诊断，然后编译终止。

在到达最右边的实例之前，命令行上 -xjobs 的多个实例相互覆盖。

相对于使用不带 -xjobs 选项的相同命令，以下示例在双处理器系统上的编译速度更快。

```
example% cc -xipo -xO4 -xjobs=3 t1.c t2.c t3.c
```

A.3.91 -xldscope={*v*}

指定 -xldscope 选项以更改外部符号定义的缺省链接程序作用域。由于实现更隐蔽，因此更改缺省值可使共享库更快、更安全。

v 必须是以下选项之一：

表 A-21 -xldscope 值

值	含义
global	全局链接程序作用域是受限制最少的链接程序作用域。该符号的所有引用都绑定到在第一个动态模块中定义该符号的定义上。该链接程序作用域是外部符号的当前链接程序作用域。
symbolic	符号链接程序作用域，其限制比全局链接程序作用域的限制更多。从所链接的动态模块内部对符号的所有引用都绑定到在模块内部定义的符号上。在模块外部，该符号如同全局符号。此链接程序作用域对应于链接程序选项 -Bsymbolic。有关链接程序的详细信息，请参见 ld(1)。
hidden	隐藏链接程序作用域的限制比符号链接程序作用域和全局链接程序作用域的限制更多。动态模块内部的所有引用都绑定到该模块内部的一个定义上。该符号在模块外部不可见。

如果不指定 -xldscope，则编译程序假设 -xldscope=global。如果您指定不带参数的 -xldscope，则编译器会发出错误信息。在到达最右边的实例之前，命令行上此选项的多个实例相互覆盖。

如果您要允许客户程序覆盖库中的函数，则您必须确保在库生成期间不会内联生成函数。编译程序在以下情况下将内联函数：指定函数的名称为 -xinline、在 -xO4 或更高级别上（此时自动进行内联处理）编译、使用内联说明符、使用内联 pragma、或者使用交叉文件优化。

例如，假设库 ABC 具有既可以由库客户使用又可以在库内使用的缺省分配程序函数：

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

如果您在 -xO4 或更高级别上创建库，则编译器内联调用库组件中出现的 ABC_allocator。如果库客户要用自定义版本替换 ABC_allocator，则该替换不会在称为 ABC_allocator 的库组件中出现。最终程序将包括此函数的不同版本。

在生成库时，可内联生成使用 __hidden 或 __symbolic 声明的库函数。它们将不会被客户程序覆盖。参见第 2-3 页上的 “链接程序作用域说明符”。

使用 __global 说明符声明的库函数不应进行内联声明，而应使用 -xinline 编译器选项来避免内联处理。

参见 -xinline、 -xO、 -xcrossfile、 #pragma inline

A.3.92 -xlibmieee

强制 IEEE 754 样式返回异常情况下的数学例程值。在这种情况下，不会打印异常消息，您不应该依赖于 errno。

A.3.93 -xlibmil

内联部分库例程以加快执行速度。您可以使用此选项为浮点选项选择适当的汇编语言内联模板，并为您的系统选择适当的平台。

无论函数的任何规范作为 -xinline 标记的一部分， -xlibmil 都会内联函数。

A.3.94 -xlic_lib=sunperf

(SPARC) 在 Sun 提供的性能库中进行链接。

A.3.95 -xlicinfo

返回所用许可证文件信息、接受的许可证标记、序列号、 RTU、试用许可证和过期前的天数。该选项不要求编译或检查许可证。

A.3.96 -xlinkopt[=*level*]

要求编译器对重定位目标文件执行链接时间优化。在链接时通过分析目标二进制代码来执行这些优化操作。不能重复写入目标文件，但是最终的可执行代码可能与源目标代码不同。

由于 -xlinkopt 在链接时很有用，因此您必须至少在某些编译命令中使用 -xlinkopt。该优化器还可以对不是使用 -xlinkopt. 编译的目标二进制文件执行某些有限的优化。

-xlinkopt 优化来自编译程序命令行上静态库的代码，但是它将跳过且不优化来自命令行上的共享 （动态）库的代码。当您创建共享库 （用 -G 编译）时，您也可以使用 -xlinkopt。

level 用来设置执行的优化级别，其值必须是 0、 1 或 2。优化级别是：

表 A-22 -xlinkopt 值

后优化器设置	行为
0	禁用后优化器。 （这是缺省值。）
1	在链接时根据控制流分析执行优化，包括指令缓存着色和分支优化。
2	在链接时执行附加数据流分析，包括停用代码消除和地址计算简化。

如果您在单独的步骤中进行编译，则在编译和链接步骤中均要出现 -xlinkopt：

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

注意，仅当编译器进行链接时才能使用级别参数。在以上示例中，即使目标二进制代码是用缺省级别 1 编译的，使用的后优化级别仍然是 2。

没有使用级别参数指定 -xlinkopt 表示 -xlinkopt=1。

此选项在用于编译整个程序并使用配置文件反馈时最有效。文件配置会显示代码中最常用和最少用的部分，生成会相应地指导优化器集中其努力方向。这对大型应用程序非常重要，因为连接时执行代码的优化放置可以减少指令的缓存缺失。一般来说，此编译的情况如下所示：

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

有关使用配置文件反馈的详情，请参见第 A-69 页上的 “-xprofile=p”。

您不能同时使用链接时间后优化器和递增链接程序 ild。-xlinkopt 将缺省链接程序设置成 ld。如果您用 -xildon 来显式启用递增链接程序并指定 -xlinkopt，则 -xlinkopt 被禁用。

当用 -xlinkopt. 编译时，请勿使用 -zcompreloc 链接程序选项。

注意，用该选项编译会稍微增加链接时间。目标文件大小也会增加，但是可执行程序的大小仍保持不变。通过包含调试信息，用 -xlinkopt 和 -g 编译将会增加可执行程序的大小。

A.3.97 -xloopinfo

(SPARC) 显示哪些循环已并行化以及哪些循环尚未并行化。给出未并行化循环的简短原因。仅当指定了 -xautopar、-xparallel 或 -xexplicitpar 时，-xloopinfo 选项才有效，否则，编译器将发出警告。

要达到更快的执行速度，则该选项需要多处理器系统。在单处理器系统上，生成的代码通常运行速度较慢。

A.3.98 -xM

仅在 C 程序中运行预处理程序，这需要生成 makefile 依赖性并将结果发送到标准输出（有关 make 文件和依赖性的详细信息，请参见 make(1)）。

例如：

```
#include <unistd.h>
void main(void)
{}
```

生成以下输出：

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

A.3.99 -xM1

收集诸如 -xM 的依赖性，但排除 /usr/include 文件。例如：

```
more hello.c
#include<stdio.h>
main()
{
    (void)printf("hello\n");
}
cc -xM hello.c
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

使用 -xM1 编译不会报告头文件的依赖性：

```
cc -xM1 hello.c
hello.o: hello.c
```

-xM1 在 -Xs 模式下不可用。

A.3.100 -xMerge

将数据段合并到文本段中。在此编译产生的目标文件中初始化的数据是只读的，并且（除非与 ld -N 链接）在进程间是共享的。

A.3.101 -xmaxopt[=*v*]

其中 *v* 为以下各值之一：off、1、2、3、4、5。该命令将 pragma opt 的级别限定为指定的级别。缺省值为 -xmaxopt=off，该值导致 pragma opt 被忽略。如果指定 -xmaxopt 而不提供参数，则相当于指定 -xmaxopt=5。

A.3.102 -xmemalign=*ab*

指定最大假定内存对齐和未对齐数据访问的行为。*a*（对齐）和 *b*（行为）必须都具有值。*a* 指定最大假定内存对齐，*b* 指定未对齐内存访问的行为。下表列出了 -xmemalign 的对齐值和行为值

表 A-23 -xmemalign 对齐值和行为值

a		*b*	
1	假定至多对齐 1 字节。	i	解释访问并继续执行。
2	假定至多对齐 2 字节。	s	生成信号 SIGBUS。
4	假定至多对齐 4 字节。	f	为少于或等于 4 字节的对齐生成信号 SIGBUS，否则解释访问并继续执行。
8	假定至多对齐 8 字节。		
16	假定至多对齐 16 字节。		

对于编译时可确定对齐的内存访问，编译器将为此数据对齐生成适当的装入/存储指令序列。

对于编译时不能确定对齐的内存访问，编译器必须假定一个对齐以生成所需的装入/存储序列。

-xmemalign 标志允许用户在这些未确定情况下指定编译器要假定的数据最大内存对齐。它还指定在运行时（即当发生未对齐内存访问时）要跟随的错误行为。

下面是 -xmemalign 的缺省值。仅当 -xmemalign 标志不存在时，才能应用下列缺省值：

- 当 -xarch 的值为 generic、v7、v8、v8a、v8plus、v8plusa 时，-xmemalgin=4s。
- 当 -xarch 的值为 v9 或 v9a 时，-xmemalign=8s。

若 -xmemalign 标志存在但未赋值，则缺省值为：

- 对于所有 -xarch 值，-xmemalign=1i。

下表显示了如何使用 -xmemalign 来处理不同的对齐情况。

表 A-24 -xmemalign 示例

命令	情况
-xmemalign=1s	存在很多未对齐访问，因此陷阱处理相当慢。
-xmemalign=8i	存在偶然的、有意的、未对齐的访问，除此之外代码正确。
-xmemalign=8s	在程序中不应该存在未对齐访问。
-xmemalign=2s	您应该检查是否存在可能的奇字节访问
-xmemalign=2i	您应该检查是否存在可能的奇字节访问，且需要程序工作。

A.3.103 -xnativeconnect[=*a*[,*a*]...]

当您要在目标文件和后续共享库中包括接口信息时，请使用 -xnativeconnect 选项，以使共享库可与用 Java[tm] 编程语言 （Java 代码）编写的代码连接在一起。当您用 cc -G 命令生成共享库时，您必须也包括 -xnativeconnect。

当您用 -xnativeconnect 编译时，您所提供的是本机代码接口的最大外部可视性。本地连接器工具 (NCT) 启用 Java 代码和 Java[tm] 本机接口 (JNI) 代码的自动生成功能。同时使用 NCT 和 -xnativeconnect 可从 Java 代码中调用 C 共享库中的函数。有关如何使用 NCT 的详细信息，请参见联机帮助。

a 可以是以下值之一：

表 A-25 -xnativeconnect 值

值	含义
%all	生成在 -xnativeconnet 单个选项下描述的所有不同数据。
%none	不生成在 -xnativeconnet 单个选项下描述的任何不同数据。
[no%]inlines	强制生成引用的内联函数的外联实例。这为本机连接器提供了一种调用内联函数的外部可视的方法。不影响调用点上的这些函数的标准内联处理。
[no%]interfaces	强制生成二进制接口描述符 (BIDS)

如果您未指定 -xnativeconnect，则编译器将该选项设置为 -xnativeconnect=%none。如果您仅指定 -xnativeconnect，则编译器将该选项设置为 -xnativeconnect=interfaces。

-xnativeconnect=interfaces 强制生成二进制接口描述符 (BIDS)。

A.3.104 -xnolib

缺省情况下不链接任何库，也就是说，不将 -l 选项传递给 ld(1)。通常情况下， cc 驱动程序将 -lc 传递给 ld。

使用 -xnolib 时，您必须亲自传递所有 -l 选项。例如：

```
% cc test.c -xnolib -Bstatic -lm -Bdynamic -lc
```

静态链接 libm，动态链接其它库。

A.3.105 -xnolibmil

不内联数学库例程。在 -fast 选项之后使用。例如：

```
% cc -fast -xnolibmil....
```

A.3.106 -xO[1|2|3|4|5]

优化目标代码；注意，大写字母后带有数字 1、 2、 3、 4 或 5。一般来说，优化级别越高，运行时性能越好。但是，较高优化级别可能导致编译时间增加以及可执行文件增大。

在少数情况下， -xO2 级别的性能比其它级别好，而 -xO3 级别可能比 -xO4 级别好。尝试使用每一级别编译，从而确定您是否遇到这些少见的情况。

如果优化器耗尽内存，则它将在较低优化级别上通过重试当前程序来恢复优化，并在命令行选项上指定初始级别上继续执行后继程序。

对于 -xO，有五个级别可供使用。以下部分描述了如何在 SPARC 平台和 x86 平台上执行这些级别。

表 A-26 适用于 SPARC 处理器的 -xO 值

值	含义
-xO1	执行基本局部优化 （窥孔优化）。
-xO2	执行基本局部和全局优化。这表示感应变量排除、局部和全局通用子表达式排除、代数简化、副本传播、常量传播、循环不变量优化、寄存器分配、基本块合并、尾部递归排除、无用代码排除、尾部调用排除和复杂表达式扩展。 -xO2 级别不会将全局、外部或间接引用或定义分配给寄存器。它将这些引用和定义视为被声明为 volatile。一般说来，-xO2 级别产生的代码最小。
-xO3	类似于执行 -xO2，但还会优化外部变量的引用或定义。还执行循环解开和软件流水线作业。该级别不会跟踪指针赋值的效果。当编译设备驱动程序或程序从信号处理程序内部修改外部变量时，您可能需要使用 volatile 类型限定符来保护对象，使其免于优化。一般说来，-xO3 级别会导致代码增大。
-xO4	类似于执行 -xO3，但是还自动内联包含在相同文件中的函数；这通常会提高执行速度。如果您要控制内联哪些函数，请参见第 A-48 页上的 “-xinline=list”。 该级别跟踪指针赋值的效果，通常导致代码增大。
-xO5	试图生成最高优化级别。使用编译时间更长或减少执行时间的程度不是很高的优化算法。如果与配置反馈配合使用，则在此级别上的优化更有可能提高性能。参见第 A-69 页上的 “-xprofile=p”。

表 A-27 适用于 x86 处理器的 -xO 值

值	含义
-xO1	从内存预装入参数、交叉跳转 （尾部合并）以及缺省优化的单个传递。
-xO2	调度高级别和低级别指令并执行改善的溢出分析、循环内存引用排除、寄存器生存期分析、增强的寄存器分配和全局通用子表达式排除。
-xO3	执行循环长度约简、感应变量排除以及在级别 2 完成的优化。
-xO4	执行循环解开，避免可能创建栈帧和自动内联包含在相同文件中的函数，以及在级别 2 和 3 上完成优化。注意，该优化级别可能导致从 adb 和 dbx 的栈跟踪不正确。
-xO5	生成最高级别优化。使用编译时间更长或减少执行时间的程度不是很高的优化算法。其中包括为输出的函数生成局部调用固定入口点、进一步优化溢出代码和增加分析，以改善指令调度。

缺省值为不优化。但是，仅在您未指定优化级别时，才可能使用缺省值。如果您指定优化级别，则无任何用于关闭优化的选项。

如果您尝试不设置优化级别，请确保不指定任何隐含优化级别的选项。例如， -fast 是一个将优化级别设置为 -xO5 的宏选项。所有其它隐含优化级别的选项会发出已设置优化的警告信息。在没有优化的情况下，唯一的编译方法是从命令行上删除所有选项或生成指定优化级别的文件。

当您将 -xO 与 -g 选项一起使用时，则仅有少数调试可用。有关详细信息，请参见第一章 《*使用* dbx *调试程序*》中的 “调试优化的代码”。

如果您在 -xO3 或 -xO4 级别上优化非常大的程序 （在同一程序中有数千行代码），则优化器可能需要大量虚拟内存。在这种情况下，机器性能可能会降低。

有关调试方面的详细信息，请参见 《*使用* dbx *调试程序*》手册。有关优化方面的详细信息，请参见 《*程序性能分析工具*》手册。

参见 -xldscope。

A.3.107 -xopenmp[=*i*]

(SPARC) 使用 -xopenmp 选项以启用用 OpenMP 指令显式并行化。下表列出了 *i* 的值：

表 A-28 -xopenmp 值

i 的值	含义
parallel	启用 OpenMP pragma 识别。当 -xopenmp=parallel 时，优化级别为 -x03。如有必要，则编译器会将优化级别更改为 -x03 并发出警告。
noopt	启用 OpenMP pragma 识别。如果优化级别低于 -O3，则编译器不会提高优化级别。 如果您将优化级别显式设置为低于 -O3，如同在 cc -O2 -xopenmp=noopt 中一样，则编译器会发出错误。如果您未使用 -xopenmp=noopt 指定优化级别，则可以识别 OpenMP pragma，从而并行化程序，但是不执行任何优化。
stubs	禁用 OpenMP pragma 识别，链接到桩模块库例程且不更改优化级别。如果应用程序显式调用 OpenMP 运行时库例程而且您要将它编译成能连续执行，则使用该选项。-xopenmp=stubs 命令还定义 _OPENMP 预处理程序标记。
none	不启用 OpenMP pragma 识别，不修改程序的优化级别且不预定义任何预处理程序标记。

如果您指定 -xopenmp，但不包括值，则编译器假定 -xopenmp=parallel。如果您不指定 -xopenmp，则编译器假定 -xopenmp=none。

如果您使用 dbx 调试 OpenMP 程序，且使用 -g 和 -xopenmp=noopt 进行编译，则您可以在并行区域内设置断点并显示变量的内容。

请不要将 -xopenmp 和 -xexplicitpar 或 -xparallel 一起指定。

-xopenmp 的缺省值可能在将来的版本中有改变。您可以通过显式指定适当的优化而避免警告信息。

如果您在单独的步骤中编译和链接，则在链接步骤中还要指定 -xopenmp。这在编译包含 OpenMP 指令的库时特别重要。

有关如何编译顺应 OpenMP 的程序的详细信息，请参见第 3-2 页上的第 3.2 节“OpenMP 并行化”。

有关特定于 OpenMP C 实现的信息，请参见附录 G。

有关用于生成多重处理应用程序的 OpenMP Fortran 95、C 和 C++ 应用程序接口 (API) 的全部汇总信息，请参见 《*OpenMP API 用户指南*》。

A.3.108 -xP

打印在以下模块中定义的所有 K&R C 函数的原型。

```
f()
{
}

main(argc,argv)
int argc;
char *argv[];
{
}
```

产生以下输出：

```
int f(void);
int main(int, char **);
```

A.3.109 -xpagesize=*n*

(SPARC) 设置栈和堆的首选页面大小。

n 必须为以下值之一：8K、64K、512K、4M、32M、256M、2G、16G 或 default。

您必须为目标平台上的 Solaris 操作环境指定有效的页面大小，该值与 getpagesize(3C) 返回的值相同。如果不指定有效的页面大小，运行时将忽略请求而无提示。Solaris 操作环境并不保证接受页面大小请求。

您可以使用 pmap(1) 或 meminfo(2) 来确定目标平台上的页面大小。

除非在编译和链接时使用它，否则 -xpagesize 选项不会生效。

注 – Solaris 7 和 Solaris 8 操作环境中不提供此功能。在 Solaris 7 和 Solaris 8 操作环境中，使用此选项编译的程序将不会链接。

如果指定 -xpagesize=default，则 Solaris 操作环境设置页面大小。不带参数的 -xpagesize 等价于 -xpagesize=default。

使用此选项编译的效果等价于使用等价选项将 LD_PRELOAD 环境变量设置为 mpss.so.1，或者在运行程序之前使用等价选项运行 Solaris 9 命令 ppgsz(1)。详细内容请参见 Solaris 9 手册页。

此选项是用于 -xpagesize_heap 和 -xpagesize_stack 的宏。这两个选项接受与 -xpagesize 参数相同的参数：8K、 64K、 512K、 4M、 32M、 256M、 2G、 16G 或 default。您可以通过指定 -xpagesize 来为二者设置相同的值，或分别为它们指定不同的值。

A.3.110 -xpagesize_heap=*n*

(SPARC) 设置堆在内存中的页面大小。*n* 可以为 8K、 64K、 512K、 4M、 32M、 256M、 2G、 16G 或 default。您必须为目标平台上的 Solaris 操作环境指定有效的页面大小，此值与 getpagesize(3C) 返回的值相同。如果不指定有效的页面大小，运行时将忽略请求而无提示。

您可以使用 pmap(1) 或 meminfo(2) 来确定目标平台上的页面大小。

如果指定 -xpagesize_heap=default，则 Solaris 操作环境设置页面大小。不带参数的 -xpagesize_heap 等价于 -xpagesize_heap=default。

用该选项编译的效果等价于使用等价选项将 LD_PRELOAD 环境变量设置为 mpss.so.1，或者在运行程序之前使用等价选项运行 Solaris 9 命令 ppgsz(1)。详细内容请参见 Solaris 9 手册页。

注 – Solaris 7 和 Solaris 8 操作环境中不提供此功能。在 Solaris 7 和 Solaris 8 操作环境中，使用此选项编译的程序将不会链接。

A.3.111 -xpagesize_stack=*n*

(SPARC) 设置栈在内存中的页面大小。*n* 可以为 8K、 64K、 512K、 4M、 32M、 256M、 2G、 16G 或 default。您必须为目标平台上的 Solaris 操作环境指定有效的页面大小，此值与 getpagesize(3C) 返回的值相同。如果不指定有效的页面大小，运行时将忽略请求而无提示。您可以使用 pmap(1) 或 meminfo(2) 来确定目标平台上的页面大小。

如果指定 -xpagesize_stack=default，则 Solaris 操作环境设置页面大小。不带参数的 -xpagesize_stack 等价于 -xpagesize_stack=default。

使用此选项编译的效果等价于使用等价选项将 LD_PRELOAD 环境变量设置为 mpss.so.1，或者在运行程序之前使用等价选项运行 Solaris 9 命令 ppgsz(1)。详细内容请参见 Solaris 9 手册页。

注 – Solaris 7 和 Solaris 8 操作环境中不提供此功能。在 Solaris 7 和 Solaris 8 操作环境中，使用此选项编译的程序将不会链接。

A.3.112 -xparallel

(SPARC) 循环的并行化可由编译器自动完成也可由程序员显式指定。-xparallel 选项是一个宏，相当于指定 -xautopar、-xdepend 和 -xexplicitpar。如果显式并行化循环，则存在产生错误结果的风险。如果优化级别不是 -xO3 或更高级别，则将优化级别将提高到 -xO3 并发出警告。

如果您执行自己的线程管理，请不要使用 -xparallel。如果您发出的是 -xopenmp，则不要使用 -xparallel。如果指定 -xopenmp，则 -xparallel 设置不应使用的 -xexplicitpar。

要使代码的运行速度更快，则该选项需要多处理器系统。在单处理器系统上，生成的代码通常运行速度较慢。

如果以*单*步编译和链接，则 -xparallel 链接微任务库和线程安全 C 运行时库。
如果在*单独的*步骤中编译和链接，则使用 -xparallel 编译，然后使用 -xparallel 链接。

A.3.113 -xpch=*v*

该编译器选项可激活预编译的头文件功能。预编译的头文件功能用来减少应用程序的编译时间，这些应用程序的源代码共享同一组包含大量源代码的包含文件。编译器首先从一个源文件收集关于某个头文件序列的信息，然后在重新编译该源文件编译其它具有相同头文件序列的源文件时使用此信息。编译器收集的信息存储在预编译的头文件中。您可以通过 -xpch 和 -xpchstop 选项以及 #pragma hdrstop 指令使用此功能。

参见：

- 第 A-67 页上的 “-xpchstop=file”
- 第 2-13 页上的 “hdrstop”

A.3.113.1 创建预编译的头文件

当指定 -xpch=*v* 时，则 *v* 可以为 collect:*pch_filename* 或 use:*pch_filename*。第一次使用 -xpch 时，必须指定 collect 模式。指定 -xpch=collect 的编译命令必须只能指定一个源文件。在以下示例中， -xpch 选项创建一个基于源文件 a.c 的称为 myheader.cpch 的预编译的头文件：

```
cc -xpch=collect:myheader a.c
```

有效的预编译的头文件通常具有后缀 .cpch。当您指定 *pch_filename* 时，您可以增加后缀或让编译器增加。例如，如果您指定 cc -xpch=collect:foo a.c，则预编译的头文件称为 foo.cpch。

当您创建预编译的头文件时，选择一个源文件，该源文件包含与预编译的头文件一起使用的所有源文件的公共包含文件序列。在这些源文件中，公共包含文件序列必须相同。注意，在 collect 模式下只有一个源文件名值是合法的。例如， cc -xpch=collect:foo bar.c 有效，但是 cc -xpch=collect:foo bar.c foobar.c 无效，因为它指定了两个源文件。

A.3.113.2 使用预编译的头文件

指定 -xpch=use:*pch_filename* 以使用预编译的头文件。您可以指定任意数量的源文件，这些源文件的包含文件序列必须与用于创建预编译的头文件的序列相同。例如，在 use 模式中的命令可类似于以下命令：cc -xpch=use:foo.cpch foo.c bar.c foobar.c.

如果下列为真，则您应该只使用现有的预编译的头文件。如果下列任何一个为假，则您应该重新创建预编译的头文件：

- 您正用于访问预编译的头文件的编译器与创建预编译的头文件的编译器相同。某个版本编译器创建的预编译的头文件不能被其它版本使用。
- 除了 -xpch 选项，凡是用 -xpch=use 指定的编译器选项都要与创建预编译的头文件时指定的选项匹配。
- 使用 -xpch=use 指定的包含头文件的设置与创建预编译的头文件时指定的头文件的设置相同。
- 使用 -xpch=use 指定的包含头文件的内容与创建预编译的头文件时指定的包含头文件的内容相同。
- 当前目录 （即正在其中进行编译并尝试使用预编译的头文件的目录）与创建预编译的头文件的目录相同。
- 用 -xpch=collect 指定的文件中的预处理指令的初始序列 （包含 #include 指令）与用 -xpch=use 指定的文件中的预处理指令序列相同。

为在多个源文件之间共享预编译的头文件，这些源文件必须共享同一组包含文件并作为它们的初始标记序列。这个初始标记序列称为活动前缀。在所有使用同一预编译的头文件的源文件中，必须始终解释该活动前缀。

该活动前缀以每个源文件的首标记开头，以 #pragma hdrstop 或在 -xpchstop 选项中命名的头文件的 #include 指令的尾标记结束。

源文件的活动前缀只能由注释和以下预处理器指令组成：

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

所有这些指令都可能引用宏。在活动前缀内， #else、 #elif 和 #endif 指令必须匹配。

在共享预编译的头文件的每个文件的活动前缀内，每个相应的 #define 和 #undef 指令必须引用相同的符号（在使用 #define 的情况下，它们必须引用相同的值）。在每个活动前缀内，它们出现的顺序也必须相同。每个相应的 pragma 命令也必须相同，而且在所有共享预编译的头文件的文件之间按相同顺序出现。

合并到预编译的头文件的头文件不能违反以下约束。编译违反以下任何约束的程序的结果都是未定义的。

- 头文件不能包含函数定义。
- 头文件不能使用 __DATE__ 和 __TIME__。使用这些预处理器宏可能产生不可预测的结果。
- 头文件不能包含 #pragma hdrstop。
- 头文件不能在活动前缀中使用 __LINE__ 和 __FILE__。允许在包含的头文件中使用 __LINE__ 和 __FILE__。

A.3.113.3 如何修改 make 文件

将以下代码增加到 make 文件中，以将 -xpch 合并到所生成的代码中。下面提供了一种使用 make 和 dmake 的 KEEP_STATE 程序的简单方法。

```
CFLAGS=-xpch=use:shared
shared.cpch : bar.c
        cc -xpch=collect:shared bar.c -xe
bar.o : bar.c shared.cpch
        cc $(CFLAGS) bar.c -c
pong.o : pong.c shared.cpch
        cc $(CFLAGS) pong.c -c
```

A.3.114 -xpchstop=*file*

使用 -xpchstop=*file* 选项为使用 -xpch 选项创建的预编译的头文件指定活动前缀的最后一个包含文件。在命令行使用 -xpchstop 相当于将 hdrstop pragma 置于第一个包含指令之后，此包含指令在您使用 cc 命令指定的每个源文件中引用 *file*。

在下列示例中， -xpchstop 选项指定预编译的头文件的活动前缀的结尾包含 projectheader.h。因此， privateheader.h 不是活动前缀的一部分。

```
example% cat a.c
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% cc -xpch=collect:foo.cpch a.c -xpchstop=projectheader.h
-c
```

参见 -xpch。

A.3.115 -xpentium

优化 *(Intel)* Pentium 处理器。

A.3.116 -xpg

准备目标代码，以便为使用 gprof(1) 进行文件配置而收集数据。此选项调用在正常终止情况下产生 gmon.out 文件的运行时记录机制。

A.3.117 -xprefetch[=*val*[,*val*]]

(SPARC) 启用支持预取的体系结构上的预取指令，例如， UltraSPARC II。（-xarch=v8plus、 v9plusa、 v9 或 v9a）

只有在受度量支持的特殊情况下，才能使用显式预取。

val 必须是以下值之一：

表 A-29 `-xprefetch` 值

值	含义
`latx:factor`	通过指定的因子调整编译器假定的预取到装入和预取到存储的等待时间。参见第 A-68 页上的第 A.3.117.1 节 “预取等待时间比率”
`[no%]auto`	[禁用] 启用预取指令的自动生成
`[no%]explicit`	[禁用] 启用显式预取宏
`yes`	与 `-xprefetch=auto,explicit` 相同
`no`	与 `-xprefetch=no%auto,no%explicit` 相同

如果不指定 `-xprefetch`，则缺省值为 `-xprefetch=no%auto,explicit`。如果没有给 `-xprefetch` 指定一个值，则等价于 `-xprefetch=auto,explicit`。

`sun_prefetch.h` 头文件提供了可用于指定显式预取指令的宏。预取的位置大约在可执行程序中对应于宏出现的地方。

A.3.117.1 预取等待时间比率

预取等待时间是指从执行预取指令到被预取数据在缓存中可以使用之间的硬件延迟。

该因子必须是形式为 *n.n* 的正数。

在确定放置预取指令和使用预取数据的装入或存储指令的距离时，编译器假定一个预取等待时间。预取和装入之间的假定等待时间可能与预取和存储之间的假定等待时间不同。

为使多种机器和应用程序间的性能达到最佳，编译器将调节预取机制。此调节并非始终是最佳的。对于内存密集型应用程序，尤其是要在大型多处理器上运行的应用程序，通过增大预取等待时间值，您可以获得更好的性能。要增大此值，请使用大于 1 （一）的因子。介于 .5 和 2.0 之间的值最有可能提供最佳性能。

对于数据集完全存储在外部缓存内的应用程序，通过减小预取等待时间值，您可以获得更好的性能。要减小此值，请使用小于 1 的因子。

要使用 `latx:factor` 子选项，则以接近 1.0 的因子值开始并对应用程序进行性能测试。然后适当地增大或减小该因子值并重新进行性能测试。继续调整因子并进行性能测试，直到获得最佳性能为止。如果增大或减小的幅度很小，则在少数步骤中看不到性能差异，然后突然会有明显差异，随后重新稳定在新级别上。

A.3.118 -xprefetch_level=*l*

使用 -xprefetch_level 选项控制自动插入值 -xprefetch=auto 的预取指令的主动性。*l* 必须为 1、 2 或 3。编译器具有更强的主动性，换言之，使用更高级别的 -xprefetch_level 会引入更多的预取。

-xprefetch_level 的适当值取决于应用程序可能具有的缓存缺失的数量。较高级别的 -xprefetch_level 值具有提高应用程序性能的潜能。

仅当使用 -xprefetch=auto 进行编译且优化级别为 3 或更高级别时此选项才有效，并在支持预取（v8plus、v8plusa、v9、v9a、v9b、generic64、native64）的平台生成代码。

-xprefetch_level=1 启用自动生成预取指令。-xprefetch_level=2 启用高于级别 1 的附加生成，-xprefetch_level=3 启用高于级别 2 的附加生成。

当您指定 -xprefetch=auto 时，缺省值为 -xprefetch_level=1。

A.3.119 -xprofile=*p*

使用该选项收集并保存执行频率数据，从而可以在以后的运行中使用该数据以提高性能。仅当您将优化级别指定为 -xO2 或更高时，该选项才有效。

通过为编译器提供运行时性能反馈，可使用更高优化级别进行编译 （例如 -xO5）。为产生运行时性能反馈，您必须使用 -xprofile=collect 进行编译，再用典型的数据集运行可执行程序，然后在最高优化级别上使用 -xprofile=use 重新编译。

对于多线程的应用程序，配置文件收集是安全的。也就是说，对执行自己的多重任务处理的程序 (-mt) 进行文件配置可产生精确的结果。

p 必须为 collect[:*name*]、 use[:*name*] 或 tcov。

- collect[:*name*]

  优化器使用 -xprofile=use 收集并保存执行频率数据以备将来使用。编译器生成用以测量语句执行频率的代码。

  name 是正在进行分析的程序的名称。该名称是可选的。如果未指定 *name*，则假定 a.out 为可执行程序的名称。

  您可以设置环境变量 SUN_PROFDATA 和 SUN_PROFDATA_DIR，以便控制用 -xprofile=collect 编译的程序存储配置文件数据的位置。如果设置了该环境变量，则 -xprofile=collect 数据将写入到 SUN_PROFDATA_DIR/SUN_PROFDATA。

  这些环境变量同样控制 tcov 写入的配置文件数据文件的路径和名称， tcov(1) 手册页描述了此点。

如果未设置这些环境变量，则配置文件数据写入当前目录中的 *name*.profile/feedback，在该目录中 *name* 是可执行程序的名称或者是在 -xprofile=collect:*name* 标志中指定的名称。如果 *name* 的末尾已经有 .profile，则 -xprofile 不会将 .profile 附加到 *name* 后面。如果您多次运行程序，则执行频率数据积累在 feedback 文件中，也就是说，以面执行的输出不会丢失。

如果您分别在单独的步骤中进行编译和链接，请确保用 -xprofile=collect 编译的所有目标文件也与 -xprofile=collect 链接。

- use[:*name*]

  程序通过以下方式进行优化：使用上次执行使用 -xprofile=collect 编译的程序时生成并保存在 feedback 文件中的执行频率数据。

  name 是正在进行分析的程序的名称。该名称是可选的。如果未指定 *name*，则假定 a.out 为可执行程序的名称。

  除可从 -xprofile=collect 更改为 -xprofile=use 的 -xprofile 选项外，源文件和其它编译器选项必须与用于编译的源文件和编译器选项严格一致，此编译创建依次生成 feedback 文件的已编译程序。收集生成和使用生成所用的编译器版本必须相同。如果使用 -xprofile=collect:*name* 进行编译，则相同的程序名称 *name* 必须出现在优化编译：-xprofile=use:*name* 中。

  在使用 -xprofile=collect 编译目标文件时，目标文件和其配置文件数据之间的关联是基于目标文件的 UNIX 路径名。在某些情况下，编译器将不会联合目标文件和其配置文件数据：由于以前没有使用 -xprofile=collect 编译目标文件，因此目标文件没有配置文件数据，程序中的目标文件未使用 -xprofile=collect 进行链接，从未执行过该程序。

  如果以前使用 -xprofile=collect 在不同目录中编译目标文件，而且该目标文件与其它使用 -xprofile=collect 编译的目标文件共享一个通用基名，但它们的包含目录名无法唯一识别它们，则编译器也会因此变得混乱。在此情况下，即使目标文件具有配置文件数据，当使用 -xprofile=use 重新编译目标文件时，编译器也无法在反馈目录中找到它。

  所有这些情况都会导致编译器丢失目标文件和其配置文件数据之间的关联。因此，如果目标文件具有配置文件数据，但是当您指定 -xprofile=use 时编译器无法将目标文件与其路径名联合，请使用 -xprofile_pathmap 选项来识别正确的目录。参见第 A-72 页上的 “-xprofile_pathmap”。

- tcov

  使用 “新”式样 tcov 的基本块覆盖分析。

  -xprofile=tcov 选项是 tcov 的基本块文件配置的新式样。其功能性与 -xa 选项相似，此外还能正确地为在头文件中具有源代码的程序收集数据。请参见第 A-28 页上的第 A.3.62 节 “-xa”，以了解文件配置的旧式样信息。有关更多详细信息，请参见 tcov(1) 手册页和 《*程序性能分析工具*》。

执行代码指令与执行 -xa 选项相似，但是不再生成 .d 文件。而生成单个文件，该文件名称基于最终可执行程序。例如，如果程序用完了 /foo/bar/myprog.profile，则数据文件存储在 /foo/bar/myprog.profile/myprog.tcovd。

-xprofile=tcov 和 -xa 选项在单个可执行程序中兼容。也就是说，您可以链接这些程序，其中部分文件已用 -xprofile=tcov 编译，而其它文件是用 -xa 编译的。您不能同时使用这两个选项编译单个文件。

当运行 tcov 时，您必须将 -x 选项传递给它，以使它使用新式样数据。否则，tcov 使用旧式 .d 文件 （如果有），这是数据的缺省值，并产生不可预测的输出。

与 -xa 选项不同，在编译时 TCOVDIR 环境变量没有任何作用。但是，在程序运行时使用其值。有关详细信息，请参见 tcov(1) 和 《*程序性能分析工具*》。

注 – 如果由于 -xO4 或 -xinline 存在例程的内联处理，则 tcov 的代码覆盖报告可能不可靠。

当您使用 -xprofile=collect 来编译用于配置文件收集的程序，而用 -xprofile=use 来编译用于配置文件反馈的程序时，在这两个编译中，除 -xprofile=collect 和 -xprofile=use 之外的源文件和编译器选项必须相同。

使用 -xprofile=use:*name* 选项指定的编译反馈目录名从此选项在编译器单个调用中的所有例程开始累积。例如，假定分别执行名称为 a、 b 和 c 的已进行文件配置的二进制文件结果是创建文件配置目录 a.profile、 b.profile 和 c.profile。

```
cc -O -c foo.c -xprofile=use:a -xprofile=use:b -xprofile=use:c
```

这三个配置文件目录均被使用。编译目标文件时，与特定目标文件有关的任何有效配置文件反馈数据从指定的反馈目录中开始积累。

如果在同一的命令行上同时指定 -xprofile=collect 和 -xprofile=use，则命令行中最右边 -xprofile 选项的应用如下所示：

- 如果最右边 -xprofile 选项为 -xprofile=use，则 -xprofile=use 选项指定的所有配置文件反馈目录名将用于定向反馈优化，同时忽略以前的 -xprofile=collect 选项。
- 如果最右边 -xprofile 选项为 -xprofile=collect，则忽略 -xprofile=use 选项指定的所有配置文件反馈目录名，同时启用配置文件生成的指令。

参见

-xhwcprof, -xprofile_ircache, -xprofile_pathmap

A.3.120 -xprofile_ircache[=*path*]

使用 -xprofile_ircache[=*path*] 及 -xprofile=collect|use，以通过重新使用在 collect 阶段保存的编译数据来减少 use 阶段的编译时间。

对于大程序，由于保存了中间数据，因此 use 阶段的编译时间可显著减少。注意，保存数据将明显增加磁盘空间需求。

使用 -xprofile_ircache[=*path*] 时， *path* 覆盖缓存的文件的存储位置。缺省情况下，这些文件保存与目标文件相同的目录中。当 collect 和 use 阶段发生在不同的目录中时，指定路径将非常有用。下面是典型的命令序列：

```
example% cc -xO5 -xprofile=collect -xprofile_ircache t1.c t2.c
example% a.out     // 运行收集反馈数据
example% cc -xO5 -xprofile=use -xprofile_ircache t1.c t2.c
```

A.3.121 -xprofile_pathmap

当您也指定 -xprofile=use 命令时，请使用 -xprofile_pathmap=*collect_prefix*:*use_prefix* 选项。当下列两个条件都为真而且编译器无法找到用 -xprofile=use 编译的目标文件的配置文件数据时，请使用 -xprofile_pathmap。

- 您使用 -xprofile=use 编译某个目录中的目标文件，该目录与以前用 -xprofile=collect 编译的目标文件所在目录不同。
- 目标文件共享配置文件中的通用基名，但可通过它们在不同目录中的位置可区分开来。

collect-prefix 是目录树中的 UNIX 路径名的前缀，在该目录树中目标文件是使用 -xprofile=collect 编译的。

use-prefix 是目录树中的 UNIX 路径名的前缀，在该目录树中目标文件是使用 -xprofile=use 编译的。

如果您指定多个 -xprofile_pathmap 实例，则编译器按它们出现的顺序进行处理。-xprofile_pathmap 指定的每个 *use-prefix* 均要与目标文件路径名进行比较，直到识别出一个匹配的 *use-prefix* 或找到最后一个与目标文件路径名不匹配的 *use-prefix* 为止。

A.3.122 -xreduction

(SPARC) 在自动并行化期间启用约简识别。必须用 -xautopar 或 -xparallel 指定 -xreduction。

当启用约简识别时，编译器并行化约简，例如 *dot* 产品、最大与最小查找。这些约简产生的舍入与通过非并行化代码获得的舍入不同。

A.3.123 -xregs=*r*[,*r*...]

(SPARC) 为生成的代码指定寄存器的用法。

r 是一个以逗号分隔的列表，该列表由以下各项中的一个或多个组成：[no%]appl、[no%]float。

例如：-xregs=appl,no%float

表 A-30 -xregs 值

值	含义
[no%]appl	[不] 允许编译器通过将应用程序寄存器用作临时寄存器来生成代码。这些应用程序寄存器为： g2, g3, g4 (v8a, v8, v8plus, v8plusa, v8plusb) g2, g3 (v9, v9a, v9b) 强烈推荐您使用 -xreg=no%appl 编译所有系统软件和库。系统软件 （包括共享库）必须为应用程序保存这些寄存器的值。它们的使用将要受到编译系统控制，并且必须在整个应用程序中始终保持一致。 有关 SPARC 指令集的详细信息，请参见第 A-29 页上的 “-xarch=isa”。 在 SPARC ABI 中，这些寄存器被描述为*应用程序*寄存器。由于需要较少装入和存储指令，因此使用这些寄存器可提高性能。但是，此类使用可能与某些用汇编代码编写的旧库程序冲突。
[no%]float	[不] 允许编译器通过将浮点寄存器用作存储整数值的临时寄存器来生成代码。无论是否使用此选项，使用浮点值都可能使用这些寄存器。如果 -xregs=no%float，则源程序不能包含任何浮点代码。

缺省值为 -xregs=appl,float。

强烈推荐您用 -xregs=no%appl,float 编译代码，该代码将用于与应用程序链接的共享库。至少，共享库应该显式说明它如何使用应用程序寄存器，从而用那些库链接的应用程序寄存器知道如何处理该问题。

例如，在某些全局检测中使用寄存器的应用程序 （例如，使用寄存器指向一些临界数据结构）将需要确切地知道带有未使用 -xregs=no%appl 编译的代码的库如何使用应用程序寄存器以便安全链接该库。

A.3.124 -xrestrict[=*f*]

(SPARC) 将赋值为指针的函数参数视为受限指针。*f* 是 %all、%none 或一个用逗号隔开的一个或多个函数名称序列：{%all|%none|*fn*[,*fn*...]}。

如果使用该选项指定函数列表，则指定的函数中的指针参数将被视为限定的；如果指定 -xrestrict=%all，则整个 C 文件中的所有指针参数均被视为限定的。有关详细信息，请参见第 3-19 页上的第 3.8.2 节 “限定指针”。

该命令行选项可用于其自身，但最佳效果是与优化同时使用。例如，命令：

```
%cc -xO3 -xrestrict=%all prog.c
```

将文件 prog.c 中的所有指针参数视为限定指针。命令：

```
%cc -xO3 -xrestrict=agc prog.c
```

将文件 prog.c 中函数 agc 中的所有指针参数视为限定指针。

缺省值为 %none；指定 -xrestrict 相当于指定 -xrestrict=%all。

A.3.125 -xs

允许在没有目标文件的情况下使用 dbx 进行调试。

该选项导致所有调试信息被复制到可执行程序中。这对 dbx 的性能或程序的运行时性能影响较小，但需要更多磁盘空间。

A.3.126 -xsafe=mem

(SPARC) 允许编译器假定不会发生基于内存的陷阱。

该选项允许在 V9 机器上使用推测装入指令。仅当您指定优化级别为 -xO5 且 -xarch=v8plus|v8plusa|v9|v9a 时，此选项才有效。

注 – 由于在发生诸如地址未对齐或段违规的故障时，非故障装入不会导致陷阱，因此您应该只对不会发生此类故障的程序使用该选项。由于很少程序会导致基于内存的陷阱，因此您可对大多数程序安全地使用该选项。对于显式依赖基于内存的陷阱来处理异常情况的程序，请勿使用该选项。

A.3.127 -xsb

使用该选项为源代码浏览器生成额外的符号表信息。该选项在编译器的 -Xs 模式下无效。

如果您分别在单独的步骤中进行编译和链接，请确保在编译步骤和链接步骤中均指定 -xsb，否则您将看到来自连接程序的错误消息。

如果您未使用 -xsb 来链接使用 -xsb 编译的对象，则会将源代码浏览器数据限制于那些用于在链接步骤中创建的可执行程序的引用。另外，如果您在单独的编译和链接步骤中未指定 -xsb，则可能会丢失源代码浏览器数据库中的一些符号引用。

通过在单独的编译和链接步骤中包含 -xsb，当对象在相同目录中用不同的方法编译并与不同的可执行程序链接时，您可以确保这两个对象中的所有符号引用对源代码浏览器是可见的。

A.3.128 -xsbfast

为源代码浏览器创建数据库。不会将源代码编译到目标文件中。该选项在编译器的 -Xs 模式下无效。

A.3.129 -xsfpconst

将没有后缀的浮点常量表示为单精度，而不是双精度的缺省模式。使用 -Xc 无效。

A.3.130 -xspace

不优化或并行化增加代码大小的循环。

例如：如果编译器增加代码大小，则它不会解开循环或并行化循环。

A.3.131 -xstrconst

在文本段（而非缺省数据段）的只读数据部分插入字符串文字。删除重复的字符串并在代码的引用间共享剩余副本。

A.3.132 -xtarget=*t*

为指令集和优化指定目标系统。

t 的值必须是下列值之一：native、 generic、 *system-name (SPARC, x86)*。

-xtarget 选项是一个宏，它允许发生在真实系统上的 -xarch、-xchip 和 -xcache 的组合的快捷易行的规范。-xtarget 的唯一含义在其扩展中。

表 A-31 -xtarget 值

值	含义
native	在主机系统上获得最佳性能。 编译器生成能在主机系统上提供最佳性能的代码。该选项决定编译器所运行的机器上的可用体系结构、芯片和缓存属性。
generic	获取通用体系结构、芯片和缓存的最佳性能。 编译器将 -xtarget=generic 扩展为： -xarch=generic -xchip=generic -xcache=generic 这是缺省值。
system-name	获取指定系统的最佳性能。 表 A-32 列出了真实系统名称和编号的助记符编码，您可以从中选择一个系统名称。

通过为编译器提供目标计算机硬件的精确描述，某些程序的性能可得到改善。当程序性能至关重要时，目标硬件的严格规范很重要。在较新的 SPARC 处理器上运行时尤其如此。但是，对于大多数程序和较旧的 SPARC 处理器，性能增效甚微，通用规范已足够。

-xtarget 的每个特定值都扩展为 -xarch、-xchip 和 -xcache 选项的特定值集。使用 fpversion(1) 命令确定 -xtarget=native 在所运行的系统上的扩展。参见表 A-32 以获取扩展的值。

例如，-xtarget=sun4/15 等价于：-xarch=v8a -xchip=micro -xcache=2/16/1。

注 – 在特定的主机平台上编译时， -xtarget 在该平台上的扩展可能导致 -xarch、-xchip 或 -xcache 设置与 -xtarget=native 时的设置不同。

表 A-32 -xtarget 在 SPARC 上的扩展

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
cs6400	v8	super	16/32/4:2048/64/1
entr150	v8	ultra	16/32/1:512/64/1
entr2	v8plusa	ultra	16/32/1:512/64/1
entr2/1170	v8plusa	ultra	16/32/1:512/64/1
entr2/1200	v8plusa	ultra	16/32/1:512/64/1
entr2/2170	v8plusa	ultra	16/32/1:512/64/1
entr2/2200	v8plusa	ultra	16/32/1:512/64/1
entr3000	v8plusa	ultra	16/32/1:512/64/1
entr4000	v8plusa	ultra	16/32/1:512/64/1
entr5000	v8plusa	ultra	16/32/1:512/64/1
entr6000	v8plusa	ultra	16/32/1:512/64/1
sc2000	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1

表 **A-32** -xtarget 在 SPARC 上的扩展 *（续）*

-xtarget=	-xarch	-xchip	-xcache
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss2p	v7	powerup	64/32/1

表 A-32 -xtarget 在 SPARC 上的扩展 *（续）*

-xtarget=	-xarch	-xchip	-xcache
ss4	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1

表 **A-32** -xtarget 在 SPARC 上的扩展 *（续）*

-xtarget=	-xarch	-xchip	-xcache
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/630	v7	old	64/32/1
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
ultra	v8plusa	ultra	16/32/1:512/64/1
ultra1/140	v8plusa	ultra	16/32/1:512/64/1
ultra1/170	v8plusa	ultra	16/32/1:512/64/1
ultra1/200	v8plusa	ultra	16/32/1:512/64/1
ultra2	v8plusa	ultra2	16/32/1:512/64/1
ultra2/1170	v8plusa	ultra	16/32/1:512/64/1
ultra2/1200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/1300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2/2170	v8plusa	ultra	16/32/1:512/64/1
ultra2/2200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/2300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4

表 A-32 -xtarget 在 SPARC 上的扩展 *（续）*

-xtarget=	-xarch	-xchip	-xcache
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	v8plusa	ultra3	64/32/4:8192/512/1
ultra3cu	v8plusa	ultra3cu	64/32/4:8192/512/2

下表列出了 Intel 体系结构的 -xtarget 值：

表 A-33 -xtarget 在 Intel 体系结构上的扩展

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
386	386	386	generic
486	386	486	generic
pentium	386	pentium	generic
pentium_pro	pentium_pro	pentium_pro	generic

A.3.133 -xtemp=*dir*

将 cc 使用的临时文件的目录设置为 *dir*。在该选项字符串内禁用空格。如果不使用该选项，则临时文件进入 /tmp。 -xtemp 优先于 TMPDIR 环境变量。

A.3.134 -xthreadvar[=*o*]

(SPARC) 指定 -xthreadvar 以控制线程局部变量的实现。将该选项与 __thread 声明说明符配合使用，以便利用编译器的线程局部存储功能。用 __thread 说明符声明线程变量之后，指定 -xthreadvar 以在 （共享）库中使用带有与位置相关的代码 （非 PIC 代码）的线程局部存储。有关如何使用 __thread 的详细信息，参见第 2-4 页上的“线程本地存储说明符”。

o 必须为下列值之一：

表 A-34 -xthreadvar 值

r 的值	含义
[no%]dynamic	[不] 编译动态装入的变量。当 -xthreadvar=no%dynamic 时，访问线程变量的速度更快，但您不能使用动态库中的目标文件。也就是说，您无法使用可执行文件中的目标文件。

如果您未指定 -xthreadvar，则编译器使用的缺省值取决于是否启用了与位置无关的代码。如果启用了与位置无关的代码，则将该选项设置为 -xthreadvar=dynamic。如果禁用了与位置无关的代码，则将该选项设置为 -xthreadvar=no%dynamic。

如果指定 -xthreadvar，但未指定任何值，则该选项设置为 -xthreadvar=dynamic。

如果动态库中存在非位置无关代码，则您必须指定 -xthreadvar。

链接程序不支持与动态库中的非 PIC 的线程变量等价的线程变量。非 PIC 线程变量的速度更快，因此它应该是可执行程序的缺省值。

在不同版本的 Solaris 软件上使用线程变量需要在命令行上使用不同选项。

- 对于 Solaris 8 软件，使用 __thread 的对象必须用 -mt 进行编译且与 -mt -L/usr/lib/lwp -R/usr/lib/lwp 链接。
- 对于 Solaris 9 软件，使用 __thread 的对象必须用 -mt 进行编译和链接。

参见：-xcode, -KPIC, -Kpic

A.3.135 -xtime

报告每个编译组件使用的时间和资源。

A.3.136 -xtransition

针对 K&R C 和 Sun ISO C 之间的差异发出警告。

-xtransition 选项与 -Xa 和 -Xt 选项一起发出警告。通过适当编码，您可以删除关于不同行为的所有警告消息。除非您使用 -xtransition 选项，否则不会再出现以下警告：

- \a 是 ISO C “报警”字符
- \x 是 ISO C 十六进制换码
- 八进制数错误
- 基本类型是真正 *type tag*: *name*
- 注释被“##“ 替换
- 注释没有并置标记
- 在 ISO C 中引入了新类型：*type tag*
- 在字符常量内进行了宏替换
- 在字符串文字内进行了宏替换
- 在字符常量内不能进行宏替换
- 在字符串文字内不能进行宏替换
- 操作数被视为无符号类型
- 三字母序列被替换
- ISO C 将常量视为无符号类型：*operator*
- *operator* 的语义在 ISO C 中发生变化；应使用显式强制类型转换

A.3.137 -xtrigraphs

-xtrigraphs 选项确定编译器是否识别 ISO C 标准定义的三字母序列。

在缺省情况下，编译器假定 -xtrigraphs=yes 并识别整个编译单元的所有三字母序列。

如果源代码具有包含疑问号 (?) 的文字字符串，并且编译器将该字符串解释为三字母序列，则您可以使用 -xtrigraph=no 子选项来关闭三字母序列识别。
-xtrigraphs=no 选项可关闭整个编译单元内的所有三字母识别。

考虑下列名为 trigraphs_demo.c 的示例源文件。

```
#include <stdio.h>

int main ()
{

  (void) printf("(\?\?) in a string appears as (??)\n");

  return 0;
}
```

如果用 -xtrigraphs=yes 编译此代码，则以下为其输出结果。

```
example% cc -xtrigraphs=yes trigraphs_demo.c
example% a.out
(??) in a string appears as (]
```

如果用 -xtrigraphs=no 编译此代码，则以下为其输出结果。

```
example% cc -xtrigraphs=no trigraphs_demo.c
example% a.out
(??) in a string appears as (??)
```

A.3.138 -xunroll=*n*

建议优化器解开循环 *n* 次。*n* 是一个正整数。当 *n* 等于 1 时，它是一个命令，此时编译器不解开任何循环。当 *n* 大于 1 时，-xunroll=*n* 只建议编译器解开循环 *n* 次。

A.3.139 -xustr={ascii_utf16_ushort|no}

如果代码中包含您希望在目标文件中被编译器转换成 UTF-16 字符串的字符串文字，则使用该选项。此选项使系统能够将 U"*ASCII_string*" 字符串文字识别为无符号短类型数组。

缺省值为 -xustr=no。-xustr=ascii_utf16_ushort 允许编译器识别 U"*ASCII_string*" 字符串文字。

A.3.140 -xvector[={yes|no}]

启用对向量库函数调用的自动生成。

如果循环内的数学库调用可转换为对等价向量数学例程的单个调用，则 -xvector=yes 允许编译器进行此类转换。此类转换可提高那些循环计数较大的循环的性能。

如果您未指定 -xvector，则缺省值为 -xvector=no。-xvector=no 撤消以前指定的 -xvector=yes。如果您指定 -xvector，但没有提供值，则缺省值为 -xvector=yes。

如果在以前没有指定 -xdepend 的情况下在命令行上使用 -xvector，则 -xvector 将会触发 -xdepend。如果未指定优化级别或优化级别低于 -x03，则 -xvector 选项还会将优化级别提高到 -x03。

在装入步骤中，编译器包含 libmvec 库。如果您使用单独的命令进行编译和链接，请确保在链接 cc 命令中使用 -xvector。

A.3.141 -xvis

(SPARC) 如果您正使用的是在 VIS[tm] 指令集软件开发者工具包 (VSDK) 中定义的汇编语言模板，请使用 -xvis=[yes|no] 命令。缺省值为 -xvis=no。指定 -xvis 相当于指定 -xvis=yes。

VIS 指令集是对 SPARC v9 指令集的扩展。尽管 UltraSPARC 处理器是 64 位，但在很多情况下数据长度仅限于 8 位或 16 位，特别是在多媒体应用程序中。VIS 指令可以用一条指令处理 4 个 16 位数据，因此极大地提高了处理新媒体的应用程序的性能，例如成像、线性代数、信号处理、音频、视频及联网。

有关 VSDK 的详细信息，请访问 http://www.sun.com/processors/vis。

A.3.142 -xvpara

(SPARC) 当可能未正确指定要并行化的循环时，使用该选项可针对指定了 #pragma MP 指令的循环发出警告。例如，当优化器检测到循环迭代间的数据依赖性时，它将发出一则警告。

有关 -xvpara 与 -xexplicitpar 选项或 -xparallel 选项配合使用以及 #pragma MP 的详细信息，请参见第 3-20 页上的第 3.8.3 节 “显式并行化和 Pragma”。

A.3.143 -Y*c*, *dir*

为组件 *c* 的位置指定一个 新目录 *dir*。 *c* 可包括在 -W 选项下列出的表示组件的任何字符。

如果已指定组件的位置，则工具的新路径名称为 *dir/tool*。如果对任何一项应用了多个 -Y 选项，则保留最后一个选项。

A.3.144 -YA, *dir*

更改查找组件的缺省目录。

A.3.145 -YI, *dir*

更改搜索 include 文件的缺省目录。

A.3.146 -YP, *dir*

更改查找库文件的缺省目录。

A.3.147 -YS, *dir*

更改启动目标文件的缺省目录。

A.3.148 -Zll

(SPARC) 为 lock_lint 创建程序数据库，但不生成可执行代码。有关详细信息，请参见 lock_lint(1) 手册页。

A.4 传递给链接程序的选项

cc 识别 -a、-e、-r、-t、-u 和 -z，并将这些选项及它们的参数传递给 ld。cc 也会将所有未识别选项传递给 ld，同时会发出警告。

附录 B

ISO C 数据表示法

本附录描述 ISO C 如何表示存储器中的数据以及向函数传递参数的机制。其目的是为想要编写或使用非 C 语言模块并将这些模块与 C 代码连接的程序员提供指导。

B.1 存储分配

下表显示数据类型及其表示方式。

注 – 栈上分配的存储器 （具有内部链接或自动链接的标识符）应限于 2 千兆字节或更小。

表 B-1 数据类型的存储分配

数据类型	内部表示
`char` 元素	在字节边界上对齐的单个 8 位字节。
`short` 整数	半字 （2 个字节或 16 位），在双字节边界上对齐
`int`	32 位 （4 个字节或 1 个字），在 4 字节边界上对齐
`long`	在 v8 和 Intel 上为 32 位 （4 个字节或 1 个字），在 4 字节边界上对齐 在 v9 上为 64 位 （8 个字节或 2 个字），在 8 字节边界上对齐
`pointer`	在 v8 和 Intel 上为 32 位 （4 个字节或 1 个字），在 4 字节边界上对齐 在 v9 上为 64 位 （8 个字节或 2 个字），在 8 字节边界上对齐
`long long`[1]	*(SPARC)* 64 位 （8 个字节或 2 个字），在 8 字节边界上对齐 *(Intel)* 64 位 （8 个字节或 2 个字），在 4 字节边界上对齐

表 **B-1** 数据类型的存储分配 *（续）*

数据类型	内部表示
`float`	32 位（4 个字节或 1 个字），在 4 字节边界上对齐。一个 `float` 包含一个 `sign` 位、 8 位指数和 23 位小数。
`double`	64 位 （8 个字节或 2 个字），在 8 字节边界上对齐 *(SPARC)* 或者在 4 字节边界上对齐 *(Intel)*。一个 `double` 元素包含一个 `sign` 位、一个 11 位指数和一个 52 位小数。
`long double`	v8 *(SPARC)* 128 位（16 个字节或 4 个字），在 8 字节边界上对齐。一个 `long double` 元素包含一个 `sign` 位、一个 15 位指数和一个 112 位小数。 v9 *(SPARC)* 128 位 （16 个字节或 4 个字），在 16 字节边界上对齐。一个 `long double` 元素包含一个 `sign` 位、一个 15 位指数和一个 112 位小数。 *(Intel)* 96 位 （12 个字节或 3 个机器字），在 4 字节边界上对齐。一个 `long double` 元素包含一个 `sign` 位、一个 16 位指数和一个 64 位小数。 16 位未使用。

1 如果设置 `-xc99=%none`， `long long` 在 `-Xc` 模式下不可用。

B.2 数据表示法

任何给定数据元素的位编号取决于使用的体系结构：SPARCstation™ 机器将位 0 用作最低有效位，将字节 0 用作最高有效字节。本节中的表描述各种表示法。

B.2.1 整数表示法

ISO C 中使用的整数类型是 `short`、 `int`、 `long` 和 `long long`：

表 **B-2** `short` 的表示

位	内容
8 - 15	字节 0 *(SPARC)* 字节 1 *(Intel)*
0 - 7	字节 1 *(SPARC)* 字节 0 *(Intel)*

表 B-3 `int` 的表示

位	内容
24 - 31	字节 0 *(SPARC)* 字节 3 *(Intel)*
16 - 23	字节 1 *(SPARC)* 字节 2 *(Intel)*
8 - 15	字节 2 *(SPARC)* 字节 1 *(Intel)*
0 - 7	字节 3 *(SPARC)* 字节 0 *(Intel)*

表 B-4 在 Intel 和 SPARC v8 与 SPARC v9 上 `long` 的表示

位	内容
24 - 31	字节 0 *(SPARC)* v8 字节 4 *(SPARC)* v9 字节 3 *(Intel)*
16 - 23	字节 1 *(SPARC)* v8 字节 5 *(SPARC)* v9 字节 2 *(Intel)*
8 - 15	字节 2 *(SPARC)* v8 字节 6 *(SPARC)* v9 字节 1 *(Intel)*
0 - 7	字节 3 *(SPARC)* v8 字节 7 *(SPARC)* v9 字节 0 *(Intel)*

表 **B-5** `long long`[1] 的表示

位	内容
56 - 63	字节 0 *(SPARC)* 字节 7 *(Intel)*
48 - 55	字节 1 *(SPARC)* 字节 6 *(Intel)*
40 - 47	字节 2 *(SPARC)* 字节 5 *(Intel)*
32 - 39	字节 3 *(SPARC)* 字节 4 *(Intel)*
24 - 31	字节 4 *(SPARC)* 字节 3 *(Intel)*
16 - 23	字节 5 *(SPARC)* 字节 2 *(Intel)*
8 - 15	字节 6 *(SPARC)* 字节 1 *(Intel)*
0 - 7	字节 7 *(SPARC)* 字节 0 *(Intel)*

1 long long 在 `-Xc` 模式下不可用。

B.2.2 浮点表示法

`float`、`double` 和 `long double` 数据元素按照 ISO IEEE 754-1985 标准表示。表示为：

$(-1)^{s(e-bias)} \times 2\ j.f$

其中：

- *s* = `sign`
- *e* = 增阶码
- *j* 为前导位，由 *e* 的值决定。在 `long double` *(Intel)* 的情况下，前导位是显式的；在所有其它情况下，它是隐式的。
- *f* = 小数
- *u* 表示位可以为 0 或 1。

下表显示各个位的位置。

表 B-6 `float` 表示

位	名称
31	符号
23 - 30	指数
0 - 22	小数

表 B-7 `double` 表示

位	名称
63	符号
52 - 62	指数
0 - 51	小数

表 B-8 `long double` 表示 *(SPARC)*

位	名称
127	符号
112 - 126	指数
0 - 111	小数

表 B-9 `long double` 表示 *(Intel)*

位	名称
80 - 95	未使用
79	符号
64 - 78	指数
63	前导位
0 - 62	小数

有关更详细的信息，请参见《*数值计算指南*》。

B.2.3 异常值

`float` 和 `double` 数被认为包含一个 “隐藏的”或隐含的位，从而比不包含该位时的精度高一位。在 `long double` 的情况下，前导位为隐式 *(SPARC)* 或显式 *(Intel)*；该位对于正规数为 1，对于次正规数为 0。

表 B-10 `float` 表示

正规数 (0<e<255):	$(-1)^{Sign}2^{(exponent - 127)}1.f$
次正规数 (e=0, f!=0):	$(-1)^{Sign}2^{(-126)}0.f$
零 (e=0, f=0):	$(-1)^{Sign}0.0$
信号 NaN	s=u， e=255(max)； f=.0uuu-uu；至少一个位必须为非零
无噪声 NaN	s=u， e=255(max)； f=.1uuu-uu
无穷	s=u， e=255(max)； f=.0000-00 （全为零）

表 B-11 `double` 表示

正规数 (0<e<2047):	$(-1)^{Sign}2^{(exponent - 1023)}1.f$
次正规数 (e=0, f!=0):	$(-1)^{Sign}2^{(-1022)}0.f$
零 (e=0, f=0):	$(-1)^{Sign}0.0$
信号 NaN	s=u， e=2047(max)； f=.0uuu-uu；至少一个位必须为非零
无噪声 NaN	s=u， e=2047(max)； f=.1uuu-uu
无穷	s=u， e=2047(max)； f=.0000-00 （全为零）

表 B-12 `long double` 表示

正规数 (0<e<32767):	$(-1)^{Sign}2^{(exponent - 16383)}1.f$
次正规数 (e=0, f!=0):	$(-1)^{Sign}2^{(-16382)}0.f$
零 (e=0, f=0):	$(-1)^{Sign}0.0$
信号 NaN	s=u， e=32767(max)； f=.0uuu-uu；至少一个位必须为非零
无噪声 NaN	s=u， e=32767(max)； f=.1uuu-uu
无穷	s=u， e=32767(max)； f=.0000-00 （全为零）

B.2.4 选定的数的十六进制表示

下表显示十六进制表示。

表 B-13 选定的数的十六进制表示 *(SPARC)*

值	float	double	long double
+0	00000000	0000000000000000	00000000000000000000000000000000
-0	80000000	8000000000000000	80000000000000000000000000000000
+1.0	3F800000	3FF0000000000000	3FFF0000000000000000000000000000
-1.0	BF800000	BFF0000000000000	BFFF0000000000000000000000000000
+2.0	40000000	4000000000000000	40000000000000000000000000000000
+3.0	40400000	4008000000000000	40080000000000000000000000000000
正无穷	7F800000	7FF0000000000000	7FFF0000000000000000000000000000
负无穷	FF800000	FFF0000000000000	FFFF0000000000000000000000000000
NaN	7FBFFFFF	7FF7FFFFFFFFFFFF	7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF

表 B-14 选定的数的十六进制的表示 *(Intel)*

值	float	double	long double
+0	00000000	0000000000000000	00000000000000000000
-0	80000000	0000000080000000	80000000000000000000
+1.0	3F800000	000000003FF00000	3FFF8000000000000000
-1.0	BF800000	00000000BFF00000	BFFF8000000000000000
+2.0	40000000	0000000040000000	40008000000000000000
+3.0	40400000	0000000040080000	4000C000000000000000
正无穷	7F800000	000000007FF00000	7FFF8000000000000000
负无穷	FF800000	00000000FFF00000	FFFF8000000000000000
NaN	7FBFFFFF	FFFFFFFF7FF7FFFF	7FFFBFFFFFFFFFFFFFFF

有关更详细的信息，请参见《*数值计算指南*》。

B.2.5 指针表示

C 中的一个指针占 4 个字节。 C 中的一个指针在 SPARC v9 体系结构中占 8 个字节。 `NULL` 值指针等于零。

B.2.6 数组存储

数组及其元素按特定的存储顺序存储。元素实际上按存储元素的线性序存储。

C 数组按行优先顺序存储；多维数组中最后一个下标变化最快。

字符串数据类型是 `char` 元素的数组。字符串文字或宽字符串文字 （并置后）中允许的字符数最大值为 4,294,967,295。

有关栈上分配的存储大小限制的信息，请参见第 B-1 页上的第 B.1 节 “存储分配”。

表 B-15 数组类型和存储

类型	**SPARC** 和 **Intel** 上的元素数最大值	**SPARC V9** 上的元素数最大值
`char`	4,294,967,295	2,305,843,009,213,693,951
`short`	2,147,483,647	1,152,921,504,606,846,975
`int`	1,073,741,823	576,460,752,303,423,487
`long`	1,073,741,823	288,230,376,151,711,743
`float`	1,073,741,823	576,460,752,303,423,487
`double`	536,870,911	288,230,376,151,711,743
`long double`	268,435,451	144,115,188,075,855,871
`long long`[1]	536,870,911	288,230,376,151,711,743

1 当设置 `-xc99=%none` 时在 -Xc 模式下无效。

静态数据和全局数组可以容纳更多元素。

B.2.7 异常值的算术运算

本节描述对异常值和普通浮点值的组合应用基本算术运算得到的结果。以下信息假定不执行陷阱或任何其它异常操作。

下表解释缩写：

表 B-16 缩写用法

缩写	含义
Num	次正规数或正规数
Inf	无穷 （正或负）
NaN	不是数
Uno	无序

下表描述对不同类型的操作数的组合执行算术运算所得值的类型。

表 B-17 加法和减法结果

	右操作数：**0**	右操作数：**Num**	右操作数：**Inf**	右操作数：**NaN**
左操作数：**0**	0	Num	Inf	NaN
左操作数：**Num**	Num	参见[1]	Inf	NaN
左操作数：**Inf**	Inf	Inf	参见 [1]	NaN
左操作数：**NaN**	NaN	NaN	NaN	NaN

1 结果太大 （溢出）时， Num + Num 可能为 Inf 而不是 Num。无穷值具有相反的 `sign` 时， Inf + Inf = NaN。

表 B-18 乘法结果

	右操作数：**0**	右操作数：**Num**	右操作数：**Inf**	右操作数：**NaN**
左操作数：**0**	0	0	NaN	NaN
左操作数：**Num**	0	Num	Inf	NaN
左操作数：**Inf**	NaN	Inf	Inf	NaN
左操作数：**NaN**	NaN	NaN	NaN	NaN

表 **B-19** 除法结果

	右操作数：**0**	右操作数：**Num**	右操作数：**Inf**	右操作数：**NaN**
左操作数：**0**	NaN	0	0	NaN
左操作数：**Num**	Inf	Num	0	NaN
左操作数：**Inf**	Inf	Inf	NaN	NaN
左操作数：**NaN**	NaN	NaN	NaN	NaN

表 **B-20** 比较结果

	右操作数：**0**	右操作数：**+Num**	右操作数：**+Inf**	右操作数：**+NaN**
左操作数：**0**	=	<	<	Uno
左操作数：**+Num**	>	比较的结果	<	Uno
左操作数：**+Inf**	>	>	=	Uno
左操作数：**+NaN**	Uno	Uno	Uno	Uno

注 – NaN 与 NaN 比较结果为无序，从而导致不相等。 +0 比较等于 -0。

B.3 参数传递机制

本节描述在 ISO C 中如何传递参数。

- 传递给 C 函数的所有参数均通过值进行传递。
- 实际参数按函数声明中声明参数的反向顺序传递。
- 本身为表达式的实际参数在函数引用之前求值。然后表达式的结果置入寄存器或推入栈。

32 位 SPARC

函数在寄存器 %o0 中返回 integer 结果，在寄存器 %f0 中返回 float 结果，在寄存器 %f0 和 %f1 中返回 double 结果。

long long[1] 整数在寄存器中传递时，高阶字在 %oN 中，低阶字在 %o(N+1) 中。寄存器内的结果按相同的顺序在 %o0 和 %o1 中返回。

除 double 和 long double 之外的所有参数均作为 4 字节值传递。一个 double 作为 8 字节值传递。前六个 4 字节值 （double 计数为 8）在寄存器 %o0 到 %o5 中传递。其余值传递到栈中。结构的传递方式是复制结构并将指针传递到副本。long double 的传递方式与结构相同。

此处描述的寄存器是可被调用程序识别的寄存器。

64 位 SPARC

所有整型参数均作为 8 字节值传递。

浮点参数尽可能在浮点寄存器中传递。

(Intel)

函数在寄存器 %eax 中返回 integer 结果。

long long 结果在寄存器 %edx 和 %eax 中*返回*。函数在寄存器 %st(0) 中返回 float、 double 和 long double 结果。

除 struct、union、long long、double 和 long double 之外的所有参数均作为 4 字节值传递； long long 作为 8 字节值传递， double 作为 8 字节值传递， long double 作为 12 字节值传递。

struct 和 union 被复制到栈中。大小向上舍入为 4 字节的倍数。返回 struct 和 union 的函数被传递一个隐藏的首参数，并指向返回的 struct 或 union 的存储位置。

从一个函数返回时，需要调用程序从栈中弹出参数，但 struct 和 union 返回的附加参数除外，它由调用的函数弹出。

1. 在设置 -xc99=%none 时在 -Xc 模式下不可用。

附录 C

实现定义的 ISO/IEC C 行为

ISO/IEC 9899:1990 编程语言 - C 标准指定以 C 语言编写的程序的形式并加以解释。但是，该标准留下许多实现定义的问题，即因编译器而异的问题。本章详述这些方面。它们很容易与 ISO/IEC 9899:1990 标准本身比较：

- 每个问题均使用 ISO 标准中的相同节文本。
- 每个问题的前面均有 ISO 标准中相应的节号。

C.1 与 ISO 标准比较的实现

C.1.1 转换 (G.3.1)

圆括号中的编号与 ISO/IEC 9899:1990 标准中的节号对应。

(5.1.1.3) Identification of diagnostics （诊断的标识）：

错误消息具有以下格式：

filename， `line` *line number: message*

警告消息具有以下格式：

filename， `line` *line number: warning message*

其中：

- *filename* 是错误或警告所在文件的名称
- *line number* 是错误或警告所在行的编号
- *message* 是诊断消息

C.1.2 环境 (G.3.2)

(5.1.2.2.1) Semantics of arguments to `main` *（*`main` *参数的语义）*：

```
int main (int argc, char *argv[])
{
....
}
```

`argc` 是调用程序时使用的命令行参数的编号。在任何 shell 扩展之后， `argc` 总是至少等于 `1`，即程序的名称。

`argv` 是命令行参数的指针数组。

(5.1.2.3) What constitutes an interactive device （交互式设备的要素）：

交互式设备是系统库调用 `isatty()` 为其返回非零值的设备。

C.1.3 标识符 (G.3.3)

(6.1.2) The number of significant initial characters (beyond 31) in an identifier without external linkage （不带外部链接的标识符中的有效初始字符 （第 31 个字符之后）的数目）：

前 1,023 个字符是有效字符。标识符区分大小写。

(6.1.2) The number of significant initial characters (beyond 6) in an identifier with external linkage （带外部链接的标识符中的有效初始字符 （第 6 个字符之后）的数目）：

前 1,023 个字符是有效字符。标识符区分大小写。

C.1.4 字符 (G.3.4)

(5.2.1) The members of the source and execution character sets, except as explicitly specified in the Standard （除非标准中明确指定，否则为源代码字符集和执行字符集的成员）：

两个集均与 ASCII 字符集相同，外加特定于语言环境的扩展。

(5.2.1.2) The shift states used for the encoding of multibyte characters （用于多字节字符编码的移位状态）：

无移位状态。

(5.2.4.2.1) The number of bits in a character in the execution character set （执行字符集内字符的位数）：

对于 ASCII 部分，一个字符中有 8 位；对于特定于语言环境的扩展部分，一个字符中的位数是 8 位的倍数，具体取决于语言环境。

(6.1.3.4) The mapping of members of the source character set (in character and string literals) to members of the execution character set （源代码字符集成员 （用字符和字符串文字表示）至执行字符集成员的映射）：

源代码字符与执行字符之间映射相同。

(6.1.3.4) The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant （包含一个字符的整型字符常量的值或不以基本执行字符集或宽字符常量的扩展字符集表示的换码序列的值）：

它是最右边字符的数值。例如，`'\q'` 等于 `'q'`。如果这样的换码序列出现，则会发出警告。

(3.1.3.4) The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character （包含多个字符的整型字符常量的值或包含多个多字节字符的宽字节常量的值）：

一个多字符的字符常量，它不是具有从每个字符的数值派生的值的换码序列。

(6.1.3.4) The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant （用于为宽字符常量将多字节字符转换为相应宽字符的当前语言环境）：

由 LC_ALL、 LC_CTYPE 或 LANG 环境变量指定的有效语言环境。

(6.2.1.1) Whether a plain `char` *has the same range of values as* `signed char` *or* `unsigned char` *（无格式* `char` *是与* `signed char` *还是与* `unsigned char` *具体相同的值范围）*：

`char` 被视为 `signed char` *(SPARC) (Intel)*。

C.1.5 整数 (G.3.5)

(6.1.2.5) The representations and sets of values of the various types of integers （各种类型的整数的表示和值集）：

表 C-1 整数的表示和值集

整数	位	最小值	最大值
`char` *(SPARC) (Intel)*	8	-128	127
`signed char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32768	32767
`signed short`	16	-32768	32767
`unsigned short`	16	0	65535

表 C-1 整数的表示和值集 *（续）*

整数	位	最小值	最大值
`int`	32	-2147483648	2147483647
`signed int`	32	-2147483648	2147483647
`unsigned int`	32	0	4294967295
`long` *(SPARC)* `v8`	32	-2147483648	2147483647
`long` *(SPARC)* `v9`	64	-9223372036854775808	9223372036854775807
`signed long` *(SPARC)*`v8`	32	-2147483648	2147483647
`signed long` *(SPARC)* `v9`	64	-9223372036854775808	9223372036854775807
`unsigned long` *(SPARC)* `v8`	32	0	4294967295
`unsigned long` *(SPARC)* `v9`	64	0	18446744073709551615
`long long`[1]	64	-9223372036854775808	9223372036854775807
`signed long long`[1]	64	-9223372036854775808	9223372036854775807
`unsigned long long`[1]	64	0	18446744073709551615

1 在 -Xc 模式下无效

(6.2.1.2) The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented （值无法表示的情况下，整数转换为较短的带符号整数的结果，或者无符号整数转换为同等长度的带符号整数的结果）：

整数转换为较短的 `signed` 整数时，将低阶位从较长的整数复制到较短的 `signed` 整数中。结果可能为负数。

无符号整数转换为同等长度的 `signed` 整数时，将低阶位从 `unsigned` 整数复制到 `signed` 整数中。结果可能为负数。

(6.3) The results of bitwise operations on signed integers （带符号整数的按位操作的结果）：

对 `signed` 类型应用按位操作的结果是操作数的按位操作，包括 `sign` 位。因此，当且仅当两个操作数中每个对应的位均已置位时，结果中的每个位才置位。

(6.3.5) The sign of the remainder on integer division （整数除法的余数的符号）：

结果的符号与被除数相同；因此，-23/4 的余数是 -3。

(6.3.7) The result of a right shift of a negative-valued signed integral type（负值带符号整数类型的右移的结果）：

右移的结果为 `signed` 右移。

C.1.6 浮点 (G.3.6)

(6.1.2.5) The representations and sets of values of the various types of floating-point numbers （各种类型的浮点数的表示和值集）：

表 C-2 float 的值

float	
位	32
最小值	1.17549435E-38
最大值	3.40282347E+38
Epsilon	1.19209290E-07

表 C-3 double 的值

double	
位	64
最小值	2.2250738585072014E-308
最大值	1.7976931348623157E+308
Epsilon	2.2204460492503131E-16

表 C-4 long double 的值

long double	
位	128 *(SPARC)* 80 *(Intel)*
最小值	3.3621031431120935062626778173217526O3E-4932 *(SPARC)* 3.362103143112093506262 7E-4932 *(Intel)*
最大值	1.189731495357231765085759326628007016E+4932 *(SPARC)* 1.18973149535723176502 13E4932 *(Intel)*
Epsilon	1.925929944387235853055977942584927319E-34 *(SPARC)* 1.084202172485504434007 5E-19 *(Intel)*

(6.2.1.3) The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value （当整数转换为不能精确地表示原始值的浮点数时截断的方向）：

数舍入为可以表示的最近的值。

(6.2.1.4) The direction of truncation or rounding when a floating- point number is converted to a narrower floating-point number （当浮点数转换为较窄的浮点数时截断或舍入的方向）：

数舍入为可以表示的最近的值。

C.1.7 数组和指针 (G.3.7)

(6.3.3.4, 7.1.1) The type of integer required to hold the maximum size of an array; that is, the type of the `sizeof` *operator,* `size_t` *（保持数组的最大大小所需的整数类型；即* `sizeof` *操作符* `size_t` *的类型）*：

`stddef.h` 中定义的 `unsigned int`。

对于 `-Xarch=v9` 为 `unsigned long`

(6.3.4) The result of casting a pointer to an integer, or vice versa （将指针强制转换为整数的结果，或将整数强制转换为指针的结果）：

对于指针以及类型 `int`、 `long`、 `unsigned int` 和 `unsigned long` 的值，位模式不变。

(6.3.6, 7.1.1) The type of integer required to hold the difference between two pointers to members of the same array, `ptrdiff_t` *（保持同一数组成员的两指针之差* `ptrdiff_t` *所需的整数类型）*：

`stddef.h` 中定义的 `int`。

对于 `-Xarch=v9` 为 `long`

C.1.8 寄存器 (G.3.8)

(6.5.1) The extent to which objects can actually be placed in registers by use of the `register` *storage-class specifier （可使用* `register` *存储类说明符实际置入寄存器中的对象的范围）*：

有效寄存器声明数取决于每个函数内使用和定义的模式，并受可供分配的寄存器数的限制。不要求编译器和优化器接受寄存器声明。

C.1.9 结构、联合、枚举和位字段 (G.3.9)

(6.3.2.3) A member of a union object is accessed using a member of a different type （使用不同类型的成员访问的联合对象的成员）：

访问存储在联合成员中的位模式，并根据访问成员时所用的成员类型解释值。

(6.5.2.1) The padding and alignment of members of structures （结构成员的填充和对齐）。

表 C-5 结构成员的填充和对齐

类型	对齐边界	字节对齐
`char`	字节	1
`short`	半字	2
`int`	字	4
`long` *(SPARC)* `v8`	字	4
`long` *(SPARC)* `v9`	双字	8
`float` *(SPARC)*	字	4
`double` *(SPARC)*	双字 *(SPARC)* 字 *(Intel)*	8 *(SPARC)* 4 *(Intel)*
`long double` *(SPARC)* `v8`	双字 *(SPARC)* 字 *(Intel)*	8 *(SPARC)* 4 *(Intel)*
`long double` *(SPARC)* `v9`	四倍长字	16
`pointer` *(SPARC)* `v8`	字	4
`pointer` *(SPARC)* `v9`	四倍长字	8
`long long`[1]	双字 *(SPARC)* 字 *(Intel)*	8 *(SPARC)* 4 *(Intel)*

1 在 `-Xc` 模式下不可用。

内部填充结构成员，以便各个元素在适当的边界上对齐。

结构的对齐与其更严格对齐的成员相同。例如，仅包含 `char` 的 `struct` 没有对齐限制，而包含 `double` 的 `struct` 将在 8 字节边界上对齐。

(6.5.2.1) Whether a plain `int` *bit-field is treated as a* `signed int` *bit-field or as an* `unsigned int` *bit-field （无格式* `int` *位字段是视为* `signed int` *位字段还是视为* `unsigned int` *位字段）*：

视为 `unsigned int`。

(6.5.2.1) The order of allocation of bit-fields within an `int` *（*`int` *中的位字段的分配顺序）*：

在存储单元中从高阶到低阶分配位字段。

(6.5.2.1) Whether a bit-field can straddle a storage-unit boundary（位字段是否可以跨存储单元边界）：

位字段不跨存储单元边界。

(6.5.2.2) The integer type chosen to represent the values of an enumeration type （选择用来表示枚举类型的值的整数类型）：

这是 `int`。

C.1.10 限定符 (G.3.10)

(6.5.5.3) What constitutes an access to an object that has volatile-qualified type （什么构成对具有 volatile 限定类型的对象的访问）：

对象名称的每个引用构成对该对象的访问。

C.1.11 声明符 (G.3.11)

(6.5.4) The maximum number of declarators that may modify an arithmetic, structure, or union type （可修改算术、结构或联合类型的声明数最大值）：

编译器不施加任何限制。

C.1.12 语句 (G.3.12)

(6.6.4.2) The maximum number of `case` *values in a* `switch` *statement（*`switch` *声明中的* `case` *值的最大数目）*：

编译器不施加任何限制。

C.1.13 预处理指令 (G.3.13)

(6.8.1) Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set（控制条件包含的常量表达式中单字符常量的值是否与执行字符集内的相同字符常量的值匹配）：

预处理指令中的字符常量与其在任何其它表达式中的数值相同。

(6.8.1) Whether such a character constant may have a negative value（这样的字符常量是否可以具有负值）：

此上下文中的字符常量可以具有负值 *(SPARC) (Intel)*。

(6.8.2) The method for locating includable source files （定位可包含的源文件的方法）：

对于其名称由 < > 定界的文件，首先在 `-I` 选项命名的目录中查找，然后在标准目录中查找。除非使用 `-YI` 选项指定另一个缺省位置，否则标准目录为 `/usr/include`。

对于其名称由引号定界的文件，首先在包含 `#include` 的源文件的目录中查找，然后在 `-I` 选项命名的目录中查找，最后在标准目录中查找。

如果括入 < > 或双引号的文件名以 / 字符开头，则文件名被解释为根目录中的路径名。查找该文件时只能从根目录开始。

(6.8.2) The support of quoted names for includable source files （对可包含的源文件的带引号名称的支持）：

支持 `include` 指令中的带引号文件名。

(6.8.2) The mapping of source file character sequences （源文件字符序列的映射）：

源文件字符映射至其相应的 ASCII 值。

(6.8.6) The behavior on each recognized `#pragma` *directive （对每个识别的* `#pragma` *指令的行为）*：

支持以下 pragma。有关详细信息，请参见第 2-10 页上的第 2.8 节 “Pragma”。

- `align` *integer (variable[, variable])*
- `does_not_read_global_data` *(funcname [, funcname])*
- `does_not_return` *(funcname[, funcname])*
- `does_not_write_global_data` *(funcname[, funcname])*
- `error_messages` *(on|off|default, tag1[tag2... tagn])*
- `fini` *(f1[, f2..., fn])*
- `ident` *string*
- `init` *(f1[, f2..., fn])*
- `inline` *(funcname[, funcname])*
- `int_to_unsigned` *(funcname)*
- `MP serial_loop`
- `MP serial_loop_nested`
- `MP taskloop`
- `no_inline` *(funcname[, funcname])*
- `nomemorydepend`
- `no_side_effect` *(funcname[, funcname])*
- `opt_level` *(funcname[, funcname])*
- `pack`*(n)*
- `pipeloop`*(n)*
- `rarely_called` *(funcname[, funcname])*
- `redefine_extname` *old_extname new_extname*
- `returns_new_memory` *(funcname[, funcname])*
- `unknown_control_flow` *(name[, name])*
- `unroll` *(unroll_factor)*
- `weak` *(symbol1 [= symbol2])*

(6.8.8) The definitions for `__DATE__` *and* `__TIME__` *when, respectively, the date and time of translation are not available （转换的日期和时间分别不可用时* `__DATE__` *和* `__TIME__` *的定义）*：

环境中总是提供这些宏。

C.1.14 库函数 (G.3.14)

(7.1.6) The null pointer constant to which the macro NULL *expands（宏* NULL *扩展的空指针常量）*：

NULL 等于 0。

(7.2) The diagnostic printed by and the termination behavior of the assert *function （*assert *函数的诊断打印依据和终止行为）*：

诊断为：

Assertion failed: *statement*. file *filename*, line *number*

其中：

- *statement* 是使断言失败的语句
- *filename* 是包含失败的文件的名称
- *line number* 是发生失败的行的编号

(7.3.1) The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, *and* isupper *functions （由* isalnum、isalpha、iscntrl、islower、isprint *和* isupper *函数测试的字符集）*：

表 C-6 由 isalpha、islower 等测试的字符集

isalnum	ASCII 字符 A-Z、a-z 和 0-9
isalpha	ASCII 字符 A-Z 和 a-z，以及特定于语言环境的单字节字母
iscntrl	值为 0-31 和 127 的 ASCII 字符
islower	ASCII 字符 a-z
isprint	特定于语言环境的单字节可打印字符
isupper	ASCII 字符 A-Z

(7.5.1) The values returned by the mathematics functions on domain errors （发生域错误时数学函数返回的值）：

表 C-7 发生域错误时返回的值

		编译器模式	
错误	**数学函数**	`-Xs, -Xt`	`-Xa, -Xc`
DOMAIN	acos(|x|>1)	0.0	0.0
DOMAIN	asin(|x|>1)	0.0	0.0
DOMAIN	atan2(+-0,+-0)	0.0	0.0
DOMAIN	y0(0)	-HUGE	-HUGE_VAL
DOMAIN	y0(x<0)	-HUGE	-HUGE_VAL
DOMAIN	y1(0)	-HUGE	-HUGE_VAL
DOMAIN	y1(x<0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,x<0)	-HUGE	-HUGE_VAL
DOMAIN	log(x<0)	-HUGE	-HUGE_VAL
DOMAIN	log10(x<0)	-HUGE	-HUGE_VAL
DOMAIN	pow(0,0)	0.0	1.0
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL
DOMAIN	pow(neg,non-integal)	0.0	NaN
DOMAIN	sqrt(x<0)	0.0	NaN
DOMAIN	fmod(x,0)	x	NaN
DOMAIN	remainder(x,0)	NaN	NaN
DOMAIN	acosh(x<1)	NaN	NaN
DOMAIN	atanh(|x|>1)	NaN	NaN

(7.5.1) Whether the mathematics functions set the integer expression `errno` *to the value of the macro* `ERANGE` *on underflow range errors （发生下溢范围错误时，数学函数是否将整数表达式* `errno` *设置为宏* `ERANGE` *的值）：*

检测到下溢时，除 `scalbn` 之外的数学函数将 `errno` 设置为 `ERANGE`。

(7.5.6.4) Whether a domain error occurs or zero is returned when the `fmod` *function has a second argument of zero* （`fmod` *函数的第二个参数为零时，发生域错误还是返回零）*：

在此情况下，它返回第一个参数并发生域错误。

(7.7.1.1) The set of signals for the `signal` *function* （`signal` *函数的信号集）*：

下表显示 `signal` 函数识别的每个信号的语义：

表 C-8 `signa l` 信号的语义

信号	编号	缺省值	事件
SIGHUP	1	退出	挂起
SIGINT	2	退出	中断
SIGQUIT	3	信息转储	退出
SIGILL	4	信息转储	非法指令 （找到时不重置）
SIGTRAP	5	信息转储	跟踪陷阱 （找到时不重置）
SIGIOT	6	信息转储	`IOT` 指令
SIGABRT	6	信息转储	由中止使用
SIGEMT	7	信息转储	`EMT` 指令
SIGFPE	8	信息转储	浮点异常
SIGKILL	9	退出	`kill` （不能被找到或忽略）
SIGBUS	10	信息转储	总线错误
SIGSEGV	11	信息转储	段违例
SIGSYS	12	信息转储	系统调用的错误参数
SIGPIPE	13	退出	在管道上写，但无读取者
SIGALRM	14	退出	闹钟
SIGTERM	15	退出	来自 `kill` 命令的软件终止信号
SIGUSR1	16	退出	用户定义的信号 `1`
SIGUSR2	17	退出	用户定义的信号 `2`
SIGCLD	18	忽略	子进程状态更改
SIGCHLD	18	忽略	子进程状态更改别名

表 **C-8** signa l 信号的语义 *（续）*

信号	编号	缺省值	事件
SIGPWR	19	忽略	电源故障，重新启动
SIGWINCH	20	忽略	窗口大小改变
SIGURG	21	忽略	紧急套接字条件
SIGPOLL	22	退出	发生可轮询事件
SIGIO	22	退出	可能的套接字输入/输出
SIGSTOP	23	停止	停止 （不能找到或忽略）
SIGTSTP	24	停止	来自 tty 的用户停止请求
SIGCONT	25	忽略	停止的进程已继续
SIGTTIN	26	停止	后台 tty 读尝试
SIGTTOU	27	停止	后台 tty 写尝试
SIGVTALRM	28	退出	虚拟计时器过期
SIGPROF	29	退出	文件配置计时器过期
SIGXCPU	30	信息转储	超出 cpu 限制
SIGXFSZ	31	信息转储	超出文件大小限制
SIGWAITINGT	32	忽略	进程的 lwps 受阻

(7.7.1.1) The default handling and the handling at program startup for each `signal` *recognized by the signal function （信号函数识别的每个* `signal` *的缺省处理以及在程序启动时的处理）*：

参见以上内容。

(7.7.1.1) If the equivalent of `signal(sig, SIG_DFL);` *is not executed prior to the call of a signal handler, the blocking of the signal that is performed （在调用信号处理程序之前未执行* `signal(sig, SIG_DFL);` *的等价时，执行的信号的阻塞）*：

总是执行 `signal(sig,SIG_DFL)` 的等价。

(7.7.1.1) Whether the default handling is reset if the `SIGILL` *signal is received by a handler specified to the signal function （如果指定给信号函数的处理程序收到* `SIGILL` *信号，缺省处理是否复位）*：

收到 `SIGILL` 时缺省处理不复位。

(7.9.2) Whether the last line of a text stream requires a terminating new-line character （文本流的最后一行是否需要一个终止换行符）：

最后一行不需要以换行符结束。

(7.9.2) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in （写出到换行符前面的文本流中的空格字符在读入时是否出现）：

读该流时，所有字符均出现。

(7.9.2) The number of null characters that may be appended to data written to a binary stream （可附加至写入二进制流的数据的空字符数）：

空字符不附加至二进制流。

(7.9.3) Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file （附加模式流的文件位置指示器最初位于文件开头还是结尾）：

文件位置指示器最初位于文件结尾。

(7.9.3) Whether a write on a text stream causes the associated file to be truncated beyond that point （文本流中的写是否导致联合文件在该点之后被截断）：

文本流中的写不会导致文件在该点之后被截断，除非硬件设备强制这种情况发生。

(7.9.3) The characteristics of file buffering （文件缓冲的特性）：

除标准错误流 (stderr) 之外，输出流在输出至文件时缺省情况下缓冲，在输出至终端时采用行缓冲。标准错误输出流 (stderr) 缺省情况下未缓冲。

缓冲的输出流保存多个字符，然后将这些字符作为块进行写。未缓冲的输出流将信息排队，以便立即在目标文件或终端上写。行缓冲的输出将输出的每行排队，直至行完成（请求换行符）时为止。

(7.9.3) Whether a zero-length file actually exists （零长度文件是否实际存在）：

由于零长度文件有目录项，因此它确实存在。

(7.9.3) The rules for composing valid file names （书写有效文件名的规则）：

有效文件名的长度可以为 1 到 1,023 个字符，并且可以使用除字符 null 和 /（斜杠）之外的所有字符。

(7.9.3) Whether the same file can be open multiple times （同一文件是否可以多次打开）：

同一文件可以多次打开。

(7.9.4.1) The effect of the remove *function on an open file （*remove *函数对打开的文件的作用）*：

在执行关闭文件的最后一个调用时删除文件。程序不能打开已删除的文件。

(7.9.4.2) The effect if a file with the new name exists prior to a call to the rename *function （在调用* rename *函数之前存在一个具有新名称的文件时的作用）*：

如果文件存在，将删除文件，并且新文件改写先前存在的文件。

(7.9.6.1) The output for `%p` *conversion in the* `fprintf` *function*（`fprintf` *函数中* `%p` *转换的输出*）：

`%p` 的输出等价于 `%x`。

(7.9.6.2) The input for `%p` *conversion in the* `fscanf` *function*（`fscanf` *函数中* `%p` *转换的输出*）：

`%p` 的输出等价于 `%x`。

(7.9.6.2) The interpretation of a - character that is neither the first nor the last character in the scan list for `%[` *conversion in the* `fscanf` *function*（`fscanf` *函数中* `%[` *转换的扫描列表中既不是第一个也不是最后一个字符的 - 字符的解释*）：

`-` 字符表示一个包括边界的范围，因此 `[0-9]` 等价于 `[0123456789]`。

C.1.15 特定于语言环境的行为 (G.4)

(7.12.1) The local time zone and Daylight Savings Time（*本地时区和夏令时*）：

本地时区由环境变量 `TZ` 设置。

(7.12.2.1) The era for the `clock` *function*（`clock` *函数的时代*）

时钟的时代表示为以程序的起始执行时间为起点的时钟计时信号。

主机环境的以下特性特定于语言环境：

(5.2.1) The content of the execution character set, in addition to the required members（*执行字符集的内容，以及必需的成员*）：

特定于语言环境（C 语言环境中无扩展）。

(5.2.2) The direction of printing （打印方向）：

打印总是从左到右进行。

(7.1.1) The decimal-point character （小数点字符）：

特定于语言环境 （C 语言环境中 “.”）。

(7.3) The implementation-defined aspects of character testing and case mapping functions （字符测试和条件映射函数的实现定义方面）：

与 4.3.1 相同。

(7.11.4.4) The collation sequence of the execution character set（执行字符集的整理序列）：

特定于语言环境 （C 语言环境中的 ASCII 整理）。

(7.12.3.5) The formats for time and date （时间和日期的格式）：

特定于语言环境。下面几个表显示 C 语言环境中的格式。

月份名称为：

表 C-9 月份名称

1 月	5 月	9 月
2 月	6 月	10 月
3 月	7 月	11 月
4 月	8 月	12 月

周日期的名称为：

表 C-10 周日期及缩写

日期		缩写	
星期日	星期四	日	四
星期一	星期五	一	五
星期二	星期六	二	六
星期三		三	

时间的格式为：

```
%H:%M:%S
```

日期的格式为：

```
%m/%d/%y
```

上午和下午的格式为：上午 下午

附录 D

支持的 C99 功能

本附录列出 ISO/IEC 9899:1999 编程语言 - C 标准的受支持功能。本附录还提供关于其中某些受支持功能的讨论和示例。有关未在本附录中讨论的受支持功能的详细信息，请访问 http://forte.sun.com/s1scc/index.html。

`-xc99` 标志控制编译器对实现功能的识别。有关 `-xc99` 语法的详细信息，请参见第 A-35 页上的 “`-xc99[=o]`”。

注 – 虽然编译器在缺省情况下支持下面列出的 C99 功能，但是 `/usr/include` 中由 Solaris 软件提供的标准头文件仍不符合 1999 ISO/IEC C 标准。如果遇到错误消息，请尝试使用 `-xc99=%none` 获取这些头文件的 1990 ISO/IEC C 标准行为。

- 子条款 6.2.5 _Bool
- 子条款 6.2.5 _Complex type （_Complex 类型）

  本发行版本支持 _Complex 的部分实现。在 Solaris 7 操作环境、 Solaris 8 操作环境和 Solaris 9 操作环境中，您必须与 `-lcplxsupp` 链接。
- 子条款 6.3.2.1 Conversion of arrays to pointers not limited to lvalues（不仅限于左值的数组至指针转换）
- 子条款 6.4.1 Keywords （关键字）
- 子条款 6.4.2.2 Predefined identifiers （预定义标识符）
- 子条款 6.4.4.2 Hexadecimal floating-point literals （十六进制浮点文字）
- 子条款 6.4.9 Comments （注释）
- 子条款 6.5.2.5 Compound literals （复合文字）
- 子条款 6.7.2 Type specifiers （类型说明符）
- 子条款 6.7.2.1 Structure and union specifiers （结构和联合说明符）
- 子条款 6.7.3 Type Qualifier （类型限定符）
- 子条款 6.7.4 Function specifiers （函数说明符）
- 子条款 6.7.5.2 Array declarator （数组声明符）
- 子条款 6.8.2 Compound statement （复合语句）
- 子条款 6.8.5 Iteration statements （迭代语句）
- 子条款 6.10.3 Macro replacement （宏替换）

- 子条款 6.10.6 STDC pragmas （STDC pragma）
- 子条款 6.10.8 __STDC_IEC_559 and __STDC_IEC_559_COMPLEX macros （__STDC_IEC_559 和 __STDC_IEC_559_COMPLEX 宏）
- 子条款 6.10.9 Pragma operator （Pragma 操作符）

D.1 幂等限定符

6.7.3 Type qualifiers （类型限定符）：

如果同一限定符在同一说明符限定符列表中出现多次 （无论直接出现还是通过一个或多个 typedef），行为与该类型限定符仅出现一次时相同。

在 C90 中，以下代码会导致错误：

```
%example cat test.c

const const int a;

int main(void) {
  return(0);
}

%example cc -xc99=%none test.c
"test.c", 第 1 行：无效类型组合
```

但是，对于 C99， C 编译器接受多个限定符。

```
%example cc -xc99 test.c
%example
```

D.2 _Pragma

形式为 _Pragma （ *string-literal* ）的一元操作符表达式处理如下：

- 如果字符串文字具有 L 前缀，则删除该前缀。
- 删除前导和结尾双引号。
- 用双引号替换每个换码序列 '。
- 用单个反斜杠替换每个换码序列 \\。

预处理标记的结果序列作为 pragma 指令中的预处理程序标记进行处理。

删除一元操作符表达式中的最初四个预处理标记。

与 #pragma 比较， _Pragma 的优势在于：_Pragma 可以用于宏定义。

_Pragma("string") 与 #pragma 字符串行为完全相同。考虑以下示例。首先列出示例的源代码，然后在预处理程序使它通过预处理之后，再列出示例的源代码。

```
example% cat test.c

#include <omp.h>
#include <stdio.h>

#define Pragma(x) _Pragma(#x)
#define OMP(directive) Pragma(omp directive)

void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  OMP(parallel)
  {
  printf("Hello!\n");
  }
}

example% cc test.c -P -xopenmp -x03
example% cat test.i
```

下面是预处理程序完成后的源代码。

```
void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  # pragma omp parallel
  {
    printf("Hellow!\n");
  }
}

example% cc test.c -xopenmp -->
example% ./a.out
Hello!
Hello!
example%
```

D.3 混合声明和代码

6.8.2 Compound statement （复合语句）

现在， C 编译器接受关于可执行代码的混合类型声明，如以下示例所示：

```
#include <stdio.h>

int main(void){
  int num1 = 3;
  printf("%d\n", num1);

  int num2 = 10;
  printf("%d\n", num2);
  return(0);
}
```

D.4 Static 及数组声明符中允许的其它类型限定符

6.7.5.2 Array declarator （数组声明符）：

关键字 `static` 可以出现在函数声明符中参数的数组声明符中，表示编译器至少可以假定许多元素将传递到所声明的函数中。使优化器能够作出以其它方式无法确定的假定。

C 编译器将数组参数调整为指针，因此 `void foo(int a[])` 与 `void foo(int *a)` 相同。

如果您指定 `void foo(int * restrict a);` 等类型限定符，则 C 编译器使用实质上与声明限定指针相同的数组语法 `void foo(int a[restrict]);` 表示它。

C 编译器还使用 `static` 限定符保留关于数组大小的信息。例如，如果您指定 `void foo(int a[10])`，则编译器仍将它表达为 `void foo(int *a)`。使用如下 `static` 限定符 `void foo(int a[static 10])` 让编译器知道指针 a 不是 NULL，并且使用它访问至少包含十个元素的整数数组。

D.5 灵活的数组成员

6.7.2.1 Structure and union specifiers （结构和联合说明符）

也称为 “struct hack”。允许数据结构的最后一个成员是长度为零的数组，如 `int foo[];`。这种数据结构一般用作访问 malloc 内存的头文件。

例如，在结构 `struct s { int n; double d[]; } S;` 中，数组 `d` 是不完全数组类型。对于 `S` 的该成员， C 编译器不对任何内存偏移进行计数。换句话说，`sizeof(struct s)` 与 `S.n` 的偏移相同。

可以像使用任何普通数组成员一样使用 `d`。`S.d[10] = 0;`。

如果没有 C 编译器对不完全数组类型的支持，您将如以下示例 `DynamicDouble` 所示定义和声明结构：

```
typedef struct { int n; double d[1]; ) DynamicDouble;
```

请注意，数组 `d` 不是不完全数组类型，并且使用一个成员进行声明。

其次，声明指针 `dd` 并分配内存，如下所示：

```
DynamicDouble *dd = malloc(sizeof(DynamicDouble)+(actual_size-
1)*sizeof(double));
```

然后，将偏移的大小存储在 `S.n` 中，如下所示：

```
dd->n = actual_size;
```

由于编译器支持不完全数组类型，因此您可以无需使用一个成员声明数组而达到同样的效果。

```
typedef struct { int n; double d[]; } DynamicDouble;
```

如以前一样声明指针 `dd` 并分配内存，只是不必再从 actual_size 中减去一：

```
DynamicDouble *dd = malloc (sizeof(DynamicDouble) +
(actual_size)*sizeof(double));
```

如以前一样将偏移存储在 S.n 中，如下所示：

```
dd->n = actual_size;
```

D.6 使用隐式 int 进行声明

6.7.2 Type specifiers （类型说明符）：

每个声明中的声明说明符中应至少指定一个类型说明符。

C 编译器对任何隐式 int 声明发出警告，如以下示例所示：

```
example% more test.c
volatile i;
const foo()
{
  return i;
}
example% cc test.c
  "test.c", 第 1 行：警告：没有显式指定类型
  "test.c", 第 3 行：警告：没有显式指定类型
example%
```

D.7 禁止的隐式 int 和隐式函数声明

与 1990 C 标准不同， 1999 C 标准不再允许隐式声明。 C 编译器的以前版本仅在设置了 -v （冗余）的情况下对隐式定义发出警告消息。只要标识符隐式定义为 int 或函数，系统会对隐式定义发出这些消息及其它新警告。

该编译器的几乎所有用户均可能注意到这种变化，原因是它会导致大量警告消息。常见原因包括未能包含用于声明所使用的函数的相应系统头文件，如需要包含 <stdio.h> 的 printf。可以使用 -xc99=%none 恢复无提示地接受隐式声明的 1990 C 标准行为。

C 编译器对隐性函数声明生成警告：

```
example% cat test.c
void main()
{
  printf("Hello, World!\n");
}
example% cc test.c
"test.c", 第 3 行： 警告： 隐式函数声明： printf
example%
```

D.8 for 循环语句中的声明

6.8.5 Iteration statements （迭代语句）

C 编译器接受作为 for 循环语句中第一个表达式的类型声明：

```
for (int i=0; i<10; i++){ //loop body };
```

for 循环的初始化语句中声明的任何变量的作用域是整个循环 （包括控制和迭代表达式）。

D.9 C99 关键字

6.4.1 Keywords （关键字）

C99 标准引入以下新关键字。在 -xc99=%none 的情况下编译时，如果您使用这些关键字作为标识符，编译器会发出警告。在不设置 -xc99=%none 的情况下，编译器会根据上下文对使用这些关键字作为标识符发出警告或错误消息。

- inline
- _Imaginary
- _Complex
- _Bool
- restrict

D.9.1 使用 restrict 关键字

通过 restrict 限定指针访问的对象需要该对象用于直接或间接访问该特定 restrict 限定指针的值的所有访问权。通过任何其它方式访问该对象可能导致未定义的行为。restrict 限定符的既定用途是允许编译器作出提升优化的假定。

有关示例以及关于如何有效地使用 restrict 限定符的说明，请参见第 3-19 页上的第 3.8.2 节 “限定指针”。

D.10 __func__ 支持

6.4.2.2 Predefined identifiers （预定义标识符）

编译器支持预定义标识符 __func__。 __func__ 定义为字符数组，它包含 __func__ 所在的当前函数的名称。

D.11 具有可变数目的参数的宏

6.10.3 Macro replacement （宏替换）

C 编译器接受以下形式的 #define 预处理程序指令：

```
#define identifier (...) replacement_list
#define identifier (identifier_list, ...) replacement_list
```

如果宏定义中的 *identifier_list* 以省略号结尾，则意味着调用中的参数比宏定义中的参数（不包括省略号）多。否则，宏定义中参数的数目 （包括由预处理标记组成的参数）与调用中参数的数目匹配。对于在其参数中使用省略号表示法的 #define 预处理指令，在其替换列表中使用标识符 __VA_ARGS__。以下示例说明可变参数列表宏工具。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
           printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n",x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

其结果如下：

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):printf("x is %d but y is %d", x, y));
```

D.12 可变长度数组 (VLA)：

6.7.5.2 Array declarators （数组声明符）

在栈中分配 VLA 时，仿佛调用了 `alloca` 函数。无论其作用域如何，其生存期与通过调用 `alloca` 函数在栈中分配数据时相同；直到函数返回时为止。如果在其中分配 VLA 的函数返回时释放栈，则释放分配的空间。

尚未对可变长度数组强制施加所有约束。约束违规导致未定义的结果。

```
#include <stdio.h>
void foo(int);

int main(void) {
  foo(4);
  return(0);
}

void foo (int n) {
  int i;
  int a[n];
  for (i = 0; i < n; i++)
    a[i] = n-i;
  for (i = n-1; i >= 0; i--)
    printf("a[%d] = %d\n", i, a[i]);
}

example% cc test.c
example% a.out
a[3] = 1
a[2] = 2
a[1] = 3
a[0] = 4
```

D.13 静态函数的 inline 说明符

6.7.4 Function specifiers （函数说明符）

已增加 C99 函数说明符 `inline`。 `inline` 对包含内部链接的函数具有完全功能。对于使用外部链接定义的函数，使用 `inline` 函数说明符只创建内联定义，不创建函数的外部定义。因此，具有外部链接的内联函数的指针对于每个转换单元是唯一的，不会相等。

D.14 使用 // 注释代码

6.4.9 Comments （注释）

字符 // 引入包含直到 （但不包括）新换行符的所有多字节字符的注释，除非 // 字符出现在字符常量、字符串文字或注释中。

附录 E

性能调节 *(SPARC)*

本附录描述 SPARC 平台上的性能调节。

E.1 限制

C 库的某些部分不能进行优化以提高速度，尽管这样做对大多数应用程序有益。某些示例：

- 整数运算例程 — 当前 SPARC V8 处理器支持整数乘法和除法指令。但是，如果标准 C 库例程要使用这些指令，则在 V7 SPARC 处理器上运行的程序将由于内核仿真开销而运行慢，或者完全中断。因此，在标准 C 库例程中不能使用整数乘法和除法指令。
- 双字内存访问 — 块复制和移动例程 （如 `memmove()` 和 `bcopy()`）如果使用 SPARC 双字装入和存储指令（`ldd` 和 `std`）, 则运行速度显著提高。某些内存映射设备（如帧缓冲区）不支持 64 位访问；然而，这些设备对于 `memmove()` 和 `bcopy()` 则工作正常。因此，在标准 C 库例程中不能使用 `ldd` 和 `std`。
- 内存分配算法 — C 库例程 `malloc()` 和 `free()` 在旧的 UNIX 程序中通常在综合考虑速度、空间以及对编码错误的非敏感性的基础上实现。基于 “伙伴系统”算法的内存分配器通常比标准库版本运行快，但是占用更多空间。

E.2 libfast.a 库

libfast.a 库提供标准 C 库函数的速度调节版本。由于它是可选库，因此它可以使用尽管对大多数应用程序可以提高性能，但是可能不适合标准 C 库的算法和数据表示。

使用文件配置确定以下清单中的例程对于您的应用程序的性能是否重要，然后根据该清单决定 libfast.a 是否有益于性能：

- 如果整数乘法或除法的性能很重要，即使应用程序的单个二进制版本必须在 V7 和 V8 SPARC 平台上运行，也请*务必*使用 libfast.a。这些重要例程是：.mul、 .div、 .rem、 .umul、 .udiv 和 .urem。
- 如果内存分配的性能很重要，并且分配的块大小普遍接近 2 的幂，请*务必*使用 libfast.a。这些重要例程是：malloc()、 free() 和 realloc()。
- 如果块移动或填充例程的性能很重要，请*务必*使用 libfast.a。这些重要例程是：bcopy()、 bzero()、 memcpy()、 memmove() 和 memset()。
- 如果应用程序要求在用户模式下对不支持 64 位内存操作的 I/O 设备进行内存映射访问，请*不要*使用 libfast.a。
- 如果应用程序是多线程应用程序，请*不要*使用 libfast.a。

链接应用程序时，将选项 -lfast 增加到链接时使用的 cc 命令中。在标准 C 库中，cc 命令先于其对手链接 libfast.a 中的例程。

附录 F

K&R Sun C 与 Sun ISO C 之间的差异

本附录描述以前的 K&R Sun C 与 Sun ISO C 之间的差异。

有关详细信息，请参见第 1-1 页上的第 1.1 节 “标准一致性”。

F.1 K&R Sun C 与 Sun ISO C 的不兼容性

表 F-1 K&R Sun C 与 Sun ISO C 的不兼容性

主题	Sun C (K&R)	Sun ISO C
`main()` 的 `envp` 参数	允许 `envp` 作为 `main()` 的第三个参数。	允许第三个参数；但是，该用法并不严格符合 ISO C 标准。
关键字	将标识符 `const`、`volatile` 和 `signed` 视为普通标识符。	`const`、`volatile` 和 `signed` 是关键字。
块内部的 `extern` 和 `static` 函数声明	将这些函数声明提升为文件作用域。	ISO 标准不保证块作用域函数声明提升为文件作用域。
标识符	允许标识符中使用美元符号 (`$`)。	`$` 不允许使用。
`long float` 类型	接受 `long float` 声明，并将这些声明视为 `double`。	不接受这些声明。

多字符的字符常量	`int mc = 'abcd';`
	产生：
	`abcd`
		`int mc = 'abcd';`
		产生：
		`dcba`

整型常量	在八进制换码序列中接受 8 或 9。	在八进制换码序列中不接受 8 或 9。

表 F-1　K&R Sun C 与 Sun ISO C 的不兼容性　*（续）*

主题	Sun C (K&R)	Sun ISO C
赋值操作符	将以下一对操作符视为两个标记，因此它们之间允许出现空白： `*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	将它们视为单个标记，因此不允许之间出现空白。
表达式的无符号保留语义	支持无符号保留，即 `unsigned char/short` 转换为 `unsigned int`。	支持值保留，即 `unsigned char/short` 转换为 `int`。
单/双精度计算	将浮点表达式的操作数提升为 `double`。 声明为返回 `float` 的函数总是将其返回值提升为 `double`。	允许在单精度计算中执行对 `float` 的操作。 允许这些函数具有 `float` 返回类型。
`struct/union` 成员的名称空间	允许使用成员选择操作符 ('.', '->') 的 `struct`、`union` 和算术类型处理其它 `struct` 或 `union` 的成员。	要求各个唯一 `struct/union` 具有自己的唯一名称空间。
作为 `lvalue` 的强制类型转换	支持作为 `lvalue` 的强制类型转换。例如： `(char *)ip = &char;`	不支持此功能。

隐含的 `int` 声明	支持不带显式类型说明符的声明。`num;` 等声明被视为隐含的 `int`。例如：
	`num; /*num implied as an int*/`
	`int num2; /* num2 explicitly*/`
	`/* declared an int */`
		不支持 `num;` 声明 （不带显式类型说明符 `int`），生成语法错误。

空声明	允许空声明，如： `int;`	除标记之外，禁止空声明。

类型定义中的类型说明符	允许 `typedef` 声明中出现 `unsigned`、`short`、`long` 等类型说明符。例如：
	`typedef short small;`
	`unsigned small x;`
		不允许类型说明符修改 `typedef` 声明。

位字段中允许的类型	允许所有整数类型的位字段，包括未命名位字段。 ABI 要求支持未命名位字段及其它整数类型。	仅支持 `int`、`unsigned int` 和 `signed int` 类型的位字段。未定义其它类型。

表 F-1　K&R Sun C 与 Sun ISO C 的不兼容性　*（续）*

主题	Sun C (K&R)	Sun ISO C

不完全声明中标记的处理	忽略不完全类型声明。在以下示例中，f1 是指外部 struct：
	`struct x { . . . } s1;`
	`{struct x; struct y {struct x f1; } s2; struct x { . . . };}`
		在符合 ISO 的实现中，不完全 struct 或 union 类型说明符隐藏具有相同标记的封装声明。

struct/union/enum 声明中的不匹配	允许嵌套的 struct/union 声明中标记的 struct/enum/union 类型出现不匹配。在以下示例中，第二个声明被视为 struct：
	`struct x {. . .}s1;`
	`{union x s2;. . .}`
		将内部声明视为新声明，隐藏外部标记。

表达式中的标签	将标签视为 (void *) lvalue。	不允许在表达式中使用标签。
switch 条件类型	通过将 float 和 double 转换为 int，允许使用这两种类型。	对于 switch 条件类型，只对整数类型（int、char 和枚举类型）求值。
条件包含指令的语法	预处理程序忽略 #else 或 #endif 指令后面的结尾标记。	禁止这样的构造。
标记粘贴和 ## 预处理程序操作符	不识别 ## 操作符。通过在要粘贴的两个标记之间加入注释，完成标记粘贴： `#define PASTE(A,B) A/*any comment*/B`	将 ## 定义为执行标记粘贴的预处理程序操作符，如以下示例所示： `#define PASTE(A,B) A##B` 而且，Sun ISO C 预处理程序不识别 Sun C 方法。相反，它将两个标记之间的注释视为空白。

预处理程序重新扫描	预处理程序递归替换：
	`#define F(X) X(arg)`
	`F(F)`
	产生
	`arg(arg)`
		在重新扫描过程中，如果在替换列表中找到宏，则不替换宏：
		`#define F(X)X(arg)`
		`F(F)`
		产生：
		`F(arg)`

形式参数列表中的 typedef 名称	您可以使用 typedef 名称作为函数声明中的形式参数名称。“隐藏” typedef 声明。	禁止使用声明为 typedef 名称的标识符作为形式参数。

表 F-1 K&R Sun C 与 Sun ISO C 的不兼容性 *（续）*

主题	Sun C (K&R)	Sun ISO C

聚集的特定于实现的初始化	对花括号中部分省略的初始化函数进行语法分析和处理时，使用自底向上算法：
	`struct{ int a[3]; int b; }\`
	`w[]={{1},2};`
	产生
	`sizeof(w)=16`
	`w[0].a=1,0,0`
	`w[0].b=2`
		使用自顶向下语法分析算法。例如：
		`struct{int a[3];int b;}\`
		`w[]={{1},2};`
		产生
		`sizeof(w)=32`
		`w[0].a=1,0,0`
		`w[0].b=0`
		`w[1].a=2,0,0`
		`w[1].b=0`

跨 include 文件的注释	允许在 #include 文件中开始的注释由包含第一个文件的文件终止。	在编译的转换阶段，注释被空白替换，然后处理 #include 指令。

字符常量中的形式参数替换	在字符常量与替换列表宏匹配时，替换字符常量中的字符：
	`#define charize(c)'c'`
	`charize(Z)`
	产生：
	`'Z'`
		不替换字符：
		`#define charize(c) 'c'`
		`charize(Z)`
		产生：
		`'c'`

字符串常量中的形式参数替换	字符串常量包含形式参数时，预处理程序替换形式参数：
	`#define stringize(str)`
	`'str'`
	`stringize(foo)`
	产生：
	`"foo"`
		应使用 # 预处理程序操作符：
		`#define stringize(str)`
		`'str'`
		`stringize(foo)`
		产生：
		`"str"`

内置到编译器“前端”的预处理程序	编译器调用 cpp(1)，然后调用编译系统的所有其它组件，取决于指定的选项。	ISO C 转换阶段 1-4 涉及预处理程序指令的处理，直接内置到 acomp 中，因此除了在 `-Xs` 模式下之外，编译时不直接调用 cpp。
使用反斜杠的行并置	不识别此上下文中的反斜杠字符。	要求换行符紧跟在反斜杠字符后面并拼接在一起。
字符串文字中的三字母	不支持此 ISO C 功能。	
`asm` 关键字	`asm` 是关键字。	将 `asm` 视为普通标识符。

表 F-1　K&R Sun C 与 Sun ISO C 的不兼容性　*（续）*

主题	Sun C (K&R)	Sun ISO C

标识符的链接	不将未初始化的 static 声明视为暂定声明。因此，第二个声明将生成“重新声明”错误，如下所示：
	`static int i = 1;`
	`static int i;`
		将未初始化的 static 声明视为暂定声明。

名称空间	只分为三类： struct/union/enum 标记、 struct/union/enum 的成员及其它。	识别四种不同的名称空间：标签名称、标记（跟在关键字 struct、union 或 enum 后面的名称）、struct/union/enum 的成员以及普通标识符。
long double 类型	不支持。	允许 long double 类型声明。
浮点常量	不支持浮点后缀 f、l、F 和 L。	
无后缀整型常量可以具有不同的类型	不支持整型常量后缀 u 和 U。	
宽字符常量	不接受宽字符常量的 ISO C 语法，如下所示： `wchar_t wc = L'x';`	支持此语法。
'\a' 和 '\x'	将它们视为字符 'a' 和 'x'。	将 '\a' 和 '\x' 视为特殊换码序列。
字符串文字的并置	不支持相邻字符串文字的 ISO C 并置。	
宽字符字符串文字语法	不支持此示例中所示的 ISO C 宽字符字符串文字语法： `wchar_t *ws = L"hello";`	支持此语法。
指针： void * 与 char *	支持 ISO C void * 功能。	
一元加运算符	不支持此 ISO C 功能。	
函数原型 — 省略号	不支持。	ISO C 定义使用省略号“...”表示变量参数列表。
类型定义	禁止另一个具有相同类型名称的声明在内部块中重新声明 typedef。	允许另一个具有相同类型名称的声明在内部块中重新声明 typedef。
extern 变量的初始化	不支持显式声明为 extern 的变量的初始化。	将显式声明为 extern 的变量的初始化视为定义。

表 F-1 K&R Sun C 与 Sun ISO C 的不兼容性 *（续）*

主题	Sun C (K&R)	Sun ISO C
聚集的初始化	不支持联合或自动结构的 ISO C 初始化。	
原型	不支持此 ISO C 功能。	
预处理指令的语法	仅识别第一列中具有 # 的指令。	ISO C 允许 # 指令前面有前导空白字符。
# 预处理程序操作符	不支持 ISO C # 预处理程序操作符。	
#error 指令	不支持此 ISO C 功能。	
预处理程序指令	支持 unknown_control_flow 和 makes_regs_inconsistent 两个 pragma 以及 #ident 指令。预处理程序发现无法识别的 pragma 时发出警告。	不指定预处理程序对无法识别的 pragma 的行为。
预定义宏名称	未定义以下 ISO C 定义的宏名称： __STDC__ __DATE__ __TIME__ __LINE__	

F.2 关键字

下面几个表列出 ISO C 标准、 Sun ISO C 编译器以及 Sun C 编译器的关键字。

第一个表列出 ISO C 标准定义的关键字。

表 F-2 ISO C 标准关键字

_Bool[1]	_Complex[1]	_Imaginary[1]	auto
break	case	char	const
continue	default	do	double
else	enum	extern	float
for	goto	if	inline[1]
int	long	register	restrict[1]

表 F-2 ISO C 标准关键字 *（续）*

return	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	void	volatile
while			

1 仅在 -xc99=%all 情况下定义。

C 编译器还定义一个附加关键字 asm。但是，在 -Xc 模式下不支持 asm。

下面列出 Sun C 中的关键字。

表 F-3 Sun C (K&R) 关键字

asm	auto	break	case
char	continue	default	do
double	else	enum	extern
float	for	fortran	goto
if	int	long	register
return	short	sizeof	static
struct	switch	typedef	union
unsigned	void	while	

附录 G

OpenMP 的实现特有信息

本附录详述 *OpenMP C 和 C++ Application Program Interface 版本 1.0 - 1998 年 10 月*（可从以下站点获得：`http://www.openmp.org`）的实现特有详细信息。

- 在缺少显式定义的 `OMP_SCHEDULE` 环境变量的情况下，该实现对包含 `schedule(runtime)` 的循环使用静态调度。
- 在缺少显式定义的调度子句的情况下，缺省值为静态调度。
- 如果不通过 `opm_set_num_threads` 函数或 `OMP_NUM_THREADS` 环境变量显式指定组中的线程数，则缺省值为 1。
- 如果不通过 `omp_set_dynamic` 函数或 `OMP_DYNAMIC` 环境变量显式指定是否启用线程的动态调整，则缺省值为启用动态调整。
- 不支持嵌套并行性，并且缺省情况下禁止嵌套并行性。

有关用于生成多重处理应用程序的 OpenMP Fortran 95、C 和 C++ 应用程序接口 (API) 的完整摘要，请参见 《*OpenMP API 用户指南*》。

索引

符号

英文字母

A

B

C

D

E

F

L

M

N

Q

R

S

T

W

X

Y

Z