Fortran 编程指南

Sun™ Studio 8

Sun Microsystems, Inc.
www.sun.com

部件号 817-5800-10
2004 年 4 月，修订版 A

请将关于本文档的意见发送至：http://www.sun.com/hwdocs/feedbacK

请回收

Adobe PostScript

目录

表

开始之前

《*Fortran 编程指南*》提供了有关 Sun™ Studio Fortran 95 编译器 `f95` 的基本信息。其中介绍了 Fortran 95 的输入 / 输出、程序开发、库、程序分析与调试、数值精度、移植、性能、优化、并行化以及互操作性。

本指南专供科学工作者、工程技术人员以及具有 Fortran 语言的应用知识并希望了解如何有效使用 Fortran 编译器的程序员使用。另外，我们还假设您大体熟悉 Solaris™ 操作环境或 UNIX®。

有关 `f95` 编译器环境和命令行选项方面的信息，另请参见手册 《*Fortran 用户指南*》。

印刷惯例

表 P-1 字样惯例

字样	含义	示例
`AaBbCc123`	命令、文件及目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `% You have mail.`

表 **P-1** 字样惯例

字样	含义	示例
AaBbCc123	键入的内容，用于与计算机屏幕输出相对比	`% su` `Password:`
AaBbCc123	书名、新词或术语、要强调的词语	参阅 *《用户指南》*第 6 章。 这些称为*类*选项。 您*必须*是超级用户才能执行此项操作。
AaBbCc123	命令行占位文字；用实际名称或值替换	要删除文件，请键入 **rm** *filename*。

- 符号 Δ 代表空格，此处的空格是有意义的：

```
ΔΔ36.001
```

- FORTRAN 77 标准采用了较早的惯例，名称 “FORTRAN”用大写字母拼写。当前的惯例是使用小写字母：“Fortran 95”
- 对联机手册页的引用以主题名加部分号形式出现。例如，对库例程 GETENV 的引用将显示为 getenv(3F)，这意味着访问该手册页的 man 命令将是：`man -s 3F getenv`。

表 **P-2** 代码惯例

代码符号	含义	表示法	代码示例
[ ]	方括号中的参数为可选参数。	O[*n*]	-O4、O
{ }	大括号中包含必需选项的选择集。	d{y|n}	-dy
|	“管道”或“竖线”符号用于分隔参数，只能选择其中之一。	B{dynamic|static}	-Bstatic
:	与逗号类似，冒号有时用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号指略去了一个系列项中的某些项。	-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

编译器和工具及它们的手册页并未安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问这些编译器和工具，必须正确设置 PATH 环境变量 （参见第 xv 页中的“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 （参见第 xvi 页中的 “访问手册页”）。

有关 PATH 变量的详细信息，参见 csh(1)、sh(1) 和 ksh(1) 手册页。有关 MANPATH 变量的详细信息，参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问本版本的详细信息，参见安装指南或咨询系统管理员。

注意 – 本部分中的信息假定您的 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，请向系统管理员询问您系统中的相当路径。

访问编译器和工具

采用以下步骤确定是否需要更改 PATH 变量才能访问编译器和工具。

▼ 确定是否需要设置 PATH 环境变量

1. **在命令提示符处键入以下命令，显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **查看输出结果，查找包含 /opt/SUNWspro/bin/ 的路径字符串。**

   如果找到了该路径，表明 PATH 变量已设置为可以访问编译器和工具。如果未找到该路径，请按照下一过程中的说明设置 PATH 环境变量。

▼ 设置 PATH 环境变量以便能够对编译器和工具进行访问

1. **如果使用 C shell，请编辑您起始目录下的 .cshrc 文件。如果使用 Bourne shell 或 Korn shell，请编辑您起始目录下的 .profile 文件。**

2. **将以下路径添加至 PATH 环境变量。如果您已安装了 Sun ONE Studio 软件或 Forte Developer 软件，请将以下路径添加到这些安装软件的路径之前。**

   /opt/SUNWspro/bin

访问手册页

使用下列步骤确定是否需要更改 MANPATH 变量才能访问手册页。

▼ 确定是否需要设置 MANPATH 环境变量

1. **在命令提示符处键入以下命令来请求 dbx 手册页。**

```
% man dbx
```

2. **查看输出结果 （如果有）。**

   如果无法找到 dbx(1) 手册页，或者所显示的手册页并非对应于所安装软件的当前版本，请按照下一过程中的说明设置 MANPATH 环境变量。

▼ 设置 MANPATH 环境变量以便能够对手册页进行访问

1. **如果使用 C shell，请编辑您起始目录下的 `.cshrc` 文件。如果使用 Bourne shell 或 Korn shell，请编辑您起始目录下的 `.profile` 文件。**
2. **将以下路径添加至 `MANPATH` 环境变量。**

   `/opt/SUNWspro/man`

访问集成开发环境

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

此 IDE 需要 Sun Studio 8 的核心平台组件。如果核心平台组件没有安装在下列位置之一，必须将 `SPRO_NETBEANS_HOME` 环境变量设置为核心平台组件的安装位置 (*installation_directory*`/netbeans/3.5R`)：

- 缺省安装目录 `/opt/netbeans/3.5R`
- 与 Sun Studio 8 编译器和工具组件相同的位置 （例如，编译器和工具组件安装在 `/foo/SUNWspro`，核心平台组件安装在 `/foo/netbeans/3.5R`）

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

可在下列位置访问文档：

- 文档可从随软件一同安装在本地系统或网络上的文档索引中获得，位置在 `file:/opt/SUNWspro/docs/index.html`。

  如果软件未安装在 `/opt` 目录中，请向系统管理员询问您系统中的相当路径。

- 大多数手册都可以从 docs.sun.com[sm] 网站获得。下列书目只能通过所安装的软件得到：
  - *《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 网站获得。
- 可通过 [帮助] 菜单获得 IDE 所有组件的联机帮助，也可通过 IDE 许多窗口和对话框中的 [帮助] 按钮来获得。

在 docs.sun.com 网站 (http://docs.sun.com) 上，您可以通过因特网阅读、打印和购买 Sun Microsystems 的手册。如果找不到手册，参见随软件一同安装在本地系统或网络上的文档索引。

注意 – Sun 对本文档中提及的第三方网站的可用性概不负责，并且不认可也不对此类站点或资源上或通过其获得的任何内容、广告、产品或其它资料负任何责任或义务。对于因使用或信赖此类站点或资源中提供或通过其提供的任何此类内容、商品或服务而导致或声称导致或与之有关的任何损害或损失， Sun 将不负任何责任或义务。

易访问格式文档

我们还提供了易访问格式的文档，便于残疾用户通过辅助技术阅读。您可以按下表所述找到文档的易访问版本。如果软件未安装在 /opt 目录中，请向系统管理员询问您系统中的相当路径。

文档类型	易访问版本的格式和位置
手册 （不包括第三方手册）	HTML 格式，位于 http://docs.sun.com
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 格式，在已安装软件中通过文档索引访问，位置在 file:/opt/SUNWspro/docs/index.html
自述文件和手册页	HTML 格式，在已安装软件中通过文档索引访问，位置在 file:/opt/SUNWspro/docs/index.html
联机帮助	HTML 格式，可通过 IDE 的 [帮助] 菜单获得
发行说明	HTML 格式，位于 http://docs.sun.com

相关编译器和工具文档

下表介绍了可在 `file:/opt/SUNWspro/docs/index.html` 和 `http://docs.sun.com` 上获得的相关文档。如果软件未安装在 `/opt` 目录中，请向系统管理员询问您系统中的相当路径。

文档名称	说明
《Fortran 用户指南》	介绍 `f95` 编译器的编译时环境以及命令行选项。另外还包括传统 `f77` 程序到 `f95` 的迁移指导。
《Fortran 库参考》	详细介绍 Fortran 库和内在例程
《OpenMP API 用户指南》	概述 OpenMP 多重处理 API，并说明与实现有关的细节。
《数值计算指南》	说明浮点计算数值精度涉及到的问题。

访问 Solaris 相关文档

下表介绍可通过 `docs.sun.com` 网站获得的相关文档。

文档集	文档名称	说明
Solaris 参考手册集	参见手册页各部分的书名。	提供 Solaris 操作环境的有关信息。
Solaris 软件开发人员集	*《链接程序和库指南》*	介绍 Solaris 链接编辑器及运行时链接程序的操作。
Solaris 软件开发人员集	*《多线程编程指南》*	内容包括 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工具。

下表介绍可通过 `docs.sun.com` 网站获得的相关文档。

文档集	文档名称	说明
Solaris 参考手册集	参见手册页各部分的书名。	提供 Solaris 操作环境的有关信息。
Solaris 软件开发人员集	*《链接程序和库指南》*	介绍 Solaris 链接编辑器及运行时链接程序的操作。
Solaris 软件开发人员集	*《多线程编程指南》*	内容包括 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工具。

开发人员资源

访问 http://developers.sun.com/prodtech/cc 可找到以下经常更新的资源：

- 介绍编程技术和最佳办法的文章
- 内含简短编程技巧的知识库
- 编译器和工具组件文档，以及对随软件安装文档的更正
- 支持级别信息
- 用户论坛
- 可下载代码范例
- 新技术预见

还可以在 http://developers.sun.com 上找到其它开发人员资源。

联系 Sun 技术支持

如果您对于本产品有任何技术问题在本文档中找不到答案，请访问：

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

发送意见

Sun 一直致力于提高其文档质量，欢迎您提出意见和建议。请将您的意见通过电子邮件发送给 Sun，邮件地址：

docfeedback@sun.com

请在您电子邮件的主题行中加入文档的部件号 (817-5800-10)。

第 1 章

简介

本书与 *《Fortran 用户指南》*所介绍的 Sun™ Studio Fortran 95 编译器 (`f95`) 可在 SPARC® 和 UltraSPARC® 平台的 Solaris™ 操作环境下使用。该编译器符合已发布的 Fortran 语言标准，并提供了多种扩展功能，其中包括多处理器并行化、完善的代码优化编译以及 C/Fortran 语言混合支持。

`f95` 编译器还提供 Fortran 77 兼容模式，可接受大多数传统 Fortran 77 源代码。该编译器集合不再包括单独的 Fortran 77 编译器。有关 FORTRAN 77 兼容性及迁移问题的信息，参见 *《Fortran 用户指南》*第 5 章。

1.1 标准一致性

- `f95` 的设计目标是兼容 ANSI X3.198-1992、 ISO/IEC 1539:1991 和 ISO/IEC 1539:1997 标准文档。
- 浮点运算基于 IEEE 标准 754-1985 和国际标准 IEC 60559:1989。
- `f95` 对 SPARC V8 和 SPARC V9（包括 UltraSPARC 实现）的优化利用功能提供支持。这些功能在 Prentice-Hall 为 SPARC International 发行的 《SPARC 体系结构手册》第 8 版 (ISBN 0-13-825001-4) 和第 9 版 (ISBN 0-13-099227-5) 中进行了定义。
- 在本文档中，“标准”意味着符合以上所列标准版本。“非标准”或 “扩展”是指超出这些标准版本的功能。

标准负责团体可能会不时地修订这些标准。这些编译器所遵循的适用标准的版本可能会进行修订或更替，从而导致 Sun Fortran 编译器后期版本功能与早期版本不兼容。

1.2 Fortran 95 编译器的功能

Sun Studio Fortran 95 编译器提供下列功能和扩展：

- 跨越例程的全局程序检查，以保证变量、公共块、参数等的一致性。
- 用于多处理器系统经过优化的自动和显式循环并行化。
- VAX/VMS Fortran 扩展，包括：
  - 结构、记录、联合、映射
  - 递归
- OpenMP 并行化指令。
- Cray 风格的并行化指令，其中包括 `TASKCOMMON`。
- 全局、细部及潜在的并行优化会产生高性能的应用程序。基准测试显示，与未经过优化的代码相比，经过优化的应用程序运行速度明显加快。
- Solaris 系统的通用调用惯例允许将用 C 或 C++ 编写的例程与 Fortran 程序合并在一起。
- 支持 UltraSPARC 平台上的具有 64 位能力的 Solaris 环境。
- 使用 `%VAL` 进行按值调用。
- Fortran 77 和 Fortran 95 程序及目标二进制代码间的兼容性。
- “区间运算”编程。
- 某些 “Fortran 2000”功能，包括 “I/O 流”。

有关随各软件版本添加到编译器中的新增和扩展功能的详细信息，参见 *《Fortran 用户指南》*附录 B。

1.3 其它 Fortran 公用程序

下列公用程序在用 Fortran 开发软件程序时可起到辅助作用：

- **Sun Studio 性能分析器 —** 单线程和多线程应用程序深层性能分析工具。参见 `analyzer`(1)。
- **`asa` —** 该 Solaris 公用程序是一个 Fortran 输出过滤器，用于打印在第一列中含有 Fortran 回车控制符的文件。使用 `asa` 可将用 Fortran 回车控制惯例格式化的文件转换成按 UNIX 行式打印机惯例格式化的文件。参见 `asa`(1)。
- **`fdumpmod` —** 用于显示文件或归档文件中所含模块名称的公用程序。参见 `fdumpmod`(1)。
- **`fpp` —** 一个 Fortran 源代码预处理程序。参见 `fpp`(1)。

- **`fsplit` —** 该公用程序将具有若干例程的一个 Fortran 文件分成若干个文件，每个文件一个例程。可对 FORTRAN 77 或 Fortran 95 源文件使用 `fsplit`。参见 `fsplit`(1)。

1.4 调试公用程序

有下列调试公用程序可用：

- **`-Xlist` —** 用于跨例程检查参数、 COMMON 块等项一致性的编译器选项。
- **Sun Studio `dbx` —** 提供强大、功能丰富的运行时和静态调试器，还包括一个性能数据收集器。

1.5 Sun Performance Library

Sun Performance Library™ 是用于计算线性代数和傅立叶变换的经过优化的子例程及函数库。它建立在标准库 LAPACK、 BLAS1、 BLAS2、 BLAS3、 FFTPACK、 VFFTPACK 和 LINPACK 基础之上，这些标准库通常可通过 Netlib (`www.netlib.org`) 获得。

Sun Performance Library 中的每个子程序均执行相同的操作，并且与标准库版本具有相同的接口，但通常速度更快、更精确，并且可用于多处理环境。

有关详细信息，参见 `performance_library` 自述文件和 *《Sun Performance Library 用户指南》*。（性能库例程手册页见第 3P 部分。）

1.6 区间运算

Fortran 95 编译器提供 `-xia` 和 `-xinterval` 编译器标志，以启用新语言扩展并生成适当代码来实现区间算术计算。

有关详细信息，参见 *《Fortran 95 区间运算编程参考》*。

1.7 手册页

联机手册 (man) 页提供命令、函数、子例程及其相应集合的直接文档。要访问 Sun Studio 手册页，参见 “前言” 部分对 MANPATH 环境变量进行正确设置。

可以通过运行以下命令来显示手册页：

```
demo% man topic
```

在整个 Fortran 文档中，手册页参考均以主题名加手册部分编号形式出现：f95(1) 用 man f95 来访问。其它部分，以 ieee_flags(3M) 为例，使用带 -s 选项的 man 命令访问：

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

Fortran 库例程编录在手册页第 3F 部分。

下面列出了 Fortran 用户需关注的**手册**页：

f95(1)	Fortran 95 命令行选项
analyzer(1)	Sun Studio 性能分析器
asa(1)	Fortran 回车控制打印输出后处理程序
dbx(1)	命令行交互式调试器
fpp(1)	Fortran 源代码预处理程序
cpp(1)	C 源代码预处理程序
fdumpmod(1)	显示 “模块” (.mod) 文件的内容。
fsplit(1)	此预处理程序可将 Fortran 源例程分成单个文件
ieee_flags(3M)	检查、设置或清除浮点异常位
ieee_handler(3M)	处理浮点异常
matherr(3M)	数学库错误处理例程
ild(1)	用于目标文件的增量式链接编辑器
ld(1)	用于目标文件的链接编辑器

1.8 自述文件

READMEs 目录下包含的文件描述新增功能、软件不兼容性、错误以及手册印刷后发现的信息。该目录的位置取决于软件安装位置。路径为：/opt/SUNWspro/READMEs/。

表 1-1 需要关注的自述文件

自述文件	描述 ...
fortran_95	本版 Fortran 95 编译器 (f95) 的新增及更改功能、已知限制、文档勘误。
fpp_readme	fpp 功能及能力概述
interval_arithmetic	f95 区间运算功能概述。
math_libraries	可用的优化及专用数学库。
profiling_tools	使用性能剖析工具， prof、 gprof 和 tcov。
runtime_libraries	依据最终用户许可证条款可以重新分发的库及可执行文件。
performance_library	Sun Performance Library 概述

用 -xhelp=readme 命令行选项，可以很方便地查看每个编译器的自述文件。例如，命令：

```
% f95 -xhelp=readme
```

将直接显示 fortran_95 自述文件。

1.9 命令行帮助

调用编译器的 -help 选项，可以查看 f95 命令行选项的扼要说明，如下所示：

```
%f95 -help=flags
[ ] 内为可选项。 < > 内为可变参数项。
竖线 | 表示文字值的选择。
-someoption[={yes|no}] 隐含表示 -someoption 与 -someoption=yes 等效
_____________________________________________________________________
-a                                  收集 tcov 基本块剖析数据
-aligncommon[=<a>] 按指定的边界要求对齐公共块元素； <a>={1|2|4|8|16}
-ansi                               报告非 ANSI 扩展。
-autopar            启用自动循环并行化
-Bdynamic           允许动态链接
-Bstatic            需要静态链接
-C                  启用运行时下标范围检查
-c                  仅编译；生成 .o 文件，但禁止链接
... 其它
```

第 2 章

Fortran 输入 / 输出

本章介绍 Sun Studio Fortran 95 编译器提供的输入 / 输出功能。

2.1 从 Fortran 程序内部访问文件

数据通过 Fortran *逻辑单元*在程序、设备或文件间进行传送。逻辑单元在 I/O 语句中用逻辑单元号来标识，逻辑单元号是从 0 到最大 4 字节整数值 (2,147,483,647) 的非负整数。

字符 * 可以作为逻辑单元标识符出现。当星号出现在 READ 语句中时，它代表*标准输入文件*；当它出现在 WRITE 或 PRINT 语句中时，代表*标准输出文件*。

Fortran 逻辑单元可通过 OPEN 语句与特定的命名文件相关联。另外，在程序开始执行时，某些预连接单元会自动与特定文件相关联。

2.1.1 访问命名文件

OPEN 语句的 FILE= 说明符在运行时建立逻辑单元到命名物理文件的关联。该文件可以是预先就有的，也可以由程序创建。

OPEN 语句中的 FILE= 说明符可以指定一个简单文件名 (FILE='myfile.out')，也可以指定一个前面带有绝对或相对目录路径的文件名 (FILE='../Amber/Qproj/myfile.out')。另外，说明符还可以是字符常量、变量或字符表达式。

可以使用库例程将命令行参数和环境变量以字符变量形式送入程序中，用作 OPEN 语句中的文件名。

以下示例 (GetFilNam.f) 展示了一种由键入的名称来构建绝对路径文件名的方法。程序使用库例程 GETENV、 LNBLNK 和 GETCWD 返回 $HOME 环境变量的值、查找字符串中最后的非空格字符、确定当前工作目录：

```
      CHARACTER F*128, FN*128, FULLNAME*128
      PRINT*, 'ENTER FILE NAME:'
      READ *, F
      FN = FULLNAME( F )
      PRINT *, 'PATH IS: ',FN
      END

      CHARACTER*128 FUNCTION FULLNAME( NAME )
      CHARACTER NAME*(*), PREFIX*128
C          This assumes C shell.
C           Leave absolute path names unchanged.
C           If name starts with '~/', replace tilde with home
C           directory; otherwise prefix relative path name with
C           path to current directory.
      IF ( NAME(1:1) .EQ. '/' ) THEN
            FULLNAME = NAME
      ELSE IF ( NAME(1:2) .EQ. '~/' ) THEN
            CALL GETENV( 'HOME', PREFIX )
            FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     1                    NAME(2:LNBLNK(NAME))
      ELSE
            CALL GETCWD( PREFIX )
            FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     1                    '/' // NAME(:LNBLNK(NAME))
      ENDIF
      RETURN
      END
```

编译并运行 GetFilNam.f，结果如下：

```
demo% pwd
/home/users/auser/subdir
demo% f95 -o getfil GetFilNam.f
demo% getfil
 ENTER FILE NAME:
getfil
 PATH IS: /home/users/auser/subdir/atest.f

demo%
```

这些例程将在第 2-5 页中的 “向程序传递文件名”中进一步说明。有关详细信息，参见与 getarg(3F)、getcwd(3F) 和 getenv(3F) 相应的手册页条目；这些内容以及其它有用的库例程在 *《Fortran 库参考》*中也有介绍。

2.1.2 不用文件名打开文件

OPEN 语句不需要指定名称；运行系统会依据若干惯例提供文件名。

2.1.2.1 打开作为临时文件

在 OPEN 语句中指定 STATUS='SCRATCH'，会打开一个名称形如 tmp.F*AAAxnnnnn* 的文件，其中，*nnnnn* 用当前进程 ID 代替，*AAA* 是三个字符的字符串，*x* 是一个字母；*AAA* 和 *x* 可保证文件名唯一。该文件在程序终止或执行 CLOSE 语句时被删除。当在 FORTRAN 77 兼容模式 (-f77) 下编译时，可以在 CLOSE 语句中指定 STATUS='KEEP' 来保留这个临时文件。（此为非标准扩展。）

2.1.2.2 已打开

如果文件已被程序打开，可以使用后续的 OPEN 语句来更改文件的某些特性；例如 BLANK 和 FORM。此时，只需指定文件的逻辑单元号以及要更改的参数。

2.1.2.3 预连接或隐式命名单元

程序执行开始时，会自动将三个单元号与特定的标准 I/O 文件相关联。这些预连接单元是*标准输入*、*标准输出*以及*标准错误*：

- 标准输入是逻辑单元 5
- 标准输出是逻辑单元 6
- 标准错误是逻辑单元 0

通常，标准输入从工作站键盘接受输入；标准输出和标准错误在工作站屏幕上显示输出。

在其它所有情况下，如果在 OPEN 语句中指定了逻辑单元号而未在 FILE= 后指定任何名称，文件将以形如 fort.*n* 的名称打开，其中 *n* 为逻辑单元号。

2.1.3 不用 OPEN 语句打开文件

在假定使用缺省惯例的情况下，不必非得使用 OPEN 语句。如果逻辑单元上的第一个操作是 I/O 语句，而不是 OPEN 或 INQUIRE，会引用文件 fort.*n*，其中 *n* 为逻辑单元号 （0、5 和 6 除外，它们有特殊意义）。

这些文件无需在程序执行前就存在。如果对文件的第一个操作不是 OPEN 或 INQUIRE 语句，则会创建这些文件。

示例：以下代码中，如果 WRITE 是该单元上的第一个输入 / 输出操作，则会创建文件 fort.25：

```
demo% cat TestUnit.f
      IU=25
      WRITE( IU, '(I4)' ) IU
      END
demo%
```

上述程序将打开文件 fort.25，并将一条格式化记录写入该文件：

```
demo% f95 -o testunit TestUnit.f
demo% testunit
demo% cat fort.25
  25
demo%
```

2.1.4 向程序传递文件名

文件系统没有任何自动便利机制可将 Fortran 程序中的逻辑单元号与物理文件相关联。

但是，有若干种令人满意的方式可将文件名传给 Fortran 程序。

2.1.4.1 通过运行时参数和 GETARG

可以使用库例程 getarg(3F) 在运行时将命令行参数读入一个字符变量。参数会被解释为文件名并在 OPEN 语句的 FILE= 说明符中使用：

```
demo% cat testarg.f
         CHARACTER outfile*40
C  Get first arg as output file name for unit 51
         CALL getarg(1,outfile)
         OPEN(51,FILE=outfile)
         WRITE(51,*) 'Writing to file: ', outfile
         END
demo% f95 -o tstarg testarg.f
demo% tstarg AnyFileName
demo% cat AnyFileName
 Writing to file: AnyFileName
demo%
```

2.1.4.2 通过环境变量和 GETENV

同样，可以使用库例程 getenv(3F) 在运行时将任何环境变量的值读入一个字符变量，该变量随后被解释为文件名：

```
demo% cat testenv.f
         CHARACTER outfile*40
C  Get $OUTFILE as output file name for unit 51
         CALL getenv('OUTFILE',outfile)
         OPEN(51,FILE=outfile)
         WRITE(51,*) 'Writing to file: ', outfile
         END
demo% f95 -o tstenv testenv.f
demo% setenv OUTFILE EnvFileName
demo% tstenv
demo% cat EnvFileName
 Writing to file: EnvFileName
demo%
```

使用 getarg 或 getenv 时，应该注意前导或尾随的空格。（Fortran 95 程序可以使用内在函数 TRIM 或更早的 FORTRAN 77 库例程 LNBLNK()）在本章开头的示例中，可以随 FULLNAME 函数的代码行编写更加灵活的代码来接受相对路径名。

2.1.4.3 命令行 I/O 重定向和管道

将物理文件与程序的逻辑单元号相关联的另一方法是通过重定向或管道输送预连接的标准 I/O 文件。重定向或管道在运行时执行命令中使用。

采用这种方式，读取标准输入 （单元 5）和写至标准输出 （单元 6）或标准错误 （单元 0）的程序可以通过重定向 （在命令行中使用 <、 >、 >>、 >&、 |、 |&、 2>、 2>&1），读或写至其它任何命名文件。

参见下表：

表 2-1 csh/sh/ksh 命令行重定向和管道

操作	使用 C Shell	使用 Bourne 或 Korn Shell
标准输入 — 从 mydata 读入数据	`myprog < mydata`	`myprog < mydata`
标准输出 — 写至 （覆写） myoutput	`myprog > myoutput`	`myprog > myoutput`
标准输出 — 写 / 追加至 myoutput	`myprog >> myoutput`	`myprog >> myoutput`
将标准错误重定向至文件	`myprog >& errorfile`	`myprog 2> errorfile`
将标准输出通过管道输送至另一程序的输入	`myprog1 | myprog2`	`myprog1 | myprog2`
将标准错误和输出通过管道输送至另一程序	`myprog1 |& myprog2`	`myprog1 2>&1 | myprog2`

有关命令行重定向和管道的详细信息，参见 csh、 ksh 和 sh 手册页。

2.2 直接 I/O

直接或随机 I/O 允许直接通过记录号访问文件。记录号在写入记录时分配。与顺序 I/O 不同，直接 I/O 记录可以按任何顺序读写。但是，在直接访问文件中，所有记录必须具有相同的固定长度。直接访问文件用文件 OPEN 语句中的 ACCESS='DIRECT' 说明符声明。

直接访问文件中的逻辑记录是字节字符串，串长度由 OPEN 语句的 RECL= 说明符指定。READ 和 WRITE 语句指定的逻辑记录不能大于所定义的记录大小。（记录大小以字节单位指定。）允许更短的记录。直接写入非格式化数据将使记录的未填写部分仍保持未定义。直接写入格式化数据将使未填写的记录用空格进行填充。

直接访问 READ 和 WRITE 语句另外还有一个参数 REC=*n*，用来指定要读取或写入的记录号。

示例：直接访问，*非格式化*：

```
      OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=200,
&              FORM='UNFORMATTED', ERR=90 )
      READ( 2, REC=13, ERR=30 ) X, Y
```

本程序以直接访问、非格式化 I/O、记录固定长度为 200 字节的方式打开一个文件，然后将第十三条记录读入 X 和 Y。

示例：直接访问，*格式化*：

```
      OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=200,
&              FORM='FORMATTED', ERR=90 )
      READ( 2, FMT='(I10,F10.3)', REC=13, ERR=30 ) X, Y
```

本程序以直接访问、格式化 I/O、记录固定长度为 200 字节的方式打开一个文件。然后读取第十三条记录，并以 (I10,F10.3) 格式对其进行转换。

对于格式化文件，所写记录的大小由 FORMAT 语句确定。在上述示例中，FORMAT 语句所定义的记录大小为 20 个字符或字节。如果列表中的数据总量大于 FORMAT 语句中指定的记录大小，则可通过单条格式化写入指令写入一条以上的记录。在此种情况下，会为随后的每一条记录赋予连续的记录号。

示例：直接访问、格式化、*多记录写入*：

```
      OPEN( 21, ACCESS='DIRECT', RECL=200, FORM='FORMATTED')
      WRITE(21,'(10F10.3)',REC=11) (X(J),J=1,100)
```

写至直接访问单元 21 的写指令会创建 10 条记录，每条记录 10 个元素 （因为格式指定每条记录 10 个元素），这些记录从 11 到 20 进行编号。

2.3 二进制 I/O

Sun Studio Fortran 95 扩展了 OPEN 语句，允许声明 “二进制” I/O 文件。

用 FORM='BINARY' 打开文件与用 FORM='UNFORMATTED' 打开文件所达到的效果大致相同，除了不会在文件中嵌入任何记录长度。没有该数据，将无法判断一条记录的开始或结束位置。因而，不能在 FORM='BINARY' 文件中执行 BACKSPACE，因为无法判断回退的位置。对 'BINARY' 文件执行 READ，将会根据需要读取尽可能多的数据来填充输入列表中的变量。

- WRITE 语句：将数据以二进制形式写入文件，根据输出列表指定的数量传送尽量多的字节。
- READ 语句：将数据读入输入列表中的变量，按列表要求传送尽量多的字节。由于文件中无记录标记，因而不会检测到 “记录结束” 错误。只能检测到 “文件结束” 或系统异常错误。
- INQUIRE 语句：对用 FORM=”BINARY” 打开的文件执行 INQUIRE 会返回以下结果：
  FORM=”BINARY”
  ACCESS=”SEQUENTIAL”
  DIRECT=”NO”
  FORMATTED=”NO”
  UNFORMATTED=”YES”
  RECL= 和 NEXTREC= 未定义
- BACKSPACE 语句：不允许 — 会返回错误。
- ENDFILE 语句：与通常一样在当前位置截断文件。
- REWIND 语句：与通常一样将文件重新定位至数据开头。

2.4 I/O 流

新的 I/O “流”方案已被提议作为 Fortran 2000 标准草案的一部分，并且在 `f95` 中得以实现。 I/O 流式访问将数据文件视作连续的字节序列，用从 1 开始的正整数来寻址。可用 `OPEN` 语句中的 `ACCESS='STREAM'` 说明符来定义 I/O 流文件。字节地址文件定位需要在 `READ` 或 `WRITE` 语句中有 `POS=`*scalar_integer_expression* 说明符。 `INQUIRE` 语句接受 `ACCESS='STREAM'`、说明符 `STREAM=`*scalar_character_variable* 以及 `POS=`*scalar_integer_variable*。

流 I/O 在与 C 程序创建或读取的文件进行互操作时非常有用，如下例所示：

Fortran 95 程序读取由 C `fwrite()` *创建的文件。*

```
program reader
 integer:: a(1024), i, result
 open(file="test", unit=8, access="stream",form="unformatted")
! read all of a
 read(8) a
 do i = 1,1024
   if (a(i) .ne. i-1) print *,'error at ', i
 enddo
! read the file backward
 do i = 1024,1,-1
   read(8, pos=(i-1)*4+1) result
   if (result .ne. i-1) print *,'error at ', i
 enddo
 close(8)
end
```

C 程序写入一个文件

```
#include <stdio.h>
int binary_data[1024];

/* Create a file with 1024 32-bit integers */
int
main(void)
{
    int i;
    FILE *fp;

    for (i = 0; i < 1024; ++i)
        binary_data[i] = i;
    fp = fopen("test", "w");
    fwrite(binary_data, sizeof(binary_data), 1, fp);
    fclose(fp);
}
```

C 程序使用 C `fwrite()` 将 1024 个 32 位整数写入文件中。 Fortran 95 读取程序以数组方式一次读取这些数据，然后再在文件中从后往前分别读取它们。第二条 `read` 语句中的 `pos=` 说明符说明位置是用字节表示的，从字节 1 开始 （这一点与 C 相反，在 C 中，位置从字节 0 开始）。

2.5 内部文件

内部文件是 CHARACTER 类型的对象，如变量、子串、数组、数组元素或者结构记录的字段。内部文件 READ 可以来自*常量*字符串。内部文件 I/O 通过由一个字符对象向另一数据对象传送和转换数据，模拟格式化 READ 和 WRITE 语句。不执行任何文件 I/O。

使用内部文件时：

- 出现在 WRITE 语句中的是接收数据的字符对象的名称而非单元号。在 READ 语句中，出现的也是字符对象源的名称而非单元号。
- 常量、变量或子串对象构成文件中的单条记录。
- 使用数组对象，每个数组元素对应于一条记录。
- 内部文件上的直接 I/O。（Fortran 95 标准只包括内部文件上的顺序格式化 I/O。）除了不能更改文件中的记录数之外，这一点与外部文件上的直接 I/O 相似。此时，记录是字符串数组的单个元素。这项非标准扩展仅在用 -f77 标志进行编译时的 FORTRAN 77 兼容模式下可用。
- 每一顺序 READ 或 WRITE 语句均始于内部文件的开头。

示例：从内部文件 （仅有一条记录）中以顺序、格式化方式进行读取：

```
demo% cat intern1.f
      CHARACTER X*80
      READ( *, '(A)' ) X
      READ( X, '(I3,I4)' ) N1, N2 ! This codeline reads the internal file X
      WRITE( *, * )  N1, N2
      END
demo% f95 -o tstintern intern1.f
demo% tstintern
 12 99
 12 99
demo%
```

示例：从内部文件 （三条记录）中以顺序、格式化方式进行读取：

```
demo% cat intern2.f
      CHARACTER  LINE(4)*16
      DATA  LINE(1) / ' 81  81 ' /
      DATA  LINE(2) / ' 82  82 ' /
      DATA  LINE(3) / ' 83  83 ' /
      DATA  LINE(4) / ' 84  84 ' /
      READ( LINE,'(2I4)') I,J,K,L,M,N
      PRINT *, I, J, K, L, M, N
      END
demo% f95 intern2.f
demo% a.out
   81  81  82  82  83  83
demo%
```

示例：在 -f77 兼容模式下，从内部文件 （一条记录）中以直接访问方式进行读取：

```
demo% cat intern3.f
      CHARACTER LINE(4)*16
      DATA  LINE(1) / ' 81  81 ' /
      DATA  LINE(2) / ' 82  82 ' /
      DATA  LINE(3) / ' 83  83 ' /
      DATA  LINE(4) / ' 84  84 ' /
      READ ( LINE, FMT=20, REC=3 ) M, N
20       FORMAT( I4, I4 )
      PRINT *, M, N
      END
demo% f95 -f77 intern3.f
demo% a.out
   83  83
demo%
```

2.6 其它 I/O 注意事项

Fortran 95 及传统 Fortran 77 程序在 I/O 上是兼容的。包含 `f77` 和 `f95` 混合编译代码的可执行文件可以同时从程序的 `f77` 和 `f95` 部分对同一单元执行 I/O。

但是， Fortran 95 还提供了一些附加功能：

- `ADVANCE='NO'` 允许进行非提前式 I/O，如下所示：

```
      write(*,'(a)',ADVANCE='NO')  'Enter size= '
      read(*,*) n
```

- `NAMELIST` 输入功能：
  - `f95` 允许在输入时在组名前冠以 `$` 或 `&`。 Fortran 95 标准只接受 `&`，并且此为 `NAMELIST` 写语句的输出内容。
  - `f95` 接受 `$` 作为输入组的终止符号，除非组中的最后一个数据项为 `CHARACTER`，此时， `$` 被视为输入数据。
  - `f95` 允许 `NAMELIST` 输入开始于记录的第一列。
- 正如 `f77` 所做的那样， `f95` 承认并实现了 `ENCODE` 和 `DECODE`。

有关 `f95` 与 `f77` 间的 Fortran 95 I/O 扩展及兼容性方面的其它信息，参见 *《Fortran 用户指南》*。

第 3 章

程序开发

本章简要介绍两个功能强大的程序开发工具 make 和 SCCS，这两个工具可以非常成功地用于 Fortran 编程项目。

目前有许多关于使用 make 和 SCCS 的优秀的商业出版书籍，其中包括 *Managing Projects with make*（Andrew Oram 和 Steve Talbott 著）和 *Applying RCS and SCCS*（Don Bolinger 和 Tan Bronson 著）。这两本书均出自 O'Reilly & Associates。

3.1 使用 make 公用程序简化程序构建

make 公用程序可以智能地执行程序编译和链接任务。大型应用程序通常由一组源文件和 INCLUDE 文件组成，同时要求与许多库进行链接。修改任何一个或多个源文件需要重新编译程序的修改部分并重新链接。通过指定组成应用程序的文件间的相互依赖性以及重新编译和重新链接每一程序块所需的命令，可以自动执行这一过程。只要在指令文件中包括这些说明， make 便会确保只重新编译那些需要重新编译的文件，并确保重新链接使用所需的选项和库来生成可行性文件。以下讨论内容提供了一个如何使用 make 的简单示例。有关摘要信息，参见 make(1S)。

3.1.1 Makefile

名为 makefile 的文件以结构化方式告知 make 都有哪些源文件和目标文件依赖其它文件。它还定义了编译和链接文件所需的命令。

例如，假设您的程序有四个源文件以及相应的 makefile 文件：

```
demo% ls
makefile
commonblock
computepts.f
pattern.f
startupcore.f
demo%
```

假设 pattern.f 和 computepts.f 都有一个名为 commonblock 的 INCLUDE，并且您希望编译每个 .f 文件并将这三个可重定位的文件与一系列库一起链接成一个名为 pattern 的程序。

这时， makefile 将会如下所示：

```
demo% cat makefile
pattern: pattern.o computepts.o startupcore.o
      f95 pattern.o computepts.o startupcore.o -lcore95 \
      -lcore -lsunwindow -lpixrect -o pattern
pattern.o: pattern.f commonblock
      f95 -c -u pattern.f
computepts.o: computepts.f commonblock
      f95 -c -u computepts.f
startupcore.o: startupcore.f
      f95 -c -u startupcore.f
demo%
```

makefile 的第一行指示 pattern 的创建取决于 pattern.o、 computepts.o 和 startupcore.o。下一行及其后续各行给出了由可重定位的 .o 文件和库创建 pattern 的命令。

makefile 中的每一条都是一项规则，它描述了目标对象的依赖性以及创建该对象所需的命令。规则的结构为：

target： *dependencies-list*
TAB *build-commands*

- *依赖性*。每条的头一行均是为目标文件命名，其后是目标依赖的所有文件。
- *命令*。每条随后还有一行或多行，这些行指定将生成本条相应的目标文件的 Bourne shell 命令。这些命令行中的每一行都必须用一个制表符缩进。

3.1.2 make 命令

make 命令可以进行无参数调用，只需键入：

```
demo% make
```

make 公用程序在当前目录中寻找名为 makefile 或 Makefile 的文件并从该文件中获取指令。

make 公用程序：

- 读取 makefile，确定其必须处理的所有目标文件、这些目标文件依赖的文件以及生成这些目标文件所需的命令。
- 查找每个文件的最后更改日期和时间。
- 如果有任何目标文件比其依赖的任一文件时间更久，使用 makefile 中与该目标相应的命令重新生成该目标文件。

3.1.3 宏

make 公用程序的宏功能允许进行简单的无参数字符串替换。例如，可将组成目标程序 pattern 的可重定位文件的列表表示为单个宏字符串，使其更易于更改。

宏字符串定义具有以下格式：

NAME = *string*

宏字符串的使用方式如下所示：

$(*NAME*)

make 会用宏字符串的实际值来替换它。

以下示例将命名所有目标文件的宏定义添加到 makefile 的开头：

```
OBJ = pattern.o computepts.o startupcore.o
```

现在便可在依赖性列表以及与 makefile 中的目标 pattern 相应的 f95 链接命令中同时使用宏了。

```
pattern: $(OBJ)
      f95 $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
```

对于名称为单个字母的宏字符串，可以省略括号。

3.1.4 覆盖宏值

make 宏的初始值可以用 make 的命令行选项进行覆盖。例如：

```
FFLAGS=-u
OBJ = pattern.o computepts.o startupcore.o
pattern: $(OBJ)
      f95 $(FFLAGS) $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
pattern.o: pattern.f commonblock
      f95 $(FFLAGS) -c pattern.f
computepts.o:
      f95 $(FFLAGS) -c computepts.f
```

现在，简单的无参数 make 命令将会使用上面设置的 FFLAGS 值。不过，这可以通过命令行来覆盖：

```
demo% make "FFLAGS=-u -O"
```

这里，make 命令行中的 FFLAGS 宏定义会覆盖 makefile 的初始值，并且会将 -O 标志和 -u 标志一起传递给 f95。注意，也可以在命令中使用 "FFLAGS="，将宏重置为空字符串以使其不再有效。

3.1.5 make 中的后缀规则

为使 makefile 更易编写， make 将根据目标文件的后缀，使用自身的缺省规则。

缺省规则在文件 /usr/share/lib/make/make.rules 中。在识别缺省的后缀规则时， make 会将 FFLAGS 宏指定的任何标志、-c 标志以及要编辑的源文件名都作为参数进行传递。此外， make.rules 文件还使用 FC 宏赋予的名称作为要使用的 Fortran 编译器的名称。

以下示例两次说明了这一规则：

```
FC = f95
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern: $(OBJ)
      f95 $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
pattern.o: pattern.f commonblock
      f95 $(FFLAGS) -c pattern.f
computepts.o: computepts.f commonblock
startupcore.o: startupcore.f
```

make 使用缺省规则编译 computepts.f 和 startupcore.f。

.f90 文件存在缺省的后缀规则，这些规则将会调用 f95 编译器。

然而，除非将 FC 宏定义为 f95，否则 .f 和 .F 文件的缺省后缀规则会调用 f77 而非 f95。

而且，当前没有为 .f95 和 .F95 文件定义后缀规则， .mod Fortran 95 模块文件将会调用 Modula 编译器。要对此进行补救，需要在调用 make 的目录下为 make.rules 文件创建您自己的本地副本，同时对该文件进行修改，添加 .f95 和 .F95 后缀规则，删除 .mod 的后缀规则。有关详细信息，参见 make(1S) 手册页。

3.1.6 .KEEP_STATE 与特殊依赖性检查

使用特殊目标 .KEEP_STATE 检查命令的依赖性及隐藏依赖性。

当 .KEEP_STATE: 目标有效时， make 会根据状态文件检查用于生成目标的命令。如果自上次 make 运行以来命令已更改， make 会重新生成此目标。

当 .KEEP_STATE: 目标有效时， make 会从 cpp(1) 以及其它编译处理程序中，读取任何 “隐藏”文件 （如 #include 文件）的相应报告。如果目标相对于这些文件中的任何文件已过期， make 会重新生成它。

3.2 用 SCCS 进行版本跟踪和控制

SCCS 代表*源代码控制系统*。SCCS 为实现以下目标提供了途径：

- 跟踪源文件的演变 — 即其更改历史
- 防止源文件被其它开发人员同时更改
- 通过提供版本标记来跟踪版本号

SCCS 的三项基本操作是：

- 将文件置于 SCCS 控制下
- 签出文件进行编辑
- 签入文件

本部分以上一程序为例向您展示如何使用 SCCS 来执行这些任务。只对基本的 SCCS 进行了说明，并且只介绍了三个 SCCS 命令：`create`、`edit` 和 `delget`。

3.2.1 用 SCCS 控制文件

将文件置于 SCCS 控制下包括以下方面：

- 建立 SCCS 目录
- 在文件中插入 SCCS ID 关键字 （这是可选的）
- 创建 SCCS 文件

3.2.1.1 创建 SCCS 目录

首先，必须在正在开发程序的目录下创建 SCCS 子目录。使用以下命令：

```
demo% mkdir SCCS
```

`SCCS` 必须采用大写字母。

3.2.1.2 插入 SCCS ID 关键字

有些开发者会在每个文件中放入一个或多个 SCCS ID 关键字，但这是可选的。以后，每次用 SCCS `get` 或 `delget` 命令签入文件时，都会用版本号来标识这些关键字。有三种可能的位置可以放置这些字符串：

- 注释行
- 参数语句
- 初始化数据

使用关键字的优点是版本信息会出现在源列表和已编译的目标程序中。如果其前面有字符串 @(#)，可用 what 命令打印目标文件中的关键字。

只含有参数和数据定义语句的已包含头文件不会生成任何初始化数据，因此这些文件的关键字通常置于注释或参数语句中。在某些文件中，如 ASCII 数据文件或 makefile，SCCS 信息将会出现在注释中。

SCCS 关键字以 %*keyword*% 形式出现，并通过 SCCS get 命令扩展成各自的值。最常用的关键字有：

%Z% 扩展为 what 命令识别的标识字符串 @(#)。
%M% 扩展为源文件名。
%I% 扩展为本 SCCS 维护文件的版本号。
%E% 扩展为当前日期。

例如，可以用包含以下关键字的 make 注释来标识 makefile。

```
#       %Z%%M%        %I%        %E%
```

源文件 startupcore.f、 computepts.f 和 pattern.f 可以通过以下格式的初始化数据来标识：

```
      CHARACTER*50 SCCSID
      DATA SCCSID/"%Z%%M%       %I%       %E%\n"/
```

用 SCCS 处理该文件，进行编译，然后用 SCCS what 命令处理目标文件，显示如下：

```
demo% f95 -c pattern.f
...
demo% what pattern
pattern:
      pattern.f 1.2 96/06/10
```

您还可以创建名为 CTIME 的 PARAMETER，无论何时用 get 命令访问文件，该参数都会自动进行更新。

```
      CHARACTER*(*) CTIME
      PARAMETER ( CTIME="%E%")
```

INCLUDE 文件可以用含有 SCCS 标记的 Fortran 注释加以注解：

```
C       %Z%%M%        %I%        %E%
```

注 – 在 Fortran 95 源代码文件中使用单字母派生类型组件名可能会与 SCCS 关键字识别产生冲突。例如，当通过 SCCS 传递时， Fortran 95 结构组件引用 `X%Y%Z` 在执行 SCCS `get` 后会变成 `XZ`。当在 Fortran 95 程序中使用 SCCS 时，应注意不要用单个字母定义结构组件。例如，假如 Fortran 95 程序中的结构引用是 `X%YY%Z`， SCCS 并不会将 `%YY%` 解释为关键字引用。或者， SCCS `get -k` 选项在检索文件时将不会扩展 SCCS 关键字 ID。

3.2.1.3 创建 SCCS 文件

现在，可以用 SCCS `create` 命令将这些文件置于 SCCS 控制之下：

```
demo% sccs create makefile commonblock startupcore.f \
  computepts.f pattern.f
demo%
```

3.2.2 签出和签入文件

一旦源代码处于 SCCS 控制之下，便可用 SCCS 执行以下两项主要任务：*签出*文件以便能对其进行编辑；*签入*已编辑完的文件。

签出文件使用 `sccs edit` 命令。例如：

```
demo% sccs edit computepts.f
```

然后， SCCS 会在当前目录下创建 `computepts.f` 的可写副本，并记录您的登录名。当文件已签出时，其他用户不能再签出该文件，但可以查出是谁签出了该文件。

完成编辑后，可用 `sccs delget` 命令签入已修改的文件。例如：

```
demo% sccs delget computepts.f
```

该命令会使 SCCS 系统做以下事情：

- 通过比较登录名确保您就是签出文件的用户
- 提示您对更改做注释
- 记录本次编辑会话所更改的内容
- 从当前目录中删除 `computepts.f` 的可写副本
- 用扩展了 SCCS 关键字的只读副本替换可写副本

`sccs delget` 命令是两个简单 SCCS 命令（`delta` 和 `get`）的复合命令。`delta` 命令执行上述列表中的前三项任务；`get` 命令执行后两项任务。

第 4 章

库

本章介绍如何使用和创建子程序库。对*静态*和*动态*库均进行了讨论。

4.1 认识库

软件*库*通常是事先已编译并组织成单个二进制*库文件*的子程序集。集中的每个成员称为库*元素*或*模块*。链接程序搜索库文件，在生成可执行二进制程序时加载用户程序所引用的目标模块。有关详细信息，参见 `ld`(1) 和 Solaris *《链接程序和库指南》*。

软件库有两种基本类型：

- *静态库*。该库中的模块在执行*前*即被联编到执行文件中。静态库通常以 `lib`*name*`.a` 命名。 `.a` 后缀指的是*归档*。
- *动态库*。该库中的模块可在运行时联编到可执行程序中。动态库通常以 `lib`*name*`.so` 命名。 `.so` 后缀指的是*共享对象*。

既有静态 (`.a`) 版本又有动态 (`.so`) 版本的典型系统库有：

- Fortran 95 库：`libfsu`、`libfui`、`libfai`、`libfai2`、`libfsumai`、`libfprodai`、`libfminlai`、`libfmaxlai`、`libminvai`、`libmaxvai`、`libifai`、`libf77compat`
- C 库：`libc`

使用库有两个优点：

- 对于程序调用的库例程，不需要有源代码。
- 只加载所需的模块。

库文件为程序共享常用的子例程提供了一条简单途径。只需在链接程序时给出库名便可，那些解析程序中引用的库模块将被链接并合并到可执行文件中。

4.2 指定链接程序调试选项

通过 LD_OPTIONS 环境变量向链接程序传递其它选项，可以获得库用法和加载方面的摘要信息。在生成目标二进制文件时，编译器会用这些选项（以及它要求的其它选项）调用链接程序。

始终建议使用编译器调用链接程序，而不是直接调用链接程序，因为许多编译器选项要求特定的链接程序选项或库引用，缺少这些，链接时会产生无法预料的结果。

示例：使用 LD_OPTIONS 创建加载映射：

```
demo% setenv LD_OPTIONS '-m -Dfiles'
demo% f95 -o myprog myprog.f
```

某些链接程序选项尚有等价的编译器命令行选项，它们可以直接在 f95 命令中出现。这些选项包括 -B*x*、 -d*x*、 -G、 -h*name*、 -R*path* 和 -ztext。有关详细信息，参见 f95(1) 手册页或 *《Fortran 用户指南》*。

在 Solaris *《链接程序和库指南》*中，可以找到链接程序选项和环境变量的更多详细示例和解释。

4.2.1 生成加载映射

链接程序 -m 选项会生成显示库链接信息的加载映射。可执行二进制程序生成期间链接的例程会与其来自的库一起被列出。

示例：使用 -m 生成加载映射：

```
demo% setenv LD_OPTIONS '-m'
demo% f95 any.f
any.f:
 MAIN:
            链接编辑器内存映射

输出     输入     虚拟
区    区  地址          大小

.interp             100d4          11
            .interp 100d4          11 (null)
.hash               100e8          2e8
            .hash    100e8         2e8 (null)
.dynsym             103d0          650
            .dynsym 103d0          650 (null)
.dynstr             10a20          366
            .dynstr 10a20          366 (null)
.text               10c90          1e70
.text               10c90     00 /opt/SUNWspro/lib/crti.o
.text               10c90     f4 /opt/SUNWspro/lib/crt1.o
.text               10d84     00 /opt/SUNWspro/lib/values-xi.o
.text               10d88     d20 sparse.o
…
```

4.2.2 列出其它信息

其它链接程序调试功能可通过链接程序的 -D*keyword* 选项获得。使用 -Dhelp 选项可以显示完整的列表。

示例：使用 -Dhelp 选项列出链接程序调试辅助选项：

```
demo% ld -Dhelp
      …
debug: args             显示输入参数处理
debug: bindings         显示符号绑定;
debug: detail           提供详细信息
debug: entry            显示入口标准描述符
      …
demo%
```

例如， -Dfiles 链接程序选项会列出链接过程中引用的所有文件和库：

```
demo% setenv LD_OPTIONS '-Dfiles'
demo% f95 direct.f
direct.f:
 MAIN direct:
debug: file=/opt/SUNWspro/lib/crti.o  [ ET_REL ]
debug: file=/opt/SUNWspro/lib/crt1.o  [ ET_REL ]
debug: file=/opt/SUNWspro/lib/values-xi.o  [ ET_REL ]
debug: file=direct.o  [ ET_REL ]
debug: file=/opt/SUNWspro/lib/libM77.a  [ archive ]
debug: file=/opt/SUNWspro/lib/libF77.so  [ ET_DYN ]
debug: file=/opt/SUNWspro/lib/libsunmath.a  [ archive ]
      …
```

有关这些链接程序选项的更为详细的信息，参见 *《链接程序和库指南》*。

4.2.3 编译和链接一致性

每当分步完成编译和链接时，确保编译和链接选项的一致选择是至关重要的。用某些选项编译程序的任何部分均需要使用相同的选项进行链接。另外，许多选项要求使用该选项编译*所有*源文件，*包括*链接步骤。

*《Fortran 用户指南》*中的选项说明具体指出了此类选项。

示例：用 -ast 编译 sbr.f，编译 C 例程，然后进行分步链接：

```
demo% f95 -c -fast sbr.f
demo% cc -c -fast simm.c
demo% f95 -fast sbr.o simm.o          链接步骤；将 -fast 传递给链接程序
```

4.3 设置库搜索路径和顺序

链接程序按某一规定顺序在若干位置搜索库。这些位置中有一些是标准路径，有一些则取决于编译器选项 -R*path*、 -l*library*、 -L*dir* 以及环境变量 LD_LIBRARY_PATH。

4.3.1 标准库路径的搜索顺序

链接程序所用的标准库搜索路径由安装路径确定，对于静态和动态加载，它们会有所不同。标准安装将 Sun Studio 编译器软件置于 /opt/SUNWspro/ 下。

4.3.1.1 静态链接

生成可执行文件时，静态链接程序按指定顺序、在以下路径 （夹在其它路径中）中搜索任何可能有的库：

/opt/SUNWspro/lib	Sun Studio 共享库
/usr/ccs/lib/	SVr4 软件的标准位置
/usr/lib	UNIX 软件的标准位置

这些是链接程序所用的*缺省*路径。

4.3.1.2 动态链接

动态链接程序在运行时按指定顺序搜索*共享*库：

- 用户使用 -R*path* 指定的路径
- /opt/SUNWspro/lib/
- /usr/lib 标准 UNIX 缺省值

这些搜索路径被内置于可执行文件中。

4.3.2 LD_LIBRARY_PATH 环境变量

使用 LD_LIBRARY_PATH 环境变量指定链接程序应在哪些目录路径中搜索用 -l*library* 选项指定的库。

可以指定多个目录，其间用冒号分隔。通常， LD_LIBRARY_PATH 变量包含两个用冒号分隔的目录列表，列表间用分号隔开：

dirlist1:*dirlist2*

首先搜索 *dirlist1* 中的目录，接着是命令行上用任何显式 -L*dir* 指定的目录，再接着是 *dirlist2* 以及标准目录。

也就是说，如果以任意多次的 -L 调用编译器，如下所示：

f95 ... -L*path1* ... -L*pathn* ...

则搜索顺序是：

dirlist1 path1 ... pathn dirlist2 standard_paths

当 LD_LIBRARY_PATH 变量只包含一个用冒号分隔的目录列表时，它会被解释为 *dirlist2*。

在 Solaris 操作环境中，在搜索 64 位依赖性时，可以用相似的环境变量 LD_LIBRARY_PATH_64 来替代 LD_LIBRARY_PATH。有关详细信息，参见 Solaris *《链接程序和库指南》*以及 ld(1) 手册页。

- 在 32 位 SPARC 处理器上，会忽略 LD_LIBRARY_PATH_64。
- 如果只定义了 LD_LIBRARY_PATH，它将被同时用于 32 位和 64 位链接。
- 如果同时定义了 LD_LIBRARY_PATH 和 LD_LIBRARY_PATH_64，则 32 位链接将用 LD_LIBRARY_PATH 来完成，而用 LD_LIBRARY_PATH_64 进行 64 位链接。

注 – 强烈建议不要对效率型软件使用 LD_LIBRARY_PATH 环境变量。尽管它作为一种影响运行时链接程序搜索路径的临时机制很有用，但是*任何*可以引用该环境变量的动态可执行程序的搜索路径都会被改变。您可能会看到了意想不到的结果或性能降低。

4.3.3 库搜索路径和顺序 — 静态链接

使用 -l*library* 编译器选项给出链接程序在解析外部引用时要搜索的其它库名。例如，用选项 -lmylib 将库 libmylib.so 或 libmylib.a 添加到搜索列表中。

链接程序会在标准目录路径中寻找其它的 libmylib 库。 -L 选项 （和 LD_LIBRARY_PATH 环境变量）会创建一个路径列表，告知链接程序到哪里寻找位于标准路径以外的库。

假如 libmylib.a 在目录 /home/proj/libs 中，则选项 -L/home/proj/libs 会告知链接程序在生成可执行文件时到哪里寻找：

```
demo% f95 -o pgram part1.o part2.o -L/home/proj/libs -lmylib
```

4.3.3.1 -l*library* 选项的命令行顺序

对于任何未解析的特殊引用，只对库进行一次搜索，并且只搜索其时在搜索中未定义的符号。如果在命令行上列出了多个库，则会按其在命令行上出现的顺序来搜索这些库。将 -l*library* 选项放置在以下位置：

- 将 -l*library* 选项置于任一 .f、 .for、 .F、 .f95 或 .o 文件之后。
- 如果调用了 lib*x* 中的函数，并且这些函数引用了 lib*y* 中的函数，则将 -l*x* 置于 -l*y* 之前。

4.3.3.2 -L*dir* 选项的命令行顺序

-L*dir* 选项会将 *dir* 目录路径添加到库搜索列表中。链接程序首先在 -L 选项指定的任何目录中搜索库，然后在标准目录中进行搜索。只有将其放在它所应用的 -l*library* 选项*之前*，该选项才有用。

4.3.4 库搜索路径和顺序 — 动态链接

对于动态库，库搜索路径和加载顺序的更改与静态情况不同。实际链接发生在运行时而不是生成时。

4.3.4.1 在生成时指定动态库

*生成*可执行文件时，链接程序会在可执行文件本身中记录共享库的路径。这些搜索路径可以用 -R*path* 选项指定。这一点与 -L*dir* 选项相反，该选项在生成时指示到哪里查找 -l*library* 选项所指定的库，但不会将该路径记录到二进制可执行文件中。

使用 dump 命令可以查看创建可执行文件时内置的目录路径。

示例：列出内置于 a.out 之中的目录路径：

```
demo% f95 program.f -R/home/proj/libs -L/home/proj/libs -lmylib
demo% dump -Lv a.out | grep RPATH
[5]      RPATH    /home/proj/libs:/opt/SUNWspro/lib
```

4.3.4.2 在运行时指定动态库

在*运行时*，链接程序会确定到哪里查找可执行文件所需的动态库：

- 运行时的 LD_LIBRARY_PATH 值
- 生成可执行文件时已由 -R 指定的路径

如前所述，使用 LD_LIBRARY_PATH 能带来意想不到的副作用，因而不建议这样做。

4.3.4.3 修复动态链接期间的错误

当动态链接程序找不到所需库的位置时，它会发出以下错误消息：

```
ld.so: prog: 致命: libmylib.so: 无法打开文件:
```

此消息指示库不在其应在的位置。您也许在生成可执行文件时指定了共享库的路径，但这些库随后已被移动。例如，您可能先用 */my/libs/* 中您自己的动态库生成了 a.out，而后来又将这些库移到了另一目录。

使用 ldd 确定可执行文件期望在哪儿找到这些库：

```
demo% ldd a.out
libfui.so.1 =>   /opt/SUNWspro/lib/libfui.so.1
    libfai.so.1 =>   /opt/SUNWspro/lib/libfai.so.1
    libfai2.so.1 =>  /opt/SUNWspro/lib/libfai2.so.1
    libfsumai.so.1 =>    /opt/SUNWspro/lib/libfsumai.so.1
    libfprodai.so.1 =>   /opt/SUNWspro/lib/libfprodai.so.1
    libfminlai.so.1 =>   /opt/SUNWspro/lib/libfminlai.so.1
    libfmaxlai.so.1 =>   /opt/SUNWspro/lib/libfmaxlai.so.1
    libfminvai.so.1 =>   /opt/SUNWspro/lib/libfminvai.so.1
    libfmaxvai.so.1 =>   /opt/SUNWspro/lib/libfmaxvai.so.1
    libfsu.so.1 =>   /opt/SUNWspro/lib/libfsu.so.1
    libsunmath.so.1 =>   /opt/SUNWspro/lib/libsunmath.so.1
    libm.so.1 =>     /usr/lib/libm.so.1
    libc.so.1 =>     /usr/lib/libc.so.1
    libdl.so.1 =>    /usr/lib/libdl.so.1
    /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
```

如果可能的话，将这些库移动或复制到正确的目录中，或者在链接程序搜索的目录中建立到该目录的软链接（使用 ln -s）。或者，也可能是没有正确设置 LD_LIBRARY_PATH。检查 LD_LIBRARY_PATH 是否包含运行时所需库的路径。

4.4 创建静态库

静态库文件是使用 ar(1) 公用程序由预编译的目标文件 （o 文件）生成的。

链接程序从库中提取在当前链接的程序内引用了其入口点的任何元素，如子程序、入口名或 BLOCKDATA 子程序中已初始化的 COMMON 块。这些提取出来的元素 （例程）会被永久联编到链接程序生成的 a.out 可执行文件中。

4.4.1 权衡静态库

与动态情况相比，关于静态库和链接，有三个主要问题需要谨记：

- 静态库更加自主，但适应能力较差。

  如果以*静态*方式联编 a.out 可执行文件，它所需的库例程会变成可执行二进制文件的一部分。但是，如果需要更新联编到 a.out 可执行文件中的静态库例程，则必须重新链接并重新生成整个 a.out 文件以利用已更新的库。对于*动态*库，库并不是 a.out 文件的一部分，并且链接是在运行时完成的。要利用已更新的动态库，只需将新库安装在系统中即可。

- 静态库中的 “元素”是单独的编译单元，即 .o 文件。

  由于单个编译单元 （源文件）可以包含多个子程序，因此这些例程在一起编译时会变成静态库中的单一模块。这就意味着会将编译单元中的*所有*例程一起装入 a.out 可执行文件中，即使实际只调用了那些子程序中的一个。通过优化库例程分发到可编译源文件中的方式，可以改善这种情况。（尽管如此，只有程序实际引用的那些库模块才会被装入可执行文件。）

- 链接静态库时，顺序很重要。

  链接程序按输入文件在命令行上出现的顺序对其进行处理 — 从左至右。当链接程序决定是否从库中加载某一元素时，其决定取决于它已经处理的库元素。该顺序不仅依赖于元素在库文件中的出现顺序，而且还依赖于编译命令行中指定库的顺序。

示例：如果 Fortran 程序在两个文件 （main.f 和 crunch.f）中，并且只有后者访问某个库，则在 crunch.f 或 crunch.o 之前引用该库是错误的：

```
demo% f95 main.f -lmylibrary crunch.f -o myprog
      （不正确）
demo% f95 main.f crunch.f -lmylibrary -o myprog
      （正确）
```

4.4.2 简单静态库的创建

假设您可以将程序中的所有例程分布在一组源文件中，同时假定这些文件全部包含在子目录 test_lib/ 中。

进一步假定这些文件是以这样一种方式组织的：它们每一个都只包含一个用户程序将会调用的主要子程序，同时还包含该子程序可能会调用的任何 “帮助程序”例程，但这些例程不会从库中的任何其它例程中调用。另外，从一个以上库例程中调用的任何帮助程序例程均被集合到单个源文件中。这样就给出了一个组织得非常合理的源文件及目标文件集。

假定每个源文件的名称均取自文件中第一个例程的名称，在多数情况下，该例程是库中的主要文件之一：

```
demo% cd test_lib
demo% ls
总数 14
   2 delte.f       2 etc.f         2 linkz.f       2 point.f
```

更低级的 “帮助程序”例程被集合到文件 etc.f 中。其它文件可以包含一个或多个子程序。

首先，使用 -c 选项编译每一个库源文件，生成相应的可重定位的 .o 文件：

```
demo% f95 -c *.f
demo% ls
总数 42
 2 dropx.f     4 etc.o      2 linkz.f    4 markx.o
 2 delte.f     4 dropx.o    2 evalx.f    4 linkz.o     2 point.f
 4 delte.o     2 etc.f      4 evalx.o    2 markx.f     4 point.o
demo%
```

现在，使用 ar 创建静态库 testlib.a：

```
demo% ar cr testlib.a *.o
```

要使用该库，或者在编译命令中包括此库文件，或者使用 -l 和 -L 编译选项。以下示例直接使用 .a 文件：

```
demo% cat trylib.f
C    program to test testlib routines
            x=21.998
            call evalx(x)
            call point(x)
            print*, 'value ',x
            end
demo% f95 -o trylib trylib.f test_lib/testlib.a
demo%
```

注意，主程序只调用库中的两个例程。您可以验证并未将库中未调用的例程装入可执行文件，方法是查找用 nm 显示的可执行文件的名称列表中是否有这些例程。

```
demo% nm trylib | grep FUNC | grep point
[146]       |     70016|     152|FUNC |GLOB |0    |8       |point_
demo% nm trylib | grep FUNC | grep evalx
[165]       |     69848|     152|FUNC |GLOB |0    |8       |evalx_
demo% nm trylib | grep FUNC | grep delte
demo% nm trylib | grep FUNC | grep markx
demo% ..etc
```

在上述示例中， grep 只在名称列表中查找与实际调用的那些库例程相应的项。

引用库的另一方法是通过 -l*library* 和 -L*path* 选项。这里，必须更改库的名称以符合 lib*name*.a 惯例：

```
demo% mv test_lib/testlib.a test_lib/libtestlib.a
demo% f95 -o trylib trylib.f -Ltest_lib -ltestlib
```

-l*library* 和 -L*path* 选项与安装在系统可公共访问目录 （如 /usr/local/lib）中的库一起使用，以便其他用户可以引用它。例如，假如将 libtestlib.a 留在 /usr/local/lib 中，可以通知其他用户使用以下命令编译：

```
demo% f95 -o myprog myprog.f -L/usr/local/lib -ltestlib
```

4.4.2.1 静态库中的替换

如果仅有几个元素需要重新编译，没有必要重新编译整个库。 ar 的 -r 选项允许替换静态库中的个别元素。

示例：重新编译并替换静态库中的单个例程：

```
demo% f95 -c point.f
demo% ar -r testlib.a point.o
```

4.4.2.2 对静态库中的例程进行排序

要在 ar 正在生成静态库时对其中的元素进行排序，请使用命令 lorder(1) 和 tsort(1)：

```
demo% ar -cr mylib.a 'lorder exg.o fofx.o diffz.o | tsort'
```

4.5 创建动态库

动态库文件是由链接程序 `ld` 自预编译目标模块生成的，这些模块可在执行开始*后*联编到可执行文件中。

动态库的另一功能是模块可以为系统中其它正在执行的程序使用，而*无需*在每个程序的内存中复制模块。鉴于此原因，动态库也是*共享*库。

动态库提供下列功能：

- 链接程序在编译链接过程中*并不*将目标模块联编到可执行文件中；此种联编被推迟到了运行时。
- 共享库模块在第一个运行程序引用它时联编到系统内存中。如果有任何后续运行程序引用它，会将该引用映射到上述第一个副本。
- 使用动态库，程序维护变得更加容易。一旦在系统中安装了已更新的动态库，无需重新链接可执行文件便会立即影响使用它的所有应用程序。

4.5.1 权衡动态库

动态库引入了其它一些权衡考虑因素：

- `a.out` 文件更小

  将库例程联编推迟到运行时意味着可执行文件的大小要小于同等意义上调用库静态版本的可执行文件；该可执行文件不包含库例程的二进制文件。

- 进程占用的内存可能更少

  当使用库的若干进程同时处于活动状态时，仅有库的一个副本驻留在内存中，为所有进程所共享。

- 有可能增加系统开销

  运行时加载和链接编辑库例程需要额外的处理器时间。另外，库中与位置无关的编码可能要比静态库中可重定位的编码执行得更慢。

- 有可能提高系统总体性能提高

  库共享可减少内存占用，其结果将会改善系统的总体性能 （减少了内存交换时的 I/O 访问时间）。

各程序间的性能特征随程序的不同会有很大变化。并非总能预先判断或估计动态库与静态库相比性能会提高 （还是降低）。但是，如果所需库的这两种形式对您都可用，则分别评估一下程序使用每个库时的性能还是很值得的。

4.5.2 位置无关代码和 -xcode

可以将 *与位置无关的代码* (PIC) 联编到程序中的任何地址，而无需由链接编辑器进行重定位。从固有性质出发，此类代码可以在同时发生的进程间共享。因而，如果要生成动态共享库，必须使用 -xcode 编译器选项将组成例程编译成与位置无关的。

在与位置无关的代码中，对全局项的每一引用均会通过全局偏移表中的指针编译为某一引用。每个函数调用均会通过过程链接表以相对编址模式进行编译。在 SPARC 处理器上，全局偏移表的大小限制为 8 K 字节。

编译器标志 -xcode=*v* 用于指定二进制对象的代码地址空间。使用该标志，不但可以生成 32、 44 或 64 位绝对地址，而且可生成大小不同模型与位置无关的代码。（-xcode=pic13 等价于传统的 -pic 标志， -xcode=pic32 等价于 -PIC。）

-xcode=pic32 编译器选项与 -xcode=pic13 类似，但允许全局偏移表跨越 32 位地址范围。有关详细信息，参见 f95(1) 手册页或 *《Fortran 用户指南》*。

4.5.3 联编选项

可以在编译时指定动态或静态库联编。这些选项实际上是链接程序选项，但它们是由编译器识别并传递给链接程序的。

4.5.3.1 -Bdynamic | -Bstatic

-Bdynamic 用于在各种可能的情况下为共享动态联编设置首选项。
-Bstatic 将联编只限制于静态库。

当库的静态和动态版本都可用时，使用该选项在命令行首选项间进行切换：

```
f95 prog.f -Bdynamic -lwells -Bstatic -lsurface
```

4.5.3.2 -dy | -dn

允许或不允许对整个可执行文件进行动态链接。（该选项只能在命令行上出现一次。）

-dy 允许链接动态共享库。 -dn 不允许链接动态库。

4.5.3.3 64 位环境中的联编

某些静态系统库 （如 libm.a 和 libc.a）不可以在 64 位 Solaris 操作环境中使用。这些库只作为动态库提供。在这些环境中使用 -dn 将会导致错误，指示缺少某些静态系统库。另外，如果编译器命令行以 -Bstatic 结尾，其结果将是一样的。

要与特定库的静态版本进行链接，请使用类似下面的命令行：

```
f95 -o prog prog.f -Bstatic -labc -lxyz -Bdynamic
```

在此，链接的是用户的 libabc.a 和 libxyz.a 文件 （而不是 libabc.so 或 libxyz.so），最后的 -Bdynamic 确保以动态方式链接包括系统库在内的其余各库。

在更复杂的情况下，可能必须在链接阶段根据需要用相应的 -Bstatic 或 -Bdynamic 显式引用每个系统库和用户库。首先使用设置为 '-Dfiles' 的 LD_OPTIONS 获得全部所需库的列表。然后用 -nolib 执行链接步骤 （禁止自动链接系统库）并显式引用所需的库。例如：

```
f95 -xarch=v9 -o cdf -nolib cdf.o -Bstatic -lsunmath \
-Bdynamic -lm -lc
```

4.5.4 命名惯例

为符合链接加载程序和编译器假定的动态库命名惯例，请为您使用前缀 lib 和后缀 .so 创建的动态库命名。例如，编译器选项 –lmyfavs 可以引用 libmyfavs.so。

链接程序还接受可选的版本号后缀：例如， libmyfavs.so.1 代表库的*第一*版。

编译器的 –h*name* 选项将 *name* 记录为正在生成的动态库的名称。

4.5.5 一个简单动态库

生成动态库需要用 –xcode 选项和链接程序选项 –G、 –ztext 和 –h*name* 编译源文件。这些链接程序选项可通过编译器命令行来提供。

您可以用静态库示例中使用的相同文件创建一个动态库。

示例：用 –pic 和其它链接程序选项编译：

```
demo% f95 -o libtestlib.so.1 -G -xcode=pic13 -ztext  \
-hlibtestlib.so.1 *.f
```

–G 告知链接程序生成一个动态库。

–ztext 会在发现与位置无关的代码以外的任何内容 （如可重定位文本）时发出警告。

示例：使用动态库生成可执行文件 a.out：

```
demo% f95 –o trylib -R`pwd` trylib.f libtestlib.so.1
demo% file trylib
trylib: ELF 32 位 MSB 可执行 SPARC 版本 1，已动态链接，未剥离
demo% ldd trylib
     libtestlib.so.1 => /export/home/U/Tests/libtestlib.so.1
     libfui.so.1 => /opt/SUNWspro/lib/libfui.so.1
     libfai.so.1 => /opt/SUNWspro/lib/libfai.so.1
     libc.so.1 => /usr/lib/libc.so.1
```

注意，此示例使用 -R 选项将动态库路径 （当前目录）联编到可执行文件中。

file 命令显示可执行文件是以动态方式链接的。

4.5.6 初始化公共块

生成动态库时，通过将已初始化的公共块集合到同一库中并在其它所有库之前引用该库，确保正确初始化公共块 （用 DATA 或 BLOCK DATA 表示的块）。

例如：

```
demo% f95 -G -xcode=pic32 -o init.so blkdat1.f blkdat2.f blkdat3.f
demo% f95 -o prog main.f init.so otherlib1.so otherlib2.so
```

首次编译会由定义公共块并在 BLOCK DATA 单元中对其进行初始化的文件创建一个动态库。第二次编译创建可执行二进制文件，将已编译的主程序与应用程序所需的动态库链接起来。注意，初始化所有公共块的动态库在其它所有库之前首先出现。这样将确保正确地初始化这些块。

4.6 随 Sun Fortran 编译器提供的库

下表展示了随编译器一同安装的库。

表 4-1 随编译器提供的主要库

库	名称	所需选项
f95 内在支持	libfsu	无
f95 接口	libfui	无
f95 数组内在库	libf*ai	无
f95 区间运算内在库	libifai	-xinterval
Sun 数学函数库	libsunmath	无

4.7 可发送库

如果您的可执行文件使用了 runtime.libraries 自述文件中列出的某个 Sun 动态库，则您的许可证包括将该库重新分发给客户的权利。

该自述文件位于 READMEs 目录：

/opt/SUNWspro/READMEs/

请勿以任何形式重新分发或透露头文件、源代码、目标模块或目标模块的静态库。

有关更多详细信息，请参阅您的软件许可证。

第 5 章

程序分析和调试

本章介绍了许多有利于程序分析和调试的编译器功能。

5.1 全局程序检查 (-Xlist)

-Xlist 选项为分析源程序中的不一致性及可能存在的运行时问题提供了一条颇有价值的途径。编译器执行的分析是*全局*性的，跨越各个子程序。

-Xlist 报告子程序变量、公共块、参数在对齐、数值与类型一致性方面的错误，以及其它各种错误。

-Xlist 还可用来生成详细的源代码列表和交叉引用表。

用 -Xlist 选项编译的程序会自动将其分析数据内置于二进制文件中。这样便能对库中的程序执行全局程序检查。

5.1.1 GPC 概述

全局程序检查 (GPC)（由 -Xlist*x* 选项调用）执行下列任务：

- 比通常更为严格地强制执行 Fortran 类型检查规则，特别是在单独编译的例程之间
- 强制执行在不同机器或操作系统之间转移程序所需的一些可移植性限制
- 检测仍有可能未达到最佳或易于出错的合法构造
- 揭示其它潜在的错误和含混不清之处

特别地，全局检查会报告如下问题：

- 接口问题
  - 伪参数和实参数的数量与类型间的冲突

- 函数值的错误类型
- 因不同子程序间公共块中的数据类型不匹配而引起的可能冲突

- 使用问题
  - 用作子例程的函数或用作函数的子例程
  - 已声明但未使用的函数、子例程、变量以及标签
  - 已引用但未声明的函数、子例程、变量以及标签
  - 未设置变量的使用
  - 执行不到的语句
  - 隐式类型变量
  - 已命名公共块的长度、名称和布局的不一致性

5.1.2 如何调用全局程序检查

命令行中的 `-Xlist` 选项用于调用编译器的全局程序分析器。该选项有许多子选项，分别在以下各部分进行说明。

示例：为基本全局程序检查编译以下三个文件：

```
demo% f95 -Xlist  any1.f  any2.f  any3.f
```

在上述示例中，编译器：

- 在文件 `any1.lst` 中产生输出列表
- 在无错误时编译并链接程序

5.1.2.1 屏幕输出

通常会将 `-Xlist`*x* 产生的输出列表写到文件中。要直接显示到屏幕上，请使用 `-Xlisto` 将输出文件写到 `/dev/tty`。

示例：显示到终端：

```
demo% f95 -Xlisto /dev/tty  any1.f
```

5.1.2.2 缺省输出功能

`-Xlist` 选项提供了可用于输出的功能组合。不使用其它 `-Xlist` 选项，缺省情况下会获得以下结果：

- 列表文件名取自出现的第一个输入源文件或目标文件，同时扩展名代之以 `.lst`
- 编有行号的源码列表

- 描述例程间不一致性的错误消息 （嵌入在列表中）
- 标识符的交叉引用表
- 以每页 66 行、每行 79 列编页码
- 无调用图
- 不扩展 `include` 文件

5.1.2.3 文件类型

检查进程可识别编译器命令行中以 `.f`、`.f90`、`.f95`、`.for`、`.F`、`.F95` 或 `.o` 结尾的所有文件。 `.o` 文件仅向进程提供与全局名称 （如子例程和函数名）有关的信息。

5.1.3 -Xlist 和全局程序检查的一些示例

此处列出了下列示例中使用的 Repeat.f 源代码：

```
demo% cat Repeat.f
       PROGRAM repeat
         pn1 = 27.005
         CALL subr1 ( pn1 )
         CALL newf ( pn1 )
         PRINT *, pn1
       END

       SUBROUTINE subr1 ( x )
         IF ( x .GT. 1.0 ) THEN
          CALL subr2 ( x * 0.5 )
         END IF
       END

       SUBROUTINE newf( ix )
         INTEGER PRNOK
         IF (ix .eq. 0) THEN
                ix = -1
         ENDIF
         PRINT *, prnok ( ix )
       END

       INTEGER FUNCTION prnok ( x )
         prnok = INT ( x ) + .05
       END

       SUBROUTINE unreach_sub()
         CALL sleep(1)
       END

       SUBROUTINE subr2 (x)
         CALL subr1(x+x)
       END
```

示例：使用 -XlistX 显示错误、警告和交叉引用

```
demo% f95 -XlistX Repeat.f
demo% cat Repeat.lst
Repeat.f                2002 年 3 月 18 日星期一 18:08:27    第 1 页

文件  "Repeat.f"
程序  repeat
     4            CALL newf ( pn1 )
                               ^
**** ERR  #418:   参数 "pn1" 是 real，但伪参数是 integer
                  参见: "Repeat.f" 第 14 行
     5            PRINT *, pn1
                           ^
**** ERR  #570:   变量 "pn1" 作为 real 引用但在下行被设置为 integer
                  第 4 行
子例程  newf
    19            PRINT *, prnok  ( ix )
                                    ^
**** ERR  #418:   参数 "ix" 是 integer，但伪参数是 real
                  参见: "Repeat.f" 第 22 行
函数  prnok
    23            prnok = INT ( x ) + .05
                  ^
**** WAR  #1024:   "real*4" 类型的值赋值给
                  "integer*4" 类型的变量是可疑的
子例程  unreach_sub
    26          SUBROUTINE unreach_sub()
                           ^
**** WAR  #338:   子例程 "unreach_sub" 从未自程序中调用
子例程  subr2
    31            CALL subr1(x+x)
                  ^
**** WAR  #348:   "subr1" 的递归调用。请参阅动态调用:
                  "Repeat.f" 第 10 行
                  "Repeat.f" 第 3 行

交叉引用                2002 年 3 月 18 日星期一 18:08:27    第 2 页




          交叉引用表

  源文件:    Repeat.f
```

```
图例:
D          定义 / 声明
U          简单使用
M          修改的事件
A          实际参数
C          子例程 / 函数调用
I          初始化:  数据或扩展声明
E          EQUIVALENCE 中事件
N          NAMELIST 中事件
L          使用模块


交叉引用                    2002 年 3 月 18 日星期一 15:40:57     第 3 页

程序形式

 程序
 -------

repeat           <repeat>         D      1:D

交叉引用                    2002 年 3 月 18 日星期一 15:40:57     第 4 页


 函数和子例程
 -------------------------

INT       固有
                 <prnok>           C     23:C

newf             <repeat>          C      4:C
                 <newf>            D     14:D

prnok     int*4  <newf>           DC     15:D     19:C
                 <prnok>          DM     22:D     23:M

sleep            <unreach_sub>              C     27:C

subr1            <repeat>          C      3:C
                 <subr1>           D      8:D
                 <subr2>           C     31:C

subr2            <subr1>           C     10:C
                 <subr2>           D     30:D

unreach_sub      <unreach_sub>              D     26:D
```

```
交叉引用                    2002 年 3 月 18 日星期一 15:40:57     第 5 页


 变量和数组
 --------------------

ix       int*4  伪参数
                <newf>       DUMA     14:D     16:U     17:M     19:A

pn1      real*4 <repeat>      UMA      2:M      3:A      4:A      5:U

x        real*4 伪参数
                 <subr1>         DU       8:D      9:U      10:U
                 <subr2>         DU      30:D     31:U      31:U
                 <prnok>         DA      22:D     23:A

------------------------------------------------------------------
-----


统计                        2002 年 3 月 18 日星期一 15:40:57     第 6 页


日期:       2002 年 3 月 18 日星期一 15:40:57
选项:   -XlistX
文件:           2 （源: 1 ; 库: 1）
行:           33 （源: 33 ; 库子程序: 1）
例程:         6 （MAIN: 1 ; 子例程: 4 ; 函数: 1）
消息:         6 （错误: 3 ; 警告: 3）
```

5.1.4 跨例程全局检查的子选项

基本的全局交叉检查选项是不带子选项的 `-Xlist`。它是子选项的组合，其中的每一项都可以单独指定。

以下部分介绍用于产生列表、错误或交叉引用表的选项。命令行中可以出现多个子选项。

5.1.4.1 子选项语法

按下列规则添加子选项：

- 将子选项添加到 `-Xlist` 的末尾。
- 不要在 `-Xlist` 和子选项间置入空格。

- 每个 -Xlist 只使用一个子选项。

5.1.4.2 -Xlist 及其子选项

按下列规则合并子选项：

- 最常用的选项是 -Xlist （列表、错误、交叉引用表）。
- 使用 -Xlistc、 -XlistE、 -XlistL 或 -XlistX 可以合并特定的功能。
- 其它子选项进一步指定其它细节。

示例：以下两个命令行中的每一个执行相同的任务：

```
demo% f95  -Xlistc  -Xlist  any.f
```

```
demo% f95  -Xlistc  any.f
```

下表展示单独由这些基本的 -Xlist 子选项生成的报告：

表 5-1 **基本的** Xlist 子选项

生成的报告	选项
错误、列表、交叉引用	-Xlist
仅错误	-Xlist**E**
仅错误以及源码列表	-Xlist**L**
仅错误以及交叉引用表	-Xlist**X**
仅错误以及调用图	-Xlist**c**

下表展示所有 -Xlist 子选项。

表 5-2 -Xlist 子选项的完整列表

选项	操作
-Xlist （*无子选项*）	显示错误、列表和交叉引用表
-Xlist**c**	显示调用图和错误 单独使用时， -Xlistc 不显示列表或交叉引用。它使用可打印字符以树的形式产生调用图。如果某些子例程未自 MAIN 中调用，会显示一个以上的图。单独打印每一个 BLOCKDATA，不连接到 MAIN。 缺省时*不*显示调用图。
-Xlist**E**	显示错误 单独使用时， -XlistE 只显示跨例程错误而不显示列表或交叉引用。
-Xlist**err**[*nnn*]	在检验报告中禁止错误 *nnn* 可使用 -Xlisterr 禁止来自列表或交叉引用的编号错误信息。 例如：-Xlisterr338 禁止错误消息 338。要禁止其它特定的错误，可重复使用该选项。如果未指定 *nnn*，会禁止所有错误消息。
-Xlist**f**	更快地产生输出 可使用 -Xlistf 产生源文件列表和交叉检查报告，并在未完全编译的情况下检查源码。
-Xlist**h**	显示来自交叉检查停止编译的错误 使用 -Xlisth，如果在交叉检查程序时检测到错误，编译将会停止。此时，会将报告重定向到 stdout 而非 *.lst 文件。
-Xlist**I**	列表和交叉检查 include 文件 如果 -XlistI 是唯一使用的子选项，会随 -Xlist 标准输出 （行编号列表、错误消息和交叉引用表）一同显示或扫描 include 文件。 *列表* — 如果未禁止列表，则会在适当位置列出 include 文件。文件会按其被包含的次数列出。这些文件是：源文件、 #include 文件、 INCLUDE 文件 *交叉引用表* — 如果未禁止交叉引用表，会在生成交叉引用表时扫描下列所有文件：源文件、 #include 文件、 INCLUDE 文件 缺省时不显示 include 文件。
-Xlist**L**	显示列表和错误 使用 -XlistL 仅产生列表和跨例程错误列表。该子选项本身并不显示交叉引用表。缺省时显示列表和交叉引用表
-Xlist**l***n*	设置分页符 可使用 -Xlistl 将页长度设置为缺省页面大小以外的值。例如，-Xlistl45 将页长度设置为 45 行。缺省值是 66。 如果令 *n*=0 (-Xlistl0)，该选项将显示不带分页符的列表和交叉引用，以便于屏幕查看。

表 **5-2** -Xlist 子选项的完整列表 *(续下)*

选项	操作
-Xlist**MP**	检查 OpenMP 指令的一致性 可使用 -XlistMP 报告源代码文件中指定的 OpenMP 指令的不一致性。有关详细信息，另请参见 *《OpenMP API 用户指南》*。
-Xlist**o** *name*	指定 -Xlist 输出报告文件 可使用 -Xlisto 指定生成的报告输出文件。（在 o 和 *name* 之间必须有一个空格。）使用 -Xlisto name，将会输出到 name 而不是 *file*.lst。 要直接显示到屏幕上，请使用以下选项： -Xlisto /dev/tty
-Xlist**s**	禁止交叉引用中未引用的符号 可使用 -Xlists 在交叉引用表中禁止 include 文件中已定义但源文件中未引用的任何标识符。 如果使用了子选项 -XlistI，该子选项将不起作用。 缺省时*不*显示 #include 或 INCLUDE 文件中出现的标识符。
-Xlist**v***n*	设置检查 “严格” 级别 *n* 为 1、 2、 3 或 4。缺省值为 2 (-Xlistv2)： • -Xlistv1 仅以摘要形式显示所有名称的交叉检查信息，不带行号。这是检查严格性的最低级别 — 仅查语法错误。 • -Xlistv2 以摘要和行号显示交叉检查信息。这是检查严格性的缺省级别，包括变量不一致性错误和变量使用错误。 • -Xlistv3 以摘要、行号和公共块映射显示交叉检查。这是检查严格性的较高级别，包括由不同子程序公共块中数据类型的不正确使用所造成的错误。 • -Xlistv4 以摘要、行号、公共块映射和等价块映射显示交叉检查。这是最为严格的检查级别，可以检测出最多的错误。
-Xlist**w**[*nnn*]	设置输出行的宽度 可使用 -Xlistw 设置输出行的宽度。例如，-Xlist132 将页宽度设置为 132 列。缺省值是 79。
-Xlist**war**[*nnn]*	在报告中禁止警告 *nnn* 可使用 -Xlistwar 禁止输出报告中的特定警告消息。如果未指定 *nnn*，则禁止打印所有警告消息。例如，-Xlistwar338 禁止警告消息号 338。要禁止一条以上的警告但并非所有警告，可重复使用该选项。
-Xlist**X**	只显示交叉引用表和错误 -XlistX 产生交叉引用表和跨例程错误列表，但不产生任何源码列表。

5.2 特殊编译器选项

一些编译器选项对于调试很有用。它们可以用来检查下标、发现未声明的变量、显示编译链接过程中的各个阶段、显示软件的版本，等等。

Solaris 链接程序还具有其它调试辅助选项。参见 ld(1)，或在 shell 提示符下运行命令 ld -Dhelp 来参看联机文档。

5.2.1 下标边界 (-C)

如果用 -C 编译，编辑器在运行时会增加对每个数组下标上的越界引用以及数组一致性的检查。本操作有助于捕获某些会引起段故障的情况。

示例：超出范围的索引：

```
demo% cat range.f
      REAL a(10,10)
      k = 11
      a(k,2) = 1.0
      END
demo% f95 -o range range.f
demo% range

 ******  FORTRAN 运行时系统  ******
下标越界。位置: 'range.f' 的第 3 行第 9 列
在数组 'A' 中，下标编号 1 的值是 11。
终止
demo%
```

5.2.2 未声明的变量类型 (-u)

-u 选项检查任何未定义的变量。

-u 选项会使所有变量被初始标识为未声明，这样，所有未用类型语句或 IMPLICIT 语句显式声明的变量都会被加上错误标志。 -u 标志对于发现类型不匹配的变量很有用。如果设置了 -u，在显式声明之前会将所有变量视为未声明。一旦使用了未声明变量，总会出现错误消息。

5.2.3 编译器版本检查 (-V)

-V 选项可将编译器每一阶段的名称和版本 ID 显示出来。该选项可用于跟踪不明错误消息的起源、报告编译器故障以及验证所安装编译器补丁程序的级别。

```
demo% f95 -V wh.f
f95: Sun Fortran 95 7.0 DEV 2002/01/30
f90comp: Sun Fortran 95 7.0 DEV 2002/01/30
f90comp: 9 个源代码行
f90comp: 0 个错误，0 个警告，0 个其它消息，0 ANSI
ld: Solaris 链接编辑器: 5.8-1.272
```

5.3 使用 dbx 调试

Sun Studio 为调试用 Fortran、C 和 C++ 编写的应用程序，提供了一个紧密集成的开发环境。

dbx 程序提供事件管理、进程控制以及数据检查。您可以监视程序执行期间发生的事件，并且可以执行下列任务：

- 修复一个例程，然后不用重新编译其它例程而继续执行
- 设置在指定项变化时要停止或跟踪的监视点
- 收集性能调节数据
- 监视变量、结构和数组
- 在行或函数中设置断点 （设置程序中的停止位置）
- 显示值 — 一旦停止，便可显示或修改变量、数组、结构
- 单步执行程序，每次执行一行源码或汇编码
- 跟踪程序流程 — 显示已发生的调用序列
- 调用正在调试的程序中的过程
- 步过或步入函数调用；向下单步执行并步出函数调用
- 在下一行或某一其它行运行、停止和继续执行
- 保存然后重新运行调试执行过程的全部或一部分
- 检查调用栈，或上下移动调用站
- 在嵌入的 Korn shell 中编写脚本
- 在程序执行 fork(2) 和 exec(2) 时跟随它们

要调试经过优化的程序，请使用 dbx fix 命令重新编译想要调试的例程：

1. **用适当的 –O*n* 优化级别编译程序。**
2. **在 dbx 下开始执行。**
3. **使用 fix -g any.f，不对要调试的例程进行优化。**
4. **对已编译的该例程使用 continue。**

如果编译命令中存在有 -g，某些优化将被禁止。有关详细信息，参见 dbx 文档。

有关详细信息，参见 Sun Studio 手册 *《使用 dbx 调试程序》*以及 dbx(1) 手册页。

第 6 章

浮点运算

本章考虑浮点运算并提出了避免和检测数值计算错误的策略。

有关 SPARC 处理器浮点计算的详细说明，参见 *《数值计算指南》*。

6.1 简介

SPARC 处理器上的 Fortran 95 浮点环境实现了 “二进制浮点运算 IEEE 标准 754” 指定的运算模型。该环境使您能够开发强大、高性能、可移植的数值应用程序。它还提供了用于分析研究数值程序任何不正常行为的工具。

在数值程序中，有许多潜在因素可引起计算错误：

- 计算模型错误
- 所使用的算法在数值上不稳定
- 病态数据
- 硬件产生意外结果

找出数值计算中出错的来源非常困难。可以尽可能使用市面上销售的经过测试的库程序包来减少编码错误几率。算法的选择是另一个关键问题。使用合适的计算机运算同样也是如此。

本章不打算教授或解释数值错误分析。此处提供的资料旨在介绍 Fortran 95 实现的 IEEE 浮点模型。

6.2 IEEE 浮点运算

IEEE 运算是一种相对较新的算术运算处理方法，它可以处理算术运算中产生的无效操作数、被零除、上溢、下溢或不精确结果等问题。不同之处在于舍入、近零数字的处理以及接近机器最大值的数字的处理。

IEEE 标准支持异常、舍入和精度用户处理。因此，此标准支持区间运算和异常诊断。IEEE 标准 754 使得标准化类似 *exp* 和 *cos* 的基本函数、创建高精度运算以及耦合数值和符号代数计算成为可能。

与其它任何种类的浮点运算相比， IEEE 运算向用户提供了对于计算的更大控制。此标准简化了编写复杂可移植数值程序的任务。浮点运算的许多有关问题都涉及数字的基本运算。例如：

- 当计算机硬件无法表示无限精确的结果时，运算结果会怎样？
- 类似乘法和加法等基本运算可以互换吗？

另一类问题与浮点异常及异常处理有关。如果进行下列运算会发生什么情况：

- 二个非常大的同号数字相乘？
- 非零值除以零？
- 零除以零？

在较早的运算模型中，第一类问题可能不会得到预期的答案，而第二类问题中的异常情况可能都会得到相同的结果：程序在该点中止或以无用结果继续执行。

此标准可确保运算产生具有预期性质的、符合数学预期的结果。它还可确保在异常情况下产生指定的结果，除非用户明确做了其它选择。

例如，直观地引入了异常值 +Inf、 -Inf 和 NaN：

```
big*big = +Inf        正无穷
big*(-big) = -Inf     负无穷
num/0.0 = +Inf        其中 num > 0.0
num/0.0 = -Inf        其中 num < 0.0
0.0/0.0 = NaN         非数字
```

此外，还明确了五种浮点异常类型：

- *无效*。运算操作数在数学上无效 — 例如， 0.0/0.0、 sqrt(-1.0) 和 log(-37.8)
- *被零除*。除数为零，被除数为有限的非零数字 — 例如， 9.9/0.0
- *上溢*。运算产生的结果超出指数范围 — 例如， MAXDOUBLE+0.0000000000001e308
- *下溢*。运算产生的结果太小，无法用正常数字表示 — 例如， MINDOUBLE * MINDOUBLE

- *不精确*。运算产生的结果无法用无限精度表示 — 例如，2.0 / 3.0、log(1.1) 和输入的 0.1

在 *《数值计算指南》*中介绍了 IEEE 标准的实现。

6.2.1 -trap= 编译器选项

利用 -trap=*mode* 选项可以捕获浮点异常。如果 ieee_handler() 调用未建立信号处理程序，异常会用内存转储核心文件终止程序。有关该编译器选项的详细信息，参见 *《Fortran 用户指南》*。例如，为了能够捕获上溢、被零除和无效运算，可使用 -ftrap=common 进行编译。（这是 f95 缺省设置。）

注 – 必须使用 -trap= 编译应用程序的主程序才能进行捕获。

6.2.2 浮点异常

f95 程序不会自动报告异常。要显示程序终止时产生的浮点异常列表，需要显式调用 ieee_retrospective(3M)。一般情况下，如果发生了无效、被零除或上溢异常中的任何一种，都会产生消息。不精确异常不会产生消息，因为它们在实际程序中发生得过于频繁。

6.2.2.1 回顾性摘要

ieee_retrospective 函数通过查询浮点状态寄存器来查明已产生了哪些异常，并且会向标准错误输出打印消息，通知您都有哪些异常曾引起但未清除。此消息通常如下所示；格式可能会随各编译器版本而变化：

```
注意：已引起的 IEEE 浮点异常标志：
     被零除；
已启用的 IEEE 浮点异常陷阱：
     不精确；下溢；上溢；无效运算；
参见《数值计算指南》、 ieee_flags(3M)、
     ieee_handler(3M)
```

Fortran 95 程序需要显式调用 ieee_retrospective，并使用 -xlang=f77 进行编译，以便与 f77 兼容库进行链接。

用 -f77 兼容标志进行编译，将启用程序终止时自动调用 ieee_retrospective 的 Fortran 77 惯例。

可以在调用 ieee_retrospective 前清除异常状态标志，用 ieee_flags() 关闭这些消息中的任何一个或全部。

6.2.3 处理异常

在 SPARC 和 x86 处理器中，缺省将依据 IEEE 标准进行异常处理。但是，在检测浮点异常和生成浮点异常信号 (SIGFPE) 之间是有区别的。

按照 IEEE 标准，当浮点运算期间出现未捕获的异常时，会发生二件事情：

- 系统返回缺省结果。例如，对于 0/0 （*无效*），系统返回结果为 NaN。
- 会设置标志来指示引起了异常。例如，对于 0/0 （*无效*），系统会设置 “无效运算” 标志。

6.2.4 捕获浮点异常

f95 在处理浮点异常方面与早期的 f77 编译器有着明显的区别。

f95 缺省会自动捕获被零除、上溢和无效运算。而 f77 缺省*不*为浮点异常自动产生信号来中断正在运行的程序。其假设是：只要返回期望的值，大多数异常都无关紧要，对其进行捕获会降低性能。

可以使用 f95 命令行选项 -ftrap 来更改缺省设置。 f95 的缺省设置为 -ftrap=common。要使用早期 f77 缺省设置，请用 -ftrap=%none 编译主程序。

6.2.5 非标准运算

可以手动禁用标准 IEEE 运算中一个称为*渐进下溢*的特征。禁用后，程序将被视为在非标准运算下运行。

IEEE 运算标准规定了一种通过动态调整有效数的小数点来渐进处理下溢结果的方法。按 IEEE 浮点格式，小数点出现在有效数之前，并且有一个隐式前导位 1。当浮点计算结果会下溢时，渐进下溢允许将此隐式前导位清为 0，并将小数点移入有效数之中，否则，浮点计算结果便会产生下溢。对于 SPARC 处理器，该结果不是在硬件而是在软件中完成的。如果程序产生的下溢很多 （也许这表示您的算法有问题），可能会导致性能损失。

可以通过以下方法来禁用渐进下溢：使用 -fns 选项进行编译，或从程序内部调用库例程 nonstandard_arithmetic() 将其关闭。调用 standard_arithmetic() 可重新开启渐进下溢。

注 – 为提高效率，必须用 `-fns` 编译应用程序的主程序。参见 *《Fortran 用户指南》*。

对于传统应用程序，请注意：

- `standard_arithmetic()` 子例程替代了早期名为 `gradual_underflow()` 的例程。
- `nonstandard_arithmetic()` 子例程替代了早期名为 `abrupt_underflow()` 的例程。

注 – `-fns` 选项和 `nonstandard_arithmetic()` 库例程仅在某些 SPARC 系统中有效。

6.3 IEEE 例程

以下接口有助于人们使用 IEEE 运算，这些接口在手册页中有介绍。这些接口多数都在数学库 `libsunmath` 和几个 `.h` 文件中。

- `ieee_flags`(3m) — 控制舍入方向和精度；查询异常状态；清除异常状态
- `ieee_handler`(3m) — 建立异常处理程序例程
- `ieee_functions`(3m) — 列出每个 IEEE 函数的名称和用途
- `ieee_values`(3m) — 列出返回特殊值的函数
- 本部分介绍的其它 `libm` 函数：
  - `ieee_retrospective`
  - `nonstandard_arithmetic`
  - `standard_arithmetic`

SPARC 处理器对不同方面的软硬件支持组合符合 IEEE 标准。

最新的 SPARC 处理器包含具有整数乘法和除法指令以及硬件平方根的浮点单元。

当编译代码与运行时浮点硬件正确匹配时，会获得最佳性能。编译器的 `-xtarget=` 选项允许指明运行时硬件。例如， `-xtarget=ultra` 会通知编译器生成在 UltraSPARC 处理器上执行效果最佳的目标代码。

公用程序 `fpversion` 显示安装了哪种浮点硬件，并指示要指定的合适的 `-xtarget` 值。该公用程序可在所有 Sun SPARC 体系结构中运行。有关详细信息，参见 `fpversion`(1)、*《Fortran 用户指南》*和 *《数值计算指南》*。

6.3.1 标志和 `ieee_flags()`

`ieee_flags` 函数用于查询和清除异常状态标志。它是 Sun 编译器随带的 `libsunmath` 库的一部分，可执行下列任务：

- 控制舍入方向和舍入精度
- 检查异常标志状态
- 清除异常状态标志

`ieee_flags` 调用的一般形式为：

```
flags = ieee_flags( action, mode, in, out )
```

四个参数中的每一个都是字符串。输入为 *action*、*mode* 和 *in*。输出为 *out* 和 *flags*。`ieee_flags` 是整数值函数。*flags* 中返回有用的信息，作为 1 位标志集合。有关完整信息，参见 `ieee_flags`(3m) 手册页。

下表中展示了可能的参数值

表 6-1 ieee_flags(*action, mode, in, out*) 的参数值

参数	允许值
action	get、 set、 clear、 clearall
mode	direction、 exception
in， out	nearest、tozero、negative、positive、extended、double single、 inexact、 division、 underflow、 overflow、 invalid all、 common

注意，这些是文字字符串，且输出参数 *out* 必须至少是 CHARACTER*9。 *in* 和 *out* 的可能值的含意取决于与其一起使用的 action 和 mode。下表对此进行了概括：

表 6-2 ieee_flags *in*、 *out* 参数的含意

in 和 *out* 的值	所指
nearest， tozero， negative， positive	舍入方向
extended， double， single	舍入精度
inexact， division， underflow， overflow， invalid	异常
all	全部五种异常
common	常见异常： 无效、除法、上溢

例如，要确定引起了标志的具有最高优先级的异常，请将输入参数 *in* 作为空字符串传递：

```
      CHARACTER *9, out
      ieeer = ieee_flags( 'get', 'exception', '', out )
      PRINT *, out, ' flag raised'
```

另外，要确定是否引起了 overflow 异常标志，请将参数 *in* 设置为 overflow。返回时，如果 out 等于 overflow，会引起 overflow 异常标志；否则不会引起该标志。

```
      ieeer = ieee_flags( 'get', 'exception', 'overflow', out )
      IF ( out.eq. 'overflow') PRINT *,'overflow flag raised'
```

示例：清除 invalid 异常：

```
      ieeer = ieee_flags( 'clear', 'exception', 'invalid', out )
```

示例：清除所有异常：

```
      ieeer = ieee_flags( 'clear', 'exception', 'all', out )
```

示例：将舍入方向设置为零：

```
      ieeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

示例：将舍入精度设置为 double：

```
      ieeer = ieee_flags( 'set', 'precision', 'double', out )
```

6.3.1.1 用 ieee_flags 关闭所有警告消息

如下例所示，以 *action* 为 clear 调用 ieee_flags，会重置任何未清除的异常。在程序退出之前进行该调用，可禁止系统在程序终止时产生浮点异常警告消息。

示例：用 ieee_flags() 清除所有产生的异常：

```
      i = ieee_flags('clear', 'exception', 'all', out )
```

6.3.1.2 用 ieee_flags 检测异常

以下示例演示如何确定早期计算引起的浮点异常。会将系统包含文件 floatingpoint.h 中定义的位屏蔽应用于 ieee_flags 的返回值。

在以下示例中，即 DetExcFlg.F，包含文件是使用 #include 预处理程序指令引入的，这就要求以 .F 后缀命名源文件。下溢是由最小的双精度数除以 2 引起的。

示例：使用 ieee_flags 检测异常并解码：

```
#include "floatingpoint.h"
         CHARACTER*16 out
         DOUBLE PRECISION d_max_subnormal, x
         INTEGER div, flgs, inv, inx, over, under

       x = d_max_subnormal() / 2.0                ! 造成下溢

       flgs=ieee_flags('get','exception','',out)  ! 引起哪些标志?

       inx   = and(rshift(flgs, fp_inexact)  , 1)  ! 解码
       div   = and(rshift(flgs, fp_division) , 1)   ! ieee_flags
       under = and(rshift(flgs, fp_underflow), 1)    ! 返回
       over  = and(rshift(flgs, fp_overflow) , 1)     ! 的
       inv   = and(rshift(flgs, fp_invalid)  , 1)      ! 值

       PRINT *, "Highest priority exception is: ", out
       PRINT *, ' invalid  divide  overflo underflo inexact'
       PRINT '(5i8)', inv, div, over, under, inx
       PRINT *, '(1 = exception is raised; 0 = it is not)'
     i = ieee_flags('clear', 'exception', 'all', out)   ! 全部清除
       END
```

示例：编译并运行上述示例 (DetExcFlg.F)：

```
demo% f95 DetExcFlg.F
demo% a.out
 Highest priority exception is: underflow
  invalid  divide  overflo underflo inexact
        0       0       0       1       1
 (1 = exception is raised; 0 = it is not)
demo%
```

6.3.2 IEEE 极值函数

编译器提供了一个函数集，可以调用其中的函数来返回特殊的 IEEE 极值。这些值，如 *infinity* 或 *minimum normal*，可以直接在应用程序中使用。

示例：基于硬件支持的最小数值的收敛测试如下所示：

```
      IF ( delta .LE. r_min_normal() ) RETURN
```

下表列出了可用的值：

表 6-3 返回 IEEE 值的函数

IEEE 值	双精度	单精度
infinity	d_infinity()	r_infinity()
quiet NaN	d_quiet_nan()	r_quiet_nan()
signaling NaN	d_signaling_nan()	r_signaling_nan()
min normal	d_min_normal()	r_min_normal()
min subnormal	d_min_subnormal()	r_min_subnormal()
max subnormal	d_max_subnormal()	r_max_subnormal()
max normal	d_max_normal()	r_max_normal()

二个 NaN 值 （quiet 和 signaling）是*无序的*，不能用于比较，如 IF(X.ne.r_quiet_nan())THEN... 要确定某些值是否是 NaN，请使用函数 ir_isnan(r) 或 id_isnan(d)。

以下手册页列出了这些函数的 Fortran 名称：

- libm_double(3f)
- libm_single(3f)
- ieee_functions(3m)

另请参见：

- ieee_values(3m)
- floatingpoint.h 头文件和 floatingpoint(3f)

6.3.3 异常处理程序和 ieee_handler()

关于 IEEE 异常，通常需要关注以下问题：

- 异常出现时会发生什么情况？
- 如何使用 ieee_handler() 将用户函数建立作为异常处理程序？
- 如何编写可用作异常处理程序的函数？
- 如何定位异常 — 它出现在何处？

用户例程的异常捕获以系统产生浮点异常信号开始。 *signal: floating-point exception* 的 UNIX 标准名称为 SIGFPE。出现异常时， SPARC 平台上的缺省情况是*不*产生 SIGFPE。要使系统产生 SIGFPE，必须先启用异常捕获，这通常通过对 ieee_handler() 的调用来完成。

6.3.3.1 建立异常处理程序函数

要将函数建立作为异常处理程序，请将函数名称与要监视的异常的名称和采取的操作一起传递给 ieee_handler()。一旦建立了处理程序，无论何时出现特定的浮点异常和调用指定的函数，都会产生 SIGFPE 信号。

ieee_handler() 的调用形式如下表所示：

表 6-4 ieee_handler(*action*, *exception*, *handler*) 的参数

参数	类型	可能值
action	**字符**	get、 set 或 clear
exception	**字符**	invalid、division、overflow、underflow 或 inexact
处理程序	函数名	用户处理函数的名称或 SIGFPE_DEFAULT、SIGFPE_IGNORE、 SIGFPE_ABORT
返回值	**整型**	0 = *正常*

用 f95 编译的、调用 ieee_handler() 的 Fortran 95 例程还应该声明：

```
#include 'floatingpoint.h'
```

特殊参数 SIGFPE_DEFAULT、 SIGFPE_IGNORE 和 SIGFPE_ABORT 定义在这些包含文件中，可用于更改与特定异常相应的程序行为：

SIGFPE_DEFAULT *或* SIGFPE_IGNORE	出现指定异常时不采取任何操作。
SIGFPE_ABORT	程序在异常时中止 （可能会使用转储文件）。

6.3.3.2 编写用户异常处理程序函数

异常处理程序采取的操作由您决定。但是，此例程必须是整型函数，且具有下面指定的三个参数：

handler_name(sig, sip, uap)

- *handler_name* 是此整型函数的名称。
- sig 是整数。
- sip 是具有结构 siginfo 的记录。
- uap 未使用。

示例：异常处理程序函数：

```
      INTEGER FUNCTION hand( sig, sip, uap )
      INTEGER sig, location
      STRUCTURE /fault/
           INTEGER address
           INTEGER trapno
      END STRUCTURE
      STRUCTURE /siginfo/
           INTEGER si_signo
           INTEGER si_code
           INTEGER si_errno
           RECORD /fault/ fault
      END STRUCTURE
      RECORD /siginfo/ sip
      location = sip.fault.address
      ... 采取的操作 ...
      END
```

要在 SPARC V9 体系结构 （`-xarch=v9` 或 `v9a`）上运行该示例，必须对其进行修改，方法是用 `INTEGER*8` 替换每个 `STRUCTURE` 中的所有 `INTEGER` 声明。

如果由 `ieee_handler()` 启用的处理程序例程与例中一样，是用 Fortran 编写的，则此例程不能对其第一个参数 (`sig`) 进行任何引用。该第一个参数*按值*传递给此例程，并且只能作为 `loc(sig)` 进行引用。此值是信号编号。

通过处理程序检测异常

下例示例展示如何创建处理程序例程来检测浮点异常。

示例：检测异常并中止：

```
demo% cat DetExcHan.f
      EXTERNAL myhandler
      REAL :: r = 14.2 , s = 0.0
      i = ieee_handler ('set', 'division', myhandler )
      t = r/s
      END

      INTEGER FUNCTION myhandler(sig,code,context)
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo% f95 DetExcHan.f
demo% a.out
Abort
demo%
```

无论该浮点异常何时出现，均会产生 SIGFPE。检测到 SIGFPE 时，控制将传递给 myhandler 函数，该函数会立即中止。用 -g 编译，并使用 dbx 查找异常位置。

通过处理程序定位异常

示例：定位异常 （打印地址）并中止：

```
demo% cat LocExcHan.F
#include "floatingpoint.h"
      EXTERNAL Exhandler
      INTEGER Exhandler, i, ieee_handler
      REAL:: r = 14.2 ,  s = 0.0 , t
C  Detect division by zero
      i = ieee_handler( 'set', 'division', Exhandler )
      t = r/s
      END

      INTEGER FUNCTION Exhandler( sig, sip, uap)
      INTEGER sig
      STRUCTURE /fault/
                  INTEGER address
      END STRUCTURE
      STRUCTURE /siginfo/
                  INTEGER si_signo
                  INTEGER si_code
                  INTEGER si_errno
                  RECORD /fault/ fault
      END STRUCTURE
      RECORD /siginfo/ sip
      WRITE (*,10)  sip.si_signo, sip.si_code, sip.fault.address
10       FORMAT('Signal ',i4,' code ',i4,' at hex address ', Z8 )
      Exhandler=1
      CALL abort()
      END
demo%  f95 -g LocExcHan.F
demo%  a.out
Signal   8 code    3 at hex address    11230
Abort
demo%
```

在 SPARC V9 环境中，请用 `INTEGER*8` 替换每个 `STRUCTURE` 中的 `INTEGER` 声明，用 `i8` 替换 `i4` 格式。（注意，该例接受 VAX Fortran `STRUCTURE` 语句，依靠的是 `f95` 编译器的扩展。）

大多数情况下，知道异常的实际*地址*并无太大用处，但对于 dbx 除外：

```
demo% dbx a.out
(dbx) stopi at 0x11230    在地址处设置断点
(2) stopi at &MAIN+0x68
(dbx) run                 运行程序
运行：a.out
（进程 id 18803）
停止于 MAIN 中的 0x11230 处
MAIN+0x68:      fdivs   %f3, %f2, %f2
(dbx) where             显示异常的行号
=>[1] MAIN(), "LocExcHan.F" 中的第 7 行
(dbx) list 7            显示源代码行
    7         t = r/s
(dbx) cont              在断点后继续，进入处理程序例程
Signal    8 code    3 at hex address    11230
abort：已调用
signal ABRT (Abort) 在 _kill 中的 0xef6e18a4 处
_kill+0x8:      bgeu    _kill+0x30
当前函数是 exhandler
   24         CALL abort()
(dbx) quit
demo%
```

当然，还有更容易的方法来确定引起错误的源码行。但是，本例确实足以展示异常处理的基础。

6.4 调试 IEEE 异常

定位异常出现的*位置*需要启用异常捕获。这可以通过使用 -trap=common 选项（用 f95 编译器编译时的缺省选项）进行编译，或通过使用 ieee_handler() 建立异常处理程序例程来完成。启用了异常捕获，便可从 dbx 中运行程序，使用 dbx catch FPE 命令来查看出错位置。

使用 -trap=common 编译的优点是：无需修改源代码即可捕获异常。但是，通过调用 ieee_handler()，您可以更有选择地查看异常。

示例：dbx 的编译及使用：

```
demo% f95 -g myprogram.f
demo% dbx a.out
读取 a.out 的符号信息
...
(dbx) catch FPE
(dbx) run
运行：a.out
（进程 id 19739）
信号 FPE（浮点数除以零）位于文件 "myprogram.f" 第 212 行的 MAIN 中
  212                 Z = X/Y
(dbx)  print Y
y = 0.0
(dbx)
```

如果发现程序因上溢和其它异常而终止，可调用 ieee_handler() 明确定位第一处上溢，以便只捕获上溢。这至少需要修改主程序的源代码，如下例所示。

示例：在其它异常出现时定位上溢：

```
demo% cat myprog.F
#include "floatingpoint.h"
        program myprogram
...
      ier = ieee_handler('set','overflow',SIGFPE_ABORT)
...
demo% f95 -g myprog.F
demo% dbx a.out
读取 a.out 的符号信息
...
(dbx) catch FPE
(dbx) run
运行：a.out
（进程 id 19793）
信号 FPE（浮点数溢出）位于文件 "myprog.F" 第 55 行的 MAIN 中
   55                w = rmax * 200.                          ！造成上溢
(dbx) cont                                                ！继续执行至完成
执行完毕，退出代码为 0
(dbx)
```

为了具有选择性，此例引入了 #include，它需要以 .F 后缀重命名源文件，并调用 ieee_handler()。您可更深入一层，创建出现上溢异常时要调用的自己的处理程序函数，执行一些应用程序特定的分析，并在中止前打印中间或调试结果。

6.5 更深层次的数值风险

本部分介绍一些与算术运算有关的现实问题，这些问题可能会在不经意间产生无效、被零除、上溢、下溢或不精确异常。

例如，在 IEEE 标准之前，如果在计算机中将二个非常小的数相乘，结果可能为零。多数大型机和小型机的行为亦是如此。对于 IEEE 运算，*渐进下溢*扩展了计算的动态范围。

例如，假设某一 32 位处理器以 `1.0E-38` 作为机器中可表示的最小值 *epsilon*。将二个小数相乘：

```
a = 1.0E-30
b = 1.0E-15
x = a * b
```

较早的运算会得到 0.0，但对于 IEEE 运算和相同的字长，却会得到 1.40130E-45。此时便出现了下溢，告诉您答案比机器自然表示的值小。该结果是通过从尾数中“窃取”一些位并将其移交给指数来完成的。得到的结果是一个*非正规化数字*，在某种意义上来说，它不够精确，但从另一方面来说，又足够精确。其深层含意已超出了本次讨论的范围。如果您感兴趣，请特别查阅 *Computer* 1980 年 1 月第 13 卷第 1 号中 J. Coonen 的文章“Underflow and the Denormalized Numbers”。

大多数科学程序都有对舍入很敏感的代码段，通常是在方程求解或矩阵因子分解中。若不采用渐进下溢，编程人员就得自己实现检测接近不准确阈值的方法。否则，他们就必须放弃对实现强大、稳定算法的追求。

有关这些主题的详细信息，参见 *《数值计算指南》*。

6.5.1 避免简单下溢

有些应用程序实际上会执行许多十分接近于零的计算。这在计算残数或微分修正的算法中很常见。为获得在数值上安全的最大性能，需要采用扩展精度运算来执行关键计算。如果应用程序是单精度应用程序，可采用双精度执行关键计算。

示例：采用单精度的简单点积计算：

```
sum = 0
DO i = 1, n
   sum = sum + a(i) * b(i)
END DO
```

如果 a(i) 和 b(i) 非常小，会出现很多下溢。通过强制计算采用双精度，计算点积时会具有更高的准确性，并且可免受下溢之苦：

```
DOUBLE PRECISION sum
DO i = 1, n
   sum = sum + dble(a(i)) * dble(b(i))
END DO
result = sum
```

通过增加对库例程 nonstandard_arithmetic() 的调用，或通过使用 -fns 选项编译应用程序的主程序，可以强制 SPARC 处理器在涉及到下溢时象较早的系统一样进行处理 （存储零）。

6.5.2 以错误答案继续

您可能会对答案明显错误还继续计算而感到不解。 IEEE 运算允许您区分可以忽略的错误答案的种类，如 NaN 或 Inf。然后，可以根据此种区分来作决定。

不妨考虑一个电路模拟的例子。在 50 行的特定计算中 （出于参数原因）唯一关注的变量是电压。进一步假设其值只可能是 +5v、 0、 -5v。

仔细安排计算的每一部分以强制每个子结果在正确范围内，这是很可能实现的：

- *如果计算值大于 4.0，返回 5.0*
- *如果计算值介于 -4.0 和 +4.0 之间，返回 0*
- *如果计算值小于 -4.0，返回 -5.0*

此外，由于 Inf 不是允许值，所以需要特殊的逻辑来确保不会与大数相乘。

利用 IEEE 运算，此逻辑可以简化许多。计算可以用显而易见的方式编写，并且只需强制最终结果为正确的值 — 因为 Inf 可以出现并且可以很容易地测出。

此外，还可检测到 0/0 的情况并按照您的意愿进行处理。结果更易读取且执行起来更快，因为无需进行不必要的比较。

6.5.3 过度下溢

如果将二个很小的数相乘，结果会下溢。

如果预知乘法 （或减法）中的操作数可能会很小并且很可能会下溢，可采用双精度进行计算，而后再将结果转换成单精度。

例如，类似下面的点积循环：

```
real sum, a(maxn), b(maxn)
...
do i =1, n
    sum = sum + a(i)*b(i)
enddo
```

其中，已知 a(*) 和 b(*) 具有小元素，为保持数值准确度，应采用双精度来运行：

```
real a(maxn), b(maxn)
double sum
...
do i =1, n
    sum = sum + a(i)*dble(b(i))
enddo
```

鉴于以软件方式解决了原始循环造成的过度下溢，这样做还有可能提高性能。但是，对此并无绝对而快速的法则；只能通过对计算代码进行高强度实验来确定最有利的解决方案。

6.6 区间运算

Fortran 95 编译器 `f95` 支持将*区间*作为内在数据类型。区间是封闭的紧集：$[a, b] = \{z \mid a \le z \le b\}$，由一对数字定义，$a \le b$。区间可用来：

- 解决非线性问题
- 执行严格的错误分析
- 检测数值不稳定的缘由

通过在 Fortran 95 中引入区间作为内在数据类型，开发人员便可以立即使用 Fortran 95 的所有适用语法及语义。除了 `INTERVAL` 数据类型外， `f95` 还包括 Fortran 95 的下列区间扩展：

- 三类 `INTERVAL` 关系操作符：
  - 确定型
  - 可能型
  - 集合型
- 内在 `INTERVAL` 专用操作符，如 `INF`、 `SUP`、 `WID` 和 `HULL`
- `INTERVAL` 输入 / 输出编辑描述符，包括单数字输入 / 输出
- 算术、三角及其它数学函数的区间扩展
- 表达式依赖于上下文的 `INTERVAL` 常量
- 混合模式区间表达式处理

`f95` 命令行选项 `-xinterval` 可启用编译器的区间运算功能。参见 *《Fortran 用户指南》*。

有关 Fortran 95 中区间运算的详细信息，参见 *《Fortran 95 区间运算编程参考》*。

第 7 章

移植

本章讨论从其它平台向 Fortran 95 移植 “旧式” Fortran 程序时可能产生的一些问题。

Fortran 95 扩展和 Fortran 77 兼容功能在 *《Fortran 用户指南》* 中介绍。

7.1 回车控制

Fortran 回车控制是由于最初开发 Fortran 时使用的设备能力有限所造成的。出于类似的历史原因，源自 UNIX 的操作系统不具备 Fortran 回车控制，但您可以用 Fortran 95 编译器以两种方式对其进行模拟。

- 在用 `lpr` 命令打印文件之前，使用 `asa` 过滤器将 Fortran 回车控制惯例转换成 UNIX 回车控制格式 （参见 `asa` (1) 手册页）。
- FORTRAN 77 编译器 `f77` 允许 `OPEN(N, FORM='PRINT')` 启用单间隔或双间隔、换页以及剥离第一列。这还可通过与 `f95 -f77` 兼容标志一起使用 `FORM='PRINT'` 编译程序来获得。此编译器允许在使用 `-f77` 编译时，重新打开单元 6，将形式参数更改为 `PRINT`。例如：

```
      OPEN( 6, FORM='PRINT')
```

可以使用 `lp`(1) 打印以此种方式打开的文件。

7.2 使用文件

早期 Fortran 系统不使用命名文件，但的确提供了一种命令行机制，用于将实际文件名与内部单元号等同起来。可以用多种方式模拟该功能，其中包括标准 UNIX 重定向。

示例：将 stdin 重定向至 redir.data（使用 csh(1)）：

```
demo% cat redir.data              数据文件
 9 9.9

demo% cat redir.f                 源文件
      read(*,*) i, z              程序读取标准输入
      print *, i, z
      stop
      end

demo% f95 -o redir redir.f        编译步骤
demo% redir < redir.data                  以重定向方式运行读取数据文件
  9 9.90000
demo%
```

7.3 从科学大型机移植

如果应用程序代码最初是为 64 位（或 60 位）大型机（如 CRAY 或 CDC）开发的，则在向 UltraSPARC-II 平台移植时，不妨使用下列选项编译这些代码，例如：

```
-fast  -xarch=v9a  -xchip=ultra2   \
-xtypemap=real:64,double:64,integer:64
```

这些选项自动将所有缺省 REAL 变量和常量提升至 REAL*8，将 COMPLEX 提升至 COMPLEX*16。只有未声明的变量或声明为简单 REAL 或 COMPLEX 的变量才会得到提升；显式声明的变量（例如， REAL*4）不会被提升。所有单精度 REAL 常量也会被提升至 REAL*8。（针对目标平台相应地设置 -xarch 和 -xchip。）若要将缺省 DOUBLE PRECISION 数据也提升至 REAL*16，请将 -xtypemap 示例中的 double:64 更改为 double:128。

有关详细信息，参见 *《Fortran 用户指南》* 或 f95(1) 手册页。

7.4 数据表示

*《Fortran 用户指南》*和 *《数值计算指南》*详细讨论了 Fortran 中数据对象的硬件表示。跨系统和硬件平台的数据表示之间的差别通常会产生最严重的可移植问题。

应注意下列问题：

- Sun 遵守浮点运算 “IEEE 标准 754”。因此， `REAL*8` 中的头四个字节与 `REAL*4` 中的头四个字节不同。
- 实数、整数和逻辑值的缺省大小在 Fortran 95 标准中进行了说明，不过这些缺省大小可以通过 `-xtypemap` 选项进行更改。
- 可以自由混合字符变量并使其等价于其它类型的变量，但需注意潜在的对齐问题。
- `f95` IEEE 浮点运算会在上溢或被零除以及信号 `SIGFPE` 或缺省捕获 （对于 `f95`，缺省为 `-ftrap=common`）时引起异常。在某些情况下，它只能传送 IEEE 不定式，否则，就将以信号方式通知异常。第 6 章对此进行了解释。
- 可以确定有限、正规化的极值。参见 `libm_single`(3F) 和 `libm_double`(3F)。不定式可以使用格式化、列表控制的 I/O 语句进行读写。

7.5 霍尔瑞斯数据

很多 “旧式” Fortran 应用程序会将霍尔瑞斯 ASCII 数据存储在数值数据对象中。在 1977 Fortran 标准 （以及 Fortran 95）中，为此目的提供了 CHARACTER 数据类型，并建议使用。您仍可利用早先的 Fortran 霍尔瑞斯 (*n*H) 功能对变量进行初始化，但这不是标准措施。下表指出了适合于某些数据类型的最大字符数。（在本表中，粗体数据类型指示应通过 -xtypemap 命令行标志提升缺省类型。）

表 7-1 数据类型的最大字符数

	最大标准 **ASCII** 字符数			
数据类型	**缺省值**	**INTEGER:64**	**REAL:64**	**DOUBLE:128**
BYTE	1	1	1	1
COMPLEX	8	8	16	16
COMPLEX*16	16	16	16	16
COMPLEX*32	32	32	32	32
DOUBLE COMPLEX	16	16	32	32
DOUBLE PRECISION	8	8	16	16
INTEGER	4	8	4	8
INTEGER*2	2	2	2	2
INTEGER*4	4	4	4	4
INTEGER*8	8	8	8	8
LOGICAL	4	8	4	8
LOGICAL*1	1	1	1	1
LOGICAL*2	2	2	2	2
LOGICAL*4	4	4	4	4
LOGICAL*8	8	8	8	8
REAL	4	4	8	8
REAL*4	4	4	4	4
REAL*8	8	8	8	8
REAL*16	16	16	16	16

示例：用霍尔瑞斯初始化变量：

```
demo% cat FourA8.f
      double complex x(2)
      data x /16Habcdefghijklmnop, 16Hqrstuvwxyz012345/
      write( 6, '(4A8, "!")' ) x
      end

demo% f95 -o FourA8 FourA8.f
demo% FourA8
abcdefghijklmnopqrstuvwxyz012345!
demo%
```

如果需要，可以用霍尔瑞斯初始化具有兼容类型的数据项，然后将其传递给其它例程。

如果将霍尔瑞斯常量作为参数传递，或者将其用在表达式或比较中，它们将被解释为字符型表达式。使用编译器选项 -xhasc=no，可以让编译器在子程序调用时的参数中将霍尔瑞斯常量视作无类型数据。在移植较早的 Fortran 程序时可能需要这样做。

7.6 非标准编码措施

作为一条通用规则，在从一个系统和编译器向另一个系统和编译器移植应用程序时，可以通过消除非标准编码来使之变得更加容易。在一个系统中成功的优化或迂回手段，在其它系统中可能只会给编译器造成晦涩和混乱。特别是，针对某一特定体系结构所做的优化手动调节，在别处可能会造成性能下降。对此将在后面关于性能和调节的章节中予以讨论。但是，就一般性移植而言，下列问题值得考虑。

7.6.1 未初始化的变量

有些系统会自动将局部和 COMMON 变量初始化为零或某个“非数字”(NaN) 值。但是，没有任何标准措施，而且程序不应对任何变量的初值进行假设。要确保最大程度的可移植性，程序应初始化所有变量。

7.6.2 别名使用和 -xalias 选项

当多个名称引用同一存储地址时会出现别名使用情况。这种情况通常发生在指针上，或者是在子程序的实参相互重叠或与子程序内的 COMMON 变量相互重叠时发生。例如，参数 X 和 Z 引用同一存储位置， B 和 H 亦然：

```
COMMON /INS/B(100)
REAL S(100), T(100)
...
CALL SUB(S,T,S,B,100)
...
SUBROUTINE SUB(X,Y,Z,H,N)
REAL X(N),Y(N),Z(N),H(N)
COMMON /INS/B(100)
...
```

作为一种手段，很多“旧式” Fortran 程序利用此种别名使用机制来提供当时程序语言中尚不具备的某种动态内存管理。

在所有可移植代码中避免别名使用。在某些平台上以及在用高于 -O2 的优化级别编译时，其结果可能是无法预料的。

f95 编译器会假定其编译的是符合标准的程序。不严格符合 Fortran 标准的程序可能会引起二义性情况，从而干扰编译器的分析和优化策略。某些情况甚至能产生错误结果。

例如，数组索引越界、使用指针或将还被直接使用的全局变量作为子程序参数传递，都可导致二义性情况，从而限制了编译器生成在所有情况下都正确的优化代码的能力。

如果知道程序的确包含一些明显的别名使用情况，可使用 -xalias 选项指定编译器的关注程度。在某些情况下，当以高于 -O2 的优化级别编译时，程序不会正确执行，除非指定了适当的 -xalias 选项。

此选项标志会获取一个以逗号分隔的、指示别名使用情况类型的关键字列表。可在每个关键字前冠以 no% 前缀，用以指示不存在的别名使用。

表 7-2 -xalias 关键字及其含意

-xalias= *keyword*	别名使用情况
dummy	伪子程序参数既可以互为别名，也可以作为全局变量的别名。
no%dummy	遵守 Fortran 标准，伪参数在实际调用中既不互为别名也不作为全局变量的别名。（这是缺省设置。）
craypointer	程序使用可指向任何地址的 Cray 指针。（这是缺省设置。）
no%craypointer	Cray 指针总是指向明确的内存区，或者不被使用。
ftnpointer	任何 Fortran 95 指针都可以指向任一目标变量，而不管其为何类型、种类或等级。
no%ftnpointer	Fortran 95 指针遵守标准规则。（这是缺省设置。）
overindex	在数组引用中违反下标边界可造成四种索引越界情况，其中的任何一种或多种都可以在程序中出现： t 对 COMMON 块中数组元素的引用可以引用 COMMON 块或等价组中的任何元素。 • 作为子程序的实际参数传递 COMMON 块或等价组的某一元素，将允许对该 COMMON 块或等价组中的任何元素进行访问。 • 会将序列派生类型的变量当作是 COMMON 块一样来看待，并且此种变量的元素可以作为该变量其它元素的别名。 • 可以违反各个数组下标边界，即使数组引用仍在该数组内。 overindex 不适用于数组语法、 WHERE 和 FORALL 语句。如果索引越界出现在这些构造中，应将它们改写为 DO 循环。
no%overindex	不违反数组边界。数组引用不引用其它变量。（这是缺省设置。）
actual	编译器将实际子程序参数当作是全局变量一样来看待。向子程序传递参数有可能通过 Cray 指针导致别名使用。
no%actual	向子程序传递参数不会进一步造成别名使用。（这是缺省设置。）

这里列举了别名使用情况的一些典型示例。在较高优化级别 （-O3 及其以上级别）上，如果您的程序中不包含如下所示的别名使用弊病，并且您是用 -xalias=no%*keyword* 进行编译的， f95 编译器可以生成更好的代码。

在某些情况下，您需要使用 -xalias=*keyword* 进行编译，以确保代码生成将会产生正确的结果。

7.6.2.1 通过伪参数和全局变量别名使用

下例需要使用 -xalias=dummy 进行编译

```
parameter (n=100)
integer a(n)
common /qq/z(n)
call sub(a,a,z,n)
...
subroutine sub(a,b,c,n)
integer a(n), b(n)
common /qq/z(n)
a(2:n) = b(1:n-1)
c(2:n) = z(1:n-1)
```

编译器必须假设伪变量和公共变量可以重叠。

7.6.2.2 随 Cray 指针带来的别名使用

本例仅在使用 -xalias=craypointer （此为缺省设置）编译时才适用：

```
parameter (n=20)
integer a(n)
integer v1(*), v2(*)
pointer (p1,v1)
pointer (p2,v2)
p1 = loc(a)
p2 = loc(a)
a = (/ (i,i=1,n) /)
...
v1(2:n) = v2(1:n-1)
```

编译器必须假设这些位置可以重叠。

下面给出了一个 Cray 指针不重叠的例子。此时，用 -xalias=no%craypointer 进行编译可以获得很可能更佳的性能：

```
parameter (n=10)
integer a(n+n)
integer v1(n), v2(n)
pointer (p1,v1)
pointer (p2,v2)
p1 = loc(a(1))
p2 = loc(a(n+1))
...
v1(:) = v2(:)
```
Cray 指针不指向重叠的内存区。

7.6.2.3 随 Fortran 95 指针带来的别名使用

用 -xalias=ftnpointer 编译以下示例

```
parameter (n=20)
integer, pointer :: a(:)
integer, target :: t(n)
interface
  subroutine sub(a,b,n)
    integer, pointer :: a(:)
    integer, pointer :: b(:)
  end subroutine
end interface

a => t
a = (/ (i, i=1,n) /)
call sub(a,a,n)
....
end
subroutine sub(a,b,n)
 integer, pointer :: a(:)
 real, pointer :: b(:)
 integer i, mold

 forall (i=2:n)
   a(i) = transfer(b(i-1), mold)
```
编译器必须假设 a 和 b 可以重叠。

注意，在本例中，编译器必须假设 a 和 b 可以重叠，即使它们指向不同数据类型的数据。这在标准 Fortran 中是非法的。如果编译器能够检测到此种情况，它会发出警告。

7.6.2.4 由索引越界造成的别名使用

用 -xalias=overindex 编译以下示例

```
integer a,z
common // a(100),z
z = 1
call sub(a)
print*, z
subroutine sub(x)
  integer x(10)
  x(101) = 2
编译器可能会假设对 sub 的调用可以写至 z
用 -xalias=overindex 编译时，此程序会打印 2 而不是 1
```

索引越界在很多传统 Fortran 77 程序中都会出现，应予以避免。在很多情况下，结果将无法预料。要确保正确性，应使用 -C（运行时数组边界检查）选项编译和测试程序，以标记任何可能的数组下标问题。

一般而言，只能与传统 Fortran 77 程序一起使用 overindex 标志。 -xalias=overindex 不适用于数组语法表达式、数组段、 WHERE 和 FORALL 语句。

为确保生成代码的正确性， Fortran 95 程序应该总是符合 Fortran 标准中的下标规则。例如，下例的一个数组语法表达式中使用了二义性下标，该表达式因数组索引越界*必定*将产生不正确的结果：

本数组语法索引越界示例不会给出正确结果！

```
   parameter (n=10)
   integer a(n),b(n)
   common /qq/a,b
   integer c(n)
   integer m, k
   a = (/ (i,i=1,n) /)
   b = a
   c(1) = 1
   c(2:n) = (/ (i,i=1,n-1) /)

   m = n
   k = n + n
C
C   对 a 的引用其实是 b 中的引用
C   所以，这实际应为  b(2:n) = b(1:n-1)
C
   a(m+2:k) = b(1:n-1)

C  或将其反过来进行
   a(k:m+2:-1) = b(n-1:1:-1)
```

用户直觉上可能会期望数组 b 现在将与数组 c 相似，但结果无法预料

xalias=overindex 标志无助于此种情况，因为 overindex 标志没有扩展至数组语法表达式。此例虽然可以编译，但不会给出正确结果。用等价的 DO 循环替换数组语法，改写本例，在用 -xalias=overindex 进行编译后，就正常了。但应完全避免这种编程习惯。

7.6.2.5 由实际参数造成的别名使用

编译器超前查看局部变量是如何使用的，然后假设变量不会随子程序调用而变化。在下例中，子程序中使用的指针使编译器优化策略失败，并且结果无法预料。要使此例正确工作，需用 -xalias=actual 标志进行编译：

```
program foo
      integer i
      call take_loc(i)
      i = 1
      print * , i
      call use_loc()
      print * , i
   end

   subroutine take_loc(i)
      integer i
      common /loc_comm/ loc_i
      loc_i = loc(i)
   end subroutine take_loc

   subroutine use_loc()
      integer vi1
      pointer (pi,vi)
      common /loc_comm/ loc_i
      pi = loc_i
      vi1 = 3
   end subroutine use_loc
```

take_loc 获取 i 的地址并将其保存起来。 use_loc 使用该地址。这违反了 Fortran 标准。

用 -xalias=actual 标志进行编译，将会通知编译器应将传给子程序的所有参数在编译单元内看作是全局性的，从而使编译器在对作为实际参数出现的变量作出假设更加小心。

应避免诸如此类违反 Fortran 标准的编程习惯。

7.6.2.6 -xalias 缺省设置

不带列表指定 -xalias，将假设程序不会违反 Fortran 别名使用规则。它等价于对所有别名使用关键字断言 no%。

不指定 -xalias 进行编译时，编译器缺省设置是：

-xalias=no%dummy,craypointer,no%actual,no%overindex,no%ftnpointer

如果程序使用 Cray 指针但符合 Fortran 别名使用规则 （据此，即使在二义情况下指针引用也不可能导致别名使用），则用 `-xalias` 进行编译，结果可能会生成更好的优化代码。

7.6.3 模糊优化

传统代码可能包含普通计算 DO 循环的源代码重构，后者旨在使较早的向量化编译器为特定体系结构生成优化代码。大多数情况下，这些重构不再需要了，而且可能会降低程序的可移植性。两种常见的重构分别是条状提取和循环展开。

7.6.3.1 条状提取

某些体系结构中的定长向量寄存器致使编程人员手动将循环中的数组计算 “条状提取” 至段中：

```
REAL TX(0:63)
...
DO IOUTER = 1,NX,64
   DO IINNER = 0,63
      TX(IINNER) = AX(IOUTER+IINNER) * BX(IOUTER+IINNER)/2.
      QX(IOUTER+IINNER) = TX(IINNER)**2
   END DO
END DO
```

条状提取对现代编译器已不再适合；可以大大降低模糊程度将循环编写为：

```
DO IX = 1,N
  TX = AX(I)*BX(I)/2.
  QX(I) = TX**2
END DO
```

7.6.3.2 循环展开

在可自动执行此种重构的编译器出现之前，手动展开循环是典型的源代码优化技术。循环被编写为：

```
 DO        K = 1, N-5, 6
    DO     J = 1, N
       DO I = 1,N
          A(I,J) = A(I,J) + B(I,K  ) * C(K  ,J)
*                         + B(I,K+1) * C(K+1,J)
*                         + B(I,K+2) * C(K+2,J)
*                         + B(I,K+3) * C(K+3,J)
*                         + B(I,K+4) * C(K+4,J)
*                         + B(I,K+5) * C(K+5,J)
       END DO
    END DO
 END DO
 DO        KK = K,N
    DO     J =1,N
       DO I =1,N
          A(I,J) = A(I,J) + B(I,KK) * C(KK,J)
       END DO
    END DO
 END DO
```

应按其原始意图进行改写：

```
 DO        K = 1,N
    DO     J = 1,N
       DO I = 1,N
          A(I,J) = A(I,J) + B(I,K) * C(K,J)
       END DO
    END DO
 END DO
```

7.7 时间和日期函数

返回当天时间或 CPU 经过时间的库函数会随着系统的不同而变化。

Fortran 库中支持的时间函数列在下表中：

表 7-3 Fortran 时间函数

名称	函数	手册页
time	返回自 1970 年 1 月 1 日以来经过的秒数	time(3F)
date	以字符串形式返回日期	date(3F)
fdate	以字符串形式返回当前时间和日期	fdate(3F)
idate	在整型数组中返回当前的年、月、日	idate(3F)
itime	在整型数组中返回当前的时、分、秒	itime(3F)
ctime	将 time 函数返回的时间转换成字符串	ctime(3F)
ltime	将 time 函数返回的时间转换成本地时间	ltime(3F)
gmtime	将 time 函数返回的时间转换成格林威治时间	gmtime(3F)
etime	*单处理器：*返回程序执行时经过的用户及系统时间 *多处理器：*返回挂钟时间	etime(3F)
dtime	返回自上次调用 dtime 以来经过的用户及系统时间	dtime(3F)
date_and_time	以字符和数字形式返回日期及时间	date_and_time(3F)

有关详细信息，参见 *《Fortran 库参考手册》*或这些函数各自的手册页。下面给出了一个使用这些时间函数的简单示例 (TestTim.f)：

```
      subroutine startclock
      common / myclock / mytime
      integer mytime, time
      mytime = time()
      return
      end
      function wallclock()
      integer wallclock
      common / myclock / mytime
      integer mytime, time, newtime
      newtime = time()
      wallclock = newtime - mytime
      mytime = newtime
      return
      end
      integer wallclock, elapsed
      character*24 greeting
      real dtime, timediff, timearray(2)
c      print a heading
      call fdate( greeting )
      print*,  "      Hello, Time Now Is: ",  greeting
      print*,      "See how long 'sleep 4' takes, in seconds"
      call startclock
      call system( 'sleep 4' )
      elapsed = wallclock()
     print*, "Elapsed time for sleep 4 was: ", elapsed," seconds"
c      now test the cpu time for some trivial computing
      timediff = dtime( timearray )
      q = 0.01
      do 30 i = 1, 100000
            q = atan( q )
30      continue
      timediff = dtime( timearray )
     print*, "atan(q) 100000 times took: ", timediff ," seconds"
      end
```

运行该程序会产生以下结果：

```
demo% TimeTest
       Hello, Time Now Is: Thu Feb  8 15:33:36 2001
 See how long 'sleep 4' takes, in seconds
 Elapsed time for sleep 4 was:  4  seconds
 atan(q) 100000 times took:  0.01  seconds
demo%
```

下表列出的例程提供了与 VMS Fortran 系统例程 `idate` 和 `time` 的兼容性。要使用这些例程，必须在 `f95` 命令行中加入 `-lV77` 选项，此时还会得到这些 VMS 版本，而非标准 `f95` 版本。

表 7-4 摘要：非标准 VMS Fortran 系统例程

名称	定义	调用序列	参数类型
`idate`	日期为年、月、日形式	`call idate( d, m, y )`	`integer`
`time`	当前时间为 *hhmmss* 形式	`call time( t )`	`character*8`

注 – `date`(3F) 例程和 `idate`(3F) 的 VMS 版本会发生千年虫问题，因为它们返回的年份值为 2 位。通过减去这些例程返回的日期来计算持续时间的程序，在 1999 年 12 月 31 日之后，其计算结果将是错误的。应代替使用 Fortran 95 例程 `date_and_time`(3F)。有关详细信息，参见 *《Fortran 库参考手册》*。

7.8 疑难解答

当移植到 Fortran 95 的程序不象预期的那样运行时，该怎么办？下面对此提出了几条建议。

7.8.1 结果贴近，但不够贴近

请尝试下列建议：

- 注意大小和工程单位。非常接近于零的数字表面上似乎有区别，但区别并不显著，特别是当该数是两个大数之差时。例如，1.9999999e-30 非常接近于 -9.9992112e-33，即使它们在符号上有区别。

  VAX 数学运算不如 IEEE 数学运算精确，甚至不同的 IEEE 处理器都可能会有区别。特别是当数学运算涉及很多三角函数时，更是如此。这些函数比人们想象的要复杂得多，而且标准只定义了基本运算函数。即使是在 IEEE 机器之间，也可能存在微妙的差别。请回顾本指南第 6 章。

- 试着用 `call nonstandard_arithmetic()` 运行。这样做还可大幅提高性能，并使 Sun 工作站操作起来更象是 VAX 系统。如果您有权访问 VAX 或某一其它系统，请照此行事。很多数值应用程序在每一种浮点实现上产生的结果会稍微有些不同，这一点相当常见。

- 检查 NaN、+Inf 及其它可能的错误迹象。有关如何捕获各种异常的说明，参见本指南第 6 章或手册页 `ieee_handler`(3m)。在大多数机器上，这些异常只是中止运行。

- 相差 6 x 10^{29} 的二个数仍可以具有相同的浮点形式。在以下示例中，不同数字具有相同的表示形式：

```
      real*4 x,y
      x=99999990e+29
      y=99999996e+29
      write (*,10) x, x
10    format('99,999,990 x 10^29 = ', e14.8, ' = ', z8)
      write(*,20) y, y
20    format('99,999,996 x 10^29 = ', e14.8, ' = ', z8)
      end
```

其输出是：

```
99,999,990 x 10^29 = 0.99999993E+37 = 7CF0BDC1
99,999,996 x 10^29 = 0.99999993E+37 = 7CF0BDC1
```

在本例中，差别达 6 x 10^{29}。 IEEE 单精度运算是造成此种无法区分的巨大差距的原因，对于任一十进制到二进制的转换，只能保证六位十进制数字。您有可能能够正确转换七位或八位数字，但这取决于具体的数。

7.8.2 程序失败而不警告

如果程序失败而未发出警告，并且在两次失败之间运行时间长度不同，那么：

- 用最小优化 (`-O1`) 进行编译。如果此时程序工作正常，请以更高的优化级别只编译挑选出的例程。
- 优化程序必须对程序作出假设，应理解这一点。非标准编码或构造均能造成问题。几乎没有任何优化程序能以所有优化级别处理所有程序。（参见第 7-6 页中的 “别名使用和 `-xalias` 选项”）

第 8 章

性能剖析

本章介绍如何测量和显示程序性能。了解程序计算周期中最耗时的地方及其系统资源使用效率是性能调节的先决条件。

8.1 Sun Studio 性能分析器

开发高性能应用程序需要综合运用编译器功能、优化例程库以及性能分析工具。

Sun Studio 软件为收集和分析程序性能数据提供了一对完善的工具：

- 收集器在称为剖析的统计基础上收集性能数据。这些数据包括调用栈、微观统计信息、线程同步延迟数据、硬件计数器上溢数据、地址空间数据以及操作系统的汇总信息。
- 性能分析器显示收集器记录的数据，以便您能检查该信息。分析器处理数据，并在程序、函数、调用者 - 被调用者、源码行和反汇编指令级别显示性能的各种度量。这些度量分为三组：基于时钟的度量、同步延迟度量和硬件计数器度量。

  性能分析器还可通过创建映射文件来帮助您对应用程序性能进行细微调节，可以使用该映射文件来改善应用程序地址空间中函数加载的顺序。

这两种工具有助于解答下列各种问题：

- 程序会耗用可用资源中的多少？
- 哪些函数或加载对象耗用的资源最多？
- 哪些源码行和反汇编指令耗用的资源最多？
- 程序在执行中是如何到达该点的？
- 函数或加载对象正在耗用哪些资源？

性能分析器主窗口显示程序函数列表，其中有每个函数的互斥及相容度量。可以用加载对象、线程、轻量进程 (LWP) 和时间片来过滤此列表。对于选中的函数，辅助窗口会显示此函数的调用者和被调用者。该窗口可以用来导航调用树 — 例如，在搜索高度量

值时。另两个窗口显示源代码和反汇编代码，源代码以性能度量逐行进行注解并与编译器注释穿插在一起，反汇编代码以每条指令的度量进行注解。如果可用，源代码和编译器注释将与这些指令穿插在一起。

尽管性能调节不是开发人员的主要职责，但收集器和分析器是为所有软件开发人员使用而设计的。与常用剖析工具 `prof` 和 `gprof` 相比，它们提供了更加灵活、详细和精确的分析，并且不会受 `gprof` 中属性错误的影响。

收集器和分析器均有等效的命令行方式：

- 可以使用 `collect`(1) 命令进行数据收集。
- 可以使用 `collector` 子命令从 `dbx` 中运行收集器。
- `er_print`(1) 命令行公用程序打印输出 ASCII 版本的各种分析器显示信息。
- 命令行公用程序 `er_src`(1) 显示以编译器注释注解的源代码及反汇编代码列表，但不含性能数据。

详细信息见 Sun Studio *《程序性能分析工具》*手册。

8.2 time 命令

收集程序性能及资源利用基本数据，最简单的方法是使用 time (1) 命令或 csh 中的 set time 命令。

用 time 命令运行程序将在程序终止时打印一行计时信息。

```
demo% time myprog
   The Answer is: 543.01
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
demo%
```

以下是解释：

用户 系统 挂钟 资源 内存 I/O 页面调度

- *用户* – 在用户代码中约为 6.5 秒

```
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
```

- *系统* – 在该任务的系统代码中约为 17.1 秒
- *挂钟* – 完成时间 1 分 16 秒
- *资源* – 该程序占用了系统资源的 31%
- *内存* – 共享程序内存为 11 千字节，专用数据内存为 21 千字节
- *I/O* – 读 354 次，写 210 次
- *页面调度* – 缺页 135 次，换出 0 次

8.2.1 time 输出的多处理器解释

当程序在多处理器环境下并行运行时，计时结果需用不同方式进行解释。由于 /bin/time 会累积不同线程上的用户时间，所以只使用挂钟时间。

由于所显示的用户时间包括花费在所有处理器上的时间，所以该值可以相当大，用于测量性能并不很好。最好是进行实时测量，即使用挂钟时间。这也意味着要获得并行程序的精确计时，必须在只供您程序使用的闲适系统中运行它。

8.3 tcov 剖析命令

在与用 -xprofile=tcov 选项编译的程序一起使用时， tcov(1) 命令会产生源代码的逐句剖析数据，显示所执行的语句及其执行频度。它还会给出有关程序基本块结构的信息摘要。

增强的语句级覆盖由 -xprofile=tcov 编译器选项和 tcov -x 选项调用。输出源文件的副本，并在页边空白处注以语句执行计数。

注 – 如果编译器已内联了例程调用，则 tcov 产生的代码覆盖报告将是不可靠的。无论何时，只要合适，编译器均会以 -O3 以上的优化级别并根据 -inline 选项内联调用。有内联发生时，编译器会用被调用例程的实际代码替换例程的调用。而且，由于无任何调用，所以 tcov 不会报告对这些内联例程的引用。因此，要获得精确的覆盖报告，请不要启用编译器内联。

8.3.1 增强的 tcov 分析

要使用 tcov，请用 -xprofile=tcov 进行编译。运行程序时，覆盖数据被存储在 *program*.profile/tcovd 中，其中 *program* 是可执行文件的名称。（如果可执行文件是 a.out，则会创建 a.out.profile/tcovd。）

运行 tcov -x *dirname source_files*，创建与每个源文件合并在一起的覆盖分析。报告被写至当前目录下的 *file*.tcov。

运行一个简单示例：

```
demo% f95 -o onetwo -xprofile=tcov one.f two.f
demo% onetwo
     ... 程序输出
demo% tcov -x onetwo.profile one.f two.f
demo% cat one.f.tcov two.f.tcov
                        program one
      1 ->              do i=1,10
     10 ->                    call two(i)
                        end do
      1 ->              end
       ..... 等等
demo%
```

可以使用环境变量 $SUN_PROFDATA 和 $SUN_PROFDATA_DIR 指定中间数据收集文件的存放位置。这些文件是由旧式和新式 tcov 分别创建的 *.d 和 tcovd 文件。

这些环境变量可以用来分离来自不同次运行的收集数据。只要设置了这些变量，运行程序便会将执行数据写至 $SUN_PROFDATA_DIR/$SUN_PROFDATA/ 中的文件。

同样， tcov 读取的目录也通过 tcov -x $SUN_PROFDATA 指定。如果设置了 $SUN_PROFDATA_DIR， tcov 会预置它，同时在 $SUN_PROFDATA_DIR/$SUN_PROFDATA/ 中查找文件，而不是在工作目录中查找。

随后的每一次运行都会将更多的覆盖数据累积到 tcovd 文件中。当程序在相应源文件重新编译后首次执行时，会将每个目标文件的数据全部清零。删除 tcovd 文件会将整个程序的数据全部清零。

有关详细信息，参见 tcov(1) 手册页。

第 9 章

性能与优化

本章探讨一些可以提高数值密集型 Fortran 程序性能的优化技术。正确使用算法、编译器选项、库例程和编码习惯可以显著增进性能。本章不讨论高速缓存、I/O 或系统环境调节。并行化问题在下章论述。

本章探讨的问题包括：

- 可以提高性能的编译器选项
- 利用运行时性能配置文件中的反馈信息进行编译
- 使用公用过程已优化的库例程
- 用于提高关键循环性能的编码策略

优化与性能调节这一主题非常复杂，无法在此面面俱到。但本章的讨论将使读者对于这些问题获得初步的有益认识。本章最后列出的书籍对这一主题进行了更加深入的全面论述。

优化与性能调节是一门艺术，它在很大程度上依赖于判定优化或调节*内容*的能力。

9.1 编译器选项的选择

正确选择编译器选项是提高性能的第一步。Sun 编译器提供了范围广泛的选项，这些选项会对目标代码产生影响。缺省情况下，如果编译命令行未显式声明任何选项，则大多数选项均为*关闭*。要提高性能，必须显式选择这些选项。

缺省时，性能选项通常都是关闭的，因为大多数优化都会强制编译器对用户源代码作出假设。符合标准编码习惯并且没有引入潜在副作用的程序应该可以正确优化。但是，用标准习惯取得特权的程序可能会与编译器的某些假设发生冲突。虽然最终代码的运行速度有可能加快，但计算结果却可能是错误的。

建议习惯是：先关闭所有选项进行编译，验证计算结果是否正确和准确，然后用这些初始结果和性能配置文件作为基准。接着，按步骤继续进行 — 用其它选项重新编译并根据基准比较执行结果和性能。如果数值结果有变化，则程序可能存在可疑代码，需要进行仔细分析，确定可疑代码位置并重新编写。

如果增加优化选项后性能没有明显提高甚至降低，则说明编码可能没有给编译器提供进一步提高性能的机会。那么，为获得更好的性能，下一步应在源代码级分析并重新构造程序。

9.1.1 性能选项

下表列出的编译器选项为用户提供了在缺省编译之上提高程序性能的一整套策略。表中只列出了更加有效的编译器性能选项中的一些选项。更完整的列表见 *《Fortran 用户指南》*。

表 9-1 一些有效性能选项

操作	选项
同时使用优化选项组合	`-fast`
将编译器优化级别设置为 *n*	`-O`*n* (`-O = -O3`)
指定通用目标硬件	`-xtarget=`*sys*
指定特殊指令集架构	`-xarch=`*isa*
使用性能配置文件数据进行优化 （用 `-O5`）	`-xprofile=use`
按 *n* 值展开循环	`-unroll=`*n*
允许简化和优化浮点	`-fsimple=1|2`
执行依赖性分析以优化循环	`-depend`
执行过程间优化	`-xipo`

这些选项中的某些选项会增加编译时间，因为它们会调用更深层的程序分析。当例程与其调用例程被一起收入文件中 （而不是将每个例程分割到自己的文件中）时，一些选项工作效果最佳；这样做将允许进行全局分析。

9.1.1.1 `-fast`

此单个选项会选用许多性能选项。

注 – 该选项定义为其它选项的特殊选择集，它会随版本及编译器的不同而变化。另外，`-fast` 选用的某些选项并非在所有平台上都可用。用 `-v` （冗余）标志编译可查看 `-fast` 的展开形式。

`-fast` 可为某些基准测试应用程序提供高性能。但特殊选择的选项不一定都适用于您的应用程序。使用 `-fast` 是编译应用程序以获得最佳性能的良好起点。但仍可能需要额外进行调节。如果用 `-fast` 编译时程序行为不正常，请仔细查看组成 `-fast` 的各个选项，只调用那些适用于您程序的选项，以使程序保持正确的行为。

另请注意，用 `-fast` 编译的程序对于一些数据集可能会表现出良好的性能和精确的结果，而对于另一些数据集则不然。避免用 `-fast` 编译那些依赖浮点运算特殊属性的程序。

由于 -fast 选用的某些选项具有链接蕴含式，因此，如果分步进行编译和链接，还请务必用 -fast 进行链接。

-fast 会选用下列选项：

- -dalign
- -depend
- -fns
- -fsimple=2
- -ftrap=common
- -libmil
- -xtarget=native
- -O5
- -xlibmopt
- -pad=local
- -xvector=yes
- -xprefetch=yes
- -xprefetch_level=2

-fast 为运用编译器的诸多强大优化能力提供了一条捷径。可以单独指定复合选项中的每一个，并且每一选项都可能具有副作用，对此应引起注意 （有关论述见 *《Fortran 用户指南》*）。随 -fast 一起使用其它选项可进一步增加优化。例如：

```
f95 -fast -xarch=v9a ...
```

为具有 64 位能力的 UltraSPARC Solaris 平台编译。

因为 -fast 会调用 -dalign、-fns、-fsimple=2，所以用 -fast 编译的程序会导致非标准浮点运算、非标准数据对齐以及非标准表达式求值顺序。这些选择项可能对大多数程序都不适用。

9.1.1.2 -O*n*

除非显式指定 -O 选项 （或用类似 -fast 的宏选项隐式指定），否则编译器不会执行任何优化。几乎在所有情况下，在编译时指定优化级别都会提高程序执行性能。另一方面，优化级别越高编译时间就越长，并有可能显著增加代码长度。

对于大多数情况，采用 -O3 级别可在性能增益、代码长度和编译时间之间取得良好的平衡。-O4 级别将同一源文件中所含例程调用的自动内联添加作为调用者例程，除此之外它还会做一些其它事情。（有关子程序调用内联的进一步信息，参见 *《Fortran 用户指南》*。）

-O5 级别会增添更多积极主动的优化技术，这些技术在更低级别不适用。一般而言，仅对于那些构成程序计算强度最高部分并因此而具有较高性能提高余地的例程，方应为其指定 -O3 以上的级别。（将用不同优化级别编译的程序部分链接起来，不存在任何问题。）

9.1.1.3 PRAGMA OPT=*n*

使用 C$ PRAGMA SUN OPT=*n* 指令可为一个源文件中的各个例程设置不同的优化级别。该指令将覆盖编译器命令行中的 -O*n* 标志，但必须与 -xmaxopt=*n* 标志联用方可设置*最高*优化级别。有关详细信息，参见 f95(1) 手册页。

9.1.1.4 利用运行时配置文件反馈信息进行优化

当编译器在 O3 以上级别应用优化策略时，如果将该选项与 -xprofile=use 结合起来，将会大大提高效率。利用该选项，可以通过具有典型输入数据的程序（用 -xprofile=collect 编译）所产生的运行时执行配置文件来指导优化器。反馈配置文件会为编译器指出在哪里优化将会获得最大效果。这对于 -O5 选项可能尤为重要。下面给出了一个具有较高优化级别的配置文件集合的典型示例：

```
demo% f95 -o prg -fast -xprofile=collect prg.f ...
demo% prg
demo% f95 -o prgx -fast -O5 -xprofile=use:prg.profile prg.f ...
demo% prgx
```

例中的首次编译会生成一个在运行时产生语句覆盖统计的可执行文件。第二次编译使用该性能数据来指导程序的优化。

（有关 -xprofile 选项的详细信息，参见 *《Fortran 用户指南》*。）

9.1.1.5 -dalign

使用 -dalign，只要有可能，编译器就能生成双字加载 / 存储指令。用该选项编译后，执行大量数据操作的程序可能会显著受益。（它是 -fast 选用的选项之一。）双字指令的速度差不多是相应的单字操作的二倍。

但是，用户应注意，对于一些程序编码期待 COMMON 块中的数据按特定方式对齐的程序，使用 -dalign 选项（因此，对于 -fast 亦是如此）可能会带来问题。使用 -dalign，编译器可能会添加补白以确保所有双精度（和四精度）数据（REAL 或 COMPLEX）在双字边界对齐，结果会造成：

- COMMON 块因添加了补白而有可能比预期的要大。
- 共享 COMMON 的程序单元，只要其中一个用 -dalign 编译，则必须全部用 -dalign 进行编译。

例如，如果某个程序是以单个数组形式通过别名使用具有混合数据类型的整个 COMMON 块来写入数据的，则该程序在 -dalign 下可能不会正常工作，原因是块比程序预期的要大（因双精度和四精度变量的补白所致）。

9.1.1.6 -depend

（在 SPARC 平台上）为 -O3 以上的优化级别增添 -depend 会扩展编译器优化 DO 循环和循环嵌套的能力。使用该选项，优化器会分析迭代间的数据依赖性，以确定是否执行某一些循环结构转换。只能重构无数据依赖性的循环。但是，增添的分析可能会增加编译时间。

9.1.1.7 -fsimple=2

除非指示这样做，否则编译器不会尝试简化浮点计算 （缺省设置为 -fsimple=0）。-fsimple=2 使优化器能够进行更富有成效的简化，同时要了解，由于舍入影响，这可能会导致一些程序产生稍稍不同的结果。如果使用 -fsimple 级别 1 或级别 2，所有程序单元应以类似方式进行编译以确保数值精度的一致性。有关该选项的重要信息，参见 *《Fortran 用户指南》*。

9.1.1.8 -unroll=*n*

展开具有长迭代计数的短循环对某些例程很有利。但是，展开也会增加程序长度，甚至可能会降低其它循环的性能。如果 *n*=1 （缺省值），优化器不会自动展开循环。如果 *n* 大于 1，优化器会尝试展开循环直至达到深度 *n*。

编译器的代码产生器根据因子个数决定是否展开循环。即使该选项是以 *n*>1 指定的，编译器也可能拒绝展开循环。

如果可以展开具有可变循环限制的 DO 循环，已展开的版本和原始循环均会被编译。对迭代计数进行的运行时测试将决定是否适合执行已展开的循环。循环展开，特别是对于只有一条或两条语句的简单循环，会增加每次迭代执行的计算量，并且会为优化器提供调度寄存器和简化操作的更好机会。迭代次数间的权衡、循环的复杂性以及展开深度的选择都不易确定，并且可能需要进行一些试验。

下例展示简单循环是如何有可能用 -unroll=4 展开成四级深度的 （源代码不会随该选项而改变）：

```
原始循环：
     DO I=1,20000
        X(I) = X(I) + Y(I)*A(I)
     END DO

经 4 次编译展开后，它会变成类似下面的样子：
     DO I=1, 19997,4
        TEMP1 = X(I) + Y(I)*A(I)
        TEMP2 = X(I+1) + Y(I+1)*A(I+1)
        TEMP3 = X(I+2) + Y(I+2)*A(I+2)
        X(I+3) = X(I+3) + Y(I+3)*A(I+3)
        X(I) = TEMP1
        X(I+1) = TEMP2
        X(I+2) = TEMP3
     END DO
```

本例展示了一个具有固定循环计数的简单循环。对于可变循环计数，重构将更加复杂。

9.1.1.9 -xtarget=*platform*

如果编译器具有目标计算机硬件的精确描述，可能会提高一些程序的性能。当程序性能至关重要时，正确说明目标硬件是非常重要的。特别是在较新的 SPARC 处理器上运行时更是如此。但是，对于大多数程序和较早的 SPARC 处理器，性能增益是微不足道的，一般性说明可能就足够了。

*《Fortran 用户指南》*列出了 -xtarget= 识别的所有系统名称。对于任意给定的系统名称 （例如，对于 UltraSPARC-II，名称为 ultra2）， -xtarget 会扩展成与该系统正确匹配的 -xarch、 -xcache 和 -xchip 的组合。优化器使用这些说明来确定遵循的策略和生成的指令。

特殊设置 -xtarget=native 使优化器能够针对主机系统 （执行编译的系统）编译目标代码。当编译和执行均在相同的系统上进行时，这显然是非常有益的。当执行系统未知时，针对*通用*体系结构进行编译较为适宜。因此，缺省设置为 -xtarget=generic，即使它有可能达不到最佳性能。

UltraSPARC-III 支持

-xtarget 和 -xchip 标志都接受 ultra3，并且均会为 UltraSPARC-III 处理器生成优化代码。当在 UltraSPARC-III 平台上编译和运行应用程序时，请指定 -fast 标志，以便为该平台自动选择正确的编译器优化选项。

对于交叉编译 （编译在非 UltraSPARC-III 平台上进行，但生成专用于在 UltraSPARC-III 处理器上运行的二进制代码），使用下列标志：

```
-fast -xtarget=ultra3 -xarch=v8plusb（或 -xarch=v9b）
```

使用 -xarch=v9b 编译可生成 64 位代码。

注意，用 -xarch=v8plusb 或 v9b 特别为 UltraSPARC-III 平台编译的程序将无法在 UltraSPARC-III 以外的平台上运行。使用 -xarch=v8plusa （或 v9a，用于生成 64 位代码）编译的程序可在 UltraSPARC-I、UltraSPARC-II *和* UltraSPARC-III 上兼容运行。

性能剖析 （使用 -xprofile=collect: 和 -xprofile=use:）对于 UltraSPARC-III 平台尤为有效，这是因为它允许编译器标识程序内执行最频繁的段并执行局部优化以获得最佳利益。

9.1.1.10 使用 -xipo 进行过程间优化

这个新的 f95 编译器标志是随 Forte Developer 6 update 2 发行版引入的，它通过调用过程间分析传递来执行整个程序优化。与 -xcrossfile 不同， -xipo 在链接步骤跨越所有目标文件进行优化，而不只限于编译命令中的源文件。

当编译和链接大型多文件应用程序时， -xipo 尤为有用。用 -xipo 编译的目标文件内存有分析信息。这样便能够在源文件和预编译程序文件之上进行过程间分析。

有关如何有效使用过程间分析的详细信息，参见 *《Fortran 用户指南》*。

9.1.1.11 添加 PRAGMA ASSUME 断言

在源代码的关键点处添加 ASSUME 指令，可以揭示用其它方法无法确定的重要程序信息，从而有助于指导编译器的优化策略。例如，可以告诉编译器 DO 循环的行程计数始终大于某个值，或某一 IF 分支很可能不会被执行。基于这些断言，编译器可以使用该信息生成更佳的代码。

作为一项附加的好处，通过启用运行时断言结果为假时的警告消息发布，程序员可以使用 ASSUME 编译指示来验证程序的执行。

有关详细信息，参见 *《Fortran 用户指南》* 第 2 章中的 ASSUME 编译指示介绍以及该手册第 3 章中的 -xassume_control 编译器命令行选项。

9.1.2 其它性能策略

假设您已试验过使用各种优化选项，在编译完程序并测量了实际运行时性能之后，下一步可能就是要仔细观察 Fortran 源程序以确定可以进一步采取什么调节措施。

将注意力集中在那些占用计算时间最多的程序部分，考虑下列策略：

- 用等价的优化库替换手写过程。
- 从关键循环中删除 I/O、调用以及不必要的条件操作。
- 消除有可能抑制优化的别名使用。
- 合理化杂乱无章的代码以使用块 IF。

这些都是一些好的编程习惯，往往可以获得更佳的性能。可以更进一步，为特定硬件配置手动调节源代码。然而，这些尝试可能会进一步使代码变得含糊不清，甚至会使编译器的优化器更难获得显著的性能提高。过度手动调节源代码会掩藏过程的原始意图，并且会对不同体系结构的性能产生重大的有害影响。

9.1.3 使用已优化的库

在大多数情况下，经过优化的商业或共享件库执行标准计算过程远比采用手动编码方式更有效率。

例如， Sun Performance Library™ 是一套经过高度优化的数学子例程，它建立在标准 LAPACK、 BLAS、 FFTPACK、 VFFTPACK 和 LINPACK 库基础上。与手动编码相比，使用这些例程，性能有明显提高。有关详细信息，参见 *《Sun Performance Library 用户指南》*。

9.1.4 消除性能抑制因素

使用 Sun WorkShop 性能分析器标识程序的关键计算部分。然后仔细分析循环或循环嵌套，消除有可能抑制优化器生成优化代码或者不然会降低性能的编码。有许多影响可移植性的非标准编码习惯也可能会抑制编译器的优化。

本章最后所列的某些参考书籍更为详细地论述了用于提高性能的重编程技巧。有三种主要方法值得在此提出。

9.1.4.1 删除关键循环中的 I/O

包含程序主要计算工作的循环或循环嵌套中的 I/O 会严重降低性能。花在 I/O 库上的 CPU 时间数量可能构成了循环所用时间的主要部分。（I/O 还会引起进程中断，因而降低程序处理能力。）尽可能将 I/O 移出计算循环，可以大大减少 I/O 库的调用次数。

9.1.4.2 消除子程序调用

循环嵌套深层调用的子程序可能会被调用数千次。即使每次调用花在每个例程上的时间很少，但累加效果却可能会很大。另外，由于编译器在调用期间不能对寄存器状态作出假设，所以子程序调用会抑制包含这些调用的循环的优化。

子程序调用的自动内联（使用 `-inline=`*x,y,..z* 或 `-O4`）是一种让编译器用子程序本身替换实际调用的方法（即将子程序*拉*到循环中）。要内联的例程的子程序源代码必须与调用例程存在于相同的文件中。

还有其它几种消除子程序调用的方法：

- 使用语句函数。如果正在调用的外部函数是一个简单的数学函数，可以将该函数改写为语句函数或语句函数集。语句函数采用内联编译，可以进行优化。
- 将循环推到子程序中。即改写子程序，减少其调用次数（循环外），并使其在每次调用时能对向量或数组值进行操作。

9.1.4.3 合理化杂乱代码

计算密集型循环内的复杂条件操作对编译器进行的优化尝试具有很强的抑制作用。一般而言，消除所有算术和逻辑 IF 操作而代之以块 IF 操作是一条很好的规则，应予以遵守：

原始代码：

```
    IF(A(I)-DELTA) 10,10,11
10  XA(I) = XB(I)*B(I,I)
    XY(I) = XA(I) - A(I)
    GOTO 13
11  XA(I) = Z(I)
    XY(I) = Z(I)
    IF(QZDATA.LT.0.) GOTO 12
    ICNT = ICNT + 1
    ROX(ICNT) = XA(I)-DELTA/2.
12  SUM = SUM + X(I)
13  SUM = SUM + XA(I)
```

经过整理的代码：

```
    IF(A(I).LE.DELTA) THEN
      XA(I) = XB(I)*B(I,I)
      XY(I) = XA(I) - A(I)
    ELSE
      XA(I) = Z(I)
      XY(I) = Z(I)
      IF(QZDATA.GE.0.) THEN
        ICNT = ICNT + 1
        ROX(ICNT) = XA(I)-DELTA/2.
      ENDIF
      SUM = SUM + X(I)
    ENDIF
    SUM = SUM + XA(I)
```

使用块 IF 不仅可以提高编译器生成优化代码的机会，而且可以增强可读性并确保可移植性。

9.1.5 查看编译器注释

如果用 -g 调试选项进行编译，可使用 er_src(1) 公用程序 （Sun Studio 性能分析工具的一部分）来查看编译器生成的源代码注释。该公用程序还用来查看用所生成的汇编语言注释的源代码。下面例举了 er_src 对一个简单的 do 循环产生的注释：

```
demo% f95 -c -g -O4 do.f
demo% er_src do.o
源文件：/home/user21/do.f
目标文件：do.o
加载对象：do.o

     1.          program do
     2.          common aa(100),bb(100)

   函数 x 从源文件 do.f 中内联成下行代码
   以下循环展开前通过管道传送时的稳态循环计数 = 3
   以下循环展开了 5 次
   以下循环每次迭代具有 2 次加载、1 次存储、0 次预取、1 次浮点加、1 次浮点减和 0 次浮点除
     3.          call x(aa,bb,100)
     4.          end
     5.                  subroutine x(a,b,n)
     6.                  real a(n), b(n)
     7.                  v = 5.
     8.                  w = 10.

   以下循环展开前通过管道传送时的稳态循环计数 = 3
   以下循环展开了 5 次
   以下循环每次迭代具有 2 次加载、1 次存储、0 次预取、1 次浮点加、1 次浮点减和 0 次浮点除
     9.                  do 1 i=1,n
    10. 1                        a(i) = a(i)+v*b(i)
    11.                  return
    12.                  end
```

注释消息详细说明编译器所采取的优化操作。在例中可以看到：编译器内联了子例程调用并将循环展开了 5 次。仔细查看该信息可能会为进一步使用优化策略提供线索。

有关编译器注释和反汇编代码的详细信息，参见 Sun Studio *《程序性能分析工具》*手册。

9.2 进阶读物

下列参考书提供更多详细信息：

- *High Performance Computing*， Kevin Dowd 和 Charles Severance 编著， O'Reilly & Associates， 1998 年第 2 版
- *Techniques for Optimizing Applications: High Performance Computing*，Rajat Garg 和 Ilya Sharapov 编著， Sun Microsystems Press Blueprint， 2001

第 10 章

并行化

本章概述多处理器并行化并介绍 SPARC 多处理器平台上的 Fortran 95 能力。

另请参见 *Techniques for Optimizing Applications: High Performance Computing*， Rajat Garg 和 Ilya Sharapov 编著， Sun Microsystems BluePrints 出版 (`http://www.sun.com/blueprints/pubs.html`)

10.1 基本概念

应用程序并行化 （或*多线程化*）是指编译程序，使之在多处理器系统或多线程环境中运行。并行化能使单个任务 （如 DO 循环）运行于多个处理器 （或线程）之上，从而有可能显著加快执行速度。

应用程序需要成为多线程程序，方能在类似 Ultra™ 60、 Sun Enterprise™ Server 6500 或 Sun Enterprise Server 10000 的多处理器系统上有效运行。也就是说，可并行执行的任务需要进行标识和重新编写，将其计算分布在多个处理器或线程之上。

应用程序多线程化可以通过正确调用 `libthread` 基元来手动完成。但可能需要进行大量的分析和重编程。（有关详细信息，参见 Solaris *《多线程编程指南》*。）

Sun 编译器能自动生成在多处理器系统上运行的多线程目标代码。作为支持并行机制的基本语言元素， Fortran 编译器将关注焦点放在 DO 循环上。并行化在若干处理器上分配循环的计算工作，*无需修改 Fortran 源程序*。

选择哪些循环进行并行化以及如何分配这些循环可以完全让编译器去决定 (-autopar)，也可以由编程人员使用源代码指令 (-explicitpar) 显式指定，还可以采用组合方式 (-parallel）来实现。

注 – *不能*用任何编译器并行化选项编译自行 （显式）管理线程的程序。显式多线程（调用 libthread 基元）不能与用这些并行化选项编译的例程结合使用。

并非程序中的所有循环都能有益地进行并行化。只包含少量计算工作的循环在并行化后，（与用于启动和同步并行任务的开销相比）有可能实际运行得更慢。另外，有些循环根本不能安全地进行并行化；鉴于语句或迭代间的依赖性，它们在并行运行时会计算出不同的结果。

隐式循环 （例如， IF 循环和 Fortran 95 数组语法）和显式 DO 循环都可以由 Fortran 编译器自动进行并行化。

f95 能够检测出那些可以安全、有益地自动进行并行化的循环。但在大多数情况下，鉴于可能存在的隐藏副作用，分析必然是保守的。（-loopinfo 选项可以产生显示信息，说明哪些循环进行了并行化、哪些未进行并行化。）在循环前面插入源代码指令，可以显式地对分析施加影响，控制如何并行化 （或不并行化）特定的循环。但是，您随后需要负责确保循环的这种显式并行化不会导致错误的结果。

Fortran 95 编译器通过实现 OpenMP 2.0 Fortran API 指令来提供显式并行化。对于传统程序， f95 还支持较早的 Sun 和 Cray 风格的指令。在 Fortran 95、 C 和 C++ 中，OpenMP 已成为显式并行化的非正式标准，建议使用它来取代较早的指令风格。

有关 OpenMP 的详细信息，参见 *《OpenMP API 用户指南》*，或访问 OpenMP 网站 http://www.openmp.org/。

有关传统并行化指令的讨论，参见第 10-21 页中的 “Sun 风格的并行化指令”和第 10-32 页中的 “Cray 风格的并行化指令”。

10.1.1 加速 — 期望目标

如果并行化一个程序使其在四个处理器上运行，那么，能否期待其花费的时间 （大致）是单个处理器时所用时间的四分之一呢 （即四倍*加速*）？

可能不行。可以证明 （依据 Amdahl 法则）：程序的总体加速性能严格受花在并行运行代码上的时间数量的限制。*无论采用多少处理器*都是如此。事实上，如果用 *p* 表示并行模式下花费的总程序执行时间的百分比，则理论加速限度为 100/(100-*p*) ；因此，如果只有 60% 的程序执行是以并行方式进行的，则*最高*加速倍数是 2.5，该值与处理器个数无关。对于只有四个处理器的情况，该程序的理论加速值 (假设可以达到最高效率）只有 1.8 而不是 4。与总开销相比，实际加速较少。

如同优化一样，循环的选择至关重要。如果并行化的循环在总的程序执行时间中只占很小一部分，则只能获得微小的效果。要提高效率，*必须*并行化耗用*大*部分运行时间的循环。因此，第一步先要确定哪些循环是主要的，然后从此开始。

问题量在确定并行运行程序片段并进而确定加速性能中也起着重要作用。增加问题量会增加循环中完成的工作量。三重嵌套循环将会使工作量呈立方级数递增。如果并行化外层嵌套循环，则少量增加问题量便能使性能有显著提高 （与未并行化性能相比）。

10.1.2 程序并行化步骤

下面非常笼统地概括了并行化应用程序所需的步骤：

1. *优化*。使用适当的编译器选项集在单个处理器上获得最佳串行性能。
2. *配置文件*。使用典型测试数据，确定程序的性能配置文件。标识最主要的循环。
3. *基准测试*。确定串行测试结果是准确的。使用这些结果以及性能配置文件作为基准。
4. *并行化*。使用选项和指令组合编译并生成并行化的可执行文件。
5. *验证*。在单处理器和单线程下运行已并行化的程序，检查结果，查找可能已悄悄潜入的不稳定性和编程错误。（将 `$PARALLEL` 或 `$OMP_NUM_THREADS` 设置为 1，参见第 10-7 页中的 “线程数”。）
6. *测试*。在几个处理器上执行各种运行以检查结果。

7. *基准测试*。在专用系统上用不同数目的处理器进行性能测量。测量性能随问题量变化的变化情况 （可量测性）。

8. *重复*步骤 4 到 7。基于性能对并行化方案进行改进。

10.1.3 数据依赖问题

并非所有循环都可以并行化。在多个处理器上以并行方式运行循环通常会导致迭代执行次序紊乱。而且，只要循环中存在数据依赖，以并行方式执行循环的多个处理器便有可能相互干扰。

会引起数据依赖问题的情况包括递归、约简、间接寻址以及依赖于数据的循环迭代。

10.1.3.1 依赖于数据的循环

您可能能够改写循环来消除数据依赖性，使其可以并行化。但需要进行大量的重构工作。

以下是一些通用规则：

- 仅当所有迭代均写至截然不同的内存位置时，循环才是与数据*无关*的。
- 迭代可以从相同位置读取，只要无任何迭代写至这些位置。

这些是进行并行化的一般条件。在决定是否并行化循环时，编译器的自动并行化分析还会考虑其它标准。但是，可以使用指令显式地强制并行化循环，甚至是那些包含抑制因素和产生错误结果的循环。

10.1.3.2 递归

在循环的某一迭代中设置并在后续迭代中使用的变量会引入交叉迭代依赖性，或称*递归*。循环中的递归要求迭代以正确顺序执行。例如：

```
DO I=2,N
   A(I) = A(I-1)*B(I)+C(I)
END DO
```

必须在上一迭代中计算出 A(I) 的值，方能在当前迭代中 （作为 A(I-1)）来使用。要产生正确的结果，迭代 I 必须先完成，迭代 I+1 方可执行。

10.1.3.3 约简

约简操作将数组元素缩减成单个值。例如，在对数组元素求和并送入单个变量时，需要在每次迭代时更新该变量：

```
DO K = 1,N
  SUM = SUM + A(I)*B(I)
END DO
```

如果以并行方式运行该循环的每个处理器均取得了迭代的一些子集，这些处理器将会相互干扰，覆盖 SUM 中的值。为使之正常工作，每个处理器必须一次一个地执行求和，但顺序并不重要。

编译器会将某些常见的约简操作视为特例进行处理。

10.1.3.4 间接寻址

在将值存入循环中的数组时，如果用于索引数组的下标值是未知的，则会产生循环依赖性。例如，如果在索引数组中存在重复的值，间接寻址会依赖于顺序：

```
DO L = 1,NW
  A(ID(L)) = A(L) + B(L)
END DO
```

在例中， ID 中的重复值会造成 A 中的元素被覆盖。在串行情况下，最后存储的是最终值。在并行情况下，顺序是不确定的。所使用的 A(L) 值 （旧的或更新后的）依赖于顺序。

10.1.4 编译以实现并行化

下表展示与并行化相关的 f95 编译选项。

表 10-1 并行化选项

选项	标志
自动 （*单独*）	-autopar
自动和约简	-autopar -reduction
显式 （*单独*）	-explicitpar
自动和显式	-parallel
自动、约简和显式	-parallel -reduction
显示并行化哪些循环	-loopinfo
显示显式情况下的警告	-vpara
在栈中分配局部变量	-stackvar
启用 Sun 风格的 MP 指令	-mp=sun
启用 Cray 风格的 MP 指令	-mp=cray
编译以实现 OpenMP 并行化	-openmp

选项注释：

- -reduction 需要 -autopar。
- -autopar 包括 -depend 和循环结构优化。
- -parallel 等价于 -autopar -explicitpar。
- -noautopar、 -noexplicitpar、 -noreduction 均为否定选项。
- 并行化选项的顺序可以任意，但必须均为小写形式。
- 对于显式并行化的循环，不分析约简操作。
- -openmp 还会自动调用 -stackvar。
- 选项 -loopinfo、-vpara 和 -mp 必须与并行化选项 -autopar、-explicitpar 或 -parallel 中的一个结合使用。

Sun Studio 编译器现在本身支持将 OpenMP 并行化模型作为主并行化模型。如本章所述， Sun 和 Cray 风格的并行化适用于传统应用程序。有关 OpenMP 并行化的详细信息，参见 *《OpenMP API 用户指南》*。

10.1.5 线程数

PARALLEL （或 OMP_NUM_THREADS）环境变量控制程序可用的最大线程数。设置该环境变量可将程序可使用的最大线程数通知给运行时系统。缺省值为 1。一般会将 PARALLEL 或 OMP_NUM_THREADS 变量设置为目标平台上可用的处理器数。

下例展示如何设置环境变量：

```
demo% setenv PARALLEL 4          C shell
                                - 或 -
demo$ PARALLEL=4                   Bourne/Korn shell
demo$ export PARALLEL
```

在本例中，将 PARALLEL 设置为四，使程序执行最多能使用四个线程。如果目标机有四个可用的处理器，这些线程将分别映射到独立的处理器。如果可用处理器数少于四个，则一些线程必须与其它线程在同一处理器上运行，这样可能会降低性能。

SunOS™ 操作系统命令 psrinfo(1M) 显示系统可用处理器的列表：

```
demo% psrinfo
0       联机    始于 03/18/99 15:51:03
1       联机    始于 03/18/99 15:51:03
2       联机    始于 03/18/99 15:51:03
3       联机    始于 03/18/99 15:51:03
```

10.1.6 栈、栈大小和并行化

正在执行的程序为执行该程序的初始线程保留主内存栈，并为每个辅助线程保留不同的栈。栈为临时内存地址空间，用来保存子程序调用期间的参数和 AUTOMATIC 变量。

主栈的缺省大小约为 8 兆字节。Fortran 编译器通常会将局部变量和数组作为 STATIC 进行分配 （而不是在栈中）。但是， -stackvar 选项强制在栈内分配*所有*局部变量和数组 （就像它们是 AUTOMATIC 变量一样）。建议在并行化时使用 -stackvar，因为它会提高优化器并行化循环中子程序调用的能力。对于包含子程序调用的显式并行化循环， -stackvar 是*必需*的。（参见 *《Fortran 用户指南》*中对 -stackvar 的讨论。）

使用 C shell (csh) 时， limit 命令会显示当前主栈大小，而且会对其进行设置：

```
demo% limit                C shell 示例
cputime         无限制
filesize         无限制
datasize        2097148 千字节
stacksize        8192 千字节              <- 当前主栈大小
coredumpsize        0 千字节
descriptors        64
memorysize       无限制
demo% limit stacksize 65536        <- 将主栈设置为 64Mb
demo% limit stacksize
stacksize       65536 千字节
```

对于 Bourne 或 Korn shell，相应的命令为 ulimit：

```
demo$ ulimit -a            Korn Shell 示例
time（秒）          无限制
file（块）           无限制
data（千字节）          2097148
stack（千字节）         8192
coredump（块）      0
nofiles（描述符） 64
vmemory（千字节）       无限制
demo$ ulimit -s 65536
demo$ ulimit -s
65536
```

多线程程序的每个辅助线程都有自己的*线程*栈。该栈模拟初始线程栈，但对于线程是唯一的。线程的 PRIVATE 数组和变量（对于线程是局部的）在线程栈中进行分配。在 SPARC V9 (UltraSPARC) 平台上缺省大小为 8 兆字节，否则为 4 兆字节。可以用环境变量 STACKSIZE 来设置该大小：

```
demo% setenv STACKSIZE 8192     <- 将线程栈大小设置为 8 Mb   C shell
                              - 或 -
demo$ STACKSIZE=8192              Bourne/Korn Shell
demo$ export STACKSIZE
```

对于某些已并行的 Fortran 代码，可能需要将线程栈大小设置为比缺省值大的值。但是，除了反复进行错误试验，不可能知道其确切大小，特别是如果涉及到专用 / 局部数组就更是如此。如果栈大小太小不足以运行线程，程序将会因段故障而中止。

10.2 自动并行化

使用 `-autopar` 和 `-parallel` 选项， `f95` 编译器会自动查找可以有效并行化的 DO 循环。然后对这些循环进行转换，将其迭代均匀分布在可用的处理器上。编译器会生成实现这一目标所需的线程调用。

10.2.1 循环并行化

编译器的依赖性分析会将 DO 循环转换成可并行化的任务。编译器可能会重构循环，分离出将要串行运行的不可并行化部分。然后将工作均匀分布在可用的处理器上。每个处理器执行不同的迭代块。

例如，对于四个 CPU 和具有 1000 次迭代的并行化循环，每个线程将执行含有 250 次迭代的程序块。

处理器 1 执行迭代	1	至	250
处理器 2 执行迭代	251	至	500
处理器 3 执行迭代	501	至	750
处理器 4 执行迭代	751	至	1000

只有不依赖于计算执行顺序的循环才能成功进行并行化。编译器的依赖性分析拒绝对那些具有内在数据依赖性的循环进行并行化。如果不能完全确定循环中的数据流，编译器会保守行事，不进行并行化。另外，如果能确定性能增益抵不上总开销，编译器也不会对循环进行并行化。

注意，编译器总是选择采用*静态*循环调度来对循环进行并行化 — 将循环中的工作简单分成相等的迭代块。其它调度方案可以用本章后面所述的显式并行化指令来指定。

10.2.2 数组、标量和纯标量

从*自动并行化*角度出发，需要下面几个定义：

- *数组*是至少以一维声明的变量。
- *标量*是非数组变量。
- *纯标量*是未别名使用的标量变量 — 未在 EQUIVALENCE 或 POINTER 语句中引用。

示例：数组 / 标量：

```
      dimension a(10)
      real m(100,10), s, u, x, z
      equivalence ( u, z )
      pointer ( px, x )
      s = 0.0
      ...
```

m 和 a 都是数组变量； s 是纯标量。变量 u、x、z 和 px 是标量变量，但不是*纯*标量。

10.2.3 自动并行化标准

不具有任何交叉迭代数据依赖性的 DO 循环由 -autopar 或 -parallel 自动并行化。自动并行化的一般标准是：

- 只有显式 DO 循环和隐式循环 （如 IF 循环和 Fortran 95 数组语法）才可以并行化。
- 循环内每个迭代的*数组*变量值不能依赖于循环内任何其它迭代的*数组*变量值。
- 循环中的计算不能*有条件地*改变循环终止后引用的任何纯标量变量。
- 循环中的计算不能在各次迭代间改变*标量*变量。这称为*循环携带依赖*。
- 循环体内的工作量必须要超过并行化开销。

10.2.3.1 直观依赖性

编译器可能会自动消除看起来会在循环中造成数据依赖的引用。这种转换有许多，其中之一会利用某些数组的专用版本。通常，如果编译器能确定此种数组在原始循环中只是作为临时存储使用，它便会这样做。

示例：使用 -autopar，通过专用数组消除了依赖性：

```
      parameter (n=1000)
      real a(n), b(n), c(n,n)
      do i = 1, 1000              <-- 已并行化
        do k = 1, n
          a(k) = b(k) + 2.0
        end do
        do j = 1, n-1
          c(i,j) = a(j+1) + 2.3
        end do
      end do
      end
```

在例中，外层循环被并行化，并在独立的处理器上运行。虽然内层循环对数组 a 的引用看起来会导致数据依赖，但编译器会生成数组的临时专用副本，使外层循环迭代变得独立。

10.2.3.2 自动并行化抑制因素

在自动并行化情况下，如果存在下列情况，编译器不会对循环进行并行化：

- DO 循环嵌套在已并行化的另一 DO 循环内
- 流控制允许跳出 DO 循环
- 用户级子程序在循环内被调用
- 循环中有 I/O 语句
- 循环内的计算会改变具有别名的标量变量

10.2.3.3 嵌套循环

在多线程、多处理器环境下，并行化循环嵌套中的最外层循环是最有效的，而不是并行化最内层。由于并行处理通常涉及相对较大的循环开销，所以并行化最外层循环会最大程度地减少开销并增加每个线程完成的工作量。在自动并行化情况下，编译器从最外层嵌套循环开始进行循环分析，然后继续向内进行直至找到可并行化的循环。一旦嵌套中的某个循环被并行化，便会略过该并行循环内所包含的循环。

10.2.4 具有约简操作的自动并行化

将数组转换成标量的计算过程称为*约简操作*。典型的约简操作是对向量元素求和或求积。约简操作违反循环中的计算不能在各次迭代间以累积方式改变标量变量这一标准。

示例：向量元素的约简求和：

```
      s = 0.0
      do i = 1, 1000
        s = s + v(i)
      end do
      t(k) = s
```

但是，对于某些操作，如果约简是阻止并行化的唯一因素，仍然可以对循环进行并行化。常见约简操作出现频率很高，因而编译器能够将其视为特例进行并行化。

自动并行化分析不包括约简操作的识别，除非随 `-autopar` 或 `-parallel` 一同指定了 `-reduction` 编译器选项。

如果某一可并行化的循环包含表 10-2 中列出的某一项约简操作，则当指定了 `-reduction` 时，编译器将会对其进行并行化。

10.2.4.1 识别的约简操作

下表列出了编译器识别的约简操作。

表 10-2 识别的约简操作

数学运算	Fortran 语句模板
求和	`s = s + v(i)`
乘积	`s = s * v(i)`
点积	`s = s + v(i) * u(i)`
最小值	`s = amin( s, v(i))`
最大值	`s = amax( s, v(i))`

求或	`do i = 1, n`
	`  b = b .or. v(i)`
	`end do`

求与	`b = .true.`
	`do i = 1, n`
	`  b = b .and. v(i)`
	`end do`

非零元素计数	`k = 0`
	`do i = 1, n`
	`  if(v(i).ne.0) k = k + 1`
	`end do`

识别所有形式的 `MIN` 和 `MAX` 函数。

10.2.4.2 数值准确性和约简操作

由于下列情况，浮点求和或求积约简操作有可能是不准确的：

- 计算并行执行的顺序与在单个处理器上串行执行的顺序不同。
- 计算顺序会影响浮点数的求和或求积结果。硬件浮点加法和乘法不是结合式的。根据操作数的结合方式，可能会产生舍入、上溢或下溢误差。例如，`(X*Y)*Z` 和 `X*(Y*Z)` 可能会得出不同的有效数。

在一些情况下，该误差是不能接受的。

示例：舍入，求介于 –1 和 +1 之间的 100,000 个随机数之和：

```
demo% cat t4.f
      parameter ( n = 100000 )
     double precision d_lcrans, lb / -1.0 /, s, ub / +1.0 /, v(n)
      s = d_lcrans ( v, n, lb, ub ) ! 获得介于-1 和+1 之间的n 个随机数
      s = 0.0
      do i = 1, n
        s = s + v(i)
      end do
      write(*, '(" s = ", e21.15)') s
      end
demo% f95 -O4 -autopar -reduction t4.f
```

结果会随着处理器的个数而变化。下表展示了介于 –1 和 +1 之间的 100,000 个随机数之和。

处理器个数	输出
1	s = 0.568582080884714E+02
2	s = 0.568582080884722E+02
3	s = 0.568582080884721E+02
4	s = 0.568582080884724E+02

在这种情况下，对于随机开始的数据而言， 10^{-14} 阶的舍入误差是可以接受的。有关详细信息，参见 Sun *《数值计算指南》*。

10.3 显式并行化

本部分介绍 `f95` 识别的源代码指令，这些指令用来显式指示要并行化的循环以及要使用的策略。

Fortran 95 编译器现在支持将 OpenMP Fortran API 作为主并行化模型。有关附加信息，参见 *《OpenMP API 用户指南》*。

`f95` 也接受传统 Sun 和 Cray 风格的并行化指令，以便于从其它平台显式移植已并行化的程序。

程序的显式并行化需要预先分析并深入理解应用程序代码以及共享内存并行化概念。

在 DO 循环紧前面放置的指令标记这些循环将要进行并行化。用 -openmp 进行编译，可启用 OpenMP Fortran 95 指令的识别以及并行化 DO 循环代码的生成。（对于传统 Sun 或 Cray 指令，用 -parallel 或 -explicitpar 进行编译。）并行化指令是注释行，用于通知编译器并行化（或不并行化）跟在指令后面的 DO 循环。指令又称*编译指示*。

在选择要标记进行并行化的循环时，要小心行事。即使存在并行运行时会导致循环计算结果错误的数据依赖性，编译器也会为所有标有并行化指令的循环生成线程化的并行代码。

如果用 libthread 基元编写自己的多线程代码，请*不要*使用任何编译器并行化选项 — 编译器不能并行化已使用线程库用户调用并行化的代码。

10.3.1 可并行化的循环

循环在下列情况下适合进行显式并行化：

- 是 DO 循环，但不是 DO WHILE 循环或 Fortran 95 数组语法。
- 循环内每个迭代的数组变量值不依赖于循环内任何其它迭代的数组变量值。
- 如果循环会改变某一标量变量，该变量值在循环终止后不会被使用。此类标量变量在循环终止后不能保证具有定义的值，因为编译器不会自动确保其正确回存。
- 对于每次迭代，循环内调用的任何子程序都不引用或改变其它任何迭代的*数组*变量值。
- DO 循环索引必须是整数。

10.3.1.1 作用域规则：专用和共享

*专用*变量或数组归循环的*单次迭代*专用。在一次迭代中赋予专用变量或数组的值不会传播给循环内的其它任何迭代。

*共享*变量或数组在所有迭代间共享。在某次迭代中赋予共享变量或数组的值为循环内的其它迭代所见。

如果显式并行化的循环包含共享引用，则必须确保共享不会造成正确性问题。编译器在共享变量的更新或访问上不同步。

如果在一个循环内将某变量指定为专用，并且其唯一一次初始化位于另一循环中，则该变量的值在此循环中可以留作未定义。

10.3.1.2 循环中的子程序调用

循环中 （或从被调用例程内调用的任何子程序中）的子程序调用可能会引入数据依赖性，若不通过调用链深入分析数据和控制流，它们可能会被忽视。尽管最好是并行化执行大量工作的最外层循环，但这些循环往往正是涉及子程序调用的循环。

由于这种过程间的分析很困难并且会大大增加编译时间，所以自动并行化模式不会尝试这样做。对于显式并行化，编译器会为标有 PARALLEL DO 或 DOALL 指令的循环生成并行化代码，即使它包含子程序调用。确保此循环以及此循环包括的所有循环 （包括被调用的子程序）中不存在任何数据依赖性仍是程序员的责任。

不同线程多次调用某个例程会造成问题，这些问题源自对互相干扰的局部静态变量的引用。使例程中的所有局部变量均为*自动*而不是*静态*可防止这种情况。此时，子程序的每次调用都会在栈中保留自己唯一的局部变量存储，任何两次调用均不会相互干扰。

在 AUTOMATIC 语句中列出子程序局部变量或用 -stackvar 选项编译子程序，通过这两种方法中的任一种，可以使子程序局部变量变为驻留在栈中的自动变量。但是，DATA 语句中初始化的局部变量必须进行改写，才能在实际赋值中进行初始化。

注 – 将局部变量分配给栈会造成栈溢出。有关增加栈大小的详细信息，参见第 10-7 页中的 “栈、栈大小和并行化”。

10.3.1.3 显式并行化抑制因素

一般而言，只要您显式令其去做，编译器就会对循环进行并行化。但也有例外情况 — 有些循环编译器不会进行并行化。

下面是可检测到的主要抑制因素，这些因素可能会妨碍显式并行化 DO 循环：

- DO 循环嵌套在已并行化的另一 DO 循环内。

  该例外情况也适用于间接嵌套。如果显式并行化包含子例程调用的循环，那么，即使要求编译器并行化该子例程中的循环，这些循环在运行时也不会以并行方式运行。

- 流控制语句允许跳出 DO 循环。
- 循环的索引变量受副作用影响，例如被等价。

用 -vpara 和 -loopinfo 进行编译，如果编译器在显式并行化循环期间检测到问题，您将会获得诊断消息。

下表列出了编译器检测到的典型并行化问题：

表 10-3 显式并行化问题

问题	并行化	警告消息
循环嵌套在并行化了的另一循环内。	否	否
循环在并行化循环体内调用的某个子例程中。	否	否
流控制语句允许跳出循环。	否	是
循环的索引变量受副作用影响。	是	否
循环中的某变量具有循环携带依赖性。	是	是
循环中有 I/O 语句 — *通常是不明智的，因为输出顺序无法预料。*	是	否

示例：嵌套循环：

```
      ...
!$OMP PARALLEL DO
      do 900 i = 1, 1000      !  已并行化 （外层循环）
        do 200 j = 1, 1000    !  未并行化，无警告
           ...
200   continue
900      continue
      ...
```

示例：子例程中已并行化的循环：

```
 program main
       ...
!$OMP PARALLEL DO
       do 100 i = 1, 200       <- 已并行化
         ...
         call calc (a, x)
         ...
100       continue
       ...
subroutine calc ( b, y )
       ...
!$OMP PARALLEL DO
       do 1 m = 1, 1000        <- 未并行化
         ...
1       continue
       return
       end
```

在例中，由于子例程本身是以并行方式运行的，所以子例程中的循环未被并行化。

示例：跳出循环：

```
!$omp parallel do
       do i = 1, 1000      ! ← 未并行化，发出了错误
         ...
         if (a(i) .gt. min_threshold ) go to 20
         ...
       end do
20       continue
       ...
```

如果标记进行并行化的循环外有转跳，编译器会发出诊断错误。

示例：循环中的某个变量具有循环携带依赖性：

```
demo% cat vpfn.f
      real function fn (n,x,y,z)
      real y(*),x(*),z(*)
      s = 0.0
!$omp parallel do private(i,s) shared(x,y,z)
      do  i = 1, n
          x(i) = s
          s = y(i)*z(i)
      enddo
      fn=x(10)
      return
      end
demo% f95 -c -vpara -loopinfo -openmp -O4 vpfn.f
"vpfn.f", 第 5 行: 警告: 循环可能有约束引用的并行
"vpfn.f", 第 5 行: 已并行化，已使用用户 pragma
```

在此，循环被并行化，但在警告中诊断出可能的循环携带依赖性。但要注意，编译器并不能诊断出所有循环依赖性。

10.3.1.4 显式并行化时的 I/O

在下列情况下，可以在并行执行的循环中执行 I/O：

- 来自不同线程的输出相互交错 （程序输出是非确定的），这一点并不重要。
- 可以确保并行执行循环的安全性。

示例：循环中有 I/O 语句

```
!$OMP PARALLEL DO PRIVATE(k)
      do i = 1, 10      !  已并行化
        k = i
        call show ( k )
      end do
      end
      subroutine show( j )
      write(6,1) j
1      format('Line number ', i3, '.')
      end
demo% f95 -openmp t13.f
demo% setenv PARALLEL 4
demo% a.out
Line number    9.
Line number   10.
Line number    4.
Line number    5.
Line number    6.
Line number    1.
Line number    2.
Line number    3.
Line number    7.
Line number    8.
```

但递归的 I/O，即 I/O 语句包含对本身执行 I/O 的函数的调用，将会造成运行时错误。

10.3.2 OpenMP 并行化指令

OpenMP 是用于多处理器平台的并行编程模型，即将成为 Fortran 95、 C 和 C++ 应用程序的标准编程实践方案。它是 Sun Studio 编译器的首选并行编程模型。

要启用 OpenMP 指令，请用 `-openmp` 选项标志进行编译。 Fortran 95 OpenMP 指令用类似注释的 `!$OMP` 标记标识，标记后紧跟指令名和从属子句。

`!$OMP PARALLEL` 指令标识程序中的并行区域。`!$OMP DO` 指令标识并行区域内即将并行化的 `DO` 循环。可以将这些指令合并成单个 `!$OMP PARALLEL DO` 指令，该指令必须置于 `DO` 循环紧前。

OpenMP 规范包括许多用于在程序并行区域中共享和同步工作的指令，还包括用于数据作用域和控制的从属子句。

OpenMP 与传统 Sun 风格指令的一个主要区别是：OpenMP 需要*专用*或*共享*形式的显式数据作用域。

有关详细信息（包括使用 Sun 和 Cray 并行化指令转换传统程序的指导原则），参见*《OpenMP API 用户指南》*。

10.3.3 Sun 风格的并行化指令

用 `-explicitpar` 或 `-parallel` 选项编译时，缺省（或用 `-mp=sun` 选项）会启用传统 Sun 风格的指令。

10.3.3.1 Sun 并行化指令语法

并行指令由一个或多个*指令行*组成。Sun 风格的指令行定义如下：

```
C$PAR  Directive   [ Qualifiers ]     <- 初始指令行
C$PAR& [More_Qualifiers]             <- 可选续行
```

- 指令行区分大小写。
- 指令行以五个字符的标记开始：`C$PAR`、`*$PAR` 或 `!$PAR`。
- 对于固定格式源：
  - *初始*指令行第 6 列为空格。
  - *续*指令行第 6 列为非空格。
  - 除非指定了 `-e` 选项，否则会忽略 72 列以上的所有列。
- 对于 Fortran 95 自由格式源：
  - 标记前允许有前导空格。
  - 唯一识别的标记是 `!$PAR`。
- 限定符后紧跟指令（如果有）— 在同一行或续行。
- 一行中的多个限定符用逗号分隔。
- 忽略指令或限定符前、后或其中的空白。

Sun 风格的并行指令有：

指令	操作
TASKCOMMON	将 COMMON 块中的变量声明成线程专用
DOALL	并行化下一循环
DOSERIAL	不并行化下一循环
DOSERIAL*	不并行化下一循环*嵌套*

Sun 风格并行指令示例：

```
C$PAR TASKCOMMON ALPHA                          声明块为专用
      COMMON /ALPHA/BZ,BY(100)

C$PAR DOALL                                     无限定符

C$PAR DOSERIAL

C$PAR DOALL SHARED(I,K,X,V), PRIVATE(A)
     这一行指令等价于下面的三行指令。
C$PAR DOALL
C$PAR& SHARED(I,K,X,V)
C$PAR& PRIVATE(A)
```

10.3.3.2 TASKCOMMON 指令

TASKCOMMON 指令将全局 COMMON 块中的变量声明为*线程专用*：公共块中声明的每个变量均变为线程的专用变量，但在线程内仍是全局的。只有命名 COMMON 块才能声明为 TASKCOMMON。

指令的语法为：

C$PAR TASKCOMMON *common_block_name*

指令必须在命名块的*每个* COMMON 声明紧后出现。

该指令仅当用 -explicitpar 或 -parallel 进行编译时才有效。否则会忽略指令，并将块视为常规的 COMMON 块。

TASKCOMMON 块中声明的变量在所有 DOALL 循环以及从 DOALL 循环内调用的例程中均被视为线程专用变量。每个线程都会获得各自的 COMMON 块副本，所以一个线程写入的数据对其它线程不是直接可见的。在执行程序串行部分期间，访问的都是初始线程的 COMMON 块副本。

TASKCOMMON 块中的变量不能在任何 DOALL 限定符中出现，例如， PRIVATE、SHARED、 READONLY 等。

在某些但非*所有*编译单元中，将公共块声明为任务公共块是错误的，如果该块是在编译单元中定义的。用 -xcommonchk=yes 标志编译程序，可以启用任务公共块一致性的运行时检查。应仅在程序开发期间启用运行时检查，因为它会降低性能。

10.3.3.3 DOALL 指令

DOALL 指令请求编译器为紧随其后的一个 DO 循环生成并行代码 （如果用 -parallel 或 -explicitpar 选项进行编译）。

注 – 约简操作的分析和转换不在显式并行化循环内执行。

示例：循环的显式并行化：

```
demo% cat t4.f
      ...
C$PAR DOALL
      do i = 1, n
         a(i) = b(i) * c(i)
      end do
      do k = 1, m
         x(k) = x(k) * z(k,k)
      end do
      ...
demo% f95 -explicitpar t4.f
```

10.3.3.4 DOALL 限定符

Sun 风格 DOALL 指令上的所有限定符都是可选的。下表对其进行了总结：

表 10-4 DOALL 限定符

限定符	断言	语法
PRIVATE	不在迭代间共享变量 *u1*、 …	DOALL PRIVATE(*u1*,*u2*,…)
SHARED	在迭代间共享变量 *v1*、 *v2*、 …	DOALL SHARED(*v1*,*v2*,…)
MAXCPUS	至多使用 *n* 个 CPU （线程）	DOALL MAXCPUS(*n*)
READONLY	*不*在 DOALL 循环中修改所列变量	DOALL READONLY(*v1*,*v2*,…)
STOREBACK	保存变量 *v1*、… 的最后一次 DO 迭代值	DOALL STOREBACK(*v1*,*v2*,…)
SAVELAST	保存所有 *专用*变量的最后一次 DO 迭代值	DOALL SAVELAST
REDUCTION	将变量 *v1*、 *v2*、 … 视为*约简*变量。	DOALL REDUCTION(*v1*,*v2*,…)
SCHEDTYPE	将调度类型设置为 *t*。	DOALL SCHEDTYPE(*t*)

PRIVATE(*varlist*)

PRIVATE(*varlist*) 限定符指定列表 *varlist* 中的所有标量和数组都是 DOALL 循环专用的。数组和标量都可以指定为专用的。对于数组来说， DOALL 循环的每个线程均会获得整个数组的一个副本。 DOALL 循环中引用 （但不包含在专用列表中）的其它所有标量和数组均符合其相应的缺省作用域规则。（参见第 10-15 页中的 “作用域规则：专用和共享”。）

示例：指定数组 a 在循环 i 中为专用的：

```
C$PAR DOALL PRIVATE(a)
      do i = 1, n
        a(1) = b(i)
        do j = 2, n
          a(j) = a(j-1) + b(j) * c(j)
        end do
        x(i) = f(a)
      end do
```

SHARED(*varlist*)

SHARED(*varlist*) 限定符指定列表 *varlist* 中的所有标量和数组对于 DOALL 循环皆为共享的。数组和标量都可以指定为共享的。共享标量和数组可以在 DOALL 循环的所有迭代中访问。 DOALL 循环中引用 （但不包含在共享列表中）的其它所有标量和数组均符合其相应的缺省作用域规则。

示例：指定共享变量：

```
C$PAR DOALL SHARED(y)
      do i = 1,n
        a(i) = y
      end do
```

在例中，变量 y 已被指定为其值将为 i 循环迭代所共享的变量。

READONLY(*varlist*)

READONLY(*varlist*) 限定符指定列表 *varlist* 中的所有标量和数组对于 DOALL 循环皆为只读的。只读标量和数组是共享标量和数组的特殊种类，在 DOALL 循环的任何迭代中都不会被修改。将标量和数组指定为 READONLY，将向编译器指出不需要为 DOALL 循环的每个线程使用该标量变量或数组的单独副本。

示例：指定只读变量：

```
      x = 3
C$PAR DOALL SHARED(x),READONLY(x)
      do i = 1, n
        b(i) = x + 1
      end do
```

在上述示例中，x 是一个共享变量，但编译器可以信赖其值在 i 循环的任何迭代中都不会被修改这一实事，因为它已被指定为 READONLY。

STOREBACK(*varlist*)

STOREBACK 标量变量或数组的值是在 DOALL 循环中进行计算的。计算出的值可以在循环终止后使用。换句话说，回存标量或数组的最后一次循环迭代值在 DOALL 循环之后是可见的。

示例：将循环索引变量指定为回存：

```
C$PAR DOALL PRIVATE(x), STOREBACK(x,i)
      do i = 1, n
        x = ...
      end do
      ... = i
      ... = x
```

在上述示例中，变量 x 和 i 都是回存变量，即使这两个变量都是 i 循环专用的。循环之后的 i 值是 n+1，而 x 的值为其在最后一次迭代结束时具有的任何值。

STOREBACK 存在一些潜在问题，应引起注意。

STOREBACK 操作出现在显式并行化循环的最后一次迭代，即使这不是最后更新 STOREBACK 变量或数组值的同一次迭代。

示例：潜在不同于串行版本的 STOREBACK 变量：

```
C$PAR DOALL PRIVATE(x), STOREBACK(x)
      do i = 1, n
        if (...) then
            x = ...
        end if
      end do
      print *,x
```

在上述示例中，打印输出的 STOREBACK 变量 x 的值与 i 循环串行版本所打印输出的值有可能不同。在显式并行化情况下，处理 i 循环最后一次迭代（当 i = n 时）并为 x 执行 STOREBACK 操作的处理器与当前含有 x 最后更新值的处理器有可能不是同一个处理器。编译器会对这些潜在问题发出警告消息。

SAVELAST

SAVELAST 限定符指定所有专用标量和数组均为 DOALL 循环的 STOREBACK 变量。

示例：指定 SAVELAST：

```
C$PAR DOALL PRIVATE(x,y), SAVELAST
      do i = 1, n
        x = ...
        y = ...
      end do
      ... = i
      ... = x
      ... = y
```

在例中，变量 x、 y 和 i 是 STOREBACK 变量。

REDUCTION(*varlist*)

REDUCTION(*varlist)* 限定符指定列表 *varlist* 中的所有变量均为 DOALL 循环的约简变量。*约简*变量 （或数组）的部分值可以分别在各个处理器上计算，并且其最终值可以从所有部分值计算得出。

约简变量列表的存在要求编译器通过为 DOALL 循环生成并行约简代码，将其作为约简循环来处理。

示例：指定约简变量：

```
C$PAR DOALL REDUCTION(x)
      do i = 1, n
        x = x + a(i)
      end do
```

在上述示例中，变量 x 是一个 （*求和*）约简变量； i 循环是一个 （*求和*）约简循环。

SCHEDTYPE(*t*)

SCHEDTYPE(*t*) 指定使用调度类型 *t* 来调度 DOALL 循环。

表 10-5 DOALL SCHEDTYPE 限定符

调度类型	操作
STATIC	对本 DO 循环使用*静态调度*。（这是 Sun 风格 DOALL 的缺省调度。） 将所有迭代一致地分配给所有可用线程。 示例：对于 1000 次迭代和 4 个处理器，每个线程会获得一个包含 250 次相邻迭代的块。
SELF[(*chunksize*)]	对本 DO 循环使用*自我调度*。 每个线程每次会获得一个包含 *chunksize* 次迭代的块，迭代以不确定顺序分配，直至处理完所有迭代。迭代块可能不是一致地分配给所有可用线程。 • 如果未提供 *chunksize*，编译器会选择一个值。 示例：对于 1000 次迭代且 *chunksize* 为 4，每个线程每次会获得 4 次迭代，直到处理完所有迭代。
FACTORING[(*m*)]	对本 DO 循环使用*因子分解调度*。 若初始为 n 次迭代且有 k 个线程，会将所有迭代分成迭代块组：第一组具有 k 个块，每个块包含 n/(2k) 次迭代；第二组具有 k 个块，每个块包含 n/(4k) 次迭代，依此类推。每组的 chunksize 为剩余迭代次数除以 2k。由于 FACTORING 是动态的，所以不保证每个线程都会从每个组中恰好获得一个块。 • 至少须为每个线程分配 *m* 次迭代。 • 最后可能会残留一个较小的块。 • 如果未提供 *m*，编译器会选择一个值。 示例：对于 1000 次迭代、 FACTORING(3) 和 4 个线程，第一组具有 4 个块，每个块包含 125 次迭代；第二组具有 4 个块，每个块包含 62 次迭代；第三组具有 4 个块，每个块包含 31 次迭代，依此类推。
GSS[(*m*)]	对本 DO 循环使用*指导式自我调度*。 若初始为 *n* 次迭代且有 *k* 个线程，则： • 将 *n/k* 次迭代分配给第一个线程。 • 将剩余迭代次数除以 *k* 分配给第二个线程，依此类推直到处理完所有迭代。 GSS 是动态的，所以不保证将迭代块一致地分配给所有可用线程。 • 至少须为每个线程分配 *m* 次迭代。 • 最后可能会残留一个较小的块。 • 如果未提供 *m*，编译器会选择一个值。 示例：对于 1000 次迭代、GSS(10) 以及 4 个线程，会将 250 次迭代分配给第一个线程，接着将 187 次迭代分配给第二个线程，接着再将 140 次迭代分配给第三个线程，依此类推。

多个限定符

限定符可以因累积效果出现多次。在限定符发生冲突时，编译器会发出警告消息，并且会以最后出现的限定符为先。

示例：三行 Sun 风格的指令 （注意， MAXCPUS、 SHARED 和 PRIVATE 限定符有冲突）：

```
C$PAR DOALL MAXCPUS(4), READONLY(S), PRIVATE(A,B,X), MAXCPUS(2)
C$PAR DOALL SHARED(B,X,Y), PRIVATE(Y,Z)
C$PAR DOALL READONLY(T)
```

示例：与上述三行指令等价的单行指令：

```
C$PAR DOALL MAXCPUS(2),PRIVATE(A,Y,Z),SHARED(B,X),READONLY(S,T)
```

10.3.3.5 DOSERIAL 指令

DOSERIAL 指令禁用指定循环的并行化。该指令适用于紧随其后的那一个循环。

示例：排除一个循环，不对其进行并行化：

```
      do i = 1, n
C$PAR DOSERIAL
        do j = 1, n
          do k = 1, n
               ...
          end do
        end do
      end do
```

在例中，当用 -parallel 进行编译时，编译器不会对 j 循环进行并行化，但 i 循环或 k 循环可能会被并行化。

10.3.3.6 DOSERIAL* 指令

DOSERIAL* 指令禁用指定循环嵌套的并行化。该指令适用于紧随其后的整个循环嵌套。

示例：排除整个循环嵌套，不对其进行并行化：

```
      do i = 1, n
C$PAR DOSERIAL*
        do j = 1, n
          do k = 1, n
              ...
          end do
        end do
      end do
```

在例中，当用 -parallel 进行编译时，编译器不会对 j 和 k 循环进行并行化，但 i 循环可能会被并行化。

10.3.3.7 DOSERIAL* 与 DOALL 间的互作用

如果为同一循环同时指定 DOSERIAL* 和 DOALL，则最后的一个优先。

示例：同时指定 DOSERIAL* 和 DOALL：

```
C$PAR DOSERIAL*
      do i = 1, 1000
C$PAR DOALL
        do j = 1, 1000
            ...
        end do
      end do
```

在例中， i 循环不会被并行化，但 j 循环会被并行化。

另外， DOSERIAL* 指令的作用域不会超出紧随其后的循环嵌套原文。此指令只限于它所在的同一函数或子例程。

示例：DOSERIAL* 不会扩展至被调用子例程中的循环：

```
      program caller
      common /block/ a(10,10)
C$PAR DOSERIAL*
      do i = 1, 10
        call callee(i)
      end do
      end

      subroutine callee(k)
      common /block/a(10,10)
      do j = 1, 10
        a(j,k) = j + k
      end do
      return
      end
```

在上述示例中，无论是否内联子例程 callee 的调用， DOSERIAL* 只适用于 i 循环，而不适用于 j 循环。

10.3.3.8 Sun 风格指令的缺省作用域规则

对于 Sun 风格 (C$PAR) 的显式指令，编译器使用缺省规则来确定标量或数组是共享的还是专用的。可以越过缺省规则来指定循环内所引用的标量或数组的属性。（对于 Cray 风格的 !MIC$ 指令，循环中出现的所有变量必须在 DOALL 指令中显式声明为共享或专用。）

编译器应用以下缺省规则：

- 将所有标量均视为*专用*。使标量的局部副本对执行循环的每个线程均可用，并且该局部副本只由该线程使用。
- 将所有数组引用均视为*共享*引用。某个线程对数组元素的任意写操作对所有线程均可见。不对共享变量的访问执行同步。

如果循环中存在迭代间依赖性，则执行可能会导致错误的结果。必须确保此类情况不会发生。编译器可能有时能够在编译时检测到这种情况并发出警告，但它不会禁用此类循环的并行化。

示例：通过等价产生的潜在问题：

```
      equivalence (a(1),y)
C$PAR DOALL
      do i = 1,n
        y = i
        a(i) = y
      end do
```

在例中，由于标量变量 y 已等价于 a(1)，而缺省 y 为专用、 a(:) 为共享，因此会出现冲突。当执行已并行化的 i 循环时，这会导致很可能错误的结果。这种情况下不会发出任何诊断消息。

可以使用 C$PAR DOALL PRIVATE(y) 来修复此示例。

10.3.4 Cray 风格的并行化指令

要使用传统 Cray 风格的并行化指令，必须用 -mp=cray 进行编译。

同时用 Sun 和 Cray 指令编译的混合程序单元会产生错误结果。

Sun 与 Cray 指令间的主要区别是：除非指定 AUTOSCOPE，否则 Cray 风格*要求循环中的每个标量和数组的显式作用域*或者为 SHARED 或者为 PRIVATE。

下表展示 Cray 风格指令的语法。

```
!MIC$ DOALL
!MIC$&  SHARED( v1, v2,  … )
!MIC$&  PRIVATE( u1, u2,  … )
    ... 可选的限定符
```

10.3.4.1 Cray 指令语法

并行指令由一个或多个*指令行*组成。指令行采用与 Sun 风格相同的语法定义 （参见第 10-21 页中的 “Sun 风格的并行化指令”），除了以下方面：

- 标记为 CMIC$、 *MIC$ 或 !MIC$，但 f95 自由格式只识别 !MIC$。
- 循环中引用的每个变量或数组均出现在 SHARED 或 PRIVATE 限定符中。

Cray 指令与 Sun 风格类似：

Cray 指令	与 Sun 风格相比
DOALL	不同的限定符及调度集
TASKCOMMON	同 Sun 风格
DOSERIAL	同 Sun 风格
DOSERIAL*	同 Sun 风格

10.3.4.2 DOALL 限定符

对于 Cray 风格的 DOALL，PRIVATE 限定符是必需的。必须将 DO 循环内的每个变量均限定为专用或共享，并且 DO 循环的索引必须始终是专用的。下表总结了可用的 Cray 风格限定符。

表 10-6 DOALL 限定符 （Cray 风格）

限定符	断言
SHARED(*v1, v2, ...*)	在迭代间共享变量 *v1*、 *v2*、 ...。
PRIVATE(*x1, x2, ...*)	不在迭代间共享变量 *x1*、 *x2*、 ...。即每个线程都自身拥有这些变量的专用副本。
AUTOSCOPE	未通过 PRIVATE 或 SHARED 限定符显式限定作用域的无作用域变量和数组依据下列作用域规则来限定作用域。
SAVELAST	保存循环中所有 *专用*变量的最后一次 DO 迭代值。
MAXCPUS(*n*)	至多使用 *n* 个线程。

AUTOSCOPE *自动作用域规则*

指定 AUTOSCOPE 将引导编译器使用下列规则来确定未显式将作用域限定为 PRIVATE 或 SHARED 的变量或数组的作用域。

对于 SHARED 变量或数组，下列任意一项都必须为真：

- 变量或数组是只读的。
- 数组通过循环索引来建立索引。
- 变量或数组先读后写。

对于 PRIVATE 变量或数组，下面一项必须为真：

- 变量或数组先写后读。

尽管如此， AUTOSCOPE 并非总能在编译时确定变量或数组的作用域。条件路径通过循环或通过其它手段能够以编译器无法确定的方式改变作用域。用 PRIVATE 和 SHARED 限定符显式限定变量作用域要安全得多。

Cray 风格的调度限定符

对于 Cray 风格的指令， DOALL 指令允许使用单个调度限定符，例如， !MIC$& CHUNKSIZE(100)。表 10-7 展示了 Cray 风格的 DOALL 指令调度限定符：

表 10-7 DOALL Cray 调度

限定符	断言
GUIDED	采用指导式自我调度来分配迭代。 这种分配方式会最大程度地减少同步开销，同时具有可接受的动态负载平衡。缺省块大小是 64。 GUIDED 等价于 Sun 风格的 GSS(64)。
SINGLE	向每个可用线程分配*一次*迭代。 SINGLE 是动态的并且等价于 Sun 风格的 SELF(1)。
CHUNKSIZE(*n*)	向每个可用线程分配 *n* 次迭代。 *n* 必须为整型表达式。要获得最佳性能， *n* 必须为整型常量。CHUNKSIZE(n) 等价于 Sun 风格的 SELF(n)。 示例：对于 100 次迭代以及 CHUNKSIZE(4)，每个线程每次会获得 4 次迭代。
NUMCHUNKS(*m*)	如果存在 *n* 次迭代，则会将 *n/m* 次迭代分配给每个可用线程。可能会残留一个较小的块。 *m* 为整型表达式。要获得最佳性能， *m* 必须为整型常量。NUMCHUNKS(m) 等价于 Sun 风格的 SELF(n/m)，其中 *n* 为总迭代次数。 示例：对于 100 次迭代以及 NUMCHUNKS(4)，每个线程每次会获得 25 次迭代。

缺省调度类型（当在 Cray 风格的 DOALL 指令中未指定任何调度类型时）是 Sun 风格的 STATIC，它没有等价的 Cray 风格调度类型。

10.4 环境变量

用于并行化的环境变量有以下四个：

- PARALLEL 和 OMP_NUM_THREADS
- SUNW_MP_WARN
- SUNW_MP_THR_IDLE

（另请参见第 10-7 页中的 “栈、栈大小和并行化”中对 STACKSIZE 的讨论。）

10.4.1 PARALLEL 和 OMP_NUM_THREADS

要在多线程环境下运行并行化程序，必须在执行前设置 PARALLEL 或 OMP_NUM_THREADS 环境变量。设置的目的是告知运行时系统程序可以创建的最大线程数。缺省值为 1。一般会将 PARALLEL 或 OMP_NUM_THREADS 变量设置为目标平台上可用的处理器数。示例：SETENV PARALLEL 4

10.4.2 SUNW_MP_WARN

控制运行时多任务库发出的警告消息。如果设置为 TRUE，该库会向 stderr 发出警告消息；若设置为 FALSE，则会禁用警告消息，此为缺省设置。
示例：SETENV SUNW_MP_WARN TRUE

10.4.3 SUNW_MP_THR_IDLE

使用 SUNW_MP_THR_IDLE 环境变量可控制主线程以外的每个线程的任务结束状态，执行程序的并行部分。可以将该值设置为下列某个值：

值	含义
SPIN	线程在完成并行任务后原地空转 （或忙碌等待），直到有新的并行任务到达为止。（缺省值）
SLEEP (*time*)	指定线程完成并行任务后空转等待的时间值。如果在线程空转期间此线程有新任务到达，此线程会立即执行新任务。否则，此线程会进入休眠，当有新任务到达时再被唤醒。 *时间*可以以秒 （（*n*s）或仅用（*n*））或毫秒（*n*ms）为单位指定。 不带参数的 SLEEP 在完成并行任务后立即将线程置于休眠状态。 SLEEP、SLEEP（0）、SLEEP（0s）和 SLEEP（0ms）都是等价的。

未显式指定 SUNW_MP_THR_IDLE 时，缺省设置为 SPIN。

示例：

```
% setenv SUNW_MP_THR_IDLE SLEEP (50ms)
% setenv PARALLEL 4
% myprog
```

在本例中，程序最多会创建四个线程。完成并行任务后，线程会空转 50 ms。如果在这段时间内此线程有新任务到达，线程便会执行该任务。否则，此线程会进入休眠直到有新任务到达为止。

10.5 调试并行化程序

Fortran 源代码：

```
      real x / 1.0 /, y / 0.0 /
      print *, x/y
      end
      character  string*5, out*20
      double precision value
      external exception_handler
      i = ieee_handler('set', 'all', exception_handler)
      string = '1e310'
      print *, 'Input string ', string, ' becomes: ', value
      print *, 'Value of 1e300 * 1e10 is:', 1e300 * 1e10
      i = ieee_flags('clear', 'exception', 'all', out)
      end

      integer function exception_handler(sig, code, sigcontext)
      integer sig, code, sigcontext(5)
      print *, '*** IEEE exception raised!'
      return
      end
```

运行时输出：

```
*** IEEE exception raised!
 Input string 1e310 becomes:  Infinity
 Value of 1e300 * 1e10 is: Inf
 注意：下列 IEEE 浮点陷阱已启用：
  参见 ieee_handler(3M)：
 不精确；   下溢；   上溢；   被零除；   无效操作数；
 IEEE 运算的 Sun 实现在
  《数值计算指南》中讨论。
```

调试已并行化的程序需要做一些额外工作。下列方案提出了处理该任务的方法。

10.5.1 调试时的首要步骤

有一些步骤可以直接进行尝试以确定错误原因。

- 关闭并行化。

  可以采取下列某一步骤：

  - 关闭并行化选项 — 用 -O3 或 -O4 选项进行编译但不使用任何并行化选项，验证程序是否正确工作。
  - 将线程数设置为一，然后打开并行化选项进行编译 — 将环境变量 PARALLEL 设置为 1，运行程序。

    如果问题消失，则可以假定问题是由于使用多线程而引起的。
  - 用 -C 进行编译，另外检查数组引用是否越界。
  - 使用 -autopar 时出现问题可能表明编译器正在进行不该进行的并行化。

- 关闭 -reduction。

  如果正在使用 -reduction 选项，求和约简可能会出现和得出稍稍不同的答案。尝试不用该选项运行。

- 使用 DOSERIAL 指令有选择地禁用各个循环的自动并行化。

- 使用 fsplit。

  如果程序中有许多子例程，可用 fsplit(1) 将它们拆成单独的文件。然后在使用和不使用 -parallel 的情况下编译某些文件，并使用 f95 链接 .o 文件。必须在该链接步骤中指定 -parallel。

  执行二进制文件并验证结果。

  重复该过程直到将问题范围缩小至一个子例程为止。

- 使用 -loopinfo。

  检查哪些循环正在进行并行化、哪些循环未进行并行化。

- 使用伪子例程。

  创建一个不执行任何操作的伪子例程或函数。将该子例程的调用置于几个正在进行并行化的循环内。重新编译并执行。使用 -loopinfo 查看哪些循环正在进行并行化。

  继续该过程直到开始获得正确的结果。

- 使用显式并行化。

  将 C$PAR DOALL 指令添加到一对正在进行并行化的循环中。用 -explicitpar 进行编译，然后执行并验证结果。使用 -loopinfo 查看哪些循环正在进行并行化。该方法允许将 I/O 语句添加到已并行化的循环中。

  重复该过程直到找到造成错误结果的循环为止。

  注意：如果只需要 -explicitpar（没有 -autopar），请*不要*用 -explicitpar 和 -depend 进行编译。该方法与用 -parallel 进行编译是一样的，当然，其中还包含 -autopar。

- 逐次*反向*运行循环。

  将 DO I=1,N 替换为 DO I=N,1,-1。如果结果不同，则表明存在数据依赖性。

- 避免使用循环索引。

替换:

```
    DO I=1,N
      ...
      CALL SNUBBER(I)
      ...
    ENDDO
```

使用:

```
      DO I1=1,N
      I=I1
      ...
      CALL SNUBBER(I)
      ...
    ENDDO
```

10.5.2 使用 dbx 调试并行代码

要对并行循环使用 dbx，请按如下方式临时改写程序：

- 隔离文件中的循环体及其自身的子例程。
- 在原始例程中，用新子例程的调用替换循环体。
- 用 -g 但不用并行化选项编译新的子例程。
- 用并行化选项而不用 -g 编译更改后的原始例程。

示例：手动转换循环以便能够以并行方式使用 dbx：

```
原始代码：
demo% cat loop.f
C$PAR DOALL
      DO i = 1,10
            WRITE(0,*) 'Iteration ', i
      END DO
      END
分成两部分：调用者循环以及作为子例程的循环体
demo% cat loop1.f
C$PAR DOALL
      DO i = 1,10
            k = i
            CALL loop_body ( k )
      END DO
      END

demo% cat loop2.f
      SUBROUTINE loop_body ( k )
      WRITE(0,*) 'Iteration ', k
      RETURN
            END
用并行化选项编译调用者循环，但不使用调试选项
demo% f95 -O3 -c -explicitpar loop1.f
用调试选项编译子程序，但不进行并行化
demo% f95 -c -g loop2.f
将两部分链接起来变成 a.out
demo% f95 loop1.o loop2.o -explicitpar
在 dbx 下运行 a.out，并在循环体子例程内置入断点
demo% dbx a.out      ← 未显示各类 dbx 消息
(dbx) stop in loop_body
(2) 在 loop_body 处停止
(dbx) run
运行：a.out
（进程 id 28163）
dbx 在断点处停止
t@1 (l@1) 停止在文件 “loop2.f” 第 2 行的 loop_body 中

    2           write(0,*) 'Iteration ', k
现在显示 k 的值
(dbx) print k
k = 1               ← 除 1 以外还可能是其它各种值
(dbx)
```

10.6 进阶读物

下列书籍提供更多的信息：

- *Techniques for Optimizing Applications: High Performance Computing*，Rajat Garg 和 Ilya Sharapov 编著， Sun Microsystems Press Blueprint， 2001。
- *High Performance Computing*， Kevin Dowd 和 Charles Severance 编著， O'Reilly and Associates， 1998 年第 2 版。
- *Parallel Programming in OpenMP*， Rohit Chandra 等编著， Morgan Kaufmann Publishers， 2001。
- *Parallel Programming*， Barry Wilkinson 编著， Prentice Hall， 1999。
- *《OpenMP Fortran 95/C/C++ API 用户指南》*

第 11 章

C-Fortran 接口

本章论述 Fortran 与 C 的互操作性方面的问题，内容仅适用于 Sun Studio Fortran 95 和 C 编译器的特定情况。

第 11-28 页中的 “Fortran 2000 与 C 的互操作性”简要讨论了 Fortran 2000 标准草案第 15 部分提出的 C 束定功能。（此标准草案可以在国际 Fortran 标准网站 `http://www.j3-fortran.org` 获得）。 Fortran 95 编译器实现了草案标准中所述的这些功能。

11.1 兼容性问题

大多数 C-Fortran 接口都必须符合下列所有方面：

- 函数和子例程的定义及调用
- 数据类型的兼容性
- 按引用或按值参数传递
- 参数的顺序
- 过程名，大写、小写或带有尾随下划线 (_)
- 向链接程序传递正确的库引用

某些 C-Fortran 接口还必须符合：

- 数组索引及顺序
- 文件描述符和 `stdio`
- 文件权限

11.1.1 函数还是子例程？

*函数*一词在 C 和 Fortran 中有着不同的含义。根据具体情况作出选择很重要：

- 在 C 中，所有子程序都是函数；但是，有些可能会返回空 (void) 值。
- 在 Fortran 中，函数会传递一个返回值，但子例程不传递返回值。

当 Fortran 例程调用 C 函数时：

- 如果被调用的 C 函数返回一个值，会从 Fortran 中将其作为函数来调用。
- 如果被调用的 C 函数不返回值，则将其作为子例程来调用。

当 C 函数调用 Fortran 子程序时：

- 如果被调用的 Fortran 子程序是一个*函数*，会从 C 中将其作为一个返回兼容数据类型的函数来调用。
- 如果被调用的 Fortran 子程序是一个*子例程*，则从 C 中将其作为一个返回 `int`（与 Fortran `INTEGER*4` 兼容）或 `void` 值的函数来调用。如果 Fortran 子例程使用交替返回（此时，返回值为 `RETURN` 语句中的表达式值），会返回一个值。如果 `RETURN` 语句中没有出现表达式，而在 `SUBROUTINE` 语句中声明了交替返回，则会返回零。

11.1.2 数据类型的兼容性

以下各表总结了 Fortran 95（与 C 相比）数据类型的数据大小和缺省对齐。在这两个表中，请注意以下方面：

- C 数据类型 `int`、`long int` 和 `long` 在 32 位环境下是等价的（4 字节）。但在 64 位环境下，当用 `-xarch=v9` 或 `v9a` 进行编译时，`long` 和指针均为 8 个字节。这称为 LP64 数据模式。
- 在 64 位环境下且用 `-xarch=v9` 或 `v9a` 进行编译时，`REAL*16` 和 `COMPLEX*32` 按 16 字节边界对齐。
- 标有 4/8 的对齐表示对齐缺省为 8 字节，但在 COMMON 块中按 4 字节边界对齐。COMMON 中的最大对齐为 4 字节。
- 数组和结构的元素及字段必须兼容。
- 不能按值传递数组、字符串或结构。
- 可以在调用地点使用 `%VAL(`*arg*`)`，按值将参数从 Fortran 95 例程传递给 C 例程。假如 Fortran 例程具有一个显式接口块，该接口块用 `VALUE` 属性声明了伪参数，则可以按值将参数从 C 传递给 Fortran 95。

11.1.2.1 Fortran 95 和 C 的数据类型

下表对 Fortran 95 与 C 的数据类型进行了比较。该表假设未应用影响对齐或提升缺省数据大小的编译选项。

表 11-1 数据大小与对齐 — （以字节为单位）按引用传递 （f95 和 cc）

Fortran 95 数据类型	**C** 数据类型	大小	对齐
`BYTE x`	char x	1	1
`CHARACTER x`	`unsigned char x ;`	1	1
`CHARACTER (LEN=n) x`	`unsigned char x[`*n*`] ;`	*n*	1
`COMPLEX x`	`struct {float r,i;} x;`	8	4
`COMPLEX (KIND=4) x`	`struct {float r,i;} x;`	8	4
`COMPLEX (KIND=8) x`	`struct {double dr,di;} x;`	16	4/8
`COMPLEX (KIND=16) x`	`struct {long double, dr,di;} x;`	32	4/8/16
`DOUBLE COMPLEX x`	`struct {double dr, di;} x;`	16	4/8
`DOUBLE PRECISION x`	`double x ;`	8	4
`REAL x`	`float x ;`	4	4
`REAL (KIND=4) x`	`float x ;`	4	4
`REAL (KIND=8) x`	`double x ;`	8	4/8
`REAL (KIND=16) x`	`long double x ;`	16	4/8/16
`INTEGER x`	`int x ;`	4	4
`INTEGER (KIND=1) x`	`signed char x ;`	1	4
`INTEGER (KIND=2) x`	`short x ;`	2	4
`INTEGER (KIND=4) x`	`int x ;`	4	4
`INTEGER (KIND=8) x`	`long long int x;`	8	4
`LOGICAL x`	`int x ;`	4	4
`LOGICAL (KIND=1) x`	`signed char x ;`	1	4
`LOGICAL (KIND=2) x`	`short x ;`	2	4
`LOGICAL (KIND=4) x`	`int x ;`	4	4
`LOGICAL (KIND=8) x`	`long long int x;`	8	4

11.1.3 大小写敏感性

C 和 Fortran 在大小写敏感性方面采取了截然相反的立场：

- C 区分大小写 — 大小写很重要。
- Fortran 在缺省情况下忽略大小写。

`f95` 缺省通过将子程序名转换成小写来忽略大小写。除了字符串常量之外，它会将所有大写字母都转换成小写字母。

对于大 / 小写问题，有两种常用解决方案：

- 在 C 子程序中，使 C 函数名全为小写。
- 用 `-U` 选项编译 Fortran 程序，该选项会通知编译器保留函数 / 子程序名称的现有大 / 小写区别。

只能采用这两种解决方案中的一种，不能同时采用。

本章大多数示例的 C 函数名均采用小写字母，并且*没有*使用 `f95 -U` 编译器选项。

11.1.4 例程名中的下划线

Fortran 编译器通常会在入口点定义和调用中出现的子程序名末尾添加一个下划线 (_)。该惯例与 C 过程或外部变量不同，它们的名称与用户指定的名称相同。几乎所有 Fortran 库过程名都具有两个前导下划线，以减少与用户指定子例程名的冲突。

对于下划线问题，有三种常用解决方案：

- 在 C 函数中，通过在函数名末尾添加下划线来更改该名称。
- 使用 `C()` 编译指示通知 Fortran 编译器忽略这些尾随下划线。
- 使用 `f95 -ext_names` 选项编译对于无下划线外部名称的引用。

只能使用上述解决方案中的一种。

本章的示例都可以使用 `C()` 编译器编译指示来避免下划线。`C()` 编译指示指令取用外部函数名作为参数。它指出这些函数是用 C 语言编写的，这样， Fortran 编译器就会像通常对待外部名称那样，不追加下划线。特定函数的 `C()` 指令必须出现在该函数的首次引用之前。它还必须出现在包含这一引用的每个子程序中。惯常用法是：

```
      EXTERNAL ABC, XYZ      !$PRAGMA C( ABC, XYZ )
```

如果使用该编译指示， C 函数不需要在函数名后追加下划线。（Pragma 指令在*《Fortran 用户指南》*中进行了介绍。）

11.1.5 按引用或值传递参数

Fortran 例程通常按引用传递参数。在调用中，如果非标准函数 `%VAL()` 中包括一个参数，调用例程会按值传递该参数。

标准 Fortran 95 通过 VALUE 属性和 INTERFACE 块来按值传递参数。参见第 11-19 页中的 “按值传递数据参数”。

C 通常按值传递参数。如果在参数前加上与号操作符 (`&`)， C 会使用指针按引用传递参数。 C 总是按引用传递数组和字符串。

11.1.6 参数顺序

除字符串参数之外， Fortran 和 C 均以相同顺序传递参数。但对于每个字符型参数，Fortran 例程都会传递一个附加参数，用以指定字符串长度。这些参数在 C 中为 `long int` 量，按值进行传递。

参数顺序为：

- 与每个参数相应的地址 （数据或函数）
- 与每个字符参数相应的 `long int` （串长度的完整列表出现在其它参数的完整列表之后）

示例：

Fortran 代码片段：	等价的 **C** 代码片段：
`CHARACTER*7 S`	`char s[7];`
`INTEGER B(3)`	`int b[3];`
…	…
`CALL SAM( S, B(2) )`	`sam_( s, &b[1], 7L ) ;`

11.1.7 数组索引和顺序

Fortran 与 C 的数组索引和顺序不同。

11.1.7.1 数组索引

C 数组总是以 0 开始，而 Fortran 数组缺省以 1 开始。有两种常用的索引处理方法。

- 如上述示例所示，可以使用 Fortran 缺省设置。此时， Fortran 元素 `B(2)` 等价于 C 元素 `b[1]`。

■ 可以指定 Fortran 数组 B 以 B(0) 开始，如下所示：

```
      INTEGER B(0:2)
```

这样， Fortran 元素 `B(1)` 就等价于 C 元素 `b[1]`。

11.1.7.2 数组顺序

Fortran 数组按列主顺序存储：`A(3,2)`

```
A(1,1)  A(2,1)  A(3,1)  A(1,2)  A(2,2)  A(3,2)
```

C 数组按行主顺序存储：`A[3][2]`

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

这对于一维数组不存在任何问题。但对于二维数组，应注意下标在所有引用和声明中是如何出现和使用的 — 可能需要做些调整。

例如，在 C 中进行部分矩阵操作，而后在 Fortran 中完成余下部分，这样做可能会产生混淆。最好是将*整个*数组传递给另一语言中的例程，然后在该例程中执行*所有*矩阵操作，以避免在 C 和 Fortran 中各执行部分操作的情况。

11.1.8 文件描述符和 `stdio`

Fortran I/O 通道采用的是单元号。底层 SunOS 操作系统不处理单元号，而是处理*文件描述符*。 Fortran 运行时系统会不断变换，所以大多数 Fortran 程序没必要识别文件描述符。

许多 C 程序都使用一组称为*标准 I/O*（或 `stdio`）的子例程。有许多 Fortran I/O 函数也使用标准 I/O，它反过来又使用操作系统 I/O 调用。下表列出了这些 I/O 系统的某些特性。

表 11-2　Fortran 与 C 之间的 I/O 比较

	Fortran 单元	标准 I/O 文件指针	文件描述符
文件打开	为读写打开	为读打开、为写打开、为读写打开，或者为追加打开；参见 open(2)	为读打开、为写打开或同时为读写打开
属性	已格式化或未格式化	始终未格式化，但可用格式解释例程进行读或写	始终未格式化
访问	直接或顺序	直接访问，如果物理文件的表示是直接访问；但总是可以按顺序读取	直接访问，如果物理文件的表示是直接访问；但总是可以按顺序读取
结构	记录	字节流	字节流
形式	0-2147483647 间的任意非负整数	指向用户地址空间中结构的指针	0-1023 间的整数

11.1.9　库与使用 f95 命令链接

要链接正确的 Fortran 和 C 库，请使用 `f95` 命令调用链接程序。

示例 1：用编译器进行链接：

```
demo% cc -c someCroutine.c
demo% f95 theF95routine.f someCroutine.o  ← 链接步骤
demo% a.out
 4.0 4.5
 8.0 9.0
demo%
```

11.2 Fortran 初始化例程

用 f95 编译的主程序在程序启动时会调用库中的 f90_init 伪初始化例程。库中的这些例程是不进行任何操作的伪例程。编译器生成的调用将指针传递给程序的参数和环境。这些调用会提供软件挂钩，可以在 C 中用它们来提供您自己的例程，以便在程序启动之前以任何自定义方式初始化程序。

这些初始化例程的一种可能用途是为国际化 Fortran 程序调用 setlocale。由于 setlocale 在 libc 以静态方式链接时不起作用，所以只有以动态方式链接了 libc 的 Fortran 程序才能进行国际化。

库中 init 例程的源代码如下：

```
void f90_init(int *argc_ptr, char ***argv_ptr, Char ***envp_ptr) {}
```

f90_init 由 f95 主程序调用。参数分别被设置为 argc、 argv 和 envp 的地址。

11.3 按引用传递数据参数

在 Fortran 例程与 C 过程之间传递数据的标准方法是按引用传递。对于 C 过程而言，Fortran 子例程或函数调用就像是一个所有参数均用指针表示的过程调用。唯一特殊的是 Fortran 将字符串和函数作为参数和 CHARACTER*n 函数的返回值进行处理的方式。

11.3.1 简单数据类型

对于简单数据类型 （非 COMPLEX 及 CHARACTER 串），将 C 例程中的每个关联参数按指针定义或传递：

表 11-3 传递简单数据类型

Fortran 调用 C	**C 调用 Fortran**

integer i
real r
external CSim
i = 100
call CSim(i,r)
...
---------- 	
void csim_(int *i, float *r)
{
*r = *i;
}
	int i=100;
	float r;
	extern void fsim_(int *i, float *r);
	fsim_(&i, &r);
	...
	---------- 	
	subroutine FSim(i,r)
	integer i
	real r
	r = i
	return
	end

11.3.2 COMPLEX 数据

将 Fortran COMPLEX 数据项作为指针传递给具有两种浮点或两种双精度数据类型的 C 结构：

表 11-4 传递 COMPLEX 数据类型

Fortran 调用 C	**C 调用 Fortran**

`complex w`
`double complex z`
`external CCmplx`
`call CCmplx(w,z)`
`...`
`----------`
`struct cpx {float r, i;};`
`struct dpx {double r,i;};`
`void ccmplx_(`
`struct cpx *w,`
`struct dpx *z)`
`{`
`w -> r = 32.;`
`w -> i = .007;`
`z -> r = 66.67;`
`z -> i = 94.1;`
`}`
	`struct cpx {float r, i;};`
	`struct cpx d1;`
	`struct cpx *w = &d1;`
	`struct dpx {double r, i;};`
	`struct dpx d2;`
	`struct dpx *z = &d2;`
	`fcmplx_( w, z );`
	`...`
	`---------------`
	`subroutine FCmplx( w, z )`
	`complex w`
	`double complex z`
	`w = (32., .007)`
	`z = (66.67, 94.1)`
	`return`
	`end`

在 64 位环境下且用 `-xarch=v9` 进行编译时， `COMPLEX` 值是在寄存器中返回的。

11.3.3 字符串

由于没有标准接口，所以不推荐在 C 与 Fortran 例程间传递字符串。不过，请注意以下方面：

- 所有 C 字符串均按引用传递。
- Fortran 调用会为参数列表中具有字符类型的每个参数传递一个附加参数。此额外参数给出字符串的长度，它等价于按值传递的 C 长整数。（这要依具体实现而定。）额外的串长度参数出现在调用中的显式参数之后。

下例展示了具有字符串参数的 Fortran 调用及其等价的 C 调用：

表 11-5 传递 CHARACTER 串

Fortran 调用：	**等价的 C 调用：**

`CHARACTER*7 S`
`INTEGER B(3)`
`...`
`CALL CSTRNG( S, B(2) )`
`...`
	`char s[7];`
	`int b[3];`
	`...`
	`cstrng_( s, &b[1], 7L );`
	`...`

如果在被调用例程中不需要串长度，则可以忽略额外的参数。但要注意， Fortran 不会自动以 C 期望的显式空字符来终结字符串。该终结符必须由调用程序添加。

字符数组调用与单个字符变量调用看起来一样。会传递数组的起始地址，所使用的长度是数组中单个元素的长度。

11.3.4 一维数组

在 C 中数组下标以 0 开始。

表 11-6 传递一维数组

Fortran 调用 C

```
  integer i, Sum
  integer a(9)
  external FixVec
  ...
  call FixVec ( a, Sum )
  ...
------------------------------
void fixvec_ (
            int v[9], int *sum )
{
  int i;
  *sum = 0;
  for ( i = 0; i <= 8; i++ )
           *sum = *sum + v[i];
}
```

C 调用 Fortran

```
 extern void vecref_
              ( int[], int * );
 ...
 int i, sum;
 int v[9] = ...
 vecref_( v, &sum );
 ...
------------------------------
 subroutine VecRef( v, total)
 integer i, total, v(9)
 total = 0
 do i = 1,9
   total = total + v(i)
 end do
 ...
```

11.3.5 二维数组

C 与 Fortran 间的行列转换。

表 11-7 传递二维数组

Fortran 调用 C

```
REAL Q(10,20)
...
Q(3,5) = 1.0
CALL FIXQ(Q)
...
------------------------------
void fixq_( float a[20][10] )
{
     ...
     a[5][3] = a[5][3] + 1.;
     ...
}
```

C 调用 Fortran

```
 extern void
      qref_( int[][10], int *);
...
  int m[20][10] = ...   ;
  int sum;
...
  qref_( m, &sum );
...
------------------------------
SUBROUTINE QREF(A,TOTAL)
INTEGER A(10,20), TOTAL
DO I = 1,10
  DO J = 1,20
    TOTAL = TOTAL + A(I,J)
  END DO
END DO
...
```

11.3.6 结构

只要相应的元素是兼容的，便可以将 C 和 Fortran 95 派生类型传递给彼此的例程。（f95 接受传统的 STRUCTURE 语句。）

表 11-8 传递传统 FORTRAN 77 STRUCTURE 记录

Fortran 调用 C

```
  STRUCTURE /POINT/
       REAL X, Y, Z
  END STRUCTURE
  RECORD /POINT/ BASE
  EXTERNAL FLIP
  ...
  CALL FLIP( BASE )
  ...
------------------------------
struct point {
   float x,y,z;
};
void flip_( struct point *v )
{
       float t;
       t = v -> x;
       v -> x = v -> y;
       v -> y = t;
       v -> z = -2.*(v -> z);
}
```

C 调用 Fortran

```
struct point {
  float x,y,z;
};
void fflip_ ( struct point *) ;
 ...
 struct point d;
 struct point *ptx = &d;
 ...
 fflip_ (ptx);
 ...
------------------------------
 SUBROUTINE FFLIP(P)
 STRUCTURE /POINT/
     REAL X,Y,Z
 END STRUCTURE
 RECORD /POINT/ P
 REAL T
 T = P.X
 P.X = P.Y
 P.Y = T
 P.Z = -2.*P.Z
 ...
```

表 11-9　传递 Fortran 95 派生类型

Fortran 95 调用 C

```
  TYPE point
    SEQUENCE
    REAL :: x, y, z
  END TYPE point
  TYPE (point) base
  EXTERNAL flip
  ...
  CALL flip( base)
  ...
------------------------------
struct point {
   float x,y,z;
};
void flip_( struct point *v )
{
       float t;
       t = v -> x;
       v -> x = v -> y;
       v -> y = t;
       v -> z = -2.*(v -> z);
}
```

C 调用 Fortran 95

```
struct point {
  float x,y,z;
};
extern void fflip_ (
               struct point *) ;
 ...
 struct point d;
 struct point *ptx = &d;
 ...
 fflip_ (ptx);
 ...
------------------------------
 SUBROUTINE FFLIP( P )
   TYPE POINT
  SEQUENCE
     REAL :: X, Y, Z
   END TYPE POINT
   TYPE (POINT) P
   REAL :: T
   T = P%X
   P%X = P%Y
   P%Y = T
   P%Z = -2.*P%Z
   ...
```

注意， Fortran 95 标准要求派生类型定义中有 SEQUENCE 语句，以确保编译器保持存储序列的顺序。

11.3.7 指针

由于 Fortran 例程按引用传递参数，所以可将 FORTRAN 77 (Cray) 指针作为指针的指针传递给 C 例程。

表 11-10 传递 FORTRAN 77 (Cray) POINTER

Fortran 调用 C	**C 调用 Fortran**

`  REAL X`
`  POINTER (P2X, X)`
`  EXTERNAL PASS`
`  P2X = MALLOC(4)`
`  X = 0.`
`  CALL PASS(P2X)`
`  ...`
`------------------------------`
`void pass_(p)`
`  float **p;`
`{`
`   **p = 100.1;`
`}`
	`extern void fpass_( float** );`
	`...`
	`float *p2x;`
	`...`
	`fpass_(&p2x) ;`
	`...`
	`------------------------------`
	`SUBROUTINE FPASS (P2X)`
	`REAL X`
	`POINTER (P2X, X)`
	`X = 0.`
	`...`

C 指针与 Fortran 95 *标量*指针兼容，但与*数组*指针不兼容。

Fortran 95 用标量指针调用 C

Fortran 95 例程:

```
 INTERFACE
     SUBROUTINE PASS(P)
      REAL, POINTER :: P
     END SUBROUTINE
 END INTERFACE

 REAL, POINTER :: P2X
 ALLOCATE (P2X)
 P2X = 0
 CALL PASS(P2X)
 PRINT*, P2X
 END
```

C 例程:

```
 void pass_(p);
  float **p;
  {
    **p = 100.1;
  }
```

Cray 与 Fortran 95 指针间的主要区别是 Cray 指针的目标始终是已命名的。在许多上下文中，声明 Fortran 95 指针会自动标识其目标。另外，被调用 C 例程还需要显式 INTERFACE 块。

要将 Fortran 95 指针传递给数组或数组段，需要特定的 INTERFACE 块，如下例所示：

Fortran 95 例程：

```
INTERFACE
  SUBROUTINE S(P)
    integer P(*)
  END SUBROUTINE S
END INTERFACE
integer, target:: A(0:9)
integer, pointer :: P(:)
P => A(0:9:2) !! 指针每隔一个元素选择一个 A 元素
call S(P)
...
```

C 例程：

```
void s_(int p[])
{
   /* 改变中间元素 */
   p[2] = 444;
}
```

注意，由于 C 例程 S 不是 Fortran 95 例程，所以不能在接口块中将其定义成假定的形状 (integer P(:))。如果 C 例程需要知道数组的实际大小，必须将其作为参数传递给 C 例程。

再次提请注意， C 与 Fortran 间的下标编排不同， C 数组以下标 0 开始。

11.4 按值传递数据参数

从 C 中调用时， Fortran 95 程序应在伪参数中使用 VALUE 属性，并且应为从 Fortran 95 中调用的 C 例程提供一个 INTERFACE 块。

表 11-11 在 C 与 Fortran 95 之间传递简单数据元素

Fortran 95 调用 C

```
PROGRAM callc
INTERFACE
  INTEGER FUNCTION crtn(I)
!$pragma C(crtn)
  INTEGER, VALUE, INTENT(IN) :: I
  END FUNCTION crtn
END INTERFACE

  M = 20
  MM = crtn(M)
  WRITE (*,*) M, MM
END PROGRAM
---------------------------------
int crtn(int x)
{
  int y;
  printf("%d input \n", x);
  y = x + 1;
  printf("%d returning \n",y);
  return(y);
}
---------------------------------
结果:
20 input
21 returning
 20 21
```

C 调用 Fortran 95

```
#include <stdlib.h>
int main(int ac, char *av[])
{
  to_fortran_(12);
}
---------------------------------

SUBROUTINE to_fortran(i)
 INTEGER, VALUE :: i
 PRINT *, i
 END
---------------------------------
```

注意，如果将要以不同数据类型作为实际参数来调用 C 例程，应在接口块中加入 !$PRAGMA IGNORE_TKR I，以制止编译器在实参与伪参之间要求进行类型、种类以及等级匹配。

对于传统 Fortran 77，按值调用仅对于简单数据可用，并且只能为调用 C 例程的 Fortran 77 例程所用。无法做到让 C 例程调用 Fortran 77 例程并按值传递参数。数组、字符串或结构最好是按引用传递。

要将值由 Fortran 77 例程传递给 C 例程，请使用非标准 Fortran 函数 %VAL(*arg*) 作为调用中的一个参数。

在以下示例中，Fortran 77 例程按值传递 x，按引用传递 y。C 例程同时增加了 x 和 y，但只有 y 发生了改变。

Fortran 调用 C

Fortran 例程：

```
  REAL x, y
  x = 1.
  y = 0.
  PRINT *, x,y
  CALL value( %VAL(x), y)
  PRINT *, x,y
  END
```

C 例程：

```
void value_( float x, float *y)
{
  printf("%f, %f\n",x,*y);
  x = x + 1.;
  *y = *y + 1.;
  printf("%f, %f\n",x,*y);
}
```

编译并运行会产生以下输出结果：

```
      1.00000   0.     来自 Fortran 的 x 和 y
1.000000, 0.000000   来自 C 的 x 和 y
2.000000, 1.000000   来自 C 的新 x 和 y
      1.00000   1.00000  来自 Fortran 的新 x 和 y
```

11.5 返回值的函数

返回 BYTE、INTEGER、REAL、LOGICAL、DOUBLE PRECISION 或 REAL*16 类型值的 Fortran 函数与返回兼容类型的 C 函数是等价的 （参见表 11-1）。字符型函数的返回值存在两个额外参数，复数型函数的返回值存在一个额外参数。

11.5.1 返回简单数据类型

下例返回一个 REAL 或 float 值。 BYTE、 INTEGER、 LOGICAL、 DOUBLE PRECISION 和 REAL*16 的处理方式类似：

表 11-12 返回 REAL 或 Float 值的函数

Fortran 调用 C

```
  real ADD1, R, S
  external ADD1
  R = 8.0
  S = ADD1( R )
  ...
------------------------------
 float add1_( pf )
 float *pf;
 {
     float f ;
     f = *pf;
     f++;
     return ( f );
 }
```

C 调用 Fortran

```
float r, s;
extern float fadd1_() ;
r = 8.0;
s = fadd1_( &r );
 ...
------------------------------
 real function fadd1 (p)
 real p
 fadd1 = p + 1.0
 return
 end
```

11.5.2 返回 COMPLEX 数据

COMPLEX 数据的互操作性情况在 SPARC V8 与 V9 实现之间有所不同。

11.5.2.1 SPARC V8 平台

SPARC V8 平台上返回 COMPLEX 或 DOUBLE COMPLEX 的 Fortran 函数等价于具有指向内存返回值的附加第一参数的 C 函数。Fortran 函数及其相应的 C 函数的一般样式如下：

Fortran 函数	**C 函数**
`COMPLEX FUNCTION CF(`*a1, a2, ..., an*`)`	`cf_ (`*return, a1, a2, ..., an*`)` `struct { float r, i; } *`*return*

表 11-13 返回 COMPLEX 数据的函数 (SPARC V8)

Fortran 调用 C	**C 调用 Fortran**

`COMPLEX U, V, RETCPX`
`EXTERNAL RETCPX`
`U = ( 7.0, -8.0)`
`V = RETCPX(U)`
`...`
`------------------------------`
`struct complex { float r, i; };`
`void retcpx_( temp, w )`
`struct complex *temp, *w;`
`{`
`  temp->r = w->r + 1.0;`
`  temp->i = w->i + 1.0;`
`  return;`
`}`
	`struct complex { float r, i; };`
	`struct complex c1, c2;`
	`struct complex *u=&c1, *v=&c2;`
	`extern retfpx_();`
	`u -> r = 7.0;`
	`u -> i = -8.0;`
	`retfpx_( v, u );`
	`...`
	`------------------------------`
	`COMPLEX FUNCTION RETFPX(Z)`
	`COMPLEX Z`
	`RETFPX = Z + (1.0, 1.0)`
	`RETURN`
	`END`

11.5.2.2 SPARC V9 平台

在 64 位环境下且用 `-xarch=v9` 进行编译时， `COMPLEX` 值是在浮点寄存器中返回的：`COMPLEX` 和 `DOUBLE COMPLEX` 在 `%f0` 和 `%f1` 中，`COMPLEX*32` 在 `%f0`、`%f1`、`%f2` 和 `%f3` 中。对于 `v9`，返回字段皆为浮点类型的结构的 C 函数将在浮点寄存器中返回该结构，如果至多需要 4 个这样的寄存器来执行此项操作。 Fortran 函数及其在 V9 平台上相应的 C 函数的通用样式如下：

Fortran 函数	C 函数
COMPLEX FUNCTION CF(*a1, a2, ..., an)*	struct {float r,i;} cf_ (*a1, a2, ..., an)*

表 11-14　返回 COMPLEX 数据的函数 (SPARC V9)

Fortran 调用 C

```
  COMPLEX U, V, RETCPX
  EXTERNAL RETCPX
  U = ( 7.0, -8.0)
  V = RETCPX(U)
  ...
-------------------------------------------------------------
 struct complex {float r, i; };
 struct complex retcpx_(struct complex *w )
{
     struct complex temp;
     temp.r = w->r + 1.0;
     temp.ii = w->i + 1.0;
     return (temp);
}
```

C 调用 Fortran

```
 struct complex { float r, i; };
 struct complex c1, c2;
 struct complex *u=&c1;
 extern struct complex retfpx_(struct complex *);
 u -> r = 7.0;
 u -> i = -8.0;
 retfpx_( u );
  ...
-------------------------------------------------------------
  COMPLEX FUNCTION RETFPX(Z)
  COMPLEX Z
  RETFPX = Z + (1.0, 1.0)
  RETURN
  END
```

11.5.3 返回 CHARACTER 串

不鼓励在 C 与 Fortran 例程之间传递字符串。但是，具有字符串值的 Fortran 函数等价于具有两个附加第一参数（数据地址和字符串长度）的 C 函数。Fortran 函数及其相应的 C 函数的一般样式如下：

Fortran 函数	**C 函数**

`CHARACTER*`*n* `FUNCTION C(`*a1, ..., an*`)`	`void c_ (`*result, length, a1, ..., an*`)`
	`char` *result*`[ ];`
	`long length;`

以下是一个示例：

表 11-15 返回 CHARACTER 串的函数

Fortran 调用 C	**C 调用 Fortran**

`CHARACTER STRING*16, CSTR*9`
`STRING = ' '`
`STRING = '123' // CSTR('*',9)`
`...`
`------------------------------`
`void cstr_( char *p2rslt,`
`long rslt_len,`
`char *p2arg,`
`int *p2n,`
`long arg_len )`
`{ /* 返回 arg 的 n 个副本 */`
`int count, i;`
`char *cp;`
`count = *p2n;`
`cp = p2rslt;`
`for (i=0; i<count; i++) {`
`*cp++ = *p2arg ;`
`}`
`}`
	`void fstr_( char *, long,`
	`char *, int *, long );`
	`char sbf[9] = "123456789";`
	`char *p2rslt = sbf;`
	`int rslt_len = sizeof(sbf);`
	`char ch = '*';`
	`int n = 4;`
	`int ch_len = sizeof(ch);`
	`…`
	`/* 在 sbf 中将 ch 复制 n 份`
	`*/`
	`fstr_( p2rslt, rslt_len,`
	`&ch, &n, ch_len );`
	`...`
	`------------------------------`
	`FUNCTION FSTR( C, N)`
	`CHARACTER FSTR*(*), C`
	`FSTR = ''`
	`DO I = 1,N`
	`FSTR(I:I) = C`
	`END DO`
	`FSTR(N+1:N+1) = CHAR(0)`
	`END`

在本例中， C 函数和调用 C 例程必须在列表 （字符参数的长度）末尾提供两个额外的初始参数 （指向结果字符串和串长度的指针）和一个附加参数。注意，在从 C 中调用的 Fortran 例程中，需要显式添加一个末尾空字符。 Fortran 字符串缺省不以空字符终结。

11.6 带标号的 COMMON

可以在 C 中使用全局 `struct` 来模拟 Fortran 带标号的 COMMON。

表 11-16 模拟带标号的 COMMON

Fortran COMMON 定义	**C “COMMON”定义**

`COMMON /BLOCK/ ALPHA,NUM`
`...`
	`extern struct block {`
	`float alpha;`
	`int num;`
	`};`
	`extern struct block block_ ;`
	
	`main ()`
	`{`
	`...`
	`block_.alpha = 32.;`
	`block_.num += 1;`
	`...`
	`}`

注意， C 例程建立的外部名必须以下划线结束才能与 Fortran 程序创建的块进行链接。另请注意，可能需要使用 C 指令 `#pragma pack` 来获得与 Fortran 相同的补白。

`f95` 缺省会将公共块中的数据与至多 4 字节边界进行对齐。要获得公共块中所有数据的自然对齐并符合缺省结构对齐，请在编译 Fortran 例程时使用 `-aligncommon=16`。

11.7 在 Fortran 与 C 之间共享 I/O

不推荐混用 Fortran I/O 与 C I/O（同时从 C 和 Fortran 例程中发出 I/O 调用）。最好是*全部*执行 Fortran I/O 或*全部*执行 C I/O，而不是两者同时使用。

Fortran I/O 库大部分是在 C 标准 I/O 库之上实现的。Fortran 程序中的每一个打开单元都有相关联的标准 I/O 文件结构。对于 `stdin`、 `stdout` 和 `stderr` 流，不需要显式引用该文件结构，所以可以进行共享。

如果 Fortran 主程序调用 C 来执行 I/O， Fortran I/O 库必须在程序启动时进行初始化，以便将 0、5 和 6 单元分别连接到 `stderr`、 `stdin` 和 `stdout`。要对打开的文件描述符执行 I/O， C 函数必须考虑 Fortran I/O 环境。

但是，如果 C 主程序调用 Fortran 子程序来执行 I/O，则不用为了将 0、5 和 6 单元连接到 `stderr`、 `stdin` 和 `stdout` 而自动初始化 Fortran I/O 库。该连接通常由 Fortran 主程序来完成。如果 Fortran 函数试图在没有正常初始化 Fortran 主程序 I/O 的情况下引用 `stderr` 流 （0 单元），则会将输出写至 `fort.0` 而不是 `stderr` 流。

通过在程序启动时调用 `f_init()` 库例程以及可选地在程序终止时调用 `f_exit()`，C 主程序可以初始化 Fortran I/O 并建立 0、 5 和 6 单元的预连接。

请记住：即使主程序在 C 中，也应该用 `f95` 链接。

11.8 交替返回

Fortran 77 的交替返回机制已经过时，如果考虑可移植性，不应再使用它。在 C 中没有与交替返回等价的机制，所以只需关注 C 例程调用具有交替返回的 Fortran 例程的情况。 Fortran 95 接受 Fortran 77 的交替返回，但不鼓励使用它。

以下实现返回 RETURN 语句中表达式的 `int` 值。这依赖于具体实现，应避免使用。

表 11-17 交替返回

C 调用 Fortran

```
int altret_ ( int * );
main ()
{
    int k, m ;
    k =0;
    m = altret_( &k ) ;
    printf( "%d %d\n", k, m);
}
------------------------------
SUBROUTINE ALTRET( I, *, *)
  INTEGER I
  I = I + 1
  IF(I .EQ. 0) RETURN 1
  IF(I .GT. 0) RETURN 2
  RETURN
END
```

运行示例

```
demo% cc -c tst.c
demo% f95 -o alt alt.f tst.o
alt.f:
      altret:
demo% alt
1 2
```

C 例程收到的返回值 2 来自 Fortran 例程，因为它执行了 RETURN 2 语句。

11.9 Fortran 2000 与 C 的互操作性

Fortran 2000 标准草案 （可从 `http://www.j3-fortran.org` 获得）为从 Fortran 95 程序内引用 C 编程语言定义的过程和全局变量提供了一种方法。反过来，它又提供了一种定义 Fortran 子程序或全局变量的方法，从而可以从 C 过程中引用它们。

根据设计，采用这些功能实现 Fortran 95 与 C 程序间的互操作性，可确保符合标准的平台间的可移植性。

Fortran 2000 为派生类型提供了 `BIND` 属性，并且提供了 `ISO_C_BINDING` 内在模块。利用此模块可以访问 Fortran 程序的某些支持可互操作对象规范的命名常量、派生类型和过程。详细信息见 Fortran 2000 标准草案第 15 部分。

索引

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z