Sun Studio 12：Fortran 编程指南

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

文件号码 820–1204–10

070723@18135

目录

前言

《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、并行处理以及互操作性。

本指南适用于精通 Fortran 语言并希望了解如何有效使用Sun Fortran 编译器的科学家、工程师和程序员。通常，还假定他们熟悉 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，另请参见其配套手册《Fortran 用户指南》。

印刷约定

表 P–1 字体约定

字体	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `% You have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`%` **`su`** `Password:`
AaBbCc123	要使用实名或值替换的命令行占位符文本	要删除文件，请键入 **`rm`** *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *class* 选项。
新词术语强调	新词或术语以及要强调的词	您**必须**成为超级用户才能执行此操作。
《书名》	书名	阅读《用户指南》的第 6 章。

表 P–2 代码约定

代码符号	含义	表示法	代码示例
[ ]	方括号中的参数是可选参数。	O[*n*]	O4, O
{ }	大括号中是针对所需选项的一组选择内容。	d{y|n}	dy
|	"|" 或 "-" 符号用于分隔多个参数，只能选择其中一个参数。	B{dynamic|static}	Bstatic
:	冒号与逗号类似，有时用于分隔多个参数。	R*dir*[:*dir*]	R/local/libs:/U/a
	省略号表示一系列省略。	xinline=*f1*[,…*fn*]	xinline=alpha,dos

- 符号 ∇ 表示有意义的空格：

∇∇36.001

- FORTRAN 77 标准采用早期惯例，名称 "FORTRAN" 的拼写采用大写字母。当前惯例是使用小写字母："Fortran 95"
- 对联机手册页的引用采用主题名称和章节号。例如，对库例程 GETENV 的引用为 getenv(3F)，这意味着访问此手册页的 man 命令为：man -s 3F getenv。

Shell 提示符

Shell	提示符
C shell	*machine-name*%
C shell 超级用户	*machine-name*#
Bourne shell、Korn shell 和 GNU Bourne-Again shell	$
Bourne shell、Korn shell 和 GNU Bourne-Again shell 超级用户	#

受支持的平台

此 Sun Studio 发行版支持使用 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。可从以下位置获得硬件兼容性列表，在列表中可以查看您正在使用的 Solaris 操作系统版本所支持的系统：`http://www.sun.com/bigadmin/hcl`。这些文档中给出了平台类型间所有实现的区别。

在本文档中，与 x86 相关的术语的含义如下：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 表示有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86”表示有关基于 x86 的系统的特定 32 位信息。

有关受支持的系统，请参阅硬件兼容性列表。

访问 Sun Studio 文档

可以访问以下位置的文档：

- 可以通过随软件一起安装在本地系统或网络上的文档索引（在 Solaris 平台上为 `file:/opt/SUNWspro/docs/index.html`）获取文档。

  如果未将软件安装在 Solaris 平台上的 `/opt` 目录中，请询问系统管理员以获取系统中的等效路径。
- 可以从 `docs.sun.com`™ Web 站点获取大多数手册。下列书目只能从 Solaris 平台上已安装的软件中获取：
  - 《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》

  Solaris 平台和 Linux 平台的相应发行说明可从 `docs.sun.com` Web 站点获取。
- IDE 所有组件的联机帮助可通过 IDE 中的“帮助”菜单以及许多窗口和对话框上的“帮助”按钮获取。

可以通过 Internet 访问 `docs.sun.com` Web 站点 (`http://www.docs.sun.com`) 阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到某手册，请参见随软件一起安装在本地系统或网络上的文档索引。

注 – Sun 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他资料，Sun 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的（或通过它们获得的）任何内容、产品或服务而造成的或连带产生的实际或名义损坏或损失，Sun 概不负责，也不承担任何责任。

采用易读格式的文档

该文档以易读格式提供，以方便残障用户使用辅助技术进行阅读。可以按照下表所述找到文档的易读版本。如果该软件未安装在 `/opt` 目录中，请询问系统管理员以获取系统中的等效路径。

文档类型	易读版本的格式和位置
手册（第三方手册除外）	HTML，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，位于 Solaris 平台上已安装软件中，可通过文档索引 (`file:/opt/SUNWspro/docs/index.html`) 获取
自述文件	HTML，位于 Sun Developer Network 门户 `http://developers.sun.com/sunstudio/documentation/ss12`
手册页	HTML，位于已安装软件中，可通过文档索引（在 Solaris 平台上为 `file:/opt/SUNWspro/docs/index.html`；在 Linux 平台上为 `file:/opt/sun/sunstudio12/docs/index.html`）获取
联机帮助	HTML，可通过 IDE 中的“帮助”菜单和“帮助”按钮获取
发行说明	HTML，位于 `http://docs.sun.com`

相关 Sun Studio 文档

下表列出了可通过 `file:/opt/SUNWspro/docs/index.html` 和 `http://docs.sun.com` 获取的 Solaris 平台相关文档。如果该软件未安装在 `/opt` 目录中，请询问系统管理员以获取系统中的等效路径。

文档标题	说明
《Fortran 库参考》	详细介绍了 Fortran 库和内部例程。
《Fortran 用户指南》	介绍了 `f95` 编译器的编译时环境和命令行选项。
《C 用户指南》	介绍了 `cc` 编译器的编译时环境和命令行选项。
《C++ 用户指南》	介绍了 `CC` 编译器的编译时环境和命令行选项。
《OpenMP API 用户指南》	概括介绍了 OpenMP 多重处理 API，并提供了有关实现的具体信息。
《数值计算指南》	介绍了与浮点计算的数值精度有关的问题。

访问 Solaris 相关文档

下表列出了可从 `docs.sun.com` Web 站点上获取的相关文档。

文档集合	文档标题	说明
Solaris Reference Manual Collection	请参见手册页各章节的标题。	提供 Solaris 操作系统的有关信息。
Solaris Software Developer Collection	《链接程序和库指南》	介绍了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris Software Developer Collection	《多线程编程指南》	介绍了 POSIX 和 Solaris 线程 API、使用同步对象进行编程、编译多线程程序和多线程程序的查找工具。

开发者资源

访问 Sun Developer Network Sun Studio 门户 (`http://developers.sun.com/sunstudio`) 查看下列经常更新的资源：

- 有关编程技术和最佳做法的文章
- 有关编程小技巧的知识库
- 软件文档以及随软件一起安装的文档的更正信息
- 有关支持级别的信息
- 用户论坛
- 可下载的代码示例
- 新技术预览

Sun Studio 门户是 Sun Developer Network Web 站点 (`http://developers.sun.com`) 上面向开发者的众多其他资源之一。

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下 URL：

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

Sun 欢迎您提出意见

Sun 致力于提高其文档的质量，并十分乐意收到您的意见和建议。您可以通过以下 URL 向 Sun 提交您的意见：

http://www.sun.com/hwdocs/feedback

请在电子邮件的主题行中注明文档的文件号码。例如，本文档的文件号码是 820-1204-10。

◆ ◆ ◆ 第 1 章

1 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®、UltraSPARC® 和 x64/x86 平台的 Solaris™ 操作环境下使用。此编译器符合发布的 Fortran 语言标准，并提供很多扩展的功能，其中包括多处理器并行化、高级的优化代码编译以及混合的 C/Fortran 语言支持。

f95 编译器还提供接受大多数传统 Fortran 77 源代码的 Fortran 77 兼容性模式。不再包含单独的 Fortran 77 编译器。有关 FORTRAN 77 兼容性及迁移问题的信息，请参见《Fortran 用户指南》第 5 章。

1.1 标准一致性

- **f95** 符合 ISO/IEC 1539-1:1997 Fortran 标准文档的第一部分。
- 浮点运算基于 IEEE 标准 754-1985 和国际标准 IEC 60559:1989。
- 在 Solaris 和 Linux 平台上，**f95** 提供了对 SPARC 和 x86 系列处理器体系结构（UltraSPARC、SPARC64、AMD64、Pentium Pro 和 Xeon Intel® 64）的优化开发功能的支持。
- Sun Studio 编译器符合 OpenMP 2.5 共享内存并行化 API 规范。有关详细信息，请参见《OpenMP API 用户指南》。
- 在本文档中，“标准”是指与上面列出的标准版本相一致。“非标准”或“扩展”是指超出这些标准版本的功能。

负责标准的一方可能会不时地修订这些标准。可能会修订或替代这些编译器遵循的适用标准的版本，因而导致 Sun Fortran 编译器将来版本中的功能与先前版本不兼容。

1.2 Fortran 95 编译器的功能

Sun Studio Fortran 95 编译器提供以下功能和扩展：

- 在例程中对参数、公共区等进行全局程序一致性检查。
- 优化多处理器系统的自动和显式循环并行化。
- VAX/VMS Fortran 扩展，其中包括：
  - 结构、记录、联合和映射
  - 递归

  OpenMP 2.5 并行化指令。
- 全局、窥孔和潜在的并行化优化可产生高性能的应用程序。基准测试表明优化的应用程序的运行速度比未优化的代码快得多。
- Solaris 系统上的通用调用惯例允许将使用 C 或 C++ 编写的例程与 Fortran 程序结合使用。
- UltraSPARC 和 x64 平台上支持 64 位的 Solaris 环境。
- 使用 **%VAL** 按值进行调用。
- Fortran 77 和 Fortran 95 程序与对象二进制文件之间的兼容性。
- 区间运算编程。
- 某些 Fortran 2003 功能，其中包括流 I/O。

有关随各软件发行版添加到编译器中的新增和扩展功能的详细信息，请参见《Fortran 用户指南》附录 B。

1.3 其他 Fortran 实用程序

以下实用程序可为使用 Fortran 进行软件程序开发提供帮助：

- **Sun Studio 性能分析器**－单线程和多线程应用程序的深层性能分析工具。请参见 **analyzer**(1)。
- **asa**－此 Solaris 实用程序是一个 Fortran 输出过滤器，用于打印第一列中包含 Fortran 回车控制符的文件。可使用 **asa** 将按照 Fortran 回车控制惯例设置格式的文件转换为按照 UNIX 行打印机惯例设置格式的文件。请参见 **asa**(1)。
- **fdumpmod**－显示包含在文件或归档文件中的模块名称的实用程序。请参见 **fdumpmod**(1)。
- **fpp**－Fortran 源代码预处理程序。请参见 **fpp**(1)。
- **fsplit**－此实用程序将一个包含几个例程的 Fortran 文件拆分成几个文件，每个文件包含一个例程。可使用 FORTRAN 77 或 Fortran 95 源文件上的 **fsplit**。请参见 **fsplit**(1)。

1.4 调试实用程序

可以使用以下调试实用程序：

- **`-Xlist`**－一个用于检查例程中参数、COMMON 块等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，还包含一个性能数据收集器。

1.5 Sun 性能库

Sun 性能库™是一个用于计算线性代数和傅立叶变换的优化子例程及函数的库。它基于一般通过 Netlib (**`www.netlib.org`**) 提供的标准库 LAPACK、BLAS1、BLAS2、BLAS3、FFTPACK、VFFTPACK 和 LINPACK。

与标准库版本相比，Sun 性能库中的每个子程序执行相同的操作并且具有相同的接口，但通常这些子程序的速度要快得多且准确得多，这些子程序可以用于多处理环境中。

有关详细信息，请参见 **`performance_library`** 自述文件和《Sun 性能库用户指南》。（性能库例程的手册页位于第 3P 节。）

1.6 区间运算

Fortran 95 编译器提供编译器标记 **`-xia`** 和 **`-xinterval`** 以启用新的语言扩展，并生成相应的代码以执行区间运算。有关详细信息，请参见《Fortran 95 区间运算编程指南》。（只有 SPARC/UltraSPARC 平台支持区间运算功能。）

1.7 手册页

联机手册 (**`man`**) 页提供了关于命令、函数、子例程或这些项的集合的当前文档。应该将用户的 **`MANPATH`** 环境变量设置为安装的 Sun Studio **`man`** 目录路径（用来访问 Sun Studio 手册页）。在 Solaris 上该目录通常是 **`/opt/SUNWspro/man`**。

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，出现的手册页参考带有主题名称和手册章节号：可使用 **`man f95`** 访问 **`f95`**(1)。例如，可在 **`man`** 命令中使用 **`-s`** 选项来访问 **`ieee_flags`**(3M) 指示的其他部分：

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

Fortran 库例程是在手册页第 3F 节中介绍的。

下面列出了对于 Fortran 用户非常重要的 **man** 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器
dbx(1)	命令行交互调试器
fpp(1)	Fortran 源代码预处理器
cpp(1)	C 源代码预处理器
fdumpmod(1)	显示模块 (**.mod**) 文件的内容
fsplit(1)	预处理器将 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)	目标文件的增量链接编辑器
ld(1)	目标文件的链接编辑器

1.8 自述文件

Sun Developer Network (SDN) 门户网站 (`http://developers.sun.com/sunstudio`) 上的**自述文件**页面介绍了新增功能、软件不兼容性、错误以及手册印刷后发现的信息。这些自述文件页面是门户网站上此发行版本文档的一部分，也可以通过已安装软件包含的 HTML 文档索引（位于 `file:/opt/SUNWspro/docs`）链接这些自述文件页面。

表 1–1 重要自述文件页面

自述文件页面	描述...
fortran_95	此版本 Fortran 95 编译器 **f95** 的新增功能、修改的功能、已知限制和文档勘误表。
fpp_readme	**fpp** 功能概述
interval_arithmetic	**f95** 中的区间运算功能概述
math_libraries	可用的优化和专用数学库。
profiling_tools	使用性能配置工具**prof**、**gprof** 和 **tcov**。
runtime_libraries	可依照最终用户许可协议的条款重新分发的库和可执行文件。

表 1–1 重要自述文件页面 （续）

自述文件页面	描述...
performance_library	Sun 性能库概述
openmp	OpenMP 并行化 API 中的新增功能和已更改的功能

可使用 **-xhelp=readme** 命令行选项来显示每个编译器自述文件页面的 URL。例如，命令：

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

显示用于查看 SDN 门户网站上此发行版 **fortran_95** 自述文件的 URL。

1.9 命令行帮助

可通过调用编译器的 **-help** 选项来查看 **f95** 命令行选项的简短描述（如下所示）：

```
%f95 -help=flags
Items within [ ] are optional. Items within < > are variable parameters.
Bar | indicates choice of literal values.
-someoption[={yes|no}] implies -someoption is equivalent to -someoption=yes

_____________________________________________________________________
-a                  Collect data for tcov basic block profiling
-aligncommon[=<a>] Align common block elements to the specified boundary requirement; <a>={1|2|4|8|16}
-ansi               Report non-ANSI extensions.
-autopar            Enable automatic loop parallelization
-Bdynamic           Allow dynamic linking
-Bstatic            Require static linking
-C                  Enable runtime subscript range checking
-c                  Compile only; produce .o files but suppress
                    linking
...etc.
```

◆ ◆ ◆ 第 2 章

2 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，则代表**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，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%
```

第 24 页中的 “2.1.4 向程序传递文件名”中对这些例程进行了详细的介绍。有关详细信息，请参见 **getarg**(3F)、**getcwd**(3F) 和 **getenv**(3F) 的相应手册页条目；这些内容以及其他有用的库例程在《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；运行时系统会根据几个惯例提供文件名。

2.1.2.1 打开作为临时文件

在 **OPEN** 语句中指定 **STATUS='SCRATCH'** 会打开一个名称形式为 **tmp.F***AAAxnnnnn* 的文件，其中 *nnnnn* 用当前进程 ID 替换，*AAA* 是一个包含三个字符的字符串，*x* 是一个字母；*AAA* 和 *x* 可确保文件名唯一。该文件在程序终止或执行 **CLOSE** 语句时被删除。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**和**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，标准输入是从工作站键盘接收输入；标准输出和标准错误是在工作站屏幕上显示输出。

在其他所有情况下，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 **fort.***n* 形式的名称打开，其中 *n* 为逻辑单元号。

2.1.3 不使用 OPEN 语句打开文件

在假定使用缺省惯例的情况下，并非必须使用 **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**，则会引用文件 **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工具可将 Fortran 程序中的逻辑单元号与物理文件相关联。

但是，有几种令人满意的方式可将文件名传递给 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字符变量，该字符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种方式是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）和写入标准输出（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**），读取或写入其他任何命名文件。

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 读取	**myprog < mydata**	**myprog < mydata**
标准输出－写入（覆写）myoutput	**myprog > myoutput**	**myprog > myoutput**
标准输出－写入/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，请参见 **csh**、**ksh** 和 **sh** 手册页。

2.2 直接 I/O

直接或随机 I/O 允许通过记录号直接访问文件。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的 **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)
```

写入直接访问单元 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。

使用 **FORM='BINARY'** 打开文件与使用 **FORM='UNFORMATTED'** 具有大致相同的效果，所不同的是文件中没有嵌入记录长度。如果没有此数据，则无法知道一条记录的开始或结束位置。因此，无法对 **FORM='BINARY'** 文件执行 **BACKSPACE** 操作，这是因为不知道要退格到什么位置。对 **'BINARY'** 文件执行 **READ** 操作时，将按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：以二进制的形式将数据写入文件，并按输出列表中指定的数量传输字节。
- **READ** 语句：将数据读取到输入列表中的变量，并传输该列表所要求数量的字节。因为文件中没有记录标记，所以不进行“记录结束”错误检测。检测到的唯一错误是“文件结束”或异常系统错误。
- **INQUIRE** 语句：在使用 **FORM="BINARY"** 打开的文件中，**INQUIRE** 返回：
  FORM="BINARY" ACCESS="SEQUENTIAL" DIRECT="NO" FORMATTED="NO" UNFORMATTED="YES"。
  RECL= 和 **NEXTREC=** 没有定义。
- **BACKSPACE** 语句：不允许使用－返回一个错误。
- **ENDFILE** 语句：在当前位置照常截断文件。
- **REWIND** 语句：将文件照常重新定位到数据的开头。

2.4 流 I/O

f95 中实现了 Fortran 2003 标准“流”I/O 的新方案。流 I/O 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'** 说明符来声明流 I/O 文件。字节地址文件定位要求 **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* 和 **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 program reads files created by C `fwrite()`

```
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 program writes to a file

```
#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、数组元素或结构化记录的字段。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** 标志编译的 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 大端字节序和小端字节序平台之间的二进制 I/O

在 SPARC 和 x86 平台间移动时，新的编译器标志 **-xfilebyteorder** 支持二进制 I/O 文件。标志标识未格式化 I/O 文件的字节顺序和字节对齐。

例如，

```
-xfilebyteorder=little4:%all,big16:20
```

将指定所有文件（以 "**SCRATCH**" 打开的文件除外）包含 4 字节边界对齐（例如 32 位 x86）的“小端字节序”数据，Fortran 单元 20 除外，该单元是 64 位“大端字节序”文件（例如 64位 SPARC V9）。

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

2.7 传统 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 操作。

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 操作，如下所示：

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

- **NAMELIST** 输入功能：
  - **f95** 允许输入时在组名前使用 **$** 或 **&**。Fortran 95 标准只接受 **&**，并且这是 **NAMELIST** 写入语句的输出内容。
  - **f95** 接受 **$** 作为输入组的终止符号，除非组中的最后一个数据项为 **CHARACTER**（在这种情况下，**$** 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** 和 **f77** 间的 Fortran 95 I/O 扩展及兼容性方面的其他信息，请参见《Fortran 用户指南》。

◆ ◆ ◆ 第 3 章

3 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，许多论述如何使用 **make** 和 SCCS 的优秀商品书籍已经出版面市，其中包括 Andrew Oram 和 Steve Talbott 合著的《Managing Projects with make》，还有 Don Bolinger 和 Tan Bronson 合著的《Applying RCS and SCCS》。这两本书均由 O'Reilly & Associates 出版。

3.1 使用 make 实用程序简化程序构建

make 实用程序使用智能化方法执行程序编译和链接任务。通常，大型应用程序由一组源文件和 **INCLUDE** 文件组成，需要与许多库进行链接。修改任何一个或多个源文件，都需要对那一部分程序进行重新编译，并且要重新链接。指定组成应用程序的各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。指令文件中有了这些说明，**make** 便会确保只重新编译那些需要重新编译的文件，并确保重新链接时使用生成可执行文件所需要的那些选项和库。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，请参见 **make**(1S)。

3.1.1 Makefile

名为 **makefile** 的文件以结构化方式告知 **make**，哪些源文件和目标文件依赖其他文件。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 的第一行表明 **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 中的每一条目都是一项规则，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**－每一条目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**－每一条目随后还有一行或多行，这些行指定将生成本条目相应的目标文件的 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 实用程序在当前目录中查找名为 **makefile** 或 **Makefile** 的文件并从该文件中获取指令。

make 实用程序：

- 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的生成时间更早，则使用 **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 实用程序的**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**。请注意，也可以在命令中使用 **"FFLAGS="**，将宏重置为空字符串使其不再有效。

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** 标志以及要编译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，请参见 **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** 将从 **cpp**(1) 以及其他编译处理器中读取任何“隐藏”文件（例如 **#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** (Source Code Control System)。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
- 防止源文件被其他开发人员同时更改
- 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

有些开发人员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**。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**文件以便对其进行编辑；**签入**已编辑完的文件。

使用 **sccs edit** 命令签出文件。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。

在您完成编辑后，使用 **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

4 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**通常是先前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，请参见 **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**之前**即被绑定到执行文件中。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绑定到可执行程序中。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他选项，可以获得库用法和库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（以及它要求的其他选项）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
```

某些链接程序选项具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。它们包括 **-B***x*、**-d***x*、**-G**、**-h***name*、**-R***path* 和 **-ztext**。有关详细信息，请参见 **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源自的库一起被列出。

示例：使用 **–m** 生成加载映射：

```
demo% setenv LD_OPTIONS '-m'
demo% f95 any.f
any.f:
 MAIN:
           LINK EDITOR MEMORY MAP

output    input    virtual
section   section  address         size

.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 列出其他信息

其他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           display input argument processing
debug: bindings       display symbol binding;
debug: detail         provide more information
debug: entry          display entrance criteria descriptors
      ...
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，请参见《链接程序和库指南》。

4.2.3 一致编译和链接

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至关重要。在使用选项编译程序的任何部分时，必须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 **-fast** 编译 **sbr.f**，编译 C 例程，然后分步进行链接：

```
demo% f95 -c -fast sbr.f
demo% cc -c -fast simm.c
demo% f95 -fast sbr.o simm.o          link step; passes  -fast  to the linker
```

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* 和 **-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（在其他路径中）中搜索 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，如果使用多个 **-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，当搜索 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，请参见 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 – 强烈建议不要对生产软件使用 **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文件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* 编译器选项对链接程序在解析外部引用时要搜索的其他库命名。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查找其他的 **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查找位于标准路径以外的库。

假如 **libmylib.a** 位于 **/home/proj/libs** 目录中，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查找：

```
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，并且只搜索在搜索时未定义的符号。如果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。**-l***library* 选项放置在以下位置：

- 将 **-l***library* 选项放置在任一 **.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 **LD_LIBRARY_PATH** 的值
- 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** 会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: fatal: libmylib.so: can't open file:
```

此消息表明这些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，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) 实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 **COMMON** 子程序中已初始化的 **BLOCKDATA** 块。这些提取出来的元素（例程）会被永久绑定到链接程序生成的 **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**方式绑定 **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，如果需要更新绑定到 **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** 文件以便利用已更新的库。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（从左至右）对其进行处理。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他例程中调用。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
total 14          2 dropx.f      2 evalx.f      2 markx.f
   2 delte.f      2 etc.f        2 linkz.f      2 point.f
```

更低级的“帮助程序”例程被集中到文件 **etc.f** 中。其他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 **.o** 文件：

```
demo% f95 -c *.f
demo% ls
total 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，可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** 置于 **/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**之后**绑定到可执行文件。

动态库的另一功能是模块可供系统中其他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，动态库也是一个**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**将目标模块绑定到可执行文件；这种绑定被推迟到了运行时。
- 共享库模块在第一个运行程序引用它时绑定到系统内存中。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

动态库引入了其他一些权衡考虑因素：

- **a.out** 文件更小

将库例程绑定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种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**位置无关代码** (position-independent code, PIC) 绑定到程序中的任何地址，而无需链接编辑器进行重新定位。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** 编译器选项将组件例程编译成与位置无关。

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，而且可生成与位置无关的、大小不同的模型代码。（**-xcode=pic13** 等效于传统的 **-pic** 标志，**-xcode=pic32** 等效于 **-PIC**。）

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

4.5.3 绑定选项

可以在编译时指定动态或静态库绑定。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绑定设置首选项。**-Bstatic** 将绑定只限制于静态库。

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 位环境中的绑定

某些静态系统库（如 **libm.a** 和 **libc.a**）不能在 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** 获取全部所需库的列表。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 -m64 -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** 和其他链接程序选项编译：

```
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–bit MSB executable SPARC Version 1, dynamically linked, not stripped
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）绑定到可执行文件中。

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中到同一库中并在其他所有库之前引用该库，可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他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 Sun Studio SDN 门户网站：

`http://developers.sun.com/sunstudio/documentation/ss12/`

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

5 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，跨各个子程序。

–Xlist 报告子程序参数、公共块、参数在对齐、数值与类型一致性方面的错误，以及其他各种错误。

–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 (global program checking, 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
- 揭示其他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值与类型间的冲突
  - 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
- 不会执行的语句
- 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** 中生成输出列表
- 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* 生成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。不使用其他 **-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，同时扩展名替换为 **.lst**
- 编有行号的源代码列表
- 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                    Mon Mar 18 18:08:27 2002    page 1
```

```
FILE  "Repeat.f"
program  repeat
     4           CALL newf ( pn1 )
                           ^
**** ERR  #418:  argument "pn1" is real, but dummy argument is integer
                 See: "Repeat.f" line #14
     5           PRINT *, pn1
                          ^
**** ERR  #570:  variable "pn1" referenced as real but set as integer in
                 line #4
subroutine  newf
    19           PRINT *, prnok  ( ix )
                                   ^
**** ERR  #418:  argument "ix" is integer, but dummy argument is real
                 See: "Repeat.f" line #22
function  prnok
    23           prnok = INT ( x ) + .05
                 ^
**** WAR  #1024:  suspicious assignment a value of type "real*4" to a
                 variable of type "integer*4"
subroutine  unreach_sub
    26         SUBROUTINE unreach_sub()
                          ^
**** WAR  #338:  subroutine "unreach_sub" never called from program
subroutine  subr2
    31           CALL subr1(x+x)
                 ^
**** WAR  #348:  recursive call for "subr1". See dynamic calls:
                 "Repeat.f" line #10
                 "Repeat.f" line #3

Cross Reference                Mon Mar 18 18:08:27 2002    page 2

        C R O S S   R E F E R E N C E   T A B L E

  Source file:   Repeat.f


Legend:
D        Definition/Declaration
U        Simple use
M        Modified occurrence
A        Actual argument
C        Subroutine/Function call
I        Initialization: DATA or extended declaration
E        Occurrence in EQUIVALENCE
N        Occurrence in NAMELIST
```

```
L        Use Module


Cross Reference                 Mon Mar 18 15:40:57 2002     page 3

P R O G R A M   F O R M

 Program
 -------

repeat          <repeat>        D      1:D

Cross Reference                 Mon Mar 18 15:40:57 2002    page 4


 Functions and Subroutines
 -------------------------

INT      intrinsic
                <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

Cross Reference                 Mon Mar 18 15:40:57 2002    page 5


 Variables and Arrays
 --------------------

ix       int*4  dummy
                 <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 dummy
                <subr1>        DU      8:D      9:U     10:U
                <subr2>        DU     30:D     31:U     31:U
                <prnok>        DA     22:D     23:A

------------------------------------------------------------------
STATISTIC                    Mon Mar 18 15:40:57 2002    page 6


Date:     Mon Mar 18 15:40:57 2002
Options:  -XlistX
Files:         2 (Sources: 1; libraries: 1)
Lines:        33 (Sources: 33; Library subprograms:1)
Routines:      6 (MAIN: 1; Subroutines: 4; Functions: 1)
Messages:      6 (Errors: 3; Warnings: 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。

以下部分介绍用于生成列表、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。
- 其他子选项进一步指定其他细节。

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 **Basic Xlist** 子选项

生成的报告	选项
错误、列表、交叉引用	**–Xlist**
仅错误	**–XlistE**
仅错误以及源码列表	**–XlistL**
仅错误以及交叉引用表	**–XlistX**
仅错误以及调用图	**–Xlistc**

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
–Xlistc	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。
–XlistE	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。
–Xlisterr[*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。要禁止其他特定的错误，可重复使用该选项。如果未指定 *nnn*，会禁止所有错误消息。
–Xlistf	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代码。
–Xlisth	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。

表 5–2 -Xlist 子选项的完整列表 （续）

选项	操作
–XlistI	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。
–XlistL	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
–Xlistl*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。缺省值为 66。 如果使 *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。
-XlistMP	(**SPARC**) 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》。
–Xlisto *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**
–Xlists	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。 缺省情况下，不显示 **#include** 或 **INCLUDE** 文件中出现的标识符。

表 5–2 **-Xlist** 子选项的完整列表 （续）

选项	操作
–Xlistv*n*	设置检查“严格”程度 *n* 可以是 **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－仅检查语法错误。 ■ **–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，包括参数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错误使用所造成的错误。 ■ **–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。
–Xlistw[*nnn***]**	设置输出行的宽度 可使用 **-Xlistw** 设置输出行的宽度。例如，**-Xlistw132** 将页宽度设置为 132 列。缺省值为 79。
–Xlistwar[*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。
–XlistX	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，但不产生任何源代码列表。

5.2 特殊编译器选项

有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 链接程序还具有其他调试辅助选项。请参见 **ld**(1)，或在 shell 提示符下运行命令 **ld –Dhelp** 来查看联机文档。

5.2.1 下标边界 (–C)

如果使用 **-C** 编译，编辑器在运行时会增加对每个数组下标上的跨界引用以及数组一致性的检查。此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 RUN-TIME SYSTEM  ******
Subscript out of range. Location: line 3 column 9 of 'range.f'
Subscript number 1 has value 11 in array 'A'
Abort
demo%
```

5.2.2 未声明的变量类型 (–u)

-u 选项检查任何未声明的变量。

-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，便会出现错误消息。

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 SOURCE LINES
f90comp: 0 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
ld: Solaris Link Editors: 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 程序提供了事件管理、过程控制和数据检查。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，然后继续执行，而无需重新编译其他例程
- 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；向下单步执行并跳出函数调用
- 在下一行或某一其他行运行、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，或上下移动调用栈
- 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，请参见 **dbx** 文档。

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

◆ ◆ ◆ 第 6 章

6 浮点运算

本章分析浮点运算并提出避免和检测数值计算错误的策略。

有关 SPARC 和 x86 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健、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原因非常困难。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一个关键问题。

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 运算是一种处理会导致如下问题的运算操作的相对较新的方法：无效操作数、被零除、上溢、下 溢或结果不精确。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 标准支持用户处理异常、舍入和精度。因此，此标准支持区间运算和异常诊断。IEEE 标准 754 可以使诸如 *exp* 和 *cos* 之类的基本函数标准化，以便创建高精确度的运算，并结合数值和符号代数计算。

与其他任何种类的浮点运算相比，IEEE 运算向用户提供了对计算的更强大控制。此标准简化了编写复杂可移植数值程序的任务。很多有关浮点算法的问题都涉及数的基本运算。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?
- 有些基本运算（比如乘法和加法）是否具有交换性?

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：程序立即中止或使用无用结果继续执行。

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。它还确保异常情况产生指定的结果，除非用户明确地指定其他选项。

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

并且，还标识出五种类型的浮点异常：

- **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来表示－例如，输入中的 2.0/3.0、log(1.1) 和 0.1

在《数值计算指南》中介绍了 IEEE 标准的实现。

6.2.1 –ftrap=*mode* 编译器选项

-ftrap=*mode* 选项可捕获浮点异常。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转储文件终止程序。有关该编译器选项的详细信息，请参见《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** 的缺省选项。）

注 – 必须使用 **-ftrap=** 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函数对浮点状态寄存器进行查询，以找出产生了哪些异常，并打印一条有关标准错误的消息来通知您哪些是已产生但尚未清除的异常。此消息通常如下所示；格式可能会随各编译器版本而变化：

```
Note: IEEE floating-point exception flags raised:
    Division by Zero;
IEEE floating-point exception traps enabled:
    inexact;  underflow;  overflow;  invalid operation;
See the Numerical Computation Guide, 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** 惯例。

可以使用 **ieee_flags()** 关闭任意或所有这些消息，方法是在调用 **ieee_retrospective** 之前清除异常状态标记。

6.2.3 处理异常

根据 IEEE 标准进行异常处理是 SPARC 和 x86 处理器上的缺省异常处理方法。但是，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，会发生两件事情：

- 系统返回缺省结果。例如，对于 0/0 (*invalid*)，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 (*invalid*)，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。

缺省情况下，**f95** 会自动捕获被零除、上溢和无效运算。而 **f77** 在缺省情况下**不**会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** 的缺省选项是 **-ftrap=common**。要按早期的 **f77** 缺省设置进行运算，请使用 **-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。

可以通过以下方式禁用然后关闭渐进下溢：使用 **-fns** 选项进行编译，或从程序内调用库例程 **nonstandard_arithmetic()**。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()** 子例程会替换名为 **gradual_underflow()** 的早期例程。

- **nonstandard_arithmetic()** 子例程会替换一个早期例程（名为 **abrupt_underflow()**）。

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

以下接口可帮助用户使用 IEEE 运算，并在手册页中进行说明。这些接口多数都在数学库 **libsunmath** 和几个 **.h** 文件中。

- **ieee_flags**(3m)－控制舍入方向和舍入精度；查询异常状态；清除异常状态
- **ieee_handler**(3m)－建立异常处理例程
- **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
- 本部分介绍的其他 **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。

    实用程序 **fpversion** 显示安装了哪种浮点硬件，并指示要指定的合适的 **-xtarget** 值。该实用程序可在所有 Sun SPARC 体系结构中运行。有关详细信息，请参见 **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** 是一个整数值函数。*flags* 中返回有用的信息，作为 1 位标志集合。有关完整信息，请参见 **ieee_flags**(3m) 手册页。

下表显示了所有可能的参数值。

表 6–1 ieee_flags (*action, mode, in, out*) 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* 的可能值的含义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* 参数的含义

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，请将输入参数 *in* 设置为 **overflow**。返回时，如果 out 等于 overflow，会出现 **overflow** 异常标志；否则不会出现该标志。

```
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。

使用清除 *action* 调用 **ieee_flags**（如下例所示）可以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 include 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（即 **DetExcFlg.F**）中，include 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，然后对其进行解码：

```
#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                ! Cause underflow

        flgs=ieee_flags('get','exception','',out)  ! Which are raised?

        inx   = and(rshift(flgs, fp_inexact)  , 1)  ! Decode
        div   = and(rshift(flgs, fp_division) , 1)   ! the value
        under = and(rshift(flgs, fp_underflow), 1)    ! returned
        over  = and(rshift(flgs, fp_overflow) , 1)     ! by
        inv   = and(rshift(flgs, fp_invalid)  , 1)      ! ieee_flags

        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)    ! Clear all
      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()**

两个 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()** 来建立一个可用作异常处理程序的用户函数?
- 如何编写可用作异常处理程序的函数?
- 如何定位异常－即异常出现在何处?

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 标准名称是 **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

要将函数作为异常处理程序建立，请将函数名称与要监视的异常的名称和要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	**character**	**get**、**set** 或 **clear**
exception	**character**	**invalid**、**division**、**overflow**、**underflow** 或 **inexact**
handler	函数名	用户处理函数的名称或 **SIGFPE_DEFAULT**、**SIGFPE_IGNORE** 或 **SIGFPE_ABORT**
返回值	**integer**	0 =*OK*

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** 是一个整数。
- **sip** 是具有结构 **siginfo** 的记录。
- 未使用 **uap**。

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
... actions you take ...
END
```

此示例需要修改才能运行在 64 位 SPARC 体系结构上，方法是使用 **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此示例中一样，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

下列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%
```

SIGFPE 可在浮点异常出现的任何时间产生。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 64 位 SPARC 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      Set breakpoint at address
(2) stopi at &MAIN+0x68
(dbx) run                   Run program
Running: a.out
(process id 18803)
stopped in MAIN at 0x11230
MAIN+0x68:      fdivs   %f3, %f2, %f2
(dbx) where             Shows the line number of the exception
=>[1] MAIN(), line 7 in "LocExcHan.F"
(dbx) list 7            Displays the source code line
    7         t = r/s
(dbx) cont              Continue after breakpoint, enter handler routine
Signal    8 code    3 at hex address    11230
abort: called
signal ABRT (Abort) in _kill at 0xef6e18a4
_kill+0x8:      bgeu    _kill+0x30
Current function is 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本内容。

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 **-ftrap=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 **-ftrap=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()**，您可以更有选择性地查看异常。

示例：**dbx** 的编译及使用：

```
demo% f95 -g myprogram.f
demo% dbx a.out
Reading symbolic information for a.out
...
(dbx) catch FPE
(dbx) run
Running: a.out
(process id 19739)
signal FPE (floating point divide by zero) in MAIN at line 212 in file "myprogram.f"
  212               Z = X/Y
(dbx)  print Y
y = 0.0
(dbx)
```

如果发现程序因上溢和其他异常而终止，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：在其他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
Reading symbolic information for a.out
...
(dbx) catch FPE
(dbx) run
Running: a.out
(process id 19793)
signal FPE (floating point overflow) in MAIN at line 55 in file "myprog.F"
   55               w = rmax * 200.                    ! Cause of the overflow
(dbx) cont                                         ! Continue execution to completion
```

```
execution completed, exit code is 0
(dbx)
```

为了具有选择性，此示例引入了 **#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结果或调试结果。

6.5 更深层次的数值风险

本部分解决一些运算操作无意间可能会产生无效、被零除、上溢、下溢或不精确异常的实际问题。

例如，在 IEEE 标准之前，如果在计算机中将两个非常小的数相乘，结果可能为零。多数大型机和小型机的行为亦是如此。使用 IEEE 运算，**渐进下溢**会扩大动态计算范围。

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*。将两个小数相乘：

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

较早的运算会得到 0.0，但如果使用 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（即一个**非规范化数**）从某种意义上说精确度比较低，但从另外一种意义上说精确度又比较高。其深层含意已超出了本次讨论的范围。如果有兴趣，可以参考《*Computer*》1980 年 1 月，第 13 卷，第 1 期，尤其是 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，程序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，请参见《数值计算指南》。

6.5.1 避免简单下溢

有些应用程序实际上执行了许多非常接近零的计算。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，并且可避免出现下溢情况：

```
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（存储零）像较早的系统一样进行处理。

6.5.2 以错误答案继续

您可能会对在答案明显错误的情况下为什么能继续进行计算感到奇怪。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较大的数相乘。

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 的特殊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

如果将两个非常小的数字相乘，结果将会出现下溢。

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

注意：目前，区间运算仅适用于 SPARC 平台。

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：$[a, b] = \{z \mid a \leq z \leq b\}$（由一对数字定义，$a \leq b$）。区间可以用于：

- 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

通过将区间作为一种内在数据类型引入 Fortran 95，开发人员立即可以使用 Fortran 95 的所有适用语法和语义。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、三角及其他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，请参见《Fortran 95 区间运算编程参考》。

◆ ◆ ◆ 第 7 章

7 移植

本章讨论从其他平台向 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 在最初开发 Fortran 时，回车控制就不受所用设备的功能限制。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')** 启用单倍行距或双倍行距、换页以及剥离第一列。这还可通过将 **FORM='PRINT'** 与 **f95 -f77** 兼容标志一起使用来编译程序的方式而获得。此编译器允许在使用 **-f77** 编译时，重新打开单元 6，以将形式参数更改为 **PRINT**。例如：

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

可以使用 **lp**(1) 打印以这种方式打开的文件。

7.2 使用文件

早期的 Fortran 系统不使用命名文件，但提供了一种命令行机制，使实际文件名可以与内部单元编号等同。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              运行重定向读取数据文件
  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）大型机开发的，如 CRAY 或 CDC，则在移植到 UltraSPARC 平台时可能要使用以下选项来编译这些代码，例如：

```
-fast -m64 -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** 常量也会被提升为 **REAL*8**。（针对目标平台相应地设置 **-xarch** 和 **-xchip**。）若要将缺省 **DOUBLE PRECISION** 数据也提升为 **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效于其他类型的变量，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时引起异常，并发送 **SIGFPE** 信号或在缺省情况下捕获异常（对于 **f95**，缺省选项为 **-ftrap=common**）。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 章一章对此进行了说明。
- 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

许多“旧式”Fortran 应用程序会将霍尔瑞斯 ASCII 数据存储到数值数据对象中。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，但这不是标准做法。下表指明了适合某种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	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，然后将其传递给其他例程。

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将参数中的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

一般情况下，通过消除所有非标准编码，可以更简单地将一个应用程序从一个系统和编译器移植到另一个系统和编译器。在一个系统中大获成功的优化或解决方法，在其他系统中可能只会给编译器造成模糊和混乱。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变量和 COMMON 变量初始化为零或者“非数字”(NaN) 值。但是，没有任何标准做法，而且程序不应对任何变量的初始值进行假设。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

当同一个存储地址被多个名称引用时，就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际参数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 程序利用这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。不严格遵循 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、使用指针或将仍在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** 关键字及其含义

-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	遵循 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。（这是缺省情况。）
craypointer	程序使用可指向任何地址的 Cray 指针。（这是缺省情况。）
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。（这是缺省情况。）

表 7–2 **-xalias** 关键字及其含义 （续）

-xalias= *keyword*	别名使用情况
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： ■ 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他元素的别名。 ■ 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。数组引用不引用其他变量。（这是缺省情况。）
actual	编译器将子程序的实际参数视为全局变量。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。（这是缺省情况。）

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，如果您的程序中不存在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用变量可以重叠。

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** 进行编译可能会获得更佳的性能：

```
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
```

编译器假设对 *sub* 的调用可以写入 *z*
用 `-xalias=overindex` 编译时，程序打印 `2` 而非 `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，以标记任何数组下标问题。

一般而言，**overindex** 标志只能与传统 Fortran 77 程序一起使用。**-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将**始终**产生不正确的结果：

本例中数组语法索引越界不会产生正确的结果！

```
   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     the reference to a is actually a reference into b
C     so this should really be  b(2:n) = b(1:n-1)
C
      a(m+2:k) = b(1:n-1)

C     or doing it in reverse
      a(k:m+2:-1) = b(n-1:1:-1)
```

从直观上，用户期望数组 *b* 现在与数组 *c* 相似，但结果是无法预料的

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 会获取 **i** 的地址，并将其保存起来。**use_loc** 将使用它。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时更加小心。

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。它等效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，其目的是使旧式的向量化编译器生成用于特定体系结构的最佳代码。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

有些体系结构中的固定长度矢量寄存器可让程序员手动将循环中的数组计算条状提取成各个段。

```
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；可按如下方式编写循环来大大降低模糊程度：

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

7.6.3.2 循环展开

在编译器可用之前手动展开循环是一项典型的源代码优化技巧，它会自动执行此重构。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

返回日期时间或经过的 CPU 时间的库函数会因系统的不同而不同。

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	功能	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%
```

下表中所列的这些例程提供了与 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 汇总：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 VMS 版本的 **idate**(3F) 存在 2000 年安全问题，因为它们返回包含两位数值的年份。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。应改用 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

此处给出了一些建议，适用于移植到 Fortran 95 的程序没有按预期运行的情形。

7.8.1 结果贴近，但不够贴近

尝试以下操作：

- 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。回顾有关浮点问题的第 6 章一章。

- 尝试用 **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 或某一其他系统，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 及其他可能的错误迹象。有关如何捕获各种异常的说明，请参见第 6 章一章或手册页 **ieee_handler**(3m)。在大多数机器上，这些异常只是中止运行。
- 相差 6×10^{29} 的两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
```

输出为：

```
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 6×10^{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字。

7.8.2 程序失败而不警告

如果程序失败而不发出警告，并且失败之间运行的时间长度不同，则：

- 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。（请参见第 86 页中的“7.6.2 别名使用和 **-xalias** 选项”）

◆ ◆ ◆ 第 8 章

8 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 软件提供了一对用于收集和分析程序性能数据的高级工具：

- 收集器按称为剖析的统计方式收集性能数据。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。
- 性能分析器用于显示收集器记录的数据，从而可以检查相应信息。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（可以使用该映射文件来改进函数在应用程序地址空间中的装入顺序）来协助优化应用程序性能。

这两个工具有助于回答以下各种问题：

- 程序消耗的可用资源有多少?
- 最消耗资源的是哪些函数或负载对象?
- 哪些源码行和反汇编指令耗用的资源最多?
- 程序在执行过程中如何出现这种问题?
- 函数或负载对象消耗的是哪些资源?

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。有些情况下（例如，搜索高度量值时），可以使用该窗口在调用树中进行导航。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。

收集器和分析器适用于所有软件开发人员，尽管性能优化并不是开发人员的主要职责。与常用剖析工具 **prof** 和 **gprof** 相比，它们提供了更加灵活、详细和准确的分析，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) 命令收集数据。
- 可以从 **dbx** 中使用 **collector** 子命令运行收集器。
- 命令行实用程序 **er_print**(1) 可输出 ASCII 形式的各种分析器显示信息。
- 命令行实用程序 **er_src**(1) 可显示标注了编译器注释的源代码与反汇编代码列表，但不包含性能数据。

有关详细信息，请参见 Sun Studio《Program Performance Analysis Tools》手册。

8.2 time 命令

收集有关程序性能和资源利用情况的基础数据的简单方法是使用 **time** (1) 命令或 **set time** 命令（在 **csh** 中）。

使用 **time** 命令运行程序会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：

user system wallclock resources memory I/O paging

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

- *user*－用户代码中约 6.5 秒
- *system*－系统代码中该任务约 17.1 秒
- *wallclock*－完成时间 1 分 16 秒
- *resources*－该程序占用了系统资源的 31%
- *memory*－共享程序内存为 11 KB，专用数据内存为 21 KB
- *I/O*－读取 354 次，写入 210 次
- *paging*－缺页 135 次，换出 0 次

8.2.1 对 time 输出的多处理器解释

在多处理器环境中以并行方式运行程序时，计时结果的解释方法不同。由于 **/bin/time** 会累加不同线程上的用户时间，因此只使用挂钟时间。

由于所显示的用户时间包括花费在所有处理器上的时间，因此该值可以相当大，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

tcov(1) 命令用于使用 **-xprofile=tcov** 选项编译的程序时，会生成源代码的逐句剖析信息，显示执行了哪些语句以及执行频率。它还会给出有关程序基本块结构的信息摘要。

增强的语句级覆盖范围通过 **-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 – 如果编译器内联了例程调用，则 **tcov** 生成的代码覆盖范围报告不可靠。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，因此 **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**，请使用 **-xprofile=tcov** 进行编译。运行程序时，覆盖范围数据存储在 *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**。）

可运行 **tcov -x** *dirname source_files* 创建与每个源文件合并在一起的覆盖范围分析。相应的报告写入当前目录中的 *file*.**tcov**。

运行一个简单示例：

```
demo% f95 -o onetwo -xprofile=tcov one.f two.f
demo% onetwo
       ... output from program
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
       .....etc
demo%
```

环境变量 **$SUN_PROFDATA** 和 **$SUN_PROFDATA_DIR** 可以用于指定中间数据收集文件的保存位置。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。设置了这些变量后，运行程序便会将执行数据写入 **$SUN_PROFDATA_DIR/$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** 会对其进行预置，从而在 **$SUN_PROFDATA_DIR/$SUN_PROFDATA/** 中查找文件，而不是在工作目录中查找。

随后的每次运行都会将更多的覆盖范围数据累加到 **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

9 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断优化或调节哪些内容的**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，编译命令行中不会显式声明任何选项，此时大多数选项均为**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，不遵循标准编码方法的程序，可能会与编译器的某些假设有冲突。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－用其他选项重新编译并

将执行结果和性能与基准相比较。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（使用 **-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 – 该选项定义为其他选项的特殊选择集，它会随版本和编译器的不同而变化。另外，**-fast** 选用的某些选项可能不是在所有平台上都可用。使用 **-dryrun** 标志进行编译可查看 **-fast** 的扩展。

-fast 可为某些基准测试应用程序提供高性能。但是，对于您的应用程序，选项的特定选择可能是合适的，也可能是不合适的。使用 **-fast** 是编译应用程序以获得最佳性能的良好起点。但是，仍然可能需要进行其他调整。如果用 **-fast** 编译时程序不能正常运行，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，使程序正常运行。

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。对于那些依赖浮点运算的特殊属性的程序，请避免用 **-fast** 进行编译。

由于 **-fast** 选择的某些选项具有链接含义，因此，如果在不同的步骤中进行编译和链接，还请务必用 **-fast** 进行链接。

–fast 会选用以下选项：

- **–dalign**
- **–depend**
- **–fns**
- **–fsimple=2**
- **-ftrap=common**
- **–fround=nearest**（仅限 Solaris）
- **–libmil**
- **–xtarget=native**
- **–O5**
- **–xlibmopt**（仅限 Solaris）
- **-pad=local**（仅限 SPARC）
- **-xvector=lib**（仅限 SPARC）
- **-nofstore**（仅限 x86）
- **-xregs=frameptr**（仅限 x86）

-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》）。另请注意，**-fast** 的确切展开形式可能会随各个编译器发行版本而发生改变。使用 **-dryrun** 进行编译会显示所有命令行标志的展开形式。

随 **-fast** 一起使用其他选项可进一步增加优化。例如：

f95 -fast -m64 ...

可对启用了 64 位的平台进行编译。

由于 **-fast** 会调用 **-dalign**、**-fns** 和 **-fsimple=2**，因此用 **-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。对于大多数程序来说，这些选择可能是不合适的。

9.1.1.2 -O*n*

除非显式指定 **-O** 选项（或使用类似 **-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他事情。（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，才应为其指定 **-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* 标志一起使用，才可设置**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** 及更高级别应用其优化策略时，如果结合使用 **-xprofile=use**，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

为 **-O3** 及更高的优化级别添加 **-depend** 可扩展编译器优化 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，甚至可能会降低其他循环的性能。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。

下例展示如何有可能用 **-unroll=4** 将简单循环展开成四级深度（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。当程序性能很重要时，目标硬件的正确说明会是非常重要的。在较新的 SPARC 处理器上运行时这一点尤其重要。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 和 UltraSPARC-IV 支持

-xtarget 和 **-xchip** 标志均接受 **ultra3** 和 **ultra3** 变体，并且为 UltraSPARC-III 和 UltraSPARC-IV 处理器生成优化代码。当在最新的 UltraSPARC 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（编译在非最新的 UltraSPARC 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
```

使用 **-m64** 编译可生成 64 位代码。

有关最新 UltraSPARC 处理器的 **-xtarget** 标志列表，请参见《Fortran 用户指南》。

在 UltraSPARC-III 和 UltraSPARC-IV 平台上进行性能剖析（使用 **-xprofile=collect:** 和 **-xprofile=use:**）效果尤为突出，这是因为它允许编译器识别经常执行的程序部分并进行最佳的本地化优化。

64 位 x86 平台支持

Sun Studio Fortran 编译器支持 Solaris 和 Linux x86 平台的 32 位和 64 位代码编译。

-xtarget=pentium3 标志将展开为：**-xarch=sse -xchip=pentium3 -xcache=16/32/4:256/32/4.**

对于 Pentium 4 系统，**-xtarget=pentium4** 将展开为：**-xarch=sse2 -xchip=pentium4 -xcache=8/64/4:256/128/8.**

新的 **-m64** 选项用来指定对 64 位 x64 指令集的编译。

新的 **-xtarget** 选项、**-xtarget=opteron** 为 32 位 AMD 编译指定了 **-xarch**、**-xchip** 和 **-xcache** 设置。

要生成 64 位代码，必须在命令行中 **-fast** 和 **-xtarget** 的后面指定 **-m64**。**-xtarget** 选项并不自动生成 64 位代码。**-fast** 选项也会产生 32 位代码，因为它也是一个定义 **-xtarget** 值的宏。所有当前 **-xtarget** 值（**-xtarget=native64** 和 **-xtarget=generic64** 除外）将产生 32 位代码，因此必须在 **-fast** 或 **-xtarget** 之后（右侧）指定 **-xarch=m64** 以编译 64 位代码，如下所示：

```
% f95 -fast -m64 或 % f95 -xtarget=opteron -m64
```

如果指定了 **-xarch=amd64**，编译器会立即预定义 **__amd64** 和 **__x86_64**。

在《Fortran 用户指南》中，可以找到有关 32 位和 64 位 x86 平台上的编译和性能的其他信息。

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。

在编译和链接大型多文件应用程序时，**-xipo** 特别有用。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，可以揭示用其他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 其他性能策略

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 库的基础上。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 User's Guide》。

9.1.4 消除性能抑制因素

使用 Sun Studio 性能分析器可确定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。

还有其他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：

```
Original Code:
    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)

Untangled Code:
    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
Source file: /home/user21/do.f
Object file: do.o
Load Object: do.o

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

   Function x inlined from source file do.f into the code for the following line
   Loop below pipelined with steady-state cycle count = 3 before unrolling
   Loop below unrolled 5 times
   Loop below has 2 loads, 1 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
     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.

   Loop below pipelined with steady-state cycle count = 3 before unrolling
   Loop below unrolled 5 times
   Loop below has 2 loads, 1 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
     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《性能分析器》手册。

9.2 进阶读物

以下参考书籍提供更多详细信息：

- 《High Performance Computing》，Kevin Dowd 和 Charles Severance 合著，O'Reilly & Associates，第二版，1998
- 《Techniques for Optimizing Applications: High Performance Computing》，Rajat Garg 和 Ilya Sharapov 合著，Sun Microsystems Press Blueprint，2001

◆ ◆ ◆ 第 10 章

10 并行化

本章概述多处理器并行化并介绍 Solaris SPARC 和 x86 多处理器平台上 Fortran 95 的功能。

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

10.1 基本概念

应用程序的并行化（或**多线程**）是指对程序进行编译，使其能够在多处理器系统上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。

只有应用程序首先成为多线程程序，才能在 Ultra™ 60、Sun Enterprise™ Server 6500 或 Sun Enterprise Server 10000 等多处理器系统上高效运行。也就是说，需要对可并行执行的任务进行标识并重新编程，使其计算分布在多个处理器或线程上。

可以通过对 **libthread** 基元进行适当调用，来手动实现应用程序的多线程化。但可能需要进行大量的分析和重新编程工作。（有关更多信息，请参见 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对象代码。作为支持并行机制的主要语言元素，Fortran 编译器将关注焦点放在 DO 循环上。并行化可将循环的计算工作分配到多个处理器上，**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 (**-autopar**) 去决定，也可以由程序员使用源代码指令 (**-explicitpar**) 来显式指定，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。

将程序中的所有循环都进行并行化处理并非都是有益的。只包含少量计算工作（与用于启动和同步并行任务的开销相比）的循环在并行化后，实际的运行速度可能更慢。另外，有些循环根本不能安全地进行并行化；由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，由于考虑到可能存在的隐藏副作用，这种分析肯定是保守的。（**-loopinfo** 选项可以显示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，以控制如何并行化（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** 也可以接受较早的 Sun 和 Cray 风格的指令，但现在这些指令已经过时，不使用了。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 的信息，请参见《OpenMP API 用户指南》，或访问 OpenMP 网站 `http://www.openmp.org`。

10.1.1 加速－期望目标

如果要使程序并行化以便该程序在四个处理器上运行，这样运行该程序花费的时间是否大致是在单个处理器上运行时所花费时间的四分之一（四倍**加速**）呢?

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

此处列出了应用程序并行化的常规步骤：

1. **优化**。使用适当的编译器选项集，以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**。在单个处理器和单个线程上运行并行化的程序，并检查结果，以找出可能在其中出现的不稳定性和编程错误。（将 **$PARALLEL** 或 **$OMP_NUM_THREADS** 设置为 1；请参见第 119 页中的 “10.1.5 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 数据依赖性问题

不是所有的循环都可并行化。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性，以并行方式执行循环的多个处理器便有可能相互干扰。

会引起数据依赖性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

您可以重新编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。在确定是否并行化循环时，编译器的自动并行化分析会考虑附加条件。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

在循环的某一次迭代中设置并在后续迭代中使用的变量会导致产生交叉迭代依赖性或**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)）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

约简操作可将数组中的元素约简为单个值。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，从而覆盖 SUM 中的值。为使之正常工作，每个处理器每次必须执行一次求和，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

如果向在循环中用下标（下标值未知）标出的数组存储数据，会导致循环依赖性。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
```

在示例中，ID 中重复的值会造成 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

Sun Studio 编译器本身支持将 OpenMP 并行化模型作为主并行化模型。有关 OpenMP 并行化的详细信息，参见《OpenMP API 用户指南》。Sun 和 Cray 风格的并行化涉及传统的应用程序，当前的 Sun Studio 编译器不再支持这些风格的并行化。

。

表 10–1 Fortran 95 并行化选项

选项	标志
自动（**单独**）	**-autopar**
自动和约简	**-autopar -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**
编译以实现 OpenMP 并行化	**-xopenmp**

选项注释：

- 大多数选项具有等效的同义字，例如 **-autopar** 和 **-xautopar**。可以使用其一。
- 不应将编译器 prof/gprof 文件配置选项 **-p**、**-xpg** 和 **-pg** 与任何并行化选项一起使用。这些文件配置选项的运行时支持并非线程安全。运行时可能产生无效结果或段故障。
- **-reduction** 需要使用 **-autopar**。
- **-autopar** 包括 **-depend** 和循环结构优化。
- **-noautopar**、**-noreduction** 是否定选项。
- 并行化选项的顺序可以任意，但必须均为小写形式。
- 约简操作不在显式并行循环中进行分析。
- **-xopenmp** 还会自动调用 **-stackvar**。
- 选项 **-loopinfo** 和 **-vpara** 必须与并行化选项之一结合使用。

10.1.5 线程数

PARALLEL（或 **OMP_NUM_THREADS**）环境变量用来控制程序可以使用的线程的最大数量。设置该环境变量可将程序能够使用的最大线程数告之运行时系统。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虚拟处理器的数量。

下例展示如何设置环境变量：

```
demo% setenv OMP_NUM_THREADS 4          C shell
```

或

```
demo$ OMP_NUM_THREADS=4                 Bourne/Korn shell
demo$ export OMP_NUM_THREADS
```

在本例中，将 **PARALLEL** 设置为 4，可以最多使用四个线程来执行程序。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他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) 显示系统上可用的处理器列表：

```
demo% psrinfo
0       on-line   since 03/18/2007 15:51:03
1       on-line   since 03/18/2007 15:51:03
2       on-line   since 03/18/2007 15:51:03
3       on-line   since 03/18/2007 15:51:03
```

10.1.6 栈、栈大小和并行化

执行程序可为执行该程序的初始线程维护一个主内存栈，还可为每个辅助线程维护不同的栈。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**，因为它可增强优化程序在循环中并行化子程序调用的功能。对于包含子程序调用的显式并行化循环，**-stackvar** 是**必需**的。（请参见《Fortran 用户指南》中对 **-stackvar** 的介绍。）

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 example
cputime       unlimited
filesize       unlimited
datasize       2097148 kbytes
stacksize       8192 kbytes              <- current main stack size
coredumpsize       0 kbytes
descriptors       64
memorysize       unlimited
demo% limit stacksize 65536        <- set main stack to 64Mb
demo% limit stacksize
stacksize       65536 kbytes
```

对于 Bourne 或 Korn shell，相应的命令为 **ulimit**：

```
demo$ ulimit -a          Korn Shell example
time(seconds)        unlimited
file(blocks)         unlimited
data(kbytes)         2097148
stack(kbytes)        8192
coredump(blocks)     0
```

```
nofiles(descriptors) 64
vmemory(kbytes)      unlimited
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（线程的局部变量）在线程栈中分配。缺省大小在 64 位 SPARC 和 64 位 x86 平台上为 8 兆字节，在其他平台上为 4 兆字节。此大小是通过 **STACKSIZE** 环境变量设置的：

```
demo% setenv STACKSIZE 8192     <- Set thread stack size to 8 Mb    C shell
                          -or-
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** 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：编译器总是选择对使用**静态**循环调度的循环进行并行化－即将循环中的工作拆分到多个等效的迭代块中。其他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**角度看，需要一些新定义：

- **数组**是至少以一维声明的变量。
- **标量**是非数组的变量。
- **纯标量**是没有别名的标量变量－在 **EQUIVALENCE** 或 **POINTER** 语句中不会被引用。

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** 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他迭代的**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**循环携带依赖性**。
- 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

编译器可以自动消除显示的引用以在循环中创建数据依赖性。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，如果编译器能确定这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              <--Parallelized
  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
```

在此例中，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性，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

在自动并行化过程中，如果以下条件下成立，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中，对循环嵌套中最外层循环（而不是最内层循环）进行并行化最有效。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

将一个数组转换为一个标量的计算称为**约简操作**。典型的约简操作有矢量元素的求和或求积。违反循环中计算标准的约简操作不能在各次迭代间以累积方式改变标量变量。

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)`
求积	`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))`

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

AND	`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

由于以下原因，浮点型数字的求和或求积约简操作可能不准确：

- 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型数的求和或求积。硬件浮点加法和乘法不是结合式的。可能会出现舍入、溢出或下溢误差，具体取决于操作数关联的方式。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 ) ! Get n random nos. between -1 and +1
      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。有关更多信息，请参见 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 编译器现在完全支持 OpenMP Fortran API 作为主并行化模型。有关其他信息，请参见《OpenMP API 用户指南》。

SPARC 平台上的 Sun Studio 编译器不再支持传统的 Sun 风格和 Cray 风格的并行化指令，x86 平台上的编译器不接受这些指令。

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 循环之前紧接着放置的指令标记这些循环将要进行并行化。用 **-xopenmp** 进行编译，可以识别 OpenMP Fortran 95 指令并生成并行化的 DO 循环代码。并行化指令是用来指示编译器并行化（或不并行化）指令后面的 **DO** 循环的注释行。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**勿**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

如果以下条件成立，则循环适用于显式并行化：

- 循环是 **DO** 循环，而不是 **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他迭代的数组变量值。
- 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他任何迭代的**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他任何迭代。

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他迭代所见。

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能保持未定义。

10.3.1.2 循环中的子程序调用

循环（或从已调用例程内调用的任何子程序）中的子程序调用可能会导致产生数据依赖性，这种数据依赖性很容易被忽略，而不通过调用链深入分析数据和控制流。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。有关如何增加栈大小的信息，请参见第 120 页中的 “10.1.6 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，如果您显式指导编译器对循环进行并行化，编译器就会执行。但也有例外情况－存在一些编译器不进行并行化的循环。

下面是可检测到的主要抑制因素，这些抑制因素可以防止对 **DO** 循环进行显式并行化：

- **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。

通过使用 **-vpara** 和 **-loopinfo** 进行编译，可以得到诊断消息，指出在显式并行化循环过程中编译器是否检测到问题。

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	已并行化	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
在循环中使用 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      !  Parallelized (outer loop)
        do 200 j = 1, 1000    !  Not parallelized, no warning
           ...
200   continue
900     continue
      ...
```

示例：子例程中已并行化的循环：

```
 program main
      ...
!$OMP PARALLEL DO
      do 100 i = 1, 200       <-parallelized
        ...
        call calc (a, x)
        ...
100     continue
      ...
subroutine calc ( b, y )
      ...
!$OMP PARALLEL DO
      do 1 m = 1, 1000        <-not parallelized
        ...
1      continue
      return
      end
```

在此例中，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      !  Not parallelized, error issued
        ...
        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", line 5: Warning: the loop may have parallelization inhibiting reference
"vpfn.f", line 5: PARALLELIZED, user pragma used
```

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      !  Parallelized
        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 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，该指令必须紧跟在 **DO** 循环之前。

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 和传统的 Sun 风格指令之间的一个主要不同点是，OpenMP 需要显式数据作用域以**专用**或**共享**方式使用，并还要提供自动作用域功能。

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

10.4 环境变量

并行化使用多个环境变量：**OMP_NUM_THREADS**、**SUNW_MP_WARN**、**SUNW_MP_THR_IDLE**、**SUNW_MP_PROCBIND**、**STACKSIZE** 和其他环境变量。《OpenMP API 用户指南》中对它们进行了介绍。

10.5 调试并行化的程序

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
 Note: Following IEEE floating-point traps enabled;
   see ieee_handler(3M):
 Inexact;  Underflow;  Overflow;  Division by Zero;  Invalid
   Operand;
 Sun's implementation of IEEE arithmetic is discussed in
  the Numerical Computation Guide.
Debugging Parallelized Programs
```

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来验证程序是否正确工作。
  - 将线程数设置为 1，然后打开并行化选项进行编译－将环境变量 **PARALLEL** 设置为 **1** 来运行程序。

    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。
  - 另外，通过用 **-C** 进行编译，检查数组引用是否越界。
  - 使用带 **–autopar** 选项的自动并行化问题可能会指示编译器正在并行化它本来不应并行化的应用程序。

  关闭 **-reduction**。

  如果正在使用 **–reduction** 选项，可能会进行求和约简，并得出稍微不同的答案。尝试不用该选项运行。
- 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。然后使用或不使用**–autopar** 编译一些文件。

  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。
- 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
```

10.6 进阶读物

下列书籍提供更多的信息：

- 《OpenMP API 用户指南》
- 由 Rajat Garg 和 Ilya Sharapov 合著的《Techniques for Optimizing Applications: High Performance Computing》， Sun Microsystems Press Blueprint 出版， 2001。
- 由 Kevin Dowd 和 Charles Severance 合著的《High Performance Computing》， O'Reilly and Associates 出版，第二版， 1998。
- 由 Rohit Chandra et al 编著的《Parallel Programming in OpenMP》， Morgan Kaufmann Publishers 出版， 2001。
- 由 Barry Wilkinson 编著的《Parallel Programming》， Prentice Hall 出版， 1999。

◆ ◆ ◆ 第 11 章

11 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 157 页中的 “11.9 Fortran 2003 与 C 的互操作性” 简要说明了 Fortran 2003 标准第 15 部分中提到的 C 绑定功能。（此标准可以从国际 Fortran 标准 Web 站点 `http://www.j3-fortran.org` 获得）。Fortran 95 编译器实现了标准中所述的这些功能。

如不特别注明，32 位 x86 处理器视为与 32 位 SPARC 处理器等同。对于 64 位 x86 处理器和 64 位 SPARC 处理器也是如此，只是 x86 系统未定义 REAL*16 和 COMPLEX*32 数据类型，这些数据类型只能用于 SPARC。

11.1 兼容性问题

大多数 C-Fortran 接口必须在以下这些方面全部保持一致：

- 函数和子例程的定义及调用
- 数据类型的兼容性
- 参数传递（按引用或按值）
- 参数的顺序
- 过程名（大写、小写或带有结尾下划线 (_)）
- 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 中有不同的含义。根据具体情况做出选择很重要：

- 在 C 中，所有的子程序都是函数；但 **void** 函数不会返回值。
- 在 Fortran 中，函数会传递一个返回值，但子例程一般不传递返回值。

当 Fortran 例程调用 C 函数时：

- 如果被调用的 C 函数返回一个值，则将其作为函数从 Fortran 中调用。
- 如果被调用的 C 函数不返回值，则将其作为子例程调用。

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**，则将其作为一个返回兼容数据类型的函数从 C 中调用。
- 如果被调用的 Fortran 子程序是一个**子例程**，则将其作为一个返回 **int**（与 Fortran **INTEGER*4** 兼容）或 **void** 值的函数从 C 中调用。如果 Fortran 子例程使用交替返回，则会返回一个值，这种情况下它是 **RETURN** 语句中的表达式的值。如果 **RETURN** 语句中没有出现表达式，但在 **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

表 11–2 总结了 Fortran 95（与 C 比较）数据类型的数据大小和缺省对齐。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。请注意以下事项：

- C 数据类型 **int**、**long int** 和 **long** 在 32 位环境下是等同的（4 字节）。但是，在 64 位环境下 **long** 和指针为 8 字节。这称为 LP64 数据模型。
- 在 64 位 SPARC 环境下，当用任意 **-m64** 选项进行编译时，**REAL*16** 和 **COMPLEX*32** 与 16 字节边界对齐。
- 标有 4/8 的对齐表示缺省情况下与 8 字节边界对齐，但在 COMMON 块中与 4 字节边界对齐。COMMON 中的最大缺省对齐为 4 字节。当用 **-m64** 选项进行编译时，4/8/16 表示与 16 字节边界对齐。
- **REAL(KIND=16)**、**REAL*16**、**COMPLEX(KIND=16)**、**COMPLEX*32** 只能用于 SPARC 平台。
- 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。
- 可以在调用点使用 **%VAL(***arg***)**，按值将参数从 Fortran 95 例程传递到 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 传递到 Fortran 95。
- 数值序列类型的组件的对齐方式与通用块的对齐方式相同，也会受到 **-aligncommon** 选项的影响。**数值序列类型**是这样一种序列类型：其中所有组件的类型为缺省整数、缺省实数、双精度实数、缺省复数或缺省逻辑，而不是指针。

- 在大多数情况下，非数值序列类型的数据类型组件以自然对齐的方式对齐，但 QUAD 变量除外。对于四精度变量，32 位 SPARC 平台和 64 位 SPARC 平台之间的对齐方式不同。
- 在所有平台上，用 BIND(C) 属性定义的 VAX 结构和数据类型的组件始终与 C 结构具有相同的对齐方式。

表 11–1 数据大小与对齐一（以字节表示）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` *(SPARC)*	`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` *(SPARC)*	`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 在区分大小写方面采取截然相反的处理方法：

- 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。除了字符串常量以外，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都出现的子程序名末尾追加一个下划线 (_)。该惯例不同于具有相同的用户指定名称的 C 过程或外部变量。几乎所有 Fortran 库过程名都有两个前导下划线，以减少与用户指定的子例程名的冲突。

对于下划线问题，有三种常用解决方案：

- 在 C 函数中，通过在函数名末尾追加下划线来更改该名称。
- 使用 **BIND(C)** 属性声明来指明外部函数是 C 语言函数。
- 使用 **f95 -ext_names** 选项编译对无下划线的外部名称的引用。

只能使用上述解决方案中的一种。

本章的示例都可以使用 **BIND(C)** 属性声明来避免下划线。**BIND(C)** 声明可从 Fortran 调用的 C 外部函数，以及可从 C 中作为参数调用的 Fortran 例程。Fortran 编译器在处理外部名称时通常不追加下划线。**BIND(C)** 必须出现在每个包含这样的引用的子程序中。惯常用法是：

```
FUNCTION ABC
 EXTERNAL XYZ
 BIND(C) ABC, XYZ
```

在此处，用户不仅指定 **XYZ** 是外部 C 函数，而且还指定 Fortran 调用程序 **ABC** 应该可以从 C 函数调用。如果使用 **BIND(C)**，C 函数不需要在函数名末尾追加下划线。

11.1.5 按引用或值传递参数

通常，Fortran 例程按引用传递参数。在调用中，如果非标准函数 **%VAL()** 中包含一个参数，则调用例程会按值传递该参数。

Fortran 95 按值传递参数的标准方法是通过 VALUE 属性和 INTERFACE 块。请参见 第 149 页中的 “11.4 按值传递数据参数” 。

C 通常按值传递参数。如果在参数前加上表示 “和” 的符号 (**&**)，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 均以相同的顺序传递参数。但对于每个字符型参数，Fortran 例程都会传递一个附加参数，用以指定串长度。这些参数在 C 中是 **long int** 数量，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）
- 与每个字符参数对应的 **long int**（字符串长度的完整列表位于其他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`
`INTEGER B(3)`
`...`
`CALL SAM( S, B(2) )`
	`char s[7];`
	`int b[3];`
	`...`
	`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 数组总是从 0 开始，而 Fortran 数组在缺省情况下是从 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)** 等同于 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)** 就等同于 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。但对于多维数组，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*（即 **stdio**）的子例程。有许多 Fortran I/O 函数也使用标准 I/O，而后者又使用操作系统 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；请参见 **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

表 11–2 Fortran 与 C 之间的 I/O 比较 （续）

	Fortran 单元	标准 I/O 文件指针	文件描述符
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伪初始化例程 **f90_init**。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到程序的参数和环境。这些调用会提供软件挂钩，您可以在 C 中用软件挂钩提供自己的例程，以便在程序启动之前以任何定制方式初始化程序。

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，因此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 例程和 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及 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 或 CHARACTER 串），将 C 例程中的每个关联参数作为指针定义或传递：

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)
...
 	
---------- 	
	int i=100;
	float r;
	extern void fsim_(int *i, float *r);
	fsim_(&i, &r);
	...
	
	---------- 	

void csim_(int *i, float *r)
{
*r = *i;
}
	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到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，在寄存器中返回 **COMPLEX** 值。

11.3.3 字符串

由于没有标准接口，因此不推荐在 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。此额外参数给出串长度，它等同于按值传递的 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 调用及其等同的 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	C 调用 Fortran

`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];`
`}`
	`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	C 调用 Fortran

`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.;`
`  ...`
`}`
	`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	C 调用 Fortran

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);
}
	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
	  ...

请注意，在所有平台上 Fortran 77 (VAX) 结构与 C 结构的对齐方式始终相同。但是，平台之间对齐方式会有所变化。

表 11–9 传递 Fortran 95 派生类型

Fortran 95 调用 C	C 调用 Fortran 95

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);
}
	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
	  ...

请注意，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。

在所有平台上，数值序列类型的组件缺省情况下与字（4 字节）边界对齐。这与 x86 平台上 C 结构的对齐方式相匹配，但是不同于 SPARC 平台上 C 结构的对齐方式。使用 **-aligncommon** 选项可更改数值序列类型的对齐方式，以便与 C 结构相匹配。使用 **-aligncommon=8** 匹配 32 位 SPARC C 结构，使用 **-aligncommon=16** 匹配 64 位 SPARC。

未使用 **SEQUENCE** 显式声明的派生类型与 SPARC 平台上的 C 结构对齐方式相同，但与 x86 平台上的对齐方式不同。这种对齐方式不随编译器选项而改变。

11.3.7 指针

由于 Fortran 例程按引用传递参数，因此可将 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) 指针

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) !! pointer selects every other element of A
call S(P)
...
```

C **例程**：

```
void s_(int p[])
{
   /* change middle element */
   p[2] = 444;
}
```

请注意，由于 C 例程 S 不是 Fortran 95 例程，因此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。

另请注意，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	C 调用 Fortran 95
PROGRAM callc **INTERFACE** **INTEGER FUNCTION crtn(I)** **BIND(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);** **}** **----------** *Results:* 20 input 21 returning 20 21	**#include <stdlib.h>** **int main(int ac, char *av[]) {** **to_fortran_(12);** **}** **----------** **SUBROUTINE to_fortran(i)** **INTEGER, VALUE :: i** **PRINT *, i** **END**

请注意，如果要以不同的数据类型作为实际参数来调用 C 例程，应该在接口块中包含 **!$PRAGMA IGNORE_TKR I**，以防止编译器在实际参数和伪参数之间要求类型、类别和等级匹配。

对于传统 Fortran 77，按值调用仅对简单数据可用，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。

要将值从 Fortran 77 例程传递到 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.          x and y from Fortran
1.000000, 0.000000  x and y from C
2.000000, 1.000000  new x and y from C
1.00000 1.00000     new x and y from Fortran
```

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 函数是等同的（请参见表 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	C 调用 Fortran

`real ADD1, R, S`
`external ADD1`
`R = 8.0`
`S = ADD1( R )`
`...`
 	
`----------`
 	
`float add1_( pf )`
`float *pf;`
`{`
`  float f ;`
`  f = *pf;`
`  f++;`
`  return ( f );`
`}`
	`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 32 位和 64 位实现之间有所不同。

11.5.2.1 32 位平台

在 32 位平台上，返回 COMPLEX 或 DOUBLE COMPLEX 的 Fortran 函数等同于具有一个指向内存中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（32 位 SPARC）

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 64 位 SPARC 平台

在 64 位 SPARC 环境下，在浮点寄存器中返回 **COMPLEX** 值：在 **%f0** 和 **%f1** 中返回 **COMPLEX** 和 **DOUBLE COMPLEX**，在 **%f0**、**%f1**、**%f2** 和 **%f3** 中返回 **COMPLEX*32**。对于 64 位 SPARC，返回结构（其字段均为浮点型）的 C 函数将在浮点寄存器中返回该结构，但条件是最多需要 4 个这样的寄存器进行此操作。在 64 位 SPARC 平台上，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)*	**struct {float r,i;} cf_ (***a1, a2, ..., an)*

表 11–14　返回 COMPLEX 数据的函数（64 位 SPARC）

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 函数等同于具有两个附加第一参数（数据地址和串长度）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)** **{ /* return n copies of arg */** **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);** **/* make n copies of ch in sbf** ***/** **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。请注意，在从 C 中调用的 Fortran 例程中，需要显式添加一个末尾空字符。缺省情况下，Fortran 字符串不以空字符终结。

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;`
	`...`
	`}`

请注意，C 例程建立的外部名称必须以下划线结束，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。

缺省情况下，**f95** 会将通用块中的数据与至多 4 字节边界进行对齐。要获得通用块中所有数据元素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

不推荐混合使用 Fortran I/O 和 C I/O（同时从 C 和 Fortran 例程发出 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，所以可以共享它们。

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，以便将单元 0、5 和 6 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。

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同的机制，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	运行示例

`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`
	`demo% cc -c tst.c`
	`demo% f95 -o alt alt.f tst.o`
	`alt.f:`
	`altret:`
	`demo% alt`
	`1 2`
	
	C 例程接受从 *Fortran* 例程返回的值 *2*，因为它执行了 *RETURN 2* 语句。

11.9 Fortran 2003 与 C 的互操作性

Fortran 2003 标准草案（可从 http://www.j3-fortran.org 获得）提供了一种从 Fortran 95 程序中引用 C 编程语言定义的过程和全局变量的方法。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 2003 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。可从 Fortran 2003 标准获取详细信息。

索引

修

选

循

疑

移

已

异

易

溢

印