数值计算指南

Sun™ Studio 11

Sun Microsystems, Inc.
www.sun.com

文件号码 819-4817-10
2005 年 11 月，修订版 A

请将有关本文档的意见和建议提交至：http://www.sun.com/hwdocs/feedback

目录

图

表

阅读本书之前

本手册介绍运行 Solaris™ 操作系统 (Solaris OS) 的基于 SPARC® 和 x86 系统上的软件和硬件所支持的浮点环境。本手册介绍了 SPARC 和 Intel 体系结构的一些基本内容，但它主要是随 Sun™ 语言产品提供的参考手册。

本手册中介绍了一些用于二进制浮点算法的 IEEE 标准的内容。要了解 IEEE 算法，请参见 18 页的 “标准”。有关 IEEE 算法的简要参考书目，请参见附录 F。

本书面向的读者

本手册适用于开发、维护、移植数学的和科学的应用程序或基准的人员。在使用本手册之前，应该熟悉所使用的编程语言 （Fortran、 C 等）、 `dbx`（源码级调试器）以及操作系统命令和概念。

本书的结构

第 1 章介绍浮点环境。

第 2 章介绍 IEEE 算法模型、 IEEE 格式和下溢。

第 3 章 介绍了 Sun™ Studio 编译器所提供的数学库。

第 4 章介绍异常并说明如何检测、查找和处理它们。

附录 A 包含示例程序。

附录 B 介绍基于 SPARC 工作站的浮点硬件选项。

附录 C 列出与 Intel 系统中所使用的浮点单元有关的 x86 和 SPARC 兼容性问题。

附录 D 是 David Goldberg 编写的浮点算法教程的编辑重印。

附录 E 介绍标准遵循性。

附录 F 包含一个参考资料和相关文档的列表。

术语表 包含术语的定义。

本手册中的示例是使用 C 和 Fortran 编写的，但这些概念适用于基于 SPARC 或 x86 的系统上的编译器。

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	保留未译的新词或术语以及要强调的词。要使用实名或值替换的命令行变量。	这些称为 *class* 选项。 要删除文件，请键入 **rm** *filename*。
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。

* 浏览器的设置可能会与这些设置不同。

表 P-2 代码约定

代码符号	含义	表示法	代码示例
[ ]	方括号中包含可选参数。	O[*n*]	-O4, -O
{ }	花括号中包含所需选项的选项集合。	d{y|n}	-dy
|	分隔变量的“|”或“-”符号，只能选择其一。	B{dynamic|static}	-Bstatic
:	与逗号一样，分号有时可用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号表示一系列的省略。	-xinline=*f1*[,…*fn*]	-xinline=alpha,dos

Shell 提示符

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

支持的平台

此 Sun Studio 发行版本支持使用 SPARC® 和 x86 系列处理器体系结构（UltraSPARC®、SPARC64、 AMD64、 Pentium 和 Xeon EM64T）的系统。通过访问 http://www.sun.com/bigadmin/hcl 中的硬件兼容性列表，可以了解您在使用的 Solaris 操作系统版本的支持系统。这些文档列出了实现各个平台类型的所有差别。

在本文档中，这些与 x86 有关的术语具有以下含义：

- “x86”是指较大的 64 位和 32 位 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。

有关所支持的系统，请参见硬件兼容性列表。

访问 Sun Studio 软件和手册页

Sun Studio 软件及其手册页未安装到 /usr/bin/ 和 /usr/share/man 标准目录中。要访问软件，必须正确设置 PATH 环境变量 （请参见 第 xvi 页的 “访问软件”）。要访问手册页，必须正确设置 MANPATH 环境变量 （请参见第 xvii 页的 “访问手册页”）。

有关 PATH 变量的详细信息，请参见 csh(1)、sh(1)、ksh(1) 和 bash(1)) 手册页。有关 MANPATH 变量的详细信息，请参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问此发行版本的详细信息，请参见安装指南或询问系统管理员。

注 – 本节中的信息假设 Sun Studio 软件安装在 Solaris 平台上的 /opt 目录中和 Linux 平台上的 /opt/sun 目录中。如果未将软件安装在默认目录中，请询问系统管理员以获取系统中的相应路径。

访问软件

使用以下步骤决定是否需要更改 PATH 变量以访问该软件。

决定是否需要设置 PATH 环境变量

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

```
% echo $PATH
```

2. **在 Solaris 平台上，查看输出中是否包含有 /opt/SUNWspro/bin 的路径字符串。在 Linux 平台上，查看输出中是否包含有 /opt/sun/sunstudio11/bin 的路径字符串。**

   如果找到该路径，则说明已设置了访问该软件的 PATH 变量。如果没有找到该路径，则按照下一步中的说明设置 PATH 环境变量。

设置 PATH 环境变量以实现对软件的访问

- **在 Solaris 平台上，将以下路径添加到 PATH 环境变量中。如果以前安装了 Forte Developer 软件、Sun ONE Studio 软件，或其他发行版本的 Sun Studio 软件，则将以下路径添加到这些安装路径之前。**

  /opt/SUNWspro/bin

- **在 Linux 平台上，将以下路径添加到 PATH 环境变量中。**

  /opt/sun/sunstudio11/bin

访问手册页

使用以下步骤决定是否需要更改 MANPATH 变量以访问手册页。

决定是否需要设置 MANPATH 环境变量

1. **通过在命令提示符后键入以下内容以请求 dbx 手册页。**

```
% man dbx
```

2. **请查看输出 （如果有）。**

   如果找不到 dbx(1) 手册页或者显示的手册页不是软件当前版本的手册页，请按照下一步的说明来设置 MANPATH 环境变量。

设置 MANPATH 环境变量以实现对手册页的访问

- **在 Solaris 平台上，将以下路径添加到 MANPATH 环境变量中。**

  /opt/SUNWspro/man

- **在 Linux 平台上，将以下路径添加到 MANPATH 环境变量中。**

  /opt/sun/sunstudio11/man

访问集成开发环境

Sun Studio 集成开发环境 (integrated development environment, IDE) 提供了创建、编辑、生成、调试 C、 C++ 或 Fortran 应用程序并分析其性能的模块。

启动 IDE 的命令是 sunstudio。有关该命令的详细信息，请参见 sunstudio(1) 手册页。

IDE 是否可以正确操作取决于 IDE 能否找到核心平台。 sunstudio 命令会查找两个位置的核心平台：

- 该命令首先查找 Solaris 平台上的默认安装目录 /opt/netbeans/3.5V11 和 Linux 平台上的默认安装目录 /opt/sun/netbeans/3.5V11。
- 如果该命令在默认目录中找不到核心平台，则它会假设包含 IDE 的目录和包含核心平台的目录均安装在同一位置上。例如，在 Solaris 平台上，如果包含 IDE 的目录的路径是 /foo/SUNWspro，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。在 Linux 平台上，如果包含 IDE 的目录的路径是 /foo/sunstudio11，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。

如果核心平台未安装在 sunstudio 命令查找它的任一位置上，则客户端系统上的每个用户必须将环境变量 SPRO_NETBEANS_HOME 设置为安装核心平台的位置 (/*installation_directory*/netbeans/3.5V11)。

在 Solaris 平台上，IDE 的每个用户还必须将 /*installation_directory*/SUNWspro/bin 添加到其他任何 Forte Developer 软件、 Sun ONE Studio 软件或 Sun Studio 软件发行版本路径前面的 $PATH 中。在 Linux 平台上， IDE 的每个用户还必须将 /*installation_directory*/sunstudio11/bin 添加到其他任何发行版本的 Sun Studio 软件路径前面的 $PATH 中。

路径 /*installation_directory*/netbeans/3.5V11/bin 不能添加到用户的 $PATH 中。

访问 Sun Studio 文档

您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获取文档，位置为 Solaris 平台上的 file:/opt/SUNWspro/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。

  如果未将软件安装在 Solaris 平台上的 /opt 目录中或 Linux 平台上的 /opt/sun 目录中，请询问系统管理员以获取系统中的相应路径。
- 大多数的手册都可以从 docs.sun.com℠ Web 站点获取。以下书目只能从 Solaris 平台上安装的软件中找到：
  - 《标准 C++ 库类参考》
  - 《标准 C++ 库用户指南》
  - 《Tools.h++ 类库参考》
  - 《Tools.h++ 用户指南》
- 适用于 Solaris 平台和 Linux 平台的发行说明可以通过 docs.sun.com Web 站点获取。
- 在 IDE 中通过“帮助”菜单以及许多窗口和对话框上的“帮助”按钮，可以访问 IDE 所有组件的联机帮助。

您可以通过 Internet 访问 docs.sun.com Web 站点 (http://docs.sun.com) 以阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到手册，请参见与软件一起安装在本地系统或网络中的文档索引。

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。

使用易读格式的文档

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。您还可以按照下表所述，找到文档的易读版本。如果未将软件安装在 /opt 目录中，请询问系统管理员以获取系统中的相应路径。

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 平台上所安装软件中的文档索引 file:/opt/SUNWspro/docs/zh/index.html
自述文件	HTML，位于 Sun Developer Network 门户网站 http://developers.sun.com/prodtech/cc/documentation/
手册页	HTML，位于安装的软件上的文档索引，位置为 Solaris 平台上的 file:/opt/SUNWspro/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。
联机帮助	HTML，可通过 IDE 中的 “帮助”菜单和 “帮助”按钮访问
发行说明	HTML，位于 http://docs.sun.com

访问相关的 Solaris 文档

下表描述了可从 docs.sun.com W eb 站点上获取的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参见手册页部分的标题。	提供有关 Solaris 操作环境的信息。
Solaris 软件开发者集合	《链接程序和库指南》	介绍了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 软件开发者集合	《多线程编程指南》	涵盖 POSIX 和 Solaris 线程 API、使用同步对象进行程序设计、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 门户网站 http://developers.sun.com/prodtech/cc 以查找以下经常更新的资源：

- 关于编程技术和最佳实例的文章
- 有关编程小技巧的知识库
- 软件的文档，以及随软件一同安装的文档的更正信息
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

Sun Studio 门户网站是 Sun Developer Network 网站 http://developers.sun.com 上的很多额外开发者资源之一。

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，请访问以下网址：

```
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。您可以通过以下网址提交您的意见和建议：

```
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 819-4817-10。

第1章

简介

可以通过 SPARC® 和 Intel x86 系统上的 Sun 浮点环境开发可靠、高性能和可移植的数值应用程序。这个浮点环境还可以帮助您研究他人编写的数值程序的反常行为。这些系统实现了符合 IEEE 754 标准的二进制浮点运算的算法模型。本手册解释了在这些系统上如何使用 IEEE 标准提供的选项和灵活性。

1.1 浮点环境

浮点环境由数据结构和运算组成，并通过硬件、系统软件和软件库提供给编程人员，实现了 IEEE 754 标准。IEEE 754 标准可以帮助您更加容易地编写数值应用程序。它是计算机算法的坚实、全面的基础，推动了数值编程技术。

例如，硬件提供与 IEEE 数据格式对应的存储格式、对此类格式数据的运算、对这些运算生成的结果舍入的控制、指示出现 IEEE 数字异常的状态标记以及当发生此类异常且没有用户定义的处理程序时 IEEE 规定的结果。系统软件支持 IEEE 异常处理。软件库（包括数学库 `libm` 和 `libsunmath`）按照 IEEE 754 标准在处理异常方面所采用的方式，实现了诸如 `exp(x)` 和 `sin(x)` 等函数。（当浮点算法运算没有明确的结果时，系统通过**引发异常**向用户通报此情况。）数学库还提供处理特殊 IEEE 值的函数调用，例如 `Inf`（无穷大）或 `NaN`（非数）。

浮点环境的三个组成部分以微妙的方式进行交互，应用程序编程人员通常看不到这些交互。编程人员只能看到 IEEE 标准规定或建议的计算机制。一般来说，本手册指导编程人员充分而有效地使用 IEEE 机制，以使他们能够有效地编写应用程序软件。

很多有关浮点算法的问题都涉及数的基本运算。例如，

- 如果在计算机系统中无法表示无限精确的结果，则运算结果是什么？
- 有些基本运算（比如乘法和加法）是否具有交换性？

另一类问题与异常和异常处理有关。例如，如果满足以下条件，会发生什么情况：

- 两个很大的数相乘？
- 被零除？
- 试图计算负数的平方根？

在某些其他算法中，第一类问题可能没有预期的答案，或者按相同的方式处理第二类中的异常情况：程序立即终止；在某些很旧的机器中，计算继续进行，但产生垃圾。

IEEE 754 标准确保运算产生预期的数学结果，并且结果具有预期的特性。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。

在本手册中，包含类似 `NaN` 或**次正规数**等术语的参考资料。术语表定义了与浮点算法有关的术语。

第2章

IEEE 算法

本章讨论用于二进制浮点算法的 ANSI/IEEE 754-1985 标准 （简称为 “IEEE 标准”或 “IEEE 754”）所指定的算法模型。所有 SPARC 和 x86 处理器均使用 IEEE 算法。所有 Sun 编译器产品均支持 IEEE 算法的特性。

2.1 IEEE 算法模型

本节介绍 IEEE 754 规范。

2.1.1 什么是 IEEE 算法？

IEEE 754 指定：

- 两种基本的浮点格式：**单精度**和**双精度**。

  IEEE 单精度格式具有 24 位有效数字精度，并总共占用 32 位。IEEE 双精度格式具有 53 位有效数字精度，并总共占用 64 位。

- 两种扩展浮点格式：**单精度扩展**和**双精度扩展**。

  此标准并未规定这些格式的精确精度和和大小，但它指定了最小精度和大小。例如，IEEE 双精度扩展格式必须至少具有 64 位有效数字精度，并总共占用至少 79 位。

- 浮点运算的准确度要求：**加、减、乘、除、平方根、余数、将浮点格式的数舍入为整数值、在不同浮点格式之间转换、在浮点和整数格式之间转换以及比较**。

  求余和比较运算必须精确无误。其他的每种运算必须向其目标提供精确的结果，除非没有此类结果，或者该结果不满足目标格式。对于后一种情况，运算必须按照下面介绍的规定舍入模式的规则对精确结果进行最低限度的修改，并将经过此类修改的结果提供给运算的目标。

- 在十进制字符串和两种基本浮点格式之一的二进制浮点数之间进行转换的准确度、单一性和一致性要求。

对于在指定范围内的操作数，这些转换必须生成精确的结果（如果可能的话），或者按照规定舍入模式的规则，对此类精确结果进行最低限度的修改。对于不在指定范围内的操作数，这些转换生成的结果与精确结果之间的差值不得超过取决于舍入模式的指定误差。

- 五种类型的 IEEE 浮点异常，以及用于向用户指示发生这些类型异常的条件。

  五种类型的浮点异常是：**无效运算**、**被零除**、**上溢**、**下溢**和**不精确**。

- 四种舍入方向：**向最接近的可表示的值**；当有两个最接近的可表示的值时首选“偶数”值；**向负无穷大**（向下）；**向正无穷大**（向上）以及**向 0**（截断）。
- 舍入精度；例如，如果系统提供双精度扩展格式的结果，则用户可以指定将此类结果舍入到单精度格式或双精度格式的精度。

IEEE 标准还建议为用户进行异常处理提供支持。

IEEE 标准所需的特性可以支持区间算法、异常的回顾诊断、有效地实现标准基本函数（如 `exp` 和 `cos`）、多精度算法以及用于数值计算的很多其他工具。

与任何其他种类的浮点算法相比，IEEE 754 浮点算法为用户提供了更好的计算控制。IEEE 标准不仅严格要求遵循实现标准，而且还使得此类实现改进和完善了标准本身，从而简化了编写复杂的可移植数值程序的任务。

2.2 IEEE 格式

本节介绍如何将浮点数据存储在内存中。它汇总了各种 IEEE 存储格式的精度和范围。

2.2.1 存储格式

浮点格式是一种数据结构，用于指定包含浮点数的字段、这些字段的布局及其算术解释。浮点**存储**格式指定如何将浮点格式存储在内存中。IEEE 标准定义了这些格式，但具体选择哪种存储格式由实现工具决定。

汇编语言软件有时取决于所使用的存储格式，但更高级别的语言通常仅处理浮点数据类型的语言概念。这些类型在不同的高级语言中具有不同的名称，并且与表 2-1 中所示的 IEEE 格式相对应。

表 2-1 IEEE 格式和语言类型

IEEE 精度	C、C++	Fortran（仅限 SPARC）
单精度	float	`REAL` 或 `REAL*4`
双精度	double	`DOUBLE PRECISION` 或 `REAL*8`
双精度扩展	long double	`REAL*16`

IEEE 754 明确规定了单精度浮点格式和双精度浮点格式，并为这两种基本格式分别定义了一组扩展格式。表 2-1 中显示的 `long double` 和 `REAL*16` 类型适用于 IEEE 标准定义的一种双精度扩展格式。

以下几节详细介绍了 SPARC 和 x86 平台上用于 IEEE 浮点格式的每种存储格式。

2.2.2 单精度格式

IEEE 单精度格式由三个字段组成：23 位小数 `f`； 8 位偏置指数 `e`；以及 1 位符号 `s`。这些字段连续存储在一个 32 位字中（如图 2-1 所示）。0:22 位包含 23 位小数 `f`，其中第 0 位是小数的最低有效位，第 22 位是最高有效位； 23:30 位包含 8 位偏置指数 `e`，第 23 位是偏置指数的最低有效位，第 30 位是最高有效位；最高的第 31 位包含符号位 `s`。

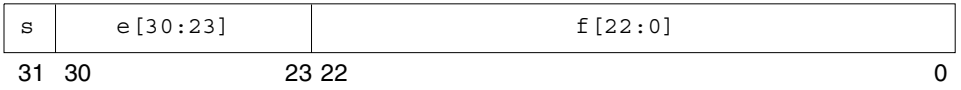


图 2-1 单精度存储格式

表 2-2 显示一侧的三个组成字段 `s`、`e` 和 `f` 的值与另一侧的单精度格式位模式表示的值之间的对应关系；*u* 意味着**无关**，即指示字段的值与确定特定单精度格式位模式的值无关。

表 2-2 IEEE 单精度格式位模式表示的值

单精度格式位模式	值
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$ （正规数）
$e = 0$; $f \neq 0$ （`f` 中至少有一位不为零）	$(-1)^s \times 2^{-126} \times 0.f$ （次正规数）
$e = 0$; $f = 0$ （`f` 中的所有位均为零）	$(-1)^s \times 0.0$ （有符号的零）

表 2-2 IEEE 单精度格式位模式表示的值 (续)

单精度格式位模式	值
s = 0; e = 255; f = 0 （f 中的所有位均为零）	+INF （正无穷大）
s = 1; e = 255; f = 0 （f 中的所有位均为零）	–INF （负无穷大）
s = u; e = 255; f ≠ 0 （f 中至少有一位不为零）	NaN （非数）

注意，当 e < 255 时，为单精度格式位模式分配的值是使用以下方法构成的：将二进制基数点插入到紧邻小数最高有效位的左侧，将一个隐含位插入到紧邻二进制点的左侧，因而以二进制位置表示法来表示一个带分数 （整数加小数，其中 0 ≤ Ò3 小数 < 1）。

如此构成的带分数称为**单精度格式有效数字**。之所以称为隐含位的原因是，在单精度格式位模式中没有显式地指定其值，但偏置指数字段的值隐式指定了该值。

对于单精度格式，正规数和次正规数的差别在于正规数有效数字的前导位 （二进制点左侧的位）为 1，而次正规数有效数字的前导位为 0。在 IEEE 754 标准中，单精度格式次正规数称为单精度格式非规格化数。

在单精度格式正规数中 23 位小数加上隐含前导有效数位提供了 24 位精度。

表 2-3 中给出了重要的单精度存储格式位模式的示例。最大正正规数是以 IEEE 单精度格式表示的最大有限数。最小正次正规数是以 IEEE 单精度格式表示的最小正数。最小正正规数通常称为下溢阈值。（最大和最小正规数和次正规数的十进制值是近似的；对于所示的数字来说，它们是正确的。）

表 2-3 单精度存储格式位模式及其 IEEE 值

通用名称	位模式 （十六进制）	十进制值
+0	00000000	0.0
-0	80000000	-0.0
1	3f800000	1.0
2	40000000	2.0
最大正规数	7f7fffff	3.40282347e+38
最小正正规数	00800000	1.17549435e-38
最大次正规数	007fffff	1.17549421e-38
最小正次正规数	00000001	1.40129846e-45
+∞	7f800000	无穷
–∞	ff800000	负无穷
非数	7fc00000	NaN

NaN（非数）可以用任何满足 NaN 定义的位模式表示。在表 2-3 中显示的 NaN 十六进制值只是可用于表示 NaN 的众多位模式之一。

2.2.3 双精度格式

IEEE 双精度格式由三个字段组成：52 位小数 `f`； 11 位偏置指数 `e`；以及 1 位符号 `s`。这些字段连续存储在两个 32 位字中 （如图 2-2 所示）。

在 SPARC 体系结构中，较高地址的 32 位字包含小数的 32 位最低有效位，而在 x86 体系结构中，则较低地址的 32-位字包含小数的 32 位最低有效位。

如果用 `f`[31:0] 表示小数的 32 位最低有效位，则在这 32 位最低有效位中，第 0 位是整个小数的最低有效位，而第 31 位则是最高有效位。

在另一个 32 位字中，0:19 位包含 20 位小数的最高有效位 `f`[51:32]，其中第 0 位是这 20 位最高有效位中的最低有效位，而第 19 位是整个小数的最高有效位； 20:30 位包含 11 位偏置指数 `e`，其中第 20 位是偏置指数的最低有效位，而第 30 位是最高有效位；最高的第 31 位包含符号位 `s`。

图 2-2 将这两个连续的 32 位字按一个 64 位字那样进行了编号，其中 0:51 位存储 52 位的小数 `f`； 52:62 位存储 11 位偏置指数 `e`；而第 63 位存储符号位 `s`。

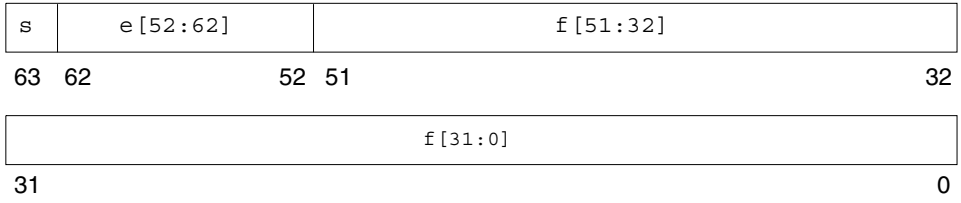


图 2-2 双精度存储格式

这三个字段中的位模式值将决定整个位模式所表示的值。

表 2-4 显示一侧的三个组成字段中位的值与另一侧双精度格式位模式表示值的对应关系；*u* 意味着**无关**，即指示字段的值与确定特定双精度格式位模式的值无关。

表 2-4 IEEE 双精度格式位模式表示的值

双精度格式位模式	值
0 < `e` < 2047	$(-1)^s \times 2^{e-1023}$ x 1.`f` （正规数）
`e` = 0; `f` ≠ 0 （`f` 中至少有一位不为零）	$(-1)^s \times 2^{-1022}$ x 0.`f` （次正规数）
`e` = 0; `f` = 0 （`f` 中的所有位均为零）	$(-1)^s \times 0.0$ （有符号的零）

表 2-4 IEEE 双精度格式位模式表示的值 (续)

双精度格式位模式	值
s = 0; e = 2047; f = 0 （f 中的所有位均为零）	+INF （正无穷大）
s = 1; e = 2047; f = 0 （f 中的所有位均为零）	-INF （负无穷大）
s = u; e = 2047; f ≠ 0 （f 中至少有一位不为零）	NaN （非数）

请注意，当 e < 2047 时，赋予双精度格式位模式的值是使用以下方法构成：将二进制基数点插入到紧邻小数最高有效位的左侧，将一个隐含位插入到紧邻二进制点的左侧。如此构成的数字称作**有效数字**。之所以称为隐含位的原因是，在双精度格式位模式中没有显式地指定其值，但偏置指数字段的值隐式指定了该值。

对于双精度格式，正规数和次正规数的差别在于正规数有效数字的前导位 （二进制点左侧的位）为 1，而次正规数有效数字的前导位为 0。在 IEEE 标准 754 中，双精度格式次正规数称为双精度格式非规格化数。

在双精度格式正规数中 52 位小数加上隐含前导有效数位提供了 53 位精度。

表 2-5 中给出了重要的双精度存储格式位模式的示例。第二列中的位模式显示为两个 8 位十六进制数。对于 SPARC 体系结构，左侧是较低地址的 32 位字的值，右侧是较高地址的 32 位字的值，而对于 x86 体系结构，左侧是较高地址的字，右侧是较低地址的字。最大正正规数是以 IEEE 双精度格式表示的最大有限数。最小正次正规数是以 IEEE 双精度格式表示的最小正数。最小正正规数通常称为下溢阈值。（最大和最小正规数和次正规数的十进制值是近似的；对于所示的数字来说，它们是正确的。）

表 2-5 双精度存储格式位模式及其 IEEE 值的位模式

通用名称	位模式 （十六进制）	十进制值
+ 0	00000000 00000000	0.0
- 0	80000000 00000000	-0.0
1	3ff00000 00000000	1.0
2	40000000 00000000	2.0
最大正规数	7fefffff ffffffff	1.7976931348623157e+308
最小正正规数	00100000 00000000	2.2250738585072014e-308
最大次正规数	000fffff ffffffff	2.2250738585072009e-308
最小正次正规数	00000000 00000001	4.9406564584124654e-324
+∞	7ff00000 00000000	无穷
–∞	fff00000 00000000	负无穷
非数	7ff80000 00000000	NaN

NaN（非数）可以用任何满足 NaN 定义的位模式表示。在表 2-5 中显示的 NaN 十六进制值只是可用于表示 NaN 的众多位模式之一。

2.2.4 双精度扩展格式 (SPARC)

SPARC 浮点环境的四倍精度格式符合双精度扩展格式的 IEEE 定义。四倍精度格式占用 32 位字并包含以下三个字段：112 位小数 f、15 位偏置指数 e 和 1 位符号 s。这三个字段连续存储，如图 2-3 所示。

地址最高的 32 位字包含小数的 32 位最低有效位，用 f[31:0] 表示。紧邻的两个 32 位字分别包含 f[63:32] 和 f[95:64]。下面的 0:15 位包含小数的 16 位最高有效位 f[111:96]，其中第 0 位是这 16 位的最低有效位，而第 15 位是整个小数的最高有效位。16:30 位包含 15 位偏置指数 e，其中第 16 位是该偏置指数的最低有效位，而第 30 位是最高有效位；第 31 位包含符号位 s。

图 2-3 将这四个连续的 32 位字按一个 128 位字那样进行了编号，其中 0:111 位存储小数 f； 112:126 位存储 15 位偏置指数 e；而第 127 位存储符号位 s。

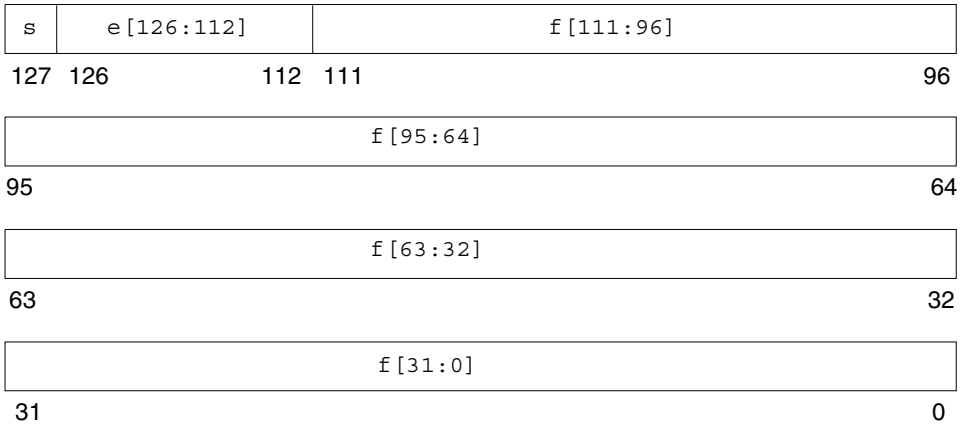


图 2-3 双精度扩展格式 (SPARC)

f、 e 和 s 三个字段中的位模式值将决定整个位模式所表示的值。

表 2-6 显示了三个组成字段的值与四倍精度格式位模式表示的值之间的对应关系。u 意味着 “无关”，即指示字段的值与确定特定位模式的值无关。

表 2-6 位模式表示的值 (SPARC)

双精度扩展位模式 (SPARC)	值
0 < e < 32767	$(1)^s \times 2^{e-16383} \times 1.f$ （正规数）
e = 0、 f ≠ 0 （f 中至少有一位不为零）	$(1)^s \times 2^{-16382} \times 0.f$ （次正规数）
e = 0、 f = 0 （f 中的所有位均为零）	$(1)^s \times 0.0$ （有符号的零）
s = 0、 e = 32767、 f = 0 （f 中的所有位均为零）	+INF （正无穷大）
s = 1、 e = 32767； f = 0 （f 中的所有位均为零）	-INF （负无穷大）
s = u、 e = 32767、 f ≠ 0 （f 中至少有一位不为零）	NaN （非数）

表 2-7 中给出了重要的四倍精度双精度扩展存储格式位模式的示例。第二列中的位模式显示为四个 8 位十六进制数。最左侧的数是地址最低的 32 位字的值，而最右侧的数是地址最高的 32 位字的值。最大正正规数是以四倍精度格式表示的最大有限数。最小正次正规数是以四倍精度精度格式表示的最小正数。最小正正规数通常称为下溢阈值。（最大和最小正规数和次正规数的十进制值是近似的；对于所示的数字来说，它们是正确的。）

表 2-7 双精度扩展格式位模式 (SPARC)

通用名称	位模式 (SPARC)	十进制值
+0	00000000 00000000 00000000 00000000	0.0
-0	80000000 00000000 00000000 00000000	-0.0
1	3fff0000 00000000 00000000 00000000	1.0
2	40000000 00000000 00000000 00000000	2.0
最大正规数	7ffeffff ffffffff ffffffff ffffffff	1.189731495357231765085759326628007 0e+4932
最小正规数	00010000 00000000 00000000 00000000	3.362103143112093506262677817321752 6e-4932
最大次正规数	0000ffff ffffffff ffffffff ffffffff	3.362103143112093506262677817321752 0e-4932
最小正次正规数	00000000 00000000 00000000 00000001	6.475175119438025110924438958227646 6e-4966

表 2-7 双精度扩展格式位模式 (SPARC)(续)

通用名称	位模式 (SPARC)	十进制值
+∞	7fff0000 00000000 00000000 00000000	+∞
−∞	ffff0000 00000000 00000000 00000000	−∞
非数	7fff8000 00000000 00000000 00000000	NaN

在表 2-7 中显示的 NaN 十六进制值只是可用于表示 NaN 的众多位模式之一。

2.2.5 双精度扩展格式 (x86)

该浮点环境双精度扩展格式符合双精度扩展格式的 IEEE 定义。它包含四个字段：63 位小数 `f`、1 位显式前导有效数位 `j`、15 位偏置指数 `e` 以及 1 位 符号 `s`。

在 x86 体系结构系列中，这些字段连续存储在十个相连地址的 8 位字节中。由于 UNIX System V Application Binary Interface Intel 386 Processor Supplement (Intel ABI) 要求双精度扩展参数，从而占用堆栈中三个相连地址的 32 位字，其中地址最高字的 16 位最高有效位未用，如图 2-4 所示。

地址最低的 32 位字包含小数的 32 位最低有效位 `f`[31:0]，其中第 0 位是整个小数的最低有效位，而第 31 位则是 32 位最低有效位的最高有效位。地址居中的 32 位字中，0:30 位包含小数的 31 位最高有效位 `f`[62:32]（其中第 0 位是这 31 位最高有效位的最低有效位，而第 30 位是整个小数的最高有效位）；地址居中 32 位字的第 31 位包含显式前导有效数位 `j`。

地址最高的 32 位字中， 0:14 位包含 15 位偏置指数 `e`，其中第 0 位是该偏置指数的最低有效位，而第 14 位是最高有效位；第 15 位包含符号位 `s`。虽然地址最高的 32 位字的最高 16 位未被 x86 体系结构系列使用，但如上所述，它们对于符合 Intel ABI 规定是至关重要的。

图 2-4 将这三个连续的 32 位字按一个 96 位字那样进行了编号，其中 0:62 位存储 63 位小数 `f`； 第 63 位存储显式前导有效数位 `j`； 64:78 位存储 15 位偏置指数 `e`； 第 79 位存储符号位 `s`。

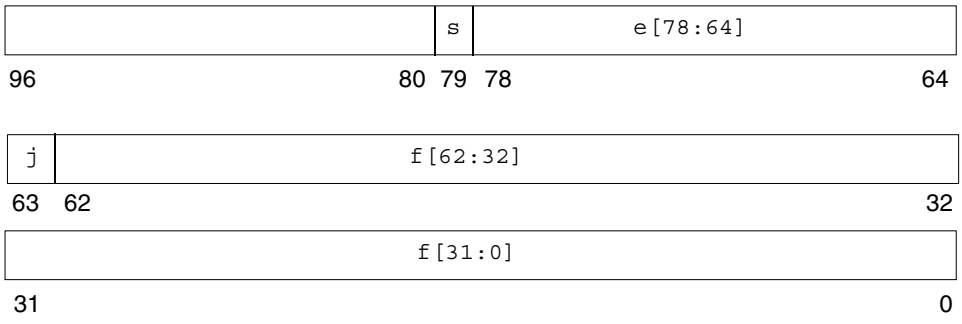


图 2-4 双精度扩展格式 (x86)

f、 j、 e 和 s 四个字段中的位模式值将决定整个位模式所表示的值。

表 2-8 显示了四个组成字段的计数值与该位模式表示的值之间的对应关系。 *u* 意味着**无关**，即指示字段的值与确定特定位模式的值无关。

表 2-8 位模式表示的值 (x86)

双精度扩展位模式 (x86)	**值**
j = 0、 0 < e <32767	不支持
j = 1、 0 < e <32767	$(1)^s$ x $2^{e-16383}$ x 1.f （正规数）
j = 0、 e = 0； f ≠ 0 （f 中至少有一位不为零）	$(1)^s$ x 2^{-16382} x 0.f （次正规数）
j = 1、 e = 0	$(1)^s$ x 2^{-16382} x 1.f （伪非正规数）
j = 0、 e = 0、 f = 0 （f 中的所有位均为零）	$(1)^s$ x 0.0 （有符号的零）
j = 1； s = 0； e = 32767； f = 0 （f 中的所有位均为零）	+INF （正无穷大）
j = 1； s = 1； e = 32767； f = 0 （f 中的所有位均为零）	-INF （负无穷大）
j = 1； s = *u*； e = 32767； f = .1*uuu uu*	QnaN （静态 NaN）
j = 1； s = *u*； e = 32767； f = .0*uuu uu* ≠ 0 （f 中至少有一个 *u* 不为零）	SnaN （信号 NaN）

请注意，双精度扩展格式的位模式**没有**显式前导有效数位。在双精度扩展格式中，前导有效数位在单独的字段 j 中明确给出。但当 e ≠ 0 时，将不支持任何 j = 0 的位模式，这是因为将这种位模式用作浮点运算中的操作数将导致无效的运算异常。

将双精度扩展格式中的分立字段 j 和 f 连接起来称为**有效数字**。当 e < 32767 和 j = 1 时，或当 e = 0 和 j = 0 时，有效数字是通过以下方法形成的：在前导有效数位 j 和小数的最高有效位之间插入二进制基数点。

在 x86 双精度扩展格式中，前导有效数位 j 是 0 并且偏置指数字段 e 也是 0 的位模式表示次正规数，而前导有效数位 j 是 1 并且偏置指数字段 e 是非零数的位模式表示正规数。由于前导有效数位是明确表示出来的，而不是从指数的值推导出来的，所以该格式还接受偏置指数是 0（与次正规数相似），而前导有效数位是 1 的位模式。每一个这样的位模式实际上都与对应的偏置指数字段是 1 的位模式表示相同的值，即正规数，因此位模式称为**伪非正规数**。（在 IEEE 标准 754 中，次正规数称为非正规化数字。）伪非正规数仅是一个 x86 双精度扩展格式编码的人为概念，当显示为操作数时，您可以将其隐式转换为相应的正规数，不能将其生成为结果。

表 2-9 中给出了重要的双精度扩展存储格式位模式的示例。第二列中的位模式显示为一个 4 位十六进制计数，它是地址最高的 32 位字的 16 位最低有效位的值（还记得上述该地址最高的 32 位字的 16 位最高有效位是未用的，所以未显示其值），后面是两个 8 位十六进制计数，其中左侧是地址居中的 32 位字的值，右侧是地址最低的 32 位字的值。最大正正规数是以 x86 双精度扩展格式表示的最大有限数。最小正次正规数是以双精度扩展格式表示的最小正数。最小正正规数通常称为下溢阈值。（最大和最小正规数和次正规数的十进制值是近似的；对于所示的数字来说，它们是正确的。）

表 2-9 双精度扩展格式位模式及其值 (x86)

通用名称	位模式 (x86)	十进制值
+0	0000 00000000 00000000	0.0
-0	8000 00000000 00000000	-0.0
1	3fff 80000000 00000000	1.0
2	4000 80000000 00000000	2.0
最大正规数	7ffe ffffffff ffffffff	1.189731495357231765O5e+4932
正最小正规数	0001 80000000 00000000	3.3621031431120935062 6e-4932
最大次正规数	0000 7fffffff ffffffff	3.3621031431120935060 8e-4932
最小正次正规数	0000 00000000 00000001	3.64519953188247460253e-4951
+∞	7fff 80000000 00000000	+∞
–∞	ffff 80000000 00000000	–∞
带有最大小数的静态 NaN	7fff ffffffff ffffffff	QNaN
带有最小小数的静态 NaN	7fff c0000000 00000000	QNaN
带有最大小数的信号 NaN	7fff bfffffff ffffffff	SNaN
带有最小小数的信号 NaN	7fff 80000000 00000001	SNaN

NaN（非数）可以用任何满足 NaN 定义的位模式表示。表 2-9 中的 NaN 十六进制值显示出，小数字段的前导位（最高有效位）决定 NaN 是静态（前导小数位 = 1）的还是信号（前导小数位 = 0）的。

2.2.6 十进制表示法的范围和精度

本节讨论给定存储格式的范围和精度概念。本节包含的范围和精度与 IEEE 单精度格式和双精度格式以及与 SPARC 和 x86 体系结构上 IEEE 双精度扩展格式的实现相对应。为了具体起见，我们用 IEEE 单精度格式来定义范围和精度概念。

IEEE 标准指定使用 32 位来表示单精度格式的浮点数。由于 32 个零和一的组合是有限的，所以使用 32 位仅能表示有限个数字。

于是会出现这样一个很自然的问题：

使用这种特定格式表示最大和最小正数的十进制表示法是什么？

下面我们将这一问题改述来引入范围的概念：

在十进制概念中，使用 IEEE 单精度格式可以表示的数字的范围是什么？

考虑到 IEEE 单精度格式的准确定义，我们可以证明使用 IEEE 单精度格式可以表示的浮点数的范围（在限定在正的正规化数的基础上）如下所示：

$1.175... \times (10^{-38})$ 到 $3.402... \times (10^{+38})$

第二个问题涉及到用给定格式表示的数字的精度（不要与准确度或有效数字数混淆）。我们将通过一些图片和示例来解释这些概念。

二进制浮点计算的 IEEE 标准指定可以用单精度格式表示数字值集。请记住，我们将这种数字值集解释为二进制浮点数字集。 IEEE 单精度格式的有效数字有 23 位，加上隐式前导位，可以得到 24 位（二进制）精度。

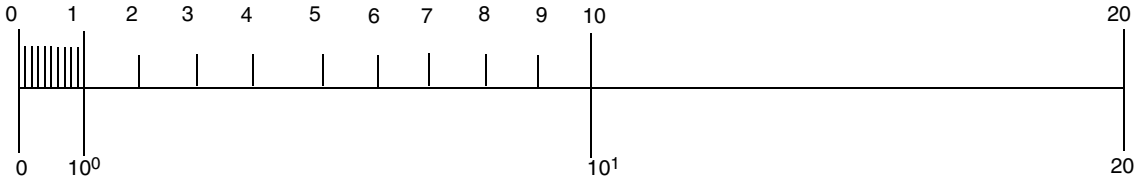
我们可以通过用以下方式标记数字来获得另一个数字值集：

$x = (x_1.x_2\ x_3...x_q)\ (10n)$

（可以表示为 q 个用有效数字表示的十进制数字），如数轴所示。

图 2-5 演示的就是这种情况：

十进制表示法：



二进制表示法：

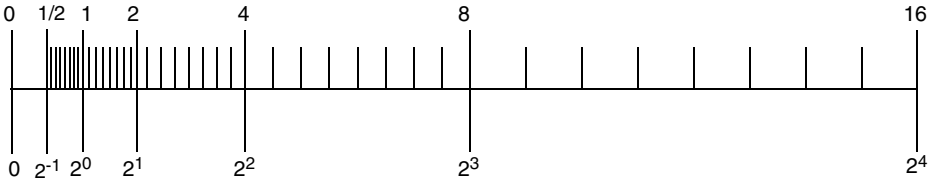


图 2-5 使用十进制表示法和二进制表示法定义的数字集比较

请注意，两个数字集是不同的。因此，要估算相对于 24 位有效二进制数字的有效十进制数字的数量，需要将该问题换一种形式表示。

根据在二进制表示法（计算机使用的内部格式）与十进制格式（用户通常使用的格式）之间转换浮点数的方法，重新表示该问题。事实上，您可能希望先从十进制转换为二进制，然后再转换为十进制，或者先从二进制转换为十进制，然后再转换为二进制。

需要记住的要点是，一般来说，由于数字集不同，转换是不精确的。在正确执行的情况下，将一个集中的数字转换为另一个集中的数字，会导致选择第二个集中两个相邻数字中的一个（确切来说，这是一个与舍入相关的问题）。

我们来考虑一些示例。假定要用以下 IEEE 单精度格式的十进制表示法表示数字：

x = x1.x2 x3... $10n$

由于只能使用 IEEE 单精度格式精确表示有限的实数，而这其中并没有包含所有上述形式的数字，所以一般来说，是无法精确表示这些数字的。例如，使

y = 838861.2、 z = 1.3

并运行以下 Fortran 程序：

```
      REAL Y, Z
      Y = 838861.2
      Z = 1.3
      WRITE(*,40) Y
40    FORMAT("y: ",1PE18.11)
      WRITE(*,50) Z
50    FORMAT("z: ",1PE18.11)
```

该程序的输出应与以下内容类似：

```
y:      8.38861187500E+05
z:      1.29999995232E+00
```

赋予 *y* 的值 8.388612x10^5 与输出的值之差是 0.000000125，它比 *y* 小七个数量级。用 IEEE 单精度格式表示 *y* 的精确度约为 6 到 7 位有效数字，也就是说，如果要表示为 IEEE 单精度格式， *y* 大约有**六位有效**数字。

同理，赋予 *z* 的值 1.3 与输出的值之差是 0.00000004768，它比 *z* 小八个数量级。用 IEEE 单精度格式表示 *z* 的精确度约为 7 到 8 位有效数字，也就是说，如果要表示为 IEEE 单精度格式， *z* 大约有七位**有效数字**。

现在，我们用公式阐述一下这个问题：

假定将十进制浮点数 *a* 转换为其 IEEE 单精度格式二进制表示法 *b*，然后将 *b* 再转换为十进制数 *c*；那么， *a* 与 *a* - *c* 之间相差多少数量级呢？

我们将这一问题改述如下：

用 IEEE 单精度格式表示法表示的 *a* 的**有效十进制数字**数是多少，或当我们用 IEEE 单精度格式表示 *x* 时，有多少十进制数可以被当作是精确的？

有效十进制数字数总是介于 6 和 9 之间，也就是说，最少 6 个，但不超过 9 个数字是精确的 （除去例外情况，例如，当转换是精确的情况，当有无限多的数字可以是精确的情况）。

反过来，如果将用 IEEE 单精度格式表示的二进制数转换为十进制数时，然后再转换为二进制，一般来说，您需要使用至少 9 位十进制数，以确保在经过两次转换后，能够获得转换前的数字。

表 2-10 列出了这一问题的完整说明：

表 2-10 存储格式的范围和精度

格式	有效数字 （二进制）	最小正正规数	最大正数	有效数字 （十进制）
单精度	24	1.175... 10^{-38}	3.402... 10^{+38}	6-9
双精度	53	2.225... 10^{-308}	1.797... 10^{+308}	15-17
双精度扩展 (SPARC)	113	3.362... 10^{-4932}	1.189... 10^{+4932}	33-36
双精度扩展 (x86)	64	3.362... 10^{-4932}	1.189... 10^{+4932}	18-21

2.2.7 Solaris 环境中的基数转换

基数转换指将在一个基数中表示的数字转换为在另一个基数中表示的数字。C 中 `printf` 和 `scanf` 以及 Fortran 中的 `read`、`write` 和 `print` 等 I/O 例程都涉及到用基数 2 和基数 10 表示的数字间的转换：

- 当读取用传统十进制表示法表示的数字并将其用内部二进制格式存储时，就会执行从基数 10 到基数 2 的转换。
- 当将内部二进制值作为十进制 ASCII 字符串打印时，则会执行从基数 2 到基数 10 的转换。

在 Solaris 环境中，供所有语言使用的基数转换基础例程包含在标准 C 库 `libc` 中。这些例程使用表驱动算法，这种算法可以在输入格式和输出格式之间实现正确舍入的转换（服从对所涉及的十进制数字字符串长度的适当限制）。除了其精确性外，表驱动算法还减少了正确舍入基数转换出现最差情况的次数。

IEEE 标准要求对数量级从 10^{-44} 到 10^{+44} 的一般数字要正确舍入，而对更大的指数则允许微小差别的舍入。（请参见 IEEE 标准 754 的 5.6 节。） `libc` 表驱动算法可以对单精度、双精度和双精度扩展格式进行正确的舍入。

在 C 中，根据 IEEE 754，总是可以对十进制字符串与二进制浮点值之间的转换进行正确舍入：转换后的结果是结果的格式可以表示的数字，在当前舍入模式指定的方向下，它与原值最接近。当舍入模式是舍入到最接近值并且原值位于两个可用结果格式表示的数字的正中间时，则转换后结果的最低有效位数字是偶数。这些规则适用于编译器执行的源代码中的常数转换，也适用于程序使用标准库例程执行的数据转换。

在 Fortran 中，可以根据与 C 默认设置相同的规则，对十进制字符串和二进制浮点值进行正确的舍入。对于 I/O 转换，可以使用程序中的 `ROUNDING=` 说明符或利用 `-iorounding` 标记编译，覆盖舍入到最接近模式中的“舍入到偶数”规则。有关详细信息，请参见《Fortran 用户指南》和 `f95`(1) 手册页。

有关基数转换的参考信息，请参见附录 F。Coonen 的论述和 Sterbenz 的书都是非常好的参考资料。

2.3 下溢

简而言之，下溢发生在以下情况下：算法运算的结果非常小，必须允许存在大于常规情况的舍入误差，才能以其预期的目标格式存储它。

2.3.1 下溢阈值

表 2-11 显示了单精度、双精度和双精度扩展格式的下溢阈值。

表 2-11 下溢阈值

目标精度	下溢阈值	
单精度	最小正规数	1.17549435e-38
	最大次正规数	1.17549421e-38
双精度	最小正规数	2.2250738585072014e-308
	最大次正规数	2.2250738585072009e-308
双精度扩展 (SPARC)	最小正规数	3.3621031431120935062626778173217526e-4932
	最大次正规数	3.3621031431120935062626778173217520e-4932
双精度扩展 (x86)	最小正规数	3.3621031431120935062e-4932
	最大次正规数	3.3621031431120935059e-4932

正次正规数是介于零和最小正规数之间的数。从最小正规数减去两个（正）与之接近的微小数可以生成次正规数。另外，用最小正正规数除以二也可以生成次正规数。

虽然次正规数本身的精度位数少于正规数，但利用次正规数可以提高微小数字的浮点计算精度。在数学计算中，当生成的正确结果的数量级低于最小正正规数时，就会生成次正规数（而不是返回零），这称为渐进下溢。

要处理这种**下溢**结果，还有其他几种方法可供使用。一种过去常用的方法是，将这些结果刷新为零。这种方法称为 *Store 0*，在引入 IEEE 标准之前，这是大多数大型机的默认设置。

一方面是获取一种强有力的数学解决方案的愿望，另一方面是创建一种可以有效实施的标准，在权衡这两方面的过程中，起草 IEEE 标准 754 的数学家和计算机设计人员考虑过多种方法。

2.3.2 IEEE 算法如何处理下溢？

IEEE 标准 754 选择渐进下溢作为处理下溢结果的首选方法。这种方法可以归结为定义两种存储值的表示方法：正规数和次正规数。

您应该还记得正规浮点数的 IEEE 格式：

$$(-1)^s \times (2^{(e-bias)}) \times 1.f$$

其中 *s* 是符号位，*e* 是偏置指数，*f* 是小数。要完整指定数字，仅需要存储 *s*、*e* 和 *f*。由于对于正规数，将有效数字的隐式前导位定义为 1，所以需要存储它。

于是，可以存储的最小正正规数具有负的最大数量级指数和全为零的小数。如果前导位是零而不是一，则可以容纳更小的数字。在双精度格式中，由于小数部分的长度为 52 位（约 16 位十进制数字），这可以有效地将最小指数从 10^{-308} 扩展到 10^{-324}。这些是**次正规数**；返回次正规数（而不是将下溢结果刷新为零）就是**渐进下溢**。

很明显，次正规数越小，其小数中的非零位越多；生成次正规数结果的计算与正规操作数计算所遵循的相对舍入误差边界不同。但是，有关渐进下溢的主要事实是其使用以下隐含条件：

- 下溢结果不必允许牺牲比普通舍入误差更大的准确度。
- 当结果非常小时，加法、减法、比较和余数运算都总是准确的。

您应该还记得次正规浮点数的 IEEE 格式：

$$(-1)^s \times (2^{(-bias+1)}) \times 0.f$$

其中 *s* 是符号位，偏置指数 *e* 是零，*f* 是小数。请注意，隐式 2 的幂偏差比正规格式偏差大一，隐式前导小数位是零。

渐进下溢允许扩展可表示数的下限。它不是值引起人怀疑的**最小限度**，而是其他相关的误差。算法使用的是较其他系统误差更小的次正规数。下一节将介绍渐进下溢的数学根据。

2.3.3 为什么使用渐进下溢？

次正规数的用途不是为了完全避免下溢 / 溢出，这与其他一些数学算法模型是不同的。次正规数使人们不再担心下溢会引起各种计算问题（通常是在执行加法后再执行乘法）。有关详细讨论，请参见 James Demmel 的《Underflow and the Reliability of Numerical Software》和 S. Linnainmaa 的《Combatting the Effects of Underflow and Overflow in Determining Real Roots of Polynomials》。

在算法中使用次正规数，在执行加法和减法运算时，就不会出现未捕获的下溢（这意味着牺牲准确度）了。如果 *x* 和 *y* 是 2 以内的因子，则 *x* - *y* 是没有误差的。对于一些需要有效提高关键位置工作精度的算法来说，这是非常重要的。

另外，渐进下溢意味着下溢引起的误差不会比一般的舍入误差更糟糕。相对于其他处理下溢的方法，这种说法更有说服力，从而，这一事实成为使用渐进下溢的最佳根据。

2.3.4 渐进下溢的误差属性

在大多数情况下，浮点结果都是舍入的：

计算结果 = 真正结果 + 舍入

误差可以达到多大程度？用来表示其大小的一种简便尺度称为**最后一位表示的单位**，简称 *ulp*。用标准表示法表示的浮点数小数的最低有效位即是其**最后一位**。用这一位表示的值（例如，除了这一位外，表示法完全相同的两个数字绝对差值）是该数字的**最后一位**

表示的单位。如果计算结果是通过将真正结果舍入到最接近的可表示数字得到的，则显然，舍入误差不会大于计算结果最后一位表示的单位的一半。换句话说，在舍入到最接近模式的 IEEE 算法中，舍入误差是

0 | 舍入 | 1/2 ulp

这是相对于计算结果来说的。

请注意，ulp 是相对值。如果数字非常大，其 ulp 本身就非常大，而如果数字非常小，则其 ulp 本身也非常小。将 ulp 表示为函数，这种关系会更明显：*ulp(x)* 表示浮点数 *x* 最后一位表示的单位。

另外，浮点数的 ulp 还取决于该数字的表示精度。例如，表 2-12 显示了上述四个浮点格式的 ulp(1) 值：

表 2-12 四种不同精度的 ulp(1)

精度	值
单精度	ulp(1) = 2^-23 ~ 1.192093e-07
双精度	ulp(1) = 2^-52 ~ 2.220446e-16
双精度扩展 (x86)	ulp(1) = 2^-63 ~ 1.084202e-19
四倍精度 (SPARC)	ulp(1) = 2^-112 ~ 1.925930e-34

还记得吗，只有有限数字集才能用计算机算法准确表示。随着数字的数量级的降低并接近零，相邻可表示数字之间的差距会变小。相反，当数字的数量级增加时，相邻可表示数字之间的差距将变大。

例如，假定您要使用只有三个精度位的二进制算法。那么，在任意两个 2 的幂之间，有 $2^3 = 8$ 个可表示数字，如图 2-6 所示。

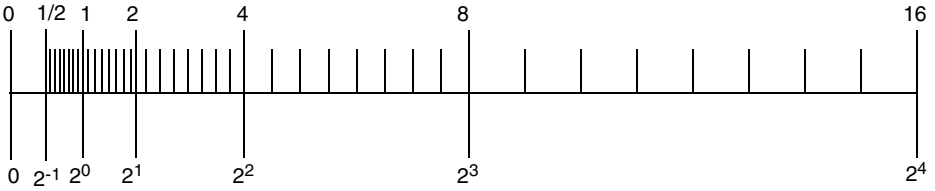


图 2-6 数轴

数轴显示了数字之间的差距是随着指数增加而加倍增加的。

在 IEEE 单精度格式中，两个最小正次正规数之间的差大约是 10^{-45}，而两个最大有限数之间的数量级差大约是 10^{31}！

在表 2-13 中，`nextafter(x,+∞)` 表示在沿着数轴向 +∞ 移动的过程中，`x` 之后的下一个可表示数字。

表 2-13 可表示的单精度浮点数之间的差距

x	nextafter(x, +∞)	差距
0.0	1.4012985e-45	1.4012985e-45
1.1754944e-38	1.1754945e-38	1.4012985e-45
1.0	1.0000001	1.1920929e-07
2.0	2.0000002	2.3841858e-07
16.000000	16.000002	1.9073486e-06
128.00000	128.00002	1.5258789e-05
1.0000000e+20	1.0000001e+20	8.7960930e+12
9.9999997e+37	1.0000001e+38	1.0141205e+31

所有传统的可表示浮点数集都有一个属性，不准确结果的最差情况是：引入一个不比与计算结果一个可表示相邻数字的距离更差的误差。当将次正规数加到可表示集上并实施渐进下溢时，不准确结果或**下溢**结果的最差情况是：引入不大于与计算结果一个可表示相邻数字的距离的误差。

尤其是，在零与最小**正规**数之间的区域，任意两个相邻数字之间的距离等于零与最小**次正规**数之间的距离。利用次正规数，可以避免出现这样的情况：引入的舍入误差大于与最近可表示数字之间的距离。

由于没有计算引起的舍入误差会大于与计算结果任意可表示相邻数字的距离，所以许多强大的算法环境都符合以下三条属性：

- $x \neq y \Leftrightarrow x - y \neq 0$
- $(x - y) + y \approx x$，在 x 和 y 二者中较大者的舍入误差内
- `1/(1/`x`)` $\approx x$，当 x 是正规化数时，表示 `1/`$x \neq 0$

另一种下溢方案是 `Store 0`，它是将下溢结果刷新为零。只要 `x - y` 下溢， `Store 0` 就会违反第一条和第二条属性。另外，当 `1/x` 下溢时， `Store 0` 还将违反第三条属性。

我们用 λ 表示最小正正规化数，也称为下溢阈值。这样，就可以用 λ 来比较渐进下溢和 `Store 0` 的误差属性。

渐进下溢：| 误差 | $< \frac{1}{2}$ λ 的 *ulp*

Store 0：| 误差 | ≈ λ

用 λ 最后一位表示单位的一半与 λ 本身存在着显著的差别。

2.3.5 有关渐进下溢与 Store 0 的两个示例

以下是两个有名的数学示例。第一个示例是计算内积的代码。

```
sum = 0;
for (i = 0; i < n; i++) {
       sum = sum + a[i] * y[i];
}
return sum;
```

利用渐进下溢，结果的准确度为舍入的准确度。在 Store 0 中，可能会在几乎所有数字中传递看上去合理实则错误的微小非零和。然而，公平来讲，我们必须承认，为了避免出现这些问题，聪明的编程人员在预见到会出现微小值影响准确度时，会扩大其计算范围的。

第二个示例源于复数商，不适用于缩放：

$$a + i \cdot b = \frac{p + i \cdot q}{r + i \cdot s}, \text{ 假定 } |r/s| \le 1$$

$$= \frac{(p \cdot (r/s) + q) + i(q \cdot (r/s) - p)}{s + r \cdot (r/s)}$$

可以显示出，虽然进行了舍入，但得出的复数结果与准确结果之间的差不大于以下条件下得出的结果与准确结果之间的差：$p + i \cdot q$ 和$r + i \cdot s$每个值的误差都不大于几个 *ulp*。除了当 *a* 和 *b* 都下溢外，这种误差分析是面向下溢的，误差的限度不会超过以下值的几个 *ulp*：$|a + i \cdot b|$。当将下溢刷新为零时，就不会得出这样的结论。

这种计算复数商的算法非常强大，并且适用于渐进下溢时的误差分析。要在 Store 0 的情况下找到同样强大、益于分析且有效的计算复数商的算法**是不可能的**。在 Store 0 的情况下，考虑低级、复杂细节的麻烦从浮点环境的实现工具转移到了用户那里。

利用渐进下溢成功解决但利用 Store 0 却失败的问题类要比支持使用 Store 0 的人想像的多。许多常用的数学技术都无法成功解决此类的问题：

- 线性方程求解
- 多项式求解
- 数值积分
- 收敛速度
- 复数除法

2.3.6 下溢有问题吗？

尽管举了以上示例，但下溢算法基本上没有任何问题，因此，我们有什么理由不使用它呢？事实上，这一观点是不言而喻的。

如果没有渐进下溢，用户程序需要对隐含的准确度阈值特别敏感。例如，在单精度计算中，如果计算的某些部分发生了下溢，在使用 `Store 0` 的情况下，会将下溢结果替换为 `0`，则所能保证的准确度只能达到大约 10^{-31}，而不是 10^{-38}，即单精度指数的一般下限。

这意味着，编程人员需要在接近不准确阈值时实施自己的检测方法，否则，将不得不放弃寻求强大、稳定的实施其算法的努力。

我们可以缩放某些算法，这样，在接近零的限定区域将不会执行计算。不过，对于所有数学计算程序来说，缩放算法和检测不准确阈值都是困难且耗时的。

第3章

数学库

本章介绍了 Solaris OS 和 Sun Studio 软件中提供的数学库。除了列出每个库及其内容外，本章还介绍了由编译器集合提供的数学库所支持的一些功能，包括 IEEE 支持函数、随机数生成器以及在 IEEE 和非 IEEE 格式之间转换数据的函数。

`Intro`(3M) 手册页上还列出了 `libm` 和 `libsunmath` 库的内容。

3.1 Solaris 数学库

本节介绍捆绑于 Solaris 10 OS 的数学库。这些库是作为共享对象提供的，安装在 Solaris 库的标准位置。

3.1.1 标准数学库

Solaris 标准数学库 `libm` 中包含基本数学函数以及 Solaris 操作环境遵循的各种标准所需的支持例程。

Solaris 10 操作系统包括两种版本的 `libm`：`libm.so.1` 和 `libm.so.2`。`libm.so.1` 提供 Solaris 9 操作系统及早期版本支持的标准所需的函数。`libm.so.2` 提供 Solaris 10 操作系统（包括 C99）支持的标准所需的函数。`libm.so.1` 用于提供向后兼容性，以保证在 Solaris 9 操作系统及早期系统上编译和链接的程序不做改动即可继续运行。`libm.so.1` 中的内容在这些系统的 3M 手册页节进行了说明。本章的剩余部分介绍 `libm.so.2` 。有关动态链接以及确定在运行程序时加载哪些共享对象的选项和环境变量的更多信息，请参见 `ld`(1) 和编译器手册页。

表 3-1 列出了 libm 中的函数。对于每个数学函数，表中只给出了函数的双精度版本名称。库中还包含函数的单精度版本，采用同样的名称，后跟一个 f；还有扩展精度 / 四倍精度版本，也采用同样的名称，后跟一个 l。

表 3-1 libm 的内容

类型	函数名
代数函数	cbrt、fdim、fma、fmax、fmin、hypot、sqrt
初等超越函数	asin、acos、atan、atan2、asinh、acosh、atanh、exp、exp2、expm1、pow、log、log1p、log10、log2、sin、cos、sincos、tan、sinh、cosh、tanh
高等超越函数	j0、j1、jn、y0、y1、yn、erf、erfc、gamma、lgamma、gamma_r、lgamma_r、tgamma
取整函数	ceil、floor、llrint、llround、lrint、lround、modf、nearbyint、rint、round、trunc
IEEE 标准推荐的函数	copysign、fmod、ilogb、nextafter、remainder、scalbn、fabs
IEEE 分类函数	isnan
旧式浮点函数	frexp、ldexp、logb、scalb、significand
错误处理例程（用户定义）	matherr
复数函数	cabs、cacos、cacosh、carg、casin、casinh、catan、catanh、ccos、ccosh、cexp、cimag、clog、conj、cpow、cproj、creal、csin、csinh、csqrt、ctan、ctanh
C99 浮点环境函数	feclearexcept、fegetenv、fegetexceptflag、fegetprec、fegetround、feholdexcept、feraiseexcept、fesetenv、fesetexceptflag、fesetprec、fesetround、fetestexcept、feupdateenv
浮点异常处理函数	fex_getexcepthandler、fex_get_handling、fex_get_log、fex_get_log_depth、fex_log_entry、fex_merge_flags、fex_setexcepthandler、fex_set_handling、fex_set_log、fex_set_log_depth
其他 C99 函数	nan、nexttoward、remquo、scalbln

表 3-1 的说明：

1. 函数 gamma_r 和 lgamma_r 是 gamma 和 lgamma 的可重入版本。

2. 函数 fegetprec 和 fesetprec 只能在 x86 系统上使用。这些函数并非 C99 标准所规定的函数。

3. libm 中的超越函数的误差界限和观察误差均在 libm(3LIB) 手册页列出。

3.1.2 矢量数学库

库 libmvec 提供计算整个参数矢量的常用数学函数的例程。应用程序可能显式地调用 libmvec 中的例程，或者，当使用了 -xvector 标志时，编译器也会调用这些例程。

libmvec 是作为主共享对象 libmvec.so.1 实现的，多种辅助共享对象提供部分或全部矢量函数的替代版本。运行使用 libmvec 链接的程序时，运行时链接程序自动选择能在主机平台上提供最佳性能的版本。因此，使用 libmvec 中的函数的程序运行在不同的系统上时，产生的结果可能略有不同。

表 3-2 列出了 libmvec 中的函数。

表 3-2 libmvec 的内容

类型	函数名
代数函数	vhypot_、vhypotf_、vrhypot_、vrhypotf_、vrsqrt_、vrsqrtf_、vsqrt_、vsqrtf_
指数及相关函数	vexp_、vexpf_、vlog_、vlogf_、vpow_、vpowf_
三角函数	vatan_、vatanf_、vatan2_、vatan2f_、vcos_、vcosf_、vsin_、vsinf_、vsincos_、vsincosf_
复数函数	vc_abs_、vc_exp_、vc_log_、vc_pow_、vz_abs_、vz_exp_、vz_log_、vz_pow_

3.2 Sun Studio 数学库

本节介绍 Sun Studio 10 编译器所包括的数学库。除非特别说明，这些库都作为静态归档文件提供。默认情况下，它们安装在以下目录中：

```
/opt/SUNWspro/prod/lib/
```

针对处理器实现特定指令集变体进行优化的库安装在以下目录的子目录下：

```
/opt/SUNWspro/prod/lib/<arch>/
```

此处的 <*arch*> 是指令集变体的名称。在基于 SPARC 的系统上，这些目录包括 v8、v8a、v8plus、v8plusa、v8plusb、v9、v9a、和 v9b。在 x86 系统上，这些目录包括 386 和 amd64。

目录 /opt/SUNWspro/lib/ 包含指向所有 Sun Studio 数学库（作为共享对象提供）的符号链接。

Sun Studio 数学库的头文件安装在 /opt/SUNWspro/prod/include/ 目录及其子目录中。

3.2.1 Sun 数学库

libsunmath 数学库中包含未被任何标准指定但在数值软件中很实用的函数。它还包含许多 libm.so.2 提供但 libm.so.1 不提供的函数。 libsunmath 同时作为共享对象和静态归档提供。

表 3-3 列出了在 libsunmath 中提供，而在 libm.so.2 中未提供的 函数。对于每个数学函数，表中只给出函数从 C 程序中调用时所使用的双精度版本的名称。 .

表 3-3 libsunmath 的内容

类型	函数名
初等超越函数	exp10
以度为单位的三角函数	asind、acosd、atand、atan2d、sind、cosd、sincosd、tand
使用 π 缩放的三角函数	asinpi、acospi、atanpi、atan2pi、sinpi、cospi、sincospi、tanpi
使用双精度 π 参数缩小的三角函数	asinp、acosp、atanp、sinp、cosp、sincosp、tanp
财务函数	annuity、compound
取整函数	aint、anint、irint、nint
IEEE 标准推荐的函数	signbit
IEEE 分类函数	fp_class、isinf、isnormal、issubnormal、iszero
提供有用 IEEE 值的函数	min_subnormal、max_subnormal、min_normal、max_normal、infinity、signaling_nan、quiet_nan
加法的随机数生成器	i_addran_、i_addrans_、i_init_addrans_、i_get_addrans_、i_set_addrans_、r_addran_、r_addrans_、r_init_addrans_、r_get_addrans_、r_set_addrans_、d_addran_、d_addrans_、d_init_addrans_、d_get_addrans_、d_set_addrans_、u_addrans_
线性同余随机数生成器	i_lcran_、i_lcrans_、i_init_lcrans_、i_get_lcrans_、i_set_lcrans_、r_lcran_、r_lcrans_、d_lcran_、d_lcrans_、u_lcrans_

表 3-3 libsunmath 的内容 (续)

类型	函数名
与进位相乘的随机数生成器	i_mwcran_、i_mwcrans_、i_init_mwcrans_、i_get_mwcrans_、i_set_mwcrans、i_lmwcran_、i_lmwcrans_、i_llmwcran_、i_llmwcrans_、u_mwcran_、u_mwcrans_、u_lmwcran_、u_lmwcrans、u_llmwcran_、u_llmwcrans_、r_mwcran_、r_mwcrans_、d_mwcran_、d_mwcrans_、smwcran_
随机数混洗器	i_shufrans_、r_shufrans_、d_shufrans_、u_shufrans_
数据转换	convert_external
控制舍入模式和浮点异常标记	ieee_flags
浮点捕获处理	ieee_handler、sigfpe
显示状态	ieee_retrospective
启用 / 禁用非标准的算法	standard_arithmetic、nonstandard_arithmetic

3.2.2 优化库

libmopt 库提供了 libm 和 libsunmath 中一些函数的更新版本。libmopt 仅作为静态归档提供。libmopt 中包含的例程替代了 libm 中的相应例程。通常，libmopt 版本的速度明显更快。与 libm 版本（支持任何 ANSI/POSIX、SVID、X/Open 或 C99/IEEE 风格的异常情况处理方式）不同，libmopt 例程仅支持 C99/IEEE 风格的异常情况处理方式。（请参见附录 E）另外，无论采用何种浮点舍入方向模式， libm 中的所有数学函数均可提供相当准确的结果，而未定义调用 libmopt 中带有舍入方向而非舍入为最接近的函数的结果。无论何时调用标准数学函数，使用 libmopt 的程序必须确保默认的舍入为最接近模式生效。要使用 libmopt 链接程序，请使用 -xlibmopt 标志。

在基于 SPARC 的系统上，库 libcx 中包含比 libc 中的 128 位四倍精度浮点算术支持例程速度略快一些的版本。这些例程不直接由用户调用。在对四倍精度 （long double 或 REAL*16）数据执行运算的程序中，编译器将用到它们。libcx 同时作为共享对象和静态归档提供。

libcx 中的四倍精度支持例程几乎与 libc 中的完全相同。 libcx 版本针对特定的指令集变体进行了优化。假如使用适当的 libcx 变体进行了链接，大量使用四倍精度的程序可能会运行得略微快一些。要使用 libcx 链接程序，请使用 -lcx 标志，并使用 -xarch 标志指定要使用的指令集变体。

此外，还提供了共享版本的 libcx（称为 libcx.so.1）。可通过将环境变量 LD_PRELOAD 设置为 libcx.so.1 文件的全路径名，在运行时预加载此版本。要获得最佳的性能，请将相应版本的 libcx.so.1 用于系统的体系结构。例如，在基于 UltraSPARC？ 的系统上 （假定将该库安装到默认位置），将 LD_PRELOAD 设置如下：

csh：

```
setenv LD_PRELOAD /opt/SUNWspro/lib/v8plus/libcx.so.1
```

sh：

```
LD_PRELOAD=/opt/SUNWspro/lib/v8plus/libcx.so.1
export LD_PRELOAD
```

3.2.3 矢量数学库 （仅 SPARC）

在基于 SPARC 的系统上，Sun Studio 数学库包括 `libmvec` 的两个静态归档文件版本。这些库提供的函数与 Solaris `libmvec` 相同。提供静态归档文件库的目的是为了保证用矢量函数的应用程序能够在运行 Solaris 9 或早期操作环境的系统上运行。对于只需在 Solaris 10 系统上运行的应用程序，则应该使用 Solaris `libmvec`。

`libmvec.a` 提供与 Solaris `libmvec.so.1` 中完全相同的单线程矢量函数。要使用 `libmvec.a` 进行链接，请使用 `-lmvec` 标志。`libmvec_mt.a` 提供依赖多处理器并行化的矢量函数的多线程版本。要使用 `libmvec_mt.a`，必须同时链接 `-xparallel` 和 `-lmvec_mt` 。

有关更多信息，请参见 `libmvec`(3m) 和 `clibmvec`(3m) 手册页。

3.2.4 `libm9x` 数学库

`libm9x` 数学库中包含 C99 `<fenv.h>` 浮点环境函数及增强函数，以支持改进的浮点异常处理。在 Solaris 10 操作系统中，`libm9x` 的内容已被归入到 `libm` 中。但仍为运行在早期 Solaris OS 版本上的应用程序提供了 `libm9x`。对于只需在 Solaris 10 系统上运行的应用程序，则应使用 `libm`。

3.3 单、双和扩展精度 / 四倍精度

大多数数值函数可以使用单精度、双精度、扩展 (x86) 精度或四倍精度 (SPARC)。表 3-4 中给出了从不同语言调用各种函数的不同精度版本的示例。

表 3-4 调用单、双和扩展 / 四倍精度函数

语言	单精度	双精度	扩展精度 / 四倍精度

C、C++	`#include <math.h>`
	`float x,y,z;`
	`x = sinf(y);`
	`x = fmodf(y,z);`
	
	`#include <sunmath.h>`
	`float x;`
	`x = max_normalf();`
	`x = r_addran_();`
		`#include <math.h>`
		`double x,y,z;`
		`x = sin(y);`
		`x = fmod(y,z);`
		
		`#include <sunmath.h>`
		`double x;`
		`x = max_normal();`
		`x = d_addran_();`
			`#include <math.h>`
			`long double x,y,z;`
			`x = sinl(y);`
			`x = fmodl(y,z);`
			
			`#include <sunmath.h>`
			`long double x;`
			`x = max_normall();`

Fortran	`REAL x,y,z`
	`x = sin(y)`
	`x = r_fmod(y,z)`
	`x = r_max_normal()`
	`x = r_addran()`
		`REAL*8 x,y,z`
		`x = sin(y)`
		`x = d_fmod(y,z)`
		`x = d_max_normal()`
		`x = d_addran()`
			`REAL*16 x,y,z`
			`x = sin(y)`
			`x = q_fmod(y,z)`
			`x = q_max_normal()`

在 C 中，单精度函数的名称是通过将 `f` 附加到双精度名称的后面形成的；扩展精度或四倍精度函数的名称是通过添加 `l` 形成的。因为 Fortran 的调用惯例不同，所以 `libsunmath` 为单、双和四倍精度分别提供 `r_…`、`_d_…`、和 `q_…` 函数。可以按所有三种精度的通用名来调用 Fortran 内函数。

并非所有的函数都有 `q_…` 版本。有关 `libm` 和 `libsunmath` 函数的名称和定义，请参见 `math.h` 和 `sunmath.h`。

在 Fortran 程序中，切记将 `r_…` 函数声明为 `real` ；将 `d_…` 函数声明为双精度，而将 `q_…` 函数声明为 `REAL*16`。否则，可能会导致类型不匹配。

注 – Sun Studio Fortran (x86) 既不支持扩展双精度，也不支持四倍精度。

3.4 IEEE 支持函数

本节介绍 IEEE 推荐的函数，这些函数提供有用的值 ieee_flags、ieee_retrospective 以及 standard_arithmetic 和 nonstandard_arithmetic。有关函数 ieee_flags 和 ieee_handler 的详细信息，请参见第 4 章。

3.4.1 ieee_functions(3m) 和 ieee_sun(3m)

ieee_functions(3m) 和 ieee_sun(3m) 描述的函数提供 IEEE 标准要求的功能或其附录中建议的功能。这些函数是按有效的位掩码运算实现的。

表 3-5 ieee_functions(3m)

功能	描述
math.h	头文件
copysign(x,y)	x 及 y 的符号位
fabs(x)	x 的绝对值
fmod(x, y)	x 除以 y 所得的余数
ilogb(x)	以 2 为基数的整数格式 x 无偏置指数
nextafter(x,y)	在 y 方向上， x 后面的下一个可被代表的数
remainder(x,y)	x 除以 y 所得的余数
scalbn(x,n)	$x \times 2^n$

表 3-6 ieee_sun(3m)

功能	描述
sunmath.h	头文件
fp_class(x)	分类函数
isinf(x)	分类函数
isnormal(x)	分类函数
issubnormal(x)	分类函数
iszero(x)	分类函数
signbit(x)	分类函数

表 3-6 ieee_sun(3m)(续)

功能	描述
nonstandard_arithmetic(void)	切换硬件
standard_arithmetic(void)	切换硬件
ieee_retrospective(*f)	

remainder(x,y) 是 IEEE 标准 754-1985 中指定的运算。remainder(x,y) 和 fmod(x,y) 的不同之处在于，remainder(x,y) 返回结果的符号可能与 x 或 y 的符号不一致，而 fmod(x,y) 返回结果的符号始终与 x 一致。这两个函数均返回精确的结果，并且不生成不精确的异常。

表 3-7 从 Fortran 中调用 ieee_functions

IEEE 函数	单精度	双精度	四倍精度
copysign(x,y)	t=r_copysign(x,y)	z=d_copysign(x,y)	z=q_copysign(x,y)
ilogb(x)	i=ir_ilogb(x)	i=id_ilogb(x)	i=iq_ilogb(x)
nextafter(x,y)	t=r_nextafter(x,y)	z=d_nextafter(x,y)	z=q_nextafter(x,y)
scalbn(x,n)	t=r_scalbn(x,n)	z=d_scalbn(x,n)	z=q_scalbn(x,n)
signbit(x)	i=ir_signbit(x)	i=id_signbit(x)	i=iq_signbit(x)

表 3-8 从 Fortran 中调用 ieee_sun

IEEE 函数	单精度	双精度	四倍精度
signbit(x)	i=ir_signbit(x)	i=id_signbit(x)	i=iq_signbit(x)

注 – 在使用 d_*function* 和 q_*function* 的 Fortran 程序中，必须将前一个函数声明为双精度，而将后一个函数声明为 REAL*16。

3.4.2 ieee_values(3m)

无穷大、 NaN、最大和最小正浮点数等 IEEE 值是由 ieee_values(3m) 手册页描述的函数提供的。表 3-9、表 3-10、表 3-11 和表 3-12 显示 ieee_values(3m) 函数提供的值的十进制和十六进制 IEEE 表示。

表 3-9 IEEE 值：单精度

IEEE 值	十进制值 十六进制表示	C、C++ Fortran
最大正规数	3.40282347e+38 7f7fffff	r = max_normalf(); r = r_max_normal()
最小正规数	1.17549435e-38 00800000	r = min_normalf(); r = r_min_normal()
最大次正规数	1.17549421e-38 007fffff	r = max_subnormalf(); r = r_max_subnormal()
最小非正规数	1.40129846e-45 00000001	r = min_subnormalf(); r = r_min_subnormal()
∞	无穷 7f800000	r = infinityf(); r = r_infinity()
静态 NaN	NaN 7fffffff	r = quiet_nanf(0); r = r_quiet_nan(0)
信号 NaN	NaN 7f800001	r = signaling_nanf(0); r = r_signaling_nan(0)

表 3-10 IEEE 值：双精度

IEEE 值	十进制值 十六进制表示	C、C++ Fortran
最大正规数	1.7976931348623157e+308 7feffffff fffffffff	d = max_normal(); d = d_max_normal()
最小正规数	2.2250738585072014e-308 00100000 00000000	d = min_normal(); d = d_min_normal()
最大次正规数	2.2250738585072009e-308 000fffff ffffffff	d = max_subnormal(); d = d_max_subnormal()
最小非正规数	4.9406564584124654e-324 00000000 00000001	d = min_subnormal(); d = d_min_subnormal()

表 3-10 IEEE 值：双精度 (续)

IEEE 值	十进制值 十六进制表示	C、C++ Fortran
∞	Infinity 7ff00000 00000000	d = infinity(); d = d_infinity()
静态 NaN	NaN 7fffffff ffffffff	d = quiet_nan(0); d = d_quiet_nan(0)
信号 NaN	NaN 7ff00000 00000001	d = signaling_nan(0); d = d_signaling_nan(0)

表 3-11 IEEE 值：四倍精度 (SPARC)

IEEE 值	十进制值 十六进制表示	C、C++ Fortran
最大正规数	1.189731495357231765085759326628007 0e+4932 7ffeffff ffffffff ffffffff ffffffff	q = max_normall(); q = q_max_normal()
最小正规数	3.362103143112093506262677817321752 6e-4932 00010000 00000000 00000000 00000000	q = min_normall(); q = q_min_normal()
最大次正规数	3.362103143112093506262677817321752 0e-4932 0000ffff ffffffff ffffffff ffffffff	q = max_subnormall(); q = q_max_subnormal()
最小非正规数	6.475175119438025110924438958227646 6e-4966 00000000 00000000 00000000 00000001	q = min_subnormall(); q = q_min_subnormal()
∞	Infinity 7fff0000 00000000 00000000 00000000	q = infinityl(); q = q_infinity()
静态 NaN	NaN 7fff8000 00000000 00000000 00000000	q = quiet_nanl(0); q = q_quiet_nan(0)
信号 NaN	NaN 7fff0000 00000000 00000000 00000001	q = signaling_nanl(0); q = q_signaling_nan(0)

表 3-12 IEEE 值：双扩展精度 (x86)

IEEE 值	十进制值 十六进制表示 （80 位）	C、C++
最大正规数	1.189731495357231765 05e+4932 7ffe ffffffff ffffffff	x = max_normall();
正最小正规数	3.362103143112093506 26e-4932 0001 80000000 00000000	x = min_normall();
最大次正规数	3.362103143112093506 08e-4932 0000 7fffffff ffffffff	x = max_subnormall();

表 3-12 IEEE 值：双扩展精度 (x86)(续)

IEEE 值	十进制值 十六进制表示 （80 位）	C、C++
最小正次正规数	1.8225997659412373012 6e-4951 0000 00000000 00000001	x = min_subnormall();
∞	Infinity 7fff 80000000 00000000	x = infinityl();
静态 NaN	NaN 7fff c0000000 00000000	x = q
信号 NaN	NaN 7fff 80000000 00000001	x = signaling_nanl(0);

3.4.3 ieee_flags(3m)

ieee_flags (3m) 是用于以下功能的 Sun 接口：

- 查询或设置舍入方向模式
- 查询或设置舍入精度模式
- 检查、清除或设置产生的异常标记

调用 ieee_flags(3m) 的语法为：

```
i = ieee_flags(action, mode, in, out);
```

可能作为参数值的 ASCII 字符串如表 3-13 所示：

表 3-13 ieee_flags 的参数值

参数	C 或 C++ 类型	所有可能的值
action	char*	get、 set、 clear、 clearall
模式	char*	direction、 precision、 exception
in	char*	nearest、tozero、negative、positive、extended、double、single、inexact、division、underflow、overflow、 invalid、 all、 common
out	char **	nearest、tozero、negative、positive、extended、double、single、inexact、division、underflow、overflow、 invalid、 all、 common

ieee_flags(3m) 手册页详细介绍了这些参数。

以下段落介绍了可使用 ieee_flags 修改的某些算法功能。第 4 章中包含有关 ieee_flags 和 IEEE 异常标记的更多信息。

如果 *mode* 为 direction，则指定操作适用于当前的舍入方向。可能的舍入方向为：向最接近的值舍入、向零舍入、向 + 舍入或向 – 舍入。IEEE 默认舍入方向为**向最接近的值舍入**。这意味着，如果运算的数学结果恰好位于两个相邻的可被代表的数字之间，则提供最接近数学结果的数字。（如果数学结果恰好位于两个最接近的可被代表的数字中间，会将最低有效位为零的数字作为结果提供。有时，将**向最接近的值舍入**模式称为**向最接近的偶数舍入**以进行强调。）

向零舍入是 IEEE 之前的很多计算机的工作方式，并以数学方式与截断结果一致。例如，如果将 2/3 舍入到 6 个十进制数字，则当舍入模式为向最接近的值舍入时结果为 .666667，当舍入模式为向零舍入时结果为 .666666。

在使用 ieee_flags 检查、清除或设置舍入方向时，四个可能的输入参数值如表 3-14 所示。

表 3-14 ieee_flags 舍入方向的输入值

参数	**可能的值 （mode 为** direction**）**
action	get、 set、 clear、 clearall
in	nearest， tozero， negative， positive
out	nearest， tozero， negative， positive

如果 *mode* 为 precision，则指定的操作适用于当前的舍入精度。在基于 x86 的系统上，可能的舍入精度为：单、双和扩展。默认的舍入精度为扩展；在这种模式下，提供 x87 浮点寄存器结果的算术运算将其结果舍入到使用扩展双寄存器格式的完整 64 位精度。当舍入精度为单或双精度时，提供 x87 浮点寄存器结果的算术运算将其结果分别舍入到 24 或 53 个有效位。虽然大多数程序在使用扩展舍入精度时生成很准确的结果 （即便不是最准确的结果），但某些要求严格遵循 IEEE 算法语义的程序在扩展舍入精度模式下无法正常工作，必须将舍入精度相应地设置为单精度或双精度以运行这些程序。

在使用 SPARC 处理器的系统上不能设置舍入精度。在这些系统上，使用 *mode* = precision 调用 ieee_flags 对计算没有影响。

最后，如果 *mode* 为 exception，则指定的操作适用于当前的 IEEE 异常标记。有关使用 ieee_flags 检查和控制 IEEE 异常标记的详细信息，请参见第 4 章。

3.4.4 ieee_retrospective(3m)

libsunmath 函数 ieee_retrospective 打印有关未处理异常和非标准 IEEE 模式的信息。它报告：

- 未处理异常。
- 启用的陷阱。
- 如果将舍入方向或精度设置为非默认值。
- 如果非标准算法生效的话。

从硬件浮点状态寄存器获得所需的信息。

ieee_retrospective 打印有关*引发*的异常标记以及启用**陷阱**的异常的信息。不应混淆这两种截然不同的信息（即使它们有关联也是如此）。如果引发异常标记，则该异常是在执行程序过程中某一时刻发生的。如果为异常启用了陷阱，则异常可能并没有实际发生（但如果发生了，会提供 SIGFPE 信号）。ieee_retrospective 消息用于提示您有关可能需要调查的异常的信息（如果**引发**了异常标记），或者提示您信号处理程序可能已处理了异常 （如果启用了异常的**陷阱**的话）。第 4 章讨论了异常、信号和陷阱，并说明如何调查引发异常的原因。

程序可以随时显式地调用 ieee_retrospective。使用 f95 在 -f77 兼容性模式下编译的 Fortran 程序在退出之前自动调用 ieee_retrospective。使用 f95 在默认模式下编译的 C/C++ 程序和 Fortran 程序则不会自动调用 ieee_retrospective。

注意，在默认情况下， f95 编译器启用对常见异常的捕获，因此，除非程序显式地禁用捕获或者安装了 SIGFPE 处理程序，否则编译器在出现异常时立即终止。在 -f77 兼容性模式下，编译器并不启用捕获，因此当发生浮点异常时，程序继续执行，并在退出时通过 ieee_retrospective 输出报告这些异常。

调用此函数的语法为：

C、C++ `ieee_retrospective(`*fp*`);`

Fortran `call ieee_retrospective()`

对于 C 函数，参数 *fp* 指定写入输出的文件。 Fortran 函数始终在 stderr 上打印输出。

以下示例显示 6 个 ieee_retrospective 警告消息中的 4 个：

```
注意：引发了 IEEE 浮点异常标记：
    不精确；下溢；
舍入方向为向零舍入
启用了 IEEE 浮点异常陷阱：
    溢出；
请参见数字计算指南 ieee_flags(3M)、
    ieee_handler(3M), ieee_sun(3m)
```

只有在启用捕获或引发异常时，才会出现警告消息。

可以在 Fortran 程序中使用三种方法之一来禁止 ieee_retrospective 消息。一种方法是清除所有未处理的异常，禁用陷阱，并在程序退出前恢复舍入到最接近的值、扩展精度和标准模式。为此，请按如下方式调用 ieee_flags、 ieee_handler 和 standard_arithmetic：

```
character*8 out
i = ieee_flags('clearall', '', '', out)
call ieee_handler('clear', 'all', 0)
call standard_arithmetic()
```

注 – 建议不要不调查其原因而清除未处理的异常。

另一种避免看到 `ieee_retrospective` 消息的方法是将 `stderr` 重定向到某个文件。当然，如果程序还将其他非 `ieee_retrospective` 消息的输出发送到 `stderr`，则不应使用这种方法。

第三种方法是在程序中包含伪 `ieee_retrospective` 函数，例如：

```
subroutine ieee_retrospective
return
end
```

3.4.5 nonstandard_arithmetic(3m)

正如第 2 章中所介绍的一样，IEEE 算法使用渐进下溢来处理出现下溢的结果。在某些基于 SPARC 的系统上，渐进下溢通常是使用算法的软件仿真部分实现的。如果很多计算出现下溢，则其性能可能会下降。

要获取某些有关特定程序中是否出现这种情况的信息，可以使用 `ieee_retrospective` 或 `ieee_flags` 来确定是否发生了下溢异常，并检查程序使用的系统时间量。如果操作系统中的程序使用非常多的时间并且引发了下溢异常，则渐进下溢可能是问题的原因。在这种情况下，使用非 IEEE 算法可以加快程序的执行速度。

函数 `nonstandard_arithmetic` 在支持非 IEEE 算法模式的处理器上启用这些模式。在 SPARC 系统上，该函数在浮点状态寄存器中设置 NS（nonstandard arithmetic, 非标准算法）位。在支持 SSE 指令的 x86 系统上，该函数在 MXCSR 寄存器中设置 FTZ（flush to zero, 刷新为零）位；它还在支持 DAZ （denormals are zero, 反向规格数为零）位的处理器上的 MXCSR 寄存器中设置该位。注意，非标准模式的作用因处理器而异，并且可能会导致可靠软件出错。建议不要使用非标准模式以确保正常使用。

函数 `standard_arithmetic` 将硬件重置为使用默认 IEEE 算法。这两种函数对只提供默认 IEEE 754 风格的算法的处理器没有影响， SuperSPARC® 就是一种此类处理器。

3.5 C99 浮点环境函数

本节介绍 C99 中的 <fenv.h> 浮点环境函数。在 Solaris 10 操作系统中，这些函数位于 `libm` 中。它们提供很多与 `ieee_flags` 函数相同的功能，但它们使用更普通的 C 接口；因为它们是由 C99 定义的，所以它们具有更大的可移植性。

注 – 为保持行为的一致，请不要在同一程序中同时使用 `libm` 中的 C99 浮点环境函数和异常处理扩展以及 `libsunmath` 中的 `ieee_flags` 和 `ieee_handler` 函数。

3.5.1 异常标记函数

`fenv.h` 文件为 5 个 IEEE 浮点异常标记中的每一个标记定义了宏：`FE_INEXACT`、`FE_UNDERFLOW`、`FE_OVERFLOW`、`FE_DIVBYZERO` 和 `FE_INVALID`。此外，将宏 `FE_ALL_EXCEPT` 定义为所有 5 个标记宏进行按位“或”运算。在以下说明中，*excepts* 参数可以是 5 个标记宏中任何一个宏的按位“或”运算或者值 `FE_ALL_EXCEPT`。对于 `fegetexceptflag` 和 `fesetexceptflag` 函数，*flagp* 参数必须是指向类型为 `fexcept_t` 的对象的指针。（这种类型是在 `fenv.h` 中定义的。）

C99 定义了以下异常标记函数：

表 3-15 C99 标准异常标记函数

功能	操作
`feclearexcept`(*excepts*)	清除指定的标记
`fetestexcept`(*excepts*)	返回指定标记的设置
`feraiseexcept`(*excepts*)	引发指定的异常
`fegetexceptflag`(*flagp, excepts*)	在 *flagp 中保存指定的标记
`fesetexceptflag`(*flagp, excepts*)	从 *flagp 中恢复指定的标记

`feclearexcept` 函数清除指定的标记。 fetestexcept 函数返回设置的 *excepts* 参数指定的标记子集对应的宏值的按位“或”运算结果。例如，如果只有当前设置的标记不准确、下溢或被零除，则

```
i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
```

将 `i` 设置为 `FE_DIVBYZERO`。

如果启用任何指定异常的陷阱，则 `feraiseexcept` 函数导致一个陷阱。（有关异常陷阱的详细信息，请参见第 4 章。）否则，它只设置相应的标记。

`fegetexceptflag` 和 `fesetexceptflag` 函数提供一种简便的方法暂时保存某些标记的状态，并在以后恢复它们。特别地， `fesetexceptflag` 函数并不导致陷阱；它只恢复指定标记的值。

3.5.2 舍入控制

fenv.h 文件为 4 个 IEEE 舍入方向模式中的每一个模式定义了宏：FE_TONEAREST、FE_UPWARD（向正无穷大）、FE_DOWNWARD（向负无穷大）和 FE_TOWARDZERO。C99 定义了两个函数以控制舍入方向模式：fesetround 将当前的舍入方向设置为其参数指定的方向 （必须是以上 4 个宏之一），而 fegetround 返回对应于当前舍入方向的宏值。

在基于 x86 的系统上， fenv.h 文件为 3 个舍入精度模式中的每一个模式定义宏。FE_FLTPREC（单精度）、FE_DBLPREC（双精度）和 FE_LDBLPREC（扩展双精度）。虽然它们不是 C99 的一部分，但 x86 上的 libm 提供两个函数来控制舍入精度模式：fesetprec 将当前的舍入精度设置为其参数指定的精度（必须是以上 3 个宏之一），而 fegetprec 返回对应于当前舍入精度的宏值。

3.5.3 环境函数

fenv.h 文件定义数据类型 fenv_t，它表示整个浮点环境，包括异常标记、异常控制模式、异常处理模式和 （在 SPARC 上的）非标准模式。在以下说明中， *envp* 参数必须是指向类型为 fenv_t 的对象的指针。

C99 定义 4 个函数以处理浮点环境。 libm 提供的一个附加函数在多线程程序中非常有用。下表概要介绍了这些函数：

表 3-16 libm 浮点环境函数

功能	操作
fegetenv(*envp*)	将环境保存在 *envp 中
fesetenv(*envp*)	从 *envp 中恢复环境
feholdexcept(*envp*)	将环境保存在 *envp 中，并建立不间断的模式
feupdateenv(*envp*)	从 *envp 中恢复环境并引发异常
fex_merge_flags(*envp*)	*envp 中的 “或” 异常标记

fegetenv 和 fesetenv 函数分别用来保存和恢复浮点环境。fesetenv 的参数可以是指向以前通过调用 fegetenv 或 feholdexcept 保存的环境的指针，或者在 fenv.h 中定义的常量 FE_DFL_ENV。后者表示默认环境，即清除所有异常标记、舍入到最接近的值（在基于 x86 的系统上舍入到扩展双精度）、不间断的异常处理模式（即禁用陷阱）以及 （在基于 SPARC 的系统上）禁用非标准的模式。

feholdexcept 函数保存当前的环境，然后清除所有异常标记并为所有异常建立不间断的异常处理模式。 feupdateenv 函数恢复保存的环境（可通过调用 fegetenv 或 feholdexcept 或常量 FE_DFL_ENV），然后引发在先前环境中设置其标记的异常。如果恢复的环境为其中的任何异常启用了陷阱，则会发生异常；否则，设置这些标记。可结合使用这两个函数，使子例程调用相对于异常为原子操作 （如以下代码示例所示）：

```
#include <fenv.h>

void myfunc(...) {
    fenv_t env;

    /* 保存环境、清除标记并禁用陷阱 */
    feholdexcept(&env);
    /* 执行可能导致异常的计算 */
    ...
    /* 检查假异常 */
    if (fetestexcept(...)) {
        /* 对它们进行相应的处理并清除其标记 */
        ...
        feclearexcept(...);
    }
    /* 恢复环境并引发相关的异常 */
    feupdateenv(&env);
}
```

fex_merge_flags 函数对当前环境和保存环境中的异常标记执行逻辑 OR 运算，而不调用任何陷阱。可以在多线程程序中使用此函数，在父线程中保存有关子线程中的计算引发的标记的信息。有关显示 fex_merge_flags 用法的示例，请参见附录 A。

3.6 libm 和 libsunmath 的实现功能

本节介绍 libm 和 libsunmath 的实现功能：

- 使用无限精确 π 的参数变形以及使用 π 缩放的三角函数。
- 用于在 IEEE 和非 IEEE 格式之间转换浮点数据的数据转换例程。
- 随机数生成器。

3.6.1 关于算法

在基于 SPARC 的系统上， `libm` 和 `libsunmath` 中的初等函数是使用不断变化的表驱动算法和多项式 / 有理数近似算法的组合实现的。在基于 x86 的系统上， `libm` 和 `libsunmath` 中的某些初等函数是使用 x86 指令集中提供的初等函数内核指令实现的；其他函数是用基于 SPARC 的系统上使用的相同表驱动算法或多项式 / 有理数近似算法实现的。

`libm` 中普通初等函数及 `libsunmath` 中普通单精度初等函数的表驱动算法和多项式 / 有理数近似算法都将准确结果传送到最后一位中的单位内 (*ulp*)。在基于 SPARC 的系统上， `libsunmath` 中的普通四倍精度初等函数将精确结果传送到一个 *ulp* 内，但是 `expm1` 和 `log1pl` 函数除外，它们将精确结果传送到两个 *ulps* 内。（普通函数包括指数函数、自然对数函数、幂函数和弧度参数的循环三角函数。其他函数，如双曲三角函数和高等超越函数是不精确的。）通过运算法则直接分析获取这些误差界限。用户通过使用 `BeEF`（Berkeley Elementary Function 测试程序）也能检测这些例程的精确性，可以从 `ucbtest` 软件包中的 `netlib` 获得该程序，网址为 `http://www.netlib.org/fp/ucbtest.tgz`。

3.6.2 三角函数的参数缩小

使用的弧度参数在 [π/4,π/4] 范围之外的三角函数通过将参数缩小到指定范围（减去 π/2 的整数倍）来进行计算。

因为 π 不是能够被机器表示的数字，所以对其进行近似。三角函数的最终计算误差取决于缩小参数造成的舍入误差 （使用近似的 π 值和舍入值）以及计算使用缩小参数的三角函数时的近似误差。即使是非常小的参数，最终结果的相对误差可能取决于该参数的缩小误差；而对于较大的参数，参数缩小造成的误差可能并不比其他误差大。

通常，人们误认为所有使用大参数的三角函数本身准确性很低，而所有小参数的准确性相对高一些。这是基于简单的观察：非常大的可以被机器表示的数字是由大于 π 的距离分隔的。

计算的三角函数值没有准确性突然变差的内在界限，不准确的函数值也没有变得无法使用的内在界限。假设参数始终不断缩小，实际上很难发现使用 π 近似值完成的参数缩小这一情况，这是因为为大参数和小参数都保留了所有基本的恒等式和关系。

`libm` 和 `libsunmath` 三角函数使用“无限”精确的 π 进行参数缩小。将值 2/π 的计算结果取 916 位十六进制数字，并将其存储在查找表中以在参数缩小过程中使用。

`sinpi`、`cospi` 和 `tanpi` 函数组（请参见表 3-3）使用 π 来缩放输入参数，以避免由于范围缩小而产生的误差。

3.6.3 数据转换例程

libm 和 libsunmath 中包含一个灵活的数据转换例程 convert_external，它用于在 IEEE 和非 IEEE 格式之间转换二进制浮点数据。

支持的格式包括 SPARC (IEEE)、IBM PC、VAX、IBM S/370 和 Cray 使用的那些格式。

有关处理 Cray 上生成的数据以及使用函数 convert_external 将数据转换为基于 SPARC 的系统上使用的 IEEE 格式的示例，请参见 convert_external(3m) 上的手册页。

3.6.4 随机数工具

可以使用三种工具来生成 32 位整数、单精度浮点和双精度浮点格式的统一伪随机数：

- addrans(3m) 手册页中描述的函数基于表驱动的加法随机数产生器系列。
- lcrans(3m) 手册页中描述的函数基于线性同余随机数生成器。
- mwcrans(3m) 手册页中描述的函数基于与进位相乘的随机数生成器。这些函数还包括以 64 位整数格式提供统一伪随机数的生成器。

此外， shufrans(3m) 手册页中描述的函数可与其中的任何生成器结合使用以混洗伪随机数的数组，因而为需要它的应用程序提供更大的随机性。（注意，没有用于混洗 64 位整数数组的工具。）

每个随机数工具都包含每次生成一个随机数（即每个函数调用一个随机数）的例程以及在单个调用中生成随机数数组的例程。每次生成一个随机数的函数提供的数字位于 表 3-17 所示的范围内。

表 3-17 单值随机数生成器的区间

功能	下界	上界
i_addran_	-2147483648	2147483647
r_addran_	0	0.9999999403953552246
d_addran_	0	0.9999999999999998890
i_lcran_	1	2147483646
r_lcran_	4.656612873077392578E-10	1
d_lcran_	4.656612875245796923E-10	0.9999999995343387127
i_mwcran_	0	2147483647
u_mwcran_	0	4294967295
i_llmwcran_	0	9223372036854775807

表 3-17 单值随机数生成器的区间 (续)

功能	下界	上界
u_llmwcran_	0	18446744073709551615
r_mwcran_	0	0.9999999403953552246
d_mwcran_	0	0.9999999999999998890

在单个调用中生成整个随机数数组的函数允许用户指定生成数所在的区间。附录 A 给出了一些示例，说明如何生成在不同区间中均匀分布的随机数数组。

注意，addrans 和 mwcrans 生成器通常比 lcrans 生成器更有效，但它们的理论基础还不足够精确。S. Park 和 K. Miller 于 1988 年 10 月在 Communications of the ACM 上发表的 “Random Number Generators: Good Ones Are Hard To Find” 介绍了线性同余算法的原理。Knuth 编写的 The Art of Computer Programming 第 2 卷中讨论了加法的随机数生成器。

第 4 章

异常和异常处理

本章描述 IEEE 浮点异常并展示如何检测、查找和处理它们。

在基于 SPARC® 和 x86 的系统上，由 Sun Studio 编译器和 Solaris 操作系统提供的浮点环境支持 IEEE 标准所需的全部异常处理功能以及许多推荐的可选功能。 IEEE 854 标准（IEEE 854 的第 18 页）对这些功能的某个目标进行了介绍：

> ... 最小化用户针对异常条件而进行的编译。算术系统旨在尽可能长久地进行持续计算，并用合理的默认响应处理异常情况 （包括设置适当的标志）。

为了实现上述目标，在标准中规定了异常运算的默认结果，并要求所实现的方案提供可由用户感应、设置或清除的状态标志，以便指出异常已经发生。这些标准还建议在所实现的方案中为程序提供一种在发生异常时捕获的方法（即，中断正常的控制流）。例如，该程序可以为异常运算提供替换结果并继续执行，从而提供一个用来以适当方式处理异常的捕获处理程序。本章列出由 IEEE 754 定义的异常及其默认结果的列表，并描述可支持状态标志、捕获和异常处理的浮点环境的功能。（本章中的某些信息只适用于 Solaris 10 操作系统，对 Solaris 操作系统的早期版本不适用。）

4.1 何为异常？

很难对异常进行定义。下面引用 W. Kahan 的话：

> 当尝试执行的原子算法运算没有生成可普遍接受的结果时，就产生算术异常。“原子”和 “可接受”两词的含义因时间和地点而异。（请参见 W. Kahan 编写的 《处理算术异常》。）

例如，当某个程序尝试求负数的平方根时，会产生异常。（这是**无效运算**异常的一个示例。）在出现这样的异常时，系统会以以下两种方法之一进行响应：

- 如果该异常的捕获处于禁用状态 （默认设置），则系统记录异常发生这一事实，并使用 IEEE 754 针对异常运算指定的默认结果继续执行该程序。
- 如果该异常的捕获处于启用状态，则系统生成 `SIGFPE` 信号。如果该程序已经安装了 `SIGFPE` 信号处理程序，系统会将控制转交给该处理程序；否则，该程序将终止。

IEEE 754 定义五个基本类型的浮点异常：**无效运算**、**被零除**、**溢出**、**下溢**和**不精确**。前三个（无效、除和溢出）有时统称为**常见异常**。这些异常一旦出现，很少可被忽略。`ieee_handler`(3m) 提供一种仅捕获常见异常的方便方法。另外两种异常（下溢和不精确）更常见－实际上，大多数浮点运算都导致不精确异常－通常（尽管不总是）可以安全地忽略。

表 4-1 包含可在 IEEE 标准 754 中找到的信息。它描述五种浮点异常以及在出现这些异常时 IEEE 算法环境的默认响应。

表 4-1 IEEE 浮点异常

IEEE 异常	出现异常的原因	示例	在捕获被禁用时出现的默认结果

无效运算	对于将要执行的运算，某个操作数无效。
	
	（在 x86 上，当浮点栈下溢或溢出时也会出现该异常，尽管这不符合 IEEE 标准。）
		`0 × ∞`
		`0 / 0`
		`∞ / ∞`
		`x REM 0`
		负数平方根的操作数
		带有信号传输的操作
		NaN 操作数
		无序比较
		（请参见注释 1）
		无效转换（请参见注释 2）
			无噪声 `NaN`

被零除	针对有限操作数执行运算时生成精确的无穷大结果。	`x / 0` 得到有限的非零 `x` `log(0)`	带有正确符号的无穷大
溢出	正确舍入的结果将比可用目标格式表示的最大有限数大很多（即，超过指数范围）。	双精度： `DBL_MAX` + 1.0e294 `exp`(709.8) 单精度： （浮点） `DBL_MAX` `FLT_MAX` + 1.0e32 `expf`(88.8)	根据舍入模式 (RM)，中间结果的符号为： RM+ - RN+∞ -∞ RZ +max -max R - +max - ∞ R+ +∞ - max
下溢	精确结果或正确舍入的结果将比可用目标格式表示的最小正常数小很多（请参见注释 3）。	双精度： `nextafter`(min_normal,-∞) `nextafter`(min_subnormal,-∞) `DBL_MIN`/3.0 `exp`(-708.5) 单精度： （浮点） `DBL_MIN` `nextafterf`(`FLT_MIN`, -∞) `expf`(-87.4)	低于正常值或零

表 4-1 IEEE 浮点异常 (续)

IEEE 异常	出现异常的原因	示例	在捕获被禁用时出现的默认结果
不精确	有效运算的舍入结果不同于无限精确结果。（大多数浮点运算都产生该异常。）	2.0 / 3.0 （浮点） 1.12345678 log(1.1) DBL_MAX + DBL_MAX，当没有溢出捕获时	该运算的结果（舍入、溢出或下溢）

4.1.1 表 4-1 的注释

1. 无序比较：对于任何浮点值对来说，即使它们的格式不同，也可以对它们进行比较。可能有四种互斥关系：小于、大于、等于或无序。无序意味着至少有一个操作数为 NaN （不是数字）。

   每个 NaN 都与任何值（包括它本身）进行“无序”比较。表 4-2 显示在关系为无序时，由哪一种判定会导致无效运算异常。

表 4-2 无序比较

Predicates			无效异常
数学	c、c++	f95	（如果无序的话）
=	==	.EQ.	no
≠	!=	.NE.	no
>	>	.GT.	yes
≥	>=	.GE.	yes
<	<	.LT.	yes
≤	<=	.LE.	yes

2. 无效转换：尝试将 NaN 或无穷大转换为整数，或者在从浮点格式转换时出现整数溢出。
3. IEEE 单精度、双精度和扩展格式能表示的最小正规数分别为 2^{-126}、2^{-1022} 和 2^{-16382}。有关 IEEE 浮点格式的说明，请参见第 2 章。

x86 浮点环境提供另一个未在 IEEE 标准中提到的异常：**非正规操作数**异常。当针对次正规数执行浮点运算时，会出现该异常。

异常的优先顺序如下所示：无效 （优先级最高）、溢出、除、下溢、不精确 （优先级最低）。在基于 x86 的系统上，非正规操作数异常在所有异常中的优先级最低。

能够在单个操作中同时发生的标准异常只有不精确的溢出和不精确的下溢这两种组合。在基于 x86 的系统上，非正规操作数异常可以与五个标准异常中的任意一个同时发生。如果启用了对溢出、下溢和不精确的捕获，则溢出和下溢捕获的优先级高于不精确捕获；在基于 x86 的系统上，溢出、下溢和不精确捕获的优先级都高于非正规操作数捕获。

4.2 检测异常

正如 IEEE 标准所要求的那样，基于 SPARC 和 x86 的系统上的浮点环境提供用来记录所出现的浮点异常的状态标志。程序可通过测试这些标志来确定已发生了哪些异常。这些标志还可以被显式设置和清除。 `ieee_flags` 函数提供一种访问这些标志的方法。在用 C 或 C++ 编写的程序中， C99 浮点环境函数提供另一种方法。

在基于 SPARC 的系统上，每种异常都有两个与之相关的标志：当前和应计。当前异常标志总是指出由上一个完成执行的浮点指令引发的异常。这些标志还累积 （即 “ 或 ”）到应计异常标志中，从而记录自该程序开始执行或者自该程序上次清除应计标志以来已经发生的所有未捕获异常。（当浮点指令导致捕获的异常时，会设置与导致该捕获的异常相对应的当前异常标志，但是不更改应计标志。）当前异常标志和应计异常标志都包含在浮点状态寄存器 `%fsr` 中。

在基于 x86 的系统上，浮点状态字 (SW) 为应计表达式以及浮点栈的状态提供标志。在基于 x86 的支持 SSE2 指令的系统上，MXCSR 寄存器包含记录由那些指令引发的应计异常的标志。

4.2.1 `ieee_flags`(3m)

调用 `ieee_flags`(3m) 的语法为：

```
i = ieee_flags(action, mode, in, out);
```

程序可提供字符串 `"exception"` 并将其作为第二个参数，从而可使用 `ieee_flags` 函数测试、设置或清除应计异常状态的标志。例如，要从 Fortran 中清除溢出异常标志，请编写：

```
      character*8 out
      call ieee_flags('clear', 'exception', 'overflow', out)
```

要确定是否在 C 或 C++ 中发生了异常，请使用：

```
      i = ieee_flags("get", "exception", in, out);
```

当操作为 "get" 时， *out* 中返回的字符串为：

- "not available"— 如果异常信息不可用
- ""（空字符串）— 如果没有应计异常，或者对于 x86，非正规操作数是唯一的应计异常
- 在第三个参数 (*in*) 中命名的异常的名称 — 如果出现了该异常
- 否则，返回已出现的、优先级最高的异常的名称。

例如，在 Fortran 调用中：

```
      character*8 out
      i = ieee_flags('get', 'exception', 'division', out)
```

如果出现了被零除异常，则 out 中返回的字符串为 "division"；否则返回已出现的、优先级最高的异常的名称。请注意，除非 *in* 指定具体的异常，否则它将被忽略；例如，在 C 调用中，将忽略参数 "all"：

```
      i = ieee_flags("get", "exception", "all", out);
```

除了在 *out* 中返回异常的名称以外， ieee_flags 还返回一个结合当前引发的所有异常的整数值。此值是所有应计异常标志的按位“或”，其中的每个标志都用一位表示，如表 4-3 中所示。与每个异常相对应的位的位置由 fp_exception_type 值 （在 sys/ieeefp.h 文件中定义）给出。（请注意，这些位的位置与机器有关且不必连续。）

表 4-3 异常位

异常	位的位置	应计异常位
无效	fp_invalid	i & (1 << fp_invalid)
上溢	fp_overflow	i & (1 << fp_overflow)
除	fp_division	i & (1 << fp_division)
underflow	fp_underflow	i & (1 << fp_underflow)
不精确	fp_inexact	i & (1 << fp_inexact)
denormalized	fp_denormalized	i & (1 << fp_denormalized) (仅限 x86)

下面的 C 或 C++ 程序段显示一种用来对返回值进行解码的方法。

```
/*
 *   Decode integer that describes all accrued exceptions.
 *   fp_inexact etc. are defined in <sys/ieeefp.h>
 */

char *out;
int invalid, division, overflow, underflow, inexact;

code = ieee_flags("get", "exception", "", &out);
printf ("out is %s, code is %d, in hex:0x%08X\n",
          out, code, code);
inexact   =       (code >> fp_inexact)& 0x1;
division  =       (code >> fp_division)& 0x1;
underflow =       (code >> fp_underflow)& 0x1;
overflow  =       (code >> fp_overflow)& 0x1;
invalid   =       (code >> fp_invalid)& 0x1;
printf("%d %d %d %d %d \n", invalid, division, overflow,
          underflow, inexact);
```

4.2.2 C99 异常标志函数

C/C++ 程序可以使用 C99 浮点环境函数测试、设置和清除浮点异常标志。头文件 `fenv.h` 定义五个与五个标准异常相对应的宏：`FE_INEXACT`、`FE_UNDERFLOW`、`FE_OVERFLOW`、`FE_DIVBYZERO` 和 `FE_INVALID`。它还定义要与所有这五个异常宏按位 “或” 的 `FE_ALL_EXCEPT` 宏。可将这些宏结合在一起，以便测试或清除异常标志的任何子集或者引发任何组合的异常。下面的示例显示如何将这些宏与几个 C99 浮点环境函数结合使用；要获得更多的信息，请参见 `feclearexcept`(3M) 手册页。

注 – 为了使行为保持一致，请不要在同一个程序中同时使用 `libm` 中的 C99 浮点环境函数和扩展以及 `libsunmath` 中的 `ieee_flags` 和 `ieee_handler` 函数。

要清除所有这五个异常标志，请编写：

```
feclearexcept(FE_ALL_EXCEPT);
```

要测试引发的是无效运算标志还是被零除标志，请编写：

```
int i;

i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
if (i & FE_INVALID)
    /* invalid flag was raised */
else if (i & FE_DIVBYZERO)
    /* division by zero flag was raised */
```

要模拟引发溢出异常 （请注意，在启用了溢出捕获时，这将引发捕获），请编写：

```
feraiseexcept(FE_OVERFLOW);
```

fegetexceptflag 和 fesetexceptflag 函数提供一种用来保存和恢复标志子集的方法。下面的示例显示这些函数的一种使用方法。

```
fexcept_t flags;

/* save the underflow, overflow, and inexact flags */
fegetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT);
/* clear these flags */
feclearexcept(FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT);
/* do a computation that can underflow or overflow */
...
/* check for underflow or overflow */
if (fetestexcept(FE_UNDERFLOW | FE_OVERFLOW) != 0) {
    ...
}
/* restore the underflow, overflow, and inexact flags */
fesetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW, | FE_INE
XACT);
```

4.3 查找异常

通常，编程人员写程序不考虑异常，因此当检测到异常时，首先要问的问题是：哪里发生异常？找出异常发生的位置的方法是在整个程序中各个点处测试异常标记，但是要用这种方法精确地隔离异常需要很多测试并且会带来巨大成本。

用来确定异常在何处发生的更方便的方法就是启用它的捕获。当发生启用了捕获的异常时，操作系统通过发送 SIGFPE 信号（请参见 –Ý½Ý(5) 手册页）来通知该程序。因此，通过启用对异常的捕获，并可通过以下方法来确定异常的发生位置：运行调试器并停止接收 SIGFPE 信号，或者建立 SIGFPE 处理程序以打印在其中发生了异常的指令的地址。请注意，必须针对异常启用捕获才能生成 SIGFPE 信号；如果在禁用捕获时发生异常，则会设置相应的标志，并使用表 4-1 中指定的默认结果继续执行程序，但不会发送任何信号。

4.3.1 使用调试器查找异常

本节举例说明如何使用 dbx 来调查浮点异常的原因并查找引发它的指令。重新调用它，以便使用 dbx 的源代码级调试功能，程序应当用術标志进行编译。要获得更多的信息，请参阅 《使用 dbx 调试程序》手册。

考虑下面的 C 程序：

```
#include <stdio.h>
#include <math.h>
double sqrtm1(double x)
{
   return sqrt(x) - 1.0;
}

int main(void)
{
   double x, y;

   x = -4.2;
   y = sqrtm1(x);
   printf("%g  %g\n", x, y);
   return 0;
}
```

编译和运行该程序会生成：

```
 -4.2  NaN
```

输出结果中 NaN 的外观表示可能发生了无效运算异常。要确定是否如此，可以用 -ftrap 选项重新编译以启用对无效运算的捕获，并使用 dbx 来运行该程序并在发出 SIGFPE 信号时停止。也可以使用 dbx，在无需重新编译该程序的情况下，通过用可启用无效运算捕获的启动例程进行链接或者手动启用该捕获来确定。

使用 dbx 来查找导致异常的指令

查找导致浮点异常的代码的最简单方法就是用 -g 和 -ftrap 标志重新编译，然后使用 dbx 来跟踪发生异常的位置。首先，按如下方式重新编译该程序：

```
example% cc -g -ftrap=invalid ex.c -lm
```

通过用 -g 进行编译，可以使用 dbx 的源代码级调试功能。指定 -ftrap=invalid 会导致在无效运算异常捕获处于启用的情况下运行该程序。接着，调用 dbx，发出 catch fpe 命令以便在 SIGFPE 发出时停止，然后运行该程序。在基于 SPARC 的系统上，结果如下所示：

```
example% dbx a.out
Reading a.out
... etc.
(dbx) catch fpe
(dbx) run
Running:a.out
(process id 2532)
signal FPE (invalid floating point operation) in __sqrt at 0xff36b3c4
0xff36b3c4:__sqrt+0x003c:be      __sqrt+0x98
Current function is sqrtm1
    6           return sqrt(x) - 1.0;
(dbx) print x
x = -4.2
(dbx)
```

输出结果表明，因试图求负数的平方根而在 sqrtm1 函数中出现异常。

也可以使用 dbx 识别代码（未使用 -g 编译，如库例程）中引发异常的原因。在这种情况下， dbx 不能给出源文件以及行号，但能够显示引发异常的指令。同样，第一步仍是使用 -ftrap 重新编译主程序：

```
example% cc -ftrap=invalid ex.c -lm
```

现在调用 dbx，使用 catch fpe 命令，然后运行该程序。在出现无效运算异常时，dbx 在导致该异常的指令后面的指令处停止。要查找导致该异常的指令，请对几个指令进行反汇编，并在 dbx 已停止的指令前面查找最后一个浮点指令。在基于 SPARC 的系统上，结果可能类似于下面的摘录代码。

```
example% dbx a.out
      Reading a.out
      ... etc.
      (dbx) catch fpe
      (dbx) run
      Running: a.out
      (process id 2532)
     signal FPE (invalid floating point operation) in __sqrt at
0xff2886f0
      0xff2886f0:__sqrt+0x0050:btst     %o5, %o2
      (dbx) dis __sqrt+0x40/4
      0xff2886e0:__sqrt+0x0040:sub      %g1, %o2, %o4
      0xff2886e4:__sqrt+0x0044:srlx     %o4, 63, %o3
      0xff2886e8:__sqrt+0x0048:xor      %o3, 1, %o2
      0xff2886ec:__sqrt+0x004c:fsqrtd   %f2, %f0
      (dbx) print $f2f3
      $f2f3 = -4.2
      (dbx)
```

输出结果表明该异常是由 fsqrtd 指令导致的。检查源寄存器会发现该异常是由于试图求负数的平方根而导致的。

在基于 x86 的系统上，因为指令没有固定的长度，所以要查找从中对代码进行反汇编的正确地址，可能要经过反复试验。在本例中，异常在接近函数的开头处发生，因此我们可从那里反汇编。（请注意，这一输出假定已用 -xlibmil 标志编译了该程序。）下面可能是典型的结果。

```
example% dbx a.out
Reading a.out
... etc.
(dbx) catch fpe
(dbx) run
Running:a.out
(process id 2532)
signal FPE (invalid floating point operation) in sqrtm1 at 0x80506bf
0x080506bf:sqrtm1+0x001f:fstpl    0xfffffff0(%ebp)
(dbx) dis sqrtm1+0x16/5
0x080506b6:sqrtm1+0x0016:pushl    %eax
0x080506b7:sqrtm1+0x0017:fldl     (%esp)
0x080506ba:sqrtm1+0x001a:fsqrt
0x080506bc:sqrtm1+0x001c:addl     $0x00000008,%esp
0x080506bf:sqrtm1+0x001f:fstpl    0xfffffff0(%ebp)
(dbx) print $st0
$st0 = -4.2000000000000001776356839400250464 7e+00
(dbx)
```

输出结果表明该异常是由 fsqrt 指令导致的；检查浮点寄存器会发现该异常是由于试图求负数的平方根而导致的。

在不重新编译的情况下启用捕获

上面的示例通过用 -ftrap 标志重新编译主要的子程序来启用对无效运算异常的捕获。在某些情况下，可能无法重新编译主程序，因此您可能需要借助于其他方法来启用捕获。启用捕获有多种方法。

如果您使用的是 dbx，则可以通过直接修改浮点状态寄存器来手动启用捕获。这会有些麻烦，因为只有当在程序中首次使用浮点 （此时浮点状态寄存器会在所有的捕获处于禁用状态时初始化）之后，操作系统才启用浮点单元。因此，之后在该程序至少执行了一个浮点指令之后，才能手动启用捕获。在我们的示例中，在调用 sqrtm1 函数之前已经访

问了浮点单元，因此我们可以在该函数的入口处设置断点，启用对无效运算异常的捕获，命令 dbx 停止接收 SIGFPE 信号，然后继续执行。在基于 SPARC 的系统上，步骤如下（请注意使用 assign 命令修改 %fsr 以启用对无效运算异常的捕获）：

```
example% dbx a.out
Reading a.out
... etc.
(dbx) stop in sqrtm1
dbx:warning:'sqrtm1' has no debugger info -- will trigger on first instruction
(2) stop in sqrtm1
(dbx) run
Running: a.out
(process id 23086)
stopped in sqrtm1 at 0x106d8
0x000106d8:sqrtm1       :save    %sp, -0x70, %sp
(dbx) assign $fsr=0x08000000
dbx:warning: unknown language, 'c' assumed
(dbx) catch fpe
(dbx) cont
signal FPE (invalid floating point operation) in __sqrt at 0xff36b3c4
0xff36b3c4:__sqrt+0x003c:be      __sqrt+0x98
(dbx)
```

在基于 x86 的系统上，同一个进程将类似如下内容：

```
example% dbx a.out
Reading a.out
... etc.
(dbx) stop in sqrtm1
dbx: warning: 'sqrtm1' has no debugger info -- will trigger on first instruction
(2) stop in sqrtm1
(dbx) run
Running: a.out
(process id 25055)
stopped in sqrtm1 at 0x80506b0
0x080506b0: sqrtm1       :pushl  %ebp
(dbx) assign $fctrl=0x137e
dbx: warning: unknown language, 'c' assumed
(dbx) catch fpe
(dbx) cont
signal FPE (invalid floating point operation) in sqrtm1 at 0x8050696
0x08050696: sqrtm1+0x0016:fstpl  -16(%ebp)
(dbx)
```

在上例中， assign 命令解除屏蔽 （即启用捕获）浮点控制字中的无效运算异常。如果程序使用 SSE2 指令，您必须对 MXCSR 寄存器中的异常解除屏蔽，从而启用对这些指令引发的异常的捕获。

还可以在不重新编译主程序或使用 dbx 的情况下，通过建立用来启用捕获的**初始化例程**来启用捕获。（这可能非常有用，例如，如果您希望在异常发生时中止该程序，而不运行调试器。）可通过两种方法来建立这样的例程。

如果存在包含该程序的目标文件和库，则可以通过用适当的初始化例程重新链接该程序来启用捕获。首先，创建一个类似如下的 C 源文件：

```
#include <ieeefp.h>

#pragma init (trapinvalid)

void trapinvalid()
{
     /* FP_X_INV et al are defined in ieeefp.h */
     fpsetmask(FP_X_INV);
}
```

现在，编译该文件，以便创建一个目标文件并用该目标文件链接初始程序：

```
example% cc -c init.c
example% cc ex.o init.o -lm
example% a.out
Arithmetic Exception
```

如果不可能进行重新链接，但是该程序已被动态链接，则可以通过使用运行时链接程序的共享对象预装功能来启用捕获。要在基于 SPARC 的系统上启用捕获，请创建一个如上所示的 C 源文件，但是按如下方式进行编译：

```
example% cc -Kpic -G -ztext init.c -o init.so -lc
```

现在，要启用捕获，请将 init.so 对象的路径名添加到由 LD_PRELOAD 环境变量指定的预装共享对象的列表中，例如：

```
example% env LD_PRELOAD=./init.so a.out
Arithmetic Exception
```

要获得有关创建和预装共享对象的更多信息，请参见 《链接程序和库指南》。

原则上，您可以通过按上述方式预装共享对象来更改任何浮点控制模式的初始化方法。但是，请注意，在运行时链接程序将控制转交给属于主可执行文件的启动代码之前，无论共享对象中的初始化例程是预装的还是显式链接的，它们都是由运行时链接程序执行的。启动代码随后建立任何通过 -ftrap、-fround、-fns (SPARC) 或 -fprecision (x86) 编译器标志选择的非默认模式，执行任何属于主可执行文件的初始化例程（包括那些静态链接的例程），最后将控制传递给主程序。因此，在 SPARC 上，(i) 任何由共享对象中的初始化例程建立的浮点控制模式（如在上例中启用的捕获）将在该程序的整个执行过程中保持有效（除非它们被覆盖）；(ii) 任何通过编辑器标志选择的**非默认**模式将覆盖由共享对象中的初始化例程建立的模式（但是，通过编译器标志选择的默认模式将不覆盖以前建立的模式）；(iii) 任何由属于主可执行文件的初始化例程或主程序本身建立的模式将覆盖由共享对象中的初始化例程建立的模式和以前建立的模式。

在基于 x86 的系统上，情况会略显复杂。通常，在建立由 -fround、-ftrap 或 -fprecision 标志选择的任何非默认模式并将控制权交给主程序前，由编译器自动提供的启动代码会调用 __fpstart 例程（位于标准 C 库 libc 中）将所有的浮点模式重置为默认设置。因此，在基于 x86 的系统上，为了通过用初始化例程预装共享对象来启用捕获（或者更改任何其他默认浮点模式），必须覆盖 __fpstart 例程，以便它不重置默认浮点模式。但是，__fpstart 替换例程仍应当像标准例程那样执行初始化函数的其余部分。下面的代码显示一种执行该操作的方法。此段代码假设主机平台运行于 Solaris 10 操作系统上。

```
#include <ieeefp.h>
#include <sys/sysi86.h>

#pragma init (trapinvalid)

void trapinvalid()
{
     /* FP_X_INV et al are defined in ieeefp.h */
     fpsetmask(FP_X_INV);
}

extern int  __fltrounds(), __flt_rounds;
extern int  _fp_hw, _sse_hw;

void __fpstart()
{
    /* perform the same floating point initializations as
       the standard __fpstart() function but leave all
       floating point modes as is */
    __flt_rounds = __fltrounds();
    (void) sysi86(SI86FPHW, &_fp_hw);

    /* set the following variable to 1 instead if the host
       platform supports SSE2 instructions */
    _sse_hw = 0;
}
```

4.3.2 使用信号处理程序来查找异常

上一节提供了几种用来在程序的开头启用捕获以查找第一次出现的异常的方法。与之相反，可通过在程序本身内启用捕获来隔离出现的任何特定异常。如果您启用了捕获，但未安装 SIGFPE 处理程序，则该程序将在下次出现捕获的异常时终止。或者，如果您安装了 SIGFPE 处理程序，则下次出现捕获的异常时将导致系统将控制转交给该处理程序，该处理程序随后打印诊断信息（如在其中发生了异常的指令的地址），然后终止或继续执行。（为了继续执行任何预计产生有意义结果的运算，处理程序可能需要为异常运算提供一个结果，如下一节所示。）

使用 ieee_handler 可以同时启用对五个 IEEE 浮点异常中的任何异常的捕获，要求在发生指定的异常时终止该程序或者建立 SIGFPE 处理程序。还可以使用较低级别的函数 sigfpe(3)、signal(3c) 或 sigaction(2) 之一来安装 SIGFPE 处理程序；但是，这些函数不像 ieee_handler 那样启用捕获。（切记，只有启用了对浮点异常的捕获时，浮点异常才触发 SIGFPE 信号。）

ieee_handler (3m)

ieee_handler 的调用语法是：

```
i = ieee_handler(action, exception, handler)
```

前两个输入参数 *action* 和 *exception* 是字符串。第三个输入参数 *handler* 的类型为 sigfpe_handler_type，该类型是在 floatingpoint.h 中进行定义的。

三个输入参数可使用以下值：

输入参数	C 或 C++ 类型	可能的值
action	char*	get、 set、 clear
异常	char*	invalid、division、overflow、 underflow、 inexact、 all、 common
handler	sigfpe_handler_type	用户定义的例程 SIGFPE_DEFAULT SIGFPE_IGNORE SIGFPE_ABORT

当请求的操作为 "set" 时， ieee_handler 建立由 *handler* 指定的名为 *exception* 异常的处理函数。处理函数可能为 SIGFPE_DEFAULT 或 SIGFPE_IGNORE，两者都选择默认的 IEEE 行为，当发生任一指定异常时， SIGFPE_ABORT 会引起程序异常终止，或者用户提供的子例程的地址引起该子例程被调用 (带有 sigaction(2) 手册页关于装有 SA_SIGINFO 标志设置的信号处理程序中所描述的参数)。如果处理程序为 SIGFPE_DEFAULT 或 SIGFPE_IGNORE，另外，指定异常发生时 ieee_handler 会禁用捕获，对

于其他处理程序，ieee_handler 会启用捕获。(在 x86 平台上，无论何时异常的捕获处于启用状态并且引起相应的标记，浮点硬件启用捕获。因此，要避免假捕获，程序应在调用 ieee_handler 来启用自陷之前清除每一个指定的 *exception* 的标记。)

当请求的操作是 "clear" 时，ieee_handler 吊销处理函数当前正针对指定的 *exception* 安装的异常并禁用对它的捕获。(这与针对 SIGFPE_DEFAULT 执行 "set" 操作相同。) 当操作为 "clear" 时，会忽略第三个参数。

对于 "set" 和 "clear" 操作，如果请求的操作可用，则 ieee_handler 返回 0；否则返回非零值。

当请求的操作是 "get" 时，ieee_handler 返回当前为指定的 *exception* 安装的处理程序的地址 (如果未安装任何处理程序，则返回 SIGFPE_DEFAULT)。

下面的示例显示几个代码段，通过它们可阐释如何使用 ieee_handler。下面的 C 代码导致该程序在遇到被零除时终止：

```
#include <sunmath.h>
/* uncomment the following line on x86 systems */
    /* ieee_flags("clear", "exception", "division", NULL); */
    if (ieee_handler("set", "division", SIGFPE_ABORT) != 0)
        printf("ieee trapping not supported here \n");
```

下面是等效的 Fortran 代码：

```
#include <floatingpoint.h>
c uncomment the following line on x86 systems
c     ieee_flags('clear', 'exception', 'division', %val(0))
      i = ieee_handler('set', 'division', SIGFPE_ABORT)
      if(i.ne.0) print *,'ieee trapping not supported here'
```

下面的 C 代码段恢复对所有异常的 IEEE 默认异常处理：

```
#include <sunmath.h>
    if (ieee_handler("clear", "all", 0) != 0)
        printf("could not clear exception handlers\n");
```

下面是 Fortran 中同样的操作：

```
      i = ieee_handler('clear', 'all', 0)
      if (i.ne.0) print *, 'could not clear exception handlers'
```

从信号处理程序报告异常

当通过 ieee_handler 安装的 SIGFPE 处理程序被调用时，操作系统提供其他信息，指出所发生的异常的类型、导致该异常的指令的地址以及机器的整数和浮点寄存器的内容。该处理程序可以检查这些信息，并打印用来标识异常及其所发生位置的消息。

要访问由系统提供的信息，请按如下方式声明处理程序。本章的其余部分提供 C 示例代码；要查看 SIGFPE 处理程序的 Fortran 示例，请参见附录 A。

```
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    ...
}
```

当该处理程序被调用时，*sig* 参数包含已发送的信号的数量。信号数量是在 sys/signal.h 中定义的； SIGFPE 的信号数量是 8。

sip 参数指向可记录有关该信号的其他信息的结构。对于 SIGFPE 信号，该结构的相关成员是 sip->si_code 和 sip->si_addr （请参见 sys/siginfo.h）。这些成员的重要性取决于系统以及用来触发 SIGFPE 信号的事件。

sip->si_code 成员是列在表 4-4 中的 SIGFPE 信号类型之一。（所显示的标记是在 sys/machsig.h 中定义的。）

表 4-4 算术异常的类型

SIGFPE 类型	IEEE 类型
FPE_INTDIV	
FPE_INTOVF	
FPE_FLTRES	不精确
FPE_FLTDIV	除
FPE_FLTUND	underflow
FPE_FLTINV	无效
FPE_FLTOVF	上溢

正如上表所示，每种类型的 IEEE 浮点异常都有一个与之对应的 SIGFPE 信号类型。整数被零除 (FPE_INTDIV) 和整数溢出 (FPE_INTOVF) 也包括在 SIGFPE 类型中，但是由于它们不是 IEEE 浮点异常，所以您不能通过 ieee_handler 来为它们安装处理程序。（可通过 sigfpe(3) 来为这些 SIGFPE 类型安装处理程序；但是，请注意，在所有的 SPARC 和 x86 平台上，会在默认情况下忽略整数溢出。特殊的指令可导致传递 FPE_INTOVF 类型的 SIGFPE 信号，但是 Sun 编译器不生成这些指令。）

对于与 IEEE 浮点异常对应的 SIGFPE 信号，sip->si_code 成员用于指明出现了哪个异常。（在基于 x86 的系统上，它实际上用于指明引发了其标志的拥有最高优先级的解除屏蔽异常。这通常与最后出现的异常是一样的。）在基于 SPARC 的系统上， sip->si_addr 成员存放导致了异常的指令的地址，而在基于 x86 的系统上，它存放用来进行捕获的指令的地址 （通常是紧跟在导致异常的指令后面的浮点指令）。

最后， *uap* 参数指向用来记录在进行捕获时系统状态的结构。该结构的内容与系统有关；要查看它的某些成员的定义，请参见 sys/reg.h。

使用操作系统提供的信息，可以编写 SIGFPE 处理程序，该处理程序报告所发生的异常的类型以及导致它的指令的地址。代码样例 4-1 显示这样的处理程序。

代码样例 4-1 SIGFPE 处理程序

```
#include <stdio.h>
#include <sys/ieeefp.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    code, addr;

    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
    fprintf(stderr, "fp exception %x at address %x\n", code,
        addr);
}

int main()
{
    double  x;

    /* trap on common floating point exceptions */
    if (ieee_handler("set", "common", handler) != 0)
        printf("Did not set exception handler\n");
    /* cause an underflow exception (will not be reported) */
    x = min_normal();
    printf("min_normal = %g\n", x);
    x = x / 13.0;
    printf("min_normal / 13.0 = %g\n", x);

    /* cause an overflow exception (will be reported) */
```

代码样例 4-1 SIGFPE 处理程序 (续)

```
    x = max_normal();
    printf("max_normal = %g\n", x);
    x = x * x;
    printf("max_normal * max_normal = %g\n", x);
    ieee_retrospective(stderr);
    return 0;
}
```

在 SPARC 系统上，该程序的输出结果类似于如下内容：

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
 Note:IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M)
```

在 x86 平台上，在调用 SIGFPE 处理程序之前，操作系统保存应计异常标志的副本，然后清除这些标志。除非该处理程序执行用来保存这些标志的步骤，否则应计标志将在该处理程序返回之后丢失。因此，上述程序的输出结果并未指出引发了下溢异常。

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 8048fe6
max_normal * max_normal = 1.79769e+308
 Note:IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_handler(3M)
```

在大多数情况下，在捕获处于启用状态时，导致异常的指令不传递 IEEE 默认结果：在上面的输出结果中，针对 max_normal * max_normal 报告的值不是溢出运算 （即，带有正确符号的无穷大）的默认结果。通常，SIGFPE 处理程序必须为导致捕获异常的运算提供一个结果，以便继续用有意义的值进行计算。要查看执行上述操作的一种方法，请参见第 26 页的 “处理异常”。

4.3.3 使用 libm 异常处理扩展来查找异常

使用 libm 中 C99 浮点环境函数的异常处理扩展，C/C++ 程序可通过多种方法来查找异常。这些扩展包括可建立处理程序并同时启用捕获（正如 ieee_handler 所执行的操作那样）的函数，但是它们提供更大的灵活性。它们还支持将有关浮点异常的诊断消息记录到选定文件中。

fex_set_handling(3m)

fex_set_handling 函数允许您选择某个选项或模式来处理每种类型的浮点异常。fex_set_handling 的调用语法是：

```
ret = fex_set_handling(ex, mode, handler);
```

ex 参数指定要应用调用的异常集合。它必须是列在表 4-5 的第一列中的值的按位“或”。（这些值是在 fenv.h 中定义的。）

表 4-5 fex_set_handling 的异常代码

值	异常
FEX_INEXACT	不精确结果
FEX_UNDERFLOW	underflow
FEX_OVERFLOW	上溢
FEX_DIVBYZERO	被零除
FEX_INV_ZDZ	0/0 无效运算
FEX_INV_IDI	无穷大 / 无穷大无效运算
FEX_INV_ISI	无穷大 - 无穷大无效运算
FEX_INV_ZMI	0* 无穷大无效运算
FEX_INV_SQRT	负数的平方根
FEX_INV_SNAN	发出 NaN 的运算
FEX_INV_INT	无效的整数转换
FEX_INV_CMP	无效的无序比较

为方便起见，fenv.h 还定义以下值：FEX_NONE（没有异常）、FEX_INVALID（所有的无效运算异常）、FEX_COMMON（溢出、被零除和所有的无效运算）和 FEX_ALL（所有异常）。

mode 参数指定要为指出的异常建立的异常处理模式。共有五种可能的模式：

- FEX_NONSTOP 模式提供 IEEE 754 默认的不间断行为。这等效于使异常捕获保持禁用。（请注意，与 ieee_handler 不同的是，fex_set_handling 允许您为某些类型的无效运算异常建立非默认处理并为其他异常保留 IEEE 默认处理。）

- FEX_NOHANDLER 模式等效于在不提供处理程序的情况下启用异常捕获。在发生异常时，如果以前安装了 SIGFPE 处理程序，系统会将控制转交给该处理程序，否则会终止。
- FEX_ABORT 模式导致程序在发生异常时调用 abort(3c)。
- FEX_SIGNAL 安装由 *handler* 参数为指出的异常指定的处理函数。当发生其中的任一异常时，会用相同的参数调用该处理程序，就好像它是由 ieee_handler 安装的一样。
- FEX_CUSTOM 安装由 *handler* 为指出的异常指定的处理函数。与 FEX_SIGNAL 模式不同的是，在发生异常时，会用简化的参数列表调用该处理程序。这些参数由一个整数（其值是列在表 4-5 中的某个值）和一个指针（它所指向的结构用来记录有关导致该异常的运算的附加信息）组成。该结构的内容在下一节和 fex_set_handling(3m) 手册页中介绍。

请注意，如果指定的 *mode* 是 FEX_NONSTOP、FEX_NOHANDLER 或 FEX_ABORT，*handler* 参数会被忽略。如果指定的模式是为指出的异常建立的，fex_set_handling 会返回非零值，否则将返回零。（在下面的示例中，返回值将被忽略。）

下面的示例假设使用 fex_set_handling 方法来查找某些类型的异常。要终止 0/0 异常，请编写：

```
fex_set_handling(FEX_INV_ZDZ, FEX_ABORT, NULL);
```

要为溢出和被零除安装 SIGFPE 处理程序，请编写：

```
fex_set_handling(FEX_OVERFLOW | FEX_DIVBYZERO, FEX_SIGNAL,
    handler);
```

在上面的示例中，处理程序函数可打印通过 *sip* 参数提供给 SIGFPE 处理程序的诊断信息，如上面的子段所示。与之相反，下面的示例打印有关该异常的、提供给安装在 FEX_CUSTOM 模式下的处理程序的信息。（要查看更多的信息，请参见 fex_set_handling(3m) 手册页。）

代码样例 4-2 打印提供给安装在 FEX_CUSTOM 模式下的处理程序的信息

```
#include <fenv.h>

void handler(int ex, fex_info_t *info)
{
    switch (ex) {
    case FEX_OVERFLOW:
        printf("Overflow in ");
        break;
```

代码样例 4-2 打印提供给安装在 FEX_CUSTOM 模式下的处理程序的信息 (续)

```
    case FEX_DIVBYZERO:
        printf("Division by zero in ");
        break;

    default:
        printf("Invalid operation in ");
    }
    switch (info->op) {
    case fex_add:
        printf("floating point add\n");
        break;
    case fex_sub:
        printf("floating point subtract\n");
        break;
    case fex_mul:
        printf("floating point multiply\n");
        break;
    case fex_div:
        printf("floating point divide\n");
        break;
    case fex_sqrt:
        printf("floating point square root\n");
        break;
    case fex_cnvt:
        printf("floating point conversion\n");
        break;
    case fex_cmp:
        printf("floating point compare\n");
        break;
    default:
        printf("unknown operation\n");
    }
    switch (info->op1.type) {
    case fex_int:
        printf("operand 1:%d\n", info->op1.val.i);
        break;
    case fex_llong:
        printf("operand 1:%lld\n", info->op1.val.l);
        break;
    case fex_float:
        printf("operand 1:%g\n", info->op1.val.f);
        break;
```

代码样例 4-2 打印提供给安装在 FEX_CUSTOM 模式下的处理程序的信息 (续)

```
    case fex_double:
        printf("operand 1:%g\n", info->op1.val.d);
        break;

    case fex_ldouble:
        printf("operand 1:%Lg\n", info->op1.val.q);
        break;
    }
    switch (info->op2.type) {
    case fex_int:
        printf("operand 2:%d\n", info->op2.val.i);
        break;
    case fex_llong:
        printf("operand 2:%lld\n", info->op2.val.l);
        break;
    case fex_float:
        printf("operand 2:%g\n", info->op2.val.f);
        break;
    case fex_double:
        printf("operand 2:%g\n", info->op2.val.d);
        break;
    case fex_ldouble:
        printf("operand 2:%Lg\n", info->op2.val.q);
        break;
    }
}
...
fex_set_handling(FEX_COMMON, FEX_CUSTOM, handler);
```

上例中的处理程序报告所发生的异常的类型、导致它的运算的类型以及操作数。它不指出异常发生的位置。要找出异常发生的位置，可使用回顾诊断方法。

回顾诊断

使用 libm 异常处理扩展查找异常的另一种方法是启用对有关浮点异常的回顾诊断消息的记录。当您启用对回顾诊断消息的记录时，系统会记录有关某些异常的信息。这些信息包括异常的类型、导致它的指令的地址、将要处理它的方式、类似于调试器生成的跟踪的栈跟踪。（用回顾诊断消息记录的栈跟踪只包含指令地址和函数名；对于其他调试信息（如行号、源文件名和参数值），必须使用调试器。）

并非每个所发生的异常都包含在回顾诊断日志中；如果出现了这种情况，则典型日志将非常大，而且将不可能隔离与众不同的异常。相反，日志记录机制会消除冗余的消息。在以下任一情况下，消息会被认为冗余：

- 以前在同一位置（即，指令地址和栈跟踪相同）记录了同一个异常，或者
- FEX_NONSTOP 模式对于该异常有效，而且它的标志以前被引发过。

尤其是，在大多数程序中，对于每种类型的异常将只记录第一次出现的异常。（当 FEX_NONSTOP 处理模式对于某个异常有效时，如果通过 C99 浮点环境的任一函数清除该异常的标志，则允许在该异常下次发生时记录它，但前提是它不在以前记录它的位置发生。）

要启用日志记录，请使用 fex_set_log 函数指定应将消息传递到的文件。例如，要将消息记录到标准错误文件，请使用：

```
fex_set_log(stderr);
```

代码样例 4-3 将以下两个功能组合在一起：对回顾诊断消息的记录；上一节中阐释的共享对象预装功能。通过创建下面的 C 源文件、将它编译为共享对象、通过在 LD_PRELOAD 环境变量中提供共享对象的路径名来预装共享对象、在 FTRAP 环境变量中指定一个或多个异常的名称（用逗号分开），可同时针对指定的异常终止该程序，并获取可显示每个异常发生位置的诊断输出结果。

代码样例 4-3 对回顾诊断消息的记录和共享对象预装功能结合在一起

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fenv.h>

static struct ftrap_string {
    const char  *name;
    int         value;
} ftrap_table[] = {
    { "inexact", FEX_INEXACT },
    { "division", FEX_DIVBYZERO },

    { "underflow", FEX_UNDERFLOW },
    { "overflow", FEX_OVERFLOW },
    { "invalid", FEX_INVALID },
    { NULL, 0 }
};

#pragma init (set_ftrap)
```

代码样例 4-3 对回顾诊断消息的记录和共享对象预装功能结合在一起 (续)

```
void set_ftrap()
{
    struct ftrap_string  *f;
    char                 *s, *s0;
    int                  ex = 0;

    if ((s = getenv("FTRAP")) == NULL)
        return;

    if ((s0 = strtok(s, ",")) == NULL)
        return;

    do {
        for (f = &trap_table[0]; f->name != NULL; f++) {
            if (!strcmp(s0, f->name))
                ex |= f->value;
        }
    } while ((s0 = strtok(NULL, ",")) != NULL);

    fex_set_handling(ex, FEX_ABORT, NULL);
    fex_set_log(stderr);
}
```

在基于 SPARC 的系统上，结合使用上面的代码和本节开头处提供的示例程序可生成以下结果：

```
example% cc -Kpic -G -ztext init.c -o init.so -R/opt/SUNWspro/lib
-L/opt/SUNWspro/lib -lm9x -lc
example% env FTRAP=invalid LD_PRELOAD=./init.so a.out
Floating point invalid operation (sqrt) at 0x00010c24 sqrtm1_, abort
  0x00010c30  sqrtm1_
  0x00010b48  MAIN_
  0x00010ccc  main
Abort
```

上面的输出结果表明，由于 `sqrtm1` 例程中的平方根运算而导致引发了无效运算异常。

（如上所述，在 x86 平台上，要从共享对象中的初始化例程启用捕获，必须覆盖 `__fpstart` 例程。）

附录 A 提供了更多显示典型日志输出的示例。要查看一般信息，请参见 `fex_set_log`(3m) 手册页。

4.4 处理异常

在以前，大多数数值软件在编写时都未考虑异常（原因多种多样），许多编程人员经常遇到异常导致程序立即终止的环境。现在，一些高质量的软件包 （如 LAPACK）经过了仔细设计，从而避免了异常 （如被零除和无效运算）并主动设置其输入范围以排除溢出和可能有害的下溢。在处理异常的这些方法中，没有一种能够适合所有情况。但是，当一个人编写的程序或子例程将要由他人 （可能是没有源代码访问权限的人）使用时，忽略异常可能会导致问题，而且尝试避免所有异常可能需要许多防御性测试和分支，并且还会带来巨大成本（请参见 Demmel 和 Li 著的“Faster Numerical Algorithms via Exception Handling”，这篇文章发表在 《IEEE Trans.Comput.》的第 43 期 (1994)，位于第 983 - 992 页。）

IEEE 算法的默认异常响应、状态标记和可选的捕获功能旨在提供第三个替换功能：在异常存在的情况下继续计算，并在事后检测它们或者在它们发生时解释和处理它们。如上所述， `ieee_flags` 或 C99 浮点环境函数可用于在事后检测异常， `ieee_handler` 或 `fex_set_handling` 可用于启用捕获并安装要在异常发生时解释它们的处理程序。但是，为了继续计算，IEEE 标准推荐使用能够为导致了异常的运算提供结果的捕获处理程序。在 `FEX_SIGNAL` 模式中通过 `ieee_handler` 或 `fex_set_handling` 安装的 `SIGFPE` 处理程序可使用 *uap* 参数（由 Solaris 操作系统提供给信号处理程序）完成上述操作。通过 `fex_set_handling` 安装的 `FEX_CUSTOM` 模式处理程序可使用提供给类似处理程序的 *info* 参数提供结果。

记得在 C 中，可按如下方式声明 `SIGFPE` 信号处理程序：

```
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    ...
}
```

当由于捕获到浮点异常而调用 `SIGFPE` 信号处理程序时，*uap* 参数将指向一个数据结构，该结构中包含机器的整数和浮点寄存器的副本，以及其他描述该异常的、依赖系统的信息。如果该信号处理程序正常返回，则所保存的数据将被恢复，该程序将从捕获点继续执行。因此，通过访问描述异常的数据结构中的信息、对该信息进行解码并 （可能）修改所保存的数据，`SIGFPE` 处理程序可将用户提供的值替换为异常运算的结果并继续计算。

可按如下方式声明 FEX_CUSTOM 模式处理程序：

```
#include <fenv.h>

void handler(int ex, fex_info_t *info)
{
    ...
}
```

当 FEX_CUSTOM 处理程序被调用时， *ex* 参数指出所发生的异常的类型 （该类型是列在表 4-5 中的某个值），*info* 参数指向一个包含更多异常信息的数据结构。特别是，该结构中包含一个代码 （代表导致了该异常的算术运算）和多个用来记录操作数 （如果它们可用的话）的结构。该结构中还包含一个用来记录默认结果 （如果异常未被捕获，这些结果将被替换）的结构和一个整数值（保存应产生的异常标志的按位“或”）。此处理程序可以修改该结构后面的成员，以便替换另一个结果或者更改应计标志的设置。（请注意，如果该处理程序在不修改这些数据的情况下返回，则该程序将继续显示默认的未捕获结果和标志，就好像异常未被捕获一样。）

上一节举例说明了如何替换下溢或溢出运算的缩放结果。要查看进一步的示例，请参见附录 A。

替换 IEEE 捕获的下溢 / 溢出结果

IEEE 标准建议：在捕获到下溢或溢出时，系统应当为捕获处理程序提供一种方法来替换**指数环绕**的结果，即，该值与将成为下溢或溢出运算的舍入结果值相一致 （不同之处在于指数在其通常范围的末端是被环绕的），从而有效地按 2 的幂缩放结果。在选择比例因子时，使下溢或溢出结果与指数范围的中间尽可能接近，这样将减少以后的计算进一步出现下溢或溢出的可能性。通过跟踪所发生的下溢或溢出的数量，程序可通过缩放最终结果来补偿指数环绕。这种下溢 / 溢出“计数模式”可用于在计算中产生准确结果，否则将超出可用浮点格式的范围。（请参见 P. Sterbenz 的浮点计算。）

在基于 SPARC 的系统上，当浮点指令导致捕获异常时，系统会使目标寄存器保持不变。因此，为了替换指数环绕结果，下溢 / 溢出处理程序必须对指令对象解码、检查操作数寄存器、生成缩放结果本身。代码样例 4-4 显示了一个执行这三个步骤的处理程序。（为了结合使用该处理程序和为基于 UltraSPARC 的系统编译的代码，请在运行 Solaris 2.6 操作系统、Solaris 7 操作系统或 Solaris 8 操作系统的系统上编译该处理程序，并定义预处理程序标记 V8PLUS。）

代码样例 4-4 对于基于 SPARC 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果

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

代码样例 4-4 对于基于 SPARC 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

#ifdef V8PLUS
/* The upper 32 floating point registers are stored in an area
   pointed to by uap->uc_mcontext.xrs.xrs_prt.Note that this
   pointer is valid ONLY when uap->uc_mcontext.xrs.xrs_id ==
   XRS_ID (defined in sys/procfs.h). */
#include <assert.h>
#include <sys/procfs.h>
#define FPxreg(x)  ((prxregset_t*)uap->uc_mcontext.xrs.xrs_ptr)
->pr_un.pr_v8p.pr_xfr.pr_regs[(x)]
#endif

#define FPreg(x)   uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[(x)]

/*
*  Supply the IEEE 754 default result for trapped under/overflow
*/
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    instr, opf, rs1, rs2, rd;
    long double qs1, qs2, qd, qscl;
    double      ds1, ds2, dd, dscl;
    float       fs1, fs2, fd, fscl;

    /* get the instruction that caused the exception */
    instr = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

    /* extract the opcode and source and destination register
       numbers */
    opf = (instr >> 5) & 0x1ff;
    rs1 = (instr >> 14) & 0x1f;
    rs2 = instr & 0x1f;
    rd = (instr >> 25) & 0x1f;
    /* get the operands */
    switch (opf & 3) {
    case 1:/* single precision */
        fs1 = *(float*)&FPreg(rs1);
        fs2 = *(float*)&FPreg(rs2);
```

代码样例 4-4 对于基于 SPARC 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
        break;

    case 2:/* double precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds1 = *(double*)&FPxreg(rs1 & 0x1e);
        }
        else
            ds1 = *(double*)&FPreg(rs1);
        if (rs2 & 1)

        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds2 = *(double*)&FPxreg(rs2 & 0x1e);
        }
        else
            ds2 = *(double*)&FPreg(rs2);
#else
        ds1 = *(double*)&FPreg(rs1);
        ds2 = *(double*)&FPreg(rs2);
#endif
        break;

    case 3:/* quad precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            qs1 = *(long double*)&FPxreg(rs1 & 0x1e);
        }
        else
            qs1 = *(long double*)&FPreg(rs1);
        if (rs2 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            qs2 = *(long double*)&FPxreg(rs2 & 0x1e);
        }
        else
            qs2 = *(long double*)&FPreg(rs2);
#else
```

代码样例 4-4 对于基于 SPARC 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
        qs1 = *(long double*)&FPreg(rs1);
        qs2 = *(long double*)&FPreg(rs2);
#endif
        break;
    }

    /* set up scale factors */
    if (sip->si_code == FPE_FLTOVF) {
        fscl = scalbnf(1.0f, -96);
        dscl = scalbn(1.0, -768);
        qscl = scalbnl(1.0, -12288);

    } else {
        fscl = scalbnf(1.0f, 96);
        dscl = scalbn(1.0, 768);
        qscl = scalbnl(1.0, 12288);
    }

    /* disable traps and generate the scaled result */
    fpsetmask(0);
    switch (opf) {
    case 0x41:/* add single */
        fd = fscl * (fscl * fs1 + fscl * fs2);
        break;

    case 0x42:/* add double */
        dd = dscl * (dscl * ds1 + dscl * ds2);
        break;

    case 0x43:/* add quad */
        qd = qscl * (qscl * qs1 + qscl * qs2);
        break;
    case 0x45:/* subtract single */
        fd = fscl * (fscl * fs1 - fscl * fs2);
        break;

    case 0x46:/* subtract double */
        dd = dscl * (dscl * ds1 - dscl * ds2);
        break;

    case 0x47:/* subtract quad */
        qd = qscl * (qscl * qs1 - qscl * qs2);
```

```
        break;

    case 0x49:/* multiply single */
        fd = (fscl * fs1) * (fscl * fs2);
        break;

    case 0x4a:/* multiply double */
        dd = (dscl * ds1) * (dscl * ds2);
        break;

    case 0x4b:/* multiply quad */
        qd = (qscl * qs1) * (qscl * qs2);
        break;

    case 0x4d:/* divide single */
        fd = (fscl * fs1) / (fs2 / fscl);
        break;

    case 0x4e:/* divide double */
        dd = (dscl * ds1) / (ds2 / dscl);
        break;

    case 0x4f:/* divide quad */
        qd = (qscl * qs1) / (qs2 / dscl);
        break;

    case 0xc6:/* convert double to single */
        fd = (float) (fscl * (fscl * ds1));
        break;
    case 0xc7:/* convert quad to single */
        fd = (float) (fscl * (fscl * qs1));
        break;

    case 0xcb:/* convert quad to double */
        dd = (double) (dscl * (dscl * qs1));
        break;
    }

    /* store the result in the destination */
    if (opf & 0x80) {
        /* conversion operation */
        if (opf == 0xcb) {
```

代码样例 4-4 对于基于 SPARC 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
            /* convert quad to double */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(double*)&FPxreg(rd & 0x1e) = dd;
            }

            else
                *(double*)&FPreg(rd) = dd;
#else
            *(double*)&FPreg(rd) = dd;
#endif
        } else
            /* convert quad/double to single */
            *(float*)&FPreg(rd) = fd;
    } else {
        /* arithmetic operation */
        switch (opf & 3) {
        case 1:/* single precision */
            *(float*)&FPreg(rd) = fd;
            break;
        case 2:/* double precision */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(double*)&FPxreg(rd & 0x1e) = dd;
            }
            else
                *(double*)&FPreg(rd) = dd;
#else
            *(double*)&FPreg(rd) = dd;
#endif
            break;

        case 3:/* quad precision */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(long double*)&FPxreg(rd & 0x1e) = qd;
```

代码样例 4-4 对于基于 SPARC 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
            }
            else
                *(long double*)&FPreg(rd & 0x1e) = qd;

#else
            *(long double*)&FPreg(rd & 0x1e) = qd;
#endif
            break;
        }
    }
}

int
main()
{
    volatile float  a, b;
    volatile double x, y;

    ieee_handler("set", "underflow", ieee_trapped_default);
    ieee_handler("set", "overflow", ieee_trapped_default);
    a = b = 1.0e30f;
    a *= b; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    a /= b; /* underflow; will wrap back */
    printf("%g\n", a);

    x = y = 1.0e300;
    x *= y; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", x);
    x /= y;
    printf("%g\n", x);
    x /= y; /* underflow; will wrap back */
    printf("%g\n", x);

    ieee_retrospective(stdout);
    return 0;
}
```

在上例中，变量 a、b、x 和 y 都已被声明为 volatile，其目的仅在于防止编译器在编译时对 a * b 等求值。在典型的用法中，将不需要进行 volatile 声明。

上述程序的输出结果是：

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note:IEEE floating-point exception traps enabled:
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

在基于 x86 的系统上，当浮点指令导致捕获到下溢或溢出，而且它的目标是寄存器时，浮点硬件提供指数环绕结果。但是，当在浮点存储指令上出现捕获到的下溢或溢出时，该硬件将在不完成存储（而且，如果存储指令不是存储－弹出，将不弹出栈）的情况下捕获。因此，为了实现计数模式，当在存储指令上出现捕获时，下溢 / 溢出处理程序必须生成缩放结果并修复栈。代码样例 4-5 阐释了这样的处理程序。

代码样例 4-5 对于基于 x86 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果

```
#include <stdio.h>
#include <ieeefp.h>
#include <math.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

/* offsets into the saved fp environment */
#define CW    0    /* control word */
#define SW    1    /* status word */
#define TW    2    /* tag word */
#define OP    4    /* opcode */
#define EA    5    /* operand address */

#define FPenv(x)    uap->uc_mcontext.fpregs.fp_reg_set.
fpchip_state.state[(x)]

#define FPreg(x)    *(long double *)(10*(x)+(char*)&uap->
uc_mcontext.fpregs.fp_reg_set.fpchip_state.state[7])
/*
*  Supply the IEEE 754 default result for trapped under/overflow

*/
```

代码样例 4-5 对于基于 x86 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    double      dscl;
    float       fscl;
    unsigned    sw, op, top;
    int         mask, e;

    /* preserve flags for untrapped exceptions */
    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status;
    FPenv(SW) |= (sw & (FPenv(CW) & 0x3f));
    /* if the excepting instruction is a store, scale the stack
       top, store it, and pop the stack if need be */
    fpsetmask(0);
    op = FPenv(OP) >> 16;
    switch (op & 0x7f8) {
    case 0x110:
    case 0x118:
    case 0x150:
    case 0x158:
    case 0x190:
    case 0x198:
        fscl = scalbnf(1.0f, (sip->si_code == FPE_FLTOVF)?
            -96 : 96);
        *(float *)FPenv(EA) = (FPreg(0) * fscl) * fscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
        break;

    case 0x510:
```

代码样例 4-5 对于基于 x86 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
    case 0x518:

    case 0x550:
    case 0x558:
    case 0x590:
    case 0x598:
        dscl = scalbn(1.0, (sip->si_code == FPE_FLTOVF)?
            -768 : 768);
        *(double *)FPenv(EA) = (FPreg(0) * dscl) * dscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
        break;
    }
}

int main()
{
    volatile float  a, b;
    volatile double x, y;

    ieee_handler("set", "underflow", ieee_trapped_default);
    ieee_handler("set", "overflow", ieee_trapped_default);
    a = b = 1.0e30f;
    a *= b;
    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    x = y = 1.0e300;
```

代码样例 4-5 对于基于 x86 系统替换 IEEE 捕获的下溢 / 溢出处理程序结果 (续)

```
    x *= y;
    printf("%g\n", x);

    x /= y;
    printf("%g\n", x);
    x /= y;
    printf("%g\n", x);

    ieee_retrospective(stdout);
    return 0;
}
```

正如在基于 SPARC 的系统上一样，上述程序在 x86 上的输出结果是：

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note:IEEE floating-point exception traps enabled:
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

C/C++ 程序可以使用 `libm` 中的 `fex_set_handling` 函数来为下溢和溢出安装 `FEX_CUSTOM` 处理程序。在基于 SPARC 的系统上，提供给类似处理程序的信息总是包括导致了异常的运算以及操作数，这些信息足以允许该处理程序计算 IEEE 指数环绕结果，如上所示。在基于 x86 的系统上，可用信息并不总是指出导致了异常的特定运算；例如，当异常由某个超越指令引发时，`info->op` 参数设置为 `fex_other`。（要查看定义，请参见 `fenv.h` 文件。）而且，x86 硬件自动提供指数环绕结果，如果异常指令的目标是浮点寄存器，这将覆盖某个操作数。

幸运的是，`fex_set_handling` 功能为在 `FEX_CUSTOM` 模式下安装的处理程序提供了一种简单的方法，用来替换下溢或溢出运算的 IEEE 指数环绕结果。当捕获到其中的任一异常时，该处理程序会设置

```
info->res.type = fex_nodata;
```

指出应当提供指数环绕结果。下面的示例显示类似的处理程序：

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

void handler(int ex, fex_info_t *info) {
    info->res.type = fex_nodata;
}
int main()
{
    volatile float  a, b;
    volatile double x, y;

    fex_set_log(stderr);
    fex_set_handling(FEX_UNDERFLOW | FEX_OVERFLOW, FEX_CUSTOM,
        handler);
    a = b = 1.0e30f;
    a *= b; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    a /= b; /* underflow; will wrap back */
    printf("%g\n", a);

    x = y = 1.0e300;
    x *= y; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", x);
    x /= y;

    printf("%g\n", x);
    x /= y; /* underflow; will wrap back */
    printf("%g\n", x);

    return 0;
}
```

上述程序的输出结果类似于如下内容：

```
Floating point overflow at 0x00010924 main, handler:handler
  0x00010928 main
159.309
1.59309e-28
Floating point underflow at 0x00010994 main, handler:handler
  0x00010998 main
1
Floating point overflow at 0x000109e4 main, handler:handler
  0x000109e8 main
4.14884e+137
4.14884e-163
Floating point underflow at 0x00010a4c main, handler:handler
  0x00010a50 main
1
```

附录 A

示例

本附录提供有关如何完成某些常见任务的示例。这些示例是用 Fortran 或 ANSI C 编写的，很多示例取决于当前的 `libm` 和 `libsunmath` 版本。这些示例已在 Solaris 操作系统中使用当前的 C 和 Fortran 编译器进行了测试。

A.1 IEEE 算法

以下示例展示了一种用于检查浮点数十六进制表示的方法。注意，您还可以使用调试器来查看存储数据的十六进制表示。

以下 C 程序打印 π 的双精度近似值以及单精度无穷大：

代码样例 A-1 双精度示例

```
#include <math.h>
#include <sunmath.h>

int main() {
       union {
              float          flt;
              unsigned       un;
       } r;
       union {
              double       dbl;
              unsigned     un[2];
       } d;

       /* double precision */
```

代码样例 A-1 双精度示例 (续)

```
        d.dbl = M_PI;
        (void) printf("DP Approx pi = %08x %08x = %18.17e \n",
               d.un[0], d.un[1], d.dbl);

        /* single precision */
        r.flt = infinityf();
        (void) printf("Single Precision %8.7e :%08x \n",
               r.flt, r.un);

        return 0;
}
```

在基于 SPARC 的系统上，上一程序的输出类似以下内容：

```
DP Approx pi = 400921fb 54442d18 = 3.14159265358979312e+00
Single Precision Infinity:7f800000
```

以下 Fortran 程序打印每种格式的最小正规数：

代码样例 A-2 打印每种格式的最小正规数

```
        program print_ieee_values
c
c the purpose of the implicit statements is to ensure
c that the floatingpoint pseudo-intrinsic functions
c are declared with the correct type
c
        implicit real*16 (q)
        implicit double precision (d)
        implicit real (r)
        real*16           z
        double precision   x
        real                r
c
        z = q_min_normal()
        write(*,7) z, z
 7      format('min normal, quad:',1pe47.37e4,/,' in hex ',z32.32)
c
        x = d_min_normal()
```

代码样例 A-2 打印每种格式的最小正规数 (续)

```
      write(*,14) x, x
 14   format('min normal, double:',1pe23.16,' in hex ',z16.16)
c
      r = r_min_normal()
      write(*,27) r, r
 27   format('min normal, single:',1pe14.7,' in hex ',z8.8)
c
      end
```

在基于 SPARC 的系统上，相应的输出为：

```
min normal, quad:3.3621031431120935062626778173217526026D-4932
 in hex 00010000000000000000000000000000
min normal, double:2.2250738585072014-308 in hex 0010000000000000
min normal, single:1.1754944E-38 in hex 00800000
```

A.2 数学库

本节介绍使用数学库中函数的示例。

A.2.1 随机数生成器

以下示例调用随机数生成器来生成一个数字数组，并使用计时函数来测量计算给定数的 EXP 所花的时间：

代码样例 A-3 随机数生成器

```
#ifdef DP
#define GENERIC double precision
#else
#define GENERIC real
#endif
#define SIZE 400000
```

代码样例 A-3 随机数生成器 (续)

```
       program example
c
       implicit GENERIC (a-h,o-z)
       GENERIC x(SIZE), y, lb, ub
       real tarray(2), u1, u2
c
c compute EXP on random numbers in [-ln2/2,ln2/2]
       lb = -0.3465735903
       ub = 0.3465735903
c
c generate array of random numbers
#ifdef DP
       call d_init_addrans()
       call d_addrans(x,SIZE,lb,ub)
#else
       call r_init_addrans()
       call r_addrans(x,SIZE,lb,ub)
#endif
c
c start the clock
       call dtime(tarray)
       u1 = tarray(1)
c
c compute exponentials

       do 16 i=1,SIZE
           y = exp(x(i))
 16    continue
c
c get the elapsed time
       call dtime(tarray)
       u2 = tarray(1)
       print *,'time used by EXP is ',u2-u1
       print *,'last values for x and exp(x) are ',x(SIZE),y
c
       call flush(6)
       end
```

要编译上一示例，请将源代码放在后缀为 `F`（而不是 `f`）的文件中，以使编译器自动调用预处理程序，并在命令行中指定 `-DSP` 或 `-DDP` 以选择单精度或双精度。

本示例说明如何使用 d_addrans 生成在用户指定的范围内均匀分布的随机数据块：

代码样例 A-4 使用 d_addrans

```
/*
 * test SIZE*LOOPS random arguments to sin in the range
 * [0, threshold] where
 * threshold = 3E30000000000000 (3.72529029846191406e-09)
 */

#include <math.h>
#include <sunmath.h>

#define SIZE 10000
#define LOOPS 100
int main()
{
       double x[SIZE], y[SIZE];
       int    i, j, n;
       double lb, ub;
       union {
           unsigned      u[2];
           double        d;
       }  upperbound;

       upperbound.u[0] = 0x3e300000;
       upperbound.u[1] = 0x00000000;

       /* initialize the random number generator */
       d_init_addrans_();

       /* test (SIZE * LOOPS) arguments to sin */
       for (j = 0; j < LOOPS; j++) {

              /*
              * generate a vector, x, of length SIZE,
              * of random numbers to use as
              * input to the trig functions.
              */
              n = SIZE;
              ub = upperbound.d;
              lb = 0.0;
```

代码样例 A-4 使用 d_addrans(续)

```
                d_addrans_(x, &n, &lb, &ub);

                for (i = 0; i < n; i++)
                     y[i] = sin(x[i]);

                /* is sin(x) == x?It ought to, for tiny x. */
                for (i = 0; i < n; i++)
                    if (x[i] != y[i])
                            printf(
                            " OOPS:%d sin(%18.17e)=%18.17e \n",
                            i, x[i], y[i]);
        }
        printf(" comparison ended; no differences\n");
        ieee_retrospective_();
        return 0;
}
```

A.2.2 IEEE 建议的函数

此 Fortran 示例使用 IEEE 标准建议的某些函数：

代码样例 A-5 IEEE 建议的函数

```
c
c       Demonstrate how to call 5 of the more interesting IEEE
c       recommended functions from Fortran.These are implemented
c       with "bit-twiddling", and so are as efficient as you could
c       hope.The IEEE standard for floating-point arithmetic
c       doesn't require these, but recommends that they be
c       included in any IEEE programming environment.
c
c       For example, to accomplish
c           y = x * 2**n,
c       since the hardware stores numbers in base 2,
c       shift the exponent by n places.
c

c       Refer to
c       ieee_functions(3m)
c       libm_double(3f)
c       libm_single(3f)
```

代码样例 A-5 IEEE 建议的函数 (续)

```
c
c      The 5 functions demonstrated here are:
c
c      ilogb(x): returns the base 2 unbiased exponent of x in
c             integer format
c      signbit(x):returns the sign bit, 0 or 1
c      copysign(x,y): returns x with y's sign bit
c      nextafter(x,y): next representable number after x, in
c             the direction y
c      scalbn(x,n): x * 2**n
c
c      function       double precision      single precision
c      ---------------------------------------------------------
c      ilogb(x)       i = id_ilogb(x)       i = ir_ilogb(r)
c      signbit(x)     i = id_signbit(x)     i = ir_signbit(r)
c      copysign(x,y)  x = d_copysign(x,y)   r = r_copysign(r,s)
c      nextafter(x,y) z = d_nextafter(x,y)  r = r_nextafter(r,s)
c      scalbn(x,n)    x = d_scalbn(x,n)     r = r_scalbn(r,n)
       program ieee_functions_demo
       implicit double precision (d)
       implicit real (r)
       double precision   x, y, z, direction
       real               r, s, t, r_direction
       integer            i, scale

       print *
       print *, 'DOUBLE PRECISION EXAMPLES:'
       print *

       x = 32.0d0
       i = id_ilogb(x)
       write(*,1) x, i
 1     format(' The base 2 exponent of ', F4.1, ' is ', I2)

       x = -5.5d0
       y = 12.4d0
       z = d_copysign(x,y)
       write(*,2) x, y, z
 2    format(F5.1, ' was given the sign of ', F4.1,
     *   ' and is now ', F4.1)


       x = -5.5d0
       i = id_signbit(x)
       print *, 'The sign bit of ', x, ' is ', i
```

代码样例 A-5 IEEE 建议的函数 (续)

```
      x = d_min_subnormal()
      direction = -d_infinity()
      y = d_nextafter(x, direction)
      write(*,3) x
3     format(' Starting from ', 1PE23.16E3,
     -   ', the next representable number ')
      write(*,4) direction, y
4     format('    towards ', F4.1, ' is ', 1PE23.16E3)

      x = d_min_subnormal()
      direction = 1.0d0
   y = d_nextafter(x, direction)
      write(*,3) x
      write(*,4) direction, y
      x = 2.0d0
      scale = 3
      y = d_scalbn(x, scale)
      write (*,5) x, scale, y
5     format(' Scaling ', F4.1, ' by 2**', I1, ' is ', F4.1)
      print *
      print *, 'SINGLE PRECISION EXAMPLES:'
      print *

      r = 32.0
      i = ir_ilogb(r)
      write (*,1) r, i

      r = -5.5
      i = ir_signbit(r)
      print *, 'The sign bit of ', r, ' is ', i

      r = -5.5
      s = 12.4
      t = r_copysign(r,s)
      write (*,2) r, s, t

      r = r_min_subnormal()
      r_direction = -r_infinity()
      s = r_nextafter(r, r_direction)
      write(*,3) r
      write(*,4) r_direction, s

      r = r_min_subnormal()
      r_direction = 1.0e0
      s = r_nextafter(r, r_direction)
      write(*,3) r
```

代码样例 A-5 IEEE 建议的函数 (续)

```
        write(*,4) r_direction, s

        r = 2.0
        scale = 3
        s = r_scalbn(r, scale)
        write (*,5) r, scale, y

        print *
        end
```

代码样例 A-6 中显示了此程序的输出。

代码样例 A-6 代码样例 A-5 的输出

```
DOUBLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
-5.5 was given the sign of 12.4 and is now  5.5
The sign bit of    -5.5 is   1
Starting from  4.9406564584124654E-324, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  4.9406564584124654E-324, the next representable
   number towards  1.0 is  9.8813129168249309E-324
Scaling  2.0 by 2**3 is 16.0

SINGLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
The sign bit of    -5.5 is   1
-5.5 was given the sign of 12.4 and is now  5.5
Starting from  1.4012984643248171E-045, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  1.4012984643248171E-045, the next representable
   number towards  1.0 is  2.8025969286496341E-045
Scaling  2.0 by 2**3 is 16.0
```

如果使用 `f95` 编译器并带有 `-f77` 兼容性选项，则显示以下附加的消息。

```
Note:IEEE floating-point exception flags raised:
   Inexact; Underflow;
See the Numerical Computation Guide, ieee_flags(3M)
```

A.2.3 IEEE 特殊值

此 C 程序调用几个 ieee_values(3m) 函数：

```
#include <math.h>
#include <sunmath.h>

int main()
{
       double           x;
       float            r;

       x = quiet_nan(0);
       printf("quiet NaN:%.16e = %08x %08x \n",
              x, ((int *) &x)[0], ((int *) &x)[1]);

       x = nextafter(max_subnormal(), 0.0);
       printf("nextafter(max_subnormal,0) = %.16e\n",x);
       printf("                           = %08x %08x\n",
              ((int *) &x)[0], ((int *) &x)[1]);

       r = min_subnormalf();
       printf("single precision min subnormal = %.8e = %08x\n",
              r, ((int *) &r)[0]);

       return 0;
}
```

请记住：在链接时要同时指定 -lsunmath 和 -lm。

在基于 SPARC 的系统上，输出类似如下内容：

```
quiet NaN:NaN = 7fffffff ffffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                           = 000fffff fffffffe
single precision min subnormal = 1.40129846e-45 = 00000001
```

因为 x86 体系结构是 “小尾数法”，所以 x86 上的输出略有不同 （双精度数十六进制表示的高位字元和低位字元的顺序颠倒过来）：

```
quiet NaN:NaN = ffffffff 7fffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                           = fffffffe 000fffff
single precision min subnormal = 1.40129846e-45 = 00000001
```

使用 ieee_values 函数的 Fortran 程序应注意声明这些函数类型：

```
       program print_ieee_values
c
c the purpose of the implicit statements is to insure
c that the floatingpoint pseudo-instrinsic
c functions are declared with the correct type
c
       implicit real*16 (q)
       implicit double precision (d)
       implicit real (r)
       real*16 z, zero, one
       double precision   x
       real                r
c
       zero = 0.0
       one = 1.0
       z = q_nextafter(zero, one)
       x = d_infinity()
       r = r_max_normal()
c
       print *, z
       print *, x
       print *, r
c
       end
```

在基于 SPARC 的系统上，输出如下：

```
  6.47517511943802511092443895822764 66-4966
  Inf
  3.40282E+38
```

A.2.4 ieee_flags －舍入方向

以下示例说明如何将舍入模式设置为**向零舍入**：

```
#include <math.h>
#include <sunmath.h>

int main()
{
       int i;
       double      x, y;
       char        *out_1, *out_2, *dummy;

       /* get prevailing rounding direction */
       i = ieee_flags("get", "direction", "", &out_1);

       x = sqrt(.5);
       printf("With rounding direction %s, \n", out_1);
       printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
              ((int *) &x)[0], ((int *) &x)[1], x);

       /* set rounding direction */
       if (ieee_flags("set", "direction", "tozero", &dummy) != 0)
              printf("Not able to change rounding direction!\n");
       i = ieee_flags("get", "direction", "", &out_2);

       x = sqrt(.5);
       /*
        * restore original rounding direction before printf, since
        * printf is also affected by the current rounding
direction
        */
       if (ieee_flags("set", "direction", out_1, &dummy) != 0)
              printf("Not able to change rounding direction!\n");
       printf("\nWith rounding direction %s,\n", out_2);
       printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
              ((int *) &x)[0], ((int *) &x)[1], x);

       return 0;
}
```

(SPARC) 此简短程序的输出显示向零舍入的效果：

```
demo% cc rounding_direction.c -lsunmath -lm
demo%  a.out
With rounding direction nearest,
sqrt(.5) = 0x3fe6a09e 0x667f3bcd  = 7.071067811865476e-01

With rounding direction tozero,
sqrt(.5) = 0x3fe6a09e 0x667f3bcc  = 7.071067811865475e-01
demo%
```

(x86) 此简短程序的输出显示向零舍入的效果：

```
demo% cc rounding_direction.c -lsunmath -lm
demo%  a.out
With rounding direction nearest,
sqrt(.5) = 0x667f3bcd 0x3fe6a09e  = 7.071067811865476e-01

With rounding direction tozero,
sqrt(.5) = 0x667f3bcc 0x3fe6a09e  = 7.071067811865475e-01
demo%
```

从 Fortran 程序中将舍入方向设置为向零舍入：

```
program ieee_flags_demo
character*16       out

i = ieee_flags('set', 'direction', 'tozero', out)
if (i.ne.0) print *, 'not able to set rounding direction'

i = ieee_flags('get', 'direction', '', out)
print *, 'Rounding direction is:', out

end
```

输出如下：

```
demo% f95 ieee_flags_demo.f
demo%  a.out
 Rounding direction is:tozero
```

如果使用 f95 编译器并带有 -f77 兼容性选项来编译程序，则输出包含以下附加的消息。

```
demo% f95 -f77 ieee_flags_demo.f
ieee_flags_demo.f:
 MAIN ieee_flags_demo:
demo%  a.out
 Rounding direction is:tozero
 Note:Rounding direction toward zero
 See the Numerical Computation Guide, ieee_flags(3M)
```

A.2.5 C99 浮点环境函数

下一个示例说明如何使用某些 C99 浮点环境函数。norm 函数计算矢量的欧几里得范数，并使用环境函数来处理下溢和溢出。主程序使用缩放的矢量调用此函数以确保发生下溢和溢出 （如回溯性诊断输出所示）。

代码样例 A-7 C99 浮点环境函数

```
#include <stdio.h>
#include <math.h>
#include <sunmath.h>
#include <fenv.h>

/*
*  Compute the euclidean norm of the vector x avoiding
*  premature underflow or overflow
*/
double norm(int n, double *x)
{
    fenv_t env;
    double  s, b, d, t;
    int     i, f;

    /* save the environment, clear flags, and establish nonstop
       exception handling */
    feholdexcept(&env);

    /* attempt to compute the dot product x.x */
    d = 1.0; /* scale factor */
    s = 0.0;
```

代码样例 A-7 C99 浮点环境函数 (续)

```
    for (i = 0; i < n; i++)
        s += x[i] * x[i];

    /* check for underflow or overflow */
    f = fetestexcept(FE_UNDERFLOW | FE_OVERFLOW);
    if (f & FE_OVERFLOW) {
        /* first attempt overflowed, try again scaling down */
        feclearexcept(FE_OVERFLOW);
        b = scalbn(1.0, -640);
        d = 1.0 / b;
        s = 0.0;
        for (i = 0; i < n; i++) {
            t = b * x[i];
            s += t * t;
        }
    }
    else if (f & FE_UNDERFLOW && s < scalbn(1.0, -970)) {
        /* first attempt underflowed, try again scaling up */
        b = scalbn(1.0, 1022);
        d = 1.0 / b;
        s = 0.0;
        for (i = 0; i < n; i++) {
            t = b * x[i];
            s += t * t;
        }
    }

    /* hide any underflows that have occurred so far */
    feclearexcept(FE_UNDERFLOW);

    /* restore the environment, raising any other exceptions
       that have occurred */
    feupdateenv(&env);

    /* take the square root and undo any scaling */
    return d * sqrt(s);
}

int main()
{
    double x[100], l, u;
    int    n = 100;
```

代码样例 A-7 C99 浮点环境函数 (续)

```
    fex_set_log(stdout);

    l = 0.0;
    u = min_normal();
    d_lcrans_(x, &n, &l, &u);
    printf("norm:%g\n", norm(n, x));
    l = sqrt(max_normal());
    u = l * 2.0;
    d_lcrans_(x, &n, &l, &u);
    printf("norm:%g\n", norm(n, x));

    return 0;
}
```

在基于 SPARC 的系统上，编译和运行此程序可生成以下消息：

```
demo% cc norm.c -lsunmath -lm
demo%  a.out
Floating point underflow at 0x000153a8 __d_lcrans_, nonstop mode
  0x000153b4  __d_lcrans_
  0x00011594  main
Floating point underflow at 0x00011244 norm, nonstop mode
  0x00011248  norm
  0x000115b4  main
norm:1.32533e-307
Floating point overflow at 0x00011244 norm, nonstop mode
  0x00011248  norm
  0x00011660  main
norm:2.02548e+155
```

代码样例 A-8 显示 `fesetprec` 函数在基于 x86 的系统上的效果。（此函数不能在基于 SPARC 的系统上使用。）`while` 循环试图通过查找可在加 1 时完全舍入的 2 的最大幂来确定可用的精度。正如第一个循环所示，此方法在类似基于 x86 的系统的体系结构上并不总能达到预期的效果，此类体系结构以扩展精度来计算所有中间的结果。因此，可以使用 `fesetprec` 函数来保证将所有结果舍入到所需的精度 （如第二个循环所示）。

代码样例 A-8 `fesetprec` 函数 (x86)

```
#include <math.h>
#include <fenv.h>
```

代码样例 A-8 fesetprec 函数 (x86)(续)

```
int main()
{
    double x ;

    x = 1.0;
    while (1.0 + x != 1.0)
        x *= 0.5;
    printf("%d significant bits\n", -ilogb(x));

    fesetprec(FE_DBLPREC);
    x = 1.0;
    while (1.0 + x != 1.0)
        x *= 0.5;
    printf("%d significant bits\n", -ilogb(x));

    return 0;
}
```

在 x86 系统上，此程序的输出为：

```
64 significant bits
53 significant bits
```

最后，代码样例 A-9 说明了一种在多线程程序中使用环境函数的方法，以将父线程中的浮点模式传播到子线程中，并在子线程和父线程合并时恢复在子线程中引发的异常标记。（有关编写多线程程序的详细信息，请参见 《Solaris 多线程编程指南》。）

代码样例 A-9 在多线程程序中使用环境函数

```
#include <thread.h>
#include <fenv.h>

fenv_t env;

void child(void *p)
{
    /* inherit the parent's environment on entry */
    fesetenv(&env);
    ...
    /* save the child's environment before exit */
```

代码样例 A-9 在多线程程序中使用环境函数 (续)

```
    fegetenv(&env);
}

void parent()
{
    thread_t tid;
    void *arg;
    ...
    /* save the parent's environment before creating the child */
    fegetenv(&env);
    thr_create(NULL, NULL, child, arg, NULL, &tid);
    ...
    /* join with the child */
    thr_join(tid, NULL, &arg);
    /* merge exception flags raised in the child into the
       parent's environment */
    fex_merge_flags(&env);
    ...
}
```

A.3 异常和异常处理

A.3.1 ieee_flags — 产生的异常

通常，用户程序**检查**或**清除**产生的异常位。代码样例 A-10 是一个检查产生的异常标记的 C 程序。

代码样例 A-10 检查产生的异常标记

```
#include <sunmath.h>
#include <sys/ieeefp.h>

int main()
{
       int     code, inexact, division, underflow, overflow,
invalid;
```

代码样例 A-10 检查产生的异常标记 (续)

```
        double  x;
        char    *out;

        /* cause an underflow exception */
        x = max_subnormal() / 2.0;

        /* this statement insures that the previous */
        /* statement is not optimized away          */
        printf("x = %g\n",x);

        /* find out which exceptions are raised */
        code = ieee_flags("get", "exception", "", &out);

        /* decode the return value */
        inexact =       (code >> fp_inexact)     & 0x1;
        underflow =     (code >> fp_underflow)   & 0x1;
        division =      (code >> fp_division)    & 0x1;
        overflow =      (code >> fp_overflow)    & 0x1;
        invalid =       (code >> fp_invalid)     & 0x1;

      /* "out" is the raised exception with the highest priority */
        printf(" Highest priority exception is:%s\n", out);
        /* The value 1 means the exception is raised, */
        /* 0 means it isn't.                          */
        printf("%d %d %d %d %d\n", invalid, overflow, division,
            underflow, inexact);
        ieee_retrospective_();
        return 0;
}
```

运行此程序的输出为：

```
demo%  a.out
x = 1.11254e-308
 Highest priority exception is:underflow
0 0 0 1 1
 Note:IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 See the Numerical Computation Guide, ieee_flags(3M)
```

可以从 Fortran 中执行相同的操作：

代码样例 A-11 检查产生的异常标记－ Fortran

```
/*
A Fortran example that:
    *  causes an underflow exception
    *  uses ieee_flags to determine which exceptions are raised
    *  decodes the integer value returned by ieee_flags
    *  clears all outstanding exceptions
Remember to save this program in a file with the suffix .F, so that
the c preprocessor is invoked to bring in the header file
floatingpoint.h.
*/
#include <floatingpoint.h>

    program decode_accrued_exceptions
    double precision   x
    integer            accrued, inx, div, under, over, inv
    character*16       out
    double precision   d_max_subnormal
c Cause an underflow exception
    x = d_max_subnormal() / 2.0

c Find out which exceptions are raised
    accrued = ieee_flags('get', 'exception', '', out)

c Decode value returned by ieee_flags using bit-shift intrinsics
    inx   = and(rshift(accrued, fp_inexact)  , 1)
    under = and(rshift(accrued, fp_underflow), 1)
    div   = and(rshift(accrued, fp_division) , 1)
    over  = and(rshift(accrued, fp_overflow) , 1)
    inv   = and(rshift(accrued, fp_invalid)  , 1)

c The exception with the highest priority is returned in "out"
    print *, "Highest priority exception is ", out

c The value 1 means the exception is raised; 0 means it is not
    print *, inv, over, div, under, inx

c Clear all outstanding exceptions
    i = ieee_flags('clear', 'exception', 'all', out )
    end
```

输出如下：

```
 Highest priority exception is underflow
   0  0  0  1  1
```

虽然极少使用用户程序来*设置*异常标记，但的确可以这样做。下面的 C 示例就说明了这一点。

```
#include <sunmath.h>

int main()
{
       int     code;
       char    *out;

       if (ieee_flags("clear", "exception", "all", &out) != 0)
       printf(" could not clear exceptions\n");
       if (ieee_flags("set", "exception", "division", &out) != 0)
           printf("could not set exception\n");
       code = ieee_flags("get", "exception", "", &out);
       printf("out is:%s , fp exception code is:%X \n",
           out, code);

       return 0;
}
```

在 SPARC 上，以上程序的输出为：

```
out is:division , fp exception code is: 2
```

在 x86 上，输出为：

```
out is:division , fp exception code is: 4
```

A.3.2 ieee_handler 一捕获异常

注 – 以下示例仅适用于 Solaris 操作系统。

以下是一个 Fortran 程序，它安装一个信号处理程序以查找异常（仅用于基于 SPARC 系统）：

代码样例 A-12 捕获下溢－ SPARC

```
       program demo

c declare signal handler function
       external fp_exc_hdl
       double precision   d_min_normal
       double precision   x

c set up signal handler
       i = ieee_handler('set', 'common', fp_exc_hdl)
       if (i.ne.0) print *, 'ieee trapping not supported here'

c cause an underflow exception (it will not be trapped)
       x = d_min_normal() / 13.0
       print *, 'd_min_normal() / 13.0 = ', x

c cause an overflow exception
c the value printed out is unrelated to the result
       x = 1.0d300*1.0d300
       print *, '1.0d300*1.0d300 = ', x

       end
c
c the floating-point exception handling function
c
       integer function fp_exc_hdl(sig, sip, uap)
       integer sig, code, addr
       character label*16
c
c The structure /siginfo/ is a translation of siginfo_t
c from <sys/siginfo.h>
c

       structure /fault/
           integer address
       end structure

       structure /siginfo/
           integer si_signo
```

代码样例 A-12 捕获下溢－ SPARC(续)

```
            integer si_code
            integer si_errno
            record /fault/ fault
        end structure

        record /siginfo/ sip

c See <sys/machsig.h> for list of FPE codes
c Figure out the name of the SIGFPE
        code = sip.si_code
        if (code.eq.3) label = 'division'
        if (code.eq.4) label = 'overflow'
        if (code.eq.5) label = 'underflow'
        if (code.eq.6) label = 'inexact'
        if (code.eq.7) label = 'invalid'
        addr = sip.fault.address

c Print information about the signal that happened
        write (*,77) code, label, addr
 77     format ('floating-point exception code ', i2, ',',
     *       a17, ',', ' at address ', z8 )

        end
```

输出为：

```
 d_min_normal() / 13.0 =     1.7115952757748-309
floating-point exception code  4, overflow       , at address
1131C
 1.0d300*1.0d300 =     1.0000000000000+300
 Note:IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
    ieee_handler(3M)
```

(SPARC) 以下是一个更复杂的 C 示例：

代码样例 A-13 捕获无效值、被 0 除、溢出、下溢和不准确的值 — SPARC

```
/*
 * Generate the 5 IEEE exceptions:invalid, division,
 * overflow, underflow and inexact.
 *
 * Trap on any floating point exception, print a message,
 * and continue.
 *
 * Note that you could also inquire about raised exceptions by
 *    i = ieee("get","exception","",&out);
 * where out contains the name of the highest exception
 * raised, and i can be decoded to find out about all the
 * exceptions raised.
 */

#include <sunmath.h>
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>

extern void trap_all_fp_exc(int sig, siginfo_t *sip,
          ucontext_t *uap);

int main()
{
       double  x, y, z;
       char    *out;

       /*
        * Use ieee_handler to establish "trap_all_fp_exc"
        * as the signal handler to use whenever any floating
        * point exception occurs.
        */

       if (ieee_handler("set", "all", trap_all_fp_exc) != 0)
           printf(" IEEE trapping not supported here.\n");
       /* disable trapping (uninteresting) inexact exceptions */
       if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
               printf("Trap handler for inexact not cleared.\n");
```

代码样例 A-13 捕获无效值、被 0 除、溢出、下溢和不准确的值－ SPARC(续)

```
        /* raise invalid */
        if (ieee_flags("clear", "exception", "all", &out) != 0)
                printf(" could not clear exceptions\n");
        printf("1. Invalid:signaling_nan(0) * 2.5\n");
        x = signaling_nan(0);
        y = 2.5;
        z = x * y;

        /* raise division */
        if (ieee_flags("clear", "exception", "all", &out) != 0)
            printf(" could not clear exceptions\n");
        printf("2. Div0:1.0 / 0.0\n");
        x = 1.0;
        y = 0.0;
        z = x / y;

        /* raise overflow */
        if (ieee_flags("clear", "exception", "all", &out) != 0)
                printf(" could not clear exceptions\n");
        printf("3. Overflow:-max_normal() - 1.0e294\n");
        x = -max_normal();
        y = -1.0e294;
        z = x + y;

        /* raise underflow */
        if (ieee_flags("clear", "exception", "all", &out) != 0)
                printf(" could not clear exceptions\n");
        printf("4. Underflow:min_normal() * min_normal()\n");
        x = min_normal();
        y = x;
        z = x * y;

        /* enable trapping on inexact exception */
        if (ieee_handler("set", "inexact", trap_all_fp_exc) != 0)
                printf("Could not set trap handler for inexact.\
n");
        /* raise inexact */
        if (ieee_flags("clear", "exception", "all", &out) != 0)
                printf(" could not clear exceptions\n");
        printf("5. Inexact:2.0 / 3.0\n");
        x = 2.0;
        y = 3.0;
```

代码样例 A-13 捕获无效值、被 0 除、溢出、下溢和不准确的值－ SPARC(续)

```
        z = x / y;

        /* don't trap on inexact */
        if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
                printf(" could not reset inexact trap\n");

        /* check that we're not trapping on inexact anymore */
        if (ieee_flags("clear", "exception", "all", &out) != 0)
                printf(" could not clear exceptions\n");
        printf("6. Inexact trapping disabled; 2.0 / 3.0\n");
        x = 2.0;
        y = 3.0;
        z = x / y;

        /* find out if there are any outstanding exceptions */
        ieee_retrospective_();

        /* exit gracefully */
        return 0;
}

void trap_all_fp_exc(int sig, siginfo_t *sip, ucontext_t *uap) {
        char    *label = "undefined";

/* see /usr/include/sys/machsig.h for SIGFPE codes */
        switch (sip->si_code) {
        case FPE_FLTRES:
                label = "inexact";
                break;
        case FPE_FLTDIV:
                label = "division";
                break;
        case FPE_FLTUND:
                label = "underflow";
                break;
        case FPE_FLTINV:
                label = "invalid";
                break;
        case FPE_FLTOVF:
                label = "overflow";
                break;
        }
```

代码样例 A-13 捕获无效值、被 0 除、溢出、下溢和不准确的值－ SPARC(续)

```
       printf(
       " signal %d, sigfpe code %d:%s exception at address %x\n",
              sig, sip->si_code, label, sip-
>_data._fault._addr);
}
```

输出类似于以下内容：

```
1. Invalid:signaling_nan(0) * 2.5
   signal 8, sigfpe code 7:invalid exception at address 10da8
2. Div0: 1.0 / 0.0
   signal 8, sigfpe code 3:division exception at address 10e44
3. Overflow:-max_normal() - 1.0e294
   signal 8, sigfpe code 4:overflow exception at address 10ee8
4. Underflow:min_normal() * min_normal()
   signal 8, sigfpe code 5:underflow exception at address 10f80
5. Inexact: 2.0 / 3.0
   signal 8, sigfpe code 6:inexact exception at address 1106c
6. Inexact trapping disabled; 2.0 / 3.0
Note:IEEE floating-point exception traps enabled:
   underflow; overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_handler(3M)
```

(SPARC) 代码样例 A-14 说明如何使用 `ieee_handler` 和包含文件来修改某些异常情况的默认结果：

代码样例 A-14 修改异常情况的默认结果

```
/*
 * Cause a division by zero exception and use the
 * signal handler to substitute MAXDOUBLE (or MAXFLOAT)
 * as the result.
 *
 * compile with the flag -Xa
 */

#include <values.h>
#include <siginfo.h>
#include <ucontext.h>
```

代码样例 A-14 修改异常情况的默认结果 (续)

```
void division_handler(int sig, siginfo_t *sip, ucontext_t *uap);

int main() {
       double            x, y, z;
       float             r, s, t;
       char              *out;

       /*
        * Use ieee_handler to establish division_handler as the
        * signal handler to use for the IEEE exception division.
        */
       if (ieee_handler("set","division",division_handler)!=0) {
          printf(" IEEE trapping not supported here.\n");
       }

       /* Cause a division-by-zero exception */
       x = 1.0;
       y = 0.0;
       z = x / y;

       /*
        * Check to see that the user-supplied value, MAXDOUBLE,
        * is indeed substituted in place of the IEEE default
        * value, infinity.
        */
       printf("double precision division:%g/%g = %g \n",x,y,z);

       /* Cause a division-by-zero exception */
       r = 1.0;
       s = 0.0;
       t = r / s;

       /*
        * Check to see that the user-supplied value, MAXFLOAT,
        * is indeed substituted in place of the IEEE default
        * value, infinity.
        */
       printf("single precision division:%g/%g = %g \n",r,s,t);

       ieee_retrospective_();

       return 0;
```

代码样例 A-14 修改异常情况的默认结果 (续)

```
}

void division_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
       int                         inst;
       unsigned                    rd, mask, single_prec=0;
       float                       f_val = MAXFLOAT;
       double                      d_val = MAXDOUBLE;
       long                        *f_val_p = (long *) &f_val;

       /* Get instruction that caused exception. */
       inst = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

       /*
        * Decode the destination register.Bits 29:25 encode the
        * destination register for any SPARC floating point
        * instruction.
        */
       mask = 0x1f;
       rd = (mask & (inst >> 25));

       /*
        * Is this a single precision or double precision
        * instruction?Bits 5:6 encode the precision of the
        * opcode; if bit 5 is 1, it's sp, else, dp.
        */

       mask = 0x1;
       single_prec = (mask & (inst >> 5));

       /* put user-defined value into destination register */
       if (single_prec) {
           uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[rd] =
               f_val_p[0];
       } else {
       uap->uc_mcontext.fpregs.fpu_fr.fpu_dregs[rd/2] = d_val;
       }
}
```

正如所预期的一样，输出为：

```
double precision division:1/0 = 1.79769e+308
single precision division:1/0 = 3.40282e+38
Note:IEEE floating-point exception traps enabled:
   division by zero;
See the Numerical Computation Guide, ieee_handler(3M)
```

A.3.3 ieee_handler — 出现异常时终止

可以使用 ieee_handler 强制程序在出现某些浮点异常时终止：

```
#include <floatingpoint.h>
       program abort
c
       ieeer = ieee_handler('set', 'division', SIGFPE_ABORT)
       if (ieeer .ne.0) print *, ' ieee trapping not supported'
       r = 14.2
       s = 0.0
       r = r/s
c
       print *, 'you should not see this; system should abort'
c
       end
```

A.3.4 libm 异常处理功能

以下示例说明如何使用 libm 提供的某些异常处理功能。第一个示例基于以下任务：给定一个数 x 及系数 $a_0, a_1, ..., a_N$ 和 $b_0, b_1, ..., b_{N-1}$，计算函数 $f(x)$ 及其一阶导数 $fi(x)$，其中 f 是连续的分数

$$f(x) = a_0 + b_0/(x + a_1 + b_1/(x + .../(x + a_{N-1} + b_{N-1}/(x + a_N))...))。$$

在 IEEE 算法中，f 计算是非常容易的：即使一个中间除法的结果出现上溢或被零除异常，标准指定的默认值（符号正确的无穷大）仍然会得出正确的结果。在另一方面，fi 计算则要更困难一些，因为它的最简单的计算形式也可能包含可删除的奇点。如果计算遇到其中的一个奇点，它就会试图计算不定式 0/0, 0* 无穷大或无穷大 / 无穷大之一，所有这些式子均引发无效运算异常。W. Kahan 提出了一种通过称为“预替换”的功能来处理这些异常的方法。

预替换是 IEEE 默认异常响应的扩展，用户可以使用它提前指定用于替换异常运算结果的值。通过使用 libm 中的异常处理工具，程序可在 FEX_CUSTOM 异常处理模式下安装处理程序以方便地实现预替换。这种模式允许处理程序只需简单地使用以下方法即可为异

常运算结果提供任何值：通过将该值存储在传递给处理程序的 *info* 参数所指向的数据结构即可实现。以下示例程序使用 FEX_CUSTOM 处理程序实现的预替换来计算连续分数及其导数。

代码样例 A-15 使用 FEX_CUSTOM 处理程序计算连续分数及其导数

```
#include <stdio.h>
#include <sunmath.h>
#include <fenv.h>
volatile double p;
void handler(int ex, fex_info_t *info)
{
    info->res.type = fex_double;
    if (ex == FEX_INV_ZMI)
        info->res.val.d = p;
    else
        info->res.val.d = infinity();
}

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double         x, double *pf, double *pf1)
{
    fex_handler_t    oldhdl; /* for saving/restoring handlers */
    volatile double  t;
    double           f, f1, d, d1, q;
    int j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    fex_set_handling(FEX_DIVBYZERO, FEX_NONSTOP, NULL);
    fex_set_handling(FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI,
        FEX_CUSTOM, handler);

    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
```

代码样例 A-15 使用 FEX_CUSTOM 处理程序计算连续分数及其导数 (续)

```
        q = b[j] / d;
        /* the following assignment to the volatile variable t
           is needed to maintain the correct sequencing between
           assignments to p and evaluation of f1 */
        t = f1 = (-d1 / d) * q;
        p = b[j-1] * d1 / b[j];
        f = a[j] + q;
    }

    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}

/* For the following coefficients, x = -3, 1, 4, and 5 will all
   encounter intermediate exceptions */
double a[] = { -1.0, 2.0, -3.0, 4.0, -5.0 };
double b[] = { 2.0, 4.0, 6.0, 8.0 };

int main()
{
    double  x, f, f1;
    int i;

    feraiseexcept(FE_INEXACT); /* prevent logging of inexact */
    fex_set_log(stdout);
    fex_set_handling(FEX_COMMON, FEX_ABORT, NULL);
    for (i = -5; i <= 5; i++) {
        x = i;
        continued_fraction(4, a, b, x, &f, &f1);
        printf("f(% g) = %12g, f'(% g) = %12g\n", x, f, x, f1);
    }
    return 0;
}
```

有关程序的一些注释按顺序给出。在入口处，函数 continued_fraction 保存被零除和所有无效运算异常的当前异常处理模式。它然后为被零除设置不间断的异常处理，并为三个不定式设置 FEX_CUSTOM 处理程序。此处理程序将 0/0 和无穷大 / 无穷大替换为无穷大，但将 0* 无穷大替换为全局变量 p 的值。注意，每次通过计算该函数的循环时，必须重新计算 p 以提供正确的值来替换后面 0* 无穷大的无效运算。还要注意，必须将 p 声明为 volatile 以防止编译器将其删除，因为在循环的其他地方没有显式地提到它。最后，为防止编译器将 p 赋值移到可能发生异常（p 为其提供预替换值）的计算的上面或下面，还将计算结果赋值给 volatile 变量（在程序中称为 t）。 fex_setexcepthandler 的最终调用恢复被零除和无效运算的原始处理模式。

主程序通过调用 fex_set_log 函数来启用对回顾诊断的记录。在执行此操作之前，它引发不准确标记；这对防止记录不准确异常有效。（让我们回想一下，正如第 23 页的 “回顾诊断”中所解释的一样，在 FEX_NONSTOP 模式下，如果引发异常标记，则不记录该异常。）主程序还为一般异常设置 FEX_ABORT 模式，以确保任何 continued_fraction 没有显式处理的不常见异常都导致程序终止。最后，程序在几个不同的点处计算某个连续分数。正如以下示例输出所示，计算确实遇到中间异常：

```
f(-5) =     -1.59649,   f'(-5) =      -0.1818
f(-4) =     -1.87302,   f'(-4) =    -0.428193
Floating point division by zero at 0x08048dbe continued_fraction,
nonstop mode
  0x08048dc1  continued_fraction
  0x08048eda  main
Floating point invalid operation (inf/inf) at 0x08048dcf
continued_fraction, handler: handler
  0x08048dd2  continued_fraction
  0x08048eda  main
Floating point invalid operation (0*inf) at 0x08048dd2
continued_fraction, handler: handler
  0x08048dd8  continued_fraction
  0x08048eda  main
f(-3) =           -3,   f'(-3) =     -3.16667
f(-2) = -4.44089e-16,   f'(-2) =     -3.41667
f(-1) =     -1.22222,   f'(-1) =    -0.444444
f( 0) =     -1.33333,   f'( 0) =     0.203704
f( 1) =           -1,   f'( 1) =     0.333333
f( 2) =    -0.777778,   f'( 2) =      0.12037
f( 3) =    -0.714286,   f'( 3) =    0.0272109
f( 4) =    -0.666667,   f'( 4) =     0.203704
f( 5) =    -0.777778,   f'( 5) =    0.0185185
```

（在 x = 1、4 和 5 时计算 *fi(x)* 的过程中发生的异常不会导致回顾诊断消息，因为在程序中它们与 x = -3 时发生的异常在同一位置。

对于在计算连续分数及其导数时发生的异常，上一个程序可能并不是最有效的异常处理方法。一个原因是，每次迭代循环时都必须重新计算预替换值，而无论是否需要重新进行计算。在这种情况下，预替换值的计算涉及浮点除法，在新型 SPARC 和 x86 处理器上，浮点除法是相对较慢的运算。另外，循环本身还涉及两种除法，由于大多数 SPARC 和 x86 处理器无法重叠执行两个不同的除法运算，所以除法可能是循环中的瓶颈；再增加一个除法必将加剧瓶颈现象。

可以重新编写循环以便只需要一种除法，特别是预替换值的计算不需要包含除法。（要按这种方式重新编写循环，用户必须预先计算 `b` 数组中系数的相邻元素之间的比率。）这可消除多个除法运算的瓶颈，但不会消除预替换值计算所产生的所有算术运算。再者，由于需要将预替换值和要预替换的运算结果赋值给 `volatile` 变量，因而需要进行额外的内存运算，这些运算可降低程序的速度。虽然这些赋值是防止编译器重新排序某些关键运算所必需的，但它们也可以有效地防止编译器重新排序其他无关运算。因此，本示例中通过预替换来处理异常的方式要求进行额外的内存运算，并防止某些本来有可能执行的优化。可以更有效地处理这些异常吗？

如果缺少快速预替换的特殊硬件支持，则本示例中异常的最有效的处理方法可能是使用标记 （如以下版本所示）：

代码样例 A-16 使用标记处理异常

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

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double         x, double *pf, double *pf1)
{
    fex_handler_t  oldhdl;
    fexcept_t      oldinvflag;
    double         f, f1, d, d1, pd1, q;
    int j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);
    fegetexceptflag(&oldinvflag, FE_INVALID);

    fex_set_handling(FEX_DIVBYZERO | FEX_INV_ZDZ | FEX_INV_IDI |
        FEX_INV_ZMI, FEX_NONSTOP, NULL);
    feclearexcept(FE_INVALID);
```

代码样例 A-16 使用标记处理异常 (续)

```
    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        f1 = (-d1 / d) * q;
        f = a[j] + q;
    }

    if (fetestexcept(FE_INVALID)) {
        /* recompute and test for NaN */
        f1 = pd1 = 0.0;
        f = a[N];
        for (j = N - 1; j >= 0; j--) {
            d = x + f;
            d1 = 1.0 + f1;
            q = b[j] / d;
            f1 = (-d1 / d) * q;
            if (isnan(f1))
                f1 = b[j] * pd1 / b[j+1];
            pd1 = d1;
            f = a[j] + q;
        }
    }

    fesetexceptflag(&oldinvflag, FE_INVALID);
    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}
```

在此版本中，第一个循环试图在默认不间断模式下计算 *f(x)* 和 *f'(x)*。如果引发无效标记，则第二个循环重新计算 *f(x)* 和 *f'(x)* 以显式地测试是否出现 NaN。通常，不会发生任何无效运算异常，因此程序只执行第一个循环。该循环不引用 `volatile` 变量，并且不执行额外的算术运算，因此，它可达到编译器提供的最大运行速度。要获得此效率的代价就是，用户必须编写与第一个循环基本相同的第二个循环以处理发生异常时的情况。这种折衷办法是浮点异常处理。

A.3.5 在 Fortran 程序中使用 libm 异常处理

主要在 C/C++ 程序中使用 libm 异常处理工具，但通过使用 Sun Fortran 语言互操作性功能，您也可以在 Fortran 程序中调用某些 libm 函数。

注 – 为了获得一致的行为，请不要在相同程序中同时使用 libm 异常处理函数以及 ieee_flags 和 ieee_handler 函数。

以下示例显示 Fortran 版本的程序，它使用预替换来计算连续分数及其导数（仅限 SPARC）：

代码样例 A-17 使用预替换来计算连续分数及其导数— SPARC

```
c
c Presubstitution handler
c
      subroutine handler(ex, info)

      structure /fex_numeric_t/
          integer type
          union
          map
              integer            i
          end map

          map
              integer*8 l
          end map
          map
              real f
          end map
          map
              real*8 d
          end map
          map
              real*16 q
          end map
          end union
      end structure
```

代码样例 A-17 使用预替换来计算连续分数及其导数－ SPARC(续)

```
      structure /fex_info_t/
          integer op, flags
          record /fex_numeric_t/ op1, op2, res
      end structure

      integer ex
      record /fex_info_t/ info

      common /presub/ p
      double precision  p, d_infinity
      volatile          p

c 4 = fex_double; see <fenv.h> for this and other constants
      info.res.type = 4

c x'80' = FEX_INV_ZMI
      if (loc(ex) .eq. x'80') then
          info.res.d = p
      else
          info.res.d = d_infinity()
      endif
      return
      end

c
c Evaluate the continued fraction given by coefficients a(j) and
c b(j) at the point x; return the function value in f and the
c derivative in f1
c

      subroutine continued_fraction(n, a, b, x, f, f1)

      integer           n
      double precision  a(*), b(*), x, f, f1

      common            /presub/ p
      integer           j, oldhdl
      dimension        oldhdl(24)
      double precision d, d1, q, p, t
      volatile         p, t
```

代码样例 A-17 使用预替换来计算连续分数及其导数－ SPARC(续)

```
      external fex_getexcepthandler, fex_setexcepthandler
      external fex_set_handling, handler
c$pragma c(fex_getexcepthandler, fex_setexcepthandler)
c$pragma c(fex_set_handling)

c x'ff2' = FEX_DIVBYZERO | FEX_INVALID
      call fex_getexcepthandler(oldhdl, %val(x'ff2'))

c x'2' = FEX_DIVBYZERO, 0 = FEX_NONSTOP
      call fex_set_handling(%val(x'2'), %val(0), %val(0))

c x'b0' = FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI, 3 = FEX_CUSTOM
      call fex_set_handling(%val(x'b0'), %val(3), handler)

      f1 = 0.0d0
      f = a(n+1)
      do j = n, 1, -1
          d = x + f
          d1 = 1.0d0 + f1
          q = b(j) / d
          f1 = (-d1 / d) * q
c
c         the following assignment to the volatile variable t
c         is needed to maintain the correct sequencing between
c         assignments to p and evaluation of f1

          t = f1
          p = b(j-1) * d1 / b(j)
          f = a(j) + q
      end do

      call fex_setexcepthandler(oldhdl, %val(x'ff2'))
      return
      end
```

代码样例 A-17 使用预替换来计算连续分数及其导数－ SPARC(续)

```
c Main program
c
      program cf
      integer            i
      double precision   a, b, x, f, f1
      dimension          a(5), b(4)
      data a /-1.0d0, 2.0d0, -3.0d0, 4.0d0, -5.0d0/
      data b /2.0d0, 4.0d0, 6.0d0, 8.0d0/

      external fex_set_handling
c$pragma c(fex_set_handling)

c x'ffa' = FEX_COMMON, 1 = FEX_ABORT
      call fex_set_handling(%val(x'ffa'), %val(1), %val(0))
      do i = -5, 5
          x = dble(i)
          call continued_fraction(4, a, b, x, f, f1)
          write (*, 1) i, f, i, f1
      end do
    1 format('f(', I2, ') = ', G12.6, ', f''(', I2, ') = ', G12.6)
      end
```

此程序的输出为：

```
f(-5) = -1.59649    , f'(-5) = -.181800
f(-4) = -1.87302    , f'(-4) = -.428193
f( -3) = -3.00000    , f'( -3) = -3.16667
f(-2) = -.444089E-15, f'(-2) = -3.41667
f(-1) = -1.22222    , f'(-1) = -.444444
f( 0) = -1.33333    , f'( 0) = .203704
f( 1) = -1.00000    , f'( 1) = 0.333333
f( 2) = -.777778    , f'( 2) = .120370
f( 3) = -.714286    , f'( 3) = 0.272109E-01
f( 4) = -.666667    , f'( 4) = 0.203704
f( 5) = -.777778    , f'( 5) = 0.185185E-01
 Note:IEEE floating-point exception flags raised:
    Inexact;  Division by Zero;  Invalid Operation;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M)
```

A.4 杂项

A.4.1 sigfpe —捕获整数异常

上一节介绍了使用 ieee_handler 的示例。通常，当选择是使用 ieee_handler 还是 sigfpe 时，建议使用前者。

注 – sigfpe 仅可在 Solaris 操作系统中使用。

(SPARC) 在某些情况下（如捕获整数算术异常），sigfpe 是要使用的处理程序。代码样例 A-18 捕获整数被零除。

代码样例 A-18 捕获整数异常

```
/* Generate the integer division by zero exception */

#include <siginfo.h>
#include <ucontext.h>
#include <signal.h>

void int_handler(int sig, siginfo_t *sip, ucontext_t *uap);

int main() {
       int   a, b, c;

/*
 * Use sigfpe(3) to establish "int_handler" as the signal handler
 * to use on integer division by zero
 */

/*
 * Integer division-by-zero aborts unless a signal
 * handler for integer division by zero is set up
 */
       sigfpe(FPE_INTDIV, int_handler);

       a = 4;
       b = 0;
```

代码样例 A-18 捕获整数异常 (续)

```
        c = a / b;
        printf("%d / %d = %d\n\n", a, b, c);
        return 0;
}

void int_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
        printf("Signal %d, code %d, at addr %x\n",
            sig, sip->si_code, sip->_data._fault._addr);

/*
 * automatically for floating-point exceptions but not for
 * integer division by zero.
 */
        uap->uc_mcontext.gregs[REG_PC] =
            uap->uc_mcontext.gregs[REG_nPC];
}
```

A.4.2 从 C 中调用 Fortran

以下是一个调用 Fortran 子例程的 C 驱动程序的简单示例。有关使用 C 和 Fortran 的详细信息，请参见相应的 C 和 Fortran 手册。以下是 C 驱动程序（将其保存在名为 `driver.c` 的文件中）：

代码样例 A-19 从 C 中调用 FORTRAN

```
/*
 * a demo program that shows:
 * 1. how to call f95 subroutine from C, passing an array argument
 * 2. how to call single precision f95 function from C
 * 3. how to call double precision f95 function from C
 */

extern int      demo_one_(double *);
extern float    demo_two_(float *);
extern double   demo_three_(double *);

int main()
{
        double  array[3][4];
```

代码样例 A-19 从 C 中调用 FORTRAN(续)

```
        float  f, g;
        double x, y;
        int    i, j;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 4; j++)
                array[i][j] = i + 2*j;

        g = 1.5;
        y = g;

        /* pass an array to a fortran function (print the array) */
        demo_one_(&array[0][0]);
        printf(" from the driver\n");
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 4; j++)
                printf("    array[%d][%d] = %e\n",
                    i, j, array[i][j]);
            printf("\n");
        }

        /* call a single precision fortran function */
        f = demo_two_(&g);
        printf(
         " f = sin(g) from a single precision fortran function\n");
        printf("    f, g:%8.7e, %8.7e\n", f, g);
        printf("\n");

        /* call a double precision fortran function */
        x = demo_three_(&y);
        printf(
         " x = sin(y) from a double precision fortran function\n");
        printf("    x, y:%18.17e, %18.17e\n", x, y);

        ieee_retrospective_();
        return 0;
}
```

将 Fortran 子例程保存在名为 drivee.f 的文件中：

```
      subroutine demo_one(array)
      double precision array(4,3)
      print *, 'from the fortran routine:'
      do 10 i =1,4
          do 20 j = 1,3
              print *, '   array[', i, '][', j, '] = ', array(i,j)
 20   continue
      print *
 10   continue
      return
                  end

      real function demo_two(number)
      real number
      demo_two = sin(number)
      return
                  end

      double precision function demo_three(number)
      double precision number
      demo_three = sin(number)
      return
                  end
```

然后，执行编译和链接：

```
cc -c driver.c
f95 -c drivee.f
      demo_one:
      demo_two:
      demo_three:
f95 -o driver driver.o drivee.o
```

输出类似如下内容：

```
from the fortran routine:
   array[ 1 ][ 1 ] =  0.0E+0
   array[ 1 ][ 2 ] =  1.0
   array[ 1 ][ 3 ] =  2.0

   array[ 2 ][ 1 ] =  2.0
   array[ 2 ][ 2 ] =  3.0
   array[ 2 ][ 3 ] =  4.0

   array[ 3 ][ 1 ] =  4.0
   array[ 3 ][ 2 ] =  5.0
   array[ 3 ][ 3 ] =  6.0

   array[ 4 ][ 1 ] =  6.0
   array[ 4 ][ 2 ] =  7.0
   array[ 4 ][ 3 ] =  8.0

from the driver
   array[0][0] = 0.000000e+00
   array[0][1] = 2.000000e+00
   array[0][2] = 4.000000e+00
   array[0][3] = 6.000000e+00

   array[1][0] = 1.000000e+00
   array[1][1] = 3.000000e+00
   array[1][2] = 5.000000e+00
   array[1][3] = 7.000000e+00

   array[2][0] = 2.000000e+00
   array[2][1] = 4.000000e+00
   array[2][2] = 6.000000e+00
   array[2][3] = 8.000000e+00

f = sin(g) from a single precision fortran function
   f, g:9.9749500e-01, 1.5000000e+00

x = sin(y) from a double precision fortran function
   x, y:9.97494986604054446e-01, 1.50000000000000000e+00
```

A.4.3 有用的调试命令

表 A-1 显示用于 SPARC 体系结构的调试命令示例。

表 A-1 一些调试命令 (SPARC)

操作	dbx	adb
设置断点		
在函数中	`stop in myfunct`	`myfunct:b`
在行号中	`stop at 29`	
在绝对地址中		`23a8:b`
在相对地址中		`main+0x40:b`
一直运行，直到遇到断点	`run`	`:r`
检查源代码	`list`	`<pc,10?ia`
检查 fp 寄存器		
IEEE 单精度	`print $f0`	`<f0=X`
十进制值的等价值（十六进制）	`print -fx $f0`	`<f0=f`
IEEE 双精度	`print $f0f1`	`<f0=X; <f1=X`
十进制值的等价值（十六进制）	`print -flx $f0f1` `print -flx $d0`	`<f0=F`
检查所有 fp 寄存器	`regs -F`	`$x for f0-f15` `$X for f16-f31`
检查所有寄存器	`regs`	`$r; $x; $X`
检查 fp 状态寄存器	`print -fx $fsr`	`<fsr=X`
将单精度 1.0 保存在 `f0` 中	`assign $f0=1.0`	`3f800000>f0`
将双精度 1.0 保存在 `f0/f1` 中	`assign $f0f1=1.0`	`3ff00000>f0; 0>f1`
继续执行	`cont`	`:c`
单步	`step (or next)`	`:s`
退出调试器	`quit`	`$q`

表 A-2 显示用于 x86 体系结构的调试命令示例。

表 A-2 一些调试命令 (x86)

操作	dbx	adb
设置断点		
在函数中	`stop in myfunct`	`myfunct:b`
在行号中	`stop at 29`	
在绝对地址中		`23a8:b`
在相对地址中		`main+0x40:b`
一直运行，直到遇到断点	`run`	`:r`
检查源代码	`list`	`<pc,10?ia`

检查 fp 寄存器	`print $st0`
	`...`
	`print $st7`
		`$x`

检查所有寄存器	`examine &$gs/19X`	`$r`
检查 fp 状态寄存器	`examine &$fstat/X`	`<fstat=X` `or $x`
继续执行	`cont`	`:c`
单步	`step (or next)`	`:s`
退出调试器	`quit`	`$q`

以下示例介绍两种在代码开头（对应于 adb 中的例程 myfunction）设置断点的方法。第一个示例为：

```
myfunction:b
```

在第二个示例中，您可以确定与 myfunction 的代码段开头相对应的绝对地址，然后在该绝对地址中设置断点：

```
myfunction=X
                23a8
23a8:b
```

对 adb 而言，使用 f95 编译的 Fortran 程序中的主子例程称为 MAIN_。要在 adb 中的 MAIN_ 处设置断点：

```
MAIN_:b
```

在检查浮点寄存器的内容时， dbx 命令 regs -F 显示的十六进制值是以 16 为基数的表示法，而不是该数的十进制表示法。对于基于 SPARC 的系统，adb 命令 $x 和 $X 显示十六进制的表示法和十进制值。对于基于 x86 的系统，adb 命令 $x 只显示十进制值。对于基于 SPARC 的系统，双精度值在奇数寄存器旁边显示十进制值。

因为操作系统在进程第一次使用浮点单元之前总是禁用浮点单元，所以只有在被调试的程序访问浮点寄存器之后，您才可以修改这些寄存器。

(SPARC) 在显示浮点数时，应该牢记：寄存器的大小为 32 位，单精度浮点数占用 32 位（因此可将它放在一个寄存器中），而双精度浮点数占用 64 位 （因此，使用两个寄存器来保存双精度数）。在十六进制表示中，32 位对应于 8 个十六进制数字。在以下使用 adb 显示的 FPU 寄存器快照中，显示的组织形式如下：

<*fpu* 寄存器的名称 > <IEEE 十六进制值 > < 单精度 > < 双精度 >

(SPARC) 第三列保存显示在第二列中的十六进制模式的单精度十进制解释。第四列解释寄存器对。例如， f11 行的第四列将 f10 和 f11 解释为 64 位 IEEE 双精度数。

(SPARC) 因为 f10 和 f11 用于保存双精度值，所以 （在 f10 行上）将该值的前 32 位 7ff00000 解释为 +NaN 是不正确的。将所有 64 位 7ff00000 00000000 解释为 +Infinity 则有可能是有意义的转换。

(SPARC) 用于显示前 16 个浮点数据寄存器的 adb 命令 $x 还显示 fsr （浮点状态寄存器）：

```
$x
fsr       40020
f0     400921fb     +2.1426990e+00
f1     54442d18     +3.3702806e+12     +3.1415926535897931e+00
f2            2     +2.8025969e-45
f3            0     +0.0000000e+00     +4.2439915819305446e-314
f4     40000000     +2.0000000e+00
f5            0     +0.0000000e+00     +2.0000000000000000e+00
f6     3de0b460     +1.0971904e-01
f7            0     +0.0000000e+00     +1.2154188766544394e-10
f8     3de0b460     +1.0971904e-01
f9            0     +0.0000000e+00     +1.2154188766544394e-10
f10    7ff00000     +NaN
f11           0     +0.0000000e+00     +Infinity
f12    ffffffff     -NaN
f13    ffffffff     -NaN                -NaN
f14    ffffffff     -NaN
f15    ffffffff     -NaN                -NaN
```

(x86) x86 上的相应输出类似于以下内容：

```
$x
80387 chip is present.
cw      0x137f
sw      0x3920
cssel 0x17  ipoff 0x2d93             datasel 0x1f  dataoff 0x5740

 st[0]  +3.24999988079071044921875 e-1            VALID
 st[1]  +5.65391332434795490344196 88 e73         EMPTY
 st[2]  +2.00000000000000008881784197
EMPTY
 st[3]  +1.80732183080704405560160 47 e-1         EMPTY
 st[4]  +7.91803002357482910156 25 e-1          EMPTY
 st[5]  +4.20163903669390492723323 4 e-13         EMPTY
 st[6]  +4.20163903669390492723323 4 e-13         EMPTY
 st[7]  +2.72249992132186946491856 36
EMPTY
```

注 – *(x86)* cw 是控制字；而 sw 是状态字。

附录 B

SPARC 行为和实现

本章讨论与基于 SPARC® 工作站中所使用的浮点单元有关的问题，并介绍一种用来确定哪个代码生成标志最适合特定工作站的方法。

B.1 浮点硬件

本节列出许多 SPARC 浮点单元，并介绍它们所支持的指令集和异常处理功能。关于以下情况的信息，包括在捕获到异常时所发生情况的简要说明、捕获到的下溢和未捕获到的下溢之间的区别，以及建议的提供非 IEEE（非标准）算法模式的 SPARC 实现可能的行为过程，请参见《SPARC 结构手册》第 8 版的附录 N“SPARC IEEE 754 实现建议”和第 9 版的附录 B “SPARC 第 9 版的 IEEE 标准 754-1985 要求”。

表 B-1 列出了由 SPARC 工作站使用的硬件浮点实现。许多早期的基于 SPARC 系统都有浮点单元，这些单元源自由 TI 或 Weitek 开发的核心：

- TI 系列 – 包括 TI8847 和 TMS390C602A
- Weitek 系列 – 包括 1164/1165、 3170 和 3171

这两个系列的 FPU 都已经授权给其他工作站供应商，因此有可能在某些基于 SPARC 工作站中找到其他半导体制造商的芯片。下表也显示了其中的一些芯片 .

表 B-1 SPARC 浮点选项

FPU	说明或 处理器名称	适于机型	说明	最佳的 -xchip 和 -xarch
基于 Weitek 1164/1165 的 FPU 或没有 FPU	内核模拟浮点指令	已过时的 （早期的）	慢；不推荐	`-xchip=old` `-xarch=v7`
基于 TI 8847 的 FPU	TI 8847； Fujitsu 或 LSI 的控制器	Sun-4™/1xx Sun-4/2xx Sun-4/3xx Sun-4/4xx SPARCstation® 1 (4/60)	1989 大多数 SPARCstation 1 工作站都使用 Weitek 3170	`-xchip=old` `-xarch=v7`
基于 Weitek 3170 的 FPU		SPARCstation 1 (4/60) SPARCstation 1+ (4/65)	1989, 1990	`-xchip=old` `-xarch=v7`
TI 602a		SPARCstation 2 (4/75)	1990	`-xchip=old` `-xarch=v7`
基于 Weitek 3172 的 FPU		SPARCstation SLC (4/20) SPARCstation IPC (4/40)	1990	`-xchip=old` `-xarch=v7`
Weitek 8601 或 Fujitsu 86903	集成 CPU 和 FPU	SPARCstation IPX (4/50) SPARCstation ELC (4/25)	1991 IPX 使用 40 MHz CPU/FPU； ELC 使用 33 MHz	`-xchip=old` `-xarch=v7`
Cypress 602	驻留在 Mbus 模块	SPARCserver® 6xx	1991	`-xchip=old` `-xarch=v7`
TI TMS390S10 (STP1010)	microSPARC®-I	SPARCstation LX SPARCclassic	1992 硬件中没有 FsMULd	`-xchip=micro` `-xarch=v8a`
Fujitsu 86904 (STP1012)	microSPARC-II	SPARCstation 4 和 5 SPARCstation Voyager	硬件中没有 FsMULd	`-xchip=micro2` `-xarch=v8a`
TI TMS390Z50 (STP1020A)	SuperSPARC®-I	SPARCserver 6xx SPARCstation 10 SPARCstation 20 SPARCserver 1000 SPARCcenter 2000		`-xchip=super` `-xarch=v8`

表 B-1 SPARC 浮点选项（续）

FPU	说明或 处理器名称	适于机型	说明	最佳的 -xchip 和 -xarch
STP1021A	SuperSPARC-II	SPARCserver 6xx SPARCstation 10 SPARCstation 20 SPARCserver 1000 SPARCcenter 2000		`-xchip=super2` `-xarch=v8`
Ross RT620	hyperSPARC_	SPARCstation 10/HSxx SPARCstation 20/HSxx		`-xchip=hyper` `-xarch=v8`
Fujitsu 86907	TurboSPARC	SPARCstation 4 和 5		`-xchip=micro2` `-xarch=v8`
STP1030A	UltraSPARC® I	Ultra-1、 Ultra-2 Ex000	V9+VIS	`-xchip=ultra` `-xarch=v8plusa`
STP1031	UltraSPARC II	Ultra-2、 E450 Ultra-30、 Ultra-60、 Ultra-80、 Ex500 Ex000、 E10000	V9+VIS	`-xchip=ultra2` `-xarch=v8plusa`
SME1040	UltraSPARC IIi	Ultra-5、 Ultra-10	V9+VIS	`-xchip=ultra2i` `-xarch=v8plusa`
	UltraSPARC IIe	Sun Blade™ 100	V9+VIS	`-xchip=ultra2e` `-xarch=v8plusa`
	UltraSPARC III	Sun Blade 1000 Sun Blade 2000	V9+VIS II	`-xchip=ultra3` `-xarch=v8plusb*`

* 使用 `v8plusb` 生成的可执行文件只能在 UltraSPARC III 系统上运行。要在所有 UltraSPARC (I,II,III) 系统上运行，必须将 `-xarch` 设置为 `v8plusa`。

上表中的最后一列显示要用来为每个 FPU 获得最快代码的编译器标志。这些标志控制代码生成的两个独立属性：`-xarch` 标志确定编译器可以使用的指令集， `-xchip` 标志确定编译器在调度代码时将针对处理器的性能特点进行的假设。因为所有的 SPARC 浮点单元都至少实现在 《SPARC 结构手册》第 7 版中定义的浮点指令集，所以用 `-xarch=v7` 编译的程序将可以在任何基于 SPARC 系统上运行 （尽管它可能无法充分利用较新处理器的功能）。同样，用特定的 `-xchip` 值编译的程序能够运行于支持用 `-xarch` 指定的指令集的任何基于 SPARC 系统，但是如果在非指定处理器的系统上运行时，则速度会下降。

在上表中，排列在 microSPARC-I 前面的浮点单元实现了在《SPARC 结构手册》第 7 版中定义的浮点指令集；应当使用 `-xarch=v7` 来编译这些必须运行于含有这类浮点单元 (FPU) 的系统上的程序。编译器不对这些处理器的性能特点进行特殊假设，因此它们都共用一个 `-xchip` 选项 `-xchip=old`。（并非所有列在 表 B-1 中的系统都仍然受到编译器的支持；列出它们的目的仅在于展现其历史。有关可与支持这些系统的编译器一起使用的代码生成标志的信息，请参阅适当版本的 《数值计算指南》。）

microSPARC-I 和 microSPARC-II 浮点单元实现在《SPARC 结构手册》第 8 版中定义的浮点指令集 （FsMULd 和四倍精度指令除外）。用 `-xarch=v8` 编译的程序将在使用这些处理程序的系统上运行，但是由于未实现的浮点指令必须由系统内核模拟，因此大量使用 FsMULd 的程序（如执行大量单精度复数运算的 Fortran 程序）的性能可能会严重下降。为避免出现该问题，请用 `-xarch=v8a` 针对包含这些处理程序的系统编译程序。

SuperSPARC-I、SuperSPARC-II、hyperSPARC 和 TurboSPARC 浮点单元实现在《SPARC 结构手册》第 8 版中定义的浮点指令集 （四倍精度指令除外）。要在包含这些处理器的系统上获得最佳性能，请使用 `-xarch=v8` 进行编译。

UltraSPARC I、UltraSPARC II、UltraSPARC IIe、UltraSPARC IIi、UltraSPARC III、UltraSPARC IIIi、UltraSPARC IV 和 UltraSPARC IV+ 浮点单元实现在《SPARC 结构手册》第 9 版中定义的浮点指令集（四倍精度指令除外）；特别是，它们提供 32 个双精度浮点寄存器。为了允许编译器使用这些寄存器，请使用 `-xarch=v8plus`（对于在 32 位 OS 下运行的程序）或 `-xarch=v9`（对于在 64 位 OS 下运行的程序）进行编译。这些处理器还提供了标准指令集的扩展。其他被称作 Visual Instruction Set（可视指令集）或 VIS 的指令很少由编译器自动生成，但是它们可用在汇编代码中。因此，为了充分采用这些处理器所支持的指令集，请使用 `-xarch=v8plusa`（32 位）或 `-xarch=v9a`（64 位）。

可以使用 `-xtarget` 宏选项同时指定 `-xarch` 和 `-xchip` 选项。（即，`-xtarget` 标志只是扩展到 `-xarch`、`-xchip` 和 `-xcache` 标志的适当组合。）默认的代码生成选项是 `-xtarget=generic`。要查看更多的信息 （包括 `-xarch`、`-xchip` 和 `-xtarget` 值的完整列表），请参见 `cc`(1)、`CC`(1) 和 `f95`(1) 手册页以及编译器手册。其他 `-xarch` 信息在 《Fortran 用户指南》、《C 用户指南》和 《C++ 用户指南》中提供。

B.1.1 浮点状态寄存器和队列

对于所有的 SPARC 浮点单元来说，无论它们实现哪种版本的 SPARC 结构，它们都提供一个浮点状态寄存器 (FSR)，该寄存器中包含与 FPU 相关的状态位和控制位。所有实现延迟浮点捕获的 SPARC FPU 都提供一个浮点队列 (FQ)，该队列中包含有关当前执行的浮点指令的信息。 FSR 可由用户软件访问，以检测已经发生的浮点异常 , 并控制舍入方向、捕获和非标准的算术模式。FQ 由操作系统的内核使用，以便处理浮点捕获；用户软件通常不可访问它。

软件通过 `STFSR` 和 `LDFSR` 指令来访问浮点状态寄存器，这两个指令的作用分别是将 FSR 存储在内存中和从内存中加载它。在 SPARC 汇编语言中，这些指令按如下方式编写：

```
        st      %fsr, [addr]   ! 在指定地址存储 FSR
        ld      [addr], %fsr   ! 从指定地址加载 FSR
```

内联模板文件 `libm.il` 所在的目录中包含随 Sun Studio 编译器提供的库，该文件包含 `STFSR` 和 `LDFSR` 指令的用法示例。

图 B-1 显示浮点状态寄存器中位字段的布局。

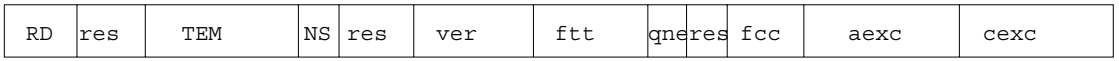


图 B-1 SPARC 浮点状态寄存器

在第 7 版本和第 8 版本的 SPARC 结构中，FSR 占用如上所示的 32 位。在第 9 版本中，FSR 扩展到 64 位，其中的低 32 位与该图相匹配；高 32 位大部分不使用，只包含了三个附加浮点条件代码字段。

在这里，`res` 是指保留位，`ver` 是用来标识 FPU 版本的只读字段，`ftt` 和 `qne` 由系统用来处理浮点捕获。其余字段将在下表中介绍 .

表 B-2 浮点状态寄存器字段

字段	包含
RM	舍入方向模式
TEM	捕获启用模式
NS	非标准模式
fcc	浮点条件代码
aexc	应计异常标志
cexc	当前的异常标志

`RM` 字段保留两个为浮点运算指定舍入方向的位。`NS` 位在实现它的 SPARC FPU 上启用非标准的算术模式；在其他系统上，该位将被忽略。`fcc` 字段保留由浮点比较指令生成的浮点条件代码并且由分支和条件移动运算使用。最后，`TEM`、`aexc` 和 `cexc` 字段包含五个位，这些位针对五个 IEEE 754 浮点异常中的每一个控制捕获并记录应计和当前异常标志。表 B-3 对这些字段进行了细分。

表 B-3 异常处理字段

字段	寄存器中的相应位				
`TEM`（捕获启用模式）	NVM	OFM	UFM	DZM	NXM
	27	26	25	24	23
`aexc`（应计异常标志）	nva	ofa	ufa	dza	nxa
	9	8	7	6	5
`cexc`（当前异常标志）	nvc	ofc	ufc	dzc	nxc
	4	3	2	1	0

（上面的符号 NV、OF、UF、DZ 和 NX 分别代表无效运算、上溢、下溢、被零除和不精确异常。）

B.1.2 需要软件支持的特殊类

在大多数情况下， SPARC 的浮点单元可以在硬件中完成指令的执行而无需软件的支持。但是，在以下四种情况下，硬件将无法成功地完成浮点指令：

- 浮点单元被禁用。
- 指令未被硬件所实现 （如，在基于 Weitek 1164/1165 的 FPU 上使用 `fsqrt[sd]`，在 microSPARC-I 和 microSPARC-II FPU 上使用 `fsmuld`，或者在任何 SPARC FPU 上使用四倍精度指令）。
- 硬件无法为指令操作数传送正确的结果。
- 指令将导致一个 IEEE 754 浮点异常 （已经启用了该异常的捕获）。

在每种情况下，最初的响应都是相同的：进程被 “捕获”到系统内核，由系统内核确定捕获的原因并采取相应的措施。（术语 “捕获”是指正常控制流的中断。）在前三种情况下，内核在软件中模拟被捕获的指令。请注意，模拟的指令也可以导致一个异常，如果对该异常的捕获已经启用。

在上面的前三种情况下，如果模拟的指令不导致 IEEE 浮点异常 （其捕获已经启用），则内核将完成该指令。如果该指令是浮点比较指令，内核将更新条件代码以反映结果；如果该指令是算术运算，内核将相应的结果传送到目标寄存器。内核还更新当前的异常标志以反映由该指令引发的任何 （未捕获）异常，并将这些异常与应计异常标志进行 “或”运算。内核随后安排继续从捕获点执行该进程。

当硬件执行或者内核软件模拟的某个指令导致一个 IEEE 浮点异常 （其捕获已经启用），则该指令将无法完成。目标寄存器、浮点条件代码和应计异常标志保持不变，当前异常标志设置为反映导致了该捕获的特定异常，内核向该进程发送一个 `SIGFPE` 信号。

下面的伪代码概述了浮点捕获的处理。请注意， `aexc` 字段通常只能由软件清除。

```
FPop provokes a trap;
if trap type is fp_disabled, unimplemented_FPop, or
  unfinished_FPop then
    emulate FPop;
texc ' all IEEE exceptions generated by FPop;
if (texc and TEM) = 0 then
    f[rd]  ' fp_result;   // 如果 fpop 是一个算术操作
    fcc ' fcc_result;  // 如果 fpop 是比较
    cexc ' texc;
    aexc ' (aexc or texc);
else
    cexc ' trapped IEEE exception generated by FPop;
    throw SIGFPE;
```

当内核必须模拟许多浮点指令时，程序的性能将严重下降。出现该问题的相对频率可能取决于多个因素 （肯定包括捕获类型）。

在正常情况下，每个进程中只应当出现一次 fp_disabled 捕获。系统内核禁用浮点单元，直至某个进程被首次启动时为止，因此由该进程执行的第一个浮点运算将导致捕获。在处理完该捕获之后，内核启用浮点单元，并使其在以后的进程过程中保持启用状态。（可以针对整个系统禁用浮点单元，但是不建议这样做，而且只在出于内核或硬件调试目时才这样做。）

无论浮点单元何时遇到它未实现的指令，显然都会出现 unimplemented_FPop 捕获。由于目前大多数的 SPARC 浮点单元都至少实现由《SPARC 结构手册》第 8 版定义的指令集（四倍精度指令除外），而且 Sun Studio 编译器不生成四倍精度指令，所以在大多数系统上应当不会出现这种类型的捕获。如上所述，两个明显的例外是 microSPARC-I 和 microSPARC-II 处理器，它们不实现 FsMULd 指令。为了避免在这些处理上出现 unimplemented_FPop 捕获，请用 -xarch=v8a 选项编译程序。

其余两个捕获类型 unfinished_FPop 和捕获的 IEEE 异常通常与涉及 NaN、无穷大和次正规数的特殊计算情况相关。

B.1.2.1 IEEE 浮点异常、 NaN 和无穷大

当某个浮点指令遇到一个 IEEE 浮点异常（其捕获已经启用）时，该指令将无法完成；相反，系统会向该进程发送一个 SIGFPE 信号。如果该进程已经建立了 SIGFPE 信号处理程序，则该处理程序将被调用，否则该进程将终止。多数程序不会产生被捕获的 IEEE 浮点异常，因为，捕获被启用的目的常常是在异常产生时终止程序，中止的方式是调用信号处理程序以打印消息并终止程序或者在未装信号处理程序的情况下借助于系统的默认行为。但是，正如第 4 章中所述，可以安排信号处理程序为捕获指令提供结果并继续执行。请注意，如果捕获到许多浮点异常并用这种方式进行处理，则性能可能严重下降。

即使捕获行为被禁用或者指令不导致异常（其捕获已经启用），大多数 SPARC 浮点单元也会在某些情况下产生捕获，至少在涉及无穷大、NaN 操作数或 IEEE 浮点异常的情况下是这样。当硬件不支持类似的特殊情况时，将出现该问题；相反，它将生成 unfinished_FPop 捕获并使内核模拟软件完成指令。不同的 SPARC FPU 对导致 unfinished_FPop 捕获的条件会有不同的反应：例如，大多数早期的 SPARC FPU 和 hyperSPARC FPU 会捕获所有的 IEEE 浮点异常而无论捕获是否被启用；在硬件无法确定某指令是否会导致某浮点异常并且该异常的捕获被启用的情况下，UltraSPARC FPU 将产生捕获。另一方面，SuperSPARC-I、SuperSPARC-II、TurboSPARC、microSPARC-I 和 microSPARC-II FPU 在硬件中处理各种异常情况，并且从不生成 unfinished_FPop 捕获。

由于大多数 unfinished_FPop 捕获都与浮点异常一起出现，因此程序可通过利用异常处理功能（即，测试异常标志、捕获和提交结果或者终止异常）来避免导致过量捕获。当然，必须认真平衡以下两个方面带来的成本：处理异常；允许异常导致 unfinished_FPop 捕获。

B.1.2.2 次正规数和非标准算法

SPARC 浮点单元产生 `unfinished_FPop` 捕获的大多数常见的情况都生成次正规数。当浮点运算生成次正规操作数或者必须生成一个非零的次正规结果（即，导致渐进下溢的结果）时，许多 SPARC FPU 将捕获。因为下溢较少见但是又很难通过编程来回避，而且因为下溢的中间结果的准确性对最终计算结果的影响很小，所以 SPARC 结构包括**非标准算法模式**，该模式为用户提供一种方法，用来避免出现与涉及次正规数的 `unfinished_FPop` 捕获相关的性能的下降。

SPARC 结构不能准确定义非标准算法模式；它只是声明当该模式处于启用状态时，支持它的处理器会生成不遵循 IEEE 754 标准的结果。但是，所有支持该模式的现有 SPARC 实现都使用它来禁用渐进下溢，并将所有的次正规操作数和结果替换为零。
（有一个例外：Weitek 1164/1165 FPU 只在非标准模式下将次正规结果刷新为零，它们不将次正规操作数视为零。）

并非所有的 SPARC 实现都提供非标准模式。特别是，SuperSPARC-I、SuperSPARC-II、TurboSPARC、microSPARC-I 和 microSPARC-II 浮点单元都处理次正规操作数并完全在硬件中生成次正规结果，因此，它们无需支持非标准算法。（对在这些处理器上启用非标准模式的任何尝试都将被忽略。）因此，渐进下溢在这些处理器上并不造成性能损失。

为了确定渐进下溢是否影响程序的性能，应当首先确定下溢是否确实发生，然后检查该程序占用了多少系统时间。为了确定是否出现下溢，可以使用数学库函数 `ieee_retrospective()` 来查看在程序退出时是否引发了下溢异常标志。在默认情况下，Fortran 程序调用 `ieee_retrospective()`。C 和 C++ 程序在退出之前需要显式调用 `ieee_retrospective()`。如果出现了任何下溢，`ieee_retrospective()` 会打印一则类似如下的消息：

```
注意：引发了 IEEE 浮点异常标记：
 不精确；下溢；
请参见 《数字计算指南》， ieee_flags(3M)
```

如果该程序遇到下溢，您可能希望通过用 `time` 命令来对该程序的执行进行计时，从而确定该程序占用多少系统时间。

```
demo% /bin/time myprog > myprog.output
305.3 real       32.4 user       271.9 sys
```

如果系统时间（上面显示的第三个数字）超长，则说明原因可能在于下溢数量太多。如果是这样的话，并且如果该程序不依赖渐进下溢的准确性，则可以启用非标准模式以获得更佳性能。可通过两种方法来完成此操作：第一，可以用 `-fns` 标志（这暗示着它是 `-fast` 和 `-fnonstd` 宏的一部分）进行编译，以便在程序启动时启用非标准模式。第二，增值的数学库 `libsunmath` 提供两个分别用来启用和禁用非标准模式的函数：调用

nonstandard_arithmetic() 会启用非标准模式（如果它受支持的话），而调用 standard_arithmetic() 会恢复 IEEE 行为。调用这些函数的 C 和 Fortran 语法如下所示：

C、C++	`nonstandard_arithmetic();` `standard_arithmetic();`
Fortran	`call nonstandard_arithmetic()` `call standard_arithmetic()`

注意 – 由于非标准算法模式与渐进下溢带来的准确性好处相矛盾，因此您在使用它时应当格外小心。要查看渐进下溢的更多信息，请参见第 2 章。

B.1.2.3 非标准算法和内核模拟

在实现非标准模式的 SPARC 浮点单元上，启用该模式会导致硬件将次正规操作数视为零并将次正规结果刷新为零。但是，用于模拟捕获的浮点指令的内核软件不实现非标准模式，其部分原因在于该模式的影响无法定义且依赖于实现，而且，与在软件中模拟浮点运算所带来的成本相比，处理渐进下溢所带来的附加成本可忽略不计。

如果将受到非标准模式影响的浮点运算被中断（例如，当出现上下文切换或者另一个浮点指令导致捕获时，已发出的运算将无法完成），它将由内核软件使用标准 IEEE 算法进行模拟。因此，在异常情况下，在非标准模式下运行的程序**可能**根据系统负荷生成稍有不同的结果。在实际中**未**发现这样的行为。它将只影响那些对以下情况非常敏感的程序：用渐进下溢还是用突然下溢执行极为罕见的特定运算。

B.2 fpversion(1) 函数 – 查找有关 FPU 的信息

用编译器分发的 fpversion 实用程序标识已安装的 CPU 并估计处理器和系统总线的时钟速度。 fpversion 通过解释由 CPU 和 FPU 存储的标识信息来确定 CPU 和 FPU 的类型。它循环地执行那些运行时间可以被预计的简单指令，并记录执行循环所占用的时间，从而估计时钟的速度。对该循环执行多次可增加计时测量的准确性。因此，fpversion 不是瞬间完成的；它可能需要运行几秒钟。

fpversion 还报告要用于主机系统的最佳 -xtarget 代码生成选项。

在 UltraSPARC IV+ 工作站上，fpversion 显示类似如下的信息。（这些信息可能因计时或机器配置的不同而异。）

```
demo% fpversion
 基于 SPARC 的 CPU 可用。
 CPU 时钟速率大约为 461.1 MHz。
 内核显示 CPU 时钟速率是 480.0 MHz。
 内核显示内存时钟速率是 120.0 MHz。

 找到 Sun-4 浮点控制器版本 0。
 UltraSPARC 芯片可用。
 FPU 频率大约为 492.7 MHz。

 使用 -xtarget=ultra2 -xcache=16/32/1:2048/64/1 代码 - 生成选项。

 Hostid = hardware_host_id
```

要查看更多的信息，请参见 fpversion(1) 手册页。

附录 C

x86 行为和实现

本附录介绍与基于 x86/x64 系统中所使用的浮点单元相关的 x86/x64 和 SPARC 兼容性问题。

基于 x86/x64 的硬件包括 Intel 生产的 Pentium™ 微处理器和较新的微处理器以及其他制造商生产的兼容微处理器，如 AMD Opteron 处理器。尽管已努力使 x86 与 SPARC 平台保持兼容，但是二者仍存在几点区别。

在基于 x86/x64 的系统上：

- x67 浮点寄存器的宽度为 80 位。因为使用 x87 浮点寄存器栈时算术计算的中间结果可能采用双扩展 （80 位）精度，故计算结果可能会有所不同。 `-fstore` 标志将最小化这些差异。但是，使用 `-fstore` 标志会导致性能下降。
- 每次将单精度或双精度浮点数加载到 x87 浮点寄存器栈或存储到内存中时，都会转换到双扩展 （80 位）精度或从双扩展精度进行转换。因此，加载和存储浮点数可能导致异常。
- 在使用 x87 浮点寄存器栈时，将通过微码协助在硬件中实现渐进下溢；没有非标准模式。
- 不提供 `fpversion` 实用程序。
- 双扩展 （80 位）格式接受某些不表示任何浮点值的位模式 （请参见表 2-8）。硬件通常将这些“不受支持的格式”视为信号 NaN，但是数学库在处理这样的表示形式时存在不一致。由于这些位模式不会由硬件生成，因此它们只能产生于无效的内存引用（例如在读取数组时发生越界），或者产生于显式地将内存中的数据从一种类型强制转换为另一种类型 （例如，通过 C 语言中的 union 数据结构）。因此，在大多数数值程序中，不出现这些位模式。

附录 D

What Every Computer Scientist Should Know About Floating-Point Arithmetic

注 – 本附录是对论文《What Every Computer Scientist Should Know About Floating-Point Arithmetic》（作者：David Goldberg，发表于 1991 年 3 月号的《Computing Surveys》）进行编辑之后的重印版本。

D.1 摘要

许多人认为浮点运算是一个深奥的主题。这相当令人吃惊，因为浮点在计算机系统中是普遍存在的。几乎每种语言都有浮点数据类型；从 PC 到超级计算机都有浮点加速器；多数编译器可随时进行编译浮点算法；而且实际上，每种操作系统都必须对浮点异常（如溢出）作出响应。本文将为您提供一个教程，涉及的方面包含对计算机系统设计人员产生直接影响的浮点运算信息。它首先介绍有关浮点表示和舍入误差的背景知识，然后讨论 IEEE 浮点标准，最后列举了许多示例来说明计算机生成器如何更好地支持浮点。

类别和主题描述符：（主要） C.0 [计算机系统组织]：概论 —— **指令集设计**； D.3.4 [程序设计语言]：处理器 —— **编译器**，**优化**； G.1.0 [数值分析]：概论 —— **计算机运算**，**错误分析**，**数值算法**（次要）

D.2.1 [软件工程]：要求 / 规范 —— **语言**； D.3.4 程序设计语言]：正式定义和理论 —— **语义**； D.4.1 操作系统]：进程管理 —— **同步**。

一般术语：算法，设计，语言

其他关键字 / 词：非规格化数值，异常，浮点，浮点标准，渐进下溢，保护数位，NaN，溢出，相对误差、舍入误差，舍入模式， ulp，下溢。

D.2 简介

计算机系统的生成器经常需要有关浮点运算的信息。但是，有关这方面的详细信息来源非常少。有关此主题的书目非常少，而且其中的一部《*Floating-Point Computation*》（作者：Pat Sterbenz）现已绝版。本文提供的教程包含与系统构建直接相关的浮点运算（以下简称为**浮点**）信息。它由三节组成（这三节并不完全相关）。第一节第 2 页的“舍入误差”讨论对加、减、乘、除基本运算使用不同舍入策略的含义。它还包含有关衡量舍入误差的两种方法 ulp 和相对误差的背景信息。第二节讨论 IEEE 浮点标准，该标准正被商业硬件制造商迅速接受。IEEE 标准中包括基本运算的舍入方法。对标准的讨论借助了第 2 页的“舍入误差”部分中的内容。第三节讨论浮点与计算机系统各个设计方面之间的关联。主题包括指令集设计、优化编译器和异常处理。

作者已尽力避免在不给出正当理由的情况下来声明浮点，这主要是因为证明将产生较为复杂的基本计算。对于那些不属于文章主旨的说明，已将其归纳到名为“详细信息”的章节中，您可以视实际情况选择跳过此节。另外，此节还包含了许多定理的证明。每个证明的结尾处均标记有符号 ∎。如果未提供证明，∎ 将紧跟在定理声明之后。

D.3 舍入误差

将无穷多位的实数缩略表示为有限位数需要使用近似。即使存在无穷多位整数，多数程序也可将整数计算的结果以 32 位进行存储。相反，如果指定一个任意的固定位数，那么多数实数计算将无法以此指定位数精准表示实际数量。因此，通常必须对浮点计算的结果进行舍入，以便与其有限表示相符。舍入误差是浮点计算所独有的特性。第 4 页的“相对误差和 Ulp”章节说明如何衡量舍入误差。

既然多数浮点计算都具有舍入误差，那么如果基本算术运算产生的舍入误差比实际需要大一些，这有没有关系？该问题是贯穿本章节的核心主题。第 5 页的“保护数位”章节讨论**保护**数位，它是一种减少两个相近的数相减时所产生的误差的方法。 IBM 认为保护数位非常重要，因此在 1968 年它将保护数位添加到 System/360 架构的双精度格式（其时单精度已具有保护数位），并更新了该领域中所有的现有计算机。下面两个示例说明了保护数位的效用。

IEEE 标准不仅仅要求使用保护数位。它提供了用于加、减、乘、除和平方根的算法，并要求实现产生与该算法相同的结果。因此，将程序从一台计算机移至另一台计算机时，如果这两台计算机均支持 IEEE 标准，那么基本运算的结果逐位相同。这大大简化了程序的移植过程。在第 10 页的 “精确舍入的运算”中介绍了此精确规范的其他用途。

D.3.1 浮点格式

已经提议了几种不同的实数表示法，但是到目前为止使用最广的是浮点表示法。[1] 浮点表示法有一个基数 β（始终假定其为偶数）和一个精度 p。如果 β = 10、p = 3，则将数 0.1 表示为 1.00×10^{-1}。如果 β = 2、p = 24，则无法准确表示十进制数 0.1，但是它近似为 $1.10011001100110011001101 \times 2^{-4}$。

通常，将浮点数表示为 $\pm d.dd\ldots d \times \beta^e$，其中 $d.dd\ldots d$ 称为**有效数字**[2]，它具有 p 个数字。更精确地说，$\pm d_0 . d_1 d_2 \ldots d_{p-1} \times \beta^e$ 表示以下数

$$\pm\left(d_0 + d_1\beta^{-1} + \ldots + d_{p-1}\beta^{-(p-1)}\right)\beta^e, (0 \le d_i < \beta)。 \quad (1)$$

术语**浮点数**用于表示一个实数，该实数可以未经全面讨论的格式准确表示。与浮点表示法相关联的其他两个参数是最大允许指数和最小允许指数，即 $e_{\max}$ 和 $e_{\min}$。由于存在 β^p 个可能的有效数字，以及 $e_{\max} - e_{\min} + 1$ 个可能的指数，因此浮点数可以按

$$[\log_2(e_{max} - e_{min} + 1)] + [\log_2(\beta^p)] + 1$$

位编码，其中最后的 +1 用于符号位。此时，精确编码并不重要。

有两个原因导致实数不能准确表示为浮点数。最常见的情况可以用十进制数 0.1 说明。虽然它具有有限的十进制表示，但是在二进制中它具有无限重复的表示。因此，当 β = 2 时，数 0.1 介于两个浮点数之间，而这两个浮点数都不能准确地表示它。一种较不常见的情况是实数超出范围；也就是说，其绝对值大于 $\beta \times \beta^{e_{max}}$ 或小于 $1.0 \times \beta^{e_{min}}$。本文的大部分内容讨论第一种原因导致的问题。然而，超出范围的数将在第 20 页的“无穷”和第 22 页的 “反向规格化的数”章节中进行讨论。

浮点表示不一定是唯一的。例如，0.01×10^1 和 1.00×10^{-1} 都表示 0.1。如果前导数字不是零（在上面的等式 (1) 中，$d_0 \neq 0$），那么该表示称为**规格化**。浮点数 1.00×10^{-1} 是规格化的，而 0.01×10^1 则不是。当 β = 2、p = 3、$e_{\min}$ = -1 且 $e_{\max}$ = 2 时，有 16 个规格化浮点数，如图 D-1 所示。粗体散列标记对应于其有效数字是 1.00 的数。要求浮点表示为规格化，则可以使该表示唯一。遗憾的是，此限制将无法表示零！表示 0 的一种自然方法是使用 $1.0 \times \beta^{e_{min}-1}$，因为 这样做保留了以下事实：非负实数的数值顺序对应于其浮点表示的词汇顺序。[3] 将指数存储在 k 位字段中时，意味着只有 2^k - 1 个值可用作指数，因为必须保留一个值来表示 0。

请注意，浮点数中的 × 是表示法的一部分，这与浮点乘法运算不同。× 符号的含义通过上下文来看应该是明确的。例如，表达式 $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$ 仅产生单个浮点乘法。

---

1. 其他表示法的示例有**浮动斜线**和**有符号对数** [Matula 和 Kornerup 1985； Swartzlander 和 Alexopoulos 1975]。
2. 此术语由 Forsythe 和 Moler [1967] 提出，现已普遍替代了旧术语 *mantissa*。
3. 这假定采用通常的排列方式，即指数存储在有效数字的左侧。

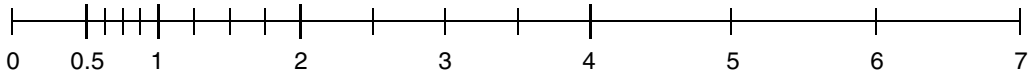


图 D-1 $\beta = 2$、$p = 3$、$e_{min} = -1$、$e_{max} = 2$ 时规格化的数

D.3.2 相对误差和 Ulp

因为舍入误差是浮点计算的固有特性，所以需要有一种方法来衡量此误差，这一点很重要。请考虑使用 $\beta = 10$ 且 $p = 3$ 的浮点格式（此格式在本节中广泛使用）。如果浮点计算的结果是 3.12×10^{-2}，并且以无限精度计算的结果是 .0314，那么您可以清楚地看到最后一位存在 2 单位的误差。同样，如果将实数 .0314159 表示为 3.14×10^{-2}，那么将在最后一位存在 .159 单位的误差。通常，如果浮点数 $d.d\ldots d \times \beta^e$ 用于表示 z，那么将在最后一位存在 $|d.d\ldots d - (z/\beta^e)|\beta^{p-1}$ 单位的误差。[4, 5] 术语 *ulp* 将用作“最后一位上的单位数”的简写。如果某个计算的结果是最接近于正确结果的浮点数，那么它仍然可能存在 .5 ulp 的误差。衡量浮点数与它所近似的实数之间差值的另一种方法是**相对误差**，它只是用两数之差除以实数的商。例如，当 3.14159 近似为 3.14×10^0 时产生的相对误差是 $.00159/3.14159 \approx .0005$。

要计算对应于 .5 ulp 的相对误差，请注意当实数用可能最接近的浮点数 $d.dd...dd \times \beta^e$ 近似表示时，误差可达到 $0.00...00\beta^e \times \beta^e$，其中 β' 是数字 $\beta/2$，在浮点数的有效数字中有 p 单位，在误差的有效数字中有 p 单位个 0。此误差是 $((\beta/2)\beta^{-p}) \times \beta^e$。因为形式为 $d.dd\ldots dd \times \beta^e$ 的数都具有相同的绝对误差，但具有介于 β^e 和 $\beta \times \beta^e$ 之间的值，所以相对误差介于 $((\beta/2)\beta^{-p}) \times \beta^e/\beta^e$ 和 $((\beta/2)\beta^{-p}) \times \beta^e/\beta^{e+1}$ 之间。即，

$$\frac{1}{2}\beta^{-p} \le \frac{1}{2}\mathrm{ulp} \le \frac{\beta}{2}\beta^{-p} \tag{2}$$

特别是，对应于 .5 ulp 的相对误差可能随因子 β 的不同而有所变化。此因子称为**浮动系数**。将 $\varepsilon = (\beta/2)\beta^{-p}$ 设置为上面 (2) 中的上限，表明：将一个实数舍入为最接近的浮点数时，相对误差总是以 ε（称为**机器 ε**）为界限。

在上例中，相对误差是 $.00159/3.14159 \approx .0005$。为了避免此类较小数，通常将相对误差表示为一个因子乘以 ε 的形式，在此例中 $\varepsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = .005$。因此，相对误差将表示为 $(.00159/3.14159)/.005)\ \varepsilon \approx 0.1\varepsilon$。

以实数 $x = 12.35$ 为例说明 ulp 与相对误差之间的差异。将其近似为 $\tilde{x} = 1.24 \times 10^1$。误差是 0.5 ulp，相对误差是 0.8ε。接下来，将考虑有关数字 8 的计算 $\tilde{x}$。精确值是 $8x = 98.8$，而计算值是 $8\tilde{x} = 9.92 \times 10^1$。误差是 4.0 ulp，相对误差仍然是 0.8ε。尽管相对误差保持不变，但以 ulp 衡量的误差却是原来的 9 倍。通常，当基数为 β 时，以 ulp 表示的固定相对误差可按最大因子 β 进行浮动。反之，如上述公式 (2) 所示，固定误差 .5 ulp 导致相对误差可按 β 进行浮动。

4. 除非数 z 大于 $\beta^{e_{max}}+1$ 或小于 $\beta^{e_{min}}$。否则在另行通知之前，将不会考虑以此方式超出范围的数。

5. 假设 z' 是近似 z 的浮点数。那么，$|d.d\ldots d - (z/\beta^e)|\beta^{p-1}$ 将等价于 $|z'-z|/\mathrm{ulp}(z')$。衡量误差的更精确公式是 $|z'-z|/\mathrm{ulp}(z)$。– 编辑者

衡量舍入误差的最常用方法是以 ulp 表示。例如，舍入为最接近的浮点数相当于一个小于或等于 .5 ulp 的误差。但是，在分析由各种公式导致的舍入误差时，使用相对误差是一种较好的方法。在第 38 页的“证明”章节中的分析很好地说明了这一点。由于浮动因子 β 的关系， ε 可能会过高估计舍入为最接近浮点数的效果，所以在使用较小 β 的计算机上，对公式的误差估计更为精确。

在仅关心舍入误差的大小顺序时，ulp 和 ε 可以互换使用，因为它们只是在因子 β 上有差异。例如，在浮点数的误差为 n ulp 时，这意味着受影响的位数是 $\log_\beta n$。如果某个计算的相对误差是 $n\varepsilon$，那么

受影响的数字是 $\approx \log_\beta n$。 (3)

D.3.3 保护数位

计算两个浮点数之差的一种方法是：精确计算二者之差，然后将其舍入为最接近的浮点数。如果两个操作数的大小差别非常大，那么系统开销将也随之上升。假定 $p = 3$， $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ 将计算为

$x = 2.15 \times 10^{12}$
$y = .000000000000000125 \times 10^{12}$
$x - y = 2.1499999999999999875 \times 10^{12}$

它舍入为 2.15×10^{12}。浮点硬件通常按固定位数执行运算，而不是使用所有这些数字。假定保留位数是 p，并且在右移较小的操作数时，只是舍弃数字（与舍入相对）。那么，$2.15 \times 10^{12} - 1.25 \times 10^{-5}$ 将变为

$x = 2.15 \times 10^{12}$
$y = 0.00 \times 10^{12}$
$x - y = 2.15 \times 10^{12}$

结果是完全相同的，就好像先对差值进行精确计算，然后再进行舍入。请看另一个示例：10.1 - 9.93。它将变成

$x = 1.01 \times 10^{1}$
$y = 0.99 \times 10^{1}$
$x - y = .02 \times 10^{1}$

正确结果是 .17，因此计算差值的误差是 30 ulp，而且每一个数字均不正确：误差可以达到多大程度？

D.3.3.1 定理 1

使用带有参数 β 和 p 的浮点格式，并使用 p 数字计算差值，结果的相对误差可能与 $\beta - 1$ 一样大。

D.3.3.2 证明

当 $x = 1.00\ldots0$ 和 $y = .\rho\rho\ldots\rho$ 时，表达式 $x - y$ 中的相对误差为 $\beta - 1$，其中 $\rho = \beta - 1$。此处，y 具有 p 数字（均等于 ρ）。精确差值是 $x - y = \beta^{-p}$。然而，如果仅使用 p 数字计算结果，那么位于 y 最右侧的数字将被舍弃，因此计算差值是 β^{-p+1}。因此，误差是 $\beta^{-p} - \beta^{-p+1} = \beta^{-p}(\beta - 1)$，相对误差是 $\beta^{-p}(\beta - 1)/\beta^{-p} = \beta - 1$。∎

当 $\beta=2$ 时，相对误差可能与结果一样大；当 $\beta=10$ 时，相对误差可能是结果的 9 倍。换言之，当 $\beta=2$ 时，公式 (3) 显示受影响的位数是 $\log_2(1/\varepsilon) = \log_2(2^p) = p$。即，结果中所有 p 数字都是错误的！假定再添加一个数字（**保护数位**）以避免发生这种情况。即，将较小数截断为 $p + 1$ 数字，然后将相减的结果舍入为 p 数字。使用保护数位后，上例将变为

$x = 1.010 \times 10^1$
$y = 0.993 \times 10^1$
$x - y = .017 \times 10^1$

并且结果是精确的。使用单个保护数位时，结果的相对误差可能大于 ε（例如位于 110 – 8.59 中）。

$x = 1.10 \times 10^2$
$y = .085 \times 10^2$
$x - y = 1.015 \times 10^2$

它舍入为 102，与正确结果 101.41 相比，相对误差为 .006，大于 $\varepsilon = .005$。通常，结果的相对误差只能比 ε 稍大。更为精确的方法是，

D.3.3.3 定理 2

如果 x 和 y 均是使用参数 β 和 p 形式表示的浮点数，并且使用 $p + 1$ 数字（即，一个保护数位）执行减法操作，那么结果中的相对舍入误差将小于 2ε。

此定理将在第 38 页的“舍入误差”中证明。在上述定理中包括了加法运算，因为 x 和 y 可以是正数或负数。

D.3.4 抵消

可以这样总结上一节：在不使用保护数位的情况下，如果在两个相近数之间执行减法，那么产生的相对误差会非常大。换言之，对包含减法运算（或在具有相反符号的数量之间进行加法运算）的任何表达式求值均可能产生较大的相对误差，以至**所有**数字将变为无意义（定理 1）。如果在两个相近数之间执行减法，那么操作数中的最高有效数位将因为相等而彼此抵消。有以下两种抵消：恶性抵消和良性抵消。

恶性抵消发生在操作数受舍入误差的制约时。例如，在二次方程式中出现了表达式 $b^2 - 4ac$。数量 b^2 和 $4ac$ 受舍入误差的制约，因为它们是浮点乘法的结果。假定将它们舍入为最接近的浮点数，因此它们的准确性使误差保持在 .5 ulp 之内。它们相减时，抵消可能会导致许多精确数字消失，而留下受舍入误差影响的数字。因此，差值的误差可能是许多个 ulp。以 $b = 3.34$，$a = 1.22$，$c = 2.28$ 为例。$b^2 - 4ac$ 的精确值是 .0292。但是 b^2 舍入为 11.2，$4ac$ 舍入为 11.1，因此最终结果是 .1，误差是 70 ulp（即使 11.2 - 11.1 的精确结果等于 .1）[6]。减法运算并未引入任何误差，而是显示出先前乘法运算中引入的误差。

良性抵消发生在精确已知数量之间执行减法运算时。如果 x 和 y 没有舍入误差，依据定理 2 可知，若使用保护数位执行减法运算，那么差值 x-y 的相对误差会非常小（小于 2ε）。

有时，表现出恶性抵消的公式可以通过重新整理的方式来消除此问题。我们还是以二次公式为例

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{4}$$

当 $b^2 \gg ac$，则 $b^2 - 4ac$ 不会产生抵消，而且

$$\sqrt{b^2 - 4ac} \approx |b| \text{。}$$

但是公式之一内的另一加法运算（减法运算）将具有恶性抵消。要避免此类情况发生，请将 r_1 的分子和分母乘以

$$-b - \sqrt{b^2 - 4ac}$$

对于 r_2 进行类似处理，得到

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, \quad r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{5}$$

如果 $b^2 \gg ac$ 且 $b > 0$，则使用公式 (4) 计算 r_1 将产生抵消。因此，使用公式 (5) 计算 r_1，使用 (4) 计算 r_2。另一方面，如果 $b < 0$，则使用 (4) 计算 r_1，使用 (5) 计算 r_2。

表达式 $x^2 - y^2$ 是表示出恶性抵消的另一公式。将它计算为 $(x - y)(x + y)$ 更为精确。[7] 与二次方程式不同，这一改进的形式仍然包含减法运算，但是它是数量的良性抵消（没有舍入误差），而不是恶性抵消。依据定理 2，$x - y$ 中的相对误差最大为 2ε。对于 $x + y$，也同样正确。将两个具有较小相对误差结果的数量相乘，会产生具有较小相对误差的乘积（请参见第 38 页的“舍入误差”章节）。

6. 700，而不是 70。因为 .1 - .0292 = .0708，以 ulp(0.0292) 表示的误差是 708 ulp。 - 编辑者

7. 虽然表达式 $(x-y)(x+y)$ 不产生恶性抵消，但是，如果符合以下两个条件之一，其精确程度要比 x^2 y^2 稍差：$X \gg Y$ 或 $X \ll Y$。在这种情况下，$(x - y)(x + y)$ 具有三个舍入误差，但是 $x^2 - y^2$ 只有两个舍入误差，因为计算 x^2 和 y^2 中的较小者时产生的舍入误差不影响最终的减法运算。

为避免将精确值与计算值相混淆，特采用以下表示法。$x \ominus y$ 表示计算的差值（即具有舍入误差），而 $x - y$ 表示 x 和 y 的精确差值。类似地，$\oplus$、$\otimes$ 和 $\oslash$ 分别表示计算的加法、乘法和除法。全大写函数表示函数的计算值，如 `LN(x)` 或 `SQRT(x)`。小写函数和传统数学符号表示其精确值，如 $\ln(x)$ 和 $\sqrt{x}$ 。

虽然 $(x \ominus y) \otimes (x \oplus y)$ 与 $x^2 - y^2$ 极为近似，但是浮点数 x 和 y 本身可能近似为某些真实数量：$\hat{x}$ 和 $\hat{y}$ 。例如，$\hat{x}$ 和 $\hat{y}$ 可能是精确已知的十进制数，且无法用二进制精确表示。在这种情况下，即使 $x \ominus y$ 与 $x - y$ 极为近似，与真实表达式 $\hat{x} - \hat{y}$ 相比，它也可能具有较大的相对误差，因此 $(x + y)(x - y)$ 相对 $x^2 - y^2$ 的优势就不那么明显了。因为计算 $(x + y)(x - y)$ 与计算 $x^2 - y^2$ 的工作量几乎相同，所以在这种情况下前者明显是首选的形式。然而在一般情况下，如果将恶性抵消替换为良性抵消会产生较大的开销，那么这种做法是不值得的，因为输入通常是（但不总是）一个近似值。但是，即使数据是不精确的，完全消除抵消（如在二次方程式中）也是值得做的。本文始终假定算法的浮点输入是精确的，而且计算结果尽可能精确。

表达式 $x^2 - y^2$ 在重新写为 $(x - y)(x + y)$ 时更为精确，因为恶性抵消被良性抵消所替换。接下来恶性抵消公式示例更为有趣，可以将公式进行改写，以便仅表示出良性抵消。

三角形面积可以直接用其各边 a、b 和 c 的长度表示，如下所示：

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad \text{其中 } s = (a+b+c)/2 \tag{6}$$

（假设三角形非常平坦，即 $a \approx b + c$。那么 $s \approx a$，公式 (6) 中的项 $(s - a)$ 减去两个邻近的数，其中一个可能有舍入误差。例如，如果 $a = 9.0$、$b = c = 4.53$、s 的正确值为 9.03，并且 A 为 2.342.... 即使计算出的值 s (9.05) 的误差只有 2 ulps，计算出的值 A 也为 3.04，误差为 70 ulps。

有一种方法可以改写公式 (6)，以便它返回精确结果，甚至对于平坦的三角形也是这样 [Kahan 1986]。即

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, \; a \geq b \geq c \tag{7}$$

如果 a、b 和 c 不满足 $a \geq b \geq c$，那么在应用 (7) 之前请将它们重命名。检查 (6) 和 (7) 右侧是否是代数恒等是一项非常简单的工作。使用上述 a、b 和 c 的值得出的计算面积是 2.35，其误差是 1 ulp，比第一个公式要精确得多。

对于此示例来说，虽然公式 (7) 比 (6) 要精确得多，但是了解 (7) 在通常情况下的执行情况是非常必要的。

D.3.4.1 定理 3

由于减法运算是使用保护数位执行的 *(e ≤ .005)*，且计算的平方根在 *1/2* ulp 内，所以使用 (7) 计算三角形面积时产生的舍入误差最大为 *11*ε。

条件 $e < .005$ 几乎在每个实际的浮点系统中都能得到满足。例如，当 $\beta = 2$ 且 $p \geq 8$ 时，可确保 $e < .005$；当 $\beta = 10$ 时，$p \geq 3$ 已足够。

在命题（如定理 3）中讨论表达式的相对误差时，认为表达式是使用浮点运算计算的。特别是，相对误差实际上是由以下表达式产生的：

$$\texttt{SQRT}((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4 \tag{8}$$

由于 (8) 的繁琐性，在定理的陈述中我们通常采用 E 的计算值，而不是写出带有圆形符号的 E。

误差界限通常是非常保守的。在上述数值示例中，(7) 的计算值是 2.35，与真实值 2.34216 相比，相对误差是 0.7ε，这比 11ε 要小得多。计算误差界限的主要原因不是获得精确界限，而是检验公式是否不包含数值问题。

请看最后一个示例：可以改写为使用良性抵消的表达式 $(1 + x)^n$，其中 $x \ll 1$ 。此表达式出现在金融计算中。以每天将 100 美元存入一个银行帐户为例，其年利率为 6%，每天按复利计算。如果 $n = 365$ 且 $i = .06$，那么在年底时累积的金额为

$$100\frac{(1 + i/n)^n - 1}{i/n}$$

美元。如果这是使用 $\beta = 2$ 和 $p = 24$ 进行计算的，那么结果将是 37615.45 美元，与精确结果 37614.05 美元相比，差额为 1.40 美元。导致此问题的原因很明显。表达式 $1 + i/n$ 将导致 1 和 .0001643836 相加，这样会丢失 i/n 的低位。将 $1 + i/n$ 扩大到 n 次幂时，将扩大此舍入误差。

这个令人头痛的表达式 $(1 + i/n)^n$ 可以改写为 $e^{n\ln(1 + i/n)}$，现在的问题是在 x 较小的情况下计算 $\ln(1 + x)$。一种方法是使用近似 $\ln(1 + x) \approx x$，在这种情况下应支付 37617.26 美元，差额为 3.21 美元，甚至还不如使用这个直观的公式精确。但是，有一种方法可以非常精确地计算 $\ln(1 + x)$，如定理 4 所示 [Hewlett-Packard 1982]。使用此公式的计算结果是 37614.07，其精确性使误差保持在两美分之内！

定理 4 假设 `LN(x)` 近似于 $\ln(x)$，其误差在 1/2 ulp 内。它解决的问题是：当 x 很小时，$\texttt{LN}(1 \oplus x)$ 不接近于 $\ln(1 + x)$，因为 $1 \oplus x$ 在 x 的低位中丢失了信息。也就是说，在以下情况下，$\ln(1 + x)$ 的计算值不接近于其实际值：$x \ll 1$ 。

D.3.4.2 定理 4

在使用公式

$$\ln(1 + x) = \begin{cases} x & \text{对于 } 1 \oplus x = 1 \\ \dfrac{x\ln(1 + x)}{(1 + x) - 1} & \text{对于 } 1 \oplus x \neq 1 \end{cases}$$

计算 ln(1 + x) 的情况下，**如果 $0 \le x < 3/4$，减法运算是使用保护数位 $(e < 0.1)$ 执行的，且 ln 的计算值误差在 $1/2$ ulp 之内，则相对误差最大为 5ε**。

此公式适用于 x 的任意值，但仅与 $x \ll 1$ 有关，这就是简易公式 ln(1 + x) 中出现恶性抵消的地方。虽然此公式看起来可能有些神秘，但是一个简单的解释就可以说明它的工作原理。将 ln(1 + x) 写为

$$x\left(\frac{\ln(1+x)}{x}\right) = x\mu(x).$$

可以精确计算左侧因子，但是在将 1 和 x 相加时右侧因子 $\mu(x) = \ln(1+x)/x$ 将产生较大的舍入误差。然而，μ 几乎是恒定的，因为 $\ln(1+x) \approx x$。因此，稍微改变 x 不会引入很大误差。换句话说，如果 $\tilde{x} \approx x$，计算 $x\mu(\tilde{x})$ 将与 $x\mu(x) = \ln(1+x)$ 极为近似。是否存在一个 $\tilde{x}$ 的值，使用它可以精确计算 $\tilde{x}$ 和 $\tilde{x}+1$？是的；该值是 $\tilde{x} = (1 \oplus x) \ominus 1$，因为使用该值后 $1 + \tilde{x}$ 与 $1 \oplus x$ 完全相等。

可以这样总结本节：保护数位在彼此接近且精确已知的数量相减 （良性抵消）时保证了精确性。有时，可以通过使用良性抵消来改写产生不精确结果的公式，以便获得相当高的数值精确性；但是，仅当使用保护数位进行减法运算时此过程才起作用。使用保护数位的开销并不高，因为它仅仅要求将加法器加宽一位。对于 54 位双精度加法器，增加的开销少于 2%。只需付出如此代价，您就能够运行许多算法 （如公式 (6)）来计算三角形的面积和表达式 ln(1 + x)。虽然多数现代计算机具有保护数位，但是也有一些计算机 （如 Cray 系统）没有保护数位。

D.3.5 精确舍入的运算

在使用保护数位进行浮点运算时，它们不如先精确计算再舍入为最接近的浮点数那样精确。以这种方式进行的运算将称为**精确舍入的运算**。[8] 紧邻定理 2 之前的示例显示出单个保护数位并不总是给出精确舍入的结果。上一节给出了需要有保护数位才能正常工作的几个算法示例。本节给出要求精确舍入的算法示例。

到目前为止，尚未给出任何舍入的定义。舍入是很好理解的，只是如何舍入中间数有些令人棘手；例如，应该将 12.5 舍入为 12 还是 13？一个学派将 10 个数字分成两半，使 {0, 1, 2, 3, 4} 向下舍入，使 {5, 6, 7, 8, 9} 向上舍入；因此 12.5 将舍入为 13。这是在 Digital Equipment Corporation VAX 计算机上的舍入方式。另一学派则认为：由于以 5 结尾的数介于两个可能的舍入数中间，向下舍入与向上舍入需视情况而定。实现这一发生概率为 50% 事件的方法之一是要求舍入结果的最低有效数字为偶数。因此 12.5 舍入为 12 而不是 13，因为 2 是一个偶数。其中的哪种方法是最佳的，是向上舍入还是舍入为偶数？Reiser 和 Knuth [1975] 倾向于舍入为偶数，并提供了以下理由。

8. 通常也称为*正确舍入的运算*。– 编辑者

D.3.5.1 定理 5

假设 x 和 y 是浮点数，并定义 $x_0 = x$, $x_1 = (x_0 \ominus y) \oplus y$, …, $x_n = (x_{n-1} \ominus y) \oplus y$。如果使用舍入为偶数精确舍入 $\oplus$ 和 $\ominus$，那么对于所有 n，$x_n = x$；对于所有 $n \geq 1$，$x_n = x_1$。■

为了阐明此结果，请以 $\beta = 10$，$p = 3$ 为例，并令 $x = 1.00$，$y = -.555$。当进行向上舍入时，该序列将变成

$$x_0 \ominus y = 1.56, \quad x_1 = 1.56 \ominus .555 = 1.01, \quad x_1 \ominus y = 1.01 \oplus .555 = 1.57,$$

x_n 的每个连续值都将增加 .01，直至 $x_n = 9.45$ $(n \leq 845)$[9] 为止。在使用舍入为偶数的情况下，x_n 始终是 1.00。此示例说明：在使用向上舍入规则时，计算可能会逐渐向上漂移，而依据此定理，这种情况在使用舍入为偶数时不会发生。本文的其余部分将使用舍入为偶数。

精确舍入的应用之一是执行多种精度运算。有两种基本方法可以获得较高的精度。一种方法使用极大的有效数字表示浮点数，将有效数字存储在字元数组中，并用汇编语言编写用于处理这些数的例程的代码。第二种方法将更高精度的浮点数表示为普通浮点数的数组，在其中增加无限精度的数组元素以便恢复高精度浮点数。此处将讨论第二种方法。使用浮点数的数组的优点是：可以使用高级语言编写其可移植代码，但是它要求精确舍入的运算。

在此系统中乘法的关键是将积 xy 表示为和，其中每个被加数都具有与 x 和 y 相同的精度。通过拆分 x 和 y，可以做到这一点。编写 $x = x_h + x_l$ 和 $y = y_h + y_l$，则精确乘积为

$$xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l \text{ 。}$$

如果 x 和 y 具有 p 位有效数字，那么被加数也将具有 p 位有效数字，但条件是 x_l、x_h、y_h、y_l 可以使用 $\lceil p/2 \rceil$ 位来表示。当 p 是偶数时，很容易找到一种拆分方法。数 $x_0.x_1 \ldots x_{p-1}$ 可以写为 $x_0.x_1 \ldots x_{p/2-1}$ 和 $0.0 \ldots 0x_{p/2} \ldots x_{p-1}$ 之和。当 p 是奇数时，这一简单的拆分方法将不起作用。但是，通过使用负数可以获得额外位。例如，如果 $\beta = 2$，$p = 5$ 且 $x = .10111$，那么 x 可以拆分为 $x_h = .11$ 和 $x_l = -.00001$。拆分数的方法不止一种。一种易于计算的拆分方法是由 Dekker [1971] 提出的，但它需要多个单个保护数位。

D.3.5.2 定理 6

假设 p 是浮点精度，要求在 $\beta > 2$ 时 p 是偶数，并假设浮点运算是精确舍入的。那么，如果 $k = \lceil p/2 \rceil$ 是精度的一半（向上舍入）且 $m = \beta^k + 1$，则可以将 x 拆分为 $x = x_h + x_l$，其中

$$x_h = (m \otimes x) \ominus (m \otimes x \ominus x), \quad x_l = x \ominus x_h,$$

每个 x_i 是可以使用 $\lceil p/2 \rceil$ 位精度表示的。

9. 当 $n = 845$ 时，$x_n = 9.45$，$x_n + 0.555 = 10.0$，$10.0 - 0.555 = 9.45$。因此，当 $n > 845$ 时，$x_n = x_{845}$。

为了在示例中说明此定理的应用，可令 β = 10，$p = 4$，$b = 3.476$，$a = 3.463$，$c = 3.479$。那么，$b^2 - ac$ 在舍入为最接近的浮点数时是 .03480，而 $b \otimes b = 12.08$，$a \otimes c = 12.05$，因此 $b^2 - ac$ 的计算值是 .03。这样，误差就是 480 ulp。使用定理 6 编写 $b = 3.5 - .024$，$a = 3.5 - .037$ 和 $c = 3.5 - .021$，b^2 将变成 $3.5^2 - 2 \times 3.5 \times .024 + .024^2$。每个被加数都是精确的，因此 $b^2 = 12.25 - .168 + .000576$，此时未计算其中的和。类似地，$ac = 3.5^2 - (3.5 \times .037 + 3.5 \times .021) + .037 \times .021 = 12.25 - .2030 + .000777$。最后，将这两个数列逐项相减将为 $0 \oplus .0350 \ominus .000201 = .03480$ 的 $b^2 - ac$ 得出估计值，它与精确舍入的结果完全相同。为了说明定理 6 确实需要精确舍入，请假设 $p = 3$，β = 2，$x = 7$。那么，$m = 5$，$mx = 35$，$m \otimes x = 32$。如果使用单个保护数位进行减法运算，那么 $(m \otimes x) \ominus x = 28$。因此，$x_h = 4$，$x_l = 3$，由此得出 x_l 不能用 $[p/2] = 1$ 位表示。

作为精确舍入的最后一个示例，假设用 10 除 m。结果是一个浮点数，一般情况下将不等于 $m/10$。当 β = 2 时，用 10 乘 $m/10$ 将复原 m，条件是使用了精确舍入。实际上，一个更为常规的事实（由 Kahan 提出）是正确的。其证明非常巧妙，但对此细节不感兴趣的读者可以直接跳至第 14 页的 “IEEE 标准”章节。

D.3.5.3 定理 7

当 β = 2 时，如果 m 和 n 是整数，且 $|m| < 2^{p-1}$，n 具有特殊形式 $n = 2^i + 2^j$，那么 $(m \oslash n) \otimes n = m$，条件是浮点运算是精确舍入的。

D.3.5.4 证明

换算为 2 的幂次方不会产生不良影响，因为这样仅改变指数，而不改变有效数字。如果 $q = m/n$，那么按比例缩放为 n 以便 $2^{p-1} \le n < 2^p$，并按比例缩放为 m 以便 $1/2 < q < 1$。因此，$2^{p-2} < m < 2^p$。因为 m 具有 p 个有效位，所以在二进制点右侧它最多有一位。改变 m 的符号不会产生不良影响，因此假设 $q > 0$。

如果 $\bar{q} = m \oslash n$，那么要证明此定理需要得到

$$|n\bar{q} - m| \le \frac{1}{4} \tag{9}$$

这是因为在二进制点右侧 m 最多具有 1 位，因此 $n\bar{q}$ 将舍入为 m。为了处理 $|n\bar{q} - m| = 1/4$ 时的中间情况，请注意，由于初始的未缩放的 m 具有 $|m| < 2^{p-1}$，其低位是 0，因此已缩放的 m 的低位也是 0。因此，中间情况将舍入为 m。

假定 $q = .q_1q_2 \ldots$，并假设 $\hat{q} = .q_1q_2 \ldots q_p1$。要估算 $|n\bar{q} - m|$，请首先计算

$$|\hat{q} - q| = |N/2^{p+1} - m/n|,$$

其中 N 是一个奇数整数。因为 $n = 2^i + 2^j$ 且 $2^{p-1} \le n < 2^p$，所以对于某些 $k \le p - 2$ 一定存在 $n = 2^{p-1} + 2^k$，因此

$$|\hat{q} - q| = \left| \frac{nN - 2^{p+1}m}{n2^{p+1}} \right| = \left| \frac{(2^{p-1-k} + 1)N - 2^{p+1-k}m}{n2^{p+1-k}} \right| \quad 。$$

分子是一个整数，并且因为 N 是一个奇数，所以实际上它是一个奇数整数。因此，

$|\hat{q} - q| \ge 1/(n2^{p+1-k})$。

假设 $q < \hat{q}$（与 $q > \hat{q}$ 的情况是类似的）。[10] 那么， n

$\bar{q} < m$，

$$|m - n\bar{q}| = m - n\bar{q} = n(q - \bar{q}) = n(q - (\hat{q} - 2^{-p-1})) \le n\left(2^{-p-1} - \frac{1}{n2^{p+1-k}}\right)$$

$$= (2^{p-1} + 2^k)2^{-p-1} - 2^{-p-1+k} = \frac{1}{4}$$

这将得到 (9) 并证明此定理。[11] ∎

只要将 $2^i + 2^j$ 替换为 $\beta^i + \beta^j$，此定理对于任何基数 β 都是成立的。但是，随着 β 的增大，$\beta^i + \beta^j$ 形式的分母差值将越来越大。

我们现在可以回答以下问题：如果基本算术运算引入比所需稍大的舍入误差，这有没有关系？答案是确实有关系，因为通过精确的基本运算我们可以证明公式在具有较小相对误差的情况下是“正确的”。在此意义上，第 6 页的“抵消”章节讨论了几种算法，它们需要保护数位以便产生正确结果。但是，如果这些公式的输入是用不精确的方式表示的数，那么定理 3 和定理 4 的限制就不需要太注意了。原因是：如果 x 和 y 仅仅是某些衡量数量的近似，那么良性抵消 $x - y$ 可能变为恶性抵消。但是，即使对于不精确的数据，精确运算也是有用的；因为通过精确运算我们可以建立精确关系（如定理 6 和定理 7 中所讨论的那些关系）。即使每个浮点变量只是某个实际值的近似，精确运算也是有用的。

10.请注意，在二进制 q 不能等于 $\hat{q}$ 。 – 编辑者

11.作为练习留给读者：将证明扩展到不是 2 的基数。 - 编辑者

D.4 IEEE 标准

浮点计算有两种不同的 IEEE 标准。IEEE 754 是一个二进制标准。对于单精度，它要求 β = 2、p = 24；对于双精度 [IEEE 1987]，它要求 p = 53。它还指定单精度和双精度中位的精确布局。 IEEE 854 允许 β = 2 或 β = 10。与 754 不同，它不指定如何将浮点数编码到位中 [Cody 等 1984]。它不需要 p 的特定值，而是为单精度和双精度指定对 p 的允许值的约束。在讨论这两个标准的公共属性时，将使用术语 IEEE **标准**。

本节简要介绍 IEEE 标准。每个小节讨论的都是标准的一个方面，以及将这方面制定为标准的原因。本文的目的不是论述 IEEE 标准是可能的最佳浮点标准，而是将其接受为指定标准，并介绍它的用途。有关详细信息，请查阅这些标准 [IEEE 1987； Cody 等 1984]。

D.4.1 格式与运算

D.4.1.1 基数

IEEE 854 允许 β = 10 的原因是显而易见的。人们就是使用基数 10 来交换和考虑数字的。β = 10 尤其适合于计算器，每个运算的结果都由计算器以十进制显示。

IEEE 854 要求：如果基数不是 10，就必须是 2。它这样要求有几个原因。第 4 页的“相对误差和 Ulp”节提到了其中的一个原因：当 β 是 2 时误差分析的结果更精确，因为进行相对误差计算时 .5 ulp 的舍入误差按 β 的因子发生变化，而且基于相对误差的误差分析几乎始终是比较简单的。其中的原因必然与大基数的有效精度有关。以比较 β = 16、p = 1 与 β = 2、 p = 4 为例。这两个系统的有效数字都是 4 位。判断 15/8 的计算结果。当 β = 2 时，15 被表示为 1.111×2^3，15/8 被表示为 1.111×2^0。因此 15/8 是精确的。但是，当 β = 16 时， 15 被表示为 $F \times 16^0$，其中 F 是表示 15 的十六进制数字。但是 15/8 被表示为 1×16^0，它只有一个正确位。通常，基数 16 最多可丢失 3 位，因此精度为 p 的十六进制数字具有的有效精度低至 $4p$ - 3，而不是 $4p$ 个二进制位。既然大的 β 值存在这些问题，为什么 IBM 为其 system/370 选择了 β = 16 呢？只有 IBM 才知道确切原因，但是有两个可能的原因。第一个是增大了指数范围。 system/370 上的单精度具有 β = 16、p = 6。因此有效数字需要 24 位。因为必须凑够 32 位，所以会为指数保留 7 位，为符号位保留 1 位。这样，可表示数值的大小范围是从 16^{-2^6} 到 $16^{2^6} = 2^{2^8}$ 。为了在 β = 2 时获得类似的指数范围，需要将 9 位用于指数部分，仅为有效数字部分保留 22 位。但是，上面刚刚指出：当 β = 16 时，有效精度可以低至 $4p$ - 3 = 21 位。更糟的是，当 β = 2 时，有可能获得额外的精度位（正如本节后面所说明的情况），因此 β = 2 的计算机具有 23 个精度位，而 β = 16 的计算机则具有 21 - 24 位的精度范围。

选择 β = 16 的另一种可能原因必然与移位有关。两个浮点数进行加法运算时，如果它们的指数不同，其中一个有效数字必须进行移位才能使小数点对齐，这将降低运算速度。在 β = 16，p = 1 的系统中，1 到 15 之间的所有数字都具有相同的指数，所以这组数字中任意两个不同的数进行加法运算，即有 $\binom{15}{2}$ = 105 可能对时，不需要进行移位。但是，在 β = 2、p = 4 的系统中，这些数的指数范围是从 0 到 3，105 对中有 70 对需要进行移位。

对于现今大多数硬件来讲，通过避免操作数子集进行移位而获得的性能可以忽略，因此 β = 2 的微小波动使其成为更可取的基数。使用 β = 2 的另一个优点是有办法获得额外的有效位。[12] 因为浮点数始终是规格化的，所以有效数字的最高有效位始终是 1，这样就没有必要浪费一个表示它的存储位。使用此技巧的格式具有**隐藏**位。在第 3 页的“浮点格式”中已经指出：这需要对 0 进行特殊约定。那里给出的方法是 $e_{\min} - 1$ 的指数，全零的有效数字不表示

$1.0 \times 2^{e_{min}-1}$，而是表示 0。

IEEE 754 单精度是用 32 位编码的，将 1 位用于符号， 8 位用于指数， 23 位用于有效数字。但是，它使用一个隐藏位，因此有效数字是 24 位 (p = 24)，即使它仅使用 23 位进行编码也是如此。

D.4.1.2 精度

IEEE 标准定义了四种不同的精度：单精度、双精度、单精度扩展和双精度扩展。在 IEEE 754 中，单精度和双精度大致对应于大多数浮点硬件所提供的精度。单精度占用一个 32 位字，双精度占用两个连续的 32 位字。扩展精度是一种至少提供一点额外精度和指数范围的格式 (表 D-1)。

表 D-1 IEEE 754 格式参数

	格式			
参数	**单精度**	**单精度扩展**	**双精度**	**双精度扩展**
p	24	≥ 32	53	≥ 64
$e_{\max}$	+127	≥ 1023	+1023	> 16383
$e_{\min}$	-126	≤ -1022	-1022	≤ -16382
指数宽度 （位）	8	≤ 11	11	≥ 15
格式宽度 （位）	32	≥ 43	64	≥ 79

IEEE 标准仅指定扩展精度提供多少个额外位的下界。允许的最小双精度扩展格式有时称为 80 **位格式**，尽管上表显示它可以使用 79 位。原因是，扩展精度的硬件实现通常不使用隐藏位，因此将使用 80 位而不是 79 位。[13]

12.这似乎是首先由 Goldberg [1967] 发表的，尽管 Knuth （[1981]，第 211 页）将此方法归功于 Konrad Zuse。

13.Kahan 认为，扩展精度具有 64 位的有效数字，因为这是可以在 Intel 8087 上不增加循环时间的情况下进行进位传送的最宽精度 [Kahan 1988]。

该标准将扩展精度作为重中之重，没有做出有关双精度的任何建议，但是强烈建议**实现应该支持对应于所支持的最宽基本格式的扩展格式，** …

使用扩展精度的一个动因来自计算器。通常计算器将显示 10 位，但在内部使用 13 位。通过仅显示 13 位中的 10 位，计算器在用户看来就像一个按 10 位精度计算指数、余弦等的“黑箱”。计算器要在 10 位精度内计算诸如 exp、log 和 cos 之类的函数，且达到合理的效率，它需要使用几个额外位。不难找到这样一个简单的有理数表达式，它求对数近似值的误差为 500 ulp。这样，使用 13 位进行计算就可以得出正确的 10 位结果。通过将额外的 3 位隐藏，计算器为操作者提供了一个简单的模型。

IEEE 标准中的扩展精度起着类似的作用。它使库可以高效地计算数量，且单（或双）精度的误差在约 .5 ulp 内，为使用那些库的用户提供了一个简单模型，即：每个基本操作（不管是简单的乘法运算还是对对数的调用）都会返回精确度大约在 .5 ulp 内的值。但是，当使用扩展精度时，确保它的使用对用户透明是很重要的。例如，在计算器上，如果显示值的内部表示没有舍入到与显示值相同的精度，则进一步运算的结果将取决于隐藏位数，并且对用户来说似乎是不可预知的。

以 IEEE 754 单精度与十进制之间的转换问题为例来进一步说明扩展精度。理想情况下，将用足够位数显示单精度数，以便在将十进制数读回时可以再现单精度数。结果是 9 个十进制位就足以再现单精度二进制数（请参见第 46 页的“二进制到十进制的转换”节）。将十进制数再转换回它的唯一二进制表示时，小至 1 ulp 的舍入误差就是致命的，因为这将给出错误的结果。这是一种扩展精度对于高效算法至关重要的情况。在单精度扩展可用时，存在一种非常简单的方法，它可以将十进制数转换为单精度二进制数。首先将 9 个十进制数字作为整数 N 读入，忽略小数点。在表 D-1, $p \geq 32$，因为 $10^9 < 2^{32} \approx 4.3 \times 10^9$, N 可以用单精度扩展精确表示。接下来，找出提升 N 所需的适当幂 10^P。这将是十进制数的指数以及（到目前为止）被忽略小数点的位置的组合。计算 $10^{|P|}$。如果 $|P| \leq 13$，则这也是精确表示的，因为 $10^{13} = 2^{13}5^{13}$ 且 $5^{13} < 2^{32}$。最后，将 N 和 $10^{|P|}$ 相乘（如果 $p < 0$，则将它们相除）。如果最后的这个运算是精确进行的，则将再现最接近的二进制数。第 46 页的“二进制到十进制的转换”节说明如何精确地进行最后的相乘（或相除）。因此，对于 $|P| \leq 13$，使用单精度扩展格式，可以将 9 位十进制数转换为最接近的二进制数（即精确舍入的数）。如果 $|P| > 13$，则使用单精度扩展不足以使上述算法始终计算出精确舍入的等价的二进制数，但是 Coonen [1984] 证明这足以保证先将二进制数转换为十进制数而后再转换回来将再现原始的二进制数。

如果支持双精度，则上述算法将以双精度而不是单精度扩展运行，但是将双精度转换为 17 位十进制数再转换回来将需要双精度扩展格式。

D.4.1.3 指数

因为指数可以是正数或负数，所以必须选择某个方法来表示其符号。表示有符号数的两种常用方法是符号 / 大小和 2 的补码。符号 / 大小是 IEEE 格式中用于有效数字符号的系统：一个位用于容纳符号，其余位用来表示数的大小。2 的补码表示通常在整数运算中使用。在此方案中，范围 $[-2^{p-1}, 2^{p-1} - 1]$ 内的数由等于它以 2^p 为模的结果的最小非负数表示。

IEEE 二进制标准不使用其中的任一方法来表示指数，而是改用**偏置**的方法表示。在使用单精度数的情况下，指数以 8 位存储，偏差是 127（对于双精度，它是 1023）。这意味着，如果 $\bar{k}$ 是被解释为无符号整数的指数位的值，则浮点数的指数是 $\bar{k}$ – 127。这通常称为**无偏指数**，以区别于偏离指数 $\bar{k}$.

从表 D-1 可以知道，单精度有 e_{max} = 127，e_{min} = -126。其 $|e_{min}| < e_{max}$ 的原因是最小数 $(1/2^{e_{min}})$ 的倒数将不上溢。虽然最大数的倒数确实会发生下溢，但是下溢通常不如上溢严重。第 14 页的 “基数”节解释了 e_{min} - 1 用于表示 0，第 18 页的 “特殊数量”将介绍 e_{max} + 1 的一个用途。在 IEEE 单精度中，这意味着偏离指数介于 e_{min} – 1 = -127 和 e_{max} + 1 = 128 之间，而无偏指数介于 0 和 255 之间 （这正好是可以使用 8 位表示的非负整数）。

D.4.1.4 运算

IEEE 标准要求精确舍入加、减、乘、除的结果。也就是说，必须先精确计算结果，然后舍入为最接近的浮点数 （通过舍入为偶数）。第 5 页的 “保护数位”节指出：当两个浮点数的指数有很大差异时，计算这两个浮点数的精确差或和的开销会非常大。该节介绍了保护数位，它提供了一种在保证相对误差很小的同时计算差值的实用方法。但是，使用单个保护数位进行计算不会总是给出与计算精确结果再进行舍入相同的结果。通过引入第二个保护数位和第三个**粘滞**位，就能够以比单个保护数位稍高的成本计算差值，但是结果是相同的，就好像是先精确计算差值再进行舍入 [Goldberg 1990]。这样，就可以高效实施该标准。

完全指定算术运算结果的一个原因是为了改进软件的可移植性。当一个程序在两台计算机之间移动且这两台计算机都支持 IEEE 运算时，如果任意的中间结果出现不同，则一定是由于软件错误，而不是运算差异。精确指定的另一优点是它使有关浮点的推理更容易进行。有关浮点的证明已足够困难，无须处理多种运算所产生的多种情况。正如整数程序的正确性是可以证明的，浮点程序的正确性也是可以证明的，不过在这种情况下要证明的是结果的舍入误差符合某些界限。定理 4 就是这样的一个证明示例。当精确指定被推理的运算时，就可以使这些证明容易得多。一旦某种算法经证明对于 IEEE 运算正确，那么它将在支持 IEEE 标准的任何计算机上正确工作。

Brown [1981] 曾建议了有关浮点的公理，这些公理包括大多数的现有浮点硬件。但是，在此系统中的证明无法检验第 6 页的 “抵消”和第 10 页的 “精确舍入的运算”节中的算法，它们需要的功能并非存在于所有硬件上。此外，与简单地定义精确执行再进行舍入的运算相比，Brown 的公理更为复杂。因此，从 Brown 的公理证明定理通常比先假定运算是精确舍入的再证明它们要困难。

在浮点标准应该涉及哪些运算这一问题上，没有完全一致的意见。除了基本运算 +、-、× 和 / 之外， IEEE 标准还指定正确舍入平方根、余数以及整数和浮点数之间的转换。它还要求正确舍入内部格式与十进制之间的转换 （非常大的数除外） 。 Kulisch 和 Miranker [1986] 曾建议将内积增加到精确指定的运算列表。他们注意到，在 IEEE 运算中计算内积时最终结果与正确结果之差可能相当大。例如，和是内积的一种特殊情况，和 $((2 \times 10^{-30} + 10^{30}) - 10^{30}) - 10^{-30}$ 正好等于 10^{-30}，但是在使用 IEEE 运算的计算机上计算结果将是 -10^{-30}。使用比实现快速乘数所用更少的硬件计算误差在 1 ulp 内的内积，这是可能的 [Kirchner 和 Kulish 1987]。[14] [15]

在标准中提到的所有运算都需要精确舍入，但十进制和二进制之间的转换除外。原因是精确舍入所有运算的高效算法是已知的（转换除外）。对于转换，已知的最好的有效算法所产生的结果比精确舍入的结果稍差 [Coonen 1984]。

IEEE 标准因**制表人的难题**而不要求精确舍入超越函数。为了说明这一点，假定您正在制作 4 位指数函数表。那么，exp(1.626) = 5.0835。应该将它舍入为 5.083 还是 5.084？如果更仔细地计算，则结果变成 5.08350。然后是 5.083500。然后是 5.0835000。因为 exp 是超越函数，所以在辨别 exp(1.626) 是 5.083500…0*ddd* 还是 5.0834999…9*ddd* 之前该过程会继续任意长的时间。因此，指定超越函数的精度与按无限精度计算它再进行舍入所得的精度相同是不实用的。另一种方法将是从算法上指定超越函数。但是，在所有硬件架构上都工作良好的单个算法似乎是不存在的。有理逼近、CORDIC、[16] 和大表是用于当代计算机上计算超越函数的三种不同方法。每种方法适用于不同种类的硬件，目前还没有一种算法可以被当前广泛的硬件所接受。

D.4.2 特殊数量

在一些浮点硬件上，每种位模式都表示一个有效的浮点数。IBM System/370 就是这样的一个示例。另一方面，VAX™ 保留一些位模式以表示称为**保留操作数**的特殊数。这一思想可追溯到 CDC 6600，它具有表示特殊数量 `INDEFINITE` 和 `INFINITY` 的位模式。

IEEE 标准继承了这一传统，它具有 NaN（**不是数**）和无穷大。如果不使用任何特殊数量，则不存在处理异常情况（例如接受负数的平方根，而不是终止计算）的好方法。在 IBM System/370 FORTRAN 中，响应计算负数（如 -4）的平方根的默认操作导致错误消息的打印。因为每种位模式都表示一个有效数，所以平方根的返回值一定是某个浮点数。对于 System/370 FORTRAN，返回的是 $\sqrt{-4} = 2$。在 IEEE 运算中，此情况下返回 NaN。

IEEE 标准指定了以下特殊值（请参见表 D-2）：± 0、反向规格化的数、±∞ 和 NaN（如下一节中所述，存在多个 NaN）。这些特殊值都是使用 $e_{max} + 1$ 或 $e_{min} - 1$ 的指数进行编码的（已经指出 0 的指数是 $e_{min} - 1$）。

表 D-2 IEEE 754 特殊值

指数	尾数部分	表示
$e = e_{min} - 1$	$f = 0$	±0
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$

14.一些反对将内积作为一种基本运算包括在内的论点由 Kahan 和 LeBlanc [1985] 提出。

15.Kirchner 写到：在每时钟周期的一个部分乘积中，在硬件中计算误差在 1 ulp 内的内积是可能的。另外需要的硬件可以与为获得该速度所需的乘数数组进行比较。

16.CORDIC 是 Coordinate Rotation Digital Computer（坐标旋转数字计算机）的缩写，它是计算超越函数的一种方法，主要使用移位和加法（即非常少的乘法和除法）[Walther 1971]。在此方法中，另外需要的硬件可以与获得在 Intel 8087 和 Motorola 68881 上使用的速度所需要的乘数数组进行比较。

表 D-2 IEEE 754 特殊值 (续)

指数	尾数部分	表示
$e_{min} \le e \le e_{max}$	—	$1.f \times 2^e$
$e = e_{max} + 1$	$f = 0$	±∞
$e = e_{max} + 1$	$f \neq 0$	NaN

D.4.3 NaN

按照传统，将 0/0 或 $\sqrt{-1}$ 视为导致计算终止的不可恢复错误。但是，存在一些示例，表明在这样的情况下继续进行计算是有意义的。以一个子例程为例，该子例程用于查找函数 f 所包含的零，假设此子例程为 `zero(f)`。按照传统，零查找器要求用户输入一个区间 $[a, b]$，函数是在其上定义的，零查找器将按其进行搜索。也就是说，子例程是以 `zero(f, a, b)` 的形式调用的。更有效的零查找器不要求用户输入此额外信息。这种较普遍的零查找器尤其适合于计算器，通常只是键入一个函数，然而不方便的是必须指定定义域。但是，可以很容易地了解到大多数零查找器需要定义域的原因。零查找器是通过在各个值上探查函数 `f` 来执行工作的。如果探查到 `f` 的定义域之外的值，则 `f` 的代码可能也计算 0/0 或 $\sqrt{-1}$，计算将停止，但未必终止零查找过程。

通过引入称为 NaN 的特殊值，并指定诸如 0/0 和 $\sqrt{-1}$ 之类的表达式计算来生成 NaN 而不是停止计算，就可以避免此问题。表 D-3 中列出了一些可以导致 NaN 的情况。这样，当 `zero(f)` 在 `f` 的定义域之外探查时，`f` 的代码将返回 NaN，并且零查找器可以继续执行。也就是说，`zero(f)` 未因作出错误推测而受到“惩罚”。记住此示例，就可以很容易地了解到将 NaN 与普通浮点数结合在一起的结果应该是什么。假定 `f` 的最后一条语句是 `return(-b + sqrt(d))/(2*a)`。如果 $d < 0$，则 `f` 应该返回 NaN。因为 $d < 0$，所以 `sqrt(d)` 是一个 NaN，而且 `-b + sqrt(d)` 将是一个 NaN（如果 NaN 和任何其他数的和是 NaN）。类似地，如果除法运算的一个操作数是 NaN，则商应该是 NaN。通常，每当 NaN 参与浮点运算时，结果都是另一 NaN。

表 D-3 产生 NaN 的运算

操作	产生 NaN 的表达式
+	$\infty + (-\infty)$
×	$0 \times \infty$
/	$0/0, \infty/\infty$
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$（当 $x < 0$ 时）

编写不要求用户输入定义域的零解算程序的另一方法是使用信号。零查找器可以为浮点异常安装一个信号处理程序。这样，如果 `f` 在其定义域之外进行计算且引发了一个异常，则将控制权返回给零解算程序。此方法的问题是每种语言都具有不同的信号处理方法（如果它有一个方法的话），因而不具备可移植性。

在 IEEE 754 中，通常将 NaN 表示为带有指数 $e_{max} + 1$ 和非零有效数字的浮点数。实现可以自由地将系统相关的信息置入有效位。这样，就不存在唯一的 NaN，而是整个系列的 NaN。将 NaN 和普通浮点数结合在一起时，结果应该与 NaN 操作数相同。因此，如果长计算的结果是 NaN，则有效数字中的系统相关信息将是生成计算中的第一个 NaN 时产生的信息。实际上，对于最后一条语句，有一个防止误解的说明。如果两个操作数都是 NaN，则结果将是其中的一个 NaN，但它可能不是首先生成的 NaN。

D.4.3.1 无穷

正如 NaN 提供了一种在遇到诸如 0/0 或 $\sqrt{-1}$ 之类的表达式时继续进行计算的方法一样，无穷大提供了一种在发生上溢时继续执行的方法。这比仅返回可表示的最大数要安全得多。以计算 $\sqrt{x^2 + y^2}$ 为例，此时 $\beta = 10$，$p = 3$ 且 $e_{max} = 98$。如果 $x = 3 \times 10^{70}$ 并且 $y = 4 \times 10^{70}$，则 x^2 将发生上溢，并被替换为 9.99×10^{98}。类似地，y^2 和 $x^2 + y^2$ 都将依次上溢，并由 9.99×10^{98} 替换。因此，最终结果将是 $\sqrt{9.99 \times 10^{98}} = 3.16 \times 10^{49}$，该结果绝对是错误的：正确结果应该是 5×10^{70}。在 IEEE 运算中，x^2 的结果是 ∞，y^2、$x^2 + y^2$ 和 $\sqrt{x^2 + y^2}$ 也是这样。因此最终结果是 ∞，这比返回根本不接近于正确结果的普通浮点数要安全。[17]

用 0 除 0 会产生一个 NaN。但是，用一个非零数去除以零则会返回无穷大：1/0 = ∞，-1/0 = -∞。这一区别的原因是：如果在 x 接近某个极限时 $f(x) \to 0$ 且 $g(x) \to 0$，则 $f(x)/g(x)$ 可以具有任意值。例如，如果 $f(x) = \sin x$ 且 $g(x) = x$ 时，则当 $x \to 0$ 时 $f(x)/g(x) \to 1$。但是，如果 $f(x) = 1 - \cos x$，则 $f(x)/g(x) \to 0$。将 0/0 视为两个非常小的数的商的极限情况时，0/0 可以表示任意数。因此，在 IEEE 标准中，0/0 产生一个 NaN。但是，如果 $c > 0$，$f(x) \to c$，且 $g(x) \to 0$，则对于任意分析函数 f 和 g，$f(x)/g(x) \to \pm\infty$。如果对于小的 x，存在 $g(x) < 0$，则 $f(x)/g(x) \to -\infty$，否则极限是 +∞。因此，IEEE 标准定义 $c/0 = \pm\infty$（只要 $c \neq 0$）。通常情况下，∞ 的符号取决于 c 的符号和 0 的符号，这样 -10/0 = -∞，而 -10/-0 = +∞。您可以区分因上溢而得到 ∞ 和因除以零而得到的 ∞，方法是检查状态标志（将在第 26 页的“标志”节中详细讨论）。在第一种情况下将设置上溢标志，而在第二种情况下设置除以零标志。

对于将无穷大作为操作数的运算，确定其结果的规则比较简单：将无穷大替换为有限数 x 并将极限视为 $x \to \infty$。因此，$3/\infty = 0$，因为

$$\lim_{x \to \infty} 3/x = 0 。$$

类似地，$4 - \infty = -\infty$，$\sqrt{\infty} = \infty$。当极限不存在时，结果是 NaN，因此 ∞/∞ 将是一个 NaN（表 D-3 包含其他示例）。这与用于得出 0/0 应为 NaN 的结论的推理是一致的。

17.细微的要点：虽然默认情况下 IEEE 运算会将发生上溢的数据舍入为 ∞，但是更改这一默认设置是可能的（请参见第 26 页的“舍入模式”）

当子表达式计算为 NaN 时，整个表达式的值也是 NaN。但是，对于 ±∞ 的情况，表达式的值可能是一个普通的浮点数，因为存在类似 $1/\infty = 0$ 的规则。下面是利用无穷大运算规则的实际示例。以计算函数 $x/(x^2+1)$ 为例。这个公式并不好，因为不仅它在 x 大于 $\sqrt{\beta}\beta^{e_{max}/2}$，但是无穷大运算规则将给出错误的答案，因为生成的结果为 0，而不是一个接近 $1/x$ 的数。但是，$x/(x^2+1)$ 可被重写为 $1/(x+x^{-1})$。这一改进的表达式将不会过早上溢，并且由于运用无穷大运算规则，当 $x = 0$: $1/(0+0^{-1}) = 1/(0+\infty) = 1/\infty = 0$ 时将生成正确的值。如果不运用无穷大运算规则，表达式 $1/(x+x^{-1})$ 需要对 $x = 0$ 进行测试，这不仅会增加额外的指令，而且也会中断流水线作业。此示例证明一种常规情况，即无穷大运算规则通常无需进行特殊情况检查；但是，需要对公式进行仔细检查以确保它们在无穷大时不会执行伪行为 （如 $x/(x^2+1)$ 时那样）。

D.4.3.2 有符号零

零是由指数 $e_{min} - 1$ 和零有效位数表示的。由于符号位可以具有两个不同的值，所以有两个零：+0 和 -0。如果在比较 +0 和 -0 时进行了区分，则类似 `if (x = 0)` 的简单测试将具有非常不可预知的行为，具体取决于 x 的符号。因此，IEEE 标准定义比较，以便 +0 = -0，而不是 -0 < +0。虽然忽略零的符号始终是可能的，但是 IEEE 标准没有这样做。当乘法或除法涉及有符号的零时，在计算结果符号时就可以应用通常的符号规则。因此，$3\cdot(+0) = +0$，$+0/-3 = -0$。如果零没有符号，则在 $x = \pm\infty$ 时关系 $1/(1/x) = x$ 将无法成立。原因是 $1/-\infty$ 和 $1/+\infty$ 的结果都是 0，而 $1/0$ 的结果是 $+\infty$，符号信息已经丢失。恢复恒等式 $1/(1/x) = x$ 的一种方法是仅具有一种无穷大，但是那将导致丢失上溢数量的符号的灾难性结果。

使用有符号零的另一示例涉及下溢和在 0 处具有不连续性的函数 （如 log）。在 IEEE 运算中，当 $x < 0$ 时将 $\log 0 = -\infty$ 和 $\log x$ 定义为 NaN 是自然的。假设 x 表示一个已经下溢到零的小负数。由于使用了有符号零，因此 x 将是负的，这样 log 可以返回 NaN。但是，如果不存在有符号零，则 log 函数无法区分下溢的负数与 0，因此将不得不返回 $-\infty$。在 0 处具有不连续性的函数的另一示例是 signum 函数，它返回数的符号。

有符号零的最值得关注的用途很可能出现在复杂运算中。举一个简单的例子，假设等式 $\sqrt{1/z} = 1/(\sqrt{z})$ 。当 $z \geq 0$ 时，这肯定是正确的。如果 $z = -1$，很明显计算结果给出 $\sqrt{1/(-1)} = \sqrt{-1} = i$ 和$1/(\sqrt{-1}) = 1/i = -i$ 。因此会出现：$\sqrt{1/z} \neq 1/(\sqrt{z})$ ！可以将此问题归因于以下事实：平方根是多值的，没有方法来选择值，因此它在整个复平面中是连续的。但是，如果不考虑包括所有负实数的**分支切割**，则平方根是连续的。这样，就留下了如何处理负实数的问题，负实数的形式为 $-x + i0$，其中 $x > 0$。有符号零提供了解决此问题的完美方法。形式为 $x + i(+0)$ 的数具有一个符号 ($i\sqrt{x}$)，分支切割另一侧上形式为 $x + i(-0)$ 的数字具有另一符号 ($-i\sqrt{x}$) 。事实上，计算 $\sqrt{\ }$ 的自然公式将给出这些结果。

返回到$\sqrt{1/z} = 1/(\sqrt{z})$ 。如果 $z = 1 = -1 + i0$，那么

$$1/z = 1/(-1 + i0) = [(-1 - i0)]/[(-1 + i0)(-1 - i0)] = (-1 - i0)/((-1)^2 - 0^2) = -1 + i(-0),$$

并且所以 $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$，而 $1/(\sqrt{z}) = 1/i = -i$。这样，IEEE 运算为所有 z 保持了此恒等式。一些更复杂的示例由 Kahan [1987] 给出。虽然区分 +0 和 -0 具有优点，但是有时候会令人迷惑。例如，有符号零破坏了关系 $x = y \Leftrightarrow 1/x = 1/y$，在 $x = +0$ 且 $y = -0$ 时它是错误的。但是， IEEE 委员会断定利用零的符号的优点大于其缺点。

D.4.3.3 反向规格化的数

以 $\beta = 10$、 $p = 3$ 且 $e_{min} = -98$ 的正规化浮点数为例。数 $x = 6.87 \times 10^{-97}$ 和 $y = 6.81 \times 10^{-97}$ 看起来完全是普通的浮点数，它们是最小浮点数 1.00×10^{-98} 的 10 倍多。但是，它们具有一个奇怪的属性：$x \ominus y = 0$ （即使 $x \neq y$! 原因是：$x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$ 太小以致于无法表示为规格化数，因此它必须清零。保留

$x = y \Leftrightarrow x - y = 0$ 属性有怎样的重要性呢？ (10)

很容易设想到编写以下代码段：`if (x ≠y) then z = 1/(x-y)`，程序会由于伪除以零而在运行了较长的时间后失败。跟踪与此类似的错误既费力又耗时。在更有哲理的级别上，计算机科学教科书通常指出：即使当前证明大程序正确是不切实际的，但利用证明程序的思想来设计程序，这通常会产生更好的代码。例如，即使不变量将不被用作证明的一部分，引入它也是相当有用的。浮点代码就像任何其他代码一样：它有助于具有所依赖的可证明事实。例如，当分析公式 (6) 时，了解 $x/2 < y < 2x \Rightarrow x \ominus y = x - y$ 是非常有帮助的。类似地，如果知道 (10) 是正确的，就可以更轻松地编写可靠的浮点代码。如果它仅对于大部分的数字是正确的，那么不能使用它来证明一切。

IEEE 标准使用反向规格化的[18]数，这保证了 (10) 以及其他有用的关系。它们是该标准中最有争议的部分，这可能就是 754 经过很长时间的延迟才得以批准的原因。声称符合 IEEE 标准的大多数高性能硬件并不直接支持反向规格化的数，而是在使用或产生非正规数时支持陷阱，并将其留给软件以模拟 IEEE 标准。[19] 支持反向规格化的数的思想可追溯到 Goldberg [1967]，这种思想非常简单。当指数是 e_{min} 时，有效数字不必进行规格化，这样在 $\beta = 10$、$p = 3$ 且 $e_{min} = -98$ 时 1.00×10^{-98} 不再是最小的浮点数，因为 0.98×10^{-98} 也是一个浮点数。

当 $\beta = 2$ 且使用隐藏位时，存在一个小的意外困难，因为指数为 e_{min} 的数将始终具有大于或等于 1.0 的有效位数（由于存在隐式前导位）。解决方法与表示 0 所使用的方法类似，并汇总在表 D-2 中。指数 e_{min} 用于表示反向规格的数。更正式的表述是：如果有效数字域中的位是 b_1， b_2， …， b_{p-1}，且指数的值是 e，那么当 $e > e_{min} - 1$ 时，所表示的数是 $1.b_1b_2\ldots b_{p-1} \times 2^e$，而在 $e = e_{min} - 1$ 时，所表示的数是 $0.b_1b_2\ldots b_{p-1} \times 2^{e+1}$。指数中的 +1 是必需的，因为反向规格数的指数是 e_{min}，而不是 $e_{min} - 1$。

请回想一下在本节开头给出的示例：$\beta = 10$， $p = 3$， $e_{min} = -98$， $x = 6.87 \times 10^{-97}$ 且 $y = 6.81 \times 10^{-97}$。对于反向规格的数，$x - y$ 不清零，而改为由反向规格化的数 $.6 \times 10^{-98}$ 来表示。这种行为称为渐进**下溢**。证实使用渐进下溢时 (10) 始终成立是很容易的。

18.在 854 中，它们被称为 *subnormal* “次正规的数”；在 754 中，称为 *denormal* “反向正规的数”。

19.这是导致标准的最棘手方面之一的原因。在使用软件陷阱的硬件上，频繁下溢的程序通常运行得非常慢。

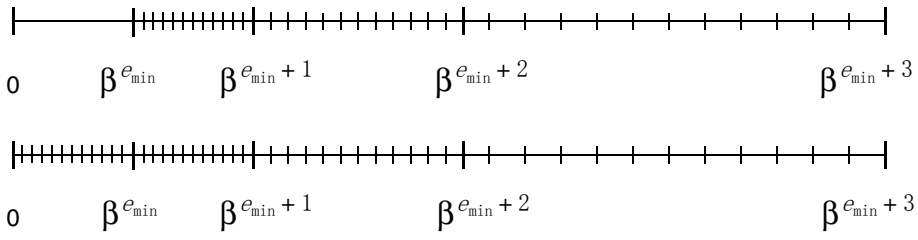


图 D-2 清零与渐进下溢的比较

图 D-2 说明反向规格化的数。图中的上实数直线显示规格化的浮点数。请注意 0 和最小规格化数 $1.0 \times \beta^{e_{min}}$ 之间的间隔。如果浮点计算的结果位于此间隔内，则将其清零。下实数直线显示将反向规格数增加到一组浮点数时的情况。“间隔”被填充，当计算的结果小于 $1.0 \times \beta^{e_{min}}$ 时，将用最接近的反向规格数来表示。将反向规格化的数增加到实数直线时，邻近浮点数之间的间距的变化是有规律的：邻近间距要么长度相同，要么按 β 的因子变化。如果不使用反向规格数，则间距突然从 $\beta^{-p+1}\beta^{e_{\min}}$ 变化到 $\beta^{e_{\min}}$（是 β^{p-1} 的因子），而不是按 β 的因子有规律的变化。由于这一原因，当使用渐进下溢时，对于接近下溢阈值的规格化数会具有较大相对误差的许多算法来说，在此范围内表现良好。

如果不使用渐进下溢，则对于规格化的输入来说，简单表达式 $x - y$ 会具有非常大的相对误差，正如上面所看到的 $x = 6.87 \times 10^{-97}$ 和 $y = 6.81 \times 10^{-97}$ 的相对误差。即使不存在消去，大的相对误差也会发生，如下例所示 [Demmel 1984]。以两个复数 $a + ib$ 和 $c + id$ 相除为例。很明显，公式

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2} \cdot i$$

受到以下问题的影响：如果分母 $c + id$ 的任一部分大于 $\sqrt{\beta}\beta^{e_{max}/2}$，则公式将发生上溢，即使最终结果可能完全处于范围之内也是如此。计算商的一种较好的方法是使用 Smith 的公式：

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{a+b(d/c)}{c+d(d/c)} + i\dfrac{b-a(d/c)}{c+d(d/c)} & \text{如果}(|d| < |c|) \\ \dfrac{b+a(c/d)}{d+c(c/d)} + i\dfrac{-a+b(c/d)}{d+c(c/d)} & \text{如果}(|d| \geq |c|) \end{cases} \tag{11}$$

将 Smith 的公式应用于 $(2 \cdot 10^{-98} + i10^{-98})/(4 \cdot 10^{-98} + i(2 \cdot 10^{-98}))$ 将得出具有渐进下溢的正确结果 0.5。在清零时它产生 0.4，此时误差为 100 ulp。反向规格化的数用于保证参数的误差边界一直下降到 $1.0 \times \beta^{e_{\min}}$。

D.4.4 异常、标志和陷阱处理程序

在 IEEE 运算中出现类似除以零或上溢等异常情况时，默认设置是传送结果并继续执行。典型的默认结果如下：对于 0/0 和 $\sqrt{-1}$，结果是 NaN；对于 1/0 和上溢，结果是 ∞。前面的几节给出了从具有这些默认值的异常情况中继续执行属于合理操作的示例。当出现任何异常时，还会设置状态标志。要为用户提供一种读取和写入状态标志的方法，需要实施 IEEE 标准。这些标志具有“粘滞性”，因为一旦设置，那么在被显式清除以前它们都将一直保持这种设置。测试标志是区分 1/0（它是来自上溢的真正无穷大）的唯一方法。

有时，在出现异常情况时继续执行是不正确的。第 20 页的“无穷”节给出了示例 $x/(x^2 + 1)$。当 $x > \sqrt{\beta}\beta^{e_{max}/2}$ 时，分母是无穷大，产生最终结果 0，这完全是错误的。虽然此公式可以通过改写为 $1/(x + x^{-1})$ 来解决这个问题，但是改写并不总是能够解决问题。IEEE 标准强烈建议实施要允许安装陷阱处理程序。这样，当出现异常时，将调用陷阱处理程序，而不是设置标志。陷阱处理程序返回的值将用作运算的结果。清除或设置状态标志是陷阱处理程序的责任；否则，允许标志的值是未定义的。

IEEE 标准将异常分成 5 类：上溢、下溢、除以零、无效运算和不精确。每类异常都有单独的状态标志。前三类异常的含义是不言而喻的。无效运算包括表 D-3 中列出的情况，以及产生 NaN 的任何比较。导致无效异常的运算的默认结果是返回一个 NaN，但反过来是不正确的。当某个运算的其中一个操作数是 NaN 时，结果是 NaN，但不引发无效异常，除非该运算还符合表 D-3 中列出的条件之一。[20]

表 D-4 IEEE 754* 中的异常

异常	禁止陷阱时的结果	陷阱处理程序的参数
上溢	$\pm\infty$ 或 $\pm x_{max}$	round($x2^{-\alpha}$)
下溢	0, $\pm 2^{e_{min}}$ 或反向规格数	round($x2^{\alpha}$)
除以零	$\pm\infty$	操作数
无效	NaN	操作数
不精确	round(x)	round(x)

*x 是运算的准确结果，对于单精度有 $\alpha = 192$；对于双精度有 1536，而且 $x_{max} = 1.11\ldots11 \times 2^{e_{max}}$。

当浮点运算的结果不精确时，将引发不精确异常。在 $\beta = 10$、$p = 3$ 系统中，$3.5 \otimes 4.2 = 14.7$ 是精确的，但是 $3.5 \otimes 4.3 = 15.0$ 是不精确的（因为 $3.5 \cdot 4.3 = 15.05$），这将引发不精确异常。第 46 页的“二进制到十进制的转换”讨论了使用不精确异常的算法。表 D-4 中列出了上述五类异常行为的摘要。

20.除非运算中产生“捕获”NaN，否则不引发无效异常。请参见 IEEE 标准 754-1985 的 6.2 节。–编辑者

有一个与以下事实有关的实施问题：不精确异常频繁引发。如果浮点硬件没有自己的标志，而是中断操作系统以发出存在浮点异常的信号，则不精确异常的成本可能会非常高。通过由软件来维护状态标志，可以消除此成本。首次引发异常时，设置相应类别的软件标志，并通知浮点硬件屏蔽该类异常。此后，所有后继异常都将在不中断操作系统的情况下运行。当用户重置这个状态标志时，将重新启用硬件屏蔽。

D.4.4.1 陷阱处理程序

陷阱处理程序的一个明显用途是用于向后兼容。在过去，代码会由于异常的出现而被中止，如今可通过安装陷阱处理程序来中止这样的进程。对于带有类似 `do S until (x >= 100)` 循环的代码，它尤其有用。由于将 NaN 与数值进行 <、≤、>、≥ 或 =（但不包括 ≠）比较操作时，始终会返回 “False”，因此，只要 x 成为 NaN，此代码就将进入无限循环。

在计算诸如 $\Pi_{i=1}^{n} x_i$ 等有可能导致上溢的乘积时，陷阱处理程序有一种更值得关注的用途。一种解决方案是使用对数并改为计算 $\exp(\Sigma \log x_i)$。这种方法存在的问题是：准确性较低，且计算成本高于简单表达式 Πx_i，即使没有出现上溢也是如此。另一种使用陷阱处理程序的解决方案称作**上溢 / 下溢计数**，它可以避免上述两个问题 [Sterbenz 1974]。

该方案的基本思想是：设置一个全局计数器并将其初始化为零。只要 $p_k = \Pi_{i=1}^{k} x_i$ 分乘积对于某个 k 值发生上溢，陷阱处理程序就会将计数器加 1，并将上溢量的指数部分折返后返回。在 IEEE 754 单精度中，$e_{max} = 127$，因此如果 $p_k = 1.45 \times 2^{130}$，则会发生上溢并导致调用陷阱处理程序，该程序将指数限制在范围内，p_k 被更改为 1.45×2^{-62}（请参见下面内容）。类似地，如果 p_k 发生下溢，则将计数器减 1，并将负指数折返为正指数。在完成所有乘法时，如果计数器是零，则最终乘积为 p_n。如果计数器是正数，则说明乘积上溢；如果计数器是负数，则表明乘积下溢。如果乘积的任一部分都没有超出范围，则永远不会调用陷阱处理程序，而且计算也不会引起额外成本。即使存在上溢 / 下溢，计算结果也比使用对数计算的结果更为准确，因为每个 p_k 都是使用完全精度乘法从 p_{k-1} 计算的。Barnett [1987] 讨论了一个公式，上溢 / 下溢计数的完全准确度在该公式的早期表中产生了一个误差。

IEEE 754 规定：当调用上溢或下溢陷阱处理程序时，折返结果将作为参数传递。上溢折返的定义是：计算结果时就好像先计算到无限精度，再除以 2α，然后舍入到相关的精度。对于下溢，将结果乘以 2α。指数 α 是 192（对于单精度）或 1536（对于双精度）。这就是在上面的示例中为什么将 1.45×2^{130} 转换为 1.45×2^{-62} 的原因。

D.4.4.2 舍入模式

在 IEEE 标准中，每当运算结果不精确时都将出现舍入操作，因为每个运算都是先精确计算然后再进行舍入（二进制与十进制转换除外）。默认情况下，舍入是指向最相近的数据进行舍入。该标准要求提供其他三种舍入模式，即向 0 舍入、向 +∞ 舍入和向 -∞ 舍入。在用于转换为整数运算时，向 -∞ 舍入使转换变成“取最小”函数，而向 +∞ 舍入使转换变成“取上限”函数。舍入模式影响上溢，因为当向 0 舍入或向 -∞ 舍入生效时，正量的上溢使得默认结果成为最大的可表示正数，而不是 +∞。类似地，在向 +∞ 舍入或向 0 舍入生效时，负量的上溢将产生最大的可表示负数。

舍入模式的一个应用发生在区间运算中（另一个应用将在第 46 页的“二进制到十进制的转换”中提及）。当使用区间运算时，两个数 x 和 y 的和是一个区间$[\underline{z}, \overline{z}]$，其中 $\underline{z}$ 是向 -∞ 舍入的 $x \oplus y$，$\overline{z}$ 是向 +∞ 舍入的 $x \oplus y$。加法运算的精确结果包含在区间 $[\underline{z}, \overline{z}]$ 中。如果不使用舍入模式，则区间运算常常通过计算$\underline{z} = (x \oplus y)(1 - \varepsilon)$ 和 $\overline{z} = (x \oplus y)(1 + \varepsilon)$，其中$\varepsilon$ 是计算机厄普西隆。[21] 这导致过高估计区间的大小。因为区间运算中的运算结果是一个区间，所以一般情况下运算输入也将是一个区间。如果增加了两个区间 $[\underline{x}, \overline{x}]$ 和$[\underline{y}, \overline{y}]$，结果是$[\underline{z}, \overline{z}]$，其中 $\underline{z}$ 是 $\underline{x} \oplus \underline{y}$（其舍入模式设置为向 -∞ 舍入），$\overline{z}$ 是 $\underline{z} \oplus \underline{y}$（其舍入模式设置为向 +∞ 舍入）。

当使用区间运算执行浮点计算时，最终结果是一个包含精确计算结果的区间。如果结果区间很大（它通常是这样），则这不是非常有帮助。因为正确结果可以处于该区间内的任意位置。当与多精度浮点包联合使用时，区间运算更有意义。首先通过某个精度 p 执行计算。如果区间运算指出最终结果可能不准确，则使用越来越高的精度重新计算，直到最终区间达到合理大小。

D.4.4.3 标志

IEEE 标准有许多标志和模式。如前所述，以下五类异常中的每一类都具有一个状态标志：下溢、上溢、除以零、无效运算和不精确。舍入模式有以下四种：向最相近的数据舍入、向 +∞ 舍入、向 0 舍入和向 -∞ 舍入。强烈建议对于五类异常中的每一类都设置一个启用模式位。此节给出了一些简单的示例，说明如何利用这些模式和标志。更复杂的示例将在第 46 页的“二进制到十进制的转换”节中进行讨论。

21. $\underline{Z}$ 可能大于 $\overline{Z}$（如果 x 和 y 都是负数）。– 编辑者

我们来考虑编写一个计算 x^n 的子例程，其中 n 是一个整数。当 $n > 0$ 时，简单例程类似于下面的程序段

```
PositivePower(x,n) {
 while (n is even) {
     x = x*x
     n = n/2
 }
 u = x
 while (true) {
     n = n/2
     if (n==0) return u
     x = x*x
     if (n is odd) u = u*x
 }
```

如果 $n < 0$，则计算 x^n 的更精确方法不是调用 `PositivePower(1/x, -n)`，而是调用 `1/PositivePower(x, -n)`，因为第一个表达式乘以 n 个数量，其中每个数量都具有来自除法运算（即 $1/x$）的舍入误差。在第二个表达式中它们是精确的（即 x），而且最后的除法只产生一个附加的舍入误差。可惜的是，此策略有一个微小的缺陷。如果 `PositivePower(x, -n)` 出现下溢，那么要么调用下溢陷阱处理程序，要么将设置下溢状态标志。这是错误的，因为如果 x^{-n} 出现下溢，则 x^n 将上溢或处于范围内。[22] 但是，由于 IEEE 标准允许用户访问所有标志，因此子例程可以轻松地为此进行更正。它只需关闭上溢和下溢陷阱启用位，并保存上溢和下溢状态位。然后计算 `1/PositivePower(x, -n)` 即可。如果既没有设置上溢状态位也没有设置下溢状态位，则子例程将它们与陷阱启用位一起恢复。如果设置了其中一种状态位，则子例程恢复标志并使用 `PositivePower(1/x, -n)`（它导致出现正确的异常）重新计算。

另一个使用标志的示例发生在通过以下公式计算余弦时

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}。$$

如果 arctan(∞) 计算为 $\pi/2$，则 arccos(-1) 将正确计算为 $2 \cdot \arctan(\infty) = \pi$（因为是无限运算）。但是，有一个小缺陷，因为 $(1 - x)/(1 + x)$ 的计算结果将导致除以零异常标志的设置，即使 arccos(-1) 不属于异常也是如此。此问题的解决方法很简单。只需在计算反余弦之前保存除以零标志的值，然后在计算之后恢复其旧值即可。

22. 它可以在范围内，因为如果 $x < 1$，$n < 0$ 且 x^{-n} 只是稍微小于下溢阈值 $2^{e_{\min}}$，则 $x^n \approx 2^{-e_{\min}} < 2^{e_{\max}}$，因此可能不上溢，因为在所有 IEEE 精度中，$-e_{\min} < e_{\max}$。

D.5 系统方面

对于计算机系统来说，几乎每个方面的设计都需要有关浮点的知识。计算机架构通常具有浮点指令，编译器必须生成那些浮点指令，而且操作系统必须决定当出现引发那些浮点指令的异常情况时所要执行的操作。计算机系统设计人员很少从数值分析文本中获得指导，这些文本通常针对软件的用户和编写人员，而不是针对计算机设计人员。作为似乎合理的设计决策如何导致未预料行为的示例，请参考下面的 BASIC 程序。

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
    else print "Not Equal"
```

在 IBM PC 上使用 Borland 的 Turbo Basic 进行编译和运行时，该程序打印 `Not Equal!` 此示例将在下一节中进行分析

顺便说一下，有些人认为此类异常的解决方法决不是比较浮点数以确定是否相等，而是当它们处于某个误差界限 E 内时将其视为相等。这根本不是一个解决一切问题的“灵药”，因为它提出的问题与它解决的问题一样多。E 的值应该是多少？如果 $x < 0$ 和 $y > 0$ 都在 E 内，那么是否确实应该将它们视为相等，即使它们具有不同的符号？此外，此规则定义的关系 $a \sim b \Leftrightarrow |a - b| < E$ 不是等价关系，因为 $a \sim b$ 且 $b \sim c$ 不能得出 $a \sim c$。

D.5.1 指令集

某种算法为了产生精确结果而需要更高精度的短暂成组传送，这是相当常见的。在二次公式中有这样的示例 $(-b \pm \sqrt{b^2 - 4ac})/2a$。如第 43 页的“定理 4 的证明”节中所述，当 $b^2 \approx 4ac$ 时，舍入误差最多可以影响使用二次公式计算的根中的一半数位。通过以双精度执行子计算 $b^2 - 4ac$ 时，会丢失根的双精度位的一半，这意味着将保留所有单精度位。

如果有一条对两个单精度数进行操作并产生双精度结果的乘法指令，则当 a、b 和 c 都是单精度数时按双精度计算 $b^2 - 4ac$ 是很容易的。为了产生两个 p 位数的精确舍入乘积，乘法器需要生成整个 $2p$ 位的乘积，尽管它可能在继续执行时抛弃位。这样，从单精度操作数计算双精度乘积的硬件通常只比单精度乘法器的成本略高一些，而且比双精度乘法器的成本低很多。尽管如此，目前的指令集往往仅提供生成结果与操作数精度相同的指令。[23]

如果作用两个单精度操作数以产生双精度乘积的指令仅适用于二次公式，那么将该指令增加到指令集中是不值得的。但是，此指令有许多其他用途。以解决线性方程组的问题为例，

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

23.这可能是因为设计人员喜欢“正交”指令集，在这样的指令集中浮点指令的精度与实际运算无关。乘法的特殊情况会破坏这一正交性。

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\cdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

可以按矩阵形式将它写为 $Ax = b$，其中

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2(1)n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

假定按某种方法（或许是高斯消元法）求出了一个解 $x^{(1)}$。那么有一种简单的方法可改进结果的准确度，这种方法称为**迭代改进**。首先计算

$$\xi = Ax^{(1)} - b \tag{12}$$

然后对方程组进行求解

$$Ay = \xi \tag{13}$$

请注意，如果 $x^{(1)}$ 是一个精确解，则 ξ 是零向量，y 也是零向量。通常，ξ 和 y 的计算会引起舍入误差，因此 $Ay \approx \xi \approx Ax^{(1)} - b = A(x^{(1)} - x)$，其中 x 是 （未知的）正确解。那么，$y \approx x^{(1)} - x$，因此解的改进估计是

$$x^{(2)} = x^{(1)} - y \tag{14}$$

重复执行三个步骤 (12)、 (13) 和 (14)，用 $x^{(2)}$ 替换 $x^{(1)}$，用 $x^{(3)}$ 替换 $x^{(2)}$。这里关于 $x^{(i+1)}$ 比 $x^{(i)}$ 更精确的论述仅是非正式的。有关详细信息，请参见 [Golub 和 Van Loan 1989]。

在执行迭代改进时， ξ 是一个向量，其元素是邻近不精确浮点数的差，因此会受到恶性抵消的影响。这样，迭代改进不是非常有用，除非 $\xi = Ax^{(1)} - b$ 是按双精度计算的。同样，这是计算两个单精度数 （A 和 $x^{(1)}$）的乘积的示例，它需要完全双精度结果。

总而言之，如果存在这样的指令：对两个浮点数做相乘操作并返回两倍于被操作数精度的乘积，那么将该指令增加到指令集中是很有用的。对于编译器而言，这其中的一些蕴涵将在下一节中进行讨论。

D.5.2 语言和编译器

编译器和浮点运算理论的相互影响在 Farnum [1988] 中有所讨论，本节中的讨论大都出自该论文。

D.5.2.1 含混性

理想情况下，一种语言的定义可以阐述该语言的语义，这种阐述应该足够准确，以便可以证明有关程序的声明。尽管对于语言整数节的定义通常可以满足上述要求，但涉及到浮点节时，语言定义通常具有很大的模糊性。其中的原因可能出自于以下事实：许多语言设计人员认为无法证明有关浮点的任何陈述，因为它会引起舍入误差。如果是这样，那么前面几节已经证明这一推理乃是谬论。此部分讨论语言定义中一些常见的模糊性，包括有关如何处理它们的建议。

很明显，某些语言没有清楚地指定如果 `x` 是浮点变量（例如，具有值 `3.0/10.0`），则（例如） `10.0*x` 的每次出现都必须具有相同值。以基于布朗模型的 Ada 为例，它似乎隐含着浮点运算只需满足布朗公理，这样表达式就可以具有多个可能值之一。以这种模糊的方式考虑浮点与 IEEE 模型形成了鲜明的对比，对于后者，每个浮点运算的结果都是精确定义的。在 IEEE 模型中，我们可以证明 `(3.0/10.0)*10.0` 计算为 `3`（定理 7）。在布朗模型中，我们却做不到这一点。

大多数语言定义中的另一种含混性与出现上溢、下溢以及其他异常时所发生的情况有关。IEEE 标准可以准确地指定异常的行为，因此将该标准作为模型的语言可以避免有关这一点的任何含混性。

另一模糊性与对圆括弧的解释有关。由于存在舍入误差，代数学的结合律不一定适用于浮点数。例如，当 $x = 10^{30}$，$y = -10^{30}$ 且 $z = 1$ 时，表达式 `(x+y)+z` 的结果与 `x+(y+z)` 的结果完全不同（在前一种情况下结果为 1，在后一种情况下结果是 0）。应该说，对于保留圆括弧的重要性怎么强调也不过分。定理 3、4 和 6 中提供的算法都取决于它。例如，在定理 6 中，如果不使用圆括号，则公式 $x_h = mx - (mx - x)$ 将简化为 $x_h = x$，因此破坏了整个算法。对于浮点运算，不要求先计算圆括弧中内容的语言定义是无用的。

子表达式计算在许多语言中没有精确定义。假定 ds 是双精度，但是 x 和 y 是单精度。那么，在表达式 ds + x*y 中，乘积是按单精度还是按双精度计算？另一个示例：在 x + m/n （其中 m 和 n 是整数）中，除法是整数运算还是浮点运算？有两种方法可以处理这类问题，其中的任意一种都不完全令人满意。第一种方法是，要求表达式中的所有变量具有相同的类型。这是最简单的解决方法，但是它有一些缺点。首先，具有子范围类型的语言 （如 Pascal）允许混合子范围变量和整型变量，因此禁止混合单精度变量和双精度变量就令人觉得有点奇怪了。另一个问题与常数有关。在表达式 0.1*x 中，大多数语言将 0.1 解释为单精度常数。现在，假定编程人员决定将所有浮点变量的声明从单精度更改为双精度。如果仍然将 0.1 视为单精度常数，则将出现编译时错误。编程人员将必须搜寻到每个浮点常数并对其进行更改。

第二种方法是，允许使用混合表达式，在这种情况下必须提供用于子表达式计算的规则。有许多指导性示例。 C 的最初定义要求按双精度计算每个浮点表达式 [Kernighan 和 Ritchie 1978]。这将导致出现与此节开始处提供的示例类似的异常。表达式 3.0/7.0 是按双精度计算的，但是，如果 q 是一个单精度变量，则将商舍入为单精度以便于存储。由于 3/7 是一个循环二进制分数，因此它的按双精度计算的值与它以单精度存储的值是不同的。这样，比较 $q = 3/7$ 将失败。这表明按最高可用精度计算每个表达式的值并不是一个好规则。

另一个指导性示例是内积。如果内积有数千项，则总和中的舍入误差可能会变得相当大。减少此舍入误差的一种方法是按双精度求和 （这一点将在第 34 页的 “优化器” 节中进行更详细的讨论）。如果 d 是一个双精度变量，且 x[] 和 y[] 是单精度数组，则内积循环将看起来就类似于 d = d + x[i]*y[i]。如果乘法是按单精度进行的，那么会失去双精度累积的许多优点，因为就在增加到双精度变量之前乘积被按照单精度进行截断。

涉及前面两个示例的规则是，按照出现在表达式中的任何变量的最高精度计算该表达式。那么，q = 3.0/7.0 将完全按单精度计算，[24]并具有布尔值 true，而 d = d + x[i]*y[i] 将按双精度计算，获得了双精度累积的全部优点。但是，此规则过于简单，无法完全包括所有情况。如果 dx 和 dy 是双精度变量，则表达式 y = x + single(dx-dy) 包含一个双精度变量，但是按双精度进行求和将是无意义的，因为这两个操作数都是单精度数，结果也是单精度数。

更复杂的子表达式计算规则如下。首先，为每个运算指定试验性精度，该精度是其操作数的最高精度。这种指定必须按表达式树中从叶到根的顺序执行。然后从根到叶执行第二遍。这一次，为每个运算指定最高试验性精度和它的父运算所需要的精度。在 q = 3.0/7.0 情况下，每个叶都是单精度的，因此所有运算都是按单精度执行的。在 d = d + x[i]*y[i] 情况下，乘法运算的试验性精度是单精度，但是在第二遍中它被提升到双精度，因为其父运算需要双精度操作数。而且，在 y = x + single(dx-dy) 中，加法运算是按单精度执行的。 Farnum [1988] 提供了实现此算法并不困难的证据。

24.这假定有以下常用约定：3.0 是一个单精度常数，而 3.0D0 是一个双精度常数。

此规则的缺点是，子表达式的计算取决于它被嵌入的表达式。这会产生一些令人生厌的结果。例如，假定您正在调试程序并希望知道子表达式的值。您不能简单地将子表达式键入调试器，并要求计算它，因为在程序中子表达式的值取决于它被嵌入的表达式。关于子表达式的最后一个注释是：由于将十进制常数转换为二进制是一个运算，因此计算规则还影响对十进制常数的解释。对于不能以二进制精确表示的常数（如 `0.1`），这一点尤其重要。

当一种语言将求幂作为其内置运算之一包括在内时，将出现另一个潜在模糊性。与基本算术运算不同，求幂运算的值并不总是显而易见的 [Kahan 和 Coonen 1982]。如果 `**` 是求幂运算符，则 `(-3)**3` 一定具有值 -27。但是，`(-3.0)**3.0` 是有疑问的。如果 `**` 运算符检查整数幂，那么它将 `(-3.0)**3.0` 计算为 $-3.0^3 = -27$。另一方面，如果公式 $x^y = e^{y \log x}$ 用于为实参定义 `**`，则结果可以是 NaN（当 $x < 0$ 时，使用 $\log(x) =$ `NaN` 的自然定义），这取决于 log 函数。但是，如果使用 FORTRAN `CLOG` 函数，则结果将是 -27，因为 ANSI FORTRAN 标准将 `CLOG(-3.0)` 定义为 $i\pi + \log 3$ [ANSI 1978]。程序设计语言 Ada 通过仅为整数幂定义求幂运算避免了这一问题，而 ANSI FORTRAN 禁止对负数进行实数幂操作。

事实上， FORTRAN 标准规定

> 禁止任何其结果在数学上没有定义的算术运算……

可惜的是，随着 IEEE 标准引入 ±∞，**在数学上没有定义**的含义不再是完全明确的。一种定义可能是使用第 20 页的 “无穷” 节中所示的方法。例如，要确定 a^b 的值，请考虑非常值解析函数 f 和 g，它们具有以下属性：当 $x \to 0$ 时，存在 $f(x) \to a$ 和 $g(x) \to b$。如果 $f(x)^{g(x)}$ 总是接近同一极限，则这应该是 a^b 的值。此定义将设置 $2^\infty = \infty$，这似乎是相当合理的。在 1.0^∞ 的情况下，当 $f(x) = 1$ 且 $g(x) = 1/x$ 时，极限接近 1，但是当 $f(x) = 1 - x$ 且 $g(x) = 1/x$ 时，极限是 e^{-1}。因此 1.0^∞ 应该是一个 NaN。在 0^0 的情况下，$f(x)^{g(x)} = e^{g(x) \log f(x)}$。因为 f 和 g 都是可解析的并在 0 处具有值 0，所以 $f(x) = a_1x^1 + a_2x^2 + \ldots$ 且 $g(x) = b_1x^1 + b_2x^2 + \ldots$。这样，
$\lim_{x \to 0} g(x) \log f(x) = \lim_{x \to 0} x \log(x(a_1 + a_2x + \ldots)) = \lim_{x \to 0} x \log(a_1x) = 0$。因此，对于 f 和 g，$f(x)^{g(x)} \to e^0 = 1$，这意味着 $0^0 = 1$.[25] [26] 使用此定义将明确地定义所有参数的指数函数，尤其是将 `(-3.0)**3.0` 定义为 -27。

25.结论 $0^0 = 1$ 取决于以下限制：f 是非常值函数。如果去掉此限制，那么允许 f 是恒等于 0 的函数就会将 0 作为 $\lim_{x \to 0} f(x)^{g(x)}$ 的可能值，因此必须将 0^0 定义为一个 NaN。

26.在 0^0 的情况下，可以提出似乎合理的论点，但令人信服的论点则包括在 “Concrete Mathematics”（作者：Graham、 Knuth 和 Patashnik）中，该论点是：要使二项式定理成立，必须满足 $0^0 = 1$。—编辑者

D.5.2.2 IEEE 标准

第 14 页的“IEEE 标准”节讨论了 IEEE 标准的许多特性。但是，IEEE 标准没有说明如何通过程序设计语言获得这些特性。因而，在支持该标准的浮点硬件与程序设计语言（如 C、Pascal 或 FORTRAN）之间常常存在着不匹配。某些 IEEE 功能可以通过子例程调用库获得。例如，IEEE 标准要求精确舍入平方根，而平方根函数通常是直接在硬件中实现的。通过平方根例程库可以轻松地获得此功能。但是，该标准的其他方面不能通过子例程来轻松实现。例如，大多数计算机语言最多指定两种浮点类型，而 IEEE 标准具有四种不同精度（尽管建议的配置是单精度加上单精度扩展或者单精度、双精度和双精度扩展）。另一示例是无穷大。表示 $\pm\infty$ 的常量可以由子例程提供。但是这样一来，它们可能无法用在需要常量表达式的位置上，例如用来初始化一个常量。

更微妙的情况是操纵与计算关联的状态，该状态由舍入模式、陷阱启用位、陷阱处理程序和异常标志组成。一种方法是为读取和写入状态提供子例程。此外，能够以原子方式设置新值并返回旧值的单次调用通常很有用。如第 26 页的“标志”节中的示例所示，修改 IEEE 状态的一种常见模式为仅在块或子例程的作用域内更改它。这样，查找块的每个出口并确保状态已恢复是编程人员的责任。如果语言支持在块的作用域 内精确设置状态，那将是很有用的。 Modula-3 是一种为陷阱处理程序实现此思想的语言 [Nelson 1991]。

在一种语言中实施 IEEE 标准时，许多次要事项还需考虑。因为对于所有 x，$x - x = +0$，所以 [27] $(+0) - (+0) = +0$。但是，$-(+0) = -0$，因此不应该将 $-x$ 定义为 $0 - x$。NaN 的引入可能是令人迷惑的，因为 NaN 永不等于任何其他数（包括另一个 NaN），因此 $x = x$ 不再总是正确的。事实上，如果没有提供 IEEE 推荐的函数 `Isnan`，则表达式 $x \neq x$ 是测试 NaN 的最简单方法。此外，NaN 不与所有其他数一起排序，因此不能将 $x \leq y$ 定义为**不是** $x > y$。由于 NaN 的引入导致浮点数变成部分排序的，因此使用返回 <、 =、 > 或 *unordered* 之一的 `compare` 函数会让编程人员可以更轻松地处理比较操作。

虽然 IEEE 标准规定如果任一操作数是 NaN 则基本浮点运算返回 NaN，但是这不可能始终是复合运算的最佳定义。例如，当计算绘制曲线图要使用的适当比例因子时，必须计算一组值的最大值。在这种情况下，最大值运算简单地忽略 NaN 是有意义的。

最后，舍入可以是一个问题。IEEE 标准非常精确地定义了舍入，而且它依赖于舍入模式的当前值。有时，这与类型转换中隐式舍入的定义或语言中的显式 `round` 函数存在冲突。也就是说，希望使用 IEEE 舍入的程序无法使用自然语言的基本要素，反过来，语言基本要素无法在数量不断增加的 IEEE 计算机上进行实施。

27.除非舍入模式是向 $-\infty$ 舍入（在这种情况下，$x - x = -0$）。

D.5.2.3 优化器

编译器教科书往往忽略浮点这一主题。例如， Aho 等 [1986] 提到将 `x/2.0` 替换为 `x*0.5`，引导读者假定 `x/10.0` 应该替换为 `0.1*x`。但是，在二进制计算机上这两个表达式并不具有相同的语义，因为 0.1 无法以二进制精确表示。此教科书还建议将 `x*y-x*z` 替换为 `x*(y-z)`，尽管我们已经看到在 $y \approx z$ 时这两个表达式可以具有完全不同的值。尽管其中指出优化器不应违反语言定义，从而对 “可以使用任意代数恒等式优化代码”这种说法做了限制，但是它给人们留下了浮点语义并不太重要的印象。不管语言标准是否指定必须先计算圆括弧中的内容， `(x+y)+z` 都可以具有与 `x+(y+z)` 完全不同的结果，如上所述。存在一个与保留圆括弧密切相关的问题，如以下代码所示：

```
eps = 1;
do eps = 0.5*eps; while (eps + 1 > 1);
```

此代码旨在估算出计算机厄普西隆。如果进行优化的编译器注意到 $eps + 1 > 1 \Leftrightarrow eps > 0$，将完全更改程序。它将不计算最小数 x （以使 $1 \oplus x$ 仍然大于 1 ($x \approx e \approx \beta^{-p}$)，而是计算最大数 x （对于它， $x/2$ 舍入为 0 ($x \approx \beta^{e_{\min}}$)。避免这种 “优化”是如此重要以致于值得提供另一被其完全毁坏的非常有用的算法。

许多问题 （如数值积分和微分方程的数值解）涉及计算多个项的和。因为每个加法运算都有可能引入大至 .5 ulp 的误差，所以涉及数千项的求和会具有相当大的舍入误差。纠正这一点的简单方法是将部分被加数存储在双精度变量中，并使用双精度执行每个加法运算。如果计算是使用单精度进行的，则在大多数计算机系统上按双精度执行求和是很容易的。但是，如果已经按双精度进行了计算，则使精度加倍就不那么简单了。有时建议的一种方法是对数值进行排序，然后按照从最小到最大的顺序将其相加。但是，有一种大大提高求和准确度，而且效率高得多的方法，即

D.5.2.4 定理 8 （Kahan 求和公式）

假设 $\Sigma_{j=1}^{N} x_j$ 是使用以下算法计算的

```
S = X[1];
C = 0;
for j = 2 to N {
    Y = X[j] - C;
    T = S + Y;
    C = (T - S) - Y;
    S = T;
}
```

那么，计算的和 S *等于* $\Sigma x_j(1+\delta_j) + O(N\varepsilon^2)\Sigma|x_j|$，*其中* ($|\delta_j| \le 2\varepsilon$) 。

使用简单公式 Σx_j，计算的和等于 $\Sigma x_j(1+\delta_j)$，其中 $|\delta_j| < (n-j)e$。将此与 Kahan 求和公式中的误差进行比较，可以看出有显著的改进。每个被加数受到只有 $2e$ 的扰动，而不是简单公式中多至 ne 的扰动。有关详细信息，请参见第 47 页的 “求和中的误差”。

认为浮点运算遵循代数法则的优化器将得出以下结论：$C = [T-S] - Y = [(S+Y)-S] - Y = 0$，从而认为该算法毫无用处。可以这样概括这些示例：在涉及浮点变量的表达式中应用代数恒等式进行优化时，应该极其小心，尽管这些恒等式在数学上对所有实数成立。

优化器可以更改浮点代码语义的另一种方法涉及到常数。在表达式 `1.0E-40*x` 中，隐含了一个十进制到二进制的转换操作，该操作将十进制数转换成了二进制常数。由于此常数无法以二进制精确表示，因此会引发不精确异常。此外，如果表达式是按单精度计算的，则下溢标志将被设置。由于常数是不精确的，因此其到二进制的精确转换取决于 IEEE 舍入模式的当前值。这样，在编译时将 `1.0E-40` 转换为二进制数的优化器将更改程序的语义。但是，可以按最低可用精度精确表示的常数 （如 27.5）能够在编译时安全地进行转换，因为它们始终是精确的，不会引发任何异常且不受舍入模式影响。希望在编译时就进行转换的常数应该使用常量声明，例如 `const pi = 3.14159265`。

关于优化可以更改浮点语义的另一示例是公共子表达式的消除，如以下代码所示

```
C = A*B;
RndMode = Up
D = A*B;
```

虽然 `A*B` 可能看上去是一个公共子表达式，但实际上并不是，因为两个计算位置的舍入模式有所不同。最后三个示例：$x = x$ 不能由布尔常量 `true` 替换，因为当 x 是一个 NaN 时前者将失败；当 $x = +0$ 时，$-x = 0 - x$ 将失败；而且， $x < y$ 并不与 $x \geq y$ 相反，因为 NaN 既不大于也不小于普通浮点数。

尽管存在这些示例，但是仍然存在着一些可以对浮点代码进行的有用优化。首先，存在对浮点数有效的代数恒等式。 IEEE 运算中的一些示例是：$x + y = y + x$、$2 \times x = x + x$、$1 \times x = x$ 和 $0.5 \times x = x/2$。但是，甚至这些简单的恒等式也会在一些计算机（如 CDC 和 Cray 超级计算机）上失败。指令调度和内联过程替换是其他两种可能有用的优化。[28]

作为最后一个示例，请考虑表达式 `dx = x*y`，其中 `x` 和 `y` 是单精度变量，`dx` 是双精度变量。在具有将两个单精度数相乘以产生双精度数的指令的计算机上，可以将 `dx = x*y` 映射到该指令，而不是编译为将操作数转换为双精度数再执行双精度数与双精度数相乘的一系列指令。

28.VAX 上的 VMS 数学库使用弱形式的内联过程替换，因为它们使用转向子例程调用的廉价跳转，而不是使用慢速的 CALLS 和 CALLG 指令。

一些编译器编写人员认为禁止将 $(x + y) + z$ 转换为 $x + (y + z)$ 的限制与他们无关，只有使用不可移植方法的编程人员才关心此限制。也许他们认为浮点数与实数非常类似，应该遵循与实数相同的定律。实数语义的问题是它们的实现成本极高。每次将两个 n 位数相乘时，积将具有 $2n$ 位。每次将两个指数相差很大的 n 位数相加时，和中的位数是 n + 指数之间的差值。和最多可以具有 $(e^{max} - e^{min})$ + n 位，或大约 $2 \cdot e^{max}$ + n 位。产生数千运算的算法 （例如对线性方程组求解）不久将对具有许多有效位的数进行运算，且运算速度慢得令人觉得无望。库函数（例如 sin 和 cos）的实现甚至更困难，因为这些超越函数的值不是有理数。精确的整数运算通常由 lisp 系统提供，对于解决某些问题是很方便的。但是，精确的浮点运算几乎是没有用的。

事实是，存在利用 $(x + y) + z \neq x + (y + z)$ 事实的有用算法 （如 Kahan 求和公式），而且只要界限

$$a \oplus b = (a + b)(1 + \delta)$$

成立 （以及 小 -、× 和 / 的类似界限），这些算法就是适用的。由于这些界限对于几乎所有商业硬件都是成立的，因此数值编程人员忽略这样的算法将是不明智的，编译器编写人员通过伪称浮点变量具有实数语义来破坏这些算法将是不负责任的。

D.5.3 异常处理

到目前为止所讨论的主题主要涉及准确性和精度的系统含意。陷阱处理程序也引发了一些值得关注的系统问题。IEEE 标准强烈建议用户应该能够为五类异常中的每一类都指定陷阱处理程序，第 25 页的“陷阱处理程序”节给出了用户定义的陷阱处理程序的一些应用。在出现无效运算异常和除以零异常的情况下，应该为处理程序提供操作数，否则应该提供精确舍入的结果。根据所使用的程序设计语言，陷阱处理程序也许还能够访问程序中的其他变量。对于所有异常，异常处理程序必须能够识别正在执行什么运算以及该运算的目标精度是什么。

IEEE 标准假定运算在概念上是串行的，当发生中断时，识别出运算及其操作数是可能的。在采用流水线技术或具有多个运算单元的计算机上，当出现异常时，只是让陷阱处理程序检查程序计数器可能是不够的。可能需要能够精确识别出捕获哪个运算的硬件支持。

另一问题如以下程序片段所示。

```
x = y*z;
z = x*w;
a = b + c;
d = a/x;
```

假定第二个乘法运算引发一个异常，陷阱处理程序希望使用 a 的值。在可以并行执行加法运算和乘法运算的硬件上，优化器有可能将加法运算提到第二个乘法运算之前，以便加法运算可以与第一个乘法运算并行进行。这样，当第二个乘法运算被捕获时，a = b + c 已经执行，潜在地更改了 a 的结果。编译器避开此类优化将不是合理的，因为每个浮点运算都有可能被捕获，从而几乎所有指令调度优化都将被消除。通过禁止陷阱处理程序直接访问程序的任何变量，可以避免这一问题。取而代之，可以将操作数或结果作为参数提供给处理程序。

但是，仍然存在问题。在以下片段中

```
x = y*z;
z = a + b;
```

这两个指令也许能够并行执行。如果乘法运算被捕获，则其参数 z 可能已经被加法运算覆盖，尤其是因为加法运算的速度通常比乘法运算快。支持 IEEE 标准的计算机系统必须提供保存 z 值的某种方法，要么在硬件中，要么让编译器首先避免这样的情况。

W. Kahan 曾提议使用**预替换**代替陷阱处理程序以避免这些问题。在此方法中，用户指定一个异常以及希望在出现该异常时用作结果的值。作为一个示例，假定在用于计算 (sin x)/x 的代码中，用户断定 $x = 0$ 如此少见以致于避免测试 $x = 0$ 将提高性能，因此改为在出现 0/0 陷阱时处理这种情况。当使用 IEEE 陷阱处理程序时，用户将编写返回值 1 的处理程序并在计算 sin x/x 之前安装它。当使用预替换时，用户将指定发生无效运算时应该使用值 1。Kahan 将此称为预替换，因为要使用的值必须在出现异常之前指定。而当使用陷阱处理程序时，要返回的值可以在出现陷阱时进行计算。

预替换的优点是它具有简单的硬件实现。[29] 一旦确定异常的类型，就可以使用它为包含所需运算结果的表建立索引。虽然预替换具有一些吸引人的属性，但是 IEEE 标准的广泛认可使其不大可能被硬件制造商广泛执行。

D.6 详细资料

在本文中已经作出了许多有关浮点运算属性的断言。现在，我们继续说明浮点并非是神秘的魔术，而是一个可以在数学上检验其断言的简单易懂的主题。说明分为三个部分。第一部分介绍误差分析，并提供第 2 页的 “舍入误差” 节的详细资料。第二部分探讨二进制到十进制的转换，对第 14 页的 “IEEE 标准” 节进行一些补充。第三部分讨论 Kahan 求和公式，它曾在第 28 页的 “系统方面” 节中用作一个示例。

29.预替换的困难在于它需要直接硬件实现或可持续的浮点陷阱 （如果在软件中实现）。– 编辑者

D.6.1 舍入误差

在讨论舍入误差时，已经指出单个保护数位足以保证加法运算和减法运算将始终是准确的（定理 2）。现在我们继续检验这一事实。定理 2 分为两个部分，一个部分用于减法运算，另一部分用于加法运算。用于减法运算的部分是

D.6.1.1 定理 9

如果 x 和 y 是正的浮点数，其格式为具有参数 β 和 p，而且减法运算是使用 $p + 1$ 位（也就是使用一个保护数位）进行的，则结果中的相对舍入误差小于

$$\left(\frac{\beta}{2} + 1\right)\beta^{-p} = \left(1 + \frac{2}{\beta}\right)e \leq 2e \text{ 。}$$

D.6.1.2 证明

有必要的话，x 和 y 可以互换，因此不妨认为 $x > y$。同样，不妨将 x 和 y 按比例缩放，以便 x 可以表示成 $x_0.x_1 \ldots x_{p-1} \times \beta^0$。如果将 y 表示为 $y_0.y_1 \ y_{p-1}$，则差值是精确的。如果将 y 表示为 $0.y_1 \ldots y_p$，则保护数位确保计算的差值将是舍入为浮点数的精确差值，因此舍入误差最大为 e。对于一般情形，令 $y = 0.0 \ldots 0y_{k+1} \ldots y_{k+p}$ 且 $\bar{y}$ 是截断到 $p + 1$ 位的 y。那么

$$y - \bar{y} < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \ldots + \beta^{-p-k})。 \tag{15}$$

按照保护数位的定义，$x - y$ 的计算值是舍入为浮点数的 $x - \bar{y}$，即 $(x - \bar{y}) + \delta$，其中舍入误差 δ 满足以下条件

$$|\delta| \leq (\beta/2)\beta^{-p}。 \tag{16}$$

精确差值是 $x - y$，因此误差是 $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y + \delta$。存在三种情况。如果 $x - y \geq 1$，则相对误差的界限是

$$\frac{y - \bar{y} + \delta}{1} \leq \beta^{-p}\left[(\beta - 1)(\beta^{-1} + \ldots + \beta^{-k}) + \beta/2\right] < \beta^{-p}(1 + \beta/2)。 \tag{17}$$

第二，如果 $x - \bar{y} < 1$，则 $\delta = 0$。因为 $x - y$ 最小可以是

$1.0 - 0.\underbrace{0\ldots0}_{k}\underbrace{\rho\ldots\rho}_{p} > (\beta - 1)(\beta^{-1} + \ldots + \beta^{-k})$，其中 $\rho = \beta - 1$，

在这种情况下，相对误差的界限是

$$\frac{y-\bar{y}+\delta}{(\beta-1)(\beta^{-1}+\ldots+\beta^{-k})} < \frac{(\beta-1)\beta^{-p}(\beta^{-1}+\ldots+\beta^{-k})}{(\beta-1)(\beta^{-1}+\ldots+\beta^{-k})} = \beta^{-p}。 \tag{18}$$

最后一种情况是：$x - y < 1$ 但 $x - \bar{y} \geq 1$。发生这种情况的唯一条件是 $x - \bar{y} = 1$，在这种情况下 $\delta = 0$。但是，如果 $\delta = 0$，则 (18) 是适用的，因此相对误差的界限同样是 $\beta^{-p} < \beta^{-p}(1 + \beta/2)$。■

当 $\beta = 2$ 时，界限正好是 $2e$，此界限是在 $p \to \infty$ 时在极限中为 $x = 1 + 2^{2-p}$ 和 $y = 2^{1-p} - 2^{1-2p}$ 实现的。将符号相同的数相加时，不使用保护数位也能得到良好的精确度，如以下结果所示。

D.6.1.3 定理 10

如果 $x \geq 0$ 且 $y \geq 0$，则计算 $x + y$ 时的相对误差最大为 2ε，即使未使用保护数位也是如此。

D.6.1.4 证明

使用 k 个保护数位的加法运算的算法与用于减法运算的算法类似。如果 $x \geq y$，则对 y 进行右移位，直到 x 和 y 的小数点对齐。舍弃移过 $p + k$ 位置的任何位。精确计算这两个 $p + k$ 位数的和。然后舍入为 p 位。

我们将检验不使用保护数位时该定理是否成立；一般的情况是类似的。不失一般性，假设 $x \geq y \geq 0$ 并将 x 乘以某一因子使其具有 $d.dd\ldots d \times \beta^0$ 的形式。首先，假定没有进位。那么，从 y 的末尾移掉的位所具有的值小于 β^{-p+1}，且和至少为 1，因此相对误差小于 $\beta^{-p+1}/1 = 2e$。如果有进位，则必须将移位误差增加到

$$\frac{1}{2}\beta^{-p+2}。$$

和至少为 β，因此相对误差小于

$$\left(\beta^{-p+1} + \frac{1}{2}\beta^{-p+2}\right)/\beta = (1 + \beta/2)\beta^{-p} \leq 2\varepsilon \quad 。■$$

显而易见，结合这两个定理可以得出定理 2。定理 2 给出了执行一个运算的相对误差。比较 $x^2 - y^2$ 和 $(x + y)(x - y)$ 的舍入误差要求知道乘法运算的相对误差。$x \ominus y$ 的相对误差是 $\delta_1 = [(x \ominus y) - (x - y)] / (x - y)$，它满足 $|\delta_1| \leq 2e$。或者，以另一种方式书写

$$x \ominus y = (x - y)(1 + \delta_1), \quad |\delta_1| \leq 2e \tag{19}$$

类似地

$$x \oplus y = (x + y)\,(1 + \delta_2), \quad |\delta_2| \le 2e \tag{20}$$

假定乘法是通过计算精确乘积再进行舍入来执行的，相对误差最大为 .5 ulp，那么，对于任何浮点数 u 和 v

$$u \otimes v = uv\,(1 + \delta_3), \qquad |\delta_3| \le e \tag{21}$$

将这三个等式放在一起 （假设 $u = x \ominus y$ 且 $v = x \oplus y$），会得出

$$(x \ominus y) \otimes (x \oplus y) = (x - y)\,(1 + \delta_1)\,(x + y)\,(1 + \delta_2)\,(1 + \delta_3) \tag{22}$$

因此，在计算 $(x - y)\,(x + y)$ 时引起的相对误差是

$$\frac{(x-y) \ominus (x+y) - (x^2 - y^2)}{(x^2 - y^2)} = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) - 1 \tag{23}$$

此相对误差等于 $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3 + \delta_1\delta_2\delta_3$，其边界是 $5\varepsilon + 8\varepsilon^2$。换句话说，最大相对误差大约是舍入误差的 5 倍 （因为 e 是一个很小的数， e^2 几乎可以忽略）。

对 $(x \otimes x) \ominus (y \otimes y)$ 的类似分析无法得出很小的相对误差值，因为将 x 和 y 的两个接近值代入 $x^2 - y^2$ 时，相对误差通常是相当大的。查看它的另一方法是尝试并重复对 $(x \ominus y) \otimes (x \oplus y)$ 进行的分析，将会得出

$$\begin{aligned}(x \otimes x) \ominus (y \otimes y) &= [x^2(1 + \delta_1) - y^2(1 + \delta_2)]\,(1 + \delta_3) \\ &= ((x^2 - y^2)\,(1 + \delta_1) + (\delta_1 - \delta_2)y^2)\,(1 + \delta_3)\end{aligned}$$

当 x 和 y 接近时，误差项 $(\delta_1 - \delta_2)y^2$ 可以与结果 $x^2 - y^2$ 一样大。这些计算证明我们的以下断言是正确的：$(x - y)\,(x + y)$ 比 $x^2 - y^2$ 更精确。

接下来，我们分析三角形面积的公式。为了估算在使用 (7) 计算时出现的最大误差，将需要以下定理。

D.6.1.5 定理 11

如果减法运算使用了保护数位且操作数满足 $y/2 \le x \le 2y$，则 $x - y$ 可以精确算出。

D.6.1.6 证明

请注意，如果 x 和 y 的指数相同，则 $x \ominus y$ 一定是精确的。否则，根据定理中的条件，指数最多可以相差 1。如有必要，x 和 y 可以按比例缩放并互换，从而使 $0 \le y \le x$，并将 x 表示为 $x_0.x_1 \ldots x_{p-1}$，将 y 表示为 $0.y_1 \ldots y_p$。那么，用于计算 $x \ominus y$ 的算法将精确计

算 $x - y$ 并舍入到一个浮点数。如果差的形式为 $0.d_1 \ldots d_p$，则差值的长度将已经是 p 位，因此不必进行舍入。由于 $x \le 2y$, $x - y \le y$，且 y 的形式是 $0.d_1 \ldots d_p$，因此 $x - y$ 也满足这样的形式。▮

当 $\beta > 2$ 时，定理 11 的假设不能由 $y/\beta \le x \le \beta y$ 替换；更严格的条件 $y/2 \le x \le 2y$ 仍然是必需的。在定理 10 被证明之后，随即对 $(x - y)(x + y)$ 中的误差进行了分析，其中利用了加法和减法两种基本运算的相对误差很小（即等式 (19) 和 (20)）这一事实。这是一种最常见的误差分析。但是，就如同以下证明过程所显示的那样，分析公式 (7) 还需要其他内容，即定理 11。

D.6.1.7 定理 12

如果减法运算使用保护数位，而且 a、b 和 c 是三角形的边 $(a \ge b \ge c)$，则计算 $(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$ 时的相对误差最大为 16ε，条件是 $e < .005$。

D.6.1.8 证明

让我们逐一检查各因子。依据定理 10，$b \oplus c = (b + c)(1 + \delta_1)$，其中 δ_1 是相对误差，$|\delta_1| \le 2\varepsilon$。所以，第一个因子的值是

$$(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1))(1 + \delta_2),$$

因此

$$\begin{aligned}(a + b + c)(1 - 2\varepsilon)^2 &\le [a + (b + c)(1 - 2\varepsilon)] \cdot (1 - 2\varepsilon) \\ &\le a \oplus (b \oplus c) \\ &\le [a + (b + c)(1 + 2\varepsilon)](1 + 2\varepsilon) \\ &\le (a + b + c)(1 + 2\varepsilon)^2\end{aligned}$$

这意味着存在一个 η_1 满足

$$(a \oplus (b \oplus c)) = (a + b + c)(1 + \eta_1)^2, \ |\eta_1| \le 2\varepsilon。 \tag{24}$$

下一项涉及 c 和 $a \ominus b$ 的有可能是恶性的相减，因为 $a \ominus b$ 可能具有舍入误差。因为 a、b 和 c 是三角形的边，所以 $a \le b + c$，将此与排序 $c \le b \le a$ 结合在一起就可以得出 $a \le b + c \le 2b \le 2a$。因此，$a - b$ 满足定理 11 的条件。这意味着 $a - b = a \ominus b$ 是精确的，因此 $c \ominus (a - b)$ 是一个无害相减，可以依据定理 9 进行估算，估算结果为

$$(c \ominus (a \ominus b)) = (c - (a - b))(1 + \eta_2), \ |\eta_2| \le 2\varepsilon \tag{25}$$

第三项是两个精确正量的和，因此有

$$(c \oplus (a \ominus b)) = (c + (a - b))\,(1 + \eta_3),\ |\eta_3| \le 2\varepsilon \tag{26}$$

最后一项是

$$(a \oplus (b \ominus c)) = (a + (b - c))\,(1 + \eta_4)^2,\ |\eta_4| \le 2\varepsilon, \tag{27}$$

其中同时使用了定理 9 和定理 10。如果假定乘法运算精确舍入，使得 $x \otimes y = xy(1 + \zeta)$ 且 $|\zeta| \le \varepsilon$，那么将 (24)、(25)、(26) 和 (27) 结合在一起就可以得出

$$\begin{aligned}&(a \oplus (b \oplus c))\,(c \ominus (a \ominus b))\,(c \oplus (a \ominus b))\,(a \oplus (b \ominus c))\\ &\quad \le (a + (b + c))\,(c - (a - b))\,(c + (a - b))\,(a + (b - c))\,E\end{aligned}$$

其中

$$E = (1 + \eta_1)^2\,(1 + \eta_2)\,(1 + \eta_3)\,(1 + \eta_4)^2\,(1 + \zeta_1)(1 + \zeta_2)\,(1 + \zeta_3)$$

E 的一个上界是 $(1 + 2\varepsilon)^6(1 + \varepsilon)^3$，其展开结果是 $1 + 15\varepsilon + O(\varepsilon^2)$。有些作者干脆忽略了 $O(e^2)$ 项，但其实说明这一项是很容易的。令 $(1 + 2\varepsilon)^6(1 + \varepsilon)^3 = 1 + 15\varepsilon + \varepsilon R(\varepsilon)$，其中 $R(\varepsilon)$ 是一个具有正系数的、e 的多项式，由此可知它是 ε 的递增函数。因为 $R(.005) = .505$，所以对于所有的 $\varepsilon < .005$，$R(\varepsilon) < 1$，由此可得 $E \le (1 + 2\varepsilon)^6(1 + \varepsilon)^3 < 1 + 16\varepsilon$。求 E 的一个下界时，我们注意 $1 - 15\varepsilon - \varepsilon R(\varepsilon) < E$，因此当 $\varepsilon < .005$ 时，$1 - 16\varepsilon < (1 - 2\varepsilon)^6(1 - \varepsilon)^3$。将这两个界限结合在一起可得 $1 - 16\varepsilon < E < 1 + 16\varepsilon$。因此，相对误差最大为 16ε。∎

定理 12 明白地显示在公式 (7) 中不存在恶性抵消。因此，虽然不必显示公式 (7) 在数值上是稳定的，但是具有整个公式的界限是令人满意的，这正是第 6 页的 “抵消” 中的定理 3 给出的。

D.6.1.9 定理 3 的证明

假设

$$q = (a + (b + c))\,(c - (a - b))\,(c + (a - b))\,(a + (b - c))$$

且

$$Q = (a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))。$$

然后，由定理 12 得 $Q = q(1 + \delta)$，且 $\delta \le 16\varepsilon$。可以很容易地得到

$$1 - 0.52|\delta| \le \sqrt{1 - |\delta|} \le \sqrt{1 + |\delta|} \le 1 + 0.52|\delta| \tag{28}$$

条件是 $\delta \le .04/(.52)^2 \approx .15$，因为 $|\delta| \le 16\varepsilon \le 16(.005) = .08$，所以 δ 确实满足此条件。因此

$$\sqrt{Q} = \sqrt{q(1+\delta)} = \sqrt{q}(1+\delta_1),$$

且 $|\delta_1| \le .52|\delta| \le 8.5\varepsilon$。如果将计算平方根时的误差控制在 .5 ulp 内，则计算 $\sqrt{Q}$ 时的误差是 $(1+\delta_1)(1+\delta_2)$，其中 $|\delta_2| \le \varepsilon$。如果 $\beta = 2$，则在除以 4 时不会产生进一步的误差。否则，在做除法时还需要另一因子 $1+\delta_3$（$|\delta_3| \le \varepsilon$），通过使用证明定理 12 时的方法，$(1+\delta_1)(1+\delta_2)(1+\delta_3)$ 的最终误差界限取决于 $1+\delta_4$（$|\delta_4| \le 11\varepsilon$）。∎

要使紧跟定理 4 陈述之后的启发式说明变得精确，下一个定理正好说明 $\mu(x)$ 近似一个常数时的接近程度。

D.6.1.10 定理 13

如果 $\mu(x) = \ln(1+x)/x$，*则对于* $0 \le x \le \frac{3}{4}$，$\frac{1}{2} \le \mu(x) \le 1$ *且导数满足* $|\mu'(x)| \le \frac{1}{2}$。

D.6.1.11 证明

请注意，$\mu(x) = 1 - x/2 + x^2/3 - \ldots$ 是具有递减项的交错级数，因此对于 $x \le 1$，$\mu(x) \ge 1 - x/2 \ge 1/2$。可以更容易地看出：因为 μ 的级数是交错的，所以 $\mu(x) \le 1$。$\mu'(x)$ 的泰勒级数也是交错的，而且如果 $x \le \frac{3}{4}$ 具有递减项，那么 $-\frac{1}{2} \le \mu'(x) \le -\frac{1}{2} + 2x/3$，或者 $-\frac{1}{2} \le \mu'(x) \le 0$，因此 $|\mu'(x)| \le \frac{1}{2}$。∎

D.6.1.12 定理 4 的证明

因为 ln 的泰勒级数

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots$$

是一个交错级数，所以 $0 < x - \ln(1+x) < x^2/2$，按 x 近似 $\ln(1+x)$ 时产生的相对误差的界限是 $x/2$。如果 $1 \oplus x = 1$，则 $|x| < \varepsilon$，因此相对误差的界限是 $\varepsilon/2$。

当 $1 \oplus x \neq 1$ 时，定义 $\hat{x}$ 满足 $1 \oplus x = 1 + \hat{x}$ 。又因为 $0 \leq x < 1$，所以 $(1 \oplus x) \ominus 1 = \hat{x}$ 。如果要将计算除法和对数时的误差控制在 $\frac{1}{2}$ ulp 内，则表达式 $\ln(1 + x)/((1 + x) - 1)$ 的计算值是

$$\frac{\ln(1 \oplus x)}{(1 \oplus x) \ominus 1}(1 + \delta_1)(1 + \delta_2) = \frac{\ln(1 + \hat{x})}{\hat{x}}(1 + \delta_1)(1 + \delta_2) = \mu(\hat{x})(1 + \delta_1)(1 + \delta_2) \tag{29}$$

其中 $|\delta_1| \leq \varepsilon$ 且 $|\delta_2| \leq \varepsilon$。要估算 $\mu(\hat{x})$，请使用中值定理，该定理指出

$$\mu(\hat{x}) - \mu(x) = (\hat{x} - x)\mu'(\xi) \tag{30}$$

其中 ξ 介于 x 和 $\hat{x}$ 之间。根据 $\hat{x}$ 的定义，可以得出 $|\hat{x} - x| \leq \varepsilon$，将此与定理 13 结合在一起可以得出 $|\mu(\hat{x}) - \mu(x)| \leq \varepsilon/2$，或 $|\mu(\hat{x})/\mu(x) - 1| \leq \varepsilon/(2|\mu(x)|) \leq \varepsilon$，这意味着 $\mu(\hat{x}) = \mu(x)(1 + \delta_3)$，$|\delta_3| \leq \varepsilon$。最后，乘以 x 会引入 δ_4，因此

$x \cdot \ln(1 \oplus x)/((1 \oplus x) \ominus 1)$ 的计算值

是

$$\frac{\ln(1 + x)}{1 + x) - 1}(1 + \delta_1)(1 + \delta_2)(1 + \delta_3)(1 + \delta_4), \qquad \left|\delta_I\right| \leq$$

很容易得出：如果 $\varepsilon < 0.1$，则

$$(1 + \delta_1)(1 + \delta_2)(1 + \delta_3)(1 + \delta_4) = 1 + \delta,$$

其中 $|\delta| \leq 5\varepsilon$。∎

使用公式 (19)、(20) 和 (21) 进行误差分析的一个有趣示例出现在二次方程求根公式 $(-b \pm \sqrt{b^2 - 4ac})/2a$ 中。第 6 页的“抵消”节解释改写公式将如何消除 ± 运算引起的潜在抵消。但是，在计算 $d = b_2 - 4ac$ 时还可能出现另一个抵消。通过对公式进行简单的重新排列不能消除该潜在抵消。大致说来，当 $b^2 \approx 4ac$ 时，舍入误差最多可以影响使用二次方程求根公式计算的根中数位的一半。下面是一个非正式证明（估算二次方程求根公式误差的另一种方法见于 Kahan [1972]）。

如果 $b^2 \approx 4ac$，则舍入误差最多可以影响使用二次方程求根公式 $(-b \pm \sqrt{b^2 - 4ac})/2a$。

证明：记 $(b \otimes b) \ominus (4a \otimes c) = (b^2(1 + \delta_1) - 4ac(1 + \delta_2))(1 + \delta_3)$，其中 $|\delta_i| \leq \varepsilon$。[30] 令 $d = b^2 - 4ac$，上式可改写为 $(d(1 + \delta_1) - 4ac(\delta_2 - \delta_1))(1 + \delta_3)$。为获得此误差的估计大小，忽略 δ_i 中的二次项，此时绝对误差为 $d(\delta_1 + \delta_3) - 4ac\delta_4$，其中 $|\delta_4| = |\delta_1 - \delta_2| \leq 2\varepsilon$。因为

30.在此非正式证明中，假定 $\beta = 2$，使得乘以 4 的结果是精确的，且不需要 δ_i。

$d \ll 4ac$，所以可以忽略第一项 $d(\delta_1 + \delta_3)$。要估算第二项，可令 $ax^2 + bx + c = a(x - r_1)(x - r_2)$，因此 $ar_1r_2 = c$。因为 $b^2 \approx 4ac$，所以 $r_1 \approx r_2$，可得第二误差项为 $4ac\delta_4 \approx 4a^2r_1\delta_4^2$。这样，$\sqrt{d}$ 的计算值是

$$\sqrt{d + 4a^2r_1^2\delta_4}$$

不等式

$$-q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q, \quad p \geq q > 0$$

显示

$$\sqrt{d + 4a^2r_1^2\delta_4} = \sqrt{d} + E$$

其中

$$|E| \leq \sqrt{4a^2r_1^2|\delta_4|}$$

因此 $\sqrt{d}/2a$ 中的绝对误差大约为 $r_1\sqrt{\delta_4}$。因为 $\delta_4 \approx \beta^{-p}$，所以 $\sqrt{\delta_4} \approx \beta^{-p/2}$，这样 $r_1\sqrt{\delta_4}$ 的绝对误差就破坏了根 $r_1 \approx r_2$ 的下半部分数位。换句话说，由于根的计算涉及计算 $(\sqrt{d})/(2a)$，并且此表达式在对应于 r_i 的一半低位的位置中没有有效位，所以 r_i 的低位不是有效的。▮

最后，我们转到定理 6 的证明。它基于以下事实（将在第 51 页的“定理 14 和定理 8”节中证明）。

D.6.1.13 定理 14

假定 $0 < k < p$，设 $m = \beta^k + 1$，并假定浮点运算是精确舍入的。那么，$(m \otimes x) \ominus (m \otimes x \ominus x)$ 与舍入到 $p - k$ 有效位的 x 完全相等。更精确的说法是，舍入 x 的方法是取 x 的有效位，可以设想为小数点正好在 k 个最低有效位的左侧并舍入为一个整数。

D.6.1.14 定理 6 的证明

依据定理 14，x_h 是舍入到 $p - k = \lfloor p/2 \rfloor$ 位的 x。如果没有进位，则 x_h 当然可以用 $\lfloor p/2 \rfloor$ 个有效位表示。假定存在进位。如果 $x = x_0.x_1 \ldots x_{p-1} \times \beta^e$，则舍入将 x_{p-k-1} 加 1。可能有进位的唯一情况是 $x_{p-k-1} = \beta - 1$，但此时 x_h 的低位数字是 $1 + x_{p-k-1} = 0$，因此 x_h 同样可用 $\lfloor p/2 \rfloor$。

要处理 x_1，请按比例增减 x 使其成为满足 $\beta^{p-1} \le x \le \beta^p - 1$ 的整数。假设 $x = \bar{x}_h + \bar{x}_l$，其中 $\bar{x}_h$ 是 x 的 $p - k$ 个高位数字，$\bar{x}_l$ 是 k 个低位数字。有三种情况需要考虑。如果 $\bar{x}_l < (\beta/2)\beta^{k-1}$，则将 x 舍入到 $p - k$ 位的结果与截断相同，而且 $x_h = \bar{x}_h$ 和 $x_l = \bar{x}_l$。因为 $\bar{x}_l$ 最多有 k 位，如果 β 是偶数，则 $\bar{x}_l$ 的位数最多为 $k = \lceil p/2 \rceil = \lfloor p/2 \rfloor$。否则，$\beta = 2$ 且 $\bar{x}_1 < 2^{k-1}$ 可以使用 $k - 1 \le \lfloor p/2 \rfloor$ 个有效位表示。第二种情况是在 $\bar{x} > (\beta/2)\beta^{k-1}$ 时，计算 x_h 涉及上舍入，因此 $x_h = \bar{x}_h + \beta^k$，且 $x_l = x - x_h = x - \bar{x}_h - \beta^k = \bar{x}_l - \beta^k$。同样，$\bar{x}_l$ 最多有 k 位，因此可以使用 $\lfloor p/2 \rfloor$ 位表示。最后，如果 $\bar{x}_l = (\beta/2)\beta^{k-1}$，则 $x_h = \bar{x}_h$ 或 $\bar{x}_h + \beta^k$，这取决于是否存在上舍入。因此 x_l 是 $(\beta/2)\beta^{k-1}$ 或 $(\beta/2)\beta^{k-1} - \beta^k = -\beta^k/2$，二者都是用 1 位表示的。▮

定理 6 提供了一种将两个工作精度数的乘积精确表示为一个和的方式。有一个用于精确表示和的相应公式。如果 $|x| \ge |y|$，则 $x + y = (x \oplus y) + (x \ominus (x \oplus y)) \oplus y$ [Dekker 1971；Knuth 1981，4.2.2 节中的定理 C]。但是，在使用精确舍入运算时，此公式仅对 $\beta = 2$ 是正确的，而对于 $\beta = 10$ 是不正确的，如示例 $x = .99998$、$y = .99997$ 所示。

D.6.2 二进制到十进制的转换

由于单精度有 $p = 24$，且 $2^{24} < 10^8$，所以您可能会认为将二进制数转换为 8 个十进制位将足以恢复原始二进制数。但是，这是不正确的。

D.6.2.1 定理 15

当将二进制 IEEE 单精度数转换为最接近的八位十进制数时，从十进制数唯一地恢复二进制数并不总是可能的。但是，如果使用九个十进制位，则将十进制数转换为最接近的二进制数将恢复原始浮点数。

D.6.2.2 证明

半开区间 $[10^3, 2^{10}) = [1000, 1024)$ 中的二进制单精度数有 10 位在二进制小数点的左侧，有 14 位在二进制小数点的右侧。因此，在该区间中有 $(2^{10} - 10^3)2^{14} = 393{,}216$ 个不同的二进制数。如果十进制数是使用 8 位表示的，则在同一区间中有 $(2^{10} - 10^3)10^4 = 240{,}000$ 个十进制数。240,000 个十进制数无法表示 393,216 个不同的二进制数。因此，8 个十进制位不足以唯一表示每个单精度二进制数。

要说明 9 位是足够的，只需说明二进制数之间的间隔始终大于十进制数之间的间隔就足够了。这将确保对于每个十进制数 N，区间

$$[N - \tfrac{1}{2}\text{ulp},\ N + \tfrac{1}{2}\text{ulp}]$$

最多包含一个二进制数。这样，每个二进制数都舍入为唯一的十进制数，而十进制数又舍入为唯一的二进制数。

以区间 $[10^n, 10^{n+1}]$ 为例来说明二进制数之间的间隔始终大于十进制数之间的间隔。在此区间上，连续二进制数之间的间隔是 $10^{(n+1)-9}$。在 $[10^n, 2^m]$ 上（其中 m 是满足 $10^n < 2^m$ 的最小整数），二进制数的间隔是 2^{m-24}，并且在该区间中此间隔变得越来越大。因此，只需证明 $10^{(n+1)-9} < 2^{m-24}$。而事实上，由于 $10^n < 2^m$，所以 $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^m 2^{-24}$。 ▮

应用于双精度的同一参数显示恢复双精度数需要 17 个十进制位。

二进制 - 十进制转换还提供了使用标志的另一示例。回想一下第 15 页的“精度”节中的内容：要从十进制扩展恢复二进制数，必须精确计算十进制到二进制的转换。该转换的方法是按单扩展精度将量 N 和 $10^{|P|}$（如果 $p < 13$，则二者都是精确的）相乘，然后舍入到单精度（如果 $p < 0$，则将它们相除；这两种情况是类似的）。当然，$N \cdot 10^{|P|}$ 的计算结果不可能是精确的；必须保持精确的是组合运算 round($N \cdot 10^{|P|}$)，其中舍入是从单扩展精度到单精度。要了解它为什么可能不是精确的，请举一个简单的例子：$\beta = 10$，$p = 2$（对于单精度）或 $p = 3$（对于单精度扩展）。如果乘积是 12.51，则这将作为单精度扩展乘法运算的一部分舍入为 12.5。舍入到单精度的结果为 12。但是该结果是不正确的，因为将乘积舍入到单精度的结果应该是 13。误差是双舍入导致的。

通过使用 IEEE 标志，可以避免双舍入，如下所示。保存不精确标志的当前值，然后将其重置。将舍入模式设置为“舍入为零”。然后，执行乘法运算 $N \cdot 10^{|P|}$。将不精确标志的新值存储在 `ixflag` 中，并恢复舍入模式和不精确标志。如果 `ixflag` 是 0，则 $N \cdot 10^{|P|}$ 是精确的，因此 round($N \cdot 10^{|P|}$) 直到最后一位都是正确的。如果 `ixflag` 是 1，则某些数字被截去了，因为舍入为零时总是截去数字。乘积的有效位类似于 $1.b_1 \ldots b_{22} b_{23} \ldots b_{31}$。如果 $b_{23} \ldots b_{31} = 10 \ldots 0$，则可能出现双舍入误差。解决这两种情况的简单方法是执行 `ixflag` 和 b_{31} 的逻辑 `OR` 运算。这样，在所有情况下 round($N \cdot 10^{|P|}$) 都能正确计算。

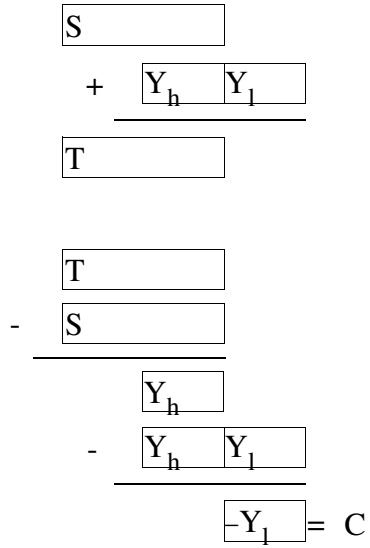
D.6.3 求和中的误差

第 34 页的“优化器”节提到了精确计算非常长的和的问题。改进精度的最简单方法是使精度加倍。为大致估计出精度加倍时和的精度的改进程度，假设 $s_1 = x_1$, $s_2 = s_1 \oplus x_2 \ldots$, $s_i = s_{i-1} \oplus x_i$。由此可得 $s_i = (1 + \delta_i)(s_{i-1} + x_i)$，其中 $|\delta_i| \le \varepsilon$。忽略 δ_i 中的二次项将得出

$$s_n = \sum_{j=1}^{n} x_j \left(1 + \sum_{k=j}^{n} \delta_k\right) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j \left(\sum_{k=j}^{n} \delta_k\right) \tag{31}$$

(31) 的第一个等式显示 Σx_j 的计算值与对 x_i 的扰动值执行精确求和时的结果是相同的。第一项 x_1 的扰动范围是 $n\varepsilon$，最后一项 x_n 的扰动范围是 ε。(31) 中的第二个等式显示误差项的界限是 $n\varepsilon\Sigma|x_j|$。使精度加倍具有对 ε 取平方的效果。如果求和是以 IEEE 双精度格式执行的，那么 $1/\varepsilon \approx 10^{16}$，因此对于 n 的任何合理值，$n\varepsilon \ll 1$。这样，使精度加倍采用了最大扰动 $n\varepsilon$，并将其更改为 $n\varepsilon^2 \ll \varepsilon$。因此，Kahan 求和公式（定理 8）的 2ε 误差界限不如使用双精度好，但比使用单精度要好得多。

有关 Kahan 求和公式为何成立的直观解释，请参考下面的过程图。



每次增加被加数时，都有一个校正因子 *C*，该因子将在下一循环中得以应用。因此，首先从 X_j 中减去在上一循环中计算的校正因子 *C*，得出校正后的被加数 *Y*。然后，将此被加数增加到连续和 *S* 中。在和中， *Y* 的低位 （即 Y_l）已丢失。接下来，通过计算 *T - S* 来计算 *Y* 的高位。当从中减去 *Y* 时，*Y* 的低位将恢复。这些是图中第一次求和时丢失的位。它们将成为下一循环的校正因子。定理 8 的正式证明包括在第 51 页的 “定理 14 和定理 8”节中 （摘自 Knuth [1981] 第 572 页）。

D.7 小结

计算机系统设计人员往往忽略系统中与浮点有关的部分。这很可能是由于计算机科学课程中很少注意浮点问题。而这又导致一种流行的看法：浮点不是一个可量化的主题，因此详细讨论处理浮点的硬件和软件没有多大意义。

本文证明了进行有关浮点的严谨推理是可能的。例如，可以证明如果基础硬件具有保护数位，则涉及抵消的浮点算法具有较小的相对误差；再如，存在一种用于二进制 - 十进制转换的高效算法，可以证明在支持扩展精度的条件下它是可逆的。当作为基础的计算机系统支持浮点时，执行构造可靠浮点软件的任务将容易得多。除了刚才提到的两个示例 （保护数位和扩展精度）外，本文的第 28 页的 “系统方面”节还包含许多说明如何更好地支持浮点的示例，内容涉及从指令集设计到编译器优化等各方面。

随着 IEEE 浮点标准得到越来越广泛的应用，利用该标准的各种特性的代码也将变得更具可移植性。第 14 页的 “IEEE 标准”节提供了很多示例，说明在编写实际的浮点代码时如何使用 IEEE 标准的各种特性。

D.8 致谢

本文受到 W. Kahan 于 1988 年 5 月至 7 月在 Sun Microsystems 讲授的课程的启发，该课程由 Sun 的 David Hough 精心组织。我希望大家可以籍此了解浮点与计算机系统之间的相互作用，而不必起床去听上午 8 点的课。在这里，我要感谢 Kahan 和 Xerox PARC 的许多同事（尤其是 John Gilbert），他们阅读了本文的草稿并提出了许多有用的意见。另外，此讲稿的改进还要归功于 Paul Hilfinger 和一位不知名人士的审阅。

D.9 参考书目

Aho, Alfred V., Sethi, R., and Ullman J. D. 1986. 《Compilers:Principles, Techniques and Tools》, Addison-Wesley, Reading, MA.

ANSI 1978. 《American National Standard Programming Language FORTRAN》, ANSI Standard X3.9-1978, American National Standards Institute, New York, NY.

Barnett, David 1987. 《A Portable Floating-Point Environment》, unpublished manuscript.

Brown, W. S. 1981. 《A Simple but Realistic Model of Floating-Point Computation》, ACM Trans. on Math.Software 7(4), pp. 445-480.

Cody, W. J et. al. 1984. 《A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic》, IEEE Micro 4(4), pp. 86-100.

Cody, W. J. 1988. 《Floating-Point Standards - Theory and Practice》, in "Reliability in Computing:the role of interval methods in scientific computing", ed. by Ramon E. Moore, pp.99-107, Academic Press, Boston, MA.

Coonen, Jerome 1984. 《Contributions to a Proposed Standard for Binary Floating-Point Arithmetic》, PhD Thesis, Univ. of California, Berkeley.

Dekker, T. J. 1971.《A Floating-Point Technique for Extending the Available Precision》, Numer. Math. 18(3), pp. 224-242.

Demmel, James 1984. 《Underflow and the Reliability of Numerical Software》, SIAM J. Sci. Stat. Comput. 5(4), pp. 887-919.

Farnum, Charles 1988. 《Compiler Support for Floating-point Computation》, Software-Practice and Experience, 18(7), pp. 701-709.

Forsythe, G. E. and Moler, C. B. 1967. 《Computer Solution of Linear Algebraic Systems》, Prentice-Hall, Englewood Cliffs, NJ.

Goldberg, I. Bennett 1967.《27 Bits Are Not Enough for 8-Digit Accuracy》, Comm. of the ACM.10(2), pp 105-106.

Goldberg, David 1990. 《Computer Arithmetic》, in "Computer Architecture:A Quantitative Approach", by David Patterson and John L. Hennessy, Appendix A, Morgan Kaufmann, Los Altos, CA.

Golub, Gene H. and Van Loan, Charles F. 1989.《Matrix Computations》, 2nd edition, The Johns Hopkins University Press, Baltimore Maryland.

Graham, Ronald L., Knuth, Donald E. and Patashnik, Oren.1989. 《*Concrete Mathematics*》, Addison-Wesley, Reading, MA, p.162.

Hewlett Packard 1982. HP-15C 《Advanced Functions Handbook》.

IEEE 1987. 《IEEE Standard 754-1985 for Binary Floating-point Arithmetic》, IEEE, (1985).Reprinted in SIGPLAN 22(2) pp. 9-25.

Kahan, W. 1972. 《A Survey Of Error Analysis》, in Information Processing 71, Vol 2, pp.1214 - 1239 (Ljubljana, Yugoslavia), North Holland, Amsterdam.

Kahan, W. 1986.《Calculating Area and Angle of a Needle-like Triangle》, unpublished manuscript.

Kahan, W. 1987. 《Branch Cuts for Complex Elementary Functions》, in "The State of the Art in Numerical Analysis", ed. by M.J.D. Powell and A. Iserles (Univ of Birmingham, England), Chapter 7, Oxford University Press, New York.

Kahan, W. 1988. Unpublished lectures given at Sun Microsystems, Mountain View, CA.

Kahan, W. and Coonen, Jerome T. 1982. 《The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments》, in "The Relationship Between Numerical Computation And Programming Languages", ed. by J. K. Reid, pp.103-115, North-Holland, Amsterdam.

Kahan, W. and LeBlanc, E. 1985. 《Anomalies in the IBM Acrith Package》, Proc.7th IEEE Symposium on Computer Arithmetic (Urbana, Illinois), pp. 322-331.

Kernighan, Brian W. and Ritchie, Dennis M. 1978. 《The C Programming Language》, Prentice-Hall, Englewood Cliffs, NJ.

Kirchner, R. and Kulisch, U. 1987. 《Arithmetic for Vector Processors》, Proc.8th IEEE Symposium on Computer Arithmetic (Como, Italy), pp. 256-269.

Knuth, Donald E., 1981. 《The Art of Computer Programming, Volume II》, Second Edition, Addison-Wesley, Reading, MA.

Kulisch, U. W., and Miranker, W. L. 1986. 《The Arithmetic of the Digital Computer:A New Approach》, SIAM Review 28(1), pp 1-36.

Matula, D. W. and Kornerup, P. 1985. 《Finite Precision Rational Arithmetic:Slash Number Systems》 , IEEE Trans. on Comput.C-34(1), pp 3-18.

Nelson, G. 1991. 《Systems Programming With Modula-3》, Prentice-Hall, Englewood Cliffs, NJ.

Reiser, John F. and Knuth, Donald E. 1975. 《Evading the Drift in Floating-point Addition》 , Information Processing Letters 3(3), pp 84-87.

Sterbenz, Pat H. 1974. 《Floating-Point Computation》 , Prentice-Hall, Englewood Cliffs, NJ.

Swartzlander, Earl E. and Alexopoulos, Aristides G. 1975. 《The Sign/Logarithm Number System》 , IEEE Trans. Comput. C-24(12), pp. 1238-1242.

Walther, J. S., 1971. 《A unified algorithm for elementary functions》 , Proceedings of the AFIP Spring Joint Computer Conf.38, pp. 379-385.

D.10 定理 14 和定理 8

此节包含正文中省略的两个技术性更强的证明。

D.10.1 定理 14

假定 $0 < k < p$，*设* $m = \beta^k + 1$，*并假定浮点运算是精确舍入的。那么，*$(m \otimes x) \ominus (m \otimes x \ominus x)$ *与舍入到* $p - k$ *个有效位的* x *完全相等。更精确的说法是，舍入* x *的方法是取* x *的有效位，可以设想为小数点正好在* k *个最低有效位的左侧并舍入为一个整数。*

D.10.2 证明

证明分为两种情况，这取决于 $mx = \beta^k x + x$ 的计算是否具有进位。

假定没有进位。按比例增减 x 使其成为一个整数是无害的。那么，$mx = x + \beta^k x$ 的计算类似于：

```
  aa...aabb...bb
  +aa...aabb...bb
  zz...zzbb...bb
```

其中 x 已经被分为两个部分。处于低位的 k 个数字被标记为 b，处于高位 $p - k$ 个数字被标记为 a。从 mx 计算 $m \otimes x$ 涉及舍弃处于低位的 k 个数字 （标记为 b 的数字），因此

$$m \otimes x = mx - x \bmod(\beta^k) + r\beta^k \tag{32}$$

如果 .bb...b 大于$\frac{1}{2}$，则 r 的值是 1，否则是 0。更准确地说，

如果 a.bb...b 舍入为 $a + 1$，则 $r = 1$，否则 $r = 0$。 (33)

接下来，计算 $m \otimes x - x = mx - x \bmod(\beta^k) + r\beta^k - x = \beta^k(x + r) - x \bmod(\beta^k)$。下图显示如何计算舍入的 $m \otimes x - x$，即 $(m \otimes x) \ominus x$。顶行是 $\beta^k(x + r)$，其中 B 是将 r 与最低位数字 b 相加而产生的数字。

```
aa...aabb...bB00...00
-bb...bb
zz...       zzZ00...00
```

如果 .bb...b < $\frac{1}{2}$，则 $r = 0$，减法运算导致从标记为 B 的数字借位，但是差值将进行上舍入，因此最终的结果是经过舍入的差等于顶行，即 $\beta^k x$。如果 .bb...b > $\frac{1}{2}$ 则 $r = 1$，由于借位而从 B 中减 1，因此结果是 $\beta^k x$。最后，以 .bb...b = $\frac{1}{2}$ 为例。如果 $r = 0$，则 B 是偶数，Z 是奇数，将差进行上舍入会得出 $\beta^k x$。类似地，当 $r = 1$ 时，B 是奇数，Z 是偶数，而差将进行下舍入，因此差同样是 $\beta^k x$。综上所述，

$$(m \otimes x) \ominus x = \beta^k x \tag{34}$$

结合等式 (32) 和 (34) 可以得出 $(m \otimes x) - (m \otimes x \ominus x) = x - x \bmod(\beta^k) + \rho \cdot \beta^k$。执行此计算的结果是

```
r00...00
         +  aa...aabb...bb
         -  .......bb...bb
            aa...aA00...00
```

计算 r 的规则 （即等式 (33)）与将 a... ab...b 舍入到 $p - k$ 位的规则是相同的。因此，在 $x + \beta^k x$ 没有进位的情况下，按浮点运算精度计算 $mx - (mx - x)$ 的结果与将 x 舍入到 $p - k$ 位完全相等。

当 $x + \beta^k x$ 确实有进位时， $mx = \beta^k x + x$ 类似于：

```
aa...aabb...bb
+aa...aabb...bb
zz...zZbb...bb
```

因此，$m \otimes x = mx - x \bmod(\beta^k) + w\beta^k$，其中 $w = -Z$（条件是 $Z < \beta/2$），但是 w 的精确值是不重要的。接下来， $m \otimes x - x = \beta^k x - x \bmod(\beta^k) + w\beta^k$。在图中

```
aa...aabb...bb00...00
-   bb...  bb
+   w
 zz     ...  zZbb  ...bb^31
```

经过舍入得出 $(m \otimes x) \ominus x = \beta^k x + w\beta^k - r\beta^k$，其中 $r = 1$（条件是 .bb...b > $\frac{1}{2}$ 或者 .bb...b = $\frac{1}{2}$ 且 $b_0 = 1$）。[32] 最后，

$$(m \otimes x) - (m \otimes x \ominus x) = mx - x \bmod (\beta^k) + w\beta^k - (\beta^k x + w\beta^k - r\beta^k)$$
$$= x - x \bmod (\beta^k) + r\beta^k。$$

同样，当 a...ab...b 舍入到 $p - k$ 位涉及上舍入时，$r = 1$ 完全成立。这样就在所有情况下，证明了定理 14。∎

D.10.2.1 定理 8（Kahan 求和公式）

假设 $\Sigma_{j=1}^{N} x_j$ 是使用以下算法计算的

```
S = X[1];
C = 0;
for j = 2 to N {
Y = X[j] - C;
   T = S + Y;
   C = (T - S) - Y;
   S = T;
}
```

那么，计算的和 S 等于 $S = \Sigma x_j (1 + \delta_j) + O(N\varepsilon^2) \Sigma |x_j|$，*其中* $|\delta_j| \le 2\varepsilon$。

D.10.2.2 证明

首先，回想一下简单公式 Σx_i 的误差是如何估算的。引入 $s_1 = x_1$，$s_i = (1 + \delta_i)(s_{i-1} + x_i)$。可知计算和为 s_n，它是各项的总和，其中的每一项都是 x_i 乘以包含 δ_j 的表达式。x_1 的精确系数是 $(1 + \delta_2)(1 + \delta_3)\ (1 + \delta_n)$，因此通过重新编号，$x_2$ 的系数一定是 $(1 + \delta_3)(1 + \delta_4) \ldots (1 + \delta n)$，依此类推。定理 8 的证明过程完全相同，只是 x_1 的系数更复杂。具体讲，就是 $s_0 = c_0 = 0$ 且

$$y_k = x_k \ominus c_{k-1} = (x_k - c_{k-1})(1 + \eta_k)$$
$$s_k = s_{k-1} \oplus y_k = (s_{k-1} + y_k)(1 + \sigma_k)$$
$$c_k = (s_k \ominus s_{k-1}) \ominus y_k = [(s_k - s_{k-1})(1 + \gamma_k) - y_k](1 + \delta_k)$$

其中的所有希腊字母的界限都是 ε。虽然 s_k 中 x_1 的系数是所需的最终表达式，但计算 $s_k - c_k$ 和 c_k 中 x_1 的系数更容易。

当 $k = 1$ 时，

31.如果加上 w 时不产生进位，那么这就是和。对于加上 w 产生进位的特殊情况，需要另外的参数。 – 编辑者

32.仅当 $(\beta^k x + w\beta^k)$ 保留 $\beta^k x$ 的形式时，经过舍入才得出 $\beta^k x + w\beta^k - r\beta^k$。 – 编辑者

$$c_1 = (s_1(1 + \gamma_1) - y_1)\ (1 + d_1)$$
$$= y_1((1 + s_1)\ (1 + \gamma_1) - 1)\ (1 + d_1)$$
$$= x_1(s_1 + \gamma_1 + s_1 g_1)\ (1 + d_1)\ (1 + h_1)$$
$$s_1 - c_1 = x_1[(1 + s_1) - (s_1 + g_1 + s_1 g_1)\ (1 + d_1)](1 + h_1)$$
$$= x_1[1 - g_1 - s_1 d_1 - s_1 g_1 - d_1 g_1 - s_1 g_1 d_1](1 + h_1)$$

分别调用这两个表达式 C_k 和 S_k 中 x_1 的系数，那么

$$C_1 = 2\varepsilon + O(\varepsilon^2)$$

$$S_1 = + \eta_1 - \gamma_1 + 4\varepsilon^2 + O(\varepsilon^3)$$

要得出 S_k 和 C_k 的通用公式，请展开 s_k 和 c_k 的定义，忽略所有涉及 x_i $(i > 1)$ 的项，得到

$$s_k = (s_{k-1} + y_k)(1 + \sigma_k)$$
$$= [s_{k-1} + (x_k - c_{k-1})\ (1 + \eta_k)](1 + \sigma_k)$$
$$= [(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}](1+\sigma_k)$$
$$c_k = [\{s_k - s_{k-1}\}(1 + \gamma_k) - y_k](1 + \delta_k)$$
$$= [\{((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1 + \sigma_k) - s_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k)$$
$$= [\{(s_{k-1} - c_{k-1})\sigma_k - \eta_k c_{k-1}(1 + \sigma_k) - c_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k)$$
$$= [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))](1 + \delta_k),$$
$$s_k - c_k = ((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})\ (1 + \sigma_k)$$
$$- [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k)](1 + \delta_k)$$
$$= (s_{k-1} - c_{k-1})((1 + \sigma_k) - \sigma_k(1 + \gamma_k)(1 + \delta_k))$$
$$+ c_{k-1}(-\eta_k(1 + \sigma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\ (1 + \delta_k))$$
$$= (s_{-1} - c_{k-1})\ (1 - \sigma_k(\gamma_k + \delta_k + \gamma_k\delta_k))$$
$$+ c_{k-1} - [\eta_k + \gamma_k + \eta_k(\gamma_k + \sigma_k\gamma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\delta_k]$$

由于 S_k 和 C_k 最多仅计算到次数 ε^2，因此这些公式可以简化为

$$C_k = (\sigma_k + O(\varepsilon^2))S_{k-1} + (-\gamma_k + O(\varepsilon^2))C_{k-1}$$
$$S_k = ((1 + 2\varepsilon^2 + O(\varepsilon^3))S_{k-1} + (2\varepsilon + O(\varepsilon^2))C_{k-1}$$

使用这些公式可以得出

$$C_2 = \sigma_2 + O(\varepsilon^2)$$

$$S_2 = 1 + \eta_1 - \gamma_1 + 10\varepsilon^2 + O(\varepsilon^3)$$

一般情况下通过归纳可以很容易地得到

$$C_k = \sigma_k + O(\varepsilon^2)$$

$$S_k = 1 + \eta_1 - \gamma_1 + (4_k+2)\varepsilon^2 + O(\varepsilon^3)$$

最后，所需的是 s_k 中 x_1 的系数。要获得此值，令 $x_{n+1} = 0$，假设下标为 $n + 1$ 的所有希腊字母都等于 0，计算 s_{n+1}。可知，$s_{n+1} = s_n - c_n$，且 s_n 中 x_1 的系数小于它在 s_{n+1} 中的系数，后者即 $S_n = 1 + \eta_1 - \gamma_1 + (4n + 2)\varepsilon^2 = (1 + 2\varepsilon + O(n\varepsilon^2))$。 ∎

D.11 各种 IEEE 754 实现的差别

注 – 此节不是已发表的论文的一部分。增加此节是为了澄清某些观点，并纠正读者可能通过论文推知的某些有关 IEEE 标准的误解。此材料不是由 David Goldberg 编写的，但却是经过了他的许可才在此处公布。

前面的论文已经表明：执行浮点运算时必须小心谨慎，因为编程人员可能要依靠其属性来实现程序的正确性和准确性。特别是在实现 IEEE 标准时需小心谨慎，只有在符合标准的系统上才能编写出能够正常工作并给出准确结果的有用程序。读者可能会据此得出结论：这样的程序应该可以移植到所有 IEEE 系统上。事实上，如果 “当一个程序在两台支持 IEEE 运算的计算机之间迁移时，如果任何中间结果是不同的，则一定是由于软件错误，而不是算法差异。”这一条件成立，那么编写可移植软件会变得更容易。

遗憾的是，IEEE 标准并不保证同一程序在所有符合该标准的系统上都将提供完全相同的结果。实际上，由于种种原因，大多数程序都会在不同的系统上产生不同的结果。其中一个原因是，大多数程序都涉及十进制格式和二进制格式之间的数字转换，而 IEEE 标准没有完全指定执行这样的转换必须使用的准确度。另一个原因是，许多程序使用由系统库提供的初等函数，而该标准并没有详细说明这些函数。当然，大多数编程人员都知道这些功能已经超出了 IEEE 标准的范围。

许多编程人员可能没有意识到，甚至是仅使用 IEEE 标准规定的数字格式和操作数的程序也可能在不同的系统上计算出不同的结果。实际上，该标准制定者的初衷就是允许不同的实现获得不同的结果。在 IEEE 754 标准中术语**目标**的定义里清晰地表达了这个意思：“目标可以被用户显式指定，也可以由系统隐式提供（例如，子表达式或过程参数中的中间结果）。某些语言会将中间计算的结果放在用户无法控制的目标中。但是，此标准根据该目标的格式和操作数的值定义运算的结果。”（IEEE 754-1985，第 7 页）换句话说，IEEE 标准要求将每个结果都正确舍入到将放置它的目标的精度，但是标准不要求由用户程序确定的该目标精度。因此，不同的系统可能将其结果提供给不同精度的目标，使同一程序产生不同的结果（有时差异很大），即使那些系统都符合标准亦如此。

前面论文中的几个实例需要有关舍入浮点运算方式的某些知识。为了灵活使用诸如这些例子的实例，编程人员必须能够预知将如何解释程序，尤其是，在 IEEE 系统上，每个算术运算的目标的精度可能是什么。但是，IEEE 标准中*目标*定义的漏洞削弱了编程人员知道将如何解释程序的能力。因此，在高级语言中作为明显可移植程序实现时，上面给出的几个实例可能无法在 IEEE 系统上正常工作，通常将结果提供给其精度与编程人员所预期不同的目标。其他实例也可能正常工作，但是证明它们是否正常工作可能超出了一般编程人员的能力。

在此节中，我们根据 IEEE 754 运算的现有实现通常使用的目标格式的精度对它们进行分类。然后，回顾论文中的一些实例，来说明以比程序预期精度更宽的精度提供结果会导致它计算出错误的结果，即使在使用期望的精度时它被证明是正确的。我们还可以再查看论文中其中一个证明，以阐明处理未预料的精度所需的脑力工作，即使该精度还没有使程序无效。这些实例说明，IEEE 标准允许在不同实现之间存在差异会阻止我们编写出可以准确预知其行为的可移植、高效数值软件，而与它规定的所有内容无关。要开发这样的软件，则首先必须创建限制 IEEE 标准允许的可变性的程序设计语言和环境，并允许编程人员表示其程序所依赖的浮点语义。

D.11.1 当前的 IEEE 754 实现

IEEE 754 运算的当前实现可以分为两组，它们是按支持硬件中不同浮点格式的程度区分的。*基于扩展的*系统，如 Intel x86 系列处理器，完全支持扩展的双精度格式，但是仅部分支持单精度和双精度：它们提供按单精度和双精度装入或存储数据的指令，飞速地将数据在单精度和双精度与扩展双精度格式之间来回转换，它们还提供特殊模式（不是默认模式），在这样的模式下按单精度或双精度舍入算术运算的结果，即使这些结果以扩展双精度格式保存在寄存器中。（Motorola 68000 系列处理器在这些模式下按单精度或双精度格式的精度和范围舍入结果。 Intel x86 及兼容处理器按单精度或双精度格式的精度舍入结果，但保留与扩展双精度格式相同的范围。）*单精度 / 双精度*系统，包括大多数 RISC 处理器，完全支持单精度格式和双精度格式，但不支持符合 IEEE 的扩展双精度格式。（IBM POWER 架构仅部分支持单精度，但出于本节组织的目的，我们将其归入单精度 / 双精度系统。）

要查看计算的行为在基于扩展的系统上与在单精度 / 双精度系统上有何不同，请参考第 28 页的 “系统方面”中实例的 C 版本：

```
int main() {
    double  q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

在这里，将常数 3.0 和 7.0 解释为双精度浮点数，而且表达式 3.0/7.0 继承双精度数据类型。在单精度 / 精度双系统上，将按双精度计算表达式的值，因为那是可以使用的最有效的格式。因此，将按双精度精确舍入 3.0/7.0 的值赋予 q。在下一行中，将再次按双精度计算表达式 3.0/7.0 的值，当然结果将等于刚赋给 q 的值，于是程序将像期望的那样打印 “Equal”。

在基于扩展的系统上，即使表达式 3.0/7.0 的类型为双精度，也将以扩展双精度格式在寄存器中计算商，因此在默认模式下，将按扩展双精度舍入它。但是，当将计算的值赋予变量 q 时，之后可能将它存储在内存中，因为 q 被声明为双精度，所以将按双精度舍入该值。在下一行中，可能再次按扩展精度计算表达式 3.0/7.0 的值，产生的结果与存储在 q 中的双精度值不同，使程序打印 “Not equal”。当然，其他结果也是可能的：编译器可以确定在将表达式 3.0/7.0 的值与 q 进行比较之前在第二行中存储该值并舍入它，或者可以按扩展精度将 q 保存在寄存器中而不存储它。优化的编译器可能在编译时 （也许是按双精度，也许是按扩展双精度）计算表达式 3.0/7.0 的值。（使用同一个 x86 编译器，在为优化进行编译时程序将打印 “Equal”，在为调试进行编译时将打印 “Not Equal”。）最后，基于扩展的系统的某些编译器自动更改舍入精度模式，以使在寄存器中产生结果的运算按单精度或双精度舍入那些结果，尽管使用更宽的范围是可能的。这样，在这些系统上，我们无法仅通过读取其源代码并运用 IEEE 754 运算的基本知识来预知程序的行为。我们也不能将未能提供符合 IEEE 754 的环境归因于硬件或编译器；硬件已经将正确舍入的结果提供给每个目标（按要求它做的那样），而且编译器已经将某些中间结果赋予超出用户控制的目标 （按允许它做的那样）。

D.11.2 在基于扩展的系统上计算的缺陷

按一般的思维，基于扩展的系统必须产生至少是精确的结果，即便不比在单精度 / 双精度系统上提供的更精确，因为前者始终提供尽可能高的精度，而且通常比后者高。通常的实例 （如上面的 C 程序）以及基于下面讨论的实例的更细致程序说明这种认识至少有一些天真：一些明显可移植的程序确实是可以跨单精度 / 双精度系统移植的，在基于扩展的系统上会提供错误的结果，完全是由于编译器和硬件协同工作，提供的精度有时候比程序所需的高。

当前的程序设计语言使程序很难指定它所需的精度。如第 30 页的“语言和编译器”上的“语言和编译器”节所述，许多程序设计语言不指定在同一上下文中某个表达式（如 `10.0*x`）每次运行应该计算出相同的值。在这方面，某些语言（如 Ada）就受到了 IEEE 标准之前的不同运算之间差异的影响。最近，类似 ANSI C 的语言受到了符合标准的、基于扩展的系统的影响。事实上，ANSI C 标准明确允许编译器按比通常与其类型关联的精度更宽的精度计算浮点表达式的值。因此，表达式 `10.0*x` 的值可能以取决于以下各种因素的方式变化：表达式是立即赋予变量还是作为子表达式出现在更大的表达式中；表达式是否参与比较；表达式是否作为参数传递给函数，如果是这样，该参数是按值还是按引用传递；当前的精度模式；编译程序时的优化级别；在编译程序时编译器所使用的精度模式和表达式计算方法；等等。

不能将表达式计算不完全一致全部归因于语言标准。可能的时候，在扩展精度寄存器中计算表达式，基于扩展的系统的运行效率最高，但是必须存储的值是按所需的最窄精度存储的。约束一种语言使其要求 `10.0*x` 在任何位置都计算为同一值将降低那些系统的性能。可惜的是，允许那些系统在语义上等价的上下文中将 `10.0*x` 计算为不同的值会对开发精确数值软件的编程人员产生它自己的负面影响，它阻止编程人员依赖其程序语法来表示预定语义。

真正的程序是否依赖给定表达式始终计算为同一值的假设？回想在定理 4 中提供的用于计算 ln(1 + *x*) 的算法，在此处用 Fortran 编写：

```
real function log1p(x)
real x
if (1.0 + x .eq.1.0) then
   log1p = x
else
   log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

在基于扩展的系统上，编译器可能在第三行中按扩展精度计算表达式 `1.0 + x` 的值并将结果与 `1.0` 进行比较。但是，将同一表达式传递给第六行中的 log 函数时，编译器可能将其值存储在内存中，并按单精度对其进行舍入。这样，如果对于按扩展精度不能将 `1.0 + x` 舍入为 `1.0`，`x` 不够小，但对于按单精度可以将 `1.0 + x` 舍入为 `1.0`，对于单精度已经足够小，那么 `log1p(x)` 返回的值将是零，而不是 `x`，相对误差将是一 -远远大于 5ε。同样，假定第六行中表达式的其余部分（包括子表达式 `1.0 + x` 的重新出现）是按扩展精度计算的。在这种情况下，如果 `x` 较小但没有小到足以按单精度将 `1.0 + x` 舍入为 `1.0`，则 `log1p(x)` 返回的值可以超过正确值接近 `x`，同样相对误差可以接近一。下面给出一个具体的实例，假设 `x` 等于 $2^{-24} + 2^{-47}$，因此 `x` 是最小的单精度数，所以 `1.0 + x` 上舍入到下一个较大的数 $1 + 2^{-23}$。那么，`log(1.0 + x)` 大约等于 2^{-23}。因为第六行中的表达式的分母是按扩展精度计算的，所以它是精确计算的并提供 `x`，这样 `log1p(x)` 返回的值大约等于 2^{-23}，该值几乎是精确值的两倍。（至少一种编译器实际出现过这种情况。当用于 x86 系统的 Sun WorkShop Compilers 4.2.1 Fortran 77 编译器使用 `-O` 优化标志编译前面的代码时，生成的代码完全像说明的那样计算 `1.0 + x`。因此，函数为 `log1p(1.0e-10)` 提供零，为 `log1p(5.97e-8)` 提供 `1.19209E-07`。）

为使定理 4 的算法正常工作，在表达式 `1.0 + x`每次出现时，必须以相同的方式对其进行计算；在基于扩展的系统上，只有当 `1.0 + x` 的计算结果在一个实例中为扩展双精度表示，在另一个实例中为单精度或双精度表示时，该算法才可能无法正常工作。当然，由于 `log` 是 Fortran 中的一个通用内函数，编译器可能从始至终采用扩展精度来计算表达式 `1.0 + x`的值，以相同的精度计算它的对数，但事实上我们不能假设编辑器将这样做。（还可以设想一个涉及用户定义函数的类似示例。在该情况下，即使用户定义的函数返回的是单精度结果，编译器可能仍以扩展精度保存参数，但是即使任何现有的 Fortran 编译器这样做，也很少。）因此，我们可以尝试确保通过将 `1.0 + x` 指定给某个变量，以便统一计算结果。遗憾的是，如果我们声明变量 `real`，由于编译器替换该变量一种外观表示以扩展精度保存在寄存器中的值，以及该变量另一种外观表示以单精度存储在内存中的值，我们的想法依然无法实现。相反，我们需要用与扩展精度格式对应的某种类型声明该变量。标准 FORTRAN 77 不支持这样做，而 Fortran 95 则提供了 `SELECTED_REAL_KIND` 机制来描述各种格式，它并不明确要求以扩展精度计算表达式结果的实现，以允许采用扩展精度声明变量。简而言之，在标准 Fortran 中没有便捷的方式来编写该程序，以保证始终以我们预想的方式计算表达式 `1.0 + x` 的值。

在基于扩展的系统上，有其他一些示例，即使每个子表达式都已存储并因此用相同的精度舍入，还是不能正常工作。原因就是*双舍入*。在默认精度模式下，基于扩展的系统最初会将每个结果舍入为扩展双精度。如果再将该结果以双精度存储，会对它再次进行舍入操作。这两次舍入的组合可能使生成的值与通过将第一次的结果正确舍入为双精度所得的结果不同。当舍入为扩展双精度是结果是“中间数”时，即，该结果恰好在两个双精度数的中间时，可能出现上述情况，所以第二次舍入由舍入为偶数规则决定。如果第二次舍入和第一次舍入都是向上舍入或向下舍入，那么净舍入误差将超过最后一位中的一半单位。（注意，尽管双舍入只影响双精度计算。但还是可以证明只要 $q \geq 2p + 2$，两个 p- 位数的加、减、乘、除，或者一个 p 位数的平方根运算，第一次舍入为 q 位然后再舍入为 p 位与只舍入一次到 p 位的结果相同。因此，扩展的双精度的宽度已足够，单精度计算无需进行双舍入。）

依赖正确舍入的一些算法会因双舍入而失败。事实上，甚至是不要求正确舍入且在不符合 IEEE 754 的各种计算机上正确工作的一些算法也会因双舍入而失败。其中最有用的算法是用于执行在第 11 页的“定理 5”节中提到的模拟多精度运算的可移植算法。例如，定理 6 中所述的用于将浮点数拆分为高位和低位部分的过程在双舍入运算中不能正确工作：试图将双精度数 $2^{52} + 3 \times 2^{26} - 1$ 拆分为两部分，各部分最多为 26 位。按双精度正确舍入每个运算时，高位部分是 $2^{52} + 2^{27}$，低位部分是 $2^{26} - 1$，但先按扩展双精度舍入每个运算再按双精度舍入时，该过程产生的高位部分是 $2^{52} + 2^{28}$，低位部分是 $-2^{26} - 1$。后一个数占用 27 位，因此无法按双精度精确计算其平方。当然，仍有可能按扩展双精度计算此数的平方，但是得出的对数将不再能够移植到单精度 / 双精度系统。此外，多精度乘法算法中的后续步骤假定按双精度计算了所有的部分乘积。正确处理双精度变量和扩展双精度变量的混合将大大提高实现的成本。

同样，用于将表示为双精度数的数组的多精度数相加的可移植算法会在双舍入运算中失败。这些算法通常依赖类似于 Kahan 求和公式的方法。正如第 47 页的“求和中的误差”节中给出的求和公式非正式解释所提出的，如果 `s` 和 `y` 是浮点变量并且 $|s| \geq |y|$，计算：

```
t = s + y;
e = (s - t) + y;
```

那么，在大多数运算中， e 精确恢复计算 t 时出现的舍入误差。但是，此方法在双舍入运算中不能正常工作：如果 s = $2^{52} + 1$ 且 y = $1/2 - 2^{-54}$，那么， s + y 先按扩展双精度舍入为 $2^{52} + 3/2$，然后依据“在中途情况下舍入为偶数”规则按双精度将此值舍入为 $2^{52} + 2$；这样，计算 t 时的最终舍入误差是 $1/2 + 2^{-54}$，它无法按双精度精确表示，因此它无法用上面所示的表达式精确计算。同样，通过按扩展双精度计算和来恢复舍入误差将是可能的，但是这样程序将必须进行额外的工作才能将最终输出约简回双精度，双舍入也会影响此过程。由于这一原因，虽然通过这些方法模拟多精度运算的可移植程序在各种各样的计算机上正确而高效地运行，但是它们在基于扩展的系统上并不像所声称的那样运行。

最后，起初看上去似乎依赖正确舍入的一些算法事实上可能与双舍入一起正确工作。在这些情况下，处理双舍入的成本不在于实现，而在于对算法是否像声称的那样工作进行的检验。为了说明这一点，我们证明定理 7 的以下变形：

D.11.2.1 定理 7′

如果 m 和 n 是可以按 *IEEE 754* 双精度表示的整数，$|m| < 2^{52}$，n 具有特殊形式 $n = 2^i + 2^j$，那么 $(m \oslash n) \otimes n = m$，条件是两个浮点运算都是按双精度正确舍入的或先按扩展双精度舍入再按双精度舍入的。

D.11.2.2 证明

假定 $m > 0$。那么 $q = m \oslash n$。将二的幂作为因子进行调整，我们可以考虑一个等价设置，其中 $2^{52} \le m < 2^{53}$，对于 q 是类似的，以便 m 和 q 都是这样的整数：其最低有效位占用单位位置（即 $\mathrm{ulp}(m) = \mathrm{ulp}(q) = 1$）。在调整之前，假定 $m < 2^{52}$，因此在调整之后， m 是一个偶数整数。此外，因为 m 和 q 的调整值满足 $m/2 < q < 2m$，所以 n 的对应值必须具有两种形式中的一种，具体取决于 m 和 q 哪个较大：如果 $q < m$，显然 $1 < n < 2$，因为 n 是两个二的幂的和，所以对于一些 k，$n = 1 + 2^{-k}$；同样，如果 $q > m$，那么 $1/2 < n < 1$，因此 $n = 1/2 + 2^{-(k+1)}$。（因为 n 是两个二的幂的和，所以 n 的最接近一的可能值是 $n = 1 + 2^{-52}$。因为 $m/(1 + 2^{-52})$ 不比小于 m 的下一个较小双精度数大，所以不能使 $q = m$。）

假设 e 表示计算 q 时的舍入误差，以便 $q = m/n + e$，计算值 $q \otimes n$ 将是 $m + ne$ 的（一次或两次）舍入值。请首先考虑按双精度正确舍入每个浮点运算的情况。在这种情况下，$|e| < 1/2$。如果 n 具有形式 $1/2 + 2^{-(k+1)}$，则 $ne = nq - m$ 是 $2^{-(k+1)}$ 的整数倍，且 $|ne| < 1/4 + 2^{-(k+2)}$。这意味着 $|ne| \le 1/4$。请回想一下 m 与下一个较大的可表示数之间的差是 1，m 和下一个较小的可表示数之间的差是 1（如果 $m > 2^{52}$）或 1/2（如果 $m = 2^{52}$）。这样，因为 $|ne| \le 1/4$，$m + ne$ 将舍入为 m。（即使 $m = 2^{52}$ 且 $ne = -1/4$，乘积也将按“在中途情况下舍入为偶数”规则舍入为 m。）同样，如果 n 具有形式 $1 + 2^{-k}$，则 ne 是 2^{-k} 的整数倍，且 $|ne| < 1/2 + 2^{-(k+1)}$；这隐含 $|ne| \le 1/2$。在这种情况下，不能使 $m = 2^{52}$，因为 m 严格大于 q，因此 m 与其最接近的可表示数相差 ±1。这样，因为 $|ne| \le 1/2$，$m + ne$ 同样将舍入为 m。（即使 $|ne| = 1/2$，乘积也将按“在中途情况下舍入为偶数”规则舍入为 m，因为 m 是偶数。）这样就完成了正确舍入运算的证明。

在双舍入运算中，仍可能发生以下情况：q 是正确舍入的商（即使它实际上舍入了两次），因此像上面那样 $|e| < 1/2$。在这种情况下，我们可以求助于上一段中的论点，条件是考虑到 $q \otimes n$ 将被舍入两次的事实。为了解释这一点，请注意 IEEE 标准要求扩展双精度格式至少有 64 个有效位，以便数 $m \pm 1/2$ 和 $m \pm 1/4$ 可以按扩展双精度精确表示。这样，如果 n 具有形式 $1/2 + 2^{-(k+1)}$，以便 $|ne| \le 1/4$，则按扩展双精度舍入 $m + ne$ 必须产生与 m 最多相差 1/4 的结果，如上所述，此值将按双精度舍入为 m。同样，如果 n 具有形式 $1 + 2^{-k}$，以便 $|ne| \le 1/2$，则按扩展双精度舍入 $m + ne$ 必须产生与 m 最多相差 1/2 的结果，而且此结果将按双精度舍入为 m。（请回想，在这种情况下 $m > 2^{52}$。）

最后，要考虑具有以下特点的其余情况：因双舍入而导致 q 不是正确舍入的商。在这些情况下，最差具有 $|e| < 1/2 + 2^{-(d+1)}$，其中 d 是扩展双精度格式中的额外位数。（所有基于扩展的现有系统都支持正好具有 64 个有效位的扩展双精度格式；对于此格式，$d = 64 - 53 = 11$。）因为在第二次舍入由“在中途情况下舍入为偶数”规则确定时双舍入仅产生不正确的舍入结果，所以 q 必须是偶数整数。这样，如果 n 具有形式 $1/2 + 2^{-(k+1)}$，则 $ne = nq - m$ 是 2^{-k} 的整数倍，且

$$|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}。$$

如果 $k \le d$，则意味着 $|ne| \le 1/4$。如果 $k > d$，则我们具有 $|ne| \le 1/4 + 2^{-(d+2)}$。在任一情况下，乘积的第一次舍入都将提供与 m 最多相差 1/4 的结果，依据上述论点，第二次舍入将舍入为 m。同样，如果 n 具有形式 $1 + 2^{-k}$，则 ne 是 $2^{-(k-1)}$ 的整数倍，且

$$|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}。$$

如果 $k \le d$，则意味着 $|ne| \le 1/2$。如果 $k > d$，则我们具有 $|ne| \le 1/2 + 2^{-(d+1)}$。在任一情况下，乘积的第一次舍入都将提供与 m 最多相差 1/2 的结果，同样依据上述论点，第二次舍入将舍入为 m。▮

前面的证明表明：仅当商引起双舍入时，乘积才会引起双舍入，甚至在此时乘积也舍入为正确的结果。该证明还表明：即使是对于只有两个浮点运算的程序来说，扩展我们的推理以包括存在双舍入的可能也可以是有挑战性的。对于更复杂的程序，系统地解释双舍入的影响也许是不可能的，更不必说双精度计算和扩展双精度计算的更一般组合。

D.11.3 扩展精度的程序设计语言支持

不应该由前面的实例得出扩展精度*本身*是无益的这一结论。当编程人员能够有选择地使用扩展精度时，许多程序可以从扩展精度受益。遗憾的是，当前的程序设计语言没有提供足够的方式，供编程人员用来指定应该使用扩展精度的时间和方式。为了指出所需的支持，我们考虑可能希望管理扩展精度的使用的方式。

在将双精度用作其标称工作精度的可移植程序中，我们可能希望控制更宽精度的使用的方式有以下五种：

1. 在基于扩展的系统上尽可能使用扩展精度进行编译，以产生速度最快的代码。显然，大多数数值软件对运算的要求仅仅是每个运算中的相对误差以“计算机厄普西隆”为界限。当内存中的数据按双精度存储时，通常认为计算机厄普西隆是该精度中的最大相对舍入误差，因为（正确或错误地）假定在输入数据被输入时已经过舍入，在存储结果时将对其进行类似舍入。这样，尽管按扩展精度计算一些中间结果可能产生更精确的结果，但是扩展精度不是必需的。在这种情况下，我们可能更希望仅当扩展精度不会明显减慢程序速度时编译器才使用扩展精度，否则使用双精度。

2. 如果比双精度宽的一种格式相当快且足够宽，则使用该格式，否则使用其他格式。当扩展精度可用时，一些计算可以更容易地执行，它们也可以按双精度执行，但是会更加复杂。以计算双精度数向量的欧几里得范数为例。通过计算元素的平方并以 IEEE 754 扩展双精度格式（使用其较宽的指数范围）累加其和，可以为实用长度的向量很普通地避免过早下溢或上溢。在基于扩展的系统上，这是计算范数的最快方式。在单精度/双精度系统上，扩展双精度格式将必须在软件中仿真（如果支持一种格式的话），而且这样的仿真将比仅使用双精度慢得多，测试异常标志以确定是否发生下溢或上溢，如果是这样，使用显式调整重复该计算。请注意，要支持扩展精度的这一使用，语言必须同时提供相当快的最宽可用格式的指示（以便程序可以选择使用哪种方法）和指示每种格式的精度及范围的环境参数（以便程序可以检验最宽的快速格式是否足够宽，例如，检验它是否具有比双精度更宽的范围）。

3. 即使比双精度更宽的格式必须在软件中仿真，也要使用该格式。对于比欧几里得范数实例更复杂的程序，编程人员可能仅希望消除编写程序的两个版本的需要；相反，即使扩展精度的速度慢也依赖它。同样，语言必须提供环境参数，程序才能确定最宽可用格式的范围和精度。

4. 请勿使用更宽的精度；即使采用扩展的范围，也将结果正确舍入为双精度格式。对于需要依赖正确舍入的双精度运算轻松编写的程序（包括上面提到的部分示例），语言必须为编程人员提供指出不得使用扩展精度的方式，即使可以在寄存器中使用比双精度更宽的指数范围来计算中间结果也不能使用扩展精度。（以此方式计算出的中间结果在存储到内存时，如果出现下溢，仍可能执行双舍入：如果算术运算的结果第一次被舍入为 53 个有效位，然后在必须对该结果进行非规格化操作时，再次将它舍入位更小的有效位数，那么最终结果可能与只舍入一次，成为非规格化数所得的结果不同。当然，这种双舍入的形式对任何实际的程序产生不良影响的可能性非常低。）

5. 按双精度格式的精度和范围正确舍入结果。对于测试数值软件或接近双精度格式的范围及精度的极限的运算本身的程序，进行双精度严格强制将是极其有用的。以可移植方式编写这样细致的测试程序往往是很困难的；当它们必须利用伪子例程和其他技巧强制按特定格式舍入结果时，编写它们更加困难（且容易出错）。因此，使用基于扩展的系统开发必须可以移植到所有 IEEE 754 实现的可靠软件的编程人员，将很快发现可以仿真单精度 / 双精度系统的运算而无需特别的努力。

当前语言都不支持所有这五种选择。事实上，几乎没有语言尝试为编程人员提供控制扩展精度的使用的能力。一个值得注意的例外是 ISO/IEC 9899:1999 程序设计语言 - C 标准，它是 C 语言的最新修订，现在处于标准化的最终阶段。

C99 标准允许实现以比通常与其类型关联的格式更宽的格式计算表达式的值，但是 C99 标准建议使用仅仅三种表达式计算方法之一。建议使用的三种方法以将表达式“提升”到更宽格式的程度为特征，鼓励实现通过定义预处理程序宏 `FLT_EVAL_METHOD` 来识别它使用哪种方法：如果 `FLT_EVAL_METHOD` 是 0，则以对应于其类型的格式计算每个表达式；如果 `FLT_EVAL_METHOD` 是 1，则将浮点表达式提升到对应于双精度的格式；如果 `FLT_EVAL_METHOD` 是 2，则将浮点和双精度表达式提升到对应于长双精度的格式。（允许实现将 `FLT_EVAL_METHOD` 设置为 -1 以指示表达式计算方法是不能确定的。）C99 标准还要求 `<math.h>` 头文件定义类型 `float_t` 和 `double_t`，它们分别至少与浮点和双精度一样宽，旨在与用于计算浮点和双精度表达式的类型匹配。例如，如果 `FLT_EVAL_METHOD` 是 2，则 `float_t` 和 `double_t` 都是长双精度。最后，C99 标准要求 `<float.h>` 头文件定义指定对应于每种浮点类型的格式的范围及精度的预处理程序宏。

C99 标准要求或建议的功能组合支持上面列出的五种选择中的一些选择，但是不支持所有选择。例如，如果实现将长双精度类型映射到扩展双精度格式，并将 `FLT_EVAL_METHOD` 定义为 2，则编程人员可以合理地假定扩展精度的速度较快，因此类似欧几里得范数实例的程序只需使用长双精度（或 `double_t`）类型的中间变量即可。另一方面，同一实现必须按扩展精度保存匿名表达式，即使它们存储在内存中（例如，当编译器必须使浮点寄存器溢出时），并且它必须存储赋予声明为双精度的变量的表达式的结果，以便将其转换为双精度，即使它们可以保存在寄存器中。这样，双精度或 `double_t` 类型都不能进行编译以便在基于扩展的当前硬件上产生速度最快的代码。

同样，C99 标准提供了此部分中实例所说明的一些问题（但不是所有问题）的解决方法。如果将表达式 `1.0 + x` 赋予一个变量（为任何类型）且该变量是自始至终使用的，则可以保证 `log1p` 函数的 C99 标准版本正确工作。但是，用于将双精度数拆分为高位部分和低位部分的可移植的、高效的 C99 标准程序更为困难：在无法保证按双精度正确舍入双精度表达式的情况下，如何在正确的位置进行拆分并避免双舍入？一种解决方法是使用 `double_t` 类型在单精度 / 双精度系统上按双精度执行拆分，在基于扩展的系统上按扩展精度进行拆分，以便在任一情况下都将正确舍入运算。定理 14 告诉我们可以在任意位的位置进行拆分，条件是知道基础运算的精度，`FLT_EVAL_METHOD` 和环境参数宏应该提供此信息。

下面的程序片段显示一种可能的实现：

```
#include <math.h>
#include <float.h>

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif

...
    double   x, xh, xl;
    double_t m;

    m = scalbn(1.0, PWR2) + 1.0;  // 2**PWR2 + 1
    xh = (m * x) - ((m * x) - x);
    xl = x - xh;
```

当然，要求出此解，编程人员必须知道：双精度表达式可能是按扩展精度计算的；随之出现的双舍入问题可以导致算法出错；依据定理 14 可能改用扩展精度。更显而易见的解决方法是只需指定按双精度正确舍入每个表达式即可。在基于扩展的系统上，这只要求更改舍入精度模式，但可惜的是， C99 标准没有提供做到这一点的可移植方法。（Floating-Point C Edits （指定为支持浮点而对 C90 标准进行的更改的工作文档）的早期草案建议具有舍入精度模式的系统上的实现提供 `fegetprec` 和 `fesetprec` 函数以获取和设置舍入精度，这两个函数与获取和设置舍入方向的 `fegetround` 和 `fesetround` 函数类似。在对 C99 标准进行更改之前删除了此建议案。）

巧合的是， C99 标准支持具有不同整数运算功能的系统之间的可移植性的途径是建议了一种支持不同浮点架构的更好方法。每个 C99 标准实现都提供一个 `<stdint.h>` 头文件，该文件定义实现所支持的那些整数类型，它们按大小和效率命名：例如， `int32_t` 是 32 位宽的整数类型， `int_fast16_t` 是实现的速度最快且至少 16 位宽的整数类型， `intmax_t` 是支持的最宽整数类型。可以为浮点类型设想一个类似的方案：例如， `float53_t` 可用于命名具有正好 53 位精度但可能具有更宽范围的一种浮点类型， `float_fast24_t` 可用于命名实现的速度最快且具有至少 24 位精度的类型， `floatmax_t` 可用于命名支持的最宽且速度相当快的类型。快速类型可能允许基于扩展的系统上的编译器生成可能最快的代码，这仅受到以下约束的影响：指定变量的值不能因寄存器溢出而改变。精确宽度类型将导致基于扩展的系统上的编译器将舍入精度模式设置为按指定精度舍入，从而允许更宽的范围受到同一约束的影响。最后， `double_t` 可用于命名一种同时具有 IEEE 754 双精度格式的精度和范围的类型，条件是使用严格的双精度计算。与相应命名的环境参数宏一起，这样的方案就可以支持上述的所有五种选择，并允许编程人员轻松而明确地指出其程序所需的浮点语义。

扩展精度的语言支持必须这样复杂吗？在单精度 / 双精度系统上，上面列出的五种选择中的四种是一致的，因此不必区分快速和精确宽度类型。但是，基于扩展的系统会给选择带来困难：它们既不支持纯双精度计算的效率也不支持纯扩展精度计算的效率与这两种运算的混合一样高，而且不同的程序要求不同的混合程度。此外，选择何时使用扩展精度不应该留给编译器编写人员，他们通常受到测试基准的影响 （有时得到数值分析员的直接通知），将浮点运算视为“本身不精确”，因此不值得也不能够预知整数运算。相反，必须由编程人员进行选择，他们将需要能够表达其选择的语言。

D.11.4 结束语

上述评论并非旨在贬低基于扩展的系统，而是要揭示几个谬论，第一个谬论是所有 IEEE 754 系统都必须为同一程序提供完全相同的结果。我们着重说明了基于扩展的系统与单精度 / 双精度系统之间的不同，但是在其中每个系列内的系统之间存在进一步的不同。例如，一些单精度 / 双精度系统提供单个指令将两个数相乘并与第三个数相加，只进行一次最终舍入。此运算称为 *合并的乘 - 加*，会导致同一程序在不同的单精度 / 双精度系统上产生不同的结果；与扩展精度一样，它甚至会导致同一程序在同一系统上产生不同的结果，这取决于是否使用它和何时使用它。（合并的乘 - 加也可以阻止定理 6 的溢出过程，尽管能够以不可移植的方式使用它来执行多精度乘法运算，而不必进行拆分。）尽管 IEEE 标准没有预期到这样的操作，但是它与此一致：中间乘积被提供给超出用户控制的“目标”，它的宽度足以精确容纳该乘积，最终的和被正确舍入以适合其单精度或双精度目标。

但是，以下意见是吸引人的：IEEE 754 精确规定给定的程序必须提供的结果。许多编程人员喜欢相信他们可以理解程序的行为，并证明它将正确运行，而不涉及编译它的编译器或运行它的计算机。在许多方面，对于计算机系统和程序设计语言的设计人员来说，支持此意见是一个需要花费精力的目标。可惜的是，当涉及到浮点运算时，该目标几乎是不可能实现的。IEEE 标准的制定者知道他们不试图实现该目标。因此，尽管在整个计算机行业几乎都符合 IEEE 754 标准 （的大部分内容），但是开发可移植软件的编程人员必须继续处理不可预知的浮点计算。

如果编程人员利用 IEEE 754 的各种特性，则他们将需要使浮点运算可预知的程序设计语言。C99 标准在一定程度上改进了可预知性，代价是要求编程人员编写其程序的多个版本，对于每个 `FLT_EVAL_METHOD` 都编写一个版本。现在还不知道：将来的语言是否将改为选择允许编程人员使用明确表达程序对 IEEE 754 语义的依赖程度的语法，来编写单个程序。基于扩展的现有系统对该前景构成了威胁，因为它们使我们很容易假定编译器和硬件对应该如何在给定系统上执行计算的了解比编程人员更好。该假定是第二个谬论：计算的结果所需的准确度不取决于产生它的计算机，而是仅取决于从它得出的结论；在编程人员、编译器和硬件中，最多只有编程人员才能知道那些结论可能是什么。

附录 E

标准遵循性

Sun Studio 编译器产品与 Solaris 10 操作环境中的头文件及库支持多个标准，包括：System V Interface Definition Edition 3 (SVID)、X/Open、ANSI C (C90)、POSIX.1-2001 (SUSv3) 和 ISO C (C99)。（请参见 `standards`(5)，已获得完整说明。）这其中的一些标准允许实现在特定方面可以有所不同。在某些情况下，这些标准规范会发生冲突。对于数学库，变化和冲突主要与特例和异常相关。此附录给出了 `libm` 中的函数在此类情况下的行为，并讨论在何种条件下可期望 C 程序符合所有的标准。本附录的最后一节给出了 Sun Studio C 和 Fortran 语言产品对 LIA-1 规范的遵循情况。

E.1 `libm` 特例

表 E-1 列出了两个或多个前述标准指定 `libm` 中的函数行为时发生冲突的情况。C 程序将遵循何种行为取决于编译和链接程序时所用的编译器标志。可能的行为包括发生浮点异常、调用用户提供的函数 `matherr`，以及有关出现的特例的信息和要返回的值（请参见 `matherr`(3M)）、打印标准错误文件上的消息和设置全局变量 `errno`（请参见 `intro`(2) 和 `perror`(3C)）。

表 E-1 中的第一列定义特例。第二列给出如果设置了 `errno` 将要为其设置的值。`errno` 的可能值在 `<errno.h>` 中定义；数学库只使用两个值：`EDOM`（域错误）和 `ERANGE`（范围错误）。如果第二列同时给出 `EDOM` 和 `ERANGE`，则要设置的 `errno` 值由第四列或第五列给出的相关标准（如下所述）决定。第三列给出将要在任何错误消息中打印的错误代码。第四、五、六列给出各个标准定义的返回的名义上的函数值。在某些情况下，用户提供的 `matherr` 例程可以覆盖这些值并提供另一个返回值。

这些特例的具体响应由链接程序时指定的编译器标志决定，如下所述。如果指定了 `-xlibmieee` 或 `-xc99=lib`，则出现表 E-1 中所列的特例时，产生相应的浮点异常，将返回该表第六列所给出的函数值。

如果既未使用 `-xlibmieee`，也未使用 `-xc99=lib`，则其行为取决于程序链接时所指定的语言规范标志。

指定 -Xa 标志则会选择 X/Open 规范。如果出现了表中所列的任何特例，将产生相应的浮点异常、设置 errno，并返回表中第五列给出的函数值。如果提供了用户定义的 matherr 例程，则说明未定义行为。请注意，如果未给定任何其他语言规范，则默认使用 -Xa。

指定 -Xc 标志则会选择严格的 C90 规范。如果出现了特例，则产生相应的浮点异常、设置 errno，并返回表中第五列给出的函数值。这种情况下不会调用 matherr。

最后，指定 -Xs 或 -Xt 标志则会选择 SVID 规范。如果出现了特例，则产生相应的浮点异常、调用 matherr。如果 matherr 返回零，则设置 errno 并打印一条错误消息。除非由 matherr 覆盖了，否则将返回表中第四列给出的函数值。

有关 -xc99、-Xa、-Xc、-Xs 和 -Xt 标志的更多信息，请请参见 cc(1) 手册页和 Sun Studio C 编译器手册。

表 E-1 特例和 libm 函数

功能	errno	错误消息	SVID	X/Open、C90	IEEE、C99、SUSv3
acos(|x|>1)	EDOM	DOMAIN	0.0	0.0	NaN
acosh(x<1)	EDOM	DOMAIN	NaN	NaN	NaN
asin(|x|>1)	EDOM	DOMAIN	0.0	0.0	NaN
atan2(+/-0,+/-0)	EDOM	DOMAIN	0.0	0.0	+/-0.0,+/-pi
atanh(|x|>1)	EDOM	DOMAIN	NaN	NaN	NaN
atanh(+/-1)	EDOM/ERANGE	SING	+/-HUGE[1] (EDOM)	+/-HUGE_VAL[2] (ERANGE)	+/-无穷
cosh overflow	ERANGE	-	HUGE	HUGE_VAL	无穷
exp overflow	ERANGE	-	HUGE	HUGE_VAL	无穷
exp underflow	ERANGE	-	0.0	0.0	0.0
fmod(x,0)	EDOM	DOMAIN	x	NaN	NaN
gamma（0 或负整数）	EDOM	SING	HUGE	HUGE_VAL	无穷
gamma overflow	ERANGE	-	HUGE	HUGE_VAL	无穷
hypot overflow	ERANGE	-	HUGE	HUGE_VAL	无穷
j0(|x|>X_TLOSS[3])	ERANGE	TLOSS	0.0	0.0	正确答案
j1(|x| > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	正确答案
jn(|x| > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	正确答案
lgamma（0 或负整数）	EDOM	SING	HUGE	HUGE_VAL	无穷
lgamma overflow	ERANGE	-	HUGE	HUGE_VAL	无穷

表 E-1 特例和 libm 函数（续）

功能	errno	错误消息	SVID	X/Open、C90	IEEE、C99、SUSv3
log(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-无穷
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-无穷
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log1p(-1)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-无穷
log1p(x<-1)	EDOM	DOMAIN	NaN	NaN	NaN
pow(0,0)	EDOM	DOMAIN	0.0	1.0（没有错误）	1.0（没有错误）
pow(NaN,0)	EDOM	DOMAIN	NaN	NaN	1.0（没有错误）
pow(0,x<0)	EDOM	DOMAIN	0.0	-HUGE_VAL	+/-无穷
pow（x<0、非负整数）	EDOM	DOMAIN	0.0	NaN	NaN
pow overflow	ERANGE	-	+/-HUGE	+/-HUGE_VAL	+/-无穷
pow underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
remainder(x,0)	EDOM	DOMAIN	NaN	NaN	NaN
scalb overflow	ERANGE	-	+-HUGE_VAL	+/-HUGE_VAL	+/-无穷
scalb underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
sinh overflow	ERANGE	-	+/-HUGE	+/-HUGE_VAL	+/-无穷
sqrt(x<0)	EDOM	DOMAIN	0.0	NaN	NaN
y0(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-无穷
y0(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y0(x > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	正确答案
y1(0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-无穷
y1(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
y1(x > X_TLOSS)	ERANGE	TLOSS	0.0	0.0	正确答案
yn(n,0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	-无穷
yn(n,x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
yn(n,x>X_TLOSS)	ERANGE	TLOSS	0.0	0.0	正确答案

注意：

1. HUGE 在 <math.h> 中定义。 SVID 要求 HUGE 等于 MAXFLOAT，大约为 3.4e+38。
2. HUGE_VAL 在 <iso/math_iso.h>（包含在 <math.h> 中）中定义。HUGE_VAL 按无穷大计算。
3. X_TLOSS 在 <values.h> 中定义。

E.1.1 影响标准规范的其他编译器标志

上面列出的编译器标志直接选择处理表 E-1 中所列特例时所遵循的标准。其他的编译器标志可以间接地影响程序是否按照上述行为执行。

首先，-xlibmil 和 -xlibmopt 标志都会替代 libm 中某些函数的较为快速的实现。这些速度较快的实现不遵循 SVID、X/Open 或 C90 标准。它们也不会设置 errno 或调用 matherr。不过它们会产生合适的浮点异常，并传递 IEEE 754 和 / 或 C99 指定的结果。-xvector 标志与此类似，因为此标志可以使编译器将对标准数学函数的调用转换为调用矢量数学函数。

其次，使用 -xbuiltin 标志，编译器可以将在 <math.h> 中定义的标准数学函数视为内部代码和替换内联代码处理，以获得更好的性能。替换代码可以不遵循 SVID、X/Open、C90 或 C99 标准。它无需设置 errno，也无需调用 matherr 或产生浮点异常。

第三，当已定义了 C 预处理程序标记 __MATHERR_ERRNO_DONTCARE 时，会编译 <math.h> 中一系列的 #pragma 指令。这些指令将告知编译器假设标准数学函数没有任何副作用。在这个假设下，编译器可以重组对数学函数的调用以及对全局数据的引用。这里的全局数据指 errno 或可能被用户提供的 matherr 例程修改（从而导致违反以上所述的预期行为）的数据。例如，请考虑下面的代码段：

```
#include <errno.h>
#include <math.h>

...
errno = 0;
x = acos(2.0);
if (errno) {
    printf("error\n");
}
```

如果编译此代码时定义了 __MATHERR_ERRNO_DONTCARE，编译器可以假设调用 acos 并未修改 errno，并据此转换代码，完全删除对 printf 的调用。

请注意， -fast 宏标记包括 -xbuiltin、 -xlibmil、-xlibmopt 和 -D__MATHERR_ERRNO_DONTCARE 标志。

最后，由于 libm 中所有的数学函数都能根据需要产生浮点异常，因此运行启用了对这些异常的捕获的程序，通常会导致出现上面列出的标准指定的行为之外的行为。因此， -ftrap 编译器标志也能够影响对标准规范的遵循。

E.1.2 关于 C99 规范的附加说明

C99 指定了两种可能的方法，实现可以使用这两种方法处理如表 E-1 中所列的各种特例。为了表明其支持这两种方法中的哪一种，实现使用标识符 math_errhandling 计算一个有值 MATH_ERRNO (1) 或 MATH_ERREXCEPT (2) 或两者的按位“或”的整型表达式的值。（这些值是在 <math.h> 中定义的。）如果表达式 (math_errhandling & MATH_ERRNO) 为非零值，则对于函数的参数位于其数学域之外的情况，实现将 errno 设置为 EDOM ；而对于函数值下溢、溢出或等于无穷的情况，则将 errno 设置为 ERANGE。如果表达式 (math_errhandling & MATH_ERREXCEPT) 为非零值，则对于函数的参数位于其数学域之外的情况，实现将产生无效操作异常；而对于函数值下溢、溢出或等于无穷的情况，将分别产生下溢异常、溢出异常或被零除异常。

在 Solaris 上， <math.h> 将 math_errhandling 定义为 MATH_ERREXCEPT。尽管表 E-1 中列出的函数可能对所列出的特例执行其他操作，但所有 libm 函数（包括 float 和 long double 函数、复数型函数以及 C99 指定的其他函数）都会产生浮点异常作为对特例的响应。这是处理所有 C99 函数统一支持的特例的唯一方法。

最后，请注意，有三个函数 C99 或 SUSv3 都要求执行与 Solaris 默认不同的行为。下表概括了其区别。表中只列出了每个函数的 double 版本，但这些区别也同样存在于 float 和 long double 版本。在每种情况中，如果程序使用 -xc99=lib 进行链接，则遵循 SUSv3 规范，否则就按照 Solaris 的默认进行。

表 E-2 Solaris 与 C99/SUSv3 的区别

函数	Solaris 行为	C99/SUSv3 行为
pow	pow(1.0, +/-inf) 返回 NaN pow(-1.0, +/-inf) 返回 NaN pow(1.0, NaN) 返回 NaN	pow(1.0, +/-inf) 返回 1 pow(-1.0, +/-inf) 返回 1 pow(1.0, NaN) 返回 1
logb	logb(subnormal) 返回 Emin	logb(x) = ilogb(x)（当 x 为次正规数时）
ilogb	ilogb(+/-0), ilogb(+/-inf), ilogb(NaN) 不引发异常	ilogb(+/-0), ilogb(+/-inf), ilogb(NaN) 引发无效操作

E.2 LIA-1 遵循性

在本节中， LIA-1 指的是 ISO/IEC 10967-1:1994 信息技术 - 与语言无关的算法 - 第 1 节：整数和浮点算术。

Sun Studio 各编译器版本包含的 C 和 Fortran 95 编译器 （cc 和 f95）在以下方面符合 LIA-1 （段落字母对应于 LIA-1 第 8 节中的段落字母）：

a. TYPES (LIA 5.1): 符合 LIA-1 的类型为 C int 和 Fortran INTEGER。其他类型也可能符合该标准，但此处没有对它们进行说明。有关特定语言的详细规范，请参见公认的语言标准组织即将发布的 LIA-1 语言汇编材料。

b. PARAMETERS (LIA 5.1):

```
#include <values.h> /* defines MAXINT */
#define TRUE 1
#define FALSE 0
#define BOUNDED TRUE
#define MODULO TRUE
#define MAXINT 2147483647
#define MININT -2147483648
```

```
                logical bounded, modulo
                integer maxint, minint
                parameter (bounded = .TRUE.)
                parameter (modulo = .TRUE.)
                parameter (maxint = 2147483647)
                parameter (minint = -2147483648)
```

d. DIV/REM/MOD (LIA 5.1.3):

C / 和 % 以及 Fortran / 和 mod() 提供 DIVtI(x,y) 和 REMtI(x,y)。此外，modaI(x,y) 也可用于以下代码：

```
int modaI(int x, int y) {
                int t = x % y;
                if (y < 0 && t > 0)
                t -= y;
                else if (y > 0 && t < 0)
                t += y;
                    return t;
                }
```

或者：

```
                integer function modaI(x, y)
                integer x, y, t
                t = mod(x, y)
                if (y .lt. 0 .and. t .gt.0) t = t - y
```

```
if (y .gt.0 .and. t .lt. 0) t = t + y
modaI = t
return
end
```

i. 表示法 (LIA 5.1.3)：下表显示用于识别 LIA 整数运算的表示法。

表 E-3 LIA-1 遵循性 - 表示法

LIA	C	Fortran（如果不同的话）
addI(x,y)	x+y	
subI(x,y)	x-y	
mulI(x,y)	x*y	
divtI(x,y)	x/y	
remtI(x,y)	x%y	mod(x,y)
modaI(x,y)	请参见以上内容	
negI(x)	-x	

absI(x)	#include <stdlib.h>
	abs(x)
		abs(x)

signI(x)	#define signI(x) (x > 0 ? 1 : (x < 0 ? -1 : 0))	请参见以下内容
eqI(x,y)	x==y	x.eq.y
neqI(x,y)	x!=y	x.ne.y
lssI(x,y)	x<y	x.lt.y
leqI(x,y)	x<=y	x.le.y
gtrI(x,y)	x>y	x.gt.y
geqI(x,y)	x>=y	x.ge.y

以下代码显示 signI(x) 的 Fortran 表示法。

```
integer function signi(x)
integer x, t
if (x .gt.0) t=1
if (x .lt. 0) t=-1
if (x .eq.0) t=0
return
            end
```

j. 计算表达式结果：在默认情况下，如果未指定优化，则以 `int` (C) 或 `INTEGER` (Fortran) 精度计算表达式结果。考虑到了括号。未指定关联无括号表达式的计算顺序，如 a + b + c 或 a * b * c。

k. 获取参数的方法：在源代码中包含上述定义。

n. 通知：整数表达式为 x/0 和 x%0 或 mod(x,0)。在默认情况下，这些表达式生成 `SIGFPE`。如果未给 `SIGFPE` 指定信号处理程序，则进程终止并转储内存。

o. 选择机制：可以使用 `signal`(3) 或 `signal`(3F) 为 `SIGFPE` 启用用户异常处理。

附录 F

参考资料

下面的手册提供有关 SPARC® 浮点硬件的更多信息：

《SPARC Architecture Manual》，第 9 版， PTR Prentice Hall， New Jersey（新泽西），1994 年。

其余参考资料都按章节进行组织。有关如何获得标准文档和测试程序的信息在最后介绍。

F.1 第 2 章：“IEEE 算法”

Cody et al., “A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic,” *IEEE Computer*, August 1984.

Coonen, J.T., “An Implementation Guide to a Proposed Standard for Floating Point Arithmetic”, *Computer*, Vol. 13, No. 1, Jan. 1980, pp 68-79.

Demmel, J., “Underflow and the Reliability of Numerical Software”, SIAM J. *Scientific Statistical Computing*, Volume 5 (1984), 887-919.

Hough, D., “Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic”, *Computer*, Vol. 13, No. 1, Jan. 1980, pp 70-74.

Kahan, W., and Coonen, J.T., “The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments”, published in *The Relationship between Numerical Computation and Programming Languages*, Reid, J.K., (editor), North-Holland Publishing Company, 1982.

Kahan, W., “Implementation of Algorithms”, *Computer Science Technical Report No. 20*, University of California, Berkeley CA, 1973. Available from National Technical Information Service, NTIS Document No. AD–769 124 (339 pages), 1-703-487-4650 (ordinary orders) or 1-800-336-4700 (rush orders.)

Karpinski, R., “Paranoia: a Floating-Point Benchmark”, *Byte*, February 1985.

Knuth, D.E., *The Art of Computer Programming, Vol.2: Semi-Numerical Algorithms*, Addison-Wesley, Reading, Mass, 1969, p 195.

Linnainmaa, S., "Combatting the effects of Underflow and Overflow in Determining Real Roots of Polynomials", SIGNUM Newsletter 16, (1981), 11-16.

Rump, S.M., "How Reliable are Results of Computers?", translation of "Wie zuverlassig sind die Ergebnisse unserer Rechenanlagen?", *Jahrbuch Uberblicke Mathematik 1983*, pp 163-168, C Bibliographisches Institut AG 1984.

Sterbenz, P, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974. (Out of print; most university libraries have copies.)

Stevenson, D. et al., Cody, W., Hough, D. Coonen, J., various papers proposing and analyzing a draft standard for binary floating-point arithmetic, *IEEE Computer*, March 1981.

The Proposed IEEE Floating-Point Standard, special issue of the ACM *SIGNUM Newsletter*, October 1979.

F.2 第 3 章："数学库"

Cody, William J. and Waite, William, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1980.

Coonen, J.T., *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, PhD Dissertation, University of California, Berkeley, 1984.

Tang, Peter Ping Tak, *Some Software Implementations of the Functions Sin and Cos*, Technical Report ANL-90/3, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1990.

Tang, Peter Ping Tak, *Table-driven Implementations of the Exponential Function EXPM1 in IEEE Floating-Point Arithmetic*, Preprint MCS-P125-0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February, 1990.

Tang, Peter Ping Tak, *Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic*, ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989, pp 144-157 communication, July 18, 1988.

Tang, Peter Ping Tak, *Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic*, preprint MCS-P55-0289, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1989 (to appear in ACM Trans. on Math. Soft.)

Park, Stephen K. and Miller, Keith W., “Random Number Generators: Good Ones Are Hard To Find”, *Communications of the ACM*, Vol. 31, No. 10, October 1988, pp 1192 - 1201.

F.3 第 4 章：“异常和异常处理”

Coonen, J.T, “Underflow and the Denormalized Numbers”, *Computer*, 14, No. 3, March 1981, pp 75-87.

Demmel, J., and X. Li, “Faster Numerical Algorithms via Exception Handling”, *IEEE Trans. Comput.* Vol. 48, No. 8, August 1994, pp 983-992.

Kahan, W., “A Survey of Error Analysis”, *Information Processing 71*, North-Holland, Amsterdam, 1972, pp 1214-1239.

F.4 附录 B：“SPARC 行为和实现”

下面的制造商文档提供有关浮点硬件和主要处理器芯片的更多信息。这些信息按系统结构进行组织。

Texas Instruments, *SN74ACT8800 Family, 32-Bit CMOS Processor Building Blocks: Date Manual*, 1st edition, Texas Instruments Incorporated, 1988.

Weitek, *WTL 3170 Floating Point Coprocessor: Preliminary Data*, 1988, published by Weitek Corporation, 1060 E. Arques Avenue, Sunnyvale, CA 94086.

Weitek, *WTL 1164/WTL 1165 64-bit IEEE Floating Point Multiplier/Divider and ALU: Preliminary Data*, 1986, published by Weitek Corporation, 1060 E. Arques Avenue, Sunnyvale, CA 94086.

F.5 标准

American National Standard for Information Systems, *ISO/IEC 9899:1990 Programming Languages - C*, American National Standards Institute, 1430 Broadway, New York, NY 10018.

American National Standard for Information Systems ISO/IEC 9899:1999 Programming Languages - C(C99), American National Standards Institute, 1430 Broadway, New York, NY 10018.

IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985 (IEEE 754), published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1985.

IEEE Standard Glossary of Mathematics of Computing Terminology, ANSI/IEEE Std 1084-1986, published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1986.

IEEE Standard Portable Operating System Interface for Computer Environments (POSIX®), IEEE Std 1003.1-1988, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017.

System V Application Binary Interface (ABI), AT&T (1-800-432-6600), 1989.

SPARC System V ABI Supplement (SPARC ABI), AT&T (1-800-432-6600), 1990.

System V Interface Definition, 3rd edition, (SVID89, or SVID Issue 3), Volumes I–IV, Part number 320-135, AT&T (1-800-432-6600), 1989.

X/OPEN Portability Guide, Set of 7 Volumes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1989.

F.6 测试程序

浮点算法和数学库的许多测试程序都可以从 ucbtest 包装中的 Netlib 中找到。这些程序包括各种版本的 `Paranoia`、Z. Alex Liu 的 Berkeley Elementary Function 测试程序、IEEE 测试矢量和基于由 W. Kahan 教授开发的数论方法（这些方法针对正确舍入的乘、除和平方根生成高难度的测试实例）的程序。

ucbtest 位于 `http://www.netlib.org/fp/ucbtest.tgz`。

术语表

本词汇表介绍计算机浮点算法术语。它还介绍了与并行处理有关的术语和缩略词。

附加到术语后面的符号 “| |”指明该术语与并行处理有关。

准确度 衡量两个数的近似程度。例如，计算结果的准确度通常反映了它与精确运算结果相差的计算误差大小。可以用有效数位来表示准确度 （例如，“结果精确到 6 位数字”）；更普遍的作法是使用保留相关数学特性来表示准确度

（例如，“结果具有正确的代数符号”）。

数组处理 | | 一组同时工作的处理器，每个处理器处理一个数组元素，因此单个运算可并行应用于所有的数组元素。

关联性 | | 请参见**缓存**、**直接映射**的**缓存**、**完全关联的缓存**、**集关联缓存**。

异步控制 | | 在收到发生了特定事件的指示（信号）时开始执行特定操作的计算机控制行为。异步控制依靠称为“锁定”的同步机制来协调处理器。另请参见**互斥**、**互斥锁定**、**信号锁定**、**单锁定策略**、**旋转锁定**。

屏障 | | 用于协调任务的同步机制，即使不涉及数据访问。屏障与门类似。并行运行的处理程序或线程在不同的时间到达门，但直至所有处理程序都到达门时，才会让它们通过。例如，假定每天下班时，要求所有银行出纳员清点存入的金额和提取的金额。然后向银行副总裁汇报这些总额，副总裁核对这些总额之和以检查借款和贷款是否相等。出纳员各自的工作速度不同；即他们在不同的时间汇总完其交易情况。屏障机制禁止出纳员在核实完这些总额之和前下班回家。如果借款和贷款不相等，则所有出纳员都必须回到其办公桌前找出错误。在副总裁获得满意的总额之和后，就会去除屏障。

偏置指数 为使存储的指数范围非负数而选择的以 2 为基数的指数和常量 （偏差）之和。例如， 2^{-100} 的指数以 IEEE 单精度格式存储时为 (-100) + (单精度偏差 127) = 27。

二进制间距 任意两个连续的 2 的幂指数的间距。

阻塞状态 | | 线程正在等待资源或数据 （例如从某个等待处理的磁盘读取操作中返回的数据），或者等待另一个线程解锁资源。

绑定线程 对于 Solaris 线程而言，永久性分配给特定 LWP 的线程称为绑定线程。可以实时地调度绑定线程，并使之具有比系统中的所有其他活动线程更高的优先级，而不是仅比进程中的线程优先级高。 LWP 是一个实体，可使用与任何 UNIX 进程相同的默认调度优先级对它进行调度。

缓存 很小、速度很快并由硬件控制的存储器，它用作处理器和主内存之间的缓冲区。缓存包含最近使用的指令和数据内存地址的副本 （地址和内容）。每个地址引用都先经过缓存。如果缓存中没有所需的指令或数据，则发生缓存未命中情况。可通过总线将内容从主内存提取到正在执行的指令所指定的 CPU 寄存器中，同时将一个副本写入到缓存中。稍后可能会使用相同的地址，如果发生这种情况，则在缓存中找到该地址，从而发生缓存命中情况。如果将数据写入到该地址中，则硬件不仅写入到缓存中，而且还生成一个到主内存的直写。

另请参见**关联性**、**电路交换**、**直接映射的缓存**、**完全关联的缓存**、*MBus*、**包交换**、**集关联缓存**、**回写**、**直写**、 *XDBus*。

缓存局域性 程序并不以相同的概率同时访问其所有代码或数据。如果将最近访问的信息存放在缓存中，则可增大本地找到信息的可能性，从而无需访问内存。局域性原则规定程序在任何瞬间访问其地址空间中相对较小的一部分。共有两种不同类型的局域性：时间和空间。

时间局域性是指重复使用最近访问的项目的倾向性。例如，大多数程序包含循环，因此可能会重复访问指令和数据。时间局域性将最近访问的项目（靠近处理器）保存在缓存中，而不是要求进行内存访问。另请参见**缓存**、**竞争性缓存**、**伪共享**、**写无效**、**写更新**。

空间局域性是指所引用项目的地址接近于其他最近访问的项目的倾向性。例如，数组元素或记录访问显示出自然的空间局域性。缓存使用以下方法来利用空间局域性：将块（多个连续的字）从内存移到缓存中并靠近处理器。另请参见**缓存**、**竞争性缓存**、**伪共享**、**写无效**、**写更新**。

链 某些流水线结构的硬件特性，它允许在将某个运算的结果写入其目标寄存器的同时，直接将该结果用作第二个运算的操作数。两个链式运算的总的时间周期比各指令单独的时间周期之和短。例如， TI 8847 支持（相同精度的）连续 `fadd`、`fsub` 和 `fmul` 链。链式 `faddd/fmuld` 需要 12 个周期，而连续非链式 `faddd/fmuld` 需要 17 个周期。

电路交换 缓存彼此之间进行通信以及与主内存之间进行通信使用的机制。专用连接 （电路）是在缓存之间以及缓存与主内存之间建立的。在建立某个电路时，不能通过总线传输任何其他通信。

一致性 在具有多个缓存的系统中，此机制可确保所有处理程序在任何时候看到的内存映像都相同。

常见异常 上溢、无效和除法这三种浮点异常统称为 `ieee_flags`(3m) 和 `ieee_handler`(3m) 的常见异常。之所以将它们称为常见异常，是因为通常将它们捕获为误差。

竞争性缓存 竞争性缓存通过结合使用写无效和写更新来保持缓存一致性。竞争性缓存使用计数器来计量共享数据的时间。系统依据最近最少使用 (LRU) 的算法，将共享数据从缓存中清除。这可导致共享数据再次变成专用数据，从而消除了使用缓存一致性协议来访问内存（通过底板带宽）以保持多个副本同步的需要。另请参见**缓存**、**缓存局域性**、**伪共享**、**写无效**、**写更新**。

并行操作 ||　并行执行两个或多个活动线程或进程。在单处理器上，可通过在线程之间快速切换，使表面上看起来是同时完成这两个线程。在多处理器系统上，可实现真正的并行执行。另请参见**异步控制**、**多处理器系统**、**线程**。

并发进程 ||　在多处理器上并行执行的进程，或在单处理器上异步执行的进程。并发进程可以彼此之间进行交互，一个进程可以暂停执行，直至从另一个进程收到信息或发生外部事件时再继续执行。另请参见**进程**、**顺序执行的进程**。

条件变量 ||　对于 Solaris 线程而言，条件变量允许一次性地阻塞线程，直至满足某种条件时为止。条件是在互斥锁定的保护下进行测试的。当条件为假时，线程因某个条件变量而阻塞，并一次性地释放互斥，等待条件的改变。当另一个线程改变条件时，它可以向关联的条件变量发信号以唤醒一个或多个等待的线程，重新获取互斥并重新评估条件。如果已为变量分配了可写入的内存，且该变量在具有协作关系的进程间共享，并已被初始化，则可以使用条件变量同步此进程和其他进程中的线程。

上下文切换　在多任务处理操作系统（如 SunOS™ 操作系统）中，进程运行固定的一段时间。在这段时间结束时，CPU 收到一个来自计时器的信号，中断当前运行的进程，并准备运行新的进程。CPU 为旧进程保存寄存器，然后为新进程加载寄存器。将旧进程的状态切换到新进程上的过程称为上下文切换。切换上下文所花的时间是系统开销；所需的时间取决于寄存器的数量以及是否需要使用特殊的指令来保存与进程关联的寄存器。

控制流模型 ||　冯 • 诺伊曼计算机模型。此模型指定控制流；即指定在程序的每一步中执行哪条指令。所有 Sun 工作站都是冯 • 诺伊曼模型的示例。另请参见**数据流模型**、**按需求驱动的数据流**。

关键区域 ||　代码中不可分割的一部分，每次只能由一个线程执行，执行时不能被其他线程中断；例如，访问共享变量的代码。另请参见**互斥**、**互斥锁定**、**信号锁定**、**单锁策略和转锁**。

关键资源 ||　在任何给定时间至多有一个线程可以使用的资源。如果要求几个异步线程协调它们对关键资源的访问，则它们使用同步机制来执行此操作。另请参见**互斥**、**互斥锁定**、**信号锁定**、**单锁策略和转锁**。

数据流模型 ||　此计算机模型指定对数据所执行的运算并忽略指令顺序。即，根据数据值的可用性而不是根据指令的可用性向前计算。另请参见**控制流模型**、**按需求驱动的数据流**。

数据争用 ||　在多线程处理中，两个或多个线程同时访问共享资源的情形。结果是不确定的，具体情况取决于线程访问资源的顺序。当使用相同输入重复运行程序时，这种情况（称为 “数据争用” ）可产生不同的结果。另请参见**互斥**、**互斥锁定**、**信号锁定**、**单锁策略和转锁**。

死锁 ||　当两个 （或多个）单独的活动进程争用资源时出现的情况。假设进程 P 要求使用资源 X 和 Y，并且按该顺序请求使用这些资源，同时进程 Q 要求使用资源 Y 和 X，并且按该顺序请求这些资源。如果进程 P 获取了资源 X，并且进程 Q 同时获取了资源 Y，则两个进程均无法继续执行 - 每个进程要求使用已分配给另一个进程的资源。

默认结果　作为导致异常的浮点运算的结果而提供的值。

按需求驱动的数据流 || 当另一个也处于激活状态的任务需要某个任务的结果时，处理器便激活该任务，以便执行操作 （如图形约简模型）。图形约简程序由可简化的表达式组成，在计算过程中可使用其计算值来替代这些表达式。通常，约简是并行完成的 - 除以前约简数据的可用性外，任何其他因素均不能禁止进行并行约简。另请参见**控制流模型**、**数据流模型**。

非规格化数 次正规数的旧名称。

直接映射的缓存 || 直接映射的缓存是单向集关联缓存。即，每个缓存条目保存一个块，并形成包含**一个**元素的**单个**集合。另请参见**缓存**、**缓存局域性**、**伪共享**、**完全关联的缓存**、**集关联缓存**、**写无效**、**写更新**。

分布式内存体系结构 || 互连网络拓扑中每个节点的本地内存和处理器的组合。每个处理器只能直接访问系统总内存的一部分。任意两个处理器之间使用消息传递进行通信，并且没有全局的共享内存。因此，当必须共享某个数据结构时，程序会向拥有该结构的进程发出发送 / 接收消息。另请参见**进程间通信**、**消息传递**。

双精度 使用两个字来表示一个数以保持或提高精度。在 SPARC® 工作站上，双精度为 64 位 IEEE 双精度。

异常 当执行的原子算法运算没有可普遍接受的结果时，就会发生算法异常。“原子”和“可接受”两词的含义因时间和地点而异。

指数 浮点数的一部分，在确定可表示数的值时，它表示底的整数幂指数。

伪共享 || 由两个线程独立访问的两个无关的数据存在于同一块中时，缓存中发生的情况。此块可能会在没有什么正常原因的情况下最终在缓存之间被弹来弹去。如果注意到此类情况并重新安排数据结构以避免出现伪共享，则可大大提高缓存性能。另请参见**缓存**、**缓存局域性**。

浮点数系统 用于表示实数子集的一种系统，可表示数之间的间距不是固定的绝对常数。此类系统可以通过底、符号、有效数字和指数（通常为偏置指数）来表征。该数的值是其有效数字和底的无偏置指数幂的符号乘积。

完全关联的缓存 || 具有 *m* 个条目的完全关联缓存是 *m* 向集关联缓存。即，它包含具有 *m* 个块的**单个**集合。缓存条目可以位于该集合内 *m* 块中的任何一个。另请参见**缓存**、**缓存局域性**、**直接映射的缓存**、**伪共享**、**集关联缓存**、**写无效**、**写更新**。

渐进下溢 当浮点运算出现下溢时，返回一个次正规数而不是 0。这种下溢处理方法最大限度地降低了较小数浮点计算的不准确度。

隐藏位 硬件为确保正确舍入而使用的额外位，软件无法访问这些位。例如，IEEE 双精度运算使用三个隐藏位来计算 56 位结果，然后舍入为 53 位。

IEEE 754 标准 电器和电子工程师协会制订的用于二进制浮点算法的标准，该标准于 1985 年发布。

内置模板 在 Sun Studio 编译器内联处理传递过程中，用于替换所定义的函数调用的汇编语言代码段。例如，内置模板文件 (`libm.il`) 中的数学库可使用它来访问 C 程序中的三角函数和其他初等函数的硬件实现。

互连网络拓扑 互连拓扑描述了处理器的连接方式。所有网络都是由交换机组成的，这些交换机链接到处理器 - 内存节点和其他交换机上。共有四种基本形式的拓扑：星形、环形、总线以及完全连接的网络。星形拓扑由一个单独的集线器处理器组成，其他处理器直接连接到该集线器上，非集线器处理器彼此并不直接相连。在环形拓扑中，所有处理器均在一个环上，通信通常沿环的一个方向进行。总线拓扑是非循环的，并且所有节点彼此相连；因此通信传输是双向进行的，并且需要进行某种形式的仲裁以确定在任意特定时刻哪个处理器可以使用总线。在完全连接（交叉）的网络中，每个处理器与每个其他处理器之间具有双向链接。

商业上使用的并行处理器使用多级网络拓扑。多级网络拓扑的特征有：二维网格和布尔型 N 立方体。

进程间通信 活动进程之间的消息传递。另请参见**电路交换**、**分布式内存体系结构**、*MBus*、**消息传递**、**包交换**、**共享内存**、*XDBus*。

IPC 请参见**进程间通信**。

轻量进程 可使用内核的控制线程将 Solaris 线程实现为用户级库，这些控制线程称为轻量进程 (LWP)。在 Solaris 环境中，一个进程就是一个共享内存的 LWP 集合。每个 LWP 都具有 UNIX 进程的调度优先级，并共享该进程的资源。 LWP 使用同步机制（如锁定）来协调其对共享内存的访问。可以将 LWP 视为一个执行代码或系统调用的虚拟 CPU。线程库调度进程中 LWP 池上的线程，其方式与内核调度处理器池上的 LWP 的方式基本相同。每个 LWP 是由内核单独调度的；执行单独的系统调用；导致单独的页面错误；以及在多处理器系统上并行运行。 LWP 是由内核按照其调度级别和优先级在可用 CPU 资源上调度的。

锁定 此机制用于实施对共享数据进行串行访问的策略。线程或进程使用特定的锁定以获取对该锁定保护的共享内存的访问权限。只有编程人员知道必须锁定哪些内容，从这个意义上讲，数据锁定和解除锁定是自发的。另请参见**数据争用**、**互斥**、**互斥锁定**、**信号锁定**、**单锁定策略**、**转锁**。

LWP 请参见**轻量进程**。

MBus MBus 是一个用于处理器 / 内存 /IO 互连的总线规范。 SPARC International 向几家生产互操作 CPU 模块、IO 接口和内存控制器的硅供应商颁发了该规范的使用许可证。 MBus 是电路交换协议，它在单个总线上合并读取请求和响应。 MBus 第 I 级定义了单处理器信号； MBus 第 II 级为写无效缓存一致性机制定义了多处理器扩展。

内存 此介质可保存信息以便随后进行检索。此术语通常指的是机器指令可直接寻址的计算机内部存储。另请参见**缓存**、**分布式内存**、**共享内存**。

消息传递 在分布式内存体系结构中，进程彼此之间进行通信使用的机制。没有进程用于存放消息的共享数据结构。消息传递允许进程将数据发送到另一个进程，并允许指定的接收者同步数据的到达时间。

MIMD 请参见**多指令多数据**、**共享内存**。

多线程安全的 在 Solaris 环境中，库内的函数调用是多线程安全的或非多线程安全的；多线程安全的代码也称为“可重入”代码。即，几个线程可以同时调用模块中的一个给定函数，并且由函数代码来处理该调用。假设条件是仅由模块函数来访问线程之间共享

的数据。如果模块客户机可以使用易变的全局数据，则还必须在接口中显示相应的锁定。再者，除非假定客户机在适当的时间一致地使用锁定，否则无法将模块函数变为可重入版本。另请参见**单锁定策略**。

多指令多数据 在此系统模型中，多个处理器可以同时对不同的数据执行不同的指令。再者，这些处理器的操作方式具有很大的独立性，就好像它们是单独的计算机一样。它们没有中央控制器，并且通常不使用锁步操作模式。大多数实际银行使用的就是这种运行方式。出纳员彼此之间并不商议，他们也不同时执行每个交易的每个步骤。相反，他们独立工作，直至发生数据访问冲突时为止。在处理交易时并不考虑时间或客户顺序。但必须显式地禁止客户 A 和 B 同时访问共有的 AB 帐户余额。 MIMD 依靠称为“锁定”的同步机制来协调对共享资源的访问。另请参见**互斥**、**互斥锁定**、**信号锁定**、**单锁策略和转锁**。

多次读单次写 在并发环境中，第一个进行数据写访问的进程对它进行独占访问，因而无法进行并发写访问或同时进行读取和写入访问。但是，可以由多个阅读器读取该数据。

多处理器 请参见**多处理器系统**。

多处理器总线 在共享内存多处理器机器中，每个 CPU 和缓存模块通过一条总线连接起来，该总线还包含内存和 IO 连接。总线强制实施缓存一致性协议。另请参见**缓存**、**一致性**、*Mbus*、 *XDBus*。

多处理器系统 在此系统中，在任意给定时刻都可以有多个活动的处理器。当这些处理器各自执行单独的进程时，它们完全异步地运行。但是，当处理器访问关键的系统资源或关键的系统代码区域时，它们之间必须进行同步。另请参见**关键区域**、**关键资源**、**多线程处理**、**单处理器系统**。

多任务处理 在单处理器系统中，表面上看起来大量线程是并行运行的。这是通过快速切换线程完成的。

多线程处理 应用程序可以同时有多个活动线程或处理器。多线程的应用程序可以在单处理器系统和多处理器系统中运行。另请参见**绑定线程**、**多线程安全的**、**单锁定策略**、**线程**、**非绑定线程**、**单处理器**。

互斥锁定 实现互斥锁定机制的同步变量。另请参见**条件变量**、**互斥**。

互斥 在并发环境中，线程在没有竞争线程访问的情况下更新关键资源的能力。另请参见**关键区域**、**关键资源**。

NaN 表示非数。以浮点格式编码的符号实体。

正规数 在 IEEE 算法中，偏置指数不是零或最大值 （全为 1）的数字，它表示正常范围内实数的一个子集，并具有很小的限定相对误差。

包交换 在共享内存体系结构中，缓存彼此之间进行通信以及与主内存之间进行通信使用的机制。在包交换中，可以将通信分为很小的段（称为“包”），可在总线上多路传输这些包。缓存和内存硬件可通过包中包含的标识来确定是将包发送到该硬件，还是将包发送到其最终目标。包交换允许多路传输总线通信，并将无序（没有顺序）包放在总线上。可在目标（缓存或主内存）处重新组装无序包。另请参见**缓存**、**共享内存**。

范例 用于描述解决问题的计算机方案的现实模型。范例提供了理解和解决现实问题的环境。因为范例是一个模型，所以它从现实情况中抽象出问题的细节，因而使问题更容易解决。但是，就像所有抽象一样，模型可能并不准确，这是因为它只对现实情况做近似处理。另请参见**多指令多数据**、**单指令多数据**、**单指令单数据**、**单程序多数据**。

并行处理 在多处理器系统中，可以实现真正的并行执行，其中可同时存在大量活动的线程或进程。另请参见**并发性**、**多处理器系统**、**多线程处理**、**单处理器**。

并行性 请参见**并发进程**、**多线程处理**。

流水线 如果可以将应用于数据的汇总函数分为不同的处理阶段，则不同的数据部分可以从一个阶段流入另一个阶段；例如，包含以下阶段的编译器：词法分析、语法分析、类型检查、代码生成等。在第一个程序或模块通过词法分析后，就可以将它传递给语法分析，同时词法分析开始对第二个程序或模块进行分析。另请参见**数组处理**、**矢量处理**。

流水线操作 在这种硬件功能中，将操作分为多个阶段，每个阶段（通常）要花一个周期来完成。当每个周期发出新的操作时，就会填充流水线。如果流水线中的指令没有相关性，则每个周期可以提供新的结果。链式执行意味着对相关指令进行流水线处理。如果无法链式处理相关的指令（硬件不支持这些特定指令链），流水线操作就会停止。

精度 定量测量可表示数的密度。例如，在具有 53 个有效位精度的二进制浮点格式中，2 的任何两个相邻幂指数之间共有 2^{53} 个可表示数（在正规数的范围内）。不要将精度和准确度混淆起来，后者表示两个数的近似程度。

进程 此活动单元具有以下特征：单个顺序执行的线程、当前状态以及相关的系统资源集合。

静态 NaN 通过几乎每个算术运算进行传递而不会引发新异常的 NaN（非数）。

基数 任何数系统的基数。例如，2 是二进制系统的基数，10 是十进制记数系统的基数。SPARC 工作站使用以 2 为基数的算法； IEEE 标准 754 是以 2 为基数的算法标准。

舍入 必须向上或向下舍入不准确的结果以获取可表示的值。在向上舍入结果时，它就会增加到下一个可表示的值。在向下舍入时，它就会减少到上一个可表示的值。

舍入误差 在将实数舍入到机器可表示的数时产生的误差。大多数浮点计算会产生舍入误差。对于任何一个浮点运算， IEEE 标准 754 规定结果不应产生多于一个舍入误差。

信号锁定 此同步机制通过协同异步线程来控制对关键资源的访问。另请参见**信号**。

信号 || E. W. Dijkstra 提出的一种专门用途的数据类型，可用于协调对特定资源或一组共享资源的访问。信号具有整数值（不能为负数），并且允许对它进行两种运算。信号（V 或 up）运算将该值增加 1，通常表示资源已变为可用资源。等待（P 或 down）运算将该值减少 1（条件是没有将该值变为负数），通常表示将要开始使用某个可用资源。另请参见**信号锁定**。

顺序执行的进程 || 按以下方式执行的进程：在完成某个进程后，才能开始执行下一个进程。另请参见**并发进程**、**进程**。

集关联缓存 || 在集关联缓存中，可以将每个块放在固定数量的地址（至少有两个）中。块具有 *n* 个地址的集关联缓存称为 *n* 向集关联缓存。*n* 向集关联缓存由多个集合组成，每个集合由 *n* 个块组成。可以将某个块放在该集合的任意位置（元素）中。增加关联性级别（集合中块的数量）就会增加缓存命中率。另请参见**缓存**、**缓存局域性**、**伪共享**、**写无效**、**写更新**。

共享内存体系结构 || 在总线连接的多处理器系统中，进程或线程通过所有处理器共享的全局内存进行通信。此共享数据段放在协作进程的地址空间中（在其专用数据段和堆栈段之间）。`fork()` 产生的后续任务复制其地址空间中除共享数据段以外的所有数据段。共享内存要求程序语言扩展和库例程支持该模型。

信号 NaN 当作为操作数出现时引发无效运算异常的 NaN（非数）。

有效数字 浮点数的一部分，可用底的有符号幂乘以它以确定该数的值。在正规化数中，有效数字由小数点左侧的单个非零数字以及右侧的小数组成。

SIMD || 请参见**单指令多数据**。

单指令多数据 || 此系统模型中有多个处理元素，但它们用于同时执行相同的指令；即，使用一个程序计数器顺序执行程序的单个副本。SIMD 在解决以下问题方面特别有用：包含大量需要统一更新的数据；例如，一般的数值计算。很多科学和工程方面的应用程序（例如，图像处理、粒子模拟以及有限元方法）自然归类为 SIMD 范例。另请参见**数组处理**、**流水线**、**矢量处理**。

单指令单数据 || 在此传统单处理器模型中，获取和执行指令序列的单个处理器对指令内部指定的数据项进行运算。这就是原始的冯 • 诺伊曼计算机运算模型。

单精度 使用一个计算机字来表示一个数。

单程序多数据 || 一种异步并行方式，在没有锁步协同的情况下同时处理不同的数据。在 SPMD 中，处理器可以同时执行不同的指令；例如，不同的 `if-then-else` 语句分支。

单锁定策略 || 在单锁定策略中，只要应用程序中有线程正在运行，线程会获取应用程序范围内的互斥锁定，然后在该线程阻塞前释放锁定。单锁定策略要求系统中所有模块和库的协作使用单个锁定进行同步。因为在任何给定时间内只能有一个线程可以访问共享数据，所以每个线程都具有一致的内存视图。此策略在单处理器中很有效，条件是在释放锁定前将共享内存置于一致的状态，并且经常释放锁定以使其他线程能够运行。并且，在单处理器系统中，如果在大多数 I/O 操作过程中锁定没有撤消，并行操作效率就会降低。不能在多处理器系统中应用单锁定策略。

SISD || 请参见**单指令单数据**。

嗅探 ||　用于保持缓存一致性的最常见协议称为 “嗅探”。缓存控制器监视或嗅探总线以确定缓存是否包含共享块的副本。

对于读取而言，多个副本可以位于不同处理器的缓存中，但因为处理器需要最新的副本，所以所有处理器在写入后必须获取新的值。另请参见**缓存**、**竞争性缓存**、**伪共享**、**写无效**、**写更新**。

对于写入而言，处理器必须拥有独占访问权限，才能写入到缓存中。写入到非共享块不会导致总线通信。写入到共享数据的结果是，要么使所有其他副本无效，要么使用写入的值更新共享副本。另请参见**缓存**、**竞争性缓存**、**伪共享**、**写无效**、**写更新**。

转锁 ||　线程使用转锁来反复测试锁定变量，直至其他某个任务释放该锁定时为止。即，等待线程旋转锁定，直至清除锁定时为止。然后，当等待线程在关键区域内部时设置锁定。当关键区域中的工作完成后，线程清除转锁，以使其他线程可以进入关键区域。转锁和互斥的差别之处在于，在试图获取其他线程保持的互斥时将阻塞和释放 LWP ；转锁不会释放 LWP。另请参见**互斥锁定**。

SPMD ||　请参见**单程序多数据**。

stderr　标准误差是指向标准误差输出的 Unix 文件指针。在启动某个程序时，就会打开此文件。

Store 0　将算术运算的下溢结果刷新为零。

次正规数　在 IEEE 算法中，偏置指数为 0 的非零浮点数。次正规数是介于零和最小正规数之间的数。

线程 ||　单个 UNIX 进程地址空间中的控制流。Solaris 线程提供轻量形式的并发任务，允许在公共用户地址空间中使用多个控制线程，并产生最小限度的调度和通信开销。线程共享相同的地址空间、文件描述符（当一个线程打开文件时，其他线程可以读取该文件）、数据结构和操作系统状态。线程使用程序计数器和堆栈来跟踪局部变量和返回地址。线程之间通过使用共享数据和线程同步操作来进行交互。另请参见**绑定线程**、**轻量进程**、**多线程处理**、**非绑定线程**。

拓扑 ||　请参见**互连网络拓扑**。

2 的补码　可使用以下方法构成二进制数基数的补码：用 1 减每个数字，然后在最低有效数位上加 1，并执行任何所需的进位。例如， 1101 的 2 的补码为 0011。

ulp　代表最后一位中的单位。在二进制格式中，尾数的最低有效位（第 0 位）是最后一位中的单位。

ulp(x)　表示以工作格式截断的 x 的 `ulp`。

非绑定线程 ||　对于 Solaris 线程而言，在 LWP 池上调度的线程称为非绑定线程。线程库调用并分配 LWP 以执行可运行的线程。如果线程在同步机制中变为阻塞线程 （如互斥锁定），则在进程内存中保存线程的状态。然后，线程库将另一个线程分配给 LWP。另请参见**绑定线程**、**多线程处理**、**线程**。

underflow　如果浮点算术运算的结果很小，仅通过正常舍入无法以目标浮点格式将其表示为正规数，就会发生这种情况。

单处理器系统 || 单处理器系统在任意给定时间只有一个处理器处于活动状态。此单处理器可以运行多线程应用程序以及传统的单指令单数据模型。另请参见**多线程处理**、**单指令单数据**、**单锁定策略**。

矢量处理 || 以一致的方式处理数据序列，在处理矩阵（其元素为矢量）或其他数据数组时通常使用这种处理方式。此顺序执行的数据处理可以使用流水线处理。另请参见**数组处理**、**流水线**。

字元 作为给定计算机内的单个实体存储、寻址、传输和操作的有序字符集合。在 SPARC 工作站环境中，一个字为 32 位。

环绕式数字 在 IEEE 算法中，可通过以下方法用某个值创建的数：为该值的指数添加固定偏移以使其环绕值位于正规数范围内；如果不进行此类处理，该值就会出现上溢或下溢。目前的 SPARC 工作站上不会产生环绕结果。

回写 || 用于在缓存和主内存之间保持一致性的写入策略。回写（也称为复制回或存储在）仅写入到本地缓存中的块。写入是在缓存内存的速度下进行的。只有在另一个处理器引用相应的内存地址时，才会将修改后的缓存块写入到主内存中。处理器可以在缓存块中写入多次，并且仅在引用时才将其写入到主内存中。因为每个写入并不经过内存，所以回写减少了对总线带宽的需求。另请参见**缓存**、**一致性**、**直写**。

写无效 || 通过从本地缓存中读取数据来保持缓存一致性，直到有写入发生。要更改变量的值，写入处理器先使其他缓存中的所有副本无效。然后，写入处理器就可以任意更新其本地副本，直至另一处理器请求该变量时为止。写入处理器通过总线发出一个无效信号，所有缓存检查它们是否有副本；如果有的话，则它们必须使包含该字的块无效。此方案允许多个读取处理器，但只允许一个写入处理器。写 - 无效仅在第一次写入时使用总线使其他副本无效；随后的本地写入不会产生总线通信，因此减少了对总线带宽的需求。另请参见**缓存**、**缓存局域性**、**一致性**、**伪共享**、**写更新**。

直写 || 用于在缓存和主内存之间保持一致性的写入策略。直写式（也称为直接存储）写入到主内存和本地缓存中的块。直写的优点是主内存具有最新的数据副本。另请参见**缓存**、**一致性**、**回写**。

写更新 || 写更新（也称为写 - 广播）通过立即更新所有缓存中共享变量的所有副本来保持缓存一致性。这是一种直写形式，因为所有写入都通过总线来更新共享数据的副本。写更新的优点是新的值立即出现在缓存中，这可减少延迟。另请参见**缓存**、**缓存局域性**、**一致性**、**伪共享**、**写无效**。

XDBus || XDBus 规范使用低阻抗的 GTL（射电收发器逻辑）收发器信号，以更高的时钟速率来驱动更长的底板。 XDBus 通过以下方法来支持大量 CPU：使用多个交叉存储的存储库来提高吞吐量。XDBus 使用包交换协议来处理分开的请求和响应，以便更有效地利用总线。XDBus 还定义了交叉存储的方案，以便将一个、两个或四个单独的总线数据路径用作单个底板以提高吞吐量。XDBus 支持写无效、写更新以及竞争性缓存一致性方案，并且包含几个拥塞控制机制。另请参见**缓存**、**一致性**、**竞争性缓存**、**写无效**、**写更新**。

索引

G

I

J

L

M

N

P

S

W

X

Y

Z