Oracle® Developer Studio 12.5：数值计算指南

文件号码 E71991
2016 年 6 月

ORACLE

文件号码 E71991

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

目录

图

表

使用本文档

- **概述** – 介绍基于 SPARC 和 x86 且运行 Oracle Solaris 操作系统的系统上的软件和硬件所支持的浮点环境。
- **目标读者** – 应用程序开发者、系统开发者、架构师、支持工程师。
- **必备知识** – 编程经验、软件开发测试以及构建和编译软件产品的能力。

产品文档库

有关本产品的最新信息和已知问题均包含在文档库中，网址为 http://docs.oracle.com/cd/E37069_01。

反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关本文档的反馈。

♦ ♦ ♦ 第 1 章

1 介绍

可以通过 SPARC 和 Intel x86 系统上的 Oracle 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。这些系统实现了由 IEEE 标准 754 为二进制浮点运算指定的算法模型。本手册解释了如何在这些系统上使用 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，并通过由共同实现 IEEE 标准 754 的硬件、系统软件和软件库提供给应用程序编程人员。IEEE 标准 754 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的发展。

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情况的控制、指示出现 IEEE 数字异常的状态标志以及当发生此类异常且没有用户为其定义的处理程序时 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 标准 754 在引发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。数学库还提供了处理特殊 IEEE 值的函数调用，例如 `Inf`（无穷大）或 `NaN`（not a number，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 标准规定或推荐的计算机制。一般来说，本手册可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。

与浮点算法相关的大多数问题都涉及数字的基本运算。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相关。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
- 除以零
- 试图计算负数的平方根

在某些其他运算中，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：程序立即中止。在某些较旧的计算机中，计算会继续进行，但产生无用结果。

IEEE 标准 754 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。词汇表 [167]定义了与浮点运算相关的术语。

♦ ♦ ♦ 第 2 章

2 IEEE 运算

本章讨论用于二进制浮点算法的 ANSI/IEEE 标准 754-1985（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。Oracle Developer Studio 编译器支持 IEEE 算法的特性。本章讨论以下主题：

2.1 IEEE 算法模型

本节介绍了 IEEE 754-1985 规范。IEEE 标准在 2008 年进行了大幅度修改。

2.1.1 什么是 IEEE 算法？

IEEE 754 规定：

- 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，并至少总共占用 79 位。
- 浮点运算的精确度要求：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性、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的异常的条件。

  五种类型的浮点异常是：无效运算、除以零、溢出、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；向负无穷大舍入（向下）；向正无穷大舍入（向上）以及向 *0* 舍入（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 标准还建议在用户进行异常处理时提供支持。

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

本节介绍了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 1 “IEEE 格式和语言类型”中所示的 IEEE 格式相对应。

表 1 IEEE 格式和语言类型

IEEE 精度	C、C++	Fortran
单精度	float	`REAL` 或 `REAL*4`
双精度	双精度	`DOUBLE PRECISION` 或 `REAL*8`
双精度扩展	long double (x86)	—
四倍精度	long double (SPARC)	`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 1 “IEEE 格式和语言类型”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，如下图所示。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 1 单精度存储格式

表 2 “IEEE 单精度格式位模式表示的值”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* 意味着所指示的字段的值与确定特定单精度格式位模式的值无关。

表 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 ≠ 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）
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（Not-a-Number，非数值）

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 ≤ 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 标准 754 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共提供了 24 位精度。

表 3 “单精度存储格式位模式及其 IEEE 值”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 3　　单精度存储格式位模式及其 IEEE 值

公用名称	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
非数字	`7fc00000`	NaN

NaN（Not a Number, 非数）可以用任何满足 NaN 定义的位模式表示。在表 3 “单精度存储格式位模式及其 IEEE 值”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，如下图所示。

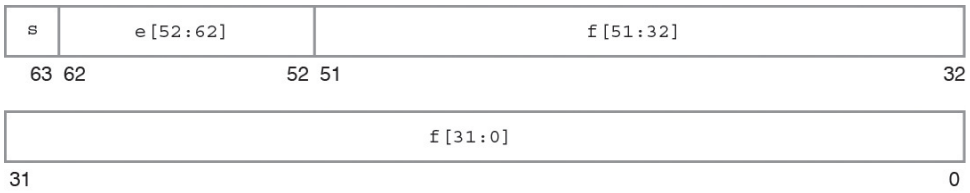
在 SPARC 体系结构中，较高地址的 32 位字包含小数的 32 位最低有效位，而在 x86 体系结构中，较低地址的 32 位字包含小数的 32 位最低有效位。

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。

下图将这两个连续的 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 双精度存储格式



这三个字段中的位模式的值将决定整个位模式所表示的值。

表 4 “IEEE 双精度格式位模式表示的值”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* 意味着所指示的字段的值与确定特定双精度格式位模式的值无关。

表 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	$(-1)^s \times 2^{e-1023} \times 1.f$（正规数）
e = 0; f ≠ 0 （f 中至少有一位不为零）	$(-1)^s \times 2^{-1022} \times 0.f$（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 = 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（Not-a-Number，非数值）

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

此构成的数字称为有效数字。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正规数。

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

表 5 “双精度存储格式位模式及其 IEEE 值”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 5　双精度存储格式位模式及其 IEEE 值

公用名称	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
非数字	`7ff80000 00000000`	NaN

NaN（Not a Number，非数）可以用任何满足 NaN 定义的位模式表示。在表 5 “双精度存储格式位模式及其 IEEE 值”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。

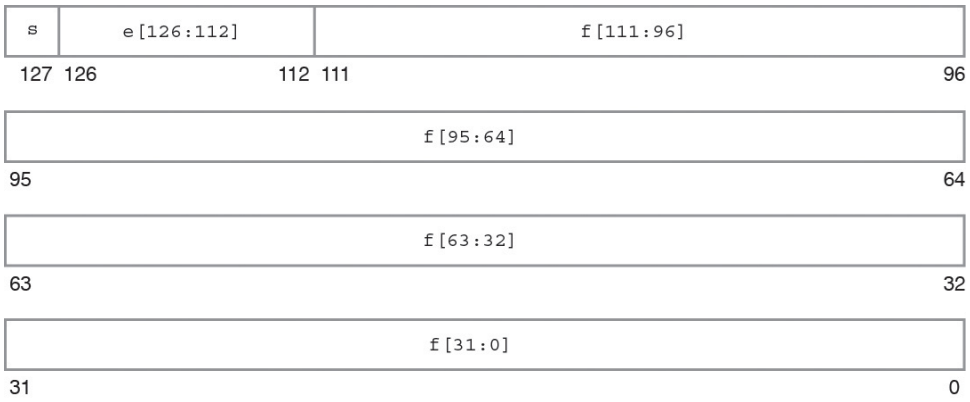
2.2.4 四倍精度格式

浮点环境的四倍精度格式同样符合双精度扩展格式的 IEEE 定义。此格式不在 x86 的 Oracle Developer Studio C/C++ 编译器中。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`；以及 1 位符号 `s`。这三个字段是连续存储的，如下图所示。

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 位字包含小数的 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。

下图将这四个连续的 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 3 四倍精度格式



f、e 和 s 三个字段中的位模式的值将决定整个位模式所表示的值。

表 6 “位模式表示的值”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”，因为指示字段的值与确定特定位模式的值无关。

表 6 位模式表示的值

四倍精度位模式	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	-INF（负无穷大）

四倍精度位模式	值
（f 中的所有位均为零） s = u, e = 32767, f ≠ 0 （f 中至少有一位不为零）	NaN（Not-a-Number，非数值）

表 7 “四倍精度格式位模式”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 7　四倍精度格式位模式

公用名称	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 far-off ffffffff`	3.362103143112093506262677817321752 0e–4932
最小正次正规数	`00000000 00000000 00000000 00000001`	6.475175119438025110924438958227646 6e–4966
+∞	`7fff0000 00000000 00000000 00000000`	+∞
–∞	`ffff0000 00000000 00000000 00000000`	–∞
非数字	`7fff8000 00000000 00000000 00000000`	NaN

在表 7 “四倍精度格式位模式”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。此格式不能作为语言类型用于 Oracle Developer Studio Fortran 或 SPARC 的 C/C++。

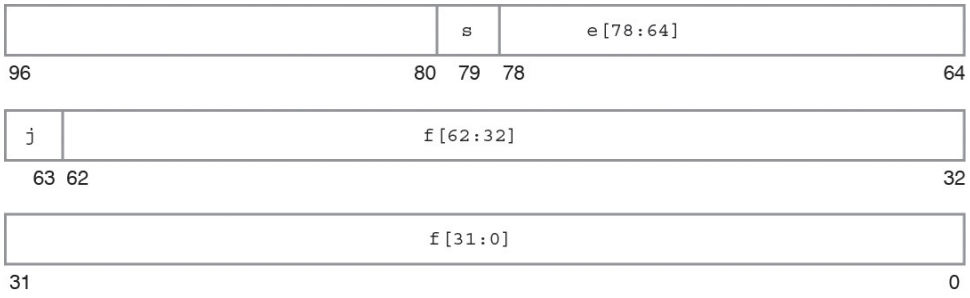
在 x86 体系结构系列中，这些字段连续存储在十个相连地址的 8 位字节中。然而，由于 UNIX System V Application Binary Interface Intel 386 Processor Supplement (Intel ABI) 要求双精度扩展参数，从而占用堆栈中三个相连地址的 32 位字，其中地址最高字的 16 位最高有效位未用，如下图所示。

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，它们符合 Intel ABI 规定，这是至关重要的。

下图将这三个连续的 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 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 8 “位模式表示的值 (x86)”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* 意味着所指示的字段的值与确定特定位模式的值无关。

表 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	$(-1)^S \times 2^{e-16383} \times 1.f$（正规数）
`j` = 0, `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）
`j` = 1, `e` = 0	$(-1)^S \times 2^{-16382} \times 1.f$（伪非正规数）
`j` = 0, `e` = 0, `f` = 0	$(-1)^S \times 0.0$（有符号的零）

双精度扩展位模式 (x86)	值
（f 中的所有位均为零）	
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（quiet NaN，静态 NaN）
j = 1; s = *u*; e = 32767; f = .0*uuu* — *uu* ≠ 0	SNaN（signaling NaN，信号 NaN）
（f 中至少一个 *u* 不为零）	

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，将前导有效数位显式指定为单独的字段 j。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显式表示的，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 754-1985 中，次正规数称为非正规数。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 9　双精度扩展格式位模式及其值 (x86)

公用名称	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`	1.189731495357231765e+4932
最小正正规数	`0001 80000000 00000000`	3.362103143112093506e–4932
最大次正规数	`0000 7fffffff ffffffff`	3.362103143112093506e–4932
最小正次正规数	`0000 00000000 00000001`	3.645199531882474602e–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

公用名称	位模式 (x86)	十进制值
带有最小小数的信号 NaN	`7fff 80000000 00000001`	SNaN

上表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 位十六进制数，它是地址最高的 32 位字的 16 位最低有效位的值（请记住，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。

NaN（Not a Number，非数）可以用任何满足 NaN 定义的位模式表示。上表中的 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 NaN（前导小数位 = 1）还是信号 NaN（前导小数位= 0）。

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、双精度格式和四倍精度格式以及与 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。我们将通过一些图片和示例来解释这些概念。

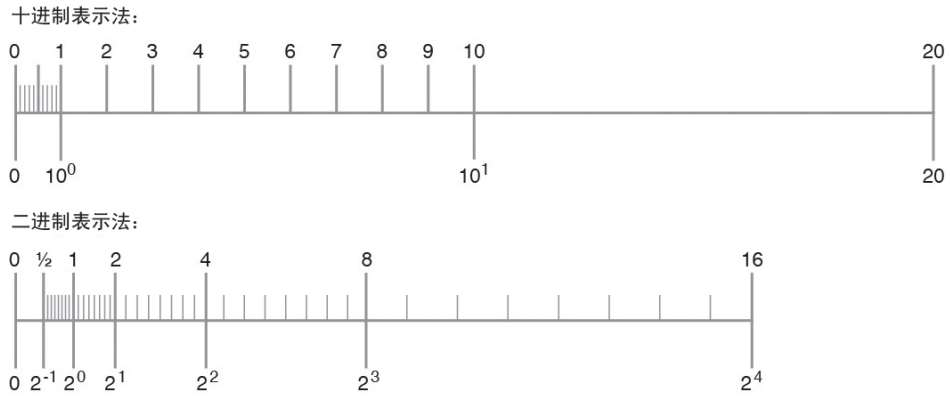
二进制浮点计算的 IEEE 标准指定可以用单精度格式表示数字值集。请记住，我们将这种数字值集解释为二进制浮点数字集。IEEE 单精度格式的有效数字有 23 位，加上隐式前导位，可以得到 24 位（二进制）精度。

用户通过在数轴上标记数字（可以表示为 q 个用有效数字表示的十进制数字）可获得不同的数值集：

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

下图演示的就是这种情况：

图 5 使用数字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，重新组织该问题。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：

$x = x_1.x_2\ x_3... \times 10^n$

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 8.388612 × 10^5 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。

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 10 “存储格式的范围和精度”中进行了全面介绍：

表 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
四倍精度	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 Oracle 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 中的 `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。
- 将内部二进制值作为十进制 ASCII 字符串打印时，系统会执行从基数 2 到基数 10 的转换。

在 Oracle 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）。除了其精确度外，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。

1985 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` 表驱动算法可以对整个范围的单精度、双精度和双精度扩展格式进行正确的舍入，如修改后的 754-2008 要求的那样。

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 缺省设置相同的规则，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` 标志编译，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《Oracle Developer Studio 12.5：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, 参考资料 ，尤其是 Coonen 的论文和 Sterbenz 的书。

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 11 “下溢阈值”显示了单精度、双精度和双精度扩展格式的下溢阈值。

表 11 下溢阈值

目标精度	下溢阈值	
单精度	最小正规数	1.17549435e–38

目标精度	下溢阈值	
	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
四倍精度	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	3.362103143112093506e–4932

正次正规数是介于最小正规数和零之间的数。从最小正规数减去两个与之接近的（正）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。这种方法称为突然下溢，在引入 IEEE 标准之前，这是大多数大型机的缺省设置。

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。

请记住，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。要完整地指定数字，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，其小数中的非零位就越少；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性。
- 当结果非常小时，加法、减法、比较和余数运算都总是准确的。

请记住，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。它不是使值出行问题的最小限度，而是其他相关的误差。算法使用的是较其他系统误差界限更小的次正规数。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。有关更详细的讨论，请参见 James Demmel 编著的《*Underflow and the Reliability of Numerical Software*》和 S. Linnainmaa 编著的《*Combating the Effects of Underflow and Overflow in Determining Real Roots of Polynomials*》。

在算法中使用次正规数，在执行加法或减法运算时，就不会出现未捕获的下溢（这意味着牺牲准确性）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：

computed result = true result + round-off

用来表示误差大小的一种简便尺度称为最后位置单元，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）是该数字的最后位置单元。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位置单元的一半。换而言之，在采用就近舍入模式的 IEEE 算法中，其计算结果如下所示：

0 ≤ |舍入| ≤½ulp

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，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* 最后位置单元。

另外，浮点数的 ulp 还取决于该数字的表示精度。例如，表 12 “四种不同精度的 ulp(1)”显示了上述四个浮点格式的 ulp(1) 值：

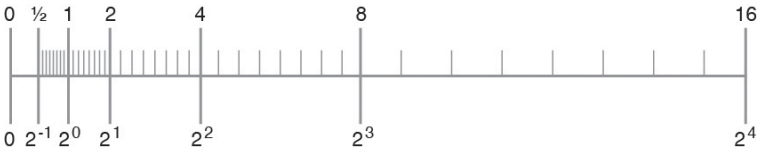
表 12 四种不同精度的 ulp(1)

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	ulp(1) = 2^-112 ~ 1.925930e-34

请记住，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，如下图所示。

图 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 13 “可表示的单精度格式浮点数之间的差距”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 13 可表示的单精度格式浮点数之间的差距

x	nextafter(x, +·)	差距
0.0	1.4012985e–45	1.4012985e–45
1.1754944e–38	1.1754945e–38	1.4012985e–45

x	nextafter(x, +·)	差距
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：引入一个不大于计算结果中可表示相邻数字之间的距离的误差。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：引入一个不大于计算结果中可表示相邻数字之间的距离的误差。

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，可以避免出现以下情况：引入的舍入误差大于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：

- $x \neq y$ 当且仅当 $x - y \neq 0$
- $(x - y) + y \simeq x$，在 x 和 y 二者中较大者的舍入误差内
- $1/(1/x) \simeq x$，当 x 是正规化数时，表示 $1/x \neq 0$，甚至对于最大正规化 x 也是如此

另一种下溢方案是突然下溢，它是将下溢结果刷新为零。只要 `x – y` 出现下溢，突然下溢就会违反第一条和第二条属性。另外，当 `1/x` 出现下溢时，突然下溢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 来比较渐进下溢和突然下溢的误差属性。

渐进下溢：|误差| < ½λ 的 *ulp*

突然下溢：|误差| ≃ λ

用 λ 最后位置单元的一半与 λ 本身存在着显著的差别。

2.3.5 有关渐进下溢与突然下溢的两个示例

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，结果的精确度为舍入的精确度。在突然下溢中，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精确度时，会扩大其计算范围。

第二个示例源于复数商，不适用于缩放：

$$a + i \cdot b = \frac{p + iq}{r + is}$$ 假定 $|r / s| \le 1$

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

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* *q* 和 *r* + *i* *s* 每个值的误差都不大于几个 *ulp*。除了当 *a* 和 *b* 都出现下溢外，这种误差分析是面向下溢的，误差是通过以下值的几个 *ulp* 来界定的：|*a* + *i* *b*|。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。要在突然下溢的情况下找到同样强大、易于分析且有效的计算复数商的算法是不可能的。在突然下溢的情况下，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突然下溢却失败的问题类要比支持使用突然下溢的人想像的多。许多常用的数值技术都无法成功解决此类问题：

- 线性方程求解
- 多项式方程求解
- 数值积分
- 收敛性加速
- 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精确度阈值特别敏感。例如，在单精度计算中，如果计算的某些部分发生了下溢，在使用突然下溢的情况下，会将下溢结果替换为 `0`，则所能保证的精确度只能达到大约 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，缩放算法和检测不准确阈值都比较困难且耗时，即使对于大多数数据而言也是没有必要的。

2.4 IEEE 标准 754-2008

本节讨论了 754-1985 与其后续标准 754-2008 的差异。和其他系统实现工具一样，随着时间的推移，Oracle 也将符合 754-2008 的建议，因为这些建议是在编程语言的标准中提出的。

在 2008 年，IEEE 采用了修订后的 IEEE 浮点运算标准，该标准取代了先前的 754-1985 IEEE 二进制浮点运算标准以及 854-1987 IEEE 与基数无关的浮点运算标准。

从硬件角度来看，新标准向上兼容早期版本；在使用足够软件增强后，现有硬件可以与新标准兼容。然而，要完全实现新标准中的所有建议，要求语言定义能够在数年后有重大的进展；然后才能对这些定义进行商业实现。本《Oracle Developer Studio 数值计算指南》未来的版本将介绍根据修订标准建议的可在 Oracle Developer Studio C、C++ 和 Fortran 编译器中使用的新功能。

以下是 Oracle Developer Studio 12.5 中已完成的一些重要更改：

- 754-2008 规定了 128 位二进制（和十进制）浮点格式；128 位二进制格式对应于 Studio Fortran `REAL*16` 以及 SPARC 上的 Studio C 和 C++ `long double`。
- 754-2008 要求二进制格式及十进制格式与字符序列之间的转换能够正确地舍入，不过对于指数极大或极小的数字，754-1985 允许误差界限可以稍大一些。
- 754-2008 建议在二进制与十六进制字符序列之间进行转换运算。
- 754-2008 需要在 754-1985 中为可选运算的运算；这些运算已存在于 Oracle Developer Studio 编译器中。
- 754-2008 指定了混合乘加运算；这些运算在 Oracle Developer Studio 12.5 支持的最新 SPARC 服务器的硬件中执行。

以下是 Oracle Developer Studio 12.5 中未完成的一些重要更改：

- 754-2008 建议了许多新运算。
- 754-2008 建议提供多个初等超越函数的正确舍入版本。
- 754-2008 建议了表达式求值属性。
- 754-2008 不再规定浮点陷阱处理；而是建议更高级的替代异常处理属性，这些属性可通过与计算机无关的方式使用。

♦ ♦ ♦ 第 3 章

3 数学库

本章介绍了 Oracle Solaris OS 和 Oracle Developer 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) 手册页中还列出了 `libm` 和 `libsunmath` 库的内容。

本章将主题分为以下几节：

3.1 Oracle Solaris 数学库

本节介绍了与 Oracle Solaris 10 OS 捆绑在一起的数学库。这些库是作为共享对象文件提供的，安装在 Oracle Solaris 库的标准位置。

3.1.1 标准数学库

Oracle Solaris 标准数学库 `libm` 中包含基本数学函数以及 Oracle Solaris 操作环境所符合的各种标准所需的支持例程。

Oracle Solaris 10 OS 包括两种版本的 `libm`：`libm.so.1` 和 `libm.so.2`。`libm.so.1` 提供 Oracle Solaris 9 OS 及早期版本支持的标准所需的函数。`libm.so.2` 提供 Oracle Solaris 10 OS（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 Oracle Solaris 9 OS 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 手册页部分进行了说明。本章的剩余部分介绍了 `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文件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 14 “libm 的内容”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。库中还包含函数的一个单精度版本，采用了相同的名称，后跟一个 f；以及一个扩展精度/四倍精度版本，也采用了相同的名称，后跟一个 l。

表 14 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

请注意下面关于表 14 “libm 的内容”的内容：

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在 *libm*(3LIB) 手册页中均以表格形式列出。

3.1.2 向量数学库

库 libmvec 提供了用于计算整个参数向量的常用数学函数的例程。应用程序可以显式调用 libmvec 中的例程，使用 -xvector 标志时，编译器也会调用这些例程。

libmvec 是作为一种主共享对象文件 libmvec.so.1 以及多种辅助共享对象文件（提供部分或全部向量函数的替代版本）实现的。运行使用 libmvec 链接的程序时，运行时链接程序自动选择可在主机平台上提供最佳性能的版本。因此，使用 libmvec 中的函数的程序在不同系统上运行时，产生的结果可能略有不同。

表 15 “libmvec 的内容”列出了 libmvec 中的函数。

表 15 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 Oracle Developer Studio 数学库

本节介绍了 Oracle Developer Studio 编译器所包含的数学库。

Oracle Developer Studio 12.5 的缺省基安装目录在 Oracle Solaris 上是 /opt/developerstudio12.5，在 Linux 上是 /opt/oracle/developerstudio12.5。但是，在安装过程中可以指定不同的基安装目录 *install-dir*。

缺省情况下，在目录 *install-dir*/lib/compilers 及其子目录中安装 Oracle Developer Studio 静态 32 位库。在 *install-dir*/lib/compilers/sparcv9 中（在 SPARC 上）和 *install-dir*/lib/compilers/amd64 中（在 x86 上）安装静态 64 位库。

本数值计算指南还介绍了共享 libm 和 libmvec 库，这些库仅可用于 Oracle Solaris，安装在 /usr/lib（对于 32 位版本）和 /usr/lib/64（对于 64 位版本）中。

对于 Oracle Solaris，math.h 和 sunmath.h 库头文件安装在 /usr/include 中。对于 Linux，math.h 库头文件安装在 *install-dir*/lib/compilers/include/cc 中。

静态归档文件、共享对象和包含文件的 Oracle Developer Studio *install-dir* 的 *subdirectories* 在不同发行版中会有所变化。

3.2.1 Oracle 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 libm.so.2 中提供但 libm.so.1 中未提供的许多函数。libsunmath 同时作为共享对象文件和静态归档文件进行提供。

表 16 “libsunmath 的内容”列出了 libsunmath 中提供，而 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 16 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
按 ?? 缩放的三角函数	asinpi、acospi、atanpi、atan2pi、sinpi、cospi、sincospi、tanpi
使用双精度 ?? 参数约简的三角函数	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_
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_
随机数置乱器	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志	ieee_flags
浮点陷阱处理	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 中一些函数的更快版本。libmopt 仅作为静态归档文件提供。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 样式的异常情况处理方式）不同，libmopt 例程仅支持 C99/IEEE 样式的异常情况处理方式。（请参见附录 E, 标准符合性。）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 的程序必须确保缺省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。

3.3 单、双和扩展/四倍精度

大多数数值函数可以使用单精度、双精度和扩展 (x86) 或四倍精度。表 17 “调用单、双和扩展/四倍精度函数”提供了从不同语言调用各种函数的不同精度版本的示例。

表 17 调用单、双和扩展/四倍精度函数

语言	单精度	双精度	扩展/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 的调用约定不同，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 - Oracle Developer Studio Fortran 不支持扩展双精度。

3.4 IEEE 支持函数

本节介绍了 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 的更多信息，请参阅第 4 章 异常和异常处理。

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

ieee_functions(3m) 和 ieee_sun(3m) 介绍的函数提供了 IEEE 标准要求的功能或其附录中推荐的功能。这些函数是按有效位掩码运算实现的。

表 18 ieee_functions(3m)

函数	说明
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 后面的下一个可表示的数
remainder(x,y)	x 除以 y 所得的余数
scalbn(x,n)	$x \times 2^n$

表 19 ieee_sun(3m)

函数	说明
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
nonstandard_arithmetic(void)	启用非标准模式
standard_arithmetic(void)	启用标准模式
ieee_retrospective(*f)	n/a

remainder(x,y) 是在 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，并且不生成不精确异常。

表 20 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)

IEEE 函数	单精度	双精度	四倍精度
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 21 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) 手册页中介绍的函数提供的。表 22 “IEEE 值：单精度”、表 23 “IEEE 值：双精度”、表 24 “IEEE 值：四倍精度”和表 25 “IEEE 值：双扩展精度 (x86)”中显示了 ieee_values(3m) 函数提供的值的十进制值及其十六进制 IEEE 表示形式。

表 22 IEEE 值：单精度

IEEE 值	十进制值十六进制表示形式	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()
最小次正规数	1.40129846e–45 00000001	r = min_subnormalf(); r = r_min_subnormal()
∞	Infinity 7f800000	r = infinityf(); r = r_infinity()
quiet NaN（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 23 IEEE 值：双精度

IEEE 值	十进制值十六进制表示形式	C、C++ Fortran
最大正规数	1.7976931348623157e+308 7fefffff ffffffff	d = max_normal(); d = d_max_normal()

IEEE 值	十进制值十六进制表示形式	C、C++ Fortran
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()
最小次正规数	4.9406564584124654e–324 00000000 00000001	d = min_subnormal(); d = d_min_subnormal()
∞	Infinity 7ff00000 00000000	d = infinity(); d = d_infinity()
quiet NaN（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 24 IEEE 值：四倍精度

IEEE 值	十进制值十六进制表示形式	C、C++ (SPARC) Fortran（所有）
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()
最小次正规数	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()
quiet NaN（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 25 IEEE 值：双扩展精度 (x86)

IEEE 值	十进制值十六进制表示形式（80 位）	C、C++
最大正规数	1.18973149535723176505e+4932	x = max_normall();

IEEE 值	十进制值十六进制表示形式（80 位）	C、C++
	7ffe ffffffff ffffffff	
最小正正规数	3.3621031431120935062 6e–4932	x = min_normall();
	0001 80000000 00000000	
最大次正规数	3.3621031431120935060 8e–4932	x = max_subnormall();
	0000 7fffffff ffffffff	
最小正次正规数	1.8225997659412373012 6e–4951	x = min_subnormall();
	0000 00000000 00000001	
∞	Infinity	x = infinityl();
	7fff 80000000 00000000	
quiet NaN（静态 NaN）	NaN	x = q
	7fff c0000000 00000000	
信号 NaN	NaN	x = signaling_nanl(0);
	7fff 80000000 00000001	

3.4.3 ieee_flags(3m)

ieee_flags (3m) 是用于以下功能的 Oracle 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、清除或设置已发生异常标志

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 26 “ieee_flags 的参数值”所示：

表 26 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
mode	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 异常标志的更多信息。

如果 *mode* 为 `direction`，则说明指定的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 缺省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表示的数字之间，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表示的数字中间，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，且在数学计算中与将结果截断操作一致。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 27 “ieee_flags 舍入方向的输入值”所示。

表 27 `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`，则说明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、双和扩展精度。缺省的舍入精度为扩展精度；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才能运行这些程序。

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`，则说明指定的操作适用于当前的 IEEE 异常标志。有关使用 `ieee_flags` 检查和控制 IEEE 异常标志的更多信息，请参见第 4 章 异常和异常处理。

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缺省值。
- 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志以及启用陷阱 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）。如果引发异常标志，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（如果引发了异常标志），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）。第 4 章 异常和异常处理讨论了异常、信号和陷阱，并说明如何调查引发异常的原因。

程序可以随时显式调用 `ieee_retrospective`。在 `-f77` 兼容模式下使用 `f95` 编译的 Fortran 程序会在其退出前自动调用 `ieee_retrospective`。在缺省模式下使用 `f95` 编译的 C/C++ 程序和 Fortran 程序不会自动调用 `ieee_retrospective`。

注意，在缺省情况下，`f95` 编译器会对常见异常启用捕获，因此，除非有程序显式禁用捕获或者安装 `SIGFPE` 处理程序，否则在发生此类异常时程序会立即中止。在 `-f77` 兼容模式下，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：

```
Note: IEEE floating-point exception flags raised:
   Inexact; Underflow;
Rounding direction toward zero
IEEE floating-point exception traps enabled:
   overflow;
See the Numerical Computation Guide, 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就清除未处理的异常。

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 IEEE 运算中所介绍的一样，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。如果很多计算都出现下溢，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。如果操作系统中的程序花费了非常多的时间并引发了下溢异常，则渐进下溢可能是引发该问题的原因。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 寄存器中设置了 FTZ（flush to zero, 刷新为零）位；它还在支持 DAZ（denormals are zero, 非正规数为零）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 将硬件重置为使用缺省 IEEE 算法。这两种函数对只提供缺省 IEEE 754 样式的算法的处理器没有影响。SPARC T4 就是一种此类处理器。

3.5 C99 浮点环境函数

本节介绍了 C99 中的 <fenv.h> 浮点环境函数。在 Oracle 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，但它们使用的 C 接口更普通；因为它们是由 C99 定义的，所以它们具有更高的可移植性。

注 - 为保持行为一致，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 异常标志函数

fenv.h 文件为 5 个 IEEE 浮点异常标志中的每一个标志都定义了宏：FE_INEXACT、FE_UNDERFLOW、FE_OVERFLOW、FE_DIVBYZERO 和 FE_INVALID。此外，将宏 FE_ALL_EXCEPT 定义为所有 5 个标志宏进行按位“或”运算。在以下说明中，excepts 参数可以是 5 个标志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 定义了下表中的异常标志函数：

表 28 C99 标准异常标志函数

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

feclearexcept 函数清除指定的标志。fetestexcept 函数返回与设置的 excepts 参数指定的标志子集对应的宏值的按位“或”运算结果。例如，如果当前设置的标志只有不精确、下溢和除以零，则以下指令将 i 设置为 FE_DIVBYZERO。

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

如果启用任何指定异常的陷阱，则 feraiseexcept 函数会导致一个陷阱。否则，它只设置相应的标志。有关异常陷阱的更多信息，请参见第 4 章 异常和异常处理。

fegetexceptflag 和 fesetexceptflag 函数提供了一种简便的方法暂时保存某些标志的状态，并在以后恢复它们。特别地，fesetexceptflag 函数并不导致陷阱；它只恢复指定标志的值。

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（向正无穷大舍入）、FE_DOWNWARD（向负无穷大舍入）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 提供了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 文件定义了数据类型 fenv_t，它表示整个浮点环境，包括异常标志、舍入控制模式、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 29 libm 浮点环境函数

函数	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”异常标志

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。后者表示缺省环境，即清除所有异常标志、舍入到最接近的值以及在基于 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）以及禁用非标准的模式。

feholdexcept 函数会保存当前的环境，然后清除所有异常标志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志的异常。如果恢复的环境为其中的任何异常启用了陷阱，则会发生陷阱；否则，设置这些标志。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，如以下代码样例所示：

```
#include <fenv.h>

void myfunc(...) {
    fenv_t env;
```

```
    /* save the environment, clear flags, and disable traps */
    feholdexcept(&env);
    /* do a computation that may incur exceptions */
    ...
    /* check for spurious exceptions */
    if (fetestexcept(...)) {
        /* handle them appropriately and clear their flags */
        ...
        feclearexcept(...);
    }
    /* restore the environment and raise relevant exceptions */
    feupdateenv(&env);
}
```

fex_merge_flags 函数对当前环境和保存环境中的异常标志执行逻辑 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志的信息。有关显示 fex_merge_flags 用法的示例，请参见附录 A, 示例。

3.6 libm 和 libsunmath 的实现功能

本节介绍了 libm 和 libsunmath 的实现功能：

- 使用无限精确 ?? 的参数约简以及按 ?? 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表驱动算法和多项式/有理数近似算法的组合实现的。所发布的这些算法不断变化，以实现更高的性能或精确度。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 中常见初等函数及 libsunmath 中常见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位置单元 (unit in last place, *ulp*) 内。在基于 SPARC 的系统上，libsunmath 中的常见四倍精度初等函数将精确结果传送到一个 *ulp* 内，但是 expm1l 和 log1pl 函数除外，它们将精确结果传送到两个 *ulps* 内。（常见函数包括指数函数、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度，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 三角函数的参数约简

弧度参数在 [–??/4,??/4] 范围之外的三角函数通常可通过将参数约简到指定范围（减去 ??/2 的整数倍）来进行计算。

因为 ?? 不是计算机可表示的数字，所以对其求近似值。三角函数的最终计算中的误差取决于参数约简中造成的舍入误差（使用近似的 ?? 值和舍入值）以及计算使用约简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约简误差；而对于较大的参数，参数约简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：计算机可表示的非常大的数字是由大于 ?? 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。假设参数始终不断进行约简，实际上很难发现使用 ?? 近似值完成的参数约简这一情况，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 ?? 进行参数约简。将值 2/?? 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约简过程中使用。

sinpi、cospi 和 tanpi 函数组（请参见表 16 “libsunmath 的内容”）使用 ?? 来对输入参数进行缩放，以避免由于范围约简而产生的误差。

3.6.3 数据转换例程

在 libm 和 libsunmath 中，有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生成器系列。

- *lcrans*(3m) 手册页中介绍的函数基于线性同余随机数生成器。
- *mwcrans*(3m) 手册页中介绍的函数基于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) 手册页中介绍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 30 “单值随机数生成器的区间”所示的范围内。

表 30 单值随机数生成器的区间

函数	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
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, 示例给出了一些示例，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 编著的《*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

本章介绍了 IEEE 浮点异常以及如何检测、查找和处理它们。本章还列出了由 IEEE 754 定义的异常及其缺省结果，并介绍支持状态标志、捕获和异常处理的浮点环境的功能。本章将这些主题分为以下几节：

4.1 异常处理目标

在基于 SPARC 和 x86 的系统上，由 Oracle Developer Studio 编译器和 Oracle Solaris OS 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 754 标准（IEEE 854，第 18 页）对这些功能的某个目标进行了如下介绍：

> ... 为用户最大程度地减少由于异常条件产生的复杂问题。运算系统设计为尽可能长久地进行持续计算，使用合理的缺省响应处理异常情况（包括设置适当的标志）。

为了实现此目标，标准指定了异常运算的缺省结果，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自陷的方法（即，中断正常的控制流）。例如，程序可以为异常运算提供替代结果并继续执行，从而提供一个用来以适当方式处理异常的陷阱处理程序。以下各节详细介绍了浮点环境的功能如何支持这些异常。

4.2 什么是异常？

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术运算没有生成可普遍接受的结果时，就产生了运算异常。原子和可接受的含义随时间和使用场合而异。（请参见 W. Kahan 编著的《*Handling Arithmetic Exceptions*》。）

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。在出现此类异常时，系统会以下列两种方式之一进行响应：

- 如果该异常的陷阱处于禁用状态（缺省情况），系统将记录异常发生这一事实，并使用 IEEE 754 针对异常运算指定的缺省结果继续执行该程序。
- 如果该异常的陷阱处于启用状态，系统将生成 `SIGFPE` 信号。如果程序安装了 `SIGFPE` 信号处理程序，系统会将控制转交给该处理程序；否则，该程序将中止。

IEEE 754 定义了五个基本类型的浮点异常：无效运算、除以零、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) 手册页介绍了一种仅适用于在出现常见异常时自陷的简便方法。另外两种异常（下溢和不精确）更常见。实际上，大多数浮点运算都会导致不精确异常。这些异常通常（尽管不总是）可以安全地忽略。缺省情况下，Oracle Developer Studio 12.5 C、C++ 和 `f77` 编译器将禁用所有 IEEE 陷阱。`f95` 编译器在缺省情况下将对常见异常启用陷阱。可通过使用 `f95 -ftrap=none` 进行编译来恢复符合 754 标准。

表 31 “IEEE 浮点异常”简要介绍了 IEEE 标准 754 中的信息。它描述了五个浮点异常，以及在出现这些异常时 IEEE 运算环境的缺省响应。

表 31　IEEE 浮点异常

IEEE 异常	出现异常的原因	示例	在陷阱被禁用时出现的缺省结果
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 ■ 信号 NaN 操作数的任何运算 ■ 无序比较（请参见注释 1） ■ 无效转换（请参见注释 2）	无噪声 `NaN`
除以零	针对有限操作数执行运算时生成精确的无穷大结果。	■ 对于有限的非零 `x`，x / 0 ■ 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的多（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)	取决于舍入模式 (RM)，以及中间结果的符号。请参见第 4.2.1 节 “表 4-1 的注释” [57]中的第 4 项。
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的多（请参见注释 3）。	■ 双精度： 　■ `nextafter`(min_normal,-·) 　■ `nextafter`(min_subnormal,-·) 　■ `DBL_MIN` §3.0 　■ `exp`(-708.5)	低于正常值或零

IEEE 异常	出现异常的原因	示例	在陷阱被禁用时出现的缺省结果
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。）	■ 单精度： ■ （浮点）DBL_MIN ■ nextafterf(FLT_MIN, -·) ■ expf(-87.4) ■ 2.0 / 3.0 ■ （浮点）1.12345678 ■ log(1.1) ■ DBL_MAX + DBL_MAX，当没有溢出陷阱时	该运算的结果（舍入、溢出或下溢）

4.2.1 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（not a number，非数）。

   每个 NaN 可与任何值（包括它本身）进行“无序”比较。下表显示在关系为无序时，由哪一种谓词导致无效运算异常。

数学谓词	C、C++ 谓词	Fortran 谓词	无效表达式（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 IEEE 运算。
4. 下表列出了针对溢出禁用陷阱时的缺省结果。以下结果取决于舍入模式和中间结果的符号。

舍入模式	正	负
最近	+∞	-∞
零	+∞	-max
向下	+max	-∞
向上	+∞	-max

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。

能够在单个运算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，则溢出和下溢陷阱的优先级高于不精确陷阱；在基于 x86 的系统上，溢出、下溢和不精确陷阱的优先级都高于非正规操作数陷阱。

4.3 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了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：当前和已发生。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”）到已发生异常标志中，从而记录自该程序开始执行或者自该程序上次清除已发生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，会设置与导致该陷阱的异常相对应的当前异常标志，但是不会更改已发生标志。当前异常标志和已发生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) 为已发生异常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 寄存器包含记录由这些指令引发的已发生异常的标志。

4.3.1 `ieee_flags`(3m)

`ieee_flags` 为类似于 Oracle Developer Studio C、C++ 和 Fortran 的 IEE 754 异常标志提供了一个接口。然而，只能在 Oracle Solaris OS 上使用此接口。对移植范围更广的 C 和 C++ 程序使用第 4.3.2 节 “C99 异常标志函数” [60]。

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、设置或清除已发生异常状态标志。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* 中返回的字符串为以下之一：

- "not available" – 如果异常信息不可用
- "" （空字符串） – 如果没有已发生异常，或者对于 x86，非正规操作数是唯一的已发生异常
- 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，在以下 Fortran 调用中，如果出现了除以零异常，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：

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

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。此值是所有已发生异常标志的按位“或”，其中的每个标志都用一位表示，如表 32 “异常位”中所示。与每个异常相对应的位的位置由 fp_exception_type 值（定义在 sys/ieeefp.h 文件中）给出。（请注意，这些位的位置与计算机有关且不必连续。）

表 32 异常位

异常	位的位置	已发生异常位
invalid	fp_invalid	i & (1 << fp_invalid)
overflow	fp_overflow	i & (1 << fp_overflow)
division	fp_division	i & (1 << fp_division)
underflow	fp_underflow	i & (1 << fp_underflow)
inexact	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);
```

4.3.2 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。有关更多信息，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。

要清除这五个异常标志，请使用以下命令：

```
feclearexcept(FE_ALL_EXCEPT);
```

要测试引发的是无效运算标志还是除以零标志，请使用以下指令：

```
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 */
```

`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_INEXACT);
```

4.4 查找异常

找出异常发生的位置的一种方法是在整个程序中各个点处测试异常标志。但是要用这种方法精确地隔离异常需要很多测试并且会产生巨大成本。

用来更方便的确定异常的发生位置的方法就是启用它的陷阱。当发生启用了陷阱的异常时，操作系统将通过发送 SIGFPE 信号来通知该程序。请参见 *signal*(5) 手册页。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，系统会设置相应的标志，并使用表 31 “IEEE 浮点异常”中指定的缺省结果继续执行程序，但不会发送任何信号。

4.4.1 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。请记住，要使用 dbx 的源代码级调试功能，程序应使用 -g 标志进行编译。有关更多信息，请参阅《Oracle Developer Studio 12.5：使用 dbx 调试程序》。

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在运算无效时自陷功能，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，通过用可启用运算无效时自陷的启动例程进行链接或者手动启用该陷阱来确定。

4.4.1.1 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 会导致对无效运算异常启用自陷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
Reading ld.so.1
Reading libm.so.2
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2773)
signal FPE (invalid floating point operation) in __sqrt at 0x7fa9839c
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
Current function is sqrtm1
    5   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
Reading ld.so.1
Reading libm.so.2
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2931)
signal FPE (invalid floating point operation) in __sqrt at 0x7fa9839c
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
(dbx) dis __sqrt+0x50/4
dbx: warning: unknown language, 'c' assumed
0x7fa98390: __sqrt+0x0050:      neg      %o4, %o1
0x7fa98394: __sqrt+0x0054:      srlx     %o2, 63, %l6
0x7fa98398: __sqrt+0x0058:      fsqrtd   %f0, %f2
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
(dbx) print $f0f1
$f0f1 = -4.2
(dbx) print $f2f3
$f2f3 = -NaN.0
(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。以下输出可能是典型的结果。

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 18566)
signal FPE (invalid floating point operation) in sqrtm1 at 0x80509ab
0x080509ab: sqrtm1+0x001b:      fstpl    0xffffffe0(%ebp)
(dbx) dis sqrtm1+0x16/5
dbx: warning: unknown language, 'c' assumed
0x080509a6: sqrtm1+0x0016:      fsqrt
0x080509a8: sqrtm1+0x0018:      addl     $0x00000008,%esp
0x080509ab: sqrtm1+0x001b:      fstpl    0xffffffe0(%ebp)
0x080509ae: sqrtm1+0x001e:      fwait
```

```
0x080509af: sqrtm1+0x001f:      movsd    0xffffffe0(%ebp),%xmm0
(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。

4.4.1.2 在不重新编译的情况下启用陷阱

上面的示例通过用 `-ftrap` 标志重新编译主要的子程序来对无效运算异常启用自陷。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自陷。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陷阱。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陷阱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，只有在该程序至少执行了一个浮点指令之后，才能手动启用自陷。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，对无效运算异常启用自陷，命令 `dbx` 停止接收 `SIGFPE` 信号，然后继续执行。在基于 SPARC 的系统上，步骤如下所示。请注意使用 `assign` 命令修改 `%fsr` 以对无效运算异常启用自陷：

```
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，从而对这些指令引发的异常启用捕获。

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陷阱。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对象文件和库，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);
}
```

编译该文件，以便创建一个对象文件并用该对象文件链接初始程序：

```
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文件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
```

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文件的列表中：

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

```
Arithmetic Exception
```

有关创建和预装共享对象文件的更多信息，请参见《Oracle Solaris 11.3 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，在运行时链接程序将控制传递给属于主可执行文件的启动代码之前，无论共享对象文件中的初始化例程是预装的还是显式链接的，它们都由运行时链接程序来执行。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缺省模式，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，记住以下几点：

- 任何由共享对象文件中的初始化例程建立的浮点控制模式（如在上例中启用的陷阱）将在该程序的整个执行过程中保持有效（除非它们被覆盖）。
- 任何通过编译器标志选择的非缺省模式将覆盖由共享对象文件中的初始化例程建立的模式（但是，通过编译器标志选择的缺省模式将不覆盖以前建立的模式）。
- 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` 标志选择的任何非缺省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缺省设置。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文件来启用捕获或者更改任何其他缺省浮点模式，必须覆盖 `__fpstart` 例程，以便它不重置缺省浮点模式。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 Oracle Solaris 10 OS 或更高版本上。

```
#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 0 instead if the host
       platform does not support SSE2 instructions */
    _sse_hw = 1;
}
```

4.4.2 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功能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中止。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权转交给该处理程序，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），然后中止或继续执行。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，如下一节所述。

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中止该程序或者建立 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 那样启用捕获功能。切记，只有启用了浮点异常的陷阱时，浮点异常才触发 SIGFPE 信号。

4.4.2.1 **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，该类型定义在 floatingpoint.h 中。

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
exception	char *	invalid、division、overflow, underflow、inexact、 all、common
handler	sigfpe_handler_type	用户定义的例程 SIGFPE_DEFAULT SIGFPE_IGNORE

输入参数	C 或 C++ 类型	可能的值
		SIGFPE_ABORT

当请求的操作为 "set" 时，ieee_handler 建立由 *handler* 指定的名为 exception 的异常的处理函数。处理函数可能为 SIGFPE_DEFAULT 或 SIGFPE_IGNORE，两者都选择缺省 IEEE 行为 SIGFPE_ABORT 或者用户提供的子例程的地址，当发生任一指定异常时，选择此缺省行为会导致程序中止，选择此地址会导致该子例程被调用（通过 sigaction(2) 手册页中关于装有 SA_SIGINFO 标志设置的信号处理程序中所介绍的参数来完成）。如果处理程序为 SIGFPE_DEFAULT 或 SIGFPE_IGNORE，发生指定异常时，ieee_handler 也会禁用捕获；对于任何其他处理程序，ieee_handler 会启用捕获。

在 x86 平台上，每当异常的陷阱处于启用状态并且引发相应标志时，浮点硬件都会启用陷阱。因此，要避免伪陷阱，程序应在调用 ieee_handler 来启用捕获之前清除每个指定的 *exception* 的标志。

当请求的 *action* 是 "clear" 时，ieee_handler 将撤销处理函数当前正针对指定 *exception* 安装的异常并禁用其陷阱。这与针对 SIGFPE_DEFAULT 执行 "set" 操作相同。当 *action* 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 *action* 是 "get" 时，ieee_handler 返回当前为指定 *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除以零时中止：

```
#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 代码段恢复针对所有异常的 IEEE 缺省异常处理：

```
#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'
```

4.4.2.2 报告来自信号处理程序的异常

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计算机的整数和浮点寄存器的内容。该处理程序会检查这些信息，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，请按如下方式声明该处理程序。本章的其余部分提供 C 样例代码；要查看 Fortran 中 SIGFPE 处理程序的示例，请参见附录 A, 示例。

```
#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。信号数量定义在 sys/signal.h 中；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 /usr/include/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 33 “运算异常的类型”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 33 运算异常的类型

SIGFPE 类型	IEEE 类型
FPE_INTDIV	n/a
FPE_INTOVF	n/a
FPE_FLTRES	不精确
FPE_FLTDIV	除
FPE_FLTUND	下溢
FPE_FLTINV	无效
FPE_FLTOVF	溢出

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。整数除以零 (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，会在缺省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 /usr/include/sys/siginfo.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。例 1 “SIGFPE 处理程序”显示了此类处理程序。

例 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) */
    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，操作系统会保存已发生异常标志的副本，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，否则已发生标志将在该处理程序返回之后丢失。因此，在使用 `-xarch=386` 进行编译时，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)
```

但在缺省情况下，或在使用 `-xarch=sse2` 进行编译时，由于 PC 永远不会使循环指令通过，因此测试程序将进行循环。对于 Oracle Developer Studio12.5 ，添加一行代码就足以使 PC 进行递增：

```
uap -> UC_mcontext.gregs[REG_PC= +=5;
```

上述代码仅对 `-xarch=sse2` 有效，并且只有在 SSE2 指令的长度恰好为五个字节时。完全通用的 SSE2 解决方案涉及对优化代码进行解码，以查找下一指令的开头。改用 `fex_set_handling`。

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 缺省结果：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）的缺省结果。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 4.5 节 “处理异常” [77]。

4.4.3 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陷阱（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回顾诊断消息记录到选定文件中。

4.4.3.1 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。它必须是表 34 “fex_set_handling 的异常代码”的第一列中列出的值的按位“或”。（这些值在 fenv.h 中定义。）

表 34 fex_set_handling 的异常代码

值	异常
FEX_INEXACT	不精确结果
FEX_UNDERFLOW	下溢
FEX_OVERFLOW	溢出
FEX_DIVBYZERO	除以零
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	信号传输 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、除以零和所有的无效运算）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。有五种可能的模式：

- FEX_NONSTOP 模式提供 IEEE 754 缺省的不间断行为。这等效于使异常的陷阱保持禁用状态。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缺省处理并为其他异常保留 IEEE 缺省处理。

- FEX_NOHANDLER 模式等效于在不提供处理程序的情况下启用异常的陷阱。在发生异常时，如果以前安装了 SIGFPE 处理程序，系统会将控制权转交给该处理程序，否则系统会中止操作。
- 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。当发生其中任一异常时，系统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，系统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 34 “fex_set_handling 的异常代码”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定的异常建立的，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。要中止 0/0 异常，请使用以下命令：

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

要针对溢出和除以零安装 SIGFPE 处理程序，请使用以下命令：

```
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。有关更多信息，请参见 *fex_set_handling*(3m) 手册页。

例 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;
    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;
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);
```

上例中的处理程序会报告所发生异常的类型、导致该异常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。

4.4.3.2 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、导致该异常的指令的地址、将要处理该异常的方式以及类似于调试器所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，具有相同的指令地址和栈跟踪）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);
```

以下代码示例将以下两个功能组合在一起：对回顾诊断消息的记录与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中止该程序，并获取可显示每个异常发生位置的诊断输出结果。

例 3 将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)
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 = ftrap_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：

```
env FTRAP=invalid LD_PRELOAD=./init.so a.out
Floating point invalid operation (sqrt) at 0x7fa98398 __sqrt, abort
  0x7fa9839c  __sqrt
  0x00010880  sqrtm1
  0x000108ec  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，要从共享对象文件中的初始化例程启用捕获，必须覆盖标准 `__fpstart` 例程。

附录 A, 示例提供了显示典型日志输出的更多示例。有关一般信息，请参见 *fex_set_log*(3m) 手册页。

4.5 处理异常

在以前，大多数数值软件在编写时都未考虑异常，许多编程人员经常遇到异常导致程序立即中止的环境。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（如除以零和无效运算）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。尝试避免所有异常可能需要许多防御性测试和分支，并且还会产生巨大成本有关更多信息，请参见 Demmel 和 Li 合著的《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 算法的缺省异常响应、状态标志和可选的捕获功能旨在提供第三个替代功能：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拦截和处理它们。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拦截它们的处理程序。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陷阱处理程序。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。

请记住，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，该结构中包含计算机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 34 “fex_set_handling 的异常代码”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缺省结果（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（保存已发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已发生标志的设置。（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缺省的未捕获结果和标志，就好像异常未被捕获一样。）

下一节举例说明了如何替换下溢或溢出运算的缩放结果。有关更多示例，请参见附录 A, 示例。

4.5.1 替换 IEEE 捕获的下溢/溢出结果

IEEE 标准建议：在捕获到下溢和溢出时，系统应当为陷阱处理程序提供一种方法来替换已封装指数的结果，即，该值与下溢或溢出运算的舍入结果值相一致（指数封装在其常用范围的末端这一情况除外），从而有效地按 2 的幂缩放结果。在选择比例因子时，使下溢或溢出结果尽可能接近指数范围的中间位置处的值，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封装的指数。这种下溢/溢出“计数模式”可用于在计算中产生准确结果，不然，结果将超出可用浮点格式的范围。有关更多信息，请参见 P.Sterbenz 编著的《*Floating-Point Computation*》。

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，为了替换已封装指数的结果，下溢/溢出处理程序必须对指令解码、检查操作数寄存器并生成缩放结果本身。以下示例显示了一个执行这三个步骤的处理程序。

例 4 对基于 SPARC 的系统替换 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>

#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);
        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
        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) {
            /* 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;
            }
            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如下所示：

```
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，浮点硬件会提供已封装指数的结果。但是，当出现捕获到的有关浮点存储指令的下溢或溢出时，如果该存储指令属于存储并弹出指令，则该硬件将在不完成存储且不弹出栈的情况下进行捕获。因此，为了实现计数模式，当出现有关存储指令的陷阱时，下溢/溢出处理程序必须生成缩放结果并修复栈。例 5 “对基于 x86 的系统替换 IEEE 捕获的下溢/溢出处理程序结果”说明了此类处理程序。

例 5　　对基于 x86 的系统替换 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

*/
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:
    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;
    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 的系统上并使用 -xarch=386 进行编译一样，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)
```

注 - 如果 -xarch=sse2，此程序将进行循环。对于 -xarch=sse2，应完全重写此程序。

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 已封装指数的结果，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 参数将设置为 fex_other。（有关定义，请参见 fenv.h 文件。）而且，x86 硬件自动提供已封装指数的结果，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 已封装指数的结果。当捕获到其中的任一异常时，该处理程序会进行如下设置：

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

以指出应当提供已封装指数的结果。下面的示例显示了此类处理程序：

```
#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
```

上述程序示例在 SPARC、`-xarch=sse2` 的 x86 以及 `-xarch=386` 的 x86 上有效。

♦ ♦ ♦ 第 5 章

5 编译器代码生成

本章介绍了专门针对数值计算的 Oracle Developer Studio 12.5 编译器的编译器代码生成功能。本章由以下主题组成：

5.1 支持的操作系统、硬件和内存模型

Oracle Developer Studio 12.5 支持 Oracle Solaris 10 Update 10 及更高版本和 Oracle Solaris 11，并支持 Oracle 和 Red Hat Enterprise Linux 发行版 5 和 6。

Oracle Developer Studio 支持的硬件与对应的 Oracle Solaris 发行版相同。对于 SPARC，Oracle Solaris 10 和 11 仅为支持 64 位地址空间内存模型的 SPARC 处理器提供支持。对于 x86，Oracle Solaris 11 仅为支持 64 位地址空间内存模型的 x86 处理器提供支持。Oracle Solaris 10 还为众多仅支持 32 位地址空间内存模型的 x86 处理器提供支持。

所有 64 位处理器都可以执行为 32 位或 64 位地址空间编译的程序。Oracle Solaris 10 和 11 支持在 64 位操作系统上执行 32 位程序。

32 位和 64 位寻址在编译时使用 `-m32` 和 `-m64` 命令行选项进行选择。这会影响 C 整数和指针变量的大小。操作系统提供了一些 32 位和 64 位运行时库，编译器则为特定语言提供附加库。

需要 64 位地址空间的程序必须使用 `-m64` 进行编译。许多程序可以使用任一地址模型编译并正确运行，因此，需要明确哪种模型更快也就理所当然了。在使用 `-m64` 时，需要与内存之间来回移动大量整数和指针数据的 C 程序速度可能只有一半。但是，64 位应用程序二进制接口 (ABI) 比 32 位 ABI 的寄存器数量要多，因此需要的内存移动较少。对于大多数程序，性能差别并不显著；但是对于特定程序，为了确保性能，最好采用两种方式编译并测试正确性和性能。

注 - 在 Sun Studio 11 和较早的发行版中，内存模型并非 -m32 或 -m64 这样的显式选项，而是内置到 -xarch 选项中，采用不同的名称对应于内存模型。在 Sun Studio 12 中，内存模型选项和架构选项已分离开。

5.2 代码生成选项

Oracle Developer Studio 12.5 支持多种不同的 SPARC 和 x86 处理器芯片。其中每个处理器芯片都有一个 -xtarget= 命令行编译器选项。-xtarget= 选项指定实现的指令集、实现该指令集的特定处理器芯片以及不同高速缓存的大小。-xtarget 对于以下内容起到宏的作用：

-xarch=*architecture*	编译器在优化代码时，使用 -xarch= 选项来确定在硬件中实现哪些指令，从而适合代码生成。
-xchip=*chip*	编译器在优化代码时，使用 -xchip= 选项来确定哪个特定芯片合适，从而确定如何调度指令。
-xcache=*cache-size*	编译器在优化代码时，使用 -xcache= 选项来确定如何阻止循环以尽可能减少内存流量。

通过针对特定目标进行优化，您可以获得最适合该目标的代码，不过，对于具有不同指令集和调度约束的不同目标，这可能会非常糟糕。当一个可执行文件需要在多个不同的目标系统上运行时，缺省的通用代码生成效果最佳，也可以使用选项 -xtarget=generic 显式选择。

一些 -xtarget= 名称的意义不太明显。要指定特定目标，您可以将 -native 选项用于 Oracle Developer Studio 编译器，这将自动为编译的目标系统选择 -xtarget=。在 SPARC 系统上，fpversion 命令将显示类似的信息。有关更多信息，请参见附录 B, SPARC 行为和实现。

5.3 缺省地址模型和代码生成

在 Oracle Developer Studio 12.5 和以前的发行版中，对于 32 位 Oracle Solaris OS，缺省地址空间模型为 -m32。但是，在 Linux 上，对于 32 位硬件，缺省值为 -m32；对于 64 位硬件，缺省值为 -m64。

对于 SPARC，缺省值为 -xarch=sparc。

对于 x86，缺省值为 -xarch=sse2。

注 - 在以前的 Oracle Developer Studio 发行版中，x86 缺省值对于 -m32 为 -xarch-386；对于 -m64 为 -xarch=sse2。

在新的 x86 中，缺省值 `-m32 -xarch=sse2` 与以前的缺省值 `-m32 -xarch=386` 实现相同的 ABI。浮点操作数和结果在 x87 浮点寄存器中传递。但是，以下单精度和双精度浮点运算通常在 `sse2` 寄存器中执行。

- `+`
- `-`
- `*`
- `/`
- `sqrt`
- `convert`

x87 寄存器仍用于长双精度型运算以及硬件初等超越函数求值。

这意味着 x86 缺省编译的浮点计算结果，相对于以前的 Studio 发行版，即使在相同的硬件和操作系统上，也可能会与 Oracle Developer Studio 12.5 略有不同。

5.4 编译选项

Oracle Developer Studio 12.5 编译器接受多个影响代码生成的选项。以下列表突出显示了并非对所有程序都有效的特定代码生成选项。在有大量程序员花费数年时间共同编写的大型程序中，通常会出现这种情况，没有人能够肯定哪个代码转换会/不会对程序的逻辑造成不利影响，因此必须谨慎使用以下选项。

`-fast`
: 用于许多通常非常有用的不同转换的宏。相比其所有组成部分，`-fast` 更易于记住。`-fast` 的定义在不同发行版中有所不同。并非其所有转换对于所有程序都是合法的。如果使用了 `-fast`，则其后跟随的附加选项可能会消除它的部分效果，例如，使用 `-fast -fsimple=0 -fns=no -xvector=no` 会禁用下面讨论的三个选项。使用 `-dryrun` 进行编译是一种很好的方法，可以查看特定编译器的命令行选项实际启用了哪些选项。

`-fsimple`
: 允许有关浮点模式、异常及舍入的特定简化假设，这些假设并非在所有程序上均成立。编译器使用这些假设来调整不同的值变化转换，这在不同的发行版中有所不同。`-fsimple=0` 是缺省且最安全的选项。`-fsimple=1` 对于许多程序而言是安全的，而 `-fsimple=2` 对于许多程序则有风险。只通过少量输入来简单测试程序执行并不足以验证 `-fsimple=2` 的适用性。对于某些输入数据而言，其效果可能会有益，对于另一些则不然。

`-fns`
: 不影响代码生成，但会导致程序在启用非标准下溢模式的情况下开始执行。这与 IEEE 标准相反，因此可能会得到非标准结果，包括无效结果和无限循环。不论如何处理下溢，许多程序都会以同样方式运行。如果运行时硬件对于标准下溢而言运行速度较慢，则这些

程序使用 -fns 选项时可能会运行得更快。最新的 SPARC 服务器对于非标准模式没有性能优势。

-ftrap=common　不影响代码生成，但会导致程序在对下溢、被零除和无效 IEEE 异常启用了陷阱的情况下执行；这与 IEEE 标准相反，可能会导致依赖于连续 IEEE 异常处理的程序提前终止。-ftrap=none 是 C、C++ 和 F77 的缺省值，但不是 F95 的缺省值。

-fnonstd　用于 -fns 和 -ftrap=common 的宏。

-xvector　与 -xvector=lib 一起使用时，用于启用向量数学库转换；与 -xvector=simd 一起使用时，用于启用 SIMD 转换。-xvector=lib 将更改数值结果，因为使用了通用初等超越函数的略微不同的、面向向量的 libmvec 实现，而不是 libm 版本。这些向量版本假定采用的是缺省舍入。

-xreduction　在使用 -xautopar 或 -xopenmp 启用了并行化时，允许并行化范围更广的程序循环。归约操作类似于向量汇总操作或者计算两个向量的点积的操作；在精确运算中，汇总操作可以按任意顺序进行，但在有限精度浮点运算中会生成略微不同的结果。实际上，累积的顺序甚至可能是不确定的，因此在使用相同程序、相同数据和相同硬件的不同运行中，结果可能会略有不同。

有关更多信息，请参见各编译器的手册页和用户指南。

5.5 可再现结果

如《数值计算指南》的 D.11 节中所述，即使符合标准的 IEEE 运算实现也可能会生成不同的结果。通常这些结果都同样有效，但是证明这种情况一般会很繁琐或非常困难。出于多种原因，最好是牺牲一些性能以减少验证结果所需的误差分析数量。输出中的细微差异或重大差异是否同样有效并不明确，而这些差异是由用户程序错误、编译器优化错误还是硬件错误造成的也不明确。

导致 IEEE 浮点运算的结果不同有几种主要根本原因。下面列出了这些原因并描述了一些方法，以便减少在不同发行版之间以及支持的 Oracle Developer Studio 平台之间出现的这种无故变化。请注意，每种方法在增加可再现性的同时，也有可能会降低性能。有时候性能下降会非常明显。

5.5.1 超越函数

大部分常用数学库函数（例如指数、对数和三角函数）按编程语言进行标准化，相比合理运算或者代数函数（例如平方根 sqrt（）），正确舍入的成本非常高昂。接近正确的

舍入函数适用于大多数用途，并且速度要快得多。但是，最快的接近正确舍入函数在不同的平台上也不同。

- 对于由应用程序使用的函数，请使用可移植代码。此类代码的一个来源是免费分发的数学库 `fdlibm`。这个库可从 Netlib 软件系统信息库获取。
- 避免使用 `-xvector` 选项。超越函数的向量化版本针对特定平台进行了优化，在不同平台上会得出略微不同的结果。
- 避免使用 x86 硬件超越指令。即使这些指令具有小得不能再小的误差界限，也很可能无法正确舍入。此外，即使 Intel 和 AMD 的效果都非常好，但这两个版本偶尔会不同。借助 Oracle Developer Studio C/C++ 编译器，可以使用 `-xbuiltin=%default`（特别是在 `-fast` 之后）来确保编译器不会使用内置超越函数来内联取代任何超越指令。与此类似，`-fast` 之后的 `-xnolibmil` 选项禁用内联模板；Oracle Developer Studio 中的 `libm.il` 可能会有一些调用超越指令的模板。

5.5.2 结合运算

在实数运算中，加法和乘法属于结合运算，总和与乘积可按任意顺序计算。但是，存在舍入时，计算的顺序会影响计算答案。

- 避免使用 `-xreduction` 并行化选项。Oracle Developer Studio 以非确定性的方式优化约简。
- 避免使用 Fortran 的 `DOT` 和 `MATMUL` 运算。Fortran 90 及更高版本中的这些内部函数在不同平台上使用不同方法计算，舍入结果不同。如果还启用了并行化，由于优化约简，结果可能不确定。点积和矩阵乘法操作可以使用可移植 Fortran 编写代码，如 Netlib 软件系统信息库的 LAPACK 库中提供的那样。

5.5.3 无定形计算

在许多语言中，外部表达式求值的顺序并非按语言指定。因此，如果 `ranf(x)()` 是随机数生成器，在两个 `ranf(x)()` 调用的求值顺序发生变化时，表达式 `ranf(x) * a + ranf(x) * b()` 可能会为不同的编译器给出不同结果，或者在相同编译器上给出不同优化级别。

避免使用具有两个外部引用的表达式，将这样的表达式拆分为多个语句，每个语句中最多有一个外部引用。因此

```
z = ranf(x) * a + ranf(x) * b()
```

可替换为

```
t = ranf(x) * a()

z = t + ranf(x) * b()
```

。

5.5.4 不可移植类型

C/C++ 中的 `long double` 实现在 SPARC 和 x86 上的 Studio 中有所不同，前者有效位为 113，后者有效位为 64。因此，使用显式 `long double` 变量的程序在 SPARC 和 x86 上的执行结果一定会不同。

5.5.5 隐式更高精度

在一些情况下，表达式的求值精度可能高于源代码中显式声明的精度。使用 x87 扩展精度寄存器来计算涉及单精度或双精度变量的表达式时，会出现这种情况。在使用混合乘加运算替换乘法加法对时，也会出现这种情况。

- 避免将乘加对优化为混合乘加运算。在 `-fast` 后使用 `-fma=none`。
- 如果必须使用 `-xarch=386` 并且没有显式使用长双精度型类型，则在所有变量为浮点数的情况下使用 `-fprecision=single` 编译，或者在所有变量为双精度的情况下使用 `-fprecision=double` 编译，可以减少扩展精度表达式求值的影响。但是，如果在 `-xarch=386` 下使用 Fortran `complex*8` 变量，则没有方法确保所有表达式求值以单精度进行。最好使用 `-m64` 而不是 `-m32`，因为寄存器中传递的函数值精度与函数的精度相同。

5.6 独立确认

在前面有关可再现性的讨论中，假定要再现的结果是正确的，即使可能是众多正确结果中的一个。但是，我们如何才能最终确定这一点？一些程序可提供证明，但是证明通常会比程序更复杂。为什么这些证明比程序更可靠？一些程序对具有守恒律的物理系统建模，可以对其进行检查，但是，如果要进行的物理发现是守恒律不完整或不正确的情况，应该怎么办？

任何重要的决策需要采用独立方法来确认。对于通过计算机辅助做出的大部分重要决策情况，可能需要采取独立方法，包括不同的计算机、采用不同指令集、运行不同操作系统、使用以不同计算机语言编写的程序、实现不同的运算，然后由位于另一区域、以不同的自然语言思考的其他调查人员完成所有验证。在这一点上需要进行到什么程度，取决于做出不正确结论的成本如何。

因此，举例而言，在编写程序以测试基本转换时，至少注意，所用测试算法与要测试的基本转换函数可能使用的任意算法应截然不同。这就是为什么即使速度很慢，基本算法在计算机科学中仍有自己的一席之地，即用于测试快速复杂的算法。

♦ ♦ ♦ 附 录 A

示例

此附录提供如何完成一些常见任务的示例。这些示例使用 Fortran 或 ANSI C 编写，多数依赖于 `libm` 和 `libsunmath` 的当前版本。这些示例已在 Oracle Solaris 10 Update 10 OS 或更高版本上使用 Oracle Developer Studio 12.5 进行了测试。C 示例是使用 `-lsunmath -lm` 选项编译的。

A.1 IEEE 运算

以下示例显示了一种检查浮点数的十六进制表示形式的方法。请注意，您还可以使用调试器来查找已存储数据的十六进制表示形式。

以下 C 程序输出双精度的近似 `??` 值，以及单精度无穷大：

例 6 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 */
 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，使用 -lsunmath 编译时，上述程序的输出如下所示：

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

以下 Fortran 程序分别以两种格式输出最小的正规数：

例 7　　分别以两种格式输出最小的正归数（续）

```
 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()

 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，对应的输出显示如下：

```
min normal, quad:   3.36210314311209350626267781732175260 26E-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

本节显示使用数学库中函数的示例。

A.2.1 随机数字生成器

以下示例调用随机数字生成器以生成数字数组，并使用计时函数测量计算指定数字的 EXP 所用的时间：

例 8 随机数字生成器

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

 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
```

要编译上述示例，请将源代码放在后缀为 F（不是 f）的文件中，这样编译器将自动调用预处理程序，然后在命令行上指定 -DSP 或 -DDP 以选择单精度或双精度。

此示例显示如何使用 d_addrans 函数来生成均匀分布在用户指定范围中的随机数据块：

例 9 使用 d_addrans 函数

```
/*
 * 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;

   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 推荐函数

以下 Fortran 示例使用 IEEE 标准推荐的一些函数：

例 10 IEEE 推荐函数

```
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)
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

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
write(*,4) r_direction, s

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

print *
end
```

此程序的输出显示在以下示例中。

例 11 示例 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` 编译器和 `-f77` 兼容性选项，将显示以下附加消息。

```
 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)
```

A.2.3 IEEE 特殊值

以下 C 程序调用多个 *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;
}
```

在链接时，请记住同时指定 `-lsunmath` 和 `-lm`。

在基于 SPARC 的系统上，输出显示类似下文：

```
quiet NaN: NaN = 7ff80000 00000000
nextafter(max_subnormal,0) = 2.2250738585072004e-308
 = 000fffff fffffffe
single precision min subnormal = 1.40129846e-45 = 00000001
```

由于 x86 体系结构是“小尾数法”，x86 上的输出略有不同，其中将保留双精度数值的十六进制表示形式的高位字和低位字：

```
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 程序应该谨慎声明这些函数类型：

```
 program print_ieee_values
c
c the purpose of the implicit statements is to insure
c that the floating-point 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 SPARC 上，输出显示如下：

```
6.475175119438025110924438958227646E-4966
Inf
3.4028235E+38
```

A.2.4 ieee_flags – 舍入方向

以下示例演示了如何将舍入模式设置为向零取整：

```
#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;
}
```

上一个简短的舍入方向程序的以下输出显示了在 SPARC 上向零取整的效果：

```
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%
```

上一个简短的舍入方向程序的以下输出显示了在 x86 上向零取整的效果：

```
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%
```

要在 Fortran 程序中设置向零取整方向，请使用以下示例：

```
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
```

输出如下所示：

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

如果使用 f95 编译器和 -f77 兼容性选项编译程序，则输出中将包括以下附加消息。

```
demo% f95 ieee_flags_demo.f -f77
demo% a.out
 Note: Rounding direction toward zero
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

A.2.5 C99 浮点环境函数

下一个示例说明了如何使用多个 C99 浮点环境函数。norm 函数计算欧几里得向量范数，并且使用环境函数来处理下溢和溢出。主程序使用缩放的向量调用此函数以确保出现下溢和溢出，如追溯诊断输出所示

例 12 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;
    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;

    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，编译和运行此程序会生成类似下文的输出：

```
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
```

以下代码示例显示 `fesetprec` 函数在基于 x86 的系统上的效果。此函数在基于 SPARC 的系统上不可用。`while` 循环通过查找在加 1 时完全舍入的 2 的最大次幂，尝试确定可用精度。显示第一个循环时，在使用扩展精度计算所有中间结果的类似于基于 x86 的系统的体系结构上，此技术并不总是按照预期工作。因此，`fesetprec` 函数可用于确保所有结果将舍入到所需精度，如第二个循环所示。

例 13 `fesetprec` 函数 (x86)

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

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 系统上使用 `cc A8.c -lm -xarch=386` 编译时将生成

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

最后，以下示例显示一种方法，在多线程程序中使用环境函数将浮点模式从父线程传播到子线程，并恢复在子线程与父线程连接时在其中引发的异常标志。有关编写多线程程序的更多信息，请参见《Multithreaded Programming Guide》。

例 14 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 */
    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` – 已发生异常

通常情况下，用户程序会检查或清除已发生异常位。以下示例是一个检查已发生异常标志的 C 程序。

例 15 检查已发生异常标志

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

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

运行上一程序的输出如下所示：

```
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)
```

从 Fortran 可以完成相同操作：

例 16 检查已发生异常标志 – 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
```

输出如下所示：

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

虽然用户程序设置异常标志是不太常见的情况，不过可以这么做。以下 C 示例中演示了这种情况。

```
#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，上一个程序的输出为：

```
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：捕获异常

注 - 以下示例仅适用于 Oracle Solaris OS。

以下 Fortran 程序安装信号处理程序以定位异常，该程序仅适用于基于 SPARC 的系统：

例 17 在下溢时自陷 (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
 x = x * x
 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
 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
```

使用 `-f77` 编译上面的代码时，输出如下所示：

```
 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)
```

以下为 SPARC 系统上的更复杂的 C 示例：

例 18　在无效、被零除、溢出、下溢和不精确时自陷 (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");
 /* 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;
 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;
 }

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

输出以下类似内容：

```
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)
```

以下代码显示了如何使用 `ieee_handler` 和 include 文件来修改 SPARC 上特定异常情况的缺省结果：

例 19　　修改异常情况的缺省结果

```
/*
 * 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>

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;
}

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;
 }
}
```

以下为预期的输出：

```
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`：出现异常时中止

可以使用 `ieee_handler` 来强制程序在出现特定浮点异常时中止：

```
#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

下面的示例显示了如何使用 `libm` 中提供的一些异常处理功能。第一个示例基于以下任务：提供数字 x 和系数 a_0、a_1、...、a_N，以及 b_0、b_1、...、b_{N-1}，对函数 `f(x)` 及其一阶导数 `f'(x)` 求值，其中 `f（）` 是连分数：

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

在 IEEE 运算中直接计算 `f`（）：即使其中一个中间除法溢出或者被零除，该标准指定的缺省值（带正确符号的无穷大）也会得到正确结果。另一方面，计算 `f'`（） 可能会更困难，因为对其最简单的求值方法具有可去奇点。如果计算过程遇到这些奇点之一，则会尝试对不定式 0/0、0*无穷大或无穷大/无穷大之一求值，这些不定式会引发无效运算异常。W. Kahan 建议了一种方法，使用称为预替换的功能来处理这些异常。

预替换是对 IEEE 缺省异常响应的扩展，允许用户预先指定在异常运算中要替换的结果值。在 `libm` 中使用异常处理工具，程序可以通过在 `FEX_CUSTOM` 异常处理模式下安装处理程序，方便地实现预替换功能。使用此模式时，处理程序只需将值存储在数据结构中，传递到处理程序的信息参数指向该数据结构，这样就能为异常运算的结果提供任意值。以下样例程序使用通过 `FEX_CUSTOM` 处理程序实现的预替换功能来计算连分数及其导数。

例 20　使用 `FEX_CUSTOM` 处理程序计算连分数及其导数

```
#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;
        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;
}
```

多个有关程序的注释是按顺序排列的。在进入时，函数 `continued_fraction` 保存当前针对被零除和所有无效运算异常的异常处理模式。然后，它为被零除建立连续异常处理，并且为三个不定式建立 `FEX_CUSTOM` 处理程序。此处理程序会将 0/0 以及无穷大/无穷大替换为无穷大，而将 0*无穷大替换为全局变量 `p` 的值。请注意，在每次执行函数求值循环时都必须重新计算 `p`，以提供正确值来替换后续的 0*无穷大这一无效运算。另请注意，`p` 必须声明为 `volatile` 以防止编译器删除它，因为它在循环中的其他位置未显式涉及。最后，对于使用 `p` 来提供预替换值的计算过程，为了防止编译器将 `p` 的赋值移动到该计算过程之上或之下（这会导致异常），该计算过程的结果还将赋值给 `volatile` 变量（程序中的变量名为 `t`）。对 `fex_setexcepthandler` 的最终调用将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` 函数来启用对追溯诊断的日志记录。执行操作之前，它会引发不精确标志；这具有防止记录不精确异常的效果。请记住，在 `FEX_NONSTOP` 模式

中，如果引发了其标志，则不记录异常；如第 4.4.3.2 节 “回顾诊断” [75]一节中所述。主程序还会为常见异常建立 FEX_ABORT 模式，以确保任何未由 continued_fraction 处理的不常见异常将导致程序终止。最后，程序在多个不同点对特定连分数求值。如以下样例输出中所示，计算过程并未真正遇到中间异常：

```
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 f'(x) 计算中，出现在 x = 1、4 和 5 时的异常不会生成追溯诊断消息，因为它们在程序中出现的位置与 x = –3 时出现的异常位置相同。

以上程序可能不是处理在连分数及其导数求值过程中出现的异常的最有效方法。原因之一是每次迭代循环时都必须重新计算预替换值，不论是否需要。在这种情况下，预替换值的计算涉及到浮点除法，在现代的 SPARC 和 x86 处理器上，浮点除法是相对较慢的运算。此外，循环本身涉及到两个除法，由于大部分 SPARC 和 x86 处理器无法交叠执行两个不同的除法运算，除法很可能成为循环中的瓶颈；添加另一个除法还会加剧这一瓶颈。

可以重新编写循环，这样只需要一次除法，特别是在预替换值的计算不需要涉及到除法的情况下。要以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比率。这将消除多次除法运算的瓶颈，但是不会消除预替换值计算中涉及的所有算术运算。此外，需要对预替换值进行赋值以及对预替换为 volatile 变量的运算结果进行赋值，这会带来额外的内存操作，从而降低程序运行速度。虽然对于防止编译器重新排序特定关键运算而言，这些赋值是必要的，但是也会有效地阻止编译器重新排序其他无关运算。因此，在本例中通过预替换处理异常需要额外的内存运算，并会预先排除一些在其他情况下可能可用的优化。能否更高效地处理这些异常？

在缺少特定硬件支持来实现快速预替换的情况下，处理本例中异常的最有效方法是使用标志，如以下版本所示：

例 21 使用标志处理异常（续）

```
#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);

    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，第一个循环尝试以缺省的连续模式计算 `f(x)` 和 `f'(x)`。如果引发了无效标志，则第二个循环将重新计算 `f(x)` 和 `f'(x)`，显式测试是否出现 NaN。通常，不会出现无效运算异常，因此程序仅执行第一个循环。此循环不引用 `volatile` 变量，也没有额外的算术运算，因此运行速度将实现编译器所能达到的最大速度。实现这种效率的代价是需要编写与第一个循环几乎相同的第二个循环，用于处理出现异常的情况。这一折衷是浮点异常处理所带来的典型难题。

A.3.5 在 Fortran 程序中使用 `libm` 异常处理

`libm` 中的异常处理工具主要在 C/C++ 程序中使用，不过利用 Sun Fortran 语言的互操作性功能，您也可以从 Fortran 程序调用某些 `libm` 函数。

注 - 如果需要一致的行为，请不要在同一个程序中同时使用 `libm` 异常处理函数和 `ieee_flags` 以及 `ieee_handler` 函数。

以下示例显示了使用预替换来对连分数及其导数进行求值的 Fortran 程序版本（仅限 SPARC）：

例 22 使用预替换对连分数及其导数进行求值 (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

 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
 data ixff2/x'ff2'/
 data ix2/x'2'/
 data ixb0/x'b0'/

 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(ixff2))
c x'2' = FEX_DIVBYZERO, 0 = FEX_NONSTOP
 call fex_set_handling(%val(ix2), %val(0), %val(0))
c x'b0' = FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI, 3 = FEX_CUSTOM
 call fex_set_handling(%val(ixb0), %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(ixff2))
 return
 end

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/
 data ixffa/x'ffa'/

 external fex_set_handling
c$pragma c(fex_set_handling)
c x'ffa' = FEX_COMMON, 1 = FEX_ABORT
 call fex_set_handling(%val(ixffa), %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
```

此程序使用 `-f77` 标志编译的输出如下所示：

```
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) = 0.203704
f( 1) = -1.00000    , f'( 1) = 0.333333
f( 2) = -.777778    , f'( 2) = 0.120370
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;  Underflow; 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 其他

A.4.1 sigfpe：捕获整型异常

上节中显示了使用 ieee_handler 的示例。通常而言，可以选择使用 ieee_handler 或 sigfpe 时，推荐使用前者。

注 - sigfpe 仅在 Oracle Solaris OS 上可用。

在要使用的处理程序是 sigfpe 时，有诸如捕获整型运算异常等实例。在基于 SPARC 的系统上，例 23 “捕获整型异常”在整数被零除时自陷。

例 23 捕获整型异常

```
/* Generate the integer division by zero exception */
#include <signal.h>
#include <siginfo.h>
#include <ucontext.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;
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 调用 Fortran

以下为 C 驱动程序调用 Fortran 子例程的简单示例。有关使用 C 和 Fortran 的更多信息，请参阅《Oracle Developer Studio 12.5：C 用户指南》和《Oracle Developer Studio 12.5：Fortran 用户指南》。以下为 C 驱动程序（将其保存在名为 `driver.c` 的文件中）：

例 24 从 C 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];
 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
```

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
```

输出结果看起来类似于下文：

```
 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

下表显示了 SPARC 体系结构调试命令的示例。

表 35 一些调试命令 (SPARC)

操作	**dbx**	**adb**
在函数上设置断点	`stop in myfunct`	`myfunct:b`
在行号上设置断点	`stop at 29`	n/a
在绝对地址上设置断点	n/a	`23a8:b`
在相对地址上设置断点	n/a	`main+0x40:b`
运行直至遇到断点	`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`
检查 fp 寄存器：（十六进制）等值十进制	`print -fx $f0`	`<f0=f`
检查 fp 寄存器：IEEE 双精度	`print $f0f1`	`<f0=X; <f1=X`

操作	**dbx**	**adb**
检查 fp 寄存器：（十六进制）等值十进制	`print -flx $f0f1`	`<f0=F`
检查所有 fp 寄存器	`regs -F`	`$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 -fllx $fsr`	`<fsr=X`
将单精度 1.0 放在 `f0` 中	`assign $f0=1.0`	`3f800000>f0`
将双精度 1.0 放在 `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`

显示浮点数时，应该注意，寄存器的大小为 32 位，单精度浮点数将占据 32 位（因此可以填入一个寄存器），而双精度浮点数占据 64 位（因此使用两个寄存器来存放双精度数）。在十六进制表示形式中，32 位对应于 8 个十六进制位。以下 FPU 寄存器快照使用 `adb` 显示，其排列如下：

<name of fpu register> <IEEE hex value> <single precision> <double precision>

第三列存放第二列中显示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。

由于 `f10` 和 `f11` 用于存放双精度值，因此在 `f10` 行上，将该值的第一个 32 位 `7ff00000` 解释为 `+NaN` 是没有任何意义的。将全部 64 位 `7ff00000 00000000` 解释为 `+Infinity` 就是有意义的转换。

`adb` 命令 `$x` 用于显示前 16 个浮点数据寄存器，同时还显示 `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
```

下表显示了 x86 体系结构调试命令的示例：

表 36 一些调试命令 (x86)

操作	**dbx**	**adb**
在函数上设置断点	`stop in myfunct`	`myfunct:b`
在行号上设置断点	`stop at 29`	n/a
在绝对地址上设置断点	n/a	`23a8:b`
在相对地址上设置断点	n/a	`main+0x40:b`
运行直至遇到断点	`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	`refs -F`	`$r`
检查 fp 状态寄存器	`print -fx $fstat`	`<fstat=X`
继续执行	`cont`	`:c`
单步	`step (or next)`	`:s`
退出调试器	`quit`	`$q`

以下示例显示了两种方法，用于在与 `adb` 中的例程 `myfunction` 对应的代码开头设置断点。首先可以使用：

```
myfunction:b
```

其次，您可以确定与对应于 `myfunction` 的代码片段开头部分相对应的绝对地址，然后在该绝对地址上设置断点：

```
myfunction=X
   23a8
23a8:b
```

Fortran 程序中使用 `f95` 编译的主子例程在 `adb` 中名为 `MAIN_`。要在 `adb` 中的 `MAIN_` 处设置断点，请键入：

```
   MAIN_:b
```

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 base-16 表示形式，而不是数字的十进制表示形式。对于基于 SPARC 的系统，`adb` 命令 `$x` 和 `$X` 同时显示十六进制表示形式和十进制值。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，双精度值显示奇数编号寄存器旁边的十进制值。

由于操作系统在进程首次使用浮点单元之前禁用浮点单元，因此在所调试的程序访问浮点寄存器之前，您无法修改浮点寄存器。

x86 上的对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]  +5.65391332434795490344196888 e73         EMPTY
 st[2]  +2.00000000000000008881784197             EMPTY
 st[3]  +1.80732183080704405560160047 e-1         EMPTY
 st[4]  +7.9180300235748291015625 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 - 对于 x86，cw 是控制字，sw 是状态字。

♦ ♦ ♦ 附 录 B

SPARC 行为和实现

本章讨论与基于 SPARC 的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 处理器，并介绍它们所支持的指令集和异常处理功能。

下表列出了由最新 SPARC 系统使用的硬件浮点实现：

表 37 在 Oracle Solaris 11 和更高版本中支持的 SPARC 系统

芯片	典型系统	最佳代码生成选项
T1	T1000、T2000、T6300、CP3060	`-xarch=sparcvis2 -xchip=ultraT1`
T2	T5120、T5220、T6320、CP3260	`-xarch=sparcvis2 -xchip=ultraT2`
T2+	T5140、T5240、T5440	`-xarch=sparcvis2 -xchip=ultraT2plus`
T3	T3-1、T3-2、T3-4	`-xarch=sparcvis3 -xchip=T3`
T4	T4-1、T4-1B、T4-2、T4-4	`-xarch=sparc4 -xchip=T4`
T5	T5-1B、T5-2、T5-4、T5-8	`-xarch=sparc4 -xchip=T5`
M5	M5-32	`-xarch=sparc4 -xchip=M5`
M6	M6-32	`-xarch=sparc4 -xchip=M6`
M7	M7-32	`-xarch=sparc5 -xchip=M7`
SPARC64-VI	M4000、M5000、M8000、M9000	`-xarch=sparcfmaf -xchip=sparc64vi`
SPARC64-VII	M3000、M4000、M5000、M8000、M9000	`-xarch=spracima -xchip=sparc64vii`
SPARC64-VII+	M3000、M4000、M5000、M8000、M9000	`-xarch=sparcima -xchip=sparc64viiplus`
SPARC64-X	M10-1、M10-4、M10-4S	`-xarch=sparcace -xchip=sparc64x`

表 38 UltraSPARC 系统在 Oracle Solaris 10 Update 10 中受支持，但在 Oracle Solaris 11 中不受支持

UltraSPARC 芯片	典型系统	最佳代码生成选项
I	Ex000	-xarch=sparcvis -xchip=ultra
II	Ex000、E10000	-xarch=sparcvis -xchip=ultra2
IIi	Ultra-5、Ultra-10	-xarch=sparcvis -xchip=ultra2i
IIe	Sun Blade 100	-xarch=sparcvis -xchip=ultra2e
III	Sun Blade 1000、2000	-xarch=sparcvis2 -xchip=ultra3
IIIi	Sun Blade 1500、2500	-xarch=sparcvis2 -xchip=ultra3i
IIICu	Sun Blade 1000、2000	-xarch=sparcvis2 -xchip=ultra3cu
IV	V490、V890、Ex900、E20K、E25K	-xarch=sparcvis2 -xchip=ultra4
IV+	V490、V890、Ex900、E20K、E25K	-xarch=sparcvis2 -xchip=ultra4plus

尽管不受支持，但在 Oracle Solaris 10 Update 10 或早期版本的 Oracle Solaris 上使用 Oracle Developer Studio 12.5 编译的程序通常在支持早期版本的 Oracle Developer Studio 的较旧 SPARC 系统上运行。要创建可执行文件以在这些平台上进行测试，可尝试使用以下选项进行编译：

```
-m32 -xarch=generic -xchip=generic
```

对于支持的解决方案，可在您要使用的早期版本的 Oracle Solaris 上编译，然后使用该 Oracle Solaris 版本支持的最新版本的 Oracle Developer Studio 进行编译。

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。使用缺省 `-xarch` 或显式 `-xarch=sparc` 编译的程序可在上面列出的任何基于 SPARC 的系统上运行，尽管它可能无法充分利用较新处理器的功能。同样，如果某系统基于 SPARC 且支持用特定 `-xchip` 指定指令集，则用特定 `-xarch` 值编译的程序可在该系统上运行，但是，如果在未指定处理器的系统上运行时，速度会非常慢。

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 Architecture Manual Version 9*》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所有这些功能，请使用 `-xarch=sparc` 进行编译。这些处理器还提供了标准指令集的扩展。使用这些 `-xarch` 值，可连续生成这些附加指令：

- `sparcvis`
- `sparcvis2`
- `sparcvis2`
- `sparc4`

- sparc5

这些附加指令很少由编译器自动生成，但可在汇编代码中使用它们。

可以使用 -xtarget 宏选项同时指定 -xarch 和 -xchip 选项。-xtarget 标志只是扩展到 -xarch、-xchip 和 -xcache 标志的适当组合。缺省的代码生成选项是 -xtarget=generic。有关包括 -xarch、-xchip 和 -xtarget 值的完整列表在内的更多信息，请参见 cc(1)、CC(1) 和 f95(1) 手册页以及《Oracle Developer Studio 12.5：Fortran 用户指南》、《Oracle Developer Studio 12.5：C 用户指南》和《Oracle Developer Studio 12.5：C++ 用户指南》编译器手册。

B.1.1 浮点状态寄存器和队列

对于所有 SPARC 浮点单元来说，无论它们实现哪种版本的 SPARC 体系结构，它们都提供一个浮点状态寄存器 (floating-point status register, FSR)，该寄存器中包含与 FPU 相关的状态位和控制位。所有实现延迟浮点陷阱的 SPARC FPU 都提供一个浮点队列 (floating-point queue, 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，以便处理浮点陷阱；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]  ! store FSR at specified address
ld      [addr], %fsr  ! load FSR from specified address
```

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。

下图显示浮点状态寄存器中位字段的布局。

图 7 SPARC 浮点状态寄存器

在第 7 版本和第 8 版本的 SPARC 体系结构中，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。

在图中，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陷阱。其余字段将在下表中介绍

表 39 浮点状态寄存器字段

字段	包含
RM	舍入方向模式
TEM	陷阱启用模式
NS	非标准模式
fcc	浮点条件代码
aexc	已发生异常标志
cexc	当前异常标志

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异常控制捕获功能并记录已发生和当前异常标志。下表对这些字段进行了细分。

表 40 异常处理字段

字段	寄存器中的相应位				
TEM，陷阱启用模式	NVM	OFM	UFM	DZM	NXM
	27	26	25	24	23
aexc，已发生异常标志	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、溢出、下溢、除以零和不精确异常。）

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且无需软件的支持。但是，在以下四种情况下，硬件将无法成功完成浮点指令：

- 浮点单元被禁用。
- 指令未被硬件所实现，如，任何 SPARC FPU 上的四倍精度指令。
- 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，并且已经启用了该异常的陷阱。

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，内核在软件中模拟捕获指令。请注意，模拟指令也可能导致已启用陷阱的异常。

在上述的前三种情况下，如果模拟指令未导致已启用陷阱的 IEEE 浮点异常，则说明内核已完成该指令。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给目标寄存器。内核还会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，并将这些异常与已发生异常标志进行“或”运算。内核随后安排继续从捕获点执行该进程。

当硬件执行的或者内核软件模拟的某个指令导致一个已启用陷阱的 IEEE 浮点异常，则该指令将无法完成。目标寄存器、浮点条件代码和已发生异常标志保持不变，当前异常标志设置为反映导致了该陷阱的特定异常，内核向该进程发送一个 `SIGFPE` 信号。

下面的伪代码概括了浮点陷阱的处理。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;  // if fpop is an arithmetic op
    fcc = fcc_result;  // if fpop is a compare
    cexc = texc;
    aexc = (aexc or texc);
else
    cexc = trapped IEEE exception generated by FPop;
    throw SIGFPE;
```

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（包括陷阱类型）。

在正常情况下，每个进程中只应出现一次 `fp_disabled` 陷阱。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陷阱。处理完该陷阱之后，内核会启用浮点单元，并使其在以后的进程中保持启用状态。（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的时才这样做。）

无论浮点单元何时遇到它未实现的指令，都会出现 `unimplemented_FPop` 陷阱。由于目前大多数的 SPARC 浮点单元都至少实现由《*SPARC Architecture Manual Version 8*》定义的指令集（四倍精度指令除外），而且 Oracle Developer Studio 编译器不生成四倍精度指令，所以不应在使用 `-xarch=sparc` 编译的大多数系统上出现这种类型的陷阱。

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已启用陷阱的 IEEE 浮点异常时，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，否则该进程将被中止。大多数程序不会产生被捕获的 IEEE 浮点异常，这是因为，启用捕获的目的常常是在异常产生时中止程序，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缺省行为。但是，正如第 4 章 异常和异常处理中所述，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。

即使禁用捕获或者指令不引发已启用陷阱的异常，大多数 SPARC 浮点单元也会在某些情况下自陷，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` 陷阱并使内核模拟软件完成指令。不同的 SPARC FPU 对导致 `unfinished_FPop` 陷阱的条件会有不同的反应。例如，大多数早期的 SPARC FPU 会捕获所有的 IEEE 浮点异常而无论捕获是否被启用；在某浮点异常的陷阱被启用并且硬件无法确定某指令是否会导致该异常的情况下，UltraSPARC FPU 将进行捕获。但任何最新的 SPARC 处理器将在硬件中处理所有异常情况，并且从不产生 `unfinished_FPop` 陷阱。

由于大多数 `unfinished_FPop` 陷阱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中止异常）来避免导致过量陷阱。当然，必须认真平衡以下两个方面带来的成本：处理异常；允许异常引发 `unfinished_FPop` 陷阱。

B.1.2.2 次正规数和非标准算法

在某些 SPARC 浮点单元产生 `unfinished_FPop` 陷阱的大多数常见情况下，都会生成次正规数。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，许多较旧的 SPARC 浮点单元将自陷。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整体准确性影响很小，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` 陷阱相关的性能的下降。

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符合 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。

并非所有的 SPARC 实现都提供非标准模式。不支持此模式的 SPARC 实现将忽略它，因此，数值和异常结果在非标准模式中是相同的。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。在缺省情况下，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()` 会打印一条类似如下的消息：

```
Note: IEEE floating-point exception flags raised:
 Inexact; Underflow;
See the Numerical Computation Guide, 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

real 305.3
user 32.4
sys 271.9
```

如果系统时间（上面输出的第三个数字）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。有关渐进下溢的更多信息，请参见第 2 章 IEEE 运算。

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，用于模拟已捕获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引发陷阱时，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。它对执行那些运行时间可以被预计的简单指令的循环进行计时，从而估计时钟速度。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 T4-2 服务器上，`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
 A SPARC-based CPU is available.
 Kernel says CPU's clock rate is 1500.0 MHz.
 Kernel says main memory's clock rate is 150.0 MHz.

 Sun-4 floating-point controller version 0 found.
 An UltraSPARC chip is available.

 Use "-xtarget=T4 -xcache=16/32/4/8:128/32/8/8:4096/64/16/64" code-generation option.

Hostid = hardware_host_id
```

有关更多信息，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。

C.1 受支持的系统的代码生成

Oracle Solaris 支持 Oracle、Sun 和其他系统供应商提供的许多系统，这些系统包含 Intel、AMD 和其他芯片供应商提供的 x86 处理器。某个特定的 Oracle Solaris 版本支持包含此类芯片的一些特定系统。对于某个特定 Oracle Solaris 版本，请参见其相应的硬件兼容性列表。

Oracle Solaris 11 支持支持 64 位寻址的 x86 处理器。Oracle Solaris 10 Update 10 支持这些 64 位处理器，以及许多具有硬件浮点和 120 MHz 或更快时钟速度的仅限 32 位的 x86 处理器。

使用 `-m32 -xarch=generic -xchip=generic` 标志编译可生成适用于大多数系统的代码。下表列出了一些典型 Oracle 和 Sun x86 系统的某些特定代码生成选项：

系统	代码生成选项
Ultra 20	`-xarch=sse2a -xchip=opteron`
X2200	`-xarch=amdsse4a -xchip=amdfam10`
X6250	`-xarch=sse3 -xchip=core2`
X4170	`-xarch=aes -xchip=westmere`
X2-4	`-xarch=sse4_2 -xchip=nehalem`
X3-2	`-xarch=avx -xchip=sandybridge`
X4-2 X4-4	`-xarch=avx_i -xchip=ivybridge`
?	`-xarch=avx2 -xchip=haswell`

有上百个不同的 x86 芯片，每个芯片都有复杂的命名规则。

使用 `cc -dryrun -native` 是了解编译器采取什么操作来优化特定系统的最好方法。生成适合一些不同 x86 系统所需的代码时，使用适合最旧系统的选项通常适用于所有系统。

C.2 与 SPARC 的区别

Oracle Developer Studio 编译器生成的代码在 SPARC 和 x86 上的行为方式通常是相同的。但是，请注意在基于 x86 的系统上有以下重要区别：

- x87 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，所以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。虽然 Oracle Developer Studio 12.5 在缺省情况下不使用 x87 寄存器进行单精度和双精度表达式求值，但如果指定了 `-xarch=386`，或使用了 x87 硬件超越指令，或使用了双精度扩展变量，就会使用这些寄存器。
- 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。使用 `-m32` 时，浮点子例程操作数和结果将在 x87 寄存器中传递。
- 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 8 “位模式表示的值 (x86)”）。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*》补充资料

在阅读本数值计算指南时，每个用户都可以在 David Goldberg 于 1991 年 3 月发表在《Computing Surveys》上的论文《*What Every Computer Scientist Should Know About Floating-Point Arithmetic*》中找到有用的信息（请参见 http://dl.acm.org/citation.cfm?id=103163）。

论文摘要如下：

许多人将浮点运算看作非常深奥的话题。这相当让人吃惊，因为浮点在计算机系统中无所不在：几乎每种语言都有浮点数据类型；从 *PC* 到超级计算机在内的计算机都具有浮点加速器；绝大多数编译器都会被不断调用来编译浮点算法；几乎所有操作系统都必须响应诸如溢出等浮点异常。本文提供了教程，介绍浮点对计算机系统的设计人员具有直接影响的各个方面。文中首先介绍了浮点表示形式和舍入误差的背景信息，然后对 *IEEE* 浮点标准进行了讨论，并通过示例来总结计算机系统开发人员如何更好地支持浮点运算。

此补充资料不是已发布的 Goldberg 论文的一部分。增加此内容的目的是说明一些要点，并更正读者在阅读论文时对 IEEE 标准可能产生的一些不正确概念。本材料并非由 David Goldberg 撰写，但在此处使用已得到他的授权。标准 754-1985 和 854-1987 已由 754-2008 取代，后者指定了二进制和十进制浮点运算。这不影响 Goldberg 的论文。

此补充资料专门讨论了第 D.1 节 “IEEE 754 实现之间的差别” [143]。本主题包含以下子主题：

D.1 IEEE 754 实现之间的差别

Goldberg 论文指出必须谨慎实现浮点运算，因为程序员需要依靠其属性来确保程序的正确性和精确度。特别是，IEEE 标准要求谨慎实现，只有在符合标准的系统上才能够编写能够正常工作并提供准确结果的有用程序。读者可能会得出结论，这样的程序应该

可以在所有 IEEE 系统上移植。实际上，如果“程序在两台均支持 IEEE 运算的计算机之间移动时，只要出现任何中间结果差异，一定是因为软件错误而不是算法的不同所造成的。”这种说法是正确的，那么编写可移植软件会容易得多。

不幸的是，IEEE 标准不确保相同的程序在所有符合标准的系统上提供完全一致的结果。在不同的系统上，由于各种原因，大部分程序实际上会生成不同的结果。例如，许多程序使用系统库提供的初等函数，而在标准中并未完全指定这些函数。1985 标准未完全指定数值在十进制和二进制格式之间的转换，也完全没有指定超越函数。

许多程序员可能未认识到，即使程序只使用 IEEE 标准中规定的数字格式和运算，在不同的系统上也会得到不同的结果。实际上，标准的作者的目的是允许不同的实现获得不同结果。在 IEEE 754 标准中定义术语目标时，其意图非常明显：“目标可以由用户显式指定，也可以由系统隐式指定（例如，子表达式中的中间结果或过程的参数）。一些语言将中间计算的结果放在用户无法控制的目标中。尽管如此，此标准按照该目标的格式和操作数的值来定义运算的结果。”（IEEE 754-1985，第 7 页）换而言之，IEEE 标准要求按照其中将放入结果的目标的精度，正确地舍入每个结果，但是该标准不要求目标的精度由用户程序确定。因此，不同系统可能会将其结果使用不同精度提供给目标，导致相同的程序生成不同结果（有时是动态生成），即使这些系统均符合标准也是这样。

引用论文中的多个示例运用了浮点运算舍入方法的一些知识。要想依靠这样的一些示例，程序员必须能够预测解释程序的方式，特别是在 IEEE 系统上，每个算术运算目标的精度是多少。IEEE 标准中关于目标定义的漏洞会影响到程序员理解如何解释程序的能力。因此，Goldberg 论文中提供了多个示例，这些示例在高级语言中作为明确可移植的程序来实现时，可能在 IEEE 系统上无法正确运行，因为这些系统通常使用与程序员预期所不同的精度向目标提供结果。另一个示例也许能运行，但也证明，程序的正常运行可能超出了程序员的一般能力范围。

在本补充资料中，IEEE 754 运算的当前实现基于通常所用的目标格式精度进行了分类。在回顾论文中的一些示例时，表明如果所提供结果的精度比程序预期的精度更高，会导致程序计算得出错误的结果，即使可以证明在使用预期精度时是正确的。论文中修订了这些证据之一，说明即使这并不表明程序是错误的，也应该掌握一些处理意外精度所需的知识技能。这些示例表明，尽管 IEEE 标准已有规定，其中允许的不同实现之间的差异会妨碍您编写可移植、高效且可准确预测结果的数值软件。要开发这样的软件，首先必须创建编程语言和环境，限制 IEEE 标准允许的可变性，从而使您能够表达出程序所需的浮点语义。1985 标准的 2008 版本针对编程语言做出了这样的推荐。

D.1.1 当前 IEEE 754 实现

当前 IEEE 754 运算的实现可以分为两组，按照它们在硬件中支持不同浮点格式的程度进行区分。扩展精度系统已使用 Intel x86 处理器系列和 `-xarch=386` 选项编译得到例证，为扩展双精度格式提供了完整支持，但对单精度和双精度只提供了部分支持：它们提供了指令，用于以单精度或双精度加载或存储数据，将其实时在扩展双精度格式之间来回转换，并提供特殊模式（而非缺省模式），其中数学运算的结果将舍入为单精度或双精度，即使将结果以扩展双精度格式保存在寄存器中也是如此。在这些模式中，Motorola 68000 系列处理器将结果舍入到单精度或双精度格式的精度和范围。在 Intel x86 和兼容处理器中，将结果舍入到单精度或双精度格式的精度，但保留与扩展双精度

格式相同的范围。单/双精度系统，包括大多数 RISC 处理器，提供了对单精度和双精度格式的完整支持，但不支持符合 IEEE 标准的扩展双精度格式。x86 SSE2 扩展提供了 SSE2 寄存器中的单/双精度系统，仍支持扩展精度寄存器中的扩展精度。

要查看计算在扩展精度系统上相比单/双精度系统可能会有什么不同的行为，请考虑 Goldberg 论文中示例 "Systems Aspects" 的 C 版本，如下所示：

```
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;
}
```

常量 3.0 和 7.0 解释为双精度浮点数，表达式 3.0/7.0 继承 `double` 数据类型。在单/双精度系统上，表达式将以双精度求值，因为这是可以使用的最有效格式。因此，将向 `q` 分配正确舍入到双精度的值 3.0/7.0。在下一行中，表达式 3.0/7.0 将以双精度格式再次求值，结果将等于刚赋值给 `q` 的值，因此程序将按预期输出 "Equal"（等于）。

在扩展精度系统上，即使表达式 3.0/7.0 具有类型 `double`，商将以扩展双精度格式在寄存器中计算，因此在缺省模式下，它将舍入到扩展双精度。但是，将得到的值赋值给变量 `q` 时，它可能会存储在内存中，由于 `q` 声明为 `double`，该值将舍入为双精度。在下一行中，表达式 3.0/7.0 可使用扩展精度再次求值，得到的结果不同于存储在 `q` 中的双精度值，这使得程序输出 "Not equal"（不等于）。

也有可能出现其他结果：编译器可以决定存储，从而舍入第二行中表达式 3.0/7.0 的值，然后再将其与 `q` 比较，或者可以将 `q` 以扩展精度格式保存在寄存器中而不存储。优化编译器可能会在编译时对表达式 3.0/7.0 求值，可能采用双精度，也可能采用扩展双精度。使用一个 x86 编译器，程序在采用优化编译的情况下输出 "Equal"（等于），在针对调试进行编译时输出 "Not Equal"（不等于）。最后，一些面向扩展精度系统的编译器会自动更改舍入精度模式，导致在寄存器中生成结果的运算将这些结果舍入到单精度或双精度，尽管可能具有更宽的范围。因此，在这些系统上，只是阅读程序的源代码并运用对 IEEE 754 运算的基本理解是无法预测其行为的。无法提供符合 IEEE 754 要求的环境的原因不能归结于硬件或编译器。正如对硬件要求的那样，硬件向每个目标提供正确舍入的结果，而编译器则将一些用户无法控制的中间结果分配给目标，这也是系统所允许的。

这一节的剩余部分将介绍使用 `-xarch=386` 编译的 x86 行为，这是 Oracle Developer Studio 12 及早期发行版中的缺省行为。在 Oracle Developer Studio 12.5 中，缺省值为 `-xarch=sse2`。

D.1.2 在扩展精度系统上执行计算的陷阱

传统观点认为，相比单/双精度系统，扩展精度系统如果不能做到更精确，至少应该生成相同精确度的结果，因为扩展精度系统至少能够提供相同精度，通常更高。以下章节中

讨论的普通示例以及基于这些示例的更精细的程序将表明，这种想法完全是一厢情愿：一些明确可移植的程序，确实可以跨单/双精度系统进行移植，但在扩展精度系统上会提供不正确的结果，恰恰是因为编译器和硬件有时提供的精度会超过程序预期的精度。

目前的编程语言使得程序难以指定预期的精度。如 Goldberg 论文中的 "Languages and Compilers" 一节所述，许多编程语言没有规定 `10.0*x` 这样的表达式在相同上下文中每次出现时都应该得到相同的值。在 IEEE 标准之前，一些语言（例如 Ada）由于不同运算之间的差别会受到这些方面的影响。近来，诸如 ANSI C 等语言已受到符合标准的扩展精度系统的影响。实际上，ANSI C 标准明确允许编译器对浮点表达式的求值结果使用高于通常与结果类型关联的精度。因此，表达式 `10.0*x` 的值可能会由于各种因素而变化：表达式是立即赋值给变量还是作为较大表达式中的子表达式；表达式是否参与比较；表达式是否作为参数传递到函数，如果是，参数是按值传递还是按引用传递；当前精度模式；编译程序的优化级别；编译器在编译程序时使用的精度模式和表达式求值方法等等。

表达式求值时的异常行为不能完全归咎于语言标准。只要表达式在扩展精度寄存器中求值，扩展精度系统的运行效率最高，即使需要存储的值以所需的最小精度存储。约束语言以便要求 `10.0*x` 在任何位置求值时得到相同值会对这些系统的性能造成不利影响。不幸的是，允许这些系统在语法等同的上下文中以不同方式对 `10.0*x` 求值时，自身会对准确数值软件的程序员造成不利影响，使他们无法依靠程序的语法来表达出所希望的语义。

以下示例探讨了真实程序是否能依赖于给定表达式始终可以计算得到相同值的假设。请回顾一下 Goldberg 论文中有关计算 ln(1 + x) 的定理 4，这是用 Fortran 编写的：

```
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
```

在扩展精度系统上，编译器可能会在扩展精度的第三行对表达式 `1.0 + x` 求值，并将结果与 `1.0` 进行比较。但是，将相同的表达式传递到第六行中的对数函数时，编译器可能会将其值存储在内存中，舍入到单精度。因此，如果 `x` 不是太小（在扩展精度下 `1.0 + x` 才会舍入到 `1.0`），而是足够小（在单精度下 `1.0 + x` 可以舍入到 `1.0`），则 `log1p(x)` 返回的值将为零而不是 `x`，相对误差为 1，而不是大于 5??。与此类似，假定第六行表达式的剩余部分（包括重复出现的子表达式 `1.0 + x` 在内）在扩展精度下求值。在这种情况下，如果 `x` 很小，但又不足够小，`1.0 + x` 在单精度下舍入到 `1.0`，则 `log1p(x)` 返回的值会超过正确值达到 `x`，相对误差仍接近 1。如果需要明确的示例，可以让 `x` 为 $2^{-24} + 2^{-47}$，因此 `x` 是最小的单精度数，这样 `1.0 + x` 向上舍入到下一个较大数 $1 + 2^{-23}$。然后，`log(1.0 + x)` 约为 2^{-23}。由于表达式中第六行的分母在扩展精度下求值，将对其精确求值并提到 `x`，因此 `log1p(x)` 返回的值大约为 2^{-23}，这接近精确值的两倍。

至少一个编译器上会实际出现这种情况。使用面向 x86 系统的 Sun WorkShop Compilers 4.2.1 Fortran 77 编译器和 `-O` 优化标志编译上述代码时，生成的代码精确按

照所述方式计算 `1.0 + x`。因此，函数将 `log1p(1.0e-10)` 计算为 0，将 `log1p(5.97e-8)` 计算为 `1.19209E-07`。

要使定理 4 的算法正确运行，表达式 `1.0 + x` 必须在每次出现时以相同方法求值。仅当 `1.0 + x` 在一个实例中求值为扩展双精度，在另一个实例中求值为单精度或双精度时，该算法在扩展精度系统上才会失败。由于 `log` 是 Fortran 中的通用内部函数，编译器可能会全程以扩展精度格式对表达式 `1.0 + x` 求值，以相同的精度计算其算法，但是显然，您不能假定编译器这样做。您还可以假设一个类似示例来包含用户定义的函数。在这种情况下，编译器仍会将参数保持在扩展精度，即使函数返回了单精度结果也是如此，不过极少数（如果有）现有 Fortran 编译器也会这样做。因此，您可能希望通过将 `1.0 + x` 传递到变量来尝试确保对它进行一致的求值。不幸的是，如果您将该变量声明为 `real`，编译器仍可能会妨碍这样做，它会在变量第一次出现时使用以扩展精度格式保存在寄存器中的值替换，变量另一次出现时则使用存储在内存中的值替换。反之，您可能需要使用与扩展精度格式对应的类型来声明变量。标准 FORTRAN 77 未提供这样做的方法，而 Fortran 95 则提供了 `SELECTED_REAL_KIND` 机制来描述不同格式，它不显式要求以扩展精度格式对表达式求值的实现允许以该精度声明变量。简而言之，没有可移植的方法能够确保以标准 Fortran 编写此程序，防止会使证据失效的方法来对表达式 `1.0 + x` 求值。

在另一些示例中，即使当存储了每个子表达式从而以相同精度舍入时，在扩展精度系统上也会无法正常运行。原因是双重舍入。在缺省精度模式下，扩展精度系统最初将每个结果舍入到扩展双精度。如果随后以双精度存储该结果，则再次舍入。这两种舍入的组合生成的值与将第一个结果正确舍入到双精度获得的结果不同。如果将结果舍入到扩展双精度是“中间情况”，即正好处于两个双精度数的中间，则可能出现这种情况，因此第二次舍入由“舍入到偶数优先”规则确定。如果第二次舍入与第一次舍入方向相同，则净舍入误差将超过最后位置单元的一半。不过请注意，双重舍入只影响双精度计算。您可以证明，两个 p 位数的和、差、积或商，或者一个 p 位数的平方根在首先舍入到 q 位然后舍入到 p 位时，只要 $q \geq 2p + 2$，就能得到相同的值，就像只舍入一次到 p 位一样。因此，扩展双精度足够宽，单精度计算不会经过双重舍入。

一些依赖于正确舍入的算法在双重舍入下可能会失败。实际上，即使一些算法不需要正确舍入，并且能够在各种不符合 IEEE 754 标准的计算机上正确工作，使用双重舍入时也会失败。这些方法中最有用的是可移植算法，用于执行 Goldberg 论文定理 5 中提到的模拟多精度运算。例如，定理 6 中所述用于将浮点数拆分为高位和低位的过程在双重舍入运算中无法正确运行：尝试将双精度数字 $2^{52} + 3 \times 2^{26} - 1$ 拆分为两部分，每一部分最多 26 位。每次运算正确舍入到双精度时，高位部分为 $2^{52} + 2^{27}$，低位部分为 $2^{26} - 1$，但每次运算首次舍入到扩展双精度然后再舍入到双精度时，该过程生成的高位为 $2^{52} + 2^{28}$，低位为 $-2^{26} - 1$。后一个数字将占据 27 位，因此在双精度下无法准确计算其平方。在扩展双精度下仍可能计算此数字的平方，但是得到的算法无法再移植到单/双精度系统上。此外，在多精度乘法算法的后面步骤中，假设所有部分乘积已经以双精度格式计算。正确处理混合的双精度和扩展双精度变量会大幅增加实现的成本。

与此类似，用于添加多精度数字（以双精度数字数组形式表示）的可移植算法也可以归类到双重舍入运算中。这些算法通常依赖类似于 Kahan 求和公式的技术。在 Goldberg 论文的 "Errors In Summation" 一节中给出的求和公式非正式说明建议，如果 `s` 和 `y` 为浮点变量，且 $|s| \geq |y|$，则可以计算如下：

```
t = s + y;
e = (s - t) + y;
```

那么，在大多数运算中，e 可以精确恢复在计算 t 时的舍入误差。此技术不适用于双重舍入运算，但是，如果 s = 2^{52} + 1 并且 y = 1/2 – 2^{-54}，则 s + y 以扩展双精度格式将前者舍入到 2^{52} + 3/2，此值以双精度格式按照“舍入到偶数优先”规则舍入为 2^{52} + 2；因此计算 t 时的净舍入误差为 1/2 + 2^{-54}，无法准确以双精度表示，因此以上显示的表达式无法准确计算。这里再次说明，通过以扩展双精度格式计算总和，有可能弥补舍入误差，但是，程序随后需要完成额外操作以将最终输出约简回双精度，而双重舍入也会扰乱此过程。因此，虽然采用这些方法模拟多重精度运算的可移植程序可在广泛的计算机上高效正确地运行，但在扩展精度系统上则不能如宣称的这样运行。

最后，直觉上依赖于正确舍入的一些算法事实上可能会在双重舍入下正确运行。在这些情况下，处理双重舍入的成本不在于实现，而在于验证算法按照宣称方式工作。为了说明这一点，现在证明定理 7 的以下变体：

D.1.2.1 定理 7

如果 *m* 和 *n* 是可以 IEEE 754 双精度表示的整数并且 $|m| < 2^{52}$ 以及 *n* 具有特殊形式 $n = 2^i + 2^j$，则 (*m* ø*n*) ⊗ *n* = *m*，前提是浮点操作正确舍入到双精度或者首先舍入到扩展双精度，然后舍入到双精度。

注 - 在此定理中，ø 和 ⊗ 分别表示计算的除法和计算的乘法。

D.1.2.2 证明

假定在无损的情况下 *m* > 0。让 *q* = *m* / *n*。按二的幂换算，我们可以考虑等同的设置，其中 $2^{52} \le m < 2^{53}$，对 *q* 也类似，因此 *m* 和 *q* 均为整数，最低有效位占据了单元位置（即，ulp(*m*) = ulp(*q*) = 1）。在换算之前，我们假设 $m < 2^{52}$，因此在扩展后，*m* 是偶数。此外，由于 *m* 和 *q* 的换算值满足 *m*/2 < *q* < 2*m*，*n* 的对应值必须具有两种形式之一，具体取决于 *m* 或 *q* 哪个更大：如果 *q* < *m*，则很明显 1 < *n* < 2，并且由于 *n* 是两个二的幂之和，对于一些 *k*，$n = 1 + 2^{-k}$；与此类似，如果 *q* > *m*，则 1/2 < *n* < 1，因此 $n = 1/2 + 2^{-(k+1)}$。由于 *n* 是两个二的幂之和，*n* 最接近于一的可能值是 $n = 1 + 2^{-52}$。由于 $m/(1 + 2^{-52})$ 不比小于 *m* 的下一个较小双精度数大，我们不能让 *q* = *m*。

用 *e* 指示计算 *q* 时的舍入错误，从而 *q* = *m*/*n* + *e*，计算的值 *q* ⊗ *n* 将是 *m* + *ne* 的（一次或两次）舍入值。请首先考虑每种浮点运算正确舍入到双精度的情况。在这种情况下，|*e*| < 1/2。如果 n 的形式为 $1/2 + 2^{-(k+1)}$，则 *ne* = *nq* – *m* 为 $2^{-(k+1)}$ 的整数倍，并且 $|ne| < 1/4 + 2^{-(k+1)}$。这表示 |*ne*| ≤ 1/4。请记住，*m* 与下一个可表示较大数之间的差为 1，*m* 与下一个可表示较小数之间的差在 $m > 2^{52}$ 时为 1，在 $m = 2^{52}$ 时为 1/2。因此，由于 |*ne*| ≤ 1/4，*m* + *ne* 将舍入到 *m*。（即使 $m = 2^{52}$ 且 *ne* = –1/4，积仍然按照“舍入到偶数优先”规则舍入为 *m*。）与此类似，如果 *n* 的形式为 $1 + 2^{-k}$，则 *ne* 为 2^{-k} 的整数

倍，并且 $|ne| < 1/2 + 2^{-(k+1)}$；这表示 $|ne| \le 1/2$。这种情况下不能让 $m = 2^{52}$，因为 m 严格大于 q，因此 m 与其最接近的可表示相邻数相差 ??1。因此，由于 $|ne| \le 1/2$，$m + ne$ 将再次舍入到 m。（即使 $|ne| = 1/2$，积仍然将按照“舍入到偶数优先”规则舍入为 m，因为 m 是偶数。）这就完成了正确舍入运算的证明。

在双重舍入运算中，可能仍会出现 q 是正确舍入的商（即使它实际上舍入了两次），因此如上所述，$|e| < 1/2$。在这种情况下，考虑到 $q \otimes n$ 将舍入两次这样一个事实，可以采用上一段中提供的参数。为了说明这一情况，请注意，IEEE 标准要求扩展双精度格式进位至少为 64 个有效位，这样数字 m ?? 1/2 和 m ?? 1/4 在扩展双精度下可准确表示。因此，如果 n 的形式为 $1/2 + 2^{-(k+1)}$，这样 $|ne| \le 1/4$，则将 $m + ne$ 舍入到扩展双精度一定会得到与 m 最多相差 1/4 的结果，如上所述，此值将以双精度舍入到 m。与此类似，如果 n 的形式为 $1 + 2^{-k}$，这样 $|ne| \le 1/2$，则将 $m + ne$ 舍入到扩展双精度一定会得到与 m 最多相差 1/2 的结果，此值将以双精度舍入到 m。请记住，这种情况中的 $m > 2^{52}$。

最后，我们只剩下考虑由于双重舍入而使 q 不是正确的舍入商的情况。在这些情况下，最坏的情况下将得到 $|e| < 1/2 + 2^{-(d+1)}$，其中 d 是扩展双精度中额外位的数量。所有现有的扩展精度系统支持恰好具有 64 个有效位的扩展双精度格式；对于此格式，$d = 64 - 53 = 11$。如果第二次舍入由“舍入到偶数优先”规则确定，则双重舍入只会得到不正确的舍入结果，因而 q 必须为偶数。因此，如果 n 的形式为 $1/2 + 2^{-(k+1)}$，则 $ne = nq - m$ 是 2^{-k} 的整数倍，并且 $|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}$。

如果 $k \le d$，则表明 $|ne| \le 1/4$。如果 $k > d$，就有了 $|ne| \le 1/4 + 2^{-(d+2)}$。在任何一种情况下，积的第一次舍入将得到与 m 最多相差 1/4 的结果，按照以前的参数，第二次舍入将舍入到 m。与此类似，如果 n 的形式为 $1 + 2^{-k}$，则 ne 为 $2^{-(k-1)}$ 的整数倍，并且 $|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}$。

如果 $k \le d$，则表明 $|ne| \le 1/2$。如果 $k > d$，就有了 $|ne| \le 1/4 + 2^{-(d+1)}$。在任何一种情况下，积的第一次舍入将得到与 m 最多相差 1/2 的结果，仍旧按照以前的参数，第二次舍入将舍入到 m。 ♦

以上证据表明，只有在商进行双重舍入时，积才会发生双重舍入。即使这样，也会舍入到正确结果。该证据还表明，扩展我们的推理以包括双重舍入可能会遇到难题，即使程序只有两次浮点运算也是如此。对于更复杂的程序，可能无法系统地解释双重舍入效果，更不用说那些更常见的双精度和扩展双精度计算组合。

D.1.3 扩展精度的编程语言支持

以上示例并非用于说明扩展精度本身有害。当程序员具备选择性使用的能力时，许多程序可从扩展精度中获益。不幸的是，当前编程语言没有提供足够的手段，来方便程序员指定适合使用扩展精度的情况和方法。为了指明需要什么支持，请考虑您希望用于管理扩展精度使用的方式。

在使用双精度作为其名义工作精度的可移植程序中，可以采用五种方法来控制更宽精度的使用：

1. 在扩展精度系统上尽可能使用扩展精度进行编译，这样可生成最快的代码。很明显，大多数数值软件所需要的算术精度并不会超过每次运算中由“计算机精度”所限制的相对误差。当内存中的数据以双精度存储时，计算机精度通常采用该双精度的最大相对舍入误差，因为假定在输入数据时这些数据已进行舍入，并且运算结果将在存储时同样地进行舍入。因此，虽然以扩展精度计算某些中间结果可能会产生更准确的结果，但扩展精度并不是必要的。在这种情况下，我们可能更希望仅当编译器使用扩展精度不会明显减慢程序速度时，才使用扩展精度，否则使用双精度。
2. 如果比双精度更宽的某种格式明显更快并且足够宽，则使用该格式，否则采用其他格式。当扩展精度可用时，可以更轻松地进行某些计算，不过这些计算也可以采用双精度运行，只需稍微付出一些劳动。请考虑计算双精度数字的欧几里得向量范数。通过使用 IEEE 754 扩展双精度格式及其更宽的指数范围来计算元素的平方并累积计算其总和，我们可以轻松地避免可行长度的向量过早下溢或溢出。在扩展精度系统上，这是计算范数的最快方法。在单/双精度系统上，如果扩展双精度格式根本不受支持，则必须在软件中进行模拟，并且此类模拟将比仅使用双精度要慢得多。需要测试异常标志以确定是否发生下溢或上溢，如果发生，将使用显式缩放重复计算过程。请注意，为支持扩展精度的这种用法，一种语言必须指示相当快的最宽可用格式（以便程序可以选择使用哪个方法），还必须提供环境参数来指示每种格式的精度和范围（以便程序可以验证相当快的最宽格式足够宽，例如，它比双精度的范围更宽）。
3. 使用比双精度更宽的某种格式，即使该格式必须在软件中进行模拟。对于比欧几里得范数示例更复杂的程序，程序员可能希望避免编写两种版本的程序，转而依赖于扩展精度（即使它速度比较慢）。另外，语言必须提供环境参数，这样程序可以确定最宽可用格式的范围和精度。
4. 不使用更宽的精度；将结果正确地舍入到双精度格式的精度，即使它可能具有更宽的范围。最容易编写的程序取决于正确舍入的双精度运算，包括上面提到的一些示例。语言必须为程序员提供一种方式来指示不得使用扩展精度，即使可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行。对于使用这种方式计算的中间结果，如果在存储到内存时发生下溢，仍然可能执行双重舍入：如果算术运算结果首先舍入为 53 个有效位，然后在必须非规范化时再舍入为更少的有效位，最终结果可能不同于仅舍入一次为非规范化数字所得到的结果。当然，这种形式的双重舍入几乎不可能对任何实际程序产生负面影响。
5. 将结果正确地舍入为双精度格式的精度和范围。这种双精度的严格实现最有益于下面这样的程序：这些程序在双精度格式的精度和范围限制附近测试数字软件或算术本身。这类严谨的测试程序一般很难以可移植方式编写；当它们必须利用伪子例程和其他技巧强制将结果舍入为特定格式时，它们甚至更难于编写并且容易出错。因此，程序员在使用扩展精度系统来开发必须可移植到所有 IEEE 754 实现的强大软件时，很快会欣喜地发现，能够模仿单/双精度系统的运算且不需要付出额外工作。

当前没有语言全部支持上面所有五种选项。实际上，少量语言已经尝试使程序员能够从根本上控制对扩展精度的使用。一个值得注意的例外情况是 ISO/IEC 9899:1999 编程语言 - C 标准，它是对 C 语言的主修订版。

C99 标准允许实现在表达式求值时使用的格式比通常与表达式类型关联的格式更宽，但 C99 标准推荐使用仅有的三种表达式求值方法中的一种。这三种推荐方法的特征在一定

程度上为哪些表达式“被提升”为更宽的格式。通过定义预处理程序宏 FLT_EVAL_METHOD 来帮助实现方式确定使用的是哪种方法：如果 FLT_EVAL_METHOD 为 0，则以每个表达式的类型所对应的格式对表达式求值；如果 FLT_EVAL_METHOD 为 1，则将 float 表达式提升为对应于 double 的格式；如果 FLT_EVAL_METHOD 为 2，则将 float 和 double 表达式提升为对应于 long double 的格式。（允许实现将 FLT_EVAL_METHOD 设置为 –1，以指示表达式求值方法是不确定的。）C99 标准还要求在 <math.h> 头文件中定义类型 float_t 和 double_t。这两种类型相应地至少与 float 和 double 一样宽，其目的是为了与 float 和 double 表达式的求值类型相符。例如，如果 FLT_EVAL_METHOD 为 2，则 float_t 和 double_t 为 long double。最后，C99 标准要求在 <float.h> 头文件中定义预处理程序宏，以指定对应于每种浮点类型的格式的范围和精度。

C99 标准要求或推荐的功能组合支持上面所列五个选项中的一部分，但并非全部支持。例如，如果实现将 long double 类型映射到扩展双精度格式，并将 FLT_EVAL_METHOD 定义为 2，则程序员可以合理地假定扩展精度相对较快，因此欧几里得范数示例这样的程序只需使用 long double 类型（或 double_t）的中间变量。另一方面，即使匿名表达式存储在内存中（例如，当编译器必须使浮点寄存器溢出时），同一实现也必须将匿名表达式保留为扩展精度，并且必须存储赋值给变量（这些变量已声明 double 来转换为双精度）的表达式结果，即使这些结果可能已保留在寄存器中。因此，double 和 double_t 类型都不能编译为在当前扩展精度硬件上生成最快的代码。

类似地，对于本节中的示例所阐述的一些问题（但不是全部），C99 标准提供了解决方法。如果将表达式 1.0 + x 赋值给变量（任何类型）并且到处使用该变量，log1p 函数的 C99 标准版本保证可以正确工作。但是，编写可移植的高效 C99 标准程序来将双精度数字拆分为高位部分和低位部分会更加困难：如果我们无法保证 double 表达式正确舍入为双精度，如何在正确位置拆分并避免双重舍入？一个解决方法是使用 double_t 类型在单/双精度系统上拆分双精度，并在扩展精度系统上拆分扩展精度，以便在其中任一情况下都能正确舍入运算。在定理 14 中说明，假设我们知道基础运算的精度，则我们可以在任何位的位置进行拆分；并且 FLT_EVAL_METHOD 和环境参数宏应该向我们提供此信息。

下面的代码片段显示了一个可能的实现：

```
#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;
```

要找到此解决方法，您必须了解：可以采用扩展精度对 `double` 表达式求值，随后发生的双重舍入问题会导致算法出现问题，并且可以根据定理 14 改用扩展精度。一个更明显的解决方法是，只需指定将每个表达式正确舍入为双精度。在扩展精度系统上，这样做只需要更改舍入精度模式，但很不幸，C99 标准并未提供可移植方法来完成此操作。在浮点 C 修订的早期草案工作文档中，指定了为支持浮点而对 C90 标准进行的更改，并建议系统上具有舍入精度模式的实现提供 `fegetprec` 和 `fesetprec` 函数来获取和设置舍入精度，这类似于获取和设置舍入方向的 `fegetround` 和 `fesetround` 函数。对 C99 标准进行更改之前，已经删除了此建议。

巧合的是，在 C99 标准用于支持在具有不同整数运算功能的系统之间进行移植的方法中，给出了一种更好的方法来支持不同的浮点体系结构。每个 C99 标准实现提供一个 `<stdint.h>` 头文件，其中定义该实现支持的那些整数类型，这些类型根据其大小和效率进行命名：例如，`int32_t` 是一种恰好为 32 位宽度的整数类型，`int_fast16_t` 是该实现的最快整数类型（至少为 16 位宽度），而 `intmax_t` 是能够支持的最宽整数类型。可以想像对于浮点类型有一个类似的方案：例如，`float53_t` 可以命名恰好为 53 位精度但范围可能更宽的浮点类型，`float_fast24_t` 可以命名实现中至少为 24 位精度的最快类型，而 `floatmax_t` 可以命名能够支持的最宽且相当快的类型。在扩展精度系统上，快类型可以允许编译器生成可能最快的代码，只是命名变量值不得在寄存器溢出后更改。在扩展精度系统上，精确宽度类型将导致编译器将舍入精度模式设置为舍入到指定精度，从而允许在相同约束限制下的更宽范围。最后，`double_t` 可以使用 IEEE 754 双精度格式的精度和范围来命名类型，从而提供严格的双精度求值。随着相应地命名环境参数宏，此类方案将轻松支持上面描述的所有五种选项，并允许程序员轻松地明确指示其程序需要的浮点语义。

对扩展精度的语言支持一定如此复杂吗？在单/双精度系统上，上面所列五种选项中的四种都一致，并且不需要区分快类型和精确宽度类型。但是，扩展精度系统带来了困难的选择：此类系统支持的纯双精度计算或纯扩展精度计算的效率不如两者的混合形式，而不同程序要求不同的混合形式。此外，使用扩展精度的选择时机不应留给编译器编写人员，他们经常受到基准的迷惑，有时数字分析员会直率地告诉他们浮点运算被视为“天生不精确”，因此整数运算既不值得预测，也无法预测。相反，选择权必须交给程序员，他们需要使用能够表达其选择的语言。

D.1.4 小结

上述评论并不是为了贬低扩展精度系统，而是揭示几个谬误。第一个谬误是所有 IEEE 754 系统针对同一程序必须产生相同的结果。我们已经着重讨论了扩展精度系统与单/双精度系统之间的区别，但在这些系统之间，每个系列仍然有进一步的差别。例如，一些单/双精度系统提供了一个指令，将两个数字相乘，然后加第三个数字，最后只进行一次舍入。此运算称为混合乘加运算，可导致同一程序在不同单/双精度系统上生成不同的结果。另外，类似于扩展精度，它甚至会导致同一程序在同一系统上生成不同结果，具体取决于是否使用此运算以及使用时间。虽然可以通过不可移植方式使用混合乘加运算来执行多精度乘法而无需拆分，但该运算还会阻挠定理 6 的拆分过程。即使 IEEE 标准未预见到此类运算，但它仍然符合要求：中间乘积提交给不受用户控制的“目标”，该目标的宽度足以精确地保存中间结果，最终的总和将正确地舍入，以符合其单精度或双精度目标。

IEEE 754 精确地规定指定程序必须提交的结果的理念仍然是非常吸引人的。许多程序员认为，他们了解程序的行为，并且可以证明程序将正确工作，而不用考虑编译该程序的编译器或运行该程序的计算机。在许多情况下，支持这一信念正是计算机系统和编程语言的设计者努力的目标。不幸的是，当遇到浮点运算时，这一目标实际上无法实现。IEEE 标准的作者知道这一点，并且没有尝试实现这一目标。因此，在整个计算机行业中，尽管几乎普遍遵守大多数 IEEE 754 标准，但可移植软件的程序员必须继续应付不可预测的浮点运算。

如果程序员打算利用 IEEE 754 的功能，他们需要使用能够使浮点运算变得可预测的编程语言。C99 标准在一定程度上改善了可预测性，代价是需要程序员编写程序的多个版本，对每个 `FLT_EVAL_METHOD` 编写一个版本。在未来的语言中，是否改为选择允许程序员编写单个程序，并且使用能够明确表示对 IEEE 754 语义的依赖程度的语法，这一点仍可期待。从现有扩展精度系统上，我们无法预见到这种远景，因为它们促使我们假定编译器和硬件比程序员更了解在给定系统上执行计算的方式。该假定是第二个谬误：计算结果中需要的精确度不依赖于产生该结果的计算机，而只依赖于从该结果得到的结论。对于程序员、编译器和硬件，最多只有程序员可能了解这些结论是什么。

♦ ♦ ♦ 附录 E

标准符合性

Oracle Developer Studio 编译器产品与 Solaris 10 操作环境中的头文件和库一起，可支持多种标准，包括 System V 接口定义第 3 版 (System V Interface Definition, SVID)、X/Open、ANSI C (C90)、POSIX.1-2001 (SUSv3) 和 ISO C (C99)。（有关更完整的讨论，请参见“标准(5)”。）其中某些标准允许在特定层面上有不同的实现。在某些情况下，这些标准的规范相互冲突。对于数学库，差异和冲突主要与特殊情况和异常有关。本附录记录了 `libm` 中的函数在这些情形下的行为，并讨论了在哪些条件下可以预期 C 程序的行为符合各个标准。本附录的最后一节记录了 Sun Studio C 和 Fortran 语言产品遵循 LIA-1 的情况。

E.1 `libm` 特殊情况

表 41 “特殊情况和 `libm` 函数”列出了所有这些情况，在这些情况中，以上涉及的两个或更多标准为 `libm` 中的函数指定了相互冲突的行为。在 C 程序上将观察到什么样的行为取决于在编译和链接程序时使用的编译器标志。可能的行为包括引发浮点异常、使用有关所发生特殊情况的信息以及返回的值调用用户提供的函数 `matherr`（请参见 *matherr*(3M)）、在标准错误文件中输出消息以及设置全局变量 `errno`（请参见 *intro*(2) 和 *perror*(3C)）。

表 41 “特殊情况和 `libm` 函数”中的第一列定义了特殊情况。第二列显示 `errno` 在设置时将设置的值。`errno` 的可能值定义在 `<errno.h>` 中；数学库中仅使用两个值，对于域错误为 `EDOM`，对于范围错误为 `ERANGE`。第二列同时显示 `EDOM` 和 `ERANGE` 时，`errno` 设置的值由相关标准确定，这些标准将在下面介绍，并显示在第四列或第五列中。第三列显示在输出的任意错误消息中指示的错误代码。第四、五、六列显示按照不同标准的定义在名义上返回的函数值。在一些情况下，用户提供的 `matherr` 例程可以覆盖这些值并提供其他返回值。

对这些特殊情况的特定响应由在链接程序时指定的编译器标志确定，如下所示。如果指定了 `-xlibmieee` 或 `-xc99=lib`，则出现表 41 “特殊情况和 `libm` 函数”中的任何特殊情况时，将引发任意适当的浮点异常，并且返回该表的第六列中列出的函数值。

如果未使用 `-xlibmieee` 和 `-xc99=lib`，则行为取决于在链接程序时指定的语言符合性标志。

指定 -Xa 标志可选择 X/Open 符合性。出现表中的任何特殊情况时，将引发任何适当的浮点异常，设置 errno，并且返回该表的第五列中列出的函数值。如果提供了用户定义的 matherr 例程，则不定义行为。请注意，在未提供任何其他语言符合性标志时，-Xa 是缺省值。

指定 -Xc 标志可选择严格 C90 符合性。出现特殊情况时，将引发任何适当的浮点异常，设置 errno，返回该表的第五列中列出的函数值。这种情况下不调用 matherr。

最后，指定 -Xs 或 -Xt 标志可选择 SVID 符合性。出现特殊情况时，将引发任意适当的浮点异常，调用 matherr，如果 matherr 返回零，则设置 errno 并返回错误消息。除非由 matherr 覆盖，否则将返回该表的第四列中列出的函数值。

有关 -xc99、-Xa、-Xc、-Xs 和 -Xt 标志的更多信息，请参见 *cc*(1) 手册页和《Oracle Developer Studio 12.5：C 用户指南》。

表 41 特殊情况和 libm 函数

函数	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	+/-HUGE1 (EDOM)	+/-HUGE_VAL2 (ERANGE)	+/-infinity
cosh overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
exp overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
exp underflow	ERANGE	-	0.0	0.0	0.0
fmod(x,0)	EDOM	DOMAIN	x	NaN	NaN
gamma(0 or -integer)	EDOM	SING	HUGE	HUGE_VAL	infinity
gamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
hypot overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
j0(X_TLOSS3<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
j1(X_TLOSS<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
jn(n,X_TLOSS<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
ldexp overflow	ERANGE	-	+-infinity	+/-infinity	+/-infinity
ldexp underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
lgamma(0 or -integer)	EDOM	SING	HUGE	HUGE_VAL	infinity
lgamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
log(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN

函数	errno	错误消息	SVID	X/Open，C90	IEEE，C99，SUSv3
log1p(−1)	EDOM/ERANGE	SING	−HUGE (EDOM)	−HUGE_VAL (ERANGE)	−infinity
log1p(x<−1)	EDOM	DOMAIN	NaN	NaN	NaN
logb(0)	EDOM	-	-HUGE_VAL	-HUGE_VAL	-infinity
nextafter overflow	ERANGE	-	+-HUGE_VAL	+/-HUGE_VAL	+/-infinity
pow(0,0)	EDOM	DOMAIN	0.0	1.0 (no error)	1.0 (no error)
pow(NaN,0)	EDOM	DOMAIN	NaN	NaN	1.0 (no error)
pow(0,x<0)	EDOM	DOMAIN	0.0	−HUGE_VAL	+/−infinity
pow(x<0, non−integer)	EDOM	DOMAIN	0.0	NaN	NaN
pow overflow	ERANGE	-	+/−HUGE	+/−HUGE_VAL	+/−infinity
pow underflow	ERANGE	-	+/−0.0	+/−0.0	+/−0.0
remainder(x,0) or remainder(inf,y)	EDOM	DOMAIN	NaN	NaN	NaN
scalb overflow	ERANGE	-	+−HUGE_VAL	+/−HUGE_VAL	+/−infinity
scalb underflow	ERANGE	-	+/−0.0	+/−0.0	+/−0.0
scalb(0,+inf) or scalb (inf,-inf)	EDOM/ERANGE	-	NaN (ERANGE)	NaN (EDOM)	NaN
scalb(|x|>0,+inf)	ERANGE	-	+−infinity	+/−infinity (no error)	+/−infinity (no error)
scalb(|x|<inf, −inf)	ERANGE	-	+/−0.0	+/−0.0 (no error)	+/−0.0 (no error)
sinh overflow	ERANGE	-	+/−HUGE	+/−HUGE_VAL	+/−infinity
sqrt(x<0)	EDOM	DOMAIN	0.0	NaN	NaN
y0(0)	EDOM	DOMAIN	−HUGE	−HUGE_VAL	−infinity
y0(x<0)	EDOM	DOMAIN	−HUGE	−HUGE_VAL	NaN
y0(X_TLOSS<x<inf)	ERANGE	TLOSS	0.0	0.0	correct answer
y1(0)	EDOM	DOMAIN	−HUGE	−HUGE_VAL	−infinity
y1(x<0)	EDOM	DOMAIN	−HUGE	−HUGE_VAL	NaN
y1(X_TLOSS<x<inf)	ERANGE	TLOSS	0.0	0.0	correct answer
yn(n,0)	EDOM	DOMAIN	−HUGE	−HUGE_VAL	−infinity
yn(n,x<0)	EDOM	DOMAIN	−HUGE	−HUGE_VAL	NaN
yn(n,X_TLOSS<x< inf)	ERANGE	TLOSS	0.0	0.0	correct answer

注：

1. HUGE 定义在 <math.h> 中。SVID 要求 HUGE 等于 MAXFLOAT，这大约为 3.4e+38。
2. HUGE_VAL 定义在 <iso/math_iso.h> 中，后者包括在 <math.h> 中。HUGE_VAL 求值为无穷。
3. X_TLOSS 定义在 <values.h> 中。

E.1.1 影响标准符合性的其他编译器标志

上面列出的编译器标志直接选择在处理表 41 “特殊情况和 libm 函数”中列出的特殊情况时应遵守哪几个标准。其他编译器标志可以间接影响是否会在程序上观察到上述行为。

首先，-xlibmil 和 -xlibmopt 标志替换 libm 中一些函数的更快的实现。这些更快的实现不符合 SVID、X/Open 或 C90。它们不设置 errno，也不调用 matherr。但是，它们会相应引发浮点异常，并提供 IEEE 754 和/或 C99 规定的结果。类似的注释适用于 -xvector 标志，因为它会导致编译器将对标准数学函数的调用转换为对向量数学函数的调用。

其次，-xbuiltin 标志允许编译器将 <math.h> 中定义的标准数学函数作为内部函数处理，并且替换内联代码以实现更好的性能。替换代码可能不符合 SVID、X/Open、C90 或 C99。它不需要设置 errno、调用 matherr 或引发浮点异常。

第三，在定义 C 预处理程序标记 __MATHERR_ERRNO_DONTCARE 的情况下，将编译 <math.h> 中的多条 #pragma 指令。这些指令告知编译器假设标准数学函数没有任何副作用。根据这一假设，编译器可以重新排序对数学函数的调用，并且引用全局数据（例如 errno）或可能由用户提供的 matherr 例程修改的数据，这样可以改变上面介绍的预期行为。例如，请考虑代码片段：

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

...
errno = 0;
x = acos(2.0);
if (errno) {
    printf(“error\n”);
}
```

如果在定义了 __MATHERR_ERRNO_DONTCARE 的情况下编译此代码，则编译器可能会假设 errno 未由对 acos 的调用修改，并相应地转换代码，完全删除对 printf 的调用。

请注意，-fast 宏标记包括标志 -xbuiltin、-xlibmil、-xlibmopt 和 -D__MATHERR_ERRNO_DONTCARE。

最后，由于 libm 中的所有数学函数会根据需要引发浮点异常，运行允许捕获这些异常的程序时，通常会导致超出以上所列标准的规定的行为。因此，-ftrap 编译器标志还会影响标准符合性。

E.1.2 关于 C99 符合性的补充说明

C99 指定了两种可能的方法，实现会按照这两种方法来处理诸如表 41 “特殊情况和 libm 函数”中的特殊情况。实现可通过定义标识符 math_errhandling 来计算具有值 MATH_ERRNO (1)、MATH_ERREXCEPT (2) 或者这两个值的按位“或”的整数表达式，以此指示支持两种方法中的哪一个。（这些值定义在 <math.h> 中。）如果表达式

(math_errhandling & MATH_ERRNO) 非零，则实现可通过将 errno 设置为 EDOM 来处理函数参数位于其数学域之外的情况，通过将 errno 设置为 ERANGE 来准确地处理函数的结果值可能下溢、溢出或等于无穷的情况。如果表达式 (math_errhandling & MATH_ERREXCEPT) 非零，则实现可通过引发无效操作异常来处理函数参数位于其数学域之外的情况，通过引发下溢、溢出或被零除来准确地处理函数的结果值可能下溢、溢出或等于无穷的情况。

在 Oracle Solaris 上，<math.h> 将 math_errhandling 定义为 MATH_ERREXCEPT。虽然表 41 “特殊情况和 libm 函数”中列出的函数可以为特殊情况执行此处列出之外的其他操作，所有 libm 函数（包括 float 和 long double 函数、复数函数和 C99 指定的其他函数）通过引发浮点异常来响应特殊情况。这是所有 C99 函数统一支持处理特殊情况的唯一方法。

最后请注意，有三个函数，不论是 C99 还是 SUSv3，都需要不同于 Oracle Solaris 缺省值的行为。下表汇总了这些区别。表中仅列出了各函数的 double 版本，但是这些区别也适用于 float 和 long double 版本。在各种情况下，使用 -xc99=lib 链接程序时将遵循 SUSv3 规范，否则将遵循 Solaris 缺省值。

表 42 Solaris 和 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 1
	pow(-1.0, +/-inf) 返回 NaN	pow(-1.0, +/-inf) 返回 1
	pow(1.0, NaN) 返回 NaN	pow(1.0, NaN) 返回 1
logb	logb(subnormal) 返回 Emin	logb(x) = ilogb(x) (x 为次正规数时)
ilogb	ilogb(+/-0), ilogb(+/-inf),	ilogb(+/-0), ilogb(+/-inf),
	ilogb(NaN) 不引发异常	ilogb(NaN) 引发无效操作

E.2 LIA-1 符合性

在本节中，LIA-1 指的是 ISO/IEC 10967-1:1994 Information Technology - Language Independent Arithmetic - Part 1: Integer and floating-point arithmetic（ISO/IEC 10967-1:1994 信心技术 - 语言独立技术 - 第 1 部分：整数和浮点计算）。

Sun Studio 编译器发行版中包含的 C 和 Fortran 95 编译器（cc 和 f95），在以下几个方面符合 LIA-1（段落字母编号与 LIA-1 第 8 节对应）：

E.2.1 a.类型 (LIA 5.1)：

LIA-1 符合标准的类型为 C int 和 Fortran INTEGER。也可以遵从其他类型，但此处未指明。特定语言的更多规范尚待审理语言标准组织将语言结合到 LIA-1。

E.2.2 b.参数 (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)
```

E.2.3 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)` 提供了以下代码：

```
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;
 }
```

还提供了以下代码：

```
 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
```

E.2.4 i.表示法 (LIA 5.1.3)：

下表显示了可以实现 LIA 整数运算的表示法。

表 43 LIA-1 符合性 – 表示法

LIA	C	Fortran（如有不同）
addI(x,y)	x+y	n/a
subI(x,y)	x-y	n/a
mulI(x,y)	x*y	n/a
divtI(x,y)	x/y	n/a
remtI(x,y)	x%y	mod(x,y)
modaI(x,y)	see above	n/a
negI(x)	-x	n/a

absI(x)	#include <stdlib.h>
	abs(x)
		abs(x)

signI(x)	#define signI(x) (x > 0 ? 1 : (x < 0 ? -1 : 0))	请参见下文
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

以下代码显示了 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
```

E.2.5 j.表达式求值：

缺省情况下，在未指定任何优化时，表达式按照 int (C) 或 INTEGER (Fortran) 精度求值。建议使用括号。未使用括号的联合表达式的求值顺序未指定，例如 a + b + c 或 a * b * c。

E.2.6 k.获取参数的方法：

将第 E.2.2 节 “b.参数 (LIA 5.1)：” [160] 中的定义包含在您的源代码中。

E.2.7 n.通知：

整数异常为 x/0 和 x%0 或 mod(x,0)。缺省情况下，这些异常生成 `SIGFPE`。没有为 `SIGFPE` 指定信号处理程序时，进程将会终止并且转储内存。

E.2.8 o.选择机制：

`signal`(3) 或 `signal`(3F) 可用于启用 `SIGFPE` 的用户异常处理。

♦ ♦ ♦ 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：

- UltraSPARC Architecture 2005 (base ISA for T1) (http://www.oracle.com/technetwork/systems/opensparc/1537734)（UltraSPARC 体系结构 2005（适用于 T1 的基本 ISA））
- UltraSPARC Architecture 2007 (base ISA for T2, T2+, T3) (http://www.oracle.com/technetwork/systems/hardware/usparcarchdoc2007-329425.pdf)（UltraSPARC 体系结构 2007（适用于 T2、T2+、T3 的基本 ISA））
- Oracle SPARC Architecture 2011 (base ISA for T4, T5, M5, M6) (http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf)（Oracle SPARC 体系结构 2011（适用于 T4、T5、M5、M6 的基本 ISA）

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将在最后介绍。

F.1 第 2 章：“IEEE 算法”

由 Cody et al. 编著的《A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic》，IEEE Computer 出版，1984 年 8 月。

由 Coonen, J.T. 编著的《An Implementation Guide to a Proposed Standard for Floating Point Arithmetic》，IEEE Computer 出版，第 13 卷 1 号，1980 年 1 月，第 68-79 页。

由 Demmel, J. 编著的《Underflow and the Reliability of Numerical Software》，由 SIAM J. 编著的《*Scientific Statistical Computing*》，第 5 卷 (1984)，第 887-919 页。

由 Hough, D. 编著的《Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic》，IEEE Computer 出版，第 13 卷 1 号，1980 年 1 月，第 70-74 页。

由 Kahan, W. 和 Coonen, J.T. 合著的《The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments》，发表于《The

Relationship between Numerical Computation and Programming Languages》，Reid, J.K. 编辑，North-Holland Publishing Company 出版，1982。

由 Kahan, W. 编著的《Implementation of Algorithms》，发表于《Computer Science Technical Report No. 20》，University of California, Berkeley CA，1973。可从美国国家技术情报社出版获得，NTIS 文档编号 AD–769 124（339 页），1-703-487-4650（普通订单）或 1-800-336-4700（紧急订单）。

由 Karpinski, R.编著的《Paranoia: a Floating-Point Benchmark》，发表于《Byte》杂志，1985 年 2 月。

由 Knuth, D.E. 编著的《The Art of Computer Programming》第 2 卷："Semi-Numerical Algorithms, Addison-Wesley, Reading, Mass"，1969，第 195 页。

由 Linnainmaa, S.编著的《Combatting the effects of Underflow and Overflow in Determining Real Roots of Polynomials》，SIGNUM Newsletter 16 (1981)，第 11-16 页。

由 Rump, S.M. 编著的《How Reliable are Results of Computers?》，《Wie zuverlassig sind die Ergebnisse unserer Rechenanlagen?》的译本，Jahrbuch Uberblicke Mathematik 1983，第 163-168 页，C Bibliographisches Institut AG 1984。

由 Sterbenz, P 编著的《Floating-Point Computation》，Prentice-Hall 出版，Englewood Cliffs, NJ，1974。（已绝版；大多数大学图书馆有复印本。）

由 Stevenson, D. et al.、Cody, W.、Hough, D.、Coonen, J. 编著的用于建议和分析二进制浮点算法的标准草案的各种论文，IEEE Computer 出版，1981 年 3 月。

《The Proposed IEEE Floating-Point Standard》，发表于《ACM SIGNUM Newsletter》的特刊中，1979 年 10 月。

F.2 第 3 章："数学库"

由 Cody, William J. 和 Waite, William 合著的《Software Manual for the Elementary Functions》，Prentice-Hall, Inc. 出版，Englewood Cliffs，New Jersey，07632，1980。

由 Coonen, J.T. 编著的《Contributions to a Proposed Standard for Binary Floating-Point Arithmetic》，PhD Dissertation 出版，University of California, Berkele，1984。

由 Tang, Peter Ping Tak 编著的《Some Software Implementations of the Functions Sin and Cos》，发表于《Technical Report ANL-90/3》，Mathematics and Computer Science Division，Argonne National Laboratory，Argonne，Illinois，1990 年 2 月。

由 Tang, Peter Ping Tak 编著的《Table-driven Implementations of the Exponential Function EXPM1 in IEEE Floating-Point Arithmetic》，MCS-P125-0290 预印本，Mathematics and Computer Science Division，Argonne National Laboratory，Argonne，Illinois，1990 年 2 月。

由 Tang, Peter Ping Tak 编著的《Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic》，发表于《ACM Transactions on Mathematical Software》，第 15 卷 2 号，1989 年 6 月，第 144-157 页，Communications of the ACM 出版，1988 年 7 月 18 日。

由 Tang, Peter Ping Tak 编著的《Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic》，MCS-P55-0289 预印版，Mathematics and Computer Science Division，Argonne National Laboratory，Argonne，Illinois，1989 年 2 月（发表于《ACM Trans. on Math.Soft》

由 Park, Stephen K. 和 Miller, Keith W. 合著的《Random Number Generators: Good Ones Are Hard To Find》，Communications of the ACM 出版，第 31 卷 10 号，1988 年 10 月，第 1192-1201 页。

F.3 第 4 章："异常和异常处理"

由 Coonen, J.T 编著的《Underflow and the Denormalized Numbers》，Computer 出版，14，3 号，1981 年 3 月，第 75-87 页。

由 Demmel, J. 和 X.Li 编著的《Faster Numerical Algorithms via Exception Handling》，IEEE Trans.Comput. 出版第 48 卷 8 号，1994 年 8 月，第 983-992 页。

由 Kahan, W. 编著的《A Survey of Error Analysis》，"Information Processing 71"，North-Holland，Amsterdam，1972，第 1214-1239 页。

F.4 标准

American National Standard for Information Systems ISO/IEC 9899:1999 Programming Languages (C99)，American National Standards Institute, 1430 Broadway, New York, NY 10018。

IEEE Standard for Floating-Point Arithmetic，ANSI/IEEE Std 754-2008，The Institute of Electrical and Electronics Engineers, Inc. 出版，3 Park Avenue, New York, NY 10016, 2008。

IEEE Standard Glossary of Mathematics of Computing Terminology，ANSI/IEEE Std 1084-1986，The Institute of Electrical and Electronics Engineers, Inc. 出版，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. 出版, 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 或 SVID Issue 3），第 I–IV 卷，部件号 320-135，AT&T (1-800-432-6600)，1989。

X/OPEN Portability Guide，一套 7 卷，Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1989。

F.5 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 测试向量和基于由 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`。

词汇表

A-E

abrupt underflow（突然下溢） 浮点运算下溢时始终返回零，即使结果处于次正规数的范围内。

accuracy（精确度） 一个数字与另一个数字的近似程度的衡量标准。例如，计算结果的精确度通常反映了导致该结果与数学精确结果之差的计算误差程度。精确度可以使用有效位数（例如，“结果精确到六位”）来表示，或者更常使用保留的相关数学属性（例如，“结果具有正确的代数符号”）来表示。

biased exponent（偏置指数） 底数为 2 的指数和一个常量（偏置）之和，选择该常量的目的是为了使存储的指数范围为非负值。例如 2^{-100} 的指数，以 IEEE 单精度格式存储为 (-100) +（单精度偏置 127）= 27。

binade 任意两个连续二次幂之间的间隔。

chaining（链接） 一些管道体系结构的硬件功能，允许将一个运算的结果立即用作第二个运算的操作数，并同时将结果写入其目标寄存器。两个链接运算的总周期时间小于指令的独立周期时间之和。例如，TI 8847 支持连续的 `fadd`、`fsub` 和 `fmul`（相同精度）链接。链接的 `faddd`/`fmuld` 需要 12 个周期，而连续的未链接 `faddd`/`fmuld` 需要 17 个周期。

common exceptions（常见异常） 对于 `ieee_flags`(3m) 和 `ieee_handler`(3m) 来说，溢出、无效以及被零除这三种浮点异常统称为常见异常。由于这些异常通常作为错误捕获，因此称为常见异常。

context switch（上下文切换） 在多任务操作系统（例如 Oracle Solaris 操作系统）中，进程按照固定的时间量程运行。在时间量程结束时，CPU 接收计时器的信号，中断当前正在运行的进程，然后准备运行新进程。CPU 保存旧进程的寄存器，然后为新进程加载寄存器。从旧进程状态切换到新进程的过程称为上下文切换。切换上下文花费的时间属于系统开销；所需的时间取决于寄存器的数量，以及是否有特殊指令用于保存与进程关联的寄存器。

default result（缺省结果） 在没有为异常指定任何其他处理的情况下，提供作为导致此异常的浮点运算结果的值。

denormalized number（非规范化数字） 次正规数以前的命名规则。

double precision（双精度） 使用两个字来表示数字，以确保精度或增加精度。在 SPARC® 工作站上，双精度是 64 位 IEEE 双精度。

exception（异常） 在尝试运行的原子算术运算没有产生可普遍接受的结果时，将引发算术异常。原子和可接受的含义随时间和使用场合而异。

exponent（指数） 浮点数的一部分，指示在确定所表示数字的值时基数的整数次幂。

F-I

floating-point number system（浮点数系统） 表示实数子集的系统，其中可表示数字之间的间隔不是固定的绝对常量。此类系统的特征包括基数、符号、有效位和指数（通常偏置）。数字的值是其有效位与基数的幂（未偏置指数）的有符号乘积。

gradual underflow（渐进下溢） 当浮点运算下溢到次正规数范围中时，将返回次正规数而不是零。这种处理下溢的方法能够尽可能减少在对小数字进行浮点计算时的精确度损失。

hidden bit（隐藏位） 硬件使用的额外位，用于确保正确的舍入，软件无法访问。例如，IEEE 双精度运算使用三个隐藏位来计算 56 位结果，这些结果随后会舍入到 53 位。

IEEE Standard 754（IEEE 标准 754） 二进制浮点算术的标准，由美国电气及电子工程师学会 (Institute of Electrical and Electronics Engineers, IEEE) 开发，1985 年出版，2008 年修订。

in-line template（内联模板） 汇编语言代码片段，在内联传递 Oracle Developer Studio 编译器时，替换它所定义的函数调用。（例如）由数学库在内联模板文件 (`libm.il`) 中使用，以便从 C 程序访问三角函数的硬件实现和其他初等函数。

L-P

NaN 表示非数。使用浮点格式编码的符号实体。

normal number（正规数） IEEE 运算中一个具有偏置指数的数字，不是零，也不是最大值（全为 1），表示实数常规范围的子集，具有相对较小的限定误差。

pipelining（流水线作业） 一个将运算约简为多个阶段的硬件功能，每个阶段（通常）用一个周期来完成。如果每个周期中可以发布新操作，将填充管道。如果管道中的指令之间没有相关性，则每个周期可以提供新结果。链接表示相

关指令的流水线作业。如果相关指令无法链接，则在硬件不支持链接这些特定指令时，管道会停止。

precision（精度） 可表示数字的密度的定量测量方法。例如，在精度为 53 个有效位的二进制浮点格式中，任意两个相邻的 2 的幂之间有 253 个可表示数字（在正规数的范围内）。请不要混淆精度与精确度，后者表示的是一个数字与另一个数字的近似程度。

Q-R

quiet NaN（静态 NaN） 通过几乎每个算术运算传播的 NaN（not a number，非数），不引发新异常。

radix（基数） 任何数字系统的基数。例如，在记数系统中，2 是二进制系统的基数，10 是十进制系统的基数。SPARC 工作站使用基数 2 运算；IEEE 标准 754 是基数 2 运算标准。

round（舍入） 不精确的结果必须向上或向下舍入以获得可表示的值。向上舍入结果时，会将它增大到后一个可表示值。向下舍入结果时，会将它减小到前一个可表示值。

roundoff error（舍入误差） 将实数舍入到计算机可表示的数字时带来的误差。大部分浮点计算都会产生舍入误差。对于任何浮点操作，IEEE 标准 754 指定结果不应导致多个舍入误差。

S-T

signaling NaN（信号 NaN） 只要作为操作数出现就会引发无效运算异常的 NaN（not a number，非数）。

significand（有效位） 浮点数的一部分，与基数的有符号次幂相乘来确定数字的值。在规范化数字中，有效位包括一个位于基数点左侧的非零位，和一个位于右侧的小数部分。

single precision（单精度） 使用一个计算机字来表示一个数字。

stderr 标准误差是指向标准误差输出的 Unix 文件指针。在启动程序时打开此文件。

store 0（存储零） 与突然下溢相同。请参见 *abrupt underflow*（突然下溢）。

subnormal number（次正规数） 在 IEEE 运算中，偏置指数为零的非零浮点数。次正规数是介于零和最小正规数之间的数字。

two's complement（二进制补码）	二进制数字的基数补码，通过将每一位减去一，然后在最低有效位上加一并执行任何所需的进位后形成。例如，1101 的二进制补码为 0011。

U-Z

ulp	表示最后位置的单元。在二进制格式中，有效位的最低有效位，即位 0 是最后位置单元。
ulp(x)	表示以工作模式截断的 x 的 `ulp`。
underflow（下溢）	表示一种情形，此时浮点算术运算的结果太小，使其在只使用正规舍入的方法时，无法以目标浮点格式中的正规数形式表示。
word（字）	一组有序的字符，在指定计算机中作为单个实体存储、寻址、传输和操作。在 SPARC 工作站中上下文中，一个字为 32 位。
wrapped number（封装数字）	在 IEEE 运算中，从一个值创建的数字，通过向其指数添加固定偏置使该封装值处于正规数范围内，否则会溢出或下溢。当前不在 SPARC 工作站上生成封装结果。

索引

W

X

Z