



# 数值计算指南

---

Sun™ Studio 8

Sun Microsystems, Inc.  
www.sun.com

部件号码 817-5810-10  
2004 年 4 月, 修订 A

有关本档的建议请发到: <http://www.sun.com/hwdocs/feedback>

版权所有 © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. 保留所有权利。

美国政府权利 — 商业软件。政府用户在使用时应遵循 Sun Microsystems, Inc. 的标准许可协议和 FAR 及其补充中的适用规定。使用本软件应遵守许可证条款。

此次分发可能包含由第三方开发的内容。

该产品的部分内容可能出自 Berkeley BSD 系统，由加州大学 (University of California) 授权。UNIX 是在美国和其它国家（地区）的注册商标，由 X/Open Company, Ltd. 独家授权。

Sun、Sun Microsystems、Sun 徽标、Java、JavaHelp 是 Sun Microsystems, Inc. 在美国和其它国家（地区）的商标或注册商标。所有的 SPARC 商标均需获得授权才能使用，它们是 SPARC International, Inc. 在美国和其它国家（地区）的商标或注册商标。标有 SPARC 商标的产品都基于由 Sun Microsystems, Inc. 开发的体系结构。

该产品为美国的出口控制法所涵盖和控制，并有可能要遵守其它国家（地区）的出口或进口法规。严禁将产品用于核、导弹、生化武器或核海事等最终目的或交给有此类企图的最终用户，无论是直接还是间接。严禁将产品出口或再次出口到美国禁运的国家（地区）或美国出口排除列表中的实体，包括但不限于被拒绝的人员以及专门指定的公民列表。

本文档按“原样”提供，对所有明示或默示的条件、陈述和担保，包括对适销性、特殊用途的适用性或非侵权性的默示保证，均不承担任何责任，除非此免责声明的适用范围在法律上无效。

---



# 目录

---

## 开始之前 13

本书面向的读者 13

本书的组织结构 13

排版惯例 14

Shell 提示符 15

访问 Sun Studio 软件和手册页 16

访问编译器和工具文档 18

访问相关 Solaris 文档 19

开发人员资源 20

与 Sun 技术支持联系 20

Sun 欢迎您提出意见和建议 20

## 1. 简介 21

浮点环境 21

## 2. IEEE 算法 23

IEEE 算法模型 23

    什么是 IEEE 算法? 23

IEEE 格式 24

    存储格式 24

|                     |    |
|---------------------|----|
| 单精度格式               | 25 |
| 双精度格式               | 27 |
| 双精度扩展格式 (SPARC)     | 29 |
| 双精度扩展格式 (x86)       | 31 |
| 十进制表示中的范围和精度        | 35 |
| Solaris 环境中的基数转换    | 38 |
| 下溢                  | 39 |
| 下溢阈值                | 39 |
| IEEE 算法如何处理下溢?      | 40 |
| 为什么使用渐进下溢?          | 40 |
| 渐进下溢的误差特性           | 41 |
| 两个渐进下溢与 Store 0 的示例 | 43 |
| 下溢方法是否重要?           | 44 |

### 3. 数学库 45

标准数学库 45

附加的数学库 47

    Sun 数学库 47

    优化库 48

    矢量数学库 (仅 SPARC) 49

    libm9x 数学库 50

单、双和长双精度 52

IEEE 支持函数 53

    ieee\_functions(3m) 和 ieee\_sun(3m) 53

    ieee\_values(3m) 55

    ieee\_flags(3m) 57

    ieee\_retrospective(3m) 59

    nonstandard\_arithmetic(3m) 60

C99 浮点环境函数 61

|                         |            |
|-------------------------|------------|
| 异常标记函数                  | 61         |
| 舍入控制                    | 62         |
| 环境函数                    | 63         |
| libm 和 libsunmath 的实现功能 | 65         |
| 关于算法                    | 65         |
| 三角函数的参数缩小               | 65         |
| 数据转换例程                  | 66         |
| 随机数工具                   | 66         |
| <b>4. 异常和异常处理</b>       | <b>69</b>  |
| 什么是异常?                  | 69         |
| 表 4-1 的注释               | 71         |
| 检测异常                    | 72         |
| ieee_flags(3m)          | 72         |
| C99 异常标志函数              | 75         |
| 查找异常                    | 76         |
| 使用调试器查找异常               | 76         |
| 使用信号处理程序来查找异常           | 85         |
| 使用 libm9x.so 异常处理扩展查找异常 | 90         |
| 处理异常                    | 97         |
| <b>A. 示例</b>            | <b>111</b> |
| IEEE 算法                 | 111        |
| 数学库                     | 113        |
| 随机数生成器                  | 113        |
| IEEE 建议的函数              | 116        |
| IEEE 特殊值                | 120        |
| ieee_flags - 舍入方向       | 122        |
| C99 浮点环境函数              | 124        |

异常和异常处理 128

ieee\_flags —产生的异常 128

ieee\_handler —捕获异常 131

ieee\_handler —出现异常时终止 140

libm9x.so 异常处理功能 141

将 libm9x.so 用于 Fortran 程序 147

杂项 151

sigfpe —捕获整数异常 151

从 C 中调用 Fortran 152

有用的调试命令 156

## **B. SPARC 行为和实现 161**

浮点硬件 161

浮点状态寄存器和队列 164

需要软件支持的特殊类 166

fpversion(1) 函数 — 查找有关 FPU 的信息 170

## **C. x86 行为和实现 171**

## **D. What Every Computer Scientist Should Know About Floating-Point Arithmetic 173**

摘要 173

简介 174

舍入误差 174

浮点格式 175

相对误差和 Ulp 176

保护数位 177

抵消 178

精确舍入的运算 182

IEEE 标准 186

|                   |            |
|-------------------|------------|
| 格式与运算             | 186        |
| 特殊数量              | 190        |
| NaN               | 191        |
| 异常、标志和陷阱处理程序      | 195        |
| 系统方面              | 199        |
| 指令集               | 200        |
| 语言和编译器            | 201        |
| 异常处理              | 207        |
| 详细资料              | 209        |
| 舍入误差              | 209        |
| 二进制到十进制的转换        | 217        |
| 求和中的误差            | 218        |
| 小结                | 220        |
| 致谢                | 220        |
| 参考书目              | 220        |
| 定理 14 和定理 8       | 223        |
| 定理 14             | 223        |
| 证明                | 223        |
| IEEE 754 各实现之间的差异 | 227        |
| 当前的 IEEE 754 实现   | 228        |
| 在基于扩展的系统上的计算缺陷    | 229        |
| 扩展精度的程序设计语言支持     | 233        |
| 结束语               | 236        |
| <b>E. 符合标准</b>    | <b>239</b> |
| SVID 历史           | 239        |
| IEEE 754 历史       | 240        |
| SVID 将来的发展方向      | 240        |
| SVID 实现           | 240        |

|                      |            |
|----------------------|------------|
| 有关异常情况和 libm 函数的一般说明 | 242        |
| libm 的说明             | 243        |
| 符合 LIA-1             | 244        |
| <b>F. 参考资料</b>       | <b>247</b> |
| 第 2 章：“IEEE 算法”      | 247        |
| 第 3 章：“数学库”          | 248        |
| 第 4 章：“异常和信号处理”      | 249        |
| 附录 B：“SPARC 行为和实现”   | 249        |
| 标准                   | 250        |
| 测试程序                 | 250        |
| <br>                 |            |
| 词汇表                  | 251        |
| <br>                 |            |
| 索引                   | 263        |



# 图

- 
- 图 2-1 单精度存储格式 25
- 图 2-2 双精度存储格式 27
- 图 2-3 双精度扩展格式 (SPARC) 30
- 图 2-4 双精度扩展格式 (x86) 32
- 图 2-5 比较由十进制和二进制表示所定义的一组数 36
- 图 2-6 实数值线 42
- 图 B-1 SPARC 浮点状态寄存器 165
- 图 D-1  $\beta = 2$ 、 $p = 3$ 、 $e_{\min} = -1$ 、 $e_{\max} = 2$  时正规化的数 176
- 图 D-2 清零与渐进下溢的比较 194



# 表

---

|        |                       |    |
|--------|-----------------------|----|
| 表 2-1  | IEEE 格式和语言类型          | 25 |
| 表 2-2  | IEEE 单精度格式位模式表示的值     | 26 |
| 表 2-3  | 单精度存储格式位模式及其 IEEE 值   | 26 |
| 表 2-4  | IEEE 双精度格式位模式表示的值     | 28 |
| 表 2-5  | 双精度存储格式位模式及其 IEEE 值   | 29 |
| 表 2-6  | 位模式表示的值 (SPARC)       | 30 |
| 表 2-7  | 双精度扩展格式位模式 (SPARC)    | 31 |
| 表 2-8  | 位模式表示的值 (x86)         | 32 |
| 表 2-9  | 双精度扩展格式位模式及它们的值 (x86) | 34 |
| 表 2-10 | 存储格式的范围和精度            | 38 |
| 表 2-11 | 下溢阈值                  | 39 |
| 表 2-12 | 4 种不同精度的 $ulp(1)$     | 41 |
| 表 2-13 | 可表示的单精度格式浮点数之间的差值     | 42 |
| 表 3-1  | libm 的内容              | 46 |
| 表 3-2  | libsunmath 的内容        | 47 |
| 表 3-3  | libmvec 的内容           | 50 |
| 表 3-4  | libm9x 的内容            | 51 |
| 表 3-5  | 调用单、双和四倍精度函数          | 52 |
| 表 3-6  | ieee_functions(3m)    | 53 |
| 表 3-7  | ieee_sun(3m)          | 53 |

|        |   |     |
|--------|---|-----|
| 表 3-8  | 从 Fortran 中调用 <code>ieee_functions</code> | 54  |
| 表 3-9  | 从 Fortran 中调用 <code>ieee_sun</code>       | 54  |
| 表 3-10 | IEEE 值: 单精度                               | 55  |
| 表 3-11 | IEEE 值: 双精度                               | 55  |
| 表 3-12 | IEEE 值: 四倍精度 (SPARC)                      | 56  |
| 表 3-14 | <code>ieee_flags</code> 的参数值              | 57  |
| 表 3-13 | IEEE 值: 双扩展精度 (x86)                       | 57  |
| 表 3-15 | <code>ieee_flags</code> 舍入方向的输入值          | 58  |
| 表 3-16 | C99 标准异常标记函数                              | 62  |
| 表 3-17 | <code>libm9x.so</code> 浮点环境函数             | 63  |
| 表 3-18 | 单值随机数生成器的区间                               | 66  |
| 表 4-1  | IEEE 浮点异常                                 | 70  |
| 表 4-2  | 无序比较                                      | 71  |
| 表 4-3  | 异常位                                       | 74  |
| 表 4-4  | 算术异常的类型                                   | 87  |
| 表 4-5  | <code>fex_set_handling</code> 的异常代码       | 91  |
| 表 A-1  | 一些调试命令 (SPARC)                            | 156 |
| 表 A-2  | 一些调试命令 (x86)                              | 157 |
| 表 B-1  | SPARC 浮点选项                                | 162 |
| 表 B-2  | 浮点状态寄存器字段                                 | 165 |
| 表 B-3  | 异常处理字段                                    | 165 |
| 表 D-1  | IEEE 754 格式参数                             | 187 |
| 表 D-2  | IEEE 754 特殊值                              | 190 |
| 表 D-3  | 产生 NaN 的运算                                | 191 |
| 表 D-4  | IEEE 754* 中的异常                            | 196 |
| 表 E-1  | 异常情况和 <code>libm</code> 函数                | 240 |
| 表 E-2  | 符合 LIA-1 - 表示法                            | 245 |

# 开始之前

---

本手册介绍运行 Solaris™ 操作环境的 SPARC® 和 x86 平台上的软件和硬件所支持的浮点环境。虽然本手册讨论了 SPARC 和 Intel 体系结构的一些基本内容，但它主要是作为随 Sun™ 语言产品提供的参考手册。

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

---

## 本书面向的读者

本手册是为那些开发、维护和移植数学及科学应用程序或基准的人员编写的。在使用本手册之前，您应该熟悉所使用的编程语言（Fortran、C 等）、dbx（源代码级调试器）以及操作系统命令和概念。

---

## 本书的组织结构

第 1 章介绍浮点环境。

第 2 章介绍 IEEE 算法模型、IEEE 格式和下溢。

第 3 章介绍 Sun™ Studio 编译器提供的数学库。

第 4 章介绍异常并说明如何检测、查找和处理它们。

附录 A 包含示例程序。

附录 B 介绍 SPARC 工作站的浮点硬件选项。

附录 C 列出了与 Intel 系统中使用的浮点单元有关的 x86 和 SPARC 兼容性问题。

附录 D 是 David Goldberg 编写的浮点算法教程的编辑重印。

附录 E 讨论标准遵循性。

附录 F 中包含参考资料和相关文档的列表。

词汇表包含术语的定义。

本手册中的示例是使用 C 和 Fortran 编写的，但概念则适用于 SPARC 或 Intel 系统上的编译器。

---

## 排版惯例

表 P-1 字体惯例

| 字体                     | 含义                   | 示例  |
|------------------------|----------------------|---|
| AaBbCc123              | 命令、文件以及目录的名称；计算机屏幕输出 | 编辑您的 .login 文件。<br>使用 <code>ls -a</code> 列出所有文件。<br>% You have mail.          |
| <b>AaBbCc123</b>       | 键入的内容，以便与计算机屏幕输出相区别  | % <b>su</b><br>Password:  |
| <i>AaBbCc123</i>       | 书名、新词或术语以及要强调的词      | 请阅读《 <i>用户指南</i> 》中的第 6 章。<br>这些被称为 <i>类</i> 选项。<br>您 <i>必须</i> 是超级用户才能执行此操作。 |
| <code>AaBbCc123</code> | 命令行占位符文本；用实际的名称或值替换  | 要删除文件，请键入 <code>rm filename</code> 。  |

表 P-2 代码惯例

| 代码符号 | 含义                            | 表示法                 | 代码示例               |
|------|-------------------------------|---------------------|--------------------|
| [ ]  | 方括号包含可选的参数。                   | O[n]                | -O4, -O            |
| { }  | 大括号包含所需选项的一组选项。               | d{y n}              | -dy                |
|      | “管道”或“竖线”符号用于分隔参数，表示只能选择其中一个。 | B{dynamic static}   | -Bstatic           |
| :    | 与逗号一样，冒号有时也用于分隔参数。            | Rdir[:dir]          | -R/local/libs:/U/a |
| ...  | 省略号表示省略了序列中的一部分。              | -xinline=f1[,...fn] | -xinline=alpha,dos |

## Shell 提示符

| Shell                          | 指示符                   |
|--------------------------------|-----------------------|
| C shell                        | <i>machine-name</i> % |
| C shell 超级用户                   | <i>machine-name</i> # |
| Bourne shell 和 Korn shell      | \$                    |
| Bourne shell 和 Korn shell 超级用户 | #                     |

---

## 访问 Sun Studio 软件和手册页

编译器和工具以及它们手册页没有安装到标准的 `/usr/bin/` 和 `/usr/share/man` 目录中。要访问编译器和工具，您必须正确设置 `PATH` 环境变量（请参见 16 页上的“访问编译器和工具”）。要访问手册页，您必须正确设置 `MANPATH` 环境变量（请参见 17 页上的“访问手册页”）。

有关 `PATH` 变量的详细信息，请参见 `csh(1)`、`sh(1)` 和 `ksh(1)` 手册页。有关 `MANPATH` 变量的详细信息，请参阅 `man(1)` 手册页。有关如何设置 `PATH` 变量和 `MANPATH` 变量以访问此版本的详细信息，请参阅安装指南或咨询系统管理员。

---

**注意** – 本节中的信息假定您已将 Sun Studio 编译器和工具安装在 `/opt` 目录中。如果软件不是安装在 `/opt` 目录中，请向系统管理员询问系统上的等效路径。

---

### 访问编译器和工具

请使用以下步骤确定您是否需要更改 `PATH` 变量以访问编译器和工具。

#### ▼ 确定您是否需要设置 `PATH` 环境变量

1. 通过在命令提示符下键入以下内容来显示当前的 `PATH` 变量值。

```
% echo $PATH
```

2. 检查输出以查找包含 `/opt/SUNWspro/bin/` 的路径字符串。

如果找到了该路径，则说明已设置了 `PATH` 变量来访问编译器和工具。如果没有找到该路径，则需要按照以下步骤中的说明来设置 `PATH` 环境变量。

#### ▼ 设置 `PATH` 环境变量以启用对编译器和工具的访问

1. 如果使用的是 **C shell**，请编辑主目录中的 `.cshrc` 文件。如果使用的是 **Bourne shell** 或 **Korn shell**，请编辑主目录中的 `.profile` 文件。
2. 将以下内容增加到 `PATH` 环境变量中。如果您已经安装了 **Sun ONE Studio** 软件或 **Forte Developer** 软件，则请在这些安装路径之前添加以下路径。

```
/opt/SUNWspro/bin
```



## 访问手册页

请通过以下步骤确定您是否需要更改 `MANPATH` 变量以访问手册页。

### ▼ 确定您是否需要设置 `MANPATH` 环境变量

1. 通过在命令提示符下键入以下内容来请求访问 `dbx` 手册页。

```
% man dbx
```

2. 查看输出（如果有的话）。

如果找不到 `dbx(1)` 手册页，或者显示的手册页不用于当前安装的软件版本，那么请按照以下步骤中有关设置 `MANPATH` 环境变量的说明进行操作。

### ▼ 设置 `MANPATH` 环境变量以启用对手册页的访问

1. 如果使用的是 **C shell**，请编辑主目录中的 `.cshrc` 文件。如果使用的是 **Bourne shell** 或 **Korn shell**，请编辑主目录中的 `.profile` 文件。
2. 将以下内容增加到 `MANPATH` 环境变量中。

```
/opt/SUNWspro/man
```

## 访问集成开发环境

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

此 IDE 要求 Sun Studio 8 核心平台组件。如果核心平台组件并未安装或装配在下列位置之中，您就必须为核心平台组件的安装或装配位置

(*installation\_directory/netbeans/3.5R*) 设置 `SPRO_NETBEANS_HOME` 环境变量：

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

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

---

# 访问编译器和工具文档

您可以在下列位置访问文档：

- 可以通过随软件安装到本地系统或网络上的文档索引 (`file:/opt/SUNWspro/docs/index.html`) 来访问文档。  
如果软件不是安装在 `/opt` 目录中，请向系统管理员询问系统上的等效路径。
- 可以从 `docs.sun.comsm` web 站点中访问大多数手册。以下手册只能通过安装的软件来访问：
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User' Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User' Guide*
- 可以从 `docs.sun.com` Web 站点访问发行说明。
- 通过帮助菜单以及通过 IDE 中一些窗口和对话框的帮助按钮都可以访问此 IDE 所有组件的联机帮助。

在 `docs.sun.com` Web 站点 (`http://docs.sun.com`) 上，可通过 Internet 阅读、打印和购买 Sun Microsystems 手册。如果找不到手册，请参见随软件安装到本地系统或网络上的文档索引。

---

**注意** – Sun 对本文中提到的第三方 Web 站点的可用性概不负责；对于此类站点或资源上（或通过它们获得）的任何内容、广告、产品或其它材料，Sun 并不予以核准，亦不承担任何责任。对由于使用或依靠此类站点或资源上（或由此获得）的任何内容、商品或服务而导致（或用户宣称如此）的任何损害或损失，Sun 概不负责。

---

## 具有易读格式的文档

我们提供了易读格式的文档，以保证残障用户能够使用辅助技术加以阅读。您可以按下表所述的方式找到文档的易读格式版本。如果软件不是安装在 `/opt` 目录中，请向系统管理员询问系统上的等效路径。

| 文档类型  | 易读版本的格式和位置   |
|---|--|
| 手册（第三方手册除外）   | HTML：位于 <a href="http://docs.sun.com">http://docs.sun.com</a>  |
| 第三方手册：<br><i>tStandard C++ Library Class Reference</i><br><i>tStandard C++ Library User' Guide</i><br><i>tTools.h++ Class Library Reference</i><br>• <i>Tools.h++ User' Guide</i> | HTML：在安装的软件中，通过文档索引 ( <a href="file:/opt/SUNWspro/docs/index.html">file:/opt/SUNWspro/docs/index.html</a> ) 进行访问 |
| 自述文件和手册页  | HTML：在安装的软件中，通过文档索引 ( <a href="file:/opt/SUNWspro/docs/index.html">file:/opt/SUNWspro/docs/index.html</a> ) 进行访问 |
| 发行说明  | HTML：位于 <a href="http://docs.sun.com">http://docs.sun.com</a>  |

---

## 访问相关 Solaris 文档

下表描述了可通过 [docs.sun.com](http://docs.sun.com) 网站访问的相关文档。

| 文档集合             | 文档标题                                   | 描述  |
|------------------|--|---|
| Solaris 参考手册集合   | 请参见手册页章节的标题。                           | 提供有关 Solaris 操作环境的信息。                                       |
| Solaris 软件开发人员集合 | <i>Linker and Libraries Guide</i>      | 描述 Solaris 链接编译器和运行时链接程序的操作。                                |
| Solaris 软件开发人员集合 | <i>Multithreaded Programming Guide</i> | 涵盖 POSIX 和 Solaris 线程 API、使用同步对象进行编程、编译多线程程序以及查找用于多线程程序的工具。 |

---

## 开发人员资源

请访问 <http://developers.sun.com/prodtech/cc> 以查找下列经常更新的资源:

- 有关编程技巧和最佳做法的文章
- 有关简明编程技巧的知识库
- 编译器和工具组件的文档以及对随软件安装的文档的修正
- 有关支持级别的信息
- 用户论坛
- 可下载的代码示例
- 新技术介绍

您可以在 <http://developers.sun.com> 上找到其它的开发人员资源。

---

## 与 Sun 技术支持联系

如果本文没有回答您有关本产品的技术问题, 请访问:

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

---

## Sun 欢迎您提出意见和建议

Sun 愿意对其文档进行改进, 并欢迎您提出意见和建议。请通过以下地址将您的评论以电子邮件的形式发送给 Sun:

[docfeedback@sun.com](mailto:docfeedback@sun.com)

请在电子邮件主题行中加入您的文档的部件号码 (817-5810-10)。

# 第 1 章

---

## 简介

---

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

---

## 浮点环境

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

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

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

很多有关浮点算法的问题都涉及数的基本运算。例如：

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

另一类问题与异常和异常处理有关。例如，如果满足以下条件，会发生什么情况：

- 两个很大的数相乘？
- 被零除？

■ 试图计算负数的平方根？

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

IEEE 754 标准确保运算产生预期的数学结果，并且结果具有预期的特性。它还确保异常情况产生指定的结果，除非用户明确地指定其它选项。

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

# IEEE 算法

---

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

---

## IEEE 算法模型

本节介绍 IEEE 754 规范。

### 什么是 IEEE 算法？

IEEE 754 指定：

- 两种基本的浮点格式：*单精度和双精度*。  
IEEE 单精度格式具有 24 位有效数字精度，并总共占用 32 位。IEEE 双精度格式具有 53 位有效数字精度，并总共占用 64 位。
- 两种扩展浮点格式：*单精度扩展和双精度扩展*。  
此标准并未规定这些格式的严格精度和大小，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，并总共占用至少 79 位。
- 浮点运算的准确度要求：*加、减、乘、除、平方根、余数、将浮点格式的数舍入为整数值，在不同浮点格式之间转换，在浮点和整数格式之间转换以及比较时*。  
求余和比较运算必须精确无误。其它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舍入模式的规则对精确结果进行最低限度的修改，并将经过此类修改的结果提供给运算的目标。

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

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

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

五种类型的浮点异常是：*无效运算、被零除、上溢、下溢和不精确*。

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

IEEE 标准还建议为用户进行异常处理提供支持。

IEEE 标准所需的特性可以支持区间算法、异常的诊断回顾、标准基本函数（如  $\exp$  和  $\cos$ ）的有效实现、多精度算法以及用于数值计算的很多其它工具。

与任何其它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。

---

## IEEE 格式

本节介绍浮点数据在内存中的存储方式。其中汇总了各种 IEEE 存储格式的精度和范围。

### 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法解释。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。



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

表 2-1 IEEE 格式和语言类型

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

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

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

## 单精度格式

IEEE 单精度格式由三个字段组成：23 位尾数  $f$ ；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 位字元中（如图 2-1 所示）。第 0:22 位包含 23 位尾数  $f$ ，第 0 位是尾数的最低有效位，第 22 位是最高有效位；第 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；最高序位（第 31 位）包含符号位  $s$ 。



图 2-1 单精度存储格式

表 2-2 显示一侧的三个组成字段  $s$ 、 $e$  和  $f$  的值与另一侧的单精度格式位模式表示的值之间的对应关系； $u$  意味着无关，说明指示字段的值与特定单精度格式位模式值的确定无关。

表 2-2 IEEE 单精度格式位模式表示的值

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

注意，当  $e < 255$  时，为单精度格式位模式分配的值是使用以下方法构成的：将二进制基数点插入到紧邻尾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，从而以二进制位置表示法来表示一个带分数（整数加尾数，其中  $0 \leq \text{尾数} < 1$ ）。

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

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

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

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

表 2-3 单精度存储格式位模式及其 IEEE 值

| 通用名称 | 位模式（十六进制） | 十进制值 |
|------|-----------|------|
| +0   | 00000000  | 0.0  |
| -0   | 80000000  | -0.0 |
| 1    | 3f800000  | 1.0  |

表 2-3 单精度存储格式位模式及其 IEEE 值 (续)

| 通用名称    | 位模式 (十六进制) | 十进制值            |
|---------|------------|-----------------|
| 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   | 正无穷大            |
| -∞      | ff800000   | 负无穷大            |
| 非数      | 7fc00000   | NaN             |

NaN (非数) 可以使用任何一个满足 NaN 定义的多个位模式来表示。表 2-3 中所示的 NaN 十六进制值只是很多可用于表示 NaN 的位模式之一。

## 双精度格式

IEEE 双精度格式包含三个字段: 52 位尾数  $f$ ; 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连续地址的 32 位字元中 (如图 2-2 所示)。

在 SPARC 体系结构中, 具有较高地址的 32 位字元包含尾数的 32 个最低有效位; 而在 x86 体系结构中, 具有较低地址的 32 位字元包含尾数的 32 个最低有效位。

如果我们将  $f[31:0]$  表示为尾数的 32 个最低有效位, 则第 0 位是整个尾数的最低有效位, 第 31 位是 32 个最低有效尾数位中的最高有效位。

在另一个 32 位字元中, 第 0:19 位包含尾数的 20 个最高有效位  $f[51:32]$ , 第 0 位为这 20 个最高有效尾数位中的最低有效位, 而第 19 位为整个尾数的最高有效位; 第 20:30 位包含 11 位偏置指数  $e$ , 第 20 位为偏置指数的最低有效位, 第 30 位为最高有效位, 而最高序位 (第 31 位) 包含符号位  $s$ 。

图 2-2 在对位进行编号时将两个连续的 32 位字元视为一个 64 位字元, 其中第 0:51 位存储 52 位尾数  $f$ ; 第 52:62 位存储 11 位偏置指数  $e$ ; 第 63 位存储符号位  $s$ 。

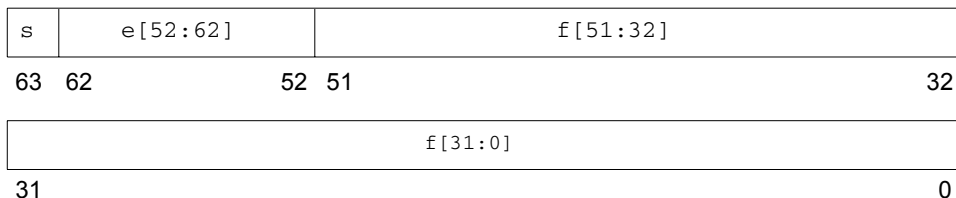


图 2-2 双精度存储格式

这三个字段中的位模式值决定了整个位模式所表示的值。

表 2-4 显示一侧的三个组成字段中的位值与另一侧的双精度格式位模式所表示的值之间的对应关系；*u* 意味着无关，说明指示字段的值与特定双精度格式位模式值的确定无关。

表 2-4 IEEE 双精度格式位模式表示的值

| 双精度格式位模式   | 值   |
|--|---|
| $0 < e < 2047$                                       | $(-1)^s \times 2^{e-1023} \times 1.f$ (正规数) |
| $e = 0; f \neq 0$<br>( <i>f</i> 中至少有一位不为零)           | $(-1)^s \times 2^{-1022} \times 0.f$ (次正规数) |
| $e = 0; f = 0$<br>( <i>f</i> 中的所有位均为零)               | $(-1)^s \times 0.0$ (有符号的零)                 |
| $s = 0; e = 2047; f = 0$<br>( <i>f</i> 中的所有位均为零)     | +INF (正无穷大)                                 |
| $s = 1; e = 2047; f = 0$<br>( <i>f</i> 中的所有位均为零)     | -INF (负无穷大)                                 |
| $s = u; e = 2047; f \neq 0$<br>( <i>f</i> 中至少有一位不为零) | NaN (非数)                                    |

注意，当  $e < 2047$  时，为双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尾数最高有效位的左侧，并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地指定其值，但偏置指数字段的值隐式指定了该值。

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

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

表 2-5 中给出了重要的双精度存储格式位模式的示例。第二列中的位模式显示为两个 8 位数字的十六进制数字。对于 SPARC 体系结构，左侧的数字是具有较低地址的 32 位字元的值，右侧的数字是具有较高地址的 32 位字元的值；而对于 x86 体系结构，左侧的数字是具有较高地址的字元，右侧的数字是具有较低地址的字元。最大正正规数是能以 IEEE 双精度格式表示的最大有限数。最小正次正规数是能以 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；对于所示的数字个数来说，它们是正确的。）

表 2-5 双精度存储格式位模式及其 IEEE 值

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

NaN（非数）可以使用满足 NaN 定义的多个位模式中的任何一个来表示。表 2-5 中所示的 NaN 十六进制值只是很多可用于表示 NaN 的位模式之一。

## 双精度扩展格式 (SPARC)

SPARC 浮点环境的四倍精度格式符合 IEEE 的双精度扩展格式定义。四倍精度格式占用四个 32 位字元并包含三个字段：112 位尾数  $f$ ；15 位偏置指数  $e$ ；以及 1 位符号  $s$ 。这些字段是连续存储的（如图 2-3 所示）。

具有最高地址的 32 位字元包含尾数的 32 个最低有效位（表示为  $f[31:0]$ ）。接下来的两个 32 位字元分别包含  $f[63:32]$  和  $f[95:64]$ 。下一个字元的第 0:15 位包含尾数的 16 个最高有效位  $f[111:96]$ ，第 0 位为这 16 位的最低有效位，而第 15 位为整个尾数的最高有效位。第 16:30 位包含 15 位偏置指数  $e$ ，第 16 位为偏置指数的最低有效位，第 30 位为最高有效位；第 31 位包含符号位  $s$ 。

图 2-3 在对位进行编号时将四个连续的 32 位字元视为一个 128 位字元，其中第 0:111 位存储尾数  $f$ ；第 112:126 位存储 15 位偏置指数  $e$ ；第 127 位存储符号位  $s$ 。

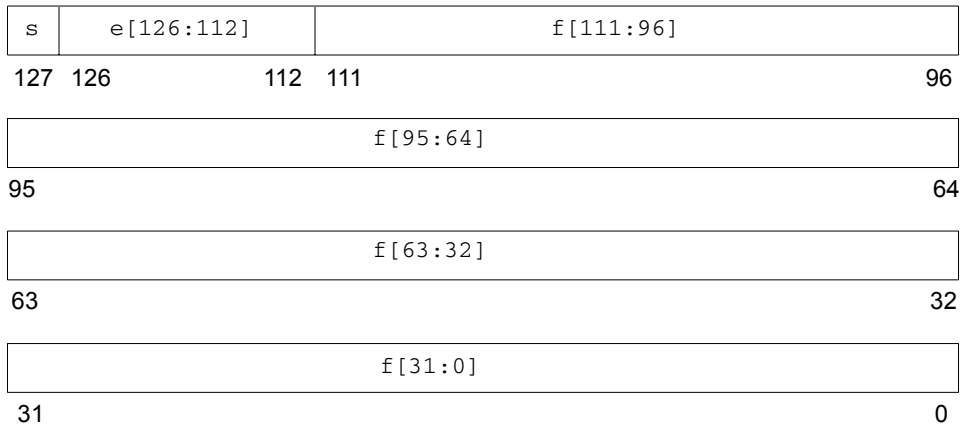


图 2-3 双精度扩展格式 (SPARC)

三个字段  $f$ 、 $e$  和  $s$  中的位模式值决定了整个位模式所表示的值。

表 2-6 显示三个组成字段中的值与四倍精度格式位模式所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 2-6 位模式表示的值 (SPARC)

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

表 2-7 中给出了重要的四倍精度双精度扩展存储格式位模式的示例。第二列中的位模式显示为四个 8 位数字的十六进制数字。最左侧的数字是具有最低地址的 32 位字元的值，最右侧的数字是具有最高地址的 32 位字元的值。最大正正规数是能以四倍精度格式表示的最大有限数。最小正次正规数是能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；对于所示的数字个数来说，它们是正确的。）

表 2-7 双精度扩展格式位模式 (SPARC)

| 通用名称    | 位模式 (SPARC) |          |          |          | 十进制值                                       |
|---------|-------------|----------|----------|----------|--|
| +0      | 00000000    | 00000000 | 00000000 | 00000000 | 0.0  |
| -0      | 80000000    | 00000000 | 00000000 | 00000000 | -0.0                                       |
| 1       | 3fff0000    | 00000000 | 00000000 | 00000000 | 1.0  |
| 2       | 40000000    | 00000000 | 00000000 | 00000000 | 2.0  |
| 最大正规数   | 7ffeffff    | ffffffff | ffffffff | ffffffff | 1.1897314953572317650857593266280070e+4932 |
| 最小正规数   | 00010000    | 00000000 | 00000000 | 00000000 | 3.3621031431120935062626778173217526e-4932 |
| 最大次正规数  | 0000ffff    | ffffffff | ffffffff | ffffffff | 3.3621031431120935062626778173217520e-4932 |
| 最小正次正规数 | 00000000    | 00000000 | 00000000 | 00000001 | 6.4751751194380251109244389582276466e-4966 |
| +∞      | 7fff0000    | 00000000 | 00000000 | 00000000 | +∞   |
| -∞      | ffff0000    | 00000000 | 00000000 | 00000000 | -∞   |
| 非数      | 7fff8000    | 00000000 | 00000000 | 00000000 | NaN  |

表 2-7 中所示的 NaN 十六进制值只是很多可用于表示 NaN 的位模式之一。

## 双精度扩展格式 (x86)

此浮点环境的双精度扩展格式符合 IEEE 的双精度扩展格式定义。它包含四个字段：63 位尾数  $f$ 、1 位显式前导有效数字位  $j$ 、15 位偏置指数  $e$  和 1 位符号  $s$ 。

在 x86 体系结构系列中，这些字段连续存储在 10 个地址连续的 8 位字节中。但是，“UNIX 系统 V 应用程序二进制接口 Intel 386 处理器增补 (Intel ABI)”要求双精度扩展参数和结果在堆栈中占用三个地址连续的 32 位字元，其中具有最高地址的字元的 16 个最高有效位没有使用（如图 2-4 所示）。

具有最低地址的 32 位字元包含尾数的 32 个最低有效位  $f[31:0]$ ，第 0 位为整个尾数的最低有效位，第 31 位为 32 个最低有效尾数位的最高有效位。在具有中间地址的 32 位字元中，第 0:30 位包含尾数的 31 个最高有效位  $f[62:32]$ ，第 0 位为这 31 个最高有效尾数位的最低有效位，第 30 位为整个尾数的最高有效位；此中间地址的 32 位字元的第 31 位包含显式前导有效数字位  $j$ 。

在具有最高地址的 32 位字元中，第 0:14 位包含 15 位偏置指数  $e$ ，第 0 位为偏置指数的最低有效位，第 14 位为最高有效位；第 15 位包含符号位  $s$ 。虽然 x86 体系结构系列不使用此最高地址的 32 位字元中的 16 个最高序位，但按照 Intel ABI 之规定它们必须存在（如上所述）。

图 2-4 在对位进行编号时将三个连续的 32 位字元视为一个 96 位字元，其中第 0:62 位存储 63 位尾数  $f$ ；第 63 位存储显式前导有效数字位  $j$ ；第 64:78 位存储 15 位偏置指数  $e$ ；第 79 位存储符号位  $s$ 。

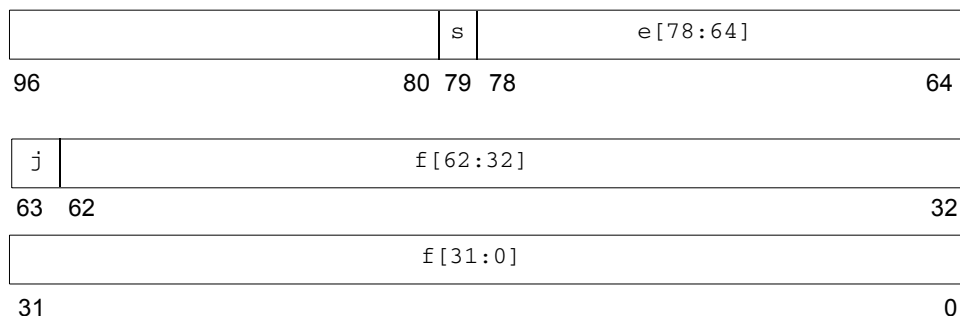


图 2-4 双精度扩展格式 (x86)

四个字段  $f$ 、 $j$ 、 $e$  和  $s$  中的位模式值决定了整个位模式所表示的值。

表 2-8 显示四个组成字段中的计数值与位模式所表示的值之间的对应关系。 $u$  意味着无关，说明指示字段的值与特定位模式值的确定无关。

表 2-8 位模式表示的值 (x86)

| 双精度扩展位模式 (x86)                               | 值   |
|--|---|
| $j = 0, 0 < e < 32767$                       | 不支持   |
| $j = 1, 0 < e < 32767$                       | $(-1)^s \times 2^{e-16383} \times 1.f$ (正规数)  |
| $j = 0, e = 0; f \neq 0$<br>( $f$ 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) |
| $j = 0, e = 0, f = 0$<br>( $f$ 中的所有位均为零)     | $(-1)^s \times 0.0$ (有符号的零)                   |



表 2-8 位模式表示的值 (x86) (续)

| 双精度扩展位模式 (x86)  | 值              |
|---|----------------|
| $j = 1; s = 0; e = 32767; f = 0$<br>( $f$ 中的所有位均为零)                       | +INF (正无穷大)    |
| $j = 1; s = 1; e = 32767; f = 0$<br>( $f$ 中的所有位均为零)                       | -INF (负无穷大)    |
| $j = 1; s = u; e = 32767; f = .1uuu - uu$                                 | QNaN (无噪声 NaN) |
| $j = 1; s = u; e = 32767; f = .0uuu - uu \neq 0$<br>( $f$ 中至少有一个 $u$ 不为零) | SNaN (信号 NaN)  |

注意，双精度扩展格式位模式没有隐含前导有效数字位。在双精度扩展格式中，前导有效数字位是作为单独的字段  $j$  显式指定的。但是，当  $e \neq 0$  时，不支持任何  $j = 0$  的位模式，这是因为在浮点运算中使用此类位模式作为操作数将导致无效运算异常。

在双精度扩展格式中，将分开的字段  $j$  和  $f$  合并后形成的字段称为有效数字。当  $e < 32767$  且  $j = 1$ ，或者当  $e = 0$  且  $j = 0$ ，可通过在前导有效数字位  $j$  和尾数的最高有效位之间插入二进制基数点来构成有效数字。

在 x86 双精度扩展格式中，前导有效数字位  $j$  为 0 且偏置指数字段  $e$  也为 0 的位模式表示次正规数，而前导有效数字位  $j$  为 1 且偏置指数字段  $e$  不为零的位模式表示正规数。因为前导有效数字位是显式表示的，而不是从指数值推出的，所以此格式也容许使用偏置指数为 0（如同次正规数一样），但前导有效数字位为 1 的位模式。每个此类位模式实际上与偏置指数字段为 1 的相应位模式（即正规数）表示相同的值，因此这些位模式称为伪次正规数。（在 IEEE 标准 754 中，次正规数称为非正规化数。）伪次正规数只是 x86 双精度扩展格式编码衍生出来的；当它们显示为操作数时，将它们隐式转换为相应的正规数，并且绝不会将它们生成为结果。

表 2-9 中给出了重要的双精度扩展存储格式位模式的示例。第二列中的位模式显示为一个 4 个数字的十六进制计数，它是具有最高地址的 32 位字元中的 16 个最低有效位（让我们回想一下，此最高地址 32 位字元中的 16 个最高有效位未使用，因此不显示它们的值），后跟两个 8 个数字的十六进制计数，左侧的数字是具有中间地址的 32 位字元的值，右侧的数字是具有最低地址的 32 位字元的值。最大正正规数是能以 x86 双精度扩展格式表示的最大有限数。最小正正规数是能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；对于所示的数字个数来说，它们是正确的。）

表 2-9 双精度扩展格式位模式及它们的值 (x86)

| 通用名称           | 位模式 (x86)              | 十进制值                         |
|----------------|------------------------|------------------------------|
| +0             | 0000 00000000 00000000 | 0.0                          |
| -0             | 8000 00000000 00000000 | -0.0                         |
| 1              | 3fff 80000000 00000000 | 1.0                          |
| 2              | 4000 80000000 00000000 | 2.0                          |
| 最大正规数          | 7ffe ffffffff ffffffff | 1.18973149535723176505e+4932 |
| 最小正正规数         | 0001 80000000 00000000 | 3.36210314311209350626e-4932 |
| 最大次正规数         | 0000 7fffffff ffffffff | 3.36210314311209350608e-4932 |
| 最小正次正规数        | 0000 00000000 00000001 | 3.64519953188247460253e-4951 |
| +∞             | 7fff 80000000 00000000 | +∞                           |
| -∞             | ffff 80000000 00000000 | -∞                           |
| 具有最大尾数的无噪声 NaN | 7fff ffffffff ffffffff | QNaN                         |
| 具有最小尾数的无噪声 NaN | 7fff c0000000 00000000 | QNaN                         |
| 具有最大尾数的信号 NaN  | 7fff bfffffff ffffffff | SNaN                         |
| 具有最小尾数的信号 NaN  | 7fff 80000000 00000001 | SNaN                         |

NaN（非数）可以使用满足 NaN 定义的多个位模式中的任何一个来表示。表 2-9 所示的 NaN 的十六进制值说明了尾数字段的前导（最高有效）位决定 NaN 是无噪声 NaN（前导尾数位 = 1）还是信号 NaN（前导尾数位 = 0）。

## 十进制表示中的范围和精度

本节介绍给定存储格式的数值范围和精度的概念。它包括与 IEEE 单精度格式和双精度格式对应的范围和精度，以及与 IEEE 双精度扩展格式在 SPARC 和 x86 体系结构上的实现对应的范围和精度。为了清晰起见，在定义范围和精度的概念时，我们指的是 IEEE 单精度格式。

IEEE 标准规定可以使用 32 位来表示单精度格式的浮点数。因为 32 个 0 和 1 只有有限多个组合，所以 32 位只能表示有限多个数。

一个很自然的问题是：

以此特定格式表示的最大和最小正数的十进制表示是什么？

将这一问题换一种提法并引入范围的概念：

在十进制表示法中，IEEE 单精度格式表示的数的范围是什么？

考虑到 IEEE 单精度格式的精度定义，用户可以证实 IEEE 单精度格式表示的浮点数的范围（如果限定为正正规化数）如下：

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

第二个问题指的是以给定格式表示的数的精度（不能将其准确度或有效数字的个数混淆）。可通过查看某些图片和示例来了解这些概念。

IEEE 的二进制浮点算法标准指定了一组以单精度格式表示的数值。请记住，这组数值被描述为一组二进制浮点数。IEEE 单精度格式的有效数字具有 23 位，这些位加上隐含前导位可提供 24 个数字（位）的（二进制）精度。

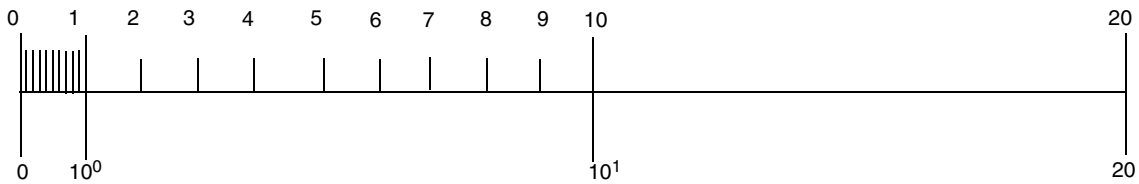
用户可通过在数字行中标记以下数字来获取一组不同的数值：

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

（可由有效数字中的  $q$  个十进制数字表示）。

图 2-5 举例说明了这种情况：

十进制表示:



二进制表示:

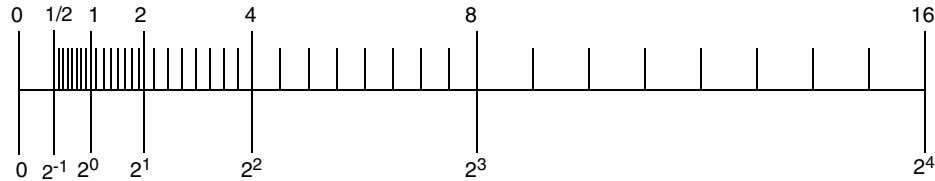


图 2-5 比较由十进制和二进制表示所定义的一组数

注意，这两组数字是不同的。因此，要估算 24 个有效二进制数字对应的有效十进制数字的个数，必须重新阐述该问题。

按以下方式重新阐述该问题：在二进制表示（计算机使用的内部格式）和十进制格式（用户通常感兴趣的格式）之间转换浮点数。事实上，您可能希望将十进制转换为二进制，再将二进制转换回十进制；以及将二进制转换为十进制，再将十进制转换回二进制。

由于两组数字是不同的，所以转换通常是不准确的，注意到这一点是非常重要的。如果执行正确的操作，则在将一组中的某个数转换到另一组中的一个数时，将会从第二组中的两个相邻数中选择一个数（具体选择哪个数是一个与舍入有关的问题）。

请参考一些示例。假定用户要用 IEEE 单精度格式来表示一个数，该数的十进制表示如下：

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

由于 IEEE 单精度格式只能精确地表示有限多个实数，并且上述格式的数字并非全部在此范围内，所以它通常无法精确地表示此类数字。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)
```

此程序的输出应类似于以下内容：

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

赋值给  $y$  的值  $8.388612 \times 10^5$  和打印输出的值之间的差值为 0.000000125，它比  $y$  小 7 个十进制数量级。以 IEEE 单精度格式表示  $y$  的准确度约为 6 到 7 个有效数字，也就是说如果以 IEEE 单精度格式表示  $y$ ，则它具有约 6 个有效数字。

类似地，赋值给  $z$  的值 1.3 和打印输出的值之间的差值为 0.00000004768，它比  $z$  小 8 个十进制数量级。以 IEEE 单精度格式表示  $z$  的准确度约为 7 到 8 个有效数字，也就是说如果以 IEEE 单精度格式表示  $z$ ，则它具有约 7 个有效数字。

现在表述这个问题：

假定将十进制浮点数  $a$  转换为其 IEEE 单精度格式二进制表示  $b$ ，然后再将  $b$  转换回十进制数  $c$ ； $a$  和  $a - c$  之间相差多少个数量级？

将该问题换一种提法：

以 IEEE 单精度格式表示的  $a$  具有多少个有效十进制数字；或者当用户以 IEEE 单精度格式表示  $x$  时，可以确信有多少十进制数字是准确的？

有效十进制数字的个数始终介于 6 和 9 之间，即至少有 6 个数字是准确的，但准确数字不会超过 9 个（如果转换是准确的，则准确数字个数不受此限制，此时可能有无限多个准确数字）。

相反，如果将 IEEE 单精度格式的二进制数转换为十进制数，然后再将其转换回二进制，则通常需要使用至少 9 个十进制数字，以确保经过两次转换后仍可以得到原始的数字。

表 2-10 中给出了完整的描述：

表 2-10 存储格式的范围和精度

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

## Solaris 环境中的基数转换

I/O 例程可使用基数转换对数字进行转换，例如 C 中的 `printf` 和 `scanf` 以及 Fortran 中的 `read`、`write` 和 `print`。对于这些函数，您需要在以 2 为基数和以 10 为基数的数字表示之间进行转换：

- 当读取以传统十进制表示法表示的数字并以内部二进制格式存储它时，就会发生从以 10 为基数到以 2 为基数的基数转换。
- 在将内部二进制值打印为十进制数字的 ASCII 字符串时，就会发生以 2 为基数到以 10 为基数的基数转换。

在 Solaris 环境中，用于所有语言基数转换的基本例程包含在标准 C 库 `libc` 中。这些例程使用表驱动算法，可在任何输入和输出格式之间进行正确舍入的转换。除了其准确度外，表驱动算法可减少在进行正确舍入的基数转换时出现最坏情况的次数。

IEEE 标准要求对  $10^{-44}$  和  $10^{+44}$  数量级范围内的典型数字进行正确的舍入，但允许对更大的指数进行略有误差的舍入。（请参见 IEEE 754 标准第 5.6 款。）`libc` 表驱动算法可在整个单精度格式、双精度格式和双精度扩展格式的范围内进行正确的舍入。

在 C 中，十进制数串和二进制浮点值之间的转换始终按照 IEEE 754 进行正确的舍入：转换后的结果是在当前舍入模式指定的方向上最接近原始值的数字（以结果格式表示）。如果舍入模式为向最接近的值舍入，并且原始值恰好位于两个使用结果格式表示的数字中间，则转换后的结果为最低有效位为偶数的数字。这些规则适用于编译器执行的源代码中的常量转换以及程序使用标准库例程执行的数据转换。

在 Fortran 中，缺省情况下，十进制字符串和二进制浮点值之间的转换按照与 C 相同的规则进行恰当的舍入。对于 I/O 转换，可以在程序中使用 `ROUNDING=` 说明符或者使用 `-iorounding` 标记进行编译以覆盖最近舍入模式中的“偶值优先”规则。详细信息，请参见《Fortran 用户指南》和 f95(1) 手册页。

有关基数转换的参考资料，请参见附录 F。特别指出，Coonen 的论文和 Sterbenz 的书就是较好的参考资料。

# 下溢

大致来说，如果算术运算的结果很小，以致于用其指定目标格式来存储这个结果时，无法避免产生大于常规误差的舍入误差，这样，就会发生下溢。

## 下溢阈值

表 2-11 显示单精度、双精度精度和双精度扩展精度的下溢阈值。

表 2-11 下溢阈值

| 目标精度          | 下溢阈值   |  |
|---------------|--------|--|
| 单精度           | 最小正规数  | 1.17549435e-38                             |
|               | 最大次正规数 | 1.17549421e-38                             |
| 双精度           | 最小正规数  | 2.2250738585072014e-308                    |
|               | 最大次正规数 | 2.2250738585072009e-308                    |
| 双精度扩展 (SPARC) | 最小正规数  | 3.3621031431120935062626778173217526e-4932 |
|               | 最大次正规数 | 3.3621031431120935062626778173217520e-4932 |
| 双精度扩展 (x86)   | 最小正规数  | 3.36210314311209350626e-4932               |
|               | 最大次正规数 | 3.36210314311209350590e-4932               |

正次正规数是介于最小正规数和零之间的数。将最小正规数附近的两个（正）极小的数相减可能生成次正规数。或者，将最小正正规数除以 2 可生成次正规结果。

虽然次正规数本身比正规数的精度位数少，但次正规数的存在给涉及很小的数的浮点计算提供了更大的精度。在正确的算术结果的数量级比最小的正正规数小时生成的次正规数（而不是返回答案 0）称为渐进下溢。

可以使用很多其它方法来处理此类下溢结果。过去常用的一种方法是将这些结果刷新为零。这种方法称为 *Store 0*；在出现 IEEE 标准之前，它是大多数主机上使用的缺省方法。

起草 IEEE 754 标准的数学家和计算机设计师考虑了一些替代方法，同时综合考虑可靠的算术解决方案和制订可有效实现的标准需要。

## IEEE 算法如何处理下溢？

IEEE 754 标准选择渐进下溢作为处理下溢结果的首选方法。这种方法相当于为存储的值定义两个表示：正规数和次正规数。

让我们回想一下，正规浮点数的 IEEE 格式为：

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

其中， $s$  是符号位， $e$  是偏置指数， $f$  是尾数。要完全指定该数字，只需要存储  $s$ 、 $e$  和  $f$ 。因为将正规数的有效数字隐含前导位定义为 1，所以不需要存储它。

可以存储的最小正正规数具有最大数量级的负指数和全部为零的尾数。如果将前导位考虑为 0（而不是 1），还可以存储甚至更小的数。在双精度格式中，这可有效地将最小指数从  $10^{-308}$  扩展到  $10^{-324}$ ，因为尾数部分为 52 位长（约为 16 个十进制数字）。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）即是渐进下溢。

无疑，次正规数越小，其尾数中的非零位就会越少；生成次正规结果的计算与使用正规操作数的计算的相对舍入误差的范围是不同的。但是，有关渐进下溢的重要事实是使用它时意味着：

- 使用下溢结果的造成的准确度降低不会超过普通舍入误差所造成的准确度降低。
- 当结果很小时，加、减、比较和求余始终精确运算。

让我们回想一下，次正规浮点数的 IEEE 格式为：

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

其中， $s$  是符号位，偏置指数  $e$  为零， $f$  是尾数。注意，2 的隐含幂指数偏差比正规格式中的偏差大 1，并且尾数的隐含前导位为零。

渐进下溢允许您扩展可表示数的下限。值不准确并不是由于小造成的，而是由于其相关误差造成的。利用次正规数的算法的误差范围比其它系统小。下一节将介绍在数学上使用渐进下溢的原因。

## 为什么使用渐进下溢？

与某些其它算法模型一样，次正规数的目的并不是完全避免下溢 / 上溢。相反，通过使用次正规数，用户在进行各种计算（通常，乘法后跟加法）时不再担心下溢问题。有关更详细的讨论，请参见 James Demmel 编写的 *Underflow and the Reliability of Numerical Software* 以及 S. Linnainmaa 编写的 *Combating the Effects of Underflow and Overflow in Determining Real Roots of Polynomials*。

在算法中使用次正规数意味着，在相加或相减时不会发生未捕获的下溢（这意味着准确度下降）。如果  $x$  和  $y$  在因子 2 的范围之内，则  $x - y$  没有误差。对于要在算法中的关键地方有效提高工作精度的一些算法来说，这是至关重要的。



此外，渐进下溢意味着下溢产生的误差不会比普通舍入误差大。对于任何其它下溢处理方法来说，我们就不能下这么肯定的结论，但这一事实就是使用渐进下溢最好的理由之一。

## 渐进下溢的误差特性

通常，要对浮点结果进行舍入处理：

$$\text{计算结果} = \text{真实结果} + \text{舍入值}$$

舍入值可以有多大？一种测量其大小的简便方法称为*最后一位中的单位*（简称为 *ulp*）。在标准表示中，浮点数尾数的最低有效位就是其*最后一位*。该位表示的值（例如，其表示除此位外均相同的两个数的绝对差值）就是该数的一个*最后一位中的单位*。如果计算结果是通过将真实结果舍入到最接近的可表示数得出的，则无疑舍入误差不会大于计算结果最后一位中的单位的一半。换句话说，在舍入模式为舍入到最接近值的 IEEE 算法中，计算结果的舍入值为：

$$0 \leq |\text{舍入值}| \leq 1/2 \text{ ulp}$$

注意，*ulp* 是一个相对数量。很大的数的 *ulp* 本身很大；而很小的数的 *ulp* 本身很小。可通过将 *ulp* 表示为一个函数来清楚地表述这种关系：*ulp(x)* 表示浮点数 *x* 最后一位中的单位。

再者，浮点数的 *ulp* 取决于表示该数的精度。例如，表 2-12 显示上述 4 种浮点格式的 *ulp(1)* 值。

表 2-12 4 种不同精度的 *ulp(1)*

| 精度           | 值   |
|--------------|---|
| 单精度          | $\text{ulp}(1) = 2^{-23} \sim 1.192093\text{e-}07$  |
| 双精度          | $\text{ulp}(1) = 2^{-52} \sim 2.220446\text{e-}16$  |
| 双精度扩展 (x86)  | $\text{ulp}(1) = 2^{-63} \sim 1.084202\text{e-}19$  |
| 四倍精度 (SPARC) | $\text{ulp}(1) = 2^{-112} \sim 1.925930\text{e-}34$ |

让我们回想一下，任何计算机算法只能准确地表示一组有限的数。随着数字的数量级变小并趋近于零，相邻可表示数之间的差值就会变小。相反，随着数字的数量级变大，相邻可表示数之间的差值就会加大。

例如，设想一下使用只有 3 位精度的二进制算法的情况。在任何两个 2 的幂指数之间，共有  $2^3 = 8$  个可表示数字（如图 2-6 所示）。

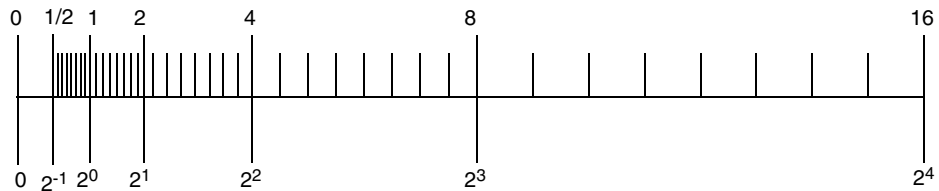


图 2-6 实数值线

实数值线显示数之间的差值随指数的增加而成倍增大。

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

在表 2-13 中， $\text{nextafter}(x, +\infty)$  表示在沿实数值线向  $+\infty$  方向移动时  $x$  后面的下一个可表示数。

表 2-13 可表示的单精度格式浮点数之间的差值

| $x$            | $\text{nextafter}(x, +\infty)$ | 差值             |
|----------------|--------------------------------|----------------|
| 0.0            | 1.4012985e -45                 | 1.4012985e -45 |
| 1.1754944e -38 | 1.1754945e -38                 | 1.4012985e -45 |
| 1.0            | 1.0000001                      | 1.1920929e -07 |
| 2.0            | 2.0000002                      | 2.3841858e -07 |
| 16.000000      | 16.000002                      | 1.9073486e -06 |
| 128.00000      | 128.00002                      | 1.5258789e -05 |
| 1.0000000e+20  | 1.0000001e+20                  | 8.7960930e+12  |
| 9.9999997e+37  | 1.0000001e+38                  | 1.0141205e+31  |

任何一组传统的可表示浮点数具有以下特性：一个不准确的结果的最坏情况是产生一个误差，该误差不会大于到计算结果的一个可表示相邻数之间的距离。如果将次正规数增加到一组可表示的数中，并且实现了渐进下溢，则一个不准确或下溢结果的最坏情况是产生一个误差，该误差不会大于到计算结果的一个可表示相邻数之间的距离。

特别地，在零和最小正规数之间的区域中，任何两个相邻数之间的距离等于零和最小正规数之间的距离。由于存在次正规数，可避免产生大于到最相邻可表示数之间距离的舍入误差。

因为任何计算产生的舍入误差均不大于到计算结果的任何可表示相邻数之间的距离，所以保留了可靠算术环境的很多重要特性，包括以下三个：

- $x \neq y \Leftrightarrow x - y \neq 0$
- $(x - y) + y \approx x$ ，在  $x$  和  $y$  两者较大的一个值的舍入误差范围内

- $1/(1/x) \approx x$ ，当  $x$  为正规化数时，意味着  $1/x \neq 0$

替代的下溢方案为 Store 0，它将下溢结果刷新为零。当  $x - y$  出现下溢时，Store 0 与第一个和第二个特性冲突。此外，当  $1/x$  出现下溢时，Store 0 与第二个特性冲突。

假定  $\lambda$  表示最小正正规化数（它也称为下溢阈值）。则可以用  $\lambda$  为单位来比较渐进下溢和 Store 0 的误差特性。

$$\text{渐进下溢: } |\text{误差}| < \frac{1}{2} \text{ulp}(\lambda)$$

$$\text{Store 0: } |\text{误差}| \approx \lambda$$

$\lambda$  最后一位中的单位的一半和  $\lambda$  本身差别很大。

## 两个渐进下溢与 Store 0 的示例

以下是两个大家熟知的算术示例。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;
```

对于渐进下溢，结果达到舍入允许的准确度。在 Store 0 中，有可能会得到一个很小但不为 0 的总和，该结果似乎合理，但几乎每个数字都是错误的。但为了公平起见，必须承认要只为了避免此类问题，聪明的程序员可以改变其计算的数值范围（如果他们能够预见很小的数可能降低准确度的话）。

第二个示例与改变数值范围无关，它产生一个复数的商。

$$\begin{aligned} a + i \cdot b &= \frac{p + i \cdot q}{r + i \cdot s}, \text{ 假定 } |r/s| \leq 1 \\ &= \frac{(p \cdot (r/s) + q) + i(q \cdot (r/s) - p)}{s + r \cdot (r/s)} \end{aligned}$$

可以看出尽管进行了舍入，计算的复数结果与准确结果的差值不会超过由于以下情况而产生的误差： $p + i \cdot q$  和  $r + i \cdot s$  两者的误差均不超过几个  $\text{ulp}$ 。此误差分析对于下溢方法也成立，但除以下情况：当  $a$  和  $b$  均出现下溢时，误差范围由  $|a + i \cdot b|$  的几个  $\text{ulp}$  界定。在将下溢刷新为零时，这两个结论均不正确。

在存在渐进下溢时，此复数商的计算算法是可靠的，并且对误差分析很有帮助。在使用 Store 0 时，没有其用于计算复数商的同样可靠、便于分析和有效的算法。在 Store 0 中，浮点环境的实现程序不再考虑底层、复杂的细节问题，而是由其用户考虑这些问题。

在渐进下溢时获得成功，而在使用 Store 0 时失败，热衷于使用 Store 0 的用户可能对此类问题认识不足。很多常用的数值技巧可归到此类别中：

- 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

## 下溢方法是否重要？

虽然以上示例进行了说明，但人们仍然对以下问题争论不休：既然下溢方法很少起作用，那么为什么还要认真考虑呢？但是，这种争论由其本身决定。

在缺少渐进下溢时，用户程序需要注意隐含的不准确阈值。例如，在单精度中，如果某些计算部分发生了下溢，并且使用 Store 0 将下溢结果替换为 0，则只能保证约  $10^{-31}$  的准确度，而不能保证  $10^{-38}$  的准确度（单精度指数常用的下限）。

这意味着程序员需要实现其自己的方法来检测什么时候他们会接近不准确度阈值，否则就无法可靠而稳定地实现其算法。

可以改变某些算法的数值范围，以避免在零附近的受限区域内进行计算。但是，对于每个数值程序而言，改变算法的数值范围和检测不准确阈值可能是很困难的，并且要花费很长的时间。

# 数学库

---

本章介绍 Solaris 操作环境和 Sun ONE Studio 编译器集合提供的数学库。本章除了列出每个库及其内容外，还讨论了编译器集合提供的数学库所支持的一些功能，其中包括 IEEE 支持函数、随机数生成器以及在 IEEE 和非 IEEE 格式之间进行数据转换的函数。

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

---

## 标准数学库

libm 数学库包含 Solaris 操作环境遵循的各种标准所需的函数。可以使用两种方式将此库与 Solaris 操作环境捆绑在一起：libm.a（静态版本）和 libm.so（共享版本）。

libm 标准安装的缺省目录为：

```
/usr/lib/libm.a  
/usr/lib/libm.so
```

libm 头文件标准安装的缺省目录为：

```
/usr/include/floatingpoint.h  
/usr/include/math.h  
/usr/include/sys/ieee.h
```

表 3-1 列出了 libm 中的函数。

表 3-1 libm 的内容

| 类型           | 函数名   |
|--------------|---|
| 代数函数         | cbrt、hypot、sqrt   |
| 基本超越函数       | asin、acos、atan、atan2、asinh、acosh、<br>atanh、<br>exp、expm1、pow、<br>log、log1p、log10、<br>sin、cos、tan、sinh、cosh、tanh |
| 高等超越函数       | j0、j1、jn、y0、y1、yn、<br>erf、erfc、gamma、lgamma、gamma_r、<br>lgamma_r  |
| 取整函数         | ceil、floor、rint   |
| IEEE 标准推荐的函数 | copysign、fmod、ilogb、nextafter、<br>remainder、scalbn、fabs   |
| IEEE 分类函数    | isnan   |
| 旧式浮点函数       | logb、scalb、significand  |
| 错误处理例程（用户定义） | matherr   |

注意，函数 `gamma_r` 和 `lgamma_r` 是 `gamma` 和 `lgamma` 的可重入版本。

有关动态和静态链接以及确定在运行程序时加载哪些共享对象的选项和环境变量的更多信息，请参见 `ld(1)` 和编译器手册页。

# 附加的数学库

## Sun 数学库

库 `libsunmath` 是随所有 Sun 语言产品提供的库的一部分。库 `libsunmath` 包含 Sun `libm` 以前版本中包含的一组函数。

`libsunmath` 标准安装的缺省目录为：

```
/opt/SUNWspr/prod/lib/libsunmath.a  
/opt/SUNWspr/lib/libsunmath.so
```

`libsunmath` 头文件标准安装的缺省目录为：

```
/opt/SUNWspr/prod/include/cc/sunmath.h  
/opt/SUNWspr/prod/include/floatingpoint.h
```

表 3-2 列出了 `libsunmath` 中的函数。对于每个数学函数，表中只给出函数从 C 程序中调用时所使用的双精度版本的名称。

表 3-2 `libsunmath` 的内容

| 类型                    | 函数名   |
|-----------------------|---|
| 表 3-1 中的函数            | 可以使用单精度、扩展精度 / 四倍精度，但 <code>matherr</code> 除外   |
| 基本超越函数                | <code>exp2</code> 、 <code>exp10</code> 、 <code>log2</code> 、 <code>sincos</code>  |
| 以度为单位的三角函数            | <code>asind</code> 、 <code>acosd</code> 、 <code>atand</code> 、 <code>atan2d</code> 、 <code>sind</code> 、 <code>cosd</code> 、 <code>sincosd</code> 、 <code>tand</code>         |
| 使用弧度表示的三角函数           | <code>asinpi</code> 、 <code>acospi</code> 、 <code>atanpi</code> 、 <code>atan2pi</code> 、 <code>sinpi</code> 、 <code>cospi</code> 、 <code>sincospi</code> 、 <code>tanpi</code> |
| 使用双精度 $\pi$ 参数缩小的三角函数 | <code>asinp</code> 、 <code>acosp</code> 、 <code>atanp</code> 、 <code>sinp</code> 、 <code>cosp</code> 、 <code>sincosp</code> 、 <code>tanp</code>                               |
| 财务函数                  | <code>annuity</code> 、 <code>compound</code>  |
| 取整函数                  | <code>aint</code> 、 <code>anint</code> 、 <code>rint</code> 、 <code>nint</code>  |
| IEEE 标准推荐的函数          | <code>signbit</code>  |
| IEEE 分类函数             | <code>fp_class</code> 、 <code>isinf</code> 、 <code>isnormal</code> 、 <code>issubnormal</code> 、 <code>iszero</code>   |

表 3-2 libsunmath 的内容 (续)

| 类型             | 函数名  |
|----------------|--|
| 提供有用 IEEE 值的函数 | min_subnormal、max_subnormal、<br>min_normal、max_normal、<br>infinity、signaling_nan、quiet_nan   |
| 加法的随机数生成器      | i_addran_、i_addrans_、i_init_addrans_、<br>i_get_addrans_、i_set_addrans_、r_addran_、<br>r_addrans_、r_init_addrans_、r_get_addrans_、<br>r_set_addrans_、d_addran_、d_addrans_、<br>d_init_addrans_、d_get_addrans_、<br>d_set_addrans_、u_addrans_                |
| 线性同余随机数生成器     | i_lcran_、i_lcrans_、i_init_lcrans_、<br>i_get_lcrans_、i_set_lcrans_、r_lcran_、<br>r_lcrans_、d_lcran_、d_lcrans_、u_lcrans_  |
| 与进位相乘的随机数生成器   | i_mwcran_、i_mwcrans_、i_init_mwcrans_、<br>i_get_mwcrans_、i_set_mwcrans_、i_lmcran_、<br>i_lmcrans_、i_llmcran_、i_llmcrans_、<br>u_mwcran_、u_mwcrans_、u_lmcran_、u_lmcrans_、<br>u_llmcran_、u_llmcrans_、r_mwcran_、<br>r_mwcrans_、d_mwcran_、d_mwcrans_、smwcran_ |
| 随机数混洗器         | i_shufrans_、r_shufrans_、d_shufrans_、<br>u_shufrans_  |
| 数据转换           | convert_external   |
| 控制舍入模式和浮点异常标记  | ieee_flags   |
| 浮点捕获处理         | ieee_handler、sigfpe  |
| 显示状态           | ieee_retrospective   |
| 启用/禁用非标准的算法    | standard_arithmetic、nonstandard_arithmetic   |

## 优化库

库 libmopt 中提供了 libm 中某些例程的优化版本。库 libcopt 中提供了 libc 中某些支持例程的优化版本。最后，在 SPARC® 系统上，libcx 中提供了某些 libc 支持例程的替代格式。

libmopt、libcopt 和 libcx 标准安装的缺省目录为：

```
/opt/SUNWspro/prod/lib/<arch>/libmopt.a
/opt/SUNWspro/prod/lib/<arch>/libcopt.a
/opt/SUNWspro/prod/lib/<arch>/libcx.a (仅 SPARC)
```



`/opt/SUNWspr/prod/lib/<arch>/libcx.so.1`（仅 SPARC）

此处，`<arch>` 表示一个特定于体系结构的库目录。在 SPARC 上，这些目录包括 `v7`、`v8`、`v8a`、`v8plus`、`v8plusa`、`v8plusb`、`v9`、`v9a` 和 `v9b`。在 x86 平台上，提供的唯一目录是 `f80387`。

有关 `-xarch` 的完整描述，请参见 Fortran、C 或 C++ 用户指南。

用户不能直接调用 `libcopt` 中包含的例程。它们替代了 `libc` 中编译器所使用的支持例程。

`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。请注意，与 `libm` 版本（可将其配置为提供任何 ANSI/POSIX、SVID、X/Open 或 IEEE 风格的异常情况处理）不同，`libmopt` 例程仅支持 IEEE 风格的异常情况处理。（请参见附录 E。）

要使用 `cc` 链接 `libmopt` 和 `libcopt`，请在命令行中指定 `-lmopt` 和 `-lcopt` 选项。（要获得最佳结果，请将 `-lmopt` 直接放在 `-lm` 的前面，并将 `-lcopt` 放在最后。）要使用任何其它编译器链接这两个库，请在命令行中的一个位置指定 `-xlibmopt` 标记。

**SPARC:** 库 `libcx` 包含更快版本的 128 位四倍精度浮点算法支持例程。用户不能直接调用这些例程；要由编译器调用它们。C++ 编译器自动链接 `libcx`，但 C 编译器并不自动链接 `libcx`。要将 `libcx` 用于 C 程序，请链接 `-lcx`。

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

`csh:`

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

`sh:`

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

## 矢量数学库（仅 SPARC）

在 SPARC 平台上，库 `libmvec` 提供计算整个参数矢量的常用数学函数的例程。

`libmvec` 标准安装的缺省目录为：

```
/opt/SUNWspr/prod/lib/<arch>/libmvec.a
/opt/SUNWspr/prod/lib/<arch>/libmvec_mt.a
```

此处，*<arch>* 表示一个特定于体系结构的库目录。在 SPARC 上，这些目录包括 v7、v8、v8a、v8plus、v8plusa、v8plusb、v9、v9a 和 v9b。在 x86 平台上，提供的唯一目录是 f80387。

表 3-3 列出了 libmvec 中的函数。

表 3-3 libmvec 的内容

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

注意，libmvec\_mt.a 提供了依赖于多处理器并行化的矢量函数并行版本。要使用 libmvec\_mt.a，您必须链接 -xparallel。

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

## libm9x 数学库

libm9x 数学库包含一些在 C99 中指定的数学和浮点相关函数。在编译器集合版本中，此库包含 *<fenv.h>* 浮点环境函数和增强函数以支持改进的浮点异常处理。

libm9x 标准安装的缺省目录为：

```
/opt/SUNWspr0/lib/libm9x.so
```

libm9x 头文件标准安装的缺省目录为：

```
/opt/SUNWspr0/prod/include/cc/fenv.h
```

```
/opt/SUNWspr0/prod/include/cc/fenv96.h
```

表 3-4 列出了 libm9x 中的函数。（仅在 x86 平台上提供精度控制函数 fegetprec 和 fesetprec。）

表 3-4 libm9x 的内容

| 类型           | 函数名   |
|--------------|---|
| C99 标准浮点环境函数 | feclearexcept、fegetenv、<br>fegetexceptflag、fegetround、<br>feholdexcept、feraiseexcept、fesetenv、<br>fesetexceptflag、fesetround、<br>fetestexcept、feupdateenv                             |
| 精度控制 (x86)   | fegetprec、fesetprec   |
| 异常处理和诊断回顾    | fex_get_handling、fex_get_log、<br>fex_get_log_depth、fex_getexcepthandler、<br>fex_log_entry、fex_merge_flags、<br>fex_set_handling、fex_set_log、<br>fex_set_log_depth、fex_setexcepthandler |

注意，libm9x 仅作为共享库提供。cc 编译器在链接时并不自动在共享库安装目录中搜索库。因此，要使用 cc 链接 libm9x，您必须启用静态链接程序和运行时链接程序以定位该库。可以使用以下三种方法之一启用静态链接程序以定位 libm9x：

- 在命令行中 -lm9x 的前面指定 -L/opt/SUNWspro/lib。
- 在命令行中给出全路径名 /opt/SUNWspro/lib/libm9x.so。
- 将 /opt/SUNWspro/lib 增加到由环境变量 LD\_LIBRARY\_PATH 指定的目录列表中。

可以使用以下三种方法之一启用运行时链接程序以定位 libm9x：

- 在链接时指定 -R/opt/SUNWspro/lib。
- 在链接时将 /opt/SUNWspro/lib 增加到由环境变量 LD\_RUN\_PATH 指定的目录列表中。
- 在运行时将 /opt/SUNWspro/lib 增加到由环境变量 LD\_LIBRARY\_PATH 指定的目录列表中。

**注意** - 如果将 /opt/SUNWspro/lib 增加到环境变量 LD\_LIBRARY\_PATH 中，可能导致与 Sun 性能库链接的程序使用并不最适于特定系统（在上面运行程序）的库版本。要在使用 cc 链接的程序中同时使用 libm9x 和 Sun 性能库，请不要将 /opt/SUNWspro/lib 增加到 LD\_LIBRARY\_PATH 中。在命令行中，只需在 -lm9x 前面指定 -xlic\_lib=sunperf。

所有其它 Sun ONE Studio 编译器集合中各编译器自动搜索共享库安装目录。要使用其中的任何编译器链接 libm9x，只需在命令行中指定 -lm9x 即可。（虽然 libm9x 主要用于 C/C++ 程序，但也可以将它用于 Fortran 程序；有关的示例，请参见附录 A。）

## 单、双和长双精度

大多数数值函数可以使用单、双、长双精度。表 3-5 中给出了从不同语言调用不同精度版本的示例。

表 3-5 调用单、双和四倍精度函数

| 语言      | 单精度   | 双精度  | 四倍精度   |
|---------|---|--|--|
| C、C++   | <pre>#include &lt;sunmath.h&gt; float x,y,z; x = sinf(y); x = fmodf(y,z); x = max_normalf(); x = r_addran_();</pre> | <pre>#include &lt;math.h&gt; double x,y,z; x = sin(y); x = fmod(y,z);  #include &lt;sunmath.h&gt; double x,y,z; x = max_normal(); x = d_addran_();</pre> | <pre>#include &lt;sunmath.h&gt; long double x,y,z; x = sinl(y); x = fmodl(y,z); x = max_normall();</pre> |
| Fortran | <pre>REAL x,y,z x = sin(y) x = r_fmmod(y,z) x = r_max_normal() x = r_addran()</pre>                                 | <pre>REAL*8 x,y,z x = sin(y) x = d_fmmod(y,z) x = d_max_normal() x = d_addran()</pre>  | <pre>REAL*16 x,y,z x = sin(y) x = q_fmmod(y,z) x = q_max_normal()</pre>                                  |

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；四倍精度函数的名称是通过增加 `l` 形成的。因为 Fortran 的调用惯例不同，所以 `libsunmath` 为单、双和四倍精度函数分别提供 `r_...`、`d_...` 和 `q_...` 版本。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。

---

**注意** - C 语言的 x86 版本支持长双精度。

---

# IEEE 支持函数

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

## `ieee_functions(3m)` 和 `ieee_sun(3m)`

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

表 3-6 `ieee_functions(3m)`

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

表 3-7 `ieee_sun(3m)`

| 函数                          | 说明   |
|-----------------------------|------|
| <code>sunmath.h</code>      | 头文件  |
| <code>fp_class(x)</code>    | 分类函数 |
| <code>isinf(x)</code>       | 分类函数 |
| <code>isnormal(x)</code>    | 分类函数 |
| <code>issubnormal(x)</code> | 分类函数 |
| <code>iszero(x)</code>      | 分类函数 |
| <code>signbit(x)</code>     | 分类函数 |

表 3-7 ieee\_sun(3m) (续)

| 函数                           | 说明   |
|------------------------------|------|
| nonstandard_arithmetic(void) | 切换硬件 |
| standard_arithmetic(void)    | 切换硬件 |
| ieee_retrospective(*f)       |      |

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

表 3-8 从 Fortran 中调用 ieee\_functions

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

表 3-9 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) |

**注意** - 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。

## ieee\_values(3m)

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

表 3-10 IEEE 值：单精度

| IEEE 值   | 十进制值<br>十六进制表示             | C、C++<br>Fortran  |
|----------|----------------------------|---|
| 最大正规数    | 3.40282347e+38<br>7f7fffff | <code>r = max_normalf();</code><br><code>r = r_max_normal();</code>         |
| 最小正规数    | 1.17549435e-38<br>00800000 | <code>r = min_normalf();</code><br><code>r = r_min_normal();</code>         |
| 最大次正规数   | 1.17549421e-38<br>007fffff | <code>r = max_subnormalf();</code><br><code>r = r_max_subnormal();</code>   |
| 最小次正规数   | 1.40129846e-45<br>00000001 | <code>r = min_subnormalf();</code><br><code>r = r_min_subnormal();</code>   |
| $\infty$ | Infinity<br>7f800000       | <code>r = infinityf();</code><br><code>r = r_infinity();</code>             |
| 无噪声 NaN  | NaN<br>7fffffff            | <code>r = quiet_nanf(0);</code><br><code>r = r_quiet_nan(0);</code>         |
| 信号 NaN   | NaN<br>7f800001            | <code>r = signaling_nanf(0);</code><br><code>r = r_signaling_nan(0);</code> |

表 3-11 IEEE 值：双精度

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

表 3-11 IEEE 值: 双精度 (续)

| IEEE 值   | 十进制值<br>十六进制表示                | C、C++<br>Fortran                                |
|----------|-------------------------------|---|
| $\infty$ | Infinity<br>7ff00000 00000000 | d = infinity();<br>d = d_infinity()             |
| 无噪声 NaN  | NaN<br>7fffffff ffffffff      | d = quiet_nan(0);<br>d = d_quiet_nan(0)         |
| 信号 NaN   | NaN<br>7ff00000 00000001      | d = signaling_nan(0);<br>d = d_signaling_nan(0) |

表 3-12 IEEE 值: 四倍精度 (SPARC)

| IEEE 值   | 十进制值<br>十六进制表示  | C、C++<br>Fortran                                 |
|----------|---|--|
| 最大正规数    | 1.1897314953572317650857593266280070e+4932<br>7ffeffff ffffffff ffffffff ffffffff | q = max_normal1();<br>q = q_max_normal()         |
| 最小正规数    | 3.3621031431120935062626778173217526e-4932<br>00010000 00000000 00000000 00000000 | q = min_normal1();<br>q = q_min_normal()         |
| 最大次正规数   | 3.3621031431120935062626778173217520e-4932<br>0000ffff ffffffff ffffffff ffffffff | q = max_subnormal1();<br>q = q_max_subnormal()   |
| 最小次正规数   | 6.4751751194380251109244389582276466e-4966<br>00000000 00000000 00000000 00000001 | q = min_subnormal1();<br>q = q_min_subnormal()   |
| $\infty$ | Infinity<br>7fff0000 00000000 00000000 00000000                                   | q = infinity1();<br>q = q_infinity()             |
| 无噪声 NaN  | NaN<br>7fff8000 00000000 00000000 00000000  | q = quiet_nan1(0);<br>q = q_quiet_nan(0)         |
| 信号 NaN   | NaN<br>7fff0000 00000000 00000000 00000001  | q = signaling_nan1(0);<br>q = q_signaling_nan(0) |



表 3-13 IEEE 值：双扩展精度 (x86)

| IEEE 值   | 十进制值<br>十六进制表示 (80 位)                                  | C、C++                  |
|----------|--|------------------------|
| 最大正规数    | 1.18973149535723176505e+4932<br>7ffe ffffffff ffffffff | x = max_normal();      |
| 正最小正规数   | 3.36210314311209350626e-4932<br>0001 80000000 00000000 | x = min_normal();      |
| 最大次正规数   | 3.36210314311209350608e-4932<br>0000 7fffffff ffffffff | x = max_subnormal();   |
| 正最小次正规数  | 1.82259976594123730126e-4951<br>0000 00000000 00000001 | x = min_subnormal();   |
| $\infty$ | Infinity<br>7fff 80000000 00000000                     | x = infinity();        |
| 无噪声 NaN  | NaN<br>7fff c0000000 00000000                          | x = q                  |
| 信号 NaN   | NaN<br>7fff 80000000 00000001                          | x = signaling_nanl(0); |

## ieee\_flags(3m)

ieee\_flags(3m) 是用于以下功能的 Sun 接口：

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

调用 ieee\_flags(3m) 的语法为：

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

可能作为参数值的 ASCII 字符串如表 3-14 所示：

表 3-14 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 |

表 3-14 ieee\_flags 的参数值 (续)

| 参数   | C 或 C++ 类型 | 所有可能的值   |
|------|------------|--|
| 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 异常标记的更多信息。

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

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

在使用 `ieee_flags` 检查、清除或设置舍入方向时，四个可能的输入参数值如表 3-15 所示。

表 3-15 ieee\_flags 舍入方向的输入值

| 参数     | 可能的值 ( <code>mode</code> 为 <code>direction</code> ) |
|--------|---|
| action | get、set、clear、clearall                              |
| in     | nearest、tozero、negative、positive                    |
| out    | nearest、tozero、negative、positive                    |

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

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

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

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

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

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

调用此函数的语法为：

```
C、C++      ieee_retrospective(fp);
Fortran     call ieee_retrospective()
```

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

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

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

只有在启用捕获或引发异常时，才会出现警告消息。

可以在 Fortran 程序中使用三种方法之一来禁止 `ieee_retrospective` 消息。一种方法是清除所有未处理的异常，禁用捕获，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()
```

---

**注意** – 建议不要不调查其原因而清除未处理的异常。

---

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

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

```
subroutine ieee_retrospective
return
end
```

## `nonstandard_arithmetic(3m)`

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

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

在使用特定硬件模式（刷新为零可加快速度）的 SPARC 实现中，函数 `nonstandard_arithmetic` 导致将下溢的结果刷新为零。加快速度的负面影响是准确度下降，这是因为没有了渐进下溢所提供的优势。

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 风格的算法没有影响，SuperSPARC® 就是一种此类实现。

---

## C99 浮点环境函数

本节介绍 C99 中的 `<fenv.h>` 浮点环境函数。在编译器集合版本中，这些函数是在 `libm9x.so` 库中提供的。它们提供很多与 `ieee_flags` 函数相同的功能，但它们使用更普通的 C 接口；因为它们是由 C99 定义的，所以它们在将来必然具有更大的可移植性。

---

**注意** - 为保持行为的一致，请不要在相同程序中同时使用 `libm9x.so` 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。

---

## 异常标记函数

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

C99 定义了以下异常标记函数：

表 3-16 C99 标准异常标记函数

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

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

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

将 `i` 设置为 `FE_DIVBYZERO`。

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

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

## 舍入控制

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

在 x86 平台上，`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 `libm9x.so` 提供两个函数来控制舍入精度模式：`fesetprec` 将当前的舍入精度设置为其参数指定的精度（必须是以上 3 个宏之一），而 `fegetprec` 返回对应于当前舍入精度的宏值。

## 环境函数

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

C99 定义 4 个函数以处理浮点环境。`libm9x.so` 提供的一个附加函数在多线程程序中非常有用。下表概要介绍了这些函数：

表 3-17 `libm9x.so` 浮点环境函数

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

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

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

```
#include <fenv.h>

void myfunc(...) {
    fenv_t env;

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

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



---

# libm 和 libsunmath 的实现功能

本节介绍 libm 和 libsunmath 的实现功能：

- 使用无限精确的  $\pi$  的参数变形以及用弧度表示的三角函数。
- 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。

## 关于算法

在 SPARC 系统上，libm 和 libsunmath 中的初等函数是使用不断变化的表驱动算法和多项式 / 有理数近似算法的组合实现的。在 x86 平台上，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；其它函数是用 SPARC 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 中的常见初等函数准确度和 libsunmath 中的常见单精度初等函数的表驱动算法和多项式 / 有理数近似算法提供的结果准确到最后一位中的一个单位 (*ulp*) 以内。在 SPARC 上，libsunmath 中的常见四倍精度初等函数提供的结果精确到一个 *ulp* 以内，`expm1l` 和 `log1pl` 函数除外，它们提供的结果精确到两个 *ulp* 以内。（常见函数包括指数、对数、幂函数和使用弧度参数的三角函数。其它函数的准确度较低，如双曲线三角函数和高等超越函数。这些误差范围是通过直接算法分析获得的。）用户也可以使用 BeEF（Berkeley 初等函数测试程序）测试这些例程的准确度，ucbtest 软件包中的 netlib 提供了这些测试程序 (<http://www.netlib.org/fp/ucbtest.tgz>)。

## 三角函数的参数缩小

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

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

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

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

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

`sinpi`、`cospi` 和 `tanpi` 函数组（请参见表 3-2）使用  $\pi$  来缩放输入参数，以避免由于范围缩小而产生的误差。

## 数据转换例程

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

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

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

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

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

每个随机数工具都包含每次生成一个随机数（即每个函数调用一个随机数）的例程以及在单个调用中生成随机数数组的例程。每次生成一个随机数的函数提供的数字位于表 3-18 所示的范围内。

表 3-18 单值随机数生成器的区间

| 函数                     | 下界          | 上界                    |
|------------------------|-------------|-----------------------|
| <code>i_addran_</code> | -2147483648 | 2147483647            |
| <code>r_addran_</code> | 0           | 0.9999999403953552246 |
| <code>d_addran_</code> | 0           | 0.999999999999998890  |

表 3-18 单值随机数生成器的区间 (续)

| 函数          | 下界                       | 上界                    |
|-------------|--------------------------|-----------------------|
| 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 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”讨论了线性同余算法的原理。Knuth 编写的 *The Art of Computer Programming* 第 2 卷中讨论了加法的随机数生成器。



# 异常和异常处理

---

本章描述 IEEE 浮点异常并展示如何检测、查找和处理它们。

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

... 为用户将由异常条件产生的复杂性降到最低。算术系统旨在尽可能长久地进行持续计算，并用合理的缺省响应处理异常情况（包括设置适当的标志）。

为了实现上述目标，在标准中规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捕获的方法（即，中断正常的控制流）。该程序可以选择性地提供一个陷阱处理程序，从而以一种适当的方式处理异常，例如为异常操作提供一个替换结果，然后恢复执行。本章列出由 IEEE 754 定义的异常及其缺省结果的列表，并描述可支持状态标志、捕获和异常处理的浮点环境的功能。

---

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

例如，当某个程序尝试求负数的平方根时，会产生异常。（这是无效运算异常的一个示例。）在出现这样的异常时，系统会以下列两种方法之一进行响应：

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

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

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

表 4-1 IEEE 浮点异常

| IEEE 异常 | 出现异常的原因   | 示例  | 在捕获被禁用时出现的缺省结果  |
|---------|---|---|---|
| 无效运算    | 对于将要执行的运算，某个操作数无效。<br><br>(在 x86 上，当浮点栈下溢或上溢时也会出现该异常，尽管这不符合 IEEE 标准。) | $0 \times \infty$<br>$0 / 0$<br>$\infty / \infty$<br>$x \text{ REM } 0$<br>负数平方根的操作数<br>带有信号传输的操作<br>NaN 操作数<br>无序比较<br>(请参见注释 1)<br>无效转换 (请参见注释 2)                 | 无噪声 NaN   |
| 被零除     | 针对有限操作数执行运算时生成精确的无穷大结果。   | $x / 0$ ，对于有限的非零 $x$<br>$\log(0)$   | 带有正确符号的无穷大  |
| 上溢      | 正确舍入后的结果在数量上大于目标格式所能表示的最大有限数（即，指数范围被超出）。                              | 双精度：<br>$\text{DBL\_MAX} + 1.0\text{e}294$<br>$\text{exp}(709.8)$<br><br>单精度：<br>(浮点) $\text{DBL\_MAX}$<br>$\text{FLT\_MAX} + 1.0\text{e}32$<br>$\text{expf}(88.8)$ | 依赖于舍入模式 (RM) 和中间结果的符号：<br>RM+ -<br>RN+ $\infty$ - $\infty$<br>RZ +max -max<br>R-+max - $\infty$<br>R+ + $\infty$ -max |

表 4-1 IEEE 浮点异常 (续)

| IEEE 异常 | 出现异常的原因                                    | 示例  | 在捕获被禁用时出现的缺省结果        |
|---------|--|---|-----------------------|
| 下溢      | 精确结果或正确舍入的结果在数量上小于目标格式能表示的最小正规数 (请参见注释 3)。 | 双精度:<br>nextafter(min_normal,-∞)<br>nextafter(min_subnormal,-∞)<br>DBL_MIN/3.0<br>exp(-708.5)<br><br>单精度:<br>(浮点) DBL_MIN<br>nextafterf(FLT_MIN, -∞)<br>expf(-87.4) | 次正规数或零                |
| 不精确     | 有效运算的舍入结果不同于无限精确结果。(大多数浮点运算都产生该异常。)        | 2.0 / 3.0<br>(浮点) 1.12345678<br>log(1.1)<br>DBL_MAX + DBL_MAX,<br>当没有上溢捕获时  | 该运算的结果 (经过舍入、上溢或下溢处理) |

## 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 (非数)。

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

表 4-2 无序比较

| 判定 | C、C++ | f95  | 无效异常<br>(如果无序的话) |
|----|-------|------|------------------|
| =  | ==    | .EQ. | 否                |
| ≠  | !=    | .NE. | 否                |
| >  | >     | .GT. | 是                |
| ≥  | >=    | .GE. | 是                |
| <  | <     | .LT. | 是                |
| ≤  | <=    | .LE. | 是                |

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上溢。
3. IEEE 单精度、双精度和扩展格式能表示的最小正规数分别为  $2^{-126}$ 、 $2^{-1022}$  和  $2^{-16382}$ 。有关 IEEE 浮点格式的说明，请参见第 2 章。

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

异常的优先顺序如下所示：无效（优先级最高）、上溢、相除、下溢、不精确（优先级最低）。在 x86 平台上，非正规操作数异常在所有异常中的优先级最低。

能够同时发生的标准异常组合只有不精确的上溢和不精确的下溢这两种。在 x86 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。如果启用了对上溢、下溢和不精确的捕获，则上溢和下溢捕获的优先级高于不精确捕获；在 x86 上，上溢、下溢和不精确捕获的优先级都高于非正规操作数捕获。

---

## 检测异常

正如 IEEE 标准所要求的那样，SPARC 和 x86 平台上的浮点环境提供用来记录所出现的浮点异常的状态标志。程序可通过测试这些标志来确定已发生了哪些异常。这些标志还可以被显式设置和清除。ieee\_flags 函数提供一种访问这些标志的方法。在用 C 或 C++ 编写的程序中，libm9x.so 中的 C99 浮点环境函数提供另一种方法。

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

在 x86 上，浮点状态字 (SW) 为应计异常以及浮点栈的状态提供标志。

### ieee\_flags(3m)

ieee\_flags(3m) 的调用语法为：

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



程序可提供字符串 “exception” 并将其作为第二个参数，从而可使用 `ieee_flags` 函数测试、设置或清除应计异常状态的标志。例如，要从 Fortran 中清除上溢异常标志，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);
```

当操作作为 “get” 时，`out` 中返回的字符串为：

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

例如，在 Fortran 调用中：

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

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

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

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

表 4-3 异常位

| 异常   | 位的位置                         | 应计异常位  |
|------|------------------------------|--|
| 无效   | <code>fp_invalid</code>      | <code>i &amp; (1 &lt;&lt; fp_invalid)</code>                           |
| 上溢   | <code>fp_overflow</code>     | <code>i &amp; (1 &lt;&lt; fp_overflow)</code>                          |
| 相除   | <code>fp_division</code>     | <code>i &amp; (1 &lt;&lt; fp_division)</code>                          |
| 下溢   | <code>fp_underflow</code>    | <code>i &amp; (1 &lt;&lt; fp_underflow)</code>                         |
| 不精确  | <code>fp_inexact</code>      | <code>i &amp; (1 &lt;&lt; fp_inexact)</code>                           |
| 非正规化 | <code>fp_denormalized</code> | <code>i &amp; (1 &lt;&lt; fp_denormalized)</code><br>(仅对于 <i>x86</i> ) |

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);

```

## C99 异常标志函数

C/C++ 程序可以使用 `libm9x.so` 中的 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` 定义五个宏（与五个标准异常相对应）：`FE_INEXACT`、`FE_UNDERFLOW`、`FE_OVERFLOW`、`FE_DIVBYZERO` 和 `FE_INVALID`。它还将宏 `FE_ALL_EXCEPT` 定义为五个异常宏的按位“或”。可将这些宏结合在一起，以便测试或清除异常标志的任何子集或者引发任何组合的异常。下面的示例显示如何将这些宏与几个 C99 浮点环境函数结合使用；要获得更多的信息，请参见 `feclearexcept(3M)` 手册页。（备注：为了使行为保持一致，请不要在同一个程序中同时使用 `libm9x.so` 中的 C99 浮点环境函数和扩展以及 `libsunmath` 中的 `ieee_flags` 和 `ieee_handler` 函数。

要清除所有这五个异常标志，请编写：

```
feclearexcept(FE_ALL_EXCEPT);
```

要测试引发的是无效运算标志还是被零除标志，请编写：

```
int i;

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

要模拟引发上溢异常（请注意，在启用了上溢捕获时，这将引发捕获）：

```
feraiseexcept(FE_OVERFLOW);
```

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

```
fexcept_t flags;

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

---

## 查找异常

通常，程序员在编写程序时不会考虑异常，因此在检测到异常时，通常会询问的第一个问题是：异常发生在何处？查找异常发生位置的一种方法就是在程序中的各个点测试异常标志，但是通过这种方法精确地查找异常会需要大量测试并带来大量开销。

用来确定异常发生位置的更方便的方法就是启用它的捕获。当发生启用了捕获的异常时，操作系统通过发送 SIGFPE 信号（请参见信号 (5) 手册页）来通知该程序。因此，通过启用对异常的捕获，可以通过以下方法来确定异常的发生位置：在调试器中运行程序，并在接收到 SIGFPE 信号时中止；或者建立 SIGFPE 处理程序以打印发生异常的指令的地址。请注意，必须针对异常启用捕获才能生成 SIGFPE 信号；如果在禁用捕获时发生异常，则会设置相应的标志，并使用表 4-1 中指定的缺省结果继续执行程序，但是不会发送任何信号。

## 使用调试器查找异常

本节举例说明如何使用 dbx（源代码级调试器）和 adb（汇编级调试器）来调查浮点异常的原因并查找引发它的指令。记住，为了使用 dbx 的源代码级调试功能，程序应当用 -g 标志进行编译。要获得更多的信息，请参考《使用 dbx 调试程序》手册。

考虑下面的 C 程序：

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

int main(void)
{
    double x, y;

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

编译和运行该程序会生成：

```
-4.2 NaN
```

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

## 使用 `dbx` 来查找导致异常的指令

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

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

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

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

输出结果表明，因试图求负数的平方根而结果在 `sqrtm1` 函数中出现异常。

## 使用 `adb` 查找导致异常的指令

尽管 `adb` 不能像 `dbx` 那样查找源文件和行号，但是也可以使用 `adb` 来确定发生异常的原因。同样，第一步也是用 `-ftrap` 重新编译：

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

现在调用 `adb` 并运行该程序。在出现无效运算异常时，`adb` 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 `adb` 已停止的指令前面查找最后一个浮点指令。在 SPARC 上，结果可能类似于下面的副本。

```
example% adb a.out
:r
SIGFPE:Arithmetic Exception (invalid floating point operation)
stopped at:
__sqrt+0x3c:be      __sqrt+0x98
__sqrt+30?4i
__sqrt+0x30:sethi   %hi(0x7ff00000), %o0
                    and     %i0, %o0, %o1
                    fsqrd  %f0, %f30
                    be     __sqrt+0x98
<f0=F
                    -4.2000000000000002e+00
```

输出结果表明该异常是由 `fsqrd` 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 x86 上，因为指令没有固定的长度，所以要从代码中查找对代码进行反汇编的正确地址，可能要经过反复试验。在本例中，异常在接近函数的开头处发生，因此我们可从那里反汇编。（请注意，这一输出假定已用 `-xlibmil` 标志编译了该程序。）下面可能是典型的结果。

```
example% adb a.out
:r
SIGFPE:Arithmetic Exception (invalid floating point operation)
stopped at:
sqrtm1+0x16:fstpl  -0x10(%ebp) [0xfffffffff0]
sqrtm1?12i
sqrtm1:
sqrtm1:pushl  %ebp
                    movl   %esp,%ebp
                    subl   $0x10,%esp [0x10,-]
                    movl   0xc(%ebp),%eax [0xc,-]
                    pushl  %eax
                    movl   0x8(%ebp),%eax [8,-]
                    pushl  %eax
                    fldl   (%esp,1)
                    fsqrt
                    addl   $0x8,%esp [8,-]
```

```

                                fstpl  -0x10(%ebp) [0xfffffffff0]
                                fldl   -0x10(%ebp) [0xfffffffff0]
$*
80387 chip is present.
cw      0x137e
sw      0x3800
cssel  0x17  ipoff 0x691          dataset 0x1f  dataoff 0x0

st[0]  -4.2000000000000001776356839          VALID
st[1]  +0.0          EMPTY
st[2]  +0.0          EMPTY
st[3]  +0.0          EMPTY
st[4]  +0.0          EMPTY
st[5]  +0.0          EMPTY
st[6]  +0.0          EMPTY
st[7]  +0.0          EMPTY

```

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

## 在不重新编译的情况下启用捕获

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

如果您使用的是 `dbx`，则可以通过直接修改浮点状态寄存器来手动启用捕获。在 `SPARC` 上这样做需要一点技巧，因为操作系统只有在程序内第一次使用浮点单元时才会启动它，并在这一点将浮点状态寄存器重置为禁用所有捕获。因此，只有在程序已经执行了至少一个浮点指令之后，才能手动启用捕获。在我们的示例中，当调用 `sqrtm1` 函数时，浮点单元已经被访问了，因此我们可以在该函数的入口处设置断点，启用对无效运算异常的捕获，命令 `dbx` 在接收到 `SIGFPE` 信号时中止，然后继续执行。步骤如下所示（请注意，使用 `assign` 命令可修改 `%fsr`，从而启用对无效运算异常的捕获）：



```

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

```

在 x86 上，在将控制转交给主程序之前，编译器自动链接到每个程序中的启动代码会初始化浮点单元。因此，在主程序开始之后，您可以随时手动启用捕获。下面的示例显示所产生的步骤：

```

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

```

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

如果构成该程序的目标文件和库存在，则可以通过将程序与一个适当的初始化例程重新链接来启用捕获。首先，创建一个类似如下的 C 源文件：

```
#include <ieeefp.h>

#pragma init (trapinvalid)

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

现在，编译该文件，以便创建一个目标文件并用该目标文件链接初始程序：

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

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

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

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

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

（要获得有关创建和预装共享对象的更多信息，请参见《链接程序和库指南》。）

原则上，您可以通过按上述方式预装共享对象来更改任何浮点控制模式的初始化方法。但是，请注意，在运行时链接程序将控制转交给属于主可执行文件的启动代码之前，无论共享对象中的初始化例程是预装的还是显式链接的，它们都是由运行时链接程序执行的。启动代码随后建立任何通过 `-ftrap`、`-fround`、`-fns (SPARC)` 或 `-fprecision (x86)` 编译器标志选择的非缺省模式，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），最后将控制传递到主程序。因此，在 SPARC 上，(i) 任何由共享对象中的初始化例程建立的浮点控制模式（如在上例中启用的捕获）将在该程序的整个执行过程中保持有效（除非它们被覆盖）；(ii) 任何通过编辑器标志选择的非缺省模式将覆盖由共享对象中的初始化例程建立的模式（但是，通过编译器标志选择的缺省模式将不覆盖以前建立的模式）；(iii) 由属于主可执行文件的初始化例程或主程序本身建立的任何模式都将覆盖前两者。

在 x86 上，情况变得复杂，这是由于在开始新进程时，系统内核用一些非缺省模式初始化浮点硬件，但是在将控制转交给主程序之前，由编译器自动提供的启动代码会将其中的一些模式重置到缺省模式。因此，除非共享对象中的初始化例程更改浮点控制模式，否则它们会在以下情况下运行：对无效运算、被零除和上溢异常的捕获处于启用状态，而且舍入精度设置为舍入 53 个有效位。在运行时链接程序将控制转交给启动代码之后，该代码调用 `__fpstart` 例程（位于 C 标准库 `libc` 中），从而禁用所有捕获并将舍入精度设置为 64 个有效位。然后，在执行任何静态链接的初始化例程并将控制转交给主程序之前，启动代码建立由 `-fround`、`-ftrap` 或 `-fprecision` 标志选择的任何非缺省模式。因此，在 x86 平台上，为了通过用初始化例程预装共享对象来启用捕获（或者更改舍入精度模式），必须覆盖 `__fpstart` 例程，以便它不重置捕获启用模式和舍入精度模式。`__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；但是，下面的代码显示一种执行该操作的方法。

```

#include <ieeefp.h>
#include <sunmath.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 long _fp_hw;

void __fpstart()
{
    char *out;

    /* perform the same floating point initializations as
       the standard __fpstart() function defined by the
       System V ABI Intel processor supplement _but_ leave
       all trapping modes as is */
    __flt_rounds = __fltrounds();
    sysi86(SI86FPHW, &_fp_hw);
    _fp_hw &= 0xff;
    ieee_flags("set", "precision", "extended", &out);
}

```

请注意，上面的 `__fpstart` 例程调用 `ieee_flags`（包含在 `libsunmath` 中）。因此，为了从该代码创建共享对象，必须使用 `-R` 和 `-L` 选项来指定共享对象 `libsunmath.so` 的位置。该库位于编译器产品安装区域中。假定您已经将编译器安装到缺省位置，则可以使用如下所示的步骤来编译上述代码，以便创建共享对象并预装该共享对象，从而启用捕获。

```

example% cc -Kpic -G -ztext init.c -o init.so -R/opt/SUNWspro/lib
-L/opt/SUNWspro/lib -lsunmath -lc
example% env LD_PRELOAD=./init.so a.out
Arithmetic Exception

```

## 使用信号处理程序来查找异常

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

使用 `ieee_handler` 可以同时启用五个 IEEE 浮点异常中的任何一个或几个的捕获，然后或者要求程序在发生指定异常时终止，或者建立 SIGFPE 处理程序。还可以使用较低级别的函数 `sigfpe(3)`、`signal(3c)` 或 `sigaction(2)` 之一来安装 SIGFPE 处理程序；但是，这些函数不像 `ieee_handler` 那样启用捕获。（切记，只有启用了浮点异常的捕获时，浮点异常才触发 SIGFPE 信号。）

### `ieee_handler(3m)`

`ieee_handler` 的调用语法是：

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

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

三个输入参数可使用下列值：

| 输入参数                   | C 或 C++ 类型                       | 可能的值  |
|------------------------|----------------------------------|---|
| <code>action</code>    | <code>char *</code>              | <code>get</code> 、 <code>set</code> 、 <code>clear</code>  |
| <code>exception</code> | <code>char *</code>              | <code>invalid</code> 、 <code>division</code> 、 <code>overflow</code> 、 <code>underflow</code> 、 <code>inexact</code> 、 <code>all</code> 、 <code>common</code> |
| <code>handler</code>   | <code>sigfpe_handler_type</code> | 用户定义的例程<br><code>SIGFPE_DEFAULT</code><br><code>SIGFPE_IGNORE</code><br><code>SIGFPE_ABORT</code>   |

当请求的操作是“set”时，`ieee_handler` 针对由 *exception* 命名的异常建立由 *handler* 指定的处理函数。处理函数可以是 `SIGFPE_DEFAULT` 或 `SIGFPE_IGNORE`，这两个函数都选择 IEEE 的缺省行为 `SIGFPE_ABORT`，它导致程序在任何指定异常出现时终止；也可以是用户提供的子例程的地址，它导致子例程在任何指定异常出现时被调用（对于用 `SA_SIGINFO` 标志集安装的信号处理程序，使用 `sigaction(2)` 手册页中描述的参数调用）。如果处理程序是 `SIGFPE_DEFAULT` 或 `SIGFPE_IGNORE`，则 `ieee_handler` 还禁用对指定异常的捕获；对于任何其他处理程序，`ieee_handler` 启用捕获。（在 x86 平台上，当某个异常处于启用状态，而且与它相对应的标志被引发时，会出现浮点硬件捕获。因此，为了避免多余的捕获，在调用 `ieee_handler` 以启用捕获之前，程序应当清除每个指定 *exception* 的标志。）

当请求的操作是“clear”时，`ieee_handler` 为指定异常取消当前安装的任何处理函数并禁用其捕获。（这与针对 `SIGFPE_DEFAULT` 执行“set”操作相同。）当操作为“clear”时，会忽略第三个参数。

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

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

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

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

下面是等效的 Fortran 代码：

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

下面的 C 代码段恢复对所有异常的 IEEE 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'
```

## 从信号处理程序报告异常

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

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

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

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

当该处理程序被调用时，`sig` 参数包含已发送的信号的编号。信号编号是在 `sys/signal.h` 中定义的；SIGFPE 的信号编号是 8。

`sip` 参数指向记录该信号附加信息的结构。对于 SIGFPE 信号，该结构的相关成员是 `sip->si_code` 和 `sip->si_addr`（请参见 `sys/siginfo.h`）。这些成员的意义依赖于系统以及触发 SIGFPE 信号的事件。

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

表 4-4 算术异常的类型

| SIGFPE 类型  | IEEE 类型 |
|------------|---------|
| FPE_INTDIV |         |
| FPE_INTOVF |         |
| FPE_FLTRES | 不精确     |
| FPE_FLTDIV | 相除      |

表 4-4 算术异常的类型 (续)

| SIGFPE 类型  | IEEE 类型 |
|------------|---------|
| FPE_FLTUND | 下溢      |
| FPE_FLTINV | 无效      |
| FPE_FLTOVF | 上溢      |

正如上表所示, 每种类型的 IEEE 浮点异常都有一个与之对应的 SIGFPE 信号类型。整数被零除 (FPE\_INTDIV) 和整数上溢 (FPE\_INTTOVF) 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上溢。特殊的指令可导致传递 FPE\_INTTOVF 类型的 SIGFPE 信号, 但是 Sun 编译器不生成这些指令。)

对于与 IEEE 浮点异常相对应的 SIGFPE 信号, 在 SPARC 系统上, `sig->si_code` 成员指出发生了哪种异常, 而在 x86 上, 它指出其标志 (不包括非正规操作数标志) 被引发的最高优先级的异常。在 SPARC 系统上, `sig->si_addr` 成员存放导致了异常的指令的地址, 而在 x86 平台上, 它存放用来进行捕获的指令的地址 (通常是紧跟在导致异常的指令后面的浮点指令)。

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

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

代码示例 4-1 SIGFPE 处理程序

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

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

#ifdef i386
    unsigned    sw;

    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status &
        ~uap->uc_mcontext.fpregs.fp_reg_set.fpship_state.state[0];
    if (sw & (1 << fp_invalid))
```



```
        code = FPE_FLTINV;
    else if (sw & (1 << fp_division))
        code = FPE_FLTDIV;
    else if (sw & (1 << fp_overflow))
        code = FPE_FLTOVF;
    else if (sw & (1 << fp_underflow))
        code = FPE_FLTUND;
    else if (sw & (1 << fp_inexact))
        code = FPE_FLTRES;
    else
        code = 0;

    addr = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.
        state[3];
#else
    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
#endif
    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 = 1.71116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
Note:IEEE floating-point exception flags raised:
    Inexact; Underflow;
IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M)
```

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

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.71116e-309
max_normal = 1.79769e+308
fp exception 4 at address 8048fe6
max_normal * max_normal = 1.79769e+308
Note:IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_handler(3M)
```

在大多数情况下，当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` 报告的值不是上溢运算（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“处理异常”（在第 97 页上）。

## 使用 libm9x.so 异常处理扩展查找异常

使用 libm9x.so 中 C99 浮点环境函数的异常处理扩展，C/C++ 程序可通过多种方法来查找异常。这些扩展包括可建立处理程序并同时启用捕获（正如 `ieee_handler` 所执行的操作那样）的函数，但是它们提供更大的灵活性。它们还支持将有关浮点异常的诊断消息记录到选定文件中。

## fex\_set\_handling(3m)

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

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

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

表 4-5 fex\_set\_handling 的异常代码

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

为方便起见，fenv.h 还定义下列值：FEX\_NONE（没有异常）、FEX\_INVALID（所有的无效运算异常）、FEX\_COMMON（上溢、被零除和所有的无效运算）和 FEX\_ALL（所有异常）。

mode 参数指定要为指出的异常建立的异常处理模式。共有五种可能的模式：

- FEX\_NONSTOP 模式提供 IEEE 754 缺省的不断行为。这等效于使异常捕获保持禁用。（请注意，与 ieee\_handler 不同的是，fex\_set\_handling 允许您为某些类型的无效运算异常建立非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捕获。在发生异常时，如果以前安装了 SIGFPE 处理程序，系统会将控制转交给该处理程序，否则会终止。
- FEX\_ABORT 模式导致程序在发生异常时调用 abort(3c)。
- FEX\_SIGNAL 安装由 handler 参数为指出的异常指定的处理函数。当发生其中的任一异常时，会用相同的参数调用该处理程序，就好像它是由 ieee\_handler 安装的一样。

- FEX\_CUSTOM 安装由 *handler* 为指出的异常指定的处理函数。与 FEX\_SIGNAL 模式不同的是，在发生异常时，会用简化的参数列表调用该处理程序。这些参数由一个整数（其值是列在表 4-5 中的某个值）和一个指针（它所指向的结构用来记录有关导致该异常的运算的附加信息）组成。该结构的内容在下一节和 `fex_set_handling(3m)` 手册页中介绍。

请注意，如果指定的 *mode* 是 FEX\_NONSTOP、FEX\_NOHANDLER 或 FEX\_ABORT，*handler* 参数会被忽略。如果为指出的异常建立了指定模式，`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。要在遇到 0/0 异常时终止，可以用：

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

要为上溢和被零除安装 SIGFPE 处理程序，可以用：

```
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 *sig* 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，下面的示例打印异常的有关信息，这些信息将提供给在 FEX\_CUSTOM 模式下安装的处理程序。（要查看更多信息，请参见 `fex_set_handling(3m)` 手册页。）

代码示例 4-2 打印提供给在 FEX\_CUSTOM 模式下安装的处理程序的信息

```
#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);
```

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

## 诊断回顾

使用 `libm9x.so` 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诊断回顾消息的记录。当您启用对诊断回顾消息的记录时，系统会记录有关某些异常的信息。这些信息包括异常的类型、导致它的指令的地址、将要处理它的方式、类似于调试器生成的跟踪的栈跟踪。（用诊断回顾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诊断回顾日志中；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：

- 同一异常以前在同一位置（即相同的指令地址和堆栈跟踪）被记录，或者
- FEX\_NONSTOP 模式对于该异常有效，而且它的标志以前被引发过。

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

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

```
fex_set_log(stderr);
```

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

代码示例 4-3 对诊断回顾消息的记录和共享对象预装功能组合在一起

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

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

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

#pragma init (set_ftrap)
```

代码示例 4-3 对诊断回顾消息的记录和共享对象预装功能组合在一起 (续)

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

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

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

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

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

在 SPARC 上, 组合使用上面的代码和本节开头处提供的示例程序可生成下列结果:

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

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

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

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



---

## 处理异常

在以前，大多数数值软件在编写时都未考虑异常（原因多种多样），许多程序员已经习惯于异常会导致程序立即终止的环境。现在，一些高质量的软件包（如 LAPACK）经过了仔细设计，从而避免了异常（如被零除和无效运算）并主动对输入进行缩放以排除上溢和可能有害的下溢。在处理异常的这些方法中，没有一种能够适合所有情况。但是，当一个人编写的程序或子例程将要由他人（可能是没有源代码访问权限的人）使用时，忽略异常可能会导致问题，而且尝试避免所有异常可能需要大量防错测试和分支，并且还会带来巨大成本（请参见 Demmel 和 Li 著的“Faster Numerical Algorithms via Exception Handling”，《IEEE Trans.Comput.》，第 43 期（1994），位于第 983 到 992 页。）

IEEE 算法的缺省异常响应、状态标记和可选的捕获功能旨在提供第三种替代方案：在异常存在的情况下继续计算，然后或者在事后检测它们，或者在它们发生时截获和处理它们。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，并安装一个处理程序以便在异常发生时截获它们。但是，为了使计算能够继续，IEEE 标准推荐捕获处理程序应该能够为引发异常的操作提供一个结果。通过使用 Solaris 操作系统提供给信号处理程序的 `uap` 参数，`ieee_handler` 或 `fex_set_handling` 在 `FEX_SIGNAL` 模式下安装的 `SIGFPE` 处理程序可以做到这一点。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。

回忆一下，在 C 中可按如下方式声明 `SIGFPE` 信号处理程序：

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

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

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

可按如下方式声明 FEX\_CUSTOM 模式处理程序：

```
#include <fenv.h>

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

当 FEX\_CUSTOM 处理程序被调用时，*ex* 参数指出所发生的异常的类型（该类型是列在表 4-5 中的某个值），*info* 参数指向一个包含更多异常信息的数据结构。特别是，该结构中包含一个代码（代表导致了该异常的算术运算）和多个用来记录操作数（如果它们可用的话）的结构。它还包含一个结构，在结构中记录了缺省结果，如果不是因为异常被捕获的话，这些缺省结果本来是要用来替换的；在结构中还保存了一个整数，整数的值是所有应计的异常标志的按位“或”。此处理程序可以修改该结构后面的成员，以便替换另一个不同结果或者更改应计标志的设置。（请注意，如果该处理程序在不修改这些数据的情况下返回，则程序将以未捕获的缺省结果和标志继续运行，就好像异常未被捕获一样。）

下一节举例说明如何用缩放结果替换下溢或上溢。要查看进一步的示例，请参见附录 A。

## 替换 IEEE 捕获的下溢 / 上溢结果

IEEE 标准建议：在捕获到下溢或上溢时，系统应当为捕获处理程序提供一种方法来替换指数折返的结果，即，该值与将成为下溢或上溢运算的舍入结果值相一致（不同之处在于指数在其通常范围的末端是被折返的），从而有效地按 2 的幂缩放结果。在选择比例因子时，使下溢或上溢结果与指数范围的中间尽可能接近，这样将减少以后的计算进一步出现下溢或上溢的可能性。通过跟踪所发生的下溢或上溢的数量，程序可通过缩放最终结果来补偿指数折返。这种下溢 / 上溢“计数模式”可用于在原本会超出浮点格式表示范围的计算中产生准确结果。（请参见 P. Sterbenz 的 *Floating-Point Computation*。）

在 SPARC 上，当浮点指令导致捕获异常时，系统会使目标寄存器保持不变。因此，为了替换指数折返结果，下溢 / 上溢处理程序必须对指令对象解码、检查操作数寄存器、生成缩放结果本身。代码示例 4-4 显示了一个执行这三个步骤的处理程序。（为了使处理程序能和在 UltraSPARC 系统上编译的代码一起使用，请在运行 Solaris 2.6、Solaris 7 或 Solaris 8 的系统上编译该处理程序，并定义处理器标记 V8PLUS。）

代码示例 4-4 对于 SPARC 系统替换 IEEE 捕获的下溢 / 上溢处理程序结果

```
#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_ptr. 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是：

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

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

代码示例 4-5 对于 x86 系统替换 IEEE 捕获的下溢 / 上溢处理程序结果

```
#include <stdio.h>
#include <ieeefp.h>
#include <math.h>
#include <sunmath.h>
#include <signinfo.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;
```

代码示例 4-5 对于 x86 系统替换 IEEE 捕获的下溢 / 上溢处理程序结果 (续)

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

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

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

正如在 SPARC 一样，上述程序在 x86 上的输出结果是：

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

C/C++ 程序可以使用 `libm9x.so` 中的 `fex_set_handling` 函数来为下溢和上溢安装 `FEX_CUSTOM` 处理程序。在 SPARC 系统上，提供给类似处理程序的信息总是包括导致了异常的运算以及操作数，这些信息足以允许该处理程序计算 IEEE 指数折返结果，如上所示。在 x86 上，可用信息并不总是指出导致了异常的特定运算；例如，当异常由某个超验指令引发时，`info->op` 参数设置为 `fex_other`。（要查看定义，请参见 `fenv.h` 文件。）而且，x86 硬件自动提供指数折返结果，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，用来替换下溢或上溢运算的 IEEE 指数折返结果。当捕获到其中的任一异常时，该处理程序会设置

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

指出应当提供指数折返结果。下面的示例显示类似的处理程序：

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

void handler(int ex, fex_info_t *info) {
    info->res.type = fex_nodata;
}

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

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

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

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

    return 0;
}
```

上述程序的输出结果类似于如下内容:

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

# 附录 A

---

## 示例

---

本附录提供有关如何完成某些常见任务的示例。这些示例是用 Fortran 或 ANSI C 编写的，很多示例依赖于当前的 libm 和 libsunmath 版本。这些示例已经在 Solaris 操作环境中使用当前的 C 和 Fortran 编译器进行了测试。

---

## IEEE 算法

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

以下 C 程序打印  $\pi$  的双精度近似值和单精度无穷大：

代码示例 A-1      双精度示例

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

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

    /* double precision */
```

代码示例 A-1          双精度示例 (续)

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

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

return 0;
}
```

在 SPARC® 上, 上一程序的输出类似以下内容:

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

以下 Fortran 程序打印每种格式的最小正规数:

代码示例 A-2          打印每种格式的最小正规数

```
program print_ieee_values
c
c the purpose of the implicit statements is to ensure
c that the floatingpoint pseudo-intrinsic functions
c are declared with the correct type
c
implicit real*16 (q)
implicit double precision (d)
implicit real (r)
real*16 z
double precision x
real r

c
z = q_min_normal()
write(*,7) z, z
7 format('min normal, quad: ',1pe47.37e4,/, ' in hex ',z32.32)
c
x = d_min_normal()
```



代码示例 A-2 打印每种格式的最小正规数 (续)

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

在 SPARC 上, 相应的输出为:

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

---

## 数学库

本节介绍使用数学库中函数的示例。

### 随机数生成器

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

代码示例 A-3 随机数生成器

```
#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

```

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

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

代码示例 A-4          使用 `d_addrans`

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

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

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

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

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

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

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

代码示例 A-4 使用 d\_addrans (续)

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

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

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

## IEEE 建议的函数

此 Fortran 示例使用 IEEE 标准建议的某些函数:

代码示例 A-5 IEEE 建议的函数

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

```

c      Refer to
c      ieee_functions(3m)
c      libm_double(3f)
c      libm_single(3f)
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)
c
c      program ieee_functions_demo
c      implicit double precision (d)
c      implicit real (r)
c      double precision  x, y, z, direction
c      real              r, s, t, r_direction
c      integer           i, scale
c
c      print *
c      print *, 'DOUBLE PRECISION EXAMPLES:'
c      print *
c
c      x = 32.0d0
c      i = id_ilogb(x)
c      write(*,1) x, i
1      format(' The base 2 exponent of ', F4.1, ' is ', I2)
c
c      x = -5.5d0
c      y = 12.4d0
c      z = d_copysign(x,y)
c      write(*,2) x, y, z
2      format(F5.1, ' was given the sign of ', F4.1,
c      *      ' 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
```

代码示例 A-5 IEEE 建议的函数 (续)

```
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
```

代码示例 A-6 中显示了此程序的输出。

代码示例 A-6 代码示例 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
```

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

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

## IEEE 特殊值

此 C 程序调用几个 `ieee_values(3m)` 函数:

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

int main()
{
    double      x;
    float       r;

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

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

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

    return 0;
}
```

请记住: 在链接时要同时指定 `-lsunmath` 和 `-lm`。

在 SPARC 上, 输出类似如下内容:

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



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

```
quiet NaN: NaN = ffffffff 7fffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                             = ffffffff000fffff
single precision min subnormal = 1.40129846e-45 = 00000001
```

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

```
      program print_ieee_values
      c
      c the purpose of the implicit statements is to insure
      c that the floatingpoint pseudo-intrinsic
      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，输出如下：

```
6.4751751194380251109244389582276466-4966
Inf
3.40282E+38
```

## ieee\_flags — 舍入方向

以下示例说明如何将舍入模式设置为*向零舍入*：

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

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

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

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

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

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

    return 0;
}
```

(SPARC) 此简短程序的输出显示向零舍入的效果:

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

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

(x86) 此简短程序的输出显示向零舍入的效果:

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

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

从 Fortran 程序中将舍入方向设置为向零舍入:

```
program ieee_flags_demo
character*16 out

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

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

end
```

输出如下:

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

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

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

## C99 浮点环境函数

下一个示例说明如何使用某些 C99 浮点环境函数。`norm` 函数计算矢量的欧几里得正规数，并使用环境函数来处理下溢和上溢。主程序使用缩放的矢量调用此函数以确保发生下溢和上溢（如诊断回顾输出所示）。

代码示例 A-7 C99 浮点环境函数

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

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

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

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

```
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 SPARC 上，编译和运行此程序可生成以下消息：

```

demo% cc norm.c -R/opt/SUNWspro/lib -L/opt/SUNWspro/lib -lm9x
-lsunmath -lm
demo% a.out
Floating point underflow at 0x000153a8 __d_lcrans_, nonstop mode
 0x000153b4 __d_lcrans_
 0x00011594 main
Floating point underflow at 0x00011244 norm, nonstop mode
 0x00011248 norm
 0x000115b4 main
norm: 1.32533e-307
Floating point overflow at 0x00011244 norm, nonstop mode
 0x00011248 norm
 0x00011660 main
norm: 2.02548e+155

```

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

```

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

```

代码示例 A-8          fesetprec 函数 (x86) (续)

```
int main()
{
    double x;

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

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

    return 0;
}
```

在 x86 系统上，此程序的输出为：

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

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

代码示例 A-9          在多线程程序中使用环境函数

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

fenv_t env;

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

```
...
/* save the child's environment before exit */
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);
    ...
}
```

---

## 异常和异常处理

### ieee\_flags — 产生的异常

通常，用户程序检查或清除产生的异常位。代码示例 A-10 是一个检查产生的异常标记的 C 程序。

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

int main()
```



代码示例 A-10 检查产生的异常标记 (续)

```

{
    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;
}

```

运行此程序的输出为:

```

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

```

可以从 Fortran 中执行相同的操作:

代码示例 A-11 检查产生的异常标记 - Fortran

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

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

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

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

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

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

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

输出如下:

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

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

```
#include <sunmath.h>

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

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

    return 0;
}
```

在 SPARC 上, 以上程序的输出为:

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

在 x86 上, 输出为:

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

## ieee\_handler — 捕获异常

---

**注意** - 以下示例仅适用于 Solaris 操作环境。

---

以下是一个 Fortran 程序，它安装一个信号处理程序以定位异常（仅用于 SPARC 系统）：

代码示例 A-12 捕获“下溢”异常 - SPARC

```
program demo

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

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

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

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

  end

c
c the floating-point exception handling function
c
  integer function fp_exc_hdl(sig, sip, uap)
  integer sig, code, addr
  character label*16

c
c The structure /siginfo/ is a translation of siginfo_t
c from <sys/siginfo.h>
c

  structure /fault/
    integer address
  end structure

  structure /siginfo/
    integer si_signo
    integer si_code
```

代码示例 A-12      捕获“下溢”异常 - SPARC (续)

```

        integer si_errno
        record /fault/ fault
    end structure

    record /siginfo/ sip

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

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

    end

```

输出为:

```

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

```

(SPARC) 以下是一个更复杂的 C 示例:

代码示例 A-13      捕获“无效值”、“被 0 除”、“上溢”、“下溢”和“不准确的值”异常 - 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");
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:
```



代码示例 A-13      捕获“无效值”、“被 0 除”、“上溢”、“下溢”和“不准确的值”异常 - SPARC (续)

```
        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);
}
```

输出类似于以下内容:

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

(SPARC) 代码示例 A-14 说明如何使用 `ieee_handler` 和包含文件来修改某些异常情况的缺省结果:

代码示例 A-14      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 {

```

代码示例 A-14 修改异常情况的缺省结果 (续)

```
        uap->uc_mcontext.fpregs.fpu_fr.fpu_dregs[rd/2] = d_val;
    }
}
```

正如所预期的一样, 输出为:

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

## ieee\_handler — 出现异常时终止

可以使用 `ieee_handler` 强制程序在出现某些浮点异常时终止:

```
#include <floatingpoint.h>
    program abort
c
    ieeeer = ieee_handler('set', 'division', SIGFPE_ABORT)
    if (ieeeer .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
```

## libm9x.so 异常处理功能

以下示例说明如何使用 `libm9x.so` 提供的某些异常处理功能。第一个示例基于以下任务：给定一个数  $x$  及系数  $a_0, a_1, \dots, a_N$  和  $b_0, b_1, \dots, b_{N-1}$ ，计算函数  $f(x)$  及其一阶导数  $f'(x)$ ，其中  $f$  是连分式

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

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

预替换是 IEEE 缺省异常响应的扩展，可以使用户提前指定用于替换异常运算结果的值。通过使用 `libm9x.so`，程序可在 `FEX_CUSTOM` 异常处理模式下安装处理程序以方便地实现预替换。这种模式允许处理程序只需简单地使用以下方法即可为异常运算结果提供任何值：通过将该值存储在传递给处理程序的 `info` 参数所指向的数据结构即可实现。以下示例程序使用 `FEX_CUSTOM` 处理程序实现的预替换来计算连分式及其导数。

代码示例 A-15      使用 `FEX_CUSTOM` 处理程序计算连分式及其导数

```
#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;
}
```

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

主程序通过调用 `fex_set_log` 函数来启用对诊断回顾的记录。在执行此操作之前，它引发不准确标记；这对防止记录不准确异常有效。（让我们回想一下，正如第 94 页上的“诊断回顾”中所解释的一样，在 `FEX_NONSTOP` 模式下，如果引发异常标记，则不记录该异常。）主程序还为一般异常设置 `FEX_ABORT` 模式，以确保任何 `continued_fraction` 没有显式处理的不常见异常都导致程序终止。最后，程序在几个不同的点处计算某个连分式。正如以下示例输出所示，计算确实遇到中间异常：

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

（在  $x = 1$ 、 $4$  和  $5$  时计算  $f'(x)$  的过程中发生的异常不会导致诊断回顾消息，因为它们发生在程序中的位置与  $x = -3$  时发生的异常相同。）

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



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

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

代码示例 A-16      使用标记处理异常

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

/*
 * Evaluate the continued fraction given by coefficients a[j] and
 * b[j] at the point x; return the function value in *pf and the
 * derivative in *pfl
 */
void continued_fraction(int N, double *a, double *b,
                        double x, double *pf, double *pfl)
{
    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，第一个循环试图在缺省不间断模式下计算  $f(x)$  和  $f'(x)$ 。如果引发无效标记，则第二个循环重新计算  $f(x)$  和  $f'(x)$  以显式地测试是否出现 NaN。通常，不会发生无效运算异常，因此程序只执行第一个循环。该循环不引用 `volatile` 变量，并且不执行额外的算术运算，因此，它可达到编译器提供的最大运行速度。要获得此效率的代价就是，用户必须编写与第一个循环基本相同的第二个循环以处理发生异常时的情况。这种折衷办法是浮点异常处理所引起的典型的两难问题。

## 将 libm9x.so 用于 Fortran 程序

libm9x.so 主要用于 C/C++ 程序，但通过使用 Sun Fortran 语言互操作性功能，您也可以从 Fortran 程序中调用某些 libm9x.so 函数。

---

**注意 -** 为了获得一致的行为，请不要在相同程序中同时使用 libm9x.so 异常处理函数以及 ieee\_flags 和 ieee\_handler 函数。

---

以下示例显示 Fortran 版本的程序，该程序使用预替换来计算连分式及其导数（仅限 SPARC）：

代码示例 A-17      使用预替换来计算连分式及其导数 - SPARC

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

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

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

```

        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

```

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

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

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

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

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

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

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

```

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

      external fex_set_handling
c$pragma c(fex_set_handling)

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

```

此程序的输出为:

```

f(-5) = -1.59649      , f'(-5) = -.181800
f(-4) = -1.87302     , f'(-4) = -.428193
f(-3) = -3.00000     , f'(-3) = -3.16667
f(-2) = -.444089E-15, f'(-2) = -3.41667
f(-1) = -1.22222     , f'(-1) = -.444444
f( 0) = -1.33333     , f'( 0) = 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; 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)

```

---

## 杂项

### sigfpe — 捕获整数异常

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

---

**注意** — `sigfpe` 仅可在 Solaris 操作环境中使用。

---

(*SPARC*) 在某些情况下（如捕获整数运算异常），`sigfpe` 是要使用的处理程序。代码示例 A-18 捕获“整数被零除”异常。

代码示例 A-18      捕获整数异常

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

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

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

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

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

    /*
     * Integer division-by-zero aborts unless a signal
     * handler for integer division by zero is set up
     */

    sigfpe(FPE_INTDIV, int_handler);

    a = 4;
```

代码示例 A-18 捕获整数异常 (续)

```
        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];
}
```

## 从 C 中调用 Fortran

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

代码示例 A-19 从 C 中调用 FORTRAN

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

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

int main()
{
```



```
double array[3][4];
float f, g;
double x, y;
int i, j;

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

g = 1.5;
y = g;

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

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

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

ieee_retrospective_();
return 0;
}
```

将 Fortran 子例程保存在名为 drivee.f 的文件中:

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

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

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

然后, 执行编译和链接:

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

输出类似如下内容:

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

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

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

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

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

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

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

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

x = sin(y) from a double precision fortran function
  x, y:9.97494986604054446e-01, 1.500000000000000000e+00
```

## 有用的调试命令

表 A-1 显示用于 SPARC 体系结构的调试命令示例。

表 A-1 一些调试命令 (SPARC)

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

表 A-2 显示用于 x86 体系结构的调试命令示例。

表 A-2 一些调试命令 (x86)

| 操作          | dbx                               | adb                |
|-------------|-----------------------------------|--------------------|
| 设置断点        |                                   |                    |
| 在函数中        | stop in myfunct                   | myfunct:b          |
| 在行号中        | stop at 29                        |                    |
| 在绝对地址中      |                                   | 23a8:b             |
| 在相对地址中      |                                   | main+0x40:b        |
| 运行程序直到遇到断点  | run                               | :r                 |
| 检查源代码       | list                              | <pc,10?ia          |
| 检查 fp 寄存器   | print \$st0<br>...<br>print \$st7 | \$x                |
| 检查所有寄存器     | examine &\$gs/19X                 | \$r                |
| 检查 fp 状态寄存器 | examine &\$fstat/X                | <fstat=X<br>or \$x |
| 继续执行        | cont                              | :c                 |
| 单步执行        | step (or next)                    | :s                 |
| 退出调试器       | quit                              | \$q                |

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

```
myfunction:b
```

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

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

对 adb 而言，使用 f95 编译的 Fortran 程序中的主子例程称为 MAIN\_。要在 adb 中的 MAIN\_ 处设置断点：

```
MAIN_:b
```

在检查浮点寄存器的内容时，`dbx` 命令 `regs -F` 显示的十六进制值是以 16 为基数的表示法，而不是该数的十进制表示法。对于 SPARC，`adb` 命令 `$x` 和 `$X` 显示十六进制的表示法和十进制值。对于 x86，`adb` 命令 `$x` 只显示十进制值。对于 SPARC，双精度值在奇数寄存器旁边显示十进制值。

因为操作系统在进程第一次使用浮点单元之前总是禁用浮点单元，所以只有在被调试的程序访问浮点寄存器之后，您才可以修改这些寄存器。

(SPARC) 在显示浮点数时，应该牢记：寄存器的大小为 32 位，单精度浮点数占用 32 位（因此可将它放在一个寄存器中），而双精度浮点数占用 64 位（因此，使用两个寄存器来保存双精度数）。在十六进制表示中，32 位对应于 8 位数字。在以下使用 `adb` 显示的 FPU 寄存器快照中，显示的组织形式如下：

`<fpu 寄存器的名称> <IEEE 十六进制值> <单精度> <双精度>`

(SPARC) 第三列保存显示在第二列中的十六进制模式的单精度十进制解释。第四列解释寄存器对。例如，`f11` 行的第四列将 `f10` 和 `f11` 解释为 64 位 IEEE 双精度数。

(SPARC) 因为 `f10` 和 `f11` 用于保存双精度值，所以（在 `f10` 行上）将该值的前 32 位 `7ff00000` 解释为 `+NaN` 是不正确的。将所有 64 位 `7ff00000 00000000` 解释为 `+Infinity` 则有可能是有意义的转换。

(SPARC) 用于显示前 16 个浮点数据寄存器的 `adb` 命令 `$x` 还显示 `fsr`（浮点状态寄存器）：

| <code>\$x</code> |                       |                             |                                       |
|------------------|-----------------------|-----------------------------|---------------------------------------|
| <code>fsr</code> | <code>40020</code>    |                             |                                       |
| <code>f0</code>  | <code>400921fb</code> | <code>+2.1426990e+00</code> |                                       |
| <code>f1</code>  | <code>54442d18</code> | <code>+3.3702806e+12</code> | <code>+3.1415926535897931e+00</code>  |
| <code>f2</code>  | <code>2</code>        | <code>+2.8025969e-45</code> |                                       |
| <code>f3</code>  | <code>0</code>        | <code>+0.0000000e+00</code> | <code>+4.2439915819305446e-314</code> |
| <code>f4</code>  | <code>40000000</code> | <code>+2.0000000e+00</code> |                                       |
| <code>f5</code>  | <code>0</code>        | <code>+0.0000000e+00</code> | <code>+2.0000000000000000e+00</code>  |
| <code>f6</code>  | <code>3de0b460</code> | <code>+1.0971904e-01</code> |                                       |
| <code>f7</code>  | <code>0</code>        | <code>+0.0000000e+00</code> | <code>+1.2154188766544394e-10</code>  |
| <code>f8</code>  | <code>3de0b460</code> | <code>+1.0971904e-01</code> |                                       |
| <code>f9</code>  | <code>0</code>        | <code>+0.0000000e+00</code> | <code>+1.2154188766544394e-10</code>  |
| <code>f10</code> | <code>7ff00000</code> | <code>+NaN</code>           |                                       |
| <code>f11</code> | <code>0</code>        | <code>+0.0000000e+00</code> | <code>+Infinity</code>                |
| <code>f12</code> | <code>ffffffff</code> | <code>-NaN</code>           |                                       |
| <code>f13</code> | <code>ffffffff</code> | <code>-NaN</code>           | <code>-NaN</code>                     |
| <code>f14</code> | <code>ffffffff</code> | <code>-NaN</code>           |                                       |
| <code>f15</code> | <code>ffffffff</code> | <code>-NaN</code>           | <code>-NaN</code>                     |

(x86) x86 上的相应输出类似于以下内容:

```
$x
80387 chip is present.
cw      0x137f
sw      0x3920
cssel 0x17 ipoff 0x2d93          dataset1 0x1f  dataoff 0x5740

st[0]  +3.24999988079071044921875 e-1          VALID
st[1]  +5.6539133243479549034419688 e73       EMPTY
st[2]  +2.00000000000000008881784197         EMPTY
st[3]  +1.8073218308070440556016047 e-1       EMPTY
st[4]  +7.9180300235748291015625 e-1         EMPTY
st[5]  +4.201639036693904927233234 e-13      EMPTY
st[6]  +4.201639036693904927233234 e-13      EMPTY
st[7]  +2.7224999213218694649185636          EMPTY
```

---

注意 - (x86) cw 是控制字; 而 sw 是状态字。

---





# SPARC 行为和实现

---

本章讨论与 SPARC® 工作站中所使用的浮点单元有关的问题，并介绍一种用来确定哪个代码生成标志最适合特定工作站的方法。

---

## 浮点硬件

本节列出许多 SPARC 浮点单元，并介绍它们所支持的指令集和异常处理功能。关于以下情况的信息，包括在捕获到异常时所发生情况的简要说明、捕获到的下溢和未捕获到的下溢之间的区别，以及建议的提供非 IEEE（非标准）算法模式的 SPARC 实现可能的行为过程，请参见 *SPARC Architecture Manual* 第 8 版的附录 N “SPARC IEEE 754 Implementation Recommendations” 和第 9 版的附录 B “IEEE Std 754-1985 Requirements for SPARC-V9”。

表 B-1 列出了由 SPARC 工作站使用的硬件浮点实现。许多早期的 SPARC 系统都有浮点单元，这些单元来自于由 TI 或 Weitek 开发的核心：

- TI 系列 - 包括 TI8847 和 TMS390C602A
- Weitek 系列 - 包括 1164/1165、3170 和 3171

这两个系列的 FPU 都已经授权给其他工作站供应商，因此有可能在某些 SPARC 工作站中找到其他半导体制造商的芯片。下表也显示了其中的一些芯片。

表 B-1 SPARC 浮点选项

| FPU                                  | 说明或处理器名称                            | 适于机型  | 注释   | 最佳的 -xchip 和 -xarch         |
|--------------------------------------|-------------------------------------|---|--|-----------------------------|
| 基于 Weitek 1164/1165 的 FPU<br>或没有 FPU | 内核模拟浮点指令                            | 已废弃   | 慢；不推荐  | -xchip=old<br>-xarch=v7     |
| 基于 TI 8847 的 FPU                     | TI 8847 ;<br>Fujitsu 或 LSI 的<br>控制器 | Sun-4™/1xx<br>Sun-4/2xx<br>Sun-4/3xx<br>Sun-4/4xx<br>SPARCstation® 1 (4/60)                   | 1989<br>多数 SPARCstation 1<br>工作站都使用<br>Weitek 3170 | -xchip=old<br>-xarch=v7     |
| 基于 Weitek 3170 的 FPU                 |                                     | SPARCstation 1 (4/60)<br>SPARCstation 1+ (4/65)   | 1989、1990  | -xchip=old<br>-xarch=v7     |
| TI 602a                              |                                     | SPARCstation 2 (4/75)   | 1990   | -xchip=old<br>-xarch=v7     |
| 基于 Weitek 3172 的 FPU                 |                                     | SPARCstation SLC (4/20)<br>SPARCstation IPC (4/40)  | 1990   | -xchip=old<br>-xarch=v7     |
| Weitek 8601 或<br>Fujitsu 86903       | 集成 CPU 和 FPU                        | SPARCstation IPX (4/50)<br>SPARCstation ELC (4/25)  | 1991<br>IPX 使用 40 MHz<br>CPU/FPU；ELC 使<br>用 33 MHz | -xchip=old<br>-xarch=v7     |
| Cypress 602                          | 驻留在 Mbus 模<br>块                     | SPARCserver® 6xx  | 1991   | -xchip=old<br>-xarch=v7     |
| TI TMS390S10<br>(STP1010)            | microSPARC®-I                       | SPARCstation LX<br>SPARCclassic   | 1992<br>硬件中没有 FsMULd                               | -xchip=micro<br>-xarch=v8a  |
| Fujitsu 86904<br>(STP1012)           | microSPARC-II                       | SPARCstation 4 和 5<br>SPARCstation Voyager  | 硬件中没有 FsMULd                                       | -xchip=micro2<br>-xarch=v8a |
| TI TMS390Z50<br>(STP1020A)           | SuperSPARC®-I                       | SPARCserver 6xx<br>SPARCstation 10<br>SPARCstation 20<br>SPARCserver 1000<br>SPARCcenter 2000 |  | -xchip=super<br>-xarch=v8   |
| STP1021A                             | SuperSPARC-II                       | SPARCserver 6xx<br>SPARCstation 10<br>SPARCstation 20<br>SPARCserver 1000<br>SPARCcenter 2000 |  | -xchip=super2<br>-xarch=v8  |

表 B-1 SPARC 浮点选项 (续)

| FPU           | 说明或处理器名称        | 适于机型   | 注释        | 最佳的 <code>-xchip</code> 和 <code>-xarch</code>              |
|---------------|-----------------|--|-----------|--|
| Ross RT620    | hyperSPARC®     | SPARCstation 10/HSxx<br>SPARCstation 20/HSxx                             |           | <code>-xchip=hyper</code><br><code>-xarch=v8</code>        |
| Fujitsu 86907 | TurboSPARC      | SPARCstation 4 和 5   |           | <code>-xchip=micro2</code><br><code>-xarch=v8</code>       |
| STP1030A      | UltraSPARC® I   | Ultra-1、 Ultra-2<br>Ex000  | V9+VIS    | <code>-xchip=ultra</code><br><code>-xarch=v8plusa</code>   |
| STP1031       | UltraSPARC II   | Ultra-2、 E450<br>Ultra-30、 Ultra-60、<br>Ultra-80、 Ex500<br>Ex000、 E10000 | V9+VIS    | <code>-xchip=ultra2</code><br><code>-xarch=v8plusa</code>  |
| SME1040       | UltraSPARC IIIi | Ultra-5、 Ultra-10  | V9+VIS    | <code>-xchip=ultra2i</code><br><code>-xarch=v8plusa</code> |
|               | UltraSPARC IIe  | Sun Blade™ 100   | V9+VIS    | <code>-xchip=ultra2e</code><br><code>-xarch=v8plusa</code> |
|               | UltraSPARC III  | Sun Blade 1000<br>Sun Blade 2000   | V9+VIS II | <code>-xchip=ultra3</code><br><code>-xarch=v8plusb*</code> |

\* 使用 `v8plusb` 生成的可执行文件只能在 UltraSPARC III 系统上运行。要在所有 UltraSPARC (I,II,III) 系统上运行, 必须将 `-xarch` 设置为 `v8plusa`。

上表中的最后一列显示要用来为每个 FPU 获得最快代码的编译器标志。这些标志控制代码生成的两个独立属性: `-xarch` 标志确定编译器可以使用的指令集, `-xchip` 标志确定编译器在调度代码时将针对处理器的性能特点进行的假设。因为所有的 SPARC 浮点单元都至少实现在 *SPARC Architecture Manual* 第 7 版中定义的浮点指令集, 所以用 `-xarch=v7` 编译的程序将可以在任何 SPARC 系统上运行 (尽管它可能无法充分利用较新处理器的功能)。同样, 用特定的 `-xchip` 值编译的程序能够运行于支持用 `-xarch` 指定的指令集的任何 SPARC 系统, 但是如果在非指定处理器的系统上运行时, 则速度会下降。

在上表中, 排列在 `microSPARC-I` 前面的浮点单元实现了在 *SPARC Architecture Manual* 第 7 版中定义的浮点指令集; 应当使用 `-xarch=v7` 来编译这些必须运行于含有这类浮点单元 (FPU) 的系统上的程序。编译器不对这些处理器的性能特点进行特殊假设, 因此它们都共用一个 `-xchip` 选项 `-xchip=old`。(并非所有列在表 B-1 中的系统都仍然受到编译器的支持; 列出它们的目的仅在于展现其历史。有关可与支持这些系统的编译器一起使用的代码生成标志的信息, 请参考适当版本的《数值计算指南》。)

microSPARC-I 和 microSPARC-II 浮点单元实现在 *SPARC Architecture Manual* 第 8 版中定义的浮点指令集（FsMULd 和四倍精度指令除外）。用 `-xarch=v8` 编译的程序将在使用这些处理程序的系统上运行，但是由于未实现的浮点指令必须由系统内核模拟，因此大量使用 FsMULd 的程序（如执行大量单精度复数运算的 Fortran 程序）的性能可能会严重下降。为避免出现该问题，请用 `-xarch=v8a` 针对包含这些处理程序的系统编译程序。

SuperSPARC-I、SuperSPARC-II、hyperSPARC 和 TurboSPARC 浮点单元实现了在 *SPARC Architecture Manual* 第 8 版中定义的浮点指令集（四倍精度指令除外）。要在包含这些处理器的系统上获得最佳性能，请使用 `-xarch=v8` 进行编译。

UltraSPARC I、UltraSPARC II、UltraSPARC Iie、UltraSPARC Ili 和 UltraSPARC III 浮点单元实现了在 *SPARC Architecture Manual* 第 9 版中定义的浮点指令集（四倍精度指令除外）；特别是，它们提供 32 个双精度浮点寄存器。为了允许编译器使用这些寄存器，请使用 `-xarch=v8plus`（对于在 32 位 OS 下运行的程序）或 `-xarch=v9`（对于在 64 位 OS 下运行的程序）进行编译。这些处理器还提供了标准指令集的扩展。其它被称作 Visual Instruction Set（可视指令集）或 VIS 的指令很少由编译器自动生成，但是它们可用在汇编代码中。因此，为了充分利用这些处理器所支持的指令集，请使用 `-xarch=v8plusa`（32 位）或 `-xarch=v9a`（64 位）。

可以使用 `-xtarget` 宏选项同时指定 `-xarch` 和 `-xchip` 选项。（即，`-xtarget` 标志仅扩展到 `-xarch`、`-xchip` 和 `-xcache` 标志的适当组合。）缺省的代码生成选项是 `-xtarget=generic`。要查看更多信息（包括 `-xarch`、`-xchip` 和 `-xtarget` 值的完整列表），请参见 `cc(1)`、`CC(1)` 和 `f95(1)` 手册页以及编译器手册。其他 `-xarch` 信息在《Fortran 用户指南》、《C 用户指南》和《C++ 用户指南》中提供。

## 浮点状态寄存器和队列

对于所有的 SPARC 浮点单元来说，无论它们实现哪种版本的 SPARC 结构，它们都提供一个浮点状态寄存器 (FSR)，该寄存器中包含与 FPU 相关的状态位和控制位。所有实现延迟浮点捕获的 SPARC FPU 都提供一个浮点队列 (FQ)，该队列中包含有关当前执行的浮点指令的信息。FSR 可由用户软件访问，以检测已经发生的浮点异常，并控制舍入方向、捕获和非标准的算法模式。FQ 由操作系统的内核使用，以便处理浮点捕获；用户软件通常不可访问它。

软件通过 `STFSR` 和 `LDFSR` 指令来访问浮点状态寄存器，这两个指令的作用分别是将 FSR 存储在内存中和从内存中加载它。在 SPARC 汇编语言中，这些指令按如下方式编写：

|                 |                           |               |
|-----------------|---------------------------|---------------|
| <code>st</code> | <code>%fsr, [addr]</code> | ! 在指定地址存储 FSR |
| <code>ld</code> | <code>[addr], %fsr</code> | ! 从指定地址加载 FSR |

内联模板文件 `libm.il` 所在的目录中包含随 Sun Studio 编译器提供的库，该文件包含 `STFSR` 和 `LDFSR` 指令的用法示例。

图 B-1 显示浮点状态寄存器中位字段的布局。

|       |       |       |    |       |       |       |        |       |      |      |
|-------|-------|-------|----|-------|-------|-------|--------|-------|------|------|
| RD    | res   | TEM   | NS | res   | ver   | ftt   | qneres | fcc   | aexc | cexc |
| 31:30 | 29:28 | 27:23 | 22 | 21:20 | 19:17 | 16:14 | 13 12  | 11:10 | 9:5  | 4:0  |

图 B-1 SPARC 浮点状态寄存器

在第 7 版本和第 8 版本的 SPARC 结构中，FSR 占用如上所示的 32 位。在第 9 版本中，FSR 扩展到 64 位，其中的低 32 位与该图相匹配；高 32 位大部分不使用，只包含了三个附加浮点条件代码字段。

在这里，res 是指保留位，ver 是用来标识 FPU 版本的只读字段，ftt 和 qne 由系统用来处理浮点捕获。其余字段将在下表中介绍。

表 B-2 浮点状态寄存器字段

| 字段   | 包含      |
|------|---------|
| RM   | 舍入方向模式  |
| TEM  | 捕获启用模式  |
| NS   | 非标准模式   |
| fcc  | 浮点条件代码  |
| aexc | 应计异常标志  |
| cexc | 当前的异常标志 |

RM 字段保留两个为浮点运算指定舍入方向的位。NS 位在实现它的 SPARC FPU 上启用非标准的算法模式；在其他系统上，该位将被忽略。fcc 字段保留由浮点比较指令生成的浮点条件代码并且由分支和条件移动运算使用。最后，TEM、aexc 和 cexc 字段包含五个位，这些位针对五个 IEEE 754 浮点异常中的每一个控制捕获并记录应计和当前异常标志。表 B-3 对这些字段进行了细分。

表 B-3 异常处理字段

| 字段            | 寄存器中的相应位  |           |          |          |          |
|---------------|-----------|-----------|----------|----------|----------|
| TEM (捕获启用模式)  | NVM<br>27 | OFM<br>26 | UFM 25   | DZM 24   | NXM 23   |
| aexc (应计异常标志) | nva<br>9  | ofa<br>8  | ufa<br>7 | dza<br>6 | nxa<br>5 |
| cexc (当前异常标志) | nvc<br>4  | ofc<br>3  | ufc<br>2 | dzc<br>1 | nxc<br>0 |

(上面的符号 NV、OF、UF、DZ 和 NX 分别代表无效运算、上溢、下溢、被零除和不精确异常。)

## 需要软件支持的特殊类

在大多数情况下，SPARC 的浮点单元可以在硬件中完成指令的执行而无需软件的支持。但是，在以下四种情况下，硬件将无法成功地完成浮点指令：

- 浮点单元被禁用。
- 指令未被硬件所实现（如，在基于 Weitek 1164/1165 的 FPU 上使用 `fsqrt[sd]`，在 `microSPARC-I` 和 `microSPARC-II` FPU 上使用 `fsmuld`，或者在任何 SPARC FPU 上使用四倍精度指令）。
- 硬件无法为指令操作数传送正确的结果。
- 指令将导致一个 IEEE 754 浮点异常（已经启用了该异常的捕获）。

在每种情况下，最初的响应都是相同的：进程被“捕获”到系统内核，由系统内核确定捕获的原因并采取相应的措施。（术语“捕获”是指正常控制流的中断。）在前三种情况下，内核在软件中模拟被捕获的指令。请注意，模拟的指令也可以导致一个异常，如果对该异常的捕获已经启用。

在上面的前三种情况下，如果模拟的指令不导致 IEEE 浮点异常（其捕获已经启用），则内核将完成该指令。如果该指令是浮点比较指令，内核将更新条件代码以反映结果；如果该指令是算术运算，内核将相应的结果传送到目标寄存器。内核还更新当前的异常标志以反映由该指令引发的任何（未捕获）异常，并将这些异常与应计异常标志进行“或”运算。内核随后安排继续从捕获点执行该进程。

当硬件执行或者内核软件模拟的某个指令导致一个 IEEE 浮点异常（其捕获已经启用），则该指令将无法完成。目标寄存器、浮点条件代码和应计异常标志保持不变，当前异常标志设置为反映导致了该捕获的特定异常，内核向该进程发送一个 SIGFPE 信号。

下面的伪代码概述了浮点捕获的处理。请注意，`aexc` 字段通常只能由软件清除。

```
FPop provokes a trap;
if trap type is fp_disabled, unimplemented_FPop, or
  unfinished_FPop then
    emulate FPop;
texc " all IEEE exceptions generated by FPop;
if (texc and TEM) = 0 then
    f[rd] " fp_result; // 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;
```

当内核必须模拟许多浮点指令时，程序的性能将严重下降。出现该问题的相对频率可能取决于多个因素（肯定包括捕获类型）。

在正常情况下，每个进程中只应当出现一次 `fp_disabled` 捕获。系统内核禁用浮点单元，直至某个进程被首次启动时为止，因此由该进程执行的第一个浮点运算将导致捕获。在处理完该捕获之后，内核启用浮点单元，并使其在以后的进程过程中保持启用状态。（可以针对整个系统禁用浮点单元，但是不建议这样做，而且只在出于内核或硬件调试目时才这样做。）

无论浮点单元何时遇到它未实现的指令，显然都会出现 `unimplemented_FPop` 捕获。由于目前大多数的 SPARC 浮点单元都至少实现了由 *SPARC Architecture Manual* 第 8 版定义的指令集（四倍精度指令除外），而且 Sun Studio 编译器不生成四倍精度指令，所以在大多数系统上应当不会出现这种类型的捕获。如上所述，两个明显的例外是 `microSPARC-I` 和 `microSPARC-II` 处理器，它们不实现 `FsMULd` 指令。为了避免在这些处理上出现 `unimplemented_FPop` 捕获，请用 `-xarch=v8a` 选项编译程序。

其余两个捕获类型 `unfinished_FPop` 和捕获的 IEEE 异常通常与涉及 NaN、无穷大和次正规数的特殊计算情况相关。

## IEEE 浮点异常、NaN 和无穷大

当某个浮点指令遇到一个 IEEE 浮点异常（其捕获已经启用）时，该指令将无法完成；相反，系统会向该进程发送一个 `SIGFPE` 信号。如果该进程已经建立了 `SIGFPE` 信号处理程序，则该处理程序将被调用，否则该进程将终止。多数程序不会产生被捕获的 IEEE 浮点异常，因为捕获被启用的目的常常是在异常发生时终止程序，终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 章中所述，可以安排信号处理程序为捕获指令提供结果并继续执行。请注意，如果捕获到许多浮点异常并用这种方式进行处理，则性能可能严重下降。

即使捕获行为被禁用或者指令不导致异常（其捕获已经启用），大多数 SPARC 浮点单元也会在某些情况下产生捕获，至少在涉及无穷大、NaN 操作数或 IEEE 浮点异常的情况下是这样。当硬件不支持类似的特殊情况时，将出现该问题；相反，它将生成 `unfinished_FPop` 捕获并使内核模拟软件完成指令。不同的 SPARC FPU 对导致 `unfinished_FPop` 捕获的条件会有不同的反应：例如，大多数早期的 SPARC FPU 和 `hyperSPARC FPU` 会捕获所有的 IEEE 浮点异常而无论捕获是否被启用；而在硬件无法确定某指令是否会导致某浮点异常并且该异常的捕获被启用的情况下，`UltraSPARC FPU` 将“悲观地”产生捕获。另一方面，`SuperSPARC-I`、`SuperSPARC-II`、`TurboSPARC`、`microSPARC-I` 和 `microSPARC-II FPU` 在硬件中处理各种异常情况，并且从不生成 `unfinished_FPop` 捕获。

由于大多数 `unfinished_FPop` 捕获都与浮点异常一起出现，因此程序可通过利用异常处理功能（即，测试异常标志、捕获和提交结果或者终止异常）来避免导致过量捕获。当然，必须认真平衡以下两个方面带来的成本：处理异常；允许异常导致 `unfinished_FPop` 捕获。

## 次正规数和非标准算法

SPARC 浮点单元产生 `unfinished_FPop` 捕获的大多数常见的情况都生成次正规数。当浮点运算生成次正规操作数或者必须生成一个非零的次正规结果（即，导致渐进下溢的结果）时，许多 SPARC FPU 将捕获。因为下溢较少见但是又很难通过编程来回避，而且因为下溢的中间结果的准确性对最终计算结果的影响很小，所以 SPARC 结构包括 *非标准算法模式*，该模式为用户提供一种方法，用来避免出现与涉及次正规数的 `unfinished_FPop` 捕获相关的性能的下降。

SPARC 结构不能准确定义非标准算法模式；它只是声明当该模式处于启用状态时，支持它的处理器会生成不遵循 IEEE 754 标准的结果。但是，所有支持该模式的现有 SPARC 实现都使用它来禁用渐进下溢，并将所有的次正规操作数和结果替换为零。有一个例外：Weitek 1164/1165 FPU 只在非标准模式下将次正规结果刷新为零，它们不将次正规操作数视为零。

并非所有的 SPARC 实现都提供非标准模式。特别是，SuperSPARC-I、SuperSPARC-II、TurboSPARC、microSPARC-I 和 microSPARC-II 浮点单元都处理次正规操作数并完全在硬件中生成次正规结果，因此，它们无需支持非标准算法。（对在这些处理器上启用非标准模式的任何尝试都将被忽略。）因此，渐进下溢在这些处理器上并不造成性能损失。

为了确定渐进下溢是否影响程序的性能，应当首先确定下溢是否确实发生，然后检查该程序占用了多少系统时间。为了确定是否出现下溢，可以使用数学库函数 `ieee_retrospective()` 来查看在程序退出时是否引发了下溢异常标志。在缺省情况下，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()` 会打印一则类似如下的消息：

```
注意：引发了 IEEE 浮点异常标记：
    不精确；下溢；
    请参见数字计算指南， ieee_flags(3M)
```

如果该程序遇到下溢，您可能希望通过用 `time` 命令来对该程序的执行进行计时，从而确定该程序占用多少系统时间。

```
demo% /bin/time myprog > myprog.output
305.3 real      32.4 user      271.9 sys
```



如果系统时间（上面显示的第三个数字）超长，则说明原因可能在于下溢数量太多。如果是这样的话，并且如果该程序不依赖渐进下溢的准确性，则可以启用非标准模式以获得更佳性能。可通过两种方法来完成此操作：第一，可以用 `-fns` 标志（这暗示着它是 `-fast` 和 `-fnonstd` 宏的一部分）进行编译，以便在程序启动时启用非标准模式。第二，增值的数学库 `libsunmath` 提供两个分别用来启用和禁用非标准模式的函数：调用 `nonstandard_arithmetic()` 会启用非标准模式（如果它受支持的话），而调用 `standard_arithmetic()` 会恢复 IEEE 行为。调用这些函数的 C 和 Fortran 语法如下所示：

---

|         |   |
|---------|---|
| C、C++   | <code>nonstandard_arithmetic();</code><br><code>standard_arithmetic();</code>         |
| Fortran | <code>call nonstandard_arithmetic()</code><br><code>call standard_arithmetic()</code> |

---

**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格外小心。要查看渐进下溢的更多信息，请参见第 2 章。

---

## 非标准算法和内核模拟

在实现非标准模式的 SPARC 浮点单元上，启用该模式会导致硬件将次正规操作数视为零并将次正规结果刷新为零。但是，用于模拟捕获的浮点指令的内核软件不实现非标准模式，其部分原因在于该模式的影响无法定义且依赖于实现，而且，与在软件中模拟浮点运算所带来的成本相比，处理渐进下溢所带来的附加成本可忽略不计。

如果将受到非标准模式影响的浮点运算被中断（例如，当出现上下文切换或者另一个浮点指令导致捕获时，已发出的运算将无法完成），它将由内核软件使用标准 IEEE 算法进行模拟。因此，在异常情况下，在非标准模式下运行的程序 *可能* 根据系统负荷生成稍有不同的结果。在实际中未发现这样的行为。它将只影响那些对以下情况非常敏感的程序：用渐进下溢还是用突然下溢执行极为罕见的特定运算。

---

## fpversion(1) 函数 — 查找有关 FPU 的信息

用编译器分发的 `fpversion` 公用程序标识已安装的 CPU 并估计处理器和系统总线的时钟速度。`fpversion` 通过解释由 CPU 和 FPU 存储的标识信息来确定 CPU 和 FPU 的类型。它循环地执行那些运行时间可以被预计的简单指令，并记录执行循环所占用的时间，从而估计时钟的速度。对该循环执行多次可增加计时测量的准确性。因此，`fpversion` 不是瞬间完成的；它可能需要运行几秒钟。

`fpversion` 还报告要用于主机系统的最佳 `-xtarget` 代码生成选项。

在 Ultra 4 工作站上，`fpversion` 显示类似如下的信息。（这些信息可能因计时或机器配置的不同而异。）

```
demo% fpversion
A SPARC-based CPU is available.
CPU's clock rate appears to be approximately 461.1 MHz.
Kernel says CPU's clock rate is 480.0 MHz.
Kernel says main memory's clock rate is 120.0 MHz.

Sun-4 floating-point controller version 0 found.
An UltraSPARC chip is available.
FPU's frequency appears to be approximately 492.7 MHz.

Use "-xtarget=ultra2 -xcache=16/32/1:2048/64/1" code-
generation option.

Hostid = hardware_host_id
```

要查看更多信息，请参见 `fpversion(1)` 手册页。

# x86 行为和实现

---

本附录讨论与 x86 平台中所用浮点单元有关的 x86 和 SPARC 兼容性问题。

硬件包括 x86 的 80386、80486 和 Pentium™ 微处理器以及其他制造商的兼容微处理器。虽然我们已经在使 x86 与 SPARC 平台保持兼容方面付出很大努力，但是二者仍存在几点区别。

在 x86 上：

- 浮点寄存器的宽度为 80 位。因为算术计算的中间结果可采用扩展精度，所以计算结果可能有所不同。-fstore 标志可尽量减小这些差异。但是，使用 -fstore 标志会导致性能下降。
- 每次加载或存储单精度或双精度浮点数时，都会出现与双扩展精度间的转换。因此，加载和存储浮点数可能导致异常。
- 渐进下溢完全是在硬件中实现的。不存在非标准模式。
- 不提供 fpversion 公用程序。
- 扩展双精度格式接受某些不代表任何浮点值的位模式（参见表 2-8）。硬件通常像对待 NaN 那样处理这些“不受支持的格式”，但是数学库在处理这样的表示形式时存在不一致。由于这些位模式决不会由硬件生成，所以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），或者产生于内存中的数据从一种类型到另一种类型的显式强制转换（例如，通过构造 C 语言中 union 数据结构）。因此，在大多数数值程序中，不会出现这些位模式。



# What Every Computer Scientist Should Know About Floating-Point Arithmetic

---

注意 - 本附录是对论文《What Every Computer Scientist Should Know About Floating-Point Arithmetic》（作者：David Goldberg，发表于 1991 年 3 月号的 *Computing Surveys*）进行编辑之后的重印版本。版权所有 1991，Association for Computing Machinery, Inc.，经许可重印。

---

---

## 摘要

许多人认为浮点运算是一个深奥的主题。这相当令人吃惊，因为浮点在计算机系统中是普遍存在的。几乎每种语言都有浮点数据类型；从 PC 到巨型计算机都有浮点加速器；多数编译器要经常用来编译浮点算法；而且实际上，每种操作系统都必须对浮点异常（如溢出）作出响应。本文将为您提供一个教程，涉及的方面包含对计算机系统设计人员产生直接影响的浮点运算信息。它首先介绍有关浮点表示和舍入误差的背景知识，然后讨论 IEEE 浮点标准，最后列举了许多示例来说明计算机生成器如何更好地支持浮点。

种类和主题描述符：（主要）C.0 [ 计算机系统组织 ]：概论 — *指令集设计*；D.3.4 [ 程序设计语言 ]：处理器 — *编译器，优化*；G.1.0 [ 数值分析 ]：概论 — *计算机运算，错误分析，数值算法*（次要）

D.2.1 [ 软件工程 ]：要求 / 规范 — *语言*；D.3.4 [ 程序设计语言 ]：正式定义和理论 — *语义*；D.4.1 [ 操作系统 ]：进程管理 — *同步*。

一般术语：算法，设计，语言

其它关键字 / 词：非正规化数值，异常，浮点，浮点标准，渐进下溢，保护数位，NaN，上溢，相对误差、舍入误差，舍入模式，ulp，下溢。

---

## 简介

计算机系统的生成器经常需要有关浮点运算的信息。但是，有关这方面的详细信息来源非常少。有关此主题的书目非常少，而且其中的一部《浮点运算》（作者：Pat Sterbenz）现已绝版。本文提供的教程包含与系统构建直接相关的浮点运算（以下简称为浮点）信息。它由三部分组成（这三部分并不完全相关）。第一部分“舍入误差”（在第 174 页上）讨论对加、减、乘、除基本运算使用不同舍入策略的含义。它还包含有关衡量舍入误差的两种方法（ulp 和相对误差）的背景信息。第二部分讨论 IEEE 浮点标准，该标准正被商业硬件制造商迅速接受。IEEE 标准中包括基本运算的舍入方法。对标准的讨论借助了“舍入误差”（在第 174 页上）部分中的内容。第三部分讨论浮点与计算机系统各个方面设计之间的关联。主题包括指令集设计、优化编译器和异常处理。

作者已尽力避免在未证明陈述为真的情况下来陈述浮点，这主要是因为证明将产生较为复杂的基本计算。对于那些不属于文章主旨的说明，已将其归纳到名为“详细信息”的章节中，您可以视实际情况选择跳过此部分。另外，此部分还包含了许多定理的证明。每个证明的结尾处均标记有符号 ■。如果未提供证明，■将紧跟在定理陈述之后。

---

## 舍入误差

将无穷多位的实数缩略表示为有限位数需要使用近似。即使存在无穷多位整数，多数程序也可将整数计算的结果以 32 位进行存储。相反，如果指定一个任意的固定位数，那么多数实数计算将无法以此指定位数精准表示实际数量。因此，通常必须对浮点计算的结果进行舍入，以便与其有限表示相符。舍入误差是浮点计算所独有的特性。“相对误差和 Ulp”（在第 176 页上）章节说明如何衡量舍入误差。

既然多数浮点计算都具有舍入误差，那么如果基本算术运算产生的舍入误差稍大于实际需要，这有没有关系？该问题是贯穿本章节的核心主题。“保护数位”（在第 177 页上）章节讨论保护数位，它是一种减少两个相近的数相减时所产生的误差的方法。IBM 认为保护数位非常重要，因此在 1968 年它将保护数位增加到 System/360 架构的双精度格式（其时单精度已具有保护数位），并更新了该领域中所有的现有计算机。下面有两个示例说明了保护数位的效用。

IEEE 标准不仅仅要求使用保护数位。它提供了用于加、减、乘、除和平方根的算法，并要求实现产生与该算法相同的结果。因此，将程序从一台计算机移至另一台计算机时，如果这两台计算机均支持 IEEE 标准，那么基本运算的结果逐位相同。这大大简化了程序的移植过程。在“精确舍入的运算”（在第 182 页上）中介绍了此精确规范的其它用途。

## 浮点格式

已经提议了几种不同的实数表示法，但是到目前为止使用最广的是浮点表示法。<sup>1</sup> 浮点表示法有一个基数  $\beta$ （始终假定其为偶数）和一个精度  $p$ 。如果  $\beta = 10$ 、 $p = 3$ ，则将数 0.1 表示为  $1.00 \times 10^{-1}$ 。如果  $\beta = 2$ 、 $p = 24$ ，则无法精确表示十进制数 0.1，但是它近似为  $1.10011001100110011001101 \times 2^{-4}$ 。

通常，将浮点数表示为  $\pm d.dd\dots d \times \beta^e$ ，其中  $d.dd\dots d$  称为有效数字<sup>2</sup>，它具有  $p$  个数位。更精确地说， $\pm d_0.d_1d_2\dots d_{p-1} \times \beta^e$  表示以下数

$$\pm \left( d_0 + d_1\beta^{-1} + \dots + d_{p-1}\beta^{-(p-1)} \right) \beta^e, (0 \leq d_i < \beta) \quad (1)$$

术语浮点数用于表示一个实数，该实数可以用正在讨论中的格式精确表示。与浮点表示法相关联的其它两个参数是最大允许指数和最小允许指数，即  $e_{\max}$  和  $e_{\min}$ 。由于存在  $\beta^p$  个可能的有效数字，以及  $e_{\max} - e_{\min} + 1$  个可能的指数，因此浮点数可以按

$$[\log_2(e_{\max} - e_{\min} + 1)] + [\log_2(\beta^p)] + 1$$

位编码，其中最后的 +1 用于符号位。此时，精确编码并不重要。

有两个原因导致实数不能精确表示为浮点数。最常见的情况可以用十进制数 0.1 说明。虽然它具有有限的十进制表示，但是在二进制中它具有无限重复的表示。因此，当  $\beta = 2$  时，数 0.1 介于两个浮点数之间，而这两个浮点数都不能准确地表示它。一种较不常见的情况是实数超出范围；也就是说，其绝对值大于  $\beta \times \beta^{e_{\max}}$  或小于  $1.0 \times \beta^{e_{\min}}$ 。本文的大部分内容讨论第一种原因导致的问题。然而，超出范围的数将在“无穷大”（在第 192 页上）和“非正规化数”（在第 193 页上）章节中进行讨论。

浮点表示不一定是唯一的。例如， $0.01 \times 10^1$  和  $1.00 \times 10^{-1}$  都表示 0.1。如果前导位非零（在上面的等式 (1) 中， $d_0 \neq 0$ ），那么该表示称为正规化。浮点数  $1.00 \times 10^{-1}$  是正规化的，而  $0.01 \times 10^1$  则不是。当  $\beta = 2$ 、 $p = 3$ 、 $e_{\min} = -1$  且  $e_{\max} = 2$  时，有 16 个正规化浮点数，如图 D-1 所示。粗体散列标记对应于其有效数字是 1.00 的数。要求浮点表示为正规化，则可以使该表示唯一。遗憾的是，此限制将无法表示零！表示 0 的一种自然方法是使用  $1.0 \times \beta^{e_{\min}-1}$ ，因为这样做保留了以下事实：非负实数的数值顺序对应于其浮点表示的词汇顺序。<sup>3</sup> 将指数存储在  $k$  位字段中时，意味着只有  $2^k - 1$  个值可用作指数，因为必须保留一个值来表示 0。

请注意，浮点数中的  $\times$  是表示法的一部分，这与浮点乘法运算不同。 $\times$  符号的含义通过上下文来看应该是明确的。例如，表达式  $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$  仅产生单个浮点乘法。

---

1. 其它表示法的示例有浮动斜线和有符号对数 [Matula 和 Kornerup 1985；Swartzlander 和 Alexopoulos 1975]。

2. 此术语由 Forsythe 和 Moler [1967] 提出，现已普遍替代了旧术语尾数。

3. 这假定采用通常的排列方式，将指数存储在有效数字的左侧。

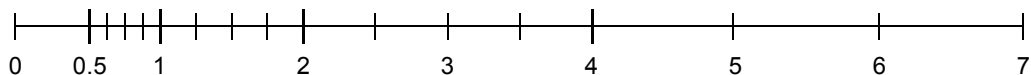


图 D-1  $\beta = 2$ 、 $p = 3$ 、 $e_{\min} = -1$ 、 $e_{\max} = 2$  时正规化的数

## 相对误差和 Ulp

因为舍入误差是浮点计算的固有特性，所以需要有一种方法来衡量此误差，这一点很重要。考虑使用  $\beta = 10$  且  $p = 3$  的浮点格式（本节中各处均使用此格式）。如果浮点计算的结果是  $3.12 \times 10^{-2}$ ，并且以无限精度计算的结果是 .0314，那么您可以清楚地看到最后一位存在 2 单位的误差。同样，如果将实数 .0314159 表示为  $3.14 \times 10^{-2}$ ，那么将在最后一位存在 .159 单位的误差。通常，如果浮点数  $d.d\dots d \times \beta^e$  用于表示  $z$ ，那么将在最后一位存在  $|d.d\dots d - (z/\beta^e)|\beta^{p-1}$  单位的误差。<sup>1,2</sup> 术语 *ulp* 将用作“最后一位上的单位数”的简写。如果某个计算的结果是最接近于正确结果的浮点数，那么它仍然存在 .5 ulp 的误差。衡量浮点数与它所近似的实数之间差值的另一种方法是 *相对误差*，它只是用两数之差除以实数的商。例如，当  $3.14159$  近似为  $3.14 \times 10^0$  时产生的相对误差是  $.00159/3.14159 \approx .0005$ 。

要计算对应于 .5 ulp 的相对误差，请注意当实数用可能最接近的浮点数  $d.dd\dots dd \times \beta^e$  近似表示时，误差可达到  $0.00\dots 00\beta^p \times \beta^e$ ，其中  $\beta'$  是数字  $\beta/2$ ，在浮点数的有效数字中有  $p$  单位，在误差的有效数字中有  $p$  单位个 0。此误差是  $((\beta/2)\beta^{-p}) \times \beta^e$ 。因为形式为  $d.dd\dots dd \times \beta^e$  的数都具有相同的绝对误差，但具有介于  $\beta^e$  和  $\beta \times \beta^e$  之间的值，所以相对误差介于  $((\beta/2)\beta^{-p}) \times \beta^e/\beta^e$  和  $((\beta/2)\beta^{-p}) \times \beta^e/\beta^{e+1}$  之间。即，

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2}\text{ulp} \leq \frac{\beta}{2}\beta^{-p} \quad (2)$$

特别是，对应于 .5 ulp 的相对误差可能随因子  $\beta$  的不同而有所变化。此因子称为 *浮动系数*。将  $\varepsilon = (\beta/2)\beta^{-p}$  设置为上面 (2) 中的上限，表明：将一个实数舍入为最接近的浮点数时，相对误差总是以  $\varepsilon$  为界限， $\varepsilon$  称为 *机器  $\varepsilon$* 。

在上例中，相对误差是  $.00159/3.14159 \approx .0005$ 。为了避免此类较小数，通常将相对误差表示为一个因子乘以  $\varepsilon$  的形式，在此例中  $\varepsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = .005$ 。因此，相对误差将表示为  $(.00159/3.14159)/.005) \varepsilon \approx 0.1\varepsilon$ 。

以实数  $x = 12.35$  为例说明 ulp 与相对误差之间的差异。将其近似为  $\tilde{x} = 1.24 \times 10^1$ 。误差是 0.5 ulp，相对误差是  $0.8\varepsilon$ 。接下来，将考虑  $8x$  的计算。精确值是  $8x = 98.8$ ，而计算值是  $8\tilde{x} = 9.92 \times 10^1$ 。此时误差是 4.0 ulp，而相对误差仍然是  $0.8\varepsilon$ 。尽管相对误差保持不变，但以 ulp 衡量的误差却是原来的 8 倍。通常，当基数为  $\beta$  时，以 ulp 表示的固定相对误差最大可按因子  $\beta$  进行浮动。反之，如上述公式 (2) 所示，固定误差 .5 ulp 导致相对误差可按  $\beta$  进行浮动。

1. 除非数  $z$  大于  $\beta^{e_{\max}+1}$  或小于  $\beta^{e_{\min}}$ 。在另行通知之前，将不会考虑以此方式超出范围的数。

2. 假设  $z'$  是近似  $z$  的浮点数。那么， $|d.d\dots d - (z/\beta^e)|\beta^{p-1}$  将等价于  $|z'-z|/\text{ulp}(z')$ 。衡量误差的更精确公式是  $|z'-z|/\text{ulp}(z)$ 。—编者



衡量舍入误差的最常用方法是以 ulp 表示。例如，舍入为最接近的浮点数相当于一个小于或等于 .5 ulp 的误差。但是，在分析由各种公式导致的舍入误差时，使用相对误差是一种较好的方法。在“证明”（在第 209 页上）章节中的分析很好地说明了这一点。由于浮动因子  $\beta$  的关系， $\varepsilon$  可能会过高估计舍入为最接近浮点数的效果，所以在使用较小  $\beta$  的计算机上，对公式的误差估计更为精确。

在仅关心舍入误差的数量级位数时，ulp 和  $\varepsilon$  可以互换使用，因为它们只是在因子  $\beta$  上有差异。例如，在浮点数的误差为  $n$  ulp 时，这意味着受影响的位数是  $\log_{\beta} n$ 。如果某个计算的相对误差是  $n\varepsilon$ ，那么

$$\text{受影响的位数} \approx \log_{\beta} n. \quad (3)$$

## 保护数位

计算两个浮点数之差的一种方法是：精确计算二者之差，然后将其舍入为最接近的浮点数。如果两个操作数的大小差别非常大，那么系统开销也将随之上升。假定  $p = 3$ ， $2.15 \times 10^{12} - 1.25 \times 10^{-5}$  将计算为

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= .0000000000000000125 \times 10^{12} \\ x - y &= 2.149999999999999875 \times 10^{12} \end{aligned}$$

它舍入为  $2.15 \times 10^{12}$ 。浮点硬件通常按固定位数执行运算，而不是使用所有这些数位。假定保留位数是  $p$ ，并且在右移较小的操作数时，只是舍弃数字（与舍入相对）。那么， $2.15 \times 10^{12} - 1.25 \times 10^{-5}$  将变为

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= 0.00 \times 10^{12} \\ x - y &= 2.15 \times 10^{12} \end{aligned}$$

结果是完全相同的，就好像先对差值进行精确计算，然后再进行舍入。请看另一个示例： $10.1 - 9.93$ 。它将变成

$$\begin{aligned} x &= 1.01 \times 10^1 \\ y &= 0.99 \times 10^1 \\ x - y &= .02 \times 10^1 \end{aligned}$$

正确结果是 .17，因此计算差值的误差是 30 ulp，而且每一位数字均不正确：误差可以达到多大程度？

### 定理 1

*使用带有参数  $\beta$  和  $p$  的浮点格式，并使用  $p$  位数字计算差值，结果的相对误差可能与  $\beta - 1$  一样大。*

## 证明

当  $x = 1.00\dots 0$  和  $y = .\rho\rho\dots\rho$  时, 表达式  $x - y$  中的相对误差为  $\beta - 1$ , 其中  $\rho = \beta - 1$ 。此处,  $y$  具有  $p$  位数字 (均等于  $\rho$ )。精确差值是  $x - y = \beta^{-p}$ 。然而, 如果仅使用  $p$  位数字计算结果, 那么位于  $y$  最右侧的数字将被舍弃, 因此计算差值是  $\beta^{-p+1}$ 。因此, 误差是  $\beta^{-p} - \beta^{-p+1} = \beta^{-p}(\beta - 1)$ , 相对误差是  $\beta^{-p}(\beta - 1)/\beta^{-p} = \beta - 1$ 。■

当  $\beta=2$  时, 相对误差可能与结果一样大; 当  $\beta=10$  时, 相对误差可能是结果的 9 倍。换言之, 当  $\beta=2$  时, 公式 (3) 显示受影响的位数是  $\log_2(1/\epsilon) = \log_2(2^p) = p$ 。即, 结果中所有  $p$  位数字都是错误的! 假定再增加一个数位 (保护数位) 以避免发生这种情况。即, 将较小数截断为  $p + 1$  位数字, 然后将相减的结果舍入为  $p$  位数字。使用保护数位后, 上例将变为

$$\begin{aligned}x &= 1.010 \times 10^1 \\y &= 0.993 \times 10^1 \\x - y &= .017 \times 10^1\end{aligned}$$

并且结果是精确的。使用单个保护数位时, 结果的相对误差可能大于  $\epsilon$  (例如  $110 - 8.59$ )。

$$\begin{aligned}x &= 1.10 \times 10^2 \\y &= .085 \times 10^2 \\x - y &= 1.015 \times 10^2\end{aligned}$$

它舍入为 102, 与正确结果 101.41 相比, 相对误差为 .006, 大于  $\epsilon = .005$ 。通常, 结果的相对误差只能比  $\epsilon$  稍大。更精确地说,

## 定理 2

如果  $x$  和  $y$  均是使用参数  $\beta$  和  $p$  的形式表示的浮点数, 并且使用  $p + 1$  位数字 (即, 有一个保护数位) 执行减法操作, 那么结果中的相对舍入误差将小于  $2\epsilon$ 。

此定理将在“舍入误差”(在第 209 页上)中证明。在上述定理中包括了加法运算, 因为  $x$  和  $y$  可以是正数或负数。

## 抵消

可以这样总结上一节: 在不使用保护数位的情况下, 如果在两个相近数之间执行减法, 那么产生的相对误差会非常大。换言之, 对包含减法运算 (或在具有相反符号的数量之间进行加法运算) 的任何表达式求值均可能产生较大的相对误差, 以至所有数位都将变为无意义 (定理 1)。如果在两个相近数之间执行减法, 那么操作数中的最高有效数位将因为相等而彼此抵消。有以下两种抵消: 恶性抵消和良性抵消。

恶性抵消发生在操作数受舍入误差的制约时。例如，在二次公式中出现了表达式  $b^2 - 4ac$ 。数量  $b^2$  和  $4ac$  受舍入误差的制约，因为它们是浮点乘法的结果。假定将它们舍入为最近的浮点数，因此它们的准确性使误差保持在  $.5 \text{ ulp}$  之内。它们相减时，抵消可能会导致许多精确数位消失，而留下受舍入误差影响的数位。因此，差值的误差可能是许多个  $\text{ulp}$ 。以  $b = 3.34$ ,  $a = 1.22$ ,  $c = 2.28$  为例。 $b^2 - 4ac$  的精确值是  $.0292$ 。但是  $b^2$  舍入为  $11.2$ ,  $4ac$  舍入为  $11.1$ , 因此最终结果是  $.1$ , 误差是  $70 \text{ ulp}$  (即使  $11.2 - 11.1$  的精确结果等于  $.1$ )<sup>1</sup>。减法运算并未引入任何误差，而是显示出先前乘法运算中引入的误差。

良性抵消发生在精确已知数量之间执行减法运算时。如果  $x$  和  $y$  没有舍入误差，依据定理 2 可知，若使用保护数位执行减法运算，那么差值  $x-y$  的相对误差会非常小 (小于  $2\epsilon$ )。

有时，表现出恶性抵消的公式可以通过重新整理的方式来消除此问题。我们还是以二次方程求根公式为例

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad (4)$$

当  $b^2 \gg ac$  时，则  $b^2 - 4ac$  不会产生抵消，而且

$$\sqrt{b^2 - 4ac} \approx |b|$$

但是公式之一内的另一加法运算 (减法运算) 将具有恶性抵消。要避免此类情况发生，请将  $r_1$  的分子和分母乘以

$$-b - \sqrt{b^2 - 4ac}$$

对于  $r_2$  进行类似处理，得到

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \quad (5)$$

如果  $b^2 \gg ac$  且  $b > 0$ ，则使用公式 (4) 计算  $r_1$  将产生抵消。因此，使用公式 (5) 计算  $r_1$ ，使用 (4) 计算  $r_2$ 。另一方面，如果  $b < 0$ ，则使用 (4) 计算  $r_1$ ，使用 (5) 计算  $r_2$ 。

表达式  $x^2 - y^2$  是表现出恶性抵消的另一公式。将它计算为  $(x-y)(x+y)$  更为精确。<sup>2</sup> 与二次方程式不同，这一改进的形式仍然包含减法运算，但是它是数量的良性抵消 (没有舍入误差)，而不是恶性抵消。依据定理 2， $x-y$  中的相对误差最大为  $2\epsilon$ 。对于  $x+y$ ，也同样正确。将两个具有较小相对误差结果的数量相乘，会产生具有较小相对误差的乘积。请参见“舍入误差”(在第 209 页上) 章节。

1. 700，而不是 70。因为  $.1 - .0292 = .0708$ ，以  $\text{ulp}(0.0292)$  表示的误差是  $708 \text{ ulp}$ 。—编者

2. 虽然表达式  $(x-y)(x+y)$  不产生恶性抵消，但是，如果符合以下两个条件之一，其精确程度要比  $x^2 - y^2$  稍差： $x \gg y$  或  $x \ll y$ 。在这种情况下， $(x-y)(x+y)$  具有三个舍入误差，但是  $x^2 - y^2$  只有两个舍入误差，因为计算  $x^2$  和  $y^2$  中的较小者时产生的舍入误差不影响最终的减法运算。

为避免将精确值与计算值相混淆，特采用以下表示法。 $x \ominus y$  表示计算的差值（即具有舍入误差），而  $x - y$  表示  $x$  和  $y$  的精确差值。类似地， $\oplus$ 、 $\otimes$  和  $\oslash$  分别表示计算的加法、乘法和除法。全大写函数表示函数的计算值，如  $\text{LN}(x)$  或  $\text{SQRT}(x)$ 。小写函数和传统数学符号表示其精确值，如  $\ln(x)$  和  $\sqrt{x}$ 。

虽然  $(x \ominus y) \otimes (x \oplus y)$  与  $x^2 - y^2$  极为近似，但是浮点数  $x$  和  $y$  本身可能近似为某些真实数量： $\hat{x}$  和  $\hat{y}$ 。例如， $\hat{x}$  和  $\hat{y}$  可能是精确已知的十进制数，且无法用二进制精确表示。在这种情况下，即使  $x \ominus y$  与  $x - y$  极为近似，与真实表达式  $\hat{x} - \hat{y}$  相比，它也可能具有较大的相对误差，因此  $(x + y)(x - y)$  相对  $x^2 - y^2$  的优势就不那么明显了。因为计算  $(x + y)(x - y)$  与计算  $x^2 - y^2$  的工作量几乎相同，所以在这种情况下前者明显是首选的形式。然而在一般情况下，如果将恶性抵消替换为良性抵消会产生较大的开销，那么这种做法是不值得的，因为输入通常是（但不总是）一个近似值。但是，即使数据是不精确的，完全消除抵消（如在二次公式中）也是值得做的。本文始终假定算法的浮点输入是精确的，而且计算结果尽可能精确。

表达式  $x^2 - y^2$  在重新写为  $(x - y)(x + y)$  时更为精确，因为恶性抵消被良性抵消所替换。接下来恶性抵消公式示例更为有趣，可以将公式进行改写，以便仅表现出良性抵消。

三角形面积可以直接用其各边  $a$ 、 $b$  和  $c$  的长度表示，如下所示：

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad \text{其中 } s = (a+b+c)/2 \quad (6)$$

（假设三角形是非常平坦的；即， $a \approx b + c$ 。那么， $s \approx a$ ，并且公式 (6) 中的项  $(s - a)$  将减去两个相近数，其中之一可能具有舍入误差。例如，如果  $a = 9.0$  且  $b = c = 4.53$ ，则  $s$  的正确值是 9.03， $A$  是 2.342...。即使  $s$  的计算值 (9.05) 的误差仅为 2 ulp， $A$  的计算值 (3.04) 的误差也高达 70 ulp。

有一种方法可以改写公式 (6)，以便使其对于平坦的三角形也能返回精确的结果 [Kahan 1986]。即

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, \quad a \geq b \geq c \quad (7)$$

如果  $a$ 、 $b$  和  $c$  不满足  $a \geq b \geq c$ ，那么在应用 (7) 之前请将它们重命名。检查 (6) 和 (7) 右侧是否是代数恒等是一项非常简单的工作。使用上述  $a$ 、 $b$  和  $c$  的值得出的计算面积是 2.35，其误差是 1 ulp，比第一个公式要精确得多。

对于此示例来说，虽然公式 (7) 比 (6) 要精确得多，但是了解 (7) 在通常情况下的执行情况是非常必要的。

### 定理 3

如果减法运算是使用保护数位执行的 ( $e \leq .005$ )，且计算的平方根在 1/2 ulp 内，则使用 (7) 计算三角形面积时产生的舍入误差最大为  $11\epsilon$ 。

条件  $e < .005$  实际上在每个浮点系统中都能得到满足。例如，当  $\beta = 2$  且  $p \geq 8$  时，可确保  $e < .005$ ；当  $\beta = 10$  时， $p \geq 3$  已足够。

在命题（如定理 3）中讨论表达式的相对误差时，认为表达式是使用浮点运算计算的。特别是，相对误差实际上是由以下表达式产生的：

$$\text{SQRT}((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4 \quad (8)$$

由于 (8) 的繁琐性，在定理的陈述中我们通常采用  $E$  的*计算值*，而不是写出带有圆形符号的  $E$ 。

误差界通常是非常保守的。在上述数值示例中，(7) 的计算值是 2.35，与真实值 2.34216 相比，相对误差是  $0.7\varepsilon$ ，这比  $11\varepsilon$  要小得多。计算误差界的主要原因不是为了获得精确界限，而是为了检验公式是否不包含数值问题。

请看最后一个示例：可以改写为使用良性抵消的表达式  $(1+x)^n$ ，其中  $x \ll 1$ 。此表达式出现在金融计算中。以每天将 100 美元存入一个银行帐户为例，其年利率为 6%，每天按复利计算。如果  $n = 365$  且  $i = .06$ ，那么在年底时累积的金额为

$$100 \frac{(1 + i/n)^n - 1}{i/n}$$

美元。如果这是使用  $\beta = 2$  和  $p = 24$  进行计算的，那么结果将是 37615.45 美元，与精确结果 37614.05 美元相比，差额为 1.40 美元。导致此问题的原因很明显。表达式  $1 + i/n$  将导致 1 和 .0001643836 相加，这样会丢失  $i/n$  的低位。将  $1 + i/n$  扩大到  $n$  次幂时，将扩大此舍入误差。

这个令人头痛的表达式  $(1 + i/n)^n$  可以改写为  $e^{n \ln(1 + i/n)}$ ，现在的问题是在  $x$  较小的情况下计算  $\ln(1 + x)$ 。一种方法是使用近似  $\ln(1 + x) \approx x$ ，在这种情况下应支付 37617.26 美元，差额为 3.21 美元，甚至还不如使用这个直观的公式精确。但是，有一种方法可以非常精确地计算  $\ln(1 + x)$ ，如定理 4 所示 [Hewlett-Packard 1982]。使用此公式的计算结果是 37614.07 美元，其精确性使误差保持在两美分之内！

定理 4 假设  $\text{LN}(x)$  近似于  $\ln(x)$ ，其误差在  $1/2 \text{ ulp}$  内。它解决的问题是：当  $x$  很小时， $\text{LN}(1 \oplus x)$  不接近于  $\ln(1 + x)$ ，因为  $1 \oplus x$  在  $x$  的低位中丢失了信息。也就是说，当  $x \ll 1$  时， $\ln(1 + x)$  的计算值不接近于其实际值。

## 定理 4

计算  $\ln(1 + x)$  时使用如下公式：

$$\ln(1 + x) = \begin{cases} x & \text{for } 1 \oplus x = 1 \\ \frac{x \ln(1 + x)}{(1 + x) - 1} & \text{for } 1 \oplus x \neq 1 \end{cases}$$

如果  $0 \leq x < 3/4$ ，减法运算使用保护数位执行 ( $e < 0.1$ )，且  $\ln$  的计算值误差在  $1/2 \text{ ulp}$  之内，则相对误差最大为  $5\varepsilon$ 。

此公式适用于  $x$  的任意值，但仅在  $x \ll 1$  时应引起注意，这是朴素公式  $\ln(1+x)$  中出现恶性抵消的地方。虽然此公式看起来可能有些神秘，但是一个简单的解释就可以说明它的工作原理。将  $\ln(1+x)$  写为

$$x \left( \frac{\ln(1+x)}{x} \right) = x\mu(x)$$

可以精确计算左侧因子，但是在将 1 和  $x$  相加时右侧因子  $\mu(x) = \ln(1+x)/x$  将产生较大的舍入误差。然而， $\mu$  几乎是恒定的，因为  $\ln(1+x) \approx x$ 。因此，稍微改变  $x$  不会引入很大误差。换句话说，如果  $\tilde{x} \approx x$ ，计算  $x\mu(\tilde{x})$  将与  $x\mu(x) = \ln(1+x)$  极为近似。是否存在一个  $\tilde{x}$  的值，使用它可以精确计算  $\tilde{x}$  和  $\tilde{x}+1$ ？是的；该值是  $\tilde{x} = (1 \oplus x) \ominus 1$ ，因为使用该值后  $1 + \tilde{x}$  与  $1 \oplus x$  完全相等。

可以这样总结本节：保护数位在彼此接近且精确已知的数量相减（良性抵消）时保证了精确性。有时，可以通过使用良性抵消来改写产生不精确结果的公式，以便获得相当高的数值精确性；但是，仅当使用保护数位进行减法运算时此过程才起作用。使用保护数位的开销并不高，因为它仅仅要求将加法器加宽一位。对于 54 位双精度加法器，增加的开销少于 2%。只需付出如此代价，您就能够运行许多算法（如公式 (6)）来计算三角形的面积和表达式  $\ln(1+x)$ 。虽然多数现代计算机具有保护数位，但是也有一些计算机（如 Cray 系统）没有保护数位。

## 精确舍入的运算

在使用保护数位进行浮点运算时，它们不如先精确计算再舍入为最接近的浮点数那样精确。以这种方式进行的运算将称为*精确舍入的运算*。<sup>1</sup> 紧邻定理 2 之前的示例显示出单个保护数位并不总是给出精确舍入的结果。上一节给出了需要有保护数位才能正常工作的几个算法示例。本节给出要求精确舍入的算法示例。

到目前为止，尚未给出任何舍入的定义。舍入是很好理解的，只是如何舍入中间数有些令人棘手；例如，应该将 12.5 舍入为 12 还是 13？一个学派将 10 个数字分成两半，使 {0, 1, 2, 3, 4} 向下舍入，使 {5, 6, 7, 8, 9} 向上舍入；因此 12.5 将舍入为 13。这是在 Digital Equipment Corporation VAX 计算机上的舍入方式。另一学派则认为：由于以 5 结尾的数介于两个可能的舍入数中间，向下舍入与向上舍入需视情况而定。实现这一发生概率为 50% 事件的方法之一是要求舍入结果的最低有效数位为偶数。因此 12.5 舍入为 12 而不是 13，因为 2 是一个偶数。其中的哪种方法是最佳的，是向上舍入还是舍入为偶数？Reiser 和 Knuth [1975] 倾向于舍入为偶数，并提供了以下理由。

---

1. 通常也称为*正确舍入的运算*。—编者

## 定理 5

假设  $x$  和  $y$  是浮点数, 并定义  $x_0 = x$ ,  $x_1 = (x_0 \ominus y) \oplus y$ , ...,  $x_n = (x_{n-1} \ominus y) \oplus y$ 。如果使用舍入为偶数精确舍入  $\oplus$  和  $\ominus$ , 那么对于所有  $n$ ,  $x_n = x$ ; 对于所有  $n \geq 1$ ,  $x_n = x_1$ 。■

为了阐明此结果, 请以  $\beta = 10$ ,  $p = 3$  为例, 并令  $x = 1.00$ ,  $y = -.555$ 。当进行向上舍入时, 该序列将变成

$$x_0 \ominus y = 1.56, x_1 = 1.56 \ominus .555 = 1.01, x_1 \oplus y = 1.01 \oplus .555 = 1.57,$$

$x_n$  的每个连续值都将增加 .01, 直至  $x_n = 9.45$  ( $n \leq 845$ )<sup>1</sup> 为止。在使用舍入为偶数的情况下,  $x_n$  始终是 1.00。此示例说明: 在使用向上舍入规则时, 计算结果可能会逐渐递增, 而依据此定理, 这种情况在使用舍入为偶数时不会发生。本文的其余部分将使用舍入为偶数。

精确舍入的应用之一是执行多种精度运算。有两种基本方法可以获得较高的精度。一种方法使用极大的有效数字表示浮点数, 将有效数字存储在字节数组中, 并用汇编语言编写用于处理这些数的例程的代码。第二种方法将更高精度的浮点数表示为普通浮点数的数组, 在其中增加无限精度的数组元素以便恢复高精度浮点数。此处将讨论第二种方法。使用浮点数的数组的优点是: 可以使用高级语言编写其可移植代码, 但是它要求精确舍入的运算。

在此系统中乘法的关键是将积  $xy$  表示为和, 其中每个被加数都具有与  $x$  和  $y$  相同的精度。通过拆分  $x$  和  $y$ , 可以做到这一点。编写  $x = x_h + x_l$  和  $y = y_h + y_l$ , 则精确乘积为

$$xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l$$

如果  $x$  和  $y$  具有  $p$  位有效数字, 那么被加数也将具有  $p$  位有效数字, 但条件是  $x_l$ 、 $x_h$ 、 $y_h$ 、 $y_l$  可以使用  $\lceil p/2 \rceil$  位来表示。当  $p$  是偶数时, 很容易找到一种拆分方法。数  $x_0 x_1 \dots x_{p-1}$  可以写为  $x_0 x_1 \dots x_{p/2-1}$  和  $0.0 \dots 0 x_{p/2} \dots x_{p-1}$  之和。当  $p$  是奇数时, 这一简单的拆分方法将不起作用。但是, 通过使用负数可以获得额外位。例如, 如果  $\beta = 2$ ,  $p = 5$  且  $x = .10111$ , 那么  $x$  可以拆分为  $x_h = .11$  和  $x_l = -.00001$ 。拆分数的方法不止一种。一种易于计算的拆分方法是由 Dekker [1971] 提出的, 但它需要多个保护数位。

## 定理 6

假设  $p$  是浮点精度, 要求在  $\beta > 2$  时  $p$  是偶数, 并假设浮点运算是精确舍入的。那么, 如果  $k = \lceil p/2 \rceil$  是精度的一半 (向上舍入) 且  $m = \beta^k + 1$ , 则可以将  $x$  拆分为  $x = x_h + x_l$ , 其中

$$x_h = (m \otimes x) \ominus (m \otimes x \ominus x), x_l = x \ominus x_h,$$

---

1. 当  $n = 845$  时,  $x_n = 9.45$ ,  $x_n + 0.555 = 10.0$ ,  $10.0 - 0.555 = 9.45$ 。因此, 当  $n > 845$  时,  $x_n = x_{845}$ 。

每个  $x_i$  都可以使用  $[p/2]$  位精度表示。

为了在示例中说明此定理的应用，可令  $\beta = 10$ ,  $p = 4$ ,  $b = 3.476$ ,  $a = 3.463$ ,  $c = 3.479$ 。那么， $b^2 - ac$  在舍入为最接近的浮点数时是 .03480，而  $b \otimes b = 12.08$ ,  $a \otimes c = 12.05$ ，因此  $b^2 - ac$  的计算值是 .03。这样，误差就是 480 ulp。使用定理 6 编写  $b = 3.5 - .024$ ,  $a = 3.5 - .037$  和  $c = 3.5 - .021$ ,  $b^2$  将变成  $3.5^2 - 2 \times 3.5 \times .024 + .024^2$ 。每个被加数都是精确的，因此  $b^2 = 12.25 - .168 + .000576$ ，此时未计算其中的和。类似地， $ac = 3.5^2 - (3.5 \times .037 + 3.5 \times .021) + .037 \times .021 = 12.25 - .2030 + .000777$ 。最后，将这两个数列逐项相减将为  $0 \oplus .0350 \ominus .000201 = .03480$  的  $b^2 - ac$  得出估计值，它与精确舍入的结果完全相同。为了说明定理 6 确实需要精确舍入，请假设  $p = 3$ ,  $\beta = 2$ ,  $x = 7$ 。那么， $m = 5$ ,  $mx = 35$ ,  $m \otimes x = 32$ 。如果使用单个保护数位进行减法运算，那么  $(m \otimes x) \ominus x = 28$ 。因此， $x_h = 4$ ,  $x_l = 3$ ，由此得出  $x_l$  不能用  $[p/2] = 1$  位表示。

作为精确舍入的最后一个示例，假设用 10 除  $m$ 。结果是一个浮点数，一般情况下将不等于  $m/10$ 。当  $\beta = 2$  时，用 10 乘  $m/10$  将复原  $m$ ，条件是使用了精确舍入。实际上，有一个更一般的事实（由 Kahan 提出）成立。其证明非常巧妙，但对此细节不感兴趣的读者可以直接跳至“IEEE 标准”（在第 186 页上）章节。

## 定理 7

当  $\beta = 2$  时，如果  $m$  和  $n$  是整数，且  $|m| < 2^{p-1}$ ,  $n$  具有特殊形式  $n = 2^i + 2^j$ ，那么  $(m \oslash n) \otimes n = m$ ，条件是浮点运算是精确舍入的。

## 证明

换算为 2 的幂次方不会产生不良影响，因为这样仅改变指数，而不改变有效数字。如果  $q = m/n$ ，那么调整  $n$  的范围以便  $2^{p-1} \leq n < 2^p$ ，并调整  $m$  的范围以便  $1/2 < q < 1$ 。因此， $2^{p-2} < m < 2^p$ 。因为  $m$  具有  $p$  个有效位，所以在二进制点右侧它最多有一位。改变  $m$  的符号不会产生不良影响，因此假设  $q > 0$ 。

如果  $\bar{q} = m \oslash n$ ，那么要证明此定理需要得到

$$|n\bar{q} - m| \leq \frac{1}{4} \tag{9}$$

这是因为在二进制点右侧  $m$  最多具有 1 位，因此  $n\bar{q}$  将舍入为  $m$ 。为了处理  $|n\bar{q} - m| = 1/4$  时的中间情况，请注意，由于初始未调整范围的  $m$  具有  $|m| < 2^{p-1}$ ，其低位是 0，因此调整后的  $m$  的低位也是 0。因此，中间情况将舍入为  $m$ 。

假定  $q = .q_1q_2 \dots$ ，并令  $\hat{q} = .q_1q_2 \dots q_p1$ 。要估算  $|n\bar{q} - m|$ ，请首先计算

$$|\hat{q} - q| = |N/2^{p+1} - m/n|$$



其中  $N$  是一个奇数整数。因为  $n = 2^i + 2^j$  且  $2^{p-1} \leq n < 2^p$ , 所以对于某些  $k \leq p-2$  一定存在  $n = 2^{p-1} + 2^k$ , 因此

$$|\hat{q} - q| = \left| \frac{nN - 2^{p+1}m}{n2^{p+1}} \right| = \left| \frac{(2^{p-1} - k + 1)N - 2^{p+1} - km}{n2^{p+1} - k} \right|$$

分子是一个整数, 并且因为  $N$  是一个奇数, 所以实际上它是一个奇数整数。因此,

$$|\hat{q} - q| \geq 1/(n2^{p+1} - k)$$

假设  $q < \hat{q}$  ( $q > \hat{q}$  的情况是类似的)。<sup>1</sup> 那么,  $n\bar{q} < m$ , 且

$$\begin{aligned} |m - n\bar{q}| &= m - n\bar{q} = n(q - \bar{q}) = n(q - (\hat{q} - 2^{p-1})) \leq n \left( 2^{-p-1} - \frac{1}{n2^{p+1} - k} \right) \\ &= (2^{p-1} + 2^k)2^{-p-1} - 2^{-p-1+k} = \frac{1}{4} \end{aligned}$$

这将得到 (9) 并证明此定理。<sup>2</sup> ■

只要将  $2^i + 2^j$  替换为  $\beta^i + \beta^j$ , 此定理对于任何基数  $\beta$  都是成立的。但是, 随着  $\beta$  的增大,  $\beta^i + \beta^j$  形式的分母差值将越来越大。

我们现在可以回答以下问题: 如果基本算术运算引入比所需稍大的舍入误差, 这有没有关系? 答案是确实有关系, 因为通过精确的基本运算我们可以证明公式在具有较小相对误差的情况下是“正确的”。在此意义上, “抵消”(在第 178 页上) 章节讨论了几种需要使用保护数位以产生正确结果的算法。但是, 如果这些公式的输入是用不精确的方式表示的数, 那么定理 3 和定理 4 的限制就不需要太注意了。原因是: 如果  $x$  和  $y$  仅仅是某些衡量数量的近似, 那么良性抵消  $x - y$  可能变为恶性抵消。但是, 即使对于不精确的数据, 精确运算也是有用的, 因为通过精确运算我们可以建立精确关系(如定理 6 和定理 7 中所讨论的那些关系)。即使每个浮点变量只是某个实际值的近似, 精确运算也是有用的。

1. 请注意, 在二进制中,  $q$  不能等于  $\hat{q}$ 。—编者

2. 作为练习留给读者: 将证明扩展到基数不为 2 的情况。—编者

---

# IEEE 标准

浮点计算有两种不同的 IEEE 标准。IEEE 754 是一个二进制标准。对于单精度，它要求  $\beta = 2$ 、 $p = 24$ ；对于双精度，它要求  $p = 53$  [IEEE 1987]。它还指定单精度和双精度中位的精确布局。IEEE 854 允许  $\beta = 2$  或  $\beta = 10$ 。与 754 不同，它不指定如何将浮点数编码到位中 [Cody 等 1984]。它不需要  $p$  的特定值，而是为单精度和双精度指定对  $p$  的允许值的约束。在讨论这两个标准的公共属性时，将使用术语“IEEE 标准”。

本章节提供了 IEEE 标准的概述。每一子章节分别讨论了该标准的一个方面及包含此方面标准的原因。本文的目的不是论述 IEEE 标准是否是可能的最佳浮点标准，而是接受既定的标准，并介绍如何使用这些标准。有关完整的详细信息，请参阅标准本身 [IEEE 1987; Cody 等 1984]。

## 格式与运算

### 基数

IEEE 854 允许  $\beta = 10$  的原因是显而易见的。人们就是使用基数 10 来交换和考虑数字的。 $\beta = 10$  尤其适合于计算器，每个运算的结果都由计算器以十进制显示。

IEEE 854 要求：如果基数不是 10，就必须是 2。它这样要求有几个原因。“相对误差和 Ulp”（在第 176 页上）部分提到了其中的一个原因：当  $\beta$  是 2 时误差分析的结果更精确，因为进行相对误差计算时  $.5 \text{ ulp}$  的舍入误差按  $\beta$  的因子发生变化，而且基于相对误差的误差分析几乎始终是比较简单的。其中的原因必然与大基数的有效精度有关。以比较  $\beta = 16$ 、 $p = 1$  与  $\beta = 2$ 、 $p = 4$  为例。这两个系统的有效数字都是 4 位。判断  $15/8$  的计算结果。当  $\beta = 2$  时，15 被表示为  $1.111 \times 2^3$ ， $15/8$  被表示为  $1.111 \times 2^0$ 。因此  $15/8$  是精确的。但是，当  $\beta = 16$  时，15 被表示为  $F \times 16^0$ ，其中  $F$  是表示 15 的十六进制数字。但是  $15/8$  被表示为  $1 \times 16^0$ ，它只有一个正确位。通常，基数 16 最多可丢失 3 位，因此精度为  $p$  的十六进制数字具有的有效精度低至  $4p - 3$ ，而不是  $4p$  个二进制位。既然大的  $\beta$  值存在这些问题，为什么 IBM 为其 system/370 选择了  $\beta = 16$  呢？只有 IBM 才知道确切原因，但是这两个可能的原因。第一个是增大了指数范围。system/370 上的单精度具有  $\beta = 16$ 、 $p = 6$ 。因此有效数字需要 24 位。因为必须凑够 32 位，所以会为指数保留 7 位，为符号位保留 1 位。这样，可表示数值的大小范围是从  $16^{-26}$  到  $16^{26} = 2^{28}$ 。为了在  $\beta = 2$  时获得类似的指数范围，需要将 9 位用于指数部分，仅为有效数字部分保留 22 位。但是，上面刚刚指出：当  $\beta = 16$  时，有效精度可以低至  $4p - 3 = 21$  位。更糟的是，当  $\beta = 2$  时，有可能获得额外的精度位（正如本部分后面所说明的情况），因此  $\beta = 2$  的计算机具有 23 个精度位，而  $\beta = 16$  的计算机则具有 21 - 24 位的精度范围。

对选择  $\beta = 16$  的另一种可能解释与移位有关。将两个浮点数相加时，如果它们的指数不同，则其中的一个有效数字需要移位以使小数点对齐，这样就减慢了运算速度。在  $\beta = 16$ 、 $p = 1$  的系统上，介于 1 和 15 之间的所有数值都具有相同的指数，因此在增加此集合中不同数值的  $\binom{15}{2} = 105$  个可能对时，不需要进行移位。但是，在  $\beta = 2$ 、 $p = 4$  的系统中，这些数的指数范围是从 0 到 3，105 对中有 70 对需要进行移位。

对于现今大多数硬件来讲，通过避免操作数字集进行移位而获得的性能可以忽略，因此  $\beta = 2$  的微小波动使其成为更可取的基数。使用  $\beta = 2$  的另一个优点是有可能获得额外的有效位。<sup>1</sup> 因为浮点数始终是正规化的，所以有效数字的最高有效位始终是 1，这样就没有必要浪费一个表示它的存储位。使用此技巧的格式具有隐藏位。在“浮点格式”（在第 175 页上）中已经指出：这需要对 0 进行特殊约定。那里给出的方法是指数为  $e_{\min} - 1$ ，全零的有效数字不表示  $1.0 \times 2^{e_{\min} - 1}$ ，而是表示 0。

IEEE 754 单精度是用 32 位编码的，将 1 位用于符号位，8 位用于指数，23 位用于有效数字。但是，它使用一个隐藏位，因此有效数字是 24 位 ( $p = 24$ )，即使它仅使用 23 位进行编码也是如此。

## 精度

IEEE 标准定义了四种不同的精度：单精度、双精度、单精度扩展和双精度扩展。在 IEEE 754 中，单精度和双精度大致对应于大多数浮点硬件所提供的精度。单精度占用一个 32 位字，双精度占用两个连续的 32 位字。扩展精度是一种至少提供一点额外精度和指数范围的格式（表 D-1）。

表 D-1 IEEE 754 格式参数

| 参数         | 格式   |              |       |               |
|------------|------|--------------|-------|---------------|
|            | 单精度  | 单精度扩展        | 双精度   | 双精度扩展         |
| $p$        | 24   | $\geq 32$    | 53    | $\geq 64$     |
| $e_{\max}$ | +127 | $\geq 1023$  | +1023 | $> 16383$     |
| $e_{\min}$ | -126 | $\leq -1022$ | -1022 | $\leq -16382$ |
| 指数宽度（位）    | 8    | $\leq 11$    | 11    | $\geq 15$     |
| 格式宽度（位）    | 32   | $\geq 43$    | 64    | $\geq 79$     |

IEEE 标准仅指定扩展精度提供多少个额外位的下界。允许的最小双精度扩展格式有时称为 80 位格式，尽管上表显示它可以使用 79 位。原因是，扩展精度的硬件实现通常不使用隐藏位，因此将使用 80 位而不是 79 位。<sup>2</sup>

该标准将扩展精度作为重中之重，没有提出任何有关双精度的建议，但是强烈建议实现应该支持对应于所支持的最宽基本格式的扩展格式，...

1. 这似乎是首先由 Goldberg [1967] 发表的，尽管 Knuth ([1981]，第 211 页) 将此方法归功于 Konrad Zuse。
2. Kahan 认为，扩展精度具有 64 位的有效数字，因为这是可以在 Intel 8087 上不增加循环时间的情况下进行进位传送的最宽精度 [Kahan 1988]。

使用扩展精度的一个动因来自计算器。通常计算器将显示 10 位数字，但在内部使用 13 位。通过仅显示 13 位数字中的 10 位，计算器在用户看来就像一个按 10 位精度计算指数、余弦等的“黑箱”。如果计算器要在 10 位精度内计算诸如  $\exp$ 、 $\log$  和  $\cos$  之类的函数，且达到合理的效率，那么它需要使用几个额外位。不难找到这样一个简单的有理数表达式，它求对数近似值的误差为 500 ulp。这样，使用 13 位进行计算就可以得出正确的 10 位结果。通过将额外的 3 位隐藏，计算器为操作者提供了一个简单的模型。

IEEE 标准中的扩展精度起着类似的作用。它使库可以高效地计算数量，且单（或双）精度的误差在约 .5 ulp 内，为使用那些库的用户提供了一个简单模型，即：每个基本运算（不管是简单的乘法运算还是调用对数）都会返回精确度大约在 .5 ulp 内的值。但是，当使用扩展精度时，确保它的使用对用户透明是很重要的。例如，在计算器上，如果显示值的内部表示没有舍入到与显示值相同的精度，则进一步运算的结果将取决于隐藏位数，并且对用户来说似乎是不可预知的。

以 IEEE 754 单精度与十进制之间的转换问题为例来进一步说明扩展精度。理想情况下，将用足够的数位显示单精度数，以便在将十进制数读回时可以再现单精度数。结果是 9 个十进制位就足以再现单精度二进制数，请参见“二进制到十进制的转换”（在第 217 页上）部分。将十进制数再转换回它的唯一二进制表示时，小至 1 ulp 的舍入误差就是致命的，因为这将给出错误的结果。这是一种扩展精度对于高效算法至关重要的情况。在单精度扩展可用时，存在一种非常简单的方法，它可以将十进制数转换为单精度二进制数。首先将 9 个十进制数字作为整数  $N$  读入，忽略小数点。在表 D-1 中， $p \geq 32$ ，因为  $10^9 < 2^{32} \approx 4.3 \times 10^9$ ， $N$  可以用单精度扩展精确表示。接下来，找出换算  $N$  所需的适当幂  $10^p$ 。这将是十进制数的指数以及（到目前为止）被忽略小数点的位置的组合。计算  $10^{|p|}$ 。如果  $|p| \leq 13$ ，则这也是精确表示的，因为  $10^{13} = 2^{135}$  且  $5^{13} < 2^{32}$ 。最后，将  $N$  和  $10^{|p|}$  相乘（如果  $p < 0$ ，则将它们相除）。如果最后的这个运算是精确进行的，则将再现最接近的二进制数。“二进制到十进制的转换”（在第 217 页上）部分说明如何精确地进行最后的相乘（或相除）。因此，对于  $|p| \leq 13$ ，使用单精度扩展格式，可以将 9 位十进制数转换为最接近的二进制数（即精确舍入的数）。如果  $|p| > 13$ ，则使用单精度扩展不足以使上述算法始终计算出精确舍入的等价的二进制数，但是 Coonen [1984] 证明这足以保证先将二进制数转换为十进制数而后再转换回来会再现原始的二进制数。

如果支持双精度，则上述算法将以双精度而不是单精度扩展运行，但是将双精度转换为 17 位十进制数而后再转换回来将需要双精度扩展格式。

## 指数

因为指数可以是正数或负数，所以必须选择某个方法来表示其符号。表示有符号数的两种常用方法是符号 / 大小和 2 的补码。符号 / 大小是 IEEE 格式中用于有效数字符号的系统：一个位用于容纳符号，其余位用来表示数的大小。2 的补码表示通常在整数运算中使用。在此方案中，范围  $[-2^{p-1}, 2^{p-1} - 1]$  内的数由等于它以  $2^p$  为模的结果的最小非负数表示。

IEEE 二进制标准不使用上述的任何一种方法来表示指数，而是改用 *偏置* 的方法来表。在单精度数的情况下，指数以 8 位存储，偏差是 127（对于双精度，它是 1023）。这意味着，如果  $\bar{k}$  被解释为无符号整数的指数位的值，则浮点数的指数是  $\bar{k} - 127$ 。这通常称为 *无偏置指数*，以区别于偏置指数  $\bar{k}$ 。

从表 D-1 可以知道,单精度有  $e_{\max} = 127$ ,  $e_{\min} = -126$ 。其  $|e_{\min}| < e_{\max}$  的原因是最小数 ( $1/2^{e_{\min}}$ ) 的倒数将不上溢。虽然最大数的倒数确实会发生下溢,但是下溢通常不如上溢严重。“基数”(在第 186 页上)部分解释了  $e_{\min} - 1$  用于表示 0,“特殊数量”(在第 190 页上)将介绍  $e_{\max} + 1$  的一个用途。在 IEEE 单精度中,这意味着偏置指数介于  $e_{\min} - 1 = -127$  和  $e_{\max} + 1 = 128$  之间,而无偏置指数介于 0 和 255 之间(这正好是可以使用 8 位表示的非负数)。

## 运算

IEEE 标准要求精确舍入加、减、乘、除的结果。也就是说,必须先精确计算结果,然后舍入为最接近的浮点数(通过舍入为偶数)。“保护数位”(在第 177 页上)部分指出:当两个浮点数的指数有很大差异时,精确计算这两个浮点数的差或和的开销会非常大。该部分介绍了保护数位,它提供了一种在保证相对误差很小的同时计算差值的实用方法。但是,使用单个保护数位进行计算不会总是给出与先计算精确结果再进行舍入相同的结果。通过引入第二个保护数位和第三个粘滞位,就能够以稍高于单个保护数位的成本计算差值,但结果是相同的,就好像是先精确计算差值再进行舍入 [Goldberg 1990]。这样,就可以高效实施该标准。

完全指定算术运算结果的一个原因是为了改进软件的可移植性。当一个程序运行在两台计算机上且这两台计算机都支持 IEEE 运算时,如果任意的中间结果出现不同,则一定是由于软件错误,而不是运算差异。精确指定的另一优点是它使有关浮点的推理更容易进行。有关浮点的证明已足够困难,无须处理多种运算所产生的多种情况。正如整数程序的正确性是可以证明的,浮点程序的正确性也是可以证明的,不过在这种情况下要证明的是结果的舍入误差符合某些界限。定理 4 就是这样的一个证明示例。当精确指定被推理的运算时,就可以使这些证明容易得多。一旦某种算法经证明对于 IEEE 运算正确,那么它将在支持 IEEE 标准的任何计算机上正确工作。

Brown [1981] 曾建议了有关浮点的公理,这些公理包括大多数的现有浮点硬件。但是,在此系统中的证明无法检验“抵消”(在第 178 页上)和“精确舍入的运算”(在第 182 页上)部分中的算法,它们需要的功能并非存在于所有硬件上。此外,与简单地定义要被精确执行再进行舍入的运算相比,Brown 的公理更为复杂。因此,根据 Brown 的公理来证明定理通常比先假定运算是精确舍入的然后再证明它们要困难。

在浮点标准应该涉及哪些运算这一问题上,没有完全一致的意见。除了基本运算 +、-、 $\times$  和  $/$  之外,IEEE 标准还指定正确舍入平方根、余数以及整数和浮点数之间的转换。它还要求正确舍入内部格式与十进制之间的转换(非常大的数除外)。Kulisch 和 Miranker [1986] 曾建议将内积增加到精确指定的运算列表。他们注意到,在 IEEE 运算中计算内积时最终结果与正确结果之差可能相当大。例如,和是内积的一种特殊情况,和  $((2 \times 10^{-30} + 10^{30}) - 10^{30}) - 10^{-30}$  正好等于  $10^{-30}$ ,但是在使用 IEEE 运算的计算机上计算结果将是  $-10^{-30}$ 。使用比实现快速乘法器所用更少的硬件计算误差在 1 ulp 内的内积,这是可能的 [Kirchner 和 Kulish 1987]。<sup>1 2</sup>

---

1. 一些反对将内积作为一种基本运算包括在内的论点由 Kahan 和 LeBlanc [1985] 提出。

2. Kirchner 写到:在每时钟周期的一个部分乘积中,硬件中计算误差在 1 ulp 内的内积是可能的。另外需要的硬件可以与为获得该速度所需的乘数数组进行比较。

在标准中提到的所有运算都需要精确舍入，但十进制和二进制之间的转换除外。原因是精确舍入所有运算的高效算法是已知的（转换除外）。对于转换，已知的最好的有效算法所产生的结果比精确舍入的结果稍差 [Coonen 1984]。

IEEE 标准因考虑到制表人的困难而不要求精确舍入超越函数。为了说明这一点，假定您正在制作 4 位指数函数表。那么， $\exp(1.626) = 5.0835$ 。应该将它舍入为 5.083 还是 5.084？如果更仔细地计算，则结果变成 5.08350。接下来是 5.083500。然后是 5.0835000。因为  $\exp$  是超越函数，所以在辨别  $\exp(1.626)$  是 5.083500...0ddd 还是 5.0834999...9ddd 之前该过程会继续任意长的时间。因此，指定超越函数的精度与按无限精度计算它再进行舍入所得的精度相同是不实用的。另一种方法是从算法上指定超越函数。但是，在所有硬件架构上都工作良好的单个算法似乎是不存在的。有理逼近、CORDIC、<sup>1</sup> 和大表是用于在当代计算机上计算超越函数的三种不同方法。每种方法适用于不同种类的硬件，目前还没有一种算法可以被当前广泛的硬件所接受。

## 特殊数量

在一些浮点硬件上，每种位模式都表示一个有效的浮点数。IBM System/370 就是这样的一个示例。另一方面，VAX™ 保留一些位模式以表示称为保留操作数的特殊数。这一思想可追溯到 CDC 6600，它具有表示特殊数量 INDEFINITE 和 INFINITY 的位模式。

IEEE 标准继承了这一传统，它具有 NaN（不是数）和无穷大。如果不使用任何特殊数量，则不存在处理异常情况（例如接受负数的平方根，而不是终止计算）的好方法。在 IBM System/370 FORTRAN 中，响应计算负数（如 -4）的平方根的缺省操作导致错误消息的打印。因为每种位模式都表示一个有效数，所以平方根的返回值一定是某个浮点数。对于 System/370 FORTRAN，返回的是  $\sqrt{-4} = 2$ 。在 IEEE 运算中，此情况下返回 NaN。

IEEE 标准指定了以下特殊值（请参见表 D-2）： $\pm 0$ 、非正规化数、 $\pm\infty$  和 NaN（如下一部分中所述，存在多个 NaN）。这些特殊值都是使用  $e_{\max} + 1$  或  $e_{\min} - 1$  的指数进行编码的（已经指出 0 的指数是  $e_{\min} - 1$ ）。

表 D-2 IEEE 754 特殊值

| 指数                              | 尾数部分       | 表示                         |
|---------------------------------|------------|----------------------------|
| $e = e_{\min} - 1$              | $f = 0$    | $\pm 0$                    |
| $e = e_{\min} - 1$              | $f \neq 0$ | $0. f \times 2^{e_{\min}}$ |
| $e_{\min} \leq e \leq e_{\max}$ | —          | $1.f \times 2^e$           |
| $e = e_{\max} + 1$              | $f = 0$    | $\pm\infty$                |
| $e = e_{\max} + 1$              | $f \neq 0$ | NaN                        |

1. CORDIC 是 Coordinate Rotation Digital Computer（坐标旋转数字计算机）的缩写，它是计算超越函数的一种方法，主要使用移位和加法（即非常少的乘法和除法）[Walther 1971]。在此方法中，另外需要的硬件可以与获得在 Intel 8087 和 Motorola 68881 上使用的速度所需要的乘法器数组进行比较。

# NaN

按照传统，将  $0/0$  或  $\sqrt{-1}$  视为导致计算终止的不可恢复的错误。但是，存在着一些示例，表明在这样的情况下继续进行计算是有意义的。以一个子例程为例，该子例程用于查找函数  $f$  所包含的零，假设此子例程为  $\text{zero}(f)$ 。传统情况下，零查找器要求用户输入一个区间  $[a, b]$ ，函数是在其上定义的，零查找器将按其进行搜索。也就是说，子例程是以  $\text{zero}(f, a, b)$  的形式调用的。更有效的零查找器不要求用户输入此额外信息。这种较普遍的零查找器尤其适合于计算器，通常只是键入一个函数，然而不方便的是必须指定定义域。您可以很容易地了解到大多数零查找器需要指定定义域的原因。零查找器是通过在各个值上探查函数  $f$  来执行工作的。如果探查到的  $f$  定义域之外的值，则  $f$  的代码可能也计算  $0/0$  或  $\sqrt{-1}$ ，计算将停止，但未必终止零查找过程。

通过引入称为 NaN 的特殊值，并指定诸如  $0/0$  和  $\sqrt{-1}$  之类的表达式计算来生成 NaN 而不是停止计算，就可以避免此问题。表 D-3 中列出了一些可以导致 NaN 的情况。这样，当  $\text{zero}(f)$  在  $f$  的定义域之外探查时， $f$  的代码将返回 NaN，并且零查找器可以继续执行。也就是说， $\text{zero}(f)$  未因作出错误推测而受到“惩罚”。记住此示例，就可以很容易地了解到将 NaN 与普通浮点数结合在一起的结果应该是什么。假定  $f$  的最后一条语句是  $\text{return}(-b + \text{sqrt}(d))/(2*a)$ 。如果  $d < 0$ ，则  $f$  应该返回 NaN。因为  $d < 0$ ，所以  $\text{sqrt}(d)$  是一个 NaN，而且  $-b + \text{sqrt}(d)$  将是一个 NaN（如果 NaN 和任何其它数的和是 NaN）。类似地，如果除法运算的一个操作数是 NaN，则商应该是 NaN。通常，每当 NaN 参与浮点运算时，结果都是另一 NaN。

表 D-3 产生 NaN 的运算

| 操作             | 产生 NaN 的表达式                               |
|----------------|---|
| +              | $\infty + (-\infty)$                      |
| ×              | $0 \times \infty$                         |
| /              | $0/0, \infty/\infty$                      |
| REM            | $x \text{ REM } 0, \infty \text{ REM } y$ |
| $\sqrt{\quad}$ | $\sqrt{x}$ (当 $x < 0$ 时)                  |

编写不要求用户输入定义域的零解算程序的另一方法是使用信号。零查找器可以为浮点异常安装一个信号处理程序。这样，如果  $f$  在其定义域之外进行计算且引发了一个异常，则将控制权返回给零解算程序。此方法的问题是每种语言都具有不同的信号处理方法（如果它有一个方法的话），因而不具备可移植性。

在 IEEE 754 中，通常将 NaN 表示为带有指数  $e_{\max} + 1$  和非零有效数字的浮点数。实现可以自由地将系统相关的信息置入有效位。这样，就不存在唯一的 NaN，而是整个系列的 NaN。将 NaN 和普通浮点数结合在一起时，结果应该与 NaN 操作数相同。因此，如果长计算的结果是 NaN，则有效数字中的系统相关信息将是生成计算中的第一个 NaN 时产生的信息。实际上，对于最后一个声明，有一个防止误解的说明：如果两个操作数都是 NaN，则结果将是其中的一个 NaN，但它可能不是首先生成的 NaN。

## 无穷大

正如 NaN 提供了一种在遇到诸如  $0/0$  或  $\sqrt{-1}$  之类的表达式时继续进行计算的方法一样，无穷大提供了一种在发生上溢时继续执行的方法。这比仅返回可表示的最大数要安全得多。以计算  $\sqrt{x^2 + y^2}$  为例，此时  $\beta = 10$ ,  $p = 3$  且  $e_{\max} = 98$ 。如果  $x = 3 \times 10^{70}$  并且  $y = 4 \times 10^{70}$ ，则  $x^2$  将发生上溢，并被替换为  $9.99 \times 10^{98}$ 。类似地， $y^2$  和  $x^2 + y^2$  都将依次上溢，并由  $9.99 \times 10^{98}$  替换。因此，最终结果将是  $\sqrt{9.99 \times 10^{98}} = 3.16 \times 10^{49}$ ，该结果绝对是错误的：正确结果应该是  $5 \times 10^{70}$ 。在 IEEE 运算中， $x^2$  的结果是  $\infty$ ， $y^2$ 、 $x^2 + y^2$  和  $\sqrt{x^2 + y^2}$  也是这样。因此最终结果是  $\infty$ ，这比返回根本不接近于正确结果的普通浮点数要安全。<sup>1</sup>

用 0 除 0 会产生一个 NaN。但是，用一个非零数去除以零则会返回无穷大： $1/0 = \infty$ ， $-1/0 = -\infty$ 。这一区别的原因是：如果在  $x$  接近某个极限时  $f(x) \rightarrow 0$  且  $g(x) \rightarrow 0$ ，则  $f(x)/g(x)$  可以具有任意值。例如，如果  $f(x) = \sin x$  且  $g(x) = x$  时，则当  $x \rightarrow 0$  时  $f(x)/g(x) \rightarrow 1$ 。但是，如果  $f(x) = 1 - \cos x$ ，则  $f(x)/g(x) \rightarrow 0$ 。将  $0/0$  视为两个非常小的数的商的极限情况时， $0/0$  可以表示任意数。因此，在 IEEE 标准中， $0/0$  产生一个 NaN。但是，如果  $c > 0$ ， $f(x) \rightarrow c$ ，且  $g(x) \rightarrow 0$ ，则对于任意分析函数  $f$  和  $g$ ， $f(x)/g(x) \rightarrow \pm\infty$ 。如果对于小的  $x$ ，存在  $g(x) < 0$ ，则  $f(x)/g(x) \rightarrow -\infty$ ，否则极限是  $+\infty$ 。因此，IEEE 标准定义  $c/0 = \pm\infty$ （只要  $c \neq 0$ ）。通常情况下， $\infty$  的符号取决于  $c$  的符号和 0 的符号，这样  $-10/0 = -\infty$ ，而  $-10/-0 = +\infty$ 。您可以区分因上溢而得到  $\infty$  和因除以零而得到的  $\infty$ ，方法是检查状态标志，该方法将“标志”（在第 198 页上）部分中详细讨论。在第一种情况下将设置上溢标志，而在第二种情况下设置除以零标志。

对于将无穷大作为操作数的运算，确定其结果的规则比较简单：将无穷大替换为有限数  $x$  并将极限视为  $x \rightarrow \infty$ 。因此， $3/\infty = 0$ ，因为

$$\lim_{x \rightarrow \infty} 3/x = 0$$

类似地， $4 - \infty = -\infty$ ， $\sqrt{\infty} = \infty$ 。当极限不存在时，结果是 NaN，因此  $\infty/\infty$  将是一个 NaN（表 D-3 包含其它示例）。这与用于得出  $0/0$  应为 NaN 的结论的推理是一致的。

当子表达式计算为 NaN 时，整个表达式的值也是 NaN。但是，对于  $\pm\infty$  的情况，表达式的值可能是一个普通的浮点数，因为存在类似  $1/\infty = 0$  的规则。下面是利用无穷大运算规则的实际示例。以计算函数  $x/(x^2 + 1)$  为例。这个公式并不好，因为不仅它在  $x$  大于  $\sqrt{\beta} \beta_{\max}^{1/2}$  时会发生上溢，而且无穷大运算将给出错误结果（因为它产生 0，而不是接近  $1/x$  的数值）。但是，可以将  $x/(x^2 + 1)$  写为  $1/(x + x^{-1})$ 。这一改进的表达式不会过早地出现上溢，并因无穷大运算而具有正确的值（在  $x = 0$  时  $1/(0 + 0^{-1}) = 1/(0 + \infty) = 1/\infty = 0$  时）。如果不使用无穷大运算，则表达式  $1/(x + x^{-1})$  要求测试  $x = 0$ ，该测试不仅增加额外的指令，而且可能使流水线中断。此示例说明了一个一般事实，即无穷大运算通常避免进行特殊情况检查的需要；但是，需要仔细检查公式以确保它们在无穷大时没有伪行为（如  $x/(x^2 + 1)$  具有伪行为）。

1. 细微的要点：虽然缺省情况下 IEEE 运算会将发生上溢的数据舍入为  $\infty$ ，但是更改这一缺省设置是可能的。请参见“舍入模式”（在第 197 页上）。



## 有符号零

零是由指数  $e_{\min} - 1$  和零有效位数表示的。由于符号位可以具有两个不同的值，所以有两个零：+0 和 -0。如果在比较 +0 和 -0 时进行了区分，则类似 `if (x == 0)` 的简单测试将具有非常不可预知的行为，具体取决于  $x$  的符号。因此，IEEE 标准定义了一种比较关系，使得 +0 = -0，而不是 -0 < +0。虽然忽略零的符号始终是可能的，但是 IEEE 标准没有这样做。当乘法或除法涉及有符号的零时，在计算结果符号时就可以应用通常的符号规则。因此， $3 \cdot (+0) = +0$ ， $+0 / 3 = -0$ 。如果零没有符号，则在  $x = \pm\infty$  时关系  $1/(1/x) = x$  将无法成立。原因是  $1/-\infty$  和  $1/+\infty$  的结果都是 0，而  $1/0$  的结果是  $+\infty$ ，符号信息已经丢失。恢复恒等式  $1/(1/x) = x$  的一种方法是仅具有一种无穷大，但是那将导致丢失上溢数量的符号的灾难性结果。

使用有符号零的另一示例涉及下溢和在 0 处具有不连续性的函数（如 `log`）。在 IEEE 运算中，设置 `log 0 = -∞` 以及当  $x < 0$  时将 `log x` 定义为 NaN 是很自然的。假设  $x$  表示一个已经下溢到零的小负数。由于使用了有符号零，因此  $x$  将是负的，这样 `log` 可以返回 NaN。但是，如果不存在有符号零，则 `log` 函数无法区分下溢的负数与 0，因此将不得不返回 -∞。在 0 处具有不连续性的函数的另一示例是 `signum` 函数，它返回数的符号。

有符号零的最值得关注的用途可能出现在复杂运算中。举一个简单的示例，请假设等式  $\sqrt{1/z} = 1/(\sqrt{z})$ 。当  $z \geq 0$  时，这肯定是正确的。如果  $z = -1$ ，很明显计算结果给出  $\sqrt{1/(-1)} = \sqrt{-1} = i$  和  $1/(\sqrt{-1}) = 1/i = -i$ 。因此会出现： $\sqrt{1/z} \neq 1/(\sqrt{z})$ ！可以将此问题归因于以下事实：平方根是多值的，无法选择值使它在整个复平面中是连续的。但是，如果不考虑包括所有负实数的分支切割，则平方根是连续的。这样就留下了如何处理负实数的问题，负实数的形式为  $-x + i0$ ，其中  $x > 0$ 。有符号零提供了解决此问题的完美方法。形式为  $x + i(+0)$  的数具有一个符号 ( $i\sqrt{x}$ )，分支切割另一侧上形式为  $x + i(-0)$  的数具有另一符号 ( $-i\sqrt{x}$ )。事实上，计算  $\sqrt{\phantom{z}}$  的自然公式将给出这些结果。

再回到  $\sqrt{1/z} = 1/(\sqrt{z})$ 。如果  $z = -1 = -1 + i0$ ，那么

$$1/z = 1/(-1 + i0) = [(-1 - i0)]/[(-1 + i0)(-1 - i0)] = (-1 - i0)/((-1)^2 - 0^2) = -1 + i(-0)$$

因此  $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$ ，而  $1/(\sqrt{z}) = 1/i = -i$ 。这样，IEEE 运算为所有  $z$  保持了此恒等式。一些更复杂的示例由 Kahan [1987] 给出。虽然区分 +0 和 -0 具有优点，但是有时候会令人迷惑。例如，有符号零破坏了关系  $x = y \Leftrightarrow 1/x = 1/y$ ，在  $x = +0$  且  $y = -0$  时它是错误的。但是，IEEE 委员会断定利用有符号零的优点大于其缺点。

## 非正规化数

以  $\beta = 10$ 、 $p = 3$  且  $e_{\min} = -98$  的正规化浮点数为例子。数  $x = 6.87 \times 10^{-97}$  和  $y = 6.81 \times 10^{-97}$  看起来完全是普通的浮点数，它们是最小浮点数  $1.00 \times 10^{-98}$  的 10 倍多。但是，它们具有一个奇怪的属性： $x \ominus y = 0$ （即使  $x \neq y$ ）！原因是： $x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$  太小以致于无法表示为正规化数，因此它必须清零。保留

$$x = y \Leftrightarrow x - y = 0 \text{ 属性有怎样的重要性呢?} \tag{10}$$

很容易设想编写以下代码段 `if (x ≠ y) then z = 1/(x-y)`, 程序会由于伪除以零而在运行了较长的时间后失败。跟踪与此类似的错误既费力又耗时。让我们站在更有哲理的级别上, 计算机科学教科书通常指出: 即使当前要证明大程序正确, 这看起来有些不切实际, 但利用证明程序的思想来设计程序, 这通常会产生更好的代码。例如, 即使不变量将不被用作证明的一部分, 引入它也是相当有用的。浮点代码就像任何其它代码一样: 它有助于具备所要依赖的可证明事实。例如, 当分析公式 (6) 时, 了解  $x/2 < y < 2x \Rightarrow x \ominus y = x - y$  是非常有帮助的。类似地, 如果知道 (10) 是正确的, 就可以更轻松地编写可靠的浮点代码。如果它仅对大部分的数字是正确的, 那么不能使用它来证明一切。

IEEE 标准使用非正规化<sup>1</sup> 数, 这保证了 (10) 以及其它有用的关系。它们是该标准中最有争议的部分, 这可能就是 754 经过很长时间的延迟才得以批准的原因。声称符合 IEEE 标准的大多数高性能硬件并不直接支持非正规化数, 而是在使用或产生非正规数时支持陷阱, 并将其留给软件来模拟 IEEE 标准。<sup>2</sup> 支持非正规化数的思想可追溯到 Goldberg [1967], 这种思想非常简单。当指数是  $e_{\min}$  时, 有效数字不必进行正规化, 这样在  $\beta = 10$ 、 $p = 3$  且  $e_{\min} = -98$  时  $1.00 \times 10^{-98}$  不再是最小的浮点数, 因为  $0.98 \times 10^{-98}$  也是一个浮点数。

当  $\beta = 2$  且使用隐藏位时, 存在一个小的意外困难, 因为指数为  $e_{\min}$  的数将始终具有大于或等于 1.0 的有效数字 (由于存在隐式前导位)。解决方法与表示 0 所使用的方法类似, 并且总结在表 D-2 中。指数  $e_{\min}$  用于表示非正规数。更正式的表述是: 如果有效数字域中的位是  $b_1, b_2, \dots, b_{p-1}$ , 且指数的值是  $e$ , 那么当  $e > e_{\min} - 1$  时, 所表示的数是  $1.b_1b_2\dots b_{p-1} \times 2^e$ , 而在  $e = e_{\min} - 1$  时, 所表示的数是  $0.b_1b_2\dots b_{p-1} \times 2^{e+1}$ 。指数中的 +1 是必需的, 因为非正规数的指数是  $e_{\min}$ , 而不是  $e_{\min} - 1$ 。

请回想一下在本部分开头给出的示例:  $\beta = 10$ ,  $p = 3$ ,  $e_{\min} = -98$ ,  $x = 6.87 \times 10^{-97}$  且  $y = 6.81 \times 10^{-97}$ 。对于非正规数,  $x - y$  不清零, 而改为由非正规化的数  $.6 \times 10^{-98}$  来表示。这种行为称为渐进下溢。验证使用渐进下溢时 (10) 始终成立是很容易的。

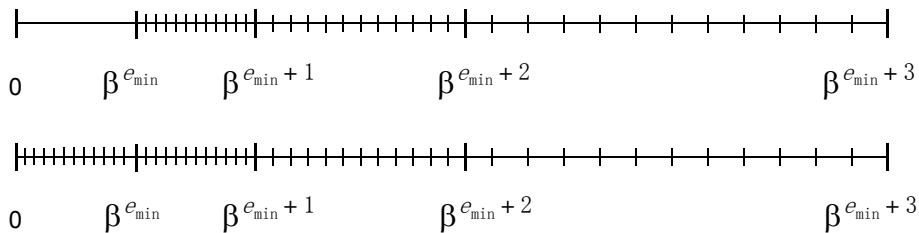


图 D-2 清零与渐进下溢的比较

1. 在 854 中, 它们被称为 *subnormal* “次正规的数”; 在 754 中, 称为 *denormal* “非正规数”。  
 2. 这是导致标准的最棘手方面之一的原因。在使用软件陷阱的硬件上, 频繁下溢的程序通常运行得非常缓慢。

图 D-2 说明非正规化的数。图中的上实数直线显示正规化的浮点数。请注意 0 和最小正规化数  $1.0 \times \beta^{e_{min}}$  之间的间隔。如果浮点计算的结果位于此间隔内，则将其清零。下实数直线显示将非正规数增加到一组浮点数时的情况。“间隔”被填充，当计算的结果小于  $1.0 \times \beta^{e_{min}}$  时，将使用最接近的非正规数来表示。将非正规化的数增加到实数直线时，邻近浮点数之间的间距的变化是有规律的：邻近间距要么长度相同，要么按  $\beta$  的因子变化。如果不使用非正规数，则间距突然从  $\beta^{-p+1}\beta^{e_{min}}$  变化到  $\beta^{e_{min}}$ （是  $\beta^{p-1}$  的因子），而不是按  $\beta$  的因子有规律地变化。由于这一原因，当使用渐进下溢时，对于接近下溢阈值的正规化数会具有较大相对误差的许多算法来说，在此范围内表现良好。

如果不使用渐进下溢，则对于正规化的输入来说，简单表达式  $x - y$  会具有非常大的相对误差，正如上面所看到的  $x = 6.87 \times 10^{-97}$  和  $y = 6.81 \times 10^{-97}$  的相对误差。即使不存在消去操作，大的相对误差也会发生，如下例所示 [Demmel 1984]。以两个复数  $a + ib$  和  $c + id$  相除为例。很明显，公式

$$\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

受到以下问题的影响：如果分母  $c + id$  的任一部分大于  $\sqrt{\beta}\beta^{e_{max}/2}$ ，则公式将发生上溢，即使最终结果可能完全处于范围之内也是如此。计算商的一种较好的方法是使用 Smith 的公式：

$$\frac{ib}{id} = \begin{cases} \frac{a + b(d/c)}{c + d(d/c)} + i \frac{b - a(d/c)}{c + d(d/c)} & \text{if } |d| < |c| \\ \frac{b + a(c/d)}{d + c(c/d)} + i \frac{-a + b(c/d)}{d + c(c/d)} & \text{if } |d| \geq |c| \end{cases} \quad (11)$$

将 Smith 的公式应用于  $(2 \cdot 10^{-98} + i10^{-98}) / (4 \cdot 10^{-98} + i(2 \cdot 10^{-98}))$  将得出具有渐进下溢的正确结果 0.5。在清零时它产生 0.4，此时误差为 100 ulp。非正规化的数通常可以保证参数的误差边界一直下降到  $1.0 \times \beta^{e_{min}}$ 。

## 异常、标志和陷阱处理程序

在 IEEE 运算中出现类似除以零或上溢等异常情况时，缺省设置是传送结果并继续执行。典型的缺省结果如下：对于 0/0 和  $\sqrt{-1}$ ，结果是 NaN；对于 1/0 和上溢，结果是  $\infty$ 。前面的几部分给出了从具有这些缺省值的异常情况中继续执行属于合理操作的示例。当出现任何异常时，还会设置状态标志。要为用户提供一种读取和写入状态标志的方法，需要实施 IEEE 标准。这些标志具有“粘滞性”，因为一旦设置，那么在被显式清除以前它们都将一直保持这种设置。测试标志是区分 1/0（它是真正的无穷大）与上溢的唯一方法。

有时，在出现异常情况时继续执行是不正确的。“无穷大”（在第 192 页上）部分给出了示例  $x/(x^2 + 1)$ 。当  $x > \sqrt{\beta} \beta^{e_{\max}^{1/2}}$  时，分母是无穷大，产生最终结果 0，这完全是错误的结果。虽然此公式可以通过改写为  $1/(x + x^{-1})$  来解决这个问题，但是改写并不总是能够解决问题。IEEE 标准强烈建议实施要允许安装陷阱处理程序。这样，当出现异常时，将调用陷阱处理程序，而不是设置标志。陷阱处理程序返回的值将用作运算的结果。清除或设置状态标志是陷阱处理程序的责任；否则，允许标志的值未定义。

IEEE 标准将异常分成 5 类：上溢、下溢、除以零、无效运算和不精确。每类异常都有各自的状态标志。前三类异常的含义是不言而喻的。无效运算包括表 D-3 中列出的情况，以及产生 NaN 的任何比较。导致无效异常的运算的缺省结果是返回一个 NaN，但反过来是不正确的。当某个运算的其中一个操作数是 NaN 时，结果是 NaN，但不引发无效异常，除非该运算还符合表 D-3 中列出的条件之一。<sup>1</sup>

表 D-4 IEEE 754\* 中的异常

| 异常   | 禁止陷阱时的结果                       | 陷阱处理程序的参数                    |
|------|--------------------------------|------------------------------|
| 上溢   | $\pm\infty$ 或 $\pm x_{\max}$   | $\text{round}(x2^{-\alpha})$ |
| 下溢   | $0$ 、 $\pm 2^{e_{\min}}$ 或非正规数 | $\text{round}(x2^{\alpha})$  |
| 除以零  | $\pm\infty$                    | 操作数                          |
| 无效运算 | NaN                            | 操作数                          |
| 不精确  | $\text{round}(x)$              | $\text{round}(x)$            |

\* $x$  是运算的准确结果，对于单精度有  $\alpha = 192$ ；对于双精度为 1536，而且  $x_{\max} = 1.11 \dots 11 \times 2^{e_{\max}}$

当浮点运算的结果不精确时，将引发不精确异常。在  $\beta = 10$ 、 $p = 3$  系统中， $3.5 \otimes 4.2 = 14.7$  是精确的，而  $3.5 \otimes 4.3 = 15.0$  是不精确的（因为  $3.5 \cdot 4.3 = 15.05$ ），这将引发不精确异常。“二进制到十进制的转换”（在第 217 页上）讨论了使用不精确异常的算法。表 D-4 中列出了上述五类异常行为的摘要。

存在一个与以下事实有关的实施问题：不精确异常频繁引发。如果浮点硬件没有自己的标志，而是中断操作系统以发出存在浮点异常的信号，则不精确异常的成本可能会非常高。通过由软件来维护状态标志，可以消除此成本。首次引发异常时，设置相应类别的软件标志，并通知浮点硬件屏蔽该类异常。此后，所有后继异常都将在不中断操作系统的情况下运行。当用户重置这个状态标志时，将重新启用硬件屏蔽。

1. 除非运算中产生“捕获”NaN，否则不引发无效异常。请参见 IEEE 标准 754-1985 的 6.2 节。—编辑

## 陷阱处理程序

陷阱处理程序的一个明显用途是用于向后兼容。那些应当在出现异常时终止的旧代码可通过安装陷阱处理程序来达到终止进程的目的。这对于带有类似 `do S until (x >= 100)` 循环的代码尤其有用。由于将 NaN 与数值进行 `<`、`<=`、`>`、`>=` 或 `=` (但不包括 `≠`) 比较操作时, 始终会返回 “False”, 所以只要 `x` 取 NaN, 此代码就将进入无限循环。

陷阱处理程序还有一种更值得关注的用途: 用于计算诸如  $\prod_{j=1}^k x_j$  等有可能导致上溢的乘积。这种问题的一种解决方案是使用对数并改为计算  $\exp(\sum \log x_j)$ 。该方法存在的问题是: 准确性较低, 且计算成本高于简单表达式  $\prod x_j$ , 即使没有出现上溢也是如此。另一种被称为 *上溢/下溢计数* 的解决方案使用陷阱处理程序, 可以避免上述两个问题 [Sterbenz 1974]。

该方案的基本思想是: 设置一个全局计数器并将其初始化为零。只要  $p_k = \prod_{j=1}^k x_j$  部分乘积对于某个  $k$  值发生上溢, 陷阱处理程序就会将计数器加 1, 并将指数部分折返后的上溢量返回。在 IEEE 754 单精度中,  $e_{\max} = 127$ , 因此如果  $p_k = 1.45 \times 2^{130}$ , 则会发生上溢并导致调用陷阱处理程序, 该程序将指数限制在范围内,  $p_k$  被更改为  $1.45 \times 2^{-62}$  (请参见下面内容)。类似地, 如果  $p_k$  发生下溢, 则将计数器减 1, 并将负指数折返为正指数。在完成所有乘法时, 如果计数器是零, 则最终乘积为  $p_n$ 。如果计数器是正数, 则说明乘积上溢; 如果计数器是负数, 则表明乘积下溢。如果乘积的任一部分都没有超出范围, 则永远不会调用陷阱处理程序, 而且计算也不会引起额外成本。即使存在上溢/下溢, 计算结果也比使用对数计算的结果更为准确, 因为每个  $p_k$  都是使用完全精度乘法从  $p_{k-1}$  计算的。Barnett [1987] 讨论了一个公式, 利用上溢/下溢计数的完全准确度在该公式的早期表中发现了一个误差。

IEEE 754 规定: 当调用上溢或下溢陷阱处理程序时, 折返结果将作为参数传递给它。上溢折返的定义是: 计算结果时先计算到近似无限精度, 再除以  $2\alpha$ , 然后舍入到相关的精度。对于下溢, 将结果乘以  $2\alpha$ 。指数  $\alpha$  是 192 (对于单精度) 或 1536 (对于双精度)。这就是上面的示例中  $1.45 \times 2^{130}$  被转换为  $1.45 \times 2^{-62}$  的原因。

## 舍入模式

在 IEEE 标准中, 每当运算结果不精确时都将出现舍入操作, 因为每个运算都是先精确计算然后再进行舍入 (二进制与十进制转换除外)。缺省情况下, 舍入是指向最相近的数据舍入。该标准要求提供其它三种舍入模式, 即向 0 舍入、向  $+\infty$  舍入和向  $-\infty$  舍入。在用于转换为整数的运算时, 向  $-\infty$  舍入使转换变成 “取最小” 函数, 而向  $+\infty$  舍入使转换变成 “取上限” 函数。舍入模式影响上溢, 因为当采用向 0 舍入或向  $-\infty$  舍入时, 正量的上溢使得缺省结果成为最大的可表示数, 而不是  $+\infty$ 。类似地, 当采用向  $+\infty$  舍入或向 0 舍入时, 负量的上溢将产生最大的负数。

舍入模式的一个应用发生在区间运算中，另一个应用将在“二进制到十进制的转换”（在第 217 页上）中提及。当使用区间运算时，两个数  $x$  和  $y$  的和是一个区间  $[z, \bar{z}]$ ，其中  $z$  是向  $-\infty$  舍入的  $x \oplus y$ ， $\bar{z}$  是向  $+\infty$  舍入的  $x \oplus y$ 。加法运算的精确结果包含在区间  $[z, \bar{z}]$  中。如果不使用舍入模式，则区间运算常常通过计算  $z = (x \oplus y)(1 - \varepsilon)$  和  $\bar{z} = (x \oplus y)(1 + \varepsilon)$  实现，其中  $\varepsilon$  是计算机厄普西隆。<sup>1</sup> 这将导致过高估计区间的大小。因为区间运算中的运算结果是一个区间，所以一般情况下运算输入也将是一个区间。如果将两个区间  $[x, \bar{x}]$  和  $[y, \bar{y}]$  相加，结果是  $[z, \bar{z}]$ ，其中  $z$  是  $x \oplus y$ （其舍入模式设置为向  $-\infty$  舍入）， $\bar{z}$  是  $z \oplus y$ （其舍入模式设置为向  $+\infty$  舍入）。

当使用区间运算执行浮点计算时，最终结果是一个包含精确计算结果的区间。如果结果区间很大（通常是这样），那么对运算的帮助意义有限。因为正确结果可以处于该区间内的任意位置。当与多精度浮点包联合使用时，区间运算更有意义。首先通过某个精度  $p$  执行计算。如果区间运算指出最终结果可能不准确，则使用更高的精度重新计算，直到最终区间达到合理大小。

## 标志

IEEE 标准有许多标志和模式。如前所述，以下五类异常中的每一类都具有一个状态标志：下溢、上溢、除以零、无效运算和不精确。舍入模式有以下四种：向最相近的数据舍入、向  $+\infty$  舍入、向 0 舍入和向  $-\infty$  舍入。强烈建议对于五类异常中的每一类都设置一个启用模式位。此部分给出了一些简单的示例，说明如何利用这些模式和标志。更复杂的示例将在“二进制到十进制的转换”（在第 217 页上）部分中进行讨论。

我们来考虑编写一个计算  $x^n$  的子例程，其中  $n$  是一个整数。当  $n > 0$  时，一种简单的例程类似于下面的程序段

```
PositivePower(x,n) {
  while (n is even) {
    x = x*x
    n = n/2
  }
  u = x
  while (true) {
    n = n/2
    if (n==0) return u
    x = x*x
    if (n is odd) u = u*x
  }
}
```

1.  $z$  可能大于  $\bar{z}$ （如果  $x$  和  $y$  都是负数）。—编辑

如果  $n < 0$ ，则计算  $x^n$  的更精确方法不是调用 `PositivePower(1/x, -n)`，而是调用 `1/PositivePower(x, -n)`，因为第一个表达式将  $n$  个数量相乘，其中每个数量都具有来自除法运算（即  $1/x$ ）的舍入误差。在第二个表达式中它们是精确的（即  $x$ ），而且最后的除法只产生一个附加的舍入误差。可惜的是，此策略有一个微小的缺陷。如果 `PositivePower(x, -n)` 出现下溢，那么要么调用下溢陷阱处理程序，要么将设置下溢状态标志。这是错误的，因为如果  $x^{-n}$  出现下溢，则  $x^n$  将上溢或处于范围内。<sup>1</sup> 但是，由于 IEEE 标准允许用户访问所有标志，所以子例程可以轻松地对此进行更正。它只需关闭上溢和下溢陷阱启用位，并保存上溢和下溢状态位。然后计算 `1/PositivePower(x, -n)` 即可。如果既没有设置上溢状态位也没有设置下溢状态位，那么子例程会将它们与陷阱启用位一起恢复。如果设置了其中一种状态位，则子例程会恢复标志并使用 `PositivePower(1/x, -n)`（这将导致出现正确的异常）重新计算。

另一个使用标志的示例是通过下列公式计算余弦

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}$$

如果  $\arctan(\infty)$  计算为  $\pi/2$ ，则  $\arccos(-1)$  将正确计算为  $2 \cdot \arctan(\infty) = \pi$ （因为是无限运算）。但是，有一个小缺陷，因为  $(1-x)/(1+x)$  的计算结果将导致除以零异常标志被设置，即使  $\arccos(-1)$  不属于异常也是如此。此问题的解决方法很简单。只需在计算反余弦之前保存除以零标志的值，然后在计算之后恢复其旧值即可。

---

## 系统方面

对于计算机系统来说，几乎每个方面的设计都需要有关浮点的知识。计算机架构通常具有浮点指令，编译器必须生成那些浮点指令，而且操作系统必须决定在这些浮点指令的异常条件被满足时应执行怎样的操作。计算机系统人员很少从数值分析教材中获得指导，这些教材通常针对软件的用户和编写人员，而不是针对计算机设计人员。为了解那些看似合理的设计决策如何导致未预料的行为，我们来研究下面的 BASIC 程序。

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
    else print "Not Equal"
```

在 IBM PC 上使用 Borland 的 Turbo Basic 进行编译和运行时，该程序打印 Not Equal! 此示例将在下一部分中进行分析

---

1. 它可以在范围内，因为如果  $x < 1$ ， $n < 0$  且  $x^{-n}$  只是稍微小于下溢阈值  $2^{e_{\min}}$ ，则  $x^n \approx 2^{-e_{\min}} < 2^{e_{\max}}$ ，因此可能不上溢，因为在所有 IEEE 精度中， $-e_{\min} < e_{\max}$ 。

顺便说一下，有些人认为此类异常的解决方法是决不要比较浮点数以确定是否相等，而是当它们处于某个误差界限  $E$  内时将其视为相等。这样并不能解决一切问题，因为它引发了新的问题。 $E$  的值应该是多少？如果  $x < 0$  和  $y > 0$  都在  $E$  内，那么是否应该忽略符号的差异而将它们视为相等？此外，此规则定义的关系  $a \sim b \Leftrightarrow |a - b| < E$  不是等价关系，因为  $a \sim b$  且  $b \sim c$  不能得出  $a \sim c$ 。

## 指令集

对于算法来讲，为了产生准确的结果，经常要求局部数据集中采用更高的精度。在二次方程求根公式中就有这样的示例  $(-b \pm \sqrt{b^2 - 4ac})/2a$ 。如“定理 4 的证明”（在第 215 页上）部分中所述，当  $b^2 \approx 4ac$  时，舍入误差最多可以影响使用二次方程求根公式计算的根中的一半数位。通过以双精度执行子计算  $b^2 - 4ac$  时，会丢失根的双精度位的一半，这意味着将保留所有单精度位。

如果有一条对两个单精度数进行操作并产生双精度结果的乘法指令，则当  $a$ 、 $b$  和  $c$  都是单精度数时按双精度计算  $b^2 - 4ac$  是很容易的。为了产生两个  $p$  位数的精确舍入乘积，乘法器需要生成整个  $2p$  位的乘积，尽管它可能在继续执行时抛弃位。这样，从单精度操作数计算双精度乘积的硬件通常只比单精度乘法器的成本略高一些，而且比双精度乘法器的成本低很多。尽管如此，目前的指令集往往仅提供生成结果与操作数精度相同的指令。<sup>1</sup>

如果这种用两个单精度操作数产生双精度乘积的指令仅适用于二次方程求根公式，那么将该指令增加到指令集中是不值得的。但是，此指令有许多其它用途。以解决线性方程组的问题为例，

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

...

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

可以按矩阵形式将它写为  $Ax = b$ ，其中

---

1. 这可能是由于设计人员喜欢“正交”指令集，在这样的指令集中浮点指令的精度与实际运算无关。乘法的特殊情况会破坏这一正交性。



$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2(1)n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

假定按某种方法（或许是高斯消元法）求出了一个解  $x^{(1)}$ 。那么有一种简单的方法可改进结果的准确度，这种方法称为 *迭代改进*。首先计算

$$\xi = Ax^{(1)} - b \quad (12)$$

然后对方程组进行求解

$$Ay = \xi \quad (13)$$

请注意，如果  $x^{(1)}$  是一个精确解，则  $\xi$  是零向量， $y$  也是零向量。通常， $\xi$  和  $y$  的计算会引起舍入误差，因此  $Ay \approx \xi \approx Ax^{(1)} - b = A(x^{(1)} - x)$ ，其中  $x$  是（未知的）正确解。那么， $y \approx x^{(1)} - x$ ，因此解的改进估计是

$$x^{(2)} = x^{(1)} - y \quad (14)$$

重复执行三个步骤 (12)、(13) 和 (14)，用  $x^{(2)}$  替换  $x^{(1)}$ ，用  $x^{(3)}$  替换  $x^{(2)}$ 。这里关于  $x^{(i+1)}$  比  $x^{(i)}$  更精确的论述仅是非正式的。有关详细信息，请参见 [Golub 和 Van Loan 1989]。

在执行迭代改进时， $\xi$  是一个向量，其元素是邻近不精确浮点数的差，因此会受到恶性抵消的影响。这样，迭代改进不是非常有用，除非  $\xi = Ax^{(1)} - b$  是按双精度计算的。同样，这是计算两个单精度数（ $A$  和  $x^{(1)}$ ）的乘积的示例，它需要完全双精度结果。

总而言之，如果存在这样的指令：对两个浮点数做相乘操作并返回两倍于被操作数精度的乘积，那么将该指令增加到指令集中是很有用的。对于编译器而言，这其中的一些含义将在下一部分进行讨论。

## 语言和编译器

编译器和浮点运算理论的相互影响在 Farnum [1988] 中有所讨论，本部分中的讨论大都出自该论文。

## 含混性

理想情况下，一种语言的定义应该对该语言的语义做出足够精确的规定，以便可以证明有关程序的陈述。尽管对于语言整数部分的定义通常可以满足上述要求，但涉及到浮点部分时，语言定义通常具有很大的模糊性。其中的原因可能出自于以下事实：许多语言设计人员认为无法证明有关浮点的任何陈述，因为其中存在舍入误差。如果是这样，那么前面几部分已经证明这一看法乃是谬论。此部分讨论语言定义中一些常见的模糊性，并给出相关的处理建议。

很明显，某些语言没有清楚地指定如果  $x$  是浮点变量（例如，具有值  $3.0/10.0$ ），则（例如） $10.0*x$  每次都必须得到相同值。以基于布朗模型的 Ada 为例，它似乎隐含着浮点运算只需满足布朗公理，这样表达式就可以具有多个可能值。以这种模糊的方式考虑浮点与 IEEE 模型形成了鲜明的对比，对于后者，每个浮点运算的结果都是精确定义的。在 IEEE 模型中，我们可以证明  $(3.0/10.0)*10.0$  计算为 3（定理 7）。在布朗模型中，我们却做不到这一点。

大多数语言定义中的另一种含混性与出现上溢、下溢以及其它异常时所发生的情况有关。IEEE 标准可以准确地指定异常的行为，因此将该标准作为模型的语言可以避免有关这一点的任何含混性。

另一模糊性与对圆括弧的解释有关。由于存在舍入误差，代数学的结合律不一定适用于浮点数。例如，当  $x = 10^{30}$ ， $y = -10^{30}$  且  $z = 1$  时，表达式  $(x+y)+z$  的结果与  $x+(y+z)$  的结果完全不同（在前一种情况下结果为 1，在后一种情况下结果是 0）。应该说，对于保留圆括弧的重要性怎么强调也不过分。定理 3、4 和 6 中提供的算法都取决于它。例如，在定理 6 中，如果不使用圆括号，则公式  $x_h = mx - (mx - x)$  将简化为  $x_h = x$ ，因此破坏了整个算法。对于浮点运算，那些不要求先计算圆括弧中内容的语言定义是无用的。

子表达式计算在许多语言中没有精确定义。假定  $ds$  是双精度，但是  $x$  和  $y$  是单精度。那么，在表达式  $ds + x*y$  中，乘积是按单精度还是按双精度计算？另一个示例：在  $x + m/n$ （其中  $m$  和  $n$  是整数）中，除法是整数运算还是浮点运算？有两种方法可以处理这类问题，其中的任意一种都不完全令人满意。第一种方法是，要求表达式中的所有变量具有相同的类型。这是最简单的解决方法，但是它有一些缺点。首先，具有子范围类型的语言（如 Pascal）允许混合子范围变量和整型变量，因此禁止混合单精度变量和双精度变量就令人觉得有点奇怪了。另一个问题与常数有关。在表达式  $0.1*x$  中，大多数语言将 0.1 解释为单精度常数。现在，假定程序员决定将所有浮点变量的声明从单精度更改为双精度。如果仍然将 0.1 视为单精度常数，则将出现编译时错误。程序员将必须搜寻到每个浮点常数并对其进行更改。

第二种方法是，允许使用混合表达式，在这种情况下必须提供用于子表达式计算的规则。有许多指导性示例。C 的最初定义要求按双精度计算每个浮点表达式 [Kernighan 和 Ritchie 1978]。这将导致出现与此部分开始处提供的示例类似的异常。表达式  $3.0/7.0$  是按双精度计算的，但是，如果  $q$  是一个单精度变量，那么在存储时商将舍入为单精度。由于  $3/7$  是一个循环二进制分数，因此它的按双精度计算的值与它以单精度存储的值是不同的。这样，比较  $q = 3/7$  将失败。这个例子表明按最高可用精度计算每个表达式的值并不是一个好规则。

另一个指导性示例是内积。如果内积有数千项，则总和中的舍入误差可能会变得相当大。减少此舍入误差的一种方法是按双精度求和，这一点将在“优化器”（在第 205 页上）部分中进行更详细的讨论。如果  $d$  是一个双精度变量，且  $x[]$  和  $y[]$  是单精度数组，则内积循环形如  $d = d + x[i]*y[i]$ 。如果乘法是按单精度进行的，那么会失去双精度累积的许多优点，因为就在增加到双精度变量之前乘积被按照单精度进行截断。

涉及前面两个示例的规则是，按照表达式中出现的任何变量的最高精度计算该表达式。那么， $q = 3.0/7.0$  将完全按单精度计算，<sup>1</sup> 并具有布尔值 `true`，而  $d = d + x[i]*y[i]$  将按双精度计算，获得了双精度累积的全部优点。但是，此规则过于简单，无法完全包括所有情况。如果  $dx$  和  $dy$  是双精度变量，则表达式  $y = x + \text{single}(dx - dy)$  包含一个双精度变量，但是按双精度进行求和将是无意义的，因为这两个操作数都是单精度数，结果也是单精度数。

下面介绍更复杂的子表达式计算规则。首先，为每个运算指定试验性精度，该精度是其操作数的最高精度。这种指定必须按表达式树中从叶到根的顺序执行。然后从根到叶执行第二遍。这一次，为每个运算指定试验性精度和其父运算所需精度的最大值。在  $q = 3.0/7.0$  情况下，每个叶都是单精度的，因此所有运算都是按单精度执行的。在  $d = d + x[i]*y[i]$  情况下，乘法运算的试验性精度是单精度，但是在第二遍中它被提升到双精度，因为其父运算需要双精度操作数。而在  $y = x + \text{single}(dx - dy)$  中，加法运算是按单精度执行的。Farnum [1988] 提供的证据表明实现此算法并不困难。

此规则的缺点是：子表达式的计算取决于它被嵌入的表达式。这会导致一些不良后果。例如，假定您正在调试程序并希望知道子表达式的值。您不能简单地子表达式键入调试器，并要求计算它，因为在程序中子表达式的值取决于它被嵌入的表达式。关于子表达式，最后要注意的一点是：由于将十进制常数转换为二进制是一个运算，因此计算规则还影响对十进制常数的解释。对于不能以二进制精确表示的常数（如 0.1），这一点尤其重要。

当一种语言将求幂作为其内置运算之一时，将出现另一种潜在模糊性。与基本算术运算不同，求幂运算的值并不总是显而易见的 [Kahan 和 Coonen 1982]。如果 `**` 是求幂运算符，则  $(-3)**3$  一定具有值 -27。但是， $(-3.0)**3.0$  是有疑问的。如果 `**` 运算符会判断算式是否是整数幂，那么它会将  $(-3.0)**3.0$  按  $-3.0^3 = -27$  计算。另一方面，如果使用公式  $xy = e^{y \log x}$  来计算参数为实数的 `**` 运算，则结果可能是 NaN（当  $x < 0$  时，使用  $\log(x) = \text{NaN}$  的自然定义），这取决于对数函数。但是，如果使用 FORTRAN `CLOG` 函数，则结果将是 -27，因为 ANSI FORTRAN 标准将 `CLOG(-3.0)` 定义为  $i\pi + \log 3$  [ANSI 1978]。程序设计语言 Ada 通过仅为整数幂定义求幂运算避免了这一问题，而 ANSI FORTRAN 禁止对负数进行实数幂操作。

事实上，FORTRAN 标准规定

禁止任何其结果在数学上没有定义的算术运算……

---

1. 这假定有以下常用约定：3.0 是一个单精度常数，而 3.0D0 是一个双精度常数。

不过，随着 IEEE 标准引入  $\pm\infty$ ，在数学上没有定义的含义不再是完全明确的。一种定义可能是使用“无穷大”（在第 192 页上）部分中所示的方法。例如，要确定  $a^b$  的值，应考虑非常值解析函数  $f$  和  $g$ ，它们具有以下属性：当  $x \rightarrow 0$  时，存在  $f(x) \rightarrow a$  且  $g(x) \rightarrow b$ 。如果  $f(x)^{g(x)}$  总是接近同一极限，则该极限即是  $a^b$  的值。此定义将设置  $2^\infty = \infty$ ，这看来是相当合理的。在  $1.0^\infty$  的情况下，当  $f(x) = 1$  且  $g(x) = 1/x$  时，极限接近 1，但是当  $f(x) = 1 - x$  且  $g(x) = 1/x$  时，极限是  $e^{-1}$ 。因此  $1.0^\infty$  应该是一个 NaN。在  $0^0$  的情况下， $f(x)^{g(x)} = e^{g(x)\log f(x)}$ 。因为  $f$  和  $g$  都是可解析的并在 0 处具有值 0，所以  $f(x) = a_1x^1 + a_2x^2 + \dots$  且  $g(x) = b_1x^1 + b_2x^2 + \dots$ 。这样， $\lim_{x \rightarrow 0} g(x) \log f(x) = \lim_{x \rightarrow 0} x \log(x(a_1 + a_2x + \dots)) = \lim_{x \rightarrow 0} x \log(a_1x) = 0$ 。因此，对于  $f$  和  $g$ ， $f(x)^{g(x)} \rightarrow e^0 = 1$ ，这意味着  $0^0 = 1$ 。<sup>12</sup> 使用此定义将明确地定义所有参数的指数函数，尤其是将  $(-3.0) ** 3.0$  定义为 -27。

## IEEE 标准

“IEEE 标准”（在第 186 页上）部分讨论了 IEEE 标准的许多特性。但是，IEEE 标准没有说明如何通过程序设计语言使用这些特性。因而，在支持该标准的浮点硬件与程序设计语言（如 C、Pascal 或 FORTRAN）之间常常存在着不匹配。某些 IEEE 功能可以通过子例程调用库获得。例如，IEEE 标准要求精确舍入平方根，而平方根函数通常是直接在硬件中实现的。通过平方根例程库可以轻松地获得此功能。但是，该标准的其它方面不能通过子例程来轻松实现。例如，大多数计算机语言最多指定两种浮点类型，而 IEEE 标准具有四种不同精度（尽管建议的配置是单精度加上单精度扩展或者单精度、双精度和双精度扩展）。另一示例是无穷大。表示 饱 的常量可以由子例程提供。但是这样一来，它们可能无法用在需要常量表达式的位置上，例如用来初始化一个常量。

更微妙的情况是操纵与计算关联的状态，该状态由舍入模式、陷阱启用位、陷阱处理程序和异常标志组成。一种方法是提供子例程来完成状态的读取和写入。此外，能够以原子方式设置新值并返回旧值的单次调用通常很有用。如“标志”（在第 198 页上）部分中的示例所示，修改 IEEE 状态的一种常见模式是仅在块或子例程的作用域内更改它。这样，查找块的每个出口并确保状态已恢复是程序员的责任。如果语言支持在块的作用域内精确设置状态，那将是很有用的。Modula-3 是一种为陷阱处理程序实现此思想的语言 [Nelson 1991]。

在一种语言中实施 IEEE 标准时，许多次要事项还需考虑。因为对于所有  $x$   $x - x = +0$ ，所以  $^3(+0) - (+0) = +0$ 。但是， $-(+0) = -0$ ，因此不应该将  $-x$  定义为  $0 - x$ 。NaN 的引入可能是令人迷惑的，因为 NaN 永远不等于其它任何数（包括另一个 NaN），因此  $x = x$  不再总是正确的。事实上，如果没有提供 IEEE 推荐的函数 Isnan，则表达式  $x \neq x$  是测试 NaN 的最简单方法。此外，NaN 不能与任何其它数一起排序，因此不能将  $x \leq y$  定义为  $\neq x > y$ 。由于 NaN 的引入导致浮点数变成部分排序的，因此使用返回  $<$ 、 $=$ 、 $>$  或 *unordered* 之一的 compare 函数能让程序员更轻松地处理比较操作。

- 
1. 结论  $0^0 = 1$  取决于以下限制： $f$  是非常值函数。如果去掉此限制，那么允许  $f$  是恒等于 0 的函数就会将 0 作为  $\lim_{x \rightarrow 0} f(x)^{g(x)}$  的可能值，因此必须将  $0^0$  定义为一个 NaN。
  2. 在  $0^0$  的情况下，可以提出似乎合理的论点，但令人信服的论点则包括在“Concrete Mathematics”（作者：Graham、Knuth 和 Patashnik）中，该论点是：要使二项式定理成立，必须满足  $0^0 = 1$ 。—编辑
  3. 除非舍入模式是向  $-\infty$  舍入（在这种情况下， $x - x = -0$ ）。

虽然 IEEE 标准规定如果任一操作数是 NaN 则基本浮点运算返回 NaN，但这并不总是复合运算的最佳定义。例如，当计算绘制曲线图要使用的适当比例因子时，必须计算一组值的最大值。在这种情况下，最大值运算简单地忽略 NaN 是有意义的。

最后，舍入也可能引发问题。IEEE 标准非常精确地定义了舍入，而且它依赖于舍入模式的当前值。有时，这与类型转换中隐式舍入的定义或语言中的显式 `round` 函数存在冲突。也就是说，希望使用 IEEE 舍入的程序无法使用自然语言的基本要素，反过来，语言基本要素无法在数量不断增加的 IEEE 计算机上进行实施。

## 优化器

编译器教科书往往忽略浮点这一主题。例如，Aho 等 [1986] 提到将  $x/2.0$  替换为  $x*0.5$ ，使读者认为  $x/10.0$  应该替换为  $0.1*x$ 。但是，在二进制计算机上这两个表达式并不具有相同的语义，因为  $0.1$  无法以二进制精确表示。此教科书还建议将  $x*y-x*z$  替换为  $x*(y-z)$ ，而我们已经看到在  $y \approx z$  时这两个表达式可以具有相当不同的值。尽管其中指出优化器不应违反语言定义，从而对“可以使用任意代数恒等式优化代码”这种说法做了限制，但是它给人们留下了浮点语义并不太重要的印象。不管语言标准是否指定必须先计算圆括弧中的内容， $(x+y)+z$  都可以具有与  $x+(y+z)$  完全不同的结果，如上所述。存在一个与保留圆括弧密切相关的问题，如下代码所示：

```
eps = 1;
do eps = 0.5*eps; while (eps + 1 > 1);
```

此代码旨在估算出计算机厄普西隆 ( $\epsilon$ )。如果进行优化的编译器注意到  $eps + 1 > 1 \Leftrightarrow eps > 0$ ，程序将被完全更改。它不是计算最小数  $x$ ，以使  $1 \oplus x$  仍然大于  $x$  ( $x \approx e \approx \beta^{-p}$ )，而是计算最大数  $x$ ，使  $x/2$  舍入为  $0$  ( $x \approx \beta^{e_{\min}}$ )。避免这种“优化”是非常重要的，我们可以再举一种很有用的算法作例子，来说明这种优化有哪些危害。

许多问题（如数值积分和微分方程的数值解）涉及计算多个项的和。因为每个加法运算都有可能引入大至  $.5$  ulp 的误差，所以产生数千项的求和会具有相当大的舍入误差。纠正这一点的简单方法是将部分被加数存储在双精度变量中，并使用双精度执行每个加法运算。如果计算是使用单精度进行的，则在大多数计算机系统上按双精度执行求和是很容易的。但是，如果已经是按双精度进行计算的，则使精度加倍就不那么简单了。有时建议的一种方法是对数值进行排序，然后按照从最小到最大的顺序将其相加。但是，有一种能大大提高求和准确度，而且效率高得多的方法，即

## 定理 8 (Kahan 求和公式)

假设  $\sum_{j=1}^N x_j$  是使用以下算法计算的

```
S = X[1];
C = 0;
for j = 2 to N {
    Y = X[j] - C;
    T = S + Y;
    C = (T - S) - Y;
    S = T;
}
```

那么, 计算的和  $S$  等于  $\sum x_j(1 + \delta_j) + O(N\epsilon^2)\sum |x_j|$ , 其中  $(|\delta_j| \leq 2\epsilon)$ 。

使用朴素公式  $\sum x_j$ , 计算的和等于  $\sum x_j(1 + \delta_j)$ , 其中  $|\delta_j| < (n - j)e$ 。将此与 Kahan 求和公式中的误差相比较, 可以看出有显著的改进。每个被加数受到的扰动只有  $2e$ , 而不是简单公式中多至  $ne$  的扰动。有关详细信息, 请参见“求和中的误差”(在第 218 页上)。

认为浮点运算遵循代数法则的优化器将得出以下结论:  $C = [T-S] - Y = [(S+Y)-S] - Y = 0$ , 从而使该算法变得毫无用处。通过这些示例我们可以看到: 优化器在使用对数学上的实数成立的代数恒等式来优化涉及浮点变量的表达式时, 应当格外小心。

优化器更改浮点代码语义的另一种方法涉及到常数。在表达式  $1.0E-40*x$  中, 隐含了一个十进制到二进制的转换操作, 该操作将十进制数转换成了二进制常数。由于此常数无法以二进制精确表示, 所以会引发不精确异常。此外, 如果表达式是按单精度计算的, 则下溢标志应被设置。由于常数是不精确的, 所以到二进制的精确转换取决于 IEEE 舍入模式的当前值。这样, 在编译时将  $1.0E-40$  转换为二进制数的优化器将更改程序的语义。但是, 可以按最低可用精度精确表示的常数 (如 27.5) 能够在编译时安全地进行转换, 因为它们始终是精确的, 不会引发任何异常且不受舍入模式影响。希望在编译时进行转换的常数应该使用常量声明, 例如 `const pi = 3.14159265`。

关于优化可以更改浮点语义的另一示例是公共子表达式的消除, 如以下代码所示

```
C = A*B;
RndMode = Up
D = A*B;
```

虽然  $A*B$  可能看上去是一个公共子表达式, 但实际上并不是, 因为两个计算位置的舍入模式有所不同。最后三个示例:  $x=x$  不能由布尔常量 `true` 替换, 因为当  $x$  是一个 NaN 时前者将失败; 当  $x = +0$  时,  $-x = 0 - x$  将失败; 而且,  $x < y$  并不与  $x \geq y$  相反, 因为 NaN 既不大于也不小于普通浮点数。

尽管存在这些示例，我们还是可以对浮点代码进行一些有用的优化。首先，存在对浮点数有效的代数恒等式。IEEE 运算中的一些示例是： $x + y = y + x$ 、 $2 \times x = x + x$ 、 $1 \times x = x$  和  $0.5 \times x = x/2$ 。但是，即使是这些简单的恒等式也会在一些计算机（如 CDC 和 Cray 超级计算机）上失败。指令调度和内联过程替换是另外两种可能有用的优化。<sup>1</sup>

作为最后一个示例，假设表达式  $dx = x*y$ ，其中  $x$  和  $y$  是单精度变量， $dx$  是双精度变量。如果计算机上有相应的指令，能够将两个单精度数相乘并产生双精度数的结果，那么可以将  $dx = x*y$  映射到该指令，而不必将其编译为先将操作数转换为双精度数再将双精度数与双精度数相乘的指令序列。

一些编译器编写人员认为不得将  $(x + y) + z$  转换为  $x + (y + z)$  的限制与他们无关，只有使用不可移植方法的程序员才关心此限制。也许他们认为浮点数与实数非常类似，应该遵循与实数相同的定律。实数语义的问题是它们的实现成本极高。将两个  $n$  位数相乘时，积将具有  $2n$  位。将两个指数相差很大的  $n$  位数相加时，和中的位数等于  $n +$  指数间的差值。和的位数可能达到  $(e^{\max} - e^{\min}) + n$  位，或大约  $2 \cdot e^{\max} + n$  位。涉及数千次运算的算法（例如对线性方程组求解）会很快进入对具有许多有效位的数进行运算的状态，使得运算速度慢得令人难以忍受。库函数（例如  $\sin$  和  $\cos$ ）的实现甚至更困难，因为这些超越函数的值不是有理数。精确的整数运算通常由 lisp 系统提供，对于解决某些问题是很方便的。但是，精确的浮点运算几乎是没有用的。

事实是，存在利用  $(x + y) + z \neq x + (y + z)$  事实的有用算法（如 Kahan 求和公式），而且只要界限

$$a \oplus b = (a + b)(1 + \delta)$$

成立（以及  $-$ 、 $\times$  和  $/$  的类似界限），这些算法就是适用的。由于这些界限对于几乎所有商业硬件都是成立的，所以数值程序员忽略此类算法的做法是不明智的，而如果编译器编写人员伪称浮点变量具有实数语义，因而破坏了这些算法，那将是很不负责任的。

## 异常处理

到目前为止所讨论的主题主要涉及准确性和精度的系统含意。陷阱处理程序也引发了一些值得关注的系统问题。IEEE 标准强烈建议用户应该能够为五类异常中的每一类都指定陷阱处理程序，“陷阱处理程序”（在第 197 页上）部分给出了用户定义的陷阱处理程序的一些应用。在出现无效运算异常和除以零异常的情况下，应该为处理程序提供操作数，否则应该提供精确舍入的结果。根据所使用的程序设计语言，陷阱处理程序或许还能够访问程序中的其它变量。对于所有异常，异常处理程序必须能够识别正在执行什么运算以及该运算的目标精度是什么。

---

1. VAX 上的 VMS 数学库使用弱形式的内联过程替换，因为它们使用转向子例程调用的廉价跳转，而不是使用快速的 CALLS 和 CALLG 指令。

IEEE 标准假定运算在概念上是串行的，当发生中断时，识别出运算及其操作数是可能的。在采用流水线技术或具有多个运算单元的计算机上，当出现异常时，只是让陷阱处理程序检查程序计数器可能是不够的。此时可能需要提供硬件支持来精确识别出哪个运算被捕获。

另一问题如以下程序片段所示。

```
x = y*z;  
z = x*w;  
a = b + c;  
d = a/x;
```

假定第二个乘法运算引发一个异常，陷阱处理程序希望使用  $a$  的值。在可以并行执行加法运算和乘法运算的硬件上，优化器有可能将加法运算提到第二个乘法运算之前，以便加法运算可以与第一个乘法运算并行进行。这样，当第二个乘法运算被捕获时， $a = b + c$  已经执行，因而有可能更改了  $a$  的结果。编译器应当考虑到此类优化；否则，因为每个浮点运算都有可能被捕获，所以这实际上是禁止了几乎所有的指令调度优化。通过禁止陷阱处理程序直接访问程序的任何变量，可以避免这一问题。我们可以将操作数或结果作为参数提供给处理程序。

但是，仍然存在问题。在以下片段中

```
x = y*z;  
z = a + b;
```

这两个指令也许能够并行执行。如果乘法运算被捕获，则其参数  $z$  可能已经被加法运算覆盖，尤其是因为加法运算的速度通常比乘法运算快。支持 IEEE 标准的计算机系统必须提供某种保存  $z$  值的方法，要么是在硬件中，要么让编译器避免这样的情况。

W. Kahan 曾提议使用 *预替换* 代替陷阱处理程序以避免这些问题。在此方法中，用户指定一个异常以及希望在出现该异常时用作结果的值。举例来说，假定在用于计算  $(\sin x)/x$  的代码中，用户断定  $x = 0$  如此少见以至于避免测试  $x = 0$  将提高性能，因此改为在出现  $0/0$  陷阱时处理这种情况。当使用 IEEE 陷阱处理程序时，用户将编写返回值为 1 的处理程序并在计算  $\sin x/x$  之前进行安装。使用预替换时，用户可指定发生无效运算时应该使用值 1。Kahan 将此称为预替换，是因为要使用的值必须在出现异常之前指定。而当使用陷阱处理程序时，要返回的值可以在出现陷阱时进行计算。

预替换的优点是它具有简单的硬件实现。<sup>1</sup> 一旦确定异常的类型，就可以使用它来查表确定所需的运算结果。虽然预替换具有一些吸引人的属性，但是 IEEE 标准的广泛使用使该方法不大可能被硬件制造商广泛执行。

---

1. 预替换的困难在于它需要直接硬件实现或可持续的浮点陷阱（如果在软件中实现）。—编辑



---

## 详细资料

在本文中已经作出了许多有关浮点运算属性的断言。现在，我们继续说明浮点并非是神秘的魔术，而是一个可以在数学上检验其断言的简单易懂的主题。此说明分为三个部分。第一部分介绍误差分析，并提供“舍入误差”（在第 174 页上）部分的详细资料。第二部分探讨二进制到十进制的转换，对“IEEE 标准”（在第 186 页上）部分进行一些补充。第三部分讨论 Kahan 求和公式，它曾在“系统方面”（在第 199 页上）部分中用作一个示例。

## 舍入误差

在讨论舍入误差时，已经指出单个保护数位足以保证加法运算和减法运算将始终是准确的（定理 2）。现在我们继续检验这一事实。定理 2 分为两个部分，一个部分用于减法运算，另一部分用于加法运算。用于减法运算的部分是

### 定理 9

如果  $x$  和  $y$  是正的浮点数，其格式为具有参数  $\beta$  和  $p$ ，而且减法运算是使用  $p + 1$  位（也就是使用一个保护数位）进行的，则结果中的相对舍入误差小于

$$\left(\frac{\beta}{2} + 1\right)\beta^{-p} = \left(1 + \frac{2}{\beta}\right)e \leq 2e$$

### 证明

有必要的话， $x$  和  $y$  可以互换，因此不妨认为  $x > y$ 。同样，不妨将  $x$  和  $y$  按比例缩放，以便  $x$  可以表示成  $x_0.x_1 \dots x_{p-1} \times \beta^0$ 。如果将  $y$  表示为  $y_0.y_1 \dots y_{p-1}$ ，则差值是精确的。如果将  $y$  表示为  $0.y_1 \dots y_p$ ，则保护数位确保计算的差值将是舍入为浮点数的精确差值，因此舍入误差最大为  $e$ 。对于一般情形，令  $y = 0.0 \dots 0y_{k+1} \dots y_{k+p}$  且  $\bar{y}$  是截断到  $p + 1$  位的  $y$ 。那么

$$y - \bar{y} < (\beta - 1)(\beta^{p-1} + \beta^{p-2} + \dots + \beta^{p-k}) \quad (15)$$

按照保护数位的定义， $x - y$  的计算值是舍入为浮点数的  $x - \bar{y}$ ，即  $(x - \bar{y}) + \delta$ ，其中舍入误差  $\delta$  满足以下条件

$$|\delta| \leq (\beta/2)\beta^{-p} \quad (16)$$

精确差值是  $x - y$ ，因此误差是  $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y + \delta$ 。存在三种情况。如果  $x - y \geq 1$ ，则相对误差的界限是

$$\frac{y - \bar{y} + \delta}{1} \leq \beta^{-p} [(\beta - 1)(\beta^{-1} + \dots + \beta^{-k}) + \beta/2] < \beta^{-p}(1 + \beta/2) \quad (17)$$

第二，如果  $x - \bar{y} < 1$ ，则  $\delta = 0$ 。因为  $x - y$  最小可以是

$$1.0 - 0.\left(\overbrace{0 \dots 0}^k\right)\left(\overbrace{\rho \dots \rho}^p\right) > (\beta - 1)(\beta^{-1} + \dots + \beta^{-k}), \text{ 其中 } \rho = \beta - 1,$$

在这种情况下，相对误差的界限是

$$\frac{y - \bar{y} + \delta}{(\beta - 1)(\beta^{-1} + \dots + \beta^{-k})} < \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \dots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \dots + \beta^{-k})} = \beta^{-p} \quad (18)$$

最后一种情况是： $x - y < 1$  但  $x - \bar{y} \geq 1$ 。发生这种情况的唯一条件是  $x - \bar{y} = 1$ ，在这种情况下  $\delta = 0$ 。但是，如果  $\delta = 0$ ，则 (18) 是适用的，因此相对误差的界限同样是  $\beta^{-p} < \beta^{-p}(1 + \beta/2)$ 。■

当  $\beta = 2$  时，界限正好是  $2e$ ，此界限是在  $p \rightarrow \infty$  的极限中为  $x = 1 + 2^{2-p}$  和  $y = 2^{1-p} - 2^{1-2p}$  实现的。将符号相同的数相加时，不使用保护数位也能得到良好的精确度，如以下结果所示。

## 定理 10

如果  $x \geq 0$  且  $y \geq 0$ ，则计算  $x + y$  时的相对误差最大为  $2e$ ，即使未使用保护数位也是如此。

## 证明

使用  $k$  个保护数位的加法运算的算法与用于减法运算的算法类似。如果  $x \geq y$ ，则对  $y$  进行右移位，直到  $x$  和  $y$  的小数点对齐。舍弃移过  $p + k$  位置的任何位。精确计算这两个  $p + k$  位数的和。然后舍入为  $p$  位。

我们将检验不使用保护数位时该定理是否成立；一般的情况是类似的。不失一般性，假设  $x \geq y \geq 0$  并将  $x$  乘以某一因子使其具有  $d.dd\dots d \times \beta^0$  的形式。首先，假定没有进位。那么，从  $y$  的末尾移掉的位所具有的值小于  $\beta^{-p+1}$ ，且和至少为 1，因此相对误差小于  $\beta^{-p+1}/1 = 2e$ 。如果有进位，则必须将移位误差增加到  $\frac{1}{2}\beta^{-p+2}$  的舍入误差中。

和至少为  $\beta$ ，因此相对误差小于

$$\left(\beta^{-p} + 1 + \frac{1}{2}\beta^{-p} + 2\right) / \beta = (1 + \beta/2)\beta^{-p} \leq 2\varepsilon$$

显而易见，结合这两个定理可以得出定理 2。定理 2 给出了执行一个运算的相对误差。比较  $x^2 \cdot y^2$  和  $(x + y)(x - y)$  的舍入误差要求知道乘法运算的相对误差。 $x \ominus y$  的相对误差是  $\delta_1 = [(x \ominus y) - (x - y)] / (x - y)$ ，它满足  $|\delta_1| \leq 2e$ 。或者，以另一种方式书写

$$x \ominus y = (x - y)(1 + \delta_1), \quad |\delta_1| \leq 2e \quad (19)$$

类似地

$$x \oplus y = (x + y)(1 + \delta_2), \quad |\delta_2| \leq 2e \quad (20)$$

假定乘法是通过计算精确乘积再进行舍入来执行的，相对误差最大为  $.5 \text{ ulp}$ ，那么，对于任意浮点数  $u$  和  $v$

$$u \otimes v = uv(1 + \delta_3), \quad |\delta_3| \leq e \quad (21)$$

将这三个方程放在一起（假设  $u = x \ominus y$  且  $v = x \oplus y$ ），会得出

$$(x \ominus y) \otimes (x \oplus y) = (x - y)(1 + \delta_1)(x + y)(1 + \delta_2)(1 + \delta_3) \quad (22)$$

因此，在计算  $(x - y)(x + y)$  时引起的相对误差是

$$\frac{(x - y) \ominus (x + y) - (x^2 - y^2)}{(x^2 - y^2)} = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) - 1 \quad (23)$$

此相对误差等于  $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3 + \delta_1\delta_2\delta_3$ ，其边界是  $5\varepsilon + 8\varepsilon^2$ 。换句话说，最大相对误差大约是舍入误差的 5 倍（因为  $e$  是一个很小的数， $e^2$  几乎可以忽略）。

对  $(x \otimes x) \ominus (y \otimes y)$  的类似分析无法得出很小的相对误差值，因为将  $x$  和  $y$  的两个接近值代入  $x^2 \cdot y^2$  时，相对误差通常是相当大的。查看它的另一方法是尝试并重复曾用于  $(x \ominus y) \otimes (x \oplus y)$  的分析方法，将会得出

$$\begin{aligned} (x \otimes x) \ominus (y \otimes y) &= [x^2(1 + \delta_1) - y^2(1 + \delta_2)](1 + \delta_3) \\ &= ((x^2 - y^2)(1 + \delta_1) + (\delta_1 - \delta_2)y^2)(1 + \delta_3) \end{aligned}$$

当  $x$  和  $y$  接近时，误差项  $(\delta_1 - \delta_2)y^2$  可以与结果  $x^2 - y^2$  一样大。这些计算证明我们的以下断言是正确的： $(x - y)(x + y)$  比  $x^2 - y^2$  更精确。

接下来，我们分析三角形面积的公式。为了估算在使用 (7) 计算时出现的最大误差，将需要以下定理。

## 定理 11

如果减法运算使用了保护数位且操作数满足  $y/2 \leq x \leq 2y$ ，则  $x - y$  是精确计算。

### 证明

请注意，如果  $x$  和  $y$  的指数相同，则  $x \ominus y$  一定是精确的。否则，根据定理中的条件，指数最多可以相差 1。如有必要， $x$  和  $y$  可以按比例缩放并互换，从而使  $0 \leq y \leq x$ ，并将  $x$  表示为  $x_0.x_1 \dots x_{p-1}$ ，将  $y$  表示为  $0.y_1 \dots y_p$ 。那么，用于计算  $x \ominus y$  的算法将精确计算  $x - y$  并舍入到一个浮点数。如果差的形式为  $0.d_1 \dots d_p$ ，则差值的长度已经是  $p$  位，因此不必进行舍入。由于  $x \leq 2y$ ， $x - y \leq y$ ，且  $y$  的形式是  $0.d_1 \dots d_p$ ，因此  $x - y$  也满足这样的形式。■

当  $\beta > 2$  时，定理 11 的假设不能由  $y/\beta \leq x \leq \beta y$  替换；更严格的条件  $y/2 \leq x \leq 2y$  仍然是必需的。在定理 10 被证明之后，随即对  $(x - y)(x + y)$  中的误差进行了分析，其中利用了加法和减法两种基本运算的相对误差很小（即方程 (19) 和 (20)）这一事实。这是一种最常见的误差分析。但是，就如同以下证明过程所显示的那样，分析公式 (7) 还需要其它内容，即定理 11。

## 定理 12

如果减法运算使用保护数位，而且  $a$ 、 $b$  和  $c$  是三角形的边 ( $a \geq b \geq c$ )，则计算  $(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$  时的相对误差最大为  $16\epsilon$ ，条件是  $\epsilon < .005$ 。

### 证明

让我们逐一检查各项因子。依据定理 10， $b \oplus c = (b + c)(1 + \delta_1)$ ，其中  $\delta_1$  是相对误差， $|\delta_1| \leq 2\epsilon$ 。所以，第一个因子的值是

$$(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1))(1 + \delta_2),$$

因此

$$\begin{aligned} (a + b + c)(1 - 2\epsilon)^2 &\leq [a + (b + c)(1 - 2\epsilon)] \cdot (1 - 2\epsilon) \\ &\leq a \oplus (b \oplus c) \\ &\leq [a + (b + c)(1 + 2\epsilon)](1 + 2\epsilon) \\ &\leq (a + b + c)(1 + 2\epsilon)^2 \end{aligned}$$

这意味着存在一个  $\eta_1$  满足

$$(a \oplus (b \oplus c)) = (a + b + c) (1 + \eta_1)^2, \quad |\eta_1| \leq 2\varepsilon. \quad (24)$$

下一项涉及  $c$  和  $a \ominus b$  潜在的恶性相减，因为  $a \ominus b$  可能具有舍入误差。由于  $a$ 、 $b$  和  $c$  是三角形的边，所以有  $a \leq b + c$ ，将此与  $c \leq b \leq a$  排列次序结合在一起就可以得出  $a \leq b + c \leq 2b \leq 2a$ 。因此， $a - b$  满足定理 11 的条件。这意味着  $a - b = a \ominus b$  是精确的，因此  $c \ominus (a - b)$  是一个无害的减法，可以依据定理 9 进行估算，估算结果为

$$(c \ominus (a \ominus b)) = (c - (a - b)) (1 + \eta_2), \quad |\eta_2| \leq 2\varepsilon \quad (25)$$

第三项是两个精确正量的和，因此有

$$(c \oplus (a \ominus b)) = (c + (a - b)) (1 + \eta_3), \quad |\eta_3| \leq 2\varepsilon \quad (26)$$

最后一项是

$$(a \oplus (b \ominus c)) = (a + (b - c)) (1 + \eta_4)^2, \quad |\eta_4| \leq 2\varepsilon, \quad (27)$$

同时使用了定理 9 和定理 10。如果假定乘法运算精确舍入，使得  $x \otimes y = xy(1 + \zeta)$  且  $|\zeta| \leq \varepsilon$ ，那么将 (24)、(25)、(26) 和 (27) 结合在一起就可以得出

$$\begin{aligned} & (a \oplus (b \oplus c)) (c \ominus (a \ominus b)) (c \oplus (a \ominus b)) (a \oplus (b \ominus c)) \\ & \leq (a + (b + c)) (c - (a - b)) (c + (a - b)) (a + (b - c)) E \end{aligned}$$

其中

$$E = (1 + \eta_1)^2 (1 + \eta_2) (1 + \eta_3) (1 + \eta_4)^2 (1 + \zeta_1)(1 + \zeta_2) (1 + \zeta_3)$$

$E$  的一个上界是  $(1 + 2\varepsilon)^6(1 + \varepsilon)^3$ ，其展开结果是  $1 + 15\varepsilon + O(\varepsilon^2)$ 。有些作者干脆忽略了  $O(\varepsilon^2)$  项，但其实说明这一项是很容易的。令  $(1 + 2\varepsilon)^6(1 + \varepsilon)^3 = 1 + 15\varepsilon + \varepsilon R(\varepsilon)$ ，其中  $R(\varepsilon)$  是一个具有正系数的  $\varepsilon$  的多项式，由此可知它是  $\varepsilon$  的递增函数。因为  $R(.005) = .505$ ，所以对于所有的  $\varepsilon < .005$ ， $R(\varepsilon) < 1$ ，由此可得  $E \leq (1 + 2\varepsilon)^6(1 + \varepsilon)^3 < 1 + 16\varepsilon$ 。求  $E$  的一个下界时，我们会注意到  $1 - 15\varepsilon - \varepsilon R(\varepsilon) < E$ ，因此当  $\varepsilon < .005$  时， $1 - 16\varepsilon < (1 - 2\varepsilon)^6(1 - \varepsilon)^3$ 。将这两个界限结合在一起可得  $1 - 16\varepsilon < E < 1 + 16\varepsilon$ 。因此，相对误差最大为  $16\varepsilon$ 。■

定理 12 清楚地显示公式 (7) 中不存在恶性抵消。因此，虽然不必显示公式 (7) 在数值上是稳定的，但是具有整个公式的界限是令人满意的，这正是“抵消”（在第 178 页上）中定理 3 给出的内容。

### 定理 3 的证明

假设

$$q = (a + (b + c)) (c - (a - b)) (c + (a - b)) (a + (b - c))$$

且

$$Q = (a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))$$

然后, 由定理 12 得  $Q = q(1 + \delta)$ , 且  $\delta \leq 16\varepsilon$ 。可以很容易地得到

$$1 - 0.52|\delta| \leq \sqrt{1 - |\delta|} \leq \sqrt{1 + |\delta|} \leq 1 + 0.52|\delta| \quad (28)$$

条件是  $\delta \leq .04/(.52)^2 \approx .15$ , 因为  $|\delta| \leq 16\varepsilon \leq 16(.005) = .08$ , 所以  $\delta$  确实满足此条件。因此

$$\sqrt{Q} = \sqrt{q(1 + \delta)} = \sqrt{q}(1 + \delta_1),$$

且  $|\delta_1| \leq .52|\delta| \leq 8.5\varepsilon$ 。如果将计算平方根时的误差控制在  $.5 \text{ ulp}$  内, 则计算  $\sqrt{Q}$  时的误差是  $(1 + \delta_1)(1 + \delta_2)$ , 其中  $|\delta_2| \leq \varepsilon$ 。如果  $\beta = 2$ , 则在除以 4 时不会产生进一步的误差。否则, 在做除法时还需要另一因子  $1 + \delta_3$  ( $|\delta_3| \leq \varepsilon$ ), 通过使用证明定理 12 时的方法,  $(1 + \delta_1)(1 + \delta_2)(1 + \delta_3)$  的最终误差界限取决于  $1 + \delta_4$  ( $|\delta_4| \leq 11\varepsilon$ )。■

要使紧跟定理 4 陈述之后的启发式说明变得精确, 下一个定理正好说明  $\mu(x)$  近似于一个常数时的接近程度。

### 定理 13

如果  $\mu(x) = \ln(1 + x)/x$ , 则对于  $0 \leq x \leq \frac{3}{4}$ ,  $\frac{1}{2} \leq \mu(x) \leq 1$  且导数满足  $|\mu'(x)| \leq \frac{1}{2}$ 。

### 证明

请注意,  $\mu(x) = 1 - x/2 + x^2/3 - \dots$  是具有递减项的交错级数, 因此对于  $x \leq 1$ ,  $\mu(x) \geq 1 - x/2 \geq 1/2$ 。可以更容易地看出: 因为  $\mu$  的级数是交错的, 所以  $\mu(x) \leq 1$ 。 $\mu'(x)$  的泰勒级数也是交错的, 而且如果  $x \leq \frac{3}{4}$  具有递减项, 那么  $-\frac{1}{2} \leq \mu'(x) \leq -\frac{1}{2} + 2x/3$ , 或者  $-\frac{1}{2} \leq \mu'(x) \leq 0$ , 因此  $|\mu'(x)| \leq \frac{1}{2}$ 。■

## 定理 4 的证明

因为  $\ln$  的泰勒级数

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$$

是一个交错级数, 所以  $0 < x - \ln(1+x) < x^2/2$ , 按  $x$  逼近  $\ln(1+x)$  时产生的相对误差的界限是  $x/2$ 。如果  $1 \oplus x = 1$ , 则  $|x| < \varepsilon$ , 因此相对误差的界限是  $\varepsilon/2$ 。

当  $1 \oplus x \neq 1$  时, 定义  $\hat{x}$  满足  $1 \oplus x = 1 + \hat{x}$ 。又因为  $0 < x < 1$ , 所以  $(1 \oplus x) \ominus 1 = \hat{x}$ 。如果要将计算除法和取对数时的误差控制在  $\frac{1}{2}$  ulp 内, 则表达式  $\ln(1+x)/((1+x)-1)$  的计算值是

$$\frac{\ln(1 \oplus x)}{1 \oplus x \ominus 1} (1 + \delta_1) (1 + \delta_2) = \frac{\ln(1 + \hat{x})}{\hat{x}} (1 + \delta_1) (1 + \delta_2) = \mu(\hat{x}) (1 + \delta_1) (1 + \delta_2) \quad (29)$$

其中,  $|\delta_1| \leq \varepsilon$  且  $|\delta_2| \leq \varepsilon$ 。要估算  $\mu(\hat{x})$ , 请使用中值定理, 该定理指出

$$\mu(\hat{x}) - \mu(x) = (\hat{x} - x)\mu'(\xi) \quad (30)$$

其中  $\xi$  介于  $x$  和  $\hat{x}$  之间。根据  $\hat{x}$  的定义, 可以得出  $|\hat{x} - x| \leq \varepsilon$ , 将此与定理 13 结合在一起可以得出  $|\mu(\hat{x}) - \mu(x)| \leq \varepsilon/2$ , 或  $|\mu(\hat{x})/\mu(x) - 1| \leq \varepsilon/(2|\mu(x)|) \leq \varepsilon$ , 这意味着  $\mu(\hat{x}) = \mu(x) (1 + \delta_3)$ ,  $|\delta_3| \leq \varepsilon$ 。最后, 乘以  $x$  会引入  $\delta_4$ , 因此

$x \cdot \ln(1 \oplus x) / ((1 \oplus x) \ominus 1)$  的值

是

$$\frac{\ln(1+x)}{1+x-1} (1 + \delta_1) (1 + \delta_2) (1 + \delta_3) (1 + \delta_4), \quad |\delta_i| \leq$$

很容易得出: 如果  $\varepsilon < 0.1$ , 则

$$(1 + \delta_1) (1 + \delta_2) (1 + \delta_3) (1 + \delta_4) = 1 + \delta,$$

其中  $|\delta| \leq 5\varepsilon$ 。■

使用公式 (19)、(20) 和 (21) 进行误差分析的一个有趣示例出现在二次方程求根公式  $(-b \pm \sqrt{b^2 - 4ac})/2a$  中。“抵消” (在第 178 页上) 部分解释改写公式将如何消除  $\pm$  运算引起的潜在抵消。但是, 在计算  $d = b^2 - 4ac$  时还可能出现另一个抵消。通过对公式进行简单的重新排列不能消除该潜在抵消。大致说来, 当  $b^2 \approx 4ac$  时, 舍入误差最多可以影响使用二次方程求根公式计算的根中数位的一半。下面是一个非正式证明 (估算二次方程求根公式误差的另一种方法见于 Kahan [1972])。

如果  $b^2 \approx 4ac$ ，则舍入误差最多可以影响使用二次方程求根公式  $(-b \pm \sqrt{b^2 - 4ac})/2a$  计算的根中数位的一半。

证明：记  $(b \otimes b) \ominus (4a \otimes c) = (b^2(1 + \delta_1) - 4ac(1 + \delta_2))(1 + \delta_3)$ ，其中  $|\delta_i| \leq \varepsilon$ 。<sup>1</sup> 令  $d = b^2 - 4ac$ ，上式可改写为  $(d(1 + \delta_1) - 4ac(\delta_2 - \delta_1))(1 + \delta_3)$ 。为获得此误差的估计大小，忽略  $\delta_i$  中的二次项，此时绝对误差为  $d(\delta_1 + \delta_3) - 4ac\delta_4$ ，其中  $|\delta_4| = |\delta_1 - \delta_2| \leq 2\varepsilon$ 。因为

$d \ll 4ac$ ，所以可以忽略第一项  $d(\delta_1 + \delta_3)$ 。要估算第二项，可令  $ax^2 + bx + c = a(x - r_1)(x - r_2)$ ，因此  $ar_1r_2 = c$ 。因为  $b^2 \approx 4ac$ ，所以  $r_1 \approx r_2$ ，可得第二误差项为  $4ac\delta_4 \approx 4a^2r_1\delta_4^2$ 。这样， $\sqrt{d}$  的计算值是

$$\sqrt{d + 4a^2r_1^2\delta_4^2}$$

不等式

$$-q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q, \quad p \geq q >$$

显示

$$\sqrt{d + 4a^2r_1^2\delta_4^2} = \sqrt{d} + E,$$

其中

$$|E| \leq \sqrt{4a^2r_1^2|\delta_4|},$$

因此  $\sqrt{d}/2a$  中的绝对误差大约为  $r_1\sqrt{\delta_4}$ 。因为  $\delta_4 \approx \beta^{-p}$ ，所以  $\sqrt{\delta_4} \approx \beta^{-p/2}$ ，这样  $r_1\sqrt{\delta_4}$  的绝对误差就破坏了根  $r_1 \approx r_2$  的下半部分数位。换句话说，由于根的计算涉及计算  $(\sqrt{d})/(2a)$ ，并且此表达式在对应于  $r_i$  的一半低位的位置中不存在有效位，所以  $r_i$  的低位不是有效的。■

最后，我们转到定理 6 的证明。它基于以下事实，这些事实将在“定理 14 和定理 8”（在第 223 页上）部分中证明。

## 定理 14

假设  $0 < k < p$ ，并设  $m = \beta^k + 1$ ，同时假定浮点运算是精确舍入的。那么， $(m \otimes x) \ominus (m \otimes x \ominus x)$  与舍入到  $p - k$  有效位的  $x$  完全相等。更精确的说法是，舍入  $x$  的方法是取  $x$  的有效数字，可以设想为小数点正好在  $k$  个最低有效位的左侧并舍入为一个整数。

1. 在此非正式证明中，假定  $\beta = 2$ ，使得乘以 4 的结果是精确的，且不需要  $\delta_i$ 。



## 定理 6 的证明

依据定理 14,  $x_h$  是舍入到  $p - k = \lfloor p/2 \rfloor$  位的  $x$ 。如果没有进位, 则  $x_h$  当然可以用  $\lfloor p/2 \rfloor$  个有效位表示。假定存在进位。如果  $x = x_0 x_1 \dots x_{p-1} \times \beta^e$ , 则舍入将  $x_{p-k-1}$  加 1。可能有进位的唯一情况是  $x_{p-k-1} = \beta - 1$ , 但此时  $x_h$  的低位数字是  $1 + x_{p-k-1} = 0$ , 因此  $x_h$  同样可用  $\lfloor p/2 \rfloor$  位表示。

要处理  $x_l$ , 请按比例增减  $x$  使其成为满足  $\beta^{p-1} \leq x \leq \beta^p - 1$  的整数。假设  $x = \bar{x}_h + \bar{x}_l$ , 其中  $\bar{x}_h$  是  $x$  的  $p - k$  个高位数字,  $\bar{x}_l$  是  $k$  个低位数字。有三种情况需要考虑。如果  $\bar{x}_l < (\beta/2)\beta^{k-1}$ , 则将  $x$  舍入到  $p - k$  位的结果与截断相同, 而且  $x_h = \bar{x}_h$ ,  $x_l = \bar{x}_l$ 。因为  $\bar{x}_l$  最多有  $k$  位, 如果  $p$  是偶数, 则  $\bar{x}_l$  的位数最多为  $k = \lceil p/2 \rceil = \lfloor p/2 \rfloor$ 。否则,  $\beta = 2$  且  $\bar{x}_l < 2^{k-1}$  可以使用  $k - 1 \leq \lfloor p/2 \rfloor$  个有效位表示。第二种情况是在  $\bar{x}_l > (\beta/2)\beta^{k-1}$  时, 计算  $x_h$  涉及上舍入, 因此  $x_h = \bar{x}_h + \beta^k$ , 且  $x_l = x - x_h = x - \bar{x}_h - \beta^k = \bar{x}_l - \beta^k$ 。同样,  $\bar{x}_l$  最多有  $k$  位, 因此可以使用  $\lfloor p/2 \rfloor$  位表示。最后, 如果  $\bar{x}_l = (\beta/2)\beta^{k-1}$ , 则  $x_h = \bar{x}_h$  或  $\bar{x}_h + \beta^k$ , 这取决于是否存在上舍入。因此  $x_l$  是  $(\beta/2)\beta^{k-1}$  或  $(\beta/2)\beta^{k-1} - \beta^k = -\beta^k/2$ , 二者都是用 1 位表示的。■

定理 6 提供了一种将两个工作精度数的乘积精确表示为一个和的方式。有一个用于精确表示和的相应公式。如果  $|x| \geq |y|$ , 则  $x + y = (x \oplus y) + (x \ominus (x \oplus y)) \oplus y$  [Dekker 1971 ; Knuth 1981, 4.2.2 节中的定理 C]。但是, 在使用精确舍入运算时, 此公式仅对  $\beta = 2$  是正确的, 而对于  $\beta = 10$  是不正确的, 如示例  $x = .99998$ 、 $y = .99997$  所示。

## 二进制到十进制的转换

由于单精度有  $p = 24$ , 且  $2^{24} < 10^8$ , 所以您可能会认为将二进制数转换为 8 位十进制数将足以恢复原始二进制数。但是, 这是不正确的。

## 定理 15

当将二进制 IEEE 单精度数转换为最接近的八位十进制数时, 从十进制数唯一地恢复二进制数并不总是可能的。但是, 如果使用九位十进制数, 则将十进制数转换为最接近的二进制数将恢复原始浮点数。

## 证明

半开区间  $[10^3, 2^{10}) = [1000, 1024)$  中的二进制单精度数有 10 位在二进制小数点的左侧, 有 14 位在二进制小数点的右侧。因此, 在该区间中有  $(2^{10} - 10^3)2^{14} = 393,216$  个不同的二进制数。如果十进制数是使用 8 位表示的, 则在同一区间中有  $(2^{10} - 10^3)10^4 = 240,000$  个十进制数。240,000 个十进制数无法表示 393,216 个不同的二进制数。因此, 8 位十进制数不足以唯一表示每个单精度二进制数。

要说明 9 位是足够的, 只需说明二进制数之间的间隔始终大于十进制数之间的间隔就足够了。

这将确保对于每个十进制数  $N$ ，区间

$$\left[ N - \frac{1}{2} \text{ulp}, N + \frac{1}{2} \text{ulp} \right]$$

最多包含一个二进制数。这样，每个二进制数都舍入为唯一的十进制数，而十进制数又舍入为唯一的二进制数。

以区间  $[10^n, 10^{n+1}]$  为例来说明二进制数之间的间隔始终大于十进制数之间的间隔。在此区间上，连续十进制数之间的间隔是  $10^{(n+1)-9}$ 。在  $[10^n, 2^m]$  上（其中  $m$  是满足  $10^n < 2^m$  的最小整数），二进制数的间隔是  $2^{m-24}$ ，并且在该区间中此间隔变得越来越大。因此，只需证明  $10^{(n+1)-9} < 2^{m-24}$ 。而事实上，由于  $10^n < 2^m$ ，所以  $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^{m-24}$ 。■

应用于双精度的同一参数显示恢复双精度数需要 17 个十进制位。

二进制 - 十进制转换还提供了使用标志的另一示例。回想一下“精度”（在第 187 页上）部分中的内容：要从十进制扩展恢复二进制数，必须精确计算十进制到二进制的转换。该转换的方法是按单扩展精度将数量  $N$  和  $10^{p_l}$ （如果  $p < 13$ ，则二者都是精确的）相乘，然后舍入到单精度（如果  $p < 0$ ，则将它们相除；这两种情况是类似的）。当然，

$N \cdot 10^{p_l}$  的计算结果不可能是精确的；必须保持精确的是组合运算  $\text{round}(N \cdot 10^{p_l})$ ，其中舍入是从单扩展精度到单精度。要了解它为什么不可能是精确的，请举一个简单的例子： $\beta = 10$ ， $p = 2$ （对于单精度）或  $p = 3$ （对于单精度扩展）。如果乘积是 12.51，则这将作为单精度扩展乘法运算的一部分舍入为 12.5。舍入到单精度的结果为 12。但是该结果是不正确的，因为将乘积舍入到单精度的结果应该是 13。误差是双舍入导致的。

通过使用 IEEE 标志，可以避免双舍入，如下所示。保存不精确标志的当前值，然后将其重置。将舍入模式设置为“舍入为零”。然后，执行乘法运算  $N \cdot 10^{p_l}$ 。将不精确标志的新值存储在  $\text{ixflag}$  中，并恢复舍入模式和不精确标志。如果  $\text{ixflag}$  是 0，则  $N \cdot 10^{p_l}$  是精确的，因此  $\text{round}(N \cdot 10^{p_l})$  直到最后一位都是正确的。如果  $\text{ixflag}$  是 1，则某些数字被截去了，因为舍入为零时总是截去数字。乘积的有效数字类似于  $1.b_1 \dots b_{22} b_{23} \dots b_{31}$ 。如果  $b_{23} \dots b_{31} = 10 \dots 0$ ，则可能出现双舍入误差。解决这两种情况的简单方法是执行  $\text{ixflag}$  和  $b_{31}$  的逻辑 OR 运算。这样，在所有情况下  $\text{round}(N \cdot 10^{p_l})$  都能正确计算。

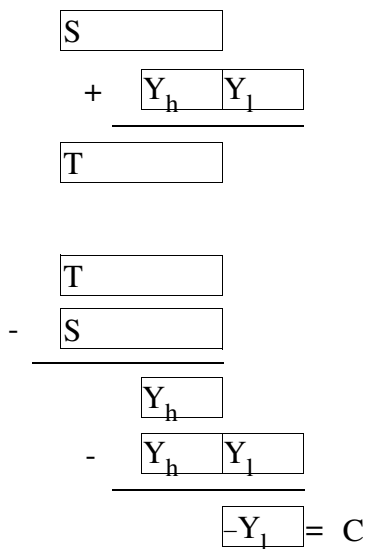
## 求和中的误差

“优化器”（在第 205 页上）部分提到了精确计算非常长的和的问题。改进精度的最简单方法是使精度加倍。为大致估算出精度加倍时和的精度改进程度，假设  $s_1 = x_1$ ， $s_2 = s_1 \oplus x_2, \dots, s_i = s_{i-1} \oplus x_i$ 。由此可得  $s_i = (1 + \delta_i)(s_{i-1} + x_i)$ ，其中  $|\delta_i| \leq \epsilon$ 。忽略  $\delta_i$  中的二次项将得出

$$s_n = \sum_{j=1}^n x_j \left( 1 + \sum_{k=j}^n \delta_k \right) = \sum_{j=1}^n x_j + \sum_{j=1}^n x_j \left( \sum_{k=j}^n \delta_k \right) \quad (31)$$

(31) 的第一个等式显示  $\sum x_j$  的计算值与对  $x_j$  的扰动值执行精确求和时的结果是相同的。第一项  $x_1$  的扰动范围是  $n\varepsilon$ ，最后一项  $x_n$  的扰动范围是  $\varepsilon$ 。(31) 中的第二个等式显示误差项的界限是  $n\varepsilon \sum |x_j|$ 。使精度加倍具有对  $\varepsilon$  取平方的效果。如果求和是以 IEEE 双精度格式执行的，那么  $1/\varepsilon \approx 10^{16}$ ，因此对于  $n$  的任何合理值， $n\varepsilon \ll 1$ 。这样，使精度加倍采用了最大扰动  $n\varepsilon$ ，并将其更改为  $n\varepsilon^2 \ll \varepsilon$ 。因此，Kahan 求和公式（定理 8）的  $2\varepsilon$  误差界限不如使用双精度好，但比使用单精度要好得多。

有关 Kahan 求和公式为何成立的直观解释，请参考下面的过程图。



每次增加被加数时，都有一个校正因子  $C$ ，该因子将在下一循环中得以应用。因此，首先从  $X_i$  中减去在上一循环中计算的校正因子  $C$ ，得出校正后的被加数  $Y$ 。然后，将此被加数增加到连续和  $S$  中。在和中， $Y$  的低位（即  $Y_l$ ）已丢失。接下来，通过计算  $T - S$  来计算  $Y$  的高位。当从中减去  $Y$  时， $Y$  的低位将恢复。这些是图中第一次求和时丢失的位。它们将成为下一循环的校正因子。定理 8 的正式证明出现在“定理 14 和定理 8”（在第 223 页上）部分中（摘自 Knuth [1981] 第 572 页）。

---

## 小结

计算机系统设计人员往往忽略系统中与浮点有关的部分。这很可能是由于计算机科学课程中很少注意浮点问题。而这又导致一种流行的看法：浮点不是一个可量化的主题，因此详细讨论处理浮点的硬件和软件没有多大意义。

本文证明了进行有关浮点的严谨推理是可能的。例如，可以证明如果基础硬件具有保护数位，则涉及抵消的浮点算法具有较小的相对误差；再如，存在一种用于二进制 - 十进制转换的高效算法，可以证明在支持扩展精度的条件下它是可逆的。当作为基础的计算机系统支持浮点时，执行构造可靠浮点软件的任务将容易得多。除了刚才提到的两个示例（保护数位和扩展精度）外，本文的“系统方面”（在第 199 页上）部分还包含许多说明如何更好地支持浮点的示例，内容涉及从指令集设计到编译器优化等各方面。

随着 IEEE 浮点标准得到越来越广泛的应用，利用该标准的各种特性的代码也将变得更具可移植性。“IEEE 标准”（在第 186 页上）部分提供了很多示例，说明在编写实际的浮点代码时如何使用 IEEE 标准的各种特性。

---

## 致谢

本文受到 W. Kahan 于 1988 年 5 月至 7 月在 Sun Microsystems 讲授的课程的启发，该课程由 Sun 的 David Hough 精心组织。我希望大家可以籍此了解浮点与计算机系统之间的相互作用，而不必起床去听上午 8 点的课。在这里，我要感谢 Kahan 和 Xerox PARC 的许多同事（尤其是 John Gilbert），他们阅读了本文的草稿并提出了许多有用的意见。另外，此讲稿的改进还要归功于 Paul Hilfinger 和一位不知名人士的审阅。

---

## 参考书目

Aho, Alfred V., Sethi, R., and Ullman J. D. 1986. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

ANSI 1978. *American National Standard Programming Language FORTRAN*, ANSI Standard X3.9-1978, American National Standards Institute, New York, NY.

Barnett, David 1987. *A Portable Floating-Point Environment*, unpublished manuscript.

Brown, W. S. 1981. *A Simple but Realistic Model of 浮点运算*, ACM Trans. on Math. Software 7(4), pp.445-480 .

- Cody, W. J et. al. 1984. *A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic*, IEEE Micro 4(4), pp.86-100 .
- Cody, W. J. 1988. *Floating-Point Standards — Theory and Practice*, in “Reliability in Computing:the role of interval methods in scientific computing”, ed. by Ramon E. Moore, pp.99-107, Academic Press, Boston, MA.
- Coonen, Jerome 1984. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, PhD Thesis, Univ. of California, Berkeley.
- Dekker, T. J. 1971. *A Floating-Point Technique for Extending the Available Precision*, Numer. Math. 18(3), pp.224-242 .
- Demmel, James 1984. *Underflow and the Reliability of Numerical Software*, SIAM J. Sci. Stat. Comput. 5(4), pp.887-919.
- Farnum, Charles 1988. *Compiler Support for 浮点运算*, Software-Practice and Experience, 18(7), pp.701-709.
- Forsythe, G. E. and Moler, C. B. 1967. *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.
- Goldberg, I. Bennett 1967. *27 Bits Are Not Enough for 8-Digit Accuracy*, Comm. of the ACM.10(2), pp 105-106.
- Goldberg, David 1990. *Computer Arithmetic*, in “Computer Architecture:A Quantitative Approach”, by David Patterson and John L. Hennessy, Appendix A, Morgan Kaufmann, Los Altos, CA.
- Golub, Gene H. and Van Loan, Charles F. 1989. *Matrix Computations*, 2nd edition, The Johns Hopkins University Press, Baltimore Maryland.
- Graham, Ronald L., Knuth, Donald E. and Patashnik, Oren.1989. *Concrete Mathematics*, Addison-Wesley, Reading, MA, p.162.
- Hewlett Packard 1982. *HP-15C Advanced Functions Handbook*.
- IEEE 1987. *IEEE Standard 754-1985 for Binary Floating-point Arithmetic*, IEEE, (1985).Reprinted in SIGPLAN 22(2) pp.9-25.
- Kahan, W. 1972. *A Survey Of Error Analysis*, in Information Processing 71, Vol 2, pp.1214 - 1239 (Ljubljana, Yugoslavia), North Holland, Amsterdam.
- Kahan, W. 1986. *Calculating Area and Angle of a Needle-like Triangle*, unpublished manuscript.
- Kahan, W. 1987. *Branch Cuts for Complex Elementary Functions*, in “The State of the Art in Numerical Analysis”, ed. by M.J.D. Powell and A. Iserles (Univ of Birmingham, England), Chapter 7, Oxford University Press, New York.
- Kahan, W. 1988. Unpublished lectures given at Sun Microsystems, Mountain View, CA.

Kahan, W. and Coonen, Jerome T. 1982. *The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments*, in “The Relationship Between Numerical Computation And Programming Languages”, ed. by J. K. Reid, pp.103-115, North-Holland, Amsterdam.

Kahan, W. and LeBlanc, E. 1985. *Anomalies in the IBM Acrith Package*, Proc.7th IEEE Symposium on Computer Arithmetic (Urbana, Illinois), pp.322-331.

Kernighan, Brian W. and Ritchie, Dennis M. 1978. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.

Kirchner, R. and Kulisch, U. 1987. *Arithmetic for Vector Processors*, Proc.8th IEEE Symposium on Computer Arithmetic (Como, Italy), pp.256-269.

Knuth, Donald E., 1981. *The Art of Computer Programming, Volume II*, Second Edition, Addison-Wesley, Reading, MA.

Kulisch, U. W., and Miranker, W. L. 1986. *The Arithmetic of the Digital Computer: A New Approach*, SIAM Review 28(1), pp 1-36.

Matula, D. W. and Kornerup, P. 1985. *Finite Precision Rational Arithmetic: Slash Number Systems*, IEEE Trans. on Comput. C-34(1), pp 3-18.

Nelson, G. 1991. *Systems Programming With Modula-3*, Prentice-Hall, Englewood Cliffs, NJ.

Reiser, John F. and Knuth, Donald E. 1975. *Evading the Drift in Floating-point Addition*, Information Processing Letters 3(3), pp 84-87.

Sterbenz, Pat H. 1974. *浮点运算*, Prentice-Hall, Englewood Cliffs, NJ.

Swartzlander, Earl E. and Alexopoulos, Aristides G. 1975. *The Sign/Logarithm Number System*, IEEE Trans. Comput. C-24(12), pp.1238-1242.

Walther, J. S., 1971. *A unified algorithm for elementary functions*, Proceedings of the AFIP Spring Joint Computer Conf.38, pp.379-385.

## 定理 14 和定理 8

此部分包含正文中省略的两个技术性更强的证明。

### 定理 14

假定  $0 < k < p$ ，设  $m = \beta^k + 1$ ，并假定浮点运算是精确舍入的。那么， $(m \otimes x) \ominus (m \otimes x \ominus x)$  与舍入到  $p - k$  个有效位的  $x$  完全相等。更精确的说法是，舍入  $x$  的方法是取  $x$  的有效数字，可以设想为小数点正好在  $k$  个最低有效位的左侧并舍入为一个整数。

### 证明

证明分为两种情况，这取决于  $mx = \beta^k x + x$  的计算是否具有进位。

假定没有进位。按比例增减  $x$  使其成为一个整数是无害的。那么， $mx = x + \beta^k x$  的计算类似于：

$$\begin{array}{r}
 aa\dots aabb\dots bb \\
 +aa\dots aabb\dots bb \\
 \hline
 zz\dots zzbb\dots bb
 \end{array}$$

其中  $x$  已经被分为两个部分。处于低位的  $k$  个数字被标记为  $b$ ，处于高位  $p - k$  个数字被标记为  $a$ 。从  $mx$  计算  $m \otimes x$  涉及舍弃处于低位的  $k$  个数字（标记为  $b$  的数字），因此

$$m \otimes x = mx - x \bmod(\beta^k) + r\beta^k \quad (32)$$

如果  $.bb\dots b$  大于  $\frac{1}{2}$ ，则  $r$  的值是 1，否则是 0。更准确地说，

$$\text{如果 } a.bb\dots b \text{ 舍入为 } a + 1, \text{ 则 } r = 1, \text{ 否则 } r = 0. \quad (33)$$

接下来，计算  $m \otimes x - x = mx - x \bmod(\beta^k) + r\beta^k - x = \beta^k(x + r) - x \bmod(\beta^k)$ 。下图显示如何计算舍入的  $m \otimes x - x$ ，即  $(m \otimes x) \ominus x$ 。顶行是  $\beta^k(x + r)$ ，其中  $B$  是将  $r$  与最低位数字  $b$  相加而产生的数字。

$$\begin{array}{r}
 aa\dots aabb\dots bB00\dots 00 \\
 -bb\dots bb \\
 \hline
 zz\dots \qquad \qquad \qquad zzZ00\dots 00
 \end{array}$$

如果  $.bb\dots b < \frac{1}{2}$ , 则  $r = 0$ , 减法运算导致从标记为  $B$  的数字借位, 但是差值将进行上舍入, 因此最终的结果是经过舍入的差等于顶行, 即  $\beta^k x$ 。如果  $.bb\dots b > \frac{1}{2}$  则  $r = 1$ , 由于借位而从  $B$  中减 1, 因此结果是  $\beta^k x$ 。最后, 以  $.bb\dots b = \frac{1}{2}$  为例。如果  $r = 0$ , 则  $B$  是偶数,  $Z$  是奇数, 将差进行上舍入会得出  $\beta^k x$ 。类似地, 当  $r = 1$  时,  $B$  是奇数,  $Z$  是偶数, 而差将进行下舍入, 因此差同样是  $\beta^k x$ 。综上所述,

$$(m \otimes x) \ominus x = \beta^k x \quad (34)$$

结合等式 (32) 和 (34) 可以得出  $(m \otimes x) - (m \otimes x \ominus x) = x - x \bmod(\beta^k) + r\beta^k$ 。执行此计算的结果是

$$\begin{array}{r} r00\dots 00 \\ + aa\dots aabb\dots bb \\ - \underline{bb\dots bb} \\ aa\dots aA00\dots 00 \end{array}$$

计算  $r$  的规则 (即等式 (33)) 与将  $a\dots ab\dots b$  舍入到  $p-k$  位的规则是相同的。因此, 在  $x + \beta^k x$  没有进位的情况下, 按浮点运算精度计算  $mx - (mx - x)$  的结果与将  $x$  舍入到  $p-k$  位完全相等。

当  $x + \beta^k x$  确实有进位时,  $mx = \beta^k x + x$  类似于:

$$\begin{array}{r} aa\dots aabb\dots bb \\ + aa\dots aabb\dots bb \\ \hline zz\dots zZbb\dots bb \end{array}$$

因此,  $m \otimes x = mx - x \bmod(\beta^k) + w\beta^k$ , 其中  $w = -Z$  (条件是  $Z < \beta/2$ ), 但是  $w$  的精确值是不重要的。接下来,  $m \otimes x - x = \beta^k x - x \bmod(\beta^k) + w\beta^k$ 。在图中

$$\begin{array}{r} aa\dots aabb\dots bb00\dots 00 \\ - bb\dots bb \\ + \underline{w} \\ zz \dots zZbb \dots bb^1 \end{array}$$

经过舍入得出  $(m \otimes x) \ominus x = \beta^k x + w\beta^k - r\beta^k$ , 其中  $r = 1$  (条件是  $.bb\dots b > \frac{1}{2}$  或者  $.bb\dots b = \frac{1}{2}$  且  $b_0 = 1$ )。<sup>2</sup> 最后,

$$\begin{aligned} (m \otimes x) - (m \otimes x \ominus x) &= mx - x \bmod(\beta^k) + w\beta^k - (\beta^k x + w\beta^k - r\beta^k) \\ &= x - x \bmod(\beta^k) + r\beta^k \end{aligned}$$

同样, 当  $a\dots ab\dots b$  舍入到  $p-k$  位涉及上舍入时,  $r = 1$  完全成立。这样就在所有情况下, 证明了定理 14。■

1. 如果加上  $w$  时不产生进位, 那么这就是和。对于加上  $w$  产生进位的特殊情况, 需要另外的参数。—编者

2. 仅当  $(\beta^k x + w\beta^k)$  保留  $\beta^k x$  的形式时, 经过舍入才得出  $\beta^k x + w\beta^k - r\beta^k$ 。—编者



## 定理 8 (Kahan 求和公式)

假定  $\sum_{j=1}^N x_j$  是使用以下算法计算的

```

S = X [1];
C = 0;
for j = 2 to N {
Y = X [j] - C;
    T = S + Y;
    C = (T - S) - Y;
    S = T;
}
    
```

那么, 计算的和  $S$  等于  $S = \sum x_j (1 + \delta_j) + O(N\epsilon^2) \sum |x_j|$ , 其中  $|\delta_j| \leq 2\epsilon$ 。

### 证明

首先, 回想一下简单公式  $\sum x_i$  的误差是如何估算的。引入  $s_1 = x_1, s_i = (1 + \delta_i)(s_{i-1} + x_i)$ 。可知计算和为  $s_n$ , 它是各项的总和, 其中的每一项都是  $x_i$  乘以包含  $\delta_i$  的表达式。  $x_1$  的精确系数是  $(1 + \delta_2)(1 + \delta_3) \dots (1 + \delta_n)$ , 因此通过重新编号,  $x_2$  的系数一定是  $(1 + \delta_3)(1 + \delta_4) \dots (1 + \delta_n)$ , 依此类推。定理 8 的证明过程完全相同, 只是  $x_1$  的系数更复杂。具体讲, 就是  $s_0 = c_0 = 0$  且

$$\begin{aligned}
 y_k &= x_k \ominus c_{k-1} = (x_k - c_{k-1})(1 + \eta_k) \\
 s_k &= s_{k-1} \oplus y_k = (s_{k-1} + y_k)(1 + \sigma_k) \\
 c_k &= (s_k \ominus s_{k-1}) \ominus y_k = [(s_k - s_{k-1})(1 + \gamma_k) - y_k](1 + \delta_k)
 \end{aligned}$$

其中所有希腊字母的界限都是  $\epsilon$ 。虽然  $s_k$  中  $x_1$  的系数是所需的最终表达式, 但计算  $s_k - c_k$  和  $c_k$  中  $x_1$  的系数更容易。

当  $k = 1$  时,

$$\begin{aligned}
 c_1 &= (s_1(1 + \gamma_1) - y_1)(1 + d_1) \\
 &= y_1((1 + s_1)(1 + \gamma_1) - 1)(1 + d_1) \\
 &= x_1(s_1 + \gamma_1 + s_1 g_1)(1 + d_1)(1 + h_1) \\
 s_1 - c_1 &= x_1[(1 + s_1) - (s_1 + g_1 + s_1 g_1)(1 + d_1)](1 + h_1) \\
 &= x_1[1 - g_1 - s_1 d_1 - s_1 g_1 - d_1 g_1 - s_1 g_1 d_1](1 + h_1)
 \end{aligned}$$

分别调用这两个表达式  $C_k$  和  $S_k$  中  $x_1$  的系数, 那么

$$C_1 = 2\epsilon + O(\epsilon^2)$$

$$S_1 = + \eta_1 - \gamma_1 + 4\epsilon^2 + O(\epsilon^3)$$

要得出  $S_k$  和  $C_k$  的通用公式, 请展开  $s_k$  和  $c_k$  的定义, 忽略所有涉及  $x_i (i > 1)$  的项, 得到

$$\begin{aligned}
 s_k &= (s_{k-1} + y_k)(1 + \sigma_k) \\
 &= [s_{k-1} + (x_k - c_{k-1})(1 + \eta_k)](1 + \sigma_k) \\
 &= [(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}](1 + \sigma_k) \\
 c_k &= [\{s_k - s_{k-1}\}(1 + \gamma_k) - y_k](1 + \delta_k) \\
 &= [\{(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}\}(1 + \sigma_k) - s_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k) \\
 &= [\{(s_{k-1} - c_{k-1})\sigma_k - \eta_k c_{k-1}(1 + \sigma_k) - c_{k-1}\}(1 + \gamma_k) + c_{k-1}(1 + \eta_k)](1 + \delta_k) \\
 &= [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))](1 + \delta_k), \\
 s_k - c_k &= ((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1 + \sigma_k) \\
 &\quad - [(s_{k-1} - c_{k-1})\sigma_k(1 + \gamma_k) - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))](1 + \delta_k) \\
 &= (s_{k-1} - c_{k-1})((1 + \sigma_k) - \sigma_k(1 + \gamma_k)(1 + \delta_k)) \\
 &\quad + c_{k-1}(-\eta_k(1 + \sigma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))(1 + \delta_k)) \\
 &= (s_{k-1} - c_{k-1})(1 - \sigma_k(\gamma_k + \delta_k + \gamma_k \delta_k)) \\
 &\quad + c_{k-1}[-\eta_k + \gamma_k + \eta_k(\gamma_k + \sigma_k \gamma_k) + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k \gamma_k))\delta_k]
 \end{aligned}$$

由于  $S_k$  和  $C_k$  最多仅计算到次数  $\varepsilon^2$ , 因此这些公式可以简化为

$$\begin{aligned}
 C_k &= (\sigma_k + O(\varepsilon^2))S_{k-1} + (-\gamma_k + O(\varepsilon^2))C_{k-1} \\
 S_k &= ((1 + 2\varepsilon^2 + O(\varepsilon^3))S_{k-1} + (2\varepsilon + O(\varepsilon^2))C_{k-1}
 \end{aligned}$$

使用这些公式可以得出

$$\begin{aligned}
 C_2 &= \sigma_2 + O(\varepsilon^2) \\
 S_2 &= 1 + \eta_1 - \gamma_1 + 10\varepsilon^2 + O(\varepsilon^3)
 \end{aligned}$$

一般情况下通过归纳可以很容易地得到

$$\begin{aligned}
 C_k &= \sigma_k + O(\varepsilon^2) \\
 S_k &= 1 + \eta_1 - \gamma_1 + (4_k + 2)\varepsilon^2 + O(\varepsilon^3)
 \end{aligned}$$

最后, 所需的是  $s_k$  中  $x_1$  的系数。要获得此值, 令  $x_{n+1} = 0$ , 假设下标为  $n+1$  的所有希腊字母都等于 0, 计算  $s_{n+1}$ 。可知,  $s_{n+1} = s_n - c_n$ , 且  $s_n$  中  $x_1$  的系数小于它在  $s_{n+1}$  中的系数, 即  $S_n = 1 + \eta_1 - \gamma_1 + (4_n + 2)\varepsilon^2 = (1 + 2\varepsilon + O(n\varepsilon^2))$ 。■

---

# IEEE 754 各实现之间的差异

---

**注意** - 此部分不是已发表的论文的一部分。增加此部分是为了澄清某些观点，并纠正读者可能通过论文推知的某些有关 IEEE 标准的误解。此材料不是由 David Goldberg 编写的，但却是经过了他的许可才在此处公布。

---

前面的论文已经表明：执行浮点运算时必须小心谨慎，因为程序员可能要依靠其属性来实现程序的正确性和准确性。特别是在实现 IEEE 标准时需小心谨慎，只有在符合标准的系统上才能编写出能够正常工作并给出准确结果的有用程序。读者可能会据此得出结论：这样的程序应该可以移植到所有 IEEE 系统上。事实上，如果“当一个程序在两台支持 IEEE 运算的计算机之间迁移时，只要任意中间结果出现不同，则一定是由于软件错误，而不是算法差异。”这一条件成立，那么编写可移植软件会变得更加容易。

遗憾的是，IEEE 标准并不保证同一程序在所有符合该标准的系统上都提供完全相同的结果。实际上，由于种种原因，大多数程序都会在不同的系统上产生不同的结果。其中一个原因是，大多数程序都涉及十进制格式和二进制格式之间的数字转换，而 IEEE 标准没有完全指定执行这样的转换必须使用的准确度。另一个原因是，许多程序使用由系统库提供的初等函数，而该标准并没有指定这些函数。当然，大多数程序员都知道这些功能已经超出了 IEEE 标准的范围。

许多程序员可能没有意识到，甚至是仅使用 IEEE 标准规定的数字格式和运算的程序也可能在不同的系统上计算出不同的结果。实际上，该标准制定者的初衷就是允许不同的实现获得不同的结果。在 IEEE 754 标准中术语 *目标* 的定义里清晰地表达了这个意思：“目标可以被用户显式指定，也可以由系统隐式提供（例如，子表达式或过程参数中的中间结果）。某些语言会将中间计算的结果放在用户无法控制的目标中。但是，此标准根据该目标的格式和操作数的值定义运算的结果。”（IEEE 754-1985，第 7 页）换句话说，IEEE 标准要求将每个结果都正确舍入到将放置它的目标的精度，但是标准不要求由用户程序确定该目标精度。因此，不同的系统可能将其结果提供给不同精度的目标，使同一程序产生不同的结果（有时差异很大），即使那些系统都符合标准亦是如此。

前面论文中的几个实例需要有关舍入浮点运算方式的某些知识。为了灵活使用这些示例，程序员必须能够预知将如何解释程序，尤其是，在 IEEE 系统上，每个算术运算的目标精度可能是什么。但是，IEEE 标准中 *目标* 定义的漏洞削弱了程序员了解如何解释程序的能力。因此，在高级语言中作为明显可移植程序实现时，上面给出的几个实例可能无法在 IEEE 系统上正常工作，通常向目标提供结果时，其精度与程序员预期的精度不同。其它实例也可能正常工作，但是要证明它们是否正常工作可能超出了一般程序员的能力。

在此部分中，我们根据 IEEE 754 运算的现有实现所通常使用的目标格式的精度对它们进行分类。然后，回顾论文中的一些实例，来说明以比程序预期精度更宽的精度提供结果会导致它计算出错误的结果，即使在使用期望的精度时它被证明是正确的。我们还可以再查看论文中其中一个证明，以阐明处理未预料的精度所需的脑力工作，即使该精度还没有使程序无效。这些实例说明，IEEE 标准允许在不同实现之间存在差异会阻止我们编写出可以准确预知其行为的可移植、高效数值软件，这与它规定的所有内容无关。要开发这样的软件，则首先必须创建限制 IEEE 标准所允许的可变性的程序设计语言和环境，并允许程序员表示其程序所依赖的浮点语义。

## 当前的 IEEE 754 实现

IEEE 754 运算的当前实现可以分为两组，它们是按支持硬件中不同浮点格式的程度区分的。基于扩展的系统，如 Intel x86 系列处理器，完全支持扩展的双精度格式，但是仅部分支持单精度和双精度：它们提供按单精度和双精度装入或存储数据的指令，飞速地将数据在单精度和双精度与扩展双精度格式之间来回转换，它们还提供特殊模式（不是缺省模式），在这样的模式下按单精度或双精度舍入算术运算的结果，即使这些结果以扩展双精度格式保存在寄存器中。（Motorola 68000 系列处理器在这些模式下按单精度或双精度格式的精度和范围舍入结果。Intel x86 及兼容处理器按单精度或双精度格式的精度舍入结果，但保留与扩展双精度格式相同的范围。）单精度 / 双精度系统，包括大多数 RISC 处理器，完全支持单精度格式和双精度格式，但不支持符合 IEEE 的扩展双精度格式。（IBM POWER 架构仅部分支持单精度，但出于本节组织的目的，我们将其归入单精度 / 双精度系统。）

要查看计算的行为在基于扩展的系统上与在单精度 / 双精度系统上有何不同，请参考“系统方面”（在第 199 页上）中的 C 版本示例：

```
int main() {
    double q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

在这里，将常数 3.0 和 7.0 解释为双精度浮点数，而且表达式 3.0/7.0 继承双精度数据类型。在单精度 / 精度双系统上，将按双精度计算表达式的值，因为那是可以使用的最有效的格式。因此，将按双精度精确舍入 3.0/7.0 的值赋予 `q`。在下一行中，将再次按双精度计算表达式 3.0/7.0 的值，当然结果将等于刚赋给 `q` 的值，于是程序将像期望的那样打印“Equal”。

在基于扩展的系统上，即使表达式  $3.0/7.0$  的类型为**双精度**，也将以扩展双精度格式在寄存器中计算商，因此在缺省模式下，将按扩展双精度舍入它。但是，当将计算的值赋予变量  $q$  时，有可能随后将它存储在内存中，因为  $q$  被声明为**双精度**，所以将按双精度舍入该值。在下一行中，可能再次按扩展精度计算表达式  $3.0/7.0$  的值，产生的结果与存储在  $q$  中的双精度值不同，使程序打印“Not equal”。当然，其它结果也是可能的：编译器可以确定在将表达式  $3.0/7.0$  的值与  $q$  进行比较之前在第二行中存储该值并舍入它，或者可以按扩展精度将  $q$  保存在寄存器中而不存储它。优化的编译器可能在编译时（也许是按双精度，也许是按扩展双精度）计算表达式  $3.0/7.0$  的值。（使用同一个 x86 编译器，在为优化进行编译时程序将打印“Equal”，在为调试进行编译时将打印“Not Equal”。）最后，基于扩展的系统的某些编译器自动更改舍入精度模式，以使在寄存器中产生结果的运算按单精度或双精度舍入那些结果，尽管使用更宽的范围是可能的。这样，在这些系统上，我们无法仅通过读取其源代码并运用 IEEE 754 运算的基本知识来预知程序的行为。我们也不能将未能提供符合 IEEE 754 的环境归因于硬件或编译器；硬件已经将正确舍入的结果提供给每个目标（按要求它做的那样），而且编译器已经将某些中间结果赋予超出用户控制的目标（按允许它做的那样）。

## 在基于扩展的系统上的计算缺陷

按一般的思维，基于扩展的系统必须产生至少是精确的结果，即便不比在单精度/双精度系统上提供的更精确；因为前者始终提供尽可能高的精度，而且通常比后者高。通常的实例（如上面的 C 程序）以及基于下面讨论的实例的更细致程序说明这种认识至少有一些天真：一些明显可移植的程序确实是可以跨单精度/双精度系统移植的，它们在基于扩展的系统上会提供错误的结果，完全是由于编译器和硬件协同工作，提供的精度有时候比程序所需的高。

当前的程序设计语言使程序很难指定它所需的精度。如“语言和编译器”（在第 201 页上）上的“语言和编译器”部分所述，许多程序设计语言不指定在同一上下文中某个表达式（如  $10.0*x$ ）每次运行应该计算出相同的值。在这方面，某些语言（如 Ada）就受到了 IEEE 标准之前的不同运算之间差异的影响。最近，类似 ANSI C 的语言受到了符合标准的、基于扩展的系统的影 响。事实上，ANSI C 标准明确允许编译器按比通常与其类型关联的精度更宽的精度计算浮点表达式的值。因此，表达式  $10.0*x$  的值可能以取决于以下各种因素的方式变化：表达式是立即赋予变量还是作为子表达式出现在更大的表达式中；表达式是否参与比较；表达式是否作为参数传递给函数，如果是这样，该参数是按值还是按引用传递；当前的精度模式；编译程序时的优化级别；在编译程序时编译器所使用的精度模式和表达式计算方法；等等。

不能将表达式计算不完全一致全部归因于语言标准。可能的时候，在扩展精度寄存器中计算表达式，基于扩展的系统的运行效率最高，但是必须存储的值是按所需的最窄精度存储的。约束一种语言使其要求  $10.0*x$  在任何位置都计算为同一值将降低那些系统的性能。可惜的是，允许那些系统在语义上等价的上下文中将  $10.0*x$  计算为不同的值会对开发精确数值软件的程序员产生它自己的负面影响，它阻止程序员依赖其程序语法来表示预定语义。

真正的程序是否依赖给定表达式始终计算为同一值的假设？回想在定理 4 中提供的用于计算  $\ln(1+x)$  的算法，在此处用 Fortran 编写：

```
real function log1p(x)
real x
if (1.0 + x .eq.1.0) then
    log1p = x
else
    log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

在基于扩展的系统上，编译器可能在第三行中按扩展精度计算表达式  $1.0 + x$  的值并将结果与  $1.0$  进行比较。但是，将同一表达式传递给第六行中的  $\log$  函数时，编译器可能将其值存储在内存中，并按单精度对其进行舍入。这样，如果对于按扩展精度不能将  $1.0 + x$  舍入为  $1.0$ ， $x$  不够小，但对于按单精度可以将  $1.0 + x$  舍入为  $1.0$ ，对于单精度已经足够小，那么  $\log1p(x)$  返回的值将是零，而不是  $x$ ，相对误差将是  $1 - \epsilon$  远大于  $5\epsilon$ 。同样，假定第六行中表达式的其余部分（包括子表达式  $1.0 + x$  的重新出现）是按扩展精度计算的。在这种情况下，如果  $x$  较小但没有小到足以按单精度将  $1.0 + x$  舍入为  $1.0$ ，则  $\log1p(x)$  返回的值可以超过正确值接近  $x$ ，同样相对误差可以接近 1。下面给出一个具体的实例，假设  $x$  等于  $2^{-24} + 2^{-47}$ ，因此  $x$  是最小的单精度数，所以  $1.0 + x$  上舍入到下一个较大的数  $1 + 2^{-23}$ 。那么， $\log(1.0 + x)$  大约等于  $2^{-23}$ 。因为第六行中的表达式的分母是按扩展精度计算的，所以它是精确计算的并提供  $x$ ，这样  $\log1p(x)$  返回的值大约等于  $2^{-23}$ ，该值几乎是精确值的两倍。（至少一种编译器实际出现过这种情况。当用于 x86 系统的 Sun WorkShop Compilers 4.2.1 Fortran 77 编译器使用  $-O$  优化标志编译前面的代码时，生成的代码完全像说明的那样计算  $1.0 + x$ 。因此，函数为  $\log1p(1.0e-10)$  提供零，为  $\log1p(5.97e-8)$  提供  $1.19209E-07$ 。）

为了使定理 4 的算法正确工作，表达式  $1.0 + x$  每次出现时都必须按相同的方式进行计算：仅当  $1.0 + x$  在一个实例中按扩展双精度计算而在另一实例中按单精度或双精度计算时，该算法才会在基于扩展的系统上失败。当然，因为  $\log$  是 Fortran 中的通用内函数，所以编译器可能自始至终都按扩展精度计算表达式  $1.0 + x$  的值，按同一精度计算其对数，但显然我们不能假定编译器将这样做。（也可以设想涉及用户定义函数的类似实例。在这种情况下，即使函数返回单精度结果，编译器也可能仍然按扩展精度保存参数，但是现有的 Fortran 编译器 [ 如果有的话 ] 几乎没有这样做的。）因此，我们可能试图通过将  $1.0 + x$  赋予一个变量来确保它是一致计算的。可惜的是，如果我们声明该变量  $real$ ，则可能仍然被这样的编译器阻止：它以按扩展精度保存在寄存器中的值替换变量的一个实例，而以按单精度存储在内存中的值替换变量的另一实例。相反，我们将需要使用对应于扩展精度格式的类型声明该变量。标准 FORTRAN 77 没有提供做到这一点的方法，尽管 Fortran 95 提供了用于说明各种格式的 SELECTED\_REAL\_KIND 机制，但它不显式要求按扩展精度计算表达式的实现允许使用该精度声明变量。简而言之，如果用标准 Fortran 编写此程序，则不存在一定阻止表达式  $1.0 + x$  按使我们的证明无效的方式进行计算的可移植方法。

还有一些在基于扩展的系统上会出错的实例，甚至是在存储每个子表达式再按同一精度舍入时。原因是双舍入。在缺省的精度模式下，基于扩展的系统最初将按扩展双精度舍入每个结果。如果此后该结果按双精度存储，则再次对它进行舍入。这两次舍入的组合产生的值与通过按双精度正确舍入第一个结果所得到的值是不同的。当按扩展双精度舍入的结果是“中途情况”（即，它恰好位于两个双精度数的中间）时，会出现这种情况，因此第二次舍入是由“在中途情况下舍入为偶数”规则确定的。如果这个第二次舍入的舍入方向与第一次的相同，则最终的舍入误差将超过最后一位上的半个单位。（但是，请注意双舍入仅影响双精度计算。可以证明：两个  $p$ - 位数的和、差、积或商，或者  $p$  位数的平方根，先舍入为  $q$  位再舍入为  $p$  位将得出与将结果只舍入为  $p$  位一次相同的值，条件是  $q \geq 2p + 2$ 。因此，扩展双精度足够宽，单精度计算不会受到双舍入的影响。）

依赖正确舍入的一些算法会因双舍入而失败。事实上，甚至是不要求正确舍入且在不符合 IEEE 754 的各种计算机上正确工作的一些算法也会因双舍入而失败。其中最有用的算法是用于执行在“定理 5”（在第 183 页上）部分中提到的模拟多精度运算的可移植算法。例如，定理 6 中所述的用于将浮点数拆分为高位和低位部分的过程在双舍入运算中不能正常工作：试图将双精度数  $2^{52} + 3 \times 2^{26} - 1$  拆分为两部分，各部分最多为 26 位。按双精度正确舍入每个运算时，高位部分是  $2^{52} + 2^{27}$ ，低位部分是  $2^{26} - 1$ ，但先按扩展双精度舍入每个运算再按双精度舍入时，该过程产生的高位部分是  $2^{52} + 2^{28}$ ，低位部分是  $-2^{26} - 1$ 。后一个数占用 27 位，因此无法按双精度精确计算其平方。当然，仍有可能按扩展双精度计算此数的平方，但是得出的对数将不再能够移植到单精度 / 双精度系统。此外，多精度乘法算法中的后续步骤假定按双精度计算了所有的部分乘积。正确处理双精度变量和扩展双精度变量的混合将大大提高实现的成本。

同样，用于将表示为双精度数的数组的多精度数相加的可移植算法会在双舍入运算中失败。这些算法通常依赖类似于 Kahan 求和公式的方法。正如“求和中的误差”（在第 218 页上）部分中给出的求和公式非正式解释所提出的，如果  $s$  和  $y$  是浮点变量并且  $|s| \geq |y|$ ，计算：

```

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

那么，在大多数运算中， $e$  精确恢复计算  $t$  时出现的舍入误差。但是，此方法在双舍入运算中不能正常工作：如果  $s = 2^{52} + 1$  且  $y = 1/2 - 2^{-54}$ ，那么， $s + y$  先按扩展双精度舍入为  $2^{52} + 3/2$ ，然后依据“在中途情况下舍入为偶数”规则按双精度将此值舍入为  $2^{52} + 2$ ；这样，计算  $t$  时的最终舍入误差是  $1/2 + 2^{-54}$ ，它无法按双精度精确表示，因此它无法用上面所示的表达式精确计算。同样，通过按扩展双精度来计算和从而恢复舍入误差将是可能的，但是这样的话程序将必须进行额外的工作才能将最终输出约简回双精度，双舍入也会影响此过程。由于这一原因，虽然通过这些方法模拟多精度运算的可移植程序在各种各样的计算机上正确而高效地运行，但是它们在基于扩展的系统上并不像所声称的那样运行。

最后，起初看上去似乎依赖正确舍入的一些算法事实上可能与双舍入一起正确工作。在这些情况下，处理双舍入的成本不在于实现，而在于对算法是否像声称的那样工作进行检验。为了说明这一点，我们证明定理 7 的以下变形：

## 定理 7'

如果  $m$  和  $n$  是可以按 IEEE 754 双精度表示的整数,  $|m| < 2^{52}$ ,  $n$  具有特殊形式  $n = 2^i + 2^j$ , 那么  $(m \oslash n) \otimes n = m$ , 条件是两个浮点运算都是按双精度正确舍入的或先按扩展双精度舍入再按双精度舍入的。

## 证明

假定  $m > 0$ , 令  $q = m \oslash n$ 。将 2 的幂作为因子进行调整, 我们可以考虑一个等价设置, 其中  $2^{52} \leq m < 2^{53}$ , 对于  $q$  是类似的, 以便  $m$  和  $q$  都是这样的整数: 其最低有效位占用单位位置 (即  $\text{ulp}(m) = \text{ulp}(q) = 1$ )。在调整之前, 假定  $m < 2^{52}$ , 因此在调整之后,  $m$  是一个偶数整数。此外, 因为  $m$  和  $q$  的调整值满足  $m/2 < q < 2m$ , 所以  $n$  的对应值必须具有两种形式中的一种, 具体取决于  $m$  和  $q$  哪个较大: 如果  $q < m$ , 显然  $1 < n < 2$ , 因为  $n$  是两个二的幂的和, 所以对于一些  $k$ ,  $n = 1 + 2^{-k}$ ; 同样, 如果  $q > m$ , 那么  $1/2 < n < 1$ , 因此  $n = 1/2 + 2^{-(k+1)}$ 。(因为  $n$  是两个二的幂的和, 所以  $n$  的最接近一的可能值是  $n = 1 + 2^{-52}$ 。因为  $m/(1 + 2^{-52})$  不比小于  $m$  的下一个较小双精度数大, 所以不能使  $q = m$ 。)

假设  $e$  表示计算  $q$  时的舍入误差, 以便  $q = m/n + e$ , 计算值  $q \otimes n$  将是  $m + ne$  的 (一次或两次) 舍入值。请首先考虑按双精度正确舍入每个浮点运算的情况。在这种情况下,  $|e| < 1/2$ 。如果  $n$  具有形式  $1/2 + 2^{-(k+1)}$ , 则  $ne = nq - m$  是  $2^{-(k+1)}$  的整数倍, 且  $|ne| < 1/4 + 2^{-(k+2)}$ 。这意味着  $|ne| \leq 1/4$ 。请回想一下  $m$  与下一个较大的可表示数之间的差是 1,  $m$  和下一个较小的可表示数之间的差是 1 (如果  $m > 2^{52}$ ) 或  $1/2$  (如果  $m = 2^{52}$ )。这样, 因为  $|ne| \leq 1/4$ ,  $m + ne$  将舍入为  $m$ 。(即使  $m = 2^{52}$  且  $ne = -1/4$ , 乘积也将按“在中途情况下舍入为偶数”规则舍入为  $m$ 。) 同样, 如果  $n$  具有形式  $1 + 2^{-k}$ , 则  $ne$  是  $2^{-k}$  的整数倍, 且  $|ne| < 1/2 + 2^{-(k+1)}$ ; 这隐含  $|ne| \leq 1/2$ 。在这种情况下, 不能使  $m = 2^{52}$ , 因为  $m$  严格大于  $q$ , 因此  $m$  与其最接近的可表示数相差  $\pm 1$ 。这样, 因为  $|ne| \leq 1/2$ ,  $m + ne$  同样将舍入为  $m$ 。(即使  $|ne| = 1/2$ , 乘积也将按“在中途情况下舍入为偶数”规则舍入为  $m$ , 因为  $m$  是偶数。) 这样就完成了正确舍入运算的证明。

在双舍入运算中, 仍可能发生以下情况:  $q$  是正确舍入的商 (即使它实际上舍入了两次), 因此像上面那样  $|e| < 1/2$ 。在这种情况下, 我们可以求助于上一段中的论点, 条件是考虑到  $q \otimes n$  将被舍入两次的事实。为了解释这一点, 请注意 IEEE 标准要求扩展双精度格式至少有 64 个有效位, 以便数  $m \pm 1/2$  和  $m \pm 1/4$  可以按扩展双精度精确表示。这样, 如果  $n$  具有形式  $1/2 + 2^{-(k+1)}$ , 以便  $|ne| \leq 1/4$ , 则按扩展双精度舍入  $m + ne$  必须产生与  $m$  最多相差  $1/4$  的结果, 如上所述, 此值将按双精度舍入为  $m$ 。同样, 如果  $n$  具有形式  $1 + 2^{-k}$ , 以便  $|ne| \leq 1/2$ , 则按扩展双精度舍入  $m + ne$  必须产生与  $m$  最多相差  $1/2$  的结果, 而且此结果将按双精度舍入为  $m$ 。(请回想, 在这种情况下  $m > 2^{52}$ 。)

最后, 要考虑具有以下特点的其余情况: 因双舍入而导致  $q$  不是正确舍入的商。在这些情况下, 最差具有  $|e| < 1/2 + 2^{-(d+1)}$ , 其中  $d$  是扩展双精度格式中的额外位数。(所有基于扩展的现有系统都支持正好具有 64 个有效位的扩展双精度格式; 对于此格式,  $d = 64 - 53 = 11$ 。) 因为在第二次舍入由“在中途情况下舍入为偶数”规则确定时双舍入仅产生不正确的舍入结果, 所以  $q$  必须是偶数整数。这样, 如果  $n$  具有形式  $1/2 + 2^{-(k+1)}$ , 则  $ne = nq - m$  是  $2^{-k}$  的整数倍, 且

$$|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}.$$



如果  $k \leq d$ ，则意味着  $|ne| \leq 1/4$ 。如果  $k > d$ ，则我们具有  $|ne| \leq 1/4 + 2^{-(d+2)}$ 。在任一情况下，乘积的第一次舍入都将提供与  $m$  最多相差  $1/4$  的结果，依据上述论点，第二次舍入将舍入为  $m$ 。同样，如果  $n$  具有形式  $1 + 2^{-k}$ ，则  $ne$  是  $2^{-(k-1)}$  的整数倍，且

$$|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}.$$

如果  $k \leq d$ ，则意味着  $|ne| \leq 1/2$ 。如果  $k > d$ ，则我们具有  $|ne| \leq 1/2 + 2^{-(d+1)}$ 。在任一情况下，乘积的第一次舍入都将提供与  $m$  最多相差  $1/2$  的结果，同样依据上述论点，第二次舍入将舍入为  $m$ 。■

前面的证明表明：仅当商引起双舍入时，乘积才会引起双舍入，甚至在此时乘积也舍入为正确的结果。该证明还表明：即使是对于只有两个浮点运算的程序来说，扩展我们的推理以包括存在双舍入的可能也可以是有挑战性的。对于更复杂的程序，系统地解释双舍入的影响也许是不可能的，更不必说双精度计算和扩展双精度计算的更一般组合。

## 扩展精度的程序设计语言支持

不应该由前面的实例得出扩展精度本身是无益的这一结论。当程序员能够有选择地使用扩展精度时，许多程序可以从扩展精度受益。遗憾的是，当前的程序设计语言没有提供足够的方式，供程序员用来指定应该使用扩展精度的时间和方式。为了指出所需的支持，我们考虑可能希望管理扩展精度的使用的方式。

在将双精度用作其标称工作精度的可移植程序中，我们可能希望控制更宽精度的使用方式有以下五种：

1. 在基于扩展的系统上尽可能使用扩展精度进行编译，以产生速度最快的代码。显然，大多数数值软件对运算的要求仅仅是每个运算中的相对误差以“计算机厄普西隆 ( $\epsilon$ )”为界限。当内存中的数据按双精度存储时，通常认为计算机厄普西隆是该精度中的最大相对舍入误差，因为（正确或错误地）假定在输入数据被输入时已经过舍入，在存储结果时将对其进行类似舍入。这样，尽管按扩展精度计算一些中间结果可能产生更精确的结果，但是扩展精度不是必需的。在这种情况下，我们可能更希望仅当扩展精度不会明显减慢程序速度时编译器才使用扩展精度，否则使用双精度。
2. 如果比双精度宽的一种格式相当快且足够宽，则使用该格式，否则使用其它格式。当扩展精度可用时，一些计算可以更容易地执行，它们也可以按双精度执行，但是会更加复杂。以计算双精度数向量的欧几里得范数为例。通过计算元素的平方并以 IEEE 754 扩展双精度格式（使用其较宽的指数范围）累加其和，可以为实用长度的向量很普遍地避免过早下溢或上溢。在基于扩展的系统上，这是计算范数的最快方式。在单精度 / 双精度系统上，扩展双精度格式将必须在软件中仿真（如果支持一种格式的话），而且这样的仿真将比仅使用双精度慢得多，测试异常标志以确定是否发生下溢或上溢，如果是这样，使用显式调整重复该计算。请注意，要支持扩展精度的这一使用，语言必须同时提供相当快的最宽可用格式的指示（以便程序可以选择使用哪种方法）和指示每种格式的精度及范围的环境参数（以便程序可以检验最宽的快速格式是否足够宽，例如，检验它是否具有比双精度更宽的范围）。

3. 即使比双精度更宽的格式必须在软件中仿真，也要使用该格式。对于比欧几里得范数实例更复杂的程序，程序员可能仅希望消除编写程序的两个版本的需要；相反，即使扩展精度的速度慢也依赖它。同样，语言必须提供环境参数，程序才能确定最宽可用格式的范围和精度。
4. 不要使用更宽的精度；按双精度格式的精度正确舍入结果（尽管可能使用扩展的范围）。为了使最容易编写的程序依赖正确舍入的双精度运算，包括上述的一些实例，语言必须提供一种方法，供程序员用来指示不得使用扩展精度，即使中间结果可能在指数范围比双精度更宽的寄存器中计算。（如果以这种方式计算的中间结果存储到内存时发生下溢，则它们仍然可能引起双舍入：如果首先将算术运算的结果舍入为 53 个有效位，然后当必须对它进行非正规化时再舍入到较少的有效位，最终的结果可能与仅通过一次舍入为非正规数所获得的结果不同。当然，此形式的双舍入几乎不大可能对实际程序产生任何负面影响。）
5. 按双精度格式的精度和范围正确舍入结果。对于测试数值软件或接近双精度格式的范围及精度的极限的运算本身的程序，进行双精度严格强制将是极其有用的。以可移植方式编写这样细致的测试程序往往是很困难的；当它们必须利用伪子例程和其它技巧强制按特定格式舍入结果时，编写它们更加困难（且容易出错）。因此，使用基于扩展的系统开发必须可以移植到所有 IEEE 754 实现的可靠软件的程序员，将很快发现可以仿真单精度 / 双精度系统的运算而无需特别的努力。

当前语言都不支持所有这五种选择。事实上，几乎没有语言尝试为程序员提供控制扩展精度的使用的能力。一个值得注意的例外是 ISO/IEC 9899:1999 程序设计语言 - C 标准，它是 C 语言的最新修订，现在处于标准化的最终阶段。

C99 标准允许实现以比通常与其类型关联的格式更宽的格式计算表达式的值，但是 C99 标准建议使用仅仅三种表达式计算方法之一。建议使用的三种方法以将表达式“提升”到更宽格式的程度为特征，鼓励实现通过定义预处理程序宏 `FLT_EVAL_METHOD` 来识别它使用哪种方法：如果 `FLT_EVAL_METHOD` 是 0，则以对应于其类型的格式计算每个表达式；如果 `FLT_EVAL_METHOD` 是 1，则将浮点表达式提升到对应于双精度的格式；如果 `FLT_EVAL_METHOD` 是 2，则将浮点和双精度表达式提升到对应于长双精度的格式。（允许实现将 `FLT_EVAL_METHOD` 设置为 -1 以指示表达式计算方法是不能确定的。）C99 标准还要求 `<math.h>` 头文件定义类型 `float_t` 和 `double_t`，它们分别至少与浮点和双精度一样宽，旨在与用于计算浮点和双精度表达式的类型匹配。例如，如果 `FLT_EVAL_METHOD` 是 2，则 `float_t` 和 `double_t` 都是长双精度。最后，C99 标准要求 `<float.h>` 头文件定义指定对应于每种浮点类型的格式的范围及精度的预处理程序宏。

C99 标准要求或建议的功能组合支持上面列出的五种选择中的一些选择，但是不支持所有选择。例如，如果实现将长双精度类型映射到扩展双精度格式，并将 `FLT_EVAL_METHOD` 定义为 2，则程序员可以合理地假定扩展精度的速度较快，因此类似欧几里得范数实例的程序只需使用长双精度（或 `double_t`）类型的中间变量即可。另一方面，同一实现必须按扩展精度保存匿名表达式，即使它们存储在内存中（例如，当编译器必须使浮点寄存器溢出时），并且它必须存储赋予声明为双精度的变量的表达式的结果，以便将其转换为双精度，即使它们可以保存在寄存器中。这样，双精度或 `double_t` 类型都不能进行编译以便在基于扩展的当前硬件上产生速度最快的代码。

同样，C99 标准提供了此部分中实例所说明的一些问题（但不是所有问题）的解决方法。如果将表达式  $1.0 + x$  赋予一个变量（为任何类型）且该变量是自始至终使用的，则可以保证 `log1p` 函数的 C99 标准版本正确工作。但是，用于将双精度数拆分为高位部分和低位部分的可移植的、高效的 C99 标准程序更为困难：在无法保证按双精度正确舍入双精度表达式的情况下，如何在正确的位置进行拆分并避免双舍入？一种解决方法是使用 `double_t` 类型在单精度 / 双精度系统上按双精度执行拆分，在基于扩展的系统上按扩展精度进行拆分，以便在任一情况下都将正确舍入运算。定理 14 告诉我们可以任意位的位置进行拆分，条件是知道基础运算的精度，`FLT_EVAL_METHOD` 和环境参数宏应该提供此信息。

下面的程序片段显示一种可能的实现：

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

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif

...
double    x, xh, xl;
double_t  m;

m = scalbn(1.0, PWR2) + 1.0; // 2**PWR2 + 1
xh = (m * x) - ((m * x) - x);
xl = x - xh;
```

当然，要求出此解，程序员必须知道：双精度表达式可能是按扩展精度计算的；随之出现的双舍入问题可以导致算法出错；依据定理 14 可能改用扩展精度。更显而易见的解决方法是只需指定按双精度正确舍入每个表达式即可。在基于扩展的系统上，这只要求更改舍入精度模式，但可惜的是，C99 标准没有提供做到这一点的可移植方法。

（*Floating-Point C Edits*（指定为支持浮点而对 C90 标准进行的更改的工作文档）的早期草案建议具有舍入精度模式的系统上的实现提供 `fegetprec` 和 `fesetprec` 函数以获取和设置舍入精度，这两个函数与获取和设置舍入方向的 `fegetround` 和 `fesetround` 函数类似。在对 C99 标准进行更改之前删除了此建议案。）

巧合的是，C99 标准支持具有不同整数运算功能的系统之间的可移植性的途径是建议了一种支持不同浮点架构的更好方法。每个 C99 标准实现都提供一个 `<stdint.h>` 头文件，该文件定义实现所支持的那些整数类型，它们按大小和效率命名：例如，`int32_t` 是 32 位宽的整数类型，`int_fast16_t` 是实现的速度最快且至少 16 位宽的整数类型，`intmax_t` 是支持的最宽整数类型。可以为浮点类型设想一个类似的方案：例如，`float53_t` 可用于命名具有正好 53 位精度但可能具有更宽范围的一种浮点类型，`float_fast24_t` 可用于命名实现的速度最快且具有至少 24 位精度的类型，`floatmax_t` 可用于命名支持的最宽且速度相当快的类型。快速类型可能允许基于扩展的系统上的编译器生成可能最快的代码，这仅受到以下约束的影响：指定变量的值不能因寄存器溢出而改变。精确宽度类型将导致基于扩展的系统上的编译器将舍入精度模式设置为按指定精度舍入，从而允许更宽的范围受到同一约束的影响。最后，`double_t` 可用于命名一种同时具有 IEEE 754 双精度格式的精度和范围的类型，条件是使用严格的双精度计算。与相应命名的环境参数宏一起，这样的方案就可以支持上述的所有五种选择，并允许程序员轻松而明确地指出其程序所需的浮点语义。

扩展精度的语言支持必须这样复杂吗？在单精度 / 双精度系统上，上面列出的五种选择中的四种是一致的，因此不必区分快速和精确宽度类型。但是，基于扩展的系统会给选择带来困难：它们既不支持纯双精度计算，也不支持纯扩展精度计算（其效率与这两种运算的混合一样高），而且不同的程序要求不同的混合程度。此外，选择何时使用扩展精度不应该留给编译器编写人员，他们通常受到测试基准的影响（有时得到数值分析员的直接通知），将浮点运算视为“本身不精确”，因此不值得也不能够预知整数运算。相反，必须由程序员进行选择，他们将需要能够表达其选择的语言。

## 结束语

上述评论并非旨在贬低基于扩展的系统，而是要揭示几个谬论，第一个谬论是所有 IEEE 754 系统都必须为同一程序提供完全相同的结果。我们着重说明了基于扩展的系统与单精度 / 双精度系统之间的不同，但是在其中每个系列内的系统之间存在进一步的不同。例如，一些单精度 / 双精度系统提供单个指令将两个数相乘并与第三个数相加，只进行一次最终舍入。此运算称为 *合并的乘-加*，会导致同一程序在不同的单精度 / 双精度系统上产生不同的结果；与扩展精度一样，它甚至会导致同一程序在同一系统上产生不同的结果，这取决于是否使用它和何时使用它。（合并的乘-加也可以阻止定理 6 的溢出过程，尽管能够以不可移植的方式使用它来执行多精度乘法运算，而不必进行拆分。）尽管 IEEE 标准没有预期到这样的操作，但是它与此一致：中间乘积被提供给超出用户控制的“目标”，它的宽度足以精确容纳该乘积，最终的和被正确舍入以适合其单精度或双精度目标。

但是，以下意见是吸引人的：IEEE 754 精确规定给定的程序所必须提供的结果。许多程序员喜欢相信他们可以理解程序的行为，并证明它将正确运行，而不涉及编译它的编译器或运行它的计算机。在许多方面，对于计算机系统和程序设计语言的设计人员来说，支持此意见是一个需要花费精力的目标。可惜的是，当涉及到浮点运算时，该目标几乎是不可能实现的。IEEE 标准的制定者知道他们不试图实现该目标。因此，尽管在整个计算机行业几乎都符合 IEEE 754 标准（的大部分内容），但是开发可移植软件的程序员必须继续处理不可预知的浮点计算。

如果程序员利用 IEEE 754 的各种特性，则他们将需要使浮点运算可预知的程序设计语言。C99 标准在一定程度上改进了可预知性，代价是要求程序员编写多个版本的程序，对于每个 FLT\_EVAL\_METHOD 都编写一个版本。现在还不知道：将来的语言是否将改为选择允许程序员使用明确表达程序对 IEEE 754 语义的依赖程度的语法，来编写单个程序。基于扩展的现有系统对该前景构成了威胁，因为它们使我们很容易假定编译器和硬件对应该如何给定系统上执行计算的了解应该比程序员更好。该假定是第二个谬论：计算的结果所需的准确度不取决于产生该准确度的计算机，而是仅取决于从它得出的结论：在程序员、编译器和硬件中，最多只有程序员才能知道那些结论可能是什么。



# 符合标准

---

用在 Solaris 环境的 Sun Studio 编译器产品中的编译器、头文件和库支持多个标准：系统 V 接口定义 (SVID) 第 3 版、X/Open 和 ANSI C。相应地，已经对数学库 `libm` 和相关文件进行了修改，以使 C 程序符合这些标准。用户程序通常不受影响，因为差别主要涉及异常处理。

---

## SVID 历史

要了解按照 SVID 和 IEEE 标准所代表观点进行的异常处理之间的差别，必须考察这两种标准的发展背景。SVID 中的很多理念源于早期的 UNIX（即第一次在大型计算机上实现 UNIX 时）。这些早期环境的共性是：有理数浮点运算 `+`、`-`、`*` 和 `/` 是原子机器指令，而 `sqrt`、到浮点格式整数值的转换以及初等超越函数都是由很多原子机器指令组成的子例程。

因为这些环境以各种不同的方式来处理浮点异常，所以只能通过在每个原子浮点指令前后检查软件中的参数和结果来强制实施一致性。因为这会对性能产生非常大的影响，所以 SVID 没有指定浮点异常（如被零除或上溢）的影响。

与单个原子浮点指令相比，子例程实现的运算速度很慢；对参数和结果进行的额外错误检查对性能几乎没有影响；因此，SVID 要求进行此类检查。在检测到异常时，就会指定缺省结果（对于不正确的操作数，将 `errno` 设置为 `EDOM`；对于出现上溢或下溢的结果，则设置为 `ERANGE`），并使用包含异常细节的记录来调用函数 `matherr()`。在最初为其开发 UNIX 的机器上，这几乎不会产生什么系统开销，但该值相对很小，因为根本没有指定基本运算 `+`、`-`、`*` 和 `/` 中的最常见异常。

---

## IEEE 754 历史

IEEE 标准明确指出保持与以前实现的兼容性并不是目的。相反，在制订异常处理方案时考虑到了效率和用户要求。这种方案在简单有理数运算（+、-、\* 和 /）和更复杂运算（如求余、求平方根和格式转换）中是一致的。虽然此标准没有指定超越函数，但标准的制订者预见到：相同的异常处理方案适用于兼容系统中的初等超越函数。

IEEE 异常处理的元素包括适用的缺省结果以及仅在提前请求的情况下中止计算。

---

## SVID 将来的发展方向

当前的 SVID（“第 3 版”或“SVR4”）为将来的发展指明了某些方向。其中之一是与 IEEE 标准的兼容性。特别地，SVID 的将来版本使用 HUGE\_VAL（它在 IEEE 系统上为无穷大）替代 HUGE（它是一个极大的有限数）。例如，返回 HUGE\_VAL 作为浮点上溢的结果。表 E-1 上 IEEE 列中的值即是 libm 函数为产生异常的输入参数返回的值。此外，不再需要设置 errno。

---

## SVID 实现

以下 libm 函数提供对应于 SVID 的操作数或结果检查。在从通过 -xlibmil 使用 libm 内置扩展模板的 C 程序中调用时，sqrt 函数是唯一不符合 SVID 的函数，因为这会导致使用平方根硬件指令 fsqrt[sd] 替代函数调用。

表 E-1 异常情况和 libm 函数

| 函数             | errno | 错误消息   | SVID | X/Open | IEEE *     |
|----------------|-------|--------|------|--------|------------|
| 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        |

\* -xc99=lib 设置强制对于异常行为返回 IEEE 754 风格的值（即引发异常，而不是设置 errno）。



表 E-1 异常情况和 libm 函数 (续)

| 函数                | errno       | 错误消息   | SVID              | X/Open                  | IEEE *     |
|-------------------|-------------|--------|-------------------|-------------------------|------------|
| atanh(+/-1)       | EDOM/ERANGE | SING   | +/-HUGE<br>(EDOM) | +/-HUGE_VAL<br>(ERANGE) | +/-无穷大     |
| cosh overflow     | ERANGE      | -      | HUGE              | HUGE_VAL                | 无穷大        |
| exp overflow      | ERANGE      | -      | HUGE              | HUGE_VAL                | 无穷大        |
| exp underflow     | ERANGE      | -      | 0.0               | 0.0                     | 0.0        |
| fmod(x,0)         | EDOM        | DOMAIN | x                 | NaN                     | NaN        |
| gamma(0 或负整数)     | EDOM        | SING   | HUGE              | HUGE_VAL                | 无穷大        |
| gamma overflow    | ERANGE      | -      | HUGE              | HUGE_VAL                | 无穷大        |
| hypot overflow    | ERANGE      | -      | HUGE              | HUGE_VAL                | 无穷大        |
| j0( x  > X_TLOSS) | ERANGE      | TLOSS  | 0.0               | 0.0                     | 正确答案       |
| j1( x  > X_TLOSS) | ERANGE      | TLOSS  | 0.0               | 0.0                     | 正确答案       |
| jn( x  > X_TLOSS) | ERANGE      | TLOSS  | 0.0               | 0.0                     | 正确答案       |
| lgamma(0 或负整数)    | EDOM        | SING   | HUGE              | HUGE_VAL                | 无穷大        |
| lgamma overflow   | ERANGE      | -      | HUGE              | HUGE_VAL                | 无穷大        |
| log(0)            | EDOM/ERANGE | SING   | -HUGE<br>(EDOM)   | -HUGE_VAL<br>(ERANGE)   | -无穷大       |
| log(x<0)          | EDOM        | DOMAIN | -HUGE             | -HUGE_VAL               | NaN        |
| log10(0)          | EDOM/ERANGE | SING   | -HUGE<br>(EDOM)   | -HUGE_VAL<br>(ERANGE)   | -无穷大       |
| log10(x<0)        | EDOM        | DOMAIN | -HUGE             | -HUGE_VAL               | NaN        |
| log1p(-1)         | EDOM/ERANGE | SING   | -HUGE<br>(EDOM)   | -HUGE_VAL<br>(ERANGE)   | -无穷大       |
| log1p(x<-1)       | EDOM        | DOMAIN | NaN               | NaN                     | NaN        |
| pow(0,0)          | EDOM        | DOMAIN | 0.0               | 1.0 (没有错误)              | 1.0 (没有错误) |
| pow(NaN,0)        | EDOM        | DOMAIN | NaN               | NaN                     | 1.0 (没有错误) |
| pow(0, 负数)        | EDOM        | DOMAIN | 0.0               | -HUGE_VAL               | +/-无穷大     |
| pow(负数, 非负整数)     | EDOM        | DOMAIN | 0.0               | NaN                     | NaN        |
| pow overflow      | ERANGE      | -      | +/-HUGE           | +/-HUGE_VAL             | +/-无穷大     |
| pow underflow     | ERANGE      | -      | +/-0.0            | +/-0.0                  | +/-0.0     |
| remainder(x,0)    | EDOM        | DOMAIN | NaN               | NaN                     | NaN        |
| scalb overflow    | ERANGE      | -      | +/-HUGE_VAL       | +/-HUGE_VAL             | +/-无穷大     |

\* -xc99=lib 设置强制对于异常行为返回 IEEE 754 风格的值 (即引发异常, 而不是设置 errno)。

表 E-1 异常情况和 libm 函数 (续)

| 函数                | errno  | 错误消息   | SVID    | X/Open      | IEEE * |
|-------------------|--------|--------|---------|-------------|--------|
| scalb underflow   | ERANGE | -      | + -0.0  | + -0.0      | + -0.0 |
| sinh overflow     | ERANGE | -      | + -HUGE | + -HUGE_VAL | + -无穷大 |
| sqrt(x<0)         | EDOM   | DOMAIN | 0.0     | NaN         | NaN    |
| y0(0)             | EDOM   | DOMAIN | -HUGE   | -HUGE_VAL   | -无穷大   |
| y0(x<0)           | EDOM   | DOMAIN | -HUGE   | -HUGE_VAL   | NaN    |
| y0(x > X_TLOSS)   | ERANGE | TLOSS  | 0.0     | 0.0         | 正确答案   |
| y1(0)             | EDOM   | DOMAIN | -HUGE   | -HUGE_VAL   | -无穷大   |
| y1(x<0)           | EDOM   | DOMAIN | -HUGE   | -HUGE_VAL   | NaN    |
| y1(x > X_TLOSS)   | ERANGE | TLOSS  | 0.0     | 0.0         | 正确答案   |
| yn(n,0)           | EDOM   | DOMAIN | -HUGE   | -HUGE_VAL   | -无穷大   |
| yn(n,x<0)         | EDOM   | DOMAIN | -HUGE   | -HUGE_VAL   | NaN    |
| yn(n, x> X_TLOSS) | ERANGE | TLOSS  | 0.0     | 0.0         | 正确答案   |

\* -xc99=lib 设置强制对于异常行为返回 IEEE 754 风格的值 (即引发异常, 而不是设置 errno)。

## 有关异常情况和 libm 函数的一般说明

表 E-1 列出所有受这些标准影响的 libm 函数。值 X\_TLOSS 是在 <values.h> 中定义的。SVID 要求 <math.h> 将 HUGE 定义为 MAXFLOAT, 它约等于 3.4e+38。在 libc 中将 HUGE\_VAL 定义为无穷大。errno 是 C 和 C++ 程序可以访问的一个全局变量。

<errno.h> 为 errno 定义 120 个左右可能的值; 数学库使用的两个值为 EDOM (用于域错误) 和 ERANGE (用于范围错误)。请参见 intro(3) 和 perror(3)。

- 在 -xc99=lib 设置中, libm 函数的操作方式就如同已针对异常情况指定了 -xlibmieee 一样。即“强制对于异常情况中的数学例程返回 IEEE 754 风格的值”。这还意味着 -Xs、-Xt、-Xa 或 -Xc 隐式指定的任何库行为 (按照“一般说明”中的描述) 都不会生效。

**注意** - 请注意 Solaris 在 /usr/lib 中提供的标准系统库尚不符合 1999 ISO/IEC C 标准。在符合此标准前, 无法设置 -xc99=lib。

- 在这些开关中, ANSI C 编译器开关 -Xt、-Xa、-Xc、-Xs 控制编译器强制实施的符合标准的级别。有关这些开关的描述, 请参见 cc(1)。
- 就 libm 而言, -Xt 和 -Xa 分别导致 SVID 和 X/Open 行为。-Xc 对应于严格的 ANSI C 行为。

在指定额外的开关 `-<x>libmieee` 后，可返回 IEEE 754 风格的值。在 Solaris 2.6、Solaris 7 和 Solaris 8 上，`libm` 和 `libsunmath` 的缺省行为是符合 SVID。

- 对于严格的 ANSI C (`-xc`)，始终设置 `errno`，不调用 `matherr()`，并且返回 X/Open 值。
- 对于 SVID (`-xt` 或 `-xs`)，使用有关异常的信息来调用函数 `matherr()`。这包括缺省 SVID 返回值。

用户提供的 `matherr()` 可能会改变返回值；请参见 `matherr(3m)`。如果没有用户提供的 `matherr()`，则 `libm` 设置 `errno`，可能将一条消息输出到标准错误中，并返回表 E-1 上 SVID 列中列出的值。

- 对于 X/Open (`-xa`)，由于调用了 `matherr()` 并相应地设置了 `errno`，其行为与 SVID 相同。但是，不会将任何错误消息写入到标准错误中，在很多情况下，X/Open 返回值与 IEEE 返回值相同。
- 对于 `libm` 异常处理，`-xs` 与 `-xt` 的行为方式相同。即，用 `-xs` 编译的程序使用表 E-1 中列出的 `libm` 函数的 SVID 兼容版本。
- 为了提高效率，如果 `sqrt` 遇到负参数，则使用内置硬件浮点编译的程序并不进行设置 EDOM 或调用 `matherr()` 所需的额外检查。在本来可能设置 EDOM 的场合，返回 NaN 作为函数值。

因此，使用 `fsqrt[sd]` 指令替代 `sqrt()` 函数调用的 C 程序符合 IEEE 浮点标准，但可能不再符合系统 V 接口定义的错误处理要求。

## libm 的说明

SVID 指定了两个浮点异常：PLOSS（损失部分有效数字）和 TLOSS（损失全部有效数字）。与 `sqrt(-1)` 不同，这两个异常没有内在的数学含义；与 `exp(+10000)` 不同的是，这两个异常没有反映浮点存储格式的内在限制。

但 PLOSS 和 TLOSS 反映了 `fmod` 和三角函数（准确度在明确界限处突然下降）的特定算法的限制。

与大多数 IEEE 实现一样，`libm` 算法没有出现此类突然下降问题，因此不会用信号通知 PLOSS。为了符合 SVID 规定，对于较大的输入参数，虽然可以安全地计算准确的结果，但贝塞尔函数仍用信号通知 TLOSS。

`sin`、`cos` 和 `tan` 实现处理在无穷大处出现的基本奇点的方式与其它基本奇点一样：为无穷大参数返回 NaN 并设置 EDOM。

同样地，SVID 规定，如果 `x/y` 出现上溢，则 `fmod(x,y)` 为零，但 `fmod` 的 `libm` 实现（从 IEEE 求余函数派生出来的）并不显式地计算 `x/y`，因此始终提供精确的结果。

---

# 符合 LIA-1

在本节中，LIA-1 指的是 ISO/IEC 10967-1:1994 信息技术 - 与语言无关的算法 - 第 1 部分：整数和浮点运算。

Sun Studio 编译器版本包含的 C 和 Fortran 95 编译器（cc 和 f95）在以下方面符合 LIA-1（段落字母对应于 LIA-1 第 8 节中的段落字母）：

a. 类型 (LIA 5.1): 符合 LIA-1 的类型为 C int 和 Fortran INTEGER。其它类型也可能符合该标准，但此处没有对它们进行说明。有关特定语言的详细规范，请参见公认的语言标准组织即将发布的 LIA-1 语言汇编材料。

b. 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)
```

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，也可按以下代码使用 modaI(x,y)：

```
int modaI(int x, int y) {
    int t = x % y;
    if (y < 0 && t > 0)
        t -= y;
    else if (y > 0 && t < 0)
        t += y;
    return t;
}
```

或者:

```
integer function modaI(x, y)
integer x, y, t
t = mod(x, y)
if (y .lt. 0 .and. t .gt.0) t = t - y
if (y .gt.0 .and. t .lt. 0) t = t + y
modaI = t
return
end
```

i. 表示法 (LIA 5.1.3): 下表显示用于识别 LIA 整数运算的表示法。

表 E-2 符合 LIA-1 - 表示法

| LIA        | C  | Fortran (如果不同的话) |
|------------|--|------------------|
| addI(x,y)  | x+y  |                  |
| subI(x,y)  | x-y  |                  |
| mulI(x,y)  | x*y  |                  |
| divtI(x,y) | x/y  |                  |
| remtI(x,y) | x%y  | mod(x,y)         |
| modaI(x,y) | 参见以上内容   |                  |
| negI(x)    | -x   |                  |
| absI(x)    | #include <stdlib.h><br>abs(x)                      | abs(x)           |
| signI(x)   | #define signI(x) (x > 0<br>? 1 : (x < 0 ? -1 : 0)) | 参见以下内容           |
| eqI(x,y)   | x==y   | x.eq.y           |
| neqI(x,y)  | x!=y   | x.ne.y           |
| lssI(x,y)  | x<y  | x.lt.y           |
| leqI(x,y)  | x<=y   | x.le.y           |
| gtrI(x,y)  | x>y  | x.gt.y           |
| geqI(x,y)  | x>=y   | x.ge.y           |

以下代码显示 signI(x) 的 Fortran 表示法。

```
integer function signi(x)
integer x, t
```

```
if (x .gt.0) t=1
if (x .lt. 0) t=-1
if (x .eq.0) t=0
return
end
```

j. 计算表达式结果：在缺省情况下，如果未指定优化，则以 `int (C)` 或 `INTEGER (Fortran)` 精度计算表达式的值。考虑括号。未指定关联无括号表达式（如 `a + b + c` 或 `a * b * c`）的计算顺序。

k. 获取参数的方法：在源代码中包含上述定义。

n. 通知：整数异常为 `x/0` 和 `x%0` 或 `mod(x,0)`。在缺省情况下，这些表达式生成 `SIGFPE`。如果未给 `SIGFPE` 指定信号处理程序，则进程终止并转储内存。

o. 选择机制：可以使用 `signal(3)` 或 `signal(3F)` 启用 `SIGFPE` 的用户异常处理。

### 参考资料

---

下面的手册提供有关 SPARC® 浮点硬件的更多信息：

*SPARC Architecture Manual, Version 9*, PTR Prentice Hall, New Jersey, 1994.

其余参考资料按章节进行组织。最后介绍有关如何获得标准文档和测试程序的信息。

---

### 第 2 章：“IEEE 算法”

Cody et al., “A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic,” *IEEE Computer*, August 1984.

Coonen, J.T., “An Implementation Guide to a Proposed Standard for Floating Point Arithmetic”, *Computer*, Vol. 13, No. 1, Jan. 1980, pp 68-79.

Demmel, J., “Underflow and the Reliability of Numerical Software”, *SIAM J. Scientific Statistical Computing*, Volume 5 (1984), 887-919.

Hough, D., “Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic”, *Computer*, Vol. 13, No. 1, Jan. 1980, pp 70-74.

Kahan, W., and Coonen, J.T., “The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments”, published in *The Relationship between Numerical Computation and Programming Languages*, Reid, J.K., (editor), North-Holland Publishing Company, 1982.

Kahan, W., “Implementation of Algorithms”, *Computer Science Technical Report No. 20*, University of California, Berkeley CA, 1973. Available from National Technical Information Service, NTIS Document No. AD-769 124 (339 pages), 1-703-487-4650 (ordinary orders) or 1-800-336-4700 (rush orders.)

Karpinski, R., “Paranoia: a Floating-Point Benchmark”, *Byte*, February 1985.

Knuth, D.E., *The Art of Computer Programming, Vol.2: Semi-Numerical Algorithms*, Addison-Wesley, Reading, Mass, 1969, p 195.

Linnainmaa, S., "Combatting the effects of Underflow and Overflow in Determining Real Roots of Polynomials", *SIGNUM Newsletter* 16, (1981), 11-16.

Rump, S.M., "How Reliable are Results of Computers?", translation of "Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?", *Jahrbuch Überblicke Mathematik* 1983, pp 163-168, C Bibliographisches Institut AG 1984.

Sterbenz, P, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974. (Out of print; most university libraries have copies.)

Stevenson, D. et al., Cody, W., Hough, D. Coonen, J., various papers proposing and analyzing a draft standard for binary floating-point arithmetic, *IEEE Computer*, March 1981.

*The Proposed IEEE Floating-Point Standard*, special issue of the *ACM SIGNUM Newsletter*, October 1979.

---

### 第 3 章：“数学库”

Cody, William J. and Waite, William, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1980.

Coonen, J.T., *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, PhD Dissertation, University of California, Berkeley, 1984.

Tang, Peter Ping Tak, *Some Software Implementations of the Functions Sin and Cos*, Technical Report ANL-90/3, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1990.

Tang, Peter Ping Tak, *Table-driven Implementations of the Exponential Function EXPM1 in IEEE Floating-Point Arithmetic*, Preprint MCS-P125-0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February, 1990.

Tang, Peter Ping Tak, *Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic*, *ACM Transactions on Mathematical Software*, Vol. 15, No. 2, June 1989, pp 144-157 communication, July 18, 1988.

Tang, Peter Ping Tak, *Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic*, preprint MCS-P55-0289, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1989 (to appear in *ACM Trans. on Math. Soft.*)



Park, Stephen K. and Miller, Keith W., "Random Number Generators: Good Ones Are Hard To Find", *Communications of the ACM*, Vol. 31, No. 10, October 1988, pp 1192 - 1201.

---

## 第 4 章：“异常和信号处理”

Coonen, J.T, "Underflow and the Denormalized Numbers", *Computer*, 14, No. 3, March 1981, pp 75-87.

Demmel, J., and X. Li, "Faster Numerical Algorithms via Exception Handling", *IEEE Trans. Comput.* Vol. 48, No. 8, August 1994, pp 983-992.

Kahan, W., "A Survey of Error Analysis", *Information Processing 71*, North-Holland, Amsterdam, 1972, pp 1214-1239.

---

## 附录 B：“SPARC 行为和实现”

下面的制造商文档提供有关浮点硬件和主要处理器芯片的更多信息。这些信息按系统结构进行组织。

Texas Instruments, *SN74ACT8800 Family, 32-Bit CMOS Processor Building Blocks: Data Manual*, 1st edition, Texas Instruments Incorporated, 1988.

Weitek, *WTL 3170 Floating Point Coprocessor: Preliminary Data*, 1988, published by Weitek Corporation, 1060 E. Arques Avenue, Sunnyvale, CA 94086.

Weitek, *WTL 1164/WTL 1165 64-bit IEEE Floating Point Multiplier/Divider and ALU: Preliminary Data*, 1986, published by Weitek Corporation, 1060 E. Arques Avenue, Sunnyvale, CA 94086.

---

## 标准

*American National Standard for Information Systems, ISO/IEC 9899:1990 Programming Languages - C*, American National Standards Institute, 1430 Broadway, New York, NY 10018.

*American National Standard for Information Systems ISO/IEC 9899:1999 Programming Languages - C*, American National Standards Institute, 1430 Broadway, New York, NY 10018.

*IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985 (IEEE 754)*, published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1985.

*IEEE Standard Glossary of Mathematics of Computing Terminology, ANSI/IEEE Std 1084-1986*, published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1986.

*IEEE Standard Portable Operating System Interface for Computer Environments (POSIX)*, IEEE Std 1003.1-1988, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017.

*System V Application Binary Interface (ABI)*, AT&T (1-800-432-6600), 1989.

*SPARC System V ABI Supplement (SPARC ABI)*, AT&T (1-800-432-6600), 1990.

*System V Interface Definition*, 3rd edition, (SVID89, or SVID Issue 3), Volumes I-IV, Part number 320-135, AT&T (1-800-432-6600), 1989.

*X/OPEN Portability Guide*, Set of 7 Volumes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1989.

---

## 测试程序

浮点算法和数学库的许多测试程序都可以从 `ucbtest` 包中的 `Netlib` 中找到。这些程序包括各种版本的 `Paranoia`、Z. Alex Liu 的 `Berkeley Elementary Function` 测试程序、IEEE 测试矢量和基于 W. Kahan 教授的数论方法的程序（这些程序针对正确舍入的乘、除和平方根运算生成高难度的测试实例）。

`ucbtest` 位于 <http://www.netlib.org/fp/ucbtest.tgz>。

# 词汇表

---

本词汇表介绍计算机浮点算法术语。它还介绍了与并行处理有关的术语和缩略词。

附加到术语后面的符号“||”指明该术语与并行处理有关。

- 2 的补码** || 可使用以下方法构成二进制数基数的补码：用 1 减每个数字，然后在最低有效数位上加 1，并执行任何所需的进位。例如，1101 的 2 的补码为 0011。
- 按需求驱动的数据流** || 当另一个也处于激活状态的任务需要某个任务的结果时，处理器便激活该任务，以便执行操作（如图形约简模型）。图形约简程序由可简化的表达式组成，在计算过程中可使用其计算值来替代这些表达式。通常，约简是并行完成的 - 除以前约简数据的可用性外，任何其它因素均不能禁止进行并行约简。另请参见 *控制流模型*、*数据流模型*。
- 绑定线程** || 对于 Solaris 线程而言，永久性分配给特定 LWP 的线程称为绑定线程。可以实时地调度绑定线程，并使之具有比系统中的所有其它活动线程更高的优先级，而不是仅比进程中的线程优先级高。LWP 是一个实体，可使用与任何 UNIX 进程相同的缺省调度优先级对它进行调度。
- 包交换** || 在共享内存体系结构中，缓存彼此之间进行通信以及与主内存之间进行通信使用的机制。在包交换中，可以将通信分为很小的段（称为“包”），可在总线上多路传输这些包。缓存和内存硬件可通过包中包含的标识来确定是将包发送到该硬件，还是将包发送到其最终目标。包交换允许多路传输总线通信，并将无序（没有顺序）包放在总线上。可在目标（缓存或主内存）处重新组装无序包。另请参见 *缓存*、*共享内存*。
- 并发进程** || 在多处理器上并行执行的进程，或在单处理器上异步执行的进程。并发进程可以彼此之间进行交互，一个进程可以暂停执行，直至从另一个进程收到信息或发生外部事件时再继续执行。另请参见 *进程*、*顺序执行的进程*。
- 并行操作** || 并行执行两个或多个活动线程或进程。在单处理器上，可通过在线程之间快速切换，使表面上看起来是同时完成这两个线程。在多处理器系统上，可实现真正的并行执行。另请参见 *异步控制*、*多处理器系统*、*线程*。
- 并行处理** || 在多处理器系统中，可以实现真正的并行执行，其中可同时存在大量活动的线程或进程。另请参见 *并发性*、*多处理器系统*、*多线程处理*、*单处理器*。
- 并行性** || 请参见 *并发进程*、*多线程处理*。

|        |  |
|--------|--|
| 常见异常   | 上溢、无效和除法这三种浮点异常统称为 <code>ieee_flags(3m)</code> 和 <code>ieee_handler(3m)</code> 的常见异常。之所以将它们称为常见异常，是因为通常将它们捕获为误差。   |
| 次正规数   | 在 IEEE 算法中，偏置指数为 0 的非零浮点数。次正规数是介于零和最小正规数之间的数。  |
| 单程序多数据 | 一种异步并行方式，在没有锁步协同的情况下同时处理不同的数据。在 SPMD 中，处理器可以同时执行不同的指令；例如，不同的 <code>if-then-else</code> 语句分支。   |
| 单处理器系统 | 单处理器系统在任意给定时间只有一个处理器处于活动状态。此单处理器可以运行多线程应用程序以及传统的单指令单数据模型。另请参见 <i>多线程处理</i> 、 <i>单指令单数据</i> 、 <i>单锁定策略</i> 。  |
| 单精度    | 使用一个计算机字元来表示一个数。   |
| 单锁定策略  | 在单锁定策略中，只要应用程序中有线程正在运行，线程会获取应用程序范围内的互斥锁定，然后在该线程阻塞前释放锁定。单锁定策略要求系统中所有模块和库的协作使用单个锁定进行同步。因为在任何给定时间内只能有一个线程可以访问共享数据，所以每个线程都具有一致的内存视图。此策略在单处理器中很有效，条件是在释放锁定前将共享内存置于一致的状态，并且经常释放锁定以使其它线程能够运行。并且，在单处理器系统中，如果在大多数 I/O 操作过程中锁定没有撤消，并行操作效率就会降低。不能在多处理器系统中应用单锁定策略。 |
| 单指令单数据 | 在此传统单处理器模型中，获取和执行指令序列的单个处理器对指令内部指定的数据项进行运算。这就是原始的冯·诺伊曼计算机运算模型。   |
| 单指令多数据 | 此系统模型中有多个处理元素，但它们用于同时执行相同的指令；即，使用一个程序计数器顺序执行程序的一个副本。SIMD 在解决以下问题方面特别有用：包含大量需要统一更新的数据；例如，一般的数值计算。很多科学和工程方面的应用程序（例如，图像处理、粒子模拟以及有限元方法）自然归类为 SIMD 范例。另请参见 <i>数组处理</i> 、 <i>流水线</i> 、 <i>矢量处理</i> 。   |
| 电路交换   | 缓存彼此之间进行通信以及与主内存之间进行通信使用的机制。专用连接（电路）是在缓存之间以及缓存与主内存之间建立的。在建立某个电路时，不能通过总线传输任何其它通信。   |
| 多处理器   | 请参见 <i>多处理器系统</i> 。  |
| 多处理器系统 | 在此系统中，在任意给定时刻都可以有多个活动的处理器。当这些处理器各自执行单独的进程时，它们完全异步地运行。但是，当处理器访问关键的系统资源或关键的系统代码区域时，它们之间必须进行同步。另请参见 <i>关键区域</i> 、 <i>关键资源</i> 、 <i>多线程处理</i> 、 <i>单处理器系统</i> 。  |
| 多处理器总线 | 在共享内存多处理器机器中，每个 CPU 和缓存模块通过一条总线连接起来，该总线还包含内存和 IO 连接。总线强制实施缓存一致性协议。另请参见 <i>缓存</i> 、 <i>一致性</i> 、 <i>Mbus</i> 、 <i>XDBus</i> 。   |
| 多次读单次写 | 在并发环境中，第一个进行数据写访问的进程对它进行独占访问，因而无法进行并发写访问或同时进行读取和写入访问。但是，可以由多个阅读器读取该数据。   |

- 多任务处理** 在单处理器系统中，表面上看起来大量线程是并行运行的。这是通过快速切换线程完成的。
- 多线程安全的** 在 Solaris 环境中，库内的函数调用是多线程安全的或非多线程安全的；多线程安全的代码也称为“可重入”代码。即，几个线程可以同时调用模块中的一个给定函数，并且由函数代码来处理该调用。假设条件是仅由模块函数来访问线程之间共享的数据。如果模块客户机可以使用易变的全局数据，则还必须在接口中显示相应的锁定。再者，除非假定客户机在适当的时间一致地使用锁定，否则无法将模块函数变为可重入版本。另请参见 *单锁定策略*。
- 多线程处理** 应用程序可以同时有多个活动线程或处理器。多线程的应用程序可以在单处理器系统和多处理器系统中运行。另请参见 *绑定线程*、*多线程安全的*、*单锁定策略*、*线程*、*非绑定线程*、*单处理器*。
- 多指令多数据** 在此系统模型中，多个处理器可以同时不同的数据执行不同的指令。再者，这些处理器的操作方式具有很大的独立性，就好像它们是单独的计算机一样。它们没有中央控制器，并且通常不使用锁步操作模式。大多数实际银行使用的就是这种运行方式。出纳员彼此之间并不商议，他们也不同步执行每个交易的每个步骤。相反，他们独立工作，直至发生数据访问冲突时为止。在处理交易时并不考虑时间或客户顺序。但必须显式地禁止客户 A 和 B 同时访问共有的 AB 帐户余额。MIMD 依靠称为“锁定”的同步机制来协调对共享资源的访问。另请参见 *互斥*、*互斥锁定*、*信号锁定*、*单锁策略*和*转锁*。
- 二进制间距** 任意两个连续的 2 的幂指数的间距。
- 范例** 用于描述解决问题的计算机方案的现实模型。范例提供了理解和解决现实问题的环境。因为范例是一个模型，所以它从现实情况中抽象出问题的细节，因而使问题更容易解决。但是，就像所有抽象一样，模型可能并不准确，这是因为它只对现实情况做近似处理。另请参见 *多指令多数据*、*单指令多数据*、*单指令单数据*、*单程序多数据*。
- 非绑定线程** 对于 Solaris 线程而言，在 LWP 池上调度的线程称为非绑定线程。线程库调用并分配 LWP 以执行可运行的线程。如果线程在同步机制中变为阻塞线程（如互斥锁定），则在进程内存中保存线程的状态。然后，线程库将另一个线程分配给 LWP。另请参见 *绑定线程*、*多线程处理*、*线程*。
- 非正规化数** 次正规数的旧名称。
- 分布式内存体系结构** 互连网络拓扑中每个节点的本地内存和处理器的组合。每个处理器只能直接访问系统总内存的一部分。任意两个处理器之间使用消息传递进行通信，并且没有全局的共享内存。因此，当必须共享某个数据结构时，程序会向拥有该结构的进程发出发送 / 接收消息。另请参见 *进程间通信*、*消息传递*。
- 浮点数系统** 用于表示实数子集的一种系统，可表示数之间的间距不是固定的绝对常数。此类系统可以通过基数、符号、有效数字和指数（通常为偏置指数）来表征。该数的值是其有效数字和基数的无偏置指数幂的符号乘积。
- 共享内存体系结构** 在总线连接的多处理器系统中，进程或线程通过所有处理器共享的全局内存进行通信。此共享数据段放在协作进程的地址空间中（在其专用数据段和堆栈段之间）。`fork()` 产生的后续任务复制其地址空间中除共享数据段以外的所有数据段。共享内存要求程序语言扩展和库例程支持该模型。

- 关键区域** || 代码中不可分割的一部分，每次只能由一个线程执行，执行时不能被其他线程中断；例如，访问共享变量的代码。另请参见 *互斥*、*互斥锁定*、*信号锁定*、*单锁定策略*、*转锁*。
- 关键资源** || 在任何给定时间至多有一个线程可以使用的资源。如果要求几个异步线程协调它们对关键资源的访问，则它们使用同步机制来执行此操作。另请参见 *互斥*、*互斥锁定*、*信号锁定*、*单锁定策略*、*转锁*。
- 关联性** || 请参见 *缓存*、*直接映射的缓存*、*完全关联的缓存*、*集关联缓存*。
- 互斥** || 在并发环境中，线程在没有竞争线程访问的情况下更新关键资源的能力。另请参见 *关键区域*、*关键资源*。
- 互斥锁定** || 实现互斥锁定机制的同步变量。另请参见 *条件变量*、*互斥*。
- 互连网络拓扑** || 互连拓扑描述了处理器的连接方式。所有网络都是由交换机组成的，这些交换机链接到处理器 - 内存节点和其它交换机上。共有四种基本形式的拓扑：星形、环形、总线以及完全连接的网路。星形拓扑由一个单独的集线器处理器组成，其它处理器直接连接到该集线器上，非集线器处理器彼此并不直接相连。在环形拓扑中，所有处理器均在一个环上，通信通常沿环的一个方向进行。总线拓扑是非循环的，并且所有节点彼此相连；因此通信传输是双向进行的，并且需要进行某种形式的仲裁以确定在任意特定时刻哪个处理器可以使用总线。在完全连接（交叉）的网络中，每个处理器与每个其它处理器之间具有双向链接。
- 缓存** || 很小、速度很快并由硬件控制的存储器，它用作处理器和主内存之间的缓冲区。缓存包含最近使用的指令和数据内存地址的副本（地址和内容）。每个地址引用都先经过缓存。如果缓存中没有所需的指令或数据，则发生缓存未命中情况。可通过总线将内容从主内存提取到正在执行的指令所指定的 CPU 寄存器中，同时将一个副本写入到缓存中。稍后可能会使用相同的地址，如果发生这种情况，则在缓存中找到该地址，从而发生缓存命中情况。如果将数据写入到该地址中，则硬件不仅写入到缓存中，而且还生成一个到主内存的直写。
- 另请参见 *关联性*、*电路交换*、*直接映射的缓存*、*完全关联的缓存*、*MBus*、*包交换*、*集关联缓存*、*回写*、*直写*、*XDBus*。
- 缓存局域性** || 程序并不以相同的概率同时访问其所有代码或数据。如果将最近访问的信息存放在缓存中，则可增大本地找到信息的可能性，从而无需访问内存。局域性原则规定程序在任何瞬间访问其地址空间中相对较小的一部分。共有两种不同类型的局域性：时间和空间。
- 时间局域性是指重复使用最近访问的项目的倾向性。例如，大多数程序包含循环，因此可能会重复访问指令和数据。时间局域性将最近访问的项目（靠近处理器）保存在缓存中，而不是要求进行内存访问。另请参见 *缓存*、*竞争性缓存*、*伪共享*、*写无效*、*写更新*。
- 空间局域性是指所引用项目的地址接近于其它最近访问的项目的倾向性。例如，数组元素或记录访问显示出自然的空间局域性。缓存使用以下方法来利用空间局域性：将块（多个连续的字元）从内存移到缓存中并靠近处理器。另请参见 *缓存*、*竞争性缓存*、*伪共享*、*写无效*、*写更新*。

- 环绕式数字** 在 IEEE 算法中，可通过以下方法用某个值创建的数：为该值的指数增加固定偏移以使其环绕值位于正规数范围内；如果不进行此类处理，该值就会出现上溢或下溢。目前的 SPARC 工作站上不会产生环绕结果。
- 回写** 用于在缓存和主内存之间保持一致性的写入策略。回写（也称为复制回或存储在）仅写入到本地缓存中的块。写入是在缓存内存的速度下进行的。只有在另一个处理器引用相应的内存地址时，才会将修改后的缓存块写入到主内存中。处理器可以在缓存块中写入多次，并且仅在引用时才将其写入到主内存中。因为每个写入并不经过内存，所以回写减少了对总线带宽的需求。另请参见 *缓存*、*一致性*、*直写*。
- IEEE 标准 754** 电器和电子工程师协会制订的用于二进制浮点算法的标准，该标准于 1985 年发布。
- IPC** 请参见 *进程间通信*。
- 集关联缓存** 在集关联缓存中，可以将每个块放在固定数量的地址（至少有两个）中。块具有  $n$  个地址的集关联缓存称为  $n$  向集关联缓存。 $n$  向集关联缓存由多个集合组成，每个集合由  $n$  个块组成。可以将某个块放在该集合的任意位置（元素）中。增加关联性级别（集合中块的数量）就会增加缓存命中率。另请参见 *缓存*、*缓存局域性*、*伪共享*、*写无效*、*写更新*。
- 基数** 任何数系统的基数。例如，2 是二进制系统的基数，10 是十进制记数系统的基数。SPARC 工作站使用以 2 为基数的算法；IEEE 标准 754 是以 2 为基数的算法标准。
- 渐进下溢** 当浮点运算出现下溢时，返回一个次正规数而不是 0。这种下溢处理方法最大限度地降低了较小数浮点计算的不准确度。
- 进程** 此活动单元具有以下特征：单个顺序执行的线程、当前状态以及相关的系统资源集合。
- 进程间通信** 活动进程之间的消息传递。另请参见 *电路交换*、*分布式内存体系结构*、*MBus*、*消息传递*、*包交换*、*共享内存*、*XDBus*。
- 精度** 定量测量可表示数的密度。例如，在具有 53 个有效位精度的二进制浮点格式中，2 的任何两个相邻幂指数之间共有  $2^{53}$  个可表示数（在正规数的范围内）。不要将精度和准确度混淆起来，后者表示两个数的近似程度。
- 竞争性缓存** 竞争性缓存通过结合使用写无效和写更新来保持缓存一致性。竞争性缓存使用计数器来计量共享数据的时间。系统依据最近最少使用 (LRU) 的算法，将共享数据从缓存中清除。这可导致共享数据再次变成专用数据，从而消除了使用缓存一致性协议来访问内存（通过底板带宽）以保持多个副本同步的需要。另请参见 *缓存*、*缓存局域性*、*伪共享*、*写无效*、*写更新*。
- 控制流模型** 冯·诺伊曼计算机模型。此模型指定控制流；即指定在程序的每一步中执行哪条指令。所有 Sun 工作站都是冯·诺伊曼模型的示例。另请参见 *数据流模型*、*按需驱动的数据流*。
- LWP** 请参见 *轻量进程*。

- 链** 某些流水线结构的硬件特性，它允许在将某个运算的结果写入其目标寄存器的同时，直接将该结果用作第二个运算的操作数。两个链式运算的总的时间周期比各指令单独的时间周期之和短。例如，TI 8847 支持（相同精度的）连续 fadd、fsub 和 fmul 链。链式 fadd/fmul 需要 12 个周期，而连续非链式 fadd/fmul 需要 17 个周期。
- 流水线** 如果可以将应用于数据的汇总函数分为不同的处理阶段，则不同的数据部分可以从一个阶段流入另一个阶段；例如，包含以下阶段的编译器：词法分析、语法分析、类型检查、代码生成等。在第一个程序或模块通过词法分析后，就可以将它传递到语法分析，同时词法分析开始对第二个程序或模块进行分析。另请参见 *数组处理*、*矢量处理*。
- 流水线操作** 在这种硬件功能中，将操作分为多个阶段，每个阶段（通常）要花一个周期来完成。当每个周期发出新的操作时，就会填充流水线。如果流水线中的指令没有相关性，则每个周期可以提供新的结果。链式执行意味着对相关指令进行流水线处理。如果无法链式处理相关的指令（硬件不支持这些特定指令链），流水线操作就会停止。
- MBus** MBus 是一个用于处理器 / 内存 / IO 互连的总线规范。SPARC International 向几家生产互操作 CPU 模块、IO 接口和内存控制器的硅供应商颁发了该规范的使用许可证。MBus 是电路交换协议，它在单个总线上合并读取请求和响应。MBus 第 I 级定义了单处理器信号；MBus 第 II 级为写无效缓存一致性机制定义了多处理器扩展。
- MIMD** 请参见 *多指令多数据*、*共享内存*。
- NaN** 表示非数。以浮点格式编码的符号实体。
- 内存** 此介质可保存信息以便随后进行检索。此术语通常指的是机器指令可直接寻址的计算机内部存储。另请参见 *缓存*、*分布式内存*、*共享内存*。
- 内置模板** 在 Sun Studio 编译器的内联处理传递过程中，用于替换所定义的函数调用的汇编语言代码段。例如，内置模板文件 (libm.il) 中的数学库可使用它来访问 C 程序中的三角函数和其它初等函数的硬件实现。
- 商业上使用的并行处理器使用多级网络拓扑。多级网络拓扑的特征有：二维网格和布尔型 N 立方体。
- 偏置指数** 为使存储的指数范围非负数而选择的以 2 为基数的指数和常量（偏差）之和。例如， $2^{-100}$  的指数以 IEEE 单精度格式存储时为  $(-100) + (\text{单精度偏差 } 127) = 27$ 。
- 屏障** 用于协调任务的同步机制，即使不涉及数据访问。屏障与门类似。并行运行的处理程序或线程在不同的时间到达门，但直至所有处理程序都到达门时，才会让它们通过。例如，假定每天下班时，要求所有银行出纳员清点存入的金额和提取的金额。然后向银行副总裁汇报这些总额，副总裁核对这些总额之和与检查借款和贷款是否相等。出纳员各自的工作速度不同；即他们在不同的时间汇总完其交易情况。屏障机制禁止出纳员在核实完这些总额之和前下班回家。如果借款和贷款不相等，则所有出纳员都必须回到其办公桌前找出错误。在副总裁获得满意的总额之和后，就会去除屏障。



- 轻量进程** || 可使用内核的控制线程将 Solaris 线程实现为用户级库，这些控制线程称为轻量进程 (LWP)。在 Solaris 环境中，一个进程就是一个共享内存的 LWP 集合。每个 LWP 都具有 UNIX 进程的调度优先级，并共享该进程的资源。LWP 使用同步机制（如锁定）来协调其对共享内存的访问。可以将 LWP 视为一个执行代码或系统调用的虚拟 CPU。线程库调度进程中 LWP 池上的线程，其方式与内核调度处理器池上的 LWP 的方式基本相同。每个 LWP 是由内核单独调度的；执行单独的系统调用；导致单独的页面错误；以及在多处理器系统上并行运行。LWP 是由内核按照其调度级别和优先级在可用 CPU 资源上调度的。
- 缺省结果** 作为导致异常的浮点运算的结果而提供的值。
- SIMD** || 请参见 *单指令多数据*。
- SISD** || 请参见 *单指令单数据*。
- SPMD** || 请参见 *单程序多数据*。
- stderr** 标准误差是指向标准误差输出的 Unix 文件指针。在启动某个程序时，就会打开此文件。
- Store 0** 将算术运算的下溢结果刷新为零。
- 上下文切换** 在多任务处理操作系统（如 SunOS™ 操作系统）中，进程运行固定的一段时间。在这段时间结束时，CPU 会收到一个来自计时器的信号，中断当前运行的进程，并准备运行新的进程。CPU 为旧进程保存寄存器，然后为新进程加载寄存器。将旧进程的状态切换到新进程上的过程称为上下文切换。切换上下文所花的时间是系统开销；所需的时间取决于寄存器的数量以及是否需要使用特殊的指令来保存与进程关联的寄存器。
- 舍入** 必须向上或向下舍入不准确的结果以获取可表示的值。在向上舍入结果时，它就会增加到下一个可表示的值。在向下舍入时，它就会减少到上一个可表示的值。
- 舍入误差** 在将实数舍入到机器可表示的数时产生的误差。大多数浮点计算会产生舍入误差。对于任何一个浮点运算，IEEE 标准 754 规定结果不应产生多于一个舍入误差。
- 矢量处理** || 以一致的方式处理数据序列，在处理矩阵（其元素为矢量）或其它数据数组时通常使用这种处理方式。此顺序执行的数据处理可以使用流水线处理。另请参见 *数组处理*、*流水线*。
- 数据流模型** || 此计算机模型指定对数据所执行的运算并忽略指令顺序。即，根据数据值的可用性而不是根据指令的可用性向前计算。另请参见 *控制流模型*、*按需求驱动的数据流*。
- 数据争用** || 在多线程处理中，两个或多个线程同时访问共享资源的情形。结果是不确定的，具体情况取决于线程访问资源的顺序。当使用相同输入重复运行程序时，这种情况（称为“数据争用”）可产生不同的结果。另请参见 *互斥*、*互斥锁定*、*信号锁定*、*单锁定策略*、*转锁*。
- 数组处理** || 一组同时工作的处理器，每个处理器处理一个数组元素，因此单个运算可并行应用于所有的数组元素。

- 双精度** 使用两个字元来表示一个数以保持或提高精度。在 SPARC® 工作站上，双精度为 64 位 IEEE 双精度。
- 顺序执行的进程** 按以下方式执行的进程：在完成某个进程后，才能开始执行下一个进程。另请参见 *并发进程*、*进程*。
- 死锁** 当两个（或多个）单独的活动进程争用资源时出现的情况。假设进程 P 要求使用资源 X 和 Y，并且按该顺序请求使用这些资源，同时进程 Q 要求使用资源 Y 和 X，并且按该顺序请求这些资源。如果进程 P 获取了资源 X，并且进程 Q 同时获取了资源 Y，则两个进程均无法继续执行 - 每个进程都要求使用已分配给另一个进程的资源。
- 锁定** 此机制用于实施对共享数据进行串行访问的策略。线程或进程使用特定的锁以获取对该锁定保护的共享内存的访问权限。只有程序员知道必须锁定哪些内容，从这个意义上讲，数据锁定和解除锁定是自发的。另请参见 *数据争用*、*互斥*、*互斥锁定*、*信号锁定*、*单锁定策略*、*转锁*。
- 条件变量** 对于 Solaris 线程而言，条件变量允许一次性地阻塞线程，直至满足某种条件时为止。条件是在互斥锁定的保护下进行测试的。当条件为假时，线程因某个条件变量而阻塞，并一次性地释放互斥，等待条件的改变。当另一个线程改变条件时，它可以向关联的条件变量发信号以唤醒一个或多个等待的线程，重新获取互斥并重新评估条件。如果已为变量分配了可写入的内存，且该变量在具有协作关系的进程间共享，并已被初始化，则可以使用条件变量同步此进程和其它进程中的线程。
- 拓扑** 请参见 *互连网络拓扑*。
- ulp** 代表最后一位中的单位。在二进制格式中，尾数的最低有效位（第 0 位）是最后一位中的单位。
- ulp(x)** 表示以工作格式截断的 x 的 ulp。
- 完全关联的缓存** 具有  $m$  个条目的完全关联缓存是  $m$  向集关联缓存。即，它包含具有  $m$  个块的单个集合。缓存条目可以位于该集合内  $m$  块中的任何一个。另请参见 *缓存*、*缓存局部性*、*直接映射的缓存*、*伪共享*、*集关联缓存*、*写无效*、*写更新*。
- 伪共享** 由两个线程独立访问的两个无关的数据存在于同一块中时，缓存中发生的情况。此块可能会在没有什么正常原因的情况下最终在缓存之间被弹来弹去。如果注意到此类情况并重新安排数据结构以避免出现伪共享，则可大大提高缓存性能。另请参见 *缓存*、*缓存局部性*。
- 无噪声 NaN** 通过几乎每个算术运算进行传递而不会引发新异常的 NaN（非数）。

- XDBus** || XDBus 规范使用低阻抗的 GTL（射电收发器逻辑）收发器信号，以更高的时钟速率来驱动更长的底板。XDBus 通过以下方法来支持大量 CPU：使用多个交叉存储的存储库来提高吞吐量。XDBus 使用包交换协议来处理分开的请求和响应，以便更有效地利用总线。XDBus 还定义了交叉存储的方案，以便将一个、两个或四个单独的总线数据路径用作单个底板以提高吞吐量。XDBus 支持写无效、写更新以及竞争性缓存一致性方案，并且包含几个拥塞控制机制。另请参见 *缓存、一致性、竞争性缓存、写无效、写更新*。
- 下溢** || 如果浮点算术运算的结果很小，仅通过正常舍入无法以目标浮点格式将其表示为正规数，就会发生这种情况。
- 线程** || 单个 UNIX 进程地址空间中的控制流。Solaris 线程提供轻量形式的并发任务，允许在公共用户地址空间中使用多个控制线程，并产生最小限度的调度和通信开销。线程共享相同的地址空间、文件描述符（当一个线程打开文件时，其它线程可以读取该文件）、数据结构和操作系统状态。线程使用程序计数器和堆栈来跟踪局部变量和返回地址。线程之间通过使用共享数据和线程同步操作来进行交互。另请参见 *绑定线程、轻量进程、多线程处理、非绑定线程*。
- 消息传递** || 在分布式内存体系结构中，进程彼此之间进行通信使用的机制。没有进程用于存放消息的共享数据结构。消息传递允许进程将数据发送到另一个进程，并允许指定的接收者同步数据的到达时间。
- 写更新** || 写更新（也称为写 - 广播）通过立即更新所有缓存中共享变量的所有副本来保持缓存一致性。这是一种直写形式，因为所有写入都通过总线来更新共享数据的副本。写更新的优点是新的值立即出现在缓存中，这可减少延迟。另请参见 *缓存、缓存局部性、一致性、伪共享、写无效*。
- 写无效** || 通过从本地缓存中读取数据来保持缓存一致性，直到有写入发生。要更改变量的值，写入处理器先使其它缓存中的所有副本无效。然后，写入处理器就可以任意更新其本地副本，直至另一处理器请求该变量时为止。写入处理器通过总线发出一个无效信号，所有缓存检查它们是否有副本；如果有的话，则它们必须使包含该字元的块无效。此方案允许多个读取处理器，但只允许一个写入处理器。写无效仅在第一次写入时使用总线使其它副本无效；随后的本地写入不会产生总线通信，因此减少了对总线带宽的需求。另请参见 *缓存、缓存局部性、一致性、伪共享、写更新*。
- 信号** || E. W. Dijkstra 提出的一种专门用途的数据类型，可用于协调对特定资源或一组共享资源的访问。信号具有整数值（不能为负数），并且允许对它进行两种运算。信号（v 或 up）运算将该值增加 1，通常表示资源已变为可用资源。等待（P 或 down）运算将该值减少 1（条件是没有将该值变为负数），通常表示将要开始使用某个可用资源。另请参见 *信号锁定*。
- 信号 NaN** || 当作为操作数出现时引发无效运算异常的 NaN（非数）。
- 信号锁定** || 此同步机制通过协同异步线程来控制对关键资源的访问。另请参见 *信号*。

- 嗅探** Ⅱ 用于保持缓存一致性的最常见协议称为“嗅探”。缓存控制器监视或嗅探总线以确定缓存是否包含共享块的副本。
- 对于读取而言，多个副本可以位于不同处理器的缓存中，但因为处理器需要最新的副本，所以所有处理器在写入后必须获取新的值。另请参见 *缓存、竞争性缓存、伪共享、写无效、写更新*。
- 对于写入而言，处理器必须拥有独占访问权限，才能写入到缓存中。写入到非共享块不会导致总线通信。写入到共享数据的结果是，要么使所有其它副本无效，要么使用写入的值更新共享副本。另请参见 *缓存、竞争性缓存、伪共享、写无效、写更新*。
- 异步控制** Ⅱ 在收到发生了特定事件的指示（信号）时开始执行特定操作的计算机控制行为。异步控制依靠称为“锁定”的同步机制来协调处理器。另请参见 *互斥、互斥锁定、信号锁定、单锁定策略、转锁*。
- 异常** 当执行的原子算法运算没有可普遍接受的结果时，就会发生算法异常。原子和可接受的含义随时间和地点而发生变化。
- 一致性** Ⅱ 在具有多个缓存的系统中，此机制可确保所有处理程序在任何时候看到的内存映像都相同。
- 隐藏位** 硬件为确保正确舍入而使用的额外位，软件无法访问这些位。例如，IEEE 双精度运算使用三个隐藏位来计算 56 位结果，然后舍入为 53 位。
- 有效数字** 浮点数的一部分，可用基数的有符号幂乘以它以确定该数的值。在正规化数中，有效数字由小数点左侧的单个非零数字以及右侧的小数组成。
- 正规数** 在 IEEE 算法中，偏置指数不是零或最大值（全为 1）的数字，它表示正常范围内实数的一个子集，并具有很小的限定相对误差。
- 直接映射的缓存** Ⅱ 直接映射的缓存是单向集关联缓存。即，每个缓存条目保存一个块，并形成包含一个元素的单个集合。另请参见 *缓存、缓存局域性、伪共享、完全关联的缓存、集关联缓存、写无效、写更新*。
- 指数** 浮点数的一部分，在确定可表示数的值时，它表示基数的整数幂指数。
- 直写** Ⅱ 用于在缓存和主内存之间保持一致性的写入策略。直写式（也称为直接存储）写入到主内存和本地缓存中的块。直写的优点是主内存具有最新的数据副本。另请参见 *缓存、一致性、回写*。
- 转锁** Ⅱ 线程使用转锁来反复测试锁定变量，直至其它某个任务释放该锁定时为止。即，等待线程旋转锁定，直至清除锁定时为止。然后，当等待线程在关键区域内部时设置锁定。当关键区域中的工作完成后，线程清除转锁，以使其它线程可以进入关键区域。转锁和互斥的差别之处在于，在试图获取其它线程保持的互斥时将阻塞和释放 LWP；转锁不会释放 LWP。另请参见 *互斥锁定*。
- 准确度** 衡量两个数的近似程度。例如，计算结果的准确度通常反映了它与精确运算结果相差的计算误差大小。可以用有效数位来表示准确度（例如，“结果精确到 6 位数字”）；更普遍的作法是使用保留相关数学特性来表示准确度（例如，“结果具有正确的代数符号”）。

**字元** 作为给定计算机内的单个实体存储、寻址、传输和操作的有序字符集合。在 SPARC 工作站环境中，一个字元为 32 位。

**阻塞状态** 线程正在等待资源或数据（例如从某个等待处理的磁盘读取操作中返回的数据），或者等待另一个线程解锁资源。



# 索引

---

## A

adb, 76  
addrans  
    随机数公用程序, 66

## B

贝塞尔函数, 243  
编译器, 访问, 16  
表示单精度值  
    C 示例, 112  
表示双精度值  
    C 示例, 112  
    FORTRAN 示例, 113  
捕获  
    出现异常时终止, 140  
捕获浮点异常  
    C 示例, 131  
捕获异常  
    C 示例, 131, 134

## C

C 驱动程序  
    示例, 从 C 中调用 FORTRAN 子例程, 152  
convert\_external  
    二进制浮点, 66  
    数据转换, 66  
参数缩小

三角函数, 65  
操作系统数学库  
    libm.a, 45  
    libm.so, 45  
出现异常时终止  
    C 示例, 140  
次正规数, 42  
    浮点计算, 39

## D

dbx, 76  
单精度表示  
    C 示例, 111  
单精度格式, 25  
底转换  
    带格式 I/O, 38  
    以 10 为底到以 2 为底, 38  
    以 2 为底到以 10 为底, 38

## E

errno.h  
    为 errno 定义值, 242

## F

-fast, 169  
floatingpoint.h

定义处理程序类型

C 和 C++, 85

fmod, 243

-fnonstd, 169

浮点

教程, 173

舍入方向, 24

舍入精度, 24

异常列表, 24

浮点队列 (FQ), 164

浮点选项, 162

浮点异常, 21

标志, 72

当前, 72

应计, 72

捕获优先级, 72

产生的异常位, 128

常见异常, 70

出现异常时终止, 140

定义, 69

ieee\_functions, 54

ieee\_retrospective, 59

缺省结果, 70

异常列表, 70

优先级, 72

浮点状态寄存器 (FSR), 158, 164

浮点准确度

十进制数字串和二进制浮点数, 23

## G

Goldberg 论文, 173

参考书目, 220

IEEE 标准, 182, 186

简介, 174

舍入误差, 174

系统方面, 199

详细资料, 209

小结, 220

摘要, 173

致谢, 220

## H

HUGE

与 IEEE 标准的兼容性, 240

HUGE\_VAL

与 IEEE 标准的兼容性, 240

## I

IEEE 754 标准

单精度格式, 23

双精度格式, 23

双精度扩展格式, 23

IEEE 单精度格式

次正规数位模式, 26

带分数, 有效数字, 26

非规格化数, 26

符号位, 25

Inf, 正无穷大, 26

精度, 正规数, 26

NaN, 非数, 27

偏置指数, 25

偏置指数, 隐含位, 26

位赋值, 25

位模式和等价值, 26

尾数, 25

位字段赋值, 25

正规数

最大正, 26

正规数位模式, 26

IEEE 格式

与语言数据类型的关系, 25

IEEE 双精度格式

次正规数, 28

非规格化数, 28

符号位, 27

Inf, 无穷大, 28

精度, 28

NaN, 非数, 29

偏置指数, 27

位模式和等价值, 29

尾数, 27

SPARC 上的存储, 27

x86 上的存储, 27



- 位字段赋值, 27
- 隐含位, 28
- 有效数字, 28
- 正规数, 28
- IEEE 双精度扩展格式
  - 次正规数
    - SPARC 体系结构, 30
    - x86 体系结构, 32
  - 符号位
    - x86 体系结构, 32
  - Inf
    - SPARC 体系结构, 30
    - x86 体系结构, 33
  - NaN
    - x86 体系结构, 34
  - 偏置指数
    - x86 体系结构, 31
  - 四倍精度
    - SPARC 体系结构, 29
  - 尾数
    - x86 体系结构, 31
  - 位字段赋值
    - x86 体系结构, 31
  - 有效数字
    - 显式前导位 (x86 体系结构), 31
  - 正规数
    - SPARC 体系结构, 30
    - x86 体系结构, 32
- ieee\_flags
  - 产生的异常标记, 57
  - 检查产生的异常位 - C 示例, 128
  - 截断舍入, 58
  - 舍入方向, 57
  - 舍入精度, 57, 58
  - 设置异常标记 - C 示例, 131
- ieee\_functions
  - 浮点异常, 54
  - 位掩码运算, 53
- ieee\_handler, 85
  - 捕获常见异常, 70
  - 捕获异常
    - C 示例, 131
  - 出现异常时终止
    - FORTTRAN 示例, 140
  - 示例, 调用顺序, 77

- ieee\_retrospective
  - 浮点异常, 59
  - 浮点状态寄存器 (FSR), 59
  - 获得有关非标准 IEEE 模式的信息, 59
  - 获得有关未处理异常的信息, 59
  - 检查下溢异常标记, 168
  - 禁止异常消息, 60
  - 精度, 59
  - nonstandard\_arithmetic 生效, 59
  - 舍入, 59
- ieee\_sun
  - IEEE 分类函数, 53
- ieee\_values
  - 表示 NaN, 55
  - 表示浮点值, 55
  - 表示无穷大, 55
  - 表示正规数, 55
  - 单精度值, 55
  - 四倍精度值, 55
- ieee\_values 函数
  - C 示例, 120
- Inf, 21, 241

## J

- 检查产生的异常标记
  - C 示例, 130
- 检查产生的异常位
  - C 示例, 128
- 渐进下溢
  - 误差特性, 41
- 教程, 浮点, 173
- 具有易读格式的文档, 19

## L

- lcrans
  - 随机数公用程序, 66
- libm
  - 标准安装, 45
  - 函数列表, 46
  - 缺省目录 - 可执行文件, 45
  - 缺省目录 - 头文件, 45

- SVID 遵循性, 240
- libm 的 SVID 行为
  - xt 编译器选项, 242
- libm 的 X/Open 行为
  - xa 编译器选项, 242
- libm 函数
  - 单精度, 52
  - 双精度, 52
  - 四倍精度, 52
- libmil (另请参见内置模板), 240
- libsunmath
  - 标准安装, 47
  - 函数列表, 47
  - 缺省目录 - 可执行文件, 47
  - 缺省目录 - 头文件, 47

## M

- MANPATH 环境变量, 设置, 17
- MAXFLOAT, 242

## N

- NaN, 21, 31, 240, 243
- nonstandard\_arithmetic
  - 关闭 IEEE 渐进下溢, 169
  - 渐进下溢, 60
  - 下溢, 60

## P

- PATH 环境变量, 设置, 16
- pi
  - 无限精确的值, 66
- 平方根指令, 166, 240

## S

- shell 提示符, 15
- shufrans
  - 混洗伪随机数, 66

- standard\_arithmetic
  - 打开 IEEE 行为, 169
- Store 0, 39
  - 刷新下溢结果, 43
- SVID 异常
  - 将 errno 设置为 EDOM
    - 不正确的操作数, 239
  - 将 errno 设置为 ERANGE
    - 上溢或下溢, 239
  - matherr, 239
  - PLOSS, 243
  - TLOSS, 243
- 三角函数
  - 参数缩小, 65
- 舍入方向, 24
  - C 示例, 122
- 舍入精度, 24
- 舍入误差
  - 准确度
    - 降低, 40
- 设置异常标记
  - C 示例, 131
- 生成一个数字数组
  - FORTRAN 示例, 113
- 十进制表示
  - 范围, 35
  - 精度, 35
  - 最大正正规数, 35
  - 最小正正规数, 35
- 实数值线
  - 2 的幂指数, 42
  - 二进制表示, 35
  - 十进制表示, 35
- 时钟速度, 170
- 手册页, 访问, 16
- 数据类型
  - 与 IEEE 格式的关系, 25
- 刷新为零 (请参见 Store 0), 39
- 双精度表示
  - C 示例, 111
  - FORTRAN 示例, 112
- 随机数公用程序
  - shufrans, 66

随机数生成器, 113

## W

values.h

定义错误消息, 242

文档, 访问, 18

文档索引, 18

无穷大

被零除时得到的缺省结果, 70

无序比较

浮点值, 71

NaN, 71

无噪声 NaN

无效运算的缺省结果, 70

## X

X\_TLOSS, 242

-Xa, 242

-Xc, 242

-Xs, 242

-Xt, 242

系统 V 接口定义 (SVID), 239

下溢

浮点计算, 39

渐进, 39

nonstandard\_arithmetic, 60

阈值, 43

下溢阈值

单精度, 39

双精度精度, 39

双精度扩展精度, 39

陷阱

ieee\_retrospective, 59

## Y

印刷惯例, 14

## Z

在十进制数字串和二进制浮点数之间进行转换, 24

在数字组之间转换, 36

正规数

最大正, 26

最小正, 39, 42

准确度, 243

浮点运算, 23

有效数字 (个数), 35

阈值, 44

最后一位中的单位 (ulp), 65

