# Oracle® Solaris Studio 12.3：C++ 用户指南

文件号码 E26439
2011 年 12 月，E26439-01

ORACLE

版权所有 © 1991, 2011, Oracle 和/或其附属公司。 保留所有权利。

本软件和相关文档是根据许可证协议提供的，该许可证协议中规定了关于使用和公开本软件和相关文档的各种限制，并受知识产权法的保护。除非在许可证协议中明确许可或适用法律明确授权，否则不得以任何形式、任何方式使用、拷贝、复制、翻译、广播、修改、授权、传播、分发、展示、执行、发布或显示本软件和相关文档的任何部分。除非法律要求实现互操作，否则严禁对本软件进行逆向工程设计、反汇编或反编译。

此文档所含信息可能随时被修改，恕不另行通知，我们不保证该信息没有错误。如果贵方发现任何问题，请书面通知我们。

如果将本软件或相关文档交付给美国政府，或者交付给以美国政府名义获得许可证的任何机构，必须符合以下规定：

U.S. GOVERNMENT END USERS:

Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

本软件或硬件是为了在各种信息管理应用领域内的一般使用而开发的。它不应被应用于任何存在危险或潜在危险的应用领域，也不是为此而开发的，其中包括可能会产生人身伤害的应用领域。如果在危险应用领域内使用本软件或硬件，贵方应负责采取所有适当的防范措施，包括备份、冗余和其它确保安全使用本软件或硬件的措施。对于因在危险应用领域内使用本软件或硬件所造成的一切损失或损害，Oracle Corporation 及其附属公司概不负责。

Oracle 和 Java 是 Oracle 和/或其附属公司的注册商标。其他名称可能是各自所有者的商标。

Intel 和 Intel Xeon 是 Intel Corporation 的商标或注册商标。所有 SPARC 商标均是 SPARC International, Inc 的商标或注册商标，并应按照许可证的规定使用。AMD、Opteron、AMD 徽标以及 AMD Opteron 徽标是 Advanced Micro Devices 的商标或注册商标。UNIX 是 The Open Group 的注册商标。

本软件或硬件以及文档可能提供了访问第三方内容、产品和服务的方式或有关这些内容、产品和服务的信息。对于第三方内容、产品和服务，Oracle Corporation 及其附属公司明确表示不承担任何种类的担保，亦不对其承担任何责任。对于因访问或使用第三方内容、产品或服务所造成的任何损失、成本或损害，Oracle Corporation 及其附属公司概不负责。

120220@25097

# 目录

**前言** ..... 19

**第 1 部分 C++ 编译器** ..... 23

**1 C++ 编译器** ..... 25
1.1 Oracle Solaris Studio 12.3 C++ 5.12 编译器的新特性和新功能 ..... 25
1.2 x86 特殊注意事项 ..... 26
1.3 针对 64 位平台进行编译 ..... 26
1.4 二进制兼容验证 ..... 27
1.5 标准符合性 ..... 27
1.6 发行版信息 ..... 27
1.7 手册页 ..... 28
1.8 本地语言支持 ..... 28

**2 使用 C++ 编译器** ..... 29
2.1 入门 ..... 29
2.2 调用编译器 ..... 30
2.2.1 命令语法 ..... 30
2.2.2 文件名称约定 ..... 31
2.2.3 使用多个源文件 ..... 32
2.3 使用不同编译器版本进行编译 ..... 32
2.4 编译和链接 ..... 32
2.4.1 编译和链接序列 ..... 32
2.4.2 分别编译和链接 ..... 33
2.4.3 一致编译和链接 ..... 33
2.4.4 针对 64 位内存模型进行编译 ..... 33
2.4.5 编译器命令行诊断 ..... 34

2.4.6 了解编译器的组织 ..... 34
2.5 预处理指令和名称 ..... 35
2.5.1 Pragma ..... 35
2.5.2 具有可变数目的参数的宏 ..... 35
2.5.3 预定义的名称 ..... 36
2.5.4 警告和错误 ..... 36
2.6 内存要求 ..... 36
2.6.1 交换空间大小 ..... 37
2.6.2 增加交换空间 ..... 37
2.6.3 虚拟内存的控制 ..... 37
2.6.4 内存要求 ..... 38
2.7 将 strip 命令用于 C++ 目标 ..... 38
2.8 简化命令 ..... 38
2.8.1 在 C Shell 中使用别名 ..... 38
2.8.2 使用 CCFLAGS 指定编译选项 ..... 39
2.8.3 使用 make ..... 39

**3 使用 C++ 编译器选项 ..... 41**
3.1 语法概述 ..... 41
3.2 通用指南 ..... 41
3.3 按功能汇总的选项 ..... 42
3.3.1 代码生成选项 ..... 42
3.3.2 编译时性能选项 ..... 43
3.3.3 编译时选项和链接时选项 ..... 43
3.3.4 调试选项 ..... 44
3.3.5 浮点选项 ..... 45
3.3.6 语言选项 ..... 46
3.3.7 库选项 ..... 46
3.3.8 已过时选项 ..... 47
3.3.9 输出选项 ..... 48
3.3.10 运行时性能选项 ..... 49
3.3.11 预处理程序选项 ..... 50
3.3.12 分析选项 ..... 51
3.3.13 参考选项 ..... 51
3.3.14 源文件选项 ..... 51

3.3.15 模板选项 ..... 51
3.3.16 线程选项 ..... 52
3.4 用户提供的缺省选项文件 ..... 52

**第 2 部分 编写 C++ 程序** ..... 55

**4 语言扩展** ..... 57
4.1 链接程序作用域 ..... 57
4.1.1 与 Microsoft Windows 兼容 ..... 58
4.2 线程局部存储 ..... 59
4.3 用限制较少的虚函数覆盖 ..... 59
4.4 对 enum 类型和变量进行前向声明 ..... 60
4.5 使用不完整 enum 类型 ..... 60
4.6 将 enum 名称作为作用域限定符 ..... 60
4.7 使用匿名 struct 声明 ..... 61
4.8 传递匿名类实例的地址 ..... 62
4.9 将静态名称空间作用域函数声明为类友元 ..... 62
4.10 将预定义 __func__ 符号用于函数名 ..... 63
4.11 支持的属性 ..... 63
4.11.1 __packed__ 属性详细信息 ..... 64
4.12 对 Intel MMX 和扩展的 x86 平台内部函数的编译器支持 ..... 65

**5 程序组织** ..... 67
5.1 头文件 ..... 67
5.1.1 可适应语言的头文件 ..... 67
5.1.2 幂等头文件 ..... 68
5.2 模板定义 ..... 69
5.2.1 包括的模板定义 ..... 69
5.2.2 独立的模板定义 ..... 69

**6 创建和使用模板** ..... 71
6.1 函数模板 ..... 71
6.1.1 函数模板声明 ..... 71
6.1.2 函数模板定义 ..... 71

6.1.3 函数模板用法 ..... 72
6.2 类模板 ..... 72
6.2.1 类模板声明 ..... 72
6.2.2 类模板定义 ..... 72
6.2.3 类模板成员定义 ..... 73
6.2.4 类模板的用法 ..... 74
6.3 模板实例化 ..... 74
6.3.1 隐式模板实例化 ..... 74
6.3.2 显式模板实例化 ..... 74
6.4 模板组合 ..... 75
6.5 缺省模板参数 ..... 76
6.6 模板专门化 ..... 76
6.6.1 模板专门化声明 ..... 76
6.6.2 模板专门化定义 ..... 76
6.6.3 模板专门化使用和实例化 ..... 77
6.6.4 部分专门化 ..... 77
6.7 模板问题部分 ..... 78
6.7.1 非本地名称解析和实例化 ..... 78
6.7.2 作为模板参数的本地类型 ..... 78
6.7.3 模板函数的友元声明 ..... 79
6.7.4 在模板定义内使用限定名称 ..... 81
6.7.5 嵌套模板名称 ..... 81
6.7.6 引用静态变量和静态函数 ..... 81
6.7.7 在同一目录中使用模板生成多个程序 ..... 82

**7 编译模板** ..... 85
7.1 详细编译 ..... 85
7.2 系统信息库管理 ..... 85
7.2.1 生成的实例 ..... 85
7.2.2 整个类实例化 ..... 85
7.2.3 编译时实例化 ..... 86
7.2.4 模板实例的放置和链接 ..... 86
7.3 外部实例 ..... 87
7.3.1 可能的高速缓存冲突 ..... 87
7.3.2 静态实例 ..... 88

7.3.3 全局实例 ..... 88
7.3.4 显式实例 ..... 88
7.3.5 半显式实例 ..... 89
7.4 模板系统信息库 ..... 89
7.4.1 系统信息库结构 ..... 89
7.4.2 写入模板系统信息库 ..... 89
7.4.3 从多模板系统信息库读取 ..... 90
7.4.4 共享模板系统信息库 ..... 90
7.4.5 通过 -instances=extern 实现模板实例自动一致 ..... 90
7.5 模板定义搜索 ..... 90
7.5.1 源文件位置约定 ..... 91
7.5.2 定义搜索路径 ..... 91
7.5.3 诊断有问题的搜索 ..... 91

**8 异常处理** ..... 93
8.1 同步和异步异常 ..... 93
8.2 指定运行时错误 ..... 93
8.3 禁用异常 ..... 94
8.4 使用运行时函数和预定义的异常 ..... 94
8.5 将异常与信号和 Setjmp/Longjmp 混合使用 ..... 95
8.6 生成具有异常的共享库 ..... 95

**9 改善程序性能** ..... 97
9.1 避免临时对象 ..... 97
9.2 使用内联函数 ..... 97
9.3 使用缺省运算符 ..... 98
9.4 使用值类 ..... 99
9.4.1 选择直接传递类 ..... 99
9.4.2 在不同的处理器上直接传递类 ..... 99
9.5 缓存成员变量 ..... 100

**10 生成多线程程序** ..... 101
10.1 生成多线程程序 ..... 101
10.1.1 表明多线程编译 ..... 101

10.1.2 与线程和信号一起使用 C++ 支持库 ..... 102
10.2 在多线程程序中使用异常 ..... 102
10.2.1 线程取消 ..... 102
10.3 在线程之间共享 C++ 标准库对象 ..... 102
10.4 内存边界内部函数 ..... 104

**第 3 部分 库** ..... 107

**11 使用库** ..... 109
11.1 C 库 ..... 109
11.2 随 C++ 编译器提供的库 ..... 109
11.2.1 C++ 库描述 ..... 110
11.2.2 访问 C++ 库的手册页 ..... 111
11.2.3 缺省 C++ 库 ..... 111
11.3 相关的库选项 ..... 111
11.4 使用类库 ..... 113
11.4.1 `iostream` 库 ..... 113
11.4.2 链接 C++ 库 ..... 114
11.5 静态链接标准库 ..... 114
11.6 使用共享库 ..... 115
11.7 替换 C++ 标准库 ..... 116
11.7.1 可以替换的内容 ..... 116
11.7.2 不可替换的内容 ..... 116
11.7.3 安装替换库 ..... 117
11.7.4 使用替换库 ..... 117
11.7.5 标准头文件实现 ..... 117

**12 使用 C++ 标准库** ..... 121
12.1 C++ 标准库头文件 ..... 122
12.2 STLport ..... 123
12.2.1 重新分发和支持的 STLport 库 ..... 124
12.3 Apache stdcxx 标准库 ..... 124

**13 使用传统 iostream 库** ..... 127
13.1 预定义的 iostream ..... 127
13.2 iostream 交互的基本结构 ..... 128
13.3 使用传统 iostream 库 ..... 128
13.3.1 使用 iostream 进行输出 ..... 129
13.3.2 使用 iostream 进行输入 ..... 131
13.3.3 定义自己的提取运算符 ..... 132
13.3.4 使用 char* 提取器 ..... 132
13.3.5 读取任何单一字符 ..... 133
13.3.6 二进制输入 ..... 133
13.3.7 查看输入 ..... 133
13.3.8 提取空白 ..... 133
13.3.9 处理输入错误 ..... 134
13.3.10 结合使用 iostream 与 stdio ..... 134
13.4 创建 iostream ..... 135
13.4.1 使用类 fstream 处理文件 ..... 135
13.5 iostream 赋值 ..... 137
13.6 格式控制 ..... 138
13.7 操纵符 ..... 138
13.7.1 使用无格式操纵符 ..... 139
13.7.2 参数化操纵符 ..... 140
13.8 strstream：用于数组的 iostream ..... 141
13.9 stdiobuf：用于 stdio 文件的 iostream ..... 141
13.10 处理 streambuf 流 ..... 141
13.10.1 streambuf 指针类型 ..... 142
13.10.2 使用 streambuf 对象 ..... 142
13.11 iostream 手册页 ..... 143
13.12 iostream 术语 ..... 144

**14 生成库** ..... 147
14.1 了解库 ..... 147
14.2 生成静态（归档）库 ..... 148
14.3 生成动态（共享）库 ..... 148
14.4 生成包含异常的共享库 ..... 149
14.5 生成专用的库 ..... 149

14.6 生成公用的库 ..... 150
14.7 生成具有 C API 的库 ..... 150
14.8 使用 dlopen 从 C 程序访问 C++ 库 ..... 151

**第 4 部分　附录** ..... 153

**A　C++ 编译器选项** ..... 155
A.1 选项信息的结构 ..... 155
A.2 选项参考 ..... 156
A.2.1 -# ..... 156
A.2.2 -### ..... 156
A.2.3 –B*binding* ..... 156
A.2.4 –c ..... 158
A.2.5 –cg{89|92} ..... 158
A.2.6 –compat=[ 5|g] ..... 158
A.2.7 +d ..... 159
A.2.8 -D*name*[ =*def*] ..... 160
A.2.9 –d{y|n} ..... 161
A.2.10 –dalign ..... 161
A.2.11 –dryrun ..... 162
A.2.12 –E ..... 162
A.2.13 -erroff[=*t*] ..... 163
A.2.14 -errtags[=*a*] ..... 164
A.2.15 -errwarn[=*t*] ..... 164
A.2.16 –fast ..... 165
A.2.17 –features=*a*[ ,*a*...] ..... 167
A.2.18 -filt[=*filter*[,*filter*...]] ..... 170
A.2.19 –flags ..... 172
A.2.20 -fma[={ none|fused}] ..... 172
A.2.21 –fnonstd ..... 172
A.2.22 –fns[={yes| no}] ..... 172
A.2.23 –fprecision=*p* ..... 174
A.2.24 –fround=*r* ..... 174
A.2.25 –fsimple[=*n*] ..... 175
A.2.26 –fstore ..... 176

A.2.27 -ftrap=*t*[ ,*t*...] ..... 177
A.2.28 –G ..... 178
A.2.29 –g ..... 179
A.2.30 –g0 ..... 180
A.2.31 -g3 ..... 180
A.2.32 –H ..... 180
A.2.33 –h[ ]*name* ..... 180
A.2.34 –help ..... 181
A.2.35 -I*pathname* ..... 181
A.2.36 -I- ..... 182
A.2.37 –i ..... 184
A.2.38 -include *filename* ..... 184
A.2.39 -inline ..... 185
A.2.40 –instances=*a* ..... 185
A.2.41 –instlib= *filename* ..... 186
A.2.42 –KPIC ..... 187
A.2.43 –Kpic ..... 187
A.2.44 –keeptmp ..... 187
A.2.45 –L路径 ..... 187
A.2.46 –l*lib* ..... 188
A.2.47 –libmieee ..... 188
A.2.48 –libmil ..... 188
A.2.49 -library=*l*[ ,*l*...] ..... 188
A.2.50 -m32|-m64 ..... 191
A.2.51 -mc ..... 192
A.2.52 –misalign ..... 192
A.2.53 -mr[, *string*] ..... 192
A.2.54 -mt[={yes |no}] ..... 192
A.2.55 –native ..... 193
A.2.56 –noex ..... 193
A.2.57 –nofstore ..... 193
A.2.58 –nolib ..... 193
A.2.59 –nolibmil ..... 194
A.2.60 –norunpath ..... 194
A.2.61 –O ..... 194
A.2.62 –O*level* ..... 194

A.2.63 –o *filename* ..... 194
A.2.64 +p ..... 195
A.2.65 –P ..... 195
A.2.66 –p ..... 196
A.2.67 –pentium ..... 196
A.2.68 –pg ..... 196
A.2.69 -PIC ..... 196
A.2.70 –pic ..... 196
A.2.71 –pta ..... 196
A.2.72 –pti*path* ..... 196
A.2.73 –pto ..... 197
A.2.74 –ptv ..... 197
A.2.75 –Qoption *phase option*[,*option*…] ..... 197
A.2.76 –qoption *phase option* ..... 198
A.2.77 –qp ..... 198
A.2.78 –Qproduce *sourcetype* ..... 198
A.2.79 –qproduce *sourcetype* ..... 199
A.2.80 –R*pathname*[ :*pathname*…] ..... 199
A.2.81 –S ..... 199
A.2.82 –s ..... 199
A.2.83 -staticlib=*l*[ ,*l*…] ..... 199
A.2.84 -sync_stdio=[yes| no] ..... 201
A.2.85 –temp=*path* ..... 202
A.2.86 –template=*opt*[,*opt*…] ..... 202
A.2.87 –time ..... 203
A.2.88 -traceback[={ %none|common|*signals_list*}] ..... 203
A.2.89 –U*name* ..... 204
A.2.90 –unroll=*n* ..... 205
A.2.91 –V ..... 205
A.2.92 –v ..... 205
A.2.93 –verbose=*v*[,*v*…] ..... 205
A.2.94 -W*c* ,*arg* ..... 206
A.2.95 +w ..... 207
A.2.96 +w2 ..... 207
A.2.97 –w ..... 207
A.2.98 -Xlinker *arg* ..... 208

A.2.99 –Xm ..... 208
A.2.100 -xaddr32 ..... 208
A.2.101 -xalias_level[=*n*] ..... 208
A.2.102 -xanalyze={code| no} ..... 210
A.2.103 -xannotate[=yes| no] ..... 211
A.2.104 –xar ..... 211
A.2.105 –xarch=*isa* ..... 212
A.2.106 -xautopar ..... 215
A.2.107 -xbinopt={prepare| off} ..... 215
A.2.108 -xbuiltin[={ %all|%default|%none}] ..... 216
A.2.109 –xcache=*c* ..... 217
A.2.110 -xchar[=*o*] ..... 218
A.2.111 -xcheck[=*i*] ..... 220
A.2.112 -xchip=*c* ..... 220
A.2.113 –xcode=*a* ..... 222
A.2.114 -xdebugformat=[stabs|dwarf] ..... 224
A.2.115 -xdepend=[yes| no] ..... 225
A.2.116 -xdumpmacros[=*value*[,*value*...]] ..... 225
A.2.117 -xe ..... 228
A.2.118 -xF[=*v*[,*v*...]] ..... 228
A.2.119 -xhelp=flags ..... 229
A.2.120 -xhwcprof ..... 230
A.2.121 -xia ..... 230
A.2.122 -xinline[=*func-spec*[,*func-spec*...]] ..... 231
A.2.123 -xinstrument=[ no%]datarace ..... 232
A.2.124 -xipo[={0|1|2}] ..... 233
A.2.125 -xipo_archive=[*a*] ..... 235
A.2.126 -xivdep[=*p*] ..... 236
A.2.127 -xjobs=*n* ..... 237
A.2.128 -xkeepframe[=[ %all,%none,*name*,no% *name*]] ..... 237
A.2.129 -xlang=*language* [,*language*] ..... 238
A.2.130 -xldscope={*v*} ..... 239
A.2.131 -xlibmieee ..... 240
A.2.132 -xlibmil ..... 240
A.2.133 –xlibmopt ..... 241
A.2.134 –xlic_lib=sunperf ..... 241

A.2.135 –xlicinfo ..... 241
A.2.136 -xlinkopt[= *level*] ..... 241
A.2.137 -xloopinfo ..... 242
A.2.138 –xM ..... 243
A.2.139 -xM1 ..... 243
A.2.140 -xMD ..... 244
A.2.141 -xMF ..... 244
A.2.142 -xMMD ..... 244
A.2.143 –xMerge ..... 244
A.2.144 -xmaxopt[=*v*] ..... 244
A.2.145 -xmemalign=*ab* ..... 245
A.2.146 -xmodel=[*a*] ..... 246
A.2.147 –xnolib ..... 247
A.2.148 –xnolibmil ..... 248
A.2.149 –xnolibmopt ..... 248
A.2.150 -xnorunpath ..... 249
A.2.151 -xO*level* ..... 249
A.2.152 -xopenmp[= *i*] ..... 251
A.2.153 -xpagesize=*n* ..... 253
A.2.154 -xpagesize_heap=*n* ..... 254
A.2.155 -xpagesize_stack=*n* ..... 254
A.2.156 -xpch=*v* ..... 255
A.2.157 -xpchstop=*file* ..... 257
A.2.158 -xpec[={yes|no}] ..... 258
A.2.159 –xpg ..... 258
A.2.160 -xport64[=(*v* )] ..... 259
A.2.161 -xprefetch[= *a*[ , *a*...]] ..... 261
A.2.162 -xprefetch_auto_type= *a* ..... 263
A.2.163 -xprefetch_level[= *i*] ..... 264
A.2.164 –xprofile=*p* ..... 264
A.2.165 -xprofile_ircache[ =*path*] ..... 267
A.2.166 -xprofile_pathmap ..... 267
A.2.167 -xreduction ..... 268
A.2.168 –xregs=*r*[ ,*r*...] ..... 268
A.2.169 -xrestrict[=*f*] ..... 270
A.2.170 –xs ..... 271

A.2.171 –xsafe=mem ..... 272
A.2.172 –xspace ..... 272
A.2.173 –xtarget=*t* ..... 272
A.2.174 -xthreadvar[= *o*] ..... 276
A.2.175 –xtime ..... 276
A.2.176 -xtrigraphs[={ yes|no}] ..... 277
A.2.177 –xunroll= *n* ..... 278
A.2.178 -xustr={ascii_utf16_ushort |no} ..... 278
A.2.179 -xvector[= *a*] ..... 279
A.2.180 -xvis[={ yes|no}] ..... 280
A.2.181 -xvpara ..... 280
A.2.182 –xwe ..... 280
A.2.183 -Y*c*,*path* ..... 280
A.2.184 -z[ ]*arg* ..... 282

**B Pragma** ..... 283
B.1 Pragma 形式 ..... 283
B.1.1 将函数作为 pragma 参数进行重载 ..... 283
B.2 Pragma 参考 ..... 284
B.2.1 #pragma align ..... 284
B.2.2 #pragma does_not_read_global_data ..... 285
B.2.3 #pragma does_not_return ..... 285
B.2.4 #pragma does_not_write_global_data ..... 285
B.2.5 #pragma dumpmacros ..... 286
B.2.6 #pragma end_dumpmacros ..... 287
B.2.7 #pragma error_messages ..... 287
B.2.8 #pragma fini ..... 287
B.2.9 #pragma hdrstop ..... 288
B.2.10 #pragma ident ..... 288
B.2.11 #pragma init ..... 288
B.2.12 #pragma ivdep ..... 289
B.2.13 #pragma must_have_frame ..... 289
B.2.14 #pragma no_side_effect ..... 289
B.2.15 #pragma opt ..... 290
B.2.16 #pragma pack( *n*) ..... 290

B.2.17 #pragma rarely_called ..... 291
B.2.18 #pragma returns_new_memory ..... 292
B.2.19 #pragma unknown_control_flow ..... 292
B.2.20 #pragma weak ..... 292

词汇表 ..... 295

索引 ..... 301

# 示例


示例 6–1 本地类型用作模板参数问题的示例 ..... 78
示例 6–2 友元声明问题的示例 ..... 79
示例 13–1 `string` 提取运算符 ..... 132
示例 A–1 预处理程序示例 `foo.cc` ..... 162
示例 A–2 使用 `-E` 选项时 `foo.cc` 的预处理程序输出 ..... 162

# 前言

本指南介绍了 Oracle Solaris Studio 12.3 C++ 编译器。

## 受支持的平台

此 Oracle Solaris Studio 发行版支持使用以下体系结构的平台：运行 Oracle Solaris 操作系统的 SPARC 系列的处理器体系结构，以及运行 Oracle Solaris 或特定 Linux 系统的 x86 系列处理器体系结构。

本文档使用以下术语说明 x86 平台之间的区别：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 表示特定的 64 位 x86 兼容 CPU。
- “32 位 x86”指出了有关基于 x86 的系统的特定 32 位信息。

在 SPARC 和 x86 系统中，特定于 Linux 系统的信息仅指受支持的 Linux x86 平台，而特定于 Oracle Solaris 系统的信息仅指受支持的 Oracle Solaris 平台。

有关受支持的硬件平台和操作系统发行版的完整列表，请参见《Oracle Solaris Studio 12.3 发行说明》。

## Oracle Solaris Studio 文档

可以查找 Oracle Solaris Studio 软件的完整文档，如下所述：

- 产品文档位于 Oracle Solaris Studio 文档 Web 站点，包括发行版说明、参考手册、用户指南和教程。
- 代码分析器、性能分析器、线程分析器、dbxtool、DLight 和 IDE 的联机帮助可以在这些工具中通过 "Help"（帮助）菜单以及 F1 键和许多窗口和对话框上的 "Help"（帮助）按钮获取。
- 命令行工具的手册页介绍了工具的命令选项。

# 相关的第三方 Web 站点引用

本文档引用了第三方 URL，以用于提供其他相关信息。

---

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

---

# 开发者资源

对于使用 Oracle Solaris Studio 的开发者，可访问 Oracle 技术网 Web 站点来查找以下资源：

- 有关编程技术和最佳做法的文章
- 软件最新发行版完整文档的链接
- 有关支持级别的信息
- 用户讨论论坛。

# 获取 Oracle 支持

Oracle 客户可通过 My Oracle Support 获取电子支持。有关信息，请访问 `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info`，或访问 `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs`（如果您听力受损）。

# 印刷约定

下表介绍了本书中的印刷约定。

**表 P–1** 印刷约定

| 字体或符号 | 含义 | 示例 |
|---|---|---|
| `AaBbCc123` | 命令、文件和目录的名称；计算机屏幕输出 | 编辑 `.login` 文件。<br>使用 `ls -a` 列出所有文件。<br>`machine_name% you have mail.` |
| **`AaBbCc123`** | 用户键入的内容，与计算机屏幕输出的显示不同 | `machine_name%` **`su`**<br>`Password:` |

表 P–1 印刷约定 （续）

| 字体或符号 | 含义 | 示例 |
| --- | --- | --- |
| *aabbcc123* | 要使用实名或值替换的命令行占位符 | 删除文件的命令为 `rm` *filename*。 |
| *AaBbCc123* | 保留未译的新词或术语以及要强调的词 | 这些称为 *Class* 选项。<br>**注意**：有些强调的项目在联机时以粗体显示。 |
| **新词术语强调** | 新词或术语以及要强调的词 | **高速缓存**是存储在本地的副本。<br>请**勿**保存文件。 |
| 《书名》 | 书名 | 阅读《用户指南》的第 6 章。 |

# 命令中的 shell 提示符示例

下表显示了 Oracle Solaris OS 中包含的缺省 UNIX shell 系统提示符和超级用户提示符。请注意，在命令示例中显示的缺省系统提示符可能会有所不同，具体取决于 Oracle Solaris 发行版。

表 P–2 shell 提示符

| shell | 提示符 |
| --- | --- |
| Bash shell、Korn shell 和 Bourne shell | `$` |
| Bash shell、Korn shell 和 Bourne shell 超级用户 | `#` |
| C shell | `machine_name%` |
| C shell 超级用户 | `machine_name#` |

第 1 部分

# C++ 编译器

◆ ◆ ◆ 第 1 章

# 1 C++ 编译器

本章提供了有关最新 Oracle Solaris Studio C++ 编译器的一般信息。

## 1.1 Oracle Solaris Studio 12.3 C++ 5.12 编译器的新特性和新功能

本节提供了一个汇总列表，介绍了 Oracle Solaris Studio 12.3 C++ 5.12 编译器发行版中的新增特性和功能以及已修改的特性和功能。

- 支持新 SPARC T4 平台：`–xtarget=T4`, `–xchip=T4`, `–xarch=sparc4`
- 支持新 x86 平台 Sandy Bridge / AVX：`–xtarget=sandybridge` `–xchip=sandybridge` `–xarch=avx`
- 支持新 x86 平台 Westmere / AES：`–xtarget=westmere` `–xchip=westmere` `–xarch=aes`
- 新编译器选项 `–g3` 添加了扩展的调试符号表信息。（第 180 页中的 “A.2.31 -g3”）
- 新编译器选项：`–Xlinker` *arg* 将参数传递给链接程序 `ld`(1)。等效于 `–Wl,`*arg*。（第 208 页中的 “A.2.98 -Xlinker *arg*”）
- OpenMP 缺省线程数 `OMP_NUM_THREADS` 现在为 2（以前是 1）。（第 251 页中的 “A.2.152 -xopenmp[= *i*]”）
- 支持 OpenMP 3.1 共享内存并行化规范。（第 251 页中的 “A.2.152 -xopenmp[= *i*]”）
- 新编译器选项 `–xivdep` 设置 `ivdep` pragma 的解释。`ivdep` pragma 指示编译器忽略在循环中找到的部分或全部对数组引用的循环附带依赖性，以进行优化。这使得编译器可以执行各种循环优化，如微向量化、分发、软件流水操作等，其他情况下，无法执行这些优化。当用户知道这些相关项无关紧要或者实际上永远不会发生时，可以使用该指令。（第 236 页中的 “A.2.126 -xivdep[= *p*]”）
- 使用 `–library=sunperf` 可链接到 Sun 性能库。这淘汰了 `-xlic_lib=sunperf`。（第 188 页中的 “A.2.49 -library=*l*[ ,*l*...]”）

- –compat=4 子选项（“兼容模式”）被删除。缺省设置现在为 –compat=5。此外，针对 g++ 源和二进制兼容性的 -compat=g 选项先前仅适用于 Linux 平台，现在已扩展到 Oracle Solaris/x86。（第 158 页中的 “A.2.6 –compat={ 5|g}”）
- 新选项 –features=cplusplus_redef 允许在命令行中通过 –D 选项重新定义以常规方式预定义的宏 __cplusplus。现在仍不允许在源代码中通过 #define 指令重新定义 __cplusplus。此外，–features=%none 和 –features=%all 用法在此发行版中现已过时。（第 167 页中的 “A.2.17 –features=*a*[ ,*a*...]”）
- 新选项 –xanalyze={code| no} 生成对源代码的静态分析，可使用 Oracle Solaris 代码分析器进行查看。（第 210 页中的 “A.2.102 -xanalyze={code| no}”）
- 新子选项 –xbuiltin=%default 仅内联未设置 errno 的函数。errno 的值在任何优化级别始终保持正确，并且可以可靠地对其进行检查。（第 216 页中的 “A.2.108 -xbuiltin[={ %all|%default|%none}]”）
- 支持用户提供的编译器选项缺省值。（第 52 页中的 “3.4 用户提供的缺省选项文件”）
- C99 头文件 stdbool.h 和 C++ 等效项 cstdbool 现在可用。在 C++ 中，头文件不起任何作用，提供它们只是为了与 C99 兼容。

# 1.2 x86 特殊注意事项

- 针对 x86 Oracle Solaris 平台进行编译时，请注意几个重要问题。
- xarch 设置为 -sse、sse2、sse2a、sse3 或更高时编译的程序只能在提供这些扩展和功能的平台上运行。
- 在 x86 上得到的数值结果可能与在 SPARC 上得到的结果不同，这是由 x86 80 位浮点寄存器造成的。为了最大限度减少这些差异，请使用 -fstore 选项或使用 -xarch=sse2 进行编译（如果硬件支持 SSE2）。
- 因为内部数学库（例如，sin(x)）不同，所以 Oracle Solaris 和 Linux 之间的数值结果也会不同。

# 1.3 针对 64 位平台进行编译

使用 -m32 选项针对 ILP32 32 位模型进行编译。使用 -m64 选项针对 LP64 64 位模型进行编译。

ILP32 模型指定 C++ 语言的 int、long 和 pointer 数据类型的宽度均为 32 位。LP64 模型指定 long 和 pointer 数据类型均为 64 位。Oracle Solaris OS 和 Linux OS 还支持 LP64 内存模型下的大型文件和大型数组。

如果使用 -m64 进行编译，则生成的可执行文件仅能在运行 64 位内核的 Oracle Solaris OS 或 Linux OS 下的 64 位 UltraSPARC 或 x86 处理器上运行。64 位对象的编译、链接和执行只能在支持 64 位执行的 Oracle Solaris OS 或 Linux OS 上进行。

## 1.4 二进制兼容验证

在 Oracle Solaris 系统上，以 Oracle Solaris Studio 编译器编译的程序二进制文件都标记了体系结构硬件标志（指示编译的二进制文件所采用的指令集）。运行时，会检查这些标记标志以验证该二进制文件是否可以在它尝试在上面执行的硬件上运行。

如果程序不包含这些体系结构硬件标志，或者如果平台没有启用适当的功能或指令集扩展，则运行此程序可能会导致段故障或错误结果，且不会显示任何显式警告消息。

此警告还会扩展到采用 `.il` 内联汇编语言函数或 `__asm()` 汇编程序代码（使用 SSE、SSE2、SSE2a 和 SSE3 以及更新指令和扩展）的程序。

## 1.5 标准符合性

C++ 编译器 (`CC`) 支持 C++ ISO 国际标准 ISO IS 14882:2003 **编程语言 – *C++***。

在 SPARC 平台上，编译器提供了对 SPARC V8 和 SPARC V9（包括 UltraSPARC 实现）优化开发功能的支持。在 Prentice-Hall for SPARC International 发行的第 8 版 (ISBN 0-13-825001-4) 和第 9 版 (ISBN 0-13-099227-5) SPARC Architecture Manual 中定义了这些功能。

在本文档中，“标准”是指与上面列出的标准版本相一致。“非标准”或“扩展”是指这些标准的这些版本之外的功能。

负责标准的一方可能会不时地修订这些标准。C++ 编译器兼容的适用标准版本可能被修订或替换，这将会导致以后的 Oracle Solaris Studio C++ 编译器发行版本在功能上与旧的发行版本产生不兼容。

## 1.6 发行版信息

《Oracle Solaris Studio 12.3 的新增功能》指南重点介绍了与该编译器发行版相关的重要信息并包括：

- 在手册印刷之后发现的信息
- 新特性和更改的特性
- 软件更正
- 问题和解决办法
- 限制和不兼容
- 可发送库
- 未实现的标准

此发行版的文档索引页上提供了新增功能指南，网址为：`http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`

# 1.7 手册页

联机手册 (man) 页提供了关于命令、函数、子例程以及收集这些信息的文档。

可以通过运行以下命令来显示手册页：

```
example% man topic
```

在整个 C++ 文档中，手册页参考都以主题名称和手册节编号表示：通过 man CC 访问 CC(1)。在 man 命令中使用 -s 选项可以访问除第 1 节之外的各节（例如，ieee_flags(3M)），如下所示：

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

# 1.8 本地语言支持

此发行版本的 C++ 支持使用英语以外的其他语言进行应用程序的开发，包括大多数欧洲语言、中文和日语。因此，您可以十分便捷地将应用程序从一种语言切换到另一种语言。此功能被称为**国际化**。

通常，C++ 编译器按如下方式实现国际化：

- C++ 从国际化的键盘识别 ASCII 字符（也就是说，它具有键盘独立性和 8 位清除）。
- C++ 允许使用本地语言打印某些消息。
- C++ 允许在注释、字符串和数据中使用本地语言。
- C++ 只支持符合扩展 UNIX 字符 (Extended UNIX Characte, EUC) 的字符集，在该字符集中，字符串内每个空字节是一个空字符，而字符串内每个 ASCII 值为 / 的字节是一个 / 字符。

变量名称不能国际化，必须使用英文字符集。

您可以设置语言环境将应用程序从一种本地语言更改为另一种语言。关于这一点和其他本地语言支持功能的信息，请参见操作系统文档。

◆ ◆ ◆ 第 2 章

# 使用 C++ 编译器

本章介绍了如何使用 C++ 编译器。

任何编译器的主要用途是将高级语言（如 C++）编写的程序转换成目标计算机硬件可执行的数据文件。可以使用 C++ 编译器执行以下操作：

- 将源文件转换成可重定位的二进制 (`.o`) 文件，以后链接到可执行文件、静态（归档）库 (`.a`) 文件（使用 `-xar`）或动态（共享）库 (`.so`) 文件
- 将目标文件或库文件（或两者）链接或重链接成可执行文件
- 在运行时调试处于启用状态的情况下编译可执行文件 (`-g`)
- 使用运行时语句或过程级分析 (`-pg`) 编译可执行文件

## 2.1 入门

本节简要概述了如何使用 C++ 编译器编译和运行 C++ 程序。有关命令行选项的完整参考，请参见附录 A，C++ 编译器选项。

---

**注 –** 本章中的命令行示例说明了 `CC` 的用法。打印输出可能会稍有不同。

---

生成和运行 C++ 程序的基本步骤包括以下任务：

1. 使用编辑器创建 C++ 源文件（后缀为表 2–1 中所列有效后缀之一）
2. 调用编译器来生成可执行文件
3. 通过输入可执行文件的名称来启动程序

以下程序在屏幕上显示消息：

```
example% cat greetings.cc
    #include <iostream>
    int main()  {
```

```
      std::cout << “Real programmers write C++!” << std::endl;
      return 0;
    }
example% CC greetings.cc
example% ./a.out
 Real programmers write C++!
example%
```

在此示例中， CC 编译源文件 greetings.cc，并且在缺省情况下编译可执行程序生成文件 a.out。要启动该程序，请在命令提示符下键入可执行文件的名称 a.out。

传统的方法是，UNIX 编译器为可执行文件命名 a.out。每次编译都写入到同一个文件是比较笨拙的方法。另外，如果已经有这样一个文件存在，下次运行编译器时该文件将被覆盖。因此，改用 -o 编译器选项来指定可执行输出文件的名称，如以下示例所示：

```
example% CC– o greetings greetings.cc
```

在此示例中，-o 选项通知编译器将可执行代码写入文件 greetings。（通常的做法是将包含单个源文件的程序的名称指定为源文件的名称，不包括后缀。）

也可以在每次编译后使用 mv 命令来为缺省的 a.out 文件重命名。无论是哪种方法，都可以通过键入可执行文件的名称来运行程序：

```
example% ./greetings
Real programmers write C++!
example%
```

# 2.2 调用编译器

本章其余部分讨论了 CC 命令使用的约定、编译器源代码行指令和其他有关编译器的使用问题。

## 2.2.1 命令语法

编译器命令行的一般语法如下所示：

CC [*options*] [*source-files*] [*object-files*] [*libraries*]

**选项**是前缀为短划线 (–) 或加号 (+) 的选项关键字。某些选项带有参数。

通常，编译器选项的处理顺序是从左到右，从而允许有选择地覆盖宏选项（包含其他选项的选项）。在大多数的情况下，如果您多次指定同一个选项，那么最右边的赋值会覆盖前面的赋值，而不会累积。注意以下特殊情况：

- 所有链接程序选项和 -features、–I -l、–L、-library、–pti、–R、-staticlib、-U、-verbose、-xdumpmacros 和 -xprefetch 选项都会累积，但它们不会覆盖。

- 所有 –U 选项都在所有 –D 选项之后处理。

源文件、目标文件和库按它们在命令行上出现的顺序编译并链接。

在以下示例中，在启用了运行时调试的情况下，使用 CC 编译两个源文件（growth.C 和 fft.C）来生成名为 growth 的可执行文件：

```
example% CC -g -o growth growth.C fft.C
```

## 2.2.2 文件名称约定

命令行上附加在文件名后面的后缀确定了编译器处理文件的方式。如果文件名称的后缀没有在下表中列出，或文件名称没有后缀，那么都要传递到链接程序。

表 2–1 C++ 编译器识别的文件名称后缀

| 后缀 | 语言 | 操作 |
|---|---|---|
| .c | C++ | 以 C++ 源文件编译，将目标文件放在当前目录中；目标文件的缺省名称是源文件名称加上 .o 后缀。 |
| .C | C++ | 操作与 .c 后缀相同。 |
| .cc | C++ | 操作与 .c 后缀相同。 |
| .cpp | C++ | 操作与 .c 后缀相同。 |
| .cxx | C++ | 操作与 .c 后缀相同。 |
| .c++ | C++ | 操作与 .c 后缀相同。 |
| .i | C++ | 将预处理程序输出文件作为 C++ 源文件处理。操作与 .c 后缀相同。 |
| .s | 汇编程序 | 使用汇编程序的汇编源文件。 |
| .S | 汇编程序 | 使用 C 语言预处理程序和汇编程序的汇编源文件。 |
| .il | 内联扩展 | 处理内联扩展的汇编内联模板文件。编译器将使用模板来扩展选定例程的内联调用。（内联模板文件是特殊的汇编文件。请参见 inline(1) 手册页。 |
| .o | 目标文件 | 将目标文件传递到链接程序。 |
| .a | 静态（归档）库 | 将目标库名传递到链接程序。 |
| .so<br>.so.*n* | 动态（共享）库 | 将共享对象的名称传递到链接程序。 |

### 2.2.3 使用多个源文件

C++ 编译器在命令行上接受多个源文件。编译器编译的单个源文件和其直接或间接支持的任何文件一起统称为**编译单元**。C++ 将每个源作为一个单独的编译单元处理。

## 2.3 使用不同编译器版本进行编译

缺省情况下，该编译器不使用高速缓存。仅当指定了 `-instances=extern` 时，才使用高速缓存。如果编译器使用高速缓存，它会检查高速缓存目录的版本，当遇到高速缓存版本问题时会发出错误消息。以后的 C++ 编译器也会检查缓存的版本。例如，具有不同模板缓存版本标识的未来版本编译器在处理此发行版本的编译器生成的缓存目录时，会发出与以下消息类似的错误：

```
Template Database at ./SunWS_cache is incompatible with
this compiler
```

编译器遇到新版本的编译器生成的缓存目录时，也会发出类似的错误。

升级编译器时，最好清除高速缓存。对每个包含模板高速缓存目录的目录运行 `CCadmin -clean`。在大多数情况下，模板高速缓存目录的名称为 `SunWS_cache`。也可以使用 `rm -rf SunWS_cache`。

## 2.4 编译和链接

本节介绍了编译和链接程序的某些方面。在以下示例中，使用 `CC` 编译三个源文件并链接目标文件以生成名为 `prgrm` 的可执行文件。

```
example% CC file1.cc file2.cc file3.cc -o prgrm
```

### 2.4.1 编译和链接序列

在前面的示例中，编译器会自动生成加载器目标文件（`file1.o`、`file2.o` 和 `file3.o`），然后调用系统链接程序来为 `prgrm` 文件创建可执行程序。

编译后，目标文件（`file1.o`、`file2.o` 和 `file3.o`）仍保留不变。此约定使您易于重新链接和重新编译文件。

---

**注 –** 如果只编译了一个源文件，且在同一个操作中链接了程序，则对应的 `.o` 文件将会被自动删除。要保留所有 `.o` 文件，就不要在同一个操作中进行编译和链接，除非编译多个源文件。

---

如果编译失败，您将收到每个错误的对应消息。对于那些出现错误的源文件，不会生成 `.o` 文件，也不会生成可执行程序。

## 2.4.2 分别编译和链接

可以在不同的步骤中进行编译和链接。如果使用 -c 选项，将编译源文件并生成 .o 目标文件，但不创建可执行文件。如果不使用 -c 选项，编译器将调用链接程序。通过将编译和链接步骤分开，仅修复一个文件就不需要完整重新编译。以下示例显示了如何以独立的步骤编译一个文件并与其他文件链接：

```
example% CC -c file1.cc          Make new object file
example% CC -o prgrm file1.o file2.o file3.o        Make executable file
```

请确保链接步骤列出了生成完整程序所需的**全部**目标文件。如果在此步骤中缺少任何目标文件，链接将会失败，并出现 "undefined external reference" 错误（缺少例程）。

## 2.4.3 一致编译和链接

如果在不同的步骤中进行编译和链接，在使用第 43 页中的 “3.3.3 编译时选项和链接时选项”中所列的编译器选项时，一定要在编译和链接时保持一致。

如果使用其中任何选项编译子程序，必须在链接时也使用相同的选项：

- 如果使用 -library 或 -m64/-m32 选项进行编译，则必须在所有 CC 命令中使用相同的选项。
- 如果使用 -p、-xpg 和 -xprofile，则在一个阶段中包括选项而在其他阶段中不包括相应选项并不影响程序的正确性，但不能进行分析。
- 如果使用 -g 和 -g0，则在一个阶段中包括选项而在其他阶段中不包括相应选项并不影响程序的正确性，但会影响调试程序的能力。对于没有使用其中任一选项编译但使用 -g 或 -g0 链接的模块，将无法正确调试。请注意，要对包含函数 main 的模块进行调试，通常必须使用 -g 选项或 -g0 选项对其进行编译。

在以下示例中，使用 -library=stlport4 编译器选项编译程序。

```
example% CC -library=stlport4 sbr.cc -c
example% CC -library=stlport4 main.cc -c
example% CC -library=stlport4 sbr.o main.o -o myprogram
```

如果没有一致地使用 -library=stlport4，程序的某些部分将使用缺省的 libCstd，其他部分将使用可选的替换 STLport 库。生成的程序可能不进行链接，因此在任何情况下都不能正常运行。

如果程序使用模板，某些模板可能会在链接时实例化。在这种情况下，来自最后一行（链接行）的命令行选项将用于编译实例化的模板。

## 2.4.4 针对 64 位内存模型进行编译

使用 -m64 选项可为目标平台指定 64 位内存模型。64 位对象的编译链接和执行只能在支持 64 位执行的 Oracle Solaris 或 Linux 平台上进行。

## 2.4.5 编译器命令行诊断

使用 -V 选项可显示 CC 调用的每个程序的名称和版本号。使用 -v 选项可显示 CC 调用的完整命令行。

使用 –verbose=%all 可显示有关编译器的其他信息。

命令行上编译器无法识别的任何参数都解释为链接程序选项、目标程序文件名或库名称。

基本区别是：

- 对于无法识别的**选项**（前面有短划线 (–) 或加号 (+)），会生成警告。
- 对于无法识别的**非选项**（即前面没有短划线或加号），不会生成警告。然而，这些选项会传递到链接程序。如果链接程序无法识别它们，将会生成链接程序错误消息。

在以下示例中，请注意，CC 无法识别 -bit，该选项传递给链接程序 (ld)，它会尝试解释该选项。因为一个字母的 ld 选项可以连在一起，所以链接程序将 -bit 视为 -b -i -t，所有这些都是合法的 ld 选项。该结果可能并不是您所希望看到的结果：

```
example% CC -bit move.cc       -bit is not a recognized compiler option
CC: Warning: Option -bit passed to ld, if ld is invoked, ignored otherwise
```

在下一个示例中，用户本想键入 CC 选项 -fast，但遗漏了前导短划线。编译器又一次将参数传递到链接程序，而链接程序将参数解释为文件名称：

```
example% CC fast move.cc                    < - The user meant to type -fast
move.CC:
ld: fatal: file fast: cannot open file; errno=2
ld: fatal: File processing errors. No output written to a.out
```

## 2.4.6 了解编译器的组织

C++ 编译器软件包由前端、优化器、代码生成器、汇编程序、模板预链接程序和链接编辑器组成。CC 命令会自动调用其中每个组件，除非使用命令行选项进行其他指定。

因为这些组件中的任何一个都可能生成错误，并且各个组件执行不同的任务，所以识别生成错误的组件可能很有帮助。使用 -v 和 -dryrun 选项可以显示编译器执行期间的更多详细信息。

正如下表所示，不同编译器组件的输入文件拥有不同的文件名后缀。后缀建立了要进行的编译类型。有关文件后缀的含义，请参阅表 2–1。

表 2–2 C++ 编译系统的组件

| 组件 | 说明 | 使用说明 |
|---|---|---|
| ccfe | 前端（编译器预处理程序和编译器） | |
| iropt | 代码优化器 | -xO[2-5], -fast |
| ir2hf | x86：中间语言转换器 | -xO[2-5], -fast |
| inline | SPARC：汇编语言模板的内联扩展 | 指定 .il 文件 |
| fbe | 汇编程序 | |
| cg | SPARC：代码生成器、内联函数、汇编程序 | |
| ube | x86：代码生成器 | -xO[2-5], -fast |
| CClink | 模板预链接程序 | 仅与 -instances=extern 选项一起使用 |
| ld | 链接编辑器 | |

# 2.5 预处理指令和名称

本节讨论了关于预处理 C++ 编译器所特有的指令的信息。

## 2.5.1 Pragma

预处理程序指令 pragma 是 C++ 标准的一部分，但每个编译器中，pragma 的形式、内容和含义都不相同。有关 C++ 编译器可识别的 pragma 的详细信息，请参见附录 B，Pragma。

Oracle Solaris Studio C++ 还支持 C99 关键字 _Pragma。以下两种调用是等效的：

```
#pragma dumpmacros(defs)
_Pragma("dumpmacros(defs)")
```

要使用 _Pragma 而不是 #pragma，请将 pragma 文本写成文字字符串（用括号括起来作为 _Pragma 关键字的一个参数）。

## 2.5.2 具有可变数目的参数的宏

C++ 编译器接受以下形式的 #define 预处理程序指令。

```
#define identifier (...) replacement-list
#define identifier (identifier-list, ...) replacement-list
```

如果列出的宏参数以省略号结尾，那么该宏的调用允许使用除了宏参数以外的其他更多参数。其他参数（包括逗号）收集到一个字符串中，宏替换列表中的名称 `__VA_ARGS__` 可以引用该字符串。

以下示例说明了如何使用可变参数列表的宏。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                          printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n",x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

其结果如下：

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):printf("x is %d but y is %d", x, y));
```

### 2.5.3 预定义的名称

附录中的第 160 页中的 "A.2.8 -D*name*[ =*def*]"介绍了预定义的宏。可以在 `#ifdef` 这样的预处理程序条件中使用这些值。`+p` 选项可防止自动定义 `sun`、`unix`、`sparc` 和 `i386` 预定义宏。

### 2.5.4 警告和错误

`#error` 和 `#warning` 预处理程序指令可用来生成编译时诊断。

`#error` *token-string*　　发送错误诊断 *token-string* 并终止编译操作

`#warning` *token-string*　　发送警告诊断 *token-string* 并继续执行编译操作。

## 2.6 内存要求

编译需要的内存量取决于多个参数，包括：

- 每个过程的大小
- 优化级别
- 为虚拟内存设置的限制
- 磁盘交换文件的大小

在 SPARC 平台上，如果优化器用完了所有内存，那么它将通过在较低优化级别上重试当前过程来尝试恢复。然后优化器将以在命令行上通过 -x0*level* 选项指定的原始级别恢复后续例程。

如果编译包括大量例程的单独源文件，编译器可能会用完所有内存或交换空间。可以尝试降低优化级别。或者，将最大的过程分为其自身的单独文件。

## 2.6.1 交换空间大小

swap -s 命令用于显示可用的交换空间。有关更多信息，请参见 swap(1M) 手册页。

以下示例演示了 swap 命令的用法：

```
example% swap -s
total: 40236k bytes allocated + 7280k reserved = 47516k used, 1058708k available
```

## 2.6.2 增加交换空间

使用 mkfile(1M) 和 swap(1M) 可增加工作站上交换空间的大小。（您必须成为超级用户才能执行该操作）。mkfile 用于命令创建特定大小的文件，而 swap -a 用于将文件增加到系统交换空间：

```
example# mkfile -v 90m /home/swapfile
/home/swapfile 94317840 bytes
example# /usr/sbin/swap -a  /home/swapfile
```

## 2.6.3 虚拟内存的控制

在 -xO3 或更高级别编译非常大型的例程（一个过程中有数几千行代码）需要大量内存。在这种情况下，系统性能可能降低。您可以通过限制单个进程的可用虚拟内存量来控制内存占用。

要在 sh shell 中限制虚拟内存，请使用 ulimit 命令。有关更多信息，请参见 sh(1) 手册页。

以下示例显示了如何将虚拟内存限制为 4GB：

```
example$ ulimit -d 4000000
```

在 csh shell 中，可使用 limit 命令限制虚拟内存。有关更多信息，请参见 csh(1) 手册页。

下一个示例也显示了如何将虚拟内存限制为 4GB：

```
example% limit datasize 4G
```

这些示例都会使优化器在数据空间达到 4GB 时尝试恢复。

虚拟空间的限制不能大于系统总的可用交换空间。在实际使用时，虚拟空间的限制要足够的小，以允许在大型编译过程中正常使用系统。

请确保编译不会消耗一半以上的交换空间。

有 8GB 的交换空间时，请使用以下命令：

在 sh shell 中：

```
example$ ulimit -d 4000000
```

在 csh shell 中：

```
example% limit datasize 4G
```

最佳设置取决于要求的优化程度、实际内存量和可用的虚拟内存量。

### 2.6.4 内存要求

工作站至少应有 2 GB 内存。有关详细要求，请参见产品发行说明。

## 2.7 将 strip 命令用于 C++ 目标

不应将 UNIX strip 命令用于 C++ 目标文件，因为这会导致这些目标文件不可用。

## 2.8 简化命令

可以通过定义特殊的 shell 别名、使用 CCFLAGS 环境变量或使用 make 来简化复杂的编译器命令。

### 2.8.1 在 C Shell 中使用别名

以下示例为带有常用选项的命令定义了别名。

```
example% alias CCfx "CC -fast -xnolibmil"
```

以下示例使用了别名 CCfx。

```
example% CCfx any.C
```

现在命令 CCfx 与以下命令等效：

```
example% CC -fast -xnolibmil any.C
```

## 2.8.2 使用 CCFLAGS 指定编译选项

可以通过设置 CCFLAGS 变量来指定选项。

可以在命令行中显式使用 CCFLAGS 变量。以下示例说明了如何设置 CCFLAGS (C Shell)：

```
example% setenv CCFLAGS '-xO2 -m64'
```

以下示例显式使用了 CCFLAGS。

```
example% CC $CCFLAGS any.cc
```

使用 make 时，如果像上述示例那样设置 CCFLAGS 变量，且 makefile 的编译规则是隐式的，那么调用 make 时生成的编译等效于以下命令：

```
CC -xO2 -m64 files...
```

## 2.8.3 使用 make

make 实用程序是功能非常强大的程序开发工具，可以方便地与所有 Oracle Solaris Studio 编译器一起使用。有关更多信息，请参见 make(1S) 手册页。

### 2.8.3.1 在 make 中使用 CCFLAGS

使用 makefile 的**隐式**编译规则（即没有 *C++* 编译行）时，make 程序会自动使用 CCFLAGS。

◆ ◆ ◆ 第 3 章

# 3 使用 C++ 编译器选项

本章解释了如何使用命令行 C++ 编译器选项，并按功能汇总它们的使用。第 156 页中的 “A.2 选项参考”中对这些选项进行了详细说明。

## 3.1 语法概述

下表显示了本书中使用的典型选项语法格式的示例。

表 3–1 选项语法格式示例

| 语法格式 | 示例 |
|---|---|
| `-option` | `–E` |
| `–option`*value* | `–I`*pathname* |
| `–option=`*value* | `–xunroll=4` |
| `–option` *value* | `–o` *filename* |

圆括号、大括号、方括号、"|" 或 "-" 字符以及省略号是选项描述中使用的**元字符**，而不是选项自身的一部分。有关用法语法的详细说明，请参见本手册前言中的印刷约定。

## 3.2 通用指南

C++ 编译器选项的某些通用指南：

- `-l`*lib* 选项用于与库 `lib`*lib*`.a`（或 `lib`*lib*`.so`）链接。较稳妥的方式是总是将 `-l`*lib* 放在源文件和对象文件后面，这样可以确保库搜索顺序。
- 通常，编译器选项的处理顺序是从左到右（但 `-U` 选项在所有 `-D` 选项之后处理这种情况除外），从而可以有选择地覆盖宏选项（包括其他选项的选项）。此规则不适用于链接程序选项。

- `-features`、`-I -l` 、`-L`、`-library`、`-pti`、`-R`、`-staticlib`、`-U`、`-verbose` 和 `-xprefetch` 选项都会累积，但它们不会覆盖。
- `-D` 选项会累积，但相同名称的多个 `-D` 选项会互相覆盖。

源文件、目标文件和库是按其在命令行上的出现顺序编译和链接。

# 3.3 按功能汇总的选项

在本节中，编译器选项按功能分组以便提供快速参考。有关各个选项的详细说明，请参阅附录 A，C++ 编译器选项。

这些选项适用于除了特别注明之外的所有平台；基于 SPARC 的系统上的 Oracle Solaris OS 特有的功能标识为 *SPARC*，基于 x86 的系统上的 Oracle Solaris OS 特有的功能标识为 *x86*。

## 3.3.1 代码生成选项

**表 3–2** 代码生成选项

| 选项 | 操作 |
|---|---|
| `-compat` | 设置编译器的主发行版本兼容模式。 |
| `-g` | 编译以便用于调试器。 |
| `-KPIC` | 生成与位置无关的代码。 |
| `-Kpic` | 生成与位置无关的代码。 |
| `-mt` | 编译和链接多线程代码。 |
| `-xaddr32` | 将代码限定于 32 位地址空间 (x86/x64) |
| `-xarch` | 指定目标体系结构。 |
| `-xcode=`*a* | (SPARC) 指定代码地址空间。 |
| `-xlinker` | 指定链接程序选项。 |
| `-xMerge` | (SPARC) 将数据段和文本段合并。 |
| `-xtarget` | 指定目标系统。 |
| `-xmodel` | 针对 Solaris x86 平台修改 64 位对象形式 |
| `+w` | 标识可能产生不可预料结果的代码。 |
| `+w2` | 发出由 `+w` 发出的所有警告以及关于技术违规的警告，这些技术违规可能是无害的，但可能会降低程序的最大可移植性。 |

表 3–2 代码生成选项 （续）

| 选项 | 操作 |
| --- | --- |
| -xregs | 如果编译器可以使用更多的寄存器用于临时存储（临时寄存器），那么编译器将能生成速度更快的代码。该选项使得附加临时寄存器可用，而这些附加寄存器通常是不适用的。 |
| -z *arg* | 链接程序选项。 |

## 3.3.2 编译时性能选项

表 3–3 编译时性能选项

| 选项 | 操作 |
| --- | --- |
| -instlib | 禁止生成已出现在指定库中的模板实例。 |
| -m32\|-m64 | 指定编译的二进制对象的内存模型。 |
| -xinstrument | 编译并检测程序，以供线程分析器进行分析。 |
| -xjobs | 设置编译器可以为完成工作而创建的进程数量。 |
| -xpch | 可以减少应用程序的编译时间，这些应用程序的源文件共享一组通用的 include 文件。 |
| -xpchstop | 指定在使用 -xpch 选项创建预编译头文件时要考虑的最后一个 include 文件。 |
| -xprofile_ircache | (SPARC) 重新使用在执行 -xprofile=collect 期间保存的编译数据。 |
| -xprofile_pathmap | (SPARC) 支持单个分析目录中有多个程序或共享库。 |

## 3.3.3 编译时选项和链接时选项

下表列出了在链接时和编译时均必须指定的选项。

表 3–4 编译时选项和链接时选项

| 选项 | 操作 |
| --- | --- |
| -fast | 选择用于控制可执行代码速度的编译选项的最优组合。 |
| -m32\|-m64 | 指定编译的二进制对象的内存模型。 |
| -mt | 扩展为 -D_REENTRANT -lthread 的宏选项。 |
| -xarch | 指定指令集体系结构。 |

表 3–4 编译时选项和链接时选项 （续）

| 选项 | 操作 |
| --- | --- |
| -xautopar | 为多个处理器启用自动并行化。 |
| -xhwcprof | (SPARC) 为基于硬件计数器的分析启用编译器支持。 |
| -xipo | 通过调用过程间分析组件来执行整个程序优化。 |
| -xlinker | 指定链接程序选项。 |
| -xlinkopt | 对可重定位目标文件执行链接时优化。 |
| -xmemalign | (SPARC) 指定假定的最大内存对齐以及未对齐的数据访问的行为。 |
| -xopenmp | 支持用于显式并行化的 OpenMP 接口，包括一组源代码指令、运行时库例程和环境变量。 |
| -xpagesize | 设置堆栈和堆的首选页面大小。 |
| -xpagesize_heap | 设置堆的首选页面大小。 |
| -xpagesize_stack | 设置堆栈的首选页面大小。 |
| -xpg | 准备目标代码来收集数据以使用 gprof(1) 进行分析。 |
| -xprofile | 收集用于分析的数据或使用分析进行优化。 |
| -xvector=lib | 启用对向量库函数调用的自动生成。 |

## 3.3.4 调试选项

表 3–5 调试选项

| 选项 | 操作 |
| --- | --- |
| -### | 等效于 -dryrun。 |
| +d | 不扩展 C++ 内联函数。 |
| -dryrun | 显示驱动程序将向编译的所有组件发出的所有命令。 |
| -E | 仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout。不编译。 |
| -g | 编译以便用于调试器。 |
| -g0 | 编译以便进行调试，但不禁用内联。 |
| -H | 打印包含文件的路径名称。 |
| -keeptmp | 保留编译时创建的临时文件。 |

表 3–5 调试选项 （续）

| 选项 | 操作 |
| --- | --- |
| `-P` | 仅预处理源文件，输出到 `.i` 文件。 |
| `-Qoption` | 直接将选项传递到编译阶段。 |
| `-s` | 从可执行文件中去掉符号表，这样可以保护调试代码的能力。 |
| `-temp=`*dir* | 为临时文件定义目录。 |
| `-verbose=`*vlst* | 控制编译器详细内容。 |
| `-xcheck` | 对堆栈溢出增加一个运行时检查。 |
| `-xdumpmacros` | 打印诸如定义、定义及未定义的位置和已使用的位置的宏信息。 |
| `-xe` | 仅检查语法和语义错误。 |
| `-xhelp=flags` | 显示编译器选项汇总列表。 |
| `-xport64` | 对 32 位体系结构到 64 位体系结构的移植过程中的常见问题发出警告。 |

## 3.3.5 浮点选项

表 3–6 浮点选项

| 选项 | 操作 |
| --- | --- |
| `-fma` | (SPARC) 启用自动生成浮点乘加指令。 |
| `-fns[={no\|yes}]` | (SPARC) 禁用或启用 SPARC 非标准浮点模式。 |
| `-fprecision=`*p* | *x86*：设置浮点精度模式。 |
| `-fround=`*r* | 设置启动时生效的 IEEE 舍入模式。 |
| `-fsimple=`*n* | 设置浮点优化首选项。 |
| `-fstore` | *x86*：强制浮点表达式的精度。 |
| `-ftrap=`*tlst* | 设置启动时生效的 IEEE 陷阱操作模式。 |
| `-nofstore` | *x86*：禁用表达式的强制精度。 |
| `-xlibmieee` | 使 `libm` 在异常情况下对于数学例程返回 IEEE 754 值。 |

## 3.3.6 语言选项

表 3–7 语言选项

| 选项 | 操作 |
| --- | --- |
| `-compat` | 设置编译器的主发行版本兼容模式。 |
| `-features=`*alst* | 启用或禁用各种 C++ 语言特性。 |
| `-xchar` | 在 char 类型定义为无符号的系统上，简化代码的移植。 |
| `-xldscope` | 控制变量和函数定义的缺省链接程序范围，以创建更快更安全的共享库。 |
| `-xthreadvar` | (SPARC) 更改缺省的线程局部存储访问模式。 |
| `-xtrigraphs` | 启用三字母序列的识别。 |
| `-xustr` | 启用识别由 16 位字符构成的文本字符串。 |

## 3.3.7 库选项

表 3–8 库选项

| 选项 | 操作 |
| --- | --- |
| `-B`*binding* | 请求符号、动态或静态库链接。 |
| `-d{y\|n}` | 允许或不允许整个可执行文件的动态库。 |
| `-G` | 生成动态共享库来取代可执行文件。 |
| `-h`*name* | 为生成的动态共享库指定内部名称。 |
| `-i` | 通知 `ld`(1) 忽略任何 `LD_LIBRARY_PATH` 设置。 |
| `-L`*dir* | 将 *dir* 添加到要在其中搜索库的目录列表。 |
| `-l`*lib* | 将 `lib`*lib*`.a` 或 `lib`*lib*`.so` 添加到链接程序的库搜索列表。 |
| `-library=`*llst* | 强制将特定库和相关文件包含到编译和链接中。 |
| `-mt` | 编译和链接多线程代码。 |
| `-norunpath` | 不将库的路径生成到可执行文件中。 |
| `-R`*plst* | 将动态库搜索路径生成到可执行文件中。 |
| `-staticlib=`*llst* | 说明哪些 C++ 库是静态链接的。 |
| `-xar` | 创建归档库。 |

表 3–8 库选项 （续）

| 选项 | 操作 |
|---|---|
| -xbuiltin[=*opt*] | 启用或禁用标准库调用的更多优化。 |
| -xia | (Solaris) 链接合适的区间运算库并设置适当的浮点环境。 |
| -xlang=*l*[,*l*] | 包含适当的运行库，并确保指定语言的正确运行时环境。 |
| -xlibmieee | 使 libm 在异常情况下对于数学例程返回 IEEE 754 值。 |
| -xlibmil | 内联选定的 libm 库例程以进行优化。 |
| -xlibmopt | 使用优化数学例程的库。 |
| -xnolib | 禁止链接缺省系统库。 |
| -xnolibmil | 在命令行上取消 –xlibmil。 |
| -xnolibmopt | 不使用数学例程库。 |

## 3.3.8 已过时选项

**注 –** 以下选项当前已过时（因此编译器不再接受它们）或者在以后的发行版中可能会被删除。

表 3–9 已过时选项

| 选项 | 操作 |
|---|---|
| -features=[%all\|%none] | 过时的子选项 %all 和 %none。 |
| -library=%all | 过时的子选项，在以后的发行版本中可能会被删除。 |
| -xlic_lib=sunperf | 使用 –library=sunperf 可链接到 Sun 性能库。 |
| -xlicinfo | 已过时。 |
| -xnativeconnect | 已废弃，没有替代选项。 |
| -xprefetch=yes | 改用 -xprefetch=auto,explicit。 |
| -xprefetch=no | 改用 -xprefetch=no%auto,no%explicit。 |
| -xvector=yes | 改用 -xvector=lib。 |
| -xvector=no | 改用 -xvector=none。 |

## 3.3.9 输出选项

表 3–10 输出选项

| 选项 | 操作 |
|---|---|
| -c | 仅编译；生成目标 (.o) 文件，但抑制链接。 |
| -dryrun | 显示但不编译由驱动程序传递到编译器的所有命令行。 |
| -E | 仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout。不编译。 |
| -erroff | 禁止编译器警告消息。 |
| -errtags | 显示每条警告消息的消息标记。 |
| -errwarn | 如果发出指示的警告消息，编译器将以失败状态退出。 |
| -filt | 禁止编译器应用到链接程序错误消息的过滤。 |
| –G | 生成动态共享库来取代可执行文件。 |
| –H | 输出被包含文件的路径名称。 |
| –migration | 解释可以从早期编译器获得有关移植信息的位置。 |
| –o *filename* | 将输出文件或可执行文件的名称设置为 *filename*。 |
| –P | 仅预处理源文件，输出到 .i 文件。 |
| –Qproduce *sourcetype* | 使 CC 驱动程序生成类型为 *sourcetype* 的输出。 |
| –s | 从可执行文件去掉符号表。 |
| –verbose=*vlst* | 控制编译器详细内容。 |
| +w | 必要时打印附加警告。 |
| +w2 | 适当时仍输出更多警告。 |
| –w | 抑制警告消息。 |
| -xdumpmacros | 打印诸如定义、定义及未定义的位置和已使用的位置的宏信息。 |
| -xe | 仅对源文件执行语法和语义检查，但不生成任何对象或可执行代码。 |
| –xhelp=flags | 显示编译器选项汇总列表。 |
| –xM | 输出 makefile 依赖性信息。 |
| –xM1 | 生成相关项信息，但排除 /usr/include。 |
| –xtime | 报告每个编译阶段的执行时间。 |

表 3–10 输出选项 （续）

| 选项 | 操作 |
| --- | --- |
| –xwe | 将所有的警告转换为错误。 |
| -z *arg* | 链接程序选项。 |

## 3.3.10 运行时性能选项

表 3–11 运行时性能选项

| 选项 | 操作 |
| --- | --- |
| –fast | 选择编译选项的组合以优化某些程序的执行速度。 |
| -fma | (SPARC) 启用自动生成浮点乘加指令。 |
| -g | 指示编译器和链接程序准备程序以进行性能分析（以及调试）。 |
| -s | 从可执行文件去掉符号表。 |
| -m32\|-m64 | 指定编译的二进制对象的内存模型。 |
| -xalias_level | 启用编译器执行基于类型的别名分析和优化。 |
| -xarch=*isa* | 指定目标体系结构指令集。 |
| -xbinopt | 准备二进制文件以便以后进行优化、转换和分析。 |
| -xbuiltin[=*opt*] | 启用或禁用标准库调用的更多优化。 |
| -xcache=*c* | (SPARC) 定义优化器的目标高速缓存属性。 |
| -xchip=*c* | 指定目标处理器芯片。 |
| -xF | 启用函数和变量的链接程序重新排序。 |
| -xinline=*flst* | 指定用户编写的哪些例程可以被优化器内联 |
| -xipo | 执行过程间优化。 |
| -xlibmil | 内联选定的 libm 库例程以进行优化。 |
| -xlibmopt | 使用优化数学例程的库。 |
| -xlinkopt | (SPARC) 除了目标文件中的所有优化之外，对生成的可执行文件或动态库执行链接时优化。 |
| -xmemalign=*ab* | (SPARC) 指定假定的最大内存对齐以及未对齐的数据访问的行为。 |
| -xnolibmil | 在命令行上取消 – xlibmil。 |

表 3–11 运行时性能选项 （续）

| 选项 | 操作 |
| --- | --- |
| -xnolibmopt | 不使用数学例程库。 |
| -xO*level* | 将优化级别指定为 *level*。 |
| -xpagesize | 设置堆栈和堆的首选页面大小。 |
| -xpagesize_heap | 设置堆的首选页面大小。 |
| -xpagesize_stack | 设置堆栈的首选页面大小。 |
| -xprefetch[=*lst*] | 在支持预取的体系结构上启用预取指令。 |
| -xprefetch_level | 控制 -xprefetch=auto 设置的自动插入预取指令的主动性。 |
| -xprofile | 收集运行时分析数据或使用运行时分析数据进行优化。 |
| -xregs=*rlst* | 控制临时寄存器的使用。 |
| -xsafe=mem | (SPARC) 不允许有基于内存的陷阱。 |
| -xspace | (SPARC) 不允许会增大代码大小的优化。 |
| -xtarget=*t* | 指定目标指令集和优化系统。 |
| -xthreadvar | 更改缺省的线程局部存储访问模式。 |
| -xunroll=*n* | 启用在可能的场合下解开循环。 |
| -xvis | (SPARC) 使编译器可以识别 VIS 指令集中定义的汇编语言模板。 |

## 3.3.11 预处理程序选项

表 3–12 预处理程序选项

| 选项 | 操作 |
| --- | --- |
| -D*name*[= *def*] | 为预处理程序定义符号 *name*。 |
| -E | 仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout。不编译。 |
| -H | 输出被包含文件的路径名称。 |
| -P | 仅预处理源文件，输出到 .i 文件。 |
| -U*name* | 删除预处理程序符号 *name* 的初始定义。 |
| -xM | 输出 makefile 依赖性信息。 |
| -xM1 | 生成依赖性信息，但排除 /usr/include。 |

## 3.3.12 分析选项

**表 3–13** 分析选项

| 选项 | 操作 |
|---|---|
| `-p` | 准备目标代码来收集数据以使用 `prof` 进行分析。 |
| `-xpg` | 编译以便使用 `gprof` 分析器进行分析。 |
| `-xprofile` | 收集运行时分析数据或使用运行时分析数据进行优化。 |

## 3.3.13 参考选项

**表 3–14** 参考选项

| 选项 | 操作 |
|---|---|
| `-xhelp=flags` | 显示编译器选项汇总列表。 |

## 3.3.14 源文件选项

**表 3–15** 源文件选项

| 选项 | 操作 |
|---|---|
| `-H` | 输出被包含文件的路径名称。 |
| `-I`*pathname* | 将 *pathname* 添加到 `include` 文件搜索路径。 |
| `-I-` | 更改包含文件搜索规则 |
| `-xM` | 输出 makefile 依赖性信息。 |
| `-xM1` | 生成依赖性信息，但排除 `/usr/include`。 |

## 3.3.15 模板选项

**表 3–16** 模板选项

| 选项 | 操作 |
|---|---|
| `-instances=`*a* | 控制模板实例的放置和链接。 |
| `-template=`*wlst* | 启用或禁用各种模板选项。 |

### 3.3.16 线程选项

表 3–17 线程选项

| 选项 | 操作 |
|---|---|
| `-mt` | 编译和链接多线程代码。 |
| `-xsafe=mem` | (SPARC) 不允许有基于内存的陷阱。 |
| `-xthreadvar` | (SPARC) 更改缺省的线程局部存储访问模式。 |

## 3.4 用户提供的缺省选项文件

通过缺省编译器选项文件，用户可以指定一组应用于所有编译的缺省选项，除非另行覆盖。例如，该文件可以指定所有编译的缺省级别为 –xO2 ，或自动包括文件 `setup.il`。

启动时，编译器会搜索缺省选项文件，并列出应对所有编译包含的缺省选项。环境变量 `SPRO_DEFAULTS_PATH` 可指定要在其中搜索缺省文件的冒号分隔目录列表。

如果该环境变量未设置，则会使用一组标准缺省设置。如果该环境变量已设置但为空，则不会使用任何缺省设置。

缺省文件名的格式必须是 compiler.defaults，其中 compiler 为以下值之一：`cc`、`c89`、`c99`、`CC`、`ftn` 或 `lint`。例如，C++ 编译器的缺省值为 `CC.defaults`。

如果在 `SPRO_DEFAULTS_PATH` 列出的目录中找到编译器的缺省文件，编译器将读取该文件并在命令行上处理各选项之前先处理这些选项。系统将使用找到的第一个缺省文件，并且会终止搜索。

系统管理员可以在 *Studio-install-path*`/prod/etc/config` 中创建系统范围的缺省文件。如果设置了该环境变量，则不会读取已安装的缺省文件。

缺省文件的格式与命令行类似。该文件的每一行都可以包含一个或多个由空格分隔的编译器选项。Shell 扩展（例如通配符和替换）将不会应用于缺省文件中的选项。

`SPRO_DEFAULTS_PATH` 的值和完全扩展的命令行将显示在由选项 `–#`, `–###`, 和 `–dryrun` 生成的详细输出中。

用户在命令行上指定的选项通常会优先于从缺省文件读取的选项。例如，如果缺省文件指定使用 –xO4 进行编译，而用户在命令行上指定 –xO2，则会使用 –xO2。

缺省选项文件中显示的某些选项将附加在命令行中指定的选项之后。这些选项包括预处理程序选项 `–I`、链接程序选项 `–B`、`–L`、`–R` 和 `–l` 以及所有文件参数（例如源文件、目标文件、归档和共享对象）。

以下是如何使用用户提供的缺省编译器选项启动文件的示例。

```
demo% cat /project/defaults/CC.defaults
-I/project/src/hdrs –L/project/libs –llibproj –xvpara
demo% setenv SPRO_DEFAULTS_PATH  /project/defaults
demo% CC –c –I/local/hdrs –L/local/libs –lliblocal tst.c
```

此命令现在等效于：

```
CC -fast –xvpara –c –I/local/hdrs –L/local/libs –lliblocal tst.c \
     –I/project/src/hdrs –L/project/libs –llibproj
```

尽管编译器缺省文件提供了可为整个项目设置缺省值的便利方法，但它也可能成为问题难以诊断的原因。将环境变量 SPRO_DEFAULTS_PATH 设置为当前目录以外的绝对路径可避免出现此类问题。

缺省选项文件的接口稳定性未确定。选项处理顺序在以后的发行版中可能会更改。

第 2 部分

# 编写 C++ 程序

◆ ◆ ◆ 第 4 章

# 4 语言扩展

本章介绍了与此编译器相关的语言扩展。在命令行上指定某些编译器选项之后，编译器才能识别本章中描述的某些功能。相关编译器选项在相应章节中列出。

使用 `-features=extensions` 选项可以编译其他 C++ 编译器通常接受的非标准代码。必须编译无效代码且不允许修改代码而使之有效时，您可以使用该选项。

本章介绍了使用 `-features=extensions` 选项时编译器支持的语言扩展。

---

**注 –** 可以很容易的将每个支持无效代码的实例转变为所有编译器接受的有效代码。如果允许使代码有效，那么您应该使代码有效而不是使用该选项。使用 `-features=extensions` 选项可以使某些编译器拒绝的无效代码永远存在。

---

## 4.1 链接程序作用域

可使用下列声明说明符来协助约束外部符号的声明和定义。文件链接到共享库或可执行文件之前，静态归档或目标文件指定的作用域限制不会生效。尽管如此，编译器仍然可以执行显示链接程序作用域说明符的某些优化。

通过使用这些说明符，您不必再使用链接程序作用域的 mapfile。也可以通过在命令行上指定 `-xldscope` 来控制变量作用域的缺省设置。

有关更多信息，请参见第 239 页中的 “A.2.130 -xldscope={*v*}”。

**表 4–1** 链接程序作用域声明说明符

| 值 | 含义 |
|---|---|
| `__global` | 符号定义具有全局链接程序作用域，是限制最小的链接程序作用域。对符号的所有引用都绑定到定义符号的第一个动态装入模块中的定义。该链接程序作用域是外部符号的当前链接程序作用域。 |

表 4–1 链接程序作用域声明说明符 （续）

| 值 | 含义 |
| --- | --- |
| `__symbolic` | 符号定义具有符号链接程序作用域，其限制程度高于全局链接程序作用域。将对链接的动态装入模块内符号的所有引用绑定到模块内定义的符号。在模块外部，符号也显示为全局符号。该链接程序作用域对应于链接程序选项 `-Bsymbolic`。尽管不能将 `-Bsymbolic` 与 C++ 库一起使用，但可以使用 `__symbolic` 说明符，而不会引起问题。有关链接程序的更多信息，请参见 `ld`(1) 手册页。 |
| `__hidden` | 符号定义具有隐藏的链接程序作用域。隐藏链接程序作用域具有比符号和全局链接程序作用域更高的限制。将动态装入模块内的所有引用绑定到该模块内的定义。符号在模块外部是不可视的。 |

符号定义可以用更多限制的说明符来重新声明，但是不可以用较少限制的说明符重新声明。符号定义后，不可以用不同的说明符声明符号。

`__global` 是限制最少的作用域，`__symbolic` 是限制较多的作用域，而 `__hidden` 是限制最多的作用域。

因为虚函数的声明影响虚拟表的结构和解释，所以所有虚函数对包括类定义的所有编译单元必须是可视的。

可以将链接程序作用域说明符应用于结构、类、联合声明和定义中，因为 C++ 类可能要求生成隐式信息，如虚拟表和运行时类型信息。在这种情况下，说明符后跟结构、类或联合关键字。这种应用程序为其所有隐式成员隐含了相同的链接程序作用域。

### 4.1.1 与 Microsoft Windows 兼容

为了在动态库方面与 Microsoft Visual C++ (MSVC++) 中的相似作用域功能兼容，也支持以下语法：

`__declspec(dllexport)` 等效于 `__symbolic`

`__declspec(dllimport)` 等效于 `__global`

在 Oracle Solaris Studio C++ 中使用此语法时，应将选项 `-xldscope=hidden` 添加到 `CC` 命令行。结果与使用 MSVC++ 得到的结果相当。在 MSVC++ 中，`__declspec(dllimport)` 应当仅用于外部符号的声明，而不用于定义。示例：

```
__declspec(dllimport) int foo(); // OK
__declspec(dllimport) int bar() { ... } // not OK
```

在 MSVC++ 中，对于将 `dllimport` 用于定义没有严格规定，而使用 Oracle Solaris Studio C++ 时结果则不同。尤其是，使用 Oracle Solaris Studio C++ 时将 `dllimport` 用于定义得到的是具有全局链接的符号而不是符号链接。Microsoft Windows 上的动态库不支持符号的全局链接。如果遇到此问题，可以更改源代码，对定义使用 `dllexport` 而不是 `dllimport`。这样，使用 MSVC++ 和使用 Oracle Solaris Studio C++ 得到的结果相同。

## 4.2 线程局部存储

通过声明线程局部变量，可以利用线程局部存储。线程局部变量声明普通变量声明与声明说明符 `__thread` 组成。有关更多信息，请参见第 276 页中的 “A.2.174 -xthreadvar[=*o*]”。

必须将 `__thread` 说明符包括在第一个线程变量声明中。使用 `__thread` 说明符声明的变量的绑定方式与没有 `__thread` 说明符时相同。

只能使用 `__thread` 说明符声明静态持续时间的变量。具有静态持续时间的变量包括了文件全局、文件静态、函数局部静态和类静态成员。不能使用 `__thread` 说明符声明动态或自动持续时间的变量。线程变量可以具有静态初始化函数，但是不可以具有动态初始化函数或析构函数。例如，允许 `__thread int x = 4;`，但不允许 `__thread int x = f();`。线程变量不能包含具有重要构造函数和析构函数的类型。具体来说，就是线程变量的类型不能为 `std::string`。

运行时对线程变量的地址运算符 (&) 求值并返回当前线程变量的地址。因此，线程变量的地址不是常量。

线程变量的地址在相应线程的生命周期中是稳定的。进程中任何线程都可以在线程变量的生命周期任意使用该变量的地址。不能在线程终止后使用线程变量的地址。线程变量的所有地址在线程终止后都是无效的。

## 4.3 用限制较少的虚函数覆盖

C++ 标准规定，覆盖虚拟函数在异常中允许的限制不得低于它覆盖的任何函数的限制。该虚函数可能与覆盖的任何函数具有相同或更多的限制。注意，不存在异常规范也允许任何异常。

例如，假定通过指向基类的指针调用函数。如果函数具有异常规范，则可以计算出没有其他正抛出的异常。如果覆盖函数具有限制较少的规范，则不可预料的异常可能会被抛出，这会导致意外的程序行为并且终止程序。

使用 `-features=extensions` 时，编译器允许覆盖异常规范限制较小的函数。

# 4.4 对 enum 类型和变量进行前向声明

使用 -features=extensions 时，编译器允许对 enum 类型和变量进行前向声明。此外，编译器允许声明不完整 enum 类型的变量。编译器总是假定不完整 enum 类型的大小和范围与当前平台上的 int 类型相同。

以下是两行无效代码示例，如果使用 -features=extensions 选项，可对其进行编译。

```
enum E; // invalid: forward declaration of enum not allowed
E e;    // invalid: type E is incomplete
```

因为 enum 定义不能互相引用，并且 enum 定义不能交叉引用另一种类型，所以从来不必对枚举类型进行前向声明。要使代码有效，可以总是先提供 enum 的完整定义，然后再使用它。

---

**注 –** 在 64 位体系结构上，enum 要求的大小可以比 int 类型大。如果是这种情况，并且如果前向声明和定义在同一编译中是可视的，那么编译器将发出错误。如果实际大小不是假定的大小并且编译器没有发现这个差异，那么代码将编译并链接，但有可能不能正常运行。可能出现意外的程序行为，尤其是 8 字节值存储在 4 字节变量中时。

---

# 4.5 使用不完整 enum 类型

使用 -features=extensions 时，不完整的 enum 类型以前向声明处理。例如，以下是无效代码，如果使用 -features=extensions 选项，可对其进行编译。

```
typedef enum E F; // invalid, E is incomplete
```

如前所述，可以总是先包括 enum 类型的定义，然后再使用。

# 4.6 将 enum 名称作为作用域限定符

因为 enum 声明并不引入作用域，所以 enum 名称不能作为作用域限定符来使用。例如，以下代码是无效的。

```
enum E {e1, e2, e3};
int i = E::e1; // invalid: E is not a scope name
```

要编译该无效代码，请使用 -features=extensions 选项。-features=extensions 选项指示编译器在作用域限定符是 enum 类型的名称时忽略该作用域限定符。

要使代码有效，请删除无效的限定符 E::。

**注 –** 使用该选项提高了排字错误的可能性，产生了编译没有错误消息的错误程序。

# 4.7 使用匿名 struct 声明

匿名结构声明是既不声明结构标记也不声明对象或 typedef 名称的声明。C++ 中不允许匿名结构。

-features=extensions 选项允许使用匿名 struct 声明，但仅作为联合的成员。

以下代码是无效匿名 struct 声明示例，如果使用 -features=extensions 选项，可对其进行编译。

```
union U {
  struct {
    int a;
    double b;
  };  // invalid: anonymous struct
  struct {
    char* c;
    unsigned d;
  };  // invalid: anonymous struct
};
```

struct 成员的名称是可视的，没有 struct 成员名称的限定。如果该代码示例中提供了 U 的定义，则可以编写：

```
U u;
u.a = 1;
```

匿名结构与匿名联合服从相同的限制。

请注意，可以通过为每个 struct 提供一个名称以使代码有效，例如：

```
union U {
  struct {
    int a;
    double b;
  } A;
  struct {
    char* c;
    unsigned d;
  } B;
};
U u;
U.A.a = 1;
```

# 4.8 传递匿名类实例的地址

不允许获取临时变量的地址。例如，因为以下代码获取了构造函数调用创建的变量地址，所以这些代码是无效的。但是，如果使用 -features=extensions 选项，编译器将接受该无效代码。

```
class C {
  public:
    C(int);
    ...
};
void f1(C*);
int main()
{
  f1(&C(2)); // invalid
}
```

注意，可以通过使用显式变量来使该代码有效。

```
C c(2);
f1(&c);
```

函数返回时，临时对象被销毁。程序员应确保临时变量的地址没有留下。此外，销毁临时变量（例如 f1）时，临时变量中存储的数据会丢失。

# 4.9 将静态名称空间作用域函数声明为类友元

下面的代码是无效的：

```
class A {
  friend static void foo(<args>);
  ...
};
```

因为类名具有外部链接并且所有定义必须是相等的，所以友元函数也必须具有外部链接。但是，如果使用 -features=extensions 选项，编译器将接受该代码。

程序员处理该无效代码的方法大概是在类 A 的实现文件中提供非成员 "helper" 函数。可以通过使 foo 成为静态成员函数得到相同效果。如果不要客户端调用函数，则可以使该函数私有化。

---

**注 –** 如果使用该扩展，则任何客户端都可以“劫取”您的类。任何客户端都可以包括类的头文件，然后定义其自身的静态函数 foo，该函数将自动成为类的友元。结果就好像是您使类的所有成员成为了公共的。

---

# 4.10 将预定义 __func__ 符号用于函数名

编译器将每个函数中的标识符 __func__ 隐式声明为静态 const char 数组。如果程序使用标识符，编译器还会提供以下定义，其中，*function-name* 是函数原始名称。类成员关系、名称空间和重载不反映在名称中。

```
static const char __func__[] = "function-name";
```

例如，请考虑以下代码段。

```
#include <stdio.h>
void myfunc(void)
{
  printf("%s\n", __func__);
}
```

每次调用函数时，函数将把以下内容打印到标准输出流。

```
myfunc
```

还会定义标识符 __FUNCTION__，它等效于 __func__。

# 4.11 支持的属性

支持以下属性：为了实现兼容，编译器会实现由 __attribute__ (( *keyword*)) 或 [ [*keyword*]] 调用的以下属性。还接受在双下划线内拼写的属性关键字 *__keyword__*。

| | |
|---|---|
| aligned | 大致等效于 #pragma align。生成警告，且在用于可变长度数组时会被忽略。 |
| always_inline | 等效于 #pragma inline 和 -xinline |
| const | 等效于 #pragma no_side_effect |
| constructor | 等效于 #pragma init |
| destructor | 等效于 #pragma fini |
| malloc | 等效于 #pragma returns_new_memory |
| mode | （无等效子句） |
| noinline | 等效于 #pragma no_inline 和 -xinline |
| noreturn | 等效于 #pragma does_not_return |
| pure | 等效于 #pragma does_not_write_global_data |
| packed | 等效于 #pragma pack()。请参见以下详细信息。 |
| returns_twice | 等效于 #pragma unknown_control_flow |

| | |
|---|---|
| `strong` | 接受它是为了与 g++ 兼容，但不起任何作用。g++ 文档建议不要使用该属性。 |
| `vector_size` | 指示变量或类型名称（使用 `typedef` 创建）表示一个向量。 |
| `visibility` | 提供链接程序作用域。（请参见第 239 页中的 “A.2.130 -xldscope={*v*}”）语法为：`__attribute__((visibility(“`*visibility-type*`”)))`，其中 *visibility-type* 是以下选项之一： |
| | `default` 与 `__global` 链接程序作用域相同 |
| | `hidden` 与 `__hidden` 链接程序作用域相同 |
| | `internal` 与 `__symbolic` 链接程序作用域相同 |
| `weak` | 等效于 `#pragma weak` |

## 4.11.1 __packed__ 属性详细信息

此属性附加于 `struct` 或 `union` 类型定义中，它指定结构或联合的每个成员（除了零宽度位字段）的放置，以最大限度地减小所需内存。附加于 `enum` 定义时，`__packed__` 指示应使用最小的整数类型。

为 `struct` 和 `union` 类型指定此属性等效于对每个结构或联合成员指定 `packed` 属性。

在以下示例中，`struct my_packed_struct` 的成员紧紧打包在一起，但其成员的内部布局并不打包。为此，还需要打包 `struct my_unpacked_struct`。

```
struct my_unpacked_struct
{
   char c;
   int i;
;

struct __attribute__ ((__packed__)) my_packed_struct
{
   char c;
   int  i;
   struct my_unpacked_struct s;
};
```

只能对 `enum`、`struct` 或 `union` 的定义指定此属性，不能对未定义枚举类型、结构或联合的 `typedef` 指定此属性。

# 4.12 对 Intel MMX 和扩展的 x86 平台内部函数的编译器支持

mmintrin.h 头文件中声明的原型支持 Intel MMX 内部函数，是为了实现兼容性而提供的。

特定头文件提供了附加扩展平台内部函数的原型，如下表所示。

**表 4–2** 头文件

| x86 平台 | 头文件 |
|---|---|
| SSE | mmintrin.h |
| SSE2 | xmmintrin.h |
| SSE3 | pmmintrin.h |
| SSSE3 | tmmintrin.h |
| SSE4A | ammintrin.h |
| SSE4.1 | smmintrin.h |
| SSE4.2 | nmmintrin.h |
| AES 加密和 PCLMULQDQ | wmmintrin.h |
| AVX | immintrin.h |

表中每个头文件在其前面都包括了原型。例如，在 SSE4.1 平台上，用户程序中的包含方 smmintrin.h 声明了支持 SSE4.1、SSSE3、SSE3、SSE2、SSE 和 MMX 平台的内部函数名称，因为 smmintrin.h 包含了 tmmintrin.h，而 tmmintrin.h 包含了 pmmintrin.h，并依次向下类推至 mmintrin.h。

请注意，ammintrin.h 是由 AMD 发布的，未包含在任何 Intel 内部函数头文件中。ammintrin.h 包含了 pmmintrin.h，因此，通过将 ammintrin.h 包含在内，就可声明所有 AMD SSE4A 以及 Intel SSE3、SSE2、SSE 和 MMX 函数。

另外，单个 Oracle Solaris Studio 头文件 sunmedia_intrin.h 包含所有 Intel 头文件中的声明，但未包含 AMD 头文件 ammintrin.h。

请注意，在主机平台（例如 SSE3）上部署的调用任何超集内部函数（例如，针对 AVX）的代码不能在 Oracle Solaris 平台上装入，在 Linux 平台上装入可能会失败并产生未定义的行为或不正确的结果。对于调用这些特定于平台的内部函数的程序，请只在支持这些函数的平台上部署这类程序。

这些为系统头文件，应按下例所示显示在程序中：

```
#include <nmmintrin.h>
```

有关详细信息，请参阅最新的 Intel C++ 编译器参考指南。

◆ ◆ ◆ 第 5 章

# 5 程序组织

C++ 程序的文件组织需要比典型的 C 程序更加小心。本章介绍了如何建立头文件和模板定义。

## 5.1 头文件

创建有效的头文件是很困难的。头文件通常必须适应 C 和 C++ 的不同版本。要提供模板，请确保头文件能容纳多个包含（幂等）。

### 5.1.1 可适应语言的头文件

可能需要开发能够包含在 C 和 C++ 程序中的头文件。但是，称为“传统 C”的 Kernighan 和 Ritchie C (K&C)、ANSI C、 Annotated Reference Manual C++ (ARM C++) 以及 ISO C++ 有时要求一个头文件中同一个程序元素有不同的声明或定义。（有关语言和版本之间的变化的其他信息，请参见《C++ 迁移指南》。）要使头文件符合所有这些标准，可能需要根据预处理程序宏 `__STDC__` 和 `__cplusplus` 的存在情况或值来使用条件编译。

在 K&R C 中没有定义宏 `__STDC__`，但在 ANSI C 和 C++ 中都对其进行了定义。可使用该宏将 K&R C 代码与 ANSI C 或 C++ 代码区分开。该宏最适用于从非原型函数定义中区分原型函数定义。

```
#ifdef __STDC__
int function(char*,...);      // C++ & ANSI C declaration
#else
int function();               // K&R C
#endif
```

在 C 中没有定义宏 `__cplusplus`，但在 C++ 中对其进行了定义。

**注 –** 早期版本的 C++ 定义了宏 `c_plusplus`，但没有定义 `cplusplus`。现在已不再定义宏 `c_plusplus`。

可使用 `__cplusplus` 宏的定义来区分 C 和 C++。该宏在保证为函数声明指定 `extern "C"` 接口时非常有用，如以下示例所示。为了防止出现 `extern "C"` 指定不一致，切勿将 `#include` 指令放在 `extern "C"` 链接指定的作用域中。

```
#include “header.h”
...                        // ... other include files...
#if defined(__cplusplus)
extern “C” {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

在 ARM C++ 中，`__cplusplus` 宏的值为 1。在 ISO C++ 中，该宏的值为 199711L（用 `long` 常量表示的标准年月）。使用这个宏的值区分 ARM C++ 和 ISO C++。这个宏值在保护模板语法的更改时极为有用。

```
// template function specialization
#if __cplusplus < 199711L
int power(int,int);                        // ARM C++
#else
template <> int power(int,int);            // ISO C++
#endif
```

## 5.1.2 幂等头文件

头文件应该具有幂等性，即，多次包括头文件的效果和仅包括一次的效果完全相同。该特性对于模板尤其重要。通过设置预处理程序条件以防止头文件体多次出现，可以很好的实现幂等。

```
#ifndef HEADER_H
#define HEADER_H
/* contents of header file */
#endif
```

## 5.2 模板定义

可以用两种方法组织模板定义：使用包括的定义和使用独立的定义。包括的定义组织允许对模板编译进行更多的控制。

### 5.2.1 包括的模板定义

在将模板的声明和定义放在使用该模板的文件中时，组织是**包括定义的**组织。例如：

main.cc

```
template <class Number> Number twice(Number original);
template <class Number> Number twice(Number original )
    { return original + original; }
int main()
    { return twice<int>(-3); }
```

使用模板的文件包括了包含模板声明和模板定义的文件时，使用模板的该文件的组织也是包括定义的组织。例如：

twice.h

```
#ifndef TWICE_H
#define TWICE_H
template <class Number>
Number twice(Number original);
template <class Number> Number twice( Number original )
    { return original + original; }
#endif
```

main.cc

```
#include “twice.h”
int main()
    { return twice(-3); }
```

**注 –** 使模板标题幂等非常重要。（请参见第 68 页中的 “5.1.2 幂等头文件”。）

### 5.2.2 独立的模板定义

另一种组织模板定义的方法是将定义保留在模板定义文件中，如以下示例所示。

twice.h

```
#ifndef TWICE_H
#define TWICE_H
template <class Number>
Number twice(Number original);
#endif TWICE_H
```

twice.cc

```
template <class Number>
Number twice( Number original )
    { return original + original; }
```

main.cc

```
#include “twice.h”
int main( )
    { return twice<int>( -3 ); }
```

模板定义文件**不得**包括任何非幂等头文件，而且通常根本不需要包括任何头文件。（请参见第 68 页中的 “5.1.2 幂等头文件”。）请注意，并非所有编译器都支持模板的独立定义模型。

一个单独的定义文件作为头文件时，该文件可能会被隐式包括在许多文件中。因此，它不应该包含任何函数或变量定义，除非这些定义是模板定义的一部分。一个单独的定义文件可以包含类型定义，包括 typedef。

---

**注 –** 尽管通常会使用模板定义文件的源文件扩展名（即 `.c`、`.C`、`.cc`、`.cpp`、`.cxx` 或 `.c++`），但模板定义文件是头文件。如果需要，编译器会自动包括它们。模板定义文件**不**应单独编译。

---

如果将模板声明放置在一个文件中，而将模板定义放置在另一个文件中，则需仔细考虑如何构造定义文件，如何命名定义文件和如何放置定义文件。此外也需要向编译器显式指定定义的位置。有关模板定义搜索规则的信息，请参阅第 90 页中的 “7.5 模板定义搜索”。

使用 `-E` 或 `-P` 选项生成预处理器输出时，定义分离的文件组织不允许在 `.i` 文件中包含模板定义。编译 `.i` 文件时会因缺少定义而失败。通过有条件地在模板声明标题中包含模板定义文件（请参见下面的代码示例），可确保通过在命令行中使用 `-template=no%extdef` 来使用模板定义。`libCtd` 和 STLport 库以此方式实现。

```
// templace declaration file
template <class T> class foo { ... };
#ifdef _TEMPLATE_NO_EXTDEF
#include "foo.cc"  //template definition file
#endif
```

但是，请勿尝试自行定义宏 `_TEMPLATE_NO_EXTDEF`。如果定义时没有使用 `–template=no%extdef` 选项，可能会因为包含多个模板定义文件而导致编译失败。

◆ ◆ ◆ 第 6 章

# 6 创建和使用模板

通过模板，可以采用类型安全方法来编写适用于多种类型的单一代码。本章介绍了模板的概念和函数模板上下文中的术语，讨论了更复杂的（更强大的）类模板，描述了模板的组成，此外还讨论了模板实例化、缺省模板参数和模板专门化。本章的结尾部分讨论了模板的潜在问题。

## 6.1 函数模板

函数模板描述了仅用参数或返回值的类型来区分的一组相关函数。

### 6.1.1 函数模板声明

使用模板之前，请先声明。以下示例中的**声明**提供了使用模板所需的足够信息，但没有提供实现模板所需的足够信息。

```
template <class Number> Number twice( Number original );
```

在此示例中，*Number* 是**模板参数**，它指定模板描述的函数范围。更具体地说，*Number* 是**模板类型参数**，在模板定义中使用它表示确定的模板使用位置处的类型。

### 6.1.2 函数模板定义

如果要声明模板，请先定义该模板。**定义**提供了实现模板所需的足够信息。以下示例定义了在前一个示例中声明的模板。

```
template <class Number> Number twice( Number original )
    { return original + original; }
```

因为模板定义通常出现在头文件中，所以模板定义必须在多个编译单元中重复。不过所有的定义都必须是相同的。该限制称为**一次定义规则**。

### 6.1.3 函数模板用法

声明后，模板可以像其他函数一样使用。它们的**使用**由命名模板和提供函数参数组成。编译器可以从函数参数类型推断出模板类型参数。例如，您可以按以下方式使用以前声明的模板：

```
double twicedouble( double item )
    { return twice( item ); }
```

如果模板参数不能从函数参数类型推断出，则调用函数时必须提供模板参数。例如：

```
template<class T> T func(); // no function arguments
int k = func<int>(); // template argument supplied explicitly
```

## 6.2 类模板

类模板描述了一组相关的类或数据类型，它们只能通过类型来区分：整数值、指向（或引用）具有全局链接的变量的指针、其他的组合。类模板尤其适用于描述通用但类型安全的数据结构。

### 6.2.1 类模板声明

类模板声明仅提供了类的名称和类的模板参数。此类声明是**不完整的类模板**。

以下示例是名为 `Array` 类的模板声明，该类可接受任何类型作为参数。

```
template <class Elem> class Array;
```

该模板用于名为 `String` 的类，该类接受 `unsigned int` 作为参数。

```
template <unsigned Size> class String;
```

### 6.2.2 类模板定义

类模板定义必须声明类数据和函数成员，如以下示例所示。

```
template <class Elem> class Array {
        Elem* data;
        int size;
    public:
        Array( int sz );
        int GetSize();
        Elem& operator[]( int idx );
};
```

```
template <unsigned Size> class String {
        char data[Size];
        static int overflows;
    public:
        String( char *initial );
        int length();
};
```

与函数模板不同，类模板可以同时有类型参数（如 `class Elem`）和表达式参数（如 `unsigned Size`）。表达式参数可以是：

- 具有整型或枚举的值
- 指向对象的指针或到对象的引用
- 指向函数的指针或到函数的引用
- 指向类成员函数的指针

## 6.2.3 类模板成员定义

类模板的完整定义需要类模板函数成员和静态数据成员的定义。动态（非静态）数据成员由类模板声明完全定义。

### 6.2.3.1 函数成员定义

模板函数成员的定义由模板参数专门化后跟函数定义组成。函数标识符通过类模板的类名称和模板参数限定。以下示例说明了 `Array` 类模板的两个函数成员的定义，该模板中指定了模板参数 `template <class Elem>`。每个函数标识符都通过模板类名称和模板参数 `Array<Elem>` 限定。

```
template <class Elem> Array<Elem>::Array( int sz )
    {size = sz; data = new Elem[size];}

template <class Elem> int Array<Elem>::GetSize()
    { return size; }
```

该示例说明了 `String` 类模板的函数成员的定义。

```
#include <string.h>
template <unsigned Size> int String<Size>::length( )
    {int len = 0;
      while (len < Size && data[len]!= '\0') len++;
      return len;}

template <unsigned Size> String<Size>::String(char *initial)
    {strncpy(data, initial, Size);
      if (length( ) == Size) overflows++;}
```

### 6.2.3.2 静态数据成员定义

模板静态数据成员的定义由后跟变量定义的模板参数专门化组成，在此处变量标识符通过类模板名称和类模板实元参数来限定。

```
template <unsigned Size> int String<Size>::overflows = 0;
```

### 6.2.4 类模板的用法

模板类可以在使用类型的任何地方使用。指定模板类包括了提供模板名称和参数的值。以下示例中的声明根据 Array 模板创建 int_array 变量。变量的类声明及其一组方法类似于 Array 模板中的声明和方法，只是 Elem 替换为了 int。请参见第 74 页中的 “6.3 模板实例化”。

```
Array<int> int_array(100);
```

此示例中的声明使用 String 模板创建 short_string 变量。

```
String<8> short_string("hello");
```

需要任何其他成员函数时，您可以使用模板类成员函数。

```
int x = int_array.GetSize( );

int x = short_string.length( );
.
```

## 6.3 模板实例化

模板**实例化**是生成采用特定模板参数组合的具体类或函数（**实例**）。例如，编译器生成一个采用 Array<int> 的类，另外生成一个采用 Array<double> 的类。通过用模板参数替换模板类定义中的模板参数，可以定义这些新的类。在第 72 页中的 “6.2 类模板”中介绍的 Array<int> 示例中，编译器用 int 替换所有 Elem。

### 6.3.1 隐式模板实例化

使用模板函数或模板类时需要实例。如果这种实例还不存在，则编译器隐式实例化模板参数组合的模板。

### 6.3.2 显式模板实例化

编译器仅为实际使用的那些模板参数组合而隐式实例化模板。该方法不适用于构造提供模板的库。C++ 提供了显式实例化模板的功能，如以下示例所示。

#### 6.3.2.1 模板函数的显式实例化

要显式实例化模板函数，请在 template 关键字后接函数的声明（不是定义），且函数标识符后接模板参数。

```
template float twice<float>(float original);
```

在编译器可以推断出模板参数时，模板参数可以省略。

```
template int twice(int original);
```

### 6.3.2.2 模板类的显式实例化

要显式实例化模板类，请在 template 关键字后接类的声明（不是定义），且在类标识符后接模板参数。

```
template class Array<char>;

template class String<19>;
```

显式实例化类时，所有的类成员也必须实例化。

### 6.3.2.3 模板类函数成员的显式实例化

要显式实例化模板类函数成员，请在 template 关键字后接函数的声明（不是定义），且在由模板类限定的函数标识符后接模板参数。

```
template int Array<char>::GetSize();

template int String<19>::length();
```

### 6.3.2.4 模板类静态数据成员的显式实例

要显式实例化模板类静态数据成员，请在 template 关键字后接成员的声明（不是定义），且在由模板类限定的成员标识符后接模板参数。

```
template int String<19>::overflows;
```

## 6.4 模板组合

可以嵌套使用模板。这种方式尤其适用于在通用数据结构上定义通用函数，与在标准 C++ 库中相同。例如，模板排序函数可以通过一个模板数组类进行声明：

```
template <class Elem> void sort(Array<Elem>);
```

并定义为：

```
template <class Elem> void sort(Array<Elem> store)
    {int num_elems = store.GetSize();
      for (int i = 0; i < num_elems-1; i++)
          for (int j = i+1; j < num_elems; j++)
              if (store[j-1] > store[j])
```

```
                {Elem temp = store[j];
                  store[j] = store[j-1];
                  store[j-1] = temp;}}
```

上述示例定义了针对预先声明的 Array 类模板对象的排序函数。下一个示例说明了排序函数的实际用法。

```
Array<int> int_array(100);   // construct an array of ints
sort(int_array);             // sort it
```

# 6.5 缺省模板参数

您可以将缺省值赋予类模板（但不是函数模板）的模板参数。

```
template <class Elem = int> class Array;
template <unsigned Size = 100> class String;
```

如果模板参数具有缺省值，则该参数后的所有参数也必须具有缺省值。模板参数仅能具有一个缺省值。

# 6.6 模板专门化

将某些模板参数组合视为一个特殊的参数可以提高性能，如本节中的 twice 示例所示。另外，模板说明对于它的一组可能参数也可能不起作用，如本节中的 sort 示例所示。模板专门化允许您定义实际模板参数给定组合的可选实现。模板专门化覆盖了缺省实例化。

## 6.6.1 模板专门化声明

使用模板参数的组合之前，您必须声明专门化。以下示例声明了 twice 和 sort 的专用实现。

```
template <> unsigned twice<unsigned>( unsigned original );

template <> sort<char*>(Array<char*> store);
```

如果编译器可以明确决定模板参数，则您可以省略模板参数。例如：

```
template <> unsigned twice(unsigned original);

template <> sort(Array<char*> store);
```

## 6.6.2 模板专门化定义

必须定义声明的所有模板专门化。下例定义了上一节中声明的函数。

```
template <> unsigned twice<unsigned>(unsigned original)
    {return original << 1;}

#include <string.h>
template <> void sort<char*>(Array<char*> store)
    {int num_elems = store.GetSize();
      for (int i = 0; i < num_elems-1; i++)
          for (int j = i+1; j < num_elems; j++)
              if (strcmp(store[j-1], store[j]) > 0)
                  {char *temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp;}}
```

## 6.6.3 模板专门化使用和实例化

专门化与其他任何模板一样使用并实例化，除此以外，完全专用模板的定义也是实例化。

## 6.6.4 部分专门化

在前一个示例中，模板是完全专用的。也就是说，模板定义了特定模板参数的实现。模板也可以部分专用，这意味着只有某些模板参数被指定，或者一个或多个参数被限定到某种类型。生成的部分专门化仍然是模板。例如，以下代码样例说明了主模板和该模板的完全专门化。

```
template<class T, class U> class A {...}; //primary template
template<> class A<int, double> {...};    //specialization
```

以下代码说明了主模板部分专门化的示例。

```
template<class U> class A<int> {...};          // Example 1
template<class T, class U> class A<T*> {...};  // Example 2
template<class T> class A<T**, char> {...};    // Example 3
```

- 示例 1 提供了用于第一个模板参数是 `int` 类型的情况的特殊模板定义。
- 示例 2 提供了用于第一个模板参数是任何指针类型的情况的特殊模板定义。
- 示例 3 提供了用于第一个模板参数是任何类型的指针到指针而第二个模板参数是 `char` 类型的情况的特殊模板定义。

# 6.7 模板问题部分

本节介绍了使用模板时会遇到的问题。

## 6.7.1 非本地名称解析和实例化

有时模板定义使用模板参数或模板本身未定义的名称。如此，编译器解决了封闭模板作用域的名称，该模板可以在定义或实例化点的上下文中。名称可以在不同的位置具有不同的含义，产生不同的解析。

名称解析比较复杂。因此，您不应该依赖除一般全局环境中提供的名称外的非本地名称。也就是说，仅使用在任何地方都用相同方法声明和定义的非本地名称。在以下示例中，模板函数 `converter` 使用了非本地名称 `intermediary` 和 `temporary`。在 `use1.cc` 和 `use2.cc` 中这些名称的定义不同，因此在不同的编译器下可能会生成不同的结果。为了能可靠地使用模板，所有非本地名称（该示例中为 `intermediary` 和 `temporary`）在任何地方都必须有相同的定义。

```
use_common.h
// Common template definition
template <class Source, class Target>
Target converter(Source source)
       {temporary = (intermediary)source;
       return (Target)temporary;}
use1.cc
typedef int intermediary;
int temporary;

#include "use_common.h"
use2.cc
typedef double intermediary;
unsigned int temporary;

#include "use_common.h"
```

一个常见的非本地名称用法是在模板内使用 `cin` 和 `cout` 流。有时程序员要将流作为模板参数传递，这时就要引用到全局变量。但 `cin` 和 `cout` 在任何地方都必须有相同的定义。

## 6.7.2 作为模板参数的本地类型

模板实例化系统取决于类型名称，等效于决定哪些模板需要实例化或重新实例化。因此本地类型用作模板参数时，会导致严重的问题。小心在代码中也出现类似的问题。

**示例 6–1** 本地类型用作模板参数问题的示例

```
array.h
template <class Type> class Array {
        Type* data;
```

**示例 6–1** 本地类型用作模板参数问题的示例 （续）

```
        int   size;
    public:
        Array(int sz);
        int GetSize();
};

array.cc
template <class Type> Array<Type>::Array(int sz)
    {size = sz; data = new Type[size];}
template <class Type> int Array<Type>::GetSize()
    {return size;}

file1.cc
#include "array.h"
struct Foo {int data;};
Array<Foo> File1Data(10);

file2.cc
#include "array.h"
struct Foo {double data;};
Array<Foo> File2Data(20);
```

在 `file1.cc` 中注册的 `Foo` 类型与在 `file2.cc` 中注册的 `Foo` 类型不同。以这种方法使用本地类型会出现错误和意外的结果。

## 6.7.3 模板函数的友元声明

模板在使用之前必须先声明。模板的使用由友元声明构成，不是由模板的声明构成。实际的模板声明必须在友元声明之前。例如，编译系统尝试链接以下示例中生成的目标文件时，对**未**实例化的 `operator<<` 函数，会生成未定义错误。

**示例 6–2** 友元声明问题的示例

```
array.h
// generates undefined error for the operator<< function
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
        operator<<(std::ostream&, const array<T>&);
};
#endif

array.cc
#include <stdlib.h>
#include <iostream>
```

示例 6–2 友元声明问题的示例 （续）

```
template<class T> array<T>::array() {size = 1024;}

template<class T>
std::ostream&
operator<<(std::ostream& out, const array<T>& rhs)
    {return out <<'[' << rhs.size <<']';}

main.cc
#include <iostream>
#include "array.h"

int main()
{
    std::cout
      << "creating an array of int... " << std::flush;
    array<int> foo;
    std::cout << "done\n";
    std::cout << foo << std::endl;
    return 0;
}
```

请注意，因为编译器将以下行作为普通函数（array 类的 friend）的声明进行读取，所以编译期间不会出现错误消息。

```
friend ostream& operator<<(ostream&, const array<T>&);
```

因为 operator<< 实际上是模板函数，所以需要在声明 template class array 之前提供模板声明。但是，由于 operator<< 有一个 array<T> 类型的参数，因此必须在声明函数之前声明 array<T>。文件 array.h 必须如此示例所示：

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

// the next two lines declare operator<< as a template function
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<< <T> (std::ostream&, const array<T>&);
};
#endif
```

## 6.7.4 在模板定义内使用限定名称

C++ 标准要求使用具有限定名的类型，这些限定名取决于要用 typename 关键字显式标注为类型名称的模板参数。即使编译器可以推断出它应当为某一类型，也需满足该要求。以下示例中的注释说明了具有要用 typename 关键字的限定名的类型。

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // not a type
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;   // not a type
template <class T> struct example {
  static typename T::a_type variable1;             // dependent
  static typename parametric<T>::a_type variable2; // dependent
  static simple::a_type variable3;                 // not dependent
};
template <class T> typename T::a_type             // dependent
  example<T>::variable1 = 0;                      // not a type
template <class T> typename parametric<T>::a_type // dependent
  example<T>::variable2 = 0;                      // not a type
template <class T> simple::a_type   // not dependent
example<T>::variable3 = 0;          // not a type
```

## 6.7.5 嵌套模板名称

由于 ">>" 字符序列解释为右移运算符，因此在一个模板名称中使用另一个模板名称时必须小心。确保相邻的 ">" 字符之间至少有一个空格。

例如，以下是形式错误的语句：

```
Array<String<10>> short_string_array(100); // >> = right-shift
```

被解释为：

```
Array<String<10 >> short_string_array(100);
```

正确的语法为：

```
Array<String<10> > short_string_array(100);
```

## 6.7.6 引用静态变量和静态函数

在模板定义中，编译器不支持引用在全局作用域或名称空间中声明为静态的对象或函数。如果生成了多个实例，由于每个实例引用了不同的对象，因此违背了单次定义规则（C++ 标准的 3.2 节）。通常的失败指示是链接时丢失符号。

如果想要所有模板实例化共享单一对象，那么请使对象成为已命名空间的非静态成员。如果想要模板类的每个实例化不同对象，那么请使对象成为模板类的静态成员。如果希望每个模板函数实例化的对象不同，请使对象成为函数的本地对象。

## 6.7.7 在同一目录中使用模板生成多个程序

如果要通过指定 `-instances=extern` 生成多个程序或库，请在不同的目录中生成这些程序或库。如果要在同一目录中生成多个程序，则在各次生成之间需要清除系统信息库。该做法可以避免出现任何不可预料的错误。有关更多信息，请参见第 90 页中的“7.4.4 共享模板系统信息库”。

考虑以下包含 makefile `a.cc`、 `b.cc`、`x.h` 和 `x.cc`. 的示例请注意，仅当指定了 `-instances=extern` 时，该示例才有意义：

```
........
Makefile
........
CCC = CC

all: a b

a:
    $(CCC) -I. -instances=extern -c a.cc
    $(CCC) -instances=extern -o a a.o

b:
    $(CCC) -I. -instances=extern -c b.cc
    $(CCC) -instances=extern -o b b.o

clean:
    /bin/rm -rf SunWS_cache *.o a b
```

```
...
x.h
...
template <class T> class X {
public:
  int open();
  int create();
  static int variable;
};
```

```
...
x.cc
...
template <class T> int X<T>::create() {
  return variable;
}

template <class T> int X<T>::open() {
  return variable;
}

template <class T> int X<T>::variable = 1;
```

```
...
a.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.open();
  temp1.create();
}


...
b.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.create();
}
```

如果同时生成 a 和 b，请在两个生成之间添加 make clean。以下命令会引起错误：

```
example% make a
example% make b
```

以下命令不会产生任何错误：

```
example% make a
example% make clean
example% make b
```

◆ ◆ ◆ 第 7 章

# 7 编译模板

C++ 编译器在模板编译方面处理的工作要比传统 UNIX 编译器处理的工作多。C++ 编译器必须按需为模板实例生成目标代码。该编译器会使用模板系统信息库在多个独立的编译间共享模板实例，此外还接受某些模板编译选项。编译器必须在各个源文件中定位模板定义，并维护模板实例和主线代码之间的一致性。

## 7.1 详细编译

如果指定了标志 `-verbose=template`，C++ 编译器会在编译模板期间针对重要事件向您发送通知。但是如果使用缺省值 `-verbose=no%template`，则编译器不会发出通知。`+w` 选项可以在进行模板实例化时提供其他有关潜在问题的指示信息。

## 7.2 系统信息库管理

`CCadmin`(1) 命令管理模板系统信息库（只能与选项 `-instances=extern` 一起使用）。例如，程序中的更改会造成某些实例化过度，这样会浪费存储空间。`CCadmin – clean` 命令（以前是 `ptclean`）清除所有实例及关联数据。实例化仅在需要时才重新创建。

### 7.2.1 生成的实例

为了生成模板实例，编译器将内联模板函数看作内联函数。编译器像管理其他内联函数一样管理这些内联模板函数，另外本章中的说明不适用于模板内联函数。

### 7.2.2 整个类实例化

编译器通常是分别实例化各个模板类成员，因此，编译器仅实例化程序中使用的成员。仅用于调试器的方法会因此而不正常地实例化。

可使用两种策略确保调试成员可供调试器使用。

- 首先，编写使用模板类实例成员（否则无用）的非模板函数，不需要调用该函数。
- 其次，使用 `-template=wholeclass` 编译器选项，该选项指示编译器实例化模板类的所有非模板非内联成员（如果实例化这些相同成员中的任何一个）。

ISO C++ 标准允许开发者编写模板类，对于这些类，可能并不是所有成员都可以合法使用某个给定模板参数。只要非法成员未被实例化，程序就仍然完好。ISO C++ 标准库使用了这种技术。但是，`-template=wholeclass` 选项会实例化所有成员，因此不能用于此类使用有问题的模板参数实例化的模板类。

## 7.2.3 编译时实例化

实例化是 C++ 编译器从模板创建可用的函数或对象的过程。C++ 编译器使用了编译时实例化，在编译对模板的引用时强制进行实例化。

编译时实例化的优点是：

- 调试更加简单。错误消息出现在上下文中，使得编译器可以给出到引用点的完整回溯。
- 模板实例化始终保持最新。
- 包括链接阶段在内的总编译时间减少了。

如果源文件位于不同的目录或您使用了具有模板符号的库，则模板可以多次实例化。

## 7.2.4 模板实例的放置和链接

缺省情况下，实例会进入特殊地址区域，链接程序会识别并丢弃重复项。您可以指示编译器使用五个实例放置和链接方法之一：外部、静态、全局、显式和半显式。

- 在下列情况下，外部实例可以达到最佳的执行效果：
  - 程序中的实例集比较小，但是每个编译单元引用了实例较大的子集。
  - 少数实例是在多于一个或两个编译单元中引用的。

  静态实例已过时。
- 缺省的全局实例适用于所有开发，并且在对象引用各种实例时可以达到最佳的执行效果。
- 显式实例适用于某些需精确控制的应用程序编译环境。
- 半显式实例对编译环境的控制要求较少，但是生成的目标文件较大，并且使用有限制。

本节讨论了五种实例放置和链接方法。第 74 页中的 “6.3 模板实例化”中提供了有关生成实例的其他信息。

## 7.3 外部实例

对于外部实例方法，所有实例都放置在模板系统信息库中。编译器确保只有一个一致的模板实例存在；这些实例既不是未定义的也不是多重定义的。模板仅在需要时才重新实例化。对于非调试代码，所有目标文件（包括模板缓存中的任何目标文件）在使用 `-instances=extern` 时的大小总量可能会小于在使用 `-instances=global` 时的大小总量。

模板实例接受系统信息库中的全局链接。实例是使用外部链接从当前编译单元引用的。

---

**注 –** 如果在不同的步骤中进行编译和链接，并且在编译步骤中指定了 `-instance=extern`，则还必须在链接步骤中指定该选项。

---

这种方法的缺点是每当您更改程序或进行重大程序更改时都必须清除高速缓存。高速缓存是并行编译的瓶颈，这与使用 `dmake` 时一样，因为每次只能有一个编译访问高速缓存。另外，您只能在每个目录内生成一个程序。

与在主目标文件中直接创建有效模板实例并在以后丢弃该实例（如果需要）相比，确定某个有效实例是否已在高速缓存中可能需要更长的时间。

使用 `-instances=extern` 选项指定外部链接。

因为实例存储在模板系统信息库中，所以必须使用 `CC` 命令将使用外部实例的 C++ 对象链接到程序中。

如果要创建包含了使用的所有模板实例的库，请使用 `-xar` 选项进行编译。而**不**要使用 `ar` 命令。例如：

```
example% CC -xar -instances=extern -o libmain.a a.o b.o c.o
```

### 7.3.1 可能的高速缓存冲突

如果指定了 `-instance=extern`，请勿在同一目录中运行不同的编译器版本，以避免可能的高速缓存冲突。使用 `-instances=extern` 模板模型进行编译时，请注意以下事项：

- 请勿在同一目录中创建不相关的二进制文件。在同一目录中创建的所有二进制文件（`.o`、`.a`、`.so`、可执行程序）都应该相关，因为两个或两个以上目标文件通用的所有对象、函数和类型的名称都具有相同的定义。
- 在同一目录中同时运行多个编译是安全的，例如使用 `dmake` 时。与另外一个链接步骤同时运行任何编译或链接步骤是不安全的。**链接步骤**是指用于创建库或可执行程序的任何操作。确保 makefile 中的依赖性不允许任何命令与链接步骤以并行方式运行。

### 7.3.2 静态实例

**注 –** `-instances=static` 选项已过时，因为 `-instances=global` 现在提供了 `static` 的所有优点，并克服了其缺点。早期的编译器中提供了此选项来克服现已不存在的问题。

对于静态实例方法，所有实例都被放置在当前编译单元内。因此，模板在每个重新编译期间重新实例化；这些实例不保存到模板系统信息库。

这种方法的缺点是不遵循语言语义，并且会生成很大的对象和可执行文件。

实例接收静态链接。这些实例在当前编译单元外部是不可视的或不可用的。因此，模板可以在多个目标文件中具有相同的实例化。因为多重实例产生了不必要的大程序，所以对于不可能多重实例化模板的小程序可以使用静态实例链接。

静态实例的编译速度很快，因此这种方法也适用于修复并继续方式的调试。（请参见《使用 `dbx` 调试程序》。）

**注 –** 如果您的程序取决于多个编译单元间的共享模板实例（例如模板类或模板函数的静态数据成员），请勿使用静态实例方法。否则程序会工作不正常。

可使用 `-instances=static` 编译器选项指定静态实例链接。

### 7.3.3 全局实例

与早期的编译器发行版不同，现在不必预防出现一个全局实例有多个副本的情况。

这种方法的优点是通常由其他编译器接受的不正确源代码也能在这种模式中接受。需要特别指出的是，从模板实例内对静态变量的引用是不合法的，但通常是可以接受的。

这种方法的缺点是单个目标文件可能会较大，因为可能会在多个文件中存在模板实例的多个副本。如果编译目标文件以便进行调试时，有些使用了 `-g` 选项，而有些没有使用该选项，则很难预测是获得链接到程序中模板实例的调试版本还是非调试版本。

模板实例接收全局链接。这些实例在当前编译单元外部是可视的和可用的。

可使用 `-instances=global` 选项（缺省值）指定全局实例。

### 7.3.4 显式实例

在显式实例方法中，仅为显式实例化的模板生成实例。隐式实例化不能满足该要求。实例被放置在当前编译单元内。

这种方法的优点是拥有最少的模板编译和最小的对象大小。

缺点是您必须手动执行所有的实例化。

模板实例接收全局链接。这些实例在当前编译单元外部是可视的和可用的。链接程序识别并丢弃重复项目。

可使用 -instances=explicit 选项指定显式实例。

### 7.3.5 半显式实例

使用半显式实例方法时，仅为显式实例化或模板体内隐式实例化的模板生成实例。显式创建的实例所必需的实例将自动生成。主线代码中隐式实例化不满足该要求。实例被放置在当前编译单元内。因此，模板在每个重新编译期间重新实例化；生成的实例接收全局链接，且不会被保存到模板系统信息库中。

可使用 -instances=semiexplicit 选项指定半显式实例。

## 7.4 模板系统信息库

在各次编译之间，模板系统信息库存储模板实例，以便仅在需要时编译模板实例。模板系统信息库包含了使用外部实例方法时模板实例化所需的所有非源文件。系统信息库不用于其他种类的实例。

### 7.4.1 系统信息库结构

缺省情况下，模板系统信息库位于名为 SunWS_cache 的高速缓存目录中。

高速缓存目录包含在放置目标文件的目录中。可以通过设置环境变量 SUNWS_CACHE_NAME 更改高速缓存目录的名称。请注意，SUNWS_CACHE_NAME 变量值必须是目录名称，而不能是路径名。编译器会自动将模板高速缓存目录放置到目标文件目录下，因此编译器已具有路径。

### 7.4.2 写入模板系统信息库

编译器必须存储模板实例时，编译器将模板实例存储在对应于输出文件的模板系统信息库中。例如，以下命令将目标文件写入 ./sub/a.o 并将模板实例写入包含在 ./sub/SunWS_cache 中的系统信息库。如果高速缓存目录不存在，且编译器需要对模板进行实例化，则编译器将创建该目录。

```
example% CC -o sub/a.o a.cc
```

### 7.4.3 从多模板系统信息库读取

编译器从对应于编译器读取的目标文件的模板系统信息库读取。例如，以下命令从 `./sub1/SunWS_cache` 和 `./sub2/SunWS_cache` 进行读取，并且在必要时向 `./SunWS_cache` 进行写入。

```
example% CC sub1/a.o sub2/b.o
```

### 7.4.4 共享模板系统信息库

系统信息库中的模板不得违反 ISO C++ 标准的一次定义规则。也就是说，使用所有的模板时模板必须具有相同的源。违反该规则会产生不可预料的行为。

确保不违反该规则的最简单和最保守的方法是在任何一个目录内仅生成一个程序或库。两个不相关的程序可以使用相同类型的名称或外部名称来表示不同的内容。如果程序共享模板系统信息库，则模板定义会出现冲突，会产生不可预料的结果。

### 7.4.5 通过 -instances=extern 实现模板实例自动一致

如果指定了 `-instances=extern`，模板系统信息库管理器可确保系统信息库中实例的状态与源文件一致且是最新的。

例如，如果您的源文件是使用 `-g` 选项（启用调试）编译的，则还会使用 `-g` 编译来自数据库中的所需文件。

此外，模板系统信息库会跟踪编译中的更改。例如，如果设置了 `-DDEBUG` 标志来定义名称 `DEBUG`，则数据库中会记录该信息。如果在以后的编译中省略该标志，则编译器重新实例化设置依赖性的这些模板。

---

**注 –** 如果删除模板的源代码或停止使用模板，模板的实例会保留在缓存中。如果更改函数模板的签名，使用旧签名的实例会保留在缓存中。如果因为这些问题在编译时或链接时遇到了异常行为，请清除模板缓存并重新生成程序。

---

## 7.5 模板定义搜索

使用独立定义模板组织时，模板定义在当前编译单元不可用，编译器必须搜索该定义。本节介绍了编译器如何找到定义。

定义搜索有些复杂，并且很容易出现错误。因此如果可能，您应该使用定义包括模板文件组织。这样有助于避免一起定义搜索。请参见第 69 页中的 “5.2.1 包括的模板定义”。

注 – 如果使用 -template=no%extdef 选项，编译器将不搜索单独的源文件。

## 7.5.1 源文件位置约定

如果没有随选项文件一起提供的特定方向，则编译器使用 Cfront 样式的方法来定位模板定义文件。此方法要求模板定义文件包含的基名与模板声明文件包含的基名相同。此方法还要求模板定义文件位于当前 include 路径中。例如，如果模板函数 foo() 位于 foo.h 中，匹配的模板定义文件应该命名为 foo.cc 或某些其他可识别的源文件扩展名（.C、.c、.cc、.cpp、.cxx 或 .c++）。模板定义文件必须位于常规的 include 目录之一中，或位于与其匹配的头文件所在目录中。

## 7.5.2 定义搜索路径

可以用另外一种方法替代用 –I 设置的常规搜索路径，即使用选项 –pti*directory* 指定模板定义文件的搜索目录。多个 -pti 标志定义多个搜索目录，即搜索路径。如果使用 -pti*directory*，则编译器在该路径查找模板定义文件并忽略 –I 标志。由于 -pti*directory* 标志会使源文件的搜索规则变得复杂，因此应使用 -I 选项而不是 -pti*directory* 选项。

## 7.5.3 诊断有问题的搜索

有时，编译器会生成令人费解的警告或错误消息，因为它会查找您不打算编译的文件。此问题通常是由于已有某个文件（如 foo.h）包含了模板声明，但又隐式包含了另一个文件（如 foo.cc）。

如果头文件 foo.h 有模板声明，缺省情况下，编译器会搜索名为 foo 且具有 C++ 文件扩展名（.C、.c、.cc、.cpp、.cxx 或 .c++）的文件。如果找到这样的文件，编译器将自动把它包含进来。有关这些搜索的更多信息，请参见第 90 页中的 “7.5 模板定义搜索”。

如果有一个不打算这样处理的文件 foo.cc，有两种解决方法：

- 更改 .h 或 .cc 文件的名称，以消除名称匹配。
- 可以通过指定 -template=no%extdef 选项来禁用对模板定义文件的自动搜索。然后必须在代码中显式包含所有模板定义，并且不能使用“独立定义”模型。

◆ ◆ ◆ 第 8 章

# 8 异常处理

本章讨论了 C++ 编译器的异常处理实现。第 102 页中的 “10.2 在多线程程序中使用异常”中提供了附加信息。有关异常处理的更多信息，请参见由 Bjarne Stroustrup 编著的《The C++ Programming Language》第三版（Addison-Wesley 出版，1997 年）。

## 8.1 同步和异步异常

异常处理旨在仅支持同步异常，例如数组范围检查。**同步异常**这一术语意味着异常只能源于 `throw` 表达式。

C++ 标准支持具有终止模型的同步异常处理。**终止**意味着一旦抛出异常，控制永远不会返回到抛出点。

异常处理不能用于直接处理诸如键盘中断等异步异常。不过，如果小心处理，在出现异步事件时也可以进行异常处理。例如，要用信号进行异常处理工作，您可以编写设置全局变量的信号处理程序，并创建另外一个例程来定期轮询该变量的值，当该变量值发生更改时抛出异常。不能从信号处理程序抛出异常。

## 8.2 指定运行时错误

五个运行时错误消息与异常相关：

- 没有异常处理程序
- 抛出了意外的异常
- 异常只能在处理程序中重新抛出
- 在堆栈展开时，析构函数必须处理自身的异常
- 内存不足

运行时检测到错误时，错误消息会显示当前异常的类型和这五个错误消息之一。缺省情况下，会调用预定义的函数 `terminate()`，该函数又会调用 `abort()`。

编译器使用异常规范中提供的信息来优化代码生成。例如，禁止不抛出异常的函数表条目，而函数异常规范的运行时检查在任何可能的地方被消除。

# 8.3 禁用异常

如果知道程序中未使用异常，可以使用编译器选项 `features=no%except` 抑制支持异常处理的代码的生成。该选项的使用可以稍微减小代码的大小，并能加快代码的执行速度。不过，用禁用的异常编译的文件链接到使用异常的文件时，在用禁用的异常编译的文件中的某些局部对象在发生异常时不会销毁。缺省情况下，编译器生成支持异常处理的代码。除非时间和空间开销是要考虑的重要因素，否则将异常保留为启用状态通常更好。

---

**注 –** 因为 C++ 标准库 `dynamic_cast` 和缺省运算符 `new` 要求使用异常，所以在标准模式（缺省模式）下编译时不应禁用异常。

---

# 8.4 使用运行时函数和预定义的异常

标准头文件 `<exception>` 提供了 C++ 标准中指定的类和异常相关函数。仅在标准模式（编译器缺省模式，或使用选项 `-compat=5`）下编译时才可访问该头文件。以下摘录的部分代码显示了 `<exception>` 头文件声明。

```
// standard header <exception>
namespace std {
    class exception {
           exception() throw();
           exception(const exception&) throw();
           exception& operator=(const exception&) throw();
           virtual ~exception() throw();
           virtual const char* what() const throw();
    };
    class bad_exception: public exception {...};
    // Unexpected exception handling
       typedef void (*unexpected_handler)();
       unexpected_handler
         set_unexpected(unexpected_handler) throw();
       void unexpected();
    // Termination handling
       typedef void (*terminate_handler)();
       terminate_handler set_terminate(terminate_handler) throw();
       void terminate();
       bool uncaught_exception() throw();
}
```

标准类 `exception` 是所选语言构造或 C++ 标准库抛出的所有异常的基类。可以构造、复制及销毁类型为 `exception` 的对象，而不会生成异常。虚拟成员函数 `what()` 返回描述异常的字符串。

为了与 C++ 4.2 版中所用的异常兼容，还提供了头文件 <exception.h> 以用于标准模式下。该头文件允许转换到标准 C++ 代码，并包含了不是标准 C++ 部分的声明。应在开发进度计划许可的情况下，更新代码以遵循 C++ 标准（使用 <exception> 而非 <exception.h>）。

```
// header <exception.h>, used for transition
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

## 8.5 将异常与信号和 Setjmp/Longjmp 混合使用

可以在会出现异常的程序中使用 setjmp/longjmp 函数，只要它们不会互相影响。

使用异常和 setjmp/longjmp 的所有规则分别适用。此外，仅当在点 A 抛出与在点 B 捕获的异常具有相同的效果时，从点 A 到点 B 的 longjmp 才有效。需特别指出的是，不得使用 longjmp 进入或跳出 try 块或 catch 块（直接或间接），或使用 longjmp 跳过自动变量或临时变量的初始化或 non-trivial 销毁。

不能从信号处理程序抛出异常。

## 8.6 生成具有异常的共享库

对于包含 C++ 代码的程序，请勿使用 -Bsymbolic 。请改用链接程序映射文件或链接程序作用域选项。请参见第 57 页中的 “4.1 链接程序作用域”。如果使用 -Bsymbolic，不同模块中的引用可能会绑定到应是一个全局对象的不同副本。

异常机制依赖对地址的比较。如果您具有某项内容的两个副本，它们的地址就不等同且异常机制可能失败，这是由于异常机制依赖对假设为唯一地址内容的比较。

◆ ◆ ◆ 第 9 章

# 9 改善程序性能

采用编译器易于编译优化的方式编写函数，可以改善 C++ 函数的性能。很多书中都对软件性能做了一般性介绍并具体介绍了 C++，本章不再重复这类重要信息，而只讨论那些显著影响 C++ 编译器的性能策略。

## 9.1 避免临时对象

C++ 函数经常会产生必须创建并销毁的隐式临时对象。对于重要的类，临时对象的创建和销毁会占用很多处理时间和内存。C++ 编译器消除了某些临时类，但是并不能消除所有的临时类。

编写函数时，在确保您的程序易于理解的同时请尽量减少临时对象的数目。这些技术包括：使用显式变量而不使用隐式临时对象，以及使用引用变量而不使用值参数。另外一种技术是实现和使用诸如 += 这样的操作，而非实现和使用只包含 + 和 = 的操作。例如，下面的第一行引入了用于保存 a + b 结果的临时对象，而第二行则不是。

```
T x = a + b;
T x(a); x += b;
```

## 9.2 使用内联函数

使用扩展内联而不使用正常调用时，对小而快速的函数的调用可以更小更快速。反过来，如果使用扩展内联而不建立分支，则对又长又慢的函数的调用会更大更慢。另外，只要函数定义更改，就必须重新编译对内联函数的所有调用。因此，使用内联函数时要格外小心。

如果预计函数定义会更改**而且**重新编译所有调用程序非常耗时，请不要使用内联函数。而如果扩展函数内联的代码比调用函数的代码少，**或**使用函数内联时应用程序执行速度显著提高，则可以使用内联函数。

编译器不能内联所有函数调用，因此要充分利用函数内联，可能需要进行一些源码更改。可使用 +w 选项了解何时不会进行函数内联。在以下情况中，编译器将**不会**内联函数：

- 函数包含了复杂控制构造，例如循环、switch 语句和 try/catch 语句。这些函数很少多次执行复杂控制构造。要内联这种函数，请将函数分割为两部分：里边的部分包含复杂的控制构造，而外边的部分决定何时调用里边的部分。即使编译器可以内联完整函数，从函数常用部分中分隔出不常用部分的这种技术也可以改善性能。
- 内联函数体又大又复杂。因为对函数体内其他内联函数的调用，或因为隐式构造函数和析构函数调用（通常发生在派生类的构造函数和析构函数中），所以简单函数体可以非常复杂。对于这种函数，内联扩展很少提供显著的性能改善，所以函数一般不内联。
- 内联函数调用的参数既大又复杂。对于内联成员函数调用的对象是内联函数调用的自身这种情况，编译器特别敏感。要内联具有复杂参数的函数，只需将函数参数计算到局部变量并将变量传递到函数。

# 9.3 使用缺省运算符

如果类定义不声明无参数的构造函数、复制构造函数、复制赋值运算符或析构函数，那么编译器将隐式声明它们。它们都是调用的缺省运算符。类似 C 的结构具有这些缺省运算符。编译器生成缺省运算符时，可以了解大量关于需要处理的工作和可以产生优良代码的工作。这种代码通常比用户编写的代码的执行速度快，原因是编译器可以利用汇编级功能的优点，而程序员则不能利用该功能的优点。因此缺省运算符执行所需的工作时，程序不能声明这些运算符的用户定义版本。

缺省运算符是内联函数，因此内联函数不合适时不使用缺省运算符（请参见上一节）。否则，缺省运算符适用于以下情况：

- 用户编写的无参数构造函数仅为构造函数的基对象和成员变量调用无参数构造函数。有效的基元类型具有“不进行任何操作”的无参数构造函数。
- 用户编写的复制构造函数仅复制所有的基对象和成员变量。
- 用户编写的复制赋值运算符仅复制所有的基对象和成员变量。
- 用户编写的析构函数可以为空。

某些 C++ 编程手册建议编写类的程序员始终定义所有的运算符，以便该代码的任何读者都能了解该程序员没有忘记考虑缺省运算符的语义。显然，该建议与以上讨论的优化有冲突。这种冲突的解决方案是在代码中放置注释以表明类正使用缺省运算符。

# 9.4 使用值类

包括结构和联合在内的 C++ 类通过值来传递和返回。对于 Plain-Old-Data (POD) 类，C++ 编译器需要像 C 编译器一样传递结构。这些类的对象**直接**进行传递。对于包含用户定义复制构造函数的类的对象，要求编译器有效地构造对象的副本，将指针传递到副本，并在返回后销毁副本。这些类的对象**间接**进行传递。编译器也可以选择介于这两个需求之间的类。不过，该选择影响二进制的兼容性，因此编译器对每个类的选择必须保持一致。

对于大多数编译器，直接传递对象可以加快执行速度。这种执行速度的改善对于小值类（例如复数和概率值）来说尤其明显。有时为了改善程序执行效率，您可以设计更可能直接传递而不是间接传递的类。

如果类具有以下任何一个特征，则它将间接传递：

- 用户定义的复制构造函数
- 用户定义的析构函数
- 间接传递的基
- 间接传递的非静态数据成员

否则，类被直接传递。

## 9.4.1 选择直接传递类

尽可能直接传递类：

- 只要可能，就使用缺省构造函数，尤其是缺省复制构造函数。
- 尽可能使用缺省析构函数。由于缺省析构函数不是虚拟的，因此具有缺省析构函数的类通常不是基类。
- 避免使用虚函数和虚拟基。

## 9.4.2 在不同的处理器上直接传递类

C++ 编译器直接传递的类和联合与 C 编译器传递结构或联合完全相同。不过，C++ 结构和联合在不同的体系结构上进行不同的传递。

**表 9–1** 在不同体系结构上结构和联合的传递

| 体系结构 | 说明 |
|---|---|
| SPARC V7/V8 | 通过在调用程序内分配存储并将指针传递到该存储，传递并返回结构和联合。（也就是说，所有的结构和联合都通过引用传递。） |

表 9–1 在不同体系结构上结构和联合的传递 （续）

| 体系结构 | 说明 |
| --- | --- |
| SPARC V9 | 不超过 16 个字节（32 个字节）的结构在寄存器中传递。通过在调用程序内分配存储并将指针传递到该存储，传递并返回联合和所有其他结构。（也就是说，小的结构在寄存器中传递，而联合和大的结构通过引用传递。）因此，小值类与基元类具有相同的传递效率。 |
| x86 平台 | 结构和联合通过在堆栈上分配空间并将参数复制到堆栈上来传递。通过在调用程序的帧中分配临时对象并将临时对象的地址作为隐式的第一个参数传递，返回结构和联合。 |

# 9.5 缓存成员变量

访问成员变量是 C++ 成员函数的通用操作。

编译器必须经常通过 `this` 指针从内存装入成员变量。因为值通过指针装入，所以编译器有时不能决定何时执行第二次装入或以前装入的值是否仍然有效。在这些情况下，编译器必须选择安全但缓慢的方法，在每次访问成员变量时重新装入成员变量。

如下所示，可以通过在局部变量中显式缓存成员变量的值来避免不必要的内存重新装入：

- 声明局部变量并使用成员变量的值初始化该变量。
- 在函数中成员变量的位置使用局部变量。
- 如果局部变量变化，那么将局部变量的最终值赋值到成员变量。不过，如果成员函数在该对象上调用另一个成员函数，那么该优化会产生不可预料的结果。

当值位于寄存器中时，这种优化最有效，而这种情况也与基元类型相同。基于内存的值的优化也会很有效，因为减少的别名使编译器获得了更多的机会来进行优化。

如果成员变量经常通过引用（显式或隐式）来传递，那么优化可能并没什么效果。

有时，类的目标语义需要成员变量的显式缓存，例如在当前对象和其中一个成员函数参数之间有潜在别名时。例如：

```
complex& operator*= (complex& left, complex& right)
{
  left.real = left.real * right.real + left.imag * right.imag;
  left.imag = left.real * right.imag + left.image * right.real;
}
```

会产生不可预料的结果，前提是调用时使用：

```
x*=x;
```

◆ ◆ ◆ 第 10 章

# 10 生成多线程程序

本章解释了如何生成多线程程序。此外，还讨论了异常的使用，解释了如何在线程之间共享 C++ 标准库对象，此外还描述了如何在多线程环境中使用传统（旧的）iostream。

有关多线程的更多信息，请参见《多线程编程指南》。

另请参见《OpenMP API 用户指南》，了解有关使用 OpenMP 共享内存并行化指令来创建多线程程序的信息。

## 10.1 生成多线程程序

C++ 编译器附带的所有库都是多线程安全的。如果需要生成多线程应用程序，或者需要将应用程序链接到多线程库，必须使用 –mt 选项编译和链接程序。此选项会将 –D_REENTRANT 传递给预处理程序，并按正确的顺序将 –l thread 链接到 ld。缺省情况下，-mt 选项可确保 libthread 在 libCrun 之前链接。推荐使用 –mt，这是指定宏和库的替代方式，它更加简单且不易出错。

### 10.1.1 表明多线程编译

可以通过使用 ldd 命令检查应用程序是否链接到 libthread：

```
example% CC -mt myprog.cc
example% ldd a.out
libm.so.1 =>      /usr/lib/libm.so.1
libCrun.so.1 =>   /usr/lib/libCrun.so.1
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

### 10.1.2 与线程和信号一起使用 C++ 支持库

C++ 支持库 libCrun、libiostream 和 libCstd 是多线程安全的，但不是 async 安全的。因此，在多线程应用程序中，支持库中可用的函数不能用于信号处理程序中。这样做的话将导致死锁状态。

在多线程应用程序的信号处理程序中使用以下功能是不安全的：

- iostream
- new 和 delete 表达式
- 异常

## 10.2 在多线程程序中使用异常

对于多线程而言，当前的异常处理实现是安全的，因为一个线程中的异常与其他线程中的异常互不影响。不过，您不能使用异常来进行线程之间的通信，因为一个线程中抛出的异常不会被其他线程捕获到。

每个线程都可设置其自己的 terminate() 或 unexpected() 函数。在一个线程中调用 set_terminate() 或 set_unexpected() 只影响该线程中的异常。对于任何线程，terminate() 的缺省函数是 abort()（请参见第 93 页中的 “8.2 指定运行时错误”）。

### 10.2.1 线程取消

通过调用 pthread_cancel(3T) 取消线程会导致销毁堆栈上的自动（局部非静态）对象，但指定了 -noex 或 -features=no%except 时例外。

pthread_cancel(3T) 使用的机制与异常相同。取消线程时，局部析构函数与用户通过 pthread_cleanup_push() 注册的清除例程交叉执行。在特定的清除例程注册之后，函数调用的本地对象在例程执行前就被销毁了。

## 10.3 在线程之间共享 C++ 标准库对象

C++ 标准库 (libCstd -library=Cstd ) 是 MT 安全的，但具有某些语言环境异常。它确保该库的内部结构在多线程环境中正常运行。但是，您仍需要将您自己在各个线程之间共享的任何库对象锁定起来。请参见 setlocale(3C) 和 attributes(5) 手册页。

例如，如果实例化字符串，然后创建新的线程并使用引用将字符串传递给线程，因为要在线程之间显示共享这个字符串对象，所以您必须锁定对于该字符串的写访问。（库提供的用于完成该任务的工具在下文中会有描述。）

另一方面，如果将字符串按值传递给新线程，即使两个不同的线程中的字符串应用 Rogue Wave 的“copy on write（写时复制）”技术共享表示，也不必担心锁定。库将自动处理锁定。只有当要使对象显式可用于多线程或在线程之间传递引用，以及使用全局或静态对象时，您才需要锁定。

C++ 标准库在内部使用锁定（同步）机制来确保在多线程下的行为正确，该机制可以描述如下：

_RWSTDMutex 和 _RWSTDGuard 这两个同步类提供了实现多线程安全的机制。

_RWSTDMutex 类通过下列成员函数提供了与平台无关的锁定机制：

- void acquire()－自己获取锁定，或者在获得此类锁定之前一致处于阻塞状态。
- void release()－自己释放锁定。

```
class _RWSTDMutex
{
public:
    _RWSTDMutex ();
    ~_RWSTDMutex ();
    void acquire ();
    void release ();
};
```

_RWSTDGuard 类是封装有 _RWSTDMutex 类的对象的公用包装器类。_RWSTDGuard 对象尝试在其构造函数中获取封装的互斥锁（抛出从 std::exception on error 派生的 ::thread_error 类型的异常），并在析构函数中释放互斥锁（析构函数从来不会抛出异常）。

```
class _RWSTDGuard
{
public:
    _RWSTDGuard (_RWSTDMutex&);
    ~_RWSTDGuard ();
};
```

另外，可以使用宏 _RWSTD_MT_GUARD(mutex)（以前的 _STDGUARD）有条件地在多线程生成中创建 _RWSTDGuard 的对象。该对象保护代码块的其余部分，并在该代码块中定义为可同时被多个线程执行。在单线程生成中，宏扩展到空表达式中。

以下示例说明了这些机制的使用。

```
#include <rw/stdmutex.h>

//
// An integer shared among multiple threads.
//
int I;

//
// A mutex used to synchronize updates to I.
//
```

```
_RWSTDMutex I_mutex;

//
// Increment I by one. Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire(); // Lock the mutex.
   I++;
   I_mutex.release(); // Unlock the mutex.
}

//
// Decrement I by one. Uses an _RWSTDGuard.
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex); // Acquire the lock on I_mutex.
   --I;
   //
   // The lock on I is released when destructor is called on guard.
   //
}
```

# 10.4 内存边界内部函数

编译器提供头文件 mbarrier.h，此头文件中针对 SPARC 和 x86 处理器定义了多种内存边界内部函数。这些内部函数可在开发者使用自己的同步基元编写多线程代码时使用。请参阅相应的处理器文档，以确定在特定情况下是否需要以及何时需要使用这些内部函数。

mbarrier.h 头文件支持以下内存排序内部函数：

- __machine_r_barrier()－这是**读取**边界。它确保边界前的所有装入操作能够在边界后的所有装入操作之前完成。
- __machine_w_barrier()－这是**写入**边界。它确保边界前的所有存储操作能够在边界后的所有存储操作之前完成。
- __machine_rw_barrier()－这是**读写**边界。它确保边界前的所有装入和存储操作能够在边界后的所有装入和存储操作之前完成。
- __machine_acq_barrier()－这是具有**获取**语义的边界。它确保边界前的所有装入操作能够在边界后的所有装入和存储操作之前完成。
- __machine_rel_barrier()－这是具有**释放**语义的边界。它确保边界前的所有装入和存储操作能够在边界后的所有存储操作之前完成。
- __compiler_barrier()－阻止编译器跨边界移动内存访问。

内部函数 __compiler_barrier() 出现异常的所有边界内部函数生成内存排序指令，在 x86 上这些指令是 mfence、sfence 或 lfence 指令，而在 SPARC 平台上这些指令是 membar 指令。

__compiler_barrier() 内部函数不会生成指令，但会通知编译器所有以前的内存操作必须在启动任何后续内存操作之前完成。所产生的实际结果为：具有 static 存储类说明符的所有非局部变量和局部变量都将在边界之前存储回到内存，然后在边界之后重新装入，编译器不会将边界之前的内存操作与边界之后的内存操作弄混。所有其他的边界均隐含 __compiler_barrier() 内部函数的行为。

例如，在以下代码中 __compiler_barrier() 内部函数的存在将阻止编译器合并两种循环：

```
#include "mbarrier.h"
int thread_start[16];
void start_work()
{
/* Start all threads */
   for (int i=0; i<8; i++)
   {
     thread_start[i]=1;
   }
   __compiler_barrier();
/* Wait for all threads to complete */
   for (int i=0; i<8; i++)
   {
      while (thread_start[i]==1){}
   }
}
```

第 3 部分

# 库

◆ ◆ ◆ 第 11 章

# 11 使用库

库提供了在多个应用程序间共享代码并减小超大型应用程序复杂度的方法。C++ 编译器使您可以访问各种库。本章说明了如何使用这些库。

## 11.1 C 库

Oracle Solaris 操作系统附带了安装在 `/usr/lib` 中的几个库。这些库大多有 C 接口。缺省情况下，其中的 `libc` 和 `libm` 库通过 `CC` 驱动程序进行链接。如果使用 `-mt` 选项，则链接 `libthread` 库。如果要链接其他系统库，请在链接时使用适当的 `-l` 选项。例如，要链接 `libdemangle` 库，请在链接时在 `CC` 命令行上传递 `–ldemangle`：

```
example% CC text.c -ldemangle
```

C++ 编译器具有自己的运行时支持库。所有 C++ 应用程序都由 `CC` 驱动程序链接到这些库。C++ 编译器还具有其他一些有用的库，如下节所述。

## 11.2 随 C++ 编译器提供的库

C++ 编译器附带了一些库。

下表列出了随 C++ 编译器提供的库，以及可以使用这些库的模式。

表 11–1 C++ 编译器附带的库

| 库 | 说明 |
|---|---|
| `libstlport` | 标准库的 STLport 实现。 |
| `libstlport_dbg` | 调试模式的 STLport 库 |
| `libCrun` | C++ 运行时 |

表 11–1 C++ 编译器附带的库 （续）

| 库 | 说明 |
|---|---|
| libCstd | C++ 标准库 |
| libiostream | 传统 iostream |
| libcsunimath | 支持 -xia 选项 |
| librwtool | Tools.h++ 7 |
| librwtool_dbg | 支持调试的 Tools.h++ 7 |
| libgc | 垃圾收集 |
| libdemangle | 取消改编 |
| sunperf | Sun 性能库 |

**注 –** 请勿重新定义或修改用于 STLport、Rogue Wave 或 Oracle Solaris Studio C++ 库的任何配置宏。库是按照适用于 C++ 编译器的方式配置和生成的。libCstd 和 Tools.h++ 配置为可互操作，因此，修改配置宏会导致程序不能编译、不能链接或不能正常运行。

### 11.2.1 C++ 库描述

本节提供了每个 C++ 库的简要描述。

- libCrun－包含缺省标准模式 (-compat=5) 下编译器所需的运行时支持。并提供了对 new/delete、异常及 RTTI 的支持。

  libCstd－C++ 标准库。需要特别指出的是，该库包含了 iostream。如果有使用传统 iostream 的现有源代码，而且要使用标准 iostream，必须修改源代码以符合新接口。有关详细信息，请参见《C++ 标准库参考》联机手册。

- libiostream－使用 -compat=5 生成的传统 iostream 库。如果有使用传统 iostream 的现有源代码，且要在标准模式 (–compat=5) 下编译这些源代码，可以使用 libiostream 而不必修改源代码。可使用 – library=iostream 获取此库。

  **注 –** 标准库的很大部分取决于使用的标准 iostream。在相同程序中使用传统的 iostream 可能会出现问题。

- libstlport－C++ 标准库的 STLport 实现。可以通过指定选项 -library=stlport4，使用该库而非缺省的 libCstd。但不能在同一程序中同时使用 libstlport 和 libCstd。您必须使用其中一个库编译和链接包括导入库在内的一切项目。

- `librwtool` (`Tools.h++`)－来自 RogueWave 的 C++ 基础类库。提供了版本 7。该库已过时，不应在新代码中使用该库。提供它是为了支持针对使用 RW Tools.h++ 的 C++ 4.2 编写的程序。
- `libgc`－在部署模式或垃圾收集模式下使用。只是与 `libgc` 库链接就会自动且永久修复程序的内存泄漏。虽然能以其他方式正常编程，但如果将程序与 `libgc` 库链接，则无需调用 `free` 或 `delete` 就可完成编程。垃圾收集库对动态装入库具有依赖性，因此在链接程序时要指定 `-lgc` 和 `-ldl`。

  有关其他信息，请参见 `gcFixPrematureFrees`(3) 和 `gcInitialize`(3) 手册页。
- `libdemangle`－用于取消改编的 C++ 名称。

### 11.2.2 访问 C++ 库的手册页

与本节所述库关联的手册页位于第 1、3、3C++ 和 3cc4 节中。

要访问 C++ 库的手册页，请输入：

```
example% man library-name
```

要访问 C++ 库版本 4.2 的手册页，请输入：

```
example% man -s 3CC4 library-name
```

### 11.2.3 缺省 C++ 库

在生成可执行程序时会缺省链接 C++ 库，但生成共享库 (`.so`) 时不会链接。在生成共享库时，必须显式列出所有必需的库。如果在生成可执行程序时省略了某个所需库，并且该库是缺省库，则 `-zdefs` 选项将会使链接程序发出警报。缺省情况下，`CC` 驱动程序链接以下库：

```
-lCstd -lCrun -lm -lc
```

有关更多信息，请参见第 188 页中的 “A.2.49 `-library=`*l*[ ,*l*...]”。

## 11.3 相关的库选项

`CC` 驱动程序提供了一些选项来帮助用户使用库。

- `-l` 选项用于指定要链接的库。
- `-L` 选项用于指定要在其中搜索库的目录。
- `-mt` 选项用于编译和链接多线程代码。
- `-xia` 选项用于链接区间运算库。

- `-xlang` 选项用于链接 Fortran 或 C99 运行时库。
- `-library` 选项用于指定 Oracle Solaris Studio C++ 编译器附带的以下库：

```
libCrun
libCstd
libiostream
libC
libcomplex
libstlport, libstlport_dbg
librwtool, librwtool_dbg
libgc
sunperf
```

**注 –** 要使用传统 iostream 形式的 `librwtool`，请使用 `-library=rwtools7` 选项。要使用标准 iostream 形式的 `librwtool`，请使用 `-library=rwtools7_std` 选项。

使用 `–library` 和 `–staticlib` 选项指定的库将静态链接。下面是一些示例：

`libstdcxx`（作为 Oracle Solaris OS 的一部分分发）

以下命令动态链接传统 iostream 形式的 `Tools.h++` 版本 7 和 `libiostream` 库。

```
example% CC test.cc -library=rwtools7,iostream
```

以下命令静态链接 `libgc` 库。

```
example% CC test.cc -library=gc -staticlib=gc
```

以下命令排除了库 `libCrun` 和 `libCstd`，否则缺省情况下这两个库包括在内。

```
example% CC test.cc -library=no%Crun,no%Cstd
```

缺省情况下，`CC` 根据命令行选项链接不同的系统库集合。如果指定 `-xnolib`（或 `-nolib`），`CC` 仅链接在命令行上使用 `-l` 选项显式指定的那些库。（如果使用 `-xnolib` 或 `-nolib`，会忽略 `-library` 选项（如果有）。）

使用 `–R` 选项可以在可执行文件中生成动态库搜索路径。执行期间，运行时链接程序使用这些路径搜索应用程序所需的共享库。缺省情况下，`CC` 驱动程序将 `–R`*<install-directory>*`/lib` 传递给 `ld`（如果编译器安装在标准位置中）。可以使用 `-norunpath` 禁止在可执行文件中生成共享库的缺省路径。

缺省情况下，链接程序会搜索 `/lib` 和 `/usr/lib`。请勿在 `-L` 选项中指定这些目录或任何编译器安装目录。

对于针对部署生成的程序，应该使用 `-norunpath` 或 `-R` 选项进行生成，这样可避免在编译器目录中查找库。请参见第 115 页中的 “11.6 使用共享库”。

# 11.4 使用类库

通常，使用类库分两个步骤：

1. 在源码中包括适当的头文件。
2. 将程序与目标库链接。

## 11.4.1 iostream 库

C++ 编译器提供两种 iostream 实现：

- **传统 iostream**。该术语指的是 C++ 4.0、4.0.1、4.1 和 4.2 编译器所附带的 iostream 库，以及基于 `cfront` 的 3.0.1 编译器附带的更早的 iostream 库。没有用于此库的标准。它在 `libiostream` 中可用。
- **标准 iostream**。该库是 C++ 标准库 `libCstd` 的一部分，仅在标准模式下可用。该库与传统 iostream 库的二进制和源码都不兼容。

如果已有 C++ 源，那么代码可能象以下示例一样使用传统 iostream。

```
// file prog1.cc
#include <iostream.h>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

以下示例使用了标准 iostream。

```
// file prog2.cc
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

以下命令编译 `prog2.cc` 并将其链接到名为 `prog2` 的可执行程序中。该程序在标准模式下编译。缺省情况下，链接包括标准 iostream 库的 `libCstd`。

```
example% CC prog2.cc -o prog2
```

### 11.4.1.1 有关传统 iostream 和传统 RogueWave 工具的说明

所谓的“传统”iostream 是指 iostream 的初始 1986 版本，已被 C++ 标准替代。它可通过 `-library=rwtools7,iostream` 选项选择。没有两个“传统”iostream 的实现是相同的，因此除了已过时之外，使用它的代码还不可移植。请注意，该库和选项在以后的 Oracle Solaris Studio 发行版中将停止使用。

随传统 Sun Studio 和 Oracle Studio 提供的 RW Tools.h++ 工具集最初发布于 1990 年，且至今未有显著的更新。它的 `time` 和 `date` 类具有无法修复的关于夏时制时间的严重问题。（该工具集的功能当前在 C++ 标准和 BOOST 之类的开源库中可用。）RW Tools.h++ 可通过 `-library=rwtools7` 或 `-library=rwtools7_std` 选项选择，并且在以后的 Oracle Solaris Studio 发行版中将停止使用。

## 11.4.2 链接 C++ 库

下表显示了用于链接 C++ 库的编译器选项。有关更多信息，请参见第 188 页中的 “A.2.49 -library=*l*[ ,*l*...]”。

**表 11–2** 链接 C++ 库的编译器选项

| 库 | 选项 |
|---|---|
| 传统 `iostream` | `-library=iostream` |
| `Tools.h++` 版本 7 | `-library=rwtools7,iostream`<br>`-library=rwtools7_std` |
| `Tools.h++` 版本 7 调试 | `-library=rwtools7_dbg,iostream`<br>`-library=rwtools7_std_dbg` |
| 垃圾收集 | `-library=gc` |
| STLport 版本 4 | `-library=stlport4` |
| STLport 版本 4 调试 | `-library=stlport4_dbg` |
| Apache `stdcxx` 版本 4 | `-library=stdcxx4` |
| Sun 性能库 | `-library=sunperf` |

# 11.5 静态链接标准库

`CC` 驱动程序缺省链接在多个库（包括 `libc` 和 `libm`）的共享版本中，这是通过为每个缺省库将 `-l`*lib* 选项传递给链接程序来实现的。（有关缺省库的列表，请参见第 111 页中的 “11.2.3 缺省 C++ 库”。）

如果要静态链接其中任何缺省库，可以使用 `-library` 选项和 `–staticlib` 选项。例如：

```
example% CC test.c -staticlib=Crun
```

在此示例中，没有在命令中显式包括 `-library` 选项。这种情况下，无需 `-library` 选项，因为在标准模式（缺省模式）下，`-library` 的缺省设置是 `Cstd,Crun`。

也可以使用 -xnolib 编译器选项。使用 -xnolib 选项时，驱动程序不会将任何 -l 选项传递给 ld，所以您必须自己传递这些选项。以下示例显示了如何静态链接 libCrun 以及如何动态链接 libm 和 libc：

```
example% CC test.c -xnolib -lCstd -Bstatic -lCrun -Bdynamic -lm -lc
```

-l 选项的顺序很重要。– lCstd、– lCrun 和 -lm 选项位于 -lc 之前。

---

**注 –** 建议不要静态链接 libCrun 和 libCstd。而是生成 /usr/lib 中的动态版本以与其所安装在的 Oracle Solaris 版本一起使用。

---

有些 CC 选项链接到其他库。也可以使用 -xnolib 抑制这些库链接。例如，使用 -mt 选项会导致 CC 驱动程序将 -lthread 传递给 ld。但如果同时使用 –mt 和 –xnolib，CC 驱动程序不会将 -lthread 传递给 ld。有关更多信息，请参见第 247 页中的 “A.2.147 –xnolib”。有关 ld 的更多信息，请参见《链接程序和库指南》。

---

**注 –** /lib 和 /usr/lib 中静态版本的 Oracle Solaris 库不再可用。例如，试图静态链接 libc 的操作将失败：

```
CC hello.cc -xnolib -lCrun -lCstd -Bstatic -lc
```

---

# 11.6 使用共享库

C++ 编译器附带下列 C++ 运行时共享库：

- libCCexcept.so.1（仅限 SPARC Solaris）
- libcomplex.so.5（仅限 Solaris）
- librwtool.so.2
- libstlport.so.1

在 Linux 上，C++ 编译器附带这些附加库：

- libCrun.so.1
- libCstd.so.1
- libdemangle.so
- libiostream.so.1

在最新的 Oracle Solaris 发行版中，这些附加库以及其他一些库作为 Oracle Solaris C++ 运行时库软件包 SUNWlibC 的一部分安装。

如果应用程序使用 C++ 编译器附带的任何共享库，则 CC 驱动程序会安排**运行路径**（请参阅 -R 选项），该运行路径指向将在可执行文件中生成库的位置。如果之后将可执行文件部署到另一台计算机上，而该计算机上并没有在同一位置安装同一编译器版本，将找不到所需的共享库。

在程序启动时，可能根本找不到此库，或可能使用错误版本的库，从而导致错误的程序行为。在这种情况下，应该将所需库与可执行文件一起提供，并使用指向这些库将要安装到的位置的**运行路径**进行生成。

文章“Using and Redistributing Solaris Studio Libraries in an Application”（在应用程序中使用和重新分配 Solaris Studio 库）包含了本主题的完整说明及示例。它位于
`http://www.oracle.com/technetwork/articles/servers-storage-dev/redistrib-libs-344133.html`

# 11.7 替换 C++ 标准库

替换与编译器一起发布的标准库是有风险的，不能保证产生预期的结果。基本操作是禁用编译器提供的标准头文件和库，指定找到新的头文件和库（及库本身的名称）的目录。

编译器支持标准库的 STLport 和 Apache `stdcxx` 实现。有关更多信息，请参见第 123 页中的 “12.2 STLport”和第 124 页中的 “12.3 Apache stdcxx 标准库”。

## 11.7.1 可以替换的内容

可以替换大多数标准库及其关联头文件。替换的库是 `libCstd`，关联的头文件如下所示：

```
<algorithm> <bitset> <complex> <deque> <fstream <functional> <iomanip> <ios>
<iosfwd> <iostream> <istream> <iterator> <limits> <list> <locale> <map> <memory>
<numeric> <ostream> <queue> <set> <sstream> <stack> <stdexcept> <streambuf>
<string> <strstream> <utility> <valarray> <vector>
```

库的可替换部分由一般称为 "STL" 的内容和字符串类、iostream 类及其帮助类组成。因为这些类和头文件是相互依赖的，所以不能仅替换其中的一部分。如果要替换任何一部分，就应该替换所有头文件和所有 `libCstd`。

## 11.7.2 不可替换的内容

标准头文件 `<exception>`、`<new>` 和 `<typeinfo>` 与编译器本身以及 `libCrun` 紧密相关，不能可靠替换。库 `libCrun` 包含了编译器依赖且不能替换的许多“帮助”函数。

从 C 继承的 17 个标准头文件（`<stdlib.h>`、`<stdio.h>`、`<string.h>` 等）与 Oracle Solaris 操作系统和 Solaris 基本运行时库 `libc` 紧密相关，不能可靠替换。这些头文件的 C++ 版本（`<cstdlib>`、`<cstdio>`、`<cstring>` 等）与基本 C 版本紧密相关，不能可靠替换。

### 11.7.3 安装替换库

要安装替换库，必须先确定替换头文件的位置和 libCstd 的替换库。为方便讨论，假定头文件放置在 /opt/mycstd/include 中，库放置在 /opt/mycstd/lib 中。假定库称为 libmyCstd.a。（通常，库名称以 "lib" 开头。）

### 11.7.4 使用替换库

每次编译时，都使用 -I 选项指向头文件的安装位置。此外，还使用 -library=no%Cstd 选项防止查找编译器自身版本的 libCstd 头文件。例如：

```
example% CC -I/opt/mycstd/include -library=no%Cstd... (compile)
```

编译期间，-library=no%Cstd 选项防止搜索编译器自身版本的这些头文件所在的目录。

每次执行程序或库链接时，都使用 -library=no%Cstd 选项防止查找编译器自身的 libCstd，使用 -L 选项指向替换库所在的目录，以及使用 -l 选项指定替换库。例如：

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd... (link)
```

也可以直接使用库的全路径名，而不使用 -L 和 -l 选项。例如：

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a... (link)
```

链接期间，-library=no%Cstd 选项防止链接编译器自身版本的 libCstd。

### 11.7.5 标准头文件实现

C 有 17 个标准头文件（<stdio.h>、<string.h>、<stdlib.h> 等）。这些头文件作为 Oracle Solaris 操作系统的一部分提供，位于 /usr/include 目录中。C++ 也有这些头文件，但另外要求在全局名称空间和 std 名称空间中都有各种声明的名称。

C++ 也有另一个版本的各个 C 标准头文件（<cstdio>、<cstring> 和 <cstdlib> 等），仅名称空间 std 中有各种声明的名称。最后，C++ 添加了 32 个自己的标准头文件（<string>、<utility>、<iostream> 等）。

标准头文件的明显实现将 C++ 源码中找到的名称用作包括的文本文件的名称。例如，标准头文件 <string>（或 <string.h>）可能指某目录中名为 string（或 string.h）的文件。这种明显实现有以下缺点：

- 如果头文件没有文件名后缀，则无法仅搜索头文件或为头文件创建 makefile 规则。
- 如果具有名为 string 的目录或可执行程序，可能会错误地找到该目录或程序而不是标准头文件。

为了解决这些问题，编译器 include 目录会包含一个与头文件同名的文件和一个指向它且具有唯一后缀 .SUNWCCh （SUNW 是所有编译器相关软件包的前缀，CC 指 C++ 编译器，h 是常用的头文件后缀）的符号链接。指定 <string> 后，编译器将其重写为 <string.SUNWCCh> 并搜索该名称。后缀名只能在编译器自己的 include 目录中找到。如果这样找到的文件是符号链接（正常情况下），编译器就对链接进行一次引用解除，并将结果（此例中是 string）用作错误消息和调试器引用的文件名。忽略文件的依赖性信息时，编译器使用带后缀的名称。

仅当出现在尖括号中且无需指定任何路径时，17 种标准 C 头文件和 32 种标准 C++ 头文件的两种格式才会发生名称重写。如果使用引号来代替尖括号指定任何路径组件或其他某些头文件，就不会有重写发生。

下表说明了通常的情况。

**表 11–3** 头文件搜索示例

| 源代码 | 编译器搜索 | 注释 |
|---|---|---|
| <string> | string.SUNWCCh | C++ 字符串模板 |
| <cstring> | cstring.SUNWCCh | C string.h 的 C++ 版本 |
| <string.h> | string.h.SUNWCCh | C string.h |
| <fcntl.h> | fcntl.h | 不是标准 C 或 C++ 头文件 |
| "string" | string | 双引号，不是尖括号 |
| <../string> | ../string | 指定的路径 |

如果编译器未找到 *header*.SUNWCCh，则编译器将重新搜索 #include 指令中提供的名称。例如，如果使用指令 #include <string>，编译器就尝试查找名为 string.SUNWCCh 的文件。如果搜索失败，编译器就查找名为 string 的文件。

### 11.7.5.1 替换标准 C++ 头文件

由于第 117 页中的 “11.7.5 标准头文件实现”中介绍的搜索算法，您无需提供第 117 页中的 “11.7.3 安装替换库”中介绍的 SUNWCCh 版本的替换头文件。但是，如果遇到了所描述的某些问题，建议为每个无后缀的头文件添加后缀为 .SUNWCCh 的符号链接。也就是说，对于文件 utility，可以运行以下命令：

```
example% ln -s utility utility.SUNWCCh
```

编译器第一次查找 utility.SUNWCCh 时，会找到它，而不会和其他名为 utility 的文件或目录混淆。

### 11.7.5.2 替换标准 C 头文件

不支持替换标准 C 头文件。如果仍然希望提供标准头文件的自己的版本，那么建议按以下步骤操作：

- 将所有替换头文件放置在一个目录中。
- 在该目录中创建指向每个替换头文件的 `.SUNWCCh` 符号链接。
- 在每次调用编译器时使用 `-I` 指令，搜索包含替换头文件的目录。

例如，假设有 `<stdio.h>` 和 `<cstdio>` 的替换。请将文件 `stdio.h` 和 `cstdio` 放在目录 `/myproject/myhdr` 中。在该目录中，运行以下命令：

```
example% ln -s stdio.h stdio.h.SUNWCCh
example% ln -s cstdio cstdio.SUNWCCh
```

每次编译时使用 `-I/myproject/mydir` 选项。

#### 忠告：

- 如果要替换任何 C 头文件，就必须成对替换。例如，如果替换 `<time.h>`，还应该替换 `<ctime>`。
- 替换头文件必须与被替换版本具有相同的效果。也就是说，各种运行时库（如 `libCrun`、`libC`、`libCstd`、`libc` 和 `librwtool`）是使用标准头文件中的定义生成的。如果替换文件不匹配，那么程序不能工作。

◆ ◆ ◆ 第 12 章

# 12 使用 C++ 标准库

当在缺省（标准）模式下编译时，编译器可以访问 C++ 标准指定的整个库。库组件包括一般称为标准模板库 (Standard Template Library, STL) 的库和下列组件：

- 字符串类
- 数字类
- 标准流 I/O 类
- 基本内存分配
- 异常类
- 运行时类型信息

术语 STL 没有正式的定义，但是通常理解为包括容器、迭代器以及算法。以下标准库头的子集可以认为包含了 STL：

- `<algorithm>`
- `<deque>`
- `<iterator>`
- `<list>`
- `<map>`
- `<memory>`
- `<queue>`
- `<set>`
- `<stack>`
- `<utility>`
- `<vector>`

C++ 标准库 (`libCstd`) 基于 RogueWave Standard C++ Library, Version 2。该库是缺省库。

C++ 编译器还支持 STLport 的标准库实现版本 4.5.3。`libCstd` 仍是缺省库，STLport 的产品只是备选的。有关更多信息，请参见第 123 页中的 “12.2 STLport”。

如果需要使用自己的 C++ 标准库版本而非编译器附带的某一版本，可以通过指定 `-library=no%Cstd` 选项来实现。替换与编译器一起发布的标准库是有风险的，不能保证产生预期的结果。有关更多信息，请参见第 116 页中的 “11.7 替换 C++ 标准库”。

# 12.1 C++ 标准库头文件

表 12–1 列出了完整标准库的头文件以及每个头文件的简要介绍。

**表 12–1** C++ 标准库头文件

| 头文件 | 说明 |
|---|---|
| <algorithm> | 操作容器的标准算法 |
| <bitset> | 位的固定大小序列 |
| <complex> | 数字类型表示复数 |
| <deque> | 支持在端点增加和删除的序列 |
| <exception> | 预定义异常类 |
| <fstream> | 文件的流 I/O |
| <functional> | 函数对象 |
| <iomanip> | iostream 操纵符 |
| <ios> | iostream 基类 |
| <iosfwd> | iostream 类的前向声明 |
| <iostream> | 基本流 I/O 功能 |
| <istream> | 输入 I/O 流 |
| <iterator> | 遍历序列的类 |
| <limits> | 数字类型的属性 |
| <list> | 排序的序列 |
| <locale> | 国际化支持 |
| <map> | 带有键/值对的关联容器 |
| <memory> | 专用内存分配器 |
| <new> | 基本内存分配和释放 |
| <numeric> | 通用的数字操作 |
| <ostream> | 输出 I/O 流 |
| <queue> | 支持在头部增加和在尾部删除的序列 |
| <set> | 有唯一键值的关联容器 |
| <sstream> | 将内存中的字符串用为源或接收器的流 I/O |

表 12–1 C++ 标准库头文件 （续）

| 头文件 | 说明 |
|---|---|
| <stack> | 支持在头部增加和删除的序列 |
| <stdexcept> | 附加的标准异常类 |
| <streambuf> | iostream 的缓冲区类 |
| <string> | 字符序列 |
| <typeinfo> | 运行时类型标识 |
| <utility> | 比较运算符 |
| <valarray> | 用于数字编程的值数组 |
| <vector> | 支持随机访问的序列 |

# 12.2 STLport

如果要使用替换 libCstd 的标准库，请使用标准库的 STLport 实现。可以使用以下编译器选项关闭 libCstd 并改用 STLport 库：

- -library=stlport4

有关更多信息，请参见第 188 页中的 “A.2.49 -library=*l*[ ,*l*...]”。

本发行版包括称为 libstlport.a 的静态归档文件和称为 libstlport.so 的动态库。

决定是否使用 STLport 实现之前，请先考虑以下信息：

- STLport 是开放源代码产品，并不能保证不同发行版本之间的兼容性。换而言之，使用 STLport 的将来版本进行编译可能会破坏使用 STLport 4.5.3 编译的应用程序。它还可能无法将使用 STLport 4.5.3 编译的二进制文件与使用 STLport 的将来版本编译的二进制文件链接在一起。
- stlport4、Cstd 和 iostream 库都提供了自己的 I/O 流实现。如果使用 -library 选项指定其中多个库，会导致出现不确定的程序行为。
- 编译器的后续发行版本可能不包括 STLport4，只包括更新版本的 STLport。在将来发行版中可能不能使用编译器选项 -library=stlport4，但可能会用引用更高 STLport 版本的选项替换该选项。
- Tools.h++ 不支持 STLport。
- STLport 与缺省的 libCstd 是二进制不兼容的。如果使用标准库的 STLport 实现，则必须使用选项 -library=stlport4 编译和链接包括第三方库在内的所有文件。这意味着存在一些限制，例如，不可同时使用 STLport 实现和 C++ 区间数学库 libCsunimath。这是因为对 libCsunimath 编译时使用的是缺省的库头文件而不是 STLport。

- 如果决定使用 STLport 实现，那么一定要包括代码隐式引用的头文件。允许标准头文件，但不必要包括另一个标准头文件作为实现的一部分。

### 12.2.1 重新分发和支持的 STLport 库

请参见分发自述文件，了解可依照“最终用户目标代码许可协议”的条款随您的可执行文件或库重新分发的库和目标文件列表。此自述文件的 C++ 部分列出该编译器发行版支持的 `STLport.so` 版本。此自述文件位于此发行版的 Oracle Solaris Studio 软件的法律页面上，网址为 `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`。

因为以下测试示例中的代码将库实现假定为不可移植，所以在该测试示例中不能使用 STLport 编译。具体来说，它假定 `<vector>` 或 `<iostream>` 自动包含 `<iterator>`，这是无效假定。

```
#include <vector>
#include <iostream>

using namespace std;

int main ()
{
    vector <int> v1 (10);
    vector <int> v3 (v1.size());
    for (int i = 0; i < v1.size (); i++)
      {v1[i] = i; v3[i] = i;}
    vector <int> v2(v1.size ());
    copy_backward (v1.begin (), v1.end (), v2.end ());
    ostream_iterator<int> iter (cout, " ");
    copy (v2.begin (), v2.end (), iter);
    cout << endl;
    return 0;
}
```

要解决该问题，请将 `<iterator>` 包含在源代码中。

## 12.3 Apache stdcxx 标准库

通过使用 `-library=stdcxx4` 进行编译，在 Oracle Solaris 中使用 Apache `stdcxx` 版本 4 C++ 标准库代替缺省的 `libCstd`。此选项还会隐式设置 `-mt` 选项。`stdcxx` 库需要使用多线程模式。必须在整个应用程序的每个编译和链接命令中一致使用此选项。用 `-library=stdcxx4` 编译的代码与用缺省的 `-library=Cstd` 或可选的 `-library=stlport4` 编译的代码不能用在同一程序中。

使用 Apache `stdcxx` 库时，请牢记以下事项：

- `stdcxx` 和 `iostream` 库都提供了自己的 I/O 流实现。如果使用 `-library` 选项指定其中多个库，会导致出现不确定的程序行为。

- `stdcxx` 不支持 Tools.h++。
- `stdcxx` 不支持 C++ 区间数学库 (`libCsunimath`)。
- `stdcxx` 库与缺省的 `libCstd` 和 STLport 是二进制不兼容的。如果使用标准库的 `stdcxx` 实现，则必须使用选项 `-library=stdcxx4` 编译和链接包括第三方库在内的所有文件。

◆ ◆ ◆ 第 13 章

# 13 使用传统 iostream 库

与 C 类似，C++ 没有内建输入或输出语句。相反，I/O 工具是由库提供的。C++ 编译器提供了 iostream 类的传统实现和 ISO 标准实现。

- 缺省情况下，传统 iostream 类包含在 libiostream 中。如果源代码使用传统 iostream 类，且要在标准模式下编译源代码，请使用 libiostream。如果要在标准模式下使用传统 iostream 功能，请将 iostream.h 头文件包括进来并使用 -library=iostream 选项进行编译。
- 标准 iostream 类只能用于标准模式下，且包含在 C++ 标准库 libCstd 中。

本章介绍了传统 iostream 库并提供了其使用示例，但并未完整介绍 iostream 库。有关更多详细信息，请参见 iostream 库手册页。要访问传统 iostream 手册页，请键入命令：**man -s 3CC4***name*

请参见第 113 页中的 “11.4.1.1 有关传统 iostream 和传统 RogueWave 工具的说明”

## 13.1 预定义的 iostream

有四个预定义的 iostream：

- cin，连接到标准输入
- cout，连接到标准输出
- cerr，连接到标准错误
- clog，连接到标准错误

除了 cerr 之外，所有预定义的 iostream 都是完全缓冲的。请参见第 129 页中的 “13.3.1 使用 iostream 进行输出”和第 131 页中的 “13.3.2 使用 iostream 进行输入”。

# 13.2 iostream 交互的基本结构

通过将 iostream 库包括进来，程序可以使用许多输入流或输出流。每个流都具有某些源或接收器，如下所示：

- 标准输入
- 标准输出
- 标准错误
- 文件
- 字符数组

流可以被限定到输入或输出，或同时具有输入和输出。iostream 库使用两个处理层来实现这些流。

- 较低层实现了序列，即字符的简单流。这些序列由 streambuf 类或从其派生的类实现。
- 较高层对序列执行格式化操作。这些格式化操作由 istream 和 ostream 类实现，这两个类将从 streambuf 类派生的类型的对象作为成员。附加类 iostream 用于执行输入和输出的流。

标准输入、输出和错误由从类 istream 或 ostream 派生的特殊类对象处理。

分别从 istream、ostream 和 iostream 派生的 ifstream、ofstream 和 fstream 类用于处理文件的输入和输出。

分别从 istream、ostream 和 iostream 派生的 istrstream、ostrstream 和 strstream 类用于处理字符数组的输入和输出。

打开输入或输出流时，要创建其中一种类型的对象，并将流的 streambuf 成员与设备或文件关联。通常通过流构造函数执行此关联，因此不用直接使用 streambuf。iostream 库为标准输入、标准输出和错误输出预定义了流对象，因此不必为这些流创建自己的对象。

可以使用运算符或 iostream 成员函数将数据插入流（输出）或从流（输入）提取数据，以及控制插入或提取的数据的格式。

如果要插入和提取新的数据类型（其中一个类），通常需要重载插入和提取运算符。

# 13.3 使用传统 iostream 库

要使用来自传统 iostream 库的例程，必须针对所需的库部分将头文件包括进来。下表对头文件进行了具体描述。

表 13–1 iostream 例程头文件

| 头文件 | 说明 |
|---|---|
| iostream.h | 声明 iostream 库的基本功能。 |
| fstream.h | 声明文件专用的 iostream 和 streambuf。包括了 iostream.h。 |
| strstream.h | 声明字符数组专用的 iostream 和 streambuf。包括了 iostream.h。 |
| iomanip.h | 声明操纵符：在 iostream 中插入或提取的值有不同的作用。包括了 iostream.h。 |
| stdiostream.h | （已过时）声明 stdio 文件专用的 iostream 和 streambuf。包括了 iostream.h。 |
| stream.h | （已过时）包括了 iostream.h、fstream.h、iomanip.h 和 stdiostream.h。用于兼容 C++ 1.2 版的旧式流。 |

通常程序中不需要所有这些头文件，而仅包括所需声明的头文件。缺省情况下，libiostream 包含传统的 iostream 库。

## 13.3.1 使用 iostream 进行输出

使用 iostream 进行的输出通常依赖于重载的左移运算符 (<<)（在 iostream 上下文中，称为插入运算符）。要将值输出到标准输出，应将值插入预定义的输出流 cout 中。例如，要将给定值 someValue 发送到标准输出，可以使用以下语句：

```
cout << someValue;
```

对于所有内置类型，都会重载插入运算符，且 someValue 表示的值转换为其适当的输出表示形式。例如，如果 someValue 是 float 值，<< 运算符会将该值转换为带小数点的适当数字序列。此处是将 float 值插入输出流中，因此 << 称为浮点插入器。通常，如果给定类型 X，<< 称为 X 插入器。ios(3CC4) 手册页中讨论了输出格式以及如何控制输出格式。

iostream 库不支持用户定义类型。如果您定义要以您自己的方式输出的类型，则必须定义插入器（即，重载 << 运算符）来正确处理它们。

<< 可以多次应用。要将两个值插入 cout 中，可以使用类似于以下示例中的语句：

```
cout << someValue << anotherValue;
```

以上示例的输出在两个值间不显示空格。因此您可能会要按以下方式编写编码：

```
cout << someValue << " " << anotherValue;
```

运算符 << 优先作为左移运算符（其内置含义）使用。与其他运算符一样，总是可以使用圆括号来指定操作顺序。必要时可使用括号来避免出现优先级问题。下列四个语句中，前两个是等价的，但后两个不是。

```
cout << a+b;              // + has higher precedence than <<
cout << (a+b);
cout << (a&y);            // << has precedence higher than &
cout << a&y;            // probably an error: (cout << a) & y
```

### 13.3.1.1 定义自己的插入运算符

以下示例定义了 string 类：

```
#include <stdlib.h>
#include <iostream.h>


class string {
private:
    char* data;
    size_t size;

public:
    // (functions not relevant here)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

在此示例中，必须将插入运算符和提取运算符定义为友元，因为 string 类的数据部分是 private。

```
ostream& operator<< (ostream& ostr, const string& output)
{    return ostr << output.data;}
```

以下示例显示了为用于 string 而重载的 operator<< 的定义。

```
cout << string1 << string2;
```

运算符 << 将 ostream&（也就是对 ostream 的引用）作为其第一个参数，并返回相同的 ostream，这样就可以在一个语句中合并多个插入。

### 13.3.1.2 处理输出错误

通常情况下，重载 operator<< 时不必检查错误，因为有 iostream 库传播错误。

出现错误时，所在的 iostream 会进入错误状态。iostream 状态中的各个位根据错误的一般类别进行设置。iostream 中定义的插入器不会尝试将数据插入处于错误状态的任何流，因此这种尝试不会更改 iostream 的状态。

通常，处理错误的推荐方法是定期检查某些中心位置中输出流的状态。如果存在错误，则应该以某种方式进行处理。本章假定您定义了函数 error，该函数接受字符串并中止程序。error 不是一个预定义函数。有关 error 函数的示例，请参见第 134 页中的 “13.3.9 处理输入错误”。您可以使用 运算符 ! 来检查 iostream 的状态，如果 iostream 处于错误状态，则会返回非零值。例如：

```
if (!cout) error("output error");
```

还有另外一种方法来测试错误。ios 类定义了 operator void *()，因此当发生错误时，它将返回 NULL 指针。您可以使用如下例所示的语句：

```
if (cout << x) return; // return if successful
```

也可以使用函数 good，它是 ios 的成员：

```
if (cout.good()) return; // return if successful
```

错误位在 enum 中声明：

```
enum io_state {goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80};
```

有关错误函数的详细信息，请参见 iostream 手册页。

#### 13.3.1.3 刷新

与大多数 I/O 库一样，iostream 通常会累积输出并将其发送到较大且效率通常较高的块中。如果要刷新缓冲区，请插入特殊值 flush。例如：

```
cout << "This needs to get out immediately." << flush;
```

flush 是一种称为**操纵符**的对象示例，它是一个值，可以插入 iostream 中以起到一定作用，而不是使输出其值。这些值实际上是函数，它们接受 ostream& 或 istream& 参数，在对其执行某些操作后返回其参数（请参见第 138 页中的“13.7 操纵符”）。

#### 13.3.1.4 二进制输出

要获得原始二进制形式的值输出，请使用以下示例所示的成员函数 write。该示例显示了原始二进制形式的 x 输出。

```
cout.write((char*)&x, sizeof(x));
```

以上示例将 &x 转换为 char*，这违反了类型规程。一般情况下，这样做无关大碍，但如果 x 的类型是具有指针、虚拟成员函数的类或是需要 nontrivial 构造函数操作的类，就无法正确读回以上示例写入的值。

### 13.3.2 使用 iostream 进行输入

使用 iostream 进行的输入类似于输出。需要使用提取运算符 >>，可以像插入操作那样将提取操作串接在一起。例如：

```
cin >> a >> b;
```

该语句从标准输入获得两个值。与其他重载的运算符一样，所使用的提取器取决于 a 和 b 的类型。如果 a 和 b 具有不同的类型，则会使用两个不同的提取器。ios(3CC4) 手册页中详细讨论了输入格式以及如何控制输入格式。通常，前导空白字符（空格、换行符、标签、换页等）被忽略。

## 13.3.3 定义自己的提取运算符

要输入新的类型时，如同重载输出的插入运算符，请重载输入的提取运算符。

类 string 定义了其提取运算符，如以下代码示例所示：

**示例 13–1** string 提取运算符

```
istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
    istr.get(holder, maxline, "\n');
    input = holder;
    return istr;
}
```

get 函数从输入流 istr 读取字符，并将其存储在 holder 中，直到读取了 maxline-1 字符、遇到新行或 EOF（无论先发生哪一项）。然后，holder 中的数据以空终止。最后，holder 中的字符复制到目标字符串。

按照约定，提取器转换其第一个参数中的字符（此示例中是 istream& istr），将其存储在第二个参数（始终是引用），然后返回第一个参数。因为提取器会将输入值存储在第二个参数中，所以第二个参数必须是引用。

## 13.3.4 使用 char* 提取器

使用该预定义提取器时，请务必小心，它可能会导致问题。请按如下方式使用该提取器：

```
char x[50];
cin >> x;
```

该提取器跳过前导空白，提取字符并将其复制到 x 中，直至遇到另一个空白字符。然后，它使用终止空 (0) 字符完成字符串。请谨慎使用该提取器，因为输入可能会溢出给定的数组。

您还必须确保指针指向了分配的存储。以下示例显示了一个常见错误：

```
char * p; // not initialized
cin >> p;
```

由于将存储输入数据的位置不确定，因此您的程序可能会中止。

### 13.3.5 读取任何单一字符

除了使用 char 提取器外，还可以使用任一形式的 get 成员函数获取一个字符。例如：

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

**注 –** 与其他提取器不同，char 提取器不会跳过前导空白。

以下示例显示了一种方法，该方法只跳过空格并在制表符、换行符或任何其他字符处停止：

```
int a;
do {
    a = cin.get();
   }
while(a ==' ');
```

### 13.3.6 二进制输入

如果需要读取二进制值（如使用成员函数 write 写入的值），可以使用 read 成员函数。以下示例说明了如何使用 read 成员函数输入原始二进制形式的 x，这是先前使用 write 的示例的反向操作。

```
cin.read((char*)&x, sizeof(x));
```

### 13.3.7 查看输入

可以使用 peek 成员函数查看流中的下一个字符，而不必提取该字符。例如：

```
if (cin.peek()!= c) return 0;
```

### 13.3.8 提取空白

缺省情况下，iostream 提取器跳过前导空白。以下示例先关闭了 cin 跳过空白功能，然后将其打开：

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
...
cin.setf(ios::skipws); // turn it on again
```

可以使用 iostream 操纵符 ws 从 iostream 中删除前导空白，不论是否启用了跳过功能。以下示例说明了如何从 iostream istr 中删除前导空白：

```
istr >> ws;
```

## 13.3.9 处理输入错误

按照约定，第一个参数为非零错误状态的提取器不能从输入流提取任何数据，且不能清除任何错误位。失败的提取器至少应该设置一个错误位。

对于输出错误，您应该定期检查错误状态，并在发现非零状态时采取某些操作（诸如终止）。! 运算符测试 iostream 的状态。例如，如果输入字母字符用于输入，以下代码就会产生输入错误：

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
     cerr << message << "\n";
     exit(1);
}
int main() {
     cout << "Enter some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
     return 0;
}
```

类 ios 具有可用于错误处理的成员函数。详细信息请参见手册页。

## 13.3.10 结合使用 iostream 与 stdio

可以将 stdio 用于 C++ 程序，但在程序内的同一标准流中混合使用 iostream 与 stdio 时，可能会发生问题。例如，如果同时向 stdout 和 cout 写入，会发生独立缓冲，并产生不可预料的结果。如果从 stdin 与 cin 两者进行输入，问题会更加严重，因为独立缓冲可能会导致输入不可用。

要消除标准输入、标准输出和标准错误中的这种问题，就请在执行输入或输出前使用以下指令。它将所有预定义的 iostream 与相应的预定义 stdio 文件连接起来。

```
ios::sync_with_stdio();
```

因为在预定义流作为连接的一部分成为无缓冲流时，性能会显著下降，所以该类型的连接不是缺省连接。您可以在同一程序中使用应用于不同文件的 stdio 和 iostream，也就是说，您可以使用 stdio 例程写入到 stdout，同时还可以写入到连接到 iostream 的其他文件。可以打开 stdio 文件进行输入，也可以从 cin 读取，只要不同时尝试从 stdin 读取即可。

## 13.4 创建 iostream

要读取或写入不是预定义的 iostream 的流，需要创建自己的 iostream。通常，这意味着创建 iostream 库中所定义类型的对象。本节讨论了可用的各种类型：

### 13.4.1 使用类 fstream 处理文件

处理文件类似于处理标准输入和标准输出；类 ifstream、ofstream 和 fstream 分别从类 istream、ostream 和 iostream 派生而来。作为派生的类，它们继承了插入和提取运算符（以及其他成员函数），还有与文件一起使用的成员和构造函数。

可将文件 fstream.h 包括进来以使用任何 fstream。如果只要执行输入，请使用 ifstream；如果只要执行输出，请使用 ofstream；如果要对流执行输入和输出，请使用 fstream。将文件名称用作构造函数参数。

例如，将文件 thisFile 复制到文件 thatFile，如以下示例所示：

```
ifstream fromFile("thisFile");
if     (!fromFile)
    error("unable to open 'thisFile' for input");
ofstream toFile ("thatFile");
if     (!toFile)
    error("unable to open 'thatFile' for output");
char c;
while (toFile && fromFile.get(c)) toFile.put(c);
```

该代码执行以下操作：

- 使用 ios::in 的缺省模式创建名为 fromFile 的 ifstream 对象，并将其连接到 thisFile。它打开 thisFile。
- 检查新的 ifstream 对象的错误状态，如果它处于失败状态，则调用 error 函数（必须在程序的其他地方定义）。
- 使用 ios::out 的缺省模式创建名为 toFile 的 ofstream 对象，并将其连接到 thatFile。
- 按上文所述检查 toFile 的错误状态。
- 创建 char 变量用于存放传递的数据。
- 将 fromFile 的内容复制到 toFile，每次一个字符。

注 – 当然，每次复制一个字符这种方法不适用于复制文件。该代码只是一个 fstreams 使用示例。应该将与输入流关联的 streambuf 插入输出流中。请参见第 141 页中的 “13.10 处理 streambuf 流”和 sbufpub(3CC4) 手册页。

### 13.4.1.1 打开模式

该模式由枚举类型 open_mode 中的 or-ing 位构造，它是类 ios 的公有类型，其定义如下：

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

注 – UNIX 中不需要 binary 标志，提供该标志是为了与需要它的系统兼容。可移植代码在打开二进制文件时要使用 binary 标志。

您可以打开文件同时用于输入和输出。例如，以下代码打开了文件 someName 用于输入和输出，同时将其连接到 fstream 变量 inoutFile。

```
fstream inoutFile("someName", ios::in|ios::out);
```

### 13.4.1.2 在未指定文件的情况下声明 fstream

可以在未指定文件的情况下声明 fstream，并在以后打开该文件。以下示例创建了 ofstream toFile，用于写入。

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

### 13.4.1.3 打开和关闭文件

可以关闭 fstream，然后使用另一文件打开它。例如，要在命令行上处理提供的文件列表：

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

### 13.4.1.4 使用文件描述符打开文件

如果了解文件描述符（如整数 1 表示标准输出），可以按如下方式将其打开：

```
ofstream outfile;
outfile.attach(1);
```

如果通过向 fstream 构造函数之一提供文件名或使用 open 函数来打开文件，则在通过 delete 销毁 fstream 或其超出作用域时，会自动关闭该文件。将文件 attach 到 fstream 时，不会自动关闭该文件。

#### 13.4.1.5 在文件内重新定位

您可以在文件中改变读取和写入的位置。有多个工具可以达到这个目的。

- streampos 是可以记录 iostream 中的位置的类型。
- tellg (tellp) 是报告文件位置的 istream (ostream) 成员函数。因为 istream 和 ostream 是 fstream 的父类，所以 tellg 和 tellp 还可以作为 fstream 类的成员函数调用。
- seekg (seekp) 是查找给定位置的 istream (ostream) 成员函数。
- seek_dir enum 指定相对位置以用于 seek。

```
enum seek_dir {beg=0, cur=1, end=2};
```

例如，给定 fstream aFile：

```
streampos original = aFile.tellp();     //save current position
aFile.seekp(0, ios::end); //reposition to end of file
aFile << x;               //write a value to file
aFile.seekp(original);    //return to original position
```

seekg (seekp) 可以采用一个或两个参数。如果有两个参数，第一个参数是相对于 seek_dir 值（也就是第二个参数）指示的位置的位置。例如：

```
aFile.seekp(-10, ios::end);
```

从终点移动 10 个字节

```
aFile.seekp(10, ios::cur);
```

从当前位置向前移 10 个字节。

---

**注 –** 并不能方便地在文本流中进行任意查找，但总是可以返回到以前保存的 streampos 值。

---

## 13.5 iostream 赋值

iostream 不允许将一个流赋值给另一个流。

复制流对象的问题是当前存在两个可以独立更改的状态信息版本，例如输出文件中指向当前写入位置的指针。因此，可能会出现某些问题。

# 13.6 格式控制

ios(3CC4) 手册页中详细讨论了格式控制。

# 13.7 操纵符

操纵符是可以在 iostream 中插入或提取以起到特殊作用的值。

参数化操纵符是具有一个或多个参数的操纵符。

因为操纵符是普通的标识符，因此会用完可能的名称，而 iostream 不会为每个可能的函数定义操纵符。本章的其他部分讨论了各种操纵符和成员函数。

下表中介绍了 13 个预定义的操纵符。该表假定以下事项属实：

- i 的类型为 long。
- n 的类型为 int。
- c 的类型为 char。
- istr 是输入流。
- ostr 是输出流。

**表 13–2** iostream 的预定义操纵符

| | 预定义的操纵符 | 说明 |
|---|---|---|
| 1 | ostr << dec、istr >> dec | 以 10 为基数进行整数转换。 |
| 2 | ostr << endl | 插入一个换行符 ('\n') 并调用 ostream::flush()。 |
| 3 | ostr << ends | 插入一个空 (0) 字符。这在处理 strstream 时很有用。 |
| 4 | ostr << flush | 调用 ostream::flush()。 |
| 5 | ostr << hex、istr >> hex | 以 16 为基数进行整数转换。 |
| 6 | ostr << oct、istr >> oct | 以 8 为基数进行整数转换。 |
| 7 | istr >> ws | 提取空白字符（跳过空白），直至找到非空白字符（留在 istr 中）。 |
| 8 | ostr << setbase(*n*), istr >> setbase(*n*) | 将转换基数设置为 n（仅限 0、8、10、16）。 |
| 9 | ostr << setw(*n*), istr >> setw( *n*) | 调用 ios::width(n)。将字段宽度设置为 n。 |
| 10 | ostr << resetiosflags(*i*), istr>> resetiosflags(*i*) | 根据 i 中设置的位，清除标志位向量。 |

表 13–2 iostream 的预定义操纵符 （续）

| | 预定义的操纵符 | 说明 |
|---|---|---|
| 11 | `ostr << setiosflags(i)`、`istr >> setiosflags(i)` | 根据 `i` 中设置的位，设置标志位向量。 |
| 12 | `ostr << setfill(c)`、`istr >> setfill(c)` | 将填充字符（用来填充字段）设置为 `c`。 |
| 13 | `ostr << setprecision(`*n*`)`, `istr >> setprecision(`*n*`)` | 将浮点精度设置为 `n` 位数。 |

要使用预定义的操纵符，必须在程序中包含文件 `iomanip.h`。

您可以定义自己的操纵符。操纵符的两个基本类型为：

- 无格式操纵符－采用 `istream&`、`ostream&` 或 `ios&` 参数，对流进行操作，然后返回其参数。
- 参数化操纵符－采用 `istream&`、`ostream&` 或 `ios&` 参数以及一个附加参数，对流进行操作，然后返回其流参数。

## 13.7.1 使用无格式操纵符

无格式操纵符是执行以下操作的一个函数：

- 执行到流的引用
- 以某种方式操作流
- 返回其参数

由于为 `iostream` 预定义了接受指向此类函数的指针的移位运算符，因此可以在输入或输出运算符序列中放入函数。移位运算符会调用函数而不是尝试读取或写入值。以下示例显示了将 `tab` 插入 `ostream` 的 `tab` 操纵符：

```
ostream& tab(ostream& os) {
              return os <<'\t';
            }
...
cout << x << tab << y;
```

该示例详细描述了实现以下代码的方法：

```
const char tab = '\t';
...
cout << x << tab << y;
```

下面示例显示了无法用简单常量来实现的代码。假设要对输入流打开或关闭空白跳过功能。可以分别调用 `ios::setf` 和 `ios::unsetf` 来打开和关闭 `skipws` 标志，也可以定义两个操纵符。

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
       is.setf(ios::skipws, ios::skipws);
       return is;
}
istream& skipoff(istream& is) {
       is.unsetf(ios::skipws);
       return is;
}
...
int main ()
{
      int x,y;
      cin >> skipon >> x >> skipoff >> y;
      return 1;
}
```

## 13.7.2 参数化操纵符

iomanip.h 中包含的其中一个参数化操纵符是 setfill。setfill 设置用于填写字段宽度的字符。该操纵符是按下例所示的方式实现的：

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

参数化操纵符的实现分为两部分：

- **操纵符**。它使用一个额外的参数。在上面的代码示例中，采用了额外的 int 参数。由于未对这个操纵符函数定义移位运算符，所以您无法将它放至输入或输出操作序列中。相反，您必须使用辅助函数 applicator。
- *applicator*。它调用该操纵符。applicator 是全局函数，您会为它生成在头文件中可用的原型。通常操纵符是文件中的静态函数，该文件包含了 applicator 的源代码。该操纵符仅由 applicator 调用。如果将其指定为静态的，您需要确保它的名称不在全局地址空间中。

头文件 iomanip.h 中定义了多个类。每个类都保存一个操纵符函数的地址和一个参数的值。manip (3CC4) 手册页中介绍了 iomanip 类。上一示例使用了 smanip_int 类，该类可与 ios 一起使用。因为该类与 ios 一起使用，所以也可以与 istream 和 ostream 一起使用。上面的示例还使用了另一个类型为 int 的参数。

applicator 创建并返回类对象。在上面的代码示例中，类对象是 smanip_int，其中包含了操纵符和 applicator 的 int 参数。iomanip.h 头文件定义了用于该类的移位运算符。如果 applicator 函数 setfill 在输入或输出操作序列中，会调用该 applicator 函数，且其返回一个类。移位运算符作用于该类，以使用其参数值（存储在类中）调用操纵符函数。

在以下示例中，操纵符 print_hex 执行以下操作：

- 将输出流设置成十六进制模式
- 将 long 值插入流中
- 恢复流的转换模式

此处使用了类 omanip_long，因为该代码示例仅用于输出。它对 long 而不是对 int 进行操作：

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

## 13.8 strstream：用于数组的 iostream

请参见 strstream(3CC4) 手册页。

## 13.9 stdiobuf：用于 stdio 文件的 iostream

请参见 stdiobuf(3CC4) 手册页。

## 13.10 处理 streambuf 流

iostream 是由两部分（输入或输出）构成的系统的格式化部分。系统的其他部分由 streambuf 流组成，这些流处理无格式字符流的输入或输出。

通常，可以通过 iostream 使用 streambuf 流，因此您不需要详细了解它们。也可以直接使用 streambuf 流，例如，当您需要提高效率，或者需要绕开 iostream 中内置的错误处理或格式设置时。

## 13.10.1 streambuf 指针类型

streambuf 由字符流或字符序列和一个或两个指向相应序列的指针组成。每个指针都指向两个字符间。（实际上，指针无法指向字符之间，但可以按这种方式理解指针。）有两种 streambuf 指针：

- *put* 指针，它指向下一个字符的存储位置前面
- *get* 指针，它指向要获取的下一个字符前面

streambuf 可以有其中一个指针，也可以两个全有。

可以使用多种方法来操作指针的位置和序列的内容。操作两个指针时它们是否都会移动取决于使用 streambuf 的种类。通常，使用队列式 streambuf 流时，get 和 put 指针独立移动。使用文件式 streambuf 流时，get 和 put 指针始终一起移动。例如，strstream 是队列式流，fstream 是文件式流。

## 13.10.2 使用 streambuf 对象

从来不创建实际的 streambuf 对象，而是只创建从 streambuf 类派生的类的对象。示例有 filebuf 和 strstreambuf，filebuf(3CC4) 和 ssbuf(3) 手册页中对它们进行了介绍。高级用户可能想从 streambuf 派生自己的类，以便提供特定设备的接口或提供基本缓冲以外的功能。sbufpub(3CC4) 和 sbufprot(3CC4) 手册页中讨论了如何执行此操作。

除了创建自己的特殊种类的 streambuf 外，您可能还想通过访问与 iostream 关联的 streambuf 来访问公用成员函数（如以上手册页中所述）。此外，每个 iostream 都有采用 streambuf 指针的已定义插入器和提取器。插入或提取 streambuf 时，会复制整个流。

以下示例显示了可用来执行之前讨论的文件复制的另一种方法，为提高清晰性，该方法中省略了错误检查：

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

如之前一样打开输入和输出文件。每个 iostream 类都有成员函数 rdbuf，它返回指向与其关联的 streambuf 对象的指针。如果是 fstream，则 streambuf 对象是类型 filebuf。与 fromFile 关联的整个文件都复制到（插入）与 toFile 关联的文件。最后一行也可以改写如下：

```
fromFile >> toFile.rdbuf();
```

然后源文件被提取到目标中。两种方法是完全等同的。

# 13.11 iostream 手册页

许多 C++ 手册页都介绍了 iostream 库的详细信息。下表概述了每个手册页中的内容。

要访问传统 iostream 库手册页，请键入：

```
example% man -s 3CC4 name
```

表 13–3 iostream 手册页概述

| 手册页 | 概述 |
| --- | --- |
| filebuf | 详细介绍了从 streambuf 派生并专用于文件的类 filebuf 的公用接口。有关从类 streambuf 继承的功能的详细信息，请参见 sbufpub(3CC4) 和 sbufprot(3CC4) 手册页。可通过类 fstream 使用 filebuf 类。 |
| fstream | 详细介绍了类 ifstream、ofstream 和 fstream 的专用成员函数，这些类是用于文件的 istream、ostream 和 iostream 专用版本。 |
| ios | 详细介绍了作为 iostream 的基类的类 ios 的各个部分。该类也包含了所有流公共的状态数据。 |
| ios.intro | 简要介绍了 iostream。 |
| istream | 详细说明了以下内容：<br>■ 类 istream 的成员函数，这些函数支持对从 streambuf 获取的字符进行解释<br>■ 输入格式化<br>■ 归为类 ostream 的一部分的定位函数<br>■ 某些相关函数<br>■ 相关操纵符 |
| manip | 介绍了 iostream 库中定义的输入和输出操纵符。 |
| ostream | 详细说明了以下内容：<br>■ 类 ostream 的成员函数，这些函数支持对写入 streambuf 的字符进行解释<br>■ 输出格式化<br>■ 归为类 ostream 的一部分的定位函数<br>■ 某些相关函数<br>■ 相关操纵符 |
| sbufprot | 介绍了对从类 streambuf 派生的类进行编码的程序员所需的接口。有关 sbufprot(3CC4) 手册页中未讨论的一些公用函数，另请参见 sbufpub(3CC4) 手册页。 |

表 13–3 iostream 手册页概述 （续）

| 手册页 | 概述 |
| --- | --- |
| sbufpub | 详细介绍了 streambuf 类的公用接口，尤其是 streambuf 的公用成员函数。该手册页包含了直接处理 streambuf 类型的对象所需的信息，或是查找从 streambuf 派生的类从其继承的函数所需的信息。如果要从 streambuf 派生类，另请参见 sbufprot(3CC4) 手册页。 |
| ssbuf | 详细介绍了从 streambuf 派生并专用于处理字符数组的类 strstreambuf 的专用公用接口。有关从类 streambuf 继承的功能的详细信息，请参见 sbufpub(3CC4) 手册页。 |
| stdiobuf | 包含类 stdiobuf 的简要描述，该类是从 streambuf 中派生的并专用于处理 stdio FILE。有关从类 streambuf 继承的功能的详细信息，请参见 sbufpub(3CC4) 手册页。 |
| strstream | 详细介绍了由从 iostream 派生并专用于处理字符数组的一组类实现的 strstream 的专用成员函数。 |

# 13.12 iostream 术语

iostream 库说明中常常使用一些与一般编程中的术语类似的术语，但有特殊含义。下表阐明了在讨论 iostream 库时使用的这些术语的定义。

表 13–4 iostream 术语

| iostream 术语 | 定义 |
| --- | --- |
| 缓冲区 | 该词有两个含义，一个特定于 iostream 软件包，另一个较常适用于输入和输出。<br>与 iostream 库特定相关时，缓冲区是由类 streambuf 定义的类型的对象。<br>通常，缓冲区是一个内存块，用于将字符高效传输到输出的输入。对于已缓冲的 I/O，缓冲区已满或被强制刷新之前，字符的实际传输会延迟。<br>无缓冲的缓冲区是指在其中没有上文定义的通用意义的缓冲区的 streambuf。本章避免了使用术语缓冲区来指代 streambuf。但是，手册页和其他 C++ 文档确实使用术语缓冲区来指代 streambuf。 |
| 提取 | 从 iostream 获取输入的过程。 |
| Fstream | 专用于文件的输入或输出流。以 monospace 字体输出时，特指从类 iostream 派生的类。 |
| 插入 | 将输出发送到 iostream 中的过程。 |
| iostream | 通常为输入或输出流。 |

表 13–4 iostream 术语 （续）

| iostream 术语 | 定义 |
| --- | --- |
| iostream 库 | 表示通过 include 文件 iostream.h、fstream.h、strstream.h、iomanip.h 和 stdiostream.h 实现的库。因为 iostream 是面向对象的库，所以应扩展该库。 |
| 流 | 通常是指 iostream、fstream、strstream 或用户定义的流。 |
| streambuf | 包含字符序列的缓冲区，其中字符具有 put 或 get 指针（或兼有）。以 monospace 字体输出时，它表示特定类。否则，通常是指 streambuf 类或从 streambuf 派生的类的对象。任何流对象都包含从 streambuf 派生的类型的对象或指向对象的指针。 |
| strstream | 专用于字符数组的 iostream。以 monospace 字体输出时，它表示特定类。 |

◆ ◆ ◆ 第 14 章

# 14 生成库

本章解释了如何生成您自己的库。

## 14.1 了解库

库具有两点好处。首先，它们提供了在多个应用程序间共享代码的方法。如果您有要共享的代码，则可以创建一个具有该代码的库，并将该库链接到需要这些代码的应用程序。其次，库提供了降低大型应用程序复杂性的方法。这类应用程序可以将相对独立的部分生成为库并进行维护，因此减轻程序员在其他部分工作的负担。

生成库只不过是创建 `.o` 文件（使用 `-c` 选项编译代码）并使用 `CC` 命令将 `.o` 文件并入库中。可以生成两种库：静态（归档）库和动态（共享）库。

对于静态（归档）库，库中的对象在链接时链接到程序的可执行文件中。只有库中属于应用程序所需的那些 `.o` 文件链接到可执行文件。静态（归档）库名称通常以 `.a` 后缀结尾。

对于动态（共享）库，库中的对象并不链接到程序的可执行文件，而是由链接程序在可执行文件中注明程序依赖于该库。执行该程序时，系统会装入程序所需的动态库。如果使用同一动态库的两个程序同时执行，那么操作系统在程序间共享这个动态库。动态（共享）库的名称以 `.so` 后缀结尾。

动态链接共享库较静态链接归档库有多个优势：

- 可执行文件较小。
- 在运行时，代码的有效部分可在程序间共享，这样就可以降低内存使用量。
- 库可以在运行时替换，无需重新链接应用程序。（这是用于使程序能够利用 Oracle Solaris 操作系统中的多项改进的主要机制，有了此机制，无需重新链接和分发程序。）
- 共享库可以在运行时通过使用 `dlopen()` 函数调用来装入。

但动态库也具有一些缺点：

- 运行时链接有执行时间成本。
- 使用动态库进行程序的分发可能会要求同时分发该程序所使用的库。
- 将共享库移动到一个不同的位置就可以阻止系统查找该库并执行程序。（环境变量 LD_LIBRARY_PATH 可以帮助克服此问题。）

# 14.2 生成静态（归档）库

生成静态（归档）库的机制与生成可执行文件相似。可以使用 CC 的 -xar 选项将一组目标 (.o) 文件组合到单个库中。

可以使用 CC -xar 而非直接使用 ar 命令来生成静态（归档）库。C++ 语言通常要求编译器维护的信息比传统 .o 文件提供的信息多，尤其是模板实例。– xar 选项可确保所有必要信息（包括模板实例）都包括在库中。在常规编程环境下，可能无法完成该操作，因为 make 可能无法确定实际创建和引用了哪些模板文件。如果没有 CC -xar，所引用的模板实例可能不能根据需要包括在库中。例如：

```
% CC -c foo.cc # Compile main file, templates objects are created.
% CC -xar -o foo.a foo.o # Gather all objects into a library.
```

–xar 标志会使 CC 创建静态（归档）库。要为新建的库命名，需要使用 – o 指令。编译器检查命令行上的目标文件，交叉引用这些目标文件与模板系统信息库中的目标文件，并将用户的目标文件所需的模板（以及主目标文件本身）添加到归档文件中。

---

**注 –** 仅可将 -xar 标志用于创建或更新现有归档文件。不要用它来维护归档。-xar 选项与 ar -cr 等效。

---

在每个 .o 文件中只放置一个函数。如果您链接了某个归档文件，则在需要该归档文件中特定 .o 文件中的符号时，该 .o 文件将整个链接到应用程序中。当每个 .o 文件中只放置一个函数时，将只会从归档文件中链接应用程序所需的那些符号。

# 14.3 生成动态（共享）库

动态（共享）库的生成方式与静态（归档）库的生成方式基本相同，不过，您要在命令行上使用 -G 而不是 -xar。

不应直接使用 ld。与静态库一样，CC 命令可以确保使用模板时，模板系统信息库中所有必要的模板实例都包括在库中。在执行 main() **之前**会调用与应用程序链接的动态库中所有静态构造函数，在 main() 退出**之后**会调用所有静态析构函数。如果使用 dlopen() 打开共享库，所有静态构造函数都在执行 dlopen() 时执行，所有静态析构函数都在执行 dlclose() 时执行。

应该使用 CC -G 来生成动态库。使用 ld（链接编辑器）或 cc（C 编译器）生成动态库时，异常可能无法生效，且库中定义的全局变量未初始化。

要生成动态（共享）库，必须使用 CC 的 –Kpic 或 –KPIC 选项编译每个对象来创建可重定位的目标文件。然后您就可以生成一个具有这些可重定位目标文件的动态库。如果遇到异常的链接故障，则可能是您忘记了使用 –Kpic 或 –KPIC 编译某些对象。

要生成名为 libfoo.so 的 C++ 动态库（该库包含源文件 lsrc1.cc 和 lsrc2.cc 中的对象），请键入：

```
% CC -G -o libfoo.so -h libfoo.so -Kpic lsrc1.cc lsrc2.cc
```

-G 选项指定动态库的构造。-o 选项指定库的文件名。-h 选项指定共享库的内部名称。-Kpic 选项指定目标文件与位置无关。

CC -G 命令不会将任何 -l 选项传递给链接程序 ld。为了确保正确的初始化顺序，共享库对其所需的每个其他共享库必须具有显式的依赖性。要创建依赖性，请对每个此类库使用 -l 选项。典型的 C++ 共享库将使用以下几组选项之一：

```
-lCstd -lCrun -lc
-library=stlport4 -lCrun -lc
```

为了确保列出了需要的所有依赖性，请使用 -zdefs 选项生成库。对于缺少的每个符号定义，链接程序都会发出错误消息。要提供缺少的定义，请针对这些库添加 -l 选项。

要确定是否包含了不需要的依赖性，请使用以下命令

```
ldd -u -r mylib.so
ldd -U -r mylib.so
```

然后可以重新生成没有不需要的依赖性的 mylib.so。

## 14.4 生成包含异常的共享库

切勿对包含 C++ 代码的程序使用 -Bsymbolic。应该改用链接程序映射文件。如果使用 -Bsymbolic，不同模块中的引用会绑定到应是一个全局对象内容的不同副本。

异常机制依赖对地址的比较。如果您具有某个对象的两个副本，它们的地址就不等同且异常机制可能失败，这是因为异常机制依赖于对假设为唯一地址的内容的比较。

## 14.5 生成专用的库

在组织生成一个仅供内部使用的库时，可以使用不建议在一般情况下使用的选项来生成这个库。具体来说，库不需要符合系统的应用程序二进制接口 (application binary interface, ABI)。例如，可以使用 -fast 选项编译库，以提高其在某已知体系结构上的性能。同样，可以使用 -xregs=float 选项编译库以提高性能。

# 14.6 生成公用的库

在组织生成一个供其他公司使用的库时，库的管理、平台的一般性以及其他问题就变得尤为重要。用于检验库是否为公用库的一个简单测试就是询问应用程序程序员是否可以轻松地重新编译该库。生成公用库时应该符合系统的应用程序二进制接口 (application binary interface, ABI)。通常，这意味着应该避免任何特定于处理器的选项。（例如，不要使用 `-fast` 或 `-xtarget`。）

SPARC ABI 为应用程序保留了一些专用寄存器。对于 SPARC V7 和 V8，这些寄存器是 `%g2`、`%g3` 和 `%g4`。对于 SPARC V9，这些寄存器是 `%g2` 和 `%g3`。由于大多数编译是针对应用程序的，所以在缺省情况下，为了提高程序的性能，C++ 编译器使用这些寄存器作为临时寄存器。但是，对公用库中寄存器的使用通常不兼容于 SPARC ABI。生成公用库时，请使用 `-xregs=no%appl` 选项编译所有对象，以确保不会使用应用程序寄存器。

# 14.7 生成具有 C API 的库

如果要生成以 C++ 编写但可用于 C 程序的库，必须创建 C API（application programming interface，应用程序编程接口）。为此，应先使所有导出的函数为 `extern "C"`。注意，只有在全局函数中才能够完成该操作，在成员函数中不行。

如果 C 接口库需要 C++ 运行时支持，且要使用 `cc` 进行链接，则在使用 C 接口库时，您还必须将您的应用程序与 `libCrun` 进行链接（标准模式）。（如果 C 接口库不需要 C++ 运行时支持，则不必与 `libCrun` 进行链接。）归档库与共享库的链接步骤是不同的。

提供**归档**的 C 接口库时，必须提供如何使用该库的说明。

- 如果 C 接口库是在**标准模式**（缺省模式）下使用 `CC` 生成的，那么在使用该 C 接口库时，将 `-lCrun` 添加到 `cc` 命令行。
- 如果 C 接口库是在**兼容模式** (`-compat=4`) 下使用 `CC` 生成的，则在使用 C 接口库时，请将 `-lC` 添加到 `cc` 命令行。

提供**共享的** C 接口库时，必须在生成库时创建对 `libCrun` 的依赖性。如果共享库具有正确的依赖性，则在使用该库时不需要将 `-lCrun` 添加到命令。

- 如果要在缺省**标准模式**下生成 C 接口库，则在生成该库时应该将 `-lCrun` 添加到 `CC` 命令。

如果要删除对 C++ 运行时库的任何依赖性，应该在库源文件中强制应用下列代码规则：

- 不要使用任何形式的 `new` 或 `delete`，除非提供了自己的相应版本。
- 不要使用异常。
- 不要使用运行时类型信息 (RTTI)。

# 14.8 使用 dlopen 从 C 程序访问 C++ 库

如果要使用 `dlopen()` 从 C 程序打开 C++ 共享库，应确保共享库依赖于适当的 C++ 运行时（对于 `-compat=5`，为 `libCrun.so.1`）。

为此，对于 `-compat=5`，在生成共享库时，应将 `-lCrun` 添加到命令行。例如：

```
example% CC -G -compat=5... -lCrun
```

如果共享库使用了异常且不具有对 C++ 运行库的依赖性，则 C 程序可能会出现无规律的行为。

第 4 部分

# 附录

◆ ◆ ◆ 附 录 A

# C++ 编译器选项

本附录详细介绍了 C++ 编译器的命令行选项。所介绍的功能适用于所有平台（特别注明的除外）；特定于 SPARC 系统的 Oracle Solaris OS 功能用 *SPARC* 标识，特定于 x86 系统的 Oracle Solaris 和 Linux OS 功能用 *x86* 标识。仅限于 Oracle Solaris OS 的功能用 *Solaris* 标记；仅限于 Linux OS 的功能用 *Linux* 标记。

本手册的此部分使用前言中列出的印刷约定来说明各个选项。

圆括号、大括号、方括号、"|" 或 "-" 字符以及省略号是选项描述中使用的**元字符**，而不是选项自身的一部分。

## A.1 选项信息的结构

为了帮助您查找信息，编译器选项描述被分为以下几个子节。如果一个选项被其他选项取代或与其他选项一致，就请参见其他选项的说明以获取完整的详细信息。

**表 A–1** 选项子节

| 子节 | 内容 |
|---|---|
| 选项定义 | 紧跟在每个选项之后的简短定义。（该类无标题。） |
| 值 | 如果选项具有一个或多个值，则本节将定义每个值。 |
| 缺省值 | 如果选项具有主缺省值或辅助缺省值，则在此处进行声明。<br>如果未指定选项，则主缺省值为有效选项值。例如，如果未指定 `–compat`，则缺省值为 `–compat=5`。<br>如果指定了选项但不给定任何值，则辅助缺省值为有效选项值。例如，如果指定了 `–compat` 但未提供值，则缺省值为 `-compat=5`。 |
| 扩展 | 如果选项具有宏扩展，则将在本节中显示。 |

表 A–1 选项子节 （续）

| 子节 | 内容 |
| --- | --- |
| 示例 | 如果要举例说明选项，则在此处给出所需示例。 |
| 交互 | 如果选项与其他选项进行交互，则在此处讨论它们的关系。 |
| 警告 | 此处将注明与选项使用相关的警告（例如可能会产生意外行为的操作）。 |
| 另请参见 | 本节包含到其他选项或文档中更多信息的引用。 |
| “替换为”、“与...相同” | 如果选项已废弃且已被其他选项替换，则在此处说明替换的选项。以后的发行版本可能不支持以这种方式描述的选项。<br>如果有两个选项具有相同的含义和用途，则在此处引用首选项。例如，“与 `-xO` 相同”表示 `-xO` 是首选项。 |

# A.2 选项参考

本节按字母顺序列出所有的 C++ 编译器选项，并指出所有的平台限制。

## A.2.1 -#

打开详细模式，显示命令选项的扩展方式。显示调用的每个组件。

## A.2.2 -###

按照调用每个组件时的方式显示该组件，但不实际执行该组件。还显示命令选项扩展的过程。

## A.2.3 –Bbinding

指定 链接的库绑定是`symbolic`、`dynamic`（共享）还是 `static`（非共享）。

可以在命令行上多次使用 `–B` 选项。该选项传递给链接程序 `ld`。

---

**注 –** 在 Oracle Solaris 64 位编译环境中，许多系统库只能用作动态库。因此，请勿在命令行上将 `-Bstatic` 用作最后一个切换开关。

---

### A.2.3.1 值

*binding* 必须是下表中列出的值之一：

| 值 | 含义 |
| --- | --- |
| dynamic | 指示链接编辑器查找 lib*lib*.so（共享）文件，如果未找到这些文件，则查找 lib*lib*.a（静态非共享）文件。当链接需要共享库绑定时，请使用该选项。 |
| static | 指示链接编辑器只查找 lib*lib*.a（静态非共享）文件。当链接需要非共享库绑定时，请使用该选项。 |
| symbolic | 如果可能，则强制在共享库中解析符号（即使符号已经在别处定义）。<br>请参见 ld(1) 手册页。 |

（–B 和 *binding* 值之间不能有空格。）

## 缺省值

如果没有指定 -B，则使用 –Bdynamic。

## 交互

要静态链接 C++ 缺省库，请使用 –staticlib 选项。

-Bstatic 和 -Bdynamic 选项会影响缺省情况下提供的库的链接。为了确保动态链接缺省库，最后使用的 –B 应该是 –Bdynamic。

在 64 位环境中，许多系统库只能用作共享动态库。其中包括 libm.so 和 libc.so（不提供 libm.a 和 libc.a）。因此，在 64 位 Oracle Solaris 操作系统环境中，-Bstatic 和 -dn 可能会导致产生链接错误。这些情况下应用程序必须与动态库链接。

## 示例

以下编译器命令链接 lib*foo*.a，即使 lib*foo*.so 存在也是如此，所有其他库都是动态链接的：

```
example% CC a.o –Bstatic –lfoo –Bdynamic
```

## 警告

对于包含 C++ 代码的程序，切勿使用 -Bsymbolic，而应使用链接程序映射文件。

如果使用 -Bsymbolic，不同模块中的引用会绑定到应是一个全局对象内容的不同副本。

异常机制依赖对地址的比较。如果您具有某项内容的两个副本，它们的地址就不等同且异常机制可能失败，这是由于异常机制依赖对假设为唯一地址内容的比较。

如果在不同的步骤中进行编译和链接，并要使用 -B*binding* 选项，就必须在链接步骤中包括该选项。

### 另请参见

–nolib、–staticlib、ld(1) 手册页以及第 114 页中的 “11.5 静态链接标准库”和《链接程序和库指南》

## A.2.4 –c

仅编译；生成 .o 目标文件，但抑制链接。

该选项指示 CC 驱动程序抑制通过 ld 进行链接，并为每个源文件生成一个 .o 文件。如果只在命令行上指定一个源文件，就可以用 -o 选项显式指定目标文件。

### A.2.4.1 示例

如果输入 **CC -c x.cc**，则会生成 x.o 目标文件。

如果输入 **CC -c x.cc -o y.o**，则会生成 y.o 目标文件。

### 警告

当编译器为输入文件 (.c、.i) 生成目标代码时，编译器总是在工作目录下生成 .o 文件。如果抑制链接步骤，则不会删除 .o 文件。

### 另请参见

–o *filename* 和 –xe

## A.2.5 –cg{89|92}

(SPARC) 已废弃，请勿使用此选项。当前的 Oracle Solaris 操作系统软件不再支持 SPARC V7 体系结构。使用此选项编译生成的代码在当前的 SPARC 平台中运行较慢。应改用 -xO，并利用 -xarch、-xchip 和 -xcache 编译器缺省值。

## A.2.6 –compat={ 5|g}

设置编译器的主要发行版兼容模式。该选项控制 __SUNPRO_CC_COMPAT 预处理程序宏。

C++ 编译器有两种主要模式。-compat=5（缺省模式）接受符合 2003 年更新的 ANSI/ISO 1998 C++ 标准的构造，并且在 -compat=5 模式下将生成与 C++ 5.0 到 5.12 兼容的代码。在 Oracle Solaris x86 和 Linux 平台上，-compat=g 选项可以添加与 gcc/g++ 编译器的源和二进制兼容性。由于在名称改编、类布局、虚拟表布局和其他 ABI 详细信息方面有不兼容的重大更改，因此这些模式互不兼容。

在以前发行版中接受由 4.2 编译器定义的语义和语言的 **兼容性模式** (`-compat=4`) 不再可用。

这些模式由 `–compat` 选项进行区分，如以下部分中所示。

### A.2.6.1 值

`-compat` 选项可以使用下表中显示的值。

| 值 | 含义 |
|---|---|
| `–compat=5` | （标准模式）设置语言和二进制使其与 ANSI/ISO 标准模式兼容。将 `__SUNPRO_CC_COMPAT` 预处理程序宏设置为 5。 |
| `-compat=g` | （仅限 x86）启用对 g++ 语言扩展的识别，并使编译器在 Solaris 和 Linux 平台上生成与 g++ 二进制兼容的代码。将 `__SUNPRO_CC_COMPAT` 预处理程序宏设置为 `'G'`。 |

使用 `-compat=g` 时，二进制兼容性仅扩展到共享（动态或 `.so`）库，而不扩展到个别 `.o` 文件或归档 (`.a`) 库。

以下示例显示了将 g++ 共享库链接到 C++ 主程序的过程：

```
% g++ -shared -o libfoo.so -fpic a.cc b.cc c.cc
% CC -compat=g main.cc -L. -lfoo
```

以下示例显示了将 C++ 共享库链接到 g++ 主程序的过程：

```
% CC -compat=g -G -o libfoo.so -Kpic a.cc b.cc c.cc
% g++ main.cc -L. -lfoo
```

#### 缺省值

如果没有指定 `–compat` 选项，则假定 `–compat=5`。

#### 交互

有关更多信息，请参见 `–features`。

#### 警告

生成共享库时，不要使用 `-Bsymbolic`。

## A.2.7 +d

请勿扩展 C++ 内联函数。

按照 C++ 语言规则，C++ 内联函数是指具有以下特征的函数，对于该函数以下陈述之一为真：

- 该函数的定义中使用了关键字 `inline`
- 该函数在类定义内部定义（不仅是声明）
- 该函数是编译器生成的类成员函数

按照 C++ 语言规则，编译器可以选择是否将调用实际内联到内联函数。C++ 编译器将调用内联到内联函数，除非下述事项之一为真：

- 函数太复杂
- 已选定 `+d` 选项
- 已选定 `-g` 选项，但未指定 `-xO`*n* 优化级别

### A.2.7.1 示例

缺省情况下，编译器可以内联以下代码示例中的函数 `f()` 和 `memf2()`。此外，该类具有编译器可以内联的由编译器生成的缺省构造函数和析构函数。使用 `+d` 时，编译器不会内联 `f()` 和 `C::mf2()`（即构造函数和析构函数）。

```
inline int f() {return 0;} // may be inlined
class C {
  int mf1(); // not inlined unless inline definition comes later
  int mf2() {return 0;} // may be inlined
};
```

#### 交互

指定了调试选项 `–g` 时将自动启用此选项，除非还指定了优化级别 (`–O` 或 `–xO`)。

但指定调试选项 `–g0` 不会启用 `+d`。

`+d` 选项对使用 `-xO4` 或 `-xO5` 时执行的自动内联没有影响。

#### 另请参见

`–g0` 和 `–g`

## A.2.8 -Dname[ =*def*]

为预处理程序定义宏符号 *name*。

使用该选项与在源文件开头包含 `#define` 指令等效。可以使用多个 `-D` 选项。

有关编译器预定义宏的列表，请参见 `CC(1)` 手册页。

## A.2.9 –d{y|n}

允许或不允许将动态库用于整个可执行文件。

该选项传递给 `ld`。

该选项只能在命令行出现一次。

### A.2.9.1 值

| 值 | 含义 |
|---|---|
| `-dy` | 在链接编辑器中指定动态链接。 |
| `–dn` | 在链接编辑器中指定静态链接。 |

#### 缺省值

如果没有指定 `-d` 选项，则使用 `–dy`。

#### 交互

在 64 位环境中，许多系统库只能用作共享动态库。其中包括 `libm.so` 和 `libc.so`（不提供 `libm.a` 和 `libc.a`）。因此，在 64 位 Oracle Solaris 操作系统中，`-Bstatic` 和 `-dn` 可能会导致产生链接错误。这些情况下应用程序必须与动态库链接。

#### 警告

如果将此选项与动态库结合使用，将导致致命错误。大多数系统库仅作为动态库可用。

#### 另请参见

`ld`(1) 手册页和《**链接程序和库指南**》

## A.2.10 –dalign

(SPARC) 已过时，不要使用。请使用 `-xmemalign=8s`。有关更多信息，请参见第 245 页中的 “A.2.145 -xmemalign=*ab*”。

在 x86 平台上会在无提示的情况下忽略此选项。

## A.2.11 –dryrun

显示但不编译驱动程序所生成的子命令。

该选项指示驱动程序 CC 显示但不执行编译驱动程序构造的子命令。

## A.2.12 –E

对源文件运行预处理程序，但不进行编译。

指示 CC 驱动程序仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout（标准输出）。此时，不进行编译，且不生成 .o 文件。

此选项会导致输出中包含预处理程序类型的行号信息。

要在源代码涉及模板时编译 -E 选项的输出，可能需要将 -template=no%extdef 选项与 -E 选项一起使用。如果应用程序代码使用“独立定义”模板源代码模型，使用这两个选项可能仍然无法编译 -E 选项的输出。有关更多信息，请参考有关模板的章节。

### A.2.12.1 示例

该选项用于确定预处理程序所进行的更改。例如，以下程序 foo.cc 会生成第 162 页中的 “A.2.12.1 示例”中所示的输出。

**示例 A–1** 预处理程序示例 foo.cc

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
.
```

**示例 A–2** 使用 -E 选项时 foo.cc 的预处理程序输出

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power (int, int);


int main () {
int x;
x = power (2, 10);
}
```

### 警告

如果代码包含采用“独立定义”模型的模板，此选项的输出可能不能用作 C++ 编译的输入。

### 另请参见

–P

## A.2.13 -erroff[= t]

此命令会抑制 C++ 编译器警告消息，但对错误消息没有影响。此选项适用于所有警告消息，无论这些警告消息是否已被 `-errwarn` 指定为导致非零退出状态。

### A.2.13.1 值

*t* 是一个以逗号分隔的列表，它包含以下项中的一项或多项：*tag*、`no%`*tag*、`%all`、`%none`。顺序是很重要的；例如 `%all,no%`*tag* 抑制除 *tag* 以外的所有警告消息。下表列出了 `-erroff` 值。

**表 A–2** -erroff值

| 值 | 含义 |
|---|---|
| *tag* | 抑制由该 *tag* 指定的警告消息。可通过 `-errtags=yes` 选项来显示消息的标记。 |
| `no%`*tag* | 启用由该 *tag* 指定的警告消息。 |
| `%all` | 禁止所有警告消息。 |
| `%none` | 启用所有警告消息（缺省）。 |

### 缺省值

缺省值为 `-erroff=%none`。指定 `-erroff` 与指定 `-erroff=%all` 等效。

### 示例

例如，`-erroff=`*tag* 将抑制由该标记指定的警告消息。另外，`-erroff=%all,no%` *tag* 抑制所有警告信息,由 *tag* 标识的消息除外。

可以使用 `-errtags=yes` 选项显示警告消息的标记。

### 警告

使用 `-erroff` 选项只能抑制来自 C++ 编译器前端且在使用 `-errtags` 选项时显示标记的警告消息。

### 另请参见

-errtags 和 -errwarn

## A.2.14 -errtags[= a]

显示来自 C++ 编译器前端且可以使用 -erroff 选项抑制或使用 -errwarn 选项使其成为致命警告的每个警告消息的消息标记。

### A.2.14.1 值和缺省值

*a* 可以是 yes 或 no。缺省值为 -errtags=no。指定 -errtags 与指定 -errtags=yes 等效。

### 警告

来自 C++ 编译器驱动程序的消息以及编译系统的其他组件不包含错误标记。因此，无法使用 -erroff 抑制这些消息或组件，也无法使用 -errwarn 将这些消息和组件标记为致命错误。

### 另请参见

-erroff 和 -errwarn

## A.2.15 -errwarn[= t]

使用 -errwarn 会导致 C++ 编译器在出现给定的警告消息时以失败状态退出。

### A.2.15.1 值

*t* 是一个以逗号分隔的列表，它包含以下项中的一项或多项：*tag*、no%*tag*、%all、%none。顺序很重要，例如如果出现除 tag 之外的任何警告，*%all,no%*tag 会使 *cc* 以致命状态退出。

下表详细列出了 -errwarn 值。

**表 A–3** -errwarn 值

| 值 | 含义 |
|---|---|
| *tag* | 如果该 *tag* 指定的消息以警告消息的形式出现，就会使 cc 以致命状态退出。如果未出现 *tag*，则没有影响。 |
| no%*tag* | 防止 cc 在由 *tag* 指定的消息只以警告消息形式出现时以致命状态退出。如果未发出 *tag* 指定的消息，则不会产生任何影响。为了避免在发出警告消息时导致 cc 以致命状态退出，可使用该选项来还原以前用该选项和 *tag* 或 %all 指定的警告消息。 |

表 A–3 -errwarn 值 （续）

| 值 | 含义 |
|---|---|
| %all | 使 CC 在出现任何警告消息时以致命状态退出。%all 可以后跟 no%*tag*，以避免该行为的特定警告消息。 |
| %none | 防止 CC 在出现任何警告消息时以致命状态退出。 |

### 缺省值

缺省值为 -errwarn=%none。单独指定 -errwarn 与 -errwarn=%all 等效。

### 警告

使用 -errwarn 选项只能对来自 C++ 编译器前端且在使用了 -errtags 选项时会显示标记的警告消息进行指定，从而使编译器以失败状态退出。

由于编译器错误检查的改善和功能的增加，C++ 编译器生成的警告消息也会因发行版本而异。使用 -errwarn=%all 进行编译而不会产生错误的代码，在编译器下一个发行版本中编译时也可能出现错误。

### 另请参见

-erroff、-errtags 和 -xwe

## A.2.16 –fast

此选项是一个宏，可以有效地用作优化可执行文件的起点，从而获得最佳运行时性能。-fast 是一个宏，随编译器发行版本的升级而变化，并扩展为目标平台特定的多个选项。可使用 -dryrun 或 -xdryrun 选项检查 -fast 的扩展，并将 -fast 的相应选项结合到正在进行的可执行文件调优过程中。

该选项是一个宏，选择编译选项的组合用于在编译代码的机器上优化执行速度。

### A.2.16.1 扩展

该选项通过扩展到以下编译选项，为大量应用程序提供了几乎最高的性能。

表 A–4 -fast 扩展

| 选项 | SPARC | x86 |
|---|---|---|
| –fns | X | X |
| –fsimple=2 | X | X |
| –nofstore | - | X |

表 A–4 -fast扩展 （续）

| 选项 | SPARC | x86 |
|---|---|---|
| -xbuiltin=%all | X | X |
| –xlibmil | X | X |
| –xlibmopt | X | X |
| –xmemalign | X | - |
| –xO5 | X | X |
| –xregs=frameptr | - | X |
| –xtarget=native | X | X |

## 交互

-fast 宏可扩展为可能影响其他指定选项的编译选项。例如，在以下命令中，-fast 宏的扩展包括了将 -xarch 还原为某个 32 位体系结构选项的 -xtarget=native。

错误：

```
example% CC -xarch=sparcvis2 -fast test.cc
```

正确：

```
example% CC -fast -xarch=sparcvis2 test.cc
```

查看每个选项的描述以确定可能的交互操作。

代码生成选项、优化级别、内建函数的优化和内联模板文件的使用可以用后续选项来覆盖（请参见示例）。指定的优化级别将覆盖以前所设置的优化级别。

–fast 选项包括 –fns–ftrap=%none，即该选项禁用所有陷阱操作。

在 x86 上，-fast 选项包括 -xregs=frameptr。有关详细信息，特别是编译混合 C、Fortran 和 C++ 源代码时，请参见该选项的介绍。

## 示例

执行以下编译器命令，优化级别将为 –xO3。

```
example% CC –fast –xO3
```

执行以下编译器命令，优化级别将为 –xO5。

```
example% CC -xO3 –fast
```

### 警告

如果在不同的步骤中进行编译和链接，则编译命令和链接命令中都必须有 -fast 选项。

使用 -fast 选项编译的目标二进制文件不可移植。例如，在 UltraSPARCIII 系统中用以下命令生成的二进制文件在 UltraSPARCII 系统中无法执行。

```
example% CC -fast test.cc
```

不要对依赖于 IEEE 标准浮点运算的程序使用此选项。可能会出现不同的数值结果、程序过早终止或出现意外的 SIGFPE 信号。

-fast 的扩展包括 -D_MATHERR_ERRNO_DONTCARE。

使用 -fast，编译器可以用不设置 errno 变量的等效优化代码自由替换对浮点函数的调用。并且，-fast 还可以定义宏 __MATHERR_ERRNO_DONTCARE，该宏允许编译器不再确保 errno 和在浮点函数调用后引发的浮点异常的有效性。因此，依赖于 errno 的值或者浮点函数调用之后引起的正确浮点异常的用户代码会产生不一致的结果。

解决此问题的一种方法是避免使用 -fast 编译此类代码。但是，如果需要 -fast 优化并且代码依赖于正确设置的 errno 值或在浮点库调用后引发的浮点异常，应在命令行上在 -fast 后使用以下选项进行编译，以禁止编译器优化此类库调用：

```
-xbuiltin=%none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

要在任何平台上显示 –fast 的扩展，请运行命令 CC –dryrun –fast，如以下示例所示。

```
>CC -dryrun -fast  |& grep ###
###     command line files and options (expanded):
### -dryrun -xO5 -xarch=sparcvis2 -xcache=64/32/4:1024/64/4 \
-xchip=ultra3i -xmemalign=8s -fsimple=2 -fns=yes -ftrap=%none \
-xlibmil -xlibmopt -xbuiltin=%all -D__MATHERR_ERRNO_DONTCARE
```

### 另请参见

-fns、-fsimple、-ftrap=%none、-xlibmil、-nofstore、-xO5、-xlibmopt 和 -xtarget=native

## A.2.17 –features=*a*[ ,a...]

启用/禁用逗号分隔的列表中指定的各种 C++ 语言功能。

### A.2.17.1 值

关键字 *a* 可以使用下表中显示的值。no% 前缀禁用关联的选项。

表 A–5 -features 值

| 值 | 含义 |
| --- | --- |
| %all | 已过时，请勿使用。启用几乎所有的 -features 选项。结果可能不可预测。 |
| [no%]altspell | 识别替用的标记拼写（例如，"and" 代替 "&&"）。缺省值为 altspell。 |
| [no%]anachronisms | 允许使用过时的构造。禁用时（即 -features=no%anachronisms），不允许使用任何过时构造。缺省值为 anachronisms。 |
| [no%]bool | 允许 bool 类型和文字。启用时，宏为 _BOOL=1。未启用时，不定义宏。缺省值为 bool。 |
| [no%]conststrings | 将文字字符串放在只读存储器中。缺省值为 conststrings。 |
| cplusplus_redef | 允许在命令行中通过 -D 选项重新定义以常规方式预定义的宏 __cplusplus。不允许在源代码中使用 #define 指令重新定义 __cplusplus。示例：<br>CC –features=cplusplus_redef –D__cplusplus=1 ...<br>g++ 编译器通常将 __cplusplus 宏预定义为 1，而某些源代码可能依赖于此非标准值。（对于实现了 1998 C++ 标准或 2003 更新的编译器，标准值是 199711L。对于该宏，将来的标准要求采用更大的值。）<br>除非您需要将 __cplusplus 重新定义为 1 以编译打算供 g++ 使用的代码，否则不要使用此选项。 |
| [no%]except | 允许 C++ 异常。C++ 异常处于禁用状态时（即 -features=no%except），接受但忽略函数的抛出规范，编译器不生成异常代码。请注意，关键字 try、throw 和 catch 始终保留。请参见第 94 页中的 “8.3 禁用异常”。缺省值为 except。 |
| explicit | 识别关键字 explicit。不允许 no%explicit 选项。 |
| [no%]export | 识别关键字 export。缺省值为 export。 |
| [no%]extensions | 允许其他 C++ 编译器通常接受的非标准代码。缺省值为 no%extensions。 |
| [no%]iddollar | 允许将 $ 符号用作非词首标识符字符。缺省值为 no%iddollar。 |
| [no%]localfor | 对 for 语句使用符合标准的局部作用域规则。缺省值为 localfor。 |
| [no%]mutable | 识别关键字 mutable。缺省值为 mutable 。 |
| namespace | 识别关键字 namespace。不允许 nono%namespace 选项。 |

表 A–5 -features 值 （续）

| 值 | 含义 |
| --- | --- |
| [no%]nestedacess | 允许嵌套类访问包容类的专有成员。缺省值：-features=nestedaccess |
| rtti | 允许运行时类型标识 (runtime type identification, RTTI)。不允许 no%rtti 选项。 |
| [no%]rvalueref | 允许将非 const 引用绑定到 rvalue 或 temporary。缺省值：-features=no%rvalueref<br>缺省情况下，C++ 强制执行非 const 引用不能绑定到 temporary 或 rvalue 的规则。要覆盖此规则，请使用选项 -features=rvalueref。 |
| [no%]split_init | 将非局部静态对象的初始化程序放到各个函数中。使用 -features=no%split_init 时，编译器将所有初始化函数放入一个函数中。使用 -features=no%split_init 可最大限度减小代码大小，但编译时间可能增加。缺省值为 split_init。 |
| [no%]transitions | 允许 ARM 语言构造，该构造在标准 C++ 下会产生问题，且可能使程序无法按预期工作或以后可能不能编译。使用 -features=no%transitions 时，编译器会将这些情况视为错误。使用 -features=transitions 时，编译器会发出关于这些构造的警告，而不是错误消息。<br>以下构造视为转换错误：在使用了模板后再重新定义模板，遗漏模板定义中所需的 typename 指令，以及隐式声明类型 int。在以后的发行版本中可能会更改转换错误的集合。缺省值为 transitions。 |
| [no%]strictdestrorder | 遵循由 C++ 标准指定的、对具有静态存储持续时间的对象的析构顺序要求。缺省值为 strictdestrorder。 |
| [no%]tmplrefstatic | 允许函数模板引用相关静态函数或静态函数模板。缺省值是遵循标准的 no%tmplrefstatic。 |
| [no%]tmplife | 清除由完整表达式末尾处的某个表达式创建的临时对象，如 ANSI/ISO C++ 标准中定义。（-features=no%tmplife 生效时，多数临时对象会在其块终结时清除。）缺省值为 tmplife。 |
| %none | 已过时，请勿使用。关闭几乎所有功能。结果可能不可预测。 |

## 交互

该选项会累积而不覆盖。

以下选项与标准库和头文件不兼容：

- no%bool
- no%except
- no%mutable

### 警告

不要使用 -features=%all 或 -features=%none。这些关键字已过时并且在以后的发行版中可能会被删除。结果可能不可预测。

使用 -features=tmplife 选项时，程序的行为可能会发生变化。测试在使用与不使用 -features=tmplife 选项两种情况下程序是否正常运行是一种测试程序可移植性的方法。

### 另请参见

表 3–17 和《C++ 迁移指南》

## A.2.18 -filt[= filter[,filter...]]

控制编译器通常应用于链接程序和编译器错误消息的过滤功能。

### A.2.18.1 值

*filter* 必须是下表中列出的值之一。%no 前缀禁用关联的子选项。

表 A–6 -filt 值

| 值 | 含义 |
|---|---|
| [no%]errors | 显示链接程序错误消息的 C++ 解释。链接程序的诊断信息被直接提供到其他工具时，可以禁止这种解释。 |
| [no%]names | 取消改编的 C++ 链接程序名称。 |
| [no%]returns | 取消改编函数的返回类型。禁止该类型的改编可以使您更快速地识别函数的名称，但请注意联合变体返回的部分函数只在返回类型上有区别。 |
| [no%]stdlib | 在链接程序和编译器错误消息中简化标准库的名称，并提供更简单的方式来识别标准库模板类型的名称。 |
| %all | 等效于 -filt=errors,names,returns,stdlib。这是缺省行为。 |
| %none | 等效于 -filt=no%errors,no%names,no%returns,no%stdlib。 |

### 缺省值

如果未指定 -filt 选项或指定了 -filt 但未提供任何值，则编译器假定 -filt=%all。

### 示例

以下示例显示了使用 -filt 选项编译该代码的效果。

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // no definition provided
};

int main()
{
    type t;
}
```

如果编译代码时不使用 `-filt` 选项，编译器就假定 `-filt=errors,names,returns,stdlib` 并显示标准输出。

```
example% CC filt_demo.cc
Undefined             first referenced
 symbol                  in file
type::~type()         filt_demo.o
type::__vtbl          filt_demo.o
[Hint: try checking whether the first non-inlined, /
non-pure virtual function of class type is defined]

ld: fatal: Symbol referencing errors. No output written to a.out
```

以下命令禁止取消改编的 C++ 链接程序名称取消改编，并禁止链接程序错误的 C++ 解释。

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

现在考虑以下代码：

```
#include <string>
#include <list>
int main()
{
    std::list<int> l;
    std::string s(l); // error here
}
```

如果指定了 `-filt=no%stdlib`，将生成以下输出：

```
Error: Cannot use std::list<int, std::allocator<int>> to initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

如果指定了 `-filt=stdlib`，将生成以下输出：

```
Error: Cannot use std::list<int> to initialize std::string .
```

### 交互

指定了 no%names 时， returns 和 no%returns 都无效。也就是说，以下选项是相同的：

- -filt=no%names
- -filt=no%names,no%returns
- -filt=no%names,returns

## A.2.19 –flags

与 – xhelp=flags 相同。

## A.2.20 -fma[={ none|fused}]

(SPARC) 启用自动生成浮点乘加指令。-fma=none 禁用这些指令的生成。-fma=fused 允许编译器通过使用浮点乘加指令，尝试寻找改进代码性能的机会。

缺省值是 -fma=none。

要使编译器生成乘加指令，最低要求是 -xarch=sparcfmaf 且优先级别至少为 -xO2。如果生成了乘加指令，编译器会标记此二进制程序，以防止该程序在不受支持的平台上执行。

乘加指令可以免除乘法和加法之间的中间舍入步骤。因此，如果使用 -fma=fused 编译，程序可能会生成不同的结果，但精度通常会增加而不是降低。

## A.2.21 –fnonstd

这是可扩展至 –ftrap=common（在 x86 上）和 –fns –ftrap=common（在 SPARC 上）的宏。

有关更多信息，请参见 –fns 和 –ftrap=common。

## A.2.22 –fns[={yes| no}]

- SPARC：启用/禁用 SPARC 的非标准浮点模式。

  如果使用 -fns=yes（或 -fns），则会在程序开始执行时启用非标准浮点模式。

  该选项提供了一种切换使用非标准浮点模式或标准浮点模式的方法，它接在包括 –fns 的其他某些宏选项（如 –fast）后面。

  在某些 SPARC 体系结构上，非标准浮点模式会禁用“逐渐下溢”，这会导致很小的结果刷新为零而不是生成次正规数。此外，还会导致次正规操作数在无提示的情况下替换为零。

对于那些硬件中不支持逐渐下溢和次正规数的 SPARC 体系结构，-fns=yes（或 -fns）可以显著提高某些程序的性能。

- x86：选择/取消选择 SSE 刷新为零模式，以及非正规数为零模式（如果可用）。

  此选项导致将次正规结果刷新为零。如果可用的话，此选项还导致将次正规操作数视为零。

  此选项对不使用 SSE 或 SSE2 指令集的传统 x86 浮点运算没有影响。

### A.2.22.1 值

-fns 选项可以使用下表中列出的值。

表 A–7 -fns 值

| 值 | 含义 |
|---|---|
| yes | 选择非标准浮点模式 |
| no | 选择标准浮点模式 |

#### 缺省值

如果未指定 -fns，则不自动启用非标准浮点模式。进行标准 IEEE 754 浮点计算（即逐渐下溢）。

如果仅指定了 –fns，则假定 –fns=yes。

#### 示例

在以下示例中，-fast 扩展为多个选项，其中一个是 -fns=yes，即选择非标准浮点模式。后续 -fns=no 选项覆盖初始设置，并选择浮点模式。

```
example% CC foo.cc -fast -fns=no
```

#### 警告

非标准模式启动时，浮点运算可以产生不符合 IEEE 754 标准要求的结果。

如果使用 -fns 选项编译一个例程，应使用 –fns 选项编译该程序的所有例程。否则，可能会产生意外的结果。

只有编译主程序时该选项才有效。

使用 –fns=yes（或 -fns）选项时，如果程序中出现通常由 IEEE 浮点陷阱处理程序管理的浮点错误，则可能会生成警告消息。

#### 另请参见

《数值计算指南》和 ieee_sun(3M) 手册页

## A.2.23 –fprecision=*p*

x86：设置非缺省浮点精度模式。

–fprecision 选项用于设置浮点控制字 (floating-point control word, FPCW) 中的舍入精度模式位。这些位控制基本算术运算（加、减、乘、除和平方根）结果的舍入精度。

### A.2.23.1 值

*p* 必须是下表中列出的值之一。

表 A–8 -fprecision 值

| 值 | 含义 |
|---|---|
| single | 舍入到 IEEE 单精度值。 |
| double | 舍入到 IEEE 双精度值。 |
| extended | 舍入到最大可用精度。 |

如果 *p* 为 single 或 double，该选项会使舍入精度模式在程序开始执行时分别设置为 single 或 double 精度。如果 *p* 是 extended 或未使用 –fprecision 选项，则舍入精度模式保持为 extended 精度。

在 single 精度舍入模式下，结果将舍入到 24 个有效位；在 double 精度舍入模式下，结果将舍入到 53 个有效位。在缺省的 extended 精度模式下，结果将舍入到 64 个有效位。该模式只控制在寄存器中结果的舍入精度，而不影响范围。寄存器中所有的结果都使用了各种已扩展的双精度格式来舍入。不过，存储在内存中的结果既舍入到目标格式的范围也舍入到目标格式的精度。

float 类型的标称精度为 single。long double 类型的标称精度为 extended。

#### 缺省值

如果未指定 –fprecision 选项，舍入精度模式缺省为 extended。

#### 警告

该选项仅在 x86 系统上且仅在编译主程序时才有效，但如果编译 64 位 (-m64) 或 SSE2 启动的 (-xarch=sse2) 处理器，忽略此选项。在 SPARC 系统上也忽略此选项。

## A.2.24 –fround=*r*

启动时设置有效的 IEEE 舍入模式。

此选项设置编译器在评估常量表达式时可以使用的 IEEE 754 舍入模式。此舍入模式是在程序初始化过程中在运行时建立的。

含义与 ieee_flags 子例程的含义相同，可用于更改运行时的模式。

### A.2.24.1 值

*r* 必须是下表中列出的值之一。

表 A–9 -fround 值

| 值 | 含义 |
|---|---|
| nearest | 舍入到最接近的数字并转变为偶数。 |
| tozero | 舍入到零。 |
| negative | 舍入到负无穷大。 |
| positive | 舍入到正无穷大。 |

#### 缺省值

如果未指定 –fround 选项，舍入模式缺省为 -fround=nearest。

#### 警告

如果使用 –fround=*r* 编译一个例程，就必须使用相同的 –fround=*r* 选项编译程序的所有例程。否则，可能会产生意外的结果。

只有编译主程序时该选项才有效。

请注意，使用 –xvector 或 –xlibmopt 进行编译时需要具有缺省的舍入模式。与使用 –xvector 或 –xlibmopt（或同时使用两者）编辑的库链接的程序必须确保缺省舍入生效。

## A.2.25 –fsimple[=*n*]

选择浮点优化首选项。

该选项允许优化器制定关于浮点运算的简化假定。

### A.2.25.1 值

如果存在 *n*，它必须是 0、1 或 2。

表 A–10 -fsimple 值

| 值 | 含义 |
|---|---|
| 0 | 不允许简化假定。保持严格的 IEEE 754 一致性。 |

表 A–10 -fsimple 值 （续）

| 值 | 含义 |
| --- | --- |
| 1 | 允许保守简化。产生的代码与 IEEE 754 不完全一致，但多数程序所产生的数值结果没有更改。<br>在 -fsimple=1 的情况下，优化器可假定：<br>■ 在进程初始化之后，IEEE 754 缺省舍入/捕获模式不发生改变。<br>■ 除产生潜在浮点异常的计算不能删除外，产生不可视结果的计算都可以删除。<br>■ 使用无穷大或 NaNs 作为操作数的计算需要将 NaNs 传送到它们的结果中，即 x*0 可以用零替换。<br>■ 计算不依赖于零的符号。<br>在 -fsimple=1 的情况下，不允许优化器不考虑舍入或异常进行完全优化。具体来讲，运行时将舍入模式保存为常量时，不能用产生不同结果的计算来替换浮点计算。 |
| 2 | 包括 -fsimple=1 的所有功能，还允许可能导致许多程序生成不同的数值结果（由于舍入的更改）的主动浮点优化。例如，允许优化器将指定循环中的所有 x/y 运算替换为 x*z，其中，要保证在循环 z=1/y 中至少对 x/y 求一次值，并且在执行该循环期间已知 y 和 z 的值为常量值。 |

### 缺省值

如果未指定 –fsimple，编译器将使用 -fsimple=0。

如果指定了 -fsimple 但未指定 *n* 值，编译器将使用 -fsimple=1。

### 交互

-fast 隐含了 – fsimple=2。

### 警告

该选项可以破坏 IEEE 754 的一致性。

### 另请参见

-fast

## A.2.26 –fstore

(x86) 强制浮点表达式的精度。

在以下值为真时，该选项使编译器将浮点表达式或函数的值转换为赋值左侧的类型，而不是将该值保留在寄存器中：

- 将表达式或函数分配到变量。
- 表达式被强制转换为较短的浮点类型。

要禁用此选项，请使用 –nofstore 选项。在 SPARC 平台上，–fstore 和 –nofstore 都将被忽略，并且出现一条警告。

### A.2.26.1 警告

由于误差和截断，结果可能会与寄存器值所生成的结果不同。

### 另请参见

–nofstore

## A.2.27 -ftrap=*t*[ ,*t*...]

设置启动时生效的IEEE 陷阱操作模式，但不安装 SIGFPE 处理程序。可以使用 ieee_handler(3M) 或 fex_set_handling(3M) 启用陷阱并同时安装 SIGFPE 处理程序。如果指定多个值，则按从左到右顺序处理列表。

### A.2.27.1 值

*t* 可以为下表中所列出的值之一。

表 A–11 -ftrap 值

| 值 | 含义 |
|---|---|
| [no%]division | 在除以零时自陷。 |
| [no%]inexact | 在结果不精确时自陷。 |
| [no%]invalid | 在操作无效时自陷。 |
| [no%]overflow | 在溢出时自陷。 |
| [no%]underflow | 在下溢时自陷。 |
| %all | 在所有以上内容中自陷。 |
| %none | 不在以上任何内容中自陷。 |
| common | 在无效、除以零和溢出时自陷。 |

注意，选项的 [no%] 形式只用于修改 %all 和 common 值的含义，且必须与其中的一个值一起使用，如以下示例所示。选项自身的 [no%] 形式不会显式导致禁用特定的陷阱。

### 缺省值

如果未指定 –ftrap，则编译器假定 –ftrap=%none。

### 示例

–ftrap=%all,no%inexact 意味着设置除 inexact 以外的所有陷阱。

### 警告

如果使用 –ftrap=*t* 编译一个例程，就应使用相同的 -ftrap=*t* 选项编译程序的所有例程。否则，可能会产生意外的结果。

使用 -ftrap=inexact 陷阱时务必谨慎。只要浮点值不能精确表示，使用 –ftrap=inexact 便会产生自陷。例如，以下语句就会产生这种情况：

```
x = 1.0 / 3.0;
```

只有编译主程序时该选项才有效。请小心使用该选项。如果要启用 IEEE 陷阱，请使用 –ftrap=common。

### 另请参见

ieee_handler(3M) 和 fex_set_handling(3M) 手册页。

## A.2.28 –G

生成动态共享库而不是可执行文件。

缺省情况下，在命令行上指定的所有资源文件都是使用 -xcode=pic13 进行编译的。

在从包含模板且用 -instances=extern 选项编译的文件中生成共享库时，将自动从模板缓存中包含 .o 文件引用的任何模板实例。

如果要通过指定 -G 与其他必须在编译时和链接时指定的编译器选项来创建共享对象，请确保在编译时和与生成的共享对象链接时也指定这些选项。

创建共享对象时，针对 64 位 SPARC 体系结构编译的所有目标文件也必须使用某个显式 -xcode 值进行编译，如第 222 页中的 “A.2.113 –xcode=*a*”中所推荐。

### A.2.28.1 交互

如果未指定 – c（仅编译选项），以下选项将传递给链接程序：

- –dy
- –G
- –R

### 警告

请勿使用 ld -G 生成共享库，而应使用 CC - G。CC 驱动程序会自动将 C++ 所需的多个选项传递给 ld。

使用 -G 选项时，编译器不将任何缺省 -l 选项传递到 ld 选项。如果您要使共享库具有对另一共享库的依赖性，就必须在命令行上传递必需的 -l 选项。例如，如果要使共享库依赖于 libCrun，必须在命令行上传递 -lCrun。

### 另请参见

-dy、-xcode=pic13、–ztext ld(1) 手册页以及第 148 页中的 “14.3 生成动态（共享）库”。

## A.2.29 –g

生成附加的符号表信息，以供使用 dbx(1) 或调试器进行调试以及使用性能分析器 analyzer(1) 进行分析。

指示编译器和链接程序准备进行调试和性能分析的文件或程序。

其任务包括：

- 生成以目标文件和可执行文件的符号表形式表示的详细信息（称为 *stabs*）。
- 生成帮助程序函数，调试器可以调用这些函数来实现其某些功能。
- 如果未指定优化级别，则会禁用函数的内联生成；也就是说，在未指定优化级别时，使用此选项意味着使用了 +d 选项。将 -g 结合 -O 或 -xO 级别不会禁用内联。
- 禁用优化的某些级别。

### A.2.29.1 交互

如果将此选项与 –x0*level*（或其等效选项，如 -O）一起使用，将会获得一些特定的调试信息。有关更多信息，请参见第 249 页中的 “A.2.151 -x0*level*”。

如果使用该选项且优化级别为 -xO4 或更高，编译器会为完全优化提供尽可能多的符号信息。如果使用 -g 且不指定优化级别，将禁用函数调用的内联。（如果随 -g 指定优化级别，会启用内联。）

指定此选项时，除非还指定 -O 或 -xO，否则会自动指定 +d 选项。

要使用性能分析器的完整功能，请使用 -g 选项进行编译。虽然某些性能分析功能不需要使用 -g，但必须使用 -g 进行编译，以便查看注释的源代码、部分函数级别信息以及编译器注释消息。有关更多信息，请参见 analyzer(1) 手册页和性能分析器手册。

使用 -g 生成的注释消息描述编译器在编译程序时进行的优化和变换。使用 er_src(1) 命令来显示与源代码交叉的消息。

### 警告

如果在不同的步骤中编译和链接程序，则在一个步骤中使用 -g 选项而在另一个步骤中不使用该选项不会影响程序的正确性，但会影响调试程序的能力。没有使用 -g（或 -g0）编译但使用 -g（或 -g0）链接的任何模块将不能正常进行调试。请注意，通常必须使用 -g 选项（或 -g0 选项）编译包含函数 main 的模块才能对其进行调试。

### 另请参见

+d、–g0、–xs、analyzer(1) 手册页、er_src(1) 手册页、ld(1) 手册页以及《使用 dbx 调试程序》（介绍了有关 stabs 的详细信息）和性能分析器手册。

## A.2.30 –g0

编译和链接以便进行调试，但不禁用内联。

此选项与 –g 相同，但 +d 处于禁用状态，dbx 无法对内联函数使用**步入**功能。

如果指定 -g0 且优化级别为 -xO3 或更低，编译器会为近乎完全优化提供尽可能多的符号信息。尾部调用优化和后端内联被禁用。

### A.2.30.1 另请参见

+d、-g 和《使用 dbx 调试程序》

## A.2.31 -g3

生成其他调试信息。

–g3 选项与具有附加调试符号表信息的 –g0 相同，允许 dbx 在源代码中显示宏扩展。与使用 –g0 进行编译相比，此附加符号表信息会增大生成的 .o 和可执行文件的大小。

## A.2.32 –H

打印所包含文件的路径名。

在标准错误输出 (stderr) 中，该选项打印当前编译中包含的每个 #include 文件的路径名（每行一个）。

## A.2.33 –h[ ]name

为生成的动态共享库指定名称 *name*。

这是一个链接程序选项，传递给 ld。通常， -h 后面的名称应该与 –o 后面的名称完全相同。–h 和 *name* 之间的空格是可选的。

编译时的加载器将指定名称分配到正在创建的共享动态库中，并将该名称作为库的内部名称记录在库文件中。如果没有 –h*name* 选项，则没有内部名称记录在库文件中。

每个可执行文件都具有所需的共享库文件列表。当运行时链接程序将库链接到可执行文件中时，链接程序将内部名称从库复制到所需共享库文件的列表中。如果没有共享文件的内部名称，链接程序就复制共享库文件的路径。

没有使用 -h 选项生成共享库时，运行时加载器仅查找库的文件名。可以将库替换为具有相同文件名的不同库。如果共享库有内部名称，加载器会在装入文件时检查内部名称。如果内部名称不匹配，加载器不会使用替换文件。

### A.2.33.1 示例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

## A.2.34 –help

与 -xhelp=flags 相同。

## A.2.35 -Ipathname

将 *pathname* 添加到 #include 文件中的搜索路径中。

该选项用于将 *pathname* 添加到在其中搜索具有相对文件名（不以斜杠开头的文件名）的 #include 文件的目录列表。

编译器按以下顺序搜索用引号引住的文件（形式为 #include "foo.h"）：

1. 在包含源代码的目录中
2. 在使用 -I 选项指定的目录（如果有）中
3. 在编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的 include 目录中
4. 在 /usr/include 目录中

编译器按以下顺序搜索用尖括号括住的文件（形式为 #include <foo.h>）：

1. 在使用 -I 选项指定的目录（如果有）中
2. 在编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的 include 目录中
3. 在 /usr/include 目录中

**注 –** 如果此拼写与标准头文件的名称匹配，另请参阅第 117 页中的 “11.7.5 标准头文件实现”。

### A.2.35.1 交互

`-I-` 选项让您可以覆盖缺省的搜索规则。

如果指定了 `-library=no%Cstd`，那么编译器在其搜索路径中就不包括编译器提供的与 C++ 标准库关联的头文件。请参见第 116 页中的 “11.7 替换 C++ 标准库”。

如果未使用 –pti*path*，编译器就会在 –I*pathname* 中查找模板文件。

请使用 –I*pathname* 而不是 –pti*path*。

该选项会累积而不覆盖。

#### 警告

任何时候都不要将编译器安装区域 `/usr/include`、`/lib` 或 `/usr/lib` 指定为搜索目录。

#### 另请参见

`-I-`

## A.2.36 -I-

更改包含文件搜索规则。

对于形式为 `#include "foo.h"` 的 include 文件，按以下顺序搜索目录：

1. 使用 `-I` 选项指定的目录（在 `-I-` 前后）。
2. 编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的目录
3. `/usr/include` 目录

对于 `#include <foo.h>` 形式的 include 文件，按以下顺序搜索目录：

1. 使用 `-I` 选项指定的目录（在 `-I-` 后面）
2. 编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的目录
3. `/usr/include` 目录

**注 –** 如果包括文件的名称与标准头的名称相匹配，另请参阅第 117 页中的 “11.7.5 标准头文件实现”。

### A.2.36.1 示例

以下示例显示了编译 `prog.cc` 时使用 `-I-` 的结果。

```
prog.cc
#include "a.h"
#include <b.h>
#include "c.h"
c.h
#ifndef _C_H_1
#define _C_H_1
int c1;
#endif
inc/a.h
#ifndef _A_H
#define _A_H
#include "c.h"
int a;
#endif
inc/b.h
#ifndef _B_H
#define _B_H
#include <c.h>
int b;
#endif
inc/c.h
#ifndef _C_H_2
#define _C_H_2
int c2;
#endif
```

以下命令显示了针对形式为 `#include "foo.h"` 的包含语句搜索当前目录（包含文件的目录）的缺省行为。在 `inc/a.h` 中处理 `#include "c.h"` 语句时，编译器将 `inc` 子目录中的 `c.h` 头文件包含进来。在 `prog.cc` 中处理 `#include "c.h"` 语句时，编译器将包含 `prog.cc` 的目录中的 `c.h` 文件包含进来。请注意，`-H` 选项指示编译器输出所包含文件的路径。

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

以下命令显示了 `-I-` 选项的效果。编译器处理形式为 `#include "foo.h"` 的语句时，并不先在包含目录中查找，而是按照通过 `-I` 选项指定的目录在命令行上的显示顺序搜索这些目录。在 `inc/a.h` 中处理 `#include "c.h"` 语句时，编译器包含 `./c.h` 头文件而不是 `inc/c.h` 头文件。

```
example% CC -c -I. -I- -Iinc -H prog.cc
inc/a.h
```

```
        ./c.h
inc/b.h
        inc/c.h
./c.h
```

### 交互

命令行上有 `-I-` 时，编译器从不搜索当前目录，除非在 `-I` 指令中显式列出了该目录。甚至是形式为 `#include "foo.h"` 的包含语句，也是这种情况。

### 警告

只有命令行上的第一个 `-I-` 会导致出现所述行为。

任何时候都不要将编译器安装区域 `/usr/include`、`/lib` 或 `/usr/lib` 指定为搜索目录。

## A.2.37 –i

指示链接程序 `ld` 忽略任何 `LD_LIBRARY_PATH` 和 `LD_LIBRARY_PATH_64` 设置。

## A.2.38 -include filename

此选项使编译器处理 *filename* 的方式就相当于其是位于主源文件首行的 `#include` 预处理程序指令。考虑源文件 `t.c`：

```
main()
{
   ...
}
```

如果使用命令 **`cc -include t.h t.c`** 编译 `t.c`，则编译时好像源文件包含以下项：

```
#include "t.h"
main()
{
   ...
}
```

编译器在其中搜索 *filename* 的第一个目录是当前工作目录而不是包含主源文件的目录，这就是显式包括某个文件时的情况。例如，下面的目录结构包含两个名称相同但位置不同的头文件：

```
foo/
   t.c
   t.h
   bar/
     u.c
     t.h
```

如果您的工作目录是 `foo/bar`，并且您使用命令 **`cc ../t.c -include t.h`** 进行编译，则编译器会包括 `foo/bar` 中的 `t.h` 而不是 `foo/` 中的此文件，后者是源文件 `t.c` 内包含 `#include` 指令时的情况。

如果编译器在当前工作目录中找不到使用 `-include` 指定的文件，则会在正常目录路径中搜索该文件。如果您指定多个 `-include` 选项，则文件的包括顺序与它们在命令行中的顺序相同。

## A.2.39 -inline

与 `-xinline` 相同。

## A.2.40 –instances=*a*

控制模板实例的放置和链接。

### A.2.40.1 值

*a* 必须是下表中列出的值之一。

**表 A–12** -instances 值

| 值 | 含义 |
|---|---|
| `extern` | 将全部所需实例放置到链接程序 *comdat* 部分内的模板系统信息库并向其赋予全局链接。（如果系统信息库中的实例过期，就会被重新实例化。）<br>**注意**： 如果在不同的步骤中进行编译和链接，并且在编译步骤中指定了 `-instance=extern`，则还必须在链接步骤中指定该选项。 |
| `explicit` | 将显式实例化的实例放置到当前目标文件中并赋予全局链接。不生成其他任何所需实例。 |
| `global` | 将全部所需的实例放置到当前目标文件中并赋予全局链接。 |
| `semiexplicit` | 将显式实例化的实例放置到当前目标文件中并赋予全局链接。将显式实例所需的全部实例放置到当前目标文件中并赋予全局链接。不生成其他任何所需实例。 |
| `static` | **注意**： `-instances=static` 已过时。您不再需要使用 `-instances=static`，因为 `-instances=global` 现在提供了静态的所有优点而没有其缺点。以前编译器中提供的该选项用于克服此编译器版本中不存在的问题。<br>将全部所需的实例放置到当前目标文件中并赋予静态链接。 |

### 缺省值

如果未指定 –instances，则假定 –instances=global。

### 另请参见

第 86 页中的 “7.2.4 模板实例的放置和链接”

## A.2.41 –instlib= filename

使用该选项可避免在库（共享或静态）和当前对象中生成重复的模板实例。一般来说，如果程序与库共享大量实例，可以尝试使用 -instlib=*filename*，看看编译时间是否会减少。

### A.2.41.1 值

使用 *filename* 参数指定包含可由当前编译生成的模板实例的库。filename 参数必须包含正斜杠 "/" 字符。对于相对于当前目录的路径，请使用点斜杠 "./"。

### 缺省值

-instlib=*filename* 选项没有缺省值，只有在指定后才能使用。该选项可被多次指定和累积。

### 示例

假定 libfoo.a 和 libbar.so 库可对与源文件 a.cc 共享的大量模板实例进行实例化。添加 -instlib=*filename* 并指定库可通过避免冗余有利于减少编译时间。

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

### 交互

使用 -g 进行编译时，如果使用 -instlib=*file* 指定的库没有使用 -g 编译，那么这些模板实例不可调试。解决方法是避免在使用 -g 时使用 -instlib=*file*。

### 警告

如果使用 -instlib 指定库，就必须与该库链接。

### 另请参见

-template、-instances 和 -pti

## A.2.42 –KPIC

SPARC：（已过时）与 `–xcode=pic32` 相同。

x86：与 `–Kpic` 相同。

生成共享库时使用该选项编译源文件。对全局数据的每个引用都生成为全局偏移表中指针的非关联化。每个函数调用都是通过过程链接表在程序计数器 (program counter, pc) 相对寻址模式下生成的。

## A.2.43 –Kpic

SPARC：（已过时）与 `–xcode=pic13` 相同。

x86：使用与位置无关的代码进行编译。

生成共享库时使用该选项编译源文件。对全局数据的每个引用都生成为全局偏移表中指针的非关联化。每个函数调用都是通过过程链接表在程序计数器 (program counter, pc) 相对寻址模式下生成的。

## A.2.44 –keeptmp

保留编译时创建的临时文件。

该选项与 `–verbose=diags` 一起使用，对调试很有用。

### A.2.44.1 另请参见

`–v, –verbose`

## A.2.45 –L路径

将 *path* 添加到要在其中搜索库的目录列表。

该选项传递给 `ld`。搜索顺序是先搜索通过 *path* 指定的目录，再搜索编译器提供的目录。

### A.2.45.1 交互

该选项会累积而不覆盖。

#### 警告

任何时候都不要将编译器安装区域 `/usr/include`、`/lib` 或 `/usr/lib` 指定为搜索目录。

## A.2.46 –l*lib*

将库 lib*lib*.a 或 lib*lib*.so 添加到链接程序的搜索库列表。

该选项传递给 ld。库的名称通常为 lib*lib*.a 或 lib*lib*.so，其中 lib 和 .a 或 .so 部分是必需的。应该使用此选项指定 *lib* 部分。您可以根据需要在单个命令行上放置任意数量的库。将按照使用 –L*dir* 指定的顺序对库进行搜索。

请在目标文件名之后使用该选项。

### A.2.46.1 交互

该选项会累积而不覆盖。

将 -l*x* 放在源文件和目标文件列表**之后**，这样可以确保按正确顺序搜索库。

#### 警告

为了确保正确的库链接顺序，必须使用 -mt（而不是 -lthread）与 libthread 链接。

#### 另请参见

–L*dir* 和 -mt

## A.2.47 –libmieee

与 –xlibmieee 相同。

## A.2.48 –libmil

与 -xlibmil 相同。

## A.2.49 -library=/[ ,/...]

将指定的 CC 提供的库加入编译和链接操作中。

### A.2.49.1 值

关键字 *l* 必须是下表中列出的值之一。no% 前缀禁用关联的选项。

表 A–13 -library 值

| 值 | 含义 |
|---|---|
| [no%]f77 | 已过时。改用 -xlang=f77。 |
| [no%]f90 | 已过时。改用 -xlang=f90。 |
| [no%]f95 | 已过时。改用 -xlang=f95。 |
| [no%]rwtools7 | 使用传统 iostream Tools.h++ 版本 7。 |
| [no%]rwtools7_dbg | 使用支持调试的传统 iostream Tools.h++ 版本 7。 |
| [no%]rwtools7_std | 使用标准 iostream Tools.h++ 版本 7。 |
| [no%]rwtools7_std_dbg | 使用支持调试的标准 iostream Tools.h++ 版本 7。 |
| [no%]interval | 已过时。不要使用。使用 -xia。 |
| [no%]iostream | 使用传统 iostream 库 libiostream。 |
| [no%]Cstd | 使用 C++ 标准库 libCstd。包括编译器提供的 C++ 标准库头文件。 |
| [no%]Crun | 使用 C++ 运行时库 libCrun。 |
| [no%]gc | 使用 libgc 垃圾收集。 |
| [no%]stlport4 | 使用 STLport 的标准库实现版本 4.5.3，而不是缺省的 libCstd。关于使用 STLport 实现的更多信息，请参见 第 123 页中的 “12.2 STLport”。 |
| [no%]stlport4_dbg | 使用 STLport 的支持调试的库。 |
| [no%]sunperf | 使用 Sun 性能库。 |
| [no%]stdcxx4 | 在 Solaris 中使用 Apache stdcxx 版本 4 C++ 标准库，而不是使用缺省的 libCstd。此选项还会隐式设置 -mt 选项。stdcxx 库需要使用多线程模式。必须在整个应用程序的每个编译和链接命令中一致使用此选项。用 -library=stdcxx4 编译的代码与用缺省的 -library=Cstd 或可选的 -library=stlport4 编译的代码不能用在同一程序中。 |
| %none | 仅使用 C++ 库 libCrun。 |

## A.2.49.2 缺省值

- **标准模式（缺省模式）**
  - libCstd 库总是包括在内，除非使用 -library=%none、-library=no%Cstd、–library=stdcxx4 或 -library=stlport4 明确将其排除。

- libCrun 库总是包括在内，除非使用 -library=no%Crun 明确将其排除。

始终会包括 libm 库，即使指定了 -library=%none。

### A.2.49.3 示例

要在标准模式下没有任何 C++ 库（libCrun 除外）的情况下进行链接，请使用：

```
example% CC -library=%none
```

要在标准模式下将传统 iostream Rogue tools.h++ 库包含进来，请使用：

```
example% CC –library=rwtools7,iostream
```

要在标准模式下将标准 iostream Rogue Wave tools.h++ 库包含进来，请使用：

```
example% CC -library=rwtools7_std
```

### A.2.49.4 交互

如果使用 -library 指定了库，则在编译期间会设置适当的 –I 路径。在链接期间会设置适当的 –L、–Y P、–R 路径和 –l 选项。

该选项会累积而不覆盖。

在使用区间运算库时，必须包括以下库之一： libC、libCstd 或 libiostream。

使用 -library 选项可确保针对指定库的 -l 选项按正确顺序处理。例如，对于 -library=rwtools7,iostream 和 -library=iostream,rwtools7， -l 选项都是按照 -lrwtool -liostream 顺序传递给 ld。

指定的库在系统支持库链接之前链接。

对于 –library=stdcxx4，必须在 Oracle Solarise 平台上的 /usr/include 和 /usr/lib 中安装 stdcxx Apache 库。

不能在同一个命令行上使用 -library=sunperf 和 -xlic_lib=sunperf。

在任何命令行中，最多只能使用 -library=stlport4、-library=stdcxx4 和 -library=Cstd 选项之一。

每次只能使用一个 Rogue Wave 工具库，而且不能将任何 Rogue Wave 工具库与 -library=stlport4 或 -library=stdcxx4 一起使用。

在标准模式（缺省模式）下包含传统 iostream Rogue Wave 工具库时，必须也要包含 libiostream（有关其他信息，请参见《C++ 迁移指南》。只能在标准模式下使用标准 iostream Rogue Wave 工具库。以下命令示例显示了有效使用和无效使用 Rogue Wave tools.h++ 库选项的情况。

```
% CC -library=rwtools7,iostream foo.cc        <-- valid, classic iostreams
% CC -library=rwtools7 foo.cc                 <-- invalid

% CC -library=rwtools7_std foo.cc             <-- valid, standard iostreams
% CC -library=rwtools7_std,iostream foo.cc    <-- invalid
```

如果同时包含 `libCstd` 和 `libiostream`，必须小心，不要在程序中同时使用新旧格式的 iostream（例如，`cout` 和 `std::cout`）访问同一个文件。如果从传统和标准 iostream 代码访问同一文件，那么在相同的程序中混合标准 iostream 和传统 iostream 可能会出现问题。

不链接 `Crun` 或任何 `Cstd` 库或 `stlport4` 库的标准模式程序无法使用 C++ 语言的所有功能。

如果指定了 `-xnolib`，则忽略 `-library`。

### A.2.49.5 警告

如果在不同的步骤中进行编译和链接，那么必须在链接命令中使用在编译命令中使用的那组 `-library` 选项。

`stlport4`、`Cstd` 和 `iostream` 库都提供了自己的 I/O 流实现。如果使用 `-library` 选项指定其中多个库，会导致出现不确定的程序行为。关于使用 STLport 实现的更多信息，请参见 第 123 页中的 “12.2 STLport”。

库的集合不稳定，会因不同的发行版本而变化。

### A.2.49.6 另请参见

请参见第 113 页中的 “11.4.1.1 有关传统 iostream 和传统 RogueWave 工具的说明”

`–I`、`–l`、`–R`、`–staticlib`、`-xia`、`-xlang`、`–xnolib`、第 119 页中的 “忠告：”、第 124 页中的 “12.2.1 重新分发和支持的 STLport 库”以及《Tools.h++ User's Guide》。

有关使用 `-library=no%cstd` 选项以便能够使用自己的 C++ 标准库的信息，请参见第 116 页中的 “11.7 替换 C++ 标准库”。

## A.2.50 -m32|-m64

指定编译的二进制对象的内存模型。

使用 `-m32` 来创建 32 位可执行文件和共享库。使用 `-m64` 来创建 64 位可执行文件和共享库。

在所有 Oracle Solaris 平台和不支持 64 位的 Linux 平台上，ILP32 内存模型（32 位 int、long、pointer 数据类型）是缺省值。在启用了 64 位的 Linux 平台上缺省为 LP64 内存模型（64 位 long 和指针数据类型）。`-m64` 仅允许在支持 LP64 模型的平台上使用。

使用 -m32 编译的目标文件或库无法与使用 -m64 编译的目标文件或库链接。

使用 -m32|-m64 编译的模块必须还使用 -m32 |-m64 进行链接。有关在编译时和链接时都必须指定的完整编译器选项列表，请参见第 43 页中的 “3.3.3 编译时选项和链接时选项”。

64 位平台上 (-m64) 使用了大量静态数据的应用程序可能还需要 -xmodel=medium。请注意，部分 Linux 平台不支持中等模型。

请注意，在早期的编译器发行版中，由 -xarch 选项中选择的指令集来指定内存模型（ILP32 或 LP64）。从 Solaris Studio 12 编译器开始，在大多数平台上，创建 64 位对象的正确方式是将 -m64 添加到命令行中。

在 Oracle Solaris 上，-m32 为缺省值。在支持 64 位程序的 Linux 系统上，-m64 -xarch=sse2 是缺省选项。

### A.2.50.1 另请参见

-xarch。

## A.2.51 -mc

从目标文件的 ELF .comment 部分中删除重复的字符串。使用 -mc 选项时，会调用 mcs -c 命令。有关详细信息，请参见 mcs(1) 手册页。

## A.2.52 –misalign

SPARC：已过时。不应使用该选项。请改用 –xmemalign=2i。

## A.2.53 -mr[, string]

从目标文件的 .comment 部分中删除所有字符串，如果提供了 *string*，则将 *string* 放入该部分。如果字符串包含空格，就必须使用引号将该字符串括入。如果使用此选项，会调用命令 mcs -d [-a *string*]。

## A.2.54 -mt[={yes |no}]

使用此选项，可以通过 Oracle Solaris 线程或 POSIX 线程 API 编译和链接多线程代码。-mt=yes 选项确保库以正确的顺序链接。

此选项将 -D_REENTRANT 传递给预处理程序。

要使用 Oracle Solaris 线程，请包括 thread.h 头文件并使用 –mt=yes 选项进行编译。要在 Oracle Solaris 平台上使用 POSIX 线程，应将 pthread.h 头文件包含进来并使用 -mt=yes -lpthread 选项进行编译。

在 Linux 平台上，只有 POSIX 线程 API 可用。（Linux 平台上没有 `libthread`）。因此，Linux 平台上的 `-mt=yes` 会添加 `-lpthread`，而不是 `-lthread`。要在 Linux 平台上使用 POSIX 线程，应使用 `-mt=yes` 进行编译。

请注意，当使用 `-G` 进行编译时，`-lthread` 和 `-lpthread` 均不会自动包括在 `-mt=yes` 内。生成共享库时，您需要显式列出这些库。

`-xopenmp` 选项（用于使用 OpenMP 共享内存并行化 API）自动包含 `-mt=yes`。

如果使用 `-mt=yes` 进行编译并在单独的步骤中进行链接，则除了在编译步骤中外，还必须在链接步骤中使用 `mt=yes` 选项。如果使用 `-mt` 编译和链接一个翻译单元，则必须使用 `-mt` 编译和链接程序的所有单元。

`-mt=yes` 是编译器的缺省行为。如果不需要此行为，请使用 `-mt=no` 进行编译。

选项 `-mt` 与 `-mt=yes` 等效。

### A.2.54.1 另请参见

–xnolib、Oracle Solaris《多线程编程指南》和《链接程序和库指南》

## A.2.55 –native

与 –xtarget=native 相同。

## A.2.56 –noex

与 –features=no%except 相同。

## A.2.57 –nofstore

x86：在命令行中取消 `-fstore`。

取消强制表达式具有由 `-fstore` 调用的目标变量的精度。`-nofstore` 是由 `-fast` 调用的。`-fstore` 是惯用的缺省值。

### A.2.57.1 另请参见

–fstore

## A.2.58 –nolib

与 –xnolib 相同。

## A.2.59 –nolibmil

与 –xnolibmil 相同。

## A.2.60 –norunpath

不将共享库的运行时搜索路径生成到可执行文件中。

如果可执行文件使用共享库，编译器通常会生成将运行时链接程序指向这些共享库的路径。为此，编译器会将 –R 选项传递给 ld。路径取决于安装编译器的目录。

如果客户可能会为程序引用的共享库使用不同的路径，在生成提供给这类客户的可执行文件时建议使用该选项。请参阅第 115 页中的 “11.6 使用共享库”。

### A.2.60.1 交互

如果使用编译器安装区域下的任何共享库，并且还使用 –norunpath，那么就应该在链接时使用 –R 选项或在运行时设置环境变量 LD_LIBRARY_PATH 来指定共享库的位置。这样做使运行时链接程序可以找到共享库。

## A.2.61 –O

-O 宏扩展到了 -xO3。（某些以前的发行版从 –O 扩展到 –xO2 ）。

这种特殊变化会提高运行时性能。但是，对于依赖于被自动视为 volatile 的所有变量的程序， -xO3 可能不适用。可能做此假定的典型程序包括设备驱动程序，以及实现其自己的同步基元的旧版多线程应用程序。解决方法是用 -xO2 而不是 -O 进行编译。

## A.2.62 –Olevel

与 –x*Olevel* 相同。

## A.2.63 –o filename

将输出文件或可执行文件的名称设置为 *filename*。

### A.2.63.1 交互

编译器必须存储模板实例时，它将模板实例存储在输出文件目录中的模板系统信息库中。例如，以下命令将目标文件写入 ./sub/a.o 并将模板实例写入包含在 ./sub/SunWS_cache 中的系统信息库。

```
example% CC -instances=extern -o sub/a.o a.cc
```

编译器从对应于编译器读取的目标文件的模板系统信息库读取。例如，以下命令从 `./sub1/SunWS_Cache` 和 `./sub2/SunWS_cache` 进行读取，并且在必要时向 `./SunWS_cache` 进行写入。

```
example% CC -instances=extern sub1/a.o sub2/b.o
```

有关更多信息，请参见第 89 页中的 “7.4 模板系统信息库”。

### 警告

*filename* 必须有与编译所生成文件的类型对应的适当后缀。当与 `-c` 一起使用时，*filename* 指定目标 `.o` 目标文件；当与 `-G` 一起指定时，它指定目标 `.so` 库文件。此选项及其参数将传递给 `ld`。

*filename* 不能与源文件是同一文件，因为 `CC` 驱动程序不覆盖源文件。

## A.2.64 +p

忽略非标准预处理程序声明。

### A.2.64.1 缺省值

如果没有 `+p`，则编译器识别非标准预处理程序声明。

### 交互

如果使用了 `+p`，就不定义下列宏：

- `sun`
- `unix`
- `sparc`
- `i386`

## A.2.65 –P

仅预处理源代码，但不编译。（输出文件的后缀为 `.i`。）

该选项不在输出中包括预处理程序类型的行号信息。

### A.2.65.1 另请参见

`–E`

## A.2.66 –p

已过时，请参见第 258 页中的 “A.2.159 –xpg”。

## A.2.67 –pentium

x86： 替换为 `–xtarget=pentium`。

## A.2.68 –pg

已过时。使用 `–xpg`。

## A.2.69 -PIC

SPARC： 与 `–xcode=pic32` 相同。

x86： 与 `–Kpic` 相同。

## A.2.70 –pic

SPARC： 与 `–xcode=pic13` 相同。

x86： 与 `-Kpic` 相同。

## A.2.71 –pta

与 `–template=wholeclass` 相同。

## A.2.72 –ptipath

为模板源文件指定附加搜索目录。

该选项可替换 `–I`*pathname* 设置的正常搜索路径。如果使用了 `-pti`*path* 选项，编译器将在该路径上查找模板定义文件，并忽略 `–I`*pathname* 选项。

如果使用 `–I`*pathname* 选项而非 `–pti`*path*，可以减少混淆情况。

### A.2.72.1 交互

该选项会累积而不覆盖。

### A.2.72.2 另请参见

–I*pathname* 和第 91 页中的 “7.5.2 定义搜索路径”

## A.2.73 –pto

与 `–instances=static` 相同。

## A.2.74 –ptv

与 `–verbose=template` 相同。

## A.2.75 –Qoption phase option[,option...]

将 *option* 传递到编译**阶段**。

要传递多个选项，按照逗号分隔列表的顺序指定它们。可以对使用 `-Qoption` 传递给组件的选项进行重新排序。驱动程序识别的选项将按正确顺序排列。对于驱动程序已识别的选项，请勿使用 `-Qoption`。例如，C++ 编译器识别用于链接程序 (`ld`) 的 `-z` 选项。如果发出类似于以下示例的命令，`-z` 选项将按顺序传递给链接程序。

```
CC -G -zallextract mylib.a -zdefaultextract ... // correct
```

但是，如果如下例所示来指定命令，`-z` 选项可能会重新排序，因而得到的结果可能不正确。

```
CC -G -Qoption ld -zallextract mylib.a -Qoption ld -zdefaultextract ... // error
```

### A.2.75.1 值

*phase* 必须是下表中列出的值之一。

**表 A–14** -Qoption 值

| SPARC | x86 |
|---|---|
| `ccfe` | `ccfe` |
| `iropt` | `iropt` |
| `cg` | `ube` |
| `CClink` | `CClink` |
| `ld` | `ld` |

### A.2.75.2 示例

在以下命令中，当 CC 驱动程序调用 ld 时，–Qoption 会将 –i 和 –m 选项传递给 ld。

```
example% CC -Qoption ld -i,-m test.c
```

### A.2.75.3 警告

请注意避免无法预料的结果。例如，以下选项序列：

```
-Qoption ccfe -features=bool,iddollar
```

被解释为：

```
-Qoption ccfe -features=bool -Qoption ccfe iddollar
```

正确的用法为

```
-Qoption ccfe -features=bool,-features=iddollar
```

这些功能不需要 –Qoption，仅用作示例。

## A.2.76 –qoption phase option

与 –Qoption 相同。

## A.2.77 –qp

与 –p 相同。

## A.2.78 –Qproduce sourcetype

使 CC 驱动程序生成类型为 *sourcetype* 的输出。

*Sourcetype* 后缀是在下表中定义的：

表 A–15 -Qproduce 值

| 后缀 | 含义 |
|---|---|
| .i | 来自 ccfe 的预处理 C++ 源代码 |
| .o | 生成的目标代码 |
| .s | 来自 cg 的汇编程序源代码 |

## A.2.79 –qproduce sourcetype

与 –Qproduce 相同。

## A.2.80 –Rpathname[ :pathname…]

将动态库搜索路径生成到可执行文件中。

该选项传递给 ld。

### A.2.80.1 缺省值

如果没有 -R 选项，则在输出对象中记录且传递给运行时链接程序的库搜索路径取决于 -xarch 选项指定的目标体系结构指令。未提供 -xarch 时，将假定 -xarch=generic。

检查 –dryrun 的输出和传递给链接程序 ld 的 –R 选项，以查看编译器假定的缺省路径。

### A.2.80.2 交互

该选项会累积而不覆盖。

如果定义了环境变量 LD_RUN_PATH 且指定了 –R 选项，则扫描 –R 指定的路径而忽略 LD_RUN_PATH 指定的路径。

### A.2.80.3 另请参见

–norunpath 和《链接程序和库指南》。

## A.2.81 –S

编译并仅生成汇编代码。

该选项使 CC 驱动程序编译程序并输出汇编源文件，但不汇编程序。汇编源文件名称的后缀为 .s。

## A.2.82 –s

从可执行文件中删除符号表。

该选项从输出可执行文件中删除所有符号信息。该选项传递给 ld。

## A.2.83 -staticlib=l[ ,l…]

指示要静态链接由 -library 选项（包括其缺省值）、-xlang 选项和 -xia 选项指定的 C++ 库。

### A.2.83.1 值

*l* 必须是下表中列出的值之一。

表 A–16 -staticlib 值

| 值 | 含义 |
|---|---|
| [no%]*library* | 静态链接 *library*。*library* 的有效值包括了 -library 的全部有效值（除 %all 和 %none 之外）、-xlang 的全部有效值以及 interval（要与 -xia 结合使用）。 |
| %all | 静态链接 -library 选项中指定的所有库、-xlang 选项中指定的所有库以及（如果在命令行上指定了 -xia）区间库。 |
| %none | 不静态链接在 -library 选项和 -xlang 选项中指定的库。如果在命令行上指定了 -xia，则不静态链接区间库。 |

### A.2.83.2 缺省值

如果没有指定 –staticlib，则假定 –staticlib=%none。

### A.2.83.3 示例

以下命令静态链接 libCrun，因为 Crun 是 –library 的缺省值：

```
example% CC –staticlib=Crun (correct)
```

但以下命令并不链接 libgc，因为只有使用 -library 选项显式指定才链接 libgc：

```
example% CC –staticlib=gc (incorrect)
```

要静态链接 libgc，请使用以下命令：

```
example% CC -library=gc -staticlib=gc (correct)
```

以下命令会动态链接 librwtool 库。因为 librwtool 不是缺省库且未使用 -library 选项选择它，因此 -staticlib 不起作用：

```
example% CC -lrwtool -library=iostream \
-staticlib=rwtools7 (incorrect)
```

以下命令静态链接 librwtool 库：

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 (correct)
```

以下命令将动态链接 Sun 性能库，因为 -library=sunperf 必须与 -staticlib=sunperf 结合使用，-staticlib 选项才能对这些库的链接有效：

```
example% CC -xlic_lib=sunperf -staticlib=sunperf (incorrect)
```

此命令静态链接 Sun 性能库：

```
example% CC -library=sunperf -staticlib=sunperf (correct)
```

### A.2.83.4 交互

该选项会累积而不覆盖。

除缺省情况下隐式选择的 C++ 库之外，`-staticlib` 选项仅对使用 `-xia` 选项、`-xlang` 选项以及 `-library` 选项显式选择的 C++ 库有效。`Cstd` 和 `Crun` 是缺省选择的。

### A.2.83.5 警告

*library* 的允许值集合不确定，会随发行版的不同而异。

在 Oracle Solaris 平台上，系统库不可用作静态库。

### A.2.83.6 另请参见

`-library` 和第 114 页中的 “11.5 静态链接标准库”

## A.2.84 -sync_stdio=[yes| no]

可在运行时性能因 C++ iostream 和 C stdio 之间的同步而降低时使用此选项。仅当您在相同的程序中使用 iostream 写入 cout 以及使用 stdio 写入 stdout 时，才需要同步。C++ 标准要求同步，因此缺省情况下 C++ 编译器打开同步。但是，不使用同步时，应用程序性能通常更佳。如果您的程序既不写入 cout 也不写入 stdout，则可以使用选项 `-sync_stdio=no` 关闭同步。

### A.2.84.1 缺省值

如果未指定 `-sync_stdio`，编译器会将其设置为 `-sync_stdio=yes`。

### A.2.84.2 示例

请看以下示例：

```
#include <stdio.h>
#include <iostream>
int main()
{
   std::cout << "Hello ";
   printf("beautiful ");
   std::cout << "world!";
   printf("\n");
}
```

使用同步时，程序自行打印在一行中

```
Hello beautiful world!
:
```

不使用同步时，输出会变得杂乱。

### A.2.84.3 警告

此选项仅对链接可执行文件有效，对库无效。

## A.2.85 –temp=*path*

为临时文件定义目录。

该选项设置目录的路径名称，用于存储在编译过程中生成的临时文件。对于 `-temp` 设置的值和 `TMPDIR` 值，编译器优先采用前者。

### A.2.85.1 另请参见

`–keeptmp`

## A.2.86 –template=opt[,opt...]

启用/禁用各种模板选项。

### A.2.86.1 值

*opt* 必须是下表中列出的值之一。

**表 A–17** -template 值

| 值 | 含义 |
|---|---|
| `[no%]extdef` | 在独立的源文件中搜索模板定义。编译器使用 `no%extdef` 预定义 `_TEMPLATE_NO_EXTDEF`。 |
| `[no%]geninlinefuncs` | 为显式实例化的类模板生成未引用的内联成员函数。 |
| `[no%]wholeclass` | 实例化整个模板类，而不仅仅是所使用的这些函数。您必须至少引用一个类成员。否则，编译器不会实例化类的任何成员。 |

### A.2.86.2 缺省值

如果未指定 `-template` 选项，则假定 `-template=no%wholeclass,extdef`。

### A.2.86.3 示例

请考虑以下代码：

```
example% cat Example.cc
     template <class T> struct S {
               void imf() {}
               static void smf() {}
     };

     template class S <int>;

     int main() {
     }
example%
```

指定了 -template=geninlinefuncs 时，即使在程序中没有调用 S 的两个成员函数，也会在目标文件中生成它们。

```
example% CC -c -template=geninlinefuncs Example.cc
example% nm -C Example.o

Example.o:

[Index] Value Size Type  Bind  Other Shndx Name
[5]       0   0    NOTY  GLOB  0     ABS   __fsr_init_value
[1]       0   0    FILE  LOCL  0     ABS   b.c
[4]      16   32   FUNC  GLOB  0     2     main
[3]     104   24   FUNC  LOCL  0     2     void S<int>::imf()
                                             [__1cBS4Ci_Dimf6M_v_]
[2]     64    20   FUNC  LOCL  0     2     void S<int>::smf()
                                             [__1cBS4Ci_Dsmf6F_v_]
```

### A.2.86.4 另请参见

第 85 页中的 “7.2.2 整个类实例化”和第 90 页中的 “7.5 模板定义搜索”

## A.2.87 –time

与 –xtime 相同。

## A.2.88 -traceback[={ %none|common|signals_list}]

如果执行中出现严重错误，将发出堆栈跟踪。

当程序生成某些信号时，-traceback 选项会导致可执行文件向 stderr 发出堆栈跟踪、转储信息并退出。如果多个线程都生成一个信号，则只为第一个生成堆栈跟踪。

要使用回溯，请在链接时将 -traceback 选项添加到编译器命令行中。编译时也接受该选项，除非生成可执行二进制文件，否则将忽略此选项。不要将 -traceback 和 -G 一起使用来创建共享库。

表 A–18 -traceback 选项

| 选项 | 含义 |
| --- | --- |
| common | 指定应在出现以下任意一组常见信号时发出堆栈跟踪：sigill、sigfpe、sigbus、sigsegv 或 sigabrt。 |
| *signals_list* | 指定应生成堆栈跟踪的信号名称的逗号分隔列表，采用小写形式。可以捕捉以下信号（导致生成信息转储文件的信号）：sigquit、sigill、sigtrap、sigabrt、sigemt、sigfpe、sigbus、sigsegv、sigsys、sigxcpu、sigxfsz。<br>可以在上述任一信号前加上 no% 以禁用信号缓存。<br>例如：如果发生 sigsegv 或 sigfpe，**-traceback=sigsegv,sigfpe** 将生成堆栈跟踪和信息转储。 |
| %none 或 none | 禁用回溯 |

如果不指定该选项，则缺省值为 -traceback=%none

只使用 -traceback 而不使用 = 符号表示 -traceback=common

注意：如果不希望进行信息转储，可以使用以下命令将信息转储文件大小限制设置为零：

```
% limit coredumpsize 0
```

-traceback 选项不影响运行时性能。

## A.2.89 –U*name*

删除预处理程序符号 *name* 的初始定义。

该选项会删除在命令行上通过 -D（包括 CC 驱动程序隐式放在命令行上的选项）创建的宏符号 *name* 的所有初始定义。该选项对任何其他预定义的宏和源文件中的宏定义都没有影响。

要查看 CC 驱动程序放在命令行上的 -D 选项，请将 -dryrun 选项添加到命令行上。

### A.2.89.1 示例

以下命令取消预定义符号 __sun 的定义。foo.cc 中的预处理程序语句（例如 #ifdef(__sun)）会知道该符号已取消定义。

```
example% CC -U__sun foo.cc
```

### A.2.89.2 交互

可以在命令行上指定多个 -U 选项。

所有 -U 选项都在出现的任何 -D 选项之后处理。也就是说，如果在命令行上为 *-D* 和 -U 指定了相同的 name，则 *name* 是未定义的，而不管这些选项出现的顺序如何。

### A.2.89.3 另请参见

-D

## A.2.90 –unroll=*n*

与 –xunroll=*n* 相同。

## A.2.91 –V

与 –verbose=version 相同。

## A.2.92 –v

与 –verbose=diags 相同。

## A.2.93 –verbose=v[,v...]

控制编译器详细程度。

### A.2.93.1 值

*v* 必须是下表中列出的值之一。no% 前缀禁用关联的选项。

表 A–19 -verbose 值

| 值 | 含义 |
|---|---|
| [no%]diags | 为每个编译传递输出命令行。 |
| [no%]template | 打开模板实例 verbose 模式（有时称为“检验”模式）。verbose 模式显示编译过程中出现的每个实例阶段。 |
| [no%]version | 指示 CC 驱动程序输出它调用的程序的名称和版本号。 |
| %all | 调用所有其他选项。 |
| %none | -verbose=%none 与 -verbose=no%template,no%diags,no%version 相同。 |

### 缺省值

如果未指定 –verbose，则假定 –verbose=%none。

### 交互

该选项会累积而不覆盖。

## A.2.94 -Wc ,arg

将参数 *arg* 传递给指定的组件 *c*。

前后参数之间只能用逗号分隔。所有 `-W` 参数均在其余的命令行参数之后进行传递。要在参数中包括逗号，请在紧靠逗号之前使用转义符 \（反斜杠）。所有 `-W` 参数均在常规命令行参数之后进行传递。

例如， `-Wa,-o,objfile` 按该顺序将 `-o` 和 `objfile` 传递给汇编程序。此外， `-Wl,-I,`*name* 将导致链接阶段覆盖动态链接程序的缺省名称 `/usr/lib/ld.so.1`。

这些参数相对于其他指定的命令行选项传递给工具的顺序可能会更改后续的编译器发行版。

下表列出了 *c* 的可能值。

**表 A–20** -W 标志

| 标志 | 含义 |
|---|---|
| `a` | 汇编程序：(`fbe`); (`gas`) |
| `c` | C++ 代码生成器：(`cg`) *(SPARC)* |
| `d` | CC 驱动程序 |
| `l` | 链接编辑器 (`ld`) |
| `m` | mcs |
| `O`（大写的 o） | 过程间优化器 |
| `o`（小写字母 o） | 后优化器 |
| `p` | 预处理程序 (`cpp`) |
| `0` (Zero) | 编译器 (`ccfe`) |
| `2` | 优化器：(`iropt`) |

注意：不能使用 `-Wd` 将 `CC` 选项传递给 C++ 编译器。

## A.2.95 +w

识别出可能会产生意外后果的代码。使用 +w 选项时，如果函数过大而无法内联或未使用声明的程序元素，就不再生成警告。这些警告不指定源代码中的真正问题，因此不适合某些开发环境。从 +w 中删除这些警告就可以在这些环境下更主动地使用 +w。在 +w2 选项中仍可以使用这些警告。

该选项生成在下列方面有问题的构造的更多相关警告：

- 不可移植
- 可能出错
- 低效

### A.2.95.1 缺省值

如果未指定 +w，则编译器发出有关极可能是问题的构造的警告。

### A.2.95.2 另请参见

–w 和 +w2

## A.2.96 +w2

发出 +w 发出的所有警告以及可能无害但可能降低程序最大可移植性的技术违规的警告。

+w2 选项不再发出关于在系统头文件中使用与实现相关的构造方面的警告。因为系统头文件是实现，所以发出警告是不合适的。从 +w2 删除这些警告可以更主动地使用该选项。

### A.2.96.1 另请参见

+w

## A.2.97 –w

抑制大部分警告消息。

该选项使编译器**不**输出警告消息。但不能抑制某些警告（尤其是有关严重记时错误的警告）。

### A.2.97.1 另请参见

+w

## A.2.98 -Xlinker arg

将 *arg* 传递给链接程序 ld(1)。与 –z *arg* 等效

## A.2.99 –Xm

与 –features=iddollar 相同。

## A.2.100 -xaddr32

（**仅限** *Solaris x86/x64*）-xaddr32=yes 编译标志将生成的可执行文件或共享对象限定于 32 位地址空间。

以这种方式编译的可执行文件会导致创建限定为 32 位地址空间的进程。

指定了 -xaddr32=no 时，将生成普通的 64 位二进制文件。

如果未指定 -xaddr32 选项，则假定 -xaddr32=no。

如果仅指定了 -xaddr32，则假定 -xaddr32=yes。

此选项仅适用于 -m64 编译，并且仅在支持 SF1_SUNW_ADDR32 软件功能的 Oracle Solaris 平台上适用。由于 Linux 内核不支持地址空间限制，此选项在 Linux 上不可用。

链接时，如果单个目标文件是使用 -xaddr32=yes 编译的，则假定整个输出文件是使用 -xaddr32=yes 编译的。

限定为 32 位地址空间的共享对象必须由在受限的 32 位模式地址空间内执行的进程装入。

有关更多信息，请参阅《**链接程序和库指南**》中的 SF1_SUNW_ADDR32 软件功能定义。

## A.2.101 -xalias_level[= n]

指定以下命令时，C++ 编译器可以执行基于类型的别名分析和优化：

-xalias_level[=*n*]

其中 *n* 是 any、simple 或 compatible。

### A.2.101.1 -xalias_level=any

在此分析级别上，编译器假定任何类型都可以为其他类型起别名。不过尽管只是假定，但还是可以执行某些优化。

### A.2.101.2 -xalias_level=simple

编译器假定简单的类型没有别名。存储对象必须是属于以下简单类型之一的动态类型：

`char`、`signed char`、`unsigned char` `wchar_t` 数据指针类型
`short int`、`unsigned short int`、`int` `unsigned int` 函数指针类型
`long int`、`unsigned long int`、`long long int`、`unsigned long long int` 数据成员指针类型
`float`、`double` `long double` 枚举类型函数成员指针类型

存储对象只能通过以下类型的左值访问：

- 对象的动态类型
- 对象动态类型的 `constant` 或 `volatile` 限定版本，与对象动态类型相对应的带符号或不带符号的类型
- 与动态类型对象的 `constant` 或 `volatile` 限定版本对应的带符号或无符号类型
- 在其成员（包括递归的子集成员或包含的联合）中包括上述类型的聚集或联合类型
- char 或 unsigned char type。

### A.2.101.3 -xalias_level=compatible

编译器假定布局不兼容类型没有别名。存储对象只能通过以下类型的左值访问：

- 对象的动态类型
- 对象动态类型的 `constant` 或 `volatile` 限定版本，与对象动态类型相对应的带符号或不带符号的类型
- 与动态类型对象的 `constant` 或 `volatile` 限定版本对应的带符号或无符号类型
- 在其成员（包括递归的子集成员或包含的联合）中包括上述类型的聚集或联合类型
- 动态类型对象的（可能是 `constant` 或 `volatile` 限定）基类类型
- char 或 unsigned char type。

编译器假定所有引用的类型都与相应存储对象的动态类型是布局兼容的。两种类型在以下情况下是布局兼容的：

- 如果两种类型是同一类型
- 如果两种类型仅在不变和可变限定方面存在区别
- 对于每个带符号整数类型，如果存在对应（但不相同）的无符号整数类型，则这些对应类型是布局兼容的。
- 如果两个枚举类型具有相同的基础类型，则它们是布局兼容的。
- 如果两个简单旧数据 (plain old data, POD) 结构类型具有相同数量的成员，并且对应的成员（按顺序）具有布局兼容的类型，那么这两个结构类型是布局兼容的。

- 如果两个 POD 联合类型具有相同数量的成员，并且对应的成员（按顺序）具有布局兼容的类型，那么这两个联合类型是布局兼容的。

在某些情况下具有存储对象动态类型的引用可能是非布局兼容的：

- 如果 POD 联合包含了两个或两个以上共享通用初始序列的 POD 结构，且 POD 联合对象当前包含了其中一个 POD 结构，就可以检查任何 POD 结构的通用初始部分。对包含一个或多个初始成员的序列来说，如果相应成员具有布局兼容类型（适用于位字段）和相同宽度，则两个 POD 结构共享一个通用初始序列。
- 指向 POD 结构对象的指针（使用 `reinterpret_cast` 适当转换）将指向该结构的初始成员，而如果该成员是位字段则指向该结构所在的单元。

### A.2.101.4 缺省值

如果未指定 `-xalias_level`，则编译器将该选项设置为 `-xalias_level=any`。如果指定了 `-xalias_level` 但未提供值，则编译器将该选项设置为 `-xalias_level=compatible`。

### A.2.101.5 交互

编译器在 `-xO2` 和更低的优化级别不执行基于类型的别名分析。

### A.2.101.6 警告

如果要使用 `reinterpret_cast` 或等价的旧式强制类型转换，程序可能会违反分析假定。此外，**联合类型**也违反了分析假设，如以下示例所示。

```
union bitbucket{
  int i;
  float f;
};

int bitsof(float f){
bitbucket var;
var.f=3.6;
return var.i;
}
```

## A.2.102 -xanalyze={code| no}

为可使用 Oracle Solaris Studio 代码分析器查看的源代码生成静态分析。

使用 `–xanalyze=code` 进行编译并在单独的步骤中进行链接时，还需要在链接步骤中包括 `–xanalyze=code`。

缺省值为 `–xanalyze=no`。有关更多信息，请参见 Oracle Solaris Studio 代码分析器文档。

## A.2.103 -xannotate[=yes|no]

（**仅限** *Solaris*）创建随后可由优化和检测工具 `binopt(1)`、code-analyzer(1)、`discover(1)`、`collect(1)` 和 `uncover(1)` 使用的二进制文件。

缺省值为 `-xannotate=yes`。指定不带值的 `-xannotate` 等效于 `-xannotate=yes`。

为优化使用优化和监测工具，`-xannotate=yes` 必须在编译时和链接时均有效。如果不使用优化和监测工具，则使用 `-xannotate=no` 进行编译和链接可以生成略小的二进制文件和库。

此选项在 Linux 系统上不可用。

## A.2.104 –xar

创建归档库。

生成使用模板的 C++ 归档文件时，请将在模板系统信息库中实例化的那些模板函数包括在该归档文件中。仅在使用 `-instances=extern` 选项编译了至少一个目标文件时才使用模板系统信息库。使用 `–xar` 进行编译可以根据需要自动将这些模板添加到归档文件中。

但是，由于编译器在缺省情况下不使用模板高速缓存，因此通常不需要使用 `–xar` 选项。您可以使用无格式 `ar(1)` 命令创建 C++ 代码的归档文件（`.a` 文件），除非某些代码是使用 `–instances=extern` 编译的。在此情况下（或者如果您不确定），请使用 `CC –xar` 命令而不是 `ar` 命令。

### A.2.104.1 值

对调用 `ar -c -r` 指定 `-xar` 并重新创建归档文件。

#### 示例

以下命令行归档包含在库和目标文件中的模板函数。

```
example% CC -xar -o libmain.a a.o b.o c.o
```

#### 警告

请勿在命令行上添加来自模板数据库中的 `.o` 文件。

请勿直接使用 `ar` 命令生成归档文件。应使用 `CC–xar` 以确保模板实例自动包括在归档文件中。

#### 另请参见

`ar(1)` 手册页

## A.2.105 –xarch=*isa*

指定目标指令集体系结构 *(instruction set architecture, ISA)*。

该选项将编译器生成的代码限制为特定指令集体系结构的指令。此选项不保证使用任何特定于目标的指令。不过，使用该选项会影响二进制程序的可移植性。

---

**注 –** 分别使用 `-m64` 或 `-m32` 选项来指定打算使用的内存模型 LP64（64 位）或 ILP32（32 位）。`-xarch` 选项不再指示内存模型，除非是为了与早期的发行版兼容，如下所示。

---

如果代码使用的 `_asm` 语句或内联模板（`.il` 文件）使用了特定于体系结构的指令，则在编译时可能需要使用相应的 `–xarch` 值以避免出现编译错误。

如果在不同的步骤中编译和链接，请确保在两个步骤中为 `-xarch` 指定了相同的值。有关在编译时和链接时都必须指定的所有编译器选项的完整列表，请参见第 43 页中的 “3.3.3 编译时选项和链接时选项”。

### A.2.105.1 用于 SPARC 和 x86 的 -xarch 标志

下表列出了 SPARC 和 x86 平台通用的 `-xarch` 关键字。

**表 A–21** 适用于 SPARC 和 x86 的 —xarch 标志

| 标志 | 含义 |
|---|---|
| `generic` | 使用大多数处理器通用的指令集。这是缺省值。 |
| `generic64` | 为了在大多数 64 位平台上获得良好性能而进行编译。<br>此选项与 `-m64 -xarch=generic` 等效，用于与早期的发行版兼容。 |
| `native` | 为了在此系统上获得良好性能而进行编译。编译器为运行它的当前系统处理器选择适当的设置。 |
| `native64` | 为了在此系统上获得良好性能而进行编译。<br>此选项与 `-m64 -xarch=native` 等效，用于与早期的发行版兼容。 |

### A.2.105.2 用于 SPARC 的 -xarch 标志

下表提供了 SPARC 平台上每个 `-xarch` 关键字的详细信息。

**表 A–22** 用于 SPARC 平台的 -xarch 标志

| 标志 | 含义 |
|---|---|
| `sparc` | 针对 SPARC-V9 ISA （但不带有可视化指令集 (Visual Instruction Set, VIS)，也不带有其他特定于实现的 ISA 扩展）进行编译。该选项在 V9 ISA 上使编译器生成高性能代码。 |

表 A–22 用于 SPARC 平台的 -xarch 标志 （续）

| 标志 | 含义 |
|---|---|
| sparcvis | 针对 SPARC-V9 加可视指令集 (Visual Instruction Set, VIS) 版本 1.0 进行编译，并具有 UltraSPARC 扩展。该选项在 UltraSPARC 体系结构上使编译器生成高性能代码。 |
| sparcvis2 | 此选项允许编译器在具有 UltraSPARC III 扩展的 UltraSPARC 体系结构以及可视化指令集 (VIS) 2.0 版上生成目标代码。 |
| sparcvis3 | 针对 SPARC-V 9 ISA 的 SPARC VIS 版本 3 进行编译。允许编译器使用 SPARC-V9 指令集、UltraSPARC 扩展（包括可视指令集 (Visual Instruction Set, VIS) 版本 1.0、UltraSPARC-III 扩展（包括可视指令集版本 2.0 以及混合乘加指令和可视指令集版本 3.0 中的指令。 |
| sparcfmaf | 允许编译器使用 SPARC-V9 指令集，加 UltraSPARC 扩展（包括可视指令集 (Visual Instruction Set, VIS) 版本 1.0）、UltraSPARC-III 扩展（包括可视指令集 (Visual Instruction Set, VIS) 版本 2.0）以及面向浮点乘加的 SPARC64 VI 扩展中的指令。<br>必须将 -xarch=sparcfmaf 与 fma=fused 结合使用，并具有某个优化级别，以使编译器尝试查找机会来自动使用乘加指令。 |
| sparcima | 针对 SPARC-V 9 ISA 的 SPARC IMA 版本进行编译。使编译器可以使用如下指令集内的指令：SPARC-V9 指令集、UltraSPARC 扩展（包括可视化指令集 (Visual Instruction Set, VIS) 版本 1.0）、UltraSPARC-III 扩展（包括可视化指令集 (Visual Instruction Set, VIS) 版本 2.0）、SPARC64 VI 扩展（用于浮点乘加）和 SPARC64 VII 扩展（用于整数乘加）。 |
| sparc4 | 针对 SPARC-V9 ISA 的 SPARC4 版本进行编译。允许编译器使用 SPARC-V9 指令集以及扩展（包括 VIS 1.0）、UltraSPARC-III 扩展（包括 VIS2.0、混合浮点乘加指令、VIS 3.0 和 SPARC4 指令）中的指令。 |
| v9 | 等效于 -m64 -xarch=sparc。使用 -xarch=v9 来获取 64 位内存模型的传统 makefile 和脚本仅需使用 -m64。 |
| v9a | 等效于 -m64 -xarch=sparcvis，并与早前的发行版兼容。 |
| v9b | 等效于 -m64 -xarch=sparcvis2，并与早前的发行版兼容。 |

另请注意：

- 用 generic,sparc, sparcvis2, sparcvis3, sparcfmaf, sparcima 编译的对象二进制文件 (.o) 可以链接起来并一起执行，但只能在支持链接的所有指令集的处理器上运行。
- 对于任何特定选择，生成的可执行文件在传统体系结构中可能不运行或运行缓慢。而且，由于所有指令集中均未实现四精度 (long double) 浮点指令，因此编译器不在其生成的代码中使用这些指令。

### A.2.105.3 用于 x86 的 -xarch 标志

下表列出了 x86 平台上的 -xarch 标志。

表 A–23 针对 x86 的 -xarch 标志

| 标志 | 含义 |
| --- | --- |
| amd64 | 等效于 -m64 -xarch=sse2（仅限 Solaris）。使用 -xarch=amd64 来获取 64 位内存模型的传统 makefile 和脚本仅需要使用 -m64。 |
| amd64a | 等效于 -m64 -xarch=sse2a（仅限 Solaris）。 |
| pentium_pro | 使指令集限于 32 位 Pentium Pro 体系结构。 |
| pentium_proa | 将 AMD 扩展（3DNow!、3DNow! 扩展和 MMX 扩展）添加到 32 位 Pentium Pro 体系结构中。 |
| sse | 将 SSE 指令集添加到 Pentium Pro 体系结构。 |
| ssea | 将 AMD 扩展（3DNow!、3DNow! 扩展和 MMX 扩展）添加到 32 位 SSE 体系结构中。 |
| sse2 | 将 SSE2 指令集添加到 Pentium Pro 体系结构。 |
| sse2a | 将 AMD 扩展（3DNow!、3DNow! 扩展和 MMX 扩展）添加到 32 位 SSE2 体系结构中。 |
| sse3 | 将 SSE3 指令集添加到 SSE2 指令集中。 |
| sse3a | 将 AMD 扩展指令（包括 3dnow）添加到 SSE3 指令集。 |
| ssse3 | 使用 SSSE3 指令集补充 Pentium Pro、SSE、SSE2 和 SSE3 指令集。 |
| sse4_1 | 使用 SSE4.1 指令集补充 Pentium Pro、SSE、SSE2、SSE3 和 SSSE3 指令集。 |
| sse4_2 | 使用 SSE4.2 指令集补充 Pentium Pro、SSE、SSE2、SSE3、SSSE3 和 SSE4.1 指令集。 |
| amdsse4a | 使用 AMD SSE4a 指令集。 |
| aes | 使用 Intel 高级加密标准指令集。 |
| avx | 使用 Intel 高级向量扩展指令集。 |

如果在 x86 平台上使用 –m64 编译或链接程序的任一部分，则也必须使用这些选项之一编译程序的所有部分。有关各种 Intel 指令集体系结构（SSE、SSE2、SSE3、SSSE3 等）的详细信息，请参阅 Intel-64 和 IA-32《Intel Architecture Software Developer's Manual》

另请参见第 26 页中的 “1.2 x86 特殊注意事项”和第 27 页中的 “1.4 二进制兼容验证”。

### A.2.105.4 交互

尽管可以单独使用该选项，但它是 -xtarget 选项的扩展的一部分，并且可用于覆盖由特定的 -xtarget 选项设置的 -xarch 值。例如， -xtarget=ultra2 可扩展为

-xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1。在以下命令中，-xarch=v8plusb 覆盖了由 -xtarget=ultra2 的扩展设置的 -xarch=v8plusa。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

不支持 –compat[=4] 与 -xarch=generic64、-xarch=native64、-xarch=v9、-xarch=v9a 或 -xarch=v9b 结合使用。

### A.2.105.5 警告

如果在进行优化时使用该选项，那么在指定体系结构上适当选择就可以提供高性能的可执行文件。但如果选择不当就会导致性能的严重降级，或导致在预定目标平台上无法执行二进制程序。

如果在不同的步骤中编译和链接，请确保在两个步骤中为 -xarch 指定了相同的值。

## A.2.106 -xautopar

为多个处理器启用自动并行化。执行依赖性分析（对循环进行迭代间数据依赖性分析）和循环重构。如果优化级别不是 -xO3 或更高，则将优化级别提高到 -xO3 并发出警告。

如果要进行自己的线程管理，请勿使用 -xautopar。

要达到更快的执行速度，则该选项需要多处理器系统。在单处理器系统中，生成的二进制文件的运行速度通常较慢。

要在多线程环境中运行已并行化的程序，必须在执行之前将环境变量 OMP_NUM_THREADS 设置为大于 1 的值。如果未设置，则缺省值为 2。要使用多个线程，请将 OMP_NUM_THREADS 设置为更大的值。将 OMP_NUM_THREADS 设置为 1，则会仅使用一个线程运行。通常，应将 OMP_NUM_THREADS 设置为正在运行的系统中的可用虚拟处理器数，该值可使用 Oracle Solaris psrinfo(1) 命令确定。

如果使用 -xautopar 且在一个步骤中进行编译和链接，则链接会自动将微任务化库和线程安全的 C 运行时库包含进来。如果使用 -xautopar 并在不同的步骤中进行编译和链接，则还必须使用 -xautopar 进行链接。

### A.2.106.1 另请参见

第 251 页中的 “A.2.152 -xopenmp[= *i*]”

## A.2.107 -xbinopt={prepare| off}

(SPARC) **此选项现在已废弃，将在以后的编译器发行版中删除。请参见**第 211 页中的 “A.2.103 -xannotate[=yes| no]”

指示编译器准备二进制文件，以便以后进行优化、转换和分析。请参见 binopt(1) 手册页。此选项可用于生成可执行文件或共享对象。如果在不同的步骤中进行编译，则在编译步骤和链接步骤中都必须有 -xbinopt：

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

如果有些源代码不可用于编译，仍可使用此选项来编译其余代码。然后，应将其用于可创建最终库的链接步骤中。在此情况下，只有用此选项编译的代码才能进行优化、转换或分析。

### A.2.107.1 缺省值

缺省值为 -xbinopt=off。

#### 交互

此选项必须与 -xO1 或更高的优化级别一起使用时才有效。使用此选项生成二进制文件时，文件大小会有所增加。

使用 -xbinopt=prepare 和 -g 编译会将调试信息包括在内，从而增加可执行文件的大小。

## A.2.108 -xbuiltin[={%all|%default|%none}]

启用或禁用对标准库调用进行优化改进。

使用 -xbuiltin 选项可改善对调用标准库函数的代码的优化。此选项使编译器可在对性能有益时替换内函数或内联系统函数。要了解如何解读编译器注释输出来确定编译器替换了哪些函数，请参见 er_src(1) 手册页。

使用 –xbuiltin=%all 时，替换会导致 errno 的设置变得不可靠。如果您的程序依赖于 errno 的值，请不要使用此选项。

–xbuiltin=%default 仅内联未设置 errno 的函数。errno 的值在任何优化级别上都始终是正确的，并且可以可靠地检查。在 –xO3 或更低级别上使用 –xbuiltin=%default 时，编译器将确定哪些调用有利于内联，并且不内联其他调用。

-xbuiltin=%none 选项表示采用缺省编译器行为，编译器对内置函数不进行任何特殊优化。

### A.2.108.1 缺省值

如果没有指定 –xbuiltin，则在以优化级别 –xO1 或更高级别进行编译时缺省值为 –xbuiltin=%default，而在以 –xO0 级别进行编译时，缺省值为 –xbuiltin=%none。如果指定 –xbuiltin 时未带参数，则缺省为 –xbuiltin=%all，编译器会更积极地替换内部函数或者对标准库函数进行内联。

请注意，–xbuiltin 仅内联系统头文件中定义的全局函数，从不内联用户定义的静态函数。尝试在全局函数上进行插入的用户代码可能会导致出现不确定的行为。

### 交互

宏 -fast 的扩展包括了 -xbuiltin=%all。

### 示例

下面的编译器命令请求标准库调用的特殊处理。

```
example% CC -xbuiltin -c foo.cc
```

下面的编译器命令请求不要对标准库调用进行特别处理。请注意，宏 -fast 的扩展包括了 -xbuiltin=%all。

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

## A.2.109 –xcache=*c*

定义要由优化器使用的高速缓存属性。此选项不保证使用每个特定的缓存属性。

**注 –** 尽管该选项可单独使用，但它是 -xtarget 选项扩展的一部分。它的主要用途是覆盖 -xtarget 选项提供的值。

可选属性 [/t*i*] 用于设置可以共享缓存的线程数。

### A.2.109.1 值

*c* 必须是下表中列出的值之一。

**表 A–24** -xcache 值

| 值 | 含义 |
|---|---|
| generic | 指示编译器在多数 x86 和 SPARC 处理器上使用缓存属性来获得高性能，而不降低任何处理器的性能。（缺省值）<br>如果需要，在每个新的发行版本中都会调整最佳定时属性。 |
| native | 设置在主机环境中最佳性能的参数。 |
| *s1*/*l1* /*a1*[/*t1*] | 定义级别 1 缓存属性 |

表 A–24 -xcache 值 （续）

| 值 | 含义 |
|---|---|
| *s1*/*l1* /*a1*[/*t1*] :*s2*/*l2* /*a2*[/*t2*] | 定义级别 1 和 2 缓存属性 |
| *s1*/*l1* /*a1*[/*t1*] :*s2*/*l2* /*a2*[/*t2*] :*s3*/*l3* /*a3*[/*t3*] | 定义级别 1、2 和 3 缓存属性 |

下表中介绍了高速缓存属性 s*i* /l*i*/a*i* /t*i* 的定义：

| 属性 | 定义 |
|---|---|
| s*i* | 级别为 *i* 时的数据高速缓存的**大小** (KB) |
| l*i* | 级别为 *i* 时的数据高速缓存的**行大小**（字节） |
| a*i* | 级别为 *i* 时的数据高速缓存的**关联性** |
| t*i* | 共享级别为 *i* 的缓存的硬件线程数 |

例如，*i*=1 指定 1 级高速缓存属性 *s1*/*l1*/*a1*。

### 缺省值

如果未指定 -xcache，则假定为缺省值 –xcache=generic。该值指示了编译器在多数 SPARC 处理器上使用缓存属性来获得高性能，而不降低任何处理器的性能。

如果没有为 *t* 指定值，则缺省值为 1。

### 示例

–xcache=16/32/4:1024/32/1 指定以下值：

Level 1 Cache（级别 1 高速缓存） 16 K 字节，32 字节行大小，四路关联

Level 2 Cache（级别 2 高速缓存） 1024 K 字节，32 字节行大小，直接映射关联

### 另请参见

–xtarget=*t*

## A.2.110 -xchar[= o]

提供此选项只是为了方便从 char 类型定义为 unsigned 的系统中迁移代码。如果不是从这样的系统中迁移，最好不要使用该选项。只有那些依赖字符类型符号的程序才需要重写，它们要改写成显式指定带符号或者无符号。

### A.2.110.1 值

可以将 *o* 替换为下表中列出的值之一。

表 A–25 -xchar 值

| 值 | 含义 |
|---|---|
| signed | 将声明为字符的字符常量和变量视为带符号的。此选项会影响已编译代码的行为，而不影响库例程的行为。 |
| s | 与 signed 等效 |
| unsigned | 将声明为字符的字符常量和变量视为无符号的。此选项会影响已编译代码的行为，而不影响库例程的行为。 |
| u | 等效于 unsigned |

#### 缺省值

如果未指定 -xchar，编译器将假定 -xchar=s。

如果指定了 -xchar 但未指定值，编译器将假定 -xchar=s。

#### 交互

-xchar 选项仅会更改使用 -xchar 编译的代码中 char 类型的值范围。该选项不会更改任何系统例程或头文件中 char 类型的值范围。具体来讲，当指定了此选项时，limits.h 定义的 CHAR_MAX 和 CHAR_MIN 值不会更改。因此，CHAR_MAX 和 CHAR_MIN 不再表示无格式 char 中可编码的值的范围。

#### 警告

如果使用 -xchar=unsigned，则在将 char 与预定义的系统宏进行比较时要特别小心，因为宏中的值可能带符号。对于任何返回错误代码而且可以用宏来访问错误代码的例程，此情况是最常见的。错误代码一般是负值，因此在将 char 与此类宏中的值进行比较时，结果始终为假。负数永远不等于无符号类型的值。

绝不要使用 -xchar 为通过库导出的任何接口编译例程。Oracle Solaris ABI 将 char 类型指定为带符号，并且系统库的行为也与此相适应。目前还未对系统库针对将 char 指定为无符号的效果进行广泛测试。可以不使用该选项，而是修改代码使其与 char 类型是否带符号没有关联。类型 char 的符号种类因编译器和操作系统而异。

# A.2.111 -xcheck[= i]

使用 -xcheck=stkovf 进行编译将增加对单线程程序中的主线程以及多线程程序中的从属线程堆栈进行堆栈溢出运行时检查。如果检测到堆栈溢出，则生成 SIGSEGV。有关如何以与其他地址空间违规的处理方式不同的方式处理堆栈溢出导致的 SIGSEGV 的信息，请参见 sigaltstack(2)。

## A.2.111.1 值

*i* 必须是下表中列出的值之一。

表 A–26 -xcheck 值

| 值 | 含义 |
|---|---|
| %all | 执行全部检查。 |
| %none | 不执行检查。 |
| stkovf | 打开堆栈溢出检查。 |
| no%stkovf | 关闭堆栈溢出检查。 |
| init_local | 初始化局部变量。有关详细信息，请参见《C 用户指南》。 |
| no%init_local | 不初始化局部变量（缺省设置）。 |

### 缺省值

如果未指定 -xcheck，则编译器缺省使用 -xcheck=%none。

如果指定了没有任何参数的 -xcheck，则编译器缺省使用 -xcheck=%none。

在命令行上 -xcheck 选项不进行累积。编译器按照上次出现的命令设置标志。

# A.2.112 -xchip=*c*

指定要由优化器使用的目标处理器。

–xchip 选项通过指定目标处理器来指定定时属性。该选项影响以下属性：

- 指令的顺序（即调度）
- 编译器使用分支的方法
- 语义上等价的其他指令可用时使用的指令

**注 –** 尽管该选项可单独使用，但它是 `-xtarget` 选项扩展的一部分。它的主要用途是覆盖 `-xtarget` 选项提供的值。

### A.2.112.1 值

*c* 必须是下面的两个表中列出的值之一。

**表 A–27** 适用于 SPARC 处理器的 -xchip 值

| | |
|---|---|
| `generic` | 在大多数 SPARC 处理器上性能良好 |
| `native` | 在运行编译器的主机 SPARC 系统上性能良好 |
| `sparc64vi` | SPARC64 VI 处理器 |
| `sparc64vii` | SPARC64 VII 处理器 |
| `sparc64viiplus` | SPARC64 VII+ 处理器 |
| `ultra` | UltraSPARC 处理器 |
| `ultra2` | UltraSPARC II 处理器 |
| `ultra2e` | UltraSPARC IIe 处理器 |
| `ultra2i` | UltraSPARC IIi 处理器 |
| `ultra3` | UltraSPARC III 处理器 |
| `ultra3cu` | UltraSPARC III Cu 处理器 |
| `ultra3i` | UltraSparc IIIi 处理器 |
| `ultra4` | UltraSPARC IV 处理器 |
| `ultra4plus` | UltraSPARC IVplus 处理器 |
| `ultraT1` | UltraSPARC T1 处理器 |
| `ultraT2` | UltraSPARC T2 处理器。 |
| `ultraT2plus` | UltraSPARC T2+ 处理器。 |
| `T3` | SPARC T3 处理器。 |
| `T4` | SPARC T4 处理器。 |

**表 A–28** 适用于 x86/x64 处理器的 -xchip 值

| | |
|---|---|
| `generic` | 在大多数 x86 处理器上性能良好 |
| native | 在运行编译器的主机 x86 系统上性能良好 |

表 A–28 适用于 x86/x64 处理器的 -xchip 值 （续）

| | |
|---|---|
| core2 | Intel Core2 处理器 |
| nehalem | Intel Nehalem 处理器 |
| opteron | AMD Opteron 处理器 |
| penryn | Intel Penryn 处理器 |
| pentium | Intel Pentium 处理器 |
| pentium_pro | Intel Pentium Pro 处理器 |
| pentium3 | Intel Pentium 3 式处理器 |
| pentium4 | Intel Pentium 4 式处理器 |
| amdfam10 | AMD AMDFAM10 处理器 |
| sandybridge | Intel Sandy Bridge 处理器 |
| westmere | Intel Westmere 处理器 |

### 缺省值

在大多数处理器中，generic 为缺省值，即指示编译器使用最佳定时属性以获得高性能，而不会显著降低任何处理器的性能。

## A.2.113 –xcode=*a*

（**仅限** *SPARC*）指定代码地址空间。

**注 –** 应通过指定 -xcode=pic13 或 -xcode=pic32 生成共享对象。未使用 pic13 或 pic32 生成的共享对象不能正常工作，也可能根本无法生成。

### A.2.113.1 值

*a* 必须是下表中列出的值之一。

表 A–29 -xcode 值

| 值 | 含义 |
|---|---|
| abs32 | 生成快速但有范围限制的 32 位绝对地址。代码 + 数据 + bss 的大小被限制为 2**32 字节。 |
| abs44 | SPARC：生成具有适当速度和范围的 44 位绝对地址。代码+数据+bss 的大小不应超过 2**44 字节。只适用于 64 位架构。请勿将该值与动态（共享）库一起使用。 |

表 A–29 -xcode 值 （续）

| 值 | 含义 |
|---|---|
| abs64 | SPARC：生成缓慢但无范围限制的 64 位绝对地址。只适用于 64 位架构。 |
| pic13 | 生成快速但有范围限制的位置无关代码（小模型）。与 -Kpic 等效。允许在 32 位架构上最多引用 2**11 个唯一的外部符号，而在 64 位架构上可以最多引用 2**10 个。 |
| pic32 | 生成与位置无关的代码（大模型），这可能没有 pic13 快，但有完整范围。等效于 -KPIC。允许在 32 位体系结构上最多引用 2**30 个唯一的外部符号，而在 64 位体系结构上最多可以引用 2**29 个。 |

要确定是使用 –xcode=pic13 还是使用 –xcode=pic32，请使用 elfdump -c 检查全局偏移表 (Global Offset Table, GOT) 的大小并查找节头 sh_name: .got。sh_size 值是 GOT 的大小。如果 GOT 小于 8,192 字节，请指定 -xcode=pic13，否则指定 -xcode=pic32。有关更多信息，请参见 elfdump(1) 手册页。

通常，应根据以下准则来确定如何使用 -xcode：

- 如果要生成可执行文件，则不应该使用 -xcode=pic13 或 -xcode=pic32。
- 如果是生成仅用于链接到可执行文件的归档库，则不应该使用 -xcode=pic13 或 -xcode=pic32。
- 如果要生成共享库，请以 – xcode=pic13 开始。一旦 GOT 大小超过 8,192 字节，请使用 -xcode=pic32。
- 如果要生成用于链接到共享库的归档库，只能使用 -xcode=pic32。

## 缺省值

对于 32 位体系结构，缺省值是 -xcode=abs32。64 位体系结构的缺省值是 -xcode=abs44。

生成共享动态库时，缺省 -xcode 值 abs44 和 abs32 将与 64 位体系结构一起使用。但指定 -xcode=pic13 或 -xcode=pic32。在 SPARC 上使用 –xcode=pic13 和 –xcode=pic32 时存在两项名义性能开销：

- 用 –xcode=pic13 或 –xcode=pic32 编译的例程会在入口点执行一些附加指令，以将寄存器设置为指向用于访问共享库的全局变量或静态变量的表 (_GLOBAL_OFFSET_TABLE_)。
- 对全局或静态变量的每次访问都会涉及通过 _GLOBAL_OFFSET_TABLE_ 的额外间接内存引用。如果是使用 -xcode=pic32 进行编译，则对于每个全局和静态内存引用还要执行另外两个指令。

在考虑上述成本时，请记住：由于受到库代码共享的影响，使用 -xcode=pic13 和 -xcode=pic32 会大大减少系统内存需求。共享库中使用 -xcode=pic13 或 – xcode=pic32

编译的每页代码都可以供使用该库的每个进程共享。如果共享库中的代码页包含非 pic（即绝对）内存引用，即使仅包含单个非 pic 内存引用，该页也将变为不可共享，而且每次执行使用该库的程序时都必须创建该页的副本。

确定是否已经使用 -xcode=pic13 或 –xcode=pic32 编译了 .o 文件的最简单方法是使用 nm 命令：

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

包含与位置无关的代码的 .o 文件将包含对 _GLOBAL_OFFSET_TABLE_ 无法解析的外部引用（用字母 U 标记）。

要确定是使用 –xcode=pic13 还是使用 –xcode=pic32，应使用 nm 确定库中使用或定义的不同全局变量和静态变量的数量。如果 _GLOBAL_OFFSET_TABLE_ 的大小小于 8,192 字节，就可以使用 -Kpic。否则，就必须使用 -xcode=pic32。

## A.2.114 -xdebugformat=[stabs|dwarf]

编译器已将调试器信息格式从 *stabs*（“符号表”）格式迁移到“DWARF 调试信息格式”中指定的 *dwarf* 格式。缺省设置为 -xdebugformat=dwarf。

如果要维护读取调试信息的软件，您现在可以选择将工具从 stabs 格式转换为 dwarf 格式。

出于移植工具的目的，可以通过此选项来使用新的格式。除非您要维护读取调试器信息的软件，或者特定工具要求使用这些格式之一的调试器信息，否则不需要使用此选项。

**表 A–30** -xdebugformat 标志

| 值 | 含义 |
|---|---|
| stabs | -xdebugformat=stabs 生成使用 stabs 标准格式的调试信息。 |
| dwarf | -xdebugformat=dwarf 生成的调试信息采用 dwarf 标准格式。 |

如果未指定 -xdebugformat，编译器将假定 -xdebugformat=dwarf。此选项需要一个参数。

此选项影响使用 -g 选项记录的数据的格式。即使在没有使用 -g 的情况下记录少量调试信息，此选项仍可控制其信息格式。因此，即使不使用 -g，-xdebugformat 仍有影响。

dbx 和性能分析器软件可识别 stabs 和 dwarf 格式，因此使用此选项对任何工具的功能都没有影响。

**注 –** Stabs 格式并不能提供当前由 dbx 使用的所有调试数据，并且某些代码可能无法成功使用 stabs 来生成调试数据。

有关更多信息，另请参见 dumpstabs(1) 和 dwarfdump(1) 手册页。

## A.2.115 -xdepend=[yes| no]

对循环进行迭代间数据相关项分析，并执行循环重构，包括循环交换、循环合并、标量替换和“死数组”赋值消除。

在 SPARC 处理器上，对于 –xO3 及更高的所有优化级别，–xdepend 缺省为 –xdepend=on。否则，–xdepend 缺省为 –xdepend=off。指定 –xdepend 的显式设置会覆盖任何缺省设置。

在 x86 处理器上，–xdepend 缺省为 –xdepend=off。指定 -xdepend 且优化级别不是 –xO3 或更高级别时，编译器会将优化级别提高到 –xO3 并发出警告。

指定不带参数的 –xdepend 等效于 –xdepend=yes。

-xautopar 中包括依赖性分析。依赖性分析在编译时完成。

依赖性分析在单处理器系统中可能很有用。但是，如果在单处理器系统上使用 –xdepend，不应该同时指定 –xautopar，因为将针对多处理器系统进行 –xdepend 优化。

### A.2.115.1 另请参见

–xprefetch_auto_type

## A.2.116 -xdumpmacros[= value[,value...]]

要查看宏在程序中的行为方式时使用此选项。该选项提供了诸如宏定义、取消定义的宏和宏用法实例的信息，并按宏的处理顺序将输出信息输出到标准错误 (stderr)。-xdumpmacros 选项在整个文件中或在 dumpmacros 或 end_dumpmacros pragma 覆盖它之前都是有效的。请参见第 286 页中的 “B.2.5 #pragma dumpmacros”。

### A.2.116.1 值

下表列出了 *value* 的有效参数。前缀 no% 可禁用关联的值。

表 A–31 -xdumpmacros 值

| 值 | 含义 |
| --- | --- |
| [no%]defs | 输出所有宏定义。 |
| [no%]undefs | 输出所有取消定义的宏。 |
| [no%]use | 输出关于使用的宏的信息。 |
| [no%]loc | 另外输出 defs、undefs 和 use 的位置（路径名和行号）。 |
| [no%]conds | 输出在条件指令中使用的宏的使用信息。 |
| [no%]sys | 输出系统头文件中的宏的所有宏定义、取消定义和使用信息。 |
| %all | 设置该选项即表示 -xdumpmacros=defs,undefs,use,loc,conds,sys。该参数最好与 [no%] 形式的其他参数配合使用。例如，-xdumpmacros=%all,no%sys 表示输出中不包含系统头文件宏，但仍提供所有其他宏的信息。 |
| %none | 不输出任何宏信息。 |

该选项的值会累积，因此指定 -xdumpmacros=sys -xdumpmacros=undefs 与 -xdumpmacros=undefs,sys 的效果相同。

**注 –** 子选项 loc、conds 和 sys 是 defs、undefs 和 use 选项的限定符。使用 loc、conds 和 sys 本身并不会生成任何结果。例如，使用 -xdumpmacros=loc,conds,sys 不会生成什么结果。

## 缺省值

指定不带任何参数的 -xdumpmacros 缺省设置为 -xdumpmacros=defs,undefs,sys。未指定 -xdumpmacros 时的缺省值为 -xdumpmacros=%none。

## 示例

如果使用选项 -xdumpmacros=use,no%loc，则使用的每个宏名称只输出一次。但是，如果要了解更多详细信息，请使用选项 -xdumpmacros=use,loc，这样每次使用宏时都会打印位置和宏名称。

例如以下文件 t.c：

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b) a+b
#else
#define COMPUTE(a,b) a-b
```

```
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

以下示例显示了使用 defs、undefs、sys 和 loc 参数时文件 t.c 的输出。

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_9 1
#define __SUNPRO_CC 0x590
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b

example% CC -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
command line: #define __SunOS_5_9 1
command line: #define __SUNPRO_CC 0x590
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUNPRO_CC_COMPAT 5
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
command line: #undef BAR
t.c, line 2: #undef FOO
t.c, line 3: #define COMPUTE(a, b) a + b
```

以下示例说明了 use、loc 和 conds 参数如何报告文件 t.c 中宏的行为：

```
example% CC -c -xdumpmacros=use t.c
used macro COMPUTE

example% CC -c -xdumpmacros=use,loc t.c
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE

example% CC -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
```

```
used macro MM

example% CC -c -xdumpmacros=use,conds,loc t.c
t.c, line 1: used macro FOO
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
t.c, line 9: used macro COMPUTE
t.c, line 9: used macro NN
t.c, line 9: used macro MM
```

例如文件 `y.c`：

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

以下示例基于 `y.c` 中的宏显示了 `-xdumpmacros=use,loc` 的输出：

```
example% CC -c -xdumpmacros=use,loc y.c
y.c, line 4: used macro Z
y.c, line 4: used macro Y
y.c, line 4: used macro X
```

### 另请参见

Pragma `dumpmacros/end_dumpmacros` 会覆盖 `-xdumpmacros` 命令行选项的作用域。

## A.2.117 -xe

仅检查语法和语义错误。指定 `-xe` 时，编译器不生成任何目标代码。 `-xe` 的输出定向到 `stderr`。

如果不需要通过编译生成目标文件，可使用 `-xe` 选项。例如，如果要尝试通过删除代码段找出导致出现错误消息的原因，可使用 `-xe` 加速编辑和编译周期。

### A.2.117.1 另请参见

–c

## A.2.118 -xF[=v[, v...]]

启用通过链接程序对函数和变量进行最佳重新排列。

该选项指示编译器将函数和/或数据变量放置到单独的分段中，这样链接程序就可以使用链接程序 `-M` 选项指定的映射文件中的指示将这些段重新排序以优化程序性能。通常，该优化仅在缺页时间构成程序运行时间的一大部分时才有效。

对变量重新排序有助于解决对运行时性能产生负面影响的以下问题：

- 在内存中存放位置很近的无关变量会造成缓存和页的争用
- 在内存中存放位置很远的相关变量会造成不必要的过大工作集
- 未用到的弱变量副本会造成不必要的过大工作集，从而降低有效数据密度

为优化性能而对变量和函数进行重新排序时，需要执行以下操作：

1. 使用 `-xF` 进行编译和链接。
2. 按照性能分析器手册中关于如何生成函数的映射文件或《链接程序和库指南》中关于如何生成数据的映射文件的说明进行操作。
3. 使用通过链接程序的 `-M` 选项生成的新映射文件重新链接。
4. 在分析器下重新执行以验证是否增强。

### A.2.118.1 值

*v* 可以是下表中列出的一个或多个值。`no%` 前缀禁用关联的值。

**表 A–32** -xF 值

| 值 | 含义 |
|---|---|
| `[no%]func` | 将函数分段到单独的段中。 |
| `[no%]gbldata` | 将全局数据（具有外部链接的变量）分段到单独的段中。 |
| `[no%]lcldata` | 将局部数据（具有内部链接的变量）分段到单独的段中。 |
| `%all` | 分段函数、全局数据和局部数据。 |
| `%none` | 不分段。 |

#### 缺省值

如果未指定 `-xF`，则缺省值为 `-xF=%none`。如果指定了没有任何参数的 `-xF`，则缺省值为 `-xF=%none,func`。

#### 交互

使用 `-xF=lcldata` 会限制某些地址计算优化，因此，只应在必要时才使用该标志。

#### 另请参见

`analyzer`(1) 和 `ld`(1) 手册页

## A.2.119 -xhelp=flags

显示了对每个编译器选项的简要描述。

## A.2.120 -xhwcprof

（**仅限** *SPARC*）为基于硬件计数器的分析启用编译器支持。

如果启用了 `-xhwcprof`，编译器将生成信息，这些信息可帮助工具将分析的加载和存储指令与其所引用的数据类型和结构成员相关联（与使用 `-g` 生成的符号信息结合）。它将分析数据与目标的数据空间（而非指令空间）相关联。使用此选项可以对行为进行深入洞察，单独通过指令分析难以做到这一点。

可使用 `-xhwcprof` 编译一组指定的目标文件。但是，当应用于应用程序中的所有目标文件时，`-xhwcprof` 是最有用的，它能全面识别并关联分布在应用程序的目标文件中的所有内存引用。

如果在不同的步骤中进行编译和链接，最好在链接时使用 `-xhwcprof`。如果将来扩展为 `-xhwcprof`，则在链接时可能需要使用它。

`-xhwcprof=enable` 或 `-xhwcprof=disable` 的实例将会覆盖同一命令行中 `-xhwcprof` 的所有以前的实例。

在缺省情况下，禁用 `-xhwcprof`。指定不带任何参数的 `-xhwcprof` 与 `-xhwcprof=enable` 等效。

`-xhwcprof` 要求启用优化并选择 DWARF 调试数据格式。请注意，DWARF 格式 (`-xdebugformat=dwarf`) 现在是缺省格式。

组合使用 `-xhwcprof` 和 `-g` 会增加编译器临时文件的存储需求，而且高于单独指定 `-xhwcprof` 和 `-g` 所引起的增加总量。

下列命令可编译 `example.cc`，并可为硬件计数器分析以及针对使用 DWARF 符号的数据类型和结构成员的符号分析指定支持：

```
example% CC -c -O -xhwcprof -g -xdebugformat=dwarf example.cc
```

有关基于硬件计数器的分析的更多信息，请参见性能分析器手册。

## A.2.121 -xia

链接适当的区间运算库，并设置适当的浮点环境。

---

**注 –** C++ 区间运算库与 Fortran 编译器中实现的区间运算相兼容。

---

在 x86 平台上，该选项需要 SSE2 指令集支持。

### A.2.121.1 扩展

-xia 选项是一个扩展到 -fsimple=0 -ftrap=%none -fns=no -library=interval 的宏。如果使用区间并通过为 -fsimple、-ftrap、-fns 或 -library 指定不同的标志来覆盖 -xia 设置的内容，则可能会导致编译器出现不正确的行为。

### A.2.121.2 交互

要使用区间运算库，请将 <suninterval.h> 包含进来。

在使用区间运算库时，必须包括以下库之一：Cstd 或 iostream。有关包括这些库的信息，请参见 -library。

### A.2.121.3 警告

如果您使用区间并为 -fsimple、-ftrap 或 -fns 指定了不同的值，则您的程序可能显示不正确的行为。

C++ 区间运算处于实验阶段且正在改进。具体功能可能随发行版本而变。

### A.2.121.4 另请参见

-library

## A.2.122 -xinline[= func-spec[,func-spec...]]

指定在 -xO3 或更高的优化级别优化器可以内联用户编写的哪些例程。

### A.2.122.1 值

*func-spec* 必须是下表中列出的值之一。

表 A–33 -xinline 值

| 值 | 含义 |
|---|---|
| %auto | 在 -xO4 或更高的优化级别上启用自动内联。此参数告知优化器它可以内联所选择的函数。请注意，如果没有指定 %auto，则在命令行上使用 -xinline=[no%]*func-name*... 指定显式内联后，通常会禁用自动内联。 |
| *func_name* | 强烈请求优化器内联函数。如果函数未声明为 extern "C"，则必须改编 *func_name* 的值。可以对可执行文件使用 nm 命令来查找改编的函数名。对于已声明为 extern "C" 的函数，编译器不改编名称。 |
| no%*func_name* | 如果为列表上的例程添加名称前缀 no%，则会禁止内联该例程。关于 *func-name* 已改编名称的规则也适用于 no%*func-name*。 |

只有使用了 -xipo[=1|2] 时，才会内联要编译的文件中的例程。优化器决定适合内联的例程。

### A.2.122.2 缺省值

如果未指定 -xinline 选项，则编译器假定 -xinline=%auto。

如果指定了没有任何参数的 -xinline=，则不内联函数，而不管优化级别是什么。

### A.2.122.3 示例

要启用自动内联同时禁用内联声明为 int foo() 的函数，请使用以下命令：

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

要强烈要求内联声明为 int foo() 的函数，并使所有其他函数作为要内联的候选函数，请使用以下命令：

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

要强烈要求内联声明为 int foo() 的函数，且不允许内联任何其他函数，请使用以下命令：

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

### A.2.122.4 交互

优化级别低于 -xO3 时，-xinline 选项不起作用。在 -xO4 或更高的优化级别上，优化器会决定应该内联哪些函数，无需指定 -xinline 选项即可完成。另外，在 -xO4 或更高的优化级别上，编译器会尝试确定内联哪些函数可以提高性能。

如果出现以下任一情况，则会内联例程。

- 优化级别为 -xO3 或更高
- 认为内联有益且安全
- 函数在要编译的文件中，或函数在使用 -xipo[=1|2] 编译了的文件中

### A.2.122.5 警告

如果使用 -xinline 强制内联函数，实际上可能会降低性能。

### A.2.122.6 另请参见

第 239 页中的 “A.2.130 -xldscope={*v*}”

## A.2.123 -xinstrument=[ no%]datarace

指定此选项编译并检测您的程序，以供线程分析器进行分析。有关线程分析器的更多详细信息，请参见 tha(1) 手册页。

然后可使用性能分析器以 collect -r races 来运行此检测过的程序，从而创建数据争用检测实验。可以单独运行已检测的代码，但其运行速度将非常缓慢。

可指定 -xinstrument=no%datarace 来关闭线程分析器的源代码准备。这是缺省值。

指定 -xinstrument 时必须带参数。

如果在不同的步骤中进行编译和链接，则在编译和链接步骤都必须指定 -xinstrument=datarace。

此选项定义了预处理程序令牌 __THA_NOTIFY。可指定 #ifdef __THA_NOTIFY 来保护对 libtha(3) 例程的调用。

该选项也设置 -g。

## A.2.124 -xipo[={0|1|2}]

执行过程间优化。

-xipo 选项通过调用过程间分析传递来执行部分程序优化。它会在链接步骤中对所有目标文件执行优化，且优化不限于只是编译命令中的那些源文件。但是，使用 -xipo 执行的整个程序优化不包括汇编 (.s) 源文件。

编译和链接大型多文件应用程序时，-xipo 选项特别有用。用该标志编译的对象目标文件具有在这些文件内编译的分析信息，这些信息实现了在源代码和预编译的程序文件中的过程间分析。但分析和优化只限于使用 -xipo 编译的目标文件，并不扩展到目标文件或库。

### A.2.124.1 值

-xipo 选项可以使用下表中列出的值。

表 A–34 -xipo 值

| 值 | 含义 |
|---|---|
| 0 | 不执行过程间的优化 |
| 1 | 执行过程间的优化 |
| 2 | 执行过程间的别名分析和内存分配及布局的优化，以提高缓存的性能 |

### A.2.124.2 缺省值

如果未指定 -xipo，则假定 -xipo=0。

如果仅指定了 -xipo，则假定 -xipo=1。

### A.2.124.3 示例

以下示例在相同的步骤中编译和链接。

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

优化器在最后一个链接步骤中在三个源文件之间执行交叉文件内联。不必一次编译所有源文件，可以分多次进行编译，每次编译时都指定 -xipo 选项。

以下示例在不同的步骤中编译和链接。

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog part1.o part2.o part3.o
```

在编译步骤中创建的目标文件具有在文件内部编译的附加分析信息，这样就可以在链接步骤中执行跨文件优化。

### A.2.124.4 何时不使用 -xipo 过程间分析

在链接步骤中使用目标文件集合时，编译器试图执行整个程序分析和优化。对于该目标文件集合中定义的任何函数或子例程 foo()，编译器做出以下两个假定：

- 运行时在该目标文件集合之外定义的其他例程不显式调用 foo()。
- 目标文件集合中的任何例程调用 foo() 时，不会插入该目标文件集合之外定义的不同版本的 foo()。

如果第一个假设对于给定的应用程序不成立，请勿使用 -xipo=2 进行编译。

如果第二个假设不成立，请不要使用 -xipo=1 也不要使用 -xipo=2 进行编译。

例如，如果对函数 malloc() 创建了您自己的版本，并使用 -xipo=2 进行编译。对于任何库中引用 malloc() 且与您的代码链接的所有函数，也都必须使用 -xipo=2 进行编译，并且需要在链接步骤中对其目标文件进行操作。由于该策略可能不适用于系统库，因此不要使用 -xipo=2 编译您的 malloc() 版本。

另举一例，如果生成了一个共享库，有两个外部调用（foo() 和 bar()）分别在两个不同的源文件中。并假设 bar() 调用 foo()。如果可能会在运行时插入 foo()，则不要使用 -xipo=1 或 -xipo=2 编译 foo() 或 bar() 的源文件。否则，foo() 会内联到 bar() 中，从而导致出现错误的结果。

### A.2.124.5 交互

-xipo 选项要求优化级别至少为 -xO4。

### A.2.124.6 警告

在不同的步骤中进行编译和链接时，必须在这两个步骤中都指定 -xipo 才有效。

没有使用 -xipo 编译的对象可以自由地与使用 -xipo 编译的对象链接。

即使使用 -xipo 对库进行了编译，这些库也不参与交叉文件的过程间分析，如以下示例中所示。

```
example% CC -xipo -xO4 one.cc two.cc three.cc
example% CC -xar -o mylib.a one.o two.o three.o
...
example% CC -xipo -xO4 -o myprog main.cc four.cc mylib.a
```

本示例中，在 one.cc、two.cc 和 three.cc 之间以及 main.cc 和 four.cc 之间执行过程间优化，但不在 main.cc 或 four.cc 和 mylib.a 中的例程之间执行过程间优化。（第一个编译可能生成有关未定义符号的警告，但仍可执行过程间优化，因为过程间优化是编译和链接的一个步骤。）

由于执行跨文件优化时需要附加信息，因此 -xipo 选项会生成更大的目标文件。不过，该附加信息不会成为最终的二进制可执行文件的一部分。可执行程序大小的增加都是由于执行的附加优化导致的。

### A.2.124.7 另请参见

-xjobs

## A.2.125 -xipo_archive=[a]

-xipo_archive 选项使编译器可在生成可执行文件之前，用通过 -xipo 编译且驻留在归档库 (.a) 中的对象文件来优化传递给链接程序的对象文件。库中包含的在编译期间优化的任何目标文件都会替换为其优化后的版本。

下表列出了 *a* 的可能值。

**表 A–35** -xipo_archive 标志

| 值 | 含义 |
|---|---|
| writeback | 生成可执行文件之前，编译器使用通过 -xipo 编译的目标文件（驻留在归档库 (.a) 中）来优化传递到链接程序的目标文件。库中包含的在编译期间优化的任何目标文件都会替换为优化后的版本。<br>对于使用归档库通用集的并行链接，每个链接都应创建自己的归档库备份，从而在链接前进行优化。 |

表 A–35 -xipo_archive 标志 （续）

| 值 | 含义 |
|---|---|
| readonly | 生成可执行文件之前，编译器使用通过 -xipo 编译的目标文件（驻留在归档库 (.a) 中）来优化传递到链接程序的目标文件。<br><br>通过 -xipo_archive=readonly 选项，可在链接时指定的归档库中进行目标文件的跨模块内联和程序间数据流分析。但是，它不启用对归档库代码的跨模块优化，除非代码已经通过跨模块内联插入到其他模块中。<br><br>要对归档库内的代码应用跨模块优化，要求 -xipo_archive=writeback。请注意，此设置将修改从中提取代码的归档库的内容。 |
| none | 这是缺省值。没有对归档文件的处理。编译器不对使用 -xipo 编译和在链接时从归档库中提取的目标文件应用跨模块内联或其他跨模块优化。要执行此操作，链接时必须指定 -xipo，以及 -xipo_archive=readonly 或 -xipo_archive=writeback 中的任一个。 |

如果不为 -xipo_archive 指定设置，编译器会将其设置为 -xipo_archive=none。

指定 -xipo_archive 时必须带标志。

## A.2.126 -xivdep[= p]

禁用或设置 #pragma ivdep pragma 的解释（**忽略向量依赖性**）。

ivdep pragma 指示编译器忽略在循环中找到的部分或全部对数组引用的循环附带依赖性，以进行优化。这样，编译器就可以执行各种循环优化，例如微向量化、分发、软件流水操作等，其他情况下，无法执行这些优化。当用户知道这些依赖性无关紧要或者实际上永远不会发生时，可以使用该指令。

#pragma ivdep 指令的解释依赖于 –xivdep 选项的值。

以下列表列出了 *p* 的值及其含义。

loop 忽略假定的循环附带依赖性

loop_any 忽略所有循环附带向量依赖性

back 忽略假定的向后循环附带向量依赖性

back_any 忽略所有向后循环附带向量依赖性

none 不忽略任何依赖性（禁用 ivdep pragma）

提供这些解释是为了与另一个供应商对 ivdep pragma 的解释兼容。

## A.2.127 -xjobs=n

指定 -xjobs 选项设置编译器为完成其工作创建的进程数。在多处理器计算机上，该选项可以缩短生成时间。目前，-xjobs 只能与 -xipo 选项一起使用。如果指定 -xjobs=*n*，过程间优化器就将 *n* 作为其在编译不同文件时可调用的最大代码生成器实例数。

### A.2.127.1 值

指定 -xjobs 时务必要指定值。否则，会发出错误诊断并使编译终止。

通常，*n* 的安全值等于 1.5 乘以可用处理器数。如果使用的值是可用处理器数的数倍，则会降低性能，因为有在产生的作业间进行的上下文切换开销。此外，如果使用很大的数值会耗尽系统资源（如交换空间）。

### A.2.127.2 缺省值

出现最合适的实例之前，-xjobs 的多重实例在命令行上会互相覆盖。

### A.2.127.3 示例

以下示例在有两个处理器的系统上进行的编译，速度比使用相同命令但没有 -xjobs 选项时进行的编译快。

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

## A.2.128 -xkeepframe[=[ %all,%none,name,no% name]]

禁止对命名函数 ( *name*) 进行与堆栈相关的优化。

`%all` 禁止对所有代码进行与堆栈相关的优化。

`%none` 允许对所有代码进行与堆栈相关的优化。

此选项是累积性的，可以多次出现在命令行中。例如，–xkeepframe=%all –xkeepframe=no%func1 表示应保留 func1 以外的所有函数的堆栈帧。而且，–xkeepframe 优先于 –xregs=frameptr。例如，–xkeepframe=%all –xregs=frameptr 表示应保留所有函数的堆栈，但会忽略 –xregs=frameptr 的优化。

如果命令行中未指定，编译器将采用 -xkeepframe=%none 作为缺省值。如果指定了但没有值，编译器将采用 -xkeepframe=%all

# A.2.129 -xlang=language [,language]

包含适当的运行时库，并确保指定语言的适当运行时环境。

## A.2.129.1 值

*language* 必须是 `f77`、`f90`、`f95` 或 `c99`。

`f90` 和 `f95` 参数等价。`c99` 参数表示为已使用 `-xc99=%all` 编译并要使用 CC 链接的对象调用 ISO 9899:1999 C 编程语言行为。

## A.2.129.2 交互

`-xlang=f90` 和 `-xlang=f95` 选项隐含了 `-library=f90`，而 `-xlang=f77` 选项隐含了 `-library=f77`。但要进行混合语言链接，只使用 `-library=f77` 和 `-library=f90` 选项是不够的，因为只有 `-xlang` 选项才能确保适当的运行时环境。

要决定在混合语言链接中使用的驱动程序，请使用下列语言分层结构：

1. C++
2. Fortran 95（或 Fortran 90）
3. Fortran 77
4. C 或 C99

将 Fortran 95、Fortran 77 和 C++ 目标文件链接在一起时，请使用最高级语言的驱动程序。例如，使用下列 C++ 编译器命令来链接 C++ 和 Fortran 95 目标文件：

```
example% CC -xlang=f95...
```

要链接 Fortran 95 和 Fortran 77 目标文件，请使用如下所示的 Fortran 95 驱动程序：

```
example% f95 -xlang=f77...
```

不能在同一编译器命令中同时使用 `-xlang` 选项和 `-xlic_lib` 选项。如果要使用 `-xlang` 且需要在 Sun 性能库中进行链接，应改用 `-library=sunperf`。

## A.2.129.3 警告

请勿将 `-xnolib` 与 `-xlang` 一起使用。

如果要将并行的 Fortran 对象与 C++ 对象混合，链接行必须指定 `-mt` 标志。

## A.2.129.4 另请参见

`-library` 和 `-staticlib`

## A.2.130 -xldscope={*v*}

可指定 -xldscope 选项来更改外部符号定义的缺省链接程序作用域。由于更好的隐藏了实现，所以对缺省的更改会产生更快速更安全的共享库和可执行文件。

### A.2.130.1 值

下表列出了 *v* 的可能值。

表 A–36 -xldscope 值

| 值 | 含义 |
|---|---|
| global | 全局链接程序作用域是限制最少的链接程序作用域。对符号的所有引用都绑定到定义符号的第一个动态装入模块中的定义。该链接程序作用域是外部符号的当前链接程序作用域。 |
| symbolic | 符号链接程序作用域比全局链接程序作用域具有更多的限制。将对链接的动态装入模块内符号的所有引用绑定到模块内定义的符号。在模块外部，符号也显示为全局符号。该链接程序作用域对应于链接程序选项 -Bsymbolic。尽管不能将 -Bsymbolic 与 C++ 库一起使用，但可以使用 -xldscope=symbolic，而不会引起问题。有关链接程序的更多信息，请参见 ld(1) 手册页。 |
| hidden | 隐藏链接程序作用域具有比符号和全局链接程序作用域更高的限制。将动态装入模块内的所有引用绑定到该模块内的定义。符号在模块外部是不可视的。 |

### A.2.130.2 缺省值

如果未指定 -xldscope，编译器将假定 -xldscope=global。如果指定了没有任何值的 -xldscope，则编译器就会发出错误。在到达最右边的实例之前，命令行上此选项的多个实例相互覆盖。

### A.2.130.3 警告

如果要使客户端覆盖库中的函数，就必须确保该库生成期间未以内联方式生成该函数。编译器在以下情况下会内联函数：

- 函数名称是使用 -xinline 指定的。
- 如果在 -xO4 或更高级别进行编译（在这种情况下将自动执行内联）。
- 如果使用内联说明符或交叉文件优化。

例如，假定库 ABC 具有缺省的分配器函数，该函数可用于库的客户端，也可在库的内部使用：

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

如果在 -xO4 或更高级别生成库，则编译器将内联库组件中出现的对 ABC_allocator 的调用。如果库的客户端要用定制的版本替换 ABC_allocator，则在调用 ABC_allocator 的库组件中不能进行该替换。最终程序将包括函数的不同版本。

生成库时，用 __hidden 或 __symbolic 说明符声明的库函数可以内联生成。假定这些说明符不被客户机覆盖。请参见第 57 页中的 “4.1 链接程序作用域”。

用 __global 说明符声明的库函数不应内联声明，并且应该使用 -xinline 编译器选项来防止内联。

### A.2.130.4 另请参见

-xinline 和 -xO

## A.2.131 -xlibmieee

使 libm 在异常情况下对于数学例程返回 IEEE 754 值。

libm 的缺省行为是兼容 XPG。

### A.2.131.1 另请参见

《数值计算指南》

## A.2.132 -xlibmil

内联选定的 libm 数学库例程以进行优化。

---

**注 –** 该选项不影响 C++ 内联函数。

---

此选项为 libm 例程选择内联模板，从而针对当前使用的浮点选项和平台生成执行速度最快的可执行文件。

### A.2.132.1 交互

–fast 选项隐含了该选项。

### 另请参见

-fast 和《数值计算指南》。

## A.2.133 –xlibmopt

使用优化的数学例程库。使用此选项时，必须通过指定 `-fround=nearest` 来使用缺省的舍入模式。

此选项使用经过了性能优化的数学例程库，通常情况下，生成的代码运行速度较快。这样生成的代码可能与普通数学库生成的代码稍有不同，不同之处通常在最后一位上。

该库选项在命令行上的顺序并不重要。

### A.2.133.1 交互

`–fast` 选项隐含了该选项。

### A.2.133.2 另请参见

`–fast`、`–xnolibmopt` 和 `-fround`

## A.2.134 –xlic_lib=sunperf

已过时，不使用。改为指定 `-library=sunperf`。有关更多信息，请参见第 188 页中的“A.2.49 -library=*l*[ ,*l*...]”。

## A.2.135 –xlicinfo

编译器忽略此选项且不显示任何提示。

## A.2.136 -xlinkopt[= *level*]

（**仅限** *SPARC*）指示编译器在对目标文件进行优化的基础上对生成的可执行文件或动态库执行链接时优化。这些优化在链接时通过分析二进制目标代码来执行。虽然未重写目标文件，但生成的可执行代码可能与初始目标代码不同。

必须至少在部分编译命令中使用 `-xlinkopt`，才能使 `-xlinkopt` 在链接时有效。优化器仍可以对未使用 `-xlinkopt` 进行编译的二进制目标文件执行部分受限的优化。

`-xlinkopt` 优化出现在编译器命令行上的静态库代码，但会跳过出现在命令行上的共享（动态）库代码而不对其进行优化。生成共享库时（使用 `-G` 进行编译），也可以使用 `-xlinkopt`。

### A.2.136.1 值

**级别**设置执行的优化级别，必须为 0、1 或 2。下表列出了优化级别：

表 A–37 -xlinkopt 值

| 值 | 含义 |
|---|---|
| 0 | 禁用链接优化器（缺省设置）。 |
| 1 | 在链接时根据控制流分析执行优化，包括指令高速缓存着色和分支优化。 |
| 2 | 在链接时执行附加的数据流分析，包括无用代码删除和地址计算简化。 |

如果在不同的步骤中编译，-xbinopt 必须同时出现在编译和链接步骤中：

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

请注意，仅当编译器链接时才使用级别参数。在该示例中，即使编译二进制目标文件时使用的是隐含的级别 1，链接优化器的级别仍然是 2。

### A.2.136.2 缺省值

指定 -xlinkopt 时若不带级别参数，则表示 -xlinkopt=1。

### A.2.136.3 交互

当编译整个程序并且使用分析反馈时，该选项才最有效。分析功能会展示代码中最常用和最不常用的部分，并指示优化器相应地进行处理。这对大型应用程序尤为重要，因为在链接时执行代码优化放置可降低指令高速缓存未命中数。此选项的常见用法如下所示：

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

有关使用分析反馈的详细信息，请参见第 264 页中的 “A.2.164 –xprofile=*p*”。

### A.2.136.4 警告

使用 -xlinkopt 编译时，请不要使用 -zcombreloc 链接程序选项。

注意，使用该选项编译会略微延长链接的时间，目标文件的大小也会增加，但可执行文件的大小保持不变。使用 -xlinkopt 和 -g 编译会将调试信息包括在内，从而增加了可执行文件的大小。

## A.2.137 -xloopinfo

此选项显示哪些循环是并行化的，通常与 -xautopar 选项一起使用。

## A.2.138 –xM

仅对指定的 C++ 程序运行 C++ 预处理程序，请求此预处理程序生成 makefile 相关项并将结果发送到标准输出。有关 make 文件和相关项的详细信息，请参见 make(1) 手册页。

但是，-xM 只报告包含的头文件的依赖性，而不报告关联的模板定义文件的依赖性。可以使用 makefile 中的 .KEEP_STATE 功能在 make 实用程序创建的 .make.state 文件中生成所有依赖性。

### A.2.138.1 示例

以下示例：

```
#include <unistd.h>
void main(void)
{}
```

生成的输出如下：

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

### A.2.138.2 交互

如果指定 -xM 和 -xMF，编译器会将所有 makefile 依赖性信息写入使用 -xMF 指定的文件。每次预处理程序向此文件写入时即将其覆盖。

### A.2.138.3 另请参见

有关 makefile 和相关项的详细信息，请参见 make(1S) 手册页。

## A.2.139 -xM1

与 –xM 一样，生成 makefile 相关项，只是它不报告 /usr/include 头文件的相关项，也不报告编译器提供的头文件的相关项。

如果指定 -xM1 和 -xMF，编译器会将所有 makefile 依赖性信息写入使用 -xMF 指定的文件。每次预处理程序向此文件写入时即将其覆盖。

## A.2.140 -xMD

像 -xM 一样生成 makefile 依赖性，但编译继续。-xMD 为从 -o 输出文件名（如果指定）或输入源文件名派生的 makefile 依赖性信息生成一个输出文件，替换（或添加）后缀为 .d 的文件名。如果指定了 -xMD 和 -xMF，预处理程序会将所有 makefile 相关项信息写入到使用 -xMF 指定的文件中。不允许使用 -xMD -xMF 或 -xMD -o *filename* 来编译多个源文件，否则会生成错误。如果已存在依赖性文件，将覆盖该文件。

## A.2.141 -xMF

此选项用于指定 makefile 依赖性输出的文件。不能在一个命令行上使用 -xMF 为多个输入文件指定单独的文件名。不允许使用 -xMD -xMF 或 -xMMD -xMF 编译多个源文件，否则会生成错误。如果已存在依赖性文件，将覆盖该文件。

## A.2.142 -xMMD

此选项用于生成不包含系统头文件的 makefile 依赖性。此选项提供的功能与 -xM1 的功能相同，但是编译会继续。-xMMD 为从 -o 输出文件名（如果指定）或输入源文件名派生的 makefile 相关项信息生成一个输出文件，替换（或添加）后缀为 .d 的文件名。如果您指定 -xMF，则编译器将改用您提供的文件名。不允许使用 -xMMD -xMF 或 -xMMD -o *filename* 来编译多个源文件，否则会生成错误。如果已存在依赖性文件，将覆盖该文件。

## A.2.143 –xMerge

（**仅限** *SPARC*）将数据段和文本段合并。

目标文件中的数据是只读数据，并在进程之间共享，除非使用 ld-N 进行链接。

三个选项 -xMerge -ztext -xprofile=collect 不应同时使用。-xMerge 会强制将静态初始化的数据存储到只读存储器中，-ztext 禁止在只读存储器中进行依赖于位置的符号重定位，而 -xprofile=collect 会在可写存储器中生成静态初始化的、依赖于位置的符号重定位。

### A.2.143.1 另请参见

ld(1) 手册页

## A.2.144 -xmaxopt[=v]

此选项将 pragma opt 的级别限制为指定级别。*v* 可以为 off、1、2、3、4 或 5。缺省值为 -xmaxopt=off，表示忽略 pragma opt。在不带参数的情况下指定 -xmaxopt 时，缺省为 -xmaxopt=5。

如果同时指定了 -xO 和 -xmaxopt，则使用 -xO 设置的优化级别不得超过 -xmaxopt 值。

## A.2.145 -xmemalign=*ab*

（**仅限** *SPARC*）使用 -xmemalign 选项控制编译器对数据对齐所做的假定。通过控制可能会出现非对齐内存访问的代码和出现非对齐内存访问时的处理程序，可以更轻松的将程序移植到 SPARC。

指定最大假定内存对齐和非对齐数据访问的行为。必须同时为 *a*（对齐）和 *b*（行为）提供值。*a* 指定最大假定内存对齐，*b* 指定未对齐内存访问行为。

对于可在编译时确定对齐的内存访问，编译器会为该数据对齐生成适当的装入/存储指令序列。

对于不能在编译时确定对齐的内存访问，编译器必须假定一个对齐以生成所需的装入/存储序列。

如果运行时的实际数据对齐小于指定的对齐，则未对齐的访问尝试（内存读取或写入）生成一个陷阱。对陷阱的两种可能响应是：

- 操作系统将陷阱转换为 SIGBUS 信号。如果程序无法捕捉到信号，则程序终止。即使程序捕捉到信号，未对齐的访问尝试仍将无法成功。
- 操作系统通过翻译未对齐的访问并将控制返回给程序（仿佛访问已成功正常结束）来处理陷阱。

### A.2.145.1 值

下面列出了 -xmemalign 的对齐值和行为值。

*a* 的值：

1 假定最多 1 字节对齐。

2 假定最多 2 字节对齐。

4 假定最多 4 字节对齐。

8 假定最多 8 字节对齐。

16 假定最多 16 字节对齐。

*b* 的值：

i 解释访问并继续执行。

s 产生信号 SIGBUS。

f 对于 64 位 SPARC 体系结构：为小于或等于 4 的对齐产生信号 SIGBUS。否则，将解释访问并继续执行。

对于所有其他的体系结构，此标志等效于 `i`。

如果要链接到某个已编译的目标文件，并且编译该目标文件时 *b* 的值设置为 `i` 或 `f`，就必须指定 `-xmemalign`。有关在编译时和链接时都必须指定的所有编译器选项的完整列表，请参见第 43 页中的 “3.3.3 编译时选项和链接时选项”。

### A.2.145.2 缺省值

以下缺省值仅适用于未使用 `-xmemalign` 选项时：

- `-xmemalign=8i`，适用于所有 32 位 SPARC 体系结构 (`-m32`)
- `-xmemalign=8s`，适用于所有 64 位 SPARC 体系结构 (`-m64`)

在提供了 `-xmemalign` 选项但未指定值时，缺省值为：

- `-xmemalign=1i`，适于所有体系结构。

### A.2.145.3 示例

下面说明了如何使用 `-xmemalign` 来处理不同的对齐情况。

`-xmemalign=1s` 所有内存访问均未对齐，从而导致陷阱处理非常慢。

`-xmemalign=8i` 在其他情况下正确的代码中可能会发生偶然的、有目的的、未对齐访问。

`-xmemalign=8s` 程序中未发生任何未对齐访问。

`-xmemalign=2s` 要检查可能存在的奇字节访问。

`-xmemalign=2i` 要检查可能存在的奇字节访问并要使程序工作。

## A.2.146 -xmodel=[*a*]

(**仅限** *x86*) `-xmodel` 选项使编译器可以针对 Oracle Solaris x86 平台修改 64 位对象形式，只应在要编译此类对象时指定该选项。

仅当启用了 64 位的 x64 处理器上还指定了 `-m64` 时，该选项才有效。

下表列出了 *a* 的可能值。

**表 A–38** -xmodel 标志

| 值 | 含义 |
|---|---|
| `small` | 此选项可为小模型生成代码，其中执行代码的虚拟地址在链接时已知，并且已知在 0 到 2^31 - 2^24 - 1 的虚拟地址范围内可以找到所有符号。 |

表 A–38 -xmodel 标志 （续）

| 值 | 含义 |
|---|---|
| kernel | 按内核模型生成代码，在该模型中， 所有符号都定义在 2^64 - 2^31 到 2^64 - 2^24 范围内。 |
| medium | 按中等模型生成代码，在该模型中，不对数据段的符号引用范围进行假定。文本段的大小和地址的限制与小型代码模型的限制相同。使用 -m64 编译时，具有大量静态数据的应用程序可能会要求 -xmodel=medium。 |

此选项不累积，因此编译器根据命令行最右侧的 -xmodel 实例设置模型值。

如果未指定 -xmodel，编译器将假定 -xmodel=small。如果指定没有参数的 -xmodel，将出现错误。

不是编译所有转换单元时都需要使用此选项。只有可以确保访问的对象在可访问范围之内，才可编译选择的文件。

您应了解，不是所有的 Linux 系统都支持中等模型。

## A.2.147 –xnolib

禁用与缺省系统库链接。

通常（不含该选项）情况下，C++ 编译器会链接多个系统库以支持 C++ 程序。使用该选项时，用于链接缺省系统支持库的 -l*lib* 不会传递给 ld。

通常情况下，编译器按照以下顺序链接系统支持库：

- 使用缺省的 —compat=5 时，库为：

```
-lCstd -lCrun -lm -lc
```

- 对于 Linux 上的 –compat=g，库为：

```
–lstdc++ –lCrunG3 –lm –lc
```

- 对于 Oracle x86 上的 –compat=g，库为：

```
–lstdc++ –lgcc_s –lCrunG3 –lm –lc
```

- l 选项的顺序非常重要。-lm 选项必须位于 -lc 之前。

**注 –** 如果指定了 -mt 编译器选项，编译器通常先与 -lthread 链接，然后再与 -lm 链接。

要确定在缺省情况下将链接哪些系统支持库，请使用 -dryrun 选项进行编译。例如，以下命令的输出：

```
example% CC foo.cc -m64 -dryrun
```

在输出中显示以下内容：

```
-lCstd -lCrun -lm -lc
```

### A.2.147.1 示例

对于符合 C 应用程序二进制接口的基本编译（即只需要支持 C 的 C++ 程序），请使用以下命令：

```
example% CC -xnolib test.cc –lc
```

要将 libm 静态链接到具有通用体系结构指令集的单线程应用程序中，请使用以下命令：

```
example% CC -xnolib test.cc -lCstd -lCrun -Bstatic -lm -Bdynamic -lc
```

### A.2.147.2 交互

如果指定了 –xnolib，就必须按给定顺序手动链接所有必需的系统支持库。必须最后链接系统支持库。

如果指定了 -xnolib，则忽略 -library。

### A.2.147.3 警告

许多 C++ 语言功能都要求使用 libCrun（标准模式）。

系统支持库的集合不稳定，会因不同的发行版本而更改。

### A.2.147.4 另请参见

–library、–staticlib 和 –l

## A.2.148 –xnolibmil

在命令行上取消 –xlibmil。

将该选项与 –fast 一起使用会忽略与优化数学库链接。

## A.2.149 –xnolibmopt

不使用数学例程库。

### A.2.149.1 示例

在命令行上 –fast 选项后面使用该选项，如下例中所示：

```
example% CC –fast –xnolibmopt
```

## A.2.150 -xnorunpath

与第 194 页中的 “A.2.60 –norunpath”相同

## A.2.151 -xOlevel

指定优化级别，请注意是大写字母 O 后跟数字 1、2、3、4 或 5。通常，程序执行速度取决于优化的级别。优化级别越高，运行时性能越好。不过，较高的优化级别会延长编译时间并生成较大的可执行文件。

在少数情况下，-x02 级别的性能可能比其他优化级别好，而 -x03 也可能胜过 -x04。尝试用每个级别进行编译，以查看您是否会遇到这种少见的情况。

如果优化器运行时内存不足，则会尝试在较低的优化等级上重试当前过程来恢复。优化器会以在 –x0*level* 选项中指定的初始级别恢复执行后续过程。

以下几节介绍了在 SPARC 平台和 x86 平台上这五个 -x0 *level* 优化级别如何运行。

### A.2.151.1 值

**在 SPARC 平台上：**

- –x01 只执行最小量的优化 (peephole)，也称为 postpass，即汇编级优化。除非使用 -x02 或 -x03 导致编译时间过长，或交换空间不足，否则请勿使用 -x01。
- –x02 执行基本的局部和全局优化，包括：
  - 归纳变量消除
  - 局部和全局的通用子表达式消除
  - 代数运算简化
  - 复制传播
  - 常量传播
  - 非循环变体优化
  - 寄存器分配
  - 基本块合并
  - 尾部递归消除
  - 终止代码消除
  - 尾部调用消除
  - 复杂表达式扩展

    该级别不优化外部变量或间接变量的引用或定义。

–x03 除了执行在 –x02 级别所执行的优化外，还会对外部变量的引用和定义进行优化。该级别不跟踪指针赋值的结果。编译未通过 `volatile` 适当保护的设备驱动程序或修改来自信号处理程序中的外部变量的程序时，请使用 `-xO2`。通常，使用此级别时会增加代码大小，除非将其与 `-xspace` 选项结合使用。

- –x04 除了执行 –x03 优化外，还自动内联同一文件中的多个函数。自动内联通常会提高执行速度，但有时却会使速度变得更慢。通常，使用此级别时会增加代码大小，除非将其与 `-xspace` 选项结合使用。
- –x05 生成最高级别的优化。它只适用于占用大量计算机时间的小部分程序。该级别采用了占用更多编译时间或无法在某种程度上减少执行时间的优化算法。如果使用分析反馈执行该级别上的优化，则更容易提高性能。请参见第 264 页中的 “A.2.164 –xprofile=*p*”。

**在 x86 平台上：**

- –x01 执行基本优化。其中包括代数运算简化、寄存器分配、基本块合并、终止代码和存储消除以及 peephole 优化。
- –x02 执行局部通用子表达式消除、局部复制和常量传播、尾部递归消除以及级别 1 执行的优化。
- –x03 执行全局通用子表达式的消除、全局复制和常量传播、循环长度约简、归纳变量消除、循环变体优化以及级别 2 执行的优化。
- –x04 会自动内联同一文件中的多个函数，并执行级别为 3 时执行的优化。自动内联通常会提高执行速度，但有时却会使速度变得更慢。该级别还释放了通用的框架指针注册 (ebp)。通常该级别会增加代码的大小。
- –x05 生成最高级别的优化。该级别采用了占用更多编译时间或无法在某种程度上减少执行时间的优化算法。

### A.2.151.2 交互

如果使用 `-g` 或 `-g0` 且优化级别是 `-xO3` 或更低，编译器会为近乎完全优化提供尽可能多的符号信息。

如果使用 `-g` 或 `-g0` 且优化级别是 `-xO4` 或更高，编译器会为完全优化提高尽可能多的符号信息。

使用 `-g` 进行调试不会抑制 –x0*level*，但 –x0*level* 会对 –g 造成一些限制。例如，–x0*level* 选项会降低调试的作用，因此无法显示 `dbx` 中的变量，但仍可使用 `dbx where` 命令获取符号回溯。有关更多信息，请参见《使用 dbx 调试程序》。

`-xipo` 选项只有与 `-xO4` 或 `-xO5` 一起使用时才有效。

优化级别低于 `-xO3` 时，`-xinline` 选项不起作用。优化级别为 `-xO4` 时，优化器会决定应该内联哪些函数，而不管是否指定了 `-xinline` 选项。优化级别为 `-xO4` 时，编译器还会尝试确定内联哪些函数可以提高性能。如果使用 `-xinline` 强制内联函数，实际上可能会降低性能。

### A.2.151.3 缺省值

缺省为不优化。不过，只有不指定优化级别时才可能使用缺省设置。如果指定了优化级别，则没有任何选项可用来关闭优化。

如果尝试不设置优化级别，请不要指定任何隐含优化级别的选项。例如，`-fast` 是将优化级别设置为 `-xO5` 的宏选项。隐含优化级别的所有其他选项都会发出优化已设置的警告消息。不使用任何优化来编译的方法是从命令行删除所有选项或创建指定优化级别的文件。

### A.2.151.4 警告

如果在 –x03 或 –x04 级别上优化多个非常大的过程（一个过程有数千行代码），优化器会需要过多内存。在这些情况下，机器的性能就会降低。

为了防止性能降低，请使用 `limit` 命令限制单一进程可用的虚拟内存大小。请参见 `csh`(1) 手册页。例如，将虚拟内存限制为 4 GB：

```
example% limit datasize 4G
```

如果它达到 4 GB 的数据空间，该命令会使优化器尝试恢复。

限制不能大于机器总的可用交换空间，而且要足够的小以允许在大型编译的过程中机器可以正常使用。

数据大小的最佳设置取决于要求的优化程度、真实内存和可用虚拟内存的大小。

要查找实际的交换空间，请输入：**`swap– l`**

要查找实际的真实内存，请输入：**`dmesg | grep mem`**

### A.2.151.5 另请参见

`-xldscope` –`fast`、 –`xprofile=`*p* 和 `csh`(1) 手册页

## A.2.152 -xopenmp[= i]

使用 `-xopenmp` 选项可通过 OpenMP 指令进行显式并行化。

### A.2.152.1 值

下表列出了 *i* 的值。

表 A–39 -xopenmp 值

| 值 | 含义 |
| --- | --- |
| parallel | 启用 OpenMP pragma 的识别。在 -xopenmp=parallel 时，最低优化级别是 -xO3。如有必要，编译器将优化级别从较低级别更改为 -xO3，并发出警告。<br>此标志还定义处理器标记 _OPENMP。 |
| noopt | 启用 OpenMP pragma 的识别。如果优化级别低于 -O3，编译器将不会提高优化级别。<br>如果显式将优化级别设置为低于 -O3（如 CC -O2 -xopenmp=noopt），编译器会发出错误。如果没有使用 -xopenmp=noopt 指定优化级别，则会识别 OpenMP pragma，并相应地对程序进行并行处理，但不进行优化。<br>此标志还定义处理器标记 _OPENMP。 |
| none | 此标志是缺省标志，它禁止识别 OpenMP pragma。它不更改编译的优化级别，也不预定义任何预处理程序标记。 |

### A.2.152.2 缺省值

如果未指定 -xopenmp，则编译器缺省使用 -xopenmp=none。

如果指定了不带参数的 -xopenmp，则编译器缺省使用 -xopenmp=parallel。

### A.2.152.3 交互

如果使用 dbx 调试 OpenMP 程序，那么编译时选用 -g 和 -xopenmp=noopt 可以在并行区设置断点并显示变量内容。

使用 OMP_NUM_THREADS 环境变量可指定在运行 OpenMP 程序时要使用的线程数。如果未设置 OMP_NUM_THREADS，则缺省使用的线程数为 2。要使用多个线程，请将 OMP_NUM_THREADS 设置为更大的值。将 OMP_NUM_THREADS 设置为 1，则会仅使用一个线程运行。通常情况下，将 OMP_NUM_THREADS 设置为运行系统上可用的虚拟处理器数，该值可使用 Oracle Solaris psrinfo(1) 命令确定。有关更多信息，请参见《Oracle Solaris Studio OpenMP API 用户指南》。

要启用嵌套并行操作，必须将 OMP_NESTED 环境变量设置为 TRUE。缺省情况下，禁用嵌套并行操作。有关详细信息，请参见《Oracle Solaris Studio OpenMP API 用户指南》。

### A.2.152.4 警告

在以后的发行版中，-xopenmp 的缺省值可能会更改。可以通过显式指定适当的优化来避免警告消息。

如果在不同的步骤中进行编译和链接，请在编译步骤和链接步骤中都指定 `-xopenmp`。如果要生成共享对象，这很重要。用于编译可执行文件的编译器的版本不得比使用 `-xopenmp` 生成 `.so` 的编译器低。这在编译包含 OpenMP 指令的库时尤其重要。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 43 页中的 “3.3.3 编译时选项和链接时选项”。

为了取得最佳的性能，请确保在系统上安装了最新的 OpenMP 运行时库 `libmtsk.so`。

#### A.2.152.5 另请参见

有关用于生成多重处理应用程序的 OpenMP Fortran 95、C 和 C++ 应用程序接口 (application program interface, API) 的完整摘要，请参见《Oracle Solaris Studio OpenMP API 用户指南》。

## A.2.153 -xpagesize=*n*

为堆栈和堆设置首选页面大小。

#### A.2.153.1 值

在 SPARC 上有效值包括：`4K`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G` 或 `default`。

在 x86/x64 上有效值包括：`4K`、`2M`、`4M`、`1G` 或 `default`。

必须指定适于目标平台的有效页面大小。如果不指定有效的页面大小，运行时将忽略该请求，且不显示任何提示。

在 Oracle Solaris 操作系统中使用 `getpagesize`(3C) 命令可以确定页面中的字节数。Solaris 操作系统不保证支持页面大小请求。可以使用 `pmap`(1) 或 `meminfo`(2) 来确定目标平台的页面大小。

---

**注 –** 使用该选项进行编译与使用等效的选项将 `LD_PRELOAD` 环境变量设置为 `mpss.so.1`，或在运行程序之前使用等效的选项运行 Oracle Solaris 命令 `ppgsz`(1) 具有相同的效果。有关详细信息，请参见 Oracle Solaris 手册页。

---

#### A.2.153.2 缺省值

如果指定 `-xpagesize=default`，Oracle Solaris 操作系统将设置页面大小。

#### A.2.153.3 扩展

此选项是用于 `-xpagesize_heap` 和 `-xpagesize_stack` 的宏。这两个选项与 `-xpagesize` 接受相同的参数：`4k`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G` 或 `default`。可以通过指定 `-xpagesize` 为它们设置相同值，也可以分别为它们指定不同的值。

### A.2.153.4 警告

除非在编译和链接时使用，否则 -xpagesize 选项不会生效。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 43 页中的 “3.3.3 编译时选项和链接时选项”。

## A.2.154 -xpagesize_heap=*n*

为堆设置内存页面大小。

### A.2.154.1 值

*n* 可以是 4k、8K、 64K、512K、2M、4M、 32M、256M、2G、16G 或 default。必须指定适于目标平台的有效页面大小。如果不指定有效的页面大小，运行时将忽略该请求，且不显示任何提示。

在 Oracle Solaris 操作系统中使用 getpagesize(3C) 命令可以确定页面中的字节数。Solaris 操作系统不保证支持页面大小请求。可以使用 pmap(1) 或 meminfo(2) 来确定目标平台的页面大小。

---

**注 –** 使用该选项进行编译与使用等效的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1，或在运行程序之前使用等效的选项运行 Oracle Solaris 命令 ppgsz(1) 具有相同的效果。有关详细信息，请参见 Oracle Solaris 手册页。

---

### A.2.154.2 缺省值

如果指定 -xpagesize_heap=default，Oracle Solaris 操作系统将设置页面大小。

### A.2.154.3 警告

除非在编译和链接时使用，否则 -xpagesize_heap 选项不会生效。

## A.2.155 -xpagesize_stack=*n*

为堆栈设置内存页面大小。

### A.2.155.1 值

*n* 可以是 4k、8K、 64K、512K、2M、4M、 32M、256M、2G、16G 或 default。必须指定适于目标平台的有效页面大小。如果不指定有效的页面大小，运行时将忽略该请求，且不显示任何提示。

在 Oracle Solaris 操作系统中使用 getpagesize(3C) 命令可以确定页面中的字节数。Oracle Solaris 操作系统不保证支持页面大小请求。可以使用 pmap(1) 或 meminfo(2) 来确定目标平台的页面大小。

**注 –** 使用该选项进行编译与使用等效的选项将 `LD_PRELOAD` 环境变量设置为 `mpss.so.1`，或在运行程序之前使用等效的选项运行 Oracle Solaris 命令 `ppgsz(1)` 具有相同的效果。有关详细信息，请参见 Oracle Solaris 手册页。

### A.2.155.2 缺省值

如果指定 `-xpagesize_stack=default`，Oracle Solaris 操作系统将设置页面大小。

### A.2.155.3 警告

除非在编译和链接时使用，否则 `-xpagesize_stack` 选项不会生效。

## A.2.156 -xpch=v

该编译器选项激活了预编译头文件特性。预编译头文件的作用是减少源代码共享同一组 include 文件的应用程序的编译时间，而且这些 include 文件往往包含大量的源代码。编译器首先从一个源文件收集一组头文件信息，然后在重新编译该源文件或者其他有同样头文件的源文件时就可以使用这些收集到的信息。编译器收集的信息存储在预编译头文件中。要使用该功能，需要指定 `-xpch` 和 `-xpchstop` 选项，并使用 `#pragma hdrstop` 指令。

### A.2.156.1 创建预编译头文件

指定 `-xpch=`*v* 时，*v* 可以是 `collect:`*pch-filename* 或 `use:`*pch-filename*。首次使用 `-xpch` 时，必须指定 `collect` 模式。指定 `-xpch=collect` 的编译命令只能指定一个源文件。在以下示例中，`-xpch` 选项根据源文件 `a.cc` 创建名为 `myheader.Cpch` 的预编译头文件：

```
CC -xpch=collect:myheader a.cc
```

有效的预编译头文件总是有后缀 `.Cpch`。在指定 *pch-filename* 时，您可以自己添加后缀，也可以让编译器为您添加。例如，如果指定 `cc -xpch=collect:foo a.cc`，则预编译头文件称为 `foo.Cpch`。

在创建预编译头文件时，请选取包含所有源文件之间 include 文件通用序列的源文件，预编译头文件与这些源文件一起使用。include 文件的共同序列在这些源文件之间必须是一样的。请记住，在 `collect` 模式中只能使用一个源文件名值。例如，`CC -xpch=collect:foo bar.cc` 有效，而 `CC -xpch=collect:foo bar.cc foobar.cc` 无效，因为它指定了两个源文件。

#### 使用预编译头文件

可指定 `-xpch=use:`*pch-filename* 以使用预编译头文件。您可以将 include 文件同一序列中任意数量的源文件指定为用于创建预编译头文件的源文件。例如，在 `use` 模式中命令类似于：`CC -xpch=use:foo.Cpch foo.c bar.cc foobar.cc`。

如果以下条件都成立，则只能使用现有的预编译的头文件。如果以下任意条件不成立，则应重新创建预编译头文件：

- 用于访问预编译头文件的编译器与创建预编译头文件的编译器相同。编译器的一个版本创建的预编译头文件可能无法用于另一版本（包括安装的修补程序产生的差异）。
- 除 `-xpch` 选项之外，用 `-xpch=use` 指定的编译器选项必须与创建预编译头文件时指定的选项相匹配。
- 用 `-xpch=use` 指定的包含头文件的集合与创建预编译头文件时指定的头文件集合是相同的。
- 用 `-xpch=use` 指定的包含头文件的内容与创建预编译头文件时指定的包含头文件的内容是相同的。
- 当前目录（即发生编译并尝试使用给定预编译头文件的目录）与创建预编译头文件所在的目录相同。
- 在用 `-xpch=collect` 指定的文件中预处理指令（包括 `#include`）的初始序列，与在用 `-xpch=use` 指定的文件中预处理指令的序列相同。

要在多个源文件间共享预编译头文件，这些源文件必须共享一组共同的 include 文件（按其初始标记序列）。该初始标记序列称为**活前缀**。活前缀必须在使用相同预编译头文件的所有源文件中解释一致。

源文件的活前缀只能包含注释和以下任意预处理程序指令：

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

以上任何指令都可以引用宏。`#else`、`#elif` 和 `#endif` 指令必须在活前缀内匹配。

在共享预编译头文件的每个文件的活前缀中，每个相应的 `#define` 和 `#undef` 指令都必须引用相同的符号。例如，每个 `#define` 必须引用同一个值。这些指令在每个活前缀中出现的顺序也必须相同。每个相应 pragma 也必须相同，并且必须按相同顺序出现在共享预编译头文件的所有文件中。

并入预编译头文件的头文件不得违反以下约束。这里没有定义对违反上述约束的程序的编译结果。

- 头文件一定不要包含函数和变量定义。
- 头文件不得使用 `__DATE__` 和 `__TIME__`。使用预处理宏会产生无法预料的结果。
- 头文件不得包含 `#pragma hdrstop`。
- 头文件的活前缀中不得使用 `__LINE__` 和 `__FILE__`。可以在包含的头文件中使用 `__LINE__` 和 `__FILE__`。

### 如何修改 makefile

本节介绍了为将 -xpch 合并到生成的程序中而对 makefile 进行修改的几种可能方法。

- 可以通过使用辅助变量 CCFLAGS 以及 make 和 dmake 的 KEEP_STATE 功能使用隐式 make 规则。预编译头文件是在独立的步骤中生成的。

```
.KEEP_STATE:
CCFLAGS_AUX = -O etc
CCFLAGS = -xpch=use:shared $(CCFLAGS_AUX)
shared.Cpch: foo.cc
     $(CCC) -xpch=collect:shared $(CCFLAGS_AUX) foo.cc
a.out: foo.o ping.o pong.o
     $(CCC) foo.o ping.o pong.o
```

您还可以定义自己的编译规则，而无不是尝试使用辅助变量 CCFLAGS。

```
.KEEP_STATE:
.SUFFIXES: .o .cc
%.o:%.cc shared.Cpch
        $(CCC) -xpch=use:shared $(CCFLAGS) -c $<
shared.Cpch: foo.cc
        $(CCC) -xpch=collect:shared $(CCFLAGS) foo.cc -xe
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

- 可以通过常规编译顺便生成预编译头文件，而无需使用 KEEP_STATE，但该方法要求使用显式编译命令。

```
shared.Cpch + foo.o: foo.cc bar.h
        $(CCC) -xpch=collect:shared foo.cc $(CCFLAGS) -c
ping.o: ping.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared ping.cc $(CCFLAGS) -c
pong.o: pong.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared pong.cc $(CCFLAGS) -c
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

### A.2.156.2 另请参见

- 第 257 页中的 “A.2.157 -xpchstop=*file* ”
- 第 288 页中的 “B.2.9 #pragma hdrstop”

## A.2.157 -xpchstop=file

可使用 -xpchstop=*file* 选项指定在使用 -xpch 选项创建预编译头文件时要考虑的最后一个 include 文件。在命令行上使用 -xpchstop 与将 hdrstop pragma 置于第一个（引用使用 cc 命令指定的每个源文件中的 file 的）包含指令之后等效。

在以下示例中，-xpchstop 选项指定了预编译头文件的活前缀以包含 projectheader.h 结束。因此，privateheader.h 不是活前缀的一部分。

```
example% cat a.cc
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% CC -xpch=collect:foo.Cpch a.cc -xpchstop=projectheader.h -c
```

### A.2.157.1 另请参见

`-xpch` 和 `pragma hdrstop`

## A.2.158 -xpec[={yes|no}]

（**仅限** *Solaris*）生成可移植的可执行代码 (Portable Executable Code, PEC) 库。此选项将程序中间表示置于目标文件和二进制文件中。该二进制文件可在以后用于调整和故障排除。

使用 `-xpec` 生成的二进制文件通常会比未使用 `-xpec` 生成的二进制文件大五到十倍。

如果不指定 `-xpec`，则编译器会将其设置为 `-xpec=no`。如果您指定 `-xpec`，但不提供标志，则编译器会将其设置为 `-xpec=yes`。

## A.2.159 –xpg

编译以便使用 `gprof` 分析器进行分析。

`-xpg` 选项可编译自分析代码，以便收集可使用 `gprof` 分析的数据。该选项调用运行时记录机制，该机制会在程序正常终止时生成 `gmon.out` 文件。

---

**注 –** 如果指定了 `-xpg`，则 `-xprofile` 将没什么用处。两者不能准备或使用对方提供的数据。

---

在 64 位 Solaris 平台上，使用 `prof(1)` 或 `gprof(1)` 生成分析，在 32 位 Solaris 平台上，则只使用 `gprof` 生成分析，分析中包含大概的用户 CPU 时间。这些时间来自主可执行文件中的例程以及共享库中例程（链接可执行文件时将共享库指定为链接程序参数）的 PC 样例数据。其他共享库（在进程启动后使用 `dlopen(3DL)` 打开的库）不进行分析。

在 32 位 Solaris 系统中，使用 `prof(1)` 生成的分析仅限于可执行文件中的例程。32 位共享库通过用 `-xpg` 和 `gprof(1)` 链接可执行程序可以进行分析。

在 x86 系统中，`-xpg` 与 `-xregs=frameptr` 不兼容，这两个选项不应一起使用。还请注意，`-fast` 中包括 `-xregs=frameptr`。

Oracle Solaris 10 软件不包括使用 -p 编译的系统库。因此，在 Solaris 10 平台上收集的分析不包含系统库例程的调用计数。

### A.2.159.1 警告

如果分别进行编译和链接，且使用 –xpg 进行编译，应确保使用 –xpg 进行链接。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 43 页中的 “3.3.3 编译时选项和链接时选项”。

使用 -xpg 编译以便进行 gprof 分析的二进制文件不应与 binopt(1) 一起使用，因为它们不兼容并可能导致内部错误。

### A.2.159.2 另请参见

–xprofile=*p*、analyzer(1) 手册页和性能分析器手册。

## A.2.160 -xport64[=(*v*)]

使用此选项可以帮助调试要移植到 64 位环境的代码。具体来说，该选项会针对遇到的问题发出警告，例如：类型（包括指针）的截断，符号扩展以及位包装更改，将代码从诸如 V8 的 32 位体系结构移植到诸如 V9 的 64 位体系结构时经常会遇到这些问题。

除非您也在 64 位模式 (–m64) 下进行编译，否则此选项没有任何效果。

### A.2.160.1 值

下表列出了*v* 的有效值：

**表 A–40** -xport64 值

| 值 | 含义 |
|---|---|
| no | 将代码从 32 位环境移植到 64 位环境时，不会生成与该代码移植有关的任何警告。 |
| implicit | 只生成隐式转换的警告。显式强制类型转换出现时不生成警告。 |
| full | 将代码从 32 位环境移植到 64 位环境时，生成了与该代码移植有关的所有警告。其中包括对 64 位值的截断警告、根据 ISO 值的保存规则对 64 位的符号扩展，以及对位字段包装的更改。 |

### A.2.160.2 缺省值

如果未指定 -xport64，则缺省值为 -xport64=no。如果指定了 -xport64 但未指定标志，则缺省值为 -xport64=full。

### A.2.160.3 示例

本节提供了可以导致类型截断、符号扩展和对位包装更改的代码示例。

## 检查 64 位值的截断

在移植到诸如 SPARC V9 的 64 位体系结构时，数据可能会被截断。截断可能会因赋值（初始化时）或显式强制类型转换而隐式地发生。两个指针的差异在于 typedef ptrdiff_t，它在 32 位模式下是 32 位整型，而在 64 位模式下是 64 位整型。将较长的整型截断为较小的整型会生成警告，如下示例所示。

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; //warn

example% CC -c -m64 -Qoption ccfe -xport64=full test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

可使用 -xport64=implicit 禁用 64 位编译模式下显式强制类型转换导致数据截断时出现截断警告。

```
example% CC -c -m64 -Qoption ccfe -xport64=implicit test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

在移植到 64 位架构过程中出现的另一个常见问题是指针的截断。该问题在 C++ 中始终是错误。如果指定了 -xport64，导致此类截断的操作（如将指针强制转换为整型）可能会导致在 64 位 SPARC 体系结构中出现错误诊断。

```
example% cat test2.c
char* p;
int main() {
  p =(char*) (((unsigned int)p) & 0xFF); // -m64 error
  return 0;
}
example% CC -c -m64 -Qoption ccfe -xport64=full test2.c
"test2.c", line 3: Error: Cannot cast from char* to unsigned.
1 Error(s) detected.
example%
```

## 检查符号扩展

还可以使用 -xport64 选项来检查这种情况：标准 ISO C 值保留规则允许在无符号整型的表达式中进行带符号整数值的符号扩展。这种符号扩展会产生细微的运行时错误。

```
example% cat test3.c
int i= -1;
void promo(unsigned long l) {}

int main() {
    unsigned long l;
    l = i;  // warn
    promo(i);       // warn
```

```
}
example% CC -c -m64 -Qoption ccfe -xport64=full test3.c
"test3.c", line 6: Warning: Sign extension from "int" to 64-bit integer.
"test3.c", line 7: Warning: Sign extension from "int" to 64-bit integer.
2 Warning(s) detected.
```

### 检查位字段包装的更改

可使用 -xport64 生成对长位字段的警告。出现这种位字段时，位字段的包装可能会显著更改。在成功移植到 64 位架构之前，依赖于假定的任何程序都需要重新检查，该假定与包装位字段的方法有关。

```
example% cat test4.c
#include <stdio.h>

union U {
   struct S {
       unsigned long b1:20;
       unsigned long b2:20;
   } s;

   long buf[2];
} u;

int main() {
   u.s.b1 = 0XFFFFF;
   u.s.b2 = 0XFFFFF;
   printf(" u.buf[0] = %lx u.buf[1] = %lx\n", u.buf[0], u.buf[1]);
   return 0;
}
example%
```

64 位 SPARC 系统 (-m64) 上的输出：

```
example% u.buf[0] = fffffffff000000 u.buf[1] = 0
```

#### A.2.160.4 警告

请注意，仅当使用 -m64 以 64 位模式编译时，才会生成警告。

#### A.2.160.5 另请参见

第 191 页中的 “A.2.50 -m32|-m64”

## A.2.161 -xprefetch[= a[,a...]]

在支持预取的体系结构上启用预取指令。

显式预取只应在度量支持的特殊环境下使用。

下表列出了 *a* 的可能值。

表 A–41 -xprefetch 值

| 值 | 含义 |
|---|---|
| `auto` | 启用预取指令的自动生成 |
| `no%auto` | 禁用预取指令的自动生成 |
| `explicit` | (SPARC) 启用显式预取宏 |
| `no%explicit` | (SPARC) 禁用显式预取宏 |
| `latx:`*factor* | 根据指定的因子，调整编译器假定的“预取到装入”和“预取到存储”延迟。只能将此标志与 `-xprefetch=auto` 结合使用。该因子必须是正浮点数或整数。 |
| `yes` | 已废弃，不使用。改用 `-xprefetch=auto,explicit`。 |
| `no` | 已废弃，不使用。改用 `-xprefetch=no%auto,no%explicit`。 |

使用 `-xprefetch` 和 `-xprefetch=auto` 时，编译器就可以将预取指令自由插入到它生成的代码中。该操作会提高支持预取的体系结构的性能。

如果是在大型多处理器上运行计算密集型代码，则使用 `-xprefetch=latx:`*factor* 可以提高性能。该选项指示代码生成器按照指定的因子调节在预取及其相关的装入或存储之间的缺省延迟时间。

预取延迟是从执行预取指令到所预取的数据在高速缓存中可用那一刻之间的硬件延迟。在确定发出预取指令到发出使用所预取数据的装入或存储指令之间的间隔时，编译器就采用预取延迟值。

---

**注 –** 在预取和装入之间采用的延迟可能与在预取和存储之间采用的延迟不同。

---

编译器可以在众多计算机与应用程序间调整预取机制，以获得最佳性能。这种调整并非总能达到最优。对于占用大量内存的应用程序，尤其是要在大型多处理器上运行的应用程序，可以通过增加预取延迟值来提高性能。要增加值，请使用大于 1 的因子。介于 .5 和 2.0 之间的值最有可能提供最佳性能。

对于数据集完全位于外部高速缓存中的应用程序，可以通过减小预取延迟值来提高性能。要减小值，请使用小于 1 的因子。

要使用 `-xprefetch=latx:`*factor* 选项，请首先使用接近 1.0 的因子值并对应用程序运行性能测试。然后适当增加或减小该因子，并再次运行性能测试。继续调整因子并运行性能测试，直到获得最佳性能。以很小的增量逐渐增加或减小因子时，前几步中不会看到性能差异，之后会突然出现差异，然后再趋于稳定。

### A.2.161.1 缺省值

缺省值为 -xprefetch=auto,explicit。此缺省值会对实质上具有非线性内存访问模式的应用程序造成负面影响。要覆盖该缺省值，请指定 -xprefetch=no%auto,no%explicit。

除非使用参数 no%auto 或参数 no 进行显式覆盖，否则使用缺省值 auto。例如，-xprefetch=explicit 与 -xprefetch=explicit,auto 相同。

除非使用参数 no%explicit 或 no 进行显式覆盖，否则使用缺省值 explicit。例如，-xprefetch=auto 与 -xprefetch=auto,explicit 相同。

如果仅指定了 -xprefetch，则假定为 -xprefetch=auto,explicit。

如果启用了自动预取，但未指定延迟因子，则假定 -xprefetch=latx:1.0。

### A.2.161.2 交互

该选项会累积而不覆盖。

sun_prefetch.h 头文件提供了用于指定显式预取指令的宏。这些预取可能位于对应于宏出现位置的可执行文件中。

要使用显式预取指令，必须在适当的体系结构上，将 sun_prefetch.h 包含进来，并且从编译器命令中排除 -xprefetch 或者使用 -xprefetch、-xprefetch=auto,explicit 或 -xprefetch=explicit。

如果调用宏并包含 sun_prefetch.h 头文件，但指定 -xprefetch=no%explicit，那么显式预取将不会出现在可执行文件中。

仅当启用了自动预取时，使用 latx:*factor* 才有效。除非将 latx:*factor* 与 -xprefetch=auto,latx:*factor* 一起使用，否则它会被忽略。

### A.2.161.3 警告

显式预取只应在度量支持的特殊环境下使用。

因为编译器可以在众多计算机与应用程序间调整预取机制以获得最佳性能，所以仅当性能测试指示性能明显提高时，才应当使用 -xprefetch=latx:*factor*。假定的预取延迟在不同发行版本中是不同的。因此，无论何时切换到不同的发行版本，强烈建议重新测试延迟因子对性能的影响。

## A.2.162 -xprefetch_auto_type= a

其中，*a* 是 [no%]indirect_array_access。

使用此选项可以确定编译器是否以为直接内存访问生成预取的方式，为由选项 -xprefetch_level 指示的循环生成间接预取。

如果不指定 -xprefetch_auto_type 的设置，编译器会将其设置为 -xprefetch_auto_type=no%indirect_array_access。

-xdepend、-xrestrict 和 -xalias_level 等选项会影响计算候选间接预取的主动性，进而影响因更好的内存别名歧义消除信息而自动插入间接预取的主动性。

## A.2.163 -xprefetch_level[= i]

可使用 -xprefetch_level=*i* 选项控制由 -xprefetch=auto 确定的自动插入预取指令的主动性。编译器变得更加主动，也就是说，引入更多预取级别 -xprefetch_level（依次增高）。

-xprefetch_level 取何适当值取决于应用程序的高速缓存未命中次数。-xprefetch_level 值越高，越有可能提高那些高速缓存未命中次数较多的应用程序的性能。

### A.2.163.1 值

*i* 必须是 1、2 或 3，如下表所示。

**表 A–42** -xprefetch_level 值

| 值 | 含义 |
|---|---|
| 1 | 启用预取指令的自动生成。 |
| 2 | 针对其他循环（超过在 -xprefetch_level=1 针对的循环）来插入预取。插入的其他预取可能超过在 -xprefetch_level=1 插入的预取。 |
| 3 | 针对其他循环（超过在 -xprefetch_level=2 针对的循环）来插入预取。插入的其他预取可能超过在 -xprefetch_level=2 插入的预取。 |

### A.2.163.2 缺省值

指定了 -xprefetch=auto 时，缺省值为 -xprefetch_level=1。

### A.2.163.3 交互

仅当使用 -xprefetch=auto 对该选项进行编译，优化级别为 3 或更高 (-xO3)，而且是在支持预取的 64 位 SPARC 平台 (-m64) 上时，该选项才有效。

## A.2.164 –xprofile=p

收集用于分析的数据或使用分析进行优化。

*p* 必须为 collect[:*profdir*]、use[:*profdir*] 或 tcov[:*profdir*]。

此选项可在执行期间收集并保存执行频率数据，然后在后续运行中可以使用该数据来改进性能。对多线程应用程序来讲，分析收集 (Profile collection) 是一种安全的方法。对执行自身多任务处理 (`-mt`) 的程序进行分析可产生准确的结果。只有指定 `-xO2` 或更高的优化级别时，此选项才有效。如果分别执行编译和链接，则链接步骤和编译步骤中必须都出现同一 `-xprofile` 选项。

`collect[:`*profdir*`]` 在 `-xprofile=use` 时优化器收集并保存执行频率，以供将来使用。编译器生成可测量语句执行频率的代码。

`-xMerge`, `-ztext`, 和 `-xprofile=collect` 不应同时使用。`-xMerge` 会强制将静态初始化的数据存储到只读存储器中，`-ztext` 禁止在只读存储器中进行依赖于位置的符号重定位，而 `-xprofile=collect` 会在可写存储器中生成静态初始化的、依赖于位置的符号重定位。

分析目录名 *profdir*（如果指定）是包含已分析的目标代码的程序或共享库在执行时用来存储分析数据的目录路径名。如果 *profdir* 路径名不是绝对路径，在使用选项 `-xprofile=use:`*profdir* 编译程序时将相对于当前工作目录来解释该路径。

如果未使用 `–xprofile=collect:` *prof_dir* 或 `–xprofile=tcov:` *prof_dir* 指定分析目录名，分析数据将在运行时存储在名为 *program*`.profile` 的目录中，其中 *program* 是已分析的进程主程序的基本名称。在这种情况下，可以使用环境变量 `SUN_PROFDATA` 和 `SUN_PROFDATA_DIR` 控制在运行时存储分析数据的位置。如果已设置，分析数据将写入 `$SUN_PROFDATA_DIR/$SUN_PROFDATA` 指定的目录。如果在编译时指定了分析目录名，则 `SUN_PROFDATA_DIR` 和 `SUN_PROFDATA` 在运行时将无效。这些环境变量同样控制由 `tcov` 写入的分析数据文件的路径和名称，如 `tcov`(1) 手册页中所述。

如果未设置这些环境变量，分析数据将写入当前目录中的目录 *profdir*`.profile`，其中 *profdir* 是可执行文件的名称或在 `-xprofile=collect:` *profdir* 标志中指定的名称。如果 *profdir* 已在 `.profile` 中结束，`-xprofile` 不会将 `.profile` 附加到 *profdir* 中。如果多次运行程序，那么执行频率数据会累积在 *profdir*`.profile` 目录中；也就是说，以前执行的输出不会丢失。

如果在不同的步骤中进行编译和链接，应确保使用 `-xprofile=collect` 编译的任何目标文件也使用 `-xprofile=collect` 进行链接。

以下示例在生成程序所在目录的 `myprof.profile` 目录中收集分析数据，然后使用这些分析数据：

```
demo: CC -xprofile=collect:myprof.profile -xO5 prog.cc -o prog
demo: ./prog
demo: CC -xprofile=use:myprof.profile -xO5 prog.cc -o prog
```

以下示例在目录 /bench/myprof.profile 中收集分析数据，然后在优化级别为 -xO5 的反馈编译中使用收集的分析数据：

```
demo: CC -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.cc -o prog
...run prog from multiple locations..
demo: CC -xprofile=use:/bench/myprof.profile
\    -xO5 prog.cc -o prog
```

use[:*profdir*]

通过从使用 –xprofile=collect[:*profdir*] 或 –xprofile=tcov[:*profdir*] 编译的代码中收集的执行频率数据，优化在执行已分析的代码时执行的工作。*profdir* 是一个目录的路径名，该目录包含运行用 –xprofile=collect[:*profdir*] 或 –xprofile=tcov[:*profdir*] 编译的程序所收集的分析数据。

要生成可供 tcov 和 –xprofile=use[:*profdir*] 使用的数据，必须在编译时通过选项 –xprofile=tcov[:*profdir*] 指定分析目录。必须在 –xprofile=tcov:*profdir* 和 –xprofile=use:*profdir* 中指定同一分析目录。为最大限度地减少混淆情况，请将 *profdir* 指定为绝对路径名。

*profdir* 路径名是可选的。如果未指定 *profdir*，将使用可执行二进制文件的名称。如果未指定 -o，将使用 a.out。如果未指定 *profdir*，编译器将查找 *profdir*.profile/feedback 或 a.out.profile/feedback。例如：

```
demo: CC -xprofile=collect -o myexe prog.cc
demo: CC -xprofile=use:myexe -xO5 -o myexe    prog.cc
```

程序是使用以前生成并保存在 feedback 文件中的执行频率数据优化的，此数据是先前执行用 -xprofile=collect 编译的程序时写入的。

除了 -xprofile 选项之外，源文件和其他编译器选项必须与用于编译（该编译过程创建了生成 feedback 文件的编译程序）的相应选项完全相同。编译器的相同版本必须同时用于 collect 生成和 use 生成。

如果用 -xprofile=collect:*profdir* 编译，则必须将相同的分析目录名 *profdir* 用在优化编译中：-xprofile=use:*profdir*。

另请参见 -xprofile_ircache，以了解有关加速 collect 阶段和 use 阶段之间的编译的说明。

tcov[:*profdir*]

使用 tcov(1) 检测目标文件以进行基本块覆盖分析。

如果指定可选的 *profdir* 参数，编译器将在指定位置创建分析目录。该分析目录中存储的数据可通过 tcov(1) 或由编译器通过 -xprofile=use:*profdir* 来使用。如果省略可选的 *profdir* 路径

名，执行已进行分析的程序时将创建分析目录。只能通过 tcov(1) 使用该分析目录中存储的数据。使用环境变量 SUN_PROFDATA 和 SUN_PROFDATA_DIR 可以控制分析目录的位置。

如果 *profdir* 指定的位置不是绝对路径名，则在编译时将相对于编译时的当前工作目录来解释该位置。如果为任何目标文件指定了 *profdir*，则必须为同一程序中的所有目标文件指定同一位置。由 *profdir* 指定位置的目录必须在要执行已进行分析的程序的所有计算机中都可以访问。除非不再需要分析目录中的内容，否则不应删除该目录，因为除非重新编译，否则编译器存储在其中的数据将无法恢复。

如果用 -xprofile=tcov:/test/profdata 编译一个或多个程序的目标文件，编译器会创建一个名为 /test/profdata.profile 的目录并将其用来存储描述已进行分析的目标文件的数据。该同一目录还可在执行时用来存储与已进行分析的目标文件关联的执行数据。

如果名为 myprog 的程序使用 -xprofile=tcov 进行编译并在目录 /home/joe 中执行，将在运行时创建目录 /home/joe/myprog.profile 并将其用来存储运行时分析数据。

## A.2.165 -xprofile_ircache[ =path]

（**仅限** *SPARC*）可以将 -xprofile_ircache[=*path*] 与 -xprofile=collect|use 一起使用，并重新使用 collect 阶段保存的编译数据，以改善 use 阶段的编译时间。

在编译大程序时，由于中间数据的保存，使得 use 阶段的编译时间大大减少。注意，所保存的数据会占用相当大的磁盘空间。

在使用 -xprofile_ircache[=*path*] 时，*path* 会覆盖保存缓存文件的位置。缺省情况下，这些文件会作为目标文件保存在同一目录下。collect 和 use 阶段出现在两个不同目录中时，指定路径很有用。以下示例显示了典型的命令序列：

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out    // run collects feedback data
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

## A.2.166 -xprofile_pathmap

（**仅限** *SPARC*）使用 -xprofile_pathmap= *collect-prefix*:*use-prefix* 选项（同时还指定 -xprofile=use 命令）。以下两个条件都成立且编译器无法找到使用 -xprofile=use 编译的目标文件的分析数据时，使用 -xprofile_pathmap。

- 使用 `-xprofile=use` 编译目标文件所在的目录与先前使用 `-xprofile=collect` 编译目标文件所在的目录不同。
- 目标文件在分析中共享公共基名，但可以根据它们在不同目录中的位置互相区分。

*collect-prefix* 是某个目录树的 UNIX 路径名的前缀，该目录树中的目标文件是使用 `-xprofile=collect` 编译的。

*use-prefix* 是目录树的 UNIX 路径名的前缀，该目录树中的目标文件是使用 `-xprofile=use` 编译的。

如果指定了 `-xprofile_pathmap` 的多个实例，编译器将按照这些实例的出现顺序对其进行处理。将 *-xprofile_pathmap* 实例指定的每个 `use-prefix` 与目标文件路径名进行比较，直至找到匹配的 *use-prefix* 或发现最后一个指定的 *use-prefix* 与目标文件路径名也不匹配。

## A.2.167 -xreduction

对自动并行化中的约简进行循环分析。只有在同时指定了 `-xautopar` 时，此选项才有效。否则，编译器会发出警告。

当启用了约简识别时，编译器会并行化约简，例如点积、最大与最小查找。这些约简产生的舍入与通过非并行化代码获得的舍入不同。

## A.2.168 –xregs=r[ ,r...]

为生成的代码指定寄存器的用法。

*r* 是一个逗号分隔列表，它包含下面的一个或多个子选项：`appl`、`float`、`frameptr`。

用 `no%` 作为子选项的前缀会禁用该子选项。例如：`-xregs=appl,no%float`

请注意，`-xregs` 子选项仅限于特定的硬件平台。

**表 A–43** -xregs 子选项

| 值 | 含义 |
|---|---|
| `appl` | (SPARC) 允许编译器将应用程序寄存器用作临时寄存器来生成代码。应用程序寄存器是：<br>g2、g3 或 g4（在 32 位平台上）<br>g2 或 g3（在 64 位平台上）<br>应使用 `-xregs=no%appl` 编译所有系统软件和库。系统软件（包括共享库）必须为应用程序保留这些寄存器值。这些值的使用将由编译系统控制，而且在整个应用程序中必须保持一致。<br>在 SPARC ABI 中，这些寄存器表示为**应用程序**寄存器。由于需要更少的装入和存储指令，因此使用这些寄存器可提高性能。但是，这样使用可能与某些用汇编代码编写的旧库程序冲突。 |
| `float` | (SPARC) 允许编译器通过将浮点寄存器用作整数值的临时寄存器来生成代码。使用浮点值可能会用到与该选项无关的这些寄存器。如果希望您的代码没有任何对浮点寄存器的引用，请使用 `-xregs=no%float` 并确保您的代码没有以任何方式使用浮点类型。 |
| `frameptr` | （**仅限** *x86*）允许编译器将帧指针寄存器（在 IA32 上为 `%ebp`，在 AMD64 上为 `%rbp`）用作通用寄存器。<br>缺省值为 `-xregs=no%frameptr`<br>除非也用 `-features=no%except` 禁用了异常，否则 C++ 编译器将忽略 `-xregs=frameptr`。请注意，`-xregs=frameptr` 是 `-fast` 的一部分，但除非同时指定 `-features=no%except`，否则 C++ 编译器将忽略此设置。<br>通过 `-xregs=framptr`，编译器可以自由使用帧指针寄存器来改进程序性能。但是，这可能会限制调试器和性能测量工具的某些功能。堆栈跟踪、调试器和性能分析器不能对通过 `-xregs=frameptr` 编译的函数生成报告。<br>而且，对 Posix `pthread_cancel()` 的 C++ 调用将找不到清理处理程序。<br>如果直接调用或从 C 或 Fortran 函数间接调用的 C++ 函数会引发异常，不应该用 `-xregs=frameptr` 编译 C、Fortran 和 C++ 混合代码。如果使用 `-fast` 编译此类混合源代码，请在命令行中的 `-fast` 选项后添加 `-xregs=no%frameptr`。<br>由于 64 位平台上的可用寄存器更多，因此与 64 位代码相比，使用 `–xregs=frameptr` 编译更容易改进 32 位代码的性能。<br>如果同时指定了 `-xpg`，编译器会忽略 `-xregs=frameptr` 并发出警告。此外，`-xkeepframe` 将覆盖 `-xregs=frameptr`。 |

SPARC 缺省值为 `-xregs=appl,float`。

x86 缺省值为 `-xregs=no%frameptr`。

在 x86 系统中，`-xpg` 与 `-xregs=frameptr` 不兼容，这两个选项不应一起使用。还请注意，`-fast` 中包括 `-xregs=frameptr`。

对于打算供将链接到应用程序的共享库使用的代码，应当使用 `-xregs=no%appl,float` 进行编译。至少共享库应该显式说明它如何使用应用程序寄存器，以便与这些库链接的应用程序知道这些寄存器分配。

例如，在某种全局意义上使用寄存器（例如，使用寄存器指向一些关键数据结构）的应用程序，需要确切地知道其代码未使用 `-xregs=no%appl` 编译的某个库如何使用应用程序寄存器，以便安全地与该库链接。

## A.2.169 -xrestrict[=f]

将赋值为指针的函数参数视为受限指针。*f* 必须是下表中列出的值之一：

**表 A–44** -xrestrict 值

| 值 | 含义 |
|---|---|
| `%all` | 整个文件中的所有指针参数均被视为限定的。 |
| `%none` | 文件中没有指针参数被视为限定的。 |
| `%source` | 只有在主源文件中定义的函数是限定的。在包含文件中定义的函数不是限定的。 |
| *fn*[,*fn*...] | 用逗号隔开的一个或多个函数名称的列表。如果指定一个函数列表，编译器会将指定函数中的指针参数视为限定指针；有关更多信息，请参阅下一节第 271 页中的 “A.2.169.1 受限指针”。 |

此命令行选项可以单独使用，但最好将其用于优化。

例如，以下命令将文件 `prog.c` 中的所有指针参数都视为受限指针。

```
%CC -xO3 -xrestrict=%all prog.cc
```

以下命令将文件 `prog.c` 中的函数 `agc` 中的所有指针参数都视为受限指针：

```
%CC -xO3 -xrestrict=agc prog.cc
```

注意，虽然 C 编程语言的 C99 标准引入了 `restrict` 关键字，但此关键字并不属于当前 C++ 标准。某些编译器具有针对 C99 `restrict` 关键字的 C++ 语言扩展，有时拼写为 `__restrict` 或 `__restrict__`。但 Oracle Solaris Studio C++ 编译器当前没有该扩展。`-xrestrict` 选项是源代码中 `restrict` 关键字的部分替代。（使用该关键字后，并非函数的所有指针参数都需要声明为 `restrict`。）关键字主要影响优化机会，还会限制可以传递给函数的参数。从源代码中删除 `restict` 或 `__restrict` 的所有实例不会影响程序的可观测行为。

缺省值为 `%none`；指定 `-xrestrict` 与指定 `-xrestrict=%source` 等效。

### A.2.169.1 受限指针

为使编译器高效并行执行循环，需要确定某些左值是否指定不同的存储区域。别名是其存储区域相同的左值。由于需要分析整个程序，因此确定对象的两个指针是否为别名是一个困难而费时的过程。请考虑以下示例中的函数 vsq()：

```
extern "C"
void vsq(int n, double *a, double *b) {
    int i;
    for (i=0; i<n; i++) {
            b[i] = a[i] * a[i];
    }
}
```

如果编译器知道指针 a 和 b 访问不同的对象，可以并行化循环的不同迭代的执行。如果通过指针 a 和 b 访问的对象存在重叠，编译器以并行方式执行循环将会不安全。

在编译时，编译器并不能通过简单地分析函数 vsq() 来获悉 a 和 b 访问的对象是否重叠。编译器可能需要分析整个程序才能获取此信息。可以使用以下命令行选项指定将赋值为指针的函数参数视为受限指针：-xrestrict[=*func1,...,funcn*]。如果指定了函数列表，则指定函数中的指针参数被视为受限的。否则，整个源文件中的所有指针参数都会被视为受限的（不推荐）。例如，-xrestrict=vsq 限定 vsq() 函数示例中给定的指针 a 和 b。

将指针参数声明为限定的表示指针指定不同的对象。编译器可以假定 a 和 b 指向不同的存储区域。有了此别名信息，编译器就能够并行化循环。

请确保正确使用 -xrestrict。如果指针被限定为指向并非不同的对象的限定指针，编译器可能会错误地并行化循环，从而导致不确定的行为。例如，假定函数 vsq() 的指针 a 和 b 指向的对象重叠，如 b[i] 和 a[i+1] 是同一对象。如果 a 和 b 未声明为限定指针，循环将以串行方式执行。如果 a 和 b 被错误地限定为受限指针，编译器可能会并行化循环的执行，这是不安全的，因为 b[i+1] 仅应在计算 b[i] 之后进行计算。

## A.2.170 –xs

允许 dbx 在没有目标 (.o) 文件的情况下进行调试。

该选项导致所有调试信息被复制到可执行程序中。此选项对程序的 dbx 性能和运行时性能影响很小，但占用很大的磁盘空间。

该选项只有在 -xdebugformat=stabs 时才有效，缺省情况下不将调试数据复制到可执行文件。使用缺省调试格式 -xdebugformat=dwarf 时，总是将调试数据复制到可执行文件，没有选项可以阻止复制。

## A.2.171 –xsafe=mem

（**仅限** *SPARC*）允许编译器假定不会发生任何内存保护违规。

该选项允许编译器使用 SPARC V9 架构中的无故障装入指令。

### A.2.171.1 交互

仅当与优化级别 `-xO5` 及以下 `-xarch` 值中的一个一起使用时，此选项才能有效：`sparc`、`sparcvis`、`sparcvis2` 或 `sparcvis3`（用于 `-m32` 和 `-m64`）。

### A.2.171.2 警告

由于在发生诸如地址未对齐或段违规的故障时，无故障装入不会导致陷阱，因此您应该只对不会发生此类故障的程序使用该选项。因为只有很少的程序会导致基于内存的陷阱，所以您可以安全地将该选项用于大多数程序。对于显式依赖基于内存的自陷来处理异常情况的程序，请勿使用该选项。

## A.2.172 –xspace

SPARC：不允许进行增加代码大小的优化。

## A.2.173 –xtarget=t

为指令集和优化指定目标平台。

通过为编译器提供目标计算机硬件的精确描述，某些程序的性能可得到提高。当程序性能很重要时，目标硬件的正确指定是非常重要的。在较新的 SPARC 处理器上运行时，尤其是这样。不过，对大多数程序和较旧的 SPARC 处理器来讲，性能的提高微不足道，因此指定 generic 就足够了。

*t* 的值必须是下列值之一：`native`、`generic`、`native64`、`generic64` 或 *system-name*。

`-xtarget` 的每个特定值都会扩展到 `-xarch`、`-xchip` 和 `-xcache` 选项值的特定集合。使用 `-xdryrun` 选项可在运行的系统上确定 `-xtarget=native` 的扩展。

例如，`-xtarget=ultraT2` 等效于 `-xarch=sparcvis2 -xchip=ultraT2-xcache=8/16/4:4096/64/16`。

---

**注 –** `-xtarget` 在特定主机平台上的扩展在该平台上编译时扩展到的 `-xarch`、`-xchip` 或 `-xcache` 设置可能与 `-xtarget=native` 不同。

---

### A.2.173.1 —xtarget 值（按平台）

本节按平台提供 `-xtarget` 值的说明。下表列出了所有平台的 `-xtarget` 值。

表 A–45 -xtarget 值（所有平台）

| 值 | 含义 |
| --- | --- |
| native | 等效于<br>–m32 –xarch=native –xchip=native –xcache=native<br>可在主机 32 位系统上提供最佳性能。 |
| native64 | 等效于<br>–m64 –xarch=native64 –xchip=native64 –xcache=native64<br>可在主机 64 位系统上提供最佳性能。 |
| generic | 等效于<br>–m32 –xarch=generic –xchip=generic –xcache=generic<br>可在大多数 32 位系统上提供最佳性能。 |
| generic64 | 等效于<br>–m64 –xarch=generic64 –xchip=generic64 –xcache=generic64<br>可在大多数 64 位系统上提供最佳性能。 |
| *system-name* | 获取指定平台的最佳性能。<br>从以下代表您所面向的实际系统的列表中选择系统名称： |

## -xtarget 值（SPARC 平台）

在 SPARC 还是 UltraSPARC V9 上针对 64 位 Solaris 软件进行编译，是由 -m64 选项指示。如果指定带有 native64 或 generic64 之外的标志的 -xtarget，还必须指定 -m64 选项，如下所示：-xtarget=ultra... -m64。否则，编译器将使用 32 位内存模型。

表 A–46 SPARC 体系结构上的 -xtarget 扩展

| -xtarget= | -xarch | -xchip | -xcache |
| --- | --- | --- | --- |
| ultra | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra1/140 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra1/170 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra1/200 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra2 | sparcvis | ultra2 | 16/32/1:512/64/1 |
| ultra2/1170 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra2/1200 | sparcvis | ultra | 16/32/1:1024/64/1 |

表 A–46 SPARC 体系结构上的 -xtarget 扩展 （续）

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| ultra2/1300 | sparcvis | ultra2 | 16/32/1:2048/64/1 |
| ultra2/2170 | sparcvis | ultra | 16/32/1:512/64/1 |
| ultra2/2200 | sparcvis | ultra | 16/32/1:1024/64/1 |
| ultra2/2300 | sparcvis | ultra2 | 16/32/1:2048/64/1 |
| ultra2e | sparcvis | ultra2e | 16/32/1:256/64/4 |
| ultra2i | sparcvis | ultra2i | 16/32/1:512/64/1 |
| ultra3 | sparcvis2 | ultra3 | 64/32/4:8192/512/1 |
| ultra3cu | sparcvis2 | ultra3cu | 64/32/4:8192/512/2 |
| ultra3i | sparcvis2 | ultra3i | 64/32/4:1024/64/4 |
| ultra4 | sparcvis2 | ultra4 | 64/32/4:8192/128/2 |
| ultra4plus | sparcvis2 | ultra4plus | 64/32/4:2048/64/4:32768/64/4 |
| ultraT1 | sparcvis2 | ultraT1 | 8/16/4/4:3072/64/12/32 |
| ultraT2 | sparc | ultraT2 | 8/16/4:4096/64/16 |
| ultraT2plus | sparcvis2 | ultraT2plus | 8/16/4:4096/64/16 |
| T3 | sparcvis3 | T3 | 8/16/4:6144/64/24 |
| T4 | sparc4 | T4 | 16/32/4:128/32/8:4096/64/16 |
| sparc64vi | sparcfmaf | sparc64vi | 128/64/2:5120/64/10 |
| sparc64vii | sparcima | sparc64vii | 64/64/2:5120/256/10 |
| sparc64viiplus | sparcima | sparc64viiplus | 64/64/2:11264/256/11 |

## -xtarget 值（x86 平台）

在 64 位 x86 平台上针对 64 位 Solaris 软件进行编译是由 `-m64` 选项指示的。如果为 `-xtarget` 指定 `native64` 或 `generic64` 以外的标志，则还必须按如下所示指定 `-m64` 选项：`-xtarget=opteron ... -m64`。否则，编译器将使用 32 位内存模型。

表 A–47 -xtarget 值（x86 平台）

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| opteron | sse2 | opteron | 64/64/2:1024/64/16 |
| pentium | 386 | pentium | generic |

表 A–47 -xtarget 值（x86 平台） （续）

| -xtarget= | -xarch | -xchip | -xcache |
|---|---|---|---|
| pentium_pro | pentium_pro | pentium_pro | generic |
| pentium3 | sse | pentium3 | 16/32/4:256/32/4 |
| pentium4 | sse2 | pentium4 | 8/64/4:256/128/8 |
| nehalem | sse4_2 | nehalem | 32/64/8:256/64/8:8192/64/16 |
| penryn | sse4_1 | penryn | 2/64/8:4096/64/16 |
| woodcrest | ssse3 | core2 | 32/64/8:4096/64/16 |
| barcelona | amdsse4a | amdfam10 | 64/64/2:512/64/16 |
| sandybridge | avx | sandybridge | 32/64/8:256/64/8:8192/64/16 |
| westmere | aes | westmere | 32/64/8:256/64/8:12288/64/16 |

### A.2.173.2 缺省值

在 SPARC 和 x86 设备上，如果未指定 –xtarget，则假定 –xtarget=generic。

### A.2.173.3 扩展

–xtarget 选项是一个宏，使用它可以快捷地指定可在从市场上购买的平台上使用的 -xarch、–xchip 和 –xcache 组合。–xtarget 的唯一含义在其扩展中。

### A.2.173.4 示例

-xtarget=ultra 表示 -xchip=ultra -xcache=16/32/1:512/64/1 -xarch=sparcvis。

### A.2.173.5 交互

针对 64 位 SPARC V9 体系结构的编译由 -m64 选项指示。不必设置 –xtarget=ultra 或 ultra2，而且这也不够。如果指定了 -xtarget，则对 –xarch、–xchip 或 –xcache 值所做的任何更改都必须位于 -xtarget 之后。例如：

```
–xtarget=ultra3 -xarch=ultra
```

### A.2.173.6 警告

在不同的步骤中进行编译和链接时，必须在编译步骤和链接步骤中使用相同的 -xtarget 设置。

## A.2.174 -xthreadvar[=*o*]

指定 -xthreadvar 来控制线程局部变量的实现。将此选项与 __thread 声明说明符结合使用，可利用编译器的线程局部存储功能。使用 __thread 说明符声明线程变量后，请指定 -xthreadvar，以便能够将线程局部存储用于动态（共享）库中的位置相关的代码（非 PIC 代码）。有关如何使用 __thread 的更多信息，请参见第 59 页中的 “4.2 线程局部存储”。

### A.2.174.1 值

下表列出了 *o* 的可能值。

表 A–48 -xthreadvar 值

| 值 | 含义 |
|---|---|
| [no%]dynamic | 编译动态装入的变量。使用 -xthreadvar=no%dynamic 时对线程变量的访问明显加快，但是不能在动态库中使用目标文件。也就是说，只能在可执行文件中使用目标文件。 |

### A.2.174.2 缺省值

如果未指定 -xthreadvar，则编译器所用的缺省设置取决于是否启用与位置无关的代码。如果启用了与位置无关的代码，则该选项设置为 -xthreadvar=dynamic。如果禁用了与位置无关的代码，则该选项设置为 -xthreadvar=no%dynamic。

如果指定了 -xthreadvar 但未指定任何参数，则该选项设置为 -xthreadvar=dynamic。

### A.2.174.3 交互

编译和链接使用 __thread 的文件时，必须使用 -mt 选项。

### A.2.174.4 警告

如果动态库包含与位置无关的代码，则必须指定 -xthreadvar。

链接程序不支持在动态库中与非 PIC 代码等效的线程变量。由于非 PIC 线程变量要快很多，所以应将其用作可执行文件的缺省设置。

### A.2.174.5 另请参见

-xcode、-KPIC 和 -Kpic

## A.2.175 –xtime

使 cc 驱动程序报告各种编译传递的执行时间。

## A.2.176 -xtrigraphs[={yes|no}]

启用或禁用对 ISO/ANSI C 标准定义的三字母序列的识别。

如果源代码具有包含问号 ( ?) 的文字串（编译器将其解释为三字符序列），那么您可以使用 -xtrigraph=no 子选项禁用对三字符序列的识别。

### A.2.176.1 值

下面列出了 -xtrigraphs 的可能值。

表 A–49 -xtrigraphs 值

| 值 | 含义 |
|---|---|
| yes | 启用整个编译单元三字母序列的识别 |
| no | 禁用整个编译单元三字母序列的识别 |

### A.2.176.2 缺省值

如果没有在命令行上指定 -xtrigraphs 选项，则编译器假定 -xtrigraphs=yes。

如果仅指定了 -xtrigraphs，则编译器假定 -xtrigraphs=yes。

### A.2.176.3 示例

请考虑以下名为 trigraphs_demo.cc 的示例源文件。

```
#include <stdio.h>

int main ()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

以下示例显示了使用 -xtrigraphs=yes 编译此代码时的输出。

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as []
```

以下示例显示了使用 -xtrigraphs=no 编译此代码时的输出。

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

### A.2.176.4 另请参见

有关三字母的信息，请参见《C 用户指南》中关于转换到 ANSI/ISO C 的章节。

# A.2.177 –xunroll= n

此选项指示编译器通过解开循环（如有可能）来优化这些循环。

## A.2.177.1 值

*n* 为 1 时，建议编译器不要解开循环。

*n* 为大于 1 的整数（`-unroll=`*n*）时，编译器会将循环解开 *n* 次。

# A.2.178 -xustr={ascii_utf16_ushort |no}

如果代码中包含要编译器转换成目标文件中 UTF-16 字符串的字符串或字符文字，可以使用该选项。如果不指定该选项，编译器既不生成、也不识别 16 位的字符串文字。使用该选项时，U"*ASCII-string*" 字符串文字会识别为无符号短整型数组。因为这样的字符串还不属于任何标准，所以该选项的作用是使非标准 C++ 得以识别。

不是所有文件都必须使用该选项编译。

## A.2.178.1 值

如果需要支持使用 ISO10646 UTF-16 文本字符串的国际化应用程序，可指定 `-xustr=ascii_utf16_ushort`。可以通过指定 *-xustr=no* 来禁用编译器对 U"`ASCII_string`" 字符串或字符文字的识别。该选项在命令行上最右侧的实例覆盖了先前的所有实例。

可以指定 `-xustr=ascii_ustf16_ushort`，而无需同时指定 U"*ASCII-string*" 字符串文字。这样执行时不会出现错误。

## A.2.178.2 缺省值

缺省值为 `-xustr=no`。如果指定了没有参数的 `-xustr`，编译器将不接受该选项，而是发出一个警告。如果 C 或 C++ 标准定义了语法的含义，那么缺省设置是可以更改的。

## A.2.178.3 示例

以下示例显示了前置有 U 的带引号文本字符串。它还显示指定 `-xustr` 的命令行。

```
example% cat file.cc
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() {return foo;}
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

8 位字符文字可以带有 `U` 前缀，以形成一个 `unsigned short` 类型的 16 位 UTF-16 字符。例如：

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

## A.2.179 -xvector[= a]

启用向量库函数调用自动生成，或在支持 SIMD（Single Instruction Multiple Data，单指令多数据）的 x86 处理器上启用 SIMD 指令生成。使用此选项时，必须通过指定 `-fround=nearest` 来使用缺省的舍入模式。

`-xvector` 选项需要 `-xO3` 或更高的优化级别。如果优化级别未指定或低于 `-xO3`，编译将不会继续，同时会发出消息。

下表列出了 *a* 的可能值。`no%` 前缀可禁用关联的子选项。

**表 A–50** -xvector 子选项

| 值 | 含义 |
|---|---|
| [no%]lib | （**仅限** *Solaris*）允许编译器将循环内的数学库调用转换为对等效向量数学例程的单个调用（如果能够进行此类转换）。此类转换可提高那些循环计数较大的循环的性能。使用 `no%lib` 可以禁用此选项。 |
| [no%]simd | （**仅限** *x86*）指示编译器使用本机 x86 SSE SIMD 指令来提高某些循环的性能。在 x86 中，缺省情况下以可产生有利结果的优化级别 3 和更高级别使用流扩展。可以使用 `no%simd` 禁用此选项。<br>仅当目标体系结构中存在流扩展（即目标 ISA 至少为 SSE2）时，编译器才会使用 SIMD。例如，可在现代平台中指定 `-xtarget=woodcrest`、`-xarch=generic64`、`-xarch=sse2`、`-xarch=sse3` 或 `-fast` 来使用它。如果目标 ISA 没有流扩展，子选项将无效。 |
| `%none` | 完全禁用此选项。 |
| `yes` | 此选项已过时；请改为指定 `-xvector=lib`。 |
| `no` | 此选项已过时；请改为指定 `-xvector=%none`。 |

### A.2.179.1 缺省值

在 x86 平台上的缺省值为 `-xvector=simd`，在 SPARC 平台上的缺省值为 `-xvector=%none`。如果指定不带子选项的 `-xvector`，在 x86 Solaris、SPARC Solaris 和 Linux 平台上，编译器将分别采用 `-xvector=simd,lib`、`-xvector=lib` 和 `-xvector=simd`。

### A.2.179.2 交互

在装入步骤中，编译器包含 `libmvec` 库。

如果使用单独的命令进行编译和链接，应确保也在链接 `CC` 命令中使用 `-xvector`。

## A.2.180 -xvis[={ yes|no}]

（**仅限** *SPARC*）在使用 VIS 软件开发者工具包 (VIS Software Developers Kit, VSDK) 中定义的汇编语言模板时，或使用采用了 VIS 指令和 `vis.h` 头文件的汇编程序内联代码时，请使用 `-xvis=[ yes|no]` 命令。

VIS 指令集是 SPARC v9 指令集的扩展。尽管 UltraSPARC 是 64 位处理器，但在很多情况下数据都限制在 8 位或 16 位范围内，特别是多媒体应用程序中。VIS 指令可以用一条指令处理四个 16 位数据，这个特性使得处理诸如图像、线性代数、信号处理、音频、视频以及网络等新媒体的应用程序的性能大大提高。

### A.2.180.1 缺省值

缺省值为 `-xvis=no`。指定 `-xvis` 与指定 `-xvis=yes` 等效。

## A.2.181 -xvpara

发出关于并行编程相关的潜在问题的警告，在使用 OpenMPI 时这些问题可能会导致不正确的结果。与 `-xopenmp` 和 OpenMP API 指令一起使用。

编译器在检测到下列情形时会发出警告。

- 循环是使用 MP 指令并行化的，而这些指令中的不同循环迭代之间存在数据依赖性
- OpenMP 数据共享属性子句存在问题。例如，声明在 OpenMP 并行区域中的访问可能导致数据争用的变量 "shared"，或者声明其在并行区域中的值在并行区域之后使用的变量 "private"。

如果所有并行化指令在处理期间均未出现问题，则不显示警告。

---

**注 –** Solaris Studio 编译器支持 OpenMP API 并行化。因此，已废弃 MP pragma 指令，不再支持此类指令。有关迁移到 OpenMP API 的信息，请参见《*OpenMP API* **用户指南**》。

---

## A.2.182 –xwe

通过返回非零的退出状态，将所有警告转换成错误。

### A.2.182.1 另请参见

第 164 页中的 “A.2.15 `-errwarn[= t]`”

## A.2.183 -Yc,path

指定组件 *c* 的位置的新路径。

如果已指定组件的位置，则组件的新路径名称为 *path/component-name*。该选项传递给 `ld`。

### A.2.183.1 值

下表列出了 *c* 的可能值。

**表 A–51** -Y 标志

| 值 | 含义 |
|---|---|
| P | 更改 `cpp` 的缺省目录。 |
| 0 | 更改 `ccfe` 的缺省目录。 |
| a | 更改 `fbe` 的缺省目录。 |
| 2 | 更改 `iropt` 的缺省目录。 |
| c (SPARC) | 更改 `cg` 的缺省目录。 |
| O | 更改 `ipo` 的缺省目录。 |
| k | 更改 `CClink` 的缺省目录。 |
| l | 更改 `ld` 的缺省目录。 |
| f | 更改 `c++filt` 的缺省目录。 |
| m | 更改 `mcs` 的缺省目录。 |
| u *(x86)* | 更改 `ube` 的缺省目录。 |
| h *(x86)* | 更改 `ir2hf` 的缺省目录。 |
| A | 指定目录以搜索所有编译器组件。如果在 *path* 中找不到组件，搜索将转至编译器的安装目录。 |
| P | 将路径添加到缺省库搜索路径。将在缺省库搜索路径之前搜索此路径。 |
| S | 更改启动目标文件的缺省目录 |

### A.2.183.2 交互

可以在命令行上指定多个 `-Y` 选项。如果对任何一个组件应用了多个 `-Y` 选项，则保留最后一个选项。

### A.2.183.3 另请参见

*Solaris*《**链接程序和库指南**》

## A.2.184 -z[ ]arg

链接编辑器选项。有关更多信息，请参见 ld(1) 手册页和 Oracle Solaris《链接程序和库指南》。

另请参见第 208 页中的 “A.2.98 -Xlinker *arg*”

◆ ◆ ◆ 附录 B

# Pragma

本附录介绍了 C++ 编译器 pragma。*pragma* 是一个编译器指令，程序员可以通过它向编译器提供额外的信息。该信息可以更改您所控制的编译详细信息。例如，`pack` pragma 会影响结构内的数据布局。编译器 pragma 也称为**指令**。

预处理程序关键字 `pragma` 是 C++ 标准的一部分，但每个编译器中，pragma 的形式、内容和含义都是不相同。C++ 标准不定义任何 pragma。

**注 –** 依赖于 pragma 的代码是不可移植的。

## B.1 Pragma 形式

C++ 编译器 pragma 的各种形式如下所示：

```
#pragma keyword
#pragma keyword ( a [ , a ] ...) [ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

变量 *keyword* 指特定指令；*a* 表示参数。

### B.1.1 将函数作为 pragma 参数进行重载

本附录中列出了几个将函数名称作为参数的 pragma。如果重载该函数，则 pragma 使用其前面的函数声明作为其参数。请看以下示例：

```
int bar(int);
int foo(int);
int foo(double);
#pragma does_not_read_global_data(foo, bar)
```

在此示例中，`foo` 指 `foo(double)`，即 pragma 紧前面的 `foo` 声明；而 `bar` 指 `bar(int)`，即唯一声明的 `bar`。现在，请看以下示例，在此示例中再次重载了 `foo`：

```
int foo(int);
int foo(double);
int bar(int);
#pragma does_not_read_global_data(foo, bar)
```

在此示例中，bar 指 bar(int)，即唯一声明的 bar。但 pragma 并不知道要使用哪个版本的 foo。要更正此问题，必须将 pragma 放在希望 pragma 使用的 foo 定义的紧后面。

以下 pragma 使用本节中介绍的选择方法：

- does_not_read_global_data
- does_not_return
- does_not_write_global_data
- no_side_effect
- opt
- rarely_called
- returns_new_memory

# B.2 Pragma 参考

本节介绍了 C++ 编译器可识别的 pragma 关键字。

## B.2.1 #pragma align

#pragma align *integer*(*variable*[,*variable*...])

使用 align 使所列变量与 *integer* 字节内存对齐，并覆盖缺省设置。请遵循以下限制：

- *integer* 必须是 2 的幂，介于 1 和 128 之间。有效值为 1、2、4、8、16、32、64 和 128。
- *variable* 是一个全局或静态变量。它不能是局部变量或类成员变量。
- 如果指定的对齐比缺省值小，就使用缺省值。
- Pragma 行必须出现在所涉及的变量的声明之前。否则，它将被忽略。
- 在 pragma 行上涉及但不在下面 pragma 行的代码中声明的任何变量都被忽略。以下示例中的变量是正确声明的。

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

#pragma align 在名称空间内部使用时，必须使用改编名称。例如，以下代码中的 #pragma align 语句就是无效的。要更正此问题，应将 #pragma align 语句中的 a、b 和 c 替换为其改编名称。

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

## B.2.2 #pragma does_not_read_global_data

```
#pragma does_not_read_global_data(funcname[, funcname])
```

此 pragma 断言，指定的例程不直接或间接读取全局数据，从而在调用此类例程的周围实现更好的代码优化。具体来讲，赋值语句或存储可以围绕这样的调用移动。

指定函数的原型被声明后，该 pragma 才可用。如果全局访问的断言不为真，那么程序的行为就是未定义的。

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.3 #pragma does_not_return

```
#pragma does_not_return(funcname[, funcname])
```

此 pragma 向编译器断言，将不会返回对指定例程的调用，从而使编译器可执行与该假定一致的优化。例如，寄存器生命周期在调用点终止，这样可以进行进一步的优化。

如果指定的函数不返回，程序的行为就是未定义的。

指定函数的原型被声明后，该 pragma 才是可用的，如以下示例所示：

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.4 #pragma does_not_write_global_data

```
#pragma does_not_write_global_data(funcname[, funcname])
```

此 pragma 断言，指定的例程列表不直接或间接写入全局数据，从而在调用此类例程的周围实现更好的代码优化。具体来讲，赋值语句或存储可以围绕这样的调用移动。

指定函数的原型被声明后，该 pragma 才可用。如果全局访问的断言不为真，那么程序的行为就是未定义的。

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.5 #pragma dumpmacros

```
#pragma dumpmacros (value[,value...])
```

要查看宏在程序中如何工作时，请使用该 pragma。该 pragma 提供了诸如宏定义、取消定义和用法实例的信息，并按宏的处理顺序将输出输出到标准错误 (stderr)。dumpmacros pragma 达到文件结尾或遇到 #pragma end_dumpmacro 之前一直有效。请参见第 287 页中的 “B.2.6 #pragma end_dumpmacros”。下表列出了 *value* 的可能值：

| 值 | 含义 |
|---|---|
| defs | 打印所有宏定义 |
| undefs | 打印所有取消定义的宏 |
| use | 打印关于使用的宏的信息 |
| loc | 另外打印 defs、undefs 和 use 的位置（路径名和行号） |
| conds | 打印在条件指令中使用的宏的信息 |
| sys | 打印系统头文件中所有宏的定义、取消定义和使用的信息 |

**注 –** 子选项 loc、conds 和 sys 是 defs、undefs 和 use 选项的限定符。使用 loc、conds 和 sys 本身并不会生成任何结果。例如，#pragma dumpmacros(loc,conds,sys) 不起任何作用。

dumpmacros pragma 与命令行选项作用相同，但 pragma 会覆盖命令行选项。请参见第 225 页中的 “A.2.116 -xdumpmacros[=*value*[,*value*...]]”。

dumpmacros pragma 并不嵌套，因此以下代码行中，处理 #pragma end_dumpmacros 时，将停止打印宏信息：

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(defs, undefs)
...
#pragma end_dumpmacros
```

dumpmacros pragma 的作用是累积的。以下代码行

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

具有和以下行相同的效果：

```
#pragma dumpmacros(defs, undefs, loc)
```

如果使用选项 `#pragma dumpmacros(use,no%loc)`，则使用的每个宏的名称仅输出一次。如果使用选项 `#pragma dumpmacros(use,loc)`，则每次使用宏时都输出位置和宏名称。

## B.2.6 #pragma end_dumpmacros

```
#pragma end_dumpmacros
```

此 pragma 标记 `dumpmacros pragma` 的结尾，并停止输出有关宏的信息。如果没有在 `dumpmacros` pragma 后面使用 `end_dumpmacros` pragma，`dumpmacros` pragma 会在文件结尾一直生成输出。

## B.2.7 #pragma error_messages

`#pragma error_messages (on|off| default,` *tag…* *tag*`)`

错误消息 pragma 提供源程序内部对编译器发出的消息的控制。Pragma 只对警告消息有效。 `-w` 命令行选项可通过抑制所有警告消息来覆盖此 pragma。

- `#pragma error_messages (on,` *tag…* *tag*`)`

  `on` 选项结束前面的任何 `#pragma error_messages` 选项（如 `off` 选项）的作用域，并覆盖 `-erroff` 选项的效果。

- `#pragma error_messages (off,` *tag…* *tag*`)`

  `off` 选项阻止编译器程序发出以 pragma 中指定的标记开头的指定消息。pragma 对任何指定的错误消息的作用域仍然有效，直到被另一个 `#pragma error_messages` 覆盖或编译结束。

- `#pragma error_messages (default,` *tag…* *tag*`)`

  `default` 选项结束前面的任何 `#pragma error_messages` 指令对指定标记的作用域。

## B.2.8 #pragma fini

`#pragma fini (`*identifier*`[,`*identifier*`...])`

可使用 `fini` 将 *identifier* 标记为完成函数。此类函数应为 `void` 类型，不接受任何参数，当程序在程序控制下终止或从内存删除包含的共享对象时调用它们。与初始化函数一样，完成函数按链接编辑器处理的顺序执行。

在源文件中，`#pragma fini` 中指定的函数在该文件中的静态析构函数后面执行。在 pragma 中使用标识符之前，请先声明这些标识符。

此类函数每出现在 `#pragma fini` 指令中一次，就会被调用一次。

## B.2.9 #pragma hdrstop

可将 hdrstop pragma 嵌入源文件的头文件中以标识源文件活前缀的结尾。例如，考虑以下文件：

```
example% cat a.cc
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.cc
#include "a.h"
#include "b.h"
#include "c.h"
```

源文件活前缀以 c.h 结束，因此可在每个文件中的 c.h 后面插入 #pragma hdrstop。

#pragma hdrstop 只能位于用 CC 命令指定的源文件活前缀的结尾处。不要在任何 include 文件中指定 #pragma hdrstop。

请参见第 255 页中的 “A.2.156 -xpch=*v*”和第 257 页中的 “A.2.157 -xpchstop=*file* ”。

## B.2.10 #pragma ident

```
#pragma ident string
```

可使用 ident 将 *string* 放在可执行文件的 .comment 部分中。

## B.2.11 #pragma init

```
#pragma init(identifier[,identifier...])
```

可使用 init 将 *identifier* 标记为初始化函数。此类函数应为 void 类型，不接受任何参数，在开始执行时构造程序的内存映像的情况下调用。执行将共享对象送入内存的操作时（程序启动时，或执行某些动态装入操作时，例如 dlopen()），执行共享对象中的初始化函数。调用到初始化函数的唯一顺序就是链接编辑器静态和动态处理该函数的顺序。

在源文件中，#pragma init 中指定的函数在该文件中的静态析构函数后面执行。在 pragma 中使用标识符之前，请先声明这些标识符。

此类函数每出现在 #pragma init 指令中一次，就会被调用一次。

## B.2.12 #pragma ivdep

`ivdep` pragma 指示编译器忽略在循环中找到的部分或全部对数组引用的循环附带依赖性，以进行优化。这将使编译器能够执行各种循环优化（例如微向量化、分发、软件流水线等），而以其他方式可能无法执行这些优化。当用户知道这些依赖性无关紧要或者实际上永远不会发生时，可以使用该指令。

`#pragma ivdep` 指令的解释依赖于 `–xivdep` 选项的值。

## B.2.13 #pragma must_have_frame

`#pragma must_have_frame(`*funcname*`[,`*funcname*`])`

该 pragma 要求总是编译指定的一组函数来获得完整的堆栈帧（如 System V ABI 中所定义）。必须在使用该 pragma 列出函数之前，声明该函数的原型。

```
extern void foo(int);
extern void bar(int);
#pragma must_have_frame(foo, bar)
```

只有在声明了指定函数的原型后，才允许使用该 pragma。该 pragma 必须位于函数结尾之前。

```
void foo(int) {
  .
  #pragma must_have_frame(foo)
  .
  return;
  }
```

请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.14 #pragma no_side_effect

`#pragma no_side_effect(`*name*`[,`*name*`...])`

可使用 `no_side_effect` 指示函数不更改任何持久状态。Pragma 声明了命名的函数不具有任何副作用。也就是说，函数将返回仅依赖于传递参数的结果。此外，函数及其调用的子孙函数行为如下所示：

- 不读取或写入调用时调用程序中可视程序状态的任何部分。
- 不执行 I/O。
- 不更改调用时不可视程序状态的任何部分。

编译器执行优化时可以使用该信息。

如果函数具有副作用，执行调用该函数的程序的结果是未定义的。

*name* 参数指定当前转换单元中函数的名称。Pragma 必须与函数在相同的作用域，并且必须在函数声明之后出现。pragma 必须在函数定义之前。

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.15 #pragma opt

`#pragma opt` *level* (*funcname[, funcname]*)

*funcname* 指定当前转换单元中定义的函数的名称。*level* 值指定用于所指定函数的优化级别。可以指定优化级别 0、1、2、3、4、5。可以通过将 *level* 设置为 0 来关闭优化。必须在该 pragma 之前使用原型或空参数列表声明函数。pragma 则必须对要优化的函数进行定义。

pragma 中所列任何函数的优化级别都降为 `-xmaxopt` 值。`-xmaxopt=off` 时，忽略 pragma。

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.16 #pragma pack( n)

`#pragma pack([`*n*`])`

可使用 pack 影响对结构成员的封装。

如果使用了该项，*n* 必须为 0 或 2 的幂。0 以外的值指示编译器针对数据类型使用 *n* 字节对齐和平台的自然对齐中的较小者。例如，以下指令使在指令后面（以及后续的 pack 指令前面）定义的所有结构成员对齐时依据的字节边界不严于 2 字节边界，即使正常对齐是 4 或 8 字节边界时也是如此。

```
#pragma pack(2)
```

*n* 为 0 或省略该项时，成员对齐还原为自然对齐值。

如果 *n* 值等于或大于平台上最严格的对齐时，则采用自然对齐。下表显示了每个平台最严格的对齐。

**表 B–1** 平台上最严格的对齐

| 平台 | 最严格的对齐 |
|---|---|
| x86 | 4 |
| SPARC 通用 | 8 |

表 B–1 平台上最严格的对齐 （续）

| 平台 | 最严格的对齐 |
|---|---|
| 64 位 SPARC V9 (`-m64`) | 16 |

pack 指令应用于自身与下一个 pack 指令之间的所有结构定义。如果在具有不同包装的不同转换单元中定义了相同的结构，那么程序可能会因某种原因而失败。具体来说，不应该在包括定义预编译库接口的头文件之前使用 pack 指令。建议将 pack 指令放在要封装的结构紧前面的程序代码中，并将 `#pragma pack()` 放在该结构紧后面。

如果在 SPARC 平台上使用 `#pragma pack` 封装效果比类型的缺省对齐紧密，必须为应用程序的编译和链接指定 `-misalign` 选项。下表显示了整型数据类型的存储大小和缺省对齐。

表 B–2 存储大小和缺省对齐字节数

| 类型 | 32 位 SPARC<br>大小，对齐 | 64 位 SPARC<br>大小，对齐 | x86<br>大小，对齐 |
|---|---|---|---|
| `bool` | 1, 1 | 1, 1 | 1, 1 |
| `char` | 1, 1 | 1, 1 | 1, 1 |
| `short` | 2, 2 | 2, 2 | 2, 2 |
| `wchar_t` | 4, 4 | 4, 4 | 4, 4 |
| `int` | 4, 4 | 4, 4 | 4, 4 |
| `long` | 4, 4 | 8, 8 | 4, 4 |
| `float` | 4, 4 | 4, 4 | 4, 4 |
| `double` | 8, 8 | 8, 8 | 8, 4 |
| `long double` | 16, 8 | 16, 16 | 12, 4 |
| 指向数据的指针 | 4, 4 | 8, 8 | 4, 4 |
| 指向函数的指针 | 4, 4 | 8, 8 | 4, 4 |
| 指向成员数据的指针 | 4, 4 | 8, 8 | 4, 4 |
| 指向成员函数的指针 | 8, 4 | 16, 8 | 8, 4 |

## B.2.17 #pragma rarely_called

`#pragms rarely_called(`*funcname[, funcname]*`)`

此 pragma 向编译器发出提示以说明很少调用指定的函数，从而使编译器可以对此类例程的调用点执行分析反馈式优化，而不需要分析收集阶段的开销。因为该 pragma 只是建议，所以编译器不执行基于该 pragma 的任何优化。

只有在声明指定函数的原型之后，才能使用 `#pragma rarely_called` 预处理程序指令。以下是 `#pragma rarely_called` 示例：

```
extern void error (char *message);
#pragma rarely_called(error)
```

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.18 #pragma returns_new_memory

`#pragma returns_new_memory(`*name*`[,`*name*`...])`

此 pragma 断言，每个指定的函数都返回新分配内存的地址，以及指针没有与任何其他指针相关的别名。使用该信息，优化器可以更好地跟踪指针值，厘清内存分配，从而改善调度和流水作业。

如果该断言为假，那么执行调用该函数的程序的结果是未定义的。

*name* 参数指定当前转换单元中函数的名称。Pragma 必须与函数在相同的作用域，并且必须在函数声明之后出现。pragma 必须在函数定义之前。

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 283 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

## B.2.19 #pragma unknown_control_flow

`#pragma unknown_control_flow(`*name*`[,`*name*`...])`

可使用 `unknown_control_flow` 指定一组违反过程调用的常规控制流属性的例程。例如，可通过任意的任何其他例程调用来访问 `setjmp()` 调用后面的语句。该语句通过调用 `longjmp()` 来访问。

因为这种例程使标准流程图分析无效，调用它们的例程不能安全地优化，所以要禁用优化器来编译这些例程。

如果函数名称被重载，那么会选择最近声明的函数。

## B.2.20 #pragma weak

`#pragma weak` *name1* `[=` *name2*`]`

可使用 weak 定义弱全局符号。该 pragma 主要在源文件中用于生成库。链接程序在不能解决弱符号时不会发出警告。

weak pragma 可以按以下两种形式之一来指定符号：

- **字符串形式。**字符串必须是 C++ 变量或函数的改编名称。无效改编名称引用的行为是不可预测的。编译器可能不会针对无效的改编名称引用生成错误。无论是否生成错误，使用了无效改编名称时编译器的行为都是不可预测的。
- **标识符形式。**标识符必须是之前在编译单元中声明的 C++ 函数的明确标识符。标识符形式不能用于变量。前端 (ccfe) 遇到无效的标识符引用时会生成错误消息。

### B.2.20.1 #pragma weak name

采用 #pragma weak *name* 形式时，指令使 *name* 成为弱符号。链接程序将不会指示是否未找到 *name* 的符号定义。它也不会在出现符号的多个弱定义时发出警告。链接程序仅执行第一个遇到的定义。

如果另一个编译单元有函数或变量的强定义，那么 *name* 将链接到它。如果没有 *name* 的强定义，那么链接程序符号的值为 0。

以下指令将 ping 定义为弱符号。链接程序找不到名为 ping 的符号的定义时，不会生成错误消息。

```
#pragma weak ping
```

#### #pragma weak name1 = name2

采用 #pragma weak *name1* = *name2* 形式时，符号 *name1* 成为对 *name2* 的弱引用。如果没有在其他地方定义 *name1*，那么 *name1* 的值为 *name2*。如果在其他地方定义了 *name1*，那么链接程序使用该定义并忽略对 *name2* 的弱引用。以下指令指示链接程序解析对 bar（如果已在程序中某处定义）的任何引用，以及解析对 foo 的引用。

```
#pragma weak bar = foo
```

采用标识符形式时，必须在当前编译单元中声明和定义 *name2*。例如：

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

使用字符串形式时，符号不需要预先声明。如果以下示例中的 _bar 和 bar 都是 extern "C"，则不需要声明函数。但 bar 必须在同一对象中定义。

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

## 重载函数

使用标识符形式时，在 pragma 位置的作用域中必须只有一个具有指定名称的函数。尝试将标识符形式 `#pragma weak` 用于重载函数会出现错误。例如：

```
int bar(int);
float bar(float);
#pragma weak bar        // error, ambiguous function name
```

要避免错误，请使用字符串形式，如以下示例所示。

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

有关更多信息，请参见 Oracle Solaris《链接程序和库指南》。

# 词汇表

**ABI** 请参见 Application Binary Interface（应用程序二进制接口）。

**abstract class（抽象类）** 包含一个或多个抽象方法并因此不能被实例化的类。定义抽象类的目的是为了通过实现抽象方法，使其他类可以扩展抽象类并使其固定。

**abstract method（抽象方法）** 不包含实现的方法。

**ANSI C** 美国国家标准学会定义的 C 编程语言。ANSI C 与 ISO 定义相同。请参见 ISO。

**ANSI/ISO C++** 美国国家标准学会和 ISO C++ 编程语言标准。请参见 ISO。

**application binary interface（应用程序二进制接口）** 编译的应用程序和运行应用程序的操作系统之间的二进制系统接口。

**array（数组）** 内存中连续存储一组单一数据类型值的数据结构。每个值可以按在数组中的位置访问。

**binary compatibility（二进制兼容性）** 链接目标文件的能力，目标文件由某一个发行版本编译，而使用另一个不同发行版本的编译器。

**binding（绑定）** 将函数调用与特定函数定义关联。更一般的说来，将名称与特定的实体关联。

**cfront** C++ 到 C 的编译器程序，可以将 C++ 转换为 C 源代码，然后用标准 C 编译器编译。

**class template（类模板）** 描述一组类或相关数据类型的模板。

**class variable（类变量）** 作为一个整体与特定类关联但与类的特定实例不关联的数据项。类变量在类定义中定义。类变量也称为静态字段。另请参见 instance variable（实例变量）。

**class（类）** 由命名的数据元素（可以是不同类型的数据元素）和可以和数据一起执行的一组操作组成的用户定义数据类型。

**compiler option（编译器选项）** 更改编译器行为的指令。例如，`-g` 选项表示通知编译器为调试器生成数据。同义字：**标志**、**开关**。

**constructor（构造函数）** 每当创建类对象时编译器都会自动调用的特殊类成员函数，用于确保初始化相应对象的实例变量。构造函数必须始终具有与该函数所属的类相同的名称。请参见 destructor（析构函数）。

**data member（数据成员）** 是**数据**而不是函数或类型定义的类元素。

**data type（数据类型）** 用来表示字符、整数或浮点数等的机制。类型决定了分配到变量的存储以及能在变量上执行的操作。

**destructor（析构函数）** 每当销毁类对象或对类指针应用运算符 `delete` 时编译器都会自动调用的特殊类成员函数。析构函数必须始终具有与该函数所属的类相同的名称，该类前有一个 (~)。请参见 *constructor*（**构造函数**）。

**dynamic binding（动态绑定）** 在运行时函数调用到函数体的连接。有虚函数时才需动态绑定。也称为**迟绑定**、**运行时绑定**。

**dynamic cast（动态强制类型转换）** 将指针或引用从声明的类型转换到与引用到的动态类型一致的任何类型的安全方法。

**dynamic type（动态类型）** 由具有不同声明类型的指针或引用访问的对象的实际类型。

**early binding（早绑定）** 请参见 static binding（静态绑定）。

**ELF file（ELF 文件）** 编译器生成的可执行和链接格式文件。

**exception handler（异常处理程序）** 为处理错误而专门编写的代码，当发生了已为其注册了处理程序的异常时，会自动调用该代码。

**exception handling（异常处理）** 设计用于拦截并防止错误的错误恢复过程。在程序执行期间，如果检测到同步错误，那么程序的控制返回到在执行初期注册的异常处理程序，并且忽略包含错误的代码。

**exception（异常）** 在正常的程序流中出现的错误，阻止程序继续运行。某些错误的原因包括了内存枯竭或被零除。

**flag（标志）** 请参见 compiler option（编译器选项）。

**function overloading（函数重载）** 将相同的名称但不同的参数类型和数字赋予不同的函数。也称为**函数多态**。

**function prototype（函数原型）** 描述函数与程序其他部分之间的接口的声明。

**function template（函数模板）** 一种机制，允许您编写单一函数，之后在编写相关函数时可以将该函数用作模型或模式。

**functional polymorphism（函数多态）** 请参见 function overloading（函数重载）。

**idempotent（幂等）** 头文件属性，在一个转换单元中包括多次与包括一次具有相同效果。

**incremental linker（增量链接程序）** 通过仅将更改后的 `.o` 文件链接到前一个可执行文件来创建新的可执行文件的链接程序。

**base class（基类）** 请参见 inheritance（继承）。

**derived class（派生类）** 请参见 inheritance（继承）。

**inheritance（继承）** 面向对象编程的一个功能，使得程序员可以从现有类（基类）派生新的类（派生类）。有以下三种继承：公共的、受保护的和专用的。

**inline function（内联函数）** 用实际函数代码替换函数调用的函数。

**instance variable（实例变量）** 与特定对象关联的任何数据项。类的每个实例具有在类中定义的实例变量的自身副本。实例变量也称为字段。另请参见 class variable（类变量）。

**instantiation（实例化）** C++ 编译器从模板创建可用的函数或对象的过程。

**ISO** 国际标准化组织。

**K&R C** Brian Kernighan 和 Dennis Ritchie 在 ANSI C 之前开发的实际上的 C 编程语言标准。

**keyword（关键字）** 在编程语言中具有唯一含义，并且仅在该语言的专用上下文中使用的字。

**late binding（迟绑定）** 请参见 dynamic binding（动态绑定）。

**linker（链接程序）** 连接目标代码和库以形成完整的可执行程序的工具。

**local variable（局部变量）** 在块内已知的数据项，但块外代码不可访问。例如，在方法内定义的任何变量都是局部变量，在方法外部无法使用。

**locale（语言环境）** 某个地理区域或语言特有的一组约定，例如日期、时间和货币格式。

**lvalue（左值）** 指定变量数据值在内存中的存储位置的表达式。即显示在赋值运算符左侧的变量实例。

**mangle（改编）** 请参见 name mangling（名称改编）。

**member function（成员函数）** 身为函数（而非数据定义或类型定义）的类的元素。

**method（方法）** 在某些面向对象的语言中，成员函数的另外一个名称。

**multiple inheritance（多继承）** 直接源于多个基类的派生类的继承。

**multithreading（多线程）** 在单处理器或多处理器系统上开发并行应用程序的软件技术。

**name mangling（名称改编）** 在 C++ 中，大量函数可以共享相同的名称，因此仅用名称并不能很好的区分不同的函数。编译器通过名称改编解决这个问题：为函数创建由函数名称及其参数的某些组合组成的函数的唯一名称。该策略启用了类型安全链接。也称为**名称修饰**。

**namespace（名称空间）** 控制全局名称的作用域的机制，做法是允许全局空间划分为独立的唯一命名的作用域。

**operator overloading（运算符重载）** 使用同一运算符表示法产生不同结果的能力。函数重载的特殊形式。

**optimization（优化）** 改善编译器生成的目标代码执行效率的过程。

**option（选项）** 请参见 compiler option（编译器选项）。

**overloading（重载）** 将相同的名称赋予多个函数或运算符。

**polymorphism（多态性）** 引用到对象的指针或引用的动态类型与声明的指针或引用类型不同的能力。

**pragma** 指示编译器执行特定操作的编译器预处理程序指令或特殊的注释。

**runtime binding（运行时绑定）** 请参见 dynamic binding（动态绑定）。

**runtime type identification（运行时类型标识）** 提供标准方法以让程序在运行时决定对象类型的机制。

**rvalue（右值）** 位于赋值运算符右侧的变量。右值可以读取而不能被更改。

**scope（作用域）** 操作或定义应用的范围。

**stab** 在目标代码中生成的符号表条目。在 `a.out` 文件和 ELF 文件中使用相同的格式来包含调试信息。

**stack（堆栈）** 一种数据存储方法，通过该方法，可以将数据添加到堆栈顶部或从堆栈顶部删除数据，采用的是后进先出策略。

**static binding（静态绑定）** 在编译期间函数调用到函数体的连接。也称为 *early binding*（**早绑定**）。

**subroutine（子例程）** 函数。在 Fortran 中，子例程是不返回值的函数。

**switch（开关）** 请参见 compiler option（编译器选项）。

**symbol table（符号表）** 程序编译时显示的所有标识符、程序中标识符的位置和属性的列表。编译器使用该表来解释标识符的使用。

**symbol（符号）** 表示某些程序实体的名称或标签。

**template database（模板数据库）** 包含需要处理并实例化模板（程序需要）的所有配置文件的目录。

**template options file（模板选项文件）** 由用户提供的文件，其中包含模板编译选项、源码位置和其他信息。模板选项文件现在已过时，不应该使用。

**template specialization（模板专门化）** 类模板成员函数的专用实例，用于缺省不能处理给出的足够类型时覆盖缺省实例。

**trapping（陷阱操作）** 为了执行其他操作，而对诸如程序执行等操作的拦截。拦截引起微处理器操作的临时中止，并将程序控制转交给另一个源。

**type（类型）** 使用符号方法的描述。基本类型包括 `integer` 和 `float`。所有其他类型都是从这些基本类型构造的，构造方法有：将基本类型收集到数组或结构中，或增加诸如指针或常量属性等的修饰符。

**variable（变量）** 标识符命名的数据项。每个变量都有类型（如 `int` 或 `void`）和作用域。另请参见 class variable（类变量）、instance variable（实例变量）、local variable（局部变量）。

**VTABLE（虚拟表）** 编译器为包含虚函数的每个类创建的表。

# 索引


## 数字和符号

–###, 编译器选项，156
\>\> 提取运算符, iostream，131
–#, 编译器选项，156
!NOT 运算符, iostream，130, 134
$ 标识符, 允许作为非词首字符，168

## A

.a, 文件名后缀，31, 147
Apache C++ 标准库，189
applicator, 参数化操纵符，140
ATS：自动调优系统，258
__attribute__，63

## B

-B**绑定**, 编译器选项，95, 156
bool 类型和文字, 允许，168

## C

.cc, 文件名后缀，31
.c++, 文件名后缀，31
C++ 标准库
　RogueWave 版本，121
　手册页，111
　替换，116–119
　组件，121
C++ 手册页, 访问，111
.c, 文件名后缀，31
.C, 文件名后缀，31
C 标准库头文件, 替换，119
C 接口
　创建库，150
　删除 C++ 运行时依赖关系，150
-c, 编译器选项，33, 158
C99 支持，238
CCadmin 命令，85
CCFLAGS, 环境变量，39
cerr 标准流，127
char, 带符号，218
char* 提取器，132–133
重新排列函数，228
cin 标准流，127
clog 标准流，127
-compat, 编译器选项，158
cout, 标准流，127
__cplusplus, 预定义宏，67
.cpp, 文件名后缀，31
.cxx, 文件名后缀，31

## D

+d, 编译器选项，159
.d 文件扩展名，244
-D, 编译器选项，41, 160
-d, 编译器选项，161
-DDEBUG，90
dec, iostream 操纵符，138
dlclose(), 函数调用，148

dlopen(), 函数调用，147
-dryrun, 编译器选项，34, 162
dwarf 调试器数据格式，224

**E**
-E 编译器选项，162
elfdump，223
endl, iostream 操纵符，138
ends, iostream 操纵符，138
enum
  不完整, 使用，60
  前向声明，60
  作用域限定符, 使用名称，60–61
errno
  保留值，167
  与 -fast 交互，167
-erroff 编译器选项，163
#error，36
error 函数，130
-errtags 编译器选项，164
-errwarn 编译器选项，164
export 关键字, 识别，168

**F**
-fast, 编译器选项，165–167
-features, 编译器选项，57, 94, 102, 167
files
  **另请参见**source files
-filt, 编译器选项，170
-flags, 编译器选项，172
flush, iostream 操纵符，131
flush, iostream 操纵符，138
-fnonstd, 编译器选项，172
-fns, 编译器选项，172–173
Fortran 运行时库, 链接，238
-fprecision=*p*, 编译器选项，174
-fround=*r*, 编译器选项，174–175
-fsimple=*n*, 编译器选项，175
fstream, 定义，128, 144
fstream.h
  iostream 头文件，129
fstream.h（**续**）
  使用，135
-ftrap, 编译器选项，177
__func__, 标识符，63
__FUNCTION__, 标识符，63

**G**
-G
  动态库命令，148
  选项描述，178
-g
  编译模板，90
  选项描述，179
-g3, 编译器选项，180
get, char 提取器，133
get 指针, streambuf，142
__global，57
-g0, 编译器选项，180

**H**
-H, 编译器选项，180
-h, 编译器选项，180–181
-help, 编译器，181
hex, iostream 操纵符，138
__hidden，58

**I**
.i, 文件名后缀，31
-I, 编译器选项，91, 181–182, 182–184
-i, 编译器选项，184
ifstream, 定义，128
.il, 文件名后缀，31
-include, 编译器选项，184
include 目录, 模板定义文件，91
include 文件, 搜索顺序，181, 182–184
-inline, 请参见 -xinline，185
I/O 库，127
-instances=*a*, 编译器选项，86, 185
-instlib, 编译器选项，186

iomanip.h, iostream 头文件，129, 139
iostream
  stdio，134–135, 141
  标准 iostream，110, 113, 190
  标准模式，127, 129, 190
  操纵符，138–141
  创建，135–137
  错误处理，134
  错误位，131
  定义，144
  复制，137
  格式，138
  构造函数，128
  混合使用新旧格式，191
  结构，128
  库，110, 113–114, 114
  流赋值，137
  使用，128–135
  手册页，127, 143–144
  输出错误，130–131
  输出到，129–131
  输入，131–132
  刷新，131
  头文件，128
  预定义的，127
  术语，144–145
  传统 iostream，110, 113, 190
iostream, 传统，113–114
iostream.h、iostream 头文件，129
ISO C++ 标准
  符合性，27
  一次定义规则，81–82, 90
ISO10646 UTF-16 文本字符串，278
istream 类, 定义，128
istrstream 类, 定义，128

## K

-keeptmp, 编译器选项，187
-Kpic, 编译器选项，149, 187
-KPIC, 编译器选项，149, 187

## L

-L, 编译器选项，111, 187
-l, 编译器选项，41, 109, 111, 188
LD_LIBRARY_PATH 环境变量，148
libc 库，109
libCrun 库，101, 102, 109, 111
libCstd 库, **请参见**C++ 标准库
libcsunimath, 库，110
libdemangle 库，110
libgc 库，110
libiostream, **请参见**iostream
libm
  库，109
  内联模板，240
  优化版本，241
-libmieee, 编译器选项，188
-libmil, 编译器选项，188
-library, 编译器选项，112, 114, 188–191
librwtool, 请参见 Tools.h++，111
libthread 库，109
limit, 命令，37
linking, iostream library，113
-lthread 编译器选项
  -xnolib 抑制，115
  使用 -mt 代替，101

## M

makefile 依赖性，244
mbarrier.h，104–105
-mc, 编译器选项，192
-misalign, 编译器选项，192
-mr, 编译器选项，192
-mt 编译器选项
  链接库，109
  选项描述，192–193
mutable 关键字, 识别，168

## N

-native, 编译器选项，193
nestedacess 关键字，169
-noex, 编译器选项，102, 193

-nofstore, 编译器选项, 193
-nolib, 编译器选项, 112, 193
-nolibmil, 编译器选项, 194
-norunpath, 编译器选项, 112, 194

## O

.o 文件
  保留, 32
  选项后缀, 31
-O, 编译器选项, 194
-O*level*, 编译器选项, 194
-o, 编译器选项, 194
oct, iostream 操纵符, 138
ofstream 类, 135
ostream 类, 定义, 128
ostrstream 类, 定义, 128
output, cout, 129

## P

+p, 编译器选项, 195
-P, 编译器选项, 195
PEC：可移植的可执行文件代码, 258
-pentium, 编译器选项, 196
-pg, 编译器选项, 196
-PIC, 编译器选项, 196
-pic, 编译器选项, 196
POSIX 线程, 192–193
#pragma align, 284
#pragma does_not_read_global_data, 285
#pragma does_not_return, 285
#pragma does_not_write_global_data, 285
#pragma dumpmacros, 286–287
#pragma end_dumpmacros, 287
#pragma error_messages, 287
#pragma fini, 287
#pragma ident, 288
#pragma init, 288
#pragma must_have_frame, 289
#pragma no_side_effect, 289
#pragma opt, 290
#pragma pack, 290
#pragma rarely_called, 292
#pragma returns_new_memory, 292
#pragma unknown_control_flow, 292
#pragma weak, 293
#pragma 关键字, 284–294
pragmas（指令）, 284–294
-pta, 编译器选项, 196
ptclean 命令, 85
pthread_cancel() 函数, 102
-pti, 编译器选项, 91, 196–197
-pto, 编译器选项, 197
-ptv, 编译器选项, 197
put 指针, streambuf, 142

## Q

-Qoption, 编译器选项, 197
-qoption, 编译器选项, 198
-qp, 编译器, 198
-Qproduce, 编译器选项, 198
-qproduce, 编译器选项, 199

## R

-R, 编译器选项, 112, 199
reinterpret_cast 运算符, 210
resetiosflags, iostream 操纵符, 138
RogueWave
  C++ 标准库, 121
  另请参见 Tools.h++, 111
rvalueref 关键字, 169
RWtools.h++, 113–114

## S

.s, 文件名后缀, 31
.S, 文件名后缀, 31
-S, 编译器选项, 199
-s, 编译器选项, 199
sbufpub, 手册页, 136
set_terminate() 函数, 102
set_unexpected() 函数, 102

setbase, iostream 操纵符，138
setfill, iostream 操纵符，139
setioflags, iostream 操纵符，139
setprecision, iostream 操纵符，139
setw, iostream 操纵符，138
shell, 限制虚拟内存，37
.so, 文件名后缀，31, 147
.so.*n*, 文件名后缀，31
Solaris 操作环境库，109
Solaris 线程，192–193
stabs 调试器数据格式，224
-staticlib, 编译器选项，112, 114, 199
__STDC__, 预定义宏，67
stdcxx4 关键字，189
stdio
  stdiobuf 手册页，141
  结合 iostream，134–135
stdiostream.h, iostream 头文件，129
STLport，123
STL（标准模板库，Standard Template Library）, 组件，121
stream.h, iostream 头文件，129
streambuf
  get 指针，142
  put 指针，142
  定义，141, 145
  队列式与文件式，142
  使用，142
  手册页，142
streampos，137
strstream, 定义，128, 145
strstream.h, iostream 头文件，129
struct, 匿名声明，61
__SUNPRO_CC_COMPAT, 预定义的宏，158
.SUNWCCh 文件名后缀，118
SunWS_cache，89
swap -s, 命令，37
__symbolic，58
-sync_stdio, 编译器选项，201

## T

tcov, -xprofile，266
-temp=*dir*, 编译器选项，202
-template, 编译器选项，85, 91, 202
terminate() 函数，102
__thread，59
调试器数据格式，224
调试
  选项，43–44
  准备程序，33, 180
-time, 编译器选项，203
Tools.h++
  编译器选项，114
  标准和兼容模式，111
  调试库，110
  文档，111
  传统和标准 iostream，111
-traceback, 编译器选项，203–204
traceback，203–204

## U

U"..." 形式的文本字符串，278
-U, 编译器选项，41, 204
ulimit, 命令，37
unexpected() 函数，102
-unroll=*n*, 编译器选项，205

## V

-V, 编译器选项，205
-v, 编译器选项，34, 205
__VA_ARGS__ 标识符，35–36
values, flush，131
-verbose, 编译器选项，85, 205
VIS 软件开发者工具包，280

## W

+w, 编译器选项，85, 207
-W 命令行选项，206
+w2, 编译器选项，207
-w, 编译器选项，207
#warning，36
ws, iostream 操纵符，134, 138

## X

X 插入器, iostream, 129
-xaddr32编译器选项, 208
-xalias_level, 编译器选项, 208
-xanalyze, 编译器选项, 210
-xannotate, 编译器选项, 211
-xar, 编译器选项, 87, 148, 211
-xarch=*isa*, 编译器选项, 212
-xautopar, 编译器选项, 215
-xbinopt, 编译器选项, 216
-xbinopt 编译器选项, 216
-xbuiltin, 编译器选项, 216
-xcache=*c*, 编译器选项, 217–218
-xcg, 编译器选项, 158
-xchar, 编译器选项, 218
-xcheck, 编译器选项, 220
-xchip=*c*, 编译器选项, 220
-xcode=*a*, 编译器选项, 222–224
-xdebugformat 编译器选项, 224
-xdepend, 编译器选项, 225
-xdumpmacros, 编译器选项, 225
-xe, 编译器选项, 228
-xF, 编译器选项, 228–229
-xhelp=flags, 编译器选项, 229
-xhreadvar, 编译器选项, 276
-xhwcprof 编译器选项, 230
-xia, 编译器选项, 230
-xinline, 编译器选项, 231
-xipo, 编译器选项, 233
-xipo_archive 编译器选项, 235
-xivdep, 编译器选项, 236
-xjobs, 编译器选项, 237
-xkeepframe, 编译器选项, 237
-xlang, 编译器选项, 238
-xldscope, 编译器选项, 57, 239
-xlibmieee, 编译器选项, 240
-xlibmil, 编译器选项, 240
-xlibmopt, 编译器选项, 241
-xlic_lib, 编译器选项, 241
-xlicinfo, 编译器选项, 241
-Xlinker, 编译器选项, 208
-xlinkopt, 编译器选项, 241
-xloopinfo, 编译器选项, 242
-Xm, 编译器选项, 208
-xM, 编译器选项, 243
-xM1, 编译器选项, 243
-xmaxopt, 编译器选项, 244
-xmaxopt 编译器选项, 244
-xMD, 编译器选项, 244
-xmemalign, 编译器选项, 245
-xMerge, 编译器选项, 244
-xMF, 编译器选项, 244
-xMMD, 编译器选项, 244
-xmodel, 编译器选项, 246
-xnolib, 编译器选项, 112, 115, 247
-xnolibmil, 编译器选项, 248
-xnolibmopt, 编译器选项, 248
-xO*level*, 编译器选项, 249–251
-xopenmp, 编译器选项, 251
-xpagesize, 编译器选项, 253
-xpagesize_heap, 编译器选项, 254
-xpagesize_stack, 编译器选项, 254
-xpec, 编译器选项, 258
-xpg, 编译器选项, 258–259
-xport64, 编译器选项, 259
-xprefetch, 编译器选项, 261
-xprefetch_auto_type, 编译器选项, 263
-xprefetch_level, 编译器选项, 264
-xprofile_ircache, 编译器选项, 267
-xprofile_pathmap, 编译器选项, 267
-xreduction, 编译器选项, 268
-xregs, 编译器选项, 150, 268–270
-xregs 编译器选项, 268
-xrestrict, 编译器选项, 270
-xs, 编译器选项, 271
-xsafe=mem, 编译器选项, 272
-xspace, 编译器选项, 272
-xtarget=*t*, 编译器选项, 272–275
-xtime, 编译器选项, 276
-xtrigraphs, 编译器选项, 277
-xunroll=*n*, 编译器选项, 278
-xustr, 编译器选项, 278
-xvector, 编译器选项, 279
-xvis, 编译器选项, 280
-xvpara, 编译器选项, 280
-xwe, 编译器选项, 280

## Z
-z *arg*, 编译器选项，282

## 半
半显式实例，86, 89

## 包
包括定义的模型，69

## 保
保留带符号字符，218

## 本
本地语言支持, 应用程序开发，28

## 编
编译, 内存要求，36–38
编译和链接，32
编译器
  版本, 不兼容，32
  诊断，34
  组件调用顺序，34

## 变
变量, 线程局部存储说明符，59
变量参数列表，35–36
变量的线程局部存储，59
变量声明说明符，57

## 标
标准, 符合性，27
标准 iostream 类，127
标准错误, iostream，127
标准模板库 (Standard Template Library, STL)，121
标准模式
  iostream，127, 129
  Tools.h++，111
  **另请参见** -compat，158
标准输出, iostream，127
标准输入, iostream，127
标准头文件
  实现，117–119
  替换，118

## 别
别名, 简化命令，38–39

## 并
并行化
  启用警告信息，280
  使用 -xautopar 为多个处理器启用，215
并行化, 使用 -xreduction，268

## 不
不兼容, 编译器版本，32

## 参
参数化操纵符, iostream，140–141

## 操
操纵符
  iostream，138–141
  无格式，139–140
  预定义的，138–139

**插**
插入
定义，144
运算符，129
插入运算符
iostream，129–131

**查**
查看输入，133

**成**
成员变量, 高速缓存，100

**程**
程序
基本生成步骤，29–30
生成多线程，101–102

**初**
初始化函数，288

**处**
处理器, 指定目标，272

**存**
存储大小，291

**错**
错误
位，131
状态, iostream，130
错误处理, 输入，134
错误消息，93
编译器版本不兼容，32
链接程序，33, 34

**大**
大小, 存储，291

**代**
代码生成, 内联函数和汇编程序, 编译组件，35
代码优化, 通过使用 -fast，165
代码优化器, 编译组件，35

**带**
带符号字符，218

**定**
定义, 搜索模板，90
定义独立模型，69

**动**
动态（共享）库，115–116, 148–149, 156, 180

**堆**
堆, 设置页面大小，253
堆栈, 设置页面大小，253

**对**
对齐
缺省，291
最严格，290–291

对象
库中, 链接时，147
临时，97
临时, 生存期，169
析构顺序，169

## 多
多个源文件, 使用，32
多媒体类型, 处理，280
多线程，192–193
编译，101
异常处理，102
应用程序，101

## 二
二进制输入, 读取，133
二进制文件优化，216

## 发
发行版信息，27

## 放
放置, 模板实例，86

## 非
非标准功能，57–66
已定义，27
允许非标准代码，168
非递增式链接编辑器, 编译组件，35

## 分
分析, -xprofile，264

## 浮
浮点
精度 (Intel)，176
无效，177
浮点插入器, iostream 输出，129

## 符
符号表, 可执行文件，199
符号声明说明符，57

## 复
复制
流对象，137
文件，142

## 覆
覆盖分析 (tcov)，266

## 赋
赋值, iostream，137

## 高
高速缓存
目录, 模板，32
优化器使用，217

## 格
格式控制, iostream，138

## 工
工作站, 内存要求，38

**共**

共享库
- 包含异常，149
- 不允许链接，161
- 从 C 程序访问，151
- 命名，180
- 生成，148–149, 178
- 生成, 异常，95

**构**

构造函数
- `iostream`，128
- 静态，148

**国**

国际化, 实现，28

**过**

过程间优化，233

过时, 不允许，168

**函**

函数
- 动态（共享）库中，148
- 覆盖，59
- 静态, 作为类友元，62–63
- 声明说明符，57
- 通过优化器内联，231

函数, `__func__` 中的名称，63

函数级重新排列，228

函数模板，71–72
- **另请参见**模板
- 定义，71
- 声明，71
- 使用，72

**后**

后缀
- `.SUNWCCh`，118
- 库，147
- 命令行文件名称，31
- 文件，117

**环**

环境变量
- `CCFLAGS`，39
- `LD_LIBRARY_PATH`，148
- `SUNWS_CACHE_NAME`，89

**缓**

缓冲区
- 定义，144
- 刷新输出，131

**汇**

汇编程序, 编译组件，35

汇编语言模板，280

**混**

混合语言链接，238

**活**

活前缀，256

**兼**

兼容模式
- `Tools.h++`，111
- **另请参见** `-compat`，158

## 交
交换空间，37

## 结
结构声明说明符，58

## 警
警告
- C 头文件替换，119
- 不可移植代码，207
- 存在问题的 ARM 语言构造，169
- 记时错误，207
- 降低可移植性的技术违规，207
- 无法识别的参数，34
- 无效代码，207
- 抑制，207

## 静
静态
- 变量, 引用，81–82
- 对象, 非局部初始化程序，169
- 归档库，147
- 函数, 引用，81–82
- 实例（过时的），86

静态链接
- 编译器提供的库，112, 199
- 库绑定，156
- 模板实例，88
- 缺省库，114–115

## 局
局部作用域规则, 启用和禁用，168

## 空
空白
- 前导，133
- 提取器，133–134
- 跳过，133, 139

## 库
库
- C++ 编译器, 提供，109
- C++ 标准，121
- C 接口，109
- Sun 性能库, 链接，241
- 共享，115–116, 161
- 后缀，147
- 类, 使用，113
- 链接顺序，42
- 链接选项，114
- 了解，147–148
- 命名共享库，180
- 配置宏，110
- 区间运算，230
- 生成共享库，223
- 使用，109–119
- 使用 `-mt` 链接，109
- 替换, C++ 标准库，116–119
- 优化的数学，241
- 传统 `iostream`，127

库, 生成
- C API，150
- 动态（共享），147
- 公用，150
- 静态（归档），147–151
- 链接选项，178
- 与异常共享，149
- 专用，149

## 扩
扩展功能，57–66
- 已定义，27
- 允许非标准代码，168

## 垃
垃圾收集
  库，111, 114

## 类
类
  传递，99
类库, 使用，113–114
类模板，72–74
  **另请参见**模板
  不完整，72
  参数, 缺省，76
  成员, 定义，73–74
  定义，72–73, 73
  静态数据成员，73–74
  声明，72–73
  使用，74
类声明说明符，58
类实例, 匿名，62

## 联
联合声明说明符，58

## 链
链接
  动态（共享）库，148, 156
  符号，118
  禁用系统库，247
  静态（归档）库，112, 114–115, 147, 156, 199
  库，109, 111, 114
  模板实例方法，86
  与编译分开，33
  与编译一致，33
链接程序作用域，57
链接时优化，241

## 流
流, 定义，145

## 幂
幂等性，67

## 命
命令行
  选项, 无法识别，34
  识别的文件后缀，31

## 模
模板
  编译，87
  标准模板库 (Standard Template Library, STL)，121
  部分专门化，77
  单独定义与包括定义的组织，90
  高速缓存目录，32
  静态对象, 引用，81–82
  链接，33
  命令，85
  内联，240
  内嵌，75
  排除定义搜索问题，91
  实例方法，86, 90
  系统信息库，89
  详细编译，85
  源文件，91
  专门化，76–77
模板定义
  独立，69
  独立, 文件，91
  排除定义搜索问题，91
  搜索路径，91
  已包含，69
模板实例化，74–75
  函数，74–75
  显式，74–75
  隐式，74

模板实例化（续）
整个类，85–86
模板问题，78–83
本地类型作为参数，78–79
非本地名称解析和实例化，78
静态对象，引用，81–82
模板函数的友元声明，79–80
排除定义搜索问题，91
在模板定义中使用限定名称，81
模板预链接程序，编译组件，35

## 目

目标文件
可重定位，149
链接顺序，42

## 内

内部函数，Intel MMX，65
内存边界内部函数，104–105
内存要求，36–38
内联函数
C++，何时使用，97–98
优化器，231
内联扩展，汇编语言模板，35

## 匿

匿名类实例，传递，62

## 配

配置宏，110

## 拼

拼写，替用，168

## 前

前端，编译组件，35

## 区

区间运算库，链接，230

## 全

全局
链接，87
实例，86

## 缺

缺省库，静态链接，114–115
缺省运算符，使用，98

## 三

三字母序列，识别，277

## 声

声明说明符
`__global`，57
`__hidden`，58
`__symbolic`，58
`__thread`，59

## 实

实例，选项，86
实例方法
半显式，89
静态，88
模板，86
全局，88
显式，89

实例化
模板函数，74–75
模板函数成员，75
模板类，75
模板类静态数据成员，75
实例状态, 一致，90

## 手

手册页
iostream，127, 136, 138, 140
访问，28, 111

## 输

输出，127
处理错误，130–131
二进制，131
缓冲区刷新，131
刷新，131
输入
iostream，131–132
查看，133
错误处理，134
二进制，133
输入/输出, complex，127

## 属

属性, 支持的，63

## 数

数学库, 优化版本，241

## 搜

搜索, 模板定义文件，90
搜索路径
include 文件, 已定义，181
搜索路径（续）
标准头文件实现，117
定义，91
动态库，112

## 提

提取
char*，132–133
定义，144
空白，133–134
用户定义 iostream，132
运算符，131–132

## 跳

跳过标志, iostream，133

## 头

头文件
C 标准，117
Intel MMX 内部函数声明，65
iostream，129, 139
sunmedia_intrin.h，65
标准库，116, 122–123
创建，67–68
幂等性，68
适应语言，67–68

## 外

外部
链接，87
实例，86

## 完

完成函数，287

**文**
文档, 访问，19
文档索引，19
文件
　C 标准头文件，117
　标准库，117
　打开和关闭，136
　对象，149
　多个源, 使用，32
　复制，136, 142
　可执行程序，32
　目标，32, 42
　使用 fstreams，135–137
　重新定位，137
文件描述符, 使用，136–137
文件名
　.SUNWCCh 文件名后缀，118
　后缀，31
　模板定义文件，91

**无**
无格式操纵符, iostream，139–140

**析**
析构函数, 静态，148

**显**
显式实例，86

**限**
限定指针，271

**陷**
陷阱操作模式，177

**信**
信号处理程序
　和多线程，102
　和异常，93

**性**
性能, 优化, 使用 -fast，165

**虚**
虚拟内存, 限制，37–38

**选**
选项, 命令行
　**请参见**字母列表下的各个选项
　C++ 编译器选项引用，156–282
　按函数汇总，42–52
　语法，41

**循**
循环，225
　-xloopinfo，242
　使用 -xreduction 约简，268

**页**
页面大小, 为堆栈或堆设置，253

**依**
依赖性, C++ 运行时库, 删除，150

**移**
移位运算符, iostream，139

## 异

异常
- longjmp 和，95
- setjmp 和，95
- 标准类，94
- 标准头文件，94
- 不允许，168
- 共享库，149
- 函数, 覆盖，59
- 和多线程，102
- 禁用，94
- 生成共享库，95
- 陷阱操作，177
- 信号处理程序和，95
- 预定义的，94–95

## 应

应用程序, 链接多线程，101

## 用

用户定义类型, iostream，129

## 优

优化
- 级别，249
- 链接时，241
- 目标硬件，272
- 使用 -fast，165
- 使用 -xmaxopt，244
- 数学库，241
- 用 pragma opt，290

优化器内存不足，38
优先级, 避免问题，129

## 右

右移运算符, iostream，131

## 语

语法
- CC 命令行，30
- 选项，41

语言
- C99 支持，238
- 支持本地，28

## 预

预编译的头文件，255
预处理程序, 定义宏，160
预定义的操纵符, iomanip.h，139
预取指令, 启用，261

## 源

源文件
- 链接顺序，42
- 位置约定，91

## 运

运算符
- iostream，129–131, 132

运行时库自述文件，124

## 在

在文件内重新定位, fstream，137

## 值

值
- float，129
- long，141
- 操纵符，129, 141
- 插入 cout，129

值类, 使用，99–100

**只**
只读存储器中的常量字符串，168
只读存储器中的文字字符串，168

**指**
指令 (pragma)，284–294

**中**
中间语言转换器，编译组件，35

**传**
传统 iostream，113–114

**子**
子程序，编译选项，33

**字**
字符，读取单个，133

**左**
左移运算符，`iostream`，129