# 内 容 提 要

GCC（GNU 编译程序集合）是目前最重要、应用最广的开放源代码软件。在本书中，专家 Arthur Griffith 提供了大量关于这种开源软件的信息，教您学会使用 GCC 为几乎所有的平台和语言开发及部署各类软件。

本书分为 3 个部分。第 1 部分，“自由软件编译程序”，介绍了编译程序的一些基础知识，以及一些可以用来下载和安装编译程序的指令。第 2 部分，“使用编译程序集合”，包含对编译程序指令的详细介绍。对 6 种主要的编程语言各用了一章的篇幅（并辅以一些实例）进行说明。另有一个特殊章节用来描述预处理程序以及连接由不同语言产生的目标文件的技术。第 3 部分，“外设和内设”，介绍了连接、调试、交叉编译、make 文件和 GNU 汇编器。第 3 部分也包含了编译程序前端和后端的内部工作信息。

本书是开源软件开发人员的必备学习资源，对广大的程序员而言，亦是一个有用的学习参考。

# 关于作者

Arthur Griffith 从 1977 年开始他的第一份编程工作，开发过编译程序、解释器、连接程序以及汇编器，曾作为小组成员为某些特殊用途的计算机开发过汇编器和连接程序。之后他加入 PL/EXUS 语言的一个编译程序（它的内部结构和 GCC 非常相似）的维护组。接下来的项目就是为 SATS 语言开发了交互式的解释器和编译程序。

以后的项目则涉及了 Forth 解释器的开发、COBOL 编译程序的扩展，以及某些用于机器控制的特殊目的解释性语言的开发。其中之一是一种交互式命令语言，提供对工业卫星通信系统的多站位地面控制。

在过去几年里，Arthur Griffith 开始转向编写计算机书籍、在线教授编程技术以及开发 Java 软件。他所著的编程书籍范围从 *Java*、*XML*、*and Jaxp* 到 *COBOL for Dummies*。他的很多软件开发项目都使用 GCC，而且当 Java 成为 GCC 语言的一部分后，编写本书就成为他的一个目标。

# 简　　介

虽然在当今的计算机界中，正在进行的自由软件运动不一定是最重要的，但我们确实正处于一个巨大的变革的中心。软件的所有权正在由公司专有变为可根据任何目的自由获取和使用。目前，自由软件不仅包括程序语言编译程序和连接程序，而且还有大量的工具，图形化的用户界面环境，甚至包括整个操作系统。

除此之外，还有一个事实就是所有的自由软件都是由 GCC 编译的，因此可否将 GCC 称作软件世界中最重要的一部分也就值得探讨了。当然，有很多种语言可用于编写程序，这些语言也都有很多编译程序，但这些编译程序中的绝大部分都是由 GCC 编写成并编译的。从某些方面来看，所有的自由软件都源起于 GCC。有些计算机公司已经开始放弃自己的编译程序而选择简单安装 GCC 编译程序了。使用 GCC 是免费的，而且还能得到持续的功能扩展和维护。

伴随着最近加入 GCC 家族的两种语言——Java 和 Ada——GCC 编译程序的羽翼扩展得更大更广了。这使得 GCC 可用的语言达到 6 种：C、C++、Objective-C、Fortran、Java 和 Ada。而且它还在不断地开发其他语言，例如 COBOL，如果有足够支持就会将这些语言也加入 GCC。

## 里程碑

GNU 项目始于 1984 年，目的是为了开发一种免费操作系统。Richard Stallman 是 GNU 项目的奠基人，并且是 GCC 最初的作者。

GCC 发布的第一个 beta 版，发布号为 0.9，是于 1987 年 3 月 22 日发布的。而第一个实际发布（版本 1.0）是在 1987 年 5 月 23 日发布的。从一开始到写本书时的版本 3.1，GCC 已经经历了 108 个发布。在过去的 15 年中，平均每 1.7 个月就会发布一个新的版本。

## 内幕

本书旨在为那些意欲使用 GCC 来开发软件的人提供有价值的信息。当中包含的有关 GCC 内幕的有用信息能让您轻松上手，开始在编译程序内部工作。但本书的主要目的还是要指导您一步步地安装并使用编译程序来开发软件。GCC 是个巨大的软件集合，而且像大

多数大型软件系统那样，它含有很多有用的功能。但只有发现了这些功能，明确知道它做了些什么，而且理解怎样使用它之后，才能更好地应用这些功能。这正是本书的主要目的。

本书分为 3 个部分。第 1 部分，“自由软件编译程序”，介绍了编译程序的一些基础知识，以及一些可以用来下载和安装编译程序的指令。第 2 部分，“使用编译程序集合”，包含对编译程序指令的详细介绍。对 6 种主要的编程语言各用了一章的篇幅（并辅以一些实例）进行说明。另有一个特殊章节用来描述预处理程序以及连接由不同语言产生的目标文件的技术。第 3 部分，“外设和内设”，介绍了连接、调试、交叉编译、makefiles 和 GNU 汇编器。第 3 部分也包含了编译程序前端和后端的内部工作信息。

GCC 拥有众多的可用命令行选项。附录 D 按字母顺序列出了这些选项，附录 C 中有对这些选项的交叉引用。第 21 章甚至包含更多的命令行选项——那些必须和特定计算机硬件一起使用以生成代码的选项。

为了加深您对本书涵盖的主题的理解，在此对每一章进行一个简短的介绍：

第 1 章是对 GCC 基本概念的总体介绍，包括它的内容和能够编译的语言的列表。

第 2 章介绍安装 GCC 的步骤。

第 3 章介绍预处理程序的工作过程，以及如何使用它来处理一种语言的源代码。

第 4 章包括编译和连接 C 语言的一些例子。

第 5 章包括编译和连接 C++语言的一些例子。

第 6 章包括编译和连接 Objective-C 语言的一些例子。

第 7 章包括编译和连接 Fortran 语言的一些例子。

第 8 章包括编译和连接 Java 语言的一些例子。

第 9 章包括编译和连接 Ada 语言的一些例子。

第 10 章包括了混合两种语言创建一个可执行程序的例子。

第 11 章解释如何在编译的程序中使用国际化工具，允许它根据所用的地域调整显示字符串。

第 12 章包括生成和使用静态库和共享库的例子。

第 13 章解释了使用 GNU 调试器的基本信息。

第 14 章描述 make 及其相关工具的用法。

第 15 章讨论 GNU 汇编器，并描述如何在 GCC 中使用它。

第 16 章描述如何配置 GCC 来编译和连接可在其他计算机上执行的程序的过程。

第 17 章描述如何使用 GCC 生成嵌入式系统的代码。

第 18 章包含的例子介绍如何从编译程序（而不是从目标代码）产生有用的输出。

第 19 章描述 lex 和 yacc 的初级用法，用来为 GCC 创建一种语言的前端。

第 20 章描述了由编译程序前端产生的中间语言的内容，而编译程序后端会读取该中间语言。

第 21 章包含了命令行选项列表，它们可应用于运行在指定硬件上的各种 GCC 版本。
附录 A 包含了 GNU 通用公共许可的拷贝。
附录 B 列出了影响 GCC 的环境变量。
附录 C 是按照类别交叉引用的命令行选项。
附录 D 是按照字母顺序排列的命令行选项列表。
附录 E 是术语表。

# 目　录

## 第 1 部分　自由软件编译程序


第 1 章　GCC 简介 .....3

1.1　GNU.....3
1.2　测量编译程序 .....4
1.3　命令行选项 .....4
1.4　平台 .....5
1.5　编译程序的功能.....6
1.6　语言 .....7
1.6.1　C 是基本语言 .....7
1.6.2　C++：第一个附加语言.....7
1.6.3　Objective-C .....8
1.6.4　Fortran.....8
1.6.5　Java .....8
1.6.6　Ada.....8
1.6.7　不再支持 Chill.....9
1.7　部分列表 .....9
1.8　联系方式 ..... 11

第 2 章　查询与安装编译程序..... 14

2.1　二进制代码下载.....15
2.2　FTP 资源下载.....15
2.3　CVS 资源下载.....17
2.3.1　以前的发布版.....18
2.3.2　测试版.....18
2.4　编译和安装 GCC.....19
2.4.1　安装过程.....19
2.4.2　配置选项.....20
2.5　binutils.....29
2.6　Win32 二进制安装 .....31
2.6.1　Cygwin.....31
2.6.2　安装.....31
2.7　运行测试包.....32


## 第 2 部分　使用编译程序集合


第 3 章　预处理程序 .....37

3.1　指示字 .....37
3.1.1　#define .....38
3.1.2　#error 和#warning.....41
3.1.3　#if，#elif，#else 和#endif.....41
3.1.4　#ifdef，#else 和#endif.....42
3.1.5　#include.....43
3.1.6　#include_next .....44

3.1.7 #line ..... 44
3.1.8 #pragma 和_Pragma ..... 45
3.1.9 #undef ..... 46
3.1.10 ## ..... 46
3.2 预定义宏 ..... 47
3.3 只包含一次头文件 ..... 49
3.4 在出错消息中包含定位信息 ..... 50
3.5 去掉适当位置的源代码 ..... 50
3.6 产生 make 程序的描述文件 ..... 51
3.7 命令行选项和环境变量 ..... 51
**第 4 章 编译 C 程序 ..... 53**
4.1 编译基础 ..... 53
4.1.1 单源程序到可执行程序 ..... 54
4.1.2 源文件到目标文件 ..... 54
4.1.3 多源文件到可执行文件 ..... 55
4.1.4 预处理 ..... 55
4.1.5 生成汇编语言 ..... 56
4.1.6 创建静态库 ..... 56
4.1.7 创建共享库 ..... 57
4.1.8 重写命名约定 ..... 58
4.2 标准 ..... 59
4.3 C 语言扩展 ..... 59
4.3.1 对齐 ..... 60
4.3.2 匿名联合 ..... 60
4.3.3 变长数组 ..... 61
4.3.4 零长度数组 ..... 61
4.3.5 属性 ..... 63
4.3.6 返回值的复合语句 ..... 68
4.3.7 条件操作数省略 ..... 69
4.3.8 枚举不完全类型 ..... 70
4.3.9 函数参数构造 ..... 70
4.3.10 函数内嵌 ..... 71
4.3.11 函数名 ..... 72
4.3.12 函数嵌套 ..... 72
4.3.13 函数原型 ..... 73
4.3.14 函数返回地址和堆栈框架 ..... 74
4.3.15 标识符 ..... 74
4.3.16 整数 ..... 75
4.3.17 更换关键字 ..... 75
4.3.18 标识地址 ..... 75
4.3.19 局部标识声明 ..... 76
4.3.20 左值表达式（lvalue） ..... 77
4.3.21 具有可变参数的宏 ..... 77
4.3.22 字符串 ..... 78
4.3.23 指针算法 ..... 78
4.3.24 Switch/Case 分支语句 ..... 78
4.3.25 typedef 名字创建 ..... 79
4.3.26 typeof 引用 ..... 80
4.3.27 联合强制类型转换 ..... 80
**第 5 章 编译 C++程序 ..... 82**
5.1 编译基础 ..... 82
5.1.1 单源文件到可执行文件 ..... 82
5.1.2 多源文件到可执行文件 ..... 83
5.1.3 源文件到目标文件 ..... 84
5.1.4 预处理 ..... 85
5.1.5 生成汇编语言 ..... 85
5.1.6 创建静态库 ..... 85
5.1.7 创建共享库 ..... 87
5.2 C++语言的扩展功能 ..... 89
5.2.1 属性 ..... 89
5.2.2 头文件 ..... 90
5.2.3 函数名 ..... 90
5.2.4 接口与实现 ..... 91
5.2.5 操作符<?和>? ..... 92
5.2.6 约束 ..... 93
5.3 编译程序操作 ..... 94
5.3.1 库 ..... 94
5.3.2 mangling 名字 ..... 94

5.3.3 连接 ..... 97
5.3.4 编译模板实例化 ..... 98
第 6 章 编译 Objective-C ..... 99
6.1 编译基础 ..... 99
6.1.1 单独源文件到可执行程序 ..... 99
6.1.2 编译对象 ..... 100
6.1.3 创建静态连接库 ..... 102
6.1.4 创建共享库 ..... 104
6.2 Objective-C 的简要介绍 ..... 105
6.2.1 预定义类型 ..... 105
6.2.2 创建接口声明 ..... 106
6.2.3 命名和 mangling ..... 106
第 7 章 编译 Fortran 程序 ..... 108
7.1 编译基础 ..... 108
7.1.1 单独源文件到可执行程序 ..... 108
7.1.2 多源文件到可执行程序 ..... 109
7.1.3 生成汇编语言 ..... 110
7.1.4 预处理 ..... 110
7.1.5 创建静态库 ..... 112
7.1.6 创建共享库 ..... 113
7.2 Ratfor ..... 113
7.3 GNU Fortran 扩展功能与变量 ..... 115
7.3.1 内在函数 ..... 115
7.3.2 源代码格式 ..... 115
7.3.3 注释 ..... 116
7.3.4 美元符号 ..... 116
7.3.5 大小写敏感 ..... 116
7.3.6 特殊 Fortran 90 功能 ..... 118
第 8 章 编译 Java ..... 123
8.1 编译基础 ..... 123
8.1.1 单一源文件到二进制可执行程序 ..... 124
8.1.2 单一源文件到类文件 ..... 124
8.1.3 单一源文件到二进制目标文件 ..... 125
8.1.4 类文件到本地可执行程序 ..... 125
8.1.5 多源文件到二进制可执行文件 ..... 125
8.1.6 多输入文件到可执行程序 ..... 127
8.1.7 生成汇编语言 ..... 127
8.1.8 创建静态库 ..... 128
8.1.9 创建共享库 ..... 129
8.1.10 创建 Jar 文件 ..... 129
8.2 Java 工具 ..... 130
8.2.1 gij ..... 130
8.2.2 jar ..... 131
8.2.3 gcjh ..... 132
8.2.4 jcf-dump ..... 134
8.2.5 jv-scan ..... 134
8.2.6 jv-convert ..... 135
8.2.7 grepjar ..... 136
8.3 RMI ..... 137
8.3.1 rmic ..... 138
8.3.2 rmiregistry ..... 139
8.4 属性 ..... 139
第 9 章 编译 Ada 程序 ..... 141
9.1 安装 ..... 141
9.2 编译基础 ..... 143
9.2.1 单独源文件到可执行程序 ..... 144
9.2.2 多源文件到可执行程序 ..... 145
9.2.3 源代码到汇编语言 ..... 146
9.3 选项 ..... 146
9.4 Ada 工具 ..... 150
9.4.1 gnatbind ..... 150
9.4.2 gnatlink ..... 153
9.4.3 gnatmake ..... 153
9.4.4 gnatchop ..... 155
9.4.5 gnatxref ..... 156
9.4.6 gnatfind ..... 157

9.4.7 gnatkr ..... 158
9.4.8 gnatprep ..... 158
9.4.9 gnatls ..... 159
9.4.10 gnatpsys 和 gnatpsta ..... 161
**第 10 章 混合语言 ..... 162**
10.1 C++和 C 的混合 ..... 162
10.1.1 在 C++中调用 C ..... 162
10.1.2 在 C 中调用 C++ ..... 163
10.2 混合 Objective-C 和 C ..... 164
10.2.1 在 Objective-C 中调用 C ..... 164
10.2.2 在 C 中调用 Objective-C ..... 165
10.3 混合 Java 和 C++ ..... 166
10.3.1 创建 Java 字符串并调用静态方法 ..... 167
10.3.2 加载并实例化 Java 类 ..... 168
10.3.3 异常 ..... 170
10.3.4 CNI 的数据类型 ..... 171
10.4 混合 Java 和 C ..... 171
10.4.1 具有本地方法的 Java 类 ..... 172
10.4.2 向本地方法传递参数 ..... 173
10.4.3 在 C 代码中调用 Java 类方法 . 175
10.5 混合 Fortran 和 C ..... 177
10.5.1 从 Fortran 中调用 C ..... 177
10.5.2 从 C 中调用 Fortran ..... 178
10.6 混合 Ada 和 C ..... 179
10.6.1 从 Ada 中调用 C ..... 180
10.6.2 从 Ada 中调用 C（带参数）... 181
**第 11 章 国际化 ..... 184**
11.1 翻译的例子 ..... 184
11.2 创建新的.po 文件 ..... 186
11.3 函数 gettext()的用法 ..... 189
11.3.1 静态字符串 ..... 189
11.3.2 从其他域中进行翻译 ..... 189
11.3.3 从其他域到指定类别的翻译 ... 190
11.3.4 多重模式 ..... 190
11.3.5 来自其他域的多重模式 ..... 190
11.3.6 某个类别其他域的多重模式 ... 190
11.4 合并两个.po 文件 ..... 190
11.5 由.po 文件生成二进制的.mo 文件 .. 192

# 第 3 部分 外设和内设

**第 12 章 连接和库 ..... 197**
12.1 目标文件和库 ..... 197
12.1.1 目录中的目标文件 ..... 197
12.1.2 静态库中的目标文件 ..... 198
12.1.3 动态库中的目标文件 ..... 200
12.2 连接程序的前端 ..... 200
12.3 定位库 ..... 201
12.3.1 在连接时定位库 ..... 201
12.3.2 运行时载入库 ..... 201
12.4 从共享库中载入函数 ..... 202
12.5 使用目标文件和库的工具程序 ..... 204
12.5.1 配置查找共享库 ..... 204
12.5.2 列举目标文件中的符号名 ..... 205
12.5.3 去除目标文件中的无用信息 ... 207
12.5.4 列出共享库的依赖关系 ..... 208
12.5.5 显示目标文件的内部结构 ..... 209

第 13 章　使用 GNU 调试器.....212

13.1　调试信息格式.....212

13.1.1　STABS.....212

13.1.2　DWARF.....213

13.1.3　COFF.....213

13.1.4　XCOFF.....214

13.2　编译调试用程序.....214

13.3　将程序载入调试器.....215

13.4　执行事后检查.....219

13.5　将调试器附带到运行程序上.....220

13.6　命令总结.....223

第 14 章　make 和 Autoconf.....225

14.1　make.....225

14.1.1　内部定义.....227

14.1.2　如何编写 makefile.....228

14.1.3　make 的选项.....229

14.2　Autoconf.....232

第 15 章　GNU 汇编器.....237

15.1　用命令行进行汇编.....237

15.2　绝对、相对以及边界.....239

15.3　内嵌汇编.....240

15.4　汇编器指示字.....243

第 16 章　交叉编译及窗体端口.....254

16.1　目标机.....254

16.2　创建交叉编译程序.....255

16.2.1　安装本地编译程序.....255

16.2.2　为目标机器建立 binutils.....256

16.2.3　安装目标机器的文件.....257

16.2.4　可配置库 libgcc1.a.....257

16.2.5　构造交叉编译程序.....257

16.2.6　运行交叉编译程序.....258

16.3　MinGW.....258

16.4　Cygwin.....259

16.4.1　编译简单的 Cygwin 操作台程序.....259

16.4.2　编译 Cygwin GUI 程序.....260

第 17 章　嵌入式系统.....261

17.1　建立编译程序和连接程序.....261

17.2　选择一种语言.....262

17.3　GCC 嵌入工具.....263

17.3.1　命令行选项.....263

17.3.2　诊断.....264

17.3.3　汇编器代码.....264

17.4　库.....264

17.4.1　整顿标准库.....265

17.4.2　为嵌入式系统设计的库.....265

17.5　GNU 连接程序脚本语言.....265

17.5.1　脚本例子 1.....266

17.5.2　脚本例子 2.....267

17.5.3　其他脚本命令.....268

第 18 章　编译程序输出.....269

18.1　程序信息.....269

18.1.1　解析树.....269

18.1.2　头文件.....270

18.1.3　程序需要的内存.....271

18.1.4　时耗.....272

18.1.5　C++中间树.....272

18.1.6　C++类层次.....273

18.2　有关 makefile 的信息.....273

18.3　编译程序信息.....274

18.3.1　编译时间.....275

18.3.2　子进程开关.....275

18.3.3　详细的编译程序调试信息.....276

18.4　关于文件和目录的信息.....279

第 19 章 实现一种语言 .....280
19.1 从前到后 ..... 280
19.2 词法扫视 ..... 281
19.2.1 简单的词法扫视程序..... 282
19.2.2 Lex 的正则表达式 ..... 282
19.3 语法分析 ..... 283
19.4 创建解析树 ..... 288
19.5 连接后端与前端..... 289
第 20 章 寄存器传送语言 .....292
20.1 RTL insn ..... 292
20.1.1 6 种基本表达式代码..... 292
20.1.2 insn 的类型和内容..... 294
20.2 模式和模式类..... 310
20.3 标志 ..... 312
第 21 章 机器相关的编译程序选项....314
21.1 机器列表 ..... 314
21.2 GCC 命令行选项 ..... 315
21.2.1 Alpha 选项..... 315
21.2.2 Alpha/VMS 选项 ..... 320
21.2.3 ARC 选项 ..... 320
21.2.4 ARM 选项 ..... 320
21.2.5 AVR 选项..... 325
21.2.6 CRIS 选项..... 326
21.2.7 D30V 选项..... 329
21.2.8 H8/300 选项..... 330
21.2.9 HPPA 选项..... 330
21.2.10 IA-64 选项 ..... 332
21.2.11 Intel 386 和 AMD x86-64 选项 ..... 334
21.2.12 Intel 960 选项 ..... 339
21.2.13 M32R/D 选项 ..... 340
21.2.14 M680x0 选项 ..... 341
21.2.15 M68HClx 选项 ..... 344
21.2.16 M88K 选项 ..... 344
21.2.17 MCore 选项 ..... 347
21.2.18 MIPS 选项 ..... 348
21.2.19 MMIX 选项 ..... 354
21.2.20 MN10200 选项 ..... 355
21.2.21 MN10300 选项 ..... 355
21.2.22 NS32K 选项..... 356
21.2.23 PDP-11 选项 ..... 358
21.2.24 RS/6000 和 PowerPC 选项..... 359
21.2.25 RT 选项..... 368
21.2.26 S/390 和 zSeries 选项 ..... 369
21.2.27 SH 选项..... 370
21.2.28 SPARC 选项 ..... 372
21.2.29 System V 选项 ..... 376
21.2.30 TMS320C3x/C4x 选项 ..... 376
21.2.31 V850 选项..... 379
21.2.32 VAX 选项..... 380
21.2.33 Xstormy16 选项..... 380

# 第 4 部分　附录

附录 A　GNU 通用公共许可证.....383
附录 B　环境变量 .....389
附录 C　命令行对照表 .....393
附录 D　命令行选项.....400
D.1　选项前缀.....400
D.1.1　命令行的顺序.....401
D.1.2　文件类型.....401
D.2　选项字母顺序列表.....402
附录 E　术语表 .....480

The Complete Reference

# 第 1 部分

# 自由软件编译程序

# 第 1 章 GCC 简介

GCC（GNU Compiler Collection，GNU 编译程序集合）是最重要的开放源码软件。事实上，其他所有开放源码软件都在某种层次上依赖于它。甚至其他语言，例如 Perl 和 Python，都是由 C 语言开发的，由 GNU 编译程序编译的。

GCC 编译程序的历史很有趣，远远不止是一个时间和事件的列表。这个软件对于整个自由软件运动而言具有根本性的意义。事实上，如果没有它或类似的软件，就不可能有自由软件运动。GCC 为 Linux 的出现提供了可能性。

本章概要介绍了 GCC 编译程序集合，以及它的相关工具。这些编译中使用的工具可以跟踪源代码、编辑文件、控制编译过程、提供调试信息。

本章介绍的内容包括一个列表以及对处理过程的一些描述。该列表描述了组成编译程序集合的文件和程序。之后介绍了将源文件变成可连接和可执行程序的步骤。

## 1.1 GNU

GCC 是 GNU 项目的一个产品。该项目始于 1984 年，目标是以自由软件的形式开发一个完整的类 UNIX 的操作系统。像所有这种规模的软件一样，GNU 项目也经历了一些波折，但目标最终还是实现了。实际上现在一个功能完备的类 UNIX 操作系统——Linux，已经在世界上广为流传了，并被不计其数的公司、政府和个人成功应用。而该系统及其所有工具和应用都是基于 GCC 的。

可用于 Linux 以及其他系统的自由软件的范围很广泛，并且还在日益增长。作为整体 GNU 项目的一部分而开发的免费 UNIX 被列在 http://www.gnu.org/directory 中的自由软件目录（Free Software Directory）中。

成千上万的程序员都在为各种 GNU 项目（及其他自由软件项目）作贡献，而实际上所有这些都在某种程度上依赖于 GCC。

## 1.2 测量编译程序

我们可以在编译的速度、生成代码的速度，以及生成代码的尺寸上对编译程序进行比较。但是很难进行更深入的比较，因为虽然可以得出一些数字，却很难对这些数字赋予某种实际意义。例如，源文件的数目（make 程序的描述文件、配置文件、头文件、可执行代码，等等）显示共有超过 15 000 个的各种文件。源文件编译成的目标文件、库和可执行程序的数目成千增长。代码的行数（这 15 000 多个文件的行数）超过 3 700 000。从任何标准来看，这都是一个大程序。

代码的质量参差不齐——因为有如此之多的程序员参与开发过程，而且代码注释中也内嵌了大量的内部文档，所以文档的质量和数量也有变化。所幸的是，有大量的程序员正在努力地提高代码和注释的质量。而且，也不是必须阅读内嵌的注释才能使用编译程序。但如果要对编译程序做些工作，你会发现还是要花些时间阅读代码中内嵌的注释。

测量编译程序的质量的惟一方法是询问它的用户。全世界的用户数目很难估计（自由软件就有这样的特性），但一定是巨大的。它被用于某些版本的 UNIX，这些 UNIX 自带系统供应商提供的本地编译程序。事实上，我知道一个很大的 UNIX 供应商就在自己内部的项目中使用 GCC，即便该供应商也有自己的很优秀的编译程序。

GCC 编译程序从未停止过改进。如同第 2 章所描述的，通过下载某个特定版本的源代码便可安装已经发布的某个版本的 GCC，也可以直接下载最新的（或测试中的）版本。测试中的版本时刻都在改进。有些更正是修改已有的 bug，还有一些是为了加入新的语言和功能，还有一些是为了去掉某些不再应用的功能。如果你曾经使用过 GCC，隔一段时间再用最新版，一定会发现一些变化。

## 1.3 命令行选项

每个命令行选项都以一个或一对连字号开始。例如，下面的命令行会编译 ANSI 标准 C 程序 muxit.c，再产生一个非连接的目标文件 muxit.o：

```
gcc -ansi -c muxit.c -o muxit.o
```

这些单字母选项后面跟着的名字可以和字母之间留有空格。例如，选项-omuxit.o 和-o muxit.o 是一样的。

下面的命令用-v 代表详细说明，而--help 会打印可用的选项，而且会打印一个详细的包括所有命令行选项的列表，包括那些适用于特定语言的选项。

```
gcc -v --help
```

有可能构造一些实际不做任何事情的命令行。例如，下面的命令将目标文件交给编译程序，然后指定-c 选项防止激活连接程序：

```
gcc -c brookm.o
```

所有的命令行选项大致可分为三类：

- **指定语言**　GCC 编译程序有能力编译多种语言，有些选项只可用于其中的一两种。例如，-C89 选项只应用于 C 语言，指定适用于 1989 年的标准。
- **指定平台**　GCC 编译程序可以为多种平台生成目标代码，而有些选项只能应用于为某个指定平台生成代码。例如，如果输出平台是 Intel 386，那么-fp-ret-in-387 选项可用来指出要将函数调用返回的浮点数保存在硬件的浮点寄存器中。
- **普适**　很多选项对所有语言和平台都适用。例如，-O 选项指示编译程序要优化输出代码。

使用编译程序不知道的选项总会产生出错消息。使用目标平台不适用的选项也会产生出错消息。

gcc 程序能够自己处理所有已知的选项，而将未知其他选项传递下去来编译指定语言。如果传递下去的选项对指定语言的处理器是未知的，就会报错。

选项可以指示 gcc 只执行特定的操作（例如连接或预处理）或什么都不做，这就是说有些标志没有什么特殊目的。除非用-W 选项产生特殊警告，否则这些标志可被识别但不采取任何动作，只简单忽略而已。

## 1.4　平台

GCC 编译程序集合可以在很多平台上运行。平台是指特定计算机芯片及其运行的操作系统的组合。

尽管 GCC 已经被移植到数以千计的硬件/软件的组合上，但只有一些基本平台可以用来测试发布的正确性。这些列在表 1-1 中的基本目标平台是最流行的，而且它们对 GCC 支持的其他平台具有代表性。

**表 1-1　GCC 应用的主要平台**

| 硬件 | 操作系统 |
|---|---|
| Alpha | Red Hat Linux 7.1 |
| HPPA | HPUX 11.0 |
| Intel x86 | Debian Linux 2.2、 Red Hat Linux 6.2 和 FreeBSD 4.5 |
| MIPS | IRIX 6.5 |

（续表）

| 硬件 | 操作系统 |
| --- | --- |
| PowerPC | AIX 4.3.3 |
| Sparc | Solaris 2.7 |

要注意保证 GCC 可以在表 1-1 中列出的主要平台上正确运行，而且也要很好地处理其次列在表 1-2 中的平台。

表 1-2 GCC 应用的次要平台

| 硬件 | 操作系统 |
| --- | --- |
| PowerPC | Linux |
| Sparc | Linux |
| ARM | Linux |
| Intel x86 | Cygwin |

在这样少的主要和次要平台上测试的原因是人力问题。即使你的平台没有被列在上面，编译程序还是可能在系统上能够良好运行的。而且，完整的测试集合和编译程序的源代码一起提供，所以很容易验证编译程序是否工作正常。另一种方法就是作为志愿者来测试你的平台，这样编译程序就可以在每次发布之前得到测试。

## 1.5 编译程序的功能

编译程序是一个翻译器。它读入一种语言格式的指令（通常是文本形式的编程语言），并将它们翻译成可在计算机上运行的指令集合（通常是二进制硬件指令的集合）。

大体上讲，编译程序可以分为两部分：前端和后端。前端读出程序的源代码，将找到的内容以树的形式转换到内存驻留表（memory-resident table）中。一旦构造了该树，编译程序的后端就会读出树中保存的信息，并将它们转换成目标机器上的汇编语言。

下面是关于将源文件翻译成可执行程序的大致步骤：

- 词法分析是编译程序前端的最开始部分。它从输入中读出字符，确定哪些是在一起的，形成符号、数字和标点符号。
- 语法分析处理会读入来自词法浏览器的符号流，以及后面跟着的一个规则集合，确定它们之间的关系。语法分析器的输出结果是树结构，会被传递给编译程序的后端。
- 语法分析树结构会被翻译成伪汇编语言（psuedo-assembly language），叫做寄存器传送语言（Register Transfer Language，RTL）。

- 编译程序的后端由分析 RTL 代码开始，然后执行一些优化操作。代码中冗余和未被使用的部分会被去掉。树中有些部分会被移动到其他位置以防止语句被不必要地多次执行。总的说来，有十个以上的优化操作，而且有些优化操作会多次浏览代码。
- RTL 被翻译成目标机器上的汇编语言。
- 激活汇编器去将汇编语言翻译成目标文件。该文件不是可执行格式——它包括可执行的目标代码，但并不是最终运行的形式。另外，它更可能包括未解析的到其他模块例程和数据的引用。
- 连接程序将来自汇编器的目标文件（其中有些可能保存在包含目标文件的库中）组合成可执行程序。

注意，前端和后端是完全分离开的。任何语言都可用语法分析器产生树结构，而由 GCC 进行编译。类似地，任何机器只要能将程序的树结构翻译成汇编语言，就能够编译由前端处理的所有语言。

实际操作过程绝对不像描述的那样简单，但这确实能够实现。

## 1.6　语言

GCC 可以编译多种语言，但所有这些语言之间有个基本关系。语法分析器由于每种语言语法的惟一性而完全不同，但随着编译过程的前进，这些语言的代码就越来越相似。如前所述，GNU 编译集合可以接受任何形式编程语言的输入，产生的输出也可以在很多不同平台上运行。

### 1.6.1　C 是基本语言

GCC 的基本语言是 C 语言。整个编译系统由 C 编译程序开始，然后渐渐加了其他的语言进来。幸运的是 C 语言是系统级的语言，能够直接处理计算机程序的基本元素，因此在它上面再创建其他语言的编译程序就相对容易得多。

如果你用其他语言而不是 C 语言编程，而你也对 GCC 很熟悉，你会发现很多东西都是以 C 语言的形式出现的。可将 C 语言想成一种位于 GCC 编译程序的汇编语言之下的语言。大多数编译程序本身都是由 C 语言实现的。

### 1.6.2　C++：第一个附加语言

C++语言是 C 语言的直接扩展（只有很小的改动），因此要向 GCC 加入其他语言，它是最佳首选。C++能够完成的所有事情 C 语言都可以做到，所以没有必要修改编译程序的后端——只需要在前端载入一个新的语法语义分析器。一旦产生中间语言，编译程序的其他部分就和 C 语言完全一样了。

### 1.6.3 Objective-C

Objective-C 并不像 C 语言或 C++语言那样流行，但这是另一种源自（并基于）C 语言的语言。它被看作是“对象化的 C 语言”，事实上也是如此。很大程度上，可以编写 C 程序，而被当作 Objective-C 编译并运行。与基本 C 语法完全不同的特殊语法是用来定义对象的，所以它和纯粹的 C 代码没有什么混淆与冲突。

### 1.6.4 Fortran

Fortran 可以做到而 C 不能做到的事情就是：科学计算。标准 Fortran 函数库（这是 Fortran 的精髓，因为它就是语言的一部分）是一种扩展，已经趋于完美，而且也历经很多年了。Fortran 如今被用于科学计算是因为它的基本能力就是快速而准确地实现复杂计算。Fortran 甚至还将复杂的数字作为它的基本数据类型，而基本数值数据类型可具有很高的精确度。

这种语言的结构比其他的现代语言要麻烦一些，但它包含的一些基本函数和功能的实现是结构化编程所必需的。最新的 Fortran 标准在这一点上有所扩展，无疑是一种非常现代的语言。

### 1.6.5 Java

Java 是包含进 GCC 的最年轻的语言。Java 语言和 C++一样是基于 C 语言的，但它使用了一些不同的方法来实现类的语法。C++更加灵活，Java 则是通过限制对象的构造函数和析构函数去除了 C++的不确定性，它继承的是一些严格无歧义的形式。

Java 和 GCC 包含的其他语言有很大的区别，这是由它对象代码的形式决定的。Java 会编译成一种对象代码的特殊格式，作为字节码（bytecodes）被解释器（叫做 Java 虚拟机）执行。所有 Java 程序都按照这种方式运行，直到 GCC 编译程序增加选项，通过挂接一个 Java 前端到已存在的 GCC 后端来产生本地可执行代码。另外，还添加了一个前端来读出 Java 字节码，并作为源代码用来产生本地可执行的二进制代码。

### 1.6.6 Ada

最新增加到 GCC 家族的是 Ada。它是作为一个功能完善的编译程序而加入的，最早是由 Ada Core Technologies 公司独立开发作为 GNAT Ada 95 编译程序，在 2001 年 10 月捐赠给 GCC。

Ada 编译程序的前端和其他语言的不同，它是由 Ada 编写的。一般来说，安装了 Ada 编译程序就可以了，但在一些系统中会需要特殊的引导过程。而所有其他语言都是由 C 和 C++编写的，因此几乎都可以普遍移植。

作为一种语言，Ada 是专门为多个程序员编写大型程序而设计的。在编译 Ada 程序的时候，它交叉引用程序其他部分的源代码来验证正确性。这种语言的语法要求每个函数和

过程都要被声明为包的一部分，而包的配置是和声明相匹配的。C 和 C++语言用原型来声明外部引用函数，而 Java 使用文件命名规则定位包的成员，但这两种技术都不像 Ada 那样严格。

### 1.6.7　不再支持 Chill

GCC 在 3.0 版之后不再支持 Chill 语言。就在发布版本 3.1 之前，Chill 语言的源代码也从 GCC 中移走了。但 GCC 非常复杂，而 Chill 语言完整地作为其一部分已经存在一段时间了，所以还会从 GCC 在线文档中和源代码的各种不同位置中看到对 Chill 语言的引用。本书是在这种转换过程中编写的，所以还会涉及到 Chill 编译程序选项和文件类型。

## 1.7　部分列表

GCC 是由许多组件组成的。表 1-3 列出了 GCC 的各个部分，但它们也并不总是出现的。有些部分是和语言相关的，所以如果没有安装某种特定语言，系统中就不会出现相关的文件。

**表 1-3　GCC 安装的各个部分**

| 部分 | 描述 |
|---|---|
| c++ | gcc 的一个版本，默认语言设置为 C++，而且在连接的时候自动包含标准 C++库。这和 g++一样 |
| cc1 | 实际的 C 编译程序 |
| cc1plus | 实际的 C++编译程序 |
| collect2 | 在不使用 GNU 连接程序的系统上，有必要运行 collect2 来产生特定的全局初始化代码（例如 C++的构造函数和析构函数） |
| configure | GCC 源代码树根目录中的一个脚本。用于设置配置值和创建 GCC 编译程序必需的 make 程序的描述文件 |
| crt0.o | 这个初始化和结束代码是为每个系统定制的，而且也被编译进该文件，该文件然后会被连接到每个可执行文件中来执行必要的启动和终止程序 |
| cygwin1.dll | Windows 的共享库提供的 API，模拟 UNIX 系统调用 |
| f77 | 该驱动程序可用于编译 Fortran |
| f771 | 实际的 Fortran 编译程序 |
| g++ | gcc 的一个版本，默认语言设置为 C++，而且在连接的时候自动包含标准 C++库。这和 c++一样 |
| gcc | 该驱动程序等同于执行编译程序和连接程序以产生需要的输出 |

（续表）

| 部分 | 描述 |
| --- | --- |
| gcj | 该驱动程序用于编译 Java |
| gnat1 | 实际的 Ada 编译程序 |
| gnatbind | 一种工具，用于执行 Ada 语言绑定 |
| gnatlink | 一种工具，用于执行 Ada 语言连接 |
| jc1 | 实际的 Java 编译程序 |
| libgcc | 该库包含的例程被作为编译程序的一部分，是因为它们可被连接到实际的可执行程序中。它们是特殊的例程，连接到可执行程序，来执行基本的任务，例如浮点运算。这些库中的例程通常都是平台相关的 |
| libgcj | 运行时库包含所有的核心 Java 类 |
| libobjc | 对所有 Objective-C 程序都必须的运行时库 |
| libstdc++ | 运行时库，包括定义为标准语言一部分的所有的 C++类和函数 |

表 1-4 列出的软件和 GCC 协同工作，目的是实现编译过程。有些是很基本的（例如 as 和 ld），而其他一些则是非常有用但不是严格需要的。尽管这些工具中的很多都是各种 UNIX 系统的本地工具，但还是能够通过 GNU 包 binutils 得到大多数工具。安装 binutils 的过程将在第 2 章中介绍。

**表 1-4 GCC 使用的软件工具**

| 工具 | 描述 |
| --- | --- |
| addr2line | 给出一个可执行文件的内部地址，addr2line 使用文件中的调试信息将地址翻译成源代码文件名和行号。该程序是 binutils 包的一部分 |
| ar | 这是一个程序，可通过从文档中增加、删除和析取文件来维护库文件。通常使用该工具是为了创建和管理连接程序使用的目标库文档。该程序是 binutils 包的一部分 |
| as | GNU 汇编器。实际上它是一族汇编器，因为它可以被编译或能够在各种不同平台上工作。该程序是 binutils 包的一部分 |
| autoconf | 产生的 shell 脚本自动配置源代码包去编译某个特定版本的 UNIX |
| c++filt | 程序接受被 C++编译程序转换过的名字（不是被重载的），而且将该名字翻译成初始形式。该程序是 binutils 包的一部分 |
| f2c | 是 Fortran 到 C 的翻译程序。不是 GCC 的一部分 |
| gcov | gprof 使用的配置工具，用来确定程序运行的时候哪一部分耗时最大 |
| gdb | GNU 调试器，可用于检查程序运行时的值和行为 |
| GNATS | GNU 的调试跟踪系统（GNU Bug Tracking System）。一个跟踪 GCC 和其他 GNU 软件问题的在线系统 |

（续表）

| 工具 | 描述 |
|---|---|
| gprof | 该程序会监督编译程序的执行过程，并报告程序中各个函数的运行时间，可以根据所提供的配置文件来优化程序。该程序是 binutils 包的一部分 |
| ld | GNU 连接程序。该程序将目标文件的集合组合成可执行程序。该程序是 binutils 包的一部分 |
| libtool | 一个基本库，支持 make 程序的描述文件使用的简化共享库用法的脚本 |
| make | 一个工具程序，它会读 makefile 脚本来确定程序中的哪个部分需要编译和连接，然后发布必要的命令。它读出的脚本（叫做 makefile 或 Makefile）定义了文件关系和依赖关系 |
| nlmconv | 将可重定位的目标文件转换成 NetWare 可加载模块（NetWare Loadable Module，NLM）。该程序是 binutils 的一部分 |
| nm | 列出目标文件中定义的符号。该程序是 binutils 包的一部分 |
| objcopy | 将目标文件从一种二进制格式复制和翻译到另外一种。该程序是 binutils 包的一部分 |
| objdump | 显示一个或多个目标文件中保存的多种不同信息。该程序是 binutils 包的一部分 |
| ranlib | 创建和添加到 ar 文档的索引。该索引被 ld 使用来定位库中的模块。该程序是 binutils 包的一部分 |
| ratfor | Ratfor 预处理程序可由 GCC 激活，但不是标准 GCC 发布版的一部分 |
| readelf | 从 ELF 格式的目标文件显示信息。该程序是 binutils 包的一部分 |
| size | 列出目标文件中每个部分的名字和尺寸。该程序是 binutils 包的一部分 |
| strings | 浏览所有类型的文件，析取出用于显示的字符串。该程序是 binutils 包的一部分 |
| strip | 从目标文件或文档库中去掉符号表，以及其他调试所需的信息。该程序是 binutils 包的一部分 |
| vcg | Ratfor 浏览器从文本文件中读取信息，并以图表形式显示它们。而 vcg 工具并不是 GCC 发布中的一部分，但-dv 选项可被用来产生 vcg 可以理解的优化数据的格式 |
| windres | Window 资源文件编译程序。该程序是 binutils 包的一部分 |

## 1.8　联系方式

GNU 的主页是 http://www.gnu.org，而 GCC 项目的主页是 http://gcc.gnu.org。

GCC 编译程序的范围很广——从简单的批处理工具程序到几百万行的规模的系统。总的来说，当软件项目变得更大或者在某些方面变得特殊，在不能处理某些奇怪问题的时候，就会出现各种情况。有些是 bug，有些是特殊习惯，但有的不可避免地需要澄清——或者至少能够在正确的方向上引起注意。所幸的是有帮助信息可供使用，而且可以找到关于 GCC 的一切信息。

信息的主要来源是邮件组。开放的邮件组（所有的成员都可以收发邮件）的好处是可以立即展开讨论。如果它有所帮助，我建议注册到 gcc-help 的邮件列表中。开放邮件组上的对话会继续到情况被澄清或问题得到解决。表 1-5 包括所有 GCC 开放邮件组的简要描述。只读邮件组列在表 1-6 中。

**表 1-5 GCC 的开放邮件组**

| 邮件组名 | 描述 |
|---|---|
| gcc | 这是开发 GCC 的基本讨论区。如果只是要注册一个邮件组，这就是合适的选择。它能够让你了解到最新的新闻和开发情况。该邮件组信件很多 |
| gcc-bugs | 讨论 bug 和报告 bug 的邮件组。这里的邮件也很多 |
| gcc-help | 该邮件组适用于那些寻找问题答案的人。该邮件组信件很多 |
| gcc-patches | 源代码补丁和关于补丁的讨论会被提交到这个邮件组。该邮件组信件很多 |
| gcc-testresults | 测试结果和对测试以及测试结果的讨论都会发到这里 |
| java | 关于 GCC 的 Java 前端的开发和维护的讨论列表，以及 Java 的运行时库的讨论列表 |
| java-patches | 关于 Java 前端和 Java 运行时库的源代码补丁被发布到该讨论组及 gcc-patches 邮件组 |
| libstdc++ | 该讨论组用来讨论标准 C++库的开发和维护 |

**表 1-6 GCC 的只读邮件组**

| 邮件组名 | 描述 |
|---|---|
| gccadmin | 该邮件组收到的消息来自 gcc.gnu.org 的 gccadmin 账号运行的长时间任务 |
| gcc-announce | 该邮件列表信息较少，主要用来公布最新的版本发布，以及其他关于 GCC 的重要事件 |
| gcc-cvs | 每个登入到 CVS 仓库的人都会发消息到该邮件组 |
| gcc-cvs-wwwdocs | 每个登入到 HTML 文档的 CVS 仓库的人都会发消息到该邮件组 |
| gcc-prs | 每次当报告的问题进入 GNATS 数据库的时候都会向该邮件组发消息 |
| gcc-regression | 发送到这个邮件组的消息包含的是 GCC 回归测试的运行结果 |
| java-announce | 这个邮件组信息很少，它是用来发布关于 Java 前端或者 Java 运行时基本函数的消息的 |
| java-cvs | 每次登入到 Java 编译程序和 CVS 仓库运行时部分的时候，都会向该邮件组（以及 gcc-cvs 邮件组）发送消息 |
| java-prs | 每次报告的与 Java 有关的问题进入 GNATS 数据库的时候，都会向该邮件组（以及 gcc-prs 邮件组）发送消息 |
| libstdc++-cvs | 每次登入到 CVS 仓库的 libstc++部分时，都会向该邮件组发送消息 |

所有的邮件组都可在网站 http://www.gnu.org/software/gcc/lists.html 中访问到。在该页面中可以订阅和退订这些邮件组。每个邮件组都有自己的网站，可用来查找并浏览该邮件组收到的消息。该邮件组的名字以 gcc.gnu.org/ml/开头是网站的名字。例如，要定位 gcc-announce 的文档网站，就要访问 http://gcc.gnu.org/ml/gcc-announce。

# 第 2 章

# 查询与安装编译程序

如果 GCC 的二进制可执行版本已经准备就绪，通常的安装步骤是下载源代码并编译它。编译 GCC 的过程非常稳定和成熟，因为它已经过多年的修正。同样的基本安装过程也可用于安装所有的 GNU 软件。简单来讲，步骤如下：

1. 下载源代码，保存到它自己的目录。
2. 创建独立的工作目录，用于编译源代码。
3. 从工作目录中执行 configure 脚本，会创建一个目录树，以及一些平台无关的文件集合，用来控制编译过程。
4. 键入命令 make 将源文件编译成目标文件。
5. 键入命令 make install 将最新编译的程序和库安装到你的系统中。

得到源代码的方法有两种：可用 FTP 得到压缩的 tar 文件，或用 CVS 得到独立的压缩文件。利用 FTP，可以得到稳定的编译程序发布版本。利用 CVS，可以获得已经发布的版本，以及目前的版本。FTP 格式是为用户设计的，而 CVS 格式是为 GCC 维护者设计的，但安装过程完全一样。

如果你的系统还没有安装其他的 GNU 软件，有必要首先安装 binutils。在 binutils 包中有很多 GCC 用到的工具，包括为 GCC 设计的直接使用的汇编器和连接程序。使用本地汇编器和连接程序未尝不可，但 GNU 的汇编器和连接程序就是设计用于和 GNU 编译程序一起工作的。这些工具的安装过程基本和 GCC 一样，但在没安装 binutils 的机器上安装 binutils 的过程很简单，而 GCC 的安装则需要 binutils。

就像几乎所有的 GNU 软件一样，编译程序由 C 语言编写，因此在编译新的编译程序之前，计算机上必须已经安装了 C 编译程序。如果要在没有安装 C 编译程序的机器上安装 GCC，必须将另外的安装了编译程序的计算机进行配置与编译，从而来交叉编译 GCC。第 16 章解释了其他机器的编译过程。

## 2.1　二进制代码下载

如果没有 C 编译程序，可以将源代码下载到某个安装了 C 编译程序的机器上，然后为你的目标机器交叉编译需要的版本；或者下载预编译的版本。GNU 并未提供编译程序的预编译版本，但有些地方有提供。GCC 有很多二进制版本，以供各种不同的计算机和操作系统使用，但表 2-1 只列出了其中的一些。

表 2-1　GCC 编译程序的预编译版本

| 平台 | 名字和位置 |
|---|---|
| AIX | Bull 的大型 AIX 自由软件和共享文档，位于 http://freeware.bull.net。南加州大学的 AIX 公共域软件库（Public Domain Software Library），位于 http://aixpdslib.seas.ucla.edu |
| DOS | DJGPP 位于 http://www.delorie.com/djgpp |
| HP-UX | 威斯康星大学的计算机辅助工程中心，位于 http://hpux.cea.wisc.edu。Utah 的 HP-UX 移植和文档中心，位于 http://hpux.cs.utah.edu。英国的 HP-UX 移植和文档中心，位于 http://hpux.connect.org/uk。SunSITE 欧洲中心，位于 ftp://sunsite.informatik.rwth-aachen.de/pub/packages/gcc_hpux |
| Solaris 2 | Solaris 自由软件项目（Intel 和 SPARC），位于 http://www.sunfreeware.com |
| SGI | SGI 自由软件，位于 http://freeware.sgi.com |
| UnixWare | 捣蛋软件（Skunkware），位于 ftp://ftp2.caldera.com/pub/skunkware/w7/Packages |
| Windows | Cygwin 项目，位于 http://sources.redhat.com/cygwin |

每个站点都有自己的下载和安装指导。GCC 编译程序是可移植的，但它的可移植性是为了能够在不同 UNIX 操作系统之间工作而设计的。编译程序的 DOS 版本很简单，只需要载入 DOS 机器上就可以运行了，但这只限于 C 和 C++编译程序。Cygwin 项目的 Windows 编译程序是完整的版本，不仅包括编译程序还包括一个工具集合，可以提供完整的 UNIX 工作环境。

## 2.2　FTP 资源下载

很多站点都提供对 GCC 源文件的匿名 FTP 访问。可以下载完整的编译程序集合，或选择下载需要安装的某种或某些语言。表 2-2 列出的文件不必全部下载。有两个选择：

- 可以选择只下载核心，然后选择下载其他需要包含的语言。
- 可以下载完整的编译程序，这与下载核心后再下载所有语言和测试包是一样的。

表 2-2 包含 GCC 源文件的 FTP 文件

| 文件名 | 内容 |
| --- | --- |
| gcc-3.1.tar.gz | 完整的编译程序，包括核心和所有组件 |
| gcc-ada-3.1.tar.gz | Ada 编译程序 |
| gcc-core-3.1.tar.gz | 核心，包括 C 编译程序和所有编译程序都通用的模块 |
| gcc-g++-3.1.tar.gz | C++编译程序 |
| gcc-g77-3.1.tar.gz | Fortran 编译程序 |
| gcc-java-3.1.tar.gz | Java 编译程序 |
| gcc-obj-3.1.tar.gz | Objective C 编译程序 |
| gcc-testsuite-3.1.tar.gz | 测试包 |

测试包是可选的。它是源代码程序的集合，可以用这些程序测试下载和编译的编译程序是否工作正常。

下面的步骤可用来下载源代码并安装，作好编译准备：

1. 选择一个 FTP 站点。GNU FTP 站点是 ftp.gnu.org/gnu，但还可能从全世界的几百个镜像站点下载。可从 http://www.gnu.org/order/ftp.html 中找到目前可用的镜像站点列表。为了使下载尽可能顺利，应该选择较近的镜像站点。
2. 将文件下载到工作目录。这可以是最后编译 GCC 的目录，但通常是临时目录，因为这些文件在解压缩以后可能就被删除了。从 FTP 下载的时候，选项一定要设置成二进制，而不是文本模式。因为有一些是压缩文件，而 FTP 的文本模式会错误解释内容，并将一些特定值转换成 ASCII 字符，这样就会破坏压缩文件。
3. 选择或创建用来保存源文件树的目录。当 GCC 解包的时候，会在当前目录创建自己的目录，这样就可以选择某个已经安装其他源目录的目录。例如，如果将源文件安装到目录/usr/local/src，从这里解包会将 GCC 源树安装到/usr/local/src/gcc-3.1。
4. 文件解包。如果你的 tar 工具支持 gzip 格式（具有 z 选项），可以如下解包：

```
cd /usr/local/src
tar -xvzf /tmp/download/gcc-core-3.1.0.tar.gz
```

5. 如果你的 tar 不支持 gzip，需要多执行一步，如下：

```
cd /usr/local/src
gunzip /tmp/download/gcc-core-3.1.0.tar.gz
tar -xvf /tmp/download/gcc-core-3.1.0.tar
```

这会创建包括源文件的目录/usr/local/src/gcc-3.1。如果选择下载多个文件，每个文件都要执行同样的命令。系统中不太可能没有 gunzip，实在没有的话可从 http://www.gzip.org 下载 gzip 的可执行版本。

## 2.3 CVS 资源下载

在某些方面上 CVS（Concurrent Versions System，并行版本系统）下载比 FTP 下载容易，而且更灵活，因为允许下载 GCC 的不同版本。GNU 软件的开发者使用 CVS 系统下载最新的版本，而且可以保持更新的过程。由于 CVS 是源代码文档，所以可以用它的工具获得编译程序的任意版本，包括当前的发布版本。

下载的 tar 源文件和 CVS 源文件的格式有一些微小差异。要对 CVS 源代码进行编译，需要安装 Bison 语法分析器，以及 Texinfo 版本 4.0 或更新版本，用来处理一些中间文件。tar 文件包括中间产生的某些文件，但 CVS 文件不包括它们。另一个区别是 CVS 下载中有一些选项，可以提供更多的诊断信息。

CVS 仓库可以跟踪源代码的每次变动。当要发布的时候，会创建一个标记来标志该发布版。该标记与库中的每个源模块当前版本或选中版本相关。如果需要下载编译程序的某个版本，需要指定标记名为本地的 cvs 工具，这会保存所有的源文件。所有源文件都按照压缩格式下载(如果正确指定命令行选项)，而在下载完成后再被解压缩保存在正确目录中。结果就可以得到和 FTP 下载同样的目录和文件集合。

下面的步骤描述了下载指定版本 GCC 的过程：

1. 通过键入如下命令检查系统是否安装了 cvs 工具：

   ```
   cvs -v
   ```

   它应该显示版本号以及其他信息。如果没有 cvs，或者如果版本比 1.10.4 要早，则需要下载最新的版本，可以从 http://www.cvshome.org 得到。

2. 指定远端 CVS 库的名字。最简单的方法就是用下面的命令定义一个环境变量名：

   ```
   CVSROOT=:pserver:anoncvs@subversions.gnu.org:/cvsroot/gcc
   export CVSROOT
   ```

   这是 CVS 库的位置。如果命令中没有指明-d 选项，cvs 工具就会查找环境变量。如果愿意，也可以在 cvs 命令中使用-d 选项来指定地址，但它必须是命令行中的第一个选项，如下：

   ```
   cvs -d :pserver:anoncvs@subversions.gnu.org:/cvsroot/gcc
   ```

3. 登录 CVS。在设置 CVSROOT 环境变量后，可以使用下面的命令直接进入库：

   ```
   cvs login
   ```

   提示要求输入密码，因此要以只读匿名方式进入，只要按回车键就可以了。如果登录成功结束，命令行提示会消失，可以键入下一个 cvs 命令。

4. 下载源文件。将父目录变成希望包含 GCC 源码树的目录。键入下面命令可以下载所有版本的源代码，并将它们保存在 gcc 目录中：

```
cvs -z 9 checkout -r gcc_3_1_0_release gcc
```

-z 9 选项很重要，因为它要求 cvs 压缩文件，这会缩短下载文件所需的时间。无论是否压缩文件，最后的结果都是一样的，因为 cvs 最后会将它展开保存到本地目录中。

5. 使用和前面一样的标记，可以获得和某个指定版本编译程序匹配的相关文档。它是 HTML 文件，保存在目录 wwwdocs 中。下载文档的命令和下载源文件的命令非常相似：

```
cvs -z 9 checkout -r gcc_3_1_0_release wwwdocs
```

### 2.3.1 以前的发布版

通常是从 CVS 下载中得到编译程序的最新发布，但如果需要不同的版本可以参考下面这些发布标记。下列标记名字可用来获得更早的发布。

| | | |
|---|---|---|
| gcc_3_0_3_release | gcc_2_95_2-release | egcs_1_1_release |
| gcc_3_0_2_release | gcc_2_95_1-release | egcs_1_0_3_release |
| gcc_3_0_1_release | gcc_2_95-release | egcs_1_0_2_release |
| gcc_3_0_release | egcs_1_1_2_release | egcs_1_0_1_release |
| gcc_2_95_3 | egcs_1_1_1_release | egcs_1_0_release |

### 2.3.2 测试版

如果没有指定标记名，可以得到最新测试版本 GCC 的快照。下面的命令会下载测试版：

```
cvs -z 9 checkout gcc
```

这种方法下载的版本是编译程序的最新测试版，因此它不一定工作无误。事实上，也不能确保一定能够编译成功。

一旦得到最新版本的所有文件，可以通过 cvs 保持它，在需要的时候更新文件。cvs 命令会比较库中文件的版本和本地磁盘中的文件版本，只下载需要的文件。更新命令如下：

```
cvs -z 9 update
```

由于该命令会更新你的本地目录，它会列出处理的每个文件名，通过一个字母指出执行的是什么操作。字母 P 指出已有版本是最新版本。字母 U 指出已经用最新版本替换了以前的版本。如果本地目录有个文件和库中文件不相匹配，就用问号进行标记。

由于 GCC 软件还在开发，文档也就不断更新以反映这些变化。可以用类似方法下载匹配的最新版本文档：

```
cvs -z 9 checkout wwwdocs
```

如果选择下载文档，就会在 cvs 更新源文件的时候更新文档。

## 2.4　编译和安装 GCC

GCC 的安装过程在过去这些年里已经在很多不同平台上执行过很多次了，因此它已经很成熟和稳定了。如果要在同一台机器上下载和安装 GCC，过程非常简单。但如果需要某些特殊操作，则有大量选项可供使用。

### 2.4.1　安装过程

下面列出了安装 GCC 的主要步骤：

1. 确保目前的 C 编译程序可用。要求路径中含有 cc 或 gcc，而且 CC 环境变量也要求设置为编译程序的名字。通过命令行键入 cc 或 gcc 就可以验证是否存在编译程序。
2. 验证已经安装了 GNU make。其他版本的 make 可能可以正确工作，但更有可能会带来一些问题。如果选择使用其他版本的 make 发现一些奇怪的错误消息，应该安装 GNU make 然后再次尝试。要验证是否已经安装了 GNU make，可以键入下面的命令要求 GNU make 显示自己的版本信息：

   ```
   make -v
   ```

3. 创建一个配置目录。该目录应该是一个目录树的根，用来包含所有的 make 程序的描述文件以及产生的目标文件。强烈推荐不要在包含源文件的目录中编译 GCC。
4. 选择希望在 configure 脚本中使用的选项。有很多选项可供选择，它们都在 2.4.2 节中有所描述。每个选项都有默认值，所以只需要为某些特殊情况指出特定的值。

   最通用的选项是--prefix，用来指定安装 GCC 二进制代码的根目录名字。安装完成之后，指定前缀的目录包含所有 GCC 可执行的代码，而子目录 bin、include、info、lib、man 和 share 中含有其他文件。默认的前缀为/usr/local。

   configure 脚本最有趣的是它有准确猜出所运行的操作系统和硬件平台的能力，而且几乎不会出错。这是通过调用脚本 config.guess 来实现的。如果需要，也可以用命令行执行该命令，就可以看到你的系统的准确信息。
5. 从内部工作目录中执行 configure 脚本。因为正在从其他目录执行脚本，因此有必要指定完全路径名。例如，如果要将源文件下载到/opt/gnu/gcc，而目标目录为/opt/build，并且要将最终可执行文件和库保存在/opt/usr/local，可以运行下面的 configure 脚本：

   ```
   cd /opt/build
   ```

```
/opt/gnu/gcc/configure --prefix=/opt/usr/local
```

6. 编译 GCC。如果 configure 脚本运行成功，目标目录就会包含很多文件和目录，包括一个叫做 Makefile 的文件。要编译所有内容，只要执行命令：

```
make
```

在编译过程中，你会看到显示的一些出错消息，但只要 make 忽略这些错误就是正常的，编译程序会继续操作下一个文件。有些错误和警告消息是可预知的——只有中断处理过程的错误才是需要考虑的。

7. 测试编译程序。运行测试包是一种可选方案，可能需要下载很多额外的软件来实现。如果决定运行测试包，可在 2.7 节“运行测试包”中找到相关步骤。
8. 安装编译程序。所有内容都编译好以后，可以用下面命令安装 GCC：

```
make install
```

9. 设置路径。为了可以直接使用编译程序，有必要将包含可执行代码的目录加到 PATH 环境变量中。除非通过指定 configure 脚本中的目录名字选项对它作过一些修改，一般 PATH 变量应该已经正确设置了。
10. 如果要创建交叉编译程序（就是说，在一种系统上运行的编译程序编译出可在其他系统上运行的代码），首先需要创建本地编译程序，然后按照第 16 章介绍的步骤创建交叉编译程序。
11. 如果要创建 Ada 编译程序——这里没有指出它的完整创建步骤——需要按照第 9 章介绍的步骤完成。

### 2.4.2 配置选项

安装选项由 configure 脚本的命令行指定。该脚本产生的文件可以控制编译和安装。每个选项都有默认值，这些默认值可以正确创建本地机器上的编译程序（或编译程序集合），但有些环境需要调整。下面是对选项的描述：

- enable 和 disable。选项的名字能够以--enable 开始，也能够以--disable 开始。二者之中必有一种是默认的，它会根据平台的不同而有所变化。下面是按字母排序的选项列表，都以--enable 开始。
- with 和 without。选项的名字能够以--with 开始，也能够以--without 开始。二者之中必有一种是默认的，它会根据平台的不同而有所变化。下面是按字母排序的选项列表，都以--with 开始。
- 语言。默认情况下，configure 脚本会准备编译下载的所有语言，但可以通过--enable-languages 选项指定实际编译的语言。

- 预先指定目录名。被安装到目录集合中的部分编译程序名字相对标准，但也可以指定想要的任何名字。当选择改动目录名字的时候，除了--prefix，很少使用其他选项，因为预先指定的目录是安装所有目录的根。注意，不推荐源文件和目标文件选择同样的目录树，因为这可能会产生冲突引发问题。
- 文件名。可以对组成编译程序的文件名进行改动。这在开发自己的实验性编译程序或要安装 GCC 的多个版本的时候非常有用。
- 库。GCC 中有一部分是库，包含不同语言使用的运行时函数。不论是共享库还是静态库，都是 GCC 安装中的一部分。有些库是必须的，有些库是可选的。
- 汇编器和连接程序。选项的集合可用于指出汇编器和连接程序的名字和位置。如果没有使用这些选项指定位置，配置过程会用下面两个步骤进行定位：

  1. configure 脚本会查找 exec-prefix/lib/gcc-lib/target/version 目录，这里的 exec-prefix 默认为/usr/local，除非在设置目录名的时候，已经通过--prefix 或--exec-prefix 指定了其他名字。target 是目标系统的名字，而 version 是 GCC 的版本号。
  2. configure 脚本会查找跟操作系统相关的目录（例如 Solaris 中的/usr/ccs/bin 和 Linux 中的/usr/bin）。

- 代码生成。有两类代码生成选项。一类用来指出作为编译程序的一部分的目标代码类型，还有一类用来指出由其他编译程序产生的代码的种类。
- 平台。平台也被称为目标或主机。有些选项应用于特定硬件上运行的特定操作系统。尽管 config.guess 脚本总能猜出使用的平台，还是有一些硬件选项是不能够预测的。有些系统看起来是一样的，但实际上有细微的差别。

--bindir=directory

目录默认为 exec-prefix/bin。这是包含可执行代码的目录。而 PATH 环境变量通常包含该目录名，因此编译程序的名字才有可能在命令行中直接运行。

--build=host

产生指定主机上运行的配置。默认和--host 选项设置的平台相同，而后者的默认值为 config.guess 脚本的输出。

--cache-file=filename

configure 脚本运行各种测试来确定本地机器的配置和性能。该文件名会包含这些测试的结果。

--datadir=directory

默认为 prefix/share。这是包含数据文件的目录名，例如现场信息。

--enable-altivec

指出目标平台是支持 AltiVec 向量扩展的 PowerPC。该选项在需要时会产生 AltiVec 代码。

--enable-checking=check[,check,...]

该选项会产生执行一些编译程序内部检查的代码。该检查会产生诊断输出并会增加编译时间，但它对编译程序的输出结果没有其他作用。在编译从 CVS 下载的文件时，该选项默认是设置的，但如果编译发布版本，就没有设置该选项。

通过选择 misc、tree、gc、rtl 和 gcac，可以指出想要检查的列表。如果忽略列表，只是指出--enable-checking，该列表默认只包含 misc、tree 和 gc。因为检查 rtl 和 gcac 的代价是巨大的。

--enable-cpp

指出用户可获得的安装的 cpp（C 预处理程序）的版本。通常使用默认值。参见--with-cpp-install-dir。

--enable-languages=language[,language,...]

指出只编译指定语言。可用的语言名为 ada、c、c++、f77、java、objc 和 CHILL。如果不指定该选项，会编译所有语言。编译 Ada 需要一些特殊步骤，第 9 章会有所描述。而且不再支持 CHILL 语言，除非使用老版本 GCC，否则不会正确编译。

--enable-libgcj

指出创建 Java 时使用的运行时库，它是默认的。指定--disable-libgcj 也可创建 Java 编译程序，但会使用其他资源的运行时库。

--enable-maintainer-mode

指出从源代码重新生成的 gcc.pot 文件。该文件是主消息分类，包括编译程序产生的所有错误和警告诊断消息。该文件用来实现国际化，第 11 章有所描述。

**注意** 要令该选项工作正常，需要完整的源代码树和 gettext 的最新版本。

--enable-multilib

在大多数系统上它都是默认设置。该选项指出构建的目标系统上的多个库。创建这些库通常是为了支持不同的目标变量、浮点运算、函数调用规则，等等。它不会禁止生成所有库，对于表 2-3 列出的平台，可以通过名字禁止某个库。例如，对于平台 arc-*-elf*，可用选项--disable-biendian 禁止生成该库。

表 2-3　平台禁止的库

| 平台 | 库名 |
| --- | --- |
| arc-*-elf* | Biendian |
| arm-*-* | fpu,26bit,underscore,interwork,biendian,nofmult |
| m68*-*-* | softfloat,m68881,m68000,m68020 |
| mips*-*-* | single-float,biendian,softfloat |
| powerpc*-*-* | aix64, pthread, softfolat, powercpu, powerpccpu, powerpcos, biendian, sysv, aix |
| rs6000*-*-* | aix64, pthread, softfolat, powercpu, powerpccpu, powerpcos, biendian, sysv, aix |

--enable-nls

指明本地语言支持（Native Language Support，NLS）是编译程序的一部分，它允许按照非英语的本地语言显示警告和出错消息。参见--with-included-gettext 和--with-catgets。

--enable-shared

这是默认设置。指出--disable-shared 会只创建静态库。

--enable-shared[=package[,package ...]]

对于 GCC 的 2.95 版以及之前的版本，该选项对创建共享库而言是必需的。之后的版本默认会为所有支持共享库的平台创建共享库。

指明一个安装包的列表，会只为这些包创建共享库。这些可识别的包为 libgcc、libstdc++、libffi、zlib、boehm-gc 和 libjava。

--enable-target-optspace

指出要求在大小上进行优化的库，而不是在速度上进行优化。

--enable-threads

对于某些平台，这是默认设置。它指出目标支持线程。这对 Objective C 的库和 C++及 Java 的异常处理有影响。如果目标系统没有线程，或者编译程序自己不能为目标系统产生多线程的代码，那么选项--disable-threads 就和--enable-threads=single 等同。

--enable-threads=library

指明的库是支持线程的库。表 2-4 列出了可能的库名。

表 2-4　选择支持线程的名字

| 库 | 描述 |
| --- | --- |
| aix | AIX 线程支持 |
| dce | DCE 线程支持 |

（续表）

| 库 | 描述 |
| --- | --- |
| mach | 一般 MACH 线程支持。该选项要求提供 gthr-mach.h 头文件的复本 |
| no | 同 single |
| posix | 标准 POSIX 线程支持 |
| rtems | RTEMS 线程支持 |
| single | 禁用线程支持 |
| solaris | Sun Solaris 2 线程支持 |
| vxworks | VxWorks 线程支持 |
| win32 | Microsoft Win32 线程支持 |

--enable-version-specific-runtime-libs

指出安装某种特定运行时库的头文件的目录，该目录以目标和版本命名，而不在通常的位置。该库被安装在 libdir/gcc-lib/target/version。除非用--with-gxx-include-dir 选项指定位置，否则 libstdc++的包含文件安装在目录 libdir/gcc-lib/target/version/include/g++。

--enable-win32-registry

指定 GCC 的 Win32 版本，它不被注册表用于定位编译程序的安装路径，编译程序的库是通过使用下面的注册表键来定位的：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Free Software Foundation\key
```

键值默认为 GCC 的版本号，可通过选项--enable-win32-registry 设置键值。

--enable-win32-registry=key

指定 GCC 的 Win32 版本，用于在使用下面注册键的 Windows 注册表中定位安装路径：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Free Software Foundation\key
```

如果没有指定该选项，默认使用 GCC 版本号作为键。以这种方式使用注册表使得可以在不同位置安装和使用 GCC 的不同版本。

--exec-prefix=directory

默认为 prefix。这是保存所有结构无关文件的最上层目录的名字。

--help

该选项会打印命令行选项的列表并会终止 configure 脚本，但不做其他事情。该选项列表按照分类进行组织。

--host=host

主机名。默认是脚本 config.guess 的输出。GCC 可在很多型号的主机上运行。

--includedir=directory

默认为 prefix/include。为包含 C 头文件的目录名。

--infodir=directory

默认为 prefix/info。这是保存 info 格式文档的目录名。

--libdir=directory

默认为 exec-prefix/lib。这是包含静态库和 GCC 内其他部分的目录名。

--libexecdir=directory

默认为 exc-prefix/libexec。这是包含和库相关的可执行程序的目录名。

--localstatedir=directory

默认为 prefix/etc。这是包含对某个机器特定的可修改数据的目录名。参见 sysconfdir。

--mandir=directory

默认为 prefix/man。这是包含手册的目录名。

--nfp

指出该机器没有浮点运算单元。该选项只用于 m68k-sun-sunos* 和 m68k-isi-bsd。

--no-create

可以运行配置脚本，但不会创建编译代码必需的输出文件。

--norecursion

该源代码树为每个目录包含独立的 configure 脚本。除非指定该选项，否则执行一个 configure 脚本也会执行子目录中的所有 configure 脚本。

--prefix=directory

默认是/usr/local。这是 GCC 完全安装用到的最上层目录。默认是将所有其他目录放在 prefix 目录里，命名为 bin、include、info、lib、man 和 share。除非使用其他命名选项进行修改，否则指明 prefix 名就会为每个目录指定路径名。

--program-prefix=prefix

默认不使用前缀。前缀名放在所有 bin 目录中的文件名的前面。例如，要将安装的 Java 编译程序名字从 gcj 改成 stim-gcj，需要使用下面的选项：

```
--program-prefix=stim-
```

--program-suffix=suffix

默认不使用后缀。后缀名被加到 bin 目录中所有文件名的后面。例如，要将安装的 Java 编译程序名字从 gcj 改成 gcj-v4，需要使用下面的选项：

```
--program-suffix=-v4
```

--program-transform-name=pattern

模式是 sed 脚本，应用于 bin 目录中的文件名。使用 sed 脚本这种方式使得有可能修改每个可执行文件的名字。例如，要将 Java 编译程序的名字从 gcj 改成 gjava，要将 g++的名字改为 gcplus，而保留其他所有名字，需要使用下面的选项：

```
--program-transform-name='s/^gcj$/gjava/; s/^g++$/gcplus/'
```

该选项可与前缀和后缀选项一同使用，因为--program-prefix 和--program-suffix 总是在使用模式脚本之前应用于这些名字。在创建交叉编译程序的时候不能使用该选项。

--sbindir=directory

默认为 exec-prefix/sbin。这是包含系统可执行程序的目录名。

--silent

该选项禁止 configure 脚本的输出，通常会列出进行的所有测试。该选项等同于--quiet。

--srcdir=directory

指出的目录用来包含文件 configure.in，它会提供 configure 使用的关于源文件名字和位置的信息。

--sysconfdir=directory

默认为 prefix/etc。这是包含对某个机器而言是只读数据的目录名。参见 localstatedir。

--target=host

目标机器（运行编译程序的机器），默认为脚本 config.guess 的输出结果。

--tmpdir=directory

指出 configure 脚本使用的目录名，用来保存临时工作文件。

--version

打印创建 configure 脚本的 autoconf 工具的版本号，不执行其他操作。

--with-as=pathname

指出汇编器的完整路径名。如果通过 configure 脚本的默认查找过程不能找到汇编器，就会需要该选项，或者当系统中存在多个汇编器时，也需要指定使用其中的哪一个。

--with-catgets

如果通过--enable-nls 启用了 NLS，但主机没有安装 settext，编译程序就需要使用主机的 catgets。

--with-cpp-install-dir=directory

指明 cpp（C 预处理程序）的复本被安装到 prefix/directory/cpp 目录，而且要在--bindir 选项指定位置安装 cpp 的目录（该选项默认目录为 exec-prefix/bin）。参见--disable-cpp。

--with-cpu=cpu

指出目标平台的 CPU。如果为平台选择一种特定的 CPU，GCC 就有可能生成比相对于一族处理器更好的代码。表 2-5 列出了 GCC 版本可识别的 CPU 名字。新的 CPU 名字会不断增加，因此如果在表中没有发现需要的 CPU，就请查看 config.gcc。

表 2-5　可指出的 CPU 名

| 平台 | CPU 名 |
|---|---|
| arm*-*-* | xarm2, xarm3, xarm6, xarm7, xarm8, xarm9, xarm250, xarm600, xarm610, xarm700, xarm710, xarm7m, xarm7dm, xarm7dmi, xarm7tdmi, xarm9tdmi, xarm7100, xarm7500, xarm7500fe, xarm810, xxscale, xstrongarm, xstrongarm110, xstrongarm1100 |
| powerpc*-*-* | xcommon, xpower, xpower2, xpower3, xpowerpc, xpowerpc64, xrios, xrios1, xrios2, xrsc, xrsc1, xrs64a, x401, x403, x405, x505, x601, x602, x603, x604, x605e, x620, x630, x740, x750, xx801, x821, x823, x8607400, x7450, xec603e |
| sparc*-*-* | supersparc, hypersparc, ultrasparc, v7, v8, v9 |

--with-dwarf2

指出编译程序产生的调试信息，默认为 DWARF2 格式。

--with-gnu-as

指出无论找到哪种汇编器，都假设它是 GNU 汇编器。在系统敏感的情况下，如果指明该选项而实际的汇编器不是 GNU 汇编器时，可能会出问题。如果指定该选项且汇编器也是 GNU 汇编器，也可能会出问题。下面列出的平台就会有这种问题：

hppa1.0-*-*
hppa1.1-*-*
i386-*-sysv
i386-*-isc
i860-*-bsd
m68k-bull-sysv
m68k-hp-hpux
m68k-sony-bsd
m68k-altos-sysv
m68000-hp-hpux
m68000-att-sysv
*-lynx-lynxos
mips-*

如果系统中有多个汇编器，应该用选项--with-as 指明使用哪个汇编器。如果在下面的系统中使用 GNU 汇编器，也必须使用 GNU 连接程序（用选项--with-ld 指明）。

| | |
|---|---|
| i386-*-sysv | m68k-altos-sysv |
| i860-*-bsd | m68000-hp-hpux |
| m68k-bull-sysv | m68000-att-sysv |
| m68k-hp-hpux | *-lynx-lynxos |
| m68k-sony-bsd | mips-* (except mips-sgi-irix5-*) |

--with-gnu-ld

和选项--with-gnu-as 相同，只不过它应用于连接程序。

--with-gxx-include-dir=directory

默认为 prefix/include/g++-v3。这是 g++头文件的目录名。参见--enable-version-specific -runtime-libs。

--with-headers=directory

指出的目录包含创建交叉编译程序的目标的头文件。如果目录 prefix/target/sys-include 不存在，则该选项是必须的。头文件必须复制到 GCC 安装目录并被修改，这样才可以兼容。参见--with-newlib 和--with-libs。

--with-included-gettext

如果 NLS 通过--enable-nls 被激活，该选项指出构建进程在使用系统安装的版本以前，要尝试使用自己的 gettext 备份。

--with-ld=pathname

与选项--with-as 相同，只不过它应用于连接程序。

--with-libs="directory [directory ...]"

该选项适用于创建交叉编译程序。指定目录中的库会被复制到 GCC 安装目录中。参见--with-headers 和--with-newlib。

--with-local-prefix=directory

默认为/usr/local。这是包含目录的前缀，编译程序用来查找安装在本地的包含文件。只有在系统已经建立了使用目录的某些规则，而不再在/usr/local/include 中查找本地安装的头文件的时候，该选项才是必须的。该选项不能设置为/usr，因为安装的头文件会和系统头文件混合，而这些冲突会造成无法编译某些程序。

指定--prefix 选项不会对该选项产生作用。--prefix 选项指出在哪里安装 GCC，而该选项指出在运行的时候在哪里查找头文件。

--with-newlib

该选项用于构建交叉编译程序。库 newlib 用于目标机器的 C 库。函数__eprintf 未包含在 libgcc.a 中的原因是假设 newlib 会提供它。参见--with-headers 和--with-libs。

--with-slibdir=directory

默认是 libdir。这是包含共享库的目录名。

--with-stabs

指出由编译程序产生的调试信息默认为 stabs 格式，而不是宿主系统的标准格式。通常 GCC 默认产生的调试信息是 ECOFF 格式，但是用该标志会将默认值改为 BSD 风格的 stabs。该选项设置默认会被写入编译程序中，可用编译程序的命令行选项-gcoff 或-gstabs 修改。

对 C 以外的语言而言，ECOFF 格式包含的调试信息不够。调试信息的 stabs 格式携带更多信息，但通常需要使用 gdb 调试器。

--with-system-zlib

指出编译程序应该使用已经安装的 zlib，而不是再创建一个新的。该选项只应用于 Java。

--with-x

指出要使用 X Windows System。

--x-includes=directory

含有 X 包含文件的目录名。

--x-libraries=directory

包含 X 库的目录名。

## 2.5 binutils

尽管 GCC 有可能使用本地编译程序和连接程序，但编译程序在与 GNU 汇编器、连接程序和其他工具的一同使用时工作得最好，兼容性最高。在表 1-4 中，我们简单描述了所有 binutils 工具和其他 GCC 工具。下面列出 binutils 包中的工具的名称：

| | | | |
|---|---|---|---|
| addr2line | grpof | objcopy | size |
| ar | ld | objdump | strings |
| as | nlmconv | ranlib | strip |
| c++filt | nm | readelf | windres |

有些工具会读写目标文件中的信息。这是通过二进制文件描述字（Binary File Descriptor，BFD）库提供的工具实现的，它也是作为 binutils 源代码的一部分被提供的。

BFD 库提供了一个函数集合，它知道目标代码的很多不同格式，可被调用以操纵它们。这使得这些工具可以被编译以在不同平台上执行相同操作。

下面的步骤用于下载并安装源代码，为最后的编译做好准备：

1. 选择一个 FTP 站点。GNU 的 FTP 站点为 ftp.gnu.org/gnu，但也可以从遍布全世界的几百个镜像站点下载。可从 http://www.gnu.org/order/ftp.html 找到当前的镜像站点列表。为使下载顺利，应该选择较近的镜像站点。
2. 将文件 binutils-2.9.tar.gz 下载到工作目录。版本号可能不同，因为这个包在持续不断地修改和更新。下载时将 FTP 选项设置为二进制而不是文本模式。这是一些压缩文件的集合，而 FTP 的文本模式通过错误解释内容会将特定值转换成 ASCII 字符，这会破坏压缩文件。
3. 选择希望用于配置脚本的选项。可用的选项基本和 GCC 脚本的一样。就像 GCC 脚本那样，binutils 配置脚本无需选项也可以正常运行，但最简单的是使用--prefix 选项指定目录名字，可用来保存安装的二进制工具。名为 prefix 的目录会包含子目录 bin、include、info、man 和 share。如果没有指定目录，默认目录为/usr/local。
4. 在工作目录内执行配置脚本。因为正在从其他目录执行脚本，所以有必要指出完全路径名。例如，如果将源代码树下载到/opt/gnu/binutils，目标目录是/opt/bubuild，而最后要将可执行文件和库保存到/opt/usr/local，可以如下执行配置脚本：

```
cd /opt/bubuild
/opt/gnu/binutils/configure --prefix=/opt/usr/local
```

5. 如果还没有执行上述步骤，需要将最新的 bin 目录加到 PATH 环境变量中，这样才能定位工具。

作为 FTP 的可选工具，可以使用 CVS 从 binutils 的当前工作版本得到复本。这通常用于程序员试图修改源代码的时候，但这也是惟一的方法可以得到最新的开发内容。CVS 访问过程和前面描述的 GCC 一样。首先，设置 CVSROOT 如下：

```
CVSROOT=:pserver:anoncvs@sources.redhat.com:/cvs/src
export CVSROOT
```

然后使用下面命令登录，并用 anoncvs 作为密码进行响应：

```
cvs login
```

下面命令会下载整个源代码树：

```
cvs -z 9 checkout binutils
```

只要把所有事情都完成了，就可在任何时间登录并键入如下命令来获得最新的更新内容：

```
cvs -z 9 update
```

## 2.6 Win32 二进制安装

如果希望在 Windows 操作系统中使用 GCC，可以使用已经编译并准备好运行的版本。可在下面的站点找到关于 Cygwin 编译程序的更多信息：

```
http://cygwin.com
```

### 2.6.1 Cygwin

GNU 软件之所以能够在 Windows 操作系统上运行，是因为一个叫做 cygwin1.dll 的共享库，它包括一个模拟 UNIX 环境的 API。它可在 Windows 95 及其之后的所有 Windows 版本上运行（除了 Windows CE）。使用这些工具，就可以编写控制台程序，也可编写 Win32 GUI 应用。编写 GUI 应用需要使用 Win32 的 API，但命令行应用可基于独立的 Cygwin 库编写。

尽管这是自由软件，但 Cygwin 的许可却是混合型的。其中一部分属于 GNU 许可证，一部分是 X11 许可证，还有一些属于公共域。其中没有共享软件，因此非商业用途是无需付费的，但如果要用于商业产品（就是说你卖的产品依赖于这些库），要意识到有些是需要版权的。通过发信查询 sales@cygwin.com，可以了解如何得到商业版权。

Windows 上运行两种程序：控制台类型（就是由命令行运行，不显示窗口）和 GUI 类型（可以从控制台程序启动，但设计为窗口程序）。编译这两种程序的过程区别不大。

下面的命令会编译和连接控制台程序：

```
gcc helloworld.c -o helloworld.exe
```

可以使用 GCC 编译程序、Windows API 以及相应的 Cygwin 工具一起来创建 Windows 程序和 DLL。第 16 章会对这一过程进行描述。

### 2.6.2 安装

一个特殊的安装程序 setup.exe 不仅可被用于最初的下载和安装，而且也可以用于下载和安装更新内容，成为一个新的版本。下载工具存在的主要原因之一就是包已经变得非常大了，而且不是每个人都需要所有的功能。setup.exe 程序管理下载过程，可以选择下载指定部分，并且可以指定如何安装软件。

以下是安装过程中各个步骤的基本描述。大部分过程是自动的，而且只要一开始，就会得到提示要求输入：

1. 创建安装目录。Cygwin 有很多地方类似 GCC，因此应该将目录命名为类似 c:\cygwin，这是默认命名。安装过程创建大量的目录（例如 bin 和 etc），所以实际

上是在创建这些目录的根目录。

2. 下载 setup.exe 到新目录中。从站点 http://cygwin.com/download.html 可以找到最新信息，或者使用链接 http://cygwin.com/setup.exe，它会提示下载的地址。在同一个页面中也会看到其他站点可提供下载，根据你的位置取其方便。
3. 执行 setup.exe 程序，会给出从因特网安装、下载、保存软件的选项。如果已经下载了软件，还会提供选项提示如何从目录安装软件。可以选择直接从因特网安装软件，或先下载软件以后再安装软件。
4. 选择镜像站点。应该选择镜像站点列表中靠近自己的站点。你的选择可能会由于下载站点太忙而被拒绝。如果这样就重新选择一个。
5. 选择下载内容。你会看到工具分类列表。Cygwin 包含的所有程序都已经编译好准备运行了。可以从很多分类中选择需要的工具，但选择 Devel 类别会列出一个软件开发工具列表，包括 GCC。默认情况下，大部分包都被标记为跳过（Skip），就是说不会下载这些包。如果想下载程序的二进制版本，只要连到你需要的版本号就可以了。如果选中某个版本，就会出现一个选择框，要想下载源代码可以选中该框。
6. 如果已经选择从因特网上直接安装软件，到此就已经完成了。如果选择从因特网上下载软件，就需要再运行 setup 程序，要求它使用下载目录中的文件进行安装。

## 2.7 运行测试包

在最终安装最新编译的 GCC 版本之前，可以运行一个测试集合来验证它是否工作正常。这是可选的步骤，因为总的来说，如果能够编译出 GCC，它就应该能够正确运行。这些测试主要是用来确保解决的问题或新加入的功能不会带来其他问题或去掉已有的功能。

可以使用一些简单的步骤在你的系统上运行测试：

1. 如果以前没有做过，将测试包作为 GCC 源代码的剩余部分下载和安装在同一目录中。可以检查目录 gcc/testsuite 是否存在来验证已经下载了测试包。
2. 安装最新版本的 DejaGnu。因为较早的版本（1.3 或更早的版本）不工作。
3. 设置环境变量。如果安装的 DejaGnu 将运行测试的目录已经包含在 PATH 设置中，就不再需要设置这些变量了。如果没有，假设 DejaGnu 已经被安装到/usr/local 上，那么需要设置下面两个环境变量：

```
TCL_LIBRARY=/usr/local/tcl8.0
DEJAGNULIBS=/usr/local/dejagnu
```

4. 运行测试。改变到编译 GCC 的同一目录，然后就可进行任意的测试。如果需要运行完整的测试包（这会持续很长时间），键入如下命令：

```
make -k check
```

-k 选项告诉 make 命令忽略失败条件，继续进行下一条测试。如果要求只测试编译程序的 C 前端，键入如下命令：

```
make -k check-gcc
```

要只测试 C++，键入如下命令：

```
make -k check-g++
```

5. 检查测试结果。运行测试之后，会发现在测试包子目录中创建了一些新文件。这些文件后缀为.log，包括每个测试所采取的详细动作。后缀为.sum 的文件包括测试结果的总结，每个测试结果都由某个结果代码表示，见表 2-6。

**表 2-6　测试结果总结代码**

| 结果 | 描述 |
|---|---|
| PASS | 希望测试通过，且通过 |
| XPASS | 希望测试不通过，但通过 |
| FAIL | 希望测试通过，但不通过 |
| XFAIL | 希望测试不通过，并且不通过 |
| UNSUPPORTED | 该平台不支持此测试 |
| ERROR | 运行测试时发现问题 |
| WARNING | 运行测试时发现可能的问题 |

The Complete Reference

# 第 2 部分

# 使用编译程序集合

# 第 3 章

# 预处理程序

预处理程序的概念来源于 C 编程语言的一部分。预处理程序读出源代码，对其中内嵌的指示字进行响应，产生源代码的修改版本，修改后的版本会被编译程序读入。预处理程序仍然是 C、C++和 Objective-C 中重要的一部分，但是也可用它（有限地）预处理其他的语言源代码。例如，可用它实现 Fortran 和 Java 的条件编译。

在 GNU 术语中，预处理程序叫做 CPP。而 GNU 的可执行程序叫做 cpp。

## 3.1 指示字

源代码中的预处理指令叫做指示字（directive），从源代码中可以轻易发现，它们以井号（#）开始，在每行都是第一个非空字符。而井号通常都在第一列，后面紧跟着指示字的关键字。所有指示字及其描述都列在表 3-1 中。如果指示字指令要求，预处理程序是可以修改源代码的，而不是修改指示字。

表 3-1　GNU 预处理程序能够理解的指示字

| 指示字 | 描述 |
|---|---|
| #define | 定义宏名字，预处理程序会把这个宏扩展到使用该名字的位置 |
| #elif | 由#if 指示字提供一个用于计算的可选表达式 |
| #else | 如果#if、#ifdef 或#ifndef 为假，提供一个用于编译的可选代码集合 |
| #error | 产生出错消息，挂起预处理程序 |
| #if | 如果计算算术表达式的结果为非零值，就编译指示字和它匹配的#endif 之间的代码 |
| #ifdef | 如果已经定义了指定的宏，就编译指示字和它匹配的#endif 之间的代码 |
| #ifndef | 如果没有定义指定的宏，就编译指示字和它匹配的#endif 之间的代码 |
| #include | 查找指示字列表，直到找到指定的文件，然后将文件内容插入，就好像在文本编辑器中插入一样 |

（续表）

| 指示字 | 描述 |
| --- | --- |
| #include_next | 和#include 一样，但该指示字从找到当前文件的目录之后的目录开始查找 |
| #line | 指出行号以及可能的文件名，报告给编译程序，用于创建目标文件中的调试信息 |
| #pragma | 提供额外信息的标准方法，可用来指出一个编译程序或一个平台 |
| #undef | 删除前面用#define 指示字创建的定义 |
| #warning | 由预处理程序创建一个警告消息 |
| ## | 连接操作符，可用于宏内将两个字符串连接成一个 |

### 3.1.1 #define

#define 指示字定义宏。宏对应一个名字，无论何时在文本中发现这个名字，就会用作为宏的值的字符串去替换该名字。可以指定参数作为扩展宏的一部分。

大多数宏定义在效果上为常数。这些名字传统上为大写字母。例如，下面的定义创建了宏 ARRAY_SIZE，无论何时在源代码中出现该宏的地方都会插入值 512。

```
#define ARRAY_SIZE 512
```

该宏以后可用来声明指定尺寸的数组，如下：

```
int valarray[ARRAY_SIZE];
```

下面是一个众所周知的宏，使用参数来创建一个表达式，可以返回两值中的最小值：

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

这个宏可以被扩展到源代码中，使用它的名字以及分配给 a 和 b 的值：

```
result = min(44,uplim);
```

从这个宏中扩展出的代码看起来如下：

```
result = ((44) < (uplim) ? (44) : (uplim));
```

下面是应用于宏定义的特征和规则的列表：

- 宏定义包含一行。如果为了表达清楚或者是因为长度问题需要将宏写成多行，只需要在每行的后面加上连续符号反斜线，如下例所示，这个表达式返回一个特定范围内的随机整数。

```
#define ran(low,high) \
        ((int)random() % (high-low+1)) \
         + low
```

- 预处理程序按照顺序处理这些文本，而且只有在定义宏之后才会替换该宏。例如，

下面是 4 行代码，宏 B 会在它定义之前使用一次，定义之后使用一次。

```
#define A 100
sum = A + B;
#define B 200
sum = A + B;
```

这 4 行代码预处理的结果如下：

```
sum = 100 + B;
sum = 100 + 200;
```

- 替换是递归式的，因此可以在一个宏中嵌入另一个宏。就是说一旦发生了一次替换，预处理程序就会再次处理同样的文本进行更进一步的替换。下面的例子显示一个宏如何被另一个替换。

```
#define TANKARD TSIZE
#define TSIZE 100
tank1 = TANKARD;
#define TSIZE 200
tank2 = TANKARD
```

预处理这 5 行代码的结果如下：

```
tank1 = 100;
tank2 = 200;
```

- 要改变一个宏的定义，有必要删掉它再定义一次，如下例所示：

```
#define MLKEYVAL 889
#undef MLKEYVAL
#define MLKEYVAL 890
```

- 对于含有参数的宏来说，在宏的名字和扩号之间不能有空格。下面的例子显示一个有参数的宏以及一个字符串替换的简单例子：

```
#define showint(a) printf("%d\n",a)
#define incrint (a) a++
showint(300);
incrint(bbls);
```

下面就是前面几行的预处理结果：

```
printf("%d\n",300);
(a) a++(bbls)
```

- 宏的名字在字符串中是不能被替换的，如下例所示：

```
#define BLOCK 8192
```

```
printf("The BLOCK number.\n");
```

输出结果如下：

```
The BLOCK number.
```

- 通过处理传递给宏的参数名字，加上井号（#）就可将其“字符串化”。下面的例子中，宏 MONCK 包括它的字符串化的参数，还包含其他字符串（被邻近它们的字符串取代）：

```
#define MONCK(ARGTERM) \
   printf("The term " #ARGTERM " is a string\n")
MONCK(A to B);
```

输出如下：

```
The term A to B is a string
```

- 可以定义没有值的宏。尽管宏没有相关的值，还是可以定义这样的宏被#ifdef 和#ifndef 用作测试的标记。
- 可变的宏是具有可变数目参数的宏。这些参数由省略号代表，被保存在一个由逗号分隔的字符串中作为变量__VA_ARGS__，它会在宏的内部进行扩展。例如，下面的宏接受任何数目的参数：

```
#define err(...) fprintf(stderr,__VA_ARGS__)
err("%s %d\n","The error code: ",48);
```

下面是预处理这两行后的输出：

```
fprintf(stderr,"%s %d\n","The error code ",48);
```

可变的宏可以包含命名的参数（只要随后有参数的变量长度列表）。例如，下面的宏有两个固定参数，以及一个变量列表：

```
#define errout(a,b,...) \
   fprintf(stderr,"File %s   Line %d\n",a,b); \
   fprintf(stderr,__VA_ARGS__)
```

下面是使用该宏的例子：

```
errout(__FILE__,__LINE__,"Unexpected termination\n");
```

前面所有形式的可变宏至少有一个参数需要满足参数变量列表的需求，因为__VA_ARGS__前面是一个逗号，它用于宏内部的 fprintf()函数调用。作为连接操作符的一个特例，可以要求在__VA_ARGS__为空时，将它插入变量列表可以去掉逗号，如下：

```
fprintf(stderr, ##__VA_ARGS__)
```

### 3.1.2　#error 和#warning

#error 指示字会引起预处理程序报告致命错误或中断。它可用来捕获尝试按照某种不可能工作的形式进行编译的条件。例如，下面的例子只有在定义了__unix__的情况下才能成功编译：

```
#ifndef __unix__
#error "This section will only work on UNIX systems"
#endif
```

#warning 指示字和#error 指示字的工作原理一样，只是它的条件不是致命性错误，而且预处理程序在发出消息之后会继续进行下去。

### 3.1.3　#if，#elif，#else 和#endif

#if 指示字会计算一个数学表达式，并检查它的结果。如果结果不为零，就认为条件为真，并会编译条件代码。否则表达式被看作假，也不会编译代码。例如，下面的字符串只有在宏 COUNT 未被定义为零的情况下才会声明：

```
#if COUNT
char *desc = "The count is non-zero";
#endif
```

下面是应用于表达式和条件指示字的特征和规则的列表：

- 表达式可包含整数常数，如果宏被声明为有值，也可以包含宏名。
- 扩号用于指出表达式计算的顺序。
- 表达式包括的算术可为+，-，*，/，<<和>>操作符，这些算术符和 C 语言中相应整数算术符的用法一致。所有算术符都可操作系统平台支持的最大尺寸的整数，通常为 64 位整数。
- 表达式可以包含比较操作符>，<，>=，<=和==，它们和 C 语言中的相应操作符一样。
- 表达式可以包含逻辑操作符&&和||。
- 非（!）逻辑操作符可用于逆转表达式的结果。例如，如果 LIMXP 不大于 12，则下面表达式为真：

  ```
  #if !(LIMXP > 12)
  ```

- defined 操作符可用于确定一个宏是否被定义了。例如，如果已经定义了宏 MINXP，下面的表达式就为真：

```
#if defined(MINXP)
```

非操作符（!）通常和 defined 操作符一起使用来测试一个宏是否被定义，如下例所示：

```
#if !defined(MINXP)
```

- 有个标识符未被定义为宏总是会返回结果零。-Wundef 选项可用于在这种环境下产生警告。
- 具有参数的宏可以定义为只计算出结果零。-Wundef 选项可用于在这种情况下产生警告。
- #else 指示字可用来提供用于编译的可选代码，如下例所示：

```
#if MINTXT <= 5
#define MINTLOG 11
#else
#define MINTLOG 14
#endif
```

- #elif 指示字可用于提供一个或多个可选的表达式，如下例所示：

```
#if MINTXT <= 5
#define MINTLOG 11
#elif MINTXT == 6
#define MINTLOG 12
#elif MINTXT == 7
#define MINTLOG 13
#else
#define MINTLOG 14
#endif
```

### 3.1.4 #ifdef，#else 和#endif

跟在#ifdef 指示字后的代码行只有在指定宏被定义的情况下才会被编译。而指示字#ifdef 被另一个指示字#endif 终止。例如，下面的数组只有在定义了宏 MINTARRAY 的情况下才会被声明：

```
#ifdef MINTARRAY
int xarray[20];
#endif  /* MINTARRAY */
```

#endif 行中的注释不是必须的，但能提高代码的可读性。

#ifdef 指示字的反义词是#ifndef 指示字，它只在宏未被定义的条件下才编译这些条件代码。

#else 指示字用在#ifdef 的后面，用来提供可选的代码。在下面的例子中，如果定义了

MINTARRAY，数组的类型为整数，否则会为字符：

```
#ifdef MINTARRAY
int xarray[20];
#else
char xarray[20];
#endif /* MINTARRAY */
```

其他指示字也可作为条件编译的代码的一部分，包括#ifdef、#ifndef 和#if，但每个指示字都要和它自己的#endif 配对。

### 3.1.5 #include

include 指示字会查找指定的文件，将它的内容插入文本，就好像已经在文本编译程序中插入其内容一样。这种方式包含的文件通常是头文件（header），后缀为.h，但其实可为任意名字的任意文本文件。

include 指示字有两种形式。一种最常用于系统头文件，名字由尖扩号包围，而用户自己定义的头文件由引号包围，如下：

```
#include <syshead.h>
#include "userhead.h"
```

下面是一个特征和规则列表，可应用于#include 指示字：

- 尖扩号包围的文件名会令查找由使用-I 选项说明的目录开始，然后继续查找系统目录的标准集合。
- 由引号包围的文件名会导致首先查找当前工作目录（包含正在处理的源文件的目录），然后继续查找由带尖括号的指示字指定的目录。
- 在 UNIX 系统中，系统目录的标准集合如下：

  /usr/local/include
  /usr/lib/gcc-lib/*target*/*version*/include
  /usr/*target*/include
  /usr/include

- 会查找两个单独的目录列表来定位头文件。标准系统头文件在第二个列表中。-I 命令行选项将目录加入首先查找的列表。选项-prefix、-withprefix 和-idirafter 都可操作第二个列表查找的目录名。
- 如果 GCC 在编译 C++程序，预处理程序就会在查找其他标准系统目录之前先查找目录/usr/include/g++v3。
- 相对路径名可用作文件名。例如，如果指定#include <sys/time.h>，会在所有标准目录的子目录 sys 中查找头文件 time.h。

- 斜线符号总是解释为路径的分隔符——即使某些系统使用不一样的符号（例如反斜线）作为路径分隔符。因此，使用斜线作为路径分隔符就总是可移植的。
- 文件名按照字母逐个处理。它不能扩展宏，也没有特殊意义的字符。如果指定名字包含星号或者反斜线，那么文件名也必须要包含字母星号或反斜线字符。
- #define 指示字可用于指定头文件的名字，如下例所示：

```
#define BOGHEADER "bog_3.h"
#include BOGHEADER
```

- 如果#include 指示字包含非注释性代码，就一定会出错。
- 为查找文件，#line 指示字不会改变当前工作目录。
- -I-选项可用来修改-I 选项指定的查找目录。见附录 D 中的更多信息。

### 3.1.6 #include_next

#include_next 指示字只用于某些特殊情况。它用在头文件内部来包含其他头文件，会令新头文件的查找由找到当前头文件的目录之后的目录开始。

例如，如果头文件的正常查找会依次查找目录 A、B、C、D 和 E，而当前头文件位于目录 B，则#include_next 指示字会要求在目录 C、D 和 E 中查找新的头文件。

该指示字可用于增加或修改系统头文件的定义，而不需要修改文件本身。例如，系统头文件/usr/include/stdio.h 包含宏定义 getc，它会从输入流中读出单个字符。要改变这个宏定义，让它总是返回同一个字符，但保留头文件的其他内容，可以创建自己的 stdio.h 头文件，包含下面内容：

```
#include_next "stdio.h"
#undef getc
#define getc(fp) ((int)'x')
```

使用该头文件会包含系统的 stdio.h，以及自己重定义的宏 getc。

### 3.1.7 #line

调试器需要将文件名和行号与数据项和可执行代码关联起来，因此预处理程序会将这类信息插入编译程序的输出结果。有必要按这种方式跟踪原始名字和行号，因为预处理程序会组合一些文件。编译程序在编译插入目标代码中的表时，会使用这些数字。

通常，允许预处理程序通过计算来确定行号，这正是需要的，但也有可能用其他一些处理来去掉这些行号。例如，实现 SQL 语句的通常方法就是将它们写成宏，然后用特殊的处理器将这些宏扩展成具体的 SQL 函数调用。这些扩展可在很多行中运行，这样计算行号就很困难。SQL 处理会通过在输出中插入#line 指示字进行更正，这样预处理程序就会跟踪原始源代码的行号。

下面是可用于#line 指示字的特征和规则的列表：

- 为#line 指示字指定一个数字，会令预处理程序将当前行号替换为指定行号。例如，下面的指示字设置当前行号为 137：

```
#line 137
```

- 为#line 指示字指定行号和文件名，会令预处理程序改变行号以及当前文件的名字。例如，下面的指示字会设置当前位置为文件 muggles.h 的第一行：

```
#line 1 "muggles.h"
```

- #line 指示字修改预定义宏__LINE__ 和 __FILE__的内容。
- #line 指示字对由#include 指示字查找到的文件名或目录没有影响。

### 3.1.8　#pragma 和_Pragma

指示字#pragma 提供一种标准方法用来指定特定于编译程序的信息。根据标准，编译程序可以附带#pragma 指示字希望的任何意义。

所有 GCC pragma 都定义了两个词——第一个为 GCC，第二个为指定 pragma 的名字。

#### #pragma GCC dependency

dependency pragma 测试当前文件的时间戳，对比其他文件的时间戳。如果其他文件更新，就会发出警告消息。例如，下面的 pragma 测试文件 lexgen.tbl 的时间戳：

```
#pragma GCC dependency "lexgen.tbl"
```

如果 lexgen.tbl 比当前文件新，预处理程序就会产生如下消息：

```
warning: current file is older than "lexgen.tbl"
```

可在 pragma 指示字中加入其他文本，它会作为警告消息的一部分，如下例所示：

```
#pragma GCC dependency "lexgen.tbl" Header lex.h needs to be updated
```

它会创建下面的警告消息：

```
show.c:26: warning: current file is older than "lexgen.tbl"
show.c:26: warning: Header lex.h needs to be updated
```

#### #pragma GCC poison

poison pragma 在每次使用指定名字的时候都会发出消息。例如，可用它确保从未调用指定函数。下面的 pragma 在调用 memcpy 复本函数时就会发出警告消息：

```
#pragma GCC poison memcpy memmove
memcpy(target,source,size);
```

预处理程序会为该代码产生如下警告消息：

```
show.c:38:9: attempt to use poisoned "memcpy"
```

#pragma GCC system_header

由 system_header pragma 打头并随后继续到文件尾的代码被看作是系统头文件的一部分。编译系统头文件代码有一些不同，因为运行时库不能被写，因此它们是严格的纯 C 标准格式。限制所有警告消息（除了#warnings 指示字）。特殊情况下，一定的宏定义和扩展不会发出警告消息。

_Pragma

通常的#pragma 指示字不能作为宏扩展中的一部分包含进来，因此设计_Pragma 操作符是为了生成宏内部的#pragma 指示字。为创建宏内部的 poison pragma，代码如下：

```
_Pragma("GCC poison printf")
```

反斜线字符用作转义字符，因此可用这种方式插入引用的字符串来创建 dependency pragma：

```
_Pragma("GCC dependency \"lexgen.tbl\"")
```

### 3.1.9 #undef

#undef 指示字用于去掉前面由#define 指示字创建的宏定义。如果不再使用该宏，或要为该宏定义新值就可以使用#undef。

### 3.1.10 ##

可用于宏内部将两个源代码权标连接成一个的连接指示字。可用来构造不会被解析器错误解释的名字。例如，下面两个宏会实现连接操作：

```
#define PASTE1(a) a##house
#define PASTE2(a,b) a##b
result = PASTE1(farm);
result = PASTE1(ranch);
result = PASTE2(front,back);
```

下面是预处理这 5 行代码的结果：

```
result = farmhouse;
result = ranchhouse;
result = frontback;
```

## 3.2 预定义宏

GCC 编译程序预定义了大量的宏。至于到底定义哪些宏，以及宏中包含什么值，这依赖于被编译的语言、指定的命令行选项、使用的平台、目标平台、使用的编译程序的版本以及设置的环境变量。可在预处理程序中使用-dM 选项来察看完整列表，命令如下：

```
cpp -E -dM myprog.c | sort | more
```

由该命令输出的包含#define 指示字的宏列表，是处理特定输入源文件和所有包含头文件之后在预处理程序中定义的。

表 3-2 列出几乎总要定义的宏，以及对它们的描述。

**表 3-2　预定义宏的基本集合**

| 宏 | 描述 |
|---|---|
| __BASE_FILE__ | 引用的字符串，包含的是命令行中指定源文件的完整路径名（不一定是使用宏的所有文件）。参见__FILE__ |
| __CHAR_UNSIGNED__ | 定义该宏用来指出目标机器的字符数据类型是无符号的。limits.h 用它来确定 CHAR_MIN 和 CHAR_MAX 的值 |
| __cplusplus | 只在 C++程序中有定义。如果编译程序不完全符合标准，该宏定义为 1，否则它会定义为标准的年和月，格式符合 C 中的__STDC_VERSION__ |
| __DATE__ | 11 个字符的引用字符串，包括编译程序的日期。它的格式为“May　3 2002” |
| __FILE__ | 引用字符串，包含使用宏的源文件名。参见__BASE_FILE__ |
| __func__ | 同__FUNCTION__ |
| __FUNCTION__ | 引用字符串，包含当前函数的名字 |
| __GNUC__ | 该宏总是定义为编译程序的主要版本号。例如，如果编译程序版本号为 3.1.2，该宏定义为 3 |
| __GNUC_MINOR__ | 该宏总是定义为编译程序的次要版本号。例如，如果编译程序版本号为 3.1.2，该宏定义为 1 |
| __GNUC_PATCHLEVEL__ | 该宏总是定义为编译程序的修正版本号。例如，如果编译程序版本号为 3.1.2，该宏定义为 2 |
| __GNUG__ | 由 C++编译程序定义。无论何时定义了__cplusplus 和__GNUC__，就会定义该宏 |

（续表）

| 宏 | 描述 |
|---|---|
| __INCLUDE_LEVEL__ | 指出 include 文件当前深度的整数值。该值在基本文件（命令行中指定的文件）时为 0，而每次#include 指示字输入文件就会加 1 |
| __LINE__ | 使用宏的文件的行号 |
| __NO_INLINE__ | 在没有扩展内嵌函数的时候，该宏定义为 1，这可能因为没有优化或者不允许进行内嵌函数 |
| __OBJC__ | 如果程序被编译成 Objective-C，该宏定义为 1 |
| __OPTIMIZE__ | 无论何时只要指定任何级别的优化处理，该宏就会定义为 1 |
| __OPTIMIZE_SIZE__ | 如果设置进行尺寸上的优化而不是速度上的优化，该宏就会定义为 1 |
| __REGISTER_PREFIX__ | 该宏为一个权标（而不是字符串），它是注册器名的前缀。可用来编写能够移植到多种环境中的汇编语言 |
| __STDC__ | 定义为 1 指出该编译程序符合标准 C。在编译 C++和 Objective-C 时不定义该宏，而且在指定-traditional 选项的时候也不会定义该宏 |
| __STDC_HOSTED__ | 定义为 1 指出“宿主”的环境（其中含有完整的标准 C 库） |
| __STDC_VERSION__ | 长整数，指出标准版本号，形式为它的年和月。例如，标准的 1999 年修正版为 199901L。在编译 C++和 Objective-C 时不会定义该宏，而且在指定-traditional 选项的时候也不会定义该宏 |
| __STRICT_ANSI__ | 只有在命令行中指定-ansi 或-std 的时候，会定义该宏。在 GNU 头文件中使用它来限制标准中的那些定义 |
| __TIME__ | 引用 7 个字符的字符串，包含编译程序的时间。格式为“18:10:34” |
| __USER_LABEL_PREFIX__ | 该宏是一个权标（而不是字符串），用作汇编语言中的符号前缀。该权标依平台有所变化，但它通常是个下划线字符 |
| __USING_SJLJ_EXCEPTIONS__ | 如果异常处理机制为 setjmp 和 longjmp，该宏定义为 1 |
| __VERSION__ | 完整版本号。该信息没有特殊格式，但它至少含有主要和次要版本号 |

表 3-3 列出 C++关键字的集合，这些关键字可被用作操作符的名字，通常为标点符号字符。如果它们是#define 指示字创建的宏，它们就会被预处理程序处理。如果想要让同样的操作符在 C 或 Objective-C 中也可用，它们定义在头文件 iso646.h 中。

表 3-3　逻辑操作符的命名形式

| 操作符名 | 等价的符号形式 |
|---|---|
| and | && |
| and_eq | &= |
| bitand | & |
| bitor | \| |
| compl | ~ |
| not | ! |
| not_eq | != |
| or | \|\| |
| or_eq | \|= |
| xor | ^ |
| xor_eq | ^= |

## 3.3　只包含一次头文件

因为头文件会包含其他头文件，程序很有可能多次包含同样的头文件。这会导致出错信息，因为一些已经定义了的项目会被再次定义。为防止发生这样的情况，可以编写头文件来检测自身是否已经被包含了。下面便是一个这样的例子：

```
/* myheader.h */
#ifndef MYHEADER_H
#define MYHEADER_H
/* The body of the header file */
#endif   /* MYHEADER_H */
```

在这个例子中，头文件名为 myheader.h。第一行测试是否已经定义了 MYHEADER_H。如果已经定义，就会跳过整个头文件。如果 MYHEADER_H 未被定义，立即定义它并处理头文件。

系统头文件都使用了这种技术。它们的名字都由下划线字符开始，以防止和用户定义的其他名字冲突。定义名字的规则是全部使用大写字母，包含文件的名字。

GNU 预处理程序能识别这种结构，并保存头文件使用的过程。通过这种方式识别头文件名，并在已经包含该文件的情况下根本不读文件，就能够优化头文件的处理过程。

## 3.4 在出错消息中包含定位信息

预定义的宏可用来自动构造出错消息，它包含发生错误的位置的详细信息。预定义的宏__FILE__、__LINE__和__func__包含这些信息，但它们必须用于创建信息的那一点上。因此，如果写一个函数包含这些宏，就会在函数发生它的时候报告出错消息。

完美的解决方案就是定义一个包含它们的宏。以这种方式，当预处理程序扩展宏的时候，它们都处于正确位置并含有正确信息。如下是一个出错宏的例子，将错误消息写到标准错误中：

```
#define msg(str) \
   fprintf(stderr,"File: %s  Line: %d  Function: %s\n%s\n", \
      __FILE__,__LINE__,__func__,str);
```

为从代码中的任意位置激活该宏，是有必要指定字符串来描述错误的：

```
msg("There is an error here.");
```

这样做的另一个优势是处理错误条件的方法可以简单通过改变宏来实现。它可以转换成抛出异常或将错误信息记录到文件中。由本例产生的消息看起来如下：

```
File: hamlink.c  Line: 822  Function: hashDown
There is an error here
```

## 3.5 去掉适当位置的源代码

在开发软件的过程中，经常有必要去掉一些代码，并在需要的时候进行恢复。代码可用注释包围起来，但这会引起问题，因为C语言中的注释不会互相嵌套，而在要被去掉的代码中可能含有大量注释。一种清晰而安全的方式就是通过使用预处理程序的#if指示字来省略代码，如下：

```
#if 0
   /* The code being removed */
#endif
```

这不仅能够清晰地处理注释，而且很明显就是要有意地去掉这些代码。

## 3.6　产生 make 程序的描述文件

预处理程序可用于读取源文件，并能产生 make 程序的描述文件中的依赖行。例如，下面的命令使用-E 指示编译程序去激活预处理程序，然后不需要编译或连接就终止。-M 选项指示预处理程序输出完整的依赖行：

```
gcc -E -M trick.c
```

源文件 trick.c 包含 include 语句，包含系统文件<stdio.h>和本地文件"barrow.h"，但依赖列表不仅包括这些文件，还包括因此包含的所有文件。结果的依赖行看起来如下：

```
trick.o: trick.c /usr/include/stdio.h /usr/include/features.h \
  /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
  /usr/lib/gcc-lib/i386-redhat-linux/2.96/include/stddef.h \
  /usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
  /usr/include/bits/sched.h /usr/include/libio.h
  /usr/include/_G_config.h \
  /usr/include/wchar.h /usr/include/bits/wchar.h
  /usr/include/gconv.h \
  /usr/lib/gcc-lib/i386-redhat-linux/2.96/include/stdarg.h \
  /usr/include/bits/stdio_lim.h barrow.h
```

如附录 D 中所述，选项-MD、-MMD、-MF、-MG、-MP、-MQ 和-MT 可用来创建不同于选项-M 的其他方式和格式的依赖关系。使用这些选项创建 make 程序的描述文件的例子在第 14 章有所描述。

## 3.7　命令行选项和环境变量

有大量的命令行选项可用来指出预处理操作的方式。这里列出的选项在附录 D 中有详细描述。

| | |
|---|---|
| -A | --include-with-prefix-after |
| -A- | --include-with-prefix-before |
| --assert | -iprefix |
| -C | -isystem |
| -D | -iwithprefix |
| --define-macro | -iwithprefixbefore |
| --dependencies | -M |

-fident
-fpreprocessed
-H
-I
-I-
-idirafter
-imacros
-include
--include-barrier
--include-directory
--include-directory-after
--include-prefix
--include-with-prefix
--preprocess
--print-missing-file-dependencies
-remap
--trace-includes
-trigraphs
-U
-undef
--undefine-macro
-MD
-MF
-MG
-MM
-MMD
-MP
-MQ
-MT
—no-line-commands
--no-standard-includes
-nostdinc
-nostdinc++
-P
--user-dependencies
-Wp
--write-dependencies
--write-user-dependencies
-Wsystem-headers
-Wundef
-Wunknown-pragmas

下面是环境变量的列表，可用来设置向预处理程序传递指令。这些环境变量在附录 B 有所描述。

```
C_INCLUDE_PATH,   CPATH,   CPLUS_INCLUDE_PATH,   DEPENDENCIES_OUTPUT,
OBJC_INCLUDE_PATH, SUNPRO_DEPENDENCIES
```

# 第 4 章

# 编译 C 程序

本章描述的命令和选项可将 C 程序编译为目标文件、可执行程序和库。本章还概括了 GCC 支持的各种 C 程序标准，并分别介绍了只有 GCC 才支持的 C 语言扩展功能。

UNIX 上最原始的 C 编译程序叫做 CC（C Compiler，C 编译程序）。源自于此，GNU 最早的 C 编译程序叫做 GCC（GNU C Compiler，GNU 的 C 编译程序）。现在的 GCC 缩写仍保持不变，但其意义已经演变为 GNU 编译程序集合（GNU Compiler Collection），这是因为该编译程序已经发展成为能够支持多种语言的编译程序集合了。然而 GCC 最基本最本质的结构仍是 C 编程语言，而原始的编译程序代码就是用来编译 C 语言程序的。所幸的是，C 语言的结构使得它能够代表一种低级的、类似于硬件语言的操作，因此在原始编译程序的基础上，可以构造其他语言的编译程序。

## 4.1 编译基础

表 4-1 列出了用于编译和连接 C 程序所需的文件扩展名。附录 D 中的表列出了 GCC 支持的所有扩展名。

**表 4-1 C 程序中的文件名后缀**

| 扩展名 | 文件内容 |
|---|---|
| .a | 静态对象库（文档） |
| .c | 需要预处理的 C 语言源代码 |
| .h | C 语言源代码的头文件 |
| .i | 无需预处理的 C 语言源代码。该类文件是编译过程的中间产物 |
| .o | 目标文件，格式和应用的连接相符。该类文件是编译过程的中间产物 |
| .s | 汇编语言代码（assembly language code）。该类文件是编译过程的中间产物 |
| .so | 共享对象库（shared object library） |

### 4.1.1 单源程序到可执行程序

下面是一个非常简单的程序“hello, world”的源代码：

```
/* helloworld.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
   printf("hello, world\n");
   return(0);
}
```

将该程序编译为可执行代码的最简单最直接的方法，就是将源代码保存到名为helloworld.c的文件中，再键入如下命令：

```
$ gcc helloworld.c
```

通过检查命令行中的文件名后缀，编译程序可以确定所编译的是C语言源文件。GCC采取的默认动作是将源文件编译为目标文件，然后将目标文件连接到可执行文件，最后再删除目标文件。由于该命令没有指定生成的可执行文件的名字，所以编译程序会用当前目录下的默认文件名a.out。在命令行中键入该程序名，会运行该程序并显示结果。

```
$ a.out
hello, world
```

选项-o可指定编译程序输出的可执行程序名。以下命令会产生一个可执行程序，名为howdy：

```
$ gcc helloworld.c -o howdy
```

在命令行键入该程序名，会运行该程序，结果如下：

```
$ howdy
hello, world
```

### 4.1.2 源文件到目标文件

选项-c会明确指示GCC去编译源代码，但不在硬盘上留下目标文件，且跳过将目标文件连接到可执行程序这一步。这种情况下的默认输出文件名和输入文件名一样，但以.o作后缀。例如，下面的命令会生成目标文件helloworld.o：

```
$ gcc -c helloworld.c
```

选项-o可用于修改生成的目标文件名。下面的命令会生成目标文件，名为harumph.o：

```
$ gcc -c helloworld.c -o harumph.o
```

在构造目标库的过程中，或在创建目标文件集合的时候，其中的目标文件可用于之后的连接操作，有条命令可由多个源文件创建出目标文件。下面的命令会创建目标文件 arglist.o、ponder.o 和 listsort.o：

```
$ gcc -c arglist.c ponder.c listsort.c
```

### 4.1.3 多源文件到可执行文件

GCC 编译程序可以自动处理连接，甚至在编译多个源文件的时候也可以自动连接。例如，下面保存在文件 hellomain.c 中的源代码，调用函数 sayhello()：

```
/* hellomain.c */
void sayhello(void);
int main(int argc,char *argv[])
{
    sayhello();
    return(0);
}
```

而以下源代码保存在文件 sayhello.c 中，定义了函数 sayhello()：

```
/* sayhello.c */
#include stdio.h
void sayhello()
{
    printf("hello, world\n");
}
```

下面的命令将两个程序编译成一个目标文件，并将它们连接到一个可执行程序 hello 上，最后删掉目标文件：

```
$ gcc hellomain.c sayhello.c -o hello
```

### 4.1.4 预处理

选项-E 指示编译程序只进行预处理操作。下面的命令会预处理 helloworld.c 源文件，并把结果列表显示在标准输出上：

```
$ gcc -E helloworld.c
```

选项-o 用于将预处理后的代码定位到某个文件上。如前面表 4-1 所示，保存不需要预处理的 C 语言源代码的方式就是将它保存在以.i 为后缀的文件中：

```
$ gcc -E helloworld.c -o helloworld.i
```

### 4.1.5 生成汇编语言

选项-S 指示编译程序生成汇编语言代码，然后停止。下面的命令会从 C 语言源文件 helloworld.c 中创建汇编语言文件，名为 helloworld.s：

```
$ gcc -S helloworld.c
```

汇编语言的形式依赖于编译程序的目标平台。如果编译多个源文件，会为每个源文件都生成一个汇编语言模块。

### 4.1.6 创建静态库

静态库是一些.o 文件的集合，它们是由编译程序按照通常的方式生成的。将程序连接到库中的目标文件和将程序连接到目录中的目标文件是一样的。静态库的另一个名字是文档（archive），而管理这些文档内容的工具叫做 ar。

要构造一个库，最基本的一步是要编译库中的目标模块。例如，以下两个源文件名为 hellofirst.c 和 hellosecond.c：

```
/* hellofirst.c */
#include <stdio.h>
void hellofirst()
{
   printf("The first hello\n");
}
/* hellosecond.c */
#include <stdio.h>
void hellosecond()
{
   printf("The second hello\n");
}
```

可用下面命令将这两个源文件编译成目标文件：

```
$ gcc -c hellofirst.c hellosecond.c
```

使用工具 ar 的-r 选项，可以创建新的库，并会将目标文件插入其中。选项-r 会创建库，如果库中不存在所命名的目标模块，就会将它加入文档（如果必要，也会替换原来的目标模块）。下面的命令创建名为 libhello.a 的库，其中包含本例的两个目标模块：

```
$ ar -r libhello.a hellofirst.o hellosecond.o
```

现在该库已经完成，可以使用了。下面的程序 twohellos.c 会调用新库中的这两个函数：

```
/* twohellos.c */
void hellofirst(void);
void hellosecond(void);
```

```
int main(int argc,char *argv[])
{
    hellofirst();
    hellosecond();
    return(0);
}
```

如下所示，通过在命令行中指定库，用一条命令就能够编译并连接程序 twohellos：

```
$ gcc twohellos.c libhello.a -o twohellos
```

静态库的命名习惯是以字母 lib 开头，以后缀.a 结束。所有系统库都使用这种命名规则，并且允许通过选项-l（ell），在命令行中使用库名的缩写形式。下面命令和前面命令的惟一区别在于期望 gcc 对库进行查找的位置有所不同：

```
$ gcc twohellos.c -lhello -o twohellos
```

明确指出完全路径名会令编译程序在所指路径中查找库。库名既可指明绝对路径（例如 /usr/worklibs/libhello.a），也可以是到当前目录的相对路径（例如../lib/ libhello.a）。选项 -l 不能指明路径，但是可以指示编译程序在系统库中进行查找。

### 4.1.7 创建共享库

共享库是目标文件的集合，但是这些目标文件是由编译程序按照特殊方式生成的。对象模块的每个地址（变量引用和函数调用）都是相对地址，不是绝对地址。因此允许在运行程序的时候，可以动态加载和执行共享模块。

构造共享库最基本的一步是编译库中的对象模块。例如，下面的两个源文件名为 shellofirst.c 和 shellosecond.c：

```
/* shellofirst.c */
#include <stdio.h>
void shellofirst()
{
    printf("The first hello from a shared library\n");
}
/* shellosecond.c */
#include <stdio.h>
void shellosecond()
{
    printf("The second hello from a shared library\n");
}
```

使用下列命令可将这两个源文件编译成一个目标文件：

```
$ gcc -c -fpic shellofirst.c shellosecond.c
```

选项-c 明确指出编译程序要生成的.o 目标文件。而选项-fpic 使得输出的对象模块是按照可重定位地址(relocatable addressing)方式生成的。缩写 pic 代表位置独立代码(position independent code)。

下面的 gcc 命令使用目标文件构造共享库 hello.so:

```
$ gcc -shared shellofirst.o shellosecond.o -o hello.so
```

选项-o 为输出文件命名，而后缀.so 告诉 GCC 该目标文件是要连接到共享库的。通常，连接程序要为主函数 main()定位，并用它作为程序的入口，但输出模块没有这样的入口，选项-shared 对防止出错信息是必要的。

编译程序将后缀为.c 的文件看作是 C 语言源代码程序，并且知道如何将它编译成目标文件。因此，可以将前面的两个命令合成一个命令，用下面的命令可以将模块直接编译并保存为共享库:

```
$ gcc -fpic -shared shellofirst.c shellosecond.c -o hello.so
```

下面是文件 stwohellos.c 中的程序，是主函数，它调用了共享库中的两个函数:

```
/* stwohellos.c */
void shellofirst(void);
void shellosecond(void);
int main(int argc,char *argv[])
{
    shellofirst();
    shellosecond();
    return(0);
}
```

可以使用下列命令将该程序编译并连接到共享库:

```
$ gcc stwohellos.c hello.so -o stwohellos
```

程序 stwohellos 已经可以运行了，但要运行程序必须也要能够定位共享库 hello.so，因为库中的函数必须在运行的时候载入。关于共享库定位的相关信息，可参见第 12 章。

### 4.1.8 重写命名约定

如果环境需要自己命名 C 语言源程序的文件名，而不是使用.c 作后缀，那么可通过使用选项-x 为该语言重写默认规则。例如，下面的命令会编译文件 helloworld.jxj 中的 C 语言源代码，并创建可执行程序 helloworld:

```
$ gcc -xc helloworld.jxj -o helloworld
```

通常不需要选项-x，但会假设所有具有不明后缀的源文件都可被连接识别，因此这些

名字会不加修改地传给连接。-x 选项作用于命令中所有具有未知后缀的文件。例如，下面的命令会假设 align.zzz 和 types.xxx 都是 C 语言源文件：

```
$ gcc -c -xc align.zzz types.xxx
```

## 4.2 标准

通过使用命令行选项可以编译任何 C 程序，从符合原始语法规则的 C 程序（目前，我们也称之为传统的 C 程序）到最新的具有 GNU 各种扩展功能的标准语言。默认情况下，GCC 编译的代码可以使用最新标准，而且也可以使用 GNU 提供的所有扩展功能。可用选项列在表 4-2 中。附录 D 包括对所有选项更详细的描述。

**表 4-2　控制 C 语言版本的选项**

| 选项 | 描述 |
|---|---|
| -ansi | 编译标准程序，也和 GNU 后缀兼容 |
| -pedantic | 严格按照标准，提示警告消息 |
| -std=c89 | ISO C89 标准 |
| -std=C99 | ISO C99 标准 |
| -std=gnu89 | 具有 GNU 扩展功能的 ISO C89 标准，以及某些 C99 的功能 |
| -traditional | 兼容最原始的 C 语法检查 |

与标准兼容的 C 程序和不兼容的 C 程序之间最本质的区别在于函数调用的参数，以及是否存在函数原型。为克服这个问题，GCC 编译程序提供 -aux-info 选项，可用于为函数自动生成原型。例如，下面的命令会创建头文件 slmwrk.h，其中包括源文件 slmwrk.c 中定义的所有函数的函数原型：

```
$ gcc slmwrk.c -aux-info slmwrk.h
```

下面的命令用于创建头文件 prototypes.h，它包括整个目录中所有 C 语言源文件中函数的函数原型：

```
$ gcc *.c -aux-info prototypes.h
```

通过工具 protoize，可将 C 程序中的函数转换为 ANSI 标准，对此第 14 章有进一步的描述。

## 4.3 C 语言扩展

使用选项-ansi 和-std 可以设置 C 编译程序，要求它根据某种 C 标准进行编译，但也可

使用其他的扩展功能。以前版本 GCC 的很多扩展功能都已经被指定为 C 语言新标准的一部分了，而以下章节所列的扩展功能不属于任何标准 C。除某些特殊情况外，只有 GCC 才能支持这些功能扩展。

使用 C 语言扩展时，指明选项-pedantic（同时包括其他选项）会引发警告消息，但可以通过关键字__extension__扩展表达式来禁止警告消息。

由 GCC 的内部结构决定，这里描述的很多 C 语言的补充、扩展都可用于 C++和面向对象的 C（Objective-C）。C++和 Objective-C 的编译程序都使用了部分的 C 编译程序，因此对 C 或者预处理加以补充，某些情况下也会为其他语言增加相同的内容。但当某些扩展与语言的基本定义相冲突时，就不会使用这些新的扩展功能，或者要为 C++或 Objective-C 重新开发满足它们要求的其他形式。

### 4.3.1 对齐

__alignof__操作返回数据类型或指定数据项的分界对齐（boundary alignment）。下面的程序显示了对齐的每种数据类型：

```
/* align.c */
#include <stdio.h>
typedef struct {
   double dvalue;
   int ivalue;
} showal;

int main(int argc,char *argv[])
{
   printf("__alignof__(char)=%d\n",__alignof__(char));
   printf("__alignof__(short)=%d\n",__alignof__(short));
   printf("__alignof__(int)=%d\n",__alignof__(int));
   printf("__alignof__(long)=%d\n",__alignof__(long));
   printf("__alignof__(long long)=%d\n",__alignof__(long long));
   printf("__alignof__(float)=%d\n",__alignof__(float));
   printf("__alignof__(double)=%d\n",__alignof__(double));
   printf("__alignof__(showal)=%d\n",__alignof__(showal));
   return(0);
}
```

实际上的数据对齐根据硬件系统的不同而不同，因为该需求是由机器设置的。对齐既可是纯粹的硬件需求，也可是一种分界的建议，以提高访问数据的效率。

### 4.3.2 匿名联合

在结构中，可以声明某个联合而不用指出名字，这样可以直接使用联合的成员，就好像它们是结构中的成员一样。下面的例子为相同的四个字节提供两个名字和两个数据类型：

```
struct {
    char code;
    union {
        char chid[4];
        int numid;
    };
    char *name;
} morx;
```

结构的成员可使用 morx.code、morx.chid、morx.numid 和 morx.name 访问。

### 4.3.3 变长数组

数组大小可声明为动态值，在运行时才指定尺寸。使用表达式定义数组下标可以实现变长数组。例如，下面的函数可将输入的两个字符串串联起来，并在其中插入一个空格。

```
void combine(char *str1,char *str2)
{
    char outstr[strlen(str1) + strlen(str2) + 2];

    strcpy(outstr,str1);
    strcat(outstr," ");
    strcat(outstr,str2);
    printf("%s\n",outstr);
}
```

变长数组可以作为参数进行传递，如下例所示：

```
void fillarray(int length,char letters[length])
{
    int i;
    char character = 'A';
    for(i=0; ilength; i++)
        letters[i] = character++;
}
```

通过向前声明可以颠倒参数的顺序，这样在读数组 letters 的时候，就已经知道 length 的类型了，如下：

```
void fillarray(int length; char letters[length], int length)
```

为满足要求，向前声明的数目可为任意数目（由逗号或分号分隔），只要最后一个跟着的是分号就可以了。

### 4.3.4 零长度数组

GNU C 允许声明长度为零的数组，用于创建变长结构。只有当零长度数组是结构 struct

的最后一个成员的时候，才有意义。可通过为数组分配必要的空间，来简化对数组大小的声明。下面的程序展示这种方法：

```
/* zarray.c */
#include <stdio.h>
typedef struct {
    int size;
    char string[0];
} vlen;

int main(int argc,char *argv[])
{
    int i;
    int count = 22;
    char letter = 'a';

    vlen *line = (vlen *)malloc(sizeof(vlen) + count);
    line->size = count;
    for(i=0; i<count; i++)
        line->string[i] = letter++;

    printf("sizeof(vlen)=%d\n",sizeof(vlen));

    for(i=0; i<line->size; i++)
        printf("%c ",line->string[i]);
    printf("\n");

    return(0);
}
```

例子中的 printf()语句打印值 4，因为 sizeof 操作只能检查出结构中 int 值的尺寸大小。程序 zarray 的输出结果如下：

```
sizeof(vlen)=4
a b c d e f g h i j k l m n o p q r s t u v
```

将数组定义为未完成类型也能实现同样的功能。这种方法的优点是满足标准 C，不仅如此，对于前面例子而言，它的使用方法也完全一致。还有一点好处，数组的大小可以在初始化的时候指定，如下例所示，这里数组的大小被设为 4 个字符：

```
/* incarray.c */
#include <stdio.h>
typedef struct {
    int size;
    char string[];
```

```
} vlen;

vlen initvlen = { 4, {'a','b','c','d' } };

int main(int argc,char *argv[])
{
    int i;

    printf("sizeof(vlen)=%d\n",sizeof(vlen));
    printf("sizeof(initvlen)=%d\n",sizeof(initvlen));
    for(i=0; i<initvlen.size; i++)
        printf("%c ",initvlen.string[i]);
    printf("\n");
    return(0);
}
```

本例输出如下：

```
sizeof(vlen)=4
sizeof(initvlen)=4
a b c d
```

### 4.3.5 属性

关键字__attribute__可用于为函数或数据声明赋属性值。给函数分配属性的主要目的是为了让编译程序可以执行优化处理。分配给函数的属性位于函数原型的声明中，如下例所示：

```
void fatal_error() __attribute__ ((noreturn));
 . . .
void fatal_error(char *message)
{
    fprintf(stderr,"FATAL ERROR: %s\n",message);
    exit(1);
}
```

本例中 noreturn 属性告诉编译程序该函数不会返回到调用者，这样优化器就会忽略在函数返回时通常会执行的所有代码。

可为一个声明赋多个属性，只需将它们加入包含列表并用逗号分隔。例如，下面声明分配的属性可以确保编译程序不会修改全局变量，而且函数不会被扩展为内嵌函数：

```
int getlim() __attribute__ ((pure,noinline));
```

属性也可赋给变量和结构中的成员。例如，为确保结构中的域具有明确的对齐方式，可以声明如下：

```
struct mong {
    char id;
    int code __attribute__ ((align(4)));
};
```

表4-3列出了函数属性集合，表4-4列出了用于数据声明的属性，而表4-5列出了可赋给数据类型声明的属性。

**表4-3 声明函数可用的属性**

| 属性 | 描述 |
|---|---|
| alias | 函数定义具有该属性，会使该定义成为其他函数的弱别名（weak alias）。该属性可和weak属性一起使用，用于定义弱别名，下面例子创建 centon()作为__centon()的弱别名：<br>`int __centon() { return(100); }`<br>`void centon() __attribute__ ((weak,alias ("__centon")));`<br>C++中mangled的名字[①]必须指明。不是所有机器都支持该属性 |
| always_inline | 声明为内嵌的函数如果具有该属性，总会被扩展为内嵌代码，即使没有指明进行优化处理。通常情况下，只有优化处理的函数才是内嵌的。下面是该种函数原型的一个例子，它总会扩展为内嵌代码：<br>`inline void infn()__attribute__((always_inline));` |
| const | 具有该属性的函数和属性pure一样，但从不访问全局内存（global memory）中的任何值。它为优化器（optimizer）提供的自由比pure提供得多，因为无需在调用函数前必须确保更新所有全局变量值 |
| constructor | 具有该属性的函数会被自动调用，比主函数main()的调用还早。参见属性destructor |
| deprecated | 具有该属性的函数无论何时被调用，都会引起编译程序提示警告消息。警告消息包括废止函数（deprecated function）的位置，以便用户得到更多有关它的信息。 |
| destructor | 具有该属性的函数，会在主函数main()返回或者调用函数exit()之后被调用。参见属性constructor |
| format | 具有该属性的函数具有一个字符串格式的参数和大量格式化的参数。这使得编译程序可以依据参数列表检查内容的格式，验证类型是否与格式匹配。因为格式的种类不同，因此有必要指出校验是否用于printf、scanf、strftime或strfmon风格。例如，如下属性说明传入函数的第二个参数是字符串格式的，而且期望该格式的字符串为printf类型，而变量长度（variable-length）参数列表从第三个参数开始： |

① 译注：mangled主要表达的意思应该是“C++语言中为函数重新命名，在编译程序内给它们定义一个惟一可标示的名字，也许不具有可读性，但类似于C评议中的函数名，没有类的含义”。

（续表）

| 属性 | 描述 |
| --- | --- |
| | int logprintf(void *log, char *fmt, ...)<br>__attribute__((format(printf,2,3)));<br><br>只有同时指定了-Wformat 选项，才有可能出现警告消息提示格式字符串的无效。 |
| format_arg | 具有该属性的函数接收字符串格式的参数，并对其进行修改，使得能够传递给函数printf()、scanf()、strftime()或 strfmon()类型函数。如果-Wformat-nonliteral 选项被设置以检测非常数的格式化字符串，那么该属性会禁止发布警告消息。下例展示了为函数设置该属性，而此函数将格式化字符串作为第二个参数：<br>void fedit(int ndx,const char *fmt)<br>__attribute__ ((format_arg(2)));<br><br>malloc 具有该属性的函数会通知编译程序对待它要和对待函数 malloc()一样。出于优化目的，编译程序会假设返回的指针不能和其他混淆。 |
| no_instrument_function | 具有该属性的函数不会被检测，也不会由编译程序插入描述代码（profiling code），即使已经设置了-finstrument-functions 选项。 |
| noinline | 具有该属性的函数永远不会被扩展为内嵌代码。 |
| noreturn | 具有该属性的函数不会返回。 |
| pure | 具有该属性的函数除了要求考虑返回值外，没有任何副作用。就是说，它不会修改全局变量、参数的位置以及文件内容。和属性 const 不同，该函数是可以访问全局变量的。因此，编译程序可以执行通用子表达式优化（common subexpression optimization），因为能够保证所有值都是稳定不变的。 |
| section | 具有该属性的函数将它的汇编语言代码（assembly language code）放在命名的部分（section），而不是默认的 text 部分。下面的例子函数被放在 specials 部分：<br>void mspec(void) __attribute__((section("specials")));<br><br>如果系统不支持分区（sectioning），就会忽略该属性。参见附录 D 的-ffunction-sections。 |
| used | 具有该属性的函数会令编译程序为函数体生成代码，即使编译程序确定并未使用该函数。如果函数只被内嵌集合调用，就很有用。 |
| weak | 具有该属性的函数会将它的名字发布为弱符号（weak symbol），而不是完整的名字。这主要用于库常规操作的命名上，因为它们会被用户代码所替代。 |

**表 4-4 声明变量可用的属性**

| 属性 | 描述 |
| --- | --- |
| aligned | 为具有该属性的变量分配内存地址，为该队列指定数目的偶数倍。例如，如下声明会为 alivalue 分配 32 位地址： |

（续表）

| 属性 | 描述 |
| --- | --- |
| | `int alivalue __attribute__ ((aligned(32)));`<br>在某些系统中，队列可以方便地提供确定的汇编语言指令。而对于需要提供文件中数据格式的结构的域，它也是很有用的。如果没有指明队列数目，编译程序会为该队列指定最大的队列数据，可使用该硬件环境下的所有数据，例如下面的例子：<br>`short shlist[312] __attribute__ ((align));` |
| deprecated | 具有该属性的变量每次被引用的时候，都会令编译程序发布警告消息。 |
| mode | 具有该属性的变量的大小和指定模式的尺寸相匹配。模式可为字节（byte）、字（word）或指针（pointer）。模式属性 mode 决定数据类型。例如，下列创建的整数尺寸为单字节：<br>`int x __attribute__ ((mode(byte)));` |
| nocommon | 具有该属性的变量不会作为公共变量分配，而是分配在自己的空间内。这种变量初始值为全零。明确使用命令行参数-fno-common 会令所有变量都具有该属性。 |
| packed | 具有该属性的变量具有最小可能的队列。变量会被分开，和前者距离不会超过一个字节。在结构 struct 中，具有该属性的域和它前面的域之间没有空间。例如，下面的结构 struct，数组 zar 的开始是在结构最顶端的一个字节之后：<br>`struct zrecord {`<br>`char id;`<br>`int zar[32] __attribute__ ((packed));`<br>`};`<br>参见附录 D 中选项-fpack-struct 和-Wpacked。 |
| section | 具有该类属性的变量会被放置在命名的部分（section），而不是默认的 data 或 bss 部分。下面函数的例子，放置在名为 domx 的部分：<br>`struct domx __attribute__ ((section("domx"))) = { 0 };`<br>`int trigger __attribute__ ((section("MONLOG"))) = 0;`<br>由于连接程序处理数据的方式不同，在它自己内部声明的数据必须具有初始值。不支持分区（sectioning）的系统会忽略该属性。通过命令行选项-fno-common，可以强制执行变量初始化。 |
| unused | 具有该类属性的变量会告诉编译程序，该属性也许未被使用，而且不会发布警告消息。 |
| vector_size | 具有该类属性的变量分配的是向量（vector）定义的所有空间。例如下面的例子声明了浮点数据类型的向量： |

（续表）

| 属性 | 描述 |
| --- | --- |
| | `float fvec __attribute__((vector_size(32)));`<br>假设浮点数据类型为 4 字节长，该声明创建了 8 个浮点变量，总共 32 字节。该属性只对整数和纯量（real scalars）有效。 |
| weak | 具有该属性的变量将名字发布为弱符号，而不是完整的名字。这主要用于库变量的命名，它们会被替换为用户代码。 |

表 4-5　声明数据类型可用的属性

| 属性 | 描述 |
| --- | --- |
| aligned | 具有该属性的类型所分配的内存地址为指定队列的偶数倍。例如，下面的结构 struct 会位于 32 位的地址范围内：<br>`struct blockm {`<br>`  char j[3];`<br>`} __attribute__ ((aligned(32)));`<br>将属性 aligned 用于结构 struct 中第一个域，可能会影响整个结构。属性 aligned 只可用于增长队列，而不能用于减少队列。一些连接程序会强制编译程序限制队列的最大值。该属性也可用于 typedef 定义的类型：<br>`typedef int alint __attribute__ ((aligned(8)));`<br>如果没有指定队列数目，编译程序会为该项分配最大的队列数目，可用于该硬件环境下的所有数据。如下例所示：<br>`typedef short alshort __attribute__ ((align));` |
| deprecated | 具有该属性的类型每次用于声明变量的时候，都会令编译程序发布警告消息。消息包括类型声明的位置信息。 |
| packed | 具有该属性的结构 struct 或联合 union 会使用最小的空间。这等同于为结构或联合中的每个成员指定 packed 属性。<br>该属性可被声明，放置在 enum 声明的关括号（closing brace）之后。命令行选项 -fshort-enum 与为所有枚举 enum 分别使用 packed 是一样的。 |
| transparent_union | 具有该属性的联合 union，用于参数的数据类型，可使函数能够接受所有定义在该联合中的类型。下面的例子用到的透明联合展示了调用同一个函数，可以具有三种不同的参数类型： |

（续表）

| 属性 | 描述 |
| --- | --- |
| | /* transp.c */<br>#include <stdio.h><br>typedef union {<br>float *f;<br>int *i;<br>} fourbytes __attribute__ ((transparent_union));<br><br>void showboth(fourbytes fb);<br><br>int main(int argc,char *argv[])<br>{<br>int ivalue = 2562;<br>float fvalue = 898.44;<br>fourbytes fb;<br>fb.i = &ivalue;<br><br>showboth(&ivalue);<br>showboth(&fvalue);<br>showboth(fb);<br><br>return(0);<br>}<br><br>void showboth(fourbytes fb)<br>{<br>printf("The int value: %d\n",*fb.i);<br>printf("The float value: %f\n",*fb.f);<br>}<br><br>声明函数 showboth()，所需的参数为联合 fourbytes，但因为联合已经声明为具有属性 transparent_union，所以联合中声明的所有类型也都可传递给函数。本例包含对函数的调用，包括浮点小数的地址、整数的地址和联合自己的地址。 |
| unused | 具有该属性的类型使得所有该类型的数据都显示为没被使用，所以不会为它们发布警告消息。 |

### 4.3.6 返回值的复合语句

复合语句是大括号包围的一个语句块。复合语句有它自己的范围级别，可以声明自己的局部变量，如下例所示：

```
{
    int a = 5;
    int b;
    b = a + 5;
}
```

在GNU C中，圆括号包围的复合语句可以生成返回值，如下例所示，它会返回值8：

```
rslt =  ({
            int a = 5;
            int b;
            b = a + 3;
        });
```

返回结果的类型和值是复合语句中最后一句的类型和值。

在写宏时，这种结构很有用。但当参数为表达式，而且需要多次计算的时候，宏就会出问题。例如，下面的宏返回一个等于或大于指定数目的偶数，如果需要，要为参数加1：

```
#define even(x) (2*(x / 2) == x ? x : x + 1)
```

该宏只有在计算表达式x时会产生副作用，其他情况还是可以工作的。例如，下面的声明会产生未定义的结果：

```
int nexteven = even(value++);
```

下面的宏实现了同样的功能，但它只对表达式计算一次，并将结果保存在局部变量中：

```
#define evenint(x) \
    ({ int y = x; \
       (2*(y / 2) == y ? y : y + 1); \
    })
```

需要注意，这种功能扩展在C++中是不能正常工作的，所以在C++ 程序的头文件中使用它会产生问题。问题来自于宏内部临时变量的析构函数，因为它的运行早于内嵌函数的析构函数。

### 4.3.7 条件操作数省略

条件表达式中，条件真或假是由表达式结果是否为零决定的。如下语句，如果y非零，则将y赋给x：

```
x = y ? y : z;
```

如果第一次计算表达式y为非零，它就会被计算两次。如果表达式如下，则y的第二次计算可以忽略：

```
x = y ? : z;
```

如果表达式 y 有副作用而不应多次计算时，这种方法就非常有用。

### 4.3.8 枚举不完全类型

枚举标识符 enum 可以声明枚举，而无需明确指定列表中的每个值，声明方式和结构 struct 一样，只要声明名字，而无需指定内容。不完全枚举可用于函数原型声明指针。

下面的例子声明一个不完全数组，之后会实际声明该数组：

```
enum color_list;
 . . .
enum color_list { BLACK, WHITE, BLUE };
```

### 4.3.9 函数参数构造

以下三个嵌入函数可用来将当前函数的参数直接传递给其他函数，然后将结果返回给原始调用者。对于函数，没有必要了解传递的参数的所有情况。

下面的函数找到并记录了参数的描述信息：

```
void *__builtin_apply_args(void);
```

一旦记录了参数信息，就可用下面的函数去构造调用所需栈（stack）的信息，并调用该函数。

```
void *__builtin_apply(void (*func)(),void *arguments,int size);
```

第一个参数是函数的地址，作为函数的地址，它没有参数，也不会返回值。第二个参数是处理记录__builtin_apply_args() 的结果。size 参数为从当前堆栈框架（stack frame）复制到新的堆栈框架中的字节数，而且应该足够大，才能够包括随着返回地址一起返回的参数。

__builtin_apply()调用函数以后，返回值位于栈中。如下函数调整堆栈框架，并将结果返回给原始调用方。

```
__builtin_return(void *result);
```

下面的例子程序调用函数 passthrough()，它用到了内嵌函数来调用 average()函数，并且返回它的值：

```
/* args.c */
#include <stdio.h>
int passthrough();
int average();
int main(int argc,char *argv[])
{
   int result;
   result = passthrough(1,7,10);
```

```
    printf("result=%d\n",result);
    return(0);
}

int passthrough(int a,int b,int c)
{
    void *record;
    void *playback;
    void (* fn)() = (void (*)())average;

    record = __builtin_apply_args();
    playback = __builtin_apply(fn,record,128);
    __builtin_return(playback);
}

int average(int a,int b,int c) {
    return((a + b + c) / 3);
}
```

注意，函数 passthrough()只知道参数和返回值，因为它们自己调用自己。如果用省略号创建变长的参数列表，而返回值为空指针（void pointer），那样函数 passthrough()就可以转换为嵌套函数调用。函数 passthrough() 可以向只知道地址的函数传递参数集合，而返回值可以是任何类型。

要编写生成的嵌套时，必须保证传递给__builtin_apply()的 size 值足够大，能够包括实际传输的所有参数。

### 4.3.10 函数内嵌

被声明为内嵌函数的函数，在调用它的位置，代码会像宏一样被展开。通过关键字 inline，可将函数声明为内嵌函数，如下所示：

```
inline int halve(double x)
{
    return(x / 2.0);
}
```

如下是函数内嵌必须满足的规则和特征：

- 除非使用选项-O 指明优化级别，否则不会有函数会被真正地、内嵌式地展开。这样做是为了简化调试器的用法。如果函数的属性为 always_inline，它就会被迫展开为内嵌函数。
- 声明内嵌函数的结果可以使代码或者更大、或者更小，这依赖于函数的尺寸、建立调用框架的复杂程度以及函数调用的次数。

- 某些函数不能扩展为内嵌函数。其中有使用大量参数的函数、alloca、变长数组、非局部的跳转语句 goto、或者是嵌套的函数。当函数是递归函数，或在代码中有到地址的引用时，也会产生问题。当函数声明为内嵌函数，但又不能展开为内嵌函数的时候，命令行选项-Winline 会提交警告消息。
- 在 ISO C 程序中，可以使用关键字__inline__替换关键字 inline。
- 命令行选项-finline-functions 用于指示编译程序，自动选择适合的函数，可被扩展为内嵌函数。
- 如果内嵌函数未被声明为静态函数，编译程序就要确保函数体，因为要能够从其他模块调用该函数。将函数声明为内嵌和静态，就会引起同时发生的所有函数都被展开成内嵌函数，而并没有生成函数代码本身。命令行选项-fkeep-inline-functions 会忽略该行为，并创建函数体。
- 在头文件中定义函数，外部和内嵌函数的声明过程和声明宏几乎一样。函数另有一份备份，不是 extern 和 inline，可被编译并保存到库函数中，这样可以解析出非内嵌的引用。

### 4.3.11 函数名

标识符__FUNCTION__保存了函数显示的名字。按照字符串的文本格式，可和其他字符串连接起来，这种方式和__FILE__以及__LINE__一样。下面的例子构造了一个单行文本，包括三个位置宏：

```
char *here = "Line " __LINE__ " of " __FILE__ " in " __FUNCTION__;
```

ISO C99 标准定义标识符__func__包括当前函数的名字，但不同于 char 数组中的字符文本格式。

**注意** *__FUNCTION__的语义已经废止，并已改为和__func__语义相符。*

标识符__PRETTY_FUNCTION__和 C 语言中__FUNCTION__保存的名字相同，但在 C++中它们包含的名字却不相同。

### 4.3.12 函数嵌套

函数可以嵌入到其他函数中，而内部函数（inner function）只能被父函数调用。以下函数包括一个嵌套的函数 randint()，它返回一个指定范围内的伪随机整数（pseudo-random int）：

```
void rangers()
{
    int randint(int low,int high) {
        return(((int)random() % (high-low+1)) + low);
```

```
    }
    printf("0 to 100: %d\n",randint(0,100));
    printf("5 to 10: %d\n",randint(5,10));
    printf("-1 to 1: %d\n",randint(-1,1));
}
```

下面列出嵌套函数必须遵守的一些规则和特性：

- 在栈上创建嵌套函数的方式和创建变量的方式几乎一样，所以在第一次执行代码前，可将它声明为代码块——代码块的位置和声明局部变量的位置相同。
- 嵌套函数的地址不能返回给调用者，因为就像其他局部变量一样，在返回给父函数的时候，这些局部变量是不可见的。
- 嵌套函数不能在外部声明。
- 可以将嵌套函数的地址传递给其他函数，并由其他函数调用，就好像可以传递其他局部变量的地址一样。
- 嵌套函数和父函数可以访问同样的变量，但只能访问比嵌套函数声明早的局部变量。
- 嵌套函数可以使用 goto 语句跳转到函数之外的某个标号位置，当然该位置应该位于父函数内部。
- 通过将其声明为自动（auto），就可以声明嵌套函数的原型，如下例所示：

```
void right()
{
    auto double hypotenuse();
    double a = 3.0;
    double b = 4.0;

    double hypotenuse(double x,double y) {
        return(sqrt(x * x + y * y));
    }

    printf("Long side of %lf and %lf is %lf\n",
              a,b,hypotenuse(a,b));
}
```

### 4.3.13 函数原型

新的函数原型可以覆盖具有旧风格参数列表声明的函数定义，只要能够和旧风格参数的升级（promotion）相匹配就可以。例如，以下是可用的函数原型，因为在函数调用时，short 参数可以自动升级为整数 int：

```
int trigzed(int zvalue);
 . . .
int trigzed(zvalue)
short zvalue;
{
   return(zvalue == 0);
}
```

如果函数是按照新的语法 int trigzed(short zvalue)声明的，编译程序会产生出错信息，因为在 int 和 short 型之间会出现冲突。

### 4.3.14 函数返回地址和堆栈框架

以下内置函数遍历函数用到的地址，并返回给其调用方：

```
void *__builtin_return_address(unsigned int level);
```

指定 level 值为 0，遍历当前函数用到的地址并返回地址。level 值为 1 会遍历当前函数调用的函数用到的地址，而 level 值为 2 会遍历前面函数的地址，直至遍历了整个调用栈中的所有函数。函数__builtin_frame_address()可用于确定何时到达栈顶。而 level 必须指定为常数，不是变量。

在有些系统中，不可能遍历其他不是当前函数的函数地址。这样的系统中，遍历值或者为 0 或者为随机数（由平台决定）。

下面的函数遍历了函数堆栈框架中的地址：

```
void *__builtin_frame_address(unsigned int level);
```

指定 level 值为 0，遍历当前函数堆栈框架的地址，level 值为 1 会遍历当前函数所调函数的堆栈框架地址，level 值为 2 会遍历前面函数的地址，直至遍历了整个调用栈中的所有函数。函数__builtin_frame_ address()用于确定何时到达栈顶。

堆栈框架是一块内存，包括了调用函数保存的注册器（registers），以及传递给函数的参数值。由于调用约定以及平台有所不同，真正使用的格式是会改变的。

一些系统不可能遍历其他不是当前函数的堆栈框架地址。这样的系统中，遍历值为 0。如果堆栈框架遍历结束，也会返回 0。

### 4.3.15 标识符

标识符可以包含美元符号（dollar signs）。这和许多传统 C 编译程序都相兼容，并且很多 C 程序的变量和函数名都含有美元符号。但它不是对所有系统都合法，因为有些汇编程序不能接受它。

### 4.3.16 整数

C99标准定义的整数类型可以为64位长。作为功能扩展，GCC 在较早版本的C和C++中也支持它们。声明例子如下：

```
long long int a; // Signed 64-bit integer
unsigned long long int b; //Unsigned 64-bit integer
```

可以定义这些类型的常数，如下：

```
a = 855LL; // Signed 64-bit constant
b = 855ULL; // Unsigned 64-bit constant
```

所有机器都支持加法操作符、减法操作符以及按位布尔操作，对这些类型都可以执行这些操作。但乘法、除法和移位就不是所有硬件都支持的操作，在这些系统中会用到一些特定的库例程。

如果要用这些数据类型作为函数的参数，那么用原型定义参数类型就很重要。如果没有函数原型，调用堆栈框架中的变量尺寸和位置就会出错。

### 4.3.17 更换关键字

命令行选项-std和-ansi会使关键字asm、typeof和inline失去功能，但在这里可使用它们的替代形式__asm__、__typeof__和__inline__。

### 4.3.18 标识地址

可以使用标识来标记地址，将它保存在指针中，再用goto语句跳转到标记处。可通过&&操作符访问该地址。而对表达式得出的所有空指针，使用goto语句可进行跳转。

下面的简单例子解释如何实现这一点：

```
/* gotoaddr.c */
#include <stdio.h>
#include <time.h>
int main(int argc,char *argv[])
{
    void *target;
    time_t now;
    now = time((time_t *)NULL);
    if(now & 0x0001)
        target = &&oddtag;
    else
        target = &&eventag;

    goto *target;
```

```
    eventag:
      printf("The time value %ld is even\n",now);
      return(0);
    oddtag:
      printf("The time value %ld is odd\n",now);
      return(0);
  }
```

使用当前时刻作为伪随机数字，用于确定向 target 中保存哪个标识地址。然后 goto 语句可以接受空指针，并跳转到合法的目标地址。跳到其他函数是不合法的。

因为所有结果为空指针的表达式都可用于 goto 语句，所以可以创建一个标识地址数组，再根据索引进行跳转，如下：

```
void *loc[] = { &&label1, &&label2, &&label3, &&label4 };
    . . .
goto *loc[i];
```

### 4.3.19 局部标识声明

可以将标识声明为只在一定范围内有效。在范围开始的位置使用关键字__label__说明标识为局部标识，然后再在该范围内声明和使用标识。下面的程序展示了如何在范围内声明两个标识：

```
/* loclabel.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    int count = 0;
    {
        __label__ restart;
        __label__ finished;
    restart:
        printf("count=%d\n",count);
        if(count > 5)
            goto finished;
        count++;
        goto restart;
    finished:
        count += 10;
    }
    return(0);
}
```

可以一行声明一个标识，如上例所示，也可使用多个逗号在一行中定义多个标识，如下所示：

```
__label__ restart, finished;
```

在扩展宏中的代码时，这种标识很有用。因为通过使用大括号可以创建范围，所以函数内可以多次使用相同的标识。

### 4.3.20 左值表达式（lvalue）

赋值操作符的左边可使用复合表达式（就是说，左值表达式——lvalue），只要复合表达式产生的结果可以利用地址。通常左值的形式是特定变量的名字，如下例所示，这里变量 a 是内存中某个位置的名字，其中保存的值为 5：

```
a = 5;
```

左值的另一种形式会出现在语句右边，但这还是左值，因为这里是地址而不是值。下例中，变量 a 作为左值：

```
ptr = &a;
```

在一些指定环境下，复合表达式可用作一个左值。下面列出了创建左值表达式的规则和特点：

- 如果可以访问复合表达式最后一个成员变量的地址，就可将该复合表达式作为左值使用。例如，下面两条语句是一样的：

  ```
  (fn(), b) = 10;
  fn(), (b = 10);
  ```

- 可使用复合表达式的地址。该地址是复合表达式语句中最后一个成员变量的地址。下面语句中 b 的地址（就是 b 的左值）被保存在 ptr 中：

  ```
  ptr = &(fn(), b);
  ```

- 条件表达式也可用作左值，提供真/假选择。例如，下面语句在 a 大于 5 时，为 b 赋值 100，否则为 c 赋值 100：

  ```
  ((a > 5) ? b : c) = 100;
  ```

- 左值可被强制类型转换为其他类型。在下例中，字符指针 chptr 被强制转换为整数，保存了绝对地址 894：

  ```
  char *chptr;
  (int)chptr = 894;
  ```

### 4.3.21 具有可变参数的宏

共有两种技术可用于创建具有各种参数的宏，因为这种需求作为 GCC 提供的扩展功能，GCC 已经开发了一种技术来实现它，而 ISO 1999 年的标准又指定了其他技术来解决

这个问题。GCC 对两者都支持。实际这两种技术是一样的，但在语法上有些不同。

下面的例子用 ISO 标准方法创建宏，该宏具有各种参数：

```
#define errout(fmt,...) fprintf(stderr,fmt,__VA_ARGS__)
```

fmt 后指定的所有参数列表，在宏内部都会被__VA_ARGS__代替。使用 GNU 语法，同样的宏可以定义如下：

```
#define errout(fmt,args...) fprintf(stderr,fmt,args)
```

可在第 3 章找到更多关于宏的信息。

### 4.3.22 字符串

一行新的字符可以被嵌入到字符串中，而不需要使用转义符\n（escape character）。可以按照字面意思将它们包含在源码中。下面的两个字符串是等价的：

```
char *str1 = "A string on\ntwo lines"
char *str2 = "A string on
two lines"
```

'\e'字符是 ASCII ESC 字符。也可在字符串中顺序使用\e。

每行最后的反斜线总是能够将两行连成一行，如下所示：

```
char *str3 = "This string will \
be joined into one line.";
```

这里是标准 C。GNU 的扩展功能简化了该规则，原规则要求新一行的字符必须紧跟在转义字符反斜线之后。在 GCC 中，允许在反斜线之后存在任意数目的空白字符；在将两行连接成一行的时候，这些多余的空格会被去掉，但进行预处理时会提示警告消息。

### 4.3.23 指针算法

对空指针 void 和函数指针都支持加减运算。指针的递加/递减运算都是通过所指项目的尺寸决定的，而 GCC 对空指针和函数指针进行递加/递减运算的单位是 1。这是因为对空指针和函数指针的操作符的大小（sizeof）为 1。

可用选项-Wpointer-arith 发出警告消息，表示已经用到了扩展功能。

### 4.3.24 Switch/Case 分支语句

可在case语句中使用省略号指定变量的范围。下面语句中的4个可选值属于同一个case分支，这里是标准用法：

```
case 8:
case 9:
case 10:
```

```
case 11:
```

也可用三个点的省略号完成同样的事情，如下：

```
case 8 ... 10:
```

有一点很重要，就是在三个点前后要有空格，这样可以避免词法分析器产生混淆，与十进制常数的小数点加以区别。正是将每个分支都看作为惟一的常数，所以编译程序会检查是否出现重复值——下面两个冲突的 case 语句会产生出错信息：

```
case 8 ... 15:
 . . .
case 12 ... 32:   // Error
```

在字符常数范围内，该技术尤其有用：

```
case 'a' ... 'm':
```

### 4.3.25 typedef 名字创建

关键字 typedef 可用于为表达式的数据类型创建名字。定义的名字可用来声明变量或将变量强制类型转换为该表达式的数据类型：

```
typedef name = expression;
```

例如，下面语句定义了 smallreal 类型为单精度浮点类型，而 largereal 为双精度浮点类型（double type）：

```
typedef smallreal=0.0f;
typedef largereal=0.0;
```

这些新名字可用于声明变量。下面的语句就声明 real1 为单精度浮点类型，而 real2 是双精度浮点类型：

```
    smallreal real1;
    largereal real2;
```

这点在宏定义中很有用，使得一个宏可以作用于多种不同的数据类型。下面的宏无需预先假定参数类型，而只需将它们看作具有相同的类型并且能够封装自己的值就可以了：

```
#define swap(a,b)        \
    ({  typedef _tp=a;   \
        _tp temp = a;    \
        a = b;           \
        b = temp; })
```

第一个参数的数据类型被用于定义_tp，作为新的局部类型，它和第一个参数的类型相

同。再构造名为 temp 的局部变量作为临时存放变量的位置，这样可以交换两个变量的值，而不需要考虑它们的类型。

### 4.3.26 typeof 引用

关键字 typeof 可以声明类型表达式。用法类似于 sizeof，但是结果是类型而不是尺寸大小，如下例所示：

```
char *chptr; // A char pointer
typeof (*chptr) ch; // A char
typeof (ch) *chptr2; // A char pointer
typeof (chptr) chparray[10]; // Ten char pointers
typeof (*chptr) charray[10]; // Ten chars
typeof (ch) charray2[10]; // Ten chars
```

例子中，chptr 声明为字符串指针。使用 typeof 定义类型指向 chptr，ch 声明为字符类型。然后使用数据类型 ch 将 chptr2 定义为指向字符类型的指针。变量 chparray 声明为数组，包含 10 个字符类型指针。数组 charray 基于 chptr，而后者是 10 个字符类型的数组。还有一个 10 个字符类型数组基于 ch。

下面代码声明的变量是函数返回的类型：

```
char func();
typeof (func) retval;
```

函数 func()返回字符串，所以 typeof 表达式将 retval 声明为字符类型。

可在 typeof 表达式中直接使用类型名。例如，以下两种声明语句等价：

```
char *charptr;
typeof (char *) charptr;
```

利用 typeof 可以创建宏，用于声明变量。以下例子定义一个宏，然后用它创建两个数组，分别包含 10 个 double 和 10 个 float 变量：

```
#define array(type,size) typeof(type[size])
array(double,10) dblarray;
array(float,10) fltarray;
```

### 4.3.27 联合强制类型转换

如果数据项和联合中成员的类型相同，可以将该数据项强制类型转换为联合。例如，下面的程序将 double 数据项强制类型转换为联合，因为该联合包括 double 数据，所以可以通过联合访问每个字节：

```
/* unioncast.c */
#include <stdio.h>
```

```
union dparts {
    unsigned char byte[8];
    double dbl;
};

int main(int argc,char *argv[])
{
    int i;
    double value = 3.14159;

    for(i =0; i<8; i++) {
        printf("%02X ",((union dparts)value).byte[i]);
    }
    printf("\n");


    return(0);
}
```

联合的强制类型转换可用作调用函数的参数，如下：

```
void procun(union dparts);
 . . .
procun((union dparts)value);
```

对联合的强制类型转换与其他类型的强制转换有些区别。因为实际上它是构造器，所以它并不会创建左值。因此，以下语句为非法语句：

```
(union dparts)value.dbl = 1.2;    // Error
```

# 第 5 章

# 编译 C++程序

GNU 的 C++编译程序是一种功能强大的编译程序，它的输出就是生成的可执行目标代码。原始的 C++编译程序来自 AT&T，名为 cfront，实际上是个翻译器，可将 C++代码翻译为 C 代码，目前还有一些编译程序按照这种方式工作。GCC 编译程序开始时也只是 C 语言编译程序，C++的加入只是作为一种可选的编译模式。

## 5.1 编译基础

表 5-1 列出了编译和连接 C++程序用到的文件名后缀。完整的文件名后缀列表参见附录 D。

表 5-1 C++程序文件名后缀

| 后缀 | 文件内容 |
|---|---|
| .a | 静态对象库（文档文件） |
| .C, .c++, .cc, .cp, .cpp, .cxx | 需要预处理的 C++ 源代码 |
| .h | C 或 C++ 的头文件 |
| .ii | 不会被预处理的 C++ 源代码。该类文件是编译过程的中间产物 |
| .o | 目标文件，格式和应用的连接程序相符。该类文件是编译过程的中间产物 |
| .s | 汇编语言源代码。该类文件是编译过程的中间产物 |
| <none> | 标准 C++ 系统头文件，无后缀 |

### 5.1.1 单源文件到可执行文件

下面是一个简单的 C++程序源代码，保存在文件 helloworld.cpp 中：

```
/* helloworld.cpp */
#include <iostream>
int main(int argc,char *argv[])
```

```
{
    std::cout  "hello, world\n";
    return(0);
}
```

程序用到的 cout 定义在头文件 iostream 中，用于将简单字符串写入标准输出设备。可用以下命令将该程序编译成可执行代码：

```
$ g++ helloworld.cpp
```

g++编译程序按照文件名后缀识别出该文件是 C++语言源文件。默认操作是将源文件编译成目标文件，将目标文件和库 libstdc++中必要的函数连接起来，再生成可执行程序文件，最后删除目标文件。命令行未指定输出文件名，所以使用默认名 a.out。如下所示，可以运行该程序：

```
$ a.out
hello, world
```

一般都是用-o 命令来指定可执行文件的名字。如下命令会生成可执行文件 helloworld：

```
$ g++ helloworld.cpp -o helloworld
```

在命令行中键入程序名，可以运行该程序：

```
$ helloworld
hello, world
```

g++程序是 gcc 的一个特殊版本，它将默认编译的语言设置为 C++语言，这样就可以自动连接到标准 C++库，而不是默认使用的标准 C 库了。遵守源文件命名约定且指出库名，就可以用 gcc 编译和连接 C++程序了，如下例所示：

```
$ gcc helloworld.cpp -lstdc++  -o helloworld
```

选项-l(ell)可更改该选项后面紧跟的名字，使其前缀为 lib 而后缀为.a，将库命名为 libstdc++.a。再在标准位置查找该库。gcc 的编译过程和输出文件与 g++的都一样。

在大多数系统上安装 GCC，都会安装一个叫 c++的程序。安装好后，该程序和 g++完全一样，而且用法也一样，如下例所示：

```
$ c++ helloworld.cpp -o helloworld
```

### 5.1.2　多源文件到可执行文件

如果 g++命令连接了多个源文件，这些源文件是可以被编译并连接成一个可执行文件的。下面的头文件 speak.h 包含类的定义，其中只有一个函数：

```
/* speak.h */
#include <iostream>
class Speak
{
public:
    void sayHello(const char *);
};
```

下面列出的 speak.cpp 文件，包含 sayHello()函数体：

```
/* speak.cpp */
#include "speak.h"
void Speak::sayHello(const char *str)
{
    std::cout << "Hello " << str << "\n";
}
```

文件 hellospeak.cpp 包含的程序用到了 Speak 类：

```
/* hellospeak.cpp */
#include "speak.h"
int main(int argc,char *argv[])
{
    Speak speak;
    speak.sayHello("world");
    return(0);
}
```

可用一条命令将这两个源文件编译并连接成一个可执行程序：

```
$ g++ hellospeak.cpp speak.cpp -o hellospeak
```

### 5.1.3 源文件到目标文件

选项-c 可用于编译源文件，但会禁止连接操作，因此生成的是目标文件。生成目标文件的默认名和源文件名相同，只是后缀为.o。例如，下面的命令会编译源文件 hellospeak.cpp，并产生目标文件 hellospeak.o：

```
$ g++ -c hellospeak.cpp
```

g++命令也可识别出.o 文件，将其作为连接操作的输入文件。下面这些命令可将两个源文件编译成目标文件，再将这两个目标文件连接成一个可执行程序：

```
$ g++ -c hellospeak.cpp
$ g++ -c speak.cpp
$ g++ hellospeak.o speak.o -o hellospeak
```

选项-o 不仅可为可执行文件命名，也可用它为编译程序产生的其他文件命名。例如，下面命令生成的可执行文件和上面那些命令生成的可执行文件相同，只是过程中生成的目标文件名有所不同：

```
$ g++ -c hellospeak.cpp -o hspk1.o
$ g++ -c speak.cpp -o hspk2.o
$ g++ hspk1.o hspk2.o -o hellospeak
```

### 5.1.4　预处理

使用选项-E 可向 g++明确指出，将源代码传递给预处理程序而不采取其他动作。如下命令预处理 helloworld.cpp 的源代码，并将结果写到标准输出上：

```
$ g++ -E helloworld.cpp
```

本章较早所示 helloworld.cpp 的源码只有 6 行，除了显示一条文本之外没做其他什么事情，但是预处理后的版本却超过 1 200 行。主要是因为它加上了 iostream 头文件以及其他一些头文件，其中定义了某些较大的类用来处理输入和输出。

利用-o 选项，可令预处理后的 C++代码产生的文件使用 GCC 后缀 .ii，如下：

```
$ gcc -E helloworld.cpp -o helloworld.ii
```

### 5.1.5　生成汇编语言

选项-S 指示编译程序将程序编译成汇编语言，输出汇编语言源码，然后停止。下面的命令会由 C++源文件生成汇编语言文件 helloworld.s：

```
$ g++ -S helloworld.cpp
```

生成的汇编语言依赖于编译程序的目标平台（target platform），但如果检查它会发现，对于 C++程序中的继承和连接而言，不只是可执行代码和数据存储声明，甚至地址表（tables of addresses）都是必要的。

### 5.1.6　创建静态库

静态库是文档文件（archive file），其中包含编译程序产生的目标文件集合。库成员可包括常规函数、类的定义和所定义类的实例。事实上所有可保存在.o 目标文件中的内容都可以保存在库中。

下面的例子先创建两个对象模块，再用它们创建一个静态库。头文件包含的信息对于某些程序是必须的，它们用到了库中保存的函数、定义的类以及对象。

头文件 say.h 包括函数 sayHello()的原型，以及名为 Say 的类：

```
/* say.h */
#include <iostream>
```

```
void sayhello(void);
class Say {
private:
    char *string;
public:
    Say(char *str)
    {
        string = str;
    }
    void sayThis(const char *str)
    {
        std::cout << str << " from a static library\n";
    }
    void sayString(void);
};
```

下面的源文件名为 say.cpp，为库中插入的两个目标文件之一的源文件。它包含类 Say 的函数 sayString()的定义，还声明了 Say 类的实例 librarysay：

```
/* say.cpp */
#include "say.h"
void Say::sayString()
{
    std::cout << string << "\n";
}
Say librarysay("Library instance of Say");
```

源文件 sayhello.cpp 是库中第二个模块的源代码。包括函数 sayhello()的定义，如下：

```
/* sayhello.cpp */
#include "say.h"
void sayhello()
{
    std::cout << "hello from a static library\n";
}
```

下面的命令将两个源文件编译成一个目标文件，而 ar 命令会将它们保存为库：

```
$ g++ -c sayhello.cpp
$ g++ -c say.cpp
$ ar -r libsay.a sayhello.o say.o
```

工具 ar 使用选项-r 会生成新的库 libsay.a，并将所列目标文件插入该库。如果不存在库，用这种方法 ar 会创建该库，而如果库已经存在，就会用最新版本取代现有的对象模块。

下面是程序 saymain.cpp 的主体，它用到了 libsay.a 库中的代码：

```
/* saymain.cpp */
#include "say.h"
int main(int argc,char *argv[])
{
    extern Say librarysay;
    Say localsay = Say("Local instance of Say");
    sayhello();
    librarysay.sayThis("howdy");
    librarysay.sayString();
    localsay.sayString();

    return(0);
}
```

用下面的命令编译并生成程序，这里g++通过查找库libsay.a，就可以解析出saymain.cpp中所有的引用。

```
$ g++ saymain.cpp libsay.a -o saymain
```

librarysay 的外部引用是指对 say.cpp 声明的保存在库中的对象的引用。librarysay.sayThis()和 librarysay.sayString()都是对库中对象的方法的调用。并且 sayhello()是对 sayhello.o 中函数的调用，它也保存在库中。程序一运行，就会产生下面这些输出：

```
hello from a static library
howdy from a static library
Library instance of Say
Local instance of Say
```

### 5.1.7 创建共享库

共享库指的是包含目标文件的档案，但其中的目标文件必须使用相对地址，这样代码就可被加载到内存的任意位置，不需要额外的重新定位操作就可以运行。允许在程序运行的时候，由共享库加载代码，而不用由连接程序（linker）直接连接到可执行程序上。

下面的头文件 average.h 定义了库中保存的类：

```
/* average.h */
class Average {
private:
    int count;
    double total;
public:
    Average(void) {
        count = 0;
        total = 0.0;
    }
```

```
    void insertValue(double value);
    int getCount(void);
    double getTotal(void);
    double getAverage(void);
};
```

被编译并保存为共享库的源文件，包含了类中定义的函数体：

```
/* average.cpp */
#include "average.h"
void Average::insertValue(double value)
{
    count++;
    total += value;
}
int Average::getCount()
{
    return(count);
}
double Average::getTotal()
{
    return(total);
}
double Average::getAverage()
{
    return(total / (double)count);
}
```

以下两条命令首先将源文件编译成目标文件，再用它创建库：

```
$ g++ -c -fpic average.cpp
$ gcc -shared average.o -o average.so
```

第一条命令使用选项-c 令编译程序产生目标文件 average.o，同时未将它连接到可执行程序。选项-fpic（位置独立代码，position independent code）指示编译程序生成的代码要适合共享库的内容——这样的代码能够根据载入内存的位置计算内部地址。第二条命令通过选项-shared 创建共享库，并用-o 选项指明生成的共享库为 average.so。第二条命令完全可以用 g++取代 gcc，因为生成共享库对 C++而言并没有任何特别要求。创建包含多个对象模块的共享库只需将所有目标文件列在同一条命令中就可以了。

前面两条命令可以组合成一条命令，将源文件编译成目标文件，并用它们创建共享库：

```
$ g++ -fpic -shared average.cpp -o average.so
```

以下程序用到共享库中定义的类来实例化对象，该对象用于保存 4 个数值并返回其平均值：

```
/* showaverage.cpp */
#include <iostream>
#include "average.h"
int main(int argc,char *argv[])
{
    Average avg;
    avg.insertValue(30.2);
    avg.insertValue(88.8);
    avg.insertValue(3.002);
    avg.insertValue(11.0);
    std::cout << "Average=" << avg.getAverage() << "\n";
    return(0);
}
```

下面命令编译程序，并将程序连接到共享库，生成可执行程序 showaverage：

```
$ g++ showaverage.cpp average.so -o showaverage
```

为运行该程序，需在目录中安装共享库，以保证程序运行时可以找到该共享库，详细讨论请见第 12 章。

## 5.2　C++语言的扩展功能

本章描述了 GNU 支持的一些 C++语言的扩展功能。C++编译程序非常复杂，而且标准定义文档（standard definition document）非常大，因此还有很多扩展功能和区别没能列出。而且因为 C++编译程序的代码和 C 编译程序的代码在很大程度上彼此共享，所以第 4 章所列 C 编译程序的扩展功能对 C++编译程序也适用。

### 5.2.1　属性

第 4 章列出了 C 中可用的属性。那些属性也可用于 C++程序，这里列出的属性只能用于 C++程序。通过关键字__attribute__使用属性，属性是包在圆括号内的。表 5-2 包含的属性都是 C++的专用属性。

**表 5-2　为 C++语言定义的属性**

| 属性 | 描述 |
|---|---|
| init_priority | 标准 C++指出按照对象在编译单元中出现的顺序初始化，但没有指出编译单元之间的顺序。属性 init_priority 通过为对象赋优先权值，可以指定在所给命名空间中初始化对象的顺序。优先值是数字，数字小的优先权高于数字大的。例如，下面的三个对象初始化顺序为 B，然后是 C，然后才是 A，而无论发现它们的源模块出现的顺序： |

（续表）

| 属性 | 描述 |
| --- | --- |
| | SpoClass A __attribute__ ((init_priority(680)));<br>SpoClass B __attribute__ ((init_priority(220)));<br>SpoClass C __attribute__ ((init_priority(400)));<br>用到的值没有什么特殊意义，只是它们彼此间的关系才有用。 |
| java_interface | 该属性指出该类定义为 Java 接口。只能用于 extern "Java"块内部定义的类。这种方式定义的类的方法调用使用 GCJ 接口表而不是 C++虚拟表。 |

### 5.2.2 头文件

所有系统头文件会默认包含，就好像它们已被封装在 extern "C" { ... }块中一样。因此如果系统头文件中有 C++代码，就会出问题，但可用下面的 pragma 解决此问题：

```
#pragma cplusplus
```

如果头文件中有此pragma，就会将文件中剩余代码当作 extern "C++" { ... }块进行编译。在显式的 extern "C" { ... }块中使用 pragma 会出错。

### 5.2.3 函数名

C 和 C++都用标识符__FUNCTION__保存当前函数名。C++中，标识符__PRETTY_FUNCTION__也保存了函数名，但它所带的信息稍微多一点。下面例子展示这些标识符的用法，同时列举了 C 标准中的标识符__func__：

```
/* showfuncname.cpp */
#include <iostream>
class Xyz
{
public:
    void NameShow(int i,double d)
    {
        std::cout << "__FUNCTION__\n"
              << __FUNCTION__ << "\n";
        std::cout  "__PRETTY_FUNCTION__\n"
              << __PRETTY_FUNCTION__ << "\n";
        std::cout << "__func__\n"
              << __func__ << "\n";
    }
};

int main(int argc,char *argv[])
{
```

```
    Xyz xyz;
    xyz.NameShow(5,5.0);
    return(0);
}
```

运行该程序的输出结果如下：

```
__FUNCTION__
    NameShow
__PRETTY_FUNCTION__
    void Xyz::NameShow (int, double)
__func__
    NameShow
```

标识符__FUNCTION__ 和__func__都定义为串，包含的是当前函数的简单函数名。而标识符__PRETTY_FUCNTION__包含了完整的函数名，包括返回值类型、类名以及参数类型列表。

### 5.2.4 接口与实现

类的接口与实现可以合而为一。这就是说，不需单独维护类的原型定义，因为完全实现类的代码本身就可以作为类的接口定义。

通过#pragma interface 说明类的定义只能用作接口的定义，而用#pragma implementation 告诉 GCC 要将类的函数和数据编译成目标代码。

**注意** 这是个很方便的功能，但它会有改动。GCC 将来的版本很可能不再使用这对 pragma，而用其他机制达到同样效果。

要实现这对 pragma，需要遵守下面的步骤：

1. 创建头文件，其中包含了完整的类的实现。例如，类 MaxHolder 的头文件可叫做 maxholder.h。
2. 在头文件内部，类定义之前，插入如下一行代码：

   ```
   #pragma interface
   ```

3. 在任何引用 MaxHolder 类的源文件中，按照通常形式包含头文件。
4. 在源文件（通常为程序的主线）的#include 指示字（directive）前，插入如下#pragma 指示字：

   ```
   #pragma implementation "maxholder.h"
   #include "maxholder.h"
   ```

按照通常方式包含头文件的文件只会含有类的接口定义，而使用#pragma implemention 指示字的源文件会包含完整的类的定义，可被编译成目标代码。就是说内嵌函数、调试信

息以及实现虚函数的内部表都只有一个备份。

- 如果头文件和实现文件的基本名相同，在 pragma 中就不需指定文件名。例如，如果文件名为 maxholder.cpp，包括头文件为 maxholder.h，那么 pragma 就可以简单写成#pragma implementation。
- 如果头文件包括其他目录中的头文件，可在 pragma 接口中按照#pragma interface "*subdirectory/filename.h*"这种形式命名。如果要在这里这样命名，就必须保证同样的文件名出现在实现 pragma 中。
- #include 语句用于包含头文件，因为 pragma 不包含这些头文件。
- pragma 接口对类中函数的作用是它们都被声明为 extern。函数体只在代码内嵌展开的时候被使用一次。
- 使用#pragma implementation 会令所有内嵌函数（inline function）都编译为非内嵌版本（non-inline function），以防其他模块中的某些函数不是内嵌函数。可通过命令行选项-fno-implement-inlines 禁止该操作。

### 5.2.5 操作符<?和>?

这种特殊操作符对返回两个参数中的最小值是很有用的。下面的表达式返回 a 和 b 中的最小值：

```
minvalue = a <? b;
```

与此相似，下面的表达式的结果是两个值中的最大值：

```
maxvalue = a >? b;
```

这两个操作符在语言中都是最基本的，所以使用它们不会有任何副作用。下面语句的结果是 x 和 y 中的最小值，然后再为它们分别加 1。

```
int minvalue = x++ ? y++;
```

操作符>?和<?是可重载的，如下例所示，类 Iholder 中定义了操作符>?，用于返回包括最大 int 值的对象。而 Iholder 类在内部也用操作符>?比较两个 int 值：

```
/* minmax.cpp */
#include <iostream>
class Iholder
{
    friend Iholder operator>?(Iholder&,Iholder&);
protected:
    int value;
public:
    Iholder(int v)
```

```
    {
        value = v;
    }
    int getValue(void)
    {
        return(value);
    }
};

Iholder operator>?(Iholder& ih1,Iholder& ih2)
{
    return(Iholder(ih1.getValue()>? ih2.getValue()));
}

int main(int argc,char *argv[])
{
    Iholder ih1 = Iholder(44);
    Iholder ih2 = Iholder(34);
    Iholder imax = ih1 >? ih2;
    std::cout << "The maximum is " << imax.getValue() << "\n";
    return(0);
}
```

### 5.2.6 约束

C99 标准中 C 语言的关键字 restrict 被 C++标准委员会弃用了，但是 GCC 用关键字__restrict__实现它的功能。所有声明为__restrict__的指针都保证能够独占访问内存中它所指向的位置。实事上，编译程序能够确保不存在到内存的别名引用（alias references）意味着可以生成更高效的代码。

关键字__restrict__可被用作限定词（qualifier），例如 const 或 volatile，如下所示：

```
double *__restrict__ avg;
```

- 关键字__restrict__只对指针和引用有效。不像 const 或 volatile，限定词__restrict__只应用于指针，不会用于定位的数据。
- 函数指针参数可被标识为已经过约束。下面例子中的函数确保指针 bp1 和 bp2 不会交叉。

```
void copy(char *__restrict__ bp1,
          char *__restrict__ bp2, int size) {
   for(int i=0; isize; i++)
      bp1[i] = bp2[i];
}
```

- 函数引用参数可用与指针相同的语法加以约束，如下例所示：

```
void icopy(int &__restrict__ ip1,
           int &__restrict__ ip2) {
   ip1 = ip2;
}
```

- 在函数匹配中关键字__restrict__会被忽略，所以原型中关键字__restrict__不是必需的。
- 在声明成员函数的时候，指针 this 可用关键字__restrict__加以约束，如下：

```
void T::fnctn() __ restrict__ { ... }
```

## 5.3 编译程序操作

本节会描述一些在某些特定环境下需要注意的 C++编译程序的内部操作。通常只需使用 g++命令就可编译并连接 C++程序，但还需了解一些内部操作才能处理某些特殊情况。

### 5.3.1 库

标准的 C++库名为 libstdc++.a，包括所有标准的 C++例程（routine）。尽管一般不会考虑程序的大小问题，但因为库很大，所以静态连接的 C++程序会包含很多不会用到的例程。这样的结果就是，虽然只需要库中某个目标文件的一个例程，程序却需将整个目标文件都连接进来。

如果需要静态连接程序且不使用库的例程，实际上可以连到 libsupc++.a，其中只包含基础语言定义（fundamental language definition）中规定的部分例程。要实现这样的改动，只需要执行 g++命令时指定库名为-lsupc。

### 5.3.2 mangling 名字

C++成员函数和 Java 的方法都可通过对参数列表指定不同数据类型加以重载。例如，下面三行代码是函数不同的原型：

```
int *cold(long);
int *cold(struct schold *);
int *cold(long, char *);
```

编译程序可以根据参数类型确定调用的是哪个函数，不会有问题。惟一的问题来自连接程序（linker），因为连接程序是将模块所引用的名字和在其他模块中定义的名字绑定在一起的，不考虑它们的类型。解决方案就是由编译程序改动函数名，这样就不会丢失参数信息，而连接程序也能和它们相匹配。改动名字的处理过程就叫 mangling。

mangling 后的名字照顺序由下面的内容组成：

1. 函数的基本名。
2. 两个下划线字符。
3. 代码列表，长度可能为 0，指向某种限定词，如 const。
4. 类名的字符数，函数就是该类的成员。
5. 类名。
6. 一列代码，指出参数的数据类型。

例如，函数 void cname::fname(void) 被编码为 fname__5cname。函数 int cname::stname(long frim) const 被编码为 stname__C5cnamel，这里的 C 指出该函数是常数 const，而结尾的 l (ell)说明有一个类型为 long 的参数。构造函数被编码为不含函数名。例如，构造函数 cname::cname(signed char)编码为__5cnameSc，这里的 Sc 指出参数是符号字符 signed char。

表 5-3 列举了各种类型和限定词的代码。这些代码的意义取决于它们在编码串中的用法，根据表中所列条目，实践一下就能把目标文件中的名字 demangle，能与源文件中的名字匹配得很好。

**表 5-3　mangling 名字中用到的代码字母**

| 代码字母 | 意义 |
|---|---|
| <number> | 定制的数据类型名中的字符数目。例如，函数 Mph::pdq(char, drip, double) 编码为 pdq__3Mphc4dripd。可选地，字母 G 可以优先于数字——就是说，pdq__3Mph4drip 等于 pdq_3MphG4drip |
| A | 数组。C++中，数组总是比指针落后，所以实际上根本看不到该类型。Java 中，数组是被编码成到 JArray 类型的指针 |
| b | 布尔数据类型，C++中的 bool 或 Java 中的 boolean |
| c | C++中的 char 数据类型，或 Java 中的 byte 数据类型 |
| C | 修改器（modifier）指明的 const 参数类型或成员函数 |
| d | double 数据类型。 |
| e | 未知类型的额外参数。例如，函数 Mph::pdq(int,...) 编码为 pdq__3Mphie |
| f | float 数据类型 |
| G | 见<number> |
| H | 模板函数 |
| i | int 数据类型 |

（续表）

| 代码字母 | 意义 |
|---|---|
| I | 特殊整数类型，所含位数不标准。例如，函数 Mph::pdq(int, int60_t, char) 将 60 位整数作为它第二个参数类型，会被编码为 pdq__3MphiI_3C_c。由下划线包围的一个十六进制数字用于指出整数的位数。只要用于区分的字符明确，该十六进制数字不必非得使用下划线加以限制 |
| J | C++的复杂类型 complex |
| l (ell) | C++的长类型 long |
| L | 局部类名 |
| p | 指针。总是跟着某个指针类型，与 P 相同 |
| P | 指针。总是跟着某个指针类型，与 p 相同 |
| Q | 合格的名字，例如由嵌套类产生 |
| r | C++的数据类型 long double |
| R | C++引用。总是跟着某个引用的类型。例如，函数 Mph::pdq(ostream&) 编码为 pdq__3MphR7ostream |
| s | 数据类型 short |
| S | 如果 S 用于类名字前，意味着是静态变量 static。例如，Mph::pdq(void) static 被编码为 pdq__S3Mph。如果 S 用于字符类型 char 前，意味着是 signed。例如，函数 Mph::pdq(signed char)编码为 pdq__3MphSc |
| t | C++的模板实例化（template instantiation） |
| T | 对以前看到的参数类型的向后引用 |
| u | 受限指针（restricted pointer）的类型限定词 |
| U | 修改器（modifier）指明的无符号整数类型。也可用于类或名字空间的修改器，用于指明 Unicode 编码 |
| v | 数据类型 void |
| V | 修改器指向 volatile 数据类型 |
| w | C++的类型 whcar_t 或 Java 的 char 类型 |
| x | C++的类型 long long 或 Java 的 long 类型 |
| X | 模板类型参数 |
| Y | 模板常数参数 |
| none | 标准 C++ 系统头文件，无后缀 |

叫做 c++filt 的 demangle 程序是 binutils 包中的一部分。可由命令行键入 mangled 后的名字，它会解析出 demangled 后的名字，如下例所示：

```
$ c++filt pdq__3MphiUsJde
Mph::pdq(int, unsigned short, __complex double, ...)
```

c++filt 工具可以 demangling 多种模式（scheme）。模式根据平台有所不同，也根据语言存在彼此差别。可通过-s 设置的选项是 lucid、arm、hp 和 edg。语言-s 选项中的两种是 Java 的 java 和 Ada 的 gnat。

**注意** GCC 为 C++和 Java 使用 mangling 方案，它们之间彼此兼容，但和其他编译程序不兼容。每种编译程序在安置类、实现多继承和处理虚函数上，都有自己的解决方案。如果使用兼容的 mangling 方案，是可能将 GCC 对象和其他编译程序产生的模块和库连接在一起的，但程序仍然不能运行。

### 5.3.3 连接

严格说来，目标文件中有些东西是不会执行的，但它们对优化代码和解析引用而言却非常重要。这类信息被划分为模糊连接（vague linkage）信息，因为它们不是普通简单的与特定名字、特定地址相关的处理过程。下面描述了 C++的模糊连接项目。

#### 虚函数表

虚表是类中虚函数地址的列表。如果类 A 包含虚函数，而函数被子类 B 重写了，那么新函数的地址就会取代虚函数表或 vtable 中的函数地址。这是由多态（polymorphism）的需求决定的——如果类 B 被强制类型转换作为类 A 的对象，那么到虚函数的调用就会用到该表，实际上就是对 B 中函数的调用，而不是调用 A 中的函数。

#### 运行时类型标识

C++中的每个对象都包含标识信息，用于实现 dynamic_cast、typeid 以及异常处理。具有虚函数的类，信息包含在表 vtable 中，这样可在运行时通过 dynamic_cast 确定类型。如果没有 vtable（就是说，该类不是多态类），信息只包含在实际使用的目标代码中（在 typeid 语句中或在抛出异常处理的位置）。

#### COMDAT

只要在头文件中声明，就会为所有包含该头文件的编译单元生成的代码中加入头文件副本。这样会令全局变量声明以及成员函数都成为类定义中的一部分。如果系统支持（GNU 连接程序在 ELF 系统中，例如 Linux 或 Solaris、微软的 Windows 以及其他系统），连接程序会去掉最终可执行程序中的代码副本。

在连接程序的相关文档中，会看到该功能可参考 folding、comdat folding、identical comdat folding、comdat discarding 或 transitive comdat elimination。

#### 内嵌函数

通常在头文件中声明内嵌函数，需要使用此函数的所有模块都要包含该头文件。尽管

被声明为内嵌函数 inline，还是会创建此函数的实例，以防某种情况下不能展开此内嵌函数，例如在用到地址的时候。

### 5.3.4 编译模板实例化

如果头文件中有模板的定义而且该头文件被多处包含，就会为该模板创建多个编译好的副本。这样做当然可行，但在含有多个模板的大程序中，所有目标文件都会包含每个编译好的模板副本。这会令编译时间变得很长，而且目标文件也会很大。这里是一些可选的其他方法：

- 在源文件中使用指示字#pragma interface 和 #pragma implementation（本章较早有所阐述），就只会创建一个编译后模板。
- 与使用两个 pragma 相似，使用命令行选项-falt-external-templates 编译所有源文件。这会告诉编译程序只有在真正用到模块的时候才将编译后的模板实例包含进来。这种方法的一个重要特征就是要求对于每个用到头文件的模块，头文件都必须相同。
- 用命令行选项-frepo 编译代码。这样创建的文件后缀为.rpo，每个.rpo 文件都列出了对应目标文件中的模板实例。然后，会激活连接打包工具 collect2 更新.rpo 文件，指示连接程序替换最后程序中的模板实例。这种方法惟一只在处理库的时候才有困难——除非相关的.rpo 文件都存在，否则连接库中的模板实例就会失败。
- 用-fno-implicit-templates 编译代码，会令隐式的模板实例化失效，它会显式地初始化所需模板。虽然这种方法需要精确了解正在使用的是哪种模板实例，但这种方法确实令源代码更明确也更清楚。

# 第 6 章

# 编译 Objective-C

Objective-C(对象-C)是 C 语言的扩充，但它支持类；或者可以将它看作是 C 和 Smalltalk 相混合的语言。它比 C++要简单。用 GCC 编译 Objective-C 程序和编译标准 C 一样，只是增加了一些工具用于定义类、使用类来实例化对象以及发送对象的消息（调用函数）。向对象发送消息的语法与 Smalltalk 的语法很相似。

与 GCC 编译的其他语言不同，Objective-C 没有标准定义。Objective-C 的 GCC 实现和由 NeXTStep 系统开发并使用的 Objective-C 程序非常相似。

## 6.1 编译基础

表 6-1 列出编译和连接 Objective-C 程序所用的文件名后缀。附录 D 中的表列出了 GCC 可识别的所有文件名后缀。

表 6-1 Objective-C 程序文件名后缀

| 后缀 | 文件内容 |
|---|---|
| .a | 包含了用于静态连接的目标文件的库(文档文件) |
| .h | 头文件 |
| .m | 需要预处理的 Objective-C 源文件 |
| .mi | 不需要预处理的 Objective-C 源文件 |
| .o | 适用于连接的目标文件格式 |
| .so | 包含了用于动态连接的目标文件的库 |

### 6.1.1 单独源文件到可执行程序

从各种角度上看，编写 Objective-C 程序都与编写 C 程序非常相似。就是说可以编写不含对象的 Objective-C 程序，它与 C 程序的语法和形式完全一样。下面是个语法简单的例子程序，可作为 Objective-C 程序进行编译并运行：

```
/* helloworld.m */
#import <stdio.h>
int main(int argc,char *argv[])
{
   printf("hello, world\n");
   return(0);
}
```

本程序和 C 程序完全一样，只是用预处理指示字（preprocessor directive）#import 取代了关键字#include。这两个指示字作用相同，但是#import 不仅可用来包含头文件，而且在编辑相同的单元时不会被多次包括进来。利用指示字#include 达到同样的功能，需要在头文件中使用条件编译，这在第 3 章中有所描述。而应用可以使用两种方法中的任何一种。

该程序可用如下命令进行编译：

```
$ gcc -Wno-import helloworld.m -lobjc -o helloworld
```

选项-Wno-import 用于禁止警告消息，这里的程序需要在头文件中使用指示字#import，而不是#include。因为有 GCC 源，可以修改 cppinit.c 文件中的命令行选项的默认设置，删除下面这一行：

```
CPP_OPTION (Pfile, warn_import) = 1;
```

选项-lobjc 指出使用的库为 libobjc.a （这是 Objective-C 的对象库），但实际上对于没有对象的简单程序，该选项并不是必须的。编译程序是通过文件名中的.m 后缀将源文件识别为 Objective-C 程序的，而选项-o 用于指定输出文件的名字。默认的输出文件名为 a.out。

### 6.1.2 编译对象

类的定义由两个源文件组成。设计 Objective-C 语言用.h 头文件包含类的接口定义，而用.m 源文件包含类的方法实现。下面的例子用头文件 Speak.h 定义 Speak 类的接口，该类能够保存内部字符串，并按要求将它显示到标准输出上：

```
/* Speak.h */
#import <objc/Object.h>
@interface Speak : Object
{
   char *string;
}
- setString: (char *) str;
- say;
- free;
@end
```

指示字#import 用于读取头文件 Object.h，其中包括类 Object 的定义。Object 类是所有

Objective-C 类的超类（super class）。Speak 类的定义是由编译程序指示字@interface 和@end 包围的。定义内部是由括号包围的代码块，其中保存的是数据定义（本例中，惟一的数据是指向字符串的指针）。数据块后面跟随的是类中定义的方法。每个方法都用减号说明，方法的名字以及参数类型的列表（如果存在参数）都会传递给相关函数。

文件 Speak.m 定义类 Speak 的方法体，如下所示：

```
/* Speak.m */
#import "Speak.h"
@implementation Speak

+ new
{
    self = [super new];
    [self setString: ""];
    return self;
}
- setString: (char *)str
{
    string = str;
    return self;
}
- say
{
    printf("%s\n",string);
    return self;
}
- free
{
    return [super free];
}
```

导入头文件 Speak.h 后，所有数据和方法的定义才是可用的。编译程序指示字@implementation 说明源文件包含的是 Speak 类方法的实现。由减号开头的方法体定义是实例方法，只能被已经存在的对象调用；而在任何时候都可以调用加号开头的类变量。

方法的定义和头文件中的相匹配，方法体的实现由括号包围。除了某些方法需要返回特殊数据类型，会假设其他的返回类型为 id（该数据类型代表 Objective-C 对象的版本）。因此大多数方法会返回自身 self，这是对象自己引用自己的一种方法。

下面的程序用 Speak 对象将"hello, world"字符串写到标准输出设备：

```
/* helloobject.m */
#import <objc/Object.h>
#import "Speak.h"

main()
{
```

```
    id speak;

    speak = [Speak new];
    [speak setString: "hello, world"];
    [speak say];
    [speak free];
}
```

可将多个源文件编译成目标文件，再将它们连接成一个程序，如下：

```
$ gcc -Wno-import -c helloobject.m -o helloobject.o
$ gcc -Wno-import -c Speak.m -o Speak.o
$ gcc -helloobject.o Speak.o -lobjc -o helloobject
```

另一种可选方法，即由如下一条命令代替这三步：

```
$ gcc -Wno-import helloobject.m Speak.m -lobjc -o helloobject
```

### 6.1.3 创建静态连接库

编译 Objective-C 产生的.o 目标文件集可保存在目标文件库（文档 archive）中。下面的例子创建了库 libcat.a，包括了类 Cat 的实现代码。类中包含的方法可接收多个字符串，并将它们串联成一个字符串。

文件 Cat.h 是头文件，定义了 Cat 类的接口：

```
/* Cat.h */
#import <objc/Object.h>
@interface Cat : Object
{
    char *string;
}
- add: (char *) str;
- (char *) get;
- init;
- free;
@end
```

文件 Cat.m 包括类 Cat 的实现。方法 add 可在字符串的结尾添加字符，而 get 方法会返回当前联接的字符串。方法 init 只在创建新的 Cat 对象的时候才会被调用。

```
/* Cat.m */
#import "Cat.h"
@implementation Cat

+ new
{
    self = [super new];
```

```
    [self init];
    return self;
}
- init
{
    string = NULL;
    return self;
}
- add: (char *)str
{
    int length;
    char *newstring;
    if(string == NULL) {
        length = strlen(str) + 1;
        string = (char *)malloc(length);
        strcpy(string,str);
    } else {
        length = strlen(str) + strlen(string) + 1;
        newstring = (char *)malloc(length);
        strcpy(newstring,string);
        strcat(newstring,str);
        free(string);
        string = newstring;
    }
    return self;
}
- (char *) get
{
    return string;
}
- free
{
    if(string != NULL)
        free(string);
    return [super free];
}
```

用下面的命令可将文件 Cat.m 编译成目标文件 Cat.o：

```
$ gcc -c -Wno-import Cat.m -o Cat.o
```

用下面的命令可将目标文件构造成库：

```
$ ar -r libcat.a Cat.o
```

选项-r 将现存版本的目标文件替换为新版本的目标文件，或在库文件不存在的情况下创建一个完整的库文件。

下面的例子程序使用 Cat 类将两个字符串联接成一个，然后得到该结果并显示它：

```
/* docat.m */
#import <objc/Object.h>
#import "Cat.h"

main()
{
    id cat;
    char *line;

    cat = [Cat new];
    [cat add: "Part one"];
    [cat add: " and part two"];
    line = [cat get];
    printf("%s\n",line);
}
```

下面的命令可将程序编译成一个可执行的 docat：

```
$ gcc -Wno-import docat.m libcat.a -libobjc -o docat
```

### 6.1.4 创建共享库

可将 Objective-C 程序编译成目标文件并保存在共享库中。构造共享库，需要将源编译成可载入内存任意位置的目标代码，并且能够从载入位置执行该目标代码。为此，必须在命令行中明确指出选项-fpic（位置独立代码）。如下命令行会为 Cat.m 文件中定义的类创建目标文件：

```
$ gcc -fpic -Wno-import -c Cat.m -o Cat.o
```

下面的命令行利用目标文件创建共享库：

```
$ gcc -shared Cat.o -o cat.so
```

这两条命令可以组合成一条命令，可由源文件直接生成共享库文件，如下：

```
$ gcc -Wno-import -fpic -shared Cat.m -o cat.so
```

下面的程序用 Cat 类的实例将三个字符串联接成一个字符串，并显示该结果：

```
/* showcat.m */
#import <objc/Object.h>
#import "Cat.h"
main()
```

```
{
    id cat;
    char *line;

    cat = [Cat new];
    [cat add: "The beginning"];
    [cat add: ", the middle"];
    [cat add: ", and the end."];
    line = [cat get];
    printf("%s\n",line);
}
```

下面的命令会编译程序 showcat.m 并连接该程序，这样就可以使用共享库了：

```
$ gcc -Wno-import showcat.m cat.so -lobjc -o showcat
```

为运行依赖于共享库的应用程序，该应用程序需要能够找到共享库，这会在第 12 章加以讨论。

## 6.2 Objective-C 的简要介绍

Objective-C 没有标准说明什么是不可以包含进来的。在编写 Objective-C 代码的时候，不要期望能够将该程序移植到其他编译程序。本节列出的有关项目都是 GCC 版本的 Objective-C 所特有的，不属于其他任何版本。因为 GCC 的 Objective-C 编译程序和完整的标准 C 编译程序是建立在一起的，因此会看到标准 C 和预处理程序的所有功能都是可用的。

### 6.2.1 预定义类型

表 6-2 列出头文件 Object.h 中定义的数据类型。大多数 Objective-C 编译程序都有相似的类型，但名字也许有些区别。

表 6-2　Objective-C 的预定义类型

| 类型 | 描述 |
|---|---|
| BOOL | 布尔数据类型，只支持 YES 或 NO。该基础数据类型与平台非常相关，但 NO 为零而 YES 为非零，因此 BOOL 类型可用于 C 风格的条件语句 |
| id | 指向 Objective-C 对象的任何类型的指针 |
| IMP | 根据地址到对象方法的引用 |
| nil, Nil | 空指针，指向 Objective-C 对象 |
| SEL | 根据名字到对象方法的引用。名字 SEL 是 selector 的前三个字母，因为它被用于选择方法 |
| STR | char *的类型定义（typedef） |

### 6.2.2 创建接口声明

gcc 选项-gen-decl 为源文件中的类更新接口。确保头文件（接口定义）和类的源文件（实现）保持一致，非常有用。如果要增加新方法，或者要修改现存方法的调用顺序，需要按照选项-gen-decl 运行 GCC，生成正确的插入操作，替代现存接口的方法定义。

例如，前面例子的类 Speak 用下面的命令可以生成新的接口定义：

```
$ gcc -Wno-import -gen-decls -c Speak.m
```

选项-gen-decls 不会防止编译程序企图进行的编译和连接操作。有必要使用-c 选项防止 gcc 连接新编译的类定义。结果为文件 w.decl，内容如下：

```
@interface Speak : Object
- free;
- say;
- setString:(char *)str;
+ new;

@end
```

### 6.2.3 命名和 mangling

Objective-C 中的方法定义或者是加号开头(+)或者是减号开头(-)，分别代表类方法或实例方法。例如，下面类接口定义包括了类的方法 new 和 copy，以及实例的方法 reset 和 sort：

```
@interface TinyList : Object
+ new;
+ copy;
- reset;
- sort;
@end
```

为了调试，需要从目标代码中可识别出名字 mangled 后的形式。类的方法以 c 开头，随后紧跟类名，这样理解就可以分离出名字部分。实例的方法以 i 开头，后面的格式和类方法一样。前面例子中的 4 个方法可如下命名：

```
_c_TinyList__new
_c_TinyList__copy
_i_TinyList__reset
_i_TinyList__sort
```

具有多参数的方法可有多个名字。例如，下面的方法可接受两个字符指针——其中一个为 string，另一个为 desc——方法有两个名字 accept 和 as：

```
- accept: (char *) string as: (char *) desc
```

下面代码是调用类 Lister 的实例 lister 的方法的例子：

```
[lister accept: "Herbert" as: "name"]
```

目标代码中，该实例方法 mangled 后的名字如下：

```
_i_Lister__accept_as
```

# 第 7 章

# 编译 Fortran 程序

Fortran 因其强大的处理复杂数学计算的功能而声名在外。该点也令它在科学领域保留下来，并成为一种重要的语言。某些学科范围内，例如物理，Fortran 是最重要的编程语言。

GNU 的 Fortran 编译程序主要是基于 Fortran 77 的 ANSI 标准定义，但并没有局限于此。它包括 Fortran 90 和 Fortran 95 文档中定义的许多功能和特征（但不是所有）。与其被称作标准，Fortran 语言更像是一种传统，而标准文档也有很多内容体现在编译程序的实现上。大体上所有 Fortran 编译程序的工作方法都相同，但每种都支持自己一些独特的语法。

## 7.1 编译基础

表 7-1 列出编译和连接 Fortran 程序所涉及到的文件名后缀。附录 D 会列举出 GCC 识别的所有文件后缀。

**表 7-1 Fortran 编程中的文件名后缀**

| 后缀 | 文件内容 |
|---|---|
| .a | 静态对象库（文档） |
| .f, .for, .FOR | 不需要预处理的 Fortran 源代码 |
| .F, .fpp, .FPP | 需要预处理的 Fortran 源代码 |
| .o | 目标文件，其格式与连接程序相符 |
| .r | 被 Ratfor 预处理的 Fortran 源代码 |
| .so | 共享对象库 |

### 7.1.1 单独源文件到可执行程序

传统的 Fortran 程序可完全用大写字母编写，每行的前六个字符位置留空用于其他特殊作用。留出第一列为字符 C 说明该行是注释，从第二列到第六列留作标号（labels），代码从第七列开始。下面的例子是传统格式的 Fortran 程序：

```
C  helloworld.f
C
      PROGRAM HELLOWORLD
      WRITE(*,10)
   10 FORMAT('hello, world')
      END PROGRAM HELLOWORLD
```

GCC 编译程序不要求源代码必须全大写，但除非特殊说明，固定格式还是需要满足的。下面的命令将程序编译成可执行代码：

```
$ g77 helloworld.f -o helloworld
```

g77 命令是 gcc 的前端，用于设置 Fortran 程序需要的基本环境。通过使用如下 gcc 命令也可以得到同样结果：

```
$ gcc helloworld.f -lfrtbegin -lg2c -lm -shared-libgcc -o helloworld
```

库 libfrtbegin.a（会被命令行选项-lfrtbegin 调用）包括启动（startup）和退出（exit）代码，这对开始运行 Fortran 程序和清楚地结束程序都是必须的。库 libg2c.a 包括必要的 Fortran 运行时例程(routines)，提供基础的输入输出功能。选项-shared-libgcc 指出所用标准库 libgcc 的共享版本。

GCC 允许 Fortran 代码按照自由格式方式（free form format）进行编译。注释以叹号（!）开头，到行尾都为注释内容。前面程序的自由格式版本的语句和标号可在任何一列开始，如下：

```
!  helloworldff.f
!
Program Helloworld
write(*,10)
10 format('hello, world')
     end Program Helloworld
```

在命令行加上选项-ffree-form 可将程序编译地和以前一样，如下：

```
$ g77 -ffree-form helloworldff.f -o helloworldff
```

因为两种语法格式的本质区别，写程序时或者按照自由格式、或者按照固定格式——很难写出一个程序可同时按两种格式进行编译，因为语法中的注释方法和基本展开原则都不相同。

### 7.1.2　多源文件到可执行程序

g77 命令能够将多个 Fortran 源文件编译并连接成一个可执行程序。下面列出的是一个简单程序的主线，保存在文件 caller.f 中，它简单地调用了函数并显示结果：

```
C  caller.f
C
      PROGRAM CALLER
      I = Iaverageof(10,20,83)
      WRITE(*,10) 'Average=', I
   10 FORMAT(A,I5)
      END PROGRAM CALLER
```

函数 Iaverage 定义在另一个源文件 called.f 中，如下：

```
C  called.f
C
      INTEGER FUNCTION Iaverageof(i,j,k)
      Iaverageof = (i + j + k) / 3
      RETURN
      END FUNCTION Iaverageof
```

利用下面的命令可将两个源文件编译并连接成一个可执行程序 caller：

```
$ g77 caller.f called.f -o caller
```

可用以下三步实现同样的功能，首先为每个源文件创建目标文件，然后将它们连接成一个可执行程序，如下：

```
$ g77 -c caller.f -o caller.o
$ g77 -c called.f -o called.o
$ g77 caller.o called.o -o caller
```

### 7.1.3 生成汇编语言

选项-S 指示 g77 生成源代码的汇编语言，然后停止。键入下面命令，可以生成本章较早所用例子 helloworld.f 的汇编代码：

```
$ g77 -S helloworld.f
```

结果产生的汇编语言文件为 helloworld.s。汇编语言的实际格式依赖于目标平台。

### 7.1.4 预处理

编译文件后缀为.F、.fpp 或.FPP 的 Fortran 程序，会在编译开始以前先进行预处理操作。第 3 章描述的预处理程序，开始是为 C 编程语言设计的。下面例子是自由格式的 Fortran 程序，使用预处理程序将函数包含进主程序：

```
!  evenup.F
!
#define ROUNDUP
#include "iruefunc.h"
```

```
!
program evenup
do 300 i=11,22
   j = irue(i)
   write(*,10) i,j
300 continue
 10 format(I5,I5)
end program evenup
```

文件 iruefunc.h 中函数 irue()的源代码，根据是否定义了宏 ROUNDUP 可将函数编译为不同的格式。函数会将所有奇数转变成偶数。默认条件下，奇数会转变为较小的下一个偶数，但如果定义了宏 ROUNDUP，函数就会将奇数变成较大的下一个偶数。函数 irue()如下：

```
integer function irue(i)
k = i / 2
k = k * 2
if (i .EQ. k) then
   irue = i
else
#ifdef ROUNDUP
   irue = i + 1
#else
   irue = i - 1
#endif
end if
end function irue
```

下面命令会将程序编译成可执行代码：

```
$ g77 -ffree-form evenup.F -o evenup
```

使用预处理程序，不必按照自由格式编写程序。因为预处理程序去掉指示字，仅将结果代码传递给编译程序，下面程序也是有效代码：

```
C  adder.F
C
#define SEVEN 7
#define NINE 9
C
     program adder
     isum = SEVEN + NINE
     write(*,10) isum
  10 format(I5)
     end program adder
```

### 7.1.5 创建静态库

通过将 Fortran 源代码编译成.o 文件，再使用 ar 工具将目标文件保存到文档文件（archive）就可创建目标模块库，也就是静态库。

下面例子创建的库包含了两个简单函数，都会被主函数调用。第一个函数为 imaximum()，返回的是三个传入整数中最大的整数：

```
C  imaximum.f
C
      INTEGER FUNCTION imaximum(i,j,k)
      iret = i
      IF (j .gt. iret) iret = j
      IF (k .gt. iret) iret = k
      imaximum = iret
      RETURN
      END FUNCTION imaximum
```

第二个函数和第一个很像，只不过它返回的是三个整数中的最小值：

```
C  iminimum.f
C
      INTEGER FUNCTION iminimum(i,j,k)
      iret = i
      IF (j .lt. iret) iret = j
      IF (k .lt. iret) iret = k
      iminimum = iret
      RETURN
      END FUNCTION iminimum
```

下面的三个命令可以编译这两个函数，并将它们保存到库中：

```
$ g77 -c iminimum.f -o iminimum.o
$ g77 -c imaximum.f -o imaximum.o
$ ar -r libmima.a imaximum.o iminimum.o
```

g77 的-c 选项指示编译程序将源文件编译成目标文件，但不会激活连接程序。如果库 libmima.a 不存在，工具 ar 的-r 选项会创建该库。如果该库存在，库中所有目标文件会被命令行中命名的目标文件替换。

下面的程序调用库中保存的两个函数，并显示结果：

```
C  minmax.f
C
      PROGRAM MINMAX
      WRITE(*,10) 'Maximum=', imaximum(10,20,30)
      WRITE(*,10) 'Minimum=', iminimum(10,20,30)
```

```
10 FORMAT(A,I5)
   END PROGRAM MINMAX
```

下面命令可以编译该程序，并将它连接到库中保存的函数上：

```
$ g77 minmax.f libmima.a -o minmax
```

编译程序识别出 minmax.f 为 Fortran 源文件，会将源程序编译成目标文件，然后通过将库名 libmima.a 传递给连接程序，将程序连接成可执行代码。

### 7.1.6 创建共享库

创建共享库和创建静态库的步骤大致相同，但是库中的目标文件必须按照选项-fpic 或-fPIC 进行编译，这样代码才能载入内存并在程序运行的时候执行。（PIC 代表位置独立代码——position independent code。）

使用和静态库相同的源代码，利用下面的命令可以创建两个目标文件和共享库：

```
$ g77 -c -fpic iminimum.f -o iminimum.o
$ g77 -c -fpic imaximum.f -o imaximum.o
$ g77 -shared iminimum.o imaximum.o -o libmima.so
```

指导编译程序生成.o 目标文件，选项-c 是必需的；而生成正确格式的可在运行时从共享库加载的目标代码，需要用到选项-fpic。命令行中的选项-shared 将所有目标文件组合起来，组成共享库 libmima.so。对于应用程序使用的库，程序能在运行时对它定位非常必要，详见第 12 章所述。

要编译并连接使用共享库的程序，只需在连接程序的时候将共享库的名字加入到命令行中就行了：

```
$ g77 minmax.f -lmima -o minmax
```

选项-l 指明库为 mima，编译程序再将其扩展为 libmima.so，并在系统配置的找寻所有共享库的地方，根据此名进行查找。

## 7.2 Ratfor

Ratfor 是 Rational Fortran（合理 Fortran）的缩写。这是公布的源代码预处理程序，允许按照 C 语言风格编写 Fortran 程序，然后再转变为可编译的标准 Fortran。

最早的 Ratfor 翻译器是 Kernighan 和 Plauger 在 1976 年开发的。从最早出现在 AT&T 以后，已经发展出各种版本的 Ratfor 了。可从很多地方免费下载到两个最新版本，包括 http://sepwww.stanford.edu/software/ratfor.html 下的 ratfor77 和 http://sepwww.standord.edu/software/ratfor90.html 下的 ratfor90。下载的东西很小，安装也非常简单。安装过程会安装

ratfor77 （这是由 C 程序编译的二进制可执行代码）或者 ratfor90 （为 Perl 语言脚本）。两者都可以用于生成 Fortran 代码作为 GCC 编译程序的输入。

Ratfor 的这两个版本非常不同，应该选择使用其中之一。Ratfor90 并不是 Ratfor77 的扩展。可用 Ratfor90 编译的简单程序不能用 Ratfor77 编译，反之亦然。

Ratfor 程序的源代码是 Fortran，可被写成纯 Fortran，但可使用很多 C 的构造（constructs）。下面的简单例子展示了 Ratfor 程序的形式及表现：

```
# ratdemo.r
program ratdemo {
    integer i;
    integer counter;

    counter = 10;
    for(i=0; i<10; i=i+1) {
        counter = counter + 5;
        write(*,10) i, counter;
    }
  10 format(I5,I5);
}
end program ratdemo
```

传给 ratfor77 的代码可用下面两条命令编译成可执行代码：

```
$ ratfor77 <ratdemo.r> ratdemo.f
$ g77 ratdemo.f -o ratdemo
```

文件 ratdemo.f 为 Ratfor 翻译器的输出结果，是 Fortran 程序，看起来如下：

```
C Output from Public domain Ratfor, version 1.0
      program ratdemo
      integer i
      integer counter
      counter = 10
      i=0
23000 if(.not.(i.lt.10))goto 23002
      counter = counter + 5
      write(*,10) i, counter
23001 i=i+1
      goto 23000
23002 continue
10    format(i5,i5)
      end program ratdemo
```

## 7.3　GNU Fortran 扩展功能与变量

GCC 编译程序支持 ANSI Fortran 77 标准，也提供其他一些 GNU 的扩展功能。它对 Fortran 90 定义的部分功能也有所支持，但不支持它定义的所有功能。

### 7.3.1　内在函数

GNU 的 Fortran 编译程序包括很多内在函数（intrinsic function）。可在 GNU 站点上找到有关这些函数的文档，而且还有一些应用实现不仅包括大量 GNU 特定的内在功能，而且还有其他地方定义的内在函数。

ANSI FORTRAN 77 语言规范定义了通用的和特定的内在函数集合。特定的（specific）内在函数会返回特定的数据类型，通用的（generic）内在函数的返回类型完全依赖于它的使用方法——通常返回类型由它的一个参数值类型决定。

GCC 的 Fortran 编译程序对于传递给内在函数的参数需求比其他编译程序要求的更多，可以找到这样的程序，它们能够编译出其他编译程序可用的程序，但是 g77 会拒绝编译此程序。例如，如果变量 X 声明为 INTEGER*8，而 ABS()内在函数不会接受这样的参数，因为编写该函数只能接受 INTEGER*4 的参数，而拒绝其他额外精度。有必要对这些源加以调整，可以是简单的强制转换。

GCC Fortran 支持 MIL-STD 1753 内在函数 BTEST、IAND、IBCLR、IBITS、IEOR、IOR、ISHIFT、ISBFTC、MVBITS 和 NOT。

f77 和 f2c 中的基本函数对 g77 都是有效的。这包括位处理内在函数 AND、LSHIFT、OR、RSHIFT 和 XOR。其他一些受支持的内在函数有 CDABS、CDCOS、CDEXP、CDLOG、CDSIN、CDSQRT、DCMPLX、DCONJG、DFLOAT、DIMAG、DREAL、IMAG、ZABS、ZCOS、ZEXP、ZLOG、ZSIN 和 ZSQRT。

一句话，GCC 支持 402 个 Fortran 内在函数，都可找到有关文档。

### 7.3.2　源代码格式

如本章较早例子所示，GNU Fortran 接受 ANSI Fortran 77 标准的源代码，也接受自由格式的源代码。自由格式源代码和 Fortran 90 格式的源代码非常相似，但是 GNU Fortran 对于制表符（tab）这类要求更加宽大。如下综合列举了自由格式和固定格式的一些特殊情况：

- **回车**　忽略源代码中的回车字符。
- **制表符**　每个制表符都被扩展为合适数目的空格，扩展为 8 字符的边界。

- **与符号** 固定格式的第一列为与符号（&），说明该行和上一行是连续的。
- **短行** 自由格式对行的长度没有要求，但是固定格式要求每行都是 72 个字符。如果少于 72 字符，行的右边会被自动添入空格以满足 72 字符的要求。这只对连续的字符和 Hollerith 常数才有效。这种固定行的需求可通过使用命令行选项-ffixed-line-length 加以修改和清除。
- **长行** 长于设计的长度的行会被截断，没有警告。这更适用于传统的 Fortran 代码，它们在 73 到 80 列会包含其他信息（通常是源代码序列号）。这种固定行的需求可通过使用命令行选项-ffixed-line-length 加以修改和清除。

### 7.3.3 注释

只有要对代码进行预处理时，才能用字符/*和*/创建注释块，因为预处理程序会去掉注释块，这样编译程序就看不到那些注释。不能使用//声明注释行，因为这些符号在 Fortran 语言中已有其他意义（串连，concatenation）。GNU Fortran 用字符!将当前行剩余部分作为注释内容，而不论此代码是否被预处理过。当然，固定格式中以字符 c 或 C 作为第一列的行是注释行。

### 7.3.4 美元符号

只要美元符号不是名字中的一个字符，而且命令行用到选项-fdollar-ok，就可使用美元符号。

### 7.3.5 大小写敏感

很多选项可以组合成特定的字母大小写规则。默认情况下，对输入源代码没有限制——大小写字母都可用，且对它们不加区分。设置选项可以限制或调整大小写规则，不影响注释内容、字符常数或 Hollerith 域。

表 7-2 列出用于设置输入源大小写规则的选项。对于 Fortran 关键字、内在函数和程序定义的符号有各自的不同设置。表 7-3 分别描述了表 7-2 列举的 4 种选项（any、upper、lower 和 initcap）。

可用三种设置来确定输出的汇编语言符号的大小写规则，如表 7-4 所示。设置选项时必须对此加以考虑，因为外部引用必须和库例程相匹配。

表 7-2 和表 7-4 中选项的组合是通用的，也可指定为表 7-5 所列举的某个选项。

**表 7-2 指明大小写需求的选项**

| 关键字 | 内在函数 | 符号 |
|---|---|---|
| -fmatch-case-any | -fintrin-case-any | -fsymbol-case-any |
| -fmatch-case-upper | -fintrin-case-upper | -fsymbol-case-upper |

（续表）

| 关键字 | 内在函数 | 符号 |
|---|---|---|
| -fmatch-case-lower | -fintrin-case-lower | -fsymbol-case-lower |
| -fmatch-case-initcap | -fintrin-case-initcap | -f symbol-case-initcap |

表 7-3　4 种可能的大小写需求

| 选项 | 描述 |
|---|---|
| any | 对大小写没有特别限制，所有组合都满足。例如，Function、FUNCTION、function 和 FuncTion 都一样 |
| upper | 所有字符都必须是大写的 |
| lower | 所有字符都必须是小写的 |
| initcap | 第一个字母必须是大写的，其他字母必须是小写的。例如，Maximum、Function、Do 和 Return |

表 7-4　输出到汇编器的大小写控制

| 选项 | 描述 |
|---|---|
| -fsource-case-preserve | 输出的汇编语言和编译程序的输入大小写一致 |
| -fsource-case-upper | 汇编语言输出的所有符号都要转换为大写字符 |
| -fsource-case-lower | 汇编语言输出的所有符号都要转换为小写字符 |

表 7-5　可指明输入和输出大小写组合的独立选项

| 选项 | 描述 |
|---|---|
| -fcase-initcap | 选项要求每个开头的字符都要大写，除了注释和字符常量。这与将表 7-2 中的三个选项都指明为 initcap 一样，还要指定-fsource-case-preserve |
| -fcase-lower | 这是“规范”的 UNIX 模式，即将所有源都翻译成小写字符。这和将表 7-2 中的三个选项都指明为 any 一样，也要指明选项-fsource-case-lower |
| -fcase-preserve | 该选项允许任何输入模式，输入的大小写和输出的汇编语言一致。这和将表 7-2 中的三个选项都指明为 any 一样，同时要指明选项-fsource-case-preserve |
| -fcase-strict-upper | 这是“严格”的 ANSI FORTRAN 77，要求除了注释和字符常量以外的所有符号都为大写。这和将表 7-2 的三个选项都指定为 upper 一样，同时要指明选项-fsource-case-preserve |
| -fcase-strict-lower | 该选项要求除了注释和字符常量以外的所有符号都为小写。这和将表 7-2 的三个选项都指定为 lower 一样，同时要指明选项-fsource-case-preserve |
| -fcase-upper | 这是“经典”的 ANSI FORTRAN 77 模式，此处源中所有符号都要翻译成大写字符。这和将表 7-2 的三个选项都指明为 any 一样，同时要指明-fsource-case-upper 选项 |

### 7.3.6 特殊 Fortran 90 功能

本节简要描述了 Fortran 90 中一些很有用的功能，它们都是 g77 支持的功能。因为语言规范非常大而且很复杂，所以不可能列出完整的内容，但下面 g77 的功能不需要任何特殊的标志或设置就可工作。

#### 字符串

字符串常量可以由双引号围绕，也可由单引号围绕起来。就是说"hello world"和'hello world'一样。在双引号定义的字符串内，单个双引号符号要依靠一对双引号才能定义。

字符常量可为零长度（不包括字符）。而且，可以按照这种方式'hello world'(7:4)声明子串，它的值为'worl'。

#### 构造名

构造名（construct name）用于指明可执行的语句块，由 IF、DO 或 SELECT CASE 语句组成。下面的例子使用构造名 cname 作为标识符，用在 IF 语句块的开始和结尾处：

```
C  conname.f
C
      PROGRAM conname
      key = 12
      cname: IF(key .gt. 10) THEN
         key = key - 1
         WRITE(*,10) key
      END IF cname
10    FORMAT('Key=',I5)
      END PROGRAM conname
```

#### 循环和循环出口

循环出口语句（EXIT）用于从循环中立即终止，并跳出循环，执行其后的代码。就是说，在循环中执行 EXIT 语句和执行 GOTO 语句一样，可以跳到跟在循环后面的语句上。（如果熟悉 C 语法，会了解 Fortran 中的 EXIT 语句和 C 语言中的 break 一样。）

循环语句（CYCLE）用于立即终止本次循环，并跳到循环体的结尾处，以便再次开始下一次的循环。就是说，在循环中执行 CYCLE 语句和在循环中执行 GOTO 语句跳到 CONTINUE 处一样，它是循环的最后一条语句。（如果熟悉 C 语法，会了解 Fortran 中的 CYCLE 和 C 中的 continue 一样。）

下面例子演示了 EXIT 和 CYCLE 语句：

```
C  cycle.f
      PROGRAM cycle
      DO 10 i=1,3
```

```
        IF (i .EQ. 2) CYCLE
        WRITE(*,30) i
 10   CONTINUE
      DO 20 i=1,3
        IF (i .EQ. 2) EXIT
        WRITE(*,30) i
 20   CONTINUE
 30   FORMAT('i=',I5)
      END PROGRAM cycle
```

下面是该程序的输出：

```
i=    1
i=    3
i=    1
```

第一次循环写数字 1，第二次循环跳过 WRITE 语句（跳到循环底部），第三次循环写数字 3。第二条循环语句在第一次循环过程中写 1，而会在第二次循环过程中终止该循环。

### DO WHILE 语句

DO WHILE 语句可用逻辑表达式，由 END DO 语句终止循环，如下例所示：

```
C  dowhile.f
      PROGRAM dowhile
      k = 5
      DO WHILE ( k .gt. 0)
         WRITE(*,20) k
         k = k - 1
      END DO
 20   FORMAT('k=',I5)
      END PROGRAM dowhile
```

### DO Forever 语句

使用不含条件的 DO 语句，会构造一个可以连续执行的循环，直到程序被终止或执行了明确的退出循环语句。下面例子会反复执行循环操作，直到 8 次后会用 GOTO 语句跳出循环：

```
C  doforever.f
      PROGRAM doforever
      k = 0
      DO
         WRITE(*,20) k
         if ( k .ge. 8 ) GOTO 100
         k = k + 1
```

```
      END DO
 20   FORMAT('k=',I5)
100   CONTINUE
      END PROGRAM doforever
```

### IMPLICIT NONE 语句

IMPLICIT NONE 语句防止自动声明变量，要求所有变量都要显式声明其类型。例如，下面程序自动定义并假设了循环计数器类型：

```
      PROGRAM imp
      DO 10 k=1,5
         PRINT *,k
10    CONTINUE
      END PROGRAM imp
```

在程序开头加上 IMPLICIT NONE 语句，要求所有变量都要在使用前加以声明，包括循环计数器，如下：

```
      PROGRAM imp
      IMPLICIT NONE
      INTEGER k
      DO 10 k=1,5
         PRINT *,k
10    CONTINUE
      END PROGRAM imp
```

### INCLUDE

标准中定义的 INCLUDE 指示字的语法如下：

```
INCLUDE filename
```

filename 的意义来自于实现本身。GNU 编译程序将 filename 解释为当前目录中的文件名，或是由命令行选项-I 指定目录内的文件名。因此指示字 INCLUDE 和预处理指示字 #include 用法一致，这在本章较早已有描述，只是 INCLUDE 指示字不需要预处理。

#### 整数常量

整数常量值可为二进制、八进制、十进制和十六进制。下面的例子用不同进制表达同一数值。二进制数字（binary）以字母 B 打头，八进制常数（octal）以字母 O 开头，十六进制常数（hexadecimal）以字母 X 或字母 Z 开头，而且十六进制数字既可为大写也可为小写字母。

下面的例子展示了声明各种不同进制常量的语法，每种进制的值都相同：

```
C  bases.f
C
      PROGRAM bases
      M = 18987
      PRINT *,M
      M = X'4A2b'
      PRINT *,M
      M = Z'4A2b'
      PRINT *,M
      M = O'45053'
      PRINT *,M
      M = B'0100101000101011'
      PRINT *,M
      END PROGRAM bases
```

### 比较操作符

表 7-6 列举的符号可以取代传统的比较操作符。

**表 7-6　原始比较符号的等价字符**

| 原始符号 | 可选方式 | 意义 |
|---|---|---|
| .GT. | > | 大于 |
| .LT. | < | 小于 |
| .GE. | >= | 大于等于 |
| .LE. | <= | 小于等于 |
| .NE. | /= | 不等于 |
| .EQ. | == | 等于 |

### 数据种类（KIND）

一种新产生的特殊符号，可用于修改传统变量类型。例如，定义 KIND 3 的 INTEGER 值的语法如下：

```
INTEGER(KIND=3)
```

KIND 的可用值为 0、1、2、3、5 和 7。该语法对所有通用类型都有效（INTEGER、REAL、COMPLEX、LOGICAL 和 CHARACTER），尽管并不是所有类型的所有值都有效。表 7-7 描述了每个值和 GCC 如何（以及能否）将它们应用于数据类型。KIND 值的确切含义根据硬件平台的不同而不同，例如默认类型的尺寸和范围。

表 7-7 为 KIND 符号定义的数值

| KIND 值 | 描述 |
|---|---|
| 0 | 该值目前没有效果，留作将来使用。有计划要让结果类型成为上下文敏感的，它的语义要根据用法调整 |
| 1 | 这是默认配置。结果和未指定 KIND 值一样。典型地为 REAL*4、INTEGER*4、LOGICAL*4 和 COMPLEX*8 |
| 2 | 这些类型占据默认空间的两倍。GNU 中变量 KIND 的值和 Fortran 90 标准的双精度一样。也就是说，REAL(KIND=2)等于 DOUBLE PRECISION，它是典型的 REAL*8。而 COMPLEX(KIND=2)也等于 DOUBLE COMPLEX，而它是典型的 COMPLEX*16。并不是所有 GNU 实现都支持 INTEGER(KIND=2)和 LOGICAL(KIND=2) |
| 3 | 这些类型占据的空间比 CHARACTER(KIND=1)类型大很多。这些是典型的 INTEGER*1 和 LOGICAL*1。对于所有 GNU 实现的所有类型而言，该变量 KIND 值不是必须实现的 |
| 5 | 这些类型占据默认类型空间的一半(默认类型空间由 KIND=1 指定)。典型的为 INTEGER*2 和 LOGICAL*2。对于所有 GNU 实现的所有类型而言，该变量 KIND 值不是必须实现的 |
| 7 | 只对 INTEGER(KIND=7)有效，这是可能的最小指针的最小尺寸，该指针用来保存 CHARACTER*1 变量的惟一地址。32 位系统中，它和 INTEGER*4 相等；而 64 位系统中，它等于 INTEGER*8 |

# 第 8 章 编译 Java

尽管 Java 语言没有像其他语言（如 C、C++和 Ada）那样公布官方标准，但 Java 语言拥有独立且非常清楚的定义。Sun Microsystems 公司完全控制语言的定义，而且负责维护语言的功能扩展。语言本身的基本操作和语法改动很小，但 API（系统类）会定期更新，比原来扩大了很多倍。

就编译程序而言，Java 和其他语言是有区别的，因为它对每个目标平台都有两种完全不同的目标代码格式。比如像 C、C++或其他编译型语言，都会使用编译程序生成二进制可执行的目标文件，这些文件在目标机器上直接就能够运行。而 Java 编译程序不仅能够生成 Java 字节码（bytecode）格式的目标代码，可在 Java 虚拟机（Java Virtual Machine，JVM）上运行；也能够将 Java 字节码作为输入生成本地可执行的目标代码。

## 8.1 编译基础

表 8-1 列出编译和连接 Java 程序涉及到的文件名后缀。附录 D 中有表格列举了 GCC 可识别的所有文件名后缀。

表 8-1 Java 编程的文件名后缀

| 后缀 | 文件内容 |
|---|---|
| .a | 库（文档文件），包括静态连接的目标文件 |
| .class | 目标文件，包括可由 Java 虚拟机执行的字节码 |
| .java | Java 源代码 |
| .o | 二进制目标文件，格式和连接程序相符 |
| .s | 汇编语言源代码 |
| .so | 共享库；包含动态连接的目标文件 |

### 8.1.1 单一源文件到二进制可执行程序

为将 Java 类编译成可执行程序，该类必须是 public，而且还必须包含公共方法 main()，如下例所示：

```
/* HelloWorld.java */
public class HelloWorld {
    public static void main(String arg[]) {
        System.out.println("hello, world");
    }
}
```

为编译 Java 程序，必须使用 gcj 命令，它是 gcc 编译程序的 Java 前端。Java 语言允许所有类都含有自己的主方法 main()，且可以运行。对于 Java 解释器而言，它能很好地工作，因为可在运行程序时通过命令行指定类名。但在处理可执行程序时，必须明确指出独立的起点。下面的命令编译 HelloWorld.java ，并将它连接成本地可执行程序。指定的--main 选项告诉程序必须使用 HelloWorld 类的 main()方法作为程序的起点：

```
$ gcj --main=HelloWorld -Wall HelloWorld.java -o HelloWorld
```

选项-o 将可执行程序命名为 HelloWorld，否则会使用默认名字 a.out。为运行该程序，只需在命令行键入名字，如下：

```
$ HelloWorld
```

因为文件是二进制可执行程序，这与解释过的 Java 类文件所需满足的命名约束无关。可执行程序可以任意命名，如下例，可将 HelloWorld.java 程序编译为可执行文件 howdy：

```
$ gcj --main=HelloWorld -Wall HelloWorld.java -o howdy
```

但只有二进制可执行文件才这么简单。公共类的源文件必须和该公共类同名。就是说，公共类 HelloWorld 必须由文件 HelloWorld.java 定义。

### 8.1.2 单一源文件到类文件

可用 GNU 编译程序产生 Java 的.class 文件，它们可在 Java 虚拟机上运行。下面命令可用选项-C 由源文件 HelloWorld.java 创建目标文件 HelloWorld.class：

```
$ gcj -C -Wall HelloWorld.java
```

选项-o 不能与选项-C 组合在一起使用，这样输出的.class 文件和输入的.java 文件的基本名总是相同的。类 HelloWorld 包括必需的方法 public static void main()，这样虚拟机利用如下命令行就可以运行它：

```
$ gij HelloWorld
```

类文件和 Java 解释器相兼容。Sun 的 Java 虚拟机可以运行同一个程序，如下：

```
$ java HelloWorld
```

### 8.1.3　单一源文件到二进制目标文件

下面命令使用-c 选项可以禁止连接操作，并且产生二进制目标文件，这样的二进制文件既可连接成可执行代码，又可保存在静态库中以备将来连接之用：

```
$ gcj -c HelloWorld.java
```

该命令行产生目标文件 HelloWorld.o。也可以通过选项-o 指定目标文件的名字，如下例所示：

```
$ gcj -c HelloWorld.java -o hello.o
```

gcj 命令用于将文件 hello.o 连接成可执行代码。文件 hello.o 包括类 HelloWorld 的定义，而该类包含的静态方法 main()用作程序的入口，因此必须在命令行中加以说明，如下：

```
$ gcj --main=HelloWorld hello.o -o hello
```

这种方式几乎不需要修改目标文件的名字。本例说明这样的实事，即--main 选项需要的是类的名字，而不是文件的名字。

### 8.1.4　类文件到本地可执行程序

用 gcj 命令可将 Java 字节码直接编译成本地可执行程序。该行命令将.class 后缀的文件看作是.java 后缀的文件。下面的例子中，第一条命令将源文件编译成类文件，而第二条命令将类文件编译成可执行代码：

```
$ gcj -C HelloWorld.java
$ gcj HelloWorld.class -o HelloWorld
```

### 8.1.5　多源文件到二进制可执行文件

将多个 Java 源文件编译成一个可执行程序时要分别编译每个源文件，然后通过指出包含 main()方法的源文件将它们连接成一个可执行程序。下面简单例子的主体部分会用另外一个类来构造字符串，再用一个类来显示该字符串。类 SayHello 包含主体部分：

```
/* SayHello.java */
public class SayHello {
    public static void main(String arg[]) {
        WordCat cat = new WordCat();
        cat.add("Hello");
        cat.add("cruel");
        cat.add("world");
```

```
        Say say = new Say(cat.toString());
        say.speak();
    }
}
```

类 WordCat 的方法 add()接受一个词，再将它附加在内部字符串之后。类 WordCat 的方法 toString()返回结果字符串，该字符串会被传递给类 Say 的对象的 speak()方法，显示字符串。下面是 WordCat 类，一次构造一个字符串：

```
/* WordCat */
public class WordCat {
    private String string = "";
    public void add(String newWord) {
        if(string.length() > 0)
            string += " ";
        string += newWord;
    }
    public String toString() {
        return(string);
    }
}
```

类 Say 包括一个字符串，以及 speak()方法用于显示字符串：

```
/* Say.java */
public class Say {
    private String string;
    Say(String str) {
        string = str;
    }
    public void speak() {
        System.out.println(string);
    }
}
```

这三个类可被按照几种方式编译成本地可执行代码。最直接的方法就是按照下面的一行命令：

```
$ gcj --main=SayHello Say.java SayHello.java WordCat.java -o SayHello
```

该命令可将三个源文件编译成目标文件，然后将目标文件连接成一个可执行文件 SayHello（将 SayHello 的 main()方法作为程序的入口）。使用下面的几行命令也可达到同样的效果，编译每个目标文件并将它们连接成可执行程序：

```
$ gcj -c SayHello.java
$ gcj -c Say.java
```

```
$ gcj -c WordCat.java
$ gcj --main=SayHello Say.o SayHello.o WordCat.o -o SayHello
```

也有这样的可能，即先将源文件编译成类文件，再编译这些类文件并将它们连接成可执行程序，下一小节对此加以描述。

### 8.1.6　多输入文件到可执行程序

用到的源代码和前面一节的相同，下面的命令可用来将三个源文件编译成三个类文件：

```
$ gcj -C SayHello.java Say.java WordCat.java
```

可利用 Java 虚拟机执行生成的类文件，如下：

```
$ java SayHello
```

使用下面的命令，可将当前目录中所有的 Java 源代码编译成类文件：

```
$ gcj -C *.java
```

Java 类文件可被看作是源代码文件，可被编译并连接成本地可执行程序。下面例子中的第一条命令将源文件编译成类文件集合，第二条命令将类文件编译成本地可执行程序：

```
$ gcj -C SayHello.java Say.java WordCat.java
$ gcj --main=SayHello Say.class WordCat.class SayHello.class -o
  SayHello
```

gcj 命令通过检查命令行中文件名后缀来确定对输入文件采取什么样的操作。如果后缀是.java，编译程序知道这是 Java 源代码文件，必须被编译。如果后缀为.class，就会假设文件为可被编译的 Java 字节码。.o 后缀则指出它是本地目标文件，可直接连接成本地可执行程序。因此可将源文件、类和目标文件混合在一起，编译并连接程序，如下：

```
$ gcj -c SayHello.java -o SayHello.o
$ gcj -C WordCat.java
$ gcj --main=SayHello SayHello.o Say.java WordCat.class -o SayHello
```

### 8.1.7　生成汇编语言

在执行如下类代码的时候，会创建自己的实例，用于在标准输出设备上显示字符串：

```
/* Jasm.java */
public class Jasm {
    public static void main(String arg[]) {
        Jasm jsm = new Jasm();
        jsm.speak();
    }
    public void speak() {
```

```
        System.out.println("Jasm speaks");
    }
}
```

该类是一个完整的应用，可被编译并运行。也可使用如下命令，将其编译成本地的汇编语言：

```
$ gcj -S Jasm.java
```

该命令的输出结果是汇编代码文件 Jasm.s，可用来创建可执行程序。

生成汇编语言文件的另一种方法是将类文件作为输入。下面两个命令从 Jasm.java 中创建类文件，再用它生成汇编语言文件：

```
$ gcj -C Jasm.java
$ gcj -S Jasm.class
```

这些命令的输出文件为 Jasm.class 和 Jasm.s。

### 8.1.8 创建静态库

静态库是.o 文件的集合，保存在一个文件中，叫做静态库或文档文件。将程序和库中的内容相连接与将程序和单独的目标文件相连接是一样的。

使用本章较早例子的源文件，利用下面的命令，可以创建目标文件 WordCat.o 和 Say.o，保存在库中：

```
$ gcj -c WordCat.java Say.java
```

工具 ar 用来构造并维护静态库。使用 ar 的选项-r 会由目标文件创建库文件，而在库文件已经存在的条件下，该选项会将目标文件更新为最新版本。下面命令创建了库 libsay.a，它包含两个目标文件：

```
$ ar -r libsay.a WordCat.o Say.o
```

要使用库中的目标文件，只需要将库名加到 gcj 命令中就可以了，如下例，会生成可执行程序名为 libhello：

```
$ gcj --main=SayHello SayHello.java libsay.a -o libhello
```

在命令行中指定库名要求该库必须位于当前目录。如果 gcj 查找的目录正是库所在的位置，可用-l 选项说明库的名字，如下例所示：

```
$ gcj --main=SayHello SayHello.java -lsay -o libhello
```

更多关于定位库的信息可参见第 12 章。

### 8.1.9 创建共享库

共享库是保存在一个文件之中的目标文件的集合，这与静态库的方式一样，但有两点主要区别。首先，共享库（也叫动态库）中的目标文件会在程序开始运行的时候，被载入并连接到程序上。其次，目标文件必须按照特殊方式编译，才能保证载入内存后不需修改就可以运行，而载入内存的位置也是可变的。下面的例子用到本章较早介绍的源文件。

要创建共享库所保存的目标文件，这些文件必须按照选项-fpic 进行编译，生成位置独立代码（position independent code）。就是说代码使用相对地址作为内部引用和分支处理，这就不必每次将代码载入内存的时候都重新定位。如下命令会生成所需格式的目标文件：

```
$ gcj -fpic -c WordCat.java Say.java
```

命令 gcj 使用选项-shared 将目标文件连接到新的共享文件 libsay.so 上，如下：

```
$ gcj -shared WordCat.o Say.o -o libsay.so
```

源文件 SayHello.java 可被编译为可执行程序 shlibhello，通过在命令中加上库名，它就可以使用库中保存的目标文件，如下：

```
$ gcj --main=SayHello SayHello.java libsay.so -o shlibhello
```

libsay.so 的实际内容没有包含在可执行的 shlibhello 中。包含在可执行代码中的是一些指令，用于从合适的共享库中加载所需对象模块。为实现该过程，可执行代码必须能在运行时找到库的位置。关于共享库定位的有关知识可参见第 12 章。

### 8.1.10 创建 Jar 文件

Java 语言有种特殊的文档文件，其中包括类文件。Java 的文档文件叫做 jar 文件。jar 文件的格式和 zip 文件一样，但是 jar 文件还包含特殊的清单（manifest），其中包含的是描述信息。Java 所有的外部引用全都基于类名，所以 Java 程序惟一需要做到的就是能够定位正确的 jar 文件（或者就是文件），这会引起对清单的查找，就是为了找到所需类的位置。下面的例子用到了本章较早介绍的源文件。

要创建 jar 文件，首先需要将源文件编译成类文件，如下例所示：

```
$ gcj -C WordCat.java Say.java
```

工具 jar 的选项 c 会创建 jar 文件，而选项 f 指出 jar 文件的名字是命令行的第二个参数。命令行的剩余部分由保存在 jar 文件中的类名组成。下面的命令创建一个 jar 文件 libsay.jar，其中包括两个类和一个清单：

```
$ jar cf libsay.jar WordCat.class Say.class
```

jar 文件中保存的类文件可被编译，并直接连接到可执行程序上，和目录中保存的 jar

文件一样。下面的命令将 Java 主线类 SayHello.java 编译成可执行程序 jarlibhello，通过编译该文件并把它连接到 jar 文件 libsay.jar 中的类来实现：

```
$ gcj --main=SayHello libsay.jar SayHello.java -o jarlibhello
```

## 8.2 Java 工具

除了 gcj 编译程序，GCC 的发布版还提供了大量工具程序，用于处理 Java 源代码和目标文件。

### 8.2.1 gij

工具 gij 是 Java 虚拟机，用于解释并执行 Java 类文件中找到的字节码。命令行包含类文件名或要执行的 jar 文件名。例如，下面的 Java 程序会回显命令行内容：

```
/* ListOptions.java */
public class ListOptions {
    public static void main(String arg[]) {
        for(int i=0; i<arg.length; i++) {
            System.out.println(arg[i]);
        }
    }
}
```

程序可编译成类文件，然后再用以下命令执行生成的类文件：

```
$ gcj -C ListOptions.java
$ gij ListOptions
```

命令行中类名后的所有参数都会传给运行中的程序。类 ListOptions 会把选项回显在标准输出设备上，因此命令行运行了类文件后的结果如下：

```
$ gij ListOptions apple butter --help
apple
butter
--help
```

表 8-2 列出命令 gij 可用的所有选项。命令行中类名或 jar 文件名前的所有选项都假设为 gij 的选项。

表 8-2　gij 可用的命令行选项

| 选项 | 描述 |
|---|---|
| -Dname[=value] | 名字成为所定义的系统属性名，具有指定的值。如果忽略值，定义的名字就是长度为零的字符串 |
| --help | 打印选项然后退出 |
| -jar | 命令行的名字被解释为 jar 文件的名字，而不是类文件的名字 |
| -ms=number | 数字为堆的初始尺寸 |
| -mx=number | 数字为堆的最大尺寸 |
| --version | 打印 gij 版本号然后退出 |

选项-jar 使得可以执行 jar 文件保存的类。必须构造该 jar 文件，使之能够包含清单文件，并且它的属性 Main-Class 应为所执行类的名字。例如，jar 文件 sayhello.jar 中，如果类文件 SayHello.class 含有 main()方法，是程序的入口，那么清单文件必须包括下面这行：

```
Main-Class: SayHello
```

下面命令会执行该 jar 文件：

```
$ gij -jar sayhello.jar
```

### 8.2.2　jar

jar（Java 文档）文件包括 Java 类文件的集合，也可以是其他文件的集合，它可被 Java 虚拟机直接读取并执行。工具 jar 可用来创建 jar 文件，并可浏览和改变其内容。表 8-3 列出 jar 的命令行选项。

表 8-3　jar 的命令行选项

| 选项 | 描述 |
|---|---|
| -@ | 从标准输入读出文件名列表 |
| -c | 创建新的 jar 文件 |
| -C dir file | 从目录 dir 中查找文件 file |
| -E dir | 指出不包含目录 dir 中的文件 |
| -f file | 命名的文件是 jar 文件 |
| --help | 打印选项列表和其他简单帮助信息 |
| -m file | 命名文件包含所包括的清单信息 |
| -M | 指出不会创建清单文件 |
| -O | 将文件保存为 jar 文件，不需要使用压缩方式 |

（续表）

| 选项 | 描述 |
|---|---|
| -t | 列出 jar 文件的内容 |
| -u | 更新存在的 jar 文件 |
| -v | 将详细的输出显示到标准输出设备上，描述所采取的动作 |
| -V | 同--version |
| --version | 显示 jar 工具的版本号 |
| -x | 展开 jar 文件中的文件 |

工具 jar 的命令行选项和 UNIX 工具 tar 的命令行选项非常相似。在命令行开始位置可使用选项字母，无需在选项前加连字号。例如，如下命令创建 jar 文件 sayhello.jar，包括当前目录中所有类文件：

```
$ jar cvf sayhello.jar *.class
```

使用下面命令，可以创建相同的 jar 文件，并且可以指定清单文件包括文本文件 hello.manifest 的信息：

```
$ jar cvfm sayhello.jar hello.manifest *.class
```

jar 文件名以及清单文件名出现的顺序和 f 选项及 m 选项的顺序相同。下面的命令和前面的命令相同，只是颠倒了命令行中文件名的顺序：

```
$ jar cvmf hello.manifest sayhello.jar *.class
```

在选项字母前使用连字号也可产生相同的结果，如下例：

```
$ jar -c -v -f sayhello.jar -m hello.manifest
```

下面命令列出 jar 文件 sayhello.jar 的内容：

```
$ jar tvf sayhello.jar
```

jar 文件的内容可以是一个简单的文件，也可以是一个完整的目录树。清单文件总是名为 MANIFEST.MF，保存在子目录 META-INF 中的 jar 文件里。

### 8.2.3 gcjh

Java 的本地方法可写作 CNI（一种 C++接口）或 JNI（一种 C 接口）。gcjh 工具读出 Java 类文件，生成 CNI 和 JNI 头文件，以及用于实现本地方法的存根文件（stub files）。CNI 头文件被包含在 C++程序中，而 JNI 文件可合法包含在 C 程序中。选项-stubs 可用来生成 starter C 和 C++文件，它们用 JNI 或 CNI 实现本地方法。表 8-4 列出命令 gcjh 可用的选项。

表 8-4　gcjh 的命令行选项

| 选项 | 描述 |
|---|---|
| -add text | 将指定文本插入 C++类体中。如果同时指定选项-jni，则会忽略本选项 |
| -append text | 将指定文本插入头文件，跟在 C++类声明之后。如果同时指定选项-jni，则会忽略本选项 |
| --bootclasspath=path | 重写内置类路径 |
| --classpath=path | 指明路径，用于定位类文件 |
| --CLASSPATH=path | 指明路径，用于定位类文件 |
| -d directory | 指明输出的目录名 |
| -friend text | 将指定文本作为友元声明（friend declaration）插入 C++的类定义中。如果同时指定选项-jni，则会忽略本选项 |
| --help | 在标准输出中列举本表中的各选项 |
| -Idirectory | 在类路径之后，附带加上指定目录 |
| -M | 禁止正常输出，将所有依赖关系都打印到标准输出设备上 |
| -MD | 将所有依赖关系都打印到标准输出设备上 |
| -MM | 禁止正常输出，将所有非系统依赖关系（non-system dependencies）都打印到标准输出设备上 |
| -MMD | 将所有非系统的依赖关系都打印到标准输出设备上 |
| -o file | 指出输出文件的名字。如果输出的是多个文件，本选项会出错 |
| -prepend text | 将指定文本插入头文件，位于 C++类声明之前。如果同时指定选项-jni，则会忽略本选项 |
| -stubs | 生成存根文件，而不生成头文件。存根文件的基本名和类名相同，但是文件后缀为.cc。如果同时指定选项-jni，后缀就为.c |
| -td directory | 临时文件使用的目录名 |
| -v | 处理过程中打印额外信息。同--verbose |
| --verbose | 处理过程中打印额外信息。同-v |
| --version | 打印版本号后退出 |

命令 gcjh 的输入是一个或多个 Java 类文件。例如，下面命令会读出类文件 Spangler.class，创建头文件 Spangler.h，该头文件可用于实现类中的本地 C++ 方法：

```
$ gcjh Spangler
```

下面命令读出文件 Spangler.class，生成文件 Spangler.cc，它可被编辑作为 C++代码和 Java 类的接口：

```
$ gcjh -stub Spangler
```

下面命令读出 Spangler.class 并生成文件 Spangler.h，它是头文件可用于实现 C 语言的本地方法：

```
$ gcjh -jni Spangler
```

下面命令读出文件 Spangler.class 并生成文件 Spangler.c，它可被编辑作为 C 代码和 Java 类的接口：

```
$ gcjh -jni -stub Spangler
```

第 10 章的例子解释如何使用 gcjh 将 C 和 C++代码与 Java 代码相混合。

### 8.2.4 jcf-dump

工具 jcf-dump 可以列出与类文件内容有关的信息。所包含的信息是常量、超类、接口、域和方法的完整列表。表 8-5 列出 jcf-dump 工具可用的选项。

**表 8-5 jcf-dump 的命令行选项**

| 选项 | 描述 |
|---|---|
| --bootclasspath=path | 重写内置类路径设置 |
| -c | 分解方法体的字节码 |
| --classpath=path | 指明路径，用于定位类文件 |
| --CLASSPATH=path | 指明路径，用于定位类文件 |
| --help | 打印选项列表后退出 |
| -Idirectory | 在类路径 classpath 后附带加上指定目录 |
| --javap | 生成输出，格式与 javap 相同。javap 是标准 Sun Microsystems Java 发布版的一部分 |
| -o file | 将输出重定位到指定文件，而不是标准输出设备 |
| -v | 处理过程中打印额外信息。同--verbose |
| --verbose | 处理过程中打印额外信息。同-v |
| --version | 打印版本号后退出 |

例如，下面命令会将类文件 SwmpMilin.class 的内部信息转储到文件 sm.dump 中：

```
$ jcf-dump SwmpMilin.class -o sm.dump
```

### 8.2.5 jv-scan

工具 jv-scan 读出一个或多个 Java 源文件的内容，再对其进行分析，并打印相关信息。表 8-6 列出 jv-scan 可用的命令行选项。

表 8-6　jv-scan 的命令行选项

| 选项 | 描述 |
|---|---|
| --complexity | 打印每个类的 cyclomatic 复杂度的值。这个数字是通过分析流程得到的，会将程序的控制流（control flow）看作有向图（directed graph）并计算节点、边和连接组件的数目 |
| --encoding=name | 指出读源文件时所用的特殊字符集的编码名。如果为本地设置了编码名称，就使用该编码方式；否则，假设使用 UTF-8 编码 |
| --help | 打印选项列表后退出 |
| --list-class | 打印命令行中所有文件里的类名 |
| --list-filename | 当和--list-class 连起来使用时，也会列出包含那些类的文件名 |
| -o file | 将输出重定位到指定文件，而不是标准输出设备 |
| --print-main | 打印包含 Public static void main()方法的类名 |
| --version | 打印版本号后退出 |

### 8.2.6　jv-convert

工具 jv-convert 能够将字符编码从一种格式转换成另一种格式。默认输入为标准输入，但也可以是命令行列出的第一个文件名或由-i 选项列出的文件名。默认的输出结果是标准输出设备，但也可以是命令行中的第二个文件名或由-o 选项列出的文件名。表 8-7 列出了 jv-convert 的命令行选项。例如，下面的命令会将 Unicode-8 编码的文件 PierNun.uni 的内容转换为文件 PierNun.java，它的编码方式为 Java 源代码格式，其中 Unicode 字符使用\u 转义序列（escape sequences）：

```
$ jv-convert --from UTF8 --to JavaSrc PierNun.uni PierNun.java
```

表 8-7　jv-convert 的命令行选项

| 选项 | 描述 |
|---|---|
| --encoding name | 输入数据的编码方案（encoding scheme）名。默认编码方案是本机的现场编码方式。同--from |
| --from name | 输入数据的编码方案名。默认编码方案是本机的现场编码方式。同--encoding |
| --help | 打印本选项列表 |
| -i file | 输入文件名 |
| -o file | 输出文件名 |
| --reverse | 将指定的--from 和--to 的编码方式互相调换 |

（续表）

| 选项 | 描述 |
| --- | --- |
| -to name | 输出数据的编码方案名。默认为 JavaSrc，它是 Java 的 ASCII 文本编码，将非 ASCII 字符按照\uXXXX 的十六进制进行编码 |
| --version | 打印 jv-convert 的版本号 |

表 8-8 列出了可用的编码类型。

表 8-8 jv-convert 可识别的字符编码

| 编码名 | 描述 |
| --- | --- |
| 8859_1 | ISO-Latin-1 (8851-1) 文本 |
| ASCII | 标准 ASCII 字符集 |
| EUCJIS | 扩展的 UNIX 日文代码 |
| JavaSrc | 标准 ASCII 字符集，其中内嵌的 Unicode 字符为 Java 的\uXXXX 十六进制编码 |
| SJIS | Shift JIS 编码，用于日文的微软视窗系统 |
| UTF8 | Unicode 字符编码形式，它将 ASCII 字符按照 8 位实体进行保存 |

没有哪个命令能够列出可用的会话选项。表 8-8 包含了现有的编码选项，但大多数的编码选项肯定都过时了。要找出哪些选项对于编译程序是可用的，需要查看源代码目录树 gcc/libjava/gnu/gcj/ convert 中名为 Input_*.c 和 Ouput_*.c 的文件，可依次用作输入或输出的文件名，这里的星号就是编码方式的名字。会话处理使用 Unicode 作为内部的中间临时格式，这样所有输入/输出对都可一起使用。有些会话是与平台相关的。

### 8.2.7 grepjar

工具 grepjar 会查询整个 jar 文件，试图找到与正则表达式相匹配的字符串，然后会打印文件的名字以及和正则表达式相匹配的实际字符串。jar 文件中的所有文件都被查找，包括清单文件。例如，下面的命令会列出 jar 文件 sayhello.jar 中所有的类，该文件包含方法 main()：

```
$ grepjar main sayhello.jar
```

下面命令会列出在清单文件中指定为 Main-Class 的类：

```
$ grepjar Main-Class sayhello.jar
```

表 8-9 列出 grepjar 的命令行选项。

表 8-9　grepjar 的命令行选项

| 选项 | 描述 |
|---|---|
| -b | 打印匹配文件的字节偏移量 |
| -c | 只打印匹配的字节数目，而不是匹配的每个字节 |
| -e | 如果命令行没有清楚说明匹配模式，该选项可被用于指明匹配模式 |
| --help | 打印选项列表 |
| -i | 确定匹配时，忽略大小写 |
| -n | 打印每个匹配中的文件行号 |
| -s | 禁止打印出错信息 |
| --version | 打印 grepjar 的版本号 |
| -w | 指明与完整的词相匹配的正则表达式模式 |

## 8.3 RMI

远程方法调用（Remote Method Invocation，RMI）工具允许在某个虚拟机上运行的 Java 对象调用其他虚拟机上 Java 对象的方法。这两个虚拟机可以是同一台机器，也可以是不同的机器。参数是序列化的（这种处理过程叫做列集（marshaling）），这样它们可从调用方传递到被调方法，返回值也是顺序返回给调用方。

中心注册处（central registry）包括被调用的激活方法的名字和位置。对象的方式使得调用方无需了解此方法是远程方法。调用对象通过名字调用方法，而本地方法调用叫做存根（stub）。就是通过存根为实际方法在注册处定位、列集参数、传递参数（以及返回地址）到其他位置的框架（skeleton）方法。传输使用 TCP/IP，因此远程虚拟机可位于任何位置。远端机器上，框架方法将参数反列集（unmarshals），再调用实际方法。方法将结果值返回给框架调用函数，它再将结果列集，然后将结果返回给存根。存根将返回值进行反列集处理，再把它返回给原始的调用者。

调用函数的虚拟机是客户端。收到调用的虚拟机是服务器端。处理远程方法调用必须考虑如下的一些特殊情况：

- 因为在处理远程调用的过程中，对象会被创建、列集处理、反列集处理，因而有必要在分布式系统中进行自动垃圾收集（automatic garbage collection）。RMI 使用计数器，每引用一次增加 1，每次释放引用就减 1。这会变得更加复杂，因为返回到调用方的远程对象会包括到其他远程对象的引用。
- 客户端虚拟机保存一个局部计数，记录到远程对象的激活引用的数目。“被引用”的消息就是发送到远程虚拟机的消息。此计数随引用的进出增减，每次变化都会发

送给远程虚拟机。当数目为零时，对象就是可被服务器收集的垃圾。

- 服务器虚拟机保存所有客户端虚拟机以及每个客户端虚拟机对应的激活对象引用的列表。如果对象不再拥有任何远程引用，就可删除该对象。而且每个远程引用都有一个计时器，每次引用对象的时候就会重置该计时器，如果计时器超时，机器的引用计数就会被重置为零。

### 8.3.1 rmic

工具 rmic 是 RMI 存根和框架编译程序。编译程序的输入是编译的类文件，它实现了 java.rmi.Remote 接口，而输出是 Java 存根和框架源文件和编译的类文件。

例如，下面是个非常简单的类，实现了 Remote 接口：

```
/* HelloRemote.java */
public class HelloRemote implements java.rmi.Remote {
    public void speak() {
        System.out.println("hello from remote");
    }
}
```

下面命令会生成存根和框架：

```
$ gcj -C HelloRemote.java
$ rmic HelloRemote
```

第一条命令的输出结果是 HelloRemote.class。第二条命令的输出结果是 HelloRemote_Stub.java、HelloRemote_Skel.java、HelloRemote_Stub.class 和 HelloRemote_Skel.class。编译程序 rmic 调用 gcj 来编译存根和框架。表 8-10 列出 rmic 可用的命令行选项。如表所示，选项都采用单连字符的格式，但写成双连字符格式也是合法的。

**表 8-10 rmic 的命令行选项**

| 选项 | 描述 |
|---|---|
| -classpath path | 用于定位被引用类的类路径 |
| -d directory | 目录名，其中包括生成的存根文件和框架文件 |
| -depend | 检查依赖关系，重新编译所有过时的文件 |
| -g | 生成的文件中包含调试信息 |
| -help | 打印选项列表 |
| -J flag | 将指定的 flag 传递给 Java 编译程序，用于编译存根类和框架类 |
| -keep | 保留产生的中间文件，不删除它们。同-keepgenerated |
| -keepgenerated | 保留产生的中间文件，不删除它们。同-keep |
| -nocompile | 指出不要将生成的存根文件和框架源文件编译成类文件 |

（续表）

| 选项 | 描述 |
|---|---|
| -nowarn | 禁止警告消息 |
| -v1.1 | 生成 Java 1.1 的桩 |
| -v1.2 | 生成 Java 1.2 的桩 |
| -vcompan | 为 Java 1.1 和 Java 1.2 生成桩 |
| -verbose | 打印生成存根文件和框架文件所采取步骤的详细描述 |
| -version | 打印 rmic 编译程序版本号 |

### 8.3.2　rmiregistry

rmiregistry 是后台程序（daemon program），它维护一个虚拟机内部方法列表，这些方法都可用于远程调用。它会监听一个端口（默认端口号为 1099）来接收信息。如果需要使用非默认的端口号，可通过命令行键入该端口号。其他的选项见表 8-11。

表 8-11　rmiregistry 的命令行选项

| 选项 | 描述 |
|---|---|
| --help | 打印选项后退出 |
| --version | 打印 rmiregistry 的当前版本号后退出 |

## 8.4　属性

Java 有一套预定义的系统属性，可从程序内部访问这些属性。每个属性都包括关键字和值，它们都是字符串。为获得属性值，只需知道属性的关键字。例如，下面的方法调用可用于确定用户运行程序的名字：

```
String username = System.getProperty("user.name");
```

以下程序列出所有的系统属性：

```
/* AllProps.java */
import java.util.Properties;
public class AllProps {
    public static void main(String arg[]) {
        Properties properties = System.getProperties();
        properties.list(System.out);
    }
}
```

预定义了30多个属性。列表包括操作系统的名字、Java编译程序的版本、操作系统的版本、用户名、按路径分隔和行分隔的字符等。另外，通过程序或命令行，都可以定义自己的属性。

以下程序显示三个标准系统属性java.vm.version、java.vm.vendor和java.vm.name的值。如果定义了magic，程序也会显示它的值：

```
/* ShowProps.java */
public class ShowProps {
    public static void main(String arg[]) {
        System.out.println(
            "vm.version="+System.getProperty("java.vm.version"));
        System.out.println(
            "vm.vendor="+System.getProperty("java.vm.vendor"));
        System.out.println(
            "vm.name="+System.getProperty("java.vm.name"));
        String magic = System.getProperty("magic");
        if(magic == null)
            System.out.println("There is no magic");
        else
            System.out.println("magic=" + magic);
    }
}
```

将程序编译成可执行代码的时候，使用命令的-D选项可以定义magic属性，如下：

```
$ gcj --main=ShowProps -Dmagic=xyzzy ShowProps.java -o showprops
```

运行程序的结果显示如下：

```
$ showprops
vm.version=3.2 20020412 (experimental)
vm.vendor=Free Software Foundation, Inc.
vm.name=GNU libgcj
magic=xyzzy
```

在编译和运行类文件程序的时候，情况有些区别。属性在运行程序时定义，不是在编译时定义。例如，下面的命令可将源文件编译成类文件：

```
$ gcj -C ShowProps.java
```

下面的命令会按照定义的属性执行此类文件：

```
$ gij -Dmagic=xyzzy ShowProps
```

作为类执行程序的输出结果，和作为二进制可执行程序的运行结果是一样的。

# 第 9 章

# 编译 Ada 程序

GNAT 代表 GNU NYU Ada95 翻译器（Translator），简单地说就是 GNU Ada 翻译器。作为 Ada 编译程序，它已被集成为 GNU 编译程序集合的一部分。

Ada 95 是 Ada 语言最新的标准，GCC 编译程序完全支持该标准。它包括面向对象的编程、继承、多态、动态分发（dispatching），以及 Ada 83 的强制类性（strong typing）。语言本身的标准包括和 C 程序及 Fortran 程序的接口定义。

Ada 作为一种语言，也可作为编译程序，有一些独特需求。最特别的一点，就是 Ada 目标文件可以根据生成它的源文件进行跟踪和校验。在编译和连接过程中，不是只有校验才是其正常的一部分，它还提供了大量的工具程序，通过命令可进行比较和校验。与 GCC 的其他语言不同，Ada 是用 Ada 写成的，因此在系统上安装它的时候，有一些引导（bootstrapping）的操作。

## 9.1 安装

Ada 前端是 GCC 增加的最新功能。在 GCC 3.1 版，已经将 Ada 前端集成为编译程序家族了，它生成的可执行代码能够在不同平台上运行，但不容易移植到新系统中。Ada 前端由 Ada 编写，这样做很有道理，就像 GCC 的 C 语言前端就是由 C 语言编写一样，但这样就造成移植 Ada 的条件状态和其他语言的有所区别。但还有希望的是随着时间推移，Ada 语言会和 GCC 其他语言一样便于移植，不过目前在能够编译 GCC 的 Ada 编译程序之前，需要先安装一个最小化的 Ada 编译程序。

要在系统中安装最新的 Ada 编译程序，首先需要安装最基本的 Ada 编译程序。然后可以使用常规的 GCC 源代码去编译更新的 Ada 编译程序。这样的处理过程一定会随着时间推移变得更加简单，最后在很多系统上安装 Ada 编译程序一定可以和安装 C 编译程序一样简单，但目前只有满足下面步骤才能在所有系统上安装 Ada 编译程序：

1. 下载 Ada 编译程序的二进制可执行代码作为最基本的 Ada 编译程序。以下位置可以找到安装机器支持的版本：

   ```
   http://www.gnuada.org
   ftp://cs.nyu.edu/pub/gnat
   http://www.gnat.com
   ```

   如果已经安装了 Ada 编译程序，可选的方式就是：将 ADAC 环境变量设置为编译程序的名字，并确保该程序名位于 PATH 变量设置的路径中。
2. 遵循安装指导将编译程序下载并安装到机器上。确切的安装步骤和平台密切相关。安装分两步。首先，doconfig 脚本解释安装步骤，询问安装的风格及安装位置，并构造实际的安装脚本 doinstall。然后，执行脚本 doinstall 会完成整个安装过程。
3. 修改变量 PATH，这样在键入命令之后会运行新安装的 gcc 程序。如果已经安装了 gcc 版本，很重要的一点就是要求新目录（包含 Ada 编译程序）必须位于已经安装的 gcc 路径之前。从这点上看，目前 Ada 编译程序的功能已经很齐全，完全可以用它编写程序了，因此只要愿意，就可停在这里开始编写 Ada 程序了。但如果想由 GNU 源代码编译自己的 Ada 编译程序，还要继续下面的步骤。
4. 执行第 2 章的 configure 脚本，但 Ada 语言和 C 语言都应该明确指明。如果以后要加入其他语言，最好一开始只包含这两种语言，因为编译时间非常长，而且如果按照我的经验，也许需要多次重启。下面这一系列命令在 gcc 源代码目录的父目录上运行，在 mybuild 目录建立构造配置（build configuration）。因为--prefix 选项的设置，最终配置会将编译程序的部分安装到目录/usr/gnat/bin、/usr/include、/usr/info、/usr/gnat/lib、/usr/man 和/usr/share 上：

   ```
   $ DIR='pwd'
   $ mkdir $DIR/mybuild
   $ cd $DIR/mybuild
   $ $DIR/gcc/configure --prefix=/usr --enable-languages=c,ada
   ```

   你可能会发现将一系列命令放在一个脚本文件中执行会比较方便。而且无论做什么，总是要激活 C 语言的，因为想构造编译程序而又不激活 C 语言，是不可能编译新版本的编译程序的。
5. 强制要求源文件目录中的特定文件必须要为最新版本，这样确保能够编译 Ada 的启动程序。执行配置脚本之后，touch 命令会更新文件的日期和时间，这样可以确保它们比其他文件更新。能够将这些命令也放进脚本当然更好：

   ```
   $ cd $DIR/gcc/gcc/ada
   $ touch treeprs.ada
   $ touch einfo.h
   $ touch sinfo.h
   ```

```
$ touch nmake.adb
$ touch nmake.ads
```

6. 编译程序帮你启动编译程序：

```
$ cd $DIR/mybuild
$ make bootstrap
```

7. 有必要单独编译 gnatlib。如果不是，也无关紧要。以下命令会编译 gnatlib：

```
$ cd $DIR/mybuild/gcc
$ make gnatlib
```

8. 如果到现在所有步骤全部正确，就可用最后一条命令 make 来安装 Ada 了。因为这样的安装过程会修改一些系统目录，因此很可能需要超级用户权限：

```
$ su
Password: *******
$ cd $DIR/mybuild
$ make install
$ exit
```

   如果浏览安装目录，会发现安装 Ada 之后有些 GCC 文件就重复了。这是正常的，将来的版本一定能够解决此问题，但现在这些是必需的。

9. 最后，保存 PATH 变量。先去掉编译引导程序需要的临时设置，以及其他 Ada 组件，然后再添加新的 bin 目录：

```
$ PATH=$PATH:/usr/gnat/bin
```

## 9.2　编译基础

表 9-1 列出编译和连接 Ada 程序用到的文件名后缀。附录 D 列出了 GCC 可以识别的所有文件名后缀。

**表 9-1　Ada 程序的文件名后缀**

| 后缀 | 文件内容 |
|---|---|
| .a | 库（文档文件）包含静态连接用到的目标文件 |
| .adb | Ada 体文件，为源代码，包括库的单元体 |
| .adc | GNAT 配置文件，用来去掉不可达代码 |
| .ads | Ada 规范文件，为源代码，包括库单元声明或者库单元重命名声明 |
| .adt | GNAT 树文件，用来去掉不可达代码 |

（续表）

| 后缀 | 文件内容 |
|---|---|
| .ali | 由编译程序生成的中间文件，包括一致性检查和连接所需的必要信息 |
| .atb | 文件包含内部树的表示方式，编译程序利用该树表达.adb 文件的内容 |
| .ats | 文件包含内部树的表示方式，编译程序利用该树表达.ads 文件的内容 |
| .o | 目标文件，格式与连接程序相符 |
| .s | 汇编语言代码。这类文件是创建目标文件的中间产物 |
| .so | 库包含动态连接用到的目标文件 |

### 9.2.1 单独源文件到可执行程序

需要参照下面三步创建 Ada 源文件的可执行程序：

1. 将 Ada 源文件编译成目标文件。
2. 必须用 Ada 绑定器（binder）对目标文件（或文件）进行处理。
3. 要创建可执行程序，需要将目标文件（或文件）连接到合适的库。

第 1 步、第 3 步与编译其他语言一样，但第 2 步是 Ada 特有的步骤。绑定器检查目标文件，如下：

- 检查目标文件的一致性，比如选项设置的兼容性以及所使用的编译程序的版本。
- 检验程序的详细执行顺序的有效性。
- 基于确定的详细步骤，生成主线程序。这里有个小小的 C 程序，按照正确的顺序调用每个函数，然后调用主程序。
- 确定组成程序的目标文件的完整集合，包含生成的 C 程序的相关信息。这些信息对 gnatlink 有效，就用它将程序连接成可执行程序。

下面是该简单程序的源代码，在屏幕上显示一行文本：

```
with Text_IO; use text_IO;
procedure HelloWorld is
begin
   Put_Line("hello world");
end HelloWorld;
```

程序保存在文件 helloworld.adb 中，使用下面的命令将它们编译成目标文件：

```
$ gcc -c helloworld.adb
```

选项-c 指出要将程序编译成目标文件，但不将它连接成可执行程序。对于 Ada 必须使用选项-c，因为它的连接处理和其他语言的不同。下面的步骤使用工具 gnatbind 进行绑定操作：

```
$ gnatbind helloworld.ali
```

命令的结果是一对临时工作文件 b~helloworld.adb 和 b~helloworld.ads。不会改动文件 helloworld.ali，它和原始的源文件 helloworld.adb 一样，现在总共有 4 个文件。

最后一步会激活 gnatlink，如下：

```
$ gnatlink helloworld.ali
```

结果为 helloworld 可执行程序。留在磁盘中的还有源文件 helloworld.adb、helloworld.ali 文件和目标文件 helloworld.o。

可以按照另一种方式编译和连接 Ada 程序。工具 gnatmake 的限制条件和 make 的相似，可以确定需要编译哪个文件，然后会激活编译程序 gnatbind 和 gnatlink，产生的结果和分别使用三个命令的都一样。下面一条命令产生的 4 个文件和前面三个命令组合产生的 4 个文件一样：

```
$ gnatmake helloworld.adb
```

如果不提供文件后缀，工具 gnatmake 会自动附加上后缀.adb，这样的过程更简单，可以使用同样的命令，如下：

```
$ gnatmake helloworld
```

### 9.2.2 多源文件到可执行程序

可将过程的集合定义为包。文件 howdy.abs 包括包 Howdy 的规范说明，它包括过程 Hello 和 Goodbye：

```
package Howdy is
   procedure Hello;
   procedure Goodbye;
end Howdy;
```

过程体本身定义在文件 howdy.adb 中，如下：

```
with Text_IO; use Text_IO;
package body Howdy is
   procedure Hello is
   begin
      Put_Line("Howdy from package");
   end Hello;
   procedure Goodbye is
   begin
      Put_Line("Goodbye from package");
   end Goodbye;
end Howdy;
```

程序使用包 Howdy 中的过程来显示文本，该程序保存在文件 howdymain.adb 中：

```
with Howdy;
procedure HowdyMain is
begin
   Howdy.hello;
   Howdy.goodbye;
end HowdyMain;
```

工具 gnatmake 理解这种组织结构，并编译源文件以实现整个程序。下面命令由源文件产生可执行程序：

```
$ gnatmake howdymain
```

键入下面这一系列命令和此命令的结果相同：

```
$ gcc -c howdymain.adb
$ gcc -c howdy.adb
$ gnatbind -x howdymain.ali
$ gnatlink howdymain.ali
```

结果会创建新文件 howdy.ali、howdymain.ali、howdy.o、howdymain.o 和可执行程序 howdymain。运行程序 howdymain，输出结果如下：

```
Howdy from package
Goodbye from package
```

### 9.2.3 源代码到汇编语言

选项-S 指示 gcc 由源代码生成汇编语言后停止。下面命令可产生 Ada 源文件 helloworld.adb 的汇编语言文件 helloworld.s：

```
$ gcc -S helloworld.adb
```

汇编语言文件的内容和编译程序的目标平台关系很大。如果命令行中有多个源文件，会为每个源文件生成独立的汇编语言文件。

## 9.3 选项

所有命令行选项都列举在附录 D 里，但大多对 Ada 都没有什么特殊意义。表 9-2 列出的命令行选项对所有语言都有效，但对 Ada 语言有特殊意义。

表 9-2　属于 Ada 的基本命令行选项

| 选项 | 描述 |
|---|---|
| -c | 指明要将源文件编译成目标代码，但不连接生成可执行程序。在编译 Ada 程序时需要该选项，因为 gcc 不会激活 gnatbind 和 gnatlink |
| -fno-linline | 禁止所有内嵌函数，而不论设置的是多高级别的优化操作 |
| -g | 目标文件中包含调试信息，由连接程序复制到可执行程序，且对调试器（debugger）也可用 |
| -Idirectory | 将指定目录加入目录列表中，就是在这些目录里查找编译程序所需的源文件 |
| -I- | 指出不在同一目录里查找其他源文件，该目录是命令行中编译文件所在的目录 |
| -O[n] | Ada 的优化级别和其他语言一样，附录 D 中有所描述，包括 n=3，它允许自动内嵌优化 |
| -S | 生成汇编语言输出 |
| -v | 显示 GCC 当前版本，显示 gcc 驱动器生成的所有命令 |
| -Vversion | 运行指定版本的 gcc 编译程序 |
| -Wuninitialized | 为所有未初始化的变量都生成警告消息。只有同时指定了-O 选项，本选项才有效 |

除了表 9-2 基本的命令行选项之外，表 9-3 列举的选项集合只适用于 Ada。这些专用于 Ada 的选项都以-gnat 这五个字符开头。每个选项都由一个或两个字符所定义，并可以单独指定。如下例所示，它要求按照详细模式进行说明，同时还激活了动态检查：

```
$ gcc -gnatv -gnatE -c helloworld.adb
```

这对选项可以组合说明，如下：

```
$ gcc -gnatvE -c helloworld.adb
```

表 9-3　Ada 专用的命令行选项

| 选项 | 描述 |
|---|---|
| -gnat83 | 指出程序按照 Ada83 标准进行编译。本选项主要用于将源代码移植到 Ada83 编译程序中。默认为-gnat95 |
| -gnat95 | 根据 Ada95 标准编译源代码。这是默认模式 |
| -gnata | 激活告警程序（pragma Assert）和调试程序（pragma Debug）。如果没有指定该选项，源文件中所有关于 pragma 的设置都会被忽略 |
| -gnatb | 所有错误都会在标准输出设备上显示相关错误信息的概括提示，同时详细的出错信息会包含在列表中 |
| -gnatc | 编译程序进行详细地检查，但不生成输出文件，只会生成可能出现的错误和警告信息 |

（续表）

| 选项 | 描述 |
| --- | --- |
| -gnatdxx | 该选项可被用作提取编译过程调试编译程序本身所产生的有关信息。值 xx 是一个或多个字母数字的组合，说明提取的调试信息类型。共有 65 种可用代码（大写字母、小写字母和数字 1 到 9）。很少用到它们，可在源文件 debug.adb 的注释中找到有关描述，该文件是编译程序的一部分 |
| -gnate | 一遇到错误，就会生成出错信息，而不是将这些错误保存起来，直到结束才总结报告编译过程中的所有错误。这样报告出错信息看起来可能没什么顺序，但可以防止由于编译程序崩溃而丢掉所有信息 |
| -gnatE | 在子程序调用和基本实例化过程开始之前，激活动态访问检查 |
| -gnatf | 编译程序生成的错误信息可能是冗余的。例如，一般当发现某个变量没有定义就会产生错误信息，但是该选项会在每次引用该变量时都生成出错信息 |
| -gnatg | 强制风格，由源文件 style.adb 中的例程（routine）定义，该程序是编译程序的一部分。所强置风格的元素在文件中作为注释被记录起来。通常该选项只用于编译编译程序本身的单元 |
| -gnatich | ch 值是单独一个字符，指出编译程序可识别的字符集。所有从字符集中选出的字符都可用作字符字母或用于标识。ch 值可为下面的任意值：<br>1：Latin-1 字符集。字符值从 0 到 127 是标准的 ASCII 字符。值 128 到 255 代表额外的欧洲字母字符，例如德语中的元音字母都有变音符，或者是瑞典语中的 A-ring。该字符集是默认字符集。<br>2：Latin-2 字符集。<br>3：Latin-3 字符集。<br>4：Latin-4 字符集。<br>P：IBM PC（代码见相关文档的 437 页）字符集。它和 Latin-1 字符集相似，但值 128 到值 255 的编码不同。<br>8：IBM PC（代码见相关文档的 850 页）字符集。它是对 437 页代码的修改，扩充以包含所有 Latin-1 字母，但编码方式和 Latin-1 不同。<br>F：128 和 255 间的所有字符代码都是合法的，每个值都有区别。因此可以自己定制字符集（通常中文字符就是这样处理的）。<br>H：128 和 255 间的所有字符代码都是非法的。它是 Ada83 兼容格式。 |
| -gnatjch | ch 值是单独一个字符，指出字母字符串和标识中宽字符（wide character）的格式。ch 值可为下面的任意值：<br>N：未指定宽字符格式。这是默认设置。 |

（续表）

| 选项 | 描述 |
| --- | --- |
| | H：十六进制编码。每个宽字符都由一个五字符序列（five-character sequence）表示。第一个字符是 ESC，后面四个大写十六进制数字代表 16 位的字符代码值。 |
| | U：半大写编码（Upper half encoding）。第一位置为 1 指出这是 16 位宽字符值中的第一个字节。宽字符的十六进制值从 16#8000#到 16#FFFF#。注意，禁止使用 Latin-1 字符集的半大写编码格式。 |
| | S：Shift JIS 编码。它和半大写编码相似，除了它要求所有宽字符都要写作两个独立的字符。第一个值有自己的大写位集合，范围从 16#80#到 16#FF#，第二个值的范围从 16#00#到 16#FF#。注意，禁止使用 Latin-1 字符集的半大写编码格式。 |
| | E：EUC 编码。它和半大写编码相似，除了它要求所有宽字符都要写作两个独立的字符，而这两个值都有自己的大写位集。这两个值的范围都在 16#80#和 16#FF#之间。注意，禁止使用 Latin-1 字符集的半大写编码格式。 |
| -gnatkn | n 值是 1 到 999 之间的数字，指出文件名允许的最大长度（不包括.ads 或.adb 扩展） |
| -gnatl | 列出完整的源文件，还有行号，以及其中包含的所有出错信息，格式由-gnatv 选项指定 |
| -gnatmn | 指出编译程序输出的出错信息的最大数目。n 值范围从 1 到 999。例如，-gnatm3 允许在终止编译之前输出 3 条出错信息。默认对信息数目不作限制 |
| -gnatn | 当 pragma inline 指明的情况下，激活同一单元以及交叉编译单元的内嵌函数。该选项只有在-O 优化标识也被指定的条件下，才能生效 |
| -gnatN | 与-gnatn 相似，除了一点，它会假设所有源文件都有内嵌程序（pragma inline） |
| -gnato | 激活对整数操作溢出的检查，该检查要在运行时进行。因为要插入测试，对所有整数的溢出条件以及除零的情况进行检查，代码会变得更大，运行起来也更慢 |
| -gnatp | 禁止创建运行时的检查，就好像源程序包含了 pragma Suppress(all_checks)一样。以防止非法数据为代价，该选项可以提高效率 |
| -gnatq | 该选项强制编译程序生成输出文件，即使源代码出现语法错误。这可能导致更多错误，也可能令编译程序崩溃或生成未定义行为的代码 |
| -gnatr | 该选项将源代码的展开格式和 Ada 语言引用菜单定义的源代码展开格式相匹配。两种格式之间的冲突视作语法错 |
| -gnats | 对源代码进行语法检查，然后中止（halt）。不会产生输出结果。使用该选项时，命令行指定多个源文件是合法的（尽管同时还要指定-c flag） |
| -gnatt | 编译程序会将内部树写到文件中。该文件和源文件的基础名相同，后缀.atb 为源文件体，而.ats 为规范的源文件 |

（续表）

| 选项 | 描述 |
|---|---|
| -gnatu | 编译程序将目前编译单元依赖的所有单元列表打印到标准输出设备上，不论是直接依赖关系，还是间接依赖 |
| -gnatv | 出错信息格式化以包含更多信息。默认格式包括文件名、行号、列号以及描述性信息，如下：<br>`hlowrld.adb:2:01: incorrect spelling of the keyword "procedure"`<br>使用-gnatv 选项，格式更像如下样子：<br>`Compiling hlowrld.adb (source file time stamp 2002-05-13 20:00:29)`<br>`2. proccedure HelloWorld is`<br>`\|`<br>`>>>incorrect spelling of keyword "procedure"` |
| -gnatwe | 所有警告消息都看作错误处理。发出的消息不会有所改动，但任何警告都会禁止生成目标文件 |
| -gnatwl | 发出与详细操作顺序有关的警告消息 |
| -gnatws | 禁止输出所有警告消息 |
| -gnatwu | 对所有定义但未使用的条目都发出警告消息。如果包中有成员只被定义而未被使用，也会发出警告消息。对 with 语句中所有未引用的内容都要发出警告 |
| -gnatx | 禁止交叉引用信息，通常包含在.ali 文件中。节约了一些空间，但如果工具需要某些信息，比如 gnatfind 和 gnatxref，就不能使用该选项 |

## 9.4 Ada 工具

Ada 编译程序包含了大量的工具。有些用于开发 Ada 程序，例如 gnatbind 和 gnatlink，而有些只应用于某些特殊环境。这些工具提供了很多分析 Ada 源代码的方法。当工作项目很大，或在浏览别人编写的代码时，这些工具尤其重要。

### 9.4.1 gnatbind

工具 gnatbind 执行 Ada 绑定操作，包括以下内容：

1. 检查程序一致性，对各种模块之间的不一致发出错误消息。
2. 确定详细步骤的顺序是否一致，如果顺序不一致就发出错误消息。

3. 生成一个小小的 C 程序，作为最后连接成可执行程序的主线。程序首先调用例程初始化包，然后调用 Ada 程序的主线。
4. 确定一列目标文件，它们可以组合起来成为最后的可执行程序。该列表要插到生成的 C 程序当中，这样它们对 gnatlink 就是有效的了。

工具 gnatbind 需要使用编译程序产生的.ali 文件作为它的输入文件。gnatbind 还会扫描其他的.ali 文件和源文件，检查它们的一致性。如果程序依赖的任何文件的源代码有所改动且没有经过编译，gnatbind 会检测它的状态并报告该情况。

将所有程序模块绑定在一起的结果就是整个程序的源代码。程序的默认名会和输入的原始.ali 文件同名，除了两个新的以 b~开头的 Ada 文件，一个以.ads 为后缀，另一个以.adb 为后缀。或者用-C 选项可以生成 C 语言源文件，以.c 为后缀。

表 9-4 列出了 gnatbind 可用的命令行选项。

**表 9-4　gnatbind 工具的命令行选项**

| 选项 | 描述 |
|---|---|
| -aI directory | 指出用于查找源文件的目录名 |
| -aO directory | 指出用于查找.ali 文件的目录名 |
| -b | 产生简要的标准错误出错信息，甚至在设置了-v 选项将出错信息重定向到标准输出设备上，也可以使用该选项 |
| -c | 不产生输出文件，但会处理输入文件，也会产生所有的出错消息 |
| -C | 输出文件是 C 语言源文件，而不是 Ada 源文件 |
| -e | 将完整的列表打印到标准输出设备上，该列表为详细的顺序依赖关系，包括每种依赖关系的理由 |
| -E | 在出现异常（Exception）对象的时候，保存回溯（trace-back）信息 |
| -h | 将本表描述的选项简要地打印到标准输出设备上 |
| -I directory | 指出查找源文件和.ali 文件的目录名 |
| -I- | 指出不要在当前目录查找源文件，也不要在命令行中.ali 文件所在目录查找其他.ali 文件 |
| -K | 将传递给连接程序（linker）的选项打印到标准输出设备上。该列表与生成的.adb 文件中的选项列表一致 |
| -l | 将选中的详细操作顺序打印到标准输出设备上 |
| -Lxxx | 为了创建库（没有主线的 Ada 程序），名为 adainit 和 adafinal 的程序要更名为 xxxinit 和 xxxfinal |
| -mnumber | 限制报告错误信息的最大数目。该数值范围从 1 到 999。一旦达到该数字，gnatbind 就会退出处理过程 |

（续表）

| 选项 | 描述 |
|---|---|
| -Mxxx | 将生成的主程序 main 更名为 xxx |
| -n | 没有主程序。（即主程序不是由 Ada 语言编写的） |
| -nostdinc | 指出不在系统默认目录中查找源文件 |
| -nostdlib | 指出不在系统默认目录中查找库文件 |
| -o filename | 指定输出文件名，替换默认的 b_name.c，这里的 name 是输入文件的基础名 |
| -O | 打印列表，其中包括完成连接操作所需的所有对象 |
| -p | 指定使用悲观（最坏情况 worst-case）的详细处理顺序 |
| -r | 将应用的额外 pragma 约束列表打印到标准输出设备上 |
| --RTS=*dir* | 指定 dir 作为默认查找源文件和目标文件的目录 |
| -s | 所有源文件必须都存在，并要接受一致性检查。通常 gnatbind 会忽略所有缺失源文件，但该选项要求与主编译单元有关的所有源文件都必须存在 |
| -S*xx* | 指出初始化标量值（scalar values）的方式。将 xx 指定为 in 会将标量值初始化为该数据类型的无效值。指定 lo 会将其初始化为最低值，而 hi 会将其初始化为最高值。其他字符对会解释为十六进制数字，按字节初始化 |
| -shared | 指出连接使用的共享运行时库 |
| -static | 指出连接使用的静态运行时库 |
| -t | 将时间戳（Timestamp）出错消息看作警告。它的效果会去掉文件一致性检查 |
| -Tnnn | 将时间片（time slice）值设为 *nnn* 毫秒，这里的 *nnn* 是大于零的整数 |
| -v | 生成详细的出错消息，将其重定位到标准输出设备上，而不是默认的标准错误设备 |
| -we | 将所有警告消息都看作是致命错误 |
| -ws | 禁止所有警告消息 |
| -x | 不检查源文件。只对.ali 文件进行一致性检查。它运行起来更快，但改动源文件可能发生未知现象。在内部使用 make 程序的描述文件更合理，因为它不会在编译到运行 gnatbind 期间改动源代码。工具 gnatmake 使用该选项激活 gnatbind |
| -z | 没有主子程序（main subprogram） |

gnatbind 要执行自己的检查任务，必须要能够查找到程序中所有的源文件和.ali 文件。按照下面的顺序查找文件：

- 由命令行指定.ali 文件所在目录，该目录也许是当前目录也许不是。如果指定了选项-I-，就会跳过该目录。
- 命令行的-I 选项指定所有目录。

- 环境变量 ADA_INCLUDE_PATH 列出源文件（不是.ali 文件）的目录。它是逗号分隔的目录名列表（PATH 环境变量的格式与此相同）。
- 环境变量 ADA_OBJECTS_PATH 列出.ali 文件（不是源文件）的目录。它是逗号分隔的目录名列表（PATH 环境变量的格式与此相同）。
- Ada 编译程序的默认安装目录，在安装编译程序的时候指定该目录。

### 9.4.2 gnatlink

工具 gnatlink 将 Ada 目标文件连接成可执行程序。该程序是通过 gcc 程序激活连接程序的前端，将它用于正确的目标文件和库文件列表。它使用 gnatbind 输出的文件决定如何处理连接。

gnatlink 需要的大多数信息都保存在 gantbind 的输出文件中，所以有一些命令行参数见表 9-5。各种元素在命令 gnatlink 上出现的顺序非常重要。下面是命令行的总体布局：

```
$ gnatlink [options] mainprog.ali [non-ada object] [linker options]
```

首先是 gnatlink 选项，然后是主线程序的.ali 文件。跟在后面的是其他非 Ada 语言生成的目标文件，这些目标文件也是最后可执行程序的一部分。目标文件后的所有命令行选项都要传递给连接程序，用于构造最后的可执行程序。

**表 9-5　gnatlink 的命令行选项**

| 选项 | 描述 |
|---|---|
| -A | 该选项令 gnatbind 生成的中间产物为 Ada 程序的源文件，这是默认选项 |
| -b target | 由 gnatbind 产生的源文件会被编译并运行在指定目标机器上 |
| -B directory | 从指定目录载入用于编译和连接的可执行程序 |
| -C | 该选项令 gnatbind 产生的中间产物为 C 程序的源文件 |
| -f | 打印连接的目标文件列表 |
| -g | 该选项包括调试信息，不会删除 gnatbind 生成的临时工作文件 |
| --GCC=name | 指出编译程序的前端名，默认为 gcc |
| --LINK=name | 指出连接程序的前端名，默认为 gcc |
| -n | 指出不要编译 gnatbind.生成的文件 |
| -o | 指定连接生成的可执行文件的名字 |
| -v | 详细模式。可两次使用该选项，这样代表更详细的模式 |

### 9.4.3 gnatmake

gnatmake 工具程序和标准 make 工具很相似，但它是为 Ada 定制的，有些特殊需求。利用 gnatmake 键入程序主线的源文件名，就可将程序编译并连接成可执行代码。它会检查

所有源文件以确定还需要什么样的源文件和目标文件，而且会对照源文件检查所有目标文件来确定是否需要编译这些目标文件。

工具 gnatmake 有很多选项，如表 9-6 所示。有些由 gnatmake 使用，但大多数选项会传给 gcc、gnatbind 或 gnatlink。注意选项-P、-vPx 和-Xnm 都可参见 project file（项目文件），它是 Emacs 编辑器的特殊功能（版本 20.2 或更新版本），可以编辑并维护项目文件来设置和控制编译过程。

**表 9-6 gnatmake 的命令行选项**

| 选项 | 描述 |
|---|---|
| -a | 考虑所有输入文件，包括所有只读的.ali 文件。默认设置中，.ali 文件是写保护的，gnatmake 对它不加检查 |
| -aIdirectory | 将指定的目录加入列表，就是在该列表的所有目录里查找源文件 |
| -aLdirectory | 会假设指定目录中的.ali 文件也会应用于其他源文件，所以 gnatmake 不会对它进行校验并编译它们。这与将.ali 文件设置成写保护是一样的 |
| -aOdirectory | 将指定的目录加入列表，就是在该列表的所有目录里查找库文件和目标文件 |
| -Adirectory | 这和同时指定-aLdirectory 和-aIdirectory 是一样的。 |
| -bargs list | 命令中-bargs 之后的选项会传递给 gnatbind。它们可以是表 9-4 中的任意选项 |
| -c | 指出只进行编译操作。不会激活 gnatbind 和 gnatlink。如果命令指定的源文件不是主线（mainline），该选项就是默认设置 |
| -cargs list | 命令中-cargs 之后的选项会传递给编译程序。它们既可是表 9-2 列举的有关 Ada 的任意选项，也可是附录 D 中描述的基本选项 |
| -f | 强制重编译所有源文件，而不考虑目标文件的时间戳（timestamp） |
| --GCC=name | 使用指定名字作为编译程序的前端名。默认为 gcc |
| --GNATBIND=name | 使用指定名字作为绑定命令名（binder command）。默认为 gnatbind |
| --GNATLINK=name | 使用指定名字作为连接程序命令名（linker command）。默认为 gnatlink |
| -i | 指定所有编译操作都应取代合适的已存在的.ali 文件。如果不存在.ali 文件，会在源文件所在目录创建该文件。默认只在当前目录创建新文件 |
| -Idirectory | 这和同时指定-aIdirectory 和-aOdirectory 是一样的 |
| -I- | 指出在命令行指定源文件的目录中，不会查找其他源文件 |
| -jnumber | number 用于指定同时编译和重编译的最大进程数目。来自不同编译进程的消息混在一起 |
| -k | 指明出现错误条件以后，继续编译。该选项会编译所有源文件，并在 gnatmake 终止前，列出失败的文件 |
| -largs list | 命令中-largs 后的选项会传递给 gnatlink。它们可以是表 9-5 列举的任意选项 |

（续表）

| 选项 | 描述 |
| --- | --- |
| -Ldirectory | 将指定的目录加入列表，就是在列表中的所有目录里查找库文件 |
| -m | 将重编译的数目保持到最小。如果只改动注释行或代码的文本格式，该选项就会忽略时间戳的差别 |
| -M | 将文件依赖关系打印到标准输出设备上，其格式与插入make程序的描述文件的格式相符。除非同时指定-q选项，否则所有文件都列出其绝对路径或相对路径名。除非同时指定-a选项，否则会忽略系统依赖关系。不包含到外部库的依赖关系 |
| -n | 禁止编译、绑定及连接步骤。该选项只会检查所有目标文件，看它们是否都是最新的。如果不是最新版本，就会列出需要编译的第一个文件名 |
| -nostdinc | 指出不会在系统默认目录里查找源文件 |
| -nostdlib | 指出不会在系统默认目录里查找库文件 |
| -o name | 指定可执行文件名。默认使用输入文件名 |
| -P name | 使用命名的项目文件 |
| -q | 按照简洁（quiet/terse）模式进行处理。不会显示gnatmake发出的命令 |
| -s | 重新编译所有改动编译选项设置的文件 |
| -u | 只编译指定文件，忽略所有可能过时的依赖关系 |
| -v | 按照详细（verbose）模式进行处理。显示原因，解释所有编译和重编译的必要性 |
| -vPx | 在使用项目文件控制编译时，按照详细模式进行处理 |
| -Xnm=value | 该选项值为项目文件使用的外部引用 |
| -z | 没有主子程序，因此不可能将目标文件连接成最终的可执行文件 |

因为选项-cargs、-bargs 和-largs 后可以跟随其他很多相关选项，因此必须将它们看作是命令行最后的成员。命令行gnatmake的基本语法如下：

```
$ gnatmake [options] filename [-cargs ...] [-bargs ...] [-largs ...]
```

命令行中的文件名既可指定为.abs后缀，也可指定为非.abs后缀。

选项-cargs之后的选项列表会传递给编译程序。该列表由选项-bargs或选项-largs终止，当然结束命令行也能终止列表。这三种选项列表之间没有顺序。-bargs 之后的选项会传给绑定器，而-largs选项都会传给连接程序。

### 9.4.4 gnatchop

工具gnatchop读出源文件，然后将每个源文件写入一个或多个新的源文件中，这些新文件严格遵守GNAT Ada文件命名规范。编译程序要求每个文件只包含一个编译单元，而且编译单元名和文件名之间必须严格遵守对应关系。gnatchop 工具允许一次就可转换所有

源文件。当然也可以通过编译命令列表（如 make 程序的描述文件所示），在每次编译程序时建立文件名规则。

命令行 gnatchop 具有下面的基本格式：

```
$ gnatchop [options] file [file ...] [directory]
```

使用该命令，指定的一个或多个文件可被分块（chopped），结果是在当前目录中保存了新生成的一个或多个文件，如果指定其他目录，也可将结果文件保存在指定目录。选项参见表 9-7。

表 9-7 gnatchop 的命令行选项

| 选项 | 描述 |
|---|---|
| -c | 激活编译模式，配置程序（pragmas）块的设置满足 Ada95 标准的规则 |
| -gnatxxx | 所有指定的-gnat 选项都要传递给解析器（parser） |
| -k[number] | 生成的文件名长度不能超过指定数字。如果没有指定数字，默认为 8 |
| -q | 简洁模式禁止输入输出文件名的普通列表 |
| -r | 在输出文件中包含 Source_Reference 程序（pragmas）。如果输出文件是临时工作文件，就可以使用该选项——编译程序会在错误和警告消息中用到 pragma 信息，令消息指向源文件而不是分块的文件。插入目标文件的调试信息以及-g 选项会引用到原始文件 |
| -v | 详细模式，这里生成的所有命令都会回显在标准输出设备上 |
| -w | 如果有必要生成输出结果，就会重写所有存在的文件。通常 gnatchop 不会替换已经存在的文件 |
| -x | 指出遇到错误立即退出 |

### 9.4.5 gnatxref

工具 gnatxref 读出编译程序保存在.ali 文件中的信息，并将它们显示出来。命令 gnatxref 的语法如下：

```
$ gnatxref [options] file [file ...]
```

列表中的所有文件都是.ali 文件，输出的结果是包和过程的列表，以及声明它们的位置、过程体和所有引用，按照字母排序。表 9-8 列举了所有选项。

表 9-8 gnatxref 命令行选项

| 选项 | 描述 |
|---|---|
| -a | 考虑所有文件。通常不包含.ali 只读文件的内容 |
| -aIdirectory | 将指定目录加入列表，就是在该列表所有目录中查找输入源文件 |

（续表）

| 选项 | 描述 |
| --- | --- |
| -aOdirectory | 将指定目录加入列表，就是在该列表所有目录中查找输入库文件和目标文件 |
| -d | 将继承的类型信息作为交叉引用的一部分 |
| -f | 按照完整的路径名显示交叉引用列出的文件，而不是只显示它们的简单文件名 |
| -g | 限制交叉引用中的符号，只能是库级别的项目。忽略局部项目 |
| -Idirectory | 这和同时指定-aIdirectory 和-aOdirectory 是一样的 |
| -pfilename | 指定项目文件名。默认条件下，gnatxref 会在当前目录定位项目文件 |
| -u | 只包括输出文件未使用的符号 |
| -v | 不是交叉引用，输出文件的文本是标签文件（tags file）格式，可用 vi 编辑器进行编辑 |

### 9.4.6 gnatfind

工具 gnatfind 读出.ali 文件的信息，并为命令行指出的项目定位。输出的是一个列表，其中包括找到指定项目的所有位置。命令行的语法如下：

```
$ gnatfind [options] pattern[:filename[:line[:column]]] [file ...]
```

指定的模式 pattern 是 grep 工具使用的正则表达式的子集。包括星号（*）代表任意字符组，问号（?）代表任意单个字符，而标准的[...]括号构造的模式与指定的字符集相匹配。而且从命令行语法可以看出，可以限制查询的文件，甚至可以指定特定的行和列。如果命令列出一个或多个文件，就只会查询这些文件。

表 9-9 列出命令行选项。

**表 9-9　gnatfind 命令行选项**

| 选项 | 描述 |
| --- | --- |
| -a | 考虑所有文件。通常不包括.ali 只读文件的内容 |
| -aIdirectory | 将指定目录加入列表，就是在该列表的所有目录里查找输入源文件 |
| -aOdirectory | 将指定目录加入列表，就是在该列表的所有目录里查找输入库文件和目标文件 |
| -d | 将继承的类型信息作为输出的一部分 |
| -e | 接受完整的正则表达式语法，而不仅仅是简单的星号、问号和方括号。完整的正则表达式语法包括下面的字符，可用作有效的操作符：<br>[ ] . * + ? ^ |
| -f | 输出结果列出文件的完整路径名，而不是默认显示的简单名 |
| -g | 限制输出的符号只是库级别项目。忽略局部项目 |
| -Idirectory | 这和同时使用-aIdirectory 和-aOdirectory 是一样的 |

（续表）

| 选项 | 描述 |
|---|---|
| -pfilename | 指定项目文件名。默认条件下，gnatxref 会在当前目录定位项目文件 |
| -r | 定位并列出所有引用。默认只列举引用声明 |
| -s | 打印找出项目的完整的源代码行，而不是只列出它的位置 |
| -t | 打印每个找到项目的类型继承关系 |

### 9.4.7 gnatkr

工具 gnatkr 可以给出指定 Ada 名的缩短后的形式。尽管工具 gnatkr 在缩短名字的时候要遵守一些原则，还是不能保证缩短后的名字能够惟一。缩短后的文件名默认为 8 个字符，但用下面的命令语法也可指定其他长度：

```
$ gnatkr name [length]
```

通过将名字分成几个部分可将名字缩短，不同部分之间用连字号和下划线连接，依次将每部分缩短，直到达到要求的长度。见下面的例子：

```
$ gnatkr longer-names-can-be-crunched
lncabecr
$ gnatkr The_Ada_Names_Are_Long
tanaarlo
$ gnatkr The_Ada_Names_Are_Long 5
tanal
```

### 9.4.8 gnatprep

工具 gnatprep 用作简单的 Ada 源代码预处理程序。命令行需要指定输入文件和输出文件，所有预处理定义必须定义在第三个文件中，或在命令行上直接指定。命令语法如下：

```
$ gnatprep inputfile outputfile [definitionsfile] [options]
```

inputfile 和 outputfile 都是必需的，而且必须指定全文件名（包括文件后缀）。因为 outputfile 通常都是要被编译的文件，一般的后缀为.adb 或.ads。表 9-10 列举了命令行选项。选项 definitionsfile 应该按照下面的格式，包括一个或多个符号定义：

```
symbol := value
```

表 9-10 gnatprep 命令行选项

| 选项 | 描述 |
|---|---|
| -b | 将所有预处理行替换为空行。默认是要删除这些行 |
| -c | 在输出源文件中将预处理行保留，作为注释行。它们都由字符串"-!"标识 |

（续表）

| 选项 | 描述 |
| --- | --- |
| -Dsymbol=value | 将符号定义为指定值，就好像在定义文件中加入这样一行 symbol := value |
| -r | 生成 Source_Reference 程序（pragma），这样所有的出错消息和调试信息才能跟踪回原始文件。除非同时指定-c 选项，该选项用-b 保持行号的一致性 |
| -s | 打印定义符号的排序列表，其中包括名字和它们的值 |
| -u | 在#if 指示字中，该选项将未定义的符号当作定义为假 false |

定义的值可为空白、引号包围的串、或有效 Ada 字符的任意子集。和 C 预处理程序不同，gnatprep 不会替代找到的所有匹配。会替代的符号必须由$符号标记。例如，假设定义文件包括下面这行：

```
bracklin := thermolimit
```

这条命令会用 thermolimit 去替换输入源文件中每次出现串$bracklin 的位置。而且通过指示字#if、#elsif 和#end if 测试定义的符号为真 true 或假 false，可进行条件编译，如下：

```
#if condrep then
   Put_Line("condrep is defined as true");
#else
   Put_Line("condrep is defined as false");
#end if;
```

可使用 not 操作符将前一条语句的逻辑逆转，如下所示：

```
#if not condrep then
   Put_Line("condrep is defined as false");
#else
   Put_Line("condrep is defined as true");
#end if;
```

### 9.4.9 gnatls

工具 gnatls 是库浏览器，用于展开并显示与编译单元相关的信息。它显示了目标、单元名和源文件之间的关系。还可用它确定编译单元的源代码依赖关系。输入文件可以是编译程序生成的.ali 或.o 文件。

输出的默认格式包括四列。第一列为要分析的目标文件名，第二列为目标文件主单元（principal unit）的名字，第三列是源文件的状态，第四列为源文件名。可能的源文件状态值见表 9-11。

表 9-11 gnatls 为源文件分配的状态代码

| 状态 | 定义 |
|---|---|
| ??? | 没有找到源文件 |
| DIF | 至少找到一个源文件相匹配，但没找到和目标文件版本相匹配的源文件 |
| HID | 源文件版本和目标文件版本相匹配，但至少存在一个源文件的版本不匹配（先找到该文件）。匹配的源文件被有效地隐藏起来 |
| MOK | 在处理目标文件的时候，源代码有些轻微变动，但这种变动不会引起重新编译。改动主要在格式或注释上 |
| OK | 目标文件是最新的，与源文件完全匹配 |

gnatls 的命令行选项见表 9-12，允许定制输出的内容和形式，以及指定查找路径。

表 9-12 gnatls 的命令行选项

| 选项 | 描述 |
|---|---|
| -a | 将相关预定义单元加到输出信息中。列出所有单元，包括 Ada 库中的预定义单元 |
| -aIdirectory | 将指定的目录加入源文件查找路径 |
| -aOdirectory | 将指定的目录加入目标文件的查找路径 |
| -d | 将与命令行中指定文件编译相关的源文件加入文件名输出列表中 |
| -h | 打印命令行选项列表 |
| -Idirectory | 这和同时指定-aIdirectory 和-aOdirectory 是一样的 |
| -I- | 指明不在系统默认目录中查找源文件或目标文件 |
| -nostdinc | 指明不在系统默认目录中查找源文件 |
| -o | 限制目标文件信息的输出 |
| -Pname | 使用指定的项目文件 |
| -s | 限制源文件信息的输出 |
| -u | 限制编译单元信息的输出 |
| -v | 生成详细的输出，包括源文件和目标文件完整的路径名。还有，所列文件都附带有描述性词语，如下：<br>• Elaborate_Body：本单元包括程序（pragma）Elaborate_Body。<br>• No_Elab_Code：编译该单元不会生成详细代码。<br>• Predefined：本单元是预定义环境的一部分，用户不能修改。<br>• Preelaborable：本单元是预详细化的（preelaborable），Ada 95 标准对此有所定义。 |

（续表）

| 选项 | 描述 |
| --- | --- |
| | • Pure：本单元为纯单元，Ada 95 标准对此有所定义。<br>• Remote_Call_Interface：本单元包括杂注 Remote_Call_Interface。<br>• Remote_Type：本单元包括杂注 Remote_Type。<br>• Shared_Passive：本单元包括杂注 Shared_Passive。 |
| -vPnumber | 将项目文件报告详细级别设置为 0、1 或 2 |
| -Xsymbol=value | 指定外部值 |

### 9.4.10　gnatpsys 和 gnatpsta

工具 gnatpsys 输出的是 Ada 包的源代码，其中包括所有和系统有关的尺寸及特征。它包含一些系统定义，例如整数包含的最大值和最小值、浮点小数的精确度、硬件的默认整数尺寸、可访问内存的最大尺寸，以及硬件是 big-endian 还是 little-endian。

工具 gnatpsta 输出的是 Ada 包的源代码，其中包含为定义分配的值，它们都是独立实现的。其中包括浮点数字的最大和最小值、编译程序可以识别的整个字符集，以及表示宽字符的方法。

这两个程序都没有命令行选项。运行这两个程序也很简单，可以动态地确定输出值。

# 第 10 章 GCC 混合语言

有可能出现这种情况：程序的不同部分由不同语言编写而成。一般出现这种情况是因为程序由某种语言编写，但又要扩展其兼容性，使之能够和其他软件相兼容。通常项目合并、部门合并、甚至公司合并都会产生这样的结果。组合语言最可能的原因在于希望在一种语言之中可以使用其他功能——使用高级语言时如果能够利用 C 语言的系统级功能会很方便。在同一个程序中使用两种语言的其他原因，就是因为这是一种惯用的技巧。

本章讨论了 GCC 家族中的语言混合。虽然将不同编译程序生成的不同语言的目标代码混合起来是可能的，但却非常困难，解决此问题的关键在于所涉及到的编译程序的特性。而这种混合的复杂程度会导致情况的不稳定。由于 GCC 编译所有语言目标代码的后端相同，它就有可能按照升级编译程序的方式混合语言，这种升级不会影响产生程序的正常操作。当然，不可能确保所有事情，因为编译程序是很复杂的，混合语言中一点很小的矛盾就可能产生很大的问题。

在混合语言时，产生的一些技巧不只是为了保证一种语言的基础结构能够和另一种的基础结构相匹配，还有更多。程序员必需准备好处理各种问题，包括全局命名约定、命名 mangling、参数传递、数据类型转换、出错处理，以及两种语言标准运行时库的混合。

## 10.1 C++和 C 的混合

C 语言和 C++可以自然混合，因为设计 C++就是作为 C 语言的功能扩展，因此调用规则是一样的，而且数据类型也基本相同。惟一的区别主要体现在函数名上——C 语言使用简单函数名，不考虑参数的个数和类型，而 C++函数总将它的参数类型列表当作函数名的一部分。因此，C++提供一些特殊的功能可以声明 C 函数，也就是说，C++程序可以直接声明并调用 C 函数。

### 10.1.1 在 C++中调用 C

下面是一个 C++程序，会调用 C 函数 csayhello()。它可以直接调用该函数，因为在 C++

程序中其声明为 extern "C"：

```
/* cpp2c.cpp */
#include <iostream>

extern "C" void csayhello(char *str);

int main(int argc,char *argv[])
{
    csayhello("Hello from cpp to c");
    return(0);
}
```

C 函数不需要特殊声明，如下：

```
/* csayhello.c */
#include <stdio.h>
void csayhello(char *str)
{
    printf("%s\n",str);
}
```

下面三条命令可将这两个程序编译连接成一个可执行程序。g++和 gcc 的灵活性允许使用其他方式，但以下命令可能最直接：

```
$ g++ -c cpp2c.cpp -o cpp2c.o
$ gcc -c csayhello.c -o csayhello.o
$ gcc cpp2c.o csayhello.o -lstdc++ -o cpp2c
```

注意，有必要在最后连接过程中指定标准 C++库，因为是用 gcc 命令而不是 g++命令激活连接的。如果使用 g++命令，就已经指明 C++库了。

通常在头文件中声明函数，将整个头文件都声明为 extern "C" 。这是标准 C++语法，如下：

```
extern "C" {
    int mlimitav(int lowend, int highend);
    void updatedesc(char *newdesc);
    double getpct(char *name);
};
```

### 10.1.2　在 C 中调用 C++

对于调用 C++程序函数的 C 程序，C++程序有必要提供函数可以使用 C 语言的调用顺序。下面例子的语法解释如何在 C++程序中创建 C 函数：

```
/* cppsayhello.cpp */
```

```
#include <iostream>

extern "C" void cppsayhello(char *str);

void cppsayhello(char *str)
{
    std::cout << str << "\n";
}
```

尽管函数 cppsayhello()声明为 extern "C"，指出它为 C 函数，但事实上它是 C++程序源代码的一部分，就是说函数内部的代码实际上是 C++代码。在函数中，可以随意创建和清除对象。而且，如果在函数 cppsayhello()内部调用 C 函数，还必须声明被调用的函数为 extern "C"。否则，编译程序会假设它为 C++函数而改动相应的函数名。

如下的 C 程序调用 C++的函数 cppsayhello()：

```
/* c2cpp.c */
int main(int argc,char *argv[])
{
    cppsayhello("Hello from C to C++");
    return(0);
}
```

下面命令编译并连接这个 c2cpp 程序：

```
$ g++ -c cppsayhello.cpp -o cppsayhello.o
$ gcc -c c2cpp.c -o c2cpp.o
$ gcc cppsayhello.o c2cpp.o -lstdc++ -o c2cpp
```

## 10.2 混合 Objective-C 和 C

因为 Objective-C 语言和 C 一样，只是它增加了一些声明类的语法，因此混合这两种语言非常简单。调用序列（calling sequences）也相同，因此除了调用函数之外不需其他操作。

### 10.2.1 在 Objective-C 中调用 C

下面的 Objective-C 程序将字符串的地址传递给 C 函数 csayhello()：

```
/* objc2c.m */
#import <stdio.h>
int main(int argc,char *argv[])
{
    csayhello("Hello from Objective-C to C");
    return(0);
}
```

函数 csayhello()将字符串显示到标准输出上，如下：

```
/* csayhello.c */
#include <stdio.h>
void csayhello(char *str)
{
    printf("%s\n",str);
}
```

下面三条命令可将程序编译并连接成可执行程序。连接程序时，有必要指出选项-lobjc，可为 Objective-C 包含运行时库：

```
$ gcc -Wno-import -c objc2c.m -o objc2c.o
$ gcc -c csayhello.c -o csayhello.o
$ gcc objc2c.o csayhnello.o -lobjc -o objc2c
```

### 10.2.2　在 C 中调用 Objective-C

如下是 C 程序，调用 Objective-C 函数 objcsayhello()：

```
/* c2objc.c */
int main(int argc,char *argv[])
{
    objcsayhello("Hello from C to Objective-C");
    return(0);
}
```

被调用的 Objective-C 函数源代码如下：

```
/* objcsayhello.m */
#import <objc/Object.h>
#import "SpeakLine.h"
void objcsayhello(char *str)
{
    id speak;

    speak = [SpeakLine new];
    [speak setString: str];
    [speak say];
    [speak free];
}
```

函数 objcsayhello 创建 SpeakLine 对象，将文本行保存在该对象中，然后使用方法 Say 显示该串。头文件 SpeakLine 和实现的文件如下：

```
/* SpeakLine.h */
#import <objc/Object.h>
```

```
@interface SpeakLine : Object
{
    char *string;
}
- setString: (char *) str;
- say;
- free;
@end
/* SpeakLine.m */
#import "SpeakLine.h"
@implementation SpeakLine

+ new
{
    self = [super new];
    return self;
}
- setString: (char *)str
{
    string = str;
    return self;
}
- say
{
    printf("%s\n",string);
    return self;
}
- free
{
    return [super free];
}
```

下面四条命令将源文件编译成目标文件，并将两个目标文件连接成可执行程序：

```
$ gcc -Wno-import -c objcsayhello.m -o objcsayhello.o
$ gcc -Wno-import -c SpeakLine.m -o SpeakLine.o
$ gcc -c c2objc.c -o c2objc.o
$ gcc c2objc.o objcsayhello.o SpeakLine.o -lobjc -o c2objc
```

## 10.3 混合 Java 和 C++

使用 Cygnus Native Interface（CNI）可从 C++代码中访问 Java 类。这两种语言区别很大，但有其基本上的相似之处：

- 类是按照名字声明的，可以继承其他类的属性。
- 类包含成员函数，可以通过匹配参数重载这些函数。
- 数据类型和表达式都满足 C 语言模式。

因为 GCC 编译 Java 和 C++类的方式相似，所以惟一的必要条件就是必须要避免或调整最基本的语言不兼容性，这样用 Java 编写的类才会可用。

### 10.3.1　创建 Java 字符串并调用静态方法

以下例子程序创建 Java 类 java.lang.String 的对象，并将它传递给方法 java.lang.System.out()加以显示：

```
/* cnistrout.cpp */
#include <gcj/cni.h>
#include <java/lang/System.h>
#include <java/io/PrintStream.h>

int main(int argc, char *argv)
{
   java::lang::String *str;

   JvCreateJavaVM(NULL);
   JvAttachCurrentThread(NULL,NULL);

   str = JvNewStringLatin1("Hello from C++ to Java");
   java::lang::System::out->println(str);

   JvDetachCurrentThread();
}
```

该程序按照以下命令进行编译及连接：

```
$ g++ cnistrout.cpp -lgcj -o cniexception
```

头文件 cni.h 包括所需函数调用的原型，可以激活 CNI 接口。而且 C 语言头文件中也有一些包含语句，用于 java.lang.System 和 java.io.PrintStream 类。包含 java.lang.String 头文件并没有坏处，但它还有其他一些系统级的头文件总会包含在 cni.h 内。

Java 使用指针（叫做引用）跟踪它的类，作为到对象 java.lang.String 的指针，将它声明为保存对象的地址。完整名包括 C++语法的一对冒号，完全满足 Java 类名字的要求。所有到 Java 类名的引用都需要这种命名规则，除非指定了命名空间。例如，String 和 System 类的声明及用法如下：

```
using namespace java::lang;
String *str;
```

```
. . .
System.out->println(str);
```

调用函数 JvCreateJavaVM()初始化 Java 运行时。它包括设置 Java 线程接口、垃圾收集、以及异常处理。函数必需在创建所有 Java 类或调用任意 Java 方法之前被调用一次。

调用函数 JvAttachCurrentThread()将程序线程注册到已经初始化过的 Java 运行时。函数也要在创建所有 Java 类或调用任意 Java 方法之前被调用，但它要在 JvCreateJavaVM()之后调用。

在程序结尾处，调用函数 JvDetachCurrentThread()会终止到 Java 运行时的注册，该注册是由前面对 JvCreateJavaVM()和 JvAttachCurrentThread()的调用引起的。该函数调用确保能够释放掉应用程序占用的所有资源。

在 CNI 接口中，Java 的 String 对象总是通过调用以下函数来构造：

- JvNewString(const char *chars,jsize length) 返回指定长度 length 的 String 对象，包含 chars 字符串中的字符。
- JvNewStringLatin1(const char *bytes,jsize length) 返回指定长度 length 的 String 对象，包含 bytes 数组的值。
- JvNewStringLatin1(const char *bytes) 返回 String 对象，包含的值从 bytes 数组开始，直到零值的第一个字节（但不包括该字节）。
- JvNewStringUTF(const char *bytes) 返回 String 对象，包含 UTF-编码值，从 bytes 数组开始，直到零值的第一个字节（但不包含该字节）。

### 10.3.2 加载并实例化 Java 类

使用 CNI 就可能在程序中随便混合 C++和 Java 类。下面的例子由简单的 C++ 主线程序和单独载入并实例化的 Java 类的对象组成，用来保存并显示字符串。

Java 类命名为 Speak，用来保存并显示简单字符串：

```
/* Speak.java */
public class Speak {
    String string;
    Speak() {
        string = "Uninitialized";
    }
    public void setString(String str) {
        string = str;
    }
    public void showString() {
        System.out.println(string);
    }
}
```

Speak 类的构造函数将内部字符串初始化为默认值，但此值可通过调用 setString()重写。可调用方法 showString()将当前字符串显示到标准输出上。该类被编译为 Java 的.class 文件，可使用任意标准 Java 编译程序，也可通过 GCC 编译程序的如下命令实现此步骤：

```
$ gcj -C Speak.java
```

下一步就是要使用 gcjh 命令和 Speak.class 文件生成 CNI 头文件 Speak.h，如下：

```
$ gcjh Speak
```

gcjh 命令可以生成 JNI 和 CNI 头文件，默认生成 CNI 头文件，因此不需要命令行选项。本命令生成的头文件 Speak.h 如下：

```
// DO NOT EDIT THIS FILE - it is machine generated -*- c++ -*-

#ifndef __Speak__
#define __Speak__

#pragma interface

#include <java/lang/Object.h>

extern "Java"
{
  class Speak;
};

class ::Speak : public ::java::lang::Object
{
public: // actually package-private
  Speak ();
public:
  virtual void setString (::java::lang::String *);
  virtual void showString ();
public: // actually package-private
  ::java::lang::String *string;
public:

  static ::java::lang::Class class$;
};

#endif /* __Speak__ */
```

如你所见，头文件 Speak.h 将 Speak 类定义为 C++类，因此该头文件可被直接包含到 C++ 程序当中，如下例所示：

```
/* cnispeak.cpp */
#include <gcj/cni.h>
#include "Speak.h"

int main(int argc, char *argv)
{
    java::lang::String *str;

    JvCreateJavaVM(NULL);
    JvAttachCurrentThread(NULL,NULL);

    Speak *speak = new Speak();
    speak->setString(JvNewStringLatin1("Hello from CNI to Java"));
    speak->showString();

    JvDetachCurrentThread();
}
```

程序基本上和前面的例子 cnistrout.cpp 一样。包含 CNI 头文件 gcj/cni.h，然后是用到的所有 Java 类的头文件。一旦创建了该线程附着的 Java 虚拟机，就可载入并执行 Java 类了。关键字 new 用来调用 Speak 类的构造函数，返回新生成的 Speak 对象的地址。调用方法 setString()将新生成的 String 对象保存到 Speak 中；然后调用方法 showString()显示字符串。

下面命令会编译并连接该程序：

```
$ g++ cnispeak.cpp Speak.class -lgcj -o cnispeak
```

### 10.3.3 异常

Java 类可以抛出异常，由 C++程序捕获并进行处理，如下例所示：

```
/* cniexception.cpp */
#include <gcj/cni.h>
#include <java/lang/System.h>
#include <java/io/PrintStream.h>
#include >java/lang/Exception.h>

using namespace java::lang;

int main(int argc, char *argv)
{
    JvCreateJavaVM(NULL);
    JvAttachCurrentThread(NULL, NULL);
    try {
```

```
        String *message = JvNewStringLatin1("Hello from CNI");
        System::out->println(message);
    } catch(Exception *e) {
        e->printStackTrace();
    }
    JvDetachCurrentThread();
}
```

本例和其他 CNI 例子一样，先由初始化 Java 虚拟机开始，再由卸载（detaching）当前线程结束。using 语句用来指出 java::lang 名字空间，到类名 String、System 和 Exception 的引用都可以自动解析出来，而不需要指定全名。

程序块 try 和 catch 的写法和 Java 类中 try 语句块的写法完全一样。如果 try 语句块生成 Exception 对象，它就会被 catch 语句捕获到，然后会打印跟踪栈的内容，主要描述了在哪里产生的这次异常。

### 10.3.4　CNI 的数据类型

C++的数据类型和 Java 的数据类型非常相似，但不完全一样。因为 Java 数据类型的定义非常详细，因此可以使用 C++的 typedef 命令声明类型，此类型和 Java 的类型完全一致。表 10-1 列举了定义好的类型。

表 10-1　为 C++定义的 Java 原始类型

| Java 类型 | C++类型名 | 描述 |
|---|---|---|
| char | Jchar | 16 位 Unicode 字符 |
| boolean | Jboolean | 逻辑值真或假 |
| byte | Jbyte | 8 位符号整数 |
| short | Jshort | 16 位符号整数 |
| int | Jint | 32 位符号整数 |
| long | Jlong | 64 位符号整数 |
| float | Jfloat | 32 位 IEEE 浮点数 |
| double | Jdouble | 64 位 IEEE 浮点数 |
| void | Void | 无值 |

## 10.4　混合 Java 和 C

Java 的本地接口（Java Native Interface，JNI）可用于在 Java 虚拟机上运行的 Java 类和由 C、C++或汇编语言编写的本地执行模块之间进行通信。设计该接口对于某些 Java 程序

十分有用，这种 Java 程序往往需要利用指定平台的某些工具，正是因为它需要具有可移植性，因此不能成为 Java 的一部分。但利用 JNI 接口，Java 代码就可保持它的移植性，只需编写不同平台的新的 C 函数就可以了。

### 10.4.1 具有本地方法的 Java 类

创建 Java 类是混合 Java 和 C 的常用方法，该 Java 类包括的方法可用 C 语言实现。同样，也可以用 C++或汇编语言来实现，只不过更通用的方式还是利用 C 语言。这个例子创建的简单 Java 类只包含一个方法，但该方法是用 C 语言开发的。

下面的类 HelloNative 包括方法 main()，使用了本地方法显示字符串。该本地方法被声明为类的一部分，但方法体并没有包含进来，因为方法体是由其他语言实现的。该类也包含了静态初始化程序，用到了系统方法 loadLibrary()载入共享库。就是由该库包含的本地方法体。

```
/* HelloNative.java */
public class HelloNative {
    static {
        System.loadLibrary("libspeak.so");
    }
    public static void main(String arg[]) {
        HelloNative hn = new HelloNative();
        hn.sayHello();
    }
    public native void sayHello();
}
```

下面命令将 HelloNative.java 编译成类文件 HelloNative.class：

```
$ gcj -C HelloNative.java
```

包括本地函数原型的头文件由文件 HelloNative.class 创建，由 gcjh 命令使用选项-jni 生成，如下：

```
$ gcjh -jni HelloNative
```

命令执行结果为文件 HelloNative.h，其内容如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */

#ifndef __HelloNative__
#define __HelloNative__

#include <jni.h>
```

```
#ifdef __cplusplus
extern "C"
{
#endif

extern void Java_HelloNative_sayHello (JNIEnv *env, jobject);

#ifdef __cplusplus
}
#endif

#endif /* __HelloNative__ */
```

本地函数的名字由类名和 Java 方法名构造而成。名字总是由 Java 开头，然后为下划线，跟着就是完整有效的类名，最后以方法名结束（前面还有一个下划线）。因此，C 函数的名字可写成 Java_HelloNative_sayHello()。

新函数原型有两个参数，尽管 Java 的定义中没有参数。这两个参数是所有当作本地方法调用的函数所必需的。第一个参数为指针，指向方法体使用的接口，可以访问传递到方法的所有参数，第二个参数是到调用对象的引用（对 HelloNative 对象为变量 this）。

函数是根据头文件 HelloNative.h 中的原型实现的，如下：

```
/* HelloNative.c */
#include <jni.h>
#include "HelloNative.h"

void Java_HelloNative_sayHello(JNIEnv *env,jobject this)
{
   printf("A native JNI hello\n");
}
```

JNI 的头文件 jni.h 以及头文件 HelloNative.h 一起被包含进来，后者包含函数原型。函数实现与原型定义的名字及参数完全一致。如下两条命令将函数编译成适于插入共享库的版本，然后使用该目标文件创建共享库：

```
$ gcc -fpic -c HelloNative.c -o HelloNative.o
$ gcc -shared HelloNative.c -o libspeak.so
```

最后一步将库 libspeak.so 放置在查询共享库的路径里，并用下面命令调用主线程序：

```
$ gij HelloNative
```

### 10.4.2 向本地方法传递参数

和其他 Java 方法一样，可以将参数传递给本地方法，调用方也可得到返回值。C 或 C++ 程序的数据类型和 CNI 接口的一样，本章较早的表 10-1 列举了这些类型。

如下例子中的类有一个本地方法 sum()，它接受 4 个 int 值作为参数，返回的 int 值为 4 个参数的和：

```
/* AddFour.java */
public class AddFour {
    static {
        System.loadLibrary("libaddfour.so");
    }
    public static void main(String arg[]) {
        AddFour af = new AddFour();
        int value = af.sum(1,2,3,4);
        System.out.println("The sum of four is " + value);
    }
    public native int sum(int a,int b,int c,int d);
}
```

本地方法实现如下：

```
/* AddFour.c */
#include <jni.h>
#include "AddFour.h"

jint Java_AddFour_sum(JNIEnv *env,jobject this,
        jint a,jint b,jint c,jint d)
{
    jint total = a + b + c + d;
    return(total);
}
```

4 个新参数加在一对默认参数的后面。Java 的 int 数据类型在头文件 jni.h 中定义为 jint，且用来定义所有的参数类型以及函数类型和返回值。

下面的 4 条命令将两个源文件编译并连接为可执行的形式。第一条命令创建文件 AddFour.class，这是程序的主线。第二条命令创建头文件 AddFour.h，其中包括本地方法原型。第三条命令使用选项-fpic 编译本地方法，可将此目标文件插入到共享库中。最后一条语句创建共享库 libaddfour.so。

```
$ gcj -C AddFour.java
$ gcjh -jni AddFour
$ gcc -fpic -c AddFour.c -o AddFour.o
$ gcc -shared AddFour.o -o libaddfour.so
```

最后要做的就是将库文件放置在适当的位置，载入器（loader）可在此位置找到它，使用如下命令运行该程序：

```
$ gij AddFour
```

### 10.4.3　在 C 代码中调用 Java 类方法

通过直接调用 Java 方法，本地方法可以回调 Java 对象。下面例子的 EchoKeystroke 是 Java 类，有一个本地方法以及一个回调方法。本地方法 getKeystrokes()从键盘读出字符，而回调函数 characterCallback()用字符作为输入：

```
/* EchoKeystrokes.java */
public class EchoKeystrokes {
    static {
        System.loadLibrary("libgetkeys.so");
    }
    public static void main(String arg[]) {
        EchoKeystrokes ek = new EchoKeystrokes();
        ek.getKeystrokes();
    }
    public native void getKeystrokes();
    public void characterCallback(char character) {
        System.out.println(character);
    }
}
```

本地方法使用传给每个本地方法的两个参数自动获得回调函数需要的信息。调用函数 GetObjectClass()返回 Class 对象，代表包含被调方法的对象的类。调用函数 GetMethodID()获得调用方法的惟一标识。利用函数 CallVoidMethod()可以反复调用该方法，如下：

```
/* getkeystrokes.c */
#include <jni.h>
#include <stdio.h>
#include "EchoKeystrokes.h"

void Java_EchoKeystrokes_getKeystrokes(JNIEnv *env,jobject obj)
{
    jchar character = ' ';
    jclass class = (*env)->GetObjectClass(env,obj);
    jmethodID id = (*env)->GetMethodID(env,class,
          "characterCallback","(C)V");

    if(id != 0) {
        while(character != '.') {
            character = getchar();
            (*env)->CallVoidMethod(env,obj,id,character);
        }
    }
}
```

调用 getMethodID()需要方法的名字、返回类型以及参数类型列表，这样才能惟一标识该方法。通过如下格式的字符串来惟一标识返回值和参数：

```
"(argument type list)return type"
```

表 10-2 列举了字符串中包含的类型指示字。

**表 10-2 回调方法的返回类型和参数类型**

| 指示字 | Java 数据类型 |
|---|---|
| Z | 布尔类型 |
| B | 字节类型 |
| C | 字符 |
| S | 短整型 |
| I | 整型 |
| J | 长整型 |
| F | 浮点数 |
| D | 双浮点小数 |
| V | 空类型 |
| Lclassname; | 指定类的对象 |
| [type | 指定类型的数组 |
| (arg type list) return type | 指定参数和返回类型的方法 |

例如，如果向方法传递的是一个整数 int 和一个 double 值，返回的是 double，那么指定的串看起来如下：

```
"(ID)D"
```

如果第一个参数为字节数组，第二个为字符串，而返回值为空，那么指定的串看起来应该如下：

```
"([BLjava/lang/String;)V"
```

如下命令序列可将 EchoKeystrokes.java 编译成类文件 EchoKeystrokes.class，使用工具 gcjh 读出文件 EchoKeystrokes.class，再生成头文件 EchoKeystrokes.h 包括本地方法的原型，将文件 getkeystrokes.c 编译成位置独立的目标文件 getkeystrokes.o，最后用 getkeystrokes.o 构造共享库 libgetkeys.so：

```
$ gcj -C EchoKeystrokes.java
$ gcjh -jni EchoKeystrokes
$ gcc -fpic -c getkeystrokes.c -o getkeystrokes.o
$ gcc -shared getkeystrokes.o -o libgetkeys.so
```

## 10.5　混合 Fortran 和 C

GNU Fortran 和 C 语言可以一起使用，非常简单，因为它们都可以直接调用彼此的函数。只要仔细确保调用传递的参数类型正确，两种语言的函数就可以彼此互相调用，就好像是同一种语言一样。

表 10-3 列出了 Fortran 的数据类型和对应的 C 语言类型。该表适用于大多数平台，但也可能有个别例外。可以谨慎地创建一个小的测试程序（利用本节的例子程序），测试需要传递的所有数据类型，保证它们是相互兼容的。

表 10-3　C 与 Fortran 之间的兼容数据类型

| C 类型 | Fortran 类型 | 描述 |
|---|---|---|
| signed char | INTEGER*1 | 8 位符号整数 |
| short | INTEGER*2 | 16 位符号整数 |
| int | INTEGER | 32 位符号整数 |
| float | REAL | 32 位浮点数 |
| double | DOUBLE PRECISION | 64 位浮点数 |
| SUBROUTINE SUB() | void sub_() | 返回空的 C 函数，等同于 Fortran 的子例程 (subroutine) |
| REAL FUNCTION FUN() | float fun_() | 返回非空的 C 函数，等同于 Fortran 函数 |

因为 Fortran 总是通过引用传递参数，而 C 总是通过地址传递数组，数组的传递更直接，也不需要改动。但对于多维数组，不同语言使用的脚本需要逆转，因为 Fortran 的数组按照列主顺序（column- major order）进行组织，而 C 的数组则按照行主顺序（row-major order）进行组织。

### 10.5.1　从 Fortran 中调用 C

下面的 Fortran 程序调用了 C 函数，传递的参数一个是字符串，另一个是浮点数：

```
C  f772c.f
C
      PROGRAM F772C
C
      CHARACTER*32 HELLO
      REAL PI
C
      HELLO = "Hello C from Fortran"
      HELLO(21:21) = CHAR(0)
```

```
      PI = 3.14159
      CALL SHOWHIPI(HELLO,PI)
      END PROGRAM F772C
```

名为 HELLO 的 CHARACTER 数据类型可保存 32 个字符，当只保存 21 个字符时，剩下的位置用空格填充。使用 C 语言格式化该串，有必要在字符串结束的位置插入零字节作为字符串终止符。PI 数据类型为 REAL，和 C 语言中的 float 数据类型一样，因此可以直接传递给函数。

注意 Fortran 通过引用传递参数，这一点很重要，因为 C 函数总是收到传递值的地址，而不是值本身。下面的 C 函数显示了由 Fortran 程序传递给它的字符串和实数：

```
/* showhipi.c */
#include <stdio.h>
void showhipi_(char *string,float *pi)
{
   printf("%s\nPI=%f\n",string,*pi);
}
```

两种平台的命名规则有些不同，而且两种语言的数据类型兼容性也有些区别。如你所见，本例有必要在函数名后添加一条下划线，但传给函数的数据格式正确。

下面的命令将编译这两个文件，并将它们连接成一个可执行程序：

```
$ g77 -c f772c.f -o f772c.o
$ gcc -c showhipi.c -o showhipi.o
$ g77 c2f77.o showhipi.o -o f772c
```

### 10.5.2 从 C 中调用 Fortran

当从 C 程序中调用 Fortran 子例程（subroutine）时，有必要将参数的地址传递给 Fortran 程序，同时要保持串格式的正确。下面的例子将字符串和浮点值传递给 Fortran 程序：

```
/* c2f77.c */
int main(int argc,char *argv[])
{
   int i;
   float e = 2.71828;
   char hello[32];
   int length = sizeof(hello);

   strcpy(hello,"Hello Fortran from C");
   for(i=strlen(hello); i<length; i++)
      hello[i] = ' ';
   showhie_(hello,&length,&e);
   return(0);
}
```

C 语言中，字符串的长度由 null 字符的位置决定，但在 Fortran 中所有字符串的长度都是固定的。因为 Fortran 没法确定传进来的参数的长度，有必要将串的长度也作为参数传给函数。在本例中，整个数组为空，第二个参数是数组的大小。注意，所有三个参数传递的都是指针，指向实际的数据——因为 Fortran 期待的总是地址而不是实际数据。调用函数时，通常有必要为函数名加一个下划线。

下面是被调用的 Fortran 子例程的源代码：

```
C  showhie.f
C
      SUBROUTINE SHOWHIE(HELLO,LENGTH,E)
      CHARACTER*(*) HELLO
      INTEGER LENGTH
      REAL E
C
      WRITE(*,100) HELLO(1:LENGTH),LENGTH,E
  100 FORMAT(3X,A,2X,I3,4X,F6.4)
      RETURN
      END SUBROUTINE SHOWHIE
```

下面的三条命令将两个源文件编译成目标文件，再将它们连接成可执行程序：

```
$ g77 -c showhie.f -o showhie.o
$ gcc -c c2f77.c -o c2f77.o
$ gcc c2f77.o showhie.o -lfrtbegin -lg2c -lm -o c2f77
```

第三条命令需要 Fortran 库，因为它指定的是 gcc 命令。而使用 g77 命令则可将库自动包含进来，因此最后一条命令可简写如下：

```
$ g77 c2f77.o showhie.o -o c2f77
```

## 10.6　混合 Ada 和 C

Ada 语言包括调用 C 和 Fortran 函数必需的工具。通过使用 pragma import 可以声明 Ada 的过程体，以便指出外部语言以及函数体的代码名。

Ada 和 C 使用的数据类型彼此的兼容性很好，尤其在使用 GCC 为二者生成目标代码的时候。表 10-4 列举了两种语言相同的数据类型。

表 10-4　Ada 数据类型以及相应的 C 数据类型

| Ada 类型 | C 类型 |
|---|---|
| Float | float |

（续表）

| Ada 类型 | C 类型 |
|---|---|
| Integer | int |
| Long_Float | double |
| Long_Integer | long |
| Long_Long_Integer | long long |
| Short_Float | float |
| Short_Integer | short |
| Short_Short_Integer | signed char |

### 10.6.1 从 Ada 中调用 C

这个简单的例子演示如何用 C 来实现 Ada 包中的过程体。如下是 Ada 程序的主线，它调用了 Howdy 包中的过程 hello 和 goodbye：

```
-- ada2c.adb
with Howdy;
procedure Ada2C is
begin
   Howdy.hello;
   Howdy.goodbye;
end Ada2C;
```

hello 和 goodbye 过程都会显示一行文本，但是 goodbye 是由 Ada 编写成的，而 hello 过程是 C 代码。Howdy 包的成员在文件 howdy.ads 中指明，如下：

```
-- howdy.ads
package Howdy is
   procedure Hello;
   procedure Goodbye;
end Howdy;
```

在文件 howdy.adb 中实现过程体，该文件包含 goodbye 的实际代码，并声明了到 hello 的外部引用：

```
-- howdy.adb
with Text_IO; use Text_IO;
with Interfaces.C;
package body Howdy is
   procedure Hello is
      procedure sayhello;
      pragma Import(C,sayhello);
   begin
```

```
        sayhello;
    end Hello;
    procedure Goodbye is
    begin
        Put_Line("Goodbye");
    end Goodbye;
end Howdy;
```

with Interfaces.C 语句用于将数据类型定义设置为在 C 和 Ada 之间兼容，但严格上讲它不是必须的，因为被调用的 C 函数没有参数和返回值。过程 hello 调用 C 函数 sayhello，因此 procedure 和 pragma Import 语句对于指定 sayhello 为外部 C 函数是很重要的。

Import 程序的第一个参数是编写外部过程的语言名字。Ada 标准声明已知的语言为 C、C++、Fortran 和 COBOL。第二个参数为函数的名字，程序会在内部使用该函数。如果函数的真实名字对 Ada 无效，就需要用第三个参数指明实际的外部名字。例如，如果希望调用远程函数_stprob()，开始的下划线对 Ada 而言是非法的，因此需要指定程序如下：

```
pragma Import(C,stprob,"_stprob")
```

这样便可以使用内部名 stprob 指向外部名_stprob。

本例调用的 C 函数非常简单，看起来如下：

```
/* sayhello.c */
#include <stdio.h>
void sayhello()
{
   printf("Hello C from Ada\n");
}
```

下面的命令序列会编译 Ada 源文件和 C 源文件并将目标文件连接成一个可执行程序：

```
$ gcc -c sayhello.c -o sayhello.o
$ gcc -c howdy.adb
$ gcc -c ada2c.adb
$ gnatbind ada2c.ali
$ gnatlink ada2c.ali sayhello.o
```

### 10.6.2　从 Ada 中调用 C（带参数）

本例和前面的例子很像，只不过它既有参数又有返回值。本例使用了 UNIX 系统调用在后台启动一个进程然后终止该进程。文件 adaspawn.adb 包括程序的主线：

```
-- adaspawn.adb
with Spawn;
procedure AdaSpawn is
pid : Integer;
```

```
status : Integer;
begin
   pid := Spawn.startProcess("flex");
   status := Spawn.stopProcess(pid);
end AdaSpawn;
```

在主线程序中，调用 startProcess()的参数为运行的程序的名字。返回值是进程 ID 号，stopProcess()使用该进程 ID 号终止运行的程序。在文件 spawn.ads 中，这两个函数都定义为 Spawn 包的成员：

```
-- spawn.ads
package Spawn is
   function startProcess(name : String) return Integer;
   function stopProcess(pid : Integer) return Integer;
end Spawn;
```

传入和传出 Spawn 函数的数据都是 Ada 类型。在这些函数中，有对 C 函数的调用，因此需要一些数据转换以保证兼容性。函数体定义在文件 spawn.adb 中：

```
-- spawn.adb
with Interfaces.C;
package body Spawn is
   function startProcess(name : String) return Integer is
      function start(name : String) return Interfaces.C.int;
      pragma Import(C,start);
   begin
      return Integer(start(name));
   end startProcess;
   function stopProcess(pid : Integer) return Integer is
      function stop(pid : Integer) return Interfaces.C.int;
      pragma Import(C,stop);
   begin
      return Integer(stop(pid));
   end stopProcess;
end Spawn;
```

Ada 函数 startProcess()和 stopProcess()作为封装器（wrappers）包围在 C 函数 start()和 stop()外。此间发生了一些较小的数据转换。start()和 stop()都会返回 C 数据类型 Interface.C.int，会将它转换为 Ada 类型 Integer，这样才可能从 startProcess()和 stopProcess()中返回值。

剩下的就是 C 函数本身，保存在文件 startstop.c 中，如下：

```
#include <unistd.h>
#include <signal.h>
```

```
#include <errno.h>

int start(char *name)
{
    int pid;
    char *argv[4];

    pid = fork();
    if(pid == -1)
        return(-1);
    if(pid == 0) {
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = name;
        argv[3] = 0;
        execve("/bin/sh",argv,0);
        exit(-1);
    } else {
        return(pid);
    }
}
int stop(int pid)
{
    if(kill(pid,SIGTERM) < 0)
        return(errno);
    return(0);
}
```

函数 start()调用系统调用 fork()，它会克隆当前进程。fork()返回的值会说明该进程为原始进程还是克隆进程，而克隆进程通过调用 execve()将自己转换到不同的进程。系统调用 execve()不会返回值，因为它通过内核从开始启动一个程序，可将自己立即转换到新的进程。从 start()只返回原始的程序，而返回新启动进程的 PID。

这种完全的组织方式能在 C 函数外包上 Ada 函数，而这里的 C 函数实现实际的系统调用。这些例子使用的组织方式使每步都变得非常清楚，但并不是惟一的途径。完全可以在 Ada 代码中直接调用 execve()、kill()或其他系统调用，方法与调用 startProcess()和 stopProcess()一样。

# 第 11 章 GCC 国际化

所有程序，包括 GCC 编译程序本身，如果编写正确，就可令其接口适应于本地语言和环境，并以这种方式运行。

国际化（Internationalization）是指程序的多语言支持能力，或是一些程序的集合可打包进行工作。这些程序只用一种语言编写，但程序内部代码的组织方式支持动态替换程序中的字符串，可将其替换为其他语言。

本地化（Localization）是一种操作，它使用了内置在程序中或程序集中的工具，将所有用户可读的字符串转换为不同的语言。这就叫做本地化设置（setting the locale），这是通过系统设置完成的，是在载入程序的时候读出并运行的。

NLS（Native language support，本地语言支持）是一个词组，用来指定国际化和本地化的所有操作。

通常可以看到术语 internationalization（国际化）被简写为 i18n。源自于 internationalization 由字母 *i* 打头，后面跟着 18 个字母，而最后以字母 *n* 结束。使用同样的模式，术语 localization 有时可写作 l10n。

总地来说，i18n 由程序员管理，而 l10n 由翻译者和用户管理。

本章的例子和解释都是 C 语言模式，但同样的处理也可用于 C++、Objective-C、Python、Lisp、EmacsLisp、Java 和 awk。

## 11.1 翻译的例子

下面的程序包含必要的代码，其中的字符串可被翻译：

```
/* starter.c */
#include <locale.h>
#include <libintl.h>

#define PACKAGE "starter"
```

```
#define LOCALEDIR "/usr/share/locale"

int main(int argc,char *argv[])
{
    setlocale(LC_ALL,"");
    bindtextdomain(PACKAGE,LOCALEDIR);
    textdomain(PACKAGE);

    printf("%s\n",gettext("This string will translate."));
}
```

头文件 locale.h 包含一些基本的宏定义，可用于指定进行本地化处理的数据类型，以及涉及到的货币符号的数据结构。头文件 libintl.h 包括需要配置和激活国际化处理的函数原型。

函数 main()调用 setlocale()指出哪个项目要进行国际化处理。用 LC_ALL 指出所有事情都要国际化处理，但你也许只希望某个项目国际化。代替使用 LC_ALL 调用 setlocale()，程序可以多次调用 setlocale()指出表 11-1 列出的每个项目。由 setlocale()返回的字符串是本地设置的当前标识。

**表 11-1　setlocale()可识别的本地化类别**

| 本地分类 | 描述 |
|---|---|
| LC_ADDRESS | 标准地址格式包括公司名、建筑名、部门名、C/O 地址、房间名、邮政编码、指定国家名等 |
| LC_ALL | 与指定本列表中所有成员一样 |
| LC_COLLATE | 正则表达式匹配。确定表达式字符的意义和范围 |
| LC_CTYPE | 正则表达式匹配。确定字符分类、规则、大小写敏感比较、以及宽字符函数 |
| LC_IDENTIFICATION | 信息格式，例如名字、地址、电话号码、e-mail 地址、传真号码，等等 |
| LC_MEASUREMENT | 将测量单位本地化为公制或英语系统 |
| LC_MESSAGES | 本地化文本自然语言消息 |
| LC_MONETARY | 格式化货币显示串 |
| LC_NAME | 格式化人名表示规范，包括大写字母、称谓、称谓简写、以及姓和名的位置 |
| LC_NUMERIC | 格式化数字值，包括十进制的小数点和千位分隔符 |
| LC_PAPER | 用于打印的标准纸张大小 |
| LC_TELEPHONE | 格式化电话号码，包括前缀及国家号 |
| LC_TIME | 格式化日期及时间字符串 |

本例中，包含程序的包名和包含本地路径的目录名由程序中的一对指示字#define 加以说明，但使用 config.h 文件命名更为普遍，或由 makefile 生成的命令行选项-D 加以说明。

要将字符串从一种语言翻译成另一种，需要调用函数 gettext()。实际上是一族 gettext()函数，下一小节有介绍，其中任何一个都可激活工具 xgettext 展开字符串，稍后有所描述。原始串（程序中列出的串）用作主键为当前本地定位翻译。如果没有找到匹配，使用原始串。函数 gettext()的返回值是字符串。因此，例子的 printf()语句只简单地将 gettext()函数的返回值显示出来，不论返回的内容是什么，也不用了解是否经过了翻译。

为了编程方便以及当前程序的转换，使用一个短小的宏来取代函数 gettext()也是常用的方法。例如，利用以下宏定义，可将函数调用简化成一个单独下划线字符：

```
#define _(a) gettext(a)
```

使用宏，例子中的语句 printf()可变成如下形式：

```
printf("%s\n",_("This string will translate"));
```

用这种技术，翻译函数的调用可减少到总共只使用 3 个字符（下划线和两个括号）。

## 11.2 创建新的.po 文件

一旦文本中需要的所有字符都被翻译了，且通过调用函数 gettext()也被正确标识了，这就有必要使用字符串和关键字开始构造文件了，将翻译应用于每个本地目标。项目通过使用工具 xgettext 开始，将文本中的行解析出来，将它们组织成新的.po 文件。如下命令从 starter.c 文件中解析出合适的字符串，并创建文件 messages.po:

```
$ xgettext starter.c
```

文件 messages.po 包括一些标准的头文件信息，如下所示：

```
msgid "This string will translate."
msgstr ""
```

为完成文件，翻译器只需要编辑.po 文件，在适当的位置键入翻译值取代 msgstr 标记右边的空串。如果程序要翻译大量的字符串，它们都会出现在同一个文件中。

工具 xgettext 可被很多编程语言使用，可将所有输入源文件的字符串组合成一个.po 文件用于整个包。工具 xgettext 的命令行选项列在表 11-2 中。

**表 11-2 xgettext 的命令行选项**

| 选项 | 描述 |
|---|---|
| - | 不是从文件中读出，而是从标准输入中读出源代码 |
| -a | 如果语言是 C 或 C++，该选项展开所有字符串 |
| --add-comments=tag | 同选项-c |

（续表）

| 选项 | 描述 |
| --- | --- |
| --add-location | 同选项-n |
| -C | --language=C 的简写 |
| -c tag | 用于在输出文件指定标记位置添加注释块 |
| --c++ | --language=C++的简写 |
| --copyright-holder=str | str 是包的商标持有者的名字，因此也是展开字符串的名字。如果没有指定，默认为自由软件协会（Free Software Foundation） |
| -d name | 输出文件为 name.po（而不是默认的 messages.po）。参见-o |
| -Ddirectory | 将指定目录添加到列表中，该列表是查找指定源文件的目录 |
| --default-domain=name | 同选项-d |
| --directory=directory | 同选项-D |
| --exclude-file=file | 同选项-x |
| --extract-all | 同选项-a |
| -F | 按照文件位置排序输出 |
| -f file | 从文件中读输入文件名，而不是从命令行中读出输入文件名 |
| --files-from=file | 同选项-f |
| --force-po | 产生输出文件，即使没有找到可翻译的字符串 |
| --foreign-user | 忽略--copyright-holder 的默认输出 |
| -h | 显示选项列表并退出 |
| --help | 显示选项列表并退出 |
| -i | 在写.po 文件时，采用缩进格式 |
| --indent | 同选项-i |
| -j | 将消息添加到现有输出文件中 |
| --join-existing | 同选项-j |
| -k keywordspec | 如果语言是 C 或 C++，额外关键字 keywordspec 会激活字符串摘要。keywordspec 的格式为 named:num，这里 num 是字符串的参数个数。默认关键字为 gettext、dgettext:2、dcgettext:2、ngettext:1、dngettext:2、3、dcngettext 和 gettext_noop。如果没有指定 keywordspec，不会使用默认关键字 |
| -keyword=keywordspec | 同选项-k |
| -L name | 输入文件的语言种类。可为 C、C++、ObjectiveC、PO、Python、Lisp、EmacsLisp、librep、Java、awk、YCP、Tcl、RST 或 Glade |
| --language=name | 同选项-L |

（续表）

| 选项 | 描述 |
|---|---|
| -m [string] | 使用指定字符串（或在没有指定字符串时，使用""）作为输出文件中所有 msgstr 项目的前缀。参见-M |
| -M [string] | 使用指定字符串（或在没有指定字符串时，使用""）作为输出文件中所有 msgstr 项目的前缀。参见-m |
| --msgstr-prefix[=string] | 同选项-m |
| -msgstr-suffix[=string] | 同选项-M |
| -n | 包含注释行用来指出字符串的来源。这是默认设置 |
| --no-location | 不包含指出字符串来源的注释行 |
| --no-wrap | 长消息行在输出文件中不被分开 |
| -o file | 输出文件为指定文件（不是默认的 messages.po）。也见选项-d |
| --omit-header | 忽略头，通常由 msgid""项目进行标识 |
| --output-dir=directory | 同选项-p |
| --output-file=file | 同选项-o |
| -p directory | 输出文件位于指定目录中 |
| -s | 生成排序的输出，而不是按照源文件出现字符串的顺序 |
| --sort-by-file | 同选项-F |
| --sort-output | 同选项-s |
| --strict | 严格按照 Uniforum 格式写.po 文件。格式不支持 GNU 功能扩展 |
| -T | 如果语言为 C，就可以识别三字母词 |
| --trigraphs | 同选项-T |
| -v | 显示版本信息，然后退出 |
| --version | 显示版本信息，然后退出 |
| -w number | 指定输出页面宽度。比宽度还长的行会被截断 |
| --width=number | 同选项-w |
| -x file | .po 或.pot 文件中的选项不会被析取出来 |

下面命令生成文件 messages.po，该文件包含的一个或多个字符串，通过调用源文件 starter.c 中的 gettext()来实现：

```
$ xgettext starter.c
```

xgettext 的一个最重要的选项是-j，它可以从源文件生成新的 messages.po 文件，但会读出较旧版本的 messages.po 文件，保留翻译器插入文件中的所有已经翻译好的内容。这非常重要，因为它自动更新了消息文件，而不用抛弃所有已经完成的工作。例如，下面的命令

会读出文件 starter.po，将最新的字符串翻译和目前仍然有效的翻译混合在一起，形成新版本的 starter.po：

```
$ xgettext -j -d starter starter.c
```

## 11.3　函数 gettext()的用法

翻译字符串最简单的方法就是将该字符串作为参数调用函数 gettext()。有必要使用不同方法的情形也有，那样需要使用其他函数解决具体问题。

### 11.3.1　静态字符串

以下例子解释如何声明字符串为全局变量的初始值，可在程序运行的时候进行动态翻译：

```
/* statictrans.c */
#include <locale.h>
#include <libintl.h>

#define PACKAGE "starter"
#define LOCALEDIR "/usr/share/locale"

#define gettext_noop(a) (a)

char *glbl = gettext_noop("This is a global static string.");

int main(int argc,char *argv[])
{
   setlocale(LC_ALL,"");
   bindtextdomain(PACKAGE,LOCALEDIR);
   textdomain(PACKAGE);

   printf("%s\n",gettext(glbl));
}
```

声明函数 gettext_noop()是什么都不做的宏，结果只是字符串本身，这会引起 xgettext 去察看哑函数的名字而跳过字符串。稍后调用 gettext()传递实际字符串的地址作为参数，这样就会在使用字符串的地方替换成翻译串。结果就好像已经将字符串声明为传递给 gettext()的常数参数。如果在程序中的多处用到了全局字符串，每次引用的位置都要对它进行翻译。

### 11.3.2　从其他域中进行翻译

如果需要从其他包中得到字符串的翻译，可以调用函数 dgettext()指定其他包的名字。例如，如果包为 hrdomain，而关键字符串"Daily average catch"在该域中已经被翻译过了，

通过使用 dgettext()可实时获得其他域的翻译：

```
dgettext("hrdomain","Daily average catch");
```

执行 xgettext 不会解析该特殊的字符串，因为已经指出在其他位置有它的翻译。

### 11.3.3 从其他域到指定类别的翻译

与 dgettext()相同，函数 dcgettext()可从其他域得到翻译的字符串。它还允许为翻译选择分类。类别为表 11-1 指定的常数。例如，如下命令可根据指定域 hrdomain 的规则翻译日期值：

```
dcgettext("hrdomain","12/04/03",LC_TIME);
```

### 11.3.4 多重模式

方法 ngettext()用于翻译字符串时会考虑复数形式。原始字符串的单数形式和复数形式都被传给函数，以及复数的程度也会传给函数。某些语言中一个是单数，两个为双数，而三个或三个以上为复数。例如，如下调用当它是两个 images 的引用时，可以翻译词“image”：

```
ngettext("picture","pictures",2L);
```

本例中，自动翻译过程需要选择目标语言的正确的复数形式来说明这是两张图片。

### 11.3.5 来自其他域的多重模式

函数 dngettext()的原理和 ngettext()相同，只不过它会在其他域里查找翻译。下面的例子查找域 hrdomain，寻找正确的复数形式以指定一对图像：

```
dngettext("hrdomain","picture","pictures",2L);
```

### 11.3.6 某个类别其他域的多重模式

函数 dcngettext()的工作原理和 dngettext()相似，不过它是根据指定类别定义查找翻译。类别为表 11-1 列举类别之一。下面的例子查找域 hrdomain，以指出一对图像正确的多重模式：

```
dcngettext("hrdomain","Mr. Garcia","Messrs. Garcia",2L,LC_NAME);
```

本例根据格式化名字使用的原则，为两位男士的名字选择正确的翻译，这两位先生的姓都为 Garcia。

## 11.4 合并两个.po 文件

尽管有可能使用 xgettext 同时生成两个翻译表，将它们自动混合成一个现有的.po 文件，

但还是会发现自己处于（或说操作）两个独立的.po 文件之间——一个较旧，现有的.po 文件包括程序以前版本的翻译，而另一个新文件包括软件新版本生成的项目。如果是这种情况，可使用如下工具 msgmerge 将两者合并起来：

```
$ msgmerge oldfile.po newfile.po
```

本例中的 oldfile.po 包含现有的所有翻译，它们都会被传递到新创建的文件当中，这些串也存在于 newfile.po 中。另外文件 newfile.po 中存在的、oldfile.po 文件中没有的字符串应该被加入到输出中。新数据要被写到标准输出中，除非在命令行参数中指定了输出文件。msgmerge 的选项列在表 11-3 中。

**表 11-3　msgmerge 的命令行选项**

| 选项 | 描述 |
|---|---|
| --add-location | 包含注释行，指出原始源中每个字符串的位置。这是默认设置。见--no-location |
| -D directory | 将指定目录加入列表，就是在此列表里查找指定输入文件 |
| --directory=directory | 同选项-D |
| -e | 指出输出文本中不使用 C 语言的转义序列。这是默认设置 |
| -E | 在输出文本中使用 C 语言的转义序列 |
| --escape | 同选项-E |
| --force-po | 写输出文件，即使输出文件为空 |
| -h | 显示选项列表，然后退出 |
| --help | 同选项-h |
| -i | 按照缩进格式生成输出 |
| --indent | 同选项-i |
| --no-location | 禁止用注释行指定原始源中每个字符串的位置。见--add-location |
| -o file | 将输出写入到指定文件。默认将输出写到标准输出设备上 |
| --output-file=file | 同选项-o |
| --strict | 严格按照 Uniforum 格式产生输出，忽略 GNU 的扩展功能 |
| -v | 生成更详细的输出描述过程 |
| -V | 显示版本号，然后退出 |
| --verbose | 同选项-v |
| --version | 同选项-V |
| -w number | 数字为最大的宽度。比指定数字还长的行会被截断 |
| -width=number | 同选项-w |

## 11.5 由.po 文件生成二进制的.mo 文件

一旦翻译文本被加入到.po 文件，下一步就是要创建.mo 文件。程序使用这个二进制文件进行翻译。使用.po 文件创建的二进制文件作为工具 msgfmt 的输入，如下：

```
$ msgfmt starter.po
```

该命令产生的二进制文件为 starter。回忆本章开始介绍的程序 starter.c，该程序由下面三个函数调用开始：

```
setlocale(LC_ALL,"");
bindtextdomain(PACKAGE,LOCALEDIR);
textdomain(PACKAGE);
```

宏 PACKAGE 定义为"starter"，宏 LOCALEDIR 定义为"/usr/share/locale"。对于查找翻译表的程序，即加拿大英语，可能将二进制文件复制到目录/usr/share/locale/en_CA/starter 中。无论何时当本地设置为 en_CA，程序就会查找并找到合适的翻译表。要为每种语言创建翻译，只需要编译 starter.po 的复本，插入合适的翻译字符串，创建其他的二进制文件，把它复制到合适的子目录中。

工具 msgfmt 的命令行选项很少，列在表 11-4 中。

**表 11-4 msgfmt 的命令行选项**

| 选项 | 描述 |
|---|---|
| -a number | 按照指定字节数排列字符串。默认为 1 |
| --alignment=number | 同选项-a |
| -c | 对字符串进行语言无关检查。包括检查 C 字符串中%格式序列的合法性，以及插入头中信息的正确性。也会检查域名和--output-file 选项是否冲突 |
| --check | 同选项-c |
| -D directory | 将指定目录加入列表，就是在该列表中查找输入文件 |
| --directory=directory | 同选项-D |
| -f | 在输入文件中使用模糊（fuzzy）项目 |
| -h | 显示选项列表，然后退出 |
| --help | 同选项-h |
| --no-hash | 二进制输出文件不包括哈希表 |
| -o file | 指出输出文件名为 file。默认使用输入文件的基础名，没有扩展名 |
| --output-file file | 同选项-o |

（续表）

| 选项 | 描述 |
| --- | --- |
| --statistics | 显示翻译表的统计信息 |
| --strict | 激活严格 Uniforum 模式 |
| --use-fuzzy | 同选项-f |
| -v | 列出输入中的所有不规则 |
| -V | 显示版本信息，然后退出 |
| --verbose | 同选项-v |
| --version | 同选项-V |

The Complete Reference

# 第 3 部分

# 外设和内设

# 第 12 章 GCC 连接和库

编译程序产生的目标文件已经包括可执行代码，但事实上编译程序产生的目标文件都不是完整的，只有和其他目标模块组合在一起才能生成可执行程序。即便简单的“hello world”程序也要使用其他目标文件中的函数，才能真正将字符串显示出来。

本章讨论连接以及可用来检查和操作目标文件的工具。目标文件就是编译程序生成的.o 文件。本章描述的很多工具都可以操作多个目标文件，作为离散文件，它们可以保存在目录里、静态库中（也被看作文档 archive），也可以保存在共享库里（也被看作动态库 dynamic library）。此外我们还会介绍一些工具，可以操作完全连接好的可执行文件。

## 12.1 目标文件和库

将目标模块组合在一起创建独立的可执行程序时，连接程序可在目录中找到作为独立文件的目标模块，以及保存在静态库或共享库中的目标模块。一个连接操作通常会考虑这三个位置中的目标文件。

### 12.1.1 目录中的目标文件

最简单的连接形式就是将目标文件编译到目录或目录集合中，然后用连接程序命令行为其命名。对于只需将目标模块连接成一两个程序的情况，这种形式工作非常正常。例如，C 程序包括源代码 main.c、inlet.c、outlet.c 和 genspru.c。以下命令序列会将它们编译成目标文件，然后将这些目标文件编译成可执行程序 spinout：

```
$ gcc -c main.c -o main.o
$ gcc -c inlet.c -o inlet.o
$ gcc -c outlet.c -o outlet.o
$ gcc -c genspru.c -o genspru.o
$ gcc main.o inlet.o outlet.o genspru.o -o spinout
```

成功执行这一系列命令之后，磁盘中包含四个目标文件和一个可执行文件。达到此目

的更简单的方法就是让编译程序用下面命令去管理整个处理过程：

```
$ gcc main.c inlet.c outlet.c genspru.c -o spinout
```

在这两种情形中，最后的可执行文件都包含四个目标文件的所有代码，以及连接程序认为必要的来自系统的其他代码。

### 12.1.2 静态库中的目标文件

目标文件可保存在静态库中，从静态库中连接这些目标文件和从各自的文件中连接的方式一样，只是连接程序需要自动查找库的内容，并且只包含必要的目标文件。如果程序中没有引用目标文件的任何内容，目标文件就不会被包含进来作为可执行程序的一部分。

静态库包含目标文件，叫做文档 archive 文件，由工具 ar 构造并维护。文档文件的名字前缀为 lib 后缀为.a。下面的命令序列编译三个目标文件，并将它们复制到库 libspin.a 中。然后连接程序会使用目标文件 main.o 和库中的内容构造可执行程序 spinner：

```
$ gcc -c inlet.c outlet.c genspru.c
$ ar -r libspin.a  inlet.o outlet.o genspru.o
$ gcc main.c libspin.a -o spinner
```

第一条 gcc 命令产生三个目标文件，接着使用 ar 命令将它们插入静态库。最后一条命令将 main.c 编译成 main.o 然后激活连接程序，它会读出 libspin.a 的内容，解析出 main.o 的外部函数和数据引用。如果保存在 libspin.a 中的模块包含的函数或者数据项被其他模块所引用，而该模块已是可执行程序的一部分了，那么必需将此模块包含进来作为最后可执行文件的一部分。因为没必要的目标模块不会包含进来，所以连接到库的可执行代码会比连接到目录中目标文件集合的代码要小（后者总会包含所有目标文件）。

静态库中和目标模块在一起的是索引，列出库中定义的所有全局数据和函数名。连接程序使用该索引确定应该包含哪个模块和忽略哪个模块。通常在创建或更新库的时候，使用工具 ar 可以创建该索引，但工具 ar 的有效选项可以限制创建索引。这在维护大型库的时候非常有用——可以有多处改动而不需要更新索引，直到所有改动都结束才会更新索引。可以利用工具 ranlib 创建索引或改动现存索引。例如，下面一对命令使用 ar 工具的-q 选项可将文件快速附加在现存文档后面，而不需要更新索引，然后使用 ranlib 更新索引反映文档当前的状态：

```
$ ar -q libspin.a mongul.o strop.o klbrgr.o
$ ranlib libspin.a
```

库中模块出现的顺序会有一些区别。如果多个模块都定义了同样的符号，那么如果连接程序查找该符号的话，就会找到并包含第一个模块。进一步，如果相同模块的不同版本保存在同一个文档中，连接程序还是只能找到第一个满足条件的模块。ar 工具的选项可用来将新的模块添加到指定位置，以此改变文档现有模块的顺序。

ar 命令的语法如下：

```
ar [options] [positionname] [count] archive objectfile [objectfile ...]
```

ar 命令是很老的 UNIX 工具之一，它的语法和其他一些老工具很类似，例如 tar，这些工具首先出现的是选项标签，而选项的字母表示之间没有空格，而且选项可通过（也可不通过）连字符来表达。可选的命令行项目 positionname 和 count 只有当用到了某些需要它们的选项时才会出现。ar 命令的选项分为两类：命令行选项告诉 ar 应该采取什么样的操作（有效命令行选项中的一个），以及修改子（modifier） 选项指出如何执行该命令操作。表 12-1 列举了可找到的所有 ar 命令选项，而修改子选项见表 12-2。

**表 12-1　指明操作的 ar 选项**

| 选项 | 描述 |
|---|---|
| d | 从文档中删除模块 objectfiles。利用 v 修改子，列出的所有模块都要被删除 |
| m | 将模块移入文档内部。默认情况下，列举的所有 objectfiles 成员都会移到文档尾部。利用修改子 a、b 和 i 可将指定模块移到其他位置 |
| p | 将指定的目标文件 objectfiles 的二进制内容打印到标准输出上。如果没有指定目标文件 objectfiles，就打印所有目标文件。修改子 v 会打印所有列在其内容之前的名字 |
| q | 将指定目标文件 objectfiles 快速附加到文档末尾，而不检查是否可以取代而不是一味添加。不会更新索引，因此 ranlib 必须在连接库之前使用 |
| r | 将指定目标文件 objectfiles 插入文档。如果指定的所有目标文件都已经在文档中了，就用新的替换老的。如果指定文档不存在，就会创建此文档。默认条件下，新的模块会添加到文件的末尾，但是修改子 a、b 或 i 可用来指定新模块的位置 |
| t | 显示一列文档文件的内容。修改子 v 会为该列表加上时间戳、所有者、组、以及每个模块的大小。如果没有指定目标文件 objectfiles，就会列出整个文档 |
| x | 将指定目标文件展开成正常的磁盘文件。如果没有指定目标文件，就会展开所有文件 |

**表 12-2　修改操作的 ar 选项**

| 选项 | 描述 |
|---|---|
| a | 将新文件添加到命令行指定文件位置之后，该文件是作为位置文件的 |
| b | 将新文件添加到命令行指定文件位置之前，该文件是作为位置文件的。这和选项 i 一样 |
| c | 如果必要就创建文档。只要需要总会创建新的文档，但使用该选项会禁止警告消息 |
| f | 截断文档内的文件名。通常 ar 允许任意长度的文件名，可是这样会导致文档的创建过程在某些系统中不能相互兼容 |
| i | 立即将所有新文件添加到命令行指定的文件位置之前。这和选项 b 一样 |

（续表）

| 选项 | 描述 |
| --- | --- |
| N | 当文档中存在多个指定的目标文件时，count 参数可用来当作指定目标文件的选择子（selector） |
| o | 当文件是从文档中展开时，原始日期可以得到保留 |
| s | 即使没有改动文档，也会创建新的文档索引。在 ar 中会单独使用修改子，这和使用 ranlib 的结果一样 |
| u | 当文件加入文档时，该选项说明只有比文档中现有文件更新的文件才会被添加。该修改子只有和选项 r 在一起才有效 |
| v | 按照详细模式运行，在运行处理过程中显示附加信息 |
| V | 显示版本信息，然后退出 |

### 12.1.3 动态库中的目标文件

动态库中包含的目标文件只有当程序开始运行的时候，才会载入内存并和程序相连。它有两个优点，一是可执行程序小了很多，而且多个程序可以共享同一个动态库载入的共享目标模块（这就是为什么动态库也叫做共享库）。

保存在动态库中的目标文件和常规目标文件有些细微的差别，那些常规目标文件都是用于静态库的。同样，这些文件也希望它们的内部地址可由编译程序产生的代码在其内部进行处理。

## 12.2 连接程序的前端

在面向对象语言中，比如 C++，程序必须能够在主线程序运行之前就能运行静态构造器（static constructors）。但并不是所有连接程序都支持这种设置，因此有必要为连接处理过程增加一个前端 collect2。

几乎在每种系统上，gcc 都会调用工具 collect2，该工具负责连接操作。collect2 处理过程会进行检查，保证静态构造器在主线程序开始之前已经运行过了。为确保执行了静态构造器，collect2 会在临时的.c 源文件中生成特殊的静态器表，编译它，并将它包含进来作为连接的可执行程序的一部分。在 main()函数开始的位置调用__main()来执行静态构造器。

执行 collect2 程序就好像它是连接程序的标识 ld。它利用同样的参数集，并将该参数传递给此标识 ld 去完成实际的连接操作。事实上也许需要连接程序两次——一次用来确定静态构造器的名字（会在连接程序的输出结果中发现它们），第二次用来生成最终的可执行文件。

不仅 collect2 可以调用连接程序标识 ld，也可以用 nm 从目标文件中 demangle 此名并展开它，并可使用 strip 去掉目标文件中的符号。

## 12.3　定位库

要使程序正确连接，连接程序必需能够定位库，该库用来解析外部的引用。对于静态连接程序而言，所有目标文件都集中在一起保存成一个独立的可执行文件，这个可执行文件完全可移植到相兼容的系统中并能正确运行，甚至目标文件都不存在也没有关系。换句话说，连接程序时共享库必须存在，而且每次运行程序的时候也必须存在。

### 12.3.1　在连接时定位库

无论何时，只要连接程序需要查找库，就会查找指定的目录列表。这些目录被包含进查询路径，这依赖于使用哪种竞争模式 ld，编译时如何配置 ld，以及命令行指定的目录。大多数的系统库保存在目录/lib 和/usr/lib 中，因此会自动查找这两个目录。通过使用一个或多个-L 选项，可以指定查询其他目录。例如，下面的命令告诉连接程序要在当前目录和/home/fred/lib 目录中查找所有库，而不使用默认的查找路径：

```
$ gcc -L. -L/home/fred/lib prog.o
```

连接程序先查找共享库再查找静态库。下面的命令会为库 libmilt.so 查找每个目录，然后是库 libmilt.a：

```
$ gcc -lmilt prog.o
```

通过在命令行中指定确切的库名可以限制所有的查找操作。下面的例子使用当前目录的库 libjj.a，然后是/home/fred/lib 中的库 libmilt.so：

```
$ gcc libjj.a /home/fred/lib/libmilt.so prog.o
```

### 12.3.2　运行时载入库

一旦程序被连接，要使用共享库，就必须在运行的时候能够找到共享库的位置。因为是通过名字定位库，而不是通过目录定位库，因此可以在连接程序的时候使用一个库，而在运行的时候使用另一个库。如果改变了库的版本，而没有改变程序版本，当然有可能出问题——这就是为什么大多数的库会把版本号作为名字的一部分（例如，libm.so.6 或 libutil-2.2.4.so）。

无论何时载入程序并打算运行的时候，共享库都应该位于以下位置：

- 环境变量 LD_LIBRARY_PATH 列出的所有用分号分隔的目录
- 文件/etc/ld.so.cache 中找到的库的列表，由工具 ldconfig 维护
- 目录/lib
- 目录/usr/lib

如果想要知道载入了哪个库，以及确切了解应用程序使用的是哪个库，可以使用本章稍后描述的工具 ldd。

另一个环境变量 LD_PRELOAD 包括的共享库名列表（由空格、制表符或新的一行来分隔）会在开始查找其他库之前被预先载入。以此方式，通常可以重载由共享库加载的函数。为安全起见，setuid 程序的技术可对此加些限制。

## 12.4 从共享库中载入函数

共享库中的函数可被加载并执行而不需要连接到程序里。只有按照名字调用所需的一个或多个函数时，才有必要将这些共享库加载到内存中。下面的例子包括保存在共享库中的两个简单函数，然后程序动态载入并依次执行这两个函数。

库中的两个函数向标准输出显示字符串，说明它们被调用了。第一个函数 sayhello，显示它内部声明的串，如下：

```
/* sayhello.c */
#include <stdio.h>
void sayhello()
{
   printf("Hello from a loaded function\n");
}
```

第二个函数 saysomething 需要传入字符串作参数：

```
/* saysomething.c */
#include <stdio.h>
void saysomething(char *string)
{
   printf("%s\n",string);
}
```

使用如下命令将这两个函数编译成位置独立代码，并用它们创建共享库 libsayfn.so：

```
$ gcc -fpic -shared sayhello.c saysomething.c -o libsayfn.so
```

可动态加载这些函数的程序使用了 4 个基本函数。调用 dlopen()将共享库载入内存（如果它还不在内存当中），返回的句柄可用来定位。调用 dlsym()返回函数的地址。调用 dlcose()从共享库中释放当前程序。如果没有其他程序与它相连，就会从内存中卸载该动态库。函数 dlerror()返回描述性字符串，描述最近调用这些函数产生的任何可能的错误。如果没发生错误，函数 dlerror() 返回 NULL。

下面的程序载入共享库 libsayfn.so，并执行其中包含的两个函数：

```
/* say.c */
#include <dlfcn.h>
#include <stdio.h>

int main(int argc,char *argv[])
{
    void *handle;
    char *error;
    void (*sayhello)(void);
    void (*saysomething)(char *);

    handle = dlopen("libsayfn.so",RTLD_LAZY);
    if(error = dlerror()) {
        printf("%s\n",error);
        exit(1);
    }

    sayhello = dlsym(handle,"sayhello");
    if(error = dlerror()) {
        printf("%s\n",error);
        exit(1);
    }

    saysomething = dlsym(handle,"saysomething");
    if(error = dlerror()) {
        printf("%s\n",error);
        exit(1);
    }

    sayhello();
    saysomething("This is something");

    dlclose(handle);
}
```

其中包含了头文件 dlfcn.h，因为它包含函数原型和其他一些定义。函数 main()顶端就声明了句柄，用来定位共享库，字符串指针包含所有错误消息的地址，可在库中找到每个函数的指针。

编译本例的命令行需要包含库，此库含有调用的函数，如下：

```
$ gcc say.c -ldl -o say
```

调用 dlopen()需要所载入的库名，以及标志值，用来指出是如何加载函数的。调用 dlopen()在下面位置查找指定的库：

- 如果库名以斜线（/）开头，会假设此名为绝对路径名，这样的名字必需完全匹配。如果名字不以斜线开头，会继续查找列表中的其他位置。
- 环境变量 LD_LIBRARY_PATH 列出的用分号分隔的所有目录。
- 文件/etc/ld.so.cache 中找到的库的列表，可以由 ldconfig 工具维护。
- 目录/usr/lib。
- 目录/lib。
- 当前目录。

作为调用 dlopen()的第二个参数，标志可为 RTLD_NOW，它让库中的所有函数都载入内存，立即就能使用。另一个选项 RTLD_LAZY 会推后每个函数实际的载入操作，直到它在调用 dlsym()中被应用。这两个标志都可以和 RTLD_GLOBAL 进行与操作（OR），它允许库中所有外部引用都可以通过调用其他动态库中找到的函数解析出来，当然其他动态库也必须是已载入的动态库。

在例子中调用 dlsym()时，利用从 dlopen()中返回的 handle 以及函数名作为参数，返回的是加载的库中函数的地址。一旦函数地址返回并存储在合适的指针后，它就能够被直接调用。

调用 dlopen()和 dlsym()之后，调用 dlerror()进行检查，报告可能出现的任何出错情况。

## 12.5 使用目标文件和库的工具程序

管理库及其中保存的目标文件非常繁杂，依赖于命名约定以及系统的组织级别。利用 gcc 和 ar 的目标管理及库管理能力，有时需要检查二进制文件内容，并根据所发现的重新组织。

### 12.5.1 配置查找共享库

工具 ldconfig 执行两个基本函数处理共享库。首先创建连接，这样到共享库的引用才能总是最新版本。然后在文件/etc/ld.so.cache 中保存有效共享库的完整列表。

工具 ldconfig 读出文件/etc/ld.so.conf，它是包含共享库的目录列表，然后使用这些目录名（还有目录/lib 和/usr/lib）来定位库，这些库用于连接，列在目录/etc/ld.so.cache 中。文件/etc/ld.so.conf 中的目录名可由新的一行、分号、制表符或空格分隔。/etc/ld.so.cache 的内容不是文本格式，也不能被编译。

在构造/etc/ld.so.cache 文件之前，ldconfig 会分析库的名字和内容，并会创建动态连接，这样就会载入最新版本的库。例如，载入 libdl.so.2 的程序通过连接，可能实际载入的是 libdl-2.2.4.so。当公布了修复问题的新版本库（例如，libdl-2.2.5.so 或 libdl-2.3.0.so 后），工具 ldconfig 就会更新连接 libld.so.2 指向新版本。但如果新版本 libdl-3.0.0.so 可能打破旧程

序，那么旧连接就不会受到干扰，而会创建新的连接 libdl.so.3。这种命名约定使得可能在同一环境下运行共享库的新旧两个版本。

因为可能需要访问外设，有必要以 root 登录运行 ldconfig。下面的命令创建所有必要的新连接，生成文件/etc/ld.so.cache 的新版本：

```
% ldconfig -v
```

选项-v 生成所有连接的列表，其中还包括处理过程中的其他信息。完整的选项列表参见表 12-3 的描述。

表 12-3　ldconfig 的命令行选项

| 选项 | 描述 |
|---|---|
| -? | 显示选项列表，然后退出 |
| -C filename | 使用指定文件名保存缓存内容，而不是将缓存内容保存在默认的/etc/ld.so.cache 中 |
| -c fmt | 同选项--format |
| -f filename | 指定输入配置文件的文件名，而不是使用默认的/etc/ld.so.conf |
| --format=fmt | 指出/etc/ld.so.cache 内容的格式。可用的选择为 old、new 和 compat。默认为 compat |
| --help | 显示选项列表，然后退出 |
| -n | 列出命令行指定目录中的库，不产生缓存文件 |
| -N | 指出不要重新生成缓存文件 |
| -p | 同选项--print-cache |
| --print-cache | 显示缓存文件中所有库按字母排序的列表，包括被连接的库的完整路径名 |
| -r directory | 改变到指定目录，并用该目录作为根目录 |
| --usage | 显示命令行语法，然后退出 |
| -v | 生成采取行动的详细列表 |
| -V | 显示版本信息 |
| --verbose | 生成采取行动的详细列表 |
| --version | 显示版本信息 |
| -X | 指出不要创建库名连接 |

### 12.5.2　列举目标文件中的符号名

工具 nm 用来列出目标文件、静态文档库或动态库中定义的（或是引用此处的）符号。如果命令没有指定文件名，会假设使用 a.out。利用命令行选项，可以根据符号的地址、尺寸或名字组织这些符号，而且可以按照很多方式格式化该输出结果。符号也可以被 demangled，产生的结果和源代码中的一样。

作为例子，下面的命令列出目标模块的名字以及库 libc.a 定义或引用的所有符号：

```
$ nm libc.a
```

表 12-4 列出 nm 命令的命令行选项。

**表 12-4 nm 工具的命令行选项**

| 选项 | 描述 |
|---|---|
| -A | 同选项--print-file-name |
| -a | 同选项--debug-syms |
| -B | 同选项--format=bsd。这是默认设置 |
| -C [type] | 同选项--demangle |
| -D | 同选项--dynamic |
| --debug-syms | 显示调试器使用的符号。通常不会显示这些符号 |
| --demangle[=type] | demangles 符号，使它变回源代码中找到的用户级的名字。如果指定类型，会为如下类型之一：auto、gnu、lucid、arm、hp、edg、gnu-v3、java、gnat 或 compaq |
| --dynamic | 对于动态目标，例如共享库，该选项可以显示动态符号而不是普通符号 |
| --extern-only | 只显示定义为外部的符号 |
| -f fmt | 同选项--format |
| --format=fmt | 使用指定的输出格式显示符号。可选的格式包括 bsd、sysv 和 posix，其中 bsd 为默认格式 |
| -g | 同选项--extern-only |
| -h | 显示选项列表，然后退出 |
| --help | 显示选项列表，然后退出 |
| -l | 同选项--line-numbers |
| --line-numbers | 使用文件中保存的调试信息来确定文件名和每个符号的行号 |
| -n | 同选项--numeric-sort |
| --no-sort | 指出不要将符号排序 |
| --numeric-sort | 按照符号地址的数值排序 |
| -o | 同选项--print-file-name |
| -p | 同选项--no-sort |
| -P | 同选项--format=posix |
| --portability | 同选项--format=posix |
| --print-armap | 在列举静态库成员符号时，该选项包含了模块的索引信息及其他信息，正是该模块含有所列符号 |
| --print-file-name | 用源文件的名字标记每个符号，而不是只在文件头命名源文件 |
| -r | 同选项--reverse-sort |

（续表）

| 选项 | 描述 |
|---|---|
| --radix=base | 指出打印符号值的数字的进制。可选的为 d 的十进制、o 的八进制、或 x 的十六进制 |
| --reverse-sort | 反序排列，不论字母还是数字均可 |
| -s | 同选项--print-armap |
| --size-sort | 按照尺寸进行符号排序。计算尺寸取的是下一个符号的最高地址和本符号的地址的差。输出列出的是尺寸而不是通常的地址 |
| -t base | 同选项--radix |
| --target=bfdname | bfdname 是目标文件格式的名字，它不是当前机器的名字。为得到已知格式名字列表，键入命令 objdump –i |
| -u | 同选项--undefined-only |
| --undefined-only | 只显示文件引用但未定义的符号 |
| -V | 同选项--version |
| --version | 显示版本信息，然后退出 |

### 12.5.3　去除目标文件中的无用信息

工具 strip 可以去掉目标文件或命令指定文件的调试符号表信息。该目标文件可以是静态库、共享库、或者是编译程序产生的.o 文件。依赖于文件里包含了多少调试信息，剥离（stripping）可动态地削减文件的尺寸。作为例子，下面的命令会剥离目标文件 main.o 以及 libglom.a 库中所有目标文件的所有调试信息：

```
$ strip main.o libglom.a
```

工具 strip 将现有文件替换为剥离后的版本，因此如果想要保留原始未剥离的版本，需要在剥离之前先保存副本，或者使用-o 选项生成不同的文件名。

工具 strip 的命令行选项列在表 12-5 中。表中的一些选项引用到了 bfdname。它是要求被剥离后的目标文件格式的名字，如果不想使用当前机器的本地格式，就需要使用该选项。为得到所有可用的 bfdnames 列表，可键入命令 objdump -i。

表 12-5　strip 的命令行选项

| 选项 | 描述 |
|---|---|
| --discard-all | 去掉所有非全局符号 |
| --discard-locals | 去掉编译程序生成的局部符号。通常由字母 L 或句号（.）开头 |
| -F bfdname | 同选项--target |
| -g | 同选项--strip-debug |

（续表）

| 选项 | 描述 |
|---|---|
| -h | 显示项目列表，然后退出 |
| --help | 显示项目列表，然后退出 |
| -I bfdname | 同选项--input-target |
| --input-target=bfdname | 将输入目标文件看作指定 bfdname 格式的文件。参见--output-target 和--target |
| -K name | 同选项--keep-symbol |
| --keep-symbol=name | 只将指定符号复制到输出文件中。保存多个名字要多次使用该选项 |
| -N name | 同选项--strip-symbol |
| -O bfdname | 同选项--output-target |
| -o filename | 不是重写原始文件，输出被写到新指定的文件中。该选项会限制命令，只能操作一个文件 |
| --output-target=bfdname | 取代原始文件为剥离（stripped）后的文件，格式满足 bfdname。参见--input-target 和--target |
| -p | 同选项--preserve-dates |
| --preserve-dates | 最新剥离后的文件和原始输入文件一样，具有相同的访问次数 |
| -R name | 同选项--remove-section |
| --remove-section=name | 从目标文件中去除指定的章节。本选项可多次使用以除去多个章节 |
| -s | 同选项--strip-all |
| -S | 同选项--strip-debug |
| --strip-all | 去掉所有符号，包括连接必要的重定位信息 |
| --strip-debug | 只去掉调试所必需的符号 |
| --strip-symbol=name | 去掉指定符号。该选项可多次使用，也可与其他剥离选项一起使用 |
| --strip-unneeded | 去掉对重定位代码没有必要的所有符号 |
| --target=bfdname | 将输入格式和输出格式都设置为指定的 bfdname。参见--input-target 和--output-target |
| -v | 同选项--verbose |
| --verbose | 通过列举所有剥离后的文件，产生更详细的输出 |
| -x | 同选项--discard-all |
| -X | 同选项--discard-locals |

### 12.5.4 列出共享库的依赖关系

工具 ldd 会遍历二进制可执行代码中的目标文件，或者是命令行指定共享库中的目标文件，然后列出所有共享库的依赖关系。例如，如下命令列出 Linux 系统上 bash shell 程序使用的共享库：

```
$ ldd /bin/bash
    libtermcap.so.2 => /lib/libtermcap.so.2 (0x40027000)
    libdl.so.2 => /lib/libdl.so.2 (0x4002b000)
    libc.so.6 => /lib/libc.so.6 (0x4002f000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

每行列出的第一个名字是共享库出现在函数内部的名字，第二个名字就是磁盘上找到的实际运行库的路径名。而库被载入内存的地址出现在行末。bash shell 使用 libtermcap 中的函数将文本显示到屏幕上，而用 libdl 载入并执行共享库中的函数。库 libc 是标准 C 函数库。文件 ld-linux.so 就是程序 ld.so，是共享库的帮助程序，会实际载入并执行共享库。

使用 ldd 来确定程序使用的共享库版本非常方便。另外，使用 ldd 能够确定到共享库的所有未解析引用。例如，如果第 4 章的程序 stwohellos 正确编译，但和它一起编译的共享库没有正确安装，那么 ldd 的输出结果看起来如下：

```
$ ldd stwohellos
    shello.so => not found
    libc.so.6 => /lib/libc.so.6 (0x40027000)
    libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x4015d000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

### 12.5.5　显示目标文件的内部结构

工具 objdump 用于展开目标文件、静态库和共享库中的信息，然后将其表示成可读模式。该命令可用于输出（dump）来自不同格式目标文件的信息。要确定 objdump 是否识别该目标文件格式，键入以下命令：

```
$ objdump -i
```

执行 objdump 从文件中展开信息，必须利用表 12-6 中的一个或多个选项（每个都有短模式和长模式）指出需要展开的是什么信息。表 12-7 列出的附加选项，用来精简输入数据的选择以及格式化输出数据。例如，要列出目标文件 helloworld.o 的头文件和章节头，需要假设输入代码为 big endian，键入如下命令：

```
$ objdump -f -h -EB helloworld.o
```

表 12-6　objdump 的 DUMP 选择项的短格式和长格式

| 短 | 长 | 显示 |
|---|---|---|
| -a | --archive-headers | 文档头文件信息 |
| -d | --disassemble | 可执行代码的汇编语言 |
| -D | --disassemble-all | 可执行代码和数据的汇编语言 |
| -f | --file-headers | 完整文件头的内容 |

（续表）

| 短 | 长 | 显示 |
|---|---|---|
| -g | --debugging | 调试信息 |
| -G | --stabs | 所有 STABS 信息的原始形式（raw form） |
| -h | --section-headers | 章节头的内容 |
| -H | --help | 本选项列表 |
| -i | --info | 支持的目标格式和结构列表 |
| -p | --private-headers | 指定目标格式的文件头内容 |
| -r | --reloc | 重定位信息 |
| -R | --dynamic-reloc | 动态重定位信息 |
| -S | --source | 混合源代码的可执行汇编语言 |
| -s | --full-contents | 混合源代码的所有代码的汇编语言 |
| -t | --syms | 符号表内容 |
| -T | --dynamic-syms | 动态符号表内容 |
| -V | --version | 版本信息 |
| -x | --all-headers | 所有头内容 |

**表 12-7 objdump 修改子的命令行选项**

| 选项 | 描述 |
|---|---|
| --adjust-vma=offset | 将指定位移 offset 添加到所有显示章节的地址中 |
| --architecture=machine | 按照硬件，指出输入目标文件的格式。要确定有效的结构类型，键入 objdump –i |
| -b bfdname | 同选项--target |
| -C type | 同选项--demangle |
| --demangle=type | 假设符号为指定类型且经过 demangled，变回成为源代码中的样式。有效的类型为 auto、gnu、lucid、arm、hp、edg、gnu-v3、java、gnat 和 compaq |
| --disassembler-options=op | 指定 op 为一个或多个传递给分解器（disassembler）的选项 |
| --disassemble-zeroes | 指出分解代码时不要跳过零块 |
| -EB | 同选项--endian=big |
| -EL | 同选项--endian=little |
| --endian=which | 指出输入目标文件为 big-endian 或 little-endian。这里的 little 是指 little endian，而 big 是指 big endian |
| --file-start-context | 使用-S 选项时，该选项会从文件开始就包括上下文信息 |
| -j name | 同选项--section |

（续表）

| 选项 | 描述 |
| --- | --- |
| -l | 同选项--line-numbers |
| --line-numbers | 指出输入的行号和文件名 |
| -M | 同选项–disassembler-option |
| -m machine | 同选项–architecture |
| --prefix-addresses | 打印每个分解的指令附带的完整地址信息 |
| --section=name | 限制目标文件指定章节的信息显示 |
| --show-raw-insn | 显示十六进制的操作代码（opcodes）和记忆的汇编语言指令 |
| --start-address=address | 只处理地址大于指定地址值的数据 |
| --stop-address=address | 只处理地址小于指定地址值的数据 |
| --target bfdname | 指出输入目标文件的格式。要确定格式是否有效，键入命令 objdump –i |
| -w | 同选项--wide |
| --wide | 格式化超过 80 列的输出 |
| -z | 同选项--disassemble-zeroes |

# 第 13 章

# 使用 GNU 调试器

工具程序 gdb 是 GNU 调试器，该命令行调试器可用于全面控制和检查运行中的程序。所有程序都会对 gdb 发出的命令有所响应，但只有按照合适选项编译并连接的程序才能包含足够信息，而这些信息才能够连接到源文件上，用于跟踪程序流程。启动交互调试过程最简单的方法就是在命令行中直接指出将由 gdb 启动该程序，当然这和先启动调试器然后再载入程序的结果一样。也可指出要将调试器自己附带到运行程序上，这样只有在程序运行一次以后，才可能对程序内部的异常处理过程进行检查。使用 gdb 的第三种目的是为了对已崩溃的程序进行事后检查，以确定崩溃的原因。

## 13.1 调试信息格式

为了能够调试程序，有必要将程序相关信息包含到目标文件中。调试器利用这些信息，可将源代码和可执行代码联系起来，并按格式提供程序有关信息，因此可以确切了解到程序的动作。没有这些信息，所有调试器都只知道执行的绝对二进制地址和机器语言代码——用这些信息很难将它们与程序的源代码联系起来。

这种保存在目标文件中的信息有多种格式。工作正常的调试器都能够理解调试数据的格式。所幸的是 gdb 调试器理解多种格式，而且也理解 gcc 插入的一些特殊扩展功能。

### 13.1.1 STABS

STABS 格式的调试信息最早用于 Pascal 语言调试器，但该格式被证明为非常有用，因此在其他调试器中也得到了广泛应用。

gcc 编译程序对其生成的汇编语言添加了 STABS（符号表）调试信息，由汇编器（assembler）将该信息包含到目标代码中。汇编器将符号表和字符串表的 STABS 信息附带在每个.o 文件之后。连接程序将这些.o 文件组合到可执行文件中，将这些表格组合成一个独立的符号表，调试器就用该表来标识可执行代码的章节（sections）。

下面的三个调试器指示字可用于创建满足以下格式的符号表：

```
.stabs "name:symdesc=typeinfo",type,other,description,value
.stabn type,other,description,value
.stabd type,other,description
```

每个指示字都由 type 域提供基本信息（例如该指示字是否为新定义的，或者是到已知定义的引用）。type 域也指出了 other 域和 description 域内容的意义。域 value 是为定义分配的值。

指示字.stabs 定义了符号表中的字符串。引号内部的 name 就是插入表中的符号。symdesc 是单独一个字符（例如 F 是全局函数，G 为全局变量，而 t 为类型名）以及类型数字（实际上，它可以为两个数字），它定义了新类型的符号，或者是到以前定义的类型数字的引用。typeinfo 提供了关于类型的更进一步的信息，例如数字的范围和大小。

指示字.stabn 定义了数字的值。

指示字.stabd 定义了当前地址的标签（就是指示字位置的地址）。它没有确定值，因为可由指示字位置得到它。

可以浏览 http://sources.redhat.com/gdb/onlinedocs/stabs.html 中的文档，当中描述了完整的 STABS 格式。

### 13.1.2　DWARF

DWARF 格式的调试信息已经发展为第二代 DWARF2 了，而且正向 DWARF3 标准过渡。这些版本之间在编码上有些差别，因此它们彼此并不兼容，但 gdb 调试器能够识别并读出原始的 DWARF 和 DWARF2。

调试信息是由代码中某些特殊章节的汇编语言产生的，例如.debug_pubnames，.debug_aranges，.debug_info 或.debug。这些特殊章节包含的数据和可执行代码用于从运行中的程序里识别和析取出相关信息。连接程序将名字相同的章节组成组，命名为目标代码中独立的块，该块可用来指出项目的位置，并建立目标代码地址和源代码行之间的关系。

可在 http://services.worldnet.fr/~stcarrez/ dwarf2.pdf 上查看有关 DWARF2 的信息。

### 13.1.3　COFF

公共目标文件格式（Common Object File Format，COFF）有时也叫做 a.out 格式，是 UNIX System V 以及许多衍生系统中目标文件的标准格式。这是微软的 DOS 和 Windows 操作系统采用的格式。而 Linux 支持的格式叫做 ELF。

COFF 格式不会包含专为调试而设计的信息——这些信息主要用于连接——但它包含了很多调试器需要的信息。符号表包含所有可重定位的符号，而可重定位表包含的是到符号表中数据类型的记录和信息的引用。它也包括行号信息，可用于联系可执行二进制代码和原始的源代码。符号表包括每个符号的完整描述，以及大小和描述信息。

COFF 格式将目标分成不同的章节（sections）。其中的.text 章节包含了可执行的代码，

而.data 章节包含的是具有初始化值的变量，.bss 部分的则是未被初始化的数据。这样划分的本质原因是如果该程序同时运行多个进程，可在内存中共享相同的.text 部分，而且.data 章节也可作为一个单独的程序块载入内存，其中都是初始值，而.bss 章节只作为独立的数字（大小）就可存在文件中，可在加载程序时将其扩展到正确尺寸。

这种格式包含的信息与 STABS 或 DWARF 包含的扩展信息不同，因此经常会看到一些基本 COFF 文件，其中插入了 STABS 或 DWARF 信息，这样可用于更高级的调试器。

### 13.1.4 XCOFF

XCOFF 目标文件格式是基本 COFF 格式的扩展。XCOFF 格式提供到表格和动态连接的适当引用。而且 XCOFF 分别包括 32-bit 或 64-bit 模式的目标代码。

XCOFF 的基本格式和 COFF 相同，但 XCOFF 格式还包含保存在.debug 章节中的 STABS 串，而不是像 COFF 那样将其保存在字串表中。就是说 XCOFF 格式混合了 COFF 和 STABS，当需要按照 COFF 格式插入 STABS 时，会留下一些 COFF 片断，因此不会有重复的数据。

## 13.2 编译调试用程序

为使调试器可以连接二进制可执行代码和源代码——需要显示可以读懂的信息——编译程序必需指示将这些信息包含到目标代码中。可通过设置命令行选项实现它，指定包含信息的数目和类型。

包含信息的数目由级别数控制，见表 13-1。设置级别数跟在选项标签后面，见表 13-2。

**表 13-1 目标文件中调试信息的三个级别**

| 级别 | 描述 |
|---|---|
| 1 | 该级别在目标代码中插入的信息最少。虽然这些信息足够跟踪函数调用和检查全局变量，但没有可执行代码和源代码相关信息，也没有足够信息可以检查局部变量 |
| 2 | 这是默认级别。该级别包括级别 1 的所有信息，而且还添加了从源代码行到可执行代码相关的必要信息，以及局部变量的名字和位置 |
| 3 | 该级别包括级别 1 和级别 2 的所有信息，此外还加入了额外的信息，包括预处理宏定义 |

**表 13-2 用于插入调试信息的 gcc 选项列表**

| 选项 | 描述 |
|---|---|
| -g[level] | 生成的调试信息格式是本地系统的格式。GNU 调试器可以按照这种格式工作，其他调试器也可以。在使用 STABS 格式的系统中，该选项可为 gdb 生成额外信息，但却可能造成其他调试器失败。该选项默认的级别为 2 |

（续表）

| 选项 | 描述 |
|---|---|
| -ggdb[level] | 按照默认格式产生调试信息，如果可能也会包括 gdb 功能扩展。会按照可以使用的最好格式处理该信息——如果没有指明 STABS 和 DWARF2，会使用本地格式 |
| -gstabs[level] | 产生 STABS 格式的调试信息（如果有效） |
| -gstabs+ | 产生 STABS 格式的调试信息（如果有效）以及只有 gdb 调试器理解的扩展功能。这些扩展功能可能造成其他调试器失效 |
| -gcoff[level] | 产生 COFF 格式的目标代码和调试信息（如果有效）。该模式最常用于版本 4（Release 4）之前的系统 5（System V） |
| -gxcoff[level] | 产生 XCOFF 格式的目标代码和调试信息（如果有效） |
| -gxcoff+ | 产生 XCOFF 格式的目标代码和调试信息（如果有效）以及只有 gdb 调试器理解的其他扩展功能。这些扩展功能可能造成其他非 GNU 调试器失效 |
| -gdwarf | 产生 DWARF 版本 1 格式的调试信息（如果有效）。该模式最常用于版本 4（Release 4）之前的系统 5（System V） |
| -gdwarf+ | 产生 DWARF 版本 1 格式的调试信息（如果有效）以及只有 gdb 调试器理解的其他扩展功能。这些扩展功能可能造成其他非 GNU 调试器失效 |
| -gdwarf-2 | 产生 DWARF 版本 2 格式的调试信息（如果有效） |
| -gvms[level] | 产生 VMS 调试格式的调试信息（如果有效）。DEC VMS 系统使用该格式 |

目标文件中的调试信息格式根据每种平台目标代码本地格式的不同而有所区别。gdb 调试器能够识别很多不同的格式，也能和它们一起工作。使用 STABS 格式的系统包括额外的调试信息，只能被 gdb 识别。

表 13-2 列出的 gcc 命令行选项可用于指示编译程序在目标代码中插入调试信息。可以和-O 优化选项一起使用调试选项，但应注意优化是会重新调整（甚至删除）代码的，这样跟踪程序的逻辑流程就会有困难。但是，调试程序的优化版本就不会有问题。

表 13-2 中的一些命令行选项允许增加级别数，另外一些则不允许。对于不允许增加级别数的命令行选项，可以使用独立的选项-g 指定级别。例如，指出-gstabs+ 设置级别为 3，可用如下的选项序列：

```
$ gcc -g3 -gstabs+ ...
```

## 13.3　将程序载入调试器

在 gdb 命令行中指出程序就足够将程序载入内存，并准备调试了。程序载入并不会开始运行，直到命令要求运行程序才行。这种暂停就允许设置断点（程序运行到断点位置可

被暂时终止）以及其他准备工作，例如指定某些变量，要求跟踪进入程序之后显示其值。

如下是调试器运行程序的例子，显示了接口如何工作，以及用于监控程序运行的一套基本命令。C 程序 fibonacci.c 显示了 Fibonacci 序列的前 20 条项目：

```
/* fibonacci.c */

int current;
int next;
int nextnext;

void calcnext();
void setstart();


int main(int argc,char *argv[])
{
    int i;

    setstart();
    for(i=0; i<20; i++) {
        printf("%2d: %d\n",i+1,current);
        calcnext();
    }
    return(0);
}
void setstart()
{
    current = 0;
    next = 1;
}
void calcnext()
{
    nextnext = current + next;
    current = next;
    next = nextnext;
}
```

要编译包含调试信息的程序，有必要使用-g 选项，如下：

```
$ gcc -g fibonacci.c -o fibonacci
```

gdb 没有必要访问源文件才能生成诊断调试信息，因为所需的东西都被包含进目标文件了。但如果启动调试器的时候发现了源文件，就要去检查该源文件是否和目标文件完全匹配。如果发现不匹配，会显示警告消息。

如下简单的调试过程会加载程序，在入口程序 main()处设置断点，设置连续显示两个

变量。然后启动程序，程序会运行直到到达断点位置，这里 step 和 next 命令都可用于一次运行一行程序：

```
$ gdb fibonacci
(gdb) break main
Breakpoint 1 at 0x80483a0: file fibonacci.c, line 14.
(gdb) display current
(gdb) display next
(gdb) run
Starting program: /home/fred/progs/fibonacci

Breakpoint 1, main (argc=1, argv=0xbffffa9c) at fibonacci.c:14
14      setstart();
2: next = 0
1: current = 0
(gdb) step
setstart () at fibonacci.c:23
23      current = 0;
2: next = 0
1: current = 0
(gdb) step
24      next = 1;
2: next = 0
1: current = 0
(gdb) step
25  }
2: next = 1
1: current = 0
(gdb) step
main (argc=1, argv=0xbffffa9c) at fibonacci.c:15
15      for(i=0; i<20; i++) {
2: next = 1
1: current = 0
(gdb) step
16          printf("%2d: %d\n",i+1,current);
2: next = 1
1: current = 0
(gdb) next
17          calcnext();
2: next = 1
1: current = 0
(gdb) step
calcnext () at fibonacci.c:28
28      nextnext = current + next;
```

```
2: next = 1
1: current = 0
(gdb) step
29      current = next;
2: next = 1
1: current = 0
(gdb) step
30      next = nextnext;
2: next = 1
1: current = 1
(gdb) step
31  }
2: next = 1
1: current = 1
(gdb) bt
#0  calcnext () at fibonacci.c:31
#1  0x080483d4 in main (argc=1, argv=0xbffffa9c) at fibonacci.c:17
#2  0x40042316 in __libc_start_main (main=0x8048390 <main>, argc=1,
    ubp_av=0xbffffa9c, init=0x8048230 <_init>, fini=0x8048460 <_fini>,
    rtld_fini=0x4000d2fc <_dl_fini>, stack_end=0xbffffa8c)
    at ../sysdeps/generic/libc-start.c:129
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

gdb 执行的第一条操作是载入可执行程序。调试器会等待你键入命令。载入的程序不会运行。在载入程序以后，gdb 会展开调试信息，建立自己内部表的集合，这就意味着调试器知道所有东西的名字和位置（假设它按照选项-g 进行编译并连接），并且已经准备好进行分析。

在程序开始运行之前，利用 display 命令指示调试器显示变量 current 和 next 的名字和值。这些变量会在程序每次停止的时候显示出来。

如果从这一点开始运行程序而不加以干预，程序就会一直运行直到结束。调试器的目的就是检查程序的执行过程，就是说有必要选择某点来中止程序，可察看程序且步步跟踪指令。本例中命令 break main 会在函数 main()的入口位置设置断点。通过列出地址和设置断点的行号，调试器可获知命令。显示第 14 行的源代码，它调用了函数 setstart()，用来设置 Fibonacci 序列中的两个内部变量。

run 命令启动程序。由 gcc 向所有程序插入的初始化代码开始执行。程序会运行初始化代码直到最后调用 main()函数。只要到达第 14 行源代码的位置，程序就会冻结了，显示两个值，而 gdb 会发出提示要求新的命令。

step 命令一次执行一行源代码。从汇编指令看它通常代表多条指令，因此 gdb 会执行

必要多的指令来完成一行源代码创建的所有指令。本例中，第一个 step 命令调用函数 setstart() 然后在函数的第一行停止，它在源文件的第 23 行。下面两个 step 命令执行函数中的两条语句，依次设置了内部变量 current 和 next 为初始值 1 和 0。下面的 step 命令执行返回语句，从函数中返回并停留在 for 循环的顶端。

一旦 step 语句进入循环顶端，程序就会中止在调用 printf()函数的位置。如果这里又用了 step 语句，就会进入 printf()函数，可能这并不是你所希望的。除非多花点力气，按照-g 选项编译 printf()函数，这样该函数就不会包含调试信息，尽管调试器可以单步跟踪函数，也不可能显示它的值或代码。如果不使用 step，可用 next 命令将函数调用看作是一行代码来执行，并且会停在函数调用后的代码行。

一连串的 step 语句可通过反复循环来执行语句。该过程可在检查程序操作之后继续，用以发现计算出错的位置。可交互地对程序进行检查，看它是否按照计划运行。

本例中，bt 命令用于生成函数调用的向后跟踪（backtrace），它会列出执行路径，其后是运行程序。向后跟踪找到的信息不只包括函数的名字，还包括传给每个函数的参数名和参数值，以及函数所在的源代码文件。

## 13.4　执行事后检查

在 UNIX 系统中，崩溃的程序可以激活操作系统中的函数，将程序的内容完全复制到文件 core 中。如果程序是按照选项-g 编译的，确定哪处代码导致程序崩溃就会相对简单。

每次运行下列程序的时候都会崩溃，因为它企图将信息保存到内存的零地址处，这是所有程序禁止访问的区域：

```
/* falldown.c */

char **nowhere;
void setbad();

int main(int argc,char *argv[])
{
    setbad();
    printf("%s\n",*nowhere);
}
void setbad()
{
    nowhere = 0;
    *nowhere = "This is a string\n";
}
```

可以用以下命令编译并运行程序，产生的 core 文件包括运行程序的映像：

```
$ gcc -g falldown.c -o falldown
$ falldown
Segmentation fault (core dumped)
```

要指导 gdb 加载程序，同时也要载入它产生的 core 文件，键入下面的命令：

```
$ gdb falldown core
```

大多数情况下，该命令可提供所有需要的信息，因为它会立即列出程序崩溃的位置的代码行号。如下的调试会话展示了 gdb 显示的信息，以及如何使用其他命令展开所需的更多信息：

```
$ gdb falldown core
Core was generated by  falldown'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x080483d0 in setbad () at falldown.c:14
14      *nowhere = "This is a string\n";
(gdb) print nowhere
$1 = (char **) 0x0
(gdb) bt
#0  0x080483d0 in setbad () at falldown.c:14
#1  0x080483a5 in main (argc=1, argv=0xbffffa8c) at falldown.c:8
#2  0x40042316 in __libc_start_main (main=0x8048390 <main>, argc=1,
   ubp_av=0xbffffa8c, init=0x8048230 <_init>, fini=0x8048410 <_fini>,
   rtld_fini=0x4000d2fc <_dl_fini>, stack_end=0xbffffa7c)
   at ../sysdeps/generic/libc-start.c:129
(gdb) quit
$
```

本例中，出问题的代码行（在此处，要将字符串的地址保存为绝对地址零），包括函数的名字和它所在的源文件名，会被打印出来。print 命令用于检查 nowhere 指针是否设置为非法地址，而命令 bt 用于生成向后跟踪，展示程序是怎样导致自己进入这种出错状态的。虽然还有很多命令都可用于检查变量内容，但通常现有的信息已经足够立即解决问题了。

## 13.5 将调试器附带到运行程序上

能够将调试器附带到运行程序上是非常有用的。例如，如果程序运行一段时间之后进

入无响应的循环，就可将 gdb 附带到程序上，来检查程序循环的具体位置。另一种情况就是交互式的程序突然开始做一些不应该做的事情——可以将调试器附带到程序上，来检查产生奇怪动作的原因。

将调试器附带到运行程序上必需满足两个先决条件。首先，必需按照-g 选项的形式编译进程。第二，必需确定运行进程的进程号（Process ID，PID）。如果不知道 PID，可用 ps 命令找到它。ps 命令的命令行参数根据系统不同而有所区别，因为不同的操作系统按照不同的格式提供运行时进程的信息，但如下格式是典型格式，可确定进程 looper 的 PID 是 29627：

```
$ ps ax | grep looper
29627 pts/4    R      1.58 looper
32298 pts/4    S      0:00 grep looper
```

ps 命令的输出结果也指出了 looper 进程是活动的（R 意味着 running）。事实上，程序 looper.c 就是一个连续的循环，用来展示 gdb 这种可将自己附带到进程的能力：

```
/* looper.c */
void goaround(int);
int main(int argc,char *argv[])
{
   printf("started\n");
   goaround(20);
   printf("done\n");
}
void goaround(int counter)
{
   int i = 0;

   while(i < counter) {
      if(i++ == 17)
         i = 10;
   }
}
```

程序 looper 的主线调用函数 goaround()永远不会返回，因为 i 永远不会到达 counter 值。通过下面的命令可将程序编译成包含调试信息：

```
$ gcc -g looper.c -o looper
```

键入如下命令，启动后台进程：

```
$ looper &
```

通常内核 shell 程序用于启动程序，会显示新进程的 PID，但如果没有显示它的 PID 数，

可以使用 ps 来确定该数字。以下序列展示了如何将调试器附带到进程上，以及如何利用它来发现问题。命令行指出了磁盘上二进制文件的名字以及它的 PID：

```
$ gdb looper 29627
Attaching to program: /home/fred/looper, process 29627
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x080483ea in goaround (counter=20) at looper.c:14
14          if(i++ == 17)
(gdb) display i
1: i = 14
(gdb) step
13      while(i < counter) {
1: i = 15
(gdb) step
14          if(i++ == 17)
1: i = 15
(gdb) step
13      while(i < counter) {
1: i = 16
(gdb) step
14          if(i++ == 17)
1: i = 16
(gdb) step
13      while(i < counter) {
1: i = 17
(gdb) step
14          if(i++ == 17)
1: i = 17
(gdb) step
15              i = 10;
1: i = 18
(gdb) step
13      while(i < counter) {
1: i = 10
(gdb) step
14          if(i++ == 17)
1: i = 10
(gdb) step
13      while(i < counter) {
1: i = 11
(gdb) quit
```

```
The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: /home/fred/looper, process 29627
$
```

开始时，调试器从磁盘中读取二进制文件，并按照输出的头几行内容所指示的，从程序中加载符号表信息，从程序使用的库中加载符号表。如果所有库都被编译为包含调试信息，那么需要的话就可以跟踪进入每个用到的库，但这里只简单展示了 goaround() 函数里的循环。

然后，gdb 调试器将它自己附带到进程上，并在当前位置冻结，显示当前的源代码行及其行号。这里可以随便设置断点、检查变量、或者任何其他正常的调试操作。在本例中，命令 display 指示 gdb 在执行每条指令时显示 i 的值。一连串的单步跟踪命令 step 用来跟踪该值在几次反复循环中的变化，就会发现 i 被重置了，因此不会退出循环。

在本代码段的结尾，使用 quit 命令，调试器将自己从进程上卸下来，并允许程序继续运行。本例中，只要 gdb 卸掉自己，程序就会继续循环，直到使用其他方法中止循环。进程完全恢复正常运行，而且可在任意时刻再次将 gdb 附带到进程上。looper 进程可在 quit 命令之前通过键入 kill 命令终止。

## 13.6　命令总结

gdb 调试器有大量可用的命令。要察看这些命令，只能在 gdb 提示下键入 help 命令。这会列出命令分类列表。有些包括命令的描述，有些包括一些子目录含有更详细的列表。

表 13-3 列出一些有用的命令，大多数调试过程可能都会需要它们。

**表 13-3　一些更有用的 gdb 命令**

| 命令 | 描述 |
|---|---|
| awatch | 设置察看点（watch point），无论何时指定位置的值被读或写了，程序执行都会停止。参见 rwatch 和 watch |
| backtrace | 向后打印出栈的内容，显示将程序带到当前位置的函数调用和参数值。该命令可简写为 bt |
| break | 设置断点，程序运行会在指定行或函数名处中断 |
| clear | 清除某行或函数上的断点，开始是由 break 命令设置的 |
| continue | 继续程序的执行，开始是由调试器中断的 |
| Ctrl-C | 中断运行程序，就好像当前行设置了断点一样 |
| disable | 禁止按照数字列出的断点 |
| display | 每次程序中断时，显示指定表达式的值 |

（续表）

| 命令 | 描述 |
| --- | --- |
| enable | 按照数字激活断点 |
| finish | 继续运行被调试器中断的程序，继续运行直到当前函数返回 |
| ignore | 设置断点的忽略数目。例如，命令 ignore 4 23 需要断点 4 运行 23 次才能起到中断的作用 |
| info breakpoints | 列出状态和描述，包括所有断点的号码 |
| info display | 列出状态和描述，包括前面定义的 display 命令的号码 |
| kill | 杀掉正在运行的当前进程 |
| list | 列出 10 行代码。如果命令中没有参数，列出当前行后的 10 行。如果指定函数，列出该函数开始的 10 行。如果指定行号，行号会在列表中间 |
| load | 将指定的可执行文件动态载入 gdb，准备调试该程序 |
| next | 继续运行被中断的程序，依次运行源代码中的指令，但将函数调用看作是一行代码，不会停止，直到函数返回 |
| nexti | 继续运行被中断的程序，运行一条汇编指令代码，但将函数调用看作是一行代码，不会停止，直到函数返回 |
| print | 立即显示指定表达式的值 |
| ptype | 显示指定项目的类型 |
| return | 强制从当前函数立即返回 |
| run | 启动程序，从开始位置运行程序 |
| rwatch | 设置察看点，只要读到了指定位置，就可以中断运行。参见 awatch 和 watch |
| set | 将指定变量值赋给表达式。例如，set nval=54 会将值 54 保存到 nval 在内存中的位置 |
| step | 继续执行中断的程序，依次运行源代码中的指令。它会进入被调函数 |
| stepi | 继续执行中断的程序，依次运行汇编语言的指令。它会进入被调函数 |
| txbreak | 在当前函数出口设置临时断点（只起一次作用）。参见 xbreak |
| undisplay | 删除数字指定的 display 表达式 |
| watch | 设置察看点，只要写到了指定位置，就可以中断程序运行。参见 rwatch 和 awatch |
| whatis | 打印数据类型和指定表达式的值 |
| xbreak | 在当前函数出口设置断点。参见 txbreak |

# 第 14 章

# make 和 Autoconf

本章介绍 make 工具程序的功能，可用它来管理软件开发项目；还会介绍 Autoconf，它可用来配置并打包开放源码软件，以便发布。本章不会介绍这些工具的所有功能，但所介绍的功能已经足够了，程序员可以轻松推断出完整的开发环境需要的内容。本章会展示它们的基本目的及基础操作。

## 14.1 make

工具程序 make 是软件开发中应用最广的工具。关于 make 最本质的想法非常简单：检查源代码和目标文件，以确定哪个源文件需要重新编译以创建新的目标文件。make 会假设所有改动过的需要重新编译的源文件都比已经存在的目标文件新。make 所做的每件事情都基于这样的基本操作。目标文件到源文件的关系用于产生依赖关系（dependency）。与依赖关系相关的命令生成的目标文件叫做目标（target）。

为确定依赖关系，make 会读取定义依赖关系的脚本。脚本通常叫做 makefile 或 Makefile。脚本包括命令依赖关系，这些命令会将源文件翻译成目标文件。例如，如下的 makefile 记录指出程序 frammis 依赖于源文件 frammis.c，而且它也指明要用 gcc 命令创建目标 frammis：

```
frammis: frammis.c
    gcc frammis.c -o frammis
```

需要指出的是 make 早在 UNIX 初期就出现了，因此它也保留了 UNIX 初期一些奇怪的命令格式，该命令跟在依赖关系命令行之后——命令行必需用制表符缩进。制表符在屏幕上和打印的时候尽管不可见，但它确实是 makefile 脚本语法的一部分。如果没有使用制表符（或用空格代替），会收到“missing separator”消息，还好该消息会指出缺少制表符的行号。

经常会有某个文件依赖的文件是其他依赖关系产生的。例如，如下依赖关系编译程序 frammis：

```
frammis: frammis.o
   gcc frammis.o -o frammis

frammis.o: frammis.c
   gcc -c frammis.c -o frammis.o
```

在本例中，可执行程序 frammis 依赖于 frammis.o，而后者按照定义依赖于 frammis.c。当 make 开始运行之后，会读出完整的 makefile 文件，并由依赖的关系链构造出内部树，文件的第一个依赖关系就为树的根。在本例中根是 frammis 依赖关系，而树中的 frammis.o 依赖关系在它的下面。一旦整个树构造成功，程序会从树的根开始降到更低的级别，然后倒着向后执行工作命令，直到到达树根，而且所有依赖关系都已满足。

应该指出 make 只会构造并执行一个内部树，因此是有可能不会执行文件中定义的某个依赖关系的，只要该依赖关系不能连接到文件中的第一个依赖关系就有这个可能。它所带来的限制并不像初看起来的那么多，因为总可以将某个特殊的依赖关系插进来，它会包含其他所有依赖关系，如下例所示，这里 all 用作哑目标（dummy target）：

```
all: frammis cooker

frammis: frammis.c frammis.h
   gcc frammis.c -o frammis

cooker: cooker.c cooker.h
   gcc cooker.c -o cooker
```

目标 all 依赖于 frammis 和 cooker。尽管目标 all 有两个依赖关系，而且没有相关命令，但由于惟一目的就是要强制满足依赖关系，因此这样也是可以的。内部的 make 树用 all 作为根，用 cooker 和 frammis 作为它下面的节点。

其他不是依赖关系的项目以及它们的相关命令也可以包含到 makefile 中，但它们存在的惟一目的也是为了定义依赖关系和命令。

用下面简单命令激活 makefile：

```
$ make
```

默认情况下，make 会首先查找当前目录，寻找 makefile 文件。如果没有找到 makefile，它会查找文件 Makefile。如果这两个文件都没有找到，就不会采取任何动作。可以选择在命令行中指定文件的名字，如下：

```
$ make -f mymakefile.text
```

### 14.1.1　内部定义

为便于基于目标和依赖关系构造规则，有可能使用预定义的宏，隐式地为 make 建立规则，将一种文件类型转换成另一种文件类型。

**宏**

可按三种不同方式定义宏。下面 makefile 中的目标显示了如何定义和使用宏：

```
showmacros:
    echo HOME is $(HOME)   # defined as an environment variable
    echo COMPILE.f is $(COMPILE.f) #defined as a makefile default
    echo HERBERT is $(HERBERT)   # defined locally in the makefile
```

读 makefile 的时候，无论什么时候 make 遇到#字符，这一行剩下的内容都会被当作注释而忽略掉。

目标 showmacros 总会执行它的相关命令，因为它没有依赖关系，默认会假设必需创建目标。变量的内容可以解析出来，用在语句当中，前面用美元符号（$）打头，并用圆括号将变量包起来。本例中 HOME 的值来自设置的环境变量，COMPILE.f 是 GNU make 自己定义的名字，而在 makefile 中定义 HERBERT 的语句如下：

```
HERBERT=Herbivore
```

makefile 的输出看起来如下：

```
echo HOME is /home/arthur
HOME is /home/arthur
echo COMPILE.f is f77   -c
COMPILE.f is f77 -c
echo HERBERT is herbivore
HERBERT is herbivore
```

在产生输出之前，总会用 echo 命令显示输出，因为 make 的默认模式是要在执行每条命令之前显示这些命令。

**文件后缀规则**

可以指定通过文件名后缀识别文件类型的规则，自动将一种文件类型转换成其他的文件类型。如下的例子是一个 makefile ，它可以识别三种后缀，定义一对命令可将一种后缀的文件转换成新的目标后缀的文件：

```
all: hello.o hello.s

hello.o: hello.c
```

```
hello.s: hello.c

.SUFFIXES: .o .c .s

.c.o:
   gcc -c $ <

.c.s:
   gcc -S $ <
```

设计的 makefile 有两个目标： hello.o 和 hello.s。因为创建这些目标的规则没有和它们相关的命令，所以可以识别三种文件后缀.c、.o 和.s。后缀规则.c.o 会将.c 后缀的文件转换为.o 后缀的文件，而后缀规则.c.s 的命令会将.c 后缀的文件转换为.s 后缀的文件。特殊的宏$<是到用于构造目标的文件名的引用。结果和将如下内容加入 makefile 一样：

```
hello.o: hello.c
   gcc -c hello.c
hello.s: hello.c
   gcc -S hello.c
```

后缀规则通常比上例要复杂得多，但一般不会需要自己编写它。大量的后缀规则被嵌入 GNU make 当中——这些规则多到无论需要什么特殊规则，只需拼出命令就可以了。

**浏览定义**

存在这样的命令行选项，可在运行 make 时看到定义的宏及后缀的完整列表。选项-p 会读出 makefile，正常运行，但会列出 makefile 的所有规则，以及宏和后缀规则。为看到完整列表，键入如下命令：

```
$ make -p | more
```

要看同样的列表而不执行 makefile 命令，可以键入如下命令：

```
$ make -p -q | more
```

如果不仅想察看嵌入 GNU make 内部的定义，还希望察看本地 makefile 的定义，可用下面方法令 make 只读出一个空的 makefile：

```
$ make -p -f /dev/null | more
```

### 14.1.2 如何编写 makefile

如果你是编写 make 程序的描述文件的新手，最好的办法就是复制一个现成的文件，再将它修改成需要的样子。在修改几个 makefile 之后，你就会对它的基本格式有感觉了。如果希望学习有关 make 的更多知识，能够自己编写 make 程序的描述文件，可能需要一些

时间研究和练习。实际上并不困难，但之间的差别会令人困惑不解，直到你掌握了决窍为止。

你可能会想到创建一个 makefile 框架，初学者能够用它创建新的 makefile，但不幸的是没有一个万能的格式可以适用于所有情况。以下是个较普通的 makefile 例子，它会编译两个 C 程序，并将它们连接成一个可执行程序：

```
CC=gcc
PROGS=howdy hello
CFLAGS=-Wall

all: $(PROGS)

howdy: howdy.c

hello: hello.c
    $(CC) $(CFLAGS) hello.c -o hello

clean:
    rm -f *.o
    rm -f *.so
    rm -f *.a
    rm -f $(PROGS)
```

CC 变量被设置为 gcc，而 CFLAGS 被设置为-Wall。目标名字的列表保存在变量 PROGS 中。makefile 按照几乎同样的方式编译这两个程序，但其中的一个默认使用内嵌命令而另一个使用显式的命令。成功运行 make 的输出看起来如下：

```
gcc -Wall   howdy.c   -o howdy
gcc -Wall hello.c -o hello
```

可在任何时刻激活一个简单的 clean 目标，删掉 makefile 生成的所有文件。目前的命令集合不会在磁盘上留下任何.o、.so 或.a 文件，因此这些命令并没有目的。但选项-f 指示 rm 不要在文件不存在的情况下发出警告，而 make 程序的描述文件随着项目的增大，可以生成所有的中间文件。make 工具默认会构造文件中的第一个目标，但只要在命令行中指定，可以构造任何一个目标，如下：

```
$ make clean
```

编译程序有点帮助。第 18 章包括的例子用编译程序生成插入 makefile 的依赖关系列表。

### 14.1.3　make 的选项

UNIX 有很多版本的 make。所有版本的基础都一样，但会在各处添加一些特殊功能和特征。而 make 的 GNU 版本的优点在于可以免费得到源代码格式，尽管它包含了自己的扩展功能，如果和 GCC 一起工作，它可能是最好的一种。尤其如果要从源代码开始创建 GCC，

由查询 binutils 开始比较明智(它包括 make 和其他的一些工具),因为可以确保它们和 GCC 相兼容。由于大多数命令行选项普遍可被所有版本 make 识别,GNU make 的所有选项都列在表 14-1 中。

表 14-1 make 的命令行选项

| 选项 | 描述 |
|---|---|
| --assume-old=filename | 指出不要 remake 指定文件,无需考虑它的改动时间,也不要 remake 任何与此依赖的其他文件 |
| --assume-new=filename | 假设指定文件为新文件,所有依赖于它的目标都会重新构造 |
| -C directory | 查找文件确定其依赖关系之前,改动指定目录 |
| --directory=directory | 同-C |
| -d | 同--debug=a |
| --debug[=flags] | 在表单中显示处理信息,对于调试 makefile 的错误非常有用。如果没有指定标记,会显示基本的调试信息。标记的值可为以下的字母组合:<br>a 显示调试信息的所有类型。这是非常详细的选项。<br>b 显示基本信息,包括过时目标列表,不论命令是否成功。<br>i 显示查找每个目标和 b 标记信息的隐含规则。<br>j 显示子命令调用信息。<br>m 通过 makefile 构造 make 程序的描述文件的时候,其他选项都失去功能,但该标记在生成 makefile 的时候会令其他所有标记有效。<br>v 显示 b 标记的信息,以及不需要命令执行的目标信息。 |
| --dry-run | 指明不执行任何命令。实际上如果不是 dry run,本选项会列出所有可执行的命令 |
| -e | 同--environment-overrides |
| --environment-overrides | 环境变量重写 makefile 内部定义的变量 |
| -f filename | 同--file |
| --file=filename | 使用指定文件作为 makefile,而不是查找文件 Makefile 或 makefile |
| -h | 显示选项列表 |
| --help | 显示选项列表 |
| -i | 同--ignore-errors |
| -I directory | 同--include-dir |
| --ignore-errors | 构造目标第一次出现错误的时候,正常处理停止,但该选项指出要继续进行下一个目标 |

（续表）

| 选项 | 描述 |
| --- | --- |
| --include-dir=directory | 查找指定目录，其中包含 make 程序的描述文件 |
| -j [number] | 同--jobs |
| --jobs[=number] | 指出可以同时运行的命令数目。如果没有指定该数字，make 尽可能多地运行任务 |
| --just-print | 同--dry-run |
| -k | 同--keep-going |
| --keep-going | 指出出现错误后，尽可能多地处理任务。所有基于出错目标的东西都不会被创建，但是一次依赖关系错不会禁止其他处理过程 |
| -l [number] | 同--max-load |
| --load-average[=number] | 同--max-load |
| --makefile=filename | 同--file |
| --max-load[=number] | 如果已经运行了至少一条命令，且系统负载均值大于指定值（浮点数），不会启动新的命令。如果没有指定该数字，不会设置负载上限 |
| -n | 同--dry-run |
| --new-file=filename | 同--assume-new |
| --no-builtin-rules | 去除内嵌规则及后缀定义，尽管自己仍然可能定义这些规则。默认的变量设置有效 |
| --no-builtin-variables | 去除内嵌变量，尽管自己仍然可以定义它们。该选项隐含--no-builtin-rules 选项 |
| --no-keep-going | 同--stop |
| --no-print-directory | 使--print-directory 设置无效 |
| -o filename | 同--assume-old |
| --old-file=filename | 同--assume-old |
| -p | 同--print-data-base |
| --print-data-base | 打印规则和变量的值。该信息组合了预定义的值和 makefile 的内容 |
| --print-directory | 打印消息，声明执行 makefile 前后的工作目录名。该选项只有在 make 程序的描述文件也调用的时候才有意义 |
| -q | 指出没有运行任何命令或生成任何其他格式的输出，除了返回状态代码。状态代码为 0 说明所有目标都是最新的，如果正常运行就不会编译任何文件。状态代码为 1 说明有一个或者更多的目标需要编译。状态代码为 2 说明出错 |
| -quiet | 同--silent |
| -r | 同--no-builtin-rules |

（续表）

| 选项 | 描述 |
|---|---|
| -R | 同--no-builtin-variables |
| --recon | 同--dry-run |
| -s | 同--silent |
| -S | 同--stop |
| --silent | 禁止正常打印每行执行的命令 |
| --stop | 取消-keep-going 选项的努力 |
| -t | 同--touch |
| --touch | 调整目标文件上设置的日期，将它们变为最新的日期，而不是实际执行命令创建最新版本的文件 |
| -v | 显示版本信息，然后退出 |
| --version | 显示版本信息，然后退出 |
| -w | 同--print-directory |
| -W filename | 同--assume-new |
| --warn-undefined-variable | 每次引用未定义变量都发出警告 |
| --what-if=filename | 同--assume-new |

其中有条命令，一个 make 进程可以激活另一个进程。发生这种情况时，运行的父 make 进程会将它的所有设置都传递给新激活的子进程。因为这种情况，你会找到保存为默认设置的选项（它们会被包含进来，作为激活 make 子进程命令的一部分）。选项恢复默认设置的另一个原因就是可由 MAKEFLAGS 环境变量修改默认设置。

## 14.2 Autoconf

Autoconf 是创建安装 shell 脚本的工具，该脚本会作为源代码的一部分发布。默认情况下，安装脚本为 configure。脚本 configure 独立运行，因此系统并非必需存在 Autoconf 才能配置并安装软件。

使用 Autoconf 打包并组织软件发布的好处很多。configure 脚本会检查是否存在一定的系统功能，会生成 makefile 反映当前环境，就是说应用程序可以立即移植到 UNIX 的所有版本上。使用 configure 脚本安装软件已经变得非常普通了，因为大多数人都已了解安装软件的过程。要安装由 Autoconf 打包的软件，过程通常如下：

```
$ ./configure
$ make
```

```
$ make install
```

Autoconf 实际上是工具集合，表 14-2 对其有所描述。

**表 14-2　Autoconf 系列工具**

| 工具 | 描述 |
|---|---|
| autoconf | 使用模板文件作为输入，该工具产生的配置脚本可生成 make 程序的描述文件，以及适合当前平台（或指定平台）的安装脚本 |
| autoheader | 该程序创建一个模板文件，当中包括 autoconf 创建的 configure 脚本使用的#include 语句 |
| autoreconf | 文件的时间戳表明需要通过运行 autoconf 更新文件时，该程序可以更新配置脚本 |
| autoscan | 本程序浏览目录树中的源文件，生成模板文件的初始版本，作为 autoconf 的输入文件 |
| autoupdate | 程序更新存在的模板文件，以匹配 autoconf 当前版本的语法 |
| ifnames | 程序浏览出现在#if、#elif、#ifdef 和#ifndef 预处理指示字中所有 C 的源文件。列表是排序的，每个名字都包括其中找到文件名的列表 |

根据应用程序的复杂程度，以及所需的可移植水平，创建安装脚本的过程可能非常简单，也可能相当复杂。任何情况下，以下的序列都可作为整个过程的指导。将目录变更到保存源文件的位置，执行如下步骤：

1. **确定条件编译**　通过在头文件中使用预处理指示字增加软件的可移植性不很常用。为了收集条件编译的信息，对所有需要预处理的源文件执行 ifnames 程序。例如，下面命令会处理所有 C 源文件和头文件：

   ```
   $ ifnames *.c *.h
   ```

   输出为条件定义的宏的列表，以及定义它们的文件。

2. **创建 configure.in 文件**　在源代码目录中，在命令行运行无参数的 autoscan，如下：

   ```
   $ autoscan
   ```

   这会生成文件 configure.scan，它是用于构造最终 configure 脚本的文件框架。复制（或移动）configure.scan 到 configure.in，这样会将适当的设置行加入其中。

3. **编辑 configure.in 文件**　这是任务的主要部分。该文件由 m4 宏指示字组成，会被 Autoconf 解析生成最后的 configure 脚本。如果安装过程比较复杂，宏已经不能处理了，该脚本也可以包含 shell 脚本片断，可以做直接复制到最终 configure 脚本中。原始的 configure.in 脚本包括很多最后版本需要的宏，也有一些描述性的注释（由 hash 符号开头）。修改文件信息时加入更多的注释是个好主意，表 14-3 包括的描述性信息可用于不同的宏。每个宏都包括一个逗号分隔的列表，格式如下：

   ```
   AC_CHECK_LIB(dl, dlopen, socket)
   ```

宏的名字和左括号之间不能有空格。可在方括号中加入参数，这是可选的，但如果参数长度大于一行，就必须使用方括号。

**表 14-3 configure.in 脚本使用的 m4 宏**

| 宏 | 描述 |
|---|---|
| AC_C_CHAR_UNSIGNED | 该宏检查默认的 char 数据类型，如果是无符号的，就定义宏__CHAR_UNSIGNED__ |
| AC_C_CONST | 该宏检查 C 编译程序处理 const 关键字的方法，如果必要，重新定义之 |
| AC_CHECK_FUNCS | 该宏检查是否存在指定函数，函数用空格分隔 |
| AC_CHECK_HEADERS | 该宏检查是否存在指定的一个或多个头文件，这些头文件由空格分隔 |
| AC_CHECK_LIB | 该宏检查是否存在指定的库。按照简写形式指定库名，函数作为库成员也要指明用于测试。例如，如果指明了 AC_CHECK_LIB(cfont,bdf)，那么 libcfont 库必需包含函数 bdf |
| AC_CONFIG_AUX_DIR | 该宏指出目录的名字包含 install-sh、config.sub 和 config.guess。默认目录通常正确，这个宏也可用于指定绝对路径和相对路径 |
| AC_CONFIG_HEADER | 该宏编辑了包括#define 指示字的头文件。创建的文件名跟着冒号，以及用于指示字的输入文件名。例如，创建 config.h 来自 AC_CONFIG_HEADER(config.h:config.in)中 config.in 的内容 |
| AC_CONFIG_SUBDIRS | 该宏指出目录列表，其中包括 configure 脚本，会由它的创建者在这里运行它。目录名字由空格分隔 |
| AC_FUNC_MEMCMP | 该宏验证了 memcmp()，正确操作于 8 位边界 |
| AC_FUNC_STRFTIME | 该宏检查 strftime()函数操作的正确性 |
| AC_FUNC_VPRINTF | 该宏检查是否存在 vprintf() |
| AC_HEADER_STDC | 该宏检查系统是否有标准 C 头文件 |
| AC_HEADER_SYS_WAIT | 该宏检查是否存在 POSIX-兼容的头文件 sys/wait.h |
| AC_HEADER_TIME | 该宏检查是否可将 time.h 和 sys/time.h 都包含进同一个编译单元 |
| AC_INIT | 该宏必需第一个出现。它包括惟一指定文件的名字，作为安全性检查，需要检查用户是在正确的目录运行该脚本——例如，AC_INIT(hello.c)。惟一需要的其他宏是 AC_OUTPUT |
| AC_OUTPUT | 该宏是必需的。它会命名并输出 makefile，以及其他一些输出文件。如果包含了其他参数，这些命令会被加到 config.status 中在其他命令之后运行。通常写成 AC_OUTPUT(Makefile)。惟一需要的其他宏为 AC_INIT |
| AC_OUTPUT_COMMANDS | 该宏指出额外的命令需要在 config.status 的结尾处运行。该宏可重复使用——例如，AC_OUTPUT_COMMANDS（响应的额外命令） |

（续表）

| 宏 | 描述 |
|---|---|
| AC_PREFIX_DEFAULT | 该宏设置安装前缀，而不是默认的/usr/local——例如，AC_PREFIX _DEFAULT(/home/fred/sets) |
| AC_PREFIX_PROGRAM | 如果用户没有使用--prefix 选项选择前缀，该宏会查找指定的程序，利用 PATH 变量，将前缀设置到包含程序的目录。 |
| AC_PREREQ | 该宏确保使用的 Autoconf 版本足够新。例如，该宏可确信使用的版本为 1.8 或更新版本：AC_PREREQ(1.8) |
| AC_PROG_MAKE_SET | 该宏预定义 MAKE 变量，就好像已经通过命令 MAKE=make 设置了环境变量 |
| AC_REVISION | 该宏将指定的修改信息复制到 configure 脚本中 |
| AC_TYPE_OFF_T | 该宏检查是否存在特定的 typedefs，如果不存在就定义它们 |
| AC_TYPE_SIZE_T | 该宏检查是否存在特定的 typedefs，如果不存在就定义它们 |

4. **创建 makefile.in**　为了利用 Autoconf 作出的配置决策，需要修改自己的 makefile（将它命名为 makefile.in）来包括 Autoconf 产生的定义。一些常见的定义参见表 14-4。

**表 14-4　Autoconf 定义的 makefile 关键字**

| 关键字 | 描述 |
|---|---|
| @CC@ | C 编译程序 |
| @CFLAGS@ | 传递给 C 编译程序的标记集合 |
| @CPP@ | C 预处理程序 |
| @CPPFLAGS@ | 传递给 C 预处理程序的标记集合 |
| @CXX@ | C++编译程序 |
| @CXXFLAGS@ | 传递给 C++编译程序的标记集合 |
| @DEFS@ | 如果使用了宏 AC_CONFIG_HEADER，通常定义为-DHAVE_CONFIG_H |
| @INSTALL@ | install 工具或 install-sh 脚本 |
| @LDFLAGS@ | 传递给连接程序的标记 |
| @LIBOBJS@ | 连接程序时包含的目标文件 |
| @LIBS@ | 连接程序时包含的库 |
| @RANLIB@ | ranlib 工具 |
| @SET_MAKE@ | 通常"MAKE=make" |
| @srcdir@ | 包含源文件的目录名 |

5. **创建 config.h.in** 创建头文件最简单的方法就是运行 autoheader，让它创建 config.h.in，该文件会被作为创建 config.h 的输入。上述操作可通过键入如下命令实现，不带参数：

```
$ autohead
```

6. **更新源文件** 所有需要考虑移植性的源文件，需要包含头文件 config.h。这样就可以根据安装环境进行条件编译。例如，如果标准 C 头文件不存在，需要改变处理过程：

```
#ifndef STDC_HEADERS
   /* Compiled only if there are no standard C headers */
#endif
```

7. **创建安装脚本** 工具 autoconf 会用如下命令读 configure.in 并产生 configure 文件：

```
$ autoconf
```

8. **复制 Autoconf 脚本** 如下三个脚本都应当包含进来，作为安装包的一部分。它们也是 Autoconf 安装的一部分，通常可在目录/usr/lib/autoconf 或/usr/share/automake 中找到它们：

```
config.guess
config.sub
install-sh
```

# 第 15 章

# GNU 汇编器

GNU 汇编器其实是汇编器的集合，因为每种平台都需要不同的汇编器。就是说，尽管汇编语言自己会改变，基本指令集对所有平台都是通用的，甚至一些操作码的记忆方法在各个平台上都是一样的。

设计 GNU 汇编器的主要目的是要将编译程序的输出结果汇总成目标代码格式，用于连接程序。由此可见，汇编器通常在幕后工作，被 GCC 自动激活，但也有可能出现某种情况需要直接操作汇编器。

## 15.1 用命令行进行汇编

在编写更高级别语言程序的时候，GCC 通常会激活汇编器，因此很少需要处理它的命令行选项。但如果决定编写汇编语言模块，就有可能为了某个特别的理由，需要使用一些命令行选项。这些选项列在表 15-1 中。

**表 15-1　GNU 汇编器命令行选项**

| 选项 | 描述 |
| --- | --- |
| -a[opts][=file] | 打开输出列表。以下一个字母或者多个字母的组合可用作选项指明输出列表的格式和内容。默认为-ahls。该列表默认为标准输出，但也可以重定向到选项中等号后面指定的文件。例如，-ahls=assembly.llist。<br>c 因为条件为假，忽略非汇编代码。<br>d 忽略所有源代码中的调试指示字。<br>h 包括更高级别语言的源代码。<br>l 包括十六进制格式的汇编代码。<br>L 包括调试统计。<br>m 包括宏扩展。 |

（续表）

| 选项 | 描述 |
| --- | --- |
| | n 忽略格式处理过程。 |
| | s 包括符号交叉引用表。 |
| --defsym symbol=value | 定义指定的符号，并为它赋值 |
| -f | 跳过空格和注释的预处理过程 |
| --fatal-warnings | 将警告当作错误 |
| --gdwarf2 | 生成 DWARF2 调试信息，并将它们包含在目标文件中 |
| --gstabs | 生成 STABS 调试信息，并将它们包含在目标文件中 |
| --help | 显示本选项列表并退出 |
| -I directory | 将指定目录加入列表，就是在此列表中查找.include 指示字的响应 |
| -J | 指出对于符号溢出不发警告 |
| -K | 对不同表格之间的变化发出警告。该表包含的绝对值，是两个可重定位的值的差，在位置调整的时候，这些值也会改变 |
| --keep-locals | 包含符号表条目，用于局部符号，它们的开头都为.L |
| -L | 同--keep-local |
| -M | 同--mri |
| -MD filename | 依赖关系信息，格式包含在 makefile 中，被写入指定的文件 |
| --mri | 编译为 MRI 兼容模式。就是说，汇编处理过程会假设其语法与 Microtec Research 的汇编程序语法相同 |
| --no-warn | 禁止所有警告消息。同--W |
| -o filename | 输出目标文件的名字 |
| -R | 将来自数据部分（section）的代码放置（fold）到文本部分（section） |
| --statistics | 显示空格总数，以及汇编程序的执行时间 |
| --strip-local-absolute | 删除所有局部汇编符号和常数值，只使用它的值 |
| --traditional-format | 指定使用的文件格式与本地汇编程序使用的输出文件格式相同 |
| --target-help | 显示目标指定的选项列表，然后退出 |
| --version | 显示版本信息，然后退出 |
| -W | 禁止所有的警告消息。同--no-warn |

如果发现需要编写汇编模块，最好的办法是用 C 编写一个简单的程序，其中包括所有需要的结构元素，然后使用 gcc 的-S 选项生成汇编语言的源代码。直接写汇编语言程序是错误的倾向，非常冗长乏味，因此最好还是从一种固定不变的基础开始。

如果不需要太多汇编语言，插入内嵌的汇编语言也许更简单，本章稍后有所描述。

## 15.2　绝对、相对以及边界

大多数的汇编语言代码不得不处理定位以及地址计算。无论何时使用了位置的名字，汇编器都会计算该位置的地址。例如，如下的 jle 语句（小于或等于时跳到某处）会进入分支地址.L3：

```
    addl    $16,%esp
    jle     .L3
    call    function
.L3:
    movl    $0,%eax
```

位置被标记为.L3 并不是纯粹的数字，因为连接程序将程序连接成可执行程序的时候，会改变它的位置。直到目前为止，这是一个相对值，因为它的值只能定义为相对于模块顶端的位移。连接程序会改动所有引用此相对地址的值，例如本例中 jle 语句的引用。

绝对表达式是一个常量值，连接程序不会改动它。它可为任意的数字常数值，或为可计算的表达式。可通过对相对地址的计算创建绝对值。例如，如下表达式是绝对值，因为它是两个位置之间距离的常数值。

```
.L6 - .L3
```

连接程序会重新定位.L6 和.L3，但它们之间的距离不会改变。但并非所有用到相对地址的表达式结果都是绝对值。例如，下面的表达式相对于.L44，因为它所做的就是计算出一个常数值并把它加到相对地址上。

```
.L44 + .L6 - .L3
```

有些对地址进行算术计算的表达式会是错误定义的。例如，以下的表达式结果为无意义的数字，是连接程序选择位置的函数：

```
.L6 + .L3
```

另一个重要概念为地址边界（address boundary）。如果地址为 16 的偶数倍，那么该地址就为 16 字节边界。对某些特定数据结构和指令这是很重要的。在有些情况下这只是效率问题，但有时由于硬件需求，这样做就是必须的了。汇编器指示字，例如.org 和.align，都可用于插入字节进行填充，强制该项目后面紧跟的就是一个边界。当然，对于正确的边界，连接程序有必要对齐模块的起始位置，这样它的内部边界仍然有效。

## 15.3 内嵌汇编

使用汇编语言编码的理由很多，但几乎没有什么理由要求编写一个完整的汇编语言程序（甚至是完整的模块）。要在机器级做一些操作，通常最好通过在较高级别语言代码中包含汇编语言的通道（passage）来实现。而 GCC 提供了直接将汇编语言命令插入 C 函数的功能。

从本质上讲，汇编语言没有什么可移植性问题。为某个平台写成的代码几乎对其他平台肯定是错的。但是，为所有平台编写代码的过程是一样的。本节描述了用与 Intel 系列处理器兼容的语法编写代码的过程。

### asm 构造器

如下的例子程序使用 asm 将汇编语言插入到 C 代码中。本例将一个 C 变量的值载入寄存器，向右移一位将它的值取半，并将结果保存在其他变量中：

```
/* half.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    int a = 40;
    int b;

    asm("movl %1,%%eax; \
         shr  %%eax; \
         movl %%eax,%0;"
           :"=r"(b)
           :"r"(a)
           :"%eax");

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

该构造器不仅仅是一种简单的插入汇编语言代码的技术——使用它，就可以用 C 语法来定位变量的地址，甚至可以指定向 C 代码生成和优化阶段传递的信息，因此可生成高效代码。如下是 asm 构造器的语法：

```
asm(assembly language template
    : output operands
    : input operands
    : list of clobbered registers);
```

如果想要防止编译程序优化汇编语言代码，可使用关键字 volatile，如下所示：

```
asm volatile ( ...
```

而且，如需适用 POSIX，可使用关键字__asm__和__volatile__，而不是 asm 和 volatile。

### 汇编语言模板

汇编语言模板包括一条或多条汇编语句，是被插入内嵌的实际代码。操作码可以定位常数的值，寄存器的内容，以及内存的位置。以下是定位值的语法规则汇总：

- 寄存器名字以两个百分号打头，例如%%eax 和 %%esi。Intel 的寄存器名字通常以一个百分号开头，而且 asm 模板也需要一个百分号，这就是为什么这里必须是两个百分号。
- 内存的位置是输入操作数或输出操作数的其中之一。每一个都是根据其声明的顺序来指定的，是一个跟着冒号的数字。第一个输出操作数为%0。如果有其他输出操作数，会为%1，以此类推。数字包括输入操作数——例如，如果有两个输出操作数，第一个输入操作数为%2。
- 内存的位置也可通过将地址保存在寄存器中然后用括号包围寄存器来定位。例如，下面的代码会通过将寄存器%%esi 的内容载入%%al 寄存器来定位：

```
movb    (%%esi),%al
```

- 即值（常量）设计为美元符号（$）跟着数字自己，例如$86 或 $0xF12A。
- 所有汇编语言都是单引号的串，每行汇编代码都需要终结子（terminator）。该终结子可为分号或换行符号（\n）。而且，可在汇编语言列表中插入制表符，以提高它们的可读性。

### 输入和输出操作数

汇编代码可以引用变量名字的列表，当作输入和输出操作数。可以使用所有有效的 C 表达式来定位内存。例如下面的代码对前面的例子作了些变动，用到数组保存输入和输出值，而且通过左移一位将它扩大一倍：

```
/* double.c */
#include <stdio.h>
int main(int argc,char *argv[])
{
    int array[2];
    array[0] = 150;
    int i = 0;

    asm("movl %1,%%eax; \
```

```
        shl  %%eax; \
        movl %%eax,%0;"
          :"=r"(array[i+1])
          :"r"(array[i])
          :"%eax");

    printf("array[0]=%d  array[1]=%d\n",array[0],array[1]);
    return(0);
}
```

用于指定输入和输出变量的规则如下：

- C 表达式的结果为程序中的一个地址，由括号包围。
- 如果地址前面跟着"r"，这意味该值必须保存在寄存器中。会在执行汇编语言值前载入输入变量，而输出变量会在代码执行以后保存在内存中。"=r"格式应该用于输出操作数。
- 可通过指出寄存器为以下之一来约束变量：

```
"a"   %%eax
"b"   %%ebx
"c"   %%ecx
"d"   %%edx
"S"   %%esi
"D"   %%edi
```

- 变量可被约束定位在内存中，而不是通过"m"约束载入寄存器。
- 在同样的变量既是输入值又是输出值的情况下，用"=a"当作它的输出约束，它的引用数字被当作它的输入约束。如下例子使用 counter 作为输入和输出：

```
asm("incw %0;"
    : "=a"(counter)
    : "0"(counter));
```

- 用逗号分隔输入和输出操作数，可使用任意多的输入和输出操作数。
- 输出和输入操作数都是数字化的，按顺序的，相当于以$0 开始依次到$n-1，这里的 n 是输出和输入操作数的总数。例如，如果总共有 6 个操作数，最后一个为$5。

**被破坏的寄存器列表**

该列表中的寄存器是被代码破坏（clobbered）的，而这个简单列表中的寄存器名字由逗号分隔，如下：

```
. . .
"%eax", "%esi");
```

此信息会传递给编译程序，这样编译程序就会了解到这些寄存器不希望保存任何值。

## 15.4　汇编器指示字

汇编语言的主要目的就是要将便于记忆的操作码（mnemonic opcode）翻译成二进制的操作码（opcodes），后者可被硬件执行或被用作数据存储位置。另外汇编器理解并能操作汇编器指示字，用来对齐代码、定义扩展宏、将代码分成指定的章节、声明指定常数、提供条件编译，或只是一种定义字符数据的简便方法。

下面是 GNU 汇编器的汇编器指示字列表。每种情况下指示字都以句号开头。一些指示字是独立的，有些参数出现在同一行，有些可以长达几行，直到遇到下一个指示字才算结束。有些指示字——尤其是那些用于插入调试信息的指示字——只对一两种目标文件格式有效。

汇编器还可识别一些指示字，它们既没有什么副作用也没有什么功能。例如，.abort 指示字可以忽略汇编处理过程，但它的作用可能会很快消失。汇编器可识别但实际没有什么作用的指示字包括 .file、.app-file、.extern、.ident 和.lflags。

某些指示字可能会有一些奇怪的行为。这是历史原因造成的。因为编译程序的历史已经足够长了，而且目前仍然使用着很多非常早的设计决策。随着时间的流逝和硬件的改动，某些满足硬件特性的汇编器指示字保持了最初的样子。GNU 汇编器被编写成可与其他宿主平台汇编器相兼容，因此它与现存指示字的行为都必须相适应。这样解释，并不是说那些指示字已经没用了，只是说有些指示字的操作方法非常奇怪。为举例说明，请参考下面列表中的.fill 指示字。

很多指示字都能使用或定义符号。符号是个名字，具有属性 value 和 type。值既可为绝对数字，也可为相对数字，类型说明了数据的尺寸以及如何解释该数据。

-align boundary [,filler] [,maximum]

在当前位置插入 filler，将地址对齐到指定的边界。所有三个值都是绝对表达式。如果没有指定 filler 值，填充器（filler）对数据部分的默认值是零，对执行部分的默认值为 noop。如果指定了 maximum 值，它会利用更多的字节来填充，直至达到边界后不采取动作。

filler 和 maximum 都是可选的。利用两个逗号，可以指明 maximum 而不指明 filler。

该指示字的确切含义并不一致，这是因为 GNU 汇编器会在每个系统中模拟本地的汇编器。例如某些系统通过.align 8 格式定位多段代码，指明使用 8 字节边界进行对齐。而其他 3 位边界的系统是通过.align 3 来说明，结束地址值需要的最小值为零。为了语法一致，可以使用.balign 或.p2align。

.ascii [string][,string ...]

将零或多个引用字符串汇编成 ASCII 字符数据。该字符串不需在结尾加零。

.asciz [string][,string ...]

将零或多个引用字符串汇编成 ASCII 字符数据。每个字符串都要在结尾处加零。

.balign boundary [,filler] [,maximum]

将 filler 字节的数据插入当前位置，以便和指定边界对齐。三个值都是绝对表达式。如果没有指定 filler 值，它对数据部分的默认值是零，而对可执行部分的默认值为 noop。如果指定了 maximum 值，它会利用更多的字节来填充，直至达到边界后不采取动作。

filler 和 maximum 都是可选的。利用两个逗号，可以指明 maximum 而不指明 filler。

.balignl boundary [,filler] [,maximum]

同.balign，只是 filler 是 32 位（long）值。

.balignw boundary [,filler] [,maximum]

同.balign，只是 filler 是 16 位值。

.byte expression [,expression ...]

每个表达式分配一个字节，将表达式的值插入分配的字节。

.comm symbol, length

声明一块 length 字节的未初始化的内存位置，并标记为名字 symbol。如果多个模块定义了相同的符号，就将它们合并在一起。如果声明的符号尺寸不一样，就用最大的那个。

ELF 系统有三个可选参数来指定对齐代码。HPPA 的指示字语法为 symbol .comm，length。

.data subsection

跟着.data 指示字的语句会被汇总到数字化的 subsection 子部分，该数字为绝对表达式。默认的 subsection 数字为零。

.def name

插入 COFF 格式目标中的内容，以一块调试信息开始，标记为符号 name。这个块会继续下去，直到指示字.endef 终止它。参见.dim、.scl、.tag、.type、.val 和.size。

.desc symbol, value

符号被定义为指定值。value 必须为绝对表达式。指示字不会产生 COFF 格式的输出。

.dim

这个指示字只能用在.def 和 .endef 之间。编译程序用它向符号表添加额外信息。只对 COFF 目标格式有效。

.double value [,value ...]

对每个指定的值，浮点数字都被汇总起来保存在内存中。浮点数字的内部表示，包括尺寸和范围，依据平台而有所变化。参见.float。

.eject

在汇编器输出列表中插入页分隔符。

.else

见.if。

.endef

见.def。

.endif

见.if。

.equ symbol, value

该指示字定义的符号有值。value 既可为绝对表达式，也可为相对表达式。同一符号可多次使用.equ 指示字，每次都改变其值。HPPA 的指示字语法是 symbol .equ value。该指示字与.set 相同。参见.equiv。

.equiv symbol, value

同.equ 或.set，只是它会对以前定义的符号生成出错消息。

.err

只要不指明–Z 选项，指示字.err 就会生成错误，防止产生目标文件。它利用内部的条件汇编代码来指定错误，如下例所示，如果没有声明符号 BLACKLINE ，就会出错：

```
.ifndef BLACKLINE
.err
.endif
```

.fill repeat, size, value

该指示字创建多个数据块，每个最多都是 8 字节。repeat 的值为绝对表达式，指出创建的块数。size 可为任意绝对值，但大于 8 的值会被当作 8 来处理，这是每个块的字节数。

用于填充每块的值来自一个 8 字节数组。最高顺序的 4 个字节总是零。最低 4 个字节来自 value，被看作 32 位二进制整数，字节的顺序依赖于本地机。每块都有必要在低端填充一些字节，这些字节来自结果数组的低端。

如果没有指定 size，默认值为 1。如果没有指定 value ，默认值为 0。参见.org 和.p2align。

.float value [,value ...]

对每个指定的值，浮点数字都被汇总起来保存在内存中。浮点数字的内部表示，包括尺寸和范围，依据平台而变化。参见.double。

.global symbol

命名的符号必须在哪里已经定义过了，它必须是全局的，这样该符号才能被连接程序

.ifndef symbol

只有未定义该符号才能进行条件汇编。见.if。

.ifnotdef symbol

只有未定义该符号才能进行条件汇编。见.if。

.include “filename”

将指定文件插入该文件，并将其汇编到指示字所在位置。-I 命令行选项可用于指定指示字，用于查找文件。

.int value [,value ... ]

对每个指定值，都会有一个整数被汇总到内存中。整数的尺寸和字节顺序依赖于平台。参见.long、.int、.short 和.word。

.irp tag,str[,str ...]

.irp 和.endr 指示字之间的代码为每个列出的值都进行一次汇编，值会被插入到反斜线打头的标记位置。例如，下面指出三个寄存器：

```
.irp    tag,esp,ebp,eax
subl    $1,%\tag
.endr
```

对每个串都会汇编一次代码，如下：

```
subl    $1,%esp
subl    $1,%ebp
subl    $1,%eax
```

参见.macro、.rept 和.irpc。

.irpc tag, charlist

.irpc 和.endr 指示字之间的代码为 charlist 中的每个字符都会汇编一次，字符会被插入到反斜线打头的标记位置。例如，下面例子将一行代码扩展为三行：

```
.irpc   tag,123
addl    $\tag,%esp
.endr
```

对串中的每个字符都会汇编一次代码，如下：

```
addl    $1,%esp
addl    $2,%esp
addl    $3,%esp
```

参见.macro、.rept 和.irp。

.lcom symbol, length

保留 length 指定的字节数，为绝对表达式，作为 bss 部分的一个局部数据块（在载入程序的时候，将块初始化为零）。符号是局部的，因此连接程序看不到。

HPPA 的语法为 symbol.lcomm,length。

.line number

将当前行号改为由绝对表达式 number 指出的行。有些系统必须使用.ln，意义相同。

.linkonce [type]

标记当前部分，因此连接程序只包含一次，即使同样的部分出现在多个模块，也只包含一次。指示字必须在每部分实例中都出现一次。因为通过名字进行选择，所以名字必须惟一。

可选的 type 参数可被去掉（discard），这样不需提示就可除去重复部分（这是默认的）。one_only 类型对找到的每个重复内容都发出警告。same_size 类型会在重复部分尺寸不一样的时候发出警告。same_contents 类型会在重复部分任意值不相同的时候发出警告。

.list

该指示字为输出列表加一。如果数字大于零，汇编器会产生列表到标准输出中。.nolist 指示字可用于为数字减一。数字默认为零，也可用命令行的-a 选项设置其他值。

.ln number

与.line 意思相同。

.long expression

该指示字与.int 意思相同。

.macro name [tag[=value]] [,tag[=value]]

递归的宏处理器，可用于为代码块分配名字，利用可选参数，可被扩展并汇编进其他位置。例如，下面的宏 saveregs 无论用于何处都会被扩展为一对语句 pushl：

```
.macro saveregs
pushl  %ebp
pushl  %eax
.endm
```

要扩展宏，无论在何处使用操作码，就可以用它的名字，像下面这样：

```
main:
    saveregs
    movl    %esp,%ebp
```

指示字.macro 可递归地使用，并接受参数。如下宏可用于声明块，其中包含所选类型的变量数目：

```
.macro block type=.int count=1
.if \count
\type 0
block \type,\count-1
.endif
.endm
```

如果宏没有参数，声明会包括一个.int。下面语句会产生 5 个.long 数据类型的声明：

```
block .long 5
```

可用.exitm 在任何位置修改扩展宏。例如，下面的语句在 trigger 值为 12 时会放弃扩展宏：

```
.if trigger-12
.exitm
.endif
```

参见.rept、.irp 和.irpc。

.mri expression

如果表达式非零，汇编就会转换到 MRI 模式。这与在命令行中使用-M 或--mri 一样。该模式的作用域会持续到文件结尾，或直到出现表达式的值为零的 mri 指示字为止。

.nolist

见.list。

.octa bignum[,bignum ...]

对于列表中的每个 bignum，会为它们分别声明一个 16 字节的数字，并将声明的值保存其中。这可被看作 8 个 16 位的值，因此名为.octa。参见.quad。

.org address[,filler]

本部分的当前地址可向前调整到 address 指定的位置。address 值相对于当前部分的顶端。基于当前部分的地址，该地址可为绝对表达式，也可为相对表达式。指示字只可向前移动地址，而不能向后移动。如果存在插入的字节，它们都被初始化为 filler 的值。默认的 filler 值为零。

也见.fill、.skip 和.p2align。

.p2align zeroes[,filler][,maximum]

如果有必要会增加当前地址，直到它低端的零位到达指定数目。例如，zeroes 值为 3

会将位置计数器扩大直到地址的低端加入了足够的零位，结果是地址为 8 字节边界。绝对值 filler 是字节值，保存在新空间中。如果没有指明 filler，对数据部分默认为零，对代码部分默认为 noop 指令。maximum 值是地址增加的最大字节数目。

也见.org、.fill、.skip、.p2alignl 和.p2alignw。

.p2alignl zeroes[,filler][,maximum]

同.p2align，只是 filler 用的是 16 位值。

.p2alignw zeroes[,filler][,maximum]

同.p2align，只是 filler 用的是 32 位值。

.psize lines[,columns]

指出每页的行数，可选地也可指出输出的列数。默认为 60 行 200 列。如果指定行数为零，不会插入换页符。

.quad bignum[,bignum ...]

每个 bignum 值都声明为 8 字节值。见.octa。

.rept count

按照指定数目，多次重复.rept 和.endr 间的代码。例如，以下序列会声明 14 次.int 值，每个都初始化为 10：

```
.rept 14
.int 10
.endr
```

也见.macro、.irp 和.irpc。

.sbttl “subtitle”

对列出的每页都使用指定的子标题。

.scl class

该指示字可用在.def 和.endef 之间来指出符号的存储类。

.section name

.section 指示字的这种格式对所有目标格式都有效，只要该目标格式支持 section 任意命名。它将如下代码汇编成指定名字的 section。

.section name[,“flags”]

.section 指示字的这种格式对 COFF 目标格式有效。每个标志都是"flags"串中的一个单独字符，如下：

- b 包括未初始化数据的部分（bss 部分）。
- n 程序运行时不载入该部分。
- w 程序运行时，该部分可被写。
- d 数据部分，与可执行部分相对。
- r 该部分只读。
- x 这是可执行部分，与数据部分相对。

如果没有指明标志，默认的设置依赖于 section 的名字。如果名字未被预定义为某种含义，默认部分被载入且可写。

.section name[,“flags”[,type]]

.section 指示字的这种格式对 ELF 目标格式有效。每个标志都是"flags"串中的一个单独字符，如下：

- a 该部分是可分配的。
- w 该部分是可写的。
- x 该部分是可执行的。

如果指定类型，可为以下之一：

- @progbits　该部分包含数据。
- @nobits　该部分不包含数据（它是空白）。

如果没有指定标志，默认设置依赖于 section 的名字。如果名字未被预定义为某种含义，默认为不可分配的、不可写的或不可执行的，而且该部分会包含数据。

.section “name”[,flag ...]

.section 指示字的这种格式对 Solaris 汇编器生成的 ELF 目标格式有效。可选的标志列表可为以下的一个或多个：

- #alloc　该部分是可分配的。
- #write　该部分是可写的。
- #execinstr　该部分包含可执行的指令。

.set symbol, value

该指示字为符号定义值。这里 value 可以是绝对表达式，也可为相对表达式。同一个符号可以多次使用.set 指示字，每此都会改变其值。HPPA 中的指示字语法是 symbol .set value。该指示字和.equ 相同。也见.equiv。

.short value[,value...]

依赖于不同平台，与.hword 或.word 的意义可能相同。也见.int。

.single value[,value]

它和.float 同义。

.size

该指示字只能用在.def 和.endef 之间。编译程序用它为符号表包含额外的信息。该指示字只对 COFF 目标格式有效。

.sleb128 value[,value]

该指示字是和“带符号的 128 进制的小 endian 模式——signed little endian base-128”同义。这是压缩的变长数据表示形式，用于 DWARF 符号调试。见.uleb128。

.skip size[,filler]

该指示字创建的块由 size 字节组成，其中包括 filler 值。默认的 filler 值为零。见.fill、.org 和.p2align。

.stabd type,other,description

见第 13 章中 STABS 的描述。

.stabs “name:symdesc=typeinfo”,type,other,description,value

见第 13 章中 STABS 的描述。

.stabn type,other,description,value

见第 13 章中 STABS 的描述。

.string “characters”,[“characters”]

字符串（串）保存在内存中。每个串都在末尾加上了空字节（值为零）作为终结符号。C 语言定义的反斜线转义字符可用于本串。

.symver name,name2@nodename

对于 ELF 目标格式，该指示字将符号绑定到指定版本节点上，在代码和共享库汇总起来的时候会用到它。该指示字创建符号 name2@nodename 作为 name 的别称，它也被定义在源文件中。别名中的 name2 部分是被解析的真实外部引用名。nodename 部分是命令行中用于连接的节点名。

.tag structname

该指示字只能用于.def 和.endef 之间。编译程序用它包含全部的符号表调试信息。该指示字只对 COFF 目标格式有效。

.text [subsection]

该指示字之后跟着的语句会附加到文本子部分 subsection 的尾部，它是绝对表达式。如果没有指定 subsection，会假设它为零。

.title “heading”

指出的串为标题，用在列表页的顶端，后面紧跟源文件的名字和页号。

.type value

该指示字只能用在.def 和.endef 之间。其值 int 可用作符号表条目的类型值。该指示字只对 COFF 目标格式有效。

.val address

该指示字只能用在.def 和.endef 之间。其值是分配给符号表条目的地址。该指示字只对 COFF 目标格式有效。

.uleb128 value[,value]

该指示字是和“不带符号的 128 进制的小 endian 模式——unsigned little endian base-128”同义。这是压缩的变长数据表示形式，用于 DWARF 符号调试。见.sleb128。

.word value[,value]

该指示字声明一个数字值，尺寸和字节顺序依赖于平台。根据不同平台，它可能分别与.hword 或.short 意义相同。

# 第 16 章

# 交叉编译及窗体端口

默认安装的 GCC 编译系统所产生的代码适用于本机，即运行 GCC 的机器，但也可将 GCC 安装成能够生成其他的机器代码。安装一些必须的模块，就可产生多种目标机器代码，而且可通过命令行选择一种希望使用的代码。

## 16.1 目标机

从网站 http://gcc.gnu.org/install/specific.html 可以得到有可能的最新目标机列表。在此站点中可找到更新过的目标机列表，以及向各种目标机进行移植的最新信息。关于每种可能的目标机都有一个简短介绍，可以查找说明得到有关移植的一些特殊需求。已知目标的列表非常长，而且也总在加入一些新端口。如下是写作本书时的端口列表：

```
#s390-*-linux*                    m6811-elf
#s390x-*-linux*                   m6812-elf
*-*-freebsd*                      m68k-att-sysv
*-*-linux-gnu                     m68k-crds-unos
*-*-solaris2*                     m68k-hp-hpux
*-*-sysv*                         m68k-ncr-*
*-ibm-aix*                        m68k-sun
*-lynx-lynxos                     m68k-sun-sunos4.1.1
alpha*-*-*                        Microsoft Windows
alpha*-dec-osf*                   mips-*-*
alphaev5-cray-unicosmk*           mips-sgi-irix5
arc-*-elf                         mips-sgi-irix6
arm*-*-linux-gnu                  Older systems
arm-*-aout                        OS/2
arm-*-elf                         powerpc*-*-*powerpc-*-sysv4
avr                               powerpc-*-darwin*
c4x                               powerpc-*-eabi
DOS                               powerpc-*-eabiaix
dsp16xx                           powerpc-*-eabisim
```

```
ELF (SVR4, Solaris 2, etc)
h8300-hms
hppa*-hp-hpux*
hppa*-hp-hpux10
hppa*-hp-hpux11
hppa*-hp-hpux9
i370-*-*
i?86-*-esix
i?86-*-linux*
i?86-*-linux*aout
i?86-*-sco
i?86-*-sco3.2v4
i?86-*-sco3.2v5*
i?86-*-udk
ia64-*-linux
m32r-*-elf
m68000-hp-bsd
powerpc-*-elf powerpc-*-sysv4
powerpc-*-linux-gnu*
powerpc-*-netbsd*
powerpcle-*-eabi
powerpcle-*-eabisim
powerpcle-*-elf powerpcle-*-sysv4
powerpcle-*-winnt powerpcle-*-pe
sparc-*-linux*
sparc-sun-solaris2*
sparc-sun-solaris2.7
sparc-sun-sunos4*
sparc-unknown-linux-gnulibc1
sparc64-*-*
sparcv9-*-solaris2*
vax-dec-ultrix
xtensa-*-elf
xtensa-*-linux*
```

## 16.2　创建交叉编译程序

GCC 文件命名约定使得可在一台机器上编译并安装多个交叉编译程序。为了能够在其他机器上运行并安装程序，需要一些基本工具（汇编器、连接程序等），它们可接受目标机格式的目标代码并产生目标机器上的可执行代码。而且，需要将目标机器需要的所有必要的库安装到自己的本地机上。

如下的步骤可作为建立交叉编译程序环境的基本指导，但要知道这并非一成不变，遇到某些特殊情况需要特殊处理。

在开始之前，应该回顾第 2 章描述的 binutils 脚本和 configure 脚本。而且需要访问 GCC 站点查询有关指定端口的所有信息。订阅第 1 章提到的相关邮件组是个好主意，这样就可以和其他同行交流信息——这里总有很多关于在不同平台上编译 GCC 遇到的问题的讨论。

除非有充分理由需要使用最前沿的技术，否则最好还是使用已发布的较为稳定的编译程序源码，而不是最新的 CVS 快照。某些机器上可用最新版本的编译程序作为本地编译程序，但没必要处理不是系统必需的其他内容。

### 16.2.1　安装本地编译程序

在交叉编译程序产生可安装到目标机器的目标代码的机器上，编译程序自身以及它支持的程序，例如汇编器和连接程序，实际是需要在本机上进行编译并运行的。因此也需要本地编译程序，如果它还不存在，第一个任务就是安装 GCC 的本地版本以及 binutils 的本地版本。

是有可能建立非 GCC 的交叉编译程序的，但那样可能会接触某些令人困惑的问题。再次注意，没有必要处理不是系统必需的其他内容。

### 16.2.2 为目标机器建立 binutils

必须为目标机器编译第 2 章描述的 binutils 。正因为 GCC 使用的命名约定，使得在为其他机器编译和安装 binutils 时不会产生冲突。编译过程中所依赖的源文件集合与创建本地 binutils 时依赖的文件集合一样。

下面例子中的 binutils 源代码位于当前目录 src 的子目录中。在一个简单脚本中，以下 4 条命令会创建新目录 sun，并配置它在本地机上编译源代码，产生的输出结果是 sparc-sun-solaris2.7：

```
DIR=`pwd`
mkdir $DIR/sun
cd $DIR/sun
$DIR/src/configure --prefix=/usr/local \
      --target=sparc-sun-solaris2.7
```

配置完成之后，换到新的 sun 目录运行 make 可编译 binutils ，如下：

```
$ cd sun
$ make
```

最后一步是将访问许可设置为超级用户，使用下面命令安装新程序：

```
$ make install
```

该命令创建新的文件集合，位于/usr/local/bin，如下：

```
$ ls /usr/local/bin/sparc-sun-solaris2.7-*
sparc-sun-solaris2.7-addr2line      sparc-sun-solaris2.7-objdump
sparc-sun-solaris2.7-ar             sparc-sun-solaris2.7-ranlib
sparc-sun-solaris2.7-as             sparc-sun-solaris2.7-readelf
sparc-sun-solaris2.7-c++filt        sparc-sun-solaris2.7-size
sparc-sun-solaris2.7-ld             sparc-sun-solaris2.7-strings
sparc-sun-solaris2.7-nm             sparc-sun-solaris2.7-strip
sparc-sun-solaris2.7-objcopy
```

你也应该会有一个新的目录，其中有两个子目录，如下：

```
$ ls /usr/local/sparc-sun-solaris2.7
bin   lib

$ ls /usr/local/sparc-sun-solaris2.7/bin
ar   as   ld   nm   ranlib   strip
```

### 16.2.3　安装目标机器的文件

要为目标机器编译源代码，必须存在系统头文件，且已为本机设置好了。而且，编译并连接在目标机器上运行的程序时，有必要将它们连接到该机器的库上。这些工作依赖于交叉编译程序的目的。如果希望得到一个通用编译程序，可为目标机器编译完整的应用程序，就会需要所有头文件和库。在另一种极端情况下，如果是为不使用标准库和头文件的嵌入式系统创建交叉编译程序，则不需要复制任何文件。

我们需要复制目标机器的目录/lib 和/usr/lib 中保存的一些库文件。需要将这些新文件保存到文件结构中，用之前安装的 binutils 开头。所有用到的库都应该从目标机器复制到本地目录/usr/local/sparc-sun-solaris2.7/lib 中。具体需要哪个库集合依赖于目标机器以及希望编写的程序类型。

除了库，还需要连接到可执行程序的目标模块——模块的名字类似于 crt0.o 和 crtn.o——可将它们也复制到库所在的目录。

目标机器的头文件应该复制到/usr/local/sparc-sun-solaris2.7/include。在构建交叉编译程序之前复制头文件很重要，因为构建过程会在构造 libgcc.a 的时候使用它们。

### 16.2.4　可配置库 libgcc1.a

如果目标机器也有 GCC，可以从它上面复制 libgcc1.a，不需要自己构造它们。如果目标机器没有 libgcc1.a，就有必要构造它。

库 libgcc1.a 包括一些必要的基本函数，可将浮点运算映射到没有浮点运算硬件的系统上。如果没必要模拟浮点运算，用空 libgcc1.a 就足够令编译程序生成所有代码了。

有些嵌入式系统需要 libgcc1.a 的浮点计算基础函数。

如果目标系统有 C 编译程序但没有 GCC，则或者需要安装 GCC 来产生库，或者可以使用本地C编译程序构造库。如果要这么做，就需要像正常情况那样在目标机器上安装GCC源代码树，创建构造目录，执行 configure 脚本，指出目标机和宿主机，这里的宿主机就是交叉编译程序的宿主机。然后构造库，如下例所示：

```
$ ./configure --host=host --target=target
$ make libgcc1.a
```

产生的结果库会被包含在目标机器的其他库中。

### 16.2.5　构造交叉编译程序

如果基本工作已经就绪，剩下的就是构造这个新的编译程序了。下面的脚本假设 GCC 的源代码在子目录 gcc 中。它创建的新目录 sun 包含相关配置，可用于编译。就像前面构造 binutils 一样，执行 configure 脚本，指出目录前缀为/usr/local 以及目标机器为 sparc-sun-solaris2.7：

```
DIR=`pwd`
mkdir $DIR/sun
cd $DIR/sun
$DIR/src/configure --prefix=/usr/local \
      --target=sparc-sun-solaris2.7
```

一旦完成配置过程，就转变到新目录来编译交叉编译程序，如下：

```
$ cd sun
$ make
```

这是 GCC 完整的编译过程，因此执行起来需要一段时间。如果所有事情都正确，就不会发出错误消息。如果 libgcc1.a 库不正确，或者没有找到该库，在遇到第一个模块的时候编译过程就会失败，会发现缺少一个头文件。

如果编译程序的构造没有出错，键入如下命令，按照超级用户权限执行它，就会安装该编译程序并准备好运行了：

```
$ make install
```

### 16.2.6 运行交叉编译程序

可通过命令行使用 gcc 命令和-b 选项运行该交叉编译程序。例如，用本章构造的编译程序编译 helloworld.c ，键入如下命令：

```
$ gcc -b sun-sparc-solaris2.7 helloworld.c -o helloworld
```

假设当前的 gcc 为版本 3.2，该命令会执行编译程序 sun-sparc-solaris2.7-gcc-3.2。如果因为某些原因将当前编译程序升级为新版本，但仍然希望使用交叉编译程序的 3.2 版本，可按照如下方法指明版本号：

```
$ gcc -b sun-sparc-solaris2.7 -V 3.2 helloworld.c -o helloworld
```

利用-V 选项，可从安装的编译程序中选择版本号。编译程序的不同版本通常保存在目录中，目录命名方法如下：

```
/usr/local/lib/gcc-lib/machine/version
```

## 16.3 MinGW

Windows 操作系统可编译两种不同的程序。其中较简单的一种——不会使用窗体接口——称为操作台（console）程序。Windows 的操作台程序可按照常规方式使用命令行运行，可接受命令行中的参数。C 语言的操作台程序开始执行的函数为 main() ，使用的是标准输入、标准输出，并且按照普通方式发出标准的出错信息。

Windows 操作台程序可使用 MinGW（GNU 对 Windows 的最小配置——Minimalist GNU for Windows）进行编译，可从网站 http://www.mingw.org 下载页面。MinGW 是一些包的集合，但可将它们作为一个安装程序下载，名字格式可为如下任何一种：

```
MinGW-<version>[-<stamp>].tar.gz
MinGW-<version>[-<stamp>].zip
```

<version>为版本号，例如 1.0 或 1.1。可选的<stamp>是日期，格式为 YYYYMMDD，各种各样的包绑定在一起成为下载文件。

要安装 MinGW，将文件下载到工作目录，创建自己希望安装的目录，例如 c:\mingw。将下载的文件内容展开到这个新目录中。文档包括目录，因此要确保解压缩工具能够维持下载文件中的目录结构——这可能需要特殊的命令行选项。

剩下要做的所有事情就是将新的 bin 目录加入 PATH 环境变量。具体做法根据 Windows 的不同版本而有所区别，但下面的命令对大多数系统有效：

```
PATH=%PATH%;c:\mingw\bin
```

可通过键入如下命令显示版本信息来快速测试安装结果：

```
gcc -v
```

MinGW 的 gcc 和 g++程序的命令行格式与 gcc 和 g++的 UNIX 版本基本相同。

## 16.4 Cygwin

Cygwin 是种可安装在 Windows 系统中的 UNIX 环境。环境包括 binutils 包端口和动态连接库 cygwin1.dll，它是 UNIX API 的应用实现。Cygwin 的安装过程非常简单，可总结为以下几步：

1. 创建包含下载文件的工作目录。这是个临时目录，不会用于最后的安装过程。
2. 使用 Web 浏览器访问站点 http://cygwin.com。选择右边的按钮“Install Cygwin now.”。这会将 setup.exe 文件下载到工作目录中。
3. 从命令行中或从系统目录中的 Run 选项中执行程序 setup.exe。由此会引导你一步步地下载并安装 Cygwin 系统。

### 16.4.1 编译简单的 Cygwin 操作台程序

编译和连接程序的命令和通常使用 GCC 的命令非常相似，但有些文件的命名规则有所变化。所有可执行文件的后缀都为.exe，而所有共享库的后缀为.dll。下面的命令会将简单的 helloworld.c 程序编译成可执行代码：

```
C:\> gcc helloworld.c -o helloworld.exe
```

### 16.4.2 编译 Cygwin GUI 程序

Windows 应用程序的源代码几乎不需要修改就可进行编译，但有几点例外。

有必要去掉__export 属性。大多数情况只要去掉属性__export 就可以了，但也有可能要将具有__export 属性的函数声明替换为如下新的形式：

```
int fn(int) __attribute__ ((__dllexport__));
int fn(int) ...
```

下面条件编译的代码可被包含在源代码中：

```
#ifdef __CYGWIN__
WinMainCRTStartup() { mainCRTStartup(); }
#endif
```

如果不包含前面的代码，就有必要在命令行中指出连接选项-e _mainCRTStartup。

下面引用了一段 makefile，可将 Windows 程序编译成 GUI 可执行代码：

```
hellowin.exe: hellowin.o hellowin.res
   gcc -mwindows hellowin.o hellowin.res -o hellowin.exe

hellowin.res: hellowin.rc resource.h
   windres hellowin.rc -O coff hellowin.res
```

工具 windres 可将文件 hellowin.rc 编译成 COFF 格式，包括图标、位图及程序需要的其他资源。如果没有选项-O coff，结果的.res 文件就为 Windows 格式，不能使用 GCC 进行连接。

# 第 17 章 GCC 嵌入式系统

编译嵌入式系统上安装的软件的过程基本上和在其他系统上运行软件的交叉编译过程一样。就是说程序在一种机器上编译，在其他机器上运行。最根本的区别就在于目标操作系统是为特殊目的而设计的，自己没有软件开发能力。

通常嵌入式系统会限制可以使用的内存数量，一般它的环境也比桌面系统要严格。这就意味着嵌入式软件编译程序不仅需要能够编译出可在嵌入式系统的 CPU 上运行的代码，而且要求可执行代码要尽可能地小，并且效率也要尽可能地高。

## 17.1 建立编译程序和连接程序

可在第 16 章中找到 GCC 为嵌入式系统产生代码的基本准备过程。GCC 尤其适合这种配置，因为它可在任何一种平台上编译并安装，并可生成在其他平台上运行的代码。其中一个任务就是要为用到库及可连接目标模块的目标 CPU 和目标操作系统之间建立交叉编译程序。另一个任务就是下载并安装 binutils（它包括汇编器和连接程序），这样就可为目标机器产生目标模块。

一旦安装了交叉编译程序和连接程序，就可从 GNU 语言中选择一种语言（如果愿意，也可将它们混合起来），这样就可使用 GCC 的所有优化功能。而且某些情况下需要与硬件有更紧密接触的时候，也可使用内嵌的汇编语言。

利用 GCC 所有可用的选项，可以调整目标代码的内容，使之满足系统的需求。通过建立 makefile 可以调整每个模块的选项，而且有能力优化结果。尤其要注意命令行选项包含的目标文件信息会通过连接处理过程传递给最终的可执行程序。有些信息——尤其是调试信息——有可能与目标系统的目标文件格式不兼容。

对于嵌入式系统，你会发现需要连接到一个特殊的启动模块，这样才能满足操作系统的要求。通常这是一个很小的汇编语言块，可以汇总并直接连接到程序中。有些基本的初

始化例程非常大，有些又很小。一般的初始化序列非常标准，接着的处理过程包括下面的所有或某些步骤：

- 禁止所有硬件中断。
- 数据域清零。
- 将数据初始化值从 ROM 复制到内存。
- 为栈分配空间，并初始化栈指针。
- 为堆分配空间。
- 激活相关硬件中断。
- 调用或跳转到程序的主执行循环。

有可能在调用主执行循环之后要插入一些代码。程序会报告一些诊断信息用于调试，然后会再次调用主循环。也可能会发出重置命令再次启动整个处理过程，或者根据嵌入式软件的不同目的，简单地终止处理。

嵌入式系统的主线几乎总是一个持续的循环，它去调用函数执行软件的基本任务。复杂一些的系统，主线会初始化一个执行线程的集合，每个线程都有自己的循环完成自己的任务。嵌入式系统中的线程通常被看作是任务（tasks）。

要使用自己的启动代码，有必要指示连接程序忽略来自 GCC 的指令。为达到这样的目的，GNU 连接程序有一个脚本语言，连接程序命令语言（Linker Command Language），通过它可为连接过程提供显式且紧密的控制。脚本语言足够强大（robust），以至于你可能想要跳过 GCC 的连接而指定自己的连接——而脚本语言也足够详细，以至于可以描述如何布置连接对象的各个部分。

## 17.2 选择一种语言

为嵌入式系统编写代码和为通用计算机编写代码是有区别的。尺寸大小是个考虑因素，而且几乎总要考虑速度问题。在桌面系统中正常运行的代码在嵌入式软件中可能会带来资源问题。这种情况实际上就是语言和编译程序的选择问题。

人们通常会有疑问是选择 C 语言还是汇编语言。当然也可使用其他常用语言，但这两种选择目前最流行。使用高级语言工作总是更好一些，因为写出的代码更易读写，也更容易理解。如果代码易读，所有事情都会更快，错误也会更少。即使 C 编译程序产生的代码没有手写的汇编代码那么紧凑和高效，它也非常清晰而且相当有用。汇编语言当然不会浪费时间，但如果不是必要的话，还是不要包含太多的汇编代码。

如果发现对时间的要求非常严格，GCC 可提供一些信息用于确定是否必须使用汇编语言。可能需要考虑以下一些因素：

- **优化**　使用-S 选项令编译程序输出汇编语言。有各种不同的优化级别和不同的优化标志设置。你会发现优化器可以满足所有需求。
- **分析指令**　使用选项-dp 以及选项-S 可令汇编语言在注释中列出每条语句的长度，以及树节点的个数和生成汇编代码行的明确树结构。
- **分析树**　使用选项-dP 和选项-S 产生的结果与使用-dp 选项一样，只是增加相关注释行，其中包含中间语言树的节点。该选项的价值对大多数人而言可能是有限的，因为它的内容首先假设你对 gcc 内部树非常熟悉。
- **检查选项设置**　指明-fverbose-asm 选项会指示编译程序在-S 产生的汇编语言列表顶端包含完整的选项设置的列表，这在程序编译成汇编语言代码的时候是有效的。修改其中的一个或多个设置就可能改变以后产生的汇编语言的类型。

关于生成的代码的其他有用信息，参见第 18 章的描述。最有用的是总尺寸和分配数。

如果确定了所需的汇编语言方案，就可以从它的多种实现中进行选择了。如果需要改动某些分析过的基本函数的内部代码，只需编辑编译程序产生的汇编语言模块。它已经有接口代码了，这些接口代码可被连接到程序的剩余部分。可能更好的办法是先确定所需进行的改动，然后用内嵌的汇编代码取代 C 代码。

## 17.3　GCC 嵌入工具

GCC 编译程序并不是特别为嵌入式软件开发设计的，但作为非常成熟的编译程序，在过去的很多年里它的灵活性已经大大提高了，能够提供嵌入式开发人员要求编译程序做到的所有事情。

### 17.3.1　命令行选项

有些命令行选项对嵌入式编程特别有用。出错检查可对某些你希望察看的特殊内容非常敏感。查找附录 D 中-W 选项集合，设置（或不设置）那些适合特殊环境的选项。也许你想从-Wall 选项开始，指示编译程序要发出警告，甚至是最小的侵犯。如果发现它在报告你不愿看到的警告，可以使用单独的选项设置关掉某个不感兴趣的警告。

优化对于生成的代码非常重要，而编译程序非常适于控制优化设置。参见附录 D 中-O 选项的有关描述。

可以使用-ffixed 命令行选项防止编译程序使用指定的寄存器。例如，如果某个特殊的 CPU 有个寄存器叫做 gr4，而该寄存器不应该被生成的代码所使用，那么可以使用如下的选项集合：

```
$ gcc -c -Wall -ffixed-gr4 mainloop.c -o mainloop.o
```

### 17.3.2 诊断

编译程序能够格式化函数的名字以及源文件的名字，使之成为一个字符串，用于构造诊断消息。例如，以下代码创建的字符串包含当前的函数名、它的源文件名，以及它被编译的日期：

```
sprintf(msg,"Function %s in file %s compiled %s\n",
    __FUNCTION__,__FILE__,__DATE__);
```

使用 C++语言的时候，宏__PRETTY_FUNCTION__对于格式化描述性的函数名做得更好。对 C 用__FUNCTION__或__PRETTY_FUNCTION__产生的结果一样。

### 17.3.3 汇编器代码

按照第 15 章的描述，将汇编器模块和其他高级语言模块相连是个相对简单的任务。而且汇编语言代码可被插入作为内嵌代码，包含在编译好的代码中。

可能需要将汇编器模块连接为可执行程序的一部分，但它不是由 C 写成的，而且名字也不必以下划线字符开头。下面语句定义的 C 符号可在内部使用，为汇编器定义的全局符号定位：

```
extern int musref asm("muslimit");
int rebar asm("rebclean");
extern int gribbit(void) asm("asmgribbit");
```

C 语言源文件中的符号 musref 会被连接到全局定义的名字 muslimit 上，作为到 int 的引用。int 变量 rebar 在汇编语言中定义为 rebclean，而 rebar 的普通声明会产生汇编语言符号_rebar。例子中的第三行是函数原型定义，可将函数声明为 gribbit，并用它进行引用，而实际在汇编语言代码中为 asmgribbit。

GCC 关键字__attribute__ 可用来将 section 名指定为汇编语言中声明的函数或数据项目。例如，下面用__attribute__ 会将变量 trigmax 安置在 convals 部分：

```
const int trigmax __attribute__ ((section("convals")));
```

能够指定部分名字，就能够使用连接程序指出确切地址和放置目标代码的顺序。

## 17.4 库

一般运行时库都是作为嵌入式操作系统软件的一部分提供的。一个好的运行时库可以提供需要的所有事情，因此你只需要编写自己的应用程序代码就可以了。一旦编译并安装了交叉编译程序，就可以进行编译和（或）汇总运行时库，建立指向它的连接程序命令。

如果没有提供运行时库，很有可能需要解析应用程序使用了 GNU 标准库中的哪些部

分。不幸的是，如果用到很多 GNU 库，这种解析过程非常繁琐，这是由于库的交叉引用的广泛性造成的。所幸的是，可通过使用 newlib 库避免该繁琐过程。

### 17.4.1　整顿标准库

使用完整的标准 C 库生成的可执行模块可能数倍大于所需的内容。设计很多标准 C 模块是因为它们的应用很广，而其实现也是从共享库载入内存就被多个进程共同使用。由于嵌入式系统尺寸的限制，这种类型的静态连接模块的代价非常高昂。

最有名的例子为 printf()函数。该函数有很多参数——第一个参数是字符串，其他参数是包括各种数据类型的可变长度列表。因为可在字符串内部动态说明格式信息，能够将任意数据格式转换为所需格式的例程就必需包含进来，作为程序的一部分。另外，很多格式化例程都需要其他库例程。结果产生巨大的多米诺骨牌效应，包含了大量从没用到的代码。

如果要用 GCC 的标准库例程，通过调整库保留需要的模块而去掉不用的模块可能有些莽撞。依据用到的库的数目，这可能是个冗长乏味的过程。可先创建一个库，其中只有你确信会用到的函数调用。然后加入其他模块，因为需要它们来满足不可解析的引用。

### 17.4.2　为嵌入式系统设计的库

嵌入式系统是可以使用标准库进行连接的。该库注册为自由软件，可从站点 http://source.redhat.com/newlib/下载。

库 newlib 是 C 语言库，是为嵌入式系统设计的。库 newlib 的大部分都是由各处收集到的例程组合而成，所有都注册为自由软件。作为资源代码足够简单直接，可以很清楚地在多种处理器上进行编译。newlib 的一个很大优点就是它来自于嵌入式系统，也是专为嵌入式系统编写的。

库的下载和安装与 GCC 和 binutils 非常相似。一旦下载好资源，就应该安装到 newlib 工作目录中了。newlib 目录应该是 gcc 和 binutils 源目录的同属目录。使用脚本可创建一个单独的构造目录，--prefix 指定安装目录，--target 指定目标系统。

察看其他配置选项来确定是否需要为自己的特殊安装使用某些选项是个好主意。使用 configure 脚本的--help 选项可以得到可用选项列表。尤其是--newlib-hw-fp 选项可以编译库的例程，这样它们可以使用浮点运算；默认情况是假设库例程不能使用浮点运算，只能使用整数的基本函数。浮点算法通常更小更快。

使用命令 make，然后是 make install，会创建库。

## 17.5　GNU 连接程序脚本语言

GNU 连接程序由脚本语言控制。如果没有指定脚本，默认使用安装时编译进连接程序

的脚本。可重写该默认脚本，提供自己的脚本，如下例应用脚本 sprig.link 连接可执行加载模块 sprig：

```
$ ld -T sprig.link start.o loop.o brspr.o -o sprig
```

-T 选项指出脚本文件的名字。-c 选项和-T 选项意义相同。

使用特殊脚本的主要原因在于寻址方案（addressing scheme）。通常，连接程序生成的可执行文件地址可调，可在模块被载入内存的时候进行设置。每个部分都有两个地址（它们的值经常一样），一个为模块运行时内部使用的虚内存地址（virtual memory address，VMA），另一个为可加载内存地址（loadable memory address，LMA），指出该部分被载入的位置。嵌入式系统中所有地址都被连接程序解析成绝对地址，因此所有地址引用都是完全解析且不可移动的。锁定地址的过程被叫做定位（locating）模块。一些系统有些独立的工具，可处理从连接程序到绝对模块的可重定位输出，但是 GNU 连接程序将定位器（locator）构造在内部了。

连接程序读出编译程序产生的目标文件，将它们组合成新的目标文件（也叫做可执行文件）作为它的输出。目标文件可分为不同的部分，每部分都有名字和尺寸。连接程序将同名的输入部分组合成一个单独的输出部分。有些部分包括可执行代码，有些部分包含有初始值的数据，其他部分包含未初始化的数据。具有未初始化数据的部分通常没有什么内容，只有名字和尺寸。

### 17.5.1 脚本例子 1

如下例子可用来生成连接好的目标文件。它使用各种输入目标文件的 section，并在指定地址将它们组合起来：

```
SECTIONS
{
    . = 0x0100000;
    .text : {
        *(.text)
    }
    . = 0x8000000;
    .data : {
        *(.data)
    }
    .bss : {
        *(.bss)
    }
}
```

关键字 SECTIONS 指出这是连接的目标模块的内存布置图。SECTIONS 命令的左右括号间的语句是按顺序的，准确指出了输出的布局。

句点是特殊变量，包括当前地址（也叫做位置计数器），用于将数据插入输出。例子中的第一条语句将当前地址设置为绝对值 0x0100000。如果没有设置，输出地址默认为 0。一旦设置了当前地址，通过加到输出中的每个项目，它会自动递增。

语句.text { ... } 位于当前地址输出的.text 部分的开始位置。大括号中的项目会被包含进来，作为输出的.text section 的一部分。本例中输出.text 部分包括输入的所有.text 部分。可在这里列出所有输入文件名，但也可用星号匹配所有文件名。

跟在.text 部分之后的是位置计数器，设置为 0x08000000。它的位置是.data 部分的输出位置。将所有输入的.data 部分组合成一个单独的.data 部分，这会导致位置计数器的前进，就是说它指向的位置超过了.data 部分的底部，而这里正是输出.bss 部分的位置。

### 17.5.2 脚本例子 2

以下连接脚本说明 section 的位置，它的格式很像嵌入式系统中用来创建目标文件的格式。会为 ROM 和 RAM 中存在的内容定位：

```
MEMORY
{
    rom (rx)  : ORIGIN = 0x00000000, LENGTH 1024K
    ram (rwx) : ORIGIN = 0x00100000, LENGTH 512K
}
SECTIONS
{
    .text rom : {
        *(.text)
    }
    .data ram : {
        _StartOfData = . ;
        *(.data)
        _EndOfData = .;
    } > rom
    .bss : {
        *(.bss)
    }
    _HeapLocation = .
    _StackLocation = 0x80000000
}
```

本例以 MEMORY 关键字开头，用来为输出地址空间块分配名字。该技术可用于将输出地址空间分成任意块，后面的指令会将指定的 section 插入指定的内存块。本例中 MEMORY 设置用于指出 RAM 和 ROM 的位置和尺寸，并为它们分配名字以待后用。

可选的内存属性 rx 意味着内存内容可被读出并执行。属性 rwx 意味着内容是可读可写的，并且是可以执行的。如果忽略属性设置，则所有权限都是许可的。

前面定义的内存位置允许使用名字，而不是用来设置地址的数字。通过将名字和位置定义为.text rom，可将输出的.text 部分定位在 ROM，而指定为.data ram，可将.data 部分定位在 RAM。按照顺序取得指定项目，因此如果某个地址没有指明，会假设它就在上一个项目的结尾处。例如，.bss 部分紧跟在.data 部分后面输出。

脚本包含的符号，例如_StartOfData 和_EndOfData，在连接处理过程中是全局定义变量。这些名字通过访问设置它的内存地址可在程序内部直接使用。符号_HeapLocation 定义为 RAM 中的地址，紧跟在.bss 部分之后，而_StackLocation 设置为绝对地址 0x80000000。

### 17.5.3 其他脚本命令

OUTPUT_FORMAT 命令对于得到某种格式的可载入开发系统的可执行模块非常重要。例如，下面命令产生的输出结果为 Intel 十六进制格式：

```
OUTPUT_FORMAT("ihex")
```

一些适用于此命令的二进制文件描述符（binary file descriptor，BFD）是"binary"、"ihex"（指 Intel hex）、"srec"（指 S-records）、"coff-sh"（指 SH-2）和"coff-m68k"（指 CPU32）。在脚本中使用 OUTPUT_FORMAT 和在连接程序的命令行中使用--oformat 是一样的，而且它有同样的 BFD 名字变量集合。

INPUT 命令用于库的列表和/或每个连接想要包含的目标文件。例如，下面的命令会促使两个库和一个目标文件总被包含进来：

```
INPUT(libc.a libg.a startmod.o)
```

可用命令 OUTPUT_FILENAME 为输出文件命名，如下：

```
OUTPUT_FILENAME("loadable.out");
```

# 第 18 章

# 编译程序输出

编译程序的基本目的是产生目标文件、包含目标文件的库，以及可执行程序。但也可以使用编译程序生成其他类型的输出。你会发现并不是经常需要这种信息，但有时在某些特殊情况下，当问题的线索非常少的时候，编译程序就非常有用了。

使用某些选项就能够发现编译程序从语法上怎样理解你的程序，这里编译程序要去查找子进程和库，得到解析程序产生的中间语言列表。可以得到程序中包含的所有头文件的列表，可为基于源代码的 makefile 自动产生依赖关系语句。

## 18.1 程序信息

编译程序构造详细的内部表，它包括编译程序的有关信息，可用命令行选项得到这些信息。不仅可以检查解析树，其中包括编译程序对你的代码的解释，也可以得到包含的所有头文件的列表，还可以得到编译花费的时间，以及程序中每个模块需要的内存。对于 C++ 程序，还可以得到类定义关系。

### 18.1.1 解析树

编译程序将程序解析成内部树。树结构（代表原始的源代码）通过使用选项 -fdump-translation-unit 可输出到后缀为.tu 的文件中，如下例所示：

```
$ gcc -fdump-translation-unit showdump.c -o showdump
```

该命令产生的输出文件 showdump.c.tu 中包含了树的文本表示。树中的每个节点都按照数字表示（例如@1，@2 等），每个树节点代表的树结构通过数字指向其他树节点。

每个节点显示的信息总数可在一定程度上加以控制。以下格式产生的树包括编译程序的内部地址信息，该地址信息可用-d 选项（本章稍后会介绍）产生的内部地址对解析树进行交叉引用。

```
$ gcc -fdump-translation-unit-address showdump.c -o showdump
```

下面两种形式会产生列表，分别包含较多或较少的信息：

```
$ gcc -fdump-translation-unit-all showdump.c -o showdump
$ gcc -fdump-translation-unit-slim showdump.c -o showdump
```

由这些选项生成的树简单易读。下面是一部分树，说明每个节点都是由惟一的 ID 号和描述性名字加以标识。也包括属性列表：

```
@1  function_decl    name: @2     type: @3     srcp: showdump.c:5
                     chan: @4     args: @5     extern
@2  identifier_node  strg: main   lngt: 4
@3  function_type    size: @6     algn: 64     retn: @7
                     prms: @8
@4  var_decl         name: @9     type: @7     srcp: showdump.c:3
                     chan: @10    init: @11    size: @12
                     algn: 32     used: 1
@5  parm_decl        name: @13    type: @7     scpe: @1
                     srcp: showdump.c:4        chan: @14
                     argt: @7     size: @12    algn: 32
                     used: 0
```

在树的这个级别上，大多数属性都以其他树节点来定义。例如，name 属性是树节点的数目，它有 strg（串）和 lngth（长度）属性。有些节点，例如 function_type，有 algn（对齐）属性。变量，例如参数和声明，都有 type 属性和 name 属性，也会有 used 属性指出该变量被程序使用的次数。许多节点都有 srcp （源位置）属性，指出产生每个节点的源文件的名字和行号。

### 18.1.2 头文件

选项-H 也可写做--trace-includes，生成所有包含文件的嵌套列表。以下例子是 Linux 系统中只包含 stdio.h 的 C 程序产生的输出：

```
. /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stdio.h
.. /usr/include/features.h
... /usr/include/sys/cdefs.h
... /usr/include/gnu/stubs.h
.. /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
.. /usr/include/bits/types.h
... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
... /usr/include/bits/pthreadtypes.h
.... /usr/include/bits/sched.h
.. /usr/include/libio.h
... /usr/include/_G_config.h
.... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
.... /usr/include/wchar.h
```

```
..... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
..... /usr/include/bits/wchar.h
.... /usr/include/gconv.h
..... /usr/include/wchar.h
...... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
..... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stddef.h
... /usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/include/stdarg.h
.. /usr/include/bits/stdio_lim.h
Multiple include guards may be useful for:
/usr/include/bits/pthreadtypes.h
/usr/include/bits/sched.h
/usr/include/bits/stdio_lim.h
/usr/include/gnu/stubs.h
```

包含的每个级别都是由名字前面的句点数指定的。而且列表的末尾是头文件的名字，很有可能是固定的，因为多次包含可能产生多重定义的问题。

### 18.1.3　程序需要的内存

可要求编译程序产生所编译程序需要的内存总数，以及如何分配这些内存的细节。如下例子输出展示了报告的详细格式：

```
RTX                 Number                Bytes   % Total
address                  7                   56     0.664
const_int              129                 1032    12.239
const_double            21                  336     3.985
const_vector            19                  152     1.803
pc                       1                    8     0.095
reg                     14                  224     2.657
mem                    216                 3456    40.987
symbol_ref             391                 3128    37.097
cc0                      1                    8     0.095
plus                     1                   16     0.190
eq                       1                   16     0.190
Total                  801                 8432

Size              Allocated                Used  Overhead
8                      8192                6216       184
16                      12k                4192       180
32                     8192                3392        88
64                      32k                 28k       288
512                     28k                 24k       196
1024                   4096                1024        28
112                     52k                 42k       416
20                     8192                2580       104
```

```
Total                 152k                    112k      1484
String pool
entries          452
identifiers      452 (100.00%)
slots            16384
bytes            4805  (3339  overhead)
table size       64k
coll/search      0.0168
ins/search       0.7609
avg. entry       10.63 bytes (+/- 5.78)
longest entry    36
```

由该列表可以确定分配给程序不同部分的内存总数，以及分配的每块内存是如何使用的。这在分析大程序和嵌入式系统的目标模块的时候尤其有用。

### 18.1.4 时耗

-time 选项在编译和连接的时候，会令 gcc 列出每个独立进程耗费的时间。例如，下面命令将三个 C 程序编译成汇编语言，激活汇编器为每一个程序都生成目标文件，再用 collect2 将它们连接起来：

```
gcc -time getshow.c strmaker.c showstring.c -o getshow
# cc1 0.15 0.02
# as 0.01 0.00
# cc1 0.08 0.03
# as 0.01 0.01
# cc1 0.13 0.03
# as 0.01 0.00
# collect2 0.13 0.05
```

每个进程列出两个时间，第一个是用户时间（执行子进程代码的时间总和），第二个是系统时间（进程执行系统调用的时间总和）。不会列出实际的时钟时间，但如果要列出整个 gcc 进程耗费的时间总和，包括时钟时间，可通过使用标准 time 工具运行 gcc 就可将其加入，如下面例子所示：

```
$ time gcc -time getshow.c strmaker.c showstring.c -o getshow
```

### 18.1.5 C++中间树

可指示 g++编译程序输出前端翻译产生的中间语言。在编译过程的任何阶段都可以输出中间语言。如下命令显示生成的原始中间语言，还未经过修改或优化：

```
$ g++ -fdump-tree-original minmax.cpp -o minmax
```

也可输出优化后的中间语言：

```
$ g++ -fdump-tree-optimized minmax.cpp -o minmax
```

可对中间语言执行函数的内嵌过程，可通过下面的命令输出内嵌的结果：

```
$ g++ -fdump-tree-inlined minmax.cpp -o minmax
```

输出格式可通过在每个输出选项名字的结尾标识修改子加以说明。在选项末尾附加 -address 会加入地址信息，与-d 选项生成的地址信息相关（本章后面会有所描述）。要减少列表中包含的信息总数，可通过 -slim 标识指明。要增加输出的信息总数，可附带-all 标识。例如，下面命令会产生优化后中间语言的详细输出：

```
$ g++ -fdump-tree-optimized-all minmax.cpp -o minmax
```

### 18.1.6　C++类层次

可指示 g++ 编译程序输出完整的类层次关系和程序的虚函数表。输出也含有程序使用的系统类层次关系，因此输出结果可能非常大。下面的命令会编译并输出程序 minmax.cpp 的完整类层次关系：

```
$ g++ -fdump-class-hierarchy minmax.cpp -o minmax
```

该命令的输出结果是可执行程序 minmax 以及一个文本文件 minmax.cpp.class，后者包含了类的层次关系。下面命令的输出结果一样，只不过类继承关系也包含了地址信息，可与-d 选项产生的信息进行交叉引用：

```
$ g++ -fdump-class-hierarchy-address -da minmax.cpp -o minmax
```

-d 选项输出一些内部的编译程序信息，本章稍后有所描述。

可通过使用如下选项减少输出的信息总数：

```
$ g++ -fdump-class-hierarchy-slim minmax.cpp -o minmax
```

可用下面的选项获得一个包含所有可用信息的更大的输出文件：

```
$ g++ -fdump-class-hierarchy-all minmax.cpp -o minmax
```

## 18.2　有关 makefile 的信息

存在一个选项集合，可用于指导编译程序扫描源文件并生成依赖关系插入 makefile。例如，下面的程序包含两个头文件：

```
/* getshow.c */
#include "strmaker.h"
#include "showstring.h"
int main(int argc,char *argv[])
```

```
{
    char *string;
    string = strmaker();
    showstring(string);
}
```

以下的编译程序命令会读该源文件并为 makefile 生成依赖关系行（本例中，头文件 strmaker.h 包含 motback.h）：

```
$ gcc -M getshow.c
getshow.o: getshow.c strmaker.h motback.h showstring.h
```

-M 选项设置-E 选项，禁止所有输出但不禁止依赖关系行。如果希望产生依赖关系行，并继续编译，键入如下命令：

```
$ gcc -MD getshow.c -o getshow
```

该命令会产生可执行的 getshow 并将依赖关系文本保存到文件 getshow.d 中。-MF 选项可用来指明文件的名字，如下例所示，会将依赖关系放置在文件 depends.text 中：

```
$ gcc -MD -MF depends.text getshow.c -o getshow
```

-MF 选项也可和-M 一起使用，禁止编译并将依赖关系保存在文件中，如下：

```
$ gcc -M -MF depends.text getshow.c
```

指明输出文件名字的另一种方法就是设置环境变量 DEPENDENCIES_OUTPUT。

选项-M 和-MM 会进行检查，如果缺少头文件就会报告错误。如果想要禁止出错消息，可以指明-MP 选项以及-M 和-MM，它会为每个头文件生成一个哑目标（dummy target）。

-MT 选项可与-M 或-MM 一起使用，指明目标名字，如下例所示：

```
$ gcc -M -MT spang.o getshow.c
spang.o: getshow.c strmaker.h motback.h showstring.h
```

## 18.3 编译程序信息

有些选项可用来确定正在使用的是什么编译程序，以及它是如何配置的。例如，可用如下命令列出编译程序的版本信息：

```
$ gcc -dumpversion
```

为了确定目标机器——即编译程序为哪种类型的计算机创建目标文件——可键入如下命令：

```
$ gcc -dumpmachine
```

### 18.3.1　编译时间

-ftime-report 选项可用于生成编译的各个阶段耗时的列表。这主要是为编译程序开发人员准备的，但也可用它检验自己的程序的复杂程度。用此选项编译的输出结果与下面的类似：

```
Execution times (seconds)
 garbage collection  :   1.13 (23%) usr  0.00 ( 0%) sys  0.50 (10%) wall
 life analysis       :   0.01 ( 0%) usr  0.00 ( 0%) sys  0.00 ( 0%) wall
 preprocessing       :   0.43 ( 9%) usr  0.08 (24%) sys  1.00 (20%) wall
 lexical analysis    :   0.38 ( 8%) usr  0.10 (29%) sys  0.00 ( 0%) wall
 parser              :   2.72 (56%) usr  0.14 (41%) sys  3.00 (60%) wall
 expand              :   0.02 ( 0%) usr  0.00 ( 0%) sys  0.00 ( 0%) wall
 varconst            :   0.05 ( 1%) usr  0.00 ( 0%) sys  0.50 (10%) wall
 integration         :   0.03 ( 1%) usr  0.01 ( 3%) sys  0.00 ( 0%) wall
 local alloc         :   0.01 ( 0%) usr  0.00 ( 0%) sys  0.00 ( 0%) wall
 global alloc        :   0.01 ( 0%) usr  0.00 ( 0%) sys  0.00 ( 0%) wall
 rest of compilation :   0.00 ( 0%) usr  0.01 ( 3%) sys  0.00 ( 0%) wall
 TOTAL               :   4.84            0.34            5.00
```

显示的这些值以秒为单位，以及它们占总数的百分比。usr 时间是编译程序内部执行代码的实际耗时。sys 时间是系统调用内部的耗时（例如输入输出），而 wall 时间是实际消耗的时间。

### 18.3.2　子进程开关

gcc 程序是其他程序（例如语言编译程序、汇编器和连接程序）的前端。在配置和编译 gcc 的时候，子进程的名字以及传递给它们的选项都被配置好并安装起来了。为确定构造子进程命令行参数所使用的规范，键入如下命令：

```
$ gcc -dumpspecs | more
```

传递给子进程的选项和参数的规范由单独一个串组成。每个基本子进程的规范定义的默认设置被构造在 gcc 中，并自动成为编译程序前端的一部分，但也可在配置编译程序的时候重写默认的规范串。

所列信息的例子是激活 C 预处理程序的规范：

```
*cpp:
%{posix:-D_POSIX_SOURCE} %{pthread:-D_REENTRANT}
```

用此规范，无论何时 gcc 激活 cpp，gcc 命令行的--posix 选项会令 cpp 命令行中出现-D_POSIX_SOURCE，而在 gcc 命令中出现--pthread 会令 cpp 命令中出现-D_REENTRANT。

规范文本中，为所有可能传递给子进程的选项定义条件的那行规范，可以相当复杂。

下面这个复杂一点的例子（但不意味这是最复杂的），它的规范设置用来激活汇编器：

```
*asm:
%{v:-V} %{Qy:} %{!Qn:-Qy} %{n} %{T}
```

在本例中，如果 gcc 命令指明-v ，汇编器就会指定-V 选项。如果 gcc 命令指明-Qy ，它不会传递给汇编器，但如果没有指明-Qn ，那就会为汇编器加上选项-Qy。如果 gcc 指明-n 或-T，它们都会传给汇编器。没有其他选项传给汇编器。

### 18.3.3 详细的编译程序调试信息

-d 选项可用来指示 GCC 系统在编译过程的各个阶段输出内部信息。该信息只对那些编译程序开发人员有意义，甚至由于该信息过于详细，以至于对于调试和分析应用程序没有什么帮助。

可要求在编译过程中的某些点上产生输出信息。完整的列表请参见附录D中的-d条目。所有输出点上的输出大体都一样，包括删除不必要指令的信息、寄存器分配、寄存器释放（当寄存器清空自己值的时候），以及在内部 RTL 语言中产生的指令。例如，下面的简单程序测试两个值以确定进入哪个分支：

```
/* showdump.c */
int a = 44;
static int b = 22;
int main(int argc,char *argv[])
{
    if(a > b) {
        b = a;
    } else {
        a = b;
    }
}
```

下面的命令会编译程序，并要求生成 RTL 代码之后立即产生输出：

```
$ gcc -dr showdump.c -o showdump
```

输出的信息保存在文件 showdump.c.00.rtl 中，看起来如下：

```
;; Function main

(note 2 0 5 NOTE_INSN_DELETED -1347440721)

(insn 5 2 6 (nil) (parallel[
          (set (reg/f:SI 7 esp)
              (and:SI (reg/f:SI 7 esp)
                  (const_int -16 [0xfffffff0])))
```

```
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (nil))

(insn 6 5 7 (nil) (set (reg:SI 59)
        (const_int 0 [0x0])) -1 (nil)
    (expr_list:REG_EQUAL (const_int 0 [0x0])
        (nil)))

(insn 7 6 8 (nil) (parallel[
            (set (reg/f:SI 7 esp)
                (minus:SI (reg/f:SI 7 esp)
                    (reg:SI 59)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (nil))

(insn 8 7 3 (nil) (set (reg/f:SI 60)
        (reg/f:SI 55 virtual-stack-dynamic)) -1 (nil)
    (nil))

(note 3 8 4 NOTE_INSN_FUNCTION_BEG -1347440721)

(note 4 3 9 NOTE_INSN_DELETED -1347440721)

(note 9 4 10 NOTE_INSN_DELETED -1347440721)

(note 10 9 12 NOTE_INSN_DELETED -1347440721)

(insn 12 10 13 (nil) (set (reg:SI 61)
        (mem/f:SI (symbol_ref:SI ("a")) [0 a+0 S4 A32])) -1 (nil)
    (nil))

(insn 13 12 14 (nil) (set (reg:CCGC 17 flags)
        (compare:CCGC (reg:SI 61)
            (mem/f:SI (symbol_ref:SI ("b")) [0 b+0 S4 A32]))) -1 (nil)
    (nil))

(jump_insn 14 13 15 (nil) (set (pc)
        (if_then_else (le (reg:CCGC 17 flags)
                (const_int 0 [0x0]))
            (label_ref 22)
            (pc))) -1 (nil)
    (nil))
```

```
(note 15 14 16 NOTE_INSN_DELETED -1347440721)

(note 16 15 18 NOTE_INSN_DELETED -1347440721)

(insn 18 16 19 (nil) (set (reg:SI 62)
        (mem/f:SI (symbol_ref:SI ("a")) [0 a+0 S4 A32])) -1 (nil)
    (nil))

(insn 19 18 20 (nil) (set (mem/f:SI (symbol_ref:SI ("b")) [0 b+0 S4 A32])
        (reg:SI 62)) -1 (nil)
    (nil))

(jump_insn 20 19 21 (nil) (set (pc)
        (label_ref 28)) -1 (nil)
    (nil))

(barrier 21 20 22)

(code_label 22 21 23 2 "" "" [0 uses])

(note 23 22 24 NOTE_INSN_DELETED -1347440721)

(note 24 23 26 NOTE_INSN_DELETED -1347440721)

(insn 26 24 27 (nil) (set (reg:SI 63)
        (mem/f:SI (symbol_ref:SI ("b")) [0 b+0 S4 A32])) -1 (nil)
    (nil))

(insn 27 26 28 (nil) (set (mem/f:SI (symbol_ref:SI ("a")) [0 a+0 S4 A32])
        (reg:SI 63)) -1 (nil)
    (nil))

(code_label 28 27 29 3 "" "" [0 uses])

(note 29 28 33 NOTE_INSN_FUNCTION_END -1347440721)

(insn 33 29 34 (nil) (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))

(insn 34 33 31 (nil) (clobber (reg:SI 58)) -1 (nil)
    (nil))

(code_label 31 34 32 1 "" "" [0 uses])
```

```
(insn 32 31 35 (nil) (set (reg/i:SI 0 eax)
        (reg:SI 58)) -1 (nil)
    (nil))

(insn 35 32 0 (nil) (use (reg/i:SI 0 eax)) -1 (nil)
    (nil))
```

## 18.4　关于文件和目录的信息

可用选项集合要求编译程序察看磁盘，以查找某些内容。因为由系统配置决定编译程序查找库的目录，这样就会有需求要能够检查实际用到的库的位置。可通过-print-file-name 选项实现这一目的。例如，下面的命令确定了库 libgcc.a 的位置：

```
$ gcc -print-file-name=libgcc.a
/usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/libgcc.a
```

选项-print-file-name 可用来定位所有库，但库 libgcc.a 有它自己的选项，如下例所示：

```
$ gcc -print-libgcc-file-name
/usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/libgcc.a
```

按照相似的模式，可以确定内部子进程的完整路径名，如 cc1 和 cc1obj。例如，键入如下命令定位 f771：

```
$ gcc -print-prog-name=f771
/usr/lib/gcc-lib/i586-pc-linux-gnu/3.2/f771
```

键入下面命令可以确定当前 GCC 的安装目录，以及程序和库的完整查找路径：

```
$ gcc -print-search-dirs > path.text
```

该命令的输出非常大，而且路径被列在一个连续行中，因此最好重定向输出到某个文件中，这样便可以使用编辑器帮助对其进行分析。首先列出的是安装目录，然后是程序和库。有些路径名来自于算法，因此可能过分详细，但如果需要了解查找顺序，可由该命令的输出得到。

# 第 19 章

# 实现一种语言

在 GCC 编译程序内部，前端（front end）分析编程语言的词法和语法，而后端（back end）产生目标机器的代码。GCC 的设计允许存在多个前端，每个前端都是不同的编程语言。如果要为 GCC 编写自己的前端，所有现有的后端（也叫端口）都可被调用，因此你的语言也可移植到很多机器上。

概念很简单，但实现起来就不那么容易了。假设你有一个语言解析器能够识别语言元素，而该语言正是希望实现的语言，但事实是必须能够将该前端连接到 GCC 的剩余部分上。解析器产生的输出必须符合可识别的格式。GCC 前端并不是与后端孤立开来的，尽管也许应该这样，因此不能只单纯考虑解析器的输出。而且，还有可能要为语言开发运行时库。

## 19.1 从前到后

GCC 的目的就是要读出编程语言的源代码，从中产生可执行程序。下面的步骤序列概括了编译过程：

- **词法分析** 读出源代码，对它进行标号。过程通常包括从流中一次一个字符地读出源代码，确定哪些字符应该在一起才有语言意义。标号通常可以分为三类：名字、数字和标点。每种语言都有自己的规则集——关于这三类哪些是合法的哪些是非法的。
- **语法分析** 标号之间有关系，这种关系很大程度上依赖于输入流的词法扫视中它们彼此的相对位置。解析器确定每个标号的类型（关键字、符号名字、数字，等等），利用这些信息要由完整的源文件形成树。树中的节点代表数据声明、函数、每条语句，等等。整个程序由树表示。
- **剪除** 在分析树中的每个条目时会进行一些优化处理。树中冗余或未被使用的部分会被去掉。树中某些部分会被移到树中的其他部分，以防止执行不必要的语句。

- **RTL**　解析树的内容会转换为寄存器传送语言（Register Transfer Language，RTL）代码。RTL 是种伪汇编语言，包括假想机器的操作码（opcodes）。解析树能“展开（unrolled）”生成线性序列的 RTL 指令。如果必要会重新组织树中的语句，有必要的话也会插入分支，这会根据解析树中定义的 if 条件进行测试。也会插入 case/switch 类型语句的分支和循环语句。大多数翻译都在目标依赖阶段执行——就是说，根据目标机器生成 RTL 代码，它包括一些事情例如寄存器的定位信息。
- **RTL 优化**　对 RTL 代码执行优化。这些优化包括例如消除尾部递归、消除通用子表达式、跳转优化及其他。这里是进行优化的绝好地点，因为它适用于每种语言前端及每种语言后端。
- **汇编语言**　RTL 被翻译成目标机器的汇编语言，并写入文件。
- **汇总**　激活汇编器将汇编语言文件翻译成目标文件。该文件不是执行格式——包括可执行的目标代码，但不是可载入形式。因为，很有可能包含了未解析的到例程的引用和其他模块的数据。
- **连接**　连接程序将来自汇编器的目标文件（有些保存在充满目标文件的库里面）组合成可执行程序。

应该注意到从前端语言解析器到后端代码生成器之间没有逻辑上的划分，而解析树是中间产物。所有解析器都能产生树结构并可通过 RTL 代码生成器连接到后端，并由 GCC 进行编译。与此相似，所有为它们编写代码生成程序将 RTL 语言翻译成本地汇编语言的机器，都能够由前端从处理的所有语言中产生编译后的程序。

实际上并不像描述的这样简单，但基本情况就是这样的。

## 19.2　词法扫视

编译程序以字符流的方式读出程序源代码，将字符分组作为标号（tokens）以待处理。标号可为数字、名字或标点。例如，下面这行就由 7 个标号组成：

```
if (grimle <= 43.1) {
```

将行划分成标号的过程叫做词法扫视（lexical scan），或简写为 lex。词法扫视的机械过程对所有语言都一样，除了规则的改动——定义哪些字符是合法符号，哪些是合法标点字符。事实上，不同编程语言的处理过程足够一致以至存在这样的标准工具，可用来编写自己的词法扫视程序。标准的 UNIX 工具 lex——或是 GNU 等效工具 flex——都被给予你的语言遵守的规则集，而且它产生程序，可用来从输入源中生成标号流。

### 19.2.1 简单的词法扫视程序

作为简单词法扫视程序定义的例子，下面定义了两个关键字 howdy 和 now：

```
%%
howdy printf("(The word is 'howdy')");
now printf("(The time is %ld)",time(0L));
%%
```

%%符号指出字符匹配列表的开始和结束。本例会检测两词有否匹配，然后执行随后的命令。命令实际是 C 程序语句，包含在由脚本产生的程序中。下面一对语句会创建 C 程序 lex.yy.c，然后将它编译成可执行的 howdy：

```
$ flex howdy.lex
$ gcc lex.yy.c -lfl -o howdy
```

本例产生的程序 lex.yy.c 是超过 1500 行的 C 代码，它调用了库 libfl.a 中的例程。输出代码如此之大的一个原因是注释的数目——代码中的注释足够令其相对简单，可确定它是如何工作的。如果使用的是标准 UNIX lex 工具，而不是 GNU 的 flex 程序，命令的格式有点区别：

```
$ lex howdy.lex
$ gcc lex.yy.c -ll -o howdy
```

该程序可从命令行执行。它运行之后会等待输入，可从键盘键入。键入的所有不是两个可识别关键字的内容都会简单地返回到标准输出上，而这两个关键字可由 printf()函数调用中的字符串所取代。

### 19.2.2 Lex 的正则表达式

下面的 lex 定义可识别关键字 switch 和 case、任意随机符号、任意整数，以及左右括号：

```
%%
switch printf("SWITCH ");
case printf("CASE ");
[a-zA-Z][_a-zA-Z0-9]*  printf("WORD(%s) ",yytext);
[0-9]+ printf("INTEGER(%s) ",yytext);
\{ printf("LEFTBRACE ");
\} printf("RIGHTBRACE ");
%%
```

前两条规则匹配关键字 switch 和 case。第三条规则和所有以大写或小写字母开头，之后跟着零个或多个字母、数字或下划线字符的符号相匹配。注意输出串包括 yytext，它是

到标号串自身的指针。第四条规则和所有一或多个数字匹配。最后两条规则和左右括号匹配。

这个 lex 例子可从如下输入文本中提取出标号：

```
blatz {
  switch big_time_do
  case HamFram
  case 889
} dend
```

以下命令序列会将 lex 脚本 kwords.lex 编译成程序 kwords，然后用它把源文件 kwtry.text 标号化：

```
$ flex kwords.lex
$ gcc lex.yy.c -lfl -o kwords
$ cat kwtry.text | kwords
WORD(blatz)  LEFTBRACE
  SWITCH  WORD(big_time_do)
  CASE  WORD(HamFram)
  CASE  INTEGER(889)
RIGHTBRACE  WORD(dend)
```

## 19.3 语法分析

本节描述的例子试图描述使用词法扫视读出标号并用语法解析器对标号进行逻辑组织的过程，以及用组织好的信息调用 C 函数集合。编译程序使用 C 函数产生输出（GCC 中，它的输出是 RTL 中间语言代码），但本例中的输出是简单的文本行，描述了被生成的代码。

实际执行语法分析的代码可由标准 UNIX 工具 yacc 生成，这是 Yet Another Compiler Compiler 的缩写。执行同样任务的 GNU 工具为 bison。这两个工具在目的和功能上几乎完全一样。

例子基于非常简单的语言，叫做 clang，可以接受命令在指定位置画彩色的圆和矩形。下面就是 clang 程序的例子：

```
set color blue;
set location (100,200);
draw circle 30;
set color red;
set location (250,200);
draw rectangle (10,10);
```

set 语句用来指出下一个待画图形的位置和颜色。draw 语句利用指定的类型和尺寸来画出图形。

下面是文件 clang.lex 的内容：

```
/* clang.lex */
%{
#include "y.tab.h"
extern int yylval;
extern char *yytext;
%}

%%
set             { return(SETTOKEN); }
color           { return(COLORTOKEN); }
location        { return(LOCATIONTOKEN); }
draw            { return(DRAWTOKEN); }
circle          { return(CIRCLETOKEN); }
rectangle       { return(RECTANGLETOKEN); }
\;              { return(SEMICOLON); }
\,              { return(COMMA); }
\(              { return(LEFTPAREN); }
\)              { return(RIGHTPAREN); }
[0-9]+          { yylval = atoi(yytext);
                  return(NUMBER);
                }
[a-zA-Z][a-zA-Z0-9]*  { yylval = strdup(yytext);
                        return(NAME);
                      }
\n              /* ignore end of line */
[ \t]+          /* ignore white space */
%%
```

包含文件 clang.tab.h 由 bison 从解析器文件中产生，稍后有所描述。每个词法定义都返回一个指定类型的值（在头文件中定义）。因为用该定义生成 C 的源代码，使用反斜线转义字符来转义识别为表号的标点更安全。

每个输入标号都保存为串，由变量 yytext 所指。为使该标号可用于 C 例程，有必要将标号值保存——作为它的可用类型——到变量 yylval 中。在本例中，NAME 标号保存为串，而 NUMBER 标号调用 atoi()转换成了整数。

最后两个标号匹配什么也没有产生，但如果希望成功扫描过多个空格、制表符以及行结束符，它们都是必须的。创建一个面向行的语言的解析器，可让新行符号返回一个值，使得解析器可以察觉到它。

下面是文件 clang.y 的内容，包括语言的语法定义：

```
%start commands
```

```
%token SETTOKEN DRAWTOKEN COLORTOKEN
%token LOCATIONTOKEN CIRCLETOKEN RECTANGLETOKEN
%token SEMICOLON LEFTPAREN RIGHTPAREN COMMA
%token NUMBER NAME

%%

commands:
    /* nothing */
    | commands command
    ;

command: SETTOKEN set SEMICOLON
    | DRAWTOKEN draw SEMICOLON
    ;

set: COLORTOKEN NAME
      { setcolor($2); }
    | LOCATIONTOKEN LEFTPAREN NUMBER COMMA NUMBER RIGHTPAREN
      { setlocation($3,$5); }
    ;

draw: CIRCLETOKEN NUMBER
      { drawcircle($2); }
    | RECTANGLETOKEN LEFTPAREN NUMBER COMMA NUMBER RIGHTPAREN
      { drawrectangle($3,$5); }
    ;

%%
```

文件的第一行指出语法树定义的开始点。接着是标号定义——这些是生成的代码中的命名常量，被用作输入流中每个标号的惟一标识。

解析定义中的每个条目都叫产品（production）。每个产品都有名字，而名字都和定义在它右边的一个或多个语法分布相关。处于右边的语法项目由竖线（|）符号分隔，最后一个由分号终止。解析器用输入的标号流和产品右边的项目进行匹配，直到找到相关匹配。

由 bison 或 yacc 产生的这种解析器从左到右读出标号以确定匹配，会向前看到不超过一个标号。这种解析器叫做 LALR(1)解析器，或简单叫做 LR(1)解析器。这对处理现代编程语言已经足够了，但较老的语法更加含糊的语言需要特殊处理。现代语言的设计都会遵照 LR(1)解析器原则。

开始的产品为 commands。commands 产品可为空（它会出现在文件的结尾）或可以包含一条或多条命令的列表。首次察看该产品时，可能出现回溯——但事实是在它指向下一

个产品之前它是指向自己的，而且不得不处理解析器所生成递归代码的这种自然属性。实际上两种方式都可以工作，但所展示的这种顺序效率更高。

commands 产品可以匹配 set 或 draw 语言关键字。匹配的那个关键字用来确定用于匹配下面标号的产品。set 关键字将解析器指向产品 set，而 draw 关键字将解析器指向 draw 产品。产品名字和关键字不一定要匹配，但这样更易读。

每个产品内部都是括号封装的一些 C 代码。这可以是任意的 C 代码，但本例简单地调用了一个函数，其中被更进一步地描述过了。函数的参数由项目在产品中的位置（或值）决定。参数$1 是第一个，$2 是第二个，以此类推。注意本例中传给函数的值可为 NAME 标号或 NUMBER 标号，在 lex 定义中有它的 C 代码，为它们赋值为 yylval。

剩下要做的所有事情就是定义 C 代码，用来从源代码中产生目标代码。本例不是产生代码，只是打印了要产生的代码的描述。下面的 C 源文件包含所有解析器都需要的函数，以及定义语言的产品调用的函数：

```
/* clmain.c */
#include "clang.tab.h"
#include <stdio.h>

char colorname[30] = "black";
int x = 0;
int y = 0;

main()
{
   yyparse();
}
int yywrap()
{
   return(1);
}
void yyerror(const char *str)
{
   fprintf(stderr,"Clang: %s\n",str);
}

int setcolor(char *name)
{
   strcpy(colorname,name);
   return(0);
}

/* Save the x and y location of the
```

```
  next figure to be drawn. */
int setlocation(int xloc,int yloc)
{
   x = xloc;
   y = yloc;
}

/* Draw a circle of the of the specified size and color
  at the current location. */
int drawcircle(int radius)
{
   printf("Draw %s circle at (%d,%d) radius=%d\n",
      colorname,x,y,radius);
}

/* Draw a rectangle of the specified height, width, and color
  at the current location. */
int drawrectangle(int height,int width)
{
   printf("Draw %s rectangle at (%d,%d) h=%d w=%d\n",
      colorname,x,y,height,width);
}
```

头文件 clang.tab.h 是 bison 由 clang.y 产生的一个产品，包括一些对代码有用的常数定义。main()函数是编译程序的主线。本例中，它调用 yyparse()执行解析过程，但实际的编译程序负责创建中间语言，管理从中间语言到目标语言的转换，执行优化处理，确定输入和输出文件的名字，负责命令行选项，以及编译程序进行的其他动作。

yywrap()函数在当前输入文件的结尾处被调用，也可用来开始读另一个源文件。返回值 1 指出没有更多的输入。

yyerror()函数在发生错误的时候被解析器调用。传递给函数的字符串包括错误的描述。例子简单地将错误消息打印到标准错误上。

setcolor()函数在任何时候使用关键字 set 指定新颜色的时候被解析器调用。依赖于产生的代码，以及基本的图形设备，该函数能够生成代码对颜色进行修改，这种情况下，在本地保存颜色信息以便以后要用到它。

setlocation()函数和 setcolor()函数相似，只不过它定义了用来画下一个图形的位置。本例将坐标保存在本地，这样可在实际画图的时候用到它。

drawcircle()和 drawrectangle()函数会被调用实现真正的画图操作。前者设置颜色和位置的信息可被用作生成指令的一部分。该例简单地打印用来生成代码的信息。

以下的命令序列用于编译源文件并连接成程序，可用于读出源代码产生画图的伪指令：

```
$ bison -d clang.y
$ flex clang.lex
$ gcc clmain.c lex.yy.c clang.tab.c -o clang
```

bison 命令读出输入源文件 clang.y，产生输出文件 clang.tab.c。文件 clang.tab.c 包括解析输入的 C 代码，这样它应被编译并连接进编译程序。而且因为指定了选项-d，也会产生文件 clang.tab.h。这是用于 clmain.c 和 clang.lex 的头文件，提供所有标号类型的数字定义。

flex 命令用于产生文件 lex.yy.c，包含的 C 函数用来读出输入流并将它们组织成标号。

gcc 命令用于编译三个 C 源文件，并连接成可执行程序 clang。就像所写的那样，编译程序从标准输入接受源代码，这样测试程序的源文件 figures.clang，可被下面命令处理：

```
$ cat figures.clang | clang
```

输出结果看起来如下：

```
Draw blue circle at (100,200) radius=30
Draw red rectangle at (250,200) h=10 w=10
```

## 19.4 创建解析树

解析操作的输出为解析树。树的实际格式为文本行的线性列表，每行就是树中的一个节点。每个节点都有一个标识以便被其他节点引用，它包括一个字符指出节点类型。节点类型以及指示它们的字符都列在表 19-1 中。

**表 19-1 语法解析树节点类型的指示符**

| 指示符 | 类型描述 |
|---|---|
| < | 比较表达式 |
| 1 | 一元算术表达式 |
| 2 | 二元算术表达式 |
| b | 词汇块 |
| c | 常数 |
| d | 变量声明或变量引用 |
| e | 非比较、一元或二元表达式，没有负效应 |
| r | 到内存位置的引用 |
| s | 必然有负效应的表达式 |
| t | 数据类型 |
| x | 不适合其他分类的特殊节点 |

节点类型指示符是在 GCC 源文件 tree.def 中定义的。很多函数的存在就是为了创建树节点，可在文件 stmt.c 中找到它们。有很多可用函数可以创建树节点——如此之多，事实上所有可能想到的语句看起来都有自己的 RTL 代码生成器。例如，下面的函数生成的代码会比较 op1 和 op2，并在二者相等的时候进入分支 label。

```
static void do_jump_if_equal(op1,op2,label,unsignedp);
```

本例中的 op1 和 op2 都是表达式树节点，而 label 是可执行代码中位置的内存引用。最后一个参数指出该比较是符号还是非符号表较。这个函数检查参数确定应该生成什么样的代码（例如，如果 op1 和 op2 都是常数值而且彼此相等，就会生成一个简单的分支指令）。一旦确定了指令类型，就会调用例程实际发布此指令。

低级的 RTL 生成例程在源文件 emit-rtl.c 中。可能其中最简单的是函数 emit_note()，它发出的指令什么也不做只是占了个位子。实际创建了指令且将它们加入 RTL 输出的代码看起来如下：

```
note = rtx_alloc(NOTE);
INSN_UID(note) = cur_insn_uid++;
NOTE_SOURCE_FILE(note) = file;
BLOCK_FOR_INSN(note) = NULL;
add_insn(note);
```

在该代码序列中，通过调用 rtx_alloc()会创建新的类型尺寸合适的 RTL 树节点。一个树节点（文件 rtl.h 中定义为结构 rtx_def）在它的头部包含标识标记的集合，跟着的变长数组包含的是操作数。宏 INSN_UID 插入惟一的树节点 ID 号。宏 NOTE_SOURCE_FILE 将源文件信息加入该节点。

对 add_insn()的调用将新构造的节点加入连接列表的尾部，它是 RTL 代码。函数 add_insn_before()可将新指令插在现有指令之前，而 add_insn_after() 可将新指令插在紧跟现有指令的后面。

符号表信息不会带到 RTL 内。前端存在的某些格式的符号表必须解决到名字的引用问题，但在这里可以忽略它，因为 RTL 代码使用节点的 ID 号进行直接引用。但是符号表必须存在，并且从编译程序的后端必须能够访问该表。

## 19.5　连接后端与前端

编译程序的后端不能与前端清晰地分离开来。必须声明大量的全局变量和函数作为前端的一部分，这样就可从后端直接进行访问。

前端代码应该和它的主 gcc 目录下的子目录相独立。例如，cp 目录包括 C++代码，而 ada 目录包括 Ada 编译程序的代码。此目录中有文件 Make-lang.in，包含在父 gcc 目录的主

makefile 中，本语言使用此 makefile 文件。也会包含 makefile.in 文件，用于产生其他语言的 makefile。configure 脚本使用文件 config-lang.in。

驱动程序 gcc 必须改动才能包含新的语言，但这种改动是自动发生的，作为构造过程中的一部分。

前端必须包括一定的全局变量和函数，它们被后端所引用。这样做的目的是提供某种方式可以访问树节点和符号表，以及总的初始化和清除工作。表 19-2 包括所需全局变量的简要描述。表 19-3 包括编译程序前端现有函数的简要描述，这些函数可被后端定位。很多这样的函数和变量也用于前端，但它们必须作为全局变量而存在，并使用这些名字。

**表 19-2 GCC 后端定位的前端变量**

| 名字 | 描述 |
|---|---|
| error_mark_node | 树的父节点，包括节点代表输入中的错误条件 |
| integer_type_node | 基本整数类型的树节点 |
| char_type_node | 基本字符类型的树节点 |
| void_type_node | 基本空类型的树节点 |
| integer_zero_node | 整数 0 的树节点 |
| integer_one_node | 整数 1 的树节点 |
| tree_current_function_decl | 代表当前翻译函数的树节点 |
| language_string | 命名语言的字符串地址 |
| flag_traditional | 必须的，只用于 C |

**表 19-3 GCC 后端调用的前端函数**

| 名字 | 描述 |
|---|---|
| lang_init() | 执行所有特定语言的初始化 |
| lang_finish() | 执行所有特定语言的终止和清除 |
| lang_decode_option() | 被命令行中选项调用 |
| init_lex() | 执行词法分析需要的所有初始化 |
| init_parse() | 执行解析器需要的所有初始化 |
| finish_parse() | 执行所有解析器的终止和清除 |
| type_for_mode() | 返回代表机器数据类型的树节点 |
| type_for_size() | 返回指定精确位数数字的整数树节点 |
| type_for_unsigned() | 返回指定尺寸的无符号整数树节点 |
| signed_type() | 返回指定尺寸的符号整数树节点 |
| signed_or_unsigned_type() | 返回指定类型和指定符号的树节点 |

（续表）

| 名字 | 描述 |
| --- | --- |
| init_decl_processing() | 初始化表 19-2 列出的树节点变量 |
| global_bindings_p() | 返回值，指出当前范围是否为全局范围 |
| kept_level_p() | 返回值，指出当前级别是否需要创建数据块 |
| getdecls() | 返回树，列出当前范围级别的所有声明 |
| pushdecl() | 将声明插入符号表后返回树节点 |
| pushlevel() | 创建符号表中新的范围级别 |
| poplevel() | 放弃符号表的当前范围级别，回到以前的状态 |
| insert_block() | 在当前范围级别的块列表尾端增加一个新块 |
| set_block() | 为当前范围级别设置块节点 |
| maybe_build_cleanup() | 也许返回树节点代表清除以前行为后所采取的动作（例如毁掉对象） |
| truthvalue_conversion() | 返回和指定表达式一样的表达式，除了它的结果是真或假 |
| mark_addressable() | 标记指定表达式为定位内存 |
| copy_lang_decl() | 复制指定的声明树节点 |
| incomplete_type_error() | 打印使用不完整类型的出错消息 |
| yyerror() | 打印解析出错消息 |
| print_lang_decl() | 将树节点声明输出到指定文件 |
| print_lang_type() | 将树节点的类型信息输出到指定文件 |
| print_lang_identifier() | 将树节点的标识信息输出到指定文件 |
| set_yydebug() | 为语法分析设置调试选项 |

# 第 20 章

# 寄存器传送语言

寄存器传送语言（Register Transfer Language，RTL）是编译过程的中心。编译程序前端的目的就是产生 RTL，而后端的目的就是将 RTL 翻译成汇编语言。对程序的大多数优化处理过程都是在 RTL 形式下完成的。本章是对 RTL 形式的描述。

RTL 代码可按照文本格式输出到磁盘文件中。第 18 章中的例子解释了将 RTL 代码输出成可打印格式所需的步骤。本章包括 RTL 代码的描述，以及需按 dumped 格式读出嵌入式代码的有关信息。

## 20.1 RTL insn

RTL 中一条语句叫做 insn。insn 在内部连接成为双向连接列表。有些 insn 实际上是指令，其他则包含一些信息，例如分支语句 swtich 用到的分支表。还有一些代表数据声明，可作为分支目标的标记。而且，每个 insn 都有惟一的 ID，任何 insn 都可以通过它引用其他 insn。

### 20.1.1 6 种基本表达式代码

insn 的种类有很多。每种都用一个表达式代码指定其类型。组成程序逻辑流的 RTL 代码有 6 种基本类型，每种都可以保存到其他类型的引用。例如，指向某个可执行语句的 insn 表达式代码，可包含其他 insns 作为操作数。例如，下面的 insn 读出 val 变量的值，保存到寄存器里。为此 RTL 代码要有一个 insn 的表达式代码，并包含表达式代码为 set 的 insn，依次用到的 insn 的表达式代码为 reg 和 mem。该 mem insn 包括 symbol_ref insn。

```
(insn 12 10 14 (nil) (set (reg:SI 61)
    (mem/f:SI (symbol_ref:SI("val")) [0 a+0 S4 A32])) -1 (nil)
    (nil))
```

insn 表达式代码是 6 种基本表达式代码类型之一。每个 RTL 程序都是由表 20-1 中列出的 6 种表达式组合而成的。

表 20-1　6 种基本 RTL 表达式代码

| 表达式代码 | 描述 |
|---|---|
| insn | 该表达式代码用于非跳转指令，且不会进行函数调用。这种 insn 可加载寄存器、执行算法、比较值，等等 |
| jump_insn | 使用该表达式代码的指令会（或可能）跳转到其他位置，这就意味着 insn 通常包含一个或多个 label_ref insns。该表达式代码也用于从当前函数中返回。code_label insn 的引用是跳转的目的地，也包含在简单的条件或非条件跳转中。对于更复杂的跳转，有必要扫描整个 insn 体来查找目标 |
| call_insn | 使用该表达式代码的指令会（或可能）执行函数调用。必须特殊处理这些指令，因为它们可能对寄存器和内存位置进行不可预知的改动。典型地，这种 insn 会包含 clobber 和 mem insns 来指出改动的是哪个寄存器和内存位置。也有一个 mem insn 指定了传入参数的内存块，或由 clobber 和 use insns 指出工作寄存器和寄存器关系参数 |
| code_label | 该表达式代码用来标识跳转目标。其中包括的 code_label_number insn 保存惟一的标识 ID 号。ID 号对整个编译单元都是惟一的，而不仅仅是在当前函数中。code_label insn 通常指向 label_ref insn 中的跳转位置。在优化过程中和优化结束以后，都维护了一个数字记录跳转目标使用的次数。每个 code_label 都是以下 4 种之一：<br>NORMAL　只有这种标识不能改变进入函数的入口。<br>STATIC_ENTRY　该标识是进入函数的入口，但只在编译单元内可见。<br>GLOBAL_ENTRY　该标识是函数的入口，对所有编译单元都可见（通过连接程序）。<br>WEAK_ENTRY　该标识是函数入口且是全局入口，但可被同名的其他符号改写。 |
| barrier | 该表达式代码放置在 insns 序列中以标识控制流不可达的位置。barrier insn 会被插入，跟在无条件跳转之后，或跟在不能返回的函数调用之后 |
| note | 该表达式代码包含了特定的调试和声明信息。每个 note insn 都有一个包含数字的域，而其他域则包括字符串。数字通常是以字符串为名的源文件的行号。note insn 控制的行号信息可用于调试。如果数字不是行号，就会是一种类型指示器，指定以下类型：<br>NOTE_INSN_DELETED　该 note 标识删除 insn 的位置。<br>NOTE_INSN_DELETED_LABEL　该 note 替换删掉的 code_label insn，去掉它是因为它从未成为跳转的目标。<br>NOTE_INSN_BLOCK_BEG　标识代码块中某级别范围的开始。<br>NOTE_INSN_BLOCK_END　标识代码块中某级别范围的结束。<br>NOTE_INSN_EH_REGION_BEG　标识异常处理级别范围的开始。<br>NOTE_INSN_EH_REGION_END　标识异常处理级别范围的结束。<br>NOTE_INSN_LOOP_BEG　标识 while 或 for 循环的开始。 |

（续表）

| 表达式代码 | 描述 |
| --- | --- |
| | NOTE_INSN_LOOP_END 标识 while 或 for 循环的结束。<br>NOTE_INSN_LOOP_CONT 在循环中标识一个位置，continue 语句会跳转到那里。<br>NOTE_INSN_LOOP_VTOP 在循环中标识一个位置，退出测试从这里开始。<br>NOTE_INSN_FUNCTION_END 标识接近函数尾的地点，return 语句会跳转到该符号前。<br>NOTE_INSN_SETJMP 标识的代码紧跟在 setjmp()类型函数的调用之后。 |

### 20.1.2 insn 的类型和内容

每种 insn 都是惟一的，设计它们是为分别实现某个特定用途的；因此每种 insn 包含的数据都是为了满足各自的目的。其中包含的数据对其他类型的 insn 是很常见的事情，但连接的 RTL 指令链总能够追溯到某些基本数据。RTL 语言的语句包括的数据格式为表 20-2 中列出的 5 种 insn 之一。

表 20-2 RTL 基本 insn 数据类型

| insn 类型 | 描述 |
| --- | --- |
| Expression | 可执行语句。RTL 表达式叫做 RTX |
| Integer | 整数数据类型，格式为 C 语言的 int 数据类型 |
| Wide integer | 整数数据类型，格式由 HOST_WIDE_INT 定义，通常为 64 位值 |
| String | 内部保存的字符序列，终止符为空 null，和标准 C 一样。字符串最可能用于符号名引用，但也可用来代表机器描述信息，如第 21 章中所述。零长度的字符串代表内部的空指针 |
| Vector | 表达式指针的任意数。其中成员的数目在 insn 中有显式的声明。零长度的 Vector 在内部代表空指针 |

RTX 的格式和内容变化很大，但总存在的三个域是：ID、前一个 insn 的地址、下一个 insn 的地址。这三个值可用下面的宏从任何 insn 中析取出来。

```
INSN_UID(insn)
PREV_INSN(insn)
NEXT_INSN(insn)
```

第一个 insn 可通过调用 get_insns()获得，而最后一个则可通过调用 get_last_insn()获得。

GCC 中处理 insn 的代码大多数是为了处理表达式。RTL 的表达式 insn 叫做 RTX。这些表达式 insn 是包括程序可执行代码的语句。在 GCC 代码内部，它们保存在结构中，通过定义为 rtx 的指针加以引用。

每个 RTX 都有自己的表达式代码 expression code（或 RTX 代码）指出这是哪种表达式。在文件 rtl.def 中定义的表达式代码是枚举集合，包含的是常数名。表达式代码是与机器无关的，这意味着 RTL 语言是机器无关的。RTX 代码可置为 rtx 结构，然后通过使用下面两个定义在文件 rtl.h 中的宏获得：

```
PUT_CODE(rtx,code);
int code = GET_CODE(rtx);
```

宏 DEF_RTL_EXPR 用来定义文件 rtl.def 中的每个 RTL。该宏有 4 个参数，如下面的例子所示：

```
DEF_RTL_EXPR(COND_EXEC,"cond_exec","ee",'x')
```

向宏传递的第一个参数为 RTL 的名字，全为大写字母。用在 C 源代码中作为 enum 类型，是 RTL 的惟一标识。第二个参数的 RTL 名为小写 ASCII 字符串，该名字会打印到诊断输出中。第三个参数是操作数的数据类型列表，每种类型都指派了一个字符。类型的描述可参见表 20-3。第四个参数是一个字母，为 RTL 的类指示子。所有类都列在表 20-4 中。

**表 20-3　RTX 操作数类型的代码**

| 操作数代码 | 描述 |
|---|---|
| * | 未指明。企图处理这种类型的操作数会产生警告消息 |
| 0 | 未使用的域 |
| b | 指向位图头的指针 |
| B | 指令基本块的定义（一个入口，一个出口） |
| e | 指向 RTL 表达式的指针 |
| E | 指向 RTL 表达式数组的指针 |
| i | 整数 |
| n | 指明以下任意一种的整数：<br>1 指令删除<br>2 块开始<br>3 块结束<br>4 循环开始<br>5 循环结束<br>6 循环继续<br>7 循环顶部的指令<br>8 循环结束条件<br>9 函数结束 |

（续表）

| 操作数代码 | 描述 |
|---|---|
| | 10 函数前言的结束 |
| | 11 函数尾声的开始 |
| | 12 删除的标识 |
| | 13 函数开始（入口点） |
| | 14 异常处理区域的开始 |
| | 15 异常处理区域的结束 |
| | 16 重复的行号 |
| | 17 基本块 |
| | 18 期望值 |
| | 19 前置量 |
| s | 字符串 |
| S | 可选字符串 |
| u | 指向其他 insn 的指针 |
| t | 树指针 |
| T | 被汇编或编译执行的代码。这是汇编语言源代码，除非第一个字符是星号，这时该语言为 C |
| V | 可选指针，指向 RTL 表达式数组 |

表 20-4 RTL 类代码

| 类 | 描述 |
|---|---|
| < | RTX 代码，为比较操作符，例如小于或等于 |
| 1 | RTX 代码，为一元算术操作符，例如负号或某人的问候语 |
| 2 | RTX 代码，为不可交换的二元操作符，例如减法或除法 |
| 3 | RTX 代码，非位操作的三元操作符，例如 if/then/else |
| a | RTX 代码，为自动递增寻址模式 |
| b | RTX 代码，位操作，例如零析取或符号析取 |
| c | RTX 代码，为可交换的二元操作符，例如加法或乘法 |
| g | RTX 代码，将指令组织在一起 |
| i | 机器指令的 RTX 代码，例如跳转或调用 |
| m | 用于匹配 insns 的 RTX 代码 |
| o | 代表对象的 RTX 代码，例如内存位置或寄存器 |
| x | 不适合任何其他类的 RTL 代码 |

不同 RTL 代码操作数的数目和类型都不一样。表 20-3 描述了指出操作数数据类型的代码。这些代码并不需要严格遵守。例如，尽管用代码 T 指定可执行代码的源代码，而用 s 指定简单串，但有些 RTL 定义也将 C 源代码作为 s 类型包含进来。每种 RTL 代码都会被分别处理，因此操作数几乎可做所有事情——用于指定类型的字母非常概括，只用于调试方便打印操作数。

下面的列表描述了 RTL 代码。每个 RTL 都列出自己的名字，跟着是类和字符串，用来说明与它相关的操作数类型和数目。每一条都包含对其目的的描述及其意义的列表，还有该操作数应用的类型。操作数有数据类型，在引用串中按顺序出现。GCC 代码中，RTL 代码名字的大写形式用来定义枚举的类型，当做 RTL 的数字 ID——例如，eq_attr RTL 用常数 EQ_ATTR 作为惟一标识。正如描述中所指出的，一些 RTL 代码的作用特殊——例如，有些是用来创建表达式列表的，还有一些只作方便之用而从未出现在 insn 中。它们都只是机器描述中的特殊条目。

- abs‘1”“e”　如果表达式的结果为负值，将它转换为正值。
- absence_set ‘x’ “ss”　只出现在机器描述中，用来指出 CPU 功能单元列表，如果不能保留其他功能单元，这种功能单元也不能保留。例如，在 VLIW 处理器中，slot0 不可保留在 slot1 或 slot2 之后。参见 presence_set。操作数 0 是逗号分隔的功能单元列表，除非至少保留了操作数 1 的一个功能单元，否则也不会保留它。操作数 1 是逗号分隔的功能单元列表。
- addr_diff_vec ‘x’ “eEee0”　它包含的地址向量与空间计算上的基本操作和目标操作有很大区别。操作数 2、3 和 4 只有当编译程序中定义了 CASE_VECTOR_SHORTEN_MODE 时才有效。操作数 0 是基本操作数。操作数 1 是地址差，代表每个操作数到基本操作之间的距离。操作数 2 是标记，代表最小的地址。操作数 3 的标记代表最大地址。操作数 4 是标志的集合，确定简化标志的原则。而标志"min_align"是指所有分支都要使用最小对齐单位。标志"base_after_vec"指出基址跟在 addr_diff_vec 之后。标志"min_after_vec"指出最小地址目标标记跟在 addr_diff_vec 之后。标志"max_after_vec"指出最大地址目标标记跟在 addr_diff_vec 之后。标志"min_after_base"指出最小地址目标标记跟在基址之后。标志"max_after_base"指出最大地址目标标记跟在基址之后。标志"offset_aligned"指出偏移量必须被看作无符号值。标志"scale"指出有必要令偏移量满足某种模式。
- addr_vec ‘x’ “E”　地址向量。包含的每个地址对 code_label 看来都是 label_ref。
- address ‘m’ “e”　到参数地址的引用。操作数 0 是指地址的表达式。
- addressof ‘o’ “eit”　到寄存器地址的引用，在尽可能多的寄存器地址由于忽略而删除的时候，编译程序的 purse_addressof()也会将它删除。操作数 0 是寄存器。操作数 1 是原始伪寄存器数，正是为此生成了 insn。操作数 2 是保存在寄存器中项

目的声明，由 put_reg_in_stack 使用。

- and ‘c’“ee” 对两个操作数进行计算，结果占一位宽，有两个值。
- ashift ‘2’“ee” 字节的逻辑向左移位。操作数 0 是表达式产生的值，会被移位。操作数 1 是表达式，产生的值为其移位的数目。
- ashiftrt ‘2’“ee” 字节的算术向右移位（符号扩展）。操作数是表达式产生的值，会被移位。操作数 1 为表达式，产生的值是所移位的数目。
- asm_input ‘x’“s” 传递给汇编器的指令串。可用在其他 insn 内部，当作代码模式的一部分，也可用于向汇编器代码插入注释。
- asm_operands ‘x’“ssiEEsi” 带操作数的汇编语言指令。操作数 0 是定义指令的模板。操作数 1 是输出的约束关系。操作数 2 是标识值，以便区分不同汇编语言的语句。操作数 3 是用作输入操作数的值的集合。操作数 4 是模式的集合，以及输入操作数的约束关系。数组中的每个成员都是 asm_input，具有约束字符串指出输入操作数的模式。操作数 5 为包含源文件的名字。操作数 6 是包含源文件的行号。
- attr ‘x’“s” 只用在机器描述中以便定义 insn 属性。这是标记符，可插入指定属性的名字。操作数 0 是属性的名字。
- attr_flag ‘x’“s” 只用在机器描述中以便定义属性。如果条件表达式为真，该设置指出执行分支的可能性，由标志指出应该执行哪条 insn。有效的标志值为"forward"、"backward"、"very_likely"、"likely"、"very_unlikely"和"unlikely"。操作数 0 是标志的值。
- automota_option ‘x’“s” 只用在机器描述中，作为生成自动操作的选项。对操作数 0，选项"no-minimization"意味着自动操作不能最小化，只有在自动状态下查询 CPU 功能单元预留时才有意义。选项"time"请求在生成自动操作过程中会额外打印时间统计数据。选项"v"请求生成的文件后缀为.dfa，包括测试和调试信息。选项"w"对不严重的错误也会产生出错消息而不是警告消息。选项"ndfa"会生成不确定最终状态的自动操作（nondeterministic finite-state automata）。
- barrier ‘x’“iuu” 一个标记，指出不会通过的控制流。操作数 0 是这个 RTX 的惟一 ID。操作数 1 是到链中前一条指令的指针。操作数 2 是到链中下一条指令的指针。
- call ‘x’“ee” 调用子例程。操作数 0 是被调子例程的地址。操作数 1 是传给子例程参数的个数。
- call_insn ‘i’“iuuBteieee” 一个 insn，可能会调用子例程，但不能改动子例程返回的地址。操作数 0 是这个 RTX 的惟一标识。操作数 1 是链中前一条指令的指针。操作数 2 是链中下一条指令的指针。操作数 3 是指令基本块。操作数 4 是树节点的指针。操作数 9 是 call_insn_function_usage insn，调用函数。

- call_placeholder ‘x’ “uuuu”　占位符，被 call_insn 取代，或被同属调用取代，或被尾递归（tail recursion）所取代。操作数 0 是这个 RTX 的惟一标识。操作数 1 是链中前一条指令的指针。操作数 2 是链中下一条指令的指针。操作数 3 是 code_label insn，作为尾递归的标记。如果不可能发现尾递归，则为空。
- clobber ‘x’ “e”　指示符，指出没必要解释某些东西的用法。例如，某个子例程调用会在内部使用寄存器，重写其中的值。也见 use。操作数 0 是表达式，指出使用的项目。
- cc0 ‘o’ “”　代表条件代码寄存器。使用 insn 的逻辑就好像条件代码寄存器已经包含了值，可与零做比较，但事实上真/假设置是前面比较的结果。
- code_label ‘x’ “iuuB00iss”　是一个标记，其后跟着的是指令。操作数 0 是这个 RTX 的惟一标识。操作数 1 是链中前一条指令的指针。操作数 2 是链中下一条指令的指针。操作数 3 为指令基本块，后面跟着的是标记。操作数 4 是标记的跳转数目。操作数 5 是指针，指向引用标记的标记链。操作数 6 是惟一标识数。操作数 7 是用户分配给标记的名字（如果有的话）。操作数 8 是内部使用的可选标记名。
- compare ‘2’ “ee”　对两个操作数分别计算并比较是否相等，如果相等产生零值，如果不等产生非零值。
- concat ‘o’ “ee”　两个表达式的串接，以此产生值的位数等于两个表达式位数的和。这可用于复杂数字，经常出现在 RTL 生成过程中，但不会出现在最终的 insn 链中。
- cond ‘x’ “Ee”　普通的条件语句。操作数 0 是一对表达式向量。依次计算每对的第一个成员，而第二个成员则是条件表达式，结果为零则是假，结果非零则为真。如果操作数 0 中的这对表达式结果都不为真，则将操作数 1 用作条件表达式。
- cond_exec ‘x’ “ee”　条件表达式以及代码块，该代码块只有在条件结果为真的情况下才会执行。条件表达式没有副作用。操作数 0 是条件表达式。操作数 1 是执行的 insn。
- const ‘o’ “e”　结果为常量的表达式。编译程序强制将它看作常量，而不是看作成为最终代码的表达式。
- const_double ‘o’ “ww”　数字浮点常数。操作数是值链，全部由双精度值组成。语法为"ww"，但它的范围实际上是从"ww"到"wwwww"，这依赖于目标硬件上浮点格式的尺寸。
- const_int ‘o’ “w”　数值整数常量。
- const_string ‘o’ “s”　字符串常量。这次它只用作属性。
- const_vector ‘x’ “E”　向量（数组）常量。
- constant_p_rtx ‘x’ “e”　__builtin_constant_p 表达式。只有打开了优化处理，才

会在生成 RTL 的时候产生它，并且会在传递第一个 CSE 的时候被删除。

- define_asm_attributes ‘x’“V” 为汇编语言指令定义属性计算。操作数 0 是向量，包含属性的列表。
- define_attr ‘x’“sse” 只用于机器描述，定义 insn 属性。操作数 0 为定义属性的名字。操作数 1 是可能的属性值列表，用逗号分隔。操作数 2 是属性默认设置的表达式。
- define_automaton‘x’“s” 只出现在机器描述中，为管道冒险识别（pipeline hazards recognition）使用的自动操作命名。该名字用在 define_cpu_unit 和 define_query_cpu_unit 中。操作数 0 是逗号分隔的名字列表。
- define_bypass ‘x’“issS” 只出现在机器描述中，指出从一种 insn 集到另一种的反应时间（latency）。操作数 0 为反应时间的值。操作数 1 是 insn 名字列表，由逗号分隔，反应时间由此开始计时。操作数 2 是 insn 名字的逗号分隔的列表，反应时间计时到此结束。操作数 3 是可选函数名，它收到两个 insn 参数，返回一个旁路值（bypass value）。如果某些特殊情况忽略该旁路，会返回零值。
- define_cond_exec ‘x’“Ess” 操作之后条件执行的定义，产生 define_insn 的新的实例。操作数 0 指出用于匹配的表达式。操作数 1 是 C 语言条件表达式，结果是与模式匹配的非零值。操作数 2 是 C 或产生汇编输出的汇编语言代码块。
- define_cpu_unit ‘x’“sS” 只出现在机器描述中，定义 CPU 功能单元的名字。操作数 0 是 CPU 功能单元名字列表，由逗号分隔。操作数 1 是自动操作的名字，见 define_automaton 描述。
- define_delay ‘x’“eE”为延时时间片定义需求。操作数 0 是条件表达式，值为真指出 insn 需要指定数目的延时片断。操作数 1 是向量（它的长度是指定所需时间片数目的长度的三倍），每个时间片包括三个条件的集合。第一个为真说明由 insn 占据该片位置。第二个为真说明该 insn 如进入分支可废除它。第三个为真说明该 insn 可在不进入分支的时候废除它。
- define_expand ‘x’“sEss” 定义如何为标准 insn 名字产生多个 insn。操作数 0 是标准 insn 名。操作数 1 是匹配的 insn 模式向量。操作数 2 是 C 语言表达式，对有效的操作其结果必须为非零值。操作数 3 是在生成 insn 之前被执行的 C 语言代码。例如，它可以生成 RTX sequence，用于产生代码。
- define_function_unit ‘x’“siieiiV” 需要函数单元的 insn 集合。就是说延时之后，每个 insn 都会产生结果，可能是对某种 insn 数目的限制，这种 insn 被调度以同步执行，因为有可能对 CPU 的函数单位数目有所限制。同一个 CPU 可生成多个 define_function，但对同样的函数单元第一个操作数必须相同。操作数 0 是函数单元的名字。操作数 1 是 CPU 内指定函数单元的数目。操作数 2 是 CPU 可同步运行

的函数单元数目。数字 0 是指没有限制。数字 1 指同时只有一个 insn 可使用函数单元。操作数 3 是条件表达式，涉及的函数属性结果只有在函数用于 insn 的时候才会非零。操作数 4 是常数延时值，之后，使用函数的 insn 结果会变得有效。操作数 5 是常数延时值，在它之后，另一条 insn 可以使用同样的函数单元。如果指出操作数 6，它是属性表达式的列表。如果任何表达式结果为非零值，函数单元正在执行，而且会插入合适的延时数目。如果所有表达式结果都为零，函数单元可用，而且可以立即调度，目的就是限制被操作数 2 指定。

- define_insn ‘x’“sEsTV”　它只出现在机器描述中，为一种指令的定义。操作数 0 是指令的名字。如果名字为空串，指令自己就是机器描述的一部分，目的是可在匹配中被识别，永远不会出现在 RTX 中。操作数 1 是表达式模式。操作数 2 是 C 表达式，指出识别模式的附加条件。如果是空串，没有其他条件。操作数 3 是汇编语言，定义匹配成功后采取的行动。如果表达式以星号开头，这是 C 语言而不是汇编语言。操作数 4 是可选的 insn 属性向量（见 set_attr 和 set_attr_alternative）。
- define_insn_and_split ‘x’“sEsTsESV”　insn 和它相应划分的定义。它是由 define_insn 和 define_split 串联而成，这里它们共享相同的模式。操作数 0 是指令名。如果名字为空，指令保存在机器描述中只用作匹配，而不会用来创建实际的 RTX。操作数 1 是匹配的表达式向量。操作数 2 是 C 的源代码，可用作匹配中的额外表达式。该操作数可为空串。操作数 3 是找到匹配后所执行动作的源代码。操作数 4 是表达式的 C 语言源代码，结果必须为非零值。操作数 5 是位于 sequence 的表达式向量。操作数 6 是产生 insn 之前执行的 C 语言代码。例如，它可以编码产生 RTX sequence，用于生成代码。操作数 7 是该 insn 的属性向量。
- define_insn_reservation ‘x’“sies”　这只出现在机器描述中，以描述保留的 CPU 功能单元。操作数 0 是串，用作调试跟踪的描述性输出。操作数 3 是正则表达式，用于选择指令。正则表达式语法为：表达式中的竖线（|）用于 OR 操作符，而加号（+）用作 AND 操作符。星号（*）用于将元素重复指定次数。cpu_function_unit_name 是 CPU 功能单元名字。reservation_name 由 define_reservation 定义。以下是正则表达式的语法：

```
regexp  = regexp "," oneof
        | oneof

oneof   = allof "+" repeat
        | repeat

repeat  = element "*" number
        | element
```

```
element = cpu_function_unit_name
        | reservation_name
        | result_name
        | "nothing"
        | "(" regexp ")"
```

- **define_peephole ‘x’ “sEsTV”** 窥孔优化（peephole optimization）的定义。操作数 0 是优化的名字。操作数 1 是指令向量，可对它实行优化。操作数 3 是 C 语言表达式，结果为非零值。操作数 4 是可选的 insn 属性列表（见 set_attr 和 set_attr_alternative）。
- **define_peephole2 ‘x’ “EsES”** RTL 窥孔优化操作的定义。操作数 0 是匹配的 insn 向量。操作数 1 是 C 语言表达式，结果为非零值。操作数 2 是位于 sequence 的 insn 向量。操作数 3 是生成 insn 之前执行的 C 语言代码。例如，它可以产生 RTX sequence，用于产生代码。
- **define_query_cpu_unit ‘x’ “sS”** 只出现在机器描述中，描述 define_cpu_unit 定义的 CPU 功能单元。操作数 0 是 CPU 功能单元名列表，由逗号分隔。操作数 1 是自动操作的名字，见 define_automaton 的描述。
- **define_reservation ‘x’ “ss”** 只出现在机器描述中，指出通常作为一组的 CPU 功能单元集合。操作数 0 是分配给功能单元集合的名字。操作数 1 是由名字指出的 CPU 功能单元列表。
- **define_split ‘x’ “EsES”** 划分操作定义。操作数 0 是匹配的 insn 向量。操作数 1 是 C 语言表达式，结果为非零值。操作数 2 是位于 sequence 的 insn 向量。操作数 3 是产生 insn 之前执行的 C 语言代码。例如，它可以产生 RTX sequence，用于产生代码。
- **div ‘2’ “ee”** 计算两个表达式，操作数 1 除以操作数 0。
- **eq ‘<’ “ee”** 产生的代码计算表达式并执行符号比较。如果第一个值等于第二个值，返回真。
- **eq_attr ‘x’ “ss”** 仅用于机器描述中，用来定义属性。通过比较属性名和比较值，可以确定是否应用了该属性。操作数 0 是属性的名字。操作数 1 是比较值。
- **exclusion_set ‘x’ “ss”** 仅出现在机器描述中，指出不能和其他 CPU 功能单元并行调度的 CPU 功能单元。列在第一个操作数中的功能单元不能和第二操作数所列单元进行调度。一个例子就是单精度和双精度浮点操作不能同步执行。操作数 0 是功能单元列表，由逗号分隔，不能和操作数 1 进行调度。操作数 1 是功能单元列表，由逗号分隔。
- **expr_list ‘x’ “ee”** 连接的表达式列表。
- **ffs ‘1’ “e”** 计算表达式，对表达式结果尾部的零位计数。

- fix ‘1’“e”　将浮点表达式的值转换成定点值。也见 unsigned_fix。
- float ‘1’“e”　将定点表达式的值转换成浮点值。也见 unsigned_float。
- float_extend ‘1’“e”　产生代码计算浮点表达式，并且如果需要的话，将它扩展成更大的格式进行保存。
- float_truncate ‘1’“e”　产生代码计算浮点表达式，并且如果需要的话，将它截断成更小的格式进行保存。
- ge ‘<’“ee”　产生代码计算表达式，执行符号比较。如果第一个值大于或等于第二个值，结果为真。
- geu ‘<’“ee”　产生代码计算表达式，执行无符号比较。如果第一个值大于或等于第二个值，结果为真。
- gt ‘<’“ee”　产生代码计算表达式，执行符号比较。如果第一个值大于第二个值，结果为真。
- gtu ‘<’“ee”　产生代码计算表达式，执行无符号比较。如果第一个值大于第二个值，结果为真。
- high ‘o’“e”　这是 RISC 机器中常量表达式的高位值。
- if_then_else ‘3’“eee”　代表条件跳转指令。操作数 0 是条件表达式。无论何时条件表达式计算为真，操作数 1 就为跳转的目标表达式。无论何时条件表达式计算为假，操作数 2 就会是跳转的目标表达式。
- include ‘x’“s”　操作数是包含文件的名字。
- insn ‘i’“iuuBteiee”　不可分的 insn。操作数 0 是这个 RTX 的惟一标识。操作数 1 是链中前一指令的指针。操作数 2 是链中下一条指令的指针。操作数 3 是不可分的指令基本块。操作数 4 是树节点的指针。
- insn_list ‘x’“ue”　连接的指令列表。
- ior ‘c’“ee”　计算两个操作数，结果为一位宽的两个值的或 OR 操作。
- jump_insn ‘i’“iuuBteiee0”　可能分支的 insn。操作数 0 是这个 RTX 的惟一标识。操作数 1 是链中前一指令的指针。操作数 2 是链中下一条指令的指针。操作数 3 是不可分的指令基本块。操作数 4 是树节点的指针。
- label_ref ‘o’“u00”　汇编语言中到标记的引用。操作数 0 是在链中找到的 code_label insn。也见 symbol_ref。操作数 1 是 flow.c 使用的 LABEL_NEXTREF 声明。操作数 2 是 flow.c 使用的 CONTAINING_INSN 声明。
- le ‘<’“ee”　产生的代码计算表达式，进行符号比较。如果第一个值小于或等于第二个值，结果为真。
- leu ‘<’“ee”　产生的代码计算表达式，进行无符号比较。如果第一个值小于或等于第二个值，结果为真。

- lo_sum ‘o’“ee” 寄存器和 RISC 机器的常量表达式的低位之和。
- lsshiftrt ‘2’“ee” 按位进行逻辑右移。操作数 0 表达式产生的值进行右移。操作数 1 表达式产生的值是指移动的位数。
- lt ‘<’“ee” 产生的代码计算表达式，执行符号比较。如果第一个值小于第二个值，结果为真。
- ltgt ‘<’“ee” 产生的代码计算表达式，执行无序浮点比较。如果第一个值不等于第二个值，结果为真。也见 uneq。
- ltu ‘<’“ee” 产生的代码计算表达式，执行无符号比较。如果第一个值小于第二个值，结果为真。
- match_insn ‘m’“is” 只出现在机器描述中。操作数 0 是操作数表的索引。操作数 1 是进行匹配的函数的名字。
- match_dup ‘m’“i” 只出现在机器描述中。无论操作数表指定位置保存的是什么值，对它进行匹配测试。操作数 0 是操作数表的索引。
- match_op_dup ‘m’“iE” 只出现在机器描述中。它只对操作数表特定索引位置保存的数据进行匹配。操作数 0 是操作数表的索引。操作数 1 是进行比较的表达式向量。
- match_operand ‘m’“iss” 只出现在机器描述中。是两个操作数函数等价性的比较。操作数 0 是操作数表的索引。操作数 1 是用于执行比较的函数名。操作数 2 是比较的第一个操作数名字。操作数 3 是进行比较的第二个操作数名字。
- match_operator ‘m’“isE” 只出现在机器描述中。它对 RTX 的表达式操作数进行递归匹配。操作数 0 是操作数表的索引。操作数 1 是被调用进行匹配的函数名。操作数 2 是匹配的操作数向量。
- match_par_dup ‘m’“iE” 只出现在机器描述中。它只对操作数表特定索引位置保存的数据进行匹配。操作数 0 是操作数表的索引。操作数 1 是匹配的表达式向量。
- match_parallel ‘m’“isE” 只出现在机器描述中。通过调用指定函数，它和操作数向量相匹配——操作数向量是并行指令集。操作数 0 是操作数表的索引。操作数 1 是被调用进行匹配操作的函数名字。操作数 2 是匹配的并行指令向量。
- match_scratch ‘m’“is” 只出现在机器描述中。对 RTX 格式，见 scratch。为测试 scratch 寄存器进行的比较。操作数 0 是操作数表的索引。操作数 1 是被调用进行匹配操作的函数名字。
- mem ‘o’“e0” 内存位置。操作数 0 是内存位置的地址。操作数 1 是内存所属的别名集合。
- minus ‘2’“ee” 计算两个操作数，操作数 1 是从操作数 0 中截取的。

- mod ‘2’“ee”　计算两个操作数，操作数 1 除以操作数 0，结果是整数除法操作的余数。
- mult ‘1’“e”　计算两个操作数，再相乘得到结果。
- ne ‘<’“ee”　产生的代码计算表达式，并进行符号比较。如果第一个值不等于第二个值，结果为真。
- neg ‘1’“e”　计算表达式，结果为负。
- nil ‘x’“*”　空指针。
- not ‘1’“e”　计算表达式，执行按位 not 操作产生结果。
- note ‘x’“iuuB0ni”　指出代码中源代码行开始的位置。操作数 0 是这个 RTX 的惟一标识。操作数 1 是到链中上一条指令的指针。操作数 2 是到链中下一条指令的指针。操作数 3 是指令的基本块，后面跟着的是标记。如果行号大于零，操作数 4 是文件的名字；否则是专为本记录的数据。操作数 5 是行号。如果数值是零，它是 enum note_insn 的值。如果行号等于 note_insn_deleted_label，操作数 6 则是惟一的。
- ordered ‘<’“ee”　产生的代码计算表达式，按照顺序进行浮点比较。如果值为 NaN，排序比较会发出异常。如果值相等，结果为真。也见 unordered。
- parallel ‘x’“E”　两个或更多并行执行的操作数组。
- pc ‘o’“”　程序计数器。操作数 0 为 pc 时，由 set 语句设置跳转。
- phi ‘x’“E”　SSA phi 操作符，只出现在基本块的开始位置。操作数是 2n RTX 向量。元素 2n+1 是 const_int insn，具有前一个块的 ID 号，使用元素 2n 的时候，通过它传递控制。
- plus ‘c’“ee”　计算两个表达式，将结果加在一起。
- post_dec ‘a’“e”　通过表达式对指定内存的地址进行事后递减（postdecrementation）。没有指定递减的数目，因为它的类型和尺寸可由表达式确定。
- post_inc ‘a’“e”　通过表达式对指定内存的地址进行事后递加（postincrementation）。没有指定递加的数目，因为它的类型和尺寸可由表达式确定。
- post_modify ‘a’“ee”　代表普通地址副作用（除了递加和递减，它们由其他操作符处理）。也见 pre_modify。操作数 0 是 reg insn，它会用于定位。操作数 1 是给寄存器分配的表达式。操作数必须为形式 plus(reg)(reg)或 plus(reg)(const_int)，这里 plus 的第一个操作数和 post_modify 的第一个操作数一样。
- pre_dec‘a’“e”　通过表达式对指定内存的地址进行事前递减（predecrementation）。没有指定递减的数目，因为它的类型和尺寸可由表达式确定。
- pre_inc‘a’“e”　通过表达式对指定内存的地址进行事前递加（preincrementation）。没有指定递加的数目，因为它的类型和尺寸可由表达式确定。

- pre_modify ‘a’“ee” 代表普通地址副作用（除了递加和递减，它们由其他操作符处理）。操作数0是reg insn，它会用于定位。操作数1是给寄存器分配的表达式。操作数必须为形式plus(reg)(reg)或plus(reg)(const_int)，这里plus的第一个操作数和pre_modify的第一个操作数一样。
- prefetch ‘x’“eee” 内存预先获取（prefetch）一些目标机器支持的属性。对不支持它们的机器，会忽略操作数1和2。操作数0是内存地址，从此处获取内容。操作数1对读访问设置为0，对于写访问设置为1。操作数2是个数字，以便确定临时地点的级别，这里0指没有，1、2、3是指临时地点增加的级别。
- presence_set ‘x’“ss” 只出现在机器描述中，指出不能保留的CPU功能单元列表，除非确定也保留了其他的功能单元。也见absence_set。操作数0是功能单元列表，逗号分隔，除非至少也保留了操作数2中的一个功能单元，否则不会保留。操作数1是功能单元的列表，逗号分隔。
- queued ‘x’“eeeee” 指向指令序列成员的指针，因为事后递加或事后递减该指令稍后会有输出，这样顺序的insn不会变成产生代码中的一部分。排序的表达式会放在指令中，这样用到的值为递加或递减之前的值。操作数0是递加或递减的变量（或寄存器）。操作数1是执行递加或递减的insn。操作数2是reg RTX，包括变量的原始值。操作数3是递加或递减指令用到的体（body）。操作数4是序列中下一个queued表达式。
- range_info ‘x’“uuEiiiiiibbii” 范围信息头。操作数0是指针，指向note insn，标记范围的开始位置。操作数1是指针，指向note insn，标记范围的结束位置。操作数2是向量，包括该范围内所有可被取代的寄存器。操作数3是范围内调用的数目。操作数4是范围内insn的总数。操作数5是该范围的惟一标识。操作数6是范围开始的基本块数目。操作数7是范围结尾的基本块数目。操作数8是循环深度。操作数9是位图，指出范围开始位置哪个寄存器是活的。操作数10是指出范围结尾位置哪个寄存器是活的位图。操作数11是范围开始的标记数字。操作数12是范围结尾的标记数字。
- range_live ‘x’“bi” 这是关于该点存活寄存器的信息。操作数0是位图，代表存活寄存器的列表。操作数1是当前块的原始块数目。
- range_reg ‘x’“iiiiiiiitt” 指出范围内可被替换的寄存器。操作数0是原始的伪寄存器数目。操作数1是伪寄存器保存的值，为范围的区间。操作数2是引用范围内寄存器的数目。操作数3是范围内clobbered的寄存器次数。操作数4是寄存器在范围内死亡的次数。操作数5包括的标志指出是否要在范围开始位置将原始寄存器内的数据复制到新的寄存器中，以及是否需要在范围结束的位置再将数据从新寄存器复制回原始寄存器。操作数6是生存长度。操作数7是调用的次数，寄存器通过

该调用保持活跃状态。操作数 8 是变量的符号节点，如果寄存器为变量的话。操作数 9 是块节点，如果寄存器是变量，就在此声明变量。

- range_var ‘x’“eti”　这是关于局部变量范围的信息。操作数 0 是 expr_list，它包含在范围中的变量被复制到伪寄存器。操作数 1 是块，变量在此声明。操作数 2 是范围中变量出现的数目。
- reg ‘o’“i0”　这是硬件或伪寄存器。也见 scratch。 操作数 0 是寄存器数目。如果数目少于 FIRST_PSEUDO_REGISTER ，那就是硬件寄存器。操作数 1 是原始寄存器数目，它对伪寄存器的值不同，该寄存器会转换为硬件寄存器。
- resx ‘x’“i”　对可能的_Unwind_Resume 而言，在确定它是函数调用或所需分支之前，它是占位符。操作数 0 是异常区域，控制流从这里开始。
- return ‘x’“”　从子例程中返回。
- rotate ‘2’“ee”　向左移位，无符号扩展。每位会向左移动到头然后转回到右端点。操作数 0 是表达式，它产生的值会进行位移。操作数 1 是表达式，它产生的值是移动的位数。
- rotatert ‘2’“ee”　向右移位，每位会向右移动到头然后转回到左端点。操作数 0 是表达式，它产生的值会进行位移。操作数 1 是表达式，它产生的值是移动的位数。
- scratch ‘o’“0”　临时寄存器。该寄存器只用于单指令，在寄存器分配或重载过程中，它会变成 reg insn。指出它只有一个操作数，以便转换成 reg。
- sequence ‘x’“E”　这种形式的 insn 序列是产生代码的结果，基于 define_expand 生成大量的 insn。函数 emit_insn()将序列 sequence 拆成独立的 insn。操作数 0 是表达式数组。
- set ‘x’“ee”　在特定位置保存值的赋值操作。所有赋值操作必须使用 set。需要多次赋值的指令必须多次使用 set insn。操作数 0 是左值，是接受赋值的位置（内存、寄存器、条件代码、或者其他）。操作数 1 是右值，是值或值的地址，被保存在左值内。
- set_attr ‘x’“ss”　可作为 define_insn、define_peephole 和 define_asm_insn 的最后一个操作数，用于指出与模式相匹配的 insn 的属性值。操作数 0 是属性的名字。操作数 1 是属性的值。
- set_attr_alternative ‘x’“sE”　可被当作 define_insn 和 define_peephole 最后的操作数，为被赋的可能属性值集合。由匹配确定分配的是什么。操作数 0 是属性的名字。操作数 1 是可能的属性值数组。
- sign_extend ‘1’“e”　计算表达式的结果为符号扩展。符号扩展的数目由机器模式和表达式类型确定。也见 zero_extend。

- sign_extract ‘b’“eee” 这是符号位字段的尺寸和位置的规范。也见 zero_extract。操作数 0 是内存单元，包括位字段中的第一位。操作数 1 是字段中位的数目。操作数 2 是到位字段的偏移量，它是内存单元位字段中第一位之前的位的数目。如果设置了 BITS_BIG_ENDIAN ，该数目就是内存单元最有效的位；否则，数目为最小有效的位。
- smax ‘c’“ee” 符号比较，结果是两个表达式中最大的那个。
- smin ‘c’“ee” 符号比较，结果是两个表达式中最小的那个。
- sqrt ‘1’“e” 计算出表达式的结果，再取其平方根。
- ss_minus ‘2’“ee” 计算两个表达式，结果是操作数 1 减去操作数 0，符号饱和（signed saturation）。也见 us_minus。
- ss_plus ‘c’“ee” 计算两个表达式，结果相加，符号饱和。也见 us_plus。
- ss_truncate ‘1’“e” 计算表达式，进行符号饱和截取。也见 us_truncate。
- strict_low_part ‘x’“e” 赋值，只修改了目标中的最小部分。操作数 0 是赋值表达式，它有 strict_low_part 限制。
- subreg ‘x’“ei” 多词值中的一个词。操作数 0 是表达式，包括完整值。操作数 1 是多词值中词的选择子。
- symbol_ref ‘o’“s” 对指定标记的引用。也见 label_ref。操作数 0 是标记串，以下划线开头。如果标记名以星号开头，会去掉星号，而且也不会加下划线。
- trap_if ‘x’“ee” 条件陷阱（trap）。对非条件陷阱，条件表达式设置为 1。操作数 0 是条件表达式。操作数 1 是在条件表达式为非零值时执行的代码。
- truncate ‘1’“e” 表达式结果被简单截断保存在收到的位置中。
- udiv ‘2’“ee” 计算两个表达式，操作数 1 除以操作数 0，执行无符号整数除法。
- unordered ‘<’“ee” 生成的代码计算表达式值，进行无序浮点比较。如果值不是 NaN，无序比较不会发出异常处理。如果值相等，结果为真。也见 ordered。
- umax ‘c’“ee” 无符号比较操作，结果是两个表达式中的最大值。
- umin ‘c’“ee” 无符号比较操作，结果是两个表达式中的最小值。
- umod ‘2’“ee” 计算两个表达式，操作数 1 除以操作数 0，结果是无符号整数除法的余数。
- uneq ‘<’“ee” 产生的代码计算表达式，执行无序浮点比较。如果第一个值等于第二个，结果为真。
- unge ‘<’“ee” 产生的代码计算表达式，执行无序浮点比较。如果第一个值大于等于第二个，结果为真。
- ungt ‘<’“ee” 产生的代码计算表达式，执行无序浮点比较。如果第一个值大于第二个，结果为真。

- unle ‘<’“ee”　产生的代码计算表达式，执行无序浮点比较。如果第一个值小于等于第二个，结果为真。
- unlt ‘<’“ee”　产生的代码计算表达式，执行无序浮点比较。如果第一个值小于第二个，结果为真。
- UnKnown ‘x’“*”　RTX 是未知类型。
- unsigned_fix ‘1’“e”　将浮点表达式的值转换成无符号定点值。也见 unsigned_fix。
- unsigned_float‘1’“e”　将无符号浮点表达式的值转换成浮点值。也见 unsigned_fix。
- unspec ‘x’“Ei”　机器有关的操作。操作数 0 是操作数向量，用于机器相关操作。操作数 1 是操作数索引，指出用的是哪个操作数。
- unspec_volatile ‘x’“Ei”　机器有关的操作，其中可能有陷阱。操作数 0 是操作数向量，被机器相关的操作使用。操作数 1 是操作数索引，指出用的是哪个操作数。
- us_minus ‘2’“ee”　计算两个表达式值，结果是操作数 0 减去操作数 1，无符号饱和。也见 ss_minus。
- us_plus ‘c’“ee”　计算两个表达式值，结果是两者之和，无符号饱和。也见 ss_plus。
- us_truncate ‘1’“e”　计算表达式，执行无符号饱和截取。也见 ss_truncate。
- use ‘x’“e”　指出有些事情的用法无需解释。例如，子例程调用使用寄存器作为调用序列的一部分。也见 clobber。操作数 0 为表达式，指出所用项目。
- value ‘o’“0”　cselib 子例程使用，描述值。
- vec_concat ‘x’“ee”　定义两个向量的串联。结果是两个向量组合的向量长度，操作数 0 向量在操作数 1 向量之前。操作数 0 是串联的第一个向量。操作数 1 是串联的第二个向量。
- vec_duplicate ‘x’“e”　定义操作，通过复制向量的所有成员，成倍扩大向量的尺寸。结果向量是输入向量尺寸的整数倍。
- vec_merge ‘x’“eee”　定义两个向量的合并操作。操作数 0 是合并的第一个向量。操作数 1 是合并的第二个向量。操作数 2 是位掩码，指出结果向量的各个部分要展开到哪里。0 位指出操作数 0 中的成员，1 位是指操作数 1 中的成员。
- vec_select ‘x’“ee”　定义操作选择向量的部分。操作数 0 是源向量。操作数 1 是包括常数值的并行 insn，指出源向量中哪个成员被保存在结果向量中。
- xor ‘c’“ee”　计算两个操作数，结果为位宽，是两个值的或（OR）操作。
- zero_extend ‘1’“e”　计算表达式结果，无符号扩展，填入的是零。也见 sign_extend。
- zero_extract ‘b’“eee”　无符号位字段的尺寸和位置的规范。也见 sign_extract。操作数 0 是内存单位，包括位字段中的第一位。操作数 1 是字段中位的数目。操作数 2 是位字段中的偏移量，它是内存单元位字段中第一位前的位的数目。如果设置了

BITS_BIG_ENDIAN，数目来自内存单元最大的位；否则，数目来自最小位。

## 20.2 模式和模式类

每个 RTX 表达式都有模式，描述它产生数据的尺寸和类型。两个一样的表达式模式不同的话，可以产生完全不同的代码。一个例子就是浮点表达式和整数表达式的比较。表 20-5 列出的模式定义了 machmode.def 中的枚举类型集合。

表 20-5 用于表达式的机器模式

| 模式 | 描述 |
|---|---|
| BImode | 位模式。指明对单个位的操作 |
| QImode | 四分之一整数模式。指出对单字节的操作，可看作一个整数值 |
| HImode | 半整数模式。指出对双字节的操作，可看作一个整数值 |
| PSImode | 部分单整数模式。指出对整数的操作，该整数占 4 个字节但并没有实际用到这 4 个字节 |
| SImode | 单整数模式。指出对 4 字节整数值的操作 |
| PDImode | 部分双整数模式。指出对整数的操作，该整数占 8 个字节但并没有实际用到这 8 个字节 |
| DImode | 双整数模式。指出对 8 字节整数值的操作 |
| TImode | 四整数模式。指出对 16 字节整数值的操作 |
| OImode | 八整数模式。指出对 32 字节整数值的操作 |
| QFmode | 四分之一浮点模式。指出对单字节浮点数的操作 |
| HFmode | 半浮点模式。指出对双字节浮点数的操作 |
| TQFmode | 四分之三浮点模式。指出对 3 字节浮点数的操作 |
| SFmode | 单浮点模式。指出对 4 字节浮点数的操作。通常 IEEE 的单精度浮点操作是指 8 位字节，但和 16 位字节机器上的有所区别，在其他硬件上有它自己的浮点算术格式 |
| DFmode | 双浮点模式。指出对 8 字节浮点数的操作。通常 IEEE 的双精度浮点操作是指 8 位字节，但和 16 位字节机器上的有所区别，在其他硬件上有它自己的浮点算术格式 |
| XFmode | 扩展浮点模式。指出 12 字节浮点数的操作。通常 IEEE 的扩展精度浮点操作是指 8 位字节。有些系统用到的少于完整的 12 字节 |
| TFmode | 四浮点模式。指出 16 字节浮点数的操作。既用于填充到 128 位的 96 位扩展 IEEE 浮点类型，也可用于真的 128 位扩展 IEEE 浮点类型 |
| CCmode | 条件代码。指出对条件代码值的操作。有些机器指定的硬件位的集合，保存的是比较操作的结果。该模式不会用于使用 cc0 insn 的机器 |
| BLKmode | 块模式。指出集合值的操作，它们不能应用其他模式。在 RTL 中，这种模式只能用于向量硬件指令的内存引用。该模式不会用于没有这种指令的机器 |

（续表）

| 模式 | 描述 |
|---|---|
| VOIDmode | 空模式。在没有指定任何模式的时候使用它。常数表达式可有这种模式，因为它们可被内容规定的任何模式取代 |
| QCmode | 四分之一复杂模式。指的是两个 QFmode 浮点值的复杂数比较 |
| HCmode | 半复杂模式。指的是两个 HFmode 浮点值的复杂数比较 |
| SCmode | 单复杂模式。指的是两个 SFmode 浮点值的复杂数比较 |
| DCmode | 双复杂模式。指的是两个 DFmode 浮点值的复杂数比较 |
| XCmode | 扩展复杂模式。指的是两个 XFmode 浮点值的复杂数比较 |
| TCmode | 四复杂模式。指的是两个 TFmode 浮点值的复杂数比较 |
| CQImode | 四分之一整数复杂模式。指的是两个 QImode 整数值的复杂数比较 |
| CHImode | 半整数复杂模式。指的是两个 HImode 整数值的复杂数比较 |
| CSImode | 单整数复杂模式。指的是两个 SImode 整数值的复杂数比较 |
| CDImode | 双整数复杂模式。指的是两个 DImode 整数值的复杂数比较 |
| CTImode | 四整数复杂模式。指的是两个 TImode 整数值的复杂数比较 |
| COImode | 八整数复杂模式。指的是两个 OImode 整数值的复杂数比较 |

在 RTL 调试输出中，以及在机器描述中，操作模式的名字列成一列，紧跟着表达式。例如，寄存器表达式可写成（reg:SI 7 esp），或在设置了寄存器标志时为（reg/f:SI 7 esp）。模式名字总是写成词“mode”，从中得到修整。如果在输出中没有出现模式，则为 VOIDmode。

并不是所有机器都能支持所有模式。机器只需要支持 QImode（单字节整数）、和 BITS_PER_WORD 定义尺寸相匹配的整数类型，以及 FLOAT_TYPE_SIZE 和 DOUBLE_TYPE_SIZE 定义的浮点尺寸。这些在配置过程中由编译程序定义的常数值和目标机器的尺寸相匹配。

尽管模式设置显式地用于 RTL 代码，编译程序自身经常会用到模式类的引用。表 20-6 描述的模式类在 machmode.h 中定义为 enum 值。

表 20-6　模式类的名字及其说明

| 模式类 | 描述 |
|---|---|
| MODE_INT | 整数模式类，由 BImode、QUmode、HImode、SImode、DImode、TImode 和 OImode 组成 |
| MODE_PARTIAL_INT | 部分整数模式类，由 PQImode、PHImode、PSImode 和 PDImode 组成 |
| MODE_FLOAT | 浮点模式类，由 QFmode、HFmode、TQFmode、SFmode、DFmode、XFmode 和 TFmode 组成 |

（续表）

| 模式类 | 描述 |
| --- | --- |
| MODE_COMPLEX_INT | 整数对组成的复杂数字类，包括 CQImode、CHImode、CSImode、CDImode、CTImode 和 COImode |
| MODE_COMPLEX_FLOAT | 浮点数对组成的复杂数字类，包括 QCmode、HCmode、SCmode、DCmode、XCmode 和 Tcmode |
| MODE_CC | 条件代码类，由 CCmode 和其他由宏 EXTRA_CC_MODES 定义的类型组成 |
| MODE_RANDOM | 随机类，用于表示所有不是其他类的模式，例如 BLKmode 和 VOIDmode |

名为 byte_mode 的全局变量包括和目标机器的 BITS_PER_UNIT（在 32 位机器中，它被设为 QImode）值相适应的模式。全局变量 word_mode 包括和目标机器的 BITS_PER_WORD（在 32 位机器中，为 SImode）值相适应的模式。

## 20.3 标志

RTL 代码中，每个 insn 都包含大量的标志。在 RTL 的打印模式下，每个标志都是单字符，以反斜线开头，如第 18 章所示。标志的确切含义依赖于 insn 的类型。表 20-7 包括 insn 的列表，也可以设置标志，这些标志都显示在 RTL 的输出中。

表 20-7 不同 insns 的标志指示字含义

| insn | 标志 | 标志含义 |
| --- | --- | --- |
| asm_input | /v | 引用内存位置的数据为易变的。insn 不能被删除、重新排序或组合 |
| asm_operands | /v | 引用内存位置的数据为易变的。insn 不能被删除、重新排序或组合 |
| call_insn | /j | 同属调用 |
| call_insn | /u | 该调用为纯函数或常数 |
| code_label | /s | 该标记是非局部跳转的目标，不能被删除。如果删除这样的标记，必须由某个条目取代，以指出该符号已被删除 |
| const | /i | 由过程集成生成的 RTL |
| expr_list | /u | 对纯函数或常数的调用 |
| insn | /f | 函数序言的一部分，它设置了栈指针。这是框架指针的一部分，保存寄存器，或设置一个临时寄存器用以取代该框架指针 |
| insn | /i | 由过程集成生成的 RTL |

（续表）

| insn | 标志 | 标志含义 |
|---|---|---|
| insn | /s | 在清除不可达代码过程中，该标志指出是否为不可达代码。在分支的延迟重组过程中，该标志指出这个 insn 来自分支目标。在调度过程中，该标志指出这个 insn 必须和以前的 insn 一起调度 |
| insn | /v | 该 insn 已被删除 |
| insn_list | /i | 由过程集成生成的 RTL |
| jump_insn | /s | 在分支的延迟重组过程中，该标志指出这个 insn 来自分支目标 |
| label_ref | /s | 被引用的标记，处于包括 insn 的最里层循环中 |
| label_ref | /v | 非局部标记的引用 |
| mem | /f | 该引用是标量 |
| mem | /j | 访问内存组件的时候，不应改动该内存引用的别名集合 |
| mem | /s | 所引用的内存是聚集的一部分（结构或数组）。如果引用涉及 C 指针，不会设置该标志，可为标量或聚集 |
| mem | /u | 此处内存的值从未改动过 |
| mem | /v | 此处内存的数据是不稳定的。insn 不能被删除、重新排序或组合 |
| note | /u | 对纯函数和常数的调用 |
| reg | /f | 包含指针的寄存器 |
| reg | /i | 该寄存器包含会被当前函数返回的值。如果机器中预设有寄存器用来返回值，则不会设置该标志 |
| reg | /s | 包含在测试循环表达式中的寄存器内容有自己的生命周期 |
| reg | /u | 寄存器中的内容从未改变 |
| reg | /v | 该标志指出用户定义的变量；否则，这是个由编译程序创建的临时变量 |
| reg_label | /v | 到非局部标记的引用 |
| set | /f | 函数序言的一部分，它设置了栈指针。建立了框架指针，保存寄存器，或设置了临时寄存器用来代替框架指针 |
| set | /j | 设置用来返回的值 |
| subreg | /s | 访问的对象，它自己提供了更宽的模式 |
| subreg | /u | 到非符号值的引用，它自己的模式为更宽的类型 |
| symbol_ref | /f | 该引用定位了函数的字符串常数池 |
| symbol_ref | /i | 引用的符号是弱符号 |
| symbol_ref | /u | 到当前函数常数池内部的引用 |
| symbol_ref | /v | 该标志用于指定机器 |

# 第 21 章

# 机器相关的编译程序选项

本章包含的信息可以在某特定平台下编译程序时使用。大多数选项和设置都可用来调整代码的生成，以便利用（或不利用）硬件的某些特殊功能，但也有一些选项用于调试和组织目标文件中的不同代码部分。

## 21.1 机器列表

GCC 的 3.1 版包括以下机器描述文件。这些文件可在 GCC 源代码发布的 config 目录中找到：

alpha/alpha.md
alpha/ev4.md
alpha/ev5.md
alpha/ev6.md
arc/arc.md
arm/arm.md
avr/avr.md
c4x/c4x.md
cris/cris.md
d30v/d30v.md
dsp16xx/dsp16xx.md
fr30/fr30.md
h8300/h8300.md
i370/i370.md
i386/i386.md
i386/athlon.md
m88k/m88k.md
mcore/mcore.md
mips/mips.md
mmix/mmix.md
mn10200/mn10200.md
mn10300/mn10300.md
ns32k/ns32k.md
pa/pa.md
pdp11/pdp11.md
romp/romp.md
rs6000/rs6000.md
s390/s390.md
sh/sh.md
sparc/cypress.md
sparc/sparc.md
sparc/hypersparc.md

i386/k6.md
i386/pentium.md
i386/ppro.md
i960/i960.md
ia64/ia64.md
m32r/m32r.md
m68hc11/m68hc11.md
m68k/m68k.md
sparc/sparclet.md
sparc/supersparc.md
sparc/ultra1_2.md
sparc/ultra3.md
stormy16/stormy16.md
v850/v850.md
vax/vax.md

每个目录对应一种平台。由于 include 指示字允许将完整的配置组织成一组文件集合，有些目录会包括多个.md 文件。例如，alpha 目录的 alpha.md 文件包含文件 ev4.md、ev5.md 和 ev6.md，而且在 i386 目录中的 i386.md 文件包含 pentium.md、ppro.md、k6.md 和 athlon.md。

## 21.2　GCC 命令行选项

某些（但不是所有）现有的端口均有某些特殊的命令行选项，它们用来告诉编译程序如何进一步精简所产生的代码，以匹配特殊的硬件模式和运行时配置。以下章节列出定义它们的端口可用的-m 选项。

大多数选项以-m 开头，但有少数例外。有些用于指定为 CPU 家族中特殊的 CPU 产生代码，而其他的用于产生代码，该代码可利用一些硬件功能以便将程序的特征和硬件更好地结合起来。某些选项则生成和特殊硬件配置相匹配的代码。

### 21.2.1　Alpha 选项

如下选项是为 DEC Alpha 实现定义的。

-malpha-as

产生的代码被供应商提供的汇编器所编译。参见-mgas。

-mcix

指定编译程序使用可选的 CIX 指令集产生代码。默认利用-mcpu 包含的指令集。CIX 指令集代码可通过-mno-cix 去除。参见-mbwx、-mfix 和-mmax。

-mbuild-constants

该选项需要编译程序使用代码构造所有整数常量，甚至可能使用更多的指令（最多 6 步）。

通常 GCC 会检查 32 位或 64 位整数常量，看它是否是由两三条指令中更小的常量构造出来的。如果不是，会输出文字常量并生成代码，在运行时从数据片断载入。该选项典型

地用于构造共享库动态载入器。共享库必须能够在自己的数据片断中找到变量和常量之前从内存中重定位自己。

-mbwx

指定编译程序使用可选的BWX 指令集产生代码。默认会用-mcpu 包含的指令集。BWX 指令集代码可通过-mno-bwx 去除。参见-mcix、-mfix 和-mmax。

-mcpu=type

设置指令集和指定机器类型的指令调度参数。可以指定 EV 风格名或相应的芯片号。GCC 支持处理器的 EV4、EV5 和 EV6 系列的调度参数，从指定的处理器中选择指令集的默认值。如果没有指定处理器类型，GCC 会默认使用构造编译程序的处理器。类型 type 的有效选择列在表 21-1 中。

表 21-1 DEC Alpha 机器的 CPU 类型选择

| 类型 | 描述 |
|---|---|
| ev4, ev45, 21064 | EV4 调控器，没有设置扩展的指令 |
| ev5, 21164 | EV5 调控器，没有设置扩展的指令 |
| ev56, 21164a | EV5 调控器，支持 BWX 扩展 |
| pca56, 21164pc, 21164pc | EV5 调控器，支持 BWX 和 MAX 扩展 |
| ev6, 21264 | EV6 调控器，支持 BWX、FIX 和 MAX 扩展 |
| ev67, 21264a | EV6 调控器，支持 BWX、CIX、FIX 和 MAX 扩展 |

-mexplicit-relocs

产生显式的符号重定位信息。

老的 Alpha 汇编器除非是通过编译程序宏，否则不能产生符号重定位。这些宏不允许优化指令调度。GNU binutils 支持新的语法，允许编译程序显式地标记哪个重定位会应用于该指令。此选项对调试最有用，因为在构造和设置默认值时，GCC 可检测到汇编器的能力。可通过-mno-explicit-relocs 禁止生成符号重定位信息。

参见-msmall-data 和-mlarge-data。

-mfix

指定编译程序使用可选的 FIX 指令集产生代码。默认使用-mcpu 包含的指令集。FIX 指令集代码可通过-mno-fix 去除。参见-mbwx、-mcix 和-mmax。

-mfloat-ieee

使用 IEEE 单精度和双精度而不是 VAX F 和 G 浮点算术来产生代码。参见-mfloat-vax。

-mfloat-vax

使用 VAX F 和 G 浮点算术而不是 IEEE 单精度和双精度来产生代码。参见-mfloat-ieee。

-mfp-reg

使用浮点寄存器集产生代码。这是默认的。

选项-mno-fp-regs 没有使用浮点寄存器集，它设置选项-msoft-float。如果不使用浮点寄存器集，浮点操作数会当作整数传给整数寄存器，而浮点结果会传给$0（而不是$f0）。这是非标准调用序列，因此任何具有浮点参数或返回值的函数被代码调用，代码是按照-mno-fp-regs 编译的，函数必须也按照-mno-fp-regs 进行编译。

-mno-fp-regs 的典型用法是在编译内核的时候，而编译内核是用不到任何浮点寄存器的，因此也不会保存和恢复浮点寄存器。

-mfp-rounding-mode=mode

选择 IEEE 取整模式。其他的 Alpha 编译程序会调用该选项-fprm mode。表 21-2 列出模式 mode 的所有可能值。

**表 21-2　浮点取整模式**

| 模式 | 描述 |
|---|---|
| n | 浮点数向最近的机器数取整，或向偶机器数取整，以防打结。这是默认的 |
| m | 向负无穷取整 |
| c | 截断取整模式。浮点数向零取整 |
| d | 动态取整模式。浮点控制寄存器（fpcr）中的字段，可以控制取整效果。C 库初始化该寄存器为向正无穷取整。因此，除非你的程序修改了该寄存器，这种模式会向正无穷取整 |

-mfp-trap-mode=mode

该选项控制激活哪个浮点相关的陷阱。其他 Alpha 编译程序调用该选项-fptm mode。表 21-3 列出 mode 的所有可能值。

**表 21-3　浮点陷阱模式设置**

| 模式 | 描述 |
|---|---|
| n | 软件中唯一一个只能使能不能终止的陷阱（例如，除零陷阱）。这是默认的 |
| u | 除了由 n 使能的陷阱，下浮陷阱也是使能的 |
| su | 该指令标志软件安全完成 |
| sui | 类似 su，但不精确陷阱也同时使能了 |

-mgas

通过 GNU 汇编器产生汇编代码。参见-malpha-as。

-mieee

产生的代码和 IEEE 完全兼容，除了不用维护 inexact-flag。Alpha 结构实现了浮点硬件，优化为最高性能，最有可能和 IEEE 浮点标准完全兼容。但是，为了完全兼容，需要软件协助。

使用该选项，在编译的时候定义预处理宏_IEEE_FP。结果代码效率较低，但可正确地支持非标准化数字和异常 IEEE 值，例如 NaN（not-a-number）和无限的加减。其他的 Alpha 编译程序调用选项-ieee_with_no_inexact。

参见-mieee-with-inexact。

-mieee-conformant

产生 IEEE 结构代码。

该选项不可使用，除非你也指定-mtrap-precision=i，并指定-mfp-trap-mode=su 或-mfp-trap-mode=sui。

该选项的惟一影响是产生的汇编文件中的函数前言忽略了行.eflag 48。在 DEC UNIX 中，它的影响是会连接 IEEE 格式的数学库例程。

-mieee-with-inexact

同-mieee 一样，只不过它产生的代码也维护 IEEE 的 inexact-flag。该选项令所产生的代码实现与 IEEE 完全兼容的算术操作。

除了定义_IEEE_FP，也定义了预处理程序宏_IEEE_FP_EXACT。有些 Alpha 实现的结果代码可能比默认产生的代码要慢很多。因为很少代码依赖于 inexact-flag，通常不会指定该选项。其他的 Alpha 编译程序调用选项-ieee_with_inexact。

-mlarge-data

使用该选项，数据域限制在 2GB 之下。需要数据大于 2GB 的程序使用 malloc() 或 mmap()在堆中分配数据，而不是在程序的数据段中分配数据。这是默认的。

当为共享库生成代码时，-fpic 设置-msmall-data，-fPIC 设置-mlarge-data。

-mmax

指定编译程序用可选的 MAX 指令集产生代码。默认使用-mcpu 包含的指令集。MAX 指令集代码可通过-mno-max 去除。参见-mbwx、-mcix 和-mfix。

-mmemory-latency=duration

设置延迟，调度器会假设应用程序最常用的典型存储访问。这个数值高度依赖于应用程序所使用的存储访问的模式以及机器所具有的外部高速缓存的大小。

可以用十进制数值指定持续时间，也就是时钟的周期数。

也可以将持续时间指定为 L1，L2，L3 或 main。编译程序包含了典型的 EV4 和 EV5 硬件的 1，2 和 3 级高速缓存（也称为 Dcache，Scache 和 Bcache），以及主存的时钟数。注意，只有 EV5 才有 L3。

-msmall-data

当选项-mexplicit-relocs 生效时，将通过 gp-relative 重定位访问静态数据。该选项指定放置在小数据区（.sdata 和.sbss 区）上的为 8 字节长的对象或更小的对象，并通过基于$gp 寄存器的 16 位重定位方式进行访问。这限制了小数据区的尺寸最大为 64KB，但允许使用一条指令直接访问变量。

当生成共享库代码时，-fpic 设置-msmall-data 而-fPIC 设置-mlarge-data。

-msoft-float

指示编译程序不要使用硬件浮点运算指令进行浮点操作。将使用 libgcc.a 中的函数完成浮点操作。

如果没有被相应的仿真浮点操作的例程替代或是编译成调用这些仿真例程的方式，这些例程将使用硬件浮点操作。如果为没有浮点硬件的 Alpha 编译代码，你必须确保所需要的库是内建的。没有浮点操作的 Alpha 实现仍然保留了浮点寄存器。

默认选项是-mno-soft-float。

-mtrap-precision=precision

设置浮点陷阱中的精度。

在 Alpha 体系结构中，浮点陷阱是非精确的。这意味着，如果没有软件的辅助，不可能从一次浮点陷阱中恢复，而且通常都需要终止程序的执行。编译程序将产生可以辅助操作系统陷阱处理函数的代码，以确定引起浮点陷阱的确切位置。

可以选择表 21-4 中列出的不同的精度级别。

**表 21-4　浮点陷阱精度设置**

| 精度级别 | 描述 |
|---|---|
| P | 程序精度。该选项是指陷阱句柄只能指出哪个程序造成异常的浮点操作。这是默认的 |
| f | 函数精度。陷阱句柄能够确定是哪个函数造成异常的浮点操作 |
| i | 指令精度。陷阱句柄能够准确确定是哪条指令造成异常的浮点操作 |

其他的 Alpha 编译程序提供类似的选项，它们是-scope_safe 和-resumption_safe。

-mtune=type

只为某一特定机器类型设置指令调度参数。指令集没有改变。表 21-1 中列出了 type 的合法选择。

### 21.2.2 Alpha/VMS 选项

针对 DEC Alpha/VMS 实现定义了如下选项。

-mvms-return-codes
从 main()中返回一个 VMS 条件代码。默认设置是返回一个 POSIX 条件代码。

### 21.2.3 ARC 选项

针对 ARC 实现定义了如下选项：

-mcpu=cpu
为指定的 ARC cpu 生成代码。
支持哪一 CPU 变体取决于配置。所有的变体均支持选项-mcpu=base，这是默认选项。

-mdata=section
将数据放置在指定的代码区中。该选项可以通过在代码中使用 section 属性重新定义。参见选项-mtext 和-mrodata。

-mmangle-cpu
在所有的公共符号名前追加 CPU 的名字。
在多处理机系统中，存在许多具有不同指令集和寄存器集的 ARC 变体。该选项防止为某一 CPU 编译的代码链接到为其他处理器而生成的代码上。目前还没有工具处理那些彼此非常相似的变体。

-mrodata=section
将只读数据放在指定的代码区中。该选项可以通过在代码中使用 section 属性重新定义。参见选项-mtext 和-mdata。

-mtext=section
将函数放在指定的代码区中。该选项可以通过在代码中使用 section 属性重新定义。参见选项-mdata 和-mrodata。

-EL
产生 little endian 模式代码。这是默认选项。

-EB
产生 big endian 模式代码。

### 21.2.4 ARM 选项

以下选项均是为 Advanced RISC Machines（ARM）体系结构定义的。

-mabort-on-noreturn

在 noreturn 函数的结尾处生成调用函数 abort()的代码。放置该调用可以在函数试图返回时被执行。

-malignment-traps

生成的代码即便在 MMU 对齐陷阱打开的情况下仍然不会陷入陷阱。所生成的代码是一系列字节访问代码，而非一个直接的半字访问。当为 ARM 体系 4 和之后的体系编译时将忽略该选项，这是因为这些处理器均能够直接访问内存中的半字。

选项-mno-alignment-traps 的规格所产生的代码将假设 MMU 不会陷入非对齐访问。当目标指令集没有半字内存操作时，该选项可以生成更好的代码。注意，不能使用该选项访问非对齐字对象，因为处理机只会从内存中存取 32 位对齐对象。

大多数目标机的默认设置是-mno-alignment-traps。当不存在半字内存指令时，该选项可以生成更好的代码。

在 ARMv4 之前的 ARM 体系中，没有访问内存中半字对象的指令。然而，当由内存读取数据时，ARM 体系中的一种功能可以允许即使在地址非对齐的情况下仍然可以使用字装载，之后处理器核心将以装载的数据进行旋转。选项-malignment-traps 通知编译程序这种非对齐访问会引起 MMU 陷阱，应以一系列字节访问替代它。如果编译程序知道地址在字边界上对齐时，它还会使用字访问指令装载半字数据。

-mapcs

与选项-mapcs-frame 相同。

-mapcs-26

生成适用于具有 26 位程序计数器的处理器的代码，并且遵从 APCS 26 位选项的函数调用标准。

选项-mapcs-26 替代选项-m2 和-m3，它们是以往版本编译程序的选项。

-mapcs-32

生成适用于具有 32 位程序计数器的处理器的代码，并且遵从 APCS 32 位选项的函数调用标准。

选项-mapcs-32 替代选项-m6，它是以往版本编译程序的选项。

-mapcs-frame

即便对代码的正确执行没有必要，也为所有函数生成符合 ARM 过程调用标准的堆栈框架。默认为-mno-apcs-frame。

与该选项同时使用的选项-fomit-frame-pointer 将会引起无法为叶子函数生成堆栈框架。

-march=name

该选项指定目标 ARM 体系的名字。GCC 使用这一名字来确定在生成汇编代码时哪些指令可以使用。该选项可以同选项-mcpu 一同使用或单独使用，可以识别的 name 选项有 armv2，armv2a，armv3，>armv3m，armv4，armv4t，armv5，armv5t 和 armv5te。

-mbig-endian

生成适用于运行 big endian 模式处理器的代码，默认选项是为 little endian 处理器编译代码。参见选项-mwords-little-endian 和-mlittle-endian。

-mcallee-super-interworking

为正在编译的文件中的所有外部可见函数指出一个 ARM 指令集头，它可以在执行函数的其他部分前切换到 Thumb 模式。这一配置可以允许非交互工作的代码调用这些函数。

-mcaller-super-interworking

无论目标代码是否编译为交互工作方式，允许通过函数指针（包括虚函数）的函数调用正确执行。如果启用了该选项会带来一定的开销，即执行函数指针的代价。

-mbsd

该选项只能用于 RISC iX 仿真 BSD 模式编译程序。如果没有使用选项-ansi 这将是默认选项。参见选项-mxopen。

-mcpu=name

该选项用来指定目标 ARM 处理器的名字。GCC 使用该名字确定在生成汇编代码时使用哪种指令。可以识别的名字有 arm2，arm250，arm3，arm6，arm60，arm600，arm610，arm620，arm7，arm7m，arm7d，arm7dm，arm7di，arm7dmi，arm70，arm700，arm700i，arm710，arm710c，arm7100，arm7500，arm7500fe，arm7tdmi，arm8，strongarm，strongarm110，strongarm1100，arm8，arm810，arm9，arm9e，arm920，arm920t，arm940t，arm9tdmi，arm10tdmi，arm1020t 和 xscale。

参见选项-mtune。

-mfp=number

参见选项-mfpe-number。

-mfpe=number

该选项指定在目标机上可用的浮点仿真的版本。可以使用的值有 2 和 3。

-mhard-float

生成包含浮点操作的代码。这是默认选项。见选项-msoft-float。

-mlittle-endian

生成适用于运行 little endian 模式处理器的代码。参见选项-mbig-endian。

-mlong-calls

该选项将生成执行函数调用的代码，这种函数调用首先将函数地址加载到一个寄存器中，然后根据保存在该寄存器中的地址进行调用。

默认行为是选项-mno-long-calls，它具有在指令#pragma long_calls_off 的范围内放置函数调用的效果。

这些选项设置不会对编译程序如何产生代码进行处理，通过函数指针进行的函数调用不会受到影响。

如果目标函数在子例程调用指令的 64MB 偏移之外就需要该选项。即使设置了该选项，某些函数也不会转换为长调用。这些函数包括静态函数，具有 short-call 属性的函数，在#pragma no_long_calls 指令范围内的函数，以及其定义已在当前编译单元内完成编译的函数。相反，具有弱定义的函数，具有 long-call 属性的函数，在具有 long-call 属性区中的函数，以及在#pragma long_calls 指令范围内的函数将始终被转换为长调用。

-mnop-fun-dllimport

禁止 dllimport 属性的支持。

-mpic-register=reg

指定用于 PIC 寻址的寄存器。当 R9 已使用时，如果没有打开栈检查，默认寄存器是 R10。参见选项-msingle-pic-base。

-mpoke-function-name

将每一个函数的名字写入文本区，直接放在函数起始段之前。生成的代码类似如下：

```
    .ascii "arm_poke_function_name", 0
    .align
t1
    .word 0xff000000 + (t1 - t0)
arm_poke_function_name
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
```

在回溯栈期间，代码可以检查存储在 fp+0 处的程序计数器的数值。如果回溯成功，则查看位于 pc-12 的单元并查看是否设置了高 8 位，然后就可以知道在这个单元前嵌入的函数名了。名字的长度是((pc[-3]) & 0xff000000)。

-msched-prolog

这是默认选项。使用选项-mno-sched-prolog 将防止函数前导段指令的乱序或是与函数体内的指令合并。

当使用选项-mno-sched-prolog 时，生成的代码将使所有的函数均以一段更易理解的指令起始。这一事实可以用来定位可执行代码中的函数起始位置。

-mshort-load-bytes

选项-malignment-traps 的废弃格式。

-mshort-load-words

选项-malignment-traps 的废弃格式。

-msingle-pic-base

按只读方式处理用于 PIC 寻址的寄存器，而不是在每个函数的前导段加载该寄存器。运行时系统负责在程序执行前用恰当的值初始化这些寄存器。参见选项-mpic-register。

-msoft-float

生成包含浮点操作的库调用。默认选项是-mhard-float。

应该了解到并不是所有的 ARM 目标均有所需的函数库。通常使用目标机器的 C 编译程序的函数库，但在交叉编译时却没有。如果使用这一选项，还需提供自己的函数库。

该选项改变代码中的调用规则，因此仅当使用该选项编译整个程序时才有用。为了这样做，你需要使用选项-msoft-float 编译 libgcc.a（GCC 自带的库）。

-mstructure-size-boundary=number

所有结构和联合的尺寸均被取整到由该选项所指定的位数的整数倍。可以使用的值有 8 和 32。指定较大的值可以生成更快和更高效的代码，但也会增大程序的尺寸。

对于不同的工具链，默认值也有所不同。对于目标为 COFF 的工具链，默认值是 8。

这两个值隐含状态下是不兼容的。如果使用结构或联合交互信息，不能期望用一个值编译的代码能够和用其他值编译的代码或是库正常工作。

-msymrename

这是默认选项。在 RISC iX 上，该选项仅能以选项-mno-symrename 形式使用，它将禁止在汇编代码后运行汇编器后处理器 symrename。通常在连接到 RISC iX C 库的时候，不需要修改一些预准备的标准符号。指定-mno-symrename 会禁止它的通过。

当编译程序构建用于交叉编译的时候，从不会运行后处理器。

-mthumb

生成 16 位 Thumb 指令集代码。默认将使用 32 位 ARM 指令集。

-mthumb-interwork

生成的代码支持在 ARM 和 Thumb 指令集之间调用。如果没有该选项，这两种指令集将不能可靠地在同一程序中使用。

该选项将稍稍增大所生成的代码，因此默认选项是-mno-thumb-interwork。

-mtpcs-frame

为所有非叶子函数所生成的堆栈框架将符合 Thumb 过程调用标准。（一个叶子函数就是不再调用任何其他函数的函数。）默认选项是-mno-tpcs-frame。参见选项-mtpcs-leaf-frame。

-mtpcs-leaf-frame

为所有叶子函数所生成的堆栈框架将符合 Thumb 过程调用标准。（一个叶子函数就是不再调用任何其他函数的函数。）默认选项是-mno-apcs-leaf-frame。参见选项-mtpcs-frame。

-mtune=name

该选项与-mcpu 非常相似，不同的是，它不指定实际使用的目标处理器类型及限制可以使用的指令，它指定 GCC 应该按照所指定的目标机类型优化代码的性能，但继续基于选项-mcpu 所指定的 CPU 来选择所生成的指令。

可以使用的 name 值是用于-mcpu 的可选值之一。对于某些 ARM 实现，使用该选项可以获得更好的性能。

-mwords-little-endian

按 little endian 字序生成代码，但满足 big endian 字节序。

该选项应该仅用于当你需要兼容版本 2.8 以前的编译程序为 big endian ARM 处理器所生成的代码。

该选项仅在为 big endian 处理器生成代码时有效——即所需要的字节序的形式为 32107654。

-mxopen

该选项仅用在 RISC iX 上来仿真 X/Open 模式编译程序。参见选项-mbsd。

### 21.2.5　AVR 选项

下面是为 AVR 实现所定义的选项。

-mcall-prologues

通过将函数的前导和结尾段生成为相应的子例程来减小代码尺寸。

-minit-stack=address

指定初始栈地址，可以是符号和数值。默认是__stack 的数值。

-mmcu=setting

数值 setting 指定 ATMEL AVR 指令集或 MCU 类型。表 21-5 列出了 setting 的所有合法值。第一列包含 AVR 指令集的名字，第二列包含了相应的 MCU 类型。第一列或第二列的任一名字都可以作为 setting 值。

表 21-5 AVR 和 MCU 设置

| AVR | MCU | 描述 |
|---|---|---|
| avr1 | at90s1200, attiny10, attiny11, attiny12, attiny15, attiny 28 | 这是最小 AVR 核心指令集，C 编译程序不支持它。该设置只对汇编程序有效 |
| avr2 | at90s2313, at90s2323, attiny22, at90s2333, at90s2343, at90s4414, at90s4433, at90s4434, at90s8515, at90s8534, at90s8535 | avr2 指令集是默认的。它是经典的 AVR 核心，程序的内存空间可以达到 8KB |
| avr3 | atmega103, atmega603, at43usb320, at76c711 | 这是经典 AVR 核心，程序的内存空间可以达到 128 KB |
| avr4 | atmega8, atmega83, atmega85 | 这是 AVR 扩展核心，程序的内存空间可以达到 8 KB |
| avr5 | atmega16, atmega161, atmega163, atmega32, atmega323, atmega64, atmega 128, at43usb355, at94k | 这是 AVR 的扩展核心，程序的内存空间可以达到 128 KB |

-mno-interrupts

通过生成与硬件中断不兼容的代码来减小代码尺寸。

-mno-tablejump

通过不生成 tablejump 指令而减小代码尺寸，但该指令有时也会增加代码尺寸。

-msize

在汇编语言文件中输出指令尺寸信息。

-mtiny-stack

生成仅改变栈指针低 8 位的代码。

### 21.2.6 CRIS 选项

下面是为 CRIS 端口定义的选项。

-m8-bit

调整堆栈框架、可写数据和常量，使它们全部达到 8 位对齐。

参见选项-mdata-align，-mconst-align，-mstack-align，-m32-bit 和-m16-bit。

-m16-bit

调整堆栈框架、可写数据和常量，使它们全部达到 16 位对齐。

参见选项-mdata-align，-mconst-align，-mstack-align，-m32-bit 和-m8-bit。

-m32-bit

调整堆栈框架、可写数据和常量，使它们全部达到 32 位对齐。这是默认选项。

参见选项-mdata-align，-mconst-align，-mstack-align，-m16-bit 和-m8-bit。

-maout

传统 no-op 选项，仅有目标 cris-axis-aout 识别该选项。

-march=architecture

与选项-mcpu 相同。

-mcc-init

禁止使用先前指令造成的条件代码，并始终在条件代码使用前生成比较和测试指令。

-mconst-align

调整常量，使其能够在所选 CPU 型号的最大数据访问尺寸上对齐。默认选项是-mno-const-align，它设置 32 位对齐。诸如结构布局等 ABI 细节不受影响。

参见选项-mdata-align，-m32-bit，-m16-bit 和-m8-bit。

-mcpu=architecture

为指定的体系结构生成代码。architecture 的合法选项有：v3 对应 ETRAX4，v8 对应 ETRAX100 和 v10 对应 ETRAX100LX。默认值是 v0，但 cris-axis-linux-gnu 除外，其默认值是 v10。

-mdata-align

调整单个数据项，以达到在所选 CPU 型号的最大独立数据访问尺寸上对齐。默认选项是-mno-data-align，它设置 32 位对齐。不影响诸如结构布局等 ABI 细节。

参见选项-mstack-align，-mconst-align，-m32-bit，-m16-bit 和-m8-bit。

-melf

传统 no-op 选项，仅有目标 cris-axis-elf 和 cris-axis-linux-gnu 识别该选项。

-melinux

为选项-mcpu=8 选择类 GNU/Linux multilib，包含文件和指令集。该选项仅对目标 cris-axis-aout 合法。

-melinux-stacksize=number

调整程序中的指示，通知内核加载器，程序的栈应该设置为 number 字节。该选项仅在目标 cris-axis-aout 上有效。

-metrax100

与选项-mcpu=v8 相同。

-metrax4

与选项-mcpu=v3 相同。

-mgotplt

与选项-fpic 和-fPIC 一同使用，所生成的指令序列将从 GOT 中的 PLT 加载函数地址，而不是对 PLT 的调用。这是默认选项，可以通过使用选项-mno-gotplt 关闭该选项。

-mlinux

传统 no-op 选项，仅有目标 cris-axis-linux-gnu 识别该选项。

-mmax-stack-frame=number

当某函数的堆栈框架超出了 number 字节时，会产生一个警告消息。

-mno-side-effects

禁止生成寻址模式中带有副作用的指令，后增量指令除外。

-mpdebug

在汇编代码中打开专用于 CRIS 的调试相关信息。该选项的另一个作用是在汇编文件的起始时，关闭对汇编器的#NO_APP 格式化代码指示器。

-mprologue-epilogue

这是产生设置堆栈框架的起始和结尾代码的默认选项。

使用选项-mno-prologue-epilogue 将禁止设置函数堆栈框架的正常函数起始和结尾段的生成。不会在代码中生成返回指令或返回序列。由于没有对调用期间需要保存的寄存器进行保存或分配局部变量产生警告和错误消息，因此在使用该选项的同时也需要使用可视代码检查。

-mstack-align

调整堆栈框架，使其在所选型号 CPU 的最大数据访问尺寸上对齐。默认选项是-mno-stack-align，它将设置 32 位对齐。该选项不影响诸如结构布局等 ABI 细节。

参见选项-mdata-align，-mconst-align，-m32-bit，-m16-bit，-m8-bit。

-mtune=architecture

针对所指定的体系结构，利用其所有可以利用的参数调整所生成的代码，但不包括 ABI 和可用的指令集。可选的 architecture 与选项-mcpu 相同。

-sim

链接模拟器库中的输入和输出函数。连续分配代码、已初始化的数据和初始化为 0 的数据。该选项只对目标 cris-axis-aout 和 cris-axis-elf 有效。参见选项-sim2。

-sim2

与选项-sim 相似，但会向链接器传递参数将初始化数据定位在 0x4000000，并将初始化为 0 的数据定位在 0x8000000。

### 21.2.7　D30V 选项

以下的选项均是为 D30V 实现而定义的。

-masm-optimize

打开在进行优化时向汇编器传递选项-O 的功能。汇编器使用选项-O 来自动“并行化”可能的相邻指令。这是默认选项，但可以使用选项-mno-asm-optimize 关闭该选项。

-mbranch-cost=number

将分支的内部代价增加到 number。较高的代价意味着编译程序将生成更多的指令来避免执行这一分支。默认值是 2。

参见选项-mcond-exec。

-mcond-exec=number

指定替代分支的条件执行指令的最大数目。默认值是 4。

-mextmem

将代码节.text，.data，.bss，.strings，.rodata，.rodata1 和.data1 链接到外存中，它将从 0x8000000 的位置开始。

-mextmemory

与选项-mextmem 相同。

-monchip

将代码节.text 链接到芯片上的文本内存，它将从 0x00000000 位置开始。该选项还将代码节.data，.bss，.strings，.rodata，.rodata1 和.data1 链接到芯片上的数据内存，它将从 0x20000000 位置开始。

### 21.2.8 H8/300 选项

下面的选项均是为 H8/300 实现而定义的。

-malign-300

在 H8/300H 和 H8/S 上使用的与 H8/300 一样的对齐规则。

默认情况下，H8/300H 和 H8/S 将 long 和 float 数据项对齐于 4 字节边界上。该选项会将这些数据对齐在 2 字节边界上。

该选项对 H8/300 没有影响。

-mh

生成 H8/300H 的代码。

-mint32

按 32 位数值生成 int 数据。

-mrelax

如果可能，缩短链接时的某些地址引用。该选项会设置链接器选项-relax。

-ms

生成 H8/S 的代码。

-ms2600

生成 H8/S2600 的代码。该开关必须同选项-ms 一同使用。

### 21.2.9 HPPA 选项

下面的选项均是为 HPPA 系列计算机定义的。

-march=architecture

为指定的体系结构生成代码。已知的 architecture 名字有 1.0(PA 1.0)，1.1(PA 1.1)和 2.0(PA 2.0)。

为编号较低的体系结构所编译的代码可以在较高编号的体系结构上运行，反之则不可以。请参考 HPUX 系统的/usr/lib/sched.models 以确定适合你的机器的体系结构选项。

-mbig-switch

生成适用于大开关表的代码。仅在汇编器/链接器对开关表中超出范围的跳转指令而产生警告时，会使用该选项。

-mdisable-fpregs

禁止以任何一种方式使用浮点寄存器。

当编译使用惰性寄存器上下文切换的内核时需要使用该选项。如果使用该选项的同时进行浮点操作，编译程序将会异常终止。

-mdisable-indexing

禁止编译程序使用索引寻址方式。该选项将防止在编译 MACH 下所生成的 MIG 代码时出现的某些非常不明显的问题。

-mfast-indirect-calls

生成假设函数定义不会跨越边界的代码。

该选项将允许编译程序生成执行更快的间接函数调用。当存在共享库或嵌套函数时，该选项将不起作用。

-mgas

使用仅有 gas 汇编器可以理解的汇编指令。

-mjump-in-delay

通过将函数调用的返回指针修改为条件跳转的目的地址，从而可以使用无条件跳转指令填充函数调用的延迟槽。

-mlinker-opt

打开传递给 HPUX 链接器的优化选项。该选项可能使符号调试失效。而且该选项还会触发 HPUX 8 和 HPUX 9 链接器中的一个错误，它可能在链接某些程序时给出假的错误信息。

-mlong-load-store

生成三指令加载和存储序列，这在某些时候是 HPUX 10 链接器所必须的。该选项与 HP 编译程序中的选项+k 等价。

-mno-space-regs

生成的代码会假设目标机上没有空闲寄存器。

这样可以使 GCC 生成更快速的间接调用，并使用非扩展的索引寻址模式。这种代码适合于级别 0 的 PA 系统和核心。

-mpa-risc-1-0

与选项-march=1.0 相同。

-mpa-risc-1-1

与选项-march=1.1 相同。

-mpa-risc-2-0

与选项-march=2.0 相同。

-mportable-runtime

使用由 HP 为 ELF 系统推荐的可移植调用形式。

-mschedule=type

根据指定的机器类型，限制调度代码。合法的类型选择值有 700，7100，7100LC，7200，7300 和 8000。参考 HPUX 系统中的/usr/lib/sched.models 文件来决定合适的调度选项。默认调度参数是 8000。

-msoft-float

为浮点操作生成包含库调用的输出。

并不是所有的 HPPA 目标机均有这样的库。在一般情况下，会使用机器的 C 编译程序的工具，但在交叉编译时并不能直接这样做。你必须为交叉编译提供一个合适的库函数。

该选项会改变目标文件中的调用形式，因此只有当你编译某个程序的包括库在内的所有模块时才会有用。这包括与 GCC 一起发行的库 libgcc.a。

### 21.2.10 IA-64 选项

下面的选项均是为 Intel IA-64 体系结构定义的。

-mauto-pic

生成可以自定位的代码。该选项同时设置选项-mconstant-gp。当编译固件代码时该选项很有用。

-mb-step

生成可以绕过 Itanium B step 的错误代码。

-mbig-endian

生成的代码用于 big endian 目标机。在 HPUX 上这是默认选项。

参见选项-mlittle-endian。

-mconstant-gp

生成使用单常量全局指针数值的代码。当编译内核代码时该选项很有用。

参见选项-mauto-pic。

-mdwarf2-asm

生成带 DWARF2 行号调试信息的汇编代码。当不使用 GNU 汇编器时该选项有一定用途。该选项可以通过使用选项-mno-dwarf2-asm 逆转。

-mfixed-range=range

生成的代码将所给范围的寄存器当作固定寄存器。寄存器的范围可以通过两个由短横线分隔的寄存器指定。还可以通过逗号分隔来指定多个寄存器范围。

固定寄存器是分配器无法使用的寄存器。该选项在编译内核代码时有用。

-mgnu-as

生成适用于 GNU 汇编器的代码。这是默认选项，可以使用选项-mno-gnu-as 逆转该选项。

-mgnu-ld

生成适用于 GNU 链接器的代码。这是默认选项，可以使用选项-mno-gnu-ld 逆转该选项。

-minline-divide-max-throughput

使用最大吞吐算法生成嵌入除法操作。参见选项-minline-divide-min-latency。

-minline-divide-min-latency

使用最小延迟算法生成嵌入除法操作。参见选项-minline-divide-max-throughput。

-mlittle-endian

生成适用于 little endian 目标的代码。在 AIX5 和 Linux 中这是默认选项。

参见选项-mbig-endian。

-mno-pic

生成的代码不会使用用于寻址的全局指针寄存器。其结果是所生成的代码不是位置独立的，而且会违反 IA-64 ABI 标准。

-mregister-names

为栈寄存器生成 in，loc 和 out 寄存器名。这可以使汇编器的输出更易阅读。该选项可以使用选项-mno-register-names 逆转。

-msdata

打开使用小数据区的优化选项。这是默认选项，它可以通过选项-mno-sdata 逆转，在绕过优化器错误时有一定用途。

-mvolatile-asm-stop

在 volatile 汇编语句之前和之后立即生成停止位。该选项可以使用选项-mno-volatile-asm-stop 逆转。

### 21.2.11 Intel 386 和 AMD x86-64 选项

下列选项是为 i386 和 x86-64 系列计算机定义的。

-m128bit-long-double

指定 long double 类型的尺寸为 128 位（16 字节）。尽管较新的体系结构（奔腾以及后继芯片）更希望 long double 对齐于 8 和 16 字节的边界上，i386 应用程序二进制接口规定的尺寸为 12 字节。这将无法达到与 12 字节 long double 一起做为数组访问。

如果使用了该选项，包含 long double 数据的结构或数组将会改变尺寸。而且，使用 long double 的函数的调用约定也会改变。

参加选项-m96bit-long-double。

-m32

在 64 位环境的 AMD x86-64 处理器中，该选项将设置 int，long 和指针数据为 32 位，并生成可以运行在任何 i386 系统上的代码。

-m386

与选项-mcpu=i386 相同。这种形式的选项已废止了。

-m3dnow

允许使用可以直接访问 3Dnow 扩展的内建函数。可以使用选项-mno-3dnow 禁止使用这些内建函数。

-m486

与选项-mcpu=i486 相同。这种形式的选项已废止了。

-m64

在 64 位环境的 AMD x86-64 处理器中，该选项将设置 int 为 32 位，设置 long 和指针数据为 64 位，并生成专门运行于 AMD 的 x86-64 体系结构的代码。

-m96bit-long-double

根据 i386 应用程序二进制接口的要求，指定 long double 数据项的尺寸为 96 位（12 字节）。这是默认选项。

参见选项-m128bit-long-double。

-maccumulate-outgoing-args

指定在函数引导段中计算输出参数所需的最大空间。由于可以消除依赖，改进调度并在所选的栈边界不等于 2 时会减少栈的使用，这是在大部分现代 CPU 中较快的方法。缺点是会增加代码的尺寸。使用该选项也会设置选项-mno-push-args。

-malign-double

设置编译程序将 double，long double 和 long long 变量对齐于双字边界上。使用选项-mno-align-double 会将这些数据项对齐于单字边界上。

将 double 变量对齐于双字边界上将生成在奔腾系统上运行得更快的代码，代价是程序的尺寸会变大。

选项-malign-double 将造成（包含前面那些类型的）结构与为 i386 所发布的应用程序二进制接口规范具有不同的对齐边界。

-march=architecture

生成指定机器体系结构的指令。architecture 的选择与选项-mcpu 中的 type 一样。指定选项-march 隐含着使用相同类型的-mcpu 选项。

-masm=dialect

使用指定的 dialect 输出汇编语言指令。合法的 dialect 选项是 intel 和 att。默认是 att。

-mcpu=type

除了 ABI 和可用的指令集之外，调整所生成代码，令其一切内容均可适应所选定的 CPU 类型。合法的选项有 i386，i486，i586，i686，pentium，pentium-mxx，pentiumpro，pentium2，pentium3，pentium4，k6，k6-2，athlon，athlon-tbird，athlon-4，athlon-xp 和 athlon-mp。

类型 i586 与 pentium 等价。类型 i686 与 pentiumpro 等价。k6 和 athlon 是 AMD 芯片的类型。

尽管选择一个特定的 CPU 将会为相应的芯片进行合理的调度，但编译程序不会生成不能运行于 i386 的代码，除非使用了选项-march。

-mfpmath=unit

生成所选择的硬件单元的浮点指令。

将 unit 指定为 387 将使用标准的 387 浮点协处理器，该协处理器存在于大多数芯片中，并在其他的处理机上存在仿真系统。该选项所编译的代码可以运行于大多数地方。临时结果将使用 80 位精度计算。

参见附录 D 中的选项-ffloat-store。在 i386 体系中，这是默认选择。

将 unit 指定为 sse 将使用 SSE 指令集中的标量浮点指令。Pentium3 及其以后的处理器支持该指令集，AMD 的 Athlon-4，Athlon-xp 和 Athlon-mp 芯片也支持该指令集。较早的 SSE 指令集只支持单精度运算，因此双精度和扩展精度运算仍然使用 387 完成。存在于 Pentium4 和 AMD x86-64 芯片中的较新版本支持双精度操作。

当指定 unit 为 i387 时，必须同时使用选项-march，-msse 和-msse2 来打开 SSE 扩展并从而使该选项生效。对于 x86-64 编译程序来说，这些扩展默认是打开的。所产生的代码应

该是相对快速的（在大多数情况下），并且可以避免387代码的数值不稳定性问题。然而，该选项将会使某些期望临时结果为80位值的应用无法工作。

将unit指定为sse，387将同时利用两种指令集。这其实加倍了芯片中可用的寄存器，但具有独立的387和SSE执行单元。该选项尚处于实验阶段，这是因为gcc的寄存器分配还没有考虑独立功能单元。

-mieee-fp

指定编译程序使用 IEEE 浮点比较。这些比较将会正确地处理比较的结果为无序的情况。可以使用选项-mno-ieee-fp禁止使用IEEE浮点比较。

-minline-all-stringops

嵌入所有的字符串操作。默认选项是仅当目的字串的地址至少对齐于4字节边界时才将字符串操作嵌入。该选项将造成更多的内嵌操作，虽然会增加代码的尺寸，但可以提高依赖于快速memcpy()，strlen()和memset()的代码的性能。

-mmmx

允许使用可以直接访问 MMX 扩展的内建函数。可以使用选项-mno-mmx 禁止这样使用。

-mno-align-stringops

不对齐内嵌字符串操作的目的地址。该选项将减少代码的尺寸，而且当目的地址已经对齐但编译程序不知道时会改善性能。

-mno-fancy-math-387

某些387模拟器不支持387的sin，cos和sqrt指令。该选项将避免产生这些指令。除非你也指定了选项-funsafe-math-optimization，否则该选项将没有任何效果。

在 FreeBSD，OpenBSD 和 NetBSD 系统上该选项是默认的。当选项-march 显示目标CPU永远都存在FPU而且无需进行仿真时，该选项将被忽略。

-mno-fp-ret-in-387

指定对于函数的返回值不使用FPU寄存器。

通常的调用规范会将类型为float和double的返回值存放在FPU寄存器中，即使不存在FPU。其工作原理是操作系统应该仿真FPU。使用该选项会在普通的CPU寄存器中返回这些值。

-mno-red-zone

在AMD x86-64处理器的64位环境中，该选项将抑制x86-64代码中所谓红区(red zone)的使用。在x86-64 ABI中红区是必须的，它是一个128字节的区域，并位于不会由信号或

中断处理函数修改的栈指针的上方，因此可以用来保存临时数据而无需调整栈指针。该选项将禁止使用红区。

-momit-leaf-frame-pointer

不为叶子函数在寄存器中保存栈指针。这将可以避免生成保存、建立和恢复栈指针的指令，而且可以留出额外的寄存器供叶子函数使用。

选项-fomit-frame-pointer 可以用来去除所有函数中的栈指针，但这样做会使调试更加困难。

-mpentium

与选项-mcpu=pentium 相同。这种形式的选项已废止。

-mpentiumpro

与选项-mcpu=pentiumpro 相同。这种形式的选项已废止。

-mpreferred-stack-boundary=number

保持栈的边界对齐到 2 的 number 幂字节边界上。默认的 number 数值为 4（16 字节或 128 位）。

使用选项-Os 优化代码尺寸，将设置最小的正确对齐尺寸（x86 系统中为 4 字节，x86-64 为 8 字节）。在 Pentium 和 Pentium Pro 中，为防止代码运行较慢，double 和 long double 数值应该对齐于 8 字节边界上。在 Pentium III 中，如果没有对齐于 16 字节边界上，流 SIMD 扩展（SSE）数据类型__m128 也会出现类似的性能损失。

为了保证栈中的数据对齐于合适的位置上，栈必须对齐于任何保存于栈中的数据所要求的对齐边界上。而且，每一个所生成的函数都必须保持堆栈对齐。这意味着用具有较小边界对齐栈的函数调用具有较大边界对齐栈的函数时，很有可能会造成栈不对齐。推荐使用回调的库始终使用默认设置。

额外的对齐会消耗栈空间并通常会增大代码的尺寸。对于对栈空间的使用敏感的代码，如嵌入式系统和操作系统内核，你可能需要将希望的对齐边界设置为-mpreferred-stack-boundary=2。

参见选项-malign-double。

-mpush-args

使用 push 操作保存外出参数。该方法更简洁而且通常与使用 sub/mov 操作同样快，而且默认情况下会使用该方法。可以使用选项-mno-push-args 改变默认设置，在某些情况下，由于调度和所减少的依赖性，不使用这一选项可能会改进性能。

-mregparm=number

指定用于传递整数参数的寄存器个数。默认情况下，不使用寄存器传递参数。number

值最大为3。你可以使用函数属性regparm来专为某一函数控制这一行为。

当这一选项所使用的number为非零值时，必须使用同一数值构建所有的模块，包括所有的库。

-mrtd

使用不同的函数调用规范，在这一规范中，具有固定数目参数的函数将使用ret num指令返回，它将在返回的同时弹出参数。由于没有必要在函数返回后弹出参数，这样可以在调用方节省一条指令。

你可以使用函数属性stdcall来专为某个函数指定它的调用规范。还可以使用函数属性cdecl取代选项-mrtd。

使用这一调用规范与通常用于UNIX的规范不兼容，因此如果你需要调用使用UNIX编译程序编译的代码，就不能使用这一规范。而且，你必须为所有具有可变参数数目的函数提供函数原型；否则编译程序可能会对这些函数生成错误的代码。

如果你用过多的参数调用函数，将生成不正确的代码。通常，额外的参数会被忽略。

-msoft-float

生成包含浮点操作库调用的代码。相应的函数库并不是GCC的一部分。可以使用目标计算机的C编译程序函数库，但不能直接用于交叉编译。因此必须为交叉编译程序提供相应的函数库。

在函数利用80387寄存器栈返回浮点结构的机器上，即使使用了选项-msoft-float，仍可能生成某些浮点运算符。

-msse

启用允许直接访问SSE扩展的内建函数的代码。可以使用选项-mno-sse禁止使用这些函数。

-msse2

启用允许使用直接访问SSE2扩展的内建函数的代码。可以使用选项-mno-sse2禁止使用这些函数。

-mthreads

支持Mingw32的线程安全异常处理。必须使用选项-mthreads对依赖于线程安全异常处理的代码进行编译并连接。使用该选项时会定义-D_MT。当连接时，它将包含使用选项-lmingwthrd连接一个特殊的线程辅助库，它将为每个线程清理异常处理数据。

-msvr3-shlib

指定编译程序将未初始化的局部变量放置在bss数据段。如果需要指定将它们放置在data数据段中，可以使用选项-mno-svr3-shlib。这些选项仅在System V Release 3中有效。

### 21.2.12　Intel 960 选项

以下选项均是为 Intel 960 实现定义的。

-mtype

假设所指定的机器类型为默认类型。该选项将设置指令调度、浮点操作支持和寻址模式。可能的选择为 ka，kb，mc，ca，cf，sa 和 sb。默认选项是 kb。

-masm-compat

打开 iC960 汇编器的兼容性支持。

-mcode-align

为了快速访问数据，将代码对齐于 8 字节边界上。目前，该选项仅在 C 序列的实现中被默认打开。对于其他实现，默认选项是-mno-code-align。

-mcomplex-addr

在这个 i960 实现上，编译程序默认使用复杂的寻址模式。复杂寻址模式在 K 系列上可能是没有意义的，但在 C 系列上是绝对有效的。除 cb 和 cc 之外，其他所有处理器的默认选项是-mcomplex-addr，cb 和 cc 的默认选项是-mno-complex-addr。

-mic-compat

打开 iC960 兼容性支持。

-mic2.0-compat

打开 iC960 版本 2.0 兼容性支持。

-mic3.0-compat

打开 iC960 版本 3.0 兼容性支持。

-mintel-asm

与选项-masm-compat 相同。

-mleaf-procedures

该选项指定编译程序应该修改叶子过程，既可由 call 调用也可由 bal 指令调用。可以使用选项-mno-leaf-procedures 禁止该选项。

使用选项-mleaf-procedures 的结果是，当可以用汇编器或链接器替代 bal 指令时，可以产生非常高效的代码，但其他情况下就并不高效了，例如通过函数指针调用。

-mlong-double-64

将 long double 数据类型实现为 64 位浮点数。如果不使用该选项，long double 将被实现为 80 位浮点数。

使用该选项的惟一原因是在 fp-bit.c 中缺少对 128 位 long double 的支持。该选项仅对使用软浮点目标系统的用户有用。

-mnumerics

该选项指定处理器支持浮点指令。参见选项-msoft-float。

-mold-align

打开与 Intel 的 gcc 发行版本 1.3（基于 gcc 1.37）的结构对齐兼容性支持。设置该选项将同时设置选项-mstrict-align。

-msoft-float

该选项指定处理器不支持浮点指令。参见选项-mnumerics。

-mstrict-align

不允许访问未对齐的地址。如果需要允许访问未对齐的地址，可以使用选项-mno-strict-align。

-mtail-call

控制编译程序进行额外的尝试（超过编译程序中机器独立部分的那些）将尾递归调用优化为分支。由于对无效情况的检测并不是完全的，因此可能不需要这一选项。默认选项是-mno-tail-call。

### 21.2.13 M32R/D 选项

如下选项是为 Mitsubishi M32R/D 体系结构定义的。

-m32r

为 M32R 生成代码。这是默认选项。

-m32rx

为 M32R/X 生成代码。

-mcode-model=name

将 name 设置为 small 将指示编译程序假设所有的对象保存于低 16MB 内存中，因此可以使用指令 ld24 加载这些数据对齐。同时，编译程序还假设所有的子例程可以通过 bl 指令访问。这是默认选项。

将 name 设置为 medium 将指示编译程序假设所有的对象保存于 32 位地址空间的任意位置（编译程序将生成 seth/add3 指令来加载它们的地址）。同时，编译程序还假设所有的子例程可以通过 bl 指令访问。

将 name 设置为 large 将指示编译程序假设所有的对齐保存于 32 位地址空间的任意位

置（编译程序将生成 seth/add3 指令来加载它们的地址）。同时，编译程序将不假设所有的子例程可以通过指令 bl 进行访问（编译程序将生成更慢的 seth/add3/jl 指令序列）。

-msdata=setting

该选项指定哪些数据项将保存在小数据区中。小数据区包含.sdata 和.sbss 数据节。数据对象可以显式地放置在具有 section 属性，设置两个小数据区中的任何一个。参见选项-G。

将 setting 设置为 none 将禁止使用小数据区。变量将被放置在.data，.bss 和.rodata 区之一（除非指定了 section 属性）。这是默认选项。

将 setting 设置为 sdata 会将小的全局和静态数据放置在小数据区中，但不会生成特殊指令访问这些数据。

将 setting 设置为 use 会将小的全局和静态数据放置在小数据区中，并生成特殊的指令访问这些数据。

-G number

将小于或等于 number 个字节的全局和静态数据对象放置在小数据或 bss 数据区（而非普通的数据或 bss 数据区）中。number 的默认值是 8。

必须将选项-msdata 设置为 sdata 或 use 才能使该选项生效。

所有的模块均应该使用相同的-msdata 和-G 选项进行编译。使用不同的 number 数值进行编译将可能无法正常工作。如果不能正常工作，链接器将探测到错误并防止生成不正确的代码。

### 21.2.14　M680x0 选项

下面的选项均是为 680x0 系列定义的。根据编译程序被配置时所选择的 680x0 系列 CPU 的不同，这些选项的默认值也有所不同。

-m5200

为 520X “coldfire” 系列 CPU 生成代码。当针对基于 520X 的系统配置编译程序时，这是默认选项。具有 5200 核心的微控制器应该使用该选项，包括 MCF5202，MCF5203，MCF5204 和 MCF5202。

该选项同时设置选项-mnobitfield。

-m68000

为 6800 生成代码。该选项是为具有 68000 或 EC000 核心的微控制器设计的，包括 68008，68302，68306，68307，68322，68328 和 68356。当将编译程序配置为生成 68000 系列代码时，这是默认选项。

该选项同时设置选项-mnobitfield。

-m68020

为 68020 系统生成代码。当针对基于 68020 的系统配置编译程序时，该选项是默认选项。

该选项同时设置选项-mbitfield。

-m68020-40

为 68040 系统生成代码，但不使用任何新指令。其结果是在 68020/68881，68030 或 68040 系统上代码运行效率会有所改善。所生成的代码会使用通过 68040 仿真的 68881 指令。

-m68020-60

为 68060 系统生成代码，但不使用任何新指令。其结果是在 68020/68881，68030 或 68040 上代码运行效率会有所改善。所生成的代码会使用通过 68060 仿真的 68881 指令。

-m68030

为 68030 系统生成代码。当针对基于 68030 的系统配置编译程序时，这是默认选项。

-m68040

为 68040 系统生成代码。当针对基于 68040 的系统配置编译程序时，这是默认选项。该选项将禁止使用不得不在 68040 上仿真的 68881/68882 指令。如果你的 68040 不需要仿真这些指令的代码，可以使用该选项。

-m68060

为 68060 系统生成代码。当针对基于 68060 的系统配置编译程序时，这是默认选项。该选项将禁止使用不得不在 68060 上仿真的 68020 和 68881/68882 指令。如果你的 68060 不需要仿真这些指令的代码，可以使用该选项。

-m68881

生成包含 68881 浮点指令的代码。对大部分 68020 系统来说，这是默认选项，除非当配置编译程序时使用了选项--nfp。

-malign-int

该选项指定 int，long，long long，float，double 和 long double 变量对齐于 32 位边界上。默认选项是-no-align-int，它将对齐于 16 位边界上。将变量对齐于 32 位边界将生成较大的代码，但会在具有 32 位总线的处理机上运行更高效。

当使用选项-malign-int 时，GCC 会对齐包含与先前所发表的 m680x0 的应用二进制接口规范不同的数据类型的结构。

-mbitfield

允许使用位字段指令。选项-m68020 将隐含打开该选项。如果针对 68020 配置编译程序，该选项将是默认的。

-mc68000

与选项-m6800 相同。

-mc68020

与选项-m68020 相同。

-mcpu32

为 CPU32 系统生成代码。当针对基于 CPU32 的系统配置编译程序时，这是默认选项。应该对具有 CPU32 或 CPU32+核心的微控制器使用该选项，这包括 68330，68331，68333，68334，68336，68340，68341，68349 和 68360。

该选项同时设置选项-mnobitfield。

-mfpa

为浮点运算生成包含 Sun FPA 指令的代码。

-mno-strict-align

编译程序假设非对齐内存引用将由系统处理。如果需要强制编译程序对齐所引用的数据，可以使用选项-mstrict-align。

-mnobitfield

禁止使用位字段指令。选项-m68000，-mcpu32 和-m5200 隐含使用该选项。

-mpcrel

直接使用 68000 的 PC 相对（程序计数器相对）寻址模式，而非使用全局偏移表。

该选项同时设置选项-fpic 以允许至多 16 位偏移的 PC 相对寻址。目前，该选项还不支持选项-fPIC。

-mrtd

使用不同的函数调用规范，其中函数使用固定数目的参数并使用 rtd 指令通过在运行时弹出参数完成函数返回。由于无需弹出参数，这样可以为调用方节省一条指令。68010，68020，68030，68040，68060 和 CPU32 处理器支持指令 rtd，但 68000 和 5200 不支持。

该调用规范与通常使用在 UNIX 系统中的调用规范不兼容，因此，如果你需要调用由 UNIX 编译程序编译的库时，不能使用这一规范。而且，你必须为所有的函数提供你自己的函数原型，如 printf()，该函数具有可变数目的参数。如果没有合适的原型，将会生成对这些函数的不正确调用。

如果用过多的参数调用某个函数，将导致不正确的代码。但使用默认的调用序列，额外的参数将被忽略。

-mshort

将类型 int 限定为 16 位数值，与 short int 相同。

-msoft-float

为浮点操作生成相应的库调用代码。

如果需要该选项，你将需要为交叉编译提供相应的库，而且这样的库并不是所有的m680x0 目标系统均具备。通常，可以使用目标机上的 C 编译程序工具，但在交叉编译时不能这样直接使用。嵌入式目标 m68k-*-aout 和 m68k-*-coff 提供了相应的软件浮点支持。

### 21.2.15 M68HClx 选项

如下这些选项是为 68HC11 和 68HC12 微控制器定义的。这些选项的默认值依赖于在配置编译程序时选择了哪一类微控制器。

-m6811

为 68HC11 生成代码。当针对基于 68HC11 的系统配置编译程序时，这是默认选项。

-m6812

为 68HC12 生成代码。当针对基于 68HC12 的系统配置编译程序时，这是默认选项。

-m68hc11

与选项-m6811 相同。

-m68hc12

与选项-m6812 相同。

-mauto-incdec

允许使用 68HC12 中的前增量和后增量寻址模式。

-mshort

将 int 数据类型视为 16 位数值，与 short int 相同。

-msoft-reg-count=count

指定伪软件寄存器的数目为 count，它将在代码生成时使用。最大的数值为 32。依赖于不同的程序，使用更多的伪软件寄存器会（或不会）达成更高效的代码。在 68HC11 中默认值是 4，在 68HC12 中为 2。

### 21.2.16 M88K 选项

如下选项是为 Motorola 88K 体系结构定义的。

-m88000

生成在 m88100 和 m88110 上均可工作的代码。

-m88100

生成在 m88100 上运行高效但在 m88110 上也可以工作的代码。

-m88110

生成在 m88110 上运行高效但在 m88100 上也可以工作的代码。

-mbig-pic

已废止的选项。应使用选项-fPIC 代替。

-mhandle-large-shift

包含检测位移操作超过 31 位的代码。这样的位移操作将操作内陷，或者需要生成相应的代码进行处理。默认情况下，GCC 不对过大的位移操作进行检测。

-midentify-revision

在汇编输出中包含 ident 指令，记录源文件名、编译程序名和版本、时间戳和编译标志集。

-mno-check-zero-division

生成不检测除数为 0 的整数除法的代码。默认选项是-mcheck-zero-division，它将检测除数为 0 的整数除法。

M88100 处理器的某些型号在一定条件下不能陷入除数为 0 的除法。默认情况下，当编译可能运行于这种处理器之上的代码时，GCC 将生成代码显式地检查除数是否为 0，并在检测到后使用异常号 503 进入陷阱。使用选项-mno-check-zero-division 将抑制在 MC88100 处理器上生成完成这种检测的代码。

GCC 假设 MC88110 处理器能够正确地检测所有除数为 0 的整数除法。当使用选项-m88110 时，不生成任何检测 0 除数的代码，并且忽略选项-mcheck-zero-division 和-mno-check-zero-division。

-mno-underscores

指定由编译程序生成的汇编语言不在每个符号的起始增加一个下划线。默认情况是在每个符号的起始处增加一个下划线。

-mocs-debug-info

根据 88open 目标代码兼容标准（Object Compatibility Standard，OCS），包含在每个堆栈框架中使用的寄存器调试信息。额外的信息将在去除栈指针后仍可能调试代码。

对 DG/UX，SVr4 和 Delta 88 SVr3.2 来说，默认情况是包含这些信息，可以使用选项-mno-ocs-debug-info 去除这些信息。默认情况下，其他的 88K 配置会忽略这些信息。

-mocs-frame-position

当为保存在栈中的自动变量和参数生成 COFF 调试信息时，该选项指定使用到栈地址的偏移，它是进入函数时的栈指针（寄存器 31）。

当为保存在栈中的自动变量和参数生成 COFF 调试信息时，使用选项-mno-ocs-frame-position 指导编译程序使用到栈指针（寄存器 30）的偏移。当该选项生效时且使用选项-g 选择调试信息时，栈指针将不会被去除。

默认情况下，DG/UX，SVr4，Delta88 SVr3.2 和 BCS 配置将使用偏移，但可以使用选项-mno-ocs-frame-position 抑制。其他的 88K 配置则默认使用选项-mno-ocs-frame-position。

-moptimize-arg-area

该选项通过重新组织堆栈框架来节省内存空间。该选项生成的代码与 88open 规范不兼容，但使用的内存较少。

使用选项-mno-optimize-arg-area 将禁止通过重新组织堆栈框架来节省内存空间。这是默认选项，它符合规范但却占用较多的内存。

-mshort-data-number

通过相对寄存器 r0 引用数据生成更小的数据引用，这样可以使用一条指令而非两条指令加载一个数值。number 的数值应该大于 0，但大于 65 535 的数值无效。

该选项通过指定最大偏移 number 控制哪些数据引用将受影响。例如，如果指定选项-mshort-data-512，所影响的是那些偏移大于 512 字节的数据引用。

-mserialize-volatile

生成保证可变内存引用顺序一致性的代码。这是默认选项，但可以使用选项-mno-serialize-volatile 禁止这一选项。

由 MC88110 处理器生成的内存引用顺序并不总能与指令请求这些引用的顺序一致。特别是，一条加载指令可以在后一条存放指令前执行。在多处理器环境中，某些乱序会破坏可变内存引用的顺序一致性。当必须保证一致性时，GCC 会生成特殊的指令来保证指令的执行顺序。

MC88100 处理器不会对内存引用产生乱序，因此它始终都可以提供顺序一致性。但是，默认情况下，即使你使用了选项-m88100，GCC 也会生成特殊指令保证一致性，以确保代码可以运行在 MC88110 处理器上。如果你只希望代码运行在 MC88100 处理器上，需要使用选项-mno-serialize-volatile。所生成的用来保证一致性的额外代码可能会影响应用程序的性能，如果确信去除这一保证仍然很安全，那就去除它。

-msvr3

关闭与 System V release 4(SVr4)相关的编译程序扩展。参见-msvr4。

-msvr4

打开与 System V release 4(SVr4)相关的编译程序扩展。该选项决定生成哪一种汇编器语法的变体，C 预处理程序是否识别#pragma weak 以及 GCC 是否生成用于 SVr4 的额外声明指令。

该选项对 m88k-motorola-sysv4 和 m88k-dg-dgux 配置是默认选项。对于这些配置，可以使用选项-msvr3 改变默认选项。对于其他 M88K 配置，默认选项是-msvr3。

-mtrap-large-shift

与选项-mhandle-large-shift 相同。

-muse-div-instruction

为 MC88100 的带符号整数除运算生成使用指令 div 的代码。默认情况下，并不使用指令 div。

在 MC88100 处理器上，当操作数为负数时，带符号整数除法指令 div 会进入操作系统的陷阱中。操作系统会透明地完成相应的操作，但执行时间会很长。默认情况下，当编译可能运行在 MC88100 处理器上的代码时，GCC 将使用无符号整数除法指令 divu，来仿真带符号整数除法，从而可以避免进入操作系统陷阱而产生的巨大代价。但这种仿真在空间和时间上也会有较小的代价。如果在你的代码中的带符号整数除法操作的是两个非负整数时，可能需要直接使用 div 指令。

在 MC88110 处理器上，指令 div（也称为指令 divs）可以不进入操作系统陷阱而正确处理负操作数。当使用选项-m88110 时，选项-muse-div-instruction 将被忽略，同时带符号整数将使用指令 div。

系统没有定义 INT_MIN 除以-1 的结果。特别是当使用（和没有使用）选项-muse-div-instruction 时，这一除法的行为可能会有所不同。

-mversion-03.00

该选项已废止并被忽略。

-mwarn-passed-structs

当函数使用结构作为参数或返回值时产生一个警告。在 C 语言演化的过程中，传递结构的方法发生了很大变化，而且通常是可移植性问题的根源。默认情况下 GCC 不产生警告。

### 21.2.17　MCore 选项

以下选项是为 Motorola MCore 处理器定义的。

-m210

生成用于 210 处理器的代码。

-m340
生成用于340处理器的代码。

-m4byte-functions
强制所有的函数均对齐于4字节边界上。可以使用-mno-4byte-functions逆转该选项。

-mbig-endian
生成用于big endian目标的代码。参见-mlittle-endian。

-mcallgraph-data
生成调用图信息。可以使用-mno-callgraph-data逆转该选项。

-mdiv
使用硬件除法指令。这是默认选项，但可以使用选项-mno-div逆转。

-mhardlit
如果可以用两条或少于两条的指令完成，常量将被内嵌在代码流中。可以使用选项-mno-hardlit关闭该选项。

-mlittle-endian
生成用于little endian目标的代码。参见-mbig-endian。

-mrelax-immediate
在位操作中允许任意尺寸的立即数。可以使用-mno-relax-immediate逆转该选项。

-mslow-bytes
在读取字节数量时尽可能使用字访问方式。该选项可以使用选项-mno-slow-bytes关闭。

-mwide-bitfields
位字段被保存在int数据类型中。可以使用-mno-wide-bitfields逆转该选项。

### 21.2.18 MIPS选项

以下选项是为MIPS系列计算机定义的。

-m4650
打开选项-msingle-float，-mmad和-mcpu=r4650。

-mabi=name
生成名为ABI的代码。可以识别的ABI名字有32，o64，n32，64，eabi和meabi。

名字eabi选择由Cygnus定义的嵌入式ABI。名字meabi选择由MIPS定义的嵌入式ABI。这些ABI均有32位和64位变体。默认情况下，当选择64位体系结构时编译程序将生成64位代码，但当你使用选项-mgp32时将生成32位代码。

-mabicalls

生成某些 System V.4 移植用于位置独立代码的伪操作.abicalls，.cpload 和.cprestore。该动作可以通过选项-mno-abicalls 取消。

-march=architecture

生成可以运行于特定体系结构的代码，该体系结构的名字可以是通用的 MIPS ISA 或特定的处理器名称。

通用的 ISA 名字有 mips1，mips2，mips3，mips4，mips32 和 mips64。

特定的处理器名字有 r2000，r3000，r3900，r4000，vr4100，vr4300，r4400，r4600，r4650，vr5000，r6000，r8000，4kc，4kp，5kc，20kc 和 orion。

特定的体系结构名称 from-abi 选择与所选 ABI 最兼容的体系，对 32 位 ABI 是 mips1，对 64 位 ABI 是 mips3。

宏 _MIPS_ARCH 将被定义为包含目标体系结构名的字串。而且，宏 _MIPS_ARCH_architecture 将被定义为使用宏_MIPS_ARCH 中的字串的大写形式。例如，-march=r2000 定义宏_MIPS_ARCH 为“r2000”并定义宏_MIPS_ARCH_R2000。如前面列出的那样，宏_MIPS_ARCH 的定义使用了名字的完整前缀形式，因此不会将 000 缩写为 k。使用 from-abi 将使宏被定义为“mips1”或“mips3”。如果没有使用选项-march，就会使用默认体系名。

-mdouble-float

允许编译程序使用双精度操作。这是默认选项。

参见选项-msingle-float。

-membedded-data

如果可能，首先为只读数据区分配变量，然后为小数据区分配。否则，它们均将被分配在数据区中。其结果是产生比默认情况下运行稍慢的代码，但当运行时可以减少所需要的 RAM 容量，因此很适合于某些嵌入式系统。该操作可以通过使用选项-mno-embedded-data 关闭。

-membedded-pic

生成适合于某些嵌入式系统的 PIC 代码。所有的调用均使用 PC 相对寻址，而且所有的数据均使用$gp 寄存器寻址。可以访问不超过 65 536 字节的全局数据。该选项需要使用 GNU as 和 GNU ld，它们将完成其中的大部分操作。当前，该选项仅在使用 ECOFF 的目标系统上有效；它不能同 ELF 一同使用。

-mentry

使用 entry 和 exit 伪操作。该选项只能同选项-mips16 一同使用。

-mfix7000

传递参数给 gas，如果对指令 mfhi 和 mflo 的目标寄存器的读取发生在其后两条指令之内时，将插入 noops。

-mflush-func=function

指定函数名，调用它会刷新 I 和 D 高速缓存。使用选项-mno-flush-func 将通知编译程序不调用任何这类函数。

如果调用了，函数必须具有与通用函数_flush_func()相同的参数——即高速缓存所需要刷新的内存范围地址、内存范围的尺寸，以及数字 3（两个高速缓存均被刷新）。默认选项将依赖于 GCC 所配置的目标，但通常就是_flush_func()或_cpu_flush()。

-mfp32

默认浮点寄存器的宽度为 32 位。

-mfp64

默认浮点寄存器的宽度为 64 位。

-mfused-madd

当浮点乘法和累加硬件指令可用时，生成使用这些指令的代码。只有当这些指令可用时才会默认生成这些指令。但是这一特性在某些情况下可能是不需要的，例如额外的运算精度会产生问题，或是在某些芯片上 denormals 被取整为 0 的模式，以及由乘法和累加指令生成的 denormal 会引起异常的情况。

可以使用选项-mno-fused-madd 禁止使用硬件浮点操作。

-mgas

为 GNU 汇编器生成代码。该选项在 OSF/1 参考平台上是默认的，并使用 OSF/rose 对象格式。而且，如果使用了配置选项-with-gnu-as，-mgas 也是默认选项。

参见选项-mmips-as。

-mgp32

假设通用寄存器的宽度为 32 位。

-mgp64

假设通用寄存器的宽度为 64 位。

-mgpopt

指示编译程序将所有的数据声明写在文本区中的指令前，这将允许 MIPS 汇编器生成单字内存引用，而非相对全局或静态数据项的双字引用。在打开优化选项时这是默认选项，但可以使用选项-mno-gpopt 关闭这一选项。

-mhalf-pic

将指针放置在数据区的外部引用中，而非代码区，并从那里加载这些指针。该动作可以通过选项-mno-half-pic 关闭。

-mhard-float

生成包含硬件浮点指令的输出。这是默认选项。

参见选项-msoft-float。

-mint64

强制 int 和 long 数据类型为 64 位宽。

int，long 和指针数据类型的默认尺寸依赖于 ABI。所有被支持的 ABI 均将 int 作为 32 位宽度使用。n64 ABI 和 64 位 Cygnus EABI 使用 64 位宽的 long。所有其他 ABI 均使用 32 位宽 long。指针的尺寸与 long 或整数寄存器的尺寸相同，取其中小的作为指针的宽度。

参见选项-mlong64 和-mlong32。

-mips1

与选项-march=mips1 相同。

-mips2

与选项-march=mips2 相同。

-mips3

与选项-march=mips3 相同。

-mips4

与选项-march=mips4 相同。

-mips16

允许使用 16 位指令。但可以使用-mno-mips16 关闭这一选项。

-mips32

与选项-march=mips32 相同。

-mips64

与选项-march=mips64 相同。

-mlong-calls

生成的代码使用 JALR 指令完成函数调用，该指令需要在调用前将函数地址加载到寄存器中。当需要调用超出现在的 512MB 段而又不能通过指针寻址时，就必须使用该选项。可以使用选项-mno-long-calls 关闭该选项。

-mlong32

强制 long，int 和指针类型为 32 位宽。

int，long 和指针数据类型的默认尺寸依赖于 ABI。所有被支持的 ABI 均将 int 作为 32 位使用。n64 ABI 和 64 位 Cygnus EABI 将 long 作为 64 位使用。所有其他 ABI 则将 long 作为 32 位使用。指针的尺寸与 long 或整数寄存器的尺寸相同，取其中较小的尺寸作为指针的宽度。

参见选项-mint64 和-mlong64。

-mlong64

强制 long 数据类型为 64 位宽。

int，long 和指针数据类型的尺寸依赖于 ABI。所有被支持的 ABI 均将 int 作为 32 位使用。n64 ABI 和 64 位 Cygnus EABI 将 long 作为 64 位使用。所有其他 ABI 则将 long 作为 32 位使用。指针的尺寸与 long 或整数寄存器的尺寸相同，取其中较小的尺寸作为指针的宽度。

参见选项-mint64 和-mlong32。

-mmad

允许使用如 r4650 芯片上的指令 mad，madu 和 mul。可以使用-mno-mad 逆转该选项。

-mmemcpy

生成调用相应的字符串函数（memcpy()或 bcopy()）的指令，但不生成内嵌代码。可以使用选项-mno-memcpy 取消这些函数调用。

-mmips-as

为 MIPS 汇编器生成代码并调用 mips-tfile 增加正常的调试信息。它使用 OSF/rose 对象格式，对所有平台而言均是默认选项（除了 OSF/1 参考平台）。如果使用了选项-gstabs 或-gstabs+，程序 mips-tfile 将使用 MIPS ECOFF 封装 STABS。

参见选项-mgas。

-mmips-tfile

由 MIPS 汇编器输出的目标文件格式将由程序 mips-tfile 进行后处理以增加调试支持。该动作可以通过使用选项-mno-mips-tfile 取消。选项-mno-mips-tfile 应该仅当 mips-tfile 程序存在使编译不成功的错误时才会被指定。

如果 mips-tfile 没有运行，调试器将没有本地变量可用。而且，stage2 和 stage3 目标文件将使用临时文件名传递给嵌入到目标文件的汇编器，这意味着目标文件将不会比较是否一致。

-mrnames

指示编译程序利用 MIPS 软件名而非硬件名（即 a0 而非$4）生成代码。已知的惟一可以支持这一选项的汇编器是 Algorithmics 汇编器。可以使用-mno-renames 取消该选项。

-msingle-float

假定浮点协处理器像 r4650 芯片一样仅支持单精度运算。参见选项-mdouble-float。

-msoft-float

生成包含浮点操作库调用的输出。

所需要的库并不是 GCC 的一部分。通常会使用目标机的正常 C 编译程序的功能，但在进行交叉编译时不能直接这样做。必须为交叉编译提供自己相应的库函数。

参见选项-mhard-float。

-msplit-addresses

生成分别加载地址常量的高地址部分和低地址部分的代码。这允许编译程序可以优化掉冗余的对高地址部分的加载。这一优化需要 GNU as 和 GNU ld。对于某些嵌入式目标机，由于 GNU as 和 GNU ld 是标准配置，默认情况下该优化是打开的。可以使用选项-mno-split-addresses 禁止该选项。

-mstats

对每一个已处理的非内嵌函数，该选项将使编译程序在标准错误输出中输出包含关于程序统计（保存的寄存器数目、栈尺寸等）的一行信息。可以使用选项-mno-stats 取消该动作。

-mtune=architecture

对指定的体系结构进行优化。除了其他的优化外，该选项将控制如何调度指令以及算术运算操作的代价。architecture 名字的列表与选项-march 相同。

当没有使用该选项时，编译程序将依据由-march 所指定的处理器进行优化。通过同时使用选项-march 和-mtune，可能生成的代码可以在一系列处理器上运行，但这些代码却专为其中的某些处理器进行了优化。

该选项将定义宏_MIPS_TUNE 和_MIPS_TUNE_architecture，它将按照由选项-march 定义的两个宏的同样规则创建。

-muninit-const-in-rodata

当与选项-membedded-data 一同使用时，该选项会始终将未初始化的常量变量存放在只读数据区。

-EL

为处于 little endian 模式的处理器编译代码。参见选项-EB。

-EB

为处于 big endian 模式的处理器编译代码。参见选项-EL。

-G number

将尺寸小于或等于 number 字节的全局和静态数据项放置在小数据区或小 bss 区（而不是正常数据区或正常 bss 区）中。这允许汇编器基于全局指针（gp 或$28）生成单字内存引用指令，而不是正常的双字指令。默认情况下，当使用 MIPS 汇编器时 number 是 8，而使用 GNU 汇编器时为 0。

该选项设置同时也被传递给汇编器和链接器，因此所有的模块都应该使用同样的 number 数值进行编译。

-nocpp

通知 MIPS 汇编器在编译用户汇编器文件（后缀为.s 的文件）时不会运行它的预处理程序。

-no-crt0

不包含默认的 crt0。

### 21.2.19 MMIX 选项

以下选项是为 MMIX 定义的。

-mabi=setting

将 settting 设置为 mmisware 时所生成的代码中，函数参数和返回值将在所调用的函数中通过$0 和以上的寄存器传递。

将 settings 设置为 gnu 将生成 GNU ABI，它将使用$231 及以上的全局寄存器。

-mbase-addresses

生成使用基地址的代码。

使用基地址将自动生成一个请求来为全局寄存器设置常量。该寄存器将用于一至多次范围在 0~255 的基地址请求，以获取其中的数值。这通常会导致短而且快的代码，但可以访问的数据项数目是有限的，这意味着一个使用很多静态数据的程序可能需要使用选项-mno-base-addresses 来取消该选项。

-mbranch-predict

当静态分支预测表明某个概率分支时，会使用概率分支指令。可以使用-mno-branch-predict 取消该动作。

-melf

生成 ELF 格式的可执行体，而不是由 MMIX 模拟器所使用的默认 mmo 格式。

-mepsilon

生成相对于 rE epsilon 寄存器的浮点比较指令。可以使用-mno-epsilon 取消该选项。

-mknuthdiv

使除法产生的余数与除数具有相同的正负号。默认选项是-mno-knuthdiv，也就是余数的符号与被除数的符号相同。从数学角度说，两种方法都是合法的，但基本均使用后者。

-mlibfuncs

指定正在编译的基本库使用寄存器传递所有的数值，无论数值的大小。可以使用-mno-libfuncs 取消该选项。

-msingle-exit

强制所生成的代码的每一个函数均只有一个出口。可以使用-mno-single-exit 取消该选项。

-mtoplevel-symbols

在所有全局符号的前面追加一个冒号，这样汇编代码就可以使用 PREFIX 汇编指令了。可以使用-mno-toplevel-symbols 取消该动作。

-mzero-extend

当从内存读取尺寸小于 64 位的数据时，默认使用零扩展加载指令，而不是符号扩展加载指令。可以使用-mno-zero-extend 取消该选项。

### 21.2.20　MN10200 选项

以下选项是为 Matsushita MN10200 体系结构定义的。

-mrelax

指定链接器应该执行松弛优化阶段，缩短分支、函数调用和绝对内存寻址。该选项仅当在最后链接阶段的命令行中使用才有效。而且，该选项将使符号调试无效。

### 21.2.21　MN10300 选项

如下选项是为 Matsushita MN10300 体系结构定义的。

-mam33

生成使用 AM33 处理器所特有的功能的代码。默认选项是-mno-am33，它将不生成使用这些特殊功能的代码。

-mmult-bug

生成避免 MN10300 处理器中乘法指令缺陷的代码。这是默认选项。可以使用选项-mno-mult-bug 关闭该选项。

-mno-crt0

默认情况下系统会连接C运行时初始化例程，但该选项将造成系统不连接该程序。

-mrelax

指定链接器应该执行松弛优化阶段，缩短分支、函数调用和绝对内存寻址。该选项仅当在最后连接阶段的命令行中使用时才有效。而且，该选项将使符号调试无效。

### 21.2.22 NS32K 选项

如下选项是为32000系列定义的。这些默认值依赖于编译程序在配置时所选择的32000型号。

-m32032

为32032生成代码。当编译程序配置为32032和基于32016的系统时这是默认选项。

-m32081

生成包含32081浮点指令的代码。对所有系统该选项均是默认的。

参见选项-m32381。

-m32332

为32332生成代码。当为基于32332的系统配置编译程序时这是默认选项。

-m32381

生成包含32381浮点指令的代码。它同时设置了选项-m32081。32381仅与32332和32532 CPU兼容。对于pc532-netbsd配置这是默认选项。

-m32532

生成32532的代码。当为基于32532的系统配置编译程序时，这是默认选项。

-mbitfield

生成使用位字段指令的代码。除了PC532之外，该选项在所有平台上均是默认的。

参见选项-mnobitfield。

-mhimem

生成可以加载在512MB内存之上的代码。

许多NS32000系列寻址模式使用的偏移达到512MB。如果地址超过了512MB，就不能再使用由0起始的偏移了。该选项可以由操作系统和ROM代码使用。

参见选项-mnosb。

-mieee-compare

指定编译程序使用IEEE浮点比较操作。这些比较操作将正确地处理无序比较的情况。

但是，所需要的内核支持可能并不存在。可以使用-mno-ieee-compare 禁止该选项。

-mmulti-add

尝试生成 multiply-add 浮点指令 polyF 和 dotF。该选项仅当选项-m32381 有效时才可用。

使用这些指令要求修改寄存器分配，通常这会对系统性能具有负面影响。该选项应当仅在编译的代码非常频繁地使用 multiply-add 指令时才会用到。

参见选项-mnomulti-add。

-mnobitfield

指定不使用位字段指令。在某些机器上，使用移位和掩码操作可能更快。对 PC532 系统来说这是默认选项。

参见选项-mbitfield。

-mnohimem

假设代码将被加载到虚地址空间的第一个 512MB 区内。在所有平台上这均是默认选项。

-mnomulti-add

指定不要尝试生成 multiply-add 指令 polyF 和 dotF。这是默认选项。

参见选项-mmulti-add。

-mnoregparam

指定不在寄存器中传递任何参数。在所有目标机上这均为默认选项。

参见选项-mrtd 和-mregparam。

-mnosb

寄存器 sb 不可用或运行时系统没有将其初始化为 0。除 pc532-netbsd 外，该选项在其他所有目标机上均为默认的。无论何时使用了选项-mhimem 或-fpic，该选项均被设置。

-mregparam

使用一种不同的调用规范，这种规范会用寄存器传递前两个参数。该调用规范与 UNIX 系统所使用的规范不兼容，因此如果需要调用由 UNIX 编译程序编译的库时不能使用该选项。

参见选项-mrtd 和-mnoregparam。

-mrtd

使用一种不同的函数调用规范，这种规范将对具有固定数目参数的函数使用 ret 指令，在其返回时弹出相应的参数。

该调用规范与 UNIX 中使用的规范不兼容，当你需要调用由 UNIX 编译程序编译的库

时不能使用该选项。而且，必须提供所有具有可变参数数目的函数原型；否则，可能会生成不正确调用这些函数的代码。

如果使用过多的参数调用函数，会导致不正确的代码。通常，过多的参数可以被忽略。

该选项借用了 680x0 的 rtd 指令的名字。

参见选项-mregparam 和-mnoregparam。

-msb

允许编译程序将寄存器 sb 用做索引寄存器，并始终加载为 0。在 pc532-netbsd 目标机上这是默认选项。

参见选项-mnosb。

-msoft-float

生成包含浮点库调用的输出。但是，包含被调函数的库可能并不可用。

### 21.2.23 PDP-11 选项

下列选项是为 PDP-11 定义的。

-mabshi

使用 abshi2 模式。这是默认选项，但可以使用选项-mno-abshi 逆转。

-mbranch-cheap

指定不要假设分支的代价很高。这是默认选项。

参见选项-mbranch-expensive。

-mbranch-expensive

假设分支的代价很高。该选项仅用于生成代码时的试验。

参见选项-mbranch-cheap。

-m10

生成 PDP-11/10 的代码。

-m40

生成 PDP-11/40 的代码。

-m45

生成 PDP-11/45 的代码。这是默认选项。

-mac0

使用 ac0（UNIX 汇编器语法中的 fr0）返回浮点结果。默认选项是-mno-ac0，它将使用内存返回浮点结果。

-mbcopy
指定不要使用内嵌 movstrhi 模式进行内存复制。
参见选项-mbcopy-builtin。

-mbcopy-builtin
使用内嵌 movstrhi 模式进行内存复制。这是默认选项。
参见选项-mbcopy。

-mdec-asm
使用 DEC 汇编器语法。当配置为非 pdp11-*-bsd 目标机的其他目标机时，这是默认选项。

-mfloat32
将浮点数据定义为 32 位。也可以通过-mno-float64 使用该选项。

-mfloat64
将浮点数据定义为 64 位。这是默认选项。也可以通过-mno-float32 使用该选项。

-mfpu
使用硬件 FPP 浮点操作。这是默认选项。（不支持 PDP-11/40 上的 FIS 浮点操作。）

-mint16
定义 int 数据类型为 16 位。这是默认选项。也可以通过-mno-int32 使用该选项。

-mint32
定义 int 数据类型为 32 位。也可以通过-mno-int16 使用该选项。

-msoft-float
禁止使用硬件浮点操作。

-msplit
为具有分离式 I&D 高速缓存的系统生成代码。默认选项是-mno-split，它将为没有分离式 I&D 高速缓存的系统生成代码。

-munix-asm
使用 UNIX 汇编器语法。当编译程序被配置为 pdp11-*-bsd 时这是默认选项。

### 21.2.24　RS/6000 和 PowerPC 选项

下列选项是为 IBM RS/6000 和 PowerPC 定义的。

GCC 支持 RS/6000 和 PowerPC 这两种相关指令集体系。Power 指令集是由原 RS/6000 系统使用的 RIOS 芯片组支持的指令。PowerPC 指令集是 Motorola MPC5xx，MPC6xx，

MPC8xx 微处理器和 IBM 4xx 微处理器的体系。

这两种体系均不是彼此的子集。但是，它们均支持一大类通用指令子集。在支持 Power 体系的处理器上还包含 MQ 寄存器。

-mabi=altivec

扩展当前的 ABI 以增加 AltiVec ABI 扩展。这并不改变默认的 ABI；相反，它将在当前的 ABI 中增加 AltiVec ABI 扩展。如果需要禁止该扩展，可以使用选项-mabi=no-altivec。

-mabi=spe

扩展当前的 ABI 以增加 SPE ABI 扩展。这并不改变默认的 ABI 而是增加 SPE ABI 扩展。如果需要禁止这些扩展，可以使用选项-mabi=no-spe。

-misel

允许生成 ISEL 指令。该选项也可以写作-misel=yes。如果需要禁止 ISEL 指令，可以使用选项-misel=no。

-mads

在嵌入式 PowerPC 系统中，该选项假设起始模块是 crt0.o，而对标准 C 库是 libads.a 和 libc.a。

-maix-struct-return

返回内存中的所有结构（根据 AIX ABI 的规定）。

-maix32

这是默认选项。该选项将禁止 64 位 ABI 并隐含使用选项-mno-powerpc64。

参见选项-maix64。

-maix64

允许 64 位 AIX ABI 及其调用规范，它将包含 64 位指针、64 位 long 数据类型以及支持它们的基础结构。该选项同时隐含使用选项-mpowerpc64 和-mpowerpc。

参见选项-maix32。

-maltivec

该选项允许使用直接操作 AltiVec 指令集的内建函数。可能还需要设置选项-mabi=altivec 将当前的 ABI 调整为带 AltiVec ABI 的扩展。如果需要禁止这些内建函数的使用，可以使用选项-mno-altivec。

-mbig

在 System V.4 和嵌入式 PowerPC 系统中，该选项将为 big endian 模式下的处理器编译代码。参见选项-mlittle。

-mbig-endian

与选项-mbig 相同。

-mbit-align

在 System V.4 和嵌入式 PowerPC 系统中，该选项强制包含位字段的结构和联合在位字段的基础类型上对齐。这是默认选项。如果不需要这种强制对齐，可以使用选项-mno-bit-align。

例如，默认情况下一个仅包含了 8 个长度为 1 的无符号位字段的结构将对齐于 4 字节边界，并且具有 4 字节的大小。通过使用选项-mno-bit-align，该结构将对齐于 1 字节边界，并且具有 1 个字节的大小。

-mcall-aix

在 System V.4 和嵌入式 PowerPC 系统中，该选项将编译的代码使用的调用规范类似于 AIX 中使用的调用规范。如果配置 GCC 使用 powerpc-*-eabiaix，这是默认选项。

-mcall-gnu

在 System V.4 和嵌入式 PowerPC 系统中，该选项将为基于 Hurd 的 GNU 系统编译代码。

-mcall-linux

在 System V.4 和嵌入式 PowerPC 系统中，该选项将为 Linux 操作系统编译代码。

-mcall-netbsd

在 System V.4 和嵌入式 PowerPC 系统中，该选项将为 NetBSD 操作系统编译代码。

-mcall-solaris

在 System V.4 和嵌入式 PowerPC 系统中，该选项将为 Solaris 操作系统编译代码。

-mcall-sysv

在 System V.4 和嵌入式 PowerPC 系统中，该选项将编译出的代码会符合 System V 应用二进制接口——PowerPC 处理器增补在 1995 年 3 月的草案。除非用 powerpc-*-eabiaix 配置 GCC，否则这是默认选项。

-mcall-sysv-eabi

同时使用选项-mcall-sysv 和-meabi。

-mcall-sysv-noeabi

同时使用选项-mcall-sysv 和-mno-eabi。

-mcpu=type

为所指定的机器 type 指定体系结构类型、寄存器的使用、助记符的选择以及指令调度参数。可识别的机器类型名称有 rios，rios1，rsc，rios2，rs64a，601，602，603，603e，604，

604e，620，630，740，7400，7450，750，power，power2，powerpc，403，505，801，821，821，860和common。

选项-mcpu=common 将为通用处理器生成代码，这样的代码可以运行在任一种 Power 或 PowerPC 处理器上。GCC 将只使用这些体系中指令的通用部分，而且不会使用 MQ 寄存器。GCC 将为调度假设一种通用处理器模型。

选项-mcpu=power，-mcpu=power2，-mcpu=powerpc 和-mcpu=powerpc64 会指定通用 Power，Power2，纯 32 位 PowerPC（即，非 MPC601）和 64 位 PowerPC 机器类型，以及为调用假设使用一种通用处理器模型。

其余的选项将生成专用于一种处理器的指令。在某个选项下所生成的代码将在相应处理器上运行得最快，并且可能根本无法在其他处理器上运行。

表 21-6 列出了可以由-mcpu 选项设置的其他选项。选择表中第一列所列的数值将隐含着第二列所给出的选项。

**表 21-6 设置-mcpu 选项可用的其他选项设置**

| -mcpu 选项 | 可应用选项 |
|---|---|
| common | -mno-power, -mno-powerc |
| power,power2, rios1, rios2, rsc | -mpower, -mno-powerpc, -mno-new-mnemonics |
| powerpc, rs64a, 602, 603, 603e, 604, 620,630, 740, 7400, 7450, 750, 505 | -mno-power, -mpowerpc, -mnew-mnemonics |
| 601 | -mpower, -mpowerpc, -mnew-mnemonics |

-meabi

在 System V.4 和嵌入式 PowerPC 系统中，该选项继承了符号嵌入式应用二进制接口(eabi)，它是一套对 System V.4 规范的修改。栈对齐于 8 字节边界，函数 main()将调用函数__eabi()来建立 eabi 环境，并且选项-msdata 可以使用寄存器 r2 和 r13 来指定两个独立的小数据区。

使用选项-mno-eabi 会将栈对齐于 16 字节边界，但不会在 main()中调用初始化函数，而且造成选项-msdata 仅用寄存器 r13 来指定一个小数据区。

如果用 powerpc*-*-eabi*选项配置 GCC，则选项-meabi 默认是打开的。

-memb

在嵌入式 PowerPC 系统中，该选项会设置 ELF 标志头的 PPC_EMB 位，以标明需要使用 eabi 扩展的重定位操作。

-mfull-toc

这是默认选项。编译程序将至少为程序中每个惟一的非自动变量引用分配一个 TOC

（table of content）表项。而且，TOC 将包含浮点常量。TOC 的最大尺寸是 16 384 个表项。

为了减少 TOC 保存的信息，参见选项-mno-fp-in-toc，-mno-sum-in-toc 和-mminimal-toc。

-mfused-madd

生成使用浮点乘和累加指令的代码。如果使用了硬件浮点操作，则默认选择该选项。要取消该选项，可以使用-mno-fused-madd。

-mhard-float

生成不使用浮点寄存器集的代码。

参见选项-msoft-float。

-minimal-toc

该选项可以用在链接器的错误信息中，用来说明可用 TOC（table of content）空间溢出的情况。该选项将减少 TOC 空间的使用，每一个文件只占用一个 TOC 表项。所生成的代码要稍微大一些并且较慢，但它会大大减少 TOC 的尺寸。

参见选项-mfull-toc。

-mlittle

在 System V.4 和嵌入式 PowerPC 系统中，该选项将为 little endian 模式的处理器编译代码。

参见选项-mbig。

-mlittle-endian

参见选项-mlittle。

-mlongcall

指定所有的函数调用均通过指令完成，以便位于当前位置 64MB（67 108 864 字节）以外的函数可以被调用。可以使用 shortcall 函数属性或#pragam longcall(0)来取消该设置。可以使用选项-mno-longcall 来禁止使用长调用。

某些链接器能够探测超出范围的调用并迅速生成胶合代码。在这些系统中，长调用就没有必要而且会比较慢。正如所描述的那样，AIX 链接器可以完成这种操作，PowerPC/64 的 GNU 链接器也可以。正在计划将该功能增加到 32 位 PowerPC 系统的 GNU 链接器中。

-mmultiple

生成的代码会使用“加载多字”指令和“存放多字指令”。Power 系统将默认生成这些指令，PowerPC 系统不会生成这些指令。

如果需要禁止生成这些指令，可以使用选项-mno-multiple。

不要在 little endian 模式的 PowerPC 系统中使用该选项，因为那些指令在处理器处于

little endian 模式时不工作。PPC740 和 PPC750 除外，它们允许在 little endian 模式下使用这些指令。

-mmvme

在嵌入式 PowerPC 系统中，该操作假设启动模块是 crt0.o，标准 C 库是 libmvme.a 和 libc.a。

-mnew-mnemonics

选择为 PowerPC 体系定义的汇编助记符。如果没有为所选择的体系定义助记符，该选项将被忽略。

参见选项-mold-mnemonics。

-mno-fp-in-toc

当链接器错误消息说明可用的 TOC 空间不足时，可使用该选项。通过禁止包含浮点常量，该选项可以减少 TOC 空间的占用。

参见选项-mfull-toc。

-mno-sum-in-toc

当链接器错误消息说明可用的 TOC 空间不足时，可使用该选项。通过生成运行时计算地址与常量的和（而不是将该和存在 TOC 中）的代码，该选项可以减少对 TOC 空间的占用。

参见选项-mfull-toc。

-mold-mnemonics

选择为 Power 体系定义的汇编语言助记符。如果没有为所选体系定义助记符，该选项将被忽略。

参见选项-mnew-menmonics。

-mpe

生成支持 IBM RS/6000 SP 并行环境(PE)的代码，它将特殊的启动代码连接到应用中，该应用会使用消息传递。

系统必须在标准位置（/usr/lpp.poe/）安装 PE 或者使用选项-specs 重新指定规范文件的目录位置。并行环境不支持线程，因此选项-mpe 和选项-pthread 不兼容。

-mpower

生成只在 Power 体系才有的指令来利用 MQ 寄存器。如果 GCC 安装指定该配置，可以使用-mn-power 来取消这一配置。

通过使用选项-mno-power 和-mno-powerpc，GCC 将只使用这两种体系中共有的指令子集，再加上一些特殊的 AIX 通用模式调用，但不使用 MQ 寄存器。同时使用选项-mpower

和-mpowerpc 将允许 GCC 使用两种体系中的任一指令，并且允许使用 MQ 寄存器。对 Motorola MPC601 要使用该选项。

-mpower2

生成在 Power2 体系但不在原始的 Power 体系中的指令。设置该选项将同时设置选项-mpower。如果 GCC 安装使用了该配置，可以使用-mno-power2 取消该选项。

-mpowerpc

生成仅在 PowerPC 体系 32 位子集中才有的指令。如果 GCC 安装使用了这一配置，可以使用-mno-powerpc 取消该选项。

参见选项-mpower。

-mpowerpc-gpopt

生成的代码使用通用组中可选的 PowerPC 体系指令，这包括浮点开平方根指令。设置这一选项将同时设置选项-mpowerpc。如果 GCC 安装指定了这一配置，可用使用-mno-powerpc-gopt 取消该选项。

-mpowerpc-gfxopt

生成的代码使用图形组中的可选 PowerPC 体系指令，这包括浮点选择指令。设置这一选项将同时设置选项-mpowerpc。如果 GCC 安装使用了这一配置，可以使用-mno-powerpc-gfxopt 取消该选项。

-mpowerpc64

生成完整的 PowerPC64 体系所具有的额外的 64 位指令，并按 64 位双字处理 GPR 寄存器。默认选项是-mno-powerpc64。

-mprototype

在 System V.4 和嵌入式 PowerPC 系统中，该选项将假设所有对变长参数函数的调用均具有适当的原型。

如果没有该选项，或使用了选项-mno-prototype，编译程序必须在每一个没有原型的调用前插入一条指令，或清除条件码寄存器(CR)的第 6 位来标明是否利用浮点寄存器传递浮点值。如果使用了-mprototype，只有对具有变长参数函数的调用才会设置或清除这一位。

-mregnames

在 System V.4 和嵌入式 PowerpPC 系统中，该选项将用符号的形式在汇编语言中输出寄存器的名字。可以使用-mno-regnames 来取消该选项。

-mrelocatable

在嵌入式 PowerPC 系统中，该选项将生成允许程序在运行时被重定位到不同地址的代

码。如果你对某一模块使用了该选项，所有连接到的目标文件都必须使用选项-mrelocatable或-mrelocatable-lib进行编译。默认选项是-mno-relocatable。

-mrelocatable-lib

在嵌入式PowerPC系统中，该选项生成的代码允许程序在运行时被重定位到不同地址。使用该选项编译的模块可以连接到没有使用选项-mrelocatable 和-mrelocatable-lib 编译的模块中，而且它们可以连接到使用选项-mrelocatable 编译的模块中。

-msdata=setting

在 System V.4 和嵌入式 PowerPC 系统中，使用选项-msdata=eabi 会将小的已初始化的全局数据和静态数据放置在.sdata2 区中，寄存器 r2 指向该区。而且，小的已初始化的非常量全局数据和静态数据将被放置在.sdata 区中，寄存器 r13 指向该区。而且，小的已初始化的全局数据和静态数据将被放置在.sbss 区中，该区和.sdata 区相连。选项-msdata=eabi 将设置选项-memb，但与选项-mrelocatable 不兼容。

在 System V.4 和嵌入式 PowerPC 系统中，使用选项-msdata=sysv 会将小的全局数据和静态数据放置在.sdata 区中，寄存器 r13 指向该区。它同时会将小的未初始化的全局数据和静态数据放置在.sbss 区，它与.sdata 区相连。选项-msdata=sysv 与选项-mrelocatable 不兼容。

在 System V.4 和嵌入式 PowerPC 系统中，使用选项-msdata=none（也可以写作-mno-sdata）会将所有已初始化的全局数据和静态数据放置在.data 区，并将所有未初始化的数据放置在.bss 区。

在 System V.4 和嵌入式 PowerPC 系统上，与选项-meabi 一起使用选项-msdata=default（也可以简单地写作-msdata）会像选项-msdata=eabi 那样编译代码。如果没有使用选项-meabi，将按选项-msdata=sysv 那样编译代码。

-msdata-data

在 System V.4 和嵌入式 PowerPC 系统中，该选项将小的全局数据和静态数据放置在.sdata 区中。它还会将小的未初始化的全局数据和静态数据放置在.sbss 区。但是，该选项不用寄存器 r13 指向小数据区。

如果没有使用其他-msdata 选项，该选项是默认选项。

-msim

在嵌入式 PowerPC 系统中，该选项假设启动模块是 sim-crt0.o，标准 C 库为 libsim.a 和 libc.a。对 powerpc-*-eabisim 配置而言，这是默认选项。

-msoft-float

生成使用软件浮点仿真的代码。

参见选项-mhard-float。

-mstrict-align

在 System V.4 和嵌入式 PowerPC 系统中，该选项生成的代码将假设未对齐的内存引用将由系统处理。这是默认选项。如果不想建立这样的假设，可以使用选项-mno-strict-align。

-mstring

生成的代码使用“加载字串”指令和“存放字串字”指令来保存多个寄存器和执行小数据块移动。Power 系统将默认生成这些指令，但在 PowerPC 系统中则不会生成这些指令。

如果需要禁止生成这些指令，可以使用选项-mno-string。

不要在 little endian 模式的 PowerPC 系统中使用该选项，这是因为当处理器在 little endian 模式时那些指令无法工作。PPC740 和 PPC750 除外，它们允许在 little endian 模式下使用这些指令。

-msvr4-struct-return

利用寄存器返回小于 8 字节的结构（按照 SVR4 ABI 的规定）。

-mtoc

在 System V.4 和嵌入式 PowerPC 系统中，该选项假设寄存器 2 包含一个由程序实现的指向全局区的指针。可以使用选项-mno-mtoc 取消这一假设。

-mtune=type

为所指定的机器 type 指定指令调度参数，但不会设置体系类型、寄存器使用或是助记符的选项，而这些选项-mcpu 都会设置。由选项-mcpu 所识别的同一组值可以由选项-mtune 使用。如果同时使用了两个选项，所生成的代码将使用由-mcpu 设置的体系结构、寄存器和助记符，但调度参数由-mtune 设置。

-mupdate

生成使用加载或存储指令（这些指令更新基地址寄存器为所计算的内存位置）的代码。这是默认选项。

如果需要禁止生成这种代码，可以使用选项-mno-update，它将在更新栈指针和更新先前栈之间打开一个小的时间窗口，跨越中断或信号遍历堆栈框架的代码可能会获取到被破坏的数据。

-mvxworks

在 System V.4 和嵌入式 PowerPC 系统中，该选项指示你将为 VxWorks 系统编译代码。

-mwindiss

指定为 WindISS 模拟环境编译代码。

-mxl-call

使用特定 AIX 编译程序的规范在栈上传递浮点参数。要禁止该选项，可以使用-mno-xl-call。

在 AIX 系统中，除了参数 FPR 外，到原型化函数的浮点参数还通过寄存器保存区(RSA)之外的区域进行传递。AIX 调用规范经过了扩展，但最初没有文档描述如何处理 K&R C 中的一个不明显的例子，即用参数的指针调用函数，但带有的参数比声明的要少。AIX XL 编译程序中，当子例程没有优化编译时，不能操作 RSA 中的浮点参数，要由栈访问。由于总是在栈上保存浮点参数非常低效而且很少是必须的，该选项默认情况下并没有打开，只有在由 AIX XL 编译程序编译的子例程没有优化时才是必要的。

-myellowknife

在嵌入式 PowerPC 系统中，该选项假设启动模块是 crt0.o 且标准 C 库是 libyk.a 和 libc.a。

-G number

在嵌入式 PowerPC 系统中，该选项将少于或等于 number 字节的全局和静态数据项放置在小数据区或 bss 区中（而不是正常的数据区或 bss 区）。number 的默认数值为 8。

该选项也会传递给链接器，因此所有的模块都应该使用同一 number 值进行编译。

-pthread

用 pthreads 库增加对多线程的支持。该选项同时设置预处理程序和链接器的标志。

### 21.2.25 RT 选项

下列选项是为 IBM RT PC 定义的。

-mcall-lib-mul

为整数乘（integer multiply）生成到 lmul$$的调用。

参见选项-min-line-mul。

-mfp-arg-in-fpregs

使用与 IBM 调用规范不兼容的调用序列，IBM 的调用规范要求浮点参数要利用浮点寄存器进行传递。如果使用该选项，头文件 stdarg.h 将无法处理浮点操作数。

参见选项-mfp-arg-in-gregs。

-mfp-arg-in-gregs

使用正常的调用规范传递浮点参数。这是默认选项。

参见选项-mfp-arg-in-fpregs。

-mfull-fp-blocks

生成全尺寸的浮点数据块，包含由 IBM 建议的最小空白区。这是默认选项。

参见选项-mminimum-fp-blocks。

-mhc-struct-return

将多于一个字的结构保存在内存而非寄存器中。这将提供与 MetaWare HighC (hc)编译程序的兼容性。使用选项-fpcc-struct-return 来保持与 Portable C 编译程序(pcc)间的兼容性。

-min-line-mul

为整数乘使用内嵌代码序列。这是默认选项。

参见选项-mcall-lib-mul。

-mminimum-fp-blocks

指定在浮点数据块中不包含额外的空白区。其结果是更小的代码但执行较慢，这是因为必须要动态分配空白区。

参见选项-mfull-fp-blocks。

-mnohc-struct-return

如果方便的话，在寄存器中返回大于一个字的某些结构。这是默认选项。为了兼容于 IBM 所提供的编译程序，可以使用选项-fpcc-struct-return 或-mhc-struct-return。

### 21.2.26　S/390 和 zSeries 选项

如下选项是为 S/390 和 zSeries 体系定义的。

-m31

生成符合 S/390 Linux 规范的代码。对于 s390 目标机来说，默认选项是-m31，而 s390x 目标机的默认选项则是-m64。

-m64

生成符合 zSeries ABI Linux 规范的代码。这将允许 GCC 生成 64 位指令。对于 s390 目标机来说，默认选项是-m31，而 s390x 目标机的默认选项则是-m64。

-mbackchain

生成的代码在指向调用者栈的堆栈框架中维护一个显式的回退链，这是调试所需要的。这是默认选项，但可以使用-mno-backchain 逆转该选项。

-mhard-float

使用硬件浮点指令和寄存器操作浮点数。编译程序将生成 IEEE 浮点指令。这是默认选项。

参见选项-msoft-float。

-mdebug

在编译时调用额外的调试信息。默认选项是-mno-debug，它表明无需打印调试信息。

-mmvcle

生成使用 mvcle 指令来执行块移动的代码。默认选项是-mno-mvcle，它将使用一个 mvc 循环。

-msmall-exec

生成使用 bras 指令进行子例程调用的代码。仅当总执行尺寸不超过 64KB 时该选项才有效。默认选项是-mno-small-exec，它将使用 basr 指令，且没有尺寸限制。

-msoft-float

指定不要使用硬件浮点指令和寄存器完成浮点操作。系统会使用 libgcc.a 中的函数完成浮点操作。

参见选项-mhard-float。

### 21.2.27 SH 选项

下列选项是为 SH 实现而定义的。

-m1

为 SH1 生成代码。

-m2

为 SH2 生成代码。

-m3

为 SH3 生成代码。

-m3e

为 SH3e 生成代码。

-m4-nofpu

为不具备浮点单元的 SH4 生成代码。

-m4-single-only

为具有只支持单精度算法的浮点单元的 SH4 生成代码。

-m4-single

为 SH4 生成代码，假设浮点单元默认处于单精度模式。

-m4
为 SH4 生成代码。

-mb
为处于 big endian 模式的处理器生成代码。
参见选项-ml。

-mbigtable
在 switch 表中生成 32 位偏移。默认选项是使用 16 位偏移。

-mdalign
将 double 对齐于 64 位边界。这将改变调用规范，因此标准 C 库中的某些函数可能无法工作，除非使用选项集-mdalign 重新编译它。

-mfmovd
允许使用指令 fmovd。

-mhitachi
为 Hitachi 定义的调用规范进行编译。
参见选项-mnomacsave。

-mieee
增加浮点代码的 IEEE 遵从程度。

-misize
输出汇编代码中的指令尺寸和位置。

-ml
为处于 little endian 模式的处理器生成代码。
参见选项-mb。

-mnomacsave
标志 MAC 寄存器已被调用破坏，即使已经使用了选项-mhitachi。

-mpadstruct
该选项已废止。它将结构填充为 4 字节的倍数，但它与 SH ABI 不兼容。

-mprefergot
当生成位置独立代码时，生成使用 Global Offset 表（而不是 Procedure Linkage 表）的函数调用。

-mrelax

缩短链接时的某些地址的引用。该选项设置链接器选项-relax。

-mspace

优化代码的尺寸而非速度。选项-Os 将设置该选项。

-musermode

在修整一个启动代码后，生成到无效指令高速缓存项的库函数调用。当目标是sh-*-linux*时这是默认选项。

该库函数调用并不假设它可以写到整个内存地址空间中。

### 21.2.28 SPARC 选项

下面的选项是为 Sun 微系统公司的 SPARC 处理器定义的。

-m32

在 64 位环境中的 SPARC V9 处理器上，该选项设置 int，long 和指针数值为 32 位。

-m64

在 64 位环境中的 SPARC V9 处理器上，该选项设置 int 为 32 位，但 long 和指针数值为 64 位。

-mapp-regs

生成的代码使用全局寄存器 2 到 4，这些寄存器是 SPARC SVR4 ABI 为应用程序保留的。这是默认选项。

为了严格地遵从 SVR4 ABI（要付出一定的性能代价），可以使用选项-mno-app-regs。库和系统软件应该使用选项-mno-app-regs 进行编译。

-mcmodel=setting

在 64 位环境中的 SPARC V9 处理器上，该选项将为指定的代码模型生成指令。表 21-7 列出了代码模型的设置。

表 21-7 SPARC V9 的模式设置

| 设置 | 代码模式 |
|---|---|
| medlow | Medium/Low 代码模式是指程序必须连接到地址空间的低 32 位上。指针是 64 位的。程序可以是静态连接的，也可是动态连接的 |
| medmid | Medium/Middle 代码模式是指程序必须连接到地址空间的低 44 位上，文本部分的尺寸必须小于 2G 字节，而数据部分必须在这文本部分的 2G 范围之内。指针是 64 位的 |

（续表）

| 设置 | 代码模式 |
|---|---|
| medany<br>embmedany | Medium/Anywhere 代码模式是指程序可以连接到地址空间的任意位置，文本部分的尺寸必须小于 2G 字节，而数据部分必须在这文本部分的 2G 范围之内。指针是 64 位的<br>对于嵌入式系统，Medium/Anywhere 代码模式会假设 32 位的文本部分和 32 位的数据部分，都可在任意位置开始（这在连接时决定）。寄存器%g4 指向数据部分的基址。指针是 64 位的。程序是静态连接的。不支持 PIC |

-mbroken-saverestore

在 SPARCLET 处理器上，该选项生成的代码将不使用非平凡形式的 save 和 restore 指令。

使用该选项的原因是，较早版本的 SPARCLET 处理器在使用这些参数时，不能正确地处理 save 和 restore 指令，但当没有参数时则能够正确处理。一条没有参数的 save 指令将递增当前的窗口指针，但不会分配新的堆栈框架，这是因为它假设（像中断处理函数一样）窗口溢出陷阱处理函数能够正确地处理这一情况。

-mcpu=type

根据 type 所指定的机器名设置指令集、寄存器集和指令调度参数。支持的 type 值有 v7，cypress，v8，supersparc，sparclite，hypersparc，sparclite86x，f930，f934，sparclet，tsc701，v9 和 ultrasparc。

默认的指令调度参数是用来选择体系结构（而不是实现）的数值。表 21-8 列出了支持这些数值的体系结构和实现。

表 21-8　支持各种体系结构的实现方法

| 体系结构 | 实现方法 |
|---|---|
| v7 | cypress |
| v8 | supersparc, hypersparc |
| sparclite | f930, f934, sparclite86x |
| sparclet | tsc701 |
| v9 | ultrasparc |

-mcypress

这是默认选项。编译程序针对 Cypress CY7C602 芯片进行优化，该芯片曾用于 SparcStation/SparcServer 3xx 系列中。对于较早的 SparcStation 1，2，IPX 等，该选项也同样适用。

该选项已废止并将在以后的编译程序版本中去除。可以使用-mcpu=cypress 替代该选项。

-mfaster-structs

该选项强制结构对齐于 8 字节边界。该选项将打开结构赋值的复制序列中使用的 ldd 和 std 指令对。该指令将两次替代多个 ld 和 st 指令对。

但使用这一变更的对齐特性将直接违反 SPARC ABI，因此该选项仅能在所产生代码不会违反 ABI 准则的目标机上使用。

默认选项是-mno-faster-structs。

-mflat

该选项指定编译程序不应该生成保存和恢复指令，而应该使用一种“平展（flat）”或“单寄存器窗口（single register window）”调用规范。默认选项是-mno-flat。

该模型将使用%i7 作为栈指针，而且与正常的寄存器窗口模型兼容。两种不同模型的代码可以混合调用。如果需要，局域寄存器和输入寄存器（0-5）仍然保存在栈中。

通过使用选项-mno-flat，编译程序将发出保存和恢复指令（除了叶子函数），这就是操作的正常模式。

-mfpu

这是默认选项，它将生成包含硬件浮点指令的代码。

为了生成包含浮点操作库调用的代码，需要使用选项-mno-fpu。并不是所有的 SPARC 平台均有实现该功能的库。目标机本地编译程序使用的库也可以用于该目的，但它们不能用于交叉编译。嵌入式目标 sparc-*-aout 和 sparclite-*-*提供软件浮点操作支持。

选项-mfpu 也可以写作-mhard-float。

参见选项-msoft-float。

-mhard-float

参见选项-mfpu。

-mhard-quad-float

使用该选项输出的代码将包含 4 字(long double)浮点指令。

-mlittle-endian

在 SPARCLET 处理机上以及 64 位环境中的 SPARC V9 处理机上，该选项将生成运行于 little endian 模式的代码。

-mlive-g0

在 SPARCLET 处理机上，该选项将寄存器%g0 当作普通寄存器使用。如果必要的话，GCC 将继续 clobber 它，但不会假设其将始终返回 0。

-msoft-float

除了所生成的代码中调用规范有所变化，它与选项-mno-fpu 相同。这意味着只有在编译整个程序时用到该选项，才能使用它。而且必须使用选项-msoft-float 编译和 GCC 一同发行的 libgcc.a，才能保证所有代码都一致。

-msoft-quad-float

这是默认选项。使用该选项所输出的代码将包含 4 字(long double)浮点指令的库调用。

不存在选项-mhard-quad-float，这是因为没有 SPARC 实现具有对 4 字浮点操作的硬件支持。它们均会调用一个陷阱处理函数，并由该陷阱处理函数仿真指令的功能。由于陷阱处理函数的开销，相比调用 ABI 库例程而言慢得多。这也就是-msoft-quad-float 是默认选项的原因。

-msparclite

该选项选择 SPARC 体系中的一个变体。除非编译程序被显式地配置为 Fujitsu SPARClite，否则 GCC 将生成 SPARC 体系的 v7 变体代码。SPARClite 具有整数乘、整数除和扫描(ffs)指令，但 SPARC V7 没有。

该选项已废止并将在以后的编译程序版本中去除。可以使用选项-mcpu=sparclite 替代。

-mstack-bias

在 64 位环境中的 SPARC V9 处理器上，该选项将假设栈指针（如果有的话，还包括栈指针）偏移-2047，当进行堆栈框架引用时必须要加上该偏移。默认选项是-no-stack-bias，默认不存在这类偏移。

-msupersparc

编译程序为 SparcStation 10，1000 和 2000 系列中的 SuperSparc CPU 优化代码。该选项将使用完整的 SPARC V8 指令集。

该选项已废止并将在以后的编译程序版本中删除。可以使用选项-mcpu=supersparc 替代。

-mtune=*type*

为所指定的机器类型设置指令调度参数，但不指明所用的指令集或寄存器集，只有使用了选项-mcpu 才会设置这些特性。

用于选项-mcpu 的参数值同样也可用于选项-mtune。惟一可用的值就是指定某一特殊 CPU 实现的值（即表 21-8 中第二列所列出的名字）。

-munaligned-doubles

该选项强制假设这样的情况，即当 double 被包含在其他类型或具有绝对地址时应当对齐于 8 字节边界上。所有其他 double 均默认对齐于 4 字节边界。使用该选项可以避免某些

少见的与其他编译程序所生成代码的兼容性问题。由于该选项会造成性能损失，因此不是默认选项，尤其是对浮点代码。

默认选项是-mno-unaligned-doubles，它假设所有的 double 类型均对齐于 8 字节边界。

-mv8

该选项选择某个 SPARC 体系的变体。该选项生成 SPARC V8 代码。与 V7 代码的惟一不同是，编译程序会生成只存在于 SPARC V8 而不存在于 SPARC V7 的整数乘和整数除指令。

该选项已废止并将在以后的编译程序版本中去除。可以使用选项-mcpu=v8 替代。

### 21.2.29 System V 选项

如下是 System V Release 4 中的一些额外选项。

-G

创建共享目标代码。建议不使用该选项而使用选项-symbolic 或-shared。

-Qn

避免在输出文件中增加任何.ident 指令。这是默认选项。

-Qy

在输出文件中使用.ident 汇编指令以标识编译程序所使用的每个工具的版本。

-Ym, *directory*

在 directory 中查找 M4 预处理程序。汇编器使用该选项。

-YP,directories

只在所指定的 directories 而不在其他目录查找选项-lname 所指定的库。

### 21.2.30 TMS320C3x/C4x 选项

下列选项是为 TMS320C3x/C4x 实现定义的。

-mbig

生成大内存模型代码。大内存模型是默认模型，并且需要在每次访问直接内存时，重新加载 DP 寄存器。

参见选项-msmall。

-mbig-memory

参见选项-mbig。

-mbk

允许将通用整数操作数分配到块计数寄存器 BK 中。如果要禁止这种分配，可以使用选项-mno-bk。

-mcpu=type

为指定的机器类型设置指令集、寄存器集和指令调度参数。其中 type 的合法值有 c30，c31，c32，c40 和 c44。默认值为 c40，它将生成 TMS320C40 的代码。

-mdb

允许使用递减和分支 DBcond(D)指令生成代码。对 C4x 系统，默认情况下该选项是打开的。可以使用选项-mno-db 关闭该选项。

由于在 C3x 上的最大重复计数为 2^23+1，因此在 C3x 上该选项默认是关闭的。注意，GCC 将试图逆转一个循环以便可以利用递减和分支指令，但如果循环体具有多于一条内存引用指令时，会放弃这种尝试。因此，具有循环计数递减的循环而又不能使用 RPTB 指令时可以生成更高效的代码。

-mdp-isr-reload

强制在中断服务例程（interrupt service routine，ISR）的入口处保存 DP 寄存器，重新加载指向数据区的指针并在 ISR 的出口处恢复数据。除非通过修改 DP 寄存器破坏了小规模内存模型，否则不需要该选项。

-mfast-fix

禁止生成用于修正 FIX 指令结果的额外代码。默认选项是-mno-fast-fix。

将浮点数转换为整数的 C3x/C4x FIX 指令将选择小于或等于浮点数的最近整数，而非离该浮点数最近的整数。因此，如果浮点数是负数时，结果可能会被错误取整，需要附加代码探测并修改这一错误。

-mloop-unsigned

允许非负迭代计数。默认选项是-mno-loop-unsigned。

由于 RPTS 和 RTPB（以及 C40 上的 DB）这些指令在检测迭代计数为负时终止，当使用这些指令时最大的迭代计数为 2^31+1。如果迭代计数为负，可能的迭代计数会超过^31+1。

参见选项-mrptb 和-mrpts。

-mmemparm

生成的代码可使用寄存器传递函数参数。默认情况下，参数尽可能通过寄存器传递，而非将参数压入栈。

参见选项-mregparm。

-mmpyi

在C3x系统中使用24位MPYI指令进行整数乘，而非默认地调用库函数以保证32位结果。可以使用选项-mno-mpyi取消该选项。

如果某个操作数是常量，乘法将使用移位和累加完成。如果C3x中没有使用该选项，平方操作将采用内嵌函数实现，而非库函数。

-mparallel-insns

允许生成并行指令。默认情况下，使用选项-O2时该选项被打开，但可以使用选项-mno-parallel-insns关闭该选项。

参见选项-mparallel-mpy。

-mparallel-mpy

如果同时也使用了选项-mparallel-insns，允许生成MPY||ADD和MPY||SUB并行指令。这些指令具有很强的寄存器限制，可能会“悲观化（pessimize）”大函数的代码生成。可以使用-mno-parallel-mpy关闭该选项。

-mparanoid

与选项-mdp-isr-reload相同。

-mregparm

生成的代码可使用栈传递函数参数。默认情况下，尽可能使用寄存器传递参数，而不使用栈传递参数。

参见选项-mmemparm。

-mrptb

允许使用RPTB指令生成重复块序列，以达到零代价循环。默认情况下选项-O2将打开该选项，但可以使用-mno-rptb关闭该选项。

RPTB构造函数仅在没有函数调用和跨循环边界跳转的最内层循环时才可用。由于保存和恢复RC，RS和RE寄存器的代价，嵌套使用RPTB并没有优势。

参见选项-mrpts和-mloop-unsigned。

-mrpts=number

允许使用单指令循环指令RPTS。默认选项是-mno-rpts。

如果循环块包含了一单独指令，而且可以保证循环计数少于所指定的number数，GCC将生成一条RPTS指令而非RPTB指令。如果没有指定number，即使无法在编译时确定循环计数系统仍将生成RPTS。

跟在RPTS后的被循环指令在每次迭代时无需从内存加载，因此可以释放CPU的总线。但由于该指令会屏蔽中断，默认情况下该选项是禁止使用的。

参见选项-mrptb 和-mloop-unsigned。

-msmall

为小规模内存模型生成代码。小规模内存模型假定所有的数据均可以放置在一个 64K 字的页面中。在运行时，DP 寄存器的内存必须设置为指向包含.bss 和.data 程序区的 64K 页面。

参见选项-mbig。

-msmall-memory

与选项-msmall 相同。

-mti

生成满足 TI 汇编器(asm30)的汇编器语法。通常是为了兼容由 TI C3x C 编译程序所生成的 API 而强制的。例如，long double 将会作为结构传递而非使用浮点寄存器传递。

### 21.2.31　V850 选项

如下选项是为 V850 实现定义的。

-mbig-switch

生成适用于大 switch 表的代码。只有在汇编器/连接程序编译 switch 表范围外的分支时，才会使用该选项。

-mep

优化这样一些基本代码块，它们 4 次或更多次地使用同一索引指针将地址复制到 ep 寄存器，并且它们使用的是较短的 sld 和 sst 指令。如果打开了优化选项，该选项就是默认的，但可以使用-mno-ep 关闭该选项。

-mlong-calls

将所有的调用均视为长调用。如果调用被认为是长调用，编译程序会始终将函数的地址加载到寄存器中并通过指针间接调用函数。

可以使用-mno-long-calls 禁止该选项。

-mprolog-function

在函数的起始和结束段使用外部函数保存和恢复寄存器。外部函数较慢，但如果不只一个函数需要保存同样的寄存器集，这样做会使用较少的代码空间。如果没有打开优化选项，这是默认选项，但可以通过-mno-prolog-function 关闭这一功能。

-msda=number

将尺寸小于或等于 number 字节的静态和全局变量放置在由 gp 所指引的小数据区。小数据区可以保存最多 64KB 数据。

参见选项-mtda 和-mzda。

-mspace

通过打开选项-mep 和-mprolog-function 尽可能地减少代码的尺寸。

-mtda=number

将尺寸小于或等于 number 字节的静态和全局变量放置在由 ep 所指引的小数据区。小数据区最多可存放 256 字节（当作为字节引用时为 128 字节）。

参见选项-msda 和-mzda。

-mv850

指定目标处理器是 V850。

-mzda=number

将静态变量和全局变量放置在内存的头 32KB 区域中，变量的尺寸为 number 字节或更少。

参见选项-msda 和-mtda。

### 21.2.32 VAX 选项

下面是为 DEC VAX 处理器定义的选项。

-mg

输出 VAX g-format 浮点数（而不是 d-format 浮点数）的代码。

-mgnu

假定使用的是 GNU 汇编器的跳转指令，并输出相应结果。

-munix

指定不要输出特定的跳转指令（例如 aobleq），VAX 的 UNIX 汇编器不能处理大范围跳转。

### 21.2.33 Xstormy16 选项

以下是为 Xstormy16 定义的选项。

-msim

选择适合模拟器的启动文件和连接程序脚本。

The Complete Reference

# 第 4 部分

# 附　录

# 附录 A

# GNU 通用公共许可证

GCC 编译程序是在 GNU 通用公共许可证（也被叫做 GNU GPL，或就叫做 GPL）之内的。由 GPL 保证的这种许可叫做 copyleft（版权所无）。简单的说，这就意味着任何人都有权利复制并使用该软件，但如果它被集成进产品，该产品就必须也是 GPL 许可证的。就是说，不能使用 GPL 软件将它转换成专有的软件。然而，并不限制任何人用 GCC 作为工具创建自己需要形式的软件。变成生成程序中的二进制位以及片断不要求该程序得到 GPL 许可。

GPL 的另一种方案是弱通用公共许可证（Lesser General Public License，LGPL）。该许可证以前叫做库 GPL，但这个名字由于可能产生误导就改变了——它满足一些库，但不会是所有库。LGPL 允许专有程序使用的库例程以及共享的和非静态连接的库。例子就是标准 C 库的 GNU 版本。

下面是 GPL 的文本。它用非常清楚的语言描述了许可的细节。文档的末尾是对处理过程的描述，遵照此过程可以将自己的软件放在 GPL 下。

## GNU GPL[1]

允许所有人复制和发布这一许可证原始文档的副本，但绝对不允许对它进行任何修改。

**导言**

大多数软件许可剥夺你共享和修改软件的自由。相比之下，GNU 通用公共许可力图保证你共享和修改自由软件的自由——保证软件对所有用户是自由的。通用公共许可适用于大多数自由软件协会的软件，以及由使用这些软件而承担义务的作者所开发的软件。（自由

---

1 Version2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

软件协会的其他软件受 GNU 库通用公共许可的保护）。你也可以将它应用到你的程序中。

当我们谈到自由软件（free software）时，我们指的是自由而不是价格。GNU 通用公共许可保证你有发布自由软件的自由（如果你愿意，可以对此项服务收取一定的费用）；保证你能收到源程序或者在你需要时能得到它；保证你能修改软件或将它的一部分用于新的自由软件；而且还保证你知道你能做这些事情。

为了保护你的权利，我们作出规定：禁止任何人不承认你的权利，或者要求你放弃这些权利。如果你修改了自由软件或者发布了软件的副本，这些规定就转化为你的责任。

例如，如果你发布了这样一个程序的副本，不管是收费的还是免费的，你必须将你具有的一切权利给予你的接受者；你必须保证他们能收到或得到源程序；并且将这些条款给他们看，使他们知道他们有这样的权利。

我们采取两项措施来保护你的权利。（1）给软件以版权保护。（2）给你提供许可证。它给你复制、发布和/或修改这些软件的法律许可。

同样，为了保护每个作者和我们自己，我们需要清楚地让每个人明白，自由软件没有担保（no warranty）。如果由于其他某个人修改了软件，并继续加以传播。我们需要它的接受者明白：他们所得到的并不是原来的自由软件。由其他人引入的任何问题，不应损害原作者的声誉。

最后，所有自由软件都不断受到软件专利的威胁。我们希望避免这样的风险，自由软件的再发布者以个人名义获得专利许可证，也就是将软件变为私有。为防止这一点，我们必须明确：任何专利必须以允许每个人自由使用为前提，否则就不准许有专利。

下面是有关复制、发布和修改的确切的条款和条件。

GNU 通用公共许可证

有关复制、发布和修改的条款和条件

0. 凡著作权人在其软件或其他著作中声明，该软件或著作得在通用公共许可证条款下才能发布，本许可对其均适用。以下所称的“程序”，是指任何一种适用通用公共许可的程序和著作；“基于本程序的著作”，则指程序或任何基于著作权法所产生的衍生著作，换言之，是指包含本程序全部或部分的著作，不论是否完整或经过修改的程序，以及（或）翻译成其他语言的程序（以下“修改”一词包括但不限于翻译行为）。被许可的人则称为“您”。

   本许可不适用于复制、发布及修改以外的行为；这些行为不在本许可范围内。执行本程序的行为并不受限制，而本程序的输出只有在其内容构成基于本程序所生的著作（而非只是因为执行本程序所造成）时，是受到本许可约束的。至于程序输出的内容是否构成本程序的衍生著作，则取决于本程序的具体用途。

1. 您可以对所收受的本程序源码，无论以何种媒介，复制与发布其完整的复制品，然

而您必须符合以下条件：以显著及适当的方式在每份复制品上发布适当的著作权标示及无担保声明；维持所有有关本许可以及无担保声明的原貌；并将本许可的副本连同本程序一起交付其他任一位程序收受者。

您可对授让复制品的实际行为请求一定的费用，也可自由决定是否提供担保作为收费的代价。

2. 您可以修改程序的一个或多个复制品或者程序的任何部分，以此形成基于本程序所生成的著作，并依前所述第一条规定，复制与发布修改过的程序或著作，但必须满足以下条件：

   a）您必须在所修改的文件上附加显著的标示，说明您修改过该文件，以及修改日期。

   b）必须就您所发布或发行的著作，无论是包含程序的全部还是部分的著作，或者是自程序或其他任何部分所衍生的著作，整体授权所有第三人可根据本许可规定使用，且不得因此项授权行为收取任何费用。

   c）若经过修改的程序在执行时通常以互动方式读取命令时，您必须在最常用的方式下，于开始进入互动方式时，列出或展示以下宣告：适当的著作权标示以及无担保声明（或声明由您提供担保）、使用者可以根据这些条件再发布此程序，以及告知使用者如何浏览本许可的副本。（例外：若程序本身是以交互方式执行的，然而通常却不回列出该宣告时，那您基于本程序所生成的著作便无需列出该宣告。）

   这些要求对修改过的著作是整体适用的。如果著作中可识别的一部分并非衍生自本程序，并且可合理地认为是一个独立、个别的著作，则当您将其作为个别著作加以发布时，本许可及其条款将不适用于该部分。当然当您将上述部分作为基于程序所生的著作的一部分而发布时，整个著作的发布就必须符合本许可条款的规定，而不管这些部分的作者是谁。

   因此，本条规定的意图不在于主张或剥夺您对完全由您所完成的著作的权利；应该说，本条规定意在行使对于基于程序的衍生著作或集合著作的发布行为的控制权。此外，非基于本程序所生的其他著作与本程序（或基于本程序产生的著作）在同一存储或发布媒介上的单纯聚集行为，并不会使该著作因此受到本许可条件的约束。

3. 可根据前面第一、二条规定，复制和发布程序（或第二条所述基于程序产生的著作）的目标代码或可执行形式，但必须符合以下条件：

   a）附上完整的相对机器可读的源码，而这些源码必须依前面第一、二条规定在经常作为软件交互的媒介上发布；或

   b）附上至少三年有效的书面报价文件，提供任何第三人在支付不超过实际发布源

码所需成本的费用下，取得相同源码的完整的机器可读的复制品，并依照前面第一、二条规定在经常用以作为软件交互的媒介上发布该复制品；或

c）附上所收受的有关发布相同源码的报价信息。（本项选择仅在非营利发布、且只在您依照前面 b 项方式自该书面报价文件收受程序目标代码或可执行形式时，才能适用。）

著作的源码是指对著作进行修改时适用的形式。对于可执行的著作而言，完整的源码是指著作中包含所有模式的全部源码，加上相关接口定义文件，还有用以控制该著作编译及安装的描述。然而，特别的例外情况是，所发布的源码并不需包括任何通常会随着所执行操作系统的主要组成部分（编译程序、核心等）而发布的软件（无论以源码或二进制格式），除非该部分本身就附加在可执行程序中。

如果可执行代码或目标代码的发布方式，是以指定的地点存取，供人复制，则提供可自相同地点复制源码的使用机会，看作是对源码的发布，然而第三人并不因此而负有将目标代码连同源码一起复制的义务。

4. 除本许可所明示的方式外，不得对程序加以复制、修改、再授权或发布。任何企图以其他任何方式进行复制、修改、再授权或发布程序的行为均为无效行为，并将自动终止您基于本许可所享有的权利。但依照本许可规定，从您手中收受复制品或权利的人，只要遵守本许可规定，他们所获得的许可并不会因此终止。
5. 因为您并未在本许可上签名，所以无需接受本许可。但除此之外，别无其他方式可以修改或发布程序或其衍生作品的授权许可。如不接受本许可，则这些行为在法律上都是被禁止的。因此对程序（或任何基于本程序所生成的著作）的修改和发布行为，表示已经接受了本许可，以及接受了所有关于复制、发布及修改程序（或基于程序生成著作）的条款和条件。
6. 每当再次发布程序（或任何基于程序所生成的著作）时，收受者就自动获得原授权人所有的关于复制、发布或修改程序的权利。不得就本授权所赋予接收者行使的权利附加任何进一步的限制。对于第三人是否履行本许可，无须负责。
7. 如果法院判决、专利侵权主张或其他任何理由（不限于专利争议）的结果，使得加诸于您的条件（无论是由法院命令、协议或其他方式造成）与本许可规定有所冲突，他们并不免除您必须遵守本许可的规定。如果无法同时符合本许可条件所生成的义务及其他相关义务而进行发布，那么后果就是不得发布该程序。例如，如果专利授权不允许直接或间接通过您而取得复制品的人，以免付权利金的方式再发布该程序时，惟一能够同时满足该义务及本许可的方式就是彻底避免发布本程序。

   如果本条任何一部分在特殊情况下被认定无效或无法执行时，本条其余部分仍应适用，且本条全部内容在其他情况下仍然应该适用。

   本条的目的并不是诱使您侵害专利或其他财产权的权利主张，或就此类主张的有效

性加以争执；本条的惟一目的是保障公共授权惯例所执行的自由软件发布系统的完整性。许多人信赖该系统一贯使用的应用程序，而对经由此系统发布的大量软件有相当多的贡献；作者/贡献者有权决定他或她是否希望经由其他系统发布软件，而被授权人则无该种选择权。

本条的用意在于将本许可其他不确定部分彻底解释清楚。

8. 如果因为专利或版权保护的接口问题，而使得本程序的发布与/或使用局限于某些国家时，则将本程序置于本许可规范之下的原著作权人得增加明确的发布地区限制条款，将一些国家排除在外，而使发布的许可只限制在未排除的国家之内。在该情况下，将限制条款如同以书面方式订定于本许可内容中，而成为本许可的条款。
9. 自由软件协会可以随时发布通用公共许可的修正版和/或新版本。新版本在精神上将近似于目前的版本，但在细节上有所不同以满足新的问题或状况。

   每个版本都有单独的版本编号。如果程序指定授权版本编号，表示其适用于该版本或是“任何新版本”时，得选择遵循该版本或任何由自由软件协会日后所发布的更新版本的条款或条件。如果程序没有指出授权版本编号，便要选择任一自由软件协会所发布的版本。
10. 如果想要将部分程序纳入其他自由程序，而其发布的条件有所不同时，请写信取得作者的许可。如果是自由软件协会享有著作权的软件，请写信到自由软件协会；我们有时会以特殊方式予以处理。我们的决定取决于两项目标：确保自由软件的所有衍生著作均维护在自由状态，并广泛地促进软件的共享或再利用。

无担保声明

11. 由于本程序是无偿授权的，因此在法律许可范围内，本许可对程序不负担责任。非经书面声明，著作权人和/或其他提供程序的人，无论显式或隐式，均是依照“现状”提供程序而并无任何形式的担保责任，其包含并不限于，就适售性以及特定目的的使用性为默认性担保。有关程序品质与效能的全部风险都由自己承担。如果程序被证明有瑕疵，您应该承担所有服务、修复或改正的费用。
12. 非经法律要求或书面同意，任何著作权人或任何可能依前述方式修改和/或发布程序的人，对于您因为使用或不能使用程序所造成的一般性、特殊性、意外性或间接性损失，不负任何责任（包括但不限于数据损失、数据执行不精确、或应由您或第三人承担的损失，或程序无法与其他程序运作等），即便前述的著作权人或其他团体已被告知这种损失的可能性。

条款结束

## 如何将这些条款用于你的新程序

如果你开发了新程序，而且希望它得到公众最大限度的利用。最好的办法就是将它变为自由软件。使得每个人都能在遵守条款的基础上对它进行修改和重新发布。

为了做到这一点，给程序附上下列声明。最安全的方式是将它放在每个源程序的开头，以便最有效地传递拒绝担保的信息。每个文件至少应有“版权所有”行以及在什么地方能看到声明全文的说明。

```
<one line to give the program's name and a brief idea of what it does.
Copyright (C) year  name of author>
```

这一程序是自由软件，你可以遵照自由软件协会出版的 GNU 通用公共许可证条款来修改和重新发布这一程序。或者用许可证的第二版，或者（根据你的选择）用任何更新的版本。

发布这一程序的目的是希望它有用，但没有任何担保。甚至没有适合于特定目的的隐含的担保。更详细的情况请参阅 GNU 通用公共许可证。

你应该已经和程序一起收到一份 GNU 通用公共许可证的副本。如果还没有，写信给自由软件协会(The Free Software Foundation, Inc.)，地址：59 Temple Place，Suite 330, Boston, MA 02111-1307 USA。

还应加上如何和你取得联系的相关信息。如果程序以交互方式进行工作，当它开始进入交互方式时，使它输出类似下面的简短声明：

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show
w'.
This is free software, and you are welcome to redistribute it under certain
conditions; type 'show c' for details.
```

假设的命令‘show c’及‘show w’应显示通用公共许可证的相应条款。当然，你使用的命令名称可以不同于‘show c’及‘show w’。根据程序的具体情况，也可以用菜单或鼠标选项来显示这些条款。

如果需要，你应该取得你的上司（如果你是程序员）或你的学校签署放弃程序版权的声明。下面只是一个例子，你应该改变相应的名称：

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.
  signature of Ty Coon, 1 April 1989
  Ty Coon, President of Vice
```

这一许可证不允许你将程序并入专用程序。如果你的程序是一个子程序库，你可能会认为用库的方式和专用应用程序连接更有用。如果你想这样做，使用 GNU 库通用公共许可证代替本许可证。

# 附录 B GCC 环境变量

有大量的环境变量可供设置以影响 GCC 编译程序的方式。利用这些变量的控制也可使用合适的命令行选项。

一些环境变量设置在目录名列表中。这些名字和 PATH 环境变量使用的格式相同。特殊字符 PATH_SEPARATOR（安装编译程序的时候定义）用在目录名之间。在 UNIX 系统中，分隔符是冒号，而 Windows 系统中为分号。

### C_INCLUDE_PATH

编译 C 程序时使用该环境变量。该环境变量指定一个或多个目录名列表，查找头文件，就好像在命令行中指定-isystem 选项一样。会首先查找-isystem 指定的所有目录。

也见 CPATH、CPLUS_INCLUDE_PATH 和 OBJC_INCLUDE_PATH。

### COMPILER_PATH

该环境变量指定一个或多个目录名列表，如果没有指定 GCC_EXEC_PREFIX 定位子程序，编译程序会在此查找它的子程序。

也见 LIBRARY_PATH、GCC_EXEC_PREFIX 和-B 命令行选项。

### CPATH

编译 C、C++和 Objective-C 程序时使用该环境变量。该环境变量指定一个或多个目录名列表，查找头文件，就好像在命令行中指定-I 选项一样。会首先查找-I 指定的所有目录。

也见 C_INCLUDE_PATH、 CPLUS_INCLUDE_PATH 和 OBJC_INCLUDE_PATH。

### CPLUS_INCLUDE_PATH

编译 C++程序时使用该环境变量。该环境变量指定一个或多个目录名列表，查找头文件，就好像在命令行中指定-isystem 选项一样。会首先查找-isystem 指定的所有目录。

也见 CPATH、C_INCLUDE_PATH 和 OBJC_INCLUDE_PATH。

### DEPENDENCIES_OUTPUT

为文件名设置该环境变量会让预处理程序将基于依赖关系的 makefile 规则写入文件。不会包括系统头文件名字。

如果环境变量设置为单名，被看作是文件名字，而依赖关系规则的名字来自源文件名字。如果定义中有两个名字，则第二个名字是用作依赖关系规则的目标名。

设置该环境变量的结果和使用命令行选项-MM、-MF 和-MT 的组合是一样的。也见 SUNPRO_DEPENDENCIES。

### GCC_EXEC_PREFIX

如果定义了该环境变量，它会作为编译程序执行的所有子程序名字的前缀。例如，如果将变量设置为 testver 而不是查找 as，汇编器首先会在名字 testveras 下查找。如果在此没有找到，编译程序会继续根据它的普通名进行查找。可在前缀名中使用斜线指出路径名。

GCC_EXEC_PREFIX 的默认设置为 *prefix*/lib/gcc-lib/，这里的 *prefix* 是安装编译程序时 configure 脚本指定的名字。该前缀也用于定位标准连接程序文件，包含进来作为可执行程序的一部分。

如果使用-B 命令行选项，会重写该设置。也见 COMPILER_PATH。

### LANG

该环境变量用于指出编译程序使用的字符集，可创建宽字符文字、串文字和注释。

定义 LANG 为 C-JIS，指出预处理程序将多字节字符按照 JIS（日语工业标准）字符进行解释。C-SJIS 可用来指出 SHIFT-JIS 字符而 C-EUCJP 指出日文 EUC。

如果没有定义 LANG，或定义为不可识别，函数 mblen()被用来确定字符宽度，而 mbtowc() 用来将多字节序列转换为宽字符。

### LC_ALL

如果设置，该环境变量的值重写 LC_MESSAGES 和 LC_CTYPE 的所有设置。

### LC_CTYPE

该环境变量指出引用串中定义的多字节字符的字符分类。主要用于确定字符串的字符边界，字符编码需要用引号或转义符，可被错误地解释为字符串的结尾或特殊字符串。对 Australian English，可将它设置为 en_AU；对 Mexican Spanish，可将它设置为 es_MX。如果没有设置该变量，默认为 LANG 变量的值，或如果没有设置 LANG，那就使用 C 英语行为。也见 LC_ALL。

### LC_MESSAGES

该环境变量指出编译程序使用何种语言发出诊断消息。对 Australian English，可设置为 en_AU；对 Mexican Spanish，可设置为 es_MX。如果变量没有设置，使用 LANG 变量的默认值，或如果没有设置 LANG，那就使用 C 英语行为。也见 LC_ALL。

### LD_LIBRARY_PATH

该环境变量不会影响编译程序，但程序运行的时候会有影响。变量指定一个目录列表，程序会查找该列表定位共享库。只有当未在编译程序的目录中找到共享库的时候，执行程序必须设置该变量。

### LD_RUN_PATH

该环境变量不会影响编译程序，但程序运行的时候会有影响。该变量在运行时指出文件的名字，运行的程序可由此得到它的符号名字和地址。地址不会重新载入，因而可能符号引用其他文件中的绝对地址。这和 ld 工具使用-R 选项完全一样。

### LIBRARY_PATH

该环境变量可设置为一个或多个目录名字列表，连接程序会搜寻该目录，以查找特殊连接程序文件，和由-l（字母 *l*）命令行选项指定名字的库。

由-L 命令行选项指定的目录在环境变量的前面，首先被查找。也见 COMPILER_PATH。

### OBJC_INCLUDE_PATH

在编译 Objective-C 程序的时候使用该环境变量。一个或多个目录名的列表由环境变量指定，用来查找头文件，就好像在命令行中指定-isystem 选项一样。所有由-isystem 选项指定的目录会首先被查找。

也见 CPATH、CPLUS_INCLUDE_PATH 和 C_INCLUDE_PATH。

### SUNPRO_OUTPUT

为文件名设置该环境变量会令预处理程序将基于依赖关系的 makefile 规则写入文件。会包含系统头文件名。

如果环境变量被设置为单个名字，它将会被当作文件名，依赖关系规则中的名字将由源文件的名字中获得。如果定义中有两个名字，第二个名字就是依赖关系规则中的目标名。

设置该环境变量的结果与在命令行中使用参数-M、-MF 和-MT 的效果一样。参见 DEPENDENCIES_OUTPUT。

### TMPDIR

这个变量包含了供编译程序存放临时工作文件的目录的路径名。这些文件通常在编译过程结束时被删除。这种文件的一个例子就是由预处理程序输出并输入给编译程序的文件。

# 附录 C

# 命令行对照表

本附录是命令行选项的对照表。命令行选项按照关键字和类别顺序列出。关键字就是从选项名中析取出来的字，类别则由选项的功能导出。你可以按照选项所作用的语言名称查找，也可以按照这些选项是否作用于编译程序内部操作等进行查找。例如，如果你想要知道哪个命令行选项会影响预处理程序或连接程序，可以在这里查找。

## 对照表

Ada -gnat, -I, --include-directory, --no-standard-includes, -nostdinc

alias -fargument-alias, -fargument-noalias, -fargument-noalias-global, -fstrict-aliasing

align -falign-functions, -falign-jumps, -falign-labels, -falign-loops, -Wcast-align

argument -fargument-alias, -fargument-noalias, -fargument-noalias-global, -fugly-args, -Wformat-extra-args

asm --assemble, -fasm, -fdata-sections, -ffunction-sections, -finhibit-size-directive, --for-assembler, -fverbose-asm, -Wa

assert -A, -A-, --assert

atexit -fuse-cxa-atexit

bitfields -fsigned-bitfields, -funsigned-bitfields

boehm -fuse-boehm-gc

bounds -fbounds-check, -ffortran-bounds-check

C --ansi, -ansi, -aux-info, -c, -C, -fallow-single-precision, -fasm, -fbuiltin, -fcommon, -fcond-mismatch, -fdollars-in-identifiers, -fdump-translation-unit, -ffreestanding, -fhosted, -finline, -fshort-wchar, -fsigned-bitfields, -fsigned-char, -funsigned-bitfields, -funsigned-char, -fwritable-strings, -pedantic, -pedantic-errors, -std, -traditional-cpp, -Waggregate-return, -Wbad-function-cast, -Wcast-align, -Wcast-qual, -Wchar-subscripts, -Wcomment, -Wconversion,

-Wdeprecated-declarations, -Werror-implicit-function-declaration, -Wformat, -Wformat-extra-args, -Wformat-nonliteral, -Wformat-security, -Wformat-y2k, -Wimplicit, -Wimplicit-function-declaration, -Wimplicit-int, -Wimport, -Winline, -Wlarger-than-size, -Wlong-long, -Wmain, -Wmissing-braces, -Wmissing-declarations, -Wmissing-format-attribute, -Wmissing-noreturn, -Wmissing-prototypes, -Wmultichar, -Wnested-externs, -Wpacked, -Wpadded, -Wparentheses, -Wpointer-arith, -Wredundant-decls, -Wreturn-type, -Wsequence-points, -Wshadow, -Wsign-compare, -Wstrict-prototypes, -Wswitch, -Wsystem-headers, -Wtraditional, -Wtrigraphs, -Wundef, -Wuninitialized, -Wwrite-strings

C++ --ansi, -ansi, -faccess-control, -falt-external-templates, -fasm, -fcheck-new, -fconserve-space, -fconst-strings, -fdefault-inline, -fdollars-in-identifiers, -fdump-class-hierarchy, -fdump-translation-unit, -fdump-tree-switch, -felide-constructors, -fenforce-eh-specs, -fexternal-templates, -ffor-scope, -fgnu-keywords, -fimplement-inlines, -fimplicit-inline-templates, -fimplicit-templates, -finline, -fmemoize-lookups, -fms-extensions, -fnonansi-builtins, -foperator-names, -foptional-diags, -fpermissive, -frepo, -frtti, -fshort-wchar, -fstats, -ftemplate-depth-number, -fuse-cxa-atexit, -fvtable-gc, -fweak, -fwritable-strings, -nostdinc++, -pedantic, -pedantic-errors, -Waggregate-return, -Wcast-align, -Wcast-qual, -Wchar-subscripts, -Wcomment, -Wconversion, -Wctor-dtor-privacy, -Wdeprecated, -Wdeprecated-declarations, -Weffc++, -Wextern-inline, -Wformat, -Wformat-extra-args, -Wformat-nonliteral, -Wformat-security, -Wformat-y2k, -Wimport, -Winline, -Wlarger-than-size, -Wlong-long, -Wmain, -Wmissing-braces, -Wmissing-format-attribute, -Wmissing-noreturn, -Wmultichar, -Wnon-template-friend, -Wnon-virtual-dtor, -Wold-style-cast, -Woverloaded-virtual, -Wpacked, -Wpadded, -Wparentheses, -Wpmf-conversions, -Wpointer-arith, -Wredundant-decls, -Wreorder, -Wreturn-type, -Wshadow, -Wsign-compare, -Wsign-promo, -Wswitch, -Wsynth, -Wsystem-headers, -Wundef, -Wuninitialized, -Wwrite-strings

call -fcall-saved-register, -fcall-used-register, -fcaller-saves, -fnon-call-exceptions, -foptimize-sibling-calls

case -fcase-initcap, -fcase-lower, -fcase-preserve, -fcase-strict-lower, -fcase-strict-upper, -fcase-upper, -fignore-case, -fintrin-case-spec, -fmatch-case-spec, -fsource-case-spec, -fsymbol-case-spec

cast -Wbad-function-cast, -Wcast-align, -Wcast-qual, -Wold-style-cast

char -fsigned-char, -funsigned-char, -Wchar-subscripts

check -fbounds-check, -fcheck-new, -fcheck-references, -fdelete-null-pointer-checks, -fforce-classes-archive-check, -ffortran-bounds-check, -fruntime-checking, -fstack-check, -fstore-check

Chill -fchill-grant-only, -fgrant-only, -fignore-case, -flocal-loop-counter, -fold-string, -fruntime-checking, -I, --include-directory, -lang-chill

class --bootclasspath, -fconstant-string-class, -fdump-class-hierarchy, -fforce-classes-archive-check, -foptimize-static-class-initialization, -foutput-class-dir, --main, --output-class-directory

comment -C, -Wcomment

compile -c, --compile, -E, -fassume-compiled, -fcommon, -fcompile-resource, -fgnu-linker, -fgrant-only, -fhosted, -fmerge-all-constants, -fmerge-constants, -fsyntax,

-ftest-coverage, -ftime-report, --help, -pass-exit-codes, -pipe, --preprocess, -Q, -s, -S, -save-temps, -syntax-only, -time, -v

complex -femulate-complex, -fugly-complex

constant -fconst-strings, -fconstant-string-class, -fkeep-static-consts, -fmerge-all-constants, -fmerge-constants, -fsingle-precision-constant

conversion -Wconversion, -Wpmf-conversions

cse -fcse-follow-jumps, -fcse-skip-blocks, -ffunction-cse, -frerun-cse-after-loop

debug -a, -d, --debug, -fdump-class-hierarchy, -fdump-translation-unit, -fdump-tree-switch, -fdump-unnumbered, -finstrument-functions, -fmem-report, -foptional-diags, -fpermissive, -fsilent, -fstats, -ftemplate-depth-number, -ftrapping-math, -ftrapv, -fverbose-asm, -g, -gcoff, -gdwarf, -gdwarf-2, -ggdb, -gstabs, -gvms, -gxcoff, -H, -v

declaration -gen-decls, -Wdeprecated-declarations, -Werror-implicit-function-declaration, -Wimplicit-function-declaration, -Wmissing-declarations, -Wredundant-decls

dependencies --dependencies, -M, -MD, -MF, -MG, -MM, -MMD, -MP, -MQ, -MT, -pass-exit-codes, --print-missing-file-dependencies, --user-dependencies, -Wout-of-date, --write-dependencies, --write-user-dependencies

deprecated -Wdeprecated, -Wdeprecated-declarations

directory -B, --bootclasspath directory, -foutput-class-dir, -I, -I-, -idirafter, -include, --include-barrier, --include-directory, --include-directory-after, --include-prefix, --include-with-prefix, --include-with-prefix-after, -iprefix, -isystem, -iwithprefix, -iwithprefixbefore, -L, --library-directory, --output-class-directory, --prefix, -print-multi-directory, -print-prog-name, -print-search-dirs

dollar -fdollar-ok, -fdollars-in-identifiers

dump -d, --dump, -dumpbase, -dumpmachine, -dumpspecs, -dumpversion, -fdump-class-hierarchy, -fdump-translation-unit, -fdump-tree-switch, -fdump-unnumbered

error -fmessage-length, -pedantic-errors, -Werror, -Werror-implicit-function-declaration

**exception** -fasynchronous-unwind-tables, -fcheck-new, -fenforce-eh-specs, -fexceptions, -fnon-call-exceptions, -fnonansi-builtins

**extern** -falt-external-templates, -fexternal-templates, -Wextern-inline, -Wnested-externs

**file** -aux-info, -B, --bootclasspath, -include, --include-barrier, --include-directory-after, --include-prefix, --include-with-prefix, --include-with-prefix-after, --language, -llibrary, -MF, --output, -print-file-name, -print-libgcc-file-name, --print-missing-file-dependencies, -print-prog-name, -remap, -save-temps, -x

**float** -ffloat-store, -fpretend-float, -Wfloat-equal

**form** -ffixed-form, -ffree-form

**format** -Wformat, -Wformat-extra-args, -Wformat-nonliteral, -Wformat-security, -Wformat-y2k, -Wmissing-format-attribute

**Fortran** -fautomatic, -fbackslash, -fbadu77-intrinsics-spec, -fbounds-check, -fcase-initcap, -fcase-lower, -fcase-preserve, -fcase-strict-lower, -fcase-strict-upper, -fcase-upper, -fdollar-ok, -femulate-complex, -ff2c, -ff2c-intrinsics-spec, -ff66, -ff77, -ff90, -ff90-intrinsics-spec, -ffixed-form, -ffixed-line-length-len, -ffortran-bounds-check, -ffree-form, -fglobals, -fgnu-intrinsics-spec, -finit-local-zero, -finline, -fintrin-case-spec, -fmatch-case-spec, -fmil-intrinsics-spec, -fonetrip, -fpedantic, -fsecond-underscore, -fsilent, -fsource-case-spec, -fsymbol-case-spec, -fsyntax, -ftypeless-boz, -fugly-args, -fugly-assign, -fugly-assumed, -fugly-comma, -fugly-complex, -fugly-init, -fugly-logint, -funderscoring, -funix-intrinsics, -fversion, -fvxt, -fvxt-intrinsics, -fzeros, -maligned-data, -pedantic, -pedantic-errors, -Wglobals, -Wimplicit, -Wsurprising, -Wuninitialized

**function** -falign-functions, -ffunction-cse, -ffunction-sections, -finline-functions, -finstrument-functions, -fkeep-inline-functions, -Wbad-function-cast, -Werror-implicit-function-declaration, -Wimplicit-function-declaration, -Wunused-function

**garbage** -fuse-boehm-gc, -fvtable-gc

**gcse** -fgcse, -fgcse-lm, -fgcse-sm

**global** -fargument-noalias-global, -fglobals, -fvolatile-global, -Wglobals

**gnu** -fgnu-intrinsics-spec, -fgnu-keywords, -fgnu-linker, -fgnu-runtime

**implicit** -fimplicit-inline-templates, -fimplicit-templates, -Werror-implicit-function-declaration, -Wimplicit, -Wimplicit-function-declaration, -Wimplicit-int

**include** -include, --include-barrier, --include-directory, --include-directory-after, --include-prefix, --include-with-prefix, --include-with-prefix-after, --include-with-prefix-before, --no-standard-includes, --trace-includes

inline -fasm, -fdefault-inline, -fimplement-inlines, -fimplicit-inline-templates, -finline, -finline-functions, -finline-limit, -fkeep-inline-functions, -Wextern-inline, -Winline

intrinsics -fbadu77-intrinsics-spec, -ff2c-intrinsics-spec, -ff90-intrinsics-spec, -fgnu-intrinsics-spec, -fmil-intrinsics-spec, -funix-intrinsics, -fvxt-intrinsics iprefix

Java --bootclasspath, -C, -D, --define-macro,--encoding, -fassume-compiled, -fbounds-check, -fcheck-references, -fcompile-resource, -fencoding, -fforce-classes-archive-check, -fhash-synchronization, -fjni, -foptimize-static-class-initialization, -foutput-class-dir, -fstore-check, -fuse-boehm-gc, -fuse-divide-subroutine, -I, --include-directory, --main, --output-class-directory, -Wextraneous-semicolon, -Wlarger-than-size, -Wout-of-date, -Wredundant-modifiers, -Wshadow

label -falign-labels, -Wunused-label

length -ffixed-line-length-len, -fmessage-length

lib -B, -L, -l, --library-directory, --no-standard-libraries, -print-libgcc-file-name, -print-multi-lib, -shared, -shared-libgcc, -static, -static-libgcc, -symbolic

link -c, --compile, -fcommon, -fgnu-linker, -fhosted, -fmerge-all-constants, -fmerge-constants, --for-linker, --force-link, -fvtable-gc, -L, --library-directory, -llibrary, -no-standard-libraries, -nodefaultlibs, -nostartfiles, -nostdlib, -s, -shared, -shared-libgcc, -static, -static-libgcc, -symbolic, -u, -Wl, -Xlinker

machine -b, --target, --target-help

macro --ansi, -D, --define-macro, -ffast-math, -ffixed-register, -imacros, -U, -undef, --undefine-macro

math -fallow-single-precision, -femulate-complex, -ffast-math, -ffloat-store, -fmath-errno, -fpretend-float, -fschedule-insns, -fschedule-insns2, -fshort-double, -fsingle-precision-constant, -ftrapping-math, -ftrapv, -ftypeless-boz, -fugly-complex, -funsafe-math-optimizations, -funsigned-bitfields, -funsigned-char, -fuse-divide-subroutine, -Wdiv-by-zero, -Wfloat-equal, -Wsign-compare, -Wsign-promo, -Wsurprising

missing --print-missing-file-dependencies, -Wmissing-braces, -Wmissing-declarations, -Wmissing-format-attribute, -Wmissing-noreturn, -Wmissing-prototypes

Objective-C --ansi,-ansi, -fasm, -fbuiltin, -fconstant-string-class, -fgnu-runtime, -finline, -gen-decls, -Waggregate-return, -Wcast-align, -Wcast-qual, -Wchar-subscripts, -Wcomment, -Wconversion, -Wdeprecated-declarations, -Wformat, -Wformat-extra-args, -Wformat-nonliteral, -Wformat-security, -Wformat-y2k, -Wimport, -Winline, -Wlarger-than-size, -Wlong-long, -Wmissing-braces, -Wmissing-format-attribute, -Wmissing-noreturn, -Wmultichar, -Wpacked, -Wpadded, -Wparentheses, -Wpointer-arith, -Wprotocol, -Wredundant-decls, -Wselector,

-Wshadow, -Wsign-compare, -Wswitch, -Wsystem-headers, -Wundef, -Wuninitialized

optimization -fasynchronous-unwind-tables, -fbranch-probabilities, -fcall-saved-register, -fcall-used-register, -fcaller-saves, -fcommon, -fconserve-space, -fcprop-registers, -fcse-follow-jumps, -fcse-skip-blocks, -fdata-sections, -fdefer-pop, -fdelayed-branch, -fdelete-null-pointer-checks, -fdiagnostics-show-location, -felide-constructors, -fexpensive-optimizations, -ffloat-store, -ffunction-cse, -ffunction-sections, -fgcse, -fgcse-lm, -fglobals, -fguess-branch-probability, -finit-local-zero, -fkeep-static-consts, -fmemoize-lookups, -fmove-all-movables, -fomit-frame-pointer, -foptimize-register-move, -foptimize-sibling-calls, -foptimize-static-class-initialization, -fpack-struct, -fpeephole, -fpeephole2, -fppc-struct-return, -fprefetch-loop-arrays, -freduce-all-givs, -freg-struct-return, -fregmove, -frename-registers, -frerun-cse-after-loop, -frerun-loop-opt, -fruntime-checking, -fschedule-insns, -fschedule-insns2, -fshort-double, -fshort-enums, -fssa, -fssa-ccp, -fssa-dce, -fstack-check, -fstore-check, -fstrength-reduce, -fstrict-aliasing, -fthread-jumps, -funroll-all-loops, -funroll-loops, -funwind-tables, -fvtable-gc, -fzeros, -O, --optimize optimize, --optimize, --param, -Wdisabled-optimization

preprocessor -A, -A-, --assert, -C, -D, --define-macro, --dependencies directory, -E, -fident, -fpreprocessed, -H, -I, -I-, -idirafter, -imacros, -include, --include-barrier, --include-directory, --include-directory-after, --include-prefix, --include-with-prefix, --include-with-prefix-after, --include-with-prefix-before, -iprefix, -isystem, -iwithprefix, -iwithprefixbefore, -M,-MD, -MF, -MG, -MM, -MMD, -MP, -MQ, -MT, --no-line-commands, --no-standard-includes, -nostdinc, -nostdinc++, -P, --preprocess, --print-missing-file-dependencies, -remap, --trace-includes, -trigraphs, -U, -undef, --undefine-macro, --user-dependencies, -Wp, --write-dependencies, --write-user-dependencies, -Wsystem-headers, -Wundef, -Wunknown-pragmas

profile -a, -fdata-sections, -fprofile-arcs, -ftest-coverage, -p, -pg, --profile, --profile-blocks

prototypes -Wmissing-prototypes, -Wstrict-prototypes

register -fcall-saved-register, -fcall-used-register, -fcprop-registers, -ffixed-register, -fforce-addr, -fforce-mem, -foptimize-register-move, -freg-struct-return, -fregmove, -frename-registers, -fstack-limit-register, -remap

return -fppc-struct-return, -freg-struct-return, -Waggregate-return, -Wreturn-type

sign -fsigned-bitfields, -fsigned-char, -Wsign-compare, -Wsign-promo

ssa -fssa, -fssa-ccp, -fssa-dce

stack -fstack-check, -fstack-limit-register, -fstack-limit-symbol

standard --ansi, -ansi, -ff2c, -ff2c-intrinsics, -ff66, -ff77,-ff90, -ff90-intrinsics,

-ffixed-form, -ffixed-line-length-len, -ffor-scope, -ffree-form, -fgnu-keywords, -fmil-intrinsics, -fms-extensions, -fnext-runtime, -fnonansi-builtins, -foperator-names, -fpedantic, -fpermissive, -fsigned-bitfields, -fsigned-char, -ftrapping-math, -ftrapv, -fugly-args, -fugly-assign, -fugly-assumed, -fugly-comma, -fugly-complex, -fugly-init, -fugly-logint, -fvxt, -fvxt-intrinsics, -fwritable-strings, --no-standard-includes, --no-standard-libraries, -pedantic,-std, -traditional-cpp, -Wtraditional

static -fkeep-static-consts, -foptimize-static-class-initialization,-fvolatile-static, -static, -static-libgcc

strings -fconst-strings, -fconstant-string-class,-fold-string, -fwritable-strings, -Wwrite-strings

syntax -fsyntax, -syntax-only

template -falt-external-templates, -fexternal-templates, -fimplicit-inline-templates, -fimplicit-templates, -ftemplate-depth-number, -Wnon-template-friend

underscore -fleading-underscore, -fsecond-underscore, -funderscoring

version -dumpversion, -fversion, --use-version, -v, -V

warn --all-warnings, --extra-warnings, -fmessage-length,--no-warnings, -w, -W, -Waggregate-return, -Wall, --warn-, -Wbad-function-cast,-Wcast-align,-Wcast-qua 1, -Wchar-subscripts, -Wcomment, -Wconversion, -Wctor-dtor-privacy, -Wdeprecated, -Wdeprecated-declarations, -Wdisabled-optimization, -Wdiv-by-zero, -Weffc++, -Werror, -Werror-implicit-function-declaration, -Wextern-inline, -Wextraneous-semicolon, -Wfloat-equal, -Wformat, -Wformat-extra-args, -Wformat-nonliteral, -Wformat-security, -Wformat-y2 k, -Wglobals, -Wimplicit, -Wimplicit-function-declaration, -Wimplicit-int, -Wimport, -Winline, -Wlarger-than-size, -Wlong-long, -Wmain, -Wmissing-braces, -Wmissing-declarations, -Wmissing-format-attribute, -Wmissing-noreturn, -Wmissing-prototypes, -Wmultichar, -Wnested-externs, -Wnon-template-friend, -Wnon-virtual-dtor, -Wold-style-cast, -Wout-of-date, -Woverloaded-virtual, -Wpacked, -Wpadded, -Wparentheses, -Wpmf-conversions, -Wpointer-arith, -Wprotocol, -Wredundant-decls, -Wredundant-modifiers, -Wreorder, -Wreturn-type, -Wselector, -Wsequence-points, -Wshadow, -Wsign-compare, -Wsign-promo, -Wstrict-prototypes, -Wsurprising, -Wswitch, -Wsynth, -Wsystem-headers, -Wtraditional, -Wtrigraphs, -Wundef, -Wuninitialized, -Wunknown-pragmas, -Wunreachable-code, -Wunused, -Wunused-function, -Wunused-label, -Wunused-parameter, -Wunused-value, -Wunused-variable, -Wwrite-strings

# 附录 D GCC 命令行选项

本附录按照字母顺序列出所有的命令行选项。这些选项中的某些可以作用于每种语言，而另一些则只能作用于一种或一些语言，它们只是多种语言中的一小部分。也有一些作用于预处理程序、汇编器或是链接器的选项。对每个选项，都在相应选项的右边标出了它所作用的一种或多种语言。如果某个选项作用于所有语言，就没有这样的标记了。

事实上用 gcc 命令可以编译任何一种语言，但可能就无法使用某种限制语言的选项了。每种语言都有自己的前端驱动处理，因此如果某选项只针对一种语言，为了识别这些选项，就需要使用相应的驱动作为 gcc 的前端。

## D.1 选项前缀

所有选项都以连字符开始。它们中的一些还会用两个连字符。某些以-f 和-W 开始的选项还有一些特殊的含义。

**前缀--**

在命令行中，标识选项的传统方法是在字母前加一个连字符。新的方式则是以两个连字符开始。本索引中列出的许多选项既有旧的形式，也有新的形式，但意义完全相同。例如，设定在生成的代码中携带调试信息的传统选项是：

```
-g
```

同一选项还可以用如下较长的形式，如：

```
--debug
```

**前缀-f**

字母 f 代表 flag（标记）。大多数选项只有两种状态，开和关。例如，下面的选项设定是否打开窥孔（peephole）优化：

```
-fpeephole
```

由于这一标记要么是开，要么是关，因而每个标记选项都有一个相反的选项，它使用相同的名字，但以 no 作为前缀。例如：

```
-fno-peephole
```

几乎所有的选项都会触发一个真或假的标记设置，因此选项中的一个值会是默认设置。但也有例外。比如，如下两个选项中的任意一个都能指出 for 循环中变量的作用域：

```
-ffor-scope
-fno-for-scope
```

两个作用域设置中的任何一个都不是默认的。默认方式会符合标准，但这两个设置都是标准的变体。

任何-f 选项都可以具有一个双连字符的选项形式。例如，如下两个选项功能相同：

```
-frtti
--rtti
```

**前缀-W**

前缀-W 通常用来指定是否需要编译程序产生某些警告消息。与-f 标记的设定方式相似，警告也可以打开或是在名字前加 no 来关闭。例如，如下的设定可以令编译程序在对函数调用使用了太多的参数时产生警告：

```
-Wformat-extra-args
```

如果需要抑制警告消息，可以使用如下选项：

```
-Wno-format-extra-args
```

### D.1.1 命令行的顺序

选项的顺序很重要。如果命令行中有两个选项，并且它们是冲突的，通常第二个选项会改变第一个选项所产生的设定，从而覆盖了第一个选项。命令行一般由左向右进行分析，每个选项设定一个值或是一个标记（或是一组值或标记），因此在命令行中设置好的内容可能在其后被修改。

对顺序的要求其实很有实用价值。例如，-O3 优化选项将打开-finline-functions 选项。但如果想使用-O3 选项，又要关闭函数内嵌功能，可以按下面的顺序输入选项：

```
-O3 -fno-inline-functions
```

### D.1.2 文件类型

编译程序通过匹配表 D-1 中的后缀检查文件名，从而判断文件的内容。任何具有未知

后缀的文件都视作目标机上链接器的输入，并在链接阶段传递给链接器。选项-x 可以指示编译程序忽略后缀并将文件作为某一类型对待。

表 D-1 GCC 可识别的文件名后缀

| 后缀 | 文件内容 |
|---|---|
| .a | 包含一个或多个供链接器使用的.o 文件的静态库（也称为归档文件） |
| .c | 需要进行预处理的 C 源代码 |
| .adb | Ada 体文件，就是包含库单元体的源文件 |
| .ads | Ada 规范文件，就是包含库单元声明或库单元重命名声明的源文件 |
| .C .c++ .cc .cp .cpp .cxx | 需要进行预处理的 C++源代码 |
| .class | 一个包含通过编译 Java 程序产生的字节码的文件 |
| .f .for .FOR | 不需要预处理的 Fortran 源代码 |
| .F .fpp .FPP | 需要预处理的 Fortran 源代码 |
| .h | C，C++或 Objective-C 的头文件 |
| .i | 不需要预处理的 C 源代码 |
| .ii | 不需要预处理的 C++源代码 |
| .java | Java 源代码 |
| .m | 需要预处理的 Objective C 源代码 |
| .mi | 不需要预处理的 Objective C 源代码 |
| .mo | 包含国际化支持翻译的二进制文件 |
| .o | 可供链接器使用的目标文件 |
| .po | 包含国际化支持翻译的文本文件 |
| .r | 需要由 RATFOR 预处理程序预处理的 Fortran 源代码 |
| .S | 需要预处理的汇编语言源代码 |
| .s | 不需要预处理的汇编语言源代码 |
| .so | 包含一个或多个供链接器使用的.o 文件的动态库（也称共享库） |
| <other> | 具有不可识别后缀或无后缀的文件，这些文件均视为链接器的输入并且不做任何修改 |

## D.2 选项字母顺序列表

**-###**

显示编译程序当前版本号，并显示编译和链接中每个阶段可用的所有命令，但并不执行任何命令。当被单独使用时，该选项显示编译程序的当前版本号。当和选项--help 一起使用时，将会列出完整的命令行选项。

参见-v。

-a

在可执行代码每个基本逻辑块的顶端，都会产生额外代码用来进行记录。运行一次基本逻辑块，就会记录一次。所记录的信息包括基本逻辑块的起始地址和逻辑块中的函数名称。如果同时给出选项-g，则每个基本逻辑块所记录的信息将包括文件名和每个逻辑块的起始行。这些信息会写到文件 bb.out 中（除非在机器描述中给出了其他的名字）。

参见-ax、-fprofile-arcs 和-ftest-coverage。该选项也可写作--profile-blocks。

-A question（answer）　Pre

按如下形式为某个断言指定问题和答案：

```
#if #question(answer)
```

该选项也可以写作--assert。参见-A-。

-A-　Pre

关闭通常用于描述目标机的标准断言。参见-A。

--all-warnings

参见-Wall。

--ansi　C C++ ObjC

与-ansi 相同。

-ansi　C C++ ObjC

该选项指示编译程序编译符合标准的程序，但并不限制与标准不冲突的其他方式。也就是说，不冲突的 GNU 扩展仍然有效。对于 C 而言，该选项能够正确地编译符合 ISO C89 的程序。对于 C++而言，与 ISO C++冲突的 GNU 扩展将被禁止。

该选项同时打开选项-fno_asm，-fno_nonansi_builtin，-trigraphs 和-fno_dollars_in_identifiers。对于 C++，它还打开了选项-fno_gnu_keywords 和-fno-nonasni-builtins。

该选项定义了宏__STRICT_ANSI__，它可以防止包含某些头文件，从而不会引起被编译程序中的名字和头文件中声明的函数或宏之间产生冲突。

该选项关闭了 GNU C 扩展关键字 asm，typeof 和 inline，但它们的替代形式__asm__，__typeof__，__inline__仍然有效。

如果要限制代码严格符合标准，可以和-ansi 一起使用-pedantic。参见-std。

该选项也可以写作--ansi。

--assemble

与-S 相同。

--assert question（answer）

Pre

与-A 相同。

-aux-info filename

C

输出所有已声明和在一个独立编译单元（一个 C 源文件和其包含的所有头文件）中定义的函数声明。

-b machine

该选项指示需要编译程序的目标机。如果没有指定该选项，默认是为编译程序所运行的目标机编译代码。目标机通过指定包含编译程序的目录来确定，通常为 /user/local/lib/gcc-lib/*machine/version*。

参见-B 和-V。该选项也可以写作--target。

-Bprefix

路径 prefix 给出库文件的位置，包括编译程序的文件、执行程序和数据文件。如果需要运行子程序，如 cpp，as 或 ld，就会用该前缀来定位这些程序。可以根据自己的意愿在前缀的末尾给出目录分隔符或不给。

所有的过程均使用同样的标准搜索过程。下面的步骤按顺序执行，直至要寻找的目标已被定位：

1. 如果用选项-B 给出前缀，就用它来构建路径名。
2. 使用前缀/usr/lib/gcc/构建路径。
3. 使用前缀/usr/local/lib/gcc-lib/构建路径。
4. 按照变量中的出现顺序，使用变量 PATH 中定义的每个目录路径。

选项-B 也可以用来为链接器定位库文件，这是因为编译程序将该选项翻译为选项-L 并传递给链接器。

选项-B 还可以用来定位头文件，这是因为所给出的前缀具有追加的目录名，它被翻译为选项-isystem 并传递给预处理程序。

环境变量 GCC_EXEC_PREFIX 可以被设置为前缀目录的名字，并具有和选项-B 相同的效果。

引导编译程序的特殊例子：如果前缀具有的形式从 dirpath/stage0 到 dirpath/stage9，它将被替换为 dirpath/include。

该选项也可以写作--prefix。

—bootclasspath=pathname **Java**

pathname 指定标准 Java 包和类（如 java.lang.String）的位置。路径名可以是目录、jar 文件或 zip 文件。

参见--classpath 和-I。

-c

不调用链接器。这一选项会将源文件编译或汇编为目标文件，但不会运行链接器来创建可执行体。产生的目标文件将存放在与源文件同名但以.o 为后缀的文件中。表 D-1 中所有不可识别的输入文件或是-x 所指示的类型都将被忽略。

这一选项也可以写作--compile。

-C **Java**

引起编译程序产生字节码类文件，而不是默认的可执行目标代码。参见-foutput-class-dir。

-C **Pre**

与选项-E 一同使用，该选项可以去除所有的注释。

该选项也可以写作--comments。

--compile

与-c 相同。

-dletters

这里可以给出一个或多个字母，它们用来指定什么时候进行调试输出和输出哪些内容。这一选项是用来调试编译程序的，可以在编译的不同阶段检查详细信息。每个输出文件都有一个后缀标识，该后缀标识是阶段号和某些标识字母。例如，在阶段 21 后输出的文件内容为全局寄存器分配，当编译文件 doline.c 时，该输出文件可能命名为 doline.21.greg。

参见选项-dumpbase，-fdump-unnumbered，-fdump-translation-unit，-fdump-class-hierarchy 和-fdump-tree-switch。该选项还可以写作--dump。

表 D-2 列出了-d 选项中可以使用的字母，它们可以按照任意组合和顺序插入。这一特征是完全为了调试编译程序所实现的，所以你会发现并不是在每个发布中都有这些字母。注意，当同选项-E 一同使用时，字母 D，I，M 和 N 对预处理程序具有特殊的含义。

表 D-2 与-d 选项联合使用的字母

| 字母 | 作用 |
|---|---|
| A | 用各种调试信息注释汇编语言输出 |
| a | 设置标记，输出除文件 name.pass.vcg 以外的所有命名文件，文件 name.pass.vcg 只能通过选项字母 v 设定 |
| b | 在计算分支概率后输出到文件 name.14.bp 中 |
| B | 在进行块重排序后输出到文件 name.29.bbro 中 |
| c | 在组合指令后输出到文件 name.16.combine 中 |
| C | 在首次 if 变换后输出到文件 name.17.ce 中 |
| d | 在延迟分支调度后输出到文件 name.31.dbr 中 |
| D | 与选项-E 一同使用，除了预处理后的正常输出外，还输出所有的宏定义 |
| e | 在静态赋值优化后输出到文件 name.04.ssa 和 name.07.ussa 中 |
| E | 在第二次 if 变换后输出到文件 name.26.ce2 中 |
| f | 在数据流分析后输出到文件 name.13.cfg 中，并在活性分析后输出到文件 name.15.life 中 |
| F | 在修剪 ADDRESSOF 代码后输出到文件 name.09.addressof 中 |
| g | 在全局寄存器分配后输出到文件 name.21.greg 中 |
| G | 在 GCSE 后输出到文件 name.10.gcse 中 |
| h | 在异常处理结束后输出到文件 name.02.eh 中 |
| i | 在同属调用优化后输出到文件 name.01.sibling 中 |
| I | 当同选项-E 一同使用时，预处理程序输出#include 指示字，还输出其他预处理程序的输出 |
| j | 在首次跳转优化后输出到文件 name.03.jump 中 |
| k | 寄存器到栈（register-to-stack）的转换后，输出到文件 name.28.stack 中；而在寄存器到栈的变换结束后，输出到文件 name.32.stack 中 |
| l | 在局部寄存器分配后输出到文件 name.20.lreg 中 |
| L | 在循环优化后输出到文件 name.11.loop 中 |
| M | 在机器依赖重构阶段后输出到文件 name.30.mach 中。当同选项-E 一同使用时，预处理程序将在所有的预处理后输出有效的宏定义列表 |
| m | 在编译结束时，向标准错误输出内存使用信息 |
| n | 在寄存器重编号后输出到文件 name.25.rnreg 中 |
| N | 在寄存器移动阶段后输出到文件 name.18.regmove 中。当同选项-E 一同使用时，除了在预处理后的正常输出外，还包括以#define name 简单形式定义的所有宏的列表 |
| o | 在后寄存器重载优化之后，输出到文件 name.22.postreload 中 |
| p | 用每条指令的长度以及标明使用了哪种模式进行优化的注释标注汇编语言 |

（续表）

| 字母 | 作用 |
|---|---|
| P | 使用可以产生每条指令的 RTL 代码标注汇编语言。选项-dp 也会打开该功能，而且会有更多的标注 |
| r | 在生成 RTL 后输出到文件 name.00.rtl 中。参见字母 x |
| R | 在第二次调度阶段后输出到文件 name.27.sched2 中 |
| s | 在 CSE 以及有时跟在首次 CSE 阶段后的跳转优化之后输出到文件 name.08.cse 中 |
| S | 在首次调度阶段后输出到文件 name.19.sched 中 |
| t | 在第二次 CSE 阶段以及有时跟在第二次 CSE 阶段后的跳转优化之后输出到文件 name.12.cse2 中 |
| u | 在 SSA 优化后输出到文件 name.06.null 中 |
| v | 将每一个其他文件（除文件 name.00.rtl 之外）控制流图表示输出到文件 name.pass.vcg 中。该文件可以由 vcg 读取和显示 |
| w | 在第二次流程阶段后输出到文件 name.23.flow2 中 |
| W | 在 SSA 条件代码传播后输出到文件 name.05.ssaccp 中 |
| X | 在 SSA 不可达代码删除阶段后输出到文件 name.06.ssadce 中 |
| x | 为函数生成 RTL 但不编译它。该字母通常与字母 r 一起使用 |
| y | 解析器将调试信息输出到标准错误中 |
| z | 在窥孔优化阶段后输出到文件 name.24.peephole2 中 |

### -Dproperty[=string]

**Java**

该选项可以在使用选项--main 的命令行中使用，同时它将定义一个可以通过调用 java.lang.System.getProperty()方法获取的属性。如果 string 没有指定，所定义的字串就是一个空字符串。

该选项也可以写作--define-macro。

### -Dmacro[=string]

**Pre**

如果指定了 string，宏名字就被定义了，就好像它就是程序的一部分。例如，-Dbrunt=logger 将产生如下的定义：

```
#define brunt=logger
```

如果没有指定 string，宏就定义为字符串“1”。例如，-Dminke 产生如下的定义：

```
#define minke 1
```

所有-D 选项均在-U 选项前处理，所有-U 选项均在-include 或-imacros 选项前处理。

该选项也可以写作--define-macro。

--debug[level]

与选项-g 相同。

--define-macro macro[=string] Pre Java

与选项-D 相同。

--dependencies Pre

与选项-M 相同。

--dump letters

与选项-d 相同。

-dumpbase base

base 是基本文件名，用于命名选项-d 所产生的输出文件。
该选项也可以写作--dumpbase。

-dumpmachine

打印该编译程序的目标机名字，同时不采取任何动作。

-dumpspecs

打印构建编译程序的规范信息，同时不采取任何动作。所打印的信息是一个很长的列表，包括用来编译、汇编和链接 GCC 编译程序自身用到的所有选项（和默认选项）。

-dumpversion

打印编译程序的版本号，同时不采取任何动作。

-E Pre

当源代码经过预处理后暂停并输出结果。除非利用选项-o 指定输出文件，否则所产生的输出将写到标准输出中。不需要预处理的输入文件将被忽略，这一过程可以通过表 D-1 中的文件名后缀判断，或用选项-x 来指示表中不包含的文件类型。

该选项定义了环境变量__GNUC__，__GNUC_MINOR__和__GNUC_PATCHLEVEL__。

选项-dD，-dI，-dM 和-dN 仅当和选项-E 一同使用时才具有特殊含义。参见选项-C 和-P。这一选项还可以写作--preprocess。

--encoding=name Java

与选项-fencoding 相同。

--extra-warnings

与选项-W 相同。

-faccess-control

这是默认选项。如果指定了-fno-access-control，编译程序将不会检查访问权限。这一选项的惟一用途是绕过编译程序中的一个访问权限错误。

-falign-functions[=number]

将函数的起始地址在大于或等于 number 的 2 的幂次方的边界上对齐，但仅当必须略过不多于 number 个字节时才使用。例如，如果 number 是 20，其最终对齐的边界将是 32 字节边界，同时不会略过多于 20 个字节。

将 number 设置为 2 的幂次方将引起所有的函数均边界对齐。如果没有指定 number，将使用机器的默认值。对于某些机器而言，这个数值是 2 的幂次方，因此会使所有的函数边界对齐。将 number 指定为 1 与使用选项-fno-align-functions 等价，不会造成边界对齐。

-falign-jumps[=number]

将分支目标在那些大于或等于 number 的 2 的幂次方的边界上对齐，但仅当无需略过多于 number 个字节时有效。例如，如果 number 是 20，则有效的对齐边界是在 32 个字节的位置上，但同时又不能忽略多于 20 个字节。和与之相似的选项-falign-labels 不同，该选项不需要在分支目标前插入空指令。

如果没有指定 number，将使用机器的默认值，一般为 1。指定数值 1 等价于使用选项-fno-align-jumps 且不会造成对齐。

-falign-labels[=number]

将所有分支目标对齐于那些大于或等于 number 的 2 的幂次方的边界，但仅当无需略过多于 number 个字节时有效。例如，如果 number 是 20，则有效的对齐边界是在 32 个字节的位置上，但同时又不能忽略多于 20 个字节。该选项将使代码变慢且变大，其原因是在分支目标前插入的空指令造成的。如果需要功能相仿但性能较好的版本，参见选项-falign-jumps。

如果使用了-falign-loops 或是-falign-jumps，但所指定的 number 大于该选项中指定的值，将使用这个更大的数值。如果没有指定 number，将使用机器的默认值，一般为 1。指定数值 1 等价于使用选项-fno-align-labels 且不会造成对齐。

-falign-loops[=number]

将所有循环顶端对齐于那些大于或等于 number 的 2 的幂次方的边界，但仅当无需略过

多于 number 个字节时有效。例如，如果 number 是 20，则有效的对齐边界是在 32 个字节的位置上，但同时又不能忽略多于 20 个字节。该选项将使代码变大，其原因是在分支目标前插入的空指令造成的，但根据机器的不同，循环可能因为每次迭代末尾的对齐跳转而运行得更快。

如果没有指定 number，将使用机器的默认值，一般为 1。指定数值 1 等价于使用选项 -fno-align-loops 且不会造成对齐。

### -fallow-single-precision

C

这是默认选项。其含义是不用双精度浮点运算替代单精度浮点进行数学操作。如果使用了-traditional，所有浮点操作均将按双精度执行。但如果使用了这一选项，就可以使用单精度运算了。

### -falt-external-templates

C++

该选项已经废止了。依赖于原始模板实例化的位置，模板实例可以或无需发生。实例化将符合#pragma interface 和#pragma implementation。参见-fexternal-templates。

### -fargument-alias

指定传递给函数的参数可以是其他参数的别名。也就是说，两个或多个参数可能代表同一个内存位置。还有一种可能就是参数还可以是全局变量的别名。该选项一般只用于编译程序内部。

对 C，C++和 ObjC 而言，这是默认选项。

参见-fargument-noalias 和-fargument-noalias-global。

### -fargument-noalias

指定传递给函数的所有参数均不能是其他参数的别名。也就是说，两个或多个参数将不会对应同一个内存位置。但有可能某个参数是一个全局变量的别名。该选项一般只用于编译程序内部。

参见-fargument-alias 和-fargument-noalias-global。

### -fargument-noalias-global

指定传递给函数的所有参数均不能是其他参数的别名。也就是说，两个或多个参数将不会对应同一个内存位置。而且该参数也不可能是全局变量的别名。该选项一般只用于编译程序内部。

对 Fortran 而言，该选项是默认的。

参见-fargument-alias 和-fargument-noalias。

### -fasm

C C++ ObjC

这是默认选项，它将打开关键字 asm，inline 和 typeof。

对 C 而言，指定-fno-asm 将关闭关键字 asm，inline 和 typeof。该选项对关键字__asm__，__inline__和__typeof__没有影响。

对 C++而言，指定-fno-asm 将关闭关键字 typeof，但由于关键字 asm 和 inline 是语言本身的一部分，因此对它们没有影响。

其他影响这些关键字的标志还有-ansi，-gnu-keywords 和-std。

### -fassume-compiled=classname

Java

根据某些类是否已经编译为本地代码，编译程序可以产生不同的代码。可以重复使用选项-fassume-compiled 和-fno-assume-compiled 来构造一组类的列表，假定它们是编译过的或是未编译的。

### -fasynchronous-unwind-tables

如果目标机支持，产生 DWARF2 格式的回退表。所生成的表格格式可由异步事件使用，如调试器和垃圾收集器。

参见选项-fexceptions，-fnon-call-exceptions 和-funwind-tables。

### -fautomatic

Fortran

这是默认选项。指定选项-fno-automatic 与为每个局部变量和数组指定一条 SAVE 指令的效果相同。该选项对公用块没有影响。可以通过指定选项-fno-automatic 关闭此功能。

参见选项-finit-local-zero。

### -fbackslash

Fortran

这是默认选项。指定-fno-backslash 后将阻止反斜线字符在 Hollerith 串中用做转义字符（像 C 字串中的用法）来定义特殊字符。

### -fbadu77-intrinsics-spec

Fortran

spec 指定了具有不恰当形式的 UNIX 特质的状态。spec 可以具有如下几种可能：

- enable 特质是可识别的且是打开的。这是默认状态。
- hide 特质是可识别的且当在一条 INTRINSIC 语句中首次出现时打开。
- disable 特质是可识别的，但对它们的引用必须通过 INTRINSIC 语句完成。
- delete 特质不可识别。

### -fbounds-check

**Java Fortran**

对 Java 而言，这是默认选项。指定选项-fno-bounds-check 将关闭所有对数组访问的边界检查。该选项将提高数组索引的性能，但当超出数组边界时，可能会造成不可接受的行为。

对 Fortran 而言，该选项产生的代码在运行时检查数组下标，并检查对 CHARACTER 子串的访问是否在所声明的最小和最大值内。

参见选项-ffortran-bounds-check。

### -fbranch-probabilities

在使用选项-fprofile-arcs 编译程序，并执行它来创建包含每个代码块执行次数的文件之后，程序可以利用这一选项再次编译，文件中所产生的信息将被用来优化那些经常发生的分支代码。如果没有这些信息，GCC 将猜测哪一分支经常发生并进行优化。这类优化信息存放在一个以源文件为名字以.da 为后缀的文件中。

参见选项-fguess-branch-probability。

### -fbuiltin

**C ObjC**

这是默认选项，用于通过名字来识别内建函数。选项-fno-builtin 指出，除非利用前缀__builtin_进行引用，否则不识别所有内建函数。例如，为了获得内建版本，应该调用__builtin_strcpy()而不是名为 strcpy()的函数。

为了避免使用选项-fno-builtin 而抑制所有内建函数，可将函数的名称追加到选项名之后，有选择地指定不使用哪些内建函数。例如，想使用除 bzero()和 sqrt()之外的所有内建函数，可以使用如下形式：

```
-fno-builtin-bzero -fno-builtin-sqrt
```

对 C++而言，-fno-builtin 始终有效，因此惟一可以直接调用 C 内建函数的方法就是使用__builtin_前缀，GNU C++标准库就使用了很多内建函数。

参见选项-fbuiltin-function，-ffreestanding 和-fnonansi-builtins。

### -fcall-saved-register

将指定的 register 作为一个可分配来包含数据的寄存器，而且即使经过函数调用仍然可以获取数据。利用这一选项编译的函数必须完成对该寄存器内容的保存和恢复。

对于具有固定作用的寄存器（如栈指针和堆指针），绝对不能使用该选项。

寄存器的名字和机器平台相关，均在机器描述的宏 REGISTER_NAMES 中给出。

参见选项-fcall-used-register 和-ffixed-register。

### -fcall-used-register

将指定的 register 看作可用于分配的寄存器，但其值将在函数调用中被破坏。该寄存器可用做临时存储，但经过函数调用后必须重载。

对于具有固定作用的寄存器（如栈指针和堆指针），绝对不能使用该选项。

寄存器的名字和机器平台相关，均在机器描述的宏 REGISTER_NAMES 中给出。

参见选项-ffixed-register 和-fcall-saved-register。

### -fcaller-saves

在调用函数之前需要加入额外的指令来保存寄存器，并且在返回函数时需要恢复它们。之后，这些寄存器就可以在函数调用和函数体内使用了。只有包含有用数据的寄存器需要保存，而且只有保存和恢复这些寄存器相比在需要时重新加载它们更好时才会这么做。默认情况下，该选项在某些机器上是打开的，而且在指定了选项-O2，-O3 和-Os 时均是打开的，但可以通过选项-fno-caller-saves 将其关闭。

### -fcase-initcap

**Fortran**

要求大部分源代码用起始大写字母书写。设置选项-fintrin-case-initcap，-fmatch-case-initcap，-fsource-case-preserve 和-fsymbol-case-initcap。

### -fcase-lower

**Fortran**

将源代码映射为小写的。设置选项-fintrin-case-any，-fmatch-case-any，-fsource-case-lower 和-fsymbol-case-any。

### -fcase-preserve

**Fortran**

保留用户定义符号的大小写，允许任意大小写的关键字和特质匹配。设置选项-fintrin-case-any，-fmatch-case-any，-fsource-case-preserve 和-fsymbol-case-any。

### -fcase-strict-lower

**Fortran**

要求大部分源代码用小写字母书写。设置选项-fintrin-case-lower，-fmatch-case-lower，-fsource-case-preserve 和-fsymbol-case-lower。

### -fcase-strict-upper

**Fortran**

要求大部分源代码用大写字母书写。设置选项-fintrin-case-upper，-fmatch-case-upper，-fsource-case-preserve 和-fsymbol-case-upper。

### -fcase-upper

**Fortran**

将源代码映射为大写的。设置选项-fintrin-case-any，-fmatch-case-any，-fsource-case-upper和-fsymbol-case-any。

### -fcheck-new

**C++**

在C++的分配内存操作中，插入代码检查new运算符返回的指针不为NULL指针。这一操作通常是不必要的，因为C++库中的new在内存不足时会产生一个异常。如果重载后的new可能返回而不产生异常，该选项就可以用来检查返回指针了。

### -fcheck-references

**Java**

当通过引用访问对象时，插入内嵌代码检查空指针引用。因为大多数处理机都会检测到这种空指针引用，这一操作通常是不必要的。

### -fcommon

**C**

这是默认选项。指定-fno-common将使编译程序在数据区为每个全局变量显式地分配空间。默认情况是在一个公用块中分配空间，并由链接器解析，因此即使将同一全局变量声明多次也会使链接器将其解析到同一地址。

可以使用选项-fno-common来检验需要编译和链接到其他不使用GCC程序的系统。

对Fortran而言，决不可以使用-fno-common选项。

参见第4章的4.3.5节“属性”。

### -fcompile-resource=resourcename

**Java**

resourcename是包含属性定义及其他资源的文件名，这些资源将编译到目标代码中，并在运行时由核心协议处理函数作为core:/resourcename访问。

### -fcond-mismatch

**C**

在条件表达式中允许类型不匹配。

### -fconserve-space

**C++**

将在编译时不初始化的全局变量放在公用段中（如C中的做法）。由于直到程序加载时才分配空间，这将降低可执行文件的尺寸。对大部分平台来说，由于已有这样的支持，能够将变量保存在BSS段中而无需将变量作为公用的，因此这一选项已不再有价值了。

**警 告** 如果使用这一选项造成程序在终止时崩溃，可能是因为某个对象的多次定义被合并且分配到同一地址上，从而进行两次释放造成的。

### -fconst-strings

C++

这是默认选项。如果指定了选项-fno-const-strings，字串声明将被定义为 char *而不是 const char *。为了能够对字串进行写操作，同时还需要使用选项-fwriteable-strings。

### -fconstant-string-class=classname

ObjC

对于每个形式为@"..."的字串，使用指定的 classname 作为实例化的类名。默认的 classname 是 NXConstantString。

### -fcprop-registers

在所有寄存器分配完成后，该选项分析复制到寄存器中的数据模式，试图发现那些不需要在寄存器中复制数值的地方，但为了能够再次使用，会向前传播前一次复制。该选项由选项-O 设置，但可以使用选项-fno-cprop-registers 关闭。

### -fcse-follow-jumps

当跳转目标除了所采用的跳转以外均不可达时，在跳转路径后将产生公用子表达式消除扫描。也就是说，所有在跳转前存在的数值将在跳转目的那一点仍然存在并可以使用。选项-O2，-O3 和-Os 将设置这一选项，但可以利用选项-fno-cse-follow-jumps 关闭。参见选项-fcse-skip-blocks 和--param。

### -fcse-skip-blocks

如果一条 if 语句体非常简单，以至于不包含任何会影响先前计算的数值，公用子表达式分析流将会跳过这一 if 语句并作用于随后的语句。选项-O2，-O3 和-Os 将会打开这一选项，但可以通过选项-fno-cse-skip-blocks 关闭这一选项。参见-fcse-skip-blocks 和--param。

### -fdata-sections

在汇编语言输出中，每个数据项均被放置在其自己的小节中。小节名是通过数据项名派生的。仅在具有可以使用小节方式来优化空间分配的链接器的机器上，这一选项才有优势。如果对可执行代码进行同样的优化，参见-ffunction-sections。

在汇编器代码不支持分节的机器上，设置该选项将会产生一条警告消息，并且被忽略。即使在支持分节的机器上，这一选项也没有优势，除非链接器利用这一组织进行优化。事实上，由于会造成目标代码更大和加载更慢，该选项还会具有副作用。

如果设置选项-p 进行性能分析，该选项将失效。而且，由于代码被重新组织，可能会在使用选项-g 和进行调试时出现问题。

-fdefault-inline

C++

这是默认选项。在类声明时，如果成员函数具有已定义的函数体，不管是否使用了关键字 inline 进行声明，它都将被定义为是内嵌的。为了防止自动内嵌，可以使用选项-fno-default-inline。参见--param。

-fdefer-pop

调用函数时压入栈的参数将不会在函数返回时立即弹出，而且允许在多次函数调用中累积，并在最后一次清除。默认情况下该选项被打开，因此要强制在每次函数调用后清除栈，需要使用选项-fno-defer-pop。

-fdelete-null-pointer-checks

如果数据流分析表明某指针不会为空，检查指针是否为空的代码将被删除。在某些环境下，可能要处理检查空指针的结果，因此该选项不能在依赖这一结果的程序中使用。选项-O2，-O3 和-Os 将打开这一选项，但可以通过选项-fno-delete-null-pointer-checks 关闭这一选项。

-fdelayed-branch

该选项仅在具有延迟分支通道的机器上有效。这一选项不得不在处理加载和执行指令时，同时决定是否采用某一分支。在作出了决策后，根据指令的位置和决策的结果，指令的结果可能会被丢弃。如果目标机支持，每一级优化均会打开该选项，但可以通过选项-fno-delayed-branch 关闭该选项。参见--param。

-fdiagnostics-show-location=where

诊断消息（错误和警告）有可能很长且是分行的，因此显示时会有很多行。where 的默认设置是 once，它会使引发诊断消息的源文件位置仅被包含一次。将 where 设置为 every-line 时将使诊断消息的每一行均包含源代码位置。

该选项可能在某些情况下没有实现，而且只有将选项-fmessage-length 设置为大于 0 时才会有效。

-fdollar-ok

Fortran

允许在符号名中使用美元符号$。

-fdollars-in-identifiers

C C++

在标识符中接受字符$作为合法字符。使用选项-fno-dollars-in-identifiers 可以显式地禁止这一功能。默认设置取决于平台和语言。传统 C 允许使用，但现在的标准则禁止使用，

因此如果你需要使用的话，最好显式指定该选项。

**-fdump-class-hierarchy[-format]** C++

每个类都会将层次信息和虚函数表输出到一个同名但以.class 为后缀的文件中。可选的 format 可为如下形式之一：

- address 打印每个节点的地址，并利用这些地址交叉引用其他输出中的树节点，如利用选项-d 产生的内容。
- slim 通过禁止输出函数体或是域成员减少输出的尺寸。
- all 打开所有选项增加输出的尺寸。

**-fdump-translation-unit[-format]** C C++

将编译程序的内部树结构输出到与源代码同名但以.tu 后缀结尾的文件中。可选的 format 可为如下形式之一：

- address 打印每个节点的地址，并且利用这些地址交叉索引其他输出的树节点，如选项-d 产生的内容。
- slim 通过禁止输出函数体或是域成员减少输出的尺寸。
- all 打开所有选项增加输出的尺寸。

**-fdump-tree-switch[-format]** C++

将中间语言树的不同阶段输出到文件中。文件名由源文件名加上 switch 部分中指定的部分作为后缀组成。switch 必须是如下形式之一：

- original 在对树进行任何优化之前，将树结构输出到文件 name.original 中。
- optimized 在所有对树的优化后，将树结构输出到文件 name.optimized 中。
- inlined 在完成函数内嵌后，将树结构输出到文件 name.inlined 中。

可选的 format 可为如下形式之一：

- address 打印每个节点的地址并且利用这些地址交叉索引其他输出的树节点，如选项-d 产生的内容。
- slim 通过禁止输出函数体或是域成员减少输出的尺寸。
- all 打开所有选项增加输出的尺寸。

**-fdump-unnumbered**

当使用选项-d 调试编译程序时，该选项将抑制输出文件中的指令号和行号，这样可以方便使用 diff 比较输出。

-felide-constructors **C++**

这是默认选项。当产生的代码所调用的函数按值返回一个对象时，如果能直接在返回的位置上产生对象，而不是复制构造函数中已构建的对象，这将使代码大大简化。但如果构造函数具有副作用，这也会引发问题，可以利用-fno-elide-constructors 关闭这一选项。

-femulate-complex **Fortran**

通过仿真而不是可以提供直接支持的 gcc 后端实现复数算术。

该选项是为了绕过 gcc 复数算术中的错误而实现的，但这个错误应该已经修正了。

-fencoding=name **Java**

指定用于读取源文件的某一特定字符集的编码名。默认情况下是计算机当前的本地设置，或者如果没有指定本地设置的话，默认会使用 UTF-8。

-fenforce-eh-specs **C++**

这是默认选项。根据 C++标准，GCC 所产生的代码将强制检查异常违例，但可以利用选项-fno-enforce-eh-specs 抑制这些代码的产生。不检查违例的代码大小会有所减少。

-fexceptions

打开异常处理。这一选项会生成必要的代码来处理异常的抛出和捕获。如果没有指定这一选项，对于 Ada，Java 和 C++等会触发异常的语言来说，默认都会指定该选项。

所生成的代码不会造成性能损失，但会造成尺寸上的损失。因此，如果想要编译不使用异常的 C++代码，可能需要指定选项-fno-exceptions。

参见选项-fnon-call-exceptions，-funwind-tables 和-fasynchornous-unwind-tables。

-fexpensive-optimizations

该选项打开一些耗费编译时间但却非常有效的优化操作。例如，在全局公用子表达式删除后再次执行公用子表达式删除。当该选项被打开时，其他一些优化操作会执行得更彻底。选项-O2，-O3 和-Os 自动打开这一选项，但可以通过-fno-expensive-optimizations 强行关闭这一选项。

-fexternal-templates **C++**

该选项已废止。根据模板定义的位置，模板实例可能会产生或不产生。实例化过程将遵从#pragma interface 和#pragma implementaion。参见选项-falt-external-templates。

### -ff2c

**Fortran**

这是默认选项，它指定生成与 f2c 兼容的代码。

指定选项-fno-f2c 将抑制产生与 f2c 兼容的代码并使用 GNU 调用规范。但该选项对与库 libf2c 接口的代码无效，只是不允许来自 libf2c 的特质作为参数传递。

如果要链接为一个程序，所有代码都必须一致地使用选项-fno-f2c。

### -ff2c-intrinsics-spec

**Fortran**

spec 指定具有不恰当形式的 f2c 相关特质的状态。spec 可以是如下形式之一：

- enable　f2c 相关特质可以识别并打开。这是默认状态。
- hide　只有当首次在 INTRINSIC 语句中涉及名字时，f2c 相关特质才可被识别并打开。
- disable　f2c 相关特质可以识别，但对它们的引用必须通过 INTRINSIC 语句完成。
- delete　f2c 相关特质不被识别。

### -ff66

**Fortran**

被编译的源码具有 Fortran 66 方言。

参见选项-ff77 和-ff90。

### -ff77

**Fortran**

被编译的源码具有 Fortran 77 方言。

该选项同时也指定这种方言是 f2c 所需要的，f2c 是种可以将 Fortran 代码转换为 C 代码的工具。

参见选项-ff66 和-ff90。

### -ff90

**Fortran**

编译程序将识别某些 Fortran 90 方言中的内容。

很多 Fortran 90 中的内容可以通过使用选项-fvxt 和-ff90-instrinsics-enable 打开。

参见选项-ff66 和-ff77。

### -ff90-intrinsics-spec

**Fortran**

spec 指定具有不恰当形式的 Fortran 90 特质的状态。spec 可为如下形式之一：

- enable　Fortran 90 特质可以识别并打开。这是默认情况。
- hide　仅在首次涉及 Fortran 90 特质名字是在 INTRINSIC 语句中时，才可以识别并打开。

- disable Fortran 90 特质可以识别，但对它们的引用必须通过 INTRINSIC 语句完成。
- delete Fortran 90 特质不可以识别。

参见选项-ff90 和-fvxt。

### -ffast-math

在违反某些 ISO 和 IEEE 规则时，一些数学运算可以加速完成。例如，如果设置了这一选项，就可以认为所有给 sqrt()的参数均为非负，因此所有的浮点值就都合法了。

设置这一选项将使预处理程序定义宏__FAST_MATH__并同时设置选项-fno-math-errno，-funsafe-math-optimizations 和-fno-trapping-math。设置选项-fno-fast-math 的同时也将设置选项-fmath-errno。

### -ffixed-register

视指定的 register 为一个不能由编译程序分配的固定寄存器。但它仍可能用作堆指针、栈指针或其他某些固定用途。

寄存器的名称与平台相关，并定义在机器描述的宏 REGISTER_NAMES 中。

参见选项-fcall-used-register 和-fcall-saved-register。

### -ffixed-form

**Fortran**

这是默认选项。它指定 Fortran 源代码必须是传统的固定形式，而不能使用 Fortran 90 中的自由格式。

该选项与选项-fno-free-form 相同。

### -ffixed-line-length-len

**Fortran**

在固定形式的源代码输入中，数字 len 指定了其后字符全部忽略的列号。

设置选项-ffixed-line-length-0 和-ffixed-line-length-none 将可以允许任意长度的源代码输入。

len 的数值一般为 72（在串接的 80 列卡片中留 8 个空白）。

### -ffloat-store

不分配寄存器保存浮点值。在某些机器上，该选项可能引起寄存器将浮点值扩展到超出语言所定义的精度范围，因此将精度更高的浮点值存储到内存中。

默认设置是-fno-float-store，即可以使用寄存器。

当程序必须严格地将精度限制为 IEEE 标准定义的精度时，该选项非常有用。

### -ffor-scope

C++

默认情况是遵守标准。这项设置确定了在 for 语句初始化部分声明的变量的作用域。

指定选项-ffor-scope 将变量的作用域限制为循环体。

指定选项-fno-for-scope 可将变量的作用域进行限制，由声明该变量的位置开始，到包含 for 语句的循环体的结束位置。下面是使用该选项的合法例子：

```
#include stdio.h
int main(int argc,char *argv[])
{
    for(int i=0; i10; i++) {
        printf("Loop one %d\n",i);
    }
    printf("Out of loop %d\n",i);
    return(0);
}
```

### -fforce-addr

必须将地址复制到寄存器中才能对它们进行算术运算。由于所需要的地址经常在前面就已经加载到寄存器中，之后就不需要加载了，因此这将改进所生成的代码。默认选项是-fno-force-addr。参见选项-fforce-mem。

### -fforce-classes-archive-check

Java

该选项强制检查类文件 java.lang.Object 中的一个属性，保证它是由 GNU 编译程序编译的。该属性具有零尺寸，名称为 gnu.gcj.gcj-compiled。除了编译程序的输出是字节码外，该属性均被检查。

### -fforce-mem

必须首先将数值复制到寄存器中才能对它们进行算术运算。所需的数值经常在前面已经被加载到寄存器中（而无需再次加载），因此这将大大改进代码。选项-O2，-O3 和-Os 将打开该选项；默认情况是选项-fno-force-mem。参见选项-fforce-addr。

### -ffortran-bounds-check

该选项会进行运行时的检查，验证数组子脚本和 CHARACTER 子串访问在所声明的最小和最大范围内。

与选项-fbounds-check 相同。

-ffree-form **Fortran**

指定 Fortran 源代码采用与 Fortran 90 相似的自由格式。

与选项-fno-fixed-form 相同。

-ffreestanding **C**

所编译的程序将在独立环境中运行，该环境可能没有标准库，而且可能也不从 main() 开始运行。该选项设置-fno-builtin 并与选项-fno-hosted 相同。

-ffunction-cse

这是默认选项。函数调用通过将函数地址保留在寄存器中实现。使用选项-fno-function-cse 将引起每条调用指令隐式地包含函数地址。默认情况下会生成非常有效的代码。参见选项--param。

-ffunction-sections

在汇编语言输出中，每个函数均被放置在其自己的小节中。小节名是由函数名派生的。只有在某些链接器根据分节方式优化空间分配的机器上，这种方法才有优势。要对数据进行同样的优化，参见选项-fdata-sections。

在不支持分节的机器上使用该选项将引起警告消息，而且该选项将被忽略。甚至在支持分节的机器上，如果没有链接器进行优化组织，这也是没有效果的。事实上，它也会使目标代码变大而加载速度变慢，带来不利影响。

如果设置了选项-p 进行性能分析，该选项将失效。而且，由于代码会重新组织，在使用-g 选项进行调试时，可能会发生问题。

参见第 4 章 4.3.5 节“属性”。

-fgcse

执行全局公用子表达式消除优化操作。如果程序中具有参与计算的 goto 语句，该选项可能具有不利影响。选项-O2，-O3 和-Os 将打开该选项；可以使用-fno-gcse 关闭该选项。参见选项--param。

-fgcse-lm

通过探测循环中的加载和存储操作，执行全局公用子表达式消除优化操作，其中加载操作可以放置在循环外，可以只执行一次。这是默认选项，但在没有指定选项-Os 时没有作用。可以利用-no-fgcse-lm 关闭该选项。

### -fgcse-sm

通过探测循环中的加载和存储操作，执行全局公用子表达式消除优化操作，其中存储操作可以放置在循环外，可以只执行一次。这是默认选项，但在没有指定选项-Os 时没有作用。可以利用-no-fgcse-sm 关闭该选项。

### -fglobals

**Fortran**

这是默认选项。指定选项-fno-gloabls 将关闭这种具有相同过程名，但具有不同参数类型的全局冲突诊断。该选项也会关闭函数内嵌以防止生成不正确代码而引发的崩溃。

### -fgnu-intrinsics-spec

**Fortran**

spec 指定了具有不恰当形式的 GNU 特质的状态。spec 可以是如下形式之一：

- enable　GNU 特质可以被识别并设置为有效。这是默认选项。
- hide　仅在首次涉及 GNU 特质名字是在 INTRINSIC 语句中时，GNU 特质才能被识别并有效。
- disable　GNU 特质可以被识别，但只有通过 INTRINSIC 语句才能引用它们。
- delete　GNU 特质不可识别。

### -fgnu-keywords

**C++**

这是默认选项。选项-fno-gnu-keywords 禁止使用关键字 typeof。参见选项--ansi。

### -fgnu-linker

这是默认选项。如果指定了选项-fno-gnu-linker，则说明将不使用 GNU 链接器，某些执行全局初始化的代码（如构造函数和析构函数）就无法保证正确执行了。

在需要该选项的系统中，gcc 被配置为自动调用 collect2。

### -fgnu-runtime

**ObjC**

对大多数系统而言，这是默认选项。该选项指导编译程序产生使用标准 GNU Object C 运行时的代码。

### -fguess-branch-probability

这是默认选项。GCC 将猜测哪个分支更可能被选取，并以此优化代码。

某些情况下使用随机模型进行猜测，因此编译同样的源代码有可能产生不同的代码。如果需要关闭该选项并强制生成相同的代码，可以指定选项-fno-guess-branch-probability。对于其他防止随机化的选项，每次均生成相同的代码，参见选项-fbranch-probabilities 和 -fprofile_arcs。

### -fhash-synchronization

Java

该选项将引起 synchronize，wait 和 notify 的位置被存放于一个哈希表中（而非每个对象中）。

### -fhosted

C

所编译的程序需要运行在宿主环境中，其中需要有完整的标准库而且 main()具有 int 返回类型。该选项设置-fbuiltin。该选项与选项-fno-freestanding 相同。

### -fident

Pre

这是默认选项。如果指定了选项-fno-indent，预处理程序将忽略#ident 指示字。

### -fimplement-inlines

C++

这是默认选项。以嵌入方式生成的函数也具有定义它们时的函数体。选项-fno-implement-inlines 将抑制生成#pragma implementation 控制的嵌入函数体的代码。如果没有生成函数体，每一次调用都需要生成内嵌代码。

### -fimplicit-templates

C++

这是默认选项。对尚不存在的模板或模板函数的引用将引起模板的实例化。除了那些编译为内嵌函数的模板以外，选项-fno-implicit-templates 将抑制隐式的模板实例化。参见选项-fimplicit-inline-templates。

### -fimplicit-inline-templates

C++

这是默认选项。对尚不存在的模板或模板函数的引用将引起模板的实例化。选项-fno-implicit-templates 将抑制隐式的模板实例化，包括那些编译为内嵌函数的模板。参见选项-fimplicit-templates。

### -finhibit-size-directive

不输出.size 汇编指令，也不输出其他可能在分割函数为两部分时引起问题的指令。该选项是一种用来编译 crtstuff.c（GCC 的一部分）的特殊情况，预计不会有其他目的。

### -finit-local-zero

Fortran

对编译单元声明的所有局部变量都初始化为 0。对于公用块和作为参数的变量无效。建议与选项-fno-automatic 一起使用，以防止初始化自动变量而引起的运行时代价。

### -finit-priority

该选项只供编译程序内部使用，它用来指定运行时的初始化次序。

### -finline

C C++ ObjC Fortran

这是默认选项，它允许用关键字 inline 指定函数在其被调用的地方展开。使用选项-fno-inline 将使编译程序忽略关键字 inline。注意，除非利用选项-O 设置了某一级别的优化操作，否则是不会展开内嵌函数的。参见选项--param 和第 4 章的 4.3.5 节“属性”。

### -finline-functions

允许编译程序选择某些简单的函数在其被调用处展开。如果函数的声明方式使其所用的调用均是已知的（例如，C 源文件中的静态函数不能从文件外部调用），由于它从不能被调用，其函数体将被忽略。除非使用选项-fno-inline-functions，否则当设置选项-O3 时该选项会自动打开。参见选项-fkeep-inline-functions，--param 和第 4 章的 4.3.5 节“属性”。

### -finline-limit=size

对伪指令超过一定数目的函数，编译程序将不进行展开。size 的默认值是 600。参见选项--param。

### -finstrument-functions

在每个函数的入口和出口处插入代码调用某个函数。函数调用的原型如下：

```
void __cyg_profile_func_enter(void *this_fn,void *call_site);
void __cyg_profile_func_exit(void *this_fn,void *call_site);
```

参数 this_fn 是被调函数的地址，可以通过符号表信息进行标识。参数 call_site 标识调用者。（某些平台尚没有 call_site 信息。）

如果函数是展开的内嵌函数，函数调用就插在内嵌代码前后。为了达到标识的目的，即便所有调用均使用内嵌代码，也必须存在一个函数的非内嵌版本。

为防止某个函数被插入代码，它可被声明为具有属性 no_instrument_function。这对于中断处理函数以及性能分析例程不能调用的函数是非常有用的。

参见第 4 章 4.3.5 节“属性”。

### -fintrin-case-spec

spec 确定了特质名的大小写。可以为如下形式之一：

- initcap 名称的首字大写，其他小写。
- upper 名称全部大写。

- lower 名称全部小写。这是默认选项。
- any 名称可以是大小写的任意组合。

参见选项-fmatch-case-，-fsource-case-，-fsymbol-case-和-fcase-。

### -fjni

**Java**

该选项将本地方法编译为 JNI，而非默认的 CNI。同时生成调用底层 JNI 方法的存根（stub）。

### -fkeep-inline-functions

即使对函数的所有引用均被扩展为内嵌函数，也就是不存在对函数的调用，编译程序也将生成函数体。默认选项是-fno-keep-inline-functions，它指定编译程序不生成没有被调用的函数体。参见选项-finline-functions 和--param。

### -fkeep-static-consts

除非设置了某些级别的优化选项，否则这是默认选项。对编译体而言，常量的数值即使没被引用，也将分配存储空间。为了防止为无用常量分配空间，可以使用选项-fno-keep-static-consts。

### -fleading-underscore

该选项强制写入目标文件的每个符号均以下划线开始。选项-fno-leading-underscore 将抑制增加下划线字符。

该选项用于链接已有汇编代码。

### -fmatch-case-spec

**Fortran**

spec 指定 Fortran 关键字的大小写。它可为如下形式之一：

- initcap 名称的首字母大写，其余小写。
- upper 名称全部大写。
- lower 名称全部小写。这是默认选项。
- any 名称可以是大小写的任意组合。

参见选项-fintrin-case-，-fsource-case-，-fsymbol-case-和-fcase-。

### -fmath-errno

这是默认选项。诸如 sqrt()之类的数学函数错误代码将被存放在名为 errno 的全局变量

中。设置选项-fno-math-errno 将使错误代码不被存放在 errno 中，从而干扰标准的 IEEE 异常处理。参见选项-ffast-math。

### -fmem-report

编译程序结束时，它会打印一张详细的列表，其中列出为每个数据类型所分配的空间和其他永久性存储分配信息。

### -fmemoize-lookups

C++

最近的内部符号表查找结果将被缓存，以便加速后续查找。

### -fmerge-all-constants

该选项设置-fmerge-constants，而且能够合并重复的字符串和数组。标准 C 和 C++要求每个变量都有不同的存储位置，因此该选项可能会生成不符合标准的目标代码。

### -fmerge-constants

该选项试图将跨编译单元的所有常量值（除了字串）合并在一个副本中。当打开某级优化选项时，这是默认选项。默认选项是-fno-merge-constants，它只允许合并同一编译单元中的常量。

### -fmessage-length=size

错误消息将被格式化，从而不会超过 size 个字符。如果 size 为 0，就不进行格式化，并且每个错误消息仅出现在同一行中。对 C++而言，默认尺寸是 72，而其他语言均为 0。该选项在某些情况下可能没有实现。

### -fmil-intrinsics-spec

Fortran

spec 指定了具有不恰当形式的 MIL-STD-1753 特质的状态。spec 可以是如下形式之一：

- enable　MIL-STD-1753 特质可以识别并打开。这是默认选项。
- hide　MIL-STD-1753 特质可以识别，但仅在首次涉及特质名字是在 INTRINSIC 语句中时才被打开。
- disable　MIL-STD-1753 特质可以识别，但只有通过 INTRINSIC 语句才能引用它们。
- delete　MIL-STD-1753 特质不可识别。

### -fmove-all-movables

所有的不变表达式将被移到循环体外。这一过程的好与坏取决于源码中循环的结构。除了 Fortran 外，默认选项是-fno-move-all-movables。参见选项-freduce-all-givs。

### -fms-extensions

C++

当使用 MFC 定义的构造函数时，关闭警告消息，如数据声明中的隐式 int 定义，使用非标准语法获取成员函数的指针等等。默认选项是-fno-ms-extensions，它将产生警告消息。

### -fnext-runtime

产生与 NeXT 运行时兼容的输出。对于基于 NeXT 的系统，如 Darwin 和 Mac OS X，该选项是默认的。

### -fno-*

任何以-fno-开头的选项都有两种形式，并按照字母顺序排在名字后面——这里没有加入 no-。例如，可以在选项-ffor-scope 下找到选项-fno-for-scope 的描述。大多数以-f 开头的选项均有一个相伴的-fno 形式，但也有例外。

### -fnon-call-exceptions

产生的代码可供陷阱指令（如非法浮点运算或非法内存寻址）抛出异常。由于需要平台相关的运行时支持，该选项并不普遍有效。

该选项只受限于硬件陷阱信号，但不包括通用信号，如 SIGALRM 或 SIGTERM。

参见选项-fexceptions，-funwind-tables 和-fasynchronous-unwind-tables。

### -fnonansi-builtins

C++

这是默认选项。指定选项-fno-nonansi-builtins 将禁止内建函数的自动生成，这里的内建函数并非是 ASNI/ISO C 所需要的。参见选项-ansi 和-fbuiltin。

### -fomit-frame-pointer

对于不需要栈指针的函数，就不在寄存器中保存栈指针，因此忽略存储和检索地址的代码，并将其他寄存器用于普通用途。所有-O 优化选项的级别都自动设置这一选项，但这只当调试器可以不依靠栈指针运行时才可行。如果调试器的运行不能没有栈指针，就必须显式地设置它。某些平台没有栈指针，因此该选项也没有效果。默认选项是-fno-omit-frame-pointer。

### -fonetrip

Fortran

DO 循环至少被执行一次（循环测试是在循环底而非循环顶处执行的）。

在 Fortran 77 之前，某些编译程序在循环体的底部进行循环检测，而另外一些则在循环体的顶部进行检测。从 Fortran 77 开始，所有测试均在循环体顶部进行，也就是说，如果一开始的检测为假，那么循环体将不被执行。

### -foperator-names

C++

这是默认选项。指定选项-fno-operator-names 将阻止编译程序将 and，bitand，bitor，compl，not，or 和 xor 识别为&&，&，|，~，!，||和^等运算符的替代关键字。

### -foptimize-register-move

通过改变移动内存中数据位置的操作中的寄存器赋值，可以优化寄存器的分配。这对某些可直接在内存间搬运数据的机器来说是非常有效的。选项-O2，-O3 和-Os 可以打开该选项，但可用-fno-optimize-register-move 关闭这一选项。

### -foptimize-sibling-calls

优化尾递归调用和同属调用。选项-O2，-O3 和-Os 将自动打开该选项。默认选项是-fno-optimize-sibling-calls。

下面是一个尾递归调用的例子：

```
int rewhim(int x,int y) {
    . . .
    return(rewhim(x+1,y));
}
```

优化操作后，程序不会产生一个新的调用，仅插入一条命令直接跳转到函数的顶部。类似地，下面的同属调用也可以进行优化：

```
int whim(int x,int y) {
    . . .
    return(wham(x+1,y));
}
```

在同属调用中，必须调用函数 wham()，但函数 whim()的堆栈框架可以直接删除，这样函数 wham()就直接将返回值返回给函数 whim()的调用者了。

### -foptimize-static-class-initialization

Java

这是默认选项。当优化选项被设置为-O2，-O3 或-Os 并且输出的是目标代码而非字节码时，静态类将在首次使用时进行初始化。可以使用选项-fno-optimize-static-class-initialization 关闭该优化。

### -foptional-diags

C++

这是默认选项。指定选项-fno-optional-diags 将抑制诊断消息，根据 C++标准，编译程序并不需要输出这些消息。

--for-assembler optionlist **Asm**

与选项-Wa 相同。

--for-linker option **Linker**

与选项-Xlinker 相同。

--force-link name **Linker**

与选项-u 相同。

-foutput-class-dir=directory **Java**

同选项-C 一起使用时，由编译程序输出的类文件将存放在 directory 中（或相应的 directory 子目录中），而非当前目录。

该选项也可以写作--output-class-directory。

-fpack-struct

将结构成员打包在一起，这样彼此间就不能插入对齐空间了。

该选项将引起访问可执行代码的结构成员的效率降低，而且可能引起代码与系统库的不兼容。

对 Fortran 而言，绝对不可以使用选项-fpack-struct。

参见第 4 章的 4.3.5 节“属性”。

-fpedantic **Fortran**

参见 Fortran 的选项-pedantic。

-fpeephole

这是默认选项，但可通过选项-fno-peephole 关闭。输出汇编语言时，打开窥孔优化。它将匹配指令的集合，并用优化的版本替代它们。该选项在优化选项没有打开时无效。该选项是与平台相关的，可能对某些平台无效。

-fpeephole2

在寄存器分配后但未进行调度时，打开 RTL 窥孔优化。该优化是将一组指令集合进行机器相关的翻译，翻译成另外一组指令。该选项是平台相关的，因此可能对某些平台无效。如果没有打开优化，该选项也无效。选项-O2，-O3 和-Os 将设置该选项，但可以通过选项-fno-peephole2 关闭该选项。

### -fpermissive

C++

指明代码与标准不相符的诊断消息会作为警告（而不是错误）输出。如果没有指定选项-fpermissive 或-pedantic，默认打开选项-fpedantic-errors。

### -fpic

生成可用于共享库的位置独立代码（PIC）。所有的内部寻址均通过全局偏移表(GOT)完成。要确定一个地址，需要将代码自身的内存位置作为表中一项插入。该选项需要操作系统支持，因此并不是在所有系统上均有效。

该选项产生可以在共享库中存放并从中加载的目标模块。

如果链接器对选项-fpic 产生了错误消息，说明位置独立代码不能工作，可以使用选项-fPIC。

某些系统的偏移表有尺寸限制。Motorola 的 m88k 上的尺寸限制是 16k，m68k 和 RS/6000 上是 32k，Sparc 上是 8k。位置独立代码需要一定的支持，因此它可能仅在某些机器上有效。

参见选项-fPIC 和-shared。

### -fPIC

该选项与选项-fpic 相同，但它可以克服在 m68k，m88k 和 Sparc 上可能遇到的偏移表尺寸限制。

参见选项-fpic 和-shared。

### -fppc-struct-return

生成的代码可将结构存储在内存中返回，即使结构可能小到可以在寄存器中保留。将返回的结构存储在内存还是寄存器的具体规范取决于平台。

尽管产生的代码效率可能不高，但当需要连接其他编译程序的代码文件时就需要该选项了。

在 Fortran 中，该选项仅在程序要编译 libg2c 版本时才会使用。

参见选项-freg-struct-return。

### -fprefetch-loop-arrays

如果平台支持，生成预取数组的指令，可改进循环的性能。

### -fpreprocessed

Pre

即使命令行含有后缀，指出文件需要预处理，也不进行预处理。表 D-1 给出了这些后缀。

即使指定了该选项，也可以使用选项-C 去掉预处理程序的注释。

### -fpretend-float

当为另一个系统交叉编译目标文件时，该选项将浮点运算格式化成就好像在本地机上运行一样。其结果很可能是一种无法在目标机上运行的格式，但指令序列和为目标机而产生的指令序列一致。

### -fprofile-arcs

在使用这一选项编译程序，并运行它创建包含每个代码块执行计数的文件后，程序可以再次使用选项-fbranch-probabilities 编译，文件中的信息可以用来优化那些经常选取的分支。如果没有这些信息，GCC 将猜想哪条路径可能被选取而进行优化。这些信息被存放在一个与源文件同名但以.da 为后缀的文件中。

该选项的另一个用处是同选项-ftest-coverage 一起使用来支持 gcoy。该选项组合将为程序中的每个函数创建一个流程图，然后由流程图确定一个生成树。然后，代码将被插到生成树中没有的函数里，生成树将为每次执行产生一个计数。对于只有一个入口和出口的代码块来说，代码将直接加在代码块中。具有多个入口和出口的代码块将会产生新的跟踪每个入口和出口的代码块。

用这些选项编译的程序和 gcov 一起运行，将会比用选项-a 和-ax 编译时慢一些，但用选项-a 产生的计数不能提供足够的信息来预测所有分支可能性。

参见选项-a，-ax，-fbranch-probabilities 和-fguess-branch-probability。

### -freduce-all-givs

强制所有通用归纳变量（循环计数器）都要进行强度削弱。该选项是否会产生好的代码取决于源代码中循环的结构。除 Fortran 外，默认选项是-fno-reduce-all-gives。

参见选项-fmove-all-movables。

### -freg-struct-return

生成用寄存器返回短结构的代码。如果不够小，无法容纳在一个寄存器中，将使用内存返回。用内存或寄存器返回结构的具体规范和平台相关。

在 Fortran 中，该选项仅在程序要编译 libg2c 版本时才会使用。

参见选项-fpcc-struct-return。

### -fregmove

与选项-foptimize-register-move 相同。

### -frename-registers

这是一种优化技术，它会试图去掉被调度代码中的假依赖关系，这是通过使用寄存器

分配和调度完成后的寄存器实现。这一优化操作对具有大量寄存器的机器非常有效。通过该选项生成的代码非常难于调试。选项-O3 将设置该选项，但可以通过选项-fno-rename-registers 关闭该选项。

### -frepo

C++

打开自动模板实例化。设置该选项将同时设置选项-fno-implicit-templates，它将抑制非内嵌模板的自动实例化。

### -frerun-cse-after-loop

该选项将在循环优化后再次进行公用子表达式优化。这是因为在循环优化后可能会产生新的子表达式。选项-O2，-O3 和-Os 设置该选项，但可以通过-fno-rerun-cse-after-loop 关闭该选项。参见选项--param。

### -frerun-loop-opt

运行循环优化两次。第二次优化并不展开循环，但它会再次利用第一次优化中删除的指令对循环进行分析。O2，-O3 和-Os 设置该选项，但可以通过-fno-rerun-loop-opt 关闭该选项。

### -frtti

C++

这是默认选项。运行时标识代码是为包含虚方法的每个类生成的。如果没有使用 dynamic_cast 和 typeid，可以使用选项-fno-rtti 抑制生成代码，从而节省空间。该选项对异常处理无效，将按需生成 rtti 代码。

### -fschedule-insns

相对其他操作而言，在浮点运算速度较慢或内存访问操作较慢的机器上，以及支持同一时间运行多条指令的机器上，可以改变指令的序列来消除处理器的阻塞。在较慢的指令正在执行的同时，执行其他指令。选项-O2，-O3 和-Os 将设置该选项，但可以使用-fno-schedule-insns 关闭该选项。

### -fschedule-insns2

该选项与-fschedule-insns 一样，只是它是在为每个函数分配全局寄存器和局部寄存器之后执行的。对于寄存器数目较少而且装载寄存器的指令较慢的机器，该选项非常有效。选项-O2，-O3 和-Os 将设置该选项，但可以使用-fno-schedule-insns2 关闭该选项。

### -fshared-data

该选项要求数据共享而非私有。该选项只对某些操作系统有意义，在这些操作系统中，

可以在运行同一程序的独立进程间访问共享数据，而且每个进程都有自己的私有数据。

-fshort-double

double 数据类型使用与 float 数据类型相同的尺寸。
对 Fortran 而言，该选项可能会引起问题。

-fshort-enums

将某个 enum 的尺寸减少到保存所需数值范围的最小整数类型。

-fshort-wchar

C C++

将数据类型 wchar_t 转换为 unsigned short int，而非当前目标机的默认类型。

-fsigned-char

C

使用该选项后，数据类型 char 默认是有符号的（从 0 到 255 的数值）。如果没有指定 char 数据类型是否有符号，默认情况将取决于平台。使用选项-fno-signed-char 与选项-funsigned-char 相同。

-fsigned-bitfields

C

这是默认选项。位字段按有符号 int 数据类型处理，但选项-fno-signed-bitfields 将按无符号 int 数据类型处理位字段。使用选项-traditional 将强制按无符号处理所有的位字段。使用选项-fno-signed-bitfields 与选项-funsigned-bitfields 相同。

-fsilent

Fortran

这是默认选项。使用选项-fno-silent 将在 stderr 上输出所有正在编译的程序单元的名字。

-fsingle-precision-constant

按常量声明的浮点数值将按单精度存储，而非双精度。

-fsource-case-spec

Fortran

spec 确定源代码文本是否被翻译为全部大写、全部小写或是不变。Holerith 常量不受该选项影响。spec 可为如下形式之一：

- upper 源代码全部翻译为大写。
- lower 源代码全部翻译为小写。这是默认选项。
- preserve 源代码文本不变。

参见选项-fintrin-case-，-fmatch-case-，-fsymbol-case-和-fcase-。

-fssa

试验功能。整个函数体将转换为一个 SSA（Static Single Assignment，静态单赋值）流图，之后在其上进行优化操作，继而再将代码由 SSA 转换为原格式。

-fssa-ccp

试验功能。打开 SSA 条件代码传播，其中实际上被当做常量来使用的变量将被转换为常量，并且会消除从未选取的分支。该选项需要选项-fssa 和任意级-O 优化。

-fssa-dce

试验功能。打开 SSA 不可达代码消除功能，将不可能被执行的代码删除。该选项需要选项-fssa 和任意级-O 优化。

-fstack-check

生成的代码会为防止程序栈溢出进行必要的检测。所生成的代码实际上并不进行检测，它只保证操作系统可以检测到栈扩展。

可能有必要在多线程环境中运行一个程序，因为在单线程程序中，栈溢出是自动检测的。

-fstack-limit-register=register

指定包含限制栈尺寸地址的寄存器名称。该选项只能用来减少栈；不能用它来扩展超出操作系统指定的栈尺寸。

参见选项-fstack-limit-symbol。

-fstack-limit-symbol=symbol

指定包含限制栈尺寸地址的变量名。该选项只能用来减少栈；不能用它来扩展超出操作系统指定的栈尺寸。

地址的值取决于平台。例如，如果栈从地址 0x8000000 开始并且通过降低地址完成增长栈，128k 的地址限额可以通过如下选项设置：

```
-fstack-limit-symbol=__stack_limit -Wl,__stack_limit=0x7FFE0000
```

还可以在程序内声明地址限额变量，但依然需要在命令行中使用选项-fstack-limit-symbol。

参见选项-fstack-limit-register。

### -fstats

C++

显示前端处理的统计信息。这些信息主要用于编译程序内部，对其输出没有影响。

### -fstore-check

Java

这是默认选项。指定选项-fno-store-check 将删除运行时检查，可保证在数组中保存正确的对象类型。

### -fstrength-reduce

执行循环强度消除并消除在循环内部使用的变量。这是用简单而快速的操作（如加法和减法）替代耗时操作（如乘法和除法）的过程。选项-funroll-loops 和-funroll-all-loops 均打开该选项。选项-O2，-O3 和-Os 也打开该选项，但可以通过-fno-strength-reduce 关闭该选项。

作为一个简单的例子，下面的循环使用一个临时变量作为索引：

```
for(int i=0; i<10; i++) {
    index = i * 2;
    frammis(valarr[index]);
}
```

内部变量 index 可被消除，乘法也可以被替代为简单的移位操作，如下所示：

```
for(int i=0; i<10; i++) {
    frammis(valarr[i << 1]);
}
```

向左移循环计数器一位就是将其加倍，之后就可以直接将结构作为数组的索引，而无需使用一个临时变量保存。

### -fstrict-aliasing

根据正在编译的语言，将使用最严格的别名规则。例如，在 C 中一个 int 不能成为一个 double 或指针的别名，但可以成为一个 unsigned int 的别名。即使使用了最严格的别名规则，对联合的成员的访问也没有问题，只要是通过联合而不是通过联合成员的地址进行访问。下面的一段代码将引起问题：

```
int *iptr;
union {
    int ivalue;
    double dvalue;
} migs;
 . . .
    migs.ivalue = 45;
```

```
iptr = &migs.ivalue;
frammis(*iptr);
migs.dvalue = 88.6;
frammis(*iptr);
```

在这个例子中，严格的别名规则可能无法识别 iptr 所指引的数值在两次函数调用间的变化。但直接通过联合成员引用就不会引起问题。

### -fsymbol-case-spec

**Fortran**

spec 决定用户自定义符号的大小写。可以是如下形式之一：

- initcap　名字的首字母大写，其余小写。
- upper　名字均为大写。
- lower　名字均为小写。
- any　名字可以是大小写的任意组合。这是默认选项。

参见选项-fmatch-case-，-fsource-case-，-fintrin-case-和-fcase-。

### -fsyntax

**Fortran**

对源代码进行语法检查，之后不进行任何处理。

### -ftemplate-depth-number

**C++**

将最大模板实例化深度设置为 number 值，以探测递归或循环模板定义。符合标准的程序不能使用深度超过 17 的模板。默认深度是 500。

### -ftest-coverage

编译程序将在所生成的文件中包含 gcov 使用的信息。输出文件与源文件具有相同的名字，但具有不同的后缀以示内容上的区别。

后缀为.bb 的文件包含了由基本执行代码块到源文件行号的映射。这些信息可供 gcov 将执行计数与源代码行号建立联系。

后缀为.bbg 的文件包含了程序流图中的所有箭头。这些信息供 gcov 使用来重建流图并从选项-fprofile-acrs 指定的以.da 为后缀的文件数据中计算执行计数。

参见选项-a，-ax，-ftest-coverage 和-fprofile-arcs。

### -fthread-jumps

如果某跳转条件表达式的值将代码转移到某处，而该值在此处恰巧又是一个跳转，那么源跳转将被直接转移到最终目的地。所有的优化级均设置该选项，但可以使用选项-fno-thread-jumps 将其关闭。

### -ftime-report

编译完成时，编译程序会打印编译耗时的统计信息。每一次扫描所涉及的用户时间、系统时间以及实际时间都将被打印出来，并在最后给出一个概要信息。

### -ftrapping-math

这是默认选项。设置选项-fno-trapping-math 将使代码假设浮点操作不会引起异常，这些异常可能被捕获并产生信号。选项-fno-trapping-math 可能会生成违反标准浮点操作规则的代码。

### -ftrapv

生成捕获有符号加、减和乘法等溢出条件的代码。通常这些溢出是被忽略的，但该选项可以在软件测试中使用，并在有整数溢出时产生 core 文件。默认选项是-fno-trapv。

### -ftypeless-boz

**Fortran**

指出带前缀基数的非十进制常量，如 Z'ABCD'，是无类型的而不是默认的 INTEGER(KIND=1)。

### -fugly-args

**Fortran**

这是默认选项。指定选项-fno-ugly-args 将不允许无类型 Hollerith 常量作为参数在函数调用中传递。例如，下面两个函数调用默认是合法的：

```
CALL FRED(4HABCX)
CALL SAM('123'O)
```

### -fugly-assign

**Fortran**

使用同样的存储包含被赋值的标记和数值数据。例如，下面两条语句将使用同一存储位置：

```
I = 3
ASSIGN 10 TO I
```

如果程序想按数据访问被赋值的数值，就需要使用该选项，因为默认情况下编译程序会为不同类型的信息单独创建存储位置。

### -fugly-assumed

**Fortran**

尺寸为 1 的数组将按声明尺寸为*的数组使用。例如，DIMENSION X(1)被看做是以 DIMENSION X(*)形式声明的。

### -fugly-comma

**Fortran**

结尾的逗号隐含地将空参数传递给子例程。例如，使用该选项时，CALL BLOG()会传递一个空参数，而 CALL RIM(,)将传递两个空参数。

如果不使用该选项，结尾的逗号将被忽略，从而不会传递空参数，即使是列表中有其他参数也不会传递。

### -fugly-complex

**Fortran**

允许在特质 REAL(expr)和 AIMAG(expr)中使用任何复数表达式。默认情况下会限制复数表达式为 COMPLEX(KIND=1)。

使用选项-ff99 时，这些特质将返回参数中未被转换的实部和虚部。

### -fugly-init

**Fortran**

这是默认选项。指定选项-fno-ugly-init 将不允许在 PARAMETER 和 DATA 语句中使用 Hollerith 格式的数据。同时不允许使用字符常量来初始化数值数据类型，反之亦然。

### -fugly-logint

**Fortran**

大多数情况下，系统会自动在数据类型 INTEGER 和 LOGICAL 间进行转换，基本上这两种类型可以互换使用。

### -funderscoring

**Fortran**

这是默认选项。在有一个下划线的名字后追加两个下划线。在没有下划线的外部名字后追加一个下划线。为防止与外部名字产生冲突，对具有下划线的内部名字同样追加两个下划线。

指定选项-fno-underscoring 将禁止通过追加下划线对名字进行转换。指定选项-fno-second-underscoring 只禁止追加第二个下划线。

除非试图控制输出以产生与其他编译程序兼容的代码，否则不建议使用这些控制下划线用法的选项。另外，控制下划线可能会产生与系统库的冲突。

### -fuse-boehm-gc

**Java**

打开 Boehm 垃圾收集器位图标记代码的使用。

### -fsecond-underscore

**Fortran**

这是默认选项。参见-funderscoring。

### -funix-intrinsics-spec

**Fortran**

spec 指定了 UNIX 特质的状态。它可以是下列形式之一：

- enable 可以识别并打开 UNIX 特质。这是默认选项。
- hide 只有在特质中的名字首次出现在 INTRINSIC 语句中时，才识别和打开 UNIX 特质。
- disable 可识别 UNIX 特质，但只有通过 INTRINSIC 语句方可引用它们。
- delete 不可识别 UNIX 特质。

### -funroll-all-loops

该选项设置-funroll-loops 并去除循环展开的尺寸限制，即使当循环的次数无法确定时仍然展开循环。该选项通常会产生比不使用该选项更慢和更大的代码。

当要展开一个循环次数无法确定的循环时，循环被展开为一系列代码块，每个代码块的结尾都有循环测试。这将生成一个包含自身多个副本的更大循环，因此不会像其他情况下那样经常迭代。

### -funroll-loops

如果在编译时可以确定迭代的次数非常小，而且循环中的指令也非常少，可以进行循环展开，通过去除循环和复制指令，从而保证正确的执行次数。如果 insns 的数值乘以迭代的次数小于一个常量（当前设置为 100），就认为循环足够小。该选项同时设置选项-fstrength-reduce 和-frerun-cse-after-loop。

### -funsafe-math-optimizations

去除浮点运算的检查并假设所有的数值均是合法的。这将允许违反 IEEE 和 ANSI 标准的数学操作。该选项还可能造成链接器包含以非标准方式优化硬件 FPU（浮点运算单元）操作的代码。

### -funsigned-bitfields

**C**

使用该选项将按照 unsigned int 数据类型处理位字段。默认情况下，将按 signed int 数据类型处理位字段。使用选项-traditional 也会将所有的位字段按照无符号类型处理。指定选项-fno-unsigned-bitfields 与选项-fsigned-bitfields 相同。

### -funsigned-char

**C**

使用该选项后，默认情况下 char 数据类型为无符号类型（值域为-127 到+128）。如果

没有使用该选项，默认情况下 char 数据类型是否有符号取决于所使用的平台。指定选项-fno-unsigned-char 与选项-fsigned-char 相同。

### -funwind-tables

除了生成必要的静态数据外，该选项与选项-fexceptions 相似，对所生成的代码没有其他影响。该选项只供内部使用，不应该在命令行中使用。

参见选项-fexceptions，-fnon-call-exceptions 和-fasynchoronous-unwind-tables。

### -fuse-cxa-atexit

C++

造成全局析构函数按照构造函数完成的相反顺序执行。仅当某构造函数内部调用了另外一个构造函数时，顺序才会改变。该选项仅在函数 cxa_exit()是 C 运行时库的一部分时工作。如果不使用该选项，可以使用函数 atexit()。

### -fuse-divide-subroutine

Java

调用库例程完成整数除法，从而可以在整数除数为 0 时抛出异常。

### -fverbose-asm

在所生成的汇编语言中插入比正常情况下更多的注释，从而使其更易读。

该选项主要用来使汇编代码更易读，并用来调试编译程序本身。默认情况下，相比汇编列表，选项-fno-verbose-asm 更有用处。

### -fversion

Fortran

运行内部测试来验证 GNU Fortran 是否正确安装，并显示版本号。选项-v 和--verbose 均设置了该选项。

### -fvolatile

将所有通过指针引用的内存视为易变的（volatile）。

参见选项-fvolatile-global 和-fvolatile-static。

### -fvolatile-global

将所有外部和全局内存引用视为包含易变数据。该开关不会将静态数据项（仅从编译单元访问的数据）视为易变的。

参见选项-fvolatile 和-fvolatile-static。

### -fvolatile-static

将所有引用静态数据项（仅从编译单元访问的数据）的内存引用视为易变的。

参见选项-fvolatile-global 和-fvolatile。

-fvtable-gc **C++**

导致重定位信息的创建，从而使连接程序可以删除没有使用的虚函数 vtable 表项。该选项需要同时使用 GNU 汇编器和链接器。

该信息也可以用来删除未使用的函数。参见选项-ffunction-sections 和-W1。

-fvxt **Fortran**

某些源代码构造函数在 GNU Fortran 和 VXT Fortran 中具有不同的含义。使用该选项将造成所有构造函数按 VXT Fortran 解释。

参见选项-ff90，-ffvxt-intrinsics 和-ff90-intrinsics-。

-fvxt-intrinsics-spec **Fortran**

spec 指定了 VXT 特质的状态。它可以是如下形式之一：

- enable 可以识别并打开 VXT 特质。这是默认选项。
- hide 仅当特质名字首次出现在 INTRINSIC 语句中时才识别并打开 VXT 特质。
- disable 可以识别 VXT 特质，但必须通过 INTRINSIC 语句方可进行引用。
- delete 不可识别 VXT 特质。

参见选项-fvxt。

-fweak **C++**

这是默认选项。指定选项-fno-weak 将禁止使用弱符号支持，即使链接器支持也不能使用。由于选项-fno-weak 会产生较低级的代码，除了测试编译程序外没有其他用处，不应该使用选项-fno-weak。

-fwritable-strings **C C++**

编译程序允许将数据写入字符串常量。选项-traditional 将设置该选项。

为了能够真正写入 C++的字符串常量，同时需要设置选项-fno-const-strings。

-fzeros **Fortran**

像其他值一样处理初始值 0。如果没有这一选项，可以有多个 DATA 语句设置变量的初始值为 0，而编译程序无法诊断这一情况。

-g[level]

输出的调试信息格式可被 gdb 使用。其格式和内容依赖于编译程序所产生的目标文件格式（stabs，COFF，XCOFF 或 DWARF）。

level 设置是可选的。level 数值指定了所包含的调试信息量。默认级别是 2。级别 1 产生的全局信息可供反向跟踪，但不包括局部变量和行号。级别 2 包括所有级别 1 的内容，同时还有局部变量和行号信息。级别 3 包括级别 2 的信息以及宏定义等额外信息。

在使用 stabs 格式的系统中，该选项将按 gdb 可用的格式生成调试信息。

使用该选项的同时可用选项-O 生成优化代码。当优化打开时，调试过程可能不像正常情况下那样容易跟踪，这是因为优化在生成代码时做了许多改动，源代码和所生成的目标文件间不存在一对一的对应关系。某些目标文件可以重定位，而某些源码可能根本不生成任何代码。

参见选项-ggdb，-gstabs，-gcoff，-gxcoff 和-gdwarf。该选项可以写作--debug。

-gcoff[level]

如果支持的话，按照 COFF 格式生成调试信息。在 SVR4 以前的 System V 的 SDB 中，该格式使用比较广泛。设置 level 是可选的。参见选项-g 中关于 level 设置 1，2 和 3 的说明。

-gdwarf[level]

如果支持的话，按照 DWARF 格式生成调试信息。设置 level 是可选的。只有将 level 设置为+才会包含 gdb 扩展，它可能令其他调试器无法使用。参见选项-g 中关于 level 设置 1，2 和 3 的说明。

这是大部分 SVR4 系统中的 SDB 使用的格式。

-gdwarf-2[level]

如果支持的话，按照 DWARF 版本 2 的格式生成调试信息。设置 level 是可选的。参见选项-g 中关于 level 设置 1，2 和 3 的说明。

这是 IRIX 6 的 DBX 中的格式。

-gen-decls

将源文件中类的接口声明写入名为 w.decl 的文件中。

-ggdb[level]

生成适用于 gdb 格式的详细调试信息，包含任意可能的 gdb 扩展。设置 level 是可选的。参见选项-g 中关于 level 设置 1，2 和 3 的说明。

### -gnatoption

Ada

指定为 Ada 前端 GNAT 提供的选项。所有选项都是单字母，跟在选项字 gnat 之后。例如，如果要指定选项 e 和 l，可以在命令行中这样指定它们：

```
$ gcc -gnate -gnatl
```

还可以将两个选项组合成一个。下面的形式与前面的相同：

```
$ gcc -gnatel
```

某些选项需要提供数值。例如，选项 k 用来指定由编译程序输出的错误消息的最大数目，下面的命令将该值设置为 15：

```
$ gcc -gnatk15
```

需要数值的选项可以与其他选项组合在一起，但需要放在列表的最后。下面的例子给出选项 e，l 和 k 的组合格式：

```
$ gcc -gnatelk15
```

下面的选项指定了数字但不带前缀字母，强制使用 Ada 83 或 Ada 95 定义的限制：

```
$ gcc -gnat83
$ gcc -gnat95
```

默认是选项-gnat95。表 D-3 包含了选项-gnat 前缀可用的选项字母列表。

**表 D-3 用于-gnat 选项的字母和值**

| 字母 | 描述 |
|---|---|
| a | 打开断言。该选项同时打开 Pragma Assert 和 Pragma Debug。如果不使用该选项，这两个 pragma 将被忽略 |
| b | 即使同时设置了 verbose 选项，在标准错误输出产生简要消息 |
| c | 只检查语法和语义。生成.ali 文件，但不生成可执行代码 |
| e | 当错误发生时显示错误消息，而不是在结束时保留和打印所有的错误 |
| E | 执行完全的动态检查 |
| f | 报告每一种可能的错误。每一行均可能检测出多个错误。报告所有未定义的引用，而不仅仅是第一个引用 |
| g | 打开 Ada 风格的检查（列对齐、缩进、大小写格式等） |
| ichar | char 的值指定了标识符的字符集。标准的 ASCII 字符（值域由 1 到 127）总是相同的，但 8 位数值中的其他部分具有不同的含义。其所标识的字符集如下：<br>1 拉丁-1 |

（续表）

| 字母 | 描述 |
|---|---|
| | 2 拉丁-2 |
| | 3 拉丁-3 |
| | 4 拉丁-4 |
| | p IBM PC |
| | f 全大写 |
| | n 无大写 |
| | w 宽字符 |
| jchar | char 的值指定了用来编码宽字符的方法。可用的方法如下： |
| | n 没有编码 |
| | h Hex 编码 |
| | u 上半区编码 |
| | s Shift JIS 编码 |
| | e EUC 编码 |
| knumber | number 的值指定了一个标识符中字母个数的上限（1~999） |
| l | 输出源代码列表，包括错误消息 |
| mnumber | number 的值是可以包括的最大错误数目 |
| n | 激活跨子程序单元边界的内嵌机制，这些子程序均指定了 inline pragma。如果使用了选项-fno-inline，该选项将被关闭 |
| N | 与使用选项-gnatn 并为每行源代码加入 inline pragma 相同。可以使用-fno-inline 关闭该选项 |
| o | 打开通常关闭的运行时检查，如整数溢出检查和描述前访问。该选项会造成程序更大且运行更慢。（该选项对浮点操作没有影响） |
| p | 禁止运行时检查。这与 pragma Suppress(all_checks)命令相同。通过去除运行时安全检查，程序可以更小而且运行更有效 |
| q | 在发现语法错误后不停止编译程序。之后将根据解析器所收集的信息继续代码生成 |
| r | 要求列格式匹配参考手册的要求 |
| s | 仅检查语法 |
| t | 生成用于不可达代码消除的.adt 树文件 |
| u | 列出本次编译所涉及的所有单元 |
| v | 详细模式。在标准错误输出中给出完全的错误报告以及引起错误的源代码行号 |
| wmode | 该模式决定如何处理警告。如果 mode 是 s，将禁止警告消息。如果 mode 是 e，警告消息将被处理为错误消息。如果 mode 是 l，将产生描述次序警告 |
| ztype | 指定生成存根。如果 type 是 r，生成接收端存根。如果 type 是 s，将生成发送端存根 |

-gstabs[level]

如果支持的话，按照 stabs 格式生成调试信息。设置 level 是可选的。只有将 level 设置为+才会包含 gdb 扩展。参见选项-g 中关于 level 设置 1，2 和 3 的说明。

该选项可以用在大多数 BSD 系统的 DBX 中，但无法在 MIPS，Alpha 或 SVR4 上的 DBX 或 SDB 上使用。而且在 SVR4 系统上需要使用 GNU 汇编器。

-gvms[level]

如果支持的话，按照 VMS 调试格式生成调试信息。设置 level 是可选的。参见选项-g 中对 level 设置 1，2 和 3 的说明。

这是 VMS 系统中 DEBUG 所使用的格式。

-gxcoff[level]

如果支持的话，按照 XCOFF 格式生成调试信息。设置 level 是可选的。只有将 level 设置为+才会包含 gdb 扩展，它可能令其他调试器无法使用，而且与非 GNU 汇编器一起使用时还会出现问题。参见选项-g 中关于 level 设置 1，2 和 3 的说明。

这是 RS/6000 系统的 DBX 所使用的格式。

-H

Pre

打印使用的所有头文件的嵌套列表，以及那些没有多次包含保护的头文件。

该选项也可以写作--trace-includes。

--help

显示 gcc 理解的命令选项列表。如果和选项-v 一起使用，列表还将包含 gcc 调用的各个进程所接受的选项。如果与选项-W 一同使用，同时会列出未给出文档的命令行选项。参见选项--target-help。

--include-directory

Pre Ada Java

与选项-I 相同。

-I name

Pre Ada Java

对预处理程序而言，name 是包含文件搜索目标主列表的目录。该选项可以重复使用，从而加入若干个目录。

该选项所指定的目录将被首先搜索，因此可用该选项重载通常所包含的系统头文件。

可以用-I.中的点引用当前目录，它向编译程序指定当前工作目录。

对 Ada 而言，name 是搜索源文件的目录。

对 Java 而言，name 是在搜索其他路径（如选项--classpath 中命名的或在环境变量 CLASSPATH 中命名的）之前搜索的路径单元。该单元可以是目录、jar 文件或 zip 文件。建议使用-I 而非--classpath。

参见选项-I-，-isystem，-B，-nostdinc 和-withprefixbefore 以及环境变量 CPATH。如果要设置二级头文件搜索目录列表，参见选项-idirafter。该选项也可以写作--include-directory。

-I-

Pre

-I-特殊形式可被指定一次，用来声明命令行中其前所有-I 指示字对#include "..."生效，但不对#include <...>生效。任何跟在选项-I-后的-I 选项都将对#include <...>生效。

选项-I-也将忽略对源文件所在目录的搜索。

该选项也可以写作--include-barrier。

-idirafter directory

Pre

将 directory 加到二级头文件搜索目录列表中。要查找一个头文件，GCC 将在一级列表（通过选项-I 加入的目录）中的目录中搜索。如果在一级列表中没有找到头文件，就继续搜索二级列表。

该选项也可以写作--include-directory-after。

-imacros filename

Pre

预处理程序将在读取程序源文件前读取和处理所指定的文件。除了宏之外，存储在 filename 中的信息将被丢弃，只有宏会用于源文件。

命令行中的选项-D 或-U 将在选项-imacros 之前处理。选项-include 和-imacros 将按其在命令行中出现的次序处理。

该选项也可以写作--imacros。

-include filename

Pre

预处理程序将在读取程序源文件前读取和处理所指定的文件，就如同其在源文件的第一行被包含了一样。

命令行中的选项-D 或-U 将在选项-include 之前处理。选项-include 和-imacros 将按其在命令行中出现的次序处理。

该选项也可以写作--include。

--include-barrier

Pre

与选项-I-相同。

--include-directory-after directory Pre

与选项-idirafter 相同。

--include-prefix prefix Pre

与选项-iprefix 相同。

--include-with-prefix directory Pre

与选项-iwithprefix 相同。

--include-with-prefix-after directory Pre

与选项-iwithprefix 相同。

--include-with-prefix-before directory Pre

与选项-withprefixbefore 相同。

-iprefix prefix Pre

指定用来构成选项-iwithprefix 和-withprefixbefore 中给出的目录路径的前缀。

该选项也可以写作--include-prefix。

-isystem directory Pre

在二级包含路径的起始处增加名为 directory 的目录，并将目录标记为系统目录，并按标准系统目录的使用方式处理。

参见选项-I 和-B。

-iwithprefix directory Pre

在用于搜索头文件的二级目录列表中增加名为 directory 的目录。通过选项-iprefix 中指定的前缀与名为 directory 的目录组合构成最终的路径名。如果在该选项之前命令行中没有指定前缀，默认使用包含安装编译程序的目录。

要搜索某个头文件，GCC 在（通过选项-I 所增加的）一级列表中搜索目录。如果在一级列表中没有找到，就搜索二级列表。

该选项也可以写作--include-with-prefix 或--include-with-prefix-after。

-iwithprefixbefore directory Pre

在主包含路径的目录中增加一个名字。通过选项-iprefix 所指定的前缀和名为 directory 的目录的组合构成名字。如果在该选项之前命令行中没有指定前缀，默认使用包含安装编译程序的目录。

该选项也可以写作--include-with-prefix-before。

--library-directory directory **Linker**

与选项-L 相同。

-Ldirectory **Linker**

将指定的目录加入选项-l 搜索的库目录中。

参见选项-B 和环境变量 LIBRARY_PATH。该选项也可以写作--library-directory。

-llibrary **Linker**

设置链接器解析引用的库名。实际的库名是通过在指定的名字中增加前缀 lib 和后缀.a 构成的。例如，设置-lconsole 后，实际的库名是 libconsole.a。

对所指定的库文件的搜索将在标准库文件集以及选项-L 所给出的目录中进行。

选项-lobjc 是链接 Objective-C 程序的特殊情况。

要解析引用，按照其在命令行中出现的顺序搜索库，这意味着它们出现的顺序是至关重要的。例如，在下面的命令行中，任何 glower.o 对 libjpeg.a 中的目标文件的引用都可以被解析，但 flower.o 中的引用就无法解析。

```
gcc glower.o -ljpeg flower.o -o showall
```

当某个库中的成员需要另外一个库中的成员才能解析时，该顺序也非常重要。有可能在两个库间形成循环引用，这需要它们在命令行中出现两次：

```
gcc sprig.o -ldflat -lturbo -ldflat -o sprig
```

可以在命令行中按 libjpeg.a 和-ljpeg 方式引用同一个库，但只有选项-l 才会指示链接器在标准目录以及选项-L 所指定的目录中搜索。

为了符合 POSIX 标准，在选项标志和库名之间可以留一个空格。

参见选项-L。

--language language

产生的规则包含目标文件名及其依赖文件列表；但不包含编译源文件的规则。

如果没有使用选项-MT 或-MQ 指定名字，在所产生的规则中的目标文件名与输入的源

该选项也可以写作--print-missing-file-dependencies。

**-MM** Pre

除了不列出系统头文件外，该选项与选项-M 相同。
该选项也可以写作--user-dependencies。

**-MP** Pre

与选项-M 或-MM 一同使用，为每个包含文件生成哑目标。其惟一目的就是防止 make 在删除头文件而又没有更新 makefile 文件时产生错误消息。

**-MQ filename** Pre

除了要用适合 makefile 的方式引用目标名外，该选项与选项-MT 相同。例如，命令 gcc -M -MT '$(OBJMRK)mrk.o' brink.c 将产生如下内容：

```
$$(OBJMRK)mrk.o: brink.c
```

**-MT filename** Pre

与选项-M 或-MM 一同使用，指定所产生的 makefile 规则的目标文件名。默认情况下，目标文件与输入的源文件名相同，但使用后缀.o。选项-MT 可以用来指定一个不同的名字、一个完整的路径或是基于环境变量的名字。例如，命令 gcc -M -MT '$(OBJMRK)mrk.o' brink.c 将产生如下内容：

```
$(OBJMRK)mrk.o: brink.c
```

参见选项-MQ。

**--no-line-commands** Pre

与选项-P 相同。

**--no-standard-includes** Pre Ada

与选项-nostdinc 相同。

**--no-standard-libraries** Linker

与选项-nostdlib 相同。

**--no-warnings**

不产生警告消息。与选项-w 相同。

-nodefaultlibs

Linker

标准系统库例程不应该作为被链接的可执行程序的一部分包含进来。用到的库仅仅是在命令行中所列出的那些库。

编译程序可能会产生对系统函数 memcpy()，memcmp()，memset()（对标准 C 和 System V 而言），bcopy()和 bzero()（对 BSD 而言）的调用。这些引用通常都可在 libc.a 中进行解析，因此需要在命令行中亲自提供这些例程。

库 libgcc.a 包含了一系列目标平台相关的特殊例程，可以将它们视为编译程序的一部分，因此在没有标准库时应该指定-lgcc。

参见选项-nostartfiles 和-nostdlib。

-nostartfiles

Linker

标准启动目标文件不应该作为链接的可执行程序的一部分被包含。参见选项-nostdlib 和-nodefaultlibs。

-nostdinc

Pre Ada

禁止编译程序在标准系统目录中搜索头文件。搜索的仅仅是当前目录和选项-I 中指定的目录。

对 Ada 来说，该选项指定了源文件不使用系统库。

该选项也可以写作--no-standard-includes。

-nostdinc++

Pre C++

禁止编译程序在标准 C++目录中搜索头文件，但可以在其他标准目录中搜索。该选项专供编译 C++库使用。

-nostdlib

Linker

只有命令行指定的项才传递给链接器。标准启动文件和库都不会传递给链接器。该选项隐式打开选项-nostartfiles 和-nodefaultlibs。

该选项也可以写作--no-standard-libraries。

-Olevel

指定用于编译程序所生成的代码的优化级别。注意，在代码尺寸和执行速度间永远都需要权衡。默认选项是-O0，即不进行优化。

如果没有指定优化级别，编译程序产生与输入源结构匹配的代码。优化不仅需要更多的处理，还需要更多的内存。没有优化的编译具有两个好处，可以缩短编译时间（优化则需要很长的时间），而且可以在调试器中轻松跟踪所产生的代码。对软件开发过程而言，所

有这些都是理想的条件。也可以使用调试器调试已优化的代码，但由于输出的代码可能被重组，因此很难进行跟踪。

该选项也可以写作--optimize。

优化级别参见表 D-4。

表 D-4 优化的 6 种级别

| 级别 | 描述 |
|---|---|
| -O | 编译程序尝试减少代码尺寸和执行时间，但不会进行造成调试困难的修改。该级别会打开选项-fno_optimize_size，-fdefer_pop，-fthread_jumps，-jguess_branch_prob，-cprop-registers 和-fdelayed_branch。当调试器在某平台上无需栈指针就可以正常工作时才会设置选项-fomit-frame-pointer |
| -O0 | 这是默认选项。关闭所有优化。关闭所有尺寸优化并设置选项-fno-merge-constants |
| -O1 | 与选项-O 相同 |
| -O2 | 该优化级打开不涉及代码尺寸和运行速度权衡的所有优化。除了选项-O 所打开的选项外，该优化级还打开选项-foptimize-sibling-calls，-fcse-follow-jumps，-fcse-skip-blocks，-fgcse，-fexpensive-optimizations，-fstrength-reduce，-frerun-cse-after-loop，-frerun-loop-opt，-fcaller-saves，-fforce-mem，-fpeephole2，-fschedule-insns，-fschedule-insn-after-reload，-fregmove，-fstruct-aliasing，-fdelete-null-pointer-checks 和-freorder-blocks。该优化级不进行循环展开、函数内嵌或是寄存器重命名 |
| -O3 | 除了-O2 所打开的选项外，该优化级还打开选项-finline-functions 和-frename-registers |
| -Os | 优化代码尺寸。打开-O2 中打开的所有选项。将选项-falign-loops，-falign-jumps，-falign-labels 和-falign-functions 设置为 1，它将禁止任何为对齐而插入的空间 |

### -o filename

将输出写入指定的文件中。该选项对各种输出均有效，可以是预处理过的源代码、汇编语言、目标文件或是已链接的可执行程序。由于只能指定一个输出文件，因此如果可能产生多个文件，就不要使用-o 选项。

如果编译程序生成一个要链接的可执行程序，而且没有用-o 选项指定其名字，默认名是 a.out。

该选项也可以写作--output。

### --optimize level

与选项-O 相同。

--output filename

与选项-o 相同。

--output-class-directory=directory

Java

与选项-foutput-class-dir 相同。

-p

包含额外的代码，这些代码将输出适合性能分析程序 prof 分析的信息。该选项必须在编译源文件和链接目标文件的同时使用。

参见选项-pg。该选项也可以写作--profile。

-P

Pre

当同选项-E 一起使用时，预处理程序不产生#line 指示字。

该选项也可以写作--no-line-commands。

--param name=value

GCC 内部存在一些优化代码程度的限制，因此调整这些值就是调整整个优化。表 D-5 给出了参数的名字和可用的数值。

该选项也可以写作--param。

表 D-5 --params 选项可接受的名字

| 名字 | 值 |
|---|---|
| max-delay-slot-insn-search | 填充一个延迟段时所选择的最大数目的指令。较大的数值可以改进所生成的代码，但会降低编译速度。默认值是 100 |
| max-delay-slot-live-search | 当查找具有合法寄存器信息的代码块时的最大代码块搜索数目。较大的数值可以改进生成的代码，但会降低编译速度。默认值是 333 |
| max-gcse-memory | 可用来分配执行 GCSE（全局命令子表达式消除）的最大的内存量。如果没有足够的内存执行该操作，优化将无法进行。默认值是 50MB（52428800） |
| max-gcse-passes | GCSE（全局命令子表达式消除）的最大迭代次数。默认值是 1 |
| max-inline-insns | 在可以被扩展内嵌的方法中的最大指令数目。默认值是 600 |
| max-pending-list-length | 在跟踪机制重置槽位列表并重新开始前，可以由槽位调度器在一个等待依赖列表中排序的最大的分支元素数目。一个大函数可能产生成千上万的依赖关系。默认值是 32 |

### -pass-exit-codes

忽略编译过程中所有阶段标志错误的退出代码，并且编译程序将继续运行。GCC 的返回值是任何阶段中最大的错误代码值。通常，编译程序在返回值为非零值时停止并退出。

### -pedantic

C C++ Fortran

严格地遵照 ISO C 和 C++标准产生所需的警告消息。如果没有该选项，GNU 扩展将被打开，但符合 ISO 标准的程序可以成功地编译（尽管某些可能需要使用选项-ansi）。

对 C 而言，所遵守的是选项-std 所指定的标准。如果-std 指定了 gnu89，选项-pedantic 使用 C89 的规则。选项-pedantic 只输出 ISO 标准所需要的诊断信息，因此很可能不完全符合标准的程序能够成功地编译，而不产生警告消息。GCC 尚无计划提供强制遵从标准的选项。

对 C 而言，选项-pedantic 对跟在__extension__后的表达式没有影响。

对 C++而言，如果既没有指定选项-fpermissive 也没有指定选项-pedantic，默认使用选项-fpedantic-errors。

对 Fortran 而言，仅在 Fortran 77 中使用扩展时给出警告消息。对在字符串常量中像 C 那样的构建字（如\n）会给出警告。对某些 GNU 语言扩展和某些传统 Fortran 功能会给出警告。但是，要注意的是，该选项并不强制要求程序严格地遵从标准。

该选项也可以写作--pedantic。

### -pedantic-errors

C C++ Fortran

除了诊断消息被当作错误（而不是警告）输出外，该选项与-pedantic 相同。

对 C++而言，如果既没有使用选项-fpermissive 也没有使用选项-pedantic，默认使用选项-fpedantic-errors。

该选项也可以写作--pedantic-errors。

### -pg

包含额外的代码，这些代码将输出性能分析程序 gprof 所需的信息。该选项必须在编译源文件和链接目标文件的同时使用。参见选项-p。

### -pipe

使用管道（而不是临时文件）作为编译程序一个阶段到另一个阶段交换输出的方式。如果本地汇编器不能从管道中读取输入，该选项将失效。

该选项也可以写作--pipe。

--prefix prefix

与选项-B 相同。

--preprocess

Pre

与选项-E 相同。

-print-file-name=library

输出指定的库的全路径名。除此之外编译程序不执行其他动作。参见选项-print-ligcc-file-name 和-print-prog-name。

该选项也可以写作--print-file-name。

-print-libgcc-file-name

输出库 libgcc.a 的全路径名。该选项与-print-file-name=libgcc.a 相同。

该选项也可以写作--print-libgcc-file-name。

--print-missing-file-dependencies

Pre

与选项-MG 相同。

-print-multi-directory

输出从命令行中的 multilib 选择项中选出的库的对应目录名。目录名由环境变量 GCC_EXEC_PREFIX 确定。

该选项也可以写作--print-multi-directory。

-print-multi-lib

输出多库（multilib）选择的目录名，同时输出用来作出该选择的命令行选项。

该选项的输出使用分号(;)分隔，并使用 at 符号(@)替代连字符（-）指定命令行选项名称。这样做可以简化文本的 shell 处理。

该选项也可以写作--print-multi-lib。

-print-prog-name=program

输出指定程序（如 cc1 或 cpp0）的全路径名。除了列出路径名列表，编译程序不进行其他动作。参见选项-print-file-name。

该选项也可以写作--print-prog-name。

### -print-search-dirs

输出所有目录的完全路径名的列表，GCC 将在该列表中搜索作为子进程运行的程序和连接过程所用的库。编译程序不进行其他动作。

如果没找到程序，可以将其放在所搜索的一个目录中，或在环境变量 GCC_EXEC_PREFIX 中加入程序所在的目录。

该选项也可以写作--print-search-dirs。

### --profile

与选项-p 相同。

### --profile-blocks

与选项-a 相同。

### -Q

编译程序输出它正在编译的每个函数的名字，并在每个阶段后输出诊断统计信息，包括编译和链接程序所需的时间。

### -remap

Pre

该选项指示预处理程序在每个含有包含文件的目录中检查名为 header.gcc 的文件是否存在。如果存在，则使用该文件来确定被查看文件的真实文件名。该文件的每一行包含被查看文件和其真实的文件名。例如，如下 header.gcc 文件中的代码行可以用短文件名代替长文件名：

```
NotSupportedException.h notsup.h
RollbackException.h rollbak.h
TransactionRequiredException.h transreq.h
```

### -s

Linker

删除可执行程序中的符号名和所有重定位信息。其结果与运行命令 strip 所达到的效果相同。

### -S

不调用汇编器或链接器。该选项允许将源文件编译为汇编语言文件，但不运行汇编器将其生成为可执行程序。所产生的输出存放在与源文件同名但以.s 为后缀的文件中。如表 D-1 中所示，任何没有可识别后缀的输入文件或没有使用选项-x 给出其类型的输入文件将被忽略。

该选项也可以写作--assemble。

### -save-temps

不进行编译程序的正常步骤，即删除编译程序所生成的临时文件。临时文件将保留在当前目录中，并具有与其内容相对应的后缀。文件的内容可以通过表 D-1 给出的文件后缀判断。

该选项也可以写作--save-temps。

### -shared

**Linker**

链接器将生成共享目标代码，该共享库可在运行时动态链接到程序形成完整可执行体。而且，如果使用 gcc 命令创建共享库作为其输出，该选项可以防止链接器将缺失 main()方法视为错误。

为了可以正确工作，应该一致地使用选项-fpic，-fPIC 以及目标平台选项编译构成同一库的所有共享目标模块。特别是，该选项可能需要生成特殊代码来让构造函数正常工作。由于不正确的选项设定而产生的错误可能很不明显，而且也没有警告消息。

参见选项-shared-libgcc，-static-libgcc 和-static。该选项也可以写作--shared。

### -shared-libgcc

**Linker**

该选项指定使用共享版本的 libgcc。在不支持共享库的系统上和没有该库的共享版本的系统上，该选项没有作用。

如果是通过 g++，gcj 或 g77 调用链接器的，该选项将被自动设定，这是由于需要对异常进行处理。当应用程序需要从一个共享库代码抛出另一个共享库捕获的异常时，需要使用共享版本的 libgcc。在这些环境下异常抛出器和捕获器都需要使用共享版本的 libgcc。

参见选项-shared，-static-libgcc 和-static。

### -specs=filename

gcc 驱动程序读取配置文件以确定哪些选项应该传递给哪些子进程（参见第 19 章）。该选项可以通过指定配置文件来覆盖默认配置，它将在默认配置文件读取后进行处理以修改调用子进程的某些规则集。

该选项也可以写作--specs。

### -static

**Linker**

链接器将忽略动态可链接库，同时通过将静态目标文件直接包含到结果目标文件完成对所有引用的解析。对于没有动态链接功能的系统，该选项无效。

参见选项-shared。该选项也可以写作--static。

### -static-libgcc

Linker

该选项指定使用静态版本的 libgcc。该选项可能会引起 C++和 Java 中的异常处理问题。参见选项-shared，-shared-libgcc 和-static。

### -std=name

C

该选项指定 C 语言标准。表 D-6 列出了可以识别的名字。

**表 D-6 指定语言标准的选项**

| 名字 | 描述 |
|---|---|
| iso9899:1990 | ISO C 89 标准。该选项同时设置了选项-fno-traditional，-fno-writable-strings，-fno-asm，-fno-nonansi-builtin 和-fno-noniso-default-format-attributes |
| iso9899:199409 | 修正的 ISO C 89 标准。该选项同时设置了选项-fno-traditional，-fno-writeable-strings，-fno-asm，-fno-nonansi-builtin 和-fno-noniso-default-format-attributes |
| iso9899:1999 | ISO C 99 标准。该选项还同时设置了选项-fno-traditional，-fno-writeable-strings，-fno-asm，-fno-nonansi-builtin 和-fno-noniso-default-format-attributes |
| c89 | 与 iso9899:1990 相同 |
| c99 | 与 iso9899:1999 相同 |
| gnu89 | 带有 GNU 扩展和某些 ISO C 99 功能的 ISO C 89 标准。该选项还同时设置了选项-fno-traditional，-fno-writeable-strings，-fasm，-fnonansi-builtin 和-fnoniso-default-format-attributes |

该选项禁止 GNU C 扩展关键字 asm，typeof 和 inline，但它们的替代关键字__asm__，__typeof__和__inline__仍然有效。

参见选项-ansi。该选项可以写作--std。

### -symbolic

Linker

当构建共享目标代码时绑定对全局符号的引用。这是使用-shared 或-static 链接可执行文件的替代方式。只有很少一部分系统支持这一选项，如某些 SVR4 系统和 DG/UX。

该选项也可以写作--symbolic。

### -syntax-only

检查输入源代码的语法，报告任何警告或错误，并停止。

### --target machine

与选项-b 相同。

--target-help

显示目标机相关的命令行选项列表。参见选项--help。

--trace-includes

Pre

与选项-H 相同。

-traditional

C

该选项已废止。它试图支持最初的 K&R C 语言中的某些功能。如果传统的程序包含了 ISO C 头文件，它将无法编译成功。该选项将设置选项-traditional-cpp 和-fwritable-strings。

参见选项-fallow-single-precision。该选项也可以写作--traditional。

-traditional-cpp

C

C 预处理程序将支持最初的预处理程序功能。

该选项也可以写作--traditional-cpp。

-trigraphs

Pre

支持 ISO C 三字母词。选项-ansi 和-std 隐含该选项。

当设置这一选项时，三字母词（均以??开始）共有 9 个，每个三字母序列都按下表被翻译为单独的字符：

```
??=  #        ??(  [        ?? <  {
??/  \        ??)  ]        ?? >  }
??'  ^        ??!  |        ??-  ~
```

该选项也可以写作--trigraphs。

-time

输出编译一个程序的每个子进程所消耗的时间。每一行所列出的时间有用户时间（子进程代码所消耗的时间）和系统时间（系统调用所消耗的时间）。下面的例子给出了将 C++ 程序编译为可执行目标文件的例子：

```
gcc -time fortest.cpp -o fortest.o
# cclplus 0.14 0.05
# as 0.00 0.01
# collect2 0.10 0.03
```

该选项也可以写作--time。

-u name

Linker

指定的名字将被插入链接器的符号表且要求必须被解析。之后，链接器将装载包含该符号定义的目标模块以解析该符号。

该选项也可以写作--force-link。

-Umacro

Pre

如果先前已定义了宏 macro，宏 macro 将被删除。

所有-D 选项将在选项-U 前处理，所有-U 选项将在选项-include 或-imacros 前处理。

该选项也可以写作--undefined-macro。

-undef

Pre

预处理程序将不会预定义任何非标准宏。该选项将禁止体系结构定义，如__unix__，__OpenBSD__，__mips__，__linux__，__vax__，等等。

--undefine-macro macro

Pre

与选项-U 相同。

--use-version version version

与选项-V 相同。

--user-dependencies

Pre

与选项-MM 相同。

-v

显示编译程序的当前版本号，并显示用于运行编译和链接进程的每一阶段的命令。当单独使用时，该选项将显示编译程序的版本号。当同选项--help 一起使用时，将显示完整的命令行选项列表。

参见选项-###。该选项也可以写作--verbose。对 Fortran 而言该选项还将设置选项-fversion。

-V version

指定 gcc 将要运行的版本。只有在安装多于一个版本的编译程序时该选项才有意义。默认情况是运行最新安装的版本。

其工作原理是修改用于选择编译程序及其组件的前缀，这些组件通常都安装在/usr/local/lib/gcc-lib/machine/version 目录下。

参见选项-b 和-B。该选项也可以写作--use-version。

**--verbose**

与选项-v 相同。

**-w**

不产生警告消息。与选项--no-warnings 相同。

**-W**

该选项将打开一系列的警告消息，这些警告消息会报告将可能引起问题的代码，但也可能是程序员所需要的代码。该选项将打开下面的警告：

- **比较** 当无符号值进行小于 0 的比较时给出警告消息。例如，因为无符号数值不可能为负值，下面的测试永远为假：

```
unsigned int x;
  . . .
if(x  0) . . .
```

- **比较** 如将有符号值和无符号值进行比较则给出警告消息。为了进行比较，无符号值要转换为有符号值，可能会引起不正确的结果。该警告可以通过选项-Wno-sign-compare 关闭。
- **比较** 代数与 C 语法中比较的表示有所不同。下面的语句将会产生警告消息：

```
if(a < b < c) . . .
```

在代数表示中，该表达式只有在 b 处于由 a 到 c 的开区间才为真。在 C 中，该表达式与下面的例子相同，这与代数表示完全不同：

```
int result;
result = a < b;
if(result < b) . . .
```

- **常量返回值** 当函数被声明为 const 时，将对函数返回值产生警告。由于函数的返回值是右值(rvalue)，因此这里的 const 没有意义。
- **聚集初始值** 当为聚集数据类型指定初始值，但并不是聚集的所有成员均被指定了初始值时，会产生警告消息。在下面的例子中，数组和结构均会产生错误消息：

```
struct {
    int a;
    int b;
    int c;
```

```
} trmp = { 1, 2 };
int arr[10] = { 1, 2, 3, 4, 5 };
```

- **没有副作用** 对于没有作用的语句会产生警告消息。例如，下面的加法操作的结果并没有用处：

```
int a = 1;
int b = 2;
a + b;
```

- **溢出** 在 Fortran 中，当浮点常量声明溢出时会产生警告消息。
- **返回值** 当函数被写成可以返回（或不返回）值时产生警告消息。下面的例子中，如果 x 为负将不会有返回值：

```
ambigret(int x)
{
    if(x >= 0)
        return(x);
}
```

- **static 语法** 如果关键字 static 没有用在一行声明的起始将产生警告消息。该次序已不再是标准 C 的需求了。
- **无用参数** 如果选项-Wall 或-Wunused 与选项-W 一同使用，函数体中任何没有作用的参数均会引发警告消息。

该选项可以写作--extra-warnings。

### -Wa,optionlist

Asm

optionlist 是一个或多个由逗号分隔的可以传递给汇编器的选项列表。这些选项由逗号分隔且每一个均可作为命令行选项传递给汇编器。

参见选项-Wp 和-Wl。该选项也可以写作--for-assembler。

### -Waggregate-return

C C++ ObjC

如果函数返回一个结构、联合或是数组时，产生警告消息。

### -Wall

对 C 和 ObjC 而言，设置该选项与设置选项-Wreturn-type，-Wunused，-Wimplicit，-Wswitch，-Wformat，-Wparentheses，-Wmissing-braces，-Wsign-compare 和-Wmultichar 相同。选项-Wunknown-pragmas 仅对那些没有在系统头文件中探测到的代码进行设定。选项-Wuninitialized 仅在指定选项-O 时设定。

对 C++而言，附加的选项设置有-Wctor-dtor-privacy，-Wnon-virtual-dtor，-Wreorder 和-Wnon-template-friend。

对 Fortran 而言，有效的选项设置仅有-Wunused 和-Wuninitialized。

对 Java 而言，这与设置选项-Wredundant-modifiers，-Wextraneous-semicolon 和-Wunused 相同。

该选项也可以写作--all-warnings。

### --warn-

与选项-W 相同。

### -Wbad-function-cast

C

无论何时函数被转换为非匹配类型时均产生警告消息。例如，下面的例子将会在函数调用时产生警告：

```
int glim()
{
   return(88);
}
 . . .
char *cp;
cp = (char *)glim();
```

### -Wcast-align

C C++ ObjC

当指针转换为另一不同类型而引起对齐问题时产生警告消息。例如，在某些机器上仅能在 2 或 4 字节边界上访问 int 类型，因此将一个 char 指针转换为 int 指针时会产生非法地址值。

### -Wcast-qual

C C++ ObjC

无论何时去掉函数调用的限定词时产生警告消息。例如，下面的例子去除了 const 限定词：

```
const char *conchp;
char *chp;
 . . .
chp = (char *)conchp;
```

### -Wchar-subscripts

C C++ ObjC

如果 char 数据类型被当作数组下标使用时产生警告消息。char 通常默认是有符号的，因此这很可能是某些错误的根源。

### -Wcomment

C C++ ObjC

当在/* ... */注释中发现/*时产生警告消息。而且，在//注释结尾发现反斜线时（这将指出下一行也是注释）也会产生警告消息。

### -Wconversion

C C++ ObjC

当原型引起类型转换（而不是原型不存在时引起的类型转换）时产生警告消息。这包括在实数和整数类型间进行转换，在有符号和无符号数值间进行转换，以及改变数值的宽度。仅对隐式的转换（强制转换）给出警告，而非指定的类型转换。例如，下面例子的第一个赋值语句将引起警告，但第二句则不会：

```
unsigned int recp;
recp = -1;
recp = (unsigned int)-1;
```

### -Wctor-dtor-privacy

C++

当某个类显然是不可用时产生警告，这是由于它只有私有构造函数和析构函数，没有友员函数和公用或静态成员。选项-Wall 设置了该选项。

### -Wdeprecated

C++

这是默认选项。当使用已废止的 C++功能时产生警告消息，除非使用了选项-Wno-deprecated。

### -Wdeprecated-declarations

C C++ ObjC

这是默认选项。当使用已废止的属性时产生警告消息，除非使用了选项-Wno-deprecated=declarations。

### -Wdisabled-optimization

当所请求的优化被关闭时产生警告消息。出现这一情况的原因不在于代码，而是因为编译程序自身的限制。GCC 在优化太复杂或是完成优化耗时太多时会拒绝进行优化。

### -Wdiv-by-zero

这是默认选项。当编译程序检测到整数除数为 0 时产生警告消息，除非使用了选项-Wno-div-by-zero。当浮点除数为 0 时不产生警告消息。

### -Weffc++

C++

当使用了违反由 Scott Myers 所著 *Effective C++*一书中给出的编码风格的代码时产生警告消息。由于标准库头文件不遵守这些原则，因此这些代码也会引起警告消息。

-Werror

将所有的警告转换为错误消息。

-Werror-implicit-function-declaration

当在声明函数前使用函数时产生错误。参见选项-Wimplicit-function-declaration。

-Wextern-inline

当函数被同时声明为 extern 和 inline 时产生警告消息。

-Wextraneous-semicolon

当分号没有可以终结的语句时产生警告消息。空语句已经被废止。

-Wfloat-equal

当对两个浮点数进行相等比较时会产生警告消息，因为从程序逻辑的角度而言这通常是错误的。

很显然，计算实数的浮点运算很少会产生相同的数值。这就是说比较两个实数是否完全相等可能会失效，即使这两个数非常接近以至于可以从程序的逻辑上将它们认为是相同的。如果你的程序将一个数落在另一个数的 0.00001 范围内视为相等的话，下面给出了一种比较浮点数的方法：

```
double delta = 0.00001;
 . . .
if((val1 > val2-delta) && (val1 < val2+delta) {
   /* val1 and val2 are considered equal */
}
```

-Wformat

检查对诸如 printf()和 scanf()等函数的调用，并当命令中的参数类型与格式字串中所指定的类型不匹配时产生警告消息。例如，下面的语句试图将一个 double 值格式化为一个 int 类型时产生警告消息：

```
double dvalue = 44.44;
 . . .
printf("The value %d is bad.\n",dvalue);
```

格式将按照 GNU libc 版本 2.2 的功能进行测试，该版本包含了 C89，C99，POSIX 和某些 BSD GNU 扩展中的功能。如果同时指定了选项-pedantic，还会对非标准部分的格式给出警告消息。

该选项所测试的带有格式字串的函数有 printf()，fprintf()，sprintf()，scanf()，fscanf()，strftime()，vprintf()，vfprintf()和 vsprintf()。对 C99 而言则包含 snprintf()，vsnprintf()，vscanf()，vfscanf()和 vsscanf()。对 X/Open 而言则包含 strfmon()，printf_unlocked()和 fprintf_unlocked()。

参见选项-Wformat-extra-args，-Wformat-nonliteral 和-Wformat-security。参见第 4 章的 4.3.5 节“属性”。选项-Wall 设置了该选项。

**-Wformat=2** C C++ ObjC

与设置了选项-Wformat，-Wformat-nonliteral 和 -Wformat-security 相同。

**-Wformat-extra-args** C C++ ObjC

这是默认选项。如果同时指定了选项-Wformat，使用选项-Wno-format-extra-args 将禁止对诸如 printf()和 scanf()等函数使用额外（未使用）参数进行调用的警告消息。

**-Wformat-nonliteral** C C++ ObjC

如果使用了选项-Wformat，而且一些函数调用（例如 printf()和 scanf()）所使用的格式字符串不是可用于检查的常量时，就会发出警告。

**-Wformat-security** C C++ ObjC

当使用选项-Wformat 而且对诸如 printf()和 scanf()的函数调用可能存在安全问题时，发出警告消息。使用变量（而非常量）作为格式化字符串进行函数调用时，由于可能使用%n，因此不信任这类调用。

**-Wformat-y2k** C C++ ObjC

这是默认选项。使用选项-Wno-format-y2k 将禁止对 strftime()产生的两位数年份产生警告消息。

**-Wglobals** Fortran

这是默认选项。使用选项-Wno-globals 将禁止对子例程、函数、数据块或是具有相同特质名的通用块的全局名产生警告消息。同时还会禁止对不一致的全局函数和子例程（如不同的参数数目或不同的参数类型）给出警告消息。

**-Wimplicit** Fortran

无论何时隐式地声明了变量、数组或是函数，均给出警告消息。其效果与声明了 IMPLICIT NONE 相似。

-Wimplicit-int

C

当声明并没有指令某个类型时产生警告消息。选项-Wimplicit 和-Wall 会设置该选项。

-Wimplicit-function-declaration

C

无论何时在声明前使用函数将产生警告消息。参见选项-Werror-implicit-function-declarations。选项-Wimplicit 和-Wall 会设置该选项。

-Wimplicit

C

与选项-Wimplicit-int 和-Wimplicit-function-definition 相同。选项-Wall 会设置该选项。

-Wimport

C C++ ObjC

这是默认选项。使用选项-Wno-import 将会禁止由预处理程序对使用#import 产生的警告消息。

-Winline

C C++ ObjC

当函数被声明为 inline 但不能被扩展为内嵌代码时产生警告消息。

-Wl,optionlist

Linker

optionlist 是由一个或多个逗号分隔的传递给链接器的选项列表。这些选项由逗号分隔且每一个选项均会作为命令行选项提供给链接器。

参见选项-Xlinker，-Wa 和-Wp。

-Wlarger-than-size

C C++ ObjC Java

无论何时目标代码大于所声明的 size 字节时产生警告消息，如果函数返回值比 size 字节大，也会产生警告消息。

-Wlong-long

C C++ ObjC

这是默认选项，但仅当同时使用了选项-pedantic 时才会有效。当使用 long long 数据类型时该选项产生警告消息。可以使用-Wno-long-long 关闭这一默认选项。

-Wmain

C C++

当 main()定义存在可疑之处时产生警告消息。main()应该是一个外部可链接且返回一个 int 的函数。它应该具有 0 个、2 个或 3 个正确类型的参数。

-Wmissing-braces

C C++ ObjC

当数组的初始值没有完全被括号包含时产生警告消息。在下面的例子中，数组 a 和 b

均初始化正确，但数组 b 的括号更准确地表达了数值的位置：

```
int a[2][2] = { 0, 1, 2, 3 };
int b[2][2] = { { 1, 2 }, { 3, 4} };
```

选项-Wall 会设置该选项。

### -Wmissing-declarations

C

当定义了全局函数但没有在之前进行声明，或者没有指定参数类型，此时产生警告消息。参见选项-Wstrict-prototypes 和-Wmissing-prototypes。

### -Wmissing-format-attribute

C C++ ObjC

针对 format 属性的候选函数产生警告消息。需要注意的是，对可能的 noreturn 候选函数产生警告消息。如果没有使用选项-Wformat 或-Wall，该选项无效。

### -Wmissing-noreturn

C C++ ObjC

针对可能用于 noreturn 属性的函数产生警告消息。需要注意的是，对可能的 noreturn 候选函数产生警告消息。特别要注意那些实际有返回值的函数，由于可能会在程序中引起很微妙的错误，决不能将它们声明为 noreturn 属性。

### -Wmissing-prototypes

C

当定义了全局函数但没有在之前给出带有参数类型的原型声明时，产生警告消息。参见选项-Wstrict-prototypes 和-Wmissing-declarations。

### -Wmultichar

C C++ ObjC

这是默认选项。当字符常量被声明为包含多于一个字符时，如‘ab’或‘Plop’，产生警告消息。由这类声明产生的代码会依赖于所使用的平台，不能在可移植的代码中使用，但可使用选项-Wno-multichar 关闭警告消息。

### -Wnested-externs

C

对函数体内的 extern 声明产生警告消息。

### -Wnon-template-friend

C++

这是默认选项。当非模板化友元函数被声明为模板的成员时产生警告消息。

在 C++标准定义之前，GNU C++将友元的名字实现为一个非限定的 id。默认情况下已不再是这样的行为了，但为了已有的代码，可以使用选项-Wno-non-template-friend 关闭该警告消息。选项-Wall 会设置该选项。

### -Wnon-virtual-dtor

C++

当非虚析构函数应该定义为虚函数时产生警告消息。如果某个对象被作为其超类引用并被释放时，非虚析构函数将无法被执行。选项-Wall 会设置该选项。

### -Wold-style-cast

C++

当使用传统风格（C 语言风格）进行类型转换（而非 C++标准中定义的新形式之一，该新形式带有类型转换操作符 static_cast，const_cast 或 reinterpret_cast）时，产生警告消息。例如：

```
class A { ... };
class B: public A { ... };
 . . .
A* a = new A();
B* b = a;                          //implicit conversion
A* a2 = static_cast <A*> (b);      //reversing an implicit conversion
```

### -Wout-of-date

Java

这是默认选项。使用选项-Wno-out-of-date 将禁止编译程序在源文件比其对应的类文件新时产生警告消息。

### -Woverloaded-virtual

C++

当函数声明隐藏了基类中声明的虚函数时产生警告消息。在下面的例子中，类 A 中的虚函数 fn()被隐藏了：

```
class A {
   virtual void fn();
};
class B: public A {
   void fn(int);
};
```

### -Wp,optionlist

Pre

optionlist 是由一个或多个逗号分隔的传递给预处理程序的选项列表。这些选项由逗号分隔，且每一个选项均被作为命令行选项传递给预处理程序。

参见选项-Wa 和-Wl。

### -Wpacked

C C++ ObjC

当使用了 packed 属性但却没有作用时产生警告消息。例如，下面的 struct 在没有 packed 属性的时候也会使用 4 个字节：

```
struct fourbyte {
    short x;
    char a;
    char b;
}__attribute__((packed));
```

参见第4章的4.3.5节“属性”。参见选项-fpack-struct和-Wpadded。

### -Wpadded

C C++ ObjC

当编译程序在结构的成员间插入填充字，以保证某一项在结构内部对齐或使整个结构对齐时，产生警告消息。在某些情况下，可以通过重新组织结构来对齐成员，而且可通过去除填充字来达到减少结构尺寸的目的。例如，下面的结构将在short数据类型前插入一个字节来保证它在2字节边界上对齐：

```
struct pad {
    char a;
    short b;
    char c;
};
```

### -Wparentheses

C C++ ObjC

对那些语法上虽然正确，但由于操作符的优先级或代码的结构而引起程序员混淆的构建子产生警告消息。

下面的表达式将产生一条警告消息，这是因为很难记住逻辑操作符是从左向右还是从右向左计算的：

```
if(a && b || c) . . .
```

下面的代码会引发一条警告消息，这是因为没有括号的话，if和else语句间的关系很容易被误解：

```
if(a)
    if(b)
        m = p;
else
    a = 0;
```

很显然，从缩进来说，程序很可能将else语句和第一个if语句联系在一起，但事实正相反。

选项-Wall会设置该选项。

### -Wpmf-conversions

C++

这是默认选项。使用选项-Wno-pmf-conversion 将引起一条警告消息，指出存在一个从指针到成员函数地址的转换。

### -Wpointer-arith

C C++ ObjC

针对那些依赖于函数类型大小或 void 大小的代码产生警告消息。由于指针运算的缘故，这些项在 GCC 中的默认大小是 1。

### -Wprotocol

ObjC

这是默认选项。当协议所需要的方法未在采纳该协议的类中实现时，指定-Wno-protocol 将禁止产生警告消息。

### -Wredundant-decls

C C++ ObjC

对在同一作用域中声明多次的内容产生警告消息。即使当声明是完全一样的也会产生警告消息。

### -Wredundant-modifiers

Java

当在声明中使用了不必要的修饰词时会产生警告消息。例如，当某 interface 的方法被声明为 public 时会产生一个警告消息。

### -Wreorder

C++

当编译程序重组成员初始化函数以匹配它们所声明的顺序时产生警告消息。例如，下面的初始化函数必须进行重组：

```
class Reo {
   int i;
   int j;
   Reo(): j(5), i(10) { }
};
```

选项-Wall 会设置该选项。

### -Wreturn-type

C C++

当所有声明的函数没有声明返回类型，并默认使用 int 时产生警告消息。而且，如果在没有声明为 void 的函数中，返回语句没有返回值时会产生警告消息。选项-Wall 会设置该选项。

--write-dependencies **Pre**

与选项-MD 相同。

--write-user-dependencies **Pre**

与选项-MMD 相同。

-Wselector **ObjC**

如果选择子为不同类型定义了多个方法时产生警告消息。

-Wsequence-points **C**

如果变量在表达式中被多次引用，且某次引用修改了它的值，则会产生警告消息。C 语言的定义允许按照任意顺序计算顺序点间的表达式（只要保持运算符的优先级），因此在某处对变量进行修改将使在其他位置需要使用变量时，无法确定其值。

可以通过如下一些操作符指定代码中的顺序点：

```
; , && || ? :
```

下面是表达式的一些例子，由于违背了顺序点，因而具有不惟一的结果：

```
s = a[s++];
s = s--;
a[s++] = b[s];
a[s] = b[s += c];
```

-Wshadow **C C++ ObjC Java**

当局部变量屏蔽了参数、全局变量或是其他局部变量时产生警告消息。如果内建函数被屏蔽，也产生警告消息。

-Wsign-compare **C C++ ObjC**

当比较有符号值和无符号值，可能因在比较前将有符号值转换为无符号值而产生不正确结果时，产生警告消息。选项-Wall 会设置该选项，但可以使用选项-Wno-sign-compare 关闭该选项。

-Wsign-promo **C++**

当从无符号数据类型（或枚举类型）向具有相同尺寸的有符号数据类型进行重载时产生警告消息。标准定义了这种类型的转换，但它可能会造成数据丢失。

### -Wstrict-prototypes

C

当函数的定义或声明没有指明参数的数据类型和数目时，就会产生警告消息。参见选项-Wmissing-prototypes 和-Wmissing-declarations。

### -Wsurprising

Fortran

该选项将对可能有多种解释方法的构建子给出警告消息，这种情况可能会使程序员感到非常困惑。不同的编译程序对语言的构建子有不同的处理方式。这些警告消息包括：

- 在同一行中使用了两个运算符的表达式。这种情况的一个例子就是 x** y * z。在没有括号的情况下，编译程序可能将该表达式解释为 x** (y * z)或((x**y) * z。选项-fpedantic 也可能对这种情况发出警告消息。
- 带有模糊的一元负号的表达式。例如，-2 ** x 可以解释为-(2 ** x)或(-2) ** x。像-x*y 这样的表达式，如果某个值接近 x 或 y 数据类型值域的最大范围时可能会产生令人吃惊的结果，表达式可能被解释为-(x*y)而非(-x)*y。
- 使用实数而非整数作为循环计数器的 DO 循环可能会产生奇怪的结果。通常这并不是问题，但结果可能会依编译程序的不同而不同。

### -Wswitch

C C++ ObjC

当使用某个枚举类型作为 switch 语句的索引但又没有 default 和 case 语句处理所有可能值时，产生警告消息。选项-Wall 设置该选项。

### -Wsynth

C++

当操作符的合成与 cfront 不同时产生警告消息。在下面的例子中，GCC 合成操作符 A& operator = (const A&)；而 cfront 则使用用户定义的默认 operator =操作符。

```
class A {
    operator int();
    A& operator = (int);
};

main() {
    A a1;
    A a2;
    a1 = a2;
}
```

### -Wsystem-headers

C C++ ObjC

对系统头文件中的代码发出警告，该代码同时也在被编译程序中。通常禁止产生系统头文件产生的警告消息。

为了对系统头文件中未知 pragma 产生警告消息，需要使用选项-Wunknown-pragmas，这是因为使用选项-Wall 仅会检查程序中的 pragma 而忽略系统头文件中的 pragma。

### -Wtraditional

为标准 C 构建子发出警告，这里的构建具有不同的含义或在传统 C 中不存在。其中一些构建子含义模糊或是很容易产生问题，因此应该尽可能避免。

- **转换** 由于原型定义而引起定点和浮点数之间的转换时，产生警告消息。参见选项-Wconversion。例如，如下的函数调用可能会引起警告消息：

```
void takedouble(double dval);
 . . .
int eighty = 80;
takedouble(eighty);
```

- **外部** 如果在代码块内部将某函数声明为外部函数，且在该代码块的外部使用了该函数时，产生警告消息。
- **初始值** 当对自动聚集数据类型（如数组和在函数体内声明的结构）进行声明时，产生警告消息。
- **初始值** 当对联合赋予初始值时产生警告消息，除非它的初始值为 0。
- **标号** 当标号与变量具有相同名字的时候产生警告消息。
- **文字常量** 当使用前缀 U 声明整数常量，并用前缀 F 和 L 声明实数常量时，产生警告消息。
- **文字常量** 如果基数为 10 的 C 常量具有不同的宽度，或具有与传统 C 不同的有符号/无符号特性时，给出警告消息。仅对基数为 10 的常量给出警告消息，这是因为通常十六进制和八进制的常量均视为位模式。
- **文字常量** 当使用标准 C 中的字串连接方法时产生警告消息。
- **预处理程序** 当所命名的宏作为字符常量的一部分出现时产生警告消息。传统 C 中的字符串可以包含宏定义，但标准 C 中则不允许。
- **预处理程序** 当预处理程序指示字第一列对传统 C 未知时，产生警告消息。可以通过缩进使用新型的预处理程序命令，如#pragma 和#elif，从而使它们不出现在第一列中。（传统预处理程序要求所有的预处理程序命令均在第一列开始。）
- **预处理程序** 当定义为函数形式的宏却没有参数时产生警告消息。
- **静态** 如果非 static 函数在一个 static 函数后被声明时产生警告消息。（某些传统编译程序不接受这样的情况。）
- **开关语句** 当 switch 语句的操作数是 long 数据类型时产生警告消息。
- **一元加号** 当存在一元加号时产生警告消息。

-Wtrigraphs

C

当某个三字母词可能改变程序的含义时，产生警告消息。除了注释中的三字母词不被翻译以外，其他各处的三字母词均被翻译，因而可能某个三字母词会出现在字符串中。例如，Linux 内核的某个版本就包含这样一个字符串"imm: parity error (???)\n"，在标准 C 中它将被翻译为"imm: parity error (?]\n"。

-Wundef

C C++ ObjC

当没有定义的标识符出现在#if 命令的表达式中时，产生警告消息。

-Wuninitialized

如果自动变量在初始化之前就被使用，产生警告消息。而且，如果 setjmp()调用可能破坏自动变量值的时候也会产生警告消息。该选项只能同选项-O 一同使用，因为这些情况的探测需要使用优化数据流的信息。

由于该选项要求数据流分析，因此不可能完全准确。例如，在下面的代码中，对 printf()语句而言，不可能保证 value 已经初始化或未经初始化。

```
int value;
if(a  b)
   value = 5;
else if(a  c)
   value = 10;
printf("%d\n",value);
```

由于数据流分析的本质，该选项不能作用于结构、联合、数组、所有声明为 volatile 的变量、所有通过地址引用的变量或用来计算从不使用的数值的变量。

数据流分析可以理解 setjmp()语句，但它不可能知道在哪里会调用 longjmp()，因此即使没有任何问题也可能会产生警告消息。

Fortran 源代码需要某种数据流分析，如同下面的例子所示，可能无法确定 TVAL 是否总是被初始化了：

```
IF (IVAL .EQ. 1) TVAL = 5
IF (IVAL .EQ. 2) TVAL = 10
CALL SMON(TVAL)
```

该选项会报告一些假的警告消息，可以通过将函数声明为不返回值 noreturn 来禁止这种警告消息。

如果与选项-O 一同使用，选项-Wall 将设置该选项。

-Wunknown-pragmas

当遇到未知的#pragma 指示字时产生警告消息。除非已由选项-Wall 设置，不然该选项将对系统头文件中任何未知的#pragma 指示字产生警告消息。

-Wunreachable-code

如果在当前的程序执行中检测到不可达代码，则产生警告消息。如果代码存在永远被选取的分支，或是调用了不会返回的函数时就存在不可达代码。

应该谨慎地去除由该警告所报告的代码。很可能从可以产生内嵌代码的函数或是展开的宏中产生警告消息，但也有其他实例不产生不可达代码。而且，不可达代码也可能是编译时的选项导致的内部可忽略代码。

-Wunused

该选项设置选项-Wunused-function，-Wunused-label，-Wunused-parameter，-Wunused-value 和-Wunused-variable。

对 Fortran 而言，当变量被声明但没有使用时产生警告消息。

选项-Wall 会设置该选项。

-Wunused-function

当存在一个未使用的静态函数的定义或是只声明但没有定义的静态函数时，产生警告消息。选项-Wunused 和-Wall 将设置该选项。

-Wunused-label

当定义了未使用的标号而没有使用 unused 属性时，产生警告消息。选项-Wunused 和-Wall 也会设置该选项。

-Wunused-parameter

当声明了未使用的参数而没有使用 unused 属性时，产生警告消息。选项-Wunused 和-Wall 也会设置该选项。

-Wunused-value

当存在未使用的局部变量，或声明了非常量的静态变量而没有使用 unused 属性时，产生警告消息。选项-Wunused 和-Wall 也会设置该选项。

-Wunused-variable

当存在未使用的局部变量，或声明了非常量的静态变量而没有使用 unused 属性时，产生警告消息。选项-Wunused 和-Wall 也会设置该选项。

### -Wwrite-strings

C C++

当 C 程序在声明为 const 的指针中保存了某字符串常量的地址时，产生警告消息。当 C++将一个字符串常量转换为 char *类型时，产生警告消息。

仅当需要在代码中谨慎地使用 const 来声明数据类型和原型时，该选项才会有用，否则所产生的警告消息是非常烦人的。

### -xlanguage

指定命令行中所指文件的内容。如果不使用该选项，文件的内容一般通过文件名中的后缀确定。该选项作用于命令行中跟在其后的所有文件名。在下面的例子中，名为 morg.jmp 和 frampl 的文件均视为 C 源代码文件。

```
gcc -xc morg.jmp frampl
```

可以多次使用该选项来改变所需语言，而且还可以使用特殊名 none 来关闭该选项。例如，下面的命令行指定名为 murk 和 stim.wad 的文件均为 C++源代码，名为 hummer.c 的文件为 Java 源代码，而 slamm.c 则默认为 C 源代码。

```
gcc -xc++ murk stim.wad -xjava hummer.c -xnone slamm.c
```

选项-x 也可以写作--language。参见表 D-7。

**表 D-7 -x 选项的语言指示字**

| 语言名 | 描述 |
|---|---|
| ada | Ada 源代码 |
| assembler | 不需要预处理的汇编语言 |
| assembler-with-cpp | 需要预处理的汇编语言 |
| c | 需要预处理的 C 源代码 |
| c++ | 需要预处理的 C++源代码 |
| c++-cpp-output | 不需要预处理的 C++源代码 |
| c-header | C 头文件 |
| cpp-output | 不需要预处理的 C 源代码 |
| f77 | 不需要预处理的 Fortran 源代码 |
| f77-cpp-input | 需要预处理的 Fortran 源代码 |
| java | Java 源代码 |
| objc-cpp-output | 不需要预处理的 Objective C 源代码 |
| objective-c | 需要预处理的 Objective C 源代码 |
| ratfor | 需要由 RATFOR 预处理程序处理的 Fortran 源代码 |

-Xlinker option

Linker

向链接器传递一个选项。这主要用来指定与系统相关的链接器选项。

例如，如果你正在使用 System V 链接器而且需要对其使用选项-all，可以使用选项-Xlinker-all 完成这一任务。在链接器命令行中，指定多个选项时应当多次重复使用-Xlinker。例如，要指定选项-woff 5,17，可能需要使用选项-Xlinker -woff -Xlinker 5,17。如写成-Xlinker "-woff 5,17"则无法工作。

参见选项-Wl。该选项也可以被写作--for-linker。

# 附录 E GCC 术语表

**absolute address（绝对地址）**

绝对地址是一个惟一的可用来指定某个内存字节的数字数值。参见相对地址。

**address（地址）**

参见绝对地址和相对地址。

**aggregate（聚集）**

包含多于一个基本数据类型的数据类型。例如，数组是一种聚集，C 的结构（struct）也是。

**aliasing（别名机制）**

同一内存地址可直接或间接地通过两个或多个不同名字（可能还有不同的数据结构）进行访问，这被叫做别名。由于在寄存器中保留数值是比较常见的做法，因此这是对于优化采取的特殊考虑。

**ANSI（American National Standards Institute，美国国家标准协会）**

管理和协调美国自发标准的组织。

**archive（文档）**

参见库。

**assembler（汇编器）**

一个平台相关的程序，它读取汇编语言源文件（机器代码的助记表示）并将其翻译为作为连接程序输入的二进制目标文件。

**backtrace（回退跟踪）**

GNU 调试器可以打印函数名和地址的列表，它会被调用以达到程序的当前执行点。这些信息，包括函数地址和参数值，称为回退跟踪。

**BFD（Binary File Descriptor，二进制文件描述字）**

包含处理各种二进制文件格式完成各种底层操作的例程的库。

**BSD（Berkeley Software/Standard Distribution，Berkeley 软件/标准发行）**

一种 UNIX 操作系统。它也是几种其他现代 UNIX 系统的基础。参见 SVR4。

**bss**

由 UNIX 连接程序生成的可执行文件的未初始化数据段。它包含的数据只有地址但不包含任何空间。因此，直到程序被加载时才会分配空间。在可执行文件中，bss 变量只被赋以名字、尺寸和位置。参见 text 和 data。

**built in function（内嵌函数）**

由编译程序所生成的函数体称为内嵌函数。一个内嵌函数可以是相应标准库函数的优化版本、编译程序的附加功能或是实现诸如变长参数列表等内部使用的函数。

**bytecode（字节码）**

编译 Java 程序所生成的目标代码的可移植形式。字节码由 Java 虚拟机解释以执行 Java 程序。

**C89**

1989 ANSI C 标准。

**C99**

1999 ANSI C 标准。

**calling convention（调用规范）**

参见调用序列。

**calling sequence（调用序列）**

用来调用函数的汇编语言的语句序列。该序列设置需要传递的参数、保存返回的地址以保证可以找到调用函数，执行调用并管理返回值（如果有的话）。也叫做调用规范。

CCP（Conditional Code Propagation，条件代码传播）

一种优化技术，可以发现某值对所有可能的执行路径均为常量，并用这一事实来探测和删除不可能被执行的代码。

cfront

最初的C++实现是一个称为cfront的AT&T程序，它将C++源代码翻译为C源代码。

class（类）

1. 在面向对象编程中，一个类就是一个对象类型的定义。由它所生成的对象，由于它们具有相同的接口和行为集，称它们属于同一类。2. 在Java中，包含已编译类的文件称为类文件，或简称为类。

clobber（破坏）

如果存储位置（通常为一个寄存器）已经被用作一个临时工作区，它就不再保存所预期的数值，这个存储位置就称为被破坏（clobbered）了。

CNI（Cygnus Native Interface，Cygnus本地接口）

在C++中，用于编写本地Java方法的工具。参见JNI。

code（代码）

这个术语是指计算机上运行的任何形式的指令列表。代码可以是所有形式，从人可读的编程源码到机器可读的操作码的位模式。

code propagation（代码传播）

参见CPP。

coercion（自动转换）

由一种基础数据类型到另一种的自动转换（没有强制类型转换或函数调用）。

COFF（Common Object File Format，通用目标文件格式）

一种可跨系统移植的并由各种不同的汇编器和连接程序所了解的目标文件的标准格式。参见ECOFF和XCOFF。

COMDAT（Common Data，通用数据）

可以在不只一个目标文件中被复制的一种数据或可执行项（或项目的集合）。当将目标

文件和库或可执行体组合起来时，连接程序会去掉所有通用数据，只留下其中一份数据。这也称为合拢或通用数据合拢。

**common（通用）**

可为公用块分配的全局变量属性。

**common block（公用块）**

GNU 连接程序创建一个公用块作为全局变量的分配空间。如果不同的目标文件多次声明了相同的全局变量，它们会被解析为公用块中单独的一个变量。参见 COMDAT。

**compilation unit（编译单元）**

可被编译为单独目标文件的单独源代码单位。它通常就是单独的一个源文件，但也可以包含其他辅助编译的源代码（如 C 程序中#include 文件）。也称为翻译单元。

**compiler（编译程序）**

读取计算机程序的源文件（或文本文件）的软件集合，可将指令翻译成计算机可执行的格式。编译程序也称为翻译器（translator）。

**copyleft（版权所属）**

同通用许可证，它声明程序是自由软件并且所有修改和扩展的软件版本均可以自由发行。参见 GPL。

**cross compile（交叉编译）**

使用一种编译程序创建在完全不同平台上执行的可执行文件。

**CPP（C Preprocessor，C 预处理程序）**

预处理程序读取程序源码，并处理其中的指示字以产生源码的修改版本。

**cruft**

随着软件的发展，以及经历了修改 bug 和更新的若干周期，它的部分代码已不再使用但仍然保留在源码中。这种代码称为 cruft。cruft 的尺寸范围可由一两行无用代码到整个源文件模块。由于很难识别 cruft，去除 cruft 往往很困难。

**CSE（Common Subexpression Elimination，通用子表达式消除）**

一种优化技术，它可以识别重复的表达式并重用它的值，而不是再次执行相应的计算。参见 GCSE。

### ctor

构造函数 constructor 的通用缩写。参见 dtor。

### CVS（Concurrent Version System，同步版本系统）

一种版本控制系统，它可以维护文本文件的版本历史信息。设计它是为使广泛分布于各地的不同开发者可同时访问。

### data

由 UNIX 连接程序生成的可执行文件的一部分，其中包含的数据具有初始值。该段包含的项具有名字、大小，且会分配相应的空间包含它的数值。参见 bss 和 text。

### DBX

一种交互式调试器，它可以用来一行行地跟踪程序的执行。DBX 是一种命令行调试器，但在许多变体中它具有 X GUI 界面和 emacs 界面。

### DCE（Dead Code Elimination，不可达代码去除)

一种删除不可能被执行的代码的优化技术。

### dead code（不可达代码）

在优化期间，可能留下某些代码但从不会被执行。这些就是不可达代码。优化器应该去除这些代码。

### demangle

解析出编码在已经拆分（mangle）了的函数名中的描述信息的过程。参见 mangle。

### deprecated（已废止的）

任何不再需要的编译程序的选项或功能（或是因为某些原因而认为不再恰当的）被称为是已废止的。它仍然存在但在以后的编译程序版本中可能去除。

### dereference（去引用）

表达式可能会涉及保存在指针中的地址。用这种方式使用的指针就称为是去引用的。

### directive（指示字）

1. 源代码中使用井号(#)开头的命令，它由预处理程序处理。2. 在汇编语言中，一条指示字就是一条针对汇编器的指令，而不是产生代码的操作码。汇编器指示字也称为伪操作。

**distention（膨胀）**

该名字用于某些特定 Fortran 语言扩展，现在均认为非常“丑陋”而不该使用。通过-fugly-*标记，g77 可支持某些扩展。

**dtor**

析构函数 destructor 的通用缩写。参见 ctor。

**DWARF（Debugging With Attribute Format，带属性的调试格式）**

一种用来在目标代码中插入调试信息的格式。

**DWARF2（Debugging With Attribute Format 2，带属性的调试格式 2）**

DWARF 的更新版本，该格式用来在目标代码中插入调试信息。

**dynamic library（动态库）**

参见库。

**ECOFF（Extended Common Object File Format，扩展通用目标文件格式）**

目标文件的标准格式，它具备系统间的可移植性，且不同汇编器和连接程序均了解该格式使用。参见 COFF 和 XCOFF。

**EH**

异常处理（exception handling）的缩写。

**elaborate（详细阐述）**

在 Ada 中，执行前的最后一步是通过插入必要的初始值和可执行指令来详细阐述代码，通常需要同一程序中其他编译单元的内容。参见可预阐述（preelaborable）。

**elaboration（详细阐述）**

在 Ada 语言中，详细阐述是所需的详细阐述软件包的过程。

**ELF（Executable and Linkable Format，可执行和可链接格式）**

Linux 二进制目标文件格式，它包含动态加载库和可执行代码的信息。ELF 由 COFF 格式演变而来，与 COFF 非常近似。

**elide（省略）**

一个被省略的函数就是使用调用者返回位置作为工作空间（而非建立自己的内部工作

空间）返回数值的函数，之后它会将返回值复制到返回地址中。优化器可以省略函数调用。

**entry point（入口点）**

可执行程序中的地址，程序从这里开始执行并称为入口点。

**fetch（读取）**

当计算机由内存中的程序读取指令并加载到 CPU 执行时，这称为指令的读取。参见预取（prefetch）。

**FPU（Floating Point Unit，浮点处理单元）**

同 CPU 一起工作处理浮点操作的硬件处理器。没有 FPU 的计算机将需要软件来仿真浮点操作。

**folding（合拢）**

参见 COMDAT。

**frame（栈）**

参见堆栈框架。

**function（函数）**

函数是一块可执行代码，它被赋予一个名字并可以在其他位置调用。一个函数也可以被定义为带有参数，它指定一列由调用者传递的参数值。参见成员函数和方法。

**garbage collection（无用信息收集）**

一个运行的程序的过程，它可以自动恢复动态分配但不再使用的内存。实现该过程有许多机制，但所有这些均可以被认为是无用信息收集。

**GCSE（Global Common Subexpression Elimination，全局公用子表达式去除）**

一种优化技术，它可以识别重复的表达式并重用它的数值，而不是再次执行相应的计算。参见 CSE。

**GNAT（GNU Ada Translator，GNU Ada 翻译器）**

原 Ada 前端的名字，现已成为 GCC 的一部分。

**GNATS（GNU Bug Tracking System，GNU 错误跟踪系统）**

用来跟踪 GCC 和其他 GNU 软件错误的在线系统。

**GOT（Global Offset Table，全局偏移表）**

目标文件中包含一组偏移的表格，可用来重新定位可执行代码。参见 PIC。

**GPL（General Public License，通用公共许可证）**

一种许可证，在其许可下软件被编写为 copyleft 格式的自由软件。

**header（头文件）**

由预处理程序执行 include 指示字并被包含在源文本文件中的文件。在 C，C++和 Objective C 中，传统上它是后缀为.h 的文件。

**Hollerith field（Hollerith 字段）**

在 Fortran 中，引用的字串被保存为字符数，后面跟着的是字符本身。可以用 Hollerith 字段创建字符串，它是一个长度后跟一个 H 字母以及字符串（例如，10HPhillips66）。

**host（主机）**

参见平台。

**i18n**

单词 internationalization 的简写形式，它由字母 i 开头，后跟 18 个字符，最后是字母 n。参见 110n。

**if-conversion（if-转换）**

一种优化过程，它会修改已生成的代码，以便分支之后的最可能被采用的路径比不可能采用的路径更有效。

**immediate（立即数）**

一个立即数值是一个常量，可以在汇编语言操作码中作为操作数使用。

**include guard（包含文件防护）**

习惯上使用预处理程序条件编译命令来定义一个环境变量，它可以在头文件的最先部分进行检测，以防止头文件被再次编译。如果变量尚未定义，头文件就会被编译了。

**induction variable（归纳变量）**

一种在循环体内递增的变量。循环计数器。

**inline（内联）**

一个函数的实体（或其他相似的语言成分）被包含在函数的调用点。就是说函数体将在代码中内嵌展开，而不是只作为对另一处函数体的一次调用。

**insn**

一种机器语言或 RTL 中间语言指令。该语言具有许多特殊用途的 insn，但最重要的一些构成了一种元汇编语言，可被翻译为目标机器上的汇编语言指令。

**instantiation（实例化）**

由类定义创建对象实例。

**intrinsic（特质）**

在 Fortran 中，特质函数是一种内嵌函数，由于它可以不声明而直接使用，因此被视为语言的一部分。

**invariant expression（不变表达式）**

循环体内的表达式，每次循环时所计算的数值是不变量，在优化时可以转移到循环外。

**ISO（International Organization or Standardization，国际标准化组织）**

一个成立于 1946 年的国际标准组织，ISO 的成员来自 75 个国家的标准组织，包括 ANSI。

**jar**

一种包含一个或多个 Java 类文件的档案文件。该文件还包含一个清单，是一个包含 jar 文件中类名列表的文本文件。

**Java Virtual Machine（Java 虚拟机）**

参见 JVM。

**JNI（Java Native Interface，Java 本地接口）**

一种标准编程接口，用于编写 Java 本地方法和将 JVM 嵌入到本地应用中。参见 CNI。

**JVM（Java Virtual Machine，Java 虚拟机）**

一个程序，它可从标准 Java 字节码格式的文件中读取指令并执行。向一个平台上移植 JVM 和标准类库实际上就是向该平台移植了所有的 Java 程序。

**l10n**

单词 localization 的简写，其起始字母是 l，其后是 10 个字母，结尾字母是 n。参见 i18n。

**lexical analysis（词法分析）**

也称为词法扫描，词法分析就是从程序源文件中读取输入字符流，并按照某种方式组合它们，形成名字、数字以及标点。由一些字符集合构成的单元称为一个 token。

**LGPL（Lesser General Public License，次要的通用公共许可证）**

该许可证可用于某些（但不是全部）GNU 库。该许可证允许库函数用于某些私有程序，通常这是 GPL 所不允许的。

**library（库）**

库是一个包含一个或多个目标文件的文件，它可以链接到其他目标文件组成可执行文件。静态库就是包含由连接程序永久链接到可执行程序的模块的库。静态库也称为文案。共享库就是包含可以临时链接到可执行程序并在执行时进行真正链接的模块的库。共享库也称为动态库。动态链接的程序就是包含到这些函数的引用的可执行程序，当程序执行时，它会从动态库中加载这些函数。参见静态库和相对地址。

**life analysis（生命周期分析）**

确定哪个数值应该留在寄存器中可供后用的过程，同时也确定哪个寄存器不会被使用，因为其中包含的数值不再使用。

**link editor（链接编辑器）**

与连接程序相同。

**linkage（链接）**

当调用函数时，必须用标准方法来存储和读取传递给函数的参数以及函数所返回的数值。这种协议称为链接，而且是混合两种语言时要解决的主要问题。

**linker（链接程序）**

一种平台相关的程序，可将一组目标文件（某些必须由库中析取）组合在一起生成一个可执行程序。

**lvalue（左值）**

一种任意类型的表达式，可以解析为内存单元的某个地址。这个术语源于 left value，就是指赋值语句中左边的部分的值。参见右值（rvalue）。

macro（宏）

在预处理程序中，使用#define 指示字声明的名字及其值就是一个宏，之后会用它来替代源码中的文本。

Makefile（make 程序的描述文件）

包含一组由 make 命令编译和连接程序用到的规则的文件，它也会根据文件的日期和时间戳执行一些其他任务。makefile 通常被命名为 makefile 或 Makefile。

mangle

C++（和 Java）编译程序修改成员函数（以及方法）的名字，来依据其参数个数和类型生成一个惟一名。这一称为 mangling 的过程允许重载简单成员函数（和方法）。参见 demangle。

manifest（清单）

参见 jar。

marshaling（列集）

将传递给远程函数的参数以及由远程函数返回的值进行串行化（转变为可以发送的字节流）的动作，这称为列集。将一个规整化的流转化为数据的过程称为反列集（unmarshaling）。参见串行化。

member function（成员函数）

在 C++中，在类中定义的函数是类的一个成员并称为成员函数。除非成员函数被声明为静态的，否则成员函数永远会在该类的某个对象的上下文中被调用并具有 this 指针，它可以在函数中引用该对象。C++中的成员函数类似于 Java 中的方法。

method（方法）

Java 中的方法就是类中定义的函数，它是类定义的一部分。除非方法被声明为静态的，否则方法始终会在类的对象的上下文中被调用并且具有 this 指针，它可以在函数中引用该对象。Java 中的方法就如同 C++中的成员函数。

MFC（Microsoft Foundation Class，微软基础类）

包装 Win32 用户接口 API 的类的层次。

mirror（镜像）

复制其他 Internet 站点的站点，可以使下载文件更方便。

**mnemonic（助记符）**

代表一个值的名字，目的是使其更容易记忆。这个术语通常用来指可由汇编器识别的 CPU 操作码，它可以转换为二进制代码。参见操作码。

**multilib**

如果单个目标机需要多于一种版本的库，这就是 multilib。例如，某平台可能具有浮点运算单元或没有，因此同样的数学库就需要使用不同选项设置编译两次，并使其可以在相应的配置下链接相应的程序。

**NaN（Not a Number，非数值）**

专指非法浮点数的 IEEE 标准名词。在下溢、溢出或某些其他非法浮点操作时会产生这一数值。

**NEXTSTEP**

一种计算机操作环境，它提供 GUI 界面并可以用于 HP，NeXT，Sun 和其他计算机上。最初该环境是为 NeXT 计算机系统开发的。

**NLM（Netware Loadable Module，Netware 可加载模块）**

可执行程序，它已被格式化为可以在 NetWare 系统上执行。

**NLS（Native Language Support，本地语言支持）**

GCC 编译程序的一种能力，它可以以本地语言（而非美国英语）输出诊断消息。NLS 是 i18n 和 l10n 的组合。

**noop（空操作）**

汇编语言指令，但它什么也不做。它通常用来作为填充字插入到可执行代码中，或用来作为分支目标处的指令。

**object（对象）**

1. 在面向对象编程中，对象就是数据项的集合，以及用于操作这些数据项的方法（或函数）。参见类。2. 编译过程的输出是目标文件（或目标代码），因为运行编译程序或连接程序的目的就是生成这种格式的文件。

**opcode（操作码）**

计算机 CPU 的一条指令。操作码可以是一条指令，能够完成两个数的相加、向寄存器

加载一个数、在内存中保存一个数值，或是硬件知道如何操作的其他指令。操作码是机器指令的一部分，不包括数据。参见助记符。

**ordered comparison（有序比较）**

两个浮点数之间的有序比较就是，当其中一个值为 NaN 时会产生异常的比较。参见无序比较。

**package（包）**

Java 中的包就是类的集合。而 Ada 中的包是过程的集合。

**pass（阶段）**

软件每次读取输入，进行分析、优化、代码生成、预处理或其他操作就称为一个阶段。一个阶段可为下一阶段修改输入，或者仅生成表格。

**peephole optimization（窥孔优化）**

一种优化技术，它仅检查相连的几条指令，以确定它们是否可由一组已改进的指令替换。

**PIC（Position Independent Code，位置独立代码）**

适合用于共享库的代码，因为它可被保存在内存中的任何一个位置并从那里开始执行。所有的内部地址都是相对于内部偏移，或是对全局表的引用。见 GOT。

**platform（平台）**

某种特定计算机硬件和操作系统的组合。编译程序必须被配置为可在某特定的操作系统和某特定的硬件计算机上运行。平台也称为主机或目标机。

**PMF（Pointer to Member Function，成员函数指针）**

C++中一种特殊数据类型，它可以保存到某个对象的成员函数的地址。

**POSIX**

由 IEEE 标准和 Open Group 的 Single UNIX 规范融和的 UNIX 标准规范。

**pragma**

一种在源代码中插入的命令，它是和编译程序相关的信息，并可由不理解该信息的编译程序忽略。

**preelaborable（可预阐述）**

如果 Ada 编译单元被描述为不需要其他单元提供信息，就以一种独立方式进行描述并称其为可预阐述的。参见详细阐述（elaborate）。

**prefetch（预取）**

当计算机从程序读取指令并将其加载到 CPU 中准备执行时，这一过程称为指令的读取。许多 CPU 都会同时加载若干条指令，这称为预取。

**preprocessor（预处理程序）**

一种文本处理器，它可以读取程序源代码并用数值（或其他名字）替换名字，使用文本替换扩展宏，以及评估表达式以确定哪些代码可以被去除。

**pseudo-op（伪操作）**

参见指示字。

**Ratfor（Rational Fortran）**

一种可供公共使用的源代码预处理程序，它允许按类似 C 的语法书写的 Fortran 可被转换为标准 Fortran。

**relative address（相对地址）**

相对地址是到某个已知地址的偏移。这种类型的寻址通常用于可重定位的模块（共享库），因为只有模块最高端的位置才是加载到内存中的可执行代码所需要的惟一信息。参见绝对地址。

**relocatable（可重定位的）**

参见相对地址。

**RM**

对 Ada 来说，这是参考手册（reference manual）的简写，它是定义 Ada 95 标准的文档。

**RTL（Register Transfer Language，寄存器传送语言）**

由某语言源代码所生成的内部代码，并可由此输出汇编语言。在 RTL 格式下，仍可能执行优化和其他操作。

**RTTI（Runtime Type Identification，运行时类型标识）**

在面向对象编程中，可以将一种类型的对象当做另一种使用。可以利用 RTTI 工具在

运行时确定对象的真实类型。

**runtime（运行时）**

也称作运行时软件包，由于它们在应用程序运行时会被调用，运行时是函数的集合，它同编译程序一同发行并链接到被编译的程序。

**rvalue（右值）**

任何可以得到一个数值的表达式。该名词源于右值（right value），是指在赋值语句中右边的表达式所计算出的数值。参见左值。

**scheduler（调度器）**

如果机器能够一次执行多条指令，指令可以被调整（调度），这样一些较快的指令就可以与一条较慢的指令同时执行。

**scope（作用域）**

一个定义可以被识别的区域。在 C 中，函数定义的变量的作用域是整个函数。定义在由括号括起的代码块中的变量的作用域是那个代码块，在函数外定义的变量可以由很多函数引用。

**semantics（语义）**

一种编程语言的语句的含义。语句的实际含义依赖于它的上下文。例如，表达式 a+b 可以是整数加、浮点加，甚至是字符串连接。确定其含义的过程称为语义分析。

**serialize（串行化）**

在 Java 中，一个对象可以被转换为字符串并传递（或存储留作后用），之后它还可以被转化回执行对象。参见 marshaling。

**sequence point（顺序点）**

程序执行中的一点，在这一点所有的计算全部完成而且变量均包含正确的结果。在这一点可以开始计算新表达式。某些优化技术会修改和改变两个顺序点之间的操作顺序。

**shared library（共享库）**

参见库。

**sibling call（同属调用）**

参见尾调用。

**side effect（副作用）**

一种会修改内存或文件中数据的操作（如函数调用或算术表达式）。这些变化称之为副作用操作。大多数副作用都是有意的并且是程序执行所必要的，但有些情况下副作用是有害的。

**slot scheduler（槽调度器）**

参见调度器。

**Single UNIX Specification**

参见 POSIX。

**Smalltalk**

一种在 20 世纪 70 年代 Xerox PARC 上开发的面向对象语言。Smalltalk 是 SIMULA 界面系统的编程语言，该系统引入了鼠标和窗口系统。

**spec file（spec 文件）**

一种包含一套规则的文件，这些规则可以控制哪些参数需由 gcc 传递给每一个子进程，以及可接受参数的格式。

**STABS（Symbol TABle directiveS，符号表命令）**

插入到汇编语言源代码中的操作子和数据位置，以提供调试信息。随后汇编器和连接程序可在目标代码和可执行程序中包含这些表格，以达到调试的目的。

**static library（静态库）**

参见库。

**SSA（Static Single Assignment，静态单赋值）**

通过代码块代表逻辑流的一种特殊形式。SSA 可用于某种特定类型的优化，例如不可达代码的删除。SSA 的一个优点就是能够跟踪寄存器保存的值。

**stack frame（堆栈框架）**

栈中保存局部变量以及当前函数寄存器值的区域。确切的格式依赖于处理器以及所使用的函数调用规范。

**static link（静态链接）**

静态链接的可执行程序就是包含了所有需要函数的程序。当程序被链接时，由运行时

库中取出函数并复制到被链接的程序中。参见共享库。

**static single assignment**

参见 SSA。

**stringize（字串化）**

预处理程序采取的动作，可将宏参数转化为一个引用的字串，而非简单地将其插入到源码中。

**strip**

strip 命令可以用来去除可执行文件中的所有调试信息。根据编译到可执行程序中的调试信息的多少，缩减的尺寸可能是非常惊人的。

**stderr（Standard Error，标准错误）**

每个 UNIX 程序在开始执行时均打开该输出流，默认情况下定向到终端上。通常该流可用来输出错误消息。

**stdin（Standard Input，标准输入）**

每个 UNIX 程序在开始运行时均打开该输入流，默认情况下会从键盘中读取所敲击的内容。

**stdout（Standard Output，标准输出）**

每个 UNIX 程序在开始执行时均打开的输出流，默认情况下定向到终端上。通常该流可以用来进行输出。

**strength reduction（强度弱化）**

一种优化技术，它可以用代价较低的操作取代代价高的操作，例如用加法或移位操作取代乘法操作。

**stub**

一种本地函数，当它被本地程序调用时，会将调用信息打包，并将调用发送给真正的函数，即位于另一台计算机上的函数。

**subexpression（子表达式）**

表达式的一部分。例如，在表达式 a * (b + c)中，因子(b + c)是一个子表达式。一个子表达式可以非常复杂，也可以简单到例如向寄存器加载地址。

**SVR4（System Five Release Four，系统 5 发行 4）**

由 AT&T 制造的 UNIX 版本。SVR4 是许多现代 UNIX 版本的基础。参见 BSD。

**syntax（句法）**

编程语言的物理结构。语法通过强制的规则确定了语言的形式，这些规则可以确定语言自身文本的合法顺序和结构。

**tail call（尾调用）**

如果函数最后一条语句递归地调用自身，这一逻辑可以变成一个循环而非调用，这样可以节省栈空间。同样的优化技术可以用来在两个或多个函数间进行递归的相互调用——这种优化也叫同属调用。

**target（目标）**

参见平台。

**text（文本）**

由 UNIX 连接程序产生的可执行文件中包含的可执行代码段。参见 bss 和 data。

**thunk（块）**

用来产生地址的一块代码。由 ALGOL 60 引入，通过名字和产生代码来传递参数，它生成的代码可以在运行时解析地址，现在块也用来专指用一列新地址覆盖旧地址的动作。

**time slice（时间片）**

操作系统允许进程在其停止并由另一个进程使用时间片前执行的时间量。

**token（权标）**

参见词法分析。

**translation unit（翻译单元）**

编译单元的另一种名称。

**translator（翻译器）**

编译程序的另一个名字，这是因为编译程序的工作就是将源码翻译为可执行代码。见编译程序。

### trap（陷阱）

一种发送给正在运行的程序的硬件信号，它用来指出硬件检测到的程序执行错误。这包括诸如浮点除法除数为0，以及非法的内存地址等。

### trigraph（三字母序列）

标准C的三字母序列，可以转换为单个字符。它被设计用来在不支持ASCII字符全集的计算机上使用C编程。有9个三字母序列：

```
??=  #        ??(  [        ??  {
??/  \        ??)  ]        ??  }
??'  ^        ??!  |        ??-  ~
```

### unalias（去别名）

去除所有对内存的调用，但一种可能的到内存位置的引用就是为该单元去别名。见别名。

### unordered comparison（无序比较）

两个浮点数间的有序比较不会在其中一个值为NaN时产生异常。参见有序比较。

### unroll（循环展开）

为了优化代码，足够小并且具有固定循环次数的循环可以通过复制代码相应的次数被展开，并且循环可以被删除。

### UTF-8（Unicode Transformation Format，Unicode传输格式）

也写作UTF8，Utf8或UTF。用这种方式编码的Unicode与8位存储的ASCII字符相似，即ASCII文本文件可被当作就是UTF-8编码的。

### vague linkage（模糊连接）

目标文件包含的信息可用于链接和运行程序，但除了数据以外的其他内容也需要解析地址引用。一个例子就是C++中的虚函数。

### variadic macro（变长宏）

变长宏就是具有可变参数个数的宏。GCC预处理程序具备对可变参数宏进行扩展的能力，这种宏会将参数列表的文本保存在名为__VA_ARGS__的变量中。

### vector（向量）

一组具有相同类型和尺寸的数据项的连续集合。

**volatile（易变的）**

内存中的某单元可能会在其他地方被修改（没有常规的途径），这被称为易变的。这对编译程序很重要，因为将数据保存在寄存器比每次在需要时从内存加载效率更高。

**vtable（virtual function table，虚函数表）**

面向对象语言的对象维护内部表，称为 vtables，它包含函数的地址。通过替换这些地址，子类可以重载并替代父类中选中的函数。

**VXT**

Fortran 的一种变体，与 VAX Fortran 非常相似，在某种程度上与 Fortran 90 也非常相似。

**weak alias（弱别名）**

与弱符号相同。

**weak symbol（弱符号）**

具有两个或多个相同名字的全局符号时，只要除一个之外的其他全部符号均声明为弱符号就不会产生冲突。连接程序将忽略弱符号的定义，并使用正常的全局符号定义解析所有的引用，但只有全局符号不可用时才会使用弱符号。一个弱符号可被用来命名可以由用户代码重载的函数和数据。弱符号也称为弱别名，或简单叫做弱。

**width（宽度）**

术语宽度通常指基本数据类型的相对尺寸。例如，在 C 中，char 可能需要 1 个字节，short 可能需要 2 个字节。在这种情况下，称 short 比 char 宽。

**word（字）**

本地机器整数的尺寸。在 16 位机器中，字是 16 位的，在 32 位机器中，字是 32 位的。

**whitespace（空白）**

空白字符就是通常情况下不可见的字符，并会在编译现代自由格式语言时作为输入被忽略。它们通常是空格、制表符、竖直制表符、进纸符、换行符和回车符。

**XCOFF（Extended Common Object File Format，扩展的通用目标文件格式）**

目标文件的一种标准格式，它具有跨平台移植性，且可以适用于不同的汇编器和连接程序。参见 COFF 和 ECOFF。