第一章 概 述

1·1 引言

这一卷《程序员指南》介绍 UNIX 系统用于创建、维护和扩展 C 程序的工具。如在本书开始"目的"一节中已经指出的，我们不打算教给你怎样用 C 编程序。这里假定你知道怎样做，或者眼下正在学习应该怎样做。

我们既不可能全面介绍由 C 编译系统提供的所有工具，也不可能全面探讨所介绍工具的每一个方面。《程序员参考手册》正是用来介绍这两个方面的。而本指南用来说明和提供一些例子，告诉用户怎样使用其中最重要的工具，同时提供怎样把它们装配到一起的相关描述。另外，本书还选入了一些我们认为多数 C 程序员将发现是极为宝贵的，但不符合参考手册格式的材料。"C 编译程序诊断"那一章是表达这种思想的很好例子。

那末，应该怎样读这本《指南》呢？如果你没有写 C 程序的经验，你可能希望顺序地阅读它，所以，我们已经尽可能地根据功能分类来编排这些工具。同时，你也可能希望有选择地阅读本书。我们不希望每个人都从头至尾读完全部 400 页左右的编译程序诊断，也不希望一个非专业的程序员去读有关目标文件的那一整章。下一节"内容提要"将告诉你如何阅读本书。该节中介绍了《指南》中涉及到的程序设计支持工具，并概述了它们之间的关系。在转入下节之前，我们想先介绍一些 C 编译和 C 语言方面的基础知识。

1·2 C 编译

我们在本书这几页中讨论的最重要的工具是 C 编译系统，它把用户的 C 源程序代码翻译为运行该程序的那台计算机的机器指令——换句话说，就是编译它。在 UNIX 操作系统上，实现这一任务的命令是 cc：

```
$ cc mycode.c
```

若你的程序放在多个源码文件中，则命令为：

```
$ cc file1.c file2.c file3.c
```

等等。如同例子所示，被编译的源码文件必须取以字符.c 结尾的名字。

在上述两个命令行中有一些隐含的参数，关于这些参数你需要读描述 C 编译系统的第二章。现在，知道这两条命令就足以创建一个可执行程序，放在你的当前目录下一个称为 a.out 的文件中。其中第二条命令还将在你的当前目录下创建与每个源文件对应的目标文件：

```
$ ls -l
a.out
file1.c
file1.o
file2.c
file2.o
file3.c
file3.o
```

每一个.o 文件中含有对应源码文件中 C 语言代码的二进制表示。cc 命令创建并连接这些目标文件，以产生可执行的目标文件 a.out。你在程序中调用的标准 C 库函数——例如，printf( )——运行时自动地与可执行程序连接。当然，你也能够使用在第二章讲述的 cc 命令行任选项，取消这些默认的安排。在下面"内容提要"一节中，我们将更正式地讨论 link 编辑。在下一节，我们还将看看一些库。

在系统提示符后键入程序的名字，就可执行该程序：

```
$ a.out
```

因为名字 a.out 仅仅是暂时有效的，所以你可能需要重新为可执行程序命名：

```
$ mv a.out myprog
```

编译的时候，通过 cc 命令行的选项，你也可以为自己的程序指定另外一个名字：

```
$ cc -o myprog file1.c file2.c file3.c
```

同样，这里我们也通过在提示符后键入程序的名字来执行它：

```
$ myprog
```

1.3 C 语言

UNIX 系统支持多种程序设计语言，而 C 编译程序在许多其它操作系统上也可以使用。尽管如此，UNIX 系统和 C 语言之间已经建立并保持着非常紧密的关系。C 语言在 UNIX 操作系统上开发成功，并用来为 UNIX 系统内核编码。大多数 UNIX 应用程序是用 C 语言写成。

第三章提供了关于 C 语言完整的参考指南。下面是该语言的一些特点：

- 基本数据类型：字符，各种长度的整数和浮点数；
- 导出数据类型：函数，数组，指针，结构和联合；
- 丰富的操作符，包括按位操作符；

■ 控制流程：if，if-else，switch，while，do while，以及for语句。

用 C 编写的应用程序通常可以容易地移植到其它机器上。依照 ANSI 标准 C(符合美国国家标准研究所制定的标准)编写的程序享有更高级别的可移植性。

那些需要与 UNIX 系统核心——用于低级 I/O、内存管理、进程之间通信和类似功能— 直接会话的程序，能够通过调用标准 C 库中的系统函数，高效率地用 C 写成。标准 C 库在《程序员参考手册》第二节中描述。

1·3·1 用 C 进行模块化程序设计

C 是一种适用于进行模块化程序设计的语言。在 C 中依据函数思考是很自然的；同时，由于 C 程序中的函数能够分别编译，因此下一个逻辑步骤就是把每一个函数，或一组相关的函数存放在不同的文件中；然后，每个文件可以作为你自己程序的一个成分，或是一个模块进行处理。

第三章讲述的是，怎样写 C 代码才能使用户程序的各个模块能够彼此通信。这里我们想强调的是，分成小块编写程序使得修改工作容易进行，因为你仅需要重新编译修正过的模块。还使得用已经写好的代码建立程序更加便利：当你为一个程序写函数时，你一定会发现，许多函数可以被其它程序采用。

1·3·2 库和前导文件

由 C 编译系统提供的标准库中含有许多函数，你可以在自己的程序中用它们实现输入/输出，字符串处理以及其它高级操作，C 语言中没有明显地提供这些函数。前导文件中含有定义和声明，你的程序调用库函数时将需要它们。例如，实现标准 I/O 的函数使用前导文件 stdio.h 中的定义和声明。当你在程序中用到下面一行

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

时，就保证了你的程序与标准 I/O 库之间的接口同建立库所使用的接口一致。

第二章描述了一些较为重要的标准库，并列出了当你调用那些库中的函数时，必须在自己程序中蕴含的前导文件。它还教给你怎样在自己的程序中使用库函数，以及怎样蕴含前导文件。当然，按照第二章描述的模块化程序设计示例，你可以创建自己的库和前导文件。

1·3·3 C 程序怎样与 shell 通信

信息或控制数据可以作为命令行上的实参传递给 C 程序，也就是说，通过 shell。例如，我们已经知道，怎样调用 cc 命令，其实参为源文件名：

```
$ cc file1.c file2.c file3.c
```

当你执行 C 程序时，命令行实参通过两个参数传递给 main()函数，一个实参计数，

一般称为 argc，一个实参向量，一般称为 argv(要求每一个 C 程序有一个名为 main 的入口点)。argc 是被调用程序带有的实参个数；argv 是一个指针数组，指向包含实参的字符串，每个字符串有一个指针。由于命令名本身被看作第一个参数，或 argv[0]，所以计数至少总是 1。下面对于 main( )的声明：

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

关于在用户程序中怎样使用这两个参数的两个例子，参见第二章最后一个子节。

shell，它使得实参可以由用户程序使用，认为实参是任何一个非空白字符的序列。用单引号括起的字符('abc def')或用双引号括起的字符("abc def")作为一个实参传送给程序，即使字符中有空白符和制表符也无妨。用户负责出错检查，还要确认所接受的实参确实是自己的程序所希望的参数。

除了 argc 和 argv，你可以使用第二个实参：envp 是一个关于环境变量的指针数组。在《程序员参考手册》第二节 exec 和第五节 environ 之中，你可以更详的了解 envp。

C 程序可以随意地退出，通过从 main( )返回或调用 exit( )函数把控制送还给操作系统。这就是说，main( )中的 return(n)等价于 exit(n)调用(记住，main( )也符合"函数返回整数")。

用户程序应该返回一个值给操作系统，说明它是成功地完成还是相反。所得到的值传送给 shell，如果用户在前台执行自己的程序，它变为 shell 变量 $? 的值。根据约定，零返回值表示成功，而非零返回值意味着发生了某类错误。你可以使用在前导文件 stdlib.h 中定义的宏 EXIT_SUCCESS 和 EXIT_FAILURE 作为 main( )的返回值，或作为对于 exit( )的实参值。

1·4 内容提要

本节按五个功能分类概括本《指南》涉及的程序设计支持工具：

- 创建可执行程序
- 程序分析
- 程序管理
- 程序开发
- 高级程序设计公用程序

斜体字印刷的注释对使用这些工具的典型方法作了提示。

除了这里讨论的那些章以外，本《指南》还包含一些附录，分别为使用关键字 asm 的汇编语言转义，映象文件，以及把目标文件输入部分映射为可执行文件输出段的设施等等。它还包含有词汇表和索引。

1·4·1 创建可执行程序

第二章讨论 C 编译系统，那是一组软件工具，用户使用它们把 C 语言源文件生成可执行程序，其中含有初学者和熟练的程序员都感兴趣的材料。

第一节"编译和连接"详细阐述了命令行的语法，它用来产生程序的二进制表示——一个可执行的目标文件。前面我们曾提到，C 程序的模块可以彼此通信。例如，在一个源文件中说明的符号可以在另一个源文件中定义。连接编辑指的是那样一个过程，借助它，第一个文件中引用的符号与在第二个文件中的定义连接起来。通过 cc 命令的命令行任选项，用户可以选择两种连接编辑方式中的任一种：

- 静态连接，其中外部引用在执行前解决。
- 动态连接，其中外部引用在执行期间解决。

除了许多其它内容之外，"编译和连接"还描述了那些任选项，它们使连接编辑程序的行为更适合于用户的需要。其中还有对每一种方式优缺点的讨论。主要的差别在于，动态连接允许库代码在运行时由不同的程序共享——同时被使用。另一点差别是，动态连接的代码可以修改和改进，而不必重新连接依赖于它的应用程序。

该章第二节"库和前导文件"集中在标准 C 库上，特别是那些用户作为标准 I/O 使用的函数。本节中还讨论了数学库和 libgen 库，当你调用这些库中的函数时，需要在自己的程序中蕴含的那些前导文件，也在这一节中列出。

使用 cc 命令和它的任选项控制编译过程，先根据源码文件创建目标文件，然后，把它们彼此连接并与程序中调用的库函数连接在一起。

如已指出的，第三章提供了 C 语言的参考指南，它讲的是由 C 编译系统接受的语言。第四章列出了由 C 编译程序产生的告诫和出错信息。当你需要澄清自己对在语言那一章概括的语法和语义规则的理解时，请核实在编译程序诊断那一章给出的例子。在很多情况下它们将是有帮助的。

1·4·2 程序分析

第五章论述的 lint 程序检查那些可能使你的 C 程序不能编译，或者执行时产生不可预测的结果的代码结构。lint 发布由 C 编译程序产生的每一个出错和告诫消息。它还发布"lint 特有的"告诫消息，有关定义的不一致性，使用交叉文件以及潜在的可移植性问题等。这一章中有这些告诫消息表，并附有引起这些告诫的源码例子。

sdb 代表"符号排错"，它意味着你可以使用程序中符号的名字极精确地定位到发生问题的地点。你可以在 sdb 控制之下运行自己的程序，看看到达程序失败的那点时，程序正在做什么。另外，你可以使用 sdb 从头至尾仔细检查由失败程序留下的内存映象。这样能够核实在失败的那一时刻程序的状态，从而有可能暴露隐藏的问题。第六章是对于 sdb 的指导。

使用 lint 核实用户程序的可移植性和交叉文件的一致性，以保证程序能进行编译。

使用 sdb 确定缺陷的位置。

梗概程序是分析用户程序动态行为的工具：每部分程序代码执行的速度和频率如何。

- prof是时间梗概程序。它报告花费的时间总数和执行每一部分程序花费时间的百分比。它还报告调用每一个函数的次数和每次调用的平均执行时间。
- lprof是行频率梗概程序。它报告每一行C源代码执行的次数。这样，它使你能确定未执行的和最频繁执行的代码部分。

《指南》的第七章详细地讨论了 lprof 程序。其中包括 C 梗概公用程序的综述，说明为了用其中任一个工具梗概一个程序，用户必须遵循的过程。

cscope 浏览程序是一个交互型的程序，它在 C、lex 或 yacc 源码文件中定位指定的代码成分。如果你愿意，它使你能够用比普通的编辑程序更有效地搜索、编辑你的源码文件。那是因为，它知道函数调用(函数什么时间被调用，什么时间调用别人)，还知道 C 语言标识符和关键字。第八章是 cscope 浏览程序指导。

用 prof 和 lprof 找出，并用 cscope 改写效率低的代码行。

cscope 可以用于任何其它程序编辑任务。

1·4·3 程序管理

许多 UNIX 系统的工具用来使 C 程序的管理更为方便。第九章的 make 用来跟踪程序模块之间依赖关系，所以当一个模块改变时，依赖于它的每一个模块都被取出进行更新。make 读取一个规格说明，该说明描述用户程序模块怎样彼此依赖，以及当其中一个被改变时，需要做些什么。当 make 发现已经改变的成分比依赖于它的模块日期更近时，指定的命令——典型地是重新编译相关的模块——传送给 shell 去执行。

源代码控制系统 SCCS 是一组程序，用户可以用来跟踪文件的进化版本，包括普通正文文件以及源码文件。当文件已经放在 SCCS 控制之下时，用户可以规定，每次只能对文件任一版本的一个拷贝检索出来进行编辑。当被编辑的文件返回到 SCCS 时，改变被记录下来。这使得有可能对更改进行审计并重新构造文件的更早版本。第十章讨论 SCCS。

对任何有多个文件的程序使用 make。使用 SCCS 以跟踪程序的版本。

1·4·4 程序开发

设计了两个 UNIX 系统的工具使得建立 C 程序更加方便。第十一章的 lex，和第十二章的 yacc，用来生成 C 语言模块，它们可能是一个更大的应用程序的有用成分，事实上也是任何一类需要对系统性输入进行识别和操作的应用程序的有用成分。

lex 生成一个用来执行输入流的词法分析的 C 语言模块。词法分析器扫描输入流寻找与用户规定的正则表达式相匹配的字符序列——单词，当找到一个单词时，执行由用户规定的一个动作。

yacc 生成一个 C 语言模块，它对已经由词法分析器传送给它的单词实行语法分析。

语法分析器根据用户规定的规则描述单词的语法形式。当找出某一个语法形式时，也取得了动作，它同样由用户规定。词法分析器不一定必须用 lex 生成。你能够稍加努力，用 C 写出它。

使用 lex 和 yacc 分别生成用户界面的词法分析器和语法分析器。

1·4·5 高级编程公用程序

第十二章"目标文件"描述了由 C 编译系统产生的目标代码的可执行和连接格式(ELF)。严格地说，仅仅是那些需要访问和操纵目标文件的程序员需要阅读这一章。还有，因为它提供了有关编译系统工作更广泛深入的观察，特别是动态连接机制，可以证明这对于那些，试图拓宽自己对前几章所介绍材料的理解的读者，是很有帮助的。

第十四章"浮点操作"详细介绍了由 C 编译程序生成的浮点运算的标准单、双精度数据的类型、操作和转换。其中还说明了提供给程序员的低级库函数，他们需要最大范围的浮点支持。大多数用户不需要在程序中调用用于浮点操作的低级函数。想要做这件事的用户可以在第十四章找到所需要的信息。

第十五章描述通用宏处理器 **m4**，它可以用来预处理 C 和汇编语言程序。

1·5 其它工具

本节节列出那些在《指南》中没有得到充分论述的编程支持工具。参阅有关上下文中涉及这些工具的索引，以及《程序员参考手册》第一节关于详细用法。

分析源代码的工具：

- **cflow**产生C, lex,yacc和汇编语言文件中外部引用的图表。使用它核实程序的依赖关系。
- **ctrace**在程序的每一个语句执行时，打印出变量。使用它可以跟踪C程序一个一个语句的执行。
- **cxref**分析一组C源码文件，建立每一个文件中关于自动、静态和全局符号的交叉引用表。使用它核实程序的依赖关系，并展示程序的结构。

阅读和操纵目标文件的工具：

- **cofzelf**把按照通常目标文件格式(COFF)的目标文件，翻译为可执行和连接格式(ELF)的目标文件。
- **dis**分解目标文件
- **dump**卸出选定的部分目标文件。
- **lorder**生成一个目标文件的有序表。
- **mcs**操纵一个目标文件的段。
- **nm**打印目标文件的符号表。
- **size**报告一个目标文件的所有段或可装入段的字节数。
- **strip**从目标文件中删除符号排错信息和符号表。

第二章　C 编译系统

2·1　引言

本章描述那些 UNIX 系统工具，用户使用它们把 C 语言源码文件生成一个可执行的程序。

第一节"编译和连接"详细叙述了命令行的语法，用户使用这个命令产生程序的二进制表示——一个可执行的目标文件。本节主要讲述 cc 命令行的任选项，它控制着下面的过程：首先根据源码文件创建目标文件；然后把它们彼此连接，并与用户程序中调用的那些库函数连接。如我们在第一章中指出的，本节的重点是静态 vs 和动态连接；每一种方式怎样实现和调用，以及它的相对优缺点。

本章第二节"库和前导函数"重点在于标准库。例如，由于 C 语言自身没有输入输出设施，I/O 必须通过显式地调用函数来实现。在 UNIX 系统中，实现 I/O 和其它高层任务的函数已经标准化并分类放在库中，它们方便，可移植，并且在多数情况下对你的机器总是适用的。本章最后还讲述了一些重要的标准库的内容。

前导文件包含定义和声明，它们作为用户程序和这些库中函数的接口。前导文件还包含一些所谓的"函数"——例如，getc( ) 和 putc( )——它们实际上定义为宏。(手册页一般将告诉你，究竟你使用的是一个宏，还是一个函数。事实上，你在自己程序中以同样的方法使用它们，基本上没有差别。)本章中标准库的说明，指出了那些当你调用这些库中的函数时，需要在自己的程序中蕴含的前导文件；每一个函数的手册页中也列出了所需的前导文件。本章最后一节，我们将教给你怎样在自己的程序中使用库函数以及怎样蕴含前导文件。我们将特别注意标准 I/O。

2·2　编译和连接

C 编译系统由编译程序、汇编程序和连接编辑程序组成。如果用户没有使用命令行任选项另外指定，cc 命令将自动地调用上述每一个成员。在转向讨论 cc 命令行的语法之前，让我们简短地查看一下创建一个可执行 C 程序的四个步骤：

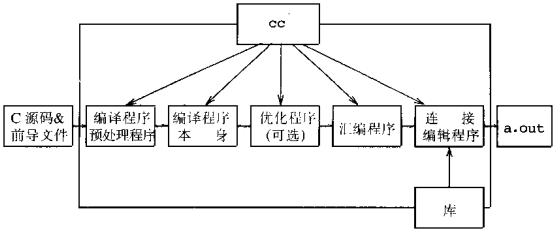
1. 编译程序的预处理程序逐行地读用户源码文件，着眼于替换带有单词字符串(#define)的名字，这也许是有条件地替换(例如#if)。它还接受用户源文件中的指令，以便包含用户程序中命名文件的内容(#include)。我们在本章第 2 节中将看到，被蕴含的大部分前导文件，由#define 指令、外部符号的声明以及那些你希望使得更多源文件可用的定义和声明组成。

2. 编译程序把用户源码文件中的C语言代码(现在文件中含有的是每一个被覆含的前导文件经预处理后的内容)翻译为汇编语言代码。
3. 汇编程序把汇编语言代码翻译为目标机的机器指令。如我们在第一章中指出的，这些指令保存在与每个用户源码文件对应的目标文件中。换句话说，每一个目标文件含有对应源文件中C语言代码的二进制表示。目标文件由段组成，通常至少有两段：正文段主要由程序指令组成，正文段一般有读和执行权限，但没有写权限；数据段一般有读、写和执行权限。参见第十三章有关目标文件格式的详细描述。
4. 连接编辑程序把这些目标文件彼此相连接，然后与用户程序中调用的每一个库函数连接，尽管当它和库函数连接时依赖于用户选定的连接编辑方式：
   - 一个档案库或静态连接库是一个目标文件的集合，其中每个文件含有库中的一个函数或一组相关函数的代码。当你在程序中使用一个库函数，并在cc命令行中规定了一个静态连接选项时，该函数的目标文件的拷贝在连接时加入你的可执行程序中。
   - 一个共享目标或动态连接库是一个单一的目标文件，它含有库中每一个函数的代码。当你在程序中调用一个函数，并在cc命令行中规定了一个动态连接选项时，整个共享目标的内容映射到你的进程运行时的虚地址空间中。正如它的名字所意味的那样，在运行时一个共享目标包含的代码能够同时由不同的程序使用。

在"连接编辑"一节中，我们将讨论这两种实现库的方法。我们还将教给你如何根据自己的需要，用不同的方法把静态和动态这两种连接方法结合起来。

图2—1给出了C编译系统的组织。注意，我们在这里省略了优化程序的讨论，因为它是可有可无的。在后面"常用的cc命令行任选项"一节中，我们将告诉你怎样调用它。

图 2-1 C 编译系统的组织



2.2.1 基本的 cc 命令行语法

现在让我们看一看，对于打印单词 hello、world 的 C 语言程序，这个进程是怎样工作的。这里是程序的源代码，我们已经把它写在文件 hello.c 中：

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

我们在第一章已经指出，将 C 语言源码文件编译成一个可执行程序的 UNIX 系统命令是 cc：

```
$ cc hello.c
```

我们在第一章中还指出，被编译的源码文件必须取以字符.c 结尾的名字。文件名的其它部分可以任意取名。

由于上述源代码中没有任何语法和语义错误，上面的命令将在当前目录下，创建一个可执行的程序存放在文件 a.out 中：

```
$ ls -l
a.out
```

```
hello.c
```

注意，当你编译单个的源文件时，不创建.o 文件。
在系统提示符后写入程序的名字可以执行该程序：

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

由于名字 a.out 仅是暂时有效的，我们将重新命名可执行程序：

```
$ mv a.out hello
```

在编译该程序的时候，也可以对 cc 命令行使用-o 选项来给出该程序的名字 hello。

```
$ cc -o hello hello.c
```

在以上两种情况下，我们都可以通过在系统提示符后写入它的名字来执行这个程序：

```
$ hello
hello, world
```

现在我们来看一看，cc 命令怎样控制我们在前面一节讲述过的四步处理过程。当我们对 cc 规定-P 选项时，仅调用编译程序的预处理程序：

```
$ cc -P hello.c
```

预处理程序的输出——源代码加上被预加工的 stdio.h 的内容 ——放在当前目录下的 hello.i 文件中：

```
$ ls -l
hello.c
hello.i
```

输出可能是有用处的，例如，若你接收到一条编译程序出错消息，它是关于在下面的源代码段中未定义的符号 a 的消息：

```
if (i > 4)
{
        /* declaration follows
        int a; /* end of declaration */
        a=4;
```

```
}
```

第二行上未终止的注释将使得编译程序把跟在它后面的声明当作注释的一部分，由于预处理程序删除注释，它的输出：

```
if (i > 4)
{
        a=4;
}
```

将清楚地表明声明中未终止的注释所产生的影响。你还能够利用预处理程序的输出检查条件编译和宏扩展的结果。

如果我们对 cc 命令使用-S 选项，则仅调用预处理程序和编译程序阶段：

```
$ cc -S hello.c
```

其输出——被编译源码的汇编语言代码——放在当前目录下的 hello.s 文件中。如果你正在写一个汇编语言例程，并希望看一看编译程序怎样完成类似的任务，那么该输出可能是非常有用的。

最后，如果我们对 cc 使用-c 选项，将调用除连接编辑以外的所有工具：

```
$ cc -c hello.c
```

其输出——程序汇编后的目标代码——放在当前目录下的目标文件 hello.o 中。当使用 make(第九章)时，你将需要这个输出。

现在，为了创建可执行的目标文件 a.out，我们只需要写入命令：

```
$ cc hello.o
```

根据默认，连接编辑程序安排那些我们在程序中调用的标准 C 库函数——printf( )——在运行时与可执行程序连接。换句话说，标准 C 库是一个共享对象，至少在我们正在描述的默认安排中是这样。

当然，上面我们描述的输出是对编译系统模块的输入，但它们不是仅有的输入。例如，连接编辑程序将提供程序前缀代码和程序后缀代码以分别执行启动和清除任务的工作。只有通过 cc 调用连接编辑程序时，这个代码才自动地与你的程序连接。那就是为什么在前面的例子中，我们使用 cc hello.o 而不是 ld hello.o。由于类似的原因，你必须使用 cc 而不是 as 来调用汇编程序：

```
$ cc hello.s
```

如我们在第一章提到的，如果你的程序在多个源文件中，编译过程基本上是相同的

(与在一个源文件中)，唯一不同的是，默认的 cc 命令行将在你的当前目录下创建目标文件以及可执行的目标文件 a.out:

```
$ cc file1.c file2.c file3.c
$ ls -l
a.out
file1.c
file1.o
file2.c
file2.o
file3.c
file3.o
```

这意味着如果你的源文件中有一个编译失败，你不需要重新编译其它文件。例如，假定在上述的命令行中，你收到了一个关于 file1.c 的编译程序出错诊断。你的当前目录看上去将是这样的:

```
$ ls -l
file1.c
file2.c
file2.o
file3.c
file3.o
```

就是说，编译继续进行，但连接被取消。假设你已经改正了那个错误，下面的命令:

```
$ cc file1.c file2.o file3.o
```

将创建目标文件 file1.o，并把它与 file2.o 和 file3.o 连接起来，产生可执行程序 a.out。正如示例表明的，C 源文件分别单独地被编译。为了创建一个可执行的程序，连接编辑程序必须把在一个源文件中符号的定义，与在另一个源文件中对它的外部引用连接起来。

最后请注意，我们已经讨论过的 cc 命令行任选项，并不都是编译程序的任选项。因为，例如，创建一个可执行程序的是连接编辑程序，-o 选项——你使用它给程序取一个不是 a.out 的名字——实际上是一个 ld 任选项，它由 cc 命令接收并传送给连接编辑程序。下面我们将看到更多这样的例子。我们提到这一点的主要原因，是使得你能够在恰当的手册页上读到有关这些任选项的说明。

2.2.2 常用的 cc 命令行任选项

在这一节中，我们将讨论 cc 命令行的任选项，它们使用户

- 为寻找被蕴含的前导文件规定搜索目录的次序；
- 为进行符号排错和梗概准备自己的程序；
- 优化自己的程序。

我们将推迟到下一节再讨论那些选项，用户使用它们把自己的程序与程序中调用的库函数连接起来。

2.2.2.1 搜索前导文件

回想上述简单程序的第一行：

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

这种形式的指令是你为了蕴含每一个由 C 编译系统提供的标准前导文件所必须采用的。尖括号(<>)告诉预处理程序在系统中存放前导文件的标准位置(通常是/usr/include 目录)搜索那个前导文件。

对于那些你保存在自己目录中的前导文件，格式是不同的：

```
#include "header.h"
```

双引号(" ")告诉预处理程序，首先在含有#include 行的文件所在的目录(通常是用户的当前目录)，然后在标准位置，搜索 header.h。

如果你的前导文件不在当前目录中，那么在 cc 命令行的要使用-I 任选项规定存有前导文件的目录的路径。例如，假设在源码文件 mycode.c 中蕴含了两个前导文件 stdio.h 和 header.h：

```
#include <stdio.h>
#include "header.h"
```

进一步假设 header.h 存放在目录../defs 中，命令：

```
$ cc -I../defs mycode.c
```

将引导预处理程序首先在当前目录，然后在../defs 目录，最后在标准位置搜索 header.h。它还将引导预处理程序首先在../defs 目录，然后在标准位置搜索 stdio.h 文件——差别是在当前目录中仅搜索那个用双引号括起来的名字。

在 cc 命令行中，你可以不止一次地规定-I 任选项。预处理程序将按它们在命令行中出现的顺序搜索指定的目录。不用说，你可以在同一命令行上，对 cc 规定多个选项：

```
$ cc -o prog -I../defs mycode.c
```

2·2·2·2 为符号排错准备程序

当你对 cc 使用-g 选项

```
$ cc -g mycode.c
```

时，就安排编译程序生成有关程序变量和语句的信息，它们将由符号排错程序 sdb(第六章)使用。提供给 sdb 的信息将使你能够使用符号排错程序追踪函数调用，显示变量的值、设置断点等等。

2·2·2·3 为梗概准备程序

为使用由 C 编译系统提供的一个梗概程序(第七章)，你必须作两件事：

1. 使用一个梗概选项编译和连接你的程序：

   对prof：`$ cc -qp mycode.c`
   对lprof：`$ cc -ql mycode.c`

2. 运行梗概后的程序：

   ```
   $ a.out
   ```

在执行结束时，有关你的程序运行时行为的数据被写进你的当前目录下的一个文件中：

对prof：mon.out
对lprof：*prog*.cnt

其中 *prog* 是被梗概的程序的名字。这两个文件梗概程序输入的。

2·2·2·4 优化用户程序

cc 命令的使用-O 选项来调用优化程序：

```
$ cc -O mycode.c
```

优化程序提高由编译程序生成的汇编语言代码的效率。它将加快目标代码的执行。当你已经结束了程序排错和梗概时使用优化程序。

2·2·3 连接编辑

注意： 由于在本节中我们试图适应最大范围的读者，所以可能提供了比许多用户为了把程序与C语言库连接所需要的更多的背景知识。如果你感兴趣的仅仅是怎样做，并且满足于单纯形式上的浏览动机和背景的描述，你可能需要跳到最后一个子节中的快速参考指南。

连接编辑指的是一个过程，其中用户程序在一个模块中引用的符号与它在另一个模块中的定义连接——更具体地说，通过该过程，把我们简单的源码文件 hello.c 中的符号 printf() 与它在标准 C 库中的定义连接。无论你选择哪一种连接编辑方式——静态的或动态的，连接编辑程序都将搜索你程序中的每一个模块，包括使用到的每一个库，以便在其它模块中寻找无定义的外部符号的定义。如果没有找到对符号的定义，连接编辑程序将根据默认报告出错，同时，可执行文件创建失败。(但是在每一种方法中，重复定义的符号的处理不同。进一步的说明参见后面"重复定义的符号"一节)。静态连接和动态连接之间的主要区别在于搜索完成以后所发生的事情：

- 在静态连接中，满足用户程序中尚未解决的外部引用的档案库目标文件的拷贝，在连接时加入用户的可执行程序。在创建可执行文件的时候，程序中的外部引用与它们的定义连接——指定内存中的地址。
- 在动态连接中，共享目标的内容被映射到运行时用户进程的虚地址空间。当程序执行时，其中的外部引用与它们的定义连接。

本节，将详细考查连接编辑过程。我们将从默认的安排开始，接着考查将用户程序与 C 编译系统提供的标准库连接的基本过程。最后，我们将讨论动态连接机制的实现，并查看有关共享库开发的一些编码准则以及维护窍门。通过讨论，我们将研究为什么用户宁可选择动态连接，而不选择静态连接的道理。概括为：

- 通过运行时共享库代码，动态连接的程序节省磁盘存储空间和系统进程内存空间。
- 动态连接的代码可以被修复和改进，而不必重新连接依赖于它的应用程序。

2·2·3·1 默认的安排

我们前面说明的默认的 cc 命令行

```
$ cc file1.c file2.c file3.c
```

将创建对应于每一个用户源文件的目标文件，并把它们彼此连接，创建一个可执行的程序。这些目标文件称为可再定位的目标文件，因为其中含有还没有与它们的定义相连接的符号引用——也就是还没有在内存中分配地址。

我们还提出，这个命令行将安排用户程序中调用的标准 C 库函数，自动地与用户的可执行程序连接。在这种默认安排中，标准 C 库是称为 libc.so 的共享目标，这意味着你调用的那些函数将在运行时与你的程序连接。(有一些例外，一些 C 库函数由于设计问题被 libc.so 遗漏。如果你在程序中使用这样一个函数，函数的代码将在连接时加入你的可执行程序。也就是说，函数仍将自动地与你的程序连接，但仅仅是静态地而不是动态地连接。)标准 C 库包含《程序员参考手册》第 2 节中说明的系统调用，以及在第 3 节 3S 和 3C 子节中说明的 C 语言函数。更详细的说明参见本章第 2 节。