第 3 章　嵌入式 Linux 开发环境

嵌入式程序设计需要采用交叉开发方式，编译器与编译器的输出文件运行于不同体系结构的计算机平台，这与本地开发有所不同。为了学习如何进行嵌入式程序设计，先要了解嵌入式程序设计的各个环节以及各环节使用的工具。在 Linux 环境下，进行程序开发主要使用以下工具：编辑器、编译器、链接器、调试器和自动编译管理工具等。正所谓“工欲善其事，必先利其器”，本章的主要内容为以上各个工具的介绍。

3.1　交叉开发环境

程序设计需要开发环境的支持。根据运行平台的不同，开发环境分为本地开发环境和交叉开发环境(Cross Development Environment)，交叉开发环境又可分为开放型以及商业型两大类。

需要交叉开发环境的支持是嵌入式应用软件开发的一个显著特点，交叉开发环境是指编译、链接和调试嵌入式应用软件的开发环境，与运行嵌入式应用软件的环境有所不同，通常采用宿主机/目标机模式。

通用计算机具有完善的人机接口界面，在其上安装必要的开发工具后即可为通用计算机本身开发程序。举例来说，如果开发开放式平台的应用程序，例如在 x86 体系结构 PC 平台用 VC++语言来设计一个游戏，完成之后游戏可以在同一台 PC 上运行。整个开发环境，包括工具链中的编辑器、编译器、链接器、调试器以及各种库，都基于 x86 体系结构，例如编译器软件就是一个基于 x86 指令集的二进制可执行文件。而且，项目完成后最终得到的产品——游戏——也运行于 x86 平台，所以这个游戏的二进制可执行文件也基于 x86 指令集，这就是通常的开发模式，使用的开发环境为本地开发环境。

但是，在嵌入式开发中，情况有所不同。一般来说，嵌入式设备的资源相对于 PC 来说十分有限，可能嵌入式设备上根本没有标准显示终端或者标准键盘，因此也就不可能在嵌入式设备上直接进行程序的编制，即嵌入式系统本身不具备自举开发能力，只能先在 PC 上完成程序的编写、编译、链接，之后把可执行程序下载到嵌入式设备上运行。读者可能会发现一个问题，嵌入式设备的处理器体系结构不同于 PC 的 x86 体系结构，二者指令集完全不同，如果仍旧用本地开发模式的工具链，得到的可执行文件基于 x86 体系结构指令集，下载到嵌入式设备上是无法运行的！解决的办法是在 PC 上安装另外一套开发环境，这个开发环境仍旧由工具链、库等各个部分组成，它们的可执行程序的二进制代码基于 x86 平台，但是用它们编译、链接出的应用程序的二进制代码基于嵌入式处理器的指令集，不能直接在 PC 上运行，需要下载到嵌入式设备中运行，具备这样功能的开发环境就称为交叉开发环境。因为开发环境中最重要的组成部分是编译器，所以有时也简称交叉开发环境为交叉编译环境。在嵌入式程序设计中，把运行交叉开发环境的 PC 称作宿主机，把嵌入式设备称为

目标机，如图 3-1 所示。

在图 3-1 中，左侧虚线框中为宿主机，一般是 PC。程序可以采用高级语言或汇编语言编写，对于前者，需要使用交叉编译器把它编译为由目标机指令集指令组成的二进制目标文件，对于后者，使用交叉汇编器把它编译为由目标机指令集指令组成的二进制目标文件。然后使用交叉链接器把二者链接为一个可执行文件，再把这个文件下载到右侧虚线框中的目标系统，也即目标机中，就可以运行了。如果运行错误，需要重新回到宿主机改写源代码，之后继续交叉编译、链接、下载、运行。这是一个循环往复的过程，直到最终代码执行无误为止。

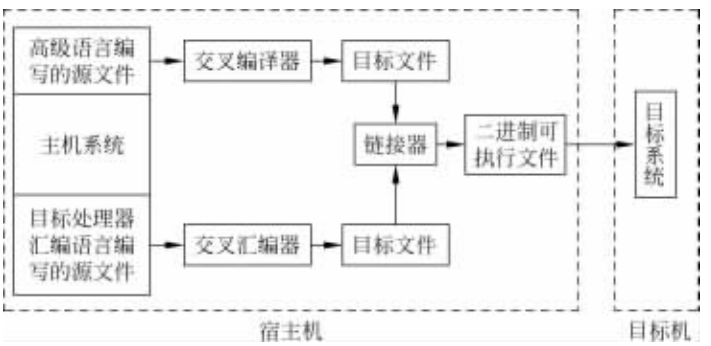

图 3-1 交叉编译

嵌入式软件开发的第一步是建立交叉开发环境，目前常用的交叉开发环境主要有开放型和商业型两种类型。

开放型交叉开发环境的典型代表是 GNU 工具链，目前已经能够支持 x86、ARM、MIPS、PowerPC 等多种处理器。商业型交叉开发环境则主要有 Metrowerks CodeWarrior、ARM Software Development Toolkit、SDS Cross Compiler、WindRiver Tornado、Microsoft Embedded Visual C++等。

在基于 ARM 体系结构的 GCC 交叉开发环境中，arm-linux-gcc 是交叉编译器，arm-linux-ld 是交叉链接器。

嵌入式 Linux 一般运行在资源紧缺的硬件平台之上，而 Linux 的 C 语言函数库 glibc 和数学函数库 libm 的功能越做越强，但体积也越来越大，很难满足嵌入式要求精简、小巧的实际需要，因此采用了它们的简化版本 uClibc、uClibm(u 应为 μ，含义同 μCLinux)和 Newlib 等。

目前嵌入式的集成开发环境都支持交叉编译和交叉链接，如 WindRiver Tornado 和 GNU 工具链等。

下面分别介绍基于 Linux 系统的开发工具链以及编辑器、编译器、链接器、调试器和自动编译管理工具等。

3.2 Linux 开发工具链

3.2.1 Linux 开发工具链简介

Linux 是自由软件的典型代表，运行于 Linux 平台的软件也遵循自由软件的规范，大部

分都可以无偿获得。编译器是所有软件中最基础的软件，有了编译器的支持，就可以编译出各种系统软件、应用软件甚至操作系统内核。GNU 开发工具链（GNU Development Toolchains）是运行于 Linux 系统的编译器集合，包括了生成一个可执行文件所需要的各个工具软件。GNU 交叉开发工具链（GNU Cross-Plateform Development Toolchains）是运行于 Linux 系统的交叉开发工具链，可以生成适合目标机指令集的可执行文件。可见 GNU 既支持本地程序开发，又支持交叉编译。

GNU 开发工具是完全的自由软件，具有常见工具链所包含的各个组件，其特点是采用命令行运行模式，功能非常强大。完整的工具链可从网站：www.gnu.org 下载。

GNU Tools 主要有以下几个部分：编译开发工具负责把源程序编译为可执行文件，调试工具负责对可执行程序进行源码或汇编级调试，软件工程工具用于协助多人开发或大型软件项目管理。具体来说，GNU Tools 包括以下几个部分。

1. GCC

GCC 是由 GNU 之父 Stallman 开发的编译器，可运行于 Linux 操作系统之上，全称为 GNU Compiler Collection，目前可以编译的语言包括：C、C++、Objective C、Fortran、Java 和 Ada，可以在其官方页面找到更加详细的信息。

GCC 是 GNU 计划的一个项目，原本只是一个 C 语言编译器，它是 GNU C Compiler 的英文缩写。随着众多自由开发者的加入和 GCC 自身的发展，如今的 GCC 已经可以支持众多语言的编译，所以 GCC 也由原来的 GNU C Compiler 演变为 GNU Compiler Collection，也即 GNU 编译器集合，并且已经被移植为多种交叉编译器。全部小写的 gcc 表示符合 ISO 标准的 C 语言编译器，它是 GCC 的一个组件。

GCC 主要包括以下组件。

（1）cpp：GNU C 编译器的预处理器。

（2）gcc：符合 ISO 标准的 C 编译器。

（3）g++：基本符合 ISO 标准的 C++编译器。

（4）gcj：GCC 的 Java 前端。

（5）gnat：GCC 的 GNU ADA 95 的前端。

2. binutils

binutils 是一组二进制工具程序集合，是辅助 GCC 的主要软件，它包括以下组件。

（1）as：GNU 汇编器。

（2）ld：GNU 链接器。

（3）ar：创建归档文件，向库中添加/提取 obj 文件。

（4）nm：列出 obj 文件中的符号。

（5）objcopy：复制和转化 obj 文件。

（6）objdump：显示对象文件的信息。

（7）ranlib：根据归档文件中内容建立索引。

（8）readelf：显示 elf 格式执行文件中的各种信息。

（9）size：显示 object 文件和执行文件各段的总大小。

（10）strings：显示文件中可以打印的字符。

（11）strip：去掉可执行文件中多余的信息，如调试信息。

(12) gprof：用来显示图表档案数据。

3. gdb

GNU 调试器称为 gdb，这是一个交互式工具，工作在命令行模式，功能十分强大。可以用来调试 C、C++和其他语言编写的程序。如加一些图形前端(如 DDD)，可以在图形环境下调试程序。

4. GNU make

GNU make 是一个用来控制可执行程序生成过程、从源码文件生成可执行程序的程序。它允许用户生成和安装软件包，而无须了解生成、安装软件包的过程。

5. diff/diff3/sidff

diff/diff3/sidff 是比较文本差异的工具，也可以用来生成补丁。

6. patch

patch 是补丁安装程序，可根据 diff 生成的补丁来更新程序。

7. 版本控制系统

版本控制系统可以帮助管理程序的历史版本，在多个程序员之间共享代码文件，甚至可以做到多个设计者同时修改同一个文件，版本控制系统负责合并修改。常见的版本控制系统有 RCS(Revision Control System)、CVS(Concurrent Version System)和 SVN(Subversion)。

对于不同的目标平台，GCC 工具的前缀各不相同，例如常见的 ARM7/μCLinux 平台的 GNU 交叉开发工具包括：

(1) arm-elf-as；

(2) arm-elf-gcc；

(3) arm-elf-g++；

(4) arm-elf-ld；

(5) arm-elf-objcopy。

对于 ARM9/Linux 平台，GNU 开发工具包括：

(1) arm-linux-as；

(2) arm-linux-gcc；

(3) arm-linux-g++；

(4) arm-linux-ld；

(5) arm-linux-objcopy。

上述软件可从网站：www.arm.linux.org.uk 下载。

3.2.2 GNU 交叉开发环境的建立

建立 GNU 交叉开发环境有两个途径：源码编译方式和直接安装二进制文件方式。前者需要下载编译器的源代码，进行配置、编译以及安装，过程较为复杂。后者可利用他人的成果，寻找他人已经编译好的编译器可执行文件压缩包，直接解压缩即可使用，缺点是对编译器集合中的各个组成部分的版本号要求较为苛刻，必须采用经验证的一系列组件才可以协调工作。

采用源码编译方式时，如果编译器只用来编译操作系统内核，那么只需要下载 binutils

和 gcc 就足够了，如果还需要交叉编译应用程序，那就还要再编译一份 glibc。编译要按顺序进行，先编译 binutils，再编译 gcc，因为在编译 gcc 的时候可能要用到前者。编译完成后要修改 PATH 环境变量，把刚编译好的工具软件的路径存放到 PATH 当中，将来就可以在任意路径下访问这些软件了。

源码编译的另一种方式是采用 crosstool shell 脚本。crosstool 是 Dan Kegel 等人开发的一套自动建立 Linux 交叉编译工具链的自由软件，支持多种 gcc、glibc 版本和多种处理器体系结构，只需经过简单配置，即可生成特定于某个处理器平台和 Linux 内核的交叉开发环境，使用非常方便。

直接安装二进制文件方式的特点是简单，非常适合初学者使用，但是这些二进制文件的寻找难度视具体的目标机而异，而且 binutils、GCC、glibc 库，以及宿主机、目标机、OS 内核之间有相互依赖关系，版本必须匹配才可以正常使用。表 3-1 是经实践检验可以协调运行的平台及各组件之间的匹配关系表。

下面以 x86 平台编译器的建立过程为例来说明如何在现有系统上利用源码编译的方式构建新版本的编译器。系统原有编译器版本是 gcc 3.2.2，现需使用此编译器编译高版本编译器源码，得到 gcc 4.9.2。注意，构建新编译器的前提是系统已经包含某个编译器，否则无法进行任何编译工作。

1. 源码准备

从 GCC 网站上(http://gcc.gnu.org/)或者通过网上搜索查找到下载资源。可供下载的文件一般有两种形式：gcc-4.9.2.tar.gz 和 gcc-4.9.2.tar.bz2，它们的区别只是压缩格式不同，内容完全一致，下载其中一种即可。下载完毕之后解压缩码包。

表 3-1　GNU Tools 与 Linux 内核关系表

主机	目标机	系统内核	binutils	Gcc	glibc	patchs
i386	i386	Linux 2.4	2.14.90	3.3.1	2.3.2	no
i386	ppc	Linux 2.4	2.10.1	2.95.3	2.2.1	no
i386	arm	Linux 2.4	2.13.90	3.2.1	2.3.1	yes
i386	mips	Linux 2.4	2.8.1	Egcs-1.1.2	2.0.6	yes
sparc	pps	Linux 2.4	2.10.1	2.95.2	2.1.3	no
ppc	arm	Linux 2.4	2.10.1	2.95.3	2.2.3	yes
i386	strongarm	Linux 2.6	2.14	3.3.3	2.3.2	yes
i386	xscale	Linux 2.6	2.14	3.3.2	2.3.2	yes

2. 配置

对源码进行配置之后才能编译，通过配置可以明确将 gcc 编译器安装到什么位置，安装 gcc 中的什么组件等选项。配置通过执行 configure 脚本来完成。命令如下：

```
# configure --prefix=/usr/local/gcc-4.9.2 --enable-threads  \
--disable-checking --enable--long-long--host=x86_84-unknown-linux-gun  \
--with-system-zlib --enable-languages=c,c++
```

出现在配置命令中的/usr/local/gcc-4.9.2 是程序的安装目录。

3. 编译、安装

在配置目录下执行 make 命令，完成对源码的编译，这个过程比较耗时。接下来执行

make install 命令，完成编译器的安装工作。修改 PATH 环境变量，添加 gcc 安装路径，然后执行 gcc -v 命令，如果能显示新版本 gcc 的版本号，说明安装成功。

对于已经编译好的工具链来说，安装过程更加简单。例如安装 ARM9/Linux 平台的 4.4.3 版本交叉编译工具链，只需如下命令操作：

```
#mkdir -p /usr/local/arm
#cd /usr/local/arm
#tar xjvf arm-linux-4.4.3.tar.bz2
```

在/usr/local/arm/4.4.3/bin 下，可以看到已经生成 arm-linux-gcc 等命令，使用-v 选项执行 arm-linux-gcc 命令，可以看到它的版本号：

```
#./arm-linux-gcc -v
Using built-in specs.
Target: arm-none-linux-gnueabiConfigured with: /opt/FriendlyARM/mini2440/build-toolschain/
working/src /gcc-4.4.3/configure --build=i386-build_redhat-linux-gnu --host=i386-build_redhat-
linux-gnu --target=arm-none-linux-gnueabi --prefix=/opt/FriendlyARM/toolschain/4.4.3
--with-sysroot=/opt/FriendlyARM/toolschain/4.4.3/arm-none-linux-gnueabi//sys-root
--enable-languages=c,c++--disable-multilib --with-arch=armv4t --with-cpu=arm920t
--with-tune=arm920t
--with-float=soft --with-pkgversion=ctng-1.6.1--disable-sjlj-exceptions
--enable-__cxa_atexit--with-gmp=/opt/FriendlyARM/toolschain/4.4.3
--with-mpfr=/opt/FriendlyARM/toolschain/4.4.3
--with-ppl=/opt/FriendlyARM/toolschain/4.4.3
--with-cloog=/opt/FriendlyARM/toolschain/4.4.3
--with-mpc=/opt/FriendlyARM/toolschain/4.4.3
--with-local-prefix=/opt/FriendlyARM/toolschain/4.4.3/arm-none-linux-gnueabi//sys-root
--disable-nls --enable-threads=posix --enable-symvers=gnu --enable-c99 --enable-long-long
--enable-target-optspace Thread model: posix gcc version 4.4.3 (ctng-1.6.1)
```

3.3 编 辑 器

vi 编辑器是最常用的文档创建和编辑工具，可用来进行文字录入、修改、删除、插入、复制、粘贴、搜索及替换等编辑工作。Linux 下文本编辑器有很多，例如图形模式的 gedit、kwrite、OpenOffice，文本模式的 vi、vim（vi 的增强版本）、Emacs。其中 vi 和 vim 是 Linux 中最常用的编辑器，包含在任何一个 Linux 发行版中，它不像图形界面编辑器那样支持鼠标操作，而是用一套键盘命令来完成各种功能，对初学者来说使用稍有不便，但在没有安装 X Window 桌面环境或桌面环境崩溃的情况下，文本模式的编辑器 vi 可能是唯一的选择。

1. 操作模式

vi 有两种基本的操作模式：命令模式和输入模式。命令模式也称指令模式，在 shell 中执行 vi 命令首先会进入命令模式，此模式下的所有按键都当作指令来处理，可进行光标移动、文本剪切、文本粘贴、文本删除、文本查找替换、文本打开、文本保存、与 shell 交互以及退出 vi 等操作。在输入模式下可进行文本录入。以下叙述如果不加特殊说明，都指在命令模式下的按键。

有多个按键可以从命令模式切换到输入模式，但是开始输入文本的位置各不相同。

(1) 按 a 键：从当前字符之后开始输入文本。

(2) 按 A 键：从光标所在行末尾输入新的文本。

(3) 按 i 键：从光标前开始插入文本。

(4) 按 I 键：从光标所在行的第一个非空格字符前开始插入文本。

(5) 按 o 键：在光标所在行下新增一行并进入输入模式，光标停留在新行行首。

(6) 按 O 键：在光标所在行上新增一行并进入输入模式，光标停留在新行行首。

按 Esc 键可以从输入模式切换到命令模式，如果不确定当前处在哪种模式下，可以多次按下 Esc 键，系统会发出嘀嘀声，提示目前处在命令模式下。

2. 进入和退出 vi

进入和退出 vi 同样有多种方法。在 shell 提示符下输入 vi 或者 vi filename 均可进入 vi，前者新建文件，后者直接编辑 filename 代表的文件。退出 vi 可以在命令模式下输入：q，然后回车。如果文件未被修改过，可以直接退出，如果文件曾被修改过，那么 vi 会先提示是否要存盘。输入：w 可以保存当前文件，然后：q 即可顺利退出，或者直接输入：q!，可以不保存当前文件强行退出 vi。输入：wq 和：x 的效果相同，均为存盘并且退出。

3. 光标移动

下述按键可完成光标移动。

(1) h：向左移动一列。

(2) j 或＋：向下移动一行。

(3) k 或－：向上移动一行。

(4) l：向右移动一列。

移动光标可以使用快捷键，nh，nj，nk，nl 分别表示按照某方向移动 n 行(列)，在有方向键的键盘上，任何模式下都可以用方向箭头来控制光标。

下面的按键可以一次移动光标多个位置，方法如下。

(1) ：n 回车：将光标移动到第 n 行。

(2) ：$ 回车或者 L：将光标移动到文本的最后一行。

(3) M：将光标移动到当前屏幕的中央行。

(4) H：将光标移动到当前屏幕文本第一行。

以下命令可以按单词移动光标。

(1) w：将光标移动到下一个单词头。

(2) b：将光标移动到前一个单词头。

(3) e：将光标移动到下一个单词尾。

上述命令可以和数字组合，一次移动若干个单位，例如：nw、nb、ne。

以下命令可以按字符移动光标。

(1) $：将光标移动到当前行末尾。

(2) ^或 0：将光标移动到当前行首。

(3) n|：将光标移动到当前行的第 n 个字符，n 为数字。

(4) fm：将光标移动到当前行的下一个字符 m 处。

以下命令可以实现屏幕翻页。

(1) Ctrl+D：下翻半屏。
(2) Ctrl+U：上翻半屏。
(3) Ctrl+F：下翻一屏。
(4) Ctrl+B：上翻一屏。
以下命令可以显示和取消显示行号。
(1) :set number：显示行号。
(2) :set nonumber：取消显示行号。

4. 删除文本

下述按键可删除文本。
(1) x：删除光标处的一个字符。
(2) dd：删除光标所在行。
(3) s：删除光标所在字符,并进入输入状态。
(4) S：删除光标所在行,并进入输入状态。
(5) dw：删除单词。
(6) D：删除从光标到行末所有字符。
(7) dfm：删除从光标到第一个字符 m 间的文本。
(8) :nd：删除第 n 行,n 为数字。
(9) :n,$d：删除从第 n 行到最后一行,n 为数字。

5. 查找和替换

下述按键可在命令模式下查找文本。
(1) /string：向当前位置的前方查找字符串 string。
(2) string：向当前位置的后方查找字符串 string。
下述按键可在命令模式下继续查找。
(1) n：沿着刚才的查找方向继续查找同一字符串。
(2) N：沿着刚才查找方向的反方向继续查找同一字符串。
(3) rm：替换当前字符为 m,替换后仍为命令模式。
(4) R：替换当前字符后的一系列字符,替换后变为输入模式。
(5) s：多个字符替换单个字符。
(6) cw：单词替换。
(7) cc：行替换。
(8) C：替换当前行剩余部分。
(9) cfm：替换当前字符到指定的字符 m。
下面举几个较复杂的例子。
(1) :s/tom/jack：把当前行的第一个 tom 换为 jack。
(2) :1,10s/tom/jack：找到第 1 行到第 10 行的所有 tom,并替换为 jack。
(3) :5,$s/tom/jack：找到第 5 行到第末行的所有 tom,并替换为 jack。
(4) :g/var/s/tom/jack：把包含 var 行中的所有 tom 替换为 jack。
(5) :s/tom/jack/g：把当前行中所有 tom 替换为 jack。
(6) :1,$s/tom/jack/g：把整个文件中的 tom 替换为 jack。

6. 复制和粘贴

(1) yy：复制当前行。

(2) dd：剪切当前行。

(3) p、P：粘贴剪切板上的内容到当前行位置。

(4) :m copy n：把第 m 行复制并粘贴到第 n 行后，m、n 为数字。

(5) :m,n copy $：把 m－n 行复制并粘贴到末行后，m、n 为数字。

(6) :.,$ copy 0：把当前行到末行复制并粘贴到文件头。

如把上述命令中的 copy 改为 move 即为移动文本块。

7. 保存文本块

下述按键可在命令模式下保存文本块。

(1) :m,n write file：把第 m 到第 n 行另存为文件 file，m、n 为数字。

(2) :n write! file：把第 n 行强行另存为文件 file，如果当前目录下已有 file 文件，则直接覆盖掉，n 为数字。

(3) :n write>>file：把第 n 行追加到文件 file 末尾，n 为数字。

8. 与 shell 交互

下述按键可在命令模式下与 shell 交互。

(1) :n read a：把文件 a 中的内容读到当前打开文件的第 n 行后，n 为数字。

(2) :! pwd：在 vi 中执行 shell 命令 pwd。

(3) :n read! pwd：在 vi 中执行 shell 命令 pwd，并把执行结果插入到第 n 行后，n 为数字。

vi 的功能远远强于 Windows 下的文本编辑器，只要熟练掌握上述命令，编辑文本的速度超过使用鼠标的可视化文本编辑器。

除了 vi，SourceInsight 也是一个面向项目开发的程序编辑器和代码浏览器。

3.4 编　译　器

源程序必须编译链接为可执行文件之后才可以运行，在程序生存周期的六个阶段中，任何一个阶段出错都要返回到编辑阶段修改源程序，然后重新进行后续阶段。GNU 编译器 gcc 通过其他软件的支持，可以完成预处理、编译、链接这三个步骤。

3.4.1 gcc 简介

作为 GCC 的一个组件，gcc 可以用来编译、链接 C/C++语言源程序，生成可执行文件。如果没有在命令行给出输出文件的名字，gcc 将生成一个名为 a.out 的可执行文件。在 Linux 系统中，可执行文件没有统一的后缀，文件是否可执行由文件属性决定，而 gcc 则通过后缀来区别输入文件的类别，表 3-2 是 gcc 所支持的文件类型。C++程序一般用 g++编译，g++和 gcc 是同一个程序，二者的区别在于：当程序中出现 using namespace std 等带有 C++特性的语句时，如果用 gcc 编译，必须显式指明这个程序要使用到 C++标准库，而g++可以直接编译。

表 3-2 gcc 支持的文件类型

后 缀 名	所支持的文件
.c	C 源程序
.C	C++源程序
.cc	C++源程序
.cxx	C++源程序
.m	Object C 源程序
.i	经过预处理的 C 源程序
.ii	经过预处理的 C++源程序
.s	汇编语言源程序
.S	汇编语言源程序
.h	头文件
.o	目标文件
.a	存档文件

gcc 是一个优秀的编译器，可以在多种硬件平台上使用，编译得到的可执行程序的执行效率与一般的编译器相比平均高出 20%～30%。以下 helloworld 代码分别用不同编译器编译，得到的可执行文件大小相差很大。g++编译得到的文件大小为 6.8KB，TC++3 编译为 7.8KB，BC4.5 编译为 53.8KB，VC++6 编译为 184KB。此外，gcc 对友元、对象的析构等方面的支持完全符合 C++标准，这是其他编译器，例如 VC++所无法比拟的。

```
#include <stdio.h>
int main()
{
    printf("hello world\n");
    return 0;
}
```

使用 gcc 编译程序时，并不是每个步骤都由 gcc 亲自完成。gcc 首先调用预处理器 cpp 对源文件进行预处理，具体工作为文件包含处理、宏替换等。接着 gcc 编译预处理之后的文件，生成以.o 为后缀的目标文件。汇编过程是针对汇编语言源程序的处理步骤，此时 gcc 调用汇编器 as，对以.S 或.s 为后缀的汇编语言源程序预处理和汇编之后，也生成以.o 为后缀的目标文件。当所有的目标文件都生成之后，gcc 调用 ld 来完成最后的关键性工作：链接。ld 负责把所有的目标文件以及程序中调用的库函数都安排到可执行程序的恰当位置，最终生成完整的可执行程序。

3.4.2 gcc 的基本用法

使用 gcc 编译器时，必须包含文件名称和一系列必要的选项。gcc 编译器的选项大约有 100 多个，其中多数平时不会用到，这里只介绍最基本、最常用的参数。gcc 命令的语法格式为：

```
gcc [options] [filenames]
```

其中 options 为编译器所需要的选项，filenames 为相关文件名称。例如：

```
#gcc -o hello hello.c
```

作用是把 hello.c 文件编译为可执行文件 hello。

也可以使用 gcc 命令来编译多源文件的程序。例如：

```
#gcc m1.c m2.c -o hello
```

会把源文件 m1.c 和 m2.c 编译链接为可执行文件 hello。

gcc 常用的选项如下。

(1) -o output_filename：指明输出文件名称为 output_filename，注意这个名称不能和源文件同名，否则就会替换掉源文件。如果不给出这个选项，gcc 默认输出文件为 a.out。

(2) -g：在可执行文件中包含符号调试工具所必需的调试信息，如需使用调试器对源代码进行调试，必须加入这个选项。

(3) -Idirname：将 dirname 所代表的目录加入到程序头文件目录列表中，预处理器根据这个选项到相应目录中寻找头文件。

(4) -Ldirname：将 dirname 所指出的目录加入到程序库文件目录列表中，使 gcc 在链接过程中能找到相应的库文件路径。

(5) -llibname：指定目标文件链接时需要的库文件名。注意，如果要链接的某个库文件名为 libabc.so，则此选项为-labc，而不需写出库的全名。

(6) -v：显示 gcc 版本号。

以下几个选项用来分阶段编译。

(1) -E：仅执行预处理。

(2) -S：由 C 源文件得到对应的汇编源文件。

(3) -c：只进行编译步骤，不链接成为可执行文件，编译器读取源代码文件，生成 .o 为后缀的目标文件，通常用于编译程序的一个模块，或者库文件。

下面几个命令能实现源程序的分步编译：

```
#gcc -E hello.c -o hello.i          //得到预处理之后的 C 源文件
#gcc -S hello.c -o hello.s          //得到汇编源文件
#gcc -c hello.c -o hello.o          //得到目标文件
#gcc hello.c -o hello               //得到可执行文件
```

gcc 提供 30 多条警告信息和 3 个警告级别，极大地方便了编程时排查错误，以下几个选项与警告信息有关。

(1) -Wall：可以显示尽可能多的警告。

(2) -Werror：可以将警告当作错误，并停止编译。

(3) -w：禁止所有的警告。

【例 3-1】 说明了 gcc 警告的用法：

```
/*   ch3_1.c   */
#include <stdio.h>
void main(void)
{
    long long a = 1;
    printf("there are three warnings!\n");
}
```

用 gcc -Wall ch3_1.c 编译时，gcc 显示三个警告信息，分别表示主函数未声明为整型、变量 var 未使用、文件末没有空行：

```
ch3_1.c:4: warning: return type of 'main' is not 'int'
ch3_1.c: In function 'main':
ch3_1.c:5: warning: unused variable 'a'
ch3_1.c:7:2: warning: no newline at end of file
```

通过观察警告信息，可以排除程序中的隐患。

gcc 支持的警告选项还有很多，如表 3-3 所示。

表 3-3 gcc 警告选项

选 项 名	作 用
-Wcomment	如果出现注释嵌套则警告(/* 后又出现/*)
-Wformat	如果传递给 printf 的参数与指定格式不匹配则警告
-Wmain	如果 main()的返回类型不是整型或者调用 main()时参数不正确则警告
-Wparentheses	根据上下文推断，如果把(n==10)写作(n=10)则警告
-Wswitch	如果 switch 中少了一个或多个 case 分支(仅对 enum 适用)则警告
-Wunused	变量声明了但未使用，或 static 类型函数未被调用则警告
-Wuninitialized	自动变量没有初始化则警告
-Wundef	如果在#if 中使用了未定义的变量做判断则警告
-Winline	函数不能被内联则警告
-Wmissing-declarations	如果定义了全局函数但却没有在任何头文件中声明则警告
-Wlong-long	使用了 long long 类型则警告

以下几个选项与程序优化有关。

(1) -O0：对程序的编译、链接不进行优化。

(2) -O/-O1：优化选项，采用这个选项，源代码会在编译、链接过程中进行优化处理，产生的可执行文件的执行效率可以提高，但是编译、链接的速度会相应变慢。

(3) -O2：包含了-O 全部的功能，以及指令调度等优化措施，编译、链接过程会更慢。

(4) -O3：包含了-O2 全部的功能，以及循环展开等优化措施，编译、链接过程最慢。

【例 3-2】 程序示例：

```
/*   ch3_2.c   */
#include <stdio.h>
int main()
{
    double i;
    double t;
    double m;
    for(i=0;i<5000.1*5000.1*5000.1/19.1+1930.5;i+=(6-4+3+1)/3.9)
    {
        m=i/2000.1;
        t=i+2.3;
    }
    return 0;
}
```

分别使用下面两个命令编译程序：

```
#gcc ch3_2.c -o a1
#gcc -O ch3_2.c -o a2
```

用 time 命令运行程序，在笔者的电脑(Intel Core i7-4710HQ，8GB 内存，VMware 虚拟机运行 Ubuntu14.04 LTS)上，得到程序执行时间：

```
#time ./a1                    //19.036 秒执行完毕
#time ./a2                    //5.248 秒执行完毕
```

由于操作系统进程调度等原因，上述两次执行程序的时间有一定的偶然性，但是也可以看出二者之间很大的差异，可见是否优化对程序执行效率影响很大。gcc 还支持许多针对特定体系结构的优化选项，以及针对特定处理器的优化选项。要注意的是，除非程序员对处理器体系结构非常了解，否则使用-O、-O2、-O3 的效果更好。

优化等级越高，编译的时间越长，在某些资源受限的场合，例如嵌入式系统中，优化可能会破坏程序的时序关系，导致程序不可用，此外，优化使跟踪调试变得困难。

gcc 还支持其他很多选项，下面这些也较常用。选项的详细情况可参阅 gcc 的手册页。

(1) -Dmacro：定义指定的宏，使它能够通过源码中的#ifdef 进行检验。

(2) -static：对库文件进行静态链接。

(3) -ANSI：支持 ANSI/ISO 语法标准，取消 GNU 所有与 ANSI 冲突的语法扩展。

(4) -pedantic：可以找到不符合 ANSI/ISO 标准的语句(并非全部)。

(5) --pedantic-errors：尽可能显示 ANSI/ISO C 标准列出的所有错误。

(6) -pipe：采用管道技术，加快程序编译。

(7) -save-temps：保留了可执行文件编译过程中的临时文件。

(8) -Q：显示编译各阶段工作需要的详细的时间。

由于 gcc 具有很多优秀特性，Internet 上的众多程序员经过努力，已经把 gcc 移植到了 Windows 平台。MinGW(Minimalist GNU on Windows)是运行于 Windows 系统之上的可视化集成开发环境，外观类似于 VC++，但采用 gcc 内核，能编译出远胜于 VC++的高质量代码。Cygwin 是运行 Windows 环境的 Linux 模拟层，支持直接在 Windows 环境运行 Linux 命令，由 Cygwin 的动态链接库(Dynamical Linking Library)负责进行两种系统之间的转换，可以在 Cygwin 之上直接运行 gcc 命令。

3.5　链　接　器

GNU 的链接器称为 ld，它负责把若干目标文件与若干库文件链接起来，并重定位它们的数据位置。在编译一个程序时，最后一步就是运行 ld 命令，通常 ld 直接由 gcc 负责调用，对用户程序员透明。ld 能接受链接描述文件的控制，这是一种用链接命令语言(Linker Script)写成的控制文件，用来在链接的整个过程中提供显式的、全局的控制。ld 比其他链接器更有用的地方在于它提供了诊断信息。许多链接器在碰到错误的时候立即放弃执行，但 ld 却能够继续执行，让程序员发现其他的错误，或者在某些情况下，产生一个带有错误信

息的输出文件。

图 3-2 说明了 ld 的工作内容。对于多源文件程序，每个源文件被汇编为目标文件(Object File)，链接器负责把这些目标文件，以及相关的库文件链接到一起，形成可执行文件，这就是链接器的作用。

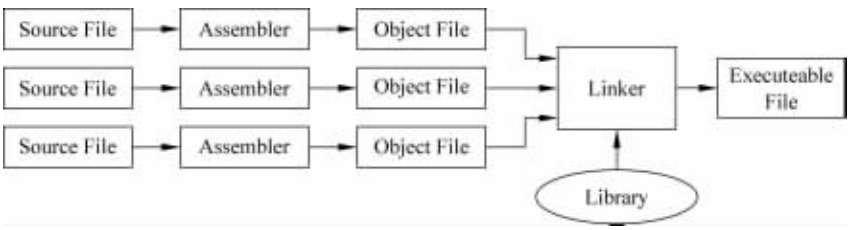

图 3-2 链接器的作用

目标文件由若干段组成，包括代码段、数据段和未初始化数据段等。链接时，ld 将打破目标文件内部结构，把所有代码段都提取出来，共同组成最终可执行程序的代码段；把所有数据段提取出来，组成最终可执行程序的数据段；未初始化的数据段也做同样操作，如图 3-3 所示。

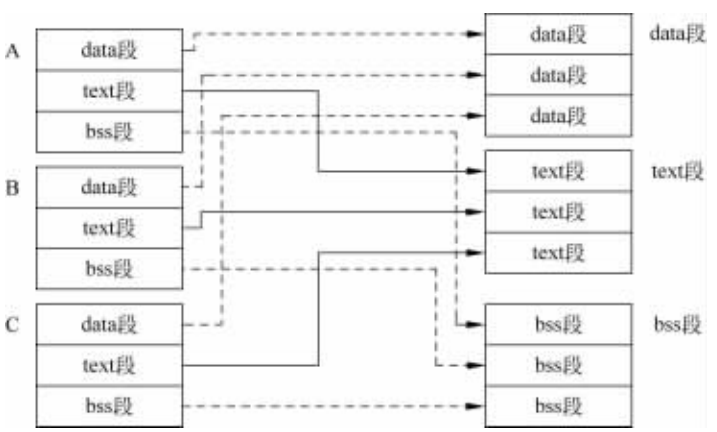

图 3-3 各段链接方式

ld 工作时，要进行两遍扫描，第一遍扫描所有目标文件和函数库，统计应为各段准备的总空间大小，然后安排各段在最终可执行文件中的顺序，并且建立一张临时符号表。第二遍扫描，把各段组合起来，生成可执行程序，并生成最终符号表。由于链接的过程重新组合了各段的位置，原来各段中的符号地址可能发生了变化，因此扫描过程中，十分重要的工作是重新计算各段、标志符号的位置，以保证程序地址的正确性。

ld 程序通过识别链接描述文件(Linker Script)来显式地控制链接的过程。链接描述文件定义了输入文件的各段在可执行文件中的位置，通过自定义的一套语法规则控制结果文件中各段的布局。链接描述文件由链接命令语言(Linker Command Language)编写而成。程序员一般并不直接使用 ld 命令去链接若干目标文件，当用 gcc 编译源文件时，gcc 会自动调用 ld 对各目标文件以及库进行链接。gcc 等编译器内置有默认的链接描述文件，如果采用默认描述文件，则生成的目标代码需要操作系统才能加载运行。

当然也可以显式用 ld 命令来链接各目标文件以及库，但是如下写法是错误的，不能完成由目标文件 hello.o 与库函数链接生成可执行文件 myhello 的目的：

```
#ld hello.o -o myhello
```

正确的写法应该是：

```
#ld -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o    /usr/lib/crti.o\
 /usr/lib/crtn.o hello.o   -lc   -o   myhello
```

可见自己执行 ld 命令的过程是很繁琐的，而由 gcc 去调用 ld 是最方便的方式。

下面介绍几种 ld 的常用选项。

(1) -EB：链接采用大端模式。

(2) -EL：链接采用小端模式。

(3) -l LIBNAME：指定要链接的库文件名。

(4) -L DIRECTORY：指定链接库文件时搜索的路径。

(5) -o FILE：指定输出文件名。

(6) -O LEVEL：指定优化级别。

(7) -T FILE：指定链接描述文件名。

(8) -e：设置执行程序入口点。

下面是一个链接命令的选项(Embest IDE 自动生成)。

```
-EL -O1 -Tconfig.ld -Larm-elf\lib -Lgcc-lib\arm-elf\3.0.2 -LE:\Build\lib -o debug\test.elf $
(<start.o>OBJ_FILES) font_lib/font.lib -lc -lgcc
```

分析可知其采用小端模式、-O1 级别优化，链接时读取链接描述文件 config.ld，利用-L 选项共设置了三个库搜索路径，其中有相对路径也有绝对路径，输出文件为 debug 目录下的 test.elf 文件，要链接 font.lib、libc 以及 libgcc 等库，程序执行的入口点在 start.o 模块中。

链接描述文件支持命令操作，支持简单的数学运算，支持对目标机存储系统的定义，以及负责实际内存在目标机上的映射。段描述是链接描述文件的重要部分，示例如下：

```
SETCTION
{
    .=0x2000000;
    .text:{*(.text)}
    .=0x5000000;
    .data:{*(.data)}
    .bss:{*(.bss)}
}
```

圆点“.”代表的含义是当前位置。这段脚本表示在地址为 0x2000000 处放置程序的代码段，此代码段是由各模块的代码段合并而成，在地址为 0x5000000 处放置数据段，接下来放置未初始化的数据段。

链接描述文件支持简单的赋值语句以及类似 C 语言的复合赋值语句，相关运算符有=、+=、-=、*=、/=、<<=、>>=、&=、!=。

下面的脚本实现了各段位置 4 字节对齐的目的，请读者自行分析其内容。

```
SECTIONS
{
    ROM_BASE = 0x00000000;
    . = 0x0C000000;
    Image_RO_Base = .;
    .text : { *(.text) }
    Image_RO_Limit = .;
    . = (. + 3) & ~3;
    Image_RW_Base = .;
    .data : { *(.data) }
    .rodata : { *(.rodata) }
    Image_RW_Limit = .;
    . = (. + 3) & ~3;
    Image_ZI_Base = .;
    .bss : { *(.bss) }
    Image_ZI_Limit = .;
}
```

3.6 调 试 器

gdb 是 GNU 开源组织发布的 UNIX 下的程序调试工具，基于命令行方式运行，功能强大。gdb 可以按照用户的要求启动程序，支持各种程序运行方式，例如单步执行、深入到函数内部的单步执行、断点(包括条件断点)等。当程序运行暂停时可以检查当前运行环境，例如变量值、内存区等，并且可以动态改变运行环境。使用 gdb 的前提是用 gcc 命令生成可执行文件时，必须加入-g 选项，以在可执行文件中包含 gdb 要用到的调试信息。

gdb 支持 C、C++、Fortran、Pascal、Java、CHILL、Assembly、Modula-2 等语言，支持很多与 UNIX shell 程序一样的命令编辑特征，例如按下 Tab 键，gdb 会自动补齐一个唯一的命令，如果能够匹配的命令不唯一，则 gdb 会列出所有候选命令。此外也能用光标键上下翻动历史命令。

在嵌入式环境下情况有些特殊，目标机可能根本不支持运行 gdb，此时可以使用 gdb 的远程调试功能。被调试的程序运行于目标机，gdb 运行于宿主机，通过 gdb 中内置的串口或网络协议通信，采用远程方式调试运行于目标机的程序。另外，可采用在程序关键点插入 printf()语句，实时打印数据值的方式来完成调试。

在 Linux 环境下，有一些图形化的调试工具可以更方便地调试程序，例如 DDD 就是其中之一。DDD 诞生于 1990 年，起源于 Andreas Zeller 编写的 VSL 结构化语言，后来又经过其他程序员的共同努力，发展为今天的形式。DDD 的功能很强大，可用来调试 C/C++、Ada、Fortran、Pascal 等语言编写的程序，具有图形显示功能，可以将数据结构以图形化显示。

下面通过一个简单的例子演示 gdb 的功能。

【例 3-3】 程序先求出 6 的阶乘，然后在把结果除以 2，代码如下：

```
1  /*  ch3_3.c  */
```

```
2  #include <stdio.h>
3  int func(int m)
4  {
5       int a;
6       float b;
7       a=m;
8       b=0.5;
9       a=b*a;
10      return a;
11 }
12 int main()
13 {
14      int i;
15      int fact=1;
16      for(i=6;i>=1;i--)
17      {
18           fact*=i;
19      }
20      printf("the factorial of 6 is %d\n",fact);
21      printf("the half of the number is: %d\n",func(fact));
22      return 0;
23 }
```

执行下面的命令得到包含调试信息的可执行文件 ch3_3：

```
#gcc -g ch3_3.c -o ch3_3
```

执行下面的命令启动 gdb 调试程序：

```
#gdb ch3_3
```

在 gdb 提示符下，执行 list 命令可以查看程序源代码：

```
(gdb)list 1                                    //从第一行开始显示源代码,默认显示 10 行
1  /*   ch3_3.c   */
2  #include <stdio.h>
3  int func(int m)
4  {
5       int a;
6       float b;
7       a=m;
8       b=0.5;
9       a=b*a;
10      return a;
(gdb)list                                      //继续显示后面的 10 行
11 }
12 int main()
13 {
14      int i;
15      int fact=1;
16      for(i=6;i>=1;i--)
17      {
```

```
18          fact * =i;
19      }
20      printf("the factorial of 6 is %d\n",fact);
```

接下来用 break 命令在 func()函数以及主函数中的第 18 行处设置两个断点,观察断点是否设置成功,然后用 run 命令运行程序,以及用 continue 命令继续执行程序,观察程序是否能在断点处停下,同时观察此时内存变量的值。

```
(gdb)break func                                    //在函数 func()处设置断点
Breakpoint 1 at 0x401066: file ch3_3.c line 7.     //提示断点 1 设置成功
                                                   //位置是文件第 7 行
(gdb)break 18                                      //在文件第 18 行设置断点
Breakpoint 2 at 0x401113: file ch3_3.c line 18.    //提示断点 2 设置成功
                                                   //位置是文件第 18 行
(gdb)info break                                    //查询所有断点
Num   Type        Disp  Enb  Address    What
1     breakpoint  keep  y    0x401066   in func at ch3_3.c:7
2     breakpoint  keep  y    0x401113   in main at ch3_3.c:18
//上述 3 行含义为 gdb 显示目前已经设置了两个断点,同时显示了断点的各种信息
(gdb)run                                           //运行程序
Starting program: /ch3_3
Breakpoint 2, main() at ch3_3.c:18                 //程序停在断点 2 处
18      fact * = i;
(gdb)print i                                       //显示变量 i 的当前值
$1 = 6
(gdb)print fact                                    //显示变量 fact 的当前值
$2 = 1
(gdb)next                                          //单步运行
16      for( i=6; i>=1; i--)                       //即将执行到第 16 行,for 语句
(gdb)next                                          //单步执行
Breakpoint 2, main() at ch3_3.c:18                 //程序仍停在断点 2 处
18      fact * = i;
(gdb)print i                                       //显示变量 i 的当前值
$3 = 5
(gdb)print fact                                    //显示变量 fact 的当前值
$4 = 6
(gdb)delete 2                                      //删除第 2 个断点
(gdb)info break                                    //查询所有断点
Num   Type        Disp  Enb  Address    What
1     breakpoint  keep  y    0x401066   in func at ch3_3.c:7
//上述 2 行含义为 gdb 显示当前只剩下一个断点
(gdb)continue                                      //继续运行程序
Continuing.
the factorial of 6 is 720                          //显示第 20 行的输出信息

Breakpoint 1, func (m=720) at ch3_3.c:7            //程序中断在第 1 个断点处
7       a=m;
(gdb)print a                                       //显示变量 a 的值
$5 = 26                                            //因变量还未赋值,显示不确定的数
(gdb)print b                                       //显示变量 b 的值
$6 = 0                                             //因变量还未赋值,显示不确定的数
```

```
(gdb)continue                                    //继续执行程序
Contining.
the half of the number is:360                    //显示第 21 行的输出信息

Program exited normally      .                   //程序调试执行完毕,正常终止
(gdb)quit                                        //退出 gdb
```

这个例子演示了 gdb 的一般应用,下面介绍 gdb 的详细使用方法。

1. 开始使用 gdb

用 gdb 启动程序有两种方式,可以直接执行:

```
#gdb program
```

或者先执行 gdb,然后利用 gdb 的 file 命令加载要调试的程序:

```
#gdb
(gdb)file program
```

两种方式效果相同。

gdb 提供了详细的帮助功能,可以通过 help 命令来查看命令的用法。

```
(gdb)help                                        //键入 help,显示如下信息
List of classes of commands:

aliases -- Aliases of other commands             //每个 gdb 命令都属于某一大类
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

每一个 gdb 命令都被归到某个大类中,如果想了解某个类,可以键入 help classname 来查询。例如:

```
(gdb)help status                                 //显示 status 类的所有命令
Status inquiries.

List of commands:                                //status 类包含 3 个命令

info -- Generic command for showing things about the program being debugged
macro -- Prefix for commands dealing with C preprocessor macros
```

```
show -- Generic command for showing things about the debugger

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

进一步还可以查询命令的具体用法，例如查询 macro 命令的用法，可以如下操作：

```
(gdb)help macro
Prefix for commands dealing with C preprocessor macros.

List of macro subcommands:

macro define -- Define a new C/C++ preprocessor macro
macro expand -- Fully expand any C/C++ preprocessor macro invocations in EXPRESSION
macro expand-once -- Expand C/C++ preprocessor macro invocations appearing directly in EXPRESSION
macro list -- List all the macros defined using the 'macro define' command
macro undef -- Remove the definition of the C/C++ preprocessor macro with the given name

Type "help macro" followed by macro subcommand name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

Linux 系统和 Linux 下的应用软件均有详细的帮助功能，可充分加以利用。

2. gdb 的常用命令

(1) file：装入想调试的可执行文件。

(2) run：在 gdb 中运行程序。

(3) continue：继续运行被中止的程序。

(4) kill：终止正在调试的程序。

(5) quit：退出 gdb。

(6) list：列出源代码。

(7) info：查看断点等信息。

(8) disassemble：显示函数反汇编代码。

(9) x addr：显示内存单元内容。

(10) print：显示变量的值，只显示一次。

(11) display：显示变量的值，每次单步运行都会显示。

(12) undisplay：取消显示变量的值。

(13) backtrace：查看栈信息。

(14) where：得到函数调用链信息。

(15) next：单步执行程序，函数调用被当作一条语句执行，相当于 step over。

(16) step：单步执行程序，会深入到函数调用内部单步执行，相当于 step into。

(17) finish：执行完函数剩余部分，相当于 step out。

(18) return：不执行函数剩余部分，直接返回。

(19) call：强制调用函数。

(20) up：向栈上方移动。

(21) down：向栈下方移动。

(22) watch：监视变量的值。

(23) break：设置断点。

(24) jump：程序跳转到某处继续执行。

(25) show：查看环境变量等信息。

(26) delete：清除停止点。

(27) disable：禁止自动显示、断点等。

(28) enable：启用自动显示、断点等。

(29) set var：设置变量的值。

(30) search：查找字符串。

(31) whatis：得到变量的类型。

(32) make：不退出 gdb 就可以重新产生可执行文件。

(33) shell：在 gdb 中执行 shell 命令。例如：

```
(gdb)shell ls
```

结果会在 gdb 界面下显示当前目录下文件列表。

在 gdb 提示符下直接回车，相当于重复执行上一条命令；可以利用 Tab 键补齐命令，加快输入速度；各条命令在不冲突的情况下都有缩写形式，例如 break 可以缩写为 b，info 命令可以缩写为 i。

在 gdb 下可以指定程序运行的参数，例如下面的语句指定参数为 10、20：

```
set args 10 20
```

在 gdb 中可以显示当前目录，执行命令：

```
(gdb)pwd
Working directory /root.
```

在 gdb 中可以重定向输出，还是以例 3-3 为例：

```
(gdb)run > output
Starting program: /root/ch3_3 > output
Program exited normally.
```

然后在当前目录下，显示文件 output 的内容，可以看到：

```
result(1-100)= 5050
result(1-250)= 31375
```

正是程序的输出内容。

3. gdb 的停止点

gdb 中程序的暂停方式有以下几种：断点(Breakpoint)、观察点(Watchpoint)、捕捉点(Catchpoint)、信号(Signals)和线程停止(Thread Stops)，统称停止点。

break 命令用于设置断点，有以下几种方法。

(1) break function：执行到某函数时暂停。

(2) break linenum：执行到某行时暂停。

(3) break +offset：执行到当前行后第 offset 行时暂停。

(4) break -offset：执行到当前行前第 offset 行时暂停。

(5) break filename:linenum：执行到某文件的某行时暂停。

(6) break filename:function：执行到某文件的某函数时暂停。

(7) break *address：执行到某内存地址处时暂停。

(8) tbreak：仅中断一次，中断后断点自动删除。

(9) break … if condition：条件断点。

(10) condition：修改条件断点的条件。

条件断点是个有用的特性，gdb 允许用各种表达式设置条件断点，例如，执行到第 10 行时，如果 i 的值为 20 则中断，可表示如下：

```
(gdb)break 10 if i==20
```

用 condition 命令可以修改断点的条件，例如修改 1 号断点的条件：

```
(gdb)condition 1 i==30                    //把中断条件改为如果 i 的值为 30,执行到此处停止
```

用 condition 加断点号可以把条件断点转变为一般断点，例如：

```
(gdb)condtion 1                           //清除 1 号断点的条件,转变为一般断点
```

ignore 命令表示程序运行时，忽略中断若干次：

```
(gdb)ignore 1     10                      //执行程序,忽略 1 号断点的前 10 次中断
```

可以使用 info 命令来查询停止点。

(1) info breakpoint：查看所有断点。

(2) info breakpoint n：查看第 n 个断点。

观察点与断点稍有差异，一般用来观察某个表达式是否发生变化，如果变化则暂停程序。使用方式如下。

(1) watch expr：表达式 expr 变化则停止。

(2) rwatch expr：表达式 expr 被读时停止。

(3) awatch expr：表达式 expr 被读写时停止。

维护停止点包括删除停止点、禁用停止点、启用停止点等操作，主要使用以下命令。

(1) delete：删除停止点。

(2) clear：清除停止点。

(3) disable：禁用停止点。

(4) enable：启用停止点。

可以一次清除所有停止点，也可以清除函数中的停止点，或某行的停止点等。

(1) clear：清除所有停止点。

(2) clear function：清除函数的停止点。

(3) clear filename:function：清除某文件中函数的停止点。

(4) clear linenum：清除某行的停止点。

(5) clear filename:linenum：清除某文件中函数的停止点。

(6) delete breakpoints range：清除号码在某个范围的断点，如 d b 1-3。

(7) delete：清除所有断点。

(8) disable breakpoints range：禁用号码在某个范围的断点，如 disalbe b 1-3。

(9) enable breakpoints range：启用号码在某个范围的断点，如 ena b 1 2 3。

(10) enable breakpoints once range：仅启用某些断点一次，停止后立刻 disable。

(11) enable breakpoints delete range：仅启用某些断点一次，停止后立刻删除。

可以利用 command 命令为停止点设定一系列运行命令，例如：

```
(gdb)commands 1                                          //为 1 号断点设置命令
Type commands for when breakpoint 1 is hit.  one per line.  //gdb 提示信息
End with a line saying just "end".
>shell pwd                                              //停止时要执行的命令
>shell ls -l
>end                                                    //设置结束
```

程序指定到 1 号断点处会中断，并执行设置好的 shell 命令。

4. 查看和修改变量值

设置停止点的目的是检查程序当前上下文变量的值，如果这些值与预期一致，说明程序执行正确，否则就出现了错误。观察变量值主要使用 print 命令和 display 命令。

(1) print /<formation> n：显示变量 n 的值。

(2) print /<formation> ::n：显示全局变量 n 的值。

(3) print /<formation> array：显示数组各元素值。

(4) print /<formation> *array@length：显示动态分配内存的数组各元素值。

其中<formation>代表数据的显示格式如下。

(1) x、a：十六进制格式显示变量。

(2) d：十进制格式显示变量。

(3) u：十六进制格式显示无符号整形。

(4) o：八进制格式显示变量。

(5) t：二进制格式显示变量。

(6) c：按字符格式显示变量。

(7) f：按浮点数格式显示变量。

【例 3-4】 程序示例：

```
1  /*   ch3_4.c   */
2  #include <stdlib.h>
3  int globle=120;
4  int main()
5  {
6       int a[]={1,2,3,4,5};
7       int i;
8       int *p;
9       p=(int*)malloc(20);
10      for(i=0;i<5;i++)
11      {
12           p[i]=globle/a[i];
```

```
13      }
14      free(p);
15      return 0;
16 }
```

用 break 14 在第 14 行设置断点，执行 run 命令，程序停在第 14 行，执行下面各显示命令：

```
(gdb)print i
$1 = 5
(gdb)print globle
$2 = 120
(gdb)print ::globle
$3 = 120
(gdb)print a
$4 = {1, 2, 3, 4, 5}
(gdb)print *p@5
$5 = {120, 60, 40, 30, 24}
```

用 display 命令也可以显示变量的值，与 print 的区别是变量值在每次单步运行后都能自动显示，而用 print 显示的变量只能显示一次，如还需显示那么还得再次运行 print 命令。

在 gdb 中还可以利用 show 命令和 set 命令查看和修改系统运行的环境变量。

(1) show paths：查看程序运行路径。

(2) show env：查看所有环境变量。

(3) show env HOME：查看某环境变量。

(4) set env LINES=25：设置某环境变量。

调试程序是编程过程中必不可少的一个环节，是否能用调试器顺利找到代码中隐藏的错误是编程技巧高低的重要标志，需要在实践中不断总结、提高。

3.7 自动化编译配置文件

本节内容改编自徐海兵翻译的 GNU make 中文手册，承蒙徐海兵允许把他的作品编入本书，在此向徐海兵致谢！

3.7.1 自动化编译配置文件简介

Linux 环境下的程序员如果不会使用 GNU make 命令工具来构建和管理自己的工程，应该不能算是一个合格的专业程序员。在 UNIX/Linux 环境下使用 GNU 的 make 工具能够比较容易地构建一个自己的工程，并且只需要一个命令就可以完成预处理、编译和链接的完整过程，不过这需要投入一些时间去编写一个或者多个称为 Makefile 的自动化编译配置文件，此文件是 make 命令正常工作的基础。

Makefile 文件描述了整个工程的编译、链接规则，其中包括：工程中的哪些源文件需要编译以及如何编译，需要创建哪些库文件以及如何创建这些库文件，如何产生期望得到的最终可执行文件。一旦提供了正确的 Makefile，编译整个工程所要做的唯一操作就是在 shell

提示符下输入 make 命令。整个工程完全自动编译，极大提高了效率。假设没有 Makefile，对于一个大工程，每次修改了源文件之后，需要在命令行模式下重新编译修改过的源文件，然后还需要把所有的目标文件链接起来，这个过程无疑是很烦琐的。

Makefile 文件描述了工程中所有文件的编译顺序、编译规则，make 命令根据这些规则来编译工程。Makefile 有自己的书写格式、关键字和函数，如同 C 语言有自己的格式、关键字和函数一样。而且在 Makefile 中可以使用系统 shell 所提供的任何命令来完成需要的工作。Makefile 可以在绝大多数 IDE 开发环境中使用，已经成为工程编译的最有效方法。

由于 C 语言是使用最广泛的编程语言之一，本节中所有示例均针对 C 语言源程序，实际上，只要某编译器能够在 shell 下运行，那么采用这种语言设计的工程就可以用 make 工具来管理。同时，make 命令不仅仅可以用来指导编译源代码，还能够完成很多其他功能。可以根据工程中源文件的修改情况来有选择地编译代码，这是 make 命令的突出特性。

【例 3-5】 程序只包含一个文件，有两种编译方式。

```
/*   ch3_5.c   */
#include <stdio.h>
int main()
{
    printf("hello everybody. \n");
    return 0;
}
```

方式 1：命令行模式下直接使用 gcc 命令编译程序 ch3_5。方式如下：

```
#gcc hello.c -o hello
```

方式 2：使用 GNU make 工具编译程序，需要编写 Makefile。内容如下：

```
hello: hello.c
gcc hello.c -o hello
```

然后在 shell 提示符下执行 make 命令，同样可以编译得到 hello 程序，方式如下：

```
#make
```

【例 3-6】 由两个源文件组成，同样有两种编译方式，代码如下：

```
/* ch3_6 main.c */
#include <stdio.h>
main()
{
    printf("hello.\n");
    foo();
}
/*   ch3_6 foo.c */
#include <stdio.h>
void foo()
{
    printf("you are in foo.\n");
}
```

方式 1：以命令行模式编译程序例 3-6。使用命令如下：

```
#gcc main.c foo.c -o hello
```

方式 2：使用 GNU make 工具，需要编写 Makefile。内容如下：

```
hello: main.o foo.o
    gcc main.o foo.o -o hello
main.o: main.c
    gcc main.c -o main.o
foo.o: foo.c
    gcc foo.c -o foo.o
```

在命令行运行 make 命令，同样可以编译出可执行程序 hello，如图 3-4 所示。

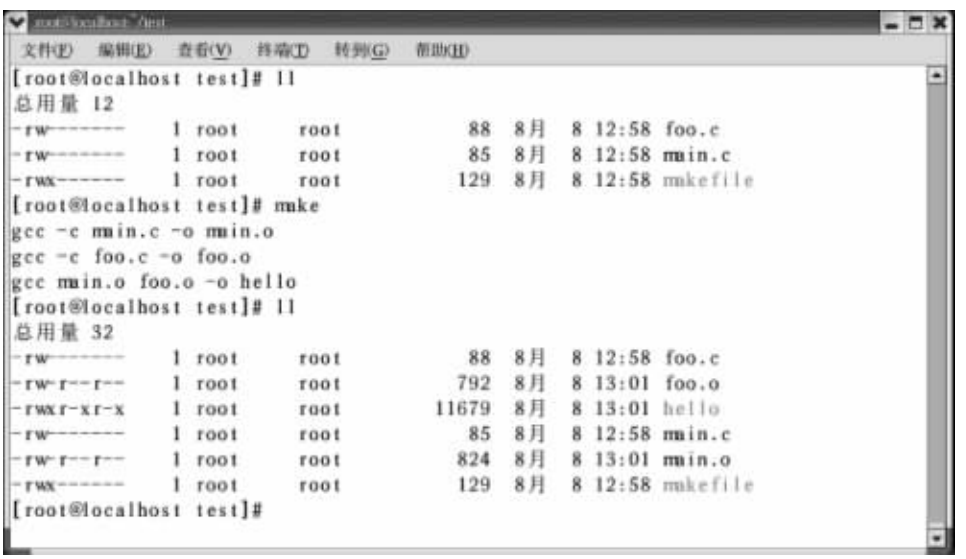

图 3-4 采用 make 命令编译程序

图 3-4 中，运行 make 命令之后，编译出 main.o 和 foo.o 两个目标文件及程序 hello，可以看到三个文件的生成时间均为 13:01。

如果修改了源文件 foo.c，采用方式 1 重编译程序，在编译 foo.c 的同时，未被修改的 main.c 也被重新编译，这显然是效率低下的方法。采用方式 2 重编译程序，make 命令会分析各源文件、目标文件和可执行程序的生成时间，发现源文件 foo.c 的修改时间较 foo.o 的生成时间晚，因此可以断定 foo.c 文件经过了修改，所以需要重编译 foo.c；同时比较发现 main.c 的时间较 main.o 早，说明 main.c 文件自从前次编译之后未被修改，因此不需要重新编译；然后，需要把 main.o 和再次编译得到的 foo.o 链接生成新版可知性文件 hello，如图 3-5 所示。观察各文件的生成时间，可以看到 main.c 和 main.o 文件的生成时间没有变化，说明 make 程序只重新编译了需要编译的部分，提高了工作效率，这就是 make 工具的工作原理。

自动化编译配置文件通常可以起名为：GNUmakefile、makefile 或 Makefile。make 命令会在当前目录下按照这个顺序搜索，读取并执行找到的第一个自动化编译配置文件。程

序员通常应该使用 makefile 或者 Makefile 作为文件名，并不推荐使用 GNUmakefile，因为以此命名的文件只有 GNU make 才可以识别，而其他版本的 make 程序只会在工作目录下搜索 makefile 和 Makefile 这两个文件。通过 make 的-f 或者—file 选项可以指定 make 读取任意文件名的 Makefile 文件，采用的语法格式为：make -f NAME 或者 make —file＝NAME，它指定文件 NAME 作为执行 make 时读取的 Makefile 文件。

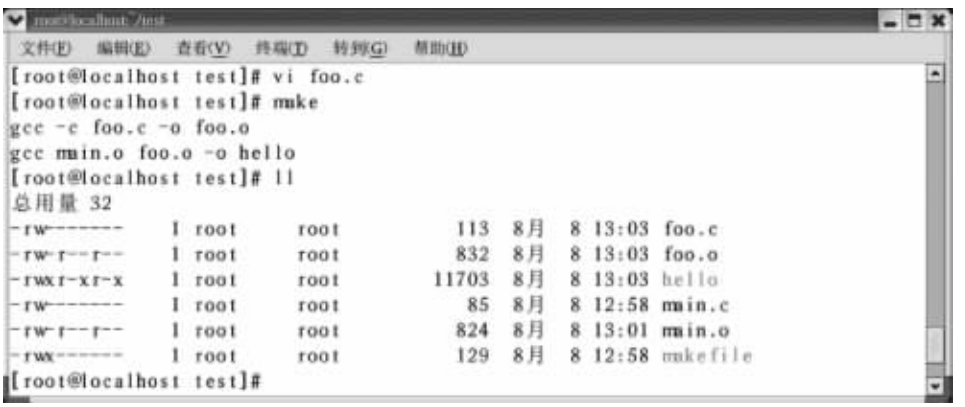

图 3-5　修改源文件后重新采用 make 命令编译程序

如果 make 程序在工作目录下无法找到以上三个文件中的任何一个，它将不读取任何其他文件作为解析对象。但是根据 make 隐含规则的特性，可以通过命令行指定一个目标，如果当前目录下存在符合此目标的依赖文件，那么此命令行所指定的目标将会被创建或者更新。例如当前目录下存在一个源文件 a.c，可以执行 make a.o 来使用 make 的隐含规则自动生成 a.o。

GUN make 的执行过程分为以下几步。第一步，在 Linux 命令提示符下输入 make，它会在当前目录下按顺序寻找 GNUmakefile、makefile、Makefile。如未找到则报错，如找到，则把 Makefile 文件中的第一个目标作为最终目标。第二步，按照“堆栈”顺序，依序找到每一个目标文件，判断新旧关系，必要时生成新的目标文件，直到生成最终目标。具体来说，包括读入被包括的其他 Makefile、初始化文件中的变量、推导隐含规则、分析所有规则、为所有的目标文件创建依赖关系链、根据依赖关系决定哪些文件要重新生成、最后执行生成命令。在寻找依赖文件过程中，如某依赖文件不存在，则 make 直接退出，并报错。如改变了源文件或头文件，与之相关的.o 文件和最终目标文件都要重新编译，但只需在 Linux 命令提示符下键入 make 即可。

3.7.2　Makefile 规则

1. Makefile 显式规则

下面介绍一下 Makefile 的基本知识，Makefile 的基本结构由描述规则组成，规则负责描述在何种情况下，如何重建目标文件，通常规则中包括了目标的依赖关系和重建目标的命令。make 命令工具执行重建目标的命令，创建或者重建规则的目标。

一个简单的 Makefile 描述规则如下：

```
target… : prerequisites…
```

command…
…

target 称为规则的目标，通常是最后需要生成的文件名，或是为了实现这个目的而必需的中间过程文件名。可以是.o 文件、也可以是最后的可执行程序的文件名等。另外，目标也可以是一个 make 需要执行的动作的名称，如目标 clean，称这样的目标是“伪目标”。伪目标一般不会被自动执行，而必须通过命令行指定伪目标名才能执行，例如 make clean。

prerequisites 称为规则的依赖文件，是生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件。

command 行称为规则的命令行，这是规则所要执行的动作，可以是任意的 shell 命令或者是可在 shell 下执行的程序，它限定了 make 执行这条规则时执行的操作。

一个规则可以有多个命令行，每一条命令占一行，并且每一个命令行必须以一个水平制表符（键盘上的 Tab 键）开始，这是命令行的标志。make 命令读取 Makefile 文件，按照命令行的内容执行相应动作。遗漏掉命令行的水平制表符是书写 Makefile 时最容易产生的错误，而且往往比较隐蔽，容易被忽视。

命令是在任何一个目标的依赖文件发生变化后重建目标的动作描述。一个目标可以没有依赖而只有动作，即指定命令。例如 Makefile 中的目标 clean，此目标没有依赖，只有命令。它所定义的命令用来删除 make 过程产生的中间文件，进行清理工作。

一个最简单的 Makefile 可能只包含规则。规则在有些 Makefile 中可能看起来非常复杂，但是无论规则的书写是多么的复杂，它都符合规则的基本格式。make 程序根据规则的依赖关系，决定是否执行规则所定义的命令的过程称之为执行规则。Makefile 文件中通常还包含了除规则以外的很多东西。

假设某个工程由 3 个头文件和 8 个源文件组成，下面通过一个简单的 Makefile，来描述如何创建最终的可执行文件 edit，此可执行文件依赖于这 8 个源文件和 3 个头文件。

【例 3-7】 Makefile 文件的内容。

```
#ch3_7      makefile1
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
    gcc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
main.o : main.c defs.h
    gcc -c main.c
kbd.o : kbd.c defs.h command.h
    gcc -c kbd.c
command.o : command.c defs.h command.h
    gcc -c command.c
display.o : display.c defs.h buffer.h
    gcc -c display.c
insert.o : insert.c defs.h buffer.h
    gcc -c insert.c
search.o : search.c defs.h buffer.h
    gcc -c search.c
files.o : files.c defs.h buffer.h command.h
    gcc -c files.c
utils.o : utils.c defs.h
    gcc -c utils.c
```

```
clean :
    rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

在这个 Makefile 中，目标是可执行文件 edit 和若干.o 文件(main.o，kbd.o…)；依赖文件(Prerequisites)就是冒号后面的.c 文件和.h 文件。所有的.o 文件既是依赖文件(相对于可执行程序 edit)又是目标文件(相对于.c 文件和.h 文件)。命令包括 gcc -c maic.c、gcc -c kbd.c 等。

如果规则的目标是一个文件，那么在它的任何一个依赖文件被修改以后，执行 make 命令时这个目标文件都将会被重新编译或者重新链接。当然，此目标的任何一个依赖文件如果有必要则首先会被重新编译。在这个例子中，edit 的依赖为 8 个.o 文件；而 main.o 的依赖文件为 main.c 和 defs.h。当 main.c 或者 defs.h 被修改以后，再次执行 make，main.o 就会被更新，而其他的.o 文件不会被更新，同时 main.o 被更新将会导致 edit 被更新。

依赖关系行之下通常就是规则的命令行，命令行定义了规则的动作，即如何根据依赖文件来更新目标文件。命令行必须以 Tab 键开始，以和 Makefile 其他行区别。

目标 clean 不是一个文件，它仅仅代表执行一个动作的标识。正常情况下，不需要执行这个规则所定义的动作，因此目标 clean 没有出现在其他任何规则的依赖列表中。因此在执行 make 时，它所指定的动作不会被执行。除非在执行 make 时明确地指定它。而且目标 clean 没有任何依赖文件，它只有一个目的，就是通过这个目标名来执行它所定义的命令。在 Makefile 中，那些没有任何依赖只有执行动作的目标称为伪目标(Phony targets)。如需要执行 clean 目标所定义的命令，可在 shell 下输入：make clean。

默认情况下，make 执行的是 Makefile 中的第一个规则，此规则的第一个目标称之为“默认目标”，是一个 Makefile 最终需要更新或者创建的目标，执行不带参数的 make 命令时，最终要生成的就是默认目标代表的文件。

当在 shell 提示符下输入 make 命令以后，make 读取当前目录下的 Makefile 文件，并将 Makefile 文件中的第一个目标作为其执行的“默认目标”，开始处理第一个规则(默认目标所在的规则)。例子中，第一个规则就是目标 edit 所在的规则。规则描述了 edit 的依赖关系，并定义了链接.o 文件生成目标 edit 的命令；make 在执行这个规则所定义的命令之前，首先处理目标 edit 的所有的依赖文件(例子中的那些.o 文件)的更新规则(以这些.o 文件为目标的规则)。对这些.o 文件为目标的规则处理有下列 3 种情况：

(1) 目标.o 文件不存在，使用其描述规则创建它；

(2) 目标.o 文件存在，目标.o 文件所依赖的.c 源文件、.h 文件中的任何一个比目标.o 文件“更新”(在上一次 make 之后被修改)，则根据规则重新编译生成它；

(3) 目标.o 文件存在，目标.o 文件比它的任何一个依赖文件的(.c 源文件、.h 文件)“更新”，这表明它的依赖文件在上一次 make 之后没有被修改，则什么也不做。

这些.o 文件所在的规则之所以会被执行，是因为这些.o 文件出现在“默认目标”的依赖列表中。一个规则的目标如果不是“默认目标”所依赖的，或者“默认目标”的依赖文件所依赖的，那么这个规则将不会被执行，除非通过 make 的命令行明确指定此目标，例如 make clean。在编译或者重新编译生成一个.o 文件时，make 同样会去寻找它的依赖文件的重建规则。在上例的 Makefile 中没有哪个规则的目标是.c 源文件或者.h 文件，所以没有重建.c 源文件和.h 文件的规则。

完成了对.o文件的创建(第一次编译)或者更新之后,make程序将处理默认目标edit所在的规则,分为以下3种情况:

(1) 目标文件edit不存在,则执行规则以创建目标edit;

(2) 目标文件edit存在,其依赖文件中有一个或者多个文件比它"更新",则根据规则重新链接生成edit;

(3) 目标文件edit存在,并且比它的任何一个依赖文件都"更新",则什么也不做。

上例中,如果更改了源文件insert.c后执行make,insert.o将被更新,之后默认目标edit将会被重生成;如果修改了头文件command.h之后运行make,那么kbd.o、command.o和files.o将会被重新编译,之后同样默认目标edit也将被重新生成。

简单总结一下:对于一个Makefile文件,make首先解析默认目标所在的规则,根据其依赖文件(例子中第一个规则的8个.o文件)依次(按照依赖文件列表从左到右的顺序)寻找创建这些依赖文件的规则。首先为第一个依赖文件(main.o)寻找创建规则,如果第一个依赖文件依赖于其他文件(main.c、defs.h),则同样为这个依赖文件寻找创建规则(创建main.c和defs.h的规则,通常源文件和头文件已经存在,也不存在重建它们的规则),依此类推,直到为所有的依赖文件找到合适的创建规则。之后make从最后一个规则(上例目标为main.o的规则)回退开始执行,最终完成默认目标的第1个依赖文件的创建和更新。之后对第2个、第3个、第4个、……默认目标的依赖文件执行同样的过程(上例的顺序是main.o、kbd.o、command.o、…)。

创建或者更新每一个规则依赖文件的过程都是这样的一个过程(类似于C语言中的递归过程)。对于任意一个规则执行的过程都是按照依赖文件列表顺序,对于规则中的每一个依赖文件,使用同样方式(按照同样的过程)去重建它,在完成对所有依赖文件的重建之后,最后一步才是重建此规则的目标。

更新(或者创建)默认目标的过程中,如果任何一个规则执行出现错误make就立即报错并退出。整个过程中,make只是负责执行规则,而对具体规则所描述的依赖关系的正确性、规则所定义的命令的正确性不做任何判断。就是说,一个规则的依赖关系是否正确、描述重建目标的规则命令行是否正确,make不做任何错误检查。因此,需要正确地编译一个工程。需要在提供给make程序的Makefile中来保证其依赖关系的正确性和执行命令的正确性。

2. 指定变量

同样是上边的例子,考察一下默认目标edit所在的规则:

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
gcc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

在这个规则中.o文件列表出现了两次;第1次作为目标edit的依赖文件列表出现,第2次为规则命令行中作为gcc的参数列表出现。这样做所带来的问题是:如果需要为目标edit增加一个的依赖文件,就需要在依赖文件列表和规则命令这两个地方添加,这显然是比较繁琐的。实际工作中可以这样处理:使用一个变量objects作为所有的.o文件列表的替代。在需要使用这些文件列表的地方,使用此变量来代替。在上例的Makefile中可以添加这样一行:

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

objects 作为一个变量，它代表所有的.o 文件的列表。规则因此可以这样写：

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
edit : $(objects)
    gcc -o edit $(objects)
    …
clean :
    rm edit $(objects)
```

当需要为默认目标 edit 增加或者删减一个.o 依赖文件时，只需要改变 objects 的定义，加入或者去掉这个.o 文件。这样做不但减少书写的工作量，而且可以减少因修改而产生错误的可能。

3. 隐含规则

在使用 make 编译.c 源文件时，规则的命令可以不必明确给出。这是因为 make 本身存在一个默认的规则，能够自动完成对.c 文件的编译并生成对应的.o 文件。它执行命令 cc -c 来编译.c 源文件，cc 是 gcc 的符号链接。在 Makefile 中只需要给出需要重建的目标文件名，通常是一个.o 文件，make 会认为它的依赖文件是除后缀.o 之外，文件名的其余部分都相同的文件，然后会自动寻找这个依赖文件，并使用正确的命令来创建目标文件。对于上边的例子，此默认规则为使用命令 gcc -c main.c -o main.o 来创建文件 main.o。对一个目标文件是 N.o，倚赖文件是 N.c 的规则，完全可以省略其规则的命令行，而由 make 自身决定使用默认命令。此默认规则称为 make 的隐含规则。

这样，在书写 Makefile 时，就可以省略掉描述.c 文件和.o 文件依赖关系的规则，而只需要给出那些额外的规则描述，例如.o 目标所需要的.h 文件。因此上边的例子就可以以更加简单的方式书写，同样使用变量 objects。Makefile 内容如下所示：

```
#ch3_8      makefile2
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
edit : $(objects)
    gcc -o edit $(objects)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
.PHONY : clean
clean :
    rm edit $(objects)
```

这种格式的 Makefile 更接近于实际应用，关于目标 clean 的详细说明在后边进行。

make 的隐含规则在实际工程中会经常使用，它使得编译过程变得方便。几乎在所有的 Makefile 中都用到了 make 的隐含规则，make 的隐含规则是非常重要的一个概念。后续将会再专门地讨论。

Makefile 中，目标可以使用隐含规则生成，再深入一步，可以书写更简洁的 Makefile。例如上述 Makefile 还可以这样来实现：

```
#ch3_9      makefile3
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
edit : $(objects)
    gcc -o edit $(objects)
$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
    .PHONY : clean
    clean :
    rm edit $(objects)
```

例子中头文件 defs.h 作为所有.o 文件的依赖文件。其他两个头文件作为其对应规则的目标中所列举的所有.o 文件的依赖文件。

但是这种风格的 Makefile 并不值得借鉴。问题在于把多个目标文件的依赖放在同一个规则中进行描述，一个规则中含有多个目标文件，这导致规则定义不够清晰明了。建议大家不要在 Makefile 中采用这种方式书写，否则后期维护将会是一件非常费力的事情。

4. 清空目标文件的规则

规则除了完成源代码编译之外，也可以完成其他任务。例如，前文提到为了实现清除当前目录中编译产生的临时文件——edit 和多个.o 文件——的规则：

```
clean :
    rm edit $(objects)
```

在实际应用时，可以把这个规则写成如下稍微复杂一些的样子，可以让 Makefile 的适应性更强。

```
clean :
    -rm edit $(objects)
```

在命令行之前使用“-”，含义是忽略命令“rm”可能出现的执行错误，例如磁盘上根本不存在指定要删除的文件等。

在 Makefile 中，不能将这样的目标作为第一个目标，即默认目标。因为用户的初衷并不是在命令行上输入 make 之后执行删除动作，而是要创建或者更新程序。目标 clean 不应出现在默认目标 edit 的直接或间接依赖关系中，所以执行 make 时，目标 clean 所在的规则将不会被处理。当需要执行此规则，要在 make 的命令行选项中明确指定这个目标，执行 make clean。

5. 模式规则

模式规则类似于普通规则。只是在模式规则中，目标名中需要包含有一个模式字符%，包含有模式字符的目标被用来匹配一个文件名。规则的依赖文件中同样可以使用模式字符%，其取值情况由目标中的%来决定。例如：对于模式规则%.o：%.c，它表示所有的.o 文件依赖于对应的.c 文件。可以使用模式规则来定义隐含规则。模式规则中的%的匹配和替换则发生在 make 执行时。

在模式规则中，目标文件是一个带有模式字符%的文件，使用模式来匹配目标文件。文件名中的模式字符%可以匹配任何非空字符串，除模式字符以外的部分要求一致。例如：%.c 匹配所有以.c 结尾的文件，匹配的文件名长度最少为 3 个字母；s%.c 匹配所有第一个字母为 s，而且必须以.c 结尾的文件。因此，一个模式规则的格式为：

```
%.o : %.c ;
```

这个模式规则指定了如何由文件 N.c 来创建文件 N.o，文件 N.c 应该是已存在的或者可被创建的。

模式规则中依赖文件也可以不包含模式字符%。当依赖文件名中不包含模式字符%时，其含义是所有符合目标模式的目标文件都依赖于一个指定的文件。例如：%.o : debug.h，表示所有的.o 文件都依赖于头文件 debug.h。这样的模式规则在很多场合是非常有用的。

考察下面的例子，其中包括编译.c 文件到.o 文件的隐含模式规则：

```
%.o : %.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

此规则描述了一个.o 文件如何由对应的.c 文件创建。规则的命令行中使用了自动化变量$<和$@，其中$<代表规则的依赖文件，$@代表规则的目标文件。此规则在执行时，命令行中的自动化变量将根据实际的目标和依赖文件取对应值。自动化变量在下文有详细说明。

6. 伪目标

伪目标是这样一个目标：它不代表一个真正的文件名，在执行 make 时可以指定这个目标来执行其所在规则定义的命令，有时也可以将一个伪目标称为标签或标号。如果需要书写这样一个规则：规则所定义的命令不是去创建目标文件，而是通过 make 命令行明确指定它来执一些特定的命令，例如常见的 clean 目标：

```
clean:
    rm *.o temp
```

规则中 rm 不是创建文件 clean 的命令，而是为了删除当前目录下的所有.o 文件和 temp 文件。当工作目录下不存在 clean 这个文件时，输入 make clean，rm *.o temp 总会被执行。但是如果在当前工作目录下存在文件 clean，情况就不一样了，同样输入 make clean，由于这个规则没有任何依赖文件，所以目标被认为是最新的而不去执行规则所定义的命令，因此命令 rm 将不会被执行。

为了解决这个问题，需要将目标 clean 显式声明为伪目标，方法是将它作为特殊目标.PHONY 的依赖。如下：

```
.PHONY : clean
```

这样目标 clean 就被显式声明为一个伪目标，无论在当前目录下是否存在 clean 这个文件，输入 make clean 之后，rm 命令都会被执行。而且，当一个目标被声明为伪目标后，make 在执行此规则时不会去试图查找隐含规则来创建它，这也提高了 make 的执行效率。在书写伪目标规则时，首先需要声明目标是一个伪目标，之后才是伪目标的规则定义，目标 clean

的完整书写格式应该如下：

```
.PHONY: clean
clean:
    rm *.o temp
```

一般情况下，一个伪目标不作为另外一个目标的依赖。这是因为当一个目标文件的依赖包含伪目标时，每一次在执行这个规则时伪目标所定义的命令都会被执行。当一个伪目标没有作为任何目标的依赖时，只能通过 make 的命令行来明确指定它为 make 的目标，来执行它所在规则所定义的命令。例如 make clean。

在 Makefile 中，一个伪目标可以有自己的依赖，可以是一个或者多个文件、一个或者多个伪目标。在一个目录下如果需要创建多个可执行程序，可以将所有程序的重建规则在一个 Makefile 中描述。因为 Makefile 中第一个目标是“默认目标”，约定的做法是使用一个称为 all 的伪目标来作为默认目标，它的依赖文件就是那些需要创建的程序。下边就是一个例子：

```
.PHONY: all
all: p1 p2 p3
p1:p1.c
    gcc p1.c -o p1
p2:p2.c
    gcc p2.c -o p2
p3:p3.c
    gcc p3.c -o p3
```

执行 make 时，目标 all 被作为默认目标。为了完成对它的更新，make 会创建（不存在）或者重建（已存在）目标 all 的所有依赖文件（p1、p2 和 p3）。当需要单独更新某一个程序时，可以通过 make 的命令行选项来明确指定需要重建的程序，例如：make p1。

当一个伪目标作为另外一个伪目标的依赖时，make 将其作为另外一个伪目标的子过程来处理，可以这样理解：它作为另外一个伪目标的必须执行的部分，就像 C 语言中的函数调用一样。下边的例子就是这种用法：

```
.PHONY: all
all: p1 p2 p3
p1:
    ls
p2:
    pwd
p3:
    date
```

p1、p2 和 p3 这三个目标有些类似“子函数”，执行目标 p1 时会触发 p1 所定义的命令 ls。可以输入 make p1 和 make p2 和 make p3 命令来达到执行不同命令的目的。

另外 make 命令包含一个内嵌隐含变量 RM，它被定义为：RM＝rm -f。在书写 clean 规则的命令行时可以使用变量 $(RM)来代替 rm，这样可以避免出现一些不必要的麻烦。

7. 标准目标

Makefile 标准目标是一些约定俗成的目标，它们代表一个确定的含义。标准目标包含

以下几种。

(1) all：编译所有的目标。

(2) clean：删除所有被 make 创建的文件。

(3) install：安装已编译好的程序。

(4) print：列出改变过的源文件。

(5) tar：打包备份源程序。

(6) dist：创建一个压缩文件。

(7) TAGS：更新所有的目标。

8. 特殊目标

在 Makefile 中，有一些名字，当它们作为规则的目标时，具有特殊含义。它们是一些特殊的目标，GNU make 所支持的特殊的目标有如下 4 种。

1) .PHONY

目标.PHONY 的所有的依赖被视作伪目标，当使用 make 命令行指定此目标时，伪目标所在规则定义的命令一定会被无条件执行、无论目标文件是否存在。

2) .PRECIOUS

目标.PRECIOUS 的所有依赖文件在 make 过程中会被特殊处理：当命令在执行过程中被中断时，make 不会删除它们。而且如果目标的依赖文件是中间过程文件，同样这些文件不会被删除。

3) .DELETE_ON_ERROR

如果在 Makefile 中存在特殊目标.DELETE_ON_ERROR，make 在执行过程中，如果规则的命令执行错误，将删除已经被修改的目标文件。

4) .SILENT

make 在创建或者重建出现在目标.SILENT 的依赖列表中的文件时，不会打印出所执行的命令。同样，给目标.SILENT 指定命令行是没有意义的。没有任何依赖文件的目标.SILENT 指示 make 在执行过程中不打印任何执行的命令。现行版本 make 支持目标.SILENT 的这种功能和用法是为了和旧版本兼容。在当前版本中如果需要禁命令执行过程的打印，可以使用 make 的命令行参数-s 或者--silent。

特殊目标还有很多，例如：

```
.EXPORT_ALL_VARIABLES
.NOTPARALLEL
.INTERMEDIATE
.SUFFIXES
.DEFAULT
.SECONDARY
.IGNORE
.LOW_RESOLUTION_TIME
```

9. 命令错误

make 命令运行时会检测每个命令的执行返回码，如果命令返回成功，make 会执行下一条命令，否则 make 终止。在 Makefile 的命令行前加一个减号，此时不管命令是否出错，都认为是成功的。例如下面的 Makefile：

```
all: p1 p2 p3
.PHONY: all
p1:p1.c
        -gcc p1.c -o p1
p2:p2.c
        -gcc p2.c -o p2
p3:p3.c
        -gcc p3.c -o p3
```

如果当前目录下的 p1.c 文件有语法错误，那么 p2.c 和 p3.c 仍然会被编译。如果 gcc 命令前没有减号，一旦编译 p1.c 出错，后续两个命令将不会被执行。

10. 通配符

Makefile 中表示文件名时可使用通配符，可使用的通配符有：* 和?。通配符可被用在规则的命令中，它是在命令被执行时由 shell 进行处理。例如 Makefile 的清空过程文件规则：

```
clean:
    rm -f *.o
```

3.7.3 Makefile 的变量

一个完整的 Makefile 包含 5 部分内容：显式规则、隐含规则、变量定义、指示符和注释。本小节介绍一些基本概念。

显式规则：它描述了在何种情况下如何更新一个或者多个目标文件。书写 Makefile 时需要明确地给出目标文件、目标的依赖文件列表以及更新目标文件所需要的命令。

隐含规则：make 根据目标文件自动推导出来的规则。根据目标文件名的后缀，自动产生目标的依赖文件并使用默认的命令来对目标进行更新，需要依靠隐含规则。

变量定义：使用一个字符或字符串代表一段文本串，这个字符或字符串称为变量。当定义了一个变量以后，在需要使用某文本串的位置，就可以用变量来替换。前面例子中的 objects 就是一个变量，用来表示一个.o 文件列表。

Makefile 指示符：指示符指明在 make 程序读取 Makefile 文件过程中所要执行的一个动作。其中包括：①读取某个文件，将其内容作为 Makefile 文件的一部分。②决定处理或者忽略 Makefile 中的某一特定部分。③定义一个多行变量。

注释：Makefile 中，#字符后的内容被作为是注释内容处理。如果此行的第一个非空字符为“#”，那么此行为注释行。注释行的结尾如果存在反斜线“\”，那么下一行也被作为注释行。一般在书写 Makefile 时推荐将注释作为一个独立的行，而不要和 Makefile 的有效行放在一行中书写。当在 Makefile 中需要使用字符 # 时，可以使用反斜线加 #（\#）来实现。

前面的内容中介绍了显式规则和隐含规则，下面介绍 Makefile 的变量。

1. Makefile 变量的基本用法

在 Makefile 中，变量是一个标识符，类似 C 语言中的宏，代表一个文本字符串，即变量的值。例如存在变量 A，它的值用 $(A)来表示。在 Makefile 的目标文件、依赖文件和命令中引用变量的地方，变量会被它的值所取代。在 GNU make 中，变量的定义有两种方式：

递归展开变量和直接展开变量。

1）递归展开变量

这一类型变量通过＝或者使用指示符 define 定义。在引用此变量时进行严格的文本替换，用变量值代替变量出现在引用它的地方。如果此变量定义中存在对其他变量的引用，无论这些其他变量定义在什么地方，都会同时被展开，例如下面的代码：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh
all:
    echo $(foo)
```

执行 make 将会打印出 Huh。变量 foo 的值为变量 bar 的值，变量 bar 的值为变量 ugh 的值，变量 ugh 的值为 Huh，因此变量 foo 的值为 Huh。程序中定义 foo 的时候变量 bar 还没有定义，却可以先使用；定义 bar 时 ugh 还未被定义，也可以先使用，这就是递归展开变量的特点。

2）直接展开式变量

GNU make 还支持另外一种风格的变量，称为“直接展开”式。这种风格的变量使用“:＝”定义。在使用“:＝”定义变量时，变量值中对其他量或者函数的引用在定义变量时被展开（对变量进行替换）。所以变量被定义后就是一个实际需要的文本串，其中不再包含任何变量的引用。因此：

```
x := foo
y := $(x) bar
x := later
```

就等价于：

```
y := foo bar
x := later
```

直接展开式变量和递归展开式变量不同，此风格变量在定义时就完成了对所引用变量和函数的展开，因此不能实现对其后定义变量的引用。

下边来看一个复杂一点的例子。分析一下直接展开式变量定义“:＝”的用法，这里也用到了 make 的 shell 函数和变量 MAKELEVEL（此变量在 make 的递归调用时代表 make 的调用深度）。

在 Makefile 中变量有以下几个特征：

（1）除规则命令行中的变量和函数以外，Makefile 中变量和函数的展开，是在 make 读取 Makefile 文件时进行的，包括使用＝定义和使用指示符 define 定义的变量。

（2）变量可以用来代表一个文件名列表、编译选项列表、程序运行的选项参数列表、搜索源文件的目录列表、编译输出的目录列表等。

（3）变量名是不包括“:”、“＃”、＝、前置空白和尾空白的任何字符串。需要注意的是，尽管在 GNU make 中没有对变量的命名有其他的限制，但定义一个包含除字母、数字和下划线以外字符的变量的做法也是不可取的，因为其他字符可能会在 make 的后续版本中被

赋予特殊含义，并且这样命名的变量对于一些 shell 来说是不能被作为环境变量来使用的。

(4) 变量名是大小写敏感的。变量 foo、Foo 和 FOO 指的是三个不同的变量。Makefile 的传统做法是变量名全部采用大写方式。推荐的做法是对于内部定义的一般变量，如目标文件列表 objects，使用小写方式；而对于一些参数列表，如编译选项 CFLAGS，采用大写方式，但这并不是强制要求的。需要强调一点：如同代码中变量命名风格一样，Makefile 中的变量命名也应保持一致的风格，否则会显得你是一个蹩脚的程序员。

(5) 另外有一些变量名只包含了一个或者很少的几个特殊的字符。称它们为自动化变量，如 $<、$@、$ 、$ * 等。

当定义了一个变量之后，就可以在 Makefile 的很多地方使用这个变量。变量的引用方式为：$(VARIABLE_NAME) 或者 ${ VARIABLE_NAME }。例如：$(foo) 或者 ${foo}，代表取变量 foo 的值。美元符号 $ 在 Makefile 中有特殊的含义，所有在命令或者文件名中使用 $ 时需要用两个美元符号 $ $ 来表示。对一个变量的引用可以在 Makefile 的任何上下文中，包括目标、依赖、命令、绝大多数指示符和新变量的赋值中。下面例子中的变量 objects 保存了所有.o 文件的列表：

```
objects = program.o foo.o utils.o
program : $(objects)
    cc -o program $(objects)
$(objects) : defs.h
```

变量引用的展开过程是严格的文本替换过程，变量代表的字符串被精确地展开在变量被引用的地方。因此规则：

```
foo = c
prog.o : prog.$(foo)
    $(foo)$(foo) -$(foo) prog.$(foo)
```

被展开后就是：

```
prog.o : prog.c
    cc -c prog.c
```

通过这个例子会发现变量的展开过程和 C 语言中宏展开的过程相同，是一个严格的文本替换过程。这个例子中变量的用法很晦涩，举例的目的是为了更清楚地了解变量的展开过程，而不是建议大家按照这样的方式来书写 Makefile，要尽量减少不必要的麻烦。

Makefile 中对一些简单变量的引用，也可以不使用()和{}来标记变量名，而直接使用 $x 的格式来实现，此种用法仅限于变量名为单字符的情况。另外自动化变量也使用这种格式。对于一般多字符变量的引用必须使用括号标记，否则 make 将把变量名的首字母作为变量而不是整个字符串，例如 $PATH 在 Makefile 中实际上代表 $(P)ATH。这一点和 shell 中变量的引用方式不同。shell 中变量的引用可以是 ${xx} 或者 $xx 格式。但在 Makefile 中多字符变量名的引用只能是 $(xx) 或者 ${xx} 格式。

下面介绍如何定义一个空格。使用直接展开变量可以实现将一个前导空格定义在变量值中。一般变量值中的前导空格字符在变量引用和函数调用时被丢弃。利用直接展开式变量在定义时对引用的其他变量进行展开的特点，可以实现在一个变量中包含前导空格并在

引用此变量时对空格加以保护。例如：

```
nullstring :=
space := $(nullstring) #end of the line
```

这里，变量 space 就表示一个空格。在 space 定义行中的注释使得程序员的目的更清晰，注释和变量引用 $(nullstring)之间存在一个空格。通过这种方式就明确地指定了一个空格。这是一个很好的实现方式。通过引用变量 nullstring 标明变量值的开始，采用 # 注释来结束，中间是一个空格字符。

make 对变量进行处理时变量值中尾空格是不被忽略的，因此定义包含一个或者多个空格的变量时，上边的实现就是一个简单并且非常直观的方式。但是需要注意，当定义不包含尾空格的变量时，就不能使用这种方式，否则，注释之前的空格会被作为变量值的一部分。例如下边的做法就是不正确的：

```
dir := /foo/bar      # a bad example of space
```

变量 dir 的值是/foo/bar　　（后面有 4 个空格），这可能并不是期望实现的。如果一个文件以它作为路径来表示 $(dir)/file，那么就大错特错了。

在书写 Makefile 时，推荐将注释书写在独立的行或者多行，防止出现上边例子中的意外情况，而且将注释书写在独立的行也使得 Makefile 更加清晰，便于阅读。对于特殊的定义，例如定义包含一个或者多个空格的变量时进行详细的说明和注释。

GNU make 中，还有一个被称为条件赋值的赋值操作符?=。被称为条件赋值是因为只有此变量在之前没有赋值的情况下才会对这个变量进行赋值。例如：

```
FOO ?= bar
```

其等价于：

```
ifeq ($(origin FOO), undefined)
    FOO = bar
endif
```

含义是：如果变量 FOO 在之前没有定义，就给它赋值 bar。否则不改变它的值。

2. Makefile 变量高级用法

下面讨论关于变量的高级用法。

1）变量的替换引用

对于一个已经定义的变量，可以使用“替换引用”将其值中的后缀字符（串）使用指定的字符（字符串）替换。格式为 $(VAR:A=B)（或者 ${VAR:A=B}），含义为，对由若干个字串组成的变量 VAR，找到其中所有以 A 字符结尾的字串，并把结尾的 A 替换为 B，而对于变量其他部分的 A 字符不进行替换。例如：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

在这个定义中，变量 bar 的值就为“a.c b.c c.c”。如果在变量 foo 中如果存在“o.o”时，那么变量 bar 的值为“a.c b.c c.c o.c”而不是“a.c b.c c.c c.c”。

变量的替换引用其实是函数 patsubst 的一个简化实现。在 GNU make 中同时提供了这两种方式来实现同样的目的，以兼容其他版本 make。

另外一种引用替换的技术使用功能更强大的 patsubst 函数。它的格式和上面 $(VAR:A=B) 的格式相类似，不过在 A 和 B 中需要包含模式字符%。这时它和 $(patsubst A,B, $(VAR))所实现功能相同。例如：

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

这个例子同样使变量 bar 的值为“a.c b.c c.c”。这种格式的替换引用方式比第一种方式更通用。关于模式和其他函数内容在后续小节有所介绍。

2) 变量的套嵌引用

计算变量名是一个比较复杂的过程，仅用在那些复杂的 Makefile 中。通常不需要对它的计算过程进行深入的了解，只要知道当一个被引用的变量名之中含有 $ 时，可得到另外一个值。

一个变量名之中可以包含对其他变量的引用，这种情况称之为“变量的套嵌引用”，先看一个例子：

```
x = y
y = z
a := $($(x))
```

这个例子中，最终定义了 a 的值为 z。来看一下变量的引用过程：首先最内层的变量引用 $(x)被替换为变量名 y，即 $($(x))被替换为 $(y)；之后 $(y)被替换为 z，即 a:=z。这个例子中，a:= $($(x))所引用的变量名不是明确声明的，而是由 $(x)扩展得到，$(x)相对于外层的引用就是套嵌的变量引用。

上个例子是一个 2 层的套嵌引用的例子，具有多层的套嵌引用在 Makefile 中也是允许的。下边来看一个 3 层套嵌引用的例子：

```
x = y
y = z
z = u
a := $($($(x)))
```

这个例子最终是定义了 a 的值为 u。它的扩展过程和上边第一个例子的过程相同。首先 $(x)被替换为 y，则 $($(x))即为 $(y)，$(y)再被替换为 z，所以就有 a:= $(z)；$(z)最后被替换为 u。

再看下面的例子：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

此例最终实现了 a:=Hello 这么一个定义。这里 $($(x))被替换成了 $($(y))，因为 $(y)值是 z，所以，最终结果是：a:= $(z)，也就是“Hello”。

下边的例子使用了 make 的文本处理函数：

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

函数 $(subst 1,2,$(x))的功能是把 x 中的所有 1 字串替换成 2 字串。此例同样实现 a:=Hello。$($($(z)))首先被替换为 $($(y)),之后再次被替换为 $($(subst 1,2,$(x)))。因为 $(x)的值是 variable1,所以上式为 $($(subst 1,2,$(variable1)))。函数处理之后为 $(variable2)。之后对它在进行替换展开。最终,变量 a 的值就是 Hello。从上边的例子可以看到,计算变量名的引用过程存在多层套嵌,也使用了文本处理函数。在书写 Makefile 时,应尽量避免使用套嵌的变量引用。在一些必需的地方,也最好不要使用高于两级的套嵌引用。使用套嵌的变量引用时,如果涉及到递归展开式变量的引用,需要特别注意,一旦处理不当就可能导致递归展开错误,从而导致难以预料的结果。

3. 变量取值

关于变量取值,有以下几点需要注意。

(1) 变量的定义值在长度上没有限制。不过在使用时还是需要根据实际情况考虑,保证计算机上有足够的可用的交换空间来处理一个超常的变量值。变量定义较长时,一个好的做法就是将比较长的行分多个行来书写,除最后一行外,行与行之间使用反斜杠(\)连接,表示一个完整的行。这样的书写方式对 make 的处理不会造成任何影响,便于后期修改维护而且使得 Makefile 更清晰。例如上边的例子就可以这样写：

```
objects = main.o foo.o \
bar.o utils.o
```

(2) 当引用一个没有定义的变量时,make 默认它的值为空。

(3) 一些特殊的变量在 make 中内嵌有固定的值,不过这些变量允许在 Makefile 中显式地重新赋值。

(4) 还存在一些由两个符号组成的特殊变量,称之为自动化变量,它们的值不能在 Makefile 中显式修改。当使用在不同的规则中时,它们会被赋予不同的值。

(5) 如果希望实现这样一个操作,仅对一个之前没有定义过的变量进行赋值。那么可以使用?=来代替=或者:=来实现。

通常,一个变量在定义之后的其他一个地方,可以对其值进行追加。这是非常有用的。可以在定义时给它赋一个基本值,之后根据需要可随时对它的值进行追加。在 Makefile 中使用+=来实现这一操作,例如：

```
objects += another.o
```

这个操作把字符串 another.o 添加到变量 objects 原有值的末尾,使用空格和原有值分开。因此可以看到：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

上边的两个操作之后变量 objects 的值变为：main.o foo.o bar.o utils.o another.o。使用＋＝操作符，相当于：

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

4. override 指示符

通常在执行 make 时，如果通过命令行定义了一个变量，那么它将替代在 Makefile 中出现的同名变量的定义。就是说，对于一个在 Makefile 中使用常规方式（使用＝、:＝或者 define）定义的变量，可以在执行 make 时通过命令行方式重新指定这个变量的值，命令行指定的值将替代出现在 Makefile 中此变量的值。如果不希望命令行指定的变量值替代在 Makefile 中的变量定义，那么需要在 Makefile 中使用指示符 override 来对这个变量进行声明，像下面这样：

```
override VARIABLE = VALUE
```

或者：

```
override VARIABLE := VALUE
```

也可以对变量使用追加方式：

```
override VARIABLE += MORE TEXT
```

对于追加方式需要说明的是：变量在定义时使用了 override，则后续对它的值进行追加时，也需要使用带有 override 指示符的追加方式。否则对此变量值的追加不会生效。

指示符 override 并不是用来调整 Makefile 和执行时命令参数的冲突，其存在的目的是为了使用户可以改变那些使用 make 命令行指定的变量的定义。从另外一个角度来说，就是实现了在 Makefile 中增加或者修改命令行参数的一种机制。有时可能会有这样的需求：可以通过命令行来指定一些附加的编译参数，对一些通用的参数或者必需的编译参数在 Makefile 中指定。对于这种需求，就需要使用指示符 override 来实现。

例如无论命令行指定哪些编译参数，编译时必须打开-g 选项，那么在 Makefile 中编译选项 CFLAGS 应该这样定义：

```
override CFLAGS += -g
```

5. 多行定义

定义变量的另外一种方式是使用 define 指示符。它定义一个包含多行字符串的变量，可以利用它的这个特点实现一个完整命令包的定义。使用 define 定义的命令包可以作为 eval 函数的参数来使用。下面就 define 定义的变量从以下几个方面来讨论。

（1）define 定义变量的语法格式：以指示符 define 开始，endef 结束，之间的所有内容就是所定义变量的值。所要定义的变量名字和指示符 define 在同一行，使用空格分开；指示符所在行的下一行开始一直到 endef 所在行的上一行之间的若干行，是变量值。

```
define two-lines
echo foo
echo $(bar)
```

```
endef
```

如果将变量 two-lines 作为命令执行时，其相当于：

```
two-lines = echo foo; echo $(bar)
```

大家应该对这个命令的执行比较熟悉。它把变量 two-lines 的值作为一个完整的 shell 命令行来处理，保证了变量完整。

(2) 变量的风格：使用 define 定义的变量和使用＝定义的变量一样，属于“递归展开”式的变量，两者只是在语法上不同。因此 define 所定义的变量值中，对其他变量或者函数引用不会在定义变量时进行替换展开，其展开是在 define 定义的变量被展开的同时完成的。

(3) 可以套嵌引用。因为是递归展开式变量，所以在嵌套引用时 $(x)将是变量的值的一部分。

(4) 变量值中可以包含换行符、空格等特殊符号，如果定义中某一行是以 Tab 字符开始时，当引用此变量时这一行会被作为命令行来处理。

(5) 定义变量时可以使用 override 指示符，这样可以防止变量的值被命令行指定的值替代。例如：

```
override define two-lines
foo
$(bar)
endef
```

6. 系统环境变量

make 命令在运行时，系统中的所有环境变量对它都是可见的。在 Makefile 中，可以引用任何已定义的系统环境变量，例如，可以设置一个命名为 CFLAGS 的环境变量，用它来指定一个默认的编译选项，这样在所有的 Makefile 中都直接使用这个变量来对 C 源代码进行编译。通常这种方式是比较安全的，但是它的前提是大家都明白这个变量所代表的含义，没有人在 Makefile 中把它用于其他的用途。当然了，也可以在 Makefile 中根据需要对它进行重新定义。

使用环境变量需要注意以下几点。

(1) 在 Makefile 中对一个变量的定义或者以 make 命令行形式对一个变量的定义，都将覆盖同名的环境变量。注意：它并不改变系统环境变量定义，被修改的环境变量只在 make 执行过程有效。而 make 使用-e 参数时，Makefile 和命令行定义的变量不会覆盖同名的环境变量，make 将使用系统环境变量中这些变量的定义值。

(2) make 的嵌套调用中，所有系统环境变量会被传递给下一级 make。默认情况下，只有环境变量和通过命令行方式定义的变量才会被传递给子 make 进程。在 Makefile 中定义的普通变量传递给子 make 时需要使用 export 指示符来对它声明。

7. 自动化变量

模式规则中，规则的目标和依赖文件代表了一类文件名，规则的命令是对所有这一类文件重建过程的描述，显然，在命令中不能出现具体的文件名，否则模式规则失去意义。那么在模式规则的命令行中该如何表示文件，将是本小节讨论的重点。

假如需要书写一个将.c 文件编译到.o 文件的模式规则，那么该如何为 gcc 书写正确的

源文件名？当然不能使用任何具体的文件名，因为在每一次执行模式规则时源文件名都是不一样的。为了解决这个问题，就需要使用“自动化变量”，自动化变量的取值是根据具体所执行的规则来决定的，取决于所执行规则的目标和依赖文件名。

下面对常用的自动化变量进行说明。

(1) $@：表示规则的目标文件名。如果目标是一个静态库文件，那么它代表这个库的文件名。

(2) $%：当规则的目标文件是一个静态库文件时，代表静态库的一个成员名。

(3) $<：规则的第一个依赖文件名。如果是一个目标文件使用隐含规则来重建，则它代表由隐含规则加入的第一个依赖文件。

(4) $?：所有比目标文件更新的依赖文件列表，空格分割。如果目标是静态库文件名，代表的是库成员(.o 文件)。

(5) $^：规则的所有依赖文件列表，使用空格分隔。如果目标是静态库文件，它所代表的只能是所有库成员(.o 文件)名。一个文件可重复地出现在目标的依赖中，变量 $^只记录它的一次引用情况。就是说变量 $^会去掉重复的依赖文件。

(6) $+：类似 $^，但是它保留了依赖文件中重复出现的文件。主要用在程序链接时库的交叉引用场合。

(7) $*：在模式规则和静态模式规则中，代表“茎”。“茎”是目标模式中%所代表的部分。当文件名中存在目录时，“茎”也包含目录部分。例如：文件“dir/a.foo.b”，当目标的模式为 a.%.b 时，$* 的值为 dir/a.foo。“茎”对于构造相关文件名非常有用。

以上罗列的自动量变量中。其中有四个在规则中代表文件名($@、$<、$%、$*)。而其他三个在规则中代表一个文件名列表。GUN make 中，还可以通过这七个自动化变量来获取一个完整文件名中的目录部分和具体文件名部分。

在讨论自动化变量时，为了和普通变量区别，直接使用了 $<的形式。这种形式仅仅是为了和普通变量进行区别，没有别的目的。其实对于自动化变量和普通变量一样，代表规则第一个依赖文件名的变量名实际上是<，完全可以使用 $(<)来替代 $<。但是在引用自动化变量时通常的做法是 $<，因为自动化变量本身是一个特殊字符。

8. 预定义变量

Makefile 中预定义了一些变量，这些变量读者可以直接使用，包括以下 20 个。

(1) AR：函数库打包程序，可创建静态库.a 文档。默认值是 ar。

(2) AS：汇编程序。默认是 as。

(3) CC：C 编译程序。默认是 cc。

(4) CXX：C++编译程序。默认是 g++。

(5) CO：从 RCS 中提取文件的程序。默认是 co。

(6) CPP：C 程序的预处理器(输出是标准输出设备)。默认是 $(CC) -E。

(7) FC：编译器和预处理 Fortran 和 Ratfor 源文件的编译器。默认是 f77。

(8) GET：从 SCCS 中提取文件程序。默认是 get。

(9) LEX：将 Lex 语言转变为 C 或 Ratfo 的程序。默认是 lex。

(10) PC：Pascal 语言编译器。默认是 pc。

(11) YACC：Yacc 文法分析器(针对于 C 程序)。默认命令是 yacc。

(12) YACCR：Yacc 文法分析器(针对于 Ratfor 程序)。默认是 yacc -r。

(13) MAKEINFO：转换 Texinfo 源文件(.texi)到 Info 文件程序。默认是 makeinfo。

(14) TEX：从 TeX 源文件创建 TeX DVI 文件的程序。默认是 tex。

(15) TEXI2DVI：从 Texinfo 源文件创建 TeX DVI 文件的程序。默认是 texi2dvi。

(16) WEAVE：转换 Web 到 TeX 的程序。默认是 weave。

(17) CWEAVE：转换 C Web 到 TeX 的程序。默认是 cweave。

(18) TANGLE：转换 Web 到 Pascal 语言的程序。默认是 tangle。

(19) CTANGLE：转换 C Web 到 C。默认是 ctangle。

(20) RM：删除命令。默认是 rm -f。

3.7.4　Makefile 的执行

1. 目录搜寻

在一个较大的工程中，一般会将源代码和二进制文件安排在不同的目录中进行区分管理。这种情况下，可以使用 make 提供的目录搜索功能，当工程的目录结构发生变化后，就可以做到不更改 Makefile 的规则，只更改依赖文件的搜索目录。

1) 一般搜索：特殊变量 VPATH

GNU make 可以识别一个特殊变量 VPATH。通过变量 VPATH 可以指定依赖文件的搜索路径，当规则的依赖文件在当前目录不存在时，make 会在此变量所指定的目录中寻找这些依赖文件。

定义变量 VPATH 时，使用空格或者冒号将多个需要搜索的目录分开。make 搜索目录的顺序是按照变量 VPATH 中定义的目录顺序进行的。例如对变量的定义如下：

```
VPATH = src:../headers
```

这样就为所有规则的依赖指定了两个搜索目录，src 和../headers。对于规则 foo:foo.c，如果 foo.c 存在于 src 目录下，此规则等价于 foo:src/foo.c。

通过 VPATH 变量指定的路径在 Makefile 中对所有文件有效。当需要为不同类型的文件指定不同的搜索目录时，需要使用另外一种方式。下一部分内容将会讨论这种更高级的方式。

2) 选择性搜索：关键字 vpath

另一个设置文件搜索路径的方法是使用 make 的 vpath 关键字(全部小写)。它不是一个变量，而是一个 make 的关键字，它所实现的功能和前面提到的 VPATH 变量很类似，但是它更为灵活，可以为不同类型的文件指定不同的搜索目录。它的使用方法有三种。

(1) vpath PATTERN DIRECTORIES 为所有符合模式 PATTERN 的文件指定搜索目录 DIRECTORIES。多个目录使用空格或者冒号分开。类似上一小节的 VPATH 变量。

(2) vpath PATTERN 清除之前为符合模式 PATTERN 的文件设置的搜索路径。

(3) vpath 清除所有已被设置的文件搜索路径。

vapth 使用方法中的 PATTERN 需要包含模式字符%，表示具有相同特征的一类文件，而 DIRECTORIES 则指定了搜索此类文件目录。当规则的依赖文件列表中的文件不能在当前目录下找到时，make 程序将依次在 DIRECTORIES 所描述的目录下寻找此文件。

例如：

```
vpath %.h ../headers
```

其含义是：Makefile 中出现的.h 文件，如果不能在当前目录下找到，则到目录…/headers 下寻找。注意，这里指定的路径仅限于在 Makefile 文件内容中出现的.h 文件，并不能指定源文件中包含的头文件所在的路径，在.c 源文件中所包含的头文件路径需要使用 gcc 的-I 选项来指定。

2. make 的嵌套执行

在一些大的工程中，会把不同模块或是不同功能的源文件放在不同的目录中，可以在每个目录中都书写一个该目录的 Makefile，这有利于使 Makefile 变得更加简洁，而不至于把所有的东西全部写在一个 Makefile 中，导致维护 Makefile 很困难。这个技术就是 make 的嵌套执行，对于模块编译和分段编译有着非常大的好处。如图 3-6 所示，最外层目录的 Makefile 称为总控 Makefile，下层的各个子目录存放有各个源文件及各自的 Makefile，总控 Makefile 负责控制整个程序，通过调用各子目录的 Makefile 来控制程序的编译。

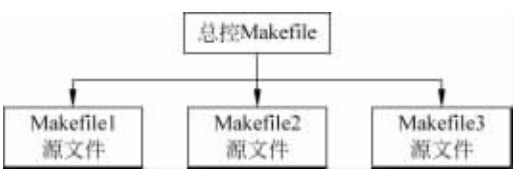

图 3-6 Makefile 层次

在如图 3-7 所示的例子中，当前目录下有总控 Makefile 及源文件 a.c，子目录 subdir1 下有 Makefile 及源文件 b.c，子目录 subdir2 下有 Makefile 及源文件 c.c。通过在当前目录下执行 make 命令，可以分别编译 a.c、b.c 和 c.c 三个文件。总控 Makefile 代码如下所示：

```
.PHONY: both
both: a b c
a:
    gcc a.c -o a
b:
    cd subdir1;make
c:
    cd subdir2;make
```

both 伪目标有三个依赖：a、b 和 c。执行 make 时会分别执行 a、b 和 c 对应的命令。也就是说，会编译 a.c 程序，并进入到目录 subdir1，然后执行此目录下的 Makefile，其内容如下：

```
b:b.c
    gcc b.c -o b
```

然后进入到目录 subdir2，然后执行此目录下的 Makefile，其内容如下：

```
c:c.c
    gcc c.c -o c
```

这个例子说明了嵌套 Makefile 的编写方法。

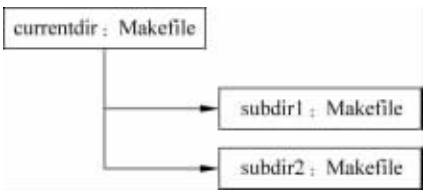

图 3-7　嵌套 Makefile

规则中的命令被执行时，如果是多行命令，那么每一行命令将在一个独立的子 shell 进程中被执行。因此，多行命令之间的执行是相互独立的，相互之间不存在依赖。书写在独立行的一条命令是一个独立的 shell 命令行。因此在一个规则的命令中，命令行 cd 改变目录不会对其后命令的执行产生影响。就是说其后的命令执行的工作目录不会是之前使用 cd 命令进入的那个目录。如果要实现这个目的，就不能把 cd 和其后的命令放在两行来书写，而应该把这两条命令写在一行上，用分号分隔，这样它们才是一个完整的 shell 命令行。

3. Makefile 包含

使用关键字 include 可以在一个 Makefile 中包含其他的 Makefile 文件，这和 C 语言对头文件的包含方式一致。

include 指示符告诉 make 暂停读取当前的 Makefile，而转去读取 include 指定的一个或者多个文件，完成以后再继续当前 Makefile 的读取。Makefile 中指示符 include 书写在独立的一行，其形式如下：

```
include FILENAMES
```

FILENAMES 是 shell 所支持的文件名。

指示符 include 所在的行可以以一个或者多个空格开始，make 程序在处理时将忽略这些空格，切忌不能以 Tab 字符开始。指示符 include 和文件名之间使用空格或者 Tab 键隔开。行尾的空白字符在处理时被忽略。

通常指示符 include 用在以下场合。

(1) 有多个不同的程序，由不同目录下的几个独立的 Makefile 来描述其建立规则。它们需要使用一组共同的变量定义或者模式规则。惯用的做法是将这些共同使用的变量或者模式规则定义在一个文件中，在需要使用的 Makefile 中使用指示符 include 来包含此文件。

(2) 当根据源文件自动产生依赖文件时，可以将自动产生的依赖关系保存在另外一个文件中，主 Makefile 使用指示符 include 包含这些文件。这样的做法比直接在主 Makefile 中追加依赖文件的方法要明智得多。其他版本的 make 已经使用这种方式来处理。

4. 条件执行

条件语句可以根据一个变量的值来控制 make 执行或者忽略 Makefile 的特定部分。条件语句可进行两个不同变量或者变量和常量值的比较。Makefile 中使用条件控制可以提高处理的灵活性和效率。

首先来看一个使用条件判断的例子。对变量 CC 进行判断，其值如果是 gcc 那么在程序链接时使用库 libgnu.so 或者 libgnu.a，否则不链接任何库。Makefile 中的条件判断部分

如下：

```
...
libs_for_gcc = -lgnu
normal_libs =
...
foo: $(objects)
ifeq ($(CC),gcc)
    $(CC) -o foo $(objects) $(libs_for_gcc)
else
    $(CC) -o foo $(objects) $(normal_libs)
endif
...
```

例子中，条件语句中使用到了三个关键字：ifeq、else 和 endif。

(1) ifeq 表示条件语句的开始，并指定了一个比较条件，判断二者是否相等。之后是用圆括号包围的、使用逗号分隔的两个参数，关键字 ifeq 用空格分开。参数中的变量引用在进行变量值比较时被展开。ifeq 之后为条件满足时 make 需要执行的命令，条件不满足时忽略。

(2) else 之后为条件不满足时的执行部分，并非所有的条件语句都需要此部分。

(3) endif 表示一个条件语句的结束，任何一个条件表达式都必须以 endif 结束。

Makefile 中，条件的解析是由 make 来完成的。make 在读取并解析 Makefile 时，根据条件表达式的值选择一个分支，并忽略另一个分支，解析完成后只保留满足条件的分支。例如上面的例子，make 命令的处理过程为：

当变量 CC 的值为 gcc 时，整个条件表达式等效于：

```
foo: $(objects)
    $(CC) -o foo $(objects) $(libs_for_gcc)
```

当变量 CC 值不等于 gcc 时，等效于：

```
foo: $(objects)
    $(CC) -o foo $(objects) $(normal_libs)
```

上面的例子，还存在一种更简洁的实现方式：

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
libs= $(libs_for_gcc)
else
libs= $(normal_libs)
endif
foo: $(objects)
    $(CC) -o foo $(objects) $(libs)
```

条件判断的语法很简单。不包含 else 分支的条件判断语句的语法格式为：

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
```

```
endif
```

表达式中 TEXT-IF-TRUE 可以是若干行任何文本，当条件为真时它就是需要执行的一部分。当条件为假时，不会执行到。

包含 else 分支的语法格式为：

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
else
TEXT-IF-FALSE
endif
```

表示如果条件为真，则将 TEXT-IF-TRUE 作为 Makefile 的一部分，否则将 TEXT-IF-FALSE 作为 Makefile 的一部分。和 TEXT-IF-TRUE 一样，TEXT-IF-FALSE 可以是若干行任何文本。

条件判断语句中 CONDITIONAL-DIRECTIVE 对于上边的两种格式都相同，可以是以下 4 种用于测试不同条件的关键字。

(1) 关键字 ifeq：用来判断参数是否相等。

(2) 关键字 ifneq：用来判断参数是否不相等。

(3) 关键字 ifdef：用来判断一个变量是否已经定义。

(4) 关键字 ifndef：与关键字 ifdef 的含义正好相反。

在 CONDITIONAL-DIRECTIVE 这一行上，可以以若干个空格开始，make 处理时会忽略这些空格。但不能以 Tab 字符作为开始，否则就被认为是命令。else 和 endif 也是条件判断语句的一部分，书写时没有任何参数，可以以多个空格开始，同样不能以 Tab 字符开始，可以多个空格或 Tab 字符结束，行尾可以有注释内容。

5. Makefile 的命令行选项

make 执行时可以附加一些选项，常用的有以下几种。

(1) n：只检查流程，不执行。

(2) t：只更新目标文件访问时间，不重新编译。

(3) B：认为所有目标都需要重新编译。

(4) C：指定 Makefile 文件的目录。

(5) e：环境变量中的值覆盖文件中的值。

(6) p：输出 Makefile 中的所有信息。

(7) r：禁止 make 使用任何隐含规则。

(8) s：命令运行时不显示任何输出。

(9) v：输出 make 的版本信息。

(10) w：输出运行 make 前后的信息。

3.7.5　make 内嵌函数

GNU make 的函数提供了处理文件名、变量、文本和命令的方法。把需要处理的文本作为函数的参数就可以在需要的地方调用函数来处理指定的文本，执行时，函数在被调用的地方替换为它的处理结果。

1. 函数的调用语法

GNU make 函数的调用格式类似于变量的引用，以 $ 开始表示一个引用。语法格式如下：

```
$(FUNCTION ARGUMENTS)
```

或者：

```
${FUNCTION ARGUMENTS}
```

对于函数调用的格式有以下几点说明。

(1) 调用语法格式中，FUNCTION 是需要调用的函数名，它应该是 make 内嵌的函数名，对于用户自己的函数需要通过 make 的 call 函数来间接调用。

(2) ARGUMENTS 是函数的参数，参数和函数名之间使用若干个空格或者 Tab 字符分割，建议使用一个空格，这样不仅在书写上比较直观，更重要的是当不能确定是否可以使用 Tab 的时候，避免不必要的麻烦。如果存在多个参数，参数之间使用逗号，分开。

(3) 以 $ 开头，使用成对的圆括号或花括号把函数名和参数括起。参数中存在变量或函数的引用时，对它们所使用的圆括号或花括号和引用函数的相同，不使用两种不同的括号。推荐在变量引用和函数引用中统一使用圆括号，这样在使用 vim 编辑器书写 Makefile 时，使用圆括可以亮度显示 make 的内嵌函数名，避免函数名的拼写错误。在 Makefile 中应该这样来书写 $(sort $(x))，而不是 $(sort ${x})或其他。

(4) 函数处理参数时，参数中如果存在对其他变量或函数的引用，首先对这些引用进行展开得到参数的实际内容。而后才对它们进行处理。参数的展开顺序是按照参数的先后顺序来进行的。

(5) 书写时，前导空格会被忽略，函数的参数不能出现逗号和空格，因为逗号被作为多个参数的分隔符。当有逗号或者空格作为函数的参数时，需要先把它们赋值给一个变量，然后在函数的参数中引用这个变量。

下面介绍一些常用的 make 内置函数。

2. 文本处理函数

1) 字符串替换函数：subst

```
$(subst FROM,TO,TEXT)
```

函数功能：把字串 TEXT 中的 FROM 字符替换为 TO。

返回值：替换后的新字符串。

示例：

```
$(subst ee,EE,feet on the street)
```

替换 feet on the street 中的 ee 为 EE，结果得到字符串 fEEt on the strEEt。

2) 模式替换函数：patsubst

```
$(patsubst PATTERN,REPLACEMENT,TEXT)
```

函数功能：搜索 TEXT 中以空格分开的单词，将符合模式 PATTERN 的单词按照 REPLACEMENT 进行模式替换。参数 PATTERN 中可以使用模式通配符%来代表一个

单词中的若干字符。如果参数 REPLACEMENT 中也包含一个%，那么 REPLACEMENT 中的%将是 TATTERN 中的那个%所代表的字符串。在 TATTERN 和 REPLACEMENT 中，只有第一个%被作为模式字符来处理，之后出现的不再作模式字符。在参数中如果需要将第一个出现的%作为字符本身而不作为模式字符时，可使用反斜杠\进行转义处理。

返回值：替换后的新字符串。

函数说明：参数 TEXT 单词之间的多个空格在处理时被合并为一个空格，并忽略前导和结尾空格。

示例：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

把字串 x.c.c bar.c 中以.c 结尾的单词替换成以.o 结尾的字符。函数的返回结果是 x.c.o 。

3）去空格函数：strip

```
$(strip STRINT)
```

函数功能：去掉字串 STRINT 开头和结尾的空字符，并将其中多个连续空字符合并为一个空字符。

返回值：无前导和结尾空字符、使用单一空格分割的多单词字符串。

函数说明：空字符包括空格、Tab 等不可显示字符。

示例：

```
STR =          a     b c
LOSTR = $(strip $(STR))
```

结果是 a b c。

strip 函数经常用在条件判断语句的表达式中，可以确保表达式的可靠和健壮。

4）查找字符串函数：findstring

```
$(findstring FIND,IN)
```

函数功能：搜索字串 IN，查找 FIND 字串。

返回值：如果在 IN 之中存在 FIND，则返回 FIND，否则返回空。

函数说明：字串 IN 之中可以包含空格、Tab 键。搜索过程是严格的文本匹配。

示例：

```
$(findstring a,a b c)
$(findstring a,b c)
```

第一个函数结果是 a，第二个为空字符。

5）过滤函数：filter

```
$(filter PATTERN…,TEXT)
```

函数功能：过滤掉字串 TEXT 中所有不符合模式 PATTERN 的单词，保留所有符合此模式的单词。模式中一般需要包含模式字符%，存在多个模式时，模式表达式之间使用空格

分割。

返回值：空格分割的 TEXT 字串中所有符合模式 PATTERN 的字串。

函数说明：filter 函数可以用来去除一个变量中的某些字符串，下边的例子中用到了此函数。

示例：

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
     cc $(filter %.c %.s, $(sources)) -o foo
```

使用 $(filter %.c %.s, $(sources))的返回值给 cc 来编译生成目标 foo，函数返回值为 foo.c bar.c baz.s。

6）反过滤函数：filter-out

```
$(filter-out PATTERN..., TEXT)
```

函数功能：和 filter 函数实现的功能相反。过滤掉字串 TEXT 中所有符合模式 PATTERN 的单词，保留所有不符合此模式的单词。存在多个模式时，模式表达式之间使用空格分割。

返回值：空格分割的 TEXT 字串中所有不符合模式 PATTERN 的字串。

函数说明：filter-out 函数可以用来去除一个变量中的某些字符串，实现和 filter 函数相反的功能。

示例：

```
    objects=main1.o foo.o main2.o bar.o
    mains=main1.o main2.o
$(filter-out $(mains), $(objects))
```

实现了去除变量 objects 中 mains 定义的字串（文件名）功能。它的返回值为 foo.o bar.o。

7）排序函数：sort

```
$(sort LIST)
```

函数功能：给字串 LIST 中的单词以首字母为准进行升序排序，并去掉重复的单词。

返回值：空格分割的没有重复单词的字串。

函数说明：排序和去除字串中的重复单词。

示例：

```
$(sort foo bar lose foo)
```

返回值为：bar foo lose。

8）取单词函数：word

```
$(word N, TEXT)
```

函数功能：取字串 TEXT 中第 N 个单词（N 的值从 1 开始）。

返回值：返回字串 TEXT 中第 N 个单词。

函数说明：如果 N 值大于字串 TEXT 中单词的数目，返回空字符串。如果 N 为 0，出错。

示例：

```
$(word 2, foo bar baz)
```

返回值为 bar。

9）取字串函数：wordlist

```
$(wordlist S,E,TEXT)
```

函数功能：从字串 TEXT 中取出从 S 开始到 E 的单词串。S 和 E 表示单词在字串中位置的数字。

返回值：字串 TEXT 中从第 S 到 E 的单词字串。

函数说明：S 和 E 都是从 1 开始的数字。当 S 比 TEXT 中的字数大时，返回空。如果 E 大于 TEXT 字数，返回从 S 开始，到 TEXT 结束的单词串。如果 S 大于 E，返回空。

示例：

```
$(wordlist 2, 3, foo bar baz)
```

返回值为：bar baz。

10）统计单词数目函数：words

```
$(words TEXT)
```

函数功能：计算字串 TEXT 中单词的数目。

返回值：TEXT 字串中的单词数。

示例：

```
$(words, foo bar)
```

返回值是 2。所以字串 TEXT 的最后一个单词就是：$(word $(words TEXT), TEXT)。

11）取首单词函数：firstword

```
$(firstword NAMES…)
```

函数功能：取字串 NAMES…中的第一个单词。

返回值：字串 NAMES…的第一个单词。

函数说明：NAMES 被认为是使用空格分割的多个单词（名字）的序列。函数忽略 NAMES…中除第一个单词以外的所有的单词。

示例：

```
$(firstword,foo bar)
```

返回值为 foo。函数 firstword 实现的功能等效于 $(word 1, NAMES…)。

以上 11 个函数是 make 内嵌的文本处理函数，书写 Makefile 时可搭配使用来实现复杂功能。下面使用这些函数来实现一个实际应用，用到了函数 subst 和 patsbust。Makefile

中可以使用变量 VPATH 来指定搜索路径，源代码所包含的头文件的搜索路径需要使用 gcc 的-I 参数来指定。下面是 Makefile 的片段：

```
...
VPATH = src:../includes
override CFLAGS += $(patsubst %,-I%, $(subst :, , $(VPATH)))
...
```

那么第二条语句所实现的功能就是 CFLAGS += -Isrc -I../includes。

3. 文件名处理函数

GNU make 除支持若干文本处理函数之外，还支持一些针对文件名的处理函数，这些函数主要用来对一系列空格分割的文件名进行转换。函数对作为参数的一组文件名按照一定方式进行处理并返回空格分割的多个文件名序列。

1）取目录函数：dir

```
$(dir NAMES…)
```

函数功能：从文件名序列 NAMES…中取出各个文件名的目录部分。文件名的目录部分为包含在文件名中的最后一个斜线“/”之前的部分，包括斜线。

返回值：空格分割的文件名序列 NAMES…中每一个文件的目录部分。

函数说明：如果文件名中没有斜线，认为此文件为当前目录下的文件。

示例：

```
$(dir src/foo.c hacks)
```

返回值为 src/ ./。

2）取文件名函数：notdir

```
$(notdir NAMES…)
```

函数功能：从文件名序列 NAMES…中取出非目录部分。

返回值：文件名序列 NAMES…中每一个文件的非目录部分。

函数说明：如果 NAMES…中存在不包含斜线的文件名，则不改变这个文件名。

示例：

```
$(notdir src/foo.c hacks)
```

返回值为：foo.c hacks。

3）取后缀函数：suffix

```
$(suffix NAMES…)
```

函数功能：从文件名序列 NAMES…中取出各个文件名的后缀。后缀是文件名中最后一个以点.开始的部分，包含点本身；如果文件名中不包含一个点号，则为空。

返回值：以空格分割的文件名序列 NAMES…中每一个文件的后缀序列。

函数说明：NAMES…是多个文件名时，返回值是多个以空格分割的单词序列。如果文件名没有后缀部分，则返回空。

示例：

```
$(suffix src/foo.c src-1.0/bar.c hacks)
```

返回值为.c .c。

4）取前缀函数：basename

```
$(basename NAMES…)
```

函数功能：从文件名序列 NAMES…中取出各个文件名的前缀部分。前缀部分指的是文件名中最后一个点号之前的部分。

返回值：空格分割的文件名序列 NAMES…中各个文件的前缀序列。如果文件没有前缀，则返回空字串。

函数说明：如果 NAMES…中包含没有后缀的文件名，此文件名不改变。如果一个文件名中存在多个点号，则返回值为此文件名的最后一个点号之前的文件名部分。

示例：

```
$(basename src/foo.c src-1.0/bar.c /home/jack/.font.cache-1 hacks)
```

返回值为：src/foo src-1.0/bar /home/jack/.font hacks。

5）加后缀函数：addsuffix。

```
$(addsuffix SUFFIX,NAMES…)
```

函数功能：为 NAMES…中的每一个文件名添加后缀 SUFFIX。参数 NAMES…为空格分割的文件名序列，将 SUFFIX 追加到此序列的每一个文件名的末尾。

返回值：以单空格分割的添加了后缀 SUFFIX 的文件名序列。

示例：

```
$(addsuffix .c,foo bar)
```

返回值为 foo.c bar.c。

6）加前缀函数：addprefix

```
$(addprefix PREFIX,NAMES…)
```

函数功能：为 NAMES…中的每一个文件名添加前缀 PREFIX。参数 NAMES…是空格分割的文件名序列，将 SUFFIX 添加到此序列的每一个文件名之前。

返回值：以单空格分割的添加了前缀 PREFIX 的文件名序列。

示例：

```
$(addprefix src/,foo bar)
```

返回值为 src/foo src/bar。

7）单词连接函数：join

```
$(join LIST1,LIST2)
```

函数功能：将字串 LIST1 和字串 LIST2 各单词进行对应连接，即将 LIST2 中的第一个单词追加 LIST1 第一个单词字后合并为一个单词；将 LIST2 中的第二个单词追加到 LIST1 的第一个单词之后并合并为一个单词，……依次类推。

返回值：单空格分割的合并后的单词序列。

函数说明：如果 LIST1 和 LIST2 中的单词数目不一致时，两者中多余部分将被作为返回序列的一部分。

示例 1：

```
$(join a b , .c .o)
```

返回值为：a.c b.o。

示例 2：

```
$(join a b c , .c .o)
```

返回值为：a.c b.o c。

8）获取匹配模式文件名函数：wildcard

```
$(wildcard PATTERN)
```

函数功能：列出当前目录下所有符合模式 PATTERN 格式的文件名。

返回值：空格分割的、存在于当前目录下的所有符合模式 PATTERN 的文件名。

函数说明：PATTERN 使用 shell 可识别的通配符，包括？（单字符）、*（多字符）等。

示例：

```
$(wildcard *.c)
```

返回值为当前目录下所有.c 源文件列表。

4. foreach 函数

函数 foreach 不同于其他函数，它是一个循环函数，类似于 Linux 的 shell 中的 for 语句。foreach 函数的语法格式为：

```
$(foreach VAR,LIST,TEXT)
```

函数功能：把 LIST 中的单词逐一取出，送入 VAR 指定的变量中，然后再执行 TEXT 表达式，每执行一次 TEXT 返回一个字串，所有字串用空格连接起来就是函数的返回值。

返回值：用空格分割的，经表达式 TEXT 多次计算的结果。

示例：

```
names:=a b c d
$(foreach  n,  $(names),  $(n).o)
```

返回值：a.o b.o c.o d.o。

5. if 函数

函数 if 提供了一个在函数上下文中实现条件判断的功能，就像 make 所支持的条件语句 ifeq 一样。函数的语法格式为：

```
$(if CONDITION,THEN-PART[,ELSE-PART])
```

函数功能：第一个参数 CONDITION，在函数执行时忽略其前导和结尾空字符，如果包含对其他变量或者函数的引用则进行展开。如果 CONDITION 的展开结果非空，则条件为

真，将第二个参数 THEN-PATR 作为函数的计算表达式；CONDITION 的展开结果为空，将第三个参数 ELSE-PART 作为函数的表达式，函数的返回结果为有效表达式的计算结果。

返回值：根据条件决定函数的返回值是第一个或者第二个参数表达式的计算结果。当不存在第三个参数 ELSE-PART，并且 CONDITION 展开为空，函数返回空。

函数说明：函数的条件表达式 CONDITION 决定了函数的返回值只能是 THEN-PART 或者 ELSE-PART 两个之一的计算结果。

函数示例：

```
SUBDIR += $(if $(SRC_DIR) $(SRC_DIR),/home/src)
```

函数的结果是：如果 SRC_DIR 变量值不为空，则将变量 SRC_DIR 指定的目录作为一个子目录；否则将目录/home/src 作为一个子目录。

6. origin 函数

函数 origin 和其他函数不同，它的功能不是操作变量，而只是获取此变量相关的信息，说明这个变量的出处。函数的语法为：

```
$(origin VARIABLE)
```

函数功能：函数 origin 查询变量参数 VARIABLE 的出处。

函数说明：VARIABLE 是一个变量名而不是一个变量的引用，因此通常不包含 $。

返回值：返回 VARIABLE 的定义方式，用字符串表示。函数的返回情况有以下几种。

(1) undefined：变量 VARIABLE 没有被定义。

(2) default：变量 VARIABLE 是内嵌变量。如 CC、MAKE、RM 等变量。如果在 Makefile 中重新定义这些变量，函数返回值将相应发生变化。

(3) environment：变量 VARIABLE 是系统环境变量，并且 make 没有使用命令行选项-e。

(4) environment override：变量 VARIABLE 是一个系统环境变量，并且 make 使用了命令行选项-e，即 Makefile 中存在一个同名的变量定义，使用 make -e 时环境变量值替代了文件中的变量定义。

(5) file：变量 VARIABLE 在某一个 Makefile 文件中定义。

(6) command line：变量 VARIABLE 在命令行中定义。

(7) override：变量 VARIABLE 在 Makefile 文件中定义并使用 override 指示符声明。

(8) automatic：变量 VARIABLE 是自动化变量。

7. shell 函数

shell 函数不同于除 wildcard 函数之外的其他函数，make 可以使用它来和外部通信。

函数功能：函数 shell 所实现的功能和 shell 中的引用(")相同，实现对命令的扩展，这意味着需要一个 shell 命令作为此函数的参数，函数的返回结果是此命令在 shell 中的执行结果。make 仅仅对它的返回结果进行处理，将返回结果中的所有换行符\n 或者一对\n\r 替换为单空格，并去掉末尾的回车符号\n 或者\n\r。进行函数展开时，它所调用的命令得到执行。除对它的引用出现在规则的命令行和递归变量的定义中以外，其他绝大多数情况下，make 是在读取解析 Makefile 时完成对函数 shell 的展开。

返回值：函数 shell 的参数在 shell 环境中的执行结果。

函数说明：函数本身的返回值是其参数的执行结果，没有进行任何处理。对结果的处理是由 make 进行的。当对函数的引用出现在规则的命令行中，命令行在执行时函数才被展开。展开时函数参数的执行是在另外一个 shell 进程中完成的，因此需要对出现在规则命令行的多级 shell 函数引用谨慎处理，否则会影响效率。

示例：

```
contents := $(shell cat foo)
```

将变量 contents 赋值为文件 foo 的内容，文件中的换行符在变量中使用空格代替。

3.7.6 make 的常见错误信息

make 执行过程中所产生的错误并不都是致命的，特别是在命令行之前存在-、或者 make 使用-k 选项执行时。make 执行过程的致命错误都带有前缀字符串 *** 。

错误信息都有前缀，一种是执行程序名作为错误前缀，通常是 make；另外一种是当 Makefile 本身存在语法错误无法被 make 解析并执行时，前缀包含了 Makefile 文件名和出现错误的行号。

在下述的错误列表中，省略了普通前缀。

```
[FOO] Error NN
[FOO] signal description
```

这类错误并不是 make 的真正错误，它表示 make 检测到其所调用的程序返回一个非零状态(Error NN)，或者此程序携带某种信号，以非正常方式退出。

如果错误信息中没有附加 *** 字符串，则表示子过程的调用失败，如果 Makefile 中此命令有前缀-，make 会忽略这个错误。

```
missing separator. Stop.
missing separator (did you mean TAB instead of 8 spaces?). Stop.
```

不可识别的命令行，make 在读取 Makefile 过程中不能解析其中包含的内容。GNU make 在读取 Makefile 时根据各种分隔符(:，=，Tab 字符等)来识别 Makefile 的每一行内容。这些错误意味着 make 不能发现一个合法的分隔符。

出现这些错误信息的可能的原因是 Makefile 中的命令之前使用了 4 个或者 8 个空格代替了 Tab 字符。这种情况，将产生上述的第二种形式产生错误信息。

```
commands commence before first target. Stop.
missing rule before commands. Stop.
```

Makefile 可能是以命令行开始：以 Tab 字符开始，但不是一个合法的命令行，例如，可能是对一个变量的赋值。命令行必须和规则一一对应。

产生第二种的错误的原因可能是一行的第一个非空字符为分号，make 会认为此处遗漏了规则的 target：prerequisite 部分。

```
No rule to make target 'XXX'.
No rule to make target 'XXX', needed by 'yyy'.
```

无法为重建目标 XXX 找到合适的规则，包括明确规则和隐含规则。

修正这个错误的方法是：在 Makefile 中添加一个重建目标的规则。其他可能导致这些错误的原因是 Makefile 中文件名拼写错误，或者是依赖文件出现了问题。

No targets specified and no makefile found. Stop.
No targets. Stop.

第一个错误表示在命令行中没有指定需要重建的目标，并且 make 不能读入任何 Makefile 文件。第二个错误表示能够找到 Makefile 文件，但没有默认目标或者没有在命令行中指出需要重建的目标。这种情况下，make 什么也不做。

Makefile 'XXX' was not found.
Included makefile 'XXX' was not found.

没有使用-f 指定 Makefile 文件，make 不能在当前目录下找到默认 Makefile(makefile、Makefile 或者 GNUmakefile)。使用-f 指定文件，但不能读取这个指定的 Makefile 文件。

warning: overriding commands for target 'XXX'
warning: ignoring old commands for target 'XXX'

对同一目标 XXX 存在一个以上的重建命令。GNU make 规定：当同一个文件作为多个规则的目标时，只能有一个规则定义重建它的命令。如果为一个目标多次指定了相同或者不同的命令，就会产生第一个警告；第二个警告信息说新指定的命令覆盖了上一次指定的命令。

Circular XXX <- YYY dependency dropped.

规则的依赖关系产生了循环：目标 XXX 的依赖文件为 YYY，而依赖 YYY 的依赖列表中又包含 XXX。

Recursive variable 'XXX' references itself (eventually). Stop.

递归展开式变量 XXX 在替换展开时，引用它自身。无论对于直接展开式变量或追加定义，这都是不允许的。

Unterminated variable reference. Stop.

变量或者函数引用语法不正确，没有使用完整的括号。

insufficient arguments to function 'XXX'. Stop.

函数 XXX 引用时参数数目不正确，函数缺少参数。