第二十四章 编译程序

In this chapter, we will look at how to build programs by compiling source code. The availability of source code is the essential freedom that makes Linux possible. The entire ecosystem of Linux development relies on free exchange between developers. For many desktop users, compiling is a lost art. It used to be quite common, but today, distribution providers maintain huge repositories of precompiled binaries, ready to download and use. At the time of this writing, the Debian repository (one of the largest of any of the distributions) contains almost 23,000 packages.

在这一章中，我们将看一下如何通过编译源代码来创建程序。源代码的可用性是至关重要的自由，从而使得 Linux 成为可能。 整个 Linux 开发生态圈就是依赖于开发者之间的自由交流。对于许多桌面用户来说，编译是一种失传的艺术。以前很常见， 但现在，由系统发行版提供商维护巨大的预编译的二进制仓库，准备供用户下载和使用。在写这篇文章的时候， Debian 仓库（最大的发行版之一）包含了几乎23,000个预编译的包。

So why compile software? There are two reasons:

那么为什么要编译软件呢？ 有两个原因：

1. Availability. Despite the number of precompiled programs in distribution repositories, some distributions may not include all the desired applications. In this case, the only way to get the desired program is to compile it from source.
2. Timeliness. While some distributions specialize in cutting edge versions of programs, many do not. This means that in order to have the very latest version of a program, compiling is necessary.
3. 可用性。尽管系统发行版仓库中已经包含了大量的预编译程序，但是一些发行版本不可能包含所有期望的应用。 在这种情况下，得到所期望程序的唯一方式是编译程序源码。
4. 及时性。虽然一些系统发行版专门打包前沿版本的应用程序，但是很多不是。这意味着， 为了拥有一个最新版本的程序，编译是必需的。

Compiling software from source code can become very complex and technical; well beyond the reach of many users. However, many compiling tasks are quite easy and involve only a few steps. It all depends on the package. We will look at a very simple case in order to provide an overview of the process and as a starting point for those who wish to undertake further study.

从源码编译软件可以变得非常复杂且具有技术性；许多用户难以企及。然而，许多编译任务是 相当简单的，只涉及到几个步骤。这都取决于程序包。我们将看一个非常简单的案例， 为的是给大家提供一个对编译过程的整体认识，并为那些愿意进一步学习的人们构筑一个起点。

We will introduce one new command:

我们将介绍一个新命令：

* make – Utility to maintain programs
* make - 维护程序的工具

**什么是编译？**

Simply put, compiling is the process of translating source code (the human-readable description of a program written by a programmer) into the native language of the computer’s processor.

简而言之，编译就是把源码（一个由程序员编写的人类可读的程序描述）翻译成计算机处理器的母语的过程。

The computer’s processor (or CPU) works at a very elemental level, executing programs in what is called machine language. This is a numeric code that describes very small operations, such as “add this byte,” “point to this location in memory,” or “copy this byte.”

计算机处理器（或 CPU）工作在一个非常基本的水平，执行用机器语言编写的程序。这是一种数值编码，描述非常小的操作， 比如“加这个字节”，“指向内存中的这个位置”，或者“复制这个字节”。

Each of these instructions is expressed in binary (ones and zeros). The earliest computer programs were written using this numeric code, which may explain why programmers who wrote it were said to smoke a lot, drink gallons of coffee, and wear thick glasses.This problem was overcome by the advent of assembly language, which replaced the numeric codes with (slightly) easier to use character mnemonics such as CPY (for copy) and MOV (for move). Programs written in assembly language are processed into machine language by a program called an assembler. Assembly language is still used today for certain specialized programming tasks, such as device drivers and embedded systems.

这些指令中的每一条都是用二进制表示的（1和0）。最早的计算机程序就是用这种数值编码写成的，这可能就 解释了为什么编写它们的程序员据说吸很多烟，喝大量咖啡，并带着厚厚的眼镜。这个问题克服了，随着汇编语言的出现， 汇编语言代替了数值编码（略微）简便地使用助记符，比如 CPY（复制）和 MOV（移动）。用汇编语言编写的程序通过 汇编器处理为机器语言。今天为了完成某些特定的程序任务，汇编语言仍在被使用，例如设备驱动和嵌入式系统。

We next come to what are called high-level programming languages. They are called this because they allow the programmer to be less concerned with the details of what the processor is doing and more with solving the problem at hand. The early ones (developed during the 1950s) included FORTRAN (designed for scientific and technical tasks) and COBOL (designed for business applications). Both are still in limited use today.

下一步我们谈论一下什么是所谓的高级编程语言。之所以这样称呼它们，是因为它们可以让程序员少操心处理器的 一举一动，而更多关心如何解决手头的问题。早期的高级语言（二十世纪60年代期间研发的）包括 FORTRAN（为科学和技术问题而设计）和 COBOL（为商业应用而设计）。今天这两种语言仍在有限的使用。

While there are many popular programming languages, two predominate. Most programs written for modern systems are written in either C or C++. In the examples to follow, we will be compiling a C program.

虽然有许多流行的编程语言，两个占主导地位。大多数为现代系统编写的程序，要么用 C 编写，要么是用 C++ 编写。 在随后的例子中，我们将编写一个 C 程序。

Programs written in high-level programming languages are converted into machine language by processing them with another program, called a compiler. Some compilers translate high-level instructions into assembly language and then use an assembler to perform the final stage of translation into machine language.

用高级语言编写的程序，经过另一个称为编译器的程序的处理，会转换成机器语言。一些编译器把 高级指令翻译成汇编语言，然后使用一个汇编器完成翻译成机器语言的最后阶段。

A process often used in conjunction with compiling is called linking. There are many common tasks performed by programs. Take, for instance, opening a file. Many programs perform this task, but it would be wasteful to have each program implement its own routine to open files. It makes more sense to have a single piece of programming that knows how to open files and to allow all programs that need it to share it. Providing support for common tasks is accomplished by what are called libraries. They contain multiple routines, each performing some common task that multiple programs can share. If we look in the /lib and /usr/lib directories, we can see where many of them live. A program called a linker is used to form the connections between the output of the compiler and the libraries that the compiled program requires. The final result of this process is the executable program file, ready for use.

一个称为链接的过程经常与编译结合在一起。有许多程序执行的常见任务。以打开文件为例。许多程序执行这个任务， 但是让每个程序实现它自己的打开文件功能，是很浪费资源的。更有意义的是，拥有单独的一段知道如何打开文件的程序， 并允许所有需要它的程序共享它。对常见任务提供支持由所谓的库完成。这些库包含多个程序，每个程序执行 一些可以由多个程序共享的常见任务。如果我们看一下 /lib 和 /usr/lib 目录，我们可以看到许多库定居在那里。 一个叫做链接器的程序用来在编译器的输出结果和要编译的程序所需的库之间建立连接。这个过程的最终结果是 一个可执行程序文件，准备使用。

**所有的程序都是可编译的吗？**

No. As we have seen, there are programs such as shell scripts that do not require compiling. They are executed directly. These are written in what are known as scripting or interpreted languages. These languages have grown in popularity in recent years and include Perl, Python, PHP, Ruby, and many others.

不是。正如我们所看到的，有些程序比如 shell 脚本就不需要编译。它们直接执行。 这些程序是用所谓的脚本或解释型语言编写的。近年来，这些语言变得越来越流行，包括 Perl， Python，PHP，Ruby，和许多其它语言。

Scripted languages are executed by a special program called an interpreter. An interpreter inputs the program file and reads and executes each instruction contained within it. Ingeneral, interpreted programs execute much more slowly than compiled programs. This is because that each source code instruction in an interpreted program is translated every time it is carried out, whereas with a compiled program, a source code instruction is only translated once, and this translation is permanently recorded in the final executable file.

脚本语言由一个叫做解释器的特殊程序执行。一个解释器输入程序文件，读取并执行程序中包含的每一条指令。 通常来说，解释型程序执行起来要比编译程序慢很多。这是因为每次解释型程序执行时，程序中每一条源码指令都需要翻译， 而一个编译程序，一条源码指令只翻译一次，翻译后的指令会永久地记录到最终的执行文件中。

So why are interpreted languages so popular? For many programming chores, the results are “fast enough,” but the real advantage is that it is generally faster and easier to develop interpreted programs than compiled programs. Programs are usually developed in a repeating cycle of code, compile, test. As a program grows in size, the compilation phase of the cycle can become quite long. Interpreted languages remove the compilation step and thus speed up program development.

那么为什么解释型程序这样流行呢？对于许多编程任务来说，原因是“足够快”，但是真正的优势是一般来说开发解释型程序 要比编译程序快速且容易。通常程序开发需要经历一个不断重复的写码，编译，测试周期。随着程序变得越来越大， 编译阶段会变得相当耗时。解释型语言删除了编译步骤，这样就加快了程序开发。

**编译一个 C 语言**

Let’s compile something. Before we do that however, we’re going to need some tools like the compiler, the linker, and make. The C compiler used almost universally in the Linux environment is called gcc (GNU C Compiler), originally written by Richard Stallman. Most distributions do not install gcc by default. We can check to see if the compiler is present like this:

让我们编译一些东西。在我们行动之前，然而我们需要一些工具，像编译器，链接器，还有 make。 在 Linux 环境中，普遍使用的 C 编译器叫做 gcc（GNU C 编译器），最初由 Richard Stallman 写出来的。 大多数 Linux 系统发行版默认不安装 gcc。我们可以这样查看该编译器是否存在：

[me@linuxbox ~]$ which gcc

/usr/bin/gcc

The results in this example indicate that the compiler is installed.

在这个例子中的输出结果表明安装了 gcc 编译器。

Tip: Your distribution may have a meta-package (a collection of packages) for soft- ware development. If so, consider installing it if you intend to compile programs on your system. If your system does not provide a meta-package, try installing the gcc and make packages. On many distributions, this is sufficient to carry out the exercise below.

小提示： 你的系统发行版可能有一个用于软件开发的 meta-package（软件包的集合）。如果是这样的话， 考虑安装它，若你打算在你的系统中编译程序。若你的系统没有提供一个 meta-package，试着安装 gcc 和 make 工具包。 在许多发行版中，这就足够完成下面的练习了。 —

**得到源码**

For our compiling exercise, we are going to compile a program from the GNU Project called diction. This is a handy little program that checks text files for writing quality and style. As programs go, it is fairly small and easy to build.

为了我们的编译练习，我们将编译一个叫做 diction 的程序，来自 GNU 项目。这是一个小巧方便的程序， 检查文本文件的书写质量和样式。就程序而言，它相当小，且容易创建。

Following convention, we’re first going to create a directory for our source code named src and then download the source code into it using ftp:

遵照惯例，首先我们要创建一个名为 src 的目录来存放我们的源码，然后使用 ftp 协议把源码下载下来。

[me@linuxbox ~]$ mkdir src

[me@linuxbox ~]$ cd src

[me@linuxbox src]$ ftp ftp.gnu.org

Connected to ftp.gnu.org.

220 GNU FTP server ready.

Name (ftp.gnu.org:me): anonymous

230 Login successful.

Remote system type is UNIX.

Using binary mode to transfer files.

ftp> cd gnu/diction

250 Directory successfully changed.

ftp> ls

200 PORT command successful. Consider using PASV.

150 Here comes the directory listing.

-rw-r--r-- 1 1003 65534 68940 Aug 28 1998 diction-0.7.tar.gz

-rw-r--r-- 1 1003 65534 90957 Mar 04 2002 diction-1.02.tar.gz

-rw-r--r-- 1 1003 65534 141062 Sep 17 2007 diction-1.11.tar.gz

226 Directory send OK.

ftp> get diction-1.11.tar.gz

local: diction-1.11.tar.gz remote: diction-1.11.tar.gz

200 PORT command successful. Consider using PASV.

150 Opening BINARY mode data connection for diction-1.11.tar.gz

(141062 bytes).

226 File send OK.

141062 bytes received in 0.16 secs (847.4 kB/s)

ftp> bye

221 Goodbye.

[me@linuxbox src]$ ls

diction-1.11.tar.gz

Note: Since we are the “maintainer” of this source code while we compile it, we will keep it in ~/src. Source code installed by your distribution will be installed in /usr/src, while source code intended for use by multiple users is usually installed in /usr/local/src.

注意：因为我们是这个源码的“维护者”，当我们编译它的时候，我们把它保存在 ~/src 目录下。 由你的系统发行版源码会把源码安装在 /usr/src 目录下，而供多个用户使用的源码，通常安装在 /usr/local/src 目录下。

As we can see, source code is usually supplied in the form of a compressed tar file. Sometimes called a tarball, this file contains the source tree, or hierarchy of directories and files that comprise the source code. After arriving at the ftp site, we examine the list of tar files available and select the newest version for download. Using the get command within ftp, we copy the file from the ftp server to the local machine.

正如我们所看到的，通常提供的源码形式是一个压缩的 tar 文件。有时候称为 tarball，这个文件包含源码树， 或者是组成源码的目录和文件的层次结构。当到达 ftp 站点之后，我们检查可用的 tar 文件列表，然后选择最新版本，下载。 使用 ftp 中的 get 命令，我们把文件从 ftp 服务器复制到本地机器。

Once the tar file is downloaded, it must be unpacked. This is done with the tar program:

一旦 tar 文件下载下来之后，必须打开。通过 tar 程序可以完成：

[me@linuxbox src]$ tar xzf diction-1.11.tar.gz

[me@linuxbox src]$ ls

diction-1.11

diction-1.11.tar.gz

Tip: The diction program, like all GNU Project software, follows certain stan- dards for source code packaging. Most other source code available in the Linux ecosystem also follows this standard. One element of the standard is that when the source code tar file is unpacked, a directory will be created which contains the source tree, and that this directory will be named project-x.xx, thus containing both the project’s name and its version number. This scheme allows easy installation of multiple versions of the same program. However, it is often a good idea to examine the layout of the tree before unpacking it. Some projects will not create the directory, but instead will deliver the files directly into the current directory. This will make a mess in your otherwise well-organized src directory. To avoid this, use the following command to examine the contents of the tar file:

小提示：该 diction 程序，像所有的 GNU 项目软件，遵循着一定的源码打包标准。其它大多数在 Linux 生态系统中 可用的源码也遵循这个标准。该标准的一个条目是，当源码 tar 文件打开的时候，会创建一个目录，该目录包含了源码树， 并且这个目录将会命名为 project-x.xx，其包含了项目名称和它的版本号两项内容。这种方案能在系统中方便安装同一程序的多个版本。 然而，通常在打开 tarball 之前检验源码树的布局是个不错的主意。一些项目不会创建该目录，反而，会把文件直接传递给当前目录。 这会把你的（除非组织良好的）src 目录弄得一片狼藉。为了避免这个，使用下面的命令，检查 tar 文件的内容：

tar tzvf tarfile | head ---

**检查源码树**

Unpacking the tar file results in the creation of a new directory, named diction-1.11. This directory contains the source tree. Let’s look inside:

打开该 tar 文件，会创建一个新的目录，名为 diction-1.11。这个目录包含了源码树。让我们看一下里面的内容：

[me@linuxbox src]$ cd diction-1.11

[me@linuxbox diction-1.11]$ ls

config.guess diction.c getopt.c nl

config.h.in diction.pot getopt.h nl.po

config.sub diction.spec getopt\_int.h README

configure diction.spec.in INSTALL sentence.c

configure.in diction.texi.in install-sh sentence.h

COPYING en Makefile.in style.1.in

de en\_GB misc.c style.c

de.po en\_GB.po misc.h test

diction.1.in getopt1.c NEWS

In it, we see a number of files. Programs belonging to the GNU Project, as well as many others, will supply the documentation files README, INSTALL, NEWS, and COPYING.

在源码树中，我们看到大量的文件。属于 GNU 项目的程序，还有其它许多程序都会，提供文档文件 README，INSTALL，NEWS，和 COPYING。

These files contain the description of the program, information on how to build and in- stall it, and its licensing terms. It is always a good idea to carefully read the README and INSTALL files before attempting to build the program.

这些文件包含了程序描述，如何建立和安装它的信息，还有它许可条款。在试图建立程序之前，仔细阅读 README 和 INSTALL 文件，总是一个不错的主意。

The other interesting files in this directory are the ones ending with .c and .h:

在这个目录中，其它有趣的文件是那些以 .c 和 .h 为后缀的文件：

[me@linuxbox diction-1.11]$ ls \*.c

diction.c getopt1.c getopt.c misc.c sentence.c style.c

[me@linuxbox diction-1.11]$ ls \*.h

getopt.h getopt\_int.h misc.h sentence.h

The .c files contain the two C programs supplied by the package (style and diction), divided into modules. It is common practice for large programs to be broken into smaller, easier to manage pieces. The source code files are ordinary text and can be examined with less:

这些 .c 文件包含了由该软件包提供的两个 C 程序（style 和 diction），被分割成模块。这是一种常见做法，把大型程序 分解成更小，更容易管理的代码块。源码文件都是普通文本，可以用 less 命令查看：

[me@linuxbox diction-1.11]$ less diction.c

The .h files are known as header files. These, too, are ordinary text. Header files contain descriptions of the routines included in a source code file or library. In order for the com- piler to connect the modules, it must receive a description of all the modules needed to complete the entire program. Near the beginning of the diction.c file, we see this line:

这些 .h 文件以头文件而著称。它们也是普通文件。头文件包含了程序的描述，这些程序被包括在源码文件或库中。 为了让编译器链接到模块，编译器必须接受所需的所有模块的描述，来完成整个程序。在 diction.c 文件的开头附近， 我们看到这行代码：

#include "getopt.h"

This instructs the compiler to read the file getopt.h as it reads the source code in diction.c in order to “know” what’s in getopt.c. The getopt.c file supplies routines that are shared by both the style and diction programs.

这行代码指示编译器去读取文件 getopt.h，因为它会读取 diction.c 中的源码，为的是“知道” getopt.c 中的内容。 getopt.c 文件提供由 style 和 diction 两个程序共享的代码。

Above the include statement for getopt.h, we see some other include statements such as these:

在 getopt.h 的 include 语句上面，我们看到一些其它的 include 语句，比如这些：

#include <regex.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

These also refer to header files, but they refer to header files that live outside the current source tree. They are supplied by the system to support the compilation of every program. If we look in /usr/include, we can see them:

这些也涉及到头文件，但是这些头文件居住在当前源码树的外面。它们由操作系统供给，来支持每个程序的编译。 如果我们看一下 /usr/include 目录，能看到它们：

[me@linuxbox diction-1.11]$ ls /usr/include

The header files in this directory were installed when we installed the compiler.

当我们安装编译器的时候，这个目录中的头文件会被安装。

**构建程序**

Most programs build with a simple, two-command sequence:

大多数程序通过一个简单的，两个命令的序列构建：

./configure

make

The configure program is a shell script which is supplied with the source tree. Its job is to analyze the build environment. Most source code is designed to be portable. That is, it is designed to build on more than one kind of Unix-like system. But in order to do that, the source code may need to undergo slight adjustments during the build to accommodate differences between systems. configure also checks to see that necessary external tools and components are installed. Let’s run configure. Since configure is not located where the shell normally expects programs to be located, we must explicitly tell the shell its location by prefixing the command with ./ to indicate that the program is located in the current working directory:

这个 configure 程序是一个 shell 脚本，由源码树提供。它的工作是分析程序建立环境。大多数源码会设计为可移植的。 也就是说，它被设计成，能建立在多于一个的类 Unix 系统中。但是为了做到这一点，在建立程序期间，为了适应系统之间的差异， 源码可能需要经过轻微的调整。configure 也会检查是否安装了必要的外部工具和组件。让我们运行 configure 命令。 因为 configure 命令所在的位置不是位于 shell 通常期望程序所呆的地方，我们必须明确地告诉 shell 它的位置，通过 在命令之前加上 ./ 字符，来表明程序位于当前工作目录：

[me@linuxbox diction-1.11]$ ./configure

configure will output a lot of messages as it tests and configures the build. When it finishes, it will look something like this:

configure 将会输出许多信息，随着它测试和配置整个构建过程。当结束后，输出结果看起来像这样：

checking libintl.h presence... yes

checking for libintl.h... yes

checking for library containing gettext... none required

configure: creating ./config.status

config.status: creating Makefile

config.status: creating diction.1

config.status: creating diction.texi

config.status: creating diction.spec

config.status: creating style.1

config.status: creating test/rundiction

config.status: creating config.h

[me@linuxbox diction-1.11]$

What’s important here is that there are no error messages. If there were, the configuration failed, and the program will not build until the errors are corrected.

这里最重要的事情是没有错误信息。如果有错误信息，整个配置过程失败，然后程序不能构建直到修正了错误。

We see configure created several new files in our source directory. The most impor- tant one is Makefile. Makefile is a configuration file that instructs the make pro- gram exactly how to build the program. Without it, make will refuse to run. Makefile is an ordinary text file, so we can view it:

我们看到在我们的源码目录中 configure 命令创建了几个新文件。最重要一个是 Makefile。Makefile 是一个配置文件， 指示 make 程序究竟如何构建程序。没有它，make 程序就不能运行。Makefile 是一个普通文本文件，所以我们能查看它：

[me@linuxbox diction-1.11]$ less Makefile

The make program takes as input a makefile (which is normally named Makefile), that describes the relationships and dependencies among the components that comprise the finished program.

这个 make 程序把一个 makefile 文件作为输入（通常命名为 Makefile），makefile 文件 描述了包括最终完成的程序的各组件之间的关系和依赖性。

The first part of the makefile defines variables that are substituted in later sections of the makefile. For example we see the line:

makefile 文件的第一部分定义了变量，这些变量在该 makefile 后续章节中会被替换掉。例如我们看看这一行代码：

CC= gcc

which defines the C compiler to be gcc. Later in the makefile, we see one instance where it gets used:

其定义了所用的 C 编译器是 gcc。文件后面部分，我们看到一个使用该变量的实例：

diction: diction.o sentence.o misc.o getopt.o getopt1.o

$(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \

getopt.o getopt1.o $(LIBS)

A substitution is performed here, and the value $(CC) is replaced by gcc at run time. Most of the makefile consists of lines, which define a target, in this case the executable file diction, and the files on which it is dependent. The remaining lines describe the command(s) needed to create the target from its components. We see in this example that the executable file diction (one of the final end products) depends on the existence of diction.o, sentence.o, misc.o, getopt.o, and getopt1.o. Later on, in the makefile, we see definitions of each of these as targets:

这里完成了一个替换操作，在程序运行时，$(CC) 的值会被替换成 gcc。大多数 makefile 文件由行组成，每行定义一个目标文件， 在这种情况下，目标文件是指可执行文件 diction，还有目标文件所依赖的文件。剩下的行描述了从目标文件的依赖组件中 创建目标文件所需的命令。在这个例子中，我们看到可执行文件 diction（最终的成品之一）依赖于文件 diction.o，sentence.o，misc.o，getopt.o，和 getopt1.o都存在。在 makefile 文件后面部分，我们看到 diction 文件所依赖的每一个文件做为目标文件的定义：

diction.o: diction.c config.h getopt.h misc.h sentence.h

getopt.o: getopt.c getopt.h getopt\_int.h

getopt1.o: getopt1.c getopt.h getopt\_int.h

misc.o: misc.c config.h misc.h

sentence.o: sentence.c config.h misc.h sentence.h

style.o: style.c config.h getopt.h misc.h sentence.h

However, we don’t see any command specified for them. This is handled by a general target, earlier in the file, that describes the command used to compile any .c file into a .o file:

然而，我们不会看到针对它们的任何命令。这个由一个通用目标解决，在文件的前面，描述了这个命令，用来把任意的 .c 文件编译成 .o 文件：

.c.o:

$(CC) -c $(CPPFLAGS) $(CFLAGS) $<

This all seems very complicated. Why not simply list all the steps to compile the parts and be done with it? The answer to this will become clear in a moment. In the meantime, let’s run make and build our programs:

这些看起来非常复杂。为什么不简单地列出所有的步骤，编译完成每一部分？一会儿就知道答案了。同时， 让我们运行 make 命令并构建我们的程序：

[me@linuxbox diction-1.11]$ make

The make program will run, using the contents of Makefile to guide its actions. It will produce a lot of messages.

这个 make 程序将会运行，使用 Makefile 文件的内容来指导它的行为。它会产生很多信息。

When it finishes, we will see that all the targets are now present in our directory:

当 make 程序运行结束后，现在我们将看到所有的目标文件出现在我们的目录中。

[me@linuxbox diction-1.11]$ ls

config.guess de.po en en\_GB sentence.c

config.h diction en\_GB.mo en\_GB.po sentence.h

config.h.in diction.1 getopt1.c getopt1.o sentence.o

config.log diction.1.in getopt.c getopt.h style

config.status diction.c getopt\_int.h getopt.o style.1

config.sub diction.o INSTALL install-sh style.1.in

configure diction.pot Makefile Makefile.in style.c

configure.in diction.spec misc.c misc.h style.o

COPYING diction.spec.in misc.o NEWS test

de diction.texi nl nl.mo

de.mo diction.texi.i nl.po README

Among the files, we see diction and style, the programs that we set out to build. Congratulations are in order! We just compiled our first programs from source code! But just out of curiosity, let’s run make again:

在这些文件之中，我们看到 diction 和 style，我们开始要构建的程序。恭喜一切正常！我们刚才源码编译了 我们的第一个程序。但是出于好奇，让我们再运行一次 make 程序：

[me@linuxbox diction-1.11]$ make

make: Nothing to be done for `all'.

It only produces this strange message. What’s going on? Why didn’t it build the program again? Ah, this is the magic of make. Rather than simply building everything again, make only builds what needs building. With all of the targets present, make determined that there was nothing to do. We can demonstrate this by deleting one of the targets and running make again to see what it does. Let’s get rid of one of the intermediate targets:

它只是产生这样一条奇怪的信息。怎么了？为什么它没有重新构建程序呢？啊，这就是 make 奇妙之处了。make 只是构建 需要构建的部分，而不是简单地重新构建所有的内容。由于所有的目标文件都存在，make 确定没有任何事情需要做。 我们可以证明这一点，通过删除一个目标文件，然后再次运行 make 程序，看看它做些什么。让我们去掉一个中间目标文件：

[me@linuxbox diction-1.11]$ rm getopt.o

[me@linuxbox diction-1.11]$ make

We see that make rebuilds it and re-links the diction and style programs, since they depend on the missing module. This behavior also points out another important feature of make: it keeps targets up to date. make insists that targets be newer than their dependencies. This makes perfect sense, as a programmer will often update a bit of source code and then use make to build a new version of the finished product. make ensures that everything that needs building based on the updated code is built. If we use the touch program to “update” one of the source code files, we can see this happen:

我们看到 make 重新构建了 getopt.o 文件，并重新链接了 diction 和 style 程序，因为它们依赖于丢失的模块。 这种行为也指出了 make 程序的另一个重要特征：它保持目标文件是最新的。make 坚持目标文件要新于它们的依赖文件。 这个非常有意义，做为一名程序员，经常会更新一点儿源码，然后使用 make 来构建一个新版本的成品。make 确保 基于更新的代码构建了需要构建的内容。如果我们使用 touch 程序，来“更新”其中一个源码文件，我们看到发生了这样的事情：

[me@linuxboxdiction-1.11]$ ls -l diction getopt.c

-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction

-rw-r--r-- 1 me me 33125 2007-03-30 17:45 getopt.c

[me@linuxboxdiction-1.11]$ touch getopt.c

[me@linuxboxdiction-1.11]$ ls -l diction getopt.c

-rwxr-xr-x 1 me me 37164 2009-03-05 06:14 diction

-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c

[me@linuxbox diction-1.11]$ make

After make runs, we see that it has restored the target to being newer than the dependency:

运行 make 之后，我们看到目标文件已经更新于它的依赖文件：

[me@linuxbox diction-1.11]$ ls -l diction getopt.c

-rwxr-xr-x 1 me me 37164 2009-03-05 06:24 diction

-rw-r--r-- 1 me me 33125 2009-03-05 06:23 getopt.c

The ability of make to intelligently build only what needs building is a great benefit to programmers. While the time savings may not be very apparent with our small project, it is very significant with larger projects. Remember, the Linux kernel (a program that undergoes continuous modification and improvement) contains several million lines of code.

make 程序这种智能地只构建所需要构建的内容的特性，对程序来说，是巨大的福利。虽然在我们的小项目中，节省的时间可能 不是非常明显，在庞大的工程中，它具有非常重大的意义。记住，Linux 内核（一个经历着不断修改和改进的程序）包含了几百万行代码。

**安装程序**

Well-packaged source code will often include a special make target called install. This target will install the final product in a system directory for use. Usually, this directory is /usr/local/bin, the traditional location for locally built software. However, this directory is not normally writable by ordinary users, so we must become the superuser to perform the installation:

打包良好的源码经常包括一个特别的 make 目标文件，叫做 install。这个目标文件将在系统目录中安装最终的产品，以供使用。 通常，这个目录是 /usr/local/bin，为在本地所构建软件的传统安装位置。然而，通常普通用户不能写入该目录，所以我们必须变成超级用户， 来执行安装操作：

[me@linuxbox diction-1.11]$ sudo make install

After we perform the installation, we can check that the program is ready to go:

[me@linuxbox diction-1.11]$ which diction

/usr/local/bin/diction

[me@linuxbox diction-1.11]$ man diction

And there we have it!

**总结**

In this chapter, we have seen how three simple commands:

在这一章中，我们已经知道了三个简单命令：

./configure

make

make install

can be used to build many source code packages. We have also seen the important role that make plays in the maintenance of programs. The make program can be used for any task that needs to maintain a target/dependency relationship, not just for compiling source code.

可以用来构建许多源码包。我们也知道了在程序维护过程中，make 程序起到了举足轻重的作用。make 程序可以用到 任何需要维护一个目标/依赖关系的任务中，不仅仅为了编译源代码。

**拓展阅读**

* The Wikipedia has good articles on compilers and the make program:
* Wikipedia 上面有关于编译器和 make 程序的好文章：

<http://en.wikipedia.org/wiki/Compiler>

<http://en.wikipedia.org/wiki/Make_(software)>

* The GNU Make Manual:
* GNU Make 手册

<http://www.gnu.org/software/make/manual/html_node/index.html>