

深入理解软件包的配置、编译与安装

作者：[金步国](#)

版权声明

本文作者是一位自由软件爱好者，所以本文虽然不是软件，但是本着 GPL 的精神发布。任何人都可以自由使用、转载、复制和再分发，但必须保留作者署名，亦不得对声明中的任何条款作任何形式的修改，也不得附加任何其它条件。您可以自由链接、下载、传播此文档，但前提是必须保证全文完整转载，包括完整的版权信息和作译者声明。

其他作品

本文作者十分愿意与他人共享劳动成果，如果你对我的其他翻译作品或者技术文章有兴趣，可以在如下位置查看现有作品的列表：

- 电信用户请访问：[金步国作品列表](#)
- 网通用户请访问：[金步国作品列表](#)
- 铁通用户请访问：[金步国作品列表](#)
- 教育网用户请访问：[金步国作品列表](#)

BUG 报告，切磋与探讨

由于作者水平有限，因此不能保证作品内容准确无误，请在阅读中自行鉴别。如果你发现了作品中的错误，请您来信指出，哪怕是错别字也好，任何提高作品质量的建议我都将虚心接纳。如果你愿意就作品中的相关内容与我进行进一步切磋与探讨，也欢迎你与我联系。联系方式：Email: csfrank@citiz.net；QQ: 70171448；MSN: csfrank122@hotmail.com

前言

从源代码安装过软件的朋友一定对 `./configure && make && make install` 安装三部曲非常熟悉了。然而究竟这个过程后的每一步幕后都发生了些什么呢？本文将带领你一探究竟。深入理解这个过程将有助于你在 LFS 的基础上玩出自己的花样来。不过需要说明的是本文对 Makefile 和 make 的讲解是相当近视和粗浅的，但是对于理解安装过程来说足够了。

概述

用一句话来解释这个过程就是：

根据源码包中 Makefile.in 文件的指示，configure 脚本检查当前的系统环境和配置选项，在当前目录中生成 Makefile 文件(还有其它本文无需关心的文件)，然后 make 程序就按照当前目录中的 Makefile 文件的指示将源代码编译为二进制文件，最后将这些二进制文件移动(即安装)到指定的地方(仍然按照 Makefile 文件的指示)。

由此可见 Makefile 文件是幕后的核心。要深入理解安装过程，必须首先对 Makefile 文件有充分的了解。本文将首先讲述 Makefile 与 make，然后再讲述 configure 脚本。并且在讲述这两部分内容时，提供了尽可能详细的、可以运用于实践的参考资料。

Makefile 与 make

用一句话来概括 Makefile 与 make 的关系就是：

Makefile 包含了所有的规则和目标，而 make 则是为了完成目标而去解释 Makefile 规则的工具。

make 语法

首先看看 make 的命令行语法：

```
make [options] [targets] [VAR=VALUE]...
```

[options]是命令行选项，可以用 `make --help` 命令查看全部，[VAR=VALUE]是在命令行上指定环境变量，这两个大家都很熟悉，将在稍后详细讲解。而[targets]是什么呢？字面的意思是“目标”，也就是希望本次 make 命令所完成的任务。凭经验猜测，这个[targets]大概可以用“check”，“install”之类(也就是常见的测试和安装命令)。但是它到底是个啥玩意儿？不带任何“目标”的 make 命令是什么意思？为什么在安装 LFS 工具链中的 Perl-5.8.8 软件包时会出现“make perlutilities”这样怪异的命令？要回答这些问题必须首先理解 Makefile 文件中的“规则”。

Makefile 规则

Makefile 规则包含了文件之间的依赖关系和更新此规则目标所需要的命令。

一个简单的 Makefile 规则是这样写的：

TARGET : PREREQUISITES
COMMAND

TARGET

规则的目标。也就是可以被 make 使用的“目标”。有些目标可以没有依赖而只有动作(命令行)，比如“clean”，通常仅仅定义一系列删除中间文件的命令。同样，有些目标可以没有动作而只有依赖，比如“all”，通常仅仅用作“终极目标”。

PREREQUISITES

规则的依赖。通常一个目标依赖于一个或者多个文件。

COMMAND

规则的命令行。一个规则可以有零个或多个命令行。

OK! 现在你明白[targets]是什么了，原来它们来自于 Makefile 文件中一条条规则的目标(TARGET)。另外，Makefile 文件中第一条规则的目标被称为“终极目标”，也就是你省略[targets]参数时的目标(通常为“all”)。

当你查看一个实际的 Makefile 文件时，你会发现有些规则非常复杂，但是它都符合规则的基本格式。此外，Makefile 文件中通常还包含了除规则以外的其它很多东西，不过本文只关心其中的变量。

Makefile 变量

Makefile 中的“变量”更像是 C 语言中的宏，代表一个文本字符串(变量的值)，可以用于规则的任何部分。变量的定义很简单：VAR = VALUE；变量的引用也很简单：\$(VAR) 或者 \${VAR}。变量引用的展开过程是严格的文本替换过程，就是说变量值的字符串被精确的展开在变量被引用的地方。比如，若定义：VAR=c，那么，“\$(VAR)\$(VAR)-\$(VAR)VAR.\$(VAR)”将被展开为“c c-c VAR.c”。

虽然在 Makefile 中可以直接使用系统的环境变量，但是也可以通过在 Makefile 中定义同名变量来“遮盖”系统的环境变量。另一方面，我们可以在调用 make 时使用 -e 参数强制使系统中的环境变量覆盖 Makefile 中的同名变量，除此之外，在调用 make 的命令行上使用 VAR=VALUE 格式指定的环境变量也可以覆盖 Makefile 中的同名变量。

Makefile 实例

下面看一个简单的、实际的 Makefile 文件：

```
CC=gcc
CPPFLAGS=
CFLAGS=-O2 -pipe
LDFLAGS=-s
PREFIX=/usr

all : prog1 prog2

prog1 : prog1.o
$(CC) $(LDFLAGS) -o prog1 prog1.o

prog1.o : prog1.c
$(CC) -c $(CFLAGS) prog1.c

prog2 : prog2.o
$(CC) $(CFLAGS) $(LDFLAGS) -o prog2 prog2.o

prog2.o : prog2.c
$(CC) -c $(CPPFLAGS) $(CFLAGS) prog2.c

clean :
rm -f *.{o,a} prog{1,2}

install : prog1 prog2
if ( test ! -d $(PREFIX)/bin ) ; then mkdir -p $(PREFIX)/bin ; fi
cp -f prog1 $(PREFIX)/bin/prog1
cp -f prog2 $(PREFIX)/bin/prog2

check test : prog1 prog2
prog1 < sample1.ref > sample1.rz
prog1 < sample2.ref > sample3.rz
cmp sample1.ok sample1.rz
cmp sample2.ok sample2.rz
```

从中可以看出，make 与 make all 以及 make prog1 prog2 三条命令其实是等价的。而常用的 make check 和 make install 也找到了归属。同时我们也看到了 Makefile 中的各种变量是如何影响编译的。针对这个特定的 Makefile ，你甚至可以省略安装三部曲中的 make 命令而直接使用 make install 进行安装。

同样，为了使用自定义的编译参数编译 prog2 ，我们可以使用 make prog2 CFLAG S = “-O 3 -m arch=athlon64” 或 CFLAG S = “-O 3 -m arch=athlon64”&& make -e prog2 命令达到此目的。

Makefile 惯例

下面是 Makefile 中一些约定俗成的目标名称及其含义：

all

编译整个软件包，但不重建任何文档。一般此目标作为默认的终极目标。此目标一般对所有源程序的编译和连接使用“-g”选项，以使

最终的可执行程序中包含调试信息。可使用 strip 程序去掉这些调试符号。

- clean
清除当前目录下在 make 过程中产生的文件。它不能删除软件包的配置文件，也不能删除 build 时创建的那些文件。
- distclean
类似于“clean”，但增加删除当前目录下的的配置文件、build 过程产生的文件。
- info
产生必要的 Info 文档。
- check 或 test
完成所有的自检功能。在执行检查之前，应确保所有程序已经被创建(但可以尚未安装)。为了进行测试，需要实现在程序没有安装的情况下被执行的测试命令。

- install
完成程序的编译并将最终的可执行程序、库文件等拷贝到指定的目录。此种安装一般不对可执行程序进行 strip 操作。
- install-strip
和“install”类似，但是会对复制到安装目录下的可执行文件进行 strip 操作。
- uninstall
删除所有由“install”安装的文件。
- installcheck
执行安装检查。在执行安装检查之前，需要确保所有程序已经被创建并且被安装。
- installdirs
创建安装目录及其子目录。它不能更改软件的编译目录，而仅仅是创建程序的安装目录。

下面是 Makefile 中一些约定俗成的变量名称及其含义：

这些约定俗成的变量分为三类。第一类代表可执行程序的名字，例如 CC 代表编译器这个可执行程序；第二类代表程序使用的参数(多个参数使用空格分开)，例如 CFLAGS 代表编译器执行时使用的参数(一种怪异的做法是直接在 CC 中包含参数)；第三类代表安装目录，例如 prefix 等等，含义简单，下面只列出它们的默认值。

- AR 函数库打包程序，可创建静态库.a 文档。默认是"ar"。
- AS 汇编程序。默认是"as"。
- CC C 编译程序。默认是"cc"。
- CXX C++编译程序。默认是"g++"。
- CPP C/C++预处理器。默认是"\${CC} -E"。
- FC Fortran 编译器。默认是"f77"。
- PC Pascal 语言编译器。默认是"pc"。
- YACC Yacc 文法分析器。默认是"yacc"。

- ARFLAGS 函数库打包程序的命令行参数。默认值是"rv"。
- ASFLAGS 汇编程序的命令行参数。
- CFLAGS C 编译程序的命令行参数。
- CXXFLAGS C++编译程序的命令行参数。
- CPPFLAGS C/C++预处理器的命令行参数。
- FFLAGS Fortran 编译器的命令行参数。
- PFLAGS Pascal 编译器的命令行参数。
- YFLAGS Yacc 文法分析器的命令行参数。
- LDFLAGS 链接器的命令行参数。

- prefix /usr/local
- exec_prefix \$(prefix)
- bindir \$(exec_prefix)/bin
- sbindir \$(exec_prefix)/sbin
- libexecdir \$(exec_prefix)/libexec
- datadir \$(prefix)/share
- sysconfdir \$(prefix)/etc
- sharedstatedir \$(prefix)/com
- localstatedir \$(prefix)/var
- libdir \$(exec_prefix)/lib
- infodir \$(prefix)/info
- includedir \$(prefix)/include
- oldincludedir \$(prefix)/include
- mandir \$(prefix)/man
- srcdir 需要编译的源文件所在的目录，无默认值

make 选项

最后说说 make 的命令行选项(以 Make-3.81 版本为准)：

- B, --always-make
无条件的重建所有规则的目标，而不是根据规则的依赖关系决定是否重建某些目标文件。
- C DIR, --directory= DIR
在做任何动作之前先切换工作目录到 DIR ，然后再执行 make 程序。
- d
在 make 执行过程中打印出所有的调试信息。包括：make 认为那些文件需要重建；那些文件需要比较它们的最后修改时间、比较的结果；重建目标所要执行的命令；使用的隐含规则等。使用该选项我们可以看到 make 构造依赖关系链、重建目标过程的所有信息，它等效于“-debug=a”。
- debug=FLAGS
在 make 执行过程中打印出调试信息。FLAGS 用于控制调试信息级别：
- a
输出所有类型的调试信息

b

输出基本调试信息。包括：那些目标过期、是否重建成功过期目标文件。

v

除 b 级别以外还包括：解析的 makefile 文件名，不需要重建文件等。

i

除 b 级别以外还包括：所有使用到的隐含规则描述。

j

输出所有执行命令的子进程，包括命令执行的 PID 等。

m

输出 make 读取、更新、执行 makefile 的信息。

-e, --environment-overrides

使用系统环境变量的定义覆盖 Makefile 中的同名变量定义。

-f FILE, --file=FILE, --makefile=FILE

将 FILE 指定为 Makefile 文件。

-h, --help

打印帮助信息。

-i, --ignore-errors

忽略规则命令执行过程中的错误。

-IDIR, --include-dir=DIR

指定包含 Makefile 文件的搜索目录。使用多个“-I”指定目录时，搜索目录按照指定顺序进行。

-j[N], --jobs[=N]

指定并行执行的命令数目。在没有指定“-j”参数的情况下，执行的命令数目将是系统允许的最大可能数目。

-k, --keep-going

遇见命令执行错误时不终止 make 的执行，也就是尽可能执行所有的命令，直到出现致命错误才终止。

-l[N], --load-average[=N], --max-load[=N]

如果系统负荷超过 LOAD (浮点数)，不再启动新任务。

-L, --check-symlink-times

同时考察符号连接的时间戳和它所指向的目标文件的时间戳，以两者中较晚的时间戳为准。

-n, --just-print, --dry-run, --recon

只打印出所要执行的命令，但并不实际执行命令。

-o FILE, --old-file=FILE, --assume-old=FILE

即使相对于它的依赖已经过期也不重建 FILE 文件；同时也不重建依赖于此文件任何文件。

-p, --print-data-base

命令执行之前，打印出 make 读取的 Makefile 的所有数据（包括规则和变量的值），同时打印出 make 的版本信息。如果只需要打印这些数据信息，可以使用 make -qp 命令。查看 make 执行前的预设规则和变量，可使用命令 make -p -f/dev/null。

-q, --question

“询问模式”。不运行任何命令，并且无输出，只是返回一个查询状态。返回状态为 0 表示没有目标需要重建，1 表示存在需要重建的目标，2 表示有错误发生。

-r, --no-builtin-rules

取消所有内嵌的隐含规则，不过你可以在 Makefile 中使用模式规则来定义规则。同时还会取消所有支持后追规则的隐含后缀列表，同样我们也可以在 Makefile 中使用“.SUFFIXES”定义我们自己的后缀规则。此选项不会取消 make 内嵌的隐含变量。

-R, --no-builtin-variables

取消 make 内嵌的隐含变量，不过我们可以在 Makefile 中明确定义某些变量。注意，此选项同时打开了“-r”选项。因为隐含规则是以内嵌的隐含变量为基础的。

-s, --silent, --quiet

不显示所执行的命令。

-S, --no-keep-going, --stop

取消“-k”选项。在递归的 make 过程中子 make 通过 MAKEFLAGS 变量继承了上层的命令行选项。我们可以在子 make 中使用“-S”选项取消上层传递的“-k”选项，或者取消系统环境变量 MAKEFLAGS 中的“-k”选项。

-t, --touch

更新所有目标文件的时间戳到当前系统时间。防止 make 对所有过时目标文件的重建。

-v, --version

打印版本信息。

-w, --print-directory

在 make 进入一个目录之前打印工作目录。使用“-C”选项时默认打开这个选项。

--no-print-directory

取消“-w”选项。可以是用在递归的 make 调用过程中，取消“-C”参数将默认打开“-w”。

-W FILE, --what-if=FILE, --new-file=FILE, --assume-new=FILE

设定 FILE 文件的时间戳为当前时间，但不改变文件实际的最后修改时间。此选项主要是为实现了对所有依赖于 FILE 文件的目标的强制重建。

--warn-undefined-variables

在发现 Makefile 中存在对未定义的变量进行引用时给出告警信息。此功能可以帮助我们调试一个存在多级套嵌变量引用的复杂 Makefile。但是：我们建议在书写 Makefile 时尽量避免超过三级以上的变量套嵌引用。

configure

此阶段的主要目的是生成 Makefile 文件，是最关键的运筹帷幄阶段，基本上所有可以对安装过程进行的个性化调整都集中在这一步。

configure 脚本能够对 Makefile 中的哪些内容产生影响呢？基本上可以这么说：所有内容，包括本文最关心的 Makefile 规则与 Makefile 变量。那么又是哪些因素影响最终生成的 Makefile 文件呢？答曰：系统环境和配置选项。

配置选项的影响是显而易见的。但是“系统环境”的概念却很宽泛，包含很多方面内容，不过我们这里只关心环境变量，具体说来就是将来会在 Makefile 中使用到的环境变量以及与 Makefile 中的变量同名的环境变量。

通用 configure 语法

在进一步讲述之前，先看看 configure 脚本的语法，一般有两种：

```
configure [OPTIONS] [VAR=VALUE]...
```

```
configure [OPTIONS] [HOST]
```

不管是哪种语法，我们都可以用 `configure --help` 查看所有可用的 [OPTIONS]，并且通常在结尾部分还能看到这个脚本所关心的环境变量有哪些。在本文中将对这两种语法进行合并，使用下面这种简化的语法：

```
configure [OPTIONS]
```

这种语法能够被所有的 configure 脚本所识别，同时也能通过设置环境变量和使用特定的 [OPTIONS] 完成上述两种语法的一切功能。

通用 configure 选项

虽然每个软件包的 configure 脚本千差万别，但是它们却都有一些共同的选项，也基本上都遵守相同的选项语法。

脚本自身选项

```
--help
显示帮助信息。
--version
显示版本信息。
--cache-file=FILE
在 FILE 文件中缓存测试结果(默认禁用)。
--no-create
configure 脚本运行结束后不输出结果文件，常用于正式编译前的测试。
--quiet, --silent
不显示脚本工作期间输出的“checking ...”消息。
```

目录选项

```
--srcdir=DIR
源代码文件所在目录，默认为 configure 脚本所在目录或其父目录。
--prefix=PREFIX
体系无关文件的顶级安装目录 PREFIX，默认值一般是 /usr/local 或 /usr/local/pkgName
--exec-prefix=EPREFIX
体系相关文件的顶级安装目录 EPREFIX，默认值一般是 PREFIX
--bindir=DIR
用户可执行文件的存放目录 DIR，默认值一般是 EPREFIX/bin
--sbindir=DIR
系统管理员可执行目录 DIR，默认值一般是 EPREFIX/sbin
--libexecdir=DIR
程序可执行目录 DIR，默认值一般是 EPREFIX/libexec
--datadir=DIR
通用数据文件的安装目录 DIR，默认值一般是 PREFIX/share
--sysconfdir=DIR
只读的单一机器数据目录 DIR，默认值一般是 PREFIX/etc
--sharedstatedir=DIR
可写的体系无关数据目录 DIR，默认值一般是 PREFIX/com
--localstatedir=DIR
可写的单一机器数据目录 DIR，默认值一般是 PREFIX/var
--libdir=DIR
库文件的安装目录 DIR，默认值一般是 EPREFIX/lib
--includedir=DIR
C 头文件目录 DIR，默认值一般是 PREFIX/include
--oldincludedir=DIR
非 gcc 的 C 头文件目录 DIR，默认值一般是 /usr/include
--infodir=DIR
Info 文档的安装目录 DIR，默认值一般是 PREFIX/info
--mandir=DIR
Man 文档的安装目录 DIR，默认值一般是 PREFIX/man
```

体系结构选项

玩交叉编译的朋友对这些选项已经很熟悉了，并且对于通常的交叉编译情况而言，`HOST == BUILD != TARGET`。但是对于不使用

交叉编译的朋友也不必担心，将它们三个都设为相同即可。

```
--host=HOST
运行工具链的机器，默认是 config.guess 脚本的输出结果。
--build=BUILD
用来建立工具链的机器，默认值是 HOST
--target=TARGET
工具链所生成的二进制代码最终运行的机器，默认值是 HOST
```

特性选项

```
--enable-FEATURE
启用 FEATURE 特性
--disable-FEATURE
禁用 FEATURE 特性
--with-PACKAGE[=DIR]
启用附加软件包 PACKAGE，亦可同时指定 PACKAGE 所在目录 DIR
--without-PACKAGE
禁用附加软件包 PACKAGE
```

通用环境变量

除了上述通用的选项外，下列环境变量影响着最终生成的 Makefile 文件：

```
CPP
C 预处理器命令
CXXCPP
C++ 预处理器命令
CPPFLAGS
C/C++ 预处理器命令行参数
CC
C 编译器命令
CFLAGS
C 编译器命令行参数
CXX
C++ 编译器命令
CXXFLAGS
C++ 编译器命令行参数
LDFLAGS
连接器命令行参数
```

至于设置这些环境变量的方法，你可以将它们 export 为全局变量在全局范围内使用，也可以在命令行上使用 [VAR=VALUE]... configure [OPTIONS] 的语法局部使用。此处就不详细描述了。

看完上述内容以后，不用多说你应当自然而然的明白该进行如何对自己的软件包进行定制安装了。祝你好运！

补充读物

根据 d00m3d 的推荐，LinuxSir.Org 上的另外两篇帖子：[《编译的一点体会》](#)和[《关于库的深入思考》](#)，可以作为本文的进一步读物，更加有助于深入理解本文的主题。另外建立在本文基础上的[《编译优化指南》](#)专门针对与优化相关的问题进行了探讨。推荐阅读。