第 4 章 debian 目录中的必需内容

目录

本教程文档已被重写为另外的 Debian 维护者指导 文档，其中包含了更新的内容与更多实际例子。请使用新的教程作为主要的教程文档。

在程序源代码目录下有一个叫做 **debian** 的新的子目录。这个目录中存放着许多文件，我们将要修改这些文件来定制软件包行为。其中最重要的文件当属 **control**，**changelog**，**copyright**，以及 **rules**，所有的软件包都必须有这几个文件。[27]

4.1. control

这个文件包含了很多供 **dpkg**、**dselect**、**apt-get**、**apt-cache**、**aptitude** 等包管理工具进行管理时所使用的许多变量。这些变量均在 Debian Policy Manual，5 "Control files and their fields" 中被定义。

这里的 **control** 文件是 **dh_make** 命令为我们创建的：

```
 1 Source: gentoo
 2 Section: unknown
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10)
 6 Standards-Version: 4.0.0
 7 Homepage: <insert the upstream URL, if relevant>
 8
 9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Description: <insert up to 60 chars description>
13  <insert long description, indented with spaces>
```

(注：我为它添加了行号。)

第 1-7 行是源代码包的控制信息。第 9-13 行是二进制包的控制信息。

第 1 行是源代码包的名称。

第 2 行是该源码包要进入发行版中的分类。

你可能已经注意到，Debian 仓库被分为几个类别：**main** (自由软件)、**non-free** (非自由软件)以及 **contrib** (依赖于非自由软件的自由软件)。在这些大的分类之下还有多个逻辑上的子分类，用以简短描述软件包的用途类别。**admin** 为供系统管理员使用的程序，**devel** 为开发工具，**doc** 为文档，**libs** 为库，**mail** 为电子邮件阅读器或邮件系统守护程序，**net** 为网络应用程序或网络服务守护进程，**x11** 为不属于其他分类的为 X11 程序，此外还有很多很多。[28]

我们将本例设置为 x11。(**main/** 前缀是默认值，可以省略。)

第 3 行描述了用户安装此软件包的优先级。[29]

- **optional** 优先级适用于与优先级为 **required**、**important** 或 **standard** 的软件包不冲突的新软件包。

Section 和 Priority 常被如 **aptitude** 的前端所使用，以分类软件包并选择默认值。一旦你把软件包上传到 Debian，这两项的值可以被仓库维护人员修改，此时你将收到提示邮件。

由于这是一个常规优先级的软件，并不与其他软件包冲突，我们将优先级改为 **optional**。

第 4 行是维护者的姓名和电子邮件地址。请确保此处的值可以直接用于电子邮件头的 **To** 项。因为一旦你将软件包上传至仓库，Bug 跟踪系统将使用它向你发送可能的 Bug 报告邮件。请避免使用逗号、“&”符号或括号。

第 5 行中的 **Build-Depends** 项列出了编译此软件包需要的软件包。你还可以在这里添加一行 **Build-Depends-Indep** 作为附加。[30] 有些被 **build-essential** 依赖的软件包，如 **gcc** 和 **make** 等，已经会被默认安装而不需再写到此处。如果你需要其他工具来编译这个软件包，请将它们加到这里。多个软件包应使用半角逗号分隔。继续阅读二进制包依赖关系以增进对这些行的语法的理解。

- 对于所有在 **debian/rules** 文件中使用 **dh** 命令打包的软件包，必须在 **Build-Depends** 中包含 **debhelper (>=9)** 以满足 Debian Policy 中对 **clean** target 的要求。
- 对于生成有标记过 **Architecture: any** 的二进制包的源码包，它们将被 autobuilder 重构建。因为 autobuilder 过程在仅安装 **Build-Depends** 中列出的程序前便执行 **debian/rules build** 中的内容(参看 第 6.2 节 “自动编译系统”)，**Build-Depends** 字段需要列出所有必须的编译依赖，而 **Build-Depends-Indep** 则很少使用。
- 对于生成全标记 **Architecture: all** 二进制包的源码包，**Build-Depends-Indep** 中应列出所有要求的软件包，除非 **Build-Depends** 中已经列出，这样以便满足Debian Policy 中对 **clean** target 的要求。

如果你不知道应该使用哪一个，则使用 **Build-Depends** 以保证安全。[31]

要找出编译你的软件所需的软件包可以使用这个命令(译注：来自devscripts包)：

```
$ dpkg-depcheck -d ./configure
```

要手工地找到 ***/usr/bin/foo*** 的编译依赖，可以执行

```
$ objdump -p /usr/bin/foo | grep NEEDED
```

对于列出的每个库(例如 **libfoo.so.6**)，运行

```
$ dpkg -S libfoo.so.6
```

接下来直接将相应的 **-dev** 版本的软件包名称放到 **Build-Depends** 项内。如果你使用 **ldd**，它也会报告出间接的库依赖关系，这可能造成填写依赖时画蛇添足。

gentoo 需要 **xlibs-dev**、**libgtk1.2-dev** 和 **libglib1.2-dev** 才能编译，所以我们将这些软件包加在 **debhelper** 之后。

第 6 行是此软件包所依据的 Debian Policy Manual 标准版本号。

在第 7 行你可以放置上游项目首页的URL。

第 9 行是二进制软件包的名称。通常情况下与源代码包相同，但不是必须的。

第 10 行描述了可以编译本二进制包的体系结构。根据二进制包的类型，这个值常常是下列中的一个：[32]

- **Architecture: any**
  - 一般而言，包含 编译型语言编写的程序 生成的二进制包依赖于具体的体系结构。
- **Architecture: all**
  - 一般而言，包含 文本、图像、或解释型语言脚本 生成的二进制包独立于体系结构。

我们不管第 10 行，鉴于本程序是用 C 语言编写的。dpkg-gencontrol(1) 命令将根据这个软件包可以编译的平台 而为此处填写合适的信息。

如果你的软件包是平台独立的(例如一个 shell 或 Perl 脚本，或一些文档)，将这项改变为 **all**，然后继续阅读 第 4.4 节 “**rules**” 中关于使用 **binary-indep** 指令替代 **binary-arch** 来编译软件包的内容。

第 11 行显示了 Debian 软件包系统中最强大的特性之一。每个软件包都可以和其他软件包有各种不同的关系。除 **Depends** 外，还有 **Recommends**、**Suggests**、**Pre-Depends**、**Breaks**、**Conflicts**、**Provides** 和 **Replaces**。

软件包管理工具通常对这些关系采取相同的操作；如果有例外，本教程将会详细解释。(参看 dpkg(8)、dselect(8)、apt(8)、aptitude(1) 等。)

这里有一篇关于软件包关系的简述：[33]

- **Depends**

  此软件包仅当它依赖的软件包均已安装后才可以安装。此处请写明你的程序所必须的软件包，如果没有要求的软件包该软件便不能正常运行(或严重抛锚)的话。

- **Recommends**

  这项中的软件包不是严格意义上必须安装才可以保证程序运行。当用户安装软件包时，所有前端工具都会询问是否要安装这些推荐的软件包。**aptitude** 和 **apt-get** 会在安装你的软件包的时候自动安装推荐的软件包(用户可以禁用这个默认行为)。**dpkg** 则会忽略此项。

- **Suggests**

  此项中的软件包可以和本程序更好地协同工作，但不是必须的。当用户安装程序时，所有的前端程序可能不会询问是否安装建议的软件包。**aptitude** 可以被配置为安装软件时自动安装建议的软件包，但这不是默认。**dpkg** 和 **apt-get** 将忽略此项。

- **Pre-Depends**

  此项中的依赖强于 **Depends** 项。软件包仅在预依赖的软件包已经安装并且 *正确配置* 后才可以正常安装。在使用此项时应 *非常慎重*，仅当在 debian-devel@lists.debian.org 邮件列表讨论后才能使用。记住：根本就不要用这项。:-)

- **Conflicts**

  仅当所有冲突的软件包都已经删除后此软件包才可以安装。当程序在某些特定软件包存在的情况下根本无法运行或存在严重问题时使用此项。

- **Breaks**

  此软件包安装后列出的软件包将会受到损坏。通常 **Breaks** 要附带一个“版本号小于多少”的说明。这样，软件包管理工具将会选择升级被损坏的特定版本的软件包作为解决方案。

- **Provides**

  某些类型的软件包会定义有多个备用的虚拟名称。你可以在 virtual-package-names-list.txt.gz文件中找到完整的列表。如果你的程序提供了某个已存在的虚拟软件包的功能则使用此项。

- **Replaces**

  当你的程序要替换其他软件包的某些文件，或是完全地替换另一个软件包(与 **Conflicts** 一起使用)。列出的软件包中的某些文件会被你软件包中的文件所覆盖。

所有的这些项都使用统一的语法。它们是一个软件包列表，软件包名称间使用半角逗号分隔。也可以写出有多个备选软件包名称，这些软件包使用 **|** 符号(管道符)分隔。

这些项内还可以限定与某些软件包的某个版本区间之间的关系。版本号限定在括号内，这紧随软件包名称之后，并在以下逻辑符号后写清具体版本：**<<**、**<=**、**=**、**>=** 和 **>>**，分别代表严格小于、小于或等于、严格等于、大于或等于以及严格大于。例如，

```
Depends: foo (>= 1.2), libbar1 (= 1.3.4)
Conflicts: baz
Recommends: libbaz4 (>> 4.0.7)
Suggests: quux
Replaces: quux (<< 5), quux-foo (<= 7.6)
```

最后一个需要了解的特性是 **${shlibs:Depends}**，**${perl:Depends}**，**${misc:Depends}**，之类。

dh_shlibdeps(1) 会为二进制包计算共享库依赖关系。它会为每个二进制包生成一份 ELF 可执行文件和共享库列表。这个列表用于替换 **${shlibs:Depends}**。

dh_perl(1) 会计算 Perl 依赖。它会为每个二进制包生成一个叫作 **perl** 或 **perlapi** 的依赖列表。这个列表用于替换 **${perl:Depends}**。

一些 **debhelper** 命令可能会使生成的软件包需要依赖于某些其他的软件包。所有这些命令将会为每一个二进制包生成一个列表。这些列表将用于替换 **${misc:Depends}** 。

dh_gencontrol(1) 会为每个二进制包生成 **DEBIAN/control** 当替换 **${shlibs:Depends}**，**${perl:Depends}**，**${misc:Depends}**，之类的时候。

说过这些以后，我们可以让 **Depends** 项保持现状，并在其下插入一行 **Suggests: file**，因为 **gentoo** 可以使用 **file** 软件包提供的某些特性。

第 9 行是主页的URL。我们假设它是 http://www.obsession.se/gentoo/。

第 12 行是简述。绝大多数人的屏幕是 80 列宽，所以描述不应超过 60 个字符。在这个例子里我把它写为 **fully GUI-configurable, two-pane X file manager**。

第 13 行是长描述开始的地方。这应当是一段更详细地描述软件包的话。每行的第一个格应当留空。描述中不应存在空行，如果必须使用空行，则在行中仅放置一个 **.** (半角句点)来近似。同时，长描述后也不应有超过一行的空白。[34]

接下来我们在第 6 和第 7 行之间添加版本控制系统位置 **Vcs-*** 项。[35] 这里我们假设 **gentoo** 软件包的VCS 处于 Debian Alioth Git 服务的 **git://git.debian.org/git/collab-maint/gentoo.git**

到此为止，我们做好了 **control** 文件：

```
 1 Source: gentoo
 2 Section: x11
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
 6 Standards-Version: 3.9.4
 7 Vcs-Git: https://anonscm.debian.org/git/collab-maint/gentoo.git
 8 Vcs-browser: https://anonscm.debian.org/git/collab-maint/gentoo.git
 9 Homepage: http://www.obsession.se/gentoo/
10
11 Package: gentoo
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Suggests: file
15 Description: fully GUI-configurable, two-pane X file manager
16  gentoo is a two-pane file manager for the X Window System. gentoo lets the
17  user do (almost) all of the configuration and customizing from within the
18  program itself. If you still prefer to hand-edit configuration files,
19  they're fairly easy to work with since they are written in an XML format.
20  .
21  gentoo features a fairly complex and powerful file identification system,
22  coupled to an object-oriented style system, which together give you a lot
23  of control over how files of different types are displayed and acted upon.
24  Additionally, over a hundred pixmap images are available for use in file
25  type descriptions.
26  .
29  gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
30  for its interface.
```

(注：我为它添加了行号。)

4.2. copyright

这个文件包含了上游软件的版权以及许可证信息。Debian Policy Manual, 12.5 "Copyright information" 掌控着它的内容，另外 DEP-5: Machine-parseable **debian/copyright** 提供了关于其格式的方针。

dh_make 可以给出一个 **copyright** 文件的模板。在这里我们使用 **--copyright gpl2** 参数来获得一个模板写明 **gentoo** 软件包是发布于 GPL-2 许可证下。

你必须填写上空缺的信息，如你从何处获得此软件，实际的版权声明和它们的许可证。对于常见的自由软件许可证，如 GNU GPL-1、GNU GPL-2、GNU GPL-3、LGPL-2、LGPL-2.1、LGPL-3、GNU FDL-1.2、GNU FDL-1.3、Apache-2.0、3-Clause BSD、CC0-1.0、MPL-1.1、MPL-2.0 或 Artistic 许可证，你可以直接将其指向所有 Debian 系统都有的 **/usr/share/common-licenses/** 目录下的文件。否则，许可证则必须包含完整的许可证文本。

简言之，**gentoo** 的 **copyright** 文件如下所示：

```
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: gentoo
Upstream-Contact: Emil Brink <emil@obsession.se>
Source: http://sourceforge.net/projects/gentoo/files/

Files: *
Copyright: 1998-2010 Emil Brink <emil@obsession.se>
License: GPL-2+

Files: icons/*
Copyright: 1998 Johan Hanson <johan@tiq.com>
License: GPL-2+

Files: debian/*
Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
License: GPL-2+

License: GPL-2+
 This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 2 of the License, or
 (at your option) any later version.
 .
 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.
 .
 You should have received a copy of the GNU General Public License along
 with this program; if not, write to the Free Software Foundation, Inc.,
 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 .
 On Debian systems, the full text of the GNU General Public
 License version 2 can be found in the file
 '/usr/share/common-licenses/GPL-2'.
```

(注：我为它添加了行号。)

另外还可以参看 ftpmasters 发送到 debian-devel-announce 的 HOWTO：announce: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

4.3. changelog

这是一个必须的文件，它的特殊格式在 Debian Policy Manual, 4.4 "debian/changelog" 中有详细的描述。这种格式被 **dpkg** 和其他程序用以解析版本号信息、适用的发行版和紧急程度。

对于你而言，详细描述你所做出的更改也是很好且很重要的。它将帮助下载你的软件包的人了解这个软件包中是否有他们需要知道的事情。它会被作为 **/usr/share/doc/gentoo/changelog.Debian.gz** 保存在二进制包中。

dh_make 创建了一个默认的文件，这是它的容貌：

```
gentoo (0.9.12-1) unstable; urgency=medium

  * Initial release. (Closes: #nnnn)  <nnnn is the bug number of your ITP>

 -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100

```

(注：我为它添加了行号。)

第 1 行是软件包名、版本号、发行版和紧急程度。软件包名必须与实际的源代码包名相同，发行版应该是 **unstable**。除非有特殊原因，紧急程度默认设置为 medium（中等）。

第 3-5 行是一个很长的条目，记录了你在这个 Debian 修订版本中做出的修改(非上游修改——上游修改由上游作者创建并由另外一个文件维护，它们应被安装为 **/usr/share/doc/gentoo/changelog.gz**)。假设你的 ITP (Intent To Package，计划打包)的 Bug 号为 **12345**。新行必须插入在上一个以星号 * 开头的行的正下方。你可以使用 dch(1) 完成这个工作，也可以使用普通的文本编辑器手工完成，只要你遵循 dch(1) 所使用的格式。

为了阻止软件包在打包完成之前被意外上传，将发行版值改成一个不可用的 **UNRELEASED** 将是一个很好的选择。

最后它会成为以下的样子：

```
gentoo (0.9.12-1) UNRELEASED; urgency=low

 * Initial Release. Closes: #12345
 * This is my first Debian package.
 * Adjusted the Makefile to fix $(DESTDIR) problems.

-- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100

```

(注：我为它添加了行号。)

如果你已经对自己所作出的改动感到满意，而且它们都被记录在了 **changelog** 中，那么你就可以将发行版值由 **UNRELEASED** 修改至目标发行版值 **unstable** (甚至 **experimental**)。[36]

你可以在关于更新的 第 8 章 *更新软件包* 中了解更多关于 **changelog** 的内容。

4.4. rules

现在我们需要看看 dpkg-buildpackage(1) 用于实际创建软件包的 rules 文件。这个文件事实上是另一个 **Makefile**，但不同于上游源代码中的那个。和 **debian** 目录中的其他文件不同，这个文件被标记为可执行。

4.4.1. rules 文件中的 Target

每一个 **rules** 文件，就像其他的 **Makefile** 一样，包含着若干 rules，其中每一个都定义了一个 target 以及其具体 操作。[37] 一个新的 rule 以自己的 target 声明(置于第一列)来起头。后续的行都以 TAB 字符 (ASCII 9) 来开头，以指示 target 的具体行为。空行和以井号 **#** 开头的行会被当作注释而被忽略。[38]

当你想要执行一个 rule 的时候，就将 target（目标）名称作为命令行参数来调用。比如说，**debian/rules *build*** 以及 **fakeroot make -f debian/rules *binary*** 会分别执行 ***build*** 和 ***binary*** 两个 target。

以下是对各 target 的简单解释：

- **clean** target：清理所有编译的、生成的或编译树中无用的文件。(必须)
- **build** target：在编译树中将代码编译为程序并生成格式化的文档。(必须)
- **build-arch** target：在编译树中将代码编译为依赖于体系结构的程序。(必须)
- **build-indep** target：在编译树中将代码编译为独立于平台的格式化文档。(必须)
- **install** target：把文件安装到 **debian** 目录内为各个二进制包构建的文件树。如果有定义，那么 **binary*** target 则会依赖于此 target。(可选)
- **binary** target：创建所有二进制包(是 **binary-arch** 和 **binary-indep** 的合并)。(必须)[39]
- **binary-arch** target：在父目录中创建平台依赖(**Architecture: any**)的二进制包。(必须)[40]
- **binary-indep** target：在父目录中创建平台独立(**Architecture: all**)的二进制包。(必须)[41]
- **get-orig-source** target：从上游站点获得最新的原始源代码包。(可选)

可能你现在感到有些迷惑，在接下来讲解 dh_make 给出的默认的 **rules** 文件时事情会变得简单。

4.4.2. 默认的 rules 文件

新版本的 dh_make 会生成一个使用 dh 命令的非常简单但非常强大的默认的 **rules** 文件：

```
1 #!/usr/bin/make -f
2 # See debhelper(7) (uncomment to enable)
3 # output every command that modifies files on the build system.
4 #DH_VERBOSE = 1
5
6 # see FEATURE AREAS in dpkg-buildflags(1)
7 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
8
9 # see ENVIRONMENT in dpkg-buildflags(1)
10 # package maintainers to append CFLAGS
11 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
12 # package maintainers to append LDFLAGS
13 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
14
15
16 %:
17         dh $@
```

(注：我添加了行号并删去了一些注释。实际的 **rules** 文件里开头的空格是 TAB 填充的。)

可能在 shell 或 Perl 脚本中你已经对第一行的形式很熟悉了，它告诉操作系统这个文件应使用 **/usr/bin/make** 处理。

可以取消第 4 行的注释，以设置 **DH_VERBOSE** 变量为 1，于是 **dh** 命令就会输出它将要使用的 **dh_*** 命令。你也可以在此添加一行 **export DH_OPTIONS=-v**，于是 **dh_*** 命令同样也会输出它正在调用的命令。这能帮助你理解在这个简单的 **rules** 文件背后发生了什么，以及帮助你进行调试。新的 **dh** 被设计来作为 **debhelper** 工具的核心部分，并不向你隐藏任何东西。

第 16 和 17 行使用了 pattern rule，以此隐式地完成所有工作。其中的百分号意味着“任何 targets”，它会以 target 名称作参数调用单个程序 **dh**。[42] **dh** 命令是一个包装脚本，它会根据参数执行妥当的 **dh_*** 程序序列。[43]

- **debian/rules clean** 运行了 **dh clean**，接下来实际执行的命令为：

```
dh_testdir
dh_auto_clean
dh_clean
```

- **debian/rules build** 运行了 **dh build**，其实际执行的命令为：

```
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
```

- **fakeroot debian/rules binary** 执行了 **fakeroot dh binary**，其实际执行的命令为[44]：

```
dh_testroot
dh_prep
dh_installdirs
dh_auto_install
dh_install
dh_installdocs
dh_installchangelogs
dh_installexamples
dh_installman
dh_installcatalogs
dh_installcron
dh_installdebconf
dh_installemacsen
dh_installifupdown
dh_installinfo
dh_installinit
dh_installmenu
dh_installmime
dh_installmodules
dh_installlogcheck
dh_installlogrotate
dh_installpam
dh_installppp
dh_installudev
dh_installwm
dh_installxfonts
dh_bugfiles
dh_lintian
dh_gconf
dh_icons
dh_perl
dh_usrlocal
dh_link
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
```

- **`fakeroot debian/rules binary-arch`** 执行了 **`fakeroot dh binary-arch`**，其效果等同于 **`fakeroot dh binary`** 并附加 **`-a`** 参数于每个命令后。
- **`fakeroot debian/rules binary-indep`** 执行了 **`fakeroot dh binary-indep`**，这会运行几乎和 **`fakeroot dh binary`** 一样的命令，但 **dh_strip**、**dh_makeshlibs** 和 **dh_shlibdeps** 除外，其他命令则均附加 **`-i`** 选项。

dh_* 命令的功能依其名称不言而喻。[45] 不过其中有一些值得在这里进行简要解释，假定有一个基于 **`Makefile`** 的典型构建环境：[46]

- **dh_auto_clean** 通常在 **`Makefile`** 存在且有 **`distclean`** target 时执行以下命令[47]

```
make distclean
```

- **dh_auto_configure** 在 **`./configure`** 存在时通常执行以下命令(省略部分参数以方便此处阅读)。

```
./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var ...
```

- **dh_auto_build** 通常使用以下命令执行 **`Makefile`** 中的第一个 target。

```
make
```

- **dh_auto_test** 通常在 **`Makefile`** 存在且有 **`test`** target 时执行以下命令。[48]

```
make test
```

- dh_auto_install 通常在 `Makefile` 存在且有 `install` target 时执行以下命令(进行了换行以便阅读)。

```
make install \
  DESTDIR=/path/to/package_version-revision/debian/package
```

所有需要 fakeroot 命令的都包含了 dh_testroot。如果你没有使用 fakeroot，那将会报错并退出。

关于 dh_make 生成的 `rules` 文件，你应该知道的最重要的事是，它仅仅是一个建议。它对多数简单的软件包有效，但对于更复杂的则要大胆对其进行定制以满足需要。

尽管 `install` target 不是必须的，但也被支持。`fakeroot dh install` 的操作就像 `fakeroot dh binary`一样，但停止于 dh_fixperms。

4.4.3. 定制 `rules` 文件

有很多方法来定制使用新的 dh 命令创建的 `rules` 文件。

`dh $@` 命令可以按以下方式定制：[49]

- 为 dh_python2 命令添加支持。(对于 Python 的最佳选择。)[50]
  - 在 `Build-Depends` 中添加 `python` 软件包。
  - 使用 `dh $@ --with python2`.
  - 这会使用 `python` 框架处理 Python 模块。
- 添加 dh_pysupport 命令的支持。(已废弃)
  - 在 `Build-Depends` 中添加 `python-support` 软件包。
  - 使用 `dh $@ --with pysupport`
  - 这会使用 `python-support` 框架处理 Python 模块。
- 添加 dh_pycentral 命令支持。(已废弃)
  - 在 `Build-Depends` 中添加 `python-central` 软件包。
  - 使用 `dh $@ --with python-central`
  - 这样会同时停用 dh_pysupport 命令。
  - 这会使用 `python-central` 框架处理 Python 模块。
- 添加 dh_installtex 命令支持。
  - 在 `Build-Depends` 中添加 `tex-common` 软件包。
  - 使用 `dh $@ --with tex`
  - 这样会注册 Type 1 字体、断句样式及其他 TeX 格式。
- 添加 dh_quilt_patch 和 dh_quilt_unpatch 命令支持。
  - 在 `Build-Depends` 中添加 `quilt` 软件包。
  - 使用 `dh $@ --with quilt`
  - 这会在你使用 `1.0` 格式的源代码包时自动应用或解除 `debian/patches` 目录中的补丁。
  - 如果你使用新的 `3.0 (quilt)` 源代码包格式则不需要这些。
- 为 dh_dkms 命令添加支持。
  - 在 `Build-Depends` 中添加 `dkms` 软件包。
  - 使用 `dh $@ --with dkms`
  - 这能使内核模块软件包正确使用 DKMS。
- 添加 dh_autotools-dev_updateconfig 和 dh_autotools-dev_restoreconfig 命令支持。

    - 在 **Build-Depends** 中添加 **autotools-dev** 软件包。
    - 使用 **dh $@ --with autotools-dev**
    - 这会自动更新或还原 **config.sub** 和 **config.guess** 文件。
- 添加 **dh_autoreconf** 和 **dh_autoreconf_clean** 命令支持。
    - 在 **Build-Depends** 中添加 **dh-autoreconf** 软件包。
    - 使用 **dh $@ --with autoreconf**
    - 这样会在编译时更新 GNU 编译系统文件并在编译后对其进行恢复。
- 添加 **dh_girepository** 命令支持。
    - 在 **Build-Depends** 中添加 **gobject-introspection** 软件包。
    - 使用 **dh $@ --with quilt**
    - 这会为带有 GObject 内省数据的软件包计算依赖，并生成 **${gir:Depends}** 这一替换变量。
- 添加 **bash** 补全特性支持。
    - 在 **Build-Depends** 中添加 **bash-completion** 软件包。
    - 使用 **dh $@ --with bash-completion**
    - 这会使用 **debian/*package*.bash-completion** 中的配置文件来安装 **bash** 补全。

很多由新的 **dh** 命令触发的 **dh_*** 都可以通过修改 **debian** 目录中的配置文件来对其行为进行定制。参看 第 5 章 *debian 目录下的其他文件* 和每个命令的 man 手册页。

某些由新的 **dh** 命令所触发的 **dh_*** 命令可能需要额外的参数，或需要附加执行或者跳过执行。对于这类情况，你可以在 **rules** 文件中创建一个 **override_dh_*foo*** target，并在其中定义一个 **override_dh_*foo*** 来使其完成你想要 **dh_*foo*** 命令作出的改变。它的作用简单说就是 *run me instead (把运行的命令换成我)*。[51]

请注意 **dh_auto_*** 命令为了照顾所有的边缘情况，它实际所做的比上述(过度)简化的步骤中介绍的内容更多。除了 **override_dh_auto_clean** 外把上面的简化命令写成 **override_dh_*** 中是不明智的，这样会使得 **debhelper** 的许多智能特性无法体现。

所以，比如，最近的**gentoo** 软件包使用了 Autotools，如果你希望把系统配置配置文件安装到 **/etc/gentoo** 而非通常的 **/etc** 目录，你可以凌驾 **dh_auto_configure** 默认的使用的 **--sysconfig=/etc** 参数，改为向 **./configure** 命令传递以下参数：

```
override_dh_auto_configure:
	dh_auto_configure -- --sysconfig=/etc/gentoo
```

在 **--** 其后给出的参数会被追加到被自动执行的程序默认参数后，以此凌驾它们并修改其默认行为。使用 **dh_auto_configure** 命令要比直接调用 **./configure** 命令好很多，因为 它只修改 **--sysconfig** 参数内容，同时保留其他任何对 **./configure** 命令良性的参数。

如果 **gentoo** 的 **Makefile** 需要指定 **build** 作为其编译用的 target[52]，你可以创建一个 **override_dh_auto_build** target 来启用它。

```
override_dh_auto_build:
	dh_auto_build -- build
```

这保证了 **$(MAKE)** 会使用 **dh_auto_build** 传递的所有默认参数并编译处理 **build** 这个 target。

如果 **gentoo** 的 **Makefile** 需要指定 **packageclean** target 来为 Debian 软件包作清理，而非 **distclean** 或 **clean** target，那你就可以创建一个 **override_dh_auto_clean** target 来启用它。

```
override_dh_auto_clean:
	$(MAKE) packageclean
```

如果 **gentoo** 的 **Makefile** 包含了一个 **test** target 但你不想在 Debian 软件包构建过程中运行它，可以使用空的 **override_dh_auto_test** target 来跳过它。

```
override_dh_auto_test:
```

如果 **gentoo** 有某个不常见的上游 changelog 文件名为 **FIXES**，默认情况下 **dh_installchangelogs** 不会安装它。**dh_installchangelogs** 命令需要将 **FIXES** 作为它的参数才会安装它。[53]

```
override_dh_installchangelogs:
	dh_installchangelogs FIXES
```

如果你使用新的 **dh** 命令时，还使用 第 4.4.1 节 “**rules** 文件中的 Target” 中除 **get-orig-source** 的显式 target，会使得其效果难以预料。如果可能的话请尽量避免使用独立的或预设的 target，如果必须修改默认设置则酌情使用 **override_dh_***。

---

[27] 在本章节中，只要不产生歧义，所有提及的 **debian** 目录下的文件均会省去 **debian/** 前缀以求简洁方便。

[28] 参见 Debian Policy Manual, 2.4 "Sections" 以及 List of sections in **sid**.

[29] 参见 Debian Policy Manual, 2.5 "Priorities"。

[30] 参见 Debian Policy Manual, 7.7 "Relationships between source and binary packages - Build-Depends, Build-Depends-Indep, Build-Conflicts, Build-Conflicts-Indep"。

[31] 这种奇怪的情况是 Debian Policy Manual, Footnotes 55 中详细描述的一种特性。这不是由于在 **debian/rules** 中使用 **dh** 命令所致的，真正的原因是 **dpkg-buildpackage** 的运行方式。相同的情形也适用于 Ubuntu 的自动编译系统。

[32] 确切信息请参见 Debian Policy Manual, 5.6.8 "Architecture"。

[33] 参见 Debian Policy Manual, 7 "Declaring relationships between packages"。

[34] 这些描述都使用英语。相应的翻译由 The Debian Description Translation Project - DDTP（Debian 描述翻译项目）项目提供。

[35] 参见 Debian Developer's Reference, 6.2.5. "Version Control System location"。

[36] 如果你用 **dch -r** 命令来使它成为最后一笔更改，请确保用编辑器显式地保存 **changelog** 文件。

[37] 你可以通过该资源来学习编写 **Makefile**：Debian Reference, 12.2. "Make"。完整文档在 http://www.gnu.org/software/make/manual/html_node/index.html 或者 **make-doc** 软件包（该包位于 **non-free** 部分）。

[38] Debian Policy Manual, 4.9 "Main building script: debian/rules" 针对细节进行了解释。

[39] 此 target 被 **dpkg-buildpackage** 用于 第 6.1 节 “完整的(重)构建” 描述的过程中。

[40] 此 target 被 **dpkg-buildpackage -B** 用于第 6.2 节 “自动编译系统” 描述的过程中。

[41] 此 target 被 **dpkg-buildpackage -A** 使用。

[42] 此处使用了新版本 **debhelper** v7+ 的特性。它的设计理念在 Not Your Grandpa's Debhelper 中进行了阐明，这在 DebConf9 中被 **debhelper** 上游进行了演示。在 **lenny** 下，**dh_make** 会创建一个更为复杂 的 **rules** 文件，伴有许多显式的 rule 和许多为每个 rule 所用的**dh_*** 脚本，其中有大部分在现在已经是不必要的了(这也显示了软件包的年龄)。新一代的 **dh** 命令更为简洁，并能将我们从"手工的重复工作"中解放出来。当然，你仍然拥有完全的力量来定制整个过程，只要使用 **override_dh_*** target。参见 第 4.4.3 节 “定制 **rules** 文件”。它仅仅基建于 **debhelper** 软件包，而且不会像 **cdbs** 软件包所倾向的那样混淆软件包构建过程。

[43] 你可以检验每一个已有的 ***target*** 所调用的实际 **dh_*** 程序序列，而并不需要真的通过 **dh *target* --no-act** 或 **debian/rules -- '*target* --no-act'** 来执行以查看。

[44] 下边的例子假定你的 **debian/compat** 有一个 值大于或等于 9，以此避免自动调用任何 python 支持命令。

[45] 关于所有 **dh_*** 脚本具体行为，以及 它们有什么选项 的完整信息，请阅读它们各自的 man 手册页 以及 **debhelper** 的文档。

[46] 这些构建系统同样支持其他的构建环境，比如 **setup.py**，这可以通过在软件包源码目录中执行 **dh_auto_build --list** 来列出。

[47] 它实际上在 **Makefile** 中搜寻第一个可用的 target，除了 **distclean**, **realclean**, 或 **clean**，接下来再执行它。

[48] 实际上它在 **Makefile** 中搜索第一个可用的 target，除了 **test** 或 **check**，然后执行它。

[49] 如果一个软件包安装了 **/usr/share/perl5/Debian/Debhelper/Sequence/*custom_name*.pm** 文件，你应当使用 **dh $@ --with *custom-name*** 命令激活其功能。

[50] 在 **dh_python2** 和 **dh_pysupport** 以及**dh_pycentral** 命令之间更推荐使用前者。不要使用 **dh_python** 命令。

[51] 在 **lenny** 下，如果你希望更改某个 **dh_*** 脚本的行为，你需要在 **rules** 中找到相应的行然后进行调整。

[52] 没有参数的 **dh_auto_build**命令将执行 **Makefile** 中的第一个 target。

[53] **debian/changelog** 和 **debian/NEWS** 总是会被自动安装。程序会将文件名转为小写并搜索以下文件名来检测上游 changelog：**changelog**、**changes**、**changelog.txt**和**changes.txt**。