Debian 新维护者手册

本文档在撰写过程中参考了以下两篇文档：

- Making a Debian Package (AKA the Debmake Manual), copyright © 1997 Jaldhar Vyas.
- The New-Maintainer’s Debian Packaging Howto, copyright © 1997 Will Lowe.

COLLABORATORS

	TITLE : Debian 新维护者手册	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Josip Rodin、Osamu Aoki（青木修）、Aron Xu、李凌、郑原真、周默和杨博远	November 3, 2017	
		November 3, 2017	
		November 3, 2017	
		November 3, 2017	
		November 3, 2017	
		November 3, 2017	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

正确的起点

本文档致力于为普通 Debian 用户，和希望对 Debian 软件包有所了解的开发人员讲述如何制作 Debian 软件包。在这里，我们尽可能使用通俗的语言，并辅以实例来直观地展示每一个细节。有一句古罗马谚语说得好：一例胜千言！

The rewrite of this tutorial document with updated contents and more practical examples is available as Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) . Please use this new tutorial as the primary tutorial document.

This document is made available for the Debian `Buster` release since it is offered in many translations but this will be dropped in the following releases since contents are getting outdated. [1]

Debian 的软件包系统是使它跻身顶级发行版行列的重要原因之一。尽管已经有相当数量的软件被打包成 Debian 的格式，但有时还是需要安装一些并非该格式的软件。可能你正为如何制作自己的软件包而感到迷惑，也可能认为这么做很难。如果你是一个刚刚接触 Debian 的初学者，那么是的，它的确很难；不过假如你真的只是一个初入此门的新手，现在大概也不会来读这篇文档了。:-) 你的确需要对 Unix 编程有所了解，但显然没必要是这方面的天才。[2]

对于 Debian 软件包维护人员来说，有一件事是非常明确的：创建并维护一个 Debian 软件包需要花费很多精力，所需的时间很可能远不只是几个小时。维护人员需要有良好的技术基础，同时也需要十分勤奋，这样才能保证我们的系统正常运行而不出现问题。

如果你在软件包制作方面需要他人帮助，请阅读第 1.4 节。

本文的最新版随时都可以在 http://www.debian.org/doc/maint-guide/ 上和 `maint-guide` 软件包里找到。文档的简体中文翻译可以在 `maint-guide-zh-cn` 软件包里找到。还有一点需要注意的是，这篇文档的内容相对于当前的开发情况可能会有略微的延迟。

由于这是一份手把手的教程，所以在一些重要的话题上会对每个步骤都做详细的解释。虽然你可能觉得它们之中有一些与你的想法毫不相干，但也请准备好足够的耐心来学习。另外，我也有意地省略了某些不必要的细节，以使这篇文档尽可能保持简洁。

1.1 Debian 的社会驱动力

以下是一些有关 Debian 的社会动力学报告，希望它们能够帮助你做好准备，以与 Debian 进行交互：

- 大家都是志愿者。
  - 任何人都不能把事情强加给他人。
  - 你应该主动地做自己想做的事情。

[1] 在写这份文档时，我们默认你使用 `jessie` 或者更新的操作系统。如果你需要在 更老版本的系统上（包括老版本的 Ubuntu 等）使用本文所记述的方法，则至少必须安装 backports 仓库中的 `dpkg` 和 `debhelper` 软件包。

[2] 在 Debian Reference (http://www.debian.org/doc/manuals/debian-reference/) 中，你可以了解到使用 Debian 系统的一些基本方法和关于 Unix 编程的一些指引。

- 友好的合作是我们前行的动力。
  - 你的贡献不应致使他人增加负担。
  - 只有当别人欣赏和感激你的贡献时，它才有真正的价值。
- Debian 并不是一所学校，在这里没有所谓的老师会自动地注意到你。
  - 你需要有自学大量知识和技能的能力。
  - 其他志愿者的关注是非常稀缺的资源。
- Debian 一直在不断进步。
  - Debian 期望你制作出高质量的软件包。
  - 你应该随时调整自己来适应世界的变化。

在 Debian 社区中有这几类常见的角色：

- **Upstream author** (上游作者)：程序的原始作者。
- **Upstream maintainer** (上游维护者)：目前在上游维护程序代码的人。
- **Maintainer** (软件包维护者)：制作并维护该程序 Debian 软件包的人。
- **Sponsor** (赞助者)：帮助维护者上传软件包到 Debian 官方仓库的人（在通过内容检查之后）。
- **Mentor** (导师)：帮助新手维护者熟悉和深入打包的人。
- **Debian Developer** (DD，Debian 开发者)：Debian 社区的官方成员。DD 拥有向 Debian 官方仓库上传的全部权限。
- **Debian Maintainer** (DM，Debian 维护者)：拥有对 Debian 官方仓库部分上传权限的人。

注意，你不可能在一夜之间成为 **Debian Developer**，因为成为 DD 所需要的远不只是技术技巧。不过别因此而气馁，如果你的软件包对其他人有用，你可以当这个软件包的 **Maintainer**，然后通过一位 **Sponsor** 来上传这份软件，或者你可以申请成为 **Debian Maintainer**。

还有，要成为 Debian Developer 不一定要创建新软件包。对已有软件做出贡献也是成为 Debian Developer 的理想途径。眼下正有很多软件包等着好的维护者来接手 (参看第 2.2 节)。

在这篇文档里，我们的重点在于制作软件包的技术细节。有关 Debian 是如何运转的，以及如何才能参与到其中之类的话题，请参考下边的文档：

- Debian: 17 years of Free Software, ”do-ocracy”, and democracy (http://upsilon.cc/~zack/talks/2011/20110321-taipei.pdf) (幻灯片介绍)
- How can you help Debian? (http://www.debian.org/intro/help) (官方文档)
- The Debian GNU/Linux FAQ, Chapter 13 - ”Contributing to the Debian Project” (http://www.debian.org/doc/FAQ/ch-contributing) (半官方文档)
- Debian Wiki, HelpDebian (http://wiki.debian.org/HelpDebian) (补充材料)
- Debian New Member site (https://nm.debian.org/) (官方站点)
- Debian Mentors FAQ (http://wiki.debian.org/DebianMentorsFaq) (补充文档)

1.2 需要的开发工具

在开始之前，请确认你是否已经正确安装了开发所需要的工具集。注意这些软件包中没有任何一个会被标记为 `essential` 或 `required` ——这里假设你已经安装了它们。

以下这些软件包已经随标准的 Debian 安装过程进入了系统，所以你可能不需要再动手安装它们（以及任何附加的依赖软件包）。然而，你还是应该用 `aptitude show` *package* 或者 `dpkg -s` *package* 来检查一下。（译注：apt show PACKAGE 亦可）

在你的开发环境中，最重要的软件包是 `build-essential`。一旦你尝试安装该包，它将 拉取其他基本构建环境所需的工具链。

对于某些类型的软件，以上的就是所需要的全部。然而还有一组工具虽不是对于所有软件包都必须，却可能对你有用，或者你的软件包制作过程中会需要它们：

- `autoconf`、`automake` 和 `autotools-dev` - 很多新程序使用 configure 脚本和 `Makefile` 文件来帮助预处理程序。(参看 `info autoconf`、`info automake`)。`autotools-dev` 则用于保持指定的自动配置文件为最新，并带有关于使用那些文件的最佳方法的文档。
- `debhelper` 和 `dh-make` - `dh-make` 是用于创建我们示例软件包骨架所必须的，它会使用 `debhelper` 中的一些工具来创建软件包。他们不是创建软件包所必须的，但对新维护人员而言，我们 强烈推荐。它可以使整个过程极为简化，并易于在将来维护。(参看 dh_make(8)、debhelper(1)、`/usr/share/doc/debhelper/README`) [3]

  新的 `debmake` 可以作为标准 `dh-make` 的代替品。debmake 能做的事情更多，并且拥有包含非常多打包实例的 HTML 文档。文档可以通过 `debmake-doc` 软件包获取。
- `devscripts` - 此软件包提供了一些非常好且有用的脚本来帮助维护者，不过这写脚本并非制作软件包所必须。此软件包所推荐或建议的软件包都值得一看。(参看 `/usr/share/doc/devscripts/README.gz`)
- `fakeroot` - 这个工具使你可以在编译过程中必要的时候以普通用户来模拟 root 用户环境。(参看 fakeroot(1))
- `file` - 这个小程序可以检测文件的类型。(参看 file(1))
- `gfortran` - GNU Fortran 95 编译器，如果你的程序是用 Fortran 编写的则必须用此工具完成编译。(参看 gfortran(1))
- `git` - 此软件包提供了用于快捷处理大型项目的著名版本控制系统 - git。它被广泛用于各种开源项目，其中最著名的是 Linux 内核项目。(参见 git(1), git Manual (`/usr/share/doc/git-doc/index.html`).)
- `gnupg` - 让你可以使用 数字签名签署你的软件包。当你想把它分发给其他人时这一点特别重要。如果你要把你的成果加入到 Debian 发行版中，那这是必须的步骤。(参看 gpg(1).)
- `gpc` - GNU Pascal 编译器。如果你的程序是用 Pascal 写的则需要此工具。值得一提的是 `fp-compiler`，Free Pascal 编译器 (FPC)，也能够很好地胜任编译任务。(参见 gpc(1), ppc386(1).)
- `lintian` - Debian 软件包检查工具，使你可以在编译软件包后知道它是否犯了常见的错误，并对其找到的错误进行解释。(参见 lintian(1), Lintian User's Manual (https://lintian.debian.org/manual/index.html) .)
- `patch` - 这是一个非常有用的工具，它可以把 **diff** 程序生成的差异清单文件应用到原先的文件上，从而生成一个打了补丁的版本。(参看 patch(1))
- `patchutils` - 此软件包提供了一些帮助处理补丁的工具，如 **lsdiff**、**interdiff** 和 **filterdiff** 命令。
- `pbuilder` - 此软件包提供了创建和维护 **chroot** 环境的工具。在它的 **chroot** 环境中编译 Debian 软件包可以检查编译依赖是否合适，并避免 FTBFS (Fails To Build From Source，源代码编译失败) 的 Bug。(参看 pbuilder(8) 和 pdebuild(1))
- `perl` - Perl 是现今类 Unix 系统中使用最普遍的解释型脚本语言，它常被称作 Unix 的瑞士军刀。(参看 perl(1))
- `python` - Python 是 Debian 系统中另一个最常用的解释型脚本语言，它拥有着可圈可点的强大功能和十分清晰的语法。(参看 python(1))
- `quilt` - 此软件包帮助你管理一系列的补丁。它们被以逻辑栈的方式组织在一起。你可以 apply (=push)、un-apply (=pop) 或简单地刷新它们然后再放入栈内。(参看 quilt(1), and `/usr/share/doc/quilt/quilt.pdf.gz`.)

[3] 还有几个类似但更针对某一类软件的工具，如 `dh-make-perl`、`dh-make-php` 等。

- `xutils-dev` - 一些通常用于 X11 的程序，使用其宏功能可以生成 `Makefile` 文件。(参看 imake(1)、xmkmf(1))

以上给出的简短描述仅仅是为了使你对这些工具有一个基本的印象。在继续前请详细阅读每个程序（包括通过依赖关系安装的程序，比如 **make**）的文档，至少了解其一般的用途和用法。现在看来这是一项耗时巨大的任务，但在接下来的工作中你将为你阅读了它们而感觉到 非常愉快。如果一会你遇到一些特定的问题，我会建议你重新阅读上面提到的文档。

1.3 需要的开发文档

以下是 非常重要的文档，你应该在读本文档时同时参看它们：

- `debian-policy` - the Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) 包含了对 Debian 软件仓库、操作系统设计问题、文件系统层级标准 (FHS，Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-2.3.html)，讲述每个文件和目录应该放在哪里) 等的描述。对于你而言，最重要的是它描述了软件包进入官方仓库前必须满足的条件。(请参见 `/usr/share/doc/debian-policy/policy.pdf.gz` 和 `/usr/share/doc/debian-policy/fhs/fhs-2.3.pdf.gz` 的本地副本)
- `developers-reference` - Debian 开发者参考 (http://www.debian.org/doc/devel-manuals#devref) 描述了打包所需的包含技术细节在内的全部详细信息，如仓库结构、如何重命名/丢弃/接手软件包、如何进行 NMU(非维护者上传)、如何管理 Bug 以及打包最佳实践、何时向何处上传等。(参见 `/usr/share/doc/developers-reference/developers-reference.pdf` 的本地副本)

以下是 重要的文档，你应该在读本文档时同时参看它们：

- Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) 为 the GNU Build System known as the GNU Autotools 中最重要的工具——Autoconf、Automake、Libtool 和 gettext 提供了很好的文档。
- `gnu-standards` - 此软件包包含了 GNU 项目中的两篇文档：GNU Coding Standards (http://www.gnu.org/prep/standards/html_node/index.html) 和 Information for Maintainers of GNU Software (http://www.gnu.org/prep/maintain/html_node/index.html)。尽管 Debian 不要求遵守这些规范，但它们作为纲领和共识仍然很有帮助。(参见 `/usr/share/doc/gnu-standards/standards.pdf.gz` 和 `/usr/share/doc/gnu-standards/maintain.pdf.gz` 的本地副本)

若本文档所叙述的内容与 Debian Policy Manual 或 Debian Developer's Reference 有不符，则按照后两者的要求为准，并向 `maint-guide` 软件包提交 Bug 报告。

以下是替代性的教程文档，你可以在读本文档时同时参看它们：

- Debian Packaging Tutorial (http://www.debian.org/doc/packaging-manuals/packaging-tutorial/packaging-tutorial)

1.4 到何处求助

在你决定到公共场合提问之前，请先阅读这些（个）不错的文档：

- 所有相关软件包的 `/usr/share/doc/`*package* 目录之中的文件
- 所有相关命令的 **man** *command* 手册页
- 所有相关命令的 **info** *command* info 页
- 邮件列表档案 debian-mentors@lists.debian.org (http://lists.debian.org/debian-mentors/)
- 邮件列表档案 debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/)

在搜索时你可以在搜索引擎中使用类似这样的字符串:`site:lists.debian.org`，以此限制域名从而提高效率。

制作小的测试软件包是学习打包细节的好方法，仔细查看维护较好的软件包则是了解他人如何制作软件包的最佳办法。

如果你仍然对打包存有疑问，并且在文档和 WEB 资源中都不能找到答案，你可以交互式地向他们提问：

- http://lists.debian.org/debian-mentors/ (http://lists.debian.org/debian-mentors/) 邮件 debian-mentors@lists.debian.org (http://lists.deb debian-mentors/) . (该邮件列表是新手专区。)
- debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) . (该邮件列表汇集各路神仙。)
- IRC (http://www.debian.org/support#irc) 比如 `#debian-mentors`。
- 专注于某个软件包集合的团队。(完整列表参见 https://wiki.debian.org/Teams (https://wiki.debian.org/Teams))
- 特定语言的邮件列表，比如 debian-devel-{french,italian,portuguese,spanish}@lists.debian.org 或 debian-devel@debian.or.jp. (完整列表参见 https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) 和 https://lists.debian.org/users.html (https://list users.html))

如果你付出了一定的努力并且提问得当，那么有经验的 Debian 开发者会很乐意帮助你的。

当你收到一个 Bug 报告后 (没错，真正的 Bug 报告！)，你需要研究 Debian Bug Tracking System (http://www.debian.org/-Bugs/) (Debian Bug 追踪系统，BTS) 并阅读相关的文档以便高效处理这些报告。我在此强烈推荐阅读 Developer's Reference, 5.8. "Handling bugs" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling) 。

即使上边的那些问题都解决了，也不能高兴得太早。为什么？因为几个小时或几天内就会有人开始使用你的软件包，如果你犯了某些严重的错误，将会被无数生气的 Debian 用户进行邮件轰炸……哦，当然这只是开个玩笑。:-)

放松一点并准备好处理 Bug 报告，在你的软件包完全符合 Debian 的各项规范前你还需要付出很多努力，处理 Bug 对你也是很好的锻炼 (再一次提醒，阅读那些 必须的文档来了解详情)。祝你好运！

Chapter 2

第一步

让我们来创建一个自己的软件包 (更好的是，“领养”一个已存在的软件包)。

2.1 Debian 软件包制作流程

如果你的工作基于某个上游程序，那么典型的 Debian 软件包制作流程就会需要如下几个特定的文件：

- 获取一份上游软件的拷贝，它通常为压缩过的 tar 格式。
  - *package-version*.tar.gz
- 在上游源码的 debian 目录下添加 Debian 打包的专用文件 (构建 Debian 软件包它们会被读取)，同时以 3.0 (quilt) 格式创建一个非本地源码包。
  - *package_version*.orig.tar.gz
  - *package_version-revision*.debian.tar.gz[1]
  - *package_version-revision*.dsc
- 用 Debian 源码包构建 Debian 二进制包；这些二进制包的格式通常是 .deb (或者 .udeb，Debian Installer 专用)
  - *package_version-revision_arch*.deb

请注意一个细节，在 Debian 软件包的文件名中，分隔 *package* 和 *version* 的字符从 tarball 名称中的 - (连字符) 换成了 _ (下划线)。

在上述的文件名中，请根据 Debian Policy，将 *package* 这个部分替换为 **package name**，将 *version* 这个部分替换为 **upstream version**，将 *revision* 这个部分替换为 **Debian revision**，以及将 *arch* 这个部分替换为 **package architecture**。[2]

本概要中的每一步，我们都会在后续的章节中辅以详细的例子进行解释。

[1] 对于老式的 1.0 格式非本地 Debian 源码包，应当使用 *package_version-revision*.diff.gz 这个命名规则。

[2] 参见 5.6.1 ”Source” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source) , 5.6.7 ”Package” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Package) , 以及 5.6.12 ”Version” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version) 。**package architecture** 遵循 Debian Policy Manual, 5.6.8 ”Architecture” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) 并且会在软件包构建的过程中被自动分配。

2.2 挑一个你喜欢的程序

可能你已经想好了要制作的软件包。那此时第一件要做的事，就是检查它是否已经被别人打包，并已经安置于发行版仓库中。这里有几种检查方法供您参考：

- **aptitude** 命令
- the Debian packages (http://www.debian.org/distrib/packages) 页面
- the Debian Package Tracker (https://tracker.debian.org/) web page

如果软件包已经存在于仓库中，那直接装上它就可以了呀！:-) 如果它是被 甩掉了 **(orphaned)** 的——也就是说它的维护者被设置为 Debian QA Group (http://qa.debian.org/)，这时候你可以尝试接手维护它。另外，你也可以“领养”那些维护者发送过“Request for Adoption”(**RFA**) 请求的软件包。[3]

软件包归属状态有这几种：

- **wnpp-alert** 命令，来自 `devscripts` 软件包
- Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp/)
- Debian Bug report logs: Bugs in pseudo-package `wnpp` 位于 `unstable` (http://bugs.debian.org/wnpp)
- Debian Packages that Need Lovin' (http://wnpp.debian.net/)
- Browse `wnpp` bugs based on debtags (http://wnpp-by-tags.debian.net/)

作为旁注必须指出，Debian 已经拥有了海量各种类型的软件包，而且处在仓库中软件包的数量也远远超过了拥有上传权限的贡献者数量。因此，为已经在仓库中的软件包贡献力量是非常受其他开发者欢迎的 (且更容易获得 sponsorship)[4]。有非常多的方式可以实现这一目的：

- 接手那些已经被甩掉，而仍然有很多人在使用的软件包
- 加入一些 打包小组 (http://wiki.debian.org/Teams)
- 帮忙为某些常用软件分类 Bug
- 在必要时，帮忙准备 QA 或 NMU 上传 (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload)

如果你有能力“领养”那个软件包，那就先下载 (使用 `apt-get source` *`packagename`* 或其他类似的工具) 并分析它的源代码。这篇文档不会详细说明如何领养软件包，不过幸运的是，领养软件包时，打包的起始工作已经有人完成，接手的工作应比从头开始轻松得多。尽管如此也请您不要轻敌，请继续阅读，下面给出的建议会对你很有帮助。

如果您要制作的软件包是完全崭新的，而您又希望它出现在 Debian 中，那请遵循以下的步骤：

- 首先，你必须知道这个软件能不能干活，而且你需要亲自试用一段时间来确保其可用性。
- 一定要在 Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp/being_packaged) 上仔细查看，以确定并没有其他人已经开始相关的工作。如果没有的话，则可以提交一份 ITP (Intent To Package，打包意向) Bug 报告给 `wnpp` 虚包 (可以使用 **reportbug**)。如果你看到已经有人在处理相关事宜，则在有需要的情况下再联系他 (们)。如果他 (们) 不需要你的帮助，那就寻找其他你感兴趣，而且没有人维护的软件包咯。
- 软件 必须有许可证。
  - 对于仓库中 `main` 区的软件，Debian Policy 要求其 完全兼容 **Debian Free Software Guidelines**（**Debian** 自由软件准则）(DFSG (http://www.debian.org/social_contract#guidelines)) 并且它 不能要求使用 **`main`** 区以外的软件来编译或执行。这即是最理想的状况。

[3] 参见 Debian Developer's Reference 5.9.5. "Adopting a package" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#adopting).
[4] 当然了，无论何时，总会有一些值得打包，而且并未进入 Debian 仓库的新软件。

  - 对于仓库中 `contrib` 区的软件，其许可证必须满足 DFSG 的全部条件，不同于 main 区软件的一点是，它们可以依赖于 `main` 之外的软件包来完成编译或运行。
  - 对于仓库中 `non-free` 区的软件，其许可证可以不满足 DFSG 中的一部分条件。其中坚决不能违背的一点是，该软件 必须是可分发的。
  - 如果你不确定你的软件应该归入仓库的哪一个区，那你可以把许可证文本发送到 debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) 邮件列表上寻求意见。
- 程序 不能给 Debian 系统带来安全或维护问题。
  - 程序应当带有良好的文档，最好是源代码也容易理解（比如，不混乱）。
  - 你应该与程序的作者取得联系，问问他是否认为程序应当被打包，以及他是否对 Debian 持友好态度。能够询问作者关于程序的任何问题都非常重要，所以千万不要尝试打包一个无人维护的软件（遇到麻烦找谁去呢）。
  - 程序一定 不能 setuid 到 root 。如果它不 setuid 或 setgid 到任何用户或组的话，那就再好不过了。
  - 程序不应该是守护进程，也不应该进入任何一个 `*/sbin` 目录，不应该以 root 权限打开任何端口。

当然，这些都是为了安全，并试图避免你在，比如 setuid 守护进程等问题上犯错误而激怒了用户... 当你在打包方面有了更多经验时，就能够处理这样的软件包了，不必着急。

我们鼓励你，作为一个新维护者，选择易于打包和维护的软件，而不鼓励选择复杂的软件包。

- 简单软件包
  - 生成单个二进制包，arch = all（比如像壁纸那样的资料集）
  - 生成单个二进制包，arch = all（用解释型语言编写的可执行脚本文件，比如 POSIX shell 语言）
- 中等复杂软件包
  - 生成单个二进制包，arch = any（用 C/C++ 等语言编写的 ELF 二进制可执行文件）
  - 生成多个二进制包，arch = any + all（包含 ELF 二进制可执行程序 + 文档）
  - 既不是 `tar.gz` 也不是 `tar.bz2` 格式的上游源代码包
  - 包含不可分发的内容物的源码包
- 高复杂度软件包
  - 被其他软件包使用的解释器模块包
  - 被其他软件包使用的 ELF 库文件包
  - 生成多个二进制包，其中包含括 ELF 库文件
  - 有多个上游的源码包
  - 内核模块包
  - 内核补丁包
  - 含有关键维护者脚本的软件包

打包高复杂软件包并非难如登天，但这个过程需要用到更多知识。你应该针对每一个所谓的复杂特性来搜寻相应的指南。比如，一些语言有它们自己的子策略文档：

- Perl policy (http://www.debian.org/doc/packaging-manuals/perl-policy/)
- Python policy (http://www.debian.org/doc/packaging-manuals/python-policy/)
- Java policy (http://www.debian.org/doc/packaging-manuals/java-policy/)

还有一句拉丁谚语：*fabricando fit faber*（熟能生巧）。我们 强烈建议你在阅读这篇教程的时候，用一个简单的软件包来对所有的 Debian 打包步骤进行实验。跟随下边的步骤您就可以创建一个微不足道的软件包 `hello-sh-1.0.tar.gz`，而且这会是一个非常好的开端：[5]

[5] 不用担心失踪的 `Makefile`。你可以参照第 5.11 节，简单地通过 **debhelper** 来安装 **hello** 程序，或者修改上游源代码来添加带有 `install` 目标的新`Makefile`，参照第 3 章。

```
$ mkdir -p hello-sh/hello-sh-1.0; cd hello-sh/hello-sh-1.0
$ cat > hello <<EOF
#!/bin/sh
# (C) 2011 Foo Bar, GPL2+
echo ”Hello!”
EOF
$ chmod 755 hello
$ cd ..
$ tar -cvzf hello-sh-1.0.tar.gz hello-sh-1.0
```

2.3 获取程序并试用

这里第一件事是找到并下载原始的源代码。我们假定你已经在作者的主页上找到了程序的源代码。一般来说，Unix 下的自由软件源代码是以 **tar**+**gzip** 格式 (扩展名为 `.tar.gz`) 或 **tar**+**bzip2** 格式 (扩展名为 `.tar.bz2`) 或 **tar**+**xz** (扩展名为 `.tar.xz`) 的文件格式提供的。有时候，归档文件中包含了一个名为 *`package-version`* 的子目录，在那里边有全部的源代码。

如果最新版本的源代码可通过像 Git，Subversion，CVS 这样的版本控制系统获得的话，你可以用 `git clone`，`svn co`, 或 `cvs co` 来下载它，然后将它重新打包压缩为 **tar**+**gzip** 格式，同时别忘了 `--exclude-vcs` 选项哟。

如果你的程序源代码是以其他文件格式提供的 (比如文件名以 `.Z` 或 `.zip` 结尾 [6])，则应当使用对应和合适的工具将其解压，再重新归档为 Tarball。

如果你挑选的程序的源代码中包含一些不符合 DFSG 的内容，你应当解压后移除它们，然后将删减过的源码重新归档。因为 DFSG 而进行删减过的源码归档的上游版本号中，需要添加 `dfsg` 这个标识。

作为示例，本教程在这里使用一个名为 **gentoo** 的程序，它是一个 GTK+ 文件管理器。[7]

在你的家目录下创建一个子目录，命名为 `debian` 或 `deb` 或者依你的喜好。(本例中使用 `~/gentoo`)。把下载好的归档文件放在其中并解压 (使用 `tar xzf gentoo-0.9.12.tar.gz` 命令)。你需要确定这个归档在解压过程中没有任何发生错误，即便是有一点 小小小问题也是不行的。因为在别人的系统上解压这些归档时，可能他们的工具并不会忽略这些非正常现象，于是就出问题了。一般的预期是，在你的终端屏幕上，应该能看到如下的情形。

```
$ mkdir ~/gentoo ; cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xvzf gentoo-0.9.12.tar.gz
$ ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
```

现在解压缩工具创建了一个新的子目录，名为 `gentoo-0.9.12`。接下来你需要进入该目录并 彻底读完其中的文档。通常情况下文档会被命名为 `README*`、`INSTALL*`、`*.lsm` 或 `*.html`。同时，你需要学会正确编译和安装这个程序 (最可能出现的一种情况是，软件会被默认安装到 `/usr/local/bin` 目录；但在 Debian 软件包制作过程中不可以那样做。详细的原因在第 3.3 节中说明)。

开始打包时的源代码目录应当是绝对干净原始，没有做过任何修改的。一般来说直接使用刚刚解压得到的源代码即可。

2.4 简单的编译系统

有些比较简单的程序源码自己带有 `Makefile`, 这时你可以很容易地使用 `make` 命令来编译它。[8] 有一些软件的 Makefile 还支持 `make check`，这个命令可以完成一系列程序检验和测试。当程序编译好后即可用 `make install` 命令，将程序安装到目标目录。

[6] 当通过文件扩展名不足以判断文件类型时，可以使用 **file** 命令来判断。

[7] 这个程序已经被打包好了。当前的版本 (http://packages.qa.debian.org/g/gentoo.html) 使用 Autotools 作为其编译系统 (build structure)，并且已经和下边的例子不一样了。下边的例子基于老的版本 0.9.12 。

[8] 许多新时代的程序都配有一个叫做 `configure` 的脚本。执行它的时候会生成一个为你的计算机专门定制的 `Makefile` 。

现在尝试编译和运行你的程序，你需要确保它能正常工作，以及它在安装和运行时不会破坏别的东西。

你还可以运行 `make clean` (或更好的 `make distclean`) 来清理编译目录。Makefile 中有时还会支持 `make uninstall`，它被用来卸载已经安装了的程序文件。

2.5 常见的可移植编译系统

有非常非常多的自由软件是使用 C 和 C++ 语言编写的。其中又有很多程序使用了 Autotools 或 CMake 来使其可以移植到不同平台上。这些工具被用于生成 `Makefile` 和其他必须的源文件。然后程序就可以使用正常的 `make; make install` 来编译和安装。

Autotools 是 GNU 编译系统工具集，包括 Autoconf、Automake、Libtool 和 gettext。你可以通过 `configure.ac`、`Makefile.am` 和 `Makefile.in` 等特征文件来识别使用 Autotools 作为编译系统的源代码。[9]

Autotools 工作流程的第一步是在上游作者在代码中运行 `autoreconf -i -f`，然后把生成的文件同源代码一起分发。

```
configure.ac-----+-> autoreconf -+-> configure
Makefile.am -----+        |      +-> Makefile.in
src/Makefile.am -+        |      +-> src/Makefile.in
                          |      +-> config.h.in
                       automake
                       aclocal
                       aclocal.m4
                       autoheader
```

编辑 `configure.ac` 和 `Makefile.am` 等文件前，你需要一些关于 **autoconf** 和 **automake** 的知识。请参看 `info autoconf` 和 `info automake`。

Autotools 工作流程的第二步是，用户获得分发的源代码后在源码目录下运行 `./configure && make` 来将其编译成为 ***binary*** (二进制文件)。

```
Makefile.in -----+                +-> Makefile -----+-> make -> binary
src/Makefile.in -+-> ./configure -+-> src/Makefile -+
config.h.in -----+                +-> config.h -----+
                 |
  config.status -+
  config.guess --+
```

你可以改变 `Makefile` 文件的许多设置，比如你可以修改默认的文件安装路径 (使用 `./configure --prefix=/usr`)。

可选：若使用 `autoreconf -i -f` 来更新 `configure` 和其他相关文件，则有可能可以提高源代码的兼容性。[10]

CMake 是另一个备选的编译系统，你可以通过 `CMakeLists.txt` 这个特征文件来识别使用 CMake 的源代码。

2.6 软件包名称和版本

如果上游源代码以 `gentoo-0.9.12.tar.gz` 这样的文件名来分发，你就可以用 `gentoo` 作为 软件包名，并用 `0.9.12` 作为 上游版本。它们会被 `debian/changelog` 这个文件用到；第 4.3 节部分会详细描述这个文件。

虽然此法在大部分情况下能显灵，但你仍需要根据 Debian 政策 (Debian Policy) 以及约定俗成的做法来调整 软件包名和 上游版本。

在 软件包名里只能含有小写字母 (`a-z`), 数字 (`0-9`), 加号 (`+`) 和减号 (`-`)，以及点号 (`.`)。软件包名最短长度两个字符；它必须以字母开头；它不能与仓库软件包名发生冲突。还有，把软件包名的长度控制在 30 字符以内是明智之举。[11]

[9] Autotools 这个庞然大物显然已经超出本教程的讨论范围，毕竟本文主要提供关键字和提示。如果你需要使用 Autotools，请认真研读 Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) 以及 `/usr/share/doc/autotools-dev/README.Debian.gz` 的本地副本。

[10] 软件包 `dh-autoreconf` 可以帮助你将这个过程自动化。参见第 4.4.3 节.

[11] 在 **aptitude** 工具中，软件包名字段的默认最大长度为 30。而 90% 以上的软件包包名都少于 24 个字符。

如果上游代码在它的名称中使用了一些通用术语比如 `test-suite`，那么你应当将其重命名，显式地说明其内容并避免污染命名空间。[12]

你应该让 **upstream version**（上游版本号）只包含字母和数字 (`0-9A-Za-z`), 以及加号 (+), 波浪号 (~), 还有点号 (`.`)。它必须以数字开头 (`0-9`)。[13] 如果可能的话，最好把它的长度控制在 8 字符以内。[14]

如果上游不使用像 `2.30.32` 这样的常规版本格式，而是用类似 `11Apr29` 这样的日期作为版本，类似于随机的代号字符串，或者以 VCS 的哈希值作为版本号的一部分，那么请确认将其从 **upstream version** 中移除。为此作出的改动信息可以记录在 `debian/changelog` 文件中。如果你需要发明一个版本字符串，请使用 `YYYYMMDD` 这个格式作为上游版本，比如 `20110429`。这会确保 **dpkg** 在升级软件包时能够正确解读新版本。如果需要确保未来能够平滑过渡到类似 `0.1` 这样的版本号的话，那就请使用 `0~YYMMDD` 格式作为上游版本，例如 `0~110429`。

版本字符串 [15] 可以用 dpkg(1) 来进行比较：

```
$ dpkg --compare-versions ver1 op ver2
```

版本比较规则可总结为以下几点：

- 字符串要从头到尾进行比较。
- 字母比数字大。
- 数字作为整数进行比较。
- 字母按照 ASCII 编码顺序进行比较。
- 对于点号 (`.`)，加号 (+)，以及波浪号 (~) 则有对应的特殊规则，具体如下：
  `0.0` < `0.5` < `0.10` < `0.99` < `1` < `1.0~rc1` < `1.0` < `1.0+b1` < `1.0+nmu1` < `1.1` < `2.0`

有一种比较棘手的情况，当上游释出 `gentoo-0.9.12-ReleaseCandidate-99.tar.gz` 作为 `gentoo-0.9.12.tar.gz` 的预发布版本时，就需要确保升级工作妥当进行：重命名该上游源代码为 `gentoo-0.9.12~rc99.tar.gz`.

2.7 设置 dh_make

首先设置两个环境变量，`$DEBEMAIL` 和 `$DEBFULLNAME`，这样大多数 Debian 维护工具就能够正确识别你用于维护软件包的姓名和电子邮件地址。[16]

```
$ cat >>~/.bashrc <<EOF
DEBEMAIL="your.email.address@example.org"
DEBFULLNAME="Firstname Lastname"
export DEBEMAIL DEBFULLNAME
EOF
$ . ~/.bashrc
```

[12] 如果你遵循 Debian Developer's Reference 5.1. "New packages" (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage)，那么在 ITP 过程中很容易遇到这样的问题。

[13] 这一条更严格的规则能帮助你避免混淆文件名。

[14] **aptitude** 的版本字段默认长度为 10。通常其中的 Debian 修订号和前置的连字符会消耗 2 个字符位置。对 80% 以上的软件包来说，上游版本小于 8 字符，Debian 修订号小于 2 字符。对 90% 以上的软件包来说，上游版本小于 10 字符，Debian 修订号小于 3 字符。

[15] 版本字符串可以是 **upstream version** (*`version`*), **Debian revision** (*`revision`*), 或者 **version** (*`version-revision`*)。关于 Debian 修订号如何增长，请参见第 8.1 节 **Debian revision**。

[16] 以下内容默认你以 Bash 作为登陆 shell。如果你使用其他的 shell，例如 Z shell，那就使用它们的配置文件代替这里提到的 `~/.bashrc`。

2.8 初始化外来 Debian 软件包

一般来说，由上游程序产生的 Debian 软件包都是外来的。若你想要用上游源代码 `gentoo-0.9.12.tar.gz` 创建一个外来 Debian 软件包，你可以在它的基础上进行外来软件包初始化，这只需要调用 **dh_make** 命令：

```
$ cd ~/gentoo
$ wget http://example.org/gentoo-0.9.12.tar.gz
$ tar -xvzf gentoo-0.9.12.tar.gz
$ cd gentoo-0.9.12
$ dh_make -f ../gentoo-0.9.12.tar.gz
```

当然，这里请用你原版源码归档的名字来替换 filename (文件名)。[17] 详情请参见 dh_make(8) 。

你会看到一些输出，询问你想要创建什么类型的软件包。这里的 Gentoo 被规划为一个单一二进制包——它仅仅产生一个二进制包，亦即单个 `.deb` 文件——于是我们就选择第一项 (按 s 键)，认真阅读屏幕上的输出信息，然后按 *ENTER* 键来确认。[18]

执行 **dh_make** 后，上级目录中自动创建了一份上游 tarball 的副本，名为 `gentoo_0.9.12.orig.tar.gz`，这个文件和稍后要介绍的 `debian.tar.gz` 合在一起便满足了一部分 Debian 非本土源码包的要求。

```
$ cd ~/gentoo ; ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
gentoo_0.9.12.orig.tar.gz
```

请注意在 `gentoo_0.9.12.orig.tar.gz` 这个文件名中有两个关键点：

- 软件包名称和版本中间以字符 `_`（下划线）来分隔。
- 后缀名 `.tar.gz` 前边插有 `.orig`。

你或许注意到了 `debian` 目录下已经有了许多模板。这些文件将在第 4 章和第 5 章中一一加以说明。另外，打包这件事情无法被完全自动化。因此你还得按照第 3 章中介绍的方法为 Debian 修改软件包。再接下来，按照第 6 章中叙述的合适的方法来构建 Debian 软件包，并根据第 7 章中的方法进行软件包测试，最后参考第 9 章这里的说明将其上传。所有的这些主要步骤本教程都会进行解释。

如果你在修改过程中不小心删除或玩坏了某些模板，你还可以使用 **dh_make** 加 `--addmissing` 参数来将其还原。(译注：也可以用 dpkg -L dh-make 来寻找您想要的模板)

更新一个已存在的软件包可能有点复杂，因为它可能使用了旧的打包技术。在学习基本功的阶段，建议只创建全新的软件包；稍后的第 8 章中会细致地讲解更新现存软件包。

请注意，源代码中不必包含任何在第 2.4 节或第 2.5 节中谈论到的编译系统。就算源码包仅仅是一组图像之类的也可以，这时候这些文件的安装可以用 `debhelper` 的配置文件来搞定，比如 `debian/install` (参见第 5.11 节)。

[17] 如果上游源代码已经提供了有内容的 `debian` 目录，那么带上参数 `--addmissing` 来执行 **dh_make** 命令。新的源码包格式 `3.0 (quilt)` 的鲁棒性 (Robust) 已经足够优秀，以不至于轻易损坏。另外，你可能需要修改上游提供的内容，以满足你的 Debian 软件包之需。

[18] 这里有这几种选择：`s` 代表单一二进制包，`i` 代表独立于体系结构的软件包，`m` 代表多个二进制包，`l` 代表共享库文件包，`k` 代表内核模块包，`n` 代表内核补丁包，`b` 代表 `cdbs` 软件包。本教程专注于使用 **dh** 命令 (来自 `debhelper` 软件包) 来创建单一二进制包，但也会涉及到创建独立于体系结构或多个二进制软件包相关的内容。软件包 `cdbs` 提供了另一套可以代替 **dh** 命令的基础打包脚本，不过这个家伙已经超出了我们的讨论范围。

Chapter 3

修改源代码

请注意这里没有足够篇幅来描述有关修改上游源码的 全部细节，这里只介绍基本步骤和常见问题。

3.1 设置 quilt

quilt 程序是 Debian 打包过程中采用的补丁管理工具。我们只需要在默认配置的基础上，加以少许修改即可。首先我们来创建一个别名 **dquilt**，以方便打包之需：添加以下几行内容到 `~/.bashrc` 文件中。其中第二行可以给 **dquilt** 命令提供与 **quilt** 命令相同的 shell 补全：

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F _quilt_completion -o filenames dquilt
```

现在我们来创建 `~/.quiltrc-dpkg` 文件：

```
d=. ; while [ ! -d $d/debian -a `readlink -e $d` != / ]; do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
    # if in Debian packaging tree with unset $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_ctx=35: ↩
        diff_cctx=33"
    if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

参见 quilt(1) 以及 `/usr/share/doc/quilt/quilt.pdf.gz` 来获取有关 **quilt** 命令用法的信息。

3.2 修复上游 Bug

假设你在上游的 `Makefile` 文件中找到了一个错误,其中的 `install: gentoo` 应该修正为 `install: gentoo-target` 。

```
install: gentoo
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

使用 **dquilt** 修复这个问题，并把补丁命名为 `fix-gentoo-target.patch`。[1]

[1] `debian/patches` 目录应当是运行 **dh_make** 时生成的。因为假设的是在更新一个已存在的软件包，所以在这个例子中我们新建它。

```
$ mkdir debian/patches
$ dquilt new fix-gentoo-target.patch
$ dquilt add Makefile
```

现在将 Makefile 修改为如下的样子：

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

使用 **dquilt** 将补丁创建到 debian/patches/fix-gentoo-targe.patch 并根据 DEP-3: Patch Tagging Guidelines (http://dep.debian.net/deps/dep3/) 添加描述：

```
$ dquilt refresh
$ dquilt header -e
... 描述补丁
```

3.3 文件安装

大多数第三方程序会默认安装在 /usr/local 目录下。在 Debian 中，这是保留给系统管理员的私有位置，因此 Debian 软件包不可以使用比如/usr/local/bin 这样的目录，而应当使用比如 /usr/bin 这样的系统目录，以遵循文件系统层级结构标准：Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-2.3.html) (FHS)。

通常情况下，make(1) 被用来自动编译程序，接着执行 make install 就可以把程序按照 Makefile 文件中的 install target 安装到指定的位置。为使 Debian 能够提供编译好的二进制软件包，打包脚本将自动修正编译系统，使其将文件安装到一个在临时目录中创建的文件系统子树中，而非直接安装到实际的目标位置。

普通程序安装过程和 Debian 打包安装过程二者的区别可以由 debhelper 软件包中的 **dh_auto_configure** 和 **dh_auto_install** 透明地处理。但必须满足以下条件：

- Makefile 文件应当遵循 GNU 的规定支持 $(DESTDIR) 变量[2]
- 源代码必须遵循文件系统层级标准 (FHS)。

使用 GNU **autoconf** 的程序默认遵守 GNU 的规定，这有利于打包过程的自动化。通过这项特点和其他启发式处理，估计 debhelper 软件可以直接制作约 90% 的软件包而不需维护者对编译系统做出大的改变。所以打包也不是看起来那样复杂。

如果你需要修改 Makefile 文件，就要确保其支持 $(DESTDIR) 变量，虽然默认情况下 $(DESTDIR) 变量没有设置并且在程序安装时会前置到每个文件的路径中。打包脚本会将 $(DESTDIR) 设置到临时目录上。

对于从源码生成单个二进制包的情况，**dh_auto_install** 将临时目录设置到 debian/*package*。[3] 临时目录中的全部文件都将成为软件包内容，并在安装该包时被安装到用户系统。这里唯一的区别是 **dpkg** 会把文件安装到真实的根目录树中，而不是你的工作目录。

请记住，即使你的程序正确安装到了debian/*package*，仍然要考虑将 .deb 软件包文件安装到根目录下的情形。所以绝对不允许构建系统将诸如 /home/me/deb/*package-version*/usr/share/*package* 这种诡异的内容硬编码到软件包文件中。

以下是 gentoo 软件包的 Makefile 文件中的相关部分[4]：

[2] 参见 GNU Coding Standards: 7.2.4 DESTDIR: Support for Staged Installs (http://www.gnu.org/prep/standards/html_node/DESTDIR.html#DESTDIR)。

[3] 对于单份源码生成多个二进制包的情况，**dh_auto_install** 将临时目录设置为debian/tmp，而 **dh_install** 命令则将文件按照 debian/*package-1*.install 和 debian/*package-2*.install 等文件的描述将 debian/tmp 中的文件分别装入 debian/*package-1* 和 debian/*package-2* 等临时目录，用以创建 *package-1*_*.deb 和 *package-2*_*.deb 等二进制软件包。

[4] 这只是一个演示 Makefile 正常形态的例子。如果 Makefile 是通过 **./configure** 命令生成的，那么修复该类 Makefile 的方法是通过 **dh_auto_configure** 来执行 **./configure**，并带上包括 --prefix=/usr 的默认选项。

```
# Where to put executable commands on 'make install'?
BIN     = /usr/local/bin
# Where to put icons on 'make install'?
ICONS   = /usr/local/share/gentoo
```

可以看到文件被放到了 /usr/local 下。按照上边的解释，该目录被 Debian 保留作本地用途，所以请把它们按如下方式修改：

```
# Where to put executable commands on 'make install'?
BIN     = $(DESTDIR)/usr/bin
# Where to put icons on 'make install'?
ICONS   = $(DESTDIR)/usr/share/gentoo
```

二进制文件、图标和文档等的更确切位置均已在文件层级标准 (FHS) 中作出了详尽描述。本教程建议你快速浏览相关章节以获取你打包需要用到的内容。

因此，我们应当把可执行二进制文件安装到 /usr/bin 而非 /usr/local/bin，而 man 手册页则应放在 /usr/share/man/man1 而非 /usr/local/man/man1，依此类推。注意，gentoo 的 Makefile 里没有提及手册页，而按照 Debian Policy 的要求，每个程序都应当有一个手册页，我们将在稍后制作一个并安装到 /usr/share/man/man1。

有些程序不使用 Makefile 变量定义路径，这意味着你可能需要去编辑 C 程序源代码来使他们使用正确的路径。但是到哪里去搜索，哪些才是呢？你可以通过以下的方法找到它们：

```
$ grep -nr --include='*.[c|h]' -e 'usr/local/lib' .
```

grep 会递归搜索整个源代码树并告诉你所有匹配项的文件名和行号。

编辑那些文件，在那些行中用 usr/lib 替换 usr/local/lib。这个过程可以用如下方法自动化完成：

```
$ sed -i -e 's#usr/local/lib#usr/lib#g' \
        $(find . -type f -name '*.[c|h]')
```

如果你想要确认每一个替换操作，那么下边的方法可以让你交互式地达成：

```
$ vim '+argdo %s#usr/local/lib#usr/lib#gce|update' +q \
        $(find . -type f -name '*.[c|h]')
```

紧接着你应该找到 install target (通常搜索以 install: 开头的行即可)，并把除 Makefile 顶部定义变量之外的目录引用妥当修改。

原始的 gentoo 的 install target 是这样：

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

让我们来修复这个上游 BUG，并把修改使用 **dquilt** 命令记录到debian/patches/install.patch。

```
$ dquilt new install.patch
$ dquilt add Makefile
```

使用你喜欢的编辑器按照以下内容为 Debian 软件包作修改：

```
install: gentoo-target
        install -d $(BIN) $(ICONS) $(DESTDIR)/etc
        install ./gentoo $(BIN)
        install -m644 icons/* $(ICONS)
        install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

你一定会注意到规则里在其他命令前有了一个 `install -d` 命令。原始的 `Makefile` 文件中没有它，因为通常情况下当你执行 `make install` 命令时，`/usr/local/bin` 和用到的其他目录早已存在于系统中。然而当我们要向新建的私有目录树中安装时，我们必须创建其中的每一个目录。

我们还可以在末尾添加上其他的内容，比如上游作者有时会省略的附加文档：

```
        install -d $(DESTDIR)/usr/share/doc/gentoo/html
        cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

仔细检查后如果没有问题，使用 **dquilt** 创建 `debian/patches/install.patch` 补丁文件并添加对它的描述：

```
$ dquilt refresh
$ dquilt header -e
... 描述补丁
```

现在你有了一系列的补丁。

1. 修复上游 Bug：`debian/patches/fix-gentoo-target.patch`
2. Debian 特有的打包时修改：`debian/patches/install.patch`

当进行任何非 Debian 特有的修改时，比如 `debian/patches/fix-gentoo-target.patch`，一定要向上游作者进行反馈，以便上游作者方便在下一版本中以使更多人受益。同时请记住不要做出特别针对 Debian 或 Linux (甚至是 Unix！) 的修改，要使其可以移植，这会使你的修改更容易被接受。

注意你不一定要把 `debian/*` 都提交到上游。

3.4 不同的库名称

还有另外一个常见的问题：不同平台之间的库名称常常因平台而异。例如一个 `Makefile` 中可能引了用一个 Debian 系统中不存在的库。这种情况下我们需要将其修改为 Debian 中存在的、提供完全相同功能的库。

如果你手中程序的 `Makefile`(或 `Makefile.in`) 中有如下的行。

```
LIBS = -lfoo -lbar
```

如果你的程序由于 `foo` 库不存在，而 Debian 系统中的 `foo2` 库提供了其等效，那么你可以修复这个构建问题并将修改记录到 `debian/patches/foo2.patch` 中，只需要将 `foo` 切换到 `foo2`:[5]

```
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e 's/-lfoo/-lfoo2/g' Makefile
$ quilt refresh
$ quilt header -e
... 描述补丁
```

[5] 如果从 `foo` 库切换到 `foo2` 库时要更改应用程序接口（API），这就要求我们修改源代码来符合新的 API。

Chapter 4

debian 目录中的必须内容

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。[1]

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” (http://www.debian.org/doc/debian-policy/ch-controlfields.html) 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。[2]

[1] 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。

[2] 参见 Debian Policy Manual, 2.4 ”Sections” (http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections) 以及 List of sections in sid (http://packages.debian.org/unstable/) .

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。[3]

- `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。[4] 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)，`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。[5]

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 (http://www.debian.org/doc/devel-manuals#policy) 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：[6]

[3] 参见 Debian Policy Manual, 2.5 ”Priorities” (http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities) 。

[4] 参见 Debian Policy Manual, 7.7 ”Relationships between source and binary packages - Build-Depends, Build-Depends-Indep, Build-Conflicts, Build-Conflicts-Indep” (http://www.debian.org/doc/debian-policy/ch-relationships.html#s-sourcebinarydeps) 。

[5] 这种奇怪的情况是 Debian Policy Manual, Footnotes 55 (http://www.debian.org/doc/debian-policy/footnotes.html#f55) 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 (https://bugs.launchpad.net/launchpad-buildd/+bug/238141) 。

[6] 确切信息请参见 Debian Policy Manual, 5.6.8 ”Architecture” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) 。

- `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：[7]

- `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 (http://lists.debian.org/debian-devel/) 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 (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) 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。

[7] 参见 Debian Policy Manual, 7 ”Declaring relationships between packages” (http://www.debian.org/doc/debian-policy/ch-relationships.html)。

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。[8]

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。[9] 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
```

[8] 这些描述都使用英语。相应的翻译由 The Debian Description Translation Project - DDTP (http://www.debian.org/intl/l10n/ddtp) （Debian 描述翻译项目）项目提供。

[9] 参见 Debian Developer's Reference, 6.2.5. "Version Control System location" (http://www.debian.org/doc/manuals/developers-reference/best-pkging-practices.html#bpp-vcs) 。

```
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" (http://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile) 掌控着它的内容，另外 DEP-5: Machine-parseable `debian/copyright` (http://dep.debian.net/deps/dep5/) 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 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：

```
 1 Format-Specification: http://svn.debian.org/wsvn/dep/web/deps/dep5.mdwn?op=file&rev=135
 2 Name: gentoo
 3 Maintainer: Josip Rodin <joy-mg@debian.org>
 4 Source: http://sourceforge.net/projects/gentoo/files/
 5
 6 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
 7 License: GPL-2+
 8
 9 Files: icons/*
10 Copyright: 1998 Johan Hanson <johan@tiq.com>
11 License: GPL-2+
12
13 Files: debian/*
14 Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
15 License: GPL-2+
16
17 License: GPL-2+
18  This program is free software; you can redistribute it and/or modify
19  it under the terms of the GNU General Public License as published by
20  the Free Software Foundation; either version 2 of the License, or
21  (at your option) any later version.
22  .
23  This program is distributed in the hope that it will be useful,
24  but WITHOUT ANY WARRANTY; without even the implied warranty of
25  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
26  GNU General Public License for more details.
27  .
28  You should have received a copy of the GNU General Public License along
29  with this program; if not, write to the Free Software Foundation, Inc.,
30  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
```

```
31  .
32  On Debian systems, the full text of the GNU General Public
33  License version 2 can be found in the file
34  ‘/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” (http://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog) 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：

```
1  gentoo (0.9.12-1) unstable; urgency=medium
2
3   * Initial release (Closes: #nnnn)  <nnnn is the bug number of your ITP>
4
5  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
6
```

(注：我为它添加了行号。)

Line 1 is the package name, version, distribution, and urgency. The name must match the source package name; distribution should be `unstable`, and urgency.

Lines 3-5 are a log entry, where you document changes made in this package revision (not the upstream changes —there is a special file for that purpose, created by the upstream authors, which you will later install as `/usr/share/doc/gentoo/changelog.gz`). Let’s assume your ITP (Intent To Package) bug report number was `12345`. New lines must be inserted just below the uppermost line that begins with `*` (asterisk). You can do it with dch(1). You can edit this manually with a text editor as long as you follow the formatting convention used by the dch(1).

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：

```
1  gentoo (0.9.12-1) UNRELEASED; urgency=low
2
3   * Initial Release. Closes: #12345
4   * This is my first Debian package.
5   * Adjusted the Makefile to fix $(DESTDIR) problems.
6
7  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
8
```

(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`)。[10]

你可以在关于更新的第 8 章中了解更多关于 `changelog` 的内容。

[10] 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。

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。[11] 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。[12]

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)[13]
- binary-arch target：在父目录中创建平台依赖 (Architecture: any) 的二进制包。(必须)[14]
- binary-indep target：在父目录中创建平台独立 (Architecture: all) 的二进制包。(必须)[15]
- 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 EXAMPLES in dpkg-buildflags(1) and read /usr/share/dpkg/*
 7 DPKG_EXPORT_BUILDFLAGS = 1
 8 include /usr/share/dpkg/default.mk
 9
10 # see FEATURE AREAS in dpkg-buildflags(1)
```

[11] 你可以通过该资源来学习编写 Makefile：Debian Reference, 12.2. ”Make” (http://www.debian.org/doc/manuals/debian-reference/ch12#_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)。

[12] Debian Policy Manual, 4.9 ”Main building script: debian/rules” (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules) 针对细节进行了解释。

[13] 此 target 被 dpkg-buildpackage 用于第 6.1 节描述的过程中。

[14] 此 target 被 dpkg-buildpackage -B 用于第 6.2 节描述的过程中。

[15] 此 target 被 dpkg-buildpackage -A 使用。

```
11 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
12
13 # see ENVIRONMENT in dpkg-buildflags(1)
14 # package maintainers to append CFLAGS
15 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
16 # package maintainers to append LDFLAGS
17 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
18
19 # main packaging script based on dh7 syntax
20 %:
21         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 20 和 21 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**。[16] **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。[17]

- `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[18]：

```
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
```

[16] 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 (http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf) 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。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。

[17] 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 --no-act` *`target`* 或 `debian/rules --'--no-act` *`target`*`'` 来执行以查看。

[18] 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。

```
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。[19] 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：[20]

- **dh_auto_install** 通常在 `Makefile` 存在且有 `distclean` target 时执行以下命令[21]

```
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_install** 通常在 `Makefile` 存在且有 `test` target 时执行以下命令。[22]

```
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)。

[19] 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。
[20] 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。
[21] 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。

[22] 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。

```
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：[23]

- 为 **dh_python2** 命令添加支持。(对于 Python 的最佳选择。)[24]
  - 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_pusuppor** 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。

[23] 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。

[24] 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。

  - 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*)*。[25]

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[26]，你可以创建一个 override_dh_auto_build target 来启用它。

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

[25] 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。
[26] 没有参数的 **dh_auto_build** 命令将执行 Makefile 中的第一个 target。

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。[27]

```
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] `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`。

Chapter 5

debian 目录下的其他文件

要控制 debhelper 在构建软件包过程中的大部分行为，可以在 debian 目录中放置可选的配置文件。本章将会对这些文件和它们的格式进行概述。请阅读 Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) 和 Debian Developer's Reference (http://www.debian.org/doc/devel-manuals#devref) 来了解更多关于打包方针的内容。

dh_make 命令会在 debian 目录中创建一些配置文件模板，它们的文件名多带有 .ex 后缀。其中的一些可能以二进制包名作为前缀，如 *package*。现在我们来对它们进行一个大致的了解。[1]

还有一些 debhelper 的配置文件模板可能未被 **dh_make** 命令创建。在这种情况下如果你需要它们，则要使用文本编辑器手工创建。

如果你希望或需要激活它们中的任意一个或多个，请按照下面的方法做：

- 如果模板文件带有 .ex 或 .EX 后缀，则重命名它去掉后缀；
- 把配置文件的名称改为实际的二进制软件包名，例如 *package*;
- 修改模板文件来满足你的需要;
- 删除不需要的模板文件;
- 如果需要，修改 control 文件 (参看第 4.1 节)。
- 如果需要，修改 rules 文件 (参看第 4.4 节)。

任何不带有 *package* 前缀的 debhelper 配置文件，比如应用到第一个二进制包的 install。当此处有多个二进制包时，可以通过在文件名中前置它们各自的名称来指定它们各自的配置文件，比如*package-1*.install, *package-2*.install, 之类。

5.1 README.Debian

所有关于你的 Debian 版本与原始版本间的额外信息或差别都应在这里记录。

dh_make 创建了一个默认的文件，以下是它的样子：

```
gentoo for Debian
-----------------
<possible notes regarding this package - if none, delete this file>
 -- Josip Rodin <joy-mg@debian.org>, Wed, 11 Nov 1998 21:02:14 +0100
```

如果你没有需要写在这里的东西，则删除这个文件。参看 dh_installdocs(1)

[1] 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。

5.2 compat

compat 文件定义了 debhelper 的兼容级别。目前你应当使用如下方法将其设置为 debhelper V10:

```
$ echo 10 > debian/compat
```

在特定场景下，你可以在需要兼容旧版本系统时使用兼容等级 9。然而，我们不建议你使用任何低于 9 的兼容等级，在新建软件包时也应避免使用这些低的等级。

5.3 conffiles

关于软件有件很恼人的事，就是当你付出了很多时间和精力来自定义一个程序，但是升级后所有的修改都被覆盖掉了。Debian 通过将配置文件单独标记来解决这个问题，[2] 当软件包升级的时候，你将会被询问是否要保留你的旧配置文件。

dh_installdeb(1) *automatically* 会把 /etc 目录下的任何文件都标记为 conffiles，所以如果你的程序在那只有 conffiles 的话就不需要再在这个文件中指定它们。对于大多数软件包类型，唯一合理的 conffiles 文件存放位置自始至终应当在 /etc 目录下，正因如此，该文件也没有存在的必要。

如果你的程序使用配置文件，但程序会自动对配置文件进行改写，则最好别将其标记为 conffiles ，因为 **dpkg** 总是会要求用户校验变更。

如果你正在打包的程序需要所有用户都为自己修改 /etc 目录中的配置文件，那么有两种常见的方法让 **dpkg** 不将其记录为 conffiles，以使其沉默。

- 在 /etc 目录中创建指向 /var 中维护者脚本 (maintainer scripts) 生成的文件的符号链接。
- 用维护者脚本 (maintainer scripts) 在 /etc 目录中生成一个文件。

更多关于维护者脚本 (maintainer scripts) 的信息，参看第 5.18 节

5.4 *package*.cron.*

如果你的软件包需要有计划运行的操作以保证正常工作，可以使用这个文件达成。你既可以设置常规的任务使其在每小时、每天、每星期、每月或其他情况或你希望的时间下运行。相应的文件名是:

- *package*.cron.hourly - 安装至 /etc/cron.hourly/*package* ； 每小时运行一次。
- *package*.cron.daily - 安装至 /etc/cron.daily/*package* ； 每天运行一次。
- *package*.cron.weekly - 安装至 /etc/cron.weekly/*package* ； 每周运行一次。
- *package*.cron.monthly - 安装至 /etc/cron.monthly/*package* : 每月运行一次。
- *package*.cron.d - 安装至 /etc/cron.d/*package*: 对于其他的任何时间。

这些文件均为 shell 脚本。唯一不同的是 *package*.cron.d，它的格式需要参照 crontab(5)

有必要显式地用 cron.* 文件来设置日志轮转；关于这个问题，请参见 dh_installlogrotate(1) 和 logrotate(8).

[2] 参见 dpkg(1) 以及 Debian Policy Manual, ”D.2.5 Conffiles” (http://www.debian.org/doc/debian-policy/ap-pkg-controlfields.html#s-pkg-f-Conffiles) .

5.5 dirs

这个文件指定了我们需要，但在正常安装过程 (dh_auto_install 触发的 make install DESTDIR=...) 里没有被安装的目录。通常这是由于 Makefile 中存在问题。

install 文件中列出的文件不需要为其提前创建目录，参看第 5.11 节。

最好是先尝试安装，如果遇到了问题再使用这个文件。在 dirs 中列出的目录名中不应有前导的 / (斜杠) 符号。

5.6 *package*.doc-base

如果你的软件包在 man 手册页和 info 信息文档外还有其他文档，你应该使用 doc-base 文件注册它们，这样用户可以使用例如 dhelp(1)、dwww(1) 或 doccentral(1) 的工具找到它们。

这通常包括 HTML、PS 和 PDF 文件，放置在 /usr/share/doc/*packagename*/。

以下是 gentoo 的 doc-base 文件 gentoo.doc-base 的样子：

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: File Management
Format: HTML
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

关于文件格式的更多信息，参看 install-docs(8) 以及 Debian doc-base 手册页，在 /usr/share/doc/doc-base/doc-base.html/index.html 有一份本地拷贝，这是由 doc-base 软件包提供的。

关于安装附加文档的更多信息，查看第 3.3 节。

5.7 docs

这个文件指定了我们使用 dh_installdocs(1) 安装到临时目录的文档文件名。

默认情况下它会加入代码目录顶层的所有名为 BUGS、README*、TODO 等的文件。

对于 gentoo，这里还加入了一些其他文件：

```
BUGS
CONFIG-CHANGES
CREDITS
NEWS
README
README.gtkrc
TODO
```

5.8 emacsen-*

如果你的软件包提供可以在安装时编译为字节码的 Emacs 文件，你可以使用这些文件设置。

它们会被 dh_installemacsen(1) 安装到临时目录。

如果你不需要这些，就删除它们。

5.9 *package*.examples

dh_installexamples(1) 会将列出的文件和目录作为示例文件安装。

5.10 *package*.init 和 *package*.default

如果你的软件包需要在系统启动时运行一个守护进程，那么你显然没有按照我最初的建议做事，不是吗？:-)

package.init 文件会被安装为 /etc/init.d/*package* 脚本，该脚本可用于启动和停止守护进程。**dh_make** 创建的 init.d.ex 是一个很好的骨架模板。你可能要对其改名并做很多修改，同时还要提供 Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb) (LSB，Linux 标准规范) 的兼容头文件。用 dh_installinit(1) 可以将它安装到临时目录中。

package.default 文件会被安装至 /etc/default/*package*。这个文件设置被 init 脚本使用的默认设置。*package*.default 文件最经常用于禁用一个正在运行的守护程序，或者设置一些默认标记或者超时。如果你的 init 脚本中有某种可配置的特性，你可以在 *package*.default 文件中设置它们，而不是 init 脚本本身。

如果你的上游程序中包含了给 init 用的脚本文件，那用不用它都可以。如果不使用，则创建相应的 *package*.init 文件；然而如果上游的 init 脚本很好且被安装到正确的位置，你仍然需要设置 rc* 符号链接。你需要按照以下的方法在 rules 文件中凌驾 **dh_installinit**：

```
override_dh_installinit:
        dh_installinit --onlyscripts
```

如果你不需要这些，就删除它们。

5.11 install

如果你的软件包需要那些标准的 make install 没有安装的文件，你可以把文件名和目标路径写入 install 文件，它们将被 dh_install(1) 安装。[3] 你需要首先检查要安装的文件是否有更有针对性的特定工具会对其进行安装。例如，文档应写在 docs 文件中安装，而不是这一个。

这个 install 文件每行安装一份文件，格式上先是相对于编译目录的文件路径，然后是一个空格，接下来是相对于安装目录的目标目录。例如，假设某个二进制文件 src/*bar* 没有被默认安装，则应让 install 呈现成这样：

```
src/bar usr/bin
```

这意味着安装这个软件包时，将有一个二进制文件 /usr/bin/*bar*.

当然，在相对路径保持不变的情况下 install 文件也可以只包含相对的源路径而不带目标位置。这样的格式通常用于使用 *package-1*.install、*package-2*.install 等将大软件包分割为多个二进制包的情况。

如果 **dh_install** 命令没有在当前目录 (或者你可能使用 --sourcedir 参数指定的位置) 找到文件，它会回滚至使用 debian/tmp 目录。

5.12 *package*.info

如果你的软件包有 info 信息页，应该使用 dh_installinfo(1) 安装，要安装的文件列于 *package*.info 文件中。

5.13 *package*.links

如果你作为包维护者需要在包中创建附加的符号链接，你应该使用 dh_link(1) 来安装，要安装的文件完整路径列于 *package*.info 文件中。

[3] 这取代了已经废弃的 dh_movefiles(1) 命令，它本是用 files 文件所配置的。

5.14 {*package*.,source/}lintian-overrides

如果 lintian 根据 Debian Policy 的某些规则允许例外从而报告了错误诊断,你可以使用 *package*.lintian-overrides 或 source/lintian-overrides 使其不再警告。请认真阅读 Lintian 用户手册 (https://lintian.debian.org/manual/index.html) 并避免滥用。

package.lintian-overrides 是对于名为 *package* 的二进制包的有效配置，会由 **dh_lintian** 命令安装到usr/share/lintian/overrides/*package*。

source/lintian-overrides 是针对源代码包的，不会安装。

5.15 manpage.*

你的程序应该有 man 手册页，如果它们没有自带则需要由你来创建。**dh_make** 命令创建了几个 man 手册页的模板。为每一个缺手册的命令拷贝一份模板，并妥善地编写，并且要删除未使用的模板。

5.15.1 manpage.1.ex

man 手册页通常是使用 nroff(1) 的格式编写的。manpage.1.ex 模板也是使用 **nroff** 格式的。参看 man(7) 手册页来简要了解如何编辑这个文件。

最终的 man 手册页文件的名称，应当为其对应的程序名称。所以我们将 manpage 重命名为 gentoo。同时，它的文件名当然要以 .1 作为第一个后缀，这表示本手册页是为一个用户命令而攒写的。再者，请校验本 man 手册处在正确的节 (section)。这个简短的列表列举了 man 手册页的章节：

Section(部分)	**Description(描述)**	**Notes(提示)**
1	User commands(用户命令)	Executable commands or scripts(可执行命令或脚本)
2	System calls(系统调用)	Functions provided by the kernel(由内核提供的功能)
3	Library calls(库调用)	Functions within system libraries(系统库中的功能)
4	Special files(特殊文件)	经常在 /dev 目录中出没
5	File formats(文件格式)	例如，/etc/passwd 的格式
6	Games(游戏)	Games or other frivolous programs(游戏或无聊的程序)
7	Macro packages(宏包)	比如 **man** 宏
8	System administrarion(系统管理)	Programs typically only run by root(典型的 root 专用程序)
9	Kernel routines(内核惯例)	Non-standard calls and internals(非标准调用及内部构建)

所以 gentoo 的 man 手册页应叫 gentoo.1。如果原始代码中没有 gentoo.1 man 手册页，你需要重命名 manpage.1.ex 模板为 gentoo.1 并按照示例和上游文档编辑它。

你也可以使用 **help2man** 命令来借助每个程序的 --help 和 --version 参数的输出来生成 man 手册页。[4]

5.15.2 manpage.sgml.ex

如果你希望使用 SGML 而非 **nroff** 格式编写 man 手册页，可以使用 manpage.sgml.ex 模板。如果你要这样，需要进行以下步骤：

- 重命名文件为类似 gentoo.sgml 的名称。
- 安装 docbook-to-man 软件包。
- 在 control 文件中将 docbook-to-man 添加到 Build-Depends 行。

[4] 注意，**help2man** 的占位符性质 man 手册页会声称在 info 系统中有着更为详尽的细节。如果命令缺少 **info** 页，那你应该手工编辑由 **help2man** 命令创建的 man 手册页。

- 在 rules 文件里添加 override_dh_auto_build target：

```
override_dh_auto_build:
        docbook-to-man debian/gentoo.sgml > debian/gentoo.1
        dh_auto_build
```

5.15.3 manpage.xml.ex

如果你希望使用 XML 而非 SGML，可以使用 manpage.xml.ex 模板。如果你要这样，需要进行以下步骤：

- 重命名源文件为类似 gentoo.1.xml 的名称。
- 安装 docbook-xsl 软件包和一个 XSLT 处理器，例如 xsltproc (推荐)。
- 在 control 文件中将 docbook-xsl, docbook-xml, 以及 xsltproc 软件包添加到 Build-Depends 行。
- 在 rules 文件里添加 override_dh_auto_build target：

```
override_dh_auto_build:
        xsltproc --nonet \
         --param make.year.ranges 1 \
         --param make.single.year.ranges 1 \
         --param man.charmap.use.subset 0 \
         -o debian/ \
http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.xsl\
        debian/gentoo.1.xml
        dh_auto_build
```

5.16 *package*.manpages

如果你的软件包有 man 手册页，你应该将它们列在 *package*.manpages 文件中以便 dh_installman(1) 进行安装。

要将 doc/gentoo.1 安装为 gentoo 的 man 手册页，创建一个 gentoo.manpages，内容如下：

```
docs/gentoo.1
```

5.17 NEWS

dh_installchangelogs(1) 命令会安装这个文件。

5.18 {pre,post}{inst,rm}

postinst、preinst、postrm 和 prerm 文件 [5] 被称为 *maintainer scripts*。它们是放置于软件包控制区域，并由 **dpkg** 在软件包安装、升级或卸载时执行的脚本。

作为一个新维护人员，你应当避免手工编辑 maintainer scripts ，因为它们常存在各种问题。更多信息请阅读 Debian Policy Manual, 6 ”Package maintainer scripts and installation procedure” (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) 并查看 **dh_make** 给出的示例文件。

如果你不听我的劝告，自己为一个软件包创建并定制了 maintainer scripts，你必须保证不仅测试 **install** 和 **upgrade**，还应测试 **remove** 和 **purge**。

[5] 尽管这里使用 **bash** 表达式速记法 {pre,post}{inst,rm} 来指示这些文件名，但你应该使用纯 POSIX 语法来编写 maintainer scripts，这是为了兼容 **dash** 作为系统 shell 的情况。

升级到新版本应当是静默且非侵入式的 (已有用户应当只在发现旧的 Bug 被修复或有新特性时注意到升级的变化)。

当更新必须以非静默模式进行时 (例如分散在多个主目录中的配置文件都要改为完全不同的结构时)，作为最后的手段，你应该考虑将软件包设置到安全的回退状态 (例如禁用服务) 并按照 Debian Policy 提供相应的文档 (`README.Debian` 和 `NEWS.Debian`)。不要在升级时使用 maintainer scripts 触发 **debconf** 来打扰用户。

`ucf` 软件包提供了 类似 *conffile* 对可能不标记为 *conffiles* 的文件的保留机制，比如由 maintainer scripts 来管理的配置文件。这可以把最大程度减少相关的问题。

这些 maintainer scripts 是 Debian 的增强特性，它们解释了 人们为什么选择 **Debian**。你必须非常小心，保证人们不因此产生烦恼。

5.19 *package*`.examples`

对于新维护者而言，打包一个库非常不易，因此不建议尝试。这样说吧，如果你的软件包有库，那你应该处理好 `debian/`*`package`*`.symbols` 文件。参见第 A.2 节.

5.20 `TODO`

dh_installdocs(1) 命令会安装这个文件。

5.21 `watch`

The `watch` file format is documented in the uscan(1) manpage. The `watch` file configures the **uscan** program (in the `devscripts` package) to watch the site where you originally got the source. This is also used by the Debian Package Tracker (https://tracker.debian.org/) service.

它的内容如下：

```
# watch control file for uscan
version=3
http://sf.net/gentoo/gentoo-(.+)\.tar\.gz debian uupdate
```

通常，在按照这个 `watch` 文件执行时，URL `http://sf.net/gentoo` 会被下载，程序会在其中搜索 `<a href=...>`。最后一个斜杠 `/` 后的基本名称会按照 Perl 正则表达式匹配 (参看 perlre(1)) `gentoo-(.+)\.tar\.gz`。在所有匹配的文件里，将会下载带有最大版本号的，此后由 **uupdate** 程序创建更新的源代码树。

尽管上述内容对于所有站点都适用，但 SourceForge 下载服务 (http://sf.net (http://sf.net)) 仍是一个例外。当 `watch` 中包含匹配 Perl 正则表达式 `^http://sf\.net/` 的 URL 时，**uscan** 程序会将其替换为 `http://qa.debian.org/watch/sf.php` 然后应用此规则。http://qa.debian.org/ (http://qa.debian.org/) 的 URL 重定向服务被设计用于提供一个稳定的重定向服务以满足 `watch` 文件中的 `http://sf.net/`*`project/tar-name`*`-(.+)\.tar\.gz` 形式，这样解决了 SourceForge 经常性的 URL 变更导致的问题。

如果上游提供 tarball 的加密签章，那么推荐校验其真实性，可以用 `pgpsigurlmangle` 选项来实现，这一点在 uscan(1) 中有描述。

5.22 `source/format`

在 `debian/source/format` 中只包含一行，写明了此源代码包的格式 (查看 dpkg-source(1) 获得完整列表)。在 `squeeze` 后，它应该是以下二者之一：

- `3.0 (native)` - Debian 本土软件或者

- `3.0 (quilt)` - 其他所有软件

全新的 `3.0 (quilt)` 源代码格式将所有修改使用 **quilt** 补丁系列记录到 `debian/patches`。这些修改会在解压源代码包时自动应用。[6] Debian 修改保存于 `debian.tar.gz` 归档文件，其中包含了整个 `debian` 目录。这个新格式支持直接添加例如 PNG 图标等的二进制文件。[7]

dpkg-source 解压 `3.0 (quilt)` 格式的源码包时会自动应用所有列于 `debian/patches/series` 的补丁。你可以使用 `--skip-patches` 选项避免在解压后自动应用补丁。

5.23 source/local-options

如果你希望使用版本控制系统（VCS）管理 Debian 打包活动，你可以创建一个分支 (例如叫做 `upstream`) 来跟踪上游代码，和另一个分支 (对于 Git 而言典型的是 `master` 分支) 来跟踪你的 Debian 软件包。对于后者，通常会将未应用补丁的上游代码和你的 `debian/*` 文件放在一起以便容易合并上游的新代码。

在编译软件包之后，源代码通常会被保持在应用补丁后的状态。你需要手工执行 `quilt pop -a` 来解除这些补丁，然后再提交到 `master` 分支。你可以向 `debian/source/local-options` 文件里添加一行 `unapply-patches` 来自动实现此目的。这个文件不会被加入到生成的源代码包，它只影响本地的编译构建行为。这个文件里还可以包含 `abort-on-upstream-changes` (参看 dpkg-source(1))。

```
unapply-patches
abort-on-upstream-changes
```

5.24 source/options

在源码树下自动生成的文件有时会超级讨厌，因为它们会生成毫无意义巨大无比的补丁文件。为了减轻这种问题，可以用一些定制模块比如 **dh_autoreconf**，参见第 4.4.3 节。

你可以给 `--extend-diff-ignore` 选项提供一个 Perl 正则表达式作为 dpkg-source(1) 的参数，以此忽略在创建源码包时自动生成的文件所发生的变更。

作为这个自动生成文件的问题的通用解决办法你可以存放像 **dpkg-source** 这样的选项参数到源码包的 `source/options` 文件中。如下将会跳过给 `config.sub`, `config.guess`, 以及 `Makefile` 创建补丁文件。

```
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

5.25 patches/*

旧的 `1.0` 源代码包格式使用单一的大 `diff.gz` 文件为源码保存 `debian` 中的维护文件和补丁。这样的软件包比较难于在事后检查和分析。这不是很好。

新的 `3.0 (quilt)` 源码格式将补丁存储在 `debian/patches/*` 中，用 **quilt** 命令。这些补丁和其他 `debian` 目录下的打包数据都会被打包成 `debian.tar.gz` 文件。由于 **dpkg-source** 命令可以处理 **quilt** 格式的补丁数据 (在格式为 `3.0 (quilt)` 的源码中)，而不需要 `quilt` 软件包；不需要在 `Build-Depends` 中添加 `quilt`。[8]

quilt 命令在 quilt(1) 中有详细描述。它将对源代码的修改维护于 `debian/patches` 中一系列 `-p1` 级别的补丁文件中，`debian` 目录外的文件没有任何修改。这些补丁的顺序记录于 `debian/patches/series` 文件中。你可以轻松地 apply (=push)、un-apply (=pop) 和 refresh 补丁。[9]

[6] 参看 DebSrc3.0 (http://wiki.debian.org/Projects/DebSrc3.0) 以了解转换到新的 `3.0 (quilt)` 和 `3.0 (native)` 源代码格式时的总结。
[7] 实际上新格式还支持多个上游 tarball 和更多的压缩方法，但这已超出本文档讲述的范围。
[8] 这里已经提出了若干种补丁集维护方法，并且正为 Debian 软件包所采用。**quilt** 系统是最推荐的维护系统。而其他的有包括 **dpatch**, **dbs**, 以及 **cdbs**。其中有许多将补丁保存为 `debian/patches/*` 文件。
[9] 如果你需要一个 sponsor 上传你的软件包，这种情况下，清晰的软件包分离和记录更改的文档对于软件包审查过程的顺利程度非常重要。

在第 3 章中，我们在 debian/patches 创建了三个补丁。

因为 Debian 补丁位于 debian/patches，请确定按照第 3.1 节中的方法正确配置 **dquilt**。

如果在之后有人 (包括你自己) 提供了一个补丁 *foo*.patch，对于 3.0 (quilt) 源代码包格式可以很容易修改：

```
$ dpkg-source -x gentoo_0.9.12.dsc
$ cd gentoo-0.9.12
$ dquilt import ../foo.patch
$ dquilt push
$ dquilt refresh
$ dquilt header -e
... describe patch
```

用新的 3.0 (quilt) 源代码包格式存储的补丁必须有 清晰的边界。你应该通过 dquilt pop -a; while quilt push; do quilt refresh; done 来验证这点。

Chapter 6

构建软件包

现在我们已经为构建软件包做好了准备。

6.1 完整的 (重) 构建

为保证完整的软件包 (重) 构建能顺利进行，你必须保证系统中已经安装

- `build-essential` 软件包；
- 列于 `Build-Depends` 域的软件包 (参看第 4.1 节)；
- 列于 `Build-Depends-indep` 域的软件包 (参看第 4.1 节)。

然后在源代码目录中执行以下命令：

```
$ dpkg-buildpackage -us -uc
```

这样会自动完成所有从源代码包构建二进制包的工作，包括：

- 清理源代码树 (`debian/rules clean`)
- 构建源代码包 (`dpkg-source -b`)
- 构建程序 (`debian/rules build`)
- 构建二进制包 (`fakeroot debian/rules binary`)
- 制作 `.dsc` 文件
- 用 **dpkg-genchanges** 命令制作 `.changes` 文件。

如果构建结果令人满意，那就用 **debsign** 命令以你的私有 GPG 密钥签署 `.dsc` 文件和 `.changes` 文件。你需要输入密码两次。[1]

对于非本土 Debian 软件包，比如 `gentoo`，构建软件包之后，你将会在上一级目录 (`~/gentoo`) 中看到下列文件：

[1] 为了连接至信任网络，本 GPG 密钥必须被一名 Debian 开发者签署，而且必须注册到 the Debian keyring (http://keyring.debian.org) (Debian 密钥环) 中。这样你上传的软件包就能被接受到 Debian 归档中了。参见 Creating a new GPG key (http://keyring.debian.org/creating-key.html) 以及 Debian Wiki on Keysigning (http://wiki.debian.org/Keysigning)。

- `gentoo_0.9.12.orig.tar.gz`

  这是原始的源代码 tarball，最初由 `dh_make -f ../gentoo-0.9.12.tar.gz` 命令创建，它的内容与上游 tarball 相同，仅被重命名以符合 Debian 的标准。

- `gentoo_0.9.12-1.dsc`

  这是一个从 `control` 文件生成的源代码概要，可被 dpkg-source(1) 程序解包。

- `gentoo_0.9.12-1.debian.tar.gz`

  这个压缩的 Tar 归档包含你的 `debian` 目录内容。其他所有对于源代码的修改都由 **quilt** 补丁存储于 `debian/patches` 中。

  如果其他人想要重新构建你的软件包，他们可以使用以上三个文件很容易地完成。只需复制三个文件，再运行 `dpkg-source -x gentoo_0.9.12-1.dsc`。[2]

- `gentoo_0.9.12-1_i386.deb`

  这是你的二进制包，可以使用 **dpkg** 程序安装或卸载它，就像其他软件包一样。

- `gentoo_0.9.12-1_i386.changes`

  这个文件描述了当前修订版本软件包中的全部变更，它被 Debian FTP 仓库维护程序用于安装二进制和源代码包。它是部分从 `changelog` 和 `.dsc` 文件生成的。

  随着你不断完善这个软件包，程序的行为会发生变化，也会有更多新特性添加进来。下载你软件包的人可以查看这个文件来快速找到有哪些变化，Debian 仓库维护程序还会把它的内容发表至 debian-devel-changes@lists.debian.org (http://lists.debian.org/debian-devel-changes/) 邮件列表。

在上传到 Debian FTP 仓库中前，`gentoo_0.9.12-1.dsc` 文件和 `gentoo_0.9.12-1_i386.changes` 文件必须用 **debsign** 命令签署，其中使用你自己存放在 `~/.gnupg/` 目录中的 GPG 私钥。用你的公钥，可以令 GPG 签名证明这些文件真的是你的。

debsign 命令可以用来以指定 ID 的 GPG 密钥进行签署（这方便了赞助 (sponsor) 软件包），只要照着下边在 `~/.devscripts` 中的内容：

```
DEBSIGN_KEYID=Your_GPG_keyID
```

`.dsc` 和 `.changes` 文件中很长的数字串是其中提及文件的 SHA1/SHA256 校验和。下载你软件包的人可以使用 sha1sum(1) 或 sha256sum(1) 来进行核对。如果校验和不符，则说明文件已被损坏或偷换。

6.2 自动编译系统

Debian 支持非常多的 移植 (port) (http://www.debian.org/ports/)，同时它有着 自动构建网络 (http://www.debian.org/devel/-buildd/)，这个网络在不同体系结构的计算机上运行着 **buildd** 守护进程。虽然你不需要自己做这件事情，你也应该知道在你的软件包身上发生了什么。让我们来简要地看看他们是如何为你给多个体系结构构建软件包的。[3]

对于 `Architecture: any` 的软件包，自动编译系统重构建它们。` 它确保系统中已经安装

- `build-essential` 软件包；
- 列于 `Build-Depends` 域的软件包 (参看第 4.1 节)。

然后在源代码目录中执行以下命令：

```
$ dpkg-buildpackage -B
```

这样会自动完成从源代码包构建平台依赖二进制包的工作，包括：

[2] 你可以使用 `--skip-patches` 选项来在正常的提取操作后避免应用 `3.0 (quilt)` 源代码包格式中的 **quilt** 补丁。你也可以在正常解压后使用 `quilt pop -a` 还原这些补丁对源码的修改。

[3] 实际中的自动构建系统包含了极为复杂 (比这里说明的还要复杂) 的体制。不过这类细节已经超出本文档范围。

- 清理源代码树 (`debian/rules clean`)
- 构建程序 (`debian/rules build`)
- 构建平台依赖二进制包 (`fakeroot debian/rules binary-arch`)
- 使用 **gpg** 签署 `.dsc` 文件
- 使用 **dpkg-genchanges** 和 **gpg** 创建并签署上传用的 `.changes` 文件

这就是你看到你的软件包在其他平台上可用的原因。

尽管通常的打包工作中 `Build-Depends-indep` 字段中列出的软件包都需要安装 (参看第 6.1 节)，但是在编译平台依赖二进制包时它们无需在自动编译系统上安装。[4]通常打包和自动编译系统的这种不同为你指出如何考虑必须的软件包应如何放在 `debian/control` 文件的 `Build-Depends` 或 `Build-Depends-indep` 域中 (参看第 4.1 节)。

6.3 debuild 命令

你可以使用 **debuild** 命令来进一步自动化 **dpkg-buildpackage** 的构建过程。参看 debuild(1)

debuild 命令会执行 **lintian** 命令，以在 Debian 软件包构建结束之后进行静态检查。**lintian** 命令可以用下边出现在 `~/.devscripts` 文件中的项来定制：

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
```

在普通用户帐号中可以使用以下这样简单的命令清理源代码并重构建软件包：

```
$ debuild
```

还可以这样简单地清理源代码树：

```
$ debuild clean
```

6.4 `pbuilder` 软件包

对于使用净室 (**chroot**) 编译环境来验证编译依赖而言，`pbuilder` 软件包是非常有用的。[5]它确保了软件包在不同构架上的 `sid` 发行版环境中的自动编译器中能干净地编译，避免了总是被归类于 RC (Release Critical，影响发布) 的严重 FTBFS (Fails To Build From Source，从源代码编译失败) Bug。[6]

我们通过以下操作来定制 `pbuilder` 软件包。

- 设置 `/var/cache/pbuilder/result` 对当前用户可写。
- 创建一个对用户可写的目录保存钩子脚本，例如 */var/cache/pbuilder/hooks*
- 在 `~/.pbuilderrc` 或 `/etc/pbuilderrc` 中添加以下内容：

```
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
```

首先使用以下命令初始化本地 `pbuilder` **chroot** 系统：

[4] 和在 `pbuilder` 中不同，自动编译系统使用的 `sbuild` 软件包所维护的 **chroot** 不强制要求最小化的编译系统，并可能保持很多软件包始终安装在其中。

[5] 由于 `pbuilder` 软件包仍然在进化，你应当查阅最新的官方文档来检查实际的配置状况。

[6] 参见 http://buildd.debian.org/ 以获取更多关于 Debian 软件包 auto-building 的信息。

```
$ sudo pbuilder create
```

如果你已经创建了源代码包，在包含 *foo*.orig.tar.gz、*foo*.debian.tar.gz 和 *foo*.dsc 文件的目录中执行下面的命令来更新 pbuilder **chroot** 系统以便在其中执行构建：

```
$ sudo pbuilder --update
$ sudo pbuilder --build foo_version.dsc
```

新构建的无 GPG 签名的软件包会被以非 root 属主放置于 /var/cache/pbuilder/result/。

.dsc 文件和 .changes 文件中的 GPG 签名可以用如下方法生成：

```
$ cd /var/cache/pbuilder/result/
$ debsign foo_version_arch.changes
```

如果你已经更新了源代码树但没有生成对应的源代码包，在存放 debian 目录的源码目录里执行：

```
$ sudo pbuilder --update
$ pdebuild
```

你可以使用 pbuilder --login --save-after-login 命令登录到这个 **chroot** 环境中并按照需要对其进行配置。通过 ^D (Control-D) 离开这个 shell 时环境会被保存。

最新版的 **lintian** 命令可以通过设置钩子脚本 */var/cache/pbuilder/hooks/*B90lintian 在 chroot 环境中运行。脚本内容如下：[7]

```
#!/bin/sh
set -e
install_packages() {
        apt-get -y --allow-downgrades install "$@"
        }
install_packages lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes" - pbuilder
# use this version if you don't want lintian to fail the build
#su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" - pbuilder
echo "+++ end of lintian output +++"
```

为 sid 编译软件包需要使用 sid 环境。在现实中 sid 存在很多问题以至于你不愿意将整个系统都迁移到其上。pbuilder 软件包可以在这种情况下很好地解决问题。

你可能需要在 stable 软件包发布后为 stable-proposed-updates、stable/updates 等仓库升级你维护的软件包。[8]对于这类情况，“我正在运行 sid 系统”并不是你不为它们进行升级的充分理由。pbuilder 软件包可以帮助你使用到相同 CPU 体系结构下几乎所有 Debian 和 Debian 衍生版系统环境。

参见 http://www.netfort.gr.jp/~dancer/software/pbuilder.html, pdebuild(1), pbuilderrc(5), 和 pbuilder(8).

6.5 git-buildpackage 及其相似命令

如果你的上游对源代码使用版本控制系统 (VCS)[9]，你也应该考虑使用它们。这会使得合并和提取上游补丁更加简单。有多个为不同 VCS 设计的包装脚本包来协助 Debian 软件包构建。

- git-buildpackage：帮助维护 Git 仓库中 Debian 软件包的套件。

[7] 此处默认 HOOKDIR=/var/cache/pbuilder/hooks。你可以在 /usr/share/doc/pbuilder/examples 目录中找到很多钩子脚本的例子。

[8] 升级你的 stable 软件包有一些规定的限制。

[9] 参见 Version control systems (http://www.debian.org/doc/manuals/debian-reference/ch10#_version_control_systems) 以获取更多信息。

- `svn-buildpackage`：帮助维护 Subversion 仓库中 Debian 软件包的程序。
- `cvs-buildpackage`：为 CVS 源代码树设计的 Debian 软件包脚本集。

`git-buildpackage` 的使用在 Debian 开发者之中非常流行，它可以使用 alioth.debian.org (http://alioth.debian.org/) 上的 Git 服务器来管理 Debian 软件包。[10] 这个软件包提供了许多命令来 自动化打包操作：

- git-import-dsc(1): 将一个早先的 Debian 软件包导入到 Git 仓库中。
- git-import-orig(1): 将上游 tar 文件导入到 Git 仓库中。
- git-dch(1): 用 Git commit 信息来生成 Debian changelog。
- git-buildpackage(1): 从 Git 仓库中构建 Debian 软件包。
- git-pbuilder(1): 用 **pbuilder**/**cowbuilder** 从 Git 仓库来构建 Debian 软件包。

这些命令使用 3 个分支来跟踪打包操作：

- `main` 用于 Debian 软件源码树。
- `upstream` 用于上游源码树。
- 由 `--pristine-tar` 为上游 tarball 生成的 `pristine-tar`。[11]

你可以设置 `git-buildpackage`，通过修改 `~/.gbp.conf` 文件。参见 gbp.conf(5). [12]

6.6 快速重构建

对于很大的软件包，在调试 `debian/rules` 的过程中你可能不想每次都对整个软件包进行重构建。仅用于测试目的，你可以不重新构建源代码包而使用以下的方法创建 `.deb` 文件[13]：

```
$ fakeroot debian/rules binary
```

或者可以通过以下方法验证它是否能通过编译：

```
$ fakeroot debian/rules build
```

一旦完成了调试，记住要按照前面所给出的正常过程重构建你的软件包。你可能无法正常上传用此种方法构建的 `.deb` 文件。

[10] Debian wiki Alioth (http://wiki.debian.org/Alioth) 这里说明了如何使用 alioth.debian.org (http://alioth.debian.org/) 服务。

[11] `--pristine-tar` 选项会调用 **pristine-tar** 命令，它能利用一个很小的二进制差量文件以及在版本控制系统中某个分支内保存的上游文件内容（分支名通常为 `upstream`）重新生成与上游 tarball 完全相同的一份副本。

[12] 这里有一些给高级读者参考的网页资源。

- 用 git-buildpackage (`/usr/share/doc/git-buildpackage/manual-html/gbp.html`) 构建 Debian 软件包
- debian packages in git (https://honk.sigxcpu.org/piki/development/debian_packages_in_git/)
- Using Git for Debian Packaging (http://www.eyrie.org/~eagle/notes/debian/git.html)
- git-dpm: Debian packages in Git manager (http://git-dpm.alioth.debian.org/)
- Using TopGit to generate quilt series for Debian packaging (http://git.debian.org/?p=collab-maint/topgit.git;a=blob_plain;f=debian/HOWTO-tg2quilt;hb=HEAD)

[13] 常规情形下被配置好的环境变量在此时不会被自动设置。永远不要将使用这个 快速方法构建的软件包上传到任何地方。

6.7 命令层级

这里有一个简练的总结，关于在命令层次结构中有多少用于构建软件包的命令能够组合到一起。做这件事情的方法非常多。

- `debian/rules` = 软件包构建过程的专用 maintainer script
- **dpkg-buildpackage** = 软件包构建之核心工具
- **debuild** = **dpkg-buildpackage** + **lintian** (在干净的环境变量下构建)
- **pbuilder** = Debian chroot 环境核心工具
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (在 chroot 环境中构建)
- **cowbuilder** = 加速 **pbuilder** 执行
- **git-pbuilder** = 命令行语法友好 **pdebuild** (被 **gbp buildpackage** 使用)
- **gbp** = 在 Git 仓库中管理 Debian 源码
- **gbp buildpackage** = **pbuilder** + **dpkg-buildpackage** + **gbp**

即便像 **gbp buildpackage** 和 **pbuilder** 这样的高级命令的应用可以确保完美的软件包构建环境，了解像 `debian/rules` 和 **dpkg-buildpackage** 这样的低级命令行工具如何被它们执行也是至关重要的。

Chapter 7

检查软件包中的错误

在上传软件包到公共仓库前，你还需要知道一些检查软件包错误的技巧。

不仅在自己的机器上测试总是一个好主意。你必须谨慎地对待以下叙述的测试中显示的任何一个警告或错误信息。

7.1 诡异可疑的改动

如果你在构建以 `3.0 (quilt)` 格式的非本土 Debian 软件包后，发现一个新的自动生成的补丁，比如 `debian-changes-*` 在 `debian/patches` 目录中有可能是你不小心更改了一些文件，或者构建脚本修改了上游源代码。如果这是你犯下的小错误，那就修复它。如果这是构建脚本干的好事，那就用 **dh-autoreconf** 来解决其根本问题，可参照第 4.4.3 节或者可以变通一下，处理 `source/options` 文件，参照第 5.24 节.

7.2 校验软件包安装过程

你必须测试你的软件包看是否存在安装问题。debi(1) 命令可以帮助你测试所有生成的二进制软件包。

```
$ sudo debi gentoo_0.9.12-1_i386.changes
```

你必须使用从 Debian 仓库下载的 `Contents-`*`i386`* 文件校验是否在与不同包存在文件冲突，以阻止在不同的系统上发生安装故障。**apt-file** 命令正适合完成这个任务。如果存在冲突，请通过重命名、将公共文件分离到另一个受其他包依赖的包中、与受影响的软件包的维护者合作使用 alternatives 机制来避免实际问题 (参看 update-alternatives(1)) 或在 `debian/control` 文件中设置 `Conflicts` 条目以声明冲突关系等方式避免问题的发生。

7.3 检验软件包的 maintainer scripts

所有 maintainer scripts，包括 `preinst`、`prerm`、`postinst` 和 `postrm` 文件，都非常难以编写，除非是由 `debhelper` 程序自动生成的。如果你是新维护人员则不要使用它们 (参看第 5.18 节)。

如果软件包使用了这些需要严格测试的 maintainer scripts，请确保不仅测试 install，还要测试 remove、purge 和 upgrade。很多 maintainer scripts 的 Bug 都显现于卸载或彻底删除软件包时。使用 **dpkg** 命令按以下方法来测试它们：

```
$ sudo dpkg -r gentoo
$ sudo dpkg -P gentoo
$ sudo dpkg -i gentoo_version-revision_i386.deb
```

整个测试过程应按照以下操作序列完成：

- 如果可能，安装前一个版本的软件包；
- 从前一个版本升级软件包；
- 降级软件包到前一个版本 (可选)；
- 彻底删除该软件包；
- 全新安装该软件包；
- 卸载该软件包；
- 再次安装该软件包。
- 彻底删除该软件包；

如果这是你的第一个软件包，你应该使用其他版本号创建一个测试用的软件包来完成升级测试，这样可以避免将来的问题。

请牢记如果你的软件包已经在以往版本的 Debian 中发布，人们通常会从最近发布的 Debian 发布里的版本升级，所以也要测试从那个版本升级到当前的版本。

尽管降级没有被正式支持，支持它也总是友好的。

7.4 使用 `lintian`

使用 lintian(1) 检查你的 `.changes` 文件。**lintian** 命令会运行很多测试脚本来检查常见的打包错误。[1]

```
$ lintian -i -I --show-overrides gentoo_0.9.12-1_i386.changes
```

当然，要替换你自己软件包所生成的 `.changes` 文件的文件名。**lintian** 命令的输出常带有以下几种标记：

- `E`：代表错误：确定违反了 Debian Policy 或是一个肯定的打包错误。
- `W`：代表警告：可能违反了 Debian Policy 或是一个可能的打包错误。
- `I`：代表信息：对于特定打包类别的信息。
- `N`：代表注释：帮助你调试的详细信息。
- `O`：代表已覆盖：一个被 `lintian-overrides` 文件覆盖的信息，但由于使用 `--show-overrides` 选项而显示。

对于警告，你应该改进软件包或者检查警告是否的确无意义。如果确定没有意义，则按照第 5.14 节中的叙述使用 `lintian-overrides` 文件将其覆盖。

注意，你可以用 **dpkg-buildpackage** 来构建软件包，并执行 **lintian**，只要你使用了 debuild(1) 或 pdebuild(1).

7.5 debc 命令

你可以使用 debc(1) 命令列出一个二进制 Debian 软件包中的文件。

```
$ debc package.changes
```

[1] 如果你按照第 6.3 节中的叙述定义了 `/etc/devscripts.conf` 或 `~/.devscripts` 文件，就不需要再添加 **lintian** 选项 `-i -I --show-overrides`。

7.6 debdiff 命令

你可以使用 debdiff(1) 命令比较两个 Debian 源代码包的内容。

```
$ debdiff old-package.dsc new-package.dsc
```

你还可以使用 debdiff(1) 命令比较两个 Debian 二进制包的文件列表。

```
$ debdiff old-package.changes new-package.changes
```

这个命令对于检查源代码包中哪些文件被修改了非常有用，还可以发现二进制包中是否有文件在更新过程中发生的变动，比如被意外替换或删除。

7.7 interdiff 命令

你可以使用 interdiff(1) 命令比较两个 `diff.gz` 文件。这对于更新使用旧的 `1.0` 源代码格式的软件包时，检查是否有意外的变更非常有用。

```
$ interdiff -z old-package.diff.gz new-package.diff.gz
```

新的 `3.0` 源码格式会将更改保存在多个补丁文件中，如第 5.25 节所述。你也可以使用 **interdiff** 跟踪每一个 `debian/patches/*` 文件中的改动。

7.8 mc 命令

很多文件检查操作可以通过使用类似 mc(1) 的文件管理器来完成，它可以帮助你直接查看 `*.deb` 文件的内容，除此之外还可以用于 `*.udeb`、`*.debian.tar.gz`、`*.diff.gz` 和 `*.orig.tar.gz` 文件。

还要检查在二进制包和源代码包中是否有不需要的文件或者空文件。这些文件经常没有被正确清理，如果存在这种情况，要调整 `rules` 文件进行处理。

Chapter 8

更新软件包

一旦你发布了一个软件包，在之后的某个时间里就需要对它进行更新。

8.1 新的 Debian 版本

假设你收到一个针对你的软件包报告的 Bug，其编号为 #654321，它描述了一个你可以解决的问题。要创建软件包的一个新 Debian 修订版本，你需要：

- 如果要将它记录于新的补丁中，这样做：
  - `dquilt new` *`bugname.patch`* 设置补丁名称；
  - `dquilt add` *`buggy-file`* 声明文件将被更改；
  - 修正软件包代码中的上游 Bug；
  - `dquilt refresh` 将修改记录到 *`bugname.patch`*；
  - `dquilt header -e` 添加对它的描述；
- 如果是更新一个已存在的补丁，这样做：
  - `dquilt pop` *`foo.patch`* 重现已存在的 *`foo.patch`*；
  - 修正旧的 *`foo.patch`* 中的问题；
  - `dquilt refresh` 更新 *`foo.patch`*；
  - `dquilt header -e` 更新对它的描述；
  - `while dquilt push; do dquilt refresh; done` 应用所有补丁以确保它们 边界清晰；
- 在 Debian `changelog` 文件的顶部添加一个条目。例如可以使用 `dch -i` 或用 `dch -v` *`version-revision`* 来指定版本，然后用你喜欢的编辑器插入信息。[1]
- 在 changlog 条目中简要描述 Bug 和相应的解决办法，并在后面添加 `Closes: #654321`。这样 Bug 报告会在你的软件包被 Debian 仓库接受的同时被仓库管理软件 自动关闭。
- 重复上述操作来修复更多的 Bug，并在需要的时候使用 `dch` 更新 Debian `changelog` 文件。
- 重复你在第 6.1 节和第 7 章中所做的事情。
- 一旦你满意了，那就将 `changelog` 中的发行版值由 `UNRELEASED` 修改成目标发行版值 `unstable` (或者是 `experimental`)。[2]

[1] 要获得需要的日期格式，使用 `LANG=C date -R`。

[2] 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。

- 按照第 9 章来上传软件包。惟一的区别是这次不需要再包含原始代码文件，因为它们没有变化且已经存在于 Debian 仓库中。

有一种棘手的情况，当你在上传正常版本到官方仓库中之前，你制作了一个本地包以进行打包实验，例如, *1.0.1-1*。为了平滑升级，创建一个 changelog 条目，其中包含类似 *1.0.1-1~rc1* 这样的版本字符串不失为一剂良方。你可以通过合并这样的本地修改条目到官方包的单个条目中来整理 changelog。参见第 2.6 节来了解版本字符串的排序。

8.2 检查新上游版本

在为 Debian 仓库准备新上游版本的软件包前，你必须首先对新的上游发布版本进行检查。

检查工作应从阅读上游 changelog、NEWS 以及所有随新版本一同发布的文档。

然后应按照以下步骤检查新旧版本之间源码的差别，小心任何可疑的内容：

```
$ diff -urN foo-oldversion foo-newversion
```

对于 Autotools 自动生成的文件发生的改动，例如 missing、aclocal.m4、config.guess、config.h.in、config.sub、configure、depcomp、install-sh、ltmain.sh 和 Makefile.in 是可以忽略的。你可以在运行 **diff** 进行代码检查前删除它们。

8.3 新上游版本

如果软件包 *foo* 是使用新的 3.0 (native) 或 3.0 (quilt) 格式打包的，制作新的上游版本时需要先把旧的 debian 目录移至新的源代码内。这可以通过在新解压的源代码目录里运行 tar xvzf */path/to/foo_oldversion*.debian 完成。[3]当然，你需要做几个很显然的杂事：

- 创建一份上游源代码的副本，命名为 foo_*newversion*.orig.tar.gz
- 使用 dch -v *newversion-1* 更新 Debian changelog 文件。
  - 添加一个条目，内容为 New upstream release。
  - 简明地介绍 在新上游版本中上游修复和关闭的 Bug (添加 Closes: *#bug_number*) 。
  - 简明地介绍维护者对 本个新上游版本做出的修改，修复和关闭的 Bug (添加 Closes: *#bug_number*)。
- 运行 while dquilt push; do dquilt refresh; done 以应用全部补丁并使它们 边界清晰。

如果补丁没有干净地被应用，检查原因 (线索在 .rej 文件里)。

- 如果你的补丁已经被上游接受，
  - 使用 dquilt delete 删除它。
- 如果你的补丁与上游代码中的变更有冲突：
  - 使用 dquilt push -f 应用旧补丁，未应用的部分会被保存为 *baz*.rej。
  - 手工编辑 *baz* 文件来在新的代码中实现 *baz*.rej 中应有的效果。
  - 使用 dquilt refresh 更新补丁。
- 正常继续，执行 while dquilt push; do dquilt refresh; done。

[3] 如果软件包 *foo* 是使用旧的 1.0 格式的，可以在新解压的源代码目录里运行 zcat */path/to/foo_oldversion*.diff.gz|patch -p1 来完成。

这个过程可以通过使用 uupdate(1) 来更自动化地完成：

```
$ apt-get source foo
...
dpkg-source: info: extracting foo in foo-oldversion
dpkg-source: info: unpacking foo_oldversion.orig.tar.gz
dpkg-source: info: applying foo_oldversion-1.debian.tar.gz
$ ls -F
foo-oldversion/
foo_oldversion-1.debian.tar.gz
foo_oldversion-1.dsc
foo_oldversion.orig.tar.gz
$ wget http://example.org/foo/foo-newversion.tar.gz
$ cd foo-oldversion
$ uupdate -v newversion ../foo-newversion.tar.gz
$ cd ../foo-newversion
$ while dquilt push; do dquilt refresh; done
$ dch
... document changes made
```

如果你按照第 5.21 节的叙述设置了 `debian/watch` 文件，你可以跳过这个 **wget** 命令，转而在 *`foo-oldversion`* 目录中运行 uscan(1)，且无需再执行 **uupdate** 命令。它会 自动查找新的源代码、下载并运行 **uupdate** 命令。[4]

重复第 6.1 节、第 7 章和第 9 章中的操作，即可发布此更新的软件包。

8.4 更新打包风格

更新打包风格不是更新软件包的必须步骤，但是这样可以使你的软件包得到对现代的 `debhelper` 系统和 `3.0` 源代码包格式完整的兼容性。[5]

- 如果你需要重新添加已删除的模板文件，可以在同一个 debian 软件包源代码树中运行 **dh_make**，并添加 `--addmissing` 选项。然后对模板进行相应的编辑。
- 如果软件包的 `debian/rules` 文件没有更新为使用 `debhelper` v7+ 的 **dh** 语法，则更新它使用 **dh**。在需要的时候更新 `debian/control` 文件。
- 如果你希望将使用 `cdbs` 的 `Makefile` 包含机制创建的 `rules` 文件更新为 **dh** 语法，参看下文并理解各 `DEB_*` 配置变量。
  - `/usr/share/doc/cdbs/cdbs-doc.pdf.gz` 的本地副本
  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)
- 如果你有一个不带有 *`foo`*`.diff.gz` 文件的 `1.0` 格式的源代码包，你可以通过创建 `debian/source/format` 文件并在其中添加 `3.0 (native)` 来将其更新为新的 `3.0 (native)` 源代码包格式。`debian` 目录中的其他文件可以直接复制过来。
- 如果你有一个带有 *`foo`*`.diff.gz` 文件的 `1.0` 格式的源代码包，你可以通过创建 `debian/source/format` 文件并在其中添加 `3.0 (quilt)` 来将其更新为新的 `3.0 (quilt)` 源代码包格式。`debian` 目录中的其他文件可以直接复制过来。如果需要，把 `filterdiff -z -x '*/debian/*' foo.diff.gz > big.diff` 生成的 `big.diff` 文件导入到 **quilt** 系统。[6]
- 如果它使用了其他的补丁系统，例如 `dpatch`、`dbs` 或 `cdbs`，使用 `-p0`、`-p1` 或 `-p2` 级别，使用 http://bugs.debian.org/-581186 (http://bugs.debian.org/581186) 的 `deb3` 命令将其转换到 `quilt` 系统。

[4] 如果 **uscan** 命令下载并更新了源代码，但没有运行 **uupdate** 命令，你应该修正 `debian/watch` 文件，使 URL 末尾后带有 `debian uupdate`。
[5] 如果你的 sponsor 或其他维护者一定反对更新已有的打包风格，则不值得去为此烦恼或争论，总是有更重要的事要做。
[6] 你可能使用 **splitdiff** 命令将 `big.diff` 分割为多个增量补丁。

- 如果它使用 **dh** 命令的 `--with quilt` 选项，或 **dh_quilt_patch** 和 **dh_quilt_unpatch** 命令，删除它们并使其使用新的 `3.0 (native)` 源代码包格式。

你应当查看 DEP - Debian Enhancement Proposals (http://dep.debian.net/) 并采纳 ACCEPTED 建议。

当然你还需要按照第 8.3 节完成其他的步骤。

8.5 UTF-8 转换

如果上游文档采用了老式编码，那么将其转换为 UTF-8 不失为一良方。

- 用 iconv(1) 来转换普通文本文件的编码。

```
iconv -f latin1 -t utf8 foo_in.txt > foo_out.txt
```

- 使用 w3m(1) 来把 HTML 文件转换为 UTF-8 普通文本文件。当你这样做的时候，请确认在 UTF-8 locale 下执行。

```
LC_ALL=en_US.UTF-8 w3m -o display_charset=UTF-8 \
        -cols 70 -dump -no-graph -T text/html \
        < foo_in.html > foo_out.txt
```

8.6 对更新软件包的几点提示

以下是对更新软件包的几点提示：

- 保留旧的 `changelog` 条目 (看似显然，但是总有可能把 `dch -i` 输入为 `dch`)。
- 已存在的 debian 修改需要被重新校验，去除上游已经合并的东西 (一种或多种形式)，除非有必要的原因，还要保留尚未被上游接受的部分。
- 如果对编译系统作出了修改 (希望你已经在检查上游变更时了解了这些)，那么要在必要时更新 `debian/rules` 和 `debian/control` 编译依赖关系。
- 检查 Debian Bug Tracking System (BTS) (http://www.debian.org/Bugs/) 是否有人为某些仍然未修复的 bug 提供了补丁。
- 检查 `.changes` 文件以确保你正要上传到正确的发行版、正确的列出 BUG 关闭 `Closes` 字段、`Maintainer` 和 `Changed-By` 字段相匹配，以及文件是否已经使用 GPG 签署等。

Chapter 9

上传软件包

现在你完成了对软件包的彻底测试，接下来将其释出到公共归档中分享它吧。

9.1 上传到 Debian 仓库

当你成为正式的开发人员[1]，你可以把软件包上传到 Debian 仓库[2]。你可以手工进行这项工作，但使用例如 dupload(1) 或 dput(1) 的自动化工具可以帮你更好地完成这项操作。在此我们将叙述如何使用 **dupload** 操作。[3]

首先需要设置 **dupload** 的配置文件。你既可以编辑系统级的 `/etc/dupload.conf` 文件，也可以使用自己的 `~/.dupload.conf` 文件凌驾一些需要修改的设置。

你可以阅读 dupload.conf(5) man 手册页来了解各选项的含义。

`$default_host` 选项决定了默认使用哪个上传队列，`anonymous-ftp-master` 是最基本的一个，但你很可能希望改用其他的。[4]

连接到互联网后，可以使用以下命令上传你的软件包：

```
$ dupload gentoo_0.9.12-1_i386.changes
```

dupload 会检查文件的 SHA1/SHA256 校验和是否与 `.changes` 文件中的相匹配，如果不匹配它会做出警告。你应按照如第 6.1 节所述来重构建软件包使得它可以被正常上传。

如果你在 ftp://ftp.upload.debian.org/pub/UploadQueue/ 遇到了上传问题，你可以通过 **ftp** 来手动上传 GPG 签署的 `*.commands` 文件。[5] 比如说，使用 `hello.commands` 命令：

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Uploader: Foo Bar <Foo.Bar@example.org>
Commands:
 rm hello_1.0-1_i386.deb
 mv hello_1.0-1.dsx hello_1.0-1.dsc
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

[...]
-----END PGP SIGNATURE-----
```

[1] 参见第 1.1 节。

[2] 有许多公开的档案比如 http://mentors.debian.net/ ，它们的运作方式几乎与 Debian 档案一致，并提供了一个非开发者的上传区域。你可以自己建立一个等效档案，只要使用 http://wiki.debian.org/HowToSetupADebianRepository 里边列举出来的工具。所以这一小节也对非开发者特别有用。

[3] `dput` 软件包提供了更多的特性，相比于 `dupload` 也越来越受欢迎。它使用 `/etc/dput` 文件作为全局配置文件、`~/.dput.cf` 作为用户配置文件。它也直接支持 ubuntu 相关的服务。

[4] 参见 Debian Developer's Reference 5.6. "Uploading a package" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#upload) .

[5] 参见 ftp://ftp.upload.debian.org/pub/UploadQueue/README。或者是，你可以使用 **dcut** 命令，它来自 `dput` 软件包。

9.2 在上传时包含 orig.tar.gz 文件

第一次向仓库上传软件包时要包含 orig.tar.gz 源代码归档。如果这个软件包的修订号既不是 1 也不是 0, 那你就必须给 **dpkg-buildpackage** 加上选项 -sa。

对于 **dpkg-buildpackage** 命令：

```
$ dpkg-buildpackage -sa
```

对于 **debuild** 命令：

```
$ debuild -sa
```

对于 **pdebuild** 命令：

```
$ pdebuild --debbuildopts -sa
```

另一方面，请注意 -sd 选项会强制排除原始的 orig.tar.gz 源代码。

9.3 跳过的上传

如果你在 debian/changelog 创建了多个条目并跳过了上传，你必须创建一个相应的 *_.changes 文件，其中包含自上次上传以来的全部变更记录。这可以通过指定 **dpkg-buildpackage** 的 -v 并将版本传递给它来完成。比如，*1.2*.

对于 **dpkg-buildpackage** 命令：

```
$ dpkg-buildpackage -v1.2
```

对于 **debuild** 命令：

```
$ debuild -v1.2
```

对于 **pdebuild** 命令：

```
$ pdebuild --debbuildopts "-v1.2"
```

Appendix A

高级打包

这里有一些关于你可能遇到的高级打包问题的提示。如果有需要的话，本教程强烈建议阅读这里引用和建议的文档。

你可能需要手工编辑由 **dh_make** 命令生成的打包模板文件，以此来解决本章中所讨论的问题。新的 **debmake** 命令应该能更好地解决这些问题。

A.1 共享库

在打包 共享库 之前，你应该阅读以下的主要参考资料：

- Debian Policy Manual, 8 ”Shared libraries” (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html)
- Debian Policy Manual, 9.1.1 ”File System Structure” (http://www.debian.org/doc/debian-policy/ch-opersys.html#s-fhs)
- Debian Policy Manual, 10.2 ”Libraries” (http://www.debian.org/doc/debian-policy/ch-files.html#s-libraries)

以下是帮助你开始的极简解释：

- 共享库均为 elf 对象文件，其包含编译好的机器码。
- 共享库以 `*.so` 文件的形式发放。(既非 `*.a` 文件也非 `*.la` 文件)
- 共享库主要用于在不同的二进制可执行程序之间共享代码，这背后使用了 **ld**（译注：链接）机制。
- 共享库有时会为一个可执行程序提供多个插件，这背后使用了 **dlopen** 机制。
- 共享库能导出代表着变量，函数和类的 symbols（符号）；并允许链接到它的可执行文件访问之。
- 共享库 `libfoo.so.1` 中的 SONAME：`objdump -p libfoo.so.1 | grep SONAME` [1]
- 共享库的 SONAME 常常与库文件自身文件名一致 (不过有特例)。
- 链接到 */usr/bin/foo* 的共享库的 SONAME：`objdump -p /usr/bin/foo | grep NEEDED` [2]
- `libfoo1`: 共享库 `libfoo.so.1` 的库文件包，其 SONAME ABI 版本为 *1*.[3]
- 在某些情况下，库软件包的 maintainer scripts 必须调用 **ldconfig** 来为 SONAME 创建必要的符号链接。[4]
- `libfoo1-dbg`: 包含了调试共享库包用的调试符号的软件包 `libfoo1`.

[1] 或者这样：`readelf -d libfoo.so.1 | grep SONAME`
[2] 或者这样：`readelf -d libfoo.so.1 | grep NEEDED`
[3] 参见 Debian Policy Manual, 8.1 ”Run-time shared libraries” (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-runtime) .
[4] 参见 Debian Policy Manual, 8.1.1 ”ldconfig” (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-ldconfig) .

- lib*foo*-dev: 包含了头文件等内容的开发包。用于 lib*foo*.so.*1*.[5]
- 一般而言，Debian 软件包不应当包含 *.la Libtool 归档文件。[6]
- 一般来说，Debian 软件包不应当使用 RPATH。[7]
- 虽然这有点过时，而且是第二参考，Debian Library Packaging Guide (http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html) 可能仍然对你有用。

A.2 管理 debian/*package*.symbols

当你给共享库打包时，你应当创建 debian/*package*.symbols 文件来管理在共享库名称不变，在同一个 SONAME 下又要提供 ABI 向后兼容性的情况下每个符号关联到的最小版本号。[8] 你可以阅读下边的主要参考以获知细节：

- Debian Policy Manual, 8.6.3 "The symbols system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-symbols) [9]
- dh_makeshlibs(1)
- dpkg-gensymbols(1)
- dpkg-shlibdeps(1)
- deb-symbols(5)

这是个粗略的例子，用来来演示创建 libfoo1 软件包的方法，此时使用上游 1.3 版本，有着妥当的debian/libfoo1.symbols 文件：

- 使用上游提供的 libfoo-1.3.tar.gz 文件来准备 Debian 化的源码骨架。
  - 如果这是库软件包 libfoo1 的第一次打包，那么以空内容创建 debian/libfoo1.symbols 文件。
  - 如果之前的上游版本 1.2 已经被 libfoo1 软件包打包了，并且其源码包中有妥当的 debian/libfoo1.symbols，再用它一次。
  - 如果前一个上游 1.2 版本打包时没有 debian/libfoo1.symbols，那就从具有相同库 SONAME 的同一个共享库包的所有可用的二进制软件包中创建它并命名为 symbols 文件。比如 1.1-1 和 1.2-1。[10]

```
$ dpkg-deb -x libfoo1_1.1-1.deb libfoo1_1.1-1
$ dpkg-deb -x libfoo1_1.2-1.deb libfoo1_1.2-1
$ : > symbols
$ dpkg-gensymbols -v1.1 -plibfoo1 -Plibfoo1_1.1-1 -Osymbols
$ dpkg-gensymbols -v1.2 -plibfoo1 -Plibfoo1_1.2-1 -Osymbols
```

- 尝试用像 **debuild** 和 **pdebuild** 这样的工具来对源码树进行试构建。(如果这因为缺失符号之类原因而失败，那么这里就有一些不向后兼容的 ABI 改变，这就需要你转移 (bump) 共享库的名称到诸如 libfoo1a，并重新开始一次。)

[5] 参见 Debian Policy Manual, 8.3 "Static libraries" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-static) and Debian Policy Manual, 8.4 "Development files" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-dev) .

[6] 参见 Debian wiki ReleaseGoals/LAFileRemoval (http://wiki.debian.org/ReleaseGoals/LAFileRemoval) .

[7] 参见 Debian wiki RpathIssue (http://wiki.debian.org/RpathIssue) .

[8] 向后不兼容的 ABI 变更常常需要你更新共享库的 SONAME，并把共享库名称换成新的。

[9] 对于 C++ 库和其他追踪单个符号过于困难的情况下，请遵循 Debian Policy Manual, 8.6.4 "The shlibs system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) 。

[10] 所有先前的 Debian 软件包版本都能在 http://snapshot.debian.org/ (http://snapshot.debian.org/) 找到。不过 Debian 修订号被去掉了，以使软件包的 backport 更为容易：1.1 << 1.1-1~bpo70+1 << 1.1-1 and 1.2 << 1.2-1~bpo70+1 << 1.2-1

```
$ cd libfoo-1.3
$ debuild
...
dpkg-gensymbols: warning: some new symbols appeared in the symbols file: ...
 see diff output below
--- debian/libfoo1.symbols (libfoo1_1.3-1_amd64)
+++ dpkg-gensymbolsFE5gzx        2012-11-11 02:24:53.609667389 +0900
@@ -127,6 +127,7 @@
  foo_get_name@Base 1.1
  foo_get_longname@Base 1.2
  foo_get_type@Base 1.1
+ foo_get_longtype@Base 1.3-1
  foo_get_symbol@Base 1.1
  foo_get_rank@Base 1.1
  foo_new@Base 1.1
...
```

- 如果你如上述看见由 **dpkg-gensymbols** 命令打印出来的差异，那就从生成的二进制库包中抽取妥当更新的 `symbols` 文件。[11]

```
$ cd ..
$ dpkg-deb -R  libfoo1_1.3_amd64.deb libfoo1-tmp
$ sed -e 's/1\.3-1/1\.3/' libfoo1-tmp/DEBIAN/symbols \
        >libfoo-1.3/debian/libfoo1.symbols
```

- 使用像 **debuild** 和 **pdebuild** 这样的工具来构建发行软件包。

```
$ cd libfoo-1.3
$ debuild clean
$ debuild
...
```

对上边这个例子补充一点，我们需要进一步检查 ABI (应用程序二进制接口) 兼容性并在需要的时候手动更新一些符号的版本。[12]

虽然这只是第二参考，Debian wiki UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) 和它指向的页面可能会有所帮助。

A.3 多体系结构

Debian wheezy 引入的多体系结构特性，集成了对二进制包跨体系结构安装的支持 (尤其是 `i386<->amd64`，其他的组合也有) 于 `dpkg` 和 `apt` 中。你可以阅读下边的参考：

- Ubuntu wiki MultiarchSpec (https://wiki.ubuntu.com/MultiarchSpec) (上游)
- Debian wiki Multiarch/Implementation (http://wiki.debian.org/Multiarch/Implementation) (Debian 的局势)

它为每个共享库的安装路径使用了类似 `i386-linux-gnu` 和 `x86_64-linux-gnu` 这样的三元名字。实际上每个二进制软件包构建的三元路径是被动态设置到 `$(DEB_HOST_MULTIARCH)` 变量中的，经由 dpkg-architecture(1) 命令。举个例子，安装多体系结构库文件的路径被按照下表进行了修改：[13]

[11] Debian 修订号已被从版本中去掉，这能让软件包的 backport 更为容易：`1.3` << `1.3-1~bpo70+1` << `1.3-1`

[12] 参见 Debian Policy Manual, 8.6.2 "Shared library ABI changes" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-updates) .

[13] 老旧且具有特殊用途的库路径，比如 `/lib32/` 和 `/lib64/` 已不再使用。

旧路径	**i386** 多体系结构路径	**amd64** 多体系结构路径
/lib/	/lib/i386-linux-gnu/	/lib/x86_64-linux-gnu/
/usr/lib/	/usr/lib/i386-linux-gnu/	/usr/lib/x86_64-linux-gnu/

下面是一些典型的多体系结构软件包分离情景：

- 库源码 lib*foo*-1.tar.gz
- 一个用编译型语言编写的工具的源码 *bar*-1.tar.gz
- 一个用解释型语言编写的工具的源码 *bar*-1.tar.gz

软件包	体系结构：	多体系结构：	软件包内容
lib*foo1*	任何	相同	共享库，可共同安装
lib*foo1*-dbg	任何	相同	共享库调试符号，可共同安装
lib*foo*-dev	任何	相同	共享库头文件之类，可共同安装
lib*foo*-tools	任何	外来	运行时支持程序，不可共同安装
lib*foo*-doc	全部	外来	共享库文档
bar	任何	外来	编译好的程序文件，不可共同安装
bar-doc	全部	外来	程序的配套文档文件
baz	全部	外来	解释型程序文件

请注意，开发软件包应该包含一个指向共享库的符号链接并且 不带有版本号。比如：/usr/lib/x86_64-linux-gnu/libfoo.so -> libfoo.so.1

A.4 构建共享库包

你可以用 dh(1) 通过以下方法构建一个支持多体系结构的 Debian 库软件包：

- 更新 debian/control。
  - 为源码包部分添加 Build-Depends: debhelper (>=10) 部分。
  - 为每个二进制库软件包添加 Pre-Depends: ${misc:Pre-Depends}。
  - 在每个二进制包的段中添加 Multi-Arch: 小节。
- 设置 debian/compat 为”10”。
- 将所有打包脚本中的路径从普通的 /usr/lib/ 调整到多体系结构的 /usr/lib/$(DEB_HOST_MULTIARCH)/。
  - 首先，在 debian/rules 中调用 DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTI 以设置 DEB_HOST_MULTIARCH 变量。
  - 在 debian/rules 中用 /usr/lib/$(DEB_HOST_MULTIARCH)/ 替换 /usr/lib/。
  - 如果 debian/rules 文件中的 override_dh_auto_configure 目标使用了 ./configure 文件，那么请确认用 dh_auto_configure -- 来替换它。[14]
  - 在debian/*foo*.install 文件中将所有 /usr/lib/ 的事件替换为 /usr/lib/*/。
  - 如需从 debian/*foo*.links.in 动态地生成像 debian/*foo*.links 这样的文件，可以添加一个脚本到 debian/rules 文件的 override_dh_auto_configure 目标中。

[14] 作为替代，你可以添加 --libdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH) 和 --libexecdir=\$${prefix}/lib/$(DEB_HOST_MULTI 参数到 ./configure 后头。请注意 --libexecdir 指定了安装可执行程序 (它们被其他程序使用，而更少是用户) 的默认路径。它的 Autotools 默认设置为 /usr/libexec/ 但是 Debian 的设置为 /usr/lib/。

```
override_dh_auto_configure:
        dh_auto_configure
        sed 's/@DEB_HOST_MULTIARCH@/$(DEB_HOST_MULTIARCH)/g' \
                debian/foo.links.in > debian/foo.links
```

请确认该共享库软件包仅仅包含预期中的文件，并且你的 -dev 软件包还奏效。

所有作为多体系结构软件包而同时安装到同一个文件路径的所有文件应当具有完全一致的文件内容。你必须小心由数据字节序和压缩算法造成的区别。

A.5 Debian 本土软件包

如果一个软件包是仅仅为 Debian 维护的，或者是可能的本地使用，那么它的源码可以容纳所有的 `debian/*` 于其中。这里有它的两种打包方式。

你可以将除 `debian/*` 文件之外的部分制作成上游 tarball，然后将其作为非本土 Debian 软件包来打包，正如第 2.1 节所述。这是一些人鼓励使用的普通方法。

另一种方法就是本土 Debian 软件包的打包工作流。

- 用包含所有文件的，单一压缩过的 tar 文件，以 `3.0 (native)` 格式来创建本土 Debian 源码包。
  - *package_version*`.tar.gz`
  - *package_version*`.dsc`
- 用 Debian 本土源码包构建二进制包。
  - *package_version_arch*`.deb`

比如说，如果你的源代码文件都存放在 `~/mypackage-1.0` 中，而且没有 `debian/*` 这些文件，那么你可以用它创建一个本土 Debian 软件包，只要按照下边的方法使用 **dh_make** 命令：

```
$ cd ~/mypackage-1.0
$ dh_make --native
```

接下来 `debian` 目录和它的内容都会被创建，正如第 2.8 节中那样。这不会创建一个 tarball，因为这是个本土 Debian 软件包。不过这也是唯一的区别。剩下的打包操作就是完全一致的了。

在执行了 **dpkg-buildpackage** 命令后，你将会在上一级目录中看到这些文件：

- `mypackage_1.0.tar.gz`
  这是 **dpkg-source** 命令用 `mypackage-1.0` 目录创建出来的源代码 tarball 。(它的文件名后缀不是 `orig.tar.gz`)
- `mypackage_1.0.dsc`
  这是对源码内容的简述，正如在非本土 Debian 软件包中那样。(没有 Debian 修订号。)
- `mypackage_1.0_i386.deb`
  这是完成的二进制包，正如在非本土 Debian 软件包中那样。(没有 Debian 修订号。)
- `mypackage_1.0_i386.changes`
  这个文件描述了这个软件作为外来 Debian 包，在当前版本所作出的所有更改。(没有 Debian 修订)