debian

Debian 维护者指南

青木修

August 9, 2018

Debian 维护者指南
by 青木修

本指南在撰写过程中参考了以下几篇文档:

- “Making a Debian Package (AKA the Debmake Manual)”, 版权所有 © 1997 Jaldhar Vyas.
- “The New-Maintainer’s Debian Packaging Howto”, 版权所有 © 1997 Will Lowe.
- “Debian New Maintainers’ Guide”, 版权所有 © 1998-2002 Josip Rodin, 2005-2017 Osamu Aoki, 2010 Craig Small 以及 2010 Raphaël Hertzog。

本指南的最新版本应当可以在下列位置找到:

- 在 debmake-doc 软件包 中，以及
- 位于 Debian 文档网站。

Contents

Abstract

本篇《Debian 维护者指南》（2018-08-09）教程文档面向普通 Debian 用户和未来的开发者，描述了使用 **debmake** 命令构建 Debian 软件包的方法。

本指南注重描述现代的打包风格，同时提供了许多简单的示例。

- POSIX shell 脚本打包
- Python3 脚本打包
- C 和 Makefile/Autotools/CMake
- 含有共享库的多个二进制软件包的打包，等等。

本篇“Debian 维护者指南”可看作“Debian 新维护者手册”的继承文档。

前言

如果您在某些方面算得上是有经验的 Debian 用户 [1] 的话，您可能遇上过这样的情况：

- 您想要安装某一个软件包，但是该软件在 Debian 仓库中尚不存在。
- 您想要将一个 Debian 软件包更新为上游的新版本。
- 您想要添加某些补丁来修复某个 Debian 软件包中的缺陷。

如果您想要创建一个 Debian 软件包来满足您的需求，并将您的工作与社区分享，您便是本篇指南的目标读者，即未来的 Debian 维护者。[2] 欢迎来到 Debian 社区。

Debian 是一个大型的、历史悠久的志愿者组织。因此，它具有许多需要遵守的社会上和技术上的规则和惯例。Debian 也开发出了一长串的打包工具和仓库维护工具，用来构建一套能够解决各种技术目标的二进制软件包：

- packages build across many architectures (第 5.4.4 节)
- reproducible build (第 5.4.5 节)
- clean build under clearly specified package dependencies and patches (第 5.5 节, 第 5.8 节, 第 7.10 节)
- optimal splits into multiple binary packages (第 5.5.1 节)
- smooth library transitions (第 5.18.2 节)
- interactive installation customization (第 5.19 节)
- multiarch support (第 5.20 节)
- security enhancement using specific compiler flags (第 5.21 节)
- continuous integration (第 5.22 节)
- boot strapping (第 5.23 节))
- ……

这些目标也许会让很多新近参与进 Debian 工作中的潜在 Debian 维护者感到迷茫而不知所措。本篇指南尝试为这些目标提供一个着手点，方便读者开展工作。它具体描述了以下内容：

- 作为未来潜在的维护者，您在参与 Debian 工作之前应该了解的东西。
- 制作一个简单的 Debian 软件包大概流程如何。
- 制作 Debian 软件包时有哪些规则。
- 制作 Debian 软件包的小窍门。
- 在某些典型场景下制作 Debian 软件包的示例。

[1] 您的确需要对 Unix 编程有所了解，但显然没必要是这方面的天才。在 Debian 参考手册 中，您可以了解到使用 Debian 系统的一些基本方法和关于 Unix 编程的一些指引。

[2] 如果您对分享 Debian 软件包不感兴趣，您当然可以在本地环境中将上游的源码包进行编译并安装至 **/usr/local** 来解决问题。

作者在更新原有的使用 **dh-make** 软件包的“新维护者手册”时感受到了文档的局限性。因此，作者决定创建一个替代工具并编写其对应的文档以解决某些现代的需求。其成果便是 **debmake**（当前版本：4.3.1)）软件包，以及这篇更新的“Debian 维护者指南”，可从 **debmake-doc**（当前版本：1.11-1）软件包获取。

许多杂项事务和小提示都集成进了 **debmake** 命令，以使本指南内容简单易懂。本指南同时提供了许多打包示例。

小心

合适地创建并维护 Debian 软件包需要占用许多时间。Debian 维护者在接受这项挑战时一定要确保 既能精通技术又能勤勉投入精力。

某些重要的主题会详细进行说明。其中某些可能看起来和您没什么关系。请保持耐心。某些边角案例会被跳过。某些主题仅使用外部链接提及。这些都是有意的行文安排，目标是让这份指南保持简单而可维护。

Chapter 1

概览

对 *package-1.0*.**tar.gz**，一个包含了简单的、符合 GNU 编码标准 和 FHS（文件系统层级规范） 的 C 语言源代码的程序来说，它在 Debian 下打包工作可以按照下列流程，使用 **debmake** 命令进行。

```
$ tar -xvzf package-1.0.tar.gz
$ cd package-1.0
$ debmake
  ... Make manual adjustments of generated configuration files
$ debuild
```

如果跳过了对生成的配置文件的手工调整流程，则最终生成的二进制软件包将缺少有意义的软件包描述信息，但是仍然能为 **dpkg** 命令所使用，在本地部署环境下正常工作。

小心

这里的 **debmake** 命令只提供一些不错的模板文件。如果生成的软件包需要发布出去供公众使用的话，这些模板文件必须手工调整至最佳状态以遵从 Debian 仓库的严格质量标准。

如果您在 Debian 打包方面还是个新手的话，此时不要过多在意细节问题，请先确立一个大致流程的印象。

如果您曾经接触过 Debian 打包工作，您会注意到这和 **dh_make** 命令很像。这是因为 **debmake** 命令设计时便旨在替代历史上由 **dh_make** 命令所提供的功能。[1]

debmake 命令设计提供如下特性与功能：

- 现代的打包风格
  - **debian/copyright**：符合 **DEP-5**
  - **debian/control**：**substvar** 支持、**multiarch** 支持、多个二进制软件包、……
  - **debian/rules**：**dh** 语法、编译器加固选项、……
- 灵活性
  - 许多选项（第 5.5.1.1 节、第 6 章、附录 A）
- 合理的默认行为
  - 执行过程不中断，输出干净的结果
  - 生成多架构支持（multiarch）的软件包，除非明确指定了 **-m** 选项。
  - 生成非本土 Debian 软件包，使用“**3.0 (quilt)**”格式，除非明确指定了 **-n** 选项。
- 额外的功能
  - 根据当前源代码对 **debian/copyright** 文件进行验证（第 6.4 节）

[1] The **deb-make** command was popular before the **dh_make** command. The current **debmake** package starts its version from **4.0** to avoid version overlaps with the obsolete **debmake** package, which provided the **deb-make** command.

debmake 命令将大多数重量级工作分派给了其后端软件包:**debhelper**、**dpkg-dev**、**devscripts**、**pbuilder**，等等。

提示

请确保将 **-b**、**-f**、**-I** 和 **-w** 选项的参数使用引号合适地保护起来，以避免 shell 环境的干扰。

提示

非本土软件包是标准的 Debian 软件包。

提示

本文档中所有软件包构建示例的详细日志可以由第 8.14 节一段给出的操作来获取。

注意

所产生的 **debian/copyright** 文件，以及 **-c**（第 6.3 节）和 **-k**（第 6.4 节）选项的输出都涉及了对版权和授权信息的启发式操作。它们具有局限性，可能会输出某些错误的结果。

Chapter 2

预备知识

这里给出您在投入 Debian 相关工作之前应当理解掌握的一些必备的预备知识。

2.1 Debian 社区的工作者

在 Debian 社区中有这几类常见的角色：

- 上游作者（**upstream author**）：程序的原始作者。
- 上游维护者（**upstream maintainer**）：目前在上游维护程序代码的人。
- 软件包维护者（**maintainer**）：制作并维护该程序 Debian 软件包的人。
- 赞助者（**sponsor**）：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）。
- 导师（**mentor**）：帮助新手维护者熟悉和深入打包的人。
- **Debian** 开发者（DD, Debian Developer）：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** 维护者（Debian Maintainer, DM）：拥有对 Debian 官方仓库部分上传权限的人。

注意，您不可能在一夜之间成为 **Debian** 开发者（DD），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，如果您的软件包对其他人有用，您可以当这个软件的软件包维护者，然后通过一位赞助者来上传这份软件，或者您可以申请成为 **Debian** 维护者。

还有，要成为 Debian 开发者不一定要创建新软件包。对已有软件做出贡献也是成为 Debian 开发者的理想途径。眼下正有很多软件包等着好的维护者来接手（参见第 2.8 节）。

2.2 如何做出贡献

请参考下列文档来了解应当如何为 Debian 作出贡献：

- 您如何协助 Debian?（官方）
- The Debian GNU/Linux FAQ, 第 12 章 - “为 Debian 项目捐赠”（半官方）
- Debian Wiki, HelpDebian（补充内容）
- Debian 新成员站点（官方）
- Debian Mentors FAQ（补充内容）

2.3 Debian 的社会驱动力

为做好准备和 Debian 进行交互，请理解 Debian 的社会动力学：

- 我们都是志愿者。
  - 任何人都不能把事情强加给他人。
  - 您应该主动地做自己想做的事情。
- 友好的合作是我们前行的动力。
  - 您的贡献不应致使他人增加负担。
  - 只有当别人欣赏和感激您的贡献时，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您。
  - 您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 期望您制作出高质量的软件包。
  - 您应该随时调整自己来适应变化。

在这篇指南之后的部分中，我们只关注打包的技术方面。因此，请参考下面的文档来理解 Debian 的社会动力学：

- Debian: 17 年的自由软件、“实干主义”、和民主（前任 DPL 制作的介绍性幻灯片）

2.4 技术提醒

这里给出一些技术上的建议，参考行事可以让您与其他维护者共同维护软件包时变得更加轻松有效，从而让 Debian 项目的输出成果最大化。

- 让您的软件包容易除错（debug）。
  - 保持您的软件包简单易懂。
  - 不要对软件包过度设计。
- 让您的软件包拥有良好的文档记录。
  - 使用可读的代码风格。
  - 在代码中写注释。
  - 格式化代码使其风格一致。
  - 维护软件包的 git 仓库 [1]。

> 注意
>
> 
>
> 对软件进行除错（debug）通常会比编写初始可用的软件花费更多的时间。

[1] 绝大多数 Debian 维护者使用 **git** 而非其它版本控制系统，如 **hg**、**bzr** 等等。

2.5 Debian 文档

请在阅读本指南的同时按需阅览下面这些 Debian 官方文档中的相关部分；这些文档提供的信息有助于创建质量优良的 Debian 软件包：

- “Debian 政策手册”
  - “必须遵循”的规则（https://www.debian.org/doc/devel-manuals#policy）
- “Debian 开发者参考”
  - “最佳实践”文档（https://www.debian.org/doc/devel-manuals#devref）

如果本指南文档的内容与官方的 Debian 文档有所冲突，那么官方的那些总是对的。请使用 **reportbug** 工具向 **debmake-doc** 软件包报告问题。

这里有一些替代性的教程文档，您可以与本指南一起阅读进行参考：

- “Debian 新维护者手册”（较旧）
  - https://www.debian.org/doc/devel-manuals#maint-guide
  - https://packages.qa.debian.org/m/maint-guide.html
- “Debian 打包教程”
  - https://www.debian.org/doc/devel-manuals#packaging-tutorial
  - https://packages.qa.debian.org/p/packaging-tutorial.html
- “Ubuntu 打包指南”（Ubuntu 基于 Debian。）
  - http://packaging.ubuntu.com/html/

提示

阅读这些教程时，您应当考虑使用 debmake 命令替代 dh_make 命令以获得更好的模板文件。

2.6 帮助资源

在您决定在某些公共场合问出您的问题之前，请先做好自己能做到的事情，例如，阅读能找到的文档：

- 软件包的信息可以使用 **aptitude**、**apt-cache** 以及 **dpkg** 命令进行查看。
- 所有相关软件包在 **/usr/share/doc/**软件包名目录下的文件。
- 所有相关命令在 **man** 命令下输出的内容。
- 所有相关命令在 **info** 命令下输出的内容。
- debian-mentors@lists.debian.org 邮件列表存档 的内容。
- debian-devel@lists.debian.org 邮件列表存档 的内容。

要获取您所需要的信息，一种有效的方法是在网页搜索引擎中构建类似“关键字 **site:lists.debian.org**”这样具有限制条件的搜索字符串来限定搜索的域名。

制作一个小型测试用软件包也是了解打包细节的一个好办法。对当前已有的维护良好的软件包进行检查则是了解其他人如何制作软件包的最好方法。

如果您对打包仍然存在疑问，您可以使用以下方式与他人进行沟通：

- debian-mentors@lists.debian.org 邮件列表。（这个邮件列表为专为新手答疑解惑。）

- debian-devel@lists.debian.org 邮件列表。（这个邮件列表针对熟练用户和高级开发者。）
- IRC（互联网中继聊天） 例如 #debian-mentors。
- 专注某个特定软件包集合的团队。（完整列表请见 https://wiki.debian.org/Teams）
- 特定语言的邮件列表。
  - debian-devel-{french,italian,portuguese,spanish}@lists.debian.org
  - debian-devel@debian.or.jp

如果您在做好功课后能在这些场合中合适地提出您的疑问的话，那些更有经验的 Debian 开发者会很愿意帮助您。

小心

Debian 的开发是一个不断变动的目标。您在网上找到的某些信息可能是过时的、不正确的或者不适用的，使用时请留意。

2.7 仓库状况

请了解 Debian 仓库的当前状况。

- Debian 已经包含了绝大多数种类程序的软件包。
- Debian 仓库内软件包的数量是活跃维护者的数十倍。
- 遗憾的是，某些软件包缺乏维护者的足够关注。

因此，对已经存在于仓库内的软件包做出贡献是十分欢迎的（这也更有可能得到其他维护者的支持和协助上传）。

提示

来自 **devscripts** 软件包的 **wnpp-alert** 命令可以检查已安装软件中需要接手或已被丢弃的软件包。

2.8 贡献流程

这里使用类 Python 伪代码，给出了向 Debian 贡献名为 **program** 的软件所走的贡献流程：

```
if exist_in_debian(program):
  if is_team_maintained(program):
    join_team(program)
  if is_orphaned(program) # maintainer: Debian QA Group
    adopt_it(program)
  elif is_RFA(program) # Request for Adoption
    adopt_it(program)
  else:
    if need_help(program):
      contact_maintainer(program)
      triaging_bugs(program)
      preparing_QA_or_NMU_uploads(program)
    else:
      leave_it(program)
else: # new packages
```

```
if not is_good_program(program):
  give_up_packaging(program)
elif not is_distributable(program):
  give_up_packaging(program)
else: # worth packaging
  if is_ITPed_by_others(program):
    if need_help(program):
      contact_ITPer_for_collaboration(program)
    else:
      leave_it_to_ITPer(program)
  else: # really new
    if is_applicable_team(program):
      join_team(program)
    if is_DFSG(program) and is_DFSG(dependency(program)):
      file_ITP(program, area="main") # This is Debian
    elif is_DFSG(program):
      file_ITP(program, area="contrib") # This is not Debian
    else: # non-DFSG
      file_ITP(program, area="non-free") # This is not Debian
    package_it_and_close_ITP(program)
```

其中：

- 对 exist_in_debian() 和 is_team_maintained()，需检查：
  - **aptitude** 命令
  - Debian 软件包 网页
  - 团队
- 对 is_orphaned()、is_RFA() 和 is_ITPed_by_others()，需检查：
  - **wnpp-alert** 命令的输出。
  - 需要投入精力和未来的软件包（WNPP）
  - Debian 缺陷报告记录：在 unstable 版本中 wnpp 伪软件包的缺陷记录
  - 需要“关爱”的 Debian 软件包
  - 基于 debtags 浏览 wnpp 缺陷记录
- 对于 is_good_program()，请检查：
  - 这个程序应当有用。
  - 这个程序不应当向 Debian 系统引入安全和维护上的问题。
  - 这个程序应当有良好的文档，其源代码需要可被理解（即，未经混淆）。
  - 这个程序的作者同意软件被打包，且对 Debian 态度友好。[2]
- 对 is_it_DFSG()，及 is_its_dependency_DFSG()，请检查：
  - Debian 自由软件指导方针（DFSG）。
- 对 is_it_distributable()，请检查：
  - 该软件必须有一个许可证，其中应当允许软件被发行。

您或是需要填写并提交一份 *ITP*，或是需要“收养”一个软件包并开始为其工作。请见“Debian 开发者参考（Debian Developer’s Reference）”文档：

- 5.1. 新软件包。
- 5.9. 移动、删除、重命名、丢弃、接手和重新引入软件包。

[2] 这一条不是绝对的要求，但请注意：遇上不友好的上游可能需要大家为此投入大量精力，而一个友好的上游则能协助解决程序的各类问题。

2.9 新手贡献者和维护者

新手贡献者和维护者可能想知道在开始向 Debian 进行贡献之前需要事先学习哪些知识。根据您个人的侧重点不同，下面有我的一些建议供您参考：

- 打包
  - **POSIX shell** 和 **make** 的基本知识。
  - 一些 **Perl** 和 **Python** 的入门知识。
- 翻译
  - 基于 PO 的翻译系统的工作原理和基本知识。
- 文档
  - 文本标记语言的基础知识（XML、ReST、Wiki 等）。

新手贡献者和维护者可能想知道从哪里开始向 Debian 进行贡献。根据您掌握的技能，下面有我的一些建议供您参考：

- **POSIX shell**、**Perl** 和 **Python** 的技巧：
  - 向 Debian 安装器提交补丁。
  - 向 Debian 打包帮助脚本（如本文档中体提及的 **devscripts**、**pbuilder** 等项目）提交补丁。
- **C** 和 **C++** 技能：
  - 向具有 **required** 和 **important** 优先级的软件包提交补丁。
- 英语之外的技能：
  - 向 Debian 安装器项目提交补丁。
  - 为具有 **required** 和 **important** 优先级的软件包中的 PO 文件提交补丁。
- 文档技能：
  - 更新 Debian 维基（Wiki） 中的内容。
  - 向已有的 Debian 文档 提交补丁。

这些活动应当能让您在各位 Debian 社区成员之间得到存在感，从而建立您的信誉与名气。
新手维护者应当避免打包具有潜在高度安全隐患的程序：

- **setuid** 或 **setgid** 程序
- 守护进程（**daemon**）程序
- 安装至 **/sbin/** 或 **/usr/sbin/** 目录的程序

在积累足够的打包经验后，您可以再尝试打包这样的程序。

Chapter 3

工具的配置

build-essential 软件包必须在构建环境内预先安装。

devscripts 软件包应当安装在维护者的工作环境中。

尽管这个不是绝对的要求，但是在维护者的工作环境内装上并配置好本章节提到的各个常用的软件包会有助于维护者高效投入工作。这些软件可以构成我们共同确立的一个基准工作环境。

如有需要，请同样按需安装在“Debian 开发者参考”文中 Debian 维护者工具概览 一节提到的各个工具。

小心

这里展示的工具配置方式仅作为示例提供，可能与系统上最新的软件包相比有所落后。Debian 的开发具有一个移动的目标。请确保阅读合适的文档并按照需要更新配置内容。

3.1 电子邮件地址

许多 Debian 维护工具识别并使用 shell 环境变量 **$DEBEMAIL** 和 **$DEBFULLNAME** 作为作为您的电子邮件地址和名称。

我们可以通过将下面几行加入 **~/.bashrc**[1] 的方式对这些软件进行配置。

添加至 **~/.bashrc** 文件

```
DEBEMAIL="your.email.address@example.org"
DEBFULLNAME="Firstname Lastname"
export DEBEMAIL DEBFULLNAME
```

3.2 mc

mc 命令提供了管理文件的简单途径。它可以打开二进制 **deb** 文件，并仅需对二进制 **deb** 文件按下回车键便能检查其内容。它调用了 **dpkg-deb** 命令作为其后端。我们可以按照下列方式对其配置，以支持简易 **chdir** 操作。

添加至 **~/.bashrc** 文件

```
# mc related
export HISTCONTROL=ignoreboth
. /usr/lib/mc/mc.sh
```

[1] 这里假设您正在使用 Bash 并以此作为登录默认 shell。如果您设置了其它登录 shell，例如 Z shell，请使用它们对应的配置文件替换 *~/.bashrc* 文件。

3.3 git

如今 **git** 命令已成为管理带历史的源码树的必要工具。

git 命令的用户级全局配置，如您的名字和电子邮件地址，保存在 **~/.gitconfig** 文件中，且可以使用如下方式配置。

```
$ git config --global user.name "Name Surname"
$ git config --global user.email yourname@example.com
```

如果您仍然只习惯 CVS 或者 Subversion 的命令风格，您可以使用如下方式设置几个命令别名。

```
$ git config --global alias.ci "commit -a"
$ git config --global alias.co checkout
```

您可以使用如下命令检查全局配置。

```
$ git config --global --list
```

提示

有必要使用某些图形界面 git 工具，例如 **gitk** 或 **gitg** 命令来有效地处理 git 仓库的历史。

3.4 quilt

quilt 命令提供了记录修改的一个基本方式。对 Debian 打包来说，该工具需要进行自定义，从而在 **debian/patches/** 目录内记录修改内容，而非使用默认的 **patches/** 目录。

为了避免改变 **quilt** 命令自身的行为，我们在这里创建一个用于 Debian 打包工作的命令别名：**dquilt**。之后，我们将对应内容写入 **~/.bashrc** 文件。下面给出的第二行为 **dquilt** 命令提供与 **quilt** 命令相同的命令行补全功能。

添加至 **~/.bashrc** 文件

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

然后我们来创建具有如下内容的 **~/.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
```

请参考 **quilt**(1) 和 处理大量补丁的方法暨对 Quilt 的介绍 以了解如何使用 **quilt** 命令。

要获取使用示例，请查看第 4.8 节。

3.5 devscripts

debsign 命令由 **devscripts** 软件包提供，它可以使用用户的 GPG 私钥对 Debian 软件包进行签名。

debuild 命令同样由 **devscripts** 软件包提供，它可以构建二进制软件包并使用 **lintian** 命令对其进行检查。**lintian** 命令的详细输出通常都很实用。

您可以将下列内容写入 **~/.devscripts** 文件来进行配置。

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

用于 **dpkg-source** 命令的 **DEBUILD_DPKG_BUILDPACKAGE_OPTS** 中可以额外使用 **-i** 和 **-I** 选项以帮助构建源码中具有外来无关内容的软件包（参见第 5.15 节）。

当前情况下，使用 4096 位的 RSA 密钥是较好的做法。另见 创建一个新 GPG 密钥。

3.6 pbuilder

pbuilder 软件包提供了净室（干净的）（**chroot**）构建环境。[2]

我们可以搭配使用另外几个辅助软件包对其自定义。

- **cowbuilder** 软件包能加速 chroot 创建过程。
- **lintian** 软件包能找到所构建软件包中的缺陷。
- **bash**、**mc** 和 **vim** 软件包在构建失败时用来查找问题。
- **ccache** 软件包可以加速 **gcc**。（可选）
- **libeatmydata1** 软件包可以加速 **dpkg**。（可选）
- 并行运行 **make** 以提高构建速度。（可选）

警告

可选的自定义项可能造成负面影响。如果有疑问，请禁用它们。

我们使用如下给出的内容来创建 **~/.pbuilderrc** 文件（所有可选功能均已禁用）。

```
AUTO_DEBSIGN="${AUTO_DEBSIGN:-no}"
PDEBUILD_PBUILDER=cowbuilder
HOOKDIR="/var/cache/pbuilder/hooks"
MIRRORSITE="http://deb.debian.org/debian/"
#APTCACHE=/var/cache/pbuilder/aptcache
APTCACHE=/var/cache/apt/archives
#BUILDRESULT=/var/cache/pbuilder/result/
BUILDRESULT=../
EXTRAPACKAGES="ccache lintian libeatmydata1"

# enable to use libeatmydata1 for pbuilder
#export LD_PRELOAD=${LD_PRELOAD+$LD_PRELOAD:}libeatmydata.so

# enable ccache for pbuilder
#export PATH="/usr/lib/ccache${PATH+:$PATH}"
#export CCACHE_DIR="/var/cache/pbuilder/ccache"
#BINDMOUNTS="${CCACHE_DIR}"

# parallel make
#DEBBUILDOPTS=-j8
```

[2] **sbuild** 软件包提供了另一套 chroot 平台。

注意

可以考虑创建从 **/root/.pbuilderrc** 到 **/home/<user>/.pbuilderrc** 的符号链接以获得一致的体验。

注意

由于 缺陷 #606542，您可能需要手动将 **EXTRAPACKAGES** 列出的软件包安装进入 chroot。请见第 7.10 节。

注意

应当在 chroot 环境内外同时安装上 **libeatmydata1** (>=82-2)，否则即为禁用 **libeatmydata1**。该软件包在某些构建系统中可能导致竞争情况（race condition）发生。

注意

并行的 **make** 可能在某些已有软件包上运行失败，它同样会使得构建日志难以阅读。

我们可以按如下方式创建钩子脚本。

/var/cache/pbuilder/hooks/A10ccache

```
#!/bin/sh
set -e
# increase the ccache caching size
ccache -M 4G
# output the current statistics
ccache -s
```

/var/cache/pbuilder/hooks/B90lintian

```
#!/bin/sh
set -e
apt-get -y --allow-downgrades install lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" -l pbuilder
echo "+++ end of lintian output +++"
```

/var/cache/pbuilder/hooks/C10shell

```
#!/bin/sh
set -e
apt-get -y --allow-downgrades install vim bash mc
# invoke shell if build fails
cd /tmp/buildd/*/debian/..
/bin/bash < /dev/tty > /dev/tty 2> /dev/tty
```

注意

所有这些脚本都需要设置为全局可执行 :“**-rwxr-xr-x 1 root root**”。

注意

ccache 的缓存目录 **/var/cache/pbuilder/cache** 需要为了 **pbuilder** 命令的使用而设置为全局可写 :“**-rwxrwxrwx 1 root root**”。您需要明白这样会带来相关的安全隐患。

3.7 git-buildpackage

您也可能需要在 **~/.gbp.conf** 中设置全局配置信息

```
# Configuration file for "gbp <command>"

[DEFAULT]
# the default build command:
builder = git-pbuilder -i -I -us -uc
# use pristine-tar:
pristine-tar = True
# Use color when on a terminal, alternatives: on/true, off/false or auto
color = auto
```

提示

这里的 **gbp** 命令是 **git-buildpackage** 命令的一个别名。

3.8 HTTP 代理

您应当在本地设置 HTTP 缓存代理以节约访问 Debian 软件仓库的带宽。可以考虑以下几种选项:

- 简单的 HTTP 缓存代理，使用 **squid** 软件包。
- 特化的 HTTP 缓存代理，使用 **apt-cacher-ng** 软件包。

3.9 私有 Debian 仓库

您可以使用 **reprepro** 软件包搭建私有 Debian 仓库。

Chapter 4

简单例子

有一句古罗马谚语说得好：“一例胜千言!”

这里给出了从简单的 C 语言源代码创建简单的 Debian 软件包的例子，并假设上游使用了 **Makefile** 作为其构建系统。

我们假设上游源码压缩包（tarball）名称为 **debhello-0.0.tar.gz**。

这一类源代码设计可以用这样的方式安装成为非系统文件：

```
$ tar -xzmf debhello-0.0.tar.gz
$ cd debhello-0.0
$ make
$ make install
```

Debian 的打包需要对“**make install**”流程进行改变，从而将文件安装至目标系统镜像所在位置，而非通常使用的 **/usr/local** 下的位置。

注意

在其它更加复杂的构建系统下构建 Debian 软件包的例子可以在第 8 章找到。

4.1 大致流程

从上游源码压缩包 **debhello-0.0.tar.gz** 构建单个非本土 Debian 软件包的大致流程可以总结如下：

- 维护者获取上游源码压缩包 **debhello-0.0.tar.gz** 并将其内容解压缩至 **debhello-0.0** 目录中。
- **debmake** 命令对上游源码树进行 debian 化（debianize），具体来说，是创建一个 **debian** 目录并仅向该目录中添加各类模板文件。
  - 名为 **debhello_0.0.orig.tar.gz** 的符号链接被创建并指向 **debhello-0.0.tar.gz** 文件。
  - 维护者须自行编辑修改模板文件。
- **debuild** 命令基于已 debian 化的源码树构建二进制软件包。
  - **debhello-0.0-1.debian.tar.xz** 将被创建，它包含了 **debian** 目录。

软件包构建的大致流程

```
$ tar -xzmf debhello-0.0.tar.gz
$ cd debhello-0.0
$ debmake
  ... manual customization
$ debuild
  ...
```

提示

此处和下面例子中的 **debuild** 命令可使用等价的命令进行替换，例如 **pdebuild** 命令。

提示

如果上游源码压缩包提供了 **.tar.xz** 格式文件，请使用这样的压缩包来替代 **.tar.gz** 或 **.tar.bz2** 格式。**xz** 压缩与 **gzip** 或 **bzip2** 压缩相比提供了更好的压缩比。

4.2 什么是 **debmake**?

文中的 **debmake** 命令是用于 Debian 打包的一个帮助脚本。

- 它总是将大多数选项的状态与参数设置为合理的默认值。
- 它能产生上游源码包，并按需创建所需的符号链接。
- 它不会覆写 **debian/** 目录下已存在的配置文件。
- 它支持多架构（**multiarch**）软件包。
- 它能创建良好的模板文件，例如符合 **DEP-5** 的 **debian/copyright** 文件。

这些特性使得使用 **debmake** 进行 Debian 打包工作变得简单而现代化。

注意

debmake 命令并不是制作一个 Debian 软件包的唯一途径。许多软件包是打包者模仿其它已有的打包示例，仅仅使用文本编辑器而编写完成打包脚本的。

4.3 什么是 **debuild**?

这里给出与 **debuild** 命令类似的一系列命令的一个汇总。

- **debian/rules** 文件定义了 Debian 二进制软件包该如何构建。
- **dpkg-buildpackge** 是构建 Debian 二进制软件包的正式命令。对于正常的二进制构建，它大体上会执行以下操作:
  - “**dpkg-source --before-build**”（应用 Debian 补丁，除非它们已被应用）
  - “**fakeroot debian/rules clean**”
  - “**dpkg-source --build**”（构建 Debian 源码包）
  - “**fakeroot debian/rules build**”
  - “**fakeroot debian/rules binary**”
  - “**dpkg-genbuildinfo**”（产生一个 ***.buildinfo** 文件）
  - “**dpkg-genchanges**”（产生一个 ***.changes** 文件）

  - "**fakeroot debian/rules clean**"
  - "**dpkg-source --after-build**"（取消 Debian 补丁，除非使用了 **--no-unapply-patches** 选项）
  - "**debsign**"（对 ***.dsc** 和 ***.changes** 文件进行签名）
    * 如果您按照第 3.5 节的说明设置了 **-us** 和 **-us** 选项的话，本步骤将会被跳过。您需要手动运行 **debsign** 命令。

- **debuild** 命令是 **dpkg-buildpackage** 命令的一个封装脚本，它可以使用合适的环境变量来构建 Debian 二进制软件包。
- **pdebuild** 命令是另一个封装脚本，它可以在合适的 chroot 环境下使用合适的环境变量构建 Debian 二进制软件包。
- **git-pbuilder** 命令是又一个用于构建 Debian 二进制软件包的封装脚本，它同样可以确保使用合适的环境变量和 chroot 环境。不同之处在于它提供了一个更容易使用的命令行用户界面，可以较方便地在不同的构建环境之间进行切换。

注意

如需了解详细内容，请见 **dpkg-buildpackage**(1)。

4.4 第一步：获取上游源代码

我们先要获取上游源代码。

下载 **debhello-0.0.tar.gz**

```
 $ wget http://www.example.org/download/debhello-0.0.tar.gz
 ...
 $ tar -xzmf debhello-0.0.tar.gz
 $ tree
.
├── debhello-0.0
│   ├── LICENSE
│   ├── Makefile
│   └── src
│       └── hello.c
└── debhello-0.0.tar.gz

2 directories, 4 files
```

这里的 C 源代码 **hello.c** 非常的简单。

hello.c

```
 $ cat debhello-0.0/src/hello.c
#include <stdio.h>
int
main()
{
        printf("Hello, world!\n");
        return 0;
}
```

这里，源码中的 **Makefile** 支持 GNU 编码标准 和 FHS（文件系统层级规范）。特别地：

- 构建二进制程序时会考虑 **$(CPPFLAGS)**、**$(CFLAGS)**、**$(LDFLAGS)**，等等。
- 安装文件时采纳 **$(DESTDIR)** 作为目标系统镜像的路径前缀
- 安装文件时使用 **$(prefix)** 的值，以便我们将其设置覆盖为 **/usr**

Makefile

```
 $ cat debhello-0.0/Makefile
prefix = /usr/local

all: src/hello

src/hello: src/hello.c
        @echo "CFLAGS=$(CFLAGS)" | \
                fold -s -w 70 | \
                sed -e 's/^/# /'
        $(CC) $(CPPFLAGS) $(CFLAGS) $(LDCFLAGS) -o $@ $^

install: src/hello
        install -D src/hello \
                $(DESTDIR)$(prefix)/bin/hello

clean:
        -rm -f src/hello

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello

.PHONY: all install clean distclean uninstall
```

注意

对 **$(CFLAGS)** 的 **echo** 命令用于在接下来的例子中验证所设置的构建参数。

4.5 第二步：使用 **debmake** 产生模板文件

提示

如果 **debmake** 命令调用时使用了 **-T** 选项，程序将为模板文件产生更加详细的注释内容。

debmake 命令的输出十分详细，如下所示，它可以展示程序的具体操作内容。

```
 $ cd debhello-0.0
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="0.0", rev="1"
I: *** start packaging in "debhello-0.0". ***
I: provide debhello_0.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-0.0.tar.gz debhello_0.0.orig.tar.gz
I: pwd = "/path/to/debhello-0.0"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: analyze the source tree
```

```
I: build_type = make
I: scan source for copyright+license text and file extensions
I: 100 %, ext = c
I: check_all_licenses
I: ..
I: check_all_licenses completed for 2 files.
I: bunch_all_licenses
I: format_all_licenses
I: make debian/* template files
I: single binary package
I: debmake -x "1" ...
I: creating => debian/control
I: creating => debian/copyright
I: substituting => /usr/share/debmake/extra0/rules
I: creating => debian/rules
I: substituting => /usr/share/debmake/extra0/changelog
I: creating => debian/changelog
I: substituting => /usr/share/debmake/extra1/compat
I: creating => debian/compat
I: substituting => /usr/share/debmake/extra1/watch
I: creating => debian/watch
I: substituting => /usr/share/debmake/extra1/README.Debian
I: creating => debian/README.Debian
I: substituting => /usr/share/debmake/extra1source/local-options
I: creating => debian/source/local-options
I: substituting => /usr/share/debmake/extra1source/format
I: creating => debian/source/format
I: substituting => /usr/share/debmake/extra1patches/series
I: creating => debian/patches/series
I: run "debmake -x2" to get more template files
I: $ wrap-and-sort
```

debmake 命令基于命令行选项产生所有这些模板文件。如果没有指定具体选项，**debmake** 命令将为您自动选择合理的默认值:

- 源码包名称：**debhello**
- 上游版本：**0.0**
- 二进制软件包名称：**debhello**
- Debian 修订版本：**1**
- 软件包类型：**bin**（ELF 二进制可执行程序软件包）
- **-x** 选项：**-x1**（是单个二进制软件包的默认值）

我们来检查一下自动产生的模板文件。
基本 **debmake** 命令运行后的源码树。

```
 $ cd ..
 $ tree
.
├── debhello-0.0
│   ├── LICENSE
│   ├── Makefile
│   ├── debian
│   │   ├── README.Debian
│   │   ├── changelog
│   │   ├── compat
│   │   ├── control
│   │   ├── copyright
│   │   ├── patches
│   │   │   └── series
│   │   ├── rules
│   │   ├── source
```

```
│   │   │   ├── format
│   │   │   └── local-options
│   │   └── watch
│   └── src
│       └── hello.c
├── debhello-0.0.tar.gz
└── debhello_0.0.orig.tar.gz -> debhello-0.0.tar.gz

5 directories, 15 files
```

这里的 **debian/rules** 文件是应当由软件包维护者提供的构建脚本。此时该文件是由 **debmake** 命令产生的模板文件。

debian/rules（模板文件）：

```
 $ cat debhello-0.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

#override_dh_auto_install:
#       dh_auto_install -- prefix=/usr

#override_dh_install:
#       dh_install --list-missing -X.pyc -X.pyo
```

这便是使用 **dh** 命令时标准的 **debian/rules** 文件。（某些内容已被注释，可供后续修改使用。）

这里的 **debian/control** 文件提供了 Debian 软件包的主要元信息。此时该文件是由 **debmake** 命令产生的模板文件。

debian/control（模板文件）：

```
 $ cat debhello-0.0/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

警告

如果您对 **debian/control** 模板文件中的“**Section: unknown**”部分不做修改的话，后续的 **lintian** 错误可能导致构建失败。

因为这是个 ELF 二进制可执行文件软件包，**debmake** 命令设置了 “**Architecture: any**” 和 “**Multi-Arch: foreign**” 两项。同时，它将所需的 **substvar** 参数设置为 “**Depends: ${shlibs:Depends}, ${misc:Depends}**”。

这些内容将在第 5 章部分进行解释。

注意

请注意这个 **debian/control** 按照“Debian 政策手册”中 5.2 源码包控制文件——debian/control 的内容使用 RFC-822 风格进行编写。文件对空行和行首空格的使用有特别的要求。

这里的 **debian/copyright** 提供了 Debian 软件包版权数据的总结。此时该文件是由 **debmake** 命令产生的模板文件。

debian/copyright（模板文件）：

```
 $ cat debhello-0.0/debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Source: <url://example.com>
#
# Please double check copyright with the licensecheck(1) command.

Files:     Makefile
           src/hello.c
Copyright: __NO_COPYRIGHT_NOR_LICENSE__
License:   __NO_COPYRIGHT_NOR_LICENSE__

#-------------------------------------------------------------------------------...
# Files marked as NO_LICENSE_TEXT_FOUND may be covered by the following
# license/copyright files.

#-------------------------------------------------------------------------------...
# License file: LICENSE
 License:
 .
 All files in this archive are licensed under the MIT License as below.
 .
 Copyright 2015 Osamu Aoki <osamu@debian.org>
 .
 Permission is hereby granted, free of charge, to any person obtaining a
 copy of this software and associated documentation files (the "Software"),
 to deal in the Software without restriction, including without limitation
 the rights to use, copy, modify, merge, publish, distribute, sublicense,
 and/or sell copies of the Software, and to permit persons to whom the
 Software is furnished to do so, subject to the following conditions:
 .
 The above copyright notice and this permission notice shall be included
 in all copies or substantial portions of the Software.
 .
 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

4.6 第三步：编辑模板文件

作为维护者，要制作一个合适的 Debian 软件包当然需要对模板内容进行一些手工的调整。

为了将安装文件变成系统文件的一部分，**Makefile** 文件中 **$(prefix)** 默认的 **/usr/local** 的值需要被覆盖为 **/usr**。要做到这点，可以按照下面给出的方法，在 **debian/rules** 文件中添加名为 **override_dh_auto_install** 的目标，在其中设置“**prefix=/usr**”。

debian/rules（维护者版本）：

```
 $ vim debhello-0.0/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-0.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

override_dh_auto_install:
        dh_auto_install -- prefix=/usr
```

如上在 **debian/rules** 文件中导出 =**DH_VERBOSE** 环境变量可以强制 **debhelper** 工具输出细粒度的构建报告。

如上导出 **DEB_BUILD_MAINT_OPTION** 变量可以如 **dpkg-buildflags**(1) 手册页中“FEATURE AREAS/ENVIRONMENT”部分所说，对加固选项进行设置。[1]

如上导出 **DEB_CFLAGS_MAINT_APPEND** 可以强制 C 编译器给出所有类型的警告内容。

如上导出 **DEB_LDFLAGS_MAINT_APPEND** 可以强制链接器只对真正需要的库进行链接。[2]

对基于 Makefile 的构建系统来说，**dh_auto_install** 命令所做的基本上就是“**$(MAKE) install DESTDIR=debian/debhello**”。这里创建的 **override_dh_auto_install** 目标将其行为修改为“**$(MAKE) install DESTDIR=debian/debhello prefix=/usr**”。

这里是维护者版本的 **debian/control** 和 **debian/copyright** 文件。

debian/control（维护者版本）：

```
 $ vim debhello-0.0/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-0.0/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

debian/copyright（维护者版本）：

```
 $ vim debhello-0.0/debian/copyright
 ... hack, hack, hack, ...
 $ cat debhello-0.0/debian/copyright
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: debhello
Source: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/tree/base...
```

[1] 这里的做法是为了进行加固而强制启用只读重定位链接，以此避免 lintian 的警告“**W: debhello: hardening-no-relro usr/bin/hello**”。其实它在本例中并不是必要的，但加上也没有什么坏处。对于没有外部链接库的本例来说，lintian 似乎给出了误报的警告。

[2] 这里的做法是为了避免在依赖库情况复杂的情况下过度链接，例如某些 GNOME 程序。这样做对这里的简单例子来说并不是必要的，但应当是无害的。

```
Files:     *
Copyright: 2015 Osamu Aoki <osamu@debian.org>
License:   MIT
 Permission is hereby granted, free of charge, to any person obtaining a
 copy of this software and associated documentation files (the "Software"),
 to deal in the Software without restriction, including without limitation
 the rights to use, copy, modify, merge, publish, distribute, sublicense,
 and/or sell copies of the Software, and to permit persons to whom the
 Software is furnished to do so, subject to the following conditions:
 .
 The above copyright notice and this permission notice shall be included
 in all copies or substantial portions of the Software.
 .
 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

debian/. 下面的模板文件（**0.0** 版）：

```
 $ tree debhello-0.0/debian
debhello-0.0/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 10 files
```

提示

对于来自 **debhelper** 软件包的各个 **dh_*** 命令来说，它们在读取所使用的配置文件时通常把以 # 开头的行视为注释行。

4.7 第四步：使用 **debuild** 构建软件包

您可以使用 **debuild** 或者等效的命令工具（参见第 4.3 节）在这个源码树内构建一个非本土 Debian 软件包。命令的输出通常十分详细，如下所示，它会对构建中执行的操作进行解释。

```
 $ cd debhello-0.0
 $ debuild
 dpkg-buildpackage -rfakeroot -us -uc -ui -i
 ...
 fakeroot debian/rules clean
dh clean
```

```
 ...
 debian/rules build
dh build
   dh_update_autotools_config
   dh_autoreconf
   dh_auto_configure
   dh_auto_build
        make -j4 "INSTALL=install --strip-program=true"
make[1]: Entering directory '/path/to/debhello-0.0'
# CFLAGS=-g -O2 -fdebug-prefix-map=/build/debmake-doc-1.11=.
# -fstack-protector-strong -Wformat -Werror=format-security
cc -Wdate-time -D_FORTIFY_SOURCE=2 -g -O2 -fdebug-prefix-map=/build/debmake-d...
 ...
 fakeroot debian/rules binary
dh binary
 ...
Now running lintian debhello_0.0-1_amd64.changes ...
 ...
W: debhello: binary-without-manpage usr/bin/hello
Finished running lintian.

 ...
```

这里验证了 **CFLAGS** 已经得到了更新,添加了 **-Wall** 和 **-pendatic** 参数;这是我们先前在 **DEB_CFLAGS_MAINT_AP** 变量中所指定的。

根据 **lintian** 的报告，您应该如同后文中的例子那样（请见第 8 章）为软件包添加 man 手册页。我们这里暂且跳过这部分内容。

现在我们来看看成果如何。

debhello 0.0 版使用 **debuild** 命令产生的文件：

```
 $ cd ..
 $ tree -FL 1
.
├── debhello-0.0/
├── debhello-0.0.tar.gz
├── debhello-dbgsym_0.0-1_amd64.deb
├── debhello_0.0-1.debian.tar.xz
├── debhello_0.0-1.dsc
├── debhello_0.0-1_amd64.build
├── debhello_0.0-1_amd64.buildinfo
├── debhello_0.0-1_amd64.changes
├── debhello_0.0-1_amd64.deb
└── debhello_0.0.orig.tar.gz -> debhello-0.0.tar.gz

1 directory, 9 files
```

您可以看见生成的全部文件。

- **debhello_0.0.orig.tar.gz** 是指向上游源码压缩包的符号链接。
- **debhello_0.0-1.debian.tar.xz** 包含了维护者生成的内容。
- **debhello_0.0-1.dsc** 是 Debian 源码包的元数据文件。
- **debhello_0.0-1_amd64.deb** 是 Debian 二进制软件包。
- The **debhello-dbgsym_0.0-1_amd64.deb** is the Debian debug symbol binary package. See 第 5.17.1 节.
- The **debhello_0.0-1_amd64.build** file is the build log file.
- The **debhello_0.0-1_amd64.buildinfo** file is the meta data file generated by **dpkg-genbuildinfo**(1).
- **debhello_0.0-1_amd64.changes** 是 Debian 二进制软件包的元数据文件。

debhello_0.0-1.debian.tar.xz 包含了 Debian 对上游源代码的修改，具体如下所示。

压缩文件 **debhello_0.0-1.debian.tar.xz** 的内容：

```
 $ tar -tzf debhello-0.0.tar.gz
debhello-0.0/
debhello-0.0/Makefile
debhello-0.0/src/
debhello-0.0/src/hello.c
debhello-0.0/LICENSE
 $ tar --xz -tf debhello_0.0-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/compat
debian/control
debian/copyright
debian/patches/
debian/patches/series
debian/rules
debian/source/
debian/source/format
debian/watch
```

The **debhello_0.0-1_amd64.deb** contains the binary files to be installed to the target system.
The **debhello-debsym_0.0-1_amd64.deb** contains the debug symbol files to be installed to the target system..
The binary package contents of all binary packages:

```
 $ dpkg -c debhello-dbgsym_0.0-1_amd64.deb
drwxr-xr-x root/root ...  ./
drwxr-xr-x root/root ...  ./usr/
drwxr-xr-x root/root ...  ./usr/lib/
drwxr-xr-x root/root ...  ./usr/lib/debug/
drwxr-xr-x root/root ...  ./usr/lib/debug/.build-id/
drwxr-xr-x root/root ...  ./usr/lib/debug/.build-id/ce/
-rw-r--r-- root/root ...  ./usr/lib/debug/.build-id/ce/88708a35cc4e7d0f0b67bc...
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/doc/
lrwxrwxrwx root/root ...  ./usr/share/doc/debhello-dbgsym -> debhello
 $ dpkg -c debhello_0.0-1_amd64.deb
drwxr-xr-x root/root ...  ./
drwxr-xr-x root/root ...  ./usr/
drwxr-xr-x root/root ...  ./usr/bin/
-rwxr-xr-x root/root ...  ./usr/bin/hello
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/doc/
drwxr-xr-x root/root ...  ./usr/share/doc/debhello/
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/README.Debian
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/changelog.Debian.gz
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/copyright
```

The generated dependency list of all binary packages.
The generated dependency list of all binary packages (v=0.0):

```
 $ dpkg -f debhello-dbgsym_0.0-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 0.0-1)
 $ dpkg -f debhello_0.0-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libc6 (>= 2.2.5)
```

小心

在将软件包上传至 Debian 仓库之前，仍然有很多细节需要进行处理。

注意

如果跳过了对 **debmake** 命令自动生成的配置文件的手工调整步骤，所生成的二进制软件包可能缺少有用的软件包描述信息，某些政策的要求也无法满足。这个不正式的软件包对于 **dpkg** 命令来说可以正常处理，也许这样对您本地的部署来说已经足够好了。

4.8 第三步（备选）：修改上游源代码

上面的例子中，在创建合适的 Debian 软件包时没有修改上游的源代码。

作为维护者，另一个备选的方案是对上游源代码做改动，如修改上游的 **Makefile** 以将 $(prefix) 的值设定为 **/usr**。

打包操作基本上和上面的分步示例相同，除了在第 4.6 节中的两点：

- 要将维护者对上游源代码的修改形成对应的补丁文件存放在 **debian/patches/** 目录内，并将它们的文件名写入 **debian/patches/series** 文件，如第 5.8 节所述。有数种生成补丁文件的方式。下面的章节中给出了一些例子：
  - 第 4.8.1 节
  - 第 4.8.2 节
  - 第 4.8.3 节
- 此时维护者对 **debian/rules** 文件的修改如下所示，它不包含 **override_dh_auto_install** 目标：

  debian/rules（备选的维护者版本）：

```
 $ cd debhello-0.0
 $ vim debian/rules
 ... hack, hack, hack, ...
 $ cat debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@
```

这个使用一系列补丁文件进行 Debian 打包的备选方案对于应对上游未来的改变可能不够健壮，但是在应对更为复杂的上游源代码时可以更灵活。（参见第 7.13 节。）

注意

对当前这个特定的打包场景，前文的第 4.6 节中使用 **debian/rules** 文件的方式更好一些。但为了演示起见，此时我们先使用本节的方式继续操作。

提示

对更复杂的打包场景，可能需要同时应用第 4.6 节和第 4.8 节中的方式。

4.8.1 使用 **diff -u** 处理补丁

这里我们使用 **diff** 命令创建一个 **000-prefix-usr.patch** 文件作为例子。

```
 $ cp -a debhello-0.0 debhello-0.0.orig
 $ vim debhello-0.0/Makefile
 ... hack, hack, hack, ...
 $ diff -Nru debhello-0.0.orig debhello-0.0 >000-prefix-usr.patch
 $ cat 000-prefix-usr.patch
diff -Nru debhello-0.0.orig/Makefile debhello-0.0/Makefile
--- debhello-0.0.orig/Makefile  2018-08-09 13:57:15.925952917 +0000
+++ debhello-0.0/Makefile       2018-08-09 13:57:16.009952918 +0000
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello

 $ rm -rf debhello-0.0
 $ mv -f debhello-0.0.orig debhello-0.0
```

请注意，上游的源码树应当恢复到原始状态，补丁文件此时的名字为 **000-prefix-usr.patch**。

这个 **000-prefix-usr.patch** 文件随后应当进行编辑以使其符合 DEP-3，并照如下方式移动至正确的位置。

```
 $ cd debhello-0.0
 $ echo '000-prefix-usr.patch' >debian/patches/series
 $ vim ../000-prefix-usr.patch
 ... hack, hack, hack, ...
 $ mv -f ../000-prefix-usr.patch debian/patches/000-prefix-usr.patch
 $ cat debian/patches/000-prefix-usr.patch
From: Osamu Aoki <osamu@debian.org>
Description: set prefix=/usr patch
diff -Nru debhello-0.0.orig/Makefile debhello-0.0/Makefile
--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello
```

4.8.2 使用 **dquilt** 处理补丁

这里的例子使用 **dquilt** 命令（一个 **quilt** 程序的简单封装）创建 **000-prefix-usr.patch**。**dquilt** 命令的语法和功能与 **quilt**(1) 命令相同，唯一的区别在于补丁存储在 **debian/patches/** 目录中。

```
 $ cd debhello-0.0
 $ dquilt new 000-prefix-usr.patch
Patch debian/patches/000-prefix-usr.patch is now on top
 $ dquilt add Makefile
File Makefile added to patch debian/patches/000-prefix-usr.patch
 ... hack, hack, hack, ...
 $ head -1 Makefile
prefix = /usr
 $ dquilt refresh
Refreshed patch debian/patches/000-prefix-usr.patch
 $ dquilt header -e --dep3
 ... edit the DEP-3 patch header with editor
 $ tree -a
.
├── .pc
│   ├── .quilt_patches
│   ├── .quilt_series
```

```
│   ├── .version
│   ├── 000-prefix-usr.patch
│   │   ├── .timestamp
│   │   └── Makefile
│   └── applied-patches
├── LICENSE
├── Makefile
├── debian
│   ├── README.Debian
│   ├── changelog
│   ├── compat
│   ├── control
│   ├── copyright
│   ├── patches
│   │   ├── 000-prefix-usr.patch
│   │   └── series
│   ├── rules
│   ├── source
│   │   ├── format
│   │   └── local-options
│   └── watch
└── src
    └── hello.c

6 directories, 20 files
 $ cat debian/patches/series
000-prefix-usr.patch
 $ cat debian/patches/000-prefix-usr.patch
Description: set prefix=/usr patch
Author: Osamu Aoki <osamu@debian.org>
Index: debhello-0.0/Makefile
===================================================================
--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello
```

这里，上游源码树中的 **Makefile** 文件没有恢复到原始状态的必要。在第 4.7 节描述的 Debian 打包过程中调用的 **dpkg-source** 命令能够理解由 **dquilt** 程序在 **.pc/** 目录中记录的补丁应用情况。只要所有这些修改都是由 **dquilt** 命令完成的，那么 Debian 源码包就可以从经过修改的源码树中进行构建。

注意

如果 **.pc/** 目录不存在，**dpkg-source** 命令就会假定没有应用任何补丁。这就是更为原始的补丁生成方法，例如第 4.8.1 节中未生成 **.pc/** 目录的情况下要求将上游源码树进行恢复的原因。

4.8.3 使用 dpkg-source --commit 处理补丁

这里给出使用“**dpkg-source --commit**”命令生成 **000-prefix-usr.patch** 的例子。

我们先来编辑上游源代码。

```
 $ cd debhello-0.0
 $ vim Makefile
 ... hack, hack, hack, ...
 $ head -n1 Makefile
prefix = /usr
```

我们来进行提交。

```
 $ dpkg-source --commit . 000-prefix-usr.patch
... editor to edit the DEP-3 patch header
...
```

我们来看看效果如何。

```
 $ cat debian/patches/series
000-prefix-usr.patch
 $ cat debian/patches/000-prefix-usr.patch
Description: set prefix=/usr patch
Author: Osamu Aoki <osamu@debian.org>
Index: debhello-0.0/Makefile

--- debhello-0.0.orig/Makefile
+++ debhello-0.0/Makefile
@@ -1,4 +1,4 @@
-prefix = /usr/local
+prefix = /usr

 all: src/hello

 $ tree -a
.
├── .pc
│   ├── .quilt_patches
│   ├── .quilt_series
│   ├── .version
│   ├── 000-prefix-usr.patch
│   │   ├── .timestamp
│   │   └── Makefile
│   └── applied-patches
├── LICENSE
├── Makefile
├── debian
│   ├── README.Debian
│   ├── changelog
│   ├── compat
│   ├── control
│   ├── copyright
│   ├── patches
│   │   ├── 000-prefix-usr.patch
│   │   └── series
│   ├── rules
│   ├── source
│   │   ├── format
│   │   └── local-options
│   └── watch
└── src
    └── hello.c

6 directories, 20 files
```

这里，**dpkg-source** 命令完成了与第 4.8.2 节一节中使用 **dquilt** 命令完全相同的流程。

Chapter 5

基本内容

这里展示了 Debian 打包工作中针对非本土软件包使用“**3.0 (quilt)**”格式进行打包所遵循基本规则的一个全局性概览。

注意

为简明起见，某些细节被有意跳过。请按需查阅对应命令的手册页，例如 **dpkg-source**(1)、**dpkg-buildpackage**(1)、**dpkg**(1)、**dpkg-deb**(1)、**deb**(5)，等等。

Debian 源码包是一组用于构建 Debian 二进制软件包的输入文件，而非单个文件。

Debian 二进制软件包是一个特殊的档案文件，其中包含了一系列可安装的二进制数据及与它们相关的信息。

单个 Debian 源码包可能根据 **debian/control** 文件定义的内容产生多个 Debian 二进制软件包。

使用“**3.0 (quilt)**”格式的非本土 Debian 软件包是最普通的 Debian 源码包格式。

注意

有许多封装脚本可用。合理使用它们可以帮助您理顺工作流程，但是请确保您能理解它们内部的基本工作原理。

5.1 打包工作流

创建 Debian 二进制软件包的 Debian 打包工作流涉及创建数个特定名称的文件（参见第 5.2 节），与“Debian 政策手册”的定义保持一致。

一个极其简化的 Debian 打包工作流可以概括为以下五步。

1. 下载上游源码压缩包（tarball）并命名为 *package-version*.**tar.gz** 文件。
2. 使上游提供的源码压缩包解压缩后的所有文件存储在 *package-version*/ 目录中。
3. 上游的源码压缩包被复制（或符号链接）至一个特定的文件名 *packagename_version*.**orig.tar.gz**。
   - 分隔 *package* 和 *version* 的符号从 **-**（连字符）更改为 **_**（下划线）
   - 文件扩展名添加了 **.orig** 部分。
4. Debian 软件包规范文件将被添加至上游源代码中，存放在 *package-version*/**debian/** 目录下。
   - **debian/*** 目录下的必需技术说明文件：

     debian/rules 构建 Debian 软件包所需的可执行脚本（参见第 5.4 节）

     debian/control 软件包配置文件包含了源码包名称、源码构建依赖、二进制软件包名称、二进制软件包依赖，等等。（参见第 5.5 节）

debian/changelog Debian 软件包历史文件，其中第一行定义了上游软件包版本号和 Debian 修订版本号（参见第 5.6 节）

debian/copyright 版权和许可证摘要信息（参看第 5.7 节）

- 在 **debian/*** 下的可选配置文件（参见第 5.11 节）：
- 在 *package-version/* 目录中调用 **debmake** 命令将会提供这些配置文件的一套模板。
  - 必备的配置文件总会生成，无论是否提供 **-x0** 选项。
  - **debmake** 命令永远不会覆写任何已经存在的配置文件。
- 这些文件必须手工编辑以达到理想状态。请使用“Debian 政策手册”和“Debian 开发者参考”作为编辑依据。

5. **dpkg-buildpackage** 命令（通常由它的封装命令 **debuild** 或 **pdebuild** 所使用）会在 *package-version/* 目录中被调用，进而以调用 **debian/rules** 脚本的方式制作 Debian 源码包和二进制软件包。
   - 当前工作目录会被设为：**$(CURDIR)**=*/path/to/package-version/*
   - 使用 **dpkg-source**(1) 以“**3.0 (quilt)**”格式创建 Debian 源码包
     - *package_version*.**orig.tar.gz**（*package-version*.**tar.gz** 的副本或符号链接）
     - *package_version-revision*.**debian.tar.xz**（*package-version*/**debian/*** 的 tar 压缩包，即 tarball）
     - *package_version-revision*.**dsc**
   - 使用“**debian/rules build**”构建源代码并安装到 **$(DESTDIR)** 中
     - **DESTDIR=debian**/*binarypackage/*（单二进制包）
     - **DESTDIR=debian/tmp/**（多个二进制包）
   - 使用 **dpkg-deb**(1)、**dpkg-genbuildinfo**(1) 和 **dpkg-genchanges**(1) 创建 Debian 二进制软件包。
     - *binarypackage_version-revision_arch*.**deb**
     - ……（可能有多个 Debian 二进制包文件。）
     - *package_version-revision_arch*.**changes**
6. 使用 **lintian** 命令检查 Debian 软件包的质量。（推荐）
   - 遵守 ftp-master 的拒绝（rejection）指导方针。
     - 软件包被拒绝常见问题解答（REJECT-FAQ）
     - 新软件包（NEW）检查清单
     - Lintian 自动拒绝（autoreject）（lintian 标签列表）
7. 使用 **debsign** 命令，用您的 GPG 私钥为 *package_version-revision*.**dsc** 和 *package_version-revision_arch*.**changes** 文件进行签名。
8. 使用 **dput** 命令向 Debian 仓库上传一套 Debian 源码包和二进制软件包文件。

这里，请将文件名中对应的部分使用下面的方式进行替换：

- 将 *package* 部分替换为 Debian 源码包名称
- 将 *binarypackage* 部分替换为 Debian 二进制软件包名称
- 将 *version* 部分替换为上游版本号
- 将 *revision* 部分替换为 Debian 修订号
- 将 *arch* 部分替换为软件包对应架构

提示

有很多种通过实践摸索而得到的补丁管理方法和版本控制系统的使用策略与技巧。您没有必要将它们全部用上。

提示

在“Debian 开发者参考”一文的 第 6 章最佳打包实践 部分有十分详尽的相关文档。请读一读这些内容。

5.1.1 debhelper 软件包

尽管 Debian 软件包可以仅由编写 **debian/rules** 脚本而不使用 **debhelper** 软件包来生成，其实这样做是不切实际的。现代的 Debian “政策” 对许多功能特性的实现做了要求，如应用适当的文件权限、使用合适的与硬件架构相关的软件库安装路径、安装脚本钩子的插入、调试符号软件包的生成、软件包依赖信息的生成、软件包信息文件的生成、对时间戳调节以符合可重现构建的要求，等等。

Debhelper 软件包提供了一套实用脚本，用来简化 Debian 打包工作流并减轻软件包维护者的负担。若能适当运用，它们可以帮助打包者自动地处理并实现 Debian” 政策 “所要求的功能。

现代化的 Debian 打包工作流可以组织成一个简单的模块化工作流，如下所示：

- 使用 **dh** 命令以自动调用来自 **debhelper** 软件包的许多实用脚本，以及
- 使用 **debian/** 目录下的声明式配置文件配置它们的行为。

您几乎总是应当将 **debhelper** 列为您的软件包的构建依赖之一。本文档在接下来的内容中也假设您正在使用一个版本足够新的 **debhelper** 协助进行打包工作。

5.2 软件包名称和版本

如果所获取上游源代码的形式为 **hello-0.9.12.tar.gz**，您可以将 **hello** 作为上游源代码名称，并将 **0.9.12** 作为上游版本号。

debmake 的目的是为软件包维护者提供开始工作的模板文件。注释行以 **#** 开始，其中包含一些教程性文字。您在将软件包上传至 Debian 仓库之前必须删除或者修改这样的注释行。

许可证信息的提取和赋值过程应用了大量启发式操作，因此在某些情况下可能不会正常工作。强烈建议您搭配使用其它工具，例如来自 **devscripts** 软件包的 **licensecheck** 工具，以配合 **debmake** 的使用。

组成 Debian 软件包名称的字符选取存在一定的限制。最明显的限制应当是软件包名称中禁止出现大写字母。这里给出正则表达式形式的规则总结：

- 上游软件包名称（**-p**）：[-+.a-z0-9]{2,}
- 二进制软件包名称（**-b**）：[-+.a-z0-9]{2,}
- 上游版本号（**-u**）：[0-9][-+.:~a-z0-9A-Z]*
- Debian 修订版本（**-r**）：[0-9][+.~a-z0-9A-Z]*

请在 “Debian 政策手册” 的 第 5 章 - Control 文件及其字段 一节中查看其精确定义。

debmake 所假设的打包情景是相对简单的。因此，所有与解释器相关的程序都会默认为 “**Architecture: all**” 的情况。当然，这个假设并非总是成立。

您必须为 Debian 打包工作适当地调整软件包名称和上游版本号。

为了能有效地使用一些流行的工具（如 **aptitude**）管理软件包名称和版本信息，最好能将软件包名称保持在 30 字符以下；版本号和修订号加起来最好能不超过 14 个字符。[1]

为了避免命名冲突，对用户可见的二进制软件包名称不应选择任何常用的单词。

如果上游没有使用像 **2.30.32** 这样正常的版本编号方案，而是使用了诸如 **11Apr29** 这样包含日期、某些代号或者一个版本控制系统散列值等字符串作为版本号的一部分的话，请在上游版本号中将这些部分移除。这些信息可以稍后在 **debian/changelog** 文件中进行记录。如果您需要为软件设计一个版本字符串，可以使用 **YYYYMMDD** 格式，如 **20110429** 的字符串作为上游版本号。这样能保证 **dpkg** 命令在升级时能正确地确定版本的先后关系。如果您想要确保万一上游在未来重新采纳正常版本编号方案，例如 **0.1** 时能够做到顺畅地迁移，可以另行使用 **0~YYMMDD** 的格式，如 **0~110429** 作为上游版本号。

版本字符串可以按如下的方式使用 **dpkg** 命令进行比较。

[1] 对九成以上的软件包来说，软件包名称都不会超过 24 个字符；上游版本号通常不超过 10 个字符，而 Debian 修订版本号也通常不超过 3 个字符。

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

版本比较的规则可以归纳如下：

- 字符串按照起始到末尾的顺序进行比较。
- 字符比数字大。
- 数字按照整数顺序进行比较。
- 字符按照 ASCII 编码的顺序进行比较。

对于某些字符，如句点（.）、加号（+）和波浪号（~），有如下的特殊规则。

```
0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0~rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0
```

有一个稍需注意的情况，即当上游将 **hello-0.9.12-ReleaseCandidate-99.tar.gz** 这样的版本当作预发布版本，而将 **hello-0.9.12.tar.gz** 作为正式版本时。为了确保 Debian 软件包升级能够顺畅进行，您应当修改版本号命名，如将上游源代码压缩包重命名为 **hello-0.9.12~rc99.tar.gz**。

5.3 本土 Debian 软件包

使用 “**3.0 (quilt)**” 格式的非本土 Debian 软件包是最常见最标准的 Debian 源码包格式。根据 **dpkg-source**(1) 的描述，此时的 **debian/source/format** 文件应当包含 “**3.0 (quilt)** 的文字内容。上述的工作流和接下来给出的打包示例都使用了这种格式。

而本土 Debian 软件包是较罕见的一种 Debian 软件包格式。它通常只用于打包仅对 Debian 项目有价值、有意义的软件。因此，该格式的使用通常不被提倡。

小心

在上游 tarball 源码压缩包无法使用其正确名称 *package_version*.**orig.tar.gz** 被 **dpkg-buildpackage** 获取到的时候，会出现意外地构建了本土 Debian 软件包的情况。这是新手常见的一个错误，通常是因构建中错误地在符号链接名称中使用了“**-**”字符而非正确的“_”字符。[译注：此处仍然假设打包的场景是已经获取或形成了名为 **package-version**.**tar.gz** 的上游源码 tarball。Debian 的打包工作很大程度上是以上游源码 tarball 作为基础的，这一点须时刻牢记在心。]

本土 Debian 软件包不对 上游代码和 **Debian** 的修改进行区分，仅包含以下内容：

- *package_version*.**tar.gz**（*package-version*.**tar.gz** 文件的副本或符号链接，包含 **debian/*** 的各个文件。）
- *package_version*.**dsc**

如果您需要手动创建本土 Debian 软件包，可以使用 **dpkg-source**(1) 工具以 “**3.0 (native)**” 格式进行创建。

提示

某些人希望推行对那些即使是仅针对 Debian 编写的那些软件也使用非本土软件包格式的做法。这种做法所需要的不包含 **debian/*** 文件的 tarball 压缩包事先需要手动生成以符合在第 5.1 节中的标准工作流。他们认为使用非本土软件包格式可以方便与下游发行版之间的沟通与交流。

提示

如果使用本土软件包格式，没有必要事先创建 tarball 压缩包。要创建一个本土 Debian 软件包，应当将 **debian/source/format** 文件的内容设置为“**3.0 (native)**”，适当编写 **debian/changelog** 文件使得版本号中不包含 Debian 修订号（例如，**1.0** 而非 **1.0-1**），最后在源码树中调用“**dpkg-source -b .**”命令。这样做便可以自动生成包含源代码的 tarball。

5.4 debian/rules

debian/rules 脚本是用于实际构建 Debian 软件包的可执行脚本。

- **debian/rules** 脚本重新封装了上游的构建系统（参见第 5.16 节）以达到将文件安装至 **$(DESTDIR)** 并将生成的文件存入各个 **deb** 格式文件中的目的。
  - 这里的 **deb** 文件用于二进制的文件分发，并将被 **dpkg** 命令所使用以将软件安装至系统中。
- **dh** 命令通常在 **debian/rules** 脚本中使用，用作构建系统的一个前端。
- **$(DESTDIR)** 路径具体值依赖于构建的类型。
  - **$(DESTDIR)=debian/***binarypackage*（单个二进制软件包）
  - **$(DESTDIR)=debian/tmp**（多个二进制软件包）

5.4.1 dh

由 **debhelper** 软件包提供的 **dh** 命令与一些相关的软件包共同工作，作为典型的上游构建系统的一层封装，同时它支持所有 Debian 政策（Debian Policy）规定必须在 **debian/rules** 实现的目标（target），以此提供一个统一的访问接口。

- **dh clean**：清理源码树中的文件。
- **dh build**：在源码树中进行构建
- **dh build-arch**：在源码树中构建架构相关的软件包
- **dh build-indep**：在源代码中构建架构无关的软件包
- **dh install**：将二进制文件安装至 **$(DESTDIR)**
- **dh install-arch**：为架构相关的软件包将二进制文件安装至 **$(DESTDIR)** 中
- **dh install-indep**：为架构无关的软件包将二进制文件安装进入 **$(DESTDIR)** 中
- **dh binary**：产生 **deb** 文件
- **dh binary-arch**：为架构相关的软件包产生 **deb** 文件
- **dh binary-indep**：为架构无关的软件包产生 **deb** 文件

注意

对使用了 **debhelper**“compat >=9”的情况，**dh** 命令将在编译参数未事先设置的情况下根据 **dpkg-buildflags** 命令返回的值设置并导出各个编译参数（如 **CFLAGS**、**CXXFLAGS**、**FFLAGS**、**CPPFLAGS** 和 **LDFLAGS**）。（**dh** 命令将调用在 **Debian::Debhelper::Dh_Lib** 模块中定义的 **set_buildflags**。）

5.4.2 简单的 debian/rules

受益于 **dh** 命令对构建目标的抽象化[2]，一个符合 Debian 政策而支持所有必需目标（target）的 **debian/rules** 文件可以简单地写成如下形式[3]：

简单的 **debian/rules**：

```
#!/usr/bin/make -f
#export DH_VERBOSE = 1

%:
        dh $@
```

从本质上来看，这里的 **dh** 命令的作用是作为一个序列化工具，在合适的时候调用所有所需的 **dh_*** 命令。

注意

debmake 命令会在 **debian/control** 文件中写入“**Build-Depends: debhelper (>=9)**”，并在 **debian/compat** 文件中写入“**9**”。

提示

设置“**export DH_VERBOSE = 1**”会输出构建系统中每一条会修改文件内容的命令。它同时会在某些构建系统中启用详细输出构建日志的选项。

5.4.3 自定义 debian/rules

通过添加合适的 **override_dh_*** 目标（target）并编写对应的规则，可以实现对 **debian/rules** 脚本的灵活定制。

如果需要在 **dh** 命令调用某些特定的 **dh_*foo*** 命令时采取某些特别的操作，则任何自动执行的操作均可以被 **debian/rules** 中额外添加的 **override_dh_*foo*** 这样的 Makefile 目标所覆写。

构建的过程可以使用某些上游提供的接口进行定制化，如使用传递给标准的源代码构建系统的参数。这些构建系统包括但不限于：

- **configure**，
- **Makefile**，
- **setup.py**，或
- **Build.PL**。

在这种情况下，您应该添加一个 **override_dh_auto_build** 目标并在其中执行“**dh_auto_build** -- 自定义参数”的命令。这样可以在 **dh_auto_build** 默认传递的参数之后确保将用户给出的 自定义参数继续传递给那些构建系统。

提示

如果上文提到的构建系统命令已知得到了 **dh_auto_build** 命令的支持的话，请避免直接调用这些命令（而让 **dh_auto_build** 自动处理）。

[2] 这个简化形式在 **debhelper** 软件包第七版或更新的版本中可用。本指南内容假设您在使用 **debhelper** 第九版或更新的版本。

[3] **debmake** 命令会产生稍微复杂一些的 **debian/rules** 文件。虽然如此，其核心结构没有什么变化。

debmake 命令所创建的初始模版文件除了应用了上文提到的简单 **debian/rules** 文件的优点外，同时为后续可能涉及的软件包加固等情景添加了一些额外的定制选项。您需要先了解整个构建系统背后的工作原理（参见第 5.16 节），之后才能收放自如地定制软件包来处理某些非常规的工作情况。

- 请参考第 4.6 节一节以了解如何对 **debmake** 命令生成的 **debian/rules** 文件模版进行定制。
- 请参见第 5.20 节以了解与 multiarch 相关的定制方法。
- 请参见第 5.21 节以了解与软件包加固相关的定制方法。

5.4.4 debian/rules 中的变量

某些对自定义 **debian/rules** 有用的变量定义可以在 **/usr/share/dpkg/** 目录下的文件中找到。比较重要的包括：

pkg-info.mk **DEB_SOURCE**、**DEB_VERSION**、**DEB_VERSION_EPOCH_UPSTREAM**、**DEB_VERSION_UPSTREA**
DEB_VERSION_UPSTREAM 和 **DEB_DISTRIBUTION** 变量。它们在向后移植（backport）支持等场景下能起到一定的作用。

vendor.mk **DEB_VENDOR** 和 **DEB_PARENT_VENDOR** 变量，以及 **dpkg_vendor_derives_from** 宏。它们在系统提供方的支持方面（Debian、Ubuntu 等）有其特定用处。

architecture.mk 设置 **DEB_HOST_*** 和 **DEB_BUILD_*** 变量。除此之外存在一种替代方案，即直接调用 **dpkg-architecture** 来获取变量，一次调用查询得到一个变量值。如显式调用 **dpkg-architecture** 以获取必需变量的话，便不再需要在 **debian/rules** 中包含 **architecture.mk** 了（后者会引入全部架构相关的变量）。

buildflags.mk 设置 **CFLAGS**、**CPPFLAGS**、**CXXFLAGS**、**OBJCFLAGS**、**OBJCXXFLAGS**、**GCJFLAGS**、**FFLAGS**、**FCFLAGS** 和 **LDFLAGS** 这些构建标志（build flags）。

如果您希望在 **debian/rules** 中使用其中的某些变量，您可以将相关的代码复制到 **debian/rules** 文件中，或是重写一份简单的替代实现。总而言之请保持 **debian/rules** 文件尽量简单。

例如，您按如下的方法在 **debian/rules** 文件中添加内容，从而为 **linux-any** 目标架构添加额外的 **CONFIGURE_FLAGS**：

```
DEB_HOST_ARCH_OS ?= $(shell dpkg-architecture -qDEB_HOST_ARCH_OS)
 ...
ifeq ($(DEB_HOST_ARCH_OS),linux)
CONFIGURE_FLAGS += --enable-wayland
endif
```

提示

历史上对于 **debhelper** 兼容等级小于等于 8 的情况下，在 **debian/rules** 文件中包含 **buildflags.mk** 文件是很有用的，它可以合适地设置一些构建标志，如 **CPPFLAGS**、**CFLAGS**、**LDFLAGS** 等，同时保证对特定选项，如 **DEB_CFLAGS_MAINT_APPEND** 和 **DEB_BUILD_MAINT_OPTIONS** 的合适处理。现在您应当使用的 **debhelper** 兼容等级大于等于 9，故如无特殊原因，请不要继续包含 **buildflags.mk**，请交由 **dh** 命令来处理和设置这些构建标志。

参见第 5.20 节、**dpkg-architecture**(1) 和 **dpkg-buildflags**(1)。

5.4.5 可重现的构建

为了做到软件包可重现的构建，这里给出一些相关的建议。

- 不要嵌入基于系统时间的时间戳。
- 在 **debian/rules** 中使用“**dh $@**”以应用最新的 **debhelper** 特性。
- Export the build environment as “**LC_ALL=C.UTF-8**” (see 第 7.15 节).

- 对上游源代码中使用的时间戳，使用 debhelper 提供的环境变量 **$SOURCE_DATE_EPOCH** 的值。
- 阅读可重现构建了解更多信息。
  - 可重现构建操作方法。
  - 可重现构建时间戳处理提议。

由 **dpkg-genbuildinfo**(1) 生成的控制文件 *source-name_source-version_arch*.**buildinfo** 记录了构建环境信息。参见 **deb-buildinfo**(5)

5.5 debian/control

debian/control 文件包含了由空行分隔的数块元信息数据。每块元数据按照如下的顺序定义了下面这些内容：

- Debian 源码包的元信息数据
- Debian 二进制软件包的元信息

参见“Debian 政策手册”中的 第 5 章 - Control 文件及其字段 一章以了解每块元信息的具体定义。

5.5.1 Debian 二进制软件包的拆分

对行为良好的构建系统来说，对 Debian 二进制包的拆分可以由如下方式实现。

- 为所有二进制软件包在 **debian/control** 文件中创建对应的二进制软件包条目。
- 在对应的 **debian/**二进制软件包名**.install** 文件中列出所有文件的路径（相对于 **debian/tmp** 目录）。

请查看本指南中相关的例子：

- 第 8.11 节（基于 Autotools）
- 第 8.12 节（基于 CMake）

5.5.1.1 debmake -b

debmake 命令的 **-b** 选项提供了一个符合直觉又灵活的功能，可以用来创建 **debian/control** 的初始模板文件，其中可以定义多个 Debian 二进制软件包，每节中含有如下字段：

- **Package:**
- **Architecture:**
- **Multi-Arch:**
- **Depends:**
- **Pre-Depends:**

debmake 命令也会在每个适当的依赖字段中设置合适的变量替换占位符（substvars）。
我们在这里直接引用 **debmake** 手册页中的相关一部分内容。

-b ” 二进制软件包名 ***[:type],***⋯**”, --binaryspec ”** 二进制软件包名 ***[:type],***⋯**”** set the binary package specs by a comma separated list of *binarypackage:type* pairs, e.g., in the full form “**foo:bin,foo-doc:doc,libfoo1:lib,libfoo-dev:dev**” or in the short form, “**-doc,libfoo1,libfoo-dev**” .

这里，二进制软件包是二进制软件包名称，可选的类型应当从下面的类型值中进行选取：

- **bin**：C/C++ 预编译 ELF 二进制代码软件包（any，foreign）（默认，别名：**””**，即，空字符串）
- **data**：数据（字体、图像、⋯⋯）软件包（all，foreign）（别名：**da**）
- **dev**：库开发软件包（any，same）（别名：**de**）
- **doc**：文档软件包（all，foreign）（别名：**do**）
- **lib**：库软件包（any，same）（别名：**l**）

- **perl**：Perl 脚本软件包（all，foreign）（别名：**pl**）
- **python**：Python 脚本软件包（all，foreign）（别名：**py**）
- **python3**：Python3 脚本软件包（all，foreign）（别名：**py3**）
- **ruby**：Ruby 脚本软件包（all，foreign）（别名：**rb**）
- **script**：Shell 脚本软件包（all，foreign）（别名：**sh**）

括号内成对的值，例如（any，foreign），是软件包的架构和多架构（**Multi-Arch**）特性的值，它们将设置在 **debian/control** 文件中。

大多数情况下，**debmake** 命令可以有效地从二进制软件包的名称猜测出正确的类型。如果类型的值并不明显，程序将回退到将类型设置为 **bin**。例如，**libfoo** 设置类型为 **lib**，而 **font-bar** 会令程序设置类型为 **data**，……

如果源码树的内容和类型的设置不一致，**debmake** 命令会发出警告。

5.5.1.2 拆包的场景和例子

对于下面这样的上游源代码示例，我们在这里给出使用 **debmake** 处理时一些典型的 multiarch 软件包拆分的场景和做法:

- 一个软件库源码 *libfoo-1.0*.**tar.gz**
- 一个软件工具源码 *bar-1.0*.**tar.gz**，软件由编译型语言编写
- 一个软件工具源码 *baz-1.0*.**tar.gz**，软件由解释型语言编写

二进制软件包	类型	**Architecture:**	**Multi-Arch:**	软件包内容
lib*foo1*	**lib***	any	same	共享库，可共同安装
lib*foo*-**dev**	**dev***	any	same	共享库头文件及相关开发文件，可共同安装
lib*foo*-**tools**	**bin***	any	foreign	运行时支持程序，不可共同安装
lib*foo*-**doc**	**doc***	all	foreign	共享库文档
bar	**bin***	any	foreign	编译好的程序文件，不可共同安装
bar-**doc**	**doc***	all	foreign	程序的配套文档文件
baz	**script**	all	foreign	解释型程序文件

5.5.1.3 库软件包名称

Let’s consider that the upstream source tarball of the **lib***foo* library is updated from **lib***foo-7.0*.**tar.gz** to **lib***foo-8.0*.**tar.gz** with a new SONAME major version which affects other packages.

库的二进制软件包将必须从 **lib***foo7* 重命名为 **lib***foo8* 以保持使用 **unstable** 套件的系统上所有依赖该库的软件包在上传了基于 **lib***foo-8.0*.**tar.gz** 的新库后仍然能够正常运行。

警告

如果这个二进制库软件包没有得到更名，许多使用 **unstable** 套件的系统上的各个依赖该库的软件包会在新的库包上传后立刻破损，即便立刻请求进行 binNMU 上传也无法避免这个问题。由于种种原因，binNMU 不可能在上传后立刻开始进行，故无法缓解问题。

-dev 软件包必须遵循以下命名规则：

- 使用不带版本号的 **-dev** 软件包名称：**lib***foo*-**dev**
  - 该情况通常适用于依赖关系处于叶节点的库软件包。
  - 仓库内只允许存在一个版本的库源码包。
    - 其相关联的库软件包在库变迁进行时需要从 **lib***foo7* 重命名为 **lib***foo8* 以避免 **unstable** 仓库内依赖关系的破坏。

  - 该方法适用于简单 binNMU 可以快速解决所有受影响软件包对该库依赖的情况下。（ABI 有变化，过时的 API 被移除而常用、活跃的 API 未变化。）
  - 该方法有时也能适用于可协调进行手动的代码更新，影响范围限定在有限的一些软件包中的情况下。（API 有变化）

- 使用带版本的 **-dev** 软件包名称：**lib***foo7*-**dev** 和 **lib***foo8*-**dev**
  - 该情况通常适用于各类重要库软件包。
  - 两个版本的库源码包可同时出现在发行版仓库中。
    * 令所有依赖该库的软件包依赖 **lib***foo*-**dev**。
    * 令 **lib***foo7*-**dev** 和 **lib***foo8*-**dev** 两者都提供 **lib***foo*-**dev**。
    * 源码包需要从 **lib***foo-?.0*.**tar.gz** 相应地重命名为 **lib***foo7-7.0*.**tar.gz** 和 **lib***foo8-8.0*.**tar.gz**。
    * 需要仔细选择 **lib***foo7* 和 **lib***foo8* 软件包文件安装时的路径，如头文件等等，以保证它们可以同时安装。
  - 可能的话，不要使用这个重量级的、需要大量人为干预的方法。
  - 该方法适用于存在多个依赖该库的软件包，且升级时常常涉及手动代码更新的场景。（API 有变化）否则，受影响的软件包会无法从源码重新构建并导致对发行而言致命的 bug 出现。

提示

如果包内数据文件编码方案有所变化（如，从 latin1 变为 utf-8），该场景应比照 API 变化做类似的考虑与处理。

参见第 5.18 节。

5.5.2 Substvar

debian/control 也定义了软件包的依赖关系，其中变量替换机制（substvar）的功能可以用来将软件包维护者从跟踪（大多数简单的）软件包依赖的重复劳动中解放出来。请参见 **deb-substvars**(5)。

debmake 命令支持下列变量替换指令：

- **${misc:Depends}**，可用于所有二进制软件包
- **${misc:Pre-Depends}**，可用于所有 multiarch 软件包
- **${shlibs:Depends}**，可用于所有含有二进制可执行文件或库的软件包
- **${python:Depends}**，可用于所有 Python 软件包
- **${python3:Depends}**，可用于所有 Python3 软件包
- **${perl:Depends}**，用于所有 Perl 软件包
- **${ruby:Depends}**，用于所有 Ruby 软件包

For the shared library, required libraries found simply by "**objdump -p** */path/to/program* **| grep NEEDED**" are covered by the **shlib** substvar.

对于 Python 和其它解释器来说，所需的模块通常由对包含类似“**import**”、“**use**”、“**require**”等等关键字的行进行解析，并会体现在各自对应的变量替换占位符所在位置上。

对其它没有部署属于自己范畴内的变量替换机制的情况，**misc** 变量替换占位符通常用来覆盖对应的依赖替换需求。

对 POSIX shell 程序来说，并没有简单的办法来验证其依赖关系，substvar 的变量替换也无法自动得出它们的依赖。

对使用动态加载机制，包括 GObject introspection 机制的库和模块来说，现在没有简单的方法可以检查依赖关系，变量替换机制也无法自动推导出所需的依赖。

5.5.3 binNMU 安全

一个 binNMU（二进制非维护者上传） 是为库迁移或其它目的所作的非维护者软件包上传。在一次 binNMU 上传中，只有“**Architecture: any**”的软件包会被重构建，其版本号会在末尾附加一个编号（例如，原来版本为 2.3.4-3，新上传的包版本会变成 2.3.4-3+b1）。所有“**Architecture: all**”的包将不会重新构建。

来自同一个源码包的各个二进制包如果在 **debian/control** 文件中有互相的依赖关系，这些二进制包通常情况下应当对 binNMU 是安全的（即，进行 binNMU 不会破坏依赖关系）。然而，在“**Architecture: any**”和“**Architecture: all**”的软件包同时由同一源码包产出，且互相之间有依赖关系时，需要小心对待所依赖的版本，必要时应做出调整。

- “**Architecture: any**”的软件包依赖于“**Architecture: any**” *foo* 软件包
  - **Depends:** *foo* **(= ${binary:Version})**
- “**Architecture: any**”的软件包依赖于“**Architecture: all**” *bar* 软件包
  - **Depends:** *bar* **(= ${source:Version}**
- “**Architecture: all**”的软件包依赖于“**Architecture: any**” *baz* 软件包
  - **Depends:** *baz* **(>= ${source:Version})**, *baz* **(<< ${source:Upstream-Version}.0~)**

5.6 debian/changelog

debian/changelog 文件记录了 Debian 软件包的历史并在其第一行定义了上游软件包的版本和 Debian 修订版本。所有改变的内容应当以明确、正式而简明的语言风格进行记录。

- 即便您在自己独立进行软件包上传，您也必须记录所有较重要、用户可见的变更，例如：
  - 安全相关的漏洞修复。
  - 用户界面变动。
- 如果您需要他人协助您进行上传，您应当更详尽地记录变更内容，包括所有打包相关的变动，从而方便他人对您的软件包进行审查。
  - 协助上传的人员不应该也通常不会猜测您没有写出来的想法，所以请认真书写变更信息。
  - 通常来说，协助您上传的人的时间比您的时间更宝贵。

The **debmake** command creates the initial template file with the upstream package version and the Debian revision. The distribution is set to **UNRELEASED** to prevent accidental upload to the Debian archive.

通常使用 **debchange** 命令（它具有一个别名，即 **dch**）对其进行编辑。

提示

您也可以手动使用任何文本编辑器修改 **debian/changelog** 文件，只要您能够遵循 **debchange** 命令所使用的特定文本排版格式即可。

提示

debian/changelog 文件使用的日期字符串可以使用“**LC_ALL=C date -R**”命令手动生成。

This is installed in the **/usr/share/doc/***binarypackage* directory as **changelog.Debian.gz** by the **dh_installchangelogs** command.

上游的变更日志则会安装至 **/usr/share/doc/***binarypackage* 目录中，文件名为 **changelog.gz**。

The upstream changelog is automatically found by the **dh_installchangelogs** using the case insensitive match of its file name to **changelog**, **changes**, **changelog.txt**, **changes.txt**, **history**, **history.txt**, or **changelog.md** and searched in the **./ doc/** or **docs/** directories.

当您完成了主要打包工作并验证了其质量之后，请考虑运行“**dch -r**”命令并将最终完成的 **debian/changelog** 文件中发行版（distribution）部分进行设置，通常该字段应当使用 **unstable**。[4] 如果您的打包是一次向后移植（backports）、是安全更新或是对长期支持版的上传等等其它情况，请使用相应合适的发行版名称。

5.7 debian/copyright

Debian 以十分严肃的态度对待版权和许可证信息。“Debian 政策手册”强制规定软件包的 **debian/copyright** 文件中需要提供相关信息的摘要。

您应当按照 机器可解析的 debian/copyright 文件（DEP-5）对其进行排版。

小心

这里的 **debian/copyright** 文件中描述的许可证信息匹配信息应当合适地进行排序，以确保越宽泛的文件匹配越靠前。请参见第 6.4 节。

debmake 命令会以扫描整个源码树的方式创建初步的、兼容 DEP-5 的模板文件。它会内部调用许可证检查工具来对许可证文本进行分类。[5]

除非明确指定（有些严格过头的）**-P** 选项，**debmake** 命令会为了实用性而跳过对自动生成的文件的检查与报告，默认它们采用宽松的许可证。

注意

如果您发现了这个许可证检查工具存在一些问题，请向 **debmake** 软件包提交缺陷报告并提供包含出现问题的许可证和版权信息在内的相关文本内容。

注意

debmake 命令专注于在创建 **debian/copyright** 模板时聚合相同的授权和许可证信息并收录其详细内容。为了在有限的时间内尽可能完成工作，工具将只会提取文件中第一块看起来像授权文本或许可证声明的部分。因此，生成的许可证信息可能并不完美。请同时考虑使用其它工具，如 **licensecheck** 辅助进行检查。

提示

我们强烈推荐您使用 **licensecheck**(1) 命令再次检查源码许可证的状态，并在有必要的情况下进行人工代码审查。

[4] 如果您在使用 **vim** 编辑器，请确保使用“**:wq**”命令对内容进行保存。

[5] 程序以前会在内部调用来自 **devscripts** 软件包的 **licensecheck** 命令来进行检查。现在的 **licensecheck** 命令由独立的 **licensecheck** 软件包所提供，相较从前的实现也有了较大的改进。

5.8 debian/patches/*

在构建过程开始之前，**debian/patches/** 目录内的 **-p1** 等级的补丁将会按照在 **debian/patches/series** 文件中指定的顺序依次应用于上游代码树中。

注意

本土 Debian 软件包（参见第 5.3 节）将不使用这些文件。

要准备这一系列 **-p1** 等级的补丁，有几种不同的方式可供您选择。

- **diff** 命令
  - 参见第 4.8.1 节
  - 原始但万能的方法
    - 补丁的来源多种多样，它可以来自其它发行版、邮件列表中的帖文或是来自上游 **git** 仓库的拣选补丁，由“**git format-patches**”生成
  - 不涉及 **.pc/** 目录的问题
  - 不修改上游源代码树
  - 手工更新 **debian/patches/series** 文件
- **dquilt** 命令
  - 参见第 3.4 节
  - 基本的便利方案
  - 会以合适方式生成 **.pc/** 目录及其中的数据
  - 会修改上游源代码树
- “**dpkg-source --commit**”命令
  - 参见第 4.8.3 节
  - 更新、更优雅一些的方案
  - 会以合适方式生成 **.pc/** 目录及其中的数据
  - 会修改上游源代码树
- 由 **dpkg-buildpackage** 自动生成补丁
  - 参见第 5.14 节
  - 在 **debian/source/local-options** 文件中添加 **single-debian-patch** 这一行
  - 设置 **debian/source/local-patch-header** 文件
  - 不涉及 **.pc/** 目录的问题
  - 在 Debian 分支中（常见为 **master** 分支）存储经过修改的上游源代码树
- **gbp-pq** 命令
  - 配合 **git-buildpackage** 工具的基本 **git** 工作流
  - 不涉及 **.pc/** 目录的问题
  - 在可丢弃分支上保存经过修改的上游源码树（**patch-queue/master**）
  - 在 Debian 分支中（常见为 **master** 分支）存储未经修改的源码树
- **gbp-dpm** 命令
  - 配合 **git-dpm** 软件包的更细致的 **git** 工作流
  - 不涉及 **.pc/** 目录的问题

- 在补丁分支中（通常命名为 **patched**/随便啥名字）存储经过修改的上游源码树
- 在 Debian 分支中（通常命名为 **master**/随便啥名字）存储未经修改的上游源码树

无论这些补丁的来源如何，都建议使用兼容于 DEP-3 的头部信息对其进行标记。

提示

dgit 软件包提供了另外一套直接使用 git 集成操作 Debian 软件包仓库的工具。

5.8.1 dpkg-source -x

“**dpkg-source -x**” 命令可以对 Debian 源码包进行解压缩。

该命令通常会将 **debian/patches/** 目录内的补丁应用在源码树中，并将补丁状态记录在 **.pc/** 目录中。

如果您想保持源码树不做修改（例如，为了在第 5.13 节中继续使用），请在命令行中使用 **--skip-patches** 选项。

5.8.2 dquilt 和 dpkg-source

The **quilt** command (or its wrapped **dquilt** command) was needed to manage the **-p1** patches in the **debian/patches/** directory before the **--commit** feature was added to the **dpkg-source** command in 1.16.1.

在使用 **dpkg-source** 命令时，补丁应当能够干净地进行应用。因此在补丁行数出现偏移或者其它情况出现时，您不能直接将旧补丁原封不动地复制到新版上游发布对应打包版本的目录中。

与此相对的是 **dquilt** 命令（参见第 3.4 节）对补丁的处理更加宽容。您可以调用 **dquilt** 命令对补丁进行正常化。

```
$ while dquilt push; do dquilt refresh ; done
$ dquilt pop -a
```

使用 **dpkg-source** 命令比起使用 **dquilt** 命令来说存在一大优势：**dquilt** 命令无法自动处理二进制文件出现变更的情况，而 **dpkg-source** 命令能够探测出现内容变动的二进制文件，并将其列入 **debian/source/include-binaries** 文件以便在 Debian 打包用压缩包中将文件囊括其中。

5.9 debian/upstream/signing-key.asc

某些软件包由 GPG 密钥进行了签名。

例如，GNU hello 可使用 HTTP 协议从 https://ftp.gnu.org/gnu/hello/ 下载。它含有以下文件：

- **hello**-*version*.**tar.gz**（上游源代码）
- **hello**-*version*.**tar.gz.sig**（分离的签名）nature)

我们现在来选择最新的版本套装。

```
$ wget https://ftp.gnu.org/gnu/hello/hello-2.9.tar.gz
 ...
$ wget https://ftp.gnu.org/gnu/hello/hello-2.9.tar.gz.sig
 ...
$ gpg --verify hello-2.9.tar.gz.sig
gpg: Signature made Thu 10 Oct 2013 08:49:23 AM JST using DSA key ID 80EE4A00
gpg: Can't check signature: public key not found
```

如果您从邮件列表获知上游维护者所使用的 GPG 公钥信息，请将它作为 **debian/upstream/signing-key.asc** 文件进行存储。否则，请使用 hkp 公钥服务器并经由您的信任网进行验证。

```
$ gpg --keyserver hkp://keys.gnupg.net --recv-key 80EE4A00
gpg: requesting key 80EE4A00 from hkp server keys.gnupg.net
gpg: key 80EE4A00: public key "Reuben Thomas <rrt@sc3d.org>" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
```

```
gpg:               imported: 1
$ gpg --verify hello-2.9.tar.gz.sig
gpg: Signature made Thu 10 Oct 2013 08:49:23 AM JST using DSA key ID 80EE4A00
gpg: Good signature from "Reuben Thomas <rrt@sc3d.org>"
  ...
Primary key fingerprint: 9297 8852 A62F A5E2 85B2  A174 6808 9F73 80EE 4A00
```

提示

如果您的网络环境阻挡了对 HKP **11371** 端口的访问，请考虑使用“**hkp://keyserver.ubuntu.com:80**”。

在确认密钥身份 **80EE4A00** 值得信任之后，应当下载其公钥并将其保存在 **debian/upstream/signing-key.asc** 文件中。

```
$ gpg --armor --export 80EE4A00 >debian/upstream/signing-key.asc
```

之后，应相应地在 **debian/watch** 文件中做如下的修改。

```
version=4
pgpsigurlmangle=s/$/.sig/  https://ftp.gnu.org/gnu/hello/ hello-(\d[\d.]*)\.tar ↩
    \.(?:gz|bz2|xz)
```

现在 **uscan** 命令会在扫描时自动使用 GPG 签名验证上游源码包的真实性。

5.10 debian/watch 和 DFSG

Debian 严肃地对待软件自由，遵循 Debian 自由软件指导方针（DFSG）。

在使用 **uscan** 命令来更新 Debian 打包所用代码时，上游源码包（tarball）中不符合Debian 自由软件指导方针（DFSG）的部分可以利用该工具简单地进行移除。

- 在 **debian/copyright** 文件中的 **Files-Excluded** 一节中列出需要移除的文件。
- 在 **debian/watch** 文件中列出下载上游源码包（tarball）所使用的 URL。
- 运行 **uscan** 命令以下载新的上游源码包（tarball）。
  - 作为替代方案，您也可以使用“**gbp import-orig --uscan --pristine-tar**”命令。
- 最后得到 tarball 的版本编号会附加一个额外的后缀 **+dfsg**。

5.11 其它 debian/* 文件

另外也可以添加一些可选的配置文件并放入 **debian/** 目录。它们大多用于控制由 **debhelper** 软件包提供的 **dh_*** 命令的行为，但也有一些文件会影响 **dpkg-source**、**lintian** 和 **gbp** 这些命令。

提示

请检查 **debhelper**(7) 的内容以了解当前可用的 **dh_*** 命令列表。

这些 **debian/***binarypackage***.*** 的文件提供了设置文件安装路径的强大功能。即使上游源代码没有构建系统，这个软件依然可以利用这里提到的这些文件来进行打包。请参考第 8.2 节的示例。

The ”**-x[1234]**” superscript notation that appears in the following list indicates the minimum value for the **debmake -x** option that will generate the associated template file. See 第 6.6 节 or **debmake**(1) for details.

下面按照字母表顺序列出一些值得注意的可选配置文件。

binarypackage*.bug-control** -x3 将安装至 *binarypackage* 软件包的 **usr/share/bug/binarypackage*/**control** 位置。另请参考第 5.24 节。

binarypackage*.bug-presubj** -x3 将安装至 *binarypackage* 软件包的 **usr/share/bug/binarypackage*/**presubj** 位置。另请参考第 5.24 节。

binarypackage*.bug-script** -x3 将安装至 *binarypackage* 软件包的 **usr/share/bug/binarypackage* or **usr/share/bug/***binarypacka* 位置。另请参考第 5.24 节。

***binarypackage*.bash-completion** 列出需要安装的 `bash` 补全脚本。

需要在构建环境和用户环境内均安装 `bash-completion` 软件包。

另请参考 **dh_bash-completion**(1)。

clean -x2 列出（构建前）未被 **dh_auto_clean** 命令清理，且需要手工清理的文件。

另请参考 **dh_auto_clean**(1) 和 **dh_clean**(1)。

compat -x1 Set the **debhelper** compatibility level (currently, "**9**").

See "COMPATIBILITY LEVELS" in **debhelper**(8).

***binarypackage*.conffile** No need for this file under "compat >= 3" since all files in the **etc/** directory are conffiles.

If the program you're packaging requires every user to modify the configuration files in the **/etc** directory, there are two popular ways to arrange for them not to be conffiles, keeping the **dpkg** command happy and quiet.

- Create a symlink under the **/etc** directory pointing to a file under the **/var** directory generated by the maintainer scripts.
- Create a file generated by the maintainer scripts under the **/etc** directory.

See **dh_installdeb**(1).

***binarypackage*.config** This is the **debconf config** script used for asking any questions necessary to configure the package. See 第 5.19 节.

binarypackage*.cron.hourly** -x3 Installed into the **etc/cron/hourly/binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

binarypackage*.cron.daily** -x3 Installed into the **etc/cron/daily/binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

binarypackage*.cron.weekly** -x3 Installed into the **etc/cron/weekly/binarypackage* file in *binarypackage*.

See **dh_installcron**(1) and **cron**(8).

binarypackage*.cron.monthly** -x3 Installed into the **etc/cron/monthly/binarypackage* file in *binarypackage*.

See **dh_installcro***(1) and **cron**(8).

binarypackage*.cron.d** -x3 Installed into the **etc/cron.d/binarypackage* file in *binarypackage*.

See **dh_installcron**(1), **cron**(8), and **crontab**(5).

binarypackage*.default** -x3 If this exists, it is installed into **etc/default/binarypackage* in *binarypackage*.

参见 **dh_installinit**(1)。

***binarypackage*.dirs** -x3 List directories to be created in *binarypackage*.

See **dh_installdirs**(1).

Usually, this is not needed since all **dh_install*** commands create required directories automatically. Use this only when you run into trouble.

***binarypackage*.doc-base** -x2 Installed as the **doc-base** control file in *binarypackage*.

See **dh_installdocs**(1) and Debian doc-base Manual provided by the **doc-base** package.

***binarypackage*.docs** -x2 List documentation files to be installed in *binarypackage*.

参见 **dh_installdocs**(1)。

binarypackage*.emacsen-compat** -x3 Installed into **usr/lib/emacsen-common/packages/compat/binarypackage* in *binarypackage*.

参见 **dh_installemacsen**(1)。

binarypackage*.emacsen-install** -x3 Installed into **usr/lib/emacsen-common/packages/install/binarypackage* in *binarypackage*.

参见 **dh_installemacsen**(1)。

binarypackage*.emacsen-remove** -x3 Installed into **usr/lib/emacsen-common/packages/remove/binarypackage* in *binarypackage*.

参见 **dh_installemacsen**(1)。

binarypackage*.emacsen-startup** -x3 Installed into **usr/lib/emacsen-common/packages/startup/binarypackage* in *binarypackage*.

参见 **dh_installemacsen**(1)。

binarypackage*.examples** -x2 List example files or directories to be installed into **usr/share/doc/binarypackage***/examples/** in *binarypackage*.

See **dh_installexamples**(1).

gbp.conf If this exists, it functions as the configuration file for the **gbp** command.

See **gbp.conf**(5), **gbp**(1), and **git-buildpackage**(1).

***binarypackage*.info** -x2 List info files to be installed in *binarypackage*.

See **dh_installinfo**(1).

binarypackage*.init** -x3 Installed into **etc/init.d/binarypackage* in *binarypackage*.

参见 **dh_installinit**(1)。

***binarypackage*.install** -x2 List files which should be installed but are not installed by the **dh_auto_install** command.

See **dh_install**(1) and **dh_auto_install**(1).

license-examples/* -x4 These are copyright file examples generated by the **debmake** command. Use these as the reference for making the **copyright** file.

Please make sure to erase these files.

***binarypackage*.links** -x2 List pairs of source and destination files to be symlinked. Each pair should be put on its own line, with the source and destination separated by whitespace.

See **dh_link**(1).

binarypackage*.lintian-overrides** -x3 Installed into **usr/share/lintian/overrides/binarypackage* in the package build directory. This file is used to suppress erroneous **lintian** diagnostics.

See **dh_lintian**(1), **lintian**(1) and Lintian User's Manual.

manpage.* -x3 These are manpage template files generated by the **debmake** command. Please rename these to appropriate file names and update their contents.

Debian Policy requires that each program, utility, and function should have an associated manual page included in the same package. Manual pages are written in **nroff**(1).

If you are new to making a manpage, use **manpage.asciidoc** or **manpage.1** as the starting point.

***binarypackage*.manpages** -x2 List man pages to be installed.

See **dh_installman**(1).

***binarypackage*.menu (deprecated, no more installed)** tech-ctte #741573 decided "Debian should use **.desktop** files as appropriate".

Debian menu file installed into **usr/share/menu**/*binarypackage* in *binarypackage*.

See **menufile**(5) for its format. See **dh_installmenu**(1).

NEWS Installed into **usr/share/doc**/*binarypackage*/**NEWS.Debian**.

See **dh_installchangelogs**(1).

patches/* Collection of **-p1** patch files which are applied to the upstream source before building the source.

See **dpkg-source**(1), 第 3.4 节 and 第 4.8 节.

No patch files are generated by the **debmake** command.

patches/series [-x1] The application sequence of the **patches/*** patch files.

***binarypackage*.preinst** [-x2] **, *binarypackage*.postinst** [-x2] **, *binarypackage*.prerm** [-x2] **, *binarypackage*.postrm** [-x2] These maintainer scripts are installed into the **DEBIAN** directory.

Inside the scripts, the token **#DEBHELPER#** is replaced with shell script snippets generated by other **debhelper** commands.

See **dh_installdeb**(1) and Chapter 6 - Package maintainer scripts and installation procedure in the "Debian Policy Manual".

See also **debconf-devel**(7) and 3.9.1 Prompting in maintainer scripts in the "Debian Policy Manual".

README.Debian [-x1] Installed into the first binary package listed in the **debian/control** file as **usr/share/doc**/*binarypackage*/**README.Debian**.

参见 **dh_installdocs**(1)。

This file provides the information specific to the Debian package.

***binarypackage*.service** [-x3] If this exists, it is installed into **lib/systemd/system**/*binarypackage*.**service** in *binarypackage*.

See **dh_systemd_enable**(1), **dh_systemd_start**(1), and **dh_installinit**(1).

source/format [-x1] The Debian package format.

- Use "**3.0 (quilt)**" to make this non-native package (recommended)
- Use "**3.0 (native)**" to make this native package

See "SOURCE PACKAGE FORMATS" in **dpkg-source**(1).

source/lintian-overrides or source.lintian-overrides [-x3] These files are not installed, but will be scanned by the **lintian** command to provide overrides for the source package.

See **dh_lintian**(1) and **lintian**(1).

source/local-options [-x1] The **dpkg-source** command uses this content as its options. Notable options are:

- **unapply-patches**
- **abort-on-upstream-changes**
- **auto-commit**
- **single-debian-patch**

This is not included in the generated source package and is meant to be committed to the VCS of the maintainer.

See "FILE FORMATS" in **dpkg-source**(1).

source/local-patch-header Free form text that is put on top of the automatic patch generated.

This is not included in the generated source package and is meant to be committed to the VCS of the maintainer.

+ See "FILE FORMATS" in **dpkg-source**(1).

***binarypackage*.symbols** [-x2] The symbols files, if present, are passed to the **dpkg-gensymbols** command to be processed and installed.

See **dh_makeshlibs**(1) and 第 5.18.1 节..

***binarypackage*.templates** This is the **debconf templates** file used for asking any questions necessary to configure the package. See 第 5.19 节.

tests/control This is the RFC822-style test meta data file defined in DEP-8. See **autopkgtest**(1) and 第 5.22 节.

TODO Installed into the first binary package listed in the **debian/control** file as **usr/share/doc/***binarypackage***/TODO.Debian**.

参见 **dh_installdocs**(1)。

binarypackage*.tmpfile** [-x3] If this exists, it is installed into **usr/lib/tmpfiles.d/binarypackage***.conf** in *binarypackage*.

See **dh_systemd_enable**(1), **dh_systemd_start**(1), and **dh_installinit**(1).

***binarypackage*.upstart** [-x3] If this exists, it is installed into etc/init/package.conf in the package build directory. (deprecated)

See **dh_installinit**(1) and 第 8.1 节.

watch [-x1] The control file for the **uscan** command to download the latest upstream version.

This control file may be configured to verify the authenticity of the tarball using its GPG signature (see 第 5.9 节).

See 第 5.10 节 and **uscan**(1).

Here are a few reminders for the above list.

- For a single binary package, the *binarypackage***.** part of the filename in the list may be removed.
- For a multi binary package, a configuration file missing the *binarypackage***.** part of the filename is applied to the first binary package listed in the **debian/control**.
- When there are many binary packages, their configurations can be specified independently by prefixing their name to their configuration filenames such as *package-1***.install**, *package-2***.install**, etc.
- Some template configuration files may not be created by the **debmake** command. In such cases, you need to create them with an editor.
- Unusual configuration template files generated by the **debmake** command with an extra **.ex** suffix need to be activated by removing that suffix.
- Unused configuration template files generated by the **debmake** command should be removed.
- Copy configuration template files as needed to the filenames matching their pertinent binary package names.

5.12 Debian 打包的定制化

我们来重新归纳一下 Debian 打包定制化的相关内容。

All customization data for the Debian package resides in the **debian/** directory. A simple example is given in 第 4.6 节. Normally, this customization involves a combination of the following:

- The Debian package build system can be customized through the **debian/rules** file (see 第 5.4.3 节).
- The Debian package installation path etc. can be customized through the addition of configuration files such as *package***.install** and *package***.docs** in the **debian/** directory for the **dh_*** commands from the **debhelper** package (see 第 5.11 节).

When these are not sufficient to make a good Debian package, modifications to the upstream source recorded as the **-p1** patches in the **debian/patches/** directory is deployed. These patches are applied in the sequence defined in the **debian/patches/series** file before building the package (see 第 5.8 节). Simple examples are given in 第 4.8 节.

You should address the root cause of the Debian packaging problem by the least invasive way. The generated package shall be more robust for future upgrades in this way.

注意

Send the patch addressing the root cause to the upstream maintainer if it is useful to the upstream.

5.13 在版本控制系统中进行记录（标准）

通常情况下，Debian 打包活动使用 Git 作为版本控制系统（VCS）进行记录；通常会用到下列的分支：

- **master** 分支
  - 记录用于 Debian 打包的源代码树。
  - 源码树的上游部分将照原样记录，不做修改。
  - Debian 打包中需要对上游源代码所作的修改记录在 **debian/patches/** 目录中，以 **-p1** 等级的补丁形式存在。
- **upstream** 分支
  - 记录从上游发布的 tarball（源码压缩文件）解压缩所得到的源代码树。

提示

添加一个 **.gitignore** 文件并将 **.pc** 文件列入其中也是一个好主意。

提示

Add **unapply-patches** and **abort-on-upstream-changes** lines to the **debian/source/local-options** file to keep the upstream portion unmodified.

提示

You may also track the upstream VCS data with a branch different from the **upstream** branch to ease cherry-picking of patches.

5.14 在版本控制系统中进行记录（备选方案）

You may not wish to keep up with creating the **-p1** patch files for all upstream changes needed. You can record the Debian packaging activity with the following branches.

- **master** 分支
  - 记录用于 Debian 打包的源代码树。
  - The upstream portion of the source tree is recorded with modifications for the Debian packaging.
- **upstream** 分支
  - 记录从上游发布的 tarball（源码压缩文件）解压缩所得到的源代码树。

Adding a few extra files in the **debian/** directory enables you to do this.

```
$ tar -xvzf <package-version>.tar.gz
$ ln -sf <package_version>.orig.tar.gz
$ cd <package-version>/
... hack...hack...
$ echo "single-debian-patch" >> debian/source/local-options
$ cat >debian/source/local-patch-header <<END
This patch contains all the Debian-specific changes mixed
together. To review them separately, please inspect the VCS
history at https://git.debian.org/?=collab-maint/foo.git.
```

Let the **dpkg-source** command invoked by the Debian package build process (**dpkg-buildpackage**, **debuild**, ⋯) generate the **-p1** patch file **debian/patches/debian-changes** automatically.

提示

This approach can be adopted for any VCS tools. Since this approach merges all changes into a merged patch, it is desirable to keep the VCS data publicly accessible.

提示

debian/source/local-options 和 **debian/source/local-patch-header** 文件只用于在版本控制系统中记录信息。它们不应包含在 Debian 源码包中。

5.15 Building package without extraneous contents

There are a few cases which cause the inclusion of undesirable contents in the generated Debian source package.

- The upstream source tree may be placed under the version control system. When the package is rebuilt from this source tree, the generated Debian source package contains extraneous contents from the version control system files.
- The upstream source tree may contain some auto-generated files. When the package is rebuilt from this source tree, the generated Debian source package contains extraneous contents from the auto-generated files.

Normally, the **-i** and **-I** options set in 第 3.5 节 for the **dpkg-source** command should avoid these. Here, the **-i** option is aimed at the non-native package while the **-I** is aimed at the native package. See **dpkg-source**(1) and the “**dpkg-source --help**” output.

There are several methods to avoid inclusion of undesirable contents.

5.15.1 Fix by debian/rules clean

The problem of extraneous contents can be fixed by removing such files in the “**debian/rules clean**” target. This is also useful for auto-generated files

注意

The “**debian/rules clean**”target is called before the “**dpkg-source --build**”command by the **dpkg-buildpackage** command and the “**dpkg-source --build**”command ignores removed files.

5.15.2 Fix using VCS

The problem of extraneous contents can be fixed by restoring the source tree by committing the source tree to the VCS before the first build.

You can restore the source tree before the second package build. For example:

```
$ git reset --hard
$ git clean -dfx
$ debuild
```

This works because the **dpkg-source** command ignores the contents of the typical VCS files in the source tree with the **DEBUILD_DPKG_BUILDPACKAGE_OPTS** setting in 第 3.5 节.

提示

If the source tree is not managed by a VCS, you should run “**git init; git add -A .; git commit**”before the first build.

5.15.3 Fix by extend-diff-ignore

This is for a non-native package.

The problem of extraneous diffs can be fixed by ignoring changes made to parts of the source tree by adding the “**extend-diff-ignore**=···” line in the **debian/source/options** file.

For excluding the **config.sub**, **config.guess** and **Makefile** files:

```
# Don't store changes on autogenerated files
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

注意

This approach always works, even when you can’t remove the file. So it saves you having to make a backup of the unmodified file just to be able to restore it before the next build.

提示

If the **debian/source/local-options** file is used instead, you can hide this setting from the generated source package. This may be useful when the local non-standard VCS files interfere with your packaging.

5.15.4 Fix by tar-ignore

This is for a native package.

You can exclude some files in the source tree from the generated tarball by tweaking the file glob by adding the "**tar-ignore=**⋯" lines in the **debian/source/options** or **debian/source/local-options** files.

注意

If, for example, the source package of a native package needs files with the file extension **.o** as a part of the test data, the setting in 第 3.5 节 is too aggressive. You can work around this problem by dropping the **-I** option for **DEBUILD_DPKG_BUILDPACKAGE_OPTS** in 第 3.5 节 while adding the "**tar-ignore=**⋯"lines in the **debian/source/local-options** file for each package.

5.16 上游构建系统

Upstream build systems are designed to go through several steps to install generated binary files to the system from the source distribution.

提示

Before attempting to make a Debian package, you should become familiar with the upstream build system of the upstream source code and try to build it.

5.16.1 Autotools

Autotools (**autoconf** + **automake**) has 4 steps.

1. set up the build system ("**vim configure.ac Makefile.am**" and "**autoreconf -ivf**")
2. configure the build system ("**./configure**")
3. build the source tree ("**make**")
4. install the binary files ("**make install**")

The upstream maintainer usually performs step 1 and builds the upstream tarball for distribution using the "**make dist**" command. (The generated tarball contains not only the pristine upstream VCS contents but also other generated files.)

The package maintainer needs to take care of steps 2 to 4 at least. This is realized by the "**dh $@ --with autotools-dev**" command used in the **debian/rules** file.

The package maintainer may wish to take care all steps 1 to 4. This is realized by the "**dh $@ --with autoreconf**" command used in the **debian/rules** file. This rebuilds all auto-generated files to the latest version and provides better support for porting to the newer architectures.

For **compat** level **10** or newer, the simple "**dh $@**" command without "**--with autoreconf**" option can take care all steps 1 to 4, too.

If you wish to learn more on Autotools, please see:

- GNU Automake 文档
- GNU Autoconf 文档
- Autotools 教程
- Introduction to the autotools (autoconf, automake, and libtool)
- Autotools Mythbuster

5.16.2 CMake

CMake has 4 steps.

1. set up the build system (“**vim CMakeLists.txt config.h.in**”)
2. configure the build system (“**cmake**”)
3. build the source tree (“**make**”)
4. install the binary files (“**make install**”)

The upstream tarball contains no auto-generated files and is generated by the **tar** command after step 1.
The package maintainer needs to take care of steps 2 to 4.
If you wish to learn more on the CMake, please see:

- CMake
- CMake 教程

5.16.3 Python distutils

Python distutils has 3 steps.

1. set up and configure the build system (“**vim setup.py**”)
2. build the source tree (“**python setup.py build**”)
3. install the binary files (“**python setup.py install**”)

The upstream maintainer usually performs step 1 and builds the upstream tarball for distribution using the “**python setup.py sdist**” command.
The package maintainer needs to take care of step 2. This is realized simply by the “**dh $@**” command used in the **debian/rules** file, after **jessie**.
The situation of other build systems, such as CMake, are very similar to this Python one.
If you wish to learn more on Python3 and **distutils**, please see:

- Python3
- distutils

5.17 调试信息

The Debian package is built with the debugging information but packaged into the binary package after stripping the debugging information as required by Chapter 10 - Files of the “Debian Policy Manual”.
See

- 6.7.9. Best practices for debug packages of the “Debian Developer’s Reference”.
- 18.2 Debugging Information in Separate Files of the “Debugging with gdb”
- **dh_strip**(1)
- **strip**(1)
- **readelf**(1)
- **objcopy**(1)
- Debian 维基 DebugPackage
- Debian 维基 AutomaticDebugPackages
- Debian debian-devel 列表发布的邮件：Status on automatic debug packages (2015-08-15)

5.17.1 新的 -dbgsym 软件包（Stretch 9.0 或更新）

The debugging information is automatically packaged separately as the debug package using the **dh_strip** command with its default behavior. The name of such a debug package normally has the **-dbgsym** suffix.

If there were no **-dbg** packages defined in the **debian/control** file, no special care is needed for updating the package after the Stretch 9.0 release.

- The **debian/rules** file shouldn’t explicitly contain **dh_strip**.
- Set **debian/compat** to **11** or newer.
- Bump the **Build-Depends** to **debhelper (>=11~)** or newer.

If there were **-dbg** packages defined in the **debian/control** file, following care is needed for updating the old package after the Stretch 9.0 release.

- Drop definition entries of such **-dbg** packages in the **debian/control** file.
- Replace “**dh_strip --dbg-package=***package*” with “**dh_strip --dbgsym-migration=***package*” in the **debian/rules** file to avoid file conflicts with the (now obsolete) **-dbg** package. See **dh_strip**(1).
- Set **debian/compat** to **11** or newer.
- Bump the **Build-Depends** to **debhelper (>=11~)** or newer.

5.18 库软件包

Packaging library software requires you to perform much more work than usual. Here are some reminders for packaging library software:

- The library binary package must be named as in 第 5.5.1.3 节.
- Debian ships shared libraries such as **/usr/lib/<triplet>/lib***foo-0.1***.so.***1.0.0* (see 第 5.20 节).
- Debian encourages using versioned symbols in the shared library (see 第 5.18.1 节).
- Debian doesn’t ship ***.la** libtool library archive files.
- Debian discourages using and shipping ***.a** static library files.

Before packaging shared library software, see:

- Chapter 8 - Shared libraries of the “Debian Policy Manual”
- 10.2 Libraries of the “Debian Policy Manual”
- 6.7.2. Libraries of the “Debian Developer’s Reference”

如需研究其历史背景，请参见：

- 逃离依赖地狱 [6]
  - 该文档鼓励在共享库中使用带版本的符号。
- Debian 库打包指南 [7]
  - Please read the discussion thread following its announcement, too.

[6] 该文档是在 **symbols** 文件被引入之前写成的。

[7] The strong preference is to use the SONAME versioned **-dev** package names over the single **-dev** package name in Chapter 6. Development (-DEV) packages, which does not seem to be shared by the former ftp-master (Steve Langasek). This document was written before the introduction of the **multiarch** system and the **symbols** file.

5.18.1 库符号

The symbols support in **dpkg** introduced in Debian **lenny** (5.0, May 2009) helps us to manage the backward ABI compatibility of the library package with the same package name. The **DEBIAN/symbols** file in the binary package provides the minimal version associated with each symbol.

An oversimplified method for the library packaging is as follows.

- Extract the old **DEBIAN/symbols** file of the immediate previous binary package with the "**dpkg-deb -e**" command.
  - Alternatively, the **mc** command may be used to extract the **DEBIAN/symbols** file.
- Copy it to the **debian/***binarypackage***.symbols** file.
  - If this is the first package, use an empty content file instead.
- Build the binary package.
  - If the **dpkg-gensymbols** command warns about some new symbols:
    - Extract the updated **DEBIAN/symbols** file with the "**dpkg-deb -e**" command.
    - Trim the Debian revision such as **-1** in it.
    - Copy it to the **debian/***binarypackage***.symbols** file.
    - 重新构建二进制软件包。
  - If the **dpkg-gensymbols** command does not warn about new symbols:
    - You are done with the library packaging.

For the details, you should read the following primary references.

- 8.6.3 The symbols system of the "Debian Policy Manual"
- **dh_makeshlibs**(1)
- **dpkg-gensymbols**(1)
- **dpkg-shlibdeps**(1)
- **deb-symbols**(5)

Yous should also check:

- Debian 维基 UsingSymbolsFiles
- Debian 维基 Projects/ImprovedDpkgShlibdeps
- Debian kde 团队 Working with symbols files
- 第 8.11 节
- 第 8.12 节

提示

For C++ libraries and other cases where the tracking of symbols is problematic, follow 8.6.4 The shlibs system of the "Debian Policy Manual", instead. Please make sure to erase the empty **debian/***binarypackage***.symbols** file generated by the **debmake** command. For this case, the **DEBIAN/shlibs** file is used.

5.18.2 库变迁

When you package a new library package version which affects other packages, you must file a transition bug report against the **release.debian.org** pseudo package using the **reportbug** command with the ben file and wait for the approval for its upload from the Release Team.

Release team has the transition tracker. See Transitions.

小心

Please make sure to rename binary packages as in 第 5.5.1.3 节.

5.19 debconf

The **debconf** package enables us to configure packages during their installation in 2 main ways:

- non-interactively from the **debian-installer** pre-seeding.
- interactively from the menu interface (**dialog**, **gnome**, **kde**, ···)
  - the package installation: invoked by the **dpkg** command
  - the installed package: invoked by the **dpkg-reconfigure** command

All user interactions for the package installation must be handled by this **debconf** system using the following files.

- **debian**/*binarypackage*.**config**
  - This is the **debconf config** script used for asking any questions necessary to configure the package.
- **debian**/*binarypackage*.**template**
  - This is the **debconf templates** file used for asking any questions necessary to configure the package.
- package configuration scripts
  - **debian**/*binarypackage*.**preinst**
  - **debian**/*binarypackage*.**prerm**
  - **debian**/*binarypackage*.**postinst**
  - **debian**/*binarypackage*.**postrm**

See **dh_installdebconf**(1), **debconf**(7), **debconf-devel**(7) and 3.9.1 Prompting in maintainer scripts in the "Debian Policy Manual".

5.20 多体系结构

Multiarch support for cross-architecture installation of binary packages (particularly **i386** and **amd64**, but also other combinations) in the **dpkg** and **apt** packages introduced in Debian **wheezy** (7.0, May 2013), demands that we pay extra attention to packaging.

You should read the following references in detail.

- Ubuntu 维基（上游）
  - MultiarchSpec
- Debian wiki (Debian situation)
  - Debian multiarch support
  - Multiarch/Implementation

The multiarch is enabled by using the **<triplet>** value such as **i386-linux-gnu** and **x86_64-linux-gnu** in the install path of shared libraries as **/usr/lib/<triplet>/**, etc..

- The **<triplet>** value required internally by **debhelper** scripts is implicitly set in themselves. The maintainer doesn't need to worry.
- The **<triplet>** value used in **override_dh_*** target scripts must be explicitly set in the **debian/rules** file by the maintainer. The **<triplet>** value is stored in the **$(DEB_HOST_MULTIARCH)** variable in the following **debian/rules** snippet example:

```
DEB_HOST_MULTIARCH = $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)
...
override_dh_install:
        mkdir -p package1/lib/$(DEB_HOST_MULTIARCH)
        cp -dR tmp/lib/. package1/lib/$(DEB_HOST_MULTIARCH)
```

See:

- 第 5.4.4 节
- **dpkg-architecture**(1)
- 第 5.5.1.1 节
- 第 5.5.1.2 节

5.20.1 The multiarch library path

Debian policy requires following Filesystem Hierarchy Standard. Its /usr/lib : Libraries for programming and packages states "**/usr/lib** includes object files, libraries, and internal binaries that are not intended to be executed directly by users or shell scripts."

Debian policy makes an exception to the Filesystem Hierarchy Standard to use **/usr/lib/<triplet>/** instead of **/usr/lib<qual>/** (e.g., **/lib32/** and **/lib64/**) to support a multiarch library.

Table 5.1 The multiarch library path options

Classic path	**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/**

For Autotools based packages under the **debhelper** package (compat>=9), this path setting is automatically taken care by the **dh_auto_configure** command.

For other packages with non-supported build systems, you need to manually adjust the install path as follows.

- If "**./configure**" is used in the **override_dh_auto_configure** target in **debian/rules**, make sure to replace it with "**dh_auto_configure --**"while re-targeting the install path from **/usr/lib/** to **/usr/lib/$(DEB_HOST_MULTIARCH)/**.
- Replace all occurrences of **/usr/lib/** with **/usr/lib/*/** in **debian/***foo***.install** files.

All files installed simultaneously as the multiarch package to the same file path should have exactly the same file content. You must be careful with differences generated by the data byte order and by the compression algorithm.

注意

The **--libexecdir** option of the **./configure** command specifies the default path to install executable programs run by other programs rather than by users. Its Autotools default is **/usr/libexec/** but its Debian non-multi-arch default is **/usr/lib/**. If such executables are a part of a "Multi-arch: foreign" package, a path such as **/usr/lib/** or **/usr/lib/***packagename* may be more desirable than **/usr/lib/<triplet>/**, which **dh_auto_configure** uses. The GNU Coding Standards: 7.2.5 Variables for Installation Directories has a description for **libexecdir** as "The definition of **libexecdir** is the same for all packages, so you should install your data in a subdirectory thereof. Most packages install their data under **$(libexecdir)/package-name/** …". (It is always a good idea to follow GNU unless it conflicts with the Debian policy.)

The shared library files in the default path **/usr/lib/** and **/usr/lib/<triplet>/** are loaded automatically.

For shared library files in another path, the GCC option **-l** must be set by the **pkg-config** command to make them load properly.

5.20.2 The multiarch header file path

GCC includes both **/usr/include/** and **/usr/include/<triplet>/** by default on the multiarch Debian system.

If the header file is not in those paths, the GCC option **-I** must be set by the **pkg-config** command to make "**#include** <*foo.h*>" work properly.

Table 5.2 The multiarch header file path options

Classic path	**i386** 多体系结构路径	**amd64** 多体系结构路径
/usr/include/	**/usr/include/i386-linux-gnu/**	**/usr/include/x86_64-linux-gnu/**
/usr/include/*packagename*/	**/usr/include/i386-linux-gnu/***packagename*/	**/usr/include/x86_64-linux-gnu/***packagename*/
	/usr/lib/i386-linux-gnu/*packagename*/	**/usr/lib/x86_64-linux-gnu/***packagename*/

The use if the **/usr/lib/<triplet>/***packagename*/ path for the header files allows the upstream maintainer to use the same install script for the multiatch system with **/usr/lib/<triplet>** and the biarch system with **/usr/lib<qual>/**. [8]

The use of the file path containing *packagename* enables having more than 2 development libraries simultaneously installed on a system.

5.20.3 The multiarch *.pc file path

The **pkg-config** program is used to retrieve information about installed libraries in the system. It stores its configuration parameters in the ***.pc** file and is used for setting the **-I** and **-l** options for GCC.

Table 5.3 The ***.pc** file path options

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

5.21 编译加固

The compiler hardening support spreading for Debian **jessie** (8.0, TBA) demands that we pay extra attention to the packaging.

[8] This path is compliant with the FHS. Filesystem Hierarchy Standard: /usr/lib : Libraries for programming and packages states "Applications may use a single subdirectory under **/usr/lib**. If an application uses a subdirectory, all architecture-dependent data exclusively used by the application must be placed within that subdirectory."

You should read the following references in detail.

- Debian 维基 Hardening（加固）
- Debian 维基 Hardening Walkthrough（加固指南）

debmake 命令会向 **debian/rules** 文件中按需添加 **DEB_BUILD_MAINT_OPTIONS**、**DEB_CFLAGS_MAINT_APPE** 和 **DEB_LDFLAGS_MAINT_APPEND** 的项目（参见第 4 章和 **dpkg-buildflags**(1)）。

5.22 持续集成

DEP-8 defines the **debian/tests/control** file as the RFC822-style test metadata file for continuous integration (CI) of the Debian package.

It is used after building the binary packages from the source package containing this **debian/tests/control** file. When the **autopkgtest** command is run, the generated binary packages are installed and tested in the virtual environment according to this file.

See documents in the **/usr/share/doc/autopkgtest/** directory and 3. autopkgtest: Automatic testing for packages of the "Ubuntu Packaging Guide".

There are several other CI tools on Debian for you to explore.

- The **debci** package: CI platform on top of the **autopkgtest** package
- The **jenkins** package: generic CI platform

5.23 Bootstrapping

Debian cares about supporting new ports or flavours. The new ports or flavours require bootstrapping operation for the cross-build of the initial minimal native-building system. In order to avoid build-dependency loops during bootstrapping, the build-dependency needs to be reduced using the profile builds feature.

提示

If a core package `foo` build depends on a package `bar` with deep build dependency chains but `bar` is only used in the **test** target in `foo`, you can safely mark the `bar` with **<!nocheck>** in the **Build-depends** of `foo` to avoid build loops.

5.24 Bug reports

The **reportbug** command used for the bug report of *binarypackage* can be customized by the files in **usr/share/bug/***binarypackage*

The **dh_bugfiles** command installs these files from the template files in the **debian/** directory.

- **debian/***binarypackage***.bug-control** → **usr/share/bug/***binarypackage***/control**
  - This file contains some directions such as redirecting the bug report to another package.
- **debian/***binarypackage***.bug-presubj** → **usr/share/bug/***binarypackage***/presubj**
  - This file is displayed to the user by the **reportbug** command.
- **debian/***binarypackage***.bug-script** → **usr/share/bug/***binarypackage* or **usr/share/bug/***binarypackage***/script**
  - The **reportbug** command runs this script to generate a template file for the bug report.

See **dh_bugfiles**(1) and reportbug's Features for Developers

提示

If you always remind the bug reporter of something or ask them about their situation, use these files to automate it.

Chapter 6

debmake options

Here are some notable options for the **debmake** command.

6.1 Shortcut options (-a, -i)

The **debmake** command offers 2 shortcut options.

- **-a** : open the upstream tarball
- **-i** : execute script to build the binary package

The example in the above 第 4 章 can be done simply as follows.

```
$ debmake -a package-1.0.tar.gz -i debuild
```

提示

A URL such as “**https://www.example.org/DL/package-1.0.tar.gz**”may be used for the **-a** option.

提示

A URL such as “**https://arm.koji.fedoraproject.org/packages/ibus/1.5.7/3.fc21/src/ibus-1.5.7-3.fc21.src.rpm**”may be used for the **-a** option, too.

6.1.1 Python module

You can generate a functioning single binary Debian package with a reasonable package description directly from the Python module package offered as a tarball, *pythonmodule-1.0***.tar.gz**. The **-b** option specifying the package type **python** and the **-s** option to copy the package description from the upstream package need to be specified.

```
$ debmake -s -b':python' -a pythonmodule-1.0.tar.gz -i debuild
```

For other interpreted languages that support the **-b** option, specify the pertinent *type* for the **-b** option.

For interpreted languages without the **-b** option support, specify the **script** type instead and add the interpreter package as a dependency of the resulting binary package by adjusting the **debian/control** file.

6.2 Upstream snapshot (-d, -t)

The upstream snapshot from the upstream source tree in the VCS can be made with the **-d** option if the upstream package supports the "**make dist**" equivalence.

```
$ cd /path/to/upstream-vcs
$ debmake -d -i debuild
```

Alternatively, the same can be made with the **-t** option if the upstream tarball can be made with the **tar** command.

```
$ cd /path/to/upstream-vcs
$ debmake -p package -t -i debuild
```

Unless you provide the upstream version with the **-u** option or with the **debian/changelog** file, a snapshot upstream version is generated in the **0~%y%m%d%H%M** format, e.g., *0~1403012359*, from the UTC date and time.

If the upstream VCS is hosted in the *package/* directory instead of the *upstream-vcs/* directory, the "**-p** *package*" can be skipped.

If the upstream source tree in the VCS contains the **debian/*** files, the **debmake** command with either the **-d** option or the **-t** option combined with the **-i** option automates the making of a non-native Debian package from the VCS snapshot while using these **debian/*** files.

```
$ cp -r /path/to/package-0~1403012359/debian/. /path/to/upstream-vcs/debian
$ dch
  ... update debian/changelog
$ git add -A .; git commit -m "vcs with debian/*"
$ debmake -t -p package -i debuild
```

This **non-native** Debian binary package building scheme using the "**debmake -t -i debuild**" command may be considered as the **quasi-native** Debian package scheme since the packaging situation resembles the **native** Debian binary package building case using the **debuild** command without the upstream tarball.

Use of a **non-native** Debian package helps to ease communication with the downstream distros such as Ubuntu for bug fixes etc.

6.3 debmake -cc

The **debmake** command with the **-cc** option can make a summary of the copyright and license for the entire source tree to standard output.

```
$ tar -xvzf package-1.0.tar.gz
$ cd package-1.0
$ debmake -cc | less
```

With the **-c** option, this provides shorter report.

6.4 debmake -k

When updating a package for the new upstream release, the **debmake** command can verify the content of the existing **debian/copyright** file against the copyright and license situation of the entire updated source tree.

```
$ cd package-vcs
$ gbp import-orig --uscan --pristine-tar
... update source with the new upstream release
$ debmake -k | less
```

The "**debmake -k**" command parses the **debian/copyright** file from the top to the bottom and compares the license of all the non-binary files in the current package with the license described in the last matching file pattern entry of the **debian/copyright** file.

When editing the auto-generated **debian/copyright** file, please make sure to keep the generic file patterns at the top of the list.

提示

For all new upstream releases, run the “**debmake -k**”command to ensure that the **debian/copyright** file is current.

6.5 debmake -j

The generation of a functioning multi-binary package always requires more manual work than that of a functioning single binary package. The test build of the source package is the essential part of it.

For example, let’s package the same *package-1.0***.tar.gz** (see 第 4 章) into a multi binary package.

- Invoke the **debmake** command with the **-j** option for the test building and the report generation.

```
$ debmake -j -a package-1.0.tar.gz
```

- Check the last lines of the *package***.build-dep.log** file to judge build dependencies for **Build-Depends**. (You do not need to list packages used by **debhelper**, **perl**, or **fakeroot** explicitly in **Build-Depends**. This technique is useful for the generation of a single binary package, too.)
- Check the contents of the *package***.install.log** file to identify the install paths for files to decide how you split them into multiple packages.
- Start packaging with the **debmake** command.

```
$ rm -rf package-1.0
$ tar -xvzf package-1.0.tar.gz
$ cd package-1.0
$ debmake -b"package1:type1, ..."
```

- Update **debian/control** and **debian/***binarypackage***.install** files using the above information.
- Update other **debian/*** files as needed.
- Build the Debian package with the **debuild** command or its equivalent.

```
$ debuild
```

- All binary package entries specified in the **debian/***binarypackage***.install** file are generated as *binarypackage_version-revision_arch***.deb**.

注意

The **-j** option for the **debmake** command invokes **dpkg-depcheck**(1) to run **debian/rules** under **strace**(1) to obtain library dependencies. Unfortunately, this is very slow. If you know the library package dependencies from other sources such as the SPEC file in the source, you may just run the ”**debmake** ⋯” command without the **-j** option and run the “**debian/rules install**”command to check the install paths of the generated files.

6.6 debmake -x

The amount of template files generated by the **debmake** command depends on the **-x**[01234] option.

- See 第 8.1 节 for cherry-picking of the template files.

注意

None of the existing configuration files are modified by the **debmake** command.

6.7 debmake -P

The **debmake** command invoked with the **-P** option pedantically checks auto-generated files for copyright+license text even if they are with permissive license.

This option affects not only the content of the **debian/copyright** file generated by normal execution, but also the output by the execution with the **-k**, **-c**, **-cc**, and **-ccc** options.

6.8 debmake -T

The **debmake** command invoked with the **-T** option additionally prints verbose tutorial comment lines. The lines marked with **###** in the template files are part of the verbose tutorial comment lines.

Chapter 7

提示

这里有一些关于 Debian 打包的值得注意的提示。

7.1 debdiff

您可以使用 **debdiff** 命令来对比两个 Debian 软件包组成的差别。

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

您也可以使用 **debdiff** 命令来对比两组二进制 Debian 软件包中的文件列表。

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

当检查源代码包中哪些文件被修改时，这个命令非常有用。它还可以用来检测二进制包中是否有文件在更新过程中发生变动，比如被意外替换或删除。

7.2 dget

您可以使用 **dget** 命令来下载 Debian 源包的文件集。

```
$ dget https://www.example.org/path/to/package_version-rev.dsc
```

7.3 debc

您应该使用 **debc** 命令安装生成的软件包以在本地测试它。

```
$ debc package_version-rev_arch.changes
```

7.4 piuparts

您应该使用 **piuparts** 命令安装生成的软件包以自动进行测试。

```
$ sudo piuparts package_version-rev_arch.changes
```

注意

这是一个非常慢的过程，因为它需要访问远程 APT 软件包仓库。

7.5 debsign

完成软件包的测试后，您可以使用 **debsign** 命令对其进行签名。

```
$ debsign package_version-rev_arch.changes
```

7.6 dput

使用 **debsign** 命令对包进行签名后，您可以使用 **dput** 命令上传 Debian 源和二进制包的文件集。

```
$ dput package_version-rev_arch.changes
```

7.7 bts

上传软件包后，您将收到错误报告。如”Debian 开发人员参考”5.8. 处理缺陷 中所述，正确地管理这些错误是软件包维护者的一项重要职责。

bts 命令是一个用以处理 Debian 缺陷追踪系统 上的错误的便捷工具。

```
$ bts severity 123123 wishlist , tags -1 pending
```

7.8 git-buildpackage

git-buildpackage 软件包提供了许多命令来使用 git 仓库自动打包

- **gbp import-dsc**：向 git 仓库中导入先前的 Debian 源码包。
- **gbp import-orig**：向 git 仓库中导入新的上游原始码
  - **git import-orig** 命令的 **--pristine-tar** 选项允许将上游源码包储存在同一个 git 仓库中。
  - 将 **--uscan** 选项作为 **gbp import-orig** 命令的最后一个参数会允许下载上游原始码并提交到 git 仓库中。
- **gbp dch**: generate the Debian changelog from the git commit messages.
- **gbp buildpackage**：从 git 仓库中构建 Debian 二进制包。
- **gbp pull**：从远程仓库中安全更新 **debian**, **upstream** and **pristine-tar** 分支。
- **git-pbuilder**：使用 **pbuilder** 软件包从 git 仓库构建 Debian 二进制软件包。
  - 使用 **cowbuilder** 软件包作为后端。
- The **gbp pq**, **git-dpm** or **quilt** (or alias **dquilt**) commands are used to manage quilt patches.
  - **dquilt** 命令是学起来最简单的，它只要求您使用 **git** 命令手动提交最后的文件至 **master** 分支。
  - The “**gbp pq**” command provides the equivalent functionality of patch set management without using **dquilt** and eases including upstream git repository changes by cherry-picking.
  - “**git dpm**” 命令提供了比 “**gbp pq**” 更强大的功能。

使用 **git-buildpackage** 软件包来管理软件包历史正成为绝大多数 Debian 维护者的实践标准。
See:

- 使用 git-buildpackage 构建 Debian 软件包
- https://wiki.debian.org/GitPackagingWorkflow
- https://wiki.debian.org/GitPackagingWorkflow/DebConf11BOF
- https://raphaelhertzog.com/2010/11/18/4-tips-to-maintain-a-3-0-quilt-debian-source-package-in-a-vcs/
- **systemd** 打包实践文档在 从源码构建。

提示

放松。您并不需要使用全部的打包工具，您只需要使用您所需要的那个就行。

7.8.1 gbp import-dscs --debsnap

对于记录在 snapshot.debian.org 归档中的名为 <*source-package*> 的 Debian 源码包，可以生成包含所有 Debian 版本历史的初始 git 存储库，如下所示。

```
$ gbp import-dscs --debsnap --pristine-tar '<source-package>'
```

7.9 上游 git 仓库

对于使用 **git-buildpackage** 打包的 Debian 软件包，远程存储库 **origin** 上的 **upstream** 分支通常用于跟踪已发布的上游原始码的内容。

也可以通过将其远程存储库命名为 **upstream** 而不是默认的 **origin** 来跟踪上游 git 仓库。然后，您可以通过使用 **gitk** 命令和 **gbp-pq** 命令进行挑选，轻松地将最近的上游更改添加到 Debian 修订版中。

提示

The "**gbp import-orig --upstream-vcs-tag**"command can create a nice packaging history by making a merge commit into the **upstream** branch from the specified tag on the upstream git repository.

小心

已发布的上游原始码的内容可能与上游 git 存储库的相应内容并不完全匹配。它可能包含一些自动生成的文件或遗漏一些文件。（Autotools、distutils.....）

7.10 chroot

可以使用第 3 章. [1] 中描述的工具创建和管理干净包构建环境的 chroot

以下是可用的软件包构建命令的快速总结。有很多方法可以做同样的事情。

- **dpkg-buildpackage** = 软件包打包工具的核心
- **debuild** = **dpkg-buildpackage** + **lintian** （在清理后的环境变量下构建）
- **pbuilder** = Debian chroot 环境工具的核心
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** （在 chroot 环境下构建）
- **cowbuilder** = 提升 **pbuilder** 执行的速度
- **git-pbuilder** = **pdebuild** 的易于使用的命令行语法（由 **gbp buildpackge** 使用）
- **gbp** = 在 git 下管理 Debian 源代码

[1] https://wiki.debian.org/git-pbuilder 采用了 **git-pbuilder** 风格管理。请小心，因为许多 HOWTO 使用了不同的组织。

- **gbp buildpackge** = **pbuilder** + **dpkg-buildpackage** + **gbp**

可以根据如下方式使用干净的 **sid** 版本的 chroot 环境。

- 用于 **sid** 发行版的 chroot 文件系统创建命令
  - **pbuilder create**
  - **git-pbuilder create**
- **sid** 版本的 chroot 文件系统的文件路径
  - **/var/cache/pbuilder/base.cow**
- **sid** 发行版 chroot 的包构建命令
  - **pdebuild**
  - **git-pbuilder**
  - **gbp buildpackage**
- 更新 **sid** chroot 的命令
  - **pbuilder --update**
  - **git-pbuilder update**
- The command to login to the **sid** chroot filesystem to modify it
  - **git-pbuilder login --save-after-login**

可以根据如下方式使用任意的 *dist* 版本环境。

- 用于 **dist** 版本的 chroot 文件系统创建命令
  - **pbuilder create --distribution** *dist*
  - **DIST=***dist* **git-pbuilder create**
- **dist** 版本的 chroot 文件系统的文件路径
  - path: **/var/cache/pbuilder/base-***dist***.cow**
- **dist** 版本 chroot 的包构建命令
  - **pdebuild -- --basepath=/var/cache/pbuilder/base-***dist***.cow**
  - **DIST=***dist* **git-pbuilder**
  - **gbp buildpackage --git-dist=***dist*
- 更新 **dist** chroot 的命令
  - **pbuilder update --basepath=/var/cache/pbuilder/base-***dist***.cow**
  - **DIST=***dist* **git-pbuilder update**
- The command to login to the **dist** chroot to modify it
  - **pbuilder --login --basepath=/var/cache/pbuilder/base-***dist***.cow --save-after-login**
  - **DIST=***dist* **git-pbuilder login --save-after-login**

提示

使用这个“git-pbuilder login --save-after-login”命令可以非常方便地创建一个包含一些新实验包所需的预加载包的自定义环境。

提示

如果您的旧 chroot 文件系统缺少例如 **libeatmydata1**、**ccache** 和 **lintian** 等软件包，您可能需要使用“**git-pbuilder login --save-after-login**”命令来安装这些软件包。

提示

只需使用“**cp -a base-***dist***.cow base-***customdist***.cow**”命令即可克隆 chroot 文件系统。新的 chroot 可以以“**gbp buildpackage --git-dist=***customdist*”和“**DIST=***customdist* **git-pbuilder** …”访问。

提示

当需要为除 **0** 和 **1** 之外的 Debian 修订版上传 **orig.tar.gz** 文件时（例如，对于安全性上传），将 **-sa** 选项添加到 **dpkg-buildpackage**，**debuild**，**pdebuild** 和 **git-pbuilder** 命令末尾。对于“**gbp buildpackage**”命令，临时修改 **~/.gbp.conf** 中的 **builder** 设置。

注意

本节中的描述过于简洁，对大多数潜在的维护者都没用。这是作者的有意为之。我们强烈建议您搜索并阅读与所用命令相关的所有文档。

7.11 新的 Debian 版本

让我们假设有一个错误报告 ***#bug_number*** 是针对您的软件包的。它描述了一个问题，您可以通过编辑上游源码中的错误文件来解决。以下是使用 **bugname.patch** 文件创建软件包的新 Debian 软件包修订版所需的操作。

使用 **dquilt** 命令进行新的 **Debian** 修订

```
$ dquilt push -a
$ dquilt new bugname.patch
$ dquilt add buggy
$ vim buggy
  ...
$ dquilt refresh
$ dquilt header -e
$ dquilt pop -a
$ dch -i
```

Alternatively if the package is managed in the git repository using the **git-buildpackage** command with its default configuration:

使用 **gbp-pq** 命令进行新的 **Debian** 修订

```
$ git checkout master
$ gbp pq import
```

```
$ vim buggy
$ git add buggy
$ git commit
$ git tag pq/<newrev>
$ gbp pq export
$ gbp drop
$ git add debian/patches/*
$ dch -i
$ git commit -a -m "Closes: #<bug_number>"
```

请确保简明扼要地描述修复报告错误的更改并通过在 **debian/changelog** 文件中添加“**Closes: #**<*bug_number*>”来关闭这些错误。

提示

在实验时使用带有版本字符串的 **debian/changelog** 条目，例如 **1.0.1-1~rc1** 。然后，将这些更改日志条目整理到官方软件包的条目中。

7.12 新上游版本

如果 **foo** 包是以现代“**3.0 (native)**”或“**3.0 (quilt)**”格式正确打包的，则打包新的上游版本时需要将旧的 **debian/** 目录移动到新的源码路径中。这可以通过在新提取的源码路径中运行“**tar -xvzf** */path/to/-foo_oldversion***.debian.tar.gz**”命令来完成。[2] 当然，你还需要做一些修改。

有很多的工具可以用以处理这些情况。在使用这些软件来更新上游版本后，请在 **debian/changelog** 文件中简要描述修复错误的上游修改，并添加“**Closes: #***bug_number*”来关闭错误。

7.12.1 uupdate + tarball

您可以使用来自 **uupdate** 软件包中的 **uupdate** 命令来自动更新到新的上游源码。该命令需要旧的 Debian 源码包和新的上游源码包。

```
$ wget https://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
```

7.12.2 uscan

您可以使用来自 **uupdate** 软件包中的 **uscan** 命令来自动更新到新的上游源码。该命令需要包含 **debian/watch** 文件的旧的 Debian 源码包。

```
$ cd foo-oldversion
$ uscan
...
$ while dquilt push; do dquilt refresh; done
$ dch
```

7.12.3 gbp

您可以使用来自 **git-buildpackage** 软件包中的“**gbp import-orig --pristine-tar**”命令来自动更新到新的上游源码。该命令需要在 git 仓库中的 Debian 源码和新的上游源码包。

[2] 如果 **foo** 包是以旧的 **1.0** 格式打包的，则相对的，只要在新的源代码路径中执行“**zcat** */path/to/foo_oldversion***.diff.gz|patch -p1**”命令。

```
$ ln -sf foo-newversion.tar.gz foo_newversion.orig.tar.gz
$ cd foo-vcs
$ git checkout master
$ gbp pq import
$ git checkout master
$ gbp import-orig --pristine-tar  ../foo_newversion.orig.tar.gz
...
$ gbp pq rebase
$ git checkout master
$ gbp pq export
$ gbp pq drop
$ git add debian/patches
$ dch -v <newversion>
$ git commit -a -m "Refresh patches"
```

提示

如果上游也使用 git 仓库，请为 **gbp import-orig** 命令加上 **--upstream-vcs-tag** 选项。

7.12.4 gbp + uscan

您可以使用来自 **git-buildpackage** 软件包中的“**gbp import-orig --pristine-tar --uscan**”命令来自动更新到新的上游源码。该命令需要在 git 仓库中的包含 **debian/watch** 文件的 Debian 源码。

```
$ cd foo-vcs
$ git checkout master
$ gbp pq import
$ git checkout master
$ gbp import-orig --pristine-tar --uscan
...
$ gbp pq rebase
$ git checkout master
$ gbp pq export
$ gbp pq drop
$ git add debian/patches
$ dch -v <newversion>
$ git commit -a -m "Refresh patches"
```

提示

如果上游也使用 git 仓库，请为 **gbp import-orig** 命令加上 **--upstream-vcs-tag** 选项。

7.13 3.0 源代码格式

更新软件包的风格并不是更新软件包所必须的步骤。但是，这么做可以让您充分利用现代 **debhelper** 和 **3.0** 源码格式的所有能力。

- 如果您因任何原因需要重新创建已删除的模板文件，您可以在同一个 Debian 软件包源码树中再次运行 **debmake**。然后适当地编辑它们。

- 如果软件包还未更新到可为 **debian/rules** 文件使用 **dh** 命令，请升级它以便使用该命令（参见第 5.4.2 节）。请根据具体情况更新 **debian/control** 文件。
- 如果你有一个带有 **foo.diff.gz** 文件的 **1.0** 格式的源码包，你可以通过创建带有“**3.0 (quilt)**”的 **debian/source/format** 文件来升级至新的“**3.0 (quilt)**”格式。剩下的 **debian/*** 文件可以直接复制。如果需要的话，可以将“**filterdiff -z -x */debian/* foo.diff.gz > big.diff**”命令生成的 **big.diff** 文件导入到你的 quilt 系统中。
- 如果它使用了其他的补丁系统，例如 **dpatch**、**dbs** 或者是带有 **-p0**、**-p1**、**-p2** 参数的 **cdbs**。请使用 **quilt** 包中的 **deb3** 脚本来转换它。
- 如果它使用了带有“**--with quilt**”选项的 **dh** 命令或者使用了 **dh_quilt_patch** 和 **dh_quilt_unpatch** 命令，请移除这些并且使其使用新的“**3.0 (quilt)**”格式。
- 如果您有一个不带 **foo.diff.gz** 文件的 **1.0** 格式的源码包，您可以通过创建包含“**3.0 (native)**”的 **debian/source/format** 文件，然后将其余的 **debian/*** 文件直接复制的方式来更新至新的“**3.0 (native)**”的源码格式。

您应该核对一下 DEP——Debian 增强提议 并且采用已接受的提议。
参见 ProjectsDebSrc3.0 以核对 Debian 工具链对新 Debian 源码格式的支持情况。

7.14 CDBS

Common Debian Build System （**CDBS**）是 **debhelper** 软件包的包装系统。**CDBS** 基于 Makefile 包含机制并且由 **debian/rules** 文件中设置的 **DEB_*** 变量配置。

在将 **dh** 命令引入第七版的 **debhelper** 软件包之前，**CDBS** 是创建简单干净的 **debian/rules** 文件的唯一方法。

For many simple packages, the **dh** command alone allows us to make a simple and clean **debian/rules** file now. It is desirable to keep the build system simple and clean by not using the superfluous **CDBS**.

注意

“**CDBS** 神奇地让我用更少的命令来完成工作”和“我不懂新的 **dh** 的语法”都不是您继续使用旧的 **CDBS** 系统的借口。

对于一些复杂的软件包，比如与 GNOME 相关的，当前的维护者有理由利用 **CDBS** 自动化完成他们的统一包装。如果是这种情况，请不要费心从 **CDBS** 转换为 **dh** 语法。

注意

如果您正在与维护 团队 合作，请遵循团队的既定惯例。

将软件包从 **CDBS** 转换为 **dh** 语法时，请使用以下内容作为参考：

- CDBS 文档
- The Common Debian Build System (CDBS), FOSDEM 2009

7.15 在 UTF-8 环境下构建

构建环境的默认语言环境是 **C**。

某些程序（如 Python3 的 **read** 函数）会根据区域设置改变行为。

Adding the following code to the **debian/rules** file ensures building the program under the **C.UTF-8** locale.

```
LC_ALL := C.UTF-8
export LC_ALL
```

7.16 UTF-8 转换

如果上游文档是用旧编码方案编码的，那么将它们转换为 UTF-8 是个好主意。

请使用 **libc-bin** 包中的 **iconv** 命令来转换纯文本文件的编码。

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

使用 **w3m**(1) 将 HTML 文件转换为 UTF-8 纯文本文件。执行此操作时，请确保在 UTF-8 语言环境下执行它。

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

在 **debian/rules** 文件的 **override_dh_*** 目标中运行这些脚本。

7.17 上传 orig.tar.gz

当您第一次向归档上传软件包时，您还需要包含原始的 **orig.tar.gz** 源码。

如果 Debian 修订码是 **1** 或者 **0**，这都是默认的。否则，您必须使用带有 **-sa** 选项的 **dpkg-buildpackage** 命令。

- **dpkg-buildpackage -sa**
- **debuild -sa**
- **pdebuild --debbuildopts -sa**
- **git-pbuilder -sa**
- 对于 **gbp buildpackage**，请编辑 **~/.gbp.conf** 文件。

提示

另一方面，**-sd** 选项将会强制排除原始的 **orig.tar.gz** 源码。

提示

Security uploads require including the **orig.tar.gz** file.

7.18 跳过的上传

如果当跳过上传时，你在 **debian/changelog** 中创建了多个条目，你必须创建一个包含自上次上传以来所有变更的 **debian/changelog** 文件。这可以通过指定 dpkg-buildpackage 选项 **-v** 以及上次上传的版本号，比如 *1.2* 来完成。

- **dpkg-buildpackage -v***1.2*
- **debuild -v***1.2*
- **pdebuild --debbuildopts -v***1.2*
- **git-pbuilder -v***1.2*
- 对于 **gbp buildpackage**，请编辑 **~/.gbp.conf** 文件。

7.19 高级打包

关于以下内容的提示可以在 **debhelper**(7) 手册页中找到：

- **debhelper** 工具在“**compat** <= 8”选项下不同的行为
- 在数种不同构建条件下构建多种二进制包
  - 制作上游源码的多个拷贝
  - 在 **override_dh_auto_configure** 目标中调用多个“**dh_auto_configure -S** …”指令
  - 在 **override_dh_auto_build** 目标中调用多个“**dh_auto_build -S** …”指令
  - 在 **override_dh_auto_install** 目标中调用多个“**dh_auto_install -S** …”指令
- 在 **debian/control** 中以“**Package-Type: udeb**”选项来构建 **udeb** 包（参见 Package-Type）
- 从引导进程中排除某些包（参见 BuildProfileSpec）
  - 在 **debian/control** 中的二进制包节中添加 **Build-Profiles** 字段
  - 在 **DEB_BUILD_PROFILES** 环境变量设置成相关配置文件名的条件下构建软件包

关于以下内容的提示可以在 **dpkg-source**(1) 手册页中找到：

- 多个上游源码包的命名约定
  - 软件包名 _ 版本**.orig.tar.gz**
  - 软件包名 _ 版本**.orig-**部件名**.tar.gz**
- 将 Debian 更改记录到上游源码包中
  - **dpkg-source --commit**

7.20 其他发行版

尽管上游的原始码有着所有构建 Debian 软件包所需的信息，找出使用何种选项的组合仍然不是一件简单的事。

此外，上游的包可能更专注于功能的增强，而并不那么重视向后兼容性等特性，这是 Debian 打包实践中的一个重要方面。

利用其他发行版的信息是解决上述问题的一种选择。

如果其他发行版是由 Debian 派生的，重新使用它是没有价值的。

如果其他发行版是基于 RPM 的发行版，请参见 Repackage src.rpm。

通过 **rget** 命令，可以下载并打开 **src.rpm** 文件。（请将 **rget** 脚本添加至 PATH 中）

rget 脚本

```
#!/bin/sh
FCSRPM=$(basename $1)
mkdir ${FCSRPM}; cd ${FCSRPM}/
wget $1
rpm2cpio ${FCSRPM} | cpio -dium
```

许多上游源码包包含 RPM 系统使用的、以 *packagename*.**spec** 或者 *packagename*.**spec.in** 命名的 SPEC 文件。这可以被用做 Debian 软件包的基点。

7.21 除错

When you face build problems or core dumps of generated binary programs, you need to resolve them yourself. That’s **debug**.

This is too deep a topic to describe here. So, let me just list few pointers and hints for some typical debug tools.

- 核心转储
  - “**man core**”

  - 更新“**/etc/security/limits.conf**”文件来包含以下代码：

```
* soft core unlimited
```

  - 在 **~/.bashrc** 中添加“**ulimit -c unlimited**”
  - 使用“**ulimit -a**”来检查
  - 按 **Ctrl-** 或者“**kill -ABRT** *PID*”来建立一个核心转储文件。
- **gdb** - The GNU Debugger
  - “**info gdb**”
  - 参见 **/usr/share/doc/gdb-doc/html/gdb/index.html** 中的“Debugging with GDB”
- **strace** - 跟踪系统调用和信号
  - 使用 **/usr/share/doc/strace/examples/** 中的 **strace-graph** 脚本来建立一个好看的树形图
  - “**man strace**”
- **ltrace** - 跟踪库调用
  - “**man ltrace**”
- “**sh -n** *script.sh*” - Shell 脚本的语法检查
- “**sh -x** *script.sh*” - 跟踪 Shell 脚本
- “**python -m py_compile** *script.py*” - Python 脚本的语法检查
- “**python -mtrace --trace** *script.py*” - 跟踪 Python 脚本
- “**perl -I ../libpath -c** *script.pl*” - Perl 脚本的语法检查
- “**perl -d:Trace** *script.pl*” - 跟踪 Perl 脚本
  - 安装 **libterm-readline-gnu-perl** 软件包或者同类型软件来添加输入行编辑功能与历史记录支持。
- **lsof** - 按进程列出打开的文件
  - “**man lsof**”

提示

script 命令能帮助记录控制台输出。

提示

在 **ssh** 命令中搭配使用 **screen** 和 **tmux** 命令，能够提供安全并且强健的远程连接终端。

提示

libreply-perl（新的）软件包和来自 **libdevel-repl-perl**（旧的）软件包的 **re.pl** 命令为 Perl 提供了一个类似 Python 和 Shell 的 REPL (=READ + EVAL + PRINT + LOOP) 环境。

提示

rlwrap 和 **rlfe** 命令为所有交互命令提供了输入行编辑功能。例如“**rlwrap dash -i**”。

Chapter 8

更多示例

有一句古老的拉丁谚语：“**fabricando fit faber**”（“熟能生巧”）。
强烈建议使用简单的包来练习和试验 Debian 打包的所有步骤。本章为您的练习提供了许多上游案例。
这也可以作为许多编程主题的介绍性示例。

- 在 POSIX shell，Python3 和 C 中编程。
- 使用图标图形创建桌面 GUI 程序启动器的方法。
- 将 CLI 转换为 GUI 命令。
- 转化使用 国际化和本地化: POSIX shell、Python3 和 C 源码的程序以使用 **gettext** 。
- 构建系统概述：Makefile、Python distutils、Autotools 以及 CMake。

请注意，Debian 对以下事项非常注意：

- 自由软件
- 操作系统的稳定性与安全性
- 通过以下方式以实现通用操作系统：
  - 上游源码的自由选择，
  - CPU 架构的自由选择，以及
  - 用户界面语言的自由选择。

在第 4 章中介绍的典型打包示例是本章节的先决条件。
在以下数小节中，有些细节被刻意模糊。请尝试阅读相关文件，并且尝试自行厘清它们。

> 提示
>
> 
>
> 打包示例的最佳来源就是目前的 Debian 归档本身。请使用“Debian 代码搜索”服务来查找相关示例。

8.1 挑选最好的模板

以下是从空目录由零开始构建一个简单的 Debian 软件包的示例。
这是一个很棒的平台，可以使您获得所有的模板文件，而不会使您正在处理的上游源码树变得一团糟。
让我们假设这个空目录为 **debhello-0.1**。

```
 $ mkdir debhello-0.1
 $ tree
.
└── debhello-0.1

1 directory, 0 files
```

让我们通过指定 **-x4** 选项来生成最大数量的模板文件。
让我们也使用“**-p** *debhello* **-t -u** *0.1* **-r** *1*”选项来制作缺失的上游原始码。

```
 $ debmake -t -p debhello -u 0.1 -r 1 -x4
I: set parameters
 ...
I: debmake -x "4" ...
I: creating => debian/control
I: creating => debian/copyright
I: substituting => /usr/share/debmake/extra0/rules
 ...
I: creating => debian/license-examples/GPL-2.0+
I: substituting => /usr/share/debmake/extra4/BSD-3-Clause
I: creating => debian/license-examples/BSD-3-Clause
I: substituting => /usr/share/debmake/extra4/Artistic-1.0
I: creating => debian/license-examples/Artistic-1.0
I: $ wrap-and-sort
```

我们来检查一下自动产生的模板文件。

```
 $ cd ..
 $ tree
.
├── debhello-0.1
│   └── debian
│       ├── README.Debian
│       ├── changelog
│       ├── clean
│       ├── compat
│       ├── control
│       ├── copyright
│       ├── debhello.bug-control.ex
│       ├── debhello.bug-presubj.ex
│       ├── debhello.bug-script.ex
│       ├── debhello.conffiles.ex
 ...
│       └── watch
├── debhello-0.1.tar.gz
└── debhello_0.1.orig.tar.gz -> debhello-0.1.tar.gz

5 directories, 51 files
```

现在，您可以复制 *debhello-0.1/***debian/** 目录下所有生成的模板文件到您的软件包中。

提示

通过使用 **-T** 选项（教程模式）调用 **debmake** 命令，可以使生成的模板文件更加详细。

8.2 No Makefile (shell, CLI)

Here is an example of creating a simple Debian package from a POSIX shell CLI program without its build system.
Let’s assume this upstream tarball to be **debhello-0.2.tar.gz**.
This type of source has no automated means and files must be installed manually.

```
 $ tar -xzmf debhello-0.2.tar.gz
 $ cd debhello-0.2
 $ sudo cp scripts/hello /bin/hello
 ...
```

Let’s get the source and make the Debian package.

Download debhello-0.2.tar.gz

```
 $ wget http://www.example.org/download/debhello-0.2.tar.gz
 ...
 $ tar -xzmf debhello-0.2.tar.gz
 $ tree
.
├── debhello-0.2
│   ├── LICENSE
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
└── debhello-0.2.tar.gz

4 directories, 6 files
```

Here, the POSIX shell script **hello** is a very simple one.

hello (v=0.2)

```
 $ cat debhello-0.2/scripts/hello
#!/bin/sh -e
echo "Hello from the shell!"
echo ""
echo -n "Type Enter to exit this program: "
read X
```

Here, **hello.desktop** supports the Desktop Entry Specification.

hello.desktop (v=0.2)

```
 $ cat debhello-0.2/data/hello.desktop
[Desktop Entry]
Name=Hello
Name[fr]=Bonjour
Comment=Greetings
Comment[fr]=Salutations
Type=Application
Keywords=hello
Exec=hello
Terminal=true
Icon=hello.png
Categories=Utility;
```

Here, **hello.png** is the icon graphics file.

Let’s package this with the **debmake** command. Here, the **-b’:sh’** option is used to specify that the generated binary package is a shell script.

```
 $ cd debhello-0.2
 $ debmake -b':sh'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="0.2", rev="1"
I: *** start packaging in "debhello-0.2". ***
I: provide debhello_0.2.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-0.2.tar.gz debhello_0.2.orig.tar.gz
I: pwd = "/path/to/debhello-0.2"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
...
```

Let’s inspect notable template files generated.

The source tree after the basic debmake execution. (v=0.2)

```
 $ cd ..
 $ tree
.
├── debhello-0.2
│   ├── LICENSE
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── debian
│   │   ├── README.Debian
│   │   ├── changelog
│   │   ├── compat
│   │   ├── control
│   │   ├── copyright
│   │   ├── patches
│   │   │   └── series
│   │   ├── rules
│   │   ├── source
│   │   │   ├── format
│   │   │   └── local-options
│   │   └── watch
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
├── debhello-0.2.tar.gz
└── debhello_0.2.orig.tar.gz -> debhello-0.2.tar.gz

7 directories, 17 files
```

debian/rules (template file, v=0.2):

```
 $ cat debhello-0.2/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
        dh $@
```

This is essentially the standard **debian/rules** file with the **dh** command. Since this is the script package, this template **debian/rules** file has no build flag related contents.

debian/control (template file, v=0.2):

```
 $ cat debhello-0.2/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Since this is the shell script package, the **debmake** command sets "**Architecture: all**" and "**Multi-Arch: foreign**". Also, it sets required **substvar** parameters as "**Depends: ${misc:Depends}**". These are explained in 第 5 章.

Since this upstream source lacks the upstream **Makefile**, that functionality needs to be provided by the maintainer. This upstream source contains only a script file and data files and no C source files; the **build** process can be skipped but the **install** process needs to be implemented. For this case, this is achieved cleanly by adding the **debian/install** and **debian/manpages** files without complicating the **debian/rules** file.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@
```

debian/control (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

警告

If you leave "**Section: unknown**"in the template **debian/control** file unchanged, the **lintian** error may cause a build failure.

debian/install (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/install
 ... hack, hack, hack, ...
 $ cat debhello-0.2/debian/install
data/hello.desktop usr/share/applications
data/hello.png usr/share/pixmaps
scripts/hello usr/bin
```

debian/manpages (maintainer version, v=0.2):

```
 $ vim debhello-0.2/debian/manpages
 ... hack, hack, hack, ...
```

```
 $ cat debhello-0.2/debian/manpages
man/hello.1
```

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

Template files under debian/. (v=0.2):

```
 $ tree debhello-0.2/debian
debhello-0.2/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── install
├── manpages
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 12 files
```

You can create a non-native Debian package using the **debuild** command (or its equivalents) in this source tree. The command output is very verbose and explains what it does as follows.

```
 $ cd debhello-0.2
 $ debuild
 dpkg-buildpackage -rfakeroot -us -uc -ui -i
 ...
 fakeroot debian/rules clean
dh clean
 ...
 debian/rules build
dh build
   dh_update_autotools_config
   dh_autoreconf
   create-stamp debian/debhelper-build-stamp
 fakeroot debian/rules binary
dh binary
   dh_testroot
   dh_prep
        rm -f -- debian/debhello.substvars
        rm -fr -- debian/.debhelper/generated/debhello/ debian/debhello/ debi...
 ...
 fakeroot debian/rules binary
dh binary
 ...
```

现在我们来看看成果如何。

The generated files of debhello version 0.2 by the debuild command:

```
 $ cd ..
 $ tree -FL 1
.
├── debhello-0.2/
├── debhello-0.2.tar.gz
├── debhello_0.2-1.debian.tar.xz
├── debhello_0.2-1.dsc
├── debhello_0.2-1_all.deb
├── debhello_0.2-1_amd64.build
├── debhello_0.2-1_amd64.buildinfo
├── debhello_0.2-1_amd64.changes
```

```
└── debhello_0.2.orig.tar.gz -> debhello-0.2.tar.gz

1 directory, 8 files
```

您可以看见生成的全部文件。

- The **debhello_0.2.orig.tar.gz** file is a symlink to the upstream tarball.
- The **debhello_0.2-1.debian.tar.xz** file contains the maintainer generated contents.
- The **debhello_0.2-1.dsc** file is the meta data file for the Debian source package.
- The **debhello_0.2-1_all.deb** file is the Debian binary package.
- The **debhello_0.2-1_amd64.build** file is the build log file.
- The **debhello_0.2-1_amd64.buildinfo** file is the meta data file generated by **dpkg-genbuildinfo**(1).
- The **debhello_0.2-1_amd64.changes** file is the meta data file for the Debian binary package.

The **debhello_0.2-1.debian.tar.xz** file contains the Debian changes to the upstream source as follows.
The compressed archive contents of debhello_0.2-1.debian.tar.xz:

```
 $ tar -tzf debhello-0.2.tar.gz
debhello-0.2/
debhello-0.2/LICENSE
debhello-0.2/data/
debhello-0.2/data/hello.desktop
debhello-0.2/data/hello.png
debhello-0.2/scripts/
debhello-0.2/scripts/hello
debhello-0.2/man/
debhello-0.2/man/hello.1
 $ tar --xz -tf debhello_0.2-1.debian.tar.xz
debian/
debian/README.Debian
debian/changelog
debian/compat
debian/control
debian/copyright
debian/install
debian/manpages
debian/patches/
debian/patches/series
debian/rules
debian/source/
debian/source/format
debian/watch
```

The **debhello_0.2-1_amd64.deb** file contains the files to be installed as follows.
The binary package contents of debhello_0.2-1_all.deb:

```
 $ dpkg -c debhello_0.2-1_all.deb
drwxr-xr-x root/root ...  ./
drwxr-xr-x root/root ...  ./usr/
drwxr-xr-x root/root ...  ./usr/bin/
-rwxr-xr-x root/root ...  ./usr/bin/hello
drwxr-xr-x root/root ...  ./usr/share/
drwxr-xr-x root/root ...  ./usr/share/applications/
-rw-r--r-- root/root ...  ./usr/share/applications/hello.desktop
drwxr-xr-x root/root ...  ./usr/share/doc/
drwxr-xr-x root/root ...  ./usr/share/doc/debhello/
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/README.Debian
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/changelog.Debian.gz
-rw-r--r-- root/root ...  ./usr/share/doc/debhello/copyright
drwxr-xr-x root/root ...  ./usr/share/man/
```

```
drwxr-xr-x root/root ...  ./usr/share/man/man1/
-rw-r--r-- root/root ...  ./usr/share/man/man1/hello.1.gz
drwxr-xr-x root/root ...  ./usr/share/pixmaps/
-rw-r--r-- root/root ...  ./usr/share/pixmaps/hello.png
```

Here is the generated dependency list of **debhello_0.2-1_all.deb**.

The generated dependency list of debhello_0.2-1_all.deb:

```
$ dpkg -f debhello_0.2-1_all.deb pre-depends depends recommends conflicts br...
```

8.3 Makefile (shell, CLI)

Here is an example of creating a simple Debian package from a POSIX shell CLI program using the **Makefile** as its build system.

Let's assume its upstream tarball to be **debhello-1.0.tar.gz**.

这一类源代码设计可以用这样的方式安装成为非系统文件:

```
$ tar -xzmf debhello-1.0.tar.gz
$ cd debhello-1.0
$ make install
```

Debian 的打包需要对“**make install**”流程进行改变，从而将文件安装至目标系统镜像所在位置，而非通常使用的 **/usr/local** 下的位置。

Let's get the source and make the Debian package.

Download debhello-1.0.tar.gz

```
$ wget http://www.example.org/download/debhello-1.0.tar.gz
...
$ tar -xzmf debhello-1.0.tar.gz
$ tree
.
├── debhello-1.0
│   ├── LICENSE
│   ├── Makefile
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
└── debhello-1.0.tar.gz

4 directories, 7 files
```

Here, the **Makefile** uses **$(DESTDIR)** and **$(prefix)** properly. All other files are the same as in 第 8.2 节 and most of the packaging activities are the same.

Makefile (v=1.0)

```
$ cat debhello-1.0/Makefile
prefix = /usr/local

all:
        : # do nothing

install:
        install -D scripts/hello \
                $(DESTDIR)$(prefix)/bin/hello
        install -m 644 -D data/hello.desktop \
                $(DESTDIR)$(prefix)/share/applications/hello.desktop
        install -m 644 -D data/hello.png \
                $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        install -m 644 -D man/hello.1 \
```

```
                $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
        : # do nothing

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello
        -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
        -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```
 $ cd debhello-1.0
 $ debmake -b':sh'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.0", rev="1"
I: *** start packaging in "debhello-1.0". ***
I: provide debhello_1.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.0.tar.gz debhello_1.0.orig.tar.gz
I: pwd = "/path/to/debhello-1.0"
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
...
```

Let's inspect the notable template files generated.

debian/rules (template file, v=1.0):

```
 $ cat debhello-1.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
        dh $@

#override_dh_auto_install:
#       dh_auto_install -- prefix=/usr

#override_dh_install:
#       dh_install --list-missing -X.pyc -X.pyo
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.0):

```
 $ vim debhello-1.0/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@

override_dh_auto_install:
        dh_auto_install -- prefix=/usr
```

Since this upstream source has the proper upstream **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

The **debian/control** file is exactly the same as the one in 第 8.2 节.
在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。
Template files under debian/. (v=1.0):

```
 $ tree debhello-1.0/debian
debhello-1.0/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 10 files
```

The rest of the packaging activities are practically the same as the ones in 第 8.2 节.

8.4 setup.py (Python3, CLI)

Here is an example of creating a simple Debian package from a Python3 CLI program using **setup.py** as its build system.
Let's assume its upstream tarball to be **debhello-1.1.tar.gz**.
这一类源代码设计可以用这样的方式安装成为非系统文件:

```
 $ tar -xzmf debhello-1.1.tar.gz
 $ cd debhello-1.1
 $ python3 setup.py install
```

Debian packaging requires changing the last line to "**python3 setup.py install --install-layout=deb**" to install files into the target system image location. This issue is automatically addressed when using the **dh** command for Debian packaging.
Let's get the source and make the Debian package.
Download debhello-1.1.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.1.tar.gz
 ...
 $ tar -xzmf debhello-1.1.tar.gz
 $ tree
.
├── debhello-1.1
│   ├── LICENSE
│   ├── MANIFEST.in
│   ├── PKG-INFO
│   ├── hello_py
│   │   └── __init__.py
│   ├── scripts
│   │   └── hello
│   └── setup.py
└── debhello-1.1.tar.gz

3 directories, 7 files
```

Here, the **hello** script and its associated **hello_py** module are as follows.
hello (v=1.1)

```
 $ cat debhello-1.1/scripts/hello
#!/usr/bin/python3
import hello_py
```

```
if __name__ == '__main__':
    hello_py.main()
```

hello_py/__init__.py (v=1.1)

```
 $ cat debhello-1.1/hello_py/__init__.py
#!/usr/bin/python3
def main():
    print('Hello Python3!')
    input("Press Enter to continue...")
    return

if __name__ == '__main__':
    main()
```

These are packaged using the Python distutils with the **setup.py** and **MANIFEST.in** files.

setup.py (v=1.1)

```
 $ cat debhello-1.1/setup.py
#!/usr/bin/python3
# vi:se ts=4 sts=4 et ai:
from distutils.core import setup

setup(name='debhello',
    version='4.0',
    description='Hello Python',
    long_description='Hello Python program.',
    author='Osamu Aoki',
    author_email='osamu@debian.org',
    url='http://people.debian.org/~osamu/',
    packages=['hello_py'],
    package_dir={'hello_py': 'hello_py'},
    scripts=['scripts/hello'],
    classifiers = ['Development Status :: 3 - Alpha',
        'Environment :: Console',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Natural Language :: English',
        'Operating System :: POSIX :: Linux',
        'Programming Language :: Python :: 3',
        'Topic :: Utilities',
    ],
    platforms   = 'POSIX',
    license     = 'MIT License'
)
```

MANIFEST.in (v=1.1)

```
 $ cat debhello-1.1/MANIFEST.in
include MANIFEST.in
include LICENSE
```

提示

Many modern Python packages are distributed using setuptools. Since setuptools is an enhanced alternative to distutils, this example is useful for them.

Let's package this with the **debmake** command. Here, the **-b':py3'** option is used to specify the generated binary package containing Python3 script and module files.

```
 $ cd debhello-1.1
 $ debmake -b':py3'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.1", rev="1"
I: *** start packaging in "debhello-1.1". ***
I: provide debhello_1.1.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.1.tar.gz debhello_1.1.orig.tar.gz
I: pwd = "/path/to/debhello-1.1"
I: parse binary package settings: :py3
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
...
```

Let's inspect the notable template files generated.

debian/rules (template file, v=1.1):

```
 $ cat debhello-1.1/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

This is essentially the standard **debian/rules** file with the **dh** command.

The use of the "**--with python3**" option invokes **dh_python3** to calculate Python dependencies, add maintainer scripts to byte compiled files, etc. See **dh_python3**(1).

The use of the "**--buildsystem=pybuild**" option invokes various build systems for requested Python versions in order to build modules and extensions. See **pybuild**(1).

debian/control (template file, v=1.1):

```
 $ cat debhello-1.1/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~), dh-python, python3-all
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>
X-Python3-Version: >= 3.2

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}, ${python3:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Since this is the Python3 package, the **debmake** command sets "**Architecture: all**" and "**Multi-Arch: foreign**". Also, it sets required **substvar** parameters as "**Depends: ${python3:Depends}, ${misc:Depends}**". These are explained in 第 5 章.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.1):

```
 $ vim debhello-1.1/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.1/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

debian/control (maintainer version, v=1.1):

```
 $ vim debhello-1.1/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.1/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~), dh-python, python3-all
Standards-Version: 4.1.3
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/
X-Python3-Version: >= 3.2

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}, ${python3:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

The **hello** command does not come with the upstream-provided manpage; let's add it as the maintainer.

debian/manpages etc. (maintainer version, v=1.1):

```
 $ vim debhello-1.1/debian/hello.1
 ... hack, hack, hack, ...
 $ vim debhello-1.1/debian/manpages
 ... hack, hack, hack, ...
 $ cat debhello-1.1/debian/manpages
debian/hello.1
```

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

The rest of the packaging activities are practically the same as the ones in 第 8.3 节.

Template files under debian/. (v=1.1):

```
 $ tree debhello-1.1/debian
debhello-1.1/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── hello.1
├── manpages
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 12 files
```

Here is the generated dependency list of **debhello_1.1-1_all.deb**.

The generated dependency list of debhello_1.1-1_all.deb:

```
 $ dpkg -f debhello_1.1-1_all.deb pre-depends depends recommends conflicts br...
Depends: python3:any (>= 3.2~)
```

8.5 Makefile (shell, GUI)

Here is an example of creating a simple Debian package from a POSIX shell GUI program using the **Makefile** as its build system.

This upstream is based on 第 8.3 节 with enhanced GUI support.

Let's assume its upstream tarball to be **debhello-1.2.tar.gz**.

Let's get the source and make the Debian package.

Download debhello-1.2.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.2.tar.gz
 ...
 $ tar -xzmf debhello-1.2.tar.gz
 $ tree
.
├── debhello-1.2
│   ├── LICENSE
│   ├── Makefile
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── scripts
│       └── hello
└── debhello-1.2.tar.gz

4 directories, 7 files
```

Here, the **hello** has been re-written to use the **zenity** command to make this a GTK+ GUI program.

hello (v=1.2)

```
 $ cat debhello-1.2/scripts/hello
#!/bin/sh -e
zenity --info --title "hello" --text "Hello from the shell!"
```

Here, the desktop file is updated to be **Terminal=false** as a GUI program.

hello.desktop (v=1.2)

```
 $ cat debhello-1.2/data/hello.desktop
[Desktop Entry]
Name=Hello
Name[fr]=Bonjour
Comment=Greetings
Comment[fr]=Salutations
Type=Application
Keywords=hello
Exec=hello
Terminal=false
Icon=hello.png
Categories=Utility;
```

All other files are the same as in 第 8.3 节.

Let's package this with the **debmake** command. Here, the **-b':sh'** option is used to specify that the generated binary package is a shell script.

```
 $ cd debhello-1.2
 $ debmake -b':sh'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.2", rev="1"
I: *** start packaging in "debhello-1.2". ***
I: provide debhello_1.2.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.2.tar.gz debhello_1.2.orig.tar.gz
I: pwd = "/path/to/debhello-1.2"
```

```
I: parse binary package settings: :sh
I: binary package=debhello Type=script / Arch=all M-A=foreign
...
```

Let's inspect the notable template files generated.
debian/control (template file, v=1.2):

```
 $ cat debhello-1.2/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Let's make this Debian package better as the maintainer.
debian/control (maintainer version, v=1.2):

```
 $ vim debhello-1.2/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.2/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~)
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: zenity, ${misc:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

Please note the manually added **zenity** dependency.
The **debian/rules** file is exactly the same as the one in 第 8.3 节.
在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。
Template files under debian/. (v=1.2):

```
 $ tree debhello-1.2/debian
debhello-1.2/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
```

```
│   ├── format
│   └── local-options
└── watch

2 directories, 10 files
```

The rest of the packaging activities are practically the same as in 第 8.3 节.
Here is the generated dependency list of **debhello_1.2-1_all.deb**.
The generated dependency list of debhello_1.2-1_all.deb:

```
 $ dpkg -f debhello_1.2-1_all.deb pre-depends depends recommends conflicts br...
Depends: zenity
```

8.6 setup.py（Python3，图形界面）

Here is an example of creating a simple Debian package from a Python3 GUI program using the **setup.py** as its build system.
This upstream is based on 第 8.4 节 with enhanced GUI, desktop icon, and manpage support.
Let's assume this upstream tarball to be **debhello-1.3.tar.gz**.
Let's get the source and make the Debian package.
Download debhello-1.3.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.3.tar.gz
 ...
 $ tar -xzmf debhello-1.3.tar.gz
 $ tree
.
├── debhello-1.3
│   ├── LICENSE
│   ├── MANIFEST.in
│   ├── PKG-INFO
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── hello_py
│   │   └── __init__.py
│   ├── man
│   │   └── hello.1
│   ├── scripts
│   │   └── hello
│   └── setup.py
└── debhello-1.3.tar.gz

5 directories, 10 files
```

Here are the upstream sources.
hello (v=1.3)

```
 $ cat debhello-1.3/scripts/hello
#!/usr/bin/python3
import hello_py

if __name__ == '__main__':
    hello_py.main()
```

hello_py/__init__.py (v=1.3)

```
 $ cat debhello-1.3/hello_py/__init__.py
#!/usr/bin/python3
from gi.repository import Gtk

class TopWindow(Gtk.Window):
```

```
    def __init__(self):
        Gtk.Window.__init__(self)
        self.title = "Hello World!"
        self.counter = 0
        self.border_width = 10
        self.set_default_size(400, 100)
        self.set_position(Gtk.WindowPosition.CENTER)
        self.button = Gtk.Button(label="Click me!")
        self.button.connect("clicked", self.on_button_clicked)
        self.add(self.button)
        self.connect("delete-event", self.on_window_destroy)

    def on_window_destroy(self, *args):
        Gtk.main_quit(*args)

    def on_button_clicked(self, widget):
        self.counter += 1
        widget.set_label("Hello, World!\nClick count = %i" % self.counter)

def main():
    window = TopWindow()
    window.show_all()
    Gtk.main()

if __name__ == '__main__':
    main()
```

setup.py (v=1.3)

```
 $ cat debhello-1.3/setup.py
#!/usr/bin/python3
# vi:se ts=4 sts=4 et ai:
from distutils.core import setup

setup(name='debhello',
    version='4.1',
    description='Hello Python',
    long_description='Hello Python program.',
    author='Osamu Aoki',
    author_email='osamu@debian.org',
    url='http://people.debian.org/~osamu/',
    packages=['hello_py'],
    package_dir={'hello_py': 'hello_py'},
    scripts=['scripts/hello'],
    data_files=[
        ('share/applications', ['data/hello.desktop']),
        ('share/pixmaps', ['data/hello.png']),
        ('share/man/man1', ['man/hello.1']),
    ],
    classifiers = ['Development Status :: 3 - Alpha',
        'Environment :: Console',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Natural Language :: English',
        'Operating System :: POSIX :: Linux',
        'Programming Language :: Python :: 3',
        'Topic :: Utilities',
    ],
    platforms   = 'POSIX',
    license     = 'MIT License'
)
```

MANIFEST.in (v=1.3)

```
 $ cat debhello-1.3/MANIFEST.in
```

```
include MANIFEST.in
include LICENSE
include data/hello.deskto
include data/hello.png
include man/hello.1
```

Let's package this with the **debmake** command. Here, the **-b':py3'** option is used to specify that the generated binary package contains Python3 script and module files.

```
 $ cd debhello-1.3
 $ debmake -b':py3'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.3", rev="1"
I: *** start packaging in "debhello-1.3". ***
I: provide debhello_1.3.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.3.tar.gz debhello_1.3.orig.tar.gz
I: pwd = "/path/to/debhello-1.3"
I: parse binary package settings: :py3
I: binary package=debhello Type=python3 / Arch=all M-A=foreign
...
```

The result is practically the same as in 第 8.4 节.

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.3):

```
 $ vim debhello-1.3/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.3/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1

%:
        dh $@ --with python3 --buildsystem=pybuild
```

debian/control (maintainer version, v=1.3):

```
 $ vim debhello-1.3/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.3/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~), dh-python, python3-all
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc
X-Python3-Version: >= 3.2

Package: debhello
Architecture: all
Multi-Arch: foreign
Depends: gir1.2-gtk-3.0, python3-gi, ${misc:Depends}, ${python3:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

Please note the manually added **python3-gi** and **gir1.2-gtk-3.0** dependencies.

Since this upstream source has a manpage and other files with matching entries in the **setup.py** file, there is no need to create them and add the **debian/install** and **debian/manpages** files that were required in 第 8.4 节.

The rest of the packaging activities are practically the same as in 第 8.4 节.

Here is the generated dependency list of **debhello_1.3-1_all.deb**.

The generated dependency list of debhello_1.3-1_all.deb:

```
 $ dpkg -f debhello_1.3-1_all.deb pre-depends depends recommends conflicts br...
Depends: gir1.2-gtk-3.0, python3-gi, python3:any (>= 3.2~)
```

8.7 Makefile（单个二进制软件包）

Here is an example of creating a simple Debian package from a simple C source program using the **Makefile** as its build system.

This is an enhanced upstream source example for 第 4 章. This comes with the manpage, the desktop file, and the desktop icon. This also links to an external library **libm** to be a more practical example.

Let's assume this upstream tarball to be **debhello-1.4.tar.gz**.

这一类源代码设计可以用这样的方式安装成为非系统文件:

```
 $ tar -xzmf debhello-1.4.tar.gz
 $ cd debhello-1.4
 $ make
 $ make install
```

Debian packaging requires changing this "**make install**" process to install files into the target system image location instead of the normal location under **/usr/local**.

Let's get the source and make the Debian package.

Download debhello-1.4.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.4.tar.gz
 ...
 $ tar -xzmf debhello-1.4.tar.gz
 $ tree
.
├── debhello-1.4
│   ├── LICENSE
│   ├── Makefile
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── src
│       ├── config.h
│       └── hello.c
└── debhello-1.4.tar.gz

4 directories, 8 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.4):

```
 $ cat debhello-1.4/src/hello.c
#include "config.h"
#include <math.h>
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
        return 0;
}
```

src/config.h (v=1.4):

```
 $ cat debhello-1.4/src/config.h
#define PACKAGE_AUTHOR "Osamu Aoki"
```

Makefile (v=1.4):

```
 $ cat debhello-1.4/Makefile
prefix = /usr/local

all: src/hello

src/hello: src/hello.c
        $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -o $@ $^ -lm

install: src/hello
        install -D src/hello \
                $(DESTDIR)$(prefix)/bin/hello
        install -m 644 -D data/hello.desktop \
                $(DESTDIR)$(prefix)/share/applications/hello.desktop
        install -m 644 -D data/hello.png \
                $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        install -m 644 -D man/hello.1 \
                $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
        -rm -f src/hello

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello
        -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
        -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

Please note that this **Makefile** has the proper **install** target for the manpage, the desktop file, and the desktop icon.

Let’s package this with the **debmake** command.

```
 $ cd debhello-1.4
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.4", rev="1"
I: *** start packaging in "debhello-1.4". ***
I: provide debhello_1.4.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.4.tar.gz debhello_1.4.orig.tar.gz
I: pwd = "/path/to/debhello-1.4"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is practically the same as in 第 4.5 节.

Let’s make this Debian package, which is practically the same as in 第 4.6 节, better as the maintainer.

If the **DEB_BUILD_MAINT_OPTIONS** environment variable is not exported in **debian/rules**, lintian warns ”W: debhello: hardening-no-relro usr/bin/hello” for the linking of **libm**.

The **debian/control** file makes it exactly the same as the one in 第 4.6 节, since the **libm** library is always available as a part of **libc6** (Priority: required).

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

Template files under debian/. (v=1.4):

```
 $ tree debhello-1.4/debian
debhello-1.4/debian
├── README.Debian
├── changelog
├── compat
```

```
├── control
├── copyright
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 10 files
```

The rest of the packaging activities are practically the same as the one in 第 4.7 节.

Here is the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=1.4):

```
 $ dpkg -f debhello-dbgsym_1.4-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 1.4-1)
 $ dpkg -f debhello_1.4-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libc6 (>= 2.3.4)
```

8.8 Makefile.in + configure (single-binary package)

Here is an example of creating a simple Debian package from a simple C source program using **Makefile.in** and **configure** as its build system.

This is an enhanced upstream source example for 第 8.7 节. This also links to an external library, **libm**, and this source is configurable using arguments to the **configure** script, which generates the **Makefile** and **src/config.h** files.

Let's assume this upstream tarball to be **debhello-1.5.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
 $ tar -xzmf debhello-1.5.tar.gz
 $ cd debhello-1.5
 $ ./configure --with-math
 $ make
 $ make install
```

Let's get the source and make the Debian package.

Download debhello-1.5.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.5.tar.gz
 ...
 $ tar -xzmf debhello-1.5.tar.gz
 $ tree
.
├── debhello-1.5
│   ├── LICENSE
│   ├── Makefile.in
│   ├── configure
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   └── hello.1
│   └── src
│       └── hello.c
└── debhello-1.5.tar.gz

4 directories, 8 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.5):

```
 $ cat debhello-1.5/src/hello.c
#include "config.h"
#ifdef WITH_MATH
#  include <math.h>
#endif
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
        printf("I can't do MATH!\n");
#endif
        return 0;
}
```

Makefile.in (v=1.5):

```
 $ cat debhello-1.5/Makefile.in
prefix = @prefix@

all: src/hello

src/hello: src/hello.c
        $(CC) @VERBOSE@ \
                $(CPPFLAGS) \
                $(CFLAGS) \
                $(LDFLAGS) \
                -o $@ $^ \
                @LINKLIB@

install: src/hello
        install -D src/hello \
                $(DESTDIR)$(prefix)/bin/hello
        install -m 644 -D data/hello.desktop \
                $(DESTDIR)$(prefix)/share/applications/hello.desktop
        install -m 644 -D data/hello.png \
                $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        install -m 644 -D man/hello.1 \
                $(DESTDIR)$(prefix)/share/man/man1/hello.1

clean:
        -rm -f src/hello

distclean: clean

uninstall:
        -rm -f $(DESTDIR)$(prefix)/bin/hello
        -rm -f $(DESTDIR)$(prefix)/share/applications/hello.desktop
        -rm -f $(DESTDIR)$(prefix)/share/pixmaps/hello.png
        -rm -f $(DESTDIR)$(prefix)/share/man/man1/hello.1

.PHONY: all install clean distclean uninstall
```

configure (v=1.5):

```
 $ cat debhello-1.5/configure
#!/bin/sh -e
# default values
PREFIX="/usr/local"
VERBOSE=""
WITH_MATH="0"
LINKLIB=""
```

```
PACKAGE_AUTHOR="John Doe"

# parse arguments
while [ "${1}" != "" ]; do
  VAR="${1%=*}" # Drop suffix =*
  VAL="${1#*=}" # Drop prefix *=
  case "${VAR}" in
  --prefix)
    PREFIX="${VAL}"
    ;;
  --verbose|-v)
    VERBOSE="-v"
    ;;
  --with-math)
    WITH_MATH="1"
    LINKLIB="-lm"
    ;;
  --author)
    PACKAGE_AUTHOR="${VAL}"
    ;;
  *)
    echo "W: Unknown argument: ${1}"
  esac
  shift
done

# setup configured Makefile and src/config.h
sed -e "s,@prefix@,${PREFIX}," \
    -e "s,@VERBOSE@,${VERBOSE}," \
    -e "s,@LINKLIB@,${LINKLIB}," \
    <Makefile.in >Makefile
if [ "${WITH_MATH}" = 1 ]; then
echo "#define WITH_MATH" >src/config.h
else
echo "/* not defined: WITH_MATH */" >src/config.h
fi
echo "#define PACKAGE_AUTHOR \"${PACKAGE_AUTHOR}\"" >>src/config.h
```

Please note that the **configure** command replaces strings with **@**···**@** in **Makefile.in** to produce **Makefile** and creates **src/config.h**.

Let's package this with the **debmake** command.

```
 $ cd debhello-1.5
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.5", rev="1"
I: *** start packaging in "debhello-1.5". ***
I: provide debhello_1.5.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.5.tar.gz debhello_1.5.orig.tar.gz
I: pwd = "/path/to/debhello-1.5"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to 第 4.5 节 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=1.5):

```
 $ cat debhello-1.5/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
```

```
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@
```

Let’s make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.5):

```
 $ vim debhello-1.5/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.5/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- \
                --with-math \
                --author="Osamu Aoki"
```

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

The rest of the packaging activities are practically the same as the one in 第 4.7 节.

8.9 Autotools (single-binary package)

Here is an example of creating a simple Debian package from a simple C source program using Autotools = Autoconf and Automake (**Makefile.am** and **configure.ac**) as its build system. See 第 5.16.1 节.

This source usually comes with the upstream auto-generated **Makefile.in** and **configure** files, too. This source can be packaged using these files as in 第 8.8 节 with the help of the **autotools-dev** package.

The better alternative is to regenerate these files using the latest Autoconf and Automake packages if the upstream provided **Makefile.am** and **configure.ac** are compatible with the latest version. This is advantageous for porting to new CPU architectures, etc. This can be automated by using the “**--with autoreconf**” option for the **dh** command.

Let’s assume this upstream tarball to be **debhello-1.6.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
 $ tar -xzmf debhello-1.6.tar.gz
 $ cd debhello-1.6
 $ autoreconf -ivf # optional
 $ ./configure --with-math
 $ make
 $ make install
```

Let’s get the source and make the Debian package.

Download debhello-1.6.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.6.tar.gz
 ...
 $ tar -xzmf debhello-1.6.tar.gz
 $ tree
.
├── debhello-1.6
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
```

```
|   ├── man
|   |   ├── Makefile.am
|   |   └── hello.1
|   └── src
|       ├── Makefile.am
|       └── hello.c
└── debhello-1.6.tar.gz

4 directories, 9 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.6):

```
 $ cat debhello-1.6/src/hello.c
#include "config.h"
#ifdef WITH_MATH
#  include <math.h>
#endif
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
        printf("I can't do MATH!\n");
#endif
        return 0;
}
```

Makefile.am (v=1.6):

```
 $ cat debhello-1.6/Makefile.am
SUBDIRS = src man
 $ cat debhello-1.6/man/Makefile.am
dist_man_MANS = hello.1
 $ cat debhello-1.6/src/Makefile.am
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

configure.ac (v=1.6):

```
 $ cat debhello-1.6/configure.ac
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.1],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign])
# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
echo "Add --with-math option functionality to ./configure"
AC_ARG_WITH([math],
  [AS_HELP_STRING([--with-math],
    [compile with math library  @<:@default=yes@:>@])],
  [],
  [with_math="yes"]
  )
echo "==== withval   := \"$withval\""
echo "==== with_math := \"$with_math\""
# m4sh if-else construct
AS_IF([test "x$with_math" != "xno"],[
```

```
  echo "==== Check include: math.h"
  AC_CHECK_HEADER(math.h,[],[
    AC_MSG_ERROR([Couldn't find math.h.] )
  ])
  echo "==== Check library: libm"
  AC_SEARCH_LIBS(atan, [m])
  #AC_CHECK_LIB(m, atan)
  echo "==== Build with LIBS := \"$LIBS\""
  AC_DEFINE(WITH_MATH, [1], [Build with the math library])
],[
  echo "==== Skip building with math.h."
  AH_TEMPLATE(WITH_MATH, [Build without the math library])
])
# Checks for programs.
AC_PROG_CC
AC_CONFIG_FILES([Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

提示

Without “**foreign**”strictness level specified in **AM_INIT_AUTOMAKE()** as above, **automake** defaults to “**gnu**”strictness level requiring several files in the top-level directory. See “3.2 Strictness”in the **automake** document.

Let’s package this with the **debmake** command.

```
 $ cd debhello-1.6
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.6", rev="1"
I: *** start packaging in "debhello-1.6". ***
I: provide debhello_1.6.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.6.tar.gz debhello_1.6.orig.tar.gz
I: pwd = "/path/to/debhello-1.6"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to 第 8.8 节 but not exactly the same.

Let’s inspect the notable template files generated.

debian/rules (template file, v=1.6):

```
 $ cat debhello-1.6/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

#override_dh_install:
#       dh_install --list-missing -X.la -X.pyc -X.pyo
```

Let’s make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.6):

```
 $ vim debhello-1.6/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.6/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

override_dh_auto_configure:
        dh_auto_configure -- \
                --with-math
```

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。
The rest of the packaging activities are practically the same as the one in 第 4.7 节.

8.10 CMake（单个二进制软件包）

Here is an example of creating a simple Debian package from a simple C source program using CMake (**CMakeLists.txt** and some files such as **config.h.in**) as its build system. See 第 5.16.2 节.

The **cmake** command generates the **Makefile** file based on the **CMakeLists.txt** file and its **-D** option. It also configures the file as specified in its **configure_file(⋯)** by replacing strings with **@⋯@** and changing the **#cmakedefine** ⋯ line.

Let’s assume this upstream tarball to be **debhello-1.7.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
 $ tar -xzmf debhello-1.7.tar.gz
 $ cd debhello-1.7
 $ mkdir obj-x86_64-linux-gnu # for out-of-tree build
 $ cd obj-x86_64-linux-gnu
 $ cmake ..
 $ make
 $ make install
```

Let’s get the source and make the Debian package.

Download debhello-1.7.tar.gz

```
 $ wget http://www.example.org/download/debhello-1.7.tar.gz
 ...
 $ tar -xzmf debhello-1.7.tar.gz
 $ tree
.
├── debhello-1.7
│   ├── CMakeLists.txt
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── man
│   │   ├── CMakeLists.txt
│   │   └── hello.1
│   └── src
│       ├── CMakeLists.txt
│       ├── config.h.in
│       └── hello.c
└── debhello-1.7.tar.gz

4 directories, 9 files
```

Here, the contents of this source are as follows.

src/hello.c (v=1.7):

```
 $ cat debhello-1.7/src/hello.c
#include "config.h"
#ifdef WITH_MATH
#  include <math.h>
#endif
#include <stdio.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
#ifdef WITH_MATH
        printf("4.0 * atan(1.0) = %10f8\n", 4.0*atan(1.0));
#else
        printf("I can't do MATH!\n");
#endif
        return 0;
}
```

src/config.h.in (v=1.7):

```
 $ cat debhello-1.7/src/config.h.in
/* name of the package author */
#define PACKAGE_AUTHOR "@PACKAGE_AUTHOR@"
/* math library support */
#cmakedefine WITH_MATH
```

CMakeLists.txt (v=1.7):

```
 $ cat debhello-1.7/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(src)
add_subdirectory(man)
 $ cat debhello-1.7/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
 $ cat debhello-1.7/src/CMakeLists.txt
# Always define HAVE_CONFIG_H
add_definitions(-DHAVE_CONFIG_H)
# Interactively define WITH_MATH
option(WITH_MATH "Build with math support" OFF)
#variable_watch(WITH_MATH)
# Generate config.h from config.h.in
configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/config.h.in"
  "${CMAKE_CURRENT_BINARY_DIR}/config.h"
)
include_directories("${CMAKE_CURRENT_BINARY_DIR}")
add_executable(hello hello.c)
install(TARGETS hello
  RUNTIME DESTINATION bin
)
```

Let’s package this with the **debmake** command.

```
 $ cd debhello-1.7
 $ debmake
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="1.7", rev="1"
```

```
I: *** start packaging in "debhello-1.7". ***
I: provide debhello_1.7.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-1.7.tar.gz debhello_1.7.orig.tar.gz
I: pwd = "/path/to/debhello-1.7"
I: parse binary package settings:
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to 第 8.8 节 but not exactly the same.

Let's inspect the notable template files generated.

debian/rules (template file, v=1.7):

```
 $ cat debhello-1.7/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

#override_dh_auto_configure:
#       dh_auto_configure -- \
#             -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
```

debian/control (template file, v=1.7):

```
 $ cat debhello-1.7/debian/control
Source: debhello
Section: unknown
Priority: optional
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: cmake, debhelper (>=11~)
Standards-Version: 4.1.4
Homepage: <insert the upstream URL, if relevant>

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: auto-generated package by debmake
 This Debian binary package was auto-generated by the
 debmake(1) command provided by the debmake package.
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=1.7):

```
 $ vim debhello-1.7/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-1.7/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- -DWITH-MATH=1
```

debian/control (maintainer version, v=1.7):

```
 $ vim debhello-1.7/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-1.7/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper (>=11~)
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example package in the debmake-doc package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
```

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。
The rest of the packaging activities are practically the same as the one in 第 8.8 节.

8.11 Autotools（多个二进制软件包）

Here is an example of creating a set of Debian binary packages including the executable package, the shared library package, the development file package, and the debug symbol package from a simple C source program using Autotools = Autoconf and Automake (which use **Makefile.am** and **configure.ac** as their input files) as its build system. See 第 5.16.1 节.

Let’s package this in the same way as in 第 8.9 节.
Let’s assume this upstream tarball to be **debhello-2.0.tar.gz**.
This type of source is meant to be installed as a non-system file, for example, as:

```
$ tar -xzmf debhello-2.0.tar.gz
$ cd debhello-2.0
$ autoreconf -ivf # optional
$ ./configure --with-math
$ make
$ make install
```

Let’s get the source and make the Debian package.
Download debhello-2.0.tar.gz

```
 $ wget http://www.example.org/download/debhello-2.0.tar.gz
 ...
 $ tar -xzmf debhello-2.0.tar.gz
 $ tree
.
├── debhello-2.0
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── lib
│   │   ├── Makefile.am
│   │   ├── sharedlib.c
│   │   └── sharedlib.h
│   ├── man
│   │   ├── Makefile.am
```

```
|   |   └── hello.1
|   └── src
|       ├── Makefile.am
|       └── hello.c
└── debhello-2.0.tar.gz

5 directories, 12 files
```

Here, the contents of this source are as follows.

src/hello.c (v=2.0):

```
 $ cat debhello-2.0/src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
        sharedlib();
        return 0;
}
```

lib/sharedlib.h and lib/sharedlib.c (v=1.6):

```
 $ cat debhello-2.0/lib/sharedlib.h
int sharedlib();
 $ cat debhello-2.0/lib/sharedlib.c
#include <stdio.h>
int
sharedlib()
{
        printf("This is a shared library!\n");
        return 0;
}
```

Makefile.am (v=2.0):

```
 $ cat debhello-2.0/Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = lib src man
 $ cat debhello-2.0/man/Makefile.am
# manpages (distributed in the source package)
dist_man_MANS = hello.1
 $ cat debhello-2.0/lib/Makefile.am
# libtool librares to be produced
lib_LTLIBRARIES = libsharedlib.la

# source files used for lib_LTLIBRARIES
libsharedlib_la_SOURCES = sharedlib.c

# C pre-processor flags used for lib_LTLIBRARIES
#libsharedlib_la_CPPFLAGS =

# Headers files to be installed in <prefix>/include
include_HEADERS = sharedlib.h

# Versioning Libtool Libraries with version triplets
libsharedlib_la_LDFLAGS = -version-info 1:0:0
 $ cat debhello-2.0/src/Makefile.am
# program executables to be produced
bin_PROGRAMS = hello

# source files used for bin_PROGRAMS
hello_SOURCES = hello.c
```

```
# C pre-processor flags used for bin_PROGRAMS
AM_CPPFLAGS = -I$(srcdir) -I$(top_srcdir)/lib

# Extra options for the linker for hello
# hello_LDFLAGS =

# Libraries the `hello` binary to be linked
hello_LDADD = $(top_srcdir)/lib/libsharedlib.la
```

configure.ac (v=2.0):

```
 $ cat debhello-2.0/configure.ac
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.2],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])

AM_INIT_AUTOMAKE([foreign])

# Set default to --enable-shared --disable-static
LT_INIT([shared disable-static])

# find the libltdl sources in the libltdl sub-directory
LT_CONFIG_LTDL_DIR([libltdl])

# choose one
LTDL_INIT([recursive])
#LTDL_INIT([subproject])
#LTDL_INIT([nonrecursive])

# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
# Checks for programs.
AC_PROG_CC

# only for the recursive case
AC_CONFIG_FILES([Makefile
                 lib/Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

Let’s package this with the **debmake** command into multiple packages:

- **debhello**: type = **bin**
- **libsharedlib1**: type = **lib**
- **libsharedlib-dev**: type = **dev**

Here, the **-b’,libsharedlib1,libsharedlib-dev’** option is used to specify the generated binary packages.

```
 $ cd debhello-2.0
 $ debmake -b',libsharedlib1,libsharedlib-dev'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="2.0", rev="1"
I: *** start packaging in "debhello-2.0". ***
I: provide debhello_2.0.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-2.0.tar.gz debhello_2.0.orig.tar.gz
I: pwd = "/path/to/debhello-2.0"
```

```
I: parse binary package settings: ,libsharedlib1,libsharedlib-dev
I: binary package=debhello Type=bin / Arch=any M-A=foreign
I: binary package=libsharedlib1 Type=lib / Arch=any M-A=same
I: binary package=libsharedlib-dev Type=dev / Arch=any M-A=same
I: analyze the source tree
I: build_type = Autotools with autoreconf
...
```

The result is similar to 第 8.8 节 but with more template files.

Let's inspect the notable template files generated.

debian/rules (template file, v=2.0):

```
 $ cat debhello-2.0/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

#override_dh_install:
#       dh_install --list-missing -X.la -X.pyc -X.pyo
```

Let's make this Debian package better as the maintainer.

debian/rules (maintainer version, v=2.0):

```
 $ vim debhello-2.0/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@ --with autoreconf

override_dh_install:
        dh_install --list-missing -X.la
```

debian/control (maintainer version, v=2.0):

```
 $ vim debhello-2.0/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=11~), dh-autoreconf
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: libsharedlib1 (= ${binary:Version}),
         ${misc:Depends},
         ${shlibs:Depends}
Description: example executable package
 This is an example package to demonstrate Debian packaging using
```

```
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package provides the executable program.

Package: libsharedlib1
Section: libs
Architecture: any
Multi-Arch: same
Pre-Depends: ${misc:Pre-Depends}
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example shared library package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the shared library.

Package: libsharedlib-dev
Section: libdevel
Architecture: any
Multi-Arch: same
Depends: libsharedlib1 (= ${binary:Version}), ${misc:Depends}
Description: example development package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the development files.
```

debian/*.install (maintainer version, v=2.0):

```
 $ vim debhello-2.0/debian/debhello.install
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/debhello.install
usr/bin/*
usr/share/man/*
 $ vim debhello-2.0/debian/libsharedlib1.install
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/libsharedlib1.install
usr/lib/*/*.so.*
 $ vim debhello-2.0/debian/libsharedlib-dev.install
 ... hack, hack, hack, ...
 $ cat debhello-2.0/debian/libsharedlib-dev.install
###usr/lib/*/pkgconfig/*.pc
usr/include
usr/lib/*/*.so
```

Since this upstream source creates the proper auto-generated **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

Template files under debian/. (v=2.0):

```
 $ tree debhello-2.0/debian
debhello-2.0/debian
├── README.Debian
├── changelog
├── compat
├── control
```

```
├── copyright
├── debhello.install
├── libsharedlib-dev.install
├── libsharedlib1.install
├── libsharedlib1.symbols
├── patches
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 14 files
```

The rest of the packaging activities are practically the same as the one in 第 8.8 节.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.0):

```
 $ dpkg -f debhello-dbgsym_2.0-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 2.0-1)
 $ dpkg -f debhello_2.0-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libsharedlib1 (= 2.0-1), libc6 (>= 2.2.5)
 $ dpkg -f libsharedlib-dev_2.0-1_amd64.deb pre-depends depends recommends co...
Depends: libsharedlib1 (= 2.0-1)
 $ dpkg -f libsharedlib1-dbgsym_2.0-1_amd64.deb pre-depends depends recommend...
Depends: libsharedlib1 (= 2.0-1)
 $ dpkg -f libsharedlib1_2.0-1_amd64.deb pre-depends depends recommends confl...
Depends: libc6 (>= 2.2.5)
```

8.12 CMake（多个二进制软件包）

Here is an example of creating a set of Debian binary packages including the executable package, the shared library package, the development file package, and the debug symbol package from a simple C source program using CMake (**CMakeLists.txt** and some files such as **config.h.in**) as its build system. See 第 5.16.2 节.

Let’s assume this upstream tarball to be **debhello-2.1.tar.gz**.

This type of source is meant to be installed as a non-system file, for example, as:

```
 $ tar -xzmf debhello-2.1.tar.gz
 $ cd debhello-2.1
 $ mkdir obj-x86_64-linux-gnu
 $ cd obj-x86_64-linux-gnu
 $ cmake ..
 $ make
 $ make install
```

Let’s get the source and make the Debian package.

Download debhello-2.1.tar.gz

```
 $ wget http://www.example.org/download/debhello-2.1.tar.gz
 ...
 $ tar -xzmf debhello-2.1.tar.gz
 $ tree
.
├── debhello-2.1
│   ├── CMakeLists.txt
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── lib
│   │   ├── CMakeLists.txt
│   │   ├── sharedlib.c
│   │   └── sharedlib.h
```

```
|   ├── man
|   |   ├── CMakeLists.txt
|   |   └── hello.1
|   └── src
|       ├── CMakeLists.txt
|       ├── config.h.in
|       └── hello.c
└── debhello-2.1.tar.gz

5 directories, 12 files
```

Here, the contents of this source are as follows.

src/hello.c (v=2.1):

```
 $ cat debhello-2.1/src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
int
main()
{
        printf("Hello, I am " PACKAGE_AUTHOR "!\n");
        sharedlib();
        return 0;
}
```

src/config.h.in (v=2.1):

```
 $ cat debhello-2.1/src/config.h.in
/* name of the package author */
#define PACKAGE_AUTHOR "@PACKAGE_AUTHOR@"
```

lib/sharedlib.c and lib/sharedlib.h (v=2.1):

```
 $ cat debhello-2.1/lib/sharedlib.h
int sharedlib();
 $ cat debhello-2.1/lib/sharedlib.c
#include <stdio.h>
int
sharedlib()
{
        printf("This is a shared library!\n");
        return 0;
}
```

CMakeLists.txt (v=2.1):

```
 $ cat debhello-2.1/CMakeLists.txt
cmake_minimum_required(VERSION 2.8)
project(debhello)
set(PACKAGE_AUTHOR "Osamu Aoki")
add_subdirectory(lib)
add_subdirectory(src)
add_subdirectory(man)
 $ cat debhello-2.1/man/CMakeLists.txt
install(
  FILES ${CMAKE_CURRENT_SOURCE_DIR}/hello.1
  DESTINATION share/man/man1
)
 $ cat debhello-2.1/src/CMakeLists.txt
# Always define HAVE_CONFIG_H
add_definitions(-DHAVE_CONFIG_H)
# Generate config.h from config.h.in
configure_file(
  "${CMAKE_CURRENT_SOURCE_DIR}/config.h.in"
  "${CMAKE_CURRENT_BINARY_DIR}/config.h"
```

```
  )
include_directories("${CMAKE_CURRENT_BINARY_DIR}")
include_directories("${CMAKE_SOURCE_DIR}/lib")

add_executable(hello hello.c)
target_link_libraries(hello sharedlib)
install(TARGETS hello
  RUNTIME DESTINATION bin
)
```

Let’s package this with the **debmake** command.

```
 $ cd debhello-2.1
 $ debmake -b',libsharedlib1,libsharedlib-dev'
I: set parameters
I: sanity check of parameters
I: pkg="debhello", ver="2.1", rev="1"
I: *** start packaging in "debhello-2.1". ***
I: provide debhello_2.1.orig.tar.gz for non-native Debian package
I: pwd = "/path/to"
I: $ ln -sf debhello-2.1.tar.gz debhello_2.1.orig.tar.gz
I: pwd = "/path/to/debhello-2.1"
I: parse binary package settings: ,libsharedlib1,libsharedlib-dev
I: binary package=debhello Type=bin / Arch=any M-A=foreign
...
```

The result is similar to 第 8.8 节 but not exactly the same.

Let’s inspect the notable template files generated.

debian/rules (template file, v=2.1):

```
 $ cat debhello-2.1/debian/rules
#!/usr/bin/make -f
# You must remove unused comment lines for the released package.
#export DH_VERBOSE = 1
#export DEB_BUILD_MAINT_OPTIONS = hardening=+all
#export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
#export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

%:
        dh $@

#override_dh_auto_configure:
#       dh_auto_configure -- \
#             -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_TARGET_MULTIARCH)"
```

Let’s make this Debian package better as the maintainer.

debian/rules (maintainer version, v=2.1):

```
 $ vim debhello-2.1/debian/rules
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/rules
#!/usr/bin/make -f
export DH_VERBOSE = 1
export DEB_BUILD_MAINT_OPTIONS = hardening=+all
export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)

%:
        dh $@

override_dh_auto_configure:
        dh_auto_configure -- \
              -DCMAKE_LIBRARY_ARCHITECTURE="$(DEB_HOST_MULTIARCH)"
```

```
override_dh_install:
        dh_install --list-missing
```

debian/control (maintainer version, v=2.1):

```
 $ vim debhello-2.1/debian/control
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/control
Source: debhello
Section: devel
Priority: optional
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper (>=11~)
Standards-Version: 4.1.3
Homepage: https://salsa.debian.org/debian/debmake-doc

Package: debhello
Architecture: any
Multi-Arch: foreign
Depends: libsharedlib1 (= ${binary:Version}),
         ${misc:Depends},
         ${shlibs:Depends}
Description: example executable package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package provides the executable program.

Package: libsharedlib1
Section: libs
Architecture: any
Multi-Arch: same
Pre-Depends: ${misc:Pre-Depends}
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: example shared library package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the shared library.

Package: libsharedlib-dev
Section: libdevel
Architecture: any
Multi-Arch: same
Depends: libsharedlib1 (= ${binary:Version}), ${misc:Depends}
Description: example development package
 This is an example package to demonstrate Debian packaging using
 the debmake command.
 .
 The generated Debian package uses the dh command offered by the
 debhelper package and the dpkg source format `3.0 (quilt)'.
 .
 This package contains the development files.
```

debian/*.install (maintainer version, v=2.1):

```
 $ vim debhello-2.1/debian/debhello.install
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/debhello.install
```

```
usr/bin/*
usr/share/man/*
 $ vim debhello-2.1/debian/libsharedlib1.install
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/libsharedlib1.install
usr/lib/*/*.so.*
 $ vim debhello-2.1/debian/libsharedlib-dev.install
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/libsharedlib-dev.install
###usr/lib/*/pkgconfig/*.pc
usr/include
usr/lib/*/*.so
```

This upstream CMakeList.txt needs to be patched to cope with the multiarch path.

debian/patches/* (maintainer version, v=2.1):

```
 ... hack, hack, hack, ...
 $ cat debhello-2.1/debian/libsharedlib1.symbols
libsharedlib.so.1 libsharedlib1 #MINVER#
 sharedlib@Base 2.1
```

Since this upstream source creates the proper auto-generated **Makefile**, there is no need to create **debian/install** and **debian/manpages** files.

在 **debian/** 目录下还有一些其它的模板文件。它们也需要进行更新。

Template files under debian/. (v=2.1):

```
 $ tree debhello-2.1/debian
debhello-2.1/debian
├── README.Debian
├── changelog
├── compat
├── control
├── copyright
├── debhello.install
├── libsharedlib-dev.install
├── libsharedlib1.install
├── libsharedlib1.symbols
├── patches
│   ├── 000-cmake-multiarch.patch
│   └── series
├── rules
├── source
│   ├── format
│   └── local-options
└── watch

2 directories, 15 files
```

The rest of the packaging activities are practically the same as the one in 第 8.8 节.

Here are the generated dependency list of all binary packages.

The generated dependency list of all binary packages (v=2.1):

```
 $ dpkg -f debhello-dbgsym_2.1-1_amd64.deb pre-depends depends recommends con...
Depends: debhello (= 2.1-1)
 $ dpkg -f debhello_2.1-1_amd64.deb pre-depends depends recommends conflicts ...
Depends: libsharedlib1 (= 2.1-1), libc6 (>= 2.2.5)
 $ dpkg -f libsharedlib-dev_2.1-1_amd64.deb pre-depends depends recommends co...
Depends: libsharedlib1 (= 2.1-1)
 $ dpkg -f libsharedlib1-dbgsym_2.1-1_amd64.deb pre-depends depends recommend...
Depends: libsharedlib1 (= 2.1-1)
 $ dpkg -f libsharedlib1_2.1-1_amd64.deb pre-depends depends recommends confl...
Depends: libc6 (>= 2.2.5)
```

8.13 国际化

Here is an example of updating the simple upstream C source **debhello-2.0.tar.gz** presented in 第 8.11 节 for internationalization (i18n) and creating the updated upstream C source **debhello-2.0.tar.gz**.

In the real situation, the package should already be internationalized. So this example is educational for you to understand how this internationalization is implemented.

提示

The routine maintainer activity for the i18n is simply to add translation po files reported to you via the Bug Tracking System (BTS) to the **po/** directory and to update the language list in the **po/LINGUAS** file.

Let’s get the source and make the Debian package.

Download debhello-2.0.tar.gz (i18n)

```
 $ wget http://www.example.org/download/debhello-2.0.tar.gz
 ...
 $ tar -xzmf debhello-2.0.tar.gz
 $ tree
.
├── debhello-2.0
│   ├── Makefile.am
│   ├── configure.ac
│   ├── data
│   │   ├── hello.desktop
│   │   └── hello.png
│   ├── lib
│   │   ├── Makefile.am
│   │   ├── sharedlib.c
│   │   └── sharedlib.h
│   ├── man
│   │   ├── Makefile.am
│   │   └── hello.1
│   └── src
│       ├── Makefile.am
│       └── hello.c
└── debhello-2.0.tar.gz

5 directories, 12 files
```

Internationalize this source tree with the **gettextize** command and remove files auto-generated by Autotools.

run gettextize (i18n):

```
 $ cd debhello-2.0
 $ gettextize
Creating po/ subdirectory
Creating build-aux/ subdirectory
Copying file ABOUT-NLS
Copying file build-aux/config.rpath
Not copying intl/ directory.
Copying file po/Makefile.in.in
Copying file po/Makevars.template
Copying file po/Rules-quot
Copying file po/boldquot.sed
Copying file po/en@boldquot.header
Copying file po/en@quot.header
Copying file po/insert-header.sin
Copying file po/quot.sed
Copying file po/remove-potcdate.sin
Creating initial po/POTFILES.in
Creating po/ChangeLog
```

```
Creating directory m4
Copying file m4/gettext.m4
Copying file m4/iconv.m4
Copying file m4/lib-ld.m4
Copying file m4/lib-link.m4
Copying file m4/lib-prefix.m4
Copying file m4/nls.m4
Copying file m4/po.m4
Copying file m4/progtest.m4
Creating m4/ChangeLog
Updating Makefile.am (backup is in Makefile.am~)
Updating configure.ac (backup is in configure.ac~)
Creating ChangeLog

Please use AM_GNU_GETTEXT([external]) in order to cause autoconfiguration
to look for an external libintl.

Please create po/Makevars from the template in po/Makevars.template.
You can then remove po/Makevars.template.

Please fill po/POTFILES.in as described in the documentation.

Please run 'aclocal' to regenerate the aclocal.m4 file.
You need aclocal from GNU automake 1.9 (or newer) to do this.
Then run 'autoconf' to regenerate the configure file.

You will also need config.guess and config.sub, which you can get from the CV...
of the 'config' project at http://savannah.gnu.org/. The commands to fetch th...
are
$ wget 'http://savannah.gnu.org/cgi-bin/viewcvs/*checkout*/config/config/conf...
$ wget 'http://savannah.gnu.org/cgi-bin/viewcvs/*checkout*/config/config/conf...

You might also want to copy the convenience header file gettext.h
from the /usr/share/gettext directory into your package.
It is a wrapper around <libintl.h> that implements the configure --disable-nl...
option.

Press Return to acknowledge the previous 6 paragraphs.
 $ rm -rf m4 build-aux *~
```

Let’s check generated files under the **po/** directory.
files in po (i18n):

```
 $ ls -l po
/build/debmake-doc-1.11/debhello-2.0-pkg2/step151.cmd: line 2: SOURCE_DATE_EP...
total 60
-rw-r--r-- 1 pbuilder pbuilder   494 Sep 25 03:42 ChangeLog
-rw-r--r-- 1 pbuilder pbuilder 17577 Sep 25 03:42 Makefile.in.in
-rw-r--r-- 1 pbuilder pbuilder  3376 Sep 25 03:42 Makevars.template
-rw-r--r-- 1 pbuilder pbuilder    59 Sep 25 03:42 POTFILES.in
-rw-r--r-- 1 pbuilder pbuilder  2203 Sep 25 03:42 Rules-quot
-rw-r--r-- 1 pbuilder pbuilder   217 Sep 25 03:42 boldquot.sed
-rw-r--r-- 1 pbuilder pbuilder  1337 Sep 25 03:42 en@boldquot.header
-rw-r--r-- 1 pbuilder pbuilder  1203 Sep 25 03:42 en@quot.header
-rw-r--r-- 1 pbuilder pbuilder   672 Sep 25 03:42 insert-header.sin
-rw-r--r-- 1 pbuilder pbuilder   153 Sep 25 03:42 quot.sed
-rw-r--r-- 1 pbuilder pbuilder   432 Sep 25 03:42 remove-potcdate.sin
```

Let’s update the **configure.ac** by adding “**AM_GNU_GETTEXT([external])**”, etc..
configure.ac (i18n):

```
 $ vim configure.ac
 ... hack, hack, hack, ...
 $ cat configure.ac
#                                               -*- Autoconf -*-
```

```
# Process this file with autoconf to produce a configure script.
AC_PREREQ([2.69])
AC_INIT([debhello],[2.2],[foo@example.org])
AC_CONFIG_SRCDIR([src/hello.c])
AC_CONFIG_HEADERS([config.h])
echo "Standard customization chores"
AC_CONFIG_AUX_DIR([build-aux])

AM_INIT_AUTOMAKE([foreign])

# Set default to --enable-shared --disable-static
LT_INIT([shared disable-static])

# find the libltdl sources in the libltdl sub-directory
LT_CONFIG_LTDL_DIR([libltdl])

# choose one
LTDL_INIT([recursive])
#LTDL_INIT([subproject])
#LTDL_INIT([nonrecursive])

# Add #define PACKAGE_AUTHOR ... in config.h with a comment
AC_DEFINE(PACKAGE_AUTHOR, ["Osamu Aoki"], [Define PACKAGE_AUTHOR])
# Checks for programs.
AC_PROG_CC

# desktop file support required
AM_GNU_GETTEXT_VERSION([0.19.3])
AM_GNU_GETTEXT([external])

# only for the recursive case
AC_CONFIG_FILES([Makefile
                 po/Makefile.in
                 lib/Makefile
                 man/Makefile
                 src/Makefile])
AC_OUTPUT
```

Let’s create the **po/Makevars** file from the **po/Makevars.template** file.
po/Makevars (i18n):

```
 ... hack, hack, hack, ...
 $ diff -u po/Makevars.template po/Makevars
--- po/Makevars.template        2018-08-09 13:57:35.829953013 +0000
+++ po/Makevars 2018-08-09 13:57:38.305953025 +0000
@@ -18,14 +18,14 @@
 # or entity, or to disclaim their copyright.  The empty string stands for
 # the public domain; in this case the translators are expected to disclaim
 # their copyright.
-COPYRIGHT_HOLDER = Free Software Foundation, Inc.
+COPYRIGHT_HOLDER = Osamu Aoki <osamu@debian.org>

 # This tells whether or not to prepend "GNU " prefix to the package
 # name that gets inserted into the header of the $(DOMAIN).pot file.
 # Possible values are "yes", "no", or empty.  If it is empty, try to
 # detect it automatically by scanning the files in $(top_srcdir) for
 # "GNU packagename" string.
-PACKAGE_GNU =
+PACKAGE_GNU = no

 # This is the email address or URL to which the translators shall report
 # bugs in the untranslated strings:
 $ rm po/Makevars.template
```

Let’s update C sources for the i18n version by wrapping strings with **_(**⋯**)**.

src/hello.c (i18n):

```
 ... hack, hack, hack, ...
 $ cat src/hello.c
#include "config.h"
#include <stdio.h>
#include <sharedlib.h>
#define _(string) gettext (string)
int
main()
{
        printf(_("Hello, I am " PACKAGE_AUTHOR "!\n"));
        sharedlib();
        return 0;
}
```

lib/sharedlib.c (i18n):

```
 ... hack, hack, hack, ...
 $ cat lib/sharedlib.c
#include <stdio.h>
#define _(string) gettext (string)
int
sharedlib()
{
        printf(_("This is a shared library!\n"));
        return 0;
}
```

The new **gettext** (v=0.19) can handle the i18n version of the desktop file directly.
data/hello.desktop.in (i18n):

```
 $ fgrep -v '[ja]=' data/hello.desktop > data/hello.desktop.in
 $ rm data/hello.desktop
 $ cat data/hello.desktop.in
[Desktop Entry]
Name=Hello
Comment=Greetings
Type=Application
Keywords=hello
Exec=hello
Terminal=true
Icon=hello.png
Categories=Utility;
```

Let’s list the input files to extract translatable strings in **po/POTFILES.in**.
po/POTFILES.in (i18n):

```
 ... hack, hack, hack, ...
 $ cat po/POTFILES.in
src/hello.c
lib/sharedlib.c
data/hello.desktop.in
```

Here is the updated root **Makefile.am** with **po** added to the **SUBDIRS** environment variable.
Makefile.am (i18n):

```
 $ cat Makefile.am
# recursively process `Makefile.am` in SUBDIRS
SUBDIRS = po lib src man

ACLOCAL_AMFLAGS = -I m4

EXTRA_DIST = build-aux/config.rpath m4/ChangeLog
```

Let’s make a translation template file, **debhello.pot**.
po/debhello.pot (i18n):

```
 $ xgettext -f po/POTFILES.in -d debhello -o po/debhello.pot -k_
 $ cat po/debhello.pot
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2018-08-09 13:57+0000\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: src/hello.c:8
#, c-format
msgid "Hello, I am "
msgstr ""

#: lib/sharedlib.c:6
#, c-format
msgid "This is a shared library!\n"
msgstr ""

#: data/hello.desktop.in:3
msgid "Hello"
msgstr ""

#: data/hello.desktop.in:4
msgid "Greetings"
msgstr ""

#: data/hello.desktop.in:6
msgid "hello"
msgstr ""

#: data/hello.desktop.in:9
msgid "hello.png"
msgstr ""
```

Let’s add a translation for French.

po/LINGUAS and po/fr.po (i18n):

```
 $ echo 'fr' > po/LINGUAS
 $ cp po/debhello.pot po/fr.po
 $ vim po/fr.po
 ... hack, hack, hack, ...
 $ cat po/fr.po
# SOME DESCRIPTIVE TITLE.
# This file is put in the public domain.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
msgid ""
msgstr ""
"Project-Id-Version: debhello 2.2\n"
"Report-Msgid-Bugs-To: foo@example.org\n"
"POT-Creation-Date: 2015-03-01 20:22+0900\n"
```

```
"PO-Revision-Date: 2015-02-21 23:18+0900\n"
"Last-Translator: Osamu Aoki <osamu@debian.org>\n"
"Language-Team: French <LL@li.org>\n"
"Language: ja\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: src/hello.c:34
#, c-format
msgid "Hello, my name is %s!\n"
msgstr "Bonjour, je m'appelle %s!\n"

#: lib/sharedlib.c:29
#, c-format
msgid "This is a shared library!\n"
msgstr "Ceci est une bibliothèque partagée!\n"

#: data/hello.desktop.in:3
msgid "Hello"
msgstr ""

#: data/hello.desktop.in:4
msgid "Greetings"
msgstr "Salutations"

#: data/hello.desktop.in:6
msgid "hello"
msgstr ""

#: data/hello.desktop.in:9
msgid "hello.png"
msgstr ""
```

The packaging activities are practically the same as the one in 第 8.11 节.
You can find more i18n examples in 第 8.14 节 for

- the POSIX shell script with Makefile (v=3.0),
- the Python3 script with distutils (v=3.1),
- the C source with Makefile.in + configure (v=3.2),
- the C source with Autotools (v=3.3), and
- the C source with CMake (v=3.4).

8.14 Details

Actual details of the examples presented and their variants can be obtained by the following.

How to get details

```
$ apt-get source debmake-doc
$ sudo apt-get install devscripts build-essentials
$ cd debmake-doc*
$ sudo apt-get build-dep ./
$ make
```

Each directory with the **-pkg[0-9]** suffix contains the Debian packaging example.

- emulated console command line activity log: the **.log** file
- emulated console command line activity log (short): the **.slog** file
- snapshot source tree image after the **debmake** command: the **debmake** directory

- snapshot source tree image after proper packaging: the **packge** directory
- snapshot source tree image after the **debuild** command: the **test** directory

Appendix A

debmake(1) manpage

A.1 NAME

debmake - program to make a Debian source package

A.2 SYNOPSIS

debmake [**-h**] [**-c** | **-k**] [**-n** | **-a** *package-version***.orig.tar.gz** | **-d** | **-t**] [**-p** *package*] [**-u** *version*] [**-r** *revision*] [**-z** *extension*] [**-b** "*binarypackage, ···*]" [**-e** *foo@example.org*] [**-f** "*firstname lastname*"] [**-i** "*buildtool*" | **-j**] [**-l** *license_file*] [**-m**] [**-o** *file*] [**-q**] [**-s**] [**-v**] [**-w** "*addon, ···*"] [**-x** [01234]] [**-y**] [**-L**] [**-P**] [**-T**]

A.3 DESCRIPTION

debmake helps to build a Debian package from the upstream source. Normally, this is done as follows:

- 下载上游源码压缩包（tarball）并命名为 *package-version***.tar.gz** 文件。
- It is untarred to create many files under the *package-version/* directory.
- debmake is invoked in the *package-version/* directory, possibly without any arguments.
- Files in the *package-version*/**debian/** directory are manually adjusted.
- **dpkg-buildpackage** (usually from its wrapper **debuild** or **pdebuild**) is invoked in the *package-version/* directory to make Debian packages.

请确保将 **-b**、**-f**、**-l** 和 **-w** 选项的参数使用引号合适地保护起来，以避免 shell 环境的干扰。

A.3.1 可选参数:

-h, --help 显示本帮助信息并退出。

-c, --copyright scan source for copyright+license text and exit.

- **-c**: simple output style
- **-cc**: normal output style (similar to the **debian/copyright** file)
- **-ccc**: debug output style

-k, --kludge compare the **debian/copyright** file with the source and exit.

The **debian/copyright** file must be organized to list the generic file patterns before the specific exceptions.

- **-k**: basic output style
- **-kk**: verbose output style

-n, --native make a native Debian source package without **.orig.tar.gz**. This makes a "**3.0 (native)**" format package.

If you are thinking of packaging a Debian-specific source tree with **debian/*** in it into a native Debian package, please think otherwise. You can use the "**debmake -d -i debuild**" or "**debmake -t -i debuild**" commands to make a "**3.0 (quilt)**" format non-native Debian package. The only difference is that the **debian/changelog** file must use the non-native version scheme: *version-revision*. The non-native package is more friendly to downstream distributions.

-a *package-version*.tar.gz, --archive *package-version*.tar.gz use the upstream source tarball directly. (**-p**, **-u**, **-z**: overridden)

The upstream tarball may be specified as *package_version*.**orig.tar.gz** and **tar.gz**. For other cases, it may be **tar.bz2**, or **tar.xz**.

If the specified upstream tarball name contains uppercase letters, the Debian package name is generated by converting them to lowercase letters.

If the specified argument is the URL (http://, https://, or ftp://) to the upstream tarball, the upstream tarball is downloaded from the URL using **wget** or **curl**.

-d, --dist run the "make dist" command equivalents first to generate the upstream tarball and use it.

The "**debmake -d**" command is designed to run in the *package/* directory hosting the upstream VCS with the build system supporting the "**make dist**" command equivalents. (automake/autoconf, Python distutils, ···)

-t, --tar run the "**tar**" command to generate the upstream tarball and use it.

The "**debmake -t**" command is designed to run in the *package/* directory hosting the upstream VCS. Unless you provide the upstream version with the **-u** option or with the **debian/changelog** file, a snapshot upstream version is generated in the **0~%y%m%d%H%M** format, e.g., *0~1403012359*, from the UTC date and time. The generated tarball excludes the **debian/** directory found in the upstream VCS. (It also excludes typical VCS directories: **.git/ .hg/ .svn/ .CVS/**.)

-p *package*, --package *package* 设置 Debian 软件包名称。

-u *version*, --upstreamversion *version* 设置上游软件包版本。

-r *revision*, --revision *revision* 设置 Debian 软件包修订号。

-z *extension*, --targz *extension* set the tarball type, *extension*=(**tar.gz**|**tar.bz2**|**tar.xz**). (alias: **z**, **b**, **x**)

-b " 二进制软件包名 ***[:type],***···**", --binaryspec "** 二进制软件包名 ***[:type],***···**"** set the binary package specs by a comma separated list of *binarypackage:type* pairs, e.g., in the full form "**foo:bin,foo-doc:doc,libfoo1:lib,libfoo-dev:dev**" or in the short form, "**-doc,libfoo1,libfoo-dev**".

这里，二进制软件包是二进制软件包名称，可选的类型应当从下面的类型值中进行选取：

- **bin**：C/C++ 预编译 ELF 二进制代码软件包（any，foreign）（默认，别名：**""**，即，空字符串）
- **data**：数据（字体、图像、·····）软件包（all，foreign）（别名：**da**）
- **dev**：库开发软件包（any，same）（别名：**de**）
- **doc**：文档软件包（all，foreign）（别名：**do**）
- **lib**：库软件包（any，same）（别名：**l**）
- **perl**：Perl 脚本软件包（all，foreign）（别名：**pl**）
- **python**：Python 脚本软件包（all，foreign）（别名：**py**）
- **python3**：Python3 脚本软件包（all，foreign）（别名：**py3**）
- **ruby**：Ruby 脚本软件包（all，foreign）（别名：**rb**）
- **script**：Shell 脚本软件包（all，foreign）（别名：**sh**）

括号内成对的值，例如（any，foreign），是软件包的架构和多架构（**Multi-Arch**）特性的值，它们将设置在 **debian/control** 文件中。

大多数情况下，**debmake** 命令可以有效地从二进制软件包的名称猜测出正确的类型。如果类型的值并不明显，程序将回退到将类型设置为 **bin**。例如，**libfoo** 设置类型为 **lib**，而 **font-bar** 会令程序设置类型为 **data**，·····

如果源码树的内容和类型的设置不一致，**debmake** 命令会发出警告。

-e *foo@example.org*, --email *foo@example.org* 设置电子邮件地址。

The default is taken from the value of the environment variable **$DEBEMAIL**.

-f *"firstname lastname"*, --fullname *"firstname lastname"* set the fullname.

The default is taken from the value of the environment variable **$DEBFULLNAME**.

-i *"buildtool"*, --invoke *"buildtool"* invoke "*buildtool*" at the end of execution. *buildtool* may be "**dpkg-buildpackage**", "**debuild**", "**pdebuild**", "**pdebuild --pbuilder cowbuilder**", etc.

The default is not to execute any program.

Setting this option automatically sets the **--local** option.

-j, --judge run **dpkg-depcheck** to judge build dependencies and identify file paths. Log files are in the parent directory.

- *package*.**build-dep.log**: Log file for **dpkg-depcheck**.
- *package*.**install.log**: Log file recording files in the **debian/tmp** directory.

-l *"license_file,⋯"*, --license *"license_file,⋯"* add formatted license text to the end of the **debian/copyright** file holding license scan results.

The default is to add **COPYING** and **LICENSE**, and *license_file* needs to list only the additional file names all separated by "**,**".

-m, --monoarch force packages to be non-multiarch.

-o *file*, --option *file* read optional parameters from *file*. (This is not for everyday use.)

The content of *file* is sourced as the Python3 code at the end of **para.py**. For example, the package description can be specified by the following file.

```
para['desc'] = 'program short description'
para['desc_long'] = '''\
 program long description which you wish to include.
 .
 Empty line is space + .
 You keep going on ...
'''
```

-q, --quitearly quit early before creating files in the **debian/** directory.

-s, --spec use upstream spec (setup.py for Python, etc.) for the package description.

-v, --version show version information.

-w *"addon,⋯"*, --with *"addon,⋯"* add extra arguments to the **--with** option of the **dh**(1) command as *addon* in **debian/rules**.

The *addon* values are listed all separated by "**,**", e.g., "**-w "python2,autoreconf"**".

For Autotools based packages, setting **autoreconf** as *addon* forces running "**autoreconf -i -v -f**" for every package building. Otherwise, **autotools-dev** as *addon* is used as the default.

For Autotools based packages, if they install Python programs, **python2** as *addon* is needed for packages with "**compat < 9**" since this is non-obvious. But for **setup.py** based packages, **python2** as *addon* is not needed since this is obvious and it is automatically set for the **dh**(1) command by the **debmake** command when it is required.

-x *n*, --extra *n* generate configuration files as templates. (Please note **debian/changelog**, **debian/control**, **debian/copyright**, and **debian/rules** are bare minimum configuration files to build a Debian binary package.)

The number *n* determines which configuration templates are generated.

- **-x0**: bare minimum configuration files. (default option if any of bare minimum configuration files already exist)

- **-x1**: all **-x0** files + desirable configuration files for the single binary package. (default option for the single binary package if none of bare minimum configuration files exist)
- **-x2**: all **-x1** files + desirable configuration files for the multi binary package. (default option for the multi binary package if none of bare minimum configuration files exist)
- **-x3**: all **-x2** files + unusual configuration template files. Unsual configuration template files are generated with the extra **.ex** suffix to ease their removal. To use these as configuration files, rename their file names to ones without the **.ex** suffix.
- **-x4**: all **-x3** files + copyright file examples.

-y, --yes "force yes" for all prompts. (without option: "ask [Y/n]"; doubled option: "force no")

-L, --local generate configuration files for the local package to fool **lintian**(1) checks.

-P, --pedantic pedantically check auto-generated files.

-T, --tutorial output tutorial comment lines in template files.

A.4 示例

For a well behaving source, you can build a good-for-local-use installable single Debian binary package easily with one command. Test install of such a package generated in this way offers a good alternative to the traditional "**make install**" command installing into the **/usr/local** directory since the Debian package can be removed cleanly by the "**dpkg -P** ⋯" command. Here are some examples of how to build such test packages. (These should work in most cases. If the **-d** option does not work, try the **-t** option instead.)

For a typical C program source tree packaged with autoconf/automake:

- **debmake -d -i debuild**

For a typical Python module source tree:

- **debmake -s -d -b":python" -i debuild**

For a typical Python module in the *package-version***.tar.gz** archive:

- **debmake -s -a *package-version*.tar.gz -b":python" -i debuild**

For a typical Perl module in the *Package-version***.tar.gz** archive:

- **debmake -a *Package-version*.tar.gz -b":perl" -i debuild**

A.5 HELPER PACKAGES

Packaging may require installation of some additional specialty helper packages.

- Python3 程序可能需要 **dh-python** 软件包。
- Autotools (Autoconf + Automake) 建构系统可能需要 **autotools-dev** 或 **dh-autoreconf** 软件包。
- Ruby 程序可能需要 **gem2deb** 软件包。
- Java 程序可能需要 **javahelper** 软件包。
- Gnome programs may require the **gobject-introspection** package.
- etc.

A.6 CAVEAT

debmake 的目的是为软件包维护者提供开始工作的模板文件。注释行以 **#** 开始，其中包含一些教程性文字。您在将软件包上传至 Debian 仓库之前必须删除或者修改这样的注释行。

许可证信息的提取和赋值过程应用了大量启发式操作，因此在某些情况下可能不会正常工作。强烈建议您搭配使用其它工具，例如来自 **devscripts** 软件包的 **licensecheck** 工具，以配合 **debmake** 的使用。

组成 Debian 软件包名称的字符选取存在一定的限制。最明显的限制应当是软件包名称中禁止出现大写字母。这里给出正则表达式形式的规则总结：

- 上游软件包名称（**-p**）：[-+.a-z0-9]{2,}
- 二进制软件包名称（**-b**）：[-+.a-z0-9]{2,}
- 上游版本号（**-u**）：[0-9][-+.:~a-z0-9A-Z]*
- Debian 修订版本（**-r**）：[0-9][+.~a-z0-9A-Z]*

请在“Debian 政策手册”的 第 5 章 - Control 文件及其字段 一节中查看其精确定义。

debmake 所假设的打包情景是相对简单的。因此，所有与解释器相关的程序都会默认为“**Architecture: all**”的情况。当然，这个假设并非总是成立。

A.7 除错

Please report bugs to the **debmake** package using the **reportbug** command.

The character set in the environment variable **$DEBUG** determines the logging output level.

- **i**: print information
- **p**: list all global parameters
- **d**: list parsed parameters for all binary packages
- **f**: input filename for the copyright scan
- **y**: year/name split of copyright line
- **s**: line scanner for format_state
- **b**: content_state scan loop: begin-loop
- **m**: content_state scan loop: after regex match
- **e**: content_state scan loop: end-loop
- **c**: print copyright section text
- **l**: print license section text
- **a**: print author/translator section text
- **k**: sort key for debian/copyright stanza
- **n**: scan result of debian/copyright (“**debmake -k**”)

Use this as:

```
$ DEBUG=pdfbmeclak debmake ...
```

查看源码中的 README.developer 文件以了解更多信息。

A.8 作者

A.9 许可证

Expat 许可证

A.10 参见

debmake-doc 软件包提供了“Debian 维护者指南”手册，以纯文本、HTML 和 PDF 三种格式存放在 **/usr/share/doc/debmake-doc/** 目录下。

另见 **dpkg-source**(1), **deb-control**(5), **debhelper**(7), **dh**(1), **dpkg-buildpackage**(1), **debuild**(1), **quilt**(1), **dpkg-depcheck**(1), **pdebuild**(1), **pbuilder**(8), **cowbuilder**(8), **gbp-buildpackage**(1), **gbp-pq**(1) 和 **git-pbuilder**(1) 的手册页。