Debian 维护者指导

青木修

September 17, 2017

Debian 维护者指导
by 青木修

本指导在撰写过程中参考了以下几篇文档：

- “Making a Debian Package (AKA the Debmake Manual)”, copyright © 1997 Jaldhar Vyas.
- “The New-Maintainer’s Debian Packaging Howto”, copyright © 1997 Will Lowe.
- “Debian New Maintainers’Guide”, copyright © 1998-2002 Josip Rodin, 2005-2014 Osamu Aoki, 2010 Craig Small, and 2010 Raphaël Hertzog.

本指导的最新版本可从 **debmake-doc** 软件包中获得。

Contents

Abstract

本教程文档面向普通 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。

它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 维护者指导”可看作“Debian 新维护者手册”的继承文档。

前言

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

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

如果你想要创建一个 Debian 软件包来满足你的需求，并将你的工作与社区分享，你便是本篇指导的目标读者，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：

- 在明确指定软件包依赖和补丁情况下干净地构建
- 跨多个架构构建软件包
- 拆分多个二进制软件包的最佳实践
- 平滑的软件库变迁
- 使用特定的编译选项进行安全增强
- 多架构（multiarch）支持
- ……

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。本篇指导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，你在参与 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。

The author felt limitations of updating the original “New Maintainers’ Guide” with the **dh-make** package and decided to create an alternative tool and its matching document to address modern requirements. The result is the **debmake** (version: 4.2.9) package and this updated “Guide for Debian Maintainers” in the **debmake-doc** (version: 1.9-1) package.

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

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

[2] 如果你对分享 Debian 软件包不感兴趣，你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。

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你没什么关系。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，目标是让这份指导保持简单而可维护。

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。

如果你在 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。

如果你曾经接触过 Debian 打包工作，你会注意到这和 **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。

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）

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**、**-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。

提示

非本土软件包是标准的 Debian 软件包。

提示

The detailed log of all the package build examples in this document can be obtained by following the instruction in 第 8.14 节.

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

这里给出你在投入 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，你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，如果你的软件包对其他人有用，你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，或者你可以申请成为 **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：

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

在这篇指导之后的部分中，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，参考行事可以让你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。

- 让你的软件包容易除错（debug）。
  - 保持你的软件包简单易懂。
  - 不要对软件包过度设计。
- 让你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

请在阅读本指导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）

如果本指导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。

这里有你当前阅读的指导的一些替代性教程文档：

- “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，你应当考虑使用 **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

在你决定在某些公共场合问出你的问题之前，请先做好自己应做的事，例如，查看这些文档与信息：

- 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。

Your desired information can be found effectively by using the well-formed search string such as “keyword **site:lists.debian.org**” to limit the search domain of the web search engine.

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。

如果你对打包仍然存在疑问，你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 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

如果你在做好功课后能在这些场合中合适地提出你的疑问的话，那些更有经验的 Debian 开发者会很愿意帮助你。

小心

Debian 的开发具有一个不断移动的目标。你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，也更有可能得到其他维护者的感谢、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。

你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。一个友好的上游则能协助解决程序的各类问题。

2.9 Novice contributor and maintainer

The novice contributor and maintainer may wonder what to learn to start your contribution to Debian. Here are my suggestions depending on your focus:

- Packaging
  - Basics of **POSIX shell** and **make**.
  - Some rudimentary knowledge of **Perl** and **Python**.
- Translation
  - Basics of how PO based translation system.
- Documentation
  - Basics of text markups (XML, ReST, Wiki, ⋯).

The novice contributor and maintainer may wonder where to start your contribution to Debian. Here are my suggestions depending on your skills:

- **POSIX shell**, **Perl**, and **Python** skills:
  - Send patches to the Debian Installer.
  - Send patches to the Debian packaging helper scripts such as **devscripts**, **pbuilder**, etc. mentioned in this document.
- **C** and **C++** skills:
  - Send patches to the packages with the **required** and **important** priorities.
- Non-English skills:
  - Send patches to the PO file of the Debian Installer.
  - Send patches to the PO file of the packages with the **required** and **important** priorities.
- Documentation skills:
  - Update contents on Debian Wiki.
  - Send patches to the existing Debian Documentation.

These activities should give you good exposure to the other Debian people to establish your credibility.
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，你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
```

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。

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

git 命令的用户级全局配置，如你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
```

如果你仍然只习惯 CVS 或者 Subversion 的命令风格，你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
```

你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。

你可以使用如下方式写入 **~/.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"
```

The **-i** and **-I** options in **DEBUILD_DPKG_BUILDPACKAGE_OPTS** for the **dpkg-source** command help rebuilding of Debian packages without extraneous contents (see 第 5.15 节).

当前情况下，使用 4096 位的 RSA 密钥是较好的做法。请见 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，你可能需要手动将 **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 --force-yes 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 --force-yes 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**”。你需要明白这样会带来相关的安全隐患。

3.7 git-buildpackage

你也可能需要在 **~/.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

你应当在本地设置 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

你可以使用 **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，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

The **debmake** command isn't the only way to make a Debian package. Many packages are packaged using only a text editor while imitating how other similar packages are packaged.

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）
        * 如果你按照第 3.5 节的说明设置了 **-us** 和 **-us** 选项的话，本步骤将会被跳过。你需要手动运行 **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/changelog
I: creating => debian/changelog
I: substituting => /usr/share/debmake/extra0/rules
I: creating => debian/rules
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/format
I: creating => debian/source/format
I: substituting => /usr/share/debmake/extra1source/local-options
I: creating => debian/source/local-options
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** 命令将为你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: extra
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=9)
Standards-Version: 3.9.8
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

如果你对 **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: http://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（export）**DH_VERBOSE** 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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=9)
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/

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 the 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

你可以使用 **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 -i
 ...
 fakeroot debian/rules clean
dh clean
```

```
 ...
 debian/rules build
dh build
   dh_update_autotools_config
   dh_auto_configure
   dh_auto_build
        make -j1
make[1]: Entering directory '/path/to/debhello-0.0'
# CFLAGS=-g -O2 -fdebug-prefix-map=/build/debmake-doc-1.9=.
# -fstack-protector-strong -Wformat -Werror=format-security
cc -Wdate-time -D_FORTIFY_SOURCE=2 -g -O2 -fdebug-prefix-map=/build/debmake-d...
make[1]: Leaving directory '/path/to/debhello-0.0'
 ...
 fakeroot debian/rules binary
dh binary
 ...
Now running lintian...
 ...
W: debhello: binary-without-manpage usr/bin/hello
Finished running lintian.
 ...
Finished running lintian.
```

这里验证了 **CFLAGS** 已经添加了 **-Wall** 和 **-pendatic** 参数,用到的便是 **DEB_CFLAGS_MAINT_APPEND** 变量。

根据 **lintian** 的报告，你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
```

你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。
- **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/LICENSE
debhello-0.0/src/
```

```
debhello-0.0/src/hello.c
 $ 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
```

debhello_0.0-1_amd64.deb 包含了将要安装至系统中的文件，如下所示。

debhello_0.0-1_amd64.deb 二进制软件包的内容：

```
 $ 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/68/
-rw-r--r-- root/root ...  ./usr/lib/debug/.build-id/68/93999c6f7f2dc7e11fb6ac...
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
```

自动产生的 **debhello_0.0-1_amd64.deb** 的依赖关系如下所示。

自动产生的 **debhello_0.0-1_amd64.deb** 的依赖关系：

```
 $ 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，也许这样对你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。下面给出了几种例子。
  - 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  2017-09-17 08:39:15.336528000 +0000
+++ debhello-0.0/Makefile       2017-09-17 08:39:15.404526432 +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**。
This **000-prefix-usr.patch** is edited to be DEP-3 conformant and moved to the right location as below.

```
 $ 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
 ... hack, hack, hack, ...
Replaced header of patch 000-prefix-usr.patch
 $ 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, ...
 $ cat Makefile
prefix = /usr
```

```
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
```

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 to /usr
Author: "Firstname Lastname" <email.address@example.org>

--- 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
│   │   └── Makefile
│   └── applied-patches
├── LICENSE
├── Makefile
├── debian
│   ├── README.Debian
│   ├── changelog
│   ├── compat
│   ├── control
│   ├── copyright
│   ├── patches
│   │   ├── 000-prefix-usr.patch
│   │   └── series
│   ├── rules
│   ├── source
│   │   ├── format
│   │   ├── local-options
│   │   └── local-patch-header
│   └── 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你理顺工作流程，但是请确保你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）。“Debian 政策手册”对此进行了定义。

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/*** 位置下的必要规范文件：

     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** 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。

提示

Although a Debian package can be made by writing a **debian/rules** script without using the **debhelper** package, it is unpractical to do so due to the complication of modern packaging concerns such as debug symbol handling, multiarch library, reproducible build, etc. The modern Debian packaging workflow can be organized into a simple modular workflow by using the **dh** command to invoke many utility scripts in the **debhelper** package and configuring their behavior with declarative configuration files in the **debian/** directory. Many “Policy”required features such as proper file permissions, insertion of installation hook scripts, etc. are automatically

5.1.1 The debhelper package

Although a Debian package can be made by writing a **debian/rules** script without using the **debhelper** package, it is unpractical to do so. There are too many modern “Policy” required features to be addressed, such as application of the proper file permissions, use of the proper architecture dependent library installation path, insertion of the installation hook scripts, generation of the debug symbol package, generation of package dependency information, generation of the package information files, application of the proper timestamp for reproducible build, etc.

The modern Debian packaging workflow can be organized into a simple modular workflow by:

- using the **dh** command to invoke many utility scripts automatically from the **debhelper** package and
- configuring their behavior with declarative configuration files in the **debian/** directory.

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指派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。下面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。如果你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读取版本信息。如果你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。

```
$ 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 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**

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

[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。

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。他们声称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传递信息，例如提供传递给某些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**。

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

在这种情况下，你应该添加一个 **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）。

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 5.19 节以了解与 multiarch 相关的定制方法。
- 请参见第 5.20 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

Some variable definitions useful for customizing **debian/rules** can be found in files under **/usr/share/dpkg/**. Notably:

pkg-info.mk **DEB_SOURCE**, **DEB_VERSION**, **DEB_VERSION_EPOCH_UPSTREAM**, **DEB_VERSION_UPSTREAM**, **DEB_VERSION_UPSTREAM**, and **DEB_DISTRIBUTION** variables. These are useful for the backport support etc..

vender.mk **DEB_VENDOR** and **DEB_PARENT_VENDOR** variables; and **dpkg_vendor_derives_from** macro. These are useful for the vendor support (Debian, Ubuntu, ···).

architecture.mk Set **DEB_HOST_*** and **DEB_BUILD_*** variables. Only variables used explicitly in **debian/rules** need to be defined using **dpkg-architecture**. So there is no need to include **architecture.mk** in **debian/rules**.

buildflags.mk Set **CFLAGS**, **CPPFLAGS**, **CXXFLAGS**, **OBJCFLAGS**, **OBJCXXFLAGS**, **GCJFLAGS**, **FFLAGS**, **FCFLAGS**, and **LDFLAGS** build flags.

If you wish to use some of these useful variables in **debian/rules**, copy relevant codes to **debian/rules** or write a simpler alternative in it. Please keep **debian/rules** simple.

For example, you can add an extra option to **CONFIGURE_FLAGS** for **linux-any** target architectures by adding the followings to **debian/rules**:

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

提示

It was useful to include **buildflags.mk** in the **debian/rules** to set the build flags such as **CPPFLAGS**, **CFLAGS**, **LDFLAGS**, etc. properly while honoring **DEB_CFLAGS_MAINT_APPEND**, **DEB_BUILD_MAINT_OPTIONS**, etc. for the **debhelper** “compat <= 8”. Now you should use the **debhelper** “compat >= 9”, should not include **buildflags.mk** without specific reasons, and should let the **dh** command set these build flags.

See 第 5.19 节, **dpkg-architecture**(1) and **dpkg-buildflags**(1).

5.4.5 Reproducible build

Here are some recommendations to attain the reproducible build result.

- Don’t embed the timestamp based on the system time.
- Use “**dh $@**” in the **debian/rules** to access the latest **debhelper** features.
- Export the build environment as “**LC_ALL=C.UTF-8**” (see 第 7.15 节).
- Set the timestamp used in the upstream source from the value of the debhelper provided environment variable **$SOURCE_DATE_EPOCH**.
- Read more at ReproducibleBuilds.
  - ReproducibleBuilds Howto。
  - ReproducibleBuilds TimestampsProposal。

The control file *source-name_source-version_arch***.buildinfo** generated by **dpkg-genbuildinfo**(1) records the build environment. See **deb-buildinfo**(5)

5.5 debian/control

The **debian/control** file consists of blocks of meta data separated by the blank line. Each block of meta data defines the following in this order:

- meta data for the Debian source package
- meta data for the Debian binary packages

See Chapter 5 - Control files and their fields of the “Debian Policy Manual” for the definition of each meta data.

5.5.1 Split of the Debian binary package

For well behaving build systems, the split of the Debian binary package into small ones can be realized as follows.

- Create binary package entries for all binary packages in the **debian/control** file.
- List all file paths (relative to **debian/tmp**) in the corresponding **debian/***binarypackage***.install** files.

Please check examples in this guide:

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

5.5.1.1 debmake -b

The **debmake** command with the **-b** option provides intuitive and flexible method to creates the initial template **debian/control** file defining the split of the Debian binary packages with following stanzas:

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

The **debmake** command also set an appropriate set of substvars used in each pertinent dependency stanza.
Let’s quote the pertinent part from the **debmake** manpage here.

-b ***"binarypackage[:type],***⋯***"*****, --binaryspec** ***"binarypackage[:type],***⋯***"*** set the binary package specs by the comma separated list of *binarypackage:type* pairs, e.g., in the full form "**foo:bin,foo-doc:doc,libfoo1:lib,libfoo1-dbg:dbg,libfoo-dev:dev**" or in the short form "**,-doc,libfoo1,libfoo1-dbg, libfoo-dev**".

Here, *binarypackage* is the binary package name; and the optional *type* is chosen from the following *type* values:

- **bin**: C/C++ compiled ELF binary code package (any, foreign) (default, alias: **""**, i.e., *null-string*)
- **data**: Data (fonts, graphics, ⋯) package (all, foreign) (alias: **da**)
- **dbg**: Debug symbol package (any, same) (alias: **db**) (deprecated for strech and after since the -dbgsym package is automatically generated)
- **dev**: Library development package (any, same) (alias: **de**)
- **doc**: Documentation package (all, foreign) (alias: **do**)
- **lib**: Library package (any, same) (alias: **l**)
- **perl**: Perl script package (all, foreign) (alias: **pl**)
- **python**: Python script package (all, foreign) (alias: **py**)
- **python3**: Python3 script package (all, foreign) (alias: **py3**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **script**: Shell script package (all, foreign) (alias: **sh**)

The pair values in the parentheses, such as (any, foreign), are the **Architecture** and **Multi-Arch** stanza values set in the **debian/control** file.

In many cases, the **debmake** command makes good guesses for *type* from *binarypackage*. If *type* is not obvious, *type* is set to **bin**. For example, **libfoo** sets *type* to **lib**, and **font-bar** sets *type* to **data**, ⋯

If the source tree contents do not match settings for *type*, the **debmake** command warns you.

5.5.1.2 Package split scenario and examples

Here are some typical multiarch package split scenarios for the following upstream source examples using the **debmake** command:

- a library source *libfoo-1.0*.**tar.gz**
- a tool source *bar-1.0*.**tar.gz** written in a compiled language
- a tool source *baz-1.0*.**tar.gz** written in an interpreted language

binarypackage	***type***	**Architecture:**	**Multi-Arch:**	软件包内容
lib*foo1*	**lib***	any	same	共享库，可共同安装
lib*foo1***-dbg**	**dbg***	any	same	共享库调试符号，可共同安装
lib*foo***-dev**	**dev***	any	same	共享库头文件之类，可共同安装
lib*foo***-tools**	**bin***	any	外来	运行时支持程序，不可共同安装
lib*foo***-doc**	**doc***	全部	外来	共享库文档
bar	**bin***	any	外来	编译好的程序文件，不可共同安装
*bar***-doc**	**doc***	全部	外来	程序的配套文档文件
baz	**script**	全部	外来	解释型程序文件

5.5.1.3 The library package name

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 the new SONAME major version which affects other packages.

The binary library package must be renamed from **lib***foo7* to **lib***foo8* to keep the **unstable** suite system working for all dependent packages after the upload of the package based on the **lib***foo-8.0*.**tar.gz**.

警告

If the binary library package isn't renamed, many dependent packages in the **unstable** suite become broken just after the library upload even if the binNMU is requested. The binNMU may not happen immediately after the upload due to several reasons.

The **-dev** package must follow one of the following naming rules:

- Use the **unversioned -dev** package name: **lib***foo***-dev**
  - This is the typical for the leaf library packages.
  - Only one version of the library source package is allowed in the archive.
    * The associated library package needs to be renamed from **lib***foo7* to **lib***foo8* to prevent the dependency breakage in the **unstable** archive during the library transition.
  - This approach should be used if the simple binNMU resolves the library dependency quickly for all affected packages. (ABI change by dropping the deprecated API while keeping the active API unchanged.)
  - This approach may still be a good idea if manual code updates, etc. can be coordinated and manageable within limited packages. (API change)
- Use the **versioned -dev** package names: **lib***foo7***-dev** and **lib***foo8***-dev**
  - This is typical for the many major library packages.
  - Two versions of the library source packages are allowed simultaneously in the archive.
    * Make all dependent packages depend on **lib***foo***-dev**.
    * Make both **lib***foo7***-dev** and **lib***foo8***-dev** provide **lib***foo***-dev**.
    * The source package needs to be renamed as **lib***foo7-7.0***.tar.gz** and **lib***foo8-8.0***.tar.gz** respectively from **lib***foo-?.0***.tar.gz**.
    * The package specific install file path including **lib***foo7* and **lib***foo8* respectively for header files etc. needs to be chosen to make them co-installable.
  - Do not use this heavy handed approach, if possible.
  - This approach should be used if the update of multiple dependent packages require manual code updates, etc. (API change) Otherwise, the affected packages becomes RC buggy with FTBFS.

提示

If data encoding scheme changes (e.g., latin1 to utf-8), the same care as the API change needs to be taken.

See 第 5.18 节.

5.5.2 Substvar

The **debian/control** file also defines the package dependency in which the variable substitutions mechanism (substvar) may be used to free package maintainers from chores of tracking most of the simple package dependency cases. See **deb-substvars**(5).

The **debmake** command supports following substvars.

- **${misc:Depends}** for all binary packages
- **${misc:Pre-Depends}** for all multiarch packages

- **${shlibs:Depends}** for all binary executable and library packages
- **${python:Depends}** for all Python packages
- **${python3:Depends}** for all Python3 packages
- **${perl:Depends}** for all Perl packages
- **${ruby:Depends}** for all Ruby packages

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

For the Python and other interpreters, required modules found simply looking for lines with "**import**", "**use**", "**require**", etc., are covered by the corresponding substvars.

For other programs which do not deploy their own substvars, the **misc** substvar covers their dependency.

For POSIX shell programs, there is no easy way to identify the dependency and no substvar covers their dependency.

For libraries and modules required via the dynamic loading mechanism including the GObject introspection mechanism, there is no easy way to identify the dependency and no substvar covers their dependency.

5.5.3 binNMU safe

A binNMU is a binary-only non-maintainer upload performed for library transitions etc. In a binNMU, only the "**Architecture: any**" packages are rebuilt with a suffixed version number (e.g. version 2.3.4-3 will become 2.3.4-3+b1). The "**Architecture: all**" packages are not built.

The dependency defined in the **debian/control** file among binary packages from a same source package should be safe for the binNMU. This needs attention if there are both "**Architecture: any**" and "**Architecture: all**" packages involved in it.

- "**Architecture: any**" package depends on "**Architecture: any**" *foo* package
  - **Depends:** *foo* **(= ${binary:Version})**
- "**Architecture: any**" package depends on "**Architecture: all**" *bar* package
  - **Depends:** *bar* **(= ${source:Version}**
- "**Architecture: all**" package depends on "**Architecture: any**" *baz* package
  - **Depends:** *baz* **(>= ${source:Version})**, *baz* **(<< ${source:Upstream-Version}.0~)**

5.6 debian/changelog

The **debian/changelog** file records the Debian package history and defines the upstream package version and the Debian revision in its first line.

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.

The **debchange** command (alias **dch**) is used to edit this.

提示

The date string used in the **debian/changelog** file can be manually generated by the "**LC_ALL=C date -R**"command.

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

The upstream changelog is installed in the **/usr/share/doc/***binarypackage* directory as **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.

After finishing your packaging and verifying its quality, please execute the “**dch -r**” command and save the finalized **debian/changelog** file with the distribution normally set to **unstable**. [4] If you are packaging for the backports, security updates, LTS, etc., please use the appropriate distribution names, instead.

5.7 debian/copyright

Debian takes the copyright and license matters very seriously. The “Debian Policy Manual” enforces to have summary of them in the **debian/copyright** file in the package.

You should format it as the machine-readable debian/copyright file (DEP-5).

小心

The **debian/copyright** file should be sorted to keep the generic file patterns at the top of the list. See 第 6.4 节.

The **debmake** command creates the initial DEP-5 compatible template file by scanning the entire source tree. It uses an internal license checker to classify each license text. [5]

Unless specifically requested to be pedantic with the **-P** option, the **debmake** command skips reporting for auto-generated files with permissive licenses to be practical.

注意

If you find issues with this license checker, please file a bug report to the **debmake** package with the problematic part of text containing the copyright and license.

注意

The **debmake** command focuses on bunching up same copyright and license claims in details to create template for **debian/copyright**. In order to do this within reasonable time, it only picks the first section which looks like copyright and license claims. So its license assignment may not be optimal. Please also use other tools such as **licensecheck**.

提示

You are highly encourage to check the license status with the **licensecheck**(1) command and, as needed, with your manual code review.

5.8 debian/patches/*

The **-p1** patches in the **debian/patches/** directory are applied in the sequence defined in the **debian/patches/series** file to the upstream source tree before the **build** process.

[4] If you are using the **vim** editor, make sure to save this with the “**:wq**” command.

[5] The **licensecheck** command from the **devscripts** package was referenced to make this internal checker. Now the **licensecheck** command is provided in an independent **licensecheck** package with a lot of improvements.

注意

The native Debian package (see 第 5.3 节) doesn’t use these files.

There are several methods to prepare a series of **-p1** patches.

- The **diff** command
  - See 第 4.8.1 节
  - Primitive but versatile method
    - Patches may come from other distros, mailing list postings, or cherry-picked patches from the upstream **git** repository with the “**git format-patches**” command
  - Missing the **.pc/** directory
  - Unmodified upstream source tree
  - Manually update the **debian/patches/series** file
- The **dquilt** command
  - See 第 3.4 节
  - Basic convenient method
  - Proper generation of the **.pc/** directory data
  - Modified upstream source tree
- The “**dpkg-source --commit**” command
  - See 第 4.8.3 节
  - Newer elegant method
  - Proper generation of the **.pc/** directory data
  - Modified upstream source tree
- The automatic patch generation by the **dpkg-buildpackage**
  - See 第 5.14 节
  - Add **single-debian-patch** in the **debian/source/local-options** file
  - Set the **debian/source/local-patch-header** file
  - Missing the **.pc/** directory
  - Modified upstream source tree in the Debian branch (**master**)
- The **gbp-pq** command
  - basic **git** work flow with the **git-buildpackage** package
  - Missing the **.pc/** directory
  - Modified upstream source tree in the through-away branch (**patch-queue/master**)
  - Unmodified upstream source tree in the Debian branch (**master**)
- The **gbp-dpm** command
  - more elaborate **git** work flow with the **git-dpm** package
  - Missing the **.pc/** directory
  - Modified upstream source tree in the patched branch (**patched/**_whatever_)
  - Unmodified upstream source tree in the Debian branch (**master/**_whatever_)

Wherever these patches come from, it is good idea to tag them with DEP-3 compatible header.

提示

The **dgit** package offeres an alternative git integration tool with the Debian package archive.

5.8.1 dpkg-source -x

The “**dpkg-source -x**” command unpacks the Debian source package.

It normally applies the patches in the **debian/patches/** directory to the source tree and records the patch state in the **.pc/** directory.

If you wish to keep the source tree unmodified (for example, for use in 第 5.13 节), please use the **--skip-patches** option.

5.8.2 dquilt and 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.

The patches should apply cleanly when using the **dpkg-source** command. Thus you can’t just copy the patches to the new packaging of the new upstream release if there are patch offset etc.

The **dquilt** command (see 第 3.4 节) is more foregiving. You can normalize the patches by the **dquilt** command.

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

There is one advantage of using the **dpkg-source** command over the **dquilt** command. While the **dquilt** command can not handle modified binary files automatically, the **dpkg-source** command detects modified binary files and lists them in the **debian/source/include-binaries** file to include them in the Debian tarball.

5.9 debian/upstream/signing-key.asc

Some packages are signed by the GPG key.

For example, GNU hello can be downloaded via HTTP from https://ftp.gnu.org/gnu/hello/ . There are sets of files:

- **hello-***version***.tar.gz** (upstream source)
- **hello-***version***.tar.gz.sig** (detached signature)

Let’s pick the latest version set.

```
$ 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
```

If you know the public GPG key of the upstream from the mailing list, use it as the **debian/upstream/signing-key.asc** file. Otherwise, use the hkp keyserver and check it via your web of trust.

```
$ 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

If your network environment blocks access to the HPK port **11371**, use "**hkp://keyserver.ubuntu.com:80**"instead.

After confirming the key ID **80EE4A00** is trust worthy one, download its public key into the **debian/upstream/signing-key.asc** file.

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

Then set the corresponding **debian/watch** file as follows.

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

Now the **uscan** command will check the authenticity of the package using the GPG signature.

5.10 debian/watch and DFSG

Debian takes software freedom seriously and follows the DFSG.

The non-DFSG components in the upstream source tarball can be easily removed when the **uscan** command is used to update the Debian package.

- List the files to be removed in the **Files-Excluded** stanza of the **debian/copyright** file.
- List the URL to download the upstream tarball in the **debian/watch** file.
- Run the **uscan** command to download the new upstream tarball.
  - Alternatively, use the “**gbp import-orig --uscan --pristine-tar**” command.
- The resulting tarball has the version number with an additional suffix **+dfsg**.

5.11 Other debian/*

Optional configuration files may be added under the **debian/** directory. Most of them are to control **dh_*** commands offered by the **debhelper** package but there are some for **dpkg-source**, **lintian** and **gbp** commands.

提示

Check **debhelper**(7) for the latest available set of the **dh_*** commands.

These **debian/***binarypackage***.*** files provide very powerful means to set the installation path of files. Even an upstream source without its build system can be packaged just by using these files. See 第 8.2 节 as an example.

Here is the alphabetical list of notable optional configuration files.

binarypackage*.bug-control** -x3 installed as **usr/share/bug/binarypackage***/control** in *binarypackage*. See 第 5.24 节.

binarypackage*.bug-presubj** ^-x3^ installed as **usr/share/bug/binarypackage***/presubj** in *binarypackage*. See 第 5.24 节.

binarypackage*.bug-script** ^-x3^ installed as **usr/share/bug/binarypackage* or **usr/share/bug/***binarypackage***/script** in *binarypackage*. See 第 5.24 节.

***binarypackage*.bash-completion** List `bash` completion scripts to be installed.

The `bash-completion` package is required for both build and user environment.

See **dh_bash-completion**(1).

clean ^-x2^ List files which should be removed but are not cleaned by the **dh_auto_clean** command.

See **dh_auto_clean**(1) and **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 to not 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 5.21 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*.

See **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*.

See **dh_installdocs**(1).

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

See **dh_installemacsen**(1).

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

See **dh_installemacsen**(1).

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

See **dh_installemacsen**(1).

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

See **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*.

See **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 **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 the 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 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 of the "Debian Policy Manual".

See also **debconf-devel**(7) and 3.9.1 Prompting in maintainer scripts of 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**.

See **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 contents 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 5.21 节.

tests/control This is the RFC822-style test meta data file defined in DEP-8. See **adt-run**(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**.

See **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 to the above list.

- For the single binary package, *binarypackage***.** part of the filename in the list may be removed.
- For the multi binary package, configuration file missing *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.
- Configuration template filenames with **-x1**, **-x2**, **-x3**, or **-x4** in the above mean that they are generated by the **debmake** command with that **-x** option (see 第 6.6 节).
- 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 the extra **.ex** suffix needs 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 Customization of the Debian packaging

Let’s recap the customization of the Debian packaging.

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

- 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 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, the modification 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). A 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 the future upgrades in this way.

注意

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

5.13 Recording in VCS (standard)

Typically, Git is used as VCS to record the Debian packaging activity with the following branches.

- **master** branch
  - Record the source tree used for the Debian packaging.
  - The upstream portion of the source tree is recorded unmodified.
  - The upstream modifications for the Debian packaging are recorded in the **debian/patches/** directory as the **-p1** patches.
- **upstream** branch
  - Record the upstream source tree untared from the released upstream tarball.

提示

It's a good idea to add the **.gitignore** file listing **.pc**.

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 Recording in VCS (alternative)

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** branch
  - Record the source tree used for the Debian packaging.

- The upstream portion of the source tree is recorded with modifications for the Debian packaging.

- **upstream** branch
  - Record the upstream source tree untared from the released upstream tarball.

Adding few extra files in **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**, ···) to 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

The **debian/source/local-options** and **debian/source/local-patch-header** files are meant to be recorded in the VCS. These aren't included in the Debian source package.

5.15 Building package without extraneous contents

There are a few cases which cause the inclusion of the 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 the 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 setting in 第 3.5 节.

提示

If the source tree is not managed by the 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 the 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 the 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.

5.16.1 Autotools

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

1. setup 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 usually performs the 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 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 ones and provides better supports for the porting to the newer architectures.

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

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

5.16.2 CMake

CMake has 4 steps.

1. setup 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 the step 1.
The package maintainer needs to take care steps 2 to 4.
If you wish to learn more on the CMake, please see:

- CMake
- CMake tutorial

5.16.3 Python distutils

Python distutils has 3 steps.

1. setup 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 usually performs the step 1 and builds the upstream tarball for distribution using the “**python setup.py sdist**” command.
The package maintainer needs to take care the 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 the Python3 and **distutils**, please see:

- Python3
- distutils

5.17 Debugging information

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 wiki DebugPackage
- Debian wiki AutomaticDebugPackages
- Debian debian-devel post: Status on automatic debug packages (2015-08-15)

5.17.1 New -dbgsym package (Strech 9.0 and after)

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

- No special care is needed in the **debian/rules** file, if there were no **-dbg** packages.
- Use the **--dbgsym-migration** option for the **dh_strip** command, if there were **-dbg** packages.

5.17.2 Old -dbg package (Jessie 8.0 or before)

The debugging information can be packaged separately as the debug package using the "**dh_strip --dbg-package=***package*" command in the **override_dh_strip:** target of the **debian/rules** file. The name of such debug package normally has the **-dbg** suffix.

The installation path of the debugging information is as follows to enable auto-loading of it by the **gdb** command.

- **/usr/lib/debug/.build-id/***12/3456*··· (compat>=9, for **buildID**=*123456*···)
- **/usr/lib/debug/***path/to/binary* (compat<<9, for */path/to/binary*)

注意

The creation of the **-dbg** package is optional.

注意

This is deprecated for Strech 9.0 and after.

5.18 库软件包

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

- The library binary package must be named as 第 5.5.1.3 节.
- Debian ships shared library such as **/usr/lib/<triplet>/lib***foo-0.1***.so.***1.0.0* (see 第 5.19 节).
- Debian encourages to use versioned symbols in the shared library (see 第 5.18.1 节).
- Debian doesn' t ship ***.la** libtool library archive files.
- Debian discourages to use and ship ***.a** static library files.

Before packaging the 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"

For the historic background study, see:

- Escaping the Dependency Hell [6]
  - This encourages to have versioned symbols in the shared library.
- Debian Library Packaging guide [7]
  - Please read the discussion thread following its announcement, too.

[6] This document was written before the introduction of the **symbols** file.

[7] The strong preference to use the SONAME versioned **-dev** package names over the single **-dev** package name in its Chapter 6. Development (-DEV) packages 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 Library symbols

The symbols support in the **dpkg** introduced to 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 to each symbol.

The 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 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.
    * Re-build the binary package.
  - If the **dpkg-gensymbols** command does not warn 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 wiki UsingSymbolsFiles
- Debian wiki Projects/ImprovedDpkgShlibdeps
- Debian kde team Working with symbols files
- 第 8.11 节
- 第 8.12 节

提示

For C++ libraries and other cases where the tracking of the 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 Library transition

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 第 5.5.1.3 节.

5.19 多体系结构

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

You should read the following references in detail.

- Ubuntu wiki (upstream)
  - 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** by the maintainer. The **<triplet>** value is stored in **$(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

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/**

5.19.1 The multiarch library path

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

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 part of **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_MULTIAI**
- 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 on 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, 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 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 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 the shared library files in the other path, GCC option **-l** must be set by the **pkg-config** command to make them loaded properly.

5.19.2 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, 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 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 to have more than 2 development libraries simultaneously installed on a system.

5.19.3 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/**

5.20 Compiler hardening

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

You should read the following references in detail.

- Debian wiki Hardening
- Debian wiki Hardening Walkthrough

The **debmake** command adds template comments to the **debian/rules** file as needed for **DEB_BUILD_MAINT_OPTIONS**, **DEB_CFLAGS_MAINT_APPEND**, and **DEB_LDFLAGS_MAINT_APPEND** (see 第 4 章 and **dpkg-buildflags**(1)).

5.21 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***.cofig**
  - 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 of the “Debian Policy Manual” .

[8] This path is compliant to 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.”

5.22 Continuous integration

DEP-8 defines the **debian/tests/control** file as the RFC822-style test metadata file for the 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 **adt-run** command provided by the **autopkgtest** package 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 the 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 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 report

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 something to the bug reporter or ask them about their situation, use these 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

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

提示

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 with the **-b** option support, 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 the 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 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 making of the non-native Debian packages 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 the **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 the 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 compare 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 the functioning multi binary package always requires extra manual works than that of the functioning single binary package. The test build of the source package is 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 the 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** or its equivalent command.

```
$ debuild
```

- All binary package entries specified in the **debian/***binarypackage***.install** file are generated as the *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 the 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 the 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 the verbose tutorial comment lines. The lines marked with **###** in the template files are part of the verbose tutorial comment lines.

Chapter 7

Tips

Here are some notable tips around the Debian packaging.

7.1 debdiff

You can compare file contents in two source Debian packages with the **debdiff** command.

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

You can also compare file lists in two sets of binary Debian packages with the **debdiff** command.

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

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

7.2 dget

You can download the set of files for the Debian source package with the **dget** command.

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

7.3 debc

You should install generated packages with the **debc** command to test it locally.

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

7.4 piuparts

You should install generated packages with the **piuparts** command to test it automatically.

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

注意

This is very slow process with the remote APT package repository access.

7.5 debsign

After completing the test of the package, you can sign it with the **debsign** command.

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

7.6 dput

After signing the package with the **debsign** command, you can upload the set of files for the Debian source and binary package with the **dput** command.

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

7.7 bts

After uploading the package, you will receive bug reports. It is an important duty of a package maintainer to manage these bugs properly as described in 5.8. Handling bugs of the “Debian Developer’s Reference”.

The **bts** command is a handy tool to manage bugs on Debian Bug Tracking System.

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

7.8 git-buildpackage

The **git-buildpackage** package offers many commands to automate packaging activities using the git repository.

- **gbp import-dsc**: import the previous Debian source package to the git repository.
- **gbp import-orig**: import the new upstream tar to the git repository.
  - The **--pristine-tar** option for the **git import-orig** command enables to store the upstream tarball in the same git repository.
  - The **--uscan** option as the last argument of the **gbp import-orig** command enables to download and commit the new upstream tarball into the git repository.
- **gbp dch**: generate the Debian changelog from the git commit messages.
- **gbp buildpackage**: build the Debian binary package from the git repository.
- **gbp pull**: update the **debian**, **upstream** and **pristine-tar** branches safely from the remote repository.
- **git-pbuilder**: build the Debian binary package from the git repository using the **pbuilder** package.
  - The **cowbuilder** package is used as its backend.
- The **gbp pq**, **git-dpm** or **quilt** (or alias **dquilt**) commands are used to manage quilt patches.
  - The use of the **dquilt** command is the simplest to learn and requires you to commit the resulting files manually with the **git** command to the **master** branch.
  - The use of the “**gbp pq**” command provides the equivalent functionality of the patch set management without using **dquilt** and eases including upstream git repository changes by the cherry-picking.
  - The use of the “**git dpm**” command provides the more enhanced functionality than that of the ‘**gbp pq**’’ command.

The package history management with the **git-buildpackage** package is becoming the standard practice for most Debian maintainers.

See:

- Building Debian Packages with git-buildpackage

- 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/
- The **systemd** packaging practice documentation on Building from source.

提示

Relax. You don’t need to use all the wrapper tools. Use only ones which match your needs.

7.8.1 gbp import-dscs --debsnap

For Debian source packages named <*source-package*> recorded in the snapshot.debian.org archive, an initial git repository with all of the Debian version history can be generated as follows.

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

7.9 Upstream git repository

For the Debian packaging with the **git-buildpackage** package, the **upstream** branch on the remote repository **origin** is normally used to track the content of the released upstream tarball.

The upstream git repository can also be tracked by naming its remote repository as **upstream** instead of the default **origin**. Then you can easily cherry-pick recent upstream changes into the Debian revision by cherry-picking with the **gitk** command and using the **gbp-pq** command.

提示

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

小心

The content of the released upstream tarball may not match exactly with the corresponding content of the upstream git repository. It may contain some auto-generated files or miss some files. (Autotools, distutils, ⋯)

7.10 chroot

The chroot for the clean package build environment can be created and managed using the tools described in 第 3 章.[1]

Here is a quick summary of available package build commands. There are many ways to do the same thing.

- **dpkg-buildpackage** = core of package building tool

[1] The **git-pbuilder** style organization is deployed here. See https://wiki.debian.org/git-pbuilder . Be careful since many HOWTOs use different organization.

- **debuild** = **dpkg-buildpackage** + **lintian** (build under the sanitized environment variables)
- **pbuilder** = core of the Debian chroot environment tool
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (build in the chroot)
- **cowbuilder** = speed up the **pbuilder** execution
- **git-pbuilder** = the easy-to-use command line syntax for **pdebuild** (used by **gbp buildpackge**)
- **gbp** = manage the Debian source under the git
- **gbp buildpackge** = **pbuilder** + **dpkg-buildpackage** + **gbp**

A clean **sid** distribution chroot environment can be used as follows.

- The chroot filesystem creation command for the **sid** distribution
  - **pbuilder create**
  - **git-pbuilder create**
- The master chroot filesystem path for the **sid** distribution chroot
  - **/var/cache/pbuilder/base.cow**
- The package build command for the **sid** distribution chroot
  - **pdebuild**
  - **git-pbuilder**
  - **gbp buildpackage**
- The command to update the **sid** chroot
  - **pbuilder --update**
  - **git-pbuilder update**
- The command to login to the **sid** chroot to modify it
  - **git-pbuilder login --save-after-login**

An arbitrary *dist* distribution environment can be used as follows.

- The chroot filesystem creation command for the *dist* distribution
  - **pbuilder create --distribution** *dist*
  - **DIST=***dist* **git-pbuilder create**
- The master chroot filesystem path for the *dist* distribution chroot
  - path: **/var/cache/pbuilder/base-***dist***.cow**
- The package build command for the *dist* distribution chroot
  - **pdebuild -- --basepath=/var/cache/pbuilder/base-***dist***.cow**
  - **DIST=***dist* **git-pbuilder**
  - **gbp buildpackage --git-dist=***dist*
- The command to update the *dist* chroot
  - **pbuilder update --basepath=/var/cache/pbuilder/base-***dist***.cow**
  - **DIST=***dist* **git-pbuilder update**
- The command to login to the **sid** 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

A custom environment with some pre-loaded packages needed for the new experimental packages, this “**git-pbuilder login --save-after-login**”command is quite handy.

提示

If your old chroot is missing packages such as **libeatmydata1**, **ccache**, and **lintian**, you may want to install these with the “**git-pbuilder login --save-after-login**”command.

提示

The chroot filesystem can be cloned simply by copying with the “**cp -a base-*dist*.cow base-*customdist*.cow**”command. The new chroot can be accessed as “**gbp buildpackage --git-dist=*customdist***” and “**DIST=*customdist* git-pbuilder** …”.

提示

When the **orig.tar.gz** file needs to be upload for the Debian revision other than **0** and **1** (e.g., for the security upload), add the **-sa** option to the end of **dpkg-buildpackage**, **debuild**, **pdebuild**, and **git-pbuilder** commands. For the “**gbp buildpackage**”command, temporarily modify the **builder** setting of **~/.gbp.conf**.

注意

The description in this section is too terse to be useful for most of the prospective maintainers. This is the intentional choice of the author. You are highly encouraged to search and read all the pertinent documents associated with the commands used.

7.11 新的 Debian 版本

Let’s assume that a bug report *#bug_number* was filed against your package, and it describes a problem that you can solve by editing **buggy** file in the upstream source. Here’s what you need to do to create a new Debian revision of the package with the **bugname.patch** file recording the fix.

New Debian revision with the dquilt command

```
$ 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.

New Debian revision with the gbp-pq command

```
$ 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>"
```

Please make sure to describe concisely the changes that fix reported bugs and close those bugs by adding "**Closes: #**<*bug_number*>" in the **debian/changelog** file.

提示

Use a **debian/changelog** entry with a version string such as **1.0.1-1~rc1** when you experiment. Then, unclutter such **changelog** entries into a single entry for the official package.

7.12 新上游版本

If a package **foo** is properly packaged in the modern "**3.0 (native)**" or "**3.0 (quilt)**" formats, packaging a new upstream release is essentially moving the old **debian/** directory to the new source. This can be done by running the "**tar -xvzf** */path/to/foo_oldversion*.**debian.tar.gz**" command in the new extracted source. [2] Of course, you need to do some obvious chores.

There are several tools to handle this situation. After updating to the new upstream release with these tools, please make sure to describe concisely the changes in the new upstream release that fix reported bugs and close those bugs by adding "**Closes: #***bug_number*" in the **debian/changelog** file.

7.12.1 uupdate + tarball

You can automatically update to the new upstream source with the **uupdate** command from the **devscripts** package. It requires to have the old Debian source package and the new upstream tarball.

```
$ 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
```

[2] If a **foo** package is packaged in the old **1.0** format, this can be done by running the "**zcat** */path/to/foo_oldversion*.**diff.gz|patch -p1**" command in the new extracted source, instead.

7.12.2 uscan

You can automatically update to the new upstream source with the **uscan** command from the **devscripts** package. It requires to have the old Debian source package and the **debian/watch** file in it.

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

7.12.3 gbp

You can automatically update to the new upstream source with the “**gbp import-orig --pristine-tar**” command from the **git-buildpackage** package. It requires to have the old Debian source in the git repository and the new upstream tarball.

```
$ 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

If upstream uses the git repository, please also use **--upstream-vcs-tag** option for the **gbp import-orig** command.

7.12.4 gbp + uscan

You can automatically update to the new upstream source with the “**gbp import-orig --pristine-tar --uscan**” command from the **git-buildpackage** package. It requires to have the old Debian source in the git repository and the **debian/watch** file in it.

```
$ 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

If upstream uses the git repository, please also use **--upstream-vcs-tag** option for the **gbp import-orig** command.

7.13 3.0 source format

Updating the package style is not a required activity for the update of a package. However, doing so lets you use the full capabilities of the modern **debhelper** system and the **3.0** source format.

- If you need to recreate deleted template files for any reason, you can run **debmake** again in the same Debian package source tree. Then edit them appropriately.
- If the package has not been updated to use the **dh** command for the **debian/rules** file, update it to use it (see 第 5.4.2 节). Update the **debian/control** file accordingly.
- If you have a **1.0** source package with the **foo.diff.gz** file, you can update it to the newer “**3.0 (quilt)**” source format by creating **debian/source/format** with “**3.0 (quilt)**”. The rest of the **debian/*** files can just be copied. Import the **big.diff** file generated by the “**filterdiff -z -x */debian/* foo.diff.gz > big.diff**” command to your quilt system, if needed. [3]
- If it was packaged using another patch system such as **dpatch**, **dbs**, or **cdbs** with **-p0**, **-p1**, or **-p2**, convert it to the quilt command using the **deb3** script in the **quilt** package.
- If it was packaged with the **dh** command with the “**--with quilt**” option or with the **dh_quilt_patch** and **dh_quilt_unpatch** commands, remove these and make it use the newer “**3.0 (quilt)**” source format.
- If you have a **1.0** source package without the **foo.diff.gz** file, you can update it to the newer “**3.0 (native)**” source format by creating **debian/source/format** with “**3.0 (native)**”. The rest of the **debian/*** files can just be copied.

You should check DEP - Debian Enhancement Proposals and adopt ACCEPTED proposals.
See ProjectsDebSrc3.0 to check the support status of the new Debian source formats by the Debian tool chains.

7.14 CDBS

The Common Debian Build System (**CDBS**) is a wrapper system over the **debhelper** package. The **CDBS** is based on the Makefile inclusion mechanism and configured by the **DEB_*** configuration variables set in the **debian/rules** file.

Before the introduction of the **dh** command to the **debhelper** package at the version 7, the **CDBS** was the only approach to create a simple and clean **debian/rules** file.

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

“The **CDBS** magically does the job for me with less typing”nor “I don’t understand the new **dh** syntax”can’t be the excuse to keep using the **CDBS** system.

For some complicated packages such as GNOME related ones, the **CDBS** is leveraged to automate their uniform packaging by the current maintainers with reasons. If this is the case, please do not bother converting from the **CDBS** to the **dh** syntax.

[3] You can split the **big.diff** file into many small incremental patch files using the **splitdiff** command.

注意

If you are working with a team of maintainers, please follow the established practice of the team.

When converting packages from the **CDBS** to the **dh** syntax, please use the following as your reference:

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

7.15 Build under UTF-8

The default locale of the build environment is **C**.

Some programs such as the **read** function of Python3 change their behavior depending on the locale.

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

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

7.16 UTF-8 转换

If upstream documents are encoded in old encoding schemes, converting them to UTF-8 is a good idea.

Use the **iconv** command in the **libc-bin** package to convert encodings of plain text files.

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

Use **w3m**(1) to convert from HTML files to UTF-8 plain text files. When you do this, make sure to execute it under UTF-8 locale.

```
 $ 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
```

Run these scripts in the **override_dh_*** target of the **debian/rules** file.

7.17 Upload orig.tar.gz

When you first upload the package to the archive, you need to include the original **orig.tar.gz** source, too.

If the Debian revision number of the package is either **1** or **0**, this is the default. Otherwise, you must provide the **dpkg-buildpackage** option **-sa** to the **dpkg-buildpackage** command.

- **dpkg-buildpackage -sa**
- **debuild -sa**
- **pdebuild --debbuildopts -sa**
- **git-pbuilder -sa**
- For **gbp buildpackage**, edit the **~/.gbp.conf** file.

提示

On the other hand, the **-sd** option will force the exclusion of the original **orig.tar.gz** source.

提示

Security uploads require to include the **orig.tar.gz**.

7.18 跳过的上传

If you created multiple entries in the **debian/changelog** while skipping uploads, you must create a proper ***_.changes** file which includes all changes since the last upload. This can be done by specifying the dpkg-buildpackage option **-v** with the last uploaded version, e.g., *1.2*.

- **dpkg-buildpackage -v***1.2*
- **debuild -v***1.2*
- **pdebuild --debbuildopts -v***1.2*
- **git-pbuilder -v***1.2*
- For **gbp buildpackage**, edit the **~/.gbp.conf** file.

7.19 高级打包

Hints for the following can be found in the **debhelper**(7) manpage:

- differences of the **debhelper** tool behavior under “**compat** <= 8”
- building several binary packages with several different build conditions
  - making multiple copies of the upstream source
  - invoking multiple “**dh_auto_configure -S** …” commands in the **override_dh_auto_configure** target
  - invoking multiple “**dh_auto_build -S** …” commands in the **override_dh_auto_build** target
  - invoking multiple “**dh_auto_install -S** …” commands in the **override_dh_auto_install** target
- building **udeb** packages with “**Package-Type: udeb**” in **debian/control** (see Package-Type)
- excluding some packages for the bootstrapping (see also BuildProfileSpec)
  - adding the **Build-Profiles** fields in binary package stanzas in **debian/control**
  - building packages with the **DEB_BUILD_PROFILES** environment variable set to the pertinent profile name

Hints for the following can be found in the **dpkg-source**(1) manpage:

- naming convention for the multiple upstream source tarballs
  - *packagename_version***.orig.tar.gz**
  - *packagename_version***.orig-***componentname***.tar.gz**
- recording the Debian changes to the upstream source package
  - **dpkg-source --commit**

7.20 Other distros

Although the upstream tarball has all the information to build the Debian package, it is not always easy to figure out which combination of options to use.

Also, the upstream may be more focused on the feature enhancements and may be less eager for backward compatibilities etc. which are important aspect of the Debian packaging practice.

The leveraging of information from other distributions is an option to address above issues.

If the other distribution in interest is a Debian derivative one, it is trivial to reuse it.

If the other distribution in interest is a RPM based distribution, see Repackage src.rpm.

Downloading and opening of the **src.rpm** file can be done with the **rget** command. (Place the **rget** script in your PATH.)

rget script

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

Many upstream tarballs contain the SPEC file named as *packagename*.**spec** or *packagename*.**spec.in** used by the RPM system. This can be used as the baseline for the Debian package, too.

7.21 Debug

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**”
  - Update the “**/etc/security/limits.conf**” file to include the following:

    ```
    * soft core unlimited
    ```

  - “**ulimit -c unlimited**” in **~/.bashrc**
  - “**ulimit -a**” to check
  - Press **Ctrl-\** or “**kill -ABRT** *PID*” to make a core dump file
- **gdb** - The GNU Debugger
  - “**info gdb**”
  - “Debugging with GDB” in **/usr/share/doc/gdb-doc/html/gdb/index.html**
- **strace** - Trace system calls and signals
  - Use **strace-graph** script found in **/usr/share/doc/strace/examples/** to make a nice tree view
  - “**man strace**”
- **ltrace** - Trace library calls
  - “**man ltrace**”
- “**sh -n** *script.sh*” - Syntax check of a Shell script
- “**sh -x** *script.sh*” - Trace a Shell script
- “**python -m py_compile** *script.py*” - Syntax check of a Python script
- “**python -mtrace --trace** *script.py*” - Trace a Python script
- “**perl -I ../libpath -c** *script.pl*” - Syntax check of a Perl script

- “**perl -d:Trace** *script.pl*” - Trace a Perl script
  - Install the **libterm-readline-gnu-perl** package or its equivalent to add the input line editing capability with history support.
- **lsof** - List open files by processes
  - “**man lsof**”

提示

The **script** command helps recording console outputs.

提示

The **screen** and **tmux** commands used with the **ssh** command offer the secure and robust remote connection terminal.

提示

Python and Shell like REPL (=READ + EVAL + PRINT + LOOP) environment for Perl is offered by the **reply** command from the **libreply-perl** (new) and the **re.pl** command from the **libdevel-repl-perl** (old).

提示

The **rlwrap** and **rlfe** commands add the input line editing capability with history support to any interactive commands. E.g. “**rlwrap dash -i**” .

Chapter 8

More Examples

There is an old Latin saying: “**fabricando fit faber**”(“practice makes perfect”).

It is highly recommended to practice and experiment with all the steps of Debian packaging with simple packages. This chapter provides you with the many upstream cases for your practice.

This should also serve as introductory examples for many programing topics.

- Programing in the POSIX shell, Python3, and C.
- Method to create the desktop GUI program launcher with the icon graphics.
- Conversion of command from CLI to GUI.
- Conversion of program to use **gettext** for internationalization and localization: POSIX shell, Python3, and C sources.
- Overview of many build systems: Makefile, Python distutils, Autotools, and CMake.

Please note Debian takes few things serious.

- Free software (a.k.a. Libre software)
- Stability and security of OS.
- Universal OS realized via:
  - free choice for upstream sources,
  - free choice of CPU architectures, and
  - free choice of UI languages.

The typical packaging example presented in 第 4 章 is the prerequisite for this chapter.

Some details are intentionally left vague in the following. Please try to read the pertinent documentation and practice yourself to find them out.

提示

The best source of the packaging example is the current Debian archive itself. Please use the “Debian Code Search”service to find pertinent examples.

8.1 Cherry-pick templates

Here is an example of creating a simple Debian package from a zero content source on an empty directory.

This is a good platform to get all the template files without making mess in the upstream source tree you are working on.

Let’s assume this empty directory to be **debhello-0.1**.

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

1 directory, 0 files
```

Let’s generate the maximum amount of template files by specifying the **-x4** option.
Let’s also use the “**-p** *debhello* **-t -u** *0.1* **-r** *1*” options to make missing upstream tarball.

```
 $ 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/changelog
 ...
I: substituting => /usr/share/debmake/extra4/GPL-2.0+
I: creating => debian/license-examples/GPL-2.0+
I: substituting => /usr/share/debmake/extra4/LGPL-3.0+
I: creating => debian/license-examples/LGPL-3.0+
I: substituting => /usr/share/debmake/extra4/LGPL-2.1+
I: creating => debian/license-examples/LGPL-2.1+
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
 ...
│       ├── source.lintian-overrides.ex
│       └── watch
├── debhello-0.1.tar.gz
└── debhello_0.1.orig.tar.gz -> debhello-0.1.tar.gz

5 directories, 52 files
```

Now you can copy any of these generated template files in the *debhello-0.1/***debian/** directory to your package as needed while renaming them as needed.

提示

The generated template files can be made more verbose ones by invoking the **debmake** command with the **-T** option (tutorial mode).

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, the **hello.desktop** supports 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, the **hello.png** is the icon graphics file.

Let' s package this with the **debmake** command. Here, the **-b':sh'** option is used to specify 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: extra
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=9)
Standards-Version: 3.9.8
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 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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=9)
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/

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 the 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

如果你对 **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。

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 -i
 ...
 fakeroot debian/rules clean
dh clean
 ...
 debian/rules build
dh build
   dh_update_autotools_config
   dh_auto_configure
   dh_auto_build
   dh_auto_test
 fakeroot debian/rules binary
dh binary
   dh_testroot
   dh_prep
        rm -f -- debian/debhello.substvars
 ...
 fakeroot debian/rules binary
dh binary
 ...
Finished running lintian.
```

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
```

你可以看见生成的全部文件。

- The **debhello_0.2.orig.tar.gz** is a symlink to the upstream tarball.
- The **debhello_0.2-1.debian.tar.xz** contains the maintainer generated contents.
- The **debhello_0.2-1.dsc** is the meta data file for the Debian source package.
- The **debhello_0.2-1_amd64.deb** is the Debian binary package.
- The **debhello_0.2-1_amd64.changes** is the meta data file for the Debian binary package.

The **debhello_0.2-1.debian.tar.xz** 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/data/
debhello-0.2/data/hello.png
debhello-0.2/data/hello.desktop
debhello-0.2/scripts/
debhello-0.2/scripts/hello
debhello-0.2/LICENSE
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** contains the files to be installed as follows.
The binary package contents of debhello_0.2-1_amd64.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
```

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 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 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 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 are no needs 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 one 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 the **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 to change the last line to “**python3 setup.py install --install-layout=deb**” to install files to the target system image location. This issue is automatically addressed when using the **dh** command for the 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 setup-tools 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 contain 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 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 "**--with python3**" option invokes **dh_python3** to calculate Python dependencies, adds maintainer scripts to byte compile files, etc. See **dh_python3**(1).

The use of "**--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: extra
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=9), dh-python, python3-all
Standards-Version: 3.9.8
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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=9), dh-python, python3-all
Standards-Version: 3.9.6
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 the 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 one 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 binary packages.

The generated dependency list of binary packages (v=1.1):

```
 $ dpkg -f debhello_1.1-1_all.deb pre-depends depends recommends conflicts br...
Depends: python3:any (>= 3.3.2-2~)
```

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 in 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 第 8.3 节.

Let’s package this with the **debmake** command. Here, the **-b’:sh’** option is used to specify 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 notable template files generated.

debian/control (template file, v=1.2):

```
 $ cat debhello-1.2/debian/control
Source: debhello
Section: unknown
Priority: extra
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: debhelper (>=9)
Standards-Version: 3.9.8
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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=9)
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/

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 the 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 manually added **zenity**.

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 the one in 第 8.3 节.

Here is the generated dependency list of binary packages.

The generated dependency list of binary packages (v=1.2):

```
 $ dpkg -f debhello_1.2-1_all.deb pre-depends depends recommends conflicts br...
Depends: zenity
```

8.6 setup.py (Python3, GUI)

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 in 第 8.4 节 with enhanced GUI, desktop icon, and manpage supports.

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 the generated binary package contain 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 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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=9), dh-python, python3-all
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/
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 the 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 manually added **python3-gi** and **gir1.2-gtk-3.0**.

Since this upstream source has manpage and other files with matching entries in the **setup.py** file, there are no needs to create them and add **debian/install** and **debian/manpages** files which was required in 第 8.4 节.

The rest of the packaging activities are practically the same as the one in 第 8.4 节.

Here is the generated dependency list of binary packages.

The generated dependency list of binary packages (v=1.3):

```
 $ dpkg -f debhello_1.3-1_all.deb pre-depends depends recommends conflicts br...
Depends: gir1.2-gtk-3.0, python3-gi, python3:any (>= 3.3.2-2~)
```

8.7 Makefile (single-binary)

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。

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 的打包需要对“**make install**”流程进行改变，从而将文件安装至系统镜像所在位置，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-1.4.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 第 4.5 节.

Let’s make this Debian package better as the maintainer which is practically the same as 第 4.6 节.

If the **DEB_BUILD_MAINT_OPTIONS** is not exported in the **debian/rules**, the lintian warns ”W: debhello: hardening-no-relro usr/bin/hello” for the linking of the **libm**.

The **debian/control** file making it exactly the same as one in 第 4.6 节, since the **libm** library is always available as a part of the **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 binary packages.

The generated dependency list of 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)

Here is an example of creating a simple Debian package from a simple C source using the **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 the **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 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)

Here is an example of creating a simple Debian package from a simple C source using the 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 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 the porting to the 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
```

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 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 (single-binary)

Here is an example of creating a simple Debian package from a simple C source using the 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 configure the file as specified in its **configure_file(**···**)** by replacing strings with **@**···**@** and changing **#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 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: extra
Maintainer: "Firstname Lastname" <email.address@example.org>
Build-Depends: cmake, debhelper (>=9)
Standards-Version: 3.9.8
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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper (>=9)
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/
```

```
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 the 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 (multi-binary)

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 using the Autotools = Autoconf and Automake (**Makefile.am** and **configure.ac**) as its build system. See 第 5.16.1 节.

Let' s package this in the same way as 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**
- **debhello-dbg**: type = **dbg**
- **libsharedlib1-dbg**: type = **dbg**

Here, the **-b',libsharedlib1,libsharedlib-dev,-dbg,libsharedlib1-dbg'** option is used to specify the generated binary packages.

```
 $ cd debhello-2.0
 $ debmake -b',libsharedlib1,libsharedlib-dev,-dbg,libsharedlib1-dbg'
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,-dbg,libsha...
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: binary package=debhello-dbg Type=dbg / Arch=any M-A=same
I: binary package=libsharedlib1-dbg Type=dbg / Arch=any M-A=same
...
```

The result is similar to 第 8.8 节 but with more template files.

Let’ s inspect 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

override_dh_strip:
	dh_strip -Xlibsharedlib1 --dbg-package=debhello-dbg
	dh_strip -Xdebhello --dbg-package=libsharedlib1-dbg
```

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

override_dh_strip:
	dh_strip -Xlibsharedlib1 --dbg-package=debhello-dbg
	dh_strip -Xdebhello --dbg-package=libsharedlib1-dbg
```

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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: debhelper (>=9), dh-autoreconf
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/

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 the 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 the 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 the 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.

Package: debhello-dbg
Section: debug
Architecture: any
Multi-Arch: same
Depends: debhello (= ${binary:Version}), ${misc:Depends}
Description: example debugging package for debhello
 This is an example package to demonstrate the 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 debugging symbols for debhello.

Package: libsharedlib1-dbg
Section: debug
Architecture: any
Multi-Arch: same
Depends: libsharedlib1 (= ${binary:Version}), ${misc:Depends}
Description: example debugging package for libsharedlib1
 This is an example package to demonstrate the 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 debugging symbols for libsharedlib1.
```

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 are no needs 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 lists of binary packages.

The generated dependency lists of binary packages (v=2.0):

```
 $ dpkg -f debhello-dbg_2.0-1_amd64.deb pre-depends depends recommends confli...
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-dbg_2.0-1_amd64.deb pre-depends depends recommends c...
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 (multi-binary)

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 using the 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,-dbg,libsharedlib1-dbg'
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,-dbg,libsha...
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 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)"

override_dh_strip:
        dh_strip -Xlibsharedlib1 --dbg-package=debhello-dbg
        dh_strip -Xdebhello --dbg-package=libsharedlib1-dbg
```

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

override_dh_strip:
        dh_strip -Xlibsharedlib1 --dbg-package=debhello-dbg
        dh_strip -Xdebhello --dbg-package=libsharedlib1-dbg
```

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: extra
Maintainer: Osamu Aoki <osamu@debian.org>
Build-Depends: cmake, debhelper (>=9)
Standards-Version: 3.9.6
Homepage: http://anonscm.debian.org/cgit/collab-maint/debmake-doc.git/

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 the 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 the 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 the 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.

Package: debhello-dbg
Section: debug
Architecture: any
Multi-Arch: same
Depends: debhello (= ${binary:Version}), ${misc:Depends}
Description: example debugging package for debhello
 This is an example package to demonstrate the 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 debugging symbols for debhello.

Package: libsharedlib1-dbg
Section: debug
Architecture: any
Multi-Arch: same
Depends: libsharedlib1 (= ${binary:Version}), ${misc:Depends}
Description: example debugging package for libsharedlib1
 This is an example package to demonstrate the 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 debugging symbols for libsharedlib1.
```

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 are no needs 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 lists of binary packages.

The generated dependency lists of binary packages (v=2.1):

```
 $ dpkg -f debhello-dbg_2.1-1_amd64.deb pre-depends depends recommends confli...
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-dbg_2.1-1_amd64.deb pre-depends depends recommends c...
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 Internationalization

Here is an example of updating the simple upstream C source **debhello-2.0.tar.gz** presented in 第 8.11 节 for the internationalization (i18n) and creating the updated upstream C source **debhello-2.0.tar.gz**.

In the real situation, the package should be already 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 BTS system to the **po/** directory and to update the language list in the **po/LINGUAS**.

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 the 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-1.9/debhello-2.0-pkg2/step151.cmd: line 2: SOURCE_DATE_EPO...
total 60
-rw-r--r-- 1 pbuilder pbuilder   494 Mar 27 13:34 ChangeLog
-rw-r--r-- 1 pbuilder pbuilder 17577 Mar 27 13:34 Makefile.in.in
-rw-r--r-- 1 pbuilder pbuilder  3376 Mar 27 13:34 Makevars.template
-rw-r--r-- 1 pbuilder pbuilder    59 Mar 27 13:34 POTFILES.in
-rw-r--r-- 1 pbuilder pbuilder  2203 Mar 27 13:34 Rules-quot
-rw-r--r-- 1 pbuilder pbuilder   217 Mar 27 13:34 boldquot.sed
-rw-r--r-- 1 pbuilder pbuilder  1337 Mar 27 13:34 en@boldquot.header
-rw-r--r-- 1 pbuilder pbuilder  1203 Mar 27 13:34 en@quot.header
-rw-r--r-- 1 pbuilder pbuilder   672 Mar 27 13:34 insert-header.sin
-rw-r--r-- 1 pbuilder pbuilder   153 Mar 27 13:34 quot.sed
-rw-r--r-- 1 pbuilder pbuilder   432 Mar 27 13:34 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** from the **po/Makevars.template**.
po/Makevars (i18n):

```
 ... hack, hack, hack, ...
 $ diff -u po/Makevars.template po/Makevars
--- po/Makevars.template        2017-09-17 08:40:22.686980438 +0000
+++ po/Makevars 2017-09-17 08:40:25.994904688 +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 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 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**.
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: 2017-09-17 08:40+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 Japanese translation.

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, I am %s!\n"
msgstr "Bonjour, je suis %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 the Makefile (v=3.0),
- the Python3 script with the distutils (v=3.1),
- the C source with the Makefile.in + configure (v=3.2),
- the C source with the Autotools (v=3.3), and
- the C source with the 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
$ sudo apt-get build-dep debmake-doc
$ cd debmake-doc*
$ 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 the 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 the 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 the 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 untared 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 optional arguments:

-h, --help show this help message and exit.

-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 the “**3.0 (native)**” format package.

If you are thinking to package 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 the “**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 the 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 all cases 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* set the Debian package name.

-u *version*, --upstreamversion *version* set the upstream package version.

-r *revision*, --revision *revision* set the Debian package revision.

-z *extension*, --targz *extension* set the tarball type, *extension*=(**tar.gz**|**tar.bz2**|**tar.xz**) (alias: **z**, **b**, **x**)

-b *”binarypackage[:type],*···”, --binaryspec *”binarypackage[:type],*···” set the binary package specs by the comma separated list of *binarypackage:type* pairs, e.g., in the full form “**foo:bin,foo-doc:doc,libfoo1:lib,libfoo1-dbg:dbg,libfoo-dev:dev**” or in the short form “**,-doc,libfoo1,libfoo1-dbg, libfoo-dev**”.

Here, *binarypackage* is the binary package name; and the optional *type* is chosen from the following *type* values:

- **bin**: C/C++ compiled ELF binary code package (any, foreign) (default, alias: **””**, i.e., *null-string*)
- **data**: Data (fonts, graphics, ···) package (all, foreign) (alias: **da**)
- **dbg**: Debug symbol package (any, same) (alias: **db**) (deprecated for strech and after since the -dbgsym package is automatically generated)
- **dev**: Library development package (any, same) (alias: **de**)
- **doc**: Documentation package (all, foreign) (alias: **do**)
- **lib**: Library package (any, same) (alias: **l**)
- **perl**: Perl script package (all, foreign) (alias: **pl**)
- **python**: Python script package (all, foreign) (alias: **py**)
- **python3**: Python3 script package (all, foreign) (alias: **py3**)
- **ruby**: Ruby script package (all, foreign) (alias: **rb**)
- **script**: Shell script package (all, foreign) (alias: **sh**)

The pair values in the parentheses, such as (any, foreign), are the **Architecture** and **Multi-Arch** stanza values set in the **debian/control** file.

In many cases, the **debmake** command makes good guesses for *type* from *binarypackage*. If *type* is not obvious, *type* is set to **bin**. For example, **libfoo** sets *type* to **lib**, and **font-bar** sets *type* to **data**, ···

If the source tree contents do not match settings for *type*, the **debmake** command warns you.

-e *foo@example.org*, --email *foo@example.org* set e-mail address.

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 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 the *file*. (This is not for everyday use.)

The *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 to run “**autoreconf -i -v -f**” for every package building. Otherwise, **autotools-dev** as *addon* is used as 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 extra configuration files as templates.

The number *n* changes which configuration templates are generated.

- **-x0**: bare minimum configuration files. (default if these files exist already)
- **-x1**: „ + desirable configuration files. (default for new packages)
- **-x2**: „ + interesting configuration files. (recommended for experts, multi binary aware)
- **-x3**: „ + unusual configuration template files with the extra **.ex** suffix to ease their removal. (recommended for new users) To use these as configuration files, rename their file names into ones without the **.ex** suffix.
- **-x4**: „ + 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 EXAMPLES

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 to 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 program may require the **dh-python** package.
- Autotools (Autoconf + Automake) build system may require **autotools-dev** or **dh-autoreconf** package.
- Ruby program may require the **gem2deb** package.
- Java program may require the **javahelper** package.
- 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指派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。下面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 DEBUG

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 ...
```

See README.developer in the source for more.

A.8 作者

Copyright © 2014-2015 Osamu Aoki <osamu@debian.org>

A.9 LICENSE

Expat 许可证

A.10 参见

The **debmake-doc** package provides the “Guide for Debian Maintainers” in the plain text, HTML and PDF formats under the **/usr/share/doc/debmake-doc/** directory.

Also, please read the original Debian New Maintainers’ Guide provided by the the **maint-guide** package.

See also **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), and **git-pbuilder**(1) manpages.