Debian 新維護人員手冊

版權 © 1998-2002 Josip Rodin

版權 © 2005-2015 Osamu Aoki

版權 © 2010 Craig Small

版權 © 2010 Raphaël Hertzog

本文件可在 GNU 通用公共許可證第二版或更高版本的條款規定下使用。

本文檔在撰寫過程中參考了以下兩篇文檔：

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

COLLABORATORS

	TITLE : Debian 新維護人員手冊	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Josip Rodin, Osamu Aoki, Aron Xu, 李凌, 郑原真, 陳侃如, 青木修, 且周默	April 20, 2020	
简体中文翻译		April 20, 2020	
简体中文翻译		April 20, 2020	
简体中文翻译		April 20, 2020	
繁簡轉換		April 20, 2020	
繁簡轉換		April 20, 2020	
简体中文翻译		April 20, 2020	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 從一條正確的路開始 **1**
1.1 Debian 的社會驅動力 . . . 1
1.2 開發時需要的軟件 . . . 3
1.3 開發時需要的文檔 . . . 4
1.4 到何處尋求幫助 . . . 4

2 第一步 **6**
2.1 Debian 軟件包構建流程 . . . 6
2.2 選擇你的程式 . . . 6
2.3 獲得程式，並且試用它 . . . 9
2.4 簡易構建系統 . . . 9
2.5 常见的可移植的構建系統 . . . 9
2.6 套件名稱和版本 . . . 10
2.7 設置 **dh_make** . . . 11
2.8 初始化外來 Debian 軟件包 . . . 11

3 修改原始碼 **13**
3.1 設置 **quilt** . . . 13
3.2 修復上游 Bug . . . 13
3.3 把文件安裝到目的位置 . . . 14
3.4 不一樣的函式庫名稱 . . . 16

4 `debian` 目錄中的必須內容 **17**
4.1 `control` . . . 17
4.2 `copyright` . . . 21
4.3 `changelog` . . . 22
4.4 `rules` . . . 23
4.4.1 `rules` 文件中的 Target . . . 23
4.4.2 默認的 `rules` 檔案 . . . 23
4.4.3 定製 `rules` 檔案 . . . 26

5 debian 目錄下的其他檔案 29
5.1 README.Debian . . . 29
5.2 compat . . . 30
5.3 conffiles . . . 30
5.4 *package*.cron.* . . . 30
5.5 dirs . . . 31
5.6 *package*.doc-base . . . 31
5.7 docs . . . 31
5.8 emacsen-* . . . 31
5.9 *package*.examples . . . 32
5.10 *package*.init 和 *package*.default . . . 32
5.11 install . . . 32
5.12 *package*.info . . . 32
5.13 *package*.links . . . 32
5.14 {*package*.,source/}lintian-overrides . . . 33
5.15 manpage.* . . . 33
5.15.1 manpage.1.ex . . . 33
5.15.2 manpage.sgml.ex . . . 33
5.15.3 manpage.xml.ex . . . 34
5.16 *package*.manpages . . . 34
5.17 NEWS . . . 34
5.18 {pre,post}{inst,rm} . . . 34
5.19 *package*.examples . . . 35
5.20 TODO . . . 35
5.21 watch . . . 35
5.22 source/format . . . 35
5.23 source/local-options . . . 36
5.24 source/options . . . 36
5.25 patches/* . . . 36

6 構建套件 38
6.1 完整的 (重) 構建 . . . 38
6.2 自動編譯系統 . . . 39
6.3 **debuild** 命令 . . . 40
6.4 pbuilder 套件 . . . 40
6.5 **git-buildpackage** 及其相似命令 . . . 41
6.6 快速重構建 . . . 42
6.7 命令層級 . . . 43

7 檢查套件中的錯誤 **44**
7.1 詭異可疑的改動 . . . 44
7.2 校驗軟件包安裝過程 . . . 44
7.3 檢驗軟件包的 maintainer scripts . . . 44
7.4 使用 `lintian` . . . 45
7.5 **debc** 命令 . . . 45
7.6 **debdiff** 命令 . . . 46
7.7 **interdiff** 命令 . . . 46
7.8 **mc** 命令 . . . 46

8 更新套件 **47**
8.1 新的 Debian 版本 . . . 47
8.2 檢查新上游版本 . . . 48
8.3 新上游版本 . . . 48
8.4 更新打包風格 . . . 49
8.5 UTF-8 轉換 . . . 50
8.6 對更新套件的幾點提示 . . . 50

9 上傳套件 **51**
9.1 上傳到 Debian 倉庫 . . . 51
9.2 在上傳時包含 `orig.tar.gz` 檔案 . . . 52
9.3 跳過的上傳 . . . 52

A 高級打包 **53**
A.1 共享庫 . . . 53
A.2 管理 `debian/`*`package`*`.symbols` . . . 54
A.3 多體繫結構 . . . 55
A.4 構建共享庫包 . . . 56
A.5 Debian 本土軟件包 . . . 57

Chapter 1

從一條正確的路開始

這篇文檔試圖爲普通 Debian 用戶，和希望對 Debian 軟件包有所瞭解的開發人員講述如何製作 Debian 軟件包。在這裏，我們儘可能使用通俗的語言，並輔以大量實例來直觀地展示每一個細節。正如一句古羅馬諺語說得好：一例勝千言！

本教程文件已被重寫為另外的 Debian 維護者指導 (https://www.debian.org/doc/devel-manuals#debmake-doc) 文件，其中包含了更新的內容與更多實際例子。請使用新的教程作為主要的教程文件。

該文件在 Debian `Buster` 釋出時仍然可用，因為它已提供了許多語言的翻譯版本。該文件會在之後的 Debian 版本中被移除，因為其內容正在逐漸過時。[1]

Debian 的軟件包系統是使它躋身頂級發行版行列的重要原因之一。儘管已經有相當數量的軟件被打包成 Debian 的格式，但有時還是需要安裝一些不是這一格式的軟件。可能你正爲如何製作自己的軟件包而感到迷惑，也可能正認爲這麼做很難。如果你是一個剛剛接觸 Debian 的初學者，那麼是的，它的確很難；不過假如你真的只是一個初入此門的新手，現在大概也不會來讀這篇文檔了。:-) 你的確需要對 Unix 編程有所瞭解，但顯然沒必要是這方面的天才。[2]

對於 Debian 軟件包維護人員來說，有一件事是非常明確的：創建並維護一個 Debian 軟件包需要花費很多精力，所需的時間很可能遠不只是幾個小時。維護人員需要有良好的技術基礎，同時也需要十分勤奮，這樣才能保證我們的系統正常運行而不出現問題。

如果你在軟體包製作方面需要他人幫助，請閱讀節 1.4。

本文的最新版隨時都可以在 http://www.debian.org/doc/maint-guide/ 上和 `maint-guide` 軟件包裏找到。文檔的簡體中文翻譯可以在 `maint-guide-zh-cn` 軟件包裏找到。還有一點需要注意的是，這篇文檔的內容相對於當前的開發情況可能會有略微的延遲。

由於這篇文檔是一份手把手的教程，所以在一些重要的話題上會對每個步驟都做詳細的解釋。因而你可能覺得它們之中有一些與你的想法毫不相干。請準備好足夠的耐心來學習。同時我也有意地省略了某些不必要的細節，以使這篇文檔儘可能保持簡潔。

1.1 Debian 的社會驅動力

以下是一些有關 Debian 的社會動力學報告，希望它們有助於你掌握與 Debian 項目進行互動的方法。

- 我們都是志願者。
  - 任何人都不能把事情強加給他人。
  - 你應該主動地做自己的事情。

[1] 在寫這份文檔時，我們默認你使用 `squeeze` 操作系統。如果你需要在 `lenny` 系統上使用本文所記述的方法，則必須安裝 backports 倉庫中的 `dpkg` 和 `debhelper` 軟件包。

[2] 在 Debian Reference (http://www.debian.org/doc/manuals/debian-reference/) 中，你可以瞭解到使用 Debian 系統的一些基本信息和關於 Unix 編程的一些指引。

- 友好合作是我們前行的動力。
  - 你的貢獻不應致使他人過勞。
  - 只有當別人欣賞你的貢獻時，它才真正有價值。
- Debian 不是一所學校，沒有老師會自動地注意你。
  - 你需要有自學大量知識的能力。
  - 其他志願者的注意是非常稀缺的資源。
- Debian 在不斷進步。
  - Debian 期望你製作出高質量的軟件包。
  - 你應該適時改變自己來適應變化。

在 Debian 社區中有這幾類常見的角色：

- **Upstream author** (上游作者)：程序的原始作者。
- **Upstream maintainer** (上游維護者)：目前在上游維護程序代碼的人。
- **Maintainer** (軟件包維護者)：製作並維護該程序的 Debian 軟件包的人。
- **Sponsor** (保證人)：檢查內容後幫助維護者上傳軟件包到 Debian 官方倉庫的人。
- **Mentor** (指導者)：幫助維護者熟悉和深入打包的人。
- **Debian Developer** (DD)：Debian 社區的官方成員。DD 擁有向 Debian 官方倉庫上傳的全部權限。
- **Debian Maintainer** (DM)：擁有對 Debian 官方倉庫部分上傳權限的人。

注意，你不可能在一夜之間成爲 **Debian Developer**，因爲成爲 DD 所需要的遠不只是技術技巧。別因此氣餒，如果你的軟件包對其他人有用，你可以作爲軟件包的 **Maintainer**，通過一位 **Sponsor** 來上傳它，或者申請成爲 **Debian Maintainer**。

還有，要成爲 Debian Developer 不一定要創建新軟件包。對已有軟件做出貢獻也是成爲 Debian Developer 的理想途徑。眼下正有很多軟件包等着好的維護者來接手 (參看節 2.2)。

在這篇文檔裏，我們的重點在於打包的技術細節，所以請參考以下的文檔來瞭解 Debian 是如何運轉的，以及如何才能參與到其中：

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

1.2 開發時需要的軟件

在開始之前，你需要確認你是否已經正確安裝了開發所需要的附加套件。注意這些套件不包含任何已經被標記爲 essential 或 required ——我們假設你已經安裝了它們。

以下這些軟體包已經隨標準的 Debian 安裝過程進入了系統，所以你可能不需要再動手安裝它們（以及任何附加的依賴軟體包）。然而，你還是應該用 aptitude show *package* 或者 dpkg -s *package* 來檢查一下。（譯註：apt show PACKAGE 亦可）

在你的開發環境中，需要安裝的最重要的軟件包是 build-essential。一旦你嘗試安裝該包，它將 拉來其他基本構建環境所需的基本軟件。

對於某些類型的軟件，以上的就是所需要的全部。然而還有一組軟件包雖不是對於所有軟件包都必須，卻可能對你有用或被你的軟件包所需要：

- autoconf、automake 和 autotools-dev - 很多新程序使用 configure 腳本和 Makefile 文件來幫助預處理程序。(參看 info autoconf、info automake)。autotools-dev 則用於保持指定的自動配置文件爲最新，並帶有關於使用那些文件的最佳方法的文檔。
- dh-make 和 debhelper - dh-make 是用於創建我們示例軟件包骨架所必須的，它會使用 debhelper 中的一些工具來創建軟件包。他們不是創建軟件包所必須的，但對新維護人員而言，我們 強烈推薦使用。它使得整個過程極爲簡化，並易於在將來維護。(參看 dh_make(8)、debhelper(1)、/usr/share/doc/debhelper/README)[3]

  新的 debmake 可以作為標準 dh-make 的代替品。debmake 能做的事情更多，並且擁有包含非常多打包例項的 HTML 文件。文件可以通過 debmake-doc 軟體包獲取。
- devscripts - 此軟件包提供了一些非常好非常有用的腳本幫助維護者，但他們並非構建軟件包所必須。此軟件包所推薦或建議的軟件包都值得一看。(參看 /usr/share/doc/devscripts/README.gz)
- fakeroot - 這個工具使你可以在編譯過程中必要的時候以普通使用者來模擬 root 使用者環境。(參看 fakeroot(1))
- file - 這個小程序可以檢測文件的類型。(參看 file(1))
- gfortran - GNU Fortran 95 編譯器，如果你的程序是用 Fortran 編寫的則必須此軟件包完成編譯。(參看 gfortran(1))
- git - 此軟件包提供了用於快捷處理大型項目的著名版本控制系統 - git。它被廣泛用於各種開源項目，最著名的是 Linux 內核項目。(參見 git(1), git Manual (/usr/share/doc/git-doc/index.html).)
- gnupg - 讓你可以使用 數字簽名簽署你的軟體包。當你想把它分發給其他人時這一點特別重要。如果你要把你的成果加入到 Debian 發行版中，那這是必須的步驟。(參看 gpg(1).)
- gpc - GNU Pascal 編譯器。如果你的程序是用 Pascal 寫的則需要此軟件包。值得一提的是 fp-compiler，Free Pascal 編譯器 (FPC)，也能夠很好地勝任。(參見 gpc(1), ppc386(1).)
- lintian - Debian 軟體包檢查工具，使你可以在編譯軟體包後知道它是否犯了常見的錯誤，並對其找到的錯誤進行解釋。(參見 lintian(1), Lintian User's Manual (https://lintian.debian.org/manual/index.html) .)
- patch - 這是一個非常有用的工具，它可以把 **diff** 程序生成的差異清單文件應用到原先的文件上，從而生成一個補丁版本。(參看 patch(1))
- patchutils - 此套件提供了一些可以幫助處理補丁的工具，如 **lsdiff**、**interdiff** 和 **filterdiff** 命令。
- pbuilder - 此軟體包提供了建立和維護 **chroot** 環境的工具。在它的 **chroot** 環境中編譯 Debian 軟體包可以檢查編譯依賴是否合適，並避免 FTBFS (Fails To Build From Source，原始碼編譯失敗) 的 Bug。(參看 pbuilder(8) 和 pdebuild(1))
- perl - Perl 是現今類 Unix 系統中使用最普遍的解釋型腳本語言，它常被稱作 Unix 的瑞士軍刀。(參看 perl(1))
- python - Python 是 Debian 系統中另一個最常用的解釋型腳本語言，它擁有着可圈可點的強大功能和十分清晰的語法。(參看 python(1))
- quilt - 此軟件包幫助你管理一系列的補丁。它們被以邏輯棧的方式組織在一起。你可以 apply (=push)、un-apply (=pop) 或簡單地刷新它們然後再放入棧內。(參看 quilt(1), and /usr/share/doc/quilt/quilt.pdf.gz.)

[3] 還有幾個類似但更針對某一類軟件的軟件包，如 dh-make-perl、dh-make-php 等。

- `xutils-dev` - 一些通常用於 X11 的程序，用於使用其宏功能生成 `Makefile` 文件。(參看 imake(1)、xmkmf(1))

以上給出的簡短描述僅僅是爲了使你對這些軟件包有一個基本的印象。在繼續前請詳細閱讀每個程序（包括通過依賴關係安裝的程序，比如 **make**）的文檔，至少瞭解其一般的用途和用法。現在看來這是一項耗時巨大的任務，但在接下來的工作中你將爲你閱讀了它們而感覺到 非常愉快。如果一會你遇到一些特定的問題，我會建議你重新閱讀上面提到的文檔。

1.3 開發時需要的文檔

以下是 非常重要的文件，你應該在讀本文件時同時參考它們：

- `debian-policy` - the Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) 包含了對 Debian 軟件倉庫、操作系統設計事宜、文件系統層級標準 (FHS，Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html)，講述每個文件和目錄應該放在哪裏) 等的描述。對於你而言，最重要的是它描述了軟件包要進入官方倉庫前必須滿足的條件。(請參見 `/usr/share/doc/debian-policy/policy.pdf.gz` 和 `/usr/share/doc/debian-policy/fhs/fhs-3.0.pdf.gz` 的本地副本)
- `developers-reference` - Debian 開發者參考 (http://www.debian.org/doc/devel-manuals#devref) 描述了打包所需的包含技術細節在內的全部詳細資訊，如倉庫結構、如何重新命名/丟棄/接手軟體包、如何進行 NMU(非維護者上傳)、如何管理 Bug 以及打包最佳實踐、何時向何處上傳等。(參見 `/usr/share/doc/developers-reference/developers-reference.pdf` 的本地副本)

以下是 重要的文檔，你應該在讀本文檔時同時參看它們：

- Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) 為 the GNU Build System known as the GNU Autotools 中最重要的工具——Autoconf、Automake、Libtool 和 gettext 提供了很好的文件。
- `gnu-standards` - 此軟件包包含了 GNU 項目中的兩篇文檔：GNU Coding Standards (http://www.gnu.org/prep/standards/html_node/index.html) 和 Information for Maintainers of GNU Software (http://www.gnu.org/prep/maintain/html_node/index.html)。儘管 Debian 不要求遵守這些規範，它們作爲綱領和共識仍然很有幫助。(參見 `/usr/share/doc/gnu-standards/standards.pdf.gz` 和 `/usr/share/doc/gnu-standards/maintain.pdf.gz` 的本地副本)

若本文檔所敘述的內容與 Debian Policy Manual 或 Debian Developer's Reference 有不符，則按照後兩者的要求進行，並向 `maint-guide` 軟件包提交 Bug 報告。

以下是替代性的教程文件，你可以在讀本文件時同時參看它們：

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

1.4 到何處尋求幫助

在你決定到公共場合提問之前，請先閱讀這些（個）不錯的文件：

- 所有相關軟件包的 `/usr/share/doc/`*package* 目錄之中的文件
- 所有相關命令的 **man** *command* 手冊頁
- 所有相關命令的 **info** *command* info 頁
- 郵件列表歸檔 debian-mentors@lists.debian.org (http://lists.debian.org/debian-mentors/)
- 郵件列表歸檔 debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/)

你可以在搜索引擎中搜索時使用類似這樣的字符串:`site:lists.debian.org` 來限制域名以提高效率。

製作小的測試軟件包是學習打包的好方法，仔細查看維護較好的軟件包則是瞭解他人如何製作軟件包的最佳辦法。

如果你仍然對打包存有疑問，並且在文檔和 WEB 資源中都不能找到答案，你可以交互式地向他們提問:

- http://lists.debian.org/debian-mentors/ (http://lists.debian.org/debian-mentors/) 郵件 debian-mentors@lists.debian.org (http://lists.deb debian-mentors/) . (該郵件列表是新手專區。)
- debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) . (該郵件列表彙集各路神通。)
- IRC (http://www.debian.org/support#irc) 比如 `#debian-mentors`。
- 專注於某個軟件包集合的團隊。(完整列表參見 https://wiki.debian.org/Teams (https://wiki.debian.org/Teams))
- 特定語言的郵件列表,比如 debian-devel-{french,italian,portuguese,spanish}@lists.debian.org 或 debian-devel@debian.or.jp. (完整列表參見 https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) 和 https://lists.debian.org/users.html (https://list users.html))

如果你付出了一定的努力並且提問得當，那麼有經驗的 Debian 開發者會很樂意幫助你。

當你收到一個 Bug 報告後 (沒錯，真正的 Bug 報告！)，你需要研究 Debian Bug Tracking System (http://www.debian.org/-Bugs/) (Debian Bug 跟蹤系統，BTS) 並閱讀相關的文檔以便高效處理這些報告。我推薦閱讀 Developer's Reference, 5.8. "Handling bugs" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling)

即使上邊的那些問題都解決了，也不能高興得太早。為什麼？因為幾個小時或幾天內就會有人開始使用你的軟體包，如果你犯了某些嚴重的錯誤，將會被無數生氣的 Debian 使用者進行郵件轟炸……哦，當然這只是開個玩笑。:-)

放鬆一點並準備好處理 Bug 報告，在你的套件完全符合 Debian 的各項規範前還需要付出很多努力，處理 Bug 也是對你很好的鍛煉 (再一次提醒，閱讀那些 必須的文件來瞭解詳情)。祝你好運！

Chapter 2

第一步

讓我們嘗試創建一個自己的軟件包 (或者，“收養”一個已存在的軟件包則更好)。

2.1 Debian 軟件包構建流程

如果你要基於某個上遊程序構建軟件包，那麼典型的 Debian 軟件包構建流程就會包含生成幾個特定的文件，如下：

- 獲取上游軟件的拷貝，通常爲壓縮過的 tar 格式。
  - *package-version*.tar.gz
- 在上遊程序的 debian 目錄下添加 Debian 特定的打包修改，並以 3.0 (quilt) 格式創建一個非本地源碼包。(也就是指用於構建 Debian 軟件包的輸入文件集合)
  - *package_version*.orig.tar.gz
  - *package_version-revision*.debian.tar.gz[1]
  - *package_version-revision*.dsc
- 從 Debian 源碼包構建 Debian 二進制包；二進制包的格式通常是 .deb (或者 .udeb，Debian Installer 專用)
  - *package_version-revision_arch*.deb

請注意，在 Debian 軟件包文件名中，分隔 *package* 和 *version* 的字符從 tarball 名稱的 - (連字符) 換成了 _ (下劃線)。

在上述的文件名中，將 *package* 部分替換爲 **package name**，將 *version* 部分替換爲 **upstream version**，將 *revision* 部分替換爲 **Debian revision**，以及將 *arch* 部分替換爲 **package architecture**，根據 Debian 方針手冊。[2]

本大綱中的每一步都會在後續的章節中輔以詳細的例子進行解釋。

2.2 選擇你的程式

可能你已經選好了要製作的軟件包。第一件要做的事是檢查它是否已經存在於發行版倉庫中了，參考方法如下：

- **aptitude** 命令

[1] 對於舊式的 1.0 格式非本地 Debian 源碼包，應當使用 *package_version-revision*.diff.gz。

[2] 參見 5.6.1 ”Source” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source)，5.6.7 ”Package” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Package)，以及 5.6.12 ”Version” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version)。**package architecture** 遵循 Debian Policy Manual, 5.6.8 ”Architecture” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) 並且會在軟件包構建的過程中被自動分配。

- the Debian packages (http://www.debian.org/distrib/packages) 頁面
- the Debian Package Tracker (https://tracker.debian.org/) 頁面

如果軟件包已經存在，直接安裝就好了！:-) 如果它是被 抛棄 **(orphaned)** 的——也就是說它的維護者被設置爲 Debian QA Group (http://qa.debian.org/)，那麼你可以嘗試接手維護它。你也可以“收養”維護者發出“Request for Adoption” (**RFA**) 請求的軟件包。[3]

軟件包歸屬狀態有這幾種：

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

作爲旁註必須指出，Debian 已經擁有了絕大多數類型軟件的軟件包，倉庫中軟件包的數量也遠遠超過了有上傳權限的貢獻者的數量。因此，爲已經在倉庫中的軟件包貢獻力量是非常受其他開發者歡迎的 (且更容易獲得 sponsorship)[4]。你可以通過非常多的方式來實現這一目的：

- 接手被抛棄而仍然被很多人使用的套件。
- 加入 打包小組 (http://wiki.debian.org/Teams)。
- 爲某些常用的套件分類 Bug。
- 在需要時準備 QA 或 NMU 上傳 (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload)。

如果你有能力“領養”那個軟體包，那就先下載 (使用 `apt-get source` *packagename* 或其他類似的工具) 並分析它的原始碼。這篇文件不會詳細說明如何領養軟體包，不過幸運的是，領養軟體包時，打包的起始工作已經有人完成，接手的工作應比從頭開始輕鬆得多。儘管如此也請您不要輕敵，請繼續閱讀，下面給出的建議會對你很有幫助。

如果你要製作的套件是全新的，並且希望它出現在 Debian 中，請按照以下的步驟進行：

- 首先，你必須知道這個軟件的可用性，並且需要試用一段時間。
- 一定要在 Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp/being_packaged) 上仔細檢視，以確定並沒有其他人已經開始相關的工作。如果沒有的話，則可以提交一份 ITP (Intent To Package，打包意向) Bug 報告給 `wnpp` 虛包 (可以使用 **reportbug**)。如果你看到已經有人在處理相關事宜，則在有需要的情況下再聯絡他 (們)。如果他 (們) 不需要你的幫助，那就尋找其他你感興趣，而且沒有人維護的軟體包咯。
- 該軟件 必須有一個許可證。
  - 對於 `main` 類的軟件，Debian 方針要求它 完全兼容 **Debian Free Software Guidelines** （**Debian** 自由軟件準則） (DFSG (http://www.debian.org/social_contract#guidelines)) 並且 不要求用 **`main`** 類以外的軟件來編譯或執行。這是最理想的狀況。
  - 對於 `contrib` 類的軟件，其許可證必須滿足 DFSG 的全部條件，但可以依賴於 `main` 之外的軟件包以完成編譯或運行。
  - 對於 `non-free` 類的軟件，其許可證可以不滿足 DFSG 中的一些條件，但至少 必須是可分發的。
  - 如果你不清楚你的軟體應該分入哪一類，則把許可證文本發送到 debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) 請他人提出意見。
- 程式 不能給 Debian 系統帶來安全或維護問題。

[3] 參見 Debian Developer's Reference 5.9.5. "Adopting a package" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#adopting) .
[4] 當然了，總有值得打包的新軟件。

- 程式應當帶有良好的文件，最好是原始碼也容易理解（比如，不混亂）。
- 你應該與程序的作者取得聯繫問一下他是否認爲程序應當被打包，以及他是否對 Debian 友好。能夠詢問作者關於程序的任何問題是非常重要的，所以不要嘗試打包一個無人維護的軟件。
- 程序一定 不應該 setuid 到 root。更好的情況是它不 setuid 或 setgid 到任何用戶或組。
- 程序不應該是守護進程，也不應該進入 `*/sbin` 目錄或者以 root 打開任何端口。

當然，這些都是為了安全，並試圖避免你在，比如 setuid 守護程序等問題上犯錯誤而激怒了使用者... 當你在打包方面有了更多經驗時，就能夠處理這樣的軟體包了，不必著急。

我們鼓勵你，作爲一個新維護人員，選擇易於打包和維護的軟件包，而不鼓勵選擇複雜的軟件包。

- 簡單軟件包
  - 單二進制軟件包，arch = all（比如像壁紙那樣的資料集）
  - 單二進制軟件包，arch = all（用解釋型語言編寫的可執行腳本文件，比如 POSIX shell）
- 中等複雜軟件包
  - 單二進制軟件包，arch = any（用 C/C++ 等語言編寫的 ELF 二進制可執行文件）
  - 複合二進制軟件包，arch = any + all（包含 ELF 二進制可執行程序 + 文檔的軟件包）
  - 既不是 `tar.gz` 也不是 `tar.bz2` 格式的上有源代碼
  - 源代碼中包含不可分發的內容。
- 高複雜軟件包
  - 被其他軟件包使用的解釋器模塊包
  - 被其他軟件包使用的一般 ELF 庫文件包
  - 複合二進制的軟件包，包含 ELF 庫文件包
  - 多上游的源碼包
  - 內核模塊軟件包
  - 內核補丁軟件包
  - 包含冷門維護者腳本的軟件包

打包高複雜軟件包並非難如登天，但需要更多知識。你應該針對每一個複雜特性來搜尋針對性的指南。比如，一些語言有它們自己的子策略文檔：

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

還有一句拉丁諺語：*fabricando fit faber*（熟能生巧）。我們 強烈建議你在閱讀這篇教程的時候，用一個簡單的軟體包來對所有的 Debian 打包步驟進行實驗。跟隨下邊的步驟您就可以建立一個微不足道的軟體包 `hello-sh-1.0.tar.gz`，而且這會是一個非常好的開端：[5]

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

[5]不用擔心失蹤的 `Makefile`。你可以參照節 5.11，簡單地通過 **debhelper** 來安裝 **hello** 程序，或者修改上游源代碼來添加帶有 `install` 目標的新`Makefile`，參照章 3

2.3 獲得程式，並且試用它

第一件要做的事就是找到並下載原始的源代碼。我們假定你已經從作者的主頁上找到了它的源代碼。Unix 下的自由軟件源代碼通常是以 **tar**+**gzip** 格式 (擴展名爲 `.tar.gz`) 或 **tar**+**bzip2** 格式 (擴展名爲 `.tar.bz2`) 或 **tar**+**xz** (擴展名爲 `.tar.xz`) 的形式提供的。通常歸檔文件中包含了一個名爲 *`package-version`* 的子目錄，裏麪包含了全部的源代碼。

如果最新版本的原始碼可通過像 Git，Subversion，CVS 這樣的版本控制系統獲得的話，你可以用 `git clone`，`svn co`, 或 `cvs co` 來下載它，然後將它重新打包壓縮為 **tar**+**gzip** 格式，同時別忘了 `--exclude-vcs` 選項喲。

如果你的程序源代碼是以其他形式提供的 (比如文件名以 `.Z` 或 `.zip` 結尾[6])，則使用合適的工具將其解包，再重新打包。

如果你的程序源代碼中包含一些不符合 DFSG 的內容，你應當解包後移除它們，再以添加了 `dfsg` 的上游版本號重新打包。

作爲示例，我將使用一個名爲 **gentoo** 的程序，它是一個 GTK+ 文件管理器。[7]

在你的用戶主目錄下創建一個子目錄，命名爲 `debian` 或 `deb` 或其他你喜歡且合適的名字 (本例中使用 `~/gentoo`)。把下載好的歸檔文件放在其中並解包 (使用 `tar xzf gentoo-0.9.12.tar.gz` 命令)。要確定解包過程中沒有發生錯誤，即便是有一點 不恰當也不行，因爲在別人的系統上解包這些文件時，可能他們的工具並不忽略這些反常的現象，於是就會出現問題。在你的終端屏幕上，應該看到如下的情形。

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

現在又有了一個新的子目錄，名爲 `gentoo-0.9.12`。進入該目錄並 徹底讀完其中的文檔。通常情況下這些文檔被命名爲 `README*`、`INSTALL*`、`*.lsm` 或 `*.html`。你必須找到關於如何正確編譯和安裝程序的指導 (最可能的是他們會默認你希望把程序安裝到 `/usr/local/bin` 目錄；但事實上你不能那樣做，詳細的內容稍後將在節 3.3 中說明)。

開始打包時原始碼目錄應當是絕對乾淨 (原始) 的，或者直接使用剛剛解壓縮得到的原始碼。

2.4 簡易構建系統

帶有 `Makefile` 文件的簡單程序可以很容易地使用 `make` 來編譯。[8] 其中的一些還支持 `make check`，這可以完成一系列自檢。編譯好後可以使用 `make install` 來將程序安裝到目標目錄。

現在嘗試編譯和運行你的程式，確保它工作正常，且在安裝和運行時不會導致其他問題。

你還可以運行 `make clean` (或更好的 `make distclean`) 來清理編譯目錄。有時還會帶有 `make uninstall` 用以反安裝已經安裝了的檔案。

2.5 常见的可移植的構建系統

非常多的自由軟件是使用 C 和 C++ 語言編寫的。其中的有很多使用 Autotools 或 CMake 來使其可以在不同平臺上移植。這些工具首先用於生成 `Makefile` 和其他必須的源文件，然後這些程序可以使用正常的 `make; make install` 來編譯和安裝。

[6] 當檔案副檔名不足以判斷檔案類型時，可以使用 **file** 命令來判斷。

[7] 需要注意的是，這個程序已經被打包好了。當前的版本 (http://packages.qa.debian.org/g/gentoo.html) 使用 Autotools 作爲其構造 (build structure)，並且已經和下邊的例子大不相同，下邊的例子基於版本 0.9.12 。

[8] 許多新時代的程式都配有一個叫做 `configure` 的指令碼。執行它的時候會生成一個為你的計算機專門定製的 `Makefile` 。

Autotools 是 GNU 編譯系統工具，包括 Autoconf、Automake、Libtool 和 gettext。你可以通過 `configure.ac`、`Makefile.am` 和 `Makefile.in` 等文件來識別這種類型的源代碼。[9]

使用 Autotools 的第一步是在上游作者在代碼中運行 `autoreconf -i -f`，然後把生成的文件同源代碼一起分發。

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

編輯 `configure.ac` 和 `Makefile.am` 等檔案需要一些關於 **autoconf** 和 **automake** 的知識。參考 `info autoconf` 和 `info automake`。

使用 Autotools 的第二步是用戶獲得分發的源代碼後在源碼目錄下運行 `./configure && make` 來將其編譯成爲 ***`binary`*** (二進制可執行程序)。

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

你可以改變 `Makefile` 文件中的許多設置，比如修改默認的文件安裝位置 (使用 `./configure --prefix=/usr`)。

儘管不是必須的：你還可以使用 `autoreconf -i -f` 來更新 `configure` 和其他相關文件，這樣做有可能提高源代碼的兼容性。[10]

CMake 是另一個可選的構建系統，你可以通過 `CMakeLists.txt` 文件來識別使用它的源代碼。

2.6 套件名稱和版本

如果上游源代碼以像 `gentoo-0.9.12.tar.gz` 的形式分發，你可以用 `gentoo` 作爲 (源代碼) 軟件包名，並用 `0.9.12` 作爲 上游版本。它們會被 `debian/changelog` 這個文件用到；該文件一會會在節 4.3 部分詳細描述。

雖然此法在大部分情況下能顯靈，但你仍需要根據 Debian 政策 (Debian Policy) 以及約定俗成的做法來調整 軟體包名和 上游版本。

在 軟體包名裡只能含有小寫字母 (`a-z`), 數字 (`0-9`), 加號 (`+`) 和減號 (`-`)，以及點號 (`.`)。軟體包名最短長度兩個字元；它必須以字母開頭；它不能與倉庫軟體包名發生衝突。還有，把軟體包名的長度控制在 30 字元以內是明智之舉。[11]

如果上游在它的名稱中使用了一些通用術語比如 `test-suite`，那麼將其重命名，以顯式指明其內容並避免命名空間污染。[12]

你應該讓 **upstream version** (上游版本號) 只包含字母和數字 (`0-9A-Za-z`), 以及加號 (`+`), 波浪號 (`~`), 還有點號 (`.`)。它必須以數字開頭 (`0-9`)。[13] 如果可能的話，最好把它的長度控制在 8 字元以內。[14]

如果上游不使用像 `2.30.32` 這樣的常規版本格式，而是用類似 `11Apr29` 這樣的日期作為版本，類似於隨機的代號字串，或者以 VCS 的雜湊值作為版本號的一部分，那麼請確認將其從 **upstream version** 中移除。為此作出的改動資訊可

[9] Autotools 這個龐然大物顯然已經超出本教程的討論範圍，畢竟本文主要提供關鍵字和提示。如果你需要使用 Autotools，請認真研讀 Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) 以及 `/usr/share/doc/autotools-dev/README.Debian.gz` 的本地副本。

[10] 你可以用 `dh-autoreconf` 軟件包來將這個過程自動化。參見節 4.4.3.

[11] 在 **aptitude** 工具中，軟件包名字段的默認最大長度爲 30。而對於 90% 以上的軟件包來說，軟件包名都少於 24 個字符。

[12] 如果你遵循 Debian Developer's Reference 5.1. "New packages" (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage)，那麼在 ITP 過程中很容易遇到這樣的問題。

[13] 這一條更嚴格的規則能幫助你避免混淆文件名。

[14] **aptitude** 命令的版本字段默認長度爲 10。其中通常 Debian 修訂號和前置的連字符會消耗 2 個位置。對於 80% 以上的軟件包來說，上游版本小於 8 字符，Debian 修訂號小於 2 字符。對於 90% 以上的軟件包來說，上游版本小於 10 字符，Debian 修訂號小於 3 字符。

以記錄在 debian/changelog 檔案中。如果你需要發明一個版本字串，請使用 YYYYMMDD 這個格式作為上游版本，比如 20110429 。這會確保 **dpkg** 在升級軟體包時能夠正確解讀新版本。如果需要確保未來能夠平滑過渡到類似 0.1 這樣的版本號的話，那就請使用 0~YYMMDD 格式作為上游版本，例如 0~110429 。

版本字符串 [15] 可以用 dpkg(1) 來進行比較：

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

版本比較規則可以總結爲以下幾點：

- 字符串會被從頭到尾進行比較。
- 字母比數字大。
- 數字作爲整數進行比較。
- 字母按照 ASCII 編碼順序進行比較。
- 對於點號 (.)，加號 (+)，以及波浪號 (~) 則要應用特殊規則，具體如下：
  0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0~rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0

有一種比較棘手的情況，當上游釋出 gentoo-0.9.12-ReleaseCandidate-99.tar.gz 作爲 gentoo-0.9.12.tar.gz 的預發佈版本時，就需要確保升級工作妥當進行：重命名該上游源代碼爲 gentoo-0.9.12~rc99.tar.gz.

2.7 設置 dh_make

首先我們設置兩個環境變量，$DEBEMAIL 和 $DEBFULLNAME，這樣能使使大多數 Debian 維護工具能夠正確識別你用於維護軟件包的姓名和電子郵件地址。 [16]

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

2.8 初始化外來 Debian 軟件包

一般的 Debian 軟件包是由上遊程序產生的外來 Debian 軟件包。若你想要用上游源代碼 gentoo-0.9.12.tar.gz 創建一個外來 Debian 軟件包，你可以爲它創建一個初始的外來 Debian 軟件包，通過如下方法調用 **dh_make** 命令：

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

當然，請用你原始源碼歸檔的名字來替換 filename (文件名)。 [17] 詳情請參見 dh_make(8) 。

[15] 版本字符串可以是 **upstream version** (*version*), **Debian revision** (*revision*), 或者 **version** (*version-revision*)。關於 Debian 修訂號如何增長的信息，請參見節 8.1**Debian revision** 。

[16] 以下默認你以 Bash 作爲登陸 shell。如果你使用其他的 shell，例如 Z shell，那就使用它們的配置文件代替這裏提到的 ~/.bashrc。

[17] 如果上游原始碼已經提供了有內容的 debian 目錄，那麼帶上引數 --addmissing 來執行 **dh_make** 命令。新的原始碼包格式 3.0 (quilt) 的魯棒性 (Robust) 已經足夠優秀，以不至於輕易損壞。另外，你可能需要修改上游提供的內容，以滿足你的 Debian 軟體包之需。

你會看到一些輸出，詢問你想要建立什麼型別的軟體包。這裡的 Gentoo 被規劃為一個單一二進位制包——它僅僅產生一個二進位制包，亦即單個 .deb 檔案——於是我們就選擇第一項 (按 s 鍵)，認真閱讀螢幕上的輸出資訊，然後按 *ENTER* 鍵來確認。[18]

執行 **dh_make** 後，上一級目錄中自動創建了一份上游 tarball 的副本，名爲 gentoo_0.9.12.orig.tar.gz，這個文件和稍後介紹的 debian.tar.gz 在一起滿足了 Debian 非本地源代碼包的要求。

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

請注意 gentoo_0.9.12.orig.tar.gz 這個文件名的兩個關鍵特點:

- 軟件包名稱和版本是以字符 _（下劃線）分隔的。
- .tar.gz 擴展名前插有 .orig。

你應該可以注意到 debian 目錄下有了許多模板文件。這些文件將在章 4 和章 5 中一一解釋。你還應該明白，打包沒辦法變成全自動的過程。你還需要按照章 3 中的方法來爲 Debian 修改軟件包。此後，你還要按照章 6 中敘述的合適的方法來構建 Debian 軟件包，並按照章 7 中的方法進行測試，最終依照章 9 的介紹將其上傳。本教程將對所有的這些步驟進行解釋。

如果你在修改過程中不小心刪除或弄壞了某些模板檔案，你可以使用 **dh_make** 加上 --addmissing 參數來將其還原。

更新一個已存在的軟件包可能比較複雜，因爲它可能使用了舊技術。在學習基本功的階段，請只創建全新的軟件包；稍後的章 8中會有更細緻的講解。

請注意，原始碼中不必包含任何在節 2.4 或節 2.5 中談論到的編譯系統。就算原始碼包僅僅是一組影象之類的也可以，這時候這些檔案的安裝可以用 debhelper 的配置檔案來搞定，比如 debian/install (參見節 5.11)。

[18] 此處有這幾種選擇：s 代表單一二進制包，i 代表獨立於體繫結構的軟件包，m 代表複合二進制包，l 代表庫文件包，k 代表內核模塊包，n 代表內核補丁包，b 代表 cdbs 軟件包。本教程專注於使用 **dh** 命令 (來自 debhelper 軟件包) 來創建單一二進制包，但也會涉及到如何用它來創建獨立於體繫結構或複合二進制軟件包。軟件包 cdbs 提供了另一套可以代替 **dh** 命令的基礎打包腳本，不過對它的描述已經超出了我們的討論範圍。

Chapter 3

修改原始碼

請注意這裏沒有足夠的篇幅來描述修改上游原始碼的 全部細節，但是這裏介紹了基本的步驟和常見的問題。

3.1 設置 quilt

quilt 程序爲 Debian 打包工作提供了記錄上游源碼修改的基本方法。對默認配置加以少許修改往往非常有用，所以我們來創建一個別名 **dquilt**，以用於打包：添加以下幾行內容到 `~/.bashrc` 文件中。其中第二行可以給 **dquilt** 命令提供與 **quilt** 命令相同的 shell 補全特性：

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

現在按下面的方法來創建 `~/.quiltrc-dpkg` 文件：

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

參見 quilt(1) 以及 `/usr/share/doc/quilt/quilt.pdf.gz` 來獲取有關 **quilt** 命令用法的信息。

3.2 修復上游 Bug

假設你在上游的`Makefile`檔案中找到了一個錯誤,其中的`install: gentoo`應該修正為`install: gentoo-target`。

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

讓我們使用 **dquilt** 修復這個問題，並把補丁命名爲 `fix-gentoo-target.patch`。[1]

[1] `debian/patches` 目錄應當是在你按照前面所述的步驟運行 **dh_make** 時生成的。而在這個例子中我們新創建它，因爲假設的是在更新一個已存在的軟件包。

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

現在將 Makefile 修改爲如下的樣子：

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

使用 **dquilt** 將補丁生成到 debian/patches/fix-gentoo-targe.patch 並根據 DEP-3: Patch Tagging Guidelines (http://dep.debian.net/deps/dep3/) 添加描述：

```
$ dquilt refresh
$ dquilt header -e
... b'' 描 b''b'' 述 b''b'' 補 b''b'' 丁 b''
```

3.3 把文件安裝到目的位置

大多數第三方程序將其本身安裝在 /usr/local 目錄下。在 Debian 中，這是保留給系統管理員的私有位置，因此 Debian 軟件包不可以使用比如/usr/local/bin 這樣的目錄，而應當使用系統目錄比如 /usr/bin，以遵循文件系統層級結構標準：Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) (FHS)。

通常在自動編譯程序時使用 make(1) 程序，接着執行 make install 就可以把程序直接按照 Makefile 文件中的 install target 安裝到指定的位置。爲了使 Debian 能夠提供編譯好的二進制軟件包，編譯系統將文件安裝到一個臨時目錄中創建的文件系統樹的鏡像中，而非直接安裝到實際的目標位置。

普通程序安裝過程和 Debian 打包安裝過程二者的區別可以由 debhelper 軟件包中的 **dh_auto_configure** 和 **dh_auto_install** 透明地處理。但必須滿足以下條件：

- Makefile 文件應當遵循 GNU 的規定支持 $(DESTDIR) 變量[2]
- 源代碼必須遵循文件系統層級標準 (FHS)。

使用 GNU **autoconf** 的程序自動遵守了 GNU 的規定，這多少有利於打包過程的自動化。通過這項特點和其他啓發式處理，估計 debhelper 軟件可以直接打包約 90% 的軟件包而不需對編譯系統做出大的改變。所以打包也不是看起來那樣複雜。

如果你需要修改 Makefile 文件，就要確保其支持 $(DESTDIR) 變量。雖然默認情況下 $(DESTDIR) 變量沒有設置並且在程序安裝時會前置到每個文件的路徑中。打包腳本會將 $(DESTDIR) 設置爲臨時目錄。

對於從源碼包生成單個二進制包，**dh_auto_install** 將臨時目錄設置爲 debian/*package*。[3] 臨時目錄中的全部文件都將在安裝軟件包時被安裝到用戶系統，唯一的區別是 **dpkg** 會把文件安裝到真實的根目錄樹中，而不是你的工作目錄。

請記住，即使你的程序正確安裝到了debian/*package*，仍然要考慮將 .deb 軟件包文件安裝到根目錄下的情形。所以絕對不允許構建系統將諸如 /home/me/deb/*package-version*/usr/share/*package* 這種詭異的內容硬編碼到軟件包文件中。

以下是 gentoo 軟件包的 Makefile 文件中的相關部分[4]：

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

[3] 對於單個源碼包生成多個二進制包，**dh_auto_install** 將臨時目錄設置爲debian/tmp，而 **dh_install** 命令則將文件按照 debian/*package-1*.install 和 debian/*package-2*.install 等文件的描述將 debian/tmp 中的文件分別裝入 debian/*package-1* 和 debian/*package-2* 等臨時目錄用以創建 *package-1*_*.deb 和 *package-2*_*.deb 等二進制軟件包。

[4] 這只是一個演示 Makefile 正常形態的例子。如果 Makefile 是通過 **./configure** 命令生成的，那麼修復該類 Makefile 的方法是通過 **dh_auto_configure** 來執行 **./configure**，並帶上包括 --prefix=/usr 的默認選項。

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

可以看到檔案被放到了 /usr/local 下。按照上邊的解釋，該目錄被 Debian 保留作本地用途，所以請把它們按如下方式修改：

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

二進制文件、圖標和文檔等的更確切位置均已在文件層級標準 (FHS) 中作出了詳盡描述。本教程建議你快速瀏覽相關章節以獲取你打包需要用到的內容。

因此，我們應當把可執行二進制文件安裝到 /usr/bin 而非 /usr/local/bin，而 man 手冊頁則應放在 /usr/share/man/man1 而非 /usr/local/man/man1，依此類推。注意，gentoo 的 Makefile 裏沒有提及手冊頁，而按照 Debian Policy 的要求，每個程序都應當有一個手冊頁，我們將在稍後製作一個並安裝到 /usr/share/man/man1。

有些程式不使用 Makefile 變量定義路徑，這意味着你可能需要去編輯 C 程式原始碼來使他們使用正確的路徑。但是到哪裏去搜索，哪些纔是呢？你可以通過以下的方法找到它們：

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

grep 會遞歸搜索整個原始碼樹並告訴你所有匹配項的檔案名和行號。

編輯那些文件，在那些行中用 usr/lib 替換 usr/local/lib。這個過程可以用如下方法自動化完成：

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

如果你想要確認每一個替換操作，那麼下邊的方法可以讓你交互式地達成：

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

緊接着你應該找到 install target (通常搜索以 install: 開頭的行即可)，並把除 Makefile 頂部定義變量之外的目錄引用妥當修改。

原始的 gentoo 的 install target 是這樣：

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

讓我們來修復這個上游 BUG，並把修改使用 **dquilt** 命令記錄到debian/patches/install.patch。

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

使用你喜歡的編輯器按照以下內容爲 Debian 軟件包作修改：

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

你一定會注意到規則裏在其他命令前有了一個 `install -d` 命令。原始的 `Makefile` 文件中沒有它，因爲通常情況下當你執行 `make install` 命令時，`/usr/local/bin` 和用到的其他目錄早已存在於系統中。然而當我們要向新建的私有目錄樹中安裝時，我們必須創建其中的每一個目錄。

我們還可以在末尾添加上其他的內容，比如上游作者有時會省略的附加文件：

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

仔細檢查後如果沒有問題，使用 **dquilt** 創建 `debian/patches/install.patch` 補丁文件並添加對它的描述：

```
$ dquilt refresh
$ dquilt header -e
... b'' 描 b''b'' 述 b''b'' 補 b''b'' 丁 b''
```

現在你有了一格系列的補丁。

1. 修復上游 Bug：`debian/patches/fix-gentoo-target.patch`
2. Debian 特有的打包修改：`debian/patches/install.patch`

當進行任何非 Debian 特有的修改時，比如 `debian/patches/fix-gentoo-target.patch`，一定要向上遊作者進行反饋，以便上游作者方便在下一版本中以使更多人受益。同時請記住不要做出特別針對 Debian 或 Linux (甚至是 Unix！) 的修改，要使其可以移植，這會使你的修改更容易被接受。

注意你不一定要把 `debian/*` 都提交到上游。

3.4 不一樣的函式庫名稱

還有另外一個常見的問題：不同平臺之間的庫名稱常常因平臺而異。例如一個 `Makefile` 中可能引了用一個 Debian 系統中不存在的庫。這種情況下我們需要將其修改為 Debian 中存在的、提供完全相同功能的庫。

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

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

如果你的程序由於 `foo` 庫不存在，而 Debian 系統中的 `foo2` 庫提供了其等效，那麼你可以修復這個構建問題並將修改記錄到 `debian/patches/foo2.patch` 中，只需要將 `foo` 切換到 `foo2`:[5]

```
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e 's/-lfoo/-lfoo2/g' Makefile
$ quilt refresh
$ quilt header -e
... b'' 描 b''b'' 述 b''b'' 補 b''b'' 丁 b''
```

[5]如果從 `foo` 庫切換到 `foo2` 庫時要更改應用程序接口（API），這就要求我們修改源代碼來符合新的 API。

Chapter 4

debian 目錄中的必須內容

在程序源代碼目錄下有一個叫做 debian 的新的子目錄。這個目錄中存放着許多文件，我們將要修改這些文件來定製軟件包行爲。其中最重要的文件當屬 control, changelog, copyright, 以及 rules, 所有的軟件包都必須有這幾個文件。[1]

4.1 control

這個文件包含了很多供 **dpkg**、**dselect**、**apt-get**、**apt-cache**、**aptitude** 等包管理工具進行管理時所使用的許多變量。這些變量均在 Debian Policy Manual, 5 ”Control files and their fields” (http://www.debian.org/doc/debian-policy/ch-controlfields.html) 中被定義。

這裏的 control 文件是 **dh_make** 命令爲我們創建的：

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

(注：我爲它添加了行號。)

第 1–7 行是原始碼包的控制資訊。第 9–13 行是二進位制包的控制資訊。

第 1 行是原始碼套件的名稱。

第 2 行是該源碼包要進入發行版中的分類。

你可能已經注意到，Debian 倉庫被分爲幾個類別：main (自由軟件)、non-free (非自由軟件) 以及 contrib (依賴於非自由軟件的自由軟件)。在這些大的分類之下還有多個邏輯上的子分類，用以簡短描述軟件包的用途類別。admin 爲供系統管理員使用的程序，devel 爲開發工具，doc 爲文檔，libs 爲庫，mail 爲電子郵件閱讀器或郵件系統守護程序，net 爲網絡應用程序或網絡服務守護進程，x11 爲不屬於其他分類的爲 X11 程序，此外還有很多很多。[2]

[1] 在本章節中，只要不產生歧義，所有提及的 debian 目錄下的文件均會省去 debian/ 前綴以求簡潔方便。

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

我們將本例設置爲 x11。(`main/` 前綴是默認值，可以省略。)

第 3 行描述了用戶安裝此軟件包的優先級。[3]

- `optional` 優先級適用於與優先級爲 `required`、`important` 或 `standard` 的軟件包不衝突的新軟件包。

Section 和 Priority 常被如 **aptitude** 的前端所使用，以分類軟件包並選擇默認值。一旦你把軟件包上傳到 Debian，這兩項的值可以被倉庫維護人員修改，此時你將收到提示郵件。

由於這是一個常規優先級的軟體，並不與其他套件衝突，我們將優先級改爲 `optional`。

第 4 行是維護者的姓名和電子郵件地址。請確保此處的值可以直接用於電子郵件頭的 `To` 項。因爲一旦你將軟件包上傳至倉庫，Bug 跟蹤系統將使用它向你發送可能的 Bug 報告郵件。請避免使用逗號、“&”符號或括號。

第 5 行中的 `Build-Depends` 項列出了編譯此軟體包需要的軟體包。你還可以在這裡新增一行 `Build-Depends-Indep` 作為附加。[4] 有些被 `build-essential` 依賴的軟體包，如 `gcc` 和 `make` 等，已經會被預設安裝而不需再寫到此處。如果你需要其他工具來編譯這個軟體包，請將它們加到這裡。多個軟體包應使用半形逗號分隔。繼續閱讀二進位制包依賴關係以增進對這些行的語法的理解。

- 對於所有在 `debian/rules` 文件中使用 **dh** 命令打包的軟件包，必須在 `Build-Depends` 中包含 `debhelper (>=9)` 以滿足 Debian Policy 中對 `clean` target 的要求。
- 對於生成有標記過 `Architecture: any` 的二進位制包的原始碼包，它們將被 autobuilder 重構建。因為 autobuilder 過程在僅安裝 `Build-Depends` 中列出的程式前便執行 `debian/rules build` 中的內容 (參看節 6.2)，`Build-Depends` 欄位需要列出所有必須的編譯依賴，而 `Build-Depends-Indep` 則很少使用。
- 對於生成全標記 `Architecture: all` 二進制包的源碼包，`Build-Depends-Indep` 中應列出所有要求的軟件包，除非 `Build-Depends` 中已經列出，這樣以便滿足 Debian Policy 中對 `clean` target 的要求。

如果你不知道應該使用哪一個，則使用 `Build-Depends` 以保證安全。[5]

要找出編譯你的軟體所需的套件可以使用這個命令:

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

要手工地找到 ***/usr/bin/foo*** 的編譯依賴，可以執行

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

對於列出的每個庫 (例如 **libfoo.so.6**)，執行

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

接下來直接將相應的 `-dev` 版本的軟件包名稱放到 `Build-Depends` 項內。如果你使用 **ldd**，它也會報告出間接的庫依賴關係，這可能造成填寫依賴時畫蛇添足。

`gentoo` 需要 `xlibs-dev`、`libgtk1.2-dev` 和 `libglib1.2-dev` 才能編譯，所以我們將這些套件加在 `debhelper` 之後。

第 6 行是此套件所依據的 Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) 標準版本號。

在第 7 行你可以放置上游項目首頁的 URL。

第 9 行是二進位套件的名稱。通常情況下與原始碼套件相同，但不是必須的。

第 10 行描述了可以編譯本二進制包的體繫結構。根據二進制包的類型，這個值常常是下列中的一個：[6]

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

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

[5]這種奇怪的情況是 Debian Policy Manual, Footnotes 55 (http://www.debian.org/doc/debian-policy/footnotes.html#f55) 中詳細描述的一種特性。這不是由於在 `debian/rules` 中使用 **dh** 命令所致的，真正的原因是 **dpkg-buildpackage** 的運行方式。相同的情形也適用於 Ubuntu 的自動編譯系統 (https://bugs.launchpad.net/launchpad-buildd/+bug/238141) 。

[6]確切信息請參見 Debian Policy Manual, 5.6.8 ”Architecture” (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) 。

- `Architecture: any`
  - 一般而言，包含編譯型語言編寫的程序生成的二進制包依賴於具體的體繫結構。
- `Architecture: all`
  - 一般而言，包含文本、圖像、或解釋型語言腳本生成的二進制包獨立於體繫結構。

我們不管第 10 行，鑑於本程序是用 C 語言編寫的。dpkg-gencontrol(1) 命令將根據這個軟件包可以編譯的平臺而爲此處填寫合適的信息。

如果你的套件是平臺獨立的 (例如一個 shell 或 Perl 腳本，或一些文件)，將這項改變爲 `all`，然後繼續閱讀節 4.4 中關於使用 `binary-indep` 指令替代 `binary-arch` 來編譯套件的內容。

第 11 行顯示了 Debian 套件系統中最強大的特性之一。每個套件都可以和其他套件有各種不同的關係。除 `Depends` 外，還有 `Recommends`、`Suggests`、`Pre-Depends`、`Breaks`、`Conflicts`、`Provides` 和 `Replaces`。

軟件包管理工具通常對這些關係採取相同的操作；如果有例外，本教程將會詳細解釋。(參看 dpkg(8)、dselect(8)、apt(8)、aptitude(1) 等。)

這裏有一篇關於軟件包關係的簡述：[7]

- `Depends`
  此套件僅當它依賴的套件均已安裝後纔可以安裝。此處寫明你的程式所必須的套件。
- `Recommends`
  這項中的軟件包不是嚴格意義上必須安裝纔可以保證程序運行。當用戶安裝軟件包時，所有前端工具都會詢問是否要安裝這些推薦的軟件包。**aptitude** 和 **apt-get** 會在安裝你的軟件包的時候自動安裝推薦的軟件包 (用戶可以禁用這個默認行爲)。**dpkg** 則會忽略此項。
- `Suggests`
  此項中的軟件包可以和本程序更好地協同工作，但不是必須的。當用戶安裝程序時，所有的前端程序可能不會詢問是否安裝建議的軟件包。**aptitude** 可以被配置爲安裝軟件時自動安裝建議的軟件包，但這不是默認。**dpkg** 和 **apt-get** 將忽略此項。
- `Pre-Depends`
  此項中的依賴強於 `Depends` 項。套件僅在預依賴的套件已經安裝後纔可以正常安裝並且 正確設定後纔可以正常安裝。在使用此項時應 非常慎重，僅當在 debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) 郵件列表討論後才能使用。記住：根本就不要用這項。:-)
- `Conflicts`
  僅當所有衝突的套件都已經刪除後此套件纔可以安裝。當程式在某些特定套件存在時根本無法運行或存在嚴重問題時使用此項。
- `Breaks`
  此軟件包安裝後列出的軟件包將會受到損壞。通常 `Breaks` 要附帶一個“版本號小於多少”的說明。這樣，軟件包管理工具將會選擇升級被損壞的特定版本的軟件包作爲解決方案。
- `Provides`
  某些型別的軟體包會定義有多個備用的虛擬名稱。你可以在 virtual-package-names-list.txt.gz (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) 檔案中找到完整的列表。如果你的程式提供了某個已存在的虛擬軟體包的功能則使用此項。
- `Replaces`
  當你的程式要替換其他套件的某些檔案，或是完整地替換另一個套件 (與 `Conflicts` 一起使用)。列出的套件中的某些檔案會被你的套件所覆蓋。

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

所有的這些項都使用相同的語法。它們是一個套件列表，套件名稱間使用半形逗號分隔。也可以寫出有多個可選的套件名稱，這些套件使用 | 符號分隔。

這些項內還可以限定與某些軟件包的某個版本區間之間的關係。版本號限定在括號內，這緊隨軟件包名稱之後，並在以下邏輯符號後寫清具體版本：<<、<=、=、>= 和 >>，分別代表嚴格小於、小於或等於、嚴格等於、大於或等於以及嚴格大於。例如，

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

最後一個需要瞭解的特性是 ${shlibs:Depends}, ${perl:Depends}, ${misc:Depends}, 之類。

dh_shlibdeps(1) 會爲二進制包計算共享庫依賴關係。它會爲每個二進制包生成一份 ELF 可執行文件和共享庫列表。這個列表用於替換 ${shlibs:Depends}。

dh_perl(1) 會計算 Perl 依賴。它會爲每個二進制包生成一個叫作 perl 或 perlapi 的依賴列表。這個列表用於替換 ${perl:Depends}。

一些 debhelper 命令可能會使生成的軟件包需要依賴於某些其他的軟件包。所有這些命令將會爲每一個二進制包生成一個列表。這些列表將用於替換 ${misc:Depends} 。

dh_gencontrol(1) 會爲每個二進制包生成 DEBIAN/control 當替換 ${shlibs:Depends}, ${perl:Depends}, ${misc:Depends}, 之類的時候。

說過這些以後，我們可以讓 Depends 項保持現狀，並在其下插入一行 Suggests: file，因爲 gentoo 可以使用 file 軟件包提供的某些特性。

第 9 行是主頁的 URL。我們假設它是 http://www.obsession.se/gentoo/。

第 12 行是簡述。絕大多數人的屏幕是 80 列寬，所以描述不應超過 60 個字符。在這個例子裏我把它寫爲 fully GUI-configurable, two-pane X file manager。

第 13 行是長描述開始的地方。這應當是一段更詳細地描述軟件包的話。每行的第一個格應當留空。描述中不應存在空行，如果必須使用空行，則在行中僅放置一個 . (半角句點) 來近似。同時，長描述後也不應有超過一行的空白。[8]

接下來我們在第 6 和第 7 行之間添加版本控制系統位置 Vcs-* 項。[9] 這裏我們假設 gentoo 軟件包的 VCS 處於 Debian Alioth Git 服務的 git://git.debian.org/git/collab-maint/gentoo.git

到此爲止，我們做好了 control 檔案：

```
 1 Source: gentoo
 2 Section: x11
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
 6 Standards-Version: 3.9.4
 7 Vcs-Git: https://anonscm.debian.org/git/collab-maint/gentoo.git
 8 Vcs-browser: https://anonscm.debian.org/git/collab-maint/gentoo.git
 9 Homepage: http://www.obsession.se/gentoo/
10
11 Package: gentoo
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Suggests: file
15 Description: fully GUI-configurable, two-pane X file manager
16  gentoo is a two-pane file manager for the X Window System. gentoo lets the
17  user do (almost) all of the configuration and customizing from within the
```

[8]這些描述都使用英語。相應的翻譯由 The Debian Description Translation Project - DDTP (http://www.debian.org/intl/l10n/ddtp) (Debian 描述翻譯項目) 項目提供。

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

```
18  program itself. If you still prefer to hand-edit configuration files,
19  they’re fairly easy to work with since they are written in an XML format.
20  .
21  gentoo features a fairly complex and powerful file identification system,
22  coupled to an object-oriented style system, which together give you a lot
23  of control over how files of different types are displayed and acted upon.
24  Additionally, over a hundred pixmap images are available for use in file
25  type descriptions.
26  .
29  gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
30  for its interface.
```

(注：我爲它添加了行號。)

4.2 copyright

這個文件包含了上游軟件的版權以及許可證信息。Debian Policy Manual, 12.5 ”Copyright information” (http://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile) 掌控着它的內容，另外 DEP-5: Machine-parseable `debian/copyright` (http://dep.debian.net/deps/dep5/) 提供了關於其格式的方針。

dh_make 可以給出一個 `copyright` 檔案的模板。在這裏我們使用 `--copyright gpl2` 參數來獲得一個模板寫明 `gentoo` 套件是發佈於 GPL-2 許可證下。

你必須填寫上空缺的信息，如你從何處獲得此軟件，實際的版權聲明和它們的許可證。對於常見的自由軟件許可證，如 GNU GPL-1、GNU GPL-2、GNU GPL-3、LGPL-2、LGPL-2.1、LGPL-3、GNU FDL-1.2、GNU FDL-1.3、Apache-2.0 或 Artistic 許可證，你可以直接將其指向所有 Debian 系統都有的 `/usr/share/common-licenses/` 目錄下的文件。否則，許可證則必須包含完整的許可證文本。

簡言之，`gentoo` 的 `copyright` 文件如下所示：

```
 1 Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
 2 Upstream-Name: gentoo
 3 Upstream-Contact: Emil Brink <emil@obsession.se>
 4 Source: http://sourceforge.net/projects/gentoo/files/
 5
 6 Files: *
 7 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
 8 License: GPL-2+
 9
10 Files: icons/*
11 Copyright: 1998 Johan Hanson <johan@tiq.com>
12 License: GPL-2+
13
14 Files: debian/*
15 Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
16 License: GPL-2+
17
18 License: GPL-2+
19  This program is free software; you can redistribute it and/or modify
20  it under the terms of the GNU General Public License as published by
21  the Free Software Foundation; either version 2 of the License, or
22  (at your option) any later version.
23  .
24  This program is distributed in the hope that it will be useful,
25  but WITHOUT ANY WARRANTY; without even the implied warranty of
26  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
27  GNU General Public License for more details.
28  .
29  You should have received a copy of the GNU General Public License along
30  with this program; if not, write to the Free Software Foundation, Inc.,
```

```
31  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
32  .
33  On Debian systems, the full text of the GNU General Public
34  License version 2 can be found in the file
35  '/usr/share/common-licenses/GPL-2'.
```

(注：我爲它添加了行號。)

另外還可以參看 ftpmasters 發送到 debian-devel-announce 的 HOWTO：announce: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

4.3 changelog

這是一個必須的文件，它的特殊格式在 Debian Policy Manual, 4.4 "debian/changelog" (http://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog) 中有詳細的描述。這種格式被 **dpkg** 和其他程序用以解析版本號信息、適用的發行版和緊急程度。

對於你而言，詳細描述你所做出的更改也是很好且很重要的。它將幫助下載你的套件的人瞭解這個套件中是否有他們需要知道的事情。它會被作爲 `/usr/share/doc/gentoo/changelog.Debian.gz` 保存在二進位套件中。

dh_make 創建了一個默認的文件，這是它的容貌：

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

(注：我爲它添加了行號。)

第 1 行是軟體包名、版本號、發行版和緊急程度。軟體包名必須與實際的原始碼包名相同，發行版應該是 `unstable`。除非有特殊原因，緊急程度預設設定為 medium (中等)。

第 3-5 行是一個很長的條目，記錄了你在這個 Debian 修訂版本中做出的修改 (非上游修改——上游修改由上游作者建立並由另外一個檔案維護，它們應被安裝為 `/usr/share/doc/gentoo/changelog.gz`)。假設你的 ITP (Intent To Package，計劃打包) 的 Bug 號為 `12345`。新行必須插入在上一個以星號 `*` 開頭的行的正下方。你可以使用 dch(1) 完成這個工作，也可以使用普通的文字編輯器手工完成，只要你遵循 dch(1) 所使用的格式。

為了阻止軟體包在打包完成之前被意外上傳，將發行版值改成一個不可用的 `UNRELEASED` 將是一個很好的選擇。

最後它會成爲以下的樣子：

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

(注：我爲它添加了行號。)

如果你已經對自己所作出的改動感到滿意，而且它們都被記錄在了 `changelog` 中，那麼你就可以將發行版值由 `UNRELEASED` 修改至目標發行版值 `unstable` (甚至 `experimental`)。[10]

你可以在關於更新的章 8 中瞭解更多關於 `changelog` 的內容。

[10]如果你用 `dch -r` 命令來使它成爲最後一筆更改，請確保用編輯器顯式地保存 `changelog` 文件。

4.4 rules

現在我們需要看看 dpkg-buildpackage(1) 用於實際建立軟體包的 rules 檔案。這個檔案事實上是另一個 Makefile，但不同於上游原始碼中的那個。和 debian 目錄中的其他檔案不同，這個檔案被標記為可執行。

4.4.1 rules 文件中的 Target

每一個 rules 文件，就像其他的 Makefile 一樣，包含着若干 rules，其中每一個都定義了一個 target 以及其具體操作。[11] 一個新的 rule 以自己的 target 聲明 (置於第一列) 來起頭。後續的行都以 TAB 字符 (ASCII 9) 來開頭，以指示 target 的具體行爲。空行和以井號 # 開頭的行會被當作註釋而被忽略。[12]

當你想要執行一個 rule 的時候，就將 target (目標) 名稱作為命令列引數來呼叫。比如說，debian/rules *build* 以及 fakeroot make -f debian/rules *binary* 會分別執行 *build* 和 *binary* 兩個 target。

以下是對各 target 的簡單解釋：

- clean target：清理所有編譯的、生成的或編譯樹中無用的文件。(必須)
- build target：在編譯樹中將代碼編譯爲程序並生成格式化的文檔。(必須)
- build-arch target：在編譯樹中將代碼編譯爲依賴於體繫結構的程序。(必須)
- build-indep target：在編譯樹中將代碼編譯爲獨立於平臺的格式化文檔。(必須)
- install target：把文件安裝到 debian 目錄內爲各個二進制包構建的文件樹。如果有定義，那麼 binary* target 則會依賴於此 target。(可選)
- binary target：創建所有二進制包 (是 binary-arch 和 binary-indep 的合併)。(必須)[13]
- binary-arch target：在父目錄中創建平臺依賴 (Architecture: any) 的二進制包。(必須)[14]
- binary-indep target：在父目錄中創建平臺獨立 (Architecture: all) 的二進制包。(必須)[15]
- get-orig-source target：從上游站點獲得最新的原始源代碼包。(可選)

可能你現在感到有些迷惑，在接下來講解 **dh_make** 給出的默認的 rules 文件時事情會變得簡單。

4.4.2 默認的 rules 檔案

新版本的 **dh_make** 會生成一個使用 **dh** 命令的非常簡單但非常強大的默認的 rules 檔案：

```
 1 #!/usr/bin/make -f
 2 # See debhelper(7) (uncomment to enable)
 3 # output every command that modifies files on the build system.
 4 #DH_VERBOSE = 1
 5
 6 # see FEATURE AREAS in dpkg-buildflags(1)
 7 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
 8
 9 # see ENVIRONMENT in dpkg-buildflags(1)
10 # package maintainers to append CFLAGS
```

[11] 你可以通過該資源來學習編寫 Makefile：Debian Reference, 12.2. ”Make” (http://www.debian.org/doc/manuals/debian-reference/ch12#_make)。完整文件在 http://www.gnu.org/software/make/manual/html_node/index.html 或者 make-doc 軟體包 (該包位於 non-free 部分)。

[12] Debian Policy Manual, 4.9 ”Main building script: debian/rules” (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules) 針對細節進行瞭解釋。

[13] 此 target 被 dpkg-buildpackage 用於節 6.1 描述的過程中。

[14] 此 target 被 dpkg-buildpackage -B 用於節 6.2 描述的過程中。

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

```
11 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
12 # package maintainers to append LDFLAGS
13 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
14
15
16 %:
17         dh $@
```

(注：我添加了行號並刪去了一些註釋。實際的 `rules` 文件裏開頭的空格是 TAB 填充的。)

可能在 shell 或 Perl 腳本中你已經對第一行的形式很熟悉了，它告訴作業系統這個檔案應使用 `/usr/bin/make` 處理。

可以取消第 4 行的註釋，以設置 `DH_VERBOSE` 變量爲 1，於是 **dh** 命令就會輸出它將要使用的 **dh_*** 命令。你也可以在此添加一行 `export DH_OPTIONS=-v`，於是 **dh_*** 命令同樣也會輸出它正在調用的命令。這能幫助你理解在這個簡單的 `rules` 文件背後發生了什麼，以及幫助你進行調試。新的 **dh** 被設計來作爲 `debhelper` 工具的核心部分，並不向你隱藏任何東西。

第 16 和 17 行使用了 pattern rule，以此隱式地完成所有工作。其中的百分號意味著“任何 targets”，它會以 target 名稱作引數呼叫單個程式 **dh**。[16] **dh** 命令是一個包裝指令碼，它會根據引數執行妥當的 **dh_*** 程式序列。[17]

- `debian/rules clean` 運行了 `dh clean`，接下來實際執行的命令爲：

```
dh_testdir
dh_auto_clean
dh_clean
```

- `debian/rules build` 運行了 `dh build`，其實際執行的命令為：

```
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
```

- `fakeroot debian/rules binary` 執行了 `fakeroot dh binary`，其實際執行的命令為[18]：

```
dh_testroot
dh_prep
dh_installdirs
dh_auto_install
dh_install
dh_installdocs
dh_installchangelogs
dh_installexamples
dh_installman
dh_installcatalogs
dh_installcron
dh_installdebconf
dh_installemacsen
dh_installifupdown
dh_installinfo
dh_installinit
dh_installmenu
dh_installmime
```

[16]此處使用了新版本 `debhelper` v7+ 的特性。它的設計理念在 Not Your Grandpa's Debhelper (http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf) 中進行了闡明，這在 DebConf9 中被 `debhelper` 上游進行了演示。在 `lenny` 下，**dh_make** 會建立一個更為複雜的 `rules` 檔案，伴有許多顯式的 rule 和許多為每個 rule 所用的 **dh_*** 指令碼，其中有大部分在現在已經是不必要的了 (這也顯示了軟體包的年齡)。新一代的 **dh** 命令更為簡潔，並能將我們從” 手工的重複工作” 中解放出來。當然，你仍然擁有完全的力量來定製整個過程，只要使用 `override_dh_*` target。參見節 4.4.3。它僅僅基建於 `debhelper` 軟體包，而且不會像 `cdbs` 軟體包所傾向的那樣混淆軟體包構建過程。

[17]你可以檢驗每一個已有的 *`target`* 所調用的實際 **dh_*** 程序序列，而並不需要真的通過 `dh --no-act` *`target`* 或 `debian/rules --'--no-act` *`target`*`'` 來執行以查看。

[18]下邊的例子假定你的 `debian/compat` 有一個值大於或等於 9，以此避免自動調用任何 python 支持命令。

```
dh_installmodules
dh_installlogcheck
dh_installlogrotate
dh_installpam
dh_installppp
dh_installudev
dh_installwm
dh_installxfonts
dh_bugfiles
dh_lintian
dh_gconf
dh_icons
dh_perl
dh_usrlocal
dh_link
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
```

- `fakeroot debian/rules binary-arch` 執行了 `fakeroot dh binary-arch`,其效果等同於 `fakeroot dh binary` 並附加 `-a` 引數於每個命令後。
- `fakeroot debian/rules binary-indep` 執行了 `fakeroot dh binary-indep`,這會執行幾乎和 `fakeroot dh binary` 一樣的命令，但 **dh_strip**、**dh_makeshlibs** 和 **dh_shlibdeps** 除外，其他命令則均附加 `-i` 選項。

dh_* 命令的功能依其名稱不言而喻。[19] 不過其中有一些值得在這裏進行簡要解釋，假定有一個基於 `Makefile` 的典型構建環境：[20]

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

```
make distclean
```

- **dh_auto_configure** 在 `./configure` 存在時通常執行以下命令 (省略了部分參數以方便此處閱讀)。

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

- **dh_auto_build** 通常使用以下命令執行 `Makefile` 中的第一個 target。

```
make
```

- **dh_auto_install** 通常在 `Makefile` 存在且有 `test` target 時執行以下命令。[22]

```
make test
```

- **dh_auto_install** 通常在 `Makefile` 存在且有 `install` target 時執行以下命令 (進行了換行以便閱讀)。

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

[19] 關於所有 **dh_*** 腳本具體行爲，以及它們有什麼選項的完整信息，請閱讀它們各自的 man 手冊頁以及 `debhelper` 的文檔。
[20] 這些構建系統同樣支援其他的構建環境，比如 `setup.py`，這可以通過在軟體包原始碼目錄中執行 `dh_auto_build --list` 來列出。
[21] 它實際上在 `Makefile` 中搜尋第一個可用的 target，除了 `distclean`, `realclean`, 或 `clean`，接下來再執行它。

[22] 實際上它在 `Makefile` 中搜索第一個可用的 target，除了 `test` 或 `check`，然後執行它。

所有需要 **fakeroot** 命令的都包含了 **dh_testroot**。如果你沒有使用 fakeroot，那將會報錯並退出。

關於 **dh_make** 生成的 `rules` 文件，你應該知道的最重要的事是，它僅僅是一個建議。它對多數簡單的軟件包有效，但對於更複雜的則要大膽對其進行定製以滿足需要。

儘管 `install` target 不是必須的，但也被支持。`fakeroot dh install` 的操作就像 `fakeroot dh binary` 一樣，但停止於 **dh_fixperms**。

4.4.3 定製 `rules` 檔案

有很多方法來定製使用新的 **dh** 命令創建的 `rules` 檔案。

`dh $@` 命令可以按以下方式定製：[23]

- 爲 **dh_python2** 命令添加支持。(對於 Python 的最佳選擇。)[24]
  - 在 `Build-Depends` 中添加 `python` 軟件包。
  - 使用 `dh $@ --with python2`.
  - 這會使用 `python` 框架處理 Python 模塊。
- 添加 **dh_pusuppor** 命令的支持。(已廢棄)
  - 在 `Build-Depends` 中添加 `python-support` 軟件包。
  - 使用 `dh $@ --with pysupport`
  - 這會使用 `python-support` 框架處理 Python 模塊。
- 添加 **dh_pycentral** 命令支持。(已廢棄)
  - 在 `Build-Depends` 中添加 `python-central` 軟件包。
  - 使用 `dh $@ --with python-central`
  - 這樣會同時停用 **dh_pysupport** 命令。
  - 這會使用 `python-central` 框架處理 Python 模塊。
- 添加 **dh_installtex** 命令支持。
  - 在 `Build-Depends` 中添加 `tex-common` 軟件包。
  - 使用 `dh $@ --with tex`
  - 這樣會註冊 Type 1 字體、斷句樣式及其他 TeX 格式。
- 添加 **dh_quilt_patch** 和 **dh_quilt_unpatch** 命令支持。
  - 在 `Build-Depends` 中添加 `quilt` 軟件包。
  - 使用 `dh $@ --with quilt`
  - 這會在你使用 `1.0` 格式的源代碼包時自動應用或解除 `debian/patches` 目錄中的補丁。
  - 如果你使用新的 `3.0 (quilt)` 源代碼包格式則不需要這些。
- 爲 **dh_dkms** 命令添加支持。
  - 在 `Build-Depends` 中添加 `dkms` 軟件包。
  - 使用 `dh $@ --with dkms`
  - 這能使內核模塊軟件包正確使用 DKMS。

[23] 如果一個軟件包安裝了 `/usr/share/perl5/Debian/Debhelper/Sequence/`*`custom_name`*`.pm` 文件，你應當使用 `dh $@ --with `*`custom-name`* 命令激活其功能。

[24] 在 **dh_python2** 和 **dh_pysupport** 以及 **dh_pycentral** 命令之間更推薦使用前者。不要使用 **dh_python** 命令。

- 添加 **dh_autotools-dev_updateconfig** 和 **dh_autotools-dev_restoreconfig** 命令支持。
  - 在 `Build-Depends` 中添加 `autotools-dev` 軟件包。
  - 使用 `dh $@ --with autotools-dev`
  - 這會自動更新或還原 `config.sub` 和 `config.guess` 檔案。
- 添加 **dh_autoreconf** 和 **dh_autoreconf_clean** 命令支持。
  - 在 `Build-Depends` 中添加 `dh-autoreconf` 軟件包。
  - 使用 `dh $@ --with autoreconf`
  - 這樣會在編譯時更新 GNU 編譯系統檔案並在編譯後對其進行恢復。
- 添加 **dh_girepository** 命令支持。
  - 在 `Build-Depends` 中添加 `gobject-introspection` 軟件包。
  - 使用 `dh $@ --with quilt`
  - 這會爲帶有 GObject 內省數據的軟件包計算依賴，並生成 `${gir:Depends}` 這一替換變量。
- 添加 **bash** 補全特性支持。
  - 在 `Build-Depends` 中添加 `bash-completion` 軟件包。
  - 使用 `dh $@ --with bash-completion`
  - 這會使用 `debian/`*`package`*`.bash-completion` 中的配置文件來安裝 **bash** 補全。

很多由新的 **dh** 命令觸發的 **dh_*** 都可以通過修改 `debian` 目錄中的設定檔案來對其行爲進行定製。參考章 5 和每個命令的 man 手冊頁。

某些由新的 **dh** 命令所觸發的 **dh_*** 命令可能需要額外的參數，或需要附加執行或者跳過執行。對於這類情況，你可以在 `rules` 文件中創建一個 `override_dh_`*`foo`* target，並在其中定義一個 `override_dh_`*`foo`* 來使其完成你想要 **dh_*foo*** 命令作出的改變。它的作用簡單說就是 *run me instead (*把運行的命令換成我*)*。[25]

請注意 **dh_auto_*** 命令爲了照顧所有的邊緣情況，它實際所做的比上述 (過度) 簡化的步驟中介紹的內容更多。除了 `override_dh_auto_clean` 外把上面的簡化命令寫成 `override_dh_*` 中是不明智的，這樣會使得 `debhelper` 的許多智能特性無法體現。

所以，比如，最近的 `gentoo` 軟件包使用了 Autotools，如果你希望把系統配置配置文件安裝到 `/etc/gentoo` 而非通常的 `/etc` 目錄，你可以凌駕 **dh_auto_configure** 默認的使用的 `--sysconfig=/etc` 參數，改爲向 **./configure** 命令傳遞以下參數：

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

在 `--` 其後給出的引數會被追加到被自動執行的程式預設引數後，以此凌駕它們並修改其預設行為。使用 **dh_auto_configure** 命令要比直接呼叫 **./configure** 命令好很多，因為它只修改 `--sysconfig` 引數內容，同時保留其他任何對 **./configure** 命令良性的引數。

如果 `gentoo` 的 `Makefile` 需要指定 `build` 作爲其編譯用的 target[26]，你可以創建一個 `override_dh_auto_build` target 來啓用它。

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

這保證了 `$(MAKE)` 會使用 **dh_auto_build** 傳遞的所有默認參數並編譯處理 `build` 這個 target。

如果 `gentoo` 的 `Makefile` 需要指定 `packageclean` target 來爲 Debian 軟件包作清理，而非 `distclean` 或 `clean` target，那你就可以創建一個 `override_dh_auto_clean` target 來啓用它。

[25] 在 `lenny` 下，如果你希望更改某個 **dh_*** 腳本的行爲，你需要在 `rules` 中找到相應的行然後進行調整。
[26] 沒有參數的 **dh_auto_build** 命令將執行 `Makefile` 中的第一個 target。

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

如果 gentoo 的 Makefile 包含了一個 test target 但你不想在 Debian 軟件包構建過程中運行它，可以使用空的 override_dh_auto_test target 來跳過它。

```
override_dh_auto_test:
```

如果 gentoo 有某個不常見的上游 changelog 檔案名爲 FIXES，默認情況下 **dh_installchangelogs** 不會安裝它。**dh_installchangelogs** 命令需要將 FIXES 作爲它的參數來安裝它。[27]

```
override_dh_installchangelogs:
        dh_installchangelogs FIXES
```

如果你使用新的 **dh** 命令時，還使用節 4.4.1 中除 get-orig-source 的顯式 target，會使得其效果難以預料。如果可能的話請儘量避免使用獨立的或預設的 target，如果必須修改默認設置則酌情使用 override_dh_*。

[27] debian/changelog 和 debian/NEWS 總是會被自動安裝。程序會將文件名轉爲小寫並搜索以下文件名來檢測上游 changelog：changelog、changes、changelog.txt 和changes.txt。

Chapter 5

debian 目錄下的其他檔案

要控制 debhelper 在構建軟件包過程中的大部分行爲，可以在 debian 目錄中放置可選的配置文件。本章將會對這些文件和它們的格式進行概述。請閱讀 Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) 和 Debian Developer's Reference (http://www.debian.org/doc/devel-manuals#devref) 來瞭解更多關於打包方針的內容。

dh_make 命令會在 debian 目錄中創建一些配置文件模板，它們的文件名多帶有 .ex 後綴。其中的一些可能以二進制包名作爲前綴，如 *package*。現在我們來對它們進行一個大致的瞭解。[1]

還有一些 debhelper 的配置文件模板可能未被 **dh_make** 命令創建。在這種情況下如果你需要它們，則要使用文本編輯器手工創建。

如果你希望或需要激活它們中的任意一個或多個，請按照下面的方法做：

- 如果模板文件帶有 .ex 或 .EX 後綴，則重命名它去掉後綴；
- 把配置文件的名稱改爲實際的二進制軟件包名，例如 *package*;
- 修改模板文件來滿足你的需要;
- 刪除不需要的模板文件;
- 如果需要，修改 control 文件 (參看節 4.1)。
- 如果需要，修改 rules 檔案 (參考節 4.4)。

任何不帶有 *package* 前綴的 debhelper 配置文件，比如應用到第一個二進制包的 install。當此處有多個二進制包時，可以通過在文件名中前置它們各自的名稱來指定它們各自的配置文件，比如*package-1*.install, *package-2*.install, 之類。

5.1 README.Debian

所有關於你的 Debian 版本與原始版本間的額外訊息或差別都應在這裏記錄。

dh_make 創建了一個默認的文件，以下是它的樣子：

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

如果你沒有需要寫在這裏的東西，則刪除這個檔案。參考 dh_installdocs(1)

[1] 在本章節中，只要不產生歧義，所有提及的 debian 目錄下的文件均會省去 debian/ 前綴以求簡潔方便。

5.2 compat

compat 檔案定義了 debhelper 的相容級別。目前你應當使用如下方法將其設定為 debhelper V10:

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

在特定場景下，你可以在需要相容舊版本系統時使用相容等級 9。然而，我們不建議你使用任何低於 9 的相容等級，在新建軟體包時也應避免使用這些低的等級。

5.3 conffiles

關於軟件有件很惱人的事，就是當你付出了很多時間和精力來自定義一個程序，但是升級後所有的修改都被覆蓋掉了。Debian 通過將配置文件單獨標記來解決這個問題，[2] 當軟件包升級的時候，你將會被詢問是否要保留你的舊配置文件。

dh_installdeb(1) *automatically* 會把 /etc 目錄下的任何文件都標記爲 conffiles，所以如果你的程序在那只有 conffiles 的話就不需要再在這個文件中指定它們。對於大多數軟件包類型，唯一合理的 conffiles 文件存放位置自始至終應當在 /etc 目錄下，正因如此，該文件也沒有存在的必要。

如果你的程序使用配置文件，但程序會自動對配置文件進行改寫，則最好別將其標記爲 conffiles ，因爲 **dpkg** 總是會要求用戶校驗變更。

如果你正在打包的程序需要所有用戶都爲自己修改 /etc 目錄中的配置文件，那麼有兩種常見的方法讓 **dpkg** 不將其記錄爲 conffiles，以使其沉默。

- 在 /etc 目錄中創建指向 /var 中維護者腳本 (maintainer scripts) 生成的文件的符號鏈接。
- 用維護者腳本 (maintainer scripts) 在 /etc 目錄中生成一個文件。

更多關於維護者腳本 (maintainer scripts) 的信息，參看節 5.18

5.4 *package*.cron.*

如果你的軟件包需要有計劃運行的操作以保證正常工作，可以使用這個文件達成。你既可以設置常規的任務使其在每小時、每天、每星期、每月或其他情況或你希望的時間下運行。相應的文件名是:

- *package*.cron.hourly - 安裝至 /etc/cron.hourly/*package* ； 每小時運行一次。
- *package*.cron.daily - 安裝至 /etc/cron.daily/*package* ； 每天運行一次。
- *package*.cron.weekly - 安裝至 /etc/cron.weekly/*package* ； 每週運行一次。
- *package*.cron.monthly - 安裝至 /etc/cron.monthly/*package* : 每月運行一次。
- *package*.cron.d - 安裝至 /etc/cron.d/*package*: 對於其他的任何時間。

這些文件均爲 shell 腳本。唯一不同的是 *package*.cron.d，它的格式需要參照 crontab(5)

有必要顯式地用 cron.* 文件來設置日誌輪轉；關於這個問題，請參見 dh_installlogrotate(1) 和 logrotate(8).

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

5.5 dirs

這個文件指定了我們需要，但在正常安裝過程 (dh_auto_install 觸發的 make install DESTDIR=...) 裏沒有被安裝的目錄。通常這是由於 Makefile 中存在問題。

install 文件中列出的文件不需要爲其提前創建目錄，參看節 5.11。

最好是先嘗試安裝，如果遇到了問題再使用這個文件。在 dirs 中列出的目錄名中不應有前導的 / (斜槓) 符號。

5.6 *package*.doc-base

如果你的軟件包在 man 手冊頁和 info 信息文檔外還有其他文檔，你應該使用 doc-base 文件註冊它們，這樣用戶可以使用例如 dhelp(1)、dwww(1) 或 doccentral(1) 的工具找到它們。

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

以下是 gentoo 的 doc-base 文件 gentoo.doc-base 的樣子：

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

關於文件格式的更多信息，參看 install-docs(8) 以及 Debian doc-base 手冊頁，在 /usr/share/doc/doc-base/doc-base.html/index.html 有一份本地拷貝，這是由 doc-base 軟件包提供的。

關於安裝附加文件的更多訊息，查看節 3.3。

5.7 docs

這個檔案制定了我們使用 dh_installdocs(1) 安裝到臨時目錄的檔案名。

默認情況下它會加入程式碼目錄頂層的所有名爲 BUGS、README*、TODO 等的檔案。

對於 gentoo，這裏還加入了一些其他文件：

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

5.8 emacsen-*

如果你的套件提供可以在安裝時編譯爲字節碼的 Emacs 檔案，你可以使用這些檔案設置。

它們會被 dh_installemacsen(1) 安裝到臨時目錄。

如果你不需要這些，就刪除它們。

5.9 *package*.examples

dh_installexamples(1) 會將列出的檔案和目錄作爲範例檔案安裝。

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

如果你的軟件包需要在系統啓動時運行一個守護進程，那麼你顯然沒有按照我最初的建議做事，不是嗎？:-)

package.init 文件會被安裝爲 /etc/init.d/*package* 腳本，該腳本可用於啓動和停止守護進程。**dh_make** 創建的 init.d.ex 是一個很好的骨架模板。你可能要對其改名並做很多修改，同時還要提供 Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb) (LSB，Linux 標準規範) 的兼容頭文件。用 dh_installinit(1) 可以將它安裝到臨時目錄中。

package.default 文件會被安裝至 /etc/default/*package*。這個文件設置被 init 腳本使用的默認設置。*package*.default 文件最經常用于禁用一個正在運行的守護程序，或者設置一些默認標記或者超時。如果你的 init 腳本中有某種可配置的特性，你可以在 *package*.default 文件中設置它們，而不是 init 腳本本身。

如果你的上遊程序中包含了給 init 用的腳本文件，那用不用它都可以。如果不使用，則創建相應的 *package*.init 文件；然而如果上游的 init 腳本很好且被安裝到正確的位置，你仍然需要設置 rc* 符號鏈接。你需要按照以下的方法在 rules 文件中凌駕 **dh_installinit**：

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

如果你不需要這些，就刪除它們。

5.11 install

如果你的軟體包需要那些標準的 make install 沒有安裝的檔案，你可以把檔名和目標路徑寫入 install 檔案，它們將被 dh_install(1) 安裝。[3] 你需要首先檢查要安裝的檔案是否有更有針對性的特定工具會對其進行安裝。例如，文件應寫在 docs 檔案中安裝，而不是這一個。

這個 install 文件每行安裝一份文件，格式上先是相對於編譯目錄的文件路徑，然後是一個空格，接下來是相對於安裝目錄的目標目錄。例如，假設某個二進制文件 src/*bar* 沒有被默認安裝，則應讓 install 呈現成這樣：

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

這意味着安裝這個軟件包時，將有一個二進制文件 /usr/bin/*bar*.

當然，在相對路徑保持不變的情況下 install 文件也可以只包含相對的源路徑而不帶目標位置。這樣的格式通常用於使用 *package-1*.install、*package-2*.install 等將大軟件包分割爲多個二進制包的情況。

如果 **dh_install** 命令沒有在當前目錄 (或者你可能使用 --sourcedir 參數指定的位置) 找到文件，它會回滾至使用 debian/tmp 目錄。

5.12 *package*.info

如果你的軟件包有 info 信息頁，應該使用 dh_installinfo(1) 安裝，要安裝的文件列於 *package*.info 文件中。

5.13 *package*.links

如果你作爲包維護者需要在包中創建附加的符號鏈接，你應該使用 dh_link(1) 來安裝，要安裝的文件完整路徑列於 *package*.info 文件中。

[3] 這取代了已經廢棄的 dh_movefiles(1) 命令，它本是用 files 檔案所設定的。

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

如果 lintian 根據 Debian Policy 的某些規則允許例外從而報告了錯誤診斷,你可以使用 *package*.lintian-overrides 或 source/lintian-overrides 使其不再警告。請認真閱讀 Lintian 使用者手冊 (https://lintian.debian.org/manual/index.html) 並避免濫用。

package.lintian-overrides 是對於名爲 *package* 的二進制包的有效配置，會由 **dh_lintian** 命令安裝到usr/share/lintian/overrides/*package*。

source/lintian-overrides 是針對原始碼套件的，不會安裝。

5.15 manpage.*

你的程序應該有 man 手冊頁，如果它們沒有自帶則需要由你來創建。**dh_make** 命令創建了幾個 man 手冊頁的模板。爲每一個缺手冊的命令拷貝一份模板，並妥善地編寫，並且要刪除未使用的模板。

5.15.1 manpage.1.ex

man 手冊頁通常是使用 nroff(1) 的格式編寫的。manpage.1.ex 模板也是使用 **nroff** 格式的。參考 man(7) 手冊頁來簡要瞭解如何編輯這個檔案。

最終的 man 手冊頁文件的名稱，應當爲其對應的程序名稱。所以我們將 manpage 重命名爲 gentoo。同時，它的文件名當然要以 .1 作爲第一個後綴，這表示本手冊頁是爲一個用戶命令而撰寫的。再者，請校驗本 man 手冊處在正確的節 (section)。這個簡短的列表列舉了 man 手冊頁的章節：

Section(部分)	**Description(描述)**	**Notes(提示)**
1	User commands(用戶命令)	Executable commands or scripts(可執行命令或腳本)
2	System calls(系統調用)	Functions provided by the kernel(由內核提供的功能)
3	Library calls(庫調用)	Functions within system libraries(系統庫中的功能)
4	Special files(特殊文件)	經常在 /dev 目錄中出沒
5	File formats(文件格式)	例如，/etc/passwd 的格式
6	Games(遊戲)	Games or other frivolous programs(遊戲或無聊的程序)
7	Macro packages(宏包)	比如 **man** 宏
8	System administrarion(系統管理)	Programs typically only run by root(典型的 root 專用程序)
9	Kernel routines(內核慣例)	Non-standard calls and internals(非標準調用及內部構建)

所以 gentoo 的 man 手冊頁應叫 gentoo.1。如果原始代碼中沒有 gentoo.1 man 手冊頁，你需要重命名 manpage.1.ex 模板爲 gentoo.1 並按照示例和上游文檔編輯它。

你也可以使用 **help2man** 命令來藉助每個程序的 --help 和 --version 參數的輸出來生成 man 手冊頁。[4]

5.15.2 manpage.sgml.ex

如果你希望使用 SGML 而非 **nroff** 格式編寫 man 手冊頁，可以使用 manpage.sgml.ex 模板。如果你要這樣，需要進行以下步驟：

- 重命名檔案爲類似 gentoo.sgml 的名稱。
- 安裝 docbook-to-man 套件。
- 在 control 文件中將 docbook-to-man 添加到 Build-Depends 行。

[4]注意，**help2man** 的佔位符性質 man 手冊頁會聲稱在 info 系統中有着更爲詳盡的細節。如果命令缺少 **info** 頁，那你應該手工編輯由 **help2man** 命令創建的 man 手冊頁。

- 在 rules 文件裏添加 override_dh_auto_build target:

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

5.15.3 manpage.xml.ex

如果你希望使用 XML 而非 SGML，可以使用 manpage.xml.ex 模板。如果你要這樣，需要進行以下步驟:

- 重命名源檔案爲類似 gentoo.1.xml 的名稱。
- 安裝 docbook-xsl 套件和一個 XSLT 處理器，例如 xsltproc (推薦)。
- 在 control 文件中將 docbook-xsl, docbook-xml, 以及 xsltproc 軟件包添加到 Build-Depends 行。
- 在 rules 文件裏添加 override_dh_auto_build target:

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

5.16 *package*.manpages

如果你的軟件包有 man 手冊頁，你應該將它們列在 *package*.manpages 文件中以便 dh_installman(1) 進行安裝。

要將 doc/gentoo.1 安裝爲 gentoo 的 man 手冊頁，創建一個 gentoo.manpages，內容如下:

```
docs/gentoo.1
```

5.17 NEWS

dh_installchangelogs(1) 命令會安裝這個檔案。

5.18 {pre,post}{inst,rm}

postinst、preinst、postrm 和 prerm 文件 [5] 被稱爲 *maintainer scripts*。它們是放置於軟件包控制區域，並由 **dpkg** 在軟件包安裝、升級或卸載時執行的腳本。

作爲一個新維護人員，你應當避免手工編輯 maintainer scripts ，因爲它們常存在各種問題。更多信息請閱讀 Debian Policy Manual, 6 ”Package maintainer scripts and installation procedure” (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) 並查看 **dh_make** 給出的示例文件。

如果你不聽我的勸告，自己爲一個軟件包創建並定製了 maintainer scripts，你必須保證不僅測試 **install** 和 **upgrade**，還應測試 **remove** 和 **purge**。

[5] 儘管這裏使用 **bash** 表達式速記法 {pre,post}{inst,rm} 來指示這些文件名，但你應該使用純 POSIX 語法來編寫 maintainer scripts，這是爲了兼容 **dash** 作爲系統 shell 的情況。

升級到新版本應當是靜默且非侵入式的 (已有用戶應當只在發現舊的 Bug 被修復或有新特性時注意到升級的變化)。

當更新必須以非靜默模式進行時 (例如分散在多個主目錄中的配置文件都要改爲完全不同的結構時)，作爲最後的手段，你應該考慮將軟件包設置到安全的回退狀態 (例如禁用服務) 並按照 Debian Policy 提供相應的文檔 (`README.Debian` 和 `NEWS.Debian`)。不要在升級時使用 maintainer scripts 觸發 **debconf** 來打擾用戶。

`ucf` 軟件包提供了 類似 *conffile* 對可能不標記爲 *conffiles* 的文件的保留機制，比如由 maintainer scripts 來管理的配置文件。這可以把最大程度減少相關的問題。

這些 maintainer scripts 是 Debian 的增強特性，它們解釋了 人們爲什麼選擇 **Debian**。你必須非常小心，保證人們不因此產生煩惱。

5.19 *package*`.examples`

對於新維護者而言，打包一個庫非常不易，因此不建議嘗試。這樣說吧，如果你的軟體包有庫，那你應該處理好 `debian/`*package*`.symbols` 檔案。參見節 A.2.

5.20 `TODO`

dh_installdocs(1) 命令會安裝這個檔案。

5.21 `watch`

`watch` 檔案的格式被詳述於 uscan(1) man 手冊頁中。`watch` 檔案配置了 **uscan** 程式 (它在 `devscripts` 中) 以便監視你獲得原始原始碼的網站。它還被用於 Debian 軟體包跟蹤系統（PTS） (https://tracker.debian.org/) 服務。

它的內容如下：

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

通常，在按照這個 `watch` 文件執行時，URL `http://sf.net/gentoo` 會被下載，程序會在其中搜索 `<a href=...>`。最後一個斜槓 `/` 後的基本名稱會按照 Perl 正則表達式匹配 (參看 perlre(1)) `gentoo-(.+)\.tar\.gz`。在所有匹配的文件裏，將會下載帶有最大版本號的，此後由 **uupdate** 程序創建更新的源代碼樹。

儘管上述內容對於所有站點都適用，但 SourceForge 下載服務 (http://sf.net (http://sf.net)) 仍是一個例外。當 `watch` 中包含匹配 Perl 正則表示式 `^http://sf\.net/` 的 URL 時，**uscan** 程式會將其替換為 `http://qa.debian.org/watch/sf.php` 然後應用此規則。http://qa.debian.org/ (http://qa.debian.org/) 的 URL 重定向服務被設計用於提供一個穩定的重定向服務以滿足 `watch` 檔案中的 `http://sf.net/`*`project/tar-name`*`-(.+)\.tar\.gz` 形式，這樣解決了 SourceForge 經常性的 URL 變更導致的問題。

如果上游提供 tarball 的加密簽章，那麼推薦校驗其真實性，可以用 `pgpsigurlmangle` 選項來實現，這一點在 uscan(1) 中有描述。

5.22 `source/format`

在 `debian/source/format` 中只包含一行，寫明瞭此原始碼套件的格式 (查看 dpkg-source(1) 獲得完整列表)。在 `squeeze` 後，它應該是以下二者之一：

- `3.0 (native)` - Debian 本土軟件或者
- `3.0 (quilt)` - 其他所有軟體

全新的 `3.0 (quilt)` 原始碼格式將所有修改使用 **quilt** 補丁系列記錄到 `debian/patches`。這些修改會在解壓原始碼套件時自動應用。[6] Debian 修改保存於 `debian.tar.gz` 壓縮檔，其中包含了整個 `debian` 目錄。這個新格式支持直接添加例如 PNG 圖示等的二進位檔案。[7]

dpkg-source 解壓 `3.0 (quilt)` 格式的源碼包時會自動應用所有列於 `debian/patches/series` 的補丁。你可以使用 `--skip-patches` 選項避免在解壓後自動應用補丁。

5.23 source/local-options

如果你希望使用版本控制系統（VCS）管理 Debian 打包活動，你可以建立一個分支 (例如叫做 `upstream`) 來跟蹤上游程式碼，和另一個分支 (對於 Git 而言典型的是 `master` 分支) 來跟蹤你的 Debian 軟體包。對於後者，通常會將未應用補丁的上游程式碼和你的 `debian/*` 檔案放在一起以便容易合併上游的新程式碼。

在編譯軟件包之後，源代碼通常會被保持在應用補丁後的狀態。你需要手工執行 `quilt pop -a` 來解除這些補丁，然後再提交到 `master` 分支。你可以向 `debian/source/local-options` 文件裏添加一行 `unapply-patches` 來自動實現此目的。這個文件不會被加入到生成的源代碼包，它隻影響本地的編譯構建行爲。這個文件裏還可以包含 `abort-on-upstream-changes` (參看 dpkg-source(1))。

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

5.24 source/options

在源碼樹下自動生成的文件有時會超級討厭，因爲它們會生成毫無意義巨大無比的補丁文件。爲了減輕這種問題，可以用一些定製模塊比如 **dh_autoreconf**，參見節 4.4.3。

你可以給 `--extend-diff-ignore` 選項提供一個 Perl 正則表達式作爲 dpkg-source(1) 的參數，以此忽略在創建源碼包時自動生成的文件所發生的變更。

作爲這個自動生成文件的問題的通用解決辦法你可以存放像 **dpkg-source** 這樣的選項參數到源碼包的 `source/options` 文件中。如下將會跳過給 `config.sub`, `config.guess`, 以及 `Makefile` 創建補丁文件。

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

5.25 patches/*

舊的 `1.0` 源代碼包格式使用單一的大 `diff.gz` 文件爲源碼保存 `debian` 中的維護文件和補丁。這樣的軟件包比較難於在事後檢查和分析。這不是很好。

新的 `3.0 (quilt)` 源碼格式將補丁存儲在 `debian/patches/*` 中，用 **quilt** 命令。這些補丁和其他 `debian` 目錄下的打包數據都會被打包成 `debian.tar.gz` 文件。由於 **dpkg-source** 命令可以處理 **quilt** 格式的補丁數據 (在格式爲 `3.0 (quilt)` 的源碼中)，而不需要 `quilt` 軟件包；不需要在 `Build-Depends` 中添加 `quilt`。[8]

quilt 命令在 quilt(1) 中有詳細描述。它將對源代碼的修改維護於 `debian/patches` 中一系列 `-p1` 級別的補丁文件中，`debian` 目錄外的文件沒有任何修改。這些補丁的順序記錄於 `debian/patches/series` 文件中。你可以輕鬆地 apply (=push)、un-apply (=pop) 和 refresh 補丁。[9]

在章 3 中，我們在 `debian/patches` 創建了三個補丁。

因爲 Debian 補丁位於 `debian/patches`，請確定按照節 3.1 中的方法正確配置 **dquilt**。

如果在之後有人 (包括你自己) 提供了一個補丁 *`foo`*`.patch`，對於 `3.0 (quilt)` 源代碼包格式可以很容易修改：

[6] 參看 DebSrc3.0 (http://wiki.debian.org/Projects/DebSrc3.0) 以瞭解轉換到新的 `3.0 (quilt)` 和 `3.0 (native)` 源代碼格式時的總結。
[7] 實際上新格式還支持多個上游 tarball 和更多的壓縮方法，但這已超出本文件講述的範圍。
[8] 這裏已經提出了若干種補丁集維護方法，並且正爲 Debian 軟件包所採用。**quilt** 系統是最推薦的維護系統。而其他的有包括 **dpatch**, **dbs**, 以及 **cdbs**。其中有許多將補丁保存爲 `debian/patches/*` 文件。
[9] 如果你需要一個 sponsor 上傳你的軟件包，這種情況下，清晰的軟件包分離和記錄更改的文檔對於軟件包審查過程的順利程度非常重要。

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

用新的 3.0 (quilt) 源代碼包格式存儲的補丁必須有 清晰的邊界。你應該通過 dquilt pop -a; while quilt push; do quilt refresh; done 來驗證這點。

Chapter 6

構建套件

現在我們已經爲構建套件做好了準備。

6.1 完整的 (重) 構建

爲保證完整的軟件包 (重) 構建能順利進行，你必須保證系統中已經安裝

- `build-essential` 套件;
- 列於 `Build-Depends` 欄位的套件 (參考節 4.1);
- 列於 `Build-Depends-indep` 欄位的套件 (參考節 4.1)。

然後在原始碼目錄中執行以下命令:

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

這樣會自動完成所有從原始碼套件構建二進位套件的工作，包括:

- 清理原始碼樹 (`debian/rules clean`)
- 構建原始碼套件 (`dpkg-source -b`)
- 構建程式 (`debian/rules build`)
- 構建二進位套件 (`fakeroot debian/rules binary`)
- 製作 `.dsc` 文件
- 用 **dpkg-genchanges** 命令製作 `.changes` 文件。

如果構建結果令人滿意，那就用 **debsign** 命令以你的私有 GPG 金鑰簽署 `.dsc` 檔案和 `.changes` 檔案。你需要輸入密碼兩次。[1]

對於非本土 Debian 軟件包，比如 `gentoo`，構建軟件包之後，你將會在上一級目錄 (`~/gentoo`) 中看到下列文件:

[1]爲了連接至信任網絡，本 GPG 密鑰必須被一名 Debian 開發者簽署，而且必須註冊到 the Debian keyring (http://keyring.debian.org) (Debian 密鑰環) 中。這樣你上傳的軟件包就能被接受到 Debian 歸檔中了。參見 Creating a new GPG key (http://keyring.debian.org/creating-key.html) 以及 Debian Wiki on Keysigning (http://wiki.debian.org/Keysigning) 。

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

  這是原始的原始碼 tarball，最初由 `dh_make -f ../gentoo-0.9.12.tar.gz` 命令建立，它的內容與上游 tarball 相同，僅被重新命名以符合 Debian 的標準。

- `gentoo_0.9.12-1.dsc`

  這是一個從 `control` 文件生成的源代碼概要，可被 dpkg-source(1) 程序解包。

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

  這個壓縮的 Tar 歸檔包含你的 `debian` 目錄內容。其他所有對於源代碼的修改都由 **quilt** 補丁存儲於 `debian/patches` 中。

  如果其他人想要重新構建你的套件，他們可以使用以上三個檔案很容易地完成。只需複製三個檔案，再運行 `dpkg-source -x gentoo_0.9.12-1.dsc`。[2]

- `gentoo_0.9.12-1_i386.deb`

  這是你的二進位套件，可以使用 **dpkg** 程式安裝或反安裝它，就像其他套件一樣。

- `gentoo_0.9.12-1_i386.changes`

  這個文件描述了當前修訂版本軟件包中的全部變更，它被 Debian FTP 倉庫維護程序用於安裝二進制和源代碼包。它是部分從 `changelog` 和 `.dsc` 文件生成的。

  隨着你不斷完善這個軟件包，程序的行爲會發生變化，也會有更多新特性添加進來。下載你軟件包的人可以查看這個文件來快速找到有哪些變化，Debian 倉庫維護程序還會把它的內容發表至 debian-devel-changes@lists.debian.org (http://lists.debian.org/debian-devel-changes/) 郵件列表。

在上傳到 Debian FTP 倉庫中前，`gentoo_0.9.12-1.dsc` 檔案和 `gentoo_0.9.12-1_i386.changes` 檔案必須用 **debsign** 命令簽署，其中使用你自己存放在 `~/.gnupg/` 目錄中的 GPG 私鑰。用你的公鑰，可以令 GPG 簽名證明這些檔案真的是你的。

debsign 命令可以用來以指定 ID 的 GPG 金鑰進行簽署（這方便了贊助 (sponsor) 軟體包），只要照著下邊在 `~/.devscripts` 中的內容：

```
DEBSIGN_KEYID=Your_GPG_keyID
```

`.dsc` 和 `.changes` 文件中很長的數字串是其中提及文件的 SHA1/SHA256 校驗和。下載你軟件包的人可以使用 sha1sum(1) 或 sha256sum(1) 來進行覈對。如果校驗和不符，則說明文件已被損壞或偷換。

6.2 自動編譯系統

Debian 支持非常多的 移植 (port) (http://www.debian.org/ports/)，同時它有着 自動構建網絡 (http://www.debian.org/devel/-buildd/)，這個網絡在不同體繫結構的計算機上運行着 **buildd** 守護進程。雖然你不需要自己做這件事情，你也應該知道在你的軟件包身上發生了什麼。讓我們來簡要地看看他們是如何爲你給多個體繫結構構建軟件包的。[3]

對於 `Architecture: any` 的軟件包，自動編譯系統重構建它們。它確保系統中已經安裝

- `build-essential` 套件；
- 列於 `Build-Depends` 欄位的套件 (參考節 4.1)。

然後在原始碼目錄中執行以下命令：

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

這樣會自動完成從原始碼套件構建平臺依賴二進位套件的工作，包括：

[2] 你可以使用 `--skip-patches` 選項來在正常的提取操作後避免應用 `3.0 (quilt)` 源代碼包格式中的 **quilt** 補丁。你也可以在正常解壓後使用 `quilt pop -a` 還原這些補丁對源碼的修改。

[3] 實際中的自動構建系統包含了極爲複雜 (比這裏說明的還要複雜) 的體制。不過這類細節已經超出本文檔範圍。

- 清理原始碼樹 (`debian/rules clean`)
- 構建程式 (`debian/rules build`)
- 構建平臺依賴二進位套件 (`fakeroot debian/rules binary-arch`)
- 使用 **gpg** 簽署 `.dsc` 檔案
- 使用 **dpkg-genchanges** 和 **gpg** 創建並簽署上傳用的 `.changes` 檔案

這就是你看到你的套件在其他平臺上可用的原因。

儘管通常的打包工作中 `Build-Depends-indep` 字段中列出的軟件包都需要安裝 (參看節 6.1)，但是在編譯平臺依賴二進制包時它們無需在自動編譯系統上安裝。[4]通常打包和自動編譯系統的這種不同爲你指出如何考慮必須的軟件包應如何放在 `debian/control` 文件的 `Build-Depends` 或 `Build-Depends-indep` 域中 (參看節 4.1)。

6.3 debuild 命令

你可以使用 **debuild** 命令來進一步自動化 **dpkg-buildpackage** 的構建過程。參看 debuild(1)

debuild 命令會執行 **lintian** 命令，以在 Debian 軟體包構建結束之後進行靜態檢查。**lintian** 命令可以用下邊出現在 `~/.devscripts` 檔案中的項來定製：

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

在普通用戶帳號中可以使用以下這樣簡單的命令清理源代碼並重構建軟件包：

```
$ debuild
```

還可以這樣簡單地清理源代碼樹：

```
$ debuild -- clean
```

6.4 `pbuilder` 套件

對於使用淨室 (**chroot**) 編譯環境來驗證編譯依賴而言，`pbuilder` 軟件包是非常有用的。[5]它確保了軟件包在不同構架上的 `sid` 發行版環境中的自動編譯器中能乾淨地編譯，避免了總是被歸類於 RC (Release Critical，影響發佈) 的嚴重 FTBFS (Fails To Build From Source，從源代碼編譯失敗) Bug。[6]

我們通過以下操作來定製 `pbuilder` 軟件包。

- 設置 `/var/cache/pbuilder/result` 對當前用戶可寫。
- 創建一個對用戶可寫的目錄保存鉤子腳本，例如 */var/cache/pbuilder/hooks*
- 在 `~/.pbuilderrc` 或 `/etc/pbuilderrc` 中添加以下內容：

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

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

[4]和在 `pbuilder` 中不同，自動編譯系統使用的 `sbuild` 軟件包所維護的 **chroot** 不強制要求最小化的編譯系統，並可能保持很多軟件包始終安裝在其中。

[5]由於 `pbuilder` 軟件包仍然在進化，你應當查閱最新的官方文檔來檢查實際的配置狀況。

[6]參見 http://buildd.debian.org/ 以獲取更多關於 Debian 軟件包 auto-building 的信息。

```
$ sudo pbuilder create
```

如果你已經創建了源代碼包，在包含 *foo*.orig.tar.gz、*foo*.debian.tar.gz 和 *foo*.dsc 文件的目錄中執行下面的命令來更新 pbuilder **chroot** 系統以便在其中執行構建：

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

新構建的無 GPG 簽名的軟件包會被以非 root 屬主放置於 /var/cache/pbuilder/result/。

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

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

如果你已經更新了源代碼樹但沒有生成對應的源代碼包，在存放 debian 目錄的源碼目錄裏執行：

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

你可以使用 pbuilder --login --save-after-login 命令登錄到這個 **chroot** 環境中並按照需要對其進行設定。通過 ^D (Control-D) 離開這個 shell 時環境會被保存。

最新版的 **lintian** 命令可以通過設置鉤子腳本 */var/cache/pbuilder/hooks/*B90lintian 在 chroot 環境中運行。腳本內容如下：[7]

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

爲 sid 編譯軟件包需要使用 sid 環境。在現實中 sid 存在很多問題以至於你不願意將整個系統都遷移到其上。pbuilder 軟件包可以在這種情況下很好地解決問題。

你可能需要在 stable 軟體包釋出後為 stable-proposed-updates、stable/updates 等倉庫升級你維護的軟體包。[8]對於這類情況，“我正在執行 sid 系統”並不是你不為它們進行升級的充分理由。pbuilder 軟體包可以幫助你使用到相同 CPU 體系結構下幾乎所有 Debian 和 Debian 衍生版系統環境。

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

6.5 git-buildpackage 及其相似命令

如果你的上游對源代碼使用版本控制系統 (VCS)[9]，你也應該考慮使用它們。這會使得合併和提取上游補丁更加簡單。有多個爲不同 VCS 設計的包裝腳本包來協助 Debian 軟件包構建。

- git-buildpackage：幫助維護 Git 倉庫中 Debian 軟件包的套件。

[7] 此處默認 HOOKDIR=/var/cache/pbuilder/hooks。你可以在 /usr/share/doc/pbuilder/examples 目錄中找到很多鉤子腳本的例子。

[8] 升級你的 stable 套件有規定限制。

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

- `svn-buildpackage`：幫助維護 Subversion 倉庫中套件的程式。
- `cvs-buildpackage`：爲 CVS 源代碼樹設計的 Debian 軟件包腳本集。

`git-buildpackage` 的使用在 Debian 開發者之中非常流行，它可以使用 alioth.debian.org (http://alioth.debian.org/) 上的 Git 服務器來管理 Debian 軟件包。[10] 這個軟件包提供了許多命令來 自動化打包操作：

- git-import-dsc(1): 將一個早先的 Debian 軟件包導入到 Git 倉庫中。
- git-import-orig(1): 將上游 tar 文件導入到 Git 倉庫中。
- git-dch(1): 用 Git commit 信息來生成 Debian changelog。
- git-buildpackage(1): 從 Git 倉庫中構建 Debian 軟件包。
- git-pbuilder(1): 用 **pbuilder**/**cowbuilder** 從 Git 倉庫來構建 Debian 軟件包。

這些命令使用 3 個分支來跟蹤打包操作：

- `main` 用於 Debian 軟件源碼樹。
- `upstream` 用於上游源碼樹。
- 由 `--pristine-tar` 爲上游 tarball 生成的 `pristine-tar`。[11]

你可以設置 `git-buildpackage`，通過修改 `~/.gbp.conf` 文件。參見 gbp.conf(5). [12]

6.6 快速重構建

對於很大的軟件包，在調試 `debian/rules` 的過程中你可能不想每次都對整個軟件包進行重構建。僅用於測試目的，你可以不重新構建源代碼包而使用以下的方法創建 `.deb` 文件[13]：

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

或者可以通過以下方法驗證它是否能通過編譯：

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

一旦完成了調試，記住要按照前面所將的正常過程重構建你的套件。你可能無法正常上傳用此種方法構建的 `.deb` 檔案。

[10] Debian wiki Alioth (http://wiki.debian.org/Alioth) 這裏說明瞭如何使用 alioth.debian.org (http://alioth.debian.org/) 服務。

[11] `--pristine-tar` 選項會呼叫 **pristine-tar** 命令，它能利用一個很小的二進位制差量檔案以及在版本控制系統中某個分支內儲存的上游檔案內容（分支名通常為 `upstream`）重新生成與上游 tarball 完全相同的一份副本。

[12] 這裏有一些給高級讀者參考的網頁資源。

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

[13] 常規情形下被配置好的環境變數在此時不會被自動設定。永遠不要將使用這個 快速方法構建的軟體包上傳到任何地方。

6.7 命令層級

這裏有一個簡練的總結，關於在命令層次結構中有多少用於構建軟件包的命令能夠組合到一起。做這件事情的方法非常多。

- `debian/rules` = 軟件包構建過程的專用 maintainer script
- **dpkg-buildpackage** = 軟件包構建之核心工具
- **debuild** = **dpkg-buildpackage** + **lintian** (在乾淨的環境變量下構建)
- **pbuilder** = Debian chroot 環境核心工具
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (在 chroot 環境中構建)
- **cowbuilder** = 加速 **pbuilder** 執行
- **git-pbuilder** = 命令行語法友好 **pdebuild** (被 **gbp buildpackage** 使用)
- **gbp** = 在 Git 倉庫中管理 Debian 源碼
- **gbp buildpackage** = **pbuilder** + **dpkg-buildpackage** + **gbp**

即便像 **gbp buildpackage** 和 **pbuilder** 這樣的高級命令的應用可以確保完美的軟件包構建環境，瞭解像 `debian/rules` 和 **dpkg-buildpackage** 這樣的低級命令行工具如何被它們執行也是至關重要的。

Chapter 7

檢查套件中的錯誤

在上傳軟件包到公共倉庫前，你還需要知道一些檢查軟件包錯誤的技巧。

不僅在自己的機器上測試總是一個好主意。你必須謹慎地對待以下敘述的測試中顯示的任何一個警告或錯誤信息。

7.1 詭異可疑的改動

如果你在構建以 `3.0 (quilt)` 格式的非本土 Debian 軟件包後，發現一個新的自動生成的補丁，比如 `debian-changes-*` 在 `debian/patches` 目錄中有可能是你不小心更改了一些文件，或者構建腳本修改了上游源代碼。如果這是你犯下的小錯誤，那就修復它。如果這是構建腳本乾的好事，那就用 **dh-autoreconf** 來解決其根本問題，可參照節 4.4.3 或者可以變通一下，處理 `source/options` 文件，參照節 5.24.

7.2 校驗軟件包安裝過程

你必須測試你的軟體包看是否存在安裝問題。debi(1) 命令可以幫助你測試所有生成的二進位制軟體包。

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

你必須使用從 Debian 倉庫下載的 `Contents-`*`i386`* 檔案校驗是否在與不同包存在檔案衝突，以阻止在不同的系統上發生安裝故障。**apt-file** 命令正適合完成這個任務。如果存在衝突，請通過重新命名、將公共檔案分離到另一個受其他包依賴的包中、與受影響的軟體包的維護者合作使用 alternatives 機制來避免實際問題 (參看 update-alternatives(1)) 或在 `debian/control` 檔案中設定 `Conflicts` 條目以宣告衝突關係等方式避免問題的發生。

7.3 檢驗軟件包的 maintainer scripts

所有 maintainer scripts，包括 `preinst`、`prerm`、`postinst` 和 `postrm` 文件，都非常難以編寫，除非是由 `debhelper` 程序自動生成的。如果你是新維護人員則不要使用它們 (參看節 5.18)。

如果軟件包使用了這些需要嚴格測試的 maintainer scripts，請確保不僅測試 install，還要測試 remove、purge 和 upgrade。很多 maintainer scripts 的 Bug 都顯現於卸載或徹底刪除軟件包時。使用 **dpkg** 命令按以下方法來測試它們：

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

整個測試過程應按照以下操作序列完成：

- 如果可能，安裝前一個版本的套件；
- 從前一個版本升級套件；
- 降級套件到前一個版本 (可選)；
- 徹底刪除該套件；
- 全新安裝該套件；
- 反安裝該套件；
- 再次安裝該套件。
- 徹底刪除該套件；

如果這是你的第一個套件，你應該使用其他版本號創建一個測試用的套件來完成升級測試，這樣可以避免將來的問題。

請牢記如果你的套件已經在以往版本的 Debian 中發佈，人們通常會從最近發佈的 Debian 發佈裏的版本升級，所以也要測試從那個版本升級到當前的版本。

儘管降級沒有被正式支持，支持它也總是友好的。

7.4 使用 `lintian`

使用 lintian(1) 檢查你的 `.changes` 檔案。**lintian** 命令會運行很多測試腳本來檢查常見的打包錯誤。[1]

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

當然，要替換你自己軟件包所生成的 `.changes` 文件的文件名。**lintian** 命令的輸出常帶有以下幾種標記：

- `E`：代表錯誤：確定違反了 Debian Policy 或是一個肯定的打包錯誤。
- `W`：代表警告：可能違反了 Debian Policy 或是一個可能的打包錯誤。
- `I`：代表信息：對於特定打包類別的信息。
- `N`：代表註釋：幫助你調試的詳細訊息。
- `O`：代表已覆蓋：一個被 `lintian-overrides` 檔案覆蓋的訊息，但由於使用 `--show-overrides` 選項而顯示。

對於警告，你應該改進軟件包或者檢查警告是否的確無意義。如果確定沒有意義，則按照節 5.14 中的敘述使用 `lintian-overrides` 文件將其覆蓋。

注意，你可以用 **dpkg-buildpackage** 來構建軟件包，並執行 **lintian**，只要你使用了 debuild(1) 或 pdebuild(1).

7.5 debc 命令

你可以使用 debc(1) 命令列出一個二進制 Debian 軟件包中的文件。

```
$ debc package.changes
```

[1] 如果你按照節 6.3 中的敘述定義了 `/etc/devscripts.conf` 或 `~/.devscripts` 檔案，就不需要再添加 **lintian** 選項 `-i -I --show-overrides`。

7.6 debdiff 命令

你可以使用 debdiff(1) 命令比較兩個 Debian 原始碼套件的內容。

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

你還可以使用 debdiff(1) 命令比較兩個 Debian 二進位套件的檔案列表。

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

這個命令對於檢查源代碼包中哪些文件被修改了非常有用，還可以發現二進制包中是否有文件在更新過程中發生的變動，比如被意外替換或刪除。

7.7 interdiff 命令

你可以使用 interdiff(1) 命令比較兩個 `diff.gz` 檔案。這對於更新使用舊的 `1.0` 原始碼格式的套件時，檢查是否有意外的變更非常有用。

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

新的 `3.0` 源碼格式會將更改保存在多個補丁文件中，如節 5.25 所述。你也可以使用 **interdiff** 跟蹤每一個 `debian/patches/*` 文件中的改動。

7.8 mc 命令

很多檔案檢查操作可以通過使用類似 mc(1) 的檔案管理器來完成，它可以幫助你直接查看 `*.deb` 檔案的內容，除此之外還可以用於 `*.udeb`、`*.debian.tar.gz`、`*.diff.gz` 和 `*.orig.tar.gz` 檔案。

還要檢查在二進制包和源代碼包中是否有不需要的文件或者空文件。這些文件經常沒有被正確清理，如果存在這種情況，要調整 `rules` 文件進行處理。

Chapter 8

更新套件

一旦你發佈了一個軟件包，在之後的某個時間裏就需要對它進行更新。

8.1 新的 Debian 版本

假設你收到一個針對你的軟件包報告的 Bug，其編號爲 #654321，它描述了一個你可以解決的問題。要創建軟件包的一個新 Debian 修訂版本，你需要：

- 如果要將它記錄於新的補丁中，這樣做：
  - `dquilt new` *`bugname.patch`* 設置補丁名稱；
  - `dquilt add` *`buggy-file`* 聲明文件將被更改；
  - 修正套件程式碼中的上游 Bug；
  - `dquilt refresh` 將修改記錄到 *`bugname.patch`*；
  - `dquilt header -e` 添加對它的描述；
- 如果是更新一個已存在的補丁，這樣做：
  - `dquilt pop` *`foo.patch`* 重現已存在的 *`foo.patch`*；
  - 修正舊的 *`foo.patch`* 中的問題；
  - `dquilt refresh` 更新 *`foo.patch`*；
  - `dquilt header -e` 更新對它的描述；
  - `while dquilt push; do dquilt refresh; done` 應用所有補丁以確保它們 邊界清晰；
- 在 Debian `changelog` 檔案的頂部添加一個條目。例如可以使用 `dch -i` 或用 `dch -v` *`version-revision`* 來指定版本，然後用你喜歡的編輯器插入訊息。[1]
- 在 changlog 條目中簡要描述 Bug 和相應的解決辦法，並在後面添加 `Closes: #654321`。這樣 Bug 報告會在你的軟件包被 Debian 倉庫接受的同時被倉庫管理軟件 自動關閉。
- 重複上述操作來修復更多的 Bug，並在需要的時候使用 `dch` 更新 Debian `changelog` 檔案。
- 重複你在節 6.1 和章 7 中所做的事情。
- 一旦你滿意了，那就將 `changelog` 中的發行版值由 `UNRELEASED` 修改成目標發行版值 `unstable` (或者是 `experimental`)。[2]

[1] 要獲得需要的日期格式，使用 `LANG=C date -R`。

[2] 如果你用 `dch -r` 命令來使它成爲最後一筆更改，請確保用編輯器顯式地保存 `changelog` 文件。

- 按照章 9 來上傳軟體包。惟一的區別是這次不需要再包含原始程式碼檔案，因為它們沒有變化且已經存在於 Debian 倉庫中。

有一種棘手的情況，當你在上傳正常版本到官方倉庫中之前，你製作了一個本地包以進行打包實驗，例如, *1.0.1-1*。為了平滑升級，建立一個 changelog 條目，其中包含類似 *1.0.1-1~rc1* 這樣的版本字串不失為一劑良方。你可以通過合併這樣的本地修改條目到官方包的單個條目中來整理 changelog。參見節 2.6 來了解版本字串的排序。

8.2 檢查新上游版本

在為 Debian 倉庫準備新上游版本的軟體包前，你必須首先對新的上游釋出版本進行檢查。

檢查工作應從閱讀上游 changelog、NEWS 以及所有隨新版本一同發佈的文檔。

然後應按照以下步驟檢查新舊版本之間源碼的差別，小心任何可疑的內容：

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

對於 Autotools 自動生成的文件發生的改動，例如 missing、aclocal.m4、config.guess、config.h.in、config.sub、configure、depcomp、install-sh、ltmain.sh 和 Makefile.in 是可以忽略的。你可以在運行 **diff** 進行代碼檢查前刪除它們。

8.3 新上游版本

如果軟件包 *foo* 是使用新的 3.0 (native) 或 3.0 (quilt) 格式打包的，製作新的上游版本時需要先把舊的 debian 目錄移至新的源代碼內。這可以通過在新解壓的源代碼目錄裏運行 tar xvzf */path/to/foo_oldversion*.debian 完成。[3]當然，你需要做幾個很顯然的雜事：

- 創建一份上游原始碼的副本，命名爲 foo_*newversion*.orig.tar.gz
- 使用 dch -v *newversion-1* 更新 Debian changelog 檔案。
  - 添加一個條目，內容爲 New upstream release。
  - 簡明地介紹 在新上游版本中上游修復和關閉的 Bug (添加 Closes: *#bug_number*) 。
  - 簡明地介紹維護者對 本個新上游版本做出的修改，修復和關閉的 Bug (添加 Closes: *#bug_number*)。
- 運行 while dquilt push; do dquilt refresh; done 以應用全部補丁並使它們 邊界清晰。

如果補丁沒有乾淨地被應用，檢查原因 (線索在 .rej 檔案裏)。

- 如果你的補丁已經被上游接受，
  - 使用 dquilt delete 刪除它。
- 如果你的補丁與上遊程式碼中的變更有衝突：
  - 使用 dquilt push -f 應用舊補丁，未應用的部分會被保存爲 *baz*.rej。
  - 手工編輯 *baz* 文件來在新的代碼中實現 *baz*.rej 中應有的效果。
  - 使用 dquilt refresh 更新補丁。
- 正常繼續，執行 while dquilt push; do dquilt refresh; done。

[3]如果套件 *foo* 是使用舊的 1.0 格式的，可以在新解壓的原始碼目錄裏運行 zcat */path/to/foo_oldversion*.diff.gz|patch -p1 來完成。

這個過程可以通過使用 uupdate(1) 來更自動化地完成：

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

如果你按照節 5.21 的敘述設置了 debian/watch 檔案，你可以跳過這個 **wget** 命令，轉而在 *foo-oldversion* 目錄中運行 uscan(1)，且無需再執行 **uupdate** 命令。它會 自動查找新的原始碼、下載並運行 **uupdate** 命令。[4]

重複節 6.1 、章 7 和章 9 中的操作，即可發佈此更新的套件。

8.4 更新打包風格

更新打包風格不是更新軟件包的必須步驟，但是這樣可以使你的軟件包得到對現代的 debhelper 系統和 3.0 源代碼包格式完整的兼容性。[5]

- 如果你需要重新添加已刪除的模板文件，可以在同一個 debian 軟件包源代碼樹中運行 **dh_make**，並添加 --addmissing 選項。然後對模板進行相應的編輯。
- 如果軟件包的 debian/rules 文件沒有更新爲使用 debhelper v7+ 的 **dh** 語法，則更新它使用 **dh**。在需要的時候更新 debian/control 文件。
- 如果你希望將使用 cdbs 的 Makefile 包含機制創建的 rules 文件更新爲 **dh** 語法，參看下文並理解各 DEB_* 配置變量。
  - /usr/share/doc/cdbs/cdbs-doc.pdf.gz 的本地副本
  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)
- 如果你有一個不帶有 *foo*.diff.gz 文件的 1.0 格式的源代碼包，你可以通過創建 debian/source/format 文件並在其中添加 3.0 (native) 來將其更新爲新的 3.0 (native) 源代碼包格式。debian 目錄中的其他文件可以直接複製過來。
- 如果你有一個帶有 *foo*.diff.gz 文件的 1.0 格式的源代碼包，你可以通過創建 debian/source/format 文件並在其中添加 3.0 (quilt) 來將其更新爲新的 3.0 (quilt) 源代碼包格式。debian 目錄中的其他文件可以直接複製過來。如果需要，把 filterdiff -z -x '*/debian/*' *foo*.diff.gz > big.diff 生成的 big.diff 文件導入到 **quilt** 系統。[6]
- 如果它使用了其他的補丁系統，例如 dpatch、dbs 或 cdbs，使用 -p0、-p1 或 -p2 級別，使用 http://bugs.debian.org/-581186 (http://bugs.debian.org/581186) 的 deb3 命令將其轉換到 quilt 系統。

[4]如果 **uscan** 命令下載並更新了原始碼，但沒有運行 **uupdate** 命令，你應該修正 debian/watch 檔案，使 URL 末尾後帶有 debian uupdate。
[5]如果你的 sponsor 或其他維護者一定反對更新已有的打包風格，則不值得去爲此煩惱或爭論，總是有更重要的事要做。
[6]你可能使用 **splitdiff** 命令將 big.diff 分割爲多個增量補丁。

- 如果它使用 **dh** 命令的 `--with quilt` 選項，或 **dh_quilt_patch** 和 **dh_quilt_unpatch** 命令，刪除它們並使其使用新的 `3.0 (native)` 源代碼包格式。

你應當查看 DEP - Debian Enhancement Proposals (http://dep.debian.net/) 並採納 ACCEPTED 建議。

當然你還需要按照節 8.3 完成其他的步驟。

8.5 UTF-8 轉換

如果上游文檔採用了老式編碼，那麼將其轉換爲 UTF-8 不失爲一良方。

- 用 iconv(1) 來轉換普通文本文件的編碼。

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

- 使用 w3m(1) 來把 HTML 文件轉換爲 UTF-8 普通文本文件。當你這樣做的時候，請確認在 UTF-8 locale 下執行。

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

8.6 對更新套件的幾點提示

以下是對更新軟體包的幾點提示：

- 保留舊的 `changelog` 條目 (看似顯然，但是總有可能把 `dch -i` 輸入爲 `dch`)。
- 已存在的 Debian 修改需要被重新校驗，去除上游已經接受的東西，除非有必要的原因，還要記錄尚未被上游接受的部分。
- 如果對編譯系統作出了修改 (希望你已經在檢查上游變更時瞭解了這些)，那麼要在必要時更新 `debian/rules` 和 `debian/control` 編譯依賴關係。
- 檢查 Debian Bug Tracking System (BTS) (http://www.debian.org/Bugs/) 是否有人爲某些仍然未修復的 bug 提供了補丁。
- 檢查 `.changes` 文件以確保你正要上傳到正確的發行版、正確的列出 BUG 關閉 `Closes` 字段、`Maintainer` 和 `Changed-By` 字段相匹配，以及文件是否已經使用 GPG 簽署等。

Chapter 9

上傳套件

現在你完成了對軟件包的徹底測試，接下來將其釋出到公共歸檔中分享它吧。

9.1 上傳到 Debian 倉庫

當你成爲正式的開發人員[1]，你可以把軟件包上傳到 Debian 倉庫[2]。你可以手工進行這項工作，但使用例如 dupload(1) 或 dput(1) 的自動化工具可以幫你更好地完成這項操作。在此我們將敘述如何使用 **dupload** 操作。[3]

首先需要設置 **dupload** 的設定檔案。你既可以編輯系統級的 `/etc/dupload.conf` 檔案，也可以使用自己的 `~/.dupload.conf` 檔案覆蓋一些需要修改的設置。

你可以閱讀 dupload.conf(5) man 手冊頁來瞭解各選項的含義。

`$default_host` 選項決定了默認使用哪個上傳隊列，`anonymous-ftp-master` 是最基本的一個，但你很可能希望改用其他的。[4]

連接到互聯網後，可以使用以下命令上傳你的軟件包：

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

dupload 會檢查文件的 SHA1/SHA256 校驗和是否與 `.changes` 文件中的相匹配，如果不匹配它會做出警告。你應按照如節 6.1 所述來重構建軟件包使得它可以被正常上傳。

如果你在 ftp://ftp.upload.debian.org/pub/UploadQueue/ 遇到了上傳問題，你可以通過 **ftp** 來手動上傳 GPG 簽署的 `*.commands` 文件。[5] 比如說，使用 `hello.commands` 命令：

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

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

[1] 參見節 1.1。

[2] 有許多公開的檔案比如 http://mentors.debian.net/ ，它們的運作方式幾乎與 Debian 檔案一致，並提供了一個非開發者的上傳區域。你可以自己建立一個等效檔案，只要使用 http://wiki.debian.org/HowToSetupADebianRepository 裏邊列舉出來的工具。所以這一小節也對非開發者特別有用。

[3] `dput` 軟件包提供了更多的特性，相比於 `dupload` 也越來越受歡迎。它使用 `/etc/dput` 文件作爲全局配置文件、`~/.dput.cf` 作爲用戶配置文件。它也直接支持 ubuntu 相關的服務。

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

[5] 參見 ftp://ftp.upload.debian.org/pub/UploadQueue/README。或者是，你可以使用 **dcut** 命令，它來自 `dput` 軟件包。

9.2 在上傳時包含 `orig.tar.gz` 檔案

第一次向倉庫上傳軟件包時要包含 `orig.tar.gz` 源代碼歸檔。如果這個軟件包的修訂號既不是 `1` 也不是 `0`, 那你就必須給 **dpkg-buildpackage** 加上選項 `-sa`。

對於 **dpkg-buildpackage** 命令：

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

對於 **debuild** 命令：

```
$ debuild -sa
```

對於 **pdebuild** 命令：

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

另一方面，請注意 `-sd` 選項會強制排除原始的 `orig.tar.gz` 源代碼。

9.3 跳過的上傳

如果你在 `debian/changelog` 建立了多個條目並跳過了上傳，你必須建立一個相應的 `*_.changes` 檔案，其中包含自上次上傳以來的全部變更記錄。這可以通過指定 **dpkg-buildpackage** 的 `-v` 並將版本傳遞給它來完成。比如，*`1.2`*.

對於 **dpkg-buildpackage** 命令：

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

對於 **debuild** 命令：

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

對於 **pdebuild** 命令：

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

Appendix A

高級打包

這裡有一些關於你可能遇到的高階打包問題的提示。如果有需要的話，本教程強烈建議閱讀這裡引用和建議的文件。

你可能需要手工編輯由 **dh_make** 命令生成的打包模板文件，以此來解決本章中所討論的問題。新的 **debmake** 命令應該能更好地解決這些問題。

A.1 共享庫

在打包 共享庫 之前，你應該閱讀以下的主要參考資料：

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

以下是幫助你開始的極簡解釋：

- 共享庫均爲 elf 對象文件，其包含編譯好的機器碼。
- 共享庫以 `*.so` 文件的形式發放。(既非 `*.a` 文件也非 `*.la` 文件)
- 共享庫主要用於在不同的二進制可執行程序之間共享代碼，這背後使用了 **ld**（譯註：鏈接）機制。
- 共享庫有時會爲一個可執行程序提供多個插件，這背後使用了 **dlopen** 機制。
- 共享庫能匯出代表著變數，函式和類的 symbols（符號）；並允許連結到它的可執行檔案訪問之。
- 共享庫 `libfoo.so.1` 中的 SONAME：`objdump -p libfoo.so.1 | grep SONAME` [1]
- 共享庫的 SONAME 常常與庫文件自身文件名一致 (不過有特例)。
- 鏈接到 `/usr/bin/foo` 的共享庫的 SONAME：`objdump -p /usr/bin/foo | grep NEEDED` [2]
- `libfoo1`: 共享庫 `libfoo.so.1` 的庫文件包，其 SONAME ABI 版本爲 `1`.[3]
- 在某些情況下，庫軟件包的 maintainer scripts 必須調用 **ldconfig** 來爲 SONAME 創建必要的符號鏈接。[4]
- `libfoo1-dbg`: 包含了除錯共享庫包用的除錯符號的軟體包 `libfoo1`.

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

- `lib`*`foo`*`-dev`: 包含了標頭檔案等內容的開發包。用於 `lib`*`foo`*`.so.`*`1`*.[5]
- 一般而言，Debian 軟體包不應當包含 `*.la` Libtool 歸檔檔案。[6]
- 一般來說，Debian 軟體包不應當使用 RPATH。[7]
- 雖然這有點過時，而且是第二參考，Debian Library Packaging Guide (http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html) 可能仍然對你有用。

A.2 管理 `debian/`*`package`*`.symbols`

當你給共享庫打包時，你應當建立 `debian/`*`package`*`.symbols` 檔案來管理在共享庫名稱不變，在同一個 SONAME 下又要提供 ABI 向後相容性的情況下每個符號關聯到的最小版本號。[8] 你可以閱讀下邊的主要參考以獲知細節：

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

這是個粗略的例子，用來來演示建立 `libfoo1` 軟體包的方法，此時使用上游 `1.3` 版本，有著妥當的`debian/libfoo1.symbols` 檔案：

- 使用上游提供的 `libfoo-1.3.tar.gz` 文件來準備 Debian 化的源碼骨架。
  - 如果這是庫軟件包 `libfoo1` 的第一次打包，那麼以空內容創建 `debian/libfoo1.symbols` 文件。
  - 如果之前的上游版本 `1.2` 已經被 `libfoo1` 軟件包打包了，並且其源碼包中有妥當的 `debian/libfoo1.symbols`，再用它一次。
  - 如果前一個上游 `1.2` 版本打包時沒有 `debian/libfoo1.symbols`，那就從具有相同庫 SONAME 的同一個共享庫包的所有可用的二進位制軟體包中建立它並命名為 `symbols` 檔案。比如 `1.1-1` 和 `1.2-1`。[10]

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

- 嘗試用像 **debuild** 和 **pdebuild** 這樣的工具來對原始碼樹進行試構建。(如果這因為缺失符號之類原因而失敗，那麼這裡就有一些不向後相容的 ABI 改變，這就需要你轉移 (bump) 共享庫的名稱到諸如 `libfoo1a`，並重新開始一次。)

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

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

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

[8] 向後不兼容的 ABI 變更常常需要你更新共享庫的 SONAME，並把共享庫名稱換成新的。

[9] 對於 C++ 庫和其他追蹤單個符號過於困難的情況下，請遵循 Debian Policy Manual, 8.6.4 "The shlibs system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) 。

[10] 所有先前的 Debian 軟件包版本都能在 http://snapshot.debian.org/ (http://snapshot.debian.org/) 找到。不過 Debian 修訂號被去掉了，以使軟件包的 backport 更爲容易：`1.1` << `1.1-1~bpo70+1` << `1.1-1` and `1.2` << `1.2-1~bpo70+1` << `1.2-1`

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

- 如果你如上述看見由 **dpkg-gensymbols** 命令打印出來的差異，那就從生成的二進位制庫包中抽取妥當更新的 `symbols` 檔案。[11]

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

- 使用像 **debuild** 和 **pdebuild** 這樣的工具來構建發行軟件包。

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

對上邊這個例子補充一點，我們需要進一步檢查 ABI (應用程序二進制接口) 兼容性並在需要的時候手動更新一些符號的版本。[12]

雖然這只是第二參考，Debian wiki UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) 和它指向的頁面可能會有所幫助。

A.3 多體繫結構

Debian wheezy 引入的多體繫結構特性，集成了對二進制包跨體繫結構安裝的支持 (尤其是 `i386<->amd64`，其他的組合也有) 於 `dpkg` 和 `apt` 中。你可以閱讀下邊的參考：

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

它爲每個共享庫的安裝路徑使用了類似 `i386-linux-gnu` 和 `x86_64-linux-gnu` 這樣的三元名字。實際上每個二進制軟件包構建的三元路徑是被動態設置到 `$(DEB_HOST_MULTIARCH)` 變量中的，經由 dpkg-architecture(1) 命令。舉個例子，安裝多體繫結構庫文件的路徑被按照下表進行了修改：[13]

[11] Debian 修訂號已被從版本中去掉，這能讓軟件包的 backport 更爲容易：`1.3` << `1.3-1~bpo70+1` << `1.3-1`

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

[13] 老舊且具有特殊用途的庫路徑，比如 `/lib32/` 和 `/lib64/` 已不再使用。

舊路徑	**i386** 多體繫結構路徑	**amd64** 多體繫結構路徑
/lib/	/lib/i386-linux-gnu/	/lib/x86_64-linux-gnu/
/usr/lib/	/usr/lib/i386-linux-gnu/	/usr/lib/x86_64-linux-gnu/

下面是一些典型的多體系結構軟體包分離情景:

- 庫源碼 lib*foo*-1.tar.gz
- 一個用編譯型語言編寫的工具的源碼 *bar*-1.tar.gz
- 一個用解釋型語言編寫的工具的源碼 *bar*-1.tar.gz

軟件包	體繫結構:	多體繫結構:	軟件包內容
lib*foo1*	任何	相同	共享庫，可共同安裝
lib*foo1*-dbg	任何	相同	共享庫調試符號，可共同安裝
lib*foo*-dev	任何	相同	共享庫頭文件之類，可共同安裝
lib*foo*-tools	任何	外來	運行時支持程序，不可共同安裝
lib*foo*-doc	全部	外來	共享庫文檔
bar	任何	外來	編譯好的程序文件，不可共同安裝
bar-doc	全部	外來	程序的配套文檔文件
baz	全部	外來	解釋型程序文件

請注意,開發軟件包應該包含一個指向共享庫的符號鏈接並且 不帶有版本號。比如:/usr/lib/x86_64-linux-gnu/libfoo.so -> libfoo.so.1

A.4 構建共享庫包

你可以用 dh(1) 通過以下方法構建一個支援多體系結構的 Debian 庫軟體包:

- 更新 debian/control。
  - 為原始碼包部分新增 Build-Depends: debhelper (>=10) 部分。
  - 爲每個二進制庫軟件包添加 Pre-Depends: ${misc:Pre-Depends}。
  - 在每個二進制包的段中添加 Multi-Arch: 小節。
- 設定 debian/compat 為”10”。
- 將所有打包腳本中的路徑從普通的 /usr/lib/ 調整到多體繫結構的 /usr/lib/$(DEB_HOST_MULTIARCH)/。
  - 首先,在 debian/rules 中呼叫 DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTI 以設定 DEB_HOST_MULTIARCH 變數。
  - 在 debian/rules 中用 /usr/lib/$(DEB_HOST_MULTIARCH)/ 替換 /usr/lib/。
  - 如果 debian/rules 檔案中的 override_dh_auto_configure 目標使用了 ./configure 檔案，那麼請確認用 dh_auto_configure -- 來替換它。[14]
  - 在debian/*foo*.install 文件中將所有 /usr/lib/ 的事件替換爲 /usr/lib/*/。
  - 如需從 debian/*foo*.links.in 動態地生成像 debian/*foo*.links 這樣的檔案，可以新增一個指令碼到 debian/rules 檔案的 override_dh_auto_configure 目標中。

[14]作爲替代,你可以添加 --libdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH) 和 --libexecdir=\$${prefix}/lib/$(DEB_HOST_MULTI 參數到 ./configure 後頭。請注意 --libexecdir 指定了安裝可執行程序 (它們被其他程序使用，而更少是用戶) 的默認路徑。它的 Autotools 默認設置爲 /usr/libexec/ 但是 Debian 的設置爲 /usr/lib/。

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

請確認該共享庫軟件包僅僅包含預期中的文件，並且你的 -dev 軟件包還奏效。

所有作為多體系結構軟體包而同時安裝到同一個檔案路徑的所有檔案應當具有完全一致的檔案內容。你必須小心由資料位元組序和壓縮演算法造成的區別。

A.5 Debian 本土軟件包

如果一個軟件包是僅僅爲 Debian 維護的，或者是可能的本地使用，那麼它的源碼可以容納所有的 `debian/*` 於其中。這裏有它的兩種打包方式。

你可以將除 `debian/*` 檔案之外的部分製作成上游 tarball，然後將其作為非本土 Debian 軟體包來打包，正如節 2.1 所述。這是一些人鼓勵使用的普通方法。

另一種方法就是本土 Debian 軟件包的打包工作流。

- 用包含所有文件的，單一壓縮過的 tar 文件，以 `3.0 (native)` 格式來創建本土 Debian 源碼包。
  - *package_version*`.tar.gz`
  - *package_version*`.dsc`
- 用 Debian 本土源碼包構建二進制包
  - *package_version_arch*`.deb`

比如說，如果你的原始碼檔案都存放在 `~/mypackage-1.0` 中，而且沒有 `debian/*` 這些檔案，那麼你可以用它建立一個本土 Debian 軟體包，只要按照下邊的方法使用 **dh_make** 命令：

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

接下來 `debian` 目錄和它的內容都會被建立，正如節 2.8 中那樣。這不會建立一個 tarball，因為這是個本土 Debian 軟體包。不過這也是唯一的區別。剩下的打包操作就是完全一致的了。

在執行了 **dpkg-buildpackage** 命令後，你將會在上一級目錄中看到這些文件：

- `mypackage_1.0.tar.gz`
  這是 **dpkg-source** 命令用 `mypackage-1.0` 目錄創建出來的源代碼 tarball 。(它的文件名後綴不是 `orig.tar.gz`)
- `mypackage_1.0.dsc`
  這是對原始碼內容的簡述，正如在非本土 Debian 軟體包中那樣。(沒有 Debian 修訂號。)
- `mypackage_1.0_i386.deb`
  這是完成的二進位制包，正如在非本土 Debian 軟體包中那樣。(沒有 Debian 修訂號。)
- `mypackage_1.0_i386.changes`
  這個文件描述了這個軟件作爲外來 Debian 包，在當前版本所作出的所有更改。(沒有 Debian 修訂)