Debian 新メンテナーガイド

Josip Rodin, Osamu Aoki, 倉澤望, 八津尾雄介, 佐々木洋平, 倉敷悟, 、青木修
この文書は GNU 一般公有使用許諾書、バージョン 2 もしくはそれ以降により規定される条件の下で使用できます。この文書は以下の二つの文書を参考にして書かれました:

- Debian パッケージの作り方（別名 Debmake マニュアル）、Copyright © 1997 Jaldhar Vyas.
- 新メンテナー向け Debian パッケージング法、copyright © 1997 Will Lowe.

The rewrite of this tutorial document with updated contents and more practical examples is available as "Guide for Debian Maintainers". Please use this new tutorial as the primary tutorial document.
## COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td>Josip Rodin, Osamu Aoki, 倉澤望, 八津尾雄介, 佐々木洋平, 倉敷悟, 青木修</td>
<td>October 8, 2022</td>
<td></td>
</tr>
<tr>
<td></td>
<td>日本語訳</td>
<td>October 8, 2022</td>
<td></td>
</tr>
<tr>
<td></td>
<td>日本語訳</td>
<td>October 8, 2022</td>
<td></td>
</tr>
<tr>
<td></td>
<td>日本語訳</td>
<td>October 8, 2022</td>
<td></td>
</tr>
<tr>
<td></td>
<td>日本語訳</td>
<td>October 8, 2022</td>
<td></td>
</tr>
<tr>
<td></td>
<td>日本語訳</td>
<td>October 8, 2022</td>
<td></td>
</tr>
</tbody>
</table>

## REVISION HISTORY

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>NAME</th>
</tr>
</thead>
</table>
## Contents

1 まずは正攻法で始めよう .......................... 1
   1.1 Debianにおける社会ダイナミクス ...................... 1
   1.2 開発に必要なプログラム ............................... 3
   1.3 開発に必要な文書 .................................. 4
   1.4 相談するには .................................... 5

2 はじめの一歩 ......................................... 6
   2.1 Debianパッケージビルドのワークフロー ................... 6
   2.2 プログラムの選定 .................................. 7
   2.3 プログラムの入手と検証 .............................. 9
   2.4 単純なビルドシステム ................................ 10
   2.5 ポピュラーなポータブルなビルドシステム ............... 10
   2.6 パッケージ名とバージョン ............................ 11
   2.7 dh_makeのセットアップ ............................... 12
   2.8 最初のノンネイティブ Debianパッケージ ............... 12

3 ソースコードの変更 ................................ 14
   3.1 quiltのセットアップ ................................. 14
   3.2 アップストリームのバグを修正する ..................... 14
   3.3 指定場所へのファイルのインストール ................... 15
   3.4 ライブラリの相違 .................................. 18

4 debian/ディレクトリー以下に無くてはならないファイル .................. 19
   4.1 control ............................................. 19
   4.2 copyright .......................................... 23
   4.3 changelog .......................................... 24
   4.4 rules ............................................... 25
      4.4.1 rulesファイルのターゲット ....................... 26
      4.4.2 デフォルトのrulesファイル ..................... 26
      4.4.3 rulesファイルのカスタマイズ ................... 29
5 debian ディレクトリーにあるその他のファイル

5.1 README.Debian ................................. 33
5.2 compat ......................................... 33
5.3 conffiles ....................................... 33
5.4 package.cron.* .............................. 34
5.5 dirs ............................................ 34
5.6 package.doc-base ......................... 34
5.7 docs ............................................ 35
5.8 emacsen-* ..................................... 35
5.9 package.examples ......................... 35
5.10 package.init と package.default .... 35
5.11 install ....................................... 36
5.12 package.info ................................. 36
5.13 package.links ............................... 36
5.14 {package,,source/}lintian-overrides .. 36
5.15 manpage.* ................................... 36
      5.15.1 manpage.1.ex ......................... 37
      5.15.2 manpage.sgml.ex ..................... 37
      5.15.3 manpage.xml.ex ...................... 37
5.16 package.manpages ......................... 38
5.17 NEWS ......................................... 38
5.18 {pre,post}{inst,rm} ...................... 38
5.19 package.symbols ......................... 39
5.20 TODO .......................................... 39
5.21 watch ......................................... 39
5.22 source/format ............................... 39
5.23 source/local-options .................... 40
5.24 source/options ............................. 40
5.25 patches/* .................................. 40

6 パッケージのビルド .................................. 42
6.1 完全な (再) ビルド .......................... 42
6.2 オートビルダー ............................... 43
6.3 debuild コマンド ............................ 44
6.4 pbuilder パッケージ ......................... 45
6.5 git-buildpackage コマンドとその仲間 .. 46
6.6 部分的な再ビルド ............................ 47
6.7 コマンド階層 ................................. 47
7 パッケージのエラーの検証 48
7.1 怪しげな変更 48
7.2 インストールに対するパッケージの検証 48
7.3 パッケージのメンテナースクリプトの検証 48
7.4 Using lintian 49
7.5 debc コマンド 50
7.6 debdiff コマンド 50
7.7 interdiff コマンド 50
7.8 mc コマンド 50

8 パッケージの更新 51
8.1 Debian リビジョンの更新 51
8.2 新規のアップストリームリリースの検査 52
8.3 アップストリームソフトウェアの新版更新 52
8.4 パッケージ化スタイルの更新 53
8.5 UTF-8 変換 54
8.6 パッケージをアップグレードする際の注意点 54

9 パッケージをアップロードする 56
9.1 Debian アーカイブへアップロードする 56
9.2 アップロード用 orig.tar.gz の内容 57
9.3 スキップされたアップロード 57

A 上級パッケージング 58
A.1 共有ライブラリー 58
A.2 debian/package.symbols の管理 59
A.3 マルチアーチ 60
A.4 共有ライブラリーパッケージのビルド 61
A.5 ネイティブ Debian パッケージ 62
Chapter 1
まずは正攻法で始めよう


この文書では、一般的な Debian ユーザーやデベロッパーを目指している人を対象に Debian パッケージのビルド方法の解説を試みます。技術用語はできるだけ避けて、実用的な例を多用しています。古いラテンの諺にもあるように、Longum iter est per praecepta, breve et efficax per exempla (百聞は一見にしかず)です。

本文書は多くの翻訳があるため Debian Buster リリースでは提供されます。本文書はこれに続くリリースでは内容が陳腐化しているため提供されなくなります。

Debian を最高峰の Linux ディストリビューションたらしみている理由のひとつで、そのパッケージ管理システムです。すでに膨大な数のソフトウェアが Debian 形式で配布されていますが、まだパッケージ化されていないソフトウェアをインストールしなければならないことがあるでしょう。どうやったら自分でパッケージが作れるんだろうとか、それはとても難しいことなんじゃないかなどと考えたことありません。確かに、もしあなたが本当に駆け出しの Linux ユーザーなら難しいでしょうが、それなら今この文書を読まなければよろしくね。-:- Unix のプログラミングについて少々知っている必要がありますが、神様みたいに精通している必要は全くありません。

ただ、確かなことがひとつあります。Debian パッケージをきちんと作成し保守していくには時間がかかるということです。間違えないでください。Debian のシステムが機能するには、メンテナーは技術的に有能であるだけでなく、勤勉であることも必要なのです。

パッケージ作成において手助けが必要な際には、項1.4を読んでください。


本書は入門書ですので、一部の重要なトピックスは詳細なステップを個々説明するようにしました。あなたには不要と思われる部分があるかもしれません。我慢して下さい。本文書を簡潔にするように一部のコーナーケースを意識的に省略したり参照を提供するだけに止めています。

1.1 Debian における社会ダイナミクス

あなたが Debian と関わる際の準備となることを望み、Debian の社会ダイナミクスの観察結果を記します。

・ 我々全員はボランティアです。

1文中では、jessie より新しいシステムを使っていると想定しています。古いシステム (古い Ubuntu システム等を含む) を使ってこの文書についていきたいのであれば、少なくともバックポートされた dpkg および debhelper パッケージをインストールする必要があります。

2Debian システムの基本的な操作は Debian Reference (http://www.debian.org/doc/manuals/debian-reference/) から学びます。Unix プログラミングに関しては学ぶいくつかのポインターも含まれています。
他人に何をするかを押し付けてはいけません。自分自身で行う意欲を持つべきです。

友好的な協力が推奨です。

あなたの寄与は他人にストレスを掛けすぎてはいけません。
あなたの寄与は他人に評価されて初めて価値があります。

Debianは教師の注意が自動的にあなたに注がれるあなたの学校とは違います。

多様な案件の独学能力を持つべきです。

他ボランティアに注意を払ってもらうことは貴重なリソースです。

Debianは常に改良されています。

あなたには高品質パッケージを作成することが期待されています。

あなたの寄与は他人にストレスを掛けすぎてはいけません。
あなたの寄与は他人に評価されて初めて価値があります。

Debian界隈では異なる役割で色々なタイプの人が交流しています。

upstream author (アップストリームの作者): 元のプログラムを作った人です。

upstream maintainer (アップストリームのメンテナー): 現在プログラムをメンテナンスしている人です。

maintainer (メンテナー): プログラムのDebianパッケージを作成している人です。

sponsor (スポンサー): メンテナーのパッケージを、(内容をチェックした後で)公式Debianパッケージアーカイブにアップロードするのを手伝う人です。

mentor (指導役): 新米メンテナーをパッケージを作成等で手助けする人です。

Debian Developer (Debian デベロッパー) (DD): 公式Debianパッケージアーカイブへの全面的アップロード権限を持っているDebianプロジェクトのメンバーです。

Debian Maintainer (Debian メンテナー) (DM): 公式Debianパッケージアーカイブへの限定的アップロード権限を持っている人です。

技術的なスキル以外の要件があるので、一夜にして正式なDebianデベロッパー(DD)になることはできません。これでがっかりしないでください。あなたのパッケージが他の人にとっても有用ならば、あなたはそれをメンテナーとしてスポンサーを通して、あるいはDebianメンテナーとして、アップロードすることが可能です。

正式なDebianデベロッパーになるのに新しいパッケージの作成は必須ではないことに注意してください。既存のパッケージに対して貢献していくことでも正式なDebianデベロッパーの道は開かれます。多くのパッケージがよいメンテナーを待っています(項2.2を参照)。

本書ではパッケージングの技術面のみフォーカスするので、Debianが如何にして機能し、あなたが如何にすれば関与できるのかは以下を参照下さい:


Debianに協力するには？(http://www.debian.org/intro/help)（正規）

The Debian GNU/Linux FAQ, Chapter 13 - "Contributing to the Debian Project" (https://www.debian.org/doc/manuals/debian-faq/contributing.en.html)（準正規）

Debian Wiki, HelpDebian (http://wiki.debian.org/HelpDebian)（補足）

Debian New Member サイト (https://nm.debian.org/)（正規）

Debian Mentors FAQ (http://wiki.debian.org/DebianMentorsFaq)（補足）
開発に必要なプログラム

何はさておき、開発に必要なパッケージがきちんとインストールされていることを確認するべきです。以下のリストにはessentialまたはrequiredなパッケージが含まれていないことに注意してください。これらのパッケージは既にインストールされていることを前提としています。

以下のパッケージはDebianの標準(standard)インストール構成に含まれており、すでに(それらが依存する他のパッケージとともに)システムに含まれているはずです。しかし、念のためにaptitude show packageもしくはdpkg -s packageを用いて確認しておきましょう。

開発用システムにインストールする一番重要なパッケージは、build-essentialです。これをインストールすると、基本的なビルド環境で必要な他のパッケージを引き込むでしょう。パッケージの種類によっては、必要になるのはこれが全てかもしれません。ただ、すべてのパッケージのビルドに必須ではないにせよ、インストールしておくと便利だったり、パッケージによっては必要になったりするパッケージ群があります:

- autoconfとautomakeとautotools-dev-多くの新しいプログラムが、これらのプログラムを使って前処理される設定スクリプトやMakefileを利用しています。(詳しくはinfo autoconf、info automakeを参照) autotools-devには、特定のautoファイルを最新版に保ち、そのようなファイルを使う最も簡単な方法についてのドキュメントが含まれています。
- debhelperとdh-make-dh-makeは例示に用いられたパッケージのひな型を用意するのに必要となり、またそれはパッケージの生成するためのdebhelperツールをいくつか使います。これらを使わなくてもパッケージ作成は可能ですが、初めてパッケージを作るには利用を強くお勧めします。こうすると、取り付けやすく、以後パッケージを管理するのもずっと簡単です。(詳しくはdh_make(8), debhelper(1)を参照。)
- 標準的なdh-makeの代わりに、新しいdebmakeが使えます。機能が多く種々のパッケージング例がdebmake-doc中の文書に含まれます。
- devscripts-このパッケージはメンテナーにとって便利であると思われる有用で優れたスクリプトを含んでいますが、だからといってパッケージをビルドするために必須というわけではありません。このパッケージが推奨(Recommends)、あるいは提案(Suggests)しているパッケージは、一見の価値があります。(詳しくは/usr/share/doc/devscripts/README.gzを参照。)
- gfortran-GNUFortran95コンパイラ。あなたのプログラムがFortran言語で書かれている場合に必要です。(詳しくはgfortran(1)参照。)
- git-このパッケージは人気のあるバージョン管理システムで、大規模なプロジェクトを素早く、効率的に扱えるように設計されています。知名度の高い多数のオープンソースプロジェクトで使われており、特に有名なものとしてLinuxカーネルがあります。(詳しくはgit(1), git Manual (/usr/share/doc/git-doc/index.html)を参照。)
- gnupg-このツールを使うと、パッケージにデジタル署名を付けることができます。もしあなたが自分の作成したパッケージを他の人々に配布したいなら、これは特に重要です。また、Debianディストリビューションにあなたの作成したパッケージが含まれるようになった時には、確実にこのデジタル署名することになります。 (詳しくはgpg(1)参照。)
- gpc-GNU Pascalコンパイラ。あなたのプログラムがPascal言語で書かれている場合に必要です。ここで注目しているのはfp-compiler Free Pascalコンパイラで、こちらもまたこの作業に適しています。(詳しくはgpc(1), ppc386(1)参照。)
・patch - このとても有用なユーティリティは、(diff プログラムによって生成された) オリジナルとの差分が列挙されたファイルを読み込んでオリジナルのファイルに適用し、パッチが当てられたバージョンを作成します。詳しくは patch(1) を参照。)

・patchutils - このパッケージには、lsdiff、interdiff や filterdiff といったパッチを扱うユーティリティが含まれています。

・pbuilder - このパッケージには chroot 環境の作成や保守に使用されるプログラムが含まれます。この chroot 環境下で Debian パッケージをビルドすることで適切なビルド依存を確認して FTBFS (Fails To Build From Source) バグを回避することができます。詳しくは pbuilder(8)と pdebuild(1)を参照。

・perl - Perl は今日の UNIX 系システムにおいてもっとも使われているインタープリタ型スクリプト言語のひとつで、その強力さはしばしば「Unix のスイス軍用チェーンソー」と形容されるほどです。詳しくは perl(1)を参照。

・python - Python は Debian システムにおいてもっとも使われているもう一つのインタープリタ型スクリプト言語で、並外れたパワーや非常に明快な書式を兼ねそなえています。詳しくは python(1)を参照。

・quilt - このパッケージは、一連のパッチそれぞれの変更点を追跡して、その管理を補助するものです。パッチは簡単に当てたり、外したり、刷新したり色々することができます。詳しくは quilt(1)、/usr/share/doc/quilt/quilt.pdf.gzを参照。

・xutils-dev - ある種のプログラム（通常 X11 のために開発されたもの）は、これらのプログラムを利用して、マクロ関数の組み合わせから Makefile 群を生成します。詳しくは imake(1)、xmkmf(1)を参照。

上記の簡単な説明は、それぞれのパッケージが何をするのか紹介するだけのものです。先に進む前に make 等のよう ユーティリティをインストールされたプログラムを含むパッケージングに関連する各プログラムに付属の文書を読み、標準的な使い方だけでも理解しておいてください。いまはきっとと思われるかもしれませんが、あとになってきっと読んでよかったなあと思えることでしょう。もし後日特定の疑問を持った場合は上述で触れた文書を読み返すことをお勧めします。

1.3 開発に必要な文書

以下は、この文書と合わせて読むべきとても重要な文書です:


・developers-reference - Debian Developer's Reference (http://www.debian.org/doc/devel-manuals#devref) (Debian デベロッパリファレンス)には、例えばアーカイブの構造、パッケージ名の変更方法、パッケージの選び方、メンテナーを定めるにはどうしたらいいか、どうやって NMU をするか、バグとのつき合い方、パッケージ作成のベストプラクティス、いつどこにアップロードすればよいかなどなど、特に技術的な事柄以外のパッケージ化についてのありとあらゆる情報がここにあります (ローカルコピーの /usr/share/doc/developers-reference/developers-reference.pdfを参照。)

以下は、この文書と合わせて読むべきとても重要な文書です:

・Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html)は、Autoconf、Automake、Libtool や gettext を最も重要な構成要素としても知られる GNU Autotools として知られる GNU ビルドシステムの非常に好適な入門書です。

この文書が、上記文書の記述と矛盾している場合は、そちらが正解です。reportbug を使って maint-guide パッケージにバグレポートをしてください。

以下は、この文書と合わせて読める同様の入門書です：


1.4 相談するには

公開の場で質問をすると決心する前に、以下に示す良くできた文書を読みましょう：

- 全ての関与するパッケージの /usr/share/doc/package 中のファイル
- 全ての関与するコマンドの man command の内容
- 全ての関与するコマンドの info command の内容
- debian-mentors@lists.debian.org メーリングリストのアーカイブ (http://lists.debian.org/debian-mentors/) の内容
- debian-devel@lists.debian.org メーリングリストのアーカイブ (http://lists.debian.org/debian-devel/) の内容

site:lists.debian.org のような検索文字列を含めてドメインを制約するようなことでウェブ検索エンジンをより効率的に利用することを考えてましょう。

小さなテストパッケージを作ることは、パッケージ作成の詳細を学ぶよい方法です。他の人がどのようにパッケージを作成しているか学ぶには、既存のよく保守されているパッケージを調べるのが一番です。

入手可能な文書やウェブのリソースからは答えが見つけられない疑問が残った場合には、インタラクティブに疑問を聞くことが出来ます：

- debian-mentors@lists.debian.org メーリングリスト (http://lists.debian.org/debian-mentors/)。(初心者向けメーリングリストです。)
- debian-devel@lists.debian.org メーリングリスト (http://lists.debian.org/debian-devel/)。(上級者向けメーリングリストです。)
- #debian-mentors の様な IRC (http://www.debian.org/support#irc)。

あなたがするべき努力をした後に適切に質問すれば、より経験を持った Debian デベロッパーは喜んで助けてくれるでしょう。

バグレポート (そう、本物のバグレポートです!) を受けとったら、レポートを効率的に処理するために、Debian バグ追跡システム (http://www.debian.org/Bugs/) に入り込み、その説明文書を読む時だということを忘れましょう。Debian デベロッパーリファレンスの 5.8. ‘Handling bugs’ (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling) を読むよう、強くおすすめします。

すべてうまくやったとしても、これからはお祈りの時間です。なぜか？それは、ほんの数時間 (あるいは数日) で、世界中のユーザーがそのパッケージを使いはじめることです。もし何か致命的なエラーをやらかしていたら、膨大な数の怒った Debian ユーザーからメール爆弾を受けとることになります……なんて冗談ですが:-)。

リラックスしてパックを報告に備えてください。なぜなら、そのパッケージが Debian ポリシーやそのベストプラクティスに完全に沿うようになるまでには、やらなくてはいけないことは沢山あるのですから (繰り返しますが、詳細は正式の文書を読んでください)。頑張ってください!
Chapter 2

はじめの一歩

(できれば既存パッケージを引き取り)自分のパッケージを作成しましょう。

2.1 Debianパッケージビルドのワークフロー

アップストリームのプログラムを使ってDebianパッケージを作成する場合、Debianパッケージビルドは以下の各ステップでいくつかの特定の命名をされたファイルを生成することからなります:

- 通常圧縮されたtarフォーマットのアップストリームソフトウェアのコピーを入手します。
  - package-version.tar.gz

- debianディレクトリ下へDebian固有のパッケージ用の変更をアップストリームプログラムへ追加し、3.0(quilt)フォーマットでノンネイティブのソースパッケージを作成します。(ソースパッケージとは、Debianパッケージビルドのために用いる入力ファイルセットのこと。)
  - package-version.orig.tar.gz
  - package-version-revision.debian.tar.gz
  - package-version-revision.dsc

- Debianソースパッケージから.debフォーマット(またはDebianのインストーラーが使う.udebフォーマット)で提供される通常のインストール可能なDebianのバイナリーパッケージをビルドします。
  - package-version-revision_arch.deb

packageとversionの間の文字がtarアーカイブの名前のハイフン)がDebianパッケージファイルの名前では_ (下線)に変更されていることに注目して下さい。

Debian Policy Manualに従い、上記のファイル名中の、package部分はパッケージ名に置き換え、version部分はアップストリームバージョンに置き換え、revision部分はDebianリビジョンに置き換え、arch部分はパッケージアーキテクチャーに置き換えます。2

以下で、このアウトラインの各ステップは後述のセクションで詳細に説明します。

11.0フォーマットの古いノンネイティブのDebianパッケージに関しては、上記の代わりにpackage_version-revision.diff.gzが使われます。

2.2 プログラムの選定

おそらく、作成したいパッケージを選んだことと思います。まず最初にしなければならないことは、ディストリビューションのアーカイブにそのパッケージがすでににあるかどうかを以下を使って確認することです:

- `aptitude` コマンド
- Debian パッケージ (http://www.debian.org/distrib/packages) ウェブページ
- Debian Package Tracker (https://tracker.debian.org/) ウェブページ

もしパッケージが既に存在していたら、インストールしましょう！:-) もしそのパッケージがみなし子状態 (メンテナーが Debian QA Group (http://qa.debian.org/) に設定されていること) なら、そのパッケージを他人にとられていなければ、そのパッケージを引き取ることができるかもしれません。パッケージのメンテナーが引き取り依頼 (RFA) を出しているパッケージも引きとれます。3 パッケージの所有状態の情報源がいくつかあります:

- devscripts パッケージ中にある `wnpp-alert` コマンド
- Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp)
- Debian Bug report logs: Bugs in pseudo-package `wnpp` in `unstable` (http://bugs.debian.org/wnpp)
- Debian Packages that Need Lovin’ (http://wnpp.debian.net/)
- Browse `wnpp` bugs based on dbtags (http://wnpp-by-tags.debian.net/)

注釈ですが、Debian にはすでにほとんどの種類のプログラムが含まれていることと、Debian アーカイブにすでに含まれているパッケージの数はアップロード権限をもつユーザーの数よりもはるかに多いことに注意しておくのは重要です。従って、すでにアーカイブに含まれているパッケージへの作業は、他のデベロッパーからはるかに喜ばれ (よりスポンサーしてもらえる見込みがある) ます。貢献の仕方はいろいろあります:

- まだよく使われている、みなしのパッケージを引き取りる
- パッケージ化チーム (http://wiki.debian.org/Teams) に参加する
- よく使われているパッケージのパドに処理する
- QA もしくは NMU アップロード (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload) を準備する

もしパッケージを引き取ることができるとなら、(`apt-get source packagename` などの方法で) ソースを入手して、調べてみてください。残念ながらこの文書では、パッケージを引き取ることについて、わかりやすく説明していません。ありがたいことに、既に他者がなかったためにパッケージを準備してくれたわけですから、そのパッケージがどのように動作するのか理解することは、それほど難しくはないでしょう。とはいえ、そうした場合でもこの文書に書かれた多くのアドバイスはそのまま適用しますから、このまま読み進めていてください。

もしあなたの選んだプログラムがまだパッケージ化されていないもので、それを Debian に入れたいと決めたなら、以下のチェック項目について確認してください:

- まず、そのプログラムが機能することを理解し、ある程度の期間使用してその有用性について確認する必要があります。
- 作業中のパッケージ (http://www.debian.org/devel/wnpp) サイトを確認し、他の誰も同じパッケージに関して作業していないことを確かめてください。誰も作業していなければ、reportbug を使って ITP (Intent To Package) のバグレポートを、wnpp 疑似パッケージに送ってください。もし既に誰かが作業していたら、連絡を取りたいならそうしてください。もしその気が無いかなら、まだ誰も手をつけている他の面白いプログラムを探しましょう。

3Debian Developer’s Reference 5.9.5 “Adopting a package” (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#adopting) を参照下さい。

4とは言っても、パッケージ化する価値のある新しいプログラムはいつだって存在するでしょう。
・プログラムには、ライセンスが必須です。
  – main セクションは、Debian ポリシーにより Debian フリーソフトウェアガイドライン（DFSG [http://www.debian.org/-social_contract#guidelines]) に完全に準拠しなければなりませんし、またコンパイル・実行時に main 以外のパッケージに依存してはなりません。これが望ましいケースです。
  – contrib セクションは、DFSG に完全に準拠していないなければなりませんが、コンパイル・実行時に main があるもの以外のパッケージに依存してても構いません。
  – non-free セクションは、DFSG に準拠していない部分があるかもしれませんが、配布可能でなければなりません。
  – どうするべきかよくわからないстранで、debian-legal@lists.debian.org [http://lists.debian.org/debian-legal/] にライセンス文を送り、アドバイスを求めしてください。

・プログラムは、Debian システムにセキュリティーやメンテナンス上の懸念を招いてはいけません。
  – ちゃんと説明書きのあるプログラムで、ソースコードが理解可能なもの（つまり、難読化されていないこと）。
  – プログラムの作者に連絡をとって、パッケージ化の承諾と Debian に友好的かどうかを確認してください。何かプログラムそのものに起因する問題が発生した際に、作者にいるかではいかなることも重要なので、メンテナンスされていないソフトウェアをパッケージ化するのはやめておきましょう。
  – プログラムは root にて setuid で実行されるべきではありません。もっと言えば、どのユーザーでの setuid や setgid もするべきではありません。
  – デーモンとして動作するプログラムや、*/sbin ディレクトリに配置するプログラム、また root 特権を使ってポートを開くプログラムで無いほうが良いでしょう。

もちろんあの最後の件は単なる安全策で、setuid デーモン等で何かミスして怒り狂ったユーザーから抗議殺到という事態を招ぐことを回避するためです。パッケージ化についてもっと経験を積めば、こうしたパッケージも作れるようになるでしょう。

新規メンテナーであるあなたには、比較的簡単なパッケージで経験を積むこともお勧めし、複雑なパッケージを作成することお勧めはできません。

・単純なパッケージ、
  – 単一のバイナリーパッケージ、arch = all (壁紙グラフィックスのようなデータのコレクション)
  – 単一のバイナリーパッケージ、arch = all (POSIX で書かれた実行可能プログラム)

・ほどほど複雑なパッケージ
  – 単一のバイナリーパッケージ、arch = all (C や C++ のような言語で書かれた実行可能プログラム)
  – 複数のバイナリーパッケージ、arch = all + any (ELF バイナリの実行可能プログラム + 文書のパッケージ)
  – ソースファイルの形式が、tar.gz. や tar.bz2 でないもの
  – 配布できない内容が含まれるアップストリームソース

・高度に複雑なパッケージ
  – 他のパッケージにより利用される、インタープリタのモジュールパッケージ
  – 他のパッケージにより利用される汎用 EL ライブラリーパッケージ
  – ELF ライブラリーパッケージを含む複数バイナリーパッケージ
  – 複数のアップストリームソースからなるソースパッケージ
  – カーネルモジュールパッケージ
  – カーネルパッチパッケージ
  – 非自明なメンテナースクリプトを含むあらゆるパッケージ
高度に複雑なパッケージをパッケージすることはそれほど難しいことではありませんが、少々知識が必要です。それぞれの複雑な特徴には固有のガイダンスを探すべきです。例えば、いくつかの言語にはそれぞれのサブポリシー文書があります:

- 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/)

もう一つの古いラテンの諺にもあるように、fabricando fit faber (習うより慣れろ) です。本入門書を読みながら単純なパッケージを用いてDebianパッケージングの全ステップを練習したり色々試したりすることを非常に推薦します。以下のように作ったhello-sh-1.0.tar.gzという他愛ないアップストリームは良好なスタートポイントとなるでしょう。

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

### 2.3 プログラムの入手と検証

さて、最初にすべきことは、オリジナルのソースコードを探してダウンロードすることです。ここでは作者のホームページから、すでにソースファイルを入手したとして話を進めます。フリーソフト用プログラムのソースは、ふつう.tar.gz拡張子が付いたtar+gzipフォーマットや、.tar.bz2拡張子が付いたtar+bzip2フォーマットで提供されています。この中にはたいてい、すべてのソースが入ったpackage-versionというディレクトリがあります。

該当ソースの最新版がGitやSubversion、CVSリポジトリのようなVCSで提供されているなら、git clone、svn coやcvs coとしてソースを取得してから、--exclude-vcsオプションを使って自分でtar+gzipフォーマットに再パックする必要があります。

プログラムのソースが、他の種類のアーカイブ(例えば、.Zで終わるファイル名や、.zip6)の場合は、適切なツールでアンパックしてから再パックしてください。

DFSGに準拠しない内容とともにあなたのプログラムのソースが提供されている場合、それをアンパックし、そのような内容を削除し、dfsgを含むアップストリームバージョンに変更してリパックしましょう。

さて、gentooというX11+ファイルマネージャを例に使い説明します。

自分のホームディレクトリ以下にdebianやdeb、または何か適当な名前のサブディレクトリを作りましょう(今回の場合には~/gentoo/としても良いでしょう)。ダウンロードしたアーカイブをここにコピーし、tar xzf gentoo-0.9.12.tar.gzを実行して展開してください。この時、一見無関係に思えるようなものも含めて、警告メッセージが一切発生しないことを確認してください。アンパックする際に問題が発生しないように、人々のシステム上に展開する際に、彼らが使っているツールがこのような異常を無視したりしなかったりするので、警告メッセージがなくなるように確実にしましょう。あなたのシェルのコマンドラインは、以下のように見えているでしょうか？

5 Makefileがないことを心配しないで下さい。hello コマンドは項5.11にあるようにしてdebsrcを単純に使用したり、第3章にあるようにしてinstallターゲットがある新規のMakefileをアップストリームソースに追加してインストールできます。
6 ファイルの拡張子で足りなければ、file コマンドを使ってアーカイブ形式を判別することができます。
7 このプログラムはすでにパッケージ化されています。その最新のバージョン(http://packages.qa.debian.org/g/gentoo.html)はAutotoolsをそのビルド構造としており、バージョン0.9.12に基づく以下の例から大きく異なります。
$ mkdir ~/gentoo
$ cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xvf gentoo-0.9.12.tar.gz
$ ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz

さて、gentoo-0.9.12という別のサブディレクトリができました。そのディレクトリーに移動し、提供されているドキュメントを徹底的に読みましょう。通常は README*や INSTALL*や *.lsmや *.htmlといった名前のファイルがあります。これらの文書の中に、どうやったら正しくコンパイルできるのか、どうインストールすればよいのかといった情報が見つかるはずです（おそらく /usr/local/binにインストールするものとして説明されていますが、そうではありません。これについては項3.3を参照してください）。

パッケージ化の作業は完全にクリーンな（オリジナルのままの）ソースディレクトリー、簡単に言えば新しく展開したソースから始めるべきです。

2.4 単純なビルディメニション

単純なプログラムは普通 Makefileファイルが付属していて、単純にmakeでコンパイルできます。それらの中には同様のセルフテストを実行する make checkをサポートするプログラムもあります。インストール先ディレクトリへのインストールは一般にmake installによって実行されます。

さあ、試しにプログラムをコンパイルし、実行してみましょう。インストールや実行の際にちゃんと動作するかどうか、そして他の何かを壊していないかを確認してください。

それから、たいていの場合はmake clean(make distcleanを使うならそのほうが良いです)を実行すると、ビルド用のディレクトリーをきれいにしてくれます。さらにmake uninstallを実行すると、インストールされたファイルをすべて削除できることさえもあります。

2.5 ポピュラーなポータブルなビルディメニション

多数の自由なプログラムは、Cや C++言語で書かれています。これらの多くは、異なるプラットフォーム間で移植を可能とするために AutotoolsやCMakeを使っています。こういったツールは、Makefileやその他の必要なソースファイルを生成するのに使われます。その後、このようなプログラムは通常どおりmake; make installでビルドされます。

AutotoolsはAutoconf,Automake,Libtoolとgettextから成るGNUのビルディメニションです。configure.ac,Makefile.amやMakefile.inファイルがあれば、そういうソースであることがわかります。Autotoolsを使ったワークフローの最初の一歩は、アップストリームの作者がソースディレクトリーでautoreconf-i -fを実行し、生成されたファイルと一緒にこのソースを配布することです。

```
configure.ac-----+--> autoreconf -v --> configure
Makefile.am ----+ --> Makefile.in
src/Makefile.am ++ | --> src/Makefile.in
| +--> config.h.in
| \---> automake
|     \---> autoconf
|          \---> autoconf
automake
aclocal
aclocal.m4
autoheader
```
configure.ac や Makefile.am ファイルを編集するには、autoconf と automake についての知識が少々必要になります。info autoconf と info automake を参照してください。

Autoconf のワークフローの通常の次のステップは、この配布されているソースをユーザーが入手し、ソースディレクトリで ./configure & make を実行することで、プログラムを binary という実行可能コマンドにコンパイルします。

- Makefile.in が ./configure と Makefile.am を実行することで生成される

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

Makefileファイルにある内容の多くは変更することができます。例えばデフォルトでファイルがインストールされる場所などは、./configure --prefix=/usr とコマンドオプションを使って変更することができます。

必須ではありませんが、autoreconf -i -f をユーザーとして実行することで configure その他のファイルを更新すると、ソースの互換性が改善される場合があります。

CMakeは代替のビルドシステムです。CMakeLists.txtファイルがあれば、そういうソースだとわかります。

2.6 パッケージ名とバージョン

gentoo-0.9.12.tar.gz としてアップストリームソースが提供される場合、パッケージ名は gentoo でアップストリームバージョンは 0.9.12 となります。

この単純なアプローチは大体うまくいくのですが、Debian Policyや従来の慣習に従うようにアップストリームソースをリネームすることでパッケージ名とアップストリームバージョンを調整する必要があるかもしれません。

パッケージ名は、英小文字(a-z) と数字(0-9) と、プラス(+) とマイナス(-) 記号と、ピリオド(,) で構成するように選ばなければいけません。少なくとも 2 文字以上で、英数字で始まり、既存のパッケージと同じではいけません。30文字以内の長さにするのが望ましいです。

アップストリームのソースが test-suite のような一般的な単語をその名称として使う場合は、その内容を明示するようにリネームし、新規のパッケージ名を新たに生成することができる。

アップストリームバージョンは、英数字(0-9A-Za-z) とプラス(+)と波線(=) とピリオド(.) からなるもののみでなければいけません。それが数字(0-9)で始まらなければいけません。

アップストリームが 2.30.32 のような通常のバージョンスキームを使わずに、11Apr29 等のような日時の類や、ランダムなコードネーム字列や、バージョンの一部に VCS のハッシュ値等を使う場合、アップストリームバージョンの変更はデフォルトのバージョンフィールドが 8文字未満に制限されます。

アップストリームバージョンは、英数字(0-9A-Za-z)とプラス(+)と波線(=)とピリオド(.)からなるもののみでなければいけません。それが数字(0-9)で始まらなければいけません。13 文字以内の長さにするのが望ましいです。

アップストリームバージョンは、通常のバージョンスキームを使用する場合、バージョン文字列は、デフォルトのバージョンフィールドは 30 文字未満と制限されます。新規のパッケージ名を新たに生成する場合、その形式は、「ビルドシステム名」でなければいけません。

バージョン文字列は、dpkg(1) コマンドを以下のように使うことで比較できます。10
dh-autoreconf パッケージを用いるとこれが自動化されます。項4.4.3 を参照下さい。

aptitude のデフォルトのパッケージ名フィールド長は 30 文字。90% を越えるパッケージ名に、デフォルトは 24 文字未満で、その他のパッケージ名は、8 文字より短いです。

Debian Developer's Reference 5.1. "New packages" (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage) の通りにすれば、ITP プロセスの許容される範囲を洗い出しやすいです。

Debian リビジョンがどのように増やされるかは項8.1 を参照下さい。
$ dpkg --compare-versions ver1 op ver2

バージョンの比較ルールは以下のようにまとめられます:

• 文字列は頭から尾へと比較されます。
• 英文字は数字よりも大きいです。
• 数字は整数として比較されます。
• 英文字は ASCII コード順で比較されます。

ピリオド (.) と、プラス (+) と波線 (-) には以下のような特殊なルールがあります:

\[ 0.0 < 0.5 < 0.10 < 0.99 < 1.0 < rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0 \]

アップストリームが gentoo-0.9.12.tar.gz のプリリリースとして gentoo-0.9.12-ReleaseCandidate-99.tar.gz をリリースする時には要注意です。アップストリームソースを gentoo-0.9.12-rc99.tar.gz と名称変更してアップグレードがうまく機能するように確実にする必要があります。

### 2.7 dh_make のセットアップ

次のようにして、シェルの環境変数 $DEBEMAIL と $DEBFULLNAME を設定して、パッケージに使うあなたの email アドレスと名前を、多くの Debian メンテナンスツールに認識させましょう。\(^1\)

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

### 2.8 最初のノンネイティブ Debian パッケージ

標準的な Debian パッケージはアップストリームプログラムから作成されたノンネイティブ Debian パッケージです。アップストリームソース gentoo-0.9.12.tar.gz からノンネイティブ Debian パッケージを作りたい場合、dh_make コマンドを以下のように実行すると作成できます:

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

当然ですが、ファイル名はあなたのオリジナルのソースアーカイブの名前と置き換えてください。\(^1\) 詳細は、dh_make(8) を参照してください。

\(^1\)以下の文章はあなたが Bash をログインシェルとして使っていると仮定しています。Z シェルのような他のログインシェルを用いている場合は、それを対応する設定ファイルを使って下さい。

\(^2\)アップストリームのソースが debian ディレクトリーとその中身を提供している場合は、かわりに dh_make コマンドを --addmissing オプションをつけて実行してください。新しい 3.0 (quilt) 形式のソースはとても堅牢なので、こういったパッケージでも壊すことはありません。自分の Debian パッケージ用に、アップストリームバージョンが提供した内容を更新する必要があるかもしれません。
情報がいくつか表示されるでしょう。どんな種類のパッケージを作ろうとしているのかを尋ねられます。Gentooは単一バイナリーパッケージ－バイナリーを一つだけ生成するので、一個の .deb ファイルです－なので、s キーで最初の選択肢を選びましょう。表示された情報をチェックして、ENTER を押して確認してください。1

dh_make を実行した後、アップストリームの tarball のコピーを、親ディレクトリーに gentoo_0.9.12.orig.tar.gz として作成します。次に、それに伴ってノンネイティブ Debian ソースパッケージを debian.tar.gz として作成します:

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

この gentoo_0.9.12.orig.tar.gz ファイル名が持っている 2 つの特徴に注意してください:

- パッケージ名とバージョンは _ (アンダースコア) で区切られています。
- .tar.gz の前に .orig と言う文字列があります。

ソース中の debian ディレクトリーにたくさんのテンプレートファイルが作成されていることにも注意が必要です。

新たにデスクトップメニューに存在する .deb ファイルの数は、パッケージ作成が自動的な過程ではないことも理解しておかねばなりません。第6章のように、アップストリームソースを Debian 向けに変更する必要があります。こういった作業の後で、第7章のように正しいやり方で Debian パッケージをビルトし、第8章のようにチェックし、そして第9章のようにアップロードする必要があります。これらすべてのステップについてこれから説明します。

作業中にテンプレートファイルを間違って消した場合は、Debian パッケージのソースツリーで dh_make を--addmissingオプションつきで再度実行することで修復できます。

既存のパッケージの更新は、古いテクニックが使われていたりして、やっかいな場合があります。基本を学習中は、新規パッケージの作成にとどめてください。第8章にて、追加で説明します。

ソースファイルには項2.4 や項2.5 で述べたようなないかなるビルドシステムを含んでいる必要は無いことに注意して下さい。それは単なる画像データ集かもしれません。ファイルのインストールは debian/install(項5.11参照)のように debs 動的コンフィギュレーションファイルだけを使っても行えます。

---

1 ここでの選択肢はわずかです。s は Single binary package (単一バイナリーパッケージ)、i は Arch-Independent package (アーキテクチャー非依存パッケージ)、m は Multiple binary package (複数バイナリーパッケージ)、l は Library package (ライブラリーパッケージ)、k は Kernel module package (カーネルモジュールパッケージ)、p は Kernel patch package (カーネルパッチパッケージ)、b は cdbs です。本文書は debs パッケージを dh コマンドとともに使うことに使うことに重点を置きます。このドキュメントでは、単一バイナリーパッケージのために dh コマンドを使うことに重点を置き、アーキテクチャー非依存や複数バイナリーパッケージに関する同様の事にも触れます。cdbs パッケージは dh コマンドに代わるパッケージスクリプトのインフラを提供し、本文書では対象外です。
Chapter 3

ソースコードの変更


アップストリームのソースを修正する具体的なやり方について、何から何まで説明するにはとても紙面が足りませんが、よくあるパターンとしては大体以下のようなものです。

3.1 quilt のセットアップ

quilt プログラムはアップストリームソースに対する Debian パッケージ向け修正を記録する、基本的な方法を提供します。Debian パッケージで使うには、多少カスタマイズしたデフォルトを使うのが望ましいので、以下の行を~/.bashrc に追加して quilt コマンドのシェル入力完了機能と同様の機能を dquilt コマンドに与えましょう。

```bash
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
/usr/share/bash-completion/completions/quilt
complete -F_quilt_completion o filenames dquilt
```

~/.quiltrc-dpkg ファイルを次のように作成しましょう:

```bash
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_OPTS="-p ab --no-timestamps --no-index --color=auto"
  QUILT_REFRESH_OPTS="-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
```

quilt の使い方については、quilt(1) と /usr/share/doc/quilt/quilt.pdf.gz を参照してください。

3.2 アップストリームのバグを修正する

以下のようなアップストリームの Makefile にエラーを見つけて、install: gentoo の部分が install: gentoo-target となっているべきだったとします。
install: gentoo
  install ./gentoo $(BIN)
  install icons/* $(ICON)
  install gentoorc-example $(HOME)/.gentoorc

これを修正して、dqinit コマンドを使って fix-gentoo-target.patch として登録しましょう:

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

Makefile ファイルを次のように変更します:

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

パッチを更新して debian/patches/fix-gentoo-target.patch を作成するように dqinit に要求し、それから説明を DEP-3: Patch Tagging Guidelines (http://dep.debian.net/dists/dep3/) に準拠して追記します:

$ dqinit refresh
$ dqinit header -e
... b'' パ b'' ツ b'' チ b'' の b'' 詳 b'' 難 b''

3.3 指定場所へのファイルのインストール

通常、サードパーティーソフトウェアは自分自身を /usr/local サブディレクトリーにインストールします。Debian 上ではこれはシステム管理者 (もしくはユーザー) の個人用に予約されているため、パッケージャーは /usr/local サブディレクトリーを使って、Filesystem Hierarchy Standard (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) (FHS) に従って /usr/bin サブディレクトリーのようなシステムディレクトリーを使わなくてはなりません。

プログラムのビルドを自動化するには、通常 make(1) が使われており、make install を実行すると、(Makefile の install ターゲットに従い) 希望する場所へ直接インストールされます。Debian ではビルトドキュメントのインストール可能なパッケージを提供するために、実際のインストール先のかわりに、一時ディレクトリーの下に作成されたファイルツリーのイメージを injust にインストールするようにビルトドキュメントを変更します。

普通のプログラムインストールと Debian パッケージ作成というこれら二つの違いには、debhelper パッケージの dh_auto_configure と dh_auto_install のコマンドを使うことで特に意識せずにに対応できます:

- Makefile ファイルが GNU の慣例に準拠し、$(DESTDIR) 変数をサポートしていること。

- ソースは Filesystem Hierarchy Standard (FHS) に準拠している必要があります。

GNU autoconf を使っているプログラムは、自動的に GNU 規約に準拠するので、そのパッケージ作成はいとま簡単になります。こういった事実を考慮すると、debhelper パッケージはビルトドキュメントに立ち入った変更を加えることなく、約 90% のパッケージ作成は单一設計でも対応可能です。そのため、パッケージ作成は見かけほど複雑ではありません。

もし Makefile ファイルを変更する必要があるなら、これら $(DESTDIR) 変数をサポートするように注意しましょう。デフォルトではアンセットされているとはいえ、プログラムのインストールに使われる各ファイルパスの前

1 前に説明しましたように dh_make を実行していたら、debian/patches ディレクトリーは応じて存在しているはずです。この操作例では、既存のパッケージを更新している場合に合わせて作成しています。

2 GNU Coding Standards: 7.2.4 DESTDIR: Support for Staged Installs (http://www.gnu.org/prep/standards/html_node/DESTDIR.html#DESTDIR) を参照下さい。
に $(DESTDIR) 变数は付与されます。パッケージ作成スクリプトは $(DESTDIR) を一時ディレクトリーにセットします。

単一バイナリーパッケージを生成するソースパッケージでは、debian/package が dh_auto_install コマンドが使う一時ディレクトリーとして指定されます。3 一時ディレクトリーに含まれているものはすべて、ユーザーがあなたのパッケージをインストールするのに、ユーザーのシステムにインストールされます。唯一の違いは、dpkg はファイルをあなたの作業ディレクトリーではなくルートディレクトリーからの相対パスにインストールするということです。

パッケージの作成時、あなたのプログラムは debian/package にインストールされますが、.deb パッケージからルートディレクトリードインストールされた場合も正しく動かす必要があることを覚えておいてください。こうするためには、ビルドシステムがパッケージファイルの中のファイル中に/home/me/deb/package-version/usr/share/といった文字列をハードコードしないようにしなければなりません。

gentoo の Makefile で該当する部分はこれです⁴:

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

ファイルが /usr/local 以下にインストールされるようになっていることがわかります。上記説明にあるように、そのディレクトリーハイパラルキーは Debian 上のローカルユーザーよりも予約されているので、これらのパスを以下のように変更してください:

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

バイナリー、アイコン、文書など、それぞれのファイルを保存すべき正確な場所については、Filesystem Hierarchy Standard (FHS) 中に規定されています。全体に目を通して、あなたのパッケージに該当する箇所を読むことをお勧めします。

そういうわけで、実行可能なバイナリーは /usr/local/bin ではなく /usr/bin へインストールしなければなりません。マニュアルページは /usr/local/man/man1 の代わりに /usr/share/man/man1 へインストールする必要があります。ここで gentoo の Makefile には、マニュアルページに関する記述がまったく無いことに注意してください。Debian ポリシーでは、すべてのプログラムがそれぞれマニュアルを用意しなければならないと定めていますから、後で gentoo のマニュアルを作成して、それを /usr/share/man/man1 以下ヘインストールすることにします。

プログラムの中には、このようなパスを定義するための Makefile 变数を使っているものもあります。このような場合、C のソースそのものをいじって、指定された場所を使うように修正しなければなりません。でもどこを探し、何を確認すればよいのでしょうか？以下のコマンドを実行すれば該当箇所を見つけることができます:

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

grep がソースツリーを再帰的に検索し、該当箇所を見つけたらそのファイルの名前と検索対象の文字列と行番号を表示します。

それらのファイルを編集し、該当行の usr/local/lib を /usr/lib に置き換えてください。これは以下の様にすると自動化出来ます:

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

③複数バイナリーパッケージを生成するソースパッケージでは、debian/tmp を dh_auto_install コマンドが使う一時ディレクトリーとして指定しますが、debian/tmp の中身を debian/package-1 や debian/package-2 一時ディレクトリーへと、debian/package-1/install または debian/package-2/install ファイルによる指定に従い dh_install コマンドが分配することで複数バイナリー * .deb パッケージを作成されます。

④これは Makefile ファイルがこうなっているべきである、ということを示すための例にすぎません。Makefile ファイルが ./configure コマンドで作成されているなら、この手の Makefile を修正する正しい方法は、./auto_config コマンドに --prefix=/usr を含むデフォルトのオプションを与えて、./configure コマンドを実行させることです。
こうする代わりに、各置換を確認したい場合は、これは以下のようにインタラクティブにできます。

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

修正が終わったら、install ターゲットを探しましょう (install: で始まる行を探してください。この方法でたい
ていうまくいきます)。Makefile の先頭で定義されているものを除いて、ディレクトリーへの参照をすべて変更し
てください。

修正前は、gentoo の install ターゲットはこうなっています:

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

dquilt コマンドを使って、debian/patches/install.patch としてアップストリームバグを修正して記録しま
しょう。

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

Debian パッケージ用に、これをエディタで次のように変更します:

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

お気づきになったでしょうかが、変更後はこのルールの他のコマンドより前に install -d コマンドが追加されて
います。make install を実行するようなシステムなら /usr/local/bin やその他のディレクトリーはたいがい
既に存在していますから、もともとの Makefile ではこのコマンドは使われていませんでした。しかし、私
たちは独自に空っぽの (あるいはまだ存在されていない) ディレクトリーにインストールするわけですから、こ
れらの各ディレクトリーを毎回作成する必要があります。

ルールの最後には、アップストリームの作者が省略することの多い付加的な資料のインストールなど、他の作業を
追加することもできます:

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

しっかりチェックをして、何も問題がないようであれば、dquilt でパッチを更新して debian/patches/install.
patch を作成し、パッチの説明を追記してください:

```
$ dquilt refresh
$ dquilt header -e
... b''バ b''バ ッ b''バ ッ b'' チ b''バ ッ の b''バ ッ 詳 b''バ ッ 細 b''
```

これで、一連のパッチができました。

1. アップストリームのパック修正: debian/patches/fix-gentoo-target.patch
2. Debian 固有のパッケージ上の変更: debian/patches/install.patch

debian/patches/fix-gentoo-target.patch のような、特に Debian パッケージだけに限定されない変更を行
った場合、その内容をアップストリームのメンテナーに報告するようにしてください。そうすれば、プログラム
の次版に反映してもらうことができ、他のすべての利用者にとっても有益な結果をもたらすことになります。また、
あなたの修正を送る前に、Debian や Linux—あるいは Unix でさえも!—に特化した修正にせずに、移植性をもたせる
ことも忘れていけないでしょう。そうすれば、あなたの変更はずっと採用されやすくなります。

アップストリームの作者へ debian/* ファイルを送らなくてもよいことに注意してください。
3.4 ライブラリーの相違

よくある問題がもう一つあります。ライブラリーはしばしばプラットフォームごとに異なります。例えば、MakefileはDebianシステム上に存在しないライブラリーへの参照を含んでいるかもしれません。その場合には、Debian上に存在する互換ライブラリーを指すように変更し、同じ目的を果たすようにしてやらなければなりません。

あなたのプログラムのMakefile(もしくはMakefile.in)が以下のようになっていると仮定しましょう。

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

fooライブラリーが存在しないためにあなたのプログラムがコンパイルしないで、foo2ライブラリーがその等価をDebianシステム上で提供する場合、fooをfoo2に變更するdebian/patches/foo2.patchとしてこのビルド問題を解決できます。

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

fooライブラリーからfoo2ライブラリーへのAPI変更があった場合、新しいAPIに合わせてソースコードへの必要な変更を加える必要があります。

---

3fooライブラリーからfoo2ライブラリーへのAPI変更があった場合、新しいAPIに合わせてソースコードへの必要な変更を加える必要があります。
Chapter 4

debian/ ディレクトリー以下に無くてはならないファイル


プログラムのソースディレクトリーの中に debian という名前の新しいディレクトリーがつくられています。このディレクトリー内には、パッケージの挙動をカスタマイズするため編集するべき多くのファイルがあります。特に、control と changelog と copyright と rules は、すべてのパッケージになくてはならないファイルです。

4.1 control


以下は、dh_make が生成した control ファイルの雛型です:

```
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>
```

(行番号は筆者による)

1-7 行目は、ソースパッケージの管理情報です。9-13 行目は、バイナリパッケージの管理情報です。

1 行目は、ソースパッケージ名です。

2 行目は、パッケージが所属するディストリビューション内のセクションです。

1）自明な場合、本章では debian ディレクトリー中のファイルは、前に付く debian/ を省略し簡明に表記しています。
ご存知のように、Debian アーカイブは main (完全にフリーなソフトウェア)、non-free (本当にフリーではないソフトウェア)、contrib (フリーだが non-free ソフトウェアに依存するソフトウェア) という複数エリアに分かれています。さらにそれらは、大まかなカテゴリーよりのセクションに分類されています。例えば、管理者専用のプログラムは admin、プログラムツールは devel、文書作成関連は doc、ライブラリは Libs、メールリーダーやメールデーモンは mail、ネットワーク関連のアプリケーションやデーモンは net、分類ができない X11 用のプログラ

このページは x11 に変更してみましょう。（省略時は main/がデフォルトとして設定されます）

3 行目は、ユーザーが当パッケージをインストールする重要度を示しています。

- required、important、standard のパッケージと競合しない新規のパッケージの場合は、optional で問題ないでしょう。

セクション (Section) と優先度 (Priority) は aptitude のようなフロントエンドがパッケージをソートする際に、デフォルトを選択する際に利用されます。Debian にアップロードしたパッケージのこれらの値は、アーカイブによってオーバーライドされることがあります。これの場合は電子メールによって通知されます。

このパッケージは通常の優先度で、競合もないので、optional にしましょう。

4 行目は、メンテナーの名前と電子メールアドレスです。バグ追跡システムは、このフィールドに記載された宛先からバグ報告を送信するので、このフィールドは有効な電子メールの To ヘッダーを含むようにして下さい。コンマ「、」、アンド記号「&」、丸括弧「()」は使用しないでください。

5 行目の Build-Depends フィールドは、新規パッケージのビルトに必要なパッケージのリストです。必要であれば、Build-Depends-Indep フィールドをここに追加できます。 gcc や make のような build-essential に含まれるパッケージは明示無くとも含まれています。他のツールがパッケージをビルトするのに必要な場合は、このフィールドに追加しましょう。複数記載する場合は、コンマで区切ります。このフィールドの書式については、後述のバイナリパッケージ依存関係でこれらの行のシナクスに関してもう少し詳しく説明します。

- debian/rules 使用し、dh コマンドでパッケージングされたパッケージは、clean ターゲットに関する Debian ポリシーを満たすために、Build-Depends フィールドに debhelper (>=9) を記載しなければなりません。

- Architecture: any のバイナリパッケージを含むソースパッケージはオートビルダーによってリビルトされます。オートビルダーは debian/rules build を実行します。その際に、Build-Depends フィールド (項 6.2 を参照) に列挙されたパッケージしかインストールしないので、Build-Depends フィールドには事実上必要なパッケージ全てを列挙しなければなりません。Build-Depends-indep はあまり使われません。

バイナリパッケージが全て Architecture: all のソースパッケージでは、clean ターゲットに関する Debian ポリシーを満たすために Build-Depends フィールドにすでに記載したパッケージ以外で必要なパッケージは、Build-Depends-Indep フィールドに記載することもできます。

どちらのフィールドを使るべきかわからなければ、Build-Depends にしておきましょう。

以下のコマンドを使えば、新規のパッケージをビルトするためにどのパッケージが必要かを調べることができます:

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

```
/usr/bin/foo の正確なビルト依存パッケージを手動でみつけるには、
```

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

を実行し、表示された各ライブラリー（例えば libfoo.so.6 の場合）について、


3Debian Policy Manual, 2.5 "Priorities" (http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities) を参照下さい。

4*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) を参照下さい。

$ dpkg -S libfoo.so.6

を実行します。Build-Depends の項目に、各ライブラリーの-dev バージョンを採用します。このために ldd を使用すると、間接的な依存も報告し、過度のビルド依存問題を引き起こします。

genoo パッケージをビルドするには xlibs-dev、libgtk1.2-dev、libglib1.2-dev が必要なので、debhelper の後に記述しましょう。

6 行目は、パッケージが準拠する Debian Policy Manual (http://www.debian.org/doc/devel-manuals#policy) のバージョンです。これは、あなたがパッケージ作成の際に参照したポリシーマニュアルのバージョンです。

7 行目にはソフトウェアのアップストリームホームページ URL を記載できます。

9 行目はバイナリーパッケージの名前です。ソースパッケージと同名にするのが通例ですが、そうでなくてもかまいません。

10 行目にはバイナリーパッケージがコンパイルされる対象のアーキテクチャーを記載します。この値はバイナリーパッケージのタイプによって通常以下の 2 つのどちらかです:6

- Architecture: any
  - 生成されるバイナリーパッケージが通常コンパイルされたマシンコードからなるアーキテクチャー依存パッケージである。

- Architecture: all
  - 生成されたバイナリーパッケージは、通常テキストやイメージやインタープリター言語のスクリプトからなる、アーキテクチャー依存の無いパッケージである。

10 行目が C で書かれているのでこのままにしておきます。dpkg-gencontrol(1) がソースパッケージがコンパイルされたマシンに合わせた適正なアーキテクチャーの値で埋めてくれます。

特定のアーキテクチャーに依存しない(例えば、シェルや Perl スクリプト、文書) パッケージであれば、パッケージをビルドする際に、これを all に変更し、binary-arch に代え binary-indep を使って後述の項 4.4 を読んでください。

11 行目からは Debian のパッケージシステムが強力なことがわかります。パッケージは様々な形で相互に関係することができます。Depends の他には、Recommends、Suggests、Pre-Depends、Breaks、Conflicts、Provides、Replaces などがあります。

パッケージ管理ツールは通常このような関係を処理するとき同様の動作をします。そうでない場合については、後から説明します。(dpkg(8)、dselect(8)、apt(8)、aptitude(1) 等を参照してください。)

パッケージの依存関係を単純化し以下に説明します:7

- Depends (依存)
  依存しているパッケージがインストールされない限り、パッケージはインストールされません。あなたのプログラムが特定のパッケージ無しでは動かない(または深刻な破損を引き起こす) 場合はこれを使います。

- Recommends (推奨)
  厳密には必須ではないけれど通常一緒に使われるようなパッケージの指定にこれを利用します。あなたのプログラ
  ムをユーザーがインストールする時、全てのフロントエンドが推奨パッケージも一緒にインストールするかをき
  っと確認します。aptitude や apt-get の場合は、推奨パッケージもデフォルトで一緒にインストールします。(ユー
  ザーはこの挙動を無効化できます。) dpkg はこのフィールドを無視します。

---


• Suggests（提案）

必須ではないが、一緒に使用すると便利なパッケージの指定にこれを用います。あなたのプログラムをユーザーやインストールする時、フロントエンドが提案パッケージも一緒にインストールするかきっと確認します。aptitudeは提案パッケージと一緒にインストールするように変更することが可能です。デフォルトではありません。dpkgとapt-getはこのフィールドを無視します。

• Pre-Depends（事前依存）

これはDependsよりも強い関係を示します。パッケージは先行依存のパッケージがあらかじめインストールされ、かつ適切に設定されていない限りインストールされません。これらをメーリングリストdebact@lists.debian.org（http://lists.debian.org/debian-devel/）で議論を尽くした上で、とても慎重に扱うべきです。	

• Conflicts（競合）

競合しているパッケージが削除されないように、パッケージはインストールされません。あなたのプログラムが特定のパッケージと一緒だと動かない（または深刻な破壊の原因になる恐れがある）場合はこれを使います。

• Breaks（破壊）

パッケージがインストールされると、全てのリストされたパッケージを破壊します。通常、Breaksの項目は条件よりも古いバージョンに対して指定します。通常、上位パッケージマネジメントツールを用い、記載されたパッケージをアップグレードし解決します。

• Provides（提供）

パッケージによっては、選択の余地があるために、仮想パッケージ名が定義されています。仮想パッケージ名の一覧はvirtual-package-names-list.txt.gz(http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt)があります。あなたのプログラムが既存の仮想パッケージの機能を提供する場合には、これを使います。

• Replaces（置換）

あなたのプログラムが別パッケージのファイルを置き換えた場合、パッケージを完全に置き換えてしまう場合（この場合はConflictsも一緒に指定してください）この指定を使います。ここで指定されたパッケージに含まれるファイルはあなたのパッケージのファイルによって上書きされます。

これらのフィールドは共通の書式で記述します。指定したいパッケージ名をコンマで区切って並べます。もし選択肢があれば、それらのパッケージ名を縦棒|で区切って並べます。

これらフィールドは、共通の書式で記述します。指定したいパッケージ名をコンマで区切って並べます。もし選択肢があれれば、それらのパッケージ名を縦棒|で区切って並べます。

これらフィールドは、共通の書式で記述します。指定したいパッケージ名をコンマで区切って並べます。もし選択肢があれば、それらのパッケージ名を縦棒|で区切って並べます。

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

知っておくべき最後の特徴は${shlibs:Depends}や${perl:Depends}や${misc:Depends}等です。

dh_shlibdeps(1)は、バイナリパッケージのライブラリ依存関係を計算します。それは各バイナリパッケージの制約はパッケージ名の後に丸カッコの中に以下の関係式を続けバージョン番号を指定しリストします。使用できる関係式は、< > <= >= = < >で、それぞれ'指定されたものより古いバージョンのみ'、'指定されたバージョン以降'（指定のバージョンを含まない）、'指定されたものより新しいバージョンのみ'を意味します。例えば、
DH_GenControl为你创建了一个文件。这文件包含有关各源代码包的依赖信息。

$\{shlibs:Depends\}$, $\{perl:Depends\}$, $\{misc:Depends\}$ 等字段用于描述其在 DEBIAN/control 文件。

以下为修正后的 DebianAliothGitService 的源代码。假设设置如下所示。

```
$shlibs:Depends
$perl:Depends
$misc:Depends

```

8 行是介绍页面的 URL。这是 http://www.obsession.se/gentoo/。

9 行是添加的两个栏目 X 文件管理器。

10 行是引入 VCS 系统的配置。

gentoo 是 VCS 系统的配置。

11 行是为 Debian 打包提供帮助。

12 行是全面支持的，两个栏目 X 文件管理器。

13 行是详细的说明。这是文件管理器的详细说明。

14 行是对版本控制系统的介绍。

15 行是创建文件管理器。

16 行是为 X Window System 创建文件管理器。

17 行是用户 (几乎) 全部配置。

18 行是相对容易的，因为它写在 XML 格式。

19 行是用户 (几乎) 全部配置。

20 行是强大的配置。

21 行是为配置文件的设计。

22 行是为配置文件的设计。

23 行是为配置文件的设计。

24 行是为配置文件的设计。

25 行是为配置文件的设计。

26 行是为配置文件的设计。

27 行是为配置文件的设计。

28 行是为配置文件的设计。

(行号由作者提供)

### 4.2 copyright

这个文件为上游源代码的属性和版本控制系统的相关著作版权、许可证及等信息提供了记录。

该内容是 Debian Policy Manual, 12.5 "Copyright information" (http://www.debian.org/doc/debian-policy/ch-docs.html#s-

上文的说明是英文原版。

上文的说明是英文原版。
copyrightfile) に規定され、DEP-5: Machine-parseable debian/copyright (http://dep.debian.net/deps/dep5/) がそのフォーマットのガイドラインを提供しています。

dh_make は copyright ファイルのテンプレートを作成します。GPL-2 でリリースされた gentoo パッケージのテンプレートを入手するには、--copyright gpl2 オプションを使用します。

You must fill in missing information to complete this file, such as the place you got the package from, the actual copyright notice, and the license. For certain common free software licenses (GNU GPL-1, GNU GPL-2, GNU GPL-3, LGPL-2, LGPL-2.1, LGPL-3, GNU FDL-1.2, GNU FDL-1.3, Apache-2.0, 3-Clause BSD, CC0-1.0, MPL-1.1, MPL-2.0 or the Artistic license), you can just refer to the appropriate file in the /usr/share/common-licenses/ directory that exists on every Debian system. Otherwise, you must include the complete license.

つまり、gentoo パッケージの copyright ファイルは以下のようになります:

```markdown
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’.
```

(行番号は筆者による)

ftpmaster により提供され debian-devel-announce に投稿された手順書 http://lists.debian.org/debian-devel-announce/2006/-03/msg00023.html に従って下さい。

### 4.3 changelog

これは必須のファイルで、Debian Policy Manual, 4.4 "debian/changelog" (http://www.debian.org/doc/debian-policy/changesource.html#s-dpkgchangelog) で既定された特別な書式となっています。この書式は、dpkg やその他のプログラムがパッケージのバージョン番号、リビジョン、ディストリビューション、緊急度 (urgency) を識別するために利用します。
あなたが行なったすべての変更をきちんと記載しておくことは良いことであり、その意味でこのファイルはまた、パッケージメンテナーであるあなたにとっても重要なものです。パッケージをダウンロードした人は、このファイルを見ることで、このパッケージに関する未解決の問題があるかどうかを知ることができます。このファイルはバイナリーパッケージ中に/usr/share/doc/gentoo/changelog.Debian.gzとして保存されます。

**dh_make**がデフォルトを生成し、以下のようになっています:

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

(行番号は筆者による)

1行目はパッケージ名、バージョン、ディストリビューション、そして緊急度です。ここに書くパッケージ名はソースパッケージの名前と一致していなければなりません。またディストリビューションは unstable で、緊急度は low より高くしてはいけません。:-)

3-5行目はログ項目で、このパッケージのリビジョンで行われた変更を記述します(アップストリームプログラムそのものの変更点ではありません - その目的のためには、アップストリーム作者によって作成され、/usr/share/doc/gentoo/changelog.gzとしてインストールされる専用のファイルが存在しています)。ITP(Intent To Package)パグレポート番号を12345と仮定しましょう。新しい行は * (アスタリスク)で始まる最初の行の直前に挿入します。この操作はdch(1)を使うと便利ですが、テキストエディタを使って実行してももちろん構いません。

パッケージの完成前にパッケージが間違ってアップロードされることを防ぐために、ディストリビューションの値を無効な値 UNRELEASED に変更するよう推奨します。

最終的にこんなふうになります:

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

(行番号は筆者による)

すべての変更に満足しそれらをchangelog 記録した時点で、ディストリビューションの値を UNRELEASED からターゲットディストリビューションの値 unstable (もしくは場合に依っては experimental) へと変更すべきです。

changelog の更新については、第8章で詳しく説明します。

### 4.4 rules

さて、今度は dpkg-buildpackage(1) が実際にパッケージを作成するために使うルールについて見ていきましょう。このファイルは、もうひとつの Makefile といった存在ですが、アップストリームソースに含まれるそれとは違います。Debianディレクトリーに含まれる他のファイルとは異なり、このファイルには実行可能属性が付与されていません。

---

10もし dch -r コマンドを使ってこの最終変更をする場合には、エディターにより changelog ファイルを明示的に保存して下さい。
4.4.1 rules ファイルのターゲット

他の Makefile 同様、全ての rules ファイルはいくつかのルールから成り立っていて、そのそれぞれにターゲットと実行方法が規定されます。11 新規のルールは最初のカラムにそのターゲット宣言をすることで始まります。それに続く TAB コード (ASCII9) で始まる数行はそのレシピを規定します。行と # (ハッシュ) で始まる行はコメントとして扱われ無視されます。12

実行したいルールは、そのターゲット名をコマンドラインの引数として実行します。例えば、debian/rules build や fakeroot make -f debian/rules binary は、それぞれ build や binary ターゲットのルールを実行します。

ターゲットについて簡単に説明します:

• clean ターゲット: ビルドツリー内にある、生成されたりコンパイルされたり役に立たなかったりする全てのファイルをクリーンします。(必須)
• build ターゲット: ソースをビルドして、ビルドツリー内にコンパイルしたプログラムと書式に落とし込んだドキュメントをビルドします。(必須)
• build-arch ターゲット: ソースをビルドして、ビルドツリー内にアーキテクチャーに依存したコンパイルしたプログラムをビルドします。(必須)
• build-indep ターゲット: ソースをビルドして、ビルドツリー内にアーキテクチャーに依存しない書式に落とし込んだドキュメントをビルドします。(必須)
• install ターゲット: debian ディレクトリー以下にある各種ライブラリパッケージのファイルツリーにファイルをインストールします。定義されている場合は、binary* ターゲットは実質的にこのターゲットに依存します。(任意)
• binary ターゲット: 全てのバイナリパッケージを作ります。(実質的には binary-arch と binary-indep の組み合わせ)(必須)13
• binary-arch ターゲット: 親ディレクトリーにアーキテクチャーに依存したバイナリパッケージ (Architecture: any) を作ります。(必須)14
• binary-indep ターゲット: 親ディレクトリーにアーキテクチャーに依存しないパッケージ (Architecture: all) を作ります。(必須)15
• get-orig-source ターゲット: アップストリームアーカイブのサイトから最新のバージョンのオリジナルソースパッケージを取得します。(任意)

今は少々圧倒されているかもしれませんが、dh_make がデフォルトとして作成する rules ファイルを調べると、状況はとても簡単です。

4.4.2 デフォルトの rules ファイル

最新の dh_make は dh コマンドでシンプルかつパワフルなrules ファイルを作ってくれます:

11Debian Reference, 12.2 "Make" (http://www.debian.org/doc/manuals/debian-reference/ch12#_make) から Makefile の書き方を学び始められます。
13このターゲットは例えば、項6.1 などで、dpkg-buildpackage が使用します。
14このターゲットは例えば、項6.2 などで、dpkg-buildpackage -B が使用します。
15このターゲットは、dpkg-buildpackage -A が使用します。
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=+
8 # see ENVIRONMENT in dpkg-buildflags(1)
9 # package maintainers to append CFLAGS
10 #export DEB_CFLAGS_MAINT_APPEND = -Wall -pedantic
11 # package maintainers to append LDFLAGS
12 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,-as-needed
13
14 # debian/rules clean
15
dh_testdir
dh_auto_clean
dh_clean
16
17 # debian/rules build
18
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
19
20 # fakeroot debian/rules binary
21
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
22
(筆者が行番号を追加しコメントは一部削除した。実際のrulesファイルでは、行頭の空白TABコードです。)

1行目はシェルやパールスクリプトでお馴染みの表現です。オペレーティングシステムに/usr/bin/makeで処理するように指示しています。

4行目のコメントを外しDH_VERBOSE変数を1に設定すれば、dhコマンドがどのdh_*コマンドを実行しているかを出力するようにできます。必要であればここに、export DH_OPTIONS=-vという行を追加すれば、dh_*コマンドが、dh_*によって実行されたコマンドを出力します。この単純なrulesファイルが影で何をしているのかを理解し、その問題デバッグの際の助けとなるでしょう。この新しいdhがdebhelperツールの中核から設計され、あなたに対して一切隠し事をしません。

16と17行目は、パターンルールを用いて非明示的ルールで全てが行われる場所です。パーセント記号「%」は「いかなるターゲット」を意味し、ターゲットの名前を引数にdhという単一行程を実行します。

以下の例は、自動的にpythonサポートコマンドが起動するのを避けるために、あなたのdebian/compatには以下の値が入っていると仮定しています。
DH_testroot
DH_prep
DH_installdirs
DH_auto_install
DH_install
DH_installdocs
DH_installchangelogs
DH_installexamples
DH_installman
DH_installcatalogs
DH_installcron
DH_installdebcnf
DH_installmacsen
DH_installifupdown
DH_installinfo
DH_installinit
DH_installmenu
DH_installmime
DH_installmodules
DH_installlogcheck
DH_installlogrotate
DH_installpam
DH_installppp
DH_installudev
DH_installwm
DH_installxfonts
DH_bugfiles
DH_lintian
DH_gconf
DH_icons
DH_perl
DH_usrllocal
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はfakeroot dh binary-archを実行します。fakeroot dh binaryの全てのコマンドに -a オプションをつけた場合と同じことを行います。

• fakeroot debian/rules binary-indepはfakeroot dh binary-indepを実行します。同様のコマンドは fakeroot dh binaryですが、dh_strip, dh_makeshlibs, dh_shlibdeps は実行せず、残りのコマンドには -i オプションを付加して実行します。

DH_* は、名前からその機能がわかるようなものばかりです。dh_auto_clean は、Makefile に distclean ターゲットがあれば以下のコマンドも通常実行します。

19 dh_* スクリプトが、実際に何をもって、どのようなオプションがあるのかを知りたい場合は、debhelper のマニュアルにある該当ページを参照してください。
20 これらのコマンドは、dh_auto_build --list を実行するとリストされる setup.py のような他のビルト環境もサポートします。
21 実際には、Makefile 中の distclean, realclean, clean のうち、最初に利用可能なものを探し実行します。
make distclean

• **dh_auto_configure** は、./configure があれば以下のコマンドを通常実行します。(読みやすくするために引数は省略しました)

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

• **dh_auto_build** は、Makefile があれば、その最初のターゲットをビルドするために、以下のコマンドを通常実行します。

  ```
  make
  ```

• **dh_auto_test** は、Makefile 中に test ターゲットがあれば、以下を通常実行します。22

  ```
  make test
  ```

• **dh_auto_install** は、Makefile 中に install ターゲットがあれば、以下のコマンドを通常実行します。(読みやすくするために畳み込みました)。

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

**fakeroot** コマンドを必要とするターゲットは **dh_testroot** を含みます。このコマンドは、ルートのぶりをしなければエラーで終了します。

dh_make によって作成された rules ファイルについて理解すべきことは、これは単なる提案ということです。もっと複雑なパッケージ以外のほぼ全てに有効ですが、必要に応じてカスタマイズをすることを避けるわけではありません。install は、必須ターゲットではありませんがサポートはされています。fakeroot dh install は fakeroot dh binary のように振る舞いますが、dh_fixperms の後で停止します。

### 4.4.3 rules ファイルのカスタマイズ

新しい dh コマンドで作成されたrules ファイルをカスタマイズする方法は何通りもあります。dh $@ コマンドは以下の方法でカスタマイズできます: 23

• **dh_python2** コマンドのサポートを追加します。(Python に最適の選択。)24
  - python パッケージを Build-Depends に含めます。
  - dh $@ --with python2 を使用します。
  - これは python フレームワークを使用して Python モジュールを取り扱います。

• **dh_pysupport** コマンドのサポートを追加します。(非推奨)
  - python-support を Build-Depends に含めます。
  - dh $@ --with pysupport を使用します。
  - これで python-support フレームワークを使用して Python モジュールを利用できます。

• **dh_pycentral** コマンドのサポートを追加します。(非推奨)
  - Build-Depends に、python-central パッケージを含めます。

22 Makefile 中の test か check のうち、最初に利用可能なものを見つけ実行します。
23 もし、パッケージが /usr/share/perl5/Debian/Debhelper/Sequence/custom_name.pm ファイルをインストールする場合、そのカスタマイズの機能を dh $@ --with custom-name で有効にしなければなりません。
24 dh_pysupport や dh_pycentral コマンドよりも dh_python2 コマンドが好まれます。dh_python コマンドは使用しないでください。
代わりに `dh $@ --with python-central` を使用します。
これで `dh_pysupport` コマンドも無効化されます。
これで `python-central` フレームワークを使用して Python モジュールを利用できます。

- `dh_installtex` コマンドのサポートを追加します。
  - Build-Depends に、`tex-common` パッケージを含めます。
  - 代わりに `dh $@ --with tex` を使用します。
  - これで、TeX による Type1 フォント、ハイフネーションパターン、またはフォーマットが登録されます。

- `dh_quilt_patch` と `dh_quilt_unpatch` コマンドのサポートを追加します。
  - Build-Depends に、`quilt` パッケージを含めます。
  - 代わりに `dh $@ --with quilt` を使用します。
  - 1.0 フォーマットのソースパッケージの `debian/patches` ディレクトリー内にあるファイルを用いて、アップストリームソースにパッチを当てたり外したりできます。
  - もし新規の3.0 (quilt) ソースパッケージフォーマットを使用している場合、これは不要です。

- `dh_dkms` コマンドのサポートを追加します。
  - Build-Depends に、`dkms` パッケージを含めます。
  - 代わりに `dh $@ --with dkms` を使用します。
  - カーネルモジュールパッケージによる DKMS の使用を正しく処理します。

- `dh_autotools-dev_updateconfig` と `dh_autotools-dev_restoreconfig` コマンドのサポートを追加します。
  - Build-Depends に、`autotools-dev` パッケージを含めます。
  - 代わりに `dh $@ --with autotools-dev` を使用します。
  - これで `config.sub` と `config.guess` をアップデートおよびレストアします。

- `dh_autoreconf` と `dh_autoreconf_clean` コマンドのサポートを追加します。
  - Build-Depends に、`dh-autoreconf` パッケージを含めます。
  - 代わりに `dh $@ --with autoreconf` を使用します。
  - これは、ビルド後に GNU ビルドシステムのファイルのアップデートおよびレストアを行います。

- `dh_girepository` コマンドのサポートを追加します。
  - Build-Depends に、`gobject-introspection` パッケージを含めます。
  - 代わりに `dh $@ --with gir` を使用します。
  - これは `GObject` イントロスペクションデータを提供しているパッケージの依存関係を計算し、パッケージ依存関係用に `${gir:Depends}` 代替変数を生成します。

- `bash` 補完機能のサポートを追加します。
  - Build-Depends に、`bash-completion` パッケージを含めます。
  - 代わりに `dh $@ --with bash-completion` を使用します。
  - このコマンドを使用すると、`bash` 補完機能から、`debian/package.bash-completion` の設定を使うことができるようになります。
新しい dh コマンドにより起動される多くの dh_* コマンドは、debian ディレクトリー内にある対応する設定ファイルによりカスタマイズすることが可能です。そのような機能のカスタマイズ方法については、第5章とコマンドごとの manpage を参照してください。

dh コマンドによって呼び出される dh_* コマンドの中には、特定の引数で実行したり、それらを追加のコマンドとともに実行したり、スキップしたりする必要があることがあります。そのような場合は、変更したい dh_foo コマンドについて、override_dh_foo ターゲットを rules ファイルに追記してください。簡単に説明すると、このターゲットはかわりにこのコマンドを使用するという意味です。2⁵

ここでは簡単に説明を行いましたが、通常以外のケースを処理するため、dh_auto_* コマンドは、もっと複雑なことを実行することを覚えておいてください。そのため、override_dh_auto_clean ターゲット以外は、override_dh_* ターゲットを使用して、簡素化された別のコマンドで代用するのは感心しません。debhelper の賢い機能を骨抜きにしてしまうからです。

例えば、最近の Autotools を用いた gentoo パッケージに関して、その設定情報を通常の /etc ディレクトリーではなく、/etc/gentoo ディレクトリーに置きたい場合には、dh_auto_configure コマンドが ./configure コマンドで代用するのではありません。debhelper の賢い機能を骨抜きにしてしまうからです。

例えば、最近の Autotools を用いた gentoo パッケージに関して、その設定情報を通常の /etc ディレクトリーではなく、/etc/gentoo ディレクトリーに置きたい場合には、dh_auto_configure コマンドが ./configure コマンドで代用するのではありません。debhelper の賢い機能を骨抜きにしてしまうからです。

\begin{verbatim}
override_dh_auto_configure:
    dh_auto_configure -- --sysconfig=/etc/gentoo
\end{verbatim}

以下の引数は、自動実行されるプログラムの引数に付け足されます。dh_auto_configure コマンドは、引数 --sysconfig のみをオーバーライドしその他の良く配慮された ./configure 引数には触れないため、./configure コマンドをここのように使うより優れている。

もしも gentoo のソースをビルドするためにその Makefile に build ターゲットを指定する必要がある場合、これを有効とするため、2⁶ override_dh_auto_build ターゲットを作らなければなりません。

\begin{verbatim}
override_dh_auto_build:
    dh_auto_build --build
\end{verbatim}

dh_auto_build コマンドのデフォルトで与えられた引数すべてに build 引数を加えたものとともに、$(MAKE) を確実に実行します。

もし、Debian パッケージのためにソースをクリーンするのに gentoo のソースの Makefile が、Makefile 中の distclean や clean ターゲットの代わりに packageclean ターゲットを指定する必要がある場合には、override_dh_auto_clean ターゲットを作るとそれが可能になります。

\begin{verbatim}
override_dh_auto_clean:
    $(MAKE) packageclean
\end{verbatim}

もし、gentoo のソースの Makefile が、test ターゲットを含み、Debian パッケージをビルドする過程で実行されたくない場合は、空の override_dh_auto_test ターゲットを作ることで、スキップできます。

\begin{verbatim}
override_dh_auto_test:
\end{verbatim}

もし、gentoo パッケージに、普通ではない FIXES というアップストリームのチェンジログファイルがある場合、dh_installchangelogs はデフォルトではそのファイルをインストールしません。このファイルをインストールするには、FIXES を引数として、dh_installchangelogs に渡してやる必要があります。2⁷

\begin{verbatim}
override_dh_installchangelogs:
    dh_installchangelogs
\end{verbatim}

この新しい dh コマンドを使う際には、get-orig-source ターゲットを除き、項 4.4.1 にあるような明示ターゲット指定の正確な影響が把握するのが難しいかもしれません。明示ターゲットは、override_dh_* ターゲットおよび完全に独立したターゲットに限定して下さい。

2⁵lenny では、dh_* スクリプトの挙動を変えたい場合、rules ファイル内該当する行を見つけ出し、調整していました。

2⁶引数なしの dh_auto_build は、Makefile の最初のターゲットを実行します。

2⁷debhelper/changelog と debhelper/NEWS は常に自動でインストールされます。アップストリームの変更履歴は、ファイル名を小文字に変換し、changelog、changes、changes.txt、changes.txt のいずれかと一致するものを見つけます。
Chapter 5

debian ディレクトリーにあるその他のファイル

The `dh_make` command had major updates since this old document was written. So some parts of this document aren’t applicable any more.


The `debmake` command is used in place of the `dh_make` command in my new Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc).


The `dh_make` command will create some template configuration files under the debian directory. Take a look at all of them.

自明な場合、本章では debian ディレクトリー中のファイルは、前に付く debian/ を省略し簡明に表記しています。

dh_make コマンドは、一部の debhelper 用の設定テンプレートファイルを作らないことがあります。その場合、自分でエディタを使いそれを作成しなければなりません。

設定ファイルを有効にしたい際は、以下を実行して下さい:

- package の代わりに、実際のバイナリーパッケージの名前に設定ファイルをリネームしてください。
- 必要に応じて、テンプレートファイルの中身を書き換えます。
- 不要なテンプレートファイルは削除してください。
- 必要であれば、control ファイル（参照項4.1）を変更します。
- 必要なら rules ファイル（参照項4.4）を編集してください。

package をプリフィックスとして持たない、例えば install のような debhelper の設定ファイルは、最初のバイナリーパッケージへ適用されます。バイナリーパッケージが多数ある場合、package-1.install, package-2.install, 等のように、パッケージ名を設定ファイルのプリフィックスとすることで指定できます。
5.1 README.Debian

パッケージに関して、何か特別にユーザーに知らせなければならない情報や、オリジナルのソフトウェアと作成したDebianパッケージとの相違点があればここに記述します。

以下はdh_makeがデフォルトとして生成するものです:

```
#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
```

もし、ドキュメントがなければこのファイルを削除してください。詳しくはdh_installdocs(1)を参照してください。

5.2 compat

compatファイルは、debhelperの互換性レベルを規定します。現段階では、以下のようにdebhelperv10に設定しましょう:

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

一定の環境下では旧システムとのコンパチビリティーのためにcompatレベルv9を使ってよろしい。しかし、v9以下の如何なるレベルを使うことは勧められないし、新規パッケージに用いることは避けるべきです。

5.3 conffiles

大変な時間と労力を費やしてプログラムをカスタマイズしても、一回のアップグレードであなたの変更をあちこち上書きされてしまうとうざりします。このような設定ファイルをconffileと記録しておくことで、Debianはこの問題を解決しました。1パッケージをアップグレードする際に、あなたは古い設定ファイルをキープしたいかどうか尋ねられます。

DH_INSTALLDEB(1)は自動的に/etcディレクトリ以下のファイルを全てconffilesとみなすので、あなたのプログラムが他のディレクトリにconffilesを持たない場合は特に指定する必要はありません。ほとんどのパッケージの場合、/etc以下のみconffilesがある（そうあるべきです）ので、このファイルの存在は不要です。

あなたのプログラムが設定ファイルを利用する場合であっても、その設定ファイルがプログラム自身によって頻繁に上書きされるような場合には、パッケージをアップグレードするたびにdpkgによって設定ファイルの変更について確認を求められることになるので、その設定ファイルをconffilesに登録しないほうが良いでしょう。

あなたのパッケージケージングするプログラムが、ユーザーに/etcディレクトリの中にある設定ファイルを編集することを要求する場合、dpkgを黙らせるためにconffilesとして登録しない良さく使われる方法が2つあります:

- /etcディレクトリ中に、メンテナースクリプトによって生成された/varディレクトリ以下のファイルシンボリックリンクを張る。
- /etcディレクトリの中にメンテナースクリプトによってファイルを生成する。

メンテナースクリプトについての詳細は、項5.18を参照してください。

---

5.4 package.cron.*

パッケージが正しく動作するために、定期的にあるタスクを実行する必要がある場合は、これらのファイルで設定します。毎時間、毎日、毎週、または指定した時間に定期的タスクを実行するように指定することができます。ファイル名は以下です:

- cron.hourly - /etc/cron.hourly/package としてインストール：1時間ごとに実行する。
- cron.daily - /etc/cron.daily/package としてインストール：1日に1度実行。
- cron.weekly - /etc/cron.weekly/package としてインストール：1週間に1度実行。
- package.cron.monthly - /etc/cron.monthly/package: としてインストール：1ヶ月に1度実行。
- package.cron.d - /etc/cron.d/package としてインストール：どの時間でも指定可能。

上記のファイルの書式はシェルスクリプトです。package.cron.d は違い、crontab(5) の書式になります。
ログローテーションの設定には明示的な cron.* は必要ありません。これについては dh_installlogrotate(1) および logrotate(8) を参照してください。

5.5 dirs

このファイルにはパッケージが必要としているのに、なぜか通常のインストール手順 (dh_auto_install によって呼び出される make install DESTDIR=...) では作成されないディレクトリを指定します。通常、これは Makefile に問題があることを示唆しています。
install ファイルに書かれるファイルは最初にディレクトリを作成する必要はありません。項 5.11 を参照してください。
まずは無理にインストールしてみて、なにか問題が起きた場合にのみ使うべきでしょう。dirs ファイル中のディレクトリ名の頭にスラッシュが付かない事に注意してください。

5.6 package.doc-base

もしあなたのパッケージがマニュアルページや info 形式の文書以外に付属文書を含む場合、doc-base ファイルを使ってそれらを登録し、ユーザーがそれらの付属文書を、例えば dhlp(1) や dwww(1)、あるいは doccentral(1) コマンドなどで参照できるようにしましょう。
これには通常、/usr/share/doc/packagename/の中に収められるような HTML、PS、および PDF などの形式の付属文書が含まれます。
例えば、gentoo の gentoo.doc-base ファイルは次のようになります:

```
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
Files: /usr/share/doc/gentoo/html/*.
```

追加ドキュメントのインストールについて、詳細は項 3.3 を見てください。
5.7 docs

このファイルには、dh_installdocs(1)を使ってパッケージ生成の一時的なディレクトリーにインストールするために、パッケージに付属する資料のファイル名を指定してください。
デフォルトでは、ソースディレクトリーのトップレベルに存在する BUGS、README*、TODO などの名前を持つファイルを全て含みます。

gentoo に関していくつか他のファイルが含まれます:

```
BUGS
CONFIG-CHANGES
CREDITS
NEWS
README
READMEgtkrc
TODO
```

5.8 emacsen-*

パッケージをインストールする際にバイオコンパイル可能な Emacs ファイルがあなたのパッケージに含まれている場合、これらの emacsen-* ファイルを利用してそれを設定することができます。
これらのファイルは dh_installemacsen(1)によってパッケージ作成用の一時的なディレクトリーにインストールされます。
不要ならこのファイルを削除してください。

5.9 package.examples

dh_installexamples(1) コマンドはこのディレクトリーに列挙されたファイルを例としてインストールします。

5.10 package.init と package.default

もしあなたのパッケージがデーモンであり、システムの起動時に自動的に動作させる必要があるとしたら、私が最初に勧めたことをあなたはまるっきり無視してしまったわけですね。そうでしょう?-)

Please read dh_installinit(1) dh_installsystemd(1) to provide startup script.

The package.default file will be installed as /etc/default/package. This file sets defaults that are sourced by the init script. This package.default file is most often used to set some default flags or timeouts. If your init script has certain configurable features, you can set them in the package.default file, instead of in the init script itself.

アップストリームプログラムが init スクリプト用ファイルを提供する場合、それを使用するかしないかは自由です。もしアップストリームからの init スクリプトを使わないのであれば package.init に新しいのを作成しましょう。アップストリームの init スクリプトが問題なく正しい場所にインストールされとしても、rc* シンボリックリンクの設定は必要です。そのためにには、rules ファイルに以下を追加して、dh_installinit をオーバーライドしましょう:

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

不要なら、このファイルを削除してください。
### 5.11 install

パッケージにとってインストールが必要なファイルがあるにも関わらず、`make install` ではインストールされない場合、そのファイル名とファイルを置く目的地を `install` ファイルに記述します。そうすることは、`dh_install(1)` によってそれらのファイルがインストールされます。2 まずは使えそうな別のツールがないかどうかを調べましょう。例えば、ドキュメントはこのファイルではなく `docs` ファイルにあるべきです。

`install` ファイルはインストールされるファイルごとに 1 行必要とします。ファイル名（ビルドディレクトリーのトップを基点とした相対パス）、スペース、インストールするディレクトリー名（インストールディレクトリーを基点とした相対パス）という書式です。例えば、バイナリー `src/bar` のインストールを忘れた場合などに、`install` ファイルの項目は以下のように記述します:

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

上記によって、パッケージがインストールされたときに、`/usr/bin/bar` というバイナリーファイルが存在することになります。

また、この `install` ファイルは相対パスが変わらない場合、インストールディレクトリーの指定を省略することもできます。この書式は、ビルドした結果を、`package-1.install, package-2.install` などを使用し、複数のバイナリーパッケージに分割するような、大規模なパッケージで使われます。

`dh_install` コマンドはもし、カレントディレクトリーでファイルが見つからなかった場合は（または、`--sourcedir` で探すように指示したディレクトリー内で見つからなかった場合は）フォールバックして `debian/tmp` 内を検索します。

### 5.12 package.info

パッケージに `info` ページがある場合、`package.info` にそれらを挙げて、`dh_installinfo(1)` を使用してインストールします。

### 5.13 package.links

パッケージメンテナーとしてパッケージビルドディレクトリー中に追加のシンボリックリンクを作成する必要がある場合、リンク元とリンク先の両方のフルパスを `package.links` ファイル中にリストすることで `dh_link(1)` コマンドでそれらをインストールするべきです。

### 5.14 `{package,.source/}lintian-overrides`


`package.lintian-overrides` は `package` と名づけられたパッケージのためのファイルで、`dh_lintian` コマンドによって `usr/share/lintian/overrides/package` にインストールされます。

`source/lintian-overrides` はソースパッケージのためのファイルです。これはインストールされません。

### 5.15 manpage.*

プログラムはマニュアルページが必要です。もし無いなら作らなければなりません。`dh_make` コマンドはマニュアルページのテンプレートを作成します。マニュアルページがないコマンドのために、コピー、編集する必要があります。不要なテンプレートファイルを削除するのを忘れないようにしてください。

2`files` ファイルによって、`dh_movefiles(1)` コマンドが設定され、置換されます。
5.15.1 manpage.1.ex

マニュアルページは通常、nroff(1) で書かれています。manpage.1.ex のテンプレートも nroff で書かれています。これらのファイルをどう編集するのかについて、簡単な説明が man(7) にあります。最新のマニュアルページのファイル名は、解説されているプログラム名を含めなければなりません。ここでは、ファイル名を manpage から gentoo に変更しましょう。ファイル名は、.1 というサフィックスも含みます。これは、このマニュアルページはユーザーコマンドのものだ、という意味です。この部分を間違わないように気をつけしてください。以下はマニュアルページのリストです:

<table>
<thead>
<tr>
<th>セクション</th>
<th>内容</th>
<th>ノート</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ユーザーコマンド</td>
<td>実行可能なコマンドやスクリプト</td>
</tr>
<tr>
<td>2</td>
<td>システムコール</td>
<td>カーネルが提供する機能</td>
</tr>
<tr>
<td>3</td>
<td>ライブラリーコール</td>
<td>システムライブラリーが提供する機能</td>
</tr>
<tr>
<td>4</td>
<td>特殊ファイル</td>
<td>通常 /dev にある</td>
</tr>
<tr>
<td>5</td>
<td>ファイルフォーマット</td>
<td>例えば、/etc/passwd のフォーマット</td>
</tr>
<tr>
<td>6</td>
<td>ゲーム</td>
<td>ゲームや他に変不ないプログラム</td>
</tr>
<tr>
<td>7</td>
<td>マクロパッケージ</td>
<td>man のマクロ等</td>
</tr>
<tr>
<td>8</td>
<td>システム管理</td>
<td>普通 root が実行するプログラム</td>
</tr>
<tr>
<td>9</td>
<td>カーネルルーチン</td>
<td>非標準のコールや内部コール</td>
</tr>
</tbody>
</table>

つまり、gentoo のマニュアルページは gentoo.1 となります。オリジナルのソースファイルに gentoo.1 というマニュアルページがなければ、アップストリームのドキュメントと例を元にして、manpage.1.ex というテンプレートファイルを編集し gentoo.1 というマニュアルページを作らなければならないでしょう。

各コマンドの --help と --version 出力から help2man コマンドを用いてマニュアルページを作成することも可能です。3

5.15.2 manpage.sgml.ex

もし、nroff より SGML のほうが好みであれば、manpage.sgml.ex のほうがひな型として使うこともできます。こちらの場合には、以下の手順が必要です:

- ファイル名を gentoo.sgml のような名前に変更します。
- docbook-to-man パッケージのインストール
- control ファイルの Build-Depends 行へ docbook-to-man を追加
- rules ファイルに override_dh_auto_build ターゲットを追加します:

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

5.15.3 manpage.xml.ex

SGML より XML が好みであれば、manpage.xml.ex をひな形として使うこともできます。こちらの場合には、以下の手順が必要です:

- ソースファイルの名前を、gentoo.1.xml のような名前に変更します。
- docbook-xsl パッケージと xsltproc のような XSLT プロセッサのインストール（推奨）

3 help2man が作成する仮のマニュアルページに、詳細なドキュメントが info システムにあると記載されることに注意して下さい。info ページ中にコマンドが無い場合は、help2man コマンドが生成したページを手動で修正するべきです。
control ファイルの Build-Depends 行へ、docbook-xsl、docbook-xml、xsltproc の各パッケージを追加します。

rules ファイルに override_dh_auto_build ターゲットを追加します:

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

### 5.16 package.manpages

パッケージがマニュアルページにある場合、package.manpages ファイルにそれぞれをリストして、dh_installman(1)を使用してインストールします。

gentoo パッケージのマニュアルページとして docs/gentoo.1 をインストールするには、以下のように gentoo.manpages ファイルを作成します:

```
docs/gentoo.1
```

### 5.17 NEWS

dh_installchangelogs(1) コマンドでインストールします。

### 5.18 {pre,post}{inst,rm}

postinst や preinst や postrm や prerm ファイルは*メンテナースクリプトと呼ばれています。これらのスクリプトは、パッケージを管理するエリアに置かれ、インストール、アップグレード、削除される際に dpkg によって実行されます。

新米メンテナーのうちに、問題になることが多いのでメンテナースクリプトを直接編集しないようにしましょう。詳しくは Debian Policy Manual, 6 "Package maintainer scripts and installation procedure" (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) を参照し、dh_make によって生成されるサンプルファイルに目を通してください。

もし私の忠告を無視して、メンテナースクリプトを直接編集した場合は、インストール、アップグレードだけでなく、削除とパージのテストもしっかりと行ってください。

新バージョンへのアップグレードは静かであるべきで、押し付けがましくてはいけません。（現行ユーザーは、バグが直されたことや新機能が追加されたことで気づかない限りアップグレードに気づかないのが理想です。）

アップグレードが出しゃばる必要がある場合（例えば、構造がまったく異なる設定ファイルがホームディレクトリーに散在する場合など）は、パッケージのデフォルトを（例えばサービスを停止する等の）安全側に設定したり、最後の手段としてはポリシーに要求されるきちんととしたドキュメント（README.DebianとNEWS.Debian）を提供するなどの対策を考えるべきです。アップグレード後にメンテナースクリプトで debconf ノートを呼び出したりしてユーザーに迷惑を掛けないでください。

*{pre,post}{inst,rm} という bash 独自の短縮形をこれらのファイル名の表記としていますが、システムシェルである dash との互換性のために、これらのメンテナースクリプトでは純粋な POSIX シンタックスを使うべきです。
ucfパッケージは、メンテナースクリプトによって管理されているようなconffilesとラベルされていないファイルに関して、ユーザーによって変更されたファイルを保存するconfileのような処理をする仕組みを提供します。この仕組みを使うとこれらの問題を最小化できます。

これら、メンテナースクリプトはなぜDebianを選ぶのかという理由の1つでもあります。これらの仕組みで、ユーザーが迷惑がる原因とならないよう細心の注意をはらいましょう。

5.19 package.symbols

新米メンテナーにとってはライブラリーのパッケージは容易ではないし、避けるべき行為です。このように言いましたが、あなたがパッケージを含む場合には、debian/package.symbolsファイルを作成すべきです。項A.2を参照下さい。

5.20 TODO
dh_installdocs(1)コマンドでインストールします。

5.21 watch

watchファイルの書式はuscan(1)を参照してください。watchファイルは、uscan(devscriptsパッケージに含まれます)を設定し、最初ソースを入手したサイトを監視します。これはDebianパッケージトラッカー(https://tracker.debian.org)サービスによっても使用されています。

以下がその内容です:

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

通常、このwatchファイルでは、http://sf.net/gentooのURLがダウンロードされ、<a href=...>ホームへのリンクを検索します。リンクされたURLのベースネーム(最後の/から後の部分のみ)はPerlの正規表現(perlre(1)参照)パターンgentoo-(.+).tar.gzに照らし合わせられます。一致したファイルの中から、バージョンの番号が一番大きいものがダウンロードされ、その後アップデートされたソースツリーを作成するためにuupdateプログラムを実行します。


ターボールの公開鍵電子署名をアップストリームが提供している際には、uscan(1)中に記載されたpgpsigurlmangleオプションを用いてその正統性を検証することが望ましい。

5.22 source/format

debian/source/formatファイルでは、ソースパッケージのための理想的な書式を示すための行があります。(完全なリストは、dpkg-source(1)を参照してください。)squeeze以降は、以下のどちらかになっているべきです:

- 3.0 (native): ネイティブDebianパッケージ
3.0 (quilt): それ以外の全て。

新しい3.0 (quilt)の文書はquiltパッチによる変更をdebian/patchesに記録します。そして、その変更は自動的にソースツリーを展開するときに適用されます。

Debianの変更は、debianディレクトリ以下のファイル全てを含め、debian.tar.gzアーカイブに保存されています。この新しい文書は、特殊な方法を用いることなく、PGNアイコンなどのパッケージメンテナーによるバイナリーファイルを含めることが可能です。

dpkg-sourceが3.0 (quilt)の文書でソースパッケージを展開する際、debian/patches(series)に列挙されたパッチを自動的に適用します。--skip-patchesオプションで、展開時にパッチを適用しないようにできます。

5.23 source/local-options

Debianをパッケージングする活動をVCSで管理したい場合、アップストリームのソースをトラックするためのブランチ（例：upstream）とデbianパッケージをトラックするための別のブランチ（Gitでの典型例：master）を作成します。後者が場合、新しいアップストリームのソースをマージするので簡単にするために、パッチの当てていないアップストリームのソースをdebian/*ファイルと一緒に持っておきます。

パッケージをビルドした後は、ソースのパッチは通常当てたままにされます。masterブランチにコミットする前に手動でquilt pop -aを実行してパッチを外す必要があります。debian/source/local-optionsファイルにunapply-patchesを書いておけば、自動的にパッチを外せます。このファイルは生成されたソースパッケージには含まれず、ローカルビルドでの挙動のみを変更します。このファイルはabort-on-upstream-changesを含むかもしれません（dpkg-source(1)参照）。

5.24 source/options

ソースツリーの中の自動生成されるファイルはパッケージングする際に無意味で大きなパッチファイルを生成するのでとても厄介です。項目4.4.3で説明したようにdh_autoreconfのようなカスタムモジュールが本問題を解消するために存在します。

dpkg-source(1)の--extend-diff-ignoreオプション引数にPerl正規表現を提供すると、ソースパッケージ生成時に自動生成ファイルへの変更を無視できます。

この自動生成ファイルの問題の一般的な解決策としてソースパッケージのsource/optionsファイル中にdpkg-sourceオプション引数を保存する事が出ます。以下の例では、config.subとconfig.guessとMakefileに関してパッチファイルの生成をスキップします。

extend-diff-ignore = "((\|/)config\sub|config\guess|Makefile)"

5.25 patches/*

古い1.0のソースフォーマットは、debian内にパッケージメンテナンスファイルと、パッチファイルを含む単一の大きなdiff.gzファイルを作っていました。そのようなファイルは、ソースツリーの変更を後から調べたり、理解するのが非常に厄介でした。これはあまりいただけません。

新しい3.0 (quilt)は、quiltコマンドを使って、パッチをdebian/patches/*に置きます。debianディレクトリ以下に含まれているパッチやその他のパッケージデータは、debian.tar.gzファイルとしてパッケージ

ソースの書式を3.0 (quilt)や3.0 (native)に移行する際の注意点などは、DebSrc3.0(http://wiki.debian.org/Projects/DebSrc3.0)を参照下さい。

この新しいフォーマットは、複数のアップストリームのtarアーカイブやこの他の圧縮方法もサポートしています。詳細は本稿の範疇を超えるため割愛します。
グされます。dpkg-source コマンドは、quilt 形式のバッチデータを quilt パッケージなしで 3.0 (quilt) として扱えるので、quilt パッケージを Build-Depends に記載する必要はありません。

quilt コマンドについては quilt(1) で説明されています。ソースへの変更は、debian/patches ディレクトリー内 -p1 パッチファイルのスタックとして記録され、debian ディレクトリーの外のソースツリーには触れません。それらのパッチの順番は debian/patches/series ファイルに記録されます。パッチの適用 (=push) も、外す (=pop) のも、更新 (=refresh) も、簡単にできます。

第3章では、debian/patches に 3 つのパッチを作りました。

Debian のパッチは debian/patches にあるので、項 3.1 の説明に従い、dquilt コマンドを正しく設定してください。

誰かが (あなた自身も含みます) foo.patch というパッチを後から提供した際の、3.0 (quilt) ソースパッケージの変更はとてもシンプルです:

```
$ 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
```

新しい 3.0 (quilt) 形式で保存されるパッチには曖昧さがあってはいけません。それを保証するために、dquilt pop -a; while dquilt push; do dquilt refresh; done としてください。

①パッチセットをメンテナンスするためのいくつかの方法が提案され、Debian パッケージで使われていますが、quilt が推奨されています。他には、dpatch、dbs、cdbs、などがあります。これらの方法は、大抵 debian/patches/* ファイルでパッチを管理しています。
②スポンサーにパッケージのアップロードを頼む時にも、あなたが加えた変更に対するこのような明確な分離とドキュメントは、スポンサーによるパッケージのレビューを促進させるためにも、非常に重要です。
Chapter 6

パッケージのビルド


これでパッケージをビルドする準備が整いました。

6.1 完全な（再）ビルド

完全なパッケージの（再）ビルドを行うには、以下を確実にインストールして下さい。

• build-essentialパッケージ
• Build-Dependsに挙げられているパッケージ（参照項4.1）
• Build-Depends-indepに挙げられているパッケージ（参照項4.1）

ソースディレクトリで以下のコマンドを実行してください:

$ dpkg-buildpackage -us -uc

このコマンドはバイナリーパッケージとソースパッケージをビルドする作業をすべて行ってくれます。これには以下の作業が含まれます:

• ソースツリーのクリーン (debian/rules clean)
• ソースパッケージのビルド (dpkg-source -b)
• プログラムのビルド (debian/rules build)
• バイナリーパッケージのビルド (fakeroot debian/rules binary)
• .dscファイルの作成
• dpkg-genchangesを使用し.changesファイルを作成

満足なビルド結果の場合には、debsignコマンドを用いてあなたの秘密GPG鍵で.dscと.changesファイルを署名します。秘密フレーズを2回入力する必要があります。1

ノンネイティブパッケージの場合、パッケージビルド後の親ディレクトリー(~/gentoo)に以下のファイルが生成されているはずです:  

1GPGキーは信頼の網に連結するようにDebianデベロッパーによって署名され、the Debian keyring (http://keyring.debian.org) に登録されていないければいけません。こうすることでDebianアーカイブにパッケージをアップロードして受け付けられるようになります。Creating a new GPG key (http://keyring.debian.org/creating-key.html) とDebian Wiki on Keysigning (http://wiki.debian.org/Keysigning) を参照下さい。
• gentoo_0.9.12.orig.tar.gz
これは単にDebian標準に合わせるために名前を変更しただけで、中身はオリジナルなソースコードのtarアーカイブです。これは元来、dh_make -f ../gentoo-0.9.12.tar.gzで作成されたということを覚えておいてください。

• gentoo_0.9.12-1.dsc
これはソースコードの内容の概要です。このファイルはあなたのcontrolファイルから生成され、dpkg-source(1)によってソースを展開する際に使われます。

• gentoo_0.9.12-1.debian.tar.gz
この圧縮されたtarアーカイブには、あなたのdebianディレクトリーの中身が含まれています。オリジナルのソースコードに行った変更や追加などの情報は全てdebian/patches内に、quiltパッチとして保存されます。
上記3つのファイルを使えば誰でも簡単にあなたのパッケージをスクラッチからビルドすることができます。

3つのファイルを任意の場所にコピーし、dpkg-source -x gentoo_0.9.12-1.dscを実行するだけです。

• gentoo_0.9.12-1_i386.deb
これは、あなたが生成した完全なバイナリーパッケージです。他の全てのパッケージと同じく、dpkgを使ってインストールしたり削除したりできます。

• gentoo_0.9.12-1_i386.changes
このファイルは現在のリビジョンパッケージにおける変更点をすべて記載したもので、DebianFTPアーカイブ管理プログラムによって、バイナリーパッケージをFTPアーカイブにインストールするために利用されます。このファイルの一部は、change logファイルと.dscをもとに生成されます。
パッケージの保守管理を続けていくと、挙動の変更や新機能の追加をすることがあります。あなたのパッケージをダウンロードする人は、このファイルを見れば何が変わったのか、一目でわかります。また、このファイルの中身はDebianアーカイブ管理プログラムによって、debian-devel-changes@lists.debian.org(http://lists.debian.org/debian-devel-changes/)メーリングリストへ流されます。

Debain FTPアーカイブにアップロードする前に、~/.gnupgディレクトリー中にあるあなたの秘密GPG鍵でdebsignコマンドを用いてgentoo_0.9.12-1.dscとgentoo_0.9.12-1_i386.changesファイルを署名しなければいけません。
以下の~/.devscriptsファイルを用いると、debsignコマンドはあなたが指定した秘密GPGキーID(パッケージをスポンサーする際に便利)で署名できます:

```
DEBSIGN_KEYID=Your_GPG_keyID
```

.dscと.changesファイルに記載されている長い数字の羅列は各ファイルのMD5/SHA1/SHA256チェックサムです。パッケージをダウンロードした人は、sha1sum(1)、sha256sum(1)を使って整合性をテストすることができます。もし、数字が一致しない場合には、ファイルが壊れているか、あるいは何者かによって改ざんされていると分かるわけです。

6.2 オートビルダー

Debianは、様々なアーキテクチャ上でbuildデーモンを走らせているオートビルダーネットワーク(http://www.debian.org/devel/buildd/)によって、色々な移植版(http://www.debian.org/ports/)をサポートしています。あなたがそれらを明示的に使う必要はありませんが、パッケージがどうなるのかを知っておくと良いでしょう。それでは、あなたのパッケージがどのように異なるアーキテクチャ向けに再ビルドされるのかを見てきましょう。

Architecture: anyのパッケージは、オートビルダーシステムによって再ビルドされます。それは、以下を確実にインストールします。

23.0 (quilt)ソースフォーマットでquiltパッチを当てないようにするには、上記コマンドに--skip-patchesオプションをつけて実行します。または、通常の操作の後、quilt pop -aを実行する方法もあります。
3実際のオートビルダーシステムは、本稿の説明よりもかなり複雑なスキームによって実現しています。それらの詳細は、本稿の範囲を超えるために割愛します。
• `build-essential`パッケージ
• `Build-Depends`に挙げられているパッケージ（参照項4.1）

そして、ソースディレクトリーで次のコマンドを実行します:

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

これは、別のアーキテクチャー上で、アーキテクチャー依存のバイナリーパッケージを生成する作業をすべて行ってくれます。これには以下の作業が含まれます:

• ソースツリーのクリーン (debian/rules clean)
• プログラムのビルド (debian/rules build)
• アーキテクチャー依存のバイナリーパッケージをビルド (fakeroot debian/rules binary-arch)
• `gpg` を使用したソース.dsc ファイルへの署名
• `dpkg-genchanges` および `gpg` を使用したアップロード用.changes ファイルの生成と署名

あなたのパッケージが他のアーキテクチャー用にも存在するのは、このためです。

`Build-Depends-indep`フィールドのパッケージは、通常のパッケージの場合はインストールを要求されますが（参照項6.1）、オートビルダーシステムでは、アーキテクチャー依存のパッケージのみをビルドするのでこれらのインストールは必須ではありません。オートビルダーを使用した場合と普通のパッケージングとのこの違いにより、`debian/control`ファイルの `Build-Depends`か `Build-Depends-indep`のどちらにパッケージを記載するかが決定されます。（参照項4.1）

### 6.3 `debuild` コマンド

`dpkg-buildpackage`によるビルドプロセス周辺は、`debuild`によりさらに自動化できます。`debuild(1)`を参照してください。

`debuild` コマンドは、Debian パッケージをビルドのあと静的チェックをする `lintian` コマンドを実行します。`lintian` コマンドは `~/.devscripts` を用いて以下のようにカスタマイズできます:

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

以下のようにすれば一般ユーザーアカウントから、簡単にソースをクリーンしパッケージを再ビルドできます:

```
$ debuild
```

ソースツリーのクリーンも簡単です:

```
$ debuild -- clean
```

⁴`pbuilder`パッケージとは違い、オートビルダーによって使用される `sbuild`パッケージ下での `chroot`環境では、最小システムを強制しないので、多くのパッケージがインストールされたままであるかもしれません。
6.4 pbuilder パッケージ

ビルト依存を確認するためのクリーンルーム（chroot）ビルト環境として、pbuilder パッケージが非常に便利です。これは使うことで、異なるアーキテクチャー向けに sid 環境オートビルダーの下でのソースからのクリーンなビルトが保証され、常に RC（リリースクリティカル）に分類される重要度が serious（深刻）の FTBFS（Fails To Build From Source、ソースからのビルト失敗）パッケを防ぎます。

それでは、pbuilder をカスタマイズしてみましょう:

- /var/cache/pbuilder/result ディレクトリーを、ユーザーアカウントから書き込むように設定してください。
- フックスクリプトを置くために、ユーザーからの書き込みが可能なディレクトリーを作成してください。例) /var/cache/pbuilder/hooks
- ~/.pbuilderrc か /etc/pbuilderrc に以下のように設定します。

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

それでは、ローカル pbuilder の chroot システムを以下のようにして初期化しましょう:

```
sudo pbuilder create
```

既に完全なソースパッケージがあれば、foo.orig.tar.gz ファイル、foo.debian.tar.gz ファイル、foo.dsc ファイルが存在するディレクトリーで、ローカルの pbuilder の chroot システムをアップデートし、バイナリーパッケージをその中でビルトしましょう:

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

新しくビルトした GPG 署名の無いパッケージは非ルート所有権で /var/cache/pbuilder/result/ に置かれます。
.dsc ファイルや.changes ファイルへの GPG 署名は次のようにするとできます:

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

更新されたソースツリーが既にあるが一致するソースパッケージを生成していない場合は、この代わりに、debian ディレクトリーが存在するディレクトリーで、以下のコマンドを発行します:

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

pbuilder --login --save-after-login コマンドで、chroot 環境にログインし、好きに設定することができます。シェルプロンプトから ^D (Control-D) で抜けると、その環境を保存しておくことができます。

最新の lintian コマンドは chroot 環境から次のように設定されたフックスクリプト/var/cache/pbuilder/hooks/B90lintian を使用して実行することができます:?

```#!/bin/sh
set -e
install_packages() {
    apt-get -y --allow-downgrades install "$@
}
```

pbuilder パッケージはまだ進化の過程なので、実際の構成は、最新の公式ドキュメントで確認して下さい。

Debian パッケージのオートビルドに関しては http://buildd.debian.org を参照下さい。

HOOKDIR=/var/cache/pbuilder/hooks をここで仮定しています。フックスクリプトのサンプルは /usr/share/doc/pbuilder/examples ディレクトリーにあります。
sid 環境向けのパッケージを正しくビルドするには最新の sid 環境が必要です。sid にはあなたの環境全てを移行するには望ましくない問題を抱えていることが少なくありません。pbuilder パッケージはそのような状況への対処の助けとなります。

stable/updates や stable-proposed-updates がリリースされた後、stable パッケージのアップデートが必要な場合があります。そのような場合に、即座にアップデートしない言い訳として sid を使っているからというのは不十分です。pbuilder パッケージは、同じアーキテクチャーのほぼ全ての Debian 派生であるディストリビューションへのアクセスを手助けします。

http://www.netfort.gr.jp/~dancer/software/pbuilder.html と pdebuild(1) と pbuilder(8) を参照下さい。

6.5 git-buildpackage コマンドとその仲間

アップストリームがソースコード管理システム (VCS) を使っているのであれば、同様に使用することを考慮すべきです。それによって、マージとアップストリームパッケの取捨選択がより簡単になります。各 VCS 毎に Debian パッケージをビルドするための特別なラッパースクリプトのパッケージもいくつかあります。

• git-buildpackage: Git リポジトリ内の Debian パッケージの支援プログラム群です。
• svn-buildpackage: Debian パッケージを Subversion で管理するための支援プログラム群です。
• cvs-buildpackage: CVS ソースツリーのための Debian パッケージスクリプト群です。

Debian デベロッパーが alioth.debian.org (http://alioth.debian.org/) 上の Git サーバーを用い Debian パッケージを管理するのに git-buildpackage を使うことがよくあります。このパッケージはパッケージ活動を自動化する多くのコマンドを提供します。

• gbp-import-dsc(1): import a previous Debian package to a Git repository.
• gbp-import-orig(1): import a new upstream tar to a Git repository.
• gbp-dch(1): generate the Debian changelog from Git commit messages.
• git-buildpackage(1): Debian パッケージを Git リポジトリからビルドします。
• git-pbuilder(1): Debian パッケージを Git リポジトリから pbuilder/cowbuilder を持ちいてビルドします。

これらのコマンドはパッケージング活動を追跡する 3 つのブランチを使用します:

• Debian パッケージのソースツリーは、main。
• アップストリームのソースツリーは、upstream。
• --pristine-tar オプションにより生成されるアップストリーム tar アーカイブは、pristine-tar。

`--pristine-tar` オプションにより生成されるアップストリーム tar アーカイブは、pristine-tar。このコマンドはDebianパッケージのソースツリーは、main。

--pristine-tar オプションにより生成されるアップストリーム tar アーカイブは、pristine-tar。このコマンドはDebianパッケージのソースツリーは、main。
6.6 部分的な再ビルド

大規模なパッケージの場合には、debian/rules をちょっといじるたびに、毎回最初からパッケージの再ビルドをやりなおすのは手間です。テスト目的であれば、以下的方法でアップストリームソースを再ビルドせずに .deb ファイルを生成することができます。13:

$ fakeroot debian/rules binary

また、以下の方法を使えば生成可能かどうかをチェックすることができます:

$ fakeroot debian/rules build

最終的にきちんとテストが完了したら、正しい手順に従ってパッケージを最初から再ビルドすることを忘れないでください。この方法でビルドした .deb ファイルをアップロードしようとしても、おそらくうまくアップロードできないでしょう。

6.7 コマンド階層

パッケージ作成に用いる多くのコマンドが、コマンド階層の中でお互いいかなる関係にあるかの簡単なまとめを以下に記します。同じ事をするのに多くの方法があります。

- debian/rules = パッケージビルド用のメンテナースクリプト
- dpkg-buildpackage = パッケージビルトツールの核
- debuild = dpkg-buildpackage + lintian (サニタイズした環境変数下でのビルド)
- pbuilder = Debian の chroot 環境ツールの核
- pdebbuild = pbuilder + dpkg-buildpackage (chroot 中でビルド)
- cowbuilder = pbuilder 実行の加速
- git-pbuilder = pdebbuild の使いやすいコマンドラインシンタックス (gbp buildpackage が使用)
- gbp = Debian ソースを git レポ下で管理
- gbp buildpackage = pbuilder + dpkg-buildpackage + gbp

gbp buildpackage や pbuilder 等のよりハイレベルなコマンドを用いることは完全なパッケージビルト環境を保証しますが、debian/rules や dpkg-buildpackage 等のよりローレベルのコマンドがそれらの下で実行されるかを理解することは必須です。

---

13その場合は、通常だと正しく設定される環境変数は設定されません。アップロード用のパッケージはこの簡易メソッドで生成しないでください。
Chapter 7

パッケージのエラーの検証


ここでは、公式アーカイブにパッケージをアップロードする前に、作成したパッケージのエラーをあなた自身で確認するために知っておかなければならない方法について、幾つか述べます。

7.1 怪しげな変更

あなたの3.0 (quilt) フォーマットのノンネイティブ Debianパッケージをビルドした後で、debian/patchesディレクトリー内にdebian-changes-*のような新規の自動生成されたパッチファイルを見つけた場合、何かの間違いで何らかのファイルを変更したか、ビルドスクリプトがアップストリームソースに変更を加えた可能性が大きいあります。あなたの間違いなら、それを修正しましょう。ビルドスクリプトが引き起こした場合は、項4.4.3にあるようにdh-autoreconfを使って根本原因を修正するか、項5.24にあるようにsource/optionsを使って回避策をとります。

7.2 インストールに対するパッケージの検証

あなたのパッケージが問題なくインストールできるかどうかをテストしなければなりません。debi(1)コマンドがあなたが作成した全てのバイナリーパッケージのインストールテストに役立ちます。

$ sudo debi gentoo_0.9.12-i386.changes

別のシステムでインストール時に問題が起きるのを防ぐために、Debian アーカイブよりダウンロードした Contents-i386 ファイルを用いて、他の既存のパッケージと重複するファイルがないことを確認しておかなければなりません。apt-fileコマンドはこの作業において恐らく役立つでしょう。重複するファイルが存在するならば、実際に問題になることを回避するために、ファイルをリネームするか、複数のパッケージが依存する独立のパッケージに共通ファイルを移動するか、他の影響のあるパッケージと調整しながら alternatives (update-alternatives(1)参照) を用いるか、debian/controlにConflicts項目を宣言して下さい。

7.3 パッケージのメンテナースクリプトの検証

全てのメンテナースクリプト(preinst,prerm,postinstそしてpostrmファイルのこと)は、それがdebhelperプログラムで自動生成されたのではない場合は、正しく書くことが非常に困難です。だからあなたが新米メンテナーならばこれでは使わないで下さい (項5.18参照)。
パッケージがこれらの重要なメンテナースクリプトを使用するならば、インストールだけではなく、削除、完全削除 (purge)、そしてアップグレードについても確実にテストしましょう。多くのメンテナースクリプトのバグは削除もしくは完全削除の場合に発生します。これらのテストのためには、以下の様に dpkg コマンドを実行します:

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

以下の様々な順番で実行すべきでしょう:

- (可能な場合は) 前回バージョンをインストールします。
- 旧バージョンからアップグレードします。
- 旧バージョンになるよう、ダウングレードします (オプション)。
- 完全削除 (purge) します。
- 新しいパッケージとしてインストールします。
- 削除します。
- もう一度インストールします。
- 完全削除 (purge) します。

これが最初に作成したパッケージならば、将来発生するかもしれぬ問題を防ぐために、異なるバージョンのダミーパッケージを作成すべきです。

あなたのパッケージが過去の Debian にてリリースされていた場合には、人々は大抵最新の Debian のリリース版に含まれているバージョンからパッケージのアップグレードをするであろう、ということに配慮しましょう。上記の手順で、そのバージョンからきちんとアップグレードできることを、忘れずに確認しておいてください。

ダウングレードは正式にはサポートされていませんが、これをサポートするのは友好的な態度です。

### 7.4 Using lintian

lintian(1) を .changes に対して実行しましょう。lintian コマンドはパッケージ作成時のよくある間違いをチェックするために多くのテストスクリプトを実行します。1

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

もちろん、.changes のファイル名はあなたが作成したパッケージに置き換えて下さい。lintian コマンドの出力は以下のようマークされています:

- E: はエラーです。実在のポリシー違反もしくはパッケージエラーです。
- W: は警告です。ポリシー違反もしくはパッケージングエラーである可能性があります。
- I: は参考情報です。パッケージのとある性質について参考となる情報を提供します。
- N: は覚書です。デバッグに有用な情報を詳細します。
- O: はオーバーライド通知です。lintian-overrides ファイルによりメッセージがオーバーライドされたメッセージです。これは --show-overrides オプションを指定した際に表示されます。

警告が出た場合には、パッケージを調整するか、その警告が不当であることを確認して下さい。もし警告が不当である場合には項 5.14 で述べた lintian-overrides を作成して下さい。

dpkg-buildpackage によるパッケージの生成と lintian の実行は、debuild(1) コマンド、もしくは pdebuild(1) コマンドを用いれば一気に実行することができます。

1/etc/devscripts.conf や ~/.devscripts において項 6.3 で述べた設定をしている場合には、lintian に -i -I --show-overrides オプションを指定する必要はありません。
7.5 debc コマンド

debc(1) コマンドを用いるとバイナリーパッケージに含まれるファイルの一覧が得られます。

$ debc package.changes

7.6 debdiff コマンド

debdiff(1) コマンドを用いると、二つのソースパッケージに含まれているファイルを比較することができます。

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

debdiff(1) コマンドは二つのバイナリーパッケージに含まれるファイルの一覧を比較することもできます。

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

これらのコマンドは、ソースパッケージ中でどんな変更をしたのか、バイナリーパッケージの中で意図せず削除したり配置を間違えたりしていないか、そしてバイナリーパッケージの更新時に不用意な変更をしていないかどうか、といった事柄を確認するのが便利です。

7.7 interdiff コマンド

interdiff(1) コマンドを用いると、二つの.diff.gz ファイルを比較することができます。旧来の 1.0 ソース形式でパッケージを更新している場合には、メンテナーがソースに意図しない変更をしていないことを確認するのが便利です。

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

新規の 3.0 ソースフォーマットは項5.25で説明したように複数のパッチファイル中に変更を保存します。各々のdebian/patches/* ファイルの変化もinterdiffを使って追いかけられます。

7.8 mc コマンド

mc(1)の様に、*.deb パッケージファイルだけでなく、*.udev, *.debian.tar.gz, *.diff.gz, *.orig.tar.gzの中身を閲覧することができるファイルマネージャを用いると、多くのファイルの検査が直感的に行なえます。

不要なファイルやサイズがゼロのファイルがバイナリーパッケージとソースパッケージに含まれていないことをよく確認して下さい。大抵は不要なファイルが正しく削除されずに残ってしまいます。rulesを調整しこれを修正して下さい。
Chapter 8

パッケージの更新


パッケージをリリースしたなら、すぐにそれを更新する必要があります。

8.1 Debian リビジョンの更新

例えば仮に、#654321 という番号のバグレポートがあなたのパッケージに対してファイルされ、解決可能な問題が記述されていたとしましょう。パッケージの新しい Debian リビジョンを作成するには、以下を実行する必要があります。

- もしこれが新規のパッチとして記録される場合には、以下のようにします。
  - dquilt new bugname.patch としてパッチ名を設定します。
  - dquilt add buggy-file として変更されるファイルを宣言します。
  - アップストリームバグに関してパッケージソース中の問題を修正します。
  - dquilt refresh として bugname.patch に記録します。
  - dquilt header -e としてその内容記述を追加します。
- もし既存のパッチをアップデートする場合には、以下のようにします。
  - dquilt pop foo.patch として既存の foo.patch を呼び出します。
  - 古い foo.patch 中の問題を修正します。
  - dquilt refresh として foo.patch を更新します。
  - dquilt header -e としてその内容記述を更新します。
  - while dquilt push; do dquilt refresh; done として fuzz を削除しながら全てのパッチを適用します。

次に Debian changelog ファイルの先頭に新しいリビジョンを追加します。例えば dch -i を実行するか、またはバージョンを明示したい場合なら dch -v version-revision を実行してあなたの好きなエディタで説明を記入すると楽にできます。

changelog の説明行に、このリビジョンで解決されたバグと、その解決方法についての簡単な説明を記載し、Closes: #54321 と続けておきます。これによってあなたのパッケージが Debian アーカイブ中に受け入れられた時、アーカイブ管理ソフトウェアによって該当するバグレポートが魔法のように自動的に閉じられます。1

1 日付を要求されるフォーマットで入力するには、LANG=C date -R を使います。
8.2 新規のアップストリームリリースの検査

Debian アーカイブ用の新しいアップストリームリリースパッケージの準備をする際は、まず、新アップストリームリリースをチェックしなければなりません。

アップストリームの changelog や NEWS、その他の新しいバージョンでリリースされたドキュメントを読んでから始めてください。

以下のようにすれば何かおかしな事がないかに注意を払いつつ新旧のアップストリームソース間の変更検査ができます。

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

missing や aclocal.m4 や config.guess や config.h.in や config.sub や configure や depcomp や install-sh や ltmain.sh や Makefile.in 等の Autotools によって自動生成されるファイルへの変更は無視していい場合があります。ソースを検査するための diff を実行する前に消去してもいいです。

8.3 アップストリームソフトウェアの新版更新

もし foo パッケージが新規の 3.0 (native) や 3.0 (quilt) フォーマットで適正にパッケージされていれば、新規のアップストリームバージョンをパッケージする際は基本的に旧 debian ディレクトリーを新規ソースへと移動するのみです。これは新規に展開されたソース内で tar xzvf /path/to/foo_oldversion.debian.tar.gz を実行すれば出来ます。もちろん当然するべき細々としたことをする必要はありません。

- アップストリームソースのコピーを foo_newversion.orig.tar.gz ファイルとして作成します。
- Debian の changelog ファイルを dch -v newversion-1 を使って更新します。
  - New upstream release という項目を追加します。
  - 報告のあったバグを修正する新規アップストリームリリース中の変更点に関して簡潔に記載しバグを Closes: # バグ番号 と追記してクローズします。
  - 報告のあったバグを修正する、メンテナーによる新規アップストリームリリースへの変更点に関して簡潔に記載しバグを Closes: # バグ番号 と追記してクローズします。

2もし dch -r コマンドを使ってこの最終変更をする場合には、エディターにより changelog ファイルを明示的に保存して下さい。
3もしパッケージ foo が旧 1.0 フォーマットでパッケージされている場合は、新規に展開されたソース内で zcat /path/to/foo_oldversion.diff.gz | patch -p1 を実行すれば出来ます。
while dquilt push; do dquilt refresh; done として fuzz を削除しながら全てのパッチを適用します。

もしパッチやマージがクリーンに適用できない場合は、状況を精査します (.rej ファイル中にヒントがあります)。

もしソースにあなたが適用していたパッチがアップストリームソースに統合された場合は、
- dquilt delete として削除します。

もしソースにあなたが適用していたパッチが新規アップストリームソースとぶつかる場合は、
- dquilt push -f として baz.rej としてリジェクトを強制しながら古いパッチを適用します。
- baz.rej の本来目指した効果を実現するために、baz ファイルを手動で編集します。
- dquilt refresh としてパッチを更新します。
- while dquilt push; do dquilt refresh; done まで戻り継続します。

このプロセスは uupdate(1) コマンドを以下のように使うことで自動化できます:

```
$ apt-get source foo
...
$ ls -F
foo_olddversion/
foo_olddversion-1.debian.tar.gz
foo_olddversion-1.dsc
foo_olddversion.orig.tar.gz
$ wget http://example.org/foo/foo-newversion.tar.gz
$ cd foo-newversion
$ uupdate -v newversion ../foo-newversion.tar.gz
$ cd ../foo-newversion
$ dch ...
```

デビアン/watch ファイルを項 5.21 に記載されたように設定していれば、wget コマンドをスキップすることが出来ます。foo_olddversion ディレクトリー中で uupdate コマンドを実行する代わりに、単に uscan(1) コマンドを実行します。こうすると魔法のように更新されたソースを見つけ、それをダウンロードし、uupdate コマンドを実行します。 ⁴

今まで項 6.1 、第 7 章、第 9 章の中で実行してきたことを繰り返し、更新したソースをリリースできます。

8.4 パッケージ化スタイルの更新

パッケージ化スタイルの更新はパッケージ更新のために必須活動ではありません。しかし、こうすることで最新の debhelper システムと 3.0 ソースフォーマットの全能力を活用できます。 ⁵

もし何らかの理由で消去されたテンプレートファイルを追加する必要がある場合には、同一の Debian ソースツリーの中で --addmissing オプションとともに dh_make をもう一度実行してもいいです。そして、それらを適正に編集しましょう。

４もし uscan コマンドが更新されたソースはダウンロードするが uupdate コマンドを実行しない場合には、URL の最後に debian uupdate となるように debian/watch ファイルを修正するべきです。

５もしあなたのスポンサーや他のメンテナーが既存のパッケージ化スタイル更新に異存がある場合には、更新することもまたその議論することとは意味がありません。他にするべきより重要なことがあります。
• *debian/rules* ファイルに関して *dh* シンタックスへとパッケージが更新されていない場合は *dh* を使うように更新しましょう。*debian/control* ファイルもそれに合わせて変更しましょう。

• コモン Debian ビルドシステム (**cdbs**) による Makefile 包含メカニズムで生成された *rules* ファイルを *dh* シンタックスに更新しようとする場合には、以下を参照し DEB_* 設定変数を理解して下さい。

```bash
- /usr/share/doc/cdbs/cdbs-doc.pdf.gz のローカルコピー
- 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/)
```

• *foo*.diff.gz ファイル無しの 1.0 ソースパッケージの場合、3.0 (native) と言う内容の *debian/source/format* を作成することで新規の 3.0 (native) ソースフォーマットに更新できます。残りの *debian/* ファイルは単にコピーするだけです。

• *foo*.diff.gz ファイルありの 1.0 ソースパッケージの場合、3.0 (quilt) と言う内容の *debian/source/format* を作成することで新規の 3.0 (quilt) ソースフォーマットに更新できます。残りの *debian/* ファイルは単にコピーするだけです。必要な場合、filterdiff -z -x ‘*/debian/**’ foo.diff.gz > big.diff コマンドにより生成される big.diff ファイルをあなたの quilt システムにインポートします。

• -p0 や -p1 や -p2 を使う dpatch や dbs や cdbs や dpatch のような他のパッチシステムを用いてパッケージされ得ている場合には、http://bugs.debian.org/581186 (http://bugs.debian.org/581186) にある deb3 を用いて quilt コマンドに変換します。

*DEP - Debian エンハンスメント提案 (http://dep.debian.net/) を確認し、ACCEPTED (採用) となった提案を取り入れるべきです。

項8.3 に記載された他の作業も行う必要があります。

### 8.5 UTF-8 変換

アップストリームの文書が旧来のエンコーディング法にてエンコードされている場合には、それらを UTF-8 に変換するのはいいことです。

```bash
• iconv(1) を使いプレインテキストのエンコーディングを変換します。

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

• w3m(1) を使用して HTML ファイルを UTF-8 のプレインテキストファイルに変換します。これを行う際には、必ず UTF-8 ロケールで実行して下さい。

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

### 8.6 パッケージをアップグレードする際の注意点

パッケージをアップグレードする際の注意点は以下です:

• changelog の旧項目を保全します (自明ですが、dch -i とするべき時に dch とした過去事例があります。)

**splitdiff** コマンドを用いると big.diff は多くの小さな増分パッチに分割できます。
• 現存の Debian による変更は再評価する必要があります。(何らかの形で) アップストリームが組み込んだことは捨て、アップストリームが組み込まなかったことは残しましょう。

• ビルドシステムに変更が加えられた場合には(アップストリームの変更を精査した際に分からないはずです), 必要に応じて debian/rules と debian/control のビルド依存関係を更新します。

• 現存もオープンのバグへのパッチを誰かが提供していないかを, Debian Bug Tracking System (BTS) (http://www.debian.org/-Bugs/) で確認します。

• 正しいディストリビューションへアップロードすること、Closes フィールドに適切なバグのクローズがリストされていること、Maintainer と Changed-By フィールドがマッチしていること、ファイルが GPG 署名されていること等を確実にするように, .changes ファイルの内容を確認します。
Chapter 9

パッケージをアップロードする


Debian now requires source-only uploads for normal upload. So this page is outdated.
あなたの新しいパッケージは徹底的にテストできたので、あなたはそれを共有すべく公開アーカイブにリリースしたいでしょう。

9.1 Debian アーカイブへアップロードする

正規デベロッパー1になると、パッケージを Debian アーカイブにアップロードできます。2 手動でもできますが、
dupload(1) や dpup(1) 等の既存の自動化ツールを用いる方が楽です。ここでは dupload を使ってどうするかを説明します。3

まず dupload の設定ファイルを調整しなければいけません。システム全体の設定ファイルである /etc/dupload. confを編集するか、あるいはあなたの専用の設定ファイルである ~/.dupload.conf を使って変更したい項目だけをオーバーライドさせてもかまいません。

またそれぞれのオプションが持つ意味を理解するため dupload.conf(5) マニュアルページを読むことができます。

$default_host オプションはデフォルトで利用されるアップロードキーを指定します。anonymous-ftp-master がメインのアップロードキーですが、他のアップロードキーを利用したいこともあるでしょう。4

インターネットにつながった状態で、以下のようすればあなたのパッケージをアップロード出来ます:

```
$ dupload gentoo_0.9.12-1_i386.changes
```
donload は各ファイルの SHA1/SHA256 チェックサムを計算し、.changes ファイルの中の情報と照合します。もしそれらが一致しない場合には、適正にアップロードされるように項6.1 の説明に従って最初から再ビルドをするよう警告します。


1 9.1.1を参照下さい。


3 dput パッケージは、より多くの機能があり dupload パッケージより人気が出てきています。それは、/etc/dput ファイルをグローバル設定に用い、/root/.dput.cf ファイルをユーザー毎の設定に用います。更にそれは、そのまま Ubuntu 関連のサービスもサポートします。

4 Debian Developer's Reference 5.6, "Uploading a package" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#upload) を参照下さい。

9.2 アップロード用 orig.tar.gz の内容

初めてパッケージをアーカイブにアップロードする際は、オリジナルのorig.tar.gz ソースファイルを含めなければならない。当該パッケージの Debian リビジョン番号が1でも0でも無い場合には、dpkg-buildpackage に -sa オプションを付けなければならない。

dpkg-buildpackage コマンドの場合:

$ dpkg-buildpackage -sa

debuild コマンドの場合:

$ debuild -sa

debuild コマンドの場合:

$ pdebuild --debuildopts -sa

逆に、-sd オプションを付けると、オリジナルのorig.tar.gz ソースファイルを強制的に除外します。

9.3 スキップされたアップロード

アップロードをスキップすることで debian/changelog 中に複数の項目を作成した場合は、前回アップロード以来の全ての変更を含む適切な *_.changes ファイルを作成しなければいけません。dpkg-buildpackage オプションの -v を、例えば 1.2 と言うバージョンに関して指定すると可能です。

dpkg-buildpackage コマンドの場合:

$ dpkg-buildpackage -v1.2

debuild コマンドの場合:

$ debuild -v1.2

debuild コマンドの場合:

$ pdebuild --debuildopts ”-v1.2”
Appendix A

上級パッケージング


あなたが出会いそうな上級パッケージング課題に関するヒントや外部参照をいくつか記します。ここに提案されたレファレンス全てに目を通すことを切にお薦めします。

本章で取り上げたトピックをカバーするには dh_make コマンドで生成されたパッケージ用テンプレートファイルではマニュアル編集する必要があるかもしれませんが。より新しい debmake コマンドはこのようなトピックへの対応が優れています。

A.1 共有ライブラリー

共有ライブラリーをパッケージングする前に、以下の一次レファレンスを詳細に読むべきです:


以下はあなたが手を付け始めるための少々簡略化していただいたヒントです:

- 共有ライブラリーとはコンパイルされたコードを含む ELF オブジェクトファイルです。
- 共有ライブラリーは *.so ファイルとして提供されます。(*.a ファイルでも *.1a ファイルでもありません)
- 主に、共有ライブラリーは ld メカニズムを用いて多数の実行プログラム間でコードを共有するのに使用されます。
- 時々、共有ライブラリーは dlopen メカニズムを用いてある実行プログラムに多数のプラグインを提供するのに使用されます。
- 共有ライブラリーは変数や関数やクラスのようなコンパイルされたオブジェクトを表すシンボルをエキスポートし、リンクされた実行プログラムからそれらへのアクセスを可能とします。
- 共有ライブラリー libfoo.so.1 の SONAME: objdump -p libfoo.so.1 | grep SONAME 1
- 共有ライブラリーの SONAME は通常ライブラリーのファイル名と一致します (例外もあります)。
- /usr/bin/foo にリンクされた共有ライブラリーの SONAME: objdump -p /usr/bin/foo | grep NEEDED 2

1もしくは: readelf -d libfoo.so.1 | grep SONAME
2もしくは: readelf -d libfoo.so.1 | grep NEEDED
• `libfoo1`: 共有ライブラリー `libfoo.so.1` で SONAME ABI バージョンが 1 のライブラリーパッケージ。

• ライブラリーパッケージのパッケージメンテナースクリプトは SONAME に必要なシンボリックリンクを作成するために特定の環境下で `ldconfig` を呼ばなければいけません。

• `libfoo1-dbgsym`: 共有ライブラリー `libfoo1` のデバッグシンボルを含むデバッグシンボルパッケージ。

• `libfoo-dev`: 共有ライブラリー `libfoo.so.1` 用のヘッダーファイルを含む開発用パッケージ。

• Debian パッケージは一般的に `*.la` の `Libtool` アーカイブファイルを含まないでください。

• Debian パッケージは一般的に `RPATH` を使うべきではありません。


A.2 `debian/package.symbols` の管理

共有ライブラリーをパッケージする際には、同一共有ライブラリーパッケージ名のライブラリーの同一 SONAME の下での後方互換性のある変更に関して、各シンボルと関連付けられる最小のバージョンが記された `debian/package.symbols` ファイルを作成すべきです。以下の一次的レファレンスを詳細に読むべきです:

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

過去バージョン 1.3 がアップストリームバージョンから適切な `debian/libfoo1.symbols` ファイルを用いて `libfoo1` パッケージを作成する概略例は以下です:

- アップストリームの `libfoo-1.3.tar.gz` ファイルを用いて Debian 化したソースツリーの骨子を準備します。
  - もし `libfoo1` パッケージの最初のパッケージの場合は、内容が空の `debian/libfoo1.symbols` ファイルを作成します。
  - もし以前のアップストリームバージョン 1.2 が `libfoo1` パッケージとして適切な `debian/libfoo1.symbols` を用いてパッケージされていた場合には、それを再び使いましょう。
  - もし過去のアップストリームバージョン 1.2 が `debian/libfoo1.symbols` を用いてパッケージされていない場合には、そのライブラリーの同一 SONAME を含む同一共有ライブラリーパッケージ名の全ての手に入るバイナリーパッケージ、例えば 1.1-1 と 1.2-1 バージョンから、それを `symbols` として生成できます。

---

1 Debian Policy Manual, 8.1 "Run-time shared libraries" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-runtime) を参照下さい。
2 Debian Policy Manual, 8.1.1 "ldconfig" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-ldconfig) を参照下さい。
5 Debian wiki RpathIssue (http://wiki.debian.org/RpathIssue) を参照下さい。
6 前方非互換な ABI 変更をした場合、通常、ライブラリーの SONAME と共有ライブラリーパッケージ名を新規なものにそれぞれアップデートしないといけません。
7 C++ ライブラリーや個別シンボルを追跡するのが非常に困難な他の場合には、これに代えて Debian Policy Manual, 8.6.4 "The shlibs system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) に従いましょう。
8 ディプロファイル（過去のバージョンを含む）パッケージは http://snapshot.debian.org/ から得られます。パッケージのバージョンは v1.1 << v1.1-1~bpo70+1 << v1.1-1 と v1.2 << v1.2-1~bpo70+1 << v1.2-1 に従います。
$ 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

- `debuild` や `pdebuild` 等のツールを使ってソースツリーのテストビルドをします。(もしシンボルの欠如等でビルドがうまく行かない場合には、共有ライブラリーパッケージ名を `libfoo1a` 等と繰り上げる必要のある何らかの後方非互換な ABI 変更があったので、最初からやり直す必要があります。)

$ 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
...

- もし上記のように `dpkg-gensymbols` によって diff がプリントされるのを発見した場合には、生成された共有ライブラリーバイナリーパッケージから適切な symbols ファイルを抽出しましょう。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

- `debuild` や `pdebuild` やのツールでリリースパッケージをビルドします。

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

上記の例に加えて、更に ABI 互換性を確認し、必要に応じていくつかのシンボルのバージョンを手作業で繰り上げる必要があります。12

二次的なレファレンスではありますが、Debian wiki UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) とそこからリンクされているウェブページは有用かもしれません。

### A.3 マルチアーチ

Debian wheezy で導入されたマルチアーチ機能は dpkg と apt の中でのバイナリーパッケージのアーキテクチャ間サポート (他の組み合わせもありますが、特に i386<->amd64) を統合します。以下のレファレンスを詳細に読んで下さい:

11Debian リビジョンはパッケージのバックポートを容易にすべく外します: 1.3 << 1.3-1-bpo70+1 << 1.3-1
マルチアーチは共有ライブラリーのインストールパスに i386-linux-gnu や x86_64-linux-gnu 等のトリプレットを使います。実際のトリプレットパスは、ビルドごとに dpkg-architecture(1) によって動的に $(DEB_HOST_MULTIARCH) 値として設定されます。例えば、マルチアーチライブラリーをインストールするパスは以下のように変更されます。

<table>
<thead>
<tr>
<th>旧パス</th>
<th>i386 サンプルパス</th>
<th>amd64 サンプルパス</th>
</tr>
</thead>
<tbody>
<tr>
<td>/lib/</td>
<td>/lib/i386-linux-gnu/</td>
<td>/lib/x86_64-linux-gnu/</td>
</tr>
<tr>
<td>/usr/lib/</td>
<td>/usr/lib/i386-linux-gnu/</td>
<td>/usr/lib/x86_64-linux-gnu/</td>
</tr>
</tbody>
</table>

以下の場合に関する典型的なパッケージ分割シナリオの例を示します。

- ライブラリーソース libfoo-1.tar.gz
- コンパイラー用の言語で書かれたツールのソース bar-1.tar.gz
- インタープリター用言語で書かれたツールのソース baz-1.tar.gz

<table>
<thead>
<tr>
<th>パッケージ</th>
<th>Architecture:</th>
<th>Multi-Arch:</th>
<th>パッケージ内容</th>
</tr>
</thead>
<tbody>
<tr>
<td>libfoo1</td>
<td>any</td>
<td>same</td>
<td>共有ライブラリー、同時インストール可能</td>
</tr>
<tr>
<td>libfoo1-dbgs</td>
<td>any</td>
<td>same</td>
<td>共有ライブラリーデバッグシンボル、同時インストール可能</td>
</tr>
<tr>
<td>libfoo-dev</td>
<td>any</td>
<td>same</td>
<td>共有ライブラリーヘッダーファイル、同時インストール可能</td>
</tr>
<tr>
<td>libfoo-tools</td>
<td>any</td>
<td>foreign</td>
<td>実行時サポートプログラム、同時インストール不可</td>
</tr>
<tr>
<td>libfoo-doc</td>
<td>all</td>
<td>foreign</td>
<td>共有ライブラリーのドキュメンテーションファイル</td>
</tr>
<tr>
<td>bar</td>
<td>any</td>
<td>foreign</td>
<td>コンパイルされ実行されるプログラムファイル、同時インストール不可</td>
</tr>
<tr>
<td>bar-doc</td>
<td>all</td>
<td>foreign</td>
<td>プログラムのドキュメンテーションファイル</td>
</tr>
<tr>
<td>baz</td>
<td>all</td>
<td>foreign</td>
<td>インタープリターで実行されるプログラムファイル</td>
</tr>
</tbody>
</table>

開発パッケージは関連した共有ライブラリーへのバージョン番号無しのシンボリックリンクを含んでいるべきです。例えば: `/usr/lib/x86_64-linux-gnu/libfoo.so -> libfoo.so.1`

### A.4 共有ライブラリーパッケージのビルド

dh(1) を以下のようにすればマルチアーチをサポートするようににして Debian のライブラリーパッケージをビルドできます。

- `debian/control` を更新します。
  - ソースパッケージセクションに Build-Depends: debhelper (>=10) を追加します。
  - 共有ライブラリーのバイナリーパッケージごとに Pre-Depends: ${misc:Pre-Depends} を追加します。
  - Multi-Arch: スタンダードをバイナリーパッケージセクション毎に追加します。

- `debian/compat` を "10" と設定します。
- すべてのパッケージングスクリプトに関して、通常の `/usr/lib/` から、マルチアーチの一覧 `$DEB_HOST_MULTIARCH` へとパスを調整します。

13旧来の `/lib32/` や `/lib64` 等の特定目的のライブラリーパスは使われなくなっています。
- 最初にdebian/rules中でDEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)
を呼びDEB_HOST_MULTIARCH変数を設定します。
- debian/rules中の/usr/lib/を/usr/lib/${DEB_HOST_MULTIARCH}/で置き換えます。
- もしdebian/rules中のoverride_dh_auto_configureターゲットの一部分として./configureが用いられている場合には、それをdh_auto_configure -- と置き換えるようにしましょう。
- debian/foo.installファイル中にある全ての/usr/lib/を/usr/lib/*/で置き換えます。
- debian/rules中のoverride_dh_auto_configureターゲットにスクリプトを追加しdebian/foo.links.inからdebian/foo.linksを動的に生成します。

override_dh_auto_configure:
  dh_auto_configure
  sed 's/@DEB_HOST_MULTIARCH@/${DEB_HOST_MULTIARCH}/g'
  debian/foo.links.in > debian/foo.links

共有ライブラリーパッケージが期待されるファイルのみを含み、-devバッケージも依然として動作することを確認しましょう。
マルチアーチパッケージとして同時に同一パスにインストールされる全てのファイルはファイルの内容が完全同じあるべきです。データのバイトオーダーや圧縮アルゴリズムにより生成される相違に注意すべきです。

A.5 ネイティブDebianパッケージ

もしパッケージがDebianのためだけとか、またローカル使用のために保守されている場合、そのソースファイルはdebian/*ファイルすべてをその中に含めます。それをパッケージするのは2つの方法があります。

debian/*ファイルを除外したアップストリームターボールを作成し、項2.1にあるようにノンネイティブDebianパッケージできます。これが一部の人に推奨される通常の方法です。
この代わりの方法は、ネイティブDebianパッケージのワークフローです。

- 全てのファイルが含まれる単一の圧縮されたtarファイルを用いる3.0(native)フォーマットでネイティブのDebianソースパッケージを作成します。
  - package_version.tar.gz
  - package_version.dsc

- ネイティブDebianソースパッケージから、Debianバイナリーパッケージをビルドします。
  - package_version_arch.deb

例えばdebian/*ファイルを含まない~/mypackage-1.0ソースファイルがあれば、以下のようにdh_makeコマンドを用いてネイティブDebianパッケージが作れます:

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

すると、debianディレクトリーとその内容は項2.8とちょうど同じように作成されます。これはネイティブDebianパッケージなのでtarアーカイブを作りません。しかし相違点はこれだけです。他のパッケージング操作は実質的にまったく同じです。

dpkg-buildpackageコマンドを実行した後、親ディレクトリーに以下のファイルが生成します:

14これに代えて、./configureに--libdir='$${prefix}/lib/${DEB_HOST_MULTIARCH}'と--libexecdir='$${prefix}/lib/${DEB_HOST_MULTIARCH}'引数を追加することもできます。注意いただきたいのは、ユーザーではなく他のプログラムによりのみ実行される実行プログラムをインストールするデフォルトパスを--libexecdirは設定します。そのAutotoolsのデフォルトは/usr/libexec/ですが、Debianのデフォルトは/usr/lib/です。
• mypackage_1.0.tar.gz
  これは、dpkg-source コマンドにより mypackage-1.0 ディレクトリーから作られたソースコードのターボールです。（そのサフィックスは orig.tar.gz ではありません。）

• mypackage_1.0.dsc
  これは、ノンネイティブ Debian パッケージと同様でソースコード内容の要約です。（Debian リビジョンはありません。）

• mypackage_1.0_i386.deb
  これは、ノンネイティブ Debian パッケージと同様で完成したバイナリーパッケージです。（Debian リビジョンはありません。）

• mypackage_1.0_i386.changes
  これは、ノンネイティブ Debian パッケージと同様で現パッケージバージョンでの全変更を記述します。（Debian リビジョンはありません。）