

Debian 開発者リファレンス

[FAMILY Given], Barth Andreas [FAMILY Given], Di
Carlo Adam [FAMILY Given], Hertzog Raphaël [FAMILY
Given], Nussbaum Lucas [FAMILY Given], Schwarz
Christian [FAMILY Given], 、 Jackson Ian [FAMILY
Given]

June 28, 2016

Debian 開発者リファレンス

by [FAMILY Given], Barth Andreas [FAMILY Given], Di Carlo Adam [FAMILY Given], Hertzog Raphaël [FAMILY Given], Nussbaum Lucas [FAMILY Given], Schwarz Christian [FAMILY Given], 、 Jackson Ian [FAMILY Given]

Published 2016-06-19

製作著作 © 2004, 2005, 2006, 2007 Andreas Barth

製作著作 © 1998, 1999, 2000, 2001, 2002, 2003 Adam Di Carlo

製作著作 © 2002, 2003, 2008, 2009 Raphaël Hertzog

製作著作 © 2008, 2009 Lucas Nussbaum

製作著作 © 1997, 1998 Christian Schwarz

このマニュアルはフリーソフトウェアです。あなたは、Free Software Foundation が発行した GNU 一般公衆利用許諾契約書の第二版あるいはそれ以降のいずれかの版の条件に基づき、本文書の再配付および変更をすることができます。

本文書はその有用性が期待されて配付されるものですが、市場性や特定の目的への適合性に関する暗黙の保証も含め、いかなる保証も行ないません。詳細については GNU 一般公衆利用許諾契約書をお読みください。

GNU 一般公衆利用許諾契約書の写しは、Debian GNU/Linux ディストリビューション中の `/usr/share/common-licenses/GPL-2`、あるいは World Wide Web 上の [GNU ウェブサイト](#) で入手できます。また Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA へ手紙 (英語) で依頼し入手することもできます。

このリファレンスを印刷したい場合は、[PDF 版](#)を利用すると良いでしょう。このページは [フランス語](#)、[ドイツ語](#)、[イタリア語](#)、[ロシア語](#)、[日本語](#)でも利用可能です。

Contents

1	この文書が扱う範囲について	1
2	開発者になるために応募する	3
2.1	さあ、はじめよう	3
2.2	Debian メンター (mentors) とスポンサー (sponsors) について	3
2.3	Debian 開発者への登録を行う	4
3	Debian 開発者の責務	7
3.1	パッケージメンテナの責務	7
3.1.1	次期安定版 (stable) リリースへの作業	7
3.1.2	安定版 (stable) にあるパッケージをメンテナンスする	7
3.1.3	リリースクリティカルバグに対処する	7
3.1.4	開発元/上流 (upstream) の開発者との調整	8
3.2	管理者的な責務	8
3.2.1	あなたの Debian に関する情報をメンテナンスする	8
3.2.2	公開鍵をメンテナンスする	8
3.2.3	投票をする	9
3.2.4	優雅に休暇を取る	9
3.2.5	脱退について	9
3.2.6	リタイア後に再加入する	10
4	Debian 開発者および Debian メンテナが利用可能なリソース	11
4.1	メーリングリスト	11
4.1.1	利用の基本ルール	11
4.1.2	開発の中心となっているメーリングリスト	11
4.1.3	特別なメーリングリスト	12
4.1.4	新規に開発関連のメーリングリストの開設を要求する	12
4.2	IRC チャンネル	12
4.3	ドキュメント化	13
4.4	Debian のマシン群	13
4.4.1	バグ報告サーバ	13
4.4.2	ftp-master サーバ	13
4.4.3	www-master サーバ	14
4.4.4	people ウェブサーバ	14
4.4.5	VCS (バージョン管理システム) サーバ	14
4.4.6	複数のディストリビューション利用のために chroot を使う	14
4.5	開発者データベース	14
4.6	Debian アーカイブ	15
4.6.1	セクション	16
4.6.2	アーキテクチャ	16
4.6.3	パッケージ	17
4.6.4	ディストリビューション	17
4.6.4.1	安定版 (stable)、テスト版 (testing)、不安定版 (unstable)	17
4.6.4.2	テスト版ディストリビューションについてのさらなる情報	18
4.6.4.3	試験版 (experimental)	18
4.6.5	リリースのコードネーム	19
4.7	Debian ミラーサーバ	19
4.8	Incoming システム	19
4.9	パッケージ情報	20
4.9.1	ウェブ上から	20
4.9.2	dak ls ユーティリティ	20
4.10	Debian パッケージトラッカー	21
4.10.1	パッケージトラッカーのメールインターフェイス	21
4.10.2	パッケージトラッカーからのメールを振り分ける	22

4.10.3	パッケージトラッカーへ VCS コミットを転送する	22
4.10.4	パッケージトラッカーのウェブインターフェイス	23
4.11	Developer's packages overview	23
4.12	Debian での FusionForge の導入例: Alioth	23
4.13	Debian 開発者と Debian メンテナへの特典	23
5	パッケージの取扱い方	25
5.1	新規パッケージ	25
5.2	パッケージの変更を記録する	26
5.3	パッケージをテストする	26
5.4	ソースパッケージの概要	27
5.5	ディストリビューションを選ぶ	27
5.5.1	特別な例: 安定版 (stable) と 旧安定版 (oldstable) ディストリビューションへアップロードする	27
5.5.2	特別な例: testing/testing-proposed-updates へアップロードする	28
5.6	パッケージをアップロードする	28
5.6.1	ftp-master にアップロードする	28
5.6.2	遅延アップロード	28
5.6.3	セキュリティアップロード	29
5.6.4	他のアップロードキュー	29
5.6.5	新しいパッケージがインストールされたことの通知	29
5.7	パッケージのセクション、サブセクション、優先度を指定する	29
5.8	バグの取扱い	29
5.8.1	バグの監視	30
5.8.2	バグへの対応	30
5.8.3	バグを掃除する	30
5.8.4	新規アップロードでバグがクローズされる時	32
5.8.5	セキュリティ関連バグの取扱い	32
5.8.5.1	セキュリティ追跡システム	33
5.8.5.2	秘匿性	33
5.8.5.3	セキュリティ勧告	34
5.8.5.4	セキュリティ問題に対処するパッケージを用意する	34
5.8.5.5	修正したパッケージをアップロードする	35
5.9	パッケージの移動、削除、リネーム、放棄、引き取り、再導入	36
5.9.1	パッケージの移動	36
5.9.2	パッケージの削除	36
5.9.2.1	Incoming からパッケージを削除する	37
5.9.3	パッケージをリプレースあるいはリネームする	37
5.9.4	パッケージを放棄する	37
5.9.5	パッケージを引き取る	38
5.9.6	パッケージの再導入	38
5.10	移植作業、そして移植できるようにすること	38
5.10.1	移植作業に対して協力的になる	39
5.10.2	移植作業者のアップロード (porter upload) に関するガイドライン	40
5.10.2.1	再コンパイル、あるいは binary-only NMU	40
5.10.2.2	あなたが移植作業者の場合、source NMU を行う時は何時か	40
5.10.3	移植用のインフラと自動化	41
5.10.3.1	メーリングリストとウェブページ	41
5.10.3.2	移植用ツール	41
5.10.3.3	wanna-build	41
5.10.4	あなたのパッケージが移植可能なものではない場合	42
5.10.5	non-free のパッケージを auto-build 可能であるとマークする	42
5.11	Non-Maintainer Upload (NMU)	42
5.11.1	いつ、どうやって NMU を行うか	42
5.11.2	NMU と debian/changelog	44
5.11.3	DELAYED/ キューを使う	44
5.11.4	メンテナの視点から見た NMU	44
5.11.5	ソース NMU vs バイナリのための NMU (binNMU)	45
5.11.6	NMU と QA アップロード	45

5.11.7	NMU とチームアップロード	45
5.12	共同メンテナンス	45
5.13	テスト版ディストリビューション	46
5.13.1	基本	46
5.13.2	不安定版からの更新	46
5.13.2.1	時代遅れ (Out-of-date)	47
5.13.2.2	テスト版からの削除	47
5.13.2.3	循環依存	48
5.13.2.4	テスト版 (testing) にあるパッケージへの影響	48
5.13.2.5	詳細について	48
5.13.3	直接テスト版を更新する	48
5.13.4	よく聞かれる質問とその答え (FAQ)	49
5.13.4.1	リリースクリティカルバグとは何ですか、どうやって数えるのですか?	49
5.13.4.2	どのようにすれば、他のパッケージを壊す可能性があるパッケージをテスト版 (testing) ヘインストールできますか?	49
6	パッケージ化のベストプラクティス	51
6.1	debian/rules についてのベストプラクティス	51
6.1.1	ヘルパースクリプト	51
6.1.2	パッチを複数のファイルに分離する	52
6.1.3	複数のバイナリパッケージ	52
6.2	debian/control のベストプラクティス	52
6.2.1	パッケージ説明文の基本的なガイドライン	52
6.2.2	パッケージの概要、あるいは短い説明文	53
6.2.3	長い説明文 (long description)	53
6.2.4	開発元のホームページ	54
6.2.5	バージョン管理システムの場所	54
6.2.5.1	Vcs-Browser	54
6.2.5.2	Vcs-*	54
6.3	debian/changelog のベストプラクティス	55
6.3.1	役立つ changelog のエントリを書く	55
6.3.2	changelog のエントリに関するよくある誤解	55
6.3.3	changelog のエントリ中のよくある間違い	55
6.3.4	NEWS.Debian ファイルで changelog を補足する	56
6.4	メンテナスクリプトのベストプラクティス	56
6.5	debconf による設定管理	57
6.5.1	debconf を乱用しない	57
6.5.2	作者と翻訳者に対する雑多な推奨	58
6.5.2.1	正しい英語を書く	58
6.5.2.2	翻訳者へ丁寧に接する	58
6.5.2.3	誤字やミススペルを修正する際に fuzzy を取る	58
6.5.2.4	インターフェイスについて仮定をしない	59
6.5.2.5	一人称を使わない	59
6.5.2.6	性差に対して中立であれ	59
6.5.3	テンプレートのフィールド定義	60
6.5.3.1	Type	60
6.5.3.1.1	string	60
6.5.3.1.2	password	60
6.5.3.1.3	boolean	60
6.5.3.1.4	select	60
6.5.3.1.5	multiselect	60
6.5.3.1.6	note	60
6.5.3.1.7	text	60
6.5.3.1.8	error	60
6.5.3.2	Description: short および extended 説明文	60
6.5.3.3	Choices	61
6.5.3.4	Default	61
6.5.4	テンプレートのフィールド固有スタイルガイド	61
6.5.4.1	Type フィールド	61

6.5.4.2	Description フィールド	61
6.5.4.2.1	String/password テンプレート	61
6.5.4.2.2	Boolean テンプレート	61
6.5.4.2.3	Select/Multiselect	62
6.5.4.2.4	Note	62
6.5.4.3	Choices フィールド	62
6.5.4.4	Default フィールド	62
6.5.4.5	Default フィールド	62
6.6	国際化	63
6.6.1	debconf の翻訳を取り扱う	63
6.6.2	ドキュメントの国際化	63
6.7	パッケージ化に於ける一般的なシチュエーション	63
6.7.1	autoconf/automake を使っているパッケージ	63
6.7.2	ライブラリ	64
6.7.3	ドキュメント化	64
6.7.4	特定の種類のパッケージ	64
6.7.5	アーキテクチャ非依存のデータ	65
6.7.6	ビルド中に特定のロケールが必要	65
6.7.7	移行パッケージを deboprhon に適合するようにする	65
6.7.8	.orig.tar.{gz,bz2,xz} についてのベストプラクティス	65
6.7.8.1	手が入れていないソース (Pristine source)	65
6.7.8.2	upstream のソースをパッケージしなおす	66
6.7.8.3	バイナリファイルの変更	67
6.7.9	デバッグパッケージのベストプラクティス	67
6.7.10	メタパッケージのベストプラクティス	67
7	パッケージ化、そして…	69
7.1	バグ報告	69
7.1.1	一度に大量のバグを報告するには (mass bug filing)	69
7.1.1.1	Usertag	70
7.2	品質維持の努力	70
7.2.1	日々の作業	70
7.2.2	バグ潰しパーティ (BSP)	70
7.3	他のメンテナに連絡を取る	71
7.4	活動的でない、あるいは連絡が取れないメンテナに対応する	71
7.5	Debian 開発者候補に対応する	72
7.5.1	パッケージのスポンサーを行う	72
7.5.1.1	新しいパッケージのスポンサーを行う	73
7.5.1.2	既存パッケージの更新をスポンサーする	74
7.5.2	新たな開発者を支持する (advocate)	74
7.5.3	新規メンテナ申請 (new maintainer applications) を取り扱う	74
8	国際化と翻訳	75
8.1	どの様にして Debian では翻訳が取り扱われているか	75
8.2	メンテナへの I18N & L10N FAQ	76
8.2.1	翻訳された文章を得るには	76
8.2.2	どの様にして提供された翻訳をレビューするか	76
8.2.3	どの様にして翻訳してもらった文章を更新するか	76
8.2.4	どの様にして翻訳関連のバグ報告を取り扱うか	76
8.3	翻訳者への I18N & L10N FAQ	76
8.3.1	どの様にして翻訳作業を支援するか	76
8.3.2	どの様にして提供した翻訳をパッケージに含めてもらうか	77
8.4	I10n に関する現状でのベストプラクティス	77

A Debian メンテナツールの概要	79
A.1 主要なツール	79
A.1.1 dpkg-dev	79
A.1.2 debconf	79
A.1.3 fakeroot	79
A.2 パッケージチェック (lint) 用ツール	80
A.2.1 lintian	80
A.2.2 debdiff	80
A.3 debian/rules の補助ツール	80
A.3.1 debhelper	80
A.3.2 dh-make	80
A.3.3 equivs	81
A.4 パッケージ作成ツール	81
A.4.1 git-buildpackage	81
A.4.2 debootstrap	81
A.4.3 pbuilder	81
A.4.4 sbuild	81
A.5 パッケージのアップロード用ツール	81
A.5.1 dupload	81
A.5.2 dput	82
A.5.3 dcut	82
A.6 メンテナンスの自動化	82
A.6.1 devscripts	82
A.6.2 autotools-dev	82
A.6.3 dpkg-repack	82
A.6.4 alien	82
A.6.5 dpkg-dev-el	82
A.6.6 dpkg-depcheck	83
A.7 移植用ツール	83
A.7.1 dpkg-cross	83
A.8 ドキュメントと情報について	83
A.8.1 docbook-xml	83
A.8.2 debiandoc-sgml	83
A.8.3 debian-keyring	83
A.8.4 debian-el	83

Chapter 1

この文書が扱う範囲について

この文書の目的は、Debian 開発者に推奨される手続きと利用可能なリソースに関する概要を提供することにあります。

ここで取り上げる手続きには、開発者になる方法 (第2章)、新しいパッケージの作り方 (項5.1) とパッケージをアップロードする方法 (項5.6)、バグ報告の取扱い方 (項5.8)、パッケージを移動、削除、放棄 (orphan) する方法 (項5.9)、パッケージ移植のやり方 (項5.10) 他のメンテナのパッケージを暫定的にリリースするのは、いつどの様にしたらよいのか (項5.11) が含まれます。

また、このリファレンスで触れるリソースには、メーリングリスト (項4.1) およびサーバ (項4.4)、Debian アーカイブの構成に関する解説 (項4.6)、パッケージのアップロードを受け付ける様々なサーバの説明 (項5.6.1)、パッケージの品質を高めるために開発者が利用できるリソースについての解説 (付録A) などがあります。

初めに明らかにしておきたいのですが、このリファレンスは Debian パッケージに関する技術的な詳細や、Debian パッケージの作成方法を説明するものではありません。また、このリファレンスは Debian に含まれるソフトウェアが準拠すべき基準を詳細に解説するようなものでもありません。その様な情報については全て、[Debian ポリシーマニユアル](#) に記述されています。

さらに、この文書は公式なポリシーを明らかにするものではありません。含まれているのは Debian システムに関する記述と、一般的な合意がなされたベストプラクティスに関する記述です。すなわち「規範」文書と呼ばれるものではない、ということです。

Chapter 2

開発者になるために応募する

2.1 さあ、はじめよう

さて、あなたは全てのドキュメントを読み終え、[Debian 新メンテナガイド](#)を通して例題の hello パッケージがどの様になっているのか全てを理解し、お気に入りのソフトウェアの一つを Debianize (Debian パッケージ化) しようとしているところです。実際のところ、どうやったら Debian 開発者になってプロジェクトに成果を受け入れてもらえるようになるのでしょうか？

まず始めに、まだであれば debian-devel@lists.debian.org を購読しましょう。subscribe という単語をサブジェクトに書いた debian-devel-REQUEST@lists.debian.org へのメールを送ってください。何か問題がある場合は、listmaster@lists.debian.org のメーリングリスト管理者に確認をとりましょう。メーリングリストについての詳しい情報は項4.1 で見つけられます。そして debian-devel-announce@lists.debian.org は、Debian の開発に携わりたいという人は誰でも読む義務がある、もう一つのメーリングリストです。

参加後、何かコーディングを始める前に、しばらくの間「待ち」(投稿せずに読むだけ) の状態でいるのが良いでしょう。それから、重複作業を避けるために何の作業をしようとしているのか表明をする必要があります。

もう一つ、購読すると良いのが debian-mentors@lists.debian.org です。詳細は項2.2 を参照してください。IRC チャンネル #debian も役に立つでしょう。項4.2 を見てください。

何らかの方法で Debian に対して貢献したいと思った時、同じような作業に従事している既存の Debian メンテナにコンタクトしてみてください。そうすれば経験豊かな開発者から学ぶことができます。例えば、既にあるソフトウェアを Debian 用にパッケージ化するのに興味を持っている場合、スポンサーを探しましょう。スポンサーはあなたと一緒にパッケージ化作業を手伝い、あなたの作業が満足する出来になったら Debian アーカイブにパッケージをアップロードしてくれます。スポンサーは、debian-mentors@lists.debian.org メーリングリストへパッケージとあなた自身の説明とスポンサーを探していることをメールして見つけましょう (詳細については項7.5.1 および <https://wiki.debian.org/DebianMentorsFaq> を参照)。さらに、Debian を他のアーキテクチャやカーネルへ移植するのに興味を持っている場合、移植関連のメーリングリストに参加して、どうやって始めればいいのかを尋ねましょう。最後に、ドキュメントや品質保証 (Quality Assurance, QA) の作業に興味がある場合は、この様な作業を行っているメンテナ達の集まりに参加して、パッチや改善案を送ってください。

メールアドレスのローカルパートが非常に一般的な場合、落とし穴にハマる可能性があります。mail、admin、root、master のような単語は使わないようにすべきです。詳しくは <https://www.debian.org/MailingLists/> を参照してください。

2.2 Debian メンター (mentors) とスポンサー (sponsors) について

メーリングリスト debian-mentors@lists.debian.org が、パッケージ化の第一歩目や他の開発者と調整が必要な問題などで手助けを必要としている新米メンテナに用意されています。新たな開発者は皆、このメーリングリストに参加することをお勧めします (詳細は項4.1 を参照してください)。

一対一での指導 (つまり、プライベートなメールのやり取りで) の方が良い、という人もこのメーリングリストに投稿しましょう。経験豊かな開発者が助けになってくれるはずです。

付け加えておくと、Debian にパッケージを含める準備が出来ているものの、新規メンテナ応募作業の完了を待っているような場合は、パッケージをアップロードしてくれるスポンサーを探すことも出来ます。スポンサーは公式 Debian 開発者で、あなたのパッケージに対して問題を探したりアップロー

ドしようとしてくれる人達です。まずは <https://wiki.debian.org/DebianMentorsFaq> にある debian-mentors FAQ を読んでください。

メンターあるいはスポンサーになりたいという人は、項7.5 でより詳細な情報が手に入ります。

2.3 Debian 開発者への登録を行う

Debian への登録を決める前に、**新メンテナのコーナー**で手に入る全ての情報に目を通してください。Debian 開発者になるための登録をする前に必要な準備がすべて記載されています。例えば、応募の前に **Debian 社会契約**を読む必要があります。開発者として登録することは、Debian 社会契約に同意し、遵守し続けることを意味します。メンテナにとって Debian の背景にある根本的な理念に従うのは大変重要な事です。**GNU Manifesto**を読むのも良い考えでしょう。

開発者としての登録手順は、あなたのアイデンティティと意図と技術的なスキルレベルの確認手順でもあります。Debian について作業をする人の数は 1000 を越えて増えつづけています。そして、我々のシステムは幾多の重要な場所で使われており、セキュリティ侵害には十分に注意しなければなりません。つまり、新たなメンテナに我々のサーバのアカウントを与えたりパッケージのアップロードを許可したりする前には審査する必要があるのです。

実際に登録する前に、あなたは良い仕事ができる貢献者となり得ることを示さねばなりません。バグ追跡システムを介してパッチを送ったり、既存の Debian 開発者のスポンサーによるパッケージの管理をしばらくの間行なうなどして、これをアピールします。付け加えておくと、我々は貢献してくれる人達が単に自分のパッケージをメンテナンスするだけでなく、プロジェクト全体について興味を持ってくれることを期待しています。バグについての追加情報、できればパッチの提供などによって他のメンテナを手助けできるのであれば、早速実行しましょう！

登録に際しては Debian の考え方と技術文書を充分理解している必要があります。さらに、既存のメンテナに署名をしてもらった GnuPG 鍵が必要です。まだ GnuPG 鍵に署名してもらっていない場合は、あなたの鍵に署名してくれる Debian 開発者に会いましょう。**GnuPG Key Signing Coordination page** が近くの Debian 開発者を探す手助けとなるでしょう。(近くに Debian 開発者がいない場合は、ID チェックを通過する別の方法としてケースバイケースで例外処理として扱うことも可能です。詳細については **身分証明のページ** を参照してください)。

OpenPGP 鍵をまだ持っていない場合は生成してください。全ての開発者がパッケージをアップロードする際に署名と確認の為に OpenPGP 鍵を必要とします。必ず、使っているソフトのマニュアルを読んでおいてください。何故ならば、セキュリティに直結している重要な情報を含んでいるからです。多くのセキュリティ事故は、殆どがソフトウェアの問題や高度なスパイテクニクではなく、人間が犯すミスや勘違いによるものです。公開鍵の管理についての詳細は項3.2.2 を参照してください。

Debian では GNU Privacy Guard (gnupg パッケージ、バージョン 1 以降) を標準として利用しています。他の OpenPGP 実装も同様に利用できます。OpenPGP は **RFC 2440** に準拠したオープン標準です。

Debian での開発においては、バージョン 4 の鍵の利用が求められます。**鍵の長さは少なくとも 1024 ビットが必要です**。それより短い鍵を利用する理由は何もありませんし、使った場合はあまり安全ではなくなります。¹

あなたの公開鍵が subkeys.pgp.net などの公開鍵サーバにない場合は、**新規メンテナ手順 2: 身分証明**にあるドキュメントを読んでください。このドキュメントにはどうやって公開鍵サーバに鍵を登録するのが記載されています。新規メンテナグループは、まだ登録されていない場合はあなたの公開鍵をサーバに登録します。

幾つかの国では、一般市民の暗号関連ソフトウェアの使用について制限をかけています。しかし、このことは暗号関連ソフトウェアを暗号化ではなく認証に利用する際には完全に合法であるような場

¹ バージョン 4 の鍵は、RFC 2440 で規定されているように OpenPGP 標準に適合します。GnuPG を使って鍵を作ると、鍵のタイプは常にバージョン 4 になります。PGP はバージョン 5.x からバージョン 4 の鍵を作ることもできるようになっています。別の選択肢としては、v3 の鍵 (PGP ではレガシー RSA と呼ばれます) と互換性のある pgp 2.6.x になります。

バージョン 4 (primary) 鍵は RSA アルゴリズムか DSA アルゴリズムのどちらでも利用できます。つまり、GnuPG が (1) DSA and Elgamal, (2) DSA (sign only), (5) RSA (sign only) の中からどの種類の鍵を使いたいかに質問してきても何も迷うことはありません。特別な理由がない限りは単にデフォルトを選んでください。

今ある鍵が v4 の鍵なのか v3 (あるいは v2) の鍵なのかを判断するもっとも簡単な方法はフィンガープリントを見ることです。バージョン 4 鍵のフィンガープリントは、鍵要素の一部を SHA-1 ハッシュにしたもので、通常 4 文字ごとに区切られた 40 文字の 16 進数になります。古いバージョンの鍵形式では、フィンガープリントは MD5 を使っており、2 文字ごとに区切られた 16 進数で表示されます。例えば、あなたのフィンガープリントが 5B00 C96D 5D54 AEE1 206B AF84 DE7A AF6E 94C0 9C7F のようになっている場合は、v4 鍵です。

別のやり方として、鍵をパイプで **pgpdump** に渡して Public Key Packet - Ver 4 のような出力を得るという方法もあります。

もちろん、鍵が自己署名されている必要があります (つまり、全ての鍵は所有者の ID によって署名されている必要があります。これによって、ユーザ ID の偽造 (タンパリング) を防いでいます)。最近の OpenPGP ソフトウェアは皆これを自動的に行いますが、古い鍵を持っているような場合は自分でこの署名を追加する必要があるでしょう。

合には、Debian パッケージメンテナとしての活動を妨げることはありません。あなたが認証目的にすら暗号技術の利用が制限される国に住んでいる、と言う場合は、我々に連絡をしていただければ特別な措置を講じることができます。

新規メンテナの応募をするのであれば、既存の Debian 開発者にあなたの応募をサポートしてもらう (支持者 (Advocate) になってもらう) 必要があります。ある程度の期間 Debian に対して貢献を行った後で、登録された開発者になるために応募をしたい場合、過去何ヶ月かに渡って一緒に作業をした既存の開発者に、あなたが Debian に間違いなく貢献できることを信じている旨を表明してもらう必要があります。

支持者 (Advocate) を見つけ、GnuPG 鍵に署名をしてもらい、既にある程度の間 Debian に対して貢献的な作業をしているのであれば、応募の準備は完了です。すぐに [応募のページ](#) で登録できます。登録後、支持者 (Advocate) はあなたの応募に関してその意志を確認する必要があります。支持者がこの手順を完了すると応募管理者 (Application Manager) に割り当てられます。応募管理者は、新規メンテナプロセスで必要となる手順をあなたと共に歩んでいく存在です。進捗状況については、常に [applications status board](#) のページで確認ができます。

より詳しい内容については Debian ウェブサイトの [新規メンテナのコーナー](#) を参考にしてください。実際の応募前に、新規メンテナプロセスでの必要な手順について詳しく理解しておいてください。十分に準備ができていたら、後で費やす時間を大いに減らすことができます。

Chapter 3

Debian 開発者の責務

3.1 パッケージメンテナの責務

あなたはパッケージメンテナとして、システムにうまく適合し、Debian ポリシーにしっかり則っている高品質のパッケージを提供していることでしょう。

3.1.1 次期安定版 (stable) リリースへの作業

高品質のパッケージを不安定版 (unstable) へ提供するだけでは十分ではありません。多くのユーザは、パッケージが次期安定版 (stable) の一部としてリリースされた時にのみ、その恩恵を受けるからです。ですから、パッケージが次期の安定版 (stable) に含まれるようにリリースチームと上手く共同で作業することが期待されています。

より具体的には、パッケージがテスト版 (testing) に移行しているかどうかを見守る必要があります (項5.13 参照)。テスト期間後に移行が行われない場合は、その理由を分析してこれを修正する必要があります。(リリースクリティカルバグや、いくつかのアーキテクチャでビルドに失敗する場合) あなたのパッケージを修正するのが必要な場合もありますし、依存関係でパッケージが絡まっている状態からの移行を完了する手助けとして、他のパッケージを更新 (あるいは修正、またはテスト版 (testing) からの削除) が必要な事を意味する場合もあります。障害となる理由 (blocker) を判別できない場合は、リリースチームが先の移行に関する現在の障害に関する情報を与えてくれることでしょう。

3.1.2 安定版 (stable) にあるパッケージをメンテナンスする

パッケージメンテナの作業の大半は、パッケージの更新されたバージョンを不安定版 (unstable) へ放り込むことですが、現状の安定版 (stable) リリースのパッケージの面倒をみることも伴っています。

安定版 (stable) への変更は推奨されてはいませんが、可能です。セキュリティ問題が報告された時はいつでも、セキュリティチームと修正版を提供するように協力する必要があります (項5.8.5 参照)。important (あるいはそれ以上) な重要度のバグが安定版 (stable) のバージョンのパッケージに報告されたら、対象となる修正の提供を検討する必要があります。安定版 (stable) リリースマネージャに、そのような更新を受け入れられるかどうかを尋ね、それから安定版 (stable) のアップロードを準備するなどができます (項5.5.1 参照)。

3.1.3 リリースクリティカルバグに対処する

大抵の場合、パッケージに対するバグ報告については項5.8で記述されているように対応する必要があります。しかしながら、注意を必要とする特別なカテゴリのバグがあります—リリースクリティカルバグ (RC bug) と呼ばれるものです。critical、grave、serious の重要度が付けられている全てのバグ報告によって、そのパッケージは次の安定版 (stable) リリースに含めるのには適切ではないとされます。そのため、(テスト版 (testing) にあるパッケージに影響する場合に) Debian のリリースを遅らせたり、(不安定版 (unstable) にあるパッケージにのみ影響する場合に) テスト版 (testing) への移行をブロックする可能性があります。最悪の場合は、パッケージの削除を招きます。これが RC バグを可能な限り素早く修正する必要がある理由です。

もし、何らかの理由で 2 週間以内に RC バグを修正できない場合 (例えば時間の制約上、あるいは修正が難しいなど)、明示的にバグ報告にそれを述べて、他のボランティアを招き入れて参加してもら

うためにバグに help タグを打ってください。大量のパッケージがテスト版 (testing) へ移行するのを妨げることがあるので、RC バグは頻繁に Non-Maintainer Upload の対象になることに注意してください (項5.11 参照)。

RC バグへの関心の無さは、しばしば QA チームによって、メンテナが正しくパッケージを放棄せずに消えてしまったサインとして判断されます。MIA チームが関わることもあり、その場合はパッケージが放棄されます (項7.4 参照)。

3.1.4 開発元/上流 (upstream) の開発者との調整

Debian メンテナとしての仕事のうちで重要な位置を占めるのは、開発元/上流 (upstream) の開発者との窓口であることです。Debian ユーザは、時折バグ報告システムに Debian 特有ではないバグを報告する事があります。Debian メンテナは、いつか上流のリリースで修正できるようにする為、このようなバグ報告を上流の開発者に転送しなくてはなりません。

Debian 固有ではないバグの修正はあなたの義務ではないとはいえ、できるなら遠慮なく修正してください。そのような修正を行った際は、上流の開発者にも送ってください。時折 Debian ユーザ/開発者が上流のバグを修正するパッチを送ってくる事があります。その場合は、あなたはパッチを確認して上流へ転送する必要があります。

ポリシーに準拠したパッケージをビルドするために上流のソースに手を入れる必要がある場合、以降の上流でのリリースにおいて手を入れなくても済むために、ここで含まれる修正を上流の開発者にとって良い形で提案する必要があります。必要な変更が何であれ、上流のソースからフォークしないように常に試みてください。

開発元の開発者らが Debian やフリーソフトウェアコミュニティに対して敵対的である、あるいは敵対的になってきているのを見つけた場合は、ソフトウェアを Debian に含める必要があるかを再考しなければならなくなるでしょう。時折 Debian コミュニティに対する社会的なコストは、そのソフトウェアがもたらすであろう利益に見合わない場合があります。

3.2 管理者的な責務

Debian のような大きさのプロジェクトは、あらゆる事を追いかけられる管理者用のインフラに依っています。プロジェクトメンバーとして、あらゆる物事が滞り無く進むように、あなたにはいくつかの義務があります。

3.2.1 あなたの Debian に関する情報をメンテナンスする

Debian 開発者に関する情報が含まれた LDAP データベースが <https://db.debian.org/> にあります。ここに情報を入力して、情報に変更があった際に更新する必要があります。特に、あなたの debian.org アドレス宛メールの転送先アドレスが常に最新になっているのを必ず確認してください。debian-private の購読をここで設定した場合、そのメールを受け取るアドレスについても同様です。

データベースについての詳細は項4.5 を参照してください。

3.2.2 公開鍵をメンテナンスする

秘密鍵の取扱いには十二分に注意してください。Debian サーバ (項4.4 参照) のような公開サーバや複数のユーザがいるマシンには置かないようにしてください。鍵をバックアップして、コピーはオフラインな場所に置きましょう。ソフトウェアの使い方については付属のドキュメントを参照してください。PGP FAQ を読みましょう。

鍵が盗難に対してだけでなく、紛失についても安全であることを保証する必要があります。失効証明書 (revocation certificate) を生成してコピーを作ってください (紙にも出力しておくのがベストです)。これは鍵を紛失した場合に必要なになります。

公開鍵に対して、署名したり身元情報を追加したりなどしたら、鍵を keyring.debian.org の鍵サーバに送付することで Debian key ring を更新できます。

まったく新しい鍵を追加したりあるいは古い鍵を削除したりする必要がある時は、別の開発者に署名された新しい鍵が必要になります。以前の鍵が侵害されたり利用不可能になった場合には、失効証明書 (revocation certificate) も追加する必要があります。新しい鍵が本当に必要な理由が見当たらない場合は、Keyring メンテナは新しい鍵を拒否することがあります。詳細は http://keyring.debian.org/replacing_keys.html で確認できます。

同様に鍵の取り出し方について項2.3 で説明されています。

Debian での鍵のメンテナンスについて、より突っ込んだ議論を `debian-keyring` パッケージ中のドキュメントで知ることができます。

3.2.3 投票をする

Debian は本来の意味での民主主義ではありませんが、我々はリーダーの選出や一般投票の承認において民主主義的なプロセスを利用しています。これらの手続きについては、**Debian 憲章**で規程されています。

毎年のリーダー選挙以外には、投票は定期的には実施されず、軽々しく提案されるものではありません。提案はそれぞれ `debian-vote@lists.debian.org` メーリングリストでまず議論され、プロジェクトの書記担当者が投票手順を開始する前に複数のエンドースメントを必要とします。

書記担当者が `debian-devel-announce@lists.debian.org` 上で複数回投票の呼びかけを行うので、投票前の議論を迫りかける必要はありません (全開発者がこのメーリングリストを購読することが求められています)。民主主義は、人々が投票に参加しないと正常に機能しません。これが我々が全ての開発者に投票を勧める理由です。投票は GPG によって署名/暗号化されたメールによって行われます。

(過去と現在の) 全ての提案リストが **Debian 投票情報** ページで閲覧できます。提案について、どの様に起案され、支持され、投票が行われたのかという関連情報の確認が可能になっています。

3.2.4 優雅に休暇を取る

予定していた休暇にせよ、それとも単に他の作業で忙しいにせよ、開発者が不在になることがあるのはごく普通のことです。注意すべき重要な点は、他の開発者達があなたが休暇中であることを知る必要があることと、あなたのパッケージについて問題が起こった場合やプロジェクト内での責務を果たすのに問題が生じたという様な場合は、必要なことを彼らが何であってでもできるようにすることです。

通常、これはあなたが休暇中にあなたのパッケージが大きな問題 (リリースクリティカルバグやセキュリティ更新など) となっている場合に、他の開発者に対して NMU (項5.11 参照) を許可することを意味しています。大抵の場合はそれほど致命的なことはおきませんが、他の開発者に対してあなたが作業できない状態であることを知らせるのは重要です。

他の開発者に通知するために行わなければならないことが2つあります。まず、`debian-private@lists.debian.org` にサブジェクトの先頭に [VAC] と付けたメールを送り¹、いつまで休暇なのかを示しておきます。何か問題が起きた際の特別な指示を書いておくこともできます。

他に行うべき事は **Debian developers' LDAP database** 上であなたを vacation とマークする事です (この情報は Debian 開発者のみがアクセスできます)。休暇から戻った時には vacation フラグを削除するのを忘れないように!

理想的には、休暇にあわせて **GPG coordination pages** に登録して、誰かサインを希望している人がいるかどうかをチェックします。開発者がまだ誰もいないけれども応募に興味を持っている人がいるようなエキゾチックな場所に行く場合、これは特に重要です。

3.2.5 脱退について

Debian プロジェクトから去るのを決めた場合は、以下の手順に従ってください:

1. 項5.9.4 の記述に従って、全てのパッケージを放棄 (orphan) してください。
2. GPG でサインされたメールを `debian-private@lists.debian.org` に投げてください。
3. 'Debian RT' という単語 (大文字小文字は関係なし) がサブジェクトのどこかに入ったメールを `keyring@rt.debian.org` に投げて Debian RT でチケットをオープンして、あなたがプロジェクトを去るのを Debian key ring メンテナに知らせてください。
4. @debian.org メールアドレスの alias (例: `press@debian.org`) 経由でメールを受け取っていて削除したい場合、Debian システム管理者に対する RT チケットをオープンしてください。チケットをオープンするには、削除したい alias のアドレスから、`admin@rt.debian.org` 宛でサブジェクトのどこかに "Debian RT" と入れて送信します。

上記のプロセスに従うのは重要です。何故なら活動を停止している開発者を探してパッケージを放棄するのは、非常に時間と手間がかかることからです。

¹ これは、休暇のメッセージを読みたくない人がメッセージを簡単に振り分け可能にするためです。

3.2.6 リタイア後に再加入する

リタイアした開発者のアカウントは、項3.2.5の手続きが開始された際に「emeritus」とマークされ、それ以外の場合は「disabled」となります。「emeritus」アカウントになっているリタイアした開発者は、以下のようにすればアカウントを再度有効にできます:

- da-manager@debian.org に連絡を取ります
- 短縮された NM プロセスを通過します (リタイアした開発者が P&P および T&S の肝心な部分を覚えているのを確認するためです)。
- アカウントに紐付けられた GPG 鍵を今でも管理していることを証明する、あるいは新しい GPG 鍵について、他の開発者から少なくとも 2 つの署名を受けることにより身分証明を行う。

リタイアした開発者で「disabled」アカウントの人は、NM をもう一度通り抜ける必要があります。

Chapter 4

Debian 開発者および Debian メンテナが 利用可能なリソース

この章では、Debian メーリングリストについてのおおよその概略、開発者であるあなたが利用できるであろう Debian マシン、メンテナとしての作業で役立つその他全ての利用可能なリソースを確認します。

4.1 メーリングリスト

Debian 開発者 (それにユーザ) の間で交わされるやり取りの大半は lists.debian.org で提供されている広範囲に渡るメーリングリスト群で行われています。どうやって購読／解除するのか、どうやって投稿するか (あるいはしないのか)、どこで過去の投稿を見つけるのか、どうやって過去の投稿の中から探すのか、どうやってメーリングリスト管理者と連絡をとるのか、その他メーリングリストに関する様々な情報については <https://www.debian.org/MailingLists/> を参照してください。

4.1.1 利用の基本ルール

メーリングリストのメッセージに返信する際には、大本の投稿者が特別に要求しない限り、同報メール (CC) を送らないようにしてください。メーリングリストに投稿する人は必ず返信を見ているはずです。

クロスポスト (同じメッセージを複数のメーリングリストに投稿する) のはお止め下さい。いつものネット上と同じ様に、返信文では引用を削って下さい。概して投稿するメッセージについては、通常の慣習をしっかりと守ってください。

詳細については [行動規範](#) を参照してください。 [Debian コミュニティガイドライン](#) も読むと良いでしょう。

4.1.2 開発の中心となっているメーリングリスト

開発者が利用すべき Debian の中核メーリングリスト:

- debian-devel-announce@lists.debian.org は開発者に重要な事を伝える際に使われます。全開発者がこのメーリングリストを購読する事が望まれます。
- debian-devel@lists.debian.org は様々な技術関連の事柄を話し合うのに使われます。
- debian-policy@lists.debian.org は Debian ポリシーについて話し合い、それに対して投票を行うのに使われます。
- debian-project@lists.debian.org はプロジェクトに関する様々な非技術関連の事柄を話し合うのに使われます。

他にも様々な事柄に特化したメーリングリストが利用できます。一覧については <https://lists.debian.org/> を参照してください。

4.1.3 特別なメーリングリスト

debian-private@lists.debian.org は Debian 開発者間でのプライベートな話し合い用に使う特別なメーリングリストです。つまり、理由がなんであれここに投稿された文章は公開するべきではないものであることを意味しています。このため、これは流量が少ないメーリングリストで、ユーザは本当に必要でない限りは debian-private@lists.debian.org を使わないように勧められています。さらに、このメーリングリストから誰かへメールを転送してはいけません。様々な理由からこのメーリングリストのアーカイブはウェブから見ることはできませんが、master.debian.org 上のシエルアカウントを使って `~debian/archive/debian-private/` ディレクトリを参照することで確認できます。

debian-email@lists.debian.org は、特別なメーリングリストです。ライセンス、バグ、その他について upstream の作者にコンタクトを取る、他の人とプロジェクトについて議論した内容をアーカイブしておくのに役立つ Debian に関するメールをまとめた「福袋」として使われています。

4.1.4 新規に開発関連のメーリングリストの開設を要求する

パッケージ (や関連する小さなパッケージ群) の開発に関するメーリングリストの開設をリクエストをする前に、エイリアスを利用するのを検討してください (master.debian.org 上で `forward-aliasname` ファイルを使うと、きちんと `you-aliasname@debian.org` アドレスに適切に変換してくれます)。あるいは、Alioth 上で自分で管理するメーリングリストを使いましょう。

lists.debian.org 上での通常のメーリングリストが本当に必要であると決意した場合は、[HOWTO](#) に従って進めてリクエストを埋めてください。

4.2 IRC チャンネル

いくつかの IRC チャンネルが Debian の開発のために用意されています。チャンネルは主に [Open and free technology community \(OFTC\)](#) のネットワーク上にホストされています。irc.debian.org の DNS エントリは irc.oftc.net へのエイリアスです。

Debian 用のメインのチャンネルは一般的に `#debian` になります。これは巨大な、多目的のチャンネルで、ユーザがトピックやボットによって提供される最新のニュースを見つけることができる場所です。`#debian` は英語を話す人たち用のもので、他の言語を話す人達のために同様なものには `#debian.de`、`#debian-fr`、`#debian-br` など他にも似通った名前のチャンネルがあります。

Debian 開発での中心のチャンネルは `#debian-devel` です。これはとてもアクティブなチャンネルで、大抵 150 人以上が常にログインしています。このチャンネルは Debian で作業する人達のためのチャンネルであって、サポート用のチャンネルではありません (そのためには `#debian` があります)。このチャンネルは、こっそり覗いてみたい (そして学びたい) 人に対してもオープンでもあります。このチャンネルのトピックは、開発者にとって興味深い情報に溢れています。

`#debian-devel` は公開チャンネルなので、debian-private@lists.debian.org で話されている話題について触れるべきではありません。この目的の為に、`#debian-private` という鍵で守られた他のチャンネルがあります。この鍵は master.debian.org:`~debian/archive/debian-private/` で取得可能です。

他にも、特定の話題について専用のチャンネルがあります。`#debian-bugs` は、バグ潰しパーティ (BSP) を開催するために使われます。`#debian-boot` は、`debian-installer` の作業を調整するのに利用されています。`#debian-doc` は、今あなたが読んでいるような文章のドキュメント化について話すために時折使われています。アーキテクチャやパッケージ群について話すため、他にも専用チャンネルがあります：`#debian-kde`、`#debian-dpkg`、`#debian-jr`、`#debian-edu`、`#debian-oo` (OpenOffice.org パッケージ) など...

同様に非英語圏の開発者のチャンネルも存在しています。例えば `#debian-devel-fr` は Debian の開発に興味があるフランス語を使う人々のためのチャンネルです。

Debian 専用のチャンネルが他の IRC ネットワーク上にもあります。特に [freenode](#) IRC ネットワークは、2006 年 6 月 4 日まで irc.debian.org のエイリアスでした。

freenode でクローク担当に手助けしてもらうには、Jörg Jaspert <joerg@debian.org> さんに対して、利用する nick (ニックネーム) を書いて GPG 署名したメールを送ります。メールの Subject: ヘッダのどこかに cloak の文字を入れてください。nick は登録をする必要があります：[ニックネームの設定の仕方を書いたページ](#)を参照下さい。メールは Debian keyring にある鍵でサインされている必要があります。クローク担当についての詳細は [Freenode のドキュメント](#)を参照して下さい。

4.3 ドキュメント化

このドキュメントは Debian 開発者にとって有用な情報をたくさん含んでいますが、全てを含めることは出来ていません。他の有用なドキュメントが**開発者のコーナー**からリンクされています。時間を取ってリンクを全部眺めれば、より多くの事柄を学べるでしょう。

4.4 Debian のマシン群

Debian ではサーバとして動いている複数のコンピュータがあり、この多くは Debian プロジェクトにおいて重要な役割を果たしています。マシンの大半は移植作業に利用されており、全てインターネットに常時接続されています。

マシンのうち幾つかは、**Debian マシン利用ポリシー**で定められたルールに従う限り、個々の開発者の利用が可能となっています。

とにかく、これらのマシンをあなたが Debian 関連の目的に合うと思ったことに利用できます。システム管理者には丁寧に接し、システム管理者からの許可を最初に得ることなく、非常に大量のディスク容量／ネットワーク帯域／CPU を消費しないようにしてください。大抵これらのマシンはボランティアによって運用されています。

Debian で利用しているパスワードと Debian のマシンにインストールしてある SSH 鍵を保護することに注意してください。ログインやアップロードの際にパスワードをインターネット越しに平文で送るような Telnet や FTP や POP などの利用方法は避けてください。

あなたが管理者でも無い限り、Debian サーバ上には Debian に関連しないものを一切置かないようにしてください。

Debian のマシン一覧は <https://db.debian.org/machines.cgi> で確認可能です。このウェブページはマシン名、管理者の連絡先、誰がログイン可能か、SSH 鍵などの情報を含んでいます。

Debian サーバでの作業について問題があり、システム管理者らに知らせる必要があると考えた場合は、<https://rt.debian.org/> にあるリクエストトラッカーの DSA (Debian System Administration) チームのキュー一覧でオープンになっている問題の一覧を確認できます (ユーザー名: "debian" と master.debian.org:~debian/misc/rt-password にあるパスワードでログインできます)。新たな問題を報告するには、単に admin@rt.debian.org にメールを送ってください。"Debian RT" をサブジェクトのどこかに入れるのを忘れずに。DSA チームに連絡を取るには、プライベートな情報あるいはその他の秘密にしておくべき情報を含む場合には dsa@debian.org を、それ以外の場合は debian-admin@lists.debian.org ヘメールしてください。DSA チームは OFTC の IRC チャンネル #debian-admin にも居ます。

システム管理に関連しない、特定のサービスについて問題がある場合 (アーカイブからパッケージを削除する、ウェブサイトの改善提案など) は、大抵の場合「擬似パッケージ」に対してバグを報告することになります。どうやってバグ報告をするかについては項 7.1 を参照してください。

中心となっているサーバのうち幾つかは利用が制限されていますが、そこにある情報は他のサーバへミラーされています。

4.4.1 バグ報告サーバ

bugs.debian.org がバグ報告システム (BTS) の中心となっています。

Debian のバグについて定量的な分析や処理をするような計画がある場合、ここで行ってください。ですが、不要な作業の重複や処理時間の浪費を減らすため、何であれ実装する前に debian-devel@lists.debian.org であなたの計画を説明してください。

4.4.2 ftp-master サーバ

<ftp-master.debian.org> サーバは、Debian アーカイブのコピーの中心位置を占めています。通常、<ftp.upload.debian.org> ヘアップロードされたパッケージは最終的にはこのサーバにあります。項 5.6 を参照してください。

このサーバの利用は制限されています。ミラーが <mirror.ftp-master.debian.org> 上で利用可能です。

Debian FTP アーカイブについて問題がある場合、通常 <ftp.debian.org> 擬似パッケージに対するバグ報告を行うか、ftpmaster@debian.org ヘメールをする必要がありますが、項 5.9 にある手順も参照してください。

4.4.3 www-master サーバ

メインの web サーバが www-master.debian.org です。公式 web ページを持ち、新たな参加者に対する Debian の顔となっています。

Debian のウェブサーバについて問題を見つけた場合、大抵の場合は擬似パッケージ www.debian.org に対してバグを報告する必要があります。誰か他の人が既に [バグ追跡システム](#) に問題を報告していないかどうかチェックするのを忘れないようにしてください。

4.4.4 people ウェブサーバ

people.debian.org は、開発者個人の何か Debian に関連するウェブページのために使われているサーバです。

ウェブに置きたい何か Debian 特有の情報を持っている場合、people.debian.org 上のホームディレクトリの `public_html` 以下にデータを置くことでこれが可能となっています。これには <https://people.debian.org/~your-user-id/> という URL でアクセス可能です。

他のホストではバックアップされないのに対して、ここではバックアップされるので、これを使うのは特定の位置づけのものだけにすべきです。

大抵の場合、他のホストを使う唯一の理由はアメリカの輸出制限に抵触する素材を公開する必要がある時です。その様な場合はアメリカ国外に位置する他のサーバのどれかを使えます。

何か質問がある場合は、debian-devel@lists.debian.org にメールして下さい。

4.4.5 VCS (バージョン管理システム) サーバ

あなたが行っている Debian に関連する作業にバージョン管理システムが必要であれば、Alioth にホストされている既存のリポジトリを使うことも新しいプロジェクトを作って、それにあなたが選んだ VCS のリポジトリを要求することも可能です。Alioth は CVS (cvs.alioth.debian.org/cvs.debian.org)、Subversion (svn.debian.org)、Arch (tla/baz、共に arch.debian.org 上)、Bazaar (bzt.debian.org)、Darcs (darcs.debian.org)、Mercurial (hg.debian.org) そして Git (git.debian.org) をサポートしています。パッケージを VCS リポジトリ上でメンテナンスする予定であれば、<https://wiki.debian.org/Alioth/PackagingProject> を確認してください。Alioth で提供されているサービスについての詳細は項 [4.12](#) を参照してください。

4.4.6 複数のディストリビューション利用のために chroot を使う

幾つかのマシン上では、異なったディストリビューション用の chroot が利用可能です。以下の様にして使うことが出来ます：

```
vore$ dchroot unstable
Executing shell in chroot: /org/vore.debian.org/chroots/user/unstable
```

全ての chroot 環境内で、一般ユーザの home ディレクトリが利用可能になっています。どの chroot が利用可能かについては <https://db.debian.org/machines.cgi> にて確認ができます。

4.5 開発者データベース

<https://db.debian.org/> にある開発者データベースは、Debian 開発者のアトリビュート进行操作する LDAP ディレクトリです。このリソースは Debian 開発者リストの検索に使えます。この情報の一部は Debian サーバの finger サービスを通じても利用可能になっています。finger yourlogin@db.debian.org として何が返ってくるかを確認してみてください。

開発者らは、以下に挙げるような自身に関する様々な情報を変更するために [データベースにログイン](#) ができます。

- [debian.org](#) アドレス宛のメールを転送するアドレス
- [debian-private](#) の購読
- 休暇中かどうか
- 住所、国名、[Debian 開発者世界地図](#) で使われている住んでいる地域の緯度経度、電話・ファックス番号、IRC でのニックネーム、そしてウェブページのアドレスなどの個人情報
- Debian プロジェクトのマシン上でのパスワードと優先的に指定するシェル

当然ですが、ほとんどの情報は外部からはアクセス可能にはなっていません。より詳細な情報については、<https://db.debian.org/doc-general.html> で参照できるオンラインドキュメントを読んでください。

開発者は公式 Debian マシンへの認証に使われる SSH 鍵を登録することもできますし、新たな *.debian.net の DNS エントリの追加すら可能です。これらの機能は <https://db.debian.org/doc-mail.html> に記述されています。

4.6 Debian アーカイブ

Debian ディストリビューションは大量のパッケージ (現在約 15000 個) と幾つかの追加ファイル (ドキュメントやインストール用ディスクイメージなど) から成り立っています。

以下が完全な Debian アーカイブのディレクトリツリーの例です：

```
dists/stable/main/
dists/stable/main/binary-amd64/
dists/stable/main/binary-armel/
dists/stable/main/binary-i386/
...
dists/stable/main/source/
...
dists/stable/main/disks-amd64/
dists/stable/main/disks-armel/
dists/stable/main/disks-i386/
...

dists/stable/contrib/
dists/stable/contrib/binary-amd64/
dists/stable/contrib/binary-armel/
dists/stable/contrib/binary-i386/
...
dists/stable/contrib/source/

dists/stable/non-free/
dists/stable/non-free/binary-amd64/
dists/stable/non-free/binary-armel/
dists/stable/non-free/binary-i386/
...
dists/stable/non-free/source/

dists/testing/
dists/testing/main/
...
dists/testing/contrib/
...
dists/testing/non-free/
...

dists/unstable
dists/unstable/main/
...
dists/unstable/contrib/
...
dists/unstable/non-free/
...

pool/
pool/main/a/
pool/main/a/apt/
...
pool/main/b/
pool/main/b/bash/
...
pool/main/liba/
```

```
pool/main/liba/libalias-perl/  
...  
pool/main/m/  
pool/main/m/mailx/  
...  
pool/non-free/f/  
pool/non-free/f/firmware-nonfree/  
...
```

見て分かるように、一番上のディレクトリは `dists/` と `pool/` という2つのディレクトリを含んでいます。後者の“pool”はパッケージが実際に置かれており、アーカイブのメンテナンスデータベースと関連するプログラムによって利用されます。前者には `stable`、`testing`、そして `unstable` というディストリビューションが含まれます。ディストリビューションサブディレクトリ中の `packages` および `Sources` ファイルは `pool/` ディレクトリ中のファイルを参照しています。以下の各ディストリビューションのディレクトリツリーは全く同じ形式になっています。以下で `stable` について述べていることは `unstable` や `testing` ディストリビューションにも同様に当てはまります。

`dists/stable` は、`main`、`contrib`、`non-free` という名前の3つのディレクトリを含んでいます。

それぞれ、`source` パッケージ (`source`) のディレクトリとサポートしている各アーキテクチャ (`binary-i386`、`binary-amd64` など) のディレクトリがあります。

`main` は特定のアーキテクチャ (`disks-i386`、`disks-amd64` など) 上で Debian ディストリビューションをインストールする際に必要となるディスクイメージと主要なドキュメントの基本部分が入っている追加ディレクトリを含んでいます。

4.6.1 セクション

Debian アーカイブの `main` セクションは公式な **Debian** ディストリビューションを構成するものです。`main` セクションが公式なのは、我々のガイドライン全てに合致するものであるからです。他の2つのセクションはそうではなく、合致は異なる程度となっています…つまり、Debian の公式な一部ではありません。

`main` セクションにある全てのパッケージは、**Debian フリーソフトウェアガイドライン (DFSG)** 及び **Debian ポリシーマニユアル** に記載されている他のポリシーの要求事項に完全に適合していなければなりません。DFSG は我々の定義する「フリーソフトウェア」です。詳細は Debian ポリシーマニユアルを確認してください。

`contrib` セクションにあるパッケージは DFSG に適合している必要がありますが、他の要求事項を満たしてはいないことでしょう。例えば、`non-free` パッケージに依存している、などです。

DFSG を満たさないパッケージは `non-free` セクションに配置されます。これらのパッケージは Debian ディストリビューションの一部としては考えられてはいませんが、我々はこれらを利用できるようにしており、`non-free` ソフトウェアのパッケージについて (バグ追跡システムやメーリングリストなどの) インフラストラクチャを提供しています。

Debian ポリシーマニユアル は3つのセクションについてより正確な定義を含んでいます。上記の説明はほんの触りに過ぎません。

アーカイブの最上位階層で3つのセクションに別れていることは、インターネット上の FTP サーバ経由であれ、CD-ROM であれ、Debian を配布したいと考える人にとって大事なことです…その様な人は `main` セクションと `contrib` セクションのみを配布することで、法的なリスクを回避できます。例えば、`non-free` セクションにあるパッケージのいくつかは商的な配布を許可していません。

その一方で、CD-ROM ベンダは `non-free` 内のパッケージ群の個々のパッケージライセンスを簡単に確認でき、問題が無ければその多くを CD-ROM に含めることが出来ます。(これはベンダによって大いに異なるので、この作業は Debian 開発者にはできません)。

セクションという用語は、利用可能なパッケージの構成や参照を簡単にしているカテゴリを指すことにも使われている点に注意ください。例えば、`admin`、`net`、`utils` などです。昔々、これらのセクション (むしろサブセクション) は Debian アーカイブ内のサブディレクトリとして存在していました。最近では、これらはパッケージのセクションヘッダにのみ存在しています。

4.6.2 アーキテクチャ

はじめのうちは、Linux カーネルは Intel i386 (またはそれより新しい) プラットフォーム用のみが利用可能で、Debian も同様でした。しかし、Linux は日に日に広まり、カーネルも他のアーキテクチャへと移植され、そして Debian はそれらのサポートを始めました。そして、沢山のハードウェアをサポート

するだけでは飽き足らず、Debian は `hurd` や `kfreebsd` のような他の Unix カーネルをベースにした移植版を作成することを決めました。

Debian GNU/Linux 1.3 では `i386` のみが利用可能でした。Debian 2.0 では、`i386` と `m68k` アーキテクチャがリリースされました。Debian 2.1 では、`i386`、`m68k`、`alpha`、そして `sparc` アーキテクチャがリリースされました。そして、Debian は巨大に成長しました。Debian 6 は、合計 9 個の Linux アーキテクチャ (`amd64`、`armel`、`i386`、`ia64`、`mips`、`mipsel`、`powerpc`、`s390`、`sparc`、そして 2 つの `kFreeBSD` アーキテクチャ (`kfreebsd-i386` および `kfreebsd-amd64`) をサポートしています。

特定の移植版についての開発者/ユーザへの情報は [Debian 移植版のウェブページ](#) で入手可能です。

4.6.3 パッケージ

Debian パッケージには 2 種類あります。ソースパッケージとバイナリパッケージです。

フォーマットに応じて、ソースパッケージは必須の `.dsc` ファイルに加え、一つあるいはそれ以上のファイルから成り立ちます:

- フォーマット “1.0” では、`.tar.gz` ファイルか、`.orig.tar.gz` ファイルと `.diff.gz` ファイルの二つを持っています。
- フォーマット “3.0 (quilt)” では、必須となる開発元の tarball である `.orig.tar.{gz,bz2,xz}`、それからオプションで、開発元の追加 tarball である `.orig-component.tar.{gz,bz2,xz}` をいくつか、そして必須の `debian tarball`、`debian.tar.{gz,bz2,xz}` です。
- フォーマット “3.0 (native)” では、単一の `.tar.{gz,bz2,xz}` tarball のみを持っています。

パッケージが特に Debian 用に作られていて Debian 以外で利用されていない場合は、プログラムのソースを含んだ単純な 1 つの `.tar.{gz,bz2,xz}` ファイルがあるだけで、“native” ソースパッケージと呼ばれます。パッケージが他にも配布されている場合は、`.orig.tar.{gz,bz2,xz}` は upstream ソースコードと呼ばれるものを保持しています (訳注: upstream = 開発元)。upstream ソースコードは upstream メンテナ (大抵の場合はソフトウェアの作者) によって配布されているソースコードです。この場合、`.diff.gz` や `debian.tar.{gz,bz2,xz}` は、Debian パッケージメンテナによって加えられた変更を含んでいます。

`.dsc` ファイルはソースパッケージ中のすべてのファイルをチェックサム (`md5sums`、`sha1sums` および `sha256sums`) と共にリストしたものと、パッケージ関連の追加情報 (メンテナ、バージョン、etc) を含んでいます。

4.6.4 ディストリビューション

前章で説明したディレクトリシステムは、それ自体がディストリビューションディレクトリ内に含まれています。それぞれのディストリビューションは、実際には Debian アーカイブ自体の最上位階層の `pool` ディレクトリに含まれています。

簡単にまとめると、Debian アーカイブは FTP サーバのルートディレクトリを持っています。例えば、ミラーサイトでいうと `ftp.us.debian.org` では Debian アーカイブそのものは `/debian` に含まれており、これは共通した配置となっています (他には `/pub/debian` があります)。

ディストリビューションは Debian ソースパッケージとバイナリパッケージと、これに対応した `Sources` と `Packages` のインデックスファイル (これら全てのパッケージのヘッダ情報を含む) から構成されています。前者は `pool/` ディレクトリに、そして後者はアーカイブの `dists/` ディレクトリに含まれています (後方互換性のため)。

4.6.4.1 安定版 (stable)、テスト版 (testing)、不安定版 (unstable)

常に 安定版 (stable) (`dists/stable` に属します)、テスト版 (testing) (`dists/testing` に属します)、不安定版 (unstable) (`dists/unstable` に属します) と呼ばれるディストリビューションが存在しています。これは Debian プロジェクトでの開発プロセスを反映しています。

活発な開発が不安定版 (unstable) ディストリビューションで行われています (これが、何故このディストリビューションが開発ディストリビューションと呼ばれることがあるかという理由です)。全ての Debian 開発者は、このディストリビューション内の自分のパッケージを何時でも更新できます。つまり、このディストリビューションの内容は日々変化しているのです。このディストリビューションの全てが正しく動作するかを保証することについては特別な努力は払われていないので、時には文字通り不安定 (unstable) となります。

テスト版 (testing) ディストリビューションは、パッケージが特定の判定規準を満たした際に不安定版から自動的に移動されることで生成されています。この判定規準はテスト版に含まれるパッケージ

として十分な品質であることを保証する必要があります。テスト版への更新は、新しいパッケージがインストールされた後、毎日 2 回実施されています。項 5.13 を参照してください。

一定の開発期間後、リリースマネージャが適当であると決定すると、テスト版 (testing) ディストリビューションはフリーズされます。これは、不安定版 (unstable) からテスト版 (testing) へのパッケージ移動をどのように行うかのポリシーがきつくなることを意味しています。バグが多すぎるパッケージは削除されます。バグ修正以外の変更がテスト版 (testing) に入ることは許可されません。いくらかの時間経過後、進行状況に応じてテスト版 (testing) ディストリビューションはより一層フリーズされます。テスト版ディストリビューションの取扱い詳細については `debian-devel-announce` にてリリースチームが発表します。リリースチームが満足する程度に残っていた問題が修正された後、ディストリビューションがリリースされます。リリースは、テスト版 (testing) が安定版 (stable) へとリネームされる事を意味しており、テスト版 (testing) 用の新しいコピーが作成され、以前の安定版 (stable) は旧安定版 (oldstable) にリネームされ、最終的にアーカイブされるまで存在しています。アーカイブ作業では、コンテンツは `archive.debian.org` へと移動されます。

この開発サイクルは、不安定版 (unstable) ディストリビューションが、一定期間テスト版 (testing) を過ごした後で安定版 (stable) になる仮定に基づいています。一旦ディストリビューションが安定したと考えられたとしても、必然的にいくつかのバグは残っています—これが安定版ディストリビューションが時折更新されている理由です。しかし、これらの更新はとても注意深くテストされており、新たなバグを招き入れるリスクを避けるためにそれぞれ個々にアーカイブに収録されるようになっています。安定版 (stable) への追加提案は、`proposed-updates` ディレクトリにて参照可能です。`proposed-updates` にある合格したこれらのパッケージは、定期的にまとめて安定版ディストリビューションに移動され、安定版ディストリビューションのリビジョンレベルが 1 つ増えることとなります (例: '6.0' が '6.0.1' に、'5.0' が '5.0.8' に、以下同様)。詳細に付いては、**安定版 (stable) ディストリビューションへのアップロード** を参照してください。

不安定版 (unstable) での開発はフリーズ期間中も続けられていることに注意してください。不安定版 (unstable) ディストリビューションはテスト版 (testing) とは平行した状態でありつづけているからです。

4.6.4.2 テスト版ディストリビューションについてのさらなる情報

パッケージは通常、不安定版 (unstable) におけるテスト版への移行基準を満たした後でテスト版 (testing) ディストリビューションへとインストールされます。

より詳細については、**テスト版ディストリビューションについての情報** を参照してください。

4.6.4.3 試験版 (experimental)

試験版 (experimental) は特殊なディストリビューションです。これは、'安定版' や '不安定版' と同じ意味での完全なディストリビューションではありません。その代わり、ソフトウェアがシステムを破壊する可能性がある、あるいは不安定版ディストリビューションに導入することですら不安定過ぎる (だが、それにもかかわらず、パッケージにする理由はある) ものであるような、とても実験的な要素を含むソフトウェアの一時的な置き場であることを意味しています。試験版 (experimental) からパッケージをダウンロードしてインストールするユーザは、十分に注意を受けているのを期待されています。要するに、試験版 (experimental) ディストリビューションを利用すると、どのようなかは全くわからないということです。

以下が、試験版 (experimental) 用の `sources.list(5)` です:

```
deb http://ftp.xy.debian.org/debian/ experimental main
deb-src http://ftp.xy.debian.org/debian/ experimental main
```

ソフトウェアがシステムに多大なダメージを与える可能性がある場合、試験版 (experimental) へ配置する方が良いでしょう。例えば、実験的な圧縮ファイルシステムは恐らく試験版 (experimental) に行くことになるでしょう。

パッケージの新しい上流バージョンが新しい機能を導入するが多くの古い機能を壊してしまう場合であれば、アップロードしないでおくか試験版 (experimental) へアップロードするかしましよう。新しいバージョン、ベータ版などで、利用する設定が完全に変わっているソフトウェアは、メンテナの配慮に従って試験版 (experimental) へ入れることができます。もしも非互換性や複雑なアップグレード対応について作業している場合などは、試験版 (experimental) をステージングエリアとして利用することができるのです。その結果、テストユーザは早期に新しいバージョンの利用が可能になります。

試験版 (experimental) のソフトウェアは不安定版 (unstable) へ説明文に幾つかの警告を加えた上で入れることも可能ではありますが、お勧めはできません。それは、不安定版 (unstable) のパッケージはテスト版 (testing) へ移行し、そして安定版 (stable) になることが期待されているからです。試験版 (experimental) を使うのをためらうべきではありません。何故なら ftpmaster には何の苦痛も与えませんし、試験版 (experimental) のパッケージは一旦不安定版 (unstable) により大きなバージョン番号でアップロードされると定期的に削除されるからです。

システムにダメージを与えないような新しいソフトウェアは直接不安定版 (unstable) へ入れることが可能です。

試験版 (experimental) の代わりとなる方法は、people.debian.org 上の個人的な web ページを使うことです。

4.6.5 リリースのコードネーム

リリースされた Debian ディストリビューションは全てコードネームを持っています：Debian 6.0 は squeeze、Debian 7 は wheezy、Debian 8 は jessie、次期リリースの Debian 9 は stretch、そして Debian 10 は buster と呼ばれます。sid と呼ばれる「擬似ディストリビューション」もあり、これは現在の unstable ディストリビューションを指します。パッケージは安定に近づくにつれて unstable から testing へと移動されるので、sid そのものは決してリリースされません。通常の Debian ディストリビューションの内容同様に、sid は Debian ではまだ公式にサポートあるいはリリースされていないアーキテクチャのパッケージを含んでいます。これらのアーキテクチャは、いつか主流ディストリビューションに統合される予定です。過去のコードネームとバージョンはウェブサイトに掲載されています。

Debian はオープンな開発体制（つまり、誰もが開発について参加／追いかけが可能）となっており、不安定版 (unstable) および テスト版 (testing) ディストリビューションすら Debian の FTP および HTTP サーバネットワークを通じてインターネットへ提供されています。従って、リリース候補版を含むディレクトリをテスト版 (testing) と呼んだ場合、このバージョンがリリースされる際に安定版 (stable) へとリネームする必要があるということを意味しており、すべての FTP ミラーがディストリビューションすべて（とても巨大です）を再回収することになります。

一方、最初からディストリビューションディレクトリを Debian-x.y と呼んでいた場合、皆 Debian リリース x.y が利用可能になっていると考えるでしょう。（これは過去にあったことで、CD-ROM ベンダが Debian 1.0 の CD-ROM を pre-1.0 開発版を元に作成したことによります。これが、何故最初の公式 Debian のリリース版が 1.0 ではなく 1.1 であったかという理由です）。

従って、アーカイブ内のディストリビューションディレクトリの名前はリリースの状態ではなくコードネームで決定されます（例えば 'jessie' など）。これらの名称は開発期間中とリリース後も同じものであり続けます。そして、簡単に変更可能なシンボリックリンクによって、現在の安定版リリースディストリビューションを示すことになります。これが、stable、testing、unstable へのシンボリックリンクがそれぞれ相応しいリリースディレクトリを指しているのに対して、実際のディストリビューションディレクトリではコードネームを使っている理由です。

4.7 Debian ミラーサーバ

各種ダウンロードアーカイブサイトおよびウェブサイトは、中央サーバを巨大なトラフィックから守るために複数ミラーが利用可能となっています。実際のところ、中央サーバのいくつかは公開アクセスが出来るようにはなっていません - 代わりに一次ミラーが負荷を捌いています。このようにして、ユーザは常にミラーにアクセスして利用可能になっており、Debian を多くのサーバやネットワーク越しに配布するのに必要な帯域が楽になり、ユーザが一次配布元に集中しすぎてサイトがダウンしてしまうのをおおそ避けられるようになります。一次配布ミラーは内部サイトからのトリガーによって更新されるので、可能な限り最新になっている（我々はこれをプッシュミラーと呼んでいます）。

利用可能な公開 FTP/HTTP サーバのリストを含む、Debian ミラーサーバについての全ての情報が <https://www.debian.org/mirror/> から入手可能です。この役立つページには、内部的なものであれ公開されるものであれ、自分のミラーを設定することに興味を持った場合に役立つ情報とツールも含まれています。

注意してほしいのは、ミラーは大抵 Debian の支援に興味を抱いてくれた第三者によって運用されていることです。そのため、開発者は通常これらのマシン上にはアカウントを持ってはいません。

4.8 Incoming システム

Incoming システムは、更新されたパッケージを集めて Debian アーカイブにインストールする役割を果たしています。これは ftp-master.debian.org 上にインストールされたディレクトリとスクリプト

トの集合体です。

全てのメンテナによってアップロードされたパッケージは UploadQueue というディレクトリに置かれます。このディレクトリは、毎分 **queued** と呼ばれるデーモンによってスキャンされ、*.command ファイルが実行されて、そのまま正しく署名された *.changes ファイルが対応するファイルと共に unchecked ディレクトリに移動されます。このディレクトリは ftp-master の様に制限されており、ほとんどの開発者には見えるようにはなっていません。ディレクトリはアップロードされたパッケージと暗号署名の完全性を照合する **dak process-upload** スクリプトによって 15 分毎にスキャンされます。パッケージがインストール可能であると判断されると、done ディレクトリに移動されます。これがパッケージの初アップロードの場合 (あるいは新たなバイナリパッケージを含んでいる場合)、ftpmaster による許可を待つ場所である new ディレクトリに移動されます。パッケージが ftpmaster によって手動でインストールされるファイルを含む場合は byhand ディレクトリに移動します。それ以外の場合は、エラーが検出されるとパッケージは拒否されて reject ディレクトリへと移動されます。

パッケージが受け入れられると、システムは確認のメールをメンテナに送り、アップロードの際に修正済みとされたバグをクローズし、auto-builder がパッケージのリコンパイルを始めます。Debian アーカイブに実際にインストールされるまで、パッケージはすぐに <http://incoming.debian.org/> にてアクセス可能になります。この作業は 1 日に 4 回行われます (様々な経緯から `dinstall run` と呼ばれています)。そしてパッケージは incoming から削除され、他のパッケージ全てと共に pool にインストールされます。他のすべての更新 (例えば Packages インデックスファイルや Sources インデックスファイル) が作成されると、一次ミラー全てを更新する特別なスクリプトが呼び出されます。

アーカイブメンテナのソフトウェアは、あなたがアップロードした OpenPGP/GnuPG で署名された .changes ファイルも適切なメーリングリストへと送信します。パッケージの Distribution が stable に設定されてリリースされた場合、案内は debian-changes@lists.debian.org に送られます。パッケージの Distribution として unstable や experimental が設定されている場合、案内は代わりに debian-devel-changes@lists.debian.org や debian-experimental-changes@lists.debian.org へと投稿されます。

ftp-master は利用が制限されているサーバなので、インストールされたもののコピーは mirror.ftp-master.debian.org 上で全ての開発者が利用できるようになっています。

4.9 パッケージ情報

4.9.1 ウェブ上から

パッケージはそれぞれ複数の目的別のウェブページを持っています。<https://packages.debian.org/package-name> は各ディストリビューション中でそれぞれ利用可能なパッケージバージョンを表示します。バージョン毎のリンク先のページはパッケージの説明、依存関係、ダウンロードへのリンクを含んだ情報を提供しています。

バグ追跡システムは個々のパッケージのバグを記録していきます。<https://bugs.debian.org/package-name> というような URL で与えたパッケージ名のバグを閲覧できます。

4.9.2 dak ls ユーティリティ

dak ls は dak ツールスイートの一部で、全ディストリビューションおよびアーキテクチャの中から利用可能なパッケージバージョンをリストアップします。**dak** ツールは ftp-master.debian.org 上、mirror.ftp-master.debian.org 上のミラーにて利用できます。パッケージ名に対して一つの引数を使います。実際に例を挙げた方が分かりやすいでしょう:

```
$ dak ls evince
evince | 0.1.5-2sarge1 |      oldstable | source, alpha, arm, hppa, i386, ia64, ←
      m68k, mips, mipsel, powerpc, s390, sparc
evince | 0.4.0-5 |      etch-m68k | source, m68k
evince | 0.4.0-5 |      stable | source, alpha, amd64, arm, hppa, i386, ia64 ←
      , mips, mipsel, powerpc, s390, sparc
evince | 2.20.2-1 |      testing | source
evince | 2.20.2-1+b1 |      testing | alpha, amd64, arm, armel, hppa, i386, ia64 ←
      , mips, mipsel, powerpc, s390, sparc
evince | 2.22.2-1 |      unstable | source, alpha, amd64, arm, armel, hppa, ←
      i386, ia64, m68k, mips, mipsel, powerpc, s390, sparc
```

この例では、不安定版 (unstable) でのバージョンはテスト版 (testing) のバージョンと違っており、テスト版のパッケージは全アーキテクチャについて、binary-only NMU されたパッケージに

なっています。それぞれのバージョンのパッケージは、全アーキテクチャ上で再コンパイルされています。

4.10 Debian パッケージトラッカー

パッケージトラッカーは、ソースパッケージの動きを追いかけるメールおよびウェブベースのツールです。Debian パッケージトラッカーでパッケージに対して購読 (subscribe) を行うだけで、パッケージメンテナが受け取るメールとまったく同じものを受け取れます。

パッケージトラッカーから送られるメールは、以下のキーワードリストのうち一つに分類されます。これによって、受け取りたいメールを選別できます。

デフォルトで受け取るもの:

bts バグ報告とそれに続くディスカッション全て。

bts-control バグ報告の状態変更について、control@bugs.debian.org からのメール通知。

upload-source アップロードされたソースパッケージが受理された際の **dak** からのメール通知。

archive dak からの他の警告/エラーメール (セクション (section) や優先度 (priority) フィールドの相違の上書きなど)。パッケージの削除通知も含まれます。

build ビルド失敗の通知はビルドデーモンのネットワークによって送信されます。この通知には分析のためのビルドログへのポイントが含まれています。

default 自動ではない、パッケージの購読者にコンタクトしたい人によってパッケージトラッカーに送られるメール。これは、dispatch+sourcepackage@tracker.debian.org にメールすることで可能になります。spam メールを防ぐため、このアドレスに送られるメッセージは全て空の値の X-Distro-Tracker-Approved ヘッダを含む必要があります。

contact [@packages.debian.org](mailto:*@packages.debian.org) のメールエイリアスを通じてメンテナに送信されたメール。

summary testing への移行を含む、パッケージの状態についての定期的なメールでのサマリ。理想的には、開発元の新しいバージョンの通知やパッケージが削除あるいは放棄 (orphaned) された情報などを含みます (…が、まだそうなってはいません)。

受け取るかどうかを決められる追加情報:

upload-binary バイナリパッケージが受け入れられた際に **dak** から送られる通知メールです。言い換えると、あなたのパッケージがビルドデーモンや他のアーキテクチャについて移植者によってアップロードされた場合は、どの様に全アーキテクチャに対してリコンパイルされているかを追跡するためのメールを受け取れるということです。

vcs パッケージが VCS リポジトリを持っていて、メンテナがコミット通知をパッケージトラッカーに転送するように設定している場合の VCS コミット通知です。

translation Debian Description 翻訳プロジェクトに投稿された descriptions の翻訳や debconf テンプレート。

derivatives 派生ディストリビューションでのパッケージに加えられた変更の情報 (例 Ubuntu)。

derivatives-bugs 派生ディストリビューションでのバグレポートやコメント (例 Ubuntu)。

4.10.1 パッケージトラッカーのメールインターフェイス

パッケージトラッカーの購読管理は control@tracker.debian.org へ様々なコマンドを送信することで可能です。

subscribe **<sourcepackage>** [**<email>**] *email* を対応するソースパッケージ *sourcepackage* への連絡先として登録します。二番目の引数が存在しない場合には送信者アドレスが使われます。*sourcepackage* が正しくないソースパッケージである場合は、警告メールが送付されます。正しいバイナリパッケージである場合には、パッケージトラッカーは対応するソースパッケージの購読者として登録します。

unsubscribe <sourcepackage> [<email>] 指定されたメールアドレス、あるいは二番目の引数が空の場合は送信者のアドレスを利用して、ソースパッケージ *sourcepackage* に対するそれまでの購読を取り消します。

unsubscribeall [<email>] 指定されたメールアドレス、あるいは二つ目の引数が空白の場合は送信者アドレスのすべての購読を停止します。

which [<email>] 送信者あるいは追加で指定したメールアドレスに対する全購読リストを確認する。

keyword [<email>] キーワードは利用したいものを入れてください。キーワードの説明は、[上記を確認してください](#)。

keyword <sourcepackage> [<email>] 前の項目同様ですが、指定したソースパッケージについて、それぞれのソースパッケージに異なったキーワードをしたい場合に利用します。

keyword [<email>] {+|-|=} <list of keywords> 指定したキーワードで、メールが許可(+)あるいは拒否(-)に分類されます。許可するキーワードのリストを定義(=)してください。これはユーザが許可したデフォルトのキーワードリストを変更します。

keywordall [<email>] {+|-|=} <list of keywords> 指定したキーワードで、メールが許可(+)あるいは拒否(-)に分類されます。許可するキーワードのリストを定義(=)してください。これは現在購読しているユーザによって許可されているデフォルトのキーワードリストを変更します。

keyword <sourcepackage> [<email>] {+|-|=} <list of keywords> 上の項目と同じですが、指定したソースパッケージのキーワードリストを上書きします。

quit | **thanks** | **--** コマンド処理を終了します。これ以下の全ての行は bot からは無視されます。

pts-subscribe コマンドラインユーティリティ (devscripts パッケージに含まれています) は、例えば non-maintainer アップロードが行われた後など、手動でパッケージの一時的な購読が行えます。

4.10.2 パッケージトラッカーからのメールを振り分ける

パッケージの購読を行うと、パッケージトラッカーから転送されるメールを受けとるようになります。このメールは (例えば **procmail** を使って) 別のメールボックスへとフィルタできるように特別なヘッダがされています。追加されるヘッダは X-Loop, X-Distro-Tracker-Package, X-Distro-Tracker-Keyword, X-Debian-Package, X-Debian, List-Id and List-Unsubscribe です。

以下は dpkg パッケージに対するソースアップロードについて付加されるヘッダの例です:

```
X-Loop: dispatch@tracker.debian.org
X-Distro-Tracker-Package: dpkg
X-Distro-Tracker-Keyword: upload-source
X-Debian-Package: dpkg
X-Debian: tracker.debian.org
List-Id: <dpkg.tracker.debian.org>
List-Unsubscribe: <mailto:control@tracker.debian.org?body=unsubscribe%20dpkg>
```

4.10.3 パッケージトラッカーへ VCS コミットを転送する

Debian パッケージのメンテナンスに公開 VCS リポジトリを使っている場合、パッケージトラッカーにコミット通知を転送することで、購読者 (や共同メンテナ) がパッケージの変更をすぐに知ることができます。

コミット通知を生成するように VCS リポジトリを設定したら、dispatch@tracker.debian.org または dispatch+sourcepackage_vcs@tracker.debian.org 宛でこのメールのコピーがパッケージトラッカーへ送付されているのを確認する必要があります。前者の場合、パッケージトラッカーがソースパッケージと適切なキーワードを認識できるようになっていることを確認する必要があります。この場合、X-Distro-Tracker-Package:sourcepackage と、X-Distro-Tracker-Keyword:vcs というヘッダを追加するか、パッケージトラッカーが X-Git-Repo ヘッダーを認識して、git リポジトリの名前はソースパッケージに一致すると仮定する事実に基づくことになります。

Subversion のリポジトリは、svnmailer の利用が推奨されています。どの様に行うかは <https://wiki.debian.org/Alioth/PackagingProject> を参照してください。

4.10.4 パッケージトラッカーのウェブインターフェイス

PTS は各ソースパッケージについての大量の情報をまとめたウェブインターフェイスを <https://tracker.debian.org/> に持っています。その機能はたくさんの有用なリンク (BTS、QA の状態、連絡先情報、DDTS の翻訳状態、build のログ) や様々な所からの情報 (最近の changelog エントリ 30 個、testing の状態など...) を集めたものです。特定のソースパッケージについて知りたい場合に非常に有用なツールです。さらに、一旦認証すれば、どのパッケージについてもクリックひとつで購読とキャンセルができます。

特定のソースパッケージに関しては <https://tracker.debian.org/pkg/sourcepackage> のような URL で直接ウェブページに飛べます。

ウェブインターフェイスは容易に拡張可能で、有用なデータを統合するのは大歓迎です。貢献したい場合は、<https://tracker.debian.org/docs/contributing.html> を参照して下さい。

4.11 Developer's packages overview

QA (quality assurance、品質保証) ウェブポータルが <https://qa.debian.org/developer.php> から利用できます。これは、一人の開発者のすべてのパッケージの一覧表を表示します (集団で行っている場合は、共同メンテナとしてとして表示されます)。この表は開発者のパッケージについてうまく要約された情報を与えてくれます: 重要度に応じたバグの数やそれぞれのディストリビューションで利用可能なバージョン番号、testing の状態やその他有用な情報源へのリンクなどを含んでいます。

open な状態のバグやどのパッケージに対して責任を持っているのかを忘れないため、定期的に自身のデータを見直すのは良い考えです。

4.12 Debian での FusionForge の導入例: Alioth

Alioth は FusionForge (SourceForge と GForge から派生したソフトウェア) に多少変更を加えたものを基にした Debian のサービスです。このソフトは開発者が容易に使えるバグトラッキングシステム、パッチ管理、プロジェクト/タスク管理、ファイルのアップロード、メーリングリスト、VCS リポジトリなどのツールを提供します。これらの機能全てはウェブインターフェイスから管理します。

Debian がバックアップする、あるいは Debian が中心となっているフリーソフトウェアプロジェクトに対して支援を提供、外部の開発者からの支援協力を Debian によって始められたプロジェクトを与えたり、Debian やその派生ディストリビューションのプロモーションが目的であるプロジェクトを支援する機能を提供するためにあります。多くの Debian チームに非常によく利用されており、あらゆる種類の VCS リポジトリを提供しています。

全ての Debian 開発者は自動的に Alioth のアカウントを持ちます。Debian 開発者は、パスワード復旧機能を使ってアカウントを有効にすることができます。他の開発者は Alioth 上でゲストアカウントを発行してもらえます。

詳細な情報については、以下のリンクを参照下さい:

- <https://wiki.debian.org/Alioth>
- <https://wiki.debian.org/Alioth/FAQ>
- <https://wiki.debian.org/Alioth/PackagingProject>
- <https://alioth.debian.org/>

4.13 Debian 開発者と Debian メンテナへの特典

Debian 開発者および Debian メンテナが利用可能な特典は <https://wiki.debian.org/MemberBenefits> に記載されています。

Chapter 5

パッケージの取扱い方

この章では、パッケージの作成、アップロード、メンテナンス、移植についての情報を扱います。

5.1 新規パッケージ

もしあなたが Debian ディストリビューションに対して新たなパッケージを作成したいという場合、まず **作業が望まれるパッケージ (Work-Needing and Prospective Packages (WNPP))** の一覧をチェックする必要があります。WNPP 一覧をチェックすることで、まだ誰もそのソフトをパッケージ化していないことや、作業が重複していないことを確認します。詳細については **WNPP のページ** を読んでください。

パッケージ化しようとしているソフトについて、誰もまだ作業していないようであれば、まずは wnpp 擬似パッケージ (pseudo-package) に対してバグ報告を投稿する必要があります (項7.1)。このバグ報告には、パッケージの説明 (他の人がレビューできます)、作業しようとしているパッケージのライセンス、ダウンロードが可能な現在の URL を含めた新規パッケージの作成予定 (自分自身が分かるだけではないもの) を記述します。

サブジェクトを `ITP:foo --short description` に設定する必要があります。ここでは `foo` は新規パッケージの名前に置き換えます。バグ報告の重要度は `wishlist` に設定しなければなりません。X-Debbugs-CC ヘッダを使ってコピーを debian-devel@lists.debian.org に送信してください (CC: は使わないでください。CC: を使った場合はメールのサブジェクトにバグ番号が付与されないためです)。大量の新規パッケージの作成 (11 個以上) を行っている場合、メーリングリストへ個別に通知するのは鬱陶しいので、代わりにバグを登録した後で `debian-devel` メーリングリストへ要約を送信してください。これによって、他の開発者らに次に来るパッケージを知らせ、説明とパッケージ名のレビューが可能になります。

新規パッケージがアーカイブヘインストールされる際にバグ報告を自動的に閉じるため、`Closes: #nnnnn` というエントリを新規パッケージの `changelog` 内に含めてください (項5.8.4 を参照)。

パッケージについて、NEW パッケージキューの管理者への説明が必要だろうと思う場合は、`changelog` に説明を含めて ftpmaster@debian.org へ送ってください。アップロード後であればメンテナとして受け取ったメールに返信してください。もしくは既に再アップロード最中の場合は `reject` メールに対して返信してください。

セキュリティバグを閉じる場合は、CVE 番号を `Closes: #nnnnn` と同じく含めるようにしてください。これは、セキュリティチームが脆弱性を追跡するのに役立ちます。アドバイザーの ID が分かる前にバグ修正のためのアップロードが行われた場合は、以前の `changelog` エントリを次のアップロード時に修正するのが推奨されています。このような場合でも、元々の `changelog` での記載に可能な限り背景情報へのポイントを全て含めてください。

我々がメンテナに意図しているところをアナウンスする様に求めるのには、いくつかの理由があります。

- (潜在的な新たな) メンテナが、メーリングリストの人々の経験を活かすのを手助けし、もし他の誰かが既に作業を行っていた場合に知らせる。
- そのパッケージについての作業を検討している他の人へ、既に作業をしているボランティアがいることを知らせ、労力が共有される。
- debian-devel-changes@lists.debian.org に流される一行の説明文 (description) と通常どおりの「Initial release」という `changelog` エントリよりも、残った他のメンテナがパッケージに関してより深く知ることができる。

- 不安定版 (unstable) で暮らす人 (そして最前線のテスターである人) の助けになる。我々はそのような人々を推奨すべきである。
- メンテナや他に興味を持つ人々へ、プロジェクトで何が行われているのか、何か新しいことがあるかということに関して、告知は良い印象を与える。

新しいパッケージに対するよくある拒否理由については <https://ftp-master.debian.org/REJECT-FAQ.html> を参照してください。

5.2 パッケージの変更を記録する

パッケージについて行った変更は `debian/changelog` に記録されなくてはなりません。これらの変更には、何が変更されたのか、(不確かであれば) 何故なのか、そしてどのバグが閉じられたのかの簡潔な説明文を付加する必要があります。このファイルは `/usr/share/doc/package/changelog.Debian.gz`、あるいはネイティブパッケージの場合は `/usr/share/doc/package/changelog.gz` にインストールされます。

`debian/changelog` ファイルは、幾つもの異なった項目からなる特定の構造に従っています。一点を取り上げてみると、`distribution` については項5.5に記述されています。このファイルの構造について、より詳細な情報は Debian ポリシーの `debian/changelog` という章で確認できます。

`changelog` への記載は、パッケージがアーカイブにインストールされる際、自動的に Debian バグを閉じるのに利用できます。項5.8.4を参照してください。

ソフトウェアの新しい開発元のバージョン (new upstream version) を含むパッケージの `changelog` エントリは、以下のようにするのが慣習です:

```
* New upstream release.
```

`changelog` エントリの作成と仕上げ処理に使えるツールがあります。項A.6.1と項A.6.5を参照してください。

項6.3も参照してください。

5.3 パッケージをテストする

パッケージをアップロードする前に、基本的なテストをする必要があります。最低限、以下の作業が必要です (同じ Debian パッケージの古いバージョンなどが必要になるでしょう):

- パッケージをインストールしてソフトウェアが動作するのを確認する、あるいは既にそのソフトの Debian パッケージが存在している場合、パッケージを以前のバージョンから新しいバージョンにアップグレードする。
- パッケージに対して **lintian** を実行する。以下のようにして **lintian** を実行できます: `lintian -v package-version.changes` これによって、バイナリパッケージ同様にソースパッケージを確認できます。**lintian** が生成した出力を理解していない場合は、**lintian** が問題の説明を非常に冗長に出力するようにする `-i` オプションを付けて実行してみてください。
通常、**lintian** がエラーを出力するようであれば、パッケージをアップロードしてはいけません (エラーは E で始まります)。

lintian についての詳細は、項A.2.1を参照してください。

- もし古いバージョンがあれば、それからの変更点を分析するために追加で **debdiff** を実行する (項A.2.2を参照)。
- (もしあれば) 以前のバージョンにダウングレードする—これは `postrm` スクリプトと `prerm` スクリプトをテストします。
- パッケージを削除して、再インストールする。
- ソースパッケージを違うディレクトリにコピーして展開し、再構築する。これは、パッケージが外部の既存ファイルに依っているか、`.diff.gz` ファイル内に含まれているファイルで保存されている権限に依るかどうかをテストします。

5.4 ソースパッケージの概要

Debian のソースパッケージには 2 種類あります：

- いわゆる ネイティブ (native) パッケージ。元のソースと Debian で当てられたパッチの間に差が無いもの
- オリジナルのソースコードの tarball ファイルに、Debian によって作成された変更点を含む別のファイルが付随している (より一般的な) パッケージ

ネイティブパッケージの場合、ソースパッケージは Debian のソース control ファイル (.dsc) とソースコードの tarball (.tar.{gz,bz2,xz}) を含んでいます。ネイティブではないパッケージのソースパッケージは Debian のソース control ファイルと、オリジナルのソースコードの tarball (.orig.tar.{gz,bz2,xz})、そして Debian での変更点 (ソース形式 “1.0” は .diff.gz、ソース形式 “3.0 (quilt)” は .debian.tar.{gz,bz2,xz}) を含んでいます。

ソース形式 “1.0” では、パッケージが native かどうかはビルド時に **dpkg-source** によって決められていました。最近では望むソース形式を debian/source/format に “3.0 (quilt)” または “3.0 (native)” と記述することによって明示することが推奨されています。この章の残りの部分は native ではないパッケージについてのみ記しています。

初回には、特定の開発元のバージョン (upstream version) に該当するバージョンがアップロードされます。元のソース tar ファイルは、アップロードされて .changes ファイルに含まれている必要があります。その後、新しい diff ファイルや .dsc ファイルの生成には全く同じ tar ファイルを使わなければならない、これを再アップロードする必要はありません。

デフォルトでは、**dpkg-genchanges** および **dpkg-buildpackage** は前述されている changelog エントリと現在のエントリが異なった upstream バージョンを持つ場合にのみ、オリジナルのソース tar ファイルを含めようとしています。この挙動は、`-sa` を使って常に含めたり、`-sd` を使うことで常に含めないようにするように変更できます。

アップロード時にオリジナルのソースが含まれていない場合、アップロードされる .dsc と diff ファイルを構築する際に **dpkg-source** が使用するオリジナルの tar ファイルは、必ず既にアーカイブにあるものと 1 バイトも違わぬものでなくてはなりません。

注意していただきたいのですが、native ではないパッケージでは、diff はパッチ内のファイルパーミッションを保存しないので、*.orig.tar.{gz,bz2,xz} 内に存在しないファイルのパーミッションは保持されません。しかし、ソース形式 “3.0 (quilt)” を使っている場合、debian ディレクトリ内にあるファイルのパーミッションは tar アーカイブで保存されるのでそのままになります。

5.5 ディストリビューションを選ぶ

アップロードでは、パッケージがどのディストリビューション向けになっているかを指定してあることが必要です。パッケージの構築プロセスでは、debian/changelog ファイルの最初の行からこの情報を展開し、.changes ファイルの Distribution 欄に配置します。

パッケージは、通常 unstable へアップロードされます。unstable あるいは experimental へのアップロードはこれらの suite を changelog のエントリに記します。サポートされている他の suite へのアップロードは、曖昧さを避けるために suite のコードネームを使う必要があります。

実際には、他にも指定可能なディストリビューションがあります: codename-security ですが、その詳細については項 5.8.5 を読んでください。

同時に複数のディストリビューションへ、パッケージをアップロードすることはできません。

5.5.1 特別な例: 安定版 (stable) と 旧安定版 (oldstable) ディストリビューションへアップロードする

安定版 (stable) へのアップロードは、安定版リリースマネージャによるレビューのため、パッケージは proposed-updates-new キューに転送され、許可された場合は Debian アーカイブの stable-proposed-updates ディレクトリにインストールされます。ここから、ここから、安定版 (stable) の次期ポイントリリースに含まれることになります。

アップロードが許可されるのを確実にするには、アップロードの前に変更点について安定版リリースチームと協議する必要があります。そのためには、**reportbug** コマンドを使って、現在の安定版 (stable) へ適用したいパッチを含めて release.debian.org 擬似パッケージへバグを登録してください。安定版 (stable) ディストリビューションへアップロードするパッケージの changelog のエントリは、常にくどいほど詳細にしてください。

安定版 (stable) へのアップロード時には特に注意を払う必要があります。基本的に、以下のいずれかが起こった際にのみ 安定版 (stable) へパッケージはアップロードされます:

- 本当に致命的な機能の問題がある
- パッケージがインストールできなくなる
- リリースされたアーキテクチャにパッケージが無い

以前、安定版 (stable) へのアップロードはセキュリティ問題への対処と同等に取り扱われていました。しかし、この慣習は廃れており、Debian セキュリティ勧告がリリースされた際、セキュリティ勧告へのアップロードに使われたものが自動的に適切な proposed-updates アーカイブにコピーされます。セキュリティ情報の取り扱い方の詳細については項 5.8.5 を参照してください。セキュリティチームがその問題は DSA を通じて修正するには軽微過ぎると思った場合であっても、安定版のリリースマネージャらはそれに関わらず 安定版 (stable) への定期アップロードに修正を含めようとするでしょう。

些細な修正でも後ほどバグを引き起こすことがあるので、重要でないものは何であろうと変更するのは推奨されません。

安定版 (stable) にアップロードされるパッケージは安定版 (stable) を動作しているシステム上でコンパイルされていなければならない、ライブラリ (やその他のパッケージ) への依存は安定版 (stable) で入手可能なものに限られます。例えば、安定版 (stable) にアップロードされたパッケージが不安定版 (unstable) にのみ存在するライブラリパッケージに依存していると reject されます。他のパッケージへの依存を (提供 (Provides) や shlibs をいじることで) 変更するのは、他のパッケージをインストールできないようにする可能性があるため認められません。

旧安定版 (oldstable) ディストリビューションへのアップロードはアーカイブされてない限り可能です。安定版 (stable) と同じルールが適用されます。

5.5.2 特別な例: testing/testing-proposed-updates へアップロードする

詳細については、[testing section](#) にある情報を参照してください。

5.6 パッケージをアップロードする

5.6.1 ftp-master にアップロードする

パッケージをアップロードするには、ファイル (署名された changes ファイルと dsc ファイル) を anonymous ftp で ftp.upload.debian.org の [/pub/UploadQueue/](#) へアップロードする必要があります。そこでファイルを処理するためには、Debian Developers keyring または Debian Maintainers keyring (<https://wiki.debian.org/DebianMaintainer> 参照) にある鍵で署名しておく必要があります。

changes ファイルは最後に転送する必要があることに注意してください。そうしないとアーカイブのメンテナンスを行っているソフトが changes ファイルをパースして全てのファイルがアップロードされていないと判断して、アップロードは reject されるかもしれません。

パッケージのアップロードを行う際には **dupload** や **dput** が便利なことにも気づくことでしょう。これらの便利なプログラムは、パッケージを Debian にアップロードする作業を自動化するのに役立ちます。

パッケージを削除するには <ftp://ftp.upload.debian.org/pub/UploadQueue/README> と **dcut** Debian パッケージを参照してください。

5.6.2 遅延アップロード

パッケージを直ちにアップロードするのが良い時もありますが、パッケージがアーカイブに入るのが数日後であるのが良いと思う時もあります。例えば、**Non-Maintainer アップロード**の準備をする際は、メンテナに対して猶予期間を数日間与えたいと思うでしょう。

delayed ディレクトリにアップロードされると、パッケージは [the deferred uploads queue](#) に保存されます。指定した待ち時間が終わると、パッケージは処理のため通常の incoming ディレクトリに移動されます。この作業は ftp.upload.debian.org の DELAYED/x-day ディレクトリへのアップロードを通じて自動的に処理されます (x は 0 から 15 の間です)。0-day は一日に複数回 ftp.upload.debian.org へアップロードするのに使われます。

dput を使うと、パッケージを遅延キューに入れるのに `--delayed DELAY` パラメータを使えます。

5.6.3 セキュリティアップロード

セキュリティアップロードキュー (security-master.debian.org) には、セキュリティチームからの事前許可無しにパッケージをアップロードしないでください。パッケージがチームの要求に完全に合致していない場合、望まれないアップロードに対処するために多くの問題が引き起こされたり遅延が生じることになります。詳細については項5.8.5を参照してください。

5.6.4 他のアップロードキュー

ヨーロッパにはもう一つのアップロードキューが ftp://ftp.eu.upload.debian.org/pub/UploadQueue/ にあります。操作方法は ftp.upload.debian.org と同じですが、ヨーロッパ圏の開発者に対しては、より速いはずです。

パッケージは `ssh` を使って `ssh.upload.debian.org` へアップロードすることも可能です。ファイルは `/srv/upload.debian.org/UploadQueue` に置く必要があります。このキューは遅延アップロードをサポートしていません。

5.6.5 新しいパッケージがインストールされたことの通知

Debian アーカイブメンテナはパッケージのアップロードに関して責任を持っています。多くの部分は、アップロードはアーカイブ用のメンテナンスツール `dak process-upload` によって日々自動的に行われています。特に、不安定版 (unstable) に存在しているパッケージの更新は自動的に処理されます。それ以外の場合、特に新規パッケージの場合は、アップロードされたパッケージをディストリビューションに含めるのは手動で行われます。アップロードが手動で処理される場合は、アーカイブへの変更は実施されるまでに一ヶ月ほどかかります。お待ちください。

どの場合であっても、パッケージがアーカイブに追加されたことや、バグがアップロードで閉じられたことを告げるメールでの通知を受け取ることになります。あなたが閉じようとしたバグが処理されていない場合は、この通知を注意深く確認してください。

インストール通知は、パッケージがどのセクションに入ったかを示す情報を含んでいます。不一致がある場合は、それを示す別のメール通知を受け取ります。以下も参照ください。

キュー経由でアップロードした場合は、キューデーモンソフトウェアもメールで通知を行うことに留意してください。

5.7 パッケージのセクション、サブセクション、優先度を指定する

`debian/control` ファイルのセクション (Section) フィールドと優先度 (Priority) フィールドは実際にアーカイブ内でどこに配置されるか、あるいはプライオリティが何かという指定ではありません。`debian/control` ファイル中の値は、実際のところは単なるヒントです。

アーカイブメンテナは、`override` ファイル内でパッケージについて定められたセクションと優先度を常に確認しています。`override` ファイルと `debian/control` で指定されたパッケージのフィールドに不一致がある場合、パッケージがアーカイブにインストールされる際に相違について記述されたメールを受け取ります。`debian/control` ファイルを次のアップロード時に修正することもできますし、`override` ファイルに変更を加えるように依頼するのもよいでしょう。

パッケージが現状で置かれているセクションを変更するには、まずパッケージの `debian/control` ファイルが正しいことを確認する必要があります。次に、ftp.debian.org に対し、あなたのパッケージに対するセクションあるいは優先度について古いものから新しいものへ変更する依頼のバグ登録をします。`override:PACKAGE1:section/priority, [...], PACKAGEX:section/priority` のようなサブジェクトを使い、バグ報告の本文に変更に関する根拠を記述してください。

`override` ファイルについての詳細は、`dpkg-scanpackages(1)` と <https://www.debian.org/Bugs/Developer#maintinincorrect> を参照してください。

項4.6.1で書かれているように、セクション (Section) フィールドにはセクション同様にサブセクションも記述するのに注意ください。セクションが `main` の場合は、それは書かないようにしてください。利用可能なサブセクションは <https://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections> で検索できます。

5.8 バグの取扱い

すべての開発者は Debian バグ追跡システムを取り扱えるようであればいけません。これは、どの様にしてバグ報告を正しく登録するか (項7.1参照)、どの様に更新及び整理するか、そしてどの様にして

処理をして完了するかを知っていることを含みます。

バグ追跡システムの機能は、**Debian BTS 開発者向け情報**に記載されています。これには、バグの完了処理・追加メッセージの送信・重要度とタグを割り当てる・バグを転送済み (Forwarded) にする・その他が含まれています。

バグを他のパッケージに割り当てし直す、同じ問題についての別々のバグ報告をマージする、早まってクローズされたバグの再オープンなどの作業は、いわゆる制御メールサーバと呼ばれるものを使って処理されています。このサーバで利用可能なすべてのコマンドは、**BTS 制御サーバドキュメント**に記載されています。

5.8.1 バグの監視

良いメンテナになりたい場合は、あなたのパッケージに関する **Debian バグ追跡システム (BTS)** のページを定期的にチェックする必要があります。BTS には、あなたのパッケージに対して登録されている全てのバグが含まれています。登録されているバグについては、以下のページを参照することで確認できます: <https://bugs.debian.org/yourlogin@debian.org>

メンテナは、bugs.debian.org のメールアドレス経由で BTS に対応します。利用可能なコマンドについてのドキュメントは <https://www.debian.org/Bugs/> で参照可能ですし、もし doc-debian パッケージをインストールしてあれば、ローカルファイル /usr/share/doc/debian/bug-* で見ることも可能です。

定期的にオープンになっているバグについてのレポートを受け取るのも良いでしょう。あなたのパッケージでオープンになっているバグの全一覧を毎週受け取りたい場合、以下のような cron ジョブを追加します:

```
# 自分のパッケージにあるバグのレポートを毎週取得する
0 17 * * fri    echo "index maint address" | mail request@bugs.debian.org
```

`address` は、あなたの公式な Debian パッケージメンテナとしてのメールアドレスに置き換えてください。

5.8.2 バグへの対応

バグに対応する際は、バグについて行った議論がバグの元々の報告者とバグ自身 (例えば 123@bugs.debian.org) の両方に送られているのを確認してください。新しくメールを書いて元々の報告者のメールアドレスを思い出せない場合は、123-submitter@bugs.debian.org というメールアドレスが報告者へ連絡するのと、さらにバグのログへあなたがメールしたのを記録するのにも使えます (これは 123@bugs.debian.org ヘメールのコピーを送らなくても済むことを意味しています)。

FTBFS である旨のバグを受け取った場合、これはソースからビルドできないこと (Fails to build from source) を意味します。移植作業をしている人たちはこの略語をよく使います。

既にバグに対処していた場合 (例えば修正済みの時)、説明のメッセージを 123-done@bugs.debian.org に送ることで done とマークしておいて (閉じて) ください。パッケージを変更してアップロードすることでバグを修正する場合は、項 5.8.4 に記載されているように自動的にバグを閉じることができます。

close コマンドを control@bugs.debian.org に送って、バグサーバ経由でバグを閉じるのは決してしてはいけません。そのようにした場合、元々の報告者は何故バグが閉じられたのかという情報を得られません。

5.8.3 バグを掃除する

パッケージメンテナになると、他のパッケージにバグを見つけたり、自分のパッケージに対して報告されたバグが実際には他のパッケージにあるバグであったりということが頻繁にあるでしょう。バグ追跡システムの機能は **Debian 開発者向けの BTS ドキュメント** に記載されています。バグ報告の再指定 (reassign) やマージ (merge)、そしてタグ付けなどの作業は **BTS 制御サーバのドキュメント** に記述されています。この章では、Debian 開発者から集められた経験を元にしたバグの扱い方のガイドラインを含んでいます。

他のパッケージで見つけた問題についてバグを登録するのは、メンテナとしての責務の一つです。詳細については項 7.1 を参照してください。しかし、自分のパッケージのバグを管理するのはさらに重要です。

以下がバグ報告を取り扱う手順です:

1. 報告が実際にバグに関連するものか否かを決めてください。ユーザはドキュメントを読んでいないため、誤ったプログラムの使い方をしているだけのことが時々あります。このように判断した

場合は、ユーザに問題を修正するのに十分な情報を与えて (良いドキュメントへのポイントを教えるなどして) バグを閉じます。同じ報告が何度も繰り返される場合には、ドキュメントが十分なものかどうか、あるいは有益なエラーメッセージを与えるよう、誤った使い方を検知していないのでは、と自身に問い直してください。これは開発元の作者に伝える必要がある問題かもしれません。

バグを閉じるという貴方の判断にバグ報告者らが同意しない場合には、それをどう取り扱うかについての同意が見つかるまで、彼らは再度オープンな状態 (reopen) にするでしょう。そのバグについてもう議論することが無いという場合は、バグは存在するが修正することはないと知らせるため、バグに対して wontfix タグを付けることになります。この決定が受け入れがたい時には、あなた (あるいは報告者) はバグを tech-ctte に再指定 (reassign) して技術委員会 (technical committee) の判断を仰いでください (パッケージへ報告されたものをそのままにしておきたい場合は、BTS の clone コマンドを使ってください)。これを行う前には**推奨手順**を読んでおいてください。

2. バグが実際にあるが、他のパッケージによって引き起こされている場合は、バグを正しいパッケージに再指定 (reassign) します。どのパッケージに再指定するべきかが分からない場合は、IRC か debian-devel@lists.debian.org で聞いてください。再指定するパッケージのメンテナに通知をしてください。例えば packagename@packages.debian.org 宛にメッセージを Cc: してメール中に理由を説明するなどします。単に再指定しただけでは再指定された先のメンテナにはメールは送信されませんので、彼らがパッケージのバグ一覧を見るまでそれを知ることはありません。

バグがあなたのパッケージの動作に影響する場合は、バグを複製し (clone)、複製したバグをその挙動を実際に起こしているパッケージに再指定することを検討してください。さもなければ、あなたのパッケージのバグ一覧にバグが見つからないので、多分ユーザに同じバグを何度も繰り返し報告される羽目になる可能性があります。あなたは、再指定 (reassign) によって「自分の」バグということを防ぎ、バグの複製 (clone) によって関係があることを記載しておく必要があります。

3. 時々、重要度の定義に合うようにバグの重要度を調整する必要があります。これは、人々はバグ修正を確実に早くしてもらうために重要度を極端に上げようとするからです。要望された変更点が単に体裁的なものな時には、バグは要望 (wishlist) に格下げされるでしょう。
4. バグが確かにあるが既に他の誰かによって同じ問題が報告されている場合は、2つの関連したバグを BTS の merge コマンドを使って1つにマージします。このようにすると、バグが修正された時に全ての投稿者に通知がいきます (ですが、そのバグ報告の投稿者へのメールは報告の他の投稿者には自動的に通知されないことに注意してください)。merge コマンドや類似の unmerge コマンドの技術詳細については、BTS 制御サーバドキュメントを参照してください。
5. バグ報告者は情報を書き漏らしている場合、必要な情報を尋ねる必要があります。その様なバグに印をつけるには moreinfo タグを使います。さらに、そのバグを再現できない場合には、unreproducible タグを付けます。誰もそのバグを再現できない場合、どうやって再現するのか、さらに情報を何ヶ月経っても、この情報が誰からも送られてこない場合はバグは閉じて構いません。
6. バグがパッケージに起因する場合、さっさと直します。自分では直せない場合は、バグに help タグを付けます。 debian-devel@lists.debian.org や debian-qa@lists.debian.org で助けを求めることも出来ます。開発元 (upstream) の問題であれば、作者に転送する必要があります。バグを転送するだけでは十分ではありません。リリースごとにバグが修正されているかどうかを確認しなければいけません。もし修正されていれば、それを閉じ、そうでなければ作者に確認を取る必要があります。必要な技能を持っていてバグを修正するパッチが用意できる場合は、同時に作者に送りましょう。パッチを BTS に送付してバグに patch タグを付けるのを忘れないでください。
7. ローカル環境でバグを修正した、あるいは VCS リポジトリに修正をコミットした場合には、バグに pending タグを付けてバグが修正されたことと次のアップロードでバグが閉じられるであろうことを回りに知らせます (changelog に closes: を追加します)。これは複数の開発者が同一のパッケージで作業している際に特に役立ちます。
8. 一旦修正されたパッケージがアーカイブから入手可能になったら、バグはどのバージョンで修正されたかを指定して閉じられる必要があります。これは自動的に行われます。項5.8.4を読んでください。

5.8.4 新規アップロードでバグがクローズされる時

バグや問題があなたのパッケージで修正されたとしたら、そのバグを閉じるのはパッケージメンテナとしての責任になります。しかし、バグを修正したパッケージが Debian アーカイブに受け入れられるまではバグを閉じてはいけません。従って、一旦更新したパッケージがアーカイブにインストールされたという通知を受け取った場合は、BTS でバグを閉じることができますし、そうしなければいけません。もちろん、バグは正しいバージョンで閉じなくてはなりません。

ですが、アップロード後に手動でバグをクローズしなくても済む方法があります—debian/changelog に以下の特定の書き方にて修正したバグを列挙すれば、それだけで後はアーカイブのメンテナがバグをクローズしてくれます。例:

```
acme-cannon (3.1415) unstable; urgency=low

* Frobbbed with options (closes: Bug#98339)
* Added safety to prevent operator dismemberment, closes: bug#98765,
  bug#98713, #98714.
* Added man page. Closes: #98725.
```

技術的に言うと、どの様にしてバグを閉じる changelog が判別されているかを以下の Perl の正規表現にて記述しています:

```
/closes:\s*(?:bug)?\#\s*\d+(?:,\s*(?:bug)?\#\s*\d+)*\s*/ig
```

closes:#xxx という書き方が推奨されています。これは、最も分かり易いエントリで、changelog の本文に挿入するのがもっとも簡単だからです。dpkg-buildpackage に -v スイッチを指定して別バージョンを指定しない限り、最も新しい changelog のエントリにあるバグだけが閉じられます (基本的には、です。正確には .changes ファイルの changelog-part で明示されたバグが閉じられます)。

歴史的に、**non-maintainer upload (NMU)** と判別されるアップロードは closed ではなく fixed とタグがされてきましたが、この習慣はバージョントラッキングの進化によって廃れています。同じことが fixed-in-experimental タグにも言えます。

もしバグ場号を間違えて入力したり、changelog のエントリにバグを入れ忘れた場合、そのミスが起こすであろうダメージを防ぐのを躊躇わないでください。誤って閉じたバグを再度オープンにするには、バグトラッキングシステムのコントロールアドレスである control@bugs.debian.org に reopen xxx コマンドを投げます。アップロードで修正されたがまだ残っているバグを閉じるには .changes ファイルを xxx-done@bugs.debian.org にメールします。xxx はバグ番号で、メールの本文の最初の 2 行に Version: yyy と空白行を入れます。yyy はバグが修正された最初のバージョンです。

上に書いたような changelog を使ったバグの閉じ方は必須ではない、ということは念頭に置いておいてください。行ったアップロードとは無関係に単にバグを閉じたい、という場合は、説明をメールに書いて xxx-done@bugs.debian.org に送ってバグを閉じてください。そのバージョンのパッケージでの変更がバグに何も関係ない場合は、そのバージョンの changelog エントリではバグを閉じないでください。

どのように changelog のエントリを書くのか、一般的な情報については項6.3 を参照してください。

5.8.5 セキュリティ関連バグの取扱い

機密性が高いその性質上、セキュリティ関連のバグは注意深く取り扱わねばなりません。この作業をコーディネートし、未処理のセキュリティ問題を追いつけ、セキュリティ問題についてメンテナを手助けしたり修正自体を行い、セキュリティ勧告を出し、security.debian.org を維持するために Debian セキュリティチームが存在します。

Debian パッケージ中のセキュリティ関連のバグに気づいたら、あなたがメンテナであるかどうかに関わらず、問題に関する正確な情報を集め、まずは team@security.debian.org 宛にメールを出してセキュリティチームへ連絡を取ってください。お望みであれば、Debian セキュリティ担当窓口の鍵を使ってメールを暗号化できます。詳細は <https://www.debian.org/security/faq#contact> を参照してください。チームに問い合わせること無く 安定版 (stable) 向けのパッケージをアップロードしないでください。例として、役に立つ情報は以下のようなものになります:

- バグが既に公開されているか否か
- バグによって、どのバージョンが影響を受けると分かっているか。サポートされている Debian のリリース、ならびにテスト版 (testing) 及び 不安定版 (unstable) にある各バージョンをチェックしてください。

- 利用可能なものがあれば、修正内容 (パッチが特に望ましい)
- 自身で準備した修正パッケージ (まずは項5.8.5.4を読んで、debdiffの結果、あるいは.diff.gzと.dscファイルだけを送ってください)
- テストについて何かしらの手助けになるもの (攻撃コード、リグレッションテストなど)
- 勧告に必要な情報 (項5.8.5.3参照)

パッケージメンテナとして、あなたは安定版リリースについてもメンテナンスする責任を持ちます。あなたがパッチの評価と更新パッケージのテストを行うのに最も適任な人です。ですから、以下のセキュリティチームによって取り扱ってもらうため、どのようにしてパッケージを用意するかについての章を参照してください。

5.8.5.1 セキュリティ追跡システム

セキュリティチームは集約的なデータベース、**Debian セキュリティ追跡システム (Debian Security Tracker)**をメンテナンスしています。これはセキュリティ問題として知られている全ての公開情報を含んでいます: どのパッケージ/バージョンが影響を受ける/修正されているか、つまりは安定版、テスト版、不安定版が脆弱かどうか、という情報です。まだ機密扱いの情報は追跡システムには追加されません。

特定の問題について検索することもできますし、パッケージ名でも検索できます。あなたのパッケージを探して、どの問題がまだ未解決かを確認してください。できれば追加情報を提供するか、パッケージの問題に対処するのを手伝ってください。やり方は追跡システムのウェブページにあります。

5.8.5.2 秘匿性

Debian 内での他の多くの活動とは違い、セキュリティ問題に関する情報については、暫くの間秘密にしておく必要がしばしばあります。これによって、ソフトウェアのディストリビュータがユーザが危険にさらされるのを最小限にするため、公開時期を合わせることができます。今回がそうであるかは、問題と対応する修正の性質や、既に既知のものとなっているかどうかによります。

開発者がセキュリティ問題を知る方法はいくつかあります:

- 公開フォーラム (メーリングリスト、ウェブサイトなど) で知らせる
- 誰かがバグ報告を登録している
- 誰かがプライベートなメールで教えてきた

最初の二つのケースでは、情報は公開されていて可能な限り早く修正することが重要です。しかしながら最後のケースは、公開情報ではないかもしれません。この場合は、問題に対処するのに幾つか取り得る選択肢があります:

- セキュリティの影響度が小さい場合、問題を秘密にしておく必要はなく、修正を行ってリリースするのが良い場合がしばしばあります。
- 問題が深刻な場合、他のベンダと情報を共有してリリースをコーディネートする方が望ましいでしょう。セキュリティチームは様々な組織/個人と連絡を取りつづけ、この問題に対応することができます。

どのような場合でも、問題を報告した人がこれを公開しないように求めているのであれば、明白な例外として Debian の安定版リリースに対する修正を作成してもらうためにセキュリティチームへ連絡すること以外、この様な要求は尊重されるべきです。機密情報をセキュリティチームに送る場合は、この点を明示しておくのを忘れないでください。

機密を要する場合は、修正を不安定版 (unstable) (や公開 VCS リポジトリなどその他どこへも) へ修正をアップロードしないよう、注意してください。コードその物が公開されている場合、変更の詳細を難読化するだけでは十分ではなく、皆によって解析され得る (そしてされる) でしょう。

機密であることを要求されたにも関わらず、情報を公開するのには2つの理由があります: 問題が一定期間既知の状態になっている、あるいは問題や攻撃コードが公開された場合です。

セキュリティチームは、機密事項に関して通信を暗号化できる PGP 鍵を持っています。詳細については、**セキュリティチーム FAQ**を参照してください。

5.8.5.3 セキュリティ勧告

セキュリティ勧告は現在のところ、リリースされた安定版ディストリビューションについてのみ、取り扱われます。テスト版 (testing) や 不安定版 (unstable) についてはありません。リリースされると、セキュリティ勧告は email-debian-security-announce; メーリングリストに送られ、[セキュリティのウェブページ](#)に掲載されます。セキュリティ勧告はセキュリティチームによって記述、掲載されます。しかし、メンテナが情報を提供できたり、文章の一部を書けるのであれば、彼らは当然そんなことは気にしません。勧告にあるべき情報は以下を含んでいます:

- 以下のようなものを含めた問題の説明と範囲:
 - 問題の種類 (権限の上昇、サービス拒否など)
 - 何の権限が得られるのか、(もし分かれば) 誰が得るのか
 - どのようにして攻撃が可能なのか
 - 攻撃はリモートから可能なのかそれともローカルから可能なのか
 - どのようにして問題が修正されたのか

この情報によって、ユーザがシステムに対する脅威を評価できるようになります。

- 影響を受けるパッケージのバージョン番号
- 修正されたパッケージのバージョン番号
- どこで更新されたパッケージを得るかという情報 (通常は Debian のセキュリティアーカイブからです)
- 開発元のアドバイザーへの参照、[CVE](#) 番号、脆弱性の相互参照について役立つその他の情報

5.8.5.4 セキュリティ問題に対処するパッケージを用意する

あなたがセキュリティチームに対し、彼らの職務に関して手助けできる方法の一つは、安定版 Debian リリース用のセキュリティ勧告に適した修正版パッケージを提供することです。

安定版について更新が作成される際、システムの挙動の変化や新しいバグの導入を避けるように注意が必要です。これを行うため、バグを修正するための変更は可能な限り少なくします。ユーザや管理者は一旦リリースされたものの厳密な挙動を当てにしているのです、どのような変更でも誰かのシステムを壊しかねません。これは特にライブラリについて当てはまります: API や ABI を決して変更していないことを確認してください。変更がどれほど小さいものでも関係ありません。

これは、開発元の新しいリリースバージョン (new upstream version) への移行が良い解決策ではないことを意味しています。代わりに、関連する変更を現在の Debian 安定版リリースに存在しているバージョンへバックポートすべきです。通常、開発元のメンテナは助けが必要であれば手伝おうとしてくれます。そうでない場合は、Debian セキュリティチームが手助けすることができます。

いくつかのケースでは、例えば大量のソースコードの変更や書き直しが必要など、セキュリティ修正をバックポートできないことがあります。この様な場合、新しいバージョン (new upstream version) へ移行する必要があるかもしれません。しかし、これは極端な状況の場合にのみ行われるものであり、実行する前に必ずセキュリティチームと調整をしなければなりません。

これに関してはもう一つ重要な指針があります: 必ず変更についてテストをしてください。攻撃用コード (exploit) が入手可能な場合には、それを試してみて、パッチを当てていないパッケージで成功するか、修正したパッケージでは失敗することかどうかを確かめてみてください。他の確認として、セキュリティ修正は時折表面上はそれと関係が無いような機能を壊すことがあるので、通常の動作も同様にテストしてください。

脆弱性の修正に直接関係しない変更をパッケージへ加えないようにしてください。この様な変更は元に戻さなくてはならなくなるだけで、時間を無駄にします。他に直したいバグがパッケージにある場合は、セキュリティ勧告が発行された後、通常通りに proposed-update にアップロードを行ってください。セキュリティ更新の仕組みは、それ以外の方法では安定版リリースから reject されるであろう変更をあなたのパッケージに加える方法ではありませんので、この様な事はしないでください。

変更点を可能な限り見直してください。以前のバージョンとの変更点を繰り返し確認してください (これには patchutils パッケージの `interdiff` や devscripts の `debdiff` が役立ちます。項 [A.2.2](#) を参照してください)。

以下の項目を必ず確認してください

- `debian/changelog` で正しいディストリビューションを対象にする: `codename-security` (例えば `jessie-security`)。 `distribution-proposed-updates` や `stable` を対象にしないでください!
- アップロードは **urgency=high** で行う必要があります。
- 説明が十分にされている、意味がある `changelog` エントリを書くこと。他の人は、これらを元に特定のバグが修正されているのかを見つけ出します。登録されている **Debian** バグに対して `closes:` 行を追加すること。外部のリファレンス、できれば **CVE** 識別番号を常に含めること、そうすれば相互参照が可能になります。しかし、CVE 識別番号がまだ付与されていない場合には、それを待たずに作業を進めてください。識別番号は後ほど付与することができます。
- バージョン番号が正しいことを確認する。現在のパッケージより大きく、しかし以降のディストリビューションよりパッケージバージョンが小さい必要があります。分からない場合は `dpkg --compare-versions` でテストしてください。以前のアップロードで既に使っているバージョン番号を再利用しないように注意してください。そうしないと番号が `binNMU` と衝突します。 `+debXul` (`x` はメジャーリリース番号) を追加するのが通例です。例えば `1:2.4.3-4+deb8u1` とします。もちろん `1` はアップロードするごとに増やします。
- これまでに (以前のセキュリティ更新によって) `security.debian.org` へ開発元のソースコードをアップロードしていなければ、開発元のソースコードを全て含めてアップロードするパッケージをビルドする (`dpkg-buildpackage -sa`)。以前、同じ開発元のバージョンで `security.debian.org` にアップロードしたことがある場合は、開発元のソースコード無しでアップロードしても構いません (`dpkg-buildpackage -sd`)。
- 通常のアーカイブで使われているのと全く同じ ***.orig.tar.{gz,bz2,xz}** を必ず使うようにしてください。さもなくば、後ほどセキュリティ修正を `main` アーカイブに移動することができません。
- ビルドを行っているディストリビューションからインストールしたパッケージだけが含まれているクリーンなシステム上でパッケージをビルドしてください。その様なシステムを自分で持っていない場合は、`debian.org` マシン (項4.4 を参照してください) を使うこともできますし、`chroot` を設定することもできます (項A.4.3 と項A.4.2 を参照してください)。

5.8.5.5 修正したパッケージをアップロードする

セキュリティアップロードキュー (`security-master.debian.org`) には、セキュリティチームからの事前許可無しにパッケージをアップロードしないでください。パッケージがチームの要求に完全に合致していない場合、望まれないアップロードに対処するために多くの問題が引き起こされたり遅延が生じることになります。詳細については項5.8.5 を参照してください。

セキュリティチームと調整する事無しに `proposed-updates` へ修正したものをアップロードしないようにしてください。 `security.debian.org` からパッケージは `proposed-updates` ディレクトリに自動的にコピーされます。アーカイブに同じ、あるいはより高いバージョン番号のものが既にインストールされている場合は、セキュリティアップデートはアーカイブシステムに `reject` されます。そうすると、安定版ディストリビューションはこのパッケージに対するセキュリティ更新無しで終了してしまうでしょう。

一旦、新しいパッケージを作成してテストし、セキュリティチームによって許可を受けたら、アーカイブへインストールできるようにするためにアップロードをする必要があります。セキュリティに関するアップロードの場合、アップロード先は `ftp://security-master.debian.org/pub/SecurityUploadQueue/` になります。

セキュリティキューへアップロードしたものが許可されると、パッケージは自動的にすべてのアーキテクチャに対してビルドされ、セキュリティチームによる確認の為に保存されます。

許可、あるいは確認を待っているアップロードには、セキュリティチームのみがアクセスできます。これは、まだ公開できないセキュリティ問題の修正があるかも知れないためです。

セキュリティチームのメンバーがパッケージを許可した場合は、`proposed` パッケージに対する `ftp-master.debian.org` 上の適切な `distribution-proposed-updates` と同様に `security.debian.org` 上にインストールされます。

5.9 パッケージの移動、削除、リネーム、放棄、引き取り、再導入

アーカイブの変更作業のいくつかは、Debian のアップロードプロセスでは自動的なものにはなっていません。これらの手続きはメンテナによる手動での作業である必要があります。この章では、この様な場合に何をするかガイドラインを提供します。

5.9.1 パッケージの移動

時折、パッケージは属しているセクションが変わることがあります。例えば「non-free」セクションのパッケージが新しいバージョンで GPL になった場合、パッケージは「main」か「contrib」に移動する必要があります。¹

パッケージのどれかがセクションを変更する必要がある場合、希望するセクションにパッケージを配置するためパッケージの control 情報を変更してから再アップロードします (詳細については [Debian ポリシーマニュアル](#) を参照してください)。必ず `.orig.tar.{gz,bz2,xz}` を (開発元のバージョンが新しいものになったのでも) アップロードに含める必要があります。新しいセクションが正しい場合は、自動的に移動されます。移動されない場合には、何が起こったのかを理解するために ftpmaster に連絡を取ります。

一方で、もしパッケージの一つのサブセクション (例: 「devel」「admin」) を変更する必要がある、という場合には、手順は全く異なります。パッケージの control ファイルにあるサブセクションを修正して、再アップロードします。また、項 5.7 に記述してあるように override ファイルを更新する必要があるでしょう。

5.9.2 パッケージの削除

何らかの理由でパッケージを完全に削除したくなった場合 (もう必要がなくなった古い互換用ライブラリの場合、など)、パッケージを削除するよう ftp.debian.org に対してバグ登録をする必要があります; すべてのバグ同様、通常このバグは重要度 normal です。バグの題名は RM: パッケージ名 [アーキテクチャー] --理由 という形式である必要があります。パッケージ名は削除されるパッケージ、理由は削除を依頼する理由の短い要約です。[アーキテクチャー] はオプションで、削除依頼が全アーキテクチャではなく一部のアーキテクチャのみに適用される場合にのみ、必要となります。reportbug は ftp.debian.org 擬似パッケージに対してバグを報告しようとした場合に、これらのルールに則ってバグの題名を作成しようとすることに注意してください。

あなたがメンテナンスしているパッケージを削除したくなった場合は、題名の先頭に ROM (Request Of Maintainer) という文字を付けたバグにこれを記述する必要があります。パッケージの削除理由に使われる他の一般的な略語がいくつかありますので、完全な一覧については <https://ftp-master.debian.org/removals.html> を参照してください。このページでは、まだ作業されていない削除依頼の便利な一覧も見ることができます。

削除は不安定版 (unstable)、実験版 (experimental)、安定版 (stable) ディストリビューションに対してのみ実施が可能であることに注意してください。パッケージはテスト版 (testing) から直接は削除されません。代わりに不安定版 (unstable) から削除された後で自動的に削除され、テスト版 (testing) にあるパッケージは削除されたパッケージに依存しなくなります (release.debian.org 擬似パッケージへ削除依頼のバグレポートを登録することで、テスト版 (testing) からの削除は可能ではありません。項 5.13.2.2 の項目を参照して下さい)。

例外として、明示的な削除依頼が必要ない場合が一つあります: (ソース、あるいはバイナリ) パッケージがソースからビルドされなくなった場合、半自動的に削除されます。バイナリパッケージの場合、これはこのバイナリパッケージを生成するソースパッケージがもはや存在しないということを意味します。バイナリパッケージがいくつかのアーキテクチャで生成されなくなったという場合には、削除依頼は必要です。ソースパッケージの場合は、関連の全バイナリパッケージが別のソースパッケージによって上書きされるのを意味しています。

削除依頼では、依頼を判断する理由を詳細に書く必要があります。これは不必要な削除を避け、何故パッケージが削除されたのかを追跡できるようにするためです。例えば、削除されるパッケージにとって代わるパッケージの名前を記述します。

通常は自分がメンテナンスしているパッケージの削除のみを依頼します。その他のパッケージを削除したい場合は、メンテナの許可を取る必要があります。パッケージが放棄されたのでメンテナがいない場合は、まず debian-qa@lists.debian.org で削除依頼について議論をしてください。パッケージの削除についての合意ができたなら、削除依頼の新規バグを登録するのではなく、wnpp パッケージに対して登録されているバグを reassign して 0: に題名を変更するべきです。

¹ パッケージがどのセクションに属するかガイドラインは [Debian ポリシーマニュアル](#) を参照してください。

この件、あるいはパッケージ削除に関するその他のトピックについて、さらなる情報を https://wiki.debian.org/ftpmaster_Removals や <https://qa.debian.org/howto-remove.html> で参照できます。

パッケージを破棄しても構わないか迷う場合には、意見を聞きに debian-devel@lists.debian.org ヘメールしてください。apt の **apt-cache** プログラムも重要です。apt-cache showpkg パッケージ名として起動した際、プログラムはパッケージ名の非依存関係を含む詳細を表示します。他にも **apt-cache rdepends**、**apt-rdepends**、**build-rdeps** (devscripts パッケージに含まれる)、**grep-dctrl** などの有用なプログラムが非依存関係を含む情報を表示します。みなしご化されたパッケージの削除は、debian-qa@lists.debian.org で話し合われます。

一旦パッケージが削除されたら、パッケージのバグを処理する必要があります。実際のコードが別のパッケージに含まれるようになったので、別のパッケージへバグを付け替える (例えば、libfoo13 が上書きするので、libfoo12 が削除される) か、あるいはソフトウェアがもう Debian の一部では無くなった場合にはバグを閉じるかする必要があります。バグを閉じる場合、過去の Debian のリリースにあるパッケージバージョンで修正されたとマークするのを避けてください。バージョン `<most-recent-version-ever-in-Debian>+rm` で修正されたとマークしなければなりません。

5.9.2.1 Incoming からパッケージを削除する

以前は、incoming からパッケージを削除することが可能でした。しかし、新しい incoming システムが導入されたことにより、これはもはや不可能となっています。その代わりに、置き換えたいパッケージよりも高いバージョンのリビジョンの新しいパッケージをアップロードする必要があります。両方のバージョンのパッケージがアーカイブにインストールされますが、一つ前のバージョンはすぐに高いバージョンで置き換えられるため、実際にはバージョンが高い方だけが不安定版 (unstable) で利用可能になります。しかし、もしあなたがパッケージをきちんとテストしていれば、パッケージを置き換える必要はそんなに頻繁には無いはずです。

5.9.3 パッケージをリプレースあるいはリネームする

あなたのパッケージのどれかの開発元のメンテナらが、ソフトウェアをリネームするのを決めた時 (あるいはパッケージを間違えて名前を付けた時)、以下の二段階のリネーム手続きに従う必要があります。最初の段階では、debian/control ファイルに新しい名前を反映し、利用しなくなるパッケージ名に対して **Replace**、**Provides**、**Conflicts** を設定する変更をします (詳細に関しては [Debian ポリシーマニュアル1](#) を参照)。注意してほしいのは、利用しなくなるパッケージ名がリネーム後も動作する場合のみ、**Provides** を付け加えるべきだということです。一旦パッケージをアップロードがアップロードされてアーカイブに移動したら、ftp.debian.org に対してバグを報告してください (項 [5.9.2](#) 参照)。同時にパッケージのバグを正しく付け替えるのを忘れないでください。

他に、パッケージの作成でミスをしたので置き換えたいという場合があるかもしれません。これを行う方法は唯一つ、バージョン番号を上げて新しいバージョンをアップロードすることです。通常、古いバージョンは無効になります。これはソースを含めた各パッケージ部分に適用されることに注意してください: パッケージの開発元のソース tarball を入れ替えたい場合には、別のバージョンをつけてアップロードする必要があります。よくある例は `foo_1.00.orig.tar.gz` を `foo_1.00+0.orig.tar.gz`、あるいは `foo_1.00.orig.tar.bz2` で置き換えるというものです。この制約によって、ftp サイト上で各ファイルが一意の名前を持つことになり、ミラーネットワークをまたがった一貫性を保障するのに役立ちます。

5.9.4 パッケージを放棄する

パッケージをもうメンテナンスできなくなってしまった場合、ほかの人に知らせて、パッケージが放棄 (orphaned) とマークされたのが分かるようにする必要があります。パッケージメンテナを Debian QA Group `<packages@qa.debian.org>` に設定し、疑似パッケージ wnpp に対してバグ報告を送信しなければなりません。バグ報告は、パッケージが今放棄されていることを示すように `O: パッケージ名 --短い要約というタイトル` にする必要があります。バグの重要度は **通常** (normal) に設定しなければなりません; パッケージの重要 (priority) が standard より高い場合には **重要** (important) に設定する必要があります。必要だと思うのなら、メッセージの X-Debbugs-CC: ヘッダのアドレスに debian-devel@lists.debian.org を入れてコピーを送ってください (そう、CC: を使わないでください。その理由は、CC: を使うと、メッセージの題名がバグ番号を含まないからです)。

パッケージを手放したいが、しばらくの間はメンテナンスを継続できる場合には、代わりに wnpp へ RFA: パッケージ名 --短い要約という題名でバグ報告を送信する必要があります。RFA は Request For Adoption (引き取り依頼) を意味しています。

より詳細な情報は [WNPP ウェブページ](#)にあります。

5.9.5 パッケージを引き取る

新たなメンテナが必要なパッケージの一覧は [作業が望まれるパッケージ \(WNPP, Work-Needing and Prospective Packages list\)](#) で入手できます。WNPP でリストに挙がっているパッケージのどれかに対するメンテナを引き継ぎたい場合には、情報と手続きについては前述のページを確認してください。

メンテナになっていないと思うパッケージを単純に持っていても構いませんか? —それはパッケージの乗っ取り (hijacking) です。できますが、もちろんのこと、現在のメンテナに確認をとってパッケージを持って行って良いか尋ねましょう。メンテナが AWOL (absent without leave、無届け欠席状態) であると信ずる理由があれば、[項7.4](#)を参照してください。

一般的に、現在のメンテナの同意なしでパッケージを引き取るべきではありません。彼らがあなたのことを無視したのだとしても、それはパッケージを引き取る理由とはなりません。メンテナへの不満は開発者のメーリングリストへ送られるべきです。議論が良い結論に至らない、かつ問題が技術的なものであれば、技術委員会に相談することを検討してください (より詳細については、[Debian 技術委員会のページ](#)を参照してください)。

古いパッケージを引き継いだ場合は、おそらくバグ追跡システムでパッケージの公式メンテナとして表示されるようにしたいことでしょう。これは、一旦 Maintainer 欄を更新した新しいバージョンをアップロードすれば自動的に行われますが、アップロードが完了してから数時間はかかります。しばらくは新しいバージョンをアップロードする予定が無い場合は、[項4.10](#)を使ってバグ報告を受け取ることができます。しかし、以前のメンテナにしばらくの間はバグ報告が届き続けても問題無いことを確認してください。

5.9.6 パッケージの再導入

パッケージは、リリースクリティカルなバグやメンテナ不在、不人気あるいは全体的な品質の低さ等により削除されることがよくあります。再導入プロセスはパッケージ化の開始時と似ていますが、あらかじめその歴史的経緯を調べておくことにより、落とし穴にはまるのをいくらか避けることができます。

まず初めに、パッケージが削除された理由を確認しましょう。この情報はそのパッケージの PTS ページのニュースから削除の項目か[削除ログ](#)を探すことにより見つけられます。削除のバグにはそのパッケージが削除された理由や、そのパッケージの再導入にあたって必要なことがいくらか提示されているでしょう。パッケージの再導入ではなくどこか他のソフトウェアの一部への乗り替えが最適であるということが提示されているかもしれません。

以前のメンテナに連絡を取り、パッケージの再導入のために作業していないか、パッケージ共同保守に関心はないか、必要になったときにパッケージのスポンサーをしてくれないか、等を確認しておくとう良いでしょう。

新しいパッケージ ([項5.1](#)) の導入前に必要なことは全てやりましょう。

利用できる中で適切な最新のパッケージをベースに作業しましょう。これは unstable の最新版かもしれません。また、[snapshot アーカイブ](#)にはまだ存在するでしょう。

前のメンテナにより利用されていたバージョン管理システムに有用な変更が記録されているかもしれないので、確認してみるのはいいことです。以前のパッケージの control ファイルにそのパッケージのバージョン管理システムにリンクしているヘッダが無い、それがまだ存在するか確認してください。

(testing や stable、oldstable ではなく) unstable からパッケージが削除されると、そのパッケージに関連するバグは全て閉じられます。閉じられたバグを全て (アーカイブされているバグを含めて) 確認し、+rm で終わるバージョンで閉じられていて現在でも有効なものを全て unarchive および reopen してください。有効ではなくなっているものは修正されているバージョンがわかればすべて修正済みとしてください。

5.10 移植作業、そして移植できるようにすること

Debian がサポートするアーキテクチャの数は増え続けています。あなたが移植作業家ではない、あるいは別のアーキテクチャを使うことが無いという場合であっても、移植性の問題に注意を払うことはメンテナとしてのあなたの義務です。従って、あなたが移植作業家でなくても、この章の大半を読む必要があります。

移植作業 (Porting) とは、パッケージメンテナが生成したバイナリパッケージの元々のアーキテクチャとは違うアーキテクチャの Debian パッケージをビルドする作業です。これは非常にユニークかつ極めて重要な活動です。事実、移植作業家は実際の Debian パッケージのコンパイルの大半を行っています。

す。例えば、メンテナが(移植可能な)ソースパッケージと i386 のバイナリをアップロードすると、他の各アーキテクチャ、11 以上の数のビルドが生成されます。

5.10.1 移植作業に対して協力的になる

移植作業は、難解かつ他には無いタスクを抱えています。それは、彼らは膨大な量のパッケージに対処する必要があるからです。理想を言えば、すべてのソースパッケージは変更を加えないできちんとビルドできるべきです。残念なことに、その様な場合はほとんどありません。この章は Debian メンテナによってよくコミットされる「潜在的な問題」のチェックリストを含んでいます—よく移植作業者を困らせ、彼らの作業を不必要に難解にする共通の問題です。

まず初めの、そして最も重要な点は、バグや移植作業から投げかけられた問題に素早く回答することです。移植作業者をパッケージの副メンテナ (co-maintainer) であるように丁寧に扱ってください (ある意味、その通りではありません)。簡潔、あるいは不明瞭なバグ報告に対して寛容になってください。問題が何であれ、原因を捉えることに最善を尽くしてください。

移植作業者が遭遇する問題のほとんどは、何といたっても、ソースパッケージ内でのパッケージ作成のバグによって引き起こされます。以下は、確認あるいは注意すべき項目のリストです。

1. debian/control 中の Build-Depends と Build-Depends-Indep の設定が正しいことを確認してください。これを検証するのに最も良い方法は debootstrap パッケージを使って 不安定版 (unstable) の chroot 環境を作成することです (項 A.4.2 参照)。chroot 環境内では、build-essential パッケージと、Build-Depends または Build-Depends-Indep に記載されている依存パッケージをインストールしてください。最後に、chroot 環境でパッケージの生成を試してください。これらの手順は pbuilder パッケージで提供される同名のプログラムの利用によって自動化することができます (項 A.4.3 参照)。

chroot を正しく設定できない場合は、dpkg-depcheck が手助けになることでしょう (項 A.6.6 参照)。

ビルドの依存情報の指定方法については、Debian ポリシーマニュアルを参照してください。

2. 意図がある場合以外は、アーキテクチャの値を all または any 以外に指定しないでください。非常に多くの場合、メンテナが Debian ポリシーマニュアルの指示に従っていません。アーキテクチャを単一のものに指定する (i386 あるいは amd64 など) は大抵誤りです。
3. ソースパッケージが正しいことを確かめてください。ソースパッケージが正しく展開されたのを確認するため、dpkg-source -x package.dsc を実行してください。そして、ここでは、一からパッケージを dpkg-buildpackage でビルドするのに挑戦してみてください。
4. debian/files や debian/substvars を含んだソースパッケージを出していないかを確かめてください。これらは、debian/rules の clean ターゲットによって削除されるべきです。
5. ローカル環境にインストールされていたり、弄くられている設定やプログラムに依存していないことを確かめてください。例えば、/usr/local/bin やその類のプログラムは決して呼び出してはいけません。特殊なやり方で設定されたプログラムには依存しないようにしてください。別のマシンでパッケージをビルドしてください。それが同じアーキテクチャであっても、です。
6. 構築中の既にインストールしてあるパッケージに依存しないでください (上記の話の一例です)。もちろん、このルールには例外はありますが、そのような場合には手で一から環境を構築する必要があり、パッケージ作成マシンで自動的に構築することはできません。
7. 可能であれば、特定のバージョンのコンパイラに依存しないでください。もし無理であれば、その制約をビルドの依存関係に反映されているのを確認してください。だとしても異なったアーキテクチャでは時折異なったバージョンのコンパイラで統一されているので、それでも恐らく問題を引き起こすことになるでしょう。
8. debian/rules で、Debian ポリシーマニュアルが定めるように、binary-arch 及び binary-indep ターゲットに分かれて含まれていることを確かめてください。両方のターゲットが独立して動作するのを確かめてください。つまり、他のターゲットを事前に呼び出さなくても、ターゲットを呼び出せるのを確かめるということです。これをテストするには、dpkg-buildpackage -B を実行してください。

5.10.2 移植作業者のアップロード (porter upload) に関するガイドライン

パッケージが移植作業を行うアーキテクチャで手を入れずに構築できるのであれば、あなたは幸運で作業は簡単です。この章は、その様な場合に当てはめられます: きちんとアーカイブにインストールされるために、どうやってバイナリパッケージを構築・アップロードするかを記述しています。他のアーキテクチャでコンパイルできるようにするため、パッケージにパッチを当てる必要がある場合は、実際のところ、ソース NMU を行なうので、代わりに項 5.11.1 を参照してください。

移植作業者のアップロード (porter upload) は、ソースに何も変更を加えません。ソースパッケージ中のファイルには触る必要はありません。これは `debian/changelog` を含みます。

dpkg-buildpackage を `dpkg-buildpackage -B -mporter-email` として起動してください。もちろん、`porter-email` にはあなたのメールアドレスを設定します。これは `debian/rules` の `binary-arch` を使ってパッケージのバイナリ依存部分のみのビルドを行います。

移植作業のために Debian マシン上で作業をしていて、アーカイブに入れてもらうためにアップロードするパッケージにローカルでサインする必要がある場合は、`.changes` に対して **debsign** を手軽に実行するのもできますし、**dpkg-sig** のリモート署名モードを使うこともできます。

5.10.2.1 再コンパイル、あるいは binary-only NMU

時折、最初の移植作業者のアップロード作業は困難なものになります。パッケージが構築された環境があまり良くないからです (古すぎる、使われていないライブラリがある、コンパイラの問題、などなど)。その場合には、更新した環境で再コンパイルする必要があるでしょう。しかし、この場合にはバージョンを上げる必要があり、古いおかしなパッケージは Debian アーカイブ中で入れ替えられることになります (現在利用可能なものよりバージョン番号が大きくない場合、**dak** は新しいパッケージのインストールを拒否します)。

binary-only NMU がパッケージをインストール不可能にしてしまっていないことを確認する必要があります。ソースパッケージが、`dpkg` の `substitution` 変数 `$(Source-Version)` を使って内部依存関係を生成しているアーキテクチャ依存パッケージとアーキテクチャ非依存パッケージを生成した場合に起こる可能性があります。

`changelog` の変更が必要かどうかに関わらず、これらは **binary-only NMU** と呼ばれます—この場合には、他の全アーキテクチャで古すぎるかどうかや再コンパイルが必要かなどを考える必要はありません。

このような再コンパイルは、特別な「magic」バージョン番号を付けるのを必要とするので、アーカイブのメンテナンスツールは、これを理解してくれます。新しい Debian バージョンで、対応するソースアップデートが無くて、です。これを間違えた場合、アーカイブメンテナは (対応するソースコードが欠落している) アップロードを拒否します。

再コンパイルのみの NMU への「magic」は、`b` 番号という形式に従った、パッケージのバージョン番号に対するサフィックスを追加することで引き起こされます。例えば、再コンパイル対象の最新バージョンが `2.9-3` の場合、バイナリのみの NMU は `2.9-3+b1` というバージョンになる必要があります。最新のバージョンが `3.4+b1` (つまり、ネイティブパッケージで、前回が再コンパイルの NMU) の場合、バイナリのみの NMU は `3.4+b2` というバージョン番号にならねばいけません。²

最初の移植作業者のアップロード (porter upload) と同様に、パッケージのアーキテクチャ依存部分をビルドするための **dpkg-buildpackage** の正しい実行の仕方は `dpkg-buildpackage -B` です。

5.10.2.2 あなたが移植作業者の場合、source NMU を行う時は何時か

移植作業者は、通常は非移植作業者同様に項 5.11 にあるガイドラインに沿ってソース NMU を行います。しかし、移植作業者のソース NMU に対する待ち時間は非移植作業者より小さくなります。これは、移植作業は大量のパッケージに対応する必要があるからです。さらに、状況はパッケージがアップロードされるディストリビューションに依って変わります。これは、アーキテクチャが次の安定版リリースに含められるかどうかによっても変わります。リリースマネージャはどのアーキテクチャが候補なのかを決定してアナウンスします。

あなたが不安定版 (unstable) へ NMU を行う移植作業者の場合、移植作業についての上記のガイドライン、そして 2 つの相違点に従う必要があります。まず、適切な待ち時間でサブバグが BTS へ投稿されてから NMU を行って OK になるまでの間—移植作業者が不安定版 (unstable) ディストリビューションに対して行う場合は 7 日間になります。問題が致命的で移植作業に困難を強いるような場合には、この期間は短くできます (注意してください。この何れもがポリシーではなく、単にガイ

² 過去においては、再コンパイルのみの状態を意味するために、このような NMU はリビジョンの Debian 部分の三つ目の番号を使っていました。しかし、この記法はネイティブパッケージの場合に曖昧で、同一パッケージでの再コンパイルのみの NMU と、ソース NMU と、セキュリティ NMU の正しい順序が付けられなかったため、この新しい記法で置き換えられました。

ドラインに沿って相互に了解されているだけです)。安定版 (stable) や テスト版 (testing) へのアップロードについては、まず適切なリリースチームと調整をしてください。

次に、ソース NMU を行う移植作業者は BTS へ登録したバグの重要度が `serious` かそれ以上であることを確認してください。これは単一のソースパッケージが、すべての Debian でサポートされているアーキテクチャでコンパイルされたことをリリース時に保証します。数多くのライセンスに従うため、すべてのアーキテクチャについて、単一のバージョンのバイナリパッケージとソースパッケージを持つことがとても重要です。

移植作業者は、現在のバージョンのコンパイル環境やカーネル、libc にあるバグのために作られた単なる力業のパッチを極力回避すべきです。この様なでっぴ上げの代物があるのは、仕方がないことが時折あります。コンパイラのバグやその他の為にでっぴ上げを行う必要がある場合には、`#ifdef` で作業したものが動作することを確認してください。また、力業についてドキュメントに載せてください。一旦外部の問題が修正されたら、それを削除するのを皆が知ることができます。

移植作業者は、待ち期間の間、作業結果を置いておける非公式の置き場所を持つこともあります。移植版を動作させている人が、待ち期間の間であっても、これによって移植作業者による作業の恩恵を受けられるようになります。もちろん、この様な場所は、公式な恩恵や状況の確認を受けることはできませんので、利用者は注意してください。

5.10.3 移植用のインフラと自動化

パッケージの自動移植に役立つインフラストラクチャと複数のツールがあります。この章には、この自動化とこれらのツールへの移植の概要が含まれています。全体の情報に付いてはパッケージのドキュメントからリファレンスを参照してください。

5.10.3.1 メーリングリストとウェブページ

各移植版についての状況を含んだウェブページは <https://www.debian.org/ports/> から参照できます。

Debian の各移植版はメーリングリストを持っています。移植作業のメーリングリストは <https://lists.debian.org/ports.html> で見るすることができます。これらのリストは移植作業者の作業の調整や移植版のユーザと移植作業者をつなぐために使われています。

5.10.3.2 移植用ツール

移植用のツールの説明をいくつか項 A.7 で見るすることができます。

5.10.3.3 wanna-build

wanna-build システムは、分散型の、クライアント・サーバでの構築配布システムとして利用されています。通常、これは `buildd` プログラムが動作しているビルドデーモンと連携して使われます。ビルドデーモンは、ビルドに必要なパッケージの一覧を受け取るために中央の wanna-build システムと通信する「slave」ホストです。

wanna-build は、まだパッケージとしては入手可能になっていません。ですが、すべての Debian の移植作業ではパッケージ構築作業の自動化にこれが使われています。実際のパッケージ構築に使われるツール、`sbuild` はパッケージとして利用可能です。項 A.4.4 で説明を参照してください。パッケージ化されたバージョンは、ビルドデーモンで使われているものとは同じではありませんが、問題を再現するには十分なものである点に注意ください。

wanna-build によって生成される移植作業者にとって大抵有用であるデータの多くは、ウェブ上の <https://buildd.debian.org/> で入手可能です。このデータには、毎晩更新される統計情報や、queue 情報、ビルド失敗のログが含まれています。

我々はこのシステムを極めて誇りに思っています。何故ならば、様々な利用方法の可能性からです。独立した開発グループは、実際に一般的な用途に合うかどうか分からない異なった別アプローチの Debian にシステムを使うことができます (例えば、`gcc` の配列境界チェック付きでビルドした Debian など)。そして、Debian がディストリビューション全体を素早く再コンパイルできるようにもなります。

`buildd` の担当である wanna-build チームには、`debian-wb-team@lists.debian.org` で連絡が取れます。誰 (wanna-build チーム、リリースチーム) に連絡を取るのか、どうやって (メール、BTS) 連絡するのかを決めるには、<https://lists.debian.org/debian-project/2009/03/msg00096.html> を参照してください。

`binNMU` や `give-back` (ビルド失敗後のやり直し) を依頼する時には、<https://release.debian.org/wanna-build.txt> で記述されている形式を使ってください。

5.10.4 あなたのパッケージが移植可能なものではない場合

いくつかのパッケージでは、Debian でサポートされているアーキテクチャのうちの幾つかで、構築や動作に問題を抱えており、全く移植できない、あるいは十分な時間内では移植ができないものがあります。例としては、SVGA に特化したパッケージ (i386 と amd64 のみで利用可能) や、すべてのアーキテクチャではサポートされていないようなハードウェア固有の機能があります。

壊れたパッケージがアーカイブにアップロードされたり builddd の時間が無駄に費やされたりするのを防ぐため、幾つかしなければならぬことがあります:

- まず、サポートできないアーキテクチャ上ではパッケージがビルドに失敗するのを確認しておく必要があります。これを行うには幾つかやり方があります。お勧めの方法は構築時に機能をテストする小さなテストスイートを用意して、動かない場合に失敗するようにすることです。これは、全てのアーキテクチャ上で、壊れたものをアップロードするのを防ぎ、必要な機能が動作するようになればパッケージがビルドできるようになる、良い考えです。

さらに、サポートしているアーキテクチャ一覧が一定量であると信ずるのであれば、debian/control 内で any からサポートしているアーキテクチャの一覧に変更すべきです。この方法であれば、ビルドが同様に失敗するようになるのに加え、実際に試すことなく人間である読み手にサポートしているアーキテクチャが分かるようになります。

- autobuilder が必要もなくパッケージをビルドしようとしないうちに、**wanna-build** スクリプトが使うリストである Packages-arch-specific に追加しておく必要があります。現在のバージョンは <https://anonscm.debian.org/cgit/mirror/packages-arch-specific.git/tree/Packages-arch-specific> から入手できます: 変更依頼をする相手はファイルの一番上を参照してください。

サポートしていないアーキテクチャ上でビルドが失敗するようにせずに、パッケージを単に Packages-arch-specific に付け加えるだけでは不十分であることに注意してください: 移植作業、あるいはあなたのパッケージをビルドしようとしている他の人は、それが動かないのに気づかないで誤ってアップロードするかもしれません。過去に、サポートされていないアーキテクチャ上にバイナリパッケージがアップロードされてしまった場合、削除依頼は ftp.debian.org に対するバグを登録することによって行われました。

5.10.5 non-free のパッケージを auto-build 可能であるとマークする

non-free セクションのパッケージは、デフォルトでは autobuilder ネットワークではビルドされません (多くの場合は、パッケージのライセンスによって許可されていないためです)。パッケージをビルドできるようにするには、以下の手順を踏む必要があります:

- 法的に許可されているか、技術的にパッケージが auto-build 可能かどうかを確認する;
- debian/control のヘッダ部分に XS-Autobuild:yes を追加する;
- メールを nonfree@release.debian.org に送り、何故パッケージが合法的、かつ技術的に auto-build できるものなのかを説明する

5.11 Non-Maintainer Upload (NMU)

すべてのパッケージには最低一人のメンテナがいます。通常、この人達がパッケージに対して作業をし、新しいバージョンをアップロードします。時折、他の開発者らが新しいバージョンをアップロードできると便利な場合があります。例えば、彼らがメンテナンスしていないパッケージにあるバグを修正したいが、メンテナが問題に対応するには助けが必要な場合です。このようなアップロードは *Non-Maintainer Upload (NMU)* と呼ばれます。

5.11.1 いつ、どうやって NMU を行うか

NMU を行う前に、以下の質問について考えてください:

- NMU がメンテナを援助するような形にしましたか? メンテナが実際に助けを必要としているかどうか、意見に不一致があるかもしれないため、DELAYED キューが存在しています。DELAYED キューは、メンテナに対処する時間を与えるために存在しており、NMU diff の個別レビューが可能になるという有益な影響があります。

- NMU がバグを本当に修正しますか? (“バグ” はあらゆる種類のバグを意味しています。例えば新しいバージョンをパッケージにしてほしいという wishlist バグもそうですが、メンテナへの影響を最小化するように注意を払う必要があります)。NMU において、些細な表面的な問題やパッケージングスタイルの変更 (例えば cdb から dh への変更) を行うのは推奨されていません。
- メンテナに十分な時間を与えましたか? BTS にバグが報告されたのは何時ですか? 一、二週間忙しいことはあり得ないことでは無いです。そのバグはすぐに修正しなければならないほど重大ですか? それとも、あと数日待てるものですか?
- その変更にとどれくらい自信がありますか? ヒポクラテスの誓いを思い出してください: 「何よりも、害を及ぼすことなかれ」動かないパッチを当てるよりもパッケージの重大なバグをそのままオープンな状態にしておく方が良いですし、パッチによってバグを解決するのではなく隠してしまうかもしれません。自分が 100% 何をしたのか分かっていないのであれば、他の人からアドバイスをもらうのも良い考えでしょう。NMU で何かを壊したのであれば、多くの人にとって不幸になるであろうことを覚えておいてください。
- 少なくとも BTS で、NMU するのを明確に表明しましたか? 何も反応が得られなかった場合、他の手段 (プライベートなメール、IRC) でメンテナに連絡をとるのも良い考えです。
- メンテナがいつも活動的に応答してくれる場合、彼に連絡を取ろうとしましたか? 大概の場合、メンテナ自身が問題に対応して、あなたのパッチをレビューする機会が与えられる方が好ましいと思われます。これは、NMU をする人が見落とししているだろう潜在的な問題にメンテナは気付くことができるからです。大抵、メンテナが自分でアップロードする機会が与えられる方が、皆の時間を使うよりも良いやり方です。

NMU をする際には、まず NMU をする意図を明確にしておかねばなりません。それから、BTS へ現在のパッケージと提案する NMU との差分をパッチとして送付する必要があります。devscripts パッケージにある `nmudiff` スクリプトが役に立つでしょう。

パッチを準備している間、メンテナが利用しているであろうパッケージ固有の慣例に注意を向けた方が良いでしょう。これを考慮に入れることは、通常のパッケージの作業工程に対してあなたの変更が入る負担を減らし、それに従って変更が入る可能性を高めます。パッケージ固有の慣例がある可能性があるのを探すと良い場所は、[debian/README.source](#) です。

そうすべき十二分な理由が無い限り、メンテナに対応する時間を与えるべきです (例えば DELAYED キューにアップロードすることによってこれを行います)。以下が delayed キューを使う際のお勧めの値です:

- 7 日以上経っているリリースクリティカルバグのみを修正するアップロードで、バグに対するメンテナの活動が 7 日間見られなく、修正が行われている形跡が無い: 0 日
- 7 日以上経っているリリースクリティカルバグのみを修正するアップロード: 2 日
- リリースクリティカルバグや重要なバグの修正のみのアップロード: 5 日
- 他の NMU: 10 日

この値は例に過ぎません。セキュリティ問題を修正するアップロードや、移行を阻む些細なバグを修正するなど、いくつかのケースでは修正されたパッケージが不安定版 (unstable) にすぐ入るようになるのは望ましいことです。

時々、リリースマネージャが特定のバグに対して短い delay 日数の NMU を許可を認めることがあります (7 日より古いリリースクリティカルバグなど)。また、一部のメンテナは [Low Threshold NMU list](#) に自身を挙げており、遅延なしの NMU アップロードを許可しています。しかしどのような場合であっても、特にパッチが BTS で以前手に入らなかったり、メンテナが大抵アクティブであることを知っている場合など、アップロードの前にメンテナに対して数日与えるのは良い考えです。

NMU アップロード後、あなたは自分が導入したであろう問題に責任を持つことになります。パッケージを見張らなければなりません (これを行うには PTS 上のパッケージを購読するのが良い方法です)。

これは、軽率な NMU を行うための許可証ではありません。明らかにメンテナがアクティブで時期を逃さずパッチについて対応している場合や、このドキュメントに書かれている推奨を無視している場合など、あなたによるアップロードはメンテナと衝突を起こすでしょう。NMU のメリットについて、自分が行ったことの賢明さを常に弁護できるようにしておく必要があります。

5.11.2 NMU と debian/changelog

他の(ソース)アップロード同様、NMU は `debian/changelog` にそのアップロードで何を変更したのかを示すエントリを追加せねばなりません。エントリの最初の行は、このアップロードが NMU であることを明白に示す必要があります。例えばこうです:

```
* Non-maintainer upload.
```

NMU のバージョンのつけ方は、ネイティブなパッケージとネイティブではないパッケージでは異なります。

パッケージがネイティブパッケージの場合(バージョン番号に Debian リビジョンが付かない)、バージョンはメンテナの最後のアップロードのバージョン `++nmuX` となり、 X は 1 から始まる数字になります。最後のアップロードが同様に NMU の場合は、数字を増やします。例えば、現在のバージョンが 1.5 だとすると、NMU はバージョンが `1.5+nmu1` になります。

パッケージがネイティブパッケージではない場合は、バージョン番号の Debian リビジョン部分(最後のハイフン以下の部分)にマイナーバージョン番号を追加します。例えば、現在のバージョンが 1.5-2 であれば、NMU は `1.5-2.1` というバージョンになります。開発元のバージョンが新しくなったものが NMU でパッケージになった場合は、Debian リビジョンは 0 に設定されます。例えば `1.6-0.1` です。

どちらの場合でも、最後のアップロードも NMU だった場合には数字が増えます。例えば、現在のバージョンが `1.5+nmu3` (既に NMU されたネイティブパッケージ)の場合、NMU は `1.5+nmu4` というバージョンになります。

特別なバージョン付け方法が必要とされるのは、メンテナの作業を混乱させるのを避けるためです。何故ならば、Debian リビジョンのために整数を使っていると、NMU の際に既に準備されていたメンテナによるアップロードや、さらには `ftp NEW queue` にあるパッケージともぶつかる可能性があります。これには、アーカイブのパッケージが公式メンテナによるものではない、と視覚的に明らかにする利点もあります。

パッケージをテスト版や安定版にアップロードする場合、バージョン番号を「分岐」する必要があります。これは例えばセキュリティアップロードが該当します。そのため、`+debXuY` 形式のバージョン番号を使うようにしてください。 X はメジャーリリース番号で Y は 1 から始まるカウンターです。例えば、`jessie` (Debian 8) が安定版の間は安定版バージョン `1.5-3` のパッケージへのセキュリティ NMU ならバージョン `1.5-3+deb8u1` となりますが、`stretch` へのセキュリティ NMU ではバージョン `1.5-3+deb9u1` となります。

5.11.3 DELAYED/ キューを使う

NMU の許可を求めた後で待っているのは効率的ではありません。NMU した人が作業にもどるために頭を切り替えるのに手間がかかるからです。DELAYED キュー(項 5.6.2 参照)は、開発者が NMU をするのに必要な作業を同時にできるようになります。例えば、メンテナに対して更新したパッケージを 7 日後にアップロードするのを伝えるのではなく、パッケージを DELAYED/7 にアップロードしてメンテナに対して対応するのに 7 日間あることを伝えるべきです。この間、メンテナはもっとアップロードを遅らせるかアップロードをキャンセルするかを尋ねることができます。

DELAYED キューは、無用のプレッシャーをメンテナに与えるために使われるべきではありません。特に、メンテナはアップロードを自身ではキャンセルできないので、`delay` が完了する前にアップロードをキャンセルあるいは遅らせることができるのはあなただ、という点が重要です。

DELAYED を使った NMU を行って `delay` が完了する前にメンテナがパッケージを更新した場合には、アーカイブに既により新しいバージョンがあるので、あなたのアップロードは拒否されます。理想的なのは、メンテナがそのアップロードであなたが提案した変更(あるいは少なくとも対応した問題の解決方法)を含めるのを取り仕切ることです。

5.11.4 メンテナの視点から見た NMU

誰かがあなたのパッケージを NMU した場合、これは彼らがパッケージを良い形に保つのを助けたいと思っているということです。これによって、ユーザへ修正したパッケージをより早く届けます。NMU した人に、パッケージの副メンテナになることを尋ねるのを考えてみるのも良いでしょう。パッケージに対して NMU を受け取るのは悪いことではありません。それは、単にそのパッケージが他の人が作業する価値があるという意味です。

NMU を承認するには、変更と `changelog` のエントリを次のメンテナアップロードに含めます。バグは BTS で `close` されたままになりますが、パッケージのメンテナバージョンに影響していると表示されます。

5.11.5 ソース NMU vs バイナリのみの NMU (binNMU)

NMU のフルネームはソース NMU です。もう一つ別の種類があって、バイナリのみの NMU (*binary-only NMU*) あるいは *binNMU* と名付けられています。binNMU も、パッケージメンテナ以外の誰かによるパッケージのアップロードです。しかし、これはバイナリのみのアップロードです。

ライブラリ (や他の依存関係) が更新された時、それを使っているパッケージを再ビルドする必要があるかもしれません。ソースへの変更は必要ないので、同じソースパッケージが利用されます。

binNMU は、通常 `wanna-build` によって `buildd` 上で引き起こされます。debian/changelog にエントリが追加され、なぜアップロードが必要だったのか、という説明と項 5.10.2.1 で記述されているようにバージョン番号を増やします。このエントリは、その次のアップロードに含めるべきではありません。

buildd は、アーカイブするために、バイナリのみのアップロードとして、そのアーキテクチャに対してパッケージをアップロードします。厳密に言えば、これは binNMU です。しかし、これは通常 NMU とは呼ばれず、debian/changelog にエントリを追加しません。

5.11.6 NMU と QA アップロード

NMU は、割り当てられているメンテナ以外の誰かによるパッケージのアップロードです。自分のものではないパッケージのアップロードについては、もう一つ、別の種類のアップロードがあります: QA アップロードです。QA アップロードは、放棄されたパッケージのアップロードです。

QA アップロードは、ほとんど通常のメンテナによるアップロードと同じです: 些細な問題であっても、なんでも修正します。バージョン番号の付け方は通常のもので、delay アップロードをする必要もありません。違いは、パッケージのメンテナあるいはアップローダとして記載されていない点です。また、QA アップロードの changelog のエントリは以下のように最初の一行が特別になっています:

```
* QA upload.
```

あなたが NMU をしたいと思い、かつ、メンテナが活動的ではない場合、パッケージが放棄されていないかどうかを確認するのが賢明です (この情報はパッケージ追跡システム (PTS) のページで表示されています)。放棄されたパッケージに対して最初の QA アップロードを行うときは、メンテナは Debian QA Group <packages@qa.debian.org> に設定する必要があります。まだ QA アップロードがされていない放棄されたパッケージには、以前のメンテナが設定されています。この一覧は <https://qa.debian.org/orphaned.html> で手に入ります。

QA アップロードをする代わりに、メンテナをあなた自身に変更してパッケージを引き取ることも考えられます。放棄されたパッケージを引き取るのには、誰からの許可も必要としません。メンテナをあなた自身に設定して新しいバージョンをアップロードするだけです (項 5.9.5 を参照)。

5.11.7 NMU とチームアップロード

パッケージングチーム (Maintainer あるいは Uploader としてメーリングリストを使う。項 5.12 参照) の一員であるため、時々パッケージを修正あるいは更新しているが、常にこの特定パッケージに貢献する予定は無いので自分を Uploaders には加えたくはない、という時があります。これがあなたのチームの方針に沿っているなら、直接 Maintainer 欄や Uploader 欄に記載せずとも通常のアップロードが可能です。この場合、changelog のエントリを以下の行で始める必要があります:

```
* Team upload.
```

5.12 共同メンテナンス

共同メンテナンス (collaborative maintenance) は、Debian パッケージのメンテナンス責任を数人でシェアすることを指す用語です。この共同作業は、通常はより上質で短いバグ修正時間をもたらすので、大抵の場合は常に良い考えです。優先度が標準 (standard) あるいは基本セット (base) の一部であるパッケージは、共同メンテナ (co-maintainer) を持つことを強くお勧めします。

大抵の場合、主メンテナに加えて一人か二人の共同メンテナが居ます。主メンテナは debian/control ファイルの Maintainer 欄に名前が記載されている人です。共同メンテナは他のすべてのメンテナで、通常 debian/control ファイルの Uploaders に記載されています。

もっとも基本的なやり方では、新しい副メンテナの追加は大変簡単です:

- 共同メンテナが、あなたがビルドしたパッケージのソースにアクセスできるように設定します。一般に、これはあなたが CVS や Subversion のようなネットワークを利用するバージョン管理システムを利用しているということを意味しています。Alioth (項4.12 参照) はこの様なツールを提供しており、他でも同様です。
- 共同メンテナの正確なメンテナ名とアドレスを `debian/control` ファイルの最初の段落の `Uploaders` 欄に追加します。

```
Uploaders: John Buzz <jbuzz@debian.org>, Adam Rex <arex@debian.org>
```

- PTS (項4.10) を使う場合、共同メンテナは適切なソースパッケージの購読をする必要があります。

共同メンテナンスのもう一つの形態はチームメンテナンスです。これは、複数のパッケージを同じ開発者グループでメンテナンスする場合にお勧めです。その場合、各パッケージの `Maintainer` 欄と `Uploaders` 欄は注意して扱わねばいけません。以下の二つの案からいずれかを選ぶのがお勧めです:

1. パッケージの主に担当をするチームメンバーを `Maintainer` 欄に追加します。Uploaders 欄には、メーリングリストのアドレスとパッケージの面倒をみるチームメンバーを追加します。
2. メーリングリストのアドレスを `Maintainer` 欄に追加します。Uploaders 欄には、パッケージの面倒をみるチームメンバーを追加します。この場合、メーリングリストは (購読者以外に対するモデレーションなどの) 人手を介さずにバグ報告を受け取ることを確認する必要があります。

どのような場合でも、すべてのチームメンバーを `Uploaders` 欄に入れるのは良くない考えです。これは、Developer's Package Overview の一覧 (項4.11 参照) を実際には対応していないパッケージで散らかしてしまい、偽りの良いメンテナンス状態を作り出します。同じ理由から、パッケージを一回アップロードするのであれば、「チームアップロード (Team Upload)」ができるので、チームメンバーは `Uploaders` 欄へ自分を追加する必要はありません (項5.11.7 参照)。逆にいえば、`Maintainer` 欄にメーリングリストのアドレスのみで `Uploaders` 欄に誰も追加していないままにしておくのは良くない考えです。

5.13 テスト版ディストリビューション

5.13.1 基本

通常、パッケージは不安定版 (unstable) で必要ないくつかのテストを潜り抜けた後で、テスト版ディストリビューション (testing) にインストールされます。

これらは、すべてのアーキテクチャ上で同期していなければならない、インストールできなくなるような依存関係を持つてはいけません。また、テスト版 (testing) にインストールされる際に既知のリリースクリティカルバグを持っていない必要があります。このようにして、テスト版 (testing) は常にリリース候補に近いものである必要があります。詳細は以下を参照してください。

5.13.2 不安定版からの更新

テスト版 (testing) ディストリビューションを更新するスクリプトは、日に二回、更新されたパッケージのインストール直後に動作します。これらのスクリプトは `britney` と呼ばれます。これは、テスト版 (testing) ディストリビューションに対して `Packages` ファイルを生成しますが、不整合を避けてバグが無いパッケージのみを利用しようとする気の利いたやり方で行います。

不安定版 (unstable) からのパッケージの取り込みは以下の条件です:

- パッケージは、urgency に応じて (high, medium, low)、2 日、5 日、10 日の間、不安定版 (unstable) で利用可能になっていなければいけません。urgency は貼り付くことに注意してください。つまり、前回のテスト版 (testing) への移行を考慮に入れてから最大の urgency でアップロードされるということです;
- 新たなリリースクリティカルバグを持っていないこと (不安定版 (unstable) で利用可能なバージョンに影響する RC バグであって、テスト版 (testing) にあるバージョンに影響するものではない);
- あらかじめ unstable でビルドされた際に、全アーキテクチャで利用可能になっていないといけません。この情報をチェックするのに `dak ls` に興味を持つかもしれません;

- 既にテスト版 (testing) で利用可能になっているパッケージの依存関係を壊さないこと;
- パッケージが依存するものは、テスト版 (testing) で利用可能なものか、テスト版 (testing) に同時に受け入れられるものでなくてはならない(そして、それらは必要な条件をすべて満たしていれば、テスト版 (testing)) に入る);
- プロジェクトの状況。つまり、テスト版 (testing) ディストリビューションのフリーズ中は、自動的な移行がオフになります。

パッケージがテスト版 (testing) に入る処理がされるかどうかは、[テスト版ディストリビューションのウェブページ](#)のテスト版 (testing) スクリプトの出力を参照するか、devscripts パッケージ中の **grep-excuses** プログラムを使ってください。このユーティリティは、パッケージがテスト版 (testing) への進行の通知をし続けるのに、**crontab(5)** で手軽に使うことができます。

update_excuses は、なぜパッケージが拒否されたのか正確な理由を必ずしも表示しません。自分自身で何がパッケージが含まれるのを妨げているのか、探す必要があるかもしれません。[テスト版のウェブページ](#)が、そのような問題を引き起こす良くある問題についての情報を与えてくれるでしょう。

時折、相互依存関係の組み合わせが非常に難解なのでスクリプトが解決できないことがあるため、パッケージがテスト版 (testing) に決して入らないことがあります。詳細は下記を参照してください。

依存性についてのさらなる分析は、<https://release.debian.org/migration/> で表示されています—ですが、このページが **britney** が処理した依存性ではないものも表示しているのに注意してください。

5.13.2.1 時代遅れ (Out-of-date)

テスト版 (testing) への移行スクリプトの場合、時代遅れ (outdated) というのが意味しているのは: リリースアーキテクチャに対して、複数の異なったバージョンが不安定版 (unstable) にある (fuckedarches にあるアーキテクチャを除く。fuckedarches は (update_out.py 中で) 更新を行わないアーキテクチャの一覧ですが、現在のところは空になっています)。時代遅れ (outdated) の場合、テスト版 (testing) でこのパッケージが存在しているアーキテクチャに対して全く何もしません。

以下の例を考えてみましょう:

	alpha	arm
テスト版	1	-
不安定版	1	2

不安定版 (unstable) での alpha のパッケージは時代遅れになっているので、テスト版 (testing) に入りません。パッケージの削除は全く役に立たず、alpha ではパッケージは時代遅れのままで、テスト版 (testing) には移行しません。

ですが、もしも **ftp-master** が不安定版 (unstable) のパッケージ (ここでは arm の) を削除した場合:

	alpha	arm	hurd-i386
テスト版	1	1	-
不安定版	2	-	1

この場合、パッケージは不安定版 (unstable) ですべてのリリースアーキテクチャで最新になります(それから、もう一つの **hurd-i386** は、リリースアーキテクチャではないので問題になりません)。

時折、すべてのアーキテクチャでまだビルドされていないパッケージを入れられるか、という質問がでできます: いいえ。単純にいいえ、です (glibcなどをメンテしている場合を除きます)。

5.13.2.2 テスト版からの削除

時折、パッケージは他のパッケージがテスト版へ入るために削除されます: これは、他のパッケージが他のすべての面で準備ができている場合にテスト版に入るようにする場合のみ発生します。例えば、a が新しいバージョンの b とはインストールできない場合を考えてみましょう。その場合、a は b が入るために削除されるかもしれません。

もちろん、他にもテスト版 (testing) からパッケージが削除される理由があります: バグが多すぎる場合です (そして RC バグが 1 個だけあるのもこの状態とみなすのには十分です)。

さらに、パッケージが不安定版 (unstable) から削除され、テスト版 (testing) にはこれに依存するパッケージがもうなくなった場合、パッケージは自動的に削除されます。

5.13.2.3 循環依存

britney によってうまく取扱われない状況は、パッケージ a がパッケージ b の新しいバージョンに依存していて、そしてその逆も、というものです。

この場合の例:

	テスト版	不安定版
a	1; depends: b=1	2; depends: b=2
b	1; depends: a=1	2; depends: a=2

パッケージ a あるいはパッケージ b が更新対象と見做されない。

現状、このような場合はリリースチームによる手動でのヒントが必要になります。あなたのパッケージのどれかにこのような状況が起きた場合は、debian-release@lists.debian.org にメールを送って連絡を取ってください。

5.13.2.4 テスト版 (testing) にあるパッケージへの影響

一般的に言って、不安定版 (unstable) にあるパッケージのステータスが意味するのは、不安定版 (unstable) からテスト版 (testing) へ移行する次のバージョン 2 つの例外があります: パッケージの RC バグ数が減っている場合、RC バグが残っていてもテスト版に入る可能性があります。2 つ目の例外は、テスト版 (testing) のパッケージのバージョンが異なったアーキテクチャで同期していない場合です: その場合、すべてのアーキテクチャがソースパッケージのバージョンへとアップグレードされることがあります。ですが、これはパッケージが以前にアーキテクチャが、テスト版 (testing) への移行の際、不安定版 (unstable) にそのアーキテクチャのバイナリパッケージが全くないという場合だけです。

この要旨: 影響は、テスト版 (testing) にあるパッケージが、同じパッケージの新しいバージョンになるのは、新しいバージョンの方が楽にできそうだから、ということです。

5.13.2.5 詳細について

詳細について知りたい場合ですが、britney の動作は以下ようになります:

パッケージが、適切な候補であるかどうかを決めるために確認が行われます。これによって、更新が実行されます。パッケージが候補とみなされない理由でもっともよくあるのは、まだ日数が経過していない (too young)、RC バグがある、いくつかのアーキテクチャで古くなりすぎている、というものです。britney のこの部分では、リリースマネージャーが britney がパッケージを検討できるように、hints と呼ばれる様々なサイズのハンマーを使います (下記を参照してください)。

さて、より複雑な部分に差し掛かります: Britney が適正候補を使ってテスト版 (testing) を更新しようとしています。このため、britney はテスト版ディストリビューションに個々の適正な候補を追加しようとしています。テスト版 (testing) でインストール不可能なパッケージの数が増えないのであれば、パッケージは受け入れられます。その時から、受け入れられたパッケージはテスト版 (testing) の一部として取り扱われ、このパッケージを含めるためのインストールチェックテストが引き続き行われます。リリースチームからの hints は、実際の内容に応じて、このメインの処理の前後に処理されます。

より詳細を見たい場合は、https://ftp-master.debian.org/testing/update_output/ で探することができます。

hints は、説明からも探せますが、<https://ftp-master.debian.org/testing/hints/> にあります。hints によって、Debian リリースチームはパッケージを block あるいは unblock することや、パッケージをテスト版 (testing) へ移動する手間を減らしたり強制的に移動させたり、あるいはテスト版 (testing) からパッケージを削除したり、[testing-proposed-updates](#) へアップロードを許可したり、urgency を上書きすることが可能になります。

5.13.3 直接テスト版を更新する

テスト版 (testing) ディストリビューションは、上記で説明したルールに従って不安定版 (unstable) からのパッケージで作られています。しかし、時折、テスト版 (testing) の為だけに構

築したパッケージをアップロードする必要があるという場合があります。そのためには、`testing-proposed-updates` にアップロードするのが良いでしょう。

アップロードされたパッケージは自動的に処理されず、リリースマネージャの手によって処理される必要があることに注意してください。ですので、アップロードするのに十分な理由があるのが望ましいでしょう。何が十分な理由かを知るには、リリースマネージャの視点で、彼らが定期的に debian-devel-announce@lists.debian.org に流している指示を読む必要があります。

不安定版 (`unstable`) でパッケージを更新できるのであれば、`testing-proposed-updates` にアップロードすべきではありません。更新できない場合 (例えば、不安定版 (`unstable`) に新しい開発版がある場合)、この機能を使うことができますが、まずはリリースマネージャから許可を得るのが良いでしょう。パッケージがフリーズされていても、不安定版 (`unstable`) 経由のアップロードが新たな依存関係を何も引き起こさない場合、不安定版 (`unstable`) での更新は可能です。

バージョン番号は、通常 `+debXuY` を付加することで指定されます。`x` は Debian のメジャーリリース番号で `y` は 1 から始まる数です。例: `1:2.4.3-4+deb8u1`

アップロードでは、以下の項目のいずれも見落とさないように必ずしてください:

- 本当に `testing-proposed-updates` を通す必要があり、`unstable` ではダメなことを確認する;
- 必ず、最小限な量だけの変更を含めるようにする;
- 必ず `changelog` 中に適切な説明を含める;
- 必ず、対象とするディストリビューションとして `testing code name` (e.g. `stretch`) を記述している;
- 必ず 不安定版 (`unstable`) ではなく テスト版 (`testing`) でパッケージを構築・テストしている;
- バージョン番号が `testing` および `testing-proposed-updates` のものよりも大きく、`unstable` のものよりも小さいのを確認する;
- アップロードしてすべてのプラットフォームで構築が成功したら、debian-release@lists.debian.org 宛でリリースチームに連絡を取って、アップロードを承認してくれるように依頼しましょう。

5.13.4 よく聞かれる質問とその答え (FAQ)

5.13.4.1 リリースクリティカルバグとは何ですか、どうやって数えるのですか?

ある重要度以上のバグすべてが通常リリースクリティカルであると見なされます。現在のところ、`critical` (致命的)、`grave` (重大)、`serious` (深刻) バグがそれにあたります。

そのようなバグは、Debian の 安定版 (`stable`) リリース時に、そのパッケージがリリースされる見込みに影響があるものと仮定されます: 一般的に、パッケージでオープンになっているリリースクリティカルバグがある場合、そのパッケージはテスト版 (`testing`) に入らず、その結果安定版 (`stable`) ではリリースされません。

不安定版 (`unstable`) でのバグのカウンタ数は、リリース対象アーキテクチャの不安定版 (`unstable`) で利用可能なパッケージ/バージョンの組み合わせに適用されるとマークされたすべてのリリースクリティカルバグです。テスト版 (`testing`) のバグのカウンタ数も同様に定義します。

5.13.4.2 どのようにすれば、他のパッケージを壊す可能性があるパッケージをテスト版 (`testing`) ヘインストールできますか?

ディストリビューションにおけるアーカイブの構造では、一つのバージョンのパッケージだけを持つことができ、パッケージは名前によって定義されています。そのため、ソースパッケージ `acmefoo` がテスト版 (`testing`) にインストールされると、付随するバイナリパッケージ `acme-foo-bin`、`acme-bar-bin`、`libacme-foo1`、`libacme-foo-dev` の古いバージョンが削除されます。

しかし、古いバージョンではライブラリに古い `soname` を含んだ `libacme-foo0` のようなバイナリパッケージを提供しているかもしれません。古い `acmefoo` を削除するのは `libacme-foo0` を削除することになり、これに依存しているパッケージを壊してしまいます。

おそらく、これが主に影響するのは、一群のバイナリパッケージを変更したのを別バージョンで提供しているパッケージ (つまり、主にライブラリ) でしょう。しかし、バージョン付きの依存関係が `==`、`<=`、`<<` などで宣言されているパッケージにも影響は及ぼします。

一つのソースパッケージによって提供されている一群のバイナリパッケージがこのようにして変更された場合、古いバイナリに依存しているすべてのパッケージは新しいバイナリに依存するように更新する必要があります。このようなソースパッケージをテスト版 (testing) にインストールするとテスト版 (testing) で依存しているすべてのパッケージを壊すことになるので、ここで注意をする必要があります: すべての依存しているパッケージを更新し、インストールできるように準備しておくことで壊されないようにしておき、そして、すべての準備ができれば、通常はリリースマネージャあるいはリリースアシスタントによる手動作業が必要になります。

この様に複雑な組み合わせのパッケージで問題がある場合は、debian-devel@lists.debian.org あるいは debian-release@lists.debian.org に連絡を取って手助けを求めてください。

Chapter 6

パッケージ化のベストプラクティス

Debian の品質は、全ての Debian パッケージが満たす必要がある基本的要求を明示的に規定する **Debian ポリシー** に大きく依存しています。そして、パッケージングでの長年の経験で溜め込まれた財産である、Debian ポリシーを越えて共有してきた経験の積み重ねというものもあります。多くの非常に優秀な人々が素晴らしいツールを作っており、このツールがあなた、つまり Debian のメンテナが素晴らしいパッケージを作り、維持していくのを手助けしてくれます。

この章では、Debian 開発者へのベストプラクティスをいくつか提供します。すべての勧めは単に勧めであり、要求事項やポリシーではありません。Debian 開発者らからの主観的なヒント、アドバイス、ポイントです。あなたにとって一番うまくいくやり方を、どうぞご自由に選んでください。

6.1 debian/rules についてのベストプラクティス

以下の推奨事項は、debian/rules ファイルに適用されます。debian/rules は、ビルド作業を管理し、(直接にせよ、そうでないにせよ) パッケージにどのファイルが入るかを選択します。大抵の場合、メンテナが最も時間を費やすファイルです。

6.1.1 ヘルパースクリプト

debian/rules でヘルパースクリプトを使う根拠は、多くのパッケージ間でメンテナらに共通のロジックを利用・共有させるようになるからです。メニューエントリのインストールについての問いを例にとってみましょう: ファイルを /usr/share/menu (必要であれば、実行形式のバイナリのメニューファイルの場合 /usr/lib/menu) に置き、メンテナスクリプトにメニューエントリを登録・解除するためのコマンドを追加する必要があります。これはパッケージが行う、非常に一般的なことです。なぜ個々のメンテナがこれらのすべてを自分で書き直し、時にはバグを埋め込む必要があるでしょう? また、メニューディレクトリが変更された場合、すべてのパッケージを変更する必要があります。

ヘルパースクリプトがこれらの問題を引き受けてくれます。ヘルパースクリプトの期待するやり方に従っているならば、ヘルパースクリプトはすべての詳細について考慮をします。ポリシーの変更はヘルパースクリプト中で行えます; そして、パッケージを新しいバージョンのヘルパースクリプトでリビルドする必要があるだけです。他に何の変更も必要ありません。

付録A には、複数の異なったヘルパーが含まれています。もっとも一般的で (我々の意見では) ベストなヘルパーシステムは debhelper です。debmake のような、以前のヘルパーシステムはモノリシックでした: 使えそうなヘルパーの一部を取り出して選ぶことはできず、何を行うにもヘルパーを使う必要がありました。ですが、debhelper は、いくつもの分割された小さな **dh_*** プログラムです。たとえば、**dh_installman** は man ページをインストールして圧縮し、**dh_installmenu** は menu ファイルをインストールするなどします。つまり、debian/rules 内で使える部分では小さなヘルパースクリプトを使い、手製のコマンドを使うといった十分な柔軟性を与えてくれます。

debhelper(1) を読んで、パッケージに付属している例を参照すれば、debhelper を使い始めることができます。dh-make パッケージ (項A.3.2 参照) の **dh_make** は、素のソースパッケージを debhelper 化されたパッケージに変換するのに利用できます。ですが、この近道では個々の **dh_*** ヘルパーをわざわざ理解する必要がないので、満足できないでしょう。ヘルパースクリプトを使おうとするのであれば、そのヘルパーを使うこと、つまり前提と動作を学ぶのに時間を割く必要があります。

6.1.2 パッチを複数のファイルに分離する

巨大で複雑なパッケージには、対処が必要なたくさんのバグが含まれているかもしれません。直接ソース中で大量のバグを修正し、あまり注意を払っていなかった場合、適用した様々なパッチを識別するのは難しいことになるでしょう。(全てではなく) 幾つか修正を取り入れた新しい開発元のバージョンへパッケージを更新する必要がある場合、とても悲惨なことになります。(例えば、`.diff.gz` から) `diff` をすべて適用することもできませんし、開発元で修正されたバグごとにどのパッチをバックアウトするようにすればよいのか分かりません。

幸いなことに、ソースフォーマット “3.0 (quilt)” では、パッチシステムを設定するために `debian/rules` を変更することなく、パッチを分割して保持できるようになっています。パッチは `debian/patches/` に保持され、ソースパッケージが展開されるときに `debian/patches/series` に記載されているパッチが自動的に適用されます。名前が指すように、パッチは **quilt** で管理することができます。

より古いソースフォーマット “1.0” を使っている場合でも、パッチを分割することは可能ですが、専用のパッチシステムを使う必要があります: パッチファイルは Debian パッチファイル (`.diff.gz`) 内に組み込まれ、通常 `debian/` ディレクトリ内にあります。違いは、すぐに **dpkg-source** では適用されないが、`debian/rules` の `build` ルールで `patch` ルールへの依存を通じて適用されることだけです。逆に言うと、これらのパッチは `unpatch` ルールへの依存を通じて `clean` ルールで外されます。

quilt はこの作業にお勧めのツールです。上記の全てを行う上、パッチ一覧の管理も可能です。詳細な情報は `quilt` パッケージを参照してください。

他にもパッチを管理するツールはあります。**dpatch** や、パッチシステムが統合されている `cdb`s などです。

6.1.3 複数のバイナリパッケージ

単一のソースパッケージはしばしば複数のバイナリパッケージを生成します。それは、同じソフトウェアで複数のフレーバーを提供することであったり (例: `vim` ソースパッケージ)、巨大なパッケージではなく複数の小さなパッケージを作ったりします (例: ユーザがサブセットのみをインストールできるようにして、ディスク容量を節約できます)。

二つ目の例は、`debian/rules` で簡単に扱うことができます。ビルドディレクトリからパッケージの一時ツリーへ、適切なファイルを移動する必要があるだけです。これは、**install** または `debhelper` の **dh_install** を使ってできます。パッケージ間の依存関係を `debian/control` 内で正しく設定したのを忘れずに確認してください。

最初の例は、同じソフトウェアでありながら異なった設定オプションで複数回再コンパイルする必要があるので、ちょっと難しくなります。`vim` ソースパッケージは、手作りの `debian/rules` ファイルを使ってどのようにこの作業を扱うか、という例です。

6.2 debian/control のベストプラクティス

以下のプラクティスは、`debian/control` ファイルに関するものです。**パッケージ説明文についてのポリシー**を補完します。

パッケージの説明文は、`control` ファイルの対応するフィールドにて定義されている様に、パッケージの概要とパッケージに関する長い説明文の両方を含んでいます。項6.2.1 では、パッケージ説明文の双方の部分についての一般的なガイドラインが記述されています。それによると、項6.2.2 が概要に特化したガイドラインを提供しており、そして項6.2.3 が説明文 (description) に特化したガイドラインを含んでいます。

6.2.1 パッケージ説明文の基本的なガイドライン

パッケージの説明文は平均的なユーザーに向けて書く必要があります。平均的な人というのは、パッケージを使って得をするであろう人のことです。例えば、開発用パッケージであれば開発者向けですし、彼ら向けの言葉でテクニカルに記述することができます。より汎用的なアプリケーション、例えばエディタなどであれば、あまり技術的ではないユーザ向けに書く必要があります。

パッケージ説明文のレビューを行った結果、ほとんどのものがテクニカルである、つまり、技術に詳しくはないユーザに通じるようには書かれてはいないという結論に達しました。あなたのパッケージが、本当に技術に精通したユーザ用のみではない限り、これは問題です。

どうやって技術に詳しくはないユーザに対して書けばいいのでしょうか? ジャーゴンを避けましょう。ユーザが詳しくないであろう他のアプリケーションやフレームワークへの参照を避けましょう。GNOME や KDE については、おそらくユーザはその言葉について知っているでしょうから構いません

が、GTK+ はおそらくダメです。まったく知識がないと仮定してみましょう。技術用語を使わねばならない場合は、説明しましょう。

客観的になりましょう。パッケージ説明文はあなたのパッケージの宣伝場所ではありません。あなたがそのパッケージをどんなに愛しているかは関係ありません。その説明文を読む人は、あなたが気にすることと同じことを気にはしないでであろうことを覚えておいてください。

他のソフトウェアパッケージ、プロトコル名、標準規格、仕様の名前を参照する場合には、もしあれば正規名称を使いましょう。X Windows や X-Windows や X Window ではなく、X Window System あるいは X11 または X を使いましょう。GTK や gtk ではなく GTK+ を使いましょう。Gnome ではなく GNOME を使いましょう。Postscript や postscript ではなく PostScript を使いましょう。

説明文を書くことに問題があれば、debian-l10n-english@lists.debian.org へそれを送ってフィードバックを求めるとよいでしょう。

6.2.2 パッケージの概要、あるいは短い説明文

ポリシーでは、概要行 (短い説明文) はパッケージ名を繰り返すのではなく、簡潔かつ有益なものである必要がある、となっています。

概要は、完全な文章ではなくパッケージを記述するフレーズとして機能します。ですので、句読点は不適切です: 追加の大文字や最後のピリオドは不要です。また、最初の不定冠詞や定冠詞 "a", "an", or "the" を削る必要があります。つまり、例えば以下ようになります:

```
Package: libeg0
Description: exemplification support library
```

技術的に言えば、動詞のフレーズに対して、これは名詞のフレーズから文章を差し引いたものです。パッケージ名と要約をこの決まり文句に代入できるのがよい見つけ方です:

パッケージの名前は概要を提供します。

関連パッケージ群は、概要を 2 つに分けた別の書き方をした方が良いでしょう。最初はその組一式の説明文で、その次はその組内でのパッケージの役割のサマリにします:

```
Package: eg-tools
Description: simple exemplification system (utilities)

Package: eg-doc
Description: simple exemplification system - documentation
```

これらの要約が、手が加えられた決まり文句に続きます。パッケージ”名”が、”プログラム一式(役割)”あるいは”プログラム一式 - 役割”という要約を持つ場合、要素はフレーズにすべきで、決まり文句に合うようになります:

パッケージ名は、プログラム一式に対する役割を表しています。

6.2.3 長い説明文 (long description)

長い説明文は、ユーザーがパッケージをインストールする前に利用可能な最初の情報です。ユーザーがインストールするか否かを決めるのに必要な情報を、すべて提供する必要があります。ユーザーがパッケージの概要を既に読んでいと仮定してください。

長い説明文は、完全な文章から成る必要があります。

長い説明文の最初の段落は、以下の質問に答えている必要があります: このパッケージは何をするの? ユーザーが作業を完了するのに、どのタスクが役に立つの? 技術寄りではない書き方でこれを記述するのが重要です。もちろんパッケージの利用者が必然的に技術者ではない限り、です。

続く段落は以下の質問に答える必要があります: 何故私はユーザーとしてこのパッケージを必要とするの? パッケージは他にどんな機能を持っているの? 他のパッケージと比べて、どんな特別な機能や不足している部分があるの? (例: Xが必要な場合、代わりにYを使いましょう) このパッケージはパッケージマネージャーで管理していない、何らかの方法で他のパッケージに関連している? (例: これは foo サーバのクライアントです)

スペルミスや文法の間違いを避けるよう、注意してください。スペルチェックを確実に行ってください。ispell と aspell の双方に、debian/control ファイルをチェックするための特別なモードがあります:

```
ispell -d american -g debian/control
```

```
aspell -d en -D -c debian/control
```

通常、ユーザは以下のような疑問がパッケージ説明文で答えられることを期待しています:

- パッケージは何をするの? 他のパッケージのアドオンだった場合、パッケージがアドオンであるということを概要文に書く必要があります。
- なぜこのパッケージを使うべきなの? これは上記に関連しますが、同じではありません (これはメールユーザーエージェントです; クールで速く、PGP や LDAP や IMAP のインターフェイスがあり、X や Y や Z の機能があります)。
- パッケージが直接インストールされるべきではないが、他のパッケージから引っ張ってこられる時には、付記しておく必要があります。
- パッケージが実験的である、あるいは使われない方が良い他の理由がある場合、同様にここに記載する必要があります。
- パッケージは競合のものと比べてどうでしょうか? より良い実装なのでしょうか? 機能がより豊富なのでしょうか? 違った機能があるのでしょうか? このパッケージを選ぶ理由は何でしょう。

6.2.4 開発元のホームページ

debian/control 中の Source セクションの Homepage フィールドへ、パッケージのホームページの URL を追加することをお勧めします。この情報をパッケージ説明文自身に追加するのは推奨されない (deprecated) であると考えられています。

6.2.5 バージョン管理システムの場合

debian/control には、バージョン管理システムの場合についての追加フィールドがあります。

6.2.5.1 Vcs-Browser

このフィールドの値は、指定したパッケージのメンテナンスに使われているバージョン管理システムのリポジトリのコピーがもしあれば、それを指し示す `http:// URL` である必要があります。

この情報は、パッケージに行われた最新の作業を閲覧したいエンドユーザにとって有用であるのが目的です (例: バグ追跡システムで pending とタグがつけられているバグを修正するパッチを探している場合)。

6.2.5.2 Vcs-*

このフィールドの値は、もし利用可能でなのであれば、指定されたパッケージをメンテナンスするのに使われているバージョン管理システムの位置を明確に識別できる文字列である必要があります。* はバージョン管理システムの識別に使われます; 現在では、以下のシステムがパッケージ追跡システムによってサポートされています: arch, bazaar (Bazaar), cvs, darcs, git, hg (Mercurial), mtn (Monotone), svn (Subversion)。同じパッケージについて異なった VCS を指定することも可能です: これらはすべて PTS のウェブインターフェイスに表示されます。

この情報は、そのバージョン管理システムについて知識があり、VCS ソースから現在のバージョンパッケージを生成ユーザにとって有益であるよう意図されています。この情報の他の使い方としては、指定されたパッケージの最新の VCS バージョンを自動ビルドするなどがあるかもしれません。このため、フィールドによって指し示される場所は、バージョンに関係なく、(そのようなコンセプトをサポートしている VCS であれば) メインブランチを指すのが良いでしょう。また、指し示される場所は一般ユーザがアクセス可能である必要があります; この要求を満たすには SSH アクセス可能なりポジトリを指すのではなく、匿名アクセスが可能なりポジトリを指すようにすることを意味します。

以下の例では、vim パッケージの Subversion リポジトリに対するフィールドの例が挙げられています。(svn+ssh:// ではなく) svn:// スキーム中で URL がどのようになっているか、trunk/ ブランチをどのように指し示しているかに注意してください。上で挙げられた Vcs-Browser フィールドと Homepage フィールドの使い方も出ています。

```
Source: vim
Section: editors
Priority: optional
<snip>
Vcs-Svn: svn://svn.debian.org/svn/pkg-vim/trunk/packages/vim
Vcs-Browser: https://svn.debian.org/wsvn/pkg-vim/trunk/packages/vim
Homepage: http://www.vim.org
```


6.3 debian/changelog のベストプラクティス

以下のプラクティスは **changelog ファイルに対するポリシー** を補完します。

6.3.1 役立つ changelog のエントリを書く

パッケージリビジョンに対する changelog エントリは、そのリビジョンでの変更それだけを記載します。最後のバージョンから行われた、重要な、そしてユーザに見える形の変更を記述することに集中しましょう。

何が変更されたかについて集中しましょう—誰が、どうやって、何時なのか通常あまり重要ではありません。そうは言っても、パッケージ作成について明記すべき手助けをしてくれた人々 (例えば、パッチを送ってくれた人) を丁寧に明記するのを忘れないようにしましょう。

些細で明白な変更を詳細に書く必要はありません。複数の変更点の一つのエントリにまとめることもできます。逆に言えば、大きな変更をした場合には、あまりに簡潔にしすぎないようにしましょう。プログラムの動作に影響を与える変更がある場合には、特に確認しておきましょう。詳細な説明については、README.Debian ファイルを使ってください。

平易な英語を使いましょう。そうすれば読者の大半が理解できます。バグをクローズする変更を説明する際には略語や、テクニカルな言い方やジャーゴンを避けましょう。特に、技術的に精通していないと思われるユーザによって登録されたバグを閉じる際には気をつけましょう。礼儀正しく、断言をしないように。

時折、changelog エントリに変更したファイルの名前を頭に付けたいことがあります。ですが、個々のすべての変更したファイルを一覧にする必要性はありません。特に変更点が小さくて繰り返される場合です。ワイルドカードを使いましょう。

バグに触れる際には、何も仮定しないようにしましょう。何が問題だったのか、どうやってそれが直されたのかについて言い、closes: #nnnnn の文字列を追加しましょう。詳細については項 5.8.4 を参照してください。

6.3.2 changelog のエントリに関するよくある誤解

changelog エントリは、一般的なパッケージ化の事柄 (ほら、foo.conf を探しているなら /etc/blah にあるよ) を記載するべきではありません。何故なら、管理者やユーザは少なくとも Debian システム上でそのようなことがどのように扱われるかを多少は知らされているでしょうから。ですが、設定ファイルの場所を変更したのであれば、それは一言添えておくべきです。

changelog エントリでクローズされるバグは、実際にそのパッケージリビジョンで修正されたものだけにすべきです。changelog で関係ないバグを閉じるのは良くない習慣です。項 5.8.4 を参照してください。

changelog エントリは、バグ報告者との各種の議論の場 (foo をオプション bar 付きで起動した際にはセグメンテーションフォルトは見られません; もっと詳しい情報を送ってください)、生命、宇宙、そして万物についての概要 (すいませんが、このアップロードに時間がかかったので風邪をひきました)、手助けの求め (このパッケージのバグ一覧は巨大です、手を貸してください) に使うべきではありません。そのようなことは、大抵の場合対象としている読者は気づくことが無く、パッケージで実際にあった変更点の情報について読みたい人々を悩ますことでしょう。どの様にバグ報告システムを使えばいいのかについて、詳細な情報は項 5.8.2 を参照してください。

正式なメンテナによるアップロードの changelog エントリの最初で、non-maintainer upload で修正されたバグを承認するのは、古い慣習です。今はバージョン管理を行っているので、NMU された changelog エントリを残しておいて自分の changelog エントリ中でその事実に触れるだけで十分です。

6.3.3 changelog のエントリ中のよくある間違い

以下の例で、changelog エントリ中のよくある間違いや間違ったスタイルの例を挙げます。

```
* Fixed all outstanding bugs.
```

これは、全く読み手に何も有用なことを教えてくれません。

```
* Applied patch from Jane Random.
```

何についてのパッチですか?

```
* Late night install target overhaul.
```

何をオーバーホールしてどうなったのですか? 深夜というのに言及しているのは、私たちにこのコードを信用すると言いたいのですか?

```
* Fix vsync FU w/ ancient CRTs.
```

略称が多すぎます。そして、ええっと、fsckup (ああ、ひどい言葉!) は実際何だったのか、あるいはどうやって修正したのかがまったく明らかではありません。

```
* This is not a bug, closes: #nnnnnn.
```

まず初めに、この情報を伝えるためにパッケージをアップロードする必要は、全くありません; 代わりにバグ追跡システムを使ってください。次に、何故この報告がバグではなかったのかについての説明がありません。

```
* Has been fixed for ages, but I forgot to close; closes: #54321.
```

何らかの理由で以前の changelog エントリ内でバグ番号について触れていなかったとしても、何も問題はありません。単にいつも通りに BTS でバグをクローズしてください。修正の記述が既にあるものと考えて、changelog ファイルに触れる必要はありません (これは、開発元の作者/メンテナによる修正にも同様に適用されます。彼らがずっと前に修正したバグを、あなたの changelog 内で追いかける必要はありません)。

```
* Closes: #12345, #12346, #15432
```

説明はどこ? 説明文を考えられないのなら、それぞれのバグのタイトルを入れるところから始めてください。

6.3.4 NEWS.Debian ファイルで changelog を補足する

パッケージの変更に関する重要なニュースは NEWS.Debian ファイルにも書くことができます。このニュースは apt-listchanges のようなツールで、残りすべての changelog の前に表示されます。これは、ユーザにパッケージ内の著しい変更点について知らせるのに好ましいやり方です。インストール後にユーザが一旦戻って NEWS.Debian ファイルを参照できるので、debconf の notes を使うより良いです。そして、目立った変更点を README.Debian に列挙するより良いです。何故ならば、ユーザは容易にそのような注意書きを見逃すからです。

ファイル形式は debian changelog ファイルと同じですが、アスタリスク (*) を取って各ニュースを changelog に書くような簡潔な要約ではなく、必要に応じて完全な段落で記述してください。changelog のようにビルド中に自動的にチェックされないの、ファイルを dpkg-parsechangelog を実行してチェックするのは良い考えです。実際の NEWS.Debian ファイルの例が、以下になります:

```
cron (3.0pl1-74) unstable; urgency=low
```

```
The checksecurity script is no longer included with the cron package:
it now has its own package, checksecurity. If you liked the
functionality provided with that script, please install the new
package.
```

```
-- Steve Greenland <stevegr@debian.org> Sat, 6 Sep 2003 17:15:03 -0500
```

NEWS.Debian ファイルは /usr/share/doc/package/NEWS.Debian.gz ファイルとしてインストールされます。圧縮されていて、Debian ネイティブパッケージ中でも常にこの名前です。debhelper を使う場合は、dh_installchangelogs が debian/NEWS ファイルをインストールしてくれます。

changelog ファイルと違って、毎回のリリースごとに NEWS.Debian ファイルを更新する必要はありません。何か特にユーザが知るべき目新しいことがあったときにのみ、更新してください。全くニュースがない場合、NEWS.Debian ファイルをパッケージに入れてリリースする必要はありません。便りが無いのは良い知らせ、です (No news is good news!).

6.4 メンテナスクリプトのベストプラクティス

メンテナスクリプトには debian/postinst、debian/preinst、debian/prerm、debian/postrm ファイルが含まれます。これらのスクリプトは、単なるファイルやディレクトリの作成や削除では扱われない、パッケージのインストールと削除のセットアップの面倒をみます。以下の説明は、**Debian ポリシー**を補完します。

メンテナスクリプトは冪等でなければなりません。これは、通常は 1 回呼ばれるスクリプトが 2 回呼ばれた場合、何も悪いことが起きないのを保証する必要があるという意味です。

標準入出力はログの取得のためにリダイレクトされることがあります (例: パイプへ向けられる)。ですので、これらが `tty` であることに依存してはいけません。

質問や対話的な設定はすべて最小限に止めておく必要があります。必要になった時は、インタースペースに `debconf` パッケージを使いましょう。どのような場合でも、質問は `postinst` スクリプトの `configure` 段階にのみ、配置することができます。

メンテナスクリプトは、できる限りシンプルなものにしておきましょう。我々は、あなたは純粋な POSIX シェルスクリプトを使っているものだと考えています。覚えておいて欲しいのですが、何かしら `bash` の機能が必要になったら、メンテナスクリプトは `bash` のシェバン行にしておく必要があります。スクリプトへ簡単にちょっとした変更を追加するのに `debhelper` を使えるので、Perl より POSIX シェル、あるいは Bash の方が好まれます。

メンテナスクリプトを変更したら、パッケージの削除や二重インストール、`purge` のテストを確認してください。`purge` されたパッケージが完全に削除されたことを確認してください。つまり、メンテナスクリプト中で直接／間接を問わず作成されたファイルを削除する必要があるということです。

コマンドの存在をチェックする必要がある場合は、このような感じで行います

```
if [ -x /usr/sbin/install-docs ]; then ...
```

メンテナスクリプト中でコマンドの `path` をハードコードしたくない場合には、以下の POSIX 互換シェル機能が役立つでしょう:

```
pathfind() {
    OLDFIFS="$IFS"
    IFS=:
    for p in $PATH; do
        if [ -x "$p/$*" ]; then
            IFS="$OLDIFS"
            return 0
        fi
    done
    IFS="$OLDIFS"
    return 1
}
```

コマンド名を引数として渡すことで、`$PATH` の検索するのにこの関数を使うことができます。コマンドが見つかった場合は `true` (ゼロ) を返し、そうでない場合は `false` を返します。`command -v`、`type`、`which` は POSIX に無いので、これがもっとも汎用性の高いやり方です。

`which` は、Required となっている `debianutils` パッケージにあるので、別解として利用可能ですが、ルートパーティションにありません。つまり、`/usr/bin` にあつて `/bin` ではないので、`/usr` パーティションがマウントする前に走るスクリプトの中では使えないということです。ほとんどのスクリプトは、この問題を持つようなことはありませんが。

6.5 debconf による設定管理

`debconf` は、(主に `postinst` をはじめとする) すべての様々なパッケージスクリプトが、ユーザはどの様にパッケージを設定したいのかというフィードバックを問い合わせるのに使うことが可能な設定管理システムです。直接ユーザとの対話処理は、`debconf` での処理の方を選んだので、現状では避けるべきです。これにより、将来には非対話的なインストールが可能になる予定です。

`debconf` は素晴らしいツールですが、しばしば適当に扱われています。多くの共通する失敗は、`debconf-devel(7)` `man` ページに記載されています。これは、`debconf` を使うのを決めた時、あなたが読むべきものです。また、ここではベストプラクティスを記述しています。

これらのガイドラインは、ディストリビューションの一部 (例えば、インストールシステム) に関する、より明確な推奨と同様に、幾つかの書き方と体裁に関する推奨、そして `debconf` の使い方についての一般的な考慮すべき事柄を含んでいます。

6.5.1 debconf を乱用しない

`debconf` が Debian に現れて以来、広く乱用され続けています。そして、`debconf` の乱用によって、ちょっとしたものをインストールする前に、大量の質問に答える必要があることに由来するいくつかの非難が Debian ディストリビューションに寄せられました。

使い方のメモは載せるべきところへ載せましょう: NEWS.Debian、または README.Debian ファイルです。notes はパッケージのユーザビリティに直接影響する重要な記述にだけ使いましょう。notes は確認する (あるいはメールでユーザを悩ます) まで、インストールを常にブロックすることを覚えておいてください。

メンテナスクリプト中の質問の優先度を注意深く選びましょう。優先度の詳細については、debconf-devel(7) を参照してください。ほとんどの質問は優先度が medium あるいは low を使うべきです。

6.5.2 作者と翻訳者に対する雑多な推奨

6.5.2.1 正しい英語を書く

ほとんどの Debian パッケージメンテナはネイティブの英語話者ではありません。ですので、正しいフレーズのテンプレートを書くのは彼らにとっては容易ではありません。

debian-i10n-english@lists.debian.org メーリングリストを利用してください (むしろ乱用してください)。テンプレートを査読してもらいましょう。

下手に書かれたテンプレートは、パッケージに対して、そしてあなたの作業に対して、さらには Debian それ自体にすら対して、悪い印象を与えます。

可能な限り技術的なジャーゴンを避けましょう。いくつかの用語があなたにとっては普通に聞こえても、他の人には理解不可能かもしれません。避けられない場合には、(説明文を使って) 解説しましょう。その場合には、冗長さとシンプルさのバランスを取るようにしましょう。

6.5.2.2 翻訳者へ丁寧に接する

debconf テンプレートは翻訳されるでしょう。debconf、そして関連する姉妹パッケージ **po-debconf** は、テンプレートを翻訳チームやさらには個人が翻訳できるようにする、シンプルなフレームワークを提供します。

gettext ベースのテンプレートを使ってください。開発用のシステムに po-debconf をインストールしてドキュメントを読みましょう (**man po-debconf** が取っ掛かりとして良いでしょう)。

テンプレートを頻繁に変更しすぎることを避けましょう。テンプレート文の変更は、翻訳を fuzzy にして、さらなる作業を翻訳作業者に強います。fuzzy になっている翻訳文は、翻訳されてから元の文章が変更された文字列であり、使えるようにするには翻訳作業者による若干の更新が必要なものです。変更点が十分に小さい場合、fuzzy とマークされますが、元の翻訳文は PO ファイル中に保持されません。

大本のテンプレートを変更する予定がある場合、po-debconf パッケージで提供されている、**podebconf-report-po** という名の通知システムを使って翻訳作業者にコンタクトを取ってください。ほとんどのアクティブな翻訳作業者たちはとても反応が良く、変更を加えたテンプレートに対応するための作業をしてくれ、あなたが追加でアップロードする必要を減らしてくれます。gettext ベースのテンプレートを使っている場合、翻訳作業者の名前とメールアドレスは PO ファイルのヘッダに表示されており、**podebconf-report-po** によって使われます。

このユーティリティの使い方のお勧めの使い方:

```
cd debian/po && podebconf-report-po --call --languagesteam --withtranslators -- ←
deadline="+10 days"
```

このコマンドは、まず debian/po にある PO ファイルと POT ファイルを debian/po/POTFILES.in に記載しているテンプレートファイルを使って同期します。そして、debian-i18n@lists.debian.org メーリングリストに call for new translations (新しい翻訳作業の募集) を送信します。最後に、call for translation updates (翻訳更新作業者の募集) を (各 PO ファイルの Language-Team 欄に記載されている) 各言語チームおよび (Last-translator に記載されている) 最後の翻訳者に送信します。

翻訳作業者に締切りを伝えるのは常にお勧めです。それによって、彼らは作業を調整できます。いくつかの翻訳作業チームは形式化された翻訳/レビュープロセスを整えており、10 日未満の猶予は不合理であると考えられています。より短い猶予期間は強すぎるプレッシャーを翻訳チームに与えるので、非常に些細な変更点に対してのみに留めるべきです。

迷った場合は、該当の言語の翻訳チーム (debian-i10n-xxxxx@lists.debian.org) か debian-i18n@lists.debian.org にも問い合わせましょう。

6.5.2.3 誤字やミススペルを修正する際に fuzzy を取る

debconf テンプレートの文章が修正されて、その変更が翻訳に影響しないと確信している場合には、翻訳作業者を労って翻訳文を fuzzy を取り除いてください。

そうしないと、翻訳作業者が更新をあなたに送るまでテンプレート全体は翻訳されていない状態になります。

翻訳を *fuzzy* ではなくすために、(po4a パッケージの一部である) **msguntypot** を使うことができます。

1. POT ファイルと PO ファイルを再生成する。

```
debconf-updatepo
```

2. POT ファイルのコピーを作成する。

```
cp templates.pot templates.pot.orig
```

3. すべての PO ファイルのコピーを作成する。

```
mkdir po_fridge; cp *.po po_fridge
```

4. debconf テンプレートファイルを誤字修正のために変更する。

5. POT ファイルと PO ファイルを再生成する (もう一度)。

```
debconf-updatepo
```

この時点では、*typo* 修正はすべての翻訳を *fuzzy* にしており、この残念な変更はメインディレクトリの PO ファイルと *fridge* の PO ファイルのみに適用されている。ここではどの様にしてこれを解決するかを示す。

6. *fuzzy* になった翻訳を捨て、*fridge* から作り直す。

```
cp po_fridge/*.po .
```

7. 手動で PO ファイルと新しい POT ファイルをマージするが、不要な *fuzzy* を考慮に入れる。

```
msguntypot -o templates.pot.orig -n templates.pot *.po
```

8. ゴミ掃除。

```
rm -rf templates.pot.orig po_fridge
```

6.5.2.4 インターフェイスについて仮定をしない

テンプレートのテキストは、*debconf* のインターフェイスに属するウィジェットには言及してはいけません。「はい」と答えた場合には... のような文章は、2 択の質問に対してチェックボックスを使うようなグラフィカルインターフェイスのユーザには何の意味もありません。

文字列テンプレートは、説明文中でのデフォルト値について言及することも避けましょう。まず、ユーザによってそして、デフォルト値はメンテナの考え方によって違う場合があるからです (たとえば、*debconf* データベースが *preseed* で指定されている場合など)。

より一般的に言うと、ユーザの行動を参照させるのを避けるようにしましょう。単に事実を与えましょう。

6.5.2.5 一人称を使わない

(*I will do this...* や *We recommend...* などの) 一人称の利用を避けましょう。コンピュータは人ではなく、*debconf* テンプレートは Debian 開発者を代弁するものではありません。中立的な解釈を行いましょう。あなたが科学論文を書いたことがあるならば、論文を書くのと同様にテンプレートを書きましょう。ですが、可能であれば *This can be enabled if...* ではなく *Enable this if...* などとして生の声を届けるようにしましょう。

6.5.2.6 性差に対して中立であれ

世界は、男と女によって成り立っています。記述の際には、性差に対して中立な書き方を行ってください。

6.5.3 テンプレートのフィールド定義

この章の情報は、ほとんどを `debconf-devel(7)` マニュアルページから得ています。

6.5.3.1 Type

6.5.3.1.1 string ユーザがどのような文字列でも記述可能な自由記述形式の入力フィールドの結果。

6.5.3.1.2 password ユーザにパスワードの入力を求めます。これを使う場合は注意して使ってください: ユーザが入力したパスワードは `debconf` のデータベースに書き込まれることに注意してください。おそらく、この値をデータベースから可能な限り早く消す必要があります。

6.5.3.1.3 boolean `true/false` の選択です。注意点: `true/false` であって、`yes/no` ではありません...

6.5.3.1.4 select 複数の値から一つを選びます。選択するものは `'Choices'` というフィールド名で指定されている必要があります。利用可能な値をコンマとスペースで区切ってください。以下のようにあります: `Choices:yes, no, maybe`

選択肢が翻訳可能な文字列である場合、`'Choices'` フィールドは `__Choices` を使って翻訳可能であることを示しておくとい良いでしょう。2つのアンダースコアは、各選択肢を分かれた文字列に分割してくれます。

po-debconf システムは、翻訳可能ないくつかの選択肢のみをマークする面白い機能を提供してくれます。例:

```
Template: foo/bar
Type: Select
#flag:translate:3
__Choices: PAL, SECAM, Other
__Description: TV standard:
Please choose the TV standard used in your country.
```

この例では、他は頭文字から構成されていて翻訳できませんが、`'Other'` 文字列だけは翻訳可能です。上記では `'Other'` だけが PO および POT ファイルに含めることができます。

`debconf` テンプレートのフラグシステムは、このような機能をたくさん提供します。`po-debconf(7)` マニュアルページでは、これらの利用可能な機能をすべて列挙しています。

6.5.3.1.5 multiselect `select` データ型とそっくりですが、ユーザが選択肢一覧からいくつでも項目を選ぶ (あるいはどれも選ばないこともできる) 点だけが違います。

6.5.3.1.6 note 本来質問ではありませんが、このデータ型が示すのはユーザに表示することができる覚え書きです。`debconf` はユーザが必ず参照するようにするため、多大な苦痛を与えることになる (キーを押すためにインストールを休止したり、ある場合にはメモをメールさえする) ので、ユーザが知っておく必要がある重要な記述にのみ使うべきです。

6.5.3.1.7 text この type は現状では古すぎるものと考えられています: 使わないでください。

6.5.3.1.8 error この type はエラーメッセージを取り扱うためにデザインされています。ほとんど `note` 型と似通っています。違いはフロントエンドが存在しているであろうことです (例えば、`cdebconf` の `dialog` フロントエンドは通常の青い画面ではなく、赤い画面を描画します)。

何かを補正するためにユーザの注意を引く必要があるメッセージに対し、この type を使うのがお勧めです。

6.5.3.2 Description: short および extended 説明文

テンプレート説明文は2つのパートに分かれます: `short` と `extended` です。短い説明文 (`short description`) はテンプレートの `Description:` 行にあります。

短い説明文は、ほとんどの `debconf` インターフェイスに適用するように、短く (50 文字程度に) しておく必要があります。通常、翻訳はオリジナルよりも長くなる傾向にあるので、短くすることは翻訳作業者を助けます。

短い説明文はそれ単体で成り立つようにしておく必要があります。いくつかのインターフェイスは、デフォルトでは長い説明文を表示せず、ユーザが明示的に尋ねたときに表示するか、あるいは全く表示しません。「What do you want to do?」のような表現を避けてください。

短い説明文は完全な文章である必要はありません。これは文章を短くしておき、効率的に推奨を行うためです。

拡張された説明文 (extended description) は、短い説明文を一語一句繰り返しをしてはなりません。長い説明文章を思いつかなければ、まず、もっと考えてください。debian-devel に投稿しましょう。助けを求めましょう。文章の書き方講座を受講しましょう! この拡張された説明文は重要です。もし、まったく何も思いつかなければ、空のままにしておきましょう。

拡張された説明文は完全な文章である必要があります。段落を短くしておくのは可読性を高めます。同じ段落で2つの考えを混ぜてはいけません。代わりに別の段落を書きます。

あまり冗長にしないようにしてください。ユーザは長すぎる画面を無視しようとしします。経験からすると、20行が越えてはならない境界です。何故ならば、クラシックなダイアログインターフェイスでは、スクロールする必要がでてくることになり、そして多くの人がスクロールなどしないからです。

拡張された説明文では、質問を含めては決していけません。

テンプレートの type (string、boolean など) に応じた特別なルールについては、以下を読んでください。

6.5.3.3 Choices

このフィールドは select あるいは multiselect 型に使ってください。これには、ユーザに表示される可能な選択肢が含まれます。これらの選択肢はコンマで区切る必要があります。

6.5.3.4 Default

このフィールドはオプションです。これには、string、select あるいは multiselect のデフォルトでの答えが含まれます。multiselect テンプレートの場合、コンマで区切られた選択肢一覧が含まれます。

6.5.4 テンプレートのフィールド固有スタイルガイド

6.5.4.1 Type フィールド

特別な指定はありません。一点だけ、その前のセクションを参照して適切な type を使ってください。

6.5.4.2 Description フィールド

以下は、テンプレートの型に応じて適切な Description (short および extended) を書くための特別な指示です。

6.5.4.2.1 String/password テンプレート

- 短い説明文は、プロンプトであってタイトルではありません。質問形式のプロンプト (IP アドレスは?) を避け、代わりに閉じていない形のプロンプト (IP アドレス:) を使ってください。コロン(:) の使用をお勧めします。
- 拡張された説明文は、短い説明文を補足します。拡張の部分では、長い文章を使って同じ質問を繰り返すのではなく、何を質問されているのかを説明します。完全な文章を書いてください。簡潔な書き方は強く忌避されます。

6.5.4.2.2 Boolean テンプレート

- 短い説明文は、短いままで大抵の場合は? で終わる質問の体裁の言い回しをせねばなりません。簡潔な書き方は許されており、質問が若干長い場合は推奨されずしています (翻訳文は、大抵の場合原文よりも長くなるのを覚えておきましょう)。
- 繰り返しますが、特定のインターフェイスのウィジェットを参照するのを避けてください。このようなテンプレートで良くある間違いは、「はい」と答える形かどうかです。

6.5.4.2.3 Select/Multiselect

- 短い説明文は、プロンプトであってタイトルではありません。「Please choose...」のような意味の無い文章を文章を使わないでください。ユーザは何かを選ぶ必要があるくらいには十分賢いです... :)
- 拡張された説明文は、短い説明文を完備します。これでは、利用可能な選択肢に言及することがあります。テンプレートが `multiselect` なものの場合、ユーザが選べる選択肢が1つではないことについても言及するかもしれません (インターフェイスが大抵これを明確にはしてくれますが)。

6.5.4.2.4 Note

- 短い説明文はタイトルとして扱われます。
- 拡張された説明文では、`note` のより詳細な説明を表示します。フレーズで、簡潔過ぎない書き方です。
- `debconf` を乱用しないでください。`note` は、`debconf` の乱用で最も良く使われます。`debconf-devel` マニュアルページに書かれています: とても深刻な問題について警告する場合のみに使うのがベストです。多くの覚え書きについては、`NEWS.Debian` ファイルや `README.Debian` ファイルが適切な場所です。もし、これを読んで、`Note` 型のテンプレートを `NEWS.Debian` あるいは `README.Debian` 中のエントリに変換することを考えた場合、既存の翻訳を捨てないことを検討してください。

6.5.4.3 Choices フィールド

もし `Choice` が頻繁に変わるようであれば、`_Choices` という使い方をするのを検討してください。これは個々の選択項目を単一の文字列に分割するので、翻訳作業者が作業を行うのを助けてくれると考えられています。

6.5.4.4 Default フィールド

`select` のテンプレートで、デフォルト値がユーザの言語に応じて変化するようであれば、`_Default` という使い方をしてください (例えば、選択肢が言語の選択の場合等)。

この特別なフィールドによって、翻訳作業者は言語に応じたもっとも適切な選択肢を挿入することができますようになります。彼らの言語が選択された場合にデフォルトの選択になりますが、英語を使うときには最初に提示された `Default Choice` が使われます。

`genweb` パッケージのテンプレートを例にとってみましょう:

```
Template: genweb/lang
Type: select
__Choices: Afrikaans (af), Bulgarian (bg), Catalan (ca), Chinese (zh), Czech (cs) ←
, Danish (da), Dutch (nl), English (en), Esperanto (eo), Estonian (et), ←
Finnish (fi), French (fr), German (de), Hebrew (he), Icelandic (is), Italian ←
(it), Japanese (ja), Latvian (lv), Norwegian (no), Polish (pl), Portuguese ( ←
pt), Romanian (ro), Russian (ru), Spanish (es), Swedish (sv)
# This is the default choice. Translators may put their own language here
# instead of the default.
# WARNING : you MUST use the ENGLISH NAME of your language
# For instance, the french translator will need to put French (fr) here.
_Default: English[ translators, please see comment in PO files]
_Description: Genweb default language:
```

ブラケットを使うと、`debconf` のフィールド中に内部コメントができることに注意してください。コメントの利用は、翻訳作業者が作業するファイルに表示されるのにも注意ください。

`_Default` を使うやり方は、若干混乱するのでコメントが必要です: 翻訳者は自身の選択肢を書きます。

6.5.4.5 Default フィールド

空のデフォルトフィールドを使つては「いけません」。デフォルト値を設定したくないのであれば、`Default` は使うべきではありません。

po-debconf を使うのであれば(そして 使うべきです、項6.5.2.2 参照)、このフィールドが翻訳できるのであれば、翻訳可能な状態にするのを検討しましょう。

(例えば言語選択のデフォルト値など) デフォルト値が言語/地域に応じて変化するようであれば、po-debconf(7) に記載されている特別な `_Default` 型を使うことを考えましょう。

6.6 国際化

この章では、開発者に対して、翻訳作業者の作業がより楽になるようにするための大まかな情報を含めています。国際化について興味を持っている翻訳作業者と開発者への詳細な情報は、[Internationalisation and localisation in Debian](#) で入手できます。

6.6.1 debconf の翻訳を取り扱う

移植作業者同様に、翻訳作業者は難しい課題を抱えています。多くのパッケージについて作業を行い、多くの異なったメンテナと共同作業をする必要があります。さらには、ほとんどの場合、彼らはネイティブな英語話者ではないので、あなたは特に忍耐強くあらねばいけません。

debconf のゴールはメンテナとユーザにとってパッケージ設定を簡単にすることでした。元々、debconf テンプレートの翻訳は `debconf-mergetemplate` で行われていました。しかし、このやり方は現在是非推奨 (deprecated) です; debconf の国際化を成し遂げる最も良いやり方は、po-debconf パッケージを使うことです。このやり方はメンテナと翻訳者の双方にとって、より楽なやり方です; 移行スクリプトが提供されています。

po-debconf を使うと、翻訳は .po ファイルに収められます (`gettext` による翻訳技術からの引き出しです)。特別なテンプレートファイルには、元の文章と、どのフィールドが翻訳可能かがマークされています。翻訳可能なフィールドの値を変更すると、`debconf-updatepo` を呼び出すことで、翻訳作業者の注意が必要のように翻訳にマークがされます。そして、生成時には `dh_installdebconf` プログラムが、テンプレートに加え、最新の翻訳をバイナリパッケージに追加するのに必要となる魔法について、すべての面倒を見ます。詳細は po-debconf(7) マニュアルページを参照してください。

6.6.2 ドキュメントの国際化

ドキュメントの国際化はユーザにとって極めて重要ですが、多くの労力がかかります。この作業をすべて除去する方法はありませんが、翻訳作業者を気楽にはできます。

どのようなサイズであれドキュメントをメンテナンスしている場合、翻訳作業者がソースコントロールシステムにアクセスできるのであれば、彼らの作業が楽になるでしょう。翻訳作業者が、ドキュメントの2つのバージョン間の違いを見ることができるので、例えば、何を再翻訳すればいいのかがわかるようになります。翻訳されたドキュメントは、翻訳作業がどのソースコントロールリビジョンをベースにしているのかという記録を保持しておくことをお勧めします。debian-installer パッケージ中の `doc-check` では興味深いシステムが提供されています。これは、翻訳すべき現在のリビジョンのファイルに対する構造化されているコメントを使って、指定されたあらゆる言語の翻訳状況の概要を表示し、翻訳されたファイルについては、翻訳がベースにしているオリジナルのファイルのリビジョンを表示します。自分の VCS 領域でこれを導入して利用した方が良いでしょう。

XML あるいは SGML のドキュメントをメンテナンスしているのであれば、言語非依存の情報を分離し、それらをすべての異なった翻訳に含まれる分割されたファイルにエンティティとして定義した方が良いでしょう。これは、URL を複数のファイル間で最新に保つなど、作業をより楽にしてくれます。

いくつかのツール (例: po4a, poxml, translate-toolkit) は異なった形式から翻訳可能な素材を展開するのに特化しています。これらは、翻訳作業者には極めて馴染み深い形式である PO ファイルを生成します。これによって、翻訳したドキュメントが更新された際に何を再翻訳すればよいのかを見ることを可能にしてくれます。

6.7 パッケージ化に於ける一般的なシチュエーション

6.7.1 autoconf/automake を使っているパッケージ

`autoconf` の `config.sub` および `config.guess` を最新に保ちつづけるのは、移植作業者、特により移行中の状況であるアーキテクチャの移植作業者にとって非常に重要です。`autoconf` や `automake` を使うあらゆるパッケージについてのとても素晴らしいパッケージ化における教訓が `autotools-dev` パ

ツケージの `/usr/share/doc/autotools-dev/README.Debian.gz` にまとめられています。このファイルを読んで書かれている推奨に従うことを強くお勧めします。

6.7.2 ライブラリ

ライブラリは様々な理由から常にパッケージにするのが難しいです。ポリシーは、メンテナンスに樂にし、新しいバージョンが開発元から出た際にアップグレードを可能な限りシンプルであることを確保するため、多くの制約を課しています。ライブラリでの破損は、依存している何十ものパッケージの破損を招き得ます。

ライブラリのパッケージ化の良い作法が [the library packaging guide](#) にまとめられています。

6.7.3 ドキュメント化

ドキュメント化のポリシーに忘れず従ってください。

あなたのパッケージが XML や SGML から生成されるドキュメントを含んでいる場合、XML や SGML のソースをバイナリパッケージに含めてリリースしないことをお勧めします。ユーザがドキュメントのソースを欲しい場合には、ソースパッケージを引っ張ってくれば良いのです。

ポリシーではドキュメントは HTML 形式でリリースされるべきであると定めています。我々は、もし手がかからないで問題ない品質の出力が可能であれば、ドキュメントを PDF 形式とテキスト形式でもリリースすることをお勧めしています。ですが、ソースの形式が HTML のドキュメントを普通のテキスト版でリリースするのは、大抵の場合は適切ではありません。

リリースされたメジャーなマニュアルは、インストール時に `doc-base` で登録されるべきです。詳細については、`doc-base` パッケージのドキュメントを参照してください。

Debian ポリシー (12.1 章) では、マニュアルページはすべてのプログラム・ユーティリティ・関数に対して付属すべきであり、設定ファイルのようなその他のものについては付属を提案を示しています。もし、あなたがパッケージングをしているものがそのようなマニュアルページを持っていない場合は、パッケージに含めるために記述を行い、開発元 (upstream) へ送付することを検討してください。

`manpage` は直接 `troff` 形式で書く必要はありません。よく使われるソース形式は、Docbook、POD、reST で、それぞれ `xsltproc`、`pod2man`、`rst2man` を使うことで変換できます。少なくとも、スタブを書くために `help2man` プログラムを使うことは可能です。

6.7.4 特定の種類のパッケージ

いくつかの特定の種類のパッケージは、特別なサブポリシーと対応するパッケージ化ルールおよびプラクティスを持っています:

- Perl 関連パッケージは、**Perl ポリシー** があり、このポリシーに従っているパッケージの例が `libdbd-pg-perl` (バイナリ perl モジュール) または `libmldbm-perl` (アーキテクチャ非依存 perl モジュール) です。
- Python 関連パッケージは、**python ポリシー** を持っています; python パッケージ中の `/usr/share/doc/python/python-policy.txt.gz` を参照してください。
- Emacs 関連パッケージには、**emacs ポリシー** があります。
- Java 関連パッケージには、**java ポリシー** があります。
- Ocaml 関連パッケージは、固有のポリシーを持っており、ocaml パッケージの `/usr/share/doc/ocaml/ocaml_packaging_policy.gz` でみることができます。良い例は `camlzip` ソースパッケージです。
- XML や SGML DTD を提供しているパッケージは、`sgml-base-doc` パッケージ中の推奨に従わねばなりません。
- lisp パッケージは、パッケージ自身を `common-lisp-controller` に登録する必要があります。これについては、`/usr/share/doc/common-lisp-controller/README.packaging` を参照してください。

6.7.5 アーキテクチャ非依存のデータ

大量のアーキテクチャ非依存データがプログラムと共にパッケージ化されるのは、良くあることではありません。例えば、音声ファイル、アイコン集、様々な模様の壁紙、その他一般的な画像ファイルです。このデータのサイズがパッケージの残りのサイズと比較して取るに足らないのであれば、おそらくは単一パッケージでひとまとめにしておくのがベストでしょう。

しかし、データのサイズが考えた方が良い程であれば、分かれたアーキテクチャ非依存パッケージ (`_all.deb`) に分割するのを考えてください。これによって、11 あるいはそれ以上の `.deb` ファイルについて、1 アーキテクチャごとに同じデータの不必要な重複を避けられます。これは `Packages` ファイルに更なるオーバーヘッドを追加しますが、`Debian` ミラーサーバ上で多くのディスク容量を節約します。アーキテクチャ非依存のデータを分割することは、`Debian` アーカイブ全体に対して実行される `lintian` の処理時間の削減にもつながります (項 A.2 参照)。

6.7.6 ビルド中に特定のロケールが必要

ビルド中に特定のロケールを必要とする場合、こんな技を使えば一時ファイルを作成できます:

`LOCPATH` を `/usr/lib/locale` と同等のものに、そして `LC_ALL` を生成したロケールの名前に設定すれば、`root` にならなくても欲しいものが手に入ります。こんな感じです:

```
LOCALE_PATH=debian/tmpdir/usr/lib/locale
LOCALE_NAME=en_IN
LOCALE_CHARSET=UTF-8

mkdir -p $LOCALE_PATH
localedef -i $LOCALE_NAME.$LOCALE_CHARSET -f $LOCALE_CHARSET $LOCALE_PATH/ ←
    $LOCALE_NAME.$LOCALE_CHARSET

# ロケールを使う
LOCPATH=$LOCALE_PATH LC_ALL=$LOCALE_NAME.$LOCALE_CHARSET date
```

6.7.7 移行パッケージを `debophran` に適合するようにする

`debophran` は、どのパッケージがシステムから安全に削除できるか、ユーザが検出するのを助けてくれるプログラムです; すなわち、どのパッケージも依存していないものです。デフォルトの動作は、使われていないライブラリを見つけ出すために `libs` と `oldlibs` セクションからのみ検索を行います。ですが、正しい引数を渡せば、他の使われていないパッケージを捕らえようとします。

例えば、`--guess-dummy` つきだと、`debophran` はアップグレードに必要ではあったが、現在は問題なく削除できるすべての移行パッケージを探そうとします。これには、それぞれの短い説明文の中に `dummy` あるいは `transitional` の文字列を探します。

ですので、あなたがそのようなパッケージを作る際には、この文章を短い説明文に必ず追加してください。例を探す場合は、以下を実行してください: `apt-cache search .lgrep dummy` または `apt-cache search .lgrep transitional`

それから、`debophran` の作業を楽にするため、`section` を `oldlibs`、`priority` を `extra` にするのもお勧めです。

6.7.8 `.orig.tar.{gz,bz2,xz}` についてのベストプラクティス

オリジナルのソース tarball には 2 種類あります: 手が入れられていない素のソース (Pristine source) と再パッケージした開発元のソース (repackaged upstream source) です。

6.7.8.1 手が入れられていないソース (Pristine source)

素のソース tarball (pristine source tarball) の特徴の定義は、`.orig.tar.{gz,bz2,xz}` は、開発元の作者によって公式に配布された tarball と 1 バイトたりとも変わらない、というものです。¹ これは、`Debian` diff 内に含まれている `Debian` バージョンと開発元のバージョン間のすべての違いを簡単に比較するの

¹ 我々は、配布されている tarball がバージョン番号をインクリメントすることなく、開発元の作者によって変更されるのを防ぐことはできません。ですので、手が入れられていない素の tarball が、どの時点でも開発元が現在配布していると等しいものである保証はありません。期待できることは、一度は配布されたものと等しいものであるということだけです。後から違いが発生した場合 (そう、例えば upstream が元々の配布物が最大の圧縮率を使っていなかったことに気づいて、再度 `gzip` したおした場合など)、それはとても残念なことです。同じバージョンに対して新しい `.orig.tar.{gz,bz2,xz}` をアップロードするのに良い方法がないので、この状況をバグと考える意味はありません。

にチェックサムを使えるようになります。また、オリジナルのソースが巨大な場合、既に upstream の tarball を持っている upstream の作者と他の者は、あなたのパッケージを詳細に調査したい場合、ダウンロード時間を節約できます。

tarball 中のディレクトリ構成に関して開発元の作者が従う、万人に受け入れられているガイドラインというものはありませんが、それにも関わらず **dpkg-source** はほとんどの upstream の tarball を素のソース (pristine source) として対処できます。そのやり方は以下の通りです:

1. 以下のようにして空の一時ディレクトリに tarball を展開します

```
zcat path/to/パッケージ名_開発元のバージョン.orig.tar.gz | tar xf -
```

2. もし、この後で、一時ディレクトリが1つのディレクトリ以外含まず他にどのファイルも無い場合、**dpkg-source** はそのディレクトリを パッケージ名-開発元のバージョン (.orig) にリネームします。tarball 中の最上位のディレクトリ名は問題にはされず、忘れられます。
3. それ以外の場合、upstream の tarball は共通の最上位ディレクトリ無しでパッケージされなくては いけません (upstream の作者よ、恥を知りなさい!)。この場合、**dpkg-source** は一時ディレクトリ そのものを パッケージ名-開発元のバージョン (.orig) へリネームします。

6.7.8.2 upstream のソースをパッケージしなおす

パッケージは手が入っていない素のソース tarball と共にアップロードすべきですが、それが可能ではない場合が色々あります。upstream がソースを gzip 圧縮した tarball を全く配布していない場合や、upstream の tarball が DFSG-free ではない、あなたがアップロード前に削除しなければならない素材を含んでいる場合がこれにあたります。

このような場合、開発者は適切な .orig.tar.{gz,bz2,xz} ファイルを自身で準備する必要があります。このような tarball を、再パッケージした開発元のソース (repackaged upstream source) と呼びます。再パッケージした開発元のソースでは Debian ネイティブパッケージとは違うことに注意してください。再パッケージしたソースは、Debian 固有の変更点は分割された .diff.gz または .debian.tar.{gz,bz2,xz} のままであり、バージョン番号は 開発元のバージョンと debian リビジョンから構成されたままです。

開発元が、原則的にはそのままの形で使える .tar.{gz,bz2,xz} を配布していたとしても再パッケージをしたくなるという場合があります。最も明解なのは、tar アーカイブを再圧縮することや upstream のアーカイブから純粋に使われていないゴミを削除することで、非常に容量を節約できる時です。ここで慎重になって頂きたいのですが、ソースを再パッケージする場合は、決定を貫く用意をしてください。

.orig.tar.{gz,bz2,xz} についてのベストプラクティス

1. ソースパッケージの由来をドキュメントにすべきです。どうやってソースを得たのかという詳細情報が得たのか、どの様にすれば再生成できるのかを debian/copyright で提供しましょう。ポリシーマニュアルで、**メイン構築スクリプト: debian/rules** に記述しているように、debian/rules に作業を繰り返してくれる get-orig-source ターゲットを追加するのも良い考えです。
2. 開発元の作者由来ではないファイルや、あなたが内容を変更したファイルを含めるべきではありません。²
3. 法的理由から不可能であるものを除いて、開発元の作者が提供しているビルドおよび移植作業環境を完全に保全すべきです。例えば、ファイルを省略する理由として MS-DOS でのビルドにしか使われないから、というのは十分な理由にはなりません。同様に、開発元から提供されている Makefile を省略すべきではありません。たとえ debian/rules が最初にすることが configure スクリプトを実行してそれを上書きすることであったとしても、です。

(理由: Debian 以外のプラットフォームのためにソフトウェアをビルドする必要がある Debian ユーザが、開発元の一次配布先を探そうとせずに Debian ミラーからソースを取得する、というのは普通です)。

² 特別な例外として、non-free なファイルが Debian diff からの助けが無いとソースからのビルド失敗を引き起こす場合、適切な処置はファイルを編集するのではなく、non-free な部分を削除して、ソースツリーのルートに README.source ファイルとして状況を説明することです。ですがその場合、開発元の作者に non-free なコンポーネントを残りのソースから分離しやすくするように促してください。

4. tarball の最上位ディレクトリの名前として パッケージ名-開発元のバージョン.orig を使うべきです。これは、大元の使用されていない tarball と再パッケージしたものを判別できるようにしてくれます。
5. gzip あるいは bzip で最大限圧縮されるべきです。

6.7.8.3 バイナリファイルの変更

オリジナルの tarball に含まれているバイナリファイルを変更する、あるいは存在していなかったバイナリファイルを追加する必要がある場合がしばしばあります。これは、ソースパッケージが “3.0 (quilt)” 形式を使っている場合は完全にサポートされています。詳細は `dpkg-source(1)` マニュアルのページを参照してください。古い “1.0” 形式を使っている場合、バイナリファイルを `.diff.gz` 中に保存できないので、**uuencode** (か類似のもの) を使ったファイルを保存し、`debian/rules` 中でビルド時にデコードします (そしてファイルを正しい位置へ移動する)。

6.7.9 デバッグパッケージのベストプラクティス

デバッグパッケージは名前が `-dbg` で終わっているもので、**gdb** が利用可能な追加情報を含んでいます。Debian のバイナリはデフォルトで `strip` されているので、関数名や行番号を含むデバッグ情報は、Debian のバイナリに **gdb** を走らせたときに利用できません。デバッグパッケージは、この追加デバッグ情報を必要とするユーザが、通常のシステムを巨大化させることなく使えるようにしてくれます。

デバッグパッケージを作るか否かは、パッケージのメンテナ次第です。メンテナは、ライブラリパッケージについてデバッグパッケージを作成するのが推奨されています。これは、ライブラリにリンクしている沢山のプログラムをデバッグする助けになるからです。一般的に言って、デバッグパッケージはすべてのプログラムに追加する必要はありません; これを行うとアーカイブが巨大なものになるでしょう。しかし、メンテナは、ユーザがデバッグ版のプログラムを頻繁に必要とするのに気づいた場合、デバッグパッケージを作るのに値するでしょう。インフラストラクチャの核となっている、`apache` や `X` サーバのようなプログラムも、デバッグパッケージの良い作成候補です。

デバッグパッケージのうちいくつかは、ライブラリあるいは他のバイナリの完全に特別なデバッグビルドを含むでしょうが、それらの大半は容量やビルドする時間を節約できます。`/usr/lib/debug/path` の場合、`path` は実行ファイルかライブラリへのパスになります。例えば、`/usr/bin/foo` のデバッグシンボルは `/usr/lib/debug/usr/bin/foo` に行き、`/usr/lib/libfoo.so.1` のデバッグシンボルは `/usr/lib/debug/usr/lib/libfoo.so.1` になります。

デバッグシンボルは **objcopy --only-keep-debug** を使ってオブジェクトファイルから展開できます。そうすればオブジェクトファイルを `strip` することができ、**objcopy --add-gnu-debuglink** がデバッグシンボルファイルへのパスを指定するのに使われます。`objcopy(1)` で、これがどの様に動作するのかを詳細に説明しています。

`debhelper` 中の **dh_strip** コマンドは、デバッグパッケージの作成をサポートし、デバッグシンボルを分離するのに **objcopy** の利用の面倒を見てくれます。あなたのパッケージが `debhelper` を使っている場合、あなたがする必要のあるのは **dh_strip --dbg-package=libfoo-dbg** を呼び出し、`debian/control` にデバッグパッケージのエントリを追加することだけです。

デバッグパッケージは、そのパッケージがデバッグシンボルを提供するパッケージに依存する必要があります、この依存関係はバージョン指定が必要であるということに注意してください。以下のような例になります:

```
Depends: libfoo (= ${binary:Version})
```

6.7.10 メタパッケージのベストプラクティス

メタパッケージは、時間がかかる一貫したセットのパッケージをインストールするのを楽にしてくれる、ほぼ空のパッケージです。そのセットの全パッケージに依存することで、これを実現しています。APT の力のおかげで、メタパッケージのメンテナは依存関係を調整すればユーザのシステムが自動的に追加パッケージを得ることができます。自動的にインストールされていてメタパッケージから落とされたパッケージは、削除候補としてマークされます (そして **aptitude** によって自動的に削除もされます)。`gnome` と `linux-image-amd64` は、メタパッケージの 2 つの例です (ソースパッケージ `meta-gnome2` and `linux-latest` から生成されています)。

メタパッケージの長い説明文は、目的を明確に記述している必要があります。これによって、ユーザはもしそのパッケージを削除した場合に何を失うことになるのかを知ることができます。のちの影響について明示的にするのもお勧めです。これは、初めのインストール時にインストールされており、

ユーザが明示的にインストールしたわけではないメタパッケージにとって、特に重要です。システムのアップグレードをスムーズに保証するのに重要になり、潜在的な破損をさけるためにユーザに対してアンインストールする気をなくさせることでしょう。

Chapter 7

パッケージ化、そして…

Debian は、単にソフトウェアのパッケージを作ってメンテナンスをしているだけではありません。この章では、単にパッケージを作ってメンテナンスする以外で Debian へ協力・貢献するやり方、多くの場合とても重要となるやり方の情報を取り扱います。

ボランティア組織の例にたがわず、Debian の活動はメンバーが何をしたいのか、時間を割くのにも重大だと思われることが何か、というメンバーの判断に依っています。

7.1 バグ報告

我々としては、Debian パッケージで見つけたバグを登録することを勧めています。実際のところ、大抵の場合は Debian 開発者が第一線でのテスト作業者となっています。他の開発者のパッケージで見つけたバグを報告することで Debian の品質が向上されています。

Debian [バグ追跡システム \(BTS\)](#) の [バグ報告のやり方について \(instructions for reporting bugs\)](#) を参照してください。

いつも使っているメールを受け取れるユーザアカウントからバグを送ってみてください。そうすることで、開発者がそのバグに関するより詳細な情報を必要とする場合にあなたに尋ねることができます。root ユーザでバグを報告しないでください。

バグを報告するには、`reportbug(1)` のようなツールが使えます。これによって作業を自動化し、かなり簡単なものにできます。

パッケージに対して既にバグが報告されていないことを確認しておいてください。個々のパッケージに対するバグのリストは <https://bugs.debian.org/> パッケージ名にて簡単に確認できます。`querybts(1)` のようなユーティリティでもこの情報を入手できます (なお、`reportbug` では大抵の場合、バグを送信する前に `querybts` の実行も行っています)。

正しい所にバグを報告する様に心がけてください。例えばあるパッケージが他のパッケージのファイルを上書きしてしまうバグの場合ですが、バグ報告が重複して登録されるのを避けるため、これらのパッケージの両方のバグリストを確認してください。

さらに言うと、他のパッケージについても、何度も報告されているバグをマージしたり既に修正されているバグに「fixed」タグをつけたりすることもできます。そのバグの報告者であったりパッケージメンテナではない場合は (メンテナから許可をもらっていないければ)、実際にバグをクローズしてはいけないことに注意してください。

時折、あなたが登録したバグ報告について何が起きているのかをチェックしたくなることでしょう。これは、もう再現できないものをクローズするきっかけになります。報告した全てのバグ報告を確認するには、<https://bugs.debian.org/from:your-email-addr> を参照すればいいだけです。

7.1.1 一度に大量のバグを報告するには (mass bug filing)

大量の異なるパッケージに対して、同じ問題についての非常に多くのバグ (例えば 10 個以上) を報告するのは、推奨されていないやり方です。不要なバグ報告を避けるため、可能な限りの手続きを踏むようにしましょう。例えば、問題の確認を自動化できる場合は `lintian` に新しくチェック項目を追加すれば、エラーや警告が明確になります。

同じ問題について一度に 10 個以上のバグを報告する場合は、バグ報告を登録する前に debian-devel@lists.debian.org へ送ることをお勧めします。バグ報告を送る前に注意点を記述し、メールのサブジェクトに事実を挙げておきます。これで他の開発者がそのバグが本当に問題であるかどうかを確認

できるようになります。さらに、これによって複数のメンテナが平行して同じバグ報告を登録するのを防止できるようになります。

dd-list プログラムを利用すること、明確になっているのであれば影響を受けるパッケージのリストを (devscripts パッケージの) **whodepends** を使って出力して、debian-devel@lists.debian.org へのメールに含めて下さい。

同じサブジェクトで大量のバグを送信する際は、バグ報告がバグ情報用メーリングリストへ転送されないように maintonly@bugs.debian.org へバグ報告を送るべきであるの注意してください。

7.1.1.1 Usertag

多数のパッケージに対するバグを登録する際に BTS の usertag を使いたくなるかもしれません。usertag は 'patch' や 'wishlist' のような通常のタグに似ていますが、ユーザが定義する事と特定のユーザに対して一意な名前空間を占めるという点で違ってきます。これによって、同じバグについて衝突する事無しに、開発者がそれぞれ別のやり方で複数の設定ができるようになります。

バグを登録する際に usertag を追加するには、擬似ヘッダ (pseudo-header) User と Usertags を指定します。

```
To: submit@bugs.debian.org
Subject: バグタイトル

Package: パッケージ名
[ ... ]
User: メールアドレス
Usertags: タグ名 [ タグ名 ... ]

バグの説明 ...
```

タグは空白で区切られ、アンダースコア () を含まれないことに注意してください。特定のグループやチームについてのバグを登録する場合は、適切なメーリングリストで注意を促した上で User をそのメーリングリストに設定するのをお勧めします。

特定の usertag でバグを参照する場合は <https://bugs.debian.org/cgi-bin/pkgreport.cgi?users=メールアドレス &tag=タグ名> を指定してください。

7.2 品質維持の努力

7.2.1 日々の作業

品質保証に割り当てられたグループの人々がいたとしても、QA 作業は彼らのみに課せられるものではありません。あなたもパッケージを可能な限りバグが無いように保ち、できるだけ lintian clean な状態 (項 A.2.1 を参照) にすることで品質保証の作業に参加することができるのです。それが可能ではないように思えるなら、パッケージをいくつか「放棄 (orphan)」してください (項 5.9.4 参照)。または、溜まったバグ処理に追いつくため、他の人々に助力を願い出ることも可能です (debian-qa@lists.debian.org や debian-devel@lists.debian.org で助けを求めることができます)。同時に共同メンテナ (co-maintainer) を探すことも可能です (項 5.12 を参照してください)。

7.2.2 バグ潰しパーティ (BSP)

時折、QA グループは可能な限りの問題を無くすためにバグ潰しパーティ (BSP) を開催しています。開催案内は debian-devel-announce@lists.debian.org で行われ、パーティがどの部分に集中して行われるのかが告知されます。大抵はリリースクリティカルバグ (RC) への対応に当てられますが、大きなアップグレード (新しい perl バージョンでの全バイナリモジュールを再コンパイルのような) を完了する手助けのために開かれることもあります。

メンテナ以外によるアップロード (non-maintainer upload) のルールはパーティの開催中とそれ以外で違います。これは、パーティのアナウンスは NMU の予告であると考えられているためです。パーティによって影響を受けるパッケージを持っている場合 (例えばリリースクリティカルバグを持っている場合) は、それぞれ対応するバグについて現状説明とこのパーティでどのようなことを期待しているのかを説明しないとイケません。NMU を望まない、パッチでの対応にのみ興味がある、あるいは自分自身で対処をする予定の場合は、BTS で説明をしてください。

パーティに参加している人には NMU についての特別ルールが割り当てられます。NMU について、少なくとも 3 日遅延してアップロードを行う場合 (DELAYED/3-day キューなどへのアップロードの場

合)は予告無しで NMU できるのです。他の NMU ルールは全て通常通りに適用されます - NMU のパッチを BTS に送付する必要があります (修正されていないバグのいずれかを NMU で修正する、あるいは新たなバグにパッチを送り、fixed とタグを付けるなど)。それから、作業者は各メンテナの要望に沿う必要があります。

NMU をする自信が無い場合は、BTS へパッチを投げるだけにしてください。NMU でパッケージを壊してしまうより、遥かに良いことです。

7.3 他のメンテナに連絡を取る

Debian と共に過ごす間、様々な理由で他のメンテナに連絡を取りたくなることでしょう。関連パッケージ間での共同作業の新たなやり方について協議したい場合や、開発元で自分が使いたい新しいバージョンが利用可能になっていることを単に知らせたい場合などです。

パッケージメンテナのメールアドレスを探しだすのは骨が折れます。幸いな事に、パッケージ名 @packages.debian.org というシンプルなメールのエイリアスがあり、メンテナらの個人アドレスが何であれメンテナへメールを届ける手段となっています。パッケージ名はパッケージのソース名かバイナリパッケージ名に置き換えてください。

項4.10 経由でソースパッケージの購読を行っている人に連絡を取りたくなるかもしれません。その場合はパッケージ名 @packages.qa.debian.org メールアドレスが使えます。

7.4 活動的でない、あるいは連絡が取れないメンテナに対応する

パッケージがメンテナンスされていないと気づいた場合、メンテナが活動的でパッケージの作業を続けるかどうかを確認する必要があります。もはや活動的な状態ではない可能性もありますが、言わばシステムに登録していなかったという可能性もあります。あるいは、単に確認が必要なだけという可能性もあります。

Missing In Action (行方不明) だと考えられているメンテナについての情報が記録されるシンプルなシステム (MIA データベース) があります。品質保証グループ (QA グループ) のメンバーが活動的ではないメンテナに連絡を取った場合や、そのメンテナについて新たな情報をもたらされた場合、その記録が MIA データベースに残されます。このシステムは qa.debian.org ホスト上の /org/qa.debian.org/mia で利用可能になっており、**mia-query** ツールで検索ができます。どうやってデータベースを検索するのかについては **mia-query --help** と入力してください。活動的ではないメンテナについての情報がまだ記録されていない、あるいはそのメンテナについての情報を追加できる場合は、おおよそ以下の手続きを行う必要があります。

最初の一步はメンテナに丁寧にコンタクトを取り、応答するのに十分な時間待つことです。十分な時間というのを定義するのは非常に困難ですが、実生活では時折非常に多忙になるのを考慮に入れると重要なことです。一つのやり方としては、リマインダーを二週間後に送るという方法があります。

メンテナが4週間(1ヶ月)応答をしない場合、おそらく反応がないと判断できます。このような場合はより詳細に確認し、可能な限り問題となっているメンテナに関する有用な情報をかき集める必要があります。これには以下のようなものが含まれています。

- **開発者 LDAP データベース** を通じて得られる echelon 情報は、開発者が最後に Debian メーリングリストに投稿したはいつなのかを示します (これには debian-devel-changes@lists.debian.org での配布物のアップロードのメールも含まれます)。また、データベースでメンテナが休暇中かどうかも確認してください
- このメンテナが対応しているパッケージ数やパッケージの状態。特に、長期間放置され続けている RC バグがあるかどうか? さらに通常どの程度の数のバグがあるか? もう一つの重要な情報はパッケージが NMU されているかどうか、されているとしたら誰によって行われているか、です。
- Debian 以外でメンテナの活動があるかどうか? 例えば、近頃 Debian 以外のメーリングリストや news グループへの投稿をしているなど。

パッケージがスポンサーされている、つまりメンテナが公式の Debian 開発者ではない場合はちょっとした問題となります。例えば echelon の情報は、スポンサーされている人は利用できません。そのため実際にパッケージをアップロードした Debian 開発者を探して確認を取る必要があります。彼らがパッケージにサインしたということは、アップロードについて何であれ責任を持ち、スポンサーした人に何が起きているかを知っていそうだとことです。

debian-devel@lists.debian.org に、活動が見えないメンテナの居所に誰か気づいているかという質問を投稿するのもあります。問題の人を Cc: に入れてください。

ここに書かれた全てを収集したなら、mia@qa.debian.orgに連絡しましょう。この名前の宛先を担当している人は、あなたが供給した情報を使ってどう進めるかを判断します。例えば、そのメンテナのパッケージの一部または全てを放棄 (Orphan) するかもしれません。パッケージが NMU されていた場合は、パッケージを放棄 (Orphan) する前に NMU をした人に連絡する事を選ぶかもしれません—NMU をした人はきっとパッケージに関心があるでしょうから。

最後に一言: 礼儀正しく振る舞いましょう。我々は所詮ボランティアで、全ての時間を Debian に捧げられるわけではありません。また、関わっている人の状況がわかるわけでもありません。重い病気にかかっているかもしれないし、あるいは死んでしまっているかもしれません - メッセージを受け取る側にどんな人がいるかは分かりません。亡くなった方のご親戚の方がメールを読んだ場合に、非常に無礼で怒った叱責調のメッセージを見つけてどうお感じになるかを想像してください。

一方で、我々はボランティアではありますが責任を持っています。全体の幸せの重要性をよく考える必要があります—もしメンテナが時間が足りなかったり、もう興味を無くしてしまった場合は、パッケージを誰か他のより時間がある人間に与えるべきです。

MIA チームで働くのに興味を持った場合は、技術上の詳細と MIA の手順が記載されている qa.debian.org 上の [/org.qa.debian.org/mia](http://org.qa.debian.org/mia) 内の README ファイルを参照して、mia@qa.debian.org に連絡を取ってください。

7.5 Debian 開発者候補に対応する

Debian の成功は新たな才能あるボランティアをどう魅了し確保するかにかかっています。あなたが経験豊かな開発者なら、新たな開発者を呼び込むプロセスに関与するべきです。このセクションでは新たな開発者候補者をどうやって手助けするのかについて記述します。

7.5.1 パッケージのスポンサーを行う

パッケージのスポンサーになるというのは、自分の権限ではパッケージをアップロードできないメンテナのためにパッケージをアップロードする、ということです。これは些細な問題ではなく、スポンサーはパッケージを精査して Debian が求めるような高いレベルの品質であることを保証する必要があります。

Debian 開発者はパッケージをスポンサーできます。Debian メンテナはできません。

パッケージのスポンサー作業の流れは以下の通りです:

1. メンテナはソースパッケージ (.dsc) を用意してオンライン上の何処か (例えば mentors.debian.net) に置く、あるいはもっと良いのは、パッケージがメンテナンスされている公開 VCS リポジトリへのリンクを提供することです (項4.4.5 参照)。
2. スポンサーはソースパッケージをダウンロード (あるいはチェックアウト) します。
3. スポンサーはソースパッケージをレビューします。問題を見つけたら、メンテナに知らせて修正版をくれるように尋ねます (作業は step 1 へやり直しされます)。
4. スポンサーは、何も問題が残っていないのを見つけられませんでした。パッケージをビルドし、署名し、Debian へアップロードします。

パッケージのスポンサーのやり方について詳細を詰める前に、提案されたパッケージを追加することが Debian にとって有益であるかどうか、自分自身に問いかける必要があります。

この質問に答えるのは単純ではなく、多くの要因に依っています: 開発元のコードは成熟していて、セキュリティホールは山ではないですか? 同じことができる既存パッケージがありませんか? そしてこの新しいパッケージと比べてどうですか? 新しいパッケージはユーザから要求されたものですか? そしてユーザ数はどの程度の大きさですか? 大本の開発者らはアクティブですか?

それから、メンテナ候補者が良いメンテナになるであろうことを保証する必要があります。他のパッケージでの経験がありますか? そうであれば、良い仕事をしていますか (バグを確認している)? パッケージと使われているプログラミング言語について詳しいですか? そのパッケージに必要なスキルを持っていますか? そうでなければ、学ぶことが可能でしょうか?

候補者が、Debian に対してどの様な態度を取っているかを知るのも良い考えです: Debian の哲学に賛同していて、Debian に参加したいと思っていますか? Debian メンテナになるのがどれくらい簡単なのかを考えて、参加を検討している人たちだけをスポンサーするのが良いでしょう。こうすれば、最初からずっとスポンサーとして行動しなくて良いと思っておけます。

7.5.1.1 新しいパッケージのスポンサーを行う

新たなメンテナは、Debian パッケージの作成する際に大抵何らかの困難に会います—これは非常に理解できることです。彼らは間違いを犯します。これは、全く新しいパッケージを Debian でスポンサーするのに、Debian パッケージングの徹底的なレビューを受ける必要がある根拠です。時折、パッケージが Debian へアップロードされるのに十分な状態になるまで、複数回のやり取りが必要になることがあります。それゆえ、スポンサーになることは、メンターになることを伴います。

レビューをせずに新しいパッケージのスポンサーをしないでください。ftpmaster による新しいパッケージのレビューは、主にソフトウェアが本当にフリーなものであるかを確認するためです。もちろん、パッケージ化に関する問題に偶然気づくことはありますが、それを期待すべきではありません。アップロードされたパッケージが、Debian フリーソフトウェアガイドラインに適合し、良い品質であることを保証するのは、あなたの仕事です。

パッケージをビルドし、ソフトウェアのテストを行うのはレビューの一部ではありますが、それだけでは十分ではありません。この章の残りの部分では、レビューでチェックするポイントの一覧を述べます (徹底的なものではありません)。¹

- upstream の tarball として提供されているものが、upstream の作者が配布しているものと同じかどうかを確認する (ソースが Debian 用に再パッケージされている場合、修正した tarball を自分自身で生成する)。
- **lintian** を実行する (項 A.2.1 参照)。多くの一般的な問題を見つけてくれます。**lintian** の overrides 設定がメンテナによって設定されている場合、完全に問題がないことを確認してください。
- **licensecheck**(項 A.6.1 の一部) を実行し、debian/copyright が正しく、そして完全な事を確認する。ライセンス問題を探してください (頭に “All rights reserved” とあるファイルや、DFSG に適合しないライセンスがあるなど)。この作業には、**grep -ri** が助けとなることでしょう。
- ビルドの依存関係が完全であることを保証するため、パッケージを **pbuilder** (やその他類似のツール) でビルドする (項 A.4.3 参照)。
- debian/control を査読する: ベストプラクティスに従っている? (項 6.2 参照) 依存関係は完璧ですか?
- debian/rules を査読する: ベストプラクティスに従っている? 改善可能な点がある?
- メンテナスクリプト (preinst, postinst, prerm, postrm, config) を査読する: 依存関係がインストールされていない時でも動作する? 全てのスクリプトが等冪 (idempotent、すなわち、問題無しに複数回実行できる)?
- 開発元のファイルに対する変更 (.diff.gz、debian/patches/、あるいは直接 debian tarball に埋め込まれているバイナリファイル) をレビューする。十分な根拠がありますか? (パッチに対し、**DEP-3** に沿って) 正しくドキュメント化されている?
- すべてのファイルについて、そのファイルが何故そこにあるのか、望んでいる結果をもたらすためにそれが正しいやり方かどうかを自身に問いかけてください。メンテナはパッケージ化のベストプラクティスに従っていますか? (第 6 章参照)
- パッケージをビルドし、インストールし、ソフトウェアを使ってみてください。パッケージを削除 (remove)、及び完全削除 (purge) できることを確認してください。**piuparts** でテストすると良いかもしれません。

監査で何も問題が見つからなかった場合には、パッケージをビルドして Debian へアップロードすることができます。あなたがメンテナでなかったとしても、スポンサーとして Debian へアップロードされたものへの責任を持つことを覚えておってください。これが、項 4.10 を使ってパッケージを追いかけておくことが推奨される理由です。

changelog ファイルや control ファイルにあなたの名前を入れるために、ソースパッケージを変更する必要はないことに注意してください。control ファイルの Maintainer 欄と changelog にはパッケージ作業を行った人を記載する必要があります。つまりはスポンサー対象者、ということです。そうすることで、メンテナはすべての BTS メールを受け取るようになります。

代わりに、署名にあなたの鍵を使うために **dpkg-buildpackage** に指示する必要があります。これは **-k** オプションを使って行います:

¹ もっと多くのチェック項目について、複数の開発者が持ち寄っている [sponsorship checklists](#) で見ることができます。

```
dpkg-buildpackage -kKEY-ID
```

debuild と **debsign** を使う場合は、`~/.devscripts` に設定を決め打ちで書いても構いません:

```
DEBSIGN_KEYID=KEY-ID
```

7.5.1.2 既存パッケージの更新をスポンサーする

通常、パッケージは既に全体的なレビューを受けているとします。ですので、もう一度すべてを行う代わりに、現在のバージョンとメンテナによって準備された新しいバージョンとの差分を注意深く分析することになります。最初のレビューをあなた自身が行っていない場合は、最初のレビューが正しい加減であった場合に備えて、より子細に確認を行った方が良いでしょう。

差分を分析する為には両方のバージョンが必要です。ソースパッケージの現在のバージョンをダウンロードし (**apt-get source**)、リビルドします (あるいは **aptitude download** で現在のバイナリパッケージをダウンロードします)。それから、スポンサー用のソースパッケージをダウンロードします (通常、**dget** を使います)。

まず、**changelog** の新しいエントリを読みます。これで、レビュー中に予期しておく事柄を知ることができます。使うことになる主なツールは **debdiff** です (`devscripts` パッケージに含まれています)。2つのソースパッケージ (`.dsc` ファイル) に対して実行するか、2つのバイナリパッケージ、あるいは2つの `.changes` ファイルに対して実行することができます (その場合は `.changes` に記載されているすべてのバイナリファイルを比較します)。

ソースパッケージを比較した場合 (新しい開発元のバージョンの場合には、例えば **debdiff** の出力を **filterdiff -i */debian/*** などとして、開発元のファイルを除外します)、確認したすべての変更点を理解して、この変更点が Debian の **changelog** に正しく記載されている必要があります。

何も問題がなければ、パッケージをビルドし、ソースパッケージ上の変更が期待していない結果 (ミスのためファイルがなくなっている、依存関係の欠落など) をもたらしていないかを確認するため、バイナリパッケージを比較します。

メンテナが何か重要なことを見逃していないかを確認する為に、パッケージ追跡システム (項4.10 参照) を見るのが良いでしょう。追加が可能な翻訳の更新が BTS にあるかもしれません。パッケージが NMU されていて、メンテナが NMU での変更をパッケージへ取り入れるのを忘れているかもしれません。リリースクリティカルバグがあって、メンテナが放置しているためにテスト版 (testing) への移行が阻まれているかもしれません。メンテナが何か (より良く) できることを見つけたら、それを伝えましょう。次回に改善ができますし、メンテナは責務についてより深く理解することになります。

何も大きな問題を見つけないければ、新しいバージョンをアップロードします。そうでなければ、メンテナに修正したバージョンをアップロードするよう要請します。

7.5.2 新たな開発者を支持する (advocate)

Debian ウェブサイトの [開発者志願者の支持者になる \(advocating a prospective developer\)](#) のページを参照してください。

7.5.3 新規メンテナ申請 (new maintainer applications) を取り扱う

Debian のウェブサイトにある [申請管理者用チェックリスト \(Checklist for Application Managers\)](#) を参照してください。

Chapter 8

国際化と翻訳

Debian がサポートしている自然言語の数は未だ増え続けています。あなたが英語圏のネイティブスピーカーで他の言語を話さないとしても、国際化の問題について注意を払うことはメンテナとしてのあなたの責務です (internationalization の 'i' と 'n' の間に 18 文字があるので i18n と略されます)。つまり、あなたが英語のみのプログラムを扱っていて問題がない場合であっても、この章の大部分を読んでおく必要があるということです。

久保田智広さんによる [Introduction to i18n](#) によると、I18N (internationalization) はソフトウェアや関連する技術を調整し、ソフトウェアが複数の言語、習慣、その他世界の物事などを扱えるようにしておくことで、対して L10N (localization) は既に国際化されているソフトウェアに対して特定の言語を実装することを意味します。

I10n と i18n は関連していますが、それぞれ関連する難しさについては違います。プログラムをユーザの設定に応じて表示されるテキストの言語を変更するようになるのはあまり難しくはありませんが、実際にメッセージを翻訳するのはとても時間がかかります。一方、文字のエンコード設定は些細な事ですが、複数の文字エンコードを扱えるようなコードにするのはとても難しい問題です。

i18n の問題を横においたとしても、一般的なガイドラインは与えられておらず、移植作業用の build 系のメカニズムと比較できるような、Debian での I10n 用の中心となるインフラは実際のところ存在していません。そのため、多くの作業は手動で行わねばなりません。

8.1 どの様にして Debian では翻訳が取り扱われているか

パッケージに含まれている文章の翻訳の取り扱いは未だ手動であり、作業のやり方は翻訳を表示させたい文の種類に因ります。

プログラムのメッセージについては、ほとんどの場合 gettext という仕組みが使われています。多くの場合、翻訳は [Free Translation Project](#) や [Gnome 翻訳プロジェクト](#)、[KDE one](#) などの開発元 (upstream) のプロジェクトで取り扱われています。Debian で唯一の集約化された情報は [Debian の翻訳に関する統計](#) で、実際のパッケージ内での翻訳ファイルの状況について確認できますが、翻訳作業を実際に容易にする仕組みではありません。

パッケージ説明文の翻訳作業はかなり昔に始まりました。実際にそれを使うツールがほんの少ししか機能を提供していなかったとしても (つまり、APT だけが設定を正確に行ったときのみ利用できたのです)。メンテナはパッケージの説明文をサポートするのに何も特別なことをする必要はありません。翻訳者は [Debian Description Translation Project \(DDTP\)](#) を使う必要があります。

debconf テンプレートについては、メンテナは翻訳者の作業を容易にするため po-debconf パッケージを使う必要があります。翻訳者は作業に DDTP を使うことが出来ます (フランスチームとブラジルチームは使っていませんが)。[DDTP のサイト](#) (実際に何が翻訳されているか) と [Debian の翻訳に関する統計](#) サイト (パッケージに何が含まれているか) の双方で統計情報を得ることが出来ます。

ウェブページについては、それぞれの I10n チームが対応する VCS にアクセスし、Debian の翻訳に関する統計サイトから統計情報が取得できます。

Debian についての一般的なドキュメントは、作業は多少の差はあれウェブページと同じです (翻訳者は VCS にアクセスします)。ですが、統計情報のページはありません。

パッケージ固有のドキュメント (man ページ、info ドキュメントその他) は、ほとんどすべてが手付かずです。

特記しておくこととして、KDE プロジェクトはドキュメントの翻訳をプログラムのメッセージと同じやり方で取り扱っています。

Debian 固有の man ページを [特定の VCS リポジトリ](#) で取り扱おうという動きもあります。

8.2 メンテナへの I18N & L10N FAQ

これはメンテナが i18n や l10n を考えるのにあたって直面するであろう問題の一覧です。読み進める間、Debian でこれらの点について実際のコンセンサスは得られていないことを念頭においてください。これは単にアドバイスです。出てきた問題についてもっと良い考えがある、あるいはいくつかの点で賛同できないという場合は、連絡をして頂いて構いません。そのことによって、この文章の質をさらに高めることができます。

8.2.1 翻訳された文章を得るには

パッケージの説明文や debconf テンプレートを翻訳してもらうには、あなたは何もする必要はありません。DDTP のインフラが作業者に翻訳してもらう素材を割り当てるのに、あなた側から働きかける必要はありません。

他の素材 (gettext ファイル、man ページ、その他のドキュメント) については、最も良い解決策は文章をインターネットのどこかに置いて、debian-i18n で他の言語へ翻訳を頼むことです。翻訳チームのメンバーの何名かはこのメーリングリストに登録しており、翻訳とレビュー作業を担当します。一旦作業が完了すれば、翻訳された文章があなたのメールボックスへと届くでしょう。

8.2.2 どの様にして提供された翻訳をレビューするか

時折、あなたのパッケージ内の文章を訳して翻訳をパッケージに含めるように依頼する人が出てきます。これはあなたがその言語に詳しくない場合、問題となり得ます。その文章を対応する l10n メーリングリストに投稿し、レビューを依頼するのが良い考えです。一旦レビューが終われば、翻訳の質に自信を持つでしょうし、パッケージに含めるのにも安心を覚えるでしょう。

8.2.3 どの様にして翻訳してもらった文章を更新するか

古いままになっていた文章に対して翻訳文がある場合、元の文章を更新する度に、以前翻訳した人に新たに変更した点に合わせて翻訳を更新してもらうように依頼する必要があります。この作業には時間がかかることを覚えておいてください。更新をレビューしてもらったりするには少なくとも 1 週間はかかります。

翻訳者が応答してこない場合、対応する l10n メーリングリストに助力を願い出しましょう。すべてうまくいかなかった場合は、翻訳した文中に翻訳がとにかく古い事の警告を入れておくの忘れないようにして、できれば読者がオリジナルの文章を参照するようにしましょう。

古くなっているからといって翻訳を全て削除するのは避けてください。非英語圏のユーザにとって何もドキュメントが無いよりは古いドキュメントがある方が有益であることが往々にしてあります。

8.2.4 どの様にして翻訳関連のバグ報告を取り扱うか

最も良い解決策は開発元のバグという印を付けておいて (forward)、以前の翻訳者と関連するチーム (対応する debian-l10n-XXX メーリングリスト) に転送することです。

8.3 翻訳者への I18N & L10N FAQ

これを読み進める間、Debian においてこれらの点に関する一般的な手続きは存在していないこと、そしていかなる場合でもチームやパッケージメンテナと協調して作業する必要があることを念頭においてください。

8.3.1 どの様にして翻訳作業を支援するか

翻訳したい文章を選び、誰もまだ作業をしていないことを確認し (debian-l10n-XXX メーリングリストを参照。日本語の場合は debian-doc@debian.or.jp を参照してください)、翻訳し、l10n メーリングリストで他のネイティブスピーカーにレビューをしてもらい、パッケージメンテナに提供します (次の段を参照)。

8.3.2 どの様にして提供した翻訳をパッケージに含めてもらうか

含めてもらう翻訳が正しいかどうかを提供する前に確認してください (110n メーリングリストでレビューを依頼しましょう)。皆の時間を節約し、バグレポートに複数バージョンの同じ文章があるというカオス状態を避けることになります。

最も良いやり方は、パッケージに対して翻訳を含めて通常のバグとして登録することです。忘れずに「patch」タグを使い、翻訳が欠けていたとしてもプログラムの動作に支障は無いので「wishlist」以上の重要度を使わないようにしましょう。

8.4 110n に関する現状でのベストプラクティス

- メンテナとしては、翻訳については関連の 110n メーリングリストに尋ねること無くどのような方法であれいじらないこと (レイアウトを変えることでさえしないこと) です。もしいじってしまうと、例えばファイルのエンコーディングを破壊する危険があります。さらに、あなたが間違いだと思っていることがその言語では正解である (または必要ですらある) ことがあります。
- 翻訳者としては、元の文章に間違いを見つけた場合は必ず報告することです。翻訳者はしばしばその文章の最も注意深い読者であり、翻訳者が見つけた間違いを報告しないのならば誰も報告しないでしょう。
- いずれの場合でも、110n に関する最も大きな問題は複数人の協調であり、誤解から小さな問題でフレームワークを起こすのはとても簡単だということです。ですから、もし、あなたの話し相手と問題が起こっている場合は、関連する 110n メーリングリストや `debian-i18n` メーリングリスト、さらにあるいは `debian-devel` メーリングリストに助けを求めてください (ですが、ご注意を。110n 関連の議論は `debian-devel` では頻繁にフレームワークになります)。
- 何にせよ、協調は互いを尊敬しあうことによってのみ成し得ます。

Appendix A

Debian メンテナツールの概要

この章には、メンテナが利用できるツールについて大まかな概要が含まれています。以下は完全なものでも決定版的なものでもありませんが、よく使われているツールについての説明です。

Debian メンテナツールは、開発者を手助けし、重要な作業のために時間を作れるようにしてくれるものです。Larry Wall が言うように、やり方は一つではありません (there's more than one way to do it)。

高度なパッケージメンテナツールの使用を好む人もいればそうではない人もいます。Debian は公式にはこの問題を不可知論であるとしています。どのようなツールでも作業ができるのであれば構いません。つまり、この章は誰もがどのツールを使うべきか、メンテナン上で何をすべきかと要求するものではないということです。あるいは競合するツールを排して特定のツールを勧める訳でもありません。

パッケージの説明文のほとんどは実際のパッケージの説明から取ったものです。より詳細な情報はパッケージ内のドキュメントで確認できます。**apt-cache show** パッケージ名コマンドでも情報を得られます。

A.1 主要なツール

以下のツールはどのメンテナであっても、必ず必要とするものです。

A.1.1 dpkg-dev

dpkg-dev は、パッケージを展開、ビルド、Debian ソースパッケージをアップロードするのに必要なツールを含んでいます (**dpkg-source** を含む)。これらのユーティリティはパッケージを作成・操作するのに必要な基礎的で、低レイヤの機能を含んでいます。そのため、これらはあらゆる Debian メンテナにとって必要不可欠なものです。

A.1.2 debconf

debconf は、パッケージを対話形式で設定できる一貫したインターフェイスを提供します。これはユーザインターフェイスに依存せず、エンドユーザがテキストのみのインターフェイス、HTML インターフェイス、ダイアログ形式のインターフェイスでパッケージを設定できます。新たなインターフェイスはモジュールとして追加できます。

このパッケージに関するドキュメントは debconf-doc パッケージ中で確認できます。

多くの人が、対話的な設定を必要とする全てのパッケージにこのシステムが使われるべきだと感じています。項 6.5 を参照してください。現在は debconf は Debian ポリシーで必要であるとはされていませんが、将来には変更されることでしょう。

A.1.3 fakeroot

fakeroot は root 特権をシミュレートします。これは root になること無しにパッケージをビルドできるようにしてくれます (パッケージは通常 root の所有権でファイルをインストールしようとします)。fakeroot をインストールしていれば、**dpkg-buildpackage** で自動的に利用します。

A.2 パッケージチェック (lint) 用ツール

コンピュータ用のフリーオンライン辞書 (Free On-line Dictionary of Computing, FOLDOC) によると、「lint」は C コンパイラよりもより網羅的なチェックを行う Unix C 言語処理器、とあります。パッケージ lint ツールは、パッケージ内の一般的な問題やポリシー違反を自動的に見つけてくれることで、パッケージメンテナを助けてくれます。

A.2.1 lintian

lintian は Debian パッケージを解剖してバグやポリシー違反の情報を出力します。一般的なエラーへのチェック同様に Debian ポリシーの多くの部分を自動チェックする機能を含んでいます。

定期的に最新の lintian を unstable から取得し、パッケージを全てチェックすべきです。-i オプションは、各エラーや警告が何を意味しているのか、ポリシーを元に、詳細な説明を提供してくれ、一般的に問題をどのように修正すべきかを説明してくれることに留意してください。

何時、どのようにして Lintian を使うのか、詳細については項 5.3 を参照してください。

あなたのパッケージに対して Lintian によって報告された問題の要約はすべて <https://lintian.debian.org/> から確認することもできます。このレポートは、最新の lintian による開発版ディストリビューション (unstable) 全体についての出力を含んでいます。

A.2.2 debdiff

(devscripts パッケージ、項 A.6.1 より) debdiff は二つのパッケージのファイルのリストと control ファイルを比較します。前回のアップロードからバイナリパッケージ数が変わったことや、control ファイル内で何が変わったのかなどに気付く手助けをしてくれるなど、簡単にグレッションテストとなります。もちろん、報告される変更の多くは問題ありませんが、様々なアクシデントを防止するのに役立ってくれるでしょう。

バイナリパッケージのペアに対して実行することができます:

```
debdiff package_1-1_arch.deb package_2-1_arch.deb
```

changes ファイルのペアに対してさえも実行できます:

```
debdiff package_1-1_arch.changes package_2-1_arch.changes
```

より詳細については、debdiff(1) を参照してください。

A.3 debian/rules の補助ツール

パッケージ構築ツールは debian/rules ファイルを書く作業を楽にしてくれます。これらが望ましい、あるいは望ましくない理由の詳細については項 6.1.1 を参照してください。

A.3.1 debhelper

debhelper は、Debian パッケージのバイナリを作成するにあたっての共通な作業を自動化するため、debian/rules 内で使うことができるプログラムの集合体です。debhelper は、パッケージに様々なファイルをインストールし、ファイルを圧縮し、ファイルの権限を修正し、パッケージを Debian のメニューシステムに統合するプログラムを含んでいます。

いくつかのアプローチとは違って、debhelper は複数の小さな、シンプルで一貫した方法で動作するコマンドに分割されています。そのため、他の debian/rules 用ツールよりも細やかなコントロールが可能になっています。

ここに記すには一時的な、大量の小さな debhelper のアドオンパッケージがあります。apt-cache search ^dh- と実行することで一覧の多くを参照できます。

A.3.2 dh-make

dh-make パッケージは、ソースツリーを Debian パッケージをビルドするのに必要な雛形ファイルを作成するプログラム **dh_make** を含んでいます。その名が示すように、**dh_make** は debmake を書き直したもので、そのテンプレートファイルは debhelper の **dh_*** プログラムを使うようになっています。

dh_make によって生成された rules ファイルは、大抵の場合作業するパッケージに対して十分な基礎にはなりますが、まだこれは下地でしかありません。メンテナに残っている責務は、生成されたファイルをきれいに整理して、完全に動作してポリシーに準拠したパッケージにすることです。

A.3.3 equivs

`equivs` はパッケージ作成用のもう一つのパッケージです。単純に依存関係を満たしたいだけのパッケージを作成する必要がある場合に、しばしばローカルでの使用を勧められます。時折、他のパッケージに依存することだけが目的のパッケージ、「メタパッケージ (meta-packages)」を作る際にも使われます。

A.4 パッケージ作成ツール

以下のパッケージは、パッケージ作成作業を手助けしてくれます。通常実行する `dpkg-buildpackage` と同様に、パッケージ作成支援の作業を取り扱ってくれます。

A.4.1 git-buildpackage

`git-buildpackage` は、Debian ソースパッケージを Git リポジトリに挿入あるいはインポートし、Debian パッケージを Git リポジトリから生成、そして開発元での変更をリポジトリに統合するのに役立つ機能を提供します。

これらのユーティリティは、Debian メンテナによる Git の利用を促進するインフラストラクチャを提供します。これは、バージョンコントロールシステムの他の利点と同様に、`stable`、`unstable`、おそらく `experimental` ディストリビューション用にパッケージに個々の Git ブランチを持つことができます。

A.4.2 debootstrap

`debootstrap` パッケージとスクリプトは、システムのどこでも Debian ベースシステムをブートストラップできるようにしてくれます。ベースシステムとは、操作するのに必要となる素の最小限パッケージ群を意味し、それに加えてシステムの残りの部分をインストールします。

このようなシステムを持つことは、様々な面で役に立つでしょう。例えば、ビルドの依存関係をテストしたい場合に `chroot` でそのシステムの中に入ることができます。あるいは素のベースシステムにインストールした際にパッケージがどのように振る舞うかをテストできます。`chroot` 作成ツールはこのパッケージを使います。以下を参照ください。

A.4.3 pbuilder

`pbuilder` は `chroot` されたシステムを構築し、パッケージを `chroot` 内部でビルドします。パッケージのビルド依存関係が正しいかどうかをチェックするのにとても役立ち、生成されたパッケージに不必要な誤ったビルド依存関係が存在していないことを確かめられるでしょう。

関連するパッケージが `cowbuilder` です。標準的な Linux ファイルシステム上で COW ファイルシステムを使うことでビルド処理を高速化します。

A.4.4 sbuild

`sbuid` はもう一つの自動ビルドシステムです。同様に `chroot` された環境を使うことが出来ます。単独で使うことも、分散ビルド環境のネットワークの一部として使うこともできます。文字通り、移植者たちによって利用可能な全アーキテクチャのバイナリパッケージをビルドするのに使われているシステムの一部分です。詳細については項 [5.10.3.3](#) を参照してください。それからシステムの動作については <https://buildd.debian.org/> を参照してください。

A.5 パッケージのアップロード用ツール

以下のパッケージはパッケージを公式アーカイブにアップロードする作業を自動化、あるいは単純化してくれるのに役立ちます。

A.5.1 dupload

`dupload` は、自動的に Debian パッケージを Debian アーカイブにアップロードし、アップロードを記録し、パッケージのアップロードについてのメールを送信してくれるパッケージであり、スクリプトです。新しいアップロード先や方法を設定することもできます。

A.5.2 dput

dput パッケージとスクリプトは dupload と同じことを違ったやり方で行います。GnuPG 署名とチェックサムをアップロード前にチェックする機能や、アップロード後に **dinstall** を dry-run モードで実行できるなど、dupload よりもいくつか機能が多くなっています。

A.5.3 dcut

dcut スクリプト (dput パッケージの一部、項A.5.2 参照) は、ftp アップロードディレクトリからファイルを削除するのに役立ちます。

A.6 メンテナンスの自動化

以下のツールは changelog のエントリや署名行の追加、Emacs 内でのバグの参照から最新かつ公式の config.sub を使うようにするまで、様々なメンテナンス作業を自動化するのに役立ちます。

A.6.1 devscripts

devscripts は、Debian パッケージをメンテナンスするのに非常に有用なラッパーやツールを含むパッケージです。スクリプトの例としては debian/changelog ファイルをコマンドラインから操作する **debchange** および **dch**、そして **dpkg-buildpackage** 関連のラッパーである **debuild** を含んでいます。**bts** ユーティリティも、バグ報告の状態をコマンドライン上で更新するのにとても役立ちます。**uscan** はパッケージの新しいバージョン (new upstream version) をチェックするのに使えます。**debsign** は、リモートでアップロード前にパッケージにサインするのに利用できます。これは GPG があるのとは違うマシンでパッケージをビルドした際に便利です。

利用可能なスクリプトの全リストについては devscripts(1) マニュアルページを参照してください。

A.6.2 autotools-dev

autotools-dev は、**autoconf** や **automake** を使っているパッケージをメンテナンスする人にとってのベストプラクティスを含んでいます。また、全ての Debian 移植版で動作することを知られている正規の config.sub および config.guess ファイルを含んでいます。

A.6.3 dpkg-repack

dpkg-repack は既にインストールされているパッケージから Debian パッケージを生成します。パッケージが展開されてから何かしら変更が加えられている場合 (例えば、/etc にあるファイルが変更されているなど)、新しいパッケージは変更を含みます。

このユーティリティは、一つのコンピュータから他のコンピュータへパッケージをコピーするのを簡単にできるようにしてくれます。また、あなたのシステムにはインストールされているがどこでも入手できなくなってしまったパッケージを再作成したり、アップグレード前にパッケージの現在の状態を保存するのに使えます。

A.6.4 alien

alien は、Debian、RPM (RedHat)、LSB (Linux Standard Base)、Solaris、Slackware などの各種バイナリパッケージのパッケージ形式を変換します。

A.6.5 dpkg-dev-el

dpkg-dev-el は、パッケージの debian ディレクトリにあるファイルを編集する際に手助けしてくれる Emacs lisp パッケージです。例えば、パッケージの現在のバグをリストアップしてくれたり、debian/changelog ファイル中の最新のエントリの終端処理してくれたりするための手軽な機能があります。

A.6.6 dpkg-depcheck

(devscripts パッケージ、項A.6.1 より) **dpkg-depcheck** は、指定されたコマンドによって使われた全てのパッケージを確認するため、コマンドを **strace** の下で実行します。

Debian パッケージについていうと、これは新しいパッケージの Build-Depends 行を構成するのが必要になった際に役立ちます。**dpkg-depcheck** を通してビルド作業を実行すると、最初の大まかなビルドの依存関係を良い形で得られます。例えば以下のようになります:

```
dpkg-depcheck -b debian/rules build
```

dpkg-depcheck は、特にパッケージが他のプログラムを実行するのに **exec(2)** を使っている場合に実行時の依存性を確認するのに使えます。

より詳細については、**dpkg-depcheck(1)** を参照してください。

A.7 移植用ツール

以下のツールが、移植作業やクロスコンパイル作業に役立ちます。

A.7.1 dpkg-cross

dpkg-cross は、**dpkg** に似た方法でクロスコンパイルするためのライブラリとヘッダをインストールするツールです。さらに、**dpkg-buildpackage** および **dpkg-shlibdeps** の機能がクロスコンパイルをサポートするように拡張されます。

A.8 ドキュメントと情報について

以下のパッケージが、メンテナへの情報提供やドキュメントの作成に役立ちます。

A.8.1 docbook-xml

docbook-xml は Debian のドキュメントで一般的に使われている DocBook XML DTD を提供します (古いものは **debiandoc SGML DTD** を使っています)。例えば、このマニュアルは DocBook XML で書かれています。

docbook-xsl パッケージは、ソースをビルドして様々な出力フォーマットに整形する XSL ファイルを提供します。XSL スタイルシートを使うには **xsltproc** のような XSLT プロセッサが必要になります。スタイルシートのドキュメントは各種 **docbook-xsl-doc-*** パッケージで確認できます。

FO から PDF を生成するには、**xmlroff** や **fop** のような FO プロセッサが必要です。他に DocBook XML から PDF を生成するツールとしては **dblatex** があります。

A.8.2 debiandoc-sgml

debiandoc-sgml は Debian のドキュメントで一般的に使われている DebianDoc SGML DTD を提供します。しかし、現在は非推奨 (deprecated) となっています (代わりに **docbook-xml** を使うようにしてください)。これも、ソースをビルドして様々な出力フォーマットに整形するスクリプトを提供します。

ドキュメント用の DTD は **debiandoc-sgml-doc** パッケージで確認できます。

A.8.3 debian-keyring

Debian 開発者および Debian メンテナの公開 GPG 鍵を含んでいます。詳細については項3.2.2 とパッケージ内のドキュメントを参照してください。

A.8.4 debian-el

debian-el は、Debian バイナリパッケージを参照する Emacs モードを提供します。これを使うと、パッケージを展開しなくても実行できるようになります。