

Debian パッケージングチュートリアル

Lucas Nussbaum

`packaging-tutorial@packages.debian.org`

version 0.13 – 2014-06-29



このチュートリアルについて

- ▶ 目的: **Debian**のパッケージ作成について、知る必要のあることの提供
 - ▶ 既存パッケージの修正
 - ▶ 自作パッケージの作成
 - ▶ Debian コミュニティとのやりとり
 - ▶ Debian のパワーユーザーになる
- ▶ 最も重要な点を押さえているが不完全
 - ▶ 詳細なドキュメントを参照
- ▶ ほとんどの内容は Debian 派生ディストリビューションにも適用可能
 - ▶ Ubuntu を含む



アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



Debian

- ▶ **GNU/Linux** ディストリビューション
- ▶ 「GNU の精神でオープンに」 開発している
第一のメジャーディストリビューション
- ▶ 非商用: 1,000 人以上のボランティアが協力して開発
- ▶ 3つの主要機能
 - ▶ 品質 – 技術的利点の文化
準備できた時にリリース
 - ▶ 自由 – 開発者とユーザーは、1993 年に成立した、
フリーソフトウェアの文化を促す社会契約で結ばれている。
 - ▶ 独立 – Debian のお守りをしている (単一) 企業はない
また、オープンな意思決定プロセス (実行主義 + 民主主義)
- ▶ 最高の アマチュア が、好きだからこそ成し遂げた



Debian パッケージ

- ▶ **.deb** ファイル (バイナリパッケージ)
- ▶ ソフトウェアをユーザーに配布する、とても強力で便利な方法
- ▶ もっとも一般的なパッケージフォーマットのひとつ (もうひとつは RPM)
- ▶ ユニバーサル:
 - ▶ 30,000 のバイナリーパッケージ
→ ほとんどのフリーソフトウェアがDebianでパッケージ化
 - ▶ 2 つの非 Linux (Hurd, KFreeBSD) を含む 12 の移植版 (アーキテクチャ)
 - ▶ 120 の Debian 派生ディストリビューションでも使用



deb パッケージフォーマット

▶ .deb ファイル: ar アーカイブ

```
$ ar tv wget_1.12-2.1_i386.deb
rw-r--r-- 0/0      4 Sep  5 15:43 2010 debian-binary
rw-r--r-- 0/0    2403 Sep  5 15:43 2010 control.tar.gz
rw-r--r-- 0/0  751613 Sep  5 15:43 2010 data.tar.gz
```

- ▶ debian-binary: deb ファイルフォーマットのバージョン "2.0\n"
 - ▶ control.tar.gz: パッケージについてのメタデータ
control, md5sums, (pre|post)(rm|inst), triggers, shlibs, ...
 - ▶ data.tar.gz: パッケージのデータファイル
- ## ▶ .deb ファイルを手で作ることも可能
- http://tldp.org/HOWTO/html_single/Debian-Binary-Package-Building-HOWTO/
- ## ▶ しかし、ほとんどの人には不要

本チュートリアル: **Debian** のパッケージを **Debian** 流に作成



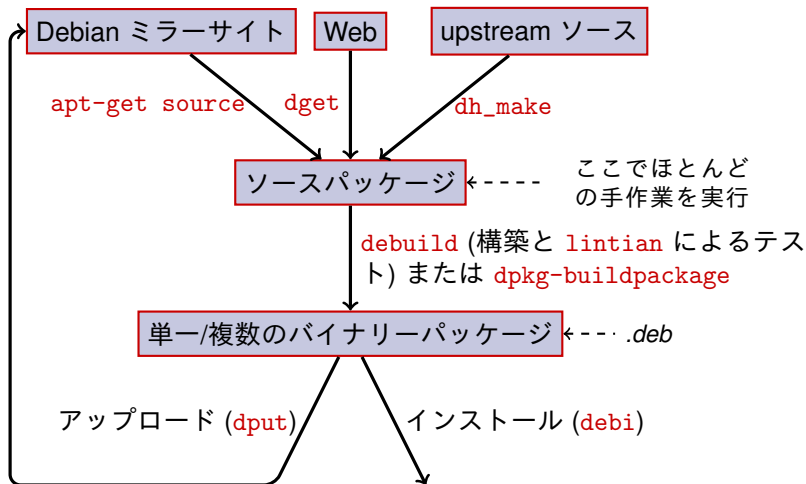
必要なツール

- ▶ Debian (ないし Ubuntu) システム (要 root アクセス)
- ▶ いくつかのパッケージ:
 - ▶ **build-essential**: 開発者のマシンで利用前提となるパッケージに依存 (パッケージの Build-Depends: コントロールフィールドに指定不要)
 - ▶ パッケージを作成する、基本的な Debian 特化ツールである **dpkg-dev** への依存関係を含む
 - ▶ **devscripts**: Debian メンテナにとって便利なスクリプト群

debhelper, cdb, quilt, pbuilder, sbuild, lintian, svn-buildpackage, git-buildpackage, ... といった、その他たくさんのパッケージ (後述) 必要に応じてインストール。



一般的なパッケージングワークフロー



例: **dash** の再構築

- 1 dash を構築するのに必要なパッケージと devscripts のインストール

```
sudo apt-get build-dep dash
```

(/etc/apt/sources.list に deb-src 行が必要)

```
sudo apt-get install --no-install-recommends devscripts fakeroot
```
- 2 作業ディレクトリーを作成し、そこに移動

```
mkdir /tmp/debian-tutorial ; cd /tmp/debian-tutorial
```
- 3 dash のソースパッケージを入手

```
apt-get source dash
```

(/etc/apt/sources.list に deb-src 行が必要)
- 4 パッケージの構築

```
cd dash-*
```

```
debuild -us -uc
```

(-us -uc は GPG によるパッケージ署名を無効化)
- 5 結果の確認
 - ▶ 新しい .deb ファイルが親ディレクトリーに
- 6 debian/ ディレクトリーを参照
 - ▶ パッケージング作業を行う場所



アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



ソースパッケージ

- ▶ 1つのソースパッケージから複数のバイナリーパッケージを生成
例: `libtar` のソースから `libtar0` と `libtar-dev` のバイナリーパッケージを生成
- ▶ 2種類のパッケージ: (よく判らなければ非ネイティブで)
 - ▶ ネイティブパッケージ: 通常 Debian 固有ソフトウェア (`dpkg`, `apt`)
 - ▶ 非ネイティブパッケージ: Debian 外で開発されたソフトウェア
- ▶ メインファイル: `.dsc` (メタデータ)
- ▶ ソースフォーマットのバージョンに依存する他のファイル
 - ▶ 1.0, 3.0 (ネイティブ): `package_version.tar.gz`
 - ▶ 1.0 (非ネイティブ):
 - ▶ `pkg_ver.orig.tar.gz`: 上流ソース
 - ▶ `pkg_debver.diff.gz`: Debian 固有の変更を加えるパッチ
 - ▶ 3.0 (quilt):
 - ▶ `pkg_ver.orig.tar.gz`: 上流ソース
 - ▶ `pkg_debver.debian.tar.gz`: Debian の変更を格納した tarball

(詳細は `dpkg-source(1)` を参照)



ソースパッケージの例 (wget_1.12-2.1.dsc)

```
Format: 3.0 (quilt)
Source: wget
Binary: wget
Architecture: any
Version: 1.12-2.1
Maintainer: Noel Kothé <noel@debian.org>
Homepage: http://www.gnu.org/software/wget/
Standards-Version: 3.8.4
Build-Depends: debhelper (>> 5.0.0), gettext, texinfo,
    libssl-dev (>= 0.9.8), dpatch, info2man
Checksums-Sha1:
    50d4ed2441e67[..]1ee0e94248 2464747 wget_1.12.orig.tar.gz
    d4c1c8bbe431d[..]dd7cef3611 48308 wget_1.12-2.1.debian.tar.gz
Checksums-Sha256:
    7578ed0974e12[..]dcba65b572 2464747 wget_1.12.orig.tar.gz
    1e9b0c4c00eae[..]89c402ad78 48308 wget_1.12-2.1.debian.tar.gz
Files:
    141461b9c04e4[..]9d1f2abf83 2464747 wget_1.12.orig.tar.gz
    e93123c934e3c[..]2f380278c2 48308 wget_1.12-2.1.debian.tar.gz
```

既存のソースパッケージの入手

▶ Debian のアーカイブから:

- ▶ `apt-get source package`
- ▶ `apt-get source package=version`
- ▶ `apt-get source package/release`

(sources.list に deb-src 行が必要)

▶ インターネットから:

- ▶ `dget url-to.dsc`
- ▶ `dget http://snapshot.debian.org/archive/debian-archive/
20090802T004153Z/debian/dists/bo/main/source/web/
wget_1.4.4-6.dsc`

(snapshot.d.o では、2005 年以降の Debian からのすべてのパッケージを提供)

▶ (公開された) バージョン管理システムから:

- ▶ `debcheckout package`

▶ ダウンロードしたら `dpkg-source -x file.dsc` で展開



基本的なソースパッケージの作成

- ▶ 上流ソースのダウンロード
(上流ソース = ソフトウェアのオリジナル開発者からのもの)
- ▶ `<source_package>_<upstream_version>.orig.tar.gz` に名前の変更
(例: `simgrid_3.6.orig.tar.gz`)
- ▶ `tar` を展開
- ▶ ディレクトリーを `<source_package>-<upstream_version>` に変更
(例: `simgrid-3.6`)
- ▶ `cd <source_package>-<upstream_version> && dh_make`
(**dh-make** パッケージに収録)
- ▶ `dh_make` の代わりに特定のパッケージ向けのものも:
dh-make-perl, dh-make-php, ...
- ▶ `debian/` ディレクトリーにたくさんのファイルが作成



debian/ 内のファイル

パッケージングの作業は、すべて debian/ 以下の変更で行う

- ▶ メインのファイル:
 - ▶ **control** – パッケージに関するメタデータ (依存関係 etc.)
 - ▶ **rules** – パッケージの構築方法を記載
 - ▶ **copyright** – パッケージの著作権情報
 - ▶ **changelog** – Debian パッケージの履歴
- ▶ その他のファイル:
 - ▶ compat
 - ▶ watch
 - ▶ dh_install* targets
*.dirs, *.docs, *.manpages, ...
 - ▶ メンテナースクリプト
*.postinst, *.prerm, ...
 - ▶ source/format
 - ▶ patches/ – 上流ソースを変更する必要がある際に使用
- ▶ ファイルのフォーマットは RFC 822 (メールヘッダー) を基にしたものも

debian/changelog

- ▶ Debian パッケージの変更点一覧
- ▶ パッケージの現在のバージョンの見方

1.2.1.1-5

上流 Debian
バージョン リビジョン

- ▶ 手で編集するか **dch** を使用
 - ▶ 新リリースの changelog エントリ作成: **dch -i**
- ▶ Debian や Ubuntu のバグ報告をクローズする特殊フォーマット
Debian: Closes: #595268; Ubuntu: LP: #616929
- ▶ `/usr/share/doc/package/changelog.Debian.gz` にインストール

```
mpich2 (1.2.1.1-5) unstable; urgency=low
```

- ```
* Use /usr/bin/python instead of /usr/bin/python2.5. Allow
to drop dependency on python2.5. Closes: #595268
* Make /usr/bin/mpdroot setuid. This is the default after
the installation of mpich2 from source, too. LP: #616929
+ Add corresponding lintian override.
```

```
-- Lucas Nussbaum <lucas@debian.org> Wed, 15 Sep 2010 18:13:44 +0200
```

# debian/control

- ▶ パッケージのメタデータ
  - ▶ ソースパッケージ向け
  - ▶ このソースから構築される各バイナリーパッケージ向け
- ▶ パッケージ名、セクション、優先度、メンテナー、アップロード担当、構築依存関係、依存関係、説明、ホームページ ...
- ▶ ドキュメント: Debian ポリシー 5 章  
<http://www.debian.org/doc/debian-policy/ch-controlfields>

---

```
Source: wget
Section: web
Priority: important
Maintainer: Noel Kothe <noel@debian.org>
Build-Depends: debhelper (> 5.0.0), gettext, texinfo,
 libssl-dev (>= 0.9.8), dpatch, info2man
Standards-Version: 3.8.4
Homepage: http://www.gnu.org/software/wget/

Package: wget
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: retrieves files from the web
 Wget is a network utility to retrieve files from the Web
```



# Architecture: all か any

## 2 種類のバイナリーパッケージ:

- ▶ Debian のアーキテクチャごとに異なる内容のパッケージ
  - ▶ 例: C プログラム
  - ▶ `debian/control` に `Architecture: any`
    - ▶ または動作するアーキテクチャのみ:  
`Architecture: amd64 i386 ia64 hurd-i386`
  - ▶ `buildd.debian.org`: アップロードした以外の全アーキテクチャを構築
  - ▶ `package_version_architecture.deb` という名前
- ▶ 全アーキテクチャで同じ内容のパッケージ
  - ▶ 例: Perl ライブラリー
  - ▶ `debian/control` に `Architecture: all`
  - ▶ `package_version_all.deb` という名前

ソースパッケージは、`Architecture: any` と `Architecture: all` のバイナリーパッケージが混在しても生成可能



# debian/rules

- ▶ Makefile
- ▶ Debian パッケージを構築するインターフェース
- ▶ Debian ポリシーの 4.8 章に記述  
<http://www.debian.org/doc/debian-policy/ch-source#s-debianrules>
- ▶ 必要なターゲット:
  - ▶ build, build-arch, build-indep: すべての設定とコンパイルを実行
  - ▶ binary, binary-arch, binary-indep: バイナリーパッケージ構築
    - ▶ dpkg-buildpackage は、binary を呼び出して全パッケージの構築、binary-arch を呼び出して Architecture: any パッケージのみの構築
  - ▶ clean: ソースディレクトリーのクリーンナップ



# パッケージングヘルパー – debhelper

- ▶ `debian/rules` に直接シェルのコードを記述可能
  - ▶ `adduser` パッケージを参考に
- ▶ よりよい方法 (多くのパッケージが採用): パッケージングヘルパー 利用
- ▶ 一番人気: **debhelper** (98% のパッケージが採用)
- ▶ 目的:
  - ▶ 全パッケージで使われる標準ツールの共通タスクを分解
  - ▶ パッケージングバグを一度直して全パッケージに適用

`dh_installdirs`, `dh_installchangelogs`, `dh_installdocs`, `dh_installexamples`, `dh_install`,  
`dh_installdebconf`, `dh_installinit`, `dh_link`, `dh_strip`, `dh_compress`, `dh_fixperms`, `dh_perl`,  
`dh_makeshlibs`, `dh_installdeb`, `dh_shlibdeps`, `dh_gencontrol`, `dh_md5sums`, `dh_builddeb`, ...

- ▶ `debian/rules` から呼ばれる
- ▶ コマンドパラメーターや `debian/` のファイルで設定可能

`package.docs`, `package.examples`, `package.install`, `package.manpages`, ...

- ▶ パッケージセット用のサードパーティーヘルパー: **python-support**, **dh\_ocaml**, ...
- ▶ Gotcha: `debian/compat`: Debhelper 互換性バージョン ("7" に)



## debhelper を用いた debian/rules (1/2)

```
#!/usr/bin/make -f
```

```
Uncomment this to turn on verbose mode.
```

```
#export DH_VERBOSE=1
```

```
build:
```

```
$(MAKE)
```

```
#docbook-to-man debian/package.sgml > package.1
```

```
clean:
```

```
dh_testdir
```

```
dh_testroot
```

```
rm -f build-stamp configure-stamp
```

```
$(MAKE) clean
```

```
dh_clean
```

```
install: build
```

```
dh_testdir
```

```
dh_testroot
```

```
dh_clean -k
```

```
dh_installdirs
```

```
Add here commands to install the package into debian/package
```

```
$(MAKE) DESTDIR=$(CURDIR)/debian/package install
```



## debhelper を用いた debian/rules (2/2)

```
Build architecture-independent files here.
```

```
binary-indep: build install
```

```
Build architecture-dependent files here.
```

```
binary-arch: build install
```

```
dh_testdir
```

```
dh_testroot
```

```
dh_installchangelogs
```

```
dh_installdocs
```

```
dh_installexamples
```

```
dh_install
```

```
dh_installman
```

```
dh_link
```

```
dh_strip
```

```
dh_compress
```

```
dh_fixperms
```

```
dh_installdeb
```

```
dh_shlibdeps
```

```
dh_gencontrol
```

```
dh_md5sums
```

```
dh_builddeb
```

```
binary: binary-indep binary-arch
```

```
.PHONY: build clean binary-indep binary-arch binary install configure
```



# CDBS

- ▶ debhelper では、まだ無駄がたくさん
- ▶ 共通機能を分解する第 2 レベルヘルパー
  - ▶ 例: `./configure && make && make install` での構築や CMake での構築
- ▶ CDBS:
  - ▶ 2005 年に *GNU make* マジックを発展させたものをベースに導入
  - ▶ ドキュメント: `/usr/share/doc/cdb/`
  - ▶ Perl, Python, Ruby, GNOME, KDE, Java, Haskell, ... をサポート
  - ▶ でも嫌いな人が:
    - ▶ パッケージ構築のカスタマイズが難しい場合がある:  
"makefile と環境変数の絡みあった迷宮"
    - ▶ 素の debhelper より遅い (無意味な `dh_*` をたくさん呼び出す)

---

```
#!/usr/bin/make -f
include /usr/share/cdb/1/rules/debhelper.mk
include /usr/share/cdb/1/class/autotools.mk
```

```
add an action after the build
build/mypackage::
 /bin/bash debian/scripts/foo.sh
```





# Dh (Debhelper 7, dh7)

- ▶ CDBS キラー として 2008 年に導入
- ▶ `dh_*` を呼び出す **dh** コマンド
- ▶ オーバーライドのみを列挙するシンプルな *debian/rules*
- ▶ CDBS よりもカスタマイズが簡単
- ▶ 文書: man ページ (`debhelper(7)`, `dh(1)`) + DebConf9 talk のスライド  
<http://kitenet.net/~joey/talks/debhelper/debhelper-slides.pdf>

---

```
#!/usr/bin/make -f
%:
 dh $@

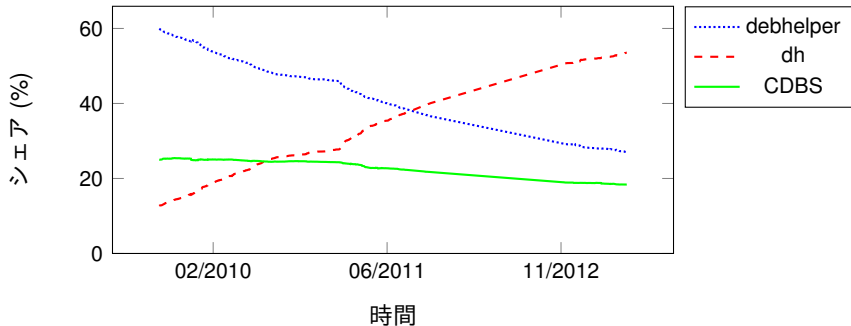
override_dh_auto_configure:
 dh_auto_configure -- --with-kitchen-sink

override_dh_auto_build:
 make world
```



# Classic debhelper vs CDBS vs dh

- ▶ Mind shares:  
Classic debhelper: 27%   CDBS: 18%   dh: 54%
- ▶ どれを学ぶべき?
  - ▶ おそらく少しずつでもすべて
  - ▶ dh や CDBS を使うには debhelper を知る必要
  - ▶ CDBS パッケージを変更するかも
- ▶ 新しいパッケージにはどれを使うべき?
  - ▶ **dh** (これだけマインドシェアが上昇)



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



# パッケージの構築

- ▶ `apt-get build-dep mypackage`  
構築依存関係をインストール (Debian にパッケージあり)  
または `mk-build-deps -ir` (まだアップロードされていないパッケージ)
- ▶ `debuild`: 構築、`lintian` によるテスト、GPG での署名
- ▶ `dpkg-buildpackage` を直接呼び出すのも可能
  - ▶ 通常は `dpkg-buildpackage -us -uc`
- ▶ クリーン & 最小の環境でパッケージを構築するのが良い
  - ▶ `pbuilder` – `chroot` 内でパッケージを構築するヘルパー  
よいドキュメント: <https://wiki.ubuntu.com/PbuilderHowto>  
(最適化: `cowbuilder ccache distcc`)
  - ▶ `schroot` と `sbuid`: Debian 構築デーモンで使用  
(`pbuilder` ほどシンプルではないが LVM スナップショットが取れる  
<https://help.ubuntu.com/community/SbuildLVMHowto> を参照)
- ▶ `.deb` ファイルと `.changes` ファイルを生成
  - ▶ `.changes`: 何を構築したかを説明 (パッケージのアップロードに使用)



# パッケージのインストールとテスト

- ▶ ローカルでパッケージをインストール: `debi` (インストール時の情報に `.changes` を利用)
- ▶ パッケージの内容一覧: `debc` `../mypackage<TAB>.changes`
- ▶ 旧バージョンのパッケージとの比較:  
`debdiff` `../mypackage_1_*.changes` `../mypackage_2_*.changes`  
もしくはソースパッケージの比較:  
`debdiff` `../mypackage_1_*.dsc` `../mypackage_2_*.dsc`
- ▶ `lintian` によるパッケージのチェック (静的解析):  
`lintian` `../mypackage<TAB>.changes`  
`lintian -i`: エラーの詳細情報を表示  
`lintian -EviIL +pedantic`: もっと問題を表示
- ▶ Debian にパッケージをアップロード (`dput`) (要設定)
- ▶ `reprepro` で個人の Debian アーカイブを管理  
ドキュメント: <http://mirrorer.alioth.debian.org/>



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



# 練習問題 1: **grep** パッケージの変更

- 1 `http://ftp.debian.org/debian/pool/main/g/grep/` からバージョン 2.6.3-3 のパッケージをダウンロード (Ubuntu 11.10 以降や Debian testing, unstable を使用している場合、バージョン 2.9-1, 2.9-2 を)
  - ▶ ソースパッケージを自動展開しなければ  
`dpkg-source -x grep_*.dsc` として展開
- 2 `debian/` の中を見よ。
  - ▶ このソースパッケージからの、バイナリーパッケージの生成数は?
  - ▶ このパッケージで利用しているパッケージヘルパーは?
- 3 パッケージを構築せよ
- 4 今度はパッケージの変更をしよう。changelog エントリーを追加し、バージョン番号を増加せよ。
- 5 今度は、perl-regexp サポートを無効にせよ (`./configure` オプション)
- 6 パッケージを再構築せよ
- 7 元のパッケージと新しいものを `debdiff` で比較せよ
- 8 新しく構築したパッケージをインストールせよ
- 9 めちゃくちゃになって泣く ;)



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答





# debian/copyright

- ▶ ソースとパッケージの著作権・ライセンス情報
- ▶ 伝統的にテキストファイル
- ▶ 新しい機械可読フォーマット:

<http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/>

---

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: X Solitaire
Source: ftp://ftp.example.com/pub/games
```

```
Files: *
Copyright: Copyright 1998 John Doe <jdoe@example.com>
License: GPL-2+
This program is free software; you can redistribute it
[...]
.
On Debian systems, the full text of the GNU General Public
License version 2 can be found in the file
'/usr/share/common-licenses/GPL-2'.
```

```
Files: debian/*
Copyright: Copyright 1998 Jane Smith <jsmith@example.net>
License:
[LICENSE TEXT]
```



# 上流ソースの変更

しばしば必要:

- ▶ バグ修正や Debian 特有のカスタマイズを追加
- ▶ 新しい上流リリースからバックポート

いくつか方法あり:

- ▶ 直接ファイルを編集
  - ▶ シンプル
  - ▶ 変更のドキュメントや追跡する方法がない
- ▶ パッチシステム利用
  - ▶ 上流へ変更を送り簡単に貢献
  - ▶ 派生物と変更を共有しやすく
  - ▶ 変更をもっと露出へ

<http://patch-tracker.debian.org/>



# パッチシステム

- ▶ 原則: 変更点は `debian/patches/` にパッチとして格納
- ▶ 適用・非適用は構築時に
- ▶ 過去: 複数の実装 – *simple-patchsys* (*cdb*s), *dpatch*, ***quilt***
  - ▶ それぞれ以下の `debian/rules` ターゲットをサポート:
    - ▶ `debian/rules patch`: 全パッチ適用
    - ▶ `debian/rules unpatch`: 全パッチ非適用
  - ▶ 詳細ドキュメント: <http://wiki.debian.org/debian/patches>
- ▶ 新ソースパッケージフォーマットはパッチシステム内蔵: **3.0 (quilt)**
  - ▶ 推奨解決法
  - ▶ *quilt* を学ぶ必要あり  
<http://pkg-perl.alioth.debian.org/howto/quilt.html>
  - ▶ `devscripts` にパッチシステム非依存ツール: `edit-patch`



# パッチのドキュメント

- ▶ パッチの先頭に標準ヘッダー
- ▶ DEP-3 にドキュメント - Patch Tagging Guidelines  
<http://dep.debian.net/deps/dep3/>

---

```
Description: Fix widget frobnication speeds
 Frobnicating widgets too quickly tended to cause explosions.
Forwarded: http://lists.example.com/2010/03/1234.html
Author: John Doe <johndoe-guest@users.alioth.debian.org>
Applied-Upstream: 1.2, http://bzd.foo.com/frobnicator/revision/123
Last-Update: 2010-03-29
```

```
--- a/src/widgets.c
+++ b/src/widgets.c
@@ -101,9 +101,6 @@ struct {
```



# インストール・削除中に行われること

- ▶ パッケージを伸張するだけでは不十分
- ▶ システムユーザー追加/削除、サービス開始/停止、*alternatives* の管理
- ▶ メンテナースクリプト で実施  
preinst, postinst, prerm, postrm
  - ▶ 共通アクションの一部は debhelper で生成可能
- ▶ ドキュメント:
  - ▶ Debian ポリシーマニュアル 6 章  
<http://www.debian.org/doc/debian-policy/ch-maintainerscripts>
  - ▶ Debian 開発者リファレンス 6.4 章  
<http://www.debian.org/doc/developers-reference/best-pkging-practices.html>
  - ▶ <http://people.debian.org/~srivasta/MaintainerScripts.html>
- ▶ ユーザーの入力
  - ▶ **debconf** で行わなければならない
  - ▶ ドキュメント: debconf-devel(7) (debconf-doc パッケージ)



# 上流バージョンの監視

- ▶ どこを監視するか debian/watch に指定 (uscan(1) 参照)

```
version=3
```

```
http://tmrc.mit.edu/mirror/twisted/Twisted/(\d\.\d)/ \
Twisted-([\d\.]*)\.tar\.bz2
```

- ▶ debian/watch を使用する Debian インフラ:

## **Debian External Health Status**

<http://dehs.alieth.debian.org/>

- ▶ パッケージ追跡システムからメンテナーにemailで警告

<http://packages.qa.debian.org/>

- ▶ uscan: 手動チェック実行

- ▶ uupdate: 最新の上流バージョンにパッケージを更新



# バージョン管理システムでのパッケージング

- ▶ パッケージング作業でブランチやタグの管理補助ツール:  
svn-buildpackage, git-buildpackage
- ▶ 例: git-buildpackage
  - ▶ upstream ブランチは upstream/version タグで上流ソースを追跡
  - ▶ master ブランチは Debian パッケージを追跡
  - ▶ アップロードごとに debian/version タグを打つ
  - ▶ pristine-tar ブランチで上流 tar ボールを再構築
- ▶ debian/control の Vcs-\* フィールドにリポジトリの場所を
  - ▶ `http://wiki.debian.org/Alioth/Git`
  - ▶ `http://wiki.debian.org/Alioth/Svn`

Vcs-Browser: `http://anonscm.debian.org/gitweb/?p=collab-maint/devscripts.git`

Vcs-Git: `git://anonscm.debian.org/collab-maint/devscripts.git`

Vcs-Browser: `http://svn.debian.org/viewsvn/pkg-perl/trunk/libwww-perl/`

Vcs-Svn: `svn://svn.debian.org/pkg-perl/trunk/libwww-perl`

- ▶ VCS 非依存インターフェース: debcheckout, debcommit, debrelease
  - ▶ debcheckout grep → Git からソースパッケージをチェックアウト



# パッケージのバックポート

- ▶ 目的: 旧システム上でパッケージの新バージョンを使用する  
例: *unstable* 由来の *mutt* を *Debian stable* で利用
- ▶ 全体的な考え方:
  - ▶ Debian unstable からソースパッケージ取得
  - ▶ Debian stable で構築・動作するよう修正
    - ▶ 時にたいしたことはない (変更不要)
    - ▶ 時に難しい
    - ▶ 時に不可能 (大量の解決不能な依存関係)
- ▶ Debian プロジェクトで提供・サポートするバックポート  
<http://backports.debian.org/>





# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



# Debian に貢献するさまざまな方法

## ▶ 貢献のよくない方法:

- ① 自分のアプリケーションをパッケージング
- ② Debian を理解した気になる
- ③ いなくなる

## ▶ 貢献のよりましな方法:

- ▶ パッケージングチームに参加
  - ▶ パッケージ群にフォーカスした、たくさんのチーム
  - ▶ <http://wiki.debian.org/Teams> に一覧
  - ▶ 経験豊富な貢献者から学ぶ、優れた方法
- ▶ メンテナンスされていないパッケージ (メンテナー不在パッケージ) の引き取り
- ▶ Debian に新しいソフトウェアを導入
  - ▶ 興味深い/便利なものならぜひ
  - ▶ すでに同じパッケージが Debian にないか?



# メンテナー不在パッケージの引き取り

- ▶ Debian にはメンテナンスされていないパッケージが大量にある
- ▶ 全リスト + 進捗: <http://www.debian.org/devel/wnpp/>
- ▶ 自分のマシンにインストール: `wnpp-alert`
- ▶ それぞれの状態:
  - ▶ **Orphaned** (メンテナー不在): このパッケージはメンテナンスされていない  
気軽に引き取って
  - ▶ **RFA: Request For Adopter** (引き取り求む)  
メンテナーが作業継続困難につき、引き取り手を探している。  
気軽に引き取って。現メンテナーにメールするのが丁寧
  - ▶ **ITA: Intent To Adopt** (引き取り予定)  
誰かがパッケージを引き取ろうとしている  
手伝いを申し込むときに!
  - ▶ **RFH: Request For Help** (助け求む)  
メンテナーが助けを求めている
- ▶ 非メンテナンスパッケージを未検出 → まだメンテナー不在ではない
- ▶ 不明点は [debian-qa@lists.debian.org](mailto:debian-qa@lists.debian.org) や  
[#debian-qa](http://irc.debian.org) で質問



# パッケージの引き取り: 例

From: You <you@yourdomain>  
To: 640454@bugs.debian.org, control@bugs.debian.org  
Cc: Francois Marier <francois@debian.org>  
Subject: ITA: verbiste -- French conjugator

retitle 640454 ITA: verbiste -- French conjugator  
owner 640454 !  
thanks

Hi,

I am using verbiste and I am willing to take care of the package.

Cheers,

You

- ▶ 元メンテナーに丁寧に連絡を (特にまだメンテナー不在ではなく RFA パッケージの時)
- ▶ 上流プロジェクトに連絡するとよい



# Debian に自分のパッケージを提供

- ▶ Debian に自分のパッケージを提供するのに公式ステータスは不要
  - ① reportbug wnpp で **ITP** バグ (Intend To Package パッケージング宣言) を送信
  - ② ソースパッケージの準備
  - ③ パッケージをスポンサーしてくれる Debian 開発者を探す
- ▶ 公式ステータス (経験豊富なパッケージメンテナーの場合):
  - ▶ **Debian Maintainer (DM):**  
自分のパッケージのアップロード権限  
<http://wiki.debian.org/DebianMaintainer> 参照
  - ▶ **Debian Developer (DD):**  
Debian プロジェクトメンバー (投票および任意のパッケージをアップロード)



# スポンサーを探す前にやっておくこと

- ▶ Debian は 品質重視
- ▶ 一般的に スポンサーは忙しく、探すのは大変
  - ▶ スポンサーを探す前に、自分のパッケージは準備万端か確認
- ▶ チェック項目:
  - ▶ 構築依存関係の不備はないか: クリーンな *sid chroot* できちんとパッケージが構築できるか確認
    - ▶ pbuilder の利用を推奨
  - ▶ 自分のパッケージに `lintian -EviIL +pedantic` を実行
    - ▶ エラーは必ず修正。その他の問題も修正すべき
  - ▶ もちろん、詳細なパッケージのテストをしておく
- ▶ 不明点は質問する



# どこで助けを探す？

## 必要とする助け

- ▶ 疑問に対する助言や回答、コードレビュー
- ▶ パッケージの準備ができたらスポンサーにアップロードしてもらう

## 助けはここから:

- ▶ パッケージングチームの他のメンバー
  - ▶ チーム一覧: <http://wiki.debian.org/Teams>
- ▶ **Debian** メンターグループ (パッケージがチームに合わない場合)
  - ▶ <http://wiki.debian.org/DebianMentorsFaq>
  - ▶ メーリングリスト: [debian-mentors@lists.debian.org](mailto:debian-mentors@lists.debian.org)  
(偶然学ぶにもいい方法)
  - ▶ IRC: [irc.debian.org](http://irc.debian.org) の #debian-mentors
  - ▶ <http://mentors.debian.net/>
  - ▶ ドキュメント: <http://mentors.debian.net/intro-maintainers>
- ▶ 地域化メーリングリスト (自分の言語で助けを求む)
  - ▶ [debian-devel@lists.debian.org](mailto:debian-devel@lists.debian.org)
  - ▶ 全メーリングリスト: <https://lists.debian.org/devel.html>
  - ▶ ユーザーのメーリングリスト:  
<https://lists.debian.org/users.html>



## さらなるドキュメント

- ▶ Debian 開発者のコーナー  
<http://www.debian.org/devel/>  
Debian 開発向けリソースへのたくさんのリンク
- ▶ Debian 新メンテナーガイド  
<http://www.debian.org/doc/maint-guide/>  
Debian パッケージングの導入。更新にも
- ▶ Debian 開発者リファレンス  
<http://www.debian.org/doc/developers-reference/>  
ほとんどが Debian の方法論。パッケージングのベストプラクティスあり (6 章)
- ▶ Debian ポリシー  
<http://www.debian.org/doc/debian-policy/>
  - ▶ すべてのパッケージが満たすべき要件
  - ▶ Perl, Java, Python, ... の具体的なポリシー
- ▶ Ubuntu パッケージングガイド  
<http://developer.ubuntu.com/resources/tools/packaging/>





# メンテナー向け **Debian** ダッシュボード

- ▶ ソースパッケージ中枢: パッケージ追跡システム (PTS)  
<http://packages.qa.debian.org/dpkg>
- ▶ メンテナー/チームの中枢: Developer's Packages Overview (DDPO)  
<http://qa.debian.org/developer.php?login=pkg-ruby-extras-maintainers@lists.alioth.debian.org>
- ▶ **TODO** リスト指向: Debian Maintainer Dashboard (DMD)  
<http://udd.debian.org/dmd.cgi>



# バグ追跡システム (BTS) の利用

- ▶ バグを管理する唯一の方法
  - ▶ バグを見る Web インターフェース
  - ▶ バグを変更する Email インターフェース
- ▶ バグに情報を付加:
  - ▶ 123456@bugs.debian.org に送信 (送信者を含まない。含める場合は 123456-submitter@bugs.debian.org を追加)
- ▶ バグの状態変更:
  - ▶ control@bugs.debian.org にコマンド送信
  - ▶ コマンドラインインターフェース: devscripts の bts コマンド
  - ▶ ドキュメント: <http://www.debian.org/Bugs/server-control>
- ▶ バグの報告: reportbug を利用
  - ▶ ローカルメールサーバー使用: ssmtp や nullmailer をインストール
  - ▶ または reportbug --template を使用し submit@bugs.debian.org へ (手動で) 送信



## BTS の利用例:

- ▶ バグと送信者へメール送信:

`http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=680822#10`

- ▶ タグ付けと重大度変更:

`http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=680227#10`

- ▶ 再割当て、重大度変更、タイトル変更...:

`http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=680822#93`

- ▶ notfound, found, notfixed, fixed は バージョン追跡 される

`https://wiki.debian.org/HowtoUseBTS#Version_tracking` 参照

- ▶ ユーザータグの利用:

`http://bugs.debian.org/cgi-bin/bugreport.cgi?msg=42;bug=642267`

`https://wiki.debian.org/bugs.debian.org/usertags` 参照

- ▶ BTS のドキュメント:

- ▶ `http://www.debian.org/Bugs/`

- ▶ `https://wiki.debian.org/HowtoUseBTS`



# Ubuntu の方が興味ある?

- ▶ Ubuntu では主に、Debian から分岐して管理
- ▶ 特定のパッケージに注目しているわけではないが、Debian チームと協力
- ▶ 通常はまず、Debian への新しいパッケージのアップロードを推奨  
<https://wiki.ubuntu.com/UbuntuDevelopment/NewPackages>
- ▶ おそらくもっと良い案:
  - ▶ Debian チームに参加し Ubuntu との橋渡し
  - ▶ 差異を縮小し Launchpad のバグの処理順を決める手伝い
  - ▶ Debian のツールの多くが助けに:
    - ▶ Developer' s Packages Overview のUbuntu 列
    - ▶ パッケージ追跡システムの Ubuntu ボックス
    - ▶ PTS 経由での launchpad バグメール受信



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



## まとめ

- ▶ Debian のパッケージングについて全体を見渡した
- ▶ しかしもっと詳細なドキュメントが必要になる
- ▶ ベストプラクティスは長年にわたって発展
  - ▶ よくわからなければ **dh** パッケージングヘルパーと **3.0 (quilt)** フォーマットを使う
- ▶ このチュートリアルで扱わなかったこと:
  - ▶ UCF – ユーザーが変更した設定ファイルのアップグレード時の扱い
  - ▶ dpkg トリガー – メンテナースクリプトに似たグループを同時に実行
  - ▶ Debian の開発組織:
    - ▶ パッケージ群: stable, testing, unstable, experimental, security, \*-updates, backports, ...
    - ▶ Debian ブレンド – 特定グループ向けの Debian サブセット

フィードバック: [packaging-tutorial@packages.debian.org](mailto:packaging-tutorial@packages.debian.org) 

# 法的事項

Copyright ©2011–2013 Lucas Nussbaum – [lucas@debian.org](mailto:lucas@debian.org)

**This document is free software:** you can redistribute it and/or modify it under either (at your option):

- ▶ The terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.  
<http://www.gnu.org/licenses/gpl.html>
- ▶ The terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License.  
<http://creativecommons.org/licenses/by-sa/3.0/>



# このチュートリアルへの貢献

## ▶ 貢献:

- ▶ `apt-get source packaging-tutorial`
- ▶ `debcheckout packaging-tutorial`
- ▶ `git clone`  
`git://git.debian.org/collab-maint/packaging-tutorial.git`
- ▶ `http://git.debian.org/?p=collab-maint/packaging-tutorial.git`
- ▶ 未修正バグ: `bugs.debian.org/src:packaging-tutorial`

## ▶ フィードバックの送り先:

- ▶ `mailto:packaging-tutorial@packages.debian.org`
  - ▶ このチュートリアルに何を追加すべき?
  - ▶ もっと良くするには?
- ▶ `reportbug packaging-tutorial`





# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



## 練習問題 2: GNUjump のパッケージング

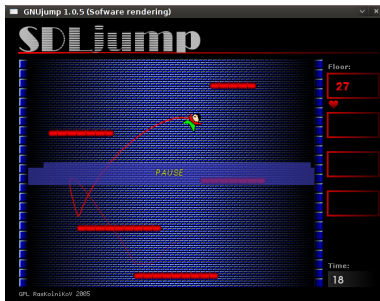
### ① GNUjump 1.0.8 を

<http://ftp.gnu.org/gnu/gnujump/gnujump-1.0.8.tar.gz> からダウンロードせよ

### ② この Debian パッケージを作成せよ

- ▶ パッケージを構築するため構築依存関係パッケージをインストール
- ▶ パッケージの基本作業を確認
- ▶ debian/control や他のファイルに記入して完成

### ③ 楽しむこと



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



## 練習問題 3: Java ライブラリーのパッケージング

① Java のパッケージングについてのドキュメントを参照せよ:

- ▶ <http://wiki.debian.org/Java>
- ▶ <http://wiki.debian.org/Java/Packaging>
- ▶ <http://www.debian.org/doc/packaging-manuals/java-policy/>
- ▶ <http://pkg-java.alioth.debian.org/docs/tutorial.html>
- ▶ javahelper について Debconf10 で語った資料やスライド:  
<http://pkg-java.alioth.debian.org/docs/debconf10-javahelper-paper.pdf>  
<http://pkg-java.alioth.debian.org/docs/debconf10-javahelper-slides.pdf>

② <http://moepii.sourceforge.net/> から IRCLib をダウンロードせよ

③ パッケージを作成せよ



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



## 練習問題 4: Ruby gem のパッケージング

---

- 1 Ruby のパッケージングについてのドキュメントを参照せよ:
  - ▶ <http://wiki.debian.org/Ruby>
  - ▶ <http://wiki.debian.org/Teams/Ruby>
  - ▶ <http://wiki.debian.org/Teams/Ruby/Packaging>
  - ▶ `gem2deb(1)`, `dh_ruby(1)` (`gem2deb` パッケージ内)
- 2 `net-ssh gem` から基本的な Debian ソースパッケージを作成せよ:  
`gem2deb net-ssh`
- 3 きちんとした Debian パッケージになるよう改良せよ



# アウトライン

- ① はじめに
- ② ソースパッケージの作成
- ③ パッケージの構築とテスト
- ④ 練習問題 1: grep パッケージの変更
- ⑤ 高度なパッケージングの話題
- ⑥ Debian でのパッケージメンテナンス
- ⑦ まとめ
- ⑧ 練習問題 2: GNUjump のパッケージング
- ⑨ 練習問題 3: Java ライブラリーのパッケージング
- ⑩ 練習問題 4: Ruby gem のパッケージング
- ⑪ 練習問題の解答



# 解答

## 練習問題





# 練習問題 1: **grep** パッケージの変更

- 1 `http://ftp.debian.org/debian/pool/main/g/grep/` からバージョン 2.6.3-3 のパッケージをダウンロード (Ubuntu 11.10 以降や Debian testing, unstable を使用している場合、バージョン 2.9-1, 2.9-2 を)
- 2 `debian/` の中を見よ。
  - ▶ このソースパッケージからの、バイナリーパッケージの生成数は?
  - ▶ このパッケージで利用しているパッケージヘルパーは?
- 3 パッケージを構築せよ
- 4 今度はパッケージの変更をしよう。changelog エントリーを追加し、バージョン番号を増加せよ。
- 5 今度は、perl-regex サポートを無効にせよ (`./configure` オプション)
- 6 パッケージを再構築せよ
- 7 元のパッケージと新しいものを `debdiff` で比較せよ
- 8 新しく構築したパッケージをインストールせよ
- 9 めちゃくちゃになって泣く ;)



## ソースの取得

- ① `http://ftp.debian.org/debian/pool/main/g/grep/` に行き、バージョン 2.6.3-3 のパッケージをダウンロードする
- ▶ `dget` を使用して `.dsc` ファイルをダウンロード:  
`dget http://cdn.debian.net/debian/pool/main/g/grep/grep_2.6.3-3.dsc`
- ▶ `http://packages.qa.debian.org/grep` によると、`grep` バージョン 2.6.3-3 は現在 *stable* (*squeeze*)。 `/etc/apt/sources.list` に *squeeze* の `deb-src` 行がある場合、以下のようにできます。  
`apt-get source grep=2.6.3-3`  
または `apt-get source grep/stable`  
または `apt-get source grep` とできればラッキー
- ▶ `grep` のソースパッケージは以下の 3 ファイル:
  - ▶ `grep_2.6.3-3.dsc`
  - ▶ `grep_2.6.3-3.debian.tar.bz2`
  - ▶ `grep_2.6.3.orig.tar.bz2`典型的な "3.0 (quilt)" フォーマット
- ▶ 必要なら以下のようにソースを展開  
`dpkg-source -x grep_2.6.3-3.dsc`



# パッケージを見回して構築

## ② debian/の中を見よ。

- ▶ このソースパッケージからの、バイナリーパッケージの生成数は？
- ▶ このパッケージで利用しているパッケージヘルパーは？
- ▶ `debian/control` によると、このパッケージは `grep` という名前のバイナリーパッケージをひとつだけ生成する。
- ▶ `debian/rules` によると、このパッケージは *CDBS* や *dh* を使わず、*classic debhelper* でパッケージングされている。`debian/rules` で、さまざまな `dh_*` コマンドを呼び出していることがわかる。

## ③ パッケージを構築せよ

- ▶ `apt-get build-dep grep` を使用して、構築依存のパッケージを取得
- ▶ その後 `debuild` や `dpkg-buildpackage -us -uc` を実行 (1 分ほどかかる)



## changelog の編集

- ④ 今度はパッケージの変更をしよう。changelog エントリーを追加し、バージョン番号を増加せよ。
  - ▶ debian/changelog はテキストファイルである。手で編集して新エントリーを追加する。
  - ▶ また、`dch -i` を使用し、エントリーを追加しエディターを起動
  - ▶ 名前とメールアドレスは環境変数 `DEBFULLNAME` と `DEBEMAIL` で定義
  - ▶ その後、パッケージを再構築: 新バージョンのパッケージを構築
  - ▶ パッケージのバージョン付けは Debian ポリシーの 5.6.12 節に <http://www.debian.org/doc/debian-policy/ch-controlfields>



# Perl 正規表現の無効化と再構築

- ⑤ 今度は、perl-regex サポートを無効にせよ (./configure オプション)
  - ⑥ パッケージを再構築せよ
- ▶ ./configure --help をチェック: Perl 正規表現を無効にするオプションは --disable-perl-regex
  - ▶ debian/rules を編集して ./configure の行を探す
  - ▶ --disable-perl-regex を追加
  - ▶ debuild や dpkg-buildpackage -us -uc で再構築



# パッケージの比較とテスト

- ⑦ 元のパッケージと新しいものを debdiff で比較せよ
- ⑧ 新しく構築したパッケージをインストールせよ

- ▶ バイナリーパッケージの比較: `debdiff ../changes`
- ▶ ソースパッケージの比較: `debdiff ../dsc`
- ▶ 新規構築パッケージをインストール: `debi`  
または `dpkg -i ../grep_<TAB>`
- ▶ `grep -P foo` がもう動作しない!

- ⑨ めちゃくちゃになって泣く ;)

ではなく: 以前のバージョンのパッケージを再インストール:

- ▶ `apt-get install --reinstall grep=2.6.3-3 (= 前バージョン)`



## 練習問題 2: GNUjump のパッケージング

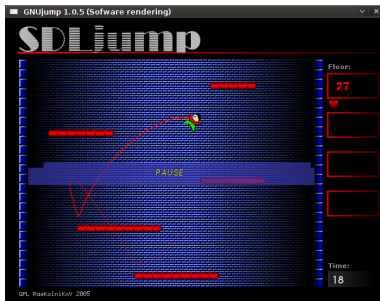
### ① GNUjump 1.0.8 を

<http://ftp.gnu.org/gnu/gnujump/gnujump-1.0.8.tar.gz> からダウンロードせよ

### ② この Debian パッケージを作成せよ

- ▶ パッケージを構築するため構築依存関係パッケージをインストール
- ▶ パッケージの基本作業を確認
- ▶ debian/control や他のファイルに記入して完成

### ③ 楽しむこと



## 一歩ずつ...

- ▶ `wget http://ftp.gnu.org/gnu/gnujump/gnujump-1.0.8.tar.gz`
- ▶ `mv gnujump-1.0.8.tar.gz gnujump_1.0.8.orig.tar.gz`
- ▶ `tar xf gnujump_1.0.8.orig.tar.gz`
- ▶ `cd gnujump-1.0.8/`
- ▶ `dh_make`
  - ▶ パッケージのタイプ: 単一バイナリー (今回は)

```
gnujump-1.0.8$ ls debian/
changelog gnujump.default.ex preinst.ex
compat gnujump.doc-base.EX prerm.ex
control init.d.ex README.Debian
copyright manpage.1.ex README.source
docs manpage.sgml.ex rules
emacsen-install.ex manpage.xml.ex source
emacsen-remove.ex menu.ex watch.ex
emacsen-startup.ex postinst.ex
gnujump.cron.d.ex postrm.ex
```





## 一歩ずつ...(2)

- ▶ `debian/changelog`, `debian/rules`, `debian/control` を見る (**dh\_make** が自動記入)
- ▶ `debian/control` では:  
Build-Depends: `debhelper (>= 7.0.50 )`, `autotools-dev`  
構築依存関係 = パッケージを構築するのに必要なパッケージの一覧
- ▶ そのままパッケージの構築を試みる (**dh**マジックに感謝)
  - ▶ 構築できるまで構築依存関係を追加
  - ▶ ヒント: `apt-cache search` や `apt-file` を使ってパッケージを探す
  - ▶ 例:

```
checking for sdl-config... no
checking for SDL - version >= 1.2.0... no
[...]
configure: error: *** SDL version 1.2.0 not found!
```

→ **libsdl1.2-dev** を Build-Depends に追加しインストールする。

- ▶ ベター: **pbuilder** を使ってクリーンな環境で構築



## 一歩ずつ...(3)

- ▶ `libSDL1.2-dev`, `libSDL-image1.2-dev`, `libSDL-mixer1.2-dev` をインストールすると構築成功する。
  - ▶ `debc` を使い、生成したパッケージの内容一覧を取得。
  - ▶ `debi` を使用し、インストール・テストを行う。
  - ▶ `lintian` でパッケージのテスト
    - ▶ 厳格な必要条件ではないが、Debian にアップロードするパッケージは *lintian-clean* を推奨
    - ▶ `lintian -EviIL +pedantic` を使用してもっと問題を列挙できる
    - ▶ ヒント:
      - ▶ `debian/` にある不要なファイルを削除
      - ▶ `debian/control` に記入
      - ▶ `dh_auto_configure` を上書きし、実行ファイルを `/usr/games` にインストール
      - ▶ ハードニングコンパイラフラグを使ってセキュリティを高める。
- <http://wiki.debian.org/Hardening> 参照



## 一歩ずつ...(4)

- ▶ Debian でパッケージ化されているものと、自分のパッケージを比較:
  - ▶ データファイルを、第 2 のパッケージへ分割し、全アーキテクチャで同じ物にしている (→ Debian アーカイブの使用量を抑える)
  - ▶ .desktop ファイル (GNOME/KDE メニュー向け) をインストールし、Debian メニューに統合もしている
  - ▶ パッチを使用し、小さな問題を修正している



## 練習問題 3: Java ライブラリーのパッケージング

① Java のパッケージングについてのドキュメントを参照せよ:

- ▶ <http://wiki.debian.org/Java>
- ▶ <http://wiki.debian.org/Java/Packaging>
- ▶ <http://www.debian.org/doc/packaging-manuals/java-policy/>
- ▶ <http://pkg-java.alioth.debian.org/docs/tutorial.html>
- ▶ javahelper について Debconf10 で語った資料やスライド:  
<http://pkg-java.alioth.debian.org/docs/debconf10-javahelper-paper.pdf>  
<http://pkg-java.alioth.debian.org/docs/debconf10-javahelper-slides.pdf>

② <http://moepii.sourceforge.net/> から IRCLib をダウンロードせよ

③ パッケージを作成せよ



## 一歩ずつ...

- ▶ `apt-get install javahelper`
- ▶ 基本的なソースパッケージを作成: `jh_makepkg`
  - ▶ ライブラリー
  - ▶ なし
  - ▶ デフォルトのフリーなコンパイラ/ランタイム
- ▶ `debian/` の中を見て修正
- ▶ `dpkg-buildpackage -us -uc` または `debuild`
- ▶ `lintian`, `debc`, `etc.`
- ▶ 自分の結果と `libirclib-java` ソースパッケージを比較



## 練習問題 4: Ruby gem のパッケージング

---

- 1 Ruby のパッケージングについてのドキュメントを参照せよ:
  - ▶ <http://wiki.debian.org/Ruby>
  - ▶ <http://wiki.debian.org/Teams/Ruby>
  - ▶ <http://wiki.debian.org/Teams/Ruby/Packaging>
  - ▶ `gem2deb(1)`, `dh_ruby(1)` (`gem2deb` パッケージ内)
- 2 `net-ssh gem` から基本的な Debian ソースパッケージを作成せよ:  
`gem2deb net-ssh`
- 3 きちんとした Debian パッケージになるよう改良せよ



## 一歩ずつ...

gem2deb net-ssh:

- ▶ rubygems.org から gem をダウンロード
- ▶ ひと揃いの .orig.tar.gz アーカイブを作成し、tar を展開
- ▶ gem のメタデータを基に Debian ソースパッケージを初期化
  - ▶ ruby-gemname という名前
- ▶ Debian バイナリーパッケージの生成を試みる (多分失敗)

dh\_ruby (gem2deb に同梱) は Ruby 特有のタスク:

- ▶ C の拡張を各 Ruby バージョン向けに構築
- ▶ 宛先ディレクトリーにファイルをコピー
- ▶ 実行スクリプトのシェバングを更新
- ▶ その他のチェックと同様に debian/ruby-tests.rb や  
debian/ruby-test-files.yaml で定義されたテストを実行



## 一歩ずつ...(2)

生成したパッケージを改良:

- ▶ `debclean` を実行してソースツリーを掃除。 `debian/` を見る。
- ▶ `changelog` や `compat` が正しいか
- ▶ `debian/control` を編集: Homepage をアンコメント。 Description を改良
- ▶ 上流ファイルを基に `copyright` ファイルを適切に記述
- ▶ `ruby-net-ssh.docs: README.rdoc` をインストール
- ▶ `ruby-tests.rb`: テストの実行。 この場合以下で十分:  

```
$: << 'test' << 'lib' << '.'
require 'test/test_all.rb'
```





## 一歩ずつ...(3)

パッケージを構築する。失敗するはず。2つ問題がある:

- ▶ テストスイートで `gem` の呼び出しを無効にする必要がある。  
`test/common.rb` で `gem "test-unit"` の行を削除:
  - ▶ `edit-patch disable-gem.patch`
  - ▶ `test/common.rb` を編集して `gem` 行を削除。その後サブシェル終了
  - ▶ `debian/changelog` の変更点を記述
  - ▶ `debian/patches/disable-gem.patch` のパッチについて記載
- ▶ パッケージの構築依存関係にテストスイートで使用する `ruby-mocha` が欠けている (問題を再度見るためには、`pbuilder` を使用し、クリーン環境でパッケージを構築する必要がある)
  - ▶ パッケージの `Build-Depends` に `ruby-mocha` を追加
  - ▶ `gem2deb` は、`gem` に書かれた依存関係を、`debian/control` にコメントとしてコピーする。しかし、`mocha` が `gem` の開発依存関係にリストされていない (`gem` のバグ)

Debian アーカイブの `ruby-net-ssh` パッケージと、自分のパッケージを比較



このチュートリアルは倉澤望が日本語訳しました。  
この翻訳についての意見・要望は  
<debian-doc@debian.or.jp> にお知らせください。

