2008年冬号

あんどきゅめんてっどでびあん

東京エリアDebian勉強会
関西エリアDebian勉強会著

デビアン勉強会

目次

1	2008 年度 Debian 勉強会企画	2
2	パッケージ作成 (debhelper、CDBS、dpatch、quilt を用いたパッケージ作成)	3
3	Linux カーネルパッチ の Debian パッケージ作成	17
4	Linux カーネルモジュールの Debian パッケージ作成	23
5	みんなも Debian GNU/Hurd を使おうよ！	31
6	Debconf 8	33
7	Debian 温泉	35
8	Po4a でドキュメント翻訳の保守を楽にしよう	40
9	【でびあん】Debian パッケージメンテナというお仕事【現在募集中】	51
10	Debconf8 参加報告	56
11	「その場で勉強会資料を作成しちゃえ」 Debian を使った LaTeX 原稿作成合宿	58
12	Debian を Windows な PC でも楽しもう –応用編	64
13	Debian Live に Ubiquity を移植できるか？	73
14	10 分でわかる Debian フリーソフトウェアガイドライン (DFSG)	76
15	はじめての CDBS	78
16	cdn.debian.or.jp, cdn.debian.net における取り組み	85
17	Debian Trivia Quiz	96
18	Debian Trivia Quiz 問題回答	99
19	索引	100

1　2008 年度 Debian 勉強会企画

上川 純一

2007 年 12 月に実施した 35 回 Debian 勉強会で、一年分の実施したい内容を出してみました。そこで策定した計画は以下です。

1. 新年会「気合を入れる」
2. Open Source Conference Tokyo (3/1)
3. データだけのパッケージを作成してみる、ライセンスの考え方
4. バイナリ一つのパッケージを作成してみるバージョン管理ツールを使い Debian パッケージを管理する (git) アップストリームの扱い (svn/git/cvs)
5. バイナリの分かれたパッケージの作成。バイナリの分け方の考え方、アップグレードなどの運用とか。
6. パッケージ作成 (dpatch/debhelper で作成するパッケージ) man の書き方 (roff or docbook)
7. パッケージ作成 (kernel patch、kernel module) 、Debconf 発表練習
8. Debconf アルゼンチン、共有ライブラリパッケージ作成
9. Open Source Conference Tokyo/Fall、デーモン系のパッケージの作成、latex、 emacs-lisp、フォントパッケージ
10. パッケージの cross-compile の方法、amd64 上で i386 のパッケージとか、OSC-Fall 報告会、Debconf 報告会
11. 国際化 po-debconf / po 化 / DDTP
12. 忘年会

2 パッケージ作成 (debhelper、CDBS、dpatch、quilt を用いたパッケージ作成)

小林儀匡

2.1 はじめに

この文書では、まず Debian パッケージのビルドの流れと debian/rules の各ターゲットの役割を概観した上で、debhelper や CDBS を用いて debian/rules をメンテナンスしやすくする方法を説明します。次に、開発元のソースコードに対する変更をメンテナンスしやすいかたちで加える方法として、パッチ管理ツールである dpatch と quilt の使い方を説明します。

一般的な Debian パッケージの作成方法に興味のある読者を対象としています。

2.2 前回までのおさらい

これまでにパッケージの作成について学んだことを整理してみましょう。簡単にまとめてしまえば、以下のようになるのではないでしょうか。

- deb パッケージの作成には、ソースツリーに debian/changelog、debian/control、debian/copyright、debian/rules という 4 つのファイルが最低限必要。
  - debian/changelog　ソースファイルの変更履歴。最初のエントリが現在のバージョンに関する記述となり、ビルドされるパッケージのバージョン情報もここから抽出される。
  - debian/copyright　著作権・ライセンス情報を収めたファイル。バイナリパッケージの/usr/share/doc/< **バイナリパッケージ名** >/copyright にインストールされる。現在のところ、debian ディレクトリにありながら機械的に処理されることのない珍しいファイルで、書式も厳格には決められていない[*1]。
  - debian/rules　ビルド方法を記述した GNU Make makefile。
  - debian/control　パッケージ (ソースパッケージおよびバイナリパッケージ) のメタ情報を指定するファイル。バージョン番号や作成日時など、バージョンごとに異なる情報は debian/changelog に書かれるため、debian/control には変更頻度の比較的低いメタ情報のみが収められる。また、生成されるソースパッケージとすべてのバイナリパッケージの名前もここで指定される。ファイルは空行で複数の段落に区切られており、そのうち最初の段落がソースパッケージおよびすべてのバイナリパッケージ用、その後に続く 1 つ以上の段落が各バイナリパッケージ用である。
- dh_make を使うと、これらのファイルや、ビルドに用いられるその他の設定ファイルの雛形を作成してくれる。

[*1] 書式を定めて機械可読にする議論がなされている。機械可読になった場合は、パッケージマネージャでライセンス情報を扱えるようになると考えられる。詳しくは http://wiki.debian.org/Proposals/CopyrightFormat を参照のこと。

それらを適当に編集して debuild を実行するとソースパッケージおよびバイナリパッケージを作成できる。

一通りのパッケージ作成の流れや、`debian/rules` 以外のファイルの内容は何となく分かったかと思います。しかし、debhelper のコマンドが連ねられている `debian/rules` については、実際のところどのようなことをしており、どのような流れでパッケージが作られているのかは、おそらくまだ理解できていないと思います。また、パッケージをさらに細かく設定し、より洗練されたものにする方法も分からないでしょう。

そこで、今回は、`debian/rules` と debhelper の各コマンドの内容を説明し、パッケージがどのような過程を経てビルドされているのかを明らかにします。その上で、debhelper や CDBS を用いて、Debian Policy Manual (*debian-policy*) に準拠したパッケージをメンテナンスしやすいかたちで作成する方法を説明します。また、開発元のソースコードに対する変更をメンテナンスしやすいかたちで加える方法として、パッチ管理ツールである dpatch と quilt の使い方を説明します。

まずビルドの流れについて見ていくことから始めましょう。

2.3 deb パッケージのビルド手順

debuild などのパッケージビルドツールでは、大まかに言うと、一般的に次のような手順でパッケージをビルドします。

1. ビルド環境を整備する。
2. 不要なファイルを削除する (`debian/rules clean`)。
3. ソースパッケージをまとめる (`dpkg-source -b` < **ディレクトリ名** >)。
4. バイナリパッケージにインストールするファイルをビルドする (`debian/rules build`)。
5. ビルドしたファイルをバイナリパッケージにまとめる (`debian/rules binary`)。
6. .changes ファイルを作成する (`dpkg-genchanges`)。
7. パッケージに署名する。

以下でこれらを詳しく説明します。

2.3.1 ビルド環境を整備する

実際にビルドを始める前に、まずはビルドのための環境を整える必要があります。

「ビルドのための環境を整える」と一口に言っても色々とありますが、例えばソースパッケージの展開などが挙げられます。ソースパッケージをビルドする場合は、ソースパッケージを展開してソースツリーの状態にするところから始めなければなりません。もちろんソースツリーでビルドする場合はこれは不要です。

また、パッケージのビルドには、通常、様々なもの (コンパイラやライブラリなど) が必要となるので、ビルド中にエラーにならないよう、それらの存在を確認しておく必要もあります。必要となるパッケージは、ソースパッケージの場合は `.dsc` ファイルの Build-Depends フィールド、ソースツリーの場合は debian/control の Build-Depends フィールドに書かれています。ビルドツールによっては、ビルドに必要なパッケージを確認するだけでなく、インストールされていない場合にインストールしてくれるものもあります。

また、pdebuild などのように chroot 環境内でパッケージをビルドするツールは、こういった作業の前にまず chroot 環境を作ってそこに入るところから始めるでしょう。

2.3.2 不要なファイルを削除する (`debian/rules clean`)

必要なパッケージが揃っていることを確認したところで、不要なファイルを削除します。一般に、以前のビルドで生成されたファイルがある場合はそれを削除して、常に同じ状態からビルドできるようにすべきです。`debian/rules` の clean ターゲットをそのような目的で使うよう、debian-policy において定められています。

2.3.3 ソースパッケージをまとめる (dpkg-source -b < ディレクトリ名 >)

ソースパッケージを作成するタイミングとしては、不要なファイルを削除した後、バイナリパッケージに含めるファイルのビルドに入る前が最もよいでしょう。dpkg-source コマンドの-b オプションを使うと、ソースツリーからソースパッケージを作成できます。

2.3.4 バイナリパッケージにインストールするファイルをビルドする (debian/rules build)

ソースパッケージを作成し終えたらいよいよバイナリパッケージの作成に移ります。バイナリパッケージの作成は大きく 2 段階に分けることができます。最初の段階は、設定やコンパイルです。

C などの言語で書かれたプログラムやライブラリがパッケージに含まれている場合、それらをインストール前にコンパイルして、バイナリのプログラムやライブラリを作成する必要があります。プログラムのビルドに GNU Autoconf を使用するようになっている場合は、コンパイルの前に configure スクリプトを走らせて設定を行う必要もあるでしょう。

この手続きは、バイナリのプログラムやライブラリを含んでいないパッケージについても必要になることが多いでしょう。例えば、LaTeX や SGML などの形式で書かれたドキュメントは、HTML や PostScript、PDF などの配布に適した形式に変換してバイナリパッケージに含めるべきです。また、ソースとなるデータを変換してインストール用のデータを作成する必要がある場合も、通常はここでその変換を行います。

debian-policy では、このような、プログラムやライブラリの設定・コンパイルやデータの変換のために、debian/rules の build ターゲットを使うよう指定されています。

2.3.5 ビルドしたファイルをバイナリパッケージにまとめる (debian/rules binary)

必要なファイルをすべてビルドしたところで、それらを適切なパーミッションで適切な場所に配置し、バイナリパッケージにまとめ上げる必要があります。あっさりと書いてしまいましたが、debian-policy に準拠するパッケージを手で作成しようとする場合には、かなり複雑で面倒な作業を要求されるプロセスです。

このプロセスは、通常、まず debian/tmp を/と見なしてソフトウェア全体のインストール (「仮インストール」) を行い、その上で debian/tmp 内の各ファイルを適切に debian/< バイナリパッケージ名 > に振り分け、最後に debian/< バイナリパッケージ名 > をそれぞれバイナリパッケージ化する、という流れで行います。debian/< バイナリパッケージ名 > をバイナリパッケージ化する際には、パーミッションの調節やファイルの圧縮など、しなければならないこと、推奨されていることが多数あります。それらは後で詳述しますので、ここでは詳しい説明は省きます。

debian-policy では、バイナリパッケージをまとめ上げるために、debian/rules の binary ターゲットを使うよう指定されています。

2.3.6 .changes ファイルを作成する (dpkg-genchanges)

ソースパッケージとバイナリパッケージのファイルが一通り揃ったところで、これらのファイルに関する情報をまとめた.changes ファイルを作成する必要があります。これには dpkg-genchanges コマンドが使用されます。

2.3.7 パッケージに署名する

私家版パッケージやテストビルドでは必要ありませんが、公式パッケージにする場合は、最後にパッケージに署名する必要があります。公式パッケージにしない場合でも、広く配布する場合には署名することをお勧めします。

署名は、.dsc ファイルと.changes ファイルに対して行います。.dsc ファイルにはソースパッケージの.tar.gz ファイルと.diff.gz ファイルのハッシュとサイズが書かれているので、署名を施すことでこれらのファイルの品質を保証できます。また、.changes ファイルにはソースパッケージとバイナリパッケージのすべてのファイルのハッシュとサイズが書かれているので、署名を施すことでこれらのファイルすべての品質を保証できます[*2]。

[*2] 厳密に言えば、以前のバージョンと同一の.tar.gz ファイルを使用している場合は、.tar.gz ファイルの情報は.changes ファイルには記載されません。したがって、この場合は.dsc ファイルを経由した間接的な保証となります。

署名には、通常、devscripts パッケージに含まれている debsign コマンドを使います。このコマンドは、最初に .dsc ファイルに対する署名を行い、その後で、.changes 内の.dsc ファイルのエントリのハッシュやサイズを署名後の値に置換した上で、.changes ファイルに署名してくれます。

2.4 debian/rules のターゲット

ビルド手順の説明から分かるかと思いますが、debian/rules には、パッケージのビルド時に必要となるターゲットがあります。説明に登場した clean、build、binary の他に、binary-arch と binary-indep も必須のターゲットです。以下でこれらのターゲットの役割を簡単に示します (詳細は debian-policy を参照してください)。

clean　build ターゲットや binary ターゲットで行った変更をすべて元に戻すためのターゲットです。ただし、生成されたバイナリパッケージの削除はすべきでありません。このターゲットは root 権限で呼び出す必要があるかもしれません。

build　バイナリパッケージに含めるプログラムやライブラリの設定・コンパイルやデータの変換に使用されるターゲットです。設定が対話的だとパッケージの自動ビルドができなくなるため、設定は非対話的なものでなければなりません。インストールパスなどの設定を対話的に行うようになっているソフトウェアについては、設定用のプログラムの書き換えなどで対応してください。このターゲットでは、root 権限が必要な操作を行ってはなりません。

binary　binary ターゲットは、build ターゲットでビルドされたバイナリパッケージをまとめ上げるのに使用されます。binary ターゲットは、通常、binary-arch ターゲットと binary-indep ターゲットに依存するだけとなります。このターゲットは root 権限で呼び出されなくてはなりません。

binary-arch　ビルドされたファイルから特定アーキテクチャ用バイナリパッケージを生成するためのターゲットです。パッケージに含めるファイルをビルドするターゲットとして、build ターゲット (または定義されている場合は build-arch ターゲット) に依存するようにしておくべきです。このターゲットは root 権限で呼び出されなくてはなりません。

binary-indep　ビルドされたファイルからアーキテクチャ非依存バイナリパッケージを生成するためのターゲットです。パッケージに含めるファイルをビルドするターゲットとして、build ターゲット (または定義されている場合は build-indep ターゲット) に依存するようにしておくべきです。このターゲットは root 権限で呼び出されなくてはなりません。

clean、build、binary については、パッケージのトップレベルディレクトリ (debian ディレクトリの親ディレクトリ) をカレントディレクトリとして実行するよう定められています。

2.5 debhelper を使わない debian/rules

debian/rules の必須ターゲットに関する規約を簡単に眺めたところで、実際のパッケージの debian/rules の例を見てみましょう。以下では、Debian パッケージ作成用のツールを何も使わずに実装した、hello パッケージの debian/rules から抽出したコード (を一部改変したもの) を例に示します。

まずは clean ターゲットです。

```
clean:
        rm -f build
        -$(MAKE) -i distclean
        rm -rf *~ debian/tmp debian/*~ debian/files* debian/substvars
```

簡単に読めますね。以下のことをやっているだけです。

- タイムスタンプとして使用した build ファイルを削除する。
- ソフトウェアの Makefile の distclean ターゲットを実行し、ソフトウェアのビルドで生成されたファイルを削除する。

- パッケージのビルド時に debian ディレクトリ内に作成されたファイルを削除する。

続いて build ターゲットです。

```
CC = gcc
CFLAGS = -g -Wall

ifeq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
  CFLAGS += -O2
endif

build:
        ./configure --prefix=/usr
        $(MAKE) CC="$(CC)" CFLAGS="$(CFLAGS)"
        touch build
```

GNU Make の makefile に慣れていないと、変数の設定の部分が分かりにくいかもしれませんが、build ターゲット自体は非常に単純ですね。ソフトウェアをソースからインストールしたことのあるかたならお馴染みの、Autoconf を用いた一般的な C プログラムのビルド方法です。

最後に、binary、binary-arch、binary-indep の 3 つのターゲットです。

```
package = hello
docdir = debian/tmp/usr/share/doc/$(package)

INSTALL_PROGRAM = install

ifeq (,$(findstring nostrip,$(DEB_BUILD_OPTIONS)))
  INSTALL_PROGRAM += -s
endif

binary-indep: build

binary-arch: build
        rm -rf debian/tmp
        install -d debian/tmp/DEBIAN $(docdir)
        $(MAKE) INSTALL_PROGRAM="$(INSTALL_PROGRAM)" \
                prefix=$(CURDIR)/debian/tmp/usr install
        cp -a NEWS debian/copyright $(docdir)
        cp -a debian/changelog $(docdir)/changelog.Debian
        cp -a ChangeLog $(docdir)/changelog
        cd $(docdir) && gzip -9 changelog changelog.Debian
        gzip -r9 debian/tmp/usr/share/man
        dpkg-shlibdeps debian/tmp/usr/bin/hello
        dpkg-gencontrol
        chown -R root:root debian/tmp
        chmod -R u+w,go=rX debian/tmp
        dpkg-deb --build debian/tmp ..

binary: binary-indep binary-arch
```

一つずつ手順を追っていけば、以下のような手順でバイナリパッケージを生成していることが分かります。

1. インストール先のディレクトリを準備する。
2. ソフトウェアの Makefile の install ターゲットで、インストール先のディレクトリにファイルをインストールする。環境変数 DEB_BUILD_OPTIONS に nostrip という文字が含まれていない場合は、インストール時に、オブジェクトファイルからシンボルを切り捨てる (strip する)。
3. インストール先のドキュメント用ディレクトリに、ソフトウェアのドキュメントや debian/copyright をインストールする。
4. インストール先のドキュメント用ディレクトリに、debian/changelog およびソフトウェアの ChangeLog を、それぞれ changelog.Debian および changelog としてインストールする。
5. インストールした changelog.Debian および changelog やマニュアルページを圧縮する。
6. インストールしたバイナリファイルの、ビルドに使われたライブラリへの依存関係を調べて、debian/substvars に変数${shlibs:Depends} を設定する。
7. debian/control を元に debian/tmp/DEBIAN/control を作成する。その際に、debian/control 内の「${shlibs:Depends}」を、先に設定した変数の値で置換する。
8. インストール先のディレクトリやファイルのパーミッションを適切に設定する。
9. deb パッケージにする。

バイナリプログラム 1 つと付随データをバイナリパッケージ 1 つにインストールするだけなのに、かなり複雑な手

順を踏んでいます。これは以下のような理由からです。

- 以下のようなものに関して debian-policy の規約に従う必要がある。
  - インストールすべきドキュメント
  - インストールされたファイルのパーミッション
  - 圧縮すべきファイル
  - オブジェクトファイル内のシンボルの扱い
- 依存関係の記述を簡単にするための変数「${shlibs:Depends}」を処理する必要がある。
- debian/tmp/DEBIAN 以下に、debian/control などのパッケージのメタ情報を含むファイルや、メンテナスクリプトを入れなければならない。

バイナリパッケージ 1 つを生成するパッケージでさえこれだけの手順を踏まなければならないので、様々なファイルを複数のバイナリパッケージに分けてインストールするようなパッケージの作成に、かなり手間がかかるのは容易に想像できます。

■deb ファイルの内容 dpkg-deb –build がどのようにして複数のファイルを 1 つの「パッケージ」にまとめているか、興味があるかもしれません。そのような場合は、以下の実行例が参考になるでしょう。

```
noritada[3:50]%  ls
hello_2.2-2_i386.deb
noritada[3:50]%  ar x hello_2.2-2_i386.deb
noritada[3:50]%  ls
control.tar.gz        data.tar.gz  debian-binary  hello_2.2-2_i386.deb
noritada[3:50]%  cat debian-binary
2.0
noritada[3:50]%  tar ztf control.tar.gz
./
./control
noritada[3:50]%  tar ztf data.tar.gz
./
./usr/
./usr/share/
./usr/share/doc/
./usr/share/doc/hello/
[snip]
./usr/share/man/
./usr/share/man/man1/
./usr/share/man/man1/hello.1.gz
./usr/bin/
./usr/bin/hello
```

2.6 debhelper を用いた debian/rules

debian-policy の規約に従った Debian パッケージを容易に作成できるようにしてくれるのが、debhelper です。例として、先程の hello パッケージに対して debhelper を使用した hello-debhelper パッケージの `debian/rules` を見てみましょう。

まずは clean ターゲットです。

```
clean:
        dh_clean
        rm -f build
        -$(MAKE) -i distclean
```

hello パッケージの場合とほぼ同じですが、debian ディレクトリの掃除に関しては `dh_clean` というコマンドを使用していることが分かります。

次は build ターゲットですが、これは hello パッケージのものとまったく同じなので省略します。

最後に、hello パッケージではかなり複雑だった、binary、binary-arch、binary-indep の 3 つのターゲットです。

```
package = hello-debhelper

install: build
        dh_clean
        dh_installdirs
        $(MAKE) prefix=$(CURDIR)/debian/$(package)/usr install

binary-indep: install

binary-arch: install
        dh_installdocs -a NEWS
        dh_installchangelogs -a ChangeLog
        dh_strip -a
        dh_compress -a
        dh_fixperms -a
        dh_installdeb -a
        dh_shlibdeps -a
        dh_gencontrol -a
        dh_md5sums -a
        dh_builddeb -a

binary: binary-indep binary-arch
```

「dh_」で始まるコマンド群で占められているのが分かります。これらが debhelper のコマンドです。オプションやファイル名の指定などは部分的にはあるものの、基本的には非常にシンプルな記述となっており、バイナリパッケージ名やインストールパスなどの具体的な情報を記述する必要がなくなっています。ここでは単一のバイナリパッケージの例を取り上げていますが、複数のバイナリパッケージを生成する `debian/rules` についても同様にシンプルに記述できます。それは、debhelper の、以下のような特長からです。

- インストールされたデータを debian-policy の規約に従うように修正するコマンドを持つ。
  - `dh_strip`: オブジェクトファイル内のシンボルの切り捨て
  - `dh_compress`: テキストファイルの圧縮
  - `dh_fixperms`: パーミッションの修正
- デフォルトで、バイナリパッケージのインストールパスを debian/< バイナリパッケージ名 > と仮定して動作する[*3]。
- デフォルトですべてのバイナリパッケージに対して動作する。
- deb パッケージ作成に必要となる dpkg のコマンドをうまくラップして、他の debhelper のコマンドと同じようなかたちで実行できるようにする。

debhelper は、`debian/rules` をメンテナンスする手間を大きく減らしてくれる、非常に有用なツールだと言えます。

2.7 debhelper のコマンド群

debhelper の特長を説明したところで、そのコマンド群を概観しましょう。と言っても、コマンドの役割やオプションの説明、必要となる設定ファイルについては、各コマンドのマニュアルページや書籍『[入門] Debian パッケージ』に詳しい記述があるので、それらに譲ります。ここでは、大まかな分類と使い方の説明に焦点を絞った話をします。

`debian/rules` の記述を減らしてくれる debhelper ですが、「多数のコマンドのどれをどのタイミングで使うべきか分かりにくい」という欠点があります。それは、上で述べたように、debian-policy の規約や dpkg のコマンドごとに、debhelper にも対応するコマンドが存在するため、debhelper を使用しても、deb パッケージにまとめるまでの手順は多いからです。そこで、ここでは、使用する状況によってコマンドを 7 つに分類しておきます。debhelper で `debian/rules` を書く際の参考にしていただけると幸いです。

なお、「binary 系ターゲット」とは、binary、binary-arch、binary-indep の 3 つのターゲットのことです。

[*3] ちなみに、上のコードで様々なコマンドについている「-a」は、「アーキテクチャ依存バイナリパッケージすべてに適用する」という意味です。

2.7.1 確認系

以下のコマンドは、各ターゲットの頭において、ターゲットを正しい条件で実行しようとしているか確認するために使われるものです。

- dh_testroot
- dh_testdir

2.7.2 掃除系

以下のコマンドは、clean ターゲットで使われるものです。

- dh_clean

2.7.3 インストール前処理系

以下のコマンドは、binary 系ターゲットにおいて、仮インストールの準備段階で実行すべきものです。

- dh_clean
- dh_installdirs

2.7.4 インストール系

以下のコマンドは、binary 系ターゲットにおいて、仮インストール後にファイルなどを debian/< バイナリパッケージ名 > 以下にインストールするのに使われます。

- dh_install
- dh_installcatalogs
- dh_installchangelogs
- dh_installcron
- dh_installdebconf
- dh_installdocs
- dh_installemacsen
- dh_installexamples
- dh_installinfo
- dh_installinit
- dh_installlogcheck
- dh_installlogrotate
- dh_installman
- dh_installmenu
- dh_installmime
- dh_installpam
- dh_installudev
- dh_link
- dh_lintian

2.7.5 後処理系

以下のコマンドは、binary 系ターゲットにおいて、debian/< バイナリパッケージ名 > 以下にインストールされたファイルを、debian-policy に適合するよう修正するのに使われます。

- `dh_compress`
- `dh_fixperms`
- `dh_strip`

2.7.6 deb 化準備系

以下のコマンドは、binary 系ターゲットにおいて、deb パッケージ生成の直前に行う処理に使用されます。

- `dh_installdeb`
- `dh_perl`
- `dh_shlibdeps`

2.7.7 deb 化系

以下のコマンドは、binary 系ターゲットにおいて、deb パッケージ生成の直前に行う処理に使用されます。

- `dh_gencontrol`
- `dh_md5sums`
- `dh_builddeb`

2.8 debian/rules をさらに簡潔に書くには

debhelper の様々なコマンドを、使用するタイミングによって 7 つに分類してみました。これによって浮かび上がってくるのは、「どのパッケージでもコマンドを呼び出す順番はほぼ同じ」という事実です。実際、`dh_make` で作成した `debian/rules` の雛形をいじる場合でも、debhelper コマンド群の呼び出し順序を変更することはほとんどないでしょう。

ここで、「ではコマンド群の呼び出しの流れを一般化してしまえば、debhelper のコマンドばかりが並んだ、似たような `debian/rules` を量産しないで済む」と気付くと思います。実際問題として、雛形から作成した、似たような `debian/rules` を多数管理するのはコストがかかります。新しいコマンドができた場合、それをすべての `debian/rules` の同じ場所に加えればなりません。

そこで登場するのが CDBS です。次は、CDBS の debhelper ルールを用いて `debian/rules` をさらに簡潔にします。

2.9 CDBS の debhelper ルールを用いた debian/rules

「似た内容の、短くはない `debian/rules` を量産する」という debhelper の欠点を解決するのが、CDBS の debhelper ルール (debhelper.mk) です。CDBS とは、`debian/rules` の記述をモジュール化して再利用できるようにしたものをライブラリとして提供し、一般的な流れに沿った `debian/rules` を非常に簡単に記述できるようにするシステムです。CDBS の debhelper ルール (debhelper.mk) では、debhelper を用いたビルドの一般的な流れが既に定義されているので、debhelper の各コマンドに与えるオプションや引数を指定するだけで `debian/rules` が書けます。

hello-debhelper の `debian/rules` を CDBS の debhelper ルールを用いて書き換えると、次のようになります。

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

include /usr/share/cdbs/1/rules/debhelper.mk

package = hello-cdbs

CC = gcc
CFLAGS = -g -Wall

ifeq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
  CFLAGS += -O2
endif

clean::
        -$(MAKE) -i distclean

install/hello-cdbs::
        $(MAKE) prefix=$(CURDIR)/debian/$(package)/usr install

common-configure-arch::
        ./configure --prefix=/usr

common-build-arch::
        $(MAKE) CC="$(CC)" CFLAGS="$(CFLAGS)"

DEB_INSTALL_DOCS_ALL := NEWS
DEB_INSTALL_CHANGELOGS_ALL := ChangeLog
```

書き換えは、以下のような手順で行いました。

1. `/usr/share/cdbs/1/rules/debhelper.mk` をインクルードする。
2. debhelper のコマンドをすべて削除し、引数やオプションは変数に設定しなおす
3. ターゲットの指定を通常のコロンから二重コロンに変更する。
4. ターゲット名を書き換える。

以下でこれらを詳しく説明します。

2.9.1 `/usr/share/cdbs/1/rules/debhelper.mk` をインクルードする

debhelper ルールの定義を取り込むには、`/usr/share/cdbs/1/rules/debhelper.mk` をインクルードする必要があります。

2.9.2 debhelper のコマンドをすべて削除し、引数やオプションは変数に設定しなおす

debhelper のコマンドは、引数やオプションがついていないものについては削除してかまいません。対応する操作が debhelper ルール内で定義されています。

引数やオプションがついている場合は、その内容を変数に設定しなおす必要があります。CDBS の debhelper ルールでは、debhelper の各コマンドの引数やオプションに対応する変数を使用するようになっています。変数の例を示します。

`DEB_INSTALL_DOCS_`< **バイナリパッケージ名** > < バイナリパッケージ名 > にインストールしたいドキュメントのリストを設定します。値が設定されると、適切なタイミングで、「`dh_installdocs -p` < **バイナリパッケージ名** > < **変数の値** >」が実行されます。

`DEB_INSTALL_DOCS_ALL` すべてのバイナリパッケージにインストールしたいドキュメントのリストを設定します。値が設定されると、適切なタイミングで、「`dh_installdocs -A` < **変数の値** >」が実行されます。

2.9.3 ターゲットの指定を通常のコロンから二重コロンに変更する

CDBS を使用する場合は、ターゲットの指定に二重コロンを使用する必要があります。これは、モジュール化に、「ターゲットを二重コロンで指定することで処理を多重定義できる」という GNU Make makefile の機能を使用していためです。以下のリストのように、通常のコロンの場合は後で指定された処理が実行されますが、二重コロンで定義すると両方の処理が実行されます。

```
noritada[10:22]%  cat Makefile
hoge:
        @echo foo

hoge:
        @echo bar
noritada[10:22]%  make
Makefile:5: 警告: ターゲット `hoge' へのコマンドを置き換えます
Makefile:2: 警告: ターゲット `hoge' への古いコマンドは無視されます
bar
noritada[10:22]%  cat Makefile
hoge::
        @echo foo

hoge::
        @echo bar
noritada[10:22]%  make
foo
bar
```

2.9.4 ターゲット名を書き換える

CDBS の debhelper ルール[*4]では、binary や build-arch、build-indep などの大きな流れを表すターゲットを複数の処理に分割し、ユーザが処理の多重定義を用いて適切なタイミングで処理を挿入できるようにしています。ターゲットの例を示します。

common-configure-arch　configure スクリプトのようなものを用いてビルド前にパッケージを設定するのに使用されます。これはアーキテクチャ依存のバイナリパッケージに対するものです。

common-configure-indep　configure スクリプトのようなものを用いてビルド前にパッケージを設定するのに使用されます。これはアーキテクチャ非依存のバイナリパッケージに対するものです。

common-build-arch　Makefile の all ターゲットのようなものを用いてソフトウェアをビルドするのに使用されます。

install/< **バイナリパッケージ名** >　Makefile の install ターゲットのようなものを用いて仮インストールを行うのに使用されます。

2.10 ここまでのまとめ

パッケージのビルド方法や `debian/rules` の各ターゲットの役割を概観した上で、`debian/rules` をより簡潔に、より分かりやすく書く方法を求めて、debhelper や CDBS を見てきました。簡潔に分かりやすく書くことはメンテナンスのしやすさに繋がるので、できるだけ `debian/rules` をシンプルに保つことをお勧めします。

2.11 パッチ管理ツールを用いた開発元のソースコードの修正

これまでは debian ディレクトリ以下のみをいじってきましたが、最後に、開発元が配布しているソースコードに修正を加える方法を説明します。`debian/rules` についてはメンテナンスのしやすさを重要視しましたが、それは、開発元が配布しているソースコードに修正を加える際にも当てはまります。

Debian パッケージを管理していると、開発元のソースコードに手を加えたくなることがよくあります。理由は様々です。

- 解決したい問題が、次期リリースに向けて開発中のソースコードでは既に修正されているのだが、次期リリースは暫く出そうにもない。パッケージについては一足早く修正しておきたい。
- 開発元で開発がなされなくなったため、ソフトウェアの問題を自分で解決する必要がある。

こんなときに、開発元のソースコードに修正を加える最も簡単な方法は、debian ディレクトリの外側のソースコードを直接いじることです。ネイティブでない Debian ソースパッケージは、`.tar.gz` ファイルと `.diff.gz` ファイル、`.dsc` ファイルから成るので、ソースコードに変更を加えれば、それは開発元のソースコードからの差分として `.diff.gz` ファイルに含まれます。

[*4] 厳密に言えば debhelper ルールがインクルードしている buildcore ルール。

しかし、この安直な方法にはもちろん大きな問題があります。

- 時間が経つと加えた変更の背景や目的が分からなくなる。
- 変更の数が増えていくと混ざってしまい、意味的なまとまりのある単位で変更を切り分けにくくなる。
- 開発元から新しいバージョンがリリースされたときに、そのバージョンでも有効な変更とそのバージョンでは無効な変更を切り分けられなくなる。

そこで、変更を開発元のソースコードに直接加えることはせず、dpatch や quilt を用いて、意味的なまとまりのある単位でパッチとして管理することが推奨されています。簡単にまとめると、以下のような方法です。

- 変更はすべてパッチとして debian ディレクトリ以下に保持しておき、開発元のソースコードには直接的な改変は一切加えない。
- パッケージのビルド時には当てたり外したりする。ソースパッケージをビルドするときには外した状態にしておき、バイナリパッケージをビルドするときには当てた状態にしておく。
- パッケージの更新時には、各パッチについて、有効性を吟味する。

ここでは、簡単にその使い方を見ていきます。

2.11.1 パッチ置き場

一般に、dpatch や quilt でパッチを管理する場合は、debian/patches ディレクトリをパッチ置き場として使用します。dpatch の場合は debian/patches/00list が、quilt の場合は debian/patches/series がそれぞれパッチのリストとなっており、debian/patches に含まれている一連のパッチが、このパッチリストに記載されている順に適用されていきます。

dpatch を使用している kazehakase パッケージの debian/patches の例:

```
noritada[16:39]%  ls debian/patches/*
debian/patches/00list
debian/patches/05_add_missing.dpatch
debian/patches/20_user_agent_tag.dpatch
debian/patches/30_bookmarkbar_DSA.dpatch
debian/patches/50_passwordmgr.dpatch
debian/patches/60_fix_ftbfs.dpatch
debian/patches/70_enable_gtk_deprecated.dpatch
debian/patches/80_NSIBADCERTLISTNER.dpatch
noritada[16:39]%  cat debian/patches/00list
20_user_agent_tag
30_bookmarkbar_DSA
50_passwordmgr
```

quilt を使用している skksearch パッケージの debian/patches の例:

```
noritada[16:52]%  ls debian/patches/*
debian/patches/clean-build-errors-and-warnings.diff
debian/patches/conf-file.diff
debian/patches/db4.3.diff
debian/patches/dic-bufsize.diff
debian/patches/plain-search.diff
debian/patches/series
noritada[16:52]%  cat debian/patches/series
conf-file.diff
db4.3.diff
dic-bufsize.diff
plain-search.diff
clean-build-errors-and-warnings.diff
```

2.11.2 パッチ管理

パッチを当てるには、dpatch の場合は dpatch apply を、quilt の場合は quilt push を使用してください。外すには、dpatch の場合は dpatch deapply を、quilt の場合は quilt pop を使用してください。

新たなパッチを作成するには、まずパッチをどこに挟むか決めてください。dpatch において < あるパッチ > の次に挟む場合は、次のように行います。

```
$ dpatch-edit-patch patch <新規パッチ> <あるパッチ>
$ editor <あるファイル>（パッチに含める変更を加えます）
$ exit 0
```

quilt において同様の操作を行う場合は、次のようにします。

```
$ quilt push <あるパッチ>（<あるパッチ>が既に当たっている場合は、quilt pop <あるパッチ>になります）
$ quilt new <新規パッチ>
$ quilt add <あるファイル>（パッチの作成を開始した後、変更する対象は add で追加しておく必要があります）
$ editor <あるファイル>（パッチに含める変更を加えます）
$ quilt refresh
```

変更を加えるためにエディタを使う場合は、「quilt edit < **あるファイル** >」を実行すれば、「quilt add < **あるファイル** >」を実行せずに編集することも可能です。

2.11.3 ビルド時に適切にパッチを取り扱う

折角パッチを debian/patches に入れておいたところで、ソースパッケージをビルドするときにはパッチを外した状態にしておき、バイナリパッケージをビルドするときにはパッチを当てた状態にしておかなければ、意味がありません。これは、debian/rules でパッチに関する依存関係を適切に設定することで実現できます。以下では、様々な場合について、debian/rules の記述例を示します。

まず、dpatch についてです。debhelper を使用している場合は、/usr/share/doc/dpatch/examples/rules/rules.new.dh.gz を参考にしてください。

```
build: build-stamp
build-stamp: patch
        dh_testdir
        # ここでソフトウェアをビルドする。
        touch build-stamp

clean: clean1 unpatch
clean1:
        dh_testdir
        dh_testroot
        dh_clean -k
        # ここで掃除をする。

patch: patch-stamp
patch-stamp:
        dpatch apply-all
        dpatch cat-all >patch-stamp
        touch patch-stamp

unpatch:
        dpatch deapply-all
        rm -rf patch-stamp debian/patched
```

CDBS を利用している場合は、CDBS のドキュメントにあるとおり、以下のような行を加えるだけでかまいません。ただし、autotools.mk をインクルードしている場合は、dpatch.mk をインクルードするのはそれよりも後ろにしてください。

```
include /usr/share/cdbs/1/rules/dpatch.mk
```

quilt についても、CDBS を利用している場合は、CDBS のドキュメントにあるとおり、以下のような行を加えるだけでかまいません。

```
include /usr/share/cdbs/1/rules/patchsys-quilt.mk
```

■パッチ管理ツールの近況と今後 Debian で長いこと使われてきたパッチ管理ツール dpatch については、dh_make が、1ヶ月ほど前にバージョン 0.45 で--dpatch オプションを提供し始めました。これまでは、パッケージ化への入口となる dh_make によるサポートがなかったため、「パッチ管理ツールはパッケージメンテナが好みで使用するもの」という雰囲気があったように思いますが、今後、パッチ管理が普及し、開発元のソースコードは一切いじらないという風潮が強くなるのだとしたら、嬉しいことです。
一方で quilt については、昔は非常にマイナーなパッチ管理ツールでしたが、xorg を管理する Debian X Strike Force や glibc を管理する Debian GNU Libc Maintainers など、大規模なパッケージで高い評判が得られ、徐々に浸透してきているようです。また、直観的なユーザインタフェースのためか、パッケージ管理の初心者からの評判もよいようで、debian-mentors メーリングリストなどでもしばしば話題に出ているのを見掛けます。Debian パッケージ以外でも様々なところで使えるツールだと思うので、今後は認知度が高まることを期待しています。
さて、このようなパッチ管理ツールですが、将来的にはソースパッケージそのものにパッチ管理の機構が取り込まれていくことが期待されます。これまでは、オリジナルからの差分を.diff.gz ファイルにすべて押し込んでいましたが、これは、「オリジナルからの変更を分かりやすく管理する」という観点からは非常に不便でした。この問題を解決するため、dpkg 1.14.18 では、新たなソースパッケージの形式「3.0 (quilt)」がサポートされました。lenny の次のリリースから、デフォルトかつ推奨されるソースパッケージ形式になる予定です。
「3.0 (quilt)」では、ソースパッケージは以下のファイルから成ります。

- .orig.tar.< 拡張子 > ファイル
- .debian.tar.< 拡張子 > ファイル
- .orig-< 部品 >.tar.< 拡張子 > ファイル群 (任意)

注目すべきは、これらの展開方法です。

1. .orig.tar.< 拡張子 > ファイルが展開される。
2. < 部品 > サブディレクトリ内で.orig-< 部品 >.tar.< 拡張子 > ファイルが展開される。< 部品 > サブディレクトリが既にある場合は置換される。
3. トップレベルディレクトリに.debian.tar.< 拡張子 > ファイルが展開される。.debian.tar.< 拡張子 > ファイルには debian ディレクトリが含まれていなければならない。debian ディレクトリが既にある場合はまず削除される。
4. debian/patches/debian.series または debian/patches/series のリストに含まれているパッチがすべて適用される。

つまり、「debian ディレクトリ以下の追加 + 開発元のソースコードに対する変更」から成るこれまでの.diff.gz ファイルは廃止され、ソースコードに対する変更はパッチのかたちでしかできなくなります。これは、メンテナンス性を高める意味で非常に大きな前進でしょう。

3 Linux カーネルパッチ の Debian パッケージ作成

岩松 信洋

3.1 はじめに

テスト段階の機能が Linux カーネルへのパッチとして公開されている場合がありますが、これらの パッチを Debian で利用したい場合には、Debian で配布している カーネルソースパッケージや vanilla カーネルにパッチを当ててパッケージ化されている方も多いのではないでしょうか。Debian の特徴の一つとしてパッケージ管理が挙げられますが、実は Linux カーネルパッチも Debian パッケージとして管理することができ、既にいくつかのパッケージが利用可能になっています。また、この Linux カーネルパッチパッケージを作成するためのサポートパッケージ **dh-kpatches** が提供されており、このパッケージを利用することによりパッチ用パッケージの作成や、カーネルへのパッチ適用およびコンパイルが可能になっています。今回は Evgeniy Polyakov（えぶじぇにー ぽるやこふ）が作成している新しい Network Filesystem POHMELFS[*5] を題材にして、Linux カーネル向けパッチのパッケージ作成方法とテスト方法について説明します。

3.2 Linux カーネルパッチ の Debian パッケージの仕組み

Linux カーネルパッチパッケージは、カーネルパッチとパッチをコントロールするためのファイル kpatches を提供します (図 1)。このファイルでは、アーキテクチャ、カーネルバージョンの指定が可能になっています。そして、パッケージ作成時に kpatches ファイルから patch/unpatch 時に使用されるパッチコントロールスクリプトが生成されます。パッケージをインストールすると、/usr/src/kernel-patches ディレクトリに各ファイルが展開されます。Debian カーネルパッケージを作成するためのコマンド make-kpkg の –added-patches オプションで指定されたパッチパッケージ名をこのディレクトリから検索し、パッチを適用してから、カーネルをコンパイルします

3.3 パッケージ作成前の準備

3.3.1 dh-kpatches パッケージのインストール

dh-kpatches を使って、作成されたパッケージを使うことにより、make-kpkg コマンド [*6]でカーネルパッチパッケージで提供しているパッチを適用した Linux カーネルパッケージが作成できるようになります。パッケージ作成サポート用として、dh-make をインストールしておくと良いでしょう。dh-make を使うことによって、パッケージの雛形を容易に作成することができます。

[*5] http://tservice.net.ru/~s0mbre/old/?section=projects&item=pohmelfs
[*6] Debian でカーネルパッケージを作成するためのコマンド。kernel-package パッケージで提供されている。

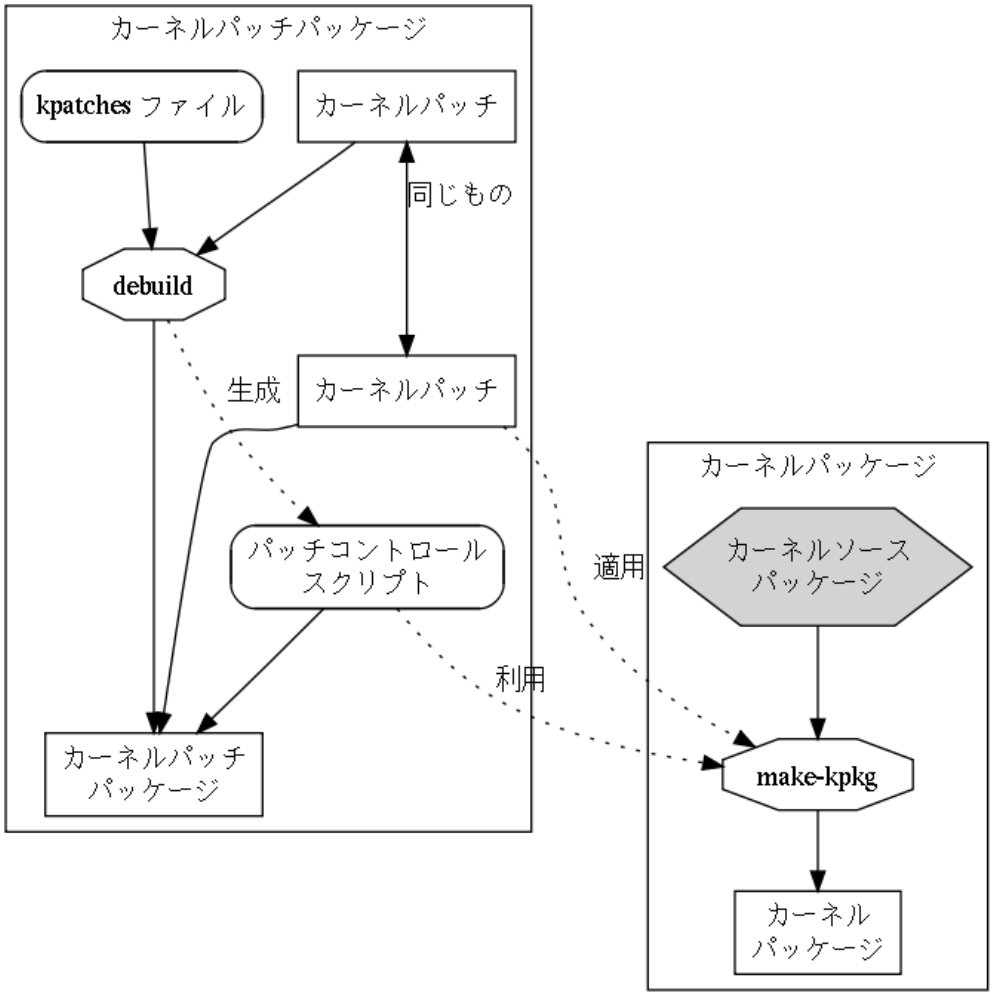

図 1　Linux カーネルパッチ の Debian パッケージの仕組み

```
$ sudo apt-get update
$ sudo apt-get install dh-kpatches dh-make build-essential
```

3.3.2　パッチの用意

まず、カーネル向けのパッチを作成する必要があります。開発者によっては、既にパッチがカーネルバージョン毎に用意されている事もありますが、最近では、Linus のツリーに容易に追従できるように、Git を使ってソースコードが管理されている場合が多いです。POHMELFS でも ソースコードは Git で管理されており、安定版のカーネル（この原稿を書いている時点では、Linux 2.6.25）に常に追従されています。この差分を取得するには、Git リポジトリを取得し、git diff コマンド等で取り出せばよいでしょう。

```
$ git clone http://tservice.net.ru/~s0mbre/archive/pohmelfs/pohmelfs.git
Initialized empty Git repository in /tmp/pohmelfs/.git/
got 37e1b82c0535386cf09b3821dff5e8cb5f9e26b4
walk 37e1b82c0535386cf09b3821dff5e8cb5f9e26b4
<snip>
$ cd pohmelfs
$ git tag
<snip>
v2.6.24-rc8
v2.6.25
v2.6.25-rc1
v2.6.25-rc2
<snip>
$ git diff v2.6.25 > ~/pohmelfs.diff
```

3.3.3 ディレクトリの作成

パッチが作成できたら、パッケージ作成用のディレクトリを作成し、その中にパッチファイルをコピーします。ディレクトリ名は Debian のポリシーに合わせたもの (ソフトウェアまたは機能名-バージョン) にしておき、パッチファイル名はパッチ機能名-カーネルバージョン にしておきます。これは、このようなファイル名を dh-kpatches が要求するためです。

```
$ mkdir linux-patch-pohmelfs-20080707
$ cd linux-patch-pohmelfs-20080707
$ cp ../pohmelfs.diff pohmelfs-2.6.x
```

3.4 dh_make を使った雛形の作成

パッケージ作成の準備ができたら、 dh_make コマンドを使って雛形を作成します。dh_make にはカーネルパッチ用のサポートはないので、single バイナリの雛形を作成するオプションを指定して、作成します。-r オプションは `orig.tar.gz` ファイルを作成し、-s オプションはシングルバイナリ用の雛形を出力します。

```
$ dh_make -r -s
Maintainer name : Nobuhiro Iwamatsu
Email-Address   : iwamatsu@nigauri.org
Date            : Mon, 07 Jul 2008 01:00:33 +0900
Package Name    : linux-patch-pohmelfs
Version         : 20080707
License         : blank
Using dpatch    : no
Type of Package : Single
Hit <enter> to confirm:
Currently there is no top level Makefile. This may require additional tuning.
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the linux-patch-pohmelfs Makefiles install into $DESTDIR and not in / .
```

3.5 debian ディレクトリ内の変更

dh_make コマンドで カーネルパッチを Debian パッケージにするための雛形が作成できました。これを元に内容を変更していきます。

3.5.1 サンプルスクリプト等の削除

dh_make で作成されるサンプルスクリプトは必要ないので全て削除します。また、 dirs ファイルや doc ファイルも必要ないため削除します。

```
$ rm -rf ./debian/*.ex ./debian/*.EX ./debian/docs ./debian/dirs
```

3.5.2 debian/copyright の修正

パッチの情報に合わせて、debian/copyright の修正を行います。pohmelfs の場合は以下のように修正します。

```
This package was debianized by Nobuhiro Iwamatsu <iwamatsu@nigauri.org> on
Mon, 07 Jul 2008 01:00:33 +0900.

It was downloaded from
        http://tservice.net.ru/~s0mbre/archive/pohmelfs/pohmelfs.git
This is Git repository. I made the difference of the latest committing a
patch from v2.6.25 tag.

Upstream Author(s):

    Evgeniy Polyakov <johnpol@2ka.mipt.ru>

Copyright:

    Copyright (C) 2008 Evgeniy Polyakov <johnpol@2ka.mipt.ru>

License:

 GPLv2

The Debian packaging is (C) 2008, Nobuhiro Iwamatsu <iwamatsu@nigauri.org> and
is licensed under the GPL, see '/usr/share/common-licenses/GPL'.

# Please also look if there are files or directories which have a
# different copyright/license attached and list them here.
```

3.5.3 debian/README.Debian の修正

debian/README.Debian ファイルに Debian で使うための HowTo などを書いておきましょう。サポートしているカーネルバージョンや、カーネルイメージの作成方法などを書いておくのが一般的なようです。

```
linux-patch-pohmelfs for Debian
-------------------------------

This patch is pohmelfs support patch for Debian Linux kernel.
You can try with linux-source-2.6.25 package.

- How to use
  $ sudo apt-get install linux-source-2.6.25 libncurses-dev kernel-package
  $ cd /usr/src ; tar -xjf linux-source-2.6.25.tar.bz2 ; cd linux-source-2.6.25
  $ make-kpkg clean
  $ cp /boot/config-2.6.25-2-686 .config
  $ make menuconfig
  $ make-kpkg --rootcmd fakeroot --append-to-version -pohmelfs --revision 0.1 --added_patches=pohmelfs kernel-image

 -- Nobuhiro Iwamatsu <iwamatsu@nigauri.org>  Mon, 07 Jul 2008 01:00:33 +0900
```

3.5.4 debian/control の修正

次に debian/control ファイルを修正します。注目すべきところは、作成されるパッケージで指定してある、**Depends: ${kpatch:Depends}** です。ここで kpatch:Depends を指定することによって、カーネルパッチパッケージを使ったカーネル作成に必要な依存パッケージを置き換えてくれます。また、ソースパッケージの **Build-Depends** に **dh-kpatches パッケージ** を追加する事を忘れないようにしましょう。

```
Source: linux-patch-pohmelfs
Section: devel
Priority: extra
Maintainer: Nobuhiro Iwamatsu <iwamatsu@nigauri.org>
Build-Depends: debhelper (>= 6), dh-kpatches
Standards-Version: 3.8.0.1
Homepage: http://tservice.net.ru/~s0mbre/old/?section=projects&item=pohmelfs

Package: linux-patch-pohmelfs
Architecture: all
Depends: ${kpatch:Depends}
Description: POHMELFS kernel patch
 POHMELFS stands for Parallel Optimized Host Message Exchange Layered File System.
 Development status can be tracked in filesystem section.
 This is a high performance network filesystem with local coherent cache of data
 and metadata.
 Its main goal is distributed parallel processing of data. Network filesystem is a
 client transport.
 POHMELFS protocol was proven to be superior to NFS in lots (if not all, then it
 is in a roadmap) operations.
```

3.5.5 debian/changelog の修正

dch コマンドを使って debian/changelog ファイルを修正します。

```
$ dch
linux-patch-pohmelfs (20080707-1) unstable; urgency=low

  * Initial release

 -- Nobuhiro Iwamatsu <iwamatsu@nigauri.org>  Mon, 07 Jul 2008 01:25:37 +090
```

3.5.6 kpatches ファイル

どのパッケージ内に収録されているパッチをどのカーネルバージョンに当てればいいのかなどをコントロールするためのファイル kpatches を用意する必要があります。このファイルは以下のようなフォーマットになっており、カーネルバージョン毎に Patch-file と Kernel-version の項目を追加する必要があります。また、このファイルを **パッケージ名.kpatches** として 名前を変更する必要があります。

```
Patch-name: POHMELFS patch
Patch-id: pohmelfs <-- make-kpkg の --added-patches で指定するパッチ名
Architecture: all <-- サポートするアーキテクチャ
Path-strip-level: 1

Patch-file: pohmelfs-2.6.x <-- パッチファイル名
Kernel-version: 2.6.25 <-- パッチがサポートするカーネルバージョン
```

3.5.7 debian/rules ファイルの修正

debian/rules ファイルで注意する点として、 install ターゲットで、dh_installkpatches を指定する必要があります。dh_installkpatches でカーネルパッチが kpatches ファイルと共ににパッケージ作成用の一時ディレクトリにコピーされます。

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

build: build-stamp
build-stamp:
        dh_testdir
        touch build-stamp

clean:
        dh_testdir
        dh_testroot
        rm -f build-stamp
        dh_clean

install: build
        dh_testdir
        dh_testroot
        dh_clean -k
        dh_installdirs
        dh_installkpatches

# Build architecture-independent files here.
binary-indep: build install
        dh_testdir
        dh_testroot
        dh_installchangelogs
        dh_link
        dh_strip
        dh_compress
        dh_fixperms
        dh_installdeb
        dh_shlibdeps
        dh_gencontrol
        dh_md5sums
        dh_builddeb

# Build architecture-dependent files here.
binary-arch: binary-indep

# We have nothing to do by default.

binary: binary-indep binary-arch
```

3.5.8 ディレクトリ構成

最終的な debian ディレクトリの構成をチェックしてみます。以下のようになっているとよいでしょう。

```
linux-patch-pohmelfs-20080707/
|-- debian
|   |-- README.Debian
|   |-- changelog
|   |-- compat
|   |-- control
|   |-- copyright
|   |-- linux-patch-pohmelfs.kpatches
|   `-- rules
`-- pohmelfs-2.6.x
```

3.6 パッケージのビルドおよびインストール

パッケージのビルドを行う際には、通常のパッケージ作成と変わりません。debuild などを使ってパッケージの作成を行ってください。作成されたパッケージをインストールすると、/usr/src/kernel-patches にパッチと制御用のスクリプトがインストールされます。make-kpkg コマンドはこのディレクトリを参照し、パッチを適用します。

```
% debuild
% sudo dpkg -i ../linux-patch-pohmelfs_20080707-1_all.deb
```

3.7 パッケージのテスト

カーネルパッチパッケージのテストは、カーネルソースコードにパッチを当て、カーネルがコンパイルできるか、確認する必要があります。また、Debian の場合は、Debian と kernel.org で配布しているカーネルの 2 種類を考慮する必要がありますが、前者の方をテストしていれば十分でしょう。インストールしたカーネルパッチパッケージを使って、カーネルをコンパイルするには、make-kpkg コマンドの–added_patches のオプションを使って、パッチを指定します。パッチが当たっているかソースを確認し、カーネルのコンパイルが正常に行われているか確認しましょう。また、作成された カーネルパッケージを実際にインストールし、テストを行うことも重要です。

```
$ sudo apt-get install linux-source-2.6.25 libncurses-dev kernel-package
$ cd /usr/src ; tar -xjf linux-source-2.6.25.tar.bz2 ; cd linux-source-2.6.25
$ make-kpkg clean
$ cp /boot/config-2.6.25-2-686 .config
$ make menuconfig
$ make-kpkg --rootcmd fakeroot --append-to-version -pohmelfs --revision 0.1 --added_patches=pohmelfs kernel-image
```

3.8 パッケージのアップデート方法

いまのところ、カーネルパッチソースパッケージのアップデート方法が決まっていません。適当なディレクトリを作成して、uupdate コマンドを実行するのが一番容易と思われます。将来的には、パッチを指定することによって、ソースパッケージをアップデートできるようにしたいと考えています。

3.9 まとめ

以上の手順を使うと、Linux カーネルパッチの Debian パッケージを作成することができます。しかし、このままではかなり手間がかかりますので、今回の成果物をまとめ、dh-make で Linux カーネルパッチパッケージの雛形が作成できるように機能を追加しました (#304688)。[*7] このパッチが取り込まれると、dh-make を使うことにより、より簡単に Linux カーネルパッチパッケージが作成できるようになるでしょう。また、cdbs を使った方法をまだ調べていないので、調べようと思っています。ちなみに、pohmelfs は linux-2.6.27 or 2.6.28 に取り込まれるようです。よかったよかった。

[*7] dh-make 0.47 でパッチが取り込まれ、利用可能になりました。

4 Linux カーネルモジュールの Debian パッケージ作成

岩松 信洋

4.1 はじめに

今回は video データ loopback 用 Linux カーネルモジュール、vloopback を題材にして、Linux カーネルモジュールの Debian パッケージ作成方法を説明します。

4.2 なぜカーネルモジュールソースコードをパッケージ化するのか

Linux の カーネルモジュールソースコードをパッケージ化する理由の一つとして、カーネルバージョンに合わせたドライバを管理する手間が省ける事とモジュールドライバコンパイルフロントエンド module-assistant による 操作のしやすさが大きいでしょう。このソフトウェアを使うことによって、容易にモジュールのコンパイルおよびインストールを行う事ができます。現時点では、カーネルバージョンが上がる度にコンパイルする必要がありますが、パッケージ化しておくと、ある程度まで自動化できるため、アップデートも容易になります。

4.3 カーネルモジュールパッケージの仕組み

パッケージの作り方の説明の前にカーネルモジュールパッケージの仕組みについて説明します (図 2)。カーネルモジュールパッケージは 内部に カーネルモジュールソースコードとコンパイルするための Makefile(`debian/rules`) と、control ファイル (debian/control に相当するもの) を固めたものである ドライバソースイメージを持っています 。このドライバソースイメージは、実際は Debian ソースパッケージと構成は同じになっており、 bzip2 で圧縮したものになります。カーネルモジュールパッケージをインストールすると、`/usr/src/` に ドライバソースイメージが置かれ、`module-assistant` がこのファイルを`/usr/src/modules` ディレクトリ以下、ビルド毎に展開し、ドライバ用の Debian パッケージビルドを行います。作成された パッケージは `/usr/src` に置かれます。モジュールソースパッケージは、このドライバソースイメージを作成し、`module-assistant` と連携できる機能を提供します。

4.4 ドライバソースコードの取得と展開

`vloopback` は `flashcam` という フリーソフトウェアと共に提供されています。今回は、`vloopback` のソースコードだけをターゲットにするので、ソースコードを取り出します。

```
$ wget http://www.swift-tools.net/Flashcam/flashcam-1.1.tgz
$ tar -xzf flashcam-1.1.tgz
$ cp -rf flashcam-1.1/vloopback-1.1.2 .
$ cd vloopback-1.1.2
```

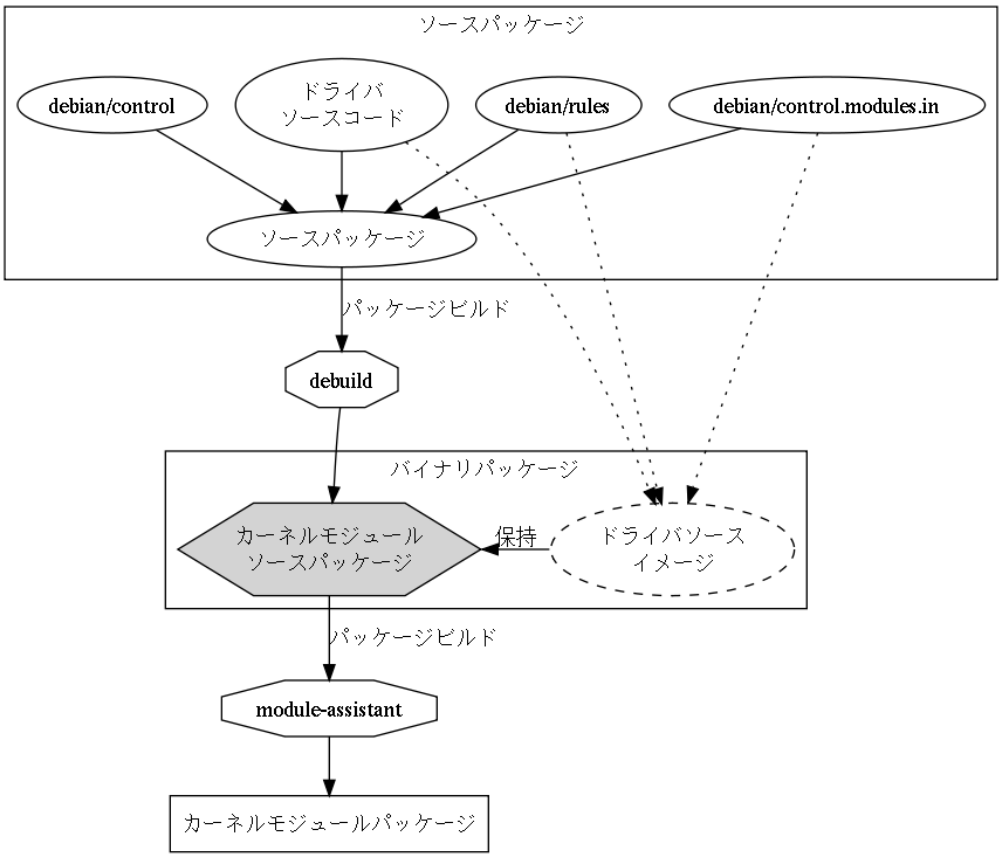

図 2　カーネルモジュールパッケージの仕組み

4.5　dh_make を使った雛形の作成

パッケージ作成の準備ができたら、 dh_make コマンドを使って雛形を作成します。dh_make には、カーネルモジュールの雛形を作成するオプション -k / --kmod が提供されています。

```
$ dh_make -k -r
Maintainer name : Nobuhiro Iwamatsu
Email-Address   : iwamatsu@nigauri.org
Date            : Tue, 15 Jul 2008 23:21:23 +0900
Package Name    : vloopback
Version         : 1.1.2
License         : blank
Using dpatch    : no
Type of Package : Kernel Module
Hit <enter> to confirm:
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the vloopback Makefiles install into $DESTDIR and not in / .
```

4.6　debian ディレクトリ内の変更

Debian パッケージにするための雛形が作成できたので、これを元に内容を変更していきます。

4.6.1　サンプルスクリプト等の削除

dh_make で作成されるサンプルスクリプトは必要ないので全て削除します。

```
$ rm -rf ./debian/*.ex ./debian/*.EX
```

4.6.2 debian/copyright の修正

パッチの情報に合わせて、debian/copyright の修正を行います。vloopback の場合は以下のように修正します。

```
This package was debianized by Nobuhiro Iwamatsu <iwamatsu@nigauri.org> on
Tue, 15 Jul 2008 23:21:23 +0900.

It was downloaded from <http://www.swift-tools.net/Flashcam/flashcam-1.1.tgz>

Upstream Author(s):

    Olivier Debon <olivier@debon.net>

Copyright:

    Copyright (C) 2008 Olivier Debon <olivier@debon.net>

License:

    GPLv2

The Debian packaging is (C) 2008, Nobuhiro Iwamatsu <iwamatsu@nigauri.org> and
is licensed under the GPL, see ‘/usr/share/common-licenses/GPL’.

# Please also look if there are files or directories which have a
# different copyright/license attached and list them here.
```

4.6.3 debian/README.Debian の修正

debian/README.Debian ファイルを修正します。雛形がある程度書いておいてくれているので、あまり修正する必要はありません。確認した Linux カーネルバージョンなどを書いておくと良いかもしれません。

```
vloopback for Debian
--------------------

Please see ./README for a description of the vloopback software.

The Debian vloopback source package provides two packages,

 vloopback-source, which provides the source for the kernel modules

The vloopback-source package can be used in several ways,

 - Using the make-kpkg(1) command provided by the kernel-package Debian
   package. This will produce a corresponding vloopback-modules package for
   the Debian kernel-image package that you are using. This is "the Debian
   way". See the "modules_image" section of the make-kpkg(1) man page.

 - Changing to the /usr/src/modules/vloopback/ directory and building as
   the README file instructs using "make; make install". This will build
   and install a module specific to the system you are building on and is
   not under control of the packaging system.

 -- Nobuhiro Iwamatsu <iwamatsu@nigauri.org>  Tue, 15 Jul 2008 23:21:23 +0900
```

4.6.4 debian/control の修正

次に debian/control ファイルを修正します。dh_make で作成した場合は、vloopback-utils といった、ドライバをサポートするためのパッケージを作成するエントリがあります。ソースコードにツールが含まれている場合は、このエントリを使って、ツール用のパッケージを作成できるようになっています。今回は必要ないので削除します。

以下に debian/control ファイルの例を示します。

```
Source: vloopback
Section: graphics
Priority: extra
Maintainer: Nobuhiro Iwamatsu <iwamatsu@nigauri.org>
Build-Depends: debhelper (>= 7), bzip2
Standards-Version: 3.8.0.1
Homepage: http://www.swift-tools.net/Flashcam/

Package: vloopback-source
Architecture: all
Depends: module-assistant, debhelper (>= 7), make, bzip2
Description: Source for the vloopback driver.
 This package provides the source code for the vloopback kernel modules.
 The vloopback package is also required in order to make use of these
 modules. Kernel source or headers are required to compile these modules.
```

4.6.5 debian/changelog の修正

dch コマンドを使って debian/changelog ファイルを修正します。

```
$ dch
vloopback (1.1.2-1) unstable; urgency=low

  * Initial release

 -- Nobuhiro Iwamatsu <iwamatsu@nigauri.org>  Tue, 15 Jul 2008 23:21:23 +0900
```

4.6.6 control.modules.in ファイルの修正

カーネルモジュールパッケージのソースパッケージは　2 つの control ファイルを持ちます。一つは、元のソースパッケージと、作成されるカーネルモジュールソースパッケージの情報が書かれた control ファイル、もう一つは、カーネルモジュールパッケージの情報が書かれた control.modules.in ファイルです。ソースパッケージ → カーネルモジュールソースパッケージ → カーネルモジュールパッケージの順で作成されるので、ユーザは リポジトリからカーネルモジュールソースパッケージを取得し、module-assistant を使って実際のカーネルモジュールパッケージを作成します。そして、ユーザは カーネルモジュールパッケージをインストールして利用します。

このファイルの中ではカーネルのバージョンによって置換される文字列が _KVERS_ という文字列になっています。**control.modules.in** は **module-assistant** などにより処理され、各 Linux カーネルバージョン用の内容に変更されます。

以下に debian/control.modules.in ファイルの例を示します。

```
Source: vloopback
Section: graphics
Priority: optional
Maintainer: Nobuhiro Iwamatsu <iwamatsu@nigauri.org>
Build-Depends: debhelper (>= 7), bzip2
Standards-Version: 3.8.0.1

Package: vloopback-modules-_KVERS_
Architecture: any
Provides: vloopback-modules
Description: vloopback modules for Linux (kernel _KVERS_).
 This package contains the set of loadable kernel modules for the
 <description>.
 .
 This package contains the compiled kernel modules for _KVERS_
 .
 If you have compiled your own kernel, you will most likely need to build
 your own vloopback-modules. The vloopback-source package has been
 provided for use with the Debian's module-assistant or kernel-package
 utilities to produce a version of vloopback-modules for your kernel.
```

4.6.7 debian/rules ファイルの修正

dh_make で作成された、`debian/rules` ファイルは パッケージ作成用のターゲット (build/install/clean) と、ドライバパッケージ作成用のターゲット binary-modules/kdist_clean が用意されています。これらのターゲットの中を修正します。

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

psource:=vloopback-source
sname:=vloopback

# prefix of the target package name
PACKAGE=vloopback-modules
# modifieable for experiments or debugging m-a
MA_DIR ?= /usr/share/modass
# load generic variable handling
-include $(MA_DIR)/include/generic.make
# load default rules, including kdist, kdist_image, ...
-include $(MA_DIR)/include/common-rules.make

kdist_config: prep-deb-files
kdist_clean: clean
	$(MAKE) $(MFLAGS) -f debian/rules clean

configure: configure-stamp
configure-stamp:
	dh_testdir
	# Add here commands to configure the package.

	touch configure-stamp

build-arch: configure-stamp  build-arch-stamp
build-arch-stamp:
	dh_testdir

	# Add here command to compile/build the package.
	# $(MAKE)

	touch $@

#k = $(shell echo $(KVERS) | grep -q ^2.6 && echo k)

# during a normal build
binary-modules:
	dh_testroot
	dh_clean -k
	dh_installdirs lib/modules/$(KVERS)/misc

	# Build the module
	$(MAKE) KVER=$(KVERS) KSRC=$(KSRC)

	# Install the module
	cp vlooopbackko debian/$(PKGNAME)/lib/modules/$(KVERS)/misc

	dh_installdocs
	dh_installchangelogs
	dh_compress
	dh_fixperms
	dh_installdeb
	dh_gencontrol -- -v$(VERSION)
	dh_md5sums
	dh_builddeb --destdir=$(DEB_DESTDIR)
	dh_clean -k
```

```
build-indep:  configure-stamp build-indep-stamp
build-indep-stamp:
        dh_testdir
        touch $@

build: build-arch build-indep

clean:
        dh_testdir
        rm -f build-arch-stamp build-indep-stamp configure-stamp
        dh_clean

install: DH_OPTIONS=
install: build
        dh_testdir
        dh_testroot
        dh_clean -k
        dh_installdirs

        # Create the directories to install the source into
        dh_installdirs -p$(psource)  usr/src/modules/$(sname)/debian

        # Copy only the driver source to the proper location
        cp vloopback.c  debian/$(psource)/usr/src/modules/$(sname)/.
        # Copy the needed debian/ pieces to the proper location
        cp debian/*modules.in* \
                debian/$(psource)/usr/src/modules/$(sname)/debian
        cp debian/rules debian/changelog debian/copyright \
                debian/compat debian/$(psource)/usr/src/modules/$(sname)/debian/
        cd debian/$(psource)/usr/src && tar c modules | bzip2 -9 > $(sname).tar.bz2 && rm -rf modules

        # Add here commands to install the package into debian/vloopback.
        # $(MAKE) DESTDIR=$(CURDIR)/debian/vloopback install

        dh_install

# Build architecture-independent files here.
# Pass -i to all debhelper commands in this target to reduce clutter.
binary-indep: build install
        dh_testdir -i
        dh_testroot -i
        dh_installchangelogs  -i
        dh_installdocs -i
        dh_installexamples -i
        dh_installman -i
        dh_link -i
        dh_compress -i
        dh_fixperms -i
        dh_installdeb -i
        dh_gencontrol -i
        dh_md5sums -i
        dh_builddeb -i

# Build architecture-dependent files here.
binary-arch: build install
        dh_testdir -s
        dh_testroot -s
        dh_installdocs -s
        dh_installinfo -s
        dh_installchangelogs  -s
        dh_compress -s
        dh_fixperms -s
        dh_installdeb -s
        dh_shlibdeps -s
        dh_gencontrol -s
        dh_md5sums -s
        dh_builddeb -s

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure binary-modules \
        kdist kdist_configure kdist_image kdist_clean
```

4.7 パッケージのビルドおよびインストール

パッケージのビルドを行う際には、通常のパッケージ作成と変わりません。debuild などを使ってパッケージの作成を行ってください。作成されたパッケージをインストールすると、`/usr/src/modules` にドライバソースイメージがインストールされます。

```
% debuild
% sudo dpkg -i ../vloopback-source_1.1.2-1_all.deb
```

4.8 モジュールパッケージの作成

4.8.1 module-assistant を使ったコンパイル

ドライバパッケージのコンパイルには module-assistant コマンドを利用します。このコマンドは GUI を持っており、GUI でドライバのコンパイルやコンパイル環境のアップデートを行う事ができます。GUI で選択できるドライバ群は `module-assistat` パッケージ内で持っているリスト (**??**) のみが対象になっているため、新しく作成したものは選択できません。このような場合は、モジュール名を指定することによってコンパイル可能です。コンパイルした後には、**??** ディレクトリ以下にカーネルモジュールパッケージが作成されるので、このパッケージをインストールすることによって、ドライバがコンパイルした Linux カーネルのバージョン上で利用できるようになります。また、module-assistant は作成したパッケージの自動インストール機能[*8]もあります。

```
$ sudo m-a build vloopback
Extracting the package tarball, /usr/src/vloopback.tar.bz2, please wait...
/usr/bin/make clean
make[1]: ディレクトリ `/usr/src/modules/vloopback' に入ります

<snip>

make[3]: ディレクトリ `/usr/src/linux-headers-2.6.25-2-686' に入ります
  CC [M]  /usr/src/modules/vloopback/vloopback.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /usr/src/modules/vloopback/vloopback.mod.o
  LD [M]  /usr/src/modules/vloopback/vloopback.ko
make[3]: ディレクトリ `/usr/src/linux-headers-2.6.25-2-686' から出ます
make[2]: ディレクトリ `/usr/src/modules/vloopback' から出ます
# Install the module
cp vloopback.ko debian/vloopback-modules-2.6.25-2-686/lib/modules/2.6.25-2-686/misc
dh_installdocs
dh_installchangelogs
dh_compress
dh_fixperms
dh_installdeb
dh_gencontrol -- -v1.1.2-1+2.6.25-6
dh_md5sums
dh_builddeb --destdir=/usr/src
dpkg-deb: `/usr/src/vloopback-modules-2.6.25-2-686_1.1.2-1+2.6.25-6_i386.deb' にパッケージ \
 `vloopback-modules-2.6.25-2-686' を構築しています。
dh_clean -k

<snip>
```

4.8.2 make-kpkg を使ったコンパイル

module-assistant 以外にも make-kpkg の –added-modules オプションを使うことによってコンパイルすることができます。しかし、make-kpkg の場合にはドライバソースイメージを展開してくれないので、手動で展開する必要があります。kernel-package はカーネルパッケージ専用と考えたほうがよさそうです。(そのために module-assistant ができたので。)

4.9 まとめ

今回は dh_make からの作成でしたが、cdbs を使うともっと簡単かつ簡潔に作成することができます。過去の勉強会資料にちょっとだけ載っているので、参考にしてください。しかし、カーネルドライバ側の `debian/rules` は cdbs での対応が行われていないので、対応する必要があるでしょう。あと、最近はドライバも linus ツリーに取り込まれやすくなったのですが、Linux のリリース間隔などを考えると、新しいドライバは簡単に使えるものではありません。新しいドライバを早く使いたい人には、ドライバパッケージ群が重要なパッケージになってくると思います。そのためには、今後はこれらのパッケージをどのようにメンテナンスしていくかなどについて考えていく必要がありそうです。

[*8] a-i オプション

4.10 おまけ

module-assistant でサポートされているパッケージがどれぐらいコンパイルできるのか調べてみました。アーキテクチャは i386 、ディストリビューションは sid です。パッケージは **??** にかかれているものです。結果は 82 個中、コンパイル OK: 22/ コンパイル NG: 22/ パッケージなし: 38 になりました。パッケージなしなのは、module-assistant のリストがメンテナンスされていなからだと思います。あとは、アーキテクチャ非対応など。既にカーネルにマージされているものがけっこうあり、コンパイルエラーは各メンテナがサボっているのでしょう。これらは今後 BTS していくつもりです。

5 みんなも Debian GNU/Hurd を使おうよ！

山本 浩之

いきなり実機で Hurd を使いたい、と言う人は稀でしょうし、Hurd のインストーラ自身も問題を抱えているらしく、私も実機への直接のインストールは成功していません。しかし仮想環境でなら、気軽で、簡単に使えます。今日は Debian GNU/Hurd の VMware 上での利用を紹介してみます。

まず、インストールの様子を見てみましょう。インストーラは、かなり昔の debian-installer を彷彿とさせ、懐かしむ人もいるでしょう。実際、woody の d-i をベースとしており、最近の d-i (lenny beta2 など) へは付いて行けてはいません。

Hurd のインストーラ CD は Linux カーネルで動いており、その中の /target として Hurd 環境を展開していきます。残念ながら woody は kernel 2.2.20 だったため、128 GB 以上のディスク (Big Drive) への対応はされていません。また、私が試した限りですが、30 GB より大きなパーティションには正常にインストールできないようです。実際、40 GB と 60 GB のディスクイメージにインストールした場合、インストールは終わったように見えますが、40 GB の時は Hurd がシングルユーザでもブートせず、60 GB の時はなぜかシングルユーザによる最初のブートはできましたが、ファイルのパーミッションが？ ---------- となっていてアクセスできない状態でした。

では実際に 30 GB のディスクイメージにインストールしてみましょう。まずインストール CD イメージからブートし、キーボードの選択をしたら、パーティションを決めます。この時点ではカーネルは Linux のままですので、hda1 とか hda5 とかの見慣れたパーティション指定ができます。この時、swap を必ず作成して下さい。swap を作ることを公式にも推奨されていますが、私が swap を作らずにインストールしたところ、まったくブートしませんでした。ここでは swap を取った残りを全て / としておきます。Hurd の対応フォーマットは、残念ながらいまだに ext2 のみです。パーティションを決めたら、フォーマットします。swap は Linux と全く同じですが、/ は mke2fs を実行する際に -o hurd を引数として与えることだけ異なっています。フォーマットしたら base.tgz の展開です。このあたりは昔の d-i を経験したかたならお手のものでしょう。

ただ、Hurd の d-i としての問題点は、grub を直接インストールできないことにあります。Hurd に対応したブートローダは、今のところ、grub だけなのですが、インストール CD には grub を入れるメニューがありません。これは既に Linux を入れていて grub をインストール済みなかたが多いためなのか、woody の d-i の問題点なのかは知りません。そこで、grub の FD あたりを用意しておき、別途 grub をインストールしなくてはなりません。最近は FDD を持たないマシンが増えており、このあたりは改善すべき点だと思います。

grub をインストールし、menu.lst には

```
default=0
timeout=10

title  Debian GNU/Hurd
root (hd0,0)
kernel (hd0,0)/boot/gnumach.gz root=device:hd0s1
module (hd0,0)/hurd/ext2fs.static --multiboot-command-line=${kernel-command-line} \
--host-priv-port=${host-port} --device-master-port=${device-port} \
--exec-server-task=${exec-task} -T typed ${root} $(task-create) $(task-resume)
module (hd0,0)/lib/ld.so.1 /hurd/exec $(exec-task=task-create)
boot

title  Debian GNU/Hurd (single user)
root (hd0,0)
kernel (hd0,0)/boot/gnumach.gz root=device:hd0s1 -s
module (hd0,0)/hurd/ext2fs.static --multiboot-command-line=${kernel-command-line} \
--host-priv-port=${host-port} --device-master-port=${device-port} \
--exec-server-task=${exec-task} -T typed ${root} $(task-create) $(task-resume)
module (hd0,0)/lib/ld.so.1 /hurd/exec $(exec-task=task-create)
boot
```

と書きます。 (hd0,0)/boot/gnumach.gz がカーネルで、hd0s1 とは Linux の hda1 (プライマリマスタ、第一パーティション) のことです。Hurd の / には /hurd と言うディレクトリが存在しており、この中に Hurd 特有のモジュールなどが存在しています。ここでは (hd0,0)/hurd/ext2fs.static と言う ext2 フォーマットを読み込むモジュールが指定されています。

ここまで行けば、インストーラを再起動し、まずシングルユーザで Hurd をブートし、

```
# ./native-install
```

で /native-install スクリプトを実行、再度シングルユーザで再起動、再度 /native-install スクリプトを実行で終わりです。

ここまでの工程が面倒なかたには、`debian-hurd-k16-qemu.img.tar.gz` と言う `qemu` 用の素晴らしいイメージが配布されていますので、これが利用できます。特に FDD が無くて grub のインストールが面倒なかたにお薦めです。

```
$ wget http://ftp.debian-ports.org/debian-cd/K16/debian-hurd-k16-qemu.img.tar.gz
$ tar zxvf debian-hurd-k16-qemu.img.tar.gz
$ qemu-img convert debian-hurd-k16-qemu.img -O vmdk
```

さて、インストールも終わったところで、実際にブートしてみましょう。普通にブートすれば

```
login >
```

とプロンプトが出てきますので、'login root' でログインします。最初はパスワードはかかっていません。

まずはとにかく、ネットワークに繋ぐために設定します。Hurd には /sbin/ifconfig などが無く、settrans コマンドで設定します。

```
# settrans -fgap /servers/socket/2 /hurd/pfinet -i eth0 -a 192.168.1.3 -g 192.168.1.1 -m 255.255.255.0
```

あとは Linux と同様に、/etc/resolv.conf に

```
nameserver 192.168.1.1
```

とか書いてやるだけでネットワークに繋がります。

Debian GNU/Hurd のユーザランドは Debian そのもので、ほとんどは Linux の知識で間に合います。ただし、まだディベロッパやメンテナなどに Hurd が浸透しておらず、FTBFS (Failure To Build From Source) が多数あり、多くのパッケージが使えない状態です。特に多いエラーは、PATH_MAX (MAXPATHLEN) の未定義エラーです。Hurd には Linux などと違い、パスの最大文字数を制限すると言う概念が無いらしく、文字どおり未定義となっております。ディベロッパやメンテナの人には、自分のパッケージが Hurd でビルドできるか、是非試していただきたいものです。

Let's enjoy Hurd!

6 Debconf 8

岩松 信洋

勉強会として、今回の Debconf 開催地である アルゼンチンから IRC を使って行われました。内容を以下に報告します。

6.1 当日のアジェンダ

1. 22:00- Debconf 参加者 の 今回の意気込み
2. 22:30- Debconf 参加者 への Debconf 質問コーナー

6.2 参加者

- debconf 側
  上川さん (dancer)
- 日本側 kmuto, henrich, yamamoto, aya, honjo, risou, mkouhei, ake, iwamatsu

6.3 参加者への質問

Debconf 参加者への質問ということで、以下の質問があり、参加者から回答を得ることができました。質問内容 (質問者名) という形式になっています。

1. 無事にたどりつけましたか (kmuto)
   予約待ちしていたバスが予約できていなかったので、retiro 経由でバス 8 時間、合計 40 時間の行程でした。今回しかも、san francisco からの旅程なので、日本からいってたら死ねる (dancerj)
2. wife 様が geek どもを見て一言 (匿名)
   「ほどほどにね」
3. 日本からは (そもそも) 誰が行っているのですか？(AyaKomuro)
   上川さん夫妻です（奥様可哀想と言う声もあるような…「アルゼンチンに旅行にいく、としか説明していなかった …」）
4. 今回の Debconf で期待しているセミナー/レクチャーは何ですか？(AyaKomuro)
   「基本的に期待していないので、Hacklab にずっといるつもりです。」
5. Debconf@Argentina ならではのこれは !? を教えてください (AyaKomuro)
   - keysigning party に出ないつもりなので個別に keysigning をする。
   - qemu maintainer group に強制収容された、qemubuilder ハックした
   - あと Don armstrong をつかまえて BTS SOAP のインタフェースのバグを直してもらうかな

6. 日本で debconf 開いたら来たいよーって人はどのくらいいるのでしょうか？ (henrich)
   「明日、19:00 から debconf10 会議がある。」
7. ごはんはおいしいのかな (kmuto)
   「ごはんうまいっす。かなり。いけてる。朝飯からケーキ。牛メインですね。港町なので魚もある。野菜はすくな目かな。パンはそうでもないです。debconf の会場のそとで食事したら多分肉肉してるんだろうけど、ホテルの食事はそうでもないです。」
8. 日本の Debian 関係者に何か一言ないか。(henrich)
   無回答
9. Lenny で DDTP のエンコードによる文字化け問題は解決の見込みがあるのか (henrich)
   無回答
10. unzip の表示がおかしくなる問題について、Ubuntu のメンテナとの協業で解決してはもらえないのか。大変この問題を憂いている。(henrich)
    ack.
11. Ubuntu のような LoCo (Local Community) などは検討していないのか? (henrich)
    無回答

7 Debian 温泉

岩松 信洋

2008 年 8 月 16 日は Debian 15 周年です。おめでとう！ Debian 勉強会では、15 周年を祝うために有志で Debian 温泉と題した 合宿を行いました。以下に合宿レポートを紹介します。

7.1 Debian 温泉の日程、場所、参加者

今回は、Debian 勉強会に参加された方から希望者を募って実験的に合宿を行う方式を取りました。当初は伊豆の伊東温泉で行う予定でしたが、予定していた場所を予約できず、急遽、草津温泉に変更なりました。伊東温泉を楽しみにしていた方、申し訳ございませんでした。

天気は雨。大雨に見舞われましたが、温泉にひきこもってハックするだけなので天気はあまり関係ありません。しかし、一部の人が**なぜか**雨で濡れたり、着替えをもってこなかったために最悪な状態になってしまったようです。民宿の方がタオルを貸してくれたり、濡れてしまった靴から水を吸い出すために新聞紙を提供してくれたおかげで、帰るころには乾いていたようです。民宿の方ありがとうございました。（見てないと思うけど。）

日程	2008 年 8 月 16 日 - 8 月 17 日
場所	草津/草津温泉
利用したところ	民宿　美山 [*9]
宿泊費	8000 円（夕食/朝食/入湯税込み）
参加者	henrich, yamamoto, mkouhei, nori1, tks, hibino, ake ,honjo, iwamatsu

表 1　Debian 温泉の日程、場所、参加者

7.2 行われたこと、合宿成果

今回の合宿の目的は以下のとおりです。

- Debian 15 周年を祝う
- 温泉に入る
- Debian 開発
- Debian 勉強会に関してのグループワーク/ワークショップを行う
- Debian JP Project 理事 打ち合わせ

また、開発結果として、以下のものが行うことができました。

名前	開発結果
henrich	po-debconf 翻訳
yamamoto	FDClone パッケージメンテナンス
mkouhei	MacBook Air まわりのハック
nori1	Debian パッケージメンテナンス
tks	V4L まわりのハック
hibino	ネットワークの設定, ocaml パッケージメンテ?
ake	よく寝た。
honjo	エミュレータハックネタの発表
iwamatsu	cairodock パッケージ化, uvc まわりのパッケージ化、翻訳、U-Boot メンテナンス

表 2　合宿での成果

7.3　温泉ワークショップ

名前 _____

下記の空欄を埋めてください:

現在 Debian 勉強会に足りないものは (　　) です。これを解決するために、私は勉強会で (　　) を企画します。
これを行うことによって (　　) を解決することができます。

企画案の図：

7.4 ワークショップの結果

7.4.1 メンテナになるためのもう一押しをやります！/やまねさん

- 月刊メンテナ報告
  メンテナ活動、やっている内容の説明
- メンテナンスの報告
- WNPP の報告
  WNPP のピックアップ

7.4.2 新人勧誘 をやります！/やまもとさん

- ビデオカンファレンス
  やっていることを動画として残す。何をやっているのかが、わかりやすくなります。

7.4.3 ユースケースの紹介をします！/ひびのさん

- Debian 私の会社の場合 という説明
  ユーザーとデベロッパのギャップを解決できる。
  DD とユーザーの方向性が違うので、汲み取りにくい。
  これらを解決する。

7.4.4 若者を集めます！/あけどさん

- Debian の魅力、すばらしさなどを伝え、とりあえず盛り上がる場を作る。仕組みをつくる。
  若者に近いコミュニティと話をしたりする。大学の研究室、東大のコンピューター関係の研究室に相談。学園祭などでやるのはどうか？

7.4.5 社内での勉強会を行います！/前田さん

- 勉強しようとしている人と各ディストリの違いの説明と誤解を解きます。
  Debian と Debian 勉強会の広報活動。知名度をあげるだけではなく、誤解を解く活動を行います。利点と欠点をうまく説明。企業ユーザなどにアピール。アピールすると事例が出てくるので、これをフィードバックとして取り入れる。

7.4.6 パッケージ作成大会を行います！/小林さん

- パッケージのしきいの高さを低くする。
  実際に手を動かす作業が足りない。パッケージハンズオンをもうちょっと行う。

7.4.7 Web 系との勉強会を行います！/すずきさん

- 新人さんが少ない Web 系での存在感がすくない。
  Web 系から問題を提出してもらう。Debian でやる RoR など。Web 系では若いひとが多いので、新人が呼び込める。あと、ユーザーが増える可能性がある。Web 系のユーザーは実際はエンドユーザーになっている。

7.4.8 サーバを立てて使う人を取り込みます！/ほんじょうさん

- 会社から家に SSH でログインして、作業する人たちを呼び込む。
  家にサーバに立ててておこなってみましょう。初心者ディストリユーザーが増える可能性がある。デスクトップを捨てる？中古の PC を使ってサーバーをやる。仕事でやっているんだけど、家でもやりたい。家で環境を構築するためにはどうしたらいいのか、をサポートする。

7.4.9 その他

- Windows + Debian の使い方
- Qemu で Debian
- デザインが足りません。
- non-free なドライバを入れるためのプロジェクトを作ってユーザライクに。
- 勉強会の課題がむずかしい。もうちょっとユースケース寄りの話にしたらどうか。

7.5 かかった費用など

以下に今回かかった費用などを報告します。

項目	費用
交通費（バス）	5600 円
宿泊費（夕食/朝食/入湯税込み）	8000 円
お酒とか	各自負担

表 3　費用など

7.6 各自の感想や課題など

今回の合宿で分かった各自の感想などをまとめました。

7.6.1 iwamatsu

- ネットワーク設備に耐えられる環境を先に用意しておく
  今回は、無線 LAN が提供されている宿を利用することができましたが、大人数が利用するとネットワークの反応が悪くなるという問題がありました。今回は hibino さんのマシンがネットワークブリッジおよび DNS キャッシュサーバになることにより回避することができましたが、今後行うためにはこれらが行える設備を用意しておくといいことがわかりました。安いルータを DD/Open-Wrt あたりを使ってブリッジが使えるようにしておくといいと思っています。
- Debian ミラーサーバの設置
  Debian のミラーサーバがあると、特にインターネットにつながっていなくても、とりあえずは Debian の開発をすることが可能です。今回は iwamatsu がミラーサーバを持っていこうと思っていましたが、思ったより rsync に時間がかかり持っていくことができませんでした（i386/amd64/source + stable + testing + unstable で 5 日かかった）。今回は無理でしたが、次回から毎日 sync しているミラーサーバを持っていくことが可能になりました。
- マシンの整備
  毎回ネタになるのですが、一部のマシンがハードウェアによる障害でネットワークにつなげることができませんでした。合宿を行う場合は、ネットワークがない場合を想定した自分の環境の整備を行うようにしましょう。

また、マシンの整備を怠らないようにしましょう。

- 着替え
  予備の着替えは持ってくるようにしましょう。
- 騒音の問題
  他の宿泊者に迷惑をかけないようにしましょう。声が大きい人は気をつけましょう。(今回は何も言われなかったけど。)
- おやつ
  うまい棒必須。

7.6.2 Henrich

- 感想
  勉強会でも集まることは集まれるのですが、狭い部屋でのワークショップが存外に楽しかったです。普段ネット越しで文字情報のみでやり取りしていた「もやもやした部分」を、あまりアルコールが入らない状態でディスカッションでクリアにしていくのは長足の進歩だと思います。
  また、今後に向けての課題が見えてきたことはモチベーションの維持向上に繋がりました。
- 今後の課題（温泉合宿に限って言うと）
  - もう少し楽に移動できる温泉地を選ぶ
  - 1 泊のみだと、本当に精根尽き果てて終わってしまうので 2 泊 3 日とかを検討する
  - 記憶は揮発性なので、録音とか録画とかしておけると良い。ただし、ストリーミングは不要だと個人的には思う（温泉まで来れた人だけの特典）
- 今後の課題
  私の課題は**パッケージメンテナになる為のもうひと押し** で勉強会で**パッケージのメンテナンス情報などの報告**を企画し、**不活性なメンテナによって不幸にもメンテナンスされていないパッケージが引き起こす諸問題やリソースの割り当て問題の解決**を目指します。
- 裏課題
  事例集を集めて、常に情報を公開し、「それ、Debian で出来ますけど、何か？」を言えるようにしておく :-)

8 Po4a でドキュメント翻訳の保守を楽にしよう

小林 儀匡

翻訳の保守は新規翻訳よりも手間のかかる作業になりがちです。しかし、これを怠ると原文との乖離は大きくなり、折角の翻訳も価値が下がっていきます。Debian で開発されている Po4a は、プレーンテキスト、XML、HTML、LaTeX、nroff (man) など様々な形式のドキュメント翻訳を PO という形式で管理し、保守性を上げるツールです。今回はこの Po4a の使い方について説明し、また PO 関連の予備知識を提供します[*10]。

8.1 はじめに ～地味で地道なドキュメント翻訳保守作業～

長期的に見れば、ドキュメント翻訳者の作業で最も大変で最も大切なのは、ドキュメントの翻訳作業ではなく翻訳の**保守**作業でしょう。何週間、何ヶ月もの作業の末、1 万行を超える長いドキュメントの翻訳作業を終えたとしても、原文が変わり続ける限り、保守をしなければ翻訳の価値は下がっていきます。

保守は地味で地道な作業です。原文に対しては、次のような変更が発生します。

- 訳文には影響のない typo の修正
- 記述の追加・変更・削除 (修正を含む)
- パラグラフの位置の変更
- マークアップの変更 (マークアップ言語で書かれたドキュメントの場合)

同時に、次のような非本質的な変更も頻繁に発生します。

- パラグラフのインデントの変更
- パラグラフの改行位置の変更 (主に語句の挿入・変更・削除に起因する)

こういった様々な種類の変更は大抵は混ざっています。ドキュメントの翻訳者は通常、`diff` で差分を眺めて変更を把握し、それを翻訳に反映させます。差分を読んで変更を把握する能力によって、翻訳に反映させる作業が楽にも大変にもなります。

このようなドキュメント翻訳の保守作業に必要な労力を和らげてくれるのが Po4a です。Po4a は、様々な形式のドキュメント翻訳を PO という形式で一括して管理し、保守性を上げるツールです。以下では、まず PO というファイル形式が、翻訳にとってどのように便利なのかを説明し、その上で Po4a がどのようなツールなのか説明します。

[*10] 新たなドキュメント形式を PO で管理するために知っておくべき内部構造について説明しようと思ったのですが、PO の説明を書いていたら余裕がなくなってしまったので、内部構造についてはまたの機会に。

8.2 PO

PO は、プログラムのメッセージの翻訳のために作られたファイル形式です。ここではその書式や編集用のツールについて見ていきましょう。

8.2.1 PO の基本的な書式

まずは、PO の基本的な書式を説明します。

PO には、原文と訳文の対からなるエントリが空行区切りで多数収められています。最初に、簡単なエントリの例を紹介します。

```
#: src/apt_config_treeitems.cc:99
msgid ""
"%BOption:%b  %s\n"
"%BDefault:%b %s\n"
"%BValue:%b   %s\n"
msgstr ""
"%B オプション:%b %s\n"
"%B デフォルト:%b %s\n"
"%B 値:%b         %s\n"
```

一目瞭然ですが、msgid から始まる一連の行、msgstr から始まる一連の行の、"で囲まれた部分は、それぞれ原文と訳文です。

「#」で始まる行は基本的にすべてコメントです。特に「#:」で始まる行は、原文を抽出した位置を示す参照用の行です。これは xgettext がメッセージを抽出して POT を生成するときに設定します。翻訳者は、msgid を見ても理解できない場合、この参照コメントをもとにしてソースコードを眺めることができます。

「#:」以外にも、様々な種類のコメントがあります。そのようなコメントを含んだエントリの例を紹介します。

```
# 最後の %s は「(core dumped)」。
#. ForTranslators: "%s update %s" gets replaced by a command line, do not translate it!
#: src/generic/apt/download_update_manager.cc:383
#, c-format
msgid "The debtags update process (%s update %s) was killed by signal %d%s."
msgstr ""
"debtags 更新プロセス (%s update %s) がシグナル %d に kill されました %s。"
```

「#.」で始まる行は、原文のメッセージと一緒にソースコードから抽出されたコメントです。主に、メッセージの翻訳に注意が必要な場合やメッセージが翻訳しにくい場合に、開発者から翻訳者への説明に使われます。これも xgettext がメッセージ抽出時に設定します。

「#,」で始まる行はフラグです。最もよく使われるフラグは fuzzy というフラグでしょう。これについては後述します。これは様々なツールが設定します。

それ以外の、「#」の後に何の記号もない行は、翻訳者が翻訳作業時に勝手に追加したコメントです。各エントリにはこのような翻訳者コメントを自由自在につけることができます。

このようなコメント情報は、プログラムのソースコードに書くコメントと同様、人間の作業にのみ必要となるものなので、MO に変換するときにすべて削除されます[*11]。

8.2.2 fuzzy エントリ

msgmerge が古い PO に新しい POT をマージする際、既に翻訳されているエントリの msgid に似た msgid を持つエントリが追加されることがあります。例えば、エントリ A の msgid に似た msgid を持つエントリ A' が追加されるとします。このとき、msgmerge は A をベースとして A' を翻訳できると判断し、A' の msgstr に A の msgstr の内容を設定した上で、A' に fuzzy フラグをつけます。このようにして作られるエントリが、次のような **fuzzy エントリ**です。

[*11] したがって、msgunfmt で MO を PO に変換すると、コメントをまったく含まない PO が得られます。

```
#: src/main.cc:181
#, fuzzy, c-format
msgid ""
" -q             In command-line mode, suppress the incremental progress\n"
"                indicators.\n"
msgstr " -q                コマンドラインモードで進行状況を逐一表示しません。"
```

しかしこれだけだと、翻訳者には、msgid がどう変化したのか分かりません。そこで GNU gettext のバージョン 0.16 から導入されたのが、次のような、「#|」で始まるコメントです。

```
#: src/main.cc:181
#, fuzzy, c-format
#| msgid ""
#| " -q             In command-line mode, suppress the incremental progress "
#| "indicators."
msgid ""
" -q             In command-line mode, suppress the incremental progress\n"
"                indicators.\n"
msgstr " -q                コマンドラインモードで進行状況を逐一表示しません。"
```

「#|」で始まるコメントは、以前の msgid です。msgmerge に--previous を指定した場合に設定されます。このコメントを使えば、以前の msgid と現在の msgid を比較できるので、どのような変更を msgstr に加えれば翻訳を現在の msgid に対応できるのかが簡単に分かります。PO の保守性を上げる仕組みと言えます。msgmerge に--previous が指定されていない場合は指定しておくとよいでしょう。

8.2.3 ヘッダ

翻訳を扱う際、翻訳者やその連絡先、原文の問題の連絡先、翻訳の更新日時、文字セットなどは重要なメタ情報です。PO では、最初のエントリを**ヘッダ**として使用し、そのエントリの msgstr にこれらのメタ情報を含めることになっています。ヘッダの msgid は空文字列にすると決まっています。

また、著作権情報など、書式が曖昧な情報や長い情報は、ヘッダのコメント (つまりファイルの冒頭) に書くことになっています。

以下は、aptitude の ja.po の例です。

```
# Japanese translations for aptitude
# aptitude の日本語訳
# Copyright (C) 2006-2008 Noritada Kobayashi <nori1@dolphin.c.u-tokyo.ac.jp>.
# This file is distributed under the same license as the aptitude package.
#
msgid ""
msgstr ""
"Project-Id-Version: aptitude 0.4.1\n"
"Report-Msgid-Bugs-To: aptitude@packages.debian.org\n"
"POT-Creation-Date: 2008-09-05 16:05+0200\n"
"PO-Revision-Date: 2008-05-16 03:35+0900\n"
"Last-Translator: Noritada Kobayashi <nori1@dolphin.c.u-tokyo.ac.jp>\n"
"Language-Team: Japanese\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=1; plural=0;\n"
```

なお、ここでは省略しましたが、筆者は、メッセージカタログ内での用語統一を容易にするため、aptitude や Subversion など大規模な PO では、ヘッダのコメントに対訳表を含めています。

8.2.4 その他の書式

ここまでで、PO のエントリの書式についてざっと見ました。

PO のエントリについては、msgid と msgstr から成るもの以外にも、複数形への対応として msgid、msgid_plural、msgstr[n] から成るものがあります。詳しくは GNU gettext のドキュメント[*12]の『3 The Format of PO Files』を参照してください。

[*12] gettext-doc パッケージ。

8.2.5 PO 編集ツール

PO で翻訳作業を行うには PO 編集ツールが便利です。PO はテキストなのでどんなテキストエディタでも編集できますが、翻訳に含まれる「"」や「\」を手でエスケープするのは厄介です。そのような機械的な処理は、PO に特化した PO 編集ツールに任せるのがよいでしょう。また、PO 編集ツールには翻訳支援機能を持つものもあるので、作業効率も上がるはずです。

PO 編集ツールとしてよく知られているのは、Emacs の PO 専用メジャーモードである po-mode、KDE の PO 編集スイートである KBabel、GNOME の PO エディタである Gtranslator、そして poedit です。

ここでは、筆者が慣れている po-mode について、筆者がよく使う操作だけざっと説明します。説明中で何度も登場する「選択エントリ」とは「現在カーソルがあるエントリ」の意味です。

まず、エントリ間の移動には以下のようなキー操作が使えます。

`n` 次のエントリに移動します。
`p` 前のエントリに移動します。
`t` 次の既訳エントリに移動します。
`T` 前の既訳エントリに移動します。
`f` 次の fuzzy エントリに移動します。
`F` 前の fuzzy エントリに移動します。
`u` 次の未訳エントリに移動します。
`U` 前の未訳エントリに移動します。
`o` 次の obsolete エントリに移動します。
`O` 前の obsolete エントリに移動します。
`m` 選択エントリの位置をスタックに push します。
`r` スタックから pop して得られたエントリに移動します。

また選択エントリの操作には以下のようなキー操作が使えます。

`RET` 編集用バッファを開いて、選択エントリの `msgstr` を編集します。
`#` 編集用バッファを開いて、選択エントリの翻訳者コメントを編集します。
`k` 選択エントリの `msgstr` をカット (kill) します。
`K` 選択エントリの翻訳者コメントをカット (kill) します。
`w` 選択エントリの `msgstr` をコピーします。
`W` 選択エントリの翻訳者コメントをコピーします。
`y` 選択エントリの `msgstr` にペースト (yank) します。
`Y` 選択エントリの翻訳者コメントにペースト (yank) します。
`TAB` 選択エントリの fuzzy フラグを取り除きます。
`DEL` 選択エントリが既訳の場合は fuzzy フラグをつけます。未訳または fuzzy の場合は obsolete にします。obsolete の場合は削除します。
`C-j` 選択エントリの `msgid` の内容を `msgstr` にコピーします。
`s` 選択エントリの参照コメントをもとにソースコードの該当行を表示します。参照コメントに複数の位置が記述されている場合は、順番に表示します。

編集用バッファでは通常の Emacs の編集操作ができます。特殊な操作は以下のとおりです。

`C-c C-c` 編集用バッファの内容を `msgstr` または翻訳者コメントの内容として設定し、バッファを閉じます[*13]。
`C-c C-k` 編集用バッファの内容を破棄してバッファを閉じます。

[*13] もちろん、この変更はメインバッファで保存操作を行うまで保存されません。

最後に、メインバッファでの PO 全体に関わる操作は以下のとおりです。

_　行った変更を取り消します。

?, h　ヘルプを表示します。

V　msgfmt に--check (-c) をつけて実行し、PO に問題点がないか確認した上で保存します。例えば、msgid が改行記号で終わっているのに msgstr が改行記号で終わっていない場合、それが報告されます。

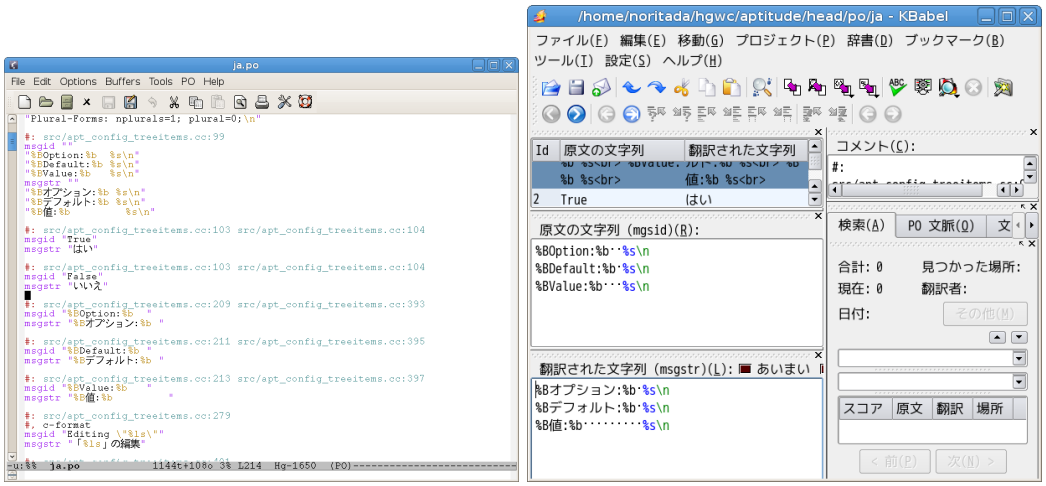

図 3　po-mode と KBabel

以上の簡単な説明から分かると思いますが、po-mode は移動や編集がしやすく、非常に便利です。

ただ、po-mode には今のところ翻訳メモリのような機能はありません。KBabel や Gtranslator は翻訳メモリを備えているので、翻訳メモリに興味のあるかたはこれらを試してみるのもよいでしょう。

8.2.6　GNU gettext ツール群

翻訳作業には PO 編集ツールを使うことになるでしょうが、一方で GNU gettext が提供する、PO や POT に対してコマンドラインから自動処理できるツール群も、知っておくと便利です。以下でざっと説明します。

msgattrib　エントリの状態に応じた一括処理をします。例えば、「fuzzy エントリのみにする」、「fuzzy エントリから fuzzy フラグを取り除く」といったことが可能です。

msgcat　複数の PO を繋げます。

msgcmp　PO と POT が含む msgid セットを比較します。

msgcomm　複数の PO に含まれる共通の msgid を抽出します。

msgconv　PO の文字セットを変換します。

msgen　英語の PO を作成します。入力ファイルの未訳のエントリの msgstr に、msgid と同じ文字列を設定します。

msgexec　PO に含まれるすべての翻訳に対してコマンドを実行します。

msgfilter　PO に含まれるすべての翻訳に対して共通の処理をします。

msgfmt　PO をコンパイルして MO に変換します。

msggrep　パターンにマッチしたエントリを PO から抽出します。

msginit　PO を初期化します。

msgmerge　PO に POT をマージします。

msgunfmt　MO から PO に逆コンパイルします。

msguniq　PO に含まれる重複した msgid を 1 つにまとめます。

xgettext　ソースコードから msgid を抽出して POT を生成します。

開発者が主に使うのは xgettext と msgmerge、msgfmt ですが、他のツールも使いこなせるようになっておくと便利です。使い方は各ツールのマニュアルページを参照してください。

8.2.7 まとめ

PO はプログラム翻訳のために作られたファイル形式で、参照コメントのような翻訳に有用な情報や、fuzzy エントリのような翻訳作業・保守作業の効率を上げるための仕組みを含んでいることを説明しました。また、PO という書式に特化した編集ツールについても説明しました。

8.3 ドキュメント翻訳で PO を使うという考え方

プログラム翻訳のために生み出された PO は、翻訳者にとって翻訳と保守の両方の作業をしやすい環境を提供する存在となりました。その理由としては以下のようなものが考えられます。

- PO 自体に翻訳と保守を支援するための仕組みがある上、翻訳支援機能を持つ PO 編集ツールも存在する。
- メッセージの管理はツールがやってくれるので、人間は翻訳の管理に集中していればよい。
  - メッセージがソースコードのどこにあるかが翻訳者にとって重要でないので、メッセージがソースコード内で移動しても翻訳者は気にする必要がない。
  - msgmerge で PO を更新すると、どのエントリの翻訳を更新すべきか瞬時に分かり、PO 編集ツールでそのエントリへ簡単に移動できる。

一方でドキュメント翻訳については、通常は原文をそのままコピーして翻訳作業を開始するため、翻訳以外の部分 (インデント、ドキュメント内での位置など) に変更があった場合、その影響を翻訳者が受けます。プログラム翻訳と比べて分量が多いのに保守性が低いので、結果として翻訳者への負担は非常に大きくなります。この状況は、翻訳者にとって大きな悩みの種でした。

そこで生まれたのが、ドキュメント翻訳に対しても PO を使おうという考え方です。具体的には、ドキュメントの各ブロックをエントリとした PO を作り、ドキュメントと PO とを相互変換できるようにすることで、翻訳者が PO での翻訳管理に集中できるようにします。以下のツールがそのような思想で作られています。

Po4a　Debian で開発されている[*14]。対象とするドキュメント形式は、プレーンテキスト、XML、HTML、LaTeX、nroff (man)、Pod など多数で、Perl で書かれたモジュールおよびプログラムの集合として実装されている。Debian パッケージは po4a。

poxml　KDE で、KDE SDK モジュールの 1 つのコンポーネントとして開発されている。対象とするドキュメント形式は DocBook XML で、C++ で実装された、po2xml、xml2pot などの実行可能プログラムから成る。Debian パッケージは poxml。

xml2po　GNOME で gnome-doc-utils の 1 つのユーティリティとして開発されている。対象とするドキュメント形式は DocBook XML である。Python で実装されたシンプルな 1 つのファイルから成るので、手軽にドキュメントのビルドに使用できる。Debian では gnome-doc-utils パッケージに含まれる。

poxml や xml2po が DocBook XML のみを扱うツールであるのに対し、Po4a が様々な形式をサポートしているのは、おそらく GNOME や KDE と Debian の立場の違いを反映しているのでしょう。GNOME や KDE はプロジェクト内でドキュメント形式を DocBook XML に統一できますが、ディストリビュータである Debian では、様々なソフトウェアの様々なドキュメントに対応する必要があるのです。

以降のセクションでは、ドキュメント翻訳に PO を使用する 3 つのツールのうち、唯一複数のドキュメント形式をサポートしている Po4a について見ていきます。

[*14] 厳密に書くと、Debian Project のメンバーが、同プロジェクトが提供しているフリーソフトウェアプロジェクトホスティングサービスである Alioth 上で、1 つのプロジェクトとして開発しています。

8.4 Po4a の基本

Po4a というソフトウェア名は、「po for anything」から来ています。名前からも分かるように、最初から様々なドキュメント形式の翻訳を PO で管理することを目的としており、そのために入力形式として複数の形式を扱うことを想定した作りとなっています。

例えば、以下のような形式が現在サポートされています。

KernelHelp　カーネル設定ヘルプのドキュメント形式です。

nroff　Unix の伝統あるマニュアルページの形式です。BSD マニュアルページで使用されている mdoc 形式もサポートされています。初心者が読み書きしにくい形式ですが、Po4a でサポートされたので、インラインのマークアップだけ分かれば (原文の真似をすれば) 翻訳できます。

POD　Perl 関連のドキュメントに使われる形式です。ソースコードの横にコメントとして書き込まれたドキュメントは翻訳しにくいですが、それを PO として抽出し、翻訳しやすくします。

SGML　最近では XML のほうが主流ですが、一昔前のドキュメント形式の主流です。現在サポートされている DTD は DebianDoc-SGML と DocBook SGML のものです。

TeX/LaTeX　著名な組版ソフトウェア LaTeX の形式です。

GNU Texinfo　GNU のドキュメントに使用される形式です。

XML　最近よく使われるドキュメント形式です。現在サポートされている DTD は Dia、DocBook XML、Guide XML、XHTML のものです。

サポートされている形式の一覧は、`po4a-gettextize`、`po4a-translate`、`po4a-updatepo` などのコマンドに `--help-format` オプションを与えると表示できます[*15]。コマンドに`--format` (`-f`) などを与えてドキュメント形式を指定する場合、この一覧に載っている名前を使用してください。

```
noritada[3:39]%  po4a-gettextize --help-format   terra:~/svnwc/build-common/doc
有効フォーマット一覧:
  - dia: 非圧縮 Dia ダイアグラム。
  - docbook: Docbook XML。
  - guide: Gentoo Linux の xml ドキュメントフォーマット。
  - ini: .INI フォーマット。
  - kernelhelp: 各カーネルのコンパイルオプションのヘルプメッセージ。
  - latex: LaTeX フォーマット。
  - man: 古き良きマニュアルページフォーマット。
  - pod: Perl オンラインドキュメントフォーマット。
  - sgml: debiandoc と docbook DTD の双方。
  - texinfo: info ページフォーマット。
  - tex: 汎用 TeX ドキュメント (latex を参照)。
  - text: シンプルなテキストフォーマット。
  - wml: WML ドキュメント。
  - xhtml: XHTML ドキュメント。
  - xml: 汎用 XML ドキュメント (docbook を参照)。
```

8.5 Po4a の使い方

Po4a を用いた翻訳および翻訳更新手順について説明します。

まず、一から文書を翻訳する場合は以下のようになります。

- 翻訳
  1. 原文ドキュメント →POT
  2. POT⇒PO [翻訳]
  3. PO+ 原文ドキュメント → 翻訳ドキュメント
- 翻訳更新
  1. 原文ドキュメント (更新版)→POT
  2. POT+PO (旧版)→PO (エントリ更新版)

[*15] 残念ながら今のところ `po4a` コマンドに与えることはできないようです。

3. PO (エントリ更新版) 翻訳更新
4. PO (エントリ更新版)+ 原文ドキュメント (エントリ更新版)→ 翻訳ドキュメント (エントリ更新版)

8.5.1 導入 (1): 原文ドキュメントファイルから POT を生成する

Po4a を使ってドキュメントの翻訳をする場合、最初にすべきことは、原文のドキュメントファイルから POT を生成することです。POT の生成には po4a-gettextize コマンドを使用します。-f オプションの引数にドキュメントファイルの形式を、-m オプションの引数に原文ドキュメントファイル (**マスタードキュメント**) の名前を、-p オプションの引数に出力する POT のファイル名を与えて、コマンドを実行します。-M オプションの引数で、原文ドキュメントファイルの文字セットを指定することも可能です。

特に問題がない場合はすんなりとコマンドの実行が終了し、POT が生成されます。ドキュメントの構造に問題がある場合 (例えば XML においてタグがきちんと閉じられていない場合) はエラーになります。

POT が生成されたら、翻訳作業はプログラム翻訳の場合と同じです。POT をコピーして PO を作成し、ヘッダを適切に設定した上で翻訳作業を始めましょう。

■例 次の例では、hoge.en.html という XHTML ドキュメントから hoge.pot を生成しています。

hoge.en.html (入力):

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
<title>Test file</title>
</head>
<body>
<h1>Test file</h1>
<p>This is an <a href="apple">apple</a>.</p>
<p>This is an <a href="orange">orange</a>.</p>
</body>
</html>
```

コマンドライン:

```
noritada[14:14]%  po4a-gettextize -v -f xhtml -m hoge.en.html -p hoge.pot
```

hoge.pot (出力):

```
# SOME DESCRIPTIVE TITLE
# Copyright (C) YEAR Free Software Foundation, Inc.
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2008-09-20 14:21+0900\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: ENCODING"

# type: Attribute 'xml:lang' of: <html>
#: hoge.en.html:4 hoge.en.html:4
msgid "en"
msgstr ""

# type: Content of: <html><body><h1>
#: hoge.en.html:6 hoge.en.html:9
msgid "Test file"
msgstr ""

# type: Content of: <html><body><p>
#: hoge.en.html:10
msgid "This is an <a href=\"apple\">apple</a>."
msgstr ""

# type: Content of: <html><body><p>
#: hoge.en.html:11
msgid "This is an <a href=\"orange\">orange</a>."
msgstr ""
```

8.5.2 導入 (2): 原文・翻訳ドキュメントファイルから PO を生成する

原文と同じ形式で翻訳したドキュメントが既にあり、それを Po4a 用の PO に移行したい場合にも、`po4a-gettextize` が使えます。「POT をコピーして PO を作った上で、翻訳ドキュメントの各ブロックをコピー・アンド・ペーストで PO の各エントリに埋め込む」という、間違いなくうんざりする作業は不要です。

ただしこの場合、どの原文がどの訳文に対応するかを Po4a が把握する必要があるので、**原文ドキュメントファイルと翻訳ドキュメントファイルが同じ構造である**ことが前提になります。「同じ構造」とは、Po4a が切り分ける単位、つまりブロックのレベルで見たときに、対応する要素が同じ順序で並んでいる、という意味です。もし翻訳側でブロックの追加や削除が行われていたら、変換はエラーで終わるでしょう。一部のブロックの順序が入れ替わっている場合、変換は見た目は無事に終わるかもしれませんが、入れ替わったブロックの `msgid` と `msgstr` の対応関係はおかしくなっているはずです。

訳注や翻訳者情報を別個のブロックとして翻訳ドキュメントに追加している場合、それは一旦取り除いてください。Po4a では、PO から翻訳ドキュメントを生成する際に、原文にはない要素を追加する方法が提供されています。方法については後述します。

原文ドキュメントファイルと翻訳ドキュメントファイルが同じ構造であれば、`po4a-gettextize` を用いた変換は成功するはずです。`-f` オプションの引数にドキュメントファイルの形式を、POT を生成する場合の一連のオプションに加えて、`-l` オプションの引数に翻訳ドキュメントファイルの名前を与えてコマンドを実行します。`-L` オプションの引数で、翻訳ドキュメントファイルの文字セットを指定することも可能です。

変換に成功すると、すべてのエントリに fuzzy フラグが設定された PO が生成されます。すべてのエントリに fuzzy フラグが設定されるのは、「ざっとでもいいので、ユーザには変換後にすべてのエントリの原文と訳文を確認してほしい」という開発者の意図を反映したものですユーザが自由に編集できる普通のドキュメントに少し制限を与えて Po4a の管理下に置く[*16]際には、何かしらの問題が発生している可能性があるのです。

■例　以下では、先程の `hoge.en.html` に対応する日本語訳 `hoge.ja.html` を PO に変換することを試みます。

まずは、とりあえず `po4a-gettextize` を実行してみました。

`hoge.ja.html` (入力):

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="ja" xml:lang="ja">
<head>
<title>テストファイル</title>
</head>
<body>
<h1>テスト用のファイル</h1>
<p>これは<a href="apple">リンゴ</a>です。</p>
<p>これは<a href="orange">オレンジ</a>です。</p>
<p>(訳注: オレンジはミカンとは違います。)</p>
</body>
</html>
```

コマンドライン:

[*16] 例えば、「ブロックの順序など全体的な構造を原文に合わせる必要がある」、「原文の同じ内容のブロックには、同じ訳文を当てる必要がある」など。

```
noritada[14:54]%  po4a-gettextize -v -f xhtml -m hoge.en.html -l hoge.ja.html -p ja.po
po4a gettextize: Original has less strings than the translation (6<7). Please
                 fix it by removing the extra entry from the translated file. You
                 may need an addendum (cf po4a(7)) to reput the chunk in place
                 after gettextization. A possible cause is that a text duplicated
                 in the original is not translated the same way each time. Remove
                 one of the translations, and you're fine.
po4a gettextization: Structure disparity between original and translated files:
msgid (at hoge.en.html:6 hoge.en.html:9) is of type 'Content of:
<html><body><h1>' while
msgstr (at hoge.ja.html:6) is of type 'Content of: <html><head><title>'.
Original text: Test file
Translated text: テストファイル
(result so far dumped to gettextization.failed.po)
The gettextization failed (once again). Don't give up, gettextizing is a subtle
art, but this is only needed once to convert a project to the gorgeous luxus
offered by po4a to translators.
Please refer to the po4a(7) documentation, the section "HOWTO convert a
pre-existing translation to po4a?" contains several hints to help you in your
task
```

訳注を独立したパラグラフにしたために PO への変換が失敗したと考え、訳注の行を削ってもう一度 `po4a-gettextize` を実行してみます。入力ファイルの内容は省略します。

コマンドライン:

```
noritada[14:55]%  po4a-gettextize -v -f xhtml -m hoge.en.html -l hoge.ja.html -p ja.po
po4a gettextization: Structure disparity between original and translated files:
msgid (at hoge.en.html:6 hoge.en.html:9) is of type 'Content of:
<html><body><h1>' while
msgstr (at hoge.ja.html:6) is of type 'Content of: <html><head><title>'.
Original text: Test file
Translated text: テストファイル
(result so far dumped to gettextization.failed.po)
The gettextization failed (once again). Don't give up, gettextizing is a subtle
art, but this is only needed once to convert a project to the gorgeous luxus
offered by po4a to translators.
Please refer to the po4a(7) documentation, the section "HOWTO convert a
pre-existing translation to po4a?" contains several hints to help you in your
task
```

なんだか変なエラーが出てしまっています。どうも、`title` と `h1` の原文がともに「Test file」なのに、一方の訳は「テストファイル」で、他方の訳が「テスト用のファイル」となっていることに原因があるようです。そこで、両方の訳を統一して `po4a-gettextize` にかけると、変換に成功します。

`hoge.ja.html` (入力):

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="ja" xml:lang="ja">
<head>
<title>テストファイル</title>
</head>
<body>
<h1>テストファイル</h1>
<p>これは<a href="apple">リンゴ</a>です。</p>
<p>これは<a href="orange">オレンジ</a>です。</p>
</body>
</html>
```

コマンドライン:

```
noritada[15:07]%  po4a-gettextize -v -f xhtml -m hoge.en.html -l hoge.ja.html -p ja.po
```

`ja.po` (出力):

```
# SOME DESCRIPTIVE TITLE
# Copyright (C) YEAR Free Software Foundation, Inc.
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2008-09-20 15:07+0900\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: ENCODING"

# type: Attribute 'xml:lang' of: <html>
#: hoge.en.html:4 hoge.en.html:4
#, fuzzy
msgid "en"
msgstr "ja"

# type: Content of: <html><body><h1>
#: hoge.en.html:6 hoge.en.html:9
#, fuzzy
msgid "Test file"
msgstr "テストファイル"

# type: Content of: <html><body><p>
#: hoge.en.html:10
#, fuzzy
msgid "This is an <a href=\"apple\">apple</a>."
msgstr "これは<a href=\"apple\">リンゴ</a>です。"

# type: Content of: <html><body><p>
#: hoge.en.html:11
#, fuzzy
msgid "This is an <a href=\"orange\">orange</a>."
msgstr "これは<a href=\"orange\">オレンジ</a>です。"
```

8.5.3 PO と原文ドキュメントファイルから翻訳ドキュメントファイルを生成する

さて、PO を作ることが目的なのではありません。当然ながら PO は翻訳を管理するための手段に過ぎません。重要なのは、翻訳済み PO と原文ドキュメントから翻訳ドキュメントを出力することです。

`po4a-translate`

8.5.4 原文ドキュメントファイルの更新を PO に反映させる

`po4a-updatepo`

8.5.5 PO と翻訳ドキュメントファイルを一括更新する

`po4a`

8.5.6 原文にない要素を翻訳ドキュメントファイルに挿入する

addenum

9 【でびあん】Debian パッケージメンテナというお仕事【現在募集中】

やまね ひでき

9.1 Debian パッケージメンテナのお仕事

実は、Debian パッケージのメンテナになるにはそんなに敷居は高くありません。よっぽど入学試験／就職／転職活動の方が難しいと思うほどです。恐れることはありません。多少なりとも興味がある方はこれからの説明をご覧ください。

9.2 メンテナになる前の下準備

1. GPG 鍵を作っておく (aptitude install gpg && gpg –gen-key)
2. 公的な身分証明書を用意しておく
3. 先ほど作った GPG 鍵と身分証明書を利用して、既存の Debian Developer と GPG キーサインの交換を行っておく
4. メンテナや Developer として問題ないと見なされる作業をしておく (contribute!)

この 4 つだけ準備したら、さぁ、メンテナになる作業の始まりです。

9.3 The way to Debian package maintainer

メンテナになる方法としては 2 点あります。

1. 自分から「このソフトを Debian パッケージにしたい」と申請する
2. 既存のパッケージのメンテナからメンテナンスを引き継ぐ

1. は ITP (Intent To Package) という形で行います。特に RFP (Request For Package) という形でユーザから「このソフトを Debian パッケージにして欲しいなぁ」という声に答えるのが望ましいでしょう。

2. は RFH (Request For Help、協力者募集中)、RFA (Request for Adoption、新メンテナ募集中) や O (Orphaned、みなしご化) というステータスになったパッケージに対して「自分がメンテナになります」と宣言します。

ITP, RFP, RFA, Orphaned は全て Debian Bug Tracking System (BTS) で管理登録されます (BTS の使い方についてはそのうち別途)。これらを総称して「作業が望まれるパッケージ (Work-Needing and Prospective Packages; WNPP)」と呼びます。この WNPP、BTS で追うのは結構辛いので wnpp.debian.net [*17] を使いましょう。各略語

[*17] `http://wnpp.debian.net` ではパッケージ化希望されているものや、メンテナが引き取り手を募集しているパッケージを検索できます。
- ページの見た目は良くない ;-)

の意味などについては http://www.jp.debian.org/devel/wnpp/ をご覧ください。

9.3.1 ITP、RFP パッケージ化

大抵 1 からパッケージ化作業ですので、ウェブサイトからソースを取得してライセンスを確認の上、パッケージ作りのお作法に従ってパッケージを作成します。パッケージを作成する際は、unstable 環境の lintian で warning などが出なくなるよう (Debian Policy Manual に準拠するよう) に頑張りましょう。パッケージが作成できたら pbuilder でクリーンルーム環境でビルド可能かどうか (Build-Depends に間違いが無いか) のチェックを行います。問題なくビルドできたら piuparts を使ってインストール／アンインストールで問題が無いか (Depends や preinst, postinst, prerm, postrm スクリプトに問題が無いか) を確認しましょう。ここまで来て問題なければ「Eat your own dog food」、つまり自分の環境にパッケージを入れて問題が起きないかどうかチェックするとベストでしょう。

■ライセンスについて ちなみに一番大変なのはここでの「ライセンス確認」作業だと個人的には思っています。そのソフトのウェブページににちょっと書いてあることを鵜呑みにしたりすると、実は中身は全然違うライセンスだ、とか、実はリンクするライブラリとは矛盾するライセンスだ、とかがありますし、英訳ライセンスがないと Debian のパッケージ管理者 (ftpmaster) がライセンス的にそのパッケージを受け入れるかどうかの判断がそもそも出来ませんので必ず適当な英訳ライセンスへの翻訳作業が必要です。
ライセンスは、独自ライセンスよりも一般的に FLOSS の代表的ライセンス (GPL, BSD, MIT, Artistic など) を使う方が全ての利害関係者にとって利益があるでしょう。何故ならば、ライセンスは「プロトコル」のようなものであり、既存プロトコルを利用する方が新たにプロトコルを実装するよりもはるかに容易に取り扱えるからです。代表的ライセンスでは目的が達成できないときにのみ、独自ライセンスを利用するのが賢いやり方です。

9.3.2 RFA、Orphan への対処

すでに現在のメンテナが引き継ぎ手を募集している状態 (RFA) や、完全にギブアップ or 興味を失ってしまった (Orphan) パッケージについては、BTS を利用してパッケージを引き取る旨の宣言 (ITA, Intend To Adapt) を行いましょう。加えて関連のメーリングリストで「自分がやろうと思うけど、どう？」と聞いてみるとさらに良いでしょう。合意がとれたら、debian/changelog にて該当の RFA/Orphan バグを閉じるエントリを追記しておきます。大抵の場合は色々とバグレポートが溜まっているので片付けて同様に changelog に記載しておきましょう。

9.4 アップロード

ここまでが終わったらアップロードを行います。Debian Developer であればパッケージを dput や dupload と呼ばれるコマンドを使って、作成されたソースパッケージとバイナリパッケージを直接 Debian へアップロードします。そうでない場合は適当な場所にアップロードして、Debian Developer に代理アップロードをお願いしましょう (アップロード先は、http:://mentors.debian.net、代理アップロード依頼先は debian-devel@debian.or.jp がおすすめです)。

```
dupload -t anonymous-ftp-master ccspatch_1.6.4-20080903-1_i386.changes
```

この時点でツールは .dsc ファイルや .changes ファイルに gpg 署名されているかどうかをチェックして、されていない場合はアップロードしないなどしてくれます。

サーバ側では、まず「既にあるパッケージの更新なのか、それとも新規パッケージなのか」の判断を行います。新規パッケージの場合は NEW Queue [*18] と呼ばれる所に自動的に移されて、ftpmaster が逐次入れてよいものかどうかのチェックを行っています。おおよそこの工程は 1、2 週間程度かかります。

- RSS で状況が取得できるので、RSS reader で適宜ピックアップにてチェックすることをお勧めします。

[*18] http://ftp-master.debian.org/new.html

チェックは特にライセンスの問題が無いかについてなどが争点となります。この作業は高度な判断が要求されるため、人的リソース的にボトルネックになりやすいところと、拒否された場合に「何故このパッケージが拒否されたのか」の記録が容易にトラッキングできないのが問題点です。人的リソースについては、アクティブなメンバーを入れることで以前と比較しても改善がされており、トラッキングについては最近になってシステマチックにチェック可能なように提案が出てきているところです。

更新パッケージについては、サーバ側でソースパッケージの gpg 署名をチェックなどをして「本当にこれは入れていいものかどうか」を判断してダメな場合は reject されます。このチェックについては、アップロード後にサーバから受け入れ／拒否のメールが届きます。

まずアップロードされたパッケージの処理を始めたよ、というメール

```
From: Archive Administrator <dak@ftp-master.debian.org>
Subject: Processing of ccspatch_1.6.4-20080903-1_i386.changes

ccspatch_1.6.4-20080903-1_i386.changes uploaded successfully to localhost
along with the files:
  ccspatch_1.6.4-20080903-1.dsc
  ccspatch_1.6.4-20080903.orig.tar.gz
  ccspatch_1.6.4-20080903-1.diff.gz
  linux-patch-tomoyo_1.6.4-20080903-1_all.deb

Greetings,

        Your Debian queue daemon
```

チェックして incoming repository に入れたよ、というメール

```
From: Debian Installer <installer@ftp-master.debian.org>
Subject: ccspatch_1.6.4-20080903-1_i386.changes ACCEPTED

Accepted:
ccspatch_1.6.4-20080903-1.diff.gz
  to pool/main/c/ccspatch/ccspatch_1.6.4-20080903-1.diff.gz
ccspatch_1.6.4-20080903-1.dsc
  to pool/main/c/ccspatch/ccspatch_1.6.4-20080903-1.dsc
ccspatch_1.6.4-20080903.orig.tar.gz
  to pool/main/c/ccspatch/ccspatch_1.6.4-20080903.orig.tar.gz
linux-patch-tomoyo_1.6.4-20080903-1_all.deb
  to pool/main/c/ccspatch/linux-patch-tomoyo_1.6.4-20080903-1_all.deb


Override entries for your package:
ccspatch_1.6.4-20080903-1.dsc - source devel
linux-patch-tomoyo_1.6.4-20080903-1_all.deb - extra admin

Announcing to debian-devel-changes@lists.debian.org


Thank you for your contribution to Debian.
```

これでパッケージのアップロードが完了しました。BTS に記載されているバグを修正した旨を debian/changelog に適切なスタイルで記述しておくと、BTS から bug closed のメールがやってきます。

9.4.1 その他アップロードに関する注意

ちなみに、Debian バージョンのみの更新の場合 (foo 1.0-1 から foo 1.0-2 への更新など) はオリジナルのソースファイル (orig.tar.gz) をアップロードしないことになっています。もし、orig.tar.gz をアップロードした場合は後から拒否のメールが届きます。

あと注意として、ライブラリファイルや他の多くのパッケージと関連しているようなパッケージで更新をかけると unstable 環境が壊れるような場合は、experimental へアップロードして関係者と調整することが推奨されています。

9.5 メンテナンス作業

BTS を通じてユーザからバグ報告メールが来るので、それに対応してパッケージの debian ディレクトリ以下を更新して、修正出来たら changelog に完了した旨を記載しておきます (dch コマンドが便利です)。特に RC (Release Critical) バグは早めに修正しましょう。バグが他のパッケージと関連する場合は適当なメーリングリストで関係者と対応を協議して進めていきます。場合によっては、自分のパッケージのバグではなく、他のパッケージのバグである

場合もありますので、その場合はバグを BTS で reassign しておきます。パッケージとしてのバグではなく、元々のソース由来のバグの場合は、バグを upstream (開発元) に転送するかパッチを作って upstream と協議しましょう。

そして upstream での更新に合わせてパッケージも更新します。upstream の更新は debian/watch ファイルを適切に記載していれば、DEHS [*19] によって定期的にチェックされ、Debian Quality Assuarance の developer ページ [*20] で状況を確認できます (あとは upstream のリリースアナウンスが流れるメーリングリストに入っておくと良いでしょう)。きちんと設定された `debian/watch` ファイルがあれば、パッケージによっては更新がものの 1 分も経たずに出来ます。

チーム制でメンテナンスをするような場合は共有 repository に更新したソースを忘れずに commit しておきましょう。commit 出来ない場合は、チームのメーリングリストにパッチを送ります。幾度か適切なパッチを送りつづけていると、いつの間にか commit 権が与えられていることが多い様です。

なお、バグ修正や Debian Policy に適合するために Debian パッケージ側で色々とパッチを当てている (dpatch や quilt を利用する) と新バージョン側のソースとコンフリクトを起こしてその修正に時間がかかることがありますので、なるべく upstream の開発者とメーリングリストなどを通じて普段からコンタクトを取り合っておいて、パッチ自体が取り込まれて不要になるように働きかけましょう。

この様にしてパッケージ化、メンテナンス等をメンテナは日々行っていきます。特に問題なく時間が経過したパッケージは unstable から testing に移行され、次のリリースを待つことになります。そして、リリースの日を迎えたパッケージは stable へと移されます。

9.6 特殊な場合

以下に頻繁には起きないはずの、ちょっと変わった場合の対応を挙げておきます。

9.6.1 NMU

Non-Maintainer Upload の略で、既存のメンテナ以外の人間がバグ修正のためにパッケージをアップロードすることを指します。あまり頻繁に NMU があるようなパッケージについては、メンテナが本当に活動しているのかどうかをチェックした方がいいでしょう。

9.6.2 FREEEEEEZE!!

特にリリース前には「システム全体の安定化」が必要なため、freeze、つまりパッケージの unstable から testing への自動移行が停止されます。しかし、freeze の間に発覚した、そのままリリースされてしまっては困るようなバグの修正が必要な場合もあります。その場合は リリースマネージャ (RM) に対して「私のこのパッケージを testing に入れるようにしてください！」とお願いします。これを「unblock (exception) request」と一般に呼びます。やり方としてはメーリングリストで RM が理解しやすいフォーマットでメールを投げるだけです。

```
To: debian-release@lists.debian.org
subject: Please unblock <package> <version>
```

- changelog の抜粋
- 理由
- debian-devel-announce で流れている freeze exception のどれに当てはまるのか
- このアップデートによって何が改善されるのか
- このアップデートでは regression が起きない説明
- その他、RM が納得してくれるような内容

[*19] Debian External Health Status`http://dehs.alioth.debian.org`
[*20] `http://qa.debian.org/developer.php`

うまくすれば RM から ”unblocked” という素っ気ないメールが届きます。これが来たら大成功です。今 unstable にあるパッケージは testing へ移行することが可能になります。

9.7 メンテナ権の移譲

パッケージメンテナも人の子、残念ながら諸事情によりメンテナンスが出来なくなるような事態が起きるかもしれません。なるべく一人でメンテナンスせずチームあるいは自分以外の詳しい／信用できるメンテナもアップロードできるようにしておきます。具体的には debian/control の Uploaders フィールドにアップロードしてもいい人の名前とメールアドレスを書いておきます (GPG 署名に使うものと同一である必要があります)。

```
Source: ttf-vlgothic
Section: x11
Priority: optional
Maintainer: Debian Fonts Task Force <pkg-fonts-devel@lists.alioth.debian.org>
Uploaders: Hideki Yamane (Debian-JP) <henrich@debian.or.jp>
DM-Upload-Allowed: yes
Build-Depends: debhelper (>= 5)
Build-Depends-Indep: defoma, bzip2
Standards-Version: 3.8.0
Homepage: http://dicey.org/vlgothic/
Vcs-Svn: svn://svn.debian.org/pkg-fonts/packages/ttf-vlgothic/
Vcs-Browser: http://svn.debian.org/wsvn/pkg-fonts/packages/ttf-vlgothic/
```

そして、メンテナンスが出来なくなりそうな場合は、あまり粘らずにさっさと宣言 (RFA、Orphan) しておきましょう。宣言が無いと「このパッケージ古いな」という悪評が立つだけ…という結果になりかねません。なお、日本の首相と違ってメンテナは頻繁に交代してもきちんとした理由があれば文句は言われることはありません :-)　（できれば最初の方から複数人でのチームメンテナンスなどしておくと困ったことになる確率は減ります）

9.8 終わりに

如何でしたか？パッケージメンテナというお仕事がどのようなものか、多少理解が深まっていただければ幸いです。

10 Debconf8 参加報告

上川 純一

10.1 Debconf とは

2008 年度の Debconf は 8 月 10 日から 8 月 16 日まで、アルゼンチンのマルデルプラタで行われました。日本からは、上川 純一が参加しました。

10.1.1 Debconf の歴史・経緯

Debian Conference [*21]は Debian の開発者たちが一堂に会するイベントです。通常顔をあわせることのないメンバーたちが一堂に会し友好を深め、技術的な議論を戦わせます。過去の開催履歴を見てみると表 4 のようになります。

表 4 歴代の Debconf 参加者推移

年	名前	場所	参加人数
2000	debconf 0	フランス ボルドー	
2001	debconf 1	フランス ボルドー	
2002	debconf 2	カナダ トロント	90 名
2003	debconf 3	ノルウェー オスロ	140 名
2004	debconf 4	ブラジル ポルトアレグレ	150 名
2005	debconf 5	フィンランド ヘルシンキ	200 名
2006	debconf 6	メキシコ オアスタペック	300 名
2007	debconf 7	英国スコットランド エジンバラ	約 400 名
2008	debconf 8	アルゼンチン マルデルプラタ	約 200 名

10.1.2 Debconf 2008

2008 年度の Debconf は海辺のホテルを二棟借りて開催しました。宿泊しているホテルの 7F と 1F がハックラボになっており、7F がカンファレンスルームになっているという便利な環境でした。

10.2 アルゼンチン・マルデルプラタ

10.2.1 行き方

日本からアルゼンチンまでは、直行便がありません。上川はサンフランシスコ・カナダのトロント・チリのサンチャゴを経由してアルゼンチンのブエノスアイレスまで移動 (ここまでで約 30 時間程度)、そこからバスで移動 (5 時間 (直行) から 10 時間 (乗り換え)) しました。

今回の会期は従来と違い長期休暇でバカンスをとっている人の多い時期です。そのため、飛行機の予約のタイミン

[*21] http://debconf8.debconf.org/

グが難しかったかもしれません。また、日本からアルゼンチンにいく便には安いものがあまりみつからず苦労しました。

ブエノスアイレスからマルデルプラタの間は東京・大阪間程度の距離があり、多数のバスが通っています。空港からブエノスアイレスの市街で乗り換えれば多数のバスがあるのですが、空港からの直行のバスは非常に少なく (Manuel Tenda Leon という会社のバスのみが運行している模様)、予約ができない人がでていたようです。

10.2.2 会場

会場はビーチリゾートとして知られるマルデルプラタの中央カジノ前のホテルでした。

10.3 スケジュール

8 月 10 日から開始し、8 月 16 日までイベントがぎっしりつまっていました。8 月 18 日にブエノスアイレスに移動し、そこで Debian Day を開催しました。

10.4 会期中で気になったこと

上川が今回参加して気になったことを紹介します。

Debconf に参加しているミラーサーバはフルミラーではなくアーキテクチャi386、 amd64、 ppc と armel でした。

qemubuilder を開発している関係で armel 関連の作業を行いました。qemubuilder の armel 移植と、armel 関連のデバッグなどです。しかしながら途中で PC の HDD が故障し後半は HDD の購入からインストール、macbook のブートローダのデバッグに費しました。

PGP キーを IC カードで利用する仕組みがあるみたいで、作成していました。ただ、仕組みがよくわからないので今回はスルーしました。

11 「その場で勉強会資料を作成しちゃえ」 Debian を使った LaTeX 原稿作成合宿

上川　純一

11.1 概要

Debian 勉強会では資料を LaTeX で管理しています。Emacs、 Git、 LaTeX で Debian 勉強会資料作成をする流れを体験してみましょう。

ハンズオンで Debian 勉強会資料のチェックアウト、作成からコミットまでの流れを一通り試します。途中でつまったらできるまで現地で対応する予定です。[*22]。

11.2 事前準備

当日の時間は限られているため、事前準備が必要です。現地に入るまでに準備しておくものです

- 無線 LAN 接続可能なノートパソコン (十分長い LAN ケーブルを持参しても可能、スイッチも持参してください)
- 必要なパッケージをインストールした Debian GNU/Linux sid のシステム
- 記事にする内容。紹介したいツール/パッケージ関連のコマンドライン出力と画面写真と簡単な説明文章。

事前にインストールしておいて欲しい必要なパッケージは

- 基本ツール: make
- LaTeX 一式: ptex-bin dvipdfmx latex-beamer okumura-clsfiles ghostscript-x[*23] xpdf xpdf-japanese evince poppler-data texlive-latex-extra
- emacs / whizzytex 関連一式: whizzytex advi emacs22-gtk[*24]yatex gs-cjk-resource gv
- Git 関連一式: git-core git-gui qgit
- DHCP / Avahi 関連一式: avahi-daemon avahi-autoipd libnss-mdns
- 日本語フォント関連: ttf-mona ttf-sazanami-mincho ttf-vlgothic
- 日本語入力関連: scim-anthy

[*22] ベストエフォート、もしかするとできない場合もあります
[*23] lenny 以前での gs-esp 相当
[*24] emacs22 でもよい、emacs22-nox は難しいかも

avahi の設定が正しいことは、ping ホスト名.local が使えるかどうかで確認します。/etc/nsswitch.conf の hosts 行に mdns[*25]を参照する設定が追加されていることを確認してください。

```
hosts:          files mdns dns
```

```
# apt-get update
# apt-get install \
 make ptex-bin dvipdfmx latex-beamer \
 okumura-clsfiles ghostscript-x xpdf xpdf-japanese whizzytex \
 evince poppler-data \
 texlive-latex-extra \
 advi emacs22-gtk yatex gs-cjk-resource gv git-core git-gui \
 qgit avahi-daemon \
 libnss-mdns avahi-autoipd \
 ttf-mona ttf-sazanami-mincho ttf-vlgothic \
 scim-anthy
# jisftconfig add
```

*26

事前にダウンロードしてビルドできる環境であることを確認しておいてください。

```
$ git clone git://git.debian.org/git/tokyodebian/monthly-report.git
$ cd monthly-report
$ cp -p git-pre-commit.sh .git/hooks/pre-commit
$ make -j4
$ ls *.pdf # 110 くらいの PDF ファイルが生成されていることを確認
```

Emacs の設定をします。yatex の設定と文字コードの設定をします。

.emacs に

```
;; YaTeX が漢字コードを毎回 ISO-2022-JP に設定しないようにする
(setq YaTeX-kanji-code nil)

;; git.el をロードする
(load-library "/usr/share/doc/git-core/contrib/emacs/git.el")
```

Git の設定もしておきましょう。ユーザ名とメールアドレスを設定します。設定しないとホスト名や /etc/passwd の設定をデフォルトで使ってしまいます。

```
$ git config --global user.email dancer@netfort.gr.jp
$ git config --global user.name "Junichi Uekawa"
```

11.3 演習 0: 環境設定

勉強会の会場で準備している環境です

- avahi 経由で接続できる Debian package cache (i386、amd64)
- 無線 LAN(WEP) ESSID:会場にて発表

DHCP でネットワーク接続してください。avahi(mDNS) で名前解決できることを確認してください。apt リポジトリの設定が可能であることを確認してください。

```
# ping coreduo.local
# cat /etc/apt/sources.list
deb http://coreduo.local:9999/debian/ sid main contrib non-free
# apt-get update
```

11.4 演習 1: リポジトリチェックアウト

```
$ git clone git://coreduo.local/git/monthly-report.git
```

*25 なんらかの mdns の設定であればよく、例えば mdns4 であってもよい。

*26 okumura-clsfiles の依存する ptex-jisfonts はインストール後に手動で設定をする必要がある。具体的には jisftconfig add を実行。

```
$ make -j4 # エラーがでます
$ cp -p git-pre-commit.sh .git/hooks/pre-commit
$ make -j4
```

11.5 演習 2: emacs whizzytex 起動

```
$ cd XXXXX
$ emacs
```

whizzytex を起動します。プリビューが開始するというメリットだけでなく、即時コンパイルエラーを検出できるのが重要です。

```
M-x whizzytex-mode
```

スペルチェックも起動しましょう。[*27]

```
M-x flyspell-mode
```

11.6 演習 3: セクションを追加

まずセクションを追加します。

```
\dancersection{XXXX}{名前}
\index{XXXX@YYYY}
\label{XXXX@YYYY}

% 本文
```

11.7 演習 4: 説明の章だてを書いてみる

```
\subsection{xxxx}

\subsection{xxxx}

\subsection{xxxx}

\subsection{xxxx}

\subsection{xxxx}
```

11.8 演習 5: 説明の中身を書いてみる

文章は空行区切りでパラグラフになります。roff などと同じです。改行は無視されます。
\ではじまる文字列が命令です

```
\XXXX{YYYY}
```

monthlyreport.sty で追加のコマンドが定義されています。[*28]
表を作ってみましょう。

表 5　キャプション

ディストリビューション	バージョン	備考
ディストリビューション	バージョン	備考
ディストリビューション	バージョン	備考

[*27] iamerican もしくは ibritish パッケージが必要です。
[*28] フットノートが追加できます

知っておきたい構文は実はあまりたくさんはありません。利用されている上位:

```
$ sed -n 's/.*\(\\[a-z]\+\).*/\1/pg' *.tex  | sort | uniq -c | sort -rn
   6948 \item
   6240 \end
   5628 \begin
   1952 \subsection
   1623 \subsubsection
   1491 \santaku
   1258 \url
   1189 \hsize
    895 \hline
    874 \texttt
    522 \dancersection
    473 \footnote
    451 \includegraphics
    447 \vspace
    405 \label
    378 \frametitle
    357 \usepackage
    323 \hspace
    316 \hrule
    300 \section
    253 \bf
    236 \index
```

利用されている上位の begin-end 構文

```
$ sed -n 's/.*\(\\begin{[a-z]\+\).*/\1/pg' *.tex  | sort | uniq -c | sort -rn
   1742 \begin{itemize
   1293 \begin{commandline
   1245 \begin{frame
    781 \begin{minipage
    271 \begin{center
    194 \begin{enumerate
    148 \begin{tabular
    133 \begin{figure
```

Debian GNU/Linux のようによく使う文字列は newcommand で定義できます。ただし、命令がファイル毎に別の意味を持つようになると管理がしにくくなるのでできるだけ monthlyreport.sty に定義は集中させています。monthlyreport.sty は共有しているので変更する際には御注意を。最初は変更内容について誰かにレビューをお願いしたほうがよいと思います。

```
$ git log --pretty=format:%an monthlyreport.sty | sort | uniq -c
     15 Junichi Uekawa
      3 Nobuhiro Iwamatsu
```

エスケープが必要な文字列の出力方法

- \ verb は見にくくなるので最終的手段
- ~
- ^
- aaaa<aaaa そのままだと:¡
- aaaa>aaaa そのままだと:¿
- aaaa#aaaa
- aaaa%aaaa
- _aaaa

よく使うコマンド

- begin itemize / item / end itemize
- begin enumerate / item / end enumerate
- commandline (勉強会用に定義)
- includegraphics 後述

11.9 演習 6: コマンドライン出力を追加してみる

```
\ begin{commandline}
コマンドライン出力
\ end{commandline}
```

11.10 演習 7: png 画像を挿入してみる

includegraphics

png

git add

ebb

make で確認。

11.11 演習 8: git で変更をコミットしてみる

```
$ make
```

```
M-x git-status
```

11.12 演習 9: git での変更を送信してみる

```
$ git pull
```

コンフリクトを解消します。

```
M-x git-status
```

```
$ git push
```

```
$ mkdir ~/patches/
$ git format-patch -o ~/patches/ HEAD^
```

生成されたパッチファイルをメールで送付します。

11.13 演習 10: git で変更をマージしてみる

11.14 演習 11: 全体を眺める

outline-minor-mode が便利です。

11.15 今回演習できない TIPS

11.15.1 alioth tips

資料の編集には alioth を利用します。`alioth.debian.org`、 `git.debian.org` です。

Alioth にアカウントを作成します。Debian Developer でない場合のユーザ名は、XXXX-guest というユーザ名になります。

便利につかうため、.ssh/config に設定を追加しましょう。

```
ServerAliveInterval 10
ServerAliveCountMax 12

Host alioth.debian.org
 ControlMaster auto
 ControlPath ~/tmp/ssh-%r@%h:%p
 User xxxx-guest

Host git.debian.org
 ControlMaster auto
 ControlPath ~/tmp/ssh-%r@%h:%p
 User xxxx-guest

Host localhost
 StrictHostKeyChecking no

Host *.local
  CheckHostIP no
```

12 Debian を Windows な PC でも楽しもう –応用編

名村 知弘

12.1 はじめに

coLinux の配布サイト[*29]では、Debian etch のイメージファイルが配布されていますので、coLinux をインストールすると、すぐに Debian を楽しむことができます。

しかし、配布されているイメージファイルは、ディスクイメージのサイズが 1GB だったり、どのようなパッケージがどういった設定でセットアップされているのかなど、中身が非常に気になったりします。

ダウンロードすればお手軽に使えるというのは良いのですが、将来の拡張のためパーティション構成を変更したり、イメージの内容をいちいち確認するのも面倒ですし、それならばいっその事、新規インストールしてみよう。と言うことで、半ば強引な導入ですが、coLinux に Debian を新規インストールする方法と、coLinux にインストールされた X アプリケーションを起動する方法をご紹介します。

説明するにあたり、以下の前提で記載していますので、適宜それぞれの環境に応じて読み替えていただくようお願いいたします。

1. coLinux を `c:\coLinux` にインストールしているものとしています。
2. coLinux はネットワーク共有にて接続されているものとしてます。
3. Debian は etch を使用するものとしています。
4. Debian は IP アドレス 192.168.0.10 が割り当てられているものとしています。

12.2 coLinux に Debian を新規インストール

coLinux への Debian インストール手順は以下のようになります。

1. Debian をインストールするためのディスクイメージの作成
2. Debian の ISO イメージを入手
3. initrd イメージの取得
4. coLinux を Debian インストール用に設定
5. Debian のインストール
6. coLinux を通常起動用に設定

[*29] `http://sourceforge.net/projects/colinux/files`

12.2.1 Debian をインストールするためのディスクイメージの作成

coLinux では、1 つのパーティションに対して 1 つのディスクイメージファイルを用意することで、HDD をエミュレートしています。ディスクイメージファイルはあらかじめ必要なサイズを割り当てる必要があり、coLinux が割り当てられたサイズを超えて書き込むことはできません。(つまり、HDD が一杯になったら書き込めないのと同様、ディスクイメージファイルに割り当てられたサイズを使用すると、それ以上書き込みが行えなくなります)

今回は以下の構成でインストールを行いたいと思います。

領域	サイズ	ディスクイメージ
/	2GB	root_fs
/home	2GB	home_fs
swap	500MB	swap_fs

表 6　パーティション割り当て

ここでは 3 パーティション使用するため、3 ファイル作成する必要があります。また、あらかじめ使用するサイズを割り当てる必要があるため、Windows の fsutil コマンドを使用します。fsutil は指定されたファイル名、サイズで空ファイルを作成することができます。

ファイル名とサイズは表 6 に基づいて決定していますので、お手持ちの PC のディスク空き容量などを考慮しながら、適切なサイズを決定していただければと思います。

コマンドプロンプトを起動し、c:\coLinux に cd してから、以下のコマンドを実行するとファイルが作成されます。

```
fsutil file createnew root_fs 2147483648
fsutil file createnew home_fs 2147483648
fsutil file createnew swap_fs 536870912
```

12.2.2 Debian の ISO イメージを入手

まずは Debian をインストールするための、ISO イメージを入手[*30]します。ここでは debian-40r4a-i386-businesscard.iso を使用します。

12.2.3 initrd イメージの取得

c:\coLinux には、coLinux 起動用の initrd イメージが用意されていますが、Debian インストール用に coLinux を起動するためには、インストール用に起動するための initrd イメージが必要となります。

インストール起動用と通常起動用の initrd イメージを混同してしまわないように、c:\coLinux にあるオリジナルの initrd.gz を、initrd-normal.gz とリネームしておきます。

インストール用の initrd イメージは、Debian の ISO イメージに収録されているため、ISO イメージを展開できるツール、Explzh[*31]や、TUGZip[*32]を使用して取り出します。

展開ツールをインストールして起動し、Debian の ISO イメージを開きます。ISO イメージの中の、「/install.386/initrd.gz」を取り出し、こちらも通常起動用の initrd イメージと混同してしまわないように、initrd-install.gz とリネームして、c:\coLinux に保存します。

[*30] http://www.debian.org/CD/netinst/
[*31] http://www.ponsoftware.com/archiver/
[*32] http://www.tugzip.com/

12.2.4 coLinux を Debian インストール用に設定

coLinux をインストール用に起動するためのファイルは全て揃いましたので、これらファイルを使用して Debian インストールが行われるよう、coLinux の設定を変更します。

c:\coLinux に debian-install.conf と言う名前でファイルを作成し、以下の内容を記載して保存します。

```
kernel=vmlinux
initrd=initrd-install.gz
mem=512
cobd0="\DosDevices\C:\coLinux\swap_fs"
cobd1="\DosDevices\C:\coLinux\root_fs"
cobd2="\DosDevices\C:\coLinux\home_fs"
cobd3="\DosDevices\C:\coLinux\debian-40r4a-i386-businesscard.iso"
cofs0="\DosDevices\C:\coLinux"
eth0=tuntap
root=/dev/cobd1
ro
```

上記設定ファイルを簡単に説明すると、2 行目でインストール起動用の initrd イメージを指定し、3 行目で coLinux の使用するメモリサイズを指定します。4〜6 行目で使用するディスクイメージ (今回は 3 つ) を指定します。7 行目で Debian の ISO イメージを指定します。8 行目で coLinux インストールディレクトリを指定します。10 行目で「/」ファイルシステムのデバイス名を指定します。

12.2.5 Debian のインストール

準備が整いましたので、debian-install.conf を使用して coLinux を起動します。コマンドプロンプトを起動し、c:\coLinux に cd してから、以下のコマンドを実行します。

```
colinux-daemon.exe -t nt "@c:\coLinux\debian-install.conf"
```

そうすると、いつもの見慣れた?インストール画面が起動されますので、以下の手順に従ってインストールを進めていきます。

まずはインストーラの環境設定を行います。以下の手順に従ってインストール作業を進めていきます。

```
# インストーラで使用する言語を選択します。
* English（Japanese ではインストール中に表示されるメッセージが文字化けします）

# 国を選択します。
* Japan （ <other> → <Japan> ）

# キーマップを選択する。
* Japanese（ご使用のキーボードに合わせてください）

# CD-ROM のドライバは coLinux が設定してくれるため必要ありません。
* <No>

# CD-ROM のドライバは coLinux が設定してくれるため、手動で使わないよう設定します。
* <Yes>→「none」
```

ここまで実行すると、図 4 のような、CD-ROM をマウントする画面が表示されます。

しかし、インストーラは coLinux が設定してくれた CD-ROM ドライブにアクセスするためのデバイスファイルを作成していないため、このままでは CD-ROM にアクセスすることができません。

そこで、coLinux が設定してくれた各種デバイスにアクセスするための、デバイスファイルを作成します。

[ALT] + [F2] キーを押し、インストーラ画面からターミナルに移動します。移動すると、図 5 のような画面が表示されます。

ターミナルにて以下のコマンドを入力します。

```
# デバイスファイル/dev/cobdx を作成します。
i=0; while [ $i -lt 32 ]; do mknod /dev/cobd$i b 117 $i; i=‘expr $i + 1‘; done

# 環境によっては「mknod: /dev/cobdx: File exists」と言うメッセージが表示されることがありますが、
# これは既に/dev/cobdx が作成されているということですので、
# そのまま次に進めてもらっても問題ありません。

# デバイスファイル/dev/cofsx を作成します。
i=0; while [ $i -lt 16 ]; do mknod /dev/cofs$i b 117 $i; i=‘expr $i + 1‘; done
```

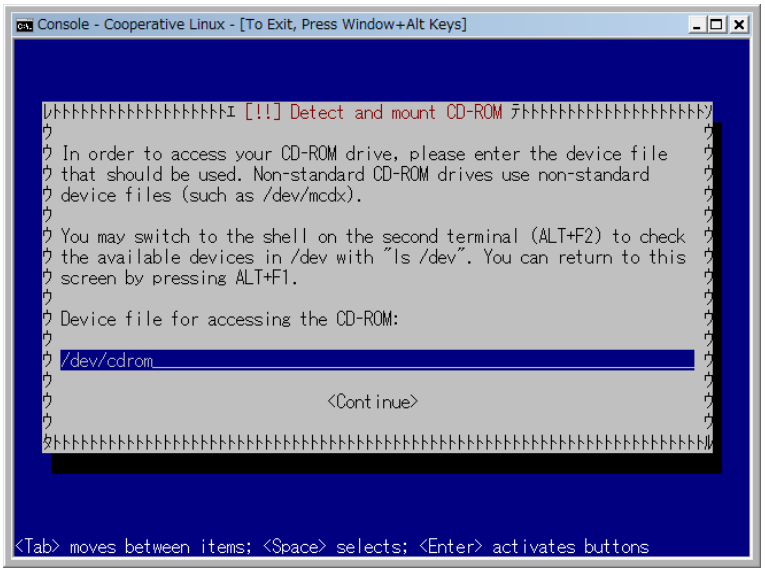

図 4　CD-ROM のマウント

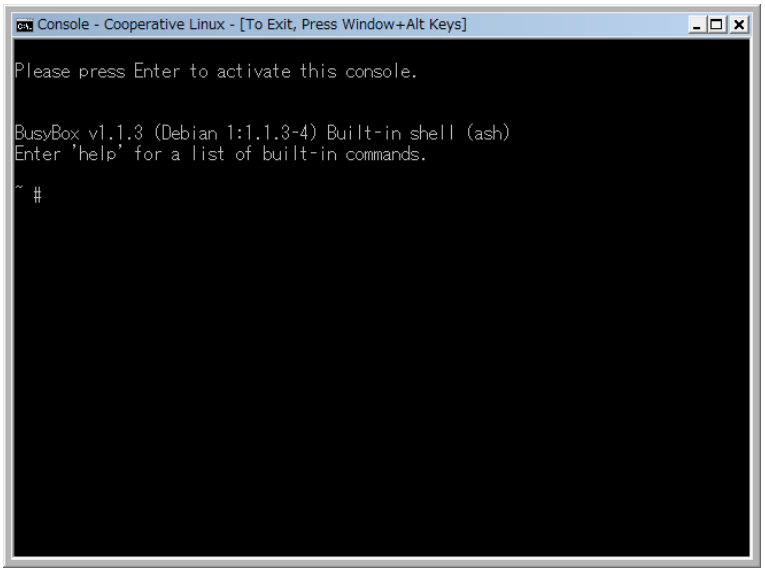

図 5　ターミナルに移動

これで coLinux が設定してくれた CD-ROM をマウントすることができるようになりましたので、[ALT] + [F1] キーを押し、ターミナルからインストーラ画面に戻ります。戻ると、先程中断した図 4 の画面が表示されますので、以下の手順に沿ってインストール作業を進めていきます。

```
# CD-ROM をマウントするため、Debian の ISO イメージが設定されているデバイスファイルを指定します。
# 今回は debian-install.conf の 7 行目にて以下のように設定していますので、「/dev/cobd3」を指定します。
# cobd3="\DosDevices\C:\coLinux\debian-40r1-i386-businesscard.iso"
* 「/dev/cobd3」と入力し<Continue>

# カーネルモジュールは coLinux のものを使用しますので、読み込まずにインストールを続けます。
* <Yes>

# ネットワークを設定が行われますが、Windows 側で設定が済んでいれば DHCP で取得できますので、
# 特に何かする必要はありません。
```

ネットワークの設定が終わると、以降のインストールは Debian のインストールでは行えませんので、続きの作業は手動にて行っていきます。

[ALT] + [F2] キーを押し、再びターミナルに移動し、以下のコマンドを順次実行していきます。

```
# まずは swap ファイルを作成し、有効化します。
# 今回は debian-install.conf の 4 行目にて以下のように設定していますので、
# 「/dev/cobd0」に対して mkswap を実行します。
# cobd0="\DosDevices\C:\coLinux\swap_fs"
mkswap /dev/cobd0
sync; sync; sync
swapon /dev/cobd0

# 次に、coLinux をインストールするために用意したディスクイメージに
# ext3 ファイルシステムを作成します。
# 今回は debian-install.conf の 5, 6 行目にて以下のように設定していますので、
# 「/dev/cobd1」、「/dev/cobd2」に対して mke2fs を実行します。
# cobd1="\DosDevices\C:\coLinux\root_fs"
# cobd2="\DosDevices\C:\coLinux\home_fs"
mke2fs -j /dev/cobd1
mke2fs -j /dev/cobd2

# インストール対象の「/」をマウントするためのディレクトリを作成し、マウントします。
mkdir /target
mount /dev/cobd1 /target
cd /target

# coLinux 附属のカーネルモジュールを使用するため、c:\coLinux をマウントし、
# c:\coLinux\vmlinux-modules.tar.gz を\texttt{/target}に展開します。
mkdir -p /mnt/windows
mount -t cofs /dev/cofs0 /mnt/windows
tar -zxvf /mnt/windows/vmlinux-modules.tar.gz

# /target に coLinux の設定するデバイスファイルを作成します。
mkdir /target/dev
i=0; while [ $i -lt 32 ]; do mknod /target/dev/cobd$i b 117 $i; i=$((i+1)); done
i=0; while [ $i -lt 16 ]; do mknod /target/dev/cofs$i b 117 $i; i=$((i+1)); done

# /target に etc ディレクトリを作成し、各種設定ファイルを作成しておきます。
mkdir /target/etc

# HTTP プロキシが必要な環境に接続されている場合、プロキシの設定を行います。
export http_proxy=http://myproxy.co.jp:8888

# debootstrap にて最小構成の Debian ディレクトリツリーを作成します。
# ここでは etch を指定していますが、sid や lenny を指定することもできます。
debootstrap --arch i386 etch /target http://cdn.debian.or.jp/debian/

# /target に chroot し、インストール環境を設定していきます。
chroot /target

# /target/etc/fstab にインストール後のマウント情報を作成します。
cat <<EOF > /etc/fstab
/dev/cobd0 swap  swap defaults 0 0
/dev/cobd1 /     ext3 defaults 1 1
/dev/cobd2 /home ext3 defaults 1 1
proc       /proc proc defaults 0 0
EOF

# ネットワーク関連を設定します。
# お使いの coLinux の設定に合わせて行ってください。
cat <<EOF >> /etc/network/interfaces
auto lo
iface lo inet loopback
auto eth0
#iface eth0 inet dhcp
iface eth0 inet static
  address 192.168.0.10
  netmask 255.255.255.0
  gateway 192.168.0.1
EOF

echo "nameserver 192.168.0.1" > /etc/resolv.conf

echo "coLinux" > /etc/hostname

cat <<EOF >> /etc/hosts
127.0.0.1      localhost
192.168.0.10   coLinux
EOF

ln -sf /usr/share/zoneinfo/Japan  /etc/localtime

# shadow パスワードを有効にします
shadowconfig on

# apt リポジトリを設定します。
# ここでは etch を指定していますが、sid や lenny を指定することもできます。
cat <<EOF > /etc/apt/sources.list
deb http://cdn.debian.or.jp/debian etch main contrib non-free
deb-src http://cdn.debian.or.jp/debian etch main contrib non-free
EOF

# ロケール関連を設定します。
aptitude update
aptitude -y install console-common
dpkg-reconfigure console-data

aptitude -y install locales
dpkg-reconfigure locales
```

これで基本となるインストール作業は完了したため、Debian インストーラを終了させます。

[ALT] + [F1] キーを押し、ターミナルからインストーラ画面に移動し、以下の手順に従ってインストーラを終了させます。

```
# Debian インストーラを終了させます。
[ESC] キーを押下し、ネットワーク設定画面に戻ります。
もう一度 [ESC] キーを押下し、メインメニューに戻ります。
メニュー一番下の「Abort the installation」を選択し [Enter] を押下します。
* <Yes>
```

これで coLinux が再起動されますが、設定ファイルがインストール起動用となっているため、再び Debian インストーラが起動されますので、一旦 coLinux を終了させます。

12.2.6 coLinux を通常起動用に設定

coLinux への Debian インストールは完了しましたので、通常起動するために coLinux の設定を変更します。

coLinux インストールディレクトリに debian.conf と言う名前でファイルを作成し、以下の内容を記載して保存します。

```
kernel=vmlinux
initrd=initrd-normal.gz
mem=512
cobd0="\DosDevices\C:\coLinux\swap_fs"
cobd1="\DosDevices\C:\coLinux\root_fs"
cobd2="\DosDevices\C:\coLinux\home_fs"
eth0=tuntap
root=/dev/cobd1
ro
```

あとは、この設定ファイルを使用して coLinux を起動すると、先程インストールしたオリジナルイメージで起動することができます。コマンドプロンプトを起動し、`c:\coLinux` に cd してから、以下のコマンドを実行します。

```
colinux-daemon.exe -t nt "@c:\coLinux\debian.conf"
```

12.3 Xming による X アプリケーション起動方法

coLinux にはディスプレイデバイスが実装されていないため、そのままでは CUI 環境でしか使用することができません。しかし、X サーバや VNC、ssh の X11 フォワードなどの機能を使用すると、X アプリケーションを使用することができます。

今回は Xming を使用した、ssh の X11 フォワードと XDMCP について紹介したいと思います。

ssh の X フォワードでは必要ないのですが、XDMCP 接続では、ログインマネージャを使用する必要がありますので、ログインマネージャとして GDM を使用した XDMCP 接続について説明したいと思います。

大まかな手順は以下のようになります。

1. Xming のインストール
2. X アプリケーションのインストール
3. ssh の X フォワードで起動
4. XDMCP で起動

12.3.1 Xming のインストール

Xming の配布サイト[*33]から、Xming のインストーラ[*34]とコア X フォントのインストーラ[*35]をダウンロードし、インストールします。

[*33] `http://sourceforge.net/projects/xming/files`
[*34] 執筆時では Xming-6-9-0-31-setup.exe が最新版となっています
[*35] 執筆時では Xming-fonts-7-3-0-22-setup.exe が最新版となっています

12.3.2 X アプリケーションのインストール

起動する X アプリケーションのインストールを行います。ここでは iceweasel をサンプルとして起動してみたいと思いますので、iceweasel をインストールしておきます。

また、XDMCP で起動する方法もご紹介いたしますので、XDMCP 接続時のログインマネージャとして GDM をインストールします。

```
sudo aptitude install ssh gdm iceweasel
```

GDM をインストールすると、Debian 起動時に GDM が起動しようとするのですが、ディスプレイデバイスを持たない coLinux では GDM 起動に失敗してしまい、図 6 のようなエラー画面が表示されます。

図 6　ターミナルに移動

エラーが表示されても、問題なく使用できるですが余り気持ちの良いものではないため、GDM のインストールが完了したらすぐに、GDM が起動しないように設定します。

??を以下のように書き換えます。

```
   [servers]
-  0=Standard
+  #0=Standard
```

次に/etc/gdm/gdm.conf を以下のように書き換えます。

```
   [xdmcp]
+  Enable=true
```

設定がうまく行えたことを確認するため、一旦 Debian を再起動し、エラーが表示されないことを確認します。

12.3.3 ssh の X フォワードで起動する

Xming には、XLaunch という Xming を指定した設定で起動するためのツールが附属しています。ここでは XLaunch を使用して起動してみたいと思います。まず XLaunch を起動すると、図 7 のような Display setting 画面が起動されます。

起動方法が 4 種類あるのですが、表 7 のような種類があります。

Fullscreen や One window は XDMCP でも起動できますので、ここでは Multiple window で起動してみます。

Multiple window を選択して「次へ」を選択すると、Session type 画面が表示され、セッションタイプについて聞

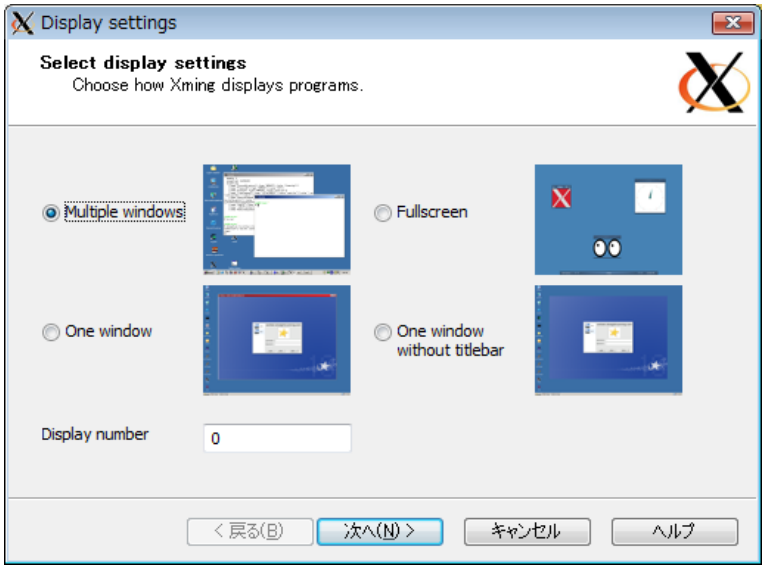

図 7　XLaunch 起動

モード	表示方法
Multiple window	起動するアプリケーション毎に window が起動します
Fullscreen	X Window System がフルスクリーンで起動します。
One window	1 つの window に X Window System が起動します
One window whithout titlebar	One window と同様ですが、タイトルバーが表示されません

表 7　ディスプレイ設定の選択

かれますので、SSH を使用する Start a program を選択して「次へ」を選択します。

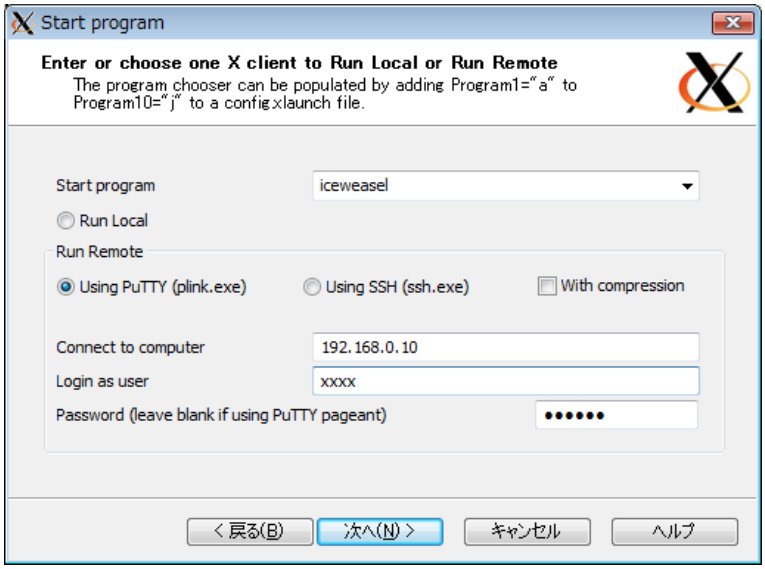

図 8　XLaunch 起動

すると図 8 のような Start program 画面が表示されますので、Start program に起動する X アプリケーションを指定します。ここでは iceweasel を起動するため、iceweasel と入力します。Using PuTTY(plink.exe) を選択し、Connect to computer に coLinux の IP を入力します。coLinux の名前解決ができるようであればホスト名を入力し

ても問題ありません。Login as user にユーザ名を入力し、パスワードを入力し、「次へ」を選択します。

Additional parameters 画面が表示されますが、特に指定する必要はありませんので、そのまま「次へ」を選択します。

Finish configuration 画面が表示されますので、「完了」を選択すると、iceweasel が起動します。「Save configuration」を選択すると、今回行った設定を保存することができます。Include PuTTY Password as insecure clear text にチェックを入れると、設定ファイルにパスワードを保存することができますが、平文で保存されますので注意が必要となります。

保存された設定ファイルをダブルクリックすると、Xming が起動され、ssh で接続を行い、iceweasel を簡単に起動することができるようになります。

12.3.4 XDMCP で起動する

XDMCP の起動では、表 7 のモードの内、Multiple window 以外の起動方法で表示することができます。ここでは One window で起動してみます。

XLaunch を起動し、Display setting 画面にて One window を選択して「次へ」を選択すると、Session type 画面が表示され、セッションタイプについて聞かれますので、Open session via XDMCP を選択して「次へ」を選択します。

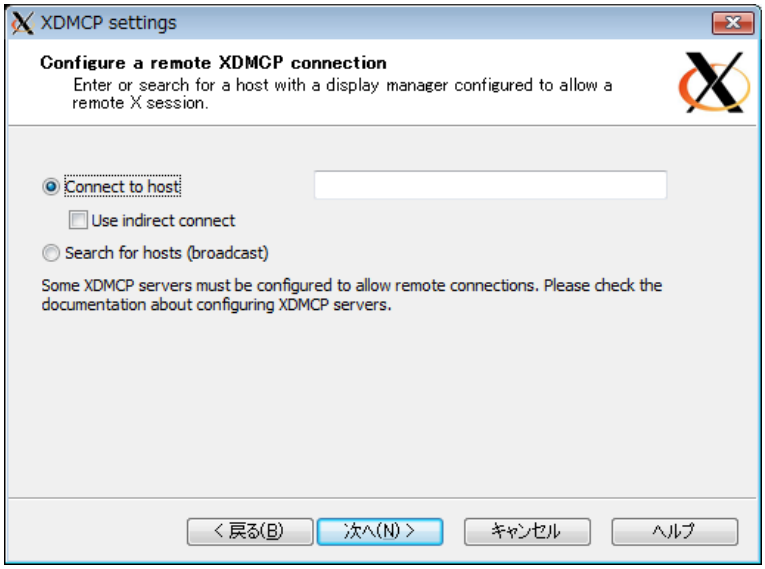

図 9　XLaunch 起動

すると図 9 のような XDMCP setting 画面が表示されますので、Connect to host に coLinux の IP アドレスを入力します。coLinux の名前解決ができるようであればホスト名を入力しても問題ありません。Additional parameters 画面が表示されますが、特に指定する必要はありませんので、そのまま「次へ」を選択します。

Finish configuration 画面が表示されますので、「完了」を選択すると、GDM が起動し、ログイン画面が表示されます。

「Save configuration」を選択すると、今回行った設定を保存することができます。ログインが完了すると、GNOME 環境が起動しますので、Application → Internet → Iceweasel Web Browser を選択し、iceweasel を起動します。

以上駆け足でしたが、coLinux に Debian を新規インストールしたり、ディスプレイデバイスを持たない coLinux での X アプリケーションの起動について紹介いたしました。

13 Debian Live に Ubiquity を移植できるか？

山下 尊也

13.1 Ubiquity パッケージをビルドする

http://jp.archive.ubuntu.com/ubuntu/ から ubiquity のソースを取ってきます。

今回、私は、ubiquity_1.8.12.tar.gz をダウンロードしてきました。

なんとか、snapshot.debian やソースパッケージを用いて、依存関係を解決しました。[*36]

しかし、ビルドの行程で、/usr/share/iso-codes/iso_3166.tab が存在しないと怒られます。iso_3166.tab は、地理情報を表しているものです。調べてみたところ、sarge や etch には iso-3166-udeb と言う形で提供されていますが、lenny からは、iso-3166-udeb パッケージが廃止されています。

調べていく過程で、tzdata パッケージに、iso3166.tab と言うファイルが存在する事が分かりました。中身を見たところ、これを利用すれば Debian でもビルド出来そうです。今回は、時間がなかったため、iso_3166.tab を Ubuntu Hardy が起動している環境から持ってきて、利用しました。

なんとか、以下のパッケージを作成する事が出来ました。

- ubiquity-frontend-gtk_1.8.12_i386.deb
- ubiquity-frontend-kde_1.8.12_all.deb
- ubiquity-frontend-mythbuntu_1.8.12_all.deb
- ubiquity-ubuntu-artwork_1.8.12_all.deb
- ubiquity_1.8.12_i386.deb

また、d-i/source/localechooser/mkshort を変更し、/usr/share/zoneinfo/iso3166.tab を用いる変更を加えれば、この問題は解決出来そうです。

13.2 Ubiquity で必要になるパッケージ

13.2.1 Ubiquity 関係パッケージの Depends で関係するもの

Ubuntu Hardy が起動している環境から、/var/lib/apt/lists/ ディレクトリ下にある

jp.archive.ubuntu.com_ubuntu_dists_hardy_main_binary-i386_Packages

を持ってきて、ubiquity 関連パッケージの Depends に書かれているものを書き出したものが以下のものです。

[*36] rules の変更だけしたら大丈夫かも？本来はちゃんとした依存関係を調べないといけません。

```
adduser
console-setup
debconf
grub
iso-codes
kwin
laptop-detect
libatk1.0-0
libc6
libcairo2
libdebconfclient0
libdebian-installer4
libffi4
libglib2.0-0
libgtk2.0-0
libpango1.0-0
libparted1.7-1
libxml2
lsb-release
os-prober
passwd
python
python-apt
python-central
python-glade2
python-gtk2
python-qt4
sudo
ubiquity
ubiquity-artwork-1.8.7
ubiquity-casper
ubiquity-frontend-1.8.7
x-window-manager
```

同様に、Debian Lenny が動いている環境の/var/lib/apt/lists/にある

- cdn.debian.or.jp_debian_dists_lenny_main_binary-i386_Packages
- cdn.debian.or.jp_debian_dists_lenny_contrib_binary-i386_Packages
- cdn.debian.or.jp_debian_dists_lenny_non-free_binary-i386_Packages

をチェックしたところ、Debian Lenny には以下のパッケージが存在しない事が分かりました。

```
libffi4
libparted1.7-1
ubiquity
ubiquity-artwork-1.8.7
ubiquity-casper
ubiquity-frontend-1.8.7
x-window-manager
```

libffi4 は、etch に存在し、libparted1.7-1 は、ソースからビルドすれば、パッケージの作成をする事が出来ます。[*37]

x-window-manager は仮想パッケージのため、このリストには存在しません。と言うことで、依存関係は解決出来たみたいです。

13.3 Debian Live に依存関係を解決し、入れてみる

OSC Kansai で配布した Debian Live で使われていたものをある程度、利用したいため、git hub から取ってきて、13.2.1 で追加したリストを追記します。

```
% git clone git://github.com/nogajun/debian-study-live-cd.git
% vi debian-study-live-cd/config/chroot_local-packageslists/06-ubiquity
```

また、GTK の環境だけを考えているため、以下のパッケージをインストールする事を考えます。

```
ubiquity-frontend-gtk_1.8.12_i386.deb
ubiquity-ubuntu-artwork_1.8.12_all.deb
ubiquity_1.8.12_i386.deb
```

debian-study-live-cd/config/chroot_local-packages| ディレクトリに入れ、パッケージの正当性チェックを無効にしておきます。

[*37] 現在、automake1.8 とのバグのため、ビルド出来ない状況になっているため、今回は、snapshot.debian からパッケージをダウンロードしました。

```
% sudo lh_build
```

しかし、ここでまたもや、依存関係のエラーがでます。

```
libffi4: Depends: gcc-4.1-base (= 4.1.1-21) but it is not going to be installed
ubiquity: Depends: ubiquity-casper but it is not installable
```

とりあえず、リストの前に追加しますが、ubiquity-casper は、Ubuntu にしかありません。ubiquity-casper は、live installer からの設定を行うものですが、これをソースからビルドすると

```
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
The following information may help to resolve the situation:
 casper: Depends: busybox-initramfs (>= 1:1.1.3-4ubuntu3) but it is not installable
         Depends: localechooser-data but it is not installable
 libffi4: Depends: gcc-4.1-base (= 4.1.1-21) but it is not going to be installed
E: Broken packages
```

Ubiquity の問題ではなく、依存関係をどうするかを考えた方が良かったみたいです。少しずつ時間を見つけて、依存関係を見直していこうと思います。

14 10 分でわかる Debian フリーソフトウェアガイドライン (DFSG)

木下 達也

14.1 Debian フリーソフトウェアガイドラインとは

ある著作物が「フリー」かどうか、フリー (自由な)OS である Debian の構成要素として適しているかどうかを判定する際の基準、それが Debian フリーソフトウェアガイドライン (DFSG: Debian Free Software Guidelines) です。

`http://www.debian.org/social_contract#guidelines`

フリー (自由な) ソフトウェアとは、自由に使ったり、変更したり、コピーして配ったりできるソフトウェアのことをいいます。

ただし、「自由」とはいっても、無制限というわけではなく、ある種の制約は認められています。(著作権表示・ライセンスの維持、変更の明示など)

14.2 著作権とライセンス

著作物には、基本的に著作者に対して「著作権」が発生しています。

つまり、著作物を変更したりコピーして配ったりするには、著作者からの許可 (ライセンス) が必要になります。

ライセンスは次のように確認します。

- 変更や配布が許可されているか
- それらに付随する制約はどうか

その他、個別の事情も検討する必要があります。(特許、商標など)

14.3 Debian フリーソフトウェアガイドラインの説明

- 1. 自由な再配布 (Free Redistribution)
  有償・無償を問わず、別途の許可を必要とすることなく、プログラムを複数まとめて配布できる。
  (Artistic License: プログラム単体への課金は禁止、複数まとめての販売は可)
- 2. ソースコード (Source Code)
  ソースコード (変更に適した形式) が必要。
  実行形式だけでなくソースコードでも配布できる。
- 3. 派生ソフトウェア (Derived Works)
  同様のライセンスで変更版を配布できる。
  (GNU GPL: 変更版全体に同様のライセンスを強制)
  (BSD License: 変更版全体には別ライセンスの適用可)
- 4. 原作者によるソースコードの整合性維持 (Integrity of The Author's Source Code)
  変更版のソースコードを配布する場合に、元のソースコードと差分 (パッチ) という形式のみ許可という制約は許容、ただし非推奨。
  (QPL: 元のソースコードと差分の要求)
- 5. すべての個人、団体の平等 (No Discrimination Against Persons or Groups)
  いかなる個人・団体も差別せずに許可。
- 6. 目標分野の平等 (No Discrimination Against Fields of Endeavor)
  商用・非商用・平和利用・軍事目的など、用途を制限しない。
- 7. ライセンスの配布 (Distribution of License)
  ライセンスは、再配布されたすべての人々に、別途の許可を必要とすることなく、適用される。
- 8. ライセンスは Debian に限定されない (License Must Not Be Specific to Debian)
  Debian の一部としてのみの許可ではなく、他のシステムにも適用できるように。特定の製品に依存しないように。
- 9. ライセンスは他のソフトウエアを侵害しない (License Must Not Contaminate Other Software)
  たとえば、同じ媒体で配布されるソフトウエアすべてがフリーソフトウエアであることを要求しないように。
- 10. フリーなライセンスの例 (Example Licenses)
  GNU GPL, BSD License, Artistic License
  (私見としては、Artistic License は推奨しません。Free Software Foundation では「曖昧過ぎる」として Non-Free に分類されています)
  `http://www.gnu.org/licenses/license-list.ja.html#ArtisticLicense`

15 はじめての CDBS

佐々木 洋平

15.1 はじめに

CDBS – **C**ommon **D**ebian **B**uild **S**ystem は、`debian/rules` の共通部分を抽出し共有することで、`debian/rules` を簡潔かつ理解しやすくすることを目的としたツールです。[CDBS Documentation Rev.0.1.2] の Introduction によれば、 当初の開発動機は GNU autoconf & GNU automake を使用している Debian パッケージの重複した `debian/rules` をなんとかしたい、 だった様です:

> The motivating factor for CDBS was originally that more and more programs today are created using GNU Autoconf configure scripts and GNU Automake, and as such they are all very similar to configure and build. It was realized that a lot of duplicated code in everyone's debian/rules could be factored out. ...

今では CDBS はもっと汎用的になっており、autotools に限らず、 Gnome、 KDE、 Python、 Haskell など、様々なパッケージの作成に使用できるようになっています。[Online CDBS Gallery] によると、現時点で CDBS を使用しているソースパッケージは 3145 個、ソースパッケージ全体の ～ 23% だそうです。

ドキュメントで挙げられている CDBS の利点は以下の通りです[*38]:

1. 簡潔で、可読性が高く、効率的な `debian/rules` を作成できる。
2. debhelper と autotools の呼び出しを自動化することによって、繰り返し作業を気にする必要がなくなる。
3. カスタマイズ性に限界が無いので、メンテナはパッケージ作成時のより重要な問題に集中できる。
4. 提供されるクラスは十分にテストされており、よくある問題を回避するための汚い hack をする必要がない。
5. 既存のパッケージを CDBS へ移行するのは容易である。
6. 拡張性が高い。

確かにそうなんですけれども、以下の例では逆に何をしているのかわからなくて不安になったりします:

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

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk
```

ここでは [CDBS Documentation Rev. 0.4.0] の冒頭をざっくり解説してみようと思います。

15.2 CDBS のディレクトリ構成& ファイル群

CDBS の実態は細分化された Makefile 群です。実際に `/usr/share/cdbs/` 以下を眺めてみましょう:

[*38] [CDBS Documentation Rev. 0.4.0] の CDBS advantages の超訳です - -;)

```
% ls -aR /usr/share/cdbs
/usr/share/cdbs/:
./  ../  1/

/usr/share/cdbs/1:
./  ../  class/  rules/

/usr/share/cdbs/1/class:
./                  autotools-vars.mk  hbuild.mk          perlmodule-vars.mk
../                 autotools.mk       kde.mk             perlmodule.mk
ant-vars.mk         cmake.mk           langcore.mk        python-distutils.mk
ant.mk              docbookxml.mk      makefile-vars.mk   qmake.mk
autotools-files.mk  gnome.mk           makefile.mk

/usr/share/cdbs/1/rules:
./   buildcore.mk  debhelper.mk  patchsys-quilt.mk   tarball.mk
../  buildvars.mk  dpatch.mk     simple-patchsys.mk  utils.mk
```

中間ディレクトリ **1** は将来の API 変更を考慮したディレクトリです。rules、 class ディレクトリ以下にあるファイルが細分化された Makefile です。パッケージを作成する際にはこれらのファイルを適宜 `debian/rules` 内で include して使用します。個々の Makefile の依存関係は図 10 の通りです:

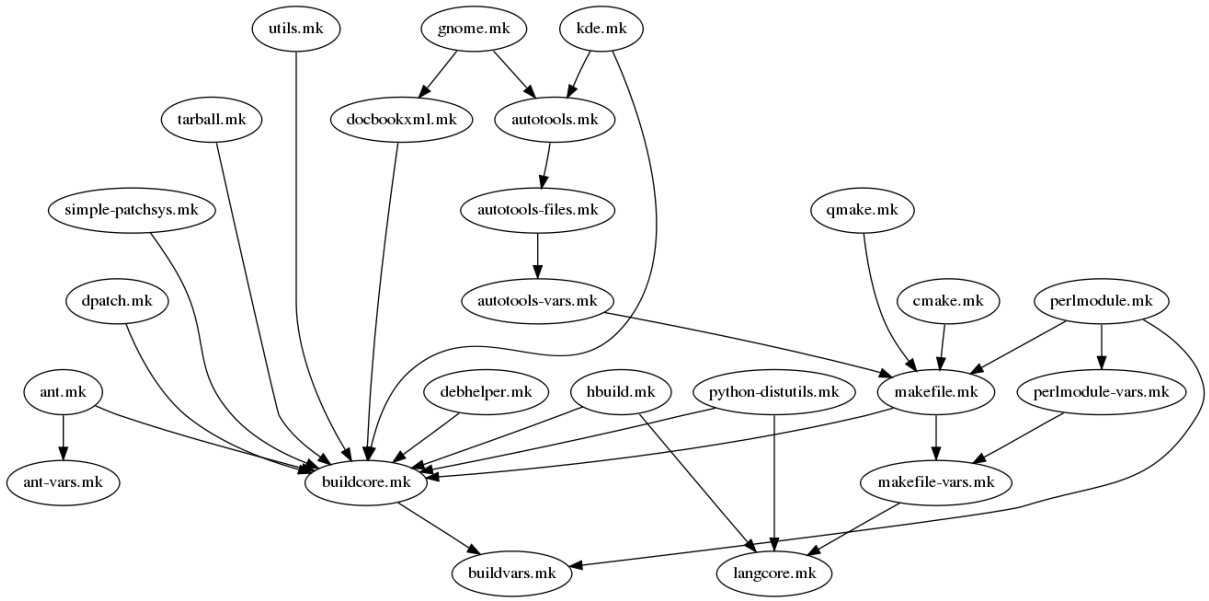

図 10　CDBS で提供される Makefile の依存関係 ([CDBS Documentation Rev.0.1.2]).

15.3　基本となる rules

15.3.1　buildvars.mk

buildvars.mk を include するとパッケージ作成のための環境変数が設定されます。設定される変数の一覧を表 8 に記載します。良く使われるのは `CURDIR` と `DEB_DESTDIR` でしょう。これらの変数を変更したい場合には、buildvars.mk を include した後で `debian/rules` 内部で以下の様に設定します:

```
# where sources are
DEB_SRCDIR = $(CURDIR)/src
# in which directory to build
DEB_BUILDDIR = $(DEB_SRCDIR)/build
# in which directory to install the sofware
DEB_DESTDIR = $(CURDIR)/destination
```

15.3.2　buildcore.mk によるターゲットの設定

buildcore.mk はパッケージ作成のための基本的なターゲットを提供します。[Debian Policy Manual] によれば、パッケージ作成の際 `debian/rules` ファイルの中で外部から参照されうるターゲットは、以下の通りです:

- build

表 8　/usr/share/cdbs/1/rules/buildvars.mk で設定される変数

CURIDR	パッケージを作成しているディレクトリの名前。
DEB_SOURCE_PACKAGE	ソースパッケージの名前。
DEB_VERSION	完全な Debian Version。
DEB_NOEPOCH_VERSION	Debian version without epoch 。
DEB_ISNATIVE	native パッケージの場合は空ではない (条件分岐に使用)。
DEB_ALL_PACKAGES	作成される全てのパッケージのリスト。
DEB_INDEP_PACKAGES	アーキテクチャに依存しないパッケージのリスト。
DEB_ARCH_PACKAGES	アーキテクチャに依存するパッケージのリスト。
DEB_PACKAGES	通常の (udeb ではない) パッケージのリスト。
DEB_UDEB_PACKAGES	udeb の場合、 そのリスト。
DEB_ARCH	Debian アーキテクチャ。 後方互換のために残されている。
DEB_HOST_ARCH_CPU	Debian アーキテクチャの CPU 情報。
DEB_HOST_ARCH_OS	Debian アーキテクチャの OS 情報。
DEB_DESTDIR	パッケージ作成の際にソフトを install するディレクトリ。
	単一パッケージの場合は $(CURDIR)debian/**パッケージ名**
	複数のパッケージを作成する場合には $(CURDIR)debian/tmp

- binary, binary-arch, binary-indep
- clean
- build-arch, build-indep (optional)
- get-orig-source (optional)
- path (optional)

buidcore.mk を include するとこれらのターゲットが **細分化されて**提供されます。buildcore.mk が提供するターゲットの一覧を図 11 に示します。

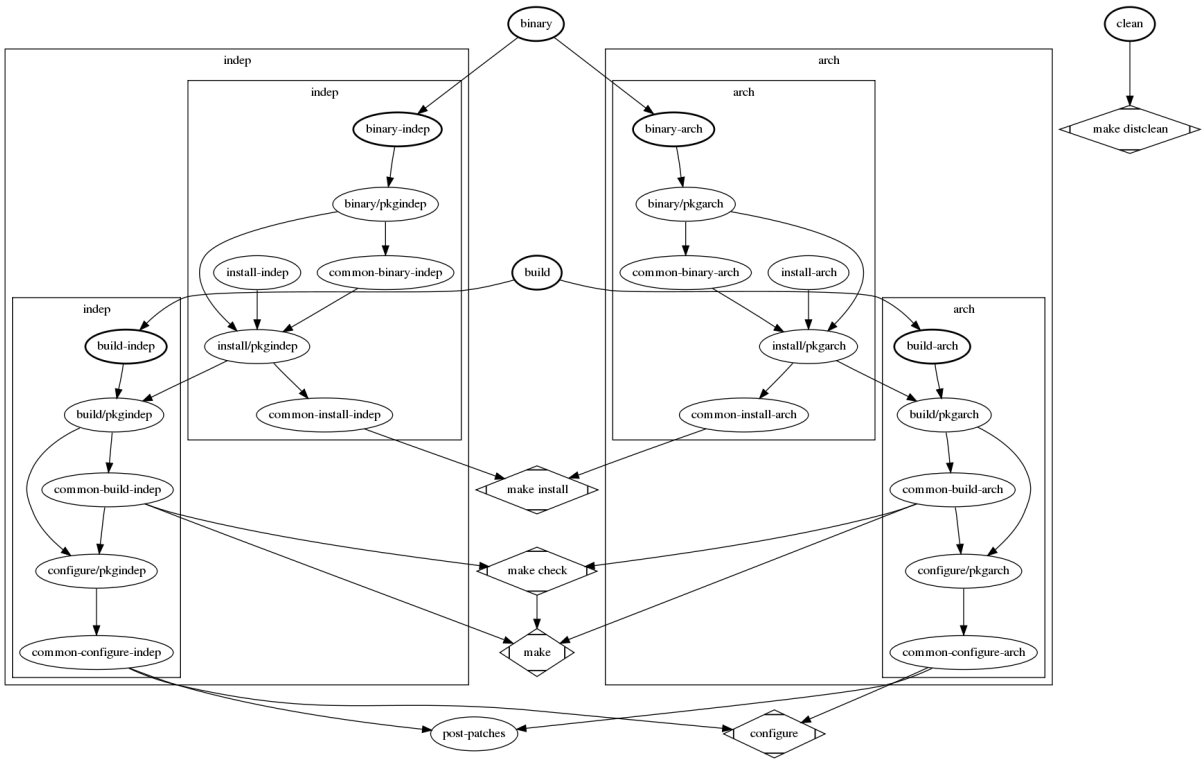

図 11　buildcore.mk で提供されるターゲットの流れ ([CDBS Documentation Rev.0.1.2])。

実際には、 buildcore.mk 自体はパッケージに対してなんの処理もしません。よって適宜 rules を記述することになります。例として、 [CDBS Documentation Rev. 0.4.0] に記載されている foo について解説します。ここで foo は

- ソースパッケージ foo を生成
- バイナリとして foo(arch-dep) と foo-data(arch-indep) を生成

するパッケージだとします。

```
#!/usr/bin/make -f
include /usr/share/cdbs/1/rules/buildcore.mk

# pre-configure action
makebuilddir/foo::
        ln -s plop plop2
# post-configure action
configure/foo::
    sed -ri 's/PLOP/PLIP/' Makefile
configure/foo-data::
    touch src/z.xml
# post-build action
build/foo::
    /bin/bash debian/scripts/toto.sh
build/foo-data::
    $(MAKE) helpfiles
# post-install action
install/foo:
    cp debian/tmp/myfoocmd debian/foo/foocmd
      find debian/foo/ -name ``CVS'' -depth -exec rm -rf {} \;
install/foo-data:
    cp data/*.png debian/foo-data/usr/share/foo-data/images/
    dh_stuff -m ipot -f plop.bz3 debian/foo-data/libexec/
# post deb action
binary/foo:
    strip --remove-section=.comment --remove-section=.note --strip-unneeded \
      debian/foo/usr/lib/foo/totoz.so
# pre-clean action
cleanbuilddir/foo::
    rm -f debian/fooman.1
```

コメントとしてターゲットを記述するタイミングを記載しました。ターゲットの記法は

target/packagename::

です。 後ろの **::** が重要です。

15.3.3 debhelper.mk による dh_ の自動化

CDBS の一番の御利益が、 この debhelper.mk です。CDBS では主な dh_ コマンドの呼び出しを debhelper.mk において行なうため、`debian/rules` 内の dh_ の殆んどが不要になります。debhelper.mk によって呼び出される dh_ を表 9 に示します。

表 9 `/usr/share/cdbs/1/rules/debhelper.mk` で管理される dh_ コマンド

dh_builddeb	dh_installchangelogs	dh_installemacsen	dh_installman
dh_perl	dh_clean	dh_installcron	dh_installexamples
dh_installmenu	dh_shlibdeps	dh_compress	dh_installdeb
dh_installinfo	dh_installpam	dh_strip	dh_fixperms
dh_installdebconf	dh_installinit	dh_link	dh_gencontrol
dh_installdirs	dh_installlogcheck	dh_makeshlibs	dh_install
dh_installdocs	dh_installlogrotate	dh_md5sums	

debhelper.mk によって呼び出される dh_ コマンドに対しては、パラメータ設定は (大抵の場合) 不要です。debelpher の呼び出しをカスタマイズする変数は debhelper.mk 冒頭のコメント行を参照して下さい。[CDBS Documentation Rev. 0.4.0] には以下の例があります:

依存関係がシビアな共有ライブラリについて

```
DEB_DH_MAKESHLIBS_ARGS_libfoo := -V''libfoo (>= 0.1.2-3)``
DEB_SHLIBDEPS_LIBRARY_arkrpg := libfoo
DEB_SHLIBDEPS_INCLUDE_arkrpg := debian/libfoo/usr/lib/
```

ChangeLog のファイル名が一般的でない場合

```
DEB_INSTALL_CHANGELOGS_ALL := ProjectChanges.txt
```

.py を圧縮せずにパッケージ化する場合

```
DEB_COMPRESS_EXCLUDE := .py
```

ここでの記法

:=

に注意して下さい。:= は上書きです。CDBS の提供する変数に追加する場合には += を使用します。

15.3.4 patch の管理

dpatch、 quilt、 そして CDBS 用の simple-patchsys の rules が提供されています。 quilt、 dpatch については include するだけで patch を適用する rule が適用されます。

simple-patchsys の場合は debian/patches に patch を置くだけで、 パッケージ作成時に patch を適用し clean の際には元に戻します。patch level は 3 まで ok です。 自動的に適用しようと試みます。

15.4 class を使用する

前節で rules について簡単にまとめました。ここでは幾つかの class について紹介します。

15.4.1 Makefile の場合:makefile.mk

autotools を使わず Makefile のみを使用するソフトウェアには makefile.mk が便利です。[CDBS Documentation Rev. 0.4.0] では、元々の Makefile が

- 名前が MaKeFile で
- make mrproper で clean
- make myprog で build
- make check で check
- make install で install

というソフトウェアの場合について例示しています:

```
#!/usr/bin/make -f
include /usr/share/cdbs/1/rules/debheper.mk
include /usr/share/cdbs/1/class/makefile.mk

DEB_MAKE_CLEAN_TARGET    := mrproper
DEB_MAKE_BUILD_TARGET    := myprog
DEB_MAKE_INSTALL_TARGET  := install DESTDIR=$(CURDIR)/debian/tmp/
# no check for this software
DEB_MAKE_CHECK_TARGET    := check
# allow changing the makefile filename in case of emergency exotic practices
DEB_MAKE_MAKEFILE        := MaKeFiLe
# example when changing environnement variables is necessary :
DEB_MAKE_ENVVARS         := CFLAGS=''-fomit-frame-pointer''
```

15.4.2 Autotools の場合: autotools.mk

いわゆる `configure && make && make install` なソフトウェアの場合は autotools.mk が便利です。冒頭にも例示しましたが、標準的な autotools を使用するソフトウェアの場合には

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

include /usr/share/cdbs/1/rules/debheper.mk
include /usr/share/cdbs/1/class/autotools.mk
```

となります。

configure へのオプションや環境変数の設定を行なう場合には以下の様にします:

```
DEB_CONFIGURE_EXTRA_FLAGS := --with-ipv6 --with-foo
COMMON_CONFIGURE_FLAGS := --program-dir=/usr
DEB_CONFIGURE_SCRIPT_ENV += LDFLAGS='' -Wl,-z,defs -Wl,-O1''
```

ここでも += 、 := の意味は変わりません。例えば

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

include /usr/share/cdbs/1/rules/debheper.mk
include /usr/share/cdbs/1/class/autotools.mk

# normally
DEB_MAKE_INSTALL_TARGET := install DESTDIR=$(DEB_DESTDIR)
# example to work around dirty makefile
# DEB_MAKE_INSTALL_TARGET := install prefix=$(CURDIR)/debian/tmp/usr
DEB_MAKE_CLEAN_TARGET := distclean
# example to activate check rule
DEB_MAKE_CHECK_TARGET := check
# overriding make-only environnement variables :
# (should never be necessary in a clean build system)
# (example borrowed from the bioapi package)
DEB_MAKE_ENVVARS    := ''SKIPCONFIG=true''
```

など。

他にも Perl、 Python、 Ruby、 GNOME、 KDE、 Ant、 HBuild(Haskell) 用の class があります。

15.5 まとまってないまとめ

そんな所で締切の時間が来てしまいました。

CDBS を使いはじめたら、 もう debhelper には戻れない体になってしまうわけですが、 いかんせん CDBS ってドキュメント少ないんですよね。この文書が最初の一歩になれば幸いです。

参考文献

[CDBS Documentation Rev. 0.4.0] Marc (Duck) Dequénes, Arnaud (Rtp) Patard, 2007: CDBS Documentation, `http://perso.duckcorp.org/duck/cdbs-doc/cdbs-doc.xhtml`

CDBS の online ドキュメンテーションです。パッケージに含まれているドキュメントより、こっちの方が情報が多く、 参考になります。

[CDBS Documentation Rev.0.1.2] Marc (Duck) Dequénes, Arnaud (Rtp) Patard, Peter Eisentraut, Colin Walters, 2007: `/usr/share/doc/cdbs/cdbs-doc.html`.

Lenny の CDBS(ver. 0.4.52) 付属のドキュメントです。Web で公開されているドキュメントよりは古いです。

[CDBS 移行への 1st step] Tatsuki Sugiura, 2006: CDBS 移行への 1st step `http://sugi.nemui.org/doc/cdbs/cdbs-trans-1st.html`

既存のパッケージを CDBS へ移行する場合、非常に参考になると思います。

[Online CDBS Gallery] 本家: Online CDBS Gallery, `http://cdbs.ueberalles.net/index.html`

CDBS を使っている `debian/rules` を見ることができます。新たに CDBS へ移行する際に参考になるでしょう。 ちなみに、 本家は 2006 年で更新が止まっている模様。 岩松さんが nigauri.org に CDBS ギャラリを上げて更新して下さいました。`http://www.nigauri.org/~iwamatsu/cdbs/archive/site/`

[Debian Policy Manual] Ian Jackson & Christian Schwarz, 1996: Debian Policy Manual, `http://www.debian.org/doc/debian-policy/`

[Debian パッケージ作成の手引き] 小林 儀匡, Debian パッケージ作成の手引き, `http://www.debian.or.jp/~nori/debian-packaging-guide/index.html`

Debian パッケージ作成の手引きです。「debhelper を使わない場合 →使う場合 → CDBS への移行」と順序立てて説明されています。

[やまだ & 鵜飼 (2006)] やまだあきら (著), 鵜飼文敏 (監修), 2006: 入門 Debian パッケージ, 技術評論社, ISBN4-7741-2768-X

apt の使い方や Debian パッケージの作り方などを順を追って解説しています。

16 cdn.debian.or.jp, cdn.debian.net における取り組み

荒木 靖宏

いつでも必要なソフトウェアやコンテンツを安価に入手する手段として CDN が広くつかわれている。 Debian は deb の安定入手手段の有無がシステム の信頼性を左右するシステムであり、その特殊性を考慮した CDN システムが必要となる。 今回は cdn.debian.or.jp, cdn.debian.net における取り組みを紹介する。

16.1 CDN とは

Content Delivery Network（CDN）はウェブコンテンツ配置および配送方法として Akamai 社によりサービスされ広く知られることになった。当初から一部の人気の高いサーバへのトラフィック集中によるサーバ停止の回避、海外のリッチコンテンツ取得の高速化、トラフィック分散によるネットワークおよびサーバの利用平準化などの理由で広く受け入れられた。

CDN という用語自体は WWW に限ることなく、一般にコンテンツを取得するための配送手段や方法全体を指す場合がある。たとえば、Winny や Bittorrent などのコンテンツを取得するために特別に設計されたプロトコルを用いて、P2P ネットワークを構成するような手法も含まれる。

16.2 Debian における CDN の現状

16.2.1 利用法とユーザから見た動作

cdn.debian.or.jp では Debian でインストール時から広く deb ファイルの入手に使われる apt で使える CDN として設計し、運用している。そのため、Debian における CDN の利用法は至極簡単である。/etc/apt/source.list に記述する APT リポジトリとして、

```
deb http://cdn.debian.or.jp/debian/ stable main contrib non-free
deb-src http://cdn.debian.or.jp/debian/ stable main contrib non-free
```

以上のように指定するだけでユーザは今までとなんら変わることなく apt コマンドを使用できる。

現在、cdn.debian.or.jp, ftp.jp.debian.org, cdn.debian.net の名で運用している。

サービス時の手順と構成は以下のようになる。（図 12）

1. ユーザが apt-get コマンドを行うと cdn.debian.or.jp を DNS で問い合わせる
2. cdn.debian.or.jp を管理する DNS はサーバ候補 (surrogate) 選択する
3. 選択結果を DNS のリプライとして返す
4. apt は cdn.debian.or.jp として Surrogate C を使用する。

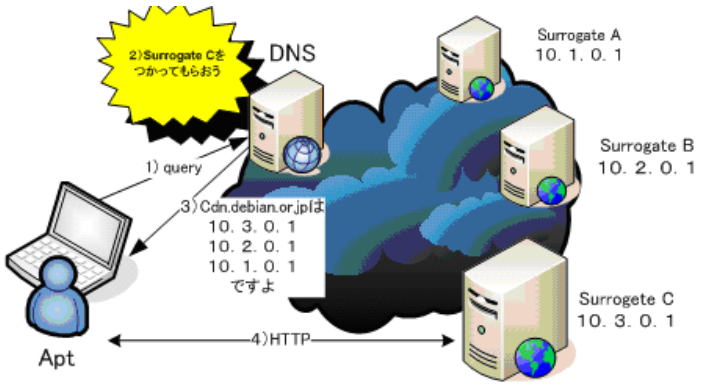

図 12　ユーザから見た cdn.debian.or.jp の動作

16.2.2　構築の方法

実際にはクライアント DNS は次のように動作している。

1. クライアント DNS が cdn.debian.or.jp を検索すると、CNAME deb.cdn.araki.net が debian.or.jp の DNS（BIND）から返答される。
2. クライアント DNS は deb.cdn.araki.net を得るために cdn.araki.net の NS レコードを問い合わせて、ns.cdn.araki.net, plat.debian.or.jp, debian.topstudio.co.jp, osdn2.debian.or.jp を得る。
3. クライアント DNS はいずれかのホストに deb.cdn.araki.net の A レコードを問い合わせて、サロゲートの IP アドレスを dns_balance から得る。
4. クライアント DNS は、ユーザクライアントに 3 で得た情報を返す。

2 を実現するための Araki.net の BIND 設定は次のようになる。

```
cdn             IN      NS      ns.cdn
                IN      NS      plat.debian.or.jp.
                IN      NS      debian.topstudio.co.jp.
                IN      NS      osdn2.debian.or.jp.
```

DNS Balance はユーザの *IP* アドレスと、何らかの方法でサーバをランクづけした表を元にユーザが接続すべきサイトを指示します。この表は一定時間毎に読み込み直され、これにより動的な負荷分散が可能です。

（DNS　Balance の配布ページ[*39] より）

3 を実現するための dns_balance の設定は次のようなサーバをランク付けした ruby 形式のファイルである。

```
$addr_db = {
  "default" => {
    "ns.cdn.araki.net" => [
      [[210,157,158,38], 0],
    ],
    "localhost" => [
      [[127,0,0,1], 0],
    ],
 "deb.cdn.araki.net" => [
                [[61,115,118,67], 1000],
                [[133,50,218,117], 10],
                [[202,229,186,27], 20],
                [[133,5,166,3], 10],
                [[130,54,59,159], 10],
                [[210,157,158,38], 9900],
        ], },}
```

[*39] http://www.dnsbalance.ring.gr.jp

この設定ファイルに基づき、ホスト IP アドレスの後ろの数字は優先度であり、1（優先度高）から 9999（優先度最低）までの整数で指定する。

16.2.3 cdn.debian.or.jp のシステムと動作

CDN システムが完全に動作しユーザから使用されるためには、システムが完全なファイルを提供すること、システムが安定して動作すること、そして CDN を使った場合に高速に動作していることが求められる。

提供ファイルの完全性

このために以下二点を満たさねばならない。

- 個々のファイルがコンテンツ提供者たる deb ファイル配布元と同一であること
- apt-get update の結果取得するファイル群がどの Surrogate でも入手できること

前者については、deb はそのファイルの md5 値、sha1 値とともに配布され、ユーザが使用する apt で確認後に利用されるため CDN を使用した場合でも問題にならない。

後者についてはユーザが apt-get update を行ったときに接続する Surrogate と apt-get dist-upgrade を行ったときに接続する Surrogate は同一であるとは限らないため、DNS が Surrogate として返すサーバが保持するファイルは同一である必要がある。cdn.debian.or.jp では Debian プロジェクトで一般に行われている方法と同様に、rsync プロトコルを用い、push ミラーを行っている（図 13）。そのため、cdn.debian.or.jp のサロゲート内で最上流にあるサーバとミラーが同一であることを 2 分毎に rsync ミラー終了時に作成されるスタンプファイルを確認して、同一でないサーバはサロゲート候補から一時的に除外している。

ただし、これはミラーの配送ツリーの管理を行う必要があるため、日本国内のミラーに限定している。

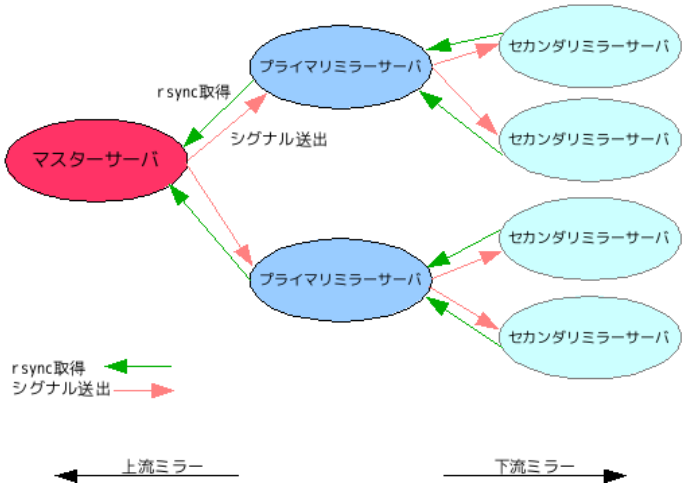

図 13　Debian サロゲートの rsync によるミラー

システムの安定動作

先に述べたように、ユーザは CDN を使用する際には DNS を最初に使用するため、DNS の安定運用がカギとなる。そのため、cdn.debian.or.jp を管理する DNS サーバはまったく独立に動作するサーバで行っている。

また、サーバの動作を確認は、5 秒以内に HTTP のレスポンスを返さないサーバはサロゲート候補から一時的に除外している。

後述するように、サロゲートの生死確認は 2 分ごとに dns_balance の動作に反映される。ローカル DNS にキャッシュされる情報とあわせて最悪でも 3 分以内に動作していないサーバは排除される。

高速動作

cdn.debian.or.jp では DNS で問い合わせされるとサロゲートリストとして複数の IP アドレスを返す。この IP アドレスはラウンドロビンで選択しているわけではなく、提供可能なサーバキャパシティやネットワーク速度を考慮し、設定している。

16.2.4 IP アドレスの位置情報を使用したサーバ選択

当初は cdn.debian.or.jp として運用していたものの、その後に global に分散する Debian ミラーに対応させた。

global に CDN を展開する場合には地理的に近いサーバ群からある程度絞込むのが有効である。現在、GeoIP など無料で IP と地理情報のマッピング提供者が現れており、debian パッケージになっていることもあり、本システムでは Maxmind の GeoIP を使用している。

Dns_balance には AS あるいはネットワークアドレスごとに振り分ける IP アドレスの指定が可能であるが、

- Debian のミラーは国別地域別に制御されていること
- 16 ビットで指定される AS と AS 間の経路の測定が難しい
- 個々のネットワーク間の経路を指定するのは現実的でない

以上の理由から、dns_balance に接続する IP アドレスの国名、大陸名を逆引きし、その結果をつかって返す A レコードを変更するように dns_balance を拡張している。

大陸別の設定ファイルは次のように、6 大陸別である。

```
yaar@loon3:~/playground/cdn$ ls continent/
AF_deb_cdn_araki_net.rb  EU_deb_cdn_araki_net.rb  OC_deb_cdn_araki_net.rb
AS_deb_cdn_araki_net.rb  NA_deb_cdn_araki_net.rb  SA_deb_cdn_araki_net.rb
```

さらに国別の設定ファイルを置く。

```
yaar@loon3:~/playground/cdn$ ls country/
FRA_deb_cdn_araki_net.rb  JPN_jp_cdn_araki_net.rb
JPN_deb_cdn_araki_net.rb  KOR_deb_cdn_araki_net.rb
```

これらの設定ファイルをつかって、dns_balance が実際に読み込む次のような設定ファイルを 2 分ごとに作成している。サーバの生死確認などはこのタイミングで反映される。

```
$addr_db = {"default"=>{"localhost"=>[[[127, 0, 0, 1], 0]], "deb.cdn.araki.net"=
>[[[61, 115, 118, 67], 1000], [[61, 206, 119, 174], 20], [[202, 229, 186, 27], 2
0], [[203, 178, 137, 175], 9000], [[210, 157, 158, 38], 9900]], "ns.cdn.araki.ne
t"=>[[[210, 157, 158, 38], 0]], "jp.cdn.araki.net"=>[[[61, 115, 118, 67], 1000],
 [[61, 206, 119, 174], 20], [[202, 229, 186, 27], 20], [[203, 178, 137, 175], 90
00], [[210, 157, 158, 38], 9900]]}, "KOR"=>{"deb.cdn.araki.net"=>[[[143, 248, 23
4, 110], 20]]}, "SA"=>{"deb.cdn.araki.net"=>[]}, "EU"=>{"deb.cdn.araki.net"=>[[[
141, 76, 2, 4], 9000]]}, "AF"=>{"deb.cdn.araki.net"=>[]}, "AS"=>{"deb.cdn.araki.
net"=>[[[61, 115, 118, 67], 1000], [[61, 206, 119, 174], 20], [[202, 229, 186, 2
7], 20], [[203, 178, 137, 175], 9000], [[210, 157, 158, 38], 9900]]}, "JPN"=>{"d
eb.cdn.araki.net"=>[[[61, 115, 118, 67], 1000], [[61, 206, 119, 174], 20], [[202
, 229, 186, 27], 50], [[203, 178, 137, 175], 9000], [[210, 157, 158, 38], 9900]]
, "jp.cdn.araki.net"=>[[[61, 115, 118, 67], 1000], [[61, 206, 119, 174], 20], [[
202, 229, 186, 27], 50], [[203, 178, 137, 175], 9000], [[210, 157, 158, 38], 990
0]]}, "NA"=>{"deb.cdn.araki.net"=>[[[204, 152, 191, 39], 9000], [[128, 30, 2, 36
], 9000], [[35, 9, 37, 225], 9000]]}, "OC"=>{"deb.cdn.araki.net"=>[[[150, 203, 1
64, 37], 9000]]}, "FRA"=>{"deb.cdn.araki.net"=>[[[193, 54, 19, 19], 9000], [[194
, 2, 0, 36], 9000]]}}
```

16.3 使用実績

以下 3ヶ月分の plat.debian.or.jp で動作している DNS へのアクセス実績を示す。

- 2008 年 7 月 16 日から 10 月 15 日までの三ヶ月間の利用実績を解析した。
- 本システムでは DNS の A または Any に対して返答を戻すことから、それ以外のクエリに関しては無効なアクセスとして処理している。
- 地域はアクセス元の IP アドレスを GeoIP ライブラリから算出している。
- 全動作ホストは、plat.debian.or.jp, osdn2.debian.or.jp, debian.topstudio.co.jp である。
- ユニークホスト数 20554

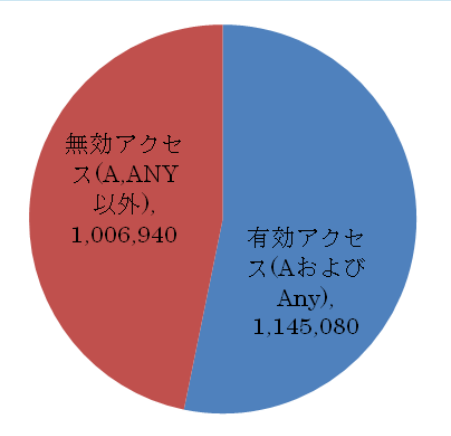

図 14　DNS アクセス種別

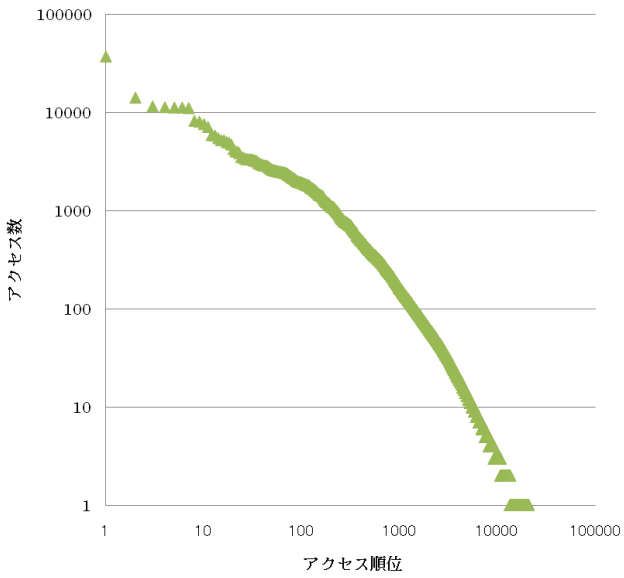

図 15　ホスト別アクセス順位とアクセス数

ユニークホスト数は 20554 であるが、その分布は極端に偏っている。

図 16　DNS クエリ数上位 20 カ国

JPN	808138
USA	87767
CAN	82566
KOR	38596
CHIN	26670
FIN	14558
TWIN	12527
IND	7941
IDN	6463
HKG	5616
RUS	3975
SGP	3656
DEU	2990
GBR	2792
AUS	2635
ESP	2632
THA	2623
MYS	2475
PHL	2359
ITA	2196

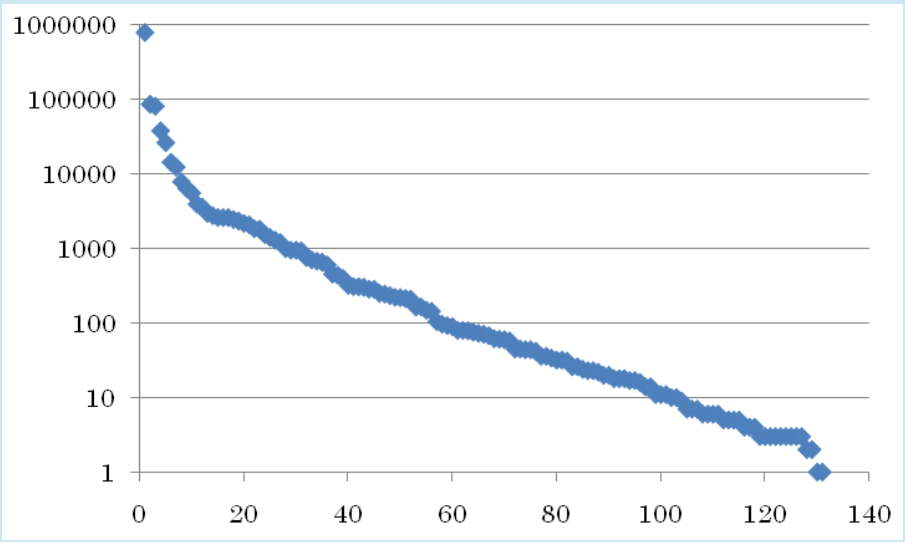

図 17　DNS クエリ数上位国 (横軸)-クエリ数 (縦軸)

なお、国が判定できないのは 1552 件、率にして 0.135% であった。

図 18　CDN 振り分け先実績

地域	アクセス数	
アジア	570137	
アジア (日本と韓国除く)		55539
日本		509367
韓国		5231
ヨーロッパ	16804	
ヨーロッパ (フランス除く)		15154
フランス		1650
北米	142207	
オセアニア	10766	
アフリカ	160	
その他	404996	
総有効アクセス数	1145070	

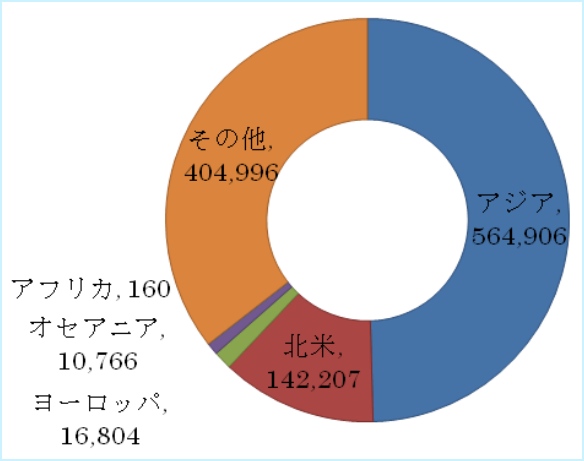

図 19　地域別振り分け実績

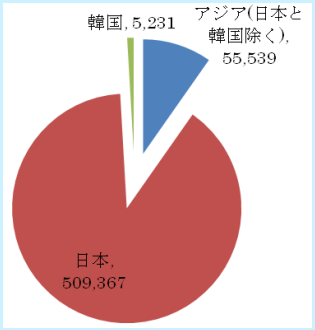

図 20　アジア地域の振り分け実績

図 21　日本での振り分け実績

ホスト名	IP	振り分け数
runner.oyu-net.jp.	61.206.119.174	690596
dwarf.topstudio.co.jp.	202.229.186.27	660091
hanzubin.st.wakwak.ne.jp	61.115.118.67	649512
studenno.kugi.kyoto-u.ac.jp.	130.54.59.159	599550
dennou-q.geo.kyushu-u.ac.jp	133.5.166.3	468730
frost.nemui.org.	218.219.152.77	445589
dennou-h.ep.sci.hokudai.ac.jp.	133.87.45.30	379888
ftp.nara.wide.ad.jp	203.178.137.175	58825
plat.debian.or.jp	210.157.158.38	6095

アジア地域向けのサーバおよび韓国地域向けのサーバは日本向けのサーバ群設定と全く同一のものを設定している。

日本での振り分け実績は、設定した頻度情報をほぼ正確に反映しており、プログラムの動作および、期間中の各サロゲートに大きな障害がなかったことを示している。

16.4　関連研究とディスカッション

`http://prisms.cs.umass.edu/~kevinfu/papers/secureupdates-hotsec06.pdf` でのサーベイ結果のように、パッケージ配布にはいくつかの方法と危険性が指摘されている。このサーベイ結果で指摘されるように、Debian では個々のファイルごとに `sha-1` および `MD5,SHA-256` のハッシュ値を配布している。

加えて、本システムの日本での運用においては、マスターサーバからサロゲートへのファイル同期時に配送ツリーの確認とファイルの一致を確認しており、障害時の排除機能を有するため、単なるファイルのミラーリング以上の安全性は確保されている。

openSUSE における配送（mirrorbrain, `http://en.opensuse.org/Build_Service/Redirector` ）は sourceforge をはじめ多くの CDN で採用している方法である。クライアントへのサロゲート決定の方法は GeoIP を使用しており、本システムと同等のものである。

ただし、Debian の apt システムの制限により、本システムでは HTTP リダイレクトではなく、DNS を使用している。

また、mirrorbrain ではサロゲートへの配送の確認は行っていないものと思われる。

apt の P2P 対応も有力な配布手段である。apt-p2p [*40] により、apt の P2P 対応が進められている。有力な候補である。ただし、Kashimir プロトコルをクライアントで直接使うものであり、ネットワーク利用ポリシーとの競合や install 時に利用できない点、取得希望ファイルが入手できない場合に HTTP にフォールバックするため速度に問題があり、今後検証すべき問題も多い。

サロゲートが正しく動作しているのか、その能力に応じたサロゲート使用ができているかどうかは、分散したファイルサーバからのファイル入手において重要な要素である。この組み合わせを考えると、表 10 のように分類される。本システムでは、サーバやサーバとクライアント間のネットワークの実測を行わないこと、ここのサロゲートの運用は独立して行うことでサロゲートへ特別なプログラムを必要としないことで、コストの劇的な削減を可能としている。

16.5　課題と将来の展望

ここまで説明してきた、cdn.debian.or.jp の動作には改善すべき点が多数存在する。改善の展望としていくつか挙げる。

[*40] `http://www.camrdale.org/apt-p2p/`

表 10

	申告ベースの能力	動的測定・実績ベースの能力
申告ベースのヘルスチェック	Debian ミラーサーバリスト	Debian ミラーの口コミ・トライアンドエラー
動的測定・実績ベースのヘルスチェック	本システム	理想的なシステムだがコスト大商用 CDN など

16.5.1 アクセス実績あるいはローカルポリシーに基いた CDN 配置

アクセス実績によると、日本、米国、カナダ、韓国、中国、フィンランド、台湾、インド、インドネシア、ホンコン、ロシアと続く。現状では、大陸毎に設定されたサロゲートは存在するものの、日本、韓国、アメリカを除けば国毎に設定されたサロゲートは存在しない。

このアクセス実績にもとづき、これらの国内のミラーを生かした設定を近い将来行いたい。

apt-get コマンドの HTTP REDIRECT

apt-get コマンドは HTTP REDIRECT に対応していない。

そのため、前述したように mirrorbrain などの CDN は使用することができない。

HTTP REDIRECT は、いったん HTTP GET などで接続してきたクライアントに対して、新たにそのリソースが存在する URL を通知するものである。この仕組みをうまくつかった CDN として、Coral Content Distribution Network (Coral CDN) がある。Coral CDN はサロゲート間で P2P によるファイル配置し、そのインターフェースとして、HTTP を使用し、しかも使用にはインターネットから取得可能なファイルであれば制限をかけていない。さらに、Apache を使った一時配布サーバでは HTTP REDIRECT をつかって Coral　CDN に誘導することも推奨されている。ただし、現状で、Coral CDN を使うために、

```
deb http://cdn.debian.or.jp.nyud.net:8090/debian/ stable main contrib non-free
```

を指定することも可能だが、少なくとも日本においては Coral CDN を担うサロゲートが存在しないこともあって非常に低速である。ただし韓国や中国では広くつかわれており、将来の拡張に使用したい。

マスターサーバからサロゲートへのミラーリング方法

本システムでは既存の rsync によるミラーリング手法には手をつけていない。行ったのは rsync をつかってタイムスタンプを確認し、サロゲートのヘルスチェックのひとつに使用したのみである。debian における rsync の使い方現状のミラーリングに由来する問題は

1. master.debian.org から末端までのあいだは冗長化されていない
2. rsync はディレクトリの同期を取る方法であり、ファイルの転送に必ずしも適していない。deb のように依存関係を記述したメタデータを含むファイルであれば、ファイル毎の push であっても配送に問題はない
3. 配送がユニキャストである
4. 利用頻度や重要度にかかわらず同様の配送を行っている

等さまざまである。

Debian が潤沢なネットワーク環境と強力な CPU を要するホストでない場合でもミラーリングないしサロゲートからのファイル入手を可能とする FLUTE などの配送手法が必要になろう。

16.6 おわりに

いつでも必要なソフトウェアやコンテンツを安価に入手する手段として CDN はこれからも様々な発展を続けると考える。Debian は deb の安定入手手段の有無がシステムの信頼性を左右するシステムであり、CDN の広範な活用が

今後ますます求められるようになると考える。

本システムの導入により、サロゲート個々の安定性がクライアントに直接影響することはかなり抑えることができるが、より確実なパッケージ入手のために、信頼性が高く安定動作しているサロゲートの増加を Debian プロジェクトでは引き続き求めている。

月例 Debian GNU/kFreeBSD

大浦 真

今月は Debian GNU/kFreeBSD のインストーラについて見てみましょう。

現在、Debian GNU/kFreeBSD には、Debian で使われている Debian Installer (d-i) は用意されていません。d-i を移植する計画はあるようですが、今はその代わりに、FreeBSD のインストーラを改造したものが使われています。アーキテクチャとしては、i386 向けと amd64 向けが用意されていて、最新版は 2008 年 2 月 18 日にリリースされたものです。

このインストーラを使って実機や QEMU などの仮想マシンにインストールすることができますが、このインストーラは、FreeBSD のインストーラを必要最低限の部分だけ改造したものです。ですので、kFreeBSD のインストールには、注意すべき点がいくつかあります。

まず、インストールの手順が FreeBSD のインストール方法ともだいぶ異なっているという点があります。これは、この稿の末尾に URL を挙げた Install Guide に手順が記載されていますが、普通の FreeBSD のインストーラのつもりでインストールを行うとうまくいかないので注意が必要です。また、インストーラはベースシステムのインストールしか行わないので、root のパスワードの設定、一般ユーザの作成、ネットワークの設定、ftp.debian-ports.org のアーカイブキーの取得などの基本的な設定は全て手動で行う必要があります。ただ、Install Guide に従いさえすれば、比較的簡単にインストールを完了することができますし、使い慣れた Debian システムなので、設定もしやすいでしょう

月例 Nexenta Operating System

上川 純一

先月につづいて Nexenta をいじって見たのでお伝えします。

今月は cowdancer をビルドするところまで、です。まず、ソースコードをビルドしてみましょう。警告も大量に出るのですが、とりあえずは絶望しないためにエラーを眺めてみましょう。

```
$ make
gcc -O2 -Wall -o cow-shell cow-shell.o ilistcreate.o
cow-shell.o: In function
 'main':/export/home/dancer/cowdancer-0.36/cow-shell.c:26:
undefined reference to 'asprintf'
:/export/home/dancer/cowdancer-0.36/cow-shell.c:60: undefined
reference to 'canonicalize_file_name'
collect2: ld returned 1 exit status
make: *** [cow-shell] Error 1
dancer@vm1:~/cowdancer-0.36$
```

まず、canonicalize_file_name, asprintf, dlvsym などの関数が無いという旨のエラーが出ています。これらがどうやら GNU/Linux(glibc) で提供されている拡張で、そのままでは Solaris 上では動かなさそうですね。

ということで、GNU 拡張をどう処理するのかに悩みます:

- dlvsym: ダイナミックライブラリの関数をバージョン指定で読み込む。あきらめて dlsym をそのまま使えば良いんじゃないか?
- canonicalize_file_name: realpath を代わりに使えばよいんじゃないか?
- asprintf: asprintf のバッファを確保してくれるバージョンなので、バッファを最初から用意して sprintf を代替として利用すればよいんじゃないか?

といったところまでいじったところでまた来月。

月例 Debian GNU/Linux eeepc port

岩松 信洋

不定期連載の Debian GNU/Linux eeepc port です。先月、会長から押し付けられた eeepc に Debian を SDHC カードにインストールしてみました。debian-eeepc project によって、インストーラーが用意されており、USB メモリにコピーして利用できるようになっています。インストール自体は Debian のインストーラと同じですが、eeepc に搭載されている無線 LAN のドライバが利用できるようになっています。インストールはさくさく進むと思っていたのですが、どうも SDHC への書き込みが遅いようです。base をインストールするのに 2 時間もかかりました。ネットワークを使ったパッケージの取得までは順調なのですが、パッケージのインストールでかなり時間がかかっています。また、インストールした後のパッケージインストールにも時間がかかります。調べたところ、Linux カーネルのプリエンプションオプションの設定が問題だということがわかりました。変更したカーネルを作ったところサクサク動いています。今度は、カーネルの設定を変更したインストーラを作って試してみようと思います。

月例 Nexenta Operating System

上川 純一

前回は実用的なアプリケーションをビルドするまでに至りませんでした。そこで、今回は dlsym を使って自由自在にライブラリをロードできるようにしてみるところからまずやってみましょう。dlsym でロードできる最低限の共有ライブラリを作成するところから始めてみます。まず、関数一つだけを定義した C のファイルから最低限の共有ライブラリを作成して、それを実行するだけのプログラムと、dlopen 経由で利用するプログラムを作成してみました。

```
// a.c : サンプルの共有ライブラリのコード
#include <stdio.h>

void func1 (int i)
{
  printf(" hello world %i\n", i);
}
```

```
// use.c: 通常の共有ライブラリ利用
#include <stdio.h>

void func1 (int i);

main()
{
  func1(1);
}
```

```
// dlopen.c: dlopen でのバイナリ利用
#include <dlfcn.h>
static int (*myfunc1)(int i) = NULL;
int main()
{
  void* h=dlopen("./liba.so", RTLD_NOW);

  myfunc1=dlsym(h, "func1");

  myfunc1(10);

  return 0;
}
```

これらをコンパイル、リンクしてみて、動作を確認してみました。どうやら Linux とあまりかわらないようです。

```
+ gcc -shared a.c -o liba.so
+ gcc use.c ./liba.so -o use
+ ./use
hello world 1
+ gcc dlopen.c -o dlopen
+ ./dlopen
hello world 10
```

ただし、これだけだとバージョンシンボルを利用した場合に dlsym ではどのバージョンを利用するのかが指定できないような雰囲気がただよっています。というところまで確認したところで今月も力尽きました。また来月続きをながめてみましょう。もしかするとバージョンシンボルの作成とその利用をするかもしれません。

17 Debian Trivia Quiz

上川 純一

ところで、みなさん Debian 関連の話題においついていますか？Debian 関連の話題はメーリングリストをよんでいると追跡できます。ただよんでいるだけでははりあいがないので、理解度のテストをします。特に一人だけでは意味がわからないところもあるかも知れません。みんなで一緒に読んでみましょう。

17.1 第 42 回勉強会

第 42 回勉強会の出題範囲は `debian-devel-announce@lists.deban.org` に投稿された内容と Debian Project News からです。

問題 1. Perl 5.10 のバグでどのような問題が発生した?

□ A Ruby と Python で書かれたアプリケーションやライブラリがインストールできない

□ B インストールされたファイルのパーミッションが 0777 になる

□ C 特定の名前のファイルがインストールできない

問題 2. Debian プロジェクト内のチームに関する調査で判明した「予想外のチーム」でないものは?

□ A 実際に動いているのは 1 人だけというチーム

□ B 誰が作業するか毎回じゃんけんで決めているチーム

□ C お願いやありがとうではなく脅迫で動いているチーム

問題 3. wxwidgets2.8 がアップロードされたが、長いこと wxwidgets2.6 の時代が続いていた。その理由は?

□ A パッケージメンテナが保守的で、2.8 の使用に対して非積極的だった

□ B アップロードしたところでどうせ誰も使ってくれないとパッケージメンテナが思った

□ C パッケージメンテナが多忙で作業時間がとれなかった

問題 4. Frans Pop の辞任によって、新たな執筆者が求められるようになったものとは?

□ A リリースノート

□ B Debian Project Blog

□ C DEB NOTE

問題 5. debian/rules の get-orig-source ターゲットは何を記述するためのものか?

□ A 「オリジ○弁当」で弁当にソースをつけてもらう方法

□ B upstream からネットワーク経由でソースコードを取得して現在のソースコードと置き換える方法

□ C upstream からネットワーク経由で最新の.orig.tar.gz ファイルを取得する方法

問題 6. リリースゴールに関する Peter Eisentraut の意見は?

□ A リリースゴールなんて所詮一部の開発者の楽しみに過ぎない

□ B Debian の機能の実装に関するリリースゴールはリリース後にポリシーへと変えるべきだ

□ C リリースなんて飾りです。偉い人にはそれがわからんのですよ

問題 7. William Pitcock が削除を提案したブートローダパッケージは?

□ A grub

□ B lilo

□ C yaboot

問題 8. Debian weather とはどんなサービスか?
☐ A 特定アーキテクチャのアーカイブの状態を要約して表示する
☐ B Debian 関連ホストが置かれている世界各地の天気を表示する
☐ C メーリングリストの流量から世界各地の天気を推測して表示する

問題 9. Debian 15 周年はいつか?
☐ A 次回の東京エリア Debian 勉強会開催予定日である 8 月 16 日
☐ B 本日 7 月 19 日
☐ C 泣く子も黙る 7 月 9 日

問題 10. Debian のメニュー (.menu ファイル) とデスクトップ環境のメニュー (.desktop ファイル) に関する議論はどのような結論に落ち着いたか?
☐ A freedesktop.org の.desktop ファイルを Debian に合うよう拡張して使っていこう
☐ B freedesktop.org の.desktop ファイルには不便な点があるので、働きかけて修正してもらおう
☐ C freedesktop.org の.desktop ファイルは使えないので Debian の.menu ファイルを使わせよう

問題 11. 6 月末に初めて誕生した Debian 開発者同士の夫婦とは?
☐ A Junichi Uekawa と Kenshi Muto
☐ B Meike Reichle と Alexander Schmehl
☐ C Debra Murdock と Ian Murdock

問題 12. 結婚した二人について述べた以下の事項のうち、正しいものは?
☐ A 最初の贈り物: DebConf5 の土産
☐ B 秘密の愛の交換手段: wiki.debian.org
☐ C 婚約の公式発表手段: lists.debian.org

問題 13. 今年も Debconf が開催されました。どこで開催されたでしょうか。
☐ A 中国
☐ B アルゼンチン
☐ C スペイン

問題 14. ギブアップ宣言をした Debian サブプロジェクト/チームは何でしょうか
☐ A The Debian Live project
☐ B The Debian EEEPC team
☐ C The Debian Jr. project

問題 15. Lenny frozen が宣言されたのはいつでしょうか?
☐ A 2008/07/26
☐ B 2008/07/27
☐ C 2008/07/28

問題 16. Andreas Schuldei が立ち上げた新しいチームは何でしょうか。
☐ A Debian マーケティング チーム
☐ B Debian Dream チーム
☐ C Debian Chrome チーム

問題 17. Debian GNU/Linux 4.0 のアップデート版が出ましたが、何と呼ばれているでしょうか。
☐ A etch and lenny
☐ B etch with you
☐ C etch and a half

問題 18. リリースに向けての作業が佳境に入っています。このような作業の中、lenny の次のバージョンとなるリリースのコードネームが決まりました。何でしょうか。
☐ A 3つ目エイリアン squeeze
☐ B 言葉遊びのオモチャ spell
☐ C 重量挙げ選手のアクションフィギュア rocky

17.2 第 44 回勉強会

第 44 回勉強会の出題範囲は `debian-devel-announce@lists.deban.org` への投稿内容と Debian Project News からです。

17.3 第 46 回勉強会

第 46 回勉強会の出題範囲は `debian-devel-announce@lists.deban.org` への投稿内容と Debian Project News からです。

問題 19. BTS　500000 番はどんなバグ報告だったでしょうか

□ A SPAM だった

□ B cdbs に defoma 対応させるためのパッチ

□ C Lenny リリースが遅れているというバグ報告

問題 20. 10 月 25 日から 30 日まで行われた BTS の景品は何でしょう？

□ A DPL からのキス

□ B 次回の Debconf 無料チケット

□ C おいしいクッキー

問題 21. screenshots.debian.net は何をまとめたサイトでしょうか？

□ A デスクトップのスクリーンショット

□ B ソフトウェアのスクリーンショット

□ C Debian Developer のプライベート写真集

問題 22. Joerg Jaspert がプロポーザルを出した Debian membership とは？

□ A 「Debian 開発者」の現在の定義を変えるための提案

□ B Debian 開発者の家族に関する提案

□ C Debian から fork したディストリに関する提案

問題 23. Debian Installer team から出たアナウンスは何でしょう？

□ A ごっめーん！インストーラー遅れちゃった。

□ B Debian Installer lenny RC1 出たよ

□ C RC1 は飛ばして リリースする予定です。

18 Debian Trivia Quiz 問題回答

上川 純一

Debian Trivia Quiz の問題回答です。あなたは何問わかりましたか？

1. B
2. B
3. A
4. A
5. C
6. B
7. B
8. A
9. A
10. A
11. B
12. A
13. B
14. C
15. B
16. A
17. C
18. A
19. B
20. C
21. B
22. A
23. A

19 索引

2008 年度計画, 2

Active DVI, 58
avahi, 58

bind, 86

cdbs, 3, 78
cdn.debian.net, 85
cdn.debian.or.jp, 85
coLinux, 64
Content Delivery Network, 85
cowdancer, 94

debconf8, 33, 56
debhelper, 3, 17, 78
Debian Conference 2008, 56
debian-onsen, 35
dh-kpatches, 17
dh-make, 22
dns balance, 86
dpatch, 3

eeePC, 95
emacs, 58

gpg, 51

Hurd, 31

ITP, 51

kFreeBSD, 94
kmod, 23
kpatch, 17

latex, 58
Linux Kernel, 17
Linux Kernel Module, 23

make-kpkg, 29
module-assistant, 29

Nexenta, 94, 95

Package maintainer, 51
pbuilder, 52
platex, 58
po4a, 40
pohmelfs, 22

quilt, 3

RFP, 51
rsync, 87

vloopback, 25

whizzytex, 58

『あんどきゅめんてっど でびあん』について

本書は、東京および関西周辺で毎月行なわれている『東京エリア Debian 勉強会』および『関西エリア Debian 勉強会』で使用された資料・小ネタ・必殺技などを一冊にまとめたものです。収録範囲は東京エリアは勉強会第 41 回から第 46 回、関西エリアは第 14 回から第 19 回まで。内容は無保証、つっこみなどがあれば勉強会にて。

あんどきゅめんてっど でびあん 2008 年冬号

2008 年 12 月 29 日　　初版第 1 刷発行
東京エリア Debian 勉強会/関西エリア Debian 勉強会　（編集・印刷・発行）