東京エリア デビアン 勉強会

Debian勉強会幹事 上川純一

2008 年 6 月 21 日

1 Introduction

岩松 信洋

今月の Debian 勉強会へようこそ。これから Debian の世界にあしを踏み入れるという方も、すでにどっぷりとつかっているという方も、月に一回 Debian について語りませんか？

Debian 勉強会の目的は下記です。

- Debian Developer (開発者) の育成。
- 日本語での「開発に関する情報」を整理してまとめ、アップデートする。
- 場 の提供。
  - 普段ばらばらな場所にいる人々が face-to-face で出会える場を提供する。
  - Debian のためになることを語る場を提供する。
  - Debian について語る場を提供する。

Debian の勉強会ということで究極的には参加者全員が Debian Package をがりがりと作るスーパーハッカーになった姿を妄想しています。情報の共有・活用を通して Debian の今後の能動的な展開への土台として、「場」としての空間を提供するのが目的です。

以上を目的とした、2008 年アジェンダです：

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 (David Smith)
4. バイナリ一つのパッケージを作成してみる (吉田@板橋)
   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)(岩松 信洋さん)
5. バイナリの分けたパッケージの作成。(前田さん)
   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)(小林儀匡さん)
   man の書き方 (roff or docbook)(でんさん)
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

デビアン勉強会

目次

2 事前課題

岩松 信洋

今回の事前課題は以下です。

1.「debhelper に追加してほしい機能/あったらいい機能」
2.「 Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと」

この課題に対して提出いただいた内容は以下です。

2.1 吉田@板橋

お題：Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと
1. ドライバーディスク機能
redhat 系ディストリービューションにあるドライバーディスク機能は欲しいですね。特に DISK 関連のドライバは無いとお手上げなので。たびたび武藤さんの手 (スペシャルカーネル入り iso 作成) を煩わせないためには、あるといいかなと思います。＃どっちみちドライバディスク作成で手が煩わされるかな（笑）インストール後は、あまり吊し (Debian 純正) のカーネルを使わないので困らないのですが。次善は lenny 等を入れて強制ダウングレードするのが手かな。
2.X の自動起動停止
runlevel で制御したいとまでは言いませんが... 基本は X なしで速く、マウスなしでログイン必要なら X を上げるというサーバっぽい運用が行いにくいです。

2.2 前田 耕平

「Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと」
Linux じゃなくて、AIX なら楽だなぁと思う機能はあります。

- ＬＶＭ＆ファイルシステムの拡張が AIX くらい楽にできれば良いなぁと思うことしばしば。
- システムバックアップの取得方法。mksysb コマンドのように楽に取れれば良いですね。
- エラーログの機能。errpt コマンドでエラー情報を整形して表示できるのも良いです。

スキルレベルを標準化して運用要員を揃えるのなら、ＡＩＸは良いですね。自宅では使いたくないですけど。w

2.3 あけど

「 Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと」
ずばり、「Hinemos」です。http://www.hinemos.info/ Redhat で動いてるくらいだから Debian で動かすのは難しくなさそうですが、apt に慣れてしまうと deb パッケージであれば…と思ってしまいます。

2.4 山本 浩之

「Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと」

TOMOYO Linux も公式に入ったことですし、AppArmor も早く公式に入って、セキュリティ・パッケージを強化すると嬉しい人が多そう。

2.5 藤沢 理聡

「Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと」

これまでサーバにしか使ってこなかったので、機能面の不足はあんまり感じたことありません。機能ではありませんが、不足を最も感じていたのは認識してもらえないハードが RedHat 系より多いんじゃないか、ということです。実際のところはどうかわかりませんが、「RedHat では認識したんだけど……」と思うことは少なからずありました。とはいっても、2 年以上前のことなので、既に改善されているのかもしれませんが。

2.6 小林 儀匡

「debhelper に追加してほしい機能/あったらいい機能」

Bug#45614 として提案されている dh_alternatives が欲しいです。alternatives を使用するパッケージを作成する場合、ほとんどテンプレート化されている postinst と prerm と自分で書く必要がありますが、メンテナスクリプトは極力メンテナに書かせないほうがよい気がするので……。

2.7 日比野 啓

「debhelper に追加してほしい機能/あったらいい機能」

perl module のパッケージを作ることが多いので、perl のプログラムでも、バイナリにおける dh_shlibdeps みたいにライブラリの依存関係の検知を支援してくれるような機能が欲しいです。

Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと

Debian というよりは linux の話かもしれませんが、AIX の errpt のような厳密なチェック機能はすばらしいと思います。あとは、FreeBSD の ktrace がちょっとうらやましいと思ったことがあります。だいぶ昔のなのでうろおぼえなんですが、kernel の中まで trace してくれたような。

2.8 岩松 信洋

「Debian を使っていて他の distro にあるこの機能／パッケージがあれば…と思ったこと」

rpm にはドキュメントはインストールしないというオプションがあるのですが、Debian にはないのでインストール時に選択できるようにしてほしいです。あと、他のディストリにはサポートしているのに Debian がサポート対象になっていないソフトウェアとかけっこうあって（サイボウズさんとか）、それだけで Debian が使えないというのが悲しいです。

3 Debian Trivia Quiz

岩松 信洋

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。

debian-devel-announce に流れた内容と Debian Project News からです。

3.1 debian-devel-announce

`debian-devel-announce@lists.deban.org` への投稿内容からです。

問題 1. Debian.ch（スイス連邦の Debian ユーザグループ）の体制が変わりました。どうなったでしょう。

☐ A 前会計が辞任したので、Martin F. Krafft が会計になりました。

☐ B 会計を全てスイス銀行の管理下に置かれます。

☐ C スイス連邦の Pascal Couchepin 大統領が Debian ユーザになりました。

問題 2. 2008 年 6 月 5 日に新しい Debian Policy がリリースされました。バージョン番号はいくつでしょうか。

☐ A 3.1415

☐ B 3.8.0.0

☐ C 4.0.0.0

問題 3. Debian installer lenny beta 2 がリリースされたのはいつでしょうか。

☐ A 実は上川さんの誕生日、5 月 22 日

☐ B 2008 年 6 月 1 日

☐ C 2008 年 6 月 10 日

3.2 Debian Project News - May 26th, 2008

`http://www.debian.org/News/weekly/2008/03/` への投稿内容からです

問題 4. Perl の新しいバージョンの移行が完了しました。lenny に入る Perl のバージョンは？

☐ A 5.10

☐ B 5.20

☐ C Perl-next-1.0

問題 5. DPL から Bits from the DPL が出ました。あれ？今の DPLって？

☐ A Sam Hocevar

☐ B Martin Michlmayr

☐ C Steve McIntyre

問題 6. OpenSSH の DSA-1571 で影響を受けたパッケージ数はいくつでしょうか。

☐ A 20

☐ B 200

☐ C 2000

問題 7. Debian Game team がデータの大きいサイズのパッケージについて提案したことは？

☐ A データは P2P で配信しようぜ！

☐ B データはデータでアーカイブを分けようぜ！

☐ C データは与えられるものではなく、与えるものだ！削除しましょう。

3.3 Debian Project News - June 9th, 2008

`http://www.debian.org/News/weekly/2008/04/`
への投稿内容からです

問題 8. ftp-master team が編成されました。FTP Master は Joerg Jaspert と誰になったでしょう？

☐ A Ryan Murray

☐ B Junichi Uekawa

☐ C Kenshi Muto

問題 9. debconf の翻訳が完了一番乗りした国は？

☐ A フランス

☐ B 日本

☐ C 実は翻訳状況の表示不具合で、まだ一番乗りはいません

問題 10. debconf の翻訳が遅れていて、CFH を出した国は？

☐ A ドイツ

☐ B 日本

☐ C 実は翻訳状況の表示ミスで、遅れていませんでした

問題 11. Bastian Venthur と Noel Koethe レポートを出したイベントは？

☐ A BSDCan 2008

☐ B LinuxWorld Expo/Tokyo 2008

☐ C LinuxTag 2008

4 最近の Debian 関連のミーティング報告

岩松 信洋

4.1 東京エリア Debian 勉強会 40 回目報告

東京エリア Debian 勉強会報告。 5 月の第 40 回東京エリア Debian 勉強会を実施しました。今回の参加者は あけどさん、後藤さん、やまねさん、前田さん、沖中さん、 岩松さん、野村健太郎さん、堀内寛之さん、 小林儀匡さん、吉田@板橋さん、山本 浩之さん、 山本琢さん、福永さん、でん@相模原さん、奥野さん、 藤沢理聡さん、日比野 啓さん、荒木さん、 吉藤さん、上川の 20 人でした。

まず最初に最近のミーティングの報告を行いました。台湾で行われた eeePC Open Source Developer's Conference の報告を簡単に行いました。

今回のクイズは小林さんが出題しました。 3 問でだいたい全員不正解になるという勢いでした。事前課題を紹介しました。 いろいろな話題が出ました。 ゴールデンウィークにさまざまなハックが行われていたようですね。

2008 年のテーマは DEB パッケージの開発・管理に関連した内容ですが、 3 回目のテーマとして前田さんが複数パッケージを生成するソースパッケージについて紹介しました。 debhelper の魔窟をすこし垣間見たようです。OpenSSL の脆弱性の問題について議論しました。 いろいろな視点から検討し、問題の深刻さと、 どう公報するのがよいかについて議論しました。 また、SSH のキーの仕組みには revoke する方法が現在はなくて いちいち自分でいれかえる必要があるため大変だ、 ssh の鍵が脆弱ということで、 Debian で管理している範囲に影響が限定されないため、 パッチの配布だけでは解決しないところが困難だねという話題が出ました。

最近の kFreeBSD, nexenta, Hurd, SuperH について紹介して終了しました。

今回も宴会は駒忠にて開催しました。おばちゃんにまたきてね、と言われました。

5 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

5.1 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。

5.2 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。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。

[*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。

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。

5.3 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。

5.3.1 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。

5.3.2 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。

5.3.3 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。

5.3.4 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。

5.3.5 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。

5.3.6 .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。

5.3.7 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。

5.4 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。

5.5 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
```

5.6 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。

5.7 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。

5.7.1 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。

[*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。

- dh_testroot
- dh_testdir

5.7.2 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

5.7.3 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

5.7.4 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

5.7.5 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

5.7.6 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`

5.7.7 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`

5.8 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。

5.9 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。

5.9.1 `/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。

5.9.2 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。

5.9.3 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
```

5.9.4 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。

5.10 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。

5.11 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。

5.11.1 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
```

5.11.2 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。

5.11.3 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。

6 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。しかし仮想ディスク環境でなら、気軽で、簡単に使えます。今日は Debian GNU/Hurd の VMware 上での利用を紹介をしてみます。

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 / としときます。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 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!

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 Debian GNU/Linux SuperH port 岩松 信洋

今月の SuperH port は作者取材のためお休みです。

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 Debian GNU/Hurd 山本 浩之

今月の Debian GNU/Hurd は作者の都合によりお休み　　です。

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 が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 上では動かさなさそうですね。
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。

Debian 勉強会資料

2008 年 6 月 21 日　　初版第 1 刷発行
東京エリア Debian 勉強会 （編集・印刷・発行）