Deb専

日本唯一のDebian専門誌

2011 年 12 月 31 日 初版発行

東京エリア Debian 勉強会

関西 Debian 勉強会

2011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Debian勉強会　著

デビアン勉強会

目次

1 Introduction

上川 純一, 山下 尊也

1.1 東京エリア Debian 勉強会

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。

1.2 関西 Debian 勉強会

関西 Debian 勉強会は Debian GNU/Linux のさまざまなトピック (新しいパッケージ、 Debian 特有の機能の仕組、Debian 界隈で起こった出来事、などなど) について話し合う会です。

目的として次の三つを考えています。

- ML や掲示板ではなく、直接顔を合わせる事での情報交換の促進
- 定期的に集まれる場所
- 資料の作成

それでは、楽しい一時をお楽しみ下さい。

2 Debian とはなにか?

岩松信洋

この章は筑波大学さんで Debian 勉強会を開催したときの資料です。

Debian 勉強会を筑波大学さんで行うにあたり、たぶん Linux や Ubuntu、 Fedora という言葉は知っているけど、Debian は知らないって方がいるでしょう。簡単に Debian とは何なのかを簡単に説明します。

2.1 Debian とは?

Debian Project の略称、または Debian OS そのものを指す場合があります。フリーかつオープンな OS を作る完全ボランティアベースのプロジェクトです。歴史が長く保守的な Linux ディストリビューションの一つです。

公式開発者は約 1000 名。非公式な開発者やパッケージメンテナ、翻訳者などを入れると 5000 名以上になります。世界のいたるところに開発者がいて、日本では約 30 名ほど公式開発者が活動しています。また、日本の開発者が集まって活動している Debian JP Project もあり、日本での Debian 環境のサポート、開発者育成、ユーザサポートなどを行なっています。

オープンソース・ライセンスの要件の定義(The Open Source Definition(OSD)) は Debian の Debian フリーソフトウェアガイドラインをベースとしたものです。

2.2 特徴

2.2.1 フリーなソフトウェアで構成されている

Debian プロジェクトはフリーソフトウェアを強く支持しています。ソフトウェアには、いくつもの違ったライセンスが使われるので、 Debian フリーソフトウェアガイドライン (DFSG) を作って、「 何をもってフリーソフトウェアと言えるのか」の妥当な定義をしています。

- 何台のマシンにもソフトウェアをインストールしても良い。
- 何人の人がソフトウェアを同時に使用しても良い。
- 何個でもソフトウェアのコピーを作ってもいいし、それを誰にあげても構わない。 (フリーもしくはオープンな再配布)
- ソフトウェアの改変に対する制約が無い。 (特定の通告を変えない事を除く)
- そのソフトウェアを配布や売る事に対する制約が無い。

Debian の「 main」ディストリビューションには、 DFSG に適合したソフトウェアしか 入れる事を許されていません。[*1]

とはいっても、フリーではないアプリケーションを使いたいユーザもいるので、そのようなユーザのために contrib、non-free といったパッケージセクションを作って、できるだけ提供できるようにしています。

[*1] http://www.debian.org/intro/free.ja.html より

2.2.2 オープンな開発をしている

完全ボランティアの団体なので、特定の企業の力によって Debian の方針が変更されたりすることがありません。バグや議論経過なども全て公開されており、プロジェクトの方針等は Debian 公式開発者の選挙によって決まります。

2.2.3 バイナリベースのディストリビューション

ディストリビューションにはバイナリベースとソースベースの 2 種類があります。前者は既にコンパイルされたパッケージを提供するディストリビューションで、 Debian や Redhat、 Ubuntu などがあります。既にコンパイルされているので直ぐに使うことができますが、一定のルールによって最適化されているため、使っている CPU 向けに最適化されているとは限りません。しかし、同じアーキテクチャならどの環境でも同じ問題が再現する可能性があり、問題の共有が容易になります。自分で使うソフトウェアは自分でコンパイルするというディストリビューションで、代表的なものとして Gentoo があります。パッケージにはソースコードはなく、コンパイルに必要な簡単なスクリプトがパッケージに同梱されています。このスクリプトで定義されている場所からソースコードをダウンロードし、コンパイルします。これは自分に合わせたソフトウェアに最適化できるという利点があります。そのかわりソフトウェアを使うには時間がかかり、問題があった場合でも他の環境では再現しにくいというデメリットもあります。

2.2.4 豊富なパッケージ数

Debian は多くのパッケージを提供しており、現在約 3 万パッケージのパッケージが利用可能です。他のディストリビューションは、 Gentoo が 15000 パッケージ、 Ubuntu だと 10000 パッケージ[*2]ほどあります。 Debian の変態的なところは、各アーキテクチャで同じバージョンのバイナリを提供している事です。リリース対象になっているアーキテクチャでパッケージが動作しない場合、ポーティングを行い、開発元に取り込むように提案します。

2.2.5 ポリシーに基づいたパッケージ

Debian で提供されているパッケージは Debian Pocily というパッケージングポリシーに基づいて作成されています。このポリシーは厳格に決められており、違反しているパッケージは Debian にインストールされません。

2.2.6 強力なパッケージングシステム

Debian では パッケージングシステムに dpkg というアプリケーションを使っており、 deb という拡張子がついたパッケージファイルをインストール、アンインストールします。パッケージの依存関係管理がしっかりされており、 depends（ 依存）、 recommends（ 推奨）、 suggests（ 提案） などの項目によってコントロールしています。パッケージマネージャの APT (Advanced Package Tool) によって、インストールしたいパッケージに依存しているパッケージがインストールされます。アンインストールも同様です。

2.2.7 アップデートが容易

Debian は約 2 年毎に安定版がリリースされます。 Debian は前回のバージョンからのアップデートをサポートしています。例えば、 2007 年にリリースされた 4.0 から、最新版の 6.0 にアップデートするには、 4.0、 5.0、 6.0 と順にアップデートすることによって可能です。

2.2.8 豊富なサポート CPU アーキテクチャ

現時点で正式サポート CPU アーキテクチャは 11、次期リリースに向けてサポート準備中が 10 あります。サーバから PC、組み込み CPU までサポートしています。新しい CPU アーキテクチャをサポートするためのインフラもあるので、何がサポートしたい CPU がある人、 debian-ports プロジェクト（ http://www.debian-ports.org）に相談するとインフラを提供してくれるかもしれません。

[*2] main と Universe がありますが、基本的に Ubuntu 側のサポートありなのは main のみ。

現在サポートしているアーキテクチャ。

- amd64
- armel
- hurd-i386
- i386
- ia64
- kfreebsd-amd64
- kfreebsd-i386
- mips
- mipsel
- powerpc
- s390
- sparc

サポート予定のアーキテクチャ

- alpha
- armhf
- avr32
- hppa
- m68k
- powerpcspe
- s390x
- sh4
- sparc64

2.2.9 Linux 以外のカーネルもサポートする

Linux をカーネルとした OS、 Debian GNU/Linux だけではなく、 FreeBSD のカーネルを使った OS Debian GNU/kFreeBSD も提供しています。 Debian 開発者の中には GNU Hurd, Minix, NetBSD カーネルをベースにした Debian を開発している人もいます。

2.2.10 他の OSS プロジェクトと関連が強い

Debian 開発者と各 OSS 開発者が兼務していることが多く、他の OSS プロジェクトと結び付きが強いです。パッケージメンテナ＝開発元の開発者ということが多いのが特徴です。自分の作ったソフトウェアを Debian に入れたい人が多いようです。また、大抵の Debian 開発者は複数のプロジェクトに顔を出しているので、更にプロジェクト間の結びつきが強いです。

2.2.11 派生しているディストリビューションの多さ

いままで説明した特徴によって、 Debian から派生したディストリビューションが多くあります。有名なところでは、Ubuntu や Knoppix、 Vyatta（ VPN/ネットワークファイアウォール）などがあります。現時点で 129 以上の派生ディストリビューション[*3]があり、 Debian の live CD システムを使った小さいディストリビューションを入れるともっと多くなります。また派生として分散させているだけでなく、派生したディストリビューションの成果を本家である Debian に取り組む仕組みもあります。ちなみに 2 番目に多いのは Fedora ベースの 63 です。

2.3 まとめ

- フリーである。
- オープンな開発プロジェクトである。
- 世界規模のボランティアベースのプロジェクトである。
- バイナリベースのディストリビューションで、サポートしているパッケージ数が多い。
- サポートしているアーキテクチャが多い。
- Linux カーネルだけをサポートしていない。
- 派生しているディストリビューションが多い。

[*3] http://distrowatch.com/dwres.php?resource=independence 参照

2.4 んで、どういう風に使えばいいの?

個人的な見解ですと、

- 開発に使いたいなら、 Debian か Gentoo。
  アップストリームに近い位置にいるためです。パッチなどがディストリビューション開発者経由で取り込まれやすい。
- デスクトップやノート PC で使いたいなら Ubuntu。いろいろデスクトップとか弄りたいなら、 Debian か Gentoo。
  2ch とかニコニコ動画みる程度なら Ubuntu で十分だと思います。もちろん Debian でも問題ありません。プログラムを最適化したいとか、「 Gnome とかイラネ! 他のデスクトップ環境が欲しい」という人なら、 Debian か Gentoo をお薦めします。
- サーバで使いたいなら、 Debian。
  無駄なものがインストールされてないから。

です。

ぜひ Debian を使って、フィードバックをください。そして開発に興味がある人は開発に参加してみてください。手取り足取り教えます。みんなで Debian を良いものにしていきましょう。

3 翻訳で Debian に貢献しよう

八津尾 雄介

3.1 はじめに

Debian を利用する以上, 英語との付き合いは避けて通れない問題だと思います. 英語の文章を読むのが苦にならない人から, エラーメッセージさえ読む努力を放棄する人まで, 様々だとは思いますが, もっと英語ができればと思う事が誰でも少なからずあると思います.

私は英語と付き合う第一歩として, 翻訳をおすすめします. 翻訳というと特殊技術だとか自分には無理と身構えてしまう人も多いかもしれませんが, それほど敷居の高い作業ではありません.

辞書と基本文法の知識さえあれば誰でもできる作業ですし, どうしても意味がとれない箇所はメーリングリストなどへ投げれば誰かが答えてくるでしょう. ついでに Debian の知識も身につくので, Debian Maintainer や Debian Developer を目指している人にとってうってつけの自習教材ではないでしょうか?

オープンソースコミュニティ全体に言えることだと思いますが, 翻訳者の数は圧倒的に少ないのが現状です. その主たる原因を私なりに分析してみました.

- 読める人は訳さない
- 読めない人も訳さない
- 時間も手間もかかる地道な作業
- 貢献に対する見返りがあまりない (なかった) [*4]

DDP[*5]にはまだ翻訳されていない文章がたくさんあります. 翻訳をしながら Debian について学び, 貢献し, そして英語力の向上に役立ててみませんか?

3.2 翻訳メモリとは

翻訳とは一般的に, 時間も手間もかかる地道な作業なわけですが, それをある程度軽減してくれるのが翻訳メモリツールです. 翻訳メモリとは, 原文と翻訳文のペアをデータベース化したもので, 翻訳メモリツールは翻訳メモリから一致率の高い文章をサジェストする翻訳支援ソフトです. 翻訳メモリツールを指して翻訳メモリということもあります.

翻訳メモリはいわば実用文例集のようなもので, 機械翻訳とは根本的に違います. 注意したいのは, 翻訳メモリを作るのは訳者自身だということです. 翻訳メモリツールが参照するのはあくまで, あなたが (もしくはメモリの提供元が) 過去に訳した文章であり, それについての正確さは一切保証されていません.

[*4] パッケージ作業以外での Debian への貢献を認める決議がなされました
[*5] Abbr. Debian Documentation Project: http://www.debian.org/doc/ddp

3.2.1 翻訳メモリを使うと何が嬉しいのか

では, 翻訳メモリを利用することで得られるメリットとは何でしょうか.

- 膨大な訳文を蓄積し使い回す事により作業効率が一気に高まる (作業の効率化)
- 複数人で翻訳作業を行う際の文体や訳語の微妙な違いを少なくする事ができる (一定品質の保持)
- PO 形式ではない場合でも原文の変更に追従しやすくなる (保守性の向上)

職業翻訳では翻訳メモリが広く利用されています. 翻訳家は業界標準の翻訳メモリである Trados などの利用率が高いようです. 一方, Debian で利用可能な翻訳支援ツールは poedit や gtranslator, OmegaT. web ベースであれば Google Translator Toolkit などがあります. どれも翻訳メモリを使用するツールです.

3.3 OmegaT とは

OmegaT とは先述の通りオープンソースで利用可能な翻訳メモリで, Java で開発されておりクロスプラットフォームです. 翻訳メモリには LISA[*6]が標準化している TMX[*7] というオープンな XML を採用しており, Trados や SDLX をはじめとした他ツール間で翻訳メモリを相互運用できます.

OmegaT では同梱版の Java を使うように推奨されていますが Debian でもパッケージを提供しており, apt からインストールすることも可能です.

■OmegaT の (一般的に言われる) 良い点

- オープンソース
- コミュニティが活発
- 業界標準の TMX を使っている
- Windows 的 (あるいは Gnome 的) な操作性

■OmegaT の (私から見て) ダメな点

- 起動が遅い (=Java)
- Look and Feel が残念 (=Java)
- ショートカットキーとニモニックが致命的にまずい
- hjkl で移動できない ← 重要

という事で Java で開発されているという点と, vim じゃないという点以外は特に不満はありません. しかしながら, マウス操作を強制されるというのはあまり気持の良いものではありませんね.

3.4 OmegaT の使い方

ここでは, apt から Debian パッケージをインストールしたものとして話を進めていきたいと思います. Debian Squeeze での現在のバージョンは 1.8.1 ですが皆さんは Wheezy あるいは Sid を使っているはずですので (?) 2.3.0 前提で進めたいと思います. 基本的な使い方についてはお手軽スタートガイドを読めばわかりますのでここでは割愛します.

3.4.1 ファイルフォーマットについて

OmegaT は次のフォーマットをサポートしています.

- OpenDocument や OpenOffice.org

[*6] abbr. Localization Industry Standarts Assosiation = ローカライゼーション産業の標準化団体
[*7] abbr. Translation Memory eXchange

- プレーンテキスト
- .po ファイル
- XHTML, HTML
- Microsoft Open XML
- 字幕ファイル (SRT)
- Android リソース
- LaTeX
- 他多数

■サポート外のファイルを訳すには DebianDoc-SGML の利用は廃止にむかっているもののまだまだ sgml のドキュメントが存在します. sgml は OmegaT によってサポートされていない形式です.[*8]OmegaT はサポートしない形式のファイルを翻訳対象のファイルに加えようとしても無視します. では, OmegaT でサポートされないフォーマットのファイルを訳すにはどうすれば良いでしょうか?

とりあえず訳したいという場合にはファイルの拡張子を OmegaT でサポートされているファイルの拡張子にしてしまえば訳すことができます. ただし, タグ付きの文書の場合はタグの扱いに注意をしなければなりません. とりあえず *.txt にしておくというのが正解のような気がします.

Debian 的な方法としては Gettext PO 形式に変換して翻訳するという方法がありますが, OmegaT 的にやろうと思うのであれば, ファイルフィルターを利用して近いフォーマットとして認識させる方法が良いでしょう. 例えば sgml を xhtml として読み込ませたい場合, ”設定” から ”ファイルフィルター” を選択し, XHTML を選択した状態で ”編集” をクリックします. それから ”追加” をクリックして ”*.sgml” を ”原文ファイルの構成名” へ追加します. こうする事で sgml ファイルが翻訳対象のファイルとして追加可能になります.

残念ながら独自のファイル定義を追加できるわけでは無いので, 今のところは拡張子を変更して対応する場合と大きな違いはありません. ただ, 例にあるような sgml などのタグ付きのファイルを編集する場合, タグの挿入などの便利な機能を使えるようなるので txt で扱うよりも少し楽になるはずです.

3.4.2 分節化規則を変更する

分節化された文章を見てみると, 中途半端な箇所で切られてしまっているような場合がままあります. 特にタグ付きの文章をプレーンテキストとして読み込ませると思わぬ所で分節化されてしまいます. そのような自体に対処したり, ユーザーの好みに柔軟に対応する為に, 分節化の規則をカスタマイズする事が可能です.

分節化の設定は ”設定” の ”分節化規則” から行えます. 分節化規則の定義は Java でサポートされている正規表現を使用します.

この分節化規則は慎重に取り扱うべきでしょう. 途中で規則を変更してしまえばメモリの一致率を下げる原因となるからです. 分節化規則は良く考えられて作られているので, 最初はデフォルトのまま使用し, もし不満があれば何度が試験的な短めのドキュメントでテストしながら徐々に自分の好みに合わせていけば良いでしょう. いきなり長文の翻訳に適用してしまうと, 思わぬ不都合が起きた場合の対処が非常に面倒です.

3.4.3 用語集を作成する

用語集: グロッサリ は プロジェクトフォルダ内の glossary に置きます. (デフォルト設定) 翻訳中にでくわした用語をグロッサリへ登録しておけば訳語に統一感を持たせる事ができます. 例えば ”upstream” を訳す際に ”開発元” とすべきか, ”上流” とすべきか ”アップストリーム” とすべきかといったような事や ”コンピュータ” と表記するか ”コンピューター” と長音を省略せず表記するかといったような曖昧になりやすい事は下読みの段階でピックアップしグロッサリにしておくと良いでしょう.[*9]言うまでもなく, 複数人で訳す場合はこれを共有すべきですね. Debian を デビアン と表記しないなど”訳さない” 単語を登録しておくのも有効です.

[*8] Java で利用可能なオープンのパーサが無いという理由らしい

[*9] 一括で置換する手法を使う人にとっては不要ですね…でも将来の為に作っておくと良いですよ

グロッサリは OmegaT で直接編集はできません.[*10]作成/編集/メンテにはテキストエディタを使いましょう.

Debian JP では対訳表というのを作っていて (?)[*11]OmegaT で利用できる形式にしてあるものも一応あります.[*12] とりあえずはこれを用語集として放り込んでおいても良いでしょう.

OmegaT のヘルプには何故か長々と OpenOffice を使ったグロッサリの作成方法が書かれています. グロッサリの元データが表計算ソフトで作られていたりワープロソフトで作られていたりしないのであれば, テキストエディタで作った方が楽だと思います.

グロッサリのフォーマットは

```
翻訳対象の言語 [TAB] 日本語 [TAB] 説明や注記
```

です. グロッサリのファイルは プロジェクトで使用しているものと同じエンコードで保存し, ファイル名は "*.tab" とします. 私は面倒なので全て UTF-8 にし, グロッサリは "*.utf8" にしています.

実際にグロッサリを作成してみます. 例えばこんなファイルを作ってみましょう.

```
Debian (tab) Debian
maintainer (tab) メンテナ
computer (tab) コンピュータ (tab) 長音省略
upstream (tab) 開発元 (tab) アップストリーム 上流 は避ける
Debian developer (tab) Debian 開発者
```

上記の例でわかるように, ソートされている必要は無いようです. これを OmegaT のプロジェクトフォルダ内の glossary に配置します. 翻訳中の分節が含む完全一致した単語を全て, 用語集ペインに表示します. グロッサリは完全一致している必要があり, 活用形はピックアップできません. 例えば

Debian [tab] Debian

があった場合, Debian's は ピックアップできません. ケースセンシティブではないので debian は表示されます. このあたりの挙動を理解しながら良く使われる活用 - 例えば上記の "Debian's" など - もある程度網羅すると良さそうです.

グロッサリに辞書ファイルを登録する事も可能ですが, 分節中の単語全てをリストアップし, まともな辞書であれば用語集ペインが溢れ返ってしまう上, 上記の通り活用まではカバーできない為あまりおすすめできません.

グロッサリは, 例えば "debian.utf8" ".utf8" など, 大雑把でもジャンル分けしておいた方が再利用しやすくなります.

3.4.4 辞書について

実は私は OmegaT の辞書を使った事がありません. 普段は wine 上の PDIC で 英辞郎を利用するか, 手元の電子辞書を使います. OmegaT では StarDict 形式の辞書ファイルをサポートしていて, tab 区切りになっている辞書ファイルであれば stardicttools で簡単に変換できます. せっかくなので英辞郎の辞書ファイルを StarDict の形式に変換し使用してみましたが, 私の欲しい感じではありませんでした. 今後に期待です.

3.5 翻訳メモリを活用する

OmegaT では翻訳メモリを複数箇所に保持しています.

■project/omegat フォルダ内

- project_save.tmx

このフォルダ内には tmx ファイルのバックアップが作成されます. 翻訳作業を開始してからの全ての分節が保存されています. プロジェクトとして実際に読み込まれているのがこれです.

■project/ 内

[*10] 顧客からグロッサリを渡されるような事があるので, 訳者が勝手に編集できないようにする配慮だと思われます

[*11] 私の知る限りでは放置中です

[*12] http://www.debian.or.jp/community/translate/ 参照

- *_omegat.tmx
- *_level1.tmx
- *_level2.tmx

target ファイル生成時の source ファイルの内容に対応した翻訳メモリが作成されます. それぞれのファイルフォーマットには微妙な差異があり, 用途によってわかれています. level1 は文書情報のみが含まれています. level2 は OmegaT 特有の情報が tmx タグ として保存されるので level2 の翻訳メモリに対応したアプリケーションでの利用が可能です. omegat は OmegaT 特有のフォーマットなので他のアプリケーションからは利用できません.

■project/tm フォルダ内 過去のプロジェクトからメモリを流用したい時はこのフォルダに配置します. level1, level2 あるいは omegat のいづれかのファイルをいくつでも置いておく事が可能です.

翻訳メモリの内部を覗いでみましょう. フォーマットによって違いはありますが, だいたいこんな風になっています.

```
<tu>
<tuv lang="lang1">
<seg>lang1 の分節</seg>
</tuv>
</tu>
<tu>
<tuv lang="lang2">
<seg>lang2 の分節</seg>
<tuv>
</tu>
```

OmegaT のプロジェクトフォルダでは /source 内へのファイル配置は基本的に自由です. このフォルダ内で "securing-howto" や "maint-guide" のように文書毎のフォルダを作り管理するという方法がおすすめです. こうする事によって, いちいちマージやファイル移動をしなくても用語集や翻訳メモリを共通で使えるようになります.

問題は翻訳メモリのサイズとソースファイルの量ですが, 試しに DDP からチェックアウトしてきたファイルのうち, 不正なファイルだとエラーで弾かれるものを除き, 全てプロジェクトフォルダに放り込んでみました. 私が確認した限りでは問題なく動作していましたが, 読み込みに時間がかかるので同一プロジェクトで扱うファイルは常識的な数に留めておく事が賢明です.

3.5.1 機械翻訳を利用する

機械翻訳はまだまだ使い物になりません. しかし, 短い文章の翻訳精度は向上しつつあり, 場合によっては て に を は を 修正すればそのまま使えるような文章ができあがるような場合もあります. うまく訳せない長目の文章でも, 特定の単語の意味がわからない場合などは有効な手段です. 別段必要な機能だとは思えませんが, あくまでも参考程度に利用するのであれば良いと思います.

3.5.2 翻訳へ参加する

翻訳を始めようと思ったら, まず Debian JP Documentation メーリングリストへ参加しましょう. 詳しくは `http://www.debian.or.jp/community/ml/openml.html` を参照するか, 会場のスタッフへ尋ねてみてください. きっとあなたの世界が広がりますよ.

3.6 まとめ

現在勉強会当日の朝 4 時ですので, そろそろまとめに入らせて頂きます. OmegaT は数ある翻訳メモリの中でもフリーで汎用性が高く, プロフェッショナルユースにも十分利用できるソフトウェアです.

翻訳作業全てを OmegaT で行う事を強制する事は望ましくありませんが, せめて翻訳メモリの使用を強く推奨し, コミュニティの資産としてメモリを蓄積できれば今後の翻訳作業が加速する事は間違いありません.

また, 翻訳者の中にはマウスの操作を嫌う人が相当数いますので [要出典] 是非ショートカットキーのカスタマイズとニーモニックキーを採用して欲しいと思います.

4 Debian JP 定例会議処理系に XSLT を使ってみた

上川純一

4.1 背景

Debian 勉強会の企画会議は IRC を中心として 2006 年に開始し、 Debian JP の定例会議として今も続いています。当初は決定事項などについてテキストファイルでまとめるという形をとっていました。 IRC でより効率よく議論する方法を模索した結果、議論しながら議事録を編集するというスタイルが確立し、それを支援するためのツールを整備しました。

議事録のソースは議長が XML で記述して、議論の最中は非同期に Javascript で内容が更新される HTML ファイルを利用します (世間一般では AJAX 的とでもよぶようです)。

IRC での定例会議の議論の前と後には議事案と議事録をメーリングリストにおくっています。メーリングリストにメールでなげる際には、テキストフォーマットにして送っています。

あと、現在用途がないですが、 LaTeX 経由で PDF 形式での出力などもサポートしています。

現在の実装は歴史的な経緯により XML の処理系は dancer-xml ライブラリと boost を利用した C++ のプログラムになっています。 dancer-xml[4] は 10 年前に若気のいたりで実装した XML 風文書のパーサーです。一部エンティティーまわりなど真面目に実装していない部分があるため、適切な処理がなされていないことがありますが、僕の好みに空白文字処理はチューニングされており快適です。

今回の挑戦は、独自 C++ コードベースを XSLT にのせ替えてみるという挑戦です。

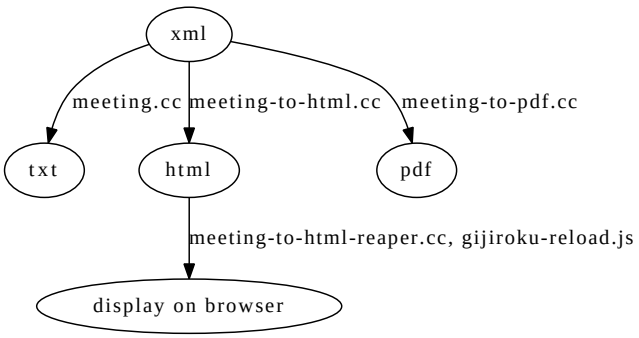

図 1 IRC 会議システムのデータ形式の概要

4.2 XSLT ってどんな言語?

XSLT は XML で記述する XML 処理言語です。

XSLT 規格には 1999 年に策定された XSLT 1.0 と、 2007 年ごろの XSLT 2.0 があります。今回は実装が十分枯れて

いると思われる XSLT 1.0 の処理系を採用しました。 XSLT2.0 の Debian で利用できる実装としては libsaxonb-java があるようですが、今回は調査していません。

プログラムを書くときにすべき文書としては、 XSLT 自体の 1999 年に策定された規格 [2] と、 XSLT の中で記述できる XPATH の規格 [3] を参照するとよいでしょう。

XSLT だけではあまり高度なプログラミングはできないんじゃないかと思われるかもしれませんが、関数型言語として十分な機能を提供できる力はあるようです [1]。

4.3 Debian で利用可能な XSLT 処理系

Debian で幅広く使われていて安定しているとおもわれるのと、簡単に利用できるという理由で処理系として xsltproc を採用しました。

Debian での xsltproc のインストールは簡単

```
$ apt-get install xsltproc
```

コマンドラインで以下のように実行すると標準出力に処理済み XML が出力されます。

```
$ xsltproc [スタイルシート] [処理する XML ファイル]
```

4.4 具体例:HTML

それでは、 HTML 出力の場合を見てみましょう。 meetinglog:html.xsl です。 XML 文書から HTML 文書を生成するにはそれなりに便利な言語です。

前半のコードをそのまま掲載します。これは、 XML ドキュメント全体にマッチするルールを記述しはじめるまでの部分です。 XML 名前空間として、デフォルトを HTML、 xsl を xslt の名前空間に割り当てています。

xml:output で出力形式を HTML と指定することで XML ヘッダが出力されずに便利です。

```
<?xml version="1.0"?>
<!DOCTYPE xsl:stylesheet>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">
  <xsl:output method="html" />
  <xsl:template match="/">
    <html>
      <head>
```

他に XSLT の特徴的なところは、 value-of で値をとってきています。 XPATH の書式で指定していますが、

```
        <h1><xsl:value-of select="meetinglog/head/title"/></h1>
```

は、 XML 文書の以下のようなエレメントに入っている値を抽出します。

```
<meetinglog>
  <head>
    <title>タイトル</title>
  </head>
</meetinglog>
```

議事録の場合のメインループは、各議事に対しての処理です。 xsl の xsl:for-each をつかい、 XML のエレメントノードの数だけループします。 HTML タグはそのまま出力されますが、もし HTML のアトリビュートなどを XSLT で生成したい場合は、 xsl:element を使ってエレメントを生成します。

position() 関数は現在のエレメント番号をくれるのでこういう場合に便利です。

```
          <xsl:for-each select="meetinglog/body">
             <tr>
               <th>
                 <xsl:element name="a">
                   <xsl:attribute name="href">#gian<xsl:value-of select="position()" /></xsl:attribute>
                 </xsl:element>
                 議案<xsl:value-of select="position()" />
               </th>
               <td class="bodytitle">
                 <xsl:value-of select="./title" />
               </td>
....
</xsl:for-each>
```

4.5 具体例:Text 出力

Text 出力の場合もみてみましょう。 meetinglog:txt.xsl テキスト出力をしようとしはじめると若干苦しくなってきます。できないわけではないのですが、空白文字の処理のルールを僕がいまいち理解できていないのと、コードがそのままテンプレートとして出力されるのでインデンテーショ ンが適切にできないのがつらいところです。

xsl:output で出力がテキスト形式であると指定すると XML ヘッダが出力されず便利です。

ヘッダ部分で、毎回 xsl:text で改行などを入力するのが面倒なので、 ENTITY を定義して省略できるようにしています。この記法が正しいのかどうかは不明です。

```
<?xml version="1.0"?>
<!DOCTYPE xsl:stylesheet [
<!ENTITY space  "<xsl:text xmlns:xsl='http://www.w3.org/1999/XSL/Transform'> </xsl:text>">
<!ENTITY indent "<xsl:text xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>  </xsl:text>">
<!ENTITY cr     "<xsl:text xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
</xsl:text>">]>
  <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">
  <xsl:output method="text" />
  <xsl:template match="/">
    <xsl:text>------------------------------------------------------------------------
概要
------------------------------------------------------------------------
```

本文のコアとなる本文の内容ですが、読めたものではないです。悩んだところとしては、文章が空白かどうかチェックするのに string-length(normalize-space()) をつかっていて、それがいまいちただしいのかどうか自信がないところ。

```
    <xsl:for-each select="meetinglog/body">
------------------------------------------------------------------------
[<xsl:value-of select="position()" />.&space;<xsl:value-of select="./title" />]
------------------------------------------------------------------------

目的: &cr;<xsl:value-of select="./aim" />&cr;
&cr;<xsl:if
 test="string-length(normalize-space(./previous))>1"
 >前回までの経緯:&cr;<xsl:value-of select="./previous"
 disable-output-escaping="yes" />&cr;&cr;</xsl:if>
<xsl:if test="string-length(normalize-space(./discussed))>1"
 >議論:&cr;<xsl:value-of select="./discussed"
 disable-output-escaping="yes" />&cr;&cr;&cr;
</xsl:if>
</xsl:for-each>
```

残念ながら C++ で実装していた文字をメールの 70 文字幅くらいにきれいにまとめるというロジックが欠落しています。めんどくさすぎる。

4.6 具体例:LaTeX

LaTeX 出力をみてみましょう。 meetinglog:latex.xsl

ヘッダ部分はどうぜ LaTeX のヘッダなのと何度もでてきているのでメインループだけ。

```
    <xsl:for-each select="meetinglog/body">

      \discussion{<xsl:value-of select="./title" />}{<xsl:value-of
select="./aim" />}{<xsl:value-of
select="translate(./previous,'#&amp;',’--’)"
disable-output-escaping="yes" />}{<xsl:value-of
select="translate(./discussed,'#&amp;',’--’)"
disable-output-escaping="yes" />}
    </xsl:for-each>
```

個人的な感想ですが、自分で書いておきながら後で読み返す気力が沸きません。

現在実装できていない点として、 LaTeX で使えない文字列#<>&などの文字列のエスケープがあります。今はハイフンに変更してお茶を濁しています。

XPATH には文字列置換のための transform() 関数がありますが、一文字を一文字に置換することしかできません。今回行いたいのは一文字を複数文字に置換することなのでそれでは機能が不十分です。

4.7 仮の定量的な比較

現状すべての機能をおきかえているわけではないので、妥当な比較ではないですが、 C++ の処理と XSLT のコードの比較をしてみると (1)、 XSLT のほうが行数は少ないことがわかります。

表 1 lines of code for each implementation

	c++	xslt
txt	151	53
html	157	99
latex(PDF)	158	87

4.8 結論

XSLT を使うことでメンテナンスする行数は少なくなります。しかし、 XPATH / XSLT により提供されている機能が制限されているため、その中で実現しにくい機能についてはがんばるか提供を諦めるのか、難しい判断を迫られます。

参考文献

[1] Dimitre Novatchev, “Functional programming in XSLT using the FXSL library,” Extreme Markup Languages 2003.

[2] James Clark, “XSL Transformations (XSLT) Version 1.0,” W3C Recommendation 16 November 1999. http://www.w3.org/TR/xslt

[3] James Clark, Steve DeRose, “XML Path Language (XPath) Version 1.0,” W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath/

[4] Junichi Uekawa, “dancer-xml - Simple non-comformant XML parsing library,” 2000. http://www.netfort.gr.jp/~dancer/software/dancer-xml.html

5 Debian で Sphinx と Doxygen を使ってみた

まえだこうへい

5.1 最近の流行りのようです

Python 関連のプロジェクトやエンジニアを中心に最近流行っているようです。 Sphinx-Users.jp のサイトを見る[*13]と、 2011 年 6 月現在、 Sphinx のデフォルトテーマだけでなく、カスタムテーマやオリジナルテーマを使った、 50 弱の日本語のサイトが紹介されています。

5.1.1 reST と Sphinx の概要

Sphinx は reST(reStructuredText) という軽量マークアップ言語で書いたソースを様々なフォーマットのドキュメントに変換・生成するためのツールです。出力可能なフォーマットには、 HTML、 LaTeX、 PDF、 ePub、 man、平文テキスト、 JSON などがあります。

5.1.2 reST のサンプル

試しに、東京エリア Debian 勉強会のページを reST で書くとこんな感じになります。

```
========================
 東京エリア Debian 勉強会
========================


背景
====

2005 年当初、東京近辺で、類似の勉強会は存在していませんでした。
Debian について語る場所を提供するため、 Debian 勉強会を開催します。
この勉強会では毎回事前課題を設定しています。
その課題を提出することが参加の条件です。
参加する方は宴会の都合もありますので、事前に登録してください。
また、当日は Debian についての知識に関した簡単な試験を実施するので、
勉強会の会場には筆記用具を持参ください。

現在、 Debian 勉強会は
‘Debian JP Project <http://www.debian.or.jp/>‘_
のメンバーが Debian JP の公式なイベントとして運営しています。

次回の勉強会
============

* ‘2011 年 6 月勉強会（第 77 回東京エリア Debian 勉強会）<2011-06.html>‘_
* ‘第 48 回関西 Debian 勉強会 <http://wiki.debian.org/KansaiDebianMeeting20110626>‘_
* ‘毎週開催のハックカフェ <hackcafe.html>‘_
(snip)
```

具体的な書式については、 Sphinx-Users.jp のドキュメント[*14]を参照してください。

[*13] http://sphinx-users.jp/example.html
[*14] http://sphinx-users.jp/doc.html

5.2 Sphinx を使うきっかけ

graphviz の dot 言語と似た書式でブロック図を生成できる blockdiag シリーズ[*15]という python で書かれたツールがあります。最近、岩松さんにスポンサーをお願いして、これらの Debian パッケージ化を行っています。これらの Sphinx 拡張機能 (spyhinxcontrib-blockdiag など) を使うと、 Sphinx で生成するドキュメントの中にブロック図を埋め込むことができます。この blockdiag シリーズが便利なので、 Sphinx を使いだしたようなものです。

また、仕事では基本的に MS Office、とくに Excel や PowerPoint での文書作成がほとんどなのですが、今期の最初に「 もう MS Office なんてでやってられっかー、 Sphinx で作ろうぜ! 」と、プロジェクト内で提案して使い始めました。私個人で作る分には LaTeX でも良いのですが、他の二人は Windows しか普段触ったことがない上、文書と言えば上述のとおり、 Excel か PowerPoint、という状態です。まったく使ったことがない人に LaTeX 文書を作成させるのは敷居が高すぎます。しかし、 reST & Sphinx なら割と簡単に入門できる上、 Windows との共同作業の環境を整えるのもメンドイけど (LaTeX 環境を整えるよりも) 楽だった、という経緯です。[*16]

5.3 Debian で使ってみる

試しに先ほどの東京エリア Debian 勉強会のホームページを reST で書いたものを Sphinx で管理してみましょう。

まずは python-sphinx パッケージをインストールしておきます。

```
$ sudo apt-get install python-sphinx
```

emacs を使う場合は、 python-docutils パッケージをインストールしておけば、拡張子が rst か rest の場合、 rst.el によって自動的に ReST モードになります。

```
$ sudo apt-get install python-docutils
```

Sphinx プロジェクトを作ります。プロジェクト用のディレクトリを作ります。

```
$ mkdir tokyodebian
$ cd tokyodebian
```

作成したディレクトリに移動して、 `sphinx-quickstart` コマンドを実行します。

```
$ sphinx-quickstart
```

このコマンドを実行すると対話形式で聞かれます。 html を生成するので、 Project Name, Author name(s), Project Version 以外はデフォルトのまま (Enter を押下) で良いでしょう。 (表 2)

先ほどの tokyodebian.rst(および、 hackcafe.rst, 2011-06.rst) をコピーします。

```
$ cp -i ~/*.rst .
```

自動的に生成される index.rst にこれらを追記します。[*17]

[*15] `http://blockdiag.com/`

[*16] Windows は改めてマンドイと思いました。 `http://d.hatena.ne.jp/mkouhei/20110521/1305905297`

[*17] 拡張子不要です。

表 2　sphinx-quickstart の設定項目

設定項目	デフォルト値	設定例
Root path for the documentation	.	デフォルト
Separate source and build directories (y/N)	n	デフォルト
Name prefix for templates and static dir	_	デフォルト
Project name:		Tokyo Debian Meeting
Author name(s)		Debian JP Project
Project version		1.0
Project release	1.0	デフォルト
Source file suffix	.rst	デフォルト
Name of your master document (without suffix)	index	デフォルト
Do you want to use the epub builder (y/N)	n	デフォルト
autodoc: automatically insert docstrings from modules (y/N)	n	デフォルト
doctest: automatically test code snippets in doctest blocks (y/N)	n	デフォルト
intersphinx: link between Sphinx documentation of different projects (y/N)	n	デフォルト
todo: write “todo” entries that can be shown or hidden on build (y/N)	n	デフォルト
coverage: checks for documentation coverage (y/N)	n	デフォルト
pngmath: include math, rendered as PNG images (y/N)	n	デフォルト
jsmath: include math, rendered in the browser by JSMath (y/N)	n	デフォルト
ifconfig: conditional inclusion of content based on config values (y/N)	n	デフォルト
viewcode: include links to the source code of documented Python objects (y/N)	n	デフォルト
Create Makefile? (Y/n)	y	デフォルト
Create Windows command file? (Y/n)	y	デフォルト

```
$ sensible-editor index.rst
---
.. Tokyo Debian Meeting documentation master file, created by
   sphinx-quickstart on Fri Jun 17 13:39:53 2011.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to Tokyo Debian Meeting's documentation!
================================================

Contents:

.. toctree::
   :maxdepth: 2

   tokyodebian  ←追加
   hackcafe  ←追加
   2011-06  ←追加

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

コンパイルします。

```
$ make html
sphinx-build -b html -d _build/doctrees   . _build/html
Running Sphinx v1.0.7
loading pickled environment... done
building [html]: targets for 4 source files that are out of date
updating environment: 0 added, 4 changed, 0 removed
reading sources... [ 25%] 2011-06
reading sources... [ 50%] hackcafe
reading sources... [ 75%] index
reading sources... [100%] tokyodebian

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [ 25%] 2011-06
writing output... [ 50%] hackcafe
writing output... [ 75%] index
writing output... [100%] tokyodebian

writing additional files... genindex search
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.

Build finished. The HTML pages are in _build/html.
```

_build/html/ディレクトリの下に reST から生成された HTML ファイルができます。

5.4 Debian の日本語環境での状況

HTML の場合は日本語も問題なく表示できました。他のフォーマットはどうでしょうか。結果は下記のとおりです。(表 3)

表 3 フォーマット毎のビルド結果

フォーマット	結果
html	OK
epub	OK (ただし、 CSS は反映されない)
text	OK
man	OK
latex	OK
latexpdf	NG

上記のとおり、 LaTeX から PDF への生成がうまくできません。

```
(snip)
! PACKAGE INPUTENC ERROR: UNICODE CHAR \U8: NOT SET UP FOR USE WITH LATEX.

SEE THE INPUTENC PACKAGE DOCUMENTATION FOR EXPLANATION.
Type  H <return>  for immediate help.
 ...

l.119 \chapter{東京エリア Debian 勉強会}

?
(snip)
```

これは生成される LaTeX 文書が UTF-8 であるためです。 Debian JP Project での課題にもなっていますが、現状の Debian の TeX 系では日本語の UTF-8 は未対応です。

また、日本語を使っていなくても、 GIF イメージを".. image::" で読み込んでいる場合に PDF の生成に失敗するようです。

5.4.1 rst2pdf を使う方法

reST から PDF への生成には、 LaTeX 経由での方法以外に、 rst2pdf というツールを使う方法もあります。まず、rst2pdf パッケージをインストールします。

```
$ sudo apt-get install rst2pdf
```

インストール後、先ほど作った Sphinx のプロジェクトディレクトリの直下に conf.py という設定ファイルがあるので、この中の extensions に下記を追記します。

```
extensions = ['sphinx.ext.autodoc','rst2pdf.pdfbuilder']
```

PDF のオプションを追記します。

```
pdf_documents = [
        ('index',u'TokyoDebianMeeting', u'Tokyo Debian Meeting', u'Debian JP Project'),
]
pdf_stylesheets = ['sphinx','kerning','a4','ja']
pdf_font_path = ['/usr/share/fonts/']
pdf_language = 'ja_JP'
```

Makefile に下記を追記します。

```
pdf:
        $(SPHINXBUILD) -b pdf $(ALLSPHINXOPTS) $(BUILDDIR)/pdf
        @echo
        @echo "Build finished. The pdf files are in $(BUILDDIR)/pdf."
```

ja.json ファイルを作ります。

```
{
  "fontsAlias" : {
    "stdFont": "ttf-japanese-gothic",
    "stdBold": "ttf-japanese-gothic",
    "stdItalic": "ttf-japanese-mincho",
    "stdBoldItalic": "ttf-japanese-mincho",
    "stdMono": "ttf-japanese-gothic"
  }
}
```

make pdf を実行すると、_build/pdf/TokyoDebianMeeting.pdf が生成されます。日本語の表示も問題ありません。詳細については、/usr/share/doc/rst2pdf/manual.pdf.gz にマニュアルがあるので、これの「Section 18 Sphinx」のページを参照してください。なお、この場合は make latexpdf ではうまくいかなかった Gif ファイルの読み込みは問題ありません。

しかし、この方法では sphinxcontrib.*diag を使うと、ビルドに失敗するという別の問題があります。

5.5 Doxygen とは

さて、今回のもう一つのドキュメント生成ツールである Doxygen について見てみます。 Doxygen はソースコードを解析してドキュメントを生成するツールです。対応する言語は C/C++、 Java、 Python、 C#、 Objective-C などをサポートし、 D や PHP も部分的にサポートしています。

一方、生成可能なフォーマットは、 HTML、 LaTeX、 RTF(MS-Word)、 PostScript、 PDF、 man などがあります。

5.5.1 Debian で使ってみる

今回は、 Debian 勉強会参加登録システムのソースコードからドキュメントを生成してみることにします。

Debian パッケージがあるので、 doxygen パッケージをインストールします。

```
$ sudo apt-get install doxygen
```

次に、ソースツリーのルートディレクトリに移動し、設定ファイルを生成します。

```
$ cd monthly-report/utils/gae/
$ doxygen -g .doxgen.conf
```

Debian 勉強会参加登録システムは Python なので、最低限次の設定項目の設定を行います。 (表 4)

表 4 Doxygen の設定項目

設定項目	デフォルト値	設定例
PROJECT_NAME		Tokyo Debian Meeting
PROJECT_NUMBER		1.0
OUTPUT_LANGUAGE	English	Japanese
TAB_SIZE	8	4
INPUT		.
FILTER_PATTERNS		*.py

doxygen コマンドを実行します。

```
$ doxygen .doxygen.conf
```

すると、 monthly-report/utils/gae/ディレクトリ以下に、 html, latex ディレクトリができます。 html ディレクトリ以下には HTML 形式で、 latex ディレクトリ以下には、 LaTeX 及び PDF 形式でドキュメントが生成されます。

w3m で html/index.html を見ると、以下のような画面が表示されます。

“クラス” リンクをクリックするとクラスの一覧が展開されます。

例えば、”admin_event::EditEvent” のリンクをクリックすると、 admin_event::EditEvent クラスについてのドキュメントを見ることができます。

Doxygen についての詳細は doxygen.jp[*18]のマニュアルを参照してください。

[*18] http://www.doxygen.jp/manual.html

5.5.2 Sphinx との連携

breathe[*19]というツールを使うと、 reST/Sphinx から Doxygen に連携できるようです[*20]。なお、 Debian パッケージにはなってません。

5.6 まとめ

ソースコードからドキュメントを作る Doxygen, またドキュメントの作成自体を簡単にする Sphinx を使うと、大変でなかなかやりたがらないドキュメントの作成の敷居を低くすることができます。また冒頭で紹介した Sphinx 拡張としても使える*diag シリーズや、まだ Doxygen と Sphinx を連携する Breathe を使うことによって、これらのドキュメント生成ツールの利用価値が上がります。

自分のドキュメント作成のモチベーションを上げる意味でも、*diag シリーズだけでなく、 Breathe についても、Debian パッケージ化を行おうと思います。

参考文献

[1] Georg Brandl, Shibukawa Yoshiki(Japanese), “Overview - Sphinx v1.0.6 documentation” 2007-2010. http://sphinx-users.jp/doc10/

[2] MiCHiLU “Sphinx で日本語 PDF を生成する” 2009. http://d.hatena.ne.jp/MiCHiLU/20091009/

[3] Dimitri van Heesch 1997-2010, OKA Toshiyuki (Japanese translation) 2001, TSUJI Takahiro (Japanese translation) 2006-2011, TAKAGI Nobuhisa (Japanese translation) 2006-2011, “Doxygen マニュアル” http://www.doxygen.jp/manual.html

[4] OKA Toshiyuki, “Doxygen を使おう” 2002. http://www.fides.dti.ne.jp/~oka-t/doxygen.html

[*19] https://github.com/michaeljones/breathe

[*20] http://sphinx.shibu.jp/faq.html

6　IPv6 のトンネル接続を試してみた話

西山和広

6.1　概要

今回の話は、プロバイダが IPv6 ネイティブ接続にまだ対応していない状況、つまり直接外に繋がるのは IPv4 のみの接続の環境で IPv6 を使う話です。

6.2　IPv6 概要

6.2.1　IPv6 とは何か

最近は情報が増えてきているので省略します。古い情報だと RFC が更新されていたり廃止されていたりして、現状とは合わなくなっているものもあるので注意が必要です。

6.2.2　IPv6 の利点と欠点

IPv6 の利点としては以下のようなものが思いつきます。

- アドレス空間が広い
- 実装が普及している
- 機能的な利点はそんなにない
  - IPsec とか Mobile IP とか IPv4 でも可能

IPv6 の欠点としては以下のようなものが思いつきます。

- IPv4 と互換性がない
- まだ広く利用されていない
  - トラブルの対処方法とかあまりない

6.2.3　なぜ IPv6 を試そうと思ったか

試し始めてから World IPv6 Day が実施されるなど、状況がどんどん変わっていますが、試そうと思った最初の理由は以下のようなものです。

- JPNIC の IPv4 アドレスも枯渇したから
- おもしろそうだったから
- 余裕のあるうちにのんびりとやりたかったから
  - 必要になってからあわててやりたくない

6.2.4 IPv6 対応とは

IPv6 対応にはいろいろな状態があると思いますが、今回は以下の種類を考えてみました。

- クライアント側
  - IPv6 のみのサーバに接続できる (`http://ipv6.google.com/` など)
  - IPv4 と IPv6 両対応のサーバに IPv6 で接続できる (`http://www.kame.net/` など)
  - DNS を IPv6 経由で解決できる (/etc/resolv.conf で nameserver に IPv6 アドレスを設定)
    - これが出来ないと IPv6 のみに移行できない
- サーバ側
  - IPv6 アドレス指定で接続できる (グローバル IPv6 アドレス設定)
  - ホスト名で指定した場合でも IPv6 で接続できる (DNS に AAAA レコード設定)

6.2.5 IPv6 アドレスの表記

IPv6 アドレスは 128 ビットを 16 ビットごとに「:」で区切って 16 進数で表記します。省略表記が RFC 4291 で決まっていますが、省略の仕方で複数の省略表記が可能なので、 RFC 5952 で推奨表記が決められました。

- 例: 2001:0db8:0000:0000:0000:0000:0000:0001
- RFC 4291, RFC 5952 のルールで省略表記
  先頭の 0 を省略　2001:db8:0:0:0:0:0:1
  0 の連続は 1 回だけ :: で省略　2001:db8::1
- 他にはアルファベットは小文字推奨など
- IPv4 のネットワークアドレスやサブネットマスクに相当するものは `2001:db8::/32` の `/32` のようにネットワークプレフィックスとそのビット長を付けて表記します。

`2001:db8::/32` は例示用 IPv6 アドレス (RFC 3849) になっているなど、用途により IP アドレスの範囲が決まっています。(RFC 5156)

6.3 トンネル

6.3.1 トンネルの種類

今回試したものは

- teredo (RFC 4380)
- 6to4 (RFC 3056)
- 6rd (RFC 5969)

の 3 種類です。 ISATAP (RFC 5214) など他の方式もありますが、今回の話の対象外です。

6.4 teredo

6.4.1 teredo の特徴

teredo は以下の特徴があります。

- NAT の中でも使えるトンネル
- UDP/IPv4 にカプセル化して通信
- クライアントで使うには手軽で簡単
- サーバには向かない (仕組みを考えるとたぶん無理)

この特徴により NAT の中から IPv6 接続したい人には便利なのですが、外部との接続を制限すべき環境では、 firewall などで UDP の 3544 番ポートへの接続を制限する必要があります。

6.4.2 IPv6 アドレス

- 接続先でも 2001:0000::/16 のアドレス (省略しない場合 2001:0000: で始まる文字列の IPv6 アドレス) は teredo とわかるので WWWW:WWWW の部分を逆引きするなどの対処が可能です。
- 2001:0000:XXXX:XXXX:YYYY:ZZZZ:WWWW:WWWW の形式
  - XXXX:XXXX は Teredo サーバの IPv4 アドレスを 16 進数にしたもの
  - YYYY は NAT の種類などのフラグ
  - ZZZZ はクライアントの外部 UDP ポート番号を変換したもの
  - WWWW:WWWW は Teredo クライアントの外部 IPv4 アドレスを変換したもの

6.4.3 使い方

- Windows は XP 以降で対応しています。
- Debian では miredo パッケージをインストールするだけで teredo で接続できます。
  - 自動で起動
  - /etc/miredo.conf で設定
  - デフォルトの ServerAddress の teredo-debian.remlab.net はフランスなので、アメリカにある teredo.ipv6.microsoft.com に変更した方が良いかもしれません

6.4.4 接続確認

/sbin/ifconfig で teredo の存在を確認します。

```
$ /sbin/ifconfig teredo
teredo    Link encap:不明なネット  ハードウェアアドレス 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet6 アドレス: fe80::ffff:ffff:ffff/64 範囲:リンク
          inet6 アドレス: 2001:0:4137:9e76:34c1:f58:XXXX:XXXX/32 範囲:グローバル
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1280  メトリック:1
          RX パケット:0 エラー:0 損失:0 オーバラン:0 フレーム:0
          TX パケット:3 エラー:0 損失:0 オーバラン:0 キャリア:0
      衝突 (Collisions):0 TX キュー長:500
          RX バイト:0 (0.0 B)  TX バイト:144 (144.0 B)
$
```

接続に成功しているとき、ログ (/var/log/syslog) に以下のように出ます。

```
miredo[4105]: Starting...
miredo[4107]: New Teredo address/MTU
miredo[4107]: Teredo pseudo-tunnel started
miredo[4107]:  (address: 2001:0:4137:9e76:34c1:f58:XXXX:XXXX, MTU: 1280)
```

6.5 6to4

6.5.1 6to4 の特徴

6to4 はグローバル IPv4 アドレスが必要ですが、申し込みなどをしなくても誰でも自由に使えます。

トンネルの方法として、プロトコル 41 (TCP や UDP ではない) でカプセル化して IPv4 で通信します。 (他の主なプロトコルの番号は ICMP: 1, TCP: 6, UDP: 17 でその層のプロトコルを使用しています。)

anycast で自動的に近いリレールータを経由 (192.88.99.1) します。

6.5.2 6to4 の問題点

6to4 には以下のような問題点があるため、廃止が検討されています。

- 往復の経路は基本的に異なる

- 通信経路が把握できないし制御もできない
- トラブルがおきると対処が困難

6.5.3 IPv6 アドレス

接続先でも 2002::/16 のアドレス (2002: で始まる文字列の IPv6 アドレス) は 6to4 とわかるので、アクセスログ解析などでは xxx.yyy.zzz.www を逆引きするなどの対処が可能です。

2002:XXYY:ZZWW::/48 が使用可能

- XXYYZZWW は IPv4 アドレスの xxx.yyy.zzz.www を 16 進数にしたもの

2002:XXYY:ZZWW:VVVV::/64 を LAN に割り当てるなどの使い方が可能 (VVVV は任意の値)

6.5.4 使い方

基本的には「 /etc/network/interfaces」で設定するだけで使えます。

6to4 による IPv6 接続 (Linux 編) ≪ さくらインターネット研究所 `http://research.sakura.ad.jp/2010/12/27/tunnel-6to4-linux/` などを参考にして設定します。

以下は会社のマシンで設定したときの例です。

まず 6to4 のアドレスを計算します。

```
$ printf "2002:%02x%02x:%02x%02x::1\n" 220 218 54 201
2002:dcda:36c9::1
```

次に tun6to4 という iface の設定を追加します。

```
$ sudoedit /etc/network/interfaces
auto tun6to4
iface tun6to4 inet6 v4tunnel
address 2002:dcda:36c9::1
netmask 16
gateway ::192.88.99.1
local 220.218.54.201
endpoint any
ttl 64
```

最後に「 sudo ifup tun6to4」で有効にします。「 auto tun6to4」も設定しているので、再起動でも有効になるはずです。

6.5.5 接続確認

/sbin/ifconfig で tun6to4 を確認します。

```
$ /sbin/ifconfig tun6to4
tun6to4   Link encap:IPv6-in-IPv4
          inet6 アドレス: 2002:dcda:36c9::1/16 範囲:グローバル
          inet6 アドレス: ::220.218.54.201/128 範囲:Compat
          UP RUNNING NOARP  MTU:1480  メトリック:1
          RX パケット:55939508 エラー:0 損失:0 オーバラン:0 フレーム:0
          TX パケット:84141144 エラー:1175 損失:0 オーバラン:0 キャリア:886
      衝突 (Collisions):0 TX キュー長:0
          RX バイト:5470501110 (5.0 GiB)  TX バイト:88417369465 (82.3 GiB)

$
```

6.6 6rd

6.6.1 6rd の特徴

- 6to4 と同様にリレールータを経由
- リレールータはプロバイダが用意
- プレフィックスもプロバイダのものになる
- teredo や 6to4 と違って IPv4 の方が優先されるということがない

6.6.2 使い方

squeeze のカーネル 2.6.32 は 6rd に対応していないバージョンなので、バックポートカーネルをインストールします。 backports の apt-line を適切に設定した後、「 sudo aptitude install -t squeeze-backports linux-image-2.6.38-bpo.2-amd64」でインストールします。依存関係で linux-base も更新されるようです。

6rd による IPv6 接続 (概要編) ≪ さくらインターネット研究所 `http://research.sakura.ad.jp/2011/01/05/tunnel-6rd-intro/` を参考にして「 /etc/network/interfaces」に設定します。

以下はさくらの VPS で squeeze にしているマシンで設定した例です。

まず 6rd のアドレスを計算します。

```
$ printf "2001:55c:%02x%02x:%02x%02x::1\n" 49 212 40 201
2001:55c:31d4:28c9::1
$
```

次に tun6rd という iface の設定を追加します。

```
$ sudoedit /etc/network/interfaces
auto tun6rd
iface tun6rd inet6 v4tunnel
  address 2001:e41:31d4:28c9::1
  netmask 32
  local 49.212.40.201
  endpoint any
  gateway ::61.211.224.125
  ttl 64
  up ip tunnel 6rd dev tun6rd 6rd-prefix 2001:e41::/32
  up ip link set mtu 1280 dev tun6rd
```

最後に「 sudo ifup tun6rd」で有効にします。「 auto tun6rd」も設定しているので、再起動でも有効になるはずです。

6.6.3 接続確認

/sbin/ifconfig で tun6rd を確認します。

```
$ /sbin/ifconfig tun6rd
tun6rd    Link encap:IPv6-in-IPv4
          inet6 addr: 2001:e41:31d4:28c9::1/32 Scope:Global
          inet6 addr: ::49.212.40.201/128 Scope:Compat
          UP RUNNING NOARP  MTU:1280  Metric:1
          RX packets:16336 errors:0 dropped:0 overruns:0 frame:0
          TX packets:21361 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1854845 (1.7 MiB)  TX bytes:6845907 (6.5 MiB)

$
```

6.7 6to4 ルータ

6.7.1 IPv6 ルータとは

- ネットワークプレフィックスや有効期限を広告 (RA (Router Advertisement): ルータ広告)
- RA を受け取った端末が IPv6 アドレスを生成して自動設定 (RFC 4862)
- RA によるステートレスアドレス自動設定 (SLAAC)
  - 端末がネットワークに繋がると RS (Router Solicitation: ルータ要請) を「 ff02::2」 (全ルータアドレスあてのマルチキャスト) に送信
  - ルータが RA を「 ff02::1」 (全ノードアドレスあてのマルチキャスト) に送信
  - RA のプレフィックスを使って IPv6 アドレス生成
  - 重複アドレス検出 (DAD) して問題なければ利用開始
- IPv4 の DHCP と違って RA の送信元をデフォルトルートとして自動設定
- DNS 設定配布は別途考える必要あり
  - DNS の設定は今のところ別途 DHCPv6 を使うのが無難?
  - RA の RDNSS というオプション (RFC 6106) は使われていない?

DNS 設定についてはすぐにはわからなかったので、今回は IPv4 の DNS をそのまま使うことにして、 IPv6 の DNS サーバ設定はしませんでした。

セキュリティ問題も関係しているので、このあたりの仕様はまだ更新される可能性が高いです。

6.7.2 LAN 側 IPv6 アドレス設定

まず LAN 側に設定するネットワークプレフィックスを決めます。

今回は 6to4 により 2002:dcda:36c9::/48 が自由に使えます。その中から JPNIC の枯渇の日の 4 月 15 日を埋め込んで 2002:dcda:36c9:415::/64 を使うことにしました。

ルータ広告を送信する iface には固定の IPv6 アドレスが必須のようなので、今回はわかりやすいように先頭の 2002:dcda:36c9:415::1 を設定しました。

ネットワーク設定全体としては以下のようにしました。

```
$ cat /etc/network/interfaces
auto lo
iface lo inet loopback

allow-hotplug eth0
iface eth0 inet static
address 192.168.0.2
netmask 255.255.255.0
iface eth0 inet6 static
address 2002:dcda:36c9:415::1
netmask 64

allow-hotplug eth1
iface eth1 inet static
address 220.218.54.201
netmask 255.255.255.240
gateway 220.218.54.193

auto tun6to4
iface tun6to4 inet6 v4tunnel
address 2002:dcda:36c9::1
netmask 16
gateway ::192.88.99.1
local 220.218.54.201
endpoint any
ttl 64
$
```

6.7.3 radvd

radvd は RA (ルータ広告) を送信するデーモンです。

インストールしただけでは自動起動しないので、/usr/share/doc/radvd/README.Debian を参考にして設定して起動します。

```
$ sudo /etc/init.d/radvd start
Starting radvd:
* /etc/radvd.conf does not exist or is empty.
* See /usr/share/doc/radvd/README.Debian
* radvd will *not* be started.
$
```

まず README.Debian などを参考にして/etc/sysctl.d/ipv6.conf を作成しました。再起動前に反映させるには「 sudo sysctl -p /etc/sysctl.d/ipv6.conf」です。

```
$ cat /etc/sysctl.d/ipv6.conf
net.ipv6.conf.all.accept_ra=0
net.ipv6.conf.all.forwarding=1
```

/etc/radvd.conf は/usr/share/doc/radvd/examples/radvd.conf.example を参考にして以下のように設定しました。

グローバル IP アドレスから一意に決まるものをわざわざ書くのは嫌だったので、 prefix の先頭は「 0:0:0」にして「 Base6to4Interface eth1;」で自動設定するようにしています。

```
$ cat /etc/radvd.conf
interface eth0
{
        AdvSendAdvert on;
        MinRtrAdvInterval 3;
        MaxRtrAdvInterval 10;
        AdvDefaultPreference low;
        AdvHomeAgentFlag off;
        prefix 0:0:0:0415::/64
        {
                AdvOnLink on;
                AdvAutonomous on;
                AdvRouterAddr off;
                Base6to4Interface eth1;
                AdvPreferredLifetime 120;
                AdvValidLifetime 300;
        };
};
```

後は「sudo /etc/init.d/radvd start」して LAN 内のマシンにグローバル IPv6 アドレスが割り振られることを確認したり `http://ipv6.google.com/` が表示できることを確認したりします。

6.7.4 libvirt に IPv6 設定

この libvirt の設定だけ Debian squeeze ではなく Ubuntu 11.04 (natty) で確認しています。

「virsh net-edit default」で network 要素直下に「<ip family='ipv6' address='2002:dcda:36c9:415::1' prefix='64' />」のように設定を追加します。普通は「</network>」の行の直前に追加すれば良いと思います。

試した環境では一度保存すると以下のように 2 行になってしまいましたが、XML 的にはほぼ同じなので気にしなくても良いと思います。

```
<ip family='ipv6' address='2002:dcda:36c9:415::1' prefix='64'>
</ip>
```

全体的な例は `http://libvirt.org/formatnetwork.html` を参考にしてください。

これだけで自動的に virbr0 に「2002:dcda:36c9:415::1/64」のアドレスが振られたり radvd が起動したりします。

6.8 プライバシ拡張アドレス

RA によるステートレスアドレス自動設定 (SLAAC) で現状の Linux の実装などでは MAC アドレスを元にしたアドレスになります。(MAC アドレスを 1 ビット変更して FF FE を真ん中に入れる)

そのため、接続する IPv6 ネットワークが変わって prefix が違っても接続元マシンが特定できる可能性があります。

その対策として、プライバシ拡張アドレス (RFC 3041) の機能を有効にすればランダムな値から IPv6 アドレスが生成されるようになります。

実際の設定については試していないので紹介しておくだけにします。

- Ubuntu Linux で匿名アドレス (RFC3041) を有効にする `http://dr.slump.jp/IPv6/rfc3041/`
  - 例: `sysctl -w net.ipv6.conf.eth0.use_tempaddr=2`
- 高木浩光@自宅の日記 - Mac ユーザは IPv6 を切るか `net.inet6.ip6.use_tempaddr=1` の設定を `http://takagi-hiromitsu.jp/diary/20080730.html`
- iPhone、RFC3041 (IPv6 プライバシ拡張) に対応
  Kenichi Maehashi's Blog `http://blog.kenichimaehashi.com/?article=13044202150`
  - iOS 4.3 のセキュリティコンテンツについて
    `http://support.apple.com/kb/HT4564?viewlocale=ja_JP&locale=ja_JP`

6.9 firewall

6.9.1 注意点

IPv6 ではルータではない一般のノードでもユニキャストアドレスやループバックアドレス以外に、リンクローカルアドレスやマルチキャストアドレスなど複数の IPv6 アドレスを持つので、 firewall の設定に IPv4 より注意が必要です。

IPv4 で OP25B などの設定をしている場合は teredo などが抜け穴にならないように注意が必要です。

6.10 ip6tables-apply

- iptables パッケージに入っている ip6tables-restore をリモートからでも安全に実行できるようにするものです。
- タイムアウトするまでに y を入力しなかったら元のルールに戻してくれるので、設定をミスして ssh の接続が遮断されるルールにしようとしてしまっても安全です。
- /usr/sbin/iptables-apply で /etc/network/iptables を適用できるのと同様に /usr/sbin/ip6tables-apply で /etc/network/ip6tables を適用できます。
- 起動時にも適用するには「 /sbin/ip6tables-restore < /etc/network/ip6tables」をどこかで実行する必要があります。
- たとえば以下のように lo の pre-up で実行します。

```
iface lo inet loopback
 pre-up /sbin/iptables-restore < /etc/network/iptables
 pre-up /sbin/ip6tables-restore < /etc/network/ip6tables
```

6.11 ufw

- Uncomplicated FireWall の略
  - Ubuntu FireWall ではない
- sudo ufw allow OpenSSH や sudo ufw allow 80/tcp などのように簡単に使える iptables のラッパー

6.11.1 ufw の設定ファイル

/etc/default/ufw で基本的な設定をして、 ufw コマンドでその他の設定をして、 before{,6}.rules で特殊な設定をするということになります。

- /etc/default/ufw
  - デフォルトのポリシーなどを設定
- /etc/ufw/ufw.conf
  -「 ufw enable」や「 ufw disable」などの設定が保存されている
- /lib/ufw/user{,6}.rules
  - ufw allow などでの設定を保存
  - なぜか /lib 以下にある
- /etc/ufw/before{,6}.rules
  - 必要なら編集
  - ポートフォワーディングなどの nat テーブルの設定は ufw コマンドでは出来ないので、このファイルで設定

6.11.2 /etc/default/ufw

- IPV6=yes で IPv6 も有効にする
- IPV6=yes にする前に設定した ufw allow は IPv4 のみのまま

- IPv6 でも許可するには sudo ufw allow 22/tcp などを実行し直す
- from や to で IPv4 アドレスや IPv6 アドレスを指定すれば個別の設定も可能 (例: ufw allow in on virbr0 proto udp from 0.0.0.0/0 port 68 to 0.0.0.0/0 port 67)

こまめに「sudo ufw status」で確認するとわかりやすいです。IPv6 でも許可できていれば以下のように (v6) の行があります。

たとえば IPV6=no のときに「sudo ufw allow OpenSSH」で許可していて、IPV6=yes にしてから IPv6 でも許可した場合は以下のようになります。

```
$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere

$ sudo ufw allow OpenSSH
Skipping adding existing rule
Rule added (v6)
$ sudo ufw status
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)

$
```

6.11.3 /etc/ufw/before{,6}.rules

/etc/default/ufw で $\mathrm{DEFAULT}_{\mathrm{OUTPUTPOLICY}}$ を REJECT にした場合は ufw{,6}-before-input と同様の icmp などの許可を ufw{,6}-before-output にする必要がありました。

before.rules の ICMP の設定として

```
# ok icmp codes
-A ufw-before-input -p icmp --icmp-type destination-unreachable -j ACCEPT
-A ufw-before-input -p icmp --icmp-type source-quench -j ACCEPT
-A ufw-before-input -p icmp --icmp-type time-exceeded -j ACCEPT
-A ufw-before-input -p icmp --icmp-type parameter-problem -j ACCEPT
-A ufw-before-input -p icmp --icmp-type echo-request -j ACCEPT
```

の下に

```
-A ufw-before-output -p icmp --icmp-type destination-unreachable -j ACCEPT
-A ufw-before-output -p icmp --icmp-type source-quench -j ACCEPT
-A ufw-before-output -p icmp --icmp-type time-exceeded -j ACCEPT
-A ufw-before-output -p icmp --icmp-type parameter-problem -j ACCEPT
-A ufw-before-output -p icmp --icmp-type echo-request -j ACCEPT
```

を追加しました。

before6.rules の ICMPv6 の設定として

```
# for stateless autoconfiguration (restrict NDP messages to hop limit of 255)
-A ufw6-before-input -p icmpv6 --icmpv6-type neighbor-solicitation -m hl --hl-eq 255 -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type neighbor-advertisement -m hl --hl-eq 255 -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type router-solicitation -m hl --hl-eq 255 -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type router-advertisement -m hl --hl-eq 255 -j ACCEPT
```

```
# ok icmp codes
-A ufw6-before-input -p icmpv6 --icmpv6-type destination-unreachable -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type packet-too-big -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type time-exceeded -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type parameter-problem -j ACCEPT
-A ufw6-before-input -p icmpv6 --icmpv6-type echo-request -j ACCEPT
```

の下にそれぞれ

```
-A ufw6-before-output -p icmpv6 --icmpv6-type neighbor-solicitation -m hl --hl-eq 255 -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type neighbor-advertisement -m hl --hl-eq 255 -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type router-solicitation -m hl --hl-eq 255 -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type router-advertisement -m hl --hl-eq 255 -j ACCEPT
```

```
-A ufw6-before-output -p icmpv6 --icmpv6-type destination-unreachable -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type packet-too-big -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type time-exceeded -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type parameter-problem -j ACCEPT
-A ufw6-before-output -p icmpv6 --icmpv6-type echo-request -j ACCEPT
```

を追加しました。

MULTICAST は 5353/udp のみ許可の設定になっていますが、何か問題があれば変更します。

before.rules の MULTICAST 設定変更例としては

```
# allow MULTICAST mDNS for service discovery (be sure the MULTICAST line above
# is uncommented)
-A ufw-before-input -p udp -d 224.0.0.251 --dport 5353 -j ACCEPT
```

となっているのを以下に変更します。

```
# allow MULTICAST, be sure the MULTICAST line above is uncommented
-A ufw-before-input -s 224.0.0.0/4 -j ACCEPT
-A ufw-before-output -d 224.0.0.0/4 -j ACCEPT
```

上記のように ICMPv6 は許可しているので、 before6.rules の変更は通常は必要ないと思います。

6.12 tcp wrapper の設定例

localhost と libvirt のデフォルトの LAN と今回設定した 6to4 経由の接続のみ許可する場合は以下のようになります。

- /etc/hosts.deny で「 ALL: ALL」と設定
- /etc/hosts.allow で以下のように設定

```
sshd: 127.0.0.1 [::1]
sshd: 192.168.122.0/24
sshd: [2002:dcda:36c9:415::]/64
```

6.13 サーバ

6.13.1 サーバ一般

`netstat -lnt` で確認して tcp6 でも LISTEN していれば IPv6 対応しています。

```
$ netstat -lnt | grep ':22 '
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
$
```

6.13.2 OpenSSH

- 少なくとも以下の設定が IPv6 対応に影響します。
  - /etc/ssh/sshd$_{\text{config}}$
    - ListenAddress
    - AllowUsers user@ipaddr の ipaddr 部分
  - tcp wrapper
  - ip6tables

6.13.3 Apache2

一部の VirtualHost を IPv6 対応環境に移動しましたが、 IP アドレスに依存する設定をしていなかったので、何も問題はありませんでした。

6.13.4 munin

munin が使っているライブラリが IPv6 に対応していないという理由で munin も IPv6 に対応していませんでした。

6.14 トラブルシューティング

6.14.1 ping6 で最初のパケットだけ時間がかかる

- トンネル接続の場合はそういうもの

6.14.2 ping6 でパケットロスが多い

libvirt で libvirt で自動起動される radvd とは別に自分で radvd を起動してしまうと ping6 での確認のときに packet loss が 70% 以上になるなどわかりにくいトラブルになりました。

そのときに「 radvd[...]: our AdvValidLifetime on eth0 for ... doesn't agree with ...」のようなログが出ていて、調べると `http://www.wiggy.net/texts/ipv6-howto/` に radvd のようなルータ広告 (RA) するデーモンが複数動いている場合におきる問題だということがわかったので、 libvirt で自動起動されている radvd が存在するのを確認し、手動で起動していた方の radvd を止めることで解決しました。

6.14.3 ifconfig で teredo がない

- miredo のログ (/var/log/syslog) で原因を調べます。
- firewall で塞がれていないか確認します。
- NAT の種類によっては繋らないかもしれません
  - 以前のバージョンだと NAT の種類によっては「 Unsupported symmetric NAT detected.」で繋らないことがありましたが、今の squeeze に入っている miredo 1.2.3-1 では対応しているように見えます。
  - VMware などで NAT の段数を増やすと繋がったことがあります。

6.14.4 亀が踊らない

- `http://www.kame.net/` で亀が踊らないときの原因の調べ方
- IPv6 で接続可能か?
  - `http://ipv6.google.com/` が表示できるか?
    - ipv6 は AAAA のみ設定されている
    - 表示できなければ、そもそも IPv6 で外と繋っていない可能性が高い
  - 詳細は `http://test-ipv6.com/` や `http://test-ipv6.jp/` でテストする
- getaddrinfo(3) を調べる
  - 繋がっていれば、次に getaddrinfo(3) で IPv6 が優先されているかを調べます。
- getaddrinfo(3) の例: IPv6 優先
  こういう結果が返ってくれば IPv6 で繋るはずです。

```
$ getent ahosts www.kame.net
2001:200:dff:fff1:216:3eff:feb1:44d7 STREAM orange.kame.net
2001:200:dff:fff1:216:3eff:feb1:44d7 DGRAM
2001:200:dff:fff1:216:3eff:feb1:44d7 RAW
203.178.141.194 STREAM
203.178.141.194 DGRAM
203.178.141.194 RAW
```

- getaddrinfo(3) の例: IPv4 のみ
  以下のように A と AAAA が設定されているはずなのに A レコードの情報しか返ってこない場合は、プロバイダが "filter-aaaa-on-v4" の設定をしていてフィルタされているのかもしれないので、常用はお勧めしませんが Google

Public DNS (8.8.8.8, 8.8.4.4) を使えば回避できるかもしれません。

```
$ getent ahosts www.kame.net
203.178.141.194 STREAM orange.kame.net
203.178.141.194 DGRAM
203.178.141.194 RAW
$
```

- getaddrinfo(3) の例: IPv4 優先
  以下のような出力の場合は teredo や 6to4 よりも IPv4 が優先されている (RFC 3484) のが原因です。ipv6.google.com のように IPv6 のみのサイトに接続するときだけ teredo や 6to4 の IPv6 接続が使われるようになっています。

```
$ getent ahosts www.kame.net
203.178.141.194 STREAM orange.kame.net
203.178.141.194 DGRAM
203.178.141.194 RAW
2001:200:dff:fff1:216:3eff:feb1:44d7 STREAM
2001:200:dff:fff1:216:3eff:feb1:44d7 DGRAM
2001:200:dff:fff1:216:3eff:feb1:44d7 RAW
$
```

  RFC になっているようにこの挙動の方が望ましいので、/etc/gai.conf で設定可能ですが、一時的に変更するだけにして試した後は戻すべきです。
  Windows は netsh でポリシーテーブルを変更すれば良いそうです[*21]
  http://www.tokyo6to4.net/index.php/6to4%E3%81%AE%E5%88%A9%E7%94%A8%E6%96%B9%E6%B3%95#.E3.83.9D.E3.83.AA.E3.82.B7.E3.83.BC.E3.83.86.E3.83.BC.E3.83.96.E3.83.AB.E3.81.AE.E7.B7.A8.E9.9B.86.E6.96.B9.E6.B3.95
- gai.conf の設定
  http://slashdot.jp/~ohhara/journal/519278 を参考にして teredo で接続している場合は/etc/gai.conf に「 label 2001:0::/32 1」と設定すると亀が踊りました。試した後は忘れずに設定を戻しておきましょう。
  RFC 3484 に関連する設定として ip addrlabel もありますが、今回は変更しなくても大丈夫でした。
  デフォルトの設定は以下のようにすべてコメントで書かれていました。

```
#label ::1/128       0
#label ::/0          1
#label 2002::/16     2
#label ::/96         3
#label ::ffff:0:0/96 4
#label fec0::/10     5
#label fc00::/7      6
#label 2001:0::/32   7
```

  teredo の場合は以下のように変更しました。

```
label ::1/128       0
label ::/0          1
label 2002::/16     2
label ::/96         3
label ::ffff:0:0/96 4
label fec0::/10     5
label fc00::/7      6
label 2001:0::/32   1
```

  6to4 で接続している場合は同様に 2002::/16 を 1 にすると亀が踊りました。

```
label ::1/128       0
label ::/0          1
label 2002::/16     1
label ::/96         3
label ::ffff:0:0/96 4
label fec0::/10     5
label fc00::/7      6
label 2001:0::/32   7
```

  glibc のリゾルバの設定ファイルになるため、プロセスの起動時のみ読み込まれているようで、設定の反映にはブラ

[*21] 以下、リンクはページの都合上折り返していますが一行で

ウザの再起動が必要でした。
glibc の設定ファイルなので静的リンクされていて glibc を使わないバイナリなどは影響を受けないと思います。

6.14.5 端末は IPv6 対応なのに IPv6 のサイトに繋がらない

WPAD (Web Proxy Auto-Discovery Protocol) で設定していた proxy が IPv6 対応していないサーバで動いていたために繋がらないということがありました。

6.14.6 一部のサイトへのアクセスが遅い・繋がらない

- IPv6 から IPv4 へのフォールバックに時間がかかっている可能性がある
  - IPv6 経由で繋がらない原因を調べる
  - サーバ側で AAAA を登録しているのにサーバが止まっている (A の方では動いている) ということもあるらしい
    - 根本的な対処はサーバ側でしてもらうしかない
- Opera で何度かリロードしないと bit.ly など短縮 URL の一部を展開しなくなったということがあったらしい
  - `http://togetter.com/li/146832`
  - IPv6 を無効にしたら直ったという話

6.14.7 何も変えていないのに繋がらなくなった

- 上流の問題かもしれないので確認
  - メンテナンス中ではないか
    - 「 昨日の「 さくらの 6rd」接続不良ですが、弊社の IPv6 バックボーン側でメンテナンスが実施されていたようです。」 `https://twitter.com/jq6xze_1/status/83335591873351680`
  - radvd が止まっていないか
    - 止まっていれば起動
  - 不正な RA が送信されていないか
    - 不正な RA を送信しているルータを探して対処
    - Windows の ICS (インターネット接続の共有) が原因になることがあるらしい
    - 意図的に不正な RA が送信されていると対処は困難 (IPv4 の ARP spoofing に似た問題になる)
    - RFC 6105 (IPv6 Router Advertisement Guard) や RFC 3971 (SEcure Neighbor Discovery (SEND)) などで対処
  - 6to4 が廃止されてリレールータ消滅?
    - すぐにはなさそうですが、将来的には可能性がありそうです
  - teredo サーバ停止?
    - teredo で最初に接続する teredo サーバが停止していないか
    - teredo サーバへの接続が firewall などで止められていないか

6.14.8 短いと通信できるのに長いと通信出来ない

- MTU 問題ではないか

6.15 まとめ

- IPv6 接続はクライアントとして試すだけなら簡単でした。
- サーバも openssh や apache2 なら問題は起きにくいようです。

- munin や proxy など問題がなさそうと思っていたところで問題が起きることもあります。
- IPv6 の仕様は更新され続けていて、セキュリティ問題もまだまだこれからのようなので、引き続き最新の情報を追いかけていく必要がありそうです。

6.16 参考

参考文献

[1] 今からはじめる IPv6 ～プロトコル基礎編～ http://www.nic.ad.jp/ja/materials/iw/2010/proceedings/s2/iw2010-s2-01.pdf

[2] 絶対わかる! IPv4 枯渇対策& IPv6 移行超入門 ISBN 978-4-8222-6769-8

7 Haskell と Debian の辛くて甘い関係

岡部究

7.1 Haskell というプログラミング言語

Haskell [*22] というプログラミング言語をご存知でしょうか。 Haskell は関数型言語の一種で以下のような特徴があります。 (以下の意見は Haskell 初心者である筆者の偏見や間違いを多量に含んでいます)

- 静的型付け
  暗黙の型変換とかそんなことは起きません。また多くのエラーをコンパイル時に検出することができます。 Haskell でプログラミングをしていると視野角が狭くなる気分になると思います。型で守られることによって「 考慮に入れておくべき前提」のコード範囲が小さくなり、そしてインターフェイスに用いている型について「 本当にこれがふさわしいのか? 」と考えることになります。もっと簡単に言うと「 型による設計」を Haskell では行います。
- 型推論
  型をすべて書く必要がないという利点もありますが、型推論がないと綺麗に表現できないこともあります。個人的には、関数自体には型を書いて、関数の内部での型は省略することが行儀が良いと思います。
- パターンマッチ
  `if` や `case` 文で場合分けを記述するよりもはるかに柔軟な場合分けができます。アルゴリズムの記述とはある種場合分けの繰り返しとも言えるので、この場合分けを型で記述できると、わかりやすく簡潔になります。
- 遅延評価
  諸刃の剣ですが、無限リストを作れたり、破壊的なデータ構造を用いなくても計算量を少なくすることができます。正格性フラグを使うことで遅延評価を部分ごとに抑制することもできます。
- コンパイルして実行
  ローカルでコンパイルすれば、配布先には Haskell がインストールされていなくても OK です。要は単なる実行バイナリになります。また `runhaskell` コマンドでコンパイルせずに実行することもできます。
- 読みやすく、書きやすい文法
  本当です! もし型を使っても不足なケースでは Template Haskell [*23] を使えばコンパイル時にメタプログラミングをすることもできます。個人的には見た目があまりにかわりすぎてしまうので、邪悪なのではないかと思っていますが。。。まぁ使いどころに気をつけましょう。

どうでしょう。わくわくしますよね! さっそく使ってみましょう。なぁに Debian なら簡単です。[*24]haskell-platform パッケージをインストールすれば Haskell コンパイラである ghc とその基本ライブラリ群が使えるようになります。 Ruby の irb コマンドや、 Python の python コマンドに似た ghci コマンドというインタラクティブな Haskell 評価コマンドも

[*22] `http://haskell.org/`
[*23] `http://www.kotha.net/ghcguide_ja/latest/template-haskell.html`
[*24] ここでは Debian Sid を使っていることを前提にしています。

使えるようになります。

```
$ sudo apt-get install haskell-platform
$ rehash
$ ghci
GHCi, version 7.0.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> print $ fmap (foldr (++) "" . flip replicate "hoge") [1..3]
["hoge","hogehoge","hogehogehoge"]
```

7.2 cabal によるパッケージ管理

先程インストールした haskell-platform というのは Haskell 言語における標準ライブラリで、 GUI フレームワークとか Web アプリケーションフレームワークなどは入っていません。 (OpenGL はなぜか入ってますけれど) それじゃあ Haskell で書かれた最新のライブラリやプログラムを使おう、と思いますよね。 Haskell で書かれたプログラムの多くは Hackage [*25] というサイトに登録されています。そう。 Perl の CPAN や、 Ruby の gem にあたるものが Haskell にも用意されているのです。

一個ずつ tar 玉をダウンロードしてコンパイルするのでしょうか? いいえ大丈夫です。 `cabal` [*26] というコマンドがあります。この `cabal` コマンドは Hackage の依存関係を考えて所望のプログラムをインストールできるすぐれものです。

Debian の場合、以下の手順で任意の Hackage をインストールできます。

```
$ sudo apt-get install cabal-install # haskell-platform をインストールすれば自動でインストールされるので本当は不要です
$ rehash
$ cabal update
$ cabal install パッケージ名
```

7.3 でも cabal には色々不都合が、、、

もし `cabal` コマンドを長期にわたって使ったことがある方であれば体験していると思うのですが、 cabal コマンドはパッケージのインストールはできてもパッケージの更新をすることができません。

Ruby の gem を思い出してみましょう。

```
$ sudo gem update
$ sudo gem install earchquake
# 月日は流れ、、、そしてある日、、、
$ sudo gem update
# これで以前インストールいした earchquake パッケージは依存ライブラリを含めて最新版になるはず
```

ところが cabal の場合、筆者は以下のような不具合によく直面していました。

```
$ cabal update # これはローカルの Hackage データベースを更新するだけ
$ cabal install yesod # 実行後、インストール完了
```

これで色々開発したりして、、、楽しい月日は流れます。後日 yesod を最新版に更新しようと思いたちました。

```
$ cabal upgrade
cabal: Use the 'cabal install' command instead of 'cabal upgrade'.
You can install the latest version of a package using 'cabal install'. The
'cabal upgrade' command has been removed because people found it confusing and
it often led to broken packages.
If you want the old upgrade behaviour then use the install command with the
--upgrade-dependencies flag (but check first with --dry-run to see what would
happen). This will try to pick the latest versions of all dependencies, rather
than the usual behaviour of trying to pick installed versions of all
dependencies. If you do use --upgrade-dependencies, it is recommended that you
do not upgrade core packages (e.g. by using appropriate --constraint= flags).
```

なにこれーーーーーしょうがない、必要なパッケージだけ更新しましょう

[*25] `http://hackage.haskell.org/`

[*26] `http://www.haskell.org/cabal/` 正確なプログラム名は cabal-install。 Cabal はライブラリの名前。ちょっとややこしいです。

```
$ cabal install yesod # しかしなぜか yesod が動作しなかったり、そもそも依存関係を cabal が自動解決しない、、、
# とりあえず cabal でインストールした Hackage を全部消そう。。。
$ rm -rf ~/.ghc ~/.cabal
$ cabal update
$ cabal install yesod  # さっきの yesod のバグが再現しない。ふつーに動いとる。なぜだーーー!?
```

あれれ。インストールした時は問題なかったの何が起きたのでしょう。どうやらこのような不具合が起きるのは筆者だけではなく、多くの Haskell 開発者も同様のようです。どの開発者も本質的には cabal の環境をマッサラ (`rm -rf .cabal .ghc`) にしてから再インストールして凌いでいるようです。。。

7.4 cabal をパッケージシステムとして使うことの問題点

どうしてこんなことが起きてしまうのでしょう? それは cabal のしくみと Hackage 作者達の文化に問題があります。

7.4.1 Hackage 作成の文化的問題

まずは例として yesod パッケージの情報を覗いてみましょう。

```
$ cabal info yesod
* yesod            (program and library)
    Synopsis:      Creation of type-safe, RESTful web applications.
    Versions available: 0.6.7, 0.7.2, 0.7.3, 0.8.0, 0.8.1, 0.8.2, 0.8.2.1,
                        0.9.1, 0.9.1.1 (and 35 others)
    Versions installed: [ Not installed ]
    Homepage:      http://www.yesodweb.com/
--snip--
    Source repo:   git://github.com/yesodweb/yesod.git
    Executables:   yesod
    Flags:         ghc7
    Dependencies:  yesod-core >=0.9.1.1 && <0.10, yesod-auth ==0.7.*,
                   yesod-json ==0.2.*, yesod-persistent ==0.2.*,
                   yesod-form ==0.3.*, monad-control ==0.2.*,
                   transformers ==0.2.*, wai ==0.4.*, wai-extra >=0.4.1 && <0.5,
                   hamlet ==0.10.*, shakespeare-js ==0.10.*,
                   shakespeare-css ==0.10.*, warp ==0.4.*, blaze-html ==0.4.*,
                   base >=4.3 && <5, base >=4 && <4.3, base >=4 && <4.3,
                   base >=4.3 && <5, process -any, blaze-builder >=0.2 && <0.4,
                   http-types >=0.6.1 && <0.7, attoparsec-text >=0.8.5 && <0.9,
                   containers >=0.2 && <0.5, unix-compat >=0.2 && <0.4,
                   Cabal >=1.8 && <1.13, directory >=1.0 && <1.2,
                   template-haskell -any, time >=1.1.4 && <1.3,
                   bytestring ==0.9.*, text ==0.11.*, parsec >=2.1 && <4
    Cached:        No
    Modules:
        Yesod
```

まず見てとれるのが、”Versions available” 行です。 yesod パッケージは HackageDB に複数のバージョンが登録されているのがわかります。もう一つ気になるのは”Dependencies” 行です。 text や bytestring などの基本的のパッケージに対して A.B の桁までバージョンを指定しています。 Debian パッケージのほとんどは、依存は同ソースパッケージから生成されたものについてはバージョン番号を完全に指定、他パッケージへの依存は下限バージョン指定、となっているのとは対照的です。

このバージョン指定のポリシーはどこからやってきたかというと、 Hackage のバージョン番号のポリシー文書[*27] からです。おおざっぱに引用すると以下のような規則です。 Haskage のバージョン番号が仮に A.B.C.X と表わされる場合、、、

1. エントリの削除、エントリの型やデータ型定義やクラスの変更、インスタンスの追加/削除、 import の変更、他パッケージの新たなバージョンへの依存。のような場合には A.B バージョンを上げるべき
2. 上記に該当せず、新たなバインディング、型、クラス、モジュールがインターフェイスに追加された場合には A.B バージョンは同値のままでも良いが C バージョンを上げるべき
3. そうでない場合、 A.B.C は同値のままでも良い。 X などそれより桁が下のバージョンを上げるままでも良い

この規則を守ると、自分の依存している Hackage の API が削除されないように期待するためには”bytestring ==0.9.*” のように指定してくなるわけです。ところが、この指定方法によって cabal コマンドが依存関係の解決に

[*27] http://www.haskell.org/haskellwiki/Package_versioning_policy

混乱することがあるようです。

7.4.2 cabal の実装上の問題

先の HaskellImplementorsWorkshop/2011 にて新しい cabal の依存解決のしくみが発表されました。[*28] この中のスライド[*29] で現状の cabal の問題点が説明されています。

上記スライドから引用して説明します。

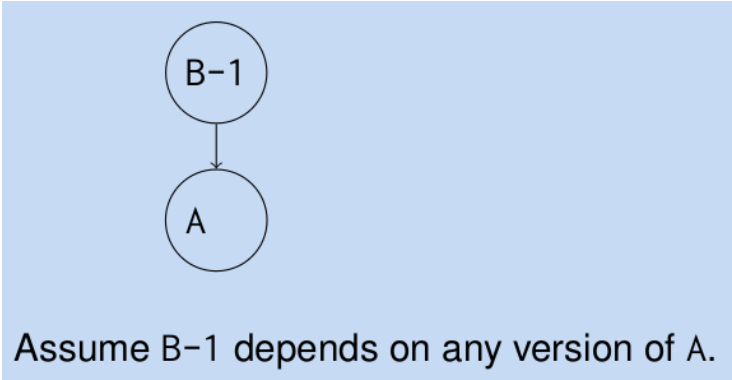

図 2　Hackage DB 上で B-1 パッケージが A に依存している場合

まず上図のように Hackage DB で B-1 パッケージが A パッケージに依存している場合を考えます。この時 B-1 は A のバージョンについて特に指定していないとします。

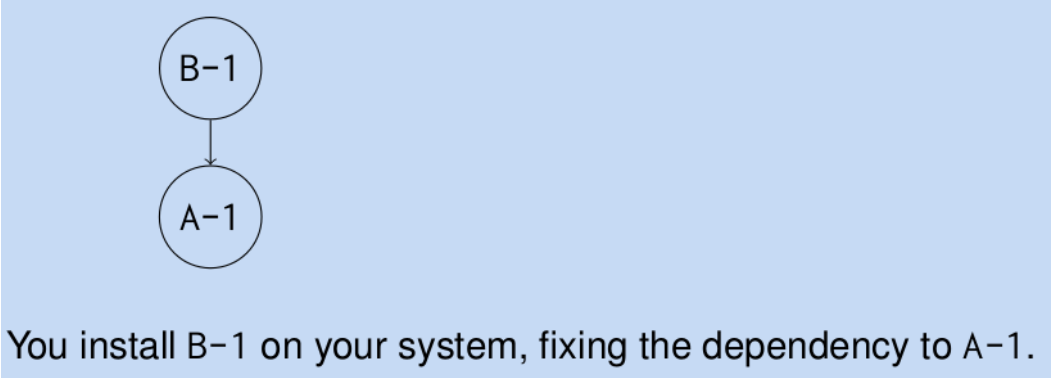

図 3　B-1 と A-1 をインストール

このような Hackage DB から B-1 をインストールすると A パッケージの最新バージョンである A-1 も一緒にインストールされます。

そうして、このような環境にさらに C-1 を含む B-1 依存した Hackage 群をインストールします。ここで、 B に依存している C-1 はローカルでは B-1 に紐づけられています。

ここで Hackage DB から D-1 をインストールしてみましょう。 D-1 は Hackage DB 上 (上図右) では A-2 と B-1 にバージョン指定で依存しています。ローカルには A-1 と B-1 がインストールされています。

このままローカルにインストールされている A-1 と B-1 を無変更で D-1 をインストールすることはできません。そこで、 cabal はインストール計画をたて、 A-1 のかわりに A-2 をインストールしようとします。

A-2,B-1,D-1 について `cabal` はインストール/更新を完了しました。しかし、 B-1 に依存していた Hackage については再コンパイルは行ないません。当然 B-1 に依存していた Hackage は依存が壊れたまま放置されてしまうことになり

[*28] `http://www.haskell.org/haskellwiki/HaskellImplementorsWorkshop/2011/Loeh`
[*29] `http://www.haskell.org/wikiupload/b/b4/HIW2011-Talk-Loeh.pdf`

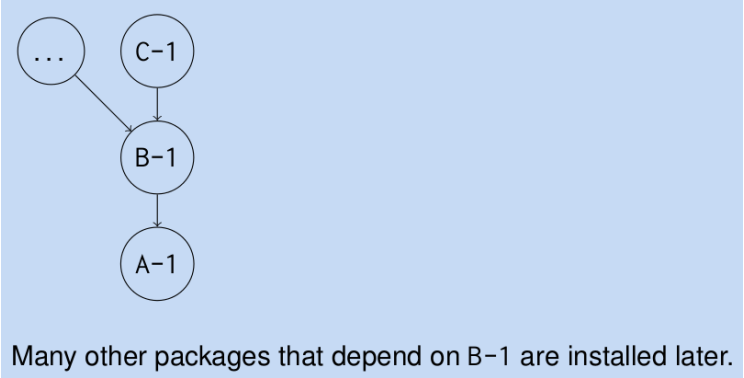

図 4　そしてさらに B-1 に依存した Hackage 群をインストール

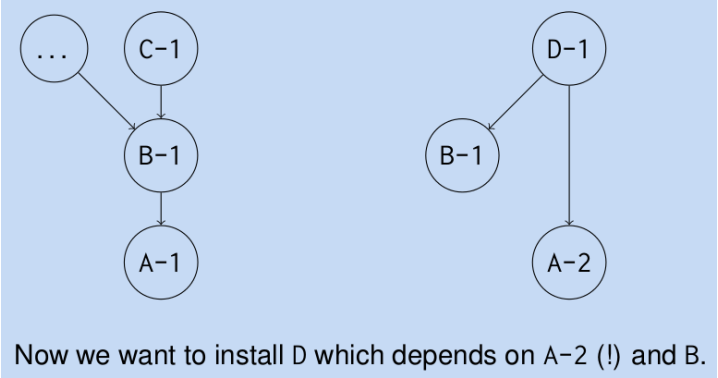

図 5　A-2 に依存している D-1 をインストールしようと試みる

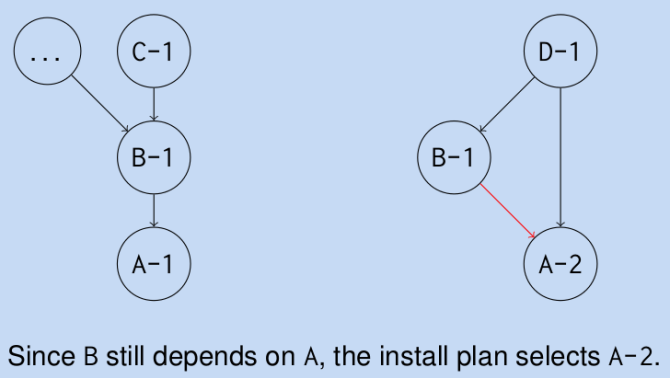

図 6　cabal は D-1 をインストールにあたって A-2 もインストールしようとする

ます。

この問題は依存解決する際のインストール計画の際にバックトラックが行なわれないためです。B-1 を再インストールするのであれば、それに依存した Hackage(C-1 など) も再インストールすべきだったのです。もちろん Haskell コミュニティではこの問題を認識しており、その解決のために新しいソルバを実装しています。[*30] 近い将来に本家 `cabal` に取り込まれることでしょう。

[*30] `http://darcs.haskell.org/cabal-branches/cabal-modular-solver`

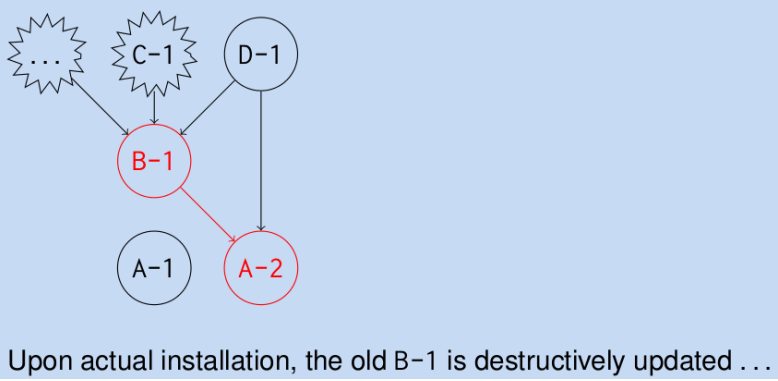

図 7　D-1 は正常にインストールされたが、 B-1 に依存していた Hackage 群は依存が壊れてしまう

7.4.3　Hackage が依存する環境について cabal コマンドは面倒をみてくれない

cairo [*31] のように C 言語に依存する Hackage については cabal コマンドは面倒を見てくれません。 Debian パッケージ libcairo2-dev が入っていない環境で cairo Hackage を `cabal` コマンドを使ってインストールしようとしても、 (当然) コンパイルエラーによってインストールに失敗します。

そもそも Debian では Haskell 以外の部分のパッケージは Debian パッケージ (deb) によって管理されています。 cabal コマンドは OS に依存していないので、 (当然)apt-get を呼び出すわけにもいきません。[*32]

7.4.4　Hackage 群全てを最新バージョンでインストールできないかもしれない

yesod [*33] ,hakyll [*34] ,hamlet [*35] の 3 つの Hackage を例に説明します。この問題は yesod-0.9.2, hakyll-3.2.0.8, hamlet-0.10.2 のバージョン間で生じていました。 (現在は解消されています)

まずそれぞれの Hackage について依存を見てみましょう。

```
$ cabal info yesod-0.9.2
* yesod-0.9.2            (program and library)
--snip--
    Dependencies:  yesod-core >=0.9.1.1 && <0.10, yesod-auth ==0.7.*,
--snip--
                   hamlet ==0.10.*, shakespeare-js ==0.10.*,
--snip--
$ cabal info hakyll-3.2.0.8
* hakyll-3.2.0.8           (library)
--snip--
    Dependencies:  base ==4.*, binary >=0.5 && <1.0, blaze-html >=0.4 && <0.6,
--snip--
                   filepath >=1.0 && <2.0, hamlet >=0.7 && <0.9,
```

あれ?　yesod-0.9.2 は hamlet-0.10.*に依存しているのに hakyll-3.2.0.8 は hamlet-0.7.*もしくは hamlet-0.8.*に依存しています。確かに表面上問題はありません。この状態でも yesod と hakyll の両方をインストールすることはできます。しかしもし hakyll と yesod 両方のライブラリを使いたいプログラムを作りたくなった場合にはどうしたら良いのでしょう?　hakyll はローカルで Web サーバを起動してプレビューする機能を持っています。たまたま hakyll はこの Web サーバのエンジンとして snap [*36] を使っていたから良かったものの yesod を使っていたら、古い yesod の機能しか使えないところです。

どうしてこんな状態で Hackage が放置されていたのでしょう? やる気がないのでしょうか? いえいえそんなことはありません。今度は hamlet について調べてみましょう。

[*31] `http://hackage.haskell.org/package/cairo`

[*32] `http://packages.debian.org/ja/sid/auto-apt` を使って cabal コマンド実行の裏で Debian パッケージを自動インストールする手はあるかもしれませんね ;-)

[*33] `http://hackage.haskell.org/package/yesod`

[*34] `http://hackage.haskell.org/package/hakyll`

[*35] `http://hackage.haskell.org/package/hamlet`

[*36] `http://hackage.haskell.org/package/snap`

hamlet-0.8.2.1 の Module リスト

```
Text
  Text.Cassius
  Text.Coffee
  Text.Hamlet
    Text.Hamlet.NonPoly
    Text.Hamlet.RT
  Text.Julius
  Text.Lucius
  Text.Romeo
  Text.Shakespeare
```

hamlet-0.9.0 の Module リスト

```
Text
  Text.Cassius
  Text.Coffee
  Text.Hamlet
  Text.Julius
  Text.Lucius
  Text.Romeo
  Text.Shakespeare
```

あれ? API に変更があるようです。ここで注目したいのは Text.Hamlet.RT モジュールが消滅していることです。嫌な予感がします。 hakyll のソースコード[*37] を見てみましょう。

```
-- | Read templates in the hamlet format
--
{-# LANGUAGE MultiParamTypeClasses #-}
module Hakyll.Web.Template.Read.Hamlet
    ( readHamletTemplate
    , readHamletTemplateWith
    ) where

import Text.Hamlet (HamletSettings, defaultHamletSettings)
import Text.Hamlet.RT

import Hakyll.Web.Template.Internal
--snip--
```

あー hakyll のコンパイルには Text.Hamlet.RT モジュールが必須なんですね。これでは新しい hamlet を使うことができない訳です。

Hackage 作者が自由に依存 Hackage のバージョンを選択可能である以上、このような Hackage 群全体の不整合は避けられません。

7.5 Hackage を Debian パッケージ化する

cabal を使って Debian パッケージと同等のレベルでパッケージ管理をするのは現状では難しいことがわかりました。それに apt-get でライブラリ環境が整うのは Debian ユーザとしてうれしいですよね。そこで、自分の良く使う Hackage は Debian パッケージ化して Debian 本体に登録してしまうのはいかがでしょうか。実は Hackage を Debian パッケージ化するのはすごく簡単です。 `cabal-debian` というまんまの名前のコマンドがあります。[*38] さっそくやってみましょう!

例題として HCWiid [*39] を Debian パッケージ化してみます。まず Hackage の Debian パッケージ化に必要な haskell-debian-utils, haskell-devscripts を apt-get install しましょう。

```
$ sudo apt-get install haskell-debian-utils haskell-devscripts
$ rehash
```

hackage をダウンロードして解凍したら、ディレクトリに移動しておもむろに `cabal-debian` コマンドを使います。

[*37] https://github.com/jaspervdj/hakyll/blob/master/src/Hakyll/Web/Template/Read/Hamlet.hs
[*38] http://hackage.haskell.org/package/debian
[*39] Wii リモコンからイベントを拾うためのライブラリ http://hackage.haskell.org/package/hcwiid

```
$ wget http://hackage.haskell.org/packages/archive/hcwiid/0.0.1/hcwiid-0.0.1.tar.gz
$ tar xfz hcwiid-0.0.1.tar.gz
$ cd hcwiid-0.0.1/
$ cabal-debian --debianize --ghc --maintainer="Kiwamu Okabe <kiwamu@debian.or.jp>"
$ ls debian
changelog  compat  control  copyright  rules*
$ debuild -rfakeroot -us -uc
--snip--
 dpkg-genchanges  >../haskell-hcwiid_0.0.1-1~hackage1_amd64.changes
dpkg-genchanges: including full source code in upload
 dpkg-source --after-build hcwiid-0.0.1
dpkg-buildpackage: full upload; Debian-native package (full source is included)
Now running lintian...
W: haskell-hcwiid source: native-package-with-dash-version
W: haskell-hcwiid source: out-of-date-standards-version 3.9.1 (current is 3.9.2)
E: libghc-hcwiid-dev: copyright-file-contains-full-gpl-license
E: libghc-hcwiid-dev: copyright-should-refer-to-common-license-file-for-lgpl
E: libghc-hcwiid-dev: description-contains-tabs
E: libghc-hcwiid-prof: copyright-file-contains-full-gpl-license
E: libghc-hcwiid-prof: copyright-should-refer-to-common-license-file-for-lgpl
E: libghc-hcwiid-prof: description-contains-tabs
E: libghc-hcwiid-doc: copyright-file-contains-full-gpl-license
E: libghc-hcwiid-doc: copyright-should-refer-to-common-license-file-for-lgpl
E: libghc-hcwiid-doc: description-contains-tabs
Finished running lintian.
$ ls ../*hcwiid*deb
../libghc-hcwiid-dev_0.0.1-1~hackage1_amd64.deb
../libghc-hcwiid-doc_0.0.1-1~hackage1_all.deb
../libghc-hcwiid-prof_0.0.1-1~hackage1_amd64.deb
```

なんかあっさり Debian パッケージができちゃいました。 lintian がなんか言ってますが、あまり深刻なものではないのでとりあえずインストールしてみましょう。

```
$ sudo dpkg -i ../libghc-hcwiid-dev_0.0.1-1\~hackage1_amd64.deb \
../libghc-hcwiid-doc_0.0.1-1\~hackage1_all.deb ../libghc-hcwiid-prof_0.0.1-1\~hackage1_amd64.deb
$ cd ~/
$ rm -rf .ghc .cabal # これで cabal でインストールしたパッケージは一切使っていないはずです
$ ghc-pkg list|grep hcwiid
    hcwiid-0.0.1
```

Hackage はインストール済みのようです。 hcwiid ライブラリを使ってみましょう。

Test.hs

```
module Main where

import Prelude
import Control.Monad
import System.CWiid
import System.Posix.Unistd

main :: IO ()
main = do
  putStrLn "Put Wiimote in discoverable mode now (press 1+2)..."
  (Just wm) <- cwiidOpen
  putStrLn "found!"
  _ <- cwiidSetLed wm
  _ <- cwiidSetRptMode wm
  _ <- forever $ do _ <- usleep 300000
                    cwiidGetBtnState wm >>= print
  return () -- not reach
$ ghc --make Test.hs
[1 of 1] Compiling Main             ( Test.hs, Test.o )
Linking Test ...
$ ./Test
Put Wiimote in discoverable mode now (press 1+2)...
```

なんて簡単なんでしょう!簡単な Hackage なら cabal-debian コマンドを使えば Debian パッケージ化が完了してしまうようです。しかも下記 3 つのライブラリに分割してくれています。やった!

- libghc-HOGE-dev - 通常使用するライブラリ
- libghc-HOGE-doc - Haddock で生成された API ドキュメント
- libghc-HOGE-prof - プロファイラ対応ライブラリ

7.6 haskell-debian-utils のしくみ

cabal-debian での Debian パッケージ化はどのようなしくみなのでしょうか。さきほど作った hcwiid パッケージの debian/rules ファイルを見てみましょう。

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

# How to install an extra file into the documentation package
#binary-fixup/libghc-hcwiid-doc::
#       echo "Some informative text" > debian/libghc-hcwiid-doc/usr/share/doc/libghc-hcwiid-doc/AnExtraDocFile
```

なんということでしょう。内容がありません。。。これは hlibrary.mk ファイルに秘密があるに相違ありません。全部を読まずにまずは libghc-HOGE-dev の build ターゲットとその近辺を hlibrary.mk から抜き出してみましょう。

```
DEB_SETUP_BIN_NAME ?= debian/hlibrary.setup
BUILD_GHC := $(DEB_SETUP_BIN_NAME) build

$(DEB_SETUP_BIN_NAME):
        if test ! -e Setup.lhs -a ! -e Setup.hs; then echo "No setup script found!"; exit 1; fi
        for setup in Setup.lhs Setup.hs; do if test -e $$setup; then ghc --make $$setup -o $(DEB_SETUP_BIN_NAME); \
          exit 0; fi; done

build/libghc-$(CABAL_PACKAGE)-prof build/libghc-$(CABAL_PACKAGE)-dev:: build-ghc-stamp

build-ghc-stamp: dist-ghc
        $(BUILD_GHC) --builddir=dist-ghc
        touch build-ghc-stamp
```

なるほど。 libghc-HOGE-dev を build しようとすると、まず Setup.lhs もしくは Setup.hs を ghc を使ってコンパイルして `debian/hlibrary.setup` コマンドを作成するようです。そうして作った debian/hlibrary.setup コマンドを使って”debian/hlibrary.setup build –builddir=dist-ghc” のようにして dist-ghc ディレクトリ上で Hackage をコンパイルするんですね。

ちょっと脱線しますが、このビルドプロセスは cabal が普段やっていることと全く同じです。 cabal はインストール対象の Hackage を取得/展開したら、まずこの Setup.hs を ghc でコンパイルして、そのコンパイルした結果できた実行バイナリを本当のビルダ/インストーラとして使います。普段使っている/usr/bin/cabal コマンドは”cabal-install”と呼ばれています。そして、 Setup.hs を書くために必要なライブラリを”Cabal”と呼びます。ややこしいですね。。。

では libghc-HOGE-dev の install はどうなっているのでしょうか?

```
debian/tmp-inst-ghc: $(DEB_SETUP_BIN_NAME) dist-ghc
        $(DEB_SETUP_BIN_NAME) copy --builddir=dist-ghc --destdir=debian/tmp-inst-ghc

install/libghc-$(CABAL_PACKAGE)-dev:: debian/tmp-inst-ghc debian/extra-depends
        cd debian/tmp-inst-ghc ; find usr/lib/haskell-packages/ghc/lib/ \
                \( ! -name "*_p.a" ! -name "*.p_hi" \) \
                -exec install -Dm 644 '{}' ../$(notdir $@)/'{}' ';'
        pkg_config=`$(DEB_SETUP_BIN_NAME) register --builddir=dist-ghc --gen-pkg-config | sed -r 's,.*: ,,'`; \
                $(if $(HASKELL_HIDE_PACKAGES),sed -i 's/^exposed: True$$/exposed: False/' $$pkg_config;) \
                install -Dm 644 $$pkg_config debian/$(notdir $@)/var/lib/ghc/package.conf.d/$$pkg_config; \
                rm -f $$pkg_config
        if [ 'z$(DEB_GHC_EXTRA_PACKAGES)' != 'z' ] ; then \
                echo '$(DEB_GHC_EXTRA_PACKAGES)' > \
                debian/$(notdir $@)/usr/lib/haskell-packages/ghc/lib/$(CABAL_PACKAGE)-$(CABAL_VERSION)/extra-packages ; \
        fi
        dh_haskell_provides -p$(notdir $@)
        dh_haskell_depends -p$(notdir $@)
        dh_haskell_shlibdeps -p$(notdir $@)
```

ちょっとわかりにくいですが、パッケージ化の後半は Debian 流儀の詳細なので踏みこまずに解釈すると、まず libghc-HOGE-dev を install しようとすると、 debian/tmp-inst-ghc ターゲットが呼び出されて”debian/hlibrary.setup copy –builddir=dist-ghc –destdir=debian/tmp-inst-ghc” のようなコマンドが実行されて、 dist-ghc でコンパイルした内容が debian/tmp-inst-ghc 以下にインストールされます。あとは、 Debian の流儀にのっとって debian/tmp-inst-ghc 以下のファイル群をパッケージ化するだけです。パッケージ化対象の Hackage が依存している Hackage も dh_haskell_shlibdeps でちゃんと検出してくれるみたいです。 :)

7.7 作ったパッケージを Debian に登録するには

せっかく作った Hackage です。自分だけで使っているのはもったいないです。 Debian 本家に登録して皆に使ってもらいましょう! Debian 本家に登録しておけばめぐりめぐって Ubuntu にも登録されるかもしれませんよ?

8 Emacs, Vim の拡張機能で学ぶ Debian パッケージ

西田孝三

8.1 はじめに

パッケージメンテナになることで Debian に関わりたいと思われている方は多いのではないでしょうか。もしあなたが Emacs, Vim のユーザであればこれらの機能拡張でパッケージを作成するところから始めてみるのはどうでしょうか。理由は、

- アーキテクチャに依存しない
- コンパイルは不要 (Emacs の場合バイトコンパイルがありますが)

などから比較的簡単と思われるためです。

ここでは大きく分けて下記のことを行い、まずは簡単な Debian パッケージを自分で作成できるようになるまでを目的としています。

- Emacs の Debian パッケージ
  - 既存の Debian パッケージの構成を知り再構成を行う
  - 独自の Debian パッケージを作成する
- Vim の Debian パッケージ
  - 既存の Debian パッケージの構成を知る

Vim では構成だけを学び、独自の Debian パッケージの作成を行いません。その理由は後に説明します。それでは Emacs の既存の Debian パッケージの構成を知り再構成を行うところから始めましょう。

8.2 Emacs の Debian パッケージ

8.2.1 Emacs の既存 Debian パッケージのソース取得と再構成

ここでは前回の関西 Debian 勉強会で発表されていた山下尊也さんがメンテナンスをされている auto-install-el のパッケージのソースを取得し、再構成を行います。

```
$ mkdir tmp; cd tmp
$ apt-get source auto-install-el
$ ls auto-install-el-1.48
```

これで tmp ディレクトリ内に auto-install-el のソース (バージョン 1.48) が取得できているはずです。ソース内容を確認することは保留し、いきなり Debian パッケージを作ってみましょう。

```
$ cd auto-install-el-1.48
$ debuild -us -uc
$ ls ..
```

これで debuild をしたディレクトリのひとつ上に auto-install の Debian パッケージができます。それではこの Debian パッケージをインストールしてみましょう。

```
$ cd ..
$ sudo dpkg -i auto-instlal-el_1.48-1_all.deb
$ aptitude show auto-install-el
```

これで auto-install-el がインストールされたことがわかります。それではうまく使えるかどうか Emacs で確認してみましょう。

```
$ emacs -nw
M-x load-library
auto-install
auto-install-from-emacswiki
grep-edit.el
```

これで /.emacs.d/auto-install/ 下に grep-edit.el(elc) がインストールされていることが確認できます。

それでは保留していたソースの構成の確認に戻りましょう。が、その前にもう一つ別のディレクトリに auto-install-el のソースを取得しましょう。というのは debuild 後にはいくつかのファイルが生成されているからです。 2 つのソースディレクトリを比較することでこれらのファイルが確認できます。詳細はご自身でご確認ください。

```
$ cd; mkdir auto-install-el
$ cd auto-install-el; apt-get source auto-install-el
```

それでは debuild する前の auto-install-el-1.48 内のファイル構成を見てみましょう。 auto-install.el というファイルと debian というディレクトリがあります。このことから Emacs の拡張機能の Debian パッケージを作るには

- Emacs の拡張機能にバージョン名を加えた名前のディレクトリを作り
- その下に Emacs Lisp と debian というディレクトリを作ればよい

ということがわかります。それではこれをまだ Debian パッケージが作られていない Emacs の拡張機能に対して行っていきましょう。

8.2.2 Emacs の新規 Debian パッケージの作成

今回は青田直大さんによる twinstall.el のパッケージを作ってみましょう。これは twittering-mode という Emacs 用 twitter クライアントの機能を使って、 auto-install.el でインストールした Emacs Lisp 名をつぶやくものです。現時点では ID が 1300477 の gist(https://gist.github.com/1300477) から取得できます。

```
$ tar zxvf gist1300477.tar.gz
$ mv gist1300477 twinstall-el-0.1
$ tar czvf twinstall-el-0.1.tar.gz twinstall-el-0.1
$ cd twinstall-el-0.1
$ cat >>~/.bashrc <<EOF
 DEBEMAIL=''your.email.address@example.org''
 DEBFULLNAME=''Firstname Lastname''
 export DEBEMAIL DEBFULLNAME
 EOF
$ . ~/.bashrc
$ dh_make -f ../twinstall-el-0.1.tar.gz
```

次にどのような種類のパッケージを作るか聞かれるので single の s にし enter を押します。すると debian ディレクトリとその下に各種設定ファイルの雛形が生成されます。多くのファイルができあがりますが、 auto-install-el と同じものがあればよいので対応するファイルは削除し、後は auto-install-el を真似てファイル内容を変更していきましょう。(変更せずにこの段階で debuild を行っても一応パッケージはできます。) それでは各ファイルの説明をします。

■README.Debian

- Debian パッケージの README
- 必須ではない?

■changelog

- 必須のファイル
- Debian のポリシーで規定された書式

■compat

- わかりませんでした!
- 作られた雛形のままにしましょう

■control

- 必須のファイル
- aptitude などのパッケージ管理ツールが利用する情報
- Debian のポリシーで規定された書式
- twinstall.el は auto-install と twittering-mode に依存している。 Depends:に追加

■copyright

- 必須のファイル
- upstream ソースに関する著作権やライセンスなどの情報を書く
- Debian のポリシーで規定された書式

■dirs

- ファイルをインストールするディレクトリを書く
- 前述ディレクトリのパスのトップの/は書かない

■emacsen-install, emacsen-remove

- install, uninstall 時に行う処理をやってくれるシェルスクリプト

■emacsen-startup

- elisp のインストールディレクトリに dirs で書いたインストール先を Emacs の load-path に追加してくれる elisp
- これ自体は/etc/emacs/site-start.d/に 50’ パッケージ名’.el という名でインストールされる

■rules

- 必須のファイル
- パッケージを作成するために使うルールを書く
- まずはとりあえず人様のものを真似る

■source

- わかりませんでした!
- 作られた雛形のままにしましょう

```
$ debuild -us -uc
$ cd ..
$ sudo dpkg -i twinstall-el_0.1-1_all.deb
```

control の Depends 以外は auto-install-el を真似て新規パッケージに応じた情報に置き換えることで twinstall の Debian パッケージが出来ます。後はこの Debian パッケージを公開する必要があるかを考え、ライセンスなどを学び、公開に必要な次のステップへ進んでください。もちろんコンパイルが必要なパッケージ作成へとレベルアップするのもよいでしょう。

8.3 Vim の既存パッケージのソース取得

Vim の拡張機能の Debian パッケージは Emacs と比較すると数は少なく、あまりパッケージを作るモチベーションが得られなさそうです。そのため Vim に関してはパッケージ作成までは行いません。実際に Rails 用の Vim script パッケージのソースを取得しその内容をみてみましょう。

```
$ apt-get source vim-rails
```

これで取得できるソースの README.Debian と control を見ると vim-addon-manager というパッケージを使い vim-addons というコマンドで vim-rails を使用可能にするということを行なっていることがわかります。これでは Debian のパッケージマネージャーと Vim の addon-manager で manager を 2 つ用いていることになり、あまり良いこととは思えません。現在の Vim ユーザの多くは vim-addon-manager より bundle や neobundle といった Vim script で書かれた manager を用いており、これらの完成度が高いため Debian のパッケージは不要なのかもしれません。

9 月刊 debhelper 第1回

岩松 信洋

9.1 debhelper とは何か?

Debian パッケージを作成する時、パッケージに必要なファイルのチェック、コンパイル前の設定、コンパイルなど様々な処理を行う必要があります。 Debian パッケージでは debian/rules という GNU Make の makefile に各処理を記述するのですが、細かい処理をひとつづつ書いていくと膨大な量になります。

またコードの量が多くなるとバグも多くなり、パッケージ作成時に問題が起きたときに修正するのは大変です。これらの処理を機能毎にまとめ、使いやすくした機能を提供しているパッケージとして debhelper があります。他にも同様のツールがいくつかありますが、 1 番使われているのがこの debhelper です。 Debian パッケージをメンテナンスしている人にとって debhelper の知識が必須と言ってもいいでしょう。

ちなみに debhelper は Debian 開発者の Joey Hess 氏[*40]によって開発/メンテナンスされ、最新のバージョンは 8.9.8 となっています。

9.2 月刊 debhelper とは?

先にも説明したように、 Debian パッケージをメンテナンスしている人にとって debhelper の知識が必須となっています。 debhelper がどのような機能を提供して、それらをどのように使えばいいのか、どのように使われているのか、理解しておく必要があります。現時点で debhelper では 59 個のコマンド(dh_で始まるコマンド) が提供されており、全部理解するのは難しいでしょう。また、 debhelper に収録されていない debhelper サポートツールを含めると 100 個ほどになります。日頃 Debian の開発を行なっている人でも「 ああ、こんな機能があるのだ」と思うことがあるぐらいです。更に debhelper 7 からコマンドがいくつか増え、 debian/rules ファイルが以下のように記述できるようになりました。

これだけでは何をやっているのかさっぱり分かりません。細かい指定を行いたい場合、どのようにしたらいいのかすらわからない状態です。

そこで debhelper で提供されているコマンドの動きと使い方を毎月数個づつ紹介し、 Debian 勉強会参加者でパッケージ作成の理解を深める企画、「 月刊 debhelper」を企画しました。全て理解した頃には、皆 Debian パッケージメンテナになっているかもしれません。ヒャッハー!

[*40] Wiki エンジンの ikiwiki, ディストリビューションのパッケージ間変換ツールである alien の開発者として有名。

```
debhelper 6:
#!/usr/bin/make -f

build: build-stamp
build-stamp:
    dh_testdir
    $(MAKE)
    touch $@

clean:
    dh_testdir
    dh_testroot
    $(MAKE) clean
    dh_clean

install: build
    dh_testdir
    dh_testroot
    dh_clean -k

binary-indep:

binary-arch: build install
    dh_testdir
    dh_testroot
    dh_installchangelogs ChangeLog
    dh_installd
.....
```

```
debhelper 7:
#!/usr/bin/make -f
%:
        dh $@
```

9.3 debian パッケージ構築、全体の流れ

いきなり個々のコマンド説明をしてもよくわからないので、パッケージ作成の全体の流れとどのようなコマンドが呼び出されるのか説明します。Debian パッケージが作成される簡単流れは以下の通りで、図にすると図 9.3 のようになります。

1. パッケージビルド環境を構築する
   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構築する必要があります。ここでは、ソースコードの展開、パッケージ構築依存のチェック等を行います。
2. 不要なファイルを削除する
   次にパッケージに不要なファイルを削除します。例えば、前に行われたパッケージ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、ソースが展開された常に同じ状態からビルドできるようにします。
   これは debian/rules ファイルの **clean** ターゲットで行われ、このターゲットは「不要なファイルを削除する」ことを目的とするように Debian Policy で定められています。
   また clean ターゲットでは、以下の dehhelper コマンドが実行されます。

   ```
   dh_testdir -> dh_auto_clean -> dh_clean
   ```

3. バイナリパッケージに格納するファイルをビルドする
   次にソースコードからバイナリをビルドします。ここでは configure などを使ったコンパイル前の設定、コンパイラを使った実行ファイルの作成、ドキュメントの変換などがおこなわれます。
   これは debian/rules ファイルの **build** ターゲットで行われ、このターゲットは「プログラムの設定、コンパイルやデータの変換」ことを目的とするように Debian Policy で定められています。
   また build ターゲットでは、以下の dehhelper コマンドが実行されます。

   ```
   dh_testdir -> dh_auto_configure -> dh_auto_build -> dh_auto_test
   ```

4. ビルドしたファイルをバイナリパッケージにまとめる
   必要なファイルをすべてビルド完了した後、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ます。
   ここでは、 debian/tmp を/(ルート) 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、各ファイルのパーミッションの設定やファイルの圧縮など、行わなければなら

ないことや、推奨されていることが多数あります。

これは debian/rules ファイルの **binary、 binary-arch、 binary-indep** ターゲットで行われ、このターゲットは「 バイナリパッケージとしてまとめる」ことを目的とするように Debian Policy で定められています。

また binary、 binary-arch、 binary-indep ターゲットでは、以下の dehhelper コマンドが実行されます。

```
dh_testdir -> dh_auto_configure -> dh_auto_build -> dh_auto_test -> dh_testroot
-> dh_prep -> dh_installdirs -> dh_auto_install -> dh_install -> dh_installdocs
-> dh_installchangelogs -> dh_installexamples -> dh_installman -> dh_installcatalogs
-> dh_installcron -> dh_installdebconf -> dh_installemacsen -> dh_installifupdown
-> dh_installinfo -> dh_pysupport -> dh_installinit -> dh_installmenu -> dh_installmime
-> dh_installmodules -> dh_installlogcheck -> dh_installlogrotate -> dh_installpam
-> dh_installppp -> dh_installudev -> dh_installwm -> dh_installxfonts -> dh_installgsettings
-> dh_bugfiles -> dh_ucf -> dh_lintian -> dh_gconf -> dh_icons -> dh_perl -> dh_usrlocal
-> dh_link -> dh_compress -> dh_fixperms -> dh_strip -> dh_makeshlibs -> dh_shlibdeps
-> dh_installdeb -> dh_gencontrol -> dh_md5sums -> dh_builddeb
```

5. .changes ファイルを作成する

   パッケージが作成されたら、そのパッケージの.changelog ファイルを作成します。このファイルの作成には dpkg-genchanges コマンドが使われます。このコマンドは debhelper のコマンドではありません。

6. パッケージに署名する

   必須ではありませんが、パッケージができたら.dsc ファイルと.changes ファイルに GPG/PGP を使って署名をします。この署名には debsign コマンドを使います。このコマンドは debhelper のコマンドではありません。

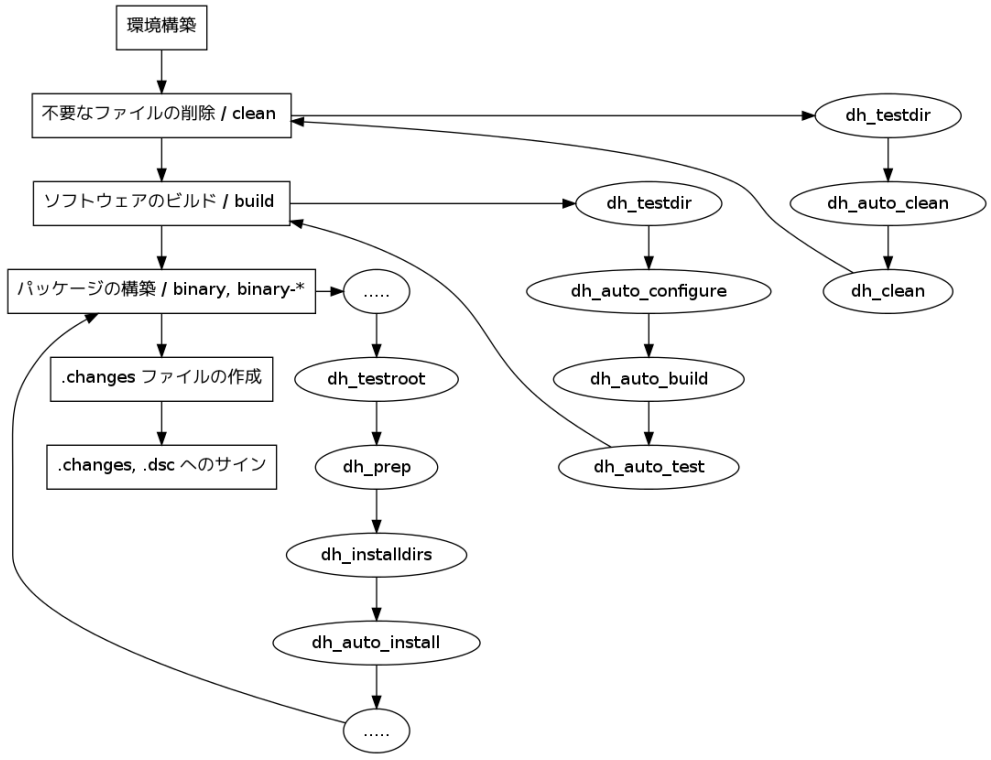

図 8　各処理と debhelper コマンドの関係図

9.4　その他 debhelper の重要な機能

9.4.1　環境にあわせたシーケンス情報を読み込む

debhelper は特定の言語や環境に合わせたシーケンスを定義し、読み込ませることによって、 makefile 内で利用できるターゲットとコマンドを増やすことができます。例えば、パッチ管理ツールである quilt を使ったターゲットは debhelper には含まれていません。使いたい場合には、**–with** オプションを使って指定します。

```
%:
	dh $@ --with quilt
```

指定することによって、 **dh_quilt_patch** が利用できるようになります。

9.4.2 各 debhelper コマンドの動きを変更する

上記で説明しように、 debhelper では各ターゲットと各コマンドの動作が予め決められています。これらを変更するには各コマンド用のターゲットに対して動作を記述します。このターゲットは **override_各 debhelper コマンド**となっており、 dh_auto_configure（ 決められた値で自動的に configure を実行するためのコマンド）の場合には以下のように使います。

```
override_dh_auto_configure:
    dh_auto_configure -- --enable-foo
```

9.5 今月のコマンド： dh_testdir

9.5.1 概要

パッケージビルドを行うときに正しいディレクトリにいるかチェックします。

9.5.2 使い方

dh_testdir コマンドはカレントディレクトリに debian/control があることによって正しいディレクトリにいるかチェックをしています。 dh_testdir はほとんどのターゲットから利用されます。ちゃんと debian パッケージをビルドできる場所にいるかチェックするためです。

```
$ mkdir foo
$ cd foo
$ dh_testdir
dh_testdir: cannot read debian/control: そのようなファイルやディレクトリはありません
echo $?
2
$ mkdir debian
$ touch debian/control
$ dh_testdir
$ echo $?
0
```

引数としてファイルパスを指定することができます。ファイルパスを指定した場合には、指定したファイルによってチェックが行われます。

```
$ touch moo
$ dh_testdir moo
$ echo $?
0
```

9.6 今月のコマンド： dh_bugfiles

9.6.1 概要

dh_bugfiles コマンドは バグレポートに必要なファイルをパッケージに格納します。バグレポートに使うファイルは script、 control、 presubj の 3 つがあり、 debian/bug ディレクトリに格納されている必要があります。各ファイルの用途を以下に説明します。

- script
  バグレポート用のスクリプトです。バグレポートを行うためのツール reportbugs 等でレポート作成時に呼び出し、結果をバグレポートの一部として追記します。例えば、 X.Org のドライバ群は /usr/share/bug/xserver-xorg-core/script にシンボリックリングを張ったファイルをバイナリパッケージ内に持ちます。このスクリプトでは、reportbug を実行した環境のカーネルバージョンや dmesg, xorg のログなどが自動的に出力されるようになっています。

- control

  control ファイルは指定したコマンドの結果をバグレポートの一部として出力します。コマンドには以下の 4 つがあります。

  - package-status 指定したパッケージのステータス(インストール状態、バージョン）をバグレポートに追加します。

    設定例:
    ```
    /usr/share/bug/mutt/control package-status: mutt mutt-patched mutt-dbg
    ```

  - report-with 指定したパッケージ情報をバグレポートに追加します。

    設定例:
    ```
    /usr/share/bug/xorg/control report-with: xserver-xorg
    ```

  - Send-To Debian BTS 以外に自動的にが行われるメールアドレスを設定します。

    ```
    Send-To: foo@example.org
    ```

  - Submit-As: 一つのパッケージにレポートが行われるようにコントールする以下のように設定した場合、linux-image-3.0.0-2-amd64 にバグレポートした場合には linux-2.6 に行われるように自動的に変更されます。

    ```
    control:Submit-As: linux-2.6
    ```

- presubj

  レポートする前の警告文を出すために使います。例えば、 gnupg パッケージの場合にはこのファイルに

  ```
  Please consider reading /usr/share/doc/gnupg/README.BUGS.Debian before
  sending a bug report. Maybe you'll find your problem there.
  ```

  と書くことによって、バグレポートを送る前に /usr/share/doc/gnupg/README.BUGS.Debian を参照するよう、誘導しています。

  reportbug を使って、 gnupg パッケージにバグレポートしようとしたとき、以下のようなメッセージが表示されます。

  ```
  Please consider reading /usr/share/doc/gnupg/README.BUGS.Debian before
  sending a bug report. Maybe you'll find your problem there.


  (You may need to press 'q' to exit your pager and continue using
  reportbug at this point.)
  ```

これらのファイルは一つだけでもかまいません。

9.6.2 使い方

このコマンドは install ターゲットで使用します。

```
install:
    ....
    dh_bugfiles
    ....
```

10 aufsbuilder - cowbuilder にたたかいをいどむ

やまだ

10.1 やってみた

数年前に aufs がマイブームだった時、

> これで aufsbuilder 書いたら cowbuiler に勝てるんじゃね?

と思ってやってみたものの、僅差ながら負けてお蔵入りしていた aufsbuilder がこのほど勝利したので報告します。

10.2 つくりかた

実は pbuilder は chroot 環境に任意の場所を指定することができ、

```
pbuilder $PBCMD --no-targz --buildplace <適当な chroot 先>
```

と呼び出してやるだけで、よろしくビルドしてくれます。なので、これを呼び出す前にテンプレート用 chroot 環境に書き込み用使い捨てフォルダをラップした aufs chroot を作ってやればいいわけです。

```
#!/bin/sh -e

: ${PB_BASE=/var/cache/pbuilder}
: ${PB_WORK=/var/cache/pbuilder/build}

usage() {
  P=$(basename $0)
  test $# -gt 0 && echo $@ >&2
  cat <<EOF 1>&2
$P - pbuilder wrapper with aufs-wrapped chroot
Usage: $P ...pbuilder-args...
Note:
- You need to define PB_BASE and PB_WORK
- For base chroot tree, \$PB_BASE/\$ARCH-\$DIST.cow/ will be used.
- For actual work tree, \$PB_WORK/\$\$/ will be used.
- Default: PB_BASE=$PB_BASE, PB_WORK=$PB_WORK
EOF
  exit 1
}

# pass all args to pbuilder (0 args == help)
test $# -gt 0 || usage
test $# -gt 0 && PBCMD="$1"; shift
test $# -gt 0 && PBARG="$@"

# prepare env
: ${DIST:=sid}
: ${ARCH:=$(dpkg-architecture -qDEB_HOST_ARCH)}
export ARCH DIST

MT="$PB_WORK/$$"
RW="$PB_WORK/$$/rw"
AD="$PB_BASE/$DIST-$ARCH"
RO="$PB_BASE/$DIST-$ARCH.cow"
```

```
# sanity check
test -d "$MT" && usage "ERROR: Workdir already exists: $MT"
test -d "$RW" && usage "ERROR: Workdir already exists: $RW"
test -d "$RO" || usage "ERROR: Missing template: $RO"

# register cleanup hook
trap "
$DEBUG umount -lf '$MT/var/cache/apt' '$MT' && $DEBUG rm -fr '$MT'
" 0 1 2 3 4 6 7 8 11 15

# prepare chroot tree
$DEBUG mkdir -p "$RW" "$AD/aptcache"
$DEBUG mount -t aufs -o "br:$RW:$RO=ro" none "$MT"
$DEBUG mount --rbind "$AD/aptcache" "$MT/var/cache/apt"

# run pbuilder
$DEBUG pbuilder $PBCMD --aptcache "" --no-targz --buildplace "$MT" "$@"
```

10.3 たたかってみる

pbuilder と cowbuilder の関係同様、 aufsbuilder も pbuilder 互換なのでそのまま

```
$ PDEBUILD_PBUILDER=aufsbuilder \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
```

としてビルドするだけです。では、比較してみましょう。

まずは素の pbuilder：

```
$ sudo rm ../cocot_20100903-1*
$ time sudo ARCH=i386 DIST=sid \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
$ time sudo ARCH=i386 DIST=sid \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
    0m44.76s real     0m21.50s user     0m13.62s system
```

続いて cowbuilder：

```
$ sudo rm ../cocot_20100903-1*
$ time sudo ARCH=i386 DIST=sid PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
$ time sudo ARCH=i386 DIST=sid PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
    0m33.17s real     0m20.42s user     0m8.84s system
```

そして aufsbuilder:

```
$ sudo rm ../cocot_20100903-1*
$ time sudo ARCH=i386 DIST=sid PDEBUILD_PBUILDER=aufsbuilder \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
$ time sudo ARCH=i386 DIST=sid PDEBUILD_PBUILDER=aufsbuilder \
git-buildpackage --git-builder=''pdebuild --buildresult ..''
    0m29.41s real     0m18.46s user     0m8.17s system
```

前回は何回やっても数秒差で負け続けたので、逆転できて嬉しい。たぶん LKML の小人さん達が頑張ってくれたおかげ mOm

10.4 まとめ

aufs を使った cowbuilder を作ってみました。以前作った時はどうやっても勝てなかったのに、いつのまにか速くなっていて嬉しい。

10.5 だがしかし・・・

```
$ sudo rm ../cocot_20100903-1*
$ debuild --no-lintian -us -uc -Tclean
$ time git-buildpackage --git-builder=''DEB_CFLAGS_APPEND=-m32 debuild -ai386''
$ time git-buildpackage --git-builder=''DEB_CFLAGS_APPEND=-m32 debuild -ai386''
    0m13.95s real     0m12.17s user     0m4.74s system
```

ネイティブビルド、やっぱり速い。しかも lintian と debsign 時間まで入ってるし。

11 vcs-buildpackage ～Git、 svn 編～

佐々木 洋平

11.1 始めに

今日のお題は、パッケージ作成に Git や Subversion を使用するソフトウェアである、 `git-buildpackage` と `svn-buildpackage` です。自分の抱えている野良パッケージの多くが Ruby 関連だったこともあって、 PkgRubyExtras に参加したところ、パッケージ管理を Subversion から Git へ移行するタイミングだった様で、 (幸か不幸か) 両 VCS を使用してのパッケージ作成を体験しました[*41]。

そこで覚えたツールの使い方とか、実際に作業する際のハマり所とかについて簡単に紹介できたら良いな、とか思います。とはいえ、実際にはコラボする人々 (=チーム) 毎に work flow があるので、あまり一般論は言えない訳ですけれど。

11.2 バージョン管理?

バージョン管理システム (VCS) についてはほとんど説明しませんが、簡単に。

VCS を使った事のない人に VCS の利点を説明する時、佐々木は良く「 良い感じのバックアップ」という言い方をしています。例えば

- 過去の変更履歴を残しておける
- 過去の任意の状態に簡単に戻せる
- 過去にどんな変更を行なったか、を把握しやすくする (ログをきちんと書いておく必要はありますが)

また、複数人で開発を進めている時には、同じファイルに同じ様な変更を加えている場合、最後に保存した人の変更だけが残ってしまい (上書きされてしまい)、それ以外の変更が失なわれてしまいます。 VCS を使っていると、同じファイルへの変更は「 衝突」として検出されるので、こういった事態を防ぐ事ができます。

さらに、特定のバージョンに名前をつけておき (tag と言います) そのバージョンへの変更を行なったり、新機能の開発を本番の開発と分離して進めて (本番が main line と呼ばれるのに対して、 branch を分ける、 branch を切る、と言います)、完成した後で本番の開発に統合したりできます。

Subversion と Git は、それぞれ「 集中型 VCS」と「 分散型 VCS」の代表です。集中型 VCS では単一のリポジトリ (データ置き場) を開発者全員が参照し作業するのに対して、分散型 VCS では、各人が個別にリポジトリを持ち、各々のデータを適宜同期を取って作業を進めます。最近は分散型 VCS(DVCS と言ったりします) がトレンドですね[*42]。

[*41] 自分一人だったら絶対 Git で作業するんですけれどね

[*42] 集中型 VCS としては CVS なんかもありますが、今更 CVS ってのは止めた方が良いと思います。また DVCS としたは Git 以外に darcs, bzr, Mercurial なんかがあります。 bzr についてはそのうち山下 尊也さんが語ってくれると思います

11.3 パッケージのバージョン管理?

さて、「パッケージのバージョン管理」って何をしているのでしょうか? Debian パッケージを「バージョン管理」する目的は幾つかありますが、例えば

- 履歴の管理。パッケージの変更点は debian/changelog に書かれます (書きます) が、実際にファイル自体を参照できると大変便利です。
- stable に含まれたパッケージにタグを付けておく。
  - セキュリティ対応などを、そのタグから派生するブランチで対応する
- 複数人でのパッケージメンテ/チームでのパッケージメンテ作業の環境を容易に構築できる
- upstream が VCS を使用している場合の連携が簡単になる (かも)

... でしょうか。

11.4 何を管理するの?

Debian のソースパッケージは、 source format 3.0 では以下のファイル群で構成されます:

.orig.tar.*ext*
: upstream のソース。複数のソースからなる場合には基本となるソースにこの名前をつける。 *ext* は圧縮の拡張子であり、 gz, bz2, xz が使用可能。

.orig-*component*.tar.*ext*
: upstream が複数のソースから構成される場合のファイル名。 `dpkg-source -x` **パッケージ名**`.dsc` などで展開するとファイルの中身は `component/`以下の展開される。

.debian.tar.*ext*
: `debian/` ディレクトリの中身

.dsc
: パッケージの情報。上記ファイル群のハッシュなどを記載している。

パッケージを VCS で管理する場合、

1. `.orig.tar.`*ext*
2. `.debian.tar.`*ext*

のバージョンを管理することになります。

11.5 どうするの?: svn-buildpackage 編

では実際にどうやっているのでしょうか? ここでは GNU Hello を例に、 svn-buildpackage を使った場合について簡単に述べてみます。なお、佐々木は既に svn-buildpackage をあまり使っていないので、 Git はまだ良くわからないけれど Subversion ならわかる、 という人向けのお話です。

11.5.1 インストール

```
$ sudo aptitude install svn-buildpackage
```

11.5.2 パッケージをリポジトリに追加する

一度適当なパッケージを作成してみましょう。 ここでは GNU hello のパッケージが作成できたとします。横着したい人は apt-get source hello でも良いでしょう。

テストのために簡単なリポジトリを作成します。ここでは /Work/svn/ 以下にリポジトリを作成します。

```
$ svnadmin create ~/Work/svn/
$ svn mkdir file:///home/uwabami/Work/svn/trunk -m "create trunk"
$ svn mkdir file:///home/uwabami/Work/svn/tags -m "create tags"
$ svn mkdir file:///home/uwabami/Work/svn/branches -m "create branches"
$ svn ls file:///home/uwabami/Work/svn
 branches/
 tags/
 trunk/
```

次に、作成したパッケージをリポジトリに追加します。追加するコマンドは svn-inject です。

```
$ svn-inject -l 2 -o -c 0 hello_2.7-1.dsc file:///home/uwabami/Work/svn
```

svn-inject のオプションの詳細は man を見てもらうとして、ここでは

```
     -l
         Layout type.
         1 (default) means package/{trunk,tags,branches,...} scheme,
         2 means the {trunk,tags,branches,...}/package scheme.
     -o
         Only keep modified files under SVN control (including the debian/ directory),
         track only parts of upstream branch

      -c number
         Checkout nothing (0), trunk directory (1) or everything (2) when
         the work is done.
```

を用いています. 単一のパッケージを単一のリポジトリで管理する場合には -l 1 が良いでしょう.

11.5.3 パッケージのビルド

svn-inject が終わったので、リポジトリからファイルを取得してみます。

```
$ svn checkout file:///home/uwabami/Work/svn/trunk svn-work
 A    trunk/hello
 A    trunk/hello/debian
 A    trunk/hello/debian/control
 A    trunk/hello/debian/source
 A    trunk/hello/debian/source/format
 ...
```

ここでは svn-work にファイルをチェックアウトしました。実際に svn-work/trunk/hello に移動して、パッケージを作成してみます。

```
$ cd svn-work/trunk/hello
$ sudo apt-get build-dep hello
$ svn-buildpackage
 Complete layout information:
         trunkDir=/home/uwabami/svn-work/trunk/hello
         trunkUrl=file:///home/uwabami/Work/svn/trunk/hello
 dpkg-checkbuilddeps
 Orig tarball not found (expected ../tarballs/hello_2.7.orig.tar.gz)
 mergeWithUpstream mode detected, looking for ../tarballs/hello_2.7.orig.tar.gz
 I: mergeWithUpstream property set, looking for upstream source tarball...
 E: Could not find the upstream source file! (should be ../tarballs/hello_2.7.orig.tar.gz)
```

... 転びました。 svn-buildpackage では、パッケージ作成時の一時ディレクトリ (build-area) と upstream のソースの保管場所 (tarballs) の存在を仮定しています。これを準備しましょう (ちなみに svn-buildpackage が失敗した時点で、これらのディレクトリが既に作成されています)。

```
$ cd ..
$ ls
 build-area/  hello/  tarballs/
$ cd hello
$ uscan --download-current-version --destdir=../tarballs/
```

watch ファイルがきちんと書かれている/書けていると、 uscan 一発で良いので楽です。ではもう一度パッケージをビ

ルドしてみます。

```
$ svn-buildpackage
 Complete layout information:
       buildArea=/home/uwabami/svn-work/trunk/build-area
       origDir=/home/uwabami/svn-work/trunk/tarballs
       trunkDir=/home/uwabami/svn-work/trunk/hello
       trunkUrl=file:///home/uwabami/Work/svn/trunk/hello
 ...
 dpkg-deb: '../hello_2.7-1_amd64.deb' にパッケージ 'hello' を構築しています。
 signfile hello_2.7-1.dsc
 gpg: " Santiago Vila <sanvila@debian.org> "をとばします: 秘密鍵が得られません
 gpg: [stdin]: clearsign failed: 秘密鍵が得られません

 dpkg-genchanges  >../hello_2.7-1_amd64.changes
 dpkg-genchanges: including full source code in upload
 dpkg-source --after-build hello-2.7
 dpkg-buildpackage: full upload (original source is included)
 dpkg-buildpackage: warning: Failed to sign .dsc and .changes file
 Command 'dpkg-buildpackage' failed in '/home/uwabami/svn-work/trunk/build-area/hello-2.7',
 how to continue now? [Qri?]: ignore
```

最後の gpg sign で止まっていますので i(ignore) で終わらせましょう。どうやら上手くできたみたいですね。あとは、通常通り debian/ 以下を更新していき、リリース時 (dput/dupload 時) にタグを付けたり、 branch を切ってメンテしたりしてきます。

11.5.4 new upstream release

upstream で新しいパッケージがリリースされた時には `svn-upgrade` コマンドを使って、新しいソースを登録します。とはいえ、今回やった様に `debian` ディレクトリのみを管理している場合には、 uscan の download 先を見て、適宜更新していくだけで良いでしょう。

11.5.5 tips?

佐々木は以下のコマンドを alias として登録しています。 svn-pbuilder は cowbuilder 呼び出しのための wrapper スクリプトです.

```
if [ -f /usr/bin/svn-buildpackage ]; then
  alias svn-b="svn-buildpackage -rfakeroot -us -uc --svn-ignore --svn-lintian --svn-dont-clean"
  alias svn-bc="svn-buildpackage --svn-builder='svn-pbuilder' --svn-lintian --svn-dont-clean"
  alias svn-bct="svn-buildpackage --svn-builder='svn-pbuilder' --svn-lintian --svn-tag --svn-retag --svn-dont-clean"
  alias svn-bcl="svn-buildpackage --svn-builder='svn-pbuilder-local' --svn-lintian --svn-dont-clean"
fi
```

11.6 どうするの?: git-buildpackage 編

以下では既存のパッケージとして `rabbit`[*43] の更新作業を例に、 git-buildpackage を使った作業を述べてみます。

11.6.1 インストール

```
$ sudo aptitude install git-buildpackage
```

Recommends に `pristine-tar` と `cowbuilder` があります。これらもインストールしておくと良いでしょう。`cowbuilder` に関しては、先月の水野さんの資料を参照して下さい。 `pristine-tar` については後述。

11.6.2 パッケージのリポジトリへの追加

Git なので、リポジトリの作成とか面倒な事はありません。既存のパッケージの Git リポジトリ作成には `git-import-dsc` を使用します。

[*43] `http://rabbit-shockers.org/`

```
$ apt-get source rabbit
$ git-import-dsc --pristine-tar rabbit_0.9.2-3.dsc
 gbp:info: No git repository found, creating one.
 Initialized empty Git repository in /home/uwabami/Downloads/rabbit/.git/
 gbp:info: Tag upstream/0.9.2 not found, importing Upstream tarball
 /usr/bin/pristine-tar: committed rabbit_0.9.2.orig.tar.gz.delta to branch pristine-tar
 gbp:info: Version '0.9.2-3' imported under 'rabbit'
```

この際に `--pristine-tar` オプションをつけることを推奨します。また、これまでのバージョンの `.dsc` ファイルと `.orig.tar.gz` ファイルがある場合には `git-import-dscs` コマンドを使うと良いでしょう。 tag を良きにはからってくれます。

さて、これで `rabbit` という Git リポジトリが作成されました。実際に中を見てみましょう。

```
$ cd rabbit
$ git branch
 * master                  <-- debian/ 入りのフルソース
   pristine-tar            <-- orig.tar.{gz,bz2} のバイナリデルタ
   upstream                <-- debian/ 無し (upstream) のソース
$ git tag
 debian/0.9.2-3
 upstream/0.9.2
```

ブランチの意味は上記の通りです。 `git-import-dscs` を使うと、バージョンに応じて tag が沢山並んでいると思います。 `master`,

`pristine-tar` はちょっと特殊です。このブランチには、 import される前の .tar.gz(もしくは.bz2) のバイナリデルタのみがコミットされています。 `git-buildpackage` を実行すると `upstream` ブランチ、もしくは `upstream/`**バージョン番号**タグから、元々の `orig.tar.{gz,bz2}` を生成します。 `pristine-tar` ブランチのバイナリデルタが無いと (圧縮率が違ったりして) 元々の upstream のソースを正しく生成できなかったりします。

11.6.3 パッケージのビルド

では、パッケージをビルドしてみましょう。

```
$ sudo apt-get build-dep rabbit
$ cd rabbit
$ git-buildpackage
 ...
 ...
 Successfully signed dsc and changes files
```

ちなみに `notify-send` コマンドがあると、 git-buildpackage の結果を通知してくれます。

あとは通常通り `debian/` 以下を更新していき、リリース毎にタグをつけたり、適当に branch を切って作業していきます。また (Subversion のところでは紹介しませんでしたが) `git-dch` コマンドによって、 Git のコミットログから `debian/changelog` を生成することができます (コミットログの最初の行しか生かされませんので、粒度の小さいコミットが求められます)。

11.6.4 New upstream release

upstream で新しいリリースが出た場合には `git-import-orig` で新しい `.orig.tar.{gz,bz2}` を取り込みます。

```
$ uscan --download
 rabbit: Newer version (0.9.3) available on remote site:
   http://rabbit-shockers.org/download/rabbit-0.9.3.tar.gz
   (local version is 0.9.2)
 rabbit: Successfully downloaded updated package rabbit-0.9.3.tar.gz
     and symlinked rabbit_0.9.3.orig.tar.gz to it
$ git-import-orig --pristine-tar ../rabbit_0.9.3.orig.tar.gz
 What is the upstream version? [0.9.3]
 gbp:info: Importing '../rabbit_0.9.3.orig.tar.gz' to branch 'upstream'...
 gbp:info: Source package is rabbit
 gbp:info: Upstream version is 0.9.3
 /usr/bin/pristine-tar: committed rabbit_0.9.3.orig.tar.gz.delta to branch pristine-tar
 gbp:info: Merging to 'master'
 ...
 gbp:info: Successfully imported version 0.9.3 of ../rabbit_0.9.3.orig.tar.gz
```

既存の master と自動的に merge が行なわれますので、適宜修正してきます。

11.6.5 patch-queue ブランチ?

source format 3.0 (quilt) では, upstream への変更点を quilt を用いてパッチで管理します。通常通り quilt を用いてパッチを作成 (もしくは debuild が走ったさいにパッチとして抽出) するのでも良いのですが、折角なので git format-patch でパッチを生成する方法について述べてみます。

git-buildpackage には gbp-pq というコマンドが提供されています。 pq は patch-queue の略です。先ず、現時点で debian/patches 以下にあるパッチを patch-queue ブランチへ登録します。

```
$ quilt pop -a
$ gbp-pq import
```

この時点で、 master ブランチから patch-queue/master ブランチへ切り代わります。 debian/patches 以下にあったパッチがファイル一つ毎に一つのコミットとして登録されます。適宜 rebase するなどして、パッチをマージしたり削除したりしていきます。パッチの修正が終わったら

```
$ git checkout master
$ gbp-pq export
```

で、 patch-queue/master のコミットがそれぞれパッチとして debian/patches 以下に置かれます。あとは

```
$ quilt push -a
$ git-buildpackge
```

で ok です。この方式の利点は、個々のパッチが追跡しやすくなること、 git format-patch の出力結果なので、upstream が Git を用いている場合には upstream に投げ易くなること、でしょうか[*44]?

ちなみに、 git-buildpackage のオプションには--git-debian-branch=がありますので、

```
$ git-buildpackage --git-debian-branch=patch-queue/master
```

とすると、パッチが当たった (quilt push -a) 状態の tree を用いてパッケージ作成ができます。

11.6.6 リモートリポジトリとのやりとり

適宜 git clone/push/fetch すれば良いと思いますが

1. pristine-tar, upstream ブランチ、 upstream/**バージョン番号**タグは必ず push する

に気をつけましょう。また、リモートリポジトリから git-buildpackage 用に clone するための gbp-clone コマンドも用意されています。

他にも upstream が Git を使用していると、結構幸せになれます。 git remote で、 upstream の Git リポジトリの master や、特定の tag とこちらの upstream を紐付けておくと、単一のリポジトリで全て作業を行なえたりします。

11.6.7 Tips?

佐々木は以下を alias に登録しています。

```
if [ -f /usr/bin/git-buildpackage ]; then
  alias git-b="git-buildpackage --git-ignore-new --git-builder='debuild -rfakeroot -i.git -I.git -sa -k891D7E07'"
  alias git-bc="git-buildpackage --git-ignore-new --git-builder='git-pbuilder'"
  alias git-bct="git-buildpackage --git-ignore-new --git-tag --git-builder='git-pbuilder'"
  alias git-bcl="git-buildpackage --git-ignore-new --git-builder='git-pbuilder-local'"
fi
```

[*44] とはいえ、毎度 quilt pop/push -a するのも面倒かしらん

11.7 最後に

以上、簡単に svn-buildpackage, git-buildpackage についてお話しました。実際にパッケージを作成する際には、共同作業者との合意や Team Policy によって、それなりにルールがありますのでそれを参考にして下さい。

また、結局 git-buildpackage、 svn-buildpackage で multiple upstream を使用する場合の方法論とかはちゃんと調べられませんでした。結構需要ありそうなんですけれどね。 bzr-buildpackage はその辺上手く動作するらしいです、つぎの章を読んでください。

12 vcs-buildpackage ～bzr の場合～

山下尊也

12.1 はじめに

あなたは、 VCS を使っていますか? 分散型バージョン管理システムが注目されるようになった頃から、 Linux カーネルなどでも用いられており、開発スピードの速い Git を使っている方が多い気がします。はてなブックマーク[*45]などを見ていても、上位の記事になるのは Git ばかりで、 Bazaar 使いとしてはとても悲しくなります。

私は、日々様々なファイルを扱っていますが、それらは Bazaar(bzr:バザー) を使って管理しています[*46]。そのためか、パッケージも Bazaar を用いて管理していましたが、最初のうちは適当に自分のリポジトリを作成し管理していました。無理やりパッケージの管理をしていたため、他の手法を探していると、 bzr-builddeb があったので、それ以来 bzr-builddeb を使って管理しています。

12.2 bzr-builddeb の基本

まずは、パッケージのインストールをしてみましょう。

```
 $ sudo aptitude update
 $ sudo aptitude install bzr-builddeb
```

lenny では 0.95、 squeeze では 2.4.2、 sid では 2.7.8 がインストールされます。

```
 $ dpkg -L bzr-builddeb  | grep bin
/usr/bin
/usr/bin/bzr-buildpackage
 $ bzr-buildpackage --help
Purpose: Builds a Debian package from a branch.
Usage:   bzr builddeb [BRANCH_OR_BUILD_OPTIONS...]

Options:
...[snip]
```

ヘルプに書かれている通り、ブランチから Debian パッケージを生成することを目的としたコマンドです。

[*45] http://b.hatena.ne.jp/

[*46] Bazaar を用いるようになった理由は、ファイル名が Unicode 文字列で管理されていたからです。

```
 $ bzr help commands
[snip] 一部抜粋
bd-do               Run a command in an exported package, copying the result
                    back. [builddeb]

builddeb            Builds a Debian package from a branch. [builddeb]
Aliases: bd

dep3-patch          Format the changes in a branch as a DEP-3 patch. [builddeb]

dh-make             Helps you create a new package. [builddeb]
Aliases: dh_make

import-dsc          Import a series of source packages. [builddeb]

import-upstream     Imports an upstream tarball. [builddeb]

mark-uploaded       Mark that this branch has been uploaded, prior to pushing it.
                    [builddeb]

merge-package       Merges source packaging branch into target packaging branch.
                    [builddeb]

merge-upstream      Merges a new upstream version into the current branch.
                    [builddeb]
Aliases: mu
```

bzr は他の VCS を使ったことがある人であれば、一通り使えると思います。bzr help commands でコマンドの一覧を見ることができるので、分からないときは活用してください。一覧の中で [builddeb] が bzr-buildpackag で追加されたコマンドです。

12.3 bzr-builddeb で選択できるモード

bzr-builddeb では、さまざまなモードがあらかじめ用意されています。パッケージのメンテナが使いたいスタイルに応じて、モードを選択することができます。図 9 に示すのは、メンテナのスタイルに応じたモードの選択肢です。

Q1 管理したいパッケージは native package[*47]ですか?
Q2 あなたがアップストリームメンテナーか?
Q3 debian/ ディレクトリ以下だけを保管したいのか?
Q4 パッケージ作業するときに、ブランチを分けて作業したいか?

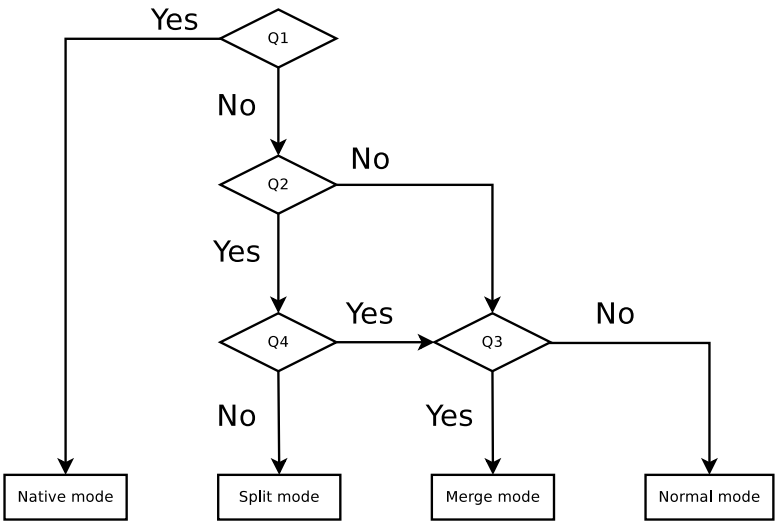

図 9 bzr-builddeb でのモードの選択

[*47] Debian 固有のパッケージ。または、ローカルでの使用のためだけに、メンテナンスしているソースファイルを含むパッケージ。例えば、debootstrap や debian-el,debian-archive-keyring など

12.4 Normal mode を用いたパッケージの管理

12.4.1 新規にパッケージを作る場合

bzr-builddeb を用いて、パッケージを新規で作成する際は、 `bzr dh-make` を用います[*48]。

```
 $ mkdir ~/src-debian-normal
 $ bzr init-repo auto-install-el
 $ cd auto-install-el
 $ bzr init unstable
 $ cd unstable

 $ bzr dh-make auto-install-el 1.53 ../auto-install-el_1.53.orig.tar.gz
Fetching tarball
Looking for a way to retrieve the upstream tarball
Upstream tarball already exists in build directory, using that
Committing to: /tmp/test/auto-install-el/unstable/
added auto-install.el
Committed revision 1.

Type of package: single binary, indep binary, multiple binary, library,
 kernel module, kernel patch?
 [s/i/m/l/k/n] s

Maintainer name  : Takaya Yamashita
Email-Address    : takaya@debian.or.jp
Date             : Sun, 25 Sep 2011 01:01:31 +0900
Package Name     : auto-install-el
Version          : 1.53
License          : blank
Type of Package  : Single
Hit <enter> to confirm:
Skipping creating ../auto-install-el_1.53.orig.tar.gz because it already
 exists
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 auto-install-el Makefiles install into $DESTDIR and not
 in / .
Package prepared in /tmp/test/auto-install-el/unstable
 $ ls
auto-install.el  debian/

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

[*48] マニュアルには、開発段階のため別の場所で dh-make して、 debian ディレクトリのファイルをコピーしろと書かれていますが、 bzr dh-make で大丈夫でしょう。

```
 $ bzr status
added:
  debian/
  debian/README.Debian
  debian/README.source
  debian/changelog
  debian/compat
  debian/control
  debian/copyright
  debian/docs
  debian/rules
  debian/source/
unknown:
  debian/auto-install-el.cron.d.ex
  debian/auto-install-el.default.ex
  debian/auto-install-el.doc-base.EX
  debian/emacsen-install.ex
  debian/emacsen-remove.ex
  debian/emacsen-startup.ex
  debian/init.d.ex
  debian/manpage.1.ex
  debian/manpage.sgml.ex
  debian/manpage.xml.ex
  debian/menu.ex
  debian/postinst.ex
  debian/postrm.ex
  debian/preinst.ex
  debian/prerm.ex
  debian/watch.ex
  debian/source/format

 $ bzr log -v --include-merges
------------------------------------------------------------
revno: 1
tags: upstream-1.53
committer: Takaya Yamashita <yamashita@takaya.biz>
branch nick: unstable
timestamp: Sun 2011-09-25 01:01:30 +0900
message:  Import upstream version 1.53
added:  auto-install.el

 $ edit
 $ edit
 $ edit
[snip]
 $ bzr builddeb
```

12.4.2 既存のパッケージを bzr-builddeb で管理する場合

```
 $ mkdir ~/src-debian-normal
 $ bzr init-repo auto-install-el
 $ cd auto-install-el

 $ apt-get source auto-install-el

 $ bzr init unstable
 $ cd unstable
 $ bzr import-dsc ../*.dsc
Committing to: /home/takaya/src-debian-normal/auto-install-el/tmpriXU3G/upstream/
added auto-install.el
Committed revision 1.
All changes applied successfully.
Committing to: /home/takaya/src-debian-normal/auto-install-el/unstable/
added .pc
added debian
added .pc/.quilt_patches
added .pc/.quilt_series
added .pc/.version
added debian/README.Debian
added debian/changelog
added debian/compat
added debian/control
added debian/copyright
added debian/dirs
added debian/emacsen-install
added debian/emacsen-remove
added debian/emacsen-startup
added debian/rules
added debian/source
added debian/source/format
Committed revision 2.
```

```
 $ bzr log -v --include-merges
------------------------------------------------------------
revno: 2
tags: 1.48-1
fixes bug(s): http://bugs.debian.org/586177
author: Takaya Yamashita <takaya@debian.or.jp>
committer: Takaya Yamashita <yamashita@takaya.biz>
branch nick: unstable
timestamp: Tue 2010-06-15 23:16:42 +0900
message:
  Initial release (Closes: #586177)
added:
  .pc/
  .pc/.quilt_patches
  .pc/.quilt_series
  .pc/.version
  debian/
  debian/README.Debian
  debian/changelog
  debian/compat
  debian/control
  debian/copyright
  debian/dirs
  debian/emacsen-install
  debian/emacsen-remove
  debian/emacsen-startup
  debian/rules
  debian/source/
  debian/source/format
------------------------------------------------------------
revno: 1
tags: upstream-1.48
author: Takaya Yamashita <takaya@debian.or.jp>
committer: Takaya Yamashita <yamashita@takaya.biz>
branch nick: upstream
timestamp: Tue 2010-06-15 23:16:42 +0900
message:
  Import upstream version 1.48
added:
  auto-install.el

 $ bzr merge-upstream ../auto-install-el-1.53.orig.tar.gz --version 1.53
 --distribution debian --package auto-install-el
下記でも大丈夫
 $ bzr merge-upstream ../auto-install-el_1.53.orig.tar.gz --version 1.53
Using distribution unstable
Using version string 1.53.
Committing to: /home/takaya/src-debian-normal/auto-install-el/tmpNlFI08/upstream/
modified auto-install.el
Committed revision 2.
All changes applied successfully.
The new upstream version has been imported.
You should now review the changes and then commit.
```

`bzr merge-upstream` コマンドを用いて、アップストリームのソースファイルをインポートします。拡張子では `.tar.gz`, `.tar`, `.tar.bz2`, `.tar.lzma`, `.tgz`, `.zip` に対応しています[*49]。

また、ワーキングツリーの変更を行わずにアップストリームの変更をインポートする `bzr import-upstream` もあります。

12.5 Merge mode を用いたパッケージの管理

Merge mode は Normal mode に比べて少し複雑な作業が必要になってきます。コマンドなども整備されていませんが、 debian ディレクトリ以下だけをリポジトリに管理することができる利点があります。また、共同で作業するときなどは、ファイルサイズを抑えることができます。

```
 $ mkdir ~/src-debian/
 $ bzr init-repo ~/src-debian/twittering-mode
 $ cd ~/src-debian/twittering-mode
 $ bzr init unstable
 $ cd unstable
 $ mkdir .bzr-builddeb/
 $ echo -e '[BUILDDEB]\nmerge = True' > .bzr-builddeb/default.conf
 $ bzr add .bzr-builddeb/default.conf
```

本来、アップストリームから新しいバージョンが出た際は、 `bzr merge-upstream` を用いますが、 Merge mode では

[*49] LZMA/XZ/Lzip の対応については、 Bug 499484 に wishlist として報告されています。これらについては、アップストリームの trunk では改善しているようです

対応していないため[*50]、バージョン番号を指定してあげる必要があります。[*51]

```
 $ dch -v 2.0.0+git20110905-1
[snip]
 $ bzr builddeb
 $ ../dput debexpo twittering-mode_2.0.0+git20110905-1_amd64.changes
 $ bzr ci -m "New upstream version 2.0.0+git20110905"
```

処理を見ると、`~/src-debian/twittering-mode/build-area` にて `debuild` の作業が行われています。

また、 backports 向けにパッケージを作る際は、 backports 専用の branch を作成して、そこで作業しています。

```
 $ cd ~/src-debian/twittering-mode
 $ bzr branch unstable bpo
 $ cd bpo
 $ bzr bd-do "dch --bpo"
[snip]
 $ bzr builddeb
[snip]
 $ ls -1 ../*bpo*
../auto-install-el_1.53-1~bpo60+1.debian.tar.gz
../auto-install-el_1.53-1~bpo60+1.dsc
../auto-install-el_1.53-1~bpo60+1_all.deb
../auto-install-el_1.53-1~bpo60+1_amd64.build
../auto-install-el_1.53-1~bpo60+1_amd64.changes
```

既存のパッケージを Merge mode に移行する場合は、リポジトリを作成し、 debian ディレクトリをコピーすれば良いでしょう。

```
 $ mkdir ~/src-debian/
 $ bzr init-repo ~/src-debian/twittering-mode
 $ cd ~/src-debian/twittering-mode

 $ apt-get source twittering-mode

 $ bzr init unstable

 $ cp -r twittering-mode-2.0.0+git20110905/debian ~/src-debian/twittering-mode/unstable

 $ cd unstable
 $ mkdir .bzr-builddeb/
 $ echo -e '[BUILDDEB]\nmerge = True' > .bzr-builddeb/default.conf
 $ bzr add .bzr-builddeb/default.conf

 $ bzr add .
 $ bzr ci -m "initial commit"
 $ bzr builddeb
```

`bzr bd-do` を使うと、 build-area に一時的にコピーを行い、 `dpatch` などのコマンドを使用することができるようです[*52]。

12.6 管理していく上でのヒント

普段は GPG 署名をせずに、必要なときだけパッケージに署名を行なっている人も多いと思います。`--`の後にコマンドを足すことによって、 builder にオプションを渡すことができます。

```
 $ debuild -rfakeroot -us -uc
 $ bzr bd -- -us -uc
```

[*50] changelog に対象としたリリースを反映させることができない? bzr merge-upstream で –distribution を使えばいけるかも-

[*51] debian ディレクトリを別の場所で管理しているため、 Bazaar の履歴が受け継がれません。

[*52] 未検証

13 vcs-buildpackage ～ Git の場合 (again)～

佐々木洋平さん

はじめに

話の枕

山下さんの bzr 編に引き続き、ここでは佐々木が Git を用いて Debian パッケージを作成する場合についてまとめます。前々回の svn と Git についての記事でも `git-buildpackage` について (簡単に) 触れましたが、その後ちゃんと調べたら、幾つかコマンドが新しく追加されていたりしました。ですので、前々回の復習も兼ねて「Git を使って Debian パッケージを作成/管理するお話」をしてみたいと思います。

前提とする知識と目的

とはいえ、パッケージング全てについて触れる事はできませんので、ここでは

- source package についてのある程度の知識
- Git に関して, 特に tag と branch についてのある程度の知識

があることを前提としています。最後に参考文献していますので、適宜参照して下さい、もしくは質問して下さい。

パッケージ作成作業 (復習)

通常、パッケージ作成は

1. upstream のソースを取得
2. (場合によっては) non-free な部分を除いたりして、
3. `./debian` ディレクトリ以下を作成/更新して、
4. 場合によっては upstream のソースにパッチを当てて、
5. ソース/バイナリ パッケージをビルド

という事を行ないます。これらの作業を VCS で管理します。

典型的なリポジトリレイアウト

前回お話した `git-import-dsc` で既存のソースパッケージを import したり、`debcheckout` で Git で管理されているパッケージを checkout すると、多くの場合、リポジトリは以下の様になります。

```
$ git branch
* master            <-- debian/ 入りのフルソース
  pristine-tar      <-- orig.tar.{gz,bz2} のバイナリデルタ
  upstream          <-- debian/ 無し (upstream) のソース
```

ここで `git-buildpackage` を実行すると、パッケージのビルドが始まります。今日は**この状態に持って行くまで**の話にフォーカスしてみます。

upstream ソースを import するには?

upstream ソースを import するには?

upstream のソースを持ってきて、 Git リポジトリを作成することを考えると、

- simple な場合
  1. tarball を展開して import
  2. upstream の VCS を import
- 調整が必要な場合
  - non-dfsg-free な部分を削除してから import

... でしょうか?

注意すべきは `pristiner-tar` を用いること、です。

pristine-tar ?

gzip の圧縮率の違いなどから、 `upstream` ブランチから生成された .tar.gz は upstream の配布物と異なる事があります。 `pristine-tar` によって、 upstream の tarball を import する際にバイナリデルタを保存しておくことで、 `upstream` ブランチから tarball(`.orig.tar.gz`) を生成する際に、 checksum の等しい tarball を生成することができます。

このバイナリデルタは `pristine-tar` ブランチに保存されます。もし忘れた場合には

```
$ pristine-tar commit foobar.tar.gz [upstream の tag]
$ pristine-tar checkout ../foobar.tar.gz
```

の様にして後からコミットできますが、最初に import する際に忘れずにバイナリデルタを保存しておくのが良いでしょう。 `git-import-orig` コマンドにはオプションとして `--pristine-tar` があります。ごく稀に上手くバイナリデルタを生成できない tarball がある (らしい) ですが...

upstream の VCS から import する

注意すべきは **tarball も必ず import すること** でしょうか? これは、履歴とともにバイナリデルタを保持するために必要な作業です。

また、リリースされている tarball は Tag が打たれている (もしくはそれに類するコミットがある) ハズなので、履歴を適宜修正すると upstream のコミットを patch として管理しやすくなります。

upstream の VCS から import する (1)

幸運にも upstream が Git だったら

```
$ git remote add upstream-repos [url]
$ git fetch upstream-repos
$ git co upstream && git merge upstream-repos
```

で ok です。

upstream の VCS から import する (2)

Subversion の場合は `git-svn` を用います。毎度 rebase しながら作業することになるので、大変面倒ですが...[*53]

[*53] もっと良い方法ありませんかね?

- Subversion: 初回

```
$ git-svn init [url]
$ git svn fetch
$ git log ref/remotes/git-svn
$ git checkout -b upstream refs/remotes/git-svn
$ git push origin upstream:upstream
```

- Subversion: 二回目以降

```
$ git config --remove-section svn-remote.svn 1>/dev/null 2>&1
$ git svn init [url]
$ git show-ref origin/upstream > \
   `git rev-parse-git-dir`/refs/remotes/git-svn
```

tarball を import するツール

以下では, tarball を import するコマンド群についてまとめておきます。

■git-import-orig

- `git-buildpackage` パッケージで提供
  - simple に tarball を import
  - (Option つければ) pristine-tar も実行
  - (あれば) 現状の master ブランチへ自動で merge して
  - タグも打ってくれる
- 一番 simple
  - 必要な事は全てやってくれるので, これで十分な事が多い。

■git-dpm import-new-upstream

- `git-dpm`: git Debian package manager
- 動作は git-import-orig とほぼ同じ
- VCS の履歴との対応や `patch-queue` ブランチ (後述) の生成/管理もしてくれる

調整が必要な場合 (1)

upstream の配布物に non-dfsg-free な部分があったりして調整が必要な場合は

- upstream ブランチで non-dfsg-free な部分を削除/調整
- new upstream version として merge/commit
- tarball として repack した後に import/タグ打ち

なんて事をします。例えば

```
$ git checkout upstream
$ git merge -s recursive -X theirs [upstream tag]
```

もしくは

```
$ git status -s | egrep '^(DU|UA| U|UD)' | cut -c4- | \
    xargs git rm --ignore-unmatch DUMMY$$
$ git commit
```

とか?

uscan に repack 用の hook script を使っているなら、それを実行したのち tarball として import する、の方が楽かもしれません。

./debian をガシガシ書く/修正する

まあこれは良いですよね?

```
$ git branch
* master
  pristine-tar
  upstream
```

- upstream のソースは全て Git リポジトリの upstream ブランチ
- ./debian での変更は master ブランチで
  - 全ての変更は master 内で行なう
  - 何をしたのかは git log で容易に追跡できる
  - patch も作成しやすい

となります。 Happy Hacking!!

patch を扱うには?

source format 3.0 (quilt) では、 upstream への変更点を quilt を用いてパッチで管理します。

単純な方法

Git の事は忘れて quilt だけでパッチを作成 (もしくは debuild が走った際にパッチとして抽出) する、です。
複雑な事は何もありませんが、 VCS の恩恵を受けることもありません。

git を使う場合 (1)

逆に quilt を忘れて Git だけで patch を管理する方法です。 debuild 等で patch を生成し, ./debian/patches/ 以下を git で管理します。まあ VCS っぽく管理するなら 1 パッチ/1 コミットとして手動で管理するのでしょうか?

patch ブランチで

パッチ**だけ**を track するための branch を用意して、 1 パッチ/1 ブランチ or 1 パッチ/1 コミットとして管理します。注意すべきは

- quilt への export を行なうにはコミット履歴が**綺麗**でないといけない
  - 1 パッチ/1 コミット
  - squash !! squash !! squash !!
- 今のところ 1-way rebase なので、 upstream の更新をする度に作業が必要。

以下, 幾つかのコマンドについて述べます。

■topgit: a Git patch queue manager

- コミット履歴をブランチで管理
- パッチ間の依存関係も管理
- 便利だけど, やりすぎな気もしないでもない
- patch-queue ブランチから quilt へ export したパッチは master ブランチにコミットしておく必要がある

■gbp-pq

- git-buildpackage で提供

- `git format-patch` の wrapper
- 1 パッチ/1 コミット, として patch を生成/取り込み
- `master` を rebase して使う

```
$ git checkout master ; git branch -D patch-queue
$ quilt pop -a
$ gbp-pq import
 ... 作業 ...
$ git checkout master ; gbp-pq export
```

■git-dpm

- パッチは一つのブランチで管理
- 1 パッチ/1 コミット
- パッチは `master` ブランチに merge されたままで管理
- パッチが当たった `upstream` ブランチを rebase
- プライベートブランチの SHA1 ハッシュを`./debian/.git-dpm` に保存

■gitpkg の quilt export hook

- 1 パッチ/1 コミット, などという制限は無い
- `debian/source/git-patches` に設定を書く

```
    upstream/[UPSTREAM_REF]...patche-queue1/[DEBIAN_REF1]
    upstream/[UPSTREAM_REF2]...topic1/[DEBIAN_REF2]
```

- パッチはコミットされない
- tag は再生成される

source package の生成

■git-dpm

- ビルド用の特定のコマンドは無い (`dpkg-source -b` とか)

■gitpkg

- pristine-tar, `upstream` ブランチから tarball を生成し source package をビルド

■git-buildpackage

- default. バイナリパッケージも作成する
- `git-pbuilder`: pbuilder/cowbuilder を呼び出せる
- タグを打ったり.

まとめ (?)

いろいろコマンドが増えてきましたが、結局のところ `git-buildpackage` が一番簡単/便利/移行コストも低い、という印象です。 workflow が他の vcs-buildpackage と似ているから、でしょうか。

また git-dpm/gitpkg は workflow/patch-queue の自由度は高い (けれど, 複雑になりがち)、な印象を受けます。

git-dpm パッケージは提供するコマンドが多くて、

- コマンドが多くて, ちょっと敷居が高い (かも)
- 一番「 git らしく」作業できる (らしい)

ですね。また、

- gitpkg
  - hook での拡張/カスタマイズが容易.
  - リポジトリのレイアウトも固定されていない

です。

14 パッケージを作ったらスポンサーアップロード

岩松 信洋

14.1 はじめに

Debian にパッケージをアップロードする場合、誰でもアップロードできるわけではなく限られた人しかアップロードできません。アップロードできるのは Debian Developer(以下、 DD) と Debian Maintainer（ 以下、 DM）だけです。また、 DM はアップロードする際に制限があります。

DD ではない人がメンテナンスしているパッケージをアップロードする場合には、 DD に頼んでアップロードしてもらう必要があります。パッケージを代理でアップロードする人をスポンサーといい、アップロードする行為をスポンサーアップロードといいます。パッケージメンテナに変わってパッケージをアップロードするので、パッケージに対して責任が問われる作業です。よって、アップロードするパッケージの内容やパッケージメンテナについてある程度知っておく必要があります。スポンサーはパッケージのチェック等を行ったりパッケージ内容に対して助言をするので、 mentor（ 助言する人）といった方がわかりやすいかもしれません。このパッケージチェックの過程は DD や DM になる場合に優位に働く場合があります。 DD や DM になる時には既存の DD に支持者になってもらう必要があるのですが、この場合スポンサーに支持者担ってもらうように依頼すると、しっかりした内容の支持内容を提供してくれるはずです。

では、スポンサーがどのようにしてパッケージをスポンサーアップロードするのか説明します。

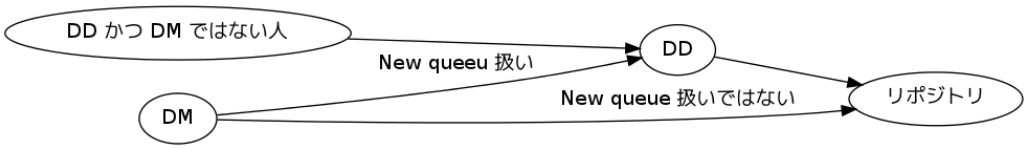

図 10　パッケージアップロード

14.2 スポンサーアップロードするときに確認する内容

パッケージメンテナに「 アップロードして! 」と言われて、何も考えずにアップロードしてしまうと変なパッケージが Debian のパッケージリポジトリに入ることになり、いろいろ問題が起きてしまいます。よって、スポンサーはアップロードするパッケージメンテナとパッケージを確認する必要があります。スポンサーは確認する前と後にチェックする内容があります。スポンサーによって内容が異なりますが、ここでは私が行っているチェック内容を紹介します。

14.2.1 パッケージをチェックする前のチェック

スポンサーをするパッケージメンテナの方に以下の内容を確認しています。

- Web of Trust（ WOT） に入っているか。
  GPG の鍵チェックを行います。 WOT に入ってない場合には近くにいる DD にキーサインしてもらうように依頼しています。
- DD や DM への意欲はあるか。
  ただパッケージメンテナになるのもいいのですが、私を含めたスポンサーの多くは、メンテナには最終的に DD になってもらって、 Debian の開発に参加して欲しいと思っていると思います。よって、パッケージメンテナの次のステップについて考えているか、確認しています。私は DD や DM になりたくないからといって、スポンサーにならないことはないです。
- Debian 新メンテナーガイド[*54] を読んだか。
- DFSG [*55] を読んだか。
- Debian Policy[*56] を読んだか。
- Debian Reference [*57]を読んだか。
  これらは読んでおくべきドキュメント類です。読んでないと話にならないので、まず読んである程度理解してもらうようにしています。

これ以外にも、たまに誰もスポンサーする人がいないようなのでスポンサーする場合があります。

14.2.2 パッケージのチェック

次に実際のパッケージのチェックを行います。内容は以下になります。

- ライセンスの確認
  ソフトウェアのライセンスが DFSG に合致するライセンスか、ライセンスが debian/copuright に書かれているか確認します。この確認には devscripts パッケージに含まれる licensecheck を使うことが多いです。
  また、最近では debian/copyright のフォーマットには、 DEP5 [*58] に対応しているか確認しています。仮に不明な場合には上流開発者に問い合わせるようにメンテナにお願いします。
- orig.tar.gz の確認
  オリジナルの tar ボールと一緒か、オリジナルのソースコードに変な改変をしていないかを確認します。
- 最新のパッケージングのルールに合っているかの確認。
  例えば、使っているプログラミング言語向けのパッケージングサポートツールが新しくなっていたり、パッケージングポリシーが決まっている場合があります。できるだけ新しいパッケージングのルールに合わせるようにします。
- debian/control ファイルの確認
  依存関係、パッケージの説明、各セクションの確認を行います。
- debian/rules の確認
  シンプルな構成になっているか、ポリシーに違反していないかの確認。
- pbuilder を使ったパッケージビルドの確認
  最新 unstable ディストリビューションでパッケージがビルドできるか確認します。 lintian によるチェックや、ビルドに必要なパッケージが依存関係から漏れていないか確認することができます。この時に使うツールは pbuilder[*59](cowbuilder[*60]) と、 sbuild[*61] です。手元でビルドしてアップロードする場合には pbuilder を使っています。スポンサーをしているパッケージは定期的にビルドの確認を行うようにしており、これには sbuild を

[*54] http://www.debian.org/doc/manuals/maint-guide/
[*55] http://www.debian.org/social_contract.ja.html
[*56] http://www.debian.org/doc/debian-policy/
[*57] http://www.debian.org/doc/manuals/developers-reference/
[*58] http://dep.debian.net/deps/dep5/
[*59] http://packages.qa.debian.org/p/pbuilder.html
[*60] http://packages.qa.debian.org/c/cowdancer.html
[*61] http://packages.qa.debian.org/s/sbuild.html

使っています。

- lintian を使ったポリシーとパッケージングミスの確認
  パッケージが Debian ポリシー に準拠しているか簡単に確認するには lintian[*62] を使います。これは Debian ポリシー の他に Debian パッケージのよくある間違いに関してチェックしてくれます。
- メンテナスクリプト (preinst、 postinst、 prerm、 postrm、コンフィグ) の確認
  これらは動くのか、必要なものなのかをチェックします。
- オリジナルの tar ボールとの差分の確認
  diff.gz の内容を確認します。作成されたパッチは上流開発者に送ってあるか、パッチは DEP3 [*63] に対応しているか、確認します。
- パッケージのインストール、アンインストール、動作確認
  パッケージはできても、インストールできない場合やアンインストールできない場合があります。またパッケージが動作しない場合もあります。このような問題がないか確認するために、 piuparts[*64] を使ってインストール、アンインストールのチェックと、実際にインストールしてみて動作するか確認をします。

14.2.3 その他

その他、スポンサーによっては以下のような理由でスポンサーしてくれない場合があるようです。注意しましょう。

- スポンサーを Uploader に入れることを要求される場合がある。
  これは、パッケージメンテナの代わりにパッケージをアップロードする場合に有効です。先に説明したように、パッケージのメンテナンスの責任はスポンサーにもあるのためです。
- パッケージング用のツールを要求される場合がある。
  例えばスポンサーによっては、パッケージに cdbs を使っている場合、 debhelper を使うように言われる場合があるようです。
- `http://mentors.debian.net` を使わない場合はスポンサーをしない。
  信頼できる以外にアップロードされたソースパッケージは信頼しないというポリシーのようです。
- 自分の知らないプログラミング言語で書かれたパッケージはスポンサーをしない。

また、 `http://wiki.debian.org/SponsorChecklist` に実際にスポンサーしている人の方針が纏められています。

14.3 アップロード

アップロードには、 dput や dupload パッケージを使います。実装が異なるだけで、基本的な機能は揃っているのでどちらでも使い方は同じです。

14.4 まとめ

以上のようにスポンサーになることはとても大変なので、メンテナの方はさっさと DM か DD がになりましょう。

[*62] `http://packages.qa.debian.org/l/lintian.html`
[*63] `http://dep.debian.net/deps/dep3/`
[*64] `http://packages.qa.debian.org/p/piuparts.html`

15 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。

問題 1. alioth.debian.org が 2 台に分かれました。そのサーバ名は?

☐ A vasks.debian.org と wagner.debian.org
☐ B volks.debian.org と don.debian.org
☐ C dennys.debian.org と gusto.debian.org

問題 2. 現在行われている Perl transition の Perl バージョンは?

☐ A 5.12
☐ B 5.13
☐ C 5.14

問題 3. プライマリミラーサーバが新しく追加された国は?

☐ A チュニジア
☐ B 中国
☐ C マダガスカル

問題 4. mentors.debian.net を構築している web アプリケーションが変更されました。何に変わったでしょう?

☐ A Debmemtors
☐ B Debcomike
☐ C Debexpo

問題 5. debian-ports に追加された新しいアーキテクチャは?

☐ A s390x
☐ B ppc64
☐ C blackfin

問題 6. 新しくサポートされた圧縮形式は?

☐ A rar
☐ B cab
☐ C xz

問題 7. Samuel Thibault がアナウンスした Debian GNU/Hurd の内容は?

☐ A Wheezy で Debian GNU/Hurd をリリースします!
☐ B なんつーか、飽きた。
☐ C DVD が読めないので DVD イメージは配布しません。

問題 8. Emdebian Grip はなぜ Debian のリポジトリに入れる事が可能なのか?

☐ A Debian だから。
☐ B Free だから。
☐ C パッケージの互換性があるから。

問題 9. mentors.debian.net を構築している web アプリケーションが変更されました。何に変わったでしょう?

☐ A Debmemtors
☐ B Debcomike
☐ C Debexpo

問題 10. debian-ports に追加された新しいアーキテクチャは?

☐ A s390x
☐ B ppc64
☐ C blackfin

問題 11. 新しくサポートされた圧縮形式は?

□ A rar

□ B cab

□ C xz

問題 12. Samuel Thibault がアナウンスした Debian GNU/Hurd の内容は?

□ A Wheezy で Debian GNU/Hurd をリリースします!

□ B なんつーか、飽きた。

□ C DVD が読めないので DVD イメージは配布しません。

問題 13. Emdebian Grip はなぜ Debian のリポジトリに入れる事が可能なのか?

□ A Debian だから。

□ B Free だから。

□ C パッケージの互換性があるから。

問題 14. Debian 温泉 2011 の 1 日目はいつでしょうか?

□ A 9/17

□ B 9/18

□ C 9/19

問題 15. 8 月に Debian は誕生日を迎えました。何周年でしたでしょうか?

□ A 17

□ B 18

□ C 19

問題 16. 最新の Debian News はいつ発行されたでしょうか?

□ A 9/17

□ B 9/18

□ C 9/19

問題 17. 10/17 の”delegation for the DSA team” で代表団に任命されなかったのは誰でしょう?

□ A Faidon Liambotis

□ B Luca Filipozzi

□ C Nobuhiro Iwamatsu

問題 18. Wheezy フリーズの予定はいつでしょう?

□ A 2012 年 4 月

□ B 2012 年 6 月

□ C 2012 年 8 月

問題 19. 1.16.1 がリリースされた dpkg に該当するのはどれ?

□ A `dpkg-buildpackage` コマンドでは `CFLAGS`, `CXXFLAGS`, `LDFLAGS`, `CPPFLAGS`, `FFLAGS` の `export` が必須になった

□ B `dpkg-deb` コマンドに`--verbose` オプションが追加された

□ C Multi-Arch フィールドがサポートされた

16 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. A：ほかはファミレスの名前
2. A：5.14 はまだ experimental です。
3. B：チュニジアとマダカスカルはミラー。プライマリではない。
4. C：Python と Turbogears で書かれた Web アプリケーション。パッケージレビューやテストスィートを提供するらしい。
5. A：s390x。 aurel32 によって開始。 blachfin はまだサポートされていない。
6. C：可逆圧縮アルゴリズム LZMA（ Lempel-Ziv-Markov chain-Algorithm）を使った圧縮形式。 GNU zip に比べ、約 40% 圧縮率が向上している。圧縮には時間がかかるが、伸長には時間がかからない。
7. A：Whezzy のリリースゴール対象に入れるようです。 PorterBox も用意されました。
8. C：Emdebian Grip はパッケージからドキュメントファイルなどの組み込みには必要のないファイルを削除したパッケージを提供するディストリビューション。
9. C：Python と Turbogears で書かれた Web アプリケーション。パッケージレビューやテストスィートを提供するらしい。
10. A：s390x。 aurel32 によって開始。 blachfin はまだサポートされていない。
11. C：可逆圧縮アルゴリズム LZMA（ Lempel-Ziv-Markov chain-Algorithm）を使った圧縮形式。 GNU zip に比べ、約 40% 圧縮率が向上している。圧縮には時間がかかるが、伸長には時間がかからない。
12. A：Whezzy のリリースゴール対象に入れるようです。 PorterBox も用意されました。
13. C：Emdebian Grip はパッケージからドキュメントファイルなどの a 組み込みには必要のないファイルを削除したパッケージを提供するディストリビューション。
14. A：さっきの話を聞いて (読んで) いればわかって当然ですね
15. B：今年もお祝いしましたよね
16. C：購読していれば知っていて当然ですね
17. C：他に任命されたのは、 Martin Zobel-Helas,Peter Palfrader,Stephen Gran,Tollef Fog Heen の全部で合計 6 名です。
18. B：あと 6 ヶ月ですよ!
19. B：`dpkg-buildpackage` ではこれらのオプションが不要になりました。 Multi-Arch は 1.16.2 からサポートされる予定です。 `dpkg-deb -x/--extract -v/--verbose` で `dpkg-deb -X/--xextract` と同じ動きをするようになりました。

17 索引

『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 2011/06〜2011/11 まで東京エリアは第 78 回から第 81 回まで (第 82 回は OSC 2011 Tokyo/Fall のため収録無し)、関西エリアは第 48 回から第 52 回まで (第 53 回は KOF 2011 のため収録無し)。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 2011 年冬号

2011 年 12 月 31 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）