

月刊

# Debian 専

日本唯一のDebian専門月刊誌

2011年10月22日

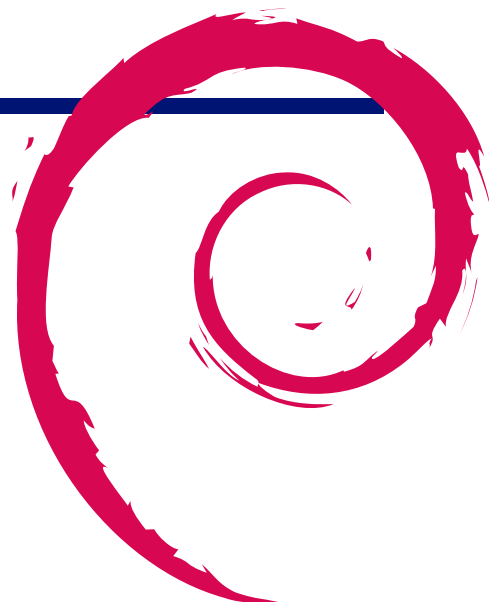
特集 1: Haskell と Debian の辛くて甘い関係

特集 2: 新連載! 月刊 debhelper



# 1 Introduction

上川 純一



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

Debian 勉強会の目的は下記です。

- Debian Developer (開発者) の育成。
- 日本語での「開発に関する情報」を整理してまとめ、アップデートする。
- 場 の提供。
  - 普段ばらばらな場所にいる人々が face-to-

face で出会える場を提供する。

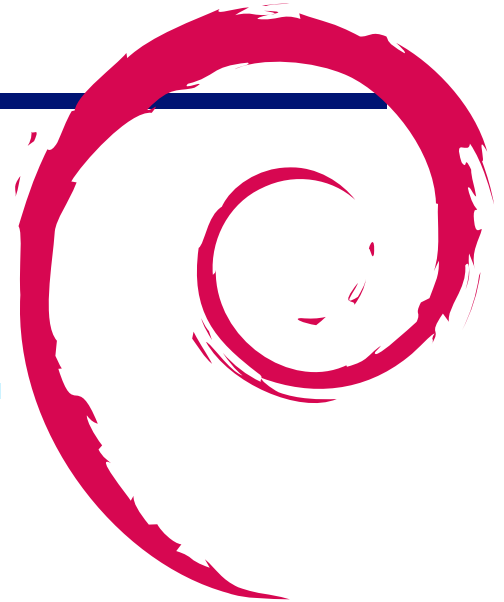
- Debian のためになることを語る場を提供する。
- Debian について語る場を提供する。

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

# Debhelper勉強会

## 目次

1	Introduction	1	5	Debian とはなにか?	7
2	事前課題	3	5.1	Debian とは? . . . . .	7
2.1	koedoyoshida . . . . .	3	5.2	特徴 . . . . .	7
2.2	sheeta38 . . . . .	3	5.3	まとめ . . . . .	9
2.3	r.matsumiya@syntaxerror.biz . . . . .	3	5.4	んで、どういう風に使えばいいの? . . . . .	10
2.4	吉野 (yy_y-ja-jp) . . . . .	3	6	Haskell と Debian の辛くて甘い関係	11
2.5	キタハラ . . . . .	3	6.1	Haskell というプログラミング言語 . . . . .	11
2.6	木村 陽一 . . . . .	4	6.2	cabal によるパッケージ管理 . . . . .	12
2.7	dictoss(杉本 典充) . . . . .	4	6.3	でも cabal には色々不都合が、、、 . . . . .	12
2.8	Kiwamu Okabe . . . . .	4	6.4	cabal をパッケージシステムとして使うことの問題点 . . . . .	13
2.9	田原悠西 . . . . .	4	6.5	Hackage を Debian パッケージ化する . . . . .	17
2.10	Arnaud Fontaine . . . . .	4	6.6	haskell-debian-utils のしくみ	18
2.11	Hirota Kawata . . . . .	4	6.7	作ったパッケージを Debian に登録するには . . . . .	19
2.12	mittty . . . . .	4	7	月刊 debhelper 第 1 回	20
2.13	osamu@debian.org . . . . .	4	7.1	debhelper とは何か? . . . . .	20
2.14	pi8027 . . . . .	4	7.2	月刊 debhelper とは? . . . . .	20
2.15	taitioooo (長谷川) . . . . .	4	7.3	debian パッケージ構築、全体の流れ . . . . .	21
2.16	野島 貴英 . . . . .	4	7.4	その他 debhelper の重要な機能	22
2.17	岩松 信洋 . . . . .	4	7.5	今月のコマンド: dh_testdir . . . . .	23
2.18	まえだこうへい . . . . .	4	7.6	今月のコマンド: dh_bugfiles . . . . .	23
3	最近の Debian 関連のミーティング報告	5	7.7	次の発表者 . . . . .	24
3.1	東京エリア Debian 勉強会 80 回目報告 . . . . .	5			
4	Debian Trivia Quiz	6			



## 2 事前課題

前田 耕平

今回の事前課題は以下です:

1. 今、コンピュータを使ってあなたがやりたいこと、やっていることはなんですか?
2. それは Debian(または Linux) でできますか? Debian(または Linux) を使ってなければその理由を教えてください。

この課題に対して提出いただいた内容は以下です。

### 2.1 koedoyoshida

1. 仕事はさておくとして、自宅環境ではサーバ系は Debian、クライアント系は Windows にしています。サーバには X も入っていないので GUI 系操作はすべて Windows ということになります。クライアントでやっていることが Debian で出来るかですが...
  - メール:Beckey:以前 (Thunderbird 2.X) 移行を検討したけれど振り分けルール (日本語 (である/でない) のメール判別、複合条件の and/or,from および宛先 (to/cc/bcc) 判別) が複雑で TB で再現ができず移行困難で断念
  - ブラウザ:メインブラウザが Opera:速度面で Firefox(iceweasel) より快適。サブは Firefox と Chrome なので iceweasel と Chromium に移行するのはもっさりすること以外は特に問題なし
  - エディタ:秀丸:Emacs へ移行すれば問題なし
  - デバイスコントロール:
    - X Video Station:TV 録画サーバ:対応ソフトウェアが WinowsXP のみ、アナログ放送が終了したのでお役御免かと思いきや、xvproxy<sup>(\*1)</sup> とデジタルアナ変換で寿命が延びた。Garapon TV 等へ移行予定。動作監視 (EPG チェック) は Debian にて実施中
    - Garapon TV:ワンセグ録画サーバ:Firefox で閲覧しているので iceweasel へ移行可能。動作監視 (ハングチェック等、異常時自動再起動) については Debian にて実施中
    - Scansnap:自炊 PDF を管理:移行可能らしいが詳細未調査
    - iPhone&iPod:音楽および動画ライブラリ:移行可能らしいが詳細未調査

### 2.2 sheeta38

1. 授業 (プログラミング入門 1) のレポート作成。 gcc と  $\LaTeX$  を用いている
2. プログラミング、および Linux にもっと触れる。現状、コンピュータに関して知っていることがあまりに少ないと感じており、これらに触れていくことで、少しずつ肩をならしていきたい。  
C 言語も  $\LaTeX$ 、あるいは他の言語 (Ruby, Python 等) も、Windows で環境を整えようとするれば可能だ。ただし Linux の方が環境構築が容易であると思う。他に具体的な Linux(Debian) の利点があるならば知りたい。

### 2.3 r.matsumiya@syntaxerror.biz

開発メイン、レポート作成。両方とも Debian 使ってます。

### 2.4 吉野 (yy-y-ja-jp)

某社製オフィススイートのファイルをレイアウトを壊さず表示・編集したいです。

### 2.5 キタハラ

時間ギリギリなので、将来やりたいことで、Linux だとやりにくそうなものをあげてみる。(よい方法があれば教えてください。)

1. 地デジの録画  
色々方法があるようですが、コンテンツ保護とかで面倒そう
2. 自炊コンテンツの作成  
スキャンはできそうだが、OCR とかスキャナ付属のおまけに負けて、Windows を使いそうな気が.....。

\*1 <http://xvproxy.local.io/c5/>

## 2.6 木村 陽一

Debian で Apache+SQLite+PHP な Web アプリ開発をしています。そろそろ C# や VB.NET もやりたいと思っていますが、Mono の VB.NET コンパイラは不安定で機能が十分でないので Windows 環境を VM に用意して開発しようと思っています。

## 2.7 dictoss(杉本 典充)

- web 閲覧、メール、プログラム開発は仕事も自宅も Debian GNU/Linux でやっている
- MIDI キーボード練習は Windows をひとまず使っている。(Linux でもできると思うがあまり調べていないため)
- ビデオ録画 (PT2 を使用) は FreeBSD でやっている。自宅 web サーバが FreeBSD だったので、それに相乗りする形でそのまま使い続けているから

## 2.8 Kiwamu Okabe

Haskell!Debian が最も適しています!

## 2.9 田原悠西

コンピュータを使ってやりたいことであり、やっていることは仕事とかインターネットを使ったりとか。Debian を使っています。

## 2.10 Arnaud Fontaine

I'm a Debian developer who arrived in Tokyo last year and interested in meeting up some other developers. I'm using Debian only at work and home.

## 2.11 Hirotaka Kawata

1. 趣味で Python で簡単なプログラムの作成や、アセンブラ (binutils) やコンパイラ (gcc) の新しいアーキテクチャへの移植。仕事で PHP, Rails や HTML, CSS をつかった Web プログラムの作成。遊びでゲームとか (フライトシミュレータなど)
2. できる

## 2.12 mitty

ファイル等のバックアップや重複管理をもっと楽にしたい。普段使いの PC とは別に、ファイルサーバ (NAS) として Ubuntu を用いているが、バックアップや世代化、ダウンロードしてきた iso イメージなどのファイル重複チェックなど、ほとんど適宜手動でやっているため、ある程度自動化出来ると嬉しい。

## 2.13 osamu@debian.org

Debian 文書の整理 (DDP の XETEX 対応)、内容更新 — 使用理由: 自明  
実際は私物の写真の編集・整理ぐらいが DEBIAN 「利用目的」で

す。 — 他のプラットフォームはすぐ消えていくが、いつも最新システムでいられるしコンパティビリティの心配がない。なんと言ってもお金がかからない。こんな所が理由です。

正直なところ、ブラウザを使う以外は意外と DEBIAN を使っています。Debian は少々荒削りですが、使いやすいのと、今までの習慣で使っています。最初は日本語サポートできるがベースが英語というのが理由でしたが、いまとなつては UBUNTU をなぜ使わないかとなります。まあ、あえていえば自分たちで作っているという気がするの理由ですかね。

## 2.14 pi8027

1. ● プログラム、証明、文書などの作成  
● 講義のノートを取る作業、レポートを作成する作業  
● 情報収集  
● 自分自身に関わる色々な物事の管理 (予定管理とか)  
● これらの作業の部分的もしくは完全な自動化
2. できます

## 2.15 taitioooo (長谷川)

家の KVM や Xenserver を管理できるセルフポータル整備しようとしています。Linux でできますが、debian でできません。Centos 使ってます。

## 2.16 野島 貴英

1. (a) 自分独自仕様のクールな GUI とクールな入出力デバイスを持つクールな PC/モバイル機を作って人に自慢する  
(b) とにもかくにもクールな爆安 PC/モバイル機を作って社会的 / 経済的に厳しいところとてにかく流行らせる。コンテンツ作らせる。お金まわす
2. いや、その。Debian(Linux) 以外だとかなり難易度高いっすよね?  
中年になってから、厨二病患ってます...

## 2.17 岩松 信洋

- やりたいこと
  1. DVCam を使った Debian での Ustream 配信。昔はできたんですけどね。サウンド周りを直していません。
  2. 完全自動クロスコンパイル環境の構築。今は一部のパッケージをネイティブでコンパイルされている必要がある。
- やっていること
  1. いつでも、どこでも、誰とでも Debian を使っています。
  2. インターネットを見るのも Debian だし、開発環境も Debian です。

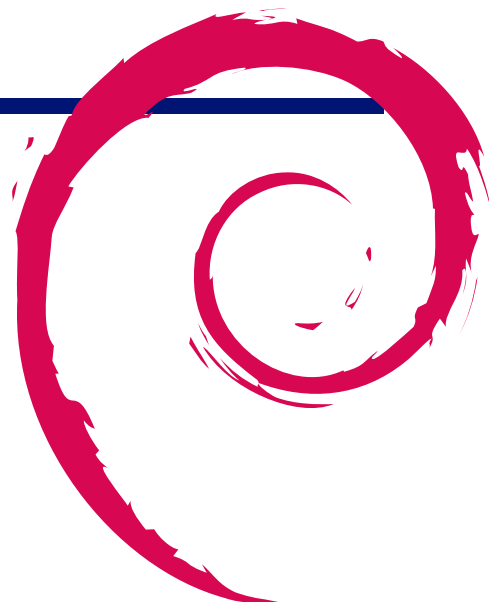
## 2.18 まえだこうへい

1. やりたいことは、自分や家族の生活を便利にすること
2. こまめ監視カメラなど、部分的には出来ているけど、まだ完全ではない。今の勤務先も CentOS や Ubuntu で、Debian は使っていない。前者は技術的 (ハードウェア) やリソース (要は自分の時間と金) の問題、後者は政治的、人的な理由  
Debian だけな生活を送るために、頑張るのです。

## 3 最近の Debian 関連のミーティング報告

前田 耕平

---



### 3.1 東京エリア Debian 勉強会 80 回目報告

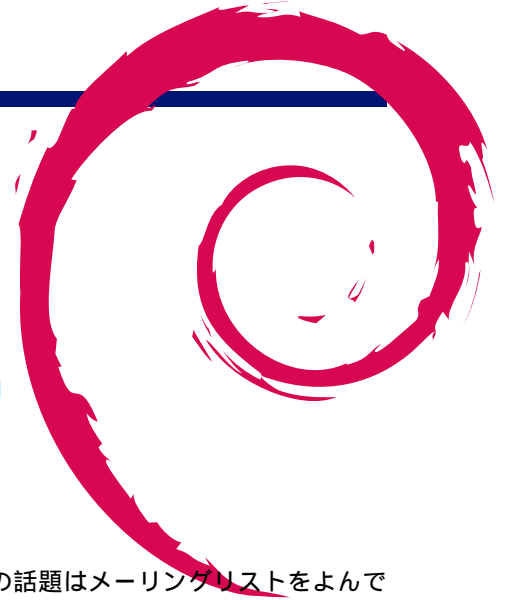
9 月の東京エリア Debian 勉強会は、最初の 3 連休の 1、2 日目の土日に伊東の山喜旅館さんで温泉合宿、通称 Debian 温泉 2011 を開催しました。参加者は 8 名。今年は 15 時にチェックインしてから、翌朝の 4-5 時まで黙々とやるガンバルマン大会になりました。吉田さんが差し入れしてくれた日本酒もあったのに、誰一人酒を飲まず、温泉も早朝寝る前か、7 時前の起床直後に入る、というストイックさ。温泉行っているのにもったいないですね。

各人それぞれの作業を行った以外、Mini DebConf についてのディスカッションも行いました。Mini DebConf については今後の進め方の方針も決まったので、良い成果を出せたと言えるでしょう。F2F でその場で聞けて、集中して作業&討議できる環境というのは重要ですね。

翌日は、温泉入って朝食後、伊東駅で解散しました。次回はどこでしょうか。お楽しみに。

## 4 Debian Trivia Quiz

前田 耕平



ところで、みなさん Debian 関連の話題においついていますか？ Debian 関連の話題はメーリングリストをよんでいると追跡できます。ただよんでいるだけでははりあがないので、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。

今回の出題範囲は `debian-devel-announce@lists.debian.org` や `debian-devel@lists.debian.org` に投稿された内容と Debian Project News などからです。

問題 1. 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 2. 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 3. 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 4. 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か？

- A Faidon Liambotis
- B Luca Filipozzi
- C Nobuhiro Iwamatsu

問題 5. Wheezy フリーズの予定はいつでしょうか？

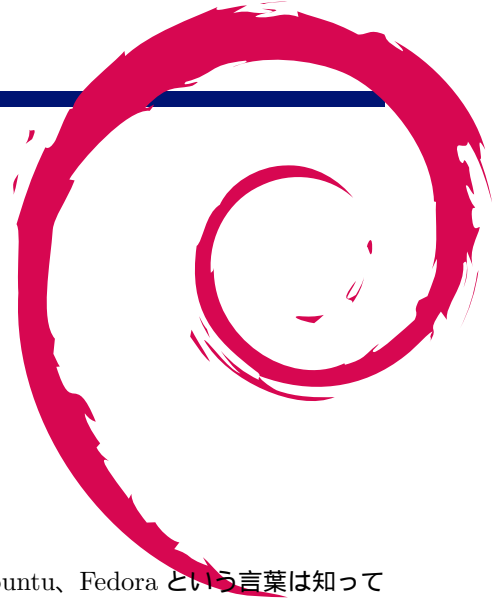
- A 2012 年 4 月
- B 2012 年 6 月
- C 2012 年 8 月

問題 6. 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

## 5 Debian とはなにか？

岩松信洋



今回は Debian 勉強会を筑波大学さんで行うということで、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と思います。簡単に Debian とは何なのかを簡単に説明します。

### 5.1 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。

### 5.2 特徴

#### 5.2.1 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。<sup>\*2</sup>

とはいえ、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。

<sup>\*2</sup> <http://www.debian.org/intro/free.ja.html> より

### 5.2.2 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。

### 5.2.3 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。

### 5.2.4 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\*3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。

### 5.2.5 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 Policy 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。

### 5.2.6 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。

### 5.2.7 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。

### 5.2.8 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 プロジェクトに連絡するとインフラを提供してくれるかもしれません。

---

\*3 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

### 5.2.9 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。

### 5.2.10 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も、更にプロジェクト間の結びつきが強いです。

### 5.2.11 派生しているディストリビューシヨンの多さ

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。有名どころでは、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\*4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。

## 5.3 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。

---

\*4 <http://distrowatch.com/dwres.php?resource=independence> 参照

## 5.4 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 をお勧めします。
- 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。

## 6 Haskell と Debian の辛くて甘い関係

岡部究



### 6.1 Haskell というプログラミング言語

Haskell <sup>\*5</sup> 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 <sup>\*6</sup> 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。。まあ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。なァに Debian なら簡単です。<sup>\*7</sup>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。

<sup>\*5</sup> <http://haskell.org/>

<sup>\*6</sup> [http://www.kotha.net/ghcguide\\_ja/latest/template-haskell.html](http://www.kotha.net/ghcguide_ja/latest/template-haskell.html)

<sup>\*7</sup> ここでは Debian Sid を使っていることを前提にしています。

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","hoge", "hoge", "hoge", "hoge", "hoge", "hoge", "hoge", "hoge"]
```

## 6.2 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 <sup>\*8</sup> 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 <sup>\*9</sup> というコマンドがあります。この cabal コマンドは Hackage の依存関係を考慮し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
```

## 6.3 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—————しょうがない、必要なパッケージだけ更新しましょう

<sup>\*8</sup> <http://hackage.haskell.org/>

<sup>\*9</sup> <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!?
```

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か。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。。

## 6.4 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か?それは cabal のしくみと Hackage 作者達の文化に問題があります。

### 6.4.1 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こと、Hackage のバージョン番号のポリシー文書<sup>\*10</sup>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。Hackage 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

<sup>\*10</sup> [http://www.haskell.org/haskellwiki/Package\\_versioning\\_policy](http://www.haskell.org/haskellwiki/Package_versioning_policy)

混乱することがあるようです。

#### 6.4.2 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。<sup>\*11</sup> この中のスライド<sup>\*12</sup> 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。

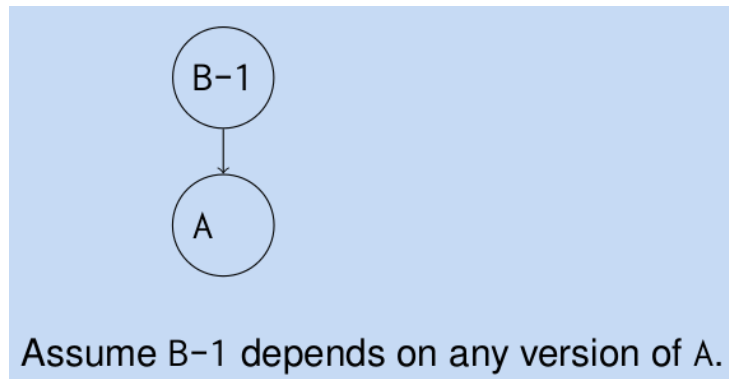


図1 Hackage DB 上で B-1 パッケージが A に依存している場合

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ません。

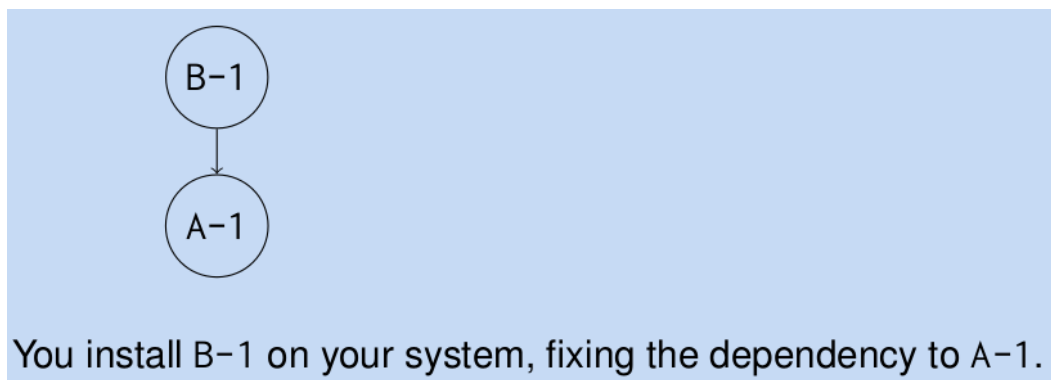


図2 B-1 と A-1 をインストール

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ています。

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

<sup>\*11</sup> <http://www.haskell.org/haskellwiki/HaskellImplementorsWorkshop/2011/Loeh>

<sup>\*12</sup> <http://www.haskell.org/wikiupload/b/b4/HIW2011-Talk-Loeh.pdf>

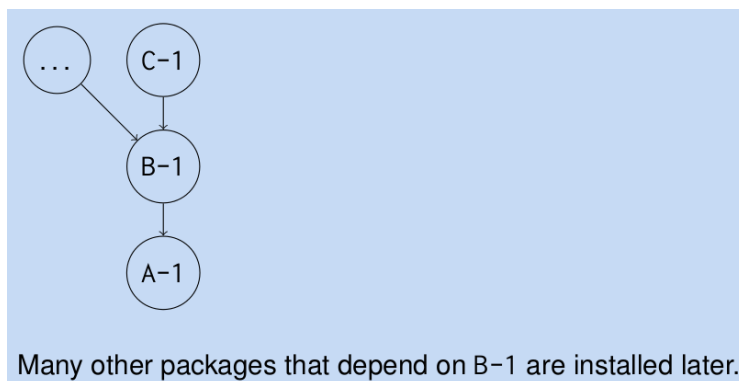


図3 そしてさらに B-1 に依存した Hackage 群をインストール

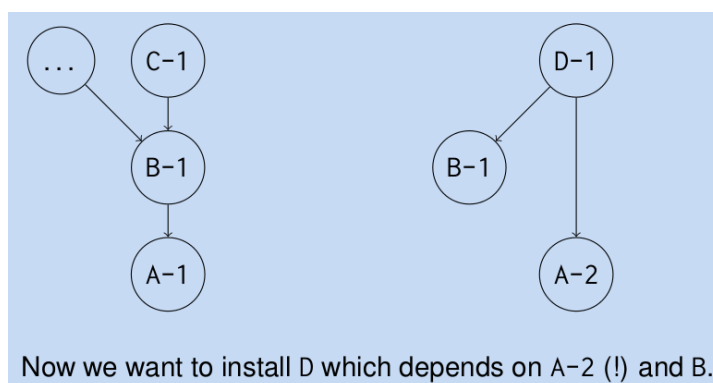


図4 A-2 に依存している D-1 をインストールしようと試みる

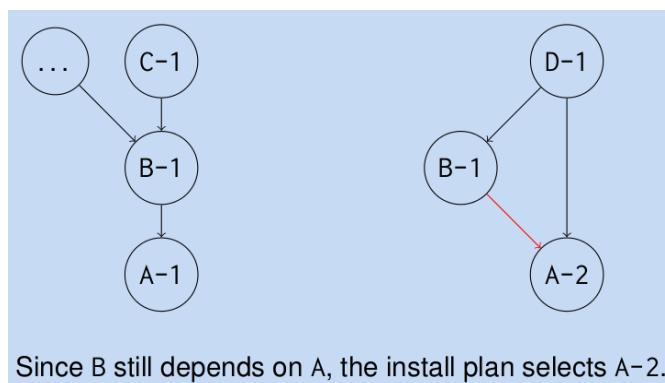


図5 cabal は D-1 をインストールにあたって A-2 もインストールしようとする

なります。

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。<sup>\*13</sup> 近い将来に本家 cabal に取り込まれることでしょう。

<sup>\*13</sup> <http://darcs.haskell.org/cabal-branches/cabal-modular-solver>

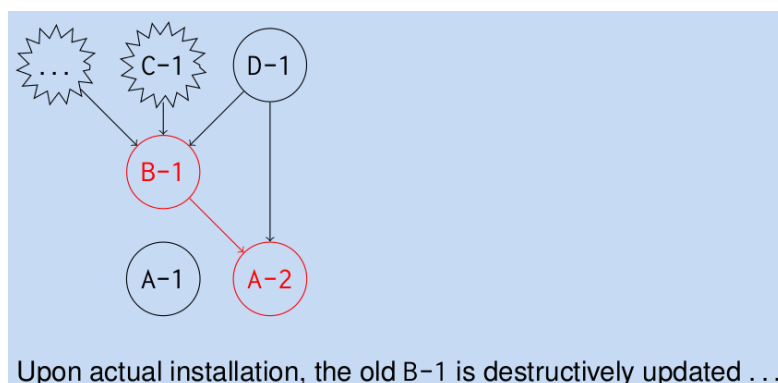


図 6 D-1 は正常にインストールされたが、B-1 に依存していた Hackage 群は依存が壊れてしまう

### 6.4.3 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<sup>\*14</sup> 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。<sup>\*15</sup>

### 6.4.4 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<sup>\*16</sup>, hakyll<sup>\*17</sup>, hamlet<sup>\*18</sup> 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か？ 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<sup>\*19</sup> 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か？ 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。

<sup>\*14</sup> <http://hackage.haskell.org/package/cairo>

<sup>\*15</sup> <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 ;)

<sup>\*16</sup> <http://hackage.haskell.org/package/yesod>

<sup>\*17</sup> <http://hackage.haskell.org/package/hakyll>

<sup>\*18</sup> <http://hackage.haskell.org/package/hamlet>

<sup>\*19</sup> <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<sup>\*20</sup> 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。

## 6.5 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。<sup>\*21</sup> さっそくやってみましょう!

例題として HCWiid <sup>\*22</sup> 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。

<sup>\*20</sup> <https://github.com/jaspervdj/hakyll/blob/master/src/Hakyll/Web/Template/Read/Hamlet.hs>

<sup>\*21</sup> <http://hackage.haskell.org/package/debian>

<sup>\*22</sup> 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 xzf 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 パッケージができてしまいました。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
$ 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

## 6.6 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。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、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、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 の流儀にのっとり 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。 :)

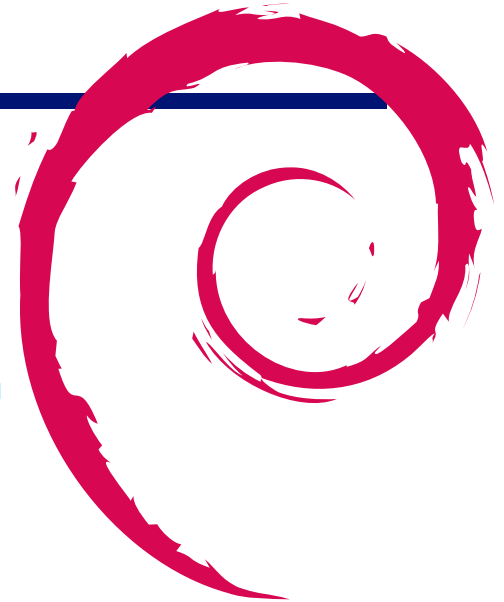
## 6.7 作ったパッケージを Debian に登録するには

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？

その秘技はプレゼン資料の方でこっそり、あなただけに、紹介します。 ;)

## 7 月刊 debhelper 第 1 回

岩松 信洋



### 7.1 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ず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<sup>\*23</sup>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。

### 7.2 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ずつ紹介し、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！

<sup>\*23</sup> 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 $@

```

### 7.3 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 7.3 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と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`



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。

## 7.4.2 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
```

## 7.5 今月のコマンド : dh\_testdir

### 7.5.1 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。

### 7.5.2 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
```

## 7.6 今月のコマンド : dh\_bugfiles

### 7.6.1 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ク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ロ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。

## 7.6.2 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  
....
```

## 7.7 次の発表者

次の発表者は 勉強会で発表します。選ばれた人、おめでとうございます。頑張ってください。



**Debian 勉強会資料**

2011年10月22日 初版第1刷発行

東京エリア Debian 勉強会（編集・印刷・発行）

---