

月刊

Debian 専

日本唯一のDebian専門月刊誌

2011 年 8 月 20 日

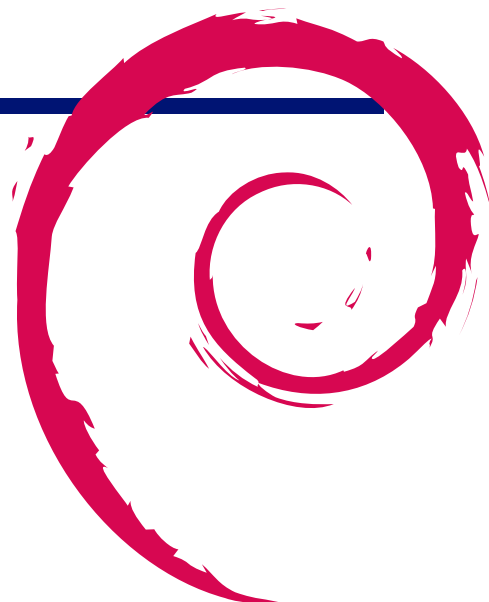
特集1: XXXX

特集2: YYYY



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 をがりがりとす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。

ビルド勉強会

目次

1	Introduction	1	5.7	ビルド 5 日目 - git-buildpackage - git と Debian の幸せな結婚	10
2	事前課題	3	5.8	ビルド 6 日目 - pbuilder - えっ、私のビルド、. . . 動かない?	11
2.1	鈴木崇文	3	5.9	ビルド 7 日目 - pdebuild でダイレクトビルド	12
2.2	キタハラ	3	5.10	ビルド 8 日目 - cowbuilder - 高速 pbuilder でビルド最終形へ	12
2.3	やまだ	3	5.11	ビルド 9 日目 - i386 on amd64	13
2.4	koedoyoshida	3	5.12	ビルド 10 日目 - cross-build on amd64	14
2.5	dictoss(杉本 典充)	3	5.13	ビルド 11 日目 - i386 build on amd64 pbuilder	14
2.6	上川純一	3	5.14	ビルド 12 日目 - cross build on amd64 pbuilder	15
2.7	Osamu MATSUMOTO	3	5.15	ビルド 13 日目 - cross build on amd64 pbuilder, better	15
2.8	なかおけいすけ	3	5.16	ビルド 14 日目 - 大統一ビルドの失敗 - pbuilder に lintian 統合	16
2.9	やまねひでき	4	5.17	まとめ	17
2.10	岩松 信洋	4	6	パッケージを作ったらスポンサーアップロード	18
2.11	吉野 (yy-y-ja-jp)	4	6.1	はじめに	18
2.12	yamamoto	4	6.2	スポンサーアップロードするときに確認する内容	18
2.13	taitioooo	4	6.3	アップロード	20
3	最近の Debian 関連のミーティング報告	5	6.4	まとめ	20
3.1	東京エリア Debian 勉強会 77 回目報告	5	7	LT: aufsbuilder - cowbuilder に たたかいをいどむ	21
3.2	東京エリア Debian 勉強会 78 回目報告	5	7.1	やってみた	21
4	Debian Trivia Quiz	6	7.2	つくりかた	21
5	パッケージビルド七転八倒 - rules から git-buildpackage まで	7	7.3	たたかってみる	22
5.1	はじめに	7	7.4	まとめ	23
5.2	ビルド 0 日目	7	7.5	だがしかし	23
5.3	ビルド 1 日目 - はじめの一步	8	8	索引	24
5.4	ビルド 2 日目 - dpkg-buildpackage と lintian	8			
5.5	ビルド 3 日目 - dpkg-buildpackage と git とソースパッケージ	9			
5.6	ビルド 4 日目 - debuild の存在意義に疑問を持つ	10			

2 事前課題

岩松 信洋



今回の事前課題は以下です:

Debian の各種ツール(apt とか dpkg とか固有のもの)について、

1. このツールの解説が欲しい!
2. これの状況はどうなってる?
3. こんなツールありませんか? (あったらいいのに)

と思っていることがあれば、それをできるだけ多く挙げて下さい。

この課題に対して提出いただいた内容は以下です。

2.1 鈴木崇文

apt のリポジトリ作成ってどうやるのでしょうか。ググったらなんかページが出てきてわかりそうですが、やったことないので書きました。あと、以前誰かが話していた apt のキャッシュサーバーって最近の動向はどうなのでしょう。

2.2 キタハラ

Debian 固有のものだと、今のところないですね。1 コマンド (又は GUI 操作) で USB メモリに Live 環境を作るツールとか? (探すとあったりして...、調べていません。)

2.3 やまだ

今後調べたい(解説歓迎)

- libapt-pkg-perl/python-apt などのパッケージデータベース API
- debhelper8 (簡単になったが奥行きがさらに増した、ような...)
- *.d.o なサイトを改良したくなったらどうすればいいか

緩募

- apt-build の ./configure の引数なども即いじれる版
- apt-get changelog 風の入れてないマニュアル等を読むツール

2.4 koedoyoshida

dpkg -i(apt-get install も同様) がエラーになったときの原因追及で困ったことがあったので、そのようなときの調査法を知りたい。 <http://www.flcl.org/~takasugi/tdiary-org/?date=20061023> に同現象があったのでとりあえずワークアラウンドは分かりましたが...

2.5 dictoss(杉本 典充)

debootstrap のおもしろい使い方ってあるのかなあ。(amd64 上で i386 環境が必要、常用環境とテスト環境の分離、くらいしか使ったことないです。)

2.6 上川純一

とくになし。

2.7 Osamu MATSUMOTO

ユーザーとしては特に困ってないです。 aptitude でしかできない事ってあるのかわかりません。開発者ツールまだまだ勉強中。

2.8 なかおけいすけ

- apt のオプションの解説

apt-get update, upgrade, install, remove, clean 位しか使ってないのでその他のオプションについて

- 推奨パッケージもインストールしてくれるオプション
これはきつとある
- apt.conf の解説
debian のページを見ていると、時々 apt.conf を修正してみたいな記述があるけれども、何をやっているのか、何ができるのか、解説が欲しい。
- apt-get と aptitude の違い

2.9 やまねひでき

debootstrap で xz がサポートされていると良いのかも。

2.10 岩松 信洋

ぱっとは思いつかないです。

2.11 吉野 (yy-y-ja-jp)

こんにちは。

2.12 yamamoto

最近 apt-get autoremove で、既に削除されたパッケージの、依存関係解決のために入れられたパッケージがごっそり削除できませんが、apt-get build-dep ホゲホゲで入れられたパッケージも同じようにビルド終了後に消せるといいよね。

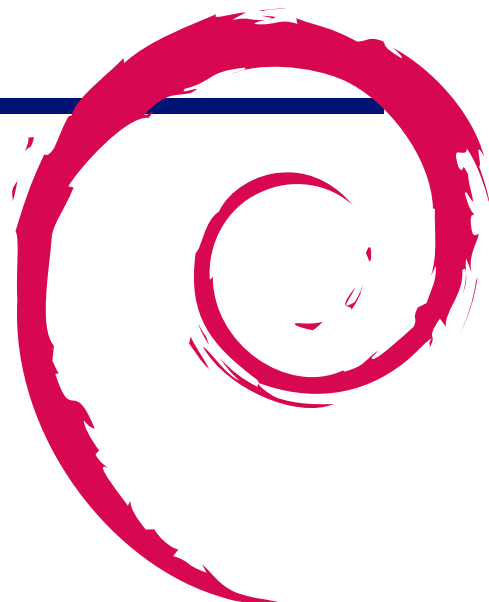
2.13 taitioooo

課題は未定ですが。debian で karesansui が使えるようになるといいですね~

あ、やればいいのか!

3 最近の Debian 関連のミーティング報告

岩松 信洋



3.1 東京エリア Debian 勉強会 77 回目報告

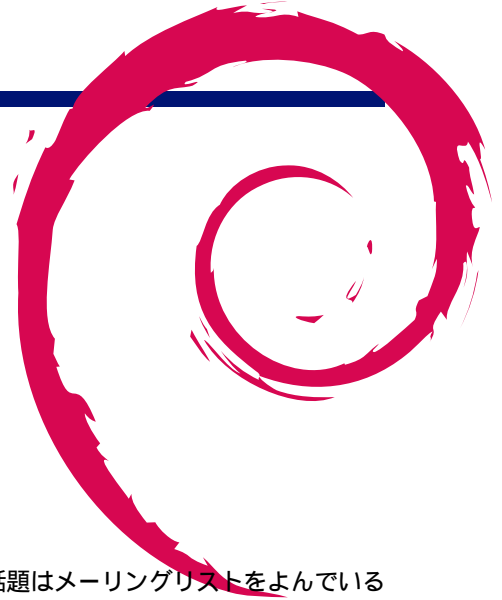
第 77 回 東京エリア Debian 勉強会 を東京オリンピックセンターで行いました。今回の参加者はやまださん、野首さん、MOTOHARA さん、hattorin さん、キタハラさん、吉田さん、杉本さん、emasaka さん、なかおさん、松本さん、吉野さん、山本さん、前田さん、野島さん、上川さん、岩松の 16 名でした。今回は Debian での文書処理紹介ということで、マニアックではありますが上川さんが XSLT の話を、前田さんが Sphinx と Doxygen の話をしました。Wiki フォーマットみたいなのがたくさんあって、覚えるのめんどくさいなと思った勉強会でした。

3.2 東京エリア Debian 勉強会 78 回目報告

第 78 回 東京エリア Debian 勉強会 をボスニア・ヘルツェゴビナで行いました。現地での参加者はやまねさん、野島さん、岩松で、IRC でたくさんの方が参加されました。レポートは別途。

4 Debian Trivia Quiz

岩松 信洋



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

問題 1. `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 2. `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 3. 新しくサポートされた圧縮形式は？

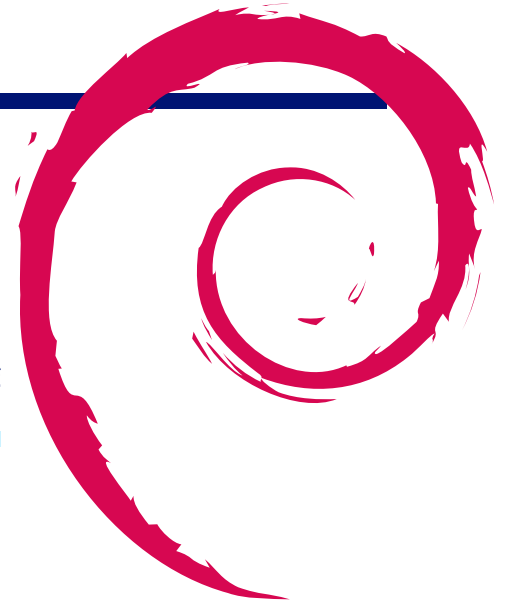
- A rar
- B cab
- C xz

問題 4. 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 5. 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。



5 パッケージビルド七転八倒 - rules から git-buildpackage まで

やまだ

5.1 はじめに

Debian パッケージを作成しようとして、関連するコマンドの層の厚さに戸惑ってしまった人はいないでしょうか？ 本稿はパッケージビルドを rules ベースから git-buildpackage まで一歩ずつ発展させることで、関わってくるコマンドチェーンへの理解を深めようとして書いてみました。

ところで、今回の内容は「パッケージビルド方法の紹介・講習」ではなく、

自分の手順を晒し、講評を貰って自分の糧とする

というのを目指しています。要するに

色々やってこういう手順になった、何かもやもや

とコメントしてもらって逆講習セッションとなります。ですので、晒している手順の中では失敗した過程や引っかかった内容と疑問もそのまま書いています。毎日の迷路での迷いぶりをぜひお楽しみ下さい。

別の方法でできている人は、別記事で手順をぜひ晒してみてください!!

5.2 ビルド 0 日目

今回の題材は cocot です。これはターミナル側の対応エンコーディングと実行コマンドの入出力エンコーディングが一致しない場合に

```
cocot -t UTF-8 -p EUC-JP program-with-euc-jp-only-support ...
```

のように実行すると UTF-8 対応コマンドのように使えるというスグレモノなのですが、今回の話題とは特に関係ないので紹介はこれくらいにします。

0 日目の今日は、まずソースの Debian パッケージ化を行います。

```
$ cd /d/src/  
$ git-clone git://github.com/vmi/cocot # ソースは git 公開のみ  
$ cd cocot  
$ dh_make -c bsd  
For dh_make to find the package name and version, the current directory  
needs to be in the format of <package>-<version>. 以下略
```

早速エラーとなりました。これはソースフォルダの形式が

```
<package>-<version>/
```


とバージョン情報を含むことが期待されているためです。が、git clone のフォルダにバージョン名をつけて運用するというのはイマイチです。さらに、これから自分が作りこむ debian/ 以下も git 管理したい訳なので、上の名前でもコピーを作って作業するのもしっくりこない。

なので、フォルダ名と関係なく開発・ビルドできる環境を作るために、従わずにこうします(しました):

```
$ dh_make -c bsd -n -p cocot_20100903 # --native --packagename を追加
Type of package: single binary, indep binary, multiple binary,
                  library, kernel module, kernel patch or cdfs?
[s/i/m/l/k/n/b] s <- 今回は cocot 単品なので single binary
...
Done. Please edit the files in the debian/ subdirectory now. cocot
uses a configure script, so you probably don't have to edit the Makefiles.
$
```

そして作られた debian/ フォルダ以下のサンプルファイルを削除・編集します。最終的に

```
$ ls debian/
total 44
4 README          4 changelog  4 compat    4 copyright  4 manpages  4 source/
4 README.source  4 cocot.1   4 control  4 docs       4 rules*
```

と整理して、ビルド準備が整いました。

5.3 ビルド1日目 - はじめの一步

それではまずは最もシンプルなビルド形態から。これは Makefile になっている debian/rules で binary ターゲットを実行するだけです:

```
$ fakeroot ./debian/rules binary
dh binary
...
dpkg-deb: building package 'cocot' in './cocot_20100903-1_amd64.deb'.
$ ls -d ../cocot*
4 ../cocot/ 20 ../cocot_20100903-1_amd64.deb
```

バイナリパッケージだけなら、これで終わりです。簡単ですね。fakeroot はパッケージ中のファイル所有者が root になるように併用されるものです。

しかし、このパッケージは Debian 品質にはなっていません。

- 生成されたバイナリパッケージは署名されていない
- ソースパッケージは生成すらされていない
- 構成が Debian Policy に準拠しているか未テスト

などの問題があるためです。これらは署名なら gpg、ソースパッケージ生成なら tar/diff/etc(はさすがに原始的過ぎるので普通は dpkg-source)、構成テストなら lintian があるわけですが、手動でやっていると面倒なので一括実行するツールが整備されているわけです。

5.4 ビルド2日目 - dpkg-buildpackage と lintian

というわけで、「バイナリのみ、しかも未署名未テスト」の状態を脱するため、debian/rules の直接実行ではなく、ビルドツールを使ったパッケージビルドに進みます。

使う道具は dpkg-buildpackage です。何はともあれまず実行:

```
$ dpkg-buildpackage -b # まずは binary-only ビルド
...
dpkg-deb: building package 'cocot' in './cocot_20100903-1_amd64.deb'.
dpkg-genchanges -b > ../cocot_20100903-1_amd64.changes
dpkg-genchanges: binary-only upload - not including any source code
signfile cocot_20100903-1_amd64.changes
...
$ ls -d ../cocot*
4 ../cocot/ 20 ../cocot_20100903-1_amd64.deb
4 ../cocot_20100903-1_amd64.changes
```

今度は deb に加えて changes も生成されました。ちゃんと署名しているのも確認できますね。changes は主にパッケージリポジトリで deb を管理するために使われるもので、(バイナリ)アップロードの際は deb と changes をペアでリポジトリにコピーします。

ちなみに、上では構成テストはされていません。これは dpkg-buildpackage はパッケージ生成に特化しているためです。

```
$ lintian ../cocot_20100903-1_amd64.changes
W: cocot: new-package-should-close-itp-bug
```

dpkg-buildpackage を使う場合はこのように別途テストします。出てくるエラーメッセージはタグ名になっており、-i オプションで詳細解説を表示させたり、また、lintian.debian.org でも同様に細かく知ることができます。

ただ、何が警告されているかは詳しく書かれていても、直し方の例などがなく、解釈に悩むこともあります。「よくある失敗と直し方」のような情報を付けられればと、lintian/l.d.o の表示の横にコメント領域を作りたくなります。

ここまでが、バイナリのみ配布する場合(最小限の)ビルド手順になります。

5.5 ビルド3日目 - dpkg-buildpackage と git とソースパッケージ

さて、ここまでソースパッケージの生成はしていませんでした。これは今回の設定が

普通の git ツリーで Debian package を作る

のため、tarball の場合には存在しない .git/ フォルダの扱いを考慮する必要があるためです。要は、ソースパッケージから .git フォルダは外したい訳です。

このあたりの問題を完全に解決し git との親和性を高くするのが後述の git-buildpackage なのですが、それを使うまでは以下のような方法で対処していました:

```
$ git-branch upstream # 素のソース保存用ブランチを作っておく
$ git-checkout master # debian/ コミの自分の開発用ブランチを master にする
$ git-add debian/ # debian/ を追加
$ git-commit -a #
```

このように素のソースツリーと Debian 化した自分の開発用ツリーでブランチを分けた上で、

```
$ git-archive --prefix=cocot-20100903/ upstream \
| gzip > ../cocot_20100903.orig.tar.gz
```

などとして素のソースツリーのみ orig.tar を生成できます。後は

```
$ dpkg-buildpackage -S # ソースパッケージのみビルド
...
dpkg-source: info: building cocot using existing ../cocot_20100903.orig.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.debian.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.dsc
signfile cocot_20100903-1.dsc
...
signfile cocot_20100903-1_source.changes

$ ls -ld ../cocot*
4 ../cocot/ 4 ../cocot_20100903-1_source.changes
4 ../cocot_20100903-1.debian.tar.gz 216 ../cocot_20100903.orig.tar.gz
4 ../cocot_20100903-1.dsc
```

で完成します。署名もされていますね。ちなみに、テストもできます:

```
$ lintian ../cocot_20100903-1_source.changes
$
```

意味があるかは不明ですが・・・。

とにかく、これでソースパッケージも生成できるようになったので、

```
$ git-archive --prefix=cocot-20100903/ upstream \
| gzip > ../cocot_20100903.orig.tar.gz
$ dpkg-buildpackage # 無指定なので dpkg-buildpackage -b -S の両方する
```

でようやく debian.org に出せるようなまっとうなパッケージビルドに届いたことになります。

5.6 ビルド 4 日目 - debuild の存在意義に疑問を持つ

ここまで ./debian/rules、dpkg-buildpackage とビルド方法を経てきましたが、次はその更に上位のコマンドとなる debuild を使ってみます。

debuild の dpkg-buildpackage に対する優位性は

- コマンド名が短くて楽
- lintian も実行してくれるので楽
- 署名に debsign や debrsign を使うことができ、柔軟
 - 例: ~/.gnupg/ のある生活環境と、パッケージビルドする作業環境が別サーバなど

というあたりだと思われます。

コマンドチェーンを長くする程の価値があるか、微妙

という感想を捨てきれませんが、この更に上位のコマンド群は debuild を呼ぶので、地位は安泰と思われます。何はともあれ実行:

```
$ git-archive --prefix=cocot-20100903/ upstream \  
| gzip > ../cocot_20100903.orig.tar.gz  
$ debuild  
...  
dpkg-source: info: building cocot using existing ./cocot_20100903.orig.tar.gz  
dpkg-source: info: building cocot in cocot_20100903-1.debian.tar.gz  
dpkg-source: info: building cocot in cocot_20100903-1.dsc  
...  
dpkg-deb: building package 'cocot' in './cocot_20100903-1_amd64.deb'.  
dpkg-genchanges >../cocot_20100903-1_amd64.changes  
...  
Now running lintian...  
W: cocot: new-package-should-close-itp-bug  
Finished running lintian.  
Now signing changes and any dsc files...  
signfile cocot_20100903-1.dsc Taisuke Yamada <tai@rakugaki.org>  
...  
signfile cocot_20100903-1_amd64.changes Taisuke Yamada <tai@rakugaki.org>  
...  
Successfully signed dsc and changes files  
$
```

たしかに dpkg-buildpackage よりは少し楽ではあるのですが・・・

5.7 ビルド 5 日目 - git-buildpackage - git と Debian の幸せな結婚

さて、いよいよ git-buildpackage にまでやってきました。これまでは

```
$ git-archive --prefix=cocot-20100903/ upstream \  
| gzip > ../cocot_20100903.orig.tar.gz  
$ debuild
```

のように orig.tar は自分で用意してから debuild などと呼んでいたのですが、git-buildpackage はこの手順を統合します。

- ブランチ A は素のソース管理用で、これを元に orig.tar を生成できる
- ブランチ B は Debian 化されたツリーで、これを使ってビルドする

という構成を理解して処理してくれる形です。この他にもビルド完了時にタグを打ってくれたり(-git-tag)、細かい機能が色々あります。

何はともあれ実行:

```

$ git-buildpackage --git-upstream-branch=upstream --git-debian-branch=master
...
cocot_20100903.orig.tar.gz does not exist, creating from 'upstream'
dpkg-buildpackage -rfakeroot -D -us -uc -i -I
...
$ ls -d ../cocot*
4 ../cocot/ 4 ../cocot_20100903-1_amd64.changes
8 ../cocot_20100903-1.debian.tar.gz 20 ../cocot_20100903-1_amd64.deb
4 ../cocot_20100903-1.dsc 208 ../cocot_20100903.orig.tar.gz
12 ../cocot_20100903-1_amd64.build

```

キタ! 以降は dpkg-buildpackage を呼んでいる通り、これまでと同じです。
 で、上では引数を長々と書いていますが、使うブランチ名がデフォルトで upstream/master なので、そのように git リポジトリを運用していれば

```
$ git-buildpackage
```

のみでビルド完了になります。

5.8 ビルド6日目 - pbuilder - えっ、私のビルド、・・・動かない?

ここまで各種ビルド方法をやってきて、git-buildpackage で一里塚に到達した感がありますが、

そのパッケージ、本当に他の環境で使えますか?

という点が実はまだ甘いままです。

- パッケージメンテナの環境にしかない非互換のライブラリ
- パッケージメンテナの環境にしかないカスタマイズされたビルドツール

こういったものに依存するバイナリやビルドプロセスが残っていると、折角アップロードしたパッケージもただのゴミファイルになってしまいます。

これをクリアするためのツールが pbuilder です。これは

- 独立した debootstrap 環境を作り、
- その中にパッケージファイルをコピーして、
- 最後に chroot してビルド

をすることで、パッケージメンテナの環境に依存するビルドミスを撲滅します。

まずは実行:

```

# まずは普通に git-buildpackage でビルド
$ git-buildpackage
$ ls -d ../cocot*
4 ../cocot/ 4 ../cocot_20100903-1_amd64.changes
8 ../cocot_20100903-1.debian.tar.gz 20 ../cocot_20100903-1_amd64.deb
4 ../cocot_20100903-1.dsc 208 ../cocot_20100903.orig.tar.gz
12 ../cocot_20100903-1_amd64.build

# 完成物を pbuilder 環境内でビルド。chroot が絡むので root 権限が必要
$ sudo pbuilder --create # 初回のみ行う(中で debootstrap する)
$ sudo pbuilder --update # 次回からは更新の上、処理に入る
$ sudo pbuilder --build ../cocot_20100903-1.dsc
...
I: extracting base tarball [/var/cache/pbuilder/sid-amd64/base.tgz]
...
I: Copying source file
I: copying [../cocot_20100903-1.dsc]
I: copying [../cocot_20100903.orig.tar.gz]
I: copying [../cocot_20100903-1.debian.tar.gz]
...
dpkg-source: info: building cocot using existing ../cocot_20100903.orig.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.debian.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.dsc
...
dpkg-deb: building package 'cocot' in '../cocot_20100903-1_amd64.deb'.
dpkg-genchanges >../cocot_20100903-1_amd64.changes
...

```

このように、chroot 環境内でリビルドを試みて、ビルドできることの検証と正しいバイナリの生成を行います。このように pbuilder は頑健なパッケージ作成のためにお勧めのツールですが、いくつか欠点もあります。

- 大掛かりになるため遅い
- root 権限が必要になる
- lintian が自動実行されない
- 署名処理も自動実行されない

本質的に解決が難しい部分もあるので、普段は debuild/git-buildpackage を使い、最終チェックに pbuilder といった使い分けをしています。また、欠点をカバーするための上位ツール群も登場しています。

5.9 ビルド 7 日目 - pdebuild でダイレクトビルド

前回は

- ソースツリーから git-buildpackage が debuild でビルド
- ビルド結果中のソースパッケージを元に pbuilder がリビルド

という 2 段階実行になっていましたが、考えてみると

それなら最初から chroot 環境にソース置いてビルドすれば?

という疑問が浮かびます。これを実現するのが pdebuild です。正確には

pdebuild に debuild と同じ CLI I/F を持たせ、入れ替える

ことで、git-buildpackage が .git/ の処理をした後で、後は最初から pdebuild にビルドさせる方式です。pdebuild はオプションで debsign も呼び出してくれるので、そこも統合できます。上の方で debuild の存在意義に疑問を持っていましたが、このあたりの処理 I/F の基準化という意味合いもあるのでしょうか?

```
$ sudo pbuilder --create # 初回のみ行っ 中で debootstrap する)
$ sudo pbuilder --update # 次回からは更新の上、以下続行
$ git-buildpackage --git-builder="pdebuild --buildresult .. --auto-debsign"
...
cocot_20100903.orig.tar.gz does not exist, creating from 'upstream'
...
dpkg-source: info: building cocot using existing ./cocot_20100903.orig.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.debian.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.dsc
dpkg-genchanges -S >../cocot_20100903-1_source.changes
...
I: extracting base tarball [/var/cache/pbuilder/sid-amd64/base.tgz]
... 最後に debsign も呼び出される...
signfile /d/src/cocot_20100903-1.dsc Taisuke Yamada <tai@rakugaki.org>
...
signfile /d/src/cocot_20100903-1_amd64.changes Taisuke Yamada <tai@rakugaki.org>
...
```

再び git ツリーからのパッケージビルドをほぼ一括でやれるようになりました。ちなみに root 権限が一見不要に見えますがそんなはずもなく、pdebuild はデフォルトで sudo -E を使って内部で pbuilder を昇格させて処理しています。

なお、Debian Planet 調べでは git-pbuilder なる更に上位のラッパーが開発途上の模様です。このコマンド階層はどこまで深くなるのか・・・

5.10 ビルド 8 日目 - cowbuilder - 高速 pbuilder でビルド最終形へ

前回で git と pbuilder の統合が果たされましたが、pbuilder の欠点はまだそのまま残っています。この決정을軽減するのが cowbuilder です。

- ベースの debootstrap 環境を tar.gz で保持せず展開状態で管理する

- 上記をビルド環境として展開する際、コピーではなくハードリンクを用いる
 - さらに、書き込みはトラップして、ベース環境に書き戻されないよう保護する

という手法で、処理時間を劇的に短縮します。

```
$ sudo cowbuilder --create # 初回のみ行う
$ sudo cowbuilder --update # 次回からは更新処理をしてから続行
$ PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder="pdebuild --buildresult .. --auto-debbuild"
...
dpkg-source: info: building cocot using existing ./cocot_20100903.orig.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.debian.tar.gz
dpkg-source: info: building cocot in cocot_20100903-1.dsc
dpkg-genchanges -S >../cocot_20100903-1_source.changes
...
-> Copying COW directory
... 以降は pbuilder が chroot/COW 環境下でビルドを行う...
```

使い勝手はほとんど同一で

- pbuilder を使うところを cowbuilder とする
- pdebuild が cowbuilder を使うよう PDEBUILD_PBUILDER=cowbuilder と定義する

のみで倍速近くになるため、非常に有用です。

この形が、現在のパッケージビルド方法の最終形になるようです。しかし本稿はまだ続きます。

5.11 ビルド 9 日目 - i386 on amd64

上の git-buildpackage + pdebuild + cowbuilder でビルド手順としてはようやく完成を見ました。しかし、

ネイティブビルドだけで、いいんですか？

という課題があります。

アップロードすればそのうち i386 や他アーキテクチャ用のパッケージも出来てくる訳ですが、自分で使う自分のパッケージなら、自分でビルドもしてしまいたいものです。アーキテクチャ依存バグを潰すこともできます。

まずは、より簡易な i386 on amd64 ビルドを見てみましょう。これは gcc レベルでは

- gcc を gcc -m32 で呼ぶ
- 依存ライブラリがあれば、その i386 build 版も用意しておく

で済みますが、これをパッケージビルド内でする方法は意外に悩みました。バイナリが i386 なのにパッケージ名が amd64 だったり、その逆が起こったり・・・

というわけで、各ビルドツールレベルでの指定方法をまとめてみました:

```
# debian/rules レベルの場合
CFLAGS=-m32 \
linux32 dpkg-architecture -ai386 -c fakeroot ./debian/rules binary

# dpkg-buildpackage レベルの場合
DEB_CFLAGS_APPEND=-m32 linux32 dpkg-buildpackage -ai386

# debuild レベルの場合
DEB_CFLAGS_APPEND=-m32 linux32 debuild -ai386

# git-buildpackage レベルの場合
git-buildpackage \
--git-builder="DEB_CFLAGS_APPEND=-m32 linux32 debuild -ai386"
```

CFLAGS の指定方法が途中で変わるのは、dpkg-* 系のツールはビルドオプションは dpkg-buildflags が供給するものを使うためです。

アーキテクチャ指定については、dpkg-architecture が出力する DEB_(BUILD_HOST)* 変数で処理する形が標準で、linux32 のエミュレーション(uname(3) の結果をいじるだけ)は実質的に意味がなくなっています。が、お守り代わりでまだ付けたり付けなかったり。

Debian Policy のどこかに

告知: linux32 はもう obsolete です

と書いてあったりしないのかな・・・？

5.12 ビルド 10 日目 - cross-build on amd64

今回は ARM や MIPS などのツールチェーンがまったく別の場合のクロスパッケージビルドです。cocot については使う予定はないのですが、通信系のツールなどではたまに自分でビルドして欲しくなります。

実は i386 on amd64 より、まったく別のアーキテクチャ向けのビルドをする方が簡単です。

```
# debian/rules レベルの場合
dpkg-architecture -aarmel -c fakeroot ./debian/rules binary

# dpkg-buildpackage レベルの場合
dpkg-buildpackage -aarmel

# debuild レベルの場合
debuild -aarmel

# git-buildpackage レベルの場合
git-buildpackage --git-builder="debuild -amipsel"
```

当然クロスコンパイラや依存ライブラリのクロス版が入っていることが前提となりますが、Debian の場合は apt-cross がこのためにあり、容易に環境を作ることができます。

5.13 ビルド 11 日目 - i386 build on amd64 pbuilder

ここまでのクロスビルドはコンパイラのモード切替やクロスコンパイラで行っていましたが、pbuilder でもできるので
しょうか？

答は「できる」で、むしろ chroot 環境でのビルドは、その環境下でのネイティブビルドとも言えるので更に単純です。

ただし、debootstrap の処理の都合上、こちらは i386/amd64 の方が other/amd64 より楽です。

まずは i386 on amd64 pbuilder から:

```
$ export DIST=sid ARCH=i386 DISTRIBUTION=sid ARCHITECTURE=i386
$ sudo -E pbuilder --create # 最初の一度だけ行う環境構築
$ sudo -E pbuilder --update # 二回目からは更新だけ行う
$ git-buildpackage --git-builder="pdebuild --buildresult .. --auto-debdesign"
```

微妙に DIST=sid ARCH=i386 と DISTRIBUTION=sid ARCHITECTURE=i386 と同じ事を二重に指定しているのが不審ですが、これは pbuilder と pdebuild で ARCH/DIST 情報の伝達・参照方法が異なるためです。なお、マニュアルでは

```
pbuilder --architecture arch --distribution dist
pdebuild --architecture arch
```

というのがありますが、これは現状あまり有効ではありません(この辺りもやもやしている)。環境変数の方がコマンドチェーンを下っても有効になっていやすいので、現在は環境変数べったりに倒してます。

高速化で cowbuilder を組込む場合、pbuilderrc に追加が必要な点のみ要注意でしたが、ビルド自体は変数設定を1つ増やすだけで済みます:

```
$ vi /etc/pbuildererrc
...
# for cowbuilder
BASEPATH=/var/cache/pbuilder/$DIST-$ARCH.cow
...
$ export DIST=sid ARCH=i386 DISTRIBUTION=sid ARCHITECTURE=i386
$ sudo -E cowbuilder --create
$ sudo -E cowbuilder --update
$ PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder="pdebuild --buildresult .. --auto-debdesign"
```


5.14 ビルド 1 2 日目 - cross build on amd64 pbuilder

だんだん話も終わりに近づいてきました。今度は i386 ではない他アーキテクチャのビルドを pbuilder/cowbuilder on amd64 で行ってみます。

他アーキテクチャを扱う上で面倒な点は以下の 2 つです:

- debootstrap が一発で chroot 環境を構築できない
- chroot 環境内のバイナリは amd64 環境で直接実行できない

これはいずれも chroot 環境さえ作ればいいので、一番手っ取り早い方法は qemu-**-static* エミュレータを使って debootstrap してしまうことです:

```
$ cd /var/cache/pbuilder/
$ sudo debootstrap --variant=buildd --include=apt,cowdancer \
--foreign --arch armel sid sid-armel.cow http://localhost:9999/debian
...
$ sudo cp /usr/bin/qemu-arm-static sid-armel.cow/usr/bin/
$ sudo chroot sid-armel.cow /debootstrap/debootstrap --second-stage
...
I: Base system installed successfully.

# 上の --second-stage 実行でクリアされてしまう sources.list を再補充
$ sudo tee sid-armel.cow/etc/apt/sources.list.d/local.list
deb http://127.0.0.1:9999/debian sid main
^D
$ sudo chroot sid-armel.cow apt-get update
```

これは欠点として pbuilder/cowbuilder `--create` で作った環境ほどパッケージ構成が最適化されていないようですが、これで

```
$ export DIST=sid ARCH=armel DISTRIBUTION=sid ARCHITECTURE=armel
$ sudo -E cowbuilder --create
$ sudo -E cowbuilder --update
$ PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder="pdebuild --buildresult .. --auto-debdesign"
```

とアーキテクチャを問わず同一手順でビルドできるようになりました。

もっとも、qemu-user-static は確実に動作するとは限らないので、アーキテクチャによって動くような、動かないようなという面はあります(MIPS ですら実は動かないことも...)。

もし cowbuilder ではなく pbuilder で実行する必要があるならば、単に tarball にして sid-armel/base.tgz に置けば転用できます:

```
$ sudo mkdir sid-armel
$ cd sid-armel.cow
$ sudo tar zcf ../sid-armel/base.tgz *
```

なお、`http://localhost:9999/debian/` は approx で立てているキャッシングプロキシですが、設定が簡便なのと効果絶大なためオススメです。

5.15 ビルド 1 3 日目 - cross build on amd64 pbuilder, better

今回は他アーキテクチャ用の chroot 環境を手動 debootstrap で作りましたが、pbuilder/cowbuilder の生成する環境とは差が出てしまうため、やはり通常の

```
pbuilder/cowbuilder --create
```

で生成したい所です。そこで、クロス環境もコマンド一回で構築できる debootstrap-cross を用意して、

```
pbuilder/cowbuilder --create --debootstrap /usr/local/bin/debootstrap-cross
```

と呼び出してみましょう。スクリプトは単純で、以下の通りです:


```
#!/bin/sh

archfix() {
  case "$1" in
    armel) echo arm;;
    amd64) echo x86_64;;
    *)     echo $1;;
  esac
}

scanarg() {
  while [ $# -gt 0 ]; do
    case "$1" in
      --arch=)
        ARCH=${1##--arch=};;
      unstable|testing|stable|oldstable|sid|wheezy|squeeze|lenny)
        set -- "$@"; SUITE=$1; TARGET=$2; MIRROR=$3; SCRIPT=$4; break;;
      esac
    shift
  done

  : ${ARCH:=$(dpkg-architecture -qDEB_HOST_ARCH)}
  : ${SUITE:=sid}
  : ${TARGET:=.}
  : ${MIRROR:=http://127.0.0.1:9999/debian}
}

scanarg "$@"

if dpkg-architecture -e$ARCH; then
  debootstrap "$@"
elif dpkg-architecture -eamd64 && [ $ARCH = i386 ]; then
  debootstrap "$@"
else
  debootstrap --foreign "$@"
  cp "/usr/bin/qemu-${archfix $ARCH}-static" "$TARGET/usr/bin/"
  chroot "$TARGET" /debootstrap/debootstrap --second-stage
  #echo $MIRROR > "$TARGET/etc/apt/sources.list.d/local.list"
  #chroot "$TARGET" apt-get update
fi
```

これは amd64 上で amd64/i386 環境を構築するケースも 2 段インストールしない本来の形で処理できるので、
/etc/pbuilderrc に

```
DEBOOTSTRAP=/usr/local/bin/debootstrap-cross
```

と書くことで、再び手順は

```
$ vi /etc/pbuilderrc
...
# for cowbuilder
BASEPATH=/var/cache/pbuilder/$DIST-$ARCH.cow
DEBOOTSTRAP=/usr/local/bin/debootstrap-cross
...
$ export DIST=sid ARCH=i386 DISTRIBUTION=sid ARCHITECTURE=i386
$ sudo -E cowbuilder --create
$ sudo -E cowbuilder --update
$ PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder="pdebuild --buildresult .. --auto-debstrip"
$ lintian ../mypackage*.deb
```

とアーキテクチャを問わず同様となります。

5.16 ビルド 14 日目 - 大統一ビルドの失敗 - pbuilder に lintian 統合

ここまで、素の debian/rules ビルドから一歩ずつ進めて、

```
debian/rules → +dpkg-buildpackage → +debuild → +git-buildpackage → +pbuilder → +pdebuild →
+cowbuilder → +i386 build support → +generic cross build support
```

と 1 つずつ機能を組み込んできました。しかし、pbuilder 以降は lintian が統合から外れている事に気付かれていますでしょうか？

これには理由がある(と思われる)のですが、組み込み自体は実は可能です。pbuilder(そしてそれを利用している cowbuilder)にはフック機能があり、各種処理の前後で任意のスキ립トでフックできます。

実は既に /usr/share/doc/pbuilder/examples/ に用意されているのですが、これを利用してビルド完了直後に lintian を呼んでみましょう。

```

$ mkdir /t/pbhook/
$ cp /usr/share/doc/pbuilder/examples/B90lintian /t/pbhook/
$ PDEBUILD_PBUILDER=cowbuilder \
git-buildpackage --git-builder="pdebuild \
--buildresult .. --auto-debstrip -- --hookdir /t/pbhook"
...
I: user script /d/cache/pbuilder/build/cow.13001/tmp/hooks/B90lintian starting
...
Get:19 http://127.0.0.1/debian/ sid/main lintian all 2.5.2 [617 kB]
...
Setting up lintian (2.5.2) ...
Generating en_US.UTF-8 locale for internal Lintian use....
+++ lintian output +++
W: cocot: new-package-should-close-itp-bug
...
+++ end of lintian output +++
I: user script /d/cache/pbuilder/build/cow.13001/tmp/hooks/B90lintian finished
...

```

これでビルド + 署名 + 検証を完全実行するビルド手順の大統一が再び果たされました。

・・・が、やってみると判るのですが、ネイティブビルドならまだしも、クロスビルドの時にエミュレータ上で lintian を実行するのは正直遅いです。上の Generating locale ... の辺りでかなりぐんにやり。

結局、餅は餅屋で

- テストビルドをネイティブの debuild 環境でやりつつ lintian も連動実行
- 最終的なクリーンビルド・バイナリ生成は pbuilder 環境
- 署名はせずに、アップロードまたは独自リポジトリ設置のタイミングで実行

と、むしろ統一しないほうがいい、というのが結論です。

ここまで引っ張っておいてそれが結論か!

どうぞ皆様は同じ過ちは犯されませんように・・・

5.17 まとめ

今回は debian/rules による単純な（といっても、この中身は debhelper という別の闇が控えています・・・）ビルドから始めて、git-buildpackage での git 統合、さらに pbuilder/cowbuilder でのクリーンビルド、そしてさらにクロスビルドまで含んでの手順統一を行ってきました。

すべてを勘案した、現在のビルドの最終形はこうなっています:

```

=== /etc/pbuilder/errc ===
...
# for cowbuilder
PDEBUILD_PBUILDER=cowbuilder
if echo $SUDO_COMMAND | grep -q qemu; then
  BASEPATH=/var/cache/pbuilder/$DIST-$ARCH.qemu
else
  BASEPATH=/var/cache/pbuilder/$DIST-$ARCH.cow
fi

# このコマンドの内容はビルド13日目を参照
DEBOOTSTRAP=/usr/local/bin/debootstrap-cross
...

```

この設定でマルチアーキテクチャ対応の pbuilder/cowbuilder 環境が整うので、後は

```

$ export DIST=sid ARCH=i386 DISTRIBUTION=sid ARCHITECTURE=i386
$ sudo -E cowbuilder --update # たまに行っておく。初回だけ --create になる
$ git-buildpackage --git-builder="pdebuild --buildresult .."
$ lintian ../${basename $PWD}*.deb
$ debstrip ../${basename $PWD}*.changes ../${basename $PWD}*.dsc

```

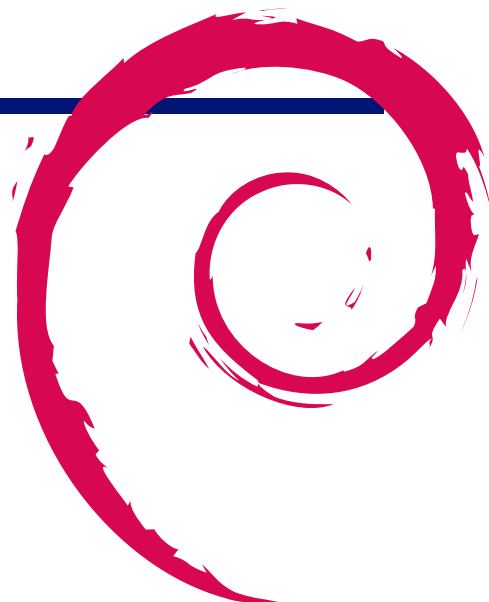
を実行、という形に落ち着いています。

この先には qemu-builder という大物や、アップロード/野良リポジトリ運営との統合などがテーマとしてまだあるのですが、今日の所はこれくらいで締めさせていただきます。それでは

Happy Building!

6 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



6.1 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が、

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。

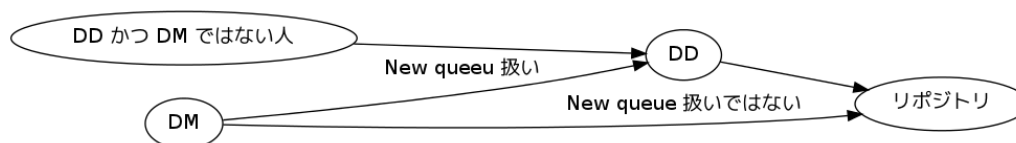


図 1 パッケージアップロード

6.2 スポンサーアップロードするときの確認する内容

パッケージメンテナに「アップロードして! 」と言われてはいはいと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。

6.2.1 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 に入ってい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 新メンテナガイド^{*1} を読んだか。
- DFSG ^{*2} を読んだか。
- Debian Policy^{*3} を読んだか。
- Debian Reference ^{*4} 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。

6.2.2 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/copyleft 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 ^{*5} 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をお願いします。
- orig.tar.gz の確認
オリジナルの tar ボールと一緒に、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^{*6}(cowbuilder^{*7}) と、sbuid^{*8} 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 sbuid を使って

^{*1} <http://www.debian.org/doc/manuals/maint-guide/>

^{*2} http://www.debian.org/social_contract.ja.html

^{*3} <http://www.debian.org/doc/debian-policy/>

^{*4} <http://www.debian.org/doc/manuals/developers-reference/>

^{*5} <http://dep.debian.net/deps/dep5/>

^{*6} <http://packages.qa.debian.org/p/pbuilder.html>

^{*7} <http://packages.qa.debian.org/c/cowdancer.html>

^{*8} <http://packages.qa.debian.org/s/sbuild.html>

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^{*9} 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^{*10} 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、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^{*11} 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。

6.2.3 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 cdb^s 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。

6.3 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。

6.4 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。

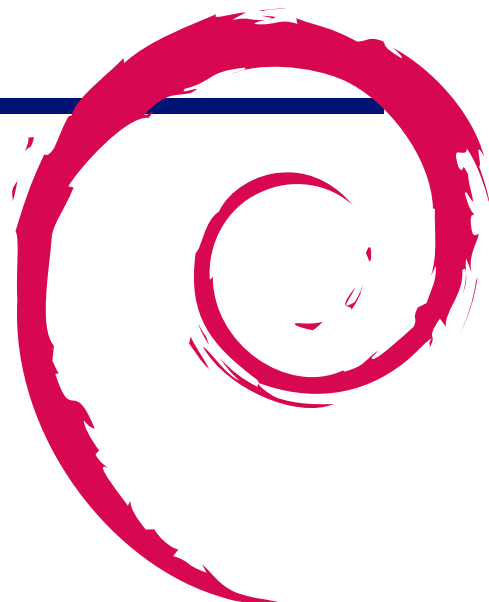
^{*9} <http://packages.qa.debian.org/l/lintian.html>

^{*10} <http://dep.debian.net/deps/dep3/>

^{*11} <http://packages.qa.debian.org/p/piuparts.html>

7 LT: aufsbuilder - cowbuilder にたたかいをいどむ

やまだ



7.1 やって見た

数年前に aufs がマイブームだった時、

これで aufsbuilder 書いたら cowbuilder に勝てるんじゃないかね?

と思ってやって見たものの、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。

7.2 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。

以下コード:

```
#!/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" "$@"
```

7.3 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

7.4 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。

7.5 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。

8 索引

sponsor upload, 18



Debian 勉強会資料

2011年8月20日 初版第1刷発行

東京エリア Debian 勉強会 (編集・印刷・発行)
