

Debian 専

日本唯一のDebian専門誌

2010 年 12 月 31 日 初版発行

東京エリアデビアン勉強会

関西デビアン勉強会



2010年冬号

あんどきゅめんとつど
でびあん



東京エリアDebian勉強会
関西エリアDebian勉強会著

会 勉 強 会 ビ ー ア ー ト

目次

1	Introduction	2
2	ある Debian な一日 その一	3
3	ある Debian な一日 その二	5
4	ある Debian な一日 その三	7
5	ある Debian な一日 その四	8
6	Debian Conference 2010 参加報告	10
7	OSC Tokyo 2010/Fall 参加報告	12
8	puppet に \$HOME を管理させてみよう	13
9	ext4 ファイルシステムを Debian で活用してみる	21
10	NILFS を Debian で活用してみる	22
11	Btrfs を Debian で活用してみる	37
12	分散ファイルシステム CEPH を Debian で活用してみる	40
13	initramfs について	45
14	最近の Debian Live の動向	49
15	Debian Backports の使い方	55
16	dh ~source format 3.0, the magic debhelper rules~	58
17	Emdebian について	66
18	Debian GNU/kFreeBSD で暮らせる環境を構築してみる。	71
19	Debian Miniconf 計画検討	79

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。

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 ある Debian な一日 その一

上川純一

2.1 始まり

朝起きる、ワシントン DC で買ってきたスターバックスのマグカップに湯を注ぎココアをいれる。まだ外は暗い。おもむろにマシンを起動する。SSD ベース^{*1}の vaio type P はしずかに起動する。grub は Linux パーティションを自動起動するように設定されており、ひとしきり Linux の起動メッセージが画面に流れたあと GD のログイン画面が登場する。ユーザ名とパスワードを入力。画面はきりかわり、いくつかのアプリケーションが自動で起動する。pidgin, google-chrome, emacs 22, gnome-terminal。

毎日ではない、でも週に一回はこの日がくる。今日はバックアップをとる。USB 経由で接続できる 2.5 インチディスクを接続し、ディスクがスピニングするのを待つ。backup.sh を実行すると LVM ボリュームのスキャン、ext3 ファイルシステムのマウントと pdumpfs の実行処理がはしり、かりかりとバックアップがとられる。定期的なバックアップ作業の一環として、ディスクの使用量を確認する。df -i で inode の残量、df -h でディスク容量の残量を確認する。まだ余裕があるようだ。

```
$ df -i
/dev/mapper/vgvaio-vaiohome
 5242880 417994 4824886 8% /home
/dev/mapper/vgusb1-coreduohome
4587520 1116140 3471380 25% /mnt
$ df -h
/dev/mapper/vgvaio-vaiohome
 79G 64G 12G 85% /home
/dev/mapper/vgusb1-coreduohome
 69G 46G 21G 70% /mnt
```

メールを取り込む。メールは ssh 経由の独自のパッチプロトコルで取得する。これは高レイテンシネットワーク (Air HTM) をつかっていた時代からの名残りだ。そして M-x wl で wanderlust を立ち上げる。

2.2 開発環境のメンテナンス

開発環境をメンテナンスする。

まず、パッケージを最新にする。apt-get update, apt-get upgrade, apt-get dist-upgrade を行う。手元の環境はときにより stable だったり testing だったりする。最近 vaio type P では stable を使っている。

開発用につかっている chroot イメージをアップデートする。cowbuilder --update で最新のパッケージをダウンロードしていく。chroot 環境は sid を基本としている。

あと他にも chroot 環境がいくつかあったりする。cowbuilder ではない環境も用意してある。/home をバインドマウントしている chroot ディレクトリがあり、chroot_sid.sh コマンドでその環境に入れるようにしてある。自作の

^{*1} 内蔵ディスクが HDD ではなくて SSD のモデルがある。

jchroot コマンドを使い現在の実行しているユーザ、現在のディレクトリと同じ場所、で chroot 内部の bash が立ち上がるようになっている。標準の chroot コマンドだけだといまいちがゆいところに手が届かないのと、dchroot コマンドの実装がいまいち気に入らないから自作してしまった。

手元の apt ミラーとして approx が動いているので、同じパッケージを複数の chroot 環境で必要となっても大丈夫だ。

2.3 バグレポートの処理

パッケージのメンテナンスをする、主となるのはバグレポートの対応だ。バグレポートは <http://bugs.debian.org/> のウェブインタフェースで確認できる。手元で欲使うのはメールインタフェースだ。バグ報告はメールとして送られてくる。

emacs をたちあげ、wanderlust を立ち上げる。バグレポートにパッチが添付されていれば、それを適用する。手順としては、“e” でメールをファイルとして保存し、そのファイルをソースコードのディレクトリに移動して git am で適用する。今日は pbuilder のバグ処理をしているので、pbuilder のソースツリーのディレクトリに移動し順にパッチを適用していく。

残念ながらこの一番シンプルなフローにしたがってくれるバグ報告は少ない。ほとんどのバグ報告はパッチつきではないし、ついていたとしても手直しなしで適用できるものではない。そういう場合はいろいろといじってから git commit することになる。

2.4 テストとリリース

パッケージをビルドしてテストするスクリプトは自動化している。emacs lisp で書いてあり、emacs からビルドとテストが実行されるようになっている。cowbuilder --build を実行して、すべて成功したら日付別になっている「成功したファイル置き場」においてくれるというもの。

管理しているパッケージはすべて debian/pbuilder-test/ ディレクトリにテストスクリプトがあり、そのスクリプト全部が正常に実行できれば基本的な動作はできていると考える。メンテナンスのコストを軽減するためにはテスト重要。

正常な動作をするパッケージが生成できるソースコードが準備できたら、denbian/changelog を整備する。まず、git dch -a で changelog の雛形を生成して、編集。debian-changelog mode をつかって、C-c C-e C-c C-d unstable C-c C-c.

git commit して、git-tag.sh というスクリプトを使って署名つきのタグを作成。前出の elisp を使って最終版のパッケージを cowdancer 環境でビルドする。debsign コマンドでできたパッケージに署名、その後 dput でアップロード。

しばらく放置していると ftp master から無事に accept されたというメールがくる。これを受けてリポジトリの内容を git push --tags, git push する。

新しいパッケージがこれでリリースできた。

3 ある Debian な一日 その二

やまねひでき



ある初夏の日の朝の話。

寒い。夏の初めに似合わぬ事を思いながら目を覚ます。まだ目覚ましもなっていない。どうやら掛け毛布が剥ぎ取られた上に扇風機がいつの間にか回されているせいだと気づく。やれやれ。

喉が多少痛むので昨日のうちに淹れておいた水出しコーヒーで湿らせる。とにかく温かいものを、と思い余ったカレーパンをレンジで温める。ホットコーヒーにした方が良かったかな、と寝ぼけた頭でぼんやり考える。レンジから音がしたので取り出して頬張る。駅構内のチェーン店にはいける味。だが同じ駅構内でも赤羽のアレはダメだ。

頭が動かないのでシャワーを浴び、ようやく人心地に。まだエアコンが取り付けされていない自室へ行き、デスクトップ PC が轟音を立っているなか、ノート PC をスリープから立ち上げる。もちろん立ち上がってくるのは Debian だ。gdm のパスワード入力に答え、雑然としたデスクトップの状態を眺める。さて、メール処理からやってみるか。sylpheed の画面を開く。昨夜でびあん傘の注文を受けて返事を書いたが、まだ送信してないのに気づいたので一気に送信。今回は何名か GPG encrypt してくれたが、sylpheed からサクッと処理が出来ずにローカルの適当なファイルにコピーしてターミナルから `gpg --decrypt なんぞ` をやっている、とほほ。相変わらず「手間を減らす」事が下手なのに嫌気が。ついでに Google Docs に注文状況のメモ書きをまとめる。ふと inbox を見るとスイスのその傘の首謀者からメール。郵送に EMS を使ってほしかったが、「それ聞いたことが無い」とのこと。彼が示した URL から EMS の国別状況のページが見つかったので、そちらをメール。うまくいくといいが。

事務処理はまだ続く。Debconf10 にいく事は決めたのだが、チケットの都合上、早く行くことになってしまった。宿を取らないと...と思って HIS で適当なのを予約したが、よくよく聞くと Debconf 会場でも支払いさえすれば前乗りできる様子。ということで NY の締切りに間に合うように `penta.debconf.org` から登録情報を修正し、SPI ヘクレジットで振込処理。ふー。HIS の方は Debconf 側が取れたのを見計らってキャンセルだな。

まだ事務処理。昨年寄稿した Software Design 誌の Debian JP サイトへの再利用許可が `gihyo` 方面から許可を出していただけたので、理事会に共有しておく。実際にウェブページへの反映はいつやるうかね...。Debconf 記事の売り込みも返信ついでにやっておく。旅費が、ね。それから、Debconf といえば GPG キーサインパーティ。今年の手順はこれだぞ！と岩松先生から送られてきていたので、登録作業を実施。`http://people.debian.org/~anibal/ksp-dc10/ksp-dc10.html` を見ながら、「あれー俺のキーっていくつ？」などとフザけたことを思い、`gpg --list-keys` して見つけるなど。pub key が分かればあとはページどおりに進めるだけ。一応 port25 ブロックの影響で届かないとか嫌なので自分宛にもメール、届いた。

まだメール処理。identi.ca で Gregkh 先生に「あなたの twitter クライアント (bti)、OAuth 対応してる？」とダイレクトで質問してたのに返答がきてた。「まだなんだよねー、わかってんだけど」とのこと。squeeze のフリーズと twitter の OAuth 以外拒否が近いので、対応していないクライアントは一旦ドロップしないといけない。リストを簡単にアップデートしておく。とりあえず、キリがなさそうなので一旦メール処理終了。

バグ潰しに入る。といっても自分のパッケージのバグではなく、FTBFS なバグ。Lucas Nussbaum さんが大量に登録してくれているので、彼の登録したバグをみていくことにしている。昨日のうちにめぼしいものは pbuilder

を使ってローカルでビルドするなどしてリストアップしてあるので、後は処理メールだけ。17 個ほど同じ原因の RC bug に対して <bugnumber>-done@bugs.debian.org 宛にメール。1 個は手元の pbuilder で再現しないので unreproducible タグをつけて control@bugs.debian.org へ。もう 2 個はエラーが指し示すとおり Build-Depends を微調整するだけでなおるので、ビルドできる事を確認してから patch タグをつけて送る。これで 20 個ほどバグが減る方向へ進んだわけだ。残りがこれくらい簡単なのばかりだと楽なんだが。

4 ある Debian な一日 その三

山本浩之

ある晴れた日曜の一日

ぐったりとして、朝、目覚める。まずモニタをつけて昨日の debuild の確認。む、なんか gcc-4.4 がビルドエラーしてるな。どうやら symbols の不整合で、最後の最後に、lib32gomp1 パッケージを組めずにエラー吐いているらしい。そういえば gcc-4.3 から ppc64 のパッチは投げられてなかったな。さて、どうしよう。よし、てきとーに、lib32gomp1.symbols.ppc64 にコピペしてみるか。お k。debuild 開始。

(んで、MythTV で録画しておいた ONEPIECE を見て、モニタを消して、寝る)

夕方、再度目覚める。ああ、よく寝た。む、また lib32gomp1 パッケージを組めずにエラー吐いたな。ええい、くそ、こうなったら libgomp1.symbols.common にコピペだ。よし、お k。debuild 開始。

(んで、MythTV で録画しておいた龍馬伝を見て、モニタを消して、寝る)

以上、ある晴れた日曜の一日でした。

5 ある Debian な一日 その四

まえだこうへい



5.1 起き抜けの一発

ドスン! 「グヘッ」

鳩尾にいつもの衝撃が目覚める。こまめが朝の餌をヨコセと、今朝もやってきた。無視して寝直す。ヨメの方の掛け布団の上の方に移動したようだ。タイマーでめざましテレビがついた。さて、起きるか。こまめも同時に”チャチャン”という鈴の音を立てて先回り。こまちゃんの餌をやる前に MacBook の電源を入れる。起動させている間にこまめの餌と水をやる。

顔を洗い、着替えを手にしてサーバールームへ。起動した MacBook の Sid にログインし、`apt-get update`; `apt-get upgrade` を実行し、その間に執筆中の本の原稿を `git pull` する。終わったら、バナナを食べて 6:33 の始発のシャトルバスで出かける。夏に電車のダイヤ改正に合わせて、シャトルバスのダイヤも変わってしまったので、通勤中に座れなくなってしまった。無論、通勤中には MacBook を使えないので、`milestone` で RSS フィードの購読をする。

5.2 始業前の一時間

8 時過ぎに会社のビルに到着後、朝食を調達し、朝一の小便を済ませ、会社のリフレッシュルームで原稿を書く。この一時間弱が一日で唯一のプライベートタイムだ。Debian 勉強会で発表をするときも、ここで資料を作成する。

5.3 始業後の日課

9 時になりオフィスの自席で、Sid の入ったデスクトップ PC を立ち上げる。仕事での検証は専らこの貧弱なマシンで行う。Sid の CouchDB は未だ 0.11 なので `svn` のリポジトリから `git-svn` で最新のリビジョンを持ってきてビルドする。おっと、今日は CouchApp のリポジトリも更新されているようだ。 `git pull` 後に、`deb` パッケージ化してアップデートしておこう。

5.4 昼休みのメンテ

昼休み、今日は組合サーバにログインする。表のサーバは未だ Sarge なのだが、Sarge の APT は Proxy サーバの NTLM 認証に対応していないので仕方ない。Sarge の背後で動かしている Squeeze のパッケージアップデートだけを行った。

5.5 帰宅後のこまめ、家事

帰宅後、着替えて、こまめの餌と水をやる。今日はワシが食事当番なので、いつもの納豆料理をする。作り終わる直前にちょうどヨメが帰宅。今夜も美味しくできたな。

食後にヨメが食器洗いをしている間に OOo で管理している家計簿をつけながら、メールのチェック。Debian パッケージのセキュリティ通知があるのに気づき、OpenBlockS の Lenny のパッケージをアップデートし、Tripwire の DB を更新しておいた。そろそろ OpenBlockS も Squeeze にアップグレードしないとあかんあ。ぷらっとホーム提供のファームウェアは Linux Kernel 2.6.16 ベースなので Squeeze にはそのままアップグレードできない。クロスコンパイル環境を作って、ファームウェアのビルドを計画しなければ。そんなことを思うが、やはり夜は頭回らない。さっさと寝ることにしよう。



6 Debian Conference 2010 参加報告

やまねひでき

6.1 DebConf とは

年一回世界中の Debian 開発者と関係者が集まって開かれるテクニカルカンファレンス「Debian Conference」のことです。通常顔をあわせることのないメンバーたちが世界各地から一同に介し友好を深め、技術的な議論を戦わせます。

6.1.1 Debconf の歴史・経緯

過去の開催履歴を見てみると表 1 のようになります。

表 1 歴代の Debconf 参加者推移

年	名前	場所	参加人数
2000	debconf 0	フランス ボルドー	
2001	debconf 1	フランス ボルドー	
2002	debconf 2	カナダ トロント	90 名
2003	debconf 3	ノルウェー オスロ	140 名
2004	debconf 4	ブラジル ボルトアレグレ	150 名
2005	debconf 5	フィンランド ヘルシンキ	200 名
2006	debconf 6	メキシコ オアスタベック	300 名
2007	debconf 7	英国スコットランド エジンバラ	約 400 名
2008	debconf 8	アルゼンチン マルデルプラタ	約 200 名
2009	debconf 9	スペイン エクストラマドゥーラ	約 250 名
2010	debconf 10	アメリカ ニューヨーク	約 350 名
2011	debconf 11	ボスニア・ヘルツェゴビナ バニャ・ルカ	? 名

6.1.2 Debconf 2010

今年の DebConf、DebConf10 (<http://debconf10.debconf.org/>) はアメリカ・ニューヨークのコロンビア大学で開催されました。日本からは、岩松 信洋、鈴木 崇文、やまねひできの 3 名が参加しました。

6.1.3 会場

コロンビア大学キャンパスにカンファレンス会場としてトークルームに 2 室、BOF ルームに 1 室、AdHoc セッション用に 1 室、それから昼夜を問わずに作業する人のためのハックラボが 2 室設けられました。NYC という大都会の大学キャンパス（すぐ側に 24 時間運転の地下鉄駅がある）での開催ということもあり、会場周りは昼夜を問わずかなり賑やかな状態でした。ニューヨークと言うと危険なイメージがあるようですが、怖いのは街中を歩いてもものすごく体格のいい人が多いこととタトゥーな人が多いことぐらいです。

今回のネットワークは、コロンビア大学の屈強なバックボーンへ GuruPlug などを使った即席手作り感満載のネットワークが接続されてました。無線が非常に繋がりにくいなんのって…。また、Debian のミラーサーバについては、前年のスペインでは DNS を乗っ取ってローカルのミラーを見せるようにしていたようですが、今回はコロンビア大学のネットワークを利用している為にそういう訳にもいかず、cdn.debian.net を使うのはどう？ ディレクトリ構成が違うからコロンビア大学は加えられない 変えたよ！ cdn に追加したよ！という感じで最終的に cdn.debian.net の改善に繋がったりしました。この後も色々 cdn.debian.net には要望が出たり、Debian Live の標準 apt line に加えられていたりといい感じでした。荒木さんに拍手。

6.2 スケジュールとイベント

今回は 8 月 1 日から 7 日が開催期間で、間の 1 日が Day Trip として Coney Island というニューヨークの端のビーチでのんびり & マイナーリーグの観戦という形でした。例によって DebConf 期間の前に 1 週間 DebCamp という開発合宿も開かれていたようです。

今回イベントとして「RCBC(release critical bugs contest、<http://wiki.debconf.org/wiki/DebConf10/RCBC>)」と題し、皆で RC(Release Critical) バグを潰そう、というものが開かれました。これによって 100 個以上のバグが一気に修正され、多くのバグを潰した人には懸賞として GuruPlug や HP の elite book やネットブック、書籍類などが進呈されていました。この期間中に岩松さんが 3 個、私は 1 個潰しましたので書籍を頂きました。でも英語なんですよね…。

6.3 セッション

今年は大きくトラックカテゴリとして「コミュニティ」「Java」「サイエンス」「エンタープライズ」「メディア & アート」が設けられ、それぞれに合わせたセッションが開催されました。で、肝心の内容ですが、これを書く時間が無い & もう Software Design 2010 年 10 月号で書いた & ウェブから大半のセッションが見れますよ、なので SD とウェブで見てください…。サイトは <http://www.debianart.org/live/> です。

Debian が多くの派生ディストロの「Hub」として振る舞うことを意識した話で「Derivatives Front Desk」を設けて派生ディストリビューション間での交流を進めていくことや、特に Canonical サイドからなるべく Debian にフィードバックしようという話が盛んに出ていたことが印象に残ります。Canonical としては自前で変更点を保持していくのがコスト高なのが懸念点なのでしょう (Nexenta も Solaris 対応のパッチ維持が大変なので、という発言をしていました)。

それから Stable に対する見直しの話が出ています。安定版をより使いやすいものにするために方針を緩めて「BTS 上で important 以上のバグ修正」「FTBFS(failed to build from source, ソースからビルドができない問題)の修正」「セキュリティ勧告がでていない致命的ではないがセキュリティ上の修正」をポイントリリースで取り入れていくそうです。

6.4 次回の Debconf

来年はボスニアのバニャルカが開催地です。政府の全面バックアップがあるようなので Free Beer に期待。

7 OSC Tokyo 2010/Fall 参加報告

まえだこうへい



2010 年 9 月 11 日 (土) に、OSC Tokyo 2010/Fall^{*2}が春と同じく明星大学にて開催されました。今回は、セッション 2 つとブース展示です。

セッションは

- GPG キーサイン説明および GPG キーサインパーティ^{*3}
- でびあん らんだむとびっくす^{*4}

で、前者は岩松さんが、後者はやまねさんが発表を行いました。

GPG キーサインパーティには岩松さんを含め 10 名の参加^{*5}があり、GPG キーサインの説明の後、その場でキーサインが行われました。キーサイン自体が初めての方も半数くらいいて、参加者のうち大半はその後ブースにも立ち寄ってくれました。

らんだむとびっくすは、Debian 17 執念、Debconf10、Squeeze の話でした。参加者は 20 名を超えて、資料は文字がほとんどなく、後から資料だけを見ると大抵さっぱりと思われる内容ですが、こちらも盛況でした。

ブースでは、Debian 関連の書籍と、Squeeze on OpenBlockS 600 の展示、Debian へのコメントを一言、を行いました。一言もらった方にはぐるぐるステッカーを無償で差し上げる、という試みを行いました。また Debian ではなく Ubuntu しか使ったことが無いですが何か？という人がかなり多かったのですが、そういう人に対しては Ubuntu に対する意見でも構わない、ということを伝えたのでコメント記入の敷居が多少下がったようでした。

終了後、やまねさん、山本さん、まえだの三名で高幡不動のとんかつ和幸で打ち上げ&反省会を、その後タリーズに移動して、Miniconf の開催についてのディスカッションを行いました。^{*6}

^{*2} <http://www.ospn.jp/osc2010-fall/>

^{*3} <http://www.ospn.jp/osc2010-fall/modules/eguide/event.php?eid=47>

^{*4} <http://www.ospn.jp/osc2010-fall/modules/eguide/event.php?eid=9>

^{*5} 事前登録は 12 名。

^{*6} 今回のネタにリンクする、というわけです。

8 puppet に \$HOME を管理させてみよう

倉敷悟



今回は、皆さんの工具箱に入っていると便利な puppet というツールをご紹介します。

8.1 はじめての糸繰り

早速ですが、まずは動かしてみてもらおうと思います。lv をネタに使うので、lv の設定を ~/.lv に書いている人は、一旦適当に退避してください。

念の為、進む前に ~/.lv が存在しないことを確認しておきましょう。

8.1.1 実行サンプル

なにはともあれ、puppet をインストールします。

puppet-el は、emacs でシンタックスハイライトさせるためのものなので、emacs を使わない人はなくても OK ですが、なしでは割とやってられません。

```
$ sudo aptitude install puppet puppet-el
$ emacs site.pp
```

エディタが起動したら、site.pp には次のように入力してください。site.pp は糸繰りの譜面、つまり設定ファイルです。なお、# はコメントなので無視して頂いて構いません。

```
# sudo が必要
package { "lv":
  ensure => installed,
}

file { ["/home/yourname/.lv":
  content => "-c",
}
```

入力が終わったら、保存してコマンドに戻ります。では、いよいよ puppet に踊ってもらうことにします。ネットに接続できるなら、aptitude purge lv してからでもいいでしょう。

```
$ sudo puppet site.pp
```

何が起きたか確認してみてください。予想通りすぎて、ちょっと拍子抜けかも知れませんが……。

8.2 概要

8.2.1 用語の整理

先に進む前に、用語の紹介だけしておきます。

site.pp に入力してもらった設定のことをマニフェストと呼び、ここに、必要なリソースを宣言することで、puppet を操っていきます。

先ほど書いてもらったマニフェストでは、package と file がそれぞれリソースの宣言になります。puppet が理解できるリソースはあらかじめ決まっていますが、自分で新しくリソースを作ることができます。

リソースの構成要素と、基本的な書式は次のようになります。区切りの記号に注意してください。

- タイプ：リソースの種類
- タイトル：「その」リソースの名前
- 属性：様々なパラメータ

```
タイプ {
  "タイトル":
    属性 => 値,
    属性 => 値,
    ...
    属性 => 値;
  "タイトル":
    ...
}
```

8.3 puppet の概要

ここで一旦手を休めて、puppet そのものについて、もう少し説明しておきます。

puppet が何か、を一言でいえば、システム構成の管理ツールということになります。普段、sudo を使ってやるような「パッケージ操作」「/etc 配下の設定変更」「プロセスの起動停止」といった作業内容を、事前にレシピを書いておくことで、puppet に任せることができます。

8.3.1 構成

puppet は、だいたい次のような構成で動きます。

- puppet：マニフェストを実行するコマンド
- puppetd：puppetmasterd にマニフェストをリクエストして、それを実行するデーモン
- puppetmasterd：マニフェストの束をもっていて、puppetd のリクエストにあわせて適切に配布するデーモン

実は、puppet を使う場合は、puppetmasterd と puppetd の組み合わせで多数のホストを集中管理するのが普通だったりします。

ですが、今回はそれは一旦おいておいて、puppet コマンドを使ったお手軽な方法をベースに進めていきますのでご承知置きください。

最後でもご紹介しますが、普通の構成についても既に山ほど記事や資料があるので、興味がある方はそちらをご参照頂ければと思います。

8.3.2 ディレクトリ配置

puppet が使用するディレクトリですが、典型的にはこんな感じですが、設定でどうしても変えられるので、配置されるものの種類がこんな感じなのだと思います。

実際のところ、puppet コマンドを使う場合は、`/etc/puppet` ではなくて `~/.puppet` が主な作業場所として想定されます。

- `/etc/puppet/` : puppet 自体の動作設定ファイルが配置されます
- `/etc/puppet/manifests` : マニフェストが配置されます
- `/etc/puppet/templates` : テンプレートファイルが配置されます
- `/etc/puppet/modules` : puppet module が配置されます
- `/var/lib/puppet` : マニフェストのキャッシュや、SSL 証明書が配置されます

ここで、最初に作ったマニフェストを、`~/.puppet/manifests` に移動させておいてください。puppet を実行した時点で、`~/.puppet` は作成されているはずですが。

8.4 設定を増やす

では、実践に戻ります。lv だけでは寂しいので、他にもレシピを追加してみることにします。ちょっと時間をとるので、自分的に必須なツールなど、適当に書いてみてください。emacs でも screen でも何でも OK です。

なお、マニフェストを書く上での注意として、「変化」「遷移」ではなく「状態」を定義するのだ、ということを頭の片隅においておくようにしてみてください。例えば、「emacs をインストールすること」ではなく、「emacs がインストールされていること」というイメージです。

8.5 リソースの整理

設定をどんどん追加していくと、`site.pp` にずらずらとリソースが並んで、大変見通しが悪くなってきます。そこで、リソースをグルーピングすることにしましょう。

puppet で用意されているリソースのグルーピング方法には、ノード、クラス、デファイン、モジュール、といったものがあります。

8.5.1 ノード

ノードは、ホストの `fqdn` と関連づけられるリソースのグループです。「この `fqdn` のホストに、このリソースをまとめて反映しといて！」というイメージです。

実際の定義はこんな感じになります。

```
node myhost01.localdomain {
  file {...}
  service {...}
}
```

node の中に書いた定義は、ホスト名が指定されたものと一致しているホストでのみ有効になります。

emacs はマシン全部に入れるけど、wicd はデスクトップには要らないや、みたいな使い分けが考えられます。

8.5.2 クラス

クラスは、ノード程具体的ではないのですが、「何となく関係しているリソースをまとめて名前つけとこ！」というイメージのものです。実際に使うときは、事前に定義した class をマニフェスト中で include する必要があります。

実際の定義はこんな感じになります。

```
class myconfig {
  file {....}
  package {.....}
}

include myconfig
```

8.5.3 デファイン

デファインは、クラスと似ているのですが、引数を取れるところと、デファインで定義したリソースは複数回設定できるところが違ってきます。

実際の定義はこんな感じになります。

```
define emacs::config ( $content, ){
  file { $name.el:
    content => $content,
  }
}

emacs::config { "howm":
  content => "(require 'howm)"
}
```

個人的には、クラスを補完する目的で使うことが多いでしょうか。クラスとの比較例でもよく使われますが、apache クラス (1 回だけ登場) の補助で、デファインしておいた apache::vhost (複数回登場) を使う、などです。

8.5.4 モジュール

モジュールは、前述したノードやクラスとはちょっと位置付けが異なっていて、マニフェストの書き方というよりは、ファイルの配置の仕方グルーピングをするものです。

基本的には、先に触れた puppetdir/modules の下に、ミニ puppet 設定ディレクトリを作ります。よく使うので一応 files と templates も書いていますが、モジュールとしては少なくとも init.pp があれば機能します。

```
modules/
モジュールの名前/
  manifests/
    init.pp
  files/
  templates/
```

一まとまりの大きな機能をモジュールとして構成しておいて、必要な場合だけ読み込む、といったことができます。

また、自分が書いているマニフェストの中でも、独立性を高くして切り離すことができるので、他の人とマニフェストを交換する場合に便利です。github などで、モジュール個別のリポジトリが公開している人などもあり、お手本としても役立ちます。

8.6 モジュールの切り出し

書いたマニフェストが充実してきたら、モジュールとして切り出すようにしたいのですが、今回は時間もないので、充実していないながら一度切り出しを試してみようと思います。

8.6.1 lv モジュール

では、一度 .puppet/modules に移動して、モジュールのためのディレクトリを用意して、モジュールを作ってみましょう。

```
mkdir -p ~/.puppet/modules/lv/manifests
emacs ~/.puppet/modules/lv/manifests/init.pp
```

```
class lv {
  ここに最初にした内容をコピー
}
```

site.pp に書いていた、同じ内容は消しておいてください。puppet では、同じ名前をもったリソース宣言は重複としてエラーになります。

8.6.2 モジュールの読み込み

さて、設定を切り出したのはいいですが、さっきと同じように puppet コマンドを実行しても、切り出した部分は動いてくれません。init.pp を直接引数にすれば動かなくもないですが、設定が複雑になったり、複数のモジュールを使いたくなったら破綻します。

puppet にモジュールを認識させるには、次のステップが必要です。

1. puppet の設定 (modulepath) でモジュールの配置場所を教える
2. site.pp からそのモジュールで定義したクラスを include する

これを実現するため、site.pp に、include lv という一文を追加して、こういうコマンドラインでもう一度実行してみましょう。

```
sudo puppet --modulepath ~/.puppet/modules ~/.puppet/manifests/site.pp
```

さて、無事に動いたでしょうか。

8.6.3 切り出しで迷ったら

愛情こめて育てた site.pp からモジュールを切り出そう、と思ったときに、どういう名前での範囲を切り取るか、迷うことがあるかも知れません。

そんな時は、使おうとしている機能を提供するパッケージとモジュールを対応づけるのがおすすめです。

すでに作った lv モジュールや、emacs モジュール、apache モジュール、xmonad モジュール、platex モジュール.....など考えるわけです。

切り分けの単位としてわかりやすいことと、puppet が OS のパッケージシステムとうまく連携してくれるために、後で複数のモジュールを組み合わせたくなった時にやりやすい、のがポイントです。

また、後で別の用途にそのモジュールを使い回すには.....と考えながら作ると、機能面でもいい具合に切り分けできるのではないかと思います。

8.6.4 puppet-module ツールの紹介

puppet を作っている puppetlabs という企業が、最近モジュール用のリポジトリとして alioth のようなサイトをオープンしました。

<http://forge.puppetlabs.com/>

ブラウザでジャンル毎に公開されているモジュールを探したりできるようになっています。今はまだ登録されているモジュールの数が少ないですが、そのうち充実してくれば、モジュールのカタログとして便利に使えるようになるのではと思います。

このサイトと連動して、検索や取得 (その気があれば公開も) をするためのコマンドラインツールもあります。それが puppet-module です。

今のところ github からの取得になりますが、そのうち ruby 関連のメンテナがパッケージにするでしょう。

実際に使うとこんな感じです。

```
puppet-module search [keyword]
```

で欲しいモジュールを探して、

```
puppet-module install xx-yyy
```

でダウンロードしてきます。指定している xx-yyy ですが、forge での名前空間は、xx (作者名)/yyy (モジュール

名) となっていて、それを指定します。自分の名前が頭につくので、モジュール名は他の人とかぶっても大丈夫、ということですね。

注意が必要なのは、install とはいいいながら、コマンドを実行したカレントディレクトリに展開するだけで、puppetdir/modules に自動で配置してくれるわけではないところです。ダウンロードした後で、自分で移動させてください。

puppet-module には generate という機能もあり、モジュールの雛形を作ってくれます。この場合も、forge の名前空間を指定する必要があるので、使用例は次のようになります。

```
$ bin/puppet-module generate lurdan-homedir
$ mv lurdan-homedir ~/.puppet/modules
$ cd lurdan-homedir
$ emacs manifests/init.pp
```

generate で作成されるディレクトリ構造はこんな感じになります。モジュールとしてフル機能が実装できるように、あれこれ気を利かせてくれている感じがしますね。

```
Modulefile
README
manifests/
files/
templates/
tests/
lib/
spec/
metadata.json
```

モジュールを作るのに慣れてきたら、これを使っていくのもいいと思います。逆に、慣れないうちは余分なものが多くて邪魔かも。

8.7 \$HOME の管理に使う

さて。長い長い前置きが終わり、ようやくここからが本題です。

puppet はその目的 (システムの構成管理) を考えても、root で実行するのがほぼ前提です。

でも、そんな大げさにシステムあっちゃこっちゃいじらないし、自分の home ディレクトリで設定ファイルを適当に管理してくれたいから、というのが今回のお題だったりするわけです。

package 操作などもあるので、やっぱり実行は root でやってもらう必要がどうしてもあります。でも、そのままファイルを作られたら、パーミッションが root:root になってしまいますし、そもそも puppet はマニフェストの中で実行ユーザのホームディレクトリを認識するようには作られていません。

8.7.1 環境変数の受け渡し

では、どうしましょう。半ば無理矢理ではありますが、\$HOME と \$USER を puppet に渡して、それをマニフェスト側で受けとればいいわけです。

渡す方は、「FACTER 環境変数」というものを使って実現します。FACTER 環境変数についての詳細は省きますが、ひとまず一種のおまじないとして、渡したい環境変数の頭に、「FACTER_」をつけるのだとおいてください。

具体的には、こういう感じで使います。

```
sudo FACTER_HOME=$HOME puppet /path/to/site.pp
```

こうして渡した環境変数は、マニフェストの中でこのように受けとることができます。

```
...
file { ["$home/.lv":
}
...
```

最初に作ってもらったマニフェストを見直してみてください。この仕掛けを使えば、ユーザ名を直接書かずに、使い回せる形で書き直すことができます。モジュールにすることを考えても、こちらの方がより融通がきくようになります。

ますね。ユーザ名埋め込みとか、ゾツとしませんよね。

当面必要になるのは、\$HOME と \$USER、後は group くらいなので、このあたりを埋めこんでシェルスクリプトにしておきます。

```
#!/bin/sh

FACTERENV="FACTER_HOME=$HOME FACTER_USER=$USER FACTER_GROUP='id -gn'"
sudo $FACTERENV puppet ~/.puppet/manifests/site.pp --modulepath ~/.puppet/modules --debug --verbose
```

これを適当に ~/.puppet/puppet.sh あたりに保存しておいて、実行する時は

```
$ ~/.puppet/puppet.sh
```

とすれば、マニフェストのどこでも、\$home と \$user、\$group にアクセスすることができます。実行ユーザの環境変数を拾っているのでも、シェルスクリプト自体には sudo は不要です。

これで下準備も整いましたので、どんどんマニフェストやモジュールを書いていってみてください。

8.8 もっと深みに

入門セッションということで、今日扱った内容からは、いろいろと重要なポイントが抜けています。一応キーワードだけ列挙しておきますので、後述の資料とあわせて、参考にしてみてください。

- `facter` : 実行環境のデータを集めて変数として提供。puppet の裏方
- ファイルサーバ : 事前に準備したファイルを各ノードに配布
- テンプレート : `eruby` で配布ファイルに変数埋め込み
- 仮想リソース : 定義の依存が被ってしまう場合に共用可能とする
- puppet 拡張 : `ruby` であれこれいじれるらしいです
- 環境 : 本番と検証と開発を分けたりできる

個人的に面白いなーと思っているトピックとして、

- `puppetdoc` : モジュールにマニュアルを埋め込む
- puppet のテスト

というのもあります。興味がある方はまた宴会でも。

8.8.1 資料は英語

puppet の公式な資料は、puppetlabs のサイトでメンテナンスされています。使うために必要な文書は比較的良好に整備されていると思いますが、基本的には英語です。

私は下記をよく参照しています。

- language tutorial : http://docs.reductivelabs.com/guides/language_tutorial.html
- type reference : <http://docs.puppetlabs.com/references/stable/type.html>
- meta parameter reference : <http://docs.puppetlabs.com/references/0.25.5/metaparameter.html>

8.8.2 日本語の資料

puppet は国内でも素晴らしい先達によってすでに語り尽くされていたりしますので、もうちょっと慣れるまで英語は.....という方は、下記の資料などにあたってみてください。

今回のセッションで扱った内容は、ほぼこれらの資料でカバーされてる範囲を出ていなかったりしますが、とっかかりにはなるのではと思います。

- オープンソースなシステム自動管理ツール Puppet : <http://gihyo.jp/admin/serial/01/puppet>

- Puppet によるインフラ管理入門 : <http://www.sssg.org/~naoya/puppet/project.html>
- puppet のススメ : <http://www.slideshare.net/mizzy/puppet-3258268>

9 ext4 ファイルシステムを Debian で活用してみる

上川 純一



ext4 ファイルシステムは Linux で広く使われている ext3 の後継ファイルシステムとして登場したファイルシステムです。特徴としては、ext3 と同じくジャーナリング機能を持ち、データ領域をブロック単位ではなくエクステント（一連の領域）で確保していること。32 ビットから 48 ビットになったので、ファイルシステムサイズが ext3 の制限を越えている。atime などが秒より細かい時間でわかるようになる、などです。^{*7,*8}

9.1 ext4 ファイルシステムを作ってみる

それでは、ext4 ファイルシステムを作ってみましょう。実は通常の ext3 ファイルシステムをそのまま ext4 としてマウントすることもできます。そうすると徐々にファイルが書き換えるたびに extent ベースになったりするようです。ただ、ファイルシステム全体のパラメータが ext4 用ではないので、ファイルシステムをいちから作成するのがよいでしょう。

アロケータのアルゴリズムも改善されているので小さなファイルのアロケーションも改善しているようですので、既存のファイルシステムの内容をバックアップしてリストアするのがよいのではないのでしょうか。

ext4 を使うには十分新しい e2fsprogs と十分あたらしい Linux Kernel があればよいです。Debian 5.0 (lenny) の時点で必要なパッケージはそろっているようです。

```
$ sudo lvcreate -L 1G -n lvext4 vghoge
Logical volume "lvext4" created
$ sudo mount /dev/vghoge/lvext4 /mnt/
$ mount -v
/dev/mapper/vghoge-lvext4 on /mnt type ext4 (rw)
$ df -h /mnt/
Filesystem      サイズ  使用  残り  使用% マウント位置
/dev/mapper/vghoge-lvext4
1008M    34M   924M    4% /mnt
$ df -T /mnt/
Filesystem      Type    1K-ブロック   使用   使用可  使用%   マウント位置
/dev/mapper/vghoge-lvext4
ext4           1032088     34052   945608    4% /mnt
$ df -i /mnt/
Filesystem      I ノード  I 使用   I 残り  I 使用%   マウント位置
/dev/mapper/vghoge-lvext4
65536          11   65525    1% /mnt
```

^{*7} A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. "The new ext4 filesystem: current status and future plans," Linux Symposium. 2007

^{*8} A. Kumar K. V., M. Cao, J. R. Santos, and A. Dilger. "Ext4 block and inode allocator improvements," Linux Symposium, Vol 1. 2008.

10 NILFS を Debian で活用してみる

山田 泰資



10.1 はじめに

NILFS ^{*9}は Linux の新型ファイルシステムの 1 つです。登場自体は v1 が 2003 年と結構前なのですが、機能を増強した v2 が 2007 年に登場し、その後 SSD 上での性能が著しく優れる [5] などのニュースで注目を集め、Linux 2.6.30 (2009/6 リリース) でメインラインに統合されるに至りました。

ここでは nilfs の特徴・使い方を紹介し、その上で他の有力(?)な選択肢としての LVM および btrfs との比較を行います。

10.2 使ってみよう - nilfs の導入と特徴

nilfs は通常のファイルシステムですので、とりあえず使い始めるだけなら簡単です。デモを兼ねてまずは使い始めましょう。

```
// 8GB の領域を使います
# parted -s -a none /dev/sda mlabel gpt
# parted -s -a none /dev/sda unit s mkpart primary ext2 2048 14682112

// 普通に mkfs して mount
# mkfs.nilfs2 /dev/sda1
mkfs.nilfs2 ver 2.0
Start writing file system initial data to the device
  Blocksize:4096 Device:/dev/sda1 Device Size:7516193280
File system initialization succeeded !!
# mount /dev/sda1 /mnt/p0
mount.nilfs2: WARNING! - The NILFS on-disk format may change at any time.
mount.nilfs2: WARNING! - Do not place critical data on a NILFS filesystem.
[2845849.612706] segtord starting. Construction interval = 5 seconds, CP frequency < 30 seconds

// さあ使ってみよう
# ls /mnt/p0
# <- 空の状態
# echo hello > /mnt/p0/file
# cat /mnt/p0/file
hello
```

何の変哲もないファイルシステムですね。しかし面白いのはここからです。

nilfs 最大の特徴は「連続スナップショット」機能です。

スナップショットは判るけど連続って？

と思われるかもしれませんが。答は nilfs の固有コマンド lscp を打つと判ります：

^{*9} 正式には NILFS2 ですが、すでに v2 が登場して久しいので本レポートではバージョン番号は略します

```
# lscp
CNO      DATE      TIME      MODE  FLG  NBLKINC  ICNT
 1 2010-11-12 07:03:53 cp    -      11      3
 2 2010-11-12 07:05:45 cp    -      14      4
```

上では2行表示されていますが、これはそれぞれの瞬間における nilfs の状態を保持していますよ、という意味になります。いわゆるスナップショットです。ただし、nilfs ではスナップショット (ss) とチェックポイント (cp) と概念が2つあります。いずれもある瞬間の状態を保持するという意味の「スナップショット」である点は同じですが、運用上の扱いがやや異なります。

そして、これらの過去の状態はファイルシステムとしてそれぞれ個別にマウントすることができます。試しに、

うっかりファイルを消してしまったけど、nilfs がチェックポイントに保存してくれていたのでもうそれをマウントすれば回復できて安心！

というデモをやってみましょう：

```
# ls /mnt/p0/
4 file0 4 file1 4 file2
# date
Fri Nov 12 07:19:14 JST 2010
# rm /mnt/p0/file2
# lscp
CNO      DATE      TIME      MODE  FLG  NBLKINC  ICNT
 1 2010-11-12 07:03:53 cp    -      11      3
 2 2010-11-12 07:05:45 cp    -      14      4
 3 2010-11-12 07:12:55 cp    -      12      5
 4 2010-11-12 07:13:21 cp    -      15      6
 5 2010-11-12 07:13:32 cp    -      27      6
 6 2010-11-12 07:16:45 cp    -      10      6
 7 2010-11-12 07:16:54 cp    -      13      6
 8 2010-11-12 07:16:56 cp    -      11      6
 9 2010-11-12 07:18:33 cp    -      13      5
10 2010-11-12 07:19:11 cp    -      14      6 <- 消す直前の段階は CNO==10
11 2010-11-12 07:19:18 cp    -      13      5
# chcp ss 10
# mount -t nilfs2 -o ro,cp=10 /dev/sda1 /mnt/p0.10
# ls /mnt/p0.10/
4 file0 4 file1 4 file2
# cp /mnt/p0.10/file2 /mnt/p0/
```

<- うっかり消してしまった！どうしよう！(棒)

<- スナップショットに変換する

<- 消す前の状態をマウント

<- 消す前の状態を確認

<- 回復！

見ての通り、無事回復できました。nilfs は書き込みをフラッシュする度に自動的にチェックポイントを作るので、編集 → 保存 → 誤削除などのオペミスをして、直後ならばまず確実に復活することができます。これは通常のスナップショットやバックアップツールにない強力な特徴です。

nilfs の管理コマンドは、このスナップショット機能を中心に整備されています。現在あるコマンドは以下の通りです 2：

コマンド名	機能
mkfs.nilfs2	nilfs(v2) ファイルシステムを作成する
mount.nilfs2	マウントする
umount.nilfs2	アンマウントする
lscp	チェックポイント・スナップショットの一覧を表示する
mkcp	チェックポイント・スナップショットを作成する
chcp	チェックポイント・スナップショットを相互変換する
rmcp	チェックポイントを削除する
lssu	ディスク上の nilfs 内セグメントの使用状態を表示する
nilfs_cleanerd	ガーベージコレクション処理を行う (詳細後述)
dumpseg	デバッグ用。nilfs 内セグメントの情報をダンプする

表 2 nilfs の管理コマンド一覧

10.3 チェックポイント (cp) とスナップショット (ss)

さて、チェックポイントとスナップショットの違いですが、以下の通りとなります 3 :

	チェックポイント	スナップショット
自動的に作られるか?	YES	NO
手動でも作れるか?	YES	YES
自動で開放されるか?	YES	NO
マウントできるか?	NO	YES

表 3 チェックポイントとスナップショットの違い

上の実行例にて `chcp` コマンドで変換していましたが、チェックポイントとスナップショットはその名称が異なるだけで、内部的には同じものです。ただ、その名称によって自動生成・自動削除の扱いが変わり、またそれに連動してマウントの可否も異なる、という形になります。

`nilfs` ではチェックポイントが自動的に存在する状態が基本線で、管理者としては永続的に残したい状態をスナップショットとして作り、必要に応じてマウントし内容を参照する、というのが通常の使い方となります。

10.4 `nilfs` の構造 - ログ構造化ファイルシステム

`nilfs` は種類としてはログ構造化ファイルシステム (LFS^{*10}) の実装です。これは `ext3` や `btrfs` などのジャーナリングファイルシステムとは異なる設計で、以下のような違いがあります^{*11} :

ジャーナリングファイルシステムの特徴

1. 書き込みはジャーナルログ領域にまず記録され、追って実際のデータ領域に反映させる
2. クラッシュ時は、ジャーナルログを走査し、未反映分をデータ領域に反映させる

ログ構造化ファイルシステムの特徴

1. 書き込みでは元データ領域は更新せず、新たに空き部分を確保して追記する
2. クラッシュ時は、追記内容を過去に遡り、正常終了していた部分まで巻き戻す

ジャーナリングファイルシステムではジャーナルログかデータ領域のいずれかに必ず正当なデータが存在することになり、クラッシュ耐性の高いファイルシステムが実現されます。デメリットは、ジャーナル領域とデータ領域に二度同じ内容を書くため、そこがオーバーヘッドとなります^{*12}。

そして、一方のログ構造化ファイルシステムでは、元データ領域か追記されたデータ領域のいずれかに必ず正当なデータが存在することになります。これをジャーナリングと比較すると、

1. 書き込みは 1 回するだけなので効率がよい
2. リカバリは書き込みの末尾から最後の有効な追記部分を遡って探すだけで高速

というメリットがあります。しかし、追記のみでは削除・書換されたファイルが開放されず容量を消費するため、これを開放するためのガーベジコレクション処理 (GC 処理) をいずれ実行する必要がある、これがオーバーヘッドとなります。

さて、`nilfs` は昨年頃より SSD 上でのベンチマークで高スコアを出すと報告され注目を集めました、これは

^{*10} LFS という名前の実装もあって、ややこしい…(JFS も)

^{*11} と言いつつ、`btrfs` ではジャーナルもデータ領域も `extents` 上の同一構造で区別がなく、書き込み処理などでも LFS 的な要素を取り込んでいます [13]

^{*12} このためメタデータのみジャーナリングするなどの対応を取る

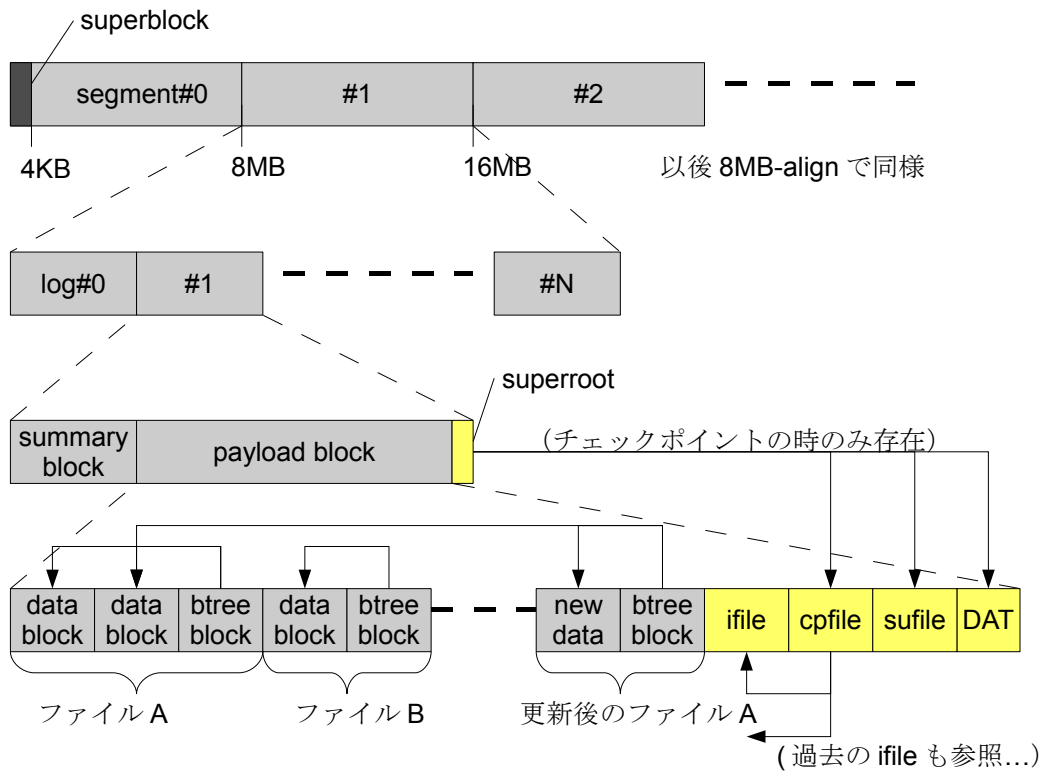


図1 nilfs のオンディスクフォーマットの概要

- SSD はブロック書換えのオーバーヘッドが大きく、追記処理で最高性能を出す
- nilfs は空きがある限りシーケンシャル書き込みでの追記になる（特に、GC が走っていない場合）

という2つの要因が合わさった事によるものです。しかし、その後

- SSD は追記以外の書き込み処理の効率化を進めた
- ジャーナリング系 FS もデータブロックの確保・書込方法を改良して行った

という事があり、nilfs が LFS であることをもって書き込み性能が特に優れたり、SSD での利用に向くという訳ではなくなってきています。特に、GC 中の性能やフラッシュメモリ上での利用において注意が必要です [7, 6]（詳細後述）。

さて、ここら少しだけ nilfs の構造をしてみることにしましょう 1。図の通りデータは前方から並び、末尾にデータを参照するメタデータブロックが付くだけのシンプルな構造であることが判ります。

基本的に 8MB のセグメントに分割され、先頭セグメント (#0) のみ冒頭 4KB の中にスーパーブロックが含まれます（つまり segment #0 だけ 4KB 小さい）。そしてセグメントの中には書き込み単位であるログが1つ以上含まれます。ログの中にはファイル自体のデータ（data block）および各ファイルを構成するデータブロックへの参照（btree block）、そして各ログの構成（summary block）や各ブロックを管理する inode のインデックス（ifile）、そしてチェックポイント情報（superroot + DAT + sufile + cpfile）を管理する nilfs としてのメタデータが含まれています [4, 1]。

チェックポイント処理では、前回からの差分（図中の「更新後のファイル A」）が書き込まれ、書き換わらなかった部分は新しい btree block 中から既存の data block を参照する形で新旧の状態を並存させます。また、複数のログをまたぐチェックポイントの場合、末尾のログのみ superroot(SR) レコードが書き込まれます。このレコードはチェックポイントの構造を管理する DAT/sufile/cpfile の3レコードへのポインタを管理し、この SR の書き込みまで完了していれば、それがリカバリ時に有効なチェックポイントとなります。

10.5 nilfs の落とし穴 - ガーベージコレクション

これは設計上どうしても発生する問題ですが、上の構造で追記を続けると、いつかはディスク末尾に書き込み位置が達して追記不可能になります。

初めて nilfs を使ってみた方は、ファイルシステムのサイズより十分小さいファイルしか書いていないにも関わらず、なぜか 100% full になって書き込めなくなったことがあるのではないのでしょうか？これは、ユーザーから見た「現在の」ディスク消費量は十分少なくとも、過去の状態を保存しているログ領域まで含めた総量では多量となり、追記不可能となったことによるものです。

```
// ほぼ 7GB の領域が空いていることを確認
# df .
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sda1       7331836       16380  6946816   1% /mnt/p0
// 1GB のデータを同じファイルに繰り返して書く
# dd if=/dev/zero of=big.bin bs=8192 count=$((8192 * 16))
...
# dd if=/dev/zero of=big.bin bs=8192 count=$((8192 * 16))
// ユーザーから見ると 1GB のファイルがあるだけが ...
# ls -l
total 1052716
1052716 -rw-r--r-- 1 root root 1073741824 Nov 15 15:31 big.bin
// ファイルシステムとしては 90% full になっている
# df .
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sda1       7331836       6201340   761856   90% /mnt/p0
// チェックポイントが 100 個ほど作られている。
// この過去の状態を指しているチェックポイントがディスク消費の原因
# lscp | wc -l
106
```

しかし、過去の内容を含むチェックポイントを開放しさえすれば、空き容量は回復できるはずです。では、最新 1 件を除いて開放してみましょう：

```
// チェックポイント番号#1 以降を全部削除する (スナップショット化されたものは削除されない)
# rmcp 1..
# lscp
      CNO      DATE      TIME  MODE  FLG  NBLKINC      ICNT
17407 2010-11-15 15:31:25 cp    i      29      4
# df .
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/sda1       7331836       5439484  1523712   79% /mnt/p0
```

たしかに若干空き容量が増えましたが、それでも 5GB 超を消費しており、実際に書かれている 1GB のファイルサイズとは乖離がかなりあります*13。

これは、nilfs ではチェックポイントの開放と開放されたチェックポイントが参照していたデータブロックの回収処理が分離していることが原因で、実際の空き容量の回復は GC (nilfs_cleanerd プロセス) による回収を待たなくてはなりません。rmcp によるチェックポイントの開放はあくまで印を付けているだけの処理になります。

10.6 nilfs_cleanerd - 標準ガーベージコレクタ

nilfs を使う上では、チェックポイントの開放・ブロック回収を行う GC のパラメータ調整が重要です。

1. 平均的な書き込みペースと開放・回収ペースが均衡する必要がある
2. 短期的な突発書き込みに対しても十分な空き容量を作れなくてはならない
3. GC は多量の IO を発生させるため、あまり実行したくない(ことがある)
 - 性能面の問題と、USB メモリや SSD などのフラッシュメモリの寿命上の問題があります
 - iotop -Pao などで確認すると、GC では書込量以上の R/W IO が発生しているのが判ります
4. スナップショットは自動開放されないが、容量が逼迫した際に 100% full とするか開放すべきか

こういったポイントを考慮しつつ、個々の用法にあわせて調整・運用ポリシーを決定します。

*13 これは GC が 30[s] 間隔で動いていた時の結果で、デフォルト設定なら早々に 100% full になっていたでしょう

この GC は最近改良が進んだ部分で、Debian に現在入っているバージョンでは比較的問題を起こしにくい設定ができるようになっています。どのようなパラメータがあるか、表にまとめてみました：

パラメータ	デフォルト	概要
protection_period	3600	生成されたチェックポイントを GC から保護する最低期間 (秒) です。この時間の間は回収されないため、逆に言えば <ul style="list-style-type: none"> 誤削除をしても、上記保護期間の間なら直前のチェックポイントから回復できる この時間内に残り空き容量を超える書込 I/O を行うと、100% full になる ということになります。
min_clean_segments	10%	空き容量 (セグメントベース) がここで指定した量を切るまでは、GC を行いません。これは過度の GC を抑止するための設定です。GC は開放しないセグメントを前方に詰め直す再配置処理もするため、この IO 負荷が性能劣化を起こしたり、フラッシュメモリへの書込負担を増大させます。このため容量が逼迫するまで GC を抑止するためにこの設定を調整します。 なお、設定は 10% のように割合で指定することも、10G のように容量で行うこともできます。
max_clean_segments	20%	空き容量 (セグメントベース) がここで指定した量を超えている間は、GC を行いません。これは上の min_clean_segments 同様、過度の GC 走行を抑止するための設定です。容量の指定方法も同様です。
clean_check_interval	10	空き容量の確認を行う間隔 (秒) です。
selection_policy	timestamp	回収ポリシーを指定します。これは現在はチェックポイントの生成時間を使う timestamp ポリシのみです。
nsegments_per_clean	2	1 回の GC で何個のセグメントを回収するかの設定です。書き換え・削除のペースに比べて回収量が小さすぎると中々処理が進まず、空き容量が生まれません。一方、過度に大きくすると GC 対象になるとすぐ回収されてしまうため、過去のチェックポイントがほとんど残されないということになります。
mc_nsegments_per_clean	4	空き容量が min_clean_segments を下回っていた場合 (= より容量が逼迫している場合)、1 回の GC で何個のセグメントを回収するかの設定です。
cleaning_interval	5	GC 開始のトリガが引かれた後、何秒間隔で GC 処理を行うかの設定です。つまり、先の nsegments_per_clean と合わせて $1 \text{ 時間での最大回収量} = 8[\text{MB}] * nsegments_per_clean * 3600[\text{s}] / cleaning_interval[\text{s}]$ となり、これと予想される書込・削除ペースを比較して調整します (8[MB] のセグメントサイズは mkfs 時に変更可能です)。

パラメータ	デフォルト	概要
mc_cleaning_interval	1	空き容量が min_clean_segments を下回っていた場合 (= より容量が逼迫している場合)、GC 開始のトリガが引かれた後、何秒間隔で GC 処理を反復するかの設定です。 これも、先の mc_nsegments_per_clean と合わせて $1 \text{ 時間での最大回収量} = 8[\text{MB}] * \text{mc_nsegments_per_clean} * 3600[\text{s}] / \text{cleaning_interval}[\text{s}]$ となり、これと予想される書込・削除ペースを比較して調整します。
retry_interval	60	空き容量がなくなっている状態で、回収可能なセグメントが見つからなかった場合の GC のリトライ待機時間です。
use_mmap	1	GC でのセグメントの読み出しに mmap(2) を使うかどうかの設定です。ただし、現在は mmap(2) が使える場合、設定に関わらず使用します。
log_priority	info	ログメッセージ出力時に使う syslog レベルです。

以上が設定項目ですが、実際に設定する内容はストレージの用途・種類によって大きく異なります。例えば、80% 程度が常時埋まり、できる限り多量のチェックポイントが残されている状態を目指すとして、その場合は

残り 20% を GC ペースを上回って埋め尽くすような突発的な IO が発生しないか？

という検討をしなくてはなりません。一方で余裕を見すぎると

空き容量は常時すぐに確保され安心だが、チェックポイントがあまり残らないし、GC が走る頻度が高すぎて IO 性能が劣化した状態が多い

ということになります。min_clean_segments/max_clean_segments が導入されて挙動 (と調整しやすさ) は改善された [9] ののですが、利用を検討される場合は、まずは取っ掛かりとして書換・削除が少ないアーカイブ的なストレージやログなどの、IO 傾向が読みやすい用途で使い始めてみることをお勧めします。

10.7 libnilfs - 手作りガーベージコレクタへ

さて、nilfs_cleanerd はユーザーランドで動く GC なので、カーネルに手を入れることなく独自の GC ポリシを持つ別の GC を自作することも可能です。このため (?) に nilfs-tools パッケージには nilfs API (libnilfs) のヘッダファイルも含まれています。

一番下のレベルでは ioctl で nilfs にリクエストを発行するのですが、このレベルで制御するのは非常に煩雑なため、nilfs_* API が libnilfs で提供されています。

まだ私も自作したことはないので予備調査の段階なのですが、標準の nilfs_cleanerd では以下の流れで GC 処理を行っています：

```
// デバイスオープンして GC 起動
main():
cleanerd->c_nilfs = nilfs_open(dev, dir, NILFS_OPEN_RAW | NILFS_OPEN_RDWR);
nilfs_cleanerd_clean_loop(cleanerd);

// 後は待機 -> 状況確認 -> 開放 -> 待機 -> ... の無限ループへ
nilfs_cleanerd_clean_loop(cleanerd):
r_segments = nilfs_get_reserved_segments(cleanerd->c_nilfs);
nilfs_cleanerd_clean_check_pause(cleanerd, &timeout);
loop:
nilfs_get_sustat(cleanerd->c_nilfs, &sustat);
nilfs_cleanerd_handle_clean_check(cleanerd, &sustat, r_segments, &timeout);
nilfs_cleanerd_select_segments(cleanerd, &sustat, segnums, &proptime, &oldest);
nilfs_cleanerd_clean_segments(cleanerd, &sustat, segnums, ns, proptime);
nilfs_cleanerd_recalc_interval(cleanerd, ns, proptime, oldest, &timeout);
nilfs_cleanerd_sleep(cleanerd, &timeout);
```

上の nilfs_cleanerd_* は libnilfs ではなく nilfs_cleanerd 固有の内部関数なので、主要部分である

```
nilfs_cleanerd_select_segments(cleanerd, &sustat, segnums, &protime, &oldest);
nilfs_cleanerd_clean_segments(cleanerd, &sustat, segnums, ns, protime);
```

が libnilfs API でどのように実現されているか、更に分け入ってみましょう。

```
// 回収できるセグメントを選び出す
nilfs_cleanerd_select_segments(..., IN nilfs_sustat *stat, OUT u64 *selected, ...):
    smv = nilfs_vector_create(sizeof(struct nilfs_segimp));
    foreach(segnum in stat->ss_nsegs):
        // 各セグメントのステータスを確認して ...
        n = nilfs_get_suinfo(nilfs, segnum, info, count);

        // 回収しても問題ないログを見つけ出す
        for (0..n):
            if (nilfs_suinfo_dirty(&info[i]) && ...):
                sm = nilfs_vector_get_new_element(smv);
                sm->si_segnums = i;

        // 回収優先度の順にソート
        nilfs_vector_sort(smv, nilfs_comp_segimp);

        // 今回回収したい数だけ頭から拾い出す
        foreach(smv):
            sm = nilfs_vector_get_element(smv, i);
            selected[i] = sm->si_segnum;
        nilfs_vector_destroy(smv);
```

これで GC 対象セグメントを抽出し、以下の開放処理に渡します：

```
// 回収対象セグメントを開放する
nilfs_cleanerd_clean_segments(cleanerd, IN nilfs_sustat *stat, IN u64 *selected, ...):
    // セグメント番号をディスク上の論理・物理ブロックアドレスに変換などする
    n = nilfs_cleanerd_acc_blocks(cleanerd,
                                sustat, segnums, nsegs, vdescv, bdescv);

    // ロックして GC 実行
    ret = nilfs_lock_write(cleanerd->c_nilfs);
    ret = nilfs_clean_segments(cleanerd->c_nilfs,
                              nilfs_vector_get_data(vdescv),
                              nilfs_vector_get_size(vdescv),
                              nilfs_vector_get_data(periodv),
                              nilfs_vector_get_size(periodv),
                              nilfs_vector_get_data(vblocknr),
                              nilfs_vector_get_size(vblocknr),
                              nilfs_vector_get_data(bdescv),
                              nilfs_vector_get_size(bdescv),
                              selected, n);
    nilfs_unlock_write(cleanerd->c_nilfs)
```

上の `nilfs_cleanerd_acc_blocks` のまま残されている部分の前後ではセグメント番号をディスクのブロックアドレス変換するなど多数の細かい処理があるのですが、全体としての流れはかなり簡潔であることがわかります。

本来なら自作 GC の作成まで完全に紹介したかったのですが、今回はここまでです。もし `nilfs_cleanerd` の挙動では対応できないケースは、自作・改造という手段もありえなくはないということで取っ掛かりとして紹介してみました。

10.8 他の選択肢との比較 (1) - btrfs

Linux の次世代ファイルシステムとしてよく取り上げられるものに `btrfs` があります。 `btrfs` は `nilfs` のような「連続スナップショット」機能は持たないものの、従来に比べて強力なスナップショット機構を備えています。ここでは `nilfs` との比較を交えて各機能を軽く紹介することにします。

`nilfs` の紹介の時と同様、まずは使ってみましょう。

```
// 他のテストの関係で sda1[456] を使いますが、まずは sda16 単体で試す
# mkfs.btrfs /dev/sda16
# mount -t btrfs /dev/sda16 /mnt/p0
# ls /mnt/p0
#
# echo hello > /mnt/p0/file
# cat /mnt/p0/file
hello
```

しかし、上のような使い方では `btrfs` の特徴がまったく出ていません。 `btrfs` の主要な特徴としては

1. 複数の物理デバイスを 1 つのストレージプールとして扱う LVM+MD 的な機能
 - ボリューム管理はデバイスの追加削除・リサイズができるようになってきた

- RAID は RAID(0/1/10) のみ&やや機能不足ですが、RAID[56] なども計画中
2. 空き領域を共有しつつ複数の独立 FS 領域を切り出せるサブボリューム機能
 - 一見サブフォルダですが、スナップショット・一括消去・ボリューム切替など多様な使い方の基盤になります
 3. 何個でもネストでき、更に書込可能な軽量・高機能なスナップショット
 - ここが nilfs との比較で直接競合する部分です。
 4. POSIX ACL などの拡張機能の完備、チェックサム機能、圧縮機能、等

と、nilfs よりもかなり広い範囲の機能をカバーしています。このためまだ開発途上の部分があるのですが、読者の方が関心を持ちそうな部分を中心に見てゆきましょう。

10.8.1 btrfs の使い方

まず、ディスクの追加・削除、ボリューム/スナップショットの作成・削除など、btrfs の固有機能は `btrfs(8)` コマンドから操作します。これは以前は `btrfsctl(8)` というコマンドだったのですが、途中で変更になりました。ちょっと前の紹介記事などでは使われていたりするので、その場合は対応するコマンドを探してください。

他にも補助的なコマンドがいくつかありますが、主なものは以下の通りです 4 :

コマンド名	機能
<code>btrfs</code>	ディスクの追加・削除、ボリューム管理、スナップショット管理など、各種のサブコマンドを駆動する btrfs の基本管理ツールです。
<code>mkfs.btrfs</code>	指定のブロックデバイスの上に btrfs を構築します。メタデータ・データそれぞれの RAID レベルを <code>-m/-d</code> オプションで <code>single</code> , <code>raid0</code> , <code>raid1</code> , <code>raid10</code> から選べます。引数のブロックデバイスの数でデフォルトのポリシーが変わるため、明示的に指定するとよいでしょう。
<code>btrfsck</code>	破損した FS を修復します。マウント中でも問答無用で実行されますが、たぶん危ないのでやめましょう。
<code>btrfs-image</code>	いわゆる <code>dump/restore</code> 。FS イメージをファイルに落としたり、そこから戻します。
<code>btrfs-convert</code>	現在 <code>ext[234]</code> なファイルシステムからの移行ツール。空き領域に btrfs を構築し、元の FS のデータブロックからスナップショット分岐する形では btrfs を試すことができます。btrfs 側から書き込んだ内容は CoW で別領域に書かれ、 <code>ext[234]</code> からは認識されないため、元の状態に戻すことができます。そのまま移行する場合はスナップショット元を削除するのみで完了します。
<code>btrfs-debug-tree</code>	デバッグ用。btrfs の内部構造をダンプします。

表 4 btrfs の管理コマンド一覧

さて、それでは

`/dev/sda1[456]` の 3 つのブロックを RAID1 相当にし、
その上でボリュームを切ったりスナップショットを取って挙動を見る

というデモをしてみましょう。使うコマンドは `btrfs(8)` と

- `btrfs filesystem` - FS のリサイズ・デフラグ・構成チャンクの再配置などをする
- `btrfs subvolume` - サブボリュームの作成やスナップショットなどをする

- btrfs device - ディスクデバイスの追加などをする

の3種類のサブコマンドです (使用時は fi → filesystem など略記可能)。

```
// まずファイルシステム初期化
# mkfs.btrfs -m raid1 -d raid1 /dev/sda1[456]
Scanning for Btrfs filesystems
[2744375.419909] device fsid eb42923602baa275-a7c6b398c8770a98 devid 3 transid 7 /dev/sda16
[2744375.439127] device fsid eb42923602baa275-a7c6b398c8770a98 devid 2 transid 7 /dev/sda15
[2744375.455875] device fsid eb42923602baa275-a7c6b398c8770a98 devid 1 transid 7 /dev/sda14
# mount -t btrfs /dev/sda14 /mnt/p0
```

これで sda1[456] から構成された btrfs をマウントできます。この3つのデバイスがセットだというのは fsid で自動認識されるので、指定するブロックデバイスはどれでも構いません。

ブロックデバイスの追加は mkfs の後からも行うことができます：

```
# mkfs.btrfs /dev/sda14
fs created label (null) on /dev/sda14
  nodesize 4096 leafsize 4096 sectorsize 4096 size 15.00GB
# mount -t btrfs /dev/sda14 /mnt/p0
[2744633.452049] device fsid 9940cc117db676de-c22b0b959bec58a3 devid 1 transid 7 /dev/sda14
# btrfs device add /dev/sda15 /dev/sda16 /mnt/p0
```

ただし、現在は RAID レベルの指定は mkfs でしか行えません。上の例では mkfs には1つしかデバイスを渡さず無指定なので、デフォルトの

- メタデータは2つコピーを作る (-m dup に相当 - ただし dup 指定はできない)
- ファイルデータはコピーを作らない (-d single に相当)

という状態のまま、使えるデバイスが増えた状態になります。dup というのは「2箇所に書くけど、それぞれを別のデバイスに置くなどの配慮はしない」という動作です。また、mkfs 時に複数デバイスを渡すと -m raid1 -d single 相当になります [14]。

それでは (サブ) ボリュームを切ってみましょう：

```
# mount -t btrfs /dev/sda14 /mnt/p0
# btrfs sub create /mnt/p0/v0
Create subvolume '/mnt/p0/v0'
# btrfs sub create /mnt/p0/v1
Create subvolume '/mnt/p0/v1'
# ls -lR /mnt/p0
/mnt/p0:
total 0
0 drwx----- 1 root root 0 Nov 11 03:09 v0/
0 drwx----- 1 root root 0 Nov 11 03:09 v1/
/mnt/p0/v0:
total 0
/mnt/p0/v1:
total 0
#
```

.....

え？これ (サブ) フォルダと何が違うの？

と思った方はいないでしょうか？ 実際、「容量を全体で共有しながら名前空間的に切り分ける」というのはサブフォルダでも同じことですよね。

以下がサブボリュームのサブフォルダとの機能上・使用上の違いです：

1. 独立した FS としてマウント、スナップショットなどの単位として機能する
2. 中に「普通の」ファイル・フォルダがあっても即時に領域開放できる
3. サブボリュームは btrfs がフォルダ風に見せているだけで rmdir は使えない

サブボリュームの「サブ」はパス名上の位置的な話で、これによって確保されたボリューム領域は「親」ボリュームに依存しない、完全に対等の領域として確保されています。スナップショットのデモを兼ねてこの意味合いを確認してみましょう：

```

// サブサブボリュームまで作ってファイルを置く
# btrfs sub create v00
Create subvolume './v00'
# btrfs sub create v00/v01
Create subvolume 'v00/v01'
# echo hoge hoge > ./v00/v01/hogehoge
# find
.
./v00
./v00/v01
./v00/v01/hogehoge

// このスナップショットをサブボリュームとして作り、
// その上で元の v00, v00/v01 ボリュームを開放してみる。
# btrfs sub snap v00/v01 s00
Create a snapshot of 'v00/v01' in './s00'
# btrfs sub del v00/v01
Delete subvolume '/mnt/p0/v00/v01'
# btrfs sub del v00
Delete subvolume '/mnt/p0/v00'

// これは「サブ」というのがパス名だけの話で、領域自体は
// どのボリュームも親子関係なく存在しており、名前の付け方で
// 同じボリューム領域を（スナップショットという名のボリュームを
// 介して）パス上のどこにでも配置できるのを見せるデモになる
# find
.
./s00
./s00/hogehoge
# cat s00/hogehoge
hogehoge

```

btrfs の内部構造上はすべてのサブボリュームとスナップショットは対等です。また、mkfs+mount 直後に見えているトップレベルのボリュームも、単にデフォルトのマウント対象^{*14}というだけで、やはりサブボリュームです。つまり「サブ」というものは内部構造的には存在せず、

ボリュームにどのような（パス）名や初期状態をセットするか

というのがユーザーから見た btrfs の使い方の基礎になっています。サブボリュームに限らず、スナップショットというも既存のボリュームを領域を CoW で共有するように初期設定されているだけで、中身は単なるボリュームになっています。

この単純化の結果、ボリューム・サブボリューム・スナップショットは同じコマンドで統一的に扱うことができます。スナップショットのスナップショット、のように無制限に分岐しつつ書き込んでゆけるといった特徴なども、すべてこのボリューム構造が基盤になっています。

10.8.2 btrfs のスナップショットはどこまで軽量か？

さて、次は btrfs のスナップショット機能がどこまで nilfs 的に使えるか（=軽量か）を確認してみましょう。連続スナップショット機能はないのでこれは諦めるとして、

- スナップショットあたりのオーバーヘッド
- スナップショットへの書き込み後のディスク使用量や IO 性能は

を確認します。

ある程度のサイズとファイル数がある場合で見えるために、Linux のカーネルソースをテストでは使いました：

^{*14} これは btrfs sub set-default で切替できる（はずですが現在動かず）

```
// 初期化と作業用サブボリュームの作成
# mkfs.btrfs -m raid1 -d raid1 /dev/sda1[456]
# mount -t btrfs /dev/sda14 /mnt/p0
# cd /mnt/p0
# btrfs sub create v00
Create subvolume './v00'

// ファイルの展開と IO 性能の確認
# tar xf /var/tmp/linux-source-2.6.32.tar.bz2 -C v00
# dd if=/dev/zero of=v00/1GB.bin bs=8k count=128k
131072+0 records in
131072+0 records out
1073741824 bytes (1.1 GB) copied, 53.1512 s, 20.2 MB/s

// ディスク消費量を確認
# find | wc -l
32576
# du -s .
1426335 .
# btrfs fi show
Label: none  uuid: 012cb75d-8aa3-4247-904a-2e7e6a9602e8
      Total devices 3 FS bytes used 1.38GB
      devid    1 size 7.00GB used 3.02GB path /dev/sda14
      devid    2 size 96.00GB used 2.01GB path /dev/sda15
      devid    3 size 96.00GB used 1.01GB path /dev/sda16

Btrfs Btrfs v0.19

// どうも 2.6.37 以降でないとい df コマンドは使えないらしい…
# btrfs fi df /mnt/p0
#
```

使用量の表示について補足すると、btrfs fi show が報告している 1.38GB というサイズはデータ・メタデータとも RAID レベルを考慮しない 1 つ分だけの数字になります。つまり、今回は raid1 を使っているためディスク上はこの倍を消費しています。この表示は 2.6.34 以降では RAID レベルを考慮した実消費量を出すように修正されました [15]。

さて、それではスナップショットを切って、前後の IO 性能と容量消費の変化を確認します。1 回では判りにくいので、1000 回切ります。

```
# for i in `seq 1000 1 1999`; do btrfs sub snap v00 s$i; done
Create a snapshot of 'v00' in './s1000'
...
Create a snapshot of 'v00' in './s1999'

// ユーザレベルで見た消費量の変化を見るが…
# du -s .
^C <- 時間がかかりすぎるので諦めた (x1000 のサイズになる)

// btrfs レベルで見た消費量の変化を見ると、0.2GB 程増えた
# btrfs fi show
Label: none  uuid: 012cb75d-8aa3-4247-904a-2e7e6a9602e8
      Total devices 3 FS bytes used 1.40GB
      devid    1 size 7.00GB used 3.02GB path /dev/sda14
      devid    2 size 96.00GB used 2.01GB path /dev/sda15
      devid    3 size 96.00GB used 1.01GB path /dev/sda16

Btrfs Btrfs v0.19

// スナップショット先のファイルに上書きして IO 性能を見る -> 変わらず
# dd if=/dev/zero of=s1500/1GB.bin bs=8192 count=8k count=128k conv=notrunc
131072+0 records in
131072+0 records out
1073741824 bytes (1.1 GB) copied, 52.3484 s, 20.5 MB/s

// 上の上書きで CoW が走ったはずなので、再度消費量を見る -> ほぼ CoW 分増えた
# btrfs fi show
Label: none  uuid: 012cb75d-8aa3-4247-904a-2e7e6a9602e8
      Total devices 3 FS bytes used 2.38GB
      devid    1 size 7.00GB used 4.02GB path /dev/sda14
      devid    2 size 96.00GB used 3.01GB path /dev/sda15
      devid    3 size 96.00GB used 1.01GB path /dev/sda16

Btrfs Btrfs v0.19
```

ここまでの書き込み量は linux-kernel(360MB) + 1GB.bin(1GB) を元にスナップショットの 1 つだけ 1GB.bin を書き換えて CoW を発生させたので計 2360MB 程度で、これにスナップショット 1000 回分を含むメタデータとあわせて 2.38GB というわけで、妥当といえる消費量になっています。また、多数回のスナップショットを行っても容量・性能の両面で劣化は抑えられており、優れたファイルシステムであるといえるでしょう。

10.9 他の選択肢との比較 (2) - LVM

実は `nilfs` や `btrfs` が実現しているようなレベルのスナップショット機能と比べると、LVM は設計方針自体が根本的に違うため、「過去 30 日の任意の時点に戻るタイムマシン」のようなものは LVM では

はっきり言って、無理

です。正確には、性能劣化や容量効率が悪すぎるため、使い物になりません。

これはスナップショット後の書き込みで発生する CoW 挙動に起因する問題で、LVM では CoW を行った時点で、生成されているすべてのスナップショットの個数分の書き込み + コピーが行われます。つまり、30 個スナップショットがあれば、CoW のタイミングで書き込み量は $\times 30$ に増幅し、性能は $1/30$ (実際はもっと劣化します) になります。CoW 後は性能は復帰しますが、今度は $\times 30$ のディスク消費を抱えることになります。

挙動をスナップショット 2 つで確認してみましょう：

```
// まずマスタとなるボリュームを確保して ...
# pvcreate /dev/sda16
Physical volume "/dev/sda16" successfully created
# vgcreate vg0 /dev/sda16
Volume group "vg0" successfully created
# lvcreate -n p0 -L 10G vg0
Logical volume "p0" created

// そこで 1GB のファイルを作る
# mkfs.xfs /dev/vg0/p0 <- XFS なのは昔のテストの時のメモのコピベだからです
# mount /dev/vg0/p0 /mnt/p0
# dd if=/dev/zero of=/mnt/p0/1GB.bin bs=8k count=128k
131072+0 records in
131072+0 records out
1073741824 bytes (1.1 GB) copied, 24.3558 s, 44.1 MB/s

// そしてスナップショットを作り、容量の使用状態を確認
# lvcreate -n p1 -L 2G -s /dev/vg0/p0
Logical volume "p1" created
# lvcreate -n p2 -L 2G -s /dev/vg0/p0
Logical volume "p2" created
# lvdisplay
--- Logical volume ---
LV Name                /dev/vg0/p0
...
LV Size                 10.00 GiB
...
--- Logical volume ---
LV Name                /dev/vg0/p1
...
CoW-table size         2.00 GiB
CoW-table LE           512
Allocated to snapshot  0.00%          <- まだ 2GB まるまる空いている
...
--- Logical volume ---
LV Name                /dev/vg0/p2
...
CoW-table size         2.00 GiB
CoW-table LE           512
Allocated to snapshot  0.00%          <- まだ 2GB まるまる空いている
...
# mount -o ro,norecovery,nouuid /dev/vg0/p1 /mnt/p1
# mount -o ro,norecovery,nouuid /dev/vg0/p2 /mnt/p2
# cd /mnt; ls -l p0 p1 p2
p0:
total 1048576
1048576 -rw-r--r-- 1 root root 1073741824 Oct 29 01:54 1GB.bin
p1:
total 1048576
1048576 -rw-r--r-- 1 root root 1073741824 Oct 29 01:54 1GB.bin
p2:
total 1048576
1048576 -rw-r--r-- 1 root root 1073741824 Oct 29 01:54 1GB.bin
```

マスタになる 10GB のボリューム上に 1GB のファイルがあり、各スナップショットは使用量 0% の状態で、CoW 前の状態なのでマスタと同じ 1GB のファイルが見えています。

さて、ここで p0 上のファイルを書き換えて見ましょう。

```
# dd if=/dev/zero of=p0/1GB.bin conv=notrunc bs=8k count=128k
131072+0 records in
131072+0 records out
1073741824 bytes (1.1 GB) copied, 144.605 s, 7.4 MB/s
# lvsdisplay
--- Logical volume ---
LV Name                /dev/vg0/p0
...
LV Size                10.00 GiB
...
--- Logical volume ---
LV Name                /dev/vg0/p1
...
CoW-table size        2.00 GiB
CoW-table LE          512
Allocated to snapshot 48.99% <- 書いたのは p0。でも p1 の容量も減っている
...
--- Logical volume ---
LV Name                /dev/vg0/p2
...
CoW-table size        2.00 GiB
CoW-table LE          512
Allocated to snapshot 48.99% <- 書いたのは p0。でも p2 の容量も減っている
```

… という訳で、マスタ側で書き込みを行った所、マスタ側が CoW で分岐するのではなく、全スナップショットで CoW 処理が走ってしまいます。この結果 CoW 時の IO 性能は激減し、更に CoW 後はスナップショット個数だけ増幅する形で容量が埋められてしまいます。

スナップショット側で書き込んだ場合は当該スナップショットだけで CoW するのですが、この非対称的な動作のため、LVM は nilfs に魅力を感じる方が期待する用途で利用することは難しいでしょう^{*15}。

10.10 まとめ

nilfs はかなり実用的に使える水準になっており、私もすでに半年ほど小規模ですがアーカイブサーバーとして実利用中です^{*16}。ただ、本格的な利用にはまだ以下のような部分で機能が不足しています。

1. リサイズがない (オンライン、オフラインとも)
 - これは追記追記で使うにしても容量が増やせず壁にあたってしまうので、困る点です
2. fsck がない
3. フラグメント時や GC 中の性能が劣化する

他にも POSIX ACL や extended attribute がないとか、クォータに未対応であるとか、機能面でも穴がある部分はまだまだあります。

しかし、こういった弱点はあるにしても、これだけの強力なスナップショット機能が安定的に利用できる^{*17} ファイルシステムは現時点で少なく、特に競合 (機能的にはもっと上ですが、この点において) の btrfs が安定してくるまでは nilfs は非常に貴重な存在です (Meego が採用した^{*18}[16] ということなので、btrfs も実は結構使える気が書きながらしていますが、現時点で Debian で評価するといきなりバグ・未実装がある訳なので …)。現在はタイムマシンのストレージを pdumpfs などのツールレベルで実現している方が多いのではないかと思います。nilfs も有力な選択肢として検討してみたいかがでしょうか？

参考文献

- [1] Ryusuke KONISHI, "The NILFS2 Filesystem: Review and Challenges",
<http://www.nilfs.org/papers/jls2009-nilfs.pdf>
- [2] Ryusuke Konishi, "Development of a New Logstructured File System for Linux",
<http://www.nilfs.org/papers/nilfs-051019.pdf>

^{*15} 本質的に不可能という話ではないので、スナップショットのネストができるように拡張されるあたりで改善されるかもしれませんが …

^{*16} 総容量 16GB の Debian-on-USB 箱というささやかなものですが …

^{*17} オンディスクフォーマットも事実上確定で、変更予定なしかつ今後は上位互換で行くとされています [10]

^{*18} こちらもオンディスクフォーマットはこれ以上変えない方針だとか

- [3] Nilfs team, "the Nilfs version 1: overview",
<http://www.nilfs.org/papers/overview-v1.pdf>
- [4] "NILFS2", Documentation/filesystems/nilfs2.txt from Linux kernel source code
- [5] Dongjun Shin. "About SSD", Feb. 2008,
http://www.usenix.org/event/lsf08/tech/shin_SSD.pdf
- [6] "Questions regarding use of nilfs2 on SSDs",
<http://www.mail-archive.com/users@nilfs.org/msg00373.html>
- [7] "Performance about nilfs2 for SSD",
<http://www.mail-archive.com/linux-nilfs@vger.kernel.org/msg00497.html>
- [8] "SSD and non-SSD Suitability",
<http://www.mail-archive.com/linux-nilfs@vger.kernel.org/msg00250.html>
- [9] "cleaner: run one cleaning pass based on minimum free space",
<http://www.mail-archive.com/linux-nilfs@vger.kernel.org/msg00058.html>
- [10] "production ready?",
<http://www.mail-archive.com/linux-nilfs@vger.kernel.org/msg00526.html>
- [11] Valerie Aurora, "A short history of btrfs", Jul 2009,
<http://lwn.net/Articles/342892/>
- [12] "Btrfs FAQ",
<https://btrfs.wiki.kernel.org/index.php/FAQ>
- [13] "Btrfs design - btrfs Wiki",
https://btrfs.wiki.kernel.org/index.php/Btrfs_design
- [14] "Using Btrfs with Multiple Devices",
https://btrfs.wiki.kernel.org/index.php/Using_Btrfs_with_Multiple_Devices
- [15] "Gotchas - btrfs Wiki",
<https://btrfs.wiki.kernel.org/index.php/Gotchas>
- [16] Jonathan Corbet, "MeeGo and Btrfs", May 2010,
<http://lwn.net/Articles/387196/>

11 Btrfs を Debian で活用してみる

鈴木 崇文



11.1 Btrfsってどんなファイルシステム?

Btrfs は、Linux kernel の 2.6.29 から kernel のリリースにも含まれるようになった、新しいコピーオンライト形式のファイルシステムであり、フォールトトレラントや修復機能、容易な管理機能などが備わっています。ZFS の影響を受けていると言われており、Oracle の Chris Mason により GPL で開発がすすめられています。現在はまだ開発中の状態にあります。

なお、今回は Squeeze/Sid を使用して解説しますが、Squeeze/Sid の README^{*19} においても、まだ現時点ではベンチマークとレビュー以外に使用するなどの注意書きがありました。

```
btrfs-tools for Debian
-----

WARNING: Btrfs is under heavy development, and is not suitable for any uses
other than benchmarking and review.

-- Daniel Baumann <daniel@debian.org> Sun, 29 Jul 2007 12:19:00 +0200
```

11.2 Debian でインストールする方法

Debian で Btrfs を使用する手順は、btrfs-tools パッケージをインストールするだけになります。

```
$ sudo apt-get install btrfs-tools
```

11.3 フォーマット・マウント・btrfsck

フォーマットは `mkfs.btrfs` で行えます。

```
# mkfs.btrfs /dev/sda

WARNING! - Btrfs Btrfs v0.19 IS EXPERIMENTAL
WARNING! - see http://btrfs.wiki.kernel.org before using

fs created label (null) on /dev/sda
        nodesize 4096 leafsize 4096 sectorsize 4096 size 500.00MB
Btrfs Btrfs v0.19
```

マウントも通常通り、`mount` を使用できます。

^{*19} /usr/share/doc/btrfs-tools/README.Debian

```
# mkdir /mnt/btrfs1
# mount /dev/sda /mnt/btrfs1
# df -T
Filesystem      Type      1K-blocks    Used Available Use% Mounted on
/dev/sde1       ext3      19272572     2833388 15460192  16% /
tmpfs           tmpfs     517260        0      517260   0% /lib/init/rw
udev           tmpfs     512936        100     512836   1% /dev
tmpfs          tmpfs     517260        0      517260   0% /dev/shm
/dev/sda       btrfs     512000        205092  306908   41% /mnt/btrfs1
```

ここで試しにファイルをコピーしてみると、次のようにコピーオンライトのおかげで高速なコピーがされていることがわかります。

```
# ls -al /mnt/btrfs1/
total 102408
dr-xr-xr-x 1 root root      8 Nov 17 04:14 .
drwxr-xr-x 3 root root    4096 Nov 17 04:01 ..
-rw-r--r-- 1 root root 104857600 Nov 17 04:03 data
# time cp /mnt/btrfs1/data /mnt/btrfs1/data-copy

real    0m0.731s
user    0m0.000s
sys     0m0.556s
# ls -al /mnt/btrfs1/
total 116584
dr-xr-xr-x 1 root root      26 Nov 17 04:14 .
drwxr-xr-x 3 root root    4096 Nov 17 04:01 ..
-rw-r--r-- 1 root root 104857600 Nov 17 04:03 data
-rw-r--r-- 1 root root 104857600 Nov 17 04:14 data-copy
```

ドキュメントには `fsck` はまだ完全には実装されておらず、アンマウントされた状態で FS エクステンツリーのチェックのみが実装されている、と記載されていましたが、実際に実行したところではマウント状態であっても実行できました。なお、「`-a`」オプションが実装されていないため現時点では `fsck.btrfs` へのシンボリックリンクは存在せず、`btrfsck` を直接実行する必要があります。

```
# btrfsck /dev/sda
found 210018304 bytes used err is 0
total csum bytes: 204800
total tree bytes: 303104
total fs tree bytes: 8192
btree space waste bytes: 74736
file data blocks allocated: 209715200
referenced 209715200
Btrfs Btrfs v0.19
```

11.4 複数のディスクを使用してみる

Btrfs では複数のディスクの束ねて使用することもできます。ここでは先ほど作成した `/dev/sda` に `/dev/sdb` を追加してみます。`btrfs device add` で簡単に追加できます。`df` で追加した分の容量が増えていることが確認できます。

```
# df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/sde1       19272572     3414148 14879432  19% /
tmpfs           517260        0      517260   0% /lib/init/rw
udev           512936        100     512836   1% /dev
tmpfs          517260        0      517260   0% /dev/shm
/dev/sda       512000        28      511972   1% /mnt/btrfs1
# btrfs device add /dev/sdb /mnt/btrfs1/
# df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/sde1       19272572     3414148 14879432  19% /
tmpfs           517260        0      517260   0% /lib/init/rw
udev           512936        100     512836   1% /dev
tmpfs          517260        0      517260   0% /dev/shm
/dev/sda       1024000        28     1023972   1% /mnt/btrfs1
```

以下のようにフォーマット時から複数のディスクを束ねてフォーマットすることもできます。

```
# mkfs.btrfs /dev/sda /dev/sdb
(省略)
adding device /dev/sdb id 2
fs created label (null) on /dev/sda
    nodesize 4096 leafsize 4096 sectorsize 4096 size 1000.00MB
Btrfs Btrfs v0.19
# mount /dev/sda /mnt/btrfs1
# df
Filesystem            1K-blocks      Used Available Use% Mounted on
/dev/sde1              19272572    3414152  14879428   19% /
tmpfs                  517260         0    517260    0% /lib/init/rw
udev                  512936         100    512836    1% /dev
tmpfs                  517260         0    517260    0% /dev/shm
/dev/sda               1024000        28    1023972    1% /mnt/btrfs1
```

11.5 バックアップ用のファイルシステムとして試してみる

まずは、サブボリュームを作成します。

```
# btrfs subvolume create /mnt/btrfs1/subvolume
Create subvolume '/mnt/btrfs1/subvolume'
```

この作成したサブボリュームは以下のようにマウントオプション「subvol=」を付けてマウントできるようになります。スナップショットはサブボリュームに対して作成できるので、バックアップが必要な操作はサブボリューム内で行うようにします。ここでは例として、「hello」が入った hello.txt を作成しておきます。

```
# mkdir /mnt/sub
# mount -o subvol=subvolume /dev/sda /mnt/sub/
# echo hello > /mnt/sub/hello.txt
```

次に、先ほど作成したサブボリュームのスナップショットを取ってみます。スナップショットを取る前に sync を実行しておかないと、書き込まれていないデータがある可能性があるため、sync を実行しておきましょう。スナップショットが取れたら hello.txt に「world」を追加書込しておきます。

```
# sync;sync
# btrfs subvolume snapshot /mnt/btrfs1/subvolume/ /mnt/btrfs1/snapshot1
Create a snapshot of '/mnt/btrfs1/subvolume/' in '/mnt/btrfs1/snapshot1'
# echo world >> /mnt/sub/hello.txt
```

すると、次のように hello.txt に差異があり、正常にスナップショットが取れていることがわかります。

```
# cat /mnt/sub/hello.txt
hello
world
# cat /mnt/btrfs1/snapshot1/hello.txt
hello
```

この作成したスナップショットも同様に「subvol=」を使用してマウントできるため、以下のように容易に過去の時点まで戻ってマウントすることができます。

```
# mount -o subvol=snapshot1 /dev/sda /mnt/sub/
# cat /mnt/sub/hello.txt
hello
```

なお、作成したサブボリュームは btrfs subvolume list で表示できるはずでしたが、現時点の Squeeze/Sid ではエラーになってしまいました。

```
# btrfs subvolume list /mnt/btrfs1
ERROR: can't perform the search
```

11.6 まとめ

新しいファイルシステムである Btrfs について説明しました。ところどころひっかかる点はあるものの、まだ開発段階であるとはいえ、一通りの機能は使用できている状態になっています。スナップショットの作成や、ディスクを束ねるのが、簡単なコマンドで実行できるという点は、サーバ管理を行う上で便利な機能といえます。

今後の開発で、開発版のメッセージが無くなり、実運用に使用できるまで成熟することを期待したいところです。

12 分散ファイルシステム CEPH を Debian で活用してみる

服部 武史



12.1 CEPHってどんなファイルシステム?

CEPH は RADOSGW(FastCGI ベースの proxy) と呼ばれる分散オブジェクトストレージ技術に基いている。Ceph は Amazon が使う S3 と互換性のあるインタフェースを librados を用いることで簡単なアクセスを提供する、らしい*20。親サーバーから子サーバーに対し複数の BrtFS を束ねて、一つのファイルシステムであるように動作する。

Ceph が動作するサーバーは BrtFS を持つサーバーに対し SSH 接続を行い、ファイルシステムを構築したりコンフィグデータの交換を行うため、サーバーからクライアントへリモート接続ができる環境が必要となる。今回は附属の SSH 接続を用いた。

12.2 インストールした環境

以下のマシンを 5 台で動作させた。親サーバーは BrtFS も兼ねさせた。

- HARD: intel
- CPU: XEON 3.2GHz
- MEM: 2048Gbyte
- NIC: 100M/FULL
- KERNEL: 2.6.36

12.3 インストール方法

```
# apt-get install automake autoconf gcc g++ libboost-dev libedit-dev libssl-dev libtool libfcgi libfcgi-dev libfuse-dev \
linux-kernel-headers libatomic-ops-dev btrfs-tools libexpat1-dev openssh-server
% tar xzpf ceph-0.22.2.tar.gz
% cd ceph-0.22.2
% ./configure --prefix=/usr/local
% make -j 4
% make install
# cp -rpi src/ceph_common.sh /etc/init.d/
```

12.4 設定

コンフィグは以下のように、BrtFS が動作するサーバーを指定するだけの簡単な設定を行う。

*20 未確認。 <http://ceph.newdream.net/> より

- mon... クラスタ管理サーバー
- osd... データの保存先サーバー
- mds... メタデータ管理サーバー

```
# mkdir /etc/ceph
# touch /etc/ceph/ceph.conf
# ln -s /etc/ceph/ceph.conf /usr/local/etc/ceph/ceph.conf
```

設定内容は以下のとおり。

```
[global]
    pid file = /var/run/ceph/$name.pid
    debug ms = 0

[mon]
    mon data = /data/mon$id

[mon1]
    host = sv1
    mon addr = 172.25.3.172:6789

[mon2]
    host = sv2
    mon addr = 172.25.3.175:6789

[mon3]
    host = sv3
    mon addr = 172.25.3.174:6789

[mon4]
    host = sv4
    mon addr = 172.25.3.173:6789

[mon5]
    host = sv5
    mon addr = 172.25.3.210:6789

[mds]
    debug mds = 0

[mds1]
    host = sv1

[mds2]
    host = sv2

[mds3]
    host = sv3

[mds4]
    host = sv4

[mds5]
    host = sv5

[osd]
    sudo = true
    osd data = /data/osd$id
    osd journal = /data/osd$id/journal
    osd journal size = 100
    debug osd = 0
    debug filestore = 0

[osd1]
    host = sv1
    btrfs devs = /dev/loop7

[osd2]
    host = sv2
    btrfs devs = /dev/loop7

[osd3]
    host = sv3
    btrfs devs = /dev/loop7

[osd4]
    host = sv4
    btrfs devs = /dev/loop7

[osd5]
    host = sv5
    btrfs devs = /dev/loop7
```

また、BrtFS を持つサーバー群には親サーバーから SSH で自動接続するために以下のように親に子供のパブリック keys を登録しておく。

```
# ssh-keygen -t rsa
# cat .ssh/id_rsa >> .ssh/authorized_keys
# cat sv1_id_rsa >> .ssh/authorized_keys
# cat sv2_id_rsa >> .ssh/authorized_keys
# cat sv3_id_rsa >> .ssh/authorized_keys
# cat sv4_id_rsa >> .ssh/authorized_keys
# cat sv5_id_rsa >> .ssh/authorized_keys
```

12.5 起動

Ceph ファイルシステムの構築

```
# mkcephfs -c /etc/ceph/ceph.conf --allhosts --mkbtrfs
# /etc/init.d/ceph --allhosts start
# mount -t ceph "172.25.3.172":/ /mnt/
```

12.6 テスト

テスト方法として5台のマシンそれぞれのHDDをBrfFSでフォーマットする方法と、5台のマシンそれぞれで組まれているMD(RAID1)上のImageをloopデバイスにmountした状態で、それをBrfFSにフォーマットしたマシン5台と性能をBonnieと単純なddで比較した。

12.6.1 raw デバイス (単純に今回テストするHDDを単体でテストした結果)

```
Using uid:0, gid:0.
Writing with putc()...done
Writing intelligently...done
Rewriting...done
Reading with getc()...done
Reading intelligently...done
start 'em...done...done...done...
Version 1.03d      -----Sequential Output----- --Sequential Input- --Random-
                  -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine          Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
sv1              4G 38262 96 43978 21 21875 7 44260 88 60659 6 290.9 1
```

12.6.2 loop デバイス (MD上のimageをCephで提供されたものをマウントした場合の結果)

```
sv1:/home/admin# bonnie++ -d /mnt/ -n 0 -u root -b
Using uid:0, gid:0.
Writing with putc()...done
Writing intelligently...done
Rewriting...done
Reading with getc()...done
Reading intelligently...done
start 'em...done...done...done...
Version 1.03d      -----Sequential Output----- --Sequential Input- --Random-
                  -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine          Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
sv1              4G 9869 21 9551 1 5009 1 10264 21 12659 1 255.6 2
```

12.6.3 dd(bonnieではなくddで書いた場合)

```
sv1:/home/admin# dd if=/dev/zero of=gomi bs=1024k count=1000
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 13.1775 s, 79.6 MB/s
```

12.6.4 sdb(HDD を 1 台まるまま ceph に渡した場合)

```
sv1:/home/admin# bonnie++ -d /mnt -n 0 -u root -b
Using uid:0, gid:0.
Writing with putc()...done
Writing intelligently...done
Rewriting...done
Reading with getc()...done
Reading intelligently...done
start 'em...done...done...done...
Version 1.03d      -----Sequential Output----- --Sequential Input- --Random-
                  -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine          Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
sv1 4G 5073 11 4746 0 4439 1 10155 20 12536 1 301.7 2
```

12.6.5 sdb dd テスト (bonnie ではなく dd でテストした結果)

```
v1:/mnt# dd if=/dev/zero of=gomi bs=1024k count=1000
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 172.361 s, 6.1 MB/s
```

テスト結果より、書きこみ性能及 Read 性能は Ceph に HDD まるまま渡すより特定のデバイス上で用意したループデバイスの方が性能適に良いと思われる。

尚、上記テスト中以下のようなログを吐き切離されるような事象が発生していた。

```
Nov 17 04:34:24 sv1 kernel: ceph: osd5 up
Nov 17 04:34:24 sv1 kernel: ceph: osd5 weight 0x10000 (in)
Nov 17 04:34:51 sv1 kernel: ceph: osd5 down
```

上記事象が発生したあと、ceph を終了させて、再度起動するとマウントができず以下になってしまうことがあった。

```
sv1:/# mount -t ceph "172.25.3.172":/ /mnt/
mount: 172.25.3.172:/: can't read superblock
```

上記の事象になってしまった場合は、再度 ceph を構築しなおすとマウントできるようになるがデータはまっさらになってしまう。

12.7 対障害性について

Ceph ファイルシステム上に膨大なディレクトリツリーを copy しながら、通信ができない状態にして copy 状態を確認した。

すると以下のようなログを繰り返しながら永遠に copy されない状態が継続した。

```
Nov 18 03:18:41 sv1 kernel: ceph: osd1 down
Nov 18 03:18:41 sv1 kernel: ceph: osd4 down
Nov 18 03:19:46 sv1 kernel: ceph: tid 69773 timed out on osd2, will reset osd
Nov 18 03:19:46 sv1 kernel: ceph: tid 69799 timed out on osd3, will reset osd
Nov 18 03:19:46 sv1 kernel: ceph: tid 69838 timed out on osd5, will reset osd
Nov 18 03:20:06 sv1 kernel: ceph: osd1 up
Nov 18 03:20:06 sv1 kernel: ceph: osd4 up
```

通信状態を復旧させると、copy を再開した。自動的に切離されたりすると嬉しいのだが。。

P.S. 以下は Ceph ファイルシステムに適当な Debian のミラーディレクトリを消したり作ったりしていた際、以下ログが発生し umount もできずリブートする羽目になった例。バグの原因までは追及できておりません。

```

Kernel BUG at f84e50c8 [verbose debug info unavailable]
invalid opcode: 0000 [#1] SMP DEBUG_PAGEALLOC
last sysfs file: /sys/module/libcrc32c/initstate
Modules linked in: ceph loop nfsd lockd auth_rpcgss sunrpc exportfs tun dm_snapshot dm_mirror dm_region_hash dm_log shpchp
pci_hotplug sd_mod sg mptspi mptscsih mptbase scsi_transport_spi scsi_mod e1000 skge btrfs crc32c libcrc32c
[last unloaded: scsi_wait_scan]

Pid: 28241, comm: cosd Tainted: G          W   2.6.36 #1 SE7520JR22S/_To Be Filled By O.E.M._
EIP: 0060:[<f84e50c8>] EFLAGS: 00210286 CPU: 1
EIP is at btrfs_truncate+0x45b/0x486 [btrfs]
EAX: ffffffff EBX: f2312ba8 ECX: 00000000 EDX: 00000070
ESI: 00000000 EDI: c842d7f0 EBP: ccda4ecc ESP: ccda4e88
DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
Process cosd (pid: 28241, ti=ccda4000 task=cd10ac50 task.ti=ccda4000)
Stack:
 00001000 00000000 dbf68cf8 00000000 00000001 00000000 dbf68d24 00000001
<0> 00000712 00000000 dbf68f40 c842d7f0 dbf68e6c 00000000 00000000 00000000
<0> dbf68e6c ccda4ee4 c105d417 00000000 dbf68e6c ccda4f34 f2312ba8 ccda4f08
Call Trace:
 [<c105d417>] ? vmtruncate+0x37/0x40
 [<f84e5316>] ? btrfs_setattr+0x223/0x269 [btrfs]
 [<c1088ae3>] ? notify_change+0x14f/0x23b
 [<c1077748>] ? do_truncate+0x62/0x7b
 [<c11c1ed8>] ? _raw_spin_unlock_irq+0x8/0xb
 [<c1077a6e>] ? do_sys_truncate+0x186/0x18c
 [<c1077a85>] ? sys_truncate64+0x11/0x13
 [<c1002610>] ? sysenter_do_call+0x12/0x26
 [<c11c0000>] ? dump_stack+0x39/0x61
Code: 8b 4d ec 83 79 28 00 74 11 89 ca 89 d8 e8 19 f0 ff ff 85 c0 74 04 0f 0b eb fe 8b 4d ec 89 d8 8b 55 e8 e8 6a a1 ff ff
85 c0 74 04 <0f> 0b eb fe 8b 55 e8 89 d8 8b 7b 1c e8 18 6c ff ff 85 c0 74 04
EIP: [<f84e50c8>] btrfs_truncate+0x45b/0x486 [btrfs] SS:ESP 0068:ccda4e88
---[ end trace dad60256e5a2b66e ]---
cosd used greatest stack depth: 1176 bytes left

```

13 initramfs について

西山 和広



13.1 initrd/initramfs とは?

- Linux の起動途中に使われる root ファイルシステム
- この中で本当の root ファイルシステム (real root) をマウント
- 実体は gzip された cpio アーカイブ
 - 昔は ext2 のディスクイメージファイル
 - gzip の代わりに lzma のこともある (casper/initrd.lz など)

13.2 いろいろなところからの Linux の起動

13.2.1 USB/HDD から起動

- BIOS (起動順位で USB/HDD が上)
- MBR (grub などのブートローダー)
- vmlinuz (カーネル) + initrd (の中の /init)
- real root (/dev/sda1 とか) の /sbin/init (MD, LVM2, LUKS などでも OK)
- /etc/inittab の処理とか

13.2.2 光学ドライブから起動

- BIOS (起動順位で光学ドライブが上)
- El Torito (isolinux などのブートローダー)
- vmlinuz (カーネル) + initrd (の中の /init)
- real root (filesystem.squashfs + aufs とか) の /sbin/init
- /etc/inittab の処理とか

13.2.3 ネットワークから起動

- BIOS (起動順位で NIC が上)
- PXE boot (pxelinux などのブートローダー)
- vmlinuz (カーネル) + initrd (の中の /init)
- real root (NFS とか) の /sbin/init
- /etc/inittab の処理とか

13.2.4 real root の場所

カーネルに `root=UUID=xxx` などで指定

- ローカルディスク (`root=/dev/sda1` など)
- 光学ドライブ (`root=/dev/hdc` など)
- NFS (`nfsroot=192.168.0.1:/path/to/nfsroot` など)

など

13.2.5 /proc/cmdline

- カーネルのコマンドライン引数
- カーネルパラメーター
- 起動後に `/proc/cmdline` で見えるもの
- `grub` などで `vmlinuz` の後ろに書いているもの
- `initramfs` の処理で使うものが多い
- カーネル自体が処理するものもある
- `real root` で起動するプログラムが参照する目的で使っても良いが、カーネルや `initramfs` が使うものと衝突しないように注意

13.2.6 /proc/cmdline で良く使われるものの例

`quiet`

起動中のコンソールへの出力を減らす

`ro / rw`

`initramfs` の中で `real root` を `readonly mount` するかどうか

`init=/path/to/real_init`

`/sbin/init` の代わりに実行するプログラムを指定

`acpi=off` `apm=off` など

カーネルが処理

`text`

`/etc/init.d/gdm3` が「`grep -wqs text /proc/cmdline`」でチェックしている

`root=/path/to/blockdevice`

ルートファイルシステムとしてマウントするデバイスを指定

`boot=local / boot=nfs / boot=casper / boot=live`

`real root` を `mount` するのに使うスクリプトを指定

13.3 initramfs について

13.3.1 update-initramfs

- `initramfs` はパッケージの中身ではない
- `update-initramfs` コマンドで生成や更新 (カーネルのパッケージのインストール時などは `dpkg-trigger` で遅延実行)
- `/usr/share/initramfs-tools/` や `/etc/initramfs-tools/` を元に生成
- 「`sudo update-initramfs -u -k all`」などで更新

13.3.2 initramfs の調べ方

```
mkdir -p /tmp/initrd && (cd /tmp/initrd && { zcat /boot/initrd.img-* | cpio -idm; })
# initramfs-tools(8)
mkdir tmp/initramfs
cd tmp/initramfs
gunzip -c /boot/initrd.img-2.6.18-1-686 | \
cpio -i -d -H newc --no-absolute-filenames
# linux-2.6/Documentation/initrd.txt
mkdir /tmp/imagefile
cd /tmp/imagefile
gzip -cd /boot/imagefile.img | cpio -imd --quiet
```

などのように展開 (カレントディレクトリにばらまかれるので展開場所には注意)

13.3.3 initramfs の中身

./init

カーネルが実行するプログラム

- Debian 系の場合は /bin/sh のシェルスクリプト
- Redhat 系の場合は以前確認したときは nash だった

./scripts

./init から読み込まれたり実行されたりするプログラム

./bin や ./sbin

busybox など

./conf や ./etc

設定ファイル

./lib や ./usr

ライブラリやカーネルモジュールなど

13.4 initramfs のカスタマイズ

- /etc/initramfs-tools/ 以下のファイルを編集したり、ファイルを追加したり。
- Debian Live のようにパッケージなら /usr/share/initramfs-tools/ 以下にファイルを追加する。

13.4.1 /etc/initramfs-tools

initramfs.conf, update-initramfs.conf

update-initramfs や mkinitramfs の設定ファイル

modules

initramfs の中でロードするモジュール (後でロードすればいいものは /etc/modules を使う)

conf.d/

initramfs の conf/conf.d/ に入る。

hooks/

/usr/share/initramfs-tools/hooks/ と同様に initramfs 作成時に実行される。

scripts/

/usr/share/initramfs-tools/scripts/ と一緒に initramfs の scripts/ に入る。

13.4.2 hooks/

/usr/share/initramfs-tools/hook-functions の関数で initramfs の中身を生成

- `copy_exec` で追加しておきたいバイナリをコピー (`ldd` でわかる範囲内で使っているライブラリを含めてコピーされる)
- `manual_add_modules` でモジュールをコピー
- その他のファイルを追加や削除など

13.5 `scripts/`

- カスタマイズする処理本体
- `live` 関係なら以下のようなものが入る。(BOOT=live のときに使われる)
 - `live`
 - `live-bottom/`
 - `live-functions`
 - `live-helpers`
 - `live-premount/`

13.6 `./init` の処理内容

- “Loading, please wait...” のメッセージが出たところから `initramfs` 中の処理
- `initramfs` 内の ディレクトリの準備
- `initramfs` 内の `/dev` や `udev` の準備
- 後で起動するシェルスクリプトなどのために、いくつかのシェル変数を環境変数に
- `conf` ファイルの読み込み
 - `./conf/initramfs.conf`
 - `./conf/conf.d/`
- カーネルのコマンドライン引数 (`/proc/cmdline`) の解析
- `noresume` と `netconsole` を処理
- `maybe_break top`
- (`scripts/functions` の中で `maybe_break` や `panic` や `run_scripts` がある)
- `run_scripts /scripts/init-top`
- `./conf/modules` に書いてあるモジュールをロード
- 「`./scripts/${BOOT}`」で定義される `mountrout` を実行して `real root` をマウント (普通の起動時は `./script/local`)
- 最初の方でマウントした `sysfs` と `proc` を本当の `root` ファイルシステム (`real root`) に移動
- プロセス ID 1 の `init` になるプログラムの存在やパーミッションをチェック
- `init` に不要な環境変数を `unsetenv`
- `real root` に移行



14 最近の Debian Live の動向

のがたじゅん

2009 年 6 月の関西 Debian 勉強会で Debian Live についてお話ししましたが、あれから、Debian Live の状況がかなり変わったのでまとめてみました。

14.1 Debian Live とは (おさらい)

改めて Debian Live を説明すると、書き込み不可 (リードオンリー) のメディアなどから起動する Debian システムです。Debian Live を作成するには live-build を使い、ライブシステムの起動を補助する live-boot と live-config を組み込んだシステムを作成します。

14.2 Debian Live 1.x から 2.x/3.x の変更点

Debian Live の解説が若干変わったことに気づいた人はいらっしゃるでしょうか。ここでは Debian Live 1.x(Lenny) から Debian Live 2.x(Squeeze)/3.0(Whizzy) の変更点について述べます。

基本的な作成方法などは Debian Live 1.x と変わらないので、Debian Live を以前使ったことがある人も、ここに書いてある変更点を変更するだけでレシピを移行できます。

14.2.1 Debian Live 作成ツールの live-helper が live-build に変更

大きな変更の一つめは、Debian Live を作成するツールが live-helper から live-build に変わりました。

live-helper からの変更としてはツールの名前が変更されたほかに、コマンドも変更され、live-helper では「lh_config」のように頭に lh_がついたコマンドを直接実行していましたが、live-build では debhelper7 のように lb コマンドにコマンドに与えて呼び出す形に変更されました。

内部的にはかなりの変更がありましたが、通常使用するには変わっていないので、以前のレシピがある人は「lh_」を「lb 」に置き換えるだけで問題なく使えます。

live-helper	live-build
lh_config	lb config
lh_build	lb build
lh_clean	lb clean

14.2.2 live-initramfs が live-boot と live-config に変わった

大きな変更の一つめは、live-initramfs が live-boot と live-config に置き換えられました。

1.x 系で使われる live-initramfs は、Debian Live 起動時にライブシステム特有の設定をおこなうスクリプトです

が、2.x/3.x 系からは機能が分割され、システム起動時の早い段階、ライブメディアにあるルートファイルシステムをマウントなどをおこなう live-boot と、ライブシステム上でデーモンなどの起動設定をおこなう live-config に新たに書き直されました。

分けられた理由としては、live-initramfs の見通しの悪さと起動の遅さがありました。live-initramfs はもともと ubuntu が開発した casper からフォークし機能を追加していましたが、Debian Live が意図する様々なデバイスからの起動が考えられていなかったため拡張していくと見通しが悪くなってきていました。

そして、live-initramfs では initrd の中でおこなわなくてもよいデーモンの起動設定などの処理を initrd の中でおこなっていて、起動時間の足を引っ張っていました。

これらの理由から置き換えられたのですが、結果、見通しがよくなり起動も高速化することになりました。

起動の高速化では、Squeeze 以降デーモンの並列起動化と insserv の導入もあって相乗効果でかなり早くなりました。(Squeeze Live Alpha2 のリリースノートでは 1 分半から 54 秒になったと出ていましたが個人的にはもっと早いように感じます)

live-initramfs は init だけの対応でしたが、live-boot からは init 以外に ubuntu で使われている upstart、fedora で使われている systemd にも対応しました。

-bootappend-live に与えるオプションについてですが、大幅に変わったので live-boot と live-config の man を参照にして書き換える必要があります。例として日本語キーボードを指定するオプションを書いておきます。

live-helper	live-build
kmodel=jp106 keyb=jp	keyboard-model=jp106 keyboard-layouts=jp

オプションについては -bootappend-live オプションで指定する以外にも、live-boot の場合は live/boot.conf や live/boot.d/以下、live-config の場合は live/config.conf や live/config.d/以下にオプションを書いたファイルを置くのと起動時に読み込まれるようになりました。

日本語環境を指定する場合、config/binary_local-includes/live/config.conf に以下のように書いておく -bootappend-live オプションの指定がなくなるのですっきりすると思います。

```
LIVE_LOCALES=ja_JP.UTF-8
LIVE_TIMEZONE=Asia/Tokyo
LIVE_UTC=no
LIVE_KEYBOARD_MODEL=jp106
LIVE_KEYBOARD_LAYOUTS=jp
```

オプション関係については man にかなり丁寧に書いてあるので一度読んでおくといいでしょう。

14.2.3 パッケージリストの指定方法が変わりました

Debian Live にインストールしたいパッケージを一括して指定するパッケージリストの指定方法が変わりました。

1.x 系では config/chroot_local-packageslists/ にパッケージリストのファイルを置き、-packages-lists オプションで置いたリストファイルの名前を列挙していましたが、2.x/3.x 系からは config/chroot_local-packageslists/ に拡張子「.list(ドットリスト)」をつけたファイルを置くだけで適用されるようになりました。

live-helper	live-build
config/chroot_local-packageslists/examplelist lh_config -packages-lists "examplelist"	config/chroot_local-packageslists/examplelist.list

14.2.4 ISO と HDD 両用のバイナリイメージを作成するオプションの追加

今まで live-helper では ISO イメージ (iso) かハードディスクイメージ (usb-hdd) どちらかのイメージしか作成できませんでしたが、live-build からは -binary-images オプションに iso-hybrid を指定して ISO,HDD どちらにも利用

できるオプションが追加されました。

例のように iso-hybrid で作成したイメージを dd で USB メモリに書き込んで使うことができます。

```
$ lb --binary-image iso-hybrid
$ sudo lb build
$ dd if=binary-hybrid.iso of=/dev/sdX bs=1M
```

14.2.5 オプションの有効/無効の指定が enabled/disabled から true/false に変更

細かい変更ですが、live-build のオプション有効/無効の指定方法が enabled/disabled から、true/false に変わりました。以前のままだでもエラーが出ない (デフォルトの設定が使われてしまう) ので気をつけましょう。

live-helper	live-build
-apt-recommends disabled	-apt-recommends false

14.2.6 自動化スクリプトを置くディレクトリが scripts から auto に変更

こちらも細かな変更ですが自動化スクリプトを置くディレクトリが scripts から auto ディレクトリに変わりました。これはディレクトリ名をリネームするだけで OK です。

14.2.7 自動化スクリプトの呼び出しオプションが noautoconfig から noauto に変更

auto ディレクトリに置いた自動化スクリプトから呼び出すオプションが、noautoconfig から noauto に変わりました。以前のままだですとループに入るので気がつくと思います。

live-helper	live-build
lh_config noautoconfig	lb config noauto

14.2.8 パッケージのセクションを指定する-categories オプションが-archive-areas に変更

main や contrib などのパッケージセクションを指定する-categories オプションが-archive-areas に変わりました。こちらも lb config を実行したときにエラーが出るので気がつくと思います。

live-helper	live-build
lh_config -categories "main contrib non-free"	lb config -archive-areas "main contrib non-free"

14.3 Debian Live の新機能など

Debian Live の新機能などについて述べます。

14.3.1 live-installer と live-install-launcher

live-installer は Debian Installer を使って Debian Live の内容をそのままインストールする d-i のモジュールです。設定方法は簡単で-debian-installer オプションに live を指定して作成すれば使えます。

```
$ lb config --debian-installer live
```

live-installer のメリットとしては Debian のインストールが簡単になることが挙げられます。

live-installer を使ったインストールでは、ライブメディアに保存したパッケージを使ってインストールを行うので、ネットワークの設定を DHCP にまかせると、ユーザーと root の設定とパーティションの設定以外することがありま

せん。(apt-line は Live の設定が使われます。)これはかなりおすすめなので、一度試してみてください。

live-install-launcher は、Debian Live 上で Debian Installer を起動するランチャーです。ようやく GUI インターフェースの d-i が動きましたが、安定してインストールするにはまだ時間がかかりそうだと思っていたら、Debian Live イメージの Daily Build では外されてしまいました。

14.3.2 live-build-cgi, live-studio

live-build-cgi と live-studio は live-build の web インターフェースで、ブラウザから設定して Debian Live の作成ができます。この 2 つについてですが、まだチェックできておらず Debian Live のサイトにあるものを試してみただけです。

live-build-cgi は、素の live-build に近い感じで設定項目も細かく設定できますが、解説がないと使うのは難しいように思いました。

live-studio は django で書かれている Web インターフェースです。設定項目は少なくパッと見のとっつきはいいのですが、GUI インターフェースの live-magic に似せてあるせいか日本語環境に適した設定ができないので、こちらもまだまだ微妙な感じでしょう。

14.4 Debian Live tips

ここでは Debian Live の作成、使用についての Tips について述べます。

14.5 Tips 1:一発で日本語環境入りの Debian Live を作る

以前は日本語環境に必要なパッケージをインストールするにはパッケージリストを作っていました。パッケージリストを作らなくとも aptitude の tasksel を使えば日本語環境に必要なパッケージを一発でインストールできます。

設定方法は `-tasksel` オプションに「`aptitude`」、`-tasks` オプションに `aptitude` のタスク (例で言うなら「`japanese japanese-desktop`」) を指定します。

```
$ lb config --tasksel aptitude --tasks "japanese japanese-desktop"
```

GNOME を使う場合は `japanese-gnome-desktop`、KDE を使う場合は `japanese-kde-desktop` も合わせて指定しておくといいでしょう。

指定できるタスクの一覧は `tasksel-data` パッケージに入っている `/usr/share/tasksel/debian-tasks.desc` に書かれています。

14.6 Tips 2:Debian Live をネットワーク上から起動する

Debian Live は CD/DVD や USB メモリなどのデバイス以外に、PXE ブートと NFS を利用してネットワーク越しに起動することもできます。

Debian Live のイメージを置くサーバーの設定は、ネットワーク越しに Debian インストーラを起動する設定とほぼ同じなので参考にしてください。^{*21}

14.6.1 ネットワーク起動用 Debian Live の作成

基本的には通常の Debian Live 作成と変わりませんが、`-binary-images` オプションに「`net`」、`-net-root-server` オプションに Debian Live イメージを置くサーバーのアドレス、`-net-root-path` オプションに Debian Live イメージのパスを指定して作成します。

```
$ lb config -b net --net-root-server "192.168.0.3" --net-root-path "/srv/debian-live"
```

^{*21} 4.5. TFTP ネットブート用ファイルの準備: <http://www.debian.org/releases/testing/i386/ch04s05.html.ja>

ビルドすると binary-net.tar.gz というアーカイブファイルができます。

14.6.2 ネットワーク起動用サーバーをインストール

Debian Live イメージを置くサーバーを設定します。起動させるには DHCP サーバーと TFTP サーバー、カーネル起動後本体をマウントするため NFS サーバーが必要になるのでインストールします。

```
# aptitude install dhcp3-server tftpd-hpa nfs-kernel-server
```

インストール後、/srv ディレクトリが出来ているので、この下に作成した Debian Live のイメージ binary-net.tar.gz を展開します。

```
# cd /srv/  
# tar xvfj binary-net.tar.gz
```

14.6.3 DHCP サーバーの設定

/etc/dhcp/dhcpd.conf の末尾あたりに配布する IP の範囲などを書きます。

```
subnet 192.168.100.0 netmask 255.255.255.0 {  
    range 192.168.100.51 192.168.100.61;    dhcp を配布するレンジ  
  
    option routers 192.168.100.1;    ゲートウェイ  
    option domain-name-servers 192.168.100.100;    ネームサーバー  
    option broadcast-address 192.168.100.255;    ブロードキャスト  
    option subnet-mask 255.255.255.0;    サブネットマスク  
  
    filename "pxelinux.0";    ブートするファイル名  
}
```

/etc/default/isc-dhcp-server に DHCP サーバーに使うインターフェース名を書きます。

```
INTERFACES="eth0"    インターフェース名を書く
```

DHCP サーバーを再起動します。

```
# service isc-dhcp-server restart
```

14.6.4 TFTP サーバーの設定

/etc/default/tftpd-hpa の設定を変更します。

```
TFTP_DIRECTORY="/srv/tftpboot"    デフォルトでは"/srv/tftp"になっているのを変更
```

TFTP サーバーを再起動します。

```
# service tftp-hpa restart
```

14.6.5 NFS サーバーの設定

/etc/exports に展開した Debian Live 本体のパスを書きます。

```
/srv/debian-live *(ro,async,no_root_squash,no_subtree_check)
```

設定を反映させます。

```
# exportfs -rv
```

14.6.6 クライアントマシンの設定

クライアントマシンの BIOS 設定をネットワークブートを最初にして起動します。

14.6.7 NFS 上に差分を保存する (未完)

Debian Live 本体を NFS を使ってマウントするということは、書き込みができる差分領域も NFS 上に保存できるのかと言えます。

保存するには `lb config` に `--net-cow-server` オプションに保存サーバーのアドレス、`--net-cow-path` オプションに保存サーバーのパスを指定して作成するか、Live の起動オプションに「`nfscow=(サーバーアドレス):(サーバーパス)`」と指定します。

```
$ lb config --net-cow-server "192.168.0.3" --net-cow-path "/srv/live-rw"
```

ですが試してみたところ、NFS 領域を `aufs` でマウントするとカーネルがエラーメッセージをバカバカ吐いて死んでしまうのでした。

14.7 Squeeze の Debian Live はどうなるのか

正直なところわかりません。

Squeeze と同時にリリースされるのは確実ですが、当初目玉として予定されていた `live-install-launcher` が Squeeze のフリーズに間に合わず、何日か前の `autobuild` のビルドイメージには入っていたものも外されたので Whizzy 以降になると思われます。

Syslinux の起動画面も去年の `debconf` で提案として発表されていたおしゃれなロゴになっていましたが、これも通常の Debian ロゴに戻されました。

このことから Squeeze の Debian Live は Lenny と同じくライブシステムのためのリリースとなる模様です。

14.8 まとめ

前回の発表で基本的なところはカバーできたかなと思っていたら、ちゃぶ台返してガッツリ変わってしまいました。

Debian Live がこの先どうなるかわかりませんが (Squeeze が安定する前に「次は 3.0 でいっけ!なんて言うぐらいだし」)、これがみなさまの助けになれば幸いです。

15 Debian Backports の使い方

佐々木洋平



15.1 新しい魅力的なソフトウェア

unstable には私たちがわくわくされるようなソフトウェアが毎日ようにインストールされていく。

15.1.1 例 (1) ibuz-mozc

open-source project originates from **Google Japanese Input**.

- ITP: Bug#581158
 - もうすぐ unstable に入る Yo! *22

15.1.2 例 (2) chromium-browser

Chromium is an open-source browser project originates from **Chrome**.

- unstable なら apt 一発

```
% sudo apt-get install chromium-browser
```

15.2 魅力的な更新

- emacs **23**
- gnome **2.30**
- KDE **4.4.5**
- ruby **1.9.2**
- python 2.7, python3 ...

15.3 最新の が使いたい!

stable に無いなら unstable を使えば良いじゃない。

15.4 でも unstable はちょっと怖い!? そんな貴方に

1. Debian Backports の紹介

*22 既に unstable に入っていますが、non-free 扱いです。

2. apt の「ピン止め」

15.5 Debian Backports

<http://backports.debian.org/>

- 2010/09 から Debian の正式なサービスになった。^{*23}
- testing, unstable のソースを stable 向けに再コンパイルしたパッケージを提供
 - security-update にも対応 (BSA (Backports Security Announcement))
 - 公式な安定版よりも新しいバージョンが使える, かも
- 普段は stable を使っているけれども, 特定のソフトウェアは新しいバージョンを使用したい時にオススメ

15.6 backports を使うなら...

- apt-line に以下を追加

```
deb http://backports.debian.org/debian-backports lenny-backports main contrib non-free
```

- 更新 & target を指定して install

```
% sudo apt-get update
% sudo apt-get install emacs23 -t lenny-backports
```

- apt-get upgrade すると
 - lenny-backports のパッケージに upgrade される
- それが嫌なら apt のピン止め を行なうべし

15.7 apt のピン止め

- 特定のパッケージ/リリースに対して 優先度 を設定
- 優先度に応じて, apt がパッケージを処理する.
 - インストール対象にしない
 - 明示的に指定すればインストール可能
 - アップグレードの対象にはならない
 - ダウングレードしてでも, そのリリースを install する... などなど

^{*23} <http://lists.debian.org/debian-announce/2010/msg00012.html>

15.8 apt pin の優先度

優先度	意味
0	install しない
1-99	指定すれば install 可能 upgrade の対象にはならない
100	現在 install されているパッケージ
(500)	現在 install されていないパッケージ
(989)	apt-pin の default
990	apt-get の target-release が指定されている場合
1000 以上	ダウングレードしてもそのパッケージを install

15.9 backports 用の pin 止めの例

/etc/apt/preferences に以下の用を書く?

```
Package: *  
Pin: release a=lenny-backports  
Pin-Priority: 99
```

- release が lenny-backports の
- 全てのパッケージ (*) を
- 優先度 99 にピン止め
 - 明示的に指定すればインストール可能
 - アップグレードの対象にはしない
- さらに pin 止めすると apt-line に testing/unstable があっても安心

```
Package: *  
Pin: release a=lenny-backports  
Pin-Priority: 99
```

```
Package: *  
Pin: release a=testing  
Pin-Priority: 98
```

```
Package: *  
Pin: release a=unstable  
Pin-Priority: 97
```

15.10 野良 backports の作り方

欲しいパッケージが

backports に無いよ? 野良 backports を作ろう!

- 材料
 1. 安定版の動いているマシン
 2. testing/unstable の ソースパッケージ
- レシピ
 1. apt の pin 止めをする
 2. apt-get source パッケージ名 -t testing(unstable)
 3. 依存するパッケージを install してパッケージ作成
 - パッケージが古くて依存関係が満たせない
 - 2. に戻ってくりかえし



16 dh ~source format 3.0, the magic debhelper rules~

佐々木洋平

次期安定版 6.0(squeeze) での RELEASE GOALS の一つに「new source package format support」があります*24。ここでの「new source package format」が source format 3.0 です。

debhelper は deb パッケージの構築を補助する様々なツール群です*25。多くのパッケージでは debhelper の呼び出しはテンプレ化されているので共通している debhelper の呼び出しを隠蔽しパッケージ作成のルール (debian/rules) を簡単にするのが dh(コマンド) と CDBS(Makefile のルールセット) です。

ここでは source format 3.0, dh, CDBS についてお話しします。ちなみに、告知タイトルに「深追い」とか「CDBS 2.0」とかいう文字列があった気がしますすが気にしないで下さい。

16.1 source format 1.0 → 3.0

16.1.1 1.0 の問題点

source format 3.0 の前に、これまでの source format 1.0 について簡単に復習しましょう。debian のソースパッケージは主に以下の構成物からなります:

```
.orig.tar.gz
    オリジナルのソース一式
.dsc
    パッケージの情報 (パッケージの説明、メンテナ、ファイルのハッシュなど)
.diff.gz
    バイナリパッケージをビルドするための変更点。Debian 固有のパッケージの場合には存在しない。
```

さて、このソースフォーマットの構成には、以下の問題点があります:

1. アーカイブの圧縮形式として .gz しか使えない。
2. upstream のソースが複数のアーカイブから構成される場合が面倒。
3. パッケージメンテナの作成したパッチが全部単一のファイルにまとめられており、混然としている。
4. diff で表現されるので (画像などの) バイナリが置きにくい。

source format 1.0 の場合、これらの問題点は、だいたい以下の様に解決していました:

1. アーカイブの圧縮形式として .gz しか使えない。

*24 <http://release.debian.org/squeeze/goals.txt>

*25 deb パッケージそのものは debhelper 無しでも作成できますが、普通は debhelper を使います。

例えば upstream が bzip2 で配布されている場合でも gzip で圧縮しなおす、もしくは tarball in tarball *26 などの形式にする、などです。後述の CDBS を使用する場合には tarball in tarball 用のルールとして tarball.mk が用意されています。

2. upstream のソースが複数のアーカイブから構成される場合が面倒。

前述の tarball in tarball で対応しています。

3. パッケージメンテナの作成したパッチが全部単一の patch に。

diff.gz は debian 以下のファイルが全て単一のファイルになっています。debian/control や debian/rules だけではなく、upstream のソースへのパッチも全て一つのファイルになっています。そこで、upstream のソースへのパッチは debian/patches 以下に意味のある単位で分割して配置し、パッケージ作成の際にパッチの apply/unapply を行なうことにしています。このためのツールとしては dpatch や quilt *27 があります。また、CDBS にはパッチの apply/unapply をするためのルールとして patchsys-quilt.mk, dpatch.mk, simple-patchsys.mk が用意されていますし、dh には --with quilt というオプションが用意されています。ちなみに、各パッチの先頭にその意図を記述するタグを追加しておくことが推奨されています*28。

4. diff で表現されるので (画像などの) バイナリが置きにくい。

uuencode/uudecode で diff が取れる形式にしておくなどして対応します。

というわけで、これまでの source format でも問題点について対応はできます。しかしながら、こういった特殊な操作を行なうことなく問題点を解決するための新たなフォーマットとして、新たに 3.0 (quilt) と 3.0 (native) が制定されました*29。

16.1.2 3.0(quilt) と 3.0(native)

3.0 (quilt) 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, lzma, xz が使用可能。

`.orig-component1.tar.ext`

upstream が複数のソースから構成される場合に作成する。-componet はソースの名前に対応してメンテナが適宜名付ける。

`.debian.tar.ext`

debian ディレクトリの中身。ext は圧縮の拡張子であり、gz, bz2, lzma, xz が使用可能。1.0 の diff.gz から .tar.ext になり、全てが混在した単一のパッチではなくなった。

`.dsc`

パッケージに関する情報。

また 3.0 (native) は 1.0 での Debian 固有パッケージに対応しており、.orig.tar.ext と .dch ファイルからなります。

これらの変更により 1.0 での問題点は以下の様に解決されました。

1. アーカイブの圧縮形式として .gz しか使えない。

*26 .orig.tar.gz を展開するとディレクトリ内に upstream の bzip2 が配置される。ビルドする際に tar を展開してからビルドする

*27 こう書くと語弊があるかもしれませんが、quilt は Debian 固有のツールではありません。一方で dpatch は「patch maintenance system for Debian source packages」とあるように Debian 固有のツールです。

*28 <http://dep.debian.net/deps/dep3/> 面倒なので佐々木はサボりがちです。すいません。

*29 <http://wiki.debian.org/Projects/DebSrc3.0> ちなみに 2.0 はどういう状況知りません。だれかご存知ですか?

- .gzip, bzip2, lzma, xz を使用できるようになりました。
- 2. upstream のソースが複数のアーカイブから構成される場合が面倒。
 - .orig-component.tar.ext により複数のソースを使用できるようになりました。これら .orig-component.tar.ext はソースの展開時に component というサブディレクトリに展開されます。
- 3. パッケージメンテナの作成したパッチが全部単一のファイルに。
- 4. diff で表現されるので (画像などの) バイナリが置きにくい。
 - .diff.gz から .debian.tar.ext になったので全てが混然となった状態は解消されており、バイナリも含めることができます。また、debian/patches に存在するパッチはソースパッケージの展開時に自動的に適用されるようになりました (これについては後述)。

現在では、日々巨大化しつつある Debian アーカイブのサイズを抑制するために新規パッケージについては gzip よりも xz による圧縮が推奨されています。

16.1.3 3.0 への移行

さて source format 3.0 に以降するにはどうすれば良いでしょうか? 複雑なパッケージでないならば、

```
% mkdir debian/source
% echo '3.0 (quilt)' > debian/source/format
% dch 'Switch to dpkg-source 3.0 (quilt) format'
```

だけです。

これまでの diff.gz に upstream へのパッチが含まれている場合には、先ずパッチを (意味のある単位に) 分割し、quilt で管理できるようにしておきましょう。そうしない場合には分割されていないパッチは debian/patches/debian-changes-<version> という patch にまとめられてしまいます^{*30}。また、更新の度に debian-changes-<version> というパッチが生成されてしまいます。また quilt は -p1 のパッチしか扱えませんので rules 内で明示的に patch -p0 とか patch -p2 としている場合や、CDBS の simple-patchsys.mk を使用している場合には -p1 で適用できるようにパッチを変更しておきます。

最後に debian/rules でのパッチに関する rules を消しておくことが推奨されています。dpkg (>=1.15.5.4) では dpkg-source によってソースを展開する際に debian/patches 以下のパッチが自動的に適用されることは既に述べました。古い buildd、とくに lenny 向けの buildd ではソースを buildd chroot の外で展開することがあるため、dpkg-source の段階でソースにパッチが適用されていない場合には不具合が生じます。パッチを更新した場合には debuild を実行する前にパッチを全て適用しておきます。こうして debuild を実行すると、3.0 形式のソースパッケージが生成されます。

ちなみに lenny の dpkg は既に 3.0 に対応しています。これは lenny で squeeze のソースパッケージを使用する場合 (backports 等) に問題にならないようにするためです。また、現在は debian/source/format で source format のバージョンを指定していますが、将来的には全て新しい source format へ移行される予定です。

ではお手元のソースを 3.0 へ移行してみましょう

16.2 debian/rules の書き方

CDBS, dh 以前の debhelper コマンド群を使用した debian/rules は一番簡単な場合でも、例えば以下のようになります^{*31}。

^{*30} というわけです > lurdan

^{*31} lenny で GNU Hello のパッケージを作成した場合です。ページに収めるために先頭のコメントを削っています。

```

#!/usr/bin/make -f
# -*- makefile -*-
# Sample debian/rules that uses debhelper.
# - snip -

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

# These are used for cross-compiling and for saving the configure script
# from having to guess our platform (since we know it already)
DEB_HOST_GNU_TYPE ?= $(shell dpkg-architecture -qDEB_HOST_GNU_TYPE)
DEB_BUILD_GNU_TYPE ?= $(shell dpkg-architecture -qDEB_BUILD_GNU_TYPE)
ifneq ($(DEB_HOST_GNU_TYPE),$(DEB_BUILD_GNU_TYPE))
CROSS= --build $(DEB_BUILD_GNU_TYPE) --host $(DEB_HOST_GNU_TYPE)
else
CROSS= --build $(DEB_BUILD_GNU_TYPE)
endif
config.status: configure
dh_testdir
# Add here commands to configure the package.
ifneq "$(wildcard /usr/share/misc/config.sub)" ""
cp -f /usr/share/misc/config.sub config.sub
endif
ifneq "$$(wildcard /usr/share/misc/config.guess)" ""
cp -f /usr/share/misc/config.guess config.guess
endif
./configure $(CROSS) --prefix=/usr --mandir=$${prefix}/share/man --info
dir=$${prefix}/share/info CFLAGS='$(CFLAGS)' LDFLAGS="-Wl,-z,defs"

build: build-stamp

build-stamp: config.status
dh_testdir

# Add here commands to compile the package.
$(MAKE)
#docbook-to-man debian/hello.sgml > hello.1

touch $@

clean:
dh_testdir
dh_testroot
rm -f build-stamp

# Add here commands to clean up after the build process.
[ ! -f Makefile ] || $(MAKE) distclean
rm -f config.sub config.guess

dh_clean

install: build
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs

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

# Build architecture-independent files here.
binary-indep: build install
# We have nothing to do by default.

# Build architecture-dependent files here.
binary-arch: build install
dh_testdir
dh_testroot
dh_installchangelogs ChangeLog
dh_installdocs
dh_installexamples
#
dh_install
#
dh_installmenu
#
dh_installdebconf
#
dh_installogrotate
#
dh_installemacsen
#
dh_installpam
#
dh_installemime
#
dh_python
#
dh_installinit
#
dh_installcron
#
dh_installinfo
dh_installman
dh_link
dh_strip
dh_compress
dh_fixperms
#
dh_perl
#
dh_makeshlibs
dh_installdeb
dh_shlibdeps
dh_gencontrol
dh_md5sums
dh_builddeb

binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install

```

configure、make、make install 以外にも随所に dh_ から始まるコマンドが呼び出されています。この dh_ から始まるコマンドが debhelper コマンド群です。幾つかの例外を除いて dh_* は適切なタイミングで呼び出すだけです。この呼び出しを隠蔽するだけで debian/rules は随分すっきりします。CDBS, dh は異なる手法で dh_* の呼び出しの隠蔽とルールセットの追加を行なっています。

16.2.1 CDBS

CDBS については以前お話ししました (2008 年 6 月, 第 18 回関西 Debian 勉強会^{*32})。debian/rules は Makefile ですので

- Makefile の target を細分化。
- ソフトウェアやパッケージの状況に応じて、呼び出す target を class としてまとめる。

としています。

CDBS を利用した場合のビルドターゲットの流れについては例えば杉浦さんの Web に良い絵が公開されています (これは buildcore.mk + debhelper.mk を用いた場合です)。

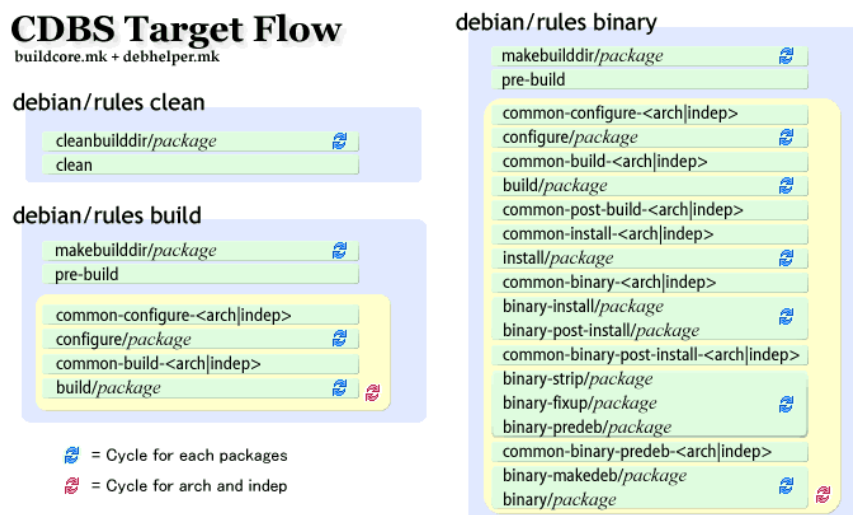


図 2 CDBS Target flow: <http://sugi.nemui.org/doc/cdb/cdb-targets.png>

CDBS を使った場合、先程の GNU Hello のルールは

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

となります。以前書きましたが、いきなりコレだと今度は何をしているのかわからなくなりますね (笑)。

最初の include が dh_* の呼び出しを設定する class, 二つ目の include が autotools を使用するソースからパッケージを作成する場合の class です。CDBS の詳細については本家のドキュメント^{*33} が参考になるでしょう。パッケージに付属のドキュメントではソースには実装されていて使えるのにドキュメント化されていない便利機能が多いです。こういう場合は「ソース見れば良いじゃん」というのはまあ正論ではあります。実際、class のカスタマイズに使用する変数はソース冒頭に定義されていますし...

^{*32} <http://tokyodebian.alioth.debian.org/pdf/debianmeetingresume200810-kansai.pdf>

^{*33} CDBS Documentation: <http://cdb-doc.duckcorp.org/en/cdb-doc.xhtml>

何も設定しない場合には `dh_auto_configure` がパッケージのビルド時に使用するツールを判定します。この判定結果に基づき `auto_build`, `auto_test`, `auto_install` が実行されます。自動判定結果は基本的に良くある GNU な設定になっており、例えば `configure` 時の `-prefix=/usr` とか `install` 時の `DESTDIR` などは GNU のパッケージ用に設定されています (なので GNU Hello は設定が不要です)。

注意して欲しいのは `configure` や `configure.ac` をパースしているわけではないので `dh_auto_*` は変数を自動設定をしているわけではない、という事です。 `configure` に対して Debian パッケージ用のフラグを適用したり、環境変数を追加する場合には、適宜変数を設定してやったりする必要があります。また、ソフトウェアによってはビルド時に `configure`, `make`, `make install` 以外の操作をする必要があったりします。このための機能が `dh` の `override` と `before`, `after` です。

16.2.3 実践! the magic debhelper rules

例えば GNU Hello の `configure` オプションには `--with-gnu-ld` や `--disable-nls` といったオプションがあります。これらを `configure` 時に指定したい、と思ったたらどうしたら良いでしょう? この場合には `dh_auto_configure` の呼び出し時にオプションを与えるようにします。例えば `lenny` の場合には `--before`, `--after` を使用します。

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

build: build-stamp
build-stamp:
    dh build --before configure
    dh_auto_configure -- --with-gnu-ld --disable-nls
    dh build --after configure
    touch build-stamp
```

`squeeze/sid` の `dh` には `override` という機能が追加されており、記述がさらに簡単になります^{*34}。

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

override_dh_auto_configure:
    dh_auto_configure -- --with-gnu-ld --disable-nls
```

`squeeze/sid` 版の `override` の方が直感的にわかりやすいですね。また、`squeeze/sid` の `dh` コマンドは `-no-act` の際に `override` を表示してくれます。

```
uwabami@vmguest-sid:~/Packages/hello-2.6$ dh binary --no-act
dh_testdir
debian/rules override_dh_auto_configure
dh_auto_build
dh_auto_test
...
```

`override` の詳細までは表示されませんが `override` されていること、及び呼び出しの順番が参照できます^{*35}。

以下では `squeeze/sid` の `dh` コマンドについて解説します。残念ながら `lenny` の `debhelper` は若干古いので、`squeeze/sid` の `debhelper` を導入しましょう。 `dh` コマンドの使い方については `Debconf9` での Joey Hess 御大による `debhelper` のプレゼン資料^{*36} が今の所一番まとまっている気がします。御大のプレゼン資料に掲載されている `dh` コマンドを利用した `rules` の例を以下に示します。

1. 自動判定結果とは別のビルドツールを使用

```
#!/usr/bin/make -f
%:
    dh $@ --buildsystem perl_build
```

^{*34} `override` は `debhelper` ($\geq 7.0.50$) 以降の機能です。

^{*35} 西山さんの `DEB_UPDATE.RCD_PARAMS` の場合は `dh_installinit` を `override` すれば良いと思います。

^{*36} <http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf>

2. upstream のソースファイルがサブディレクトリ src にありビルドを make -C で別ディレクトリ (ここでは build) で実行

```
#!/usr/bin/make -f
%:
    dh $@ --sourcedirectory=src \
          --builddirectory=build
```

3. パッチ当てに quilt を使用して、configure にフラッグを指定、changelog として changelog.html を収録

```
#!/usr/bin/make -f
%:
    dh $@ --with quilt \
          --builddirectory obj \
          --sourcedirectory source

override_dh_auto_configure:
    autoconf
    dh_auto_configure -- --with-sdl CC=/usr/bin/gcc-4.1

override_dh_installchangelogs:
    dh_installchangelogs changelog.html
```

などなど。

16.3 debhelper vs CDBS vs dh

Joey Hess 御大のプレゼン資料では

- debhelper で設定可能な項目は 138
- CDBS で設定可能な項目は 138 + 153
- dh で設定可能な項目は 138 + 12

となっています。

御大は「私と比較するのは unfair だ」とおっしゃってますが、rules の可読性/可変性は dh の方が圧倒的に優れていると思います。特に、dh の一番嬉しい所は-no-act による work flow が把握しやすいことでしょう。また、Build-Depends が減る、というのは一つの利点かもしれません。

一方、CDBS は work flow は把握しにくいですが対応しているビルドツールが多い事があげられます。CDBS 本体には Make, SCons(Make の代替品), Perl, Python, Cmake, Ant, Qmake, Gnome, KDE の class がありますし、本体には収録されていませんが pkg-kde-tools, pkg-ruby-tools が提供されています。これらの class を使用することで rules は共通化され非常に簡潔に記述できます。dh の場合、現時点对応しているビルドツールは

```
uwabami@vmguest-sid$ dh_auto_build --list
autoconf          GNU Autoconf (configure)
perl_makemaker    Perl ExtUtils::MakeMaker (Makefile.PL)
makefile          simple Makefile
python_distutils  Python Distutils (setup.py)
perl_build        Perl Module::Build (Build.PL)
cmake             CMake (CMakeLists.txt)
ant               Ant (build.xml)
qmake             qmake (*.pro)
```

といった所です。

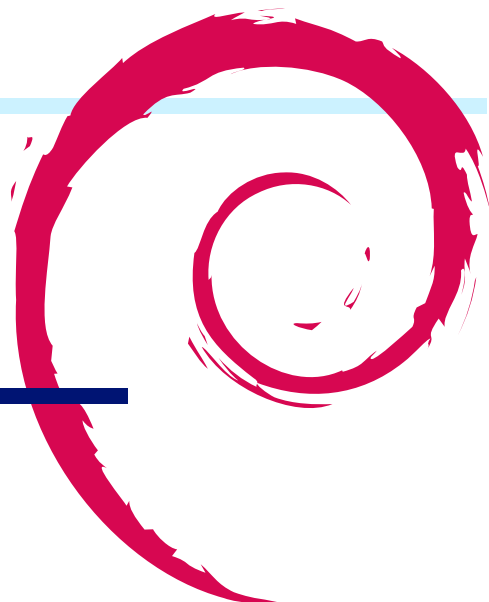
ではお手元のソースを CDBS or dh へ移行してみましょう。

既に CDBS を使用している場合には dh で書きかえて見て下さい。

最後に妄想など。現在手元で dh_ruby 関連を作成中です。setup.rb, extconf.rb ができれば多分 CDBS とは別の選択肢になるだろう、と思っています (あとは rake 用の build 拡張ができれば dh_make_gems の完成です)。これについては... 時間と紙面の都合で今日は載せられません。そのうち勉強会でお話しします。というわけで、「深追い」はタイトルに偽りアリ、ですね。すみません...Orz

17 Emdebian について

たなかとしひさ



17.1 前書き

この資料は、筆者が Emdebian を使う上で勉強した事を記載しています。まだまだ、完全に理解したわけではなく、部分部分でしか Emdebian を使えていませんが、中間報告と言う事でご容赦ください。

17.2 Emdebian って?

Emdebian (Embedded Debian) は、Debian GNU/Linux を元に、組み込み機器用途に最適化していくプロジェクトです。

Debian GNU/Linux は、まず Debian 自身がマルチアーキテクチャ(勉強会課題 1)に対応しています。また、どのベンダーからも独立しており、Debian 社会契約、および膨大な利用可能ソフトウェアは様々な選択肢を可能にしますが、デスクトップ環境(大きなハードディスクとメモリ)に向けられています。

‘Embedded Debian’は、Debian のメリットを活かしつつ、組み込み機器の様な小さいシステム向けに Debian を軽量化にするものです。

(上記は、<http://www.emdebian.org/> から意識したものです)

17.2.1 勉強会課題 1: Debian が動作する CPU、ターゲット機器

<http://www.jp.debian.org/ports/> から、Debian の移植版に関する情報が得られます。

- Intel x86 / IA-32 (i386) - 1 番身近で使われていますね。
- (Motorola 68k (m68k)) - Etch 以降のリリースには含まれていません。
- Sun SPARC (sparc)
- Alpha (alpha)
- Motorola/IBM PowerPC (powerpc)
- ARM (arm および armel) - 今回取り上げる CPU です。
- MIPS CPUs (mips と mipsel)
- HP PA-RISC (hppa)
- IA-64 (ia64)
- S/390 (s390)
- AMD64 (amd64)

また、移植中のターゲットとして、以下のものがあります。

- ARM のハードウェア float 対応 (armhf)

- Atmel 社 AVR32 向け (avr32)
- powerpc + SPE 命令 向け (powerpcspe)
- Renesas Electronics 社 sh4 向け
- Sparc 64bit 向け (sparc64)

Debian は、カーネルに Linux カーネルを使いますが、Debian GNU/kFreeBSD の様に、カーネルに FreeBSD のカーネルを使うものもあります。

17.2.2 勉強会課題 2: 皆さん、上記の内、使った事のあるアーキテクチャを教えてください。

なお、Emdebian は、下記のアーキテクチャが利用可能です。

i386, amd64, powerpc, armel, mips, mipsel, sh4

17.3 Emdebian は何を作っているか (何を作ろうとしているか)。

Smaller packages

Emdebian Grip - binary-compatible with Debian (今回お話しするものはこれです)

これは、Debian からインストールできる (要するに debootstrap でインストールできる) ものです。

Emdebian Crush - cross-built, customised Emdebian installations without perl

Web ページによると、Busybox をベースにした root filesystem との事です。

Busybox であるため、Debian そのものと構成が変わっている事が考えられますが、Emdebian Grip と比べると、もっと容量は小さいと考えています。

Cross building tools

その名の通り、クロス開発ツールです。

「クロス開発」とは、例えばパソコン (i386) 上で、ARM のバイナリを生成する様な、ホスト (コンパイル) 環境とターゲット (実行) 環境の CPU や OS が異なる場合の開発を言います。

組み込み機器は、その殆どがクロス開発で作ります。

他方、ホスト環境とターゲット環境が同じ、単純に言えば、i386 上で、i386 上で動くソフトウェアを開発する場合は、「セルフ開発」と言います。

Emdebian は、Debian 正規のものと同期しながら、クロス開発環境に焦点を当てています。i386 と amd64 アーキテクチャ上で、arm, ia64, m68k, mips, mipsel, powerpc, sparc のビルドが可能です。

Root filesystem generation is based on multistrap package.

multistrap は、Emdebian で root filesystem を作るうえでのメインとなるツールです。

(ごめんなさい、multistrap はまだ筆者が十分に勉強できていません...)

Emdebian 自身、まだ作業中のものが多く、協力者を募集しています。

<http://www.emdebian.org/emdebian/helpout.php> このページに、Emdebian の ToDo(バグリスト) があります。

17.4 なぜ、「組み込み Linux」なのか?(なぜ「組み込み Debian」なのか?)

理由は人それぞれですが、筆者自身が強く感じるのは、組み込み Linux は、プログラムの動作確認が容易になり、ソフトウェアの品質を確保しやすくなるという事です。

例えば、日本の組み込み機器で使う OS には ITRON を使う事が多いです。海外だと VxWorks を使う事が多いです。ITRON を使った開発の場合、ITRON のシステムコールは、PC ではシミュレータ (あるいはエミュレータ) を使わない限り、動きを含めた動作確認は出来ません。

そのため、JTAG 等のデバッガを使って、実機にプログラムを焼きこんでデバッグする事が殆どです。

組み込み Linux の場合、i386 版 Linux で、ある程度動きも含めたデバッグも可能になります。

ARM 版 Linux 向けのプログラムを作るとして、一々ARM 版プログラムをビルドして ARM CPU なターゲットボードに転送するよりも、i386 版 Linux で粗方デバッグしておき、ターゲット環境では実際のターゲットならではのデバッグに注力すれば、デバッグ時間を削減できます。デバッグ時間を削減できるという事は、ソフトウェアテスト等の時間を増やす事が出来ると言う事であり、ソフトウェアの品質向上に繋げることが出来ます。

確かに、ターゲット機器上で全て動作確認をすべきですが、ターゲット機器上でトレースデバッグをするよりも、ホスト環境上で基本的なデバッグができるのは魅力で効果的です。

Linux は、無料で使用できますが、筆者は、有料/無料とは別に、ソフトウェアの品質を確保しやすいという点で、組み込み Linux は他の OS よりも優位性があると考えています。

さらに、組み込み Debian は、PC 等で得た Debian の知識を、組み込み機器にも活かす事が出来ます。ソフトウェアの不具合のいくつかは、不慣れな (未知な) 環境下であった事に起因する不具合があります。普段から使い慣れている OS (Debian) が使える事も、ソフトウェアの品質を確保する上で重要なのです。

最後に、Debian はベンダー独立、別の言い方をすると...倒産する事がないです :-)。

「Linux 企業」は、自主独立で歩き続けているベンダーもあれば、吸収合併、あるいは部門売却などで看板が変わり、契約が変わる場合があります。

「Linux 企業」と契約する側にしてみれば、Linux 企業と契約したものの、ある日部門売却等で契約先が変わり、再度新規契約からやり直し...となるのは手間です。もしそこで費用面から話をしなければならずとすると、Linux を使うこと自身に消極的になります。

Debian は、その様な事はありませんので、安心して使い続けることが出来ます。

17.5 Emdebian Grip を試してみる。

Emdebian Grip を、MINI2440 にインストールしてみました。

厳密に言うと... rootfs は Emdebian Grip ですが、Linux カーネルは Emdebian そのものではありません。すみません。これも引き続きの勉強課題とさせていただきます。

写真は、MINI2440 に Emdebian Grip をインストールして、iceweasel で OpenStreetMap のページを参照したものです。ARM 上で Web ブラウザが普通に使えます (但し遅いです)。rootfs は SD カードを使っています。



MINI2440 の詳しい情報は、<http://www.friendlyarm.net/products/mini2440> を参照してください。

17.5.1 必要なもの

まず、MINI2440 の NAND フラッシュROM のバックアップを取るには、残念ながら MS-Windows 上で動くソフトウェア (DNW) が必要です。筆者は、NAND フラッシュROM のバックアップと、ブートローダ (U-Boot) 書き込みに (不本意ですが)MS-Windows を使いました。(後すみません、TeraTerm も使いました)

他に必要なものを以下に記します。

- MINI2440 本体
- i386(amd64) な Debian
- ネットワーク
- シリアルケーブル (クロスケーブル)
- SD カード (1GByte 2GByte)(Debian PC からアクセスするので、カードアダプタも必要)

17.5.2 MINI2440 の Emdebian 化

MINI2440 NAND フラッシュのバックアップ (必要なら)

MINI2440 は、購入した状態では NAND フラッシュROM に Qt/Embedded がインストールされていますので、もし必要ならばバックアップしてください。

Emdebian Grip rootfs の取得

<http://code.google.com/p/mini2440/wiki/Emdebian> を参考に、Emdebian Grip の rootfs を取得します。ここでは debootstrap を使いますので、適宜 apt-get 等でインストールしておいてください。

Emdebian Grip rootfs は、上記 URL に記載している手順で進められます...が、この URL では、Emdebian Grip 向け Linux カーネルビルドに関する事が無いので、Emdebian Grip 向けの Linux カーネルをビルドする必要があります。

MINI2440(Emdebian Grip) 向け Linux カーネルのビルド

MINI2440(Emdebian Grip) 向け Linux カーネルをビルドするには、最低限下記のものがが必要です。

1. Toolchain(gcc)
2. (MINI2440 に対応した)Linux カーネルソース

できれば、それこそ Emdebian で一撃に...と思ったのですが、筆者は <http://code.google.com/p/mini2440/downloads/list> から得られる mini2440-bootstrap-v2.sh というシェルスクリプトを改造して使いました。

このシェルスクリプト (mini2440-bootstrap-v2.sh) は、開発環境 (CodeSourcery) や MINI2440 向け Linux カーネルを自動的にダウンロードしてビルドします。

なお、mini2440-bootstrap-v2.sh は、Debian 上でも使えますが、U-Boot のソースコードはダウンロードしますがビルドはしないので、下記の様に修正して使います。

```
tosihisa@lavie:~/Downloads$ diff -ca mini2440-bootstrap-v2.sh mini2440-bootstrap-v2.sh.change
*** mini2440-bootstrap-v2.sh      2010-08-17 17:19:45.000000000 +0900
--- mini2440-bootstrap-v2.sh.change    2010-08-17 17:19:13.000000000 +0900
*****
*** 72,78 ****
# compile bits
cd ${DEST}/uboot/mini2440
make mini2440_config
! # make clean all

cd ${DEST}/kernel/mini2440

--- 72,78 ----
# compile bits
cd ${DEST}/uboot/mini2440
make mini2440_config
! make clean all

cd ${DEST}/kernel/mini2440
tosihisa@lavie:~/Downloads$
```

U-Boot を MINI2440 にインストールする。

MINI2440 は、Supervivi というブートローダが NOR フラッシュ側にありますが、NAND フラッシュ側に U-Boot をインストールします。

U-Boot(the Universal Boot Loader) は、Linux カーネルだけでなく、ELF 形式であればロードできるブートロー

ダです。また、ブートローダとしての機能だけではなく、メモリ操作が可能なので、デバッグツールとしても利用できます。

U-Boot は、<http://www.friendlyarm.net/downloads> から入手できるビルド済みのバイナリ (u-boot.20100701.zip) を使いました。

Emdebian Grip を起動...

rootfs を作り、Linux カーネルをビルドできたら、それらを SD カードにコピーします。U-Boot を MINI2440 に焼きこめたら、Linux を起動します。

debootstrap 直後の SD カードで起動した場合、完全にはインストールが完了していませんので、U-Boot のブートパラメータに 'init=/bin/sh' を与えて、シェルを起動させるようにします。

その後、(MINI2440 で起動した Emdebian で) シェルが起動したら、ネットワークが DHCP で割り振られた状態で、下記のコマンドを実行する事で Emdebian のインストールが継続されます。

```
sh-3.2# /debootstrap/debootstrap --second-stage
```

これが終われば、Emdebian Grip が楽しめます。apt-get でソフトを入れてみてください。

17.6 出来てないことだらけ (今後の勉強課題)

- Toolchain も Emdebian のものを使う。
- Linux カーネルも Emdebian 由来のものを使ってみたい。
- Emdebian Crush を試す。
- U-Boot はソースからビルドして使う。
- この資料を充実させる (年末には...)

18 Debian GNU/kFreeBSD で暮らせる環境を構築してみる。

杉本 典充



18.1 Debian GNU/kFreeBSD について

18.1.1 Debian GNU/kFreeBSD とは

「Debian GNU/kFreeBSD」とはカーネルに FreeBSD カーネル、ユーザランドに Debian のポリシーやパッケージシステムを取り入れた OS です。Debian Project は Linux カーネル以外のカーネルを用いた OS を作成する取り組みも行っており^{*37}、Debian GNU/kFreeBSD は Squeeze でのリリースを目指して開発が進んでいます。

Debian GNU/kFreeBSD は i386 版と amd64 版のアーキテクチャが利用できます。

Debian GNU/kFreeBSD については以下に情報が公開されています。

- Debian Wiki : http://wiki.debian.org/Debian_GNU/kFreeBSD
- Debian Wiki(FAQ) : http://wiki.debian.org/Debian_GNU/kFreeBSD_FAQ
- Mailing List : <http://lists.debian.org/debian-bsd/>
- IRC : #debian-kbsd at irc.debian.org

18.1.2 Debian GNU/kFreeBSD と Debian GNU/Linux の違い

Debian GNU/kFreeBSD から見た Debian GNU/Linux との違いについて以下に一例を上げます。

- デバイスドライバは FreeBSD の流儀に従う。
 - サウンドデバイスは OSS を利用する。
 - ネットワークデバイス名が「eth0」等の固定名ではなくネットワークドライバによって変わる。
 - ディスクデバイス名が「/dev/ad4s1」のような形式になる。
 - mount コマンドのオプションが若干異なる。(USB メモリで利用される FAT32 は vfat ではなく msdosfs を用いる。)
- ファイルシステムは (FreeBSD で実装している)UFS、ZFS、ext2^{*38}が使える。
- 仮想化は FreeBSD Jail、VirtualBox、qemu を使う。(KVM は Linux 特有の機能のため使えない。他の OS を動かしたい場合は Debian GNU/kFreeBSD は不利。)

その他の apt によるパッケージシステムやディレクトリ構造は Debian GNU/kFreeBSD も Debian GNU/Linux も同じため、Debian GNU/Linux の利用者であればすぐに慣れます。

^{*37} <http://www.debian.org/ports/>

^{*38} Debian GNU/kFreeBSD で ext3 は読み込みのみサポート。 http://wiki.debian.org/Debian_GNU/kFreeBSD_FAQ

18.2 Debian GNU/kFreeBSD のインストール

Debian GNU/kFreeBSD のインストーラは daily ビルドのイメージがありますので、以下からダウンロードします。

- i386 : <http://d-i.debian.org/daily-images/kfreebsd-i386/>
- amd64 : <http://d-i.debian.org/daily-images/kfreebsd-amd64/>

今回使用したインストーラは i386 版の 2010 年 6 月 19 日のビルドを利用しました。^{*39}

インストール CD を作成し、PC に CD をセットして起動するとインストーラが起動します。



図 3 Debian GNU/kFreeBSD の Debian インストーラ

インストール中の設定は以下を指定してインストールしました。

- locale は「C」。(現時点のインストーラでは「C」と「English」のみの locale しか指定できないため。)
- タイムゾーンは Asia/Japan。
- インストールは「Standard system utilities」(=Base System) のみ。

18.3 初回起動

18.3.1 前準備

X Window System をインストールしていないため、再起動後はコンソール環境で起動します。起動時に DHCP クライアントが起動するため、有線 LAN 環境であれば IP アドレスを自動で取得してネットワークにつながります。(以前は手動で dhclient を起動しないとつながらなかった。)

カーネル起動後にインストーラで作成したユーザでログインし、最低限必要な以下のパッケージをインストールし設定します。

```
$ su
# apt-get update
# apt-get install sudo vim
# visudo
```

^{*39} インストーラは当たり外れがあるようで、ディスクのパーティションを作成する処理が正常に動作せずインストール処理を進めることができないビルドが多かったです。そのため、パーティション作成に失敗するビルドは諦めて、ビルド時期が少し前のビルドを使用して再チャレンジすることをおすすめします。

18.3.2 カーネルの更新

カーネルの起動メッセージを眺めていると CPU が 1 つしか認識していないように見えます。現在起動中のカーネルと CPU の認識数を確認するとやはり 1 つしか認識していません。

```
$ uname -a
GNU/kFreeBSD deb-NorTP60 7.3-1-686 #0
Tue Jul 20 02:12:21 CEST 2010 i686 i386
Genuine Intel(R) CPU          T2400 @ 1.83GHz GNU/kFreeBSD
```

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 7
model name    : Genuine Intel(R) CPU          T2400 @ 1.83GHz
stepping      : 8
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
               cmov pat b19 b21 mmxext mmx fxsr xmm b26 b27 b28 b29 3dnow
cpu MHz       : 1828.76
bogomips     : 1828.76
```

現在利用できるカーネルを検索すると以下の候補が出てきます。

```
$ apt-cache search kfreebsd-image-*
kfreebsd-headers-7.3-1-486 - header files for kernel of FreeBSD 7.3
kfreebsd-headers-7.3-1-686-smp - header files for kernel of FreeBSD 7.3
kfreebsd-headers-7.3-1-686 - header files for kernel of FreeBSD 7.3
kfreebsd-image-7-486 - kernel of FreeBSD 7 image
kfreebsd-image-7-686-smp - kernel of FreeBSD 7 image
kfreebsd-image-7-686 - kernel of FreeBSD 7 image
kfreebsd-image-7.3-1-486 - kernel of FreeBSD 7.3 image
kfreebsd-image-7.3-1-686-smp - kernel of FreeBSD 7.3 image
kfreebsd-image-7.3-1-686 - kernel of FreeBSD 7.3 image
kfreebsd-headers-8.0-1-486 - header files for kernel of FreeBSD 8.0
kfreebsd-headers-8.0-1-686-smp - header files for kernel of FreeBSD 8.0
kfreebsd-headers-8.0-1-686 - header files for kernel of FreeBSD 8.0
kfreebsd-image-8-486 - kernel of FreeBSD 8 image
kfreebsd-image-8-686-smp - kernel of FreeBSD 8 image
kfreebsd-image-8-686 - kernel of FreeBSD 8 image
kfreebsd-image-8.0-1-486 - kernel of FreeBSD 8.0 image
kfreebsd-image-8.0-1-686-smp - kernel of FreeBSD 8.0 image
kfreebsd-image-8.0-1-686 - kernel of FreeBSD 8.0
```

シングルプロセッサ用とマルチプロセッサ用のカーネルは別々のようですのでマルチプロセッサ用カーネルをインストールし、再起動します。^{*40}

カーネルのインストール処理で grub2 もアップデートされます。^{*41}

```
$ sudo apt-get install kfreebsd-image-7.3-1-686-smp
$ sudo reboot
```

再起動し、カーネルと CPU の認識数を確認します。

```
$ uname -a
GNU/kFreeBSD deb-NorTP60 7.3-1-686-smp #0
Tue Jul 20 02:43:20 CEST 2010 i686 i386
Genuine Intel(R) CPU          T2400 @ 1.83GHz GNU/kFreeBSD
```

^{*40} HyperThreading のセキュリティ上の脆弱性に対応するため FreeBSD 本家がリリースするカーネルは HyperThreading がデフォルトで OFF になっています。Debian GNU/kFreeBSD でデフォルトが OFF であるかは対応 CPU を持っていないため確認できていません。

^{*41} kfreebsd-image-8.0-1-686-smp をインストールしてみましたが、インストール後に /ヘパーティションをマウントする処理に失敗し起動できませんでした。FreeBSD 8.0 Release Note に「”dangerously dedicated” mode for the UFS file system is no longer supported. Important: Such disks will need to be reformatted to work with this release.」という記述があるため、7.3 から 8.0 へのカーネルアップグレードは少し難しいようです。

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 7
model name    : Genuine Intel(R) CPU          T2400 @ 1.83GHz
stepping      : 8
processor     : 1
vendor_id     : GenuineIntel
cpu family    : 6
model         : 7
model name    : Genuine Intel(R) CPU          T2400 @ 1.83GHz
stepping      : 8
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
               cmov pat b19 b21 mmxext mmx fxsr xmm b26 b27 b28 b29 3dnow
cpu MHz       : 1828.76
bogomips     : 1828.76
```

18.4 Xorg のインストール

X Window System 環境が日常を過ごすため、xorg をインストールします。しかしパッケージのインストール処理で以下のエラーが発生し、途中でインストールが停止しました。

```
$ sudo apt-get install xorg
Setting up hal (0.5.14-3) ...
Reloading system message bus config...
Failed to open connection to "system" message bus:
Failed to connect to socket /var/run/dbus/system_bus_socket: Connection refused
invoke-rc.d: initscript dbus, action "force-reload" failed.
Starting Hardware abstraction layer: haldinvoke-rc.d: initscript hal,
action "start" failed.
dpkg: error processing hal (--configure):
 subprocess installed post-installation script returned error exit status 1
```

18.4.1 対応

上記は hal のインストールの後処理で dbus の設定を再読み込みしようとしてエラーが発生したため、パッケージのインストールが停止しました。^{*42}

ps コマンドで dbus プロセスがいるか確認すると存在しません。バグ修正はまだ行われていないため、とりあえずの措置として「\$ sudo /etc/init.d/dbus start」を実行して dbus を起動し再度「\$ sudo apt-get install xorg」を試みると次は以下のエラーに変わりました。

```
$ sudo /etc/init.d/dbus start
$ sudo apt-get install xorg
Setting up xserver-xorg (1:7.5+6) ...
invoke-rc.d: initscript hal, action "restart" failed.
dpkg: error processing xserver-xorg (--configure):
 subprocess installed post-installation script returned error exit status 1
```

今度は xserver-xorg のインストールの後処理で hal と同様のエラーが発生したため、再度 hal と同様に以下コマンドを実行し、xorg のインストールは完了しました。

```
$ sudo /etc/init.d/dbus start
$ sudo apt-get install xorg
```

18.4.2 X Window System 起動及び設定

この状態で startx コマンドを実行すると X Window System の起動に成功しました。しかし「/etc/X11/xorg.conf」の設定を確認しようとするファイル自体が存在しないため xorg.conf を手動で作成します。

```
$ sudo X -config
$ sudo cp xorg.conf.new /etc/X11/xorg.conf
```

再度 startx を実行し、作成した xorg.conf を用いて X Window System の起動が確認できました。「/var/log/Xorg.0.log」を確認し'(EE) 'の部分はないため、動作は正常です。

^{*42} <http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=469528>

18.4.3 gdm のインストール (断念)

ログイン画面も GUI で行いたいので gdm をインストールします。

```
$ sudo apt-get install gdm
```

gdm をインストールし reboot せずに gdm コマンドを実行するとマウス、キーボードは問題なく動作します。reboot 後、自動で gdm が起動するのですが gdm のログイン画面でマウスは動きますがキーボードが動作しません。(USB キーボードも動作しなかった。)

不本意ながら gdm によるログインは諦め startx による X Window System の起動に切り替えることにしました。しかしこの状態ではカーネルの起動直後に gdm が勝手に起動しキーボード入力ができなくなるため仮想コンソールに切り替えることもできず、gdm を purge できません。

そのため、一度電源スイッチを長押しして PC を停止、再起動して gdm が起動する前に「Ctrl + Alt + F1」を連打してなんとか仮想コンソールに入り以下を実行して gdm を purge しました。

```
$ sudo apt-get purge gdm
```

18.5 デスクトップ環境の構築

統合デスクトップ環境である「Xfce4」をインストールし、startx で Xfce4 を起動するように .xinitrc を記述します。 .xinitrc は実行権が必要なため、実行権を付与します。

```
$ sudo apt-get xfce4 xfce4-goodies
$ vim ~/.xinitrc
exec xfce4-session

$ chmod 744 ~/.xinitrc
$ startx
```

18.6 日本語環境

18.6.1 日本語の表示

日本語フォントをインストールします。

```
$ sudo apt-get otf-ipafont otf-ipaexfont
```

日本語環境の「ja_JP.UTF-8」ロケールはインストールされていなかったため追加インストールします。

```
$ sudo apt-get locales-all
```

startx コマンドで X Window System を起動するため locale の設定を .xinitrc に追記します。Xfce4 を終了して再度 startx コマンドを実行すると Xfce4 が日本語環境で起動します。

```
$ vim ~/.xinitrc

export LANGUAGE='ja_JP.UTF-8'
export LC_ALL='ja_JP.UTF-8'
export LANG='ja_JP.UTF-8'
exec xfce4-session

$ startx
```

18.6.2 日本語入力

日本語入力環境として uim を使用したいため、uim のインストールと設定をします。

```
$ sudo apt-get install uim uim-anthy
$ vim ~/.xinitrc

export LANGUAGE='ja_JP.UTF-8'
export LC_ALL='ja_JP.UTF-8'
export LANG='ja_JP.UTF-8'

export XMODIFIERS='@im=uim'
export GTK_IM_MODULE='uim'
export QT_IM_MODULE='uim'

exec xfce4-session

$ startx
```

18.7 開発環境のインストール

Debian GNU/kFreeBSD のデバッグに必要なコンパイラ、パッケージの作成ツールをインストールします。

```
$ sudo apt-get update
$ sudo apt-get install gcc g++ gdb make
$ sudo apt-get install build-essential pbuilder debian-keyring
```

deb パッケージのビルドができるか tcsh をビルドして確認します。

```
$ apt-get source tcsh
$ sudo apt-get build-dep tcsh
$ cd tcsh-6.17.00
$ dch
$ debuild -i -us -uc -b
$ sudo dpkg -i tcsh_6.17.00-3.1_kfreebsd-i386.deb
```

18.8 Debian 勉強会資料のビルド環境

Debian 勉強会での原稿は tex を採用しているため、原稿ファイルのビルド環境を構築します。

tex のビルドには contrib、non-free のパッケージが必要なため、apt-line を修正します。

```
$ sudo vim /etc/apt/sources.list
# deb http://ftp.jp.debian.org/debian/ squeeze main

deb http://ftp.jp.debian.org/debian/ squeeze main contrib non-free
deb-src http://ftp.jp.debian.org/debian/ squeeze main contrib non-free

deb http://security.debian.org/ squeeze/updates main
deb-src http://security.debian.org/ squeeze/updates main
```

必要なパッケージをインストールします。

```
$ sudo apt-get install git-core
$ sudo apt-get install gs gs-esp gs-cjk-resource
$ sudo apt-get install ptex-bin xdvik-ja dvipsk-ja
$ sudo apt-get install okumura-clsfles vfdata-morisawa5
$ sudo apt-get install texlive-latex-extra
$ sudo apt-get install poppler-data
$ sudo apt-get install evince
```

資料をダウンロードし、ビルドします。

```
$ cd
$ git clone git://git.debian.org/git/tokyodebian/monthly-report.git/
$ cd monthly-report
$ make
```

18.9 その他の日常環境の構築

その他のよく使うソフトをインストールします。

```
$ sudo apt-get install emacs emacs23-el
$ sudo apt-get install sylpheed sylpheed-i18n
$ sudo apt-get install iceweasel iceweasel-l10n-ja
$ sudo apt-get install audacious audacity
$ sudo apt-get install gxine
$ sudo apt-get install jd
$ sudo apt-get install gftp
```

18.10 デバイスドライバ

audacious は音楽再生ソフトなのですが、実行しても音が鳴りません。サウンドドライバをロードしているか確認するため「kldstat」を実行します。^{*43}

```
$ kldstat
1  10 0xc0400000 890000 kfreebsd-7.3-1-686-smp.gz
2  1 0xc0d9c000 57fdc  acpi.ko
3  1 0xc5c7a000 67000  radeon.ko
4  1 0xc5ce1000 14000  drm.ko
```

サウンドドライバがロードされていないためロードします。^{*44}

```
$ sudo kldload snd_hda
$ kldstat
1  10 0xc0400000 890000 kfreebsd-7.3-1-686-smp.gz
2  1 0xc0d9c000 57fdc  acpi.ko
3  1 0xc5c7a000 67000  radeon.ko
4  1 0xc5ce1000 14000  drm.ko
5  1 0xc611f000 1a000  snd_hda.ko
6  1 0xc6139000 40000  sound.ko
```

音が鳴ることは確認できましたが、再起動するとサウンドドライバが読み込まれていない状態になります。そのため「/etc/modules」ファイルにカーネル起動時に自動で読み込むドライバを設定します。^{*45}

```
$ sudo vim /etc/modules
# /etc/modules: kernel modules to load at boot time.
#
# This file should contain the names of kernel modules that are
# to be loaded at boot time, one per line. Comments begin with
# a '#', and everything on the line after them is ignored.
snd_hda.ko
```

18.11 今後の課題

Debian GNU/kFreeBSD でセットアップ作業を行いました但し日常生活を送るに向けていくつか課題が残っています。

- Web ブラウザでの Flash の再生 (Adobe 公式の Flash バイナリに FreeBSD 版はまだない)
- ビデオ再生 (映像が乱れる。おそらく codec の問題)

今後に向けた取り組みとして以下を試していきたいです。

- Linux バイナリ互換機能
- 仮想化機能 (Jail、VirtualBox、qemu)
- ZFS

^{*43} FreeBSD カーネルで読み込み中のカーネルモジュールの一覧を出力するコマンドは kldstat ですが、Debian GNU/kFreeBSD では lsmod が kldstat のリンクになっているため lsmod でも一覧を出力できます。

^{*44} snd_hda は Intel945 チップセットで音を鳴らすために必要なサウンドドライバです。異なるサウンドチップを利用している場合は環境に合わせてロードしてください。

^{*45} Debian GNU/kFreeBSD に/sbin/modprobe コマンドはあるのですが/sbin/kldload のリンクになっているため、modprobe をただだけで再起動後も自動でモジュールを読み込むようにはなっていないようです。

18.12 環境構築を終えて

インストーラの整備も進んでおり、X Window System の導入後は Debian GNU/Linux となんら変わらない手順で環境構築ができました。

ただ問題が発生したときに FreeBSD カーネルと Debian の両方の知識が必要なため、原因を調べるのが大変で FreeBSD と Debian の両方の情報源に当たってなんとか解決しました。

バグ報告があると同じところで見つめている人と情報を共有できるのでできるだけバグ報告は上げましょう。

Debian GNU/kFreeBSD は品質も徐々に上がってきており、Squeeze で技術プレビュー扱いのリリースがされる予定です。^{*46} みなさんもぜひ Debian GNU/kFreeBSD をデバッグして Squeeze リリースまでに品質を高めましょう。

18.13 参考資料

- 東京エリア Debian 出張勉強会 発表スライド (岩松信洋) : <http://tokyodebian.alioth.debian.org/pdf/debianmeetingresume201006-iwamatsu-presentation.pdf>
- Debian wiki : http://wiki.debian.org/Debian_GNU/kFreeBSD_FAQ

^{*46} <http://www.debian.org/News/2010/20100806>

19 Debian Miniconf 計画検討

山本 浩之



毎月恒例になるらしい、プレストタイム。Debian Miniconf in Japan に向けてみんなでプレストします。

19.1 先月までに決まっていること

19.1.1 聴衆のメインターゲット

Debian だけに限らない開発者やユーザ。アップストリーム開発者や翻訳者も含む。ビジネス関係の人たちは二の次。

19.1.2 開催形態

日本で開催される FLOSS 関係の国際会議に相乗りして開催。勿論、基本的に英語で。

19.1.3 現在までに出ているネタ

- Embedded セッション
組込における Debian の取り組み。
- ライセンスセッション
ソフトウェア作成者とコンテンツ作成者とのフリーなライセンスでの交流。
- 事例集セッション
Debian を導入した企業や団体の事例を通じ、なぜ Debian なのかを語る。
- プログラム言語系セッション
特に Ruby や関数言語系における Debian の取り組み。
- VPS・クラウドセッション
Debian における VPS・クラウドに関するパッケージや開発状況。
- WEB アプリケーションセッション
Debian における WEB アプリケーションに関するパッケージや開発状況。
- Debian からの派生ディストリビューションセッション
数多くある Debian からの派生ディストリビューションとの、お互いの交流や要求。
- ハックラボ
圧倒的人気により、当確(?)。

『あんどきゅめんてっど でびあん』について

本書は、東京および関西周辺で毎月行なわれている『東京エリア Debian 勉強会』および『関西エリア Debian 勉強会』で使用された資料・小ネタ・必殺技などを一冊にまとめたものです。収録範囲は東京エリアは2010年6月勉強会(第65回)から2010年11月勉強会(第70回)、関西エリアは第36回から第41回まで。内容は無保証、つっこみなどがあれば勉強会にて。



あんどきゅめんてっど でびあん 2010年冬号

2010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)
