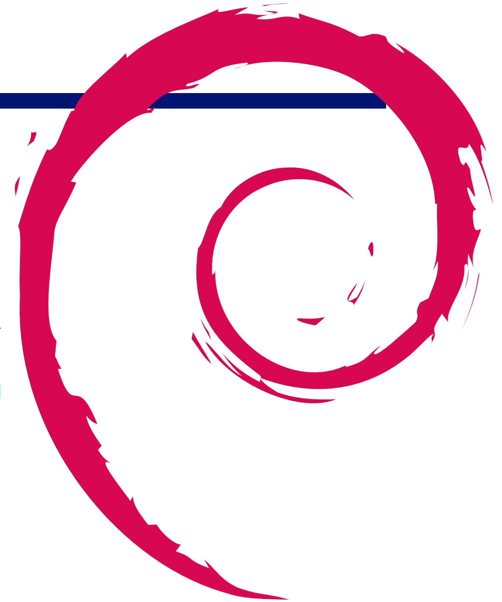
第70回 東京エリア Debian 勉強会資料

# 月刊 Deb専

日本唯一のDebian専門月刊誌

2010年11月20日

## 特集: 俺のファイルシステムは熱いぜ!



# 1 Introduction

上川 純一

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

Debian 勉強会の目的は下記です。

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

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

デビアン勉強会

# 目次


1 Introduction 1

2 事前課題 3
- 2.1 yamamoto 3
- 2.2 吉野 (yy_y_ja_jp) 3
- 2.3 キタハラ 3
- 2.4 高橋斉大 3
- 2.5 MATOHARA 3
- 2.6 やまだ 3
- 2.7 henrich 3
- 2.8 emasaka 4
- 2.9 本庄 4
- 2.10 吉田@小江戸 4
- 2.11 nozzy123nozzy 4
- 2.12 まえだこうへい 4
- 2.13 荒木靖宏 4

3 最近の Debian 関連のミーティング報告 5
- 3.1 東京エリア Debian 勉強会 69 回目報告 5

4 ext4 ファイルシステムを Debian で活用してみる 6
- 4.1 ext4 ファイルシステムを作ってみる 6

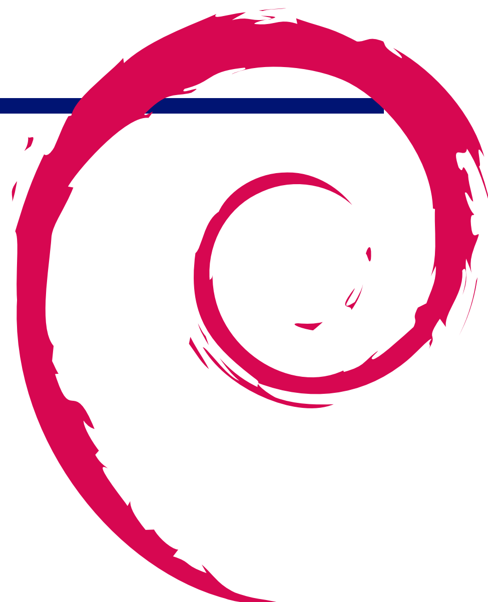
5 NILFS を Debian で活用してみる 7
- 5.1 はじめに 7
- 5.2 使ってみよう - nilfs の導入と特徴 7
- 5.3 チェックポイント (cp) とスナップショット (ss) 9
- 5.4 nilfs の構造 - ログ構造化ファイルシステム 9
- 5.5 nilfs の落とし穴 - ガーベージコレクション 10
- 5.6 nilfs_cleanerd - 標準ガーベージコレクタ 11
- 5.7 libnilfs - 手作りガーベージコレクタへ 13
- 5.8 他の選択肢との比較（１） - btrfs 14
- 5.9 他の選択肢との比較（２） - LVM 19
- 5.10 まとめ 20

6 Btrfs を Debian で活用してみる 22
- 6.1 Btrfs ってどんなファイルシステム? 22
- 6.2 Debian でインストールする方法 22
- 6.3 フォーマット・マウント・btrfsck 22
- 6.4 複数のディスクを使用してみる 23
- 6.5 バックアップ用のファイルシステムとして使ってみる 24
- 6.6 まとめ 24

7 分散ファイルシステム CEPH を Debian で活用してみる 25
- 7.1 CEPH ってどんなファイルシステム? 25
- 7.2 インストールした環境 25
- 7.3 インストール方法 25
- 7.4 設定 25
- 7.5 起動 27
- 7.6 テスト 27
- 7.7 対障害性について 28

8 Debian Miniconf 計画検討 30
- 8.1 本日のミッション 30
- 8.2 先月までに決まっていること 30

9 索引 31



# 2 事前課題

上川 純一

今回の事前課題は以下です:

- 日常的に活用しているファイルシステム設定について紹介する。(例、 LVM, ext3, jffs, ...)

この課題に対して提出いただいた内容は以下です。

## 2.1 yamamoto

ext2 一点勝負。理由は recover の呪文が使えるから。

確認せずによくファイルを消してしまうのよ。だから inode からデータブロックへの参照をクリアしてしまう fs には移行できない。

最近、テレビの録画ファイルの増加で、やっと lvm を使うようになったけど、フォーマットは相変わらず ext2 です。

## 2.2 吉野 (yy_y_ja_jp)

あまり面白設定ではないですが... 手元のラップトップでは /boot には ext2 その他には ext3, 最近使っているデスクトップでは /boot には ext3 その他には LVM 上に ext4 を使っています。

## 2.3 キタハラ

デフォルトの ext3 一択。設定もデフォルトのまま。(NTFS 上の VM イメージ上の ext3 もあるけど)

## 2.4 高橋斉大

ext3/4 を主に使ってます。 ext3 のデータサルベージについて日経 Linux2011 年 1 月号に執筆しました。

## 2.5 MATOHARA

以前 inode 枯渇に遭遇してから inode が動的に割り当てられる XFS を選択することが多いです. NILFS は少し試してみたのですが, mount 時に 5.2 節の Warning メッセージが出てまだ怖いなと思いました。

その他 NotePC では dm-crypt の上にファイルシステムを置いて暗号化したり、 eCryptfs で暗号化したりしています。書き込み時に CPU をかなり消費します…。

## 2.6 やまだ

過去、 ext2->reiserfs->jfs->xfs->reiserfs->ext3 と使ってきました。

雑感:

- reiserfs: 小ファイルや多数ファイルのアクセスがキビキビしてて良かった! しかしふとした浮気心がその後の七転八倒ロードに・・・
- jfs: 当時 v1.0 を名乗った IBM アリエナサス! 翌月には xfs に逃避
- xfs: fsck==true に感動。しかし数年使ったもののマシン不調時の 0byte ファイル問題の嵐で耐えられず逃げた

そして reiser4 に超期待するうちにあれがああなって、結局 ext3 に固定化。 htree も入ったし、もう鉄板なら何でもいいです・・・ といいつつ nilfs などに手を出しています。 ext3 の設定は noatime 程度ですが、これに aufs を被せて自分自身だとか *strap 環境をクローニングしています。実験・ テストに便利でお勧めです。 USB メモリ稼動でも有用。

後は LVM ではなく MD を使って冗長化+バックアップをしています。 MD(sda,sdb,sdc) で構築し、通常は MD( 2 台) で稼動。バックアップの時は attach/detach をする。ブロックレベルなのでリカバリは FS 任せですが、容量が増えると他に方法がない・ ・ 。

## 2.7 henrich

使ってる時間は NTFS が長いんじゃないですかね@職場。先日ノート用の新しいディスクを ext4 でフォーマットしましたが全く違いが判らない使い方しかしていません。

## 2.8 emasaka

つるしの FS を使ってます

## 2.9 本庄

ext3 を使用しています。特に変わったことはしていません。FS じゃないですが、最近 Lenny で 2TB の HDD を使ったら parted っての使われて驚きました。

## 2.10 吉田@小江戸

自宅で日常的に活用しているファイルシステムは ReiserFS です。仕事で使ってる環境は ext3 ですが、 ext3 は電源断等でジャーナルが壊れて壊滅した過去の経験（古いカーネルですが...）があり、あまり信用していません。自宅の ReiserFS 環境ではこれまでの所いきなり電源を切ったり、夏に HDD が壊れかけたりしても被害を被った経験が無いので。継続的に使っています。数年前は ReiserFS のメイン開発者 (Hans Reiser) が逮捕されてしまい、メンテナンスを心配していました。しかし、その後も ReiserFS は他の開発者により継続して保守されているので、安心しました。

## 2.11 nozzy123nozzy

1. LVM については、 CentOS5.5 が導入するものそのままに利用してます。ただ、システムが乗っている Volume 名はデフォルトから手作業（実際には kickstart にて）で変更して使ってます。（障害時のサルベージに困るため）
2. ext3 については、 debian-sid がそのまま指定しているものをそのまま使っていたりします。 relatime, noatime ぐらいは将来追加してみたいなーとは思ってます。

## 2.12 まえだこうへい

- Debian では特に凝ったことせず、 ext3 を使ってます。LVM は仮想マシンで qcow2 イメージディスク利用時のみ使ってます。
- うちで一番特殊なのは、自宅の DHCP サーバ用の Armadillo-J で使っている JFFS です。デフォルトのファームウェアではリブートすると設定は全て初期化されてしまうので、 RAM 領域に書きこみ、電源切っても消えない点が便利です。 Debian の udhcp のソースパッケージからビルドして使ってます[*1]
- Debian 絡みで、今自分にとって一番ホットなのは palm webOS です。 Ubuntu をカスタマイズしたものらしいです。マウントポイントは/etc/mtab を見ると 35 行もある、かなり変態構成です。

## 2.13 荒木靖宏

### 2.13.1 日常で活用しているのは ext3 と xfs

LVM は使いますね。典型的なパターンとしては、

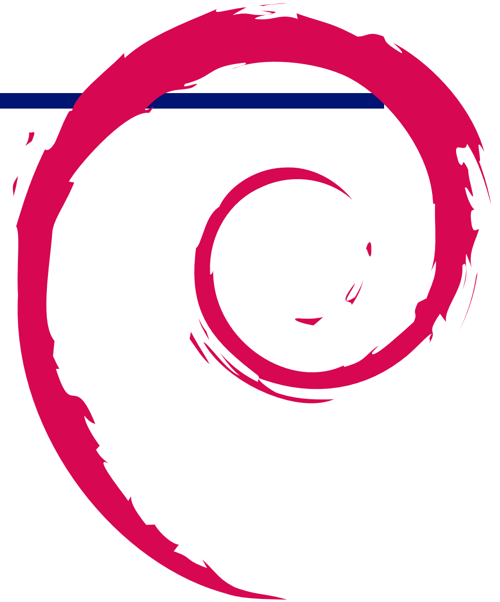
1. vgdisplay、 lvdisplay、 lvcreate をキメる
2. 
   - KVM 用に切りだし (あとは知らん)
   - iSCSI として外だし (あとは知らん)
   - NFS 用に mkfs.xfs
   - ローカル用に意識して使っているものは、 ext3 か xfs

こんなことをやらない日はないとは言いませんが、やらない週はありません。やる日だと延々とやっては壊しの繰り返しております。

- 最近は FUSE を使わなくなりました。 iso の loopback もしなくなりました。
- Ceph はたのしーぜ!って思ってたのですが、ベンチかけてへこんで dis られたので最近は見ていません。
- POSIX の API がない fs はバケツだと思ってるので個人的にはあまり語りたくない。
- PVFS2 は未着手.. これもけっこうおもしろげな分散 fs と思うんですが。
- petardfs はマゾ的でいいですよ。

NILFS、 BtrFS、 Ceph いずれも発表楽しみにしています。

[*1] http://d.hatena.ne.jp/mkouhei/20080601/1212330630



# 3 最近の Debian 関連のミーティング報告

まえだこうへい

## 3.1 東京エリア Debian 勉強会 69 回目報告

久々の通常どおりの Debian 勉強会は 10/16 に、場所は大森のニフティさんのセミナールームをお借りして開催しました。勉強会のサイトでは第 69 回となっていますが、 8 月は開催していないので今回は正確には 68 回です*1。鯖読んでます。が、 68 回は欠番としました。

### 3.1.1 開催内容

今回は最近のイベント、とくに LinuxCon Japan 2010 と Debconf10 の参加報告のあと、事前課題の「 俺の Debian な一日」を参加者全員に自己紹介を兼ねて発表して貰い、日本での Mini Debconf 開催に向けてのブレストを行いました。今回初参加の hattorin さんと tan さんが次回ネタを発表してくれることになったこと、 Mini Conf に向けてのディスカッションは次回以降も続けていくことになりました。

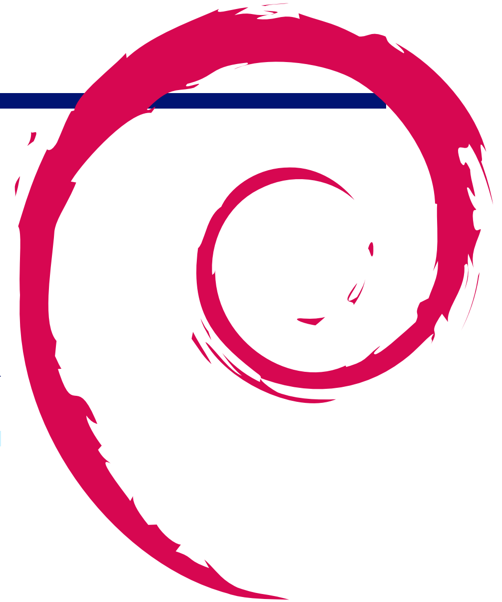
### 3.1.2 参加者

参加者 (敬称略) は、 tai、日比野、あらき、 hattorin、吉野、キタハラ、小室、山本、鈴木、あけど、やまね、上川、まえだの計 13 人でした。

ニフティさんありがとう! ニフティさんには会場をお借りしただけでなく、勉強会の最後に無茶ぶりしてもきちんとコメントも頂き、誠にありがとうございました。

### 3.1.3 宴会

宴会はしゃぶしゃぶ温野菜 大森店で行いました。



# 4 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もち、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。 [1, 2]

## 4.1 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るようです。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。

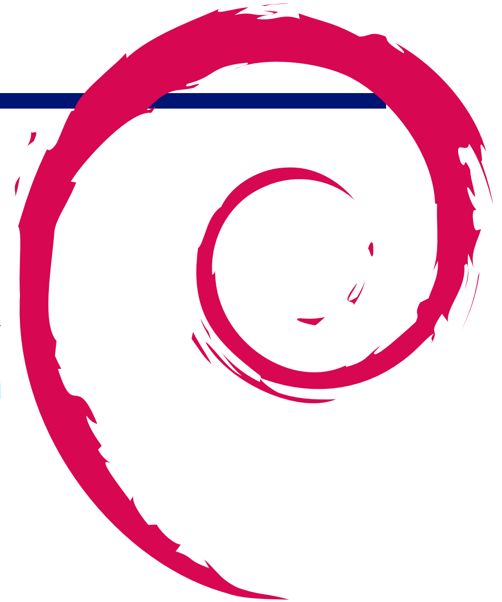
ext4 を使うには十分新しい e2fsprogs と十分あたらしい Linux Kernel があればよいです。 Debian 5.0 (lenny) の時点で必要なパッケージはそろっているようです。

```
$ 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
```

# 参考文献

[1] 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

[2] A. Kumar K. V., M. Cao, J. R. Santos, and A. Dilger. ”Ext4 block and inode allocator improvements,” Linux Symposium, Vol 1. 2008.



# 5 NILFS を Debian で活用してみる

山田 泰資

## 5.1 はじめに

NILFS [*2]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。

## 5.2 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 mklabel 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] segctord 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。答は nilfs の固有コマンド lscp を打つと判ります:

[*2] 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                                        <- うっかり消してしまった! どうしよう! （ 棒
# 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                                              <- スナップショットに変換する
# mount -t nilfs2 -o ro,cp=10 /dev/sda1 /mnt/p0.10        <- 消す前の状態をマウント
# ls /mnt/p0.10/
4 file0  4 file1  4 file2                                 <- 消す前の状態を確認
# cp /mnt/p0.10/file2 /mnt/p0/                            <- 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も、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 1：

| 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 1　nilfs の管理コマンド一覧

## 5.3 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 2：

| | 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 2　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。

## 5.4 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[*3]）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[*4]:

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[*5]。

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ージコレクション処理（GC 処理）をいずれ実行する必要があり、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が、これは

- 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

---

[*3] LFS という名前の実装もあって、ややこしい・・・（JFS も）

[*4] 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]

[*5] 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

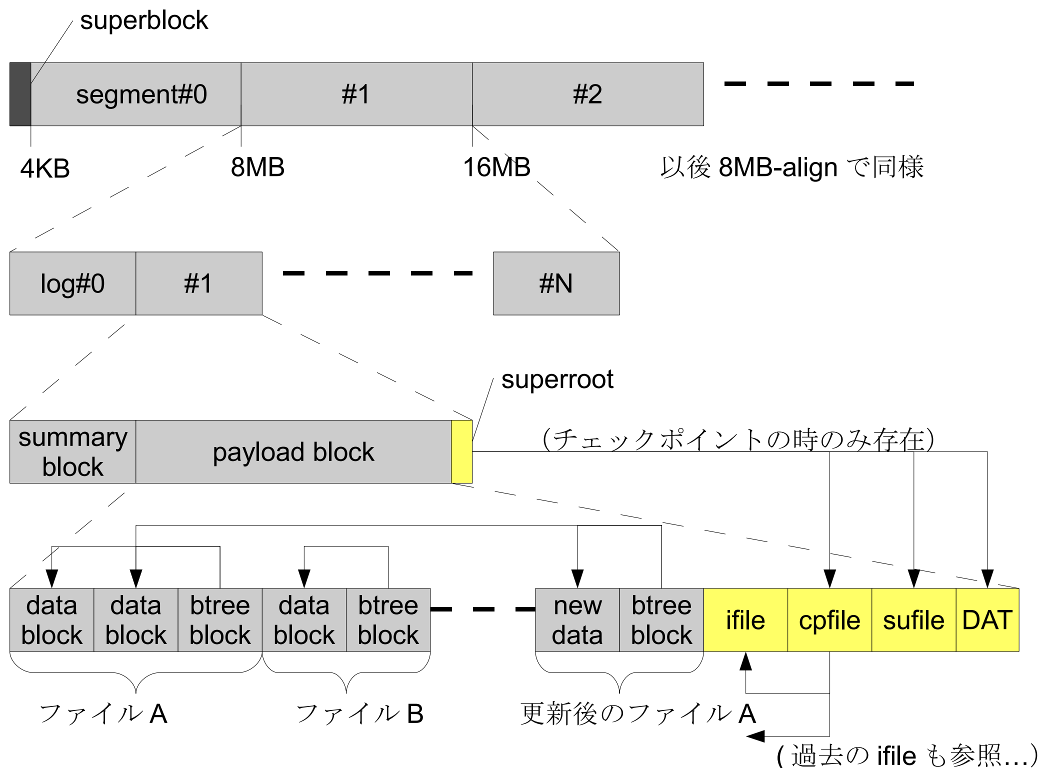



図 1　nilfs のオンディスクフォーマットの概要

- 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 の構造を見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。

## 5.5　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))
// ユーザから見ると 1GB のファイルがあるだけだが・・・
# 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[*6]。

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。

## 5.6 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・運用ポリシを決定します。

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:

---

[*6] 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

| 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<br>• 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<br>• 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<br>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。<br>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<br>1 時間での最大回収量 =<br>8[MB] * nsegents_per_clean * 3600[s] / cleaning_interval[s]<br>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。<br>これも、先の mc_nsegments_per_clean と合わせて<br>1 時間での最大回収量 ＝<br>8[MB] * mc_nsegents_per_clean * 3600[s] / cleaning_interval[s]<br>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 | 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。

## 5.7 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, &prottime, &oldest);
        nilfs_cleanerd_clean_segments(cleanerd, &sustat, segnums, ns, prottime);
        nilfs_cleanerd_recalc_interval(cleanerd, ns, prottime, 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, &prottime, &oldest);
nilfs_cleanerd_clean_segments(cleanerd, &sustat, segnums, ns, prottime);
```

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(vblocknrv),
                           nilfs_vector_get_size(vblocknrv),
                           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。

## 5.8 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。

### 5.8.1 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 3：

| コマンド名 | 機能 |
|---|---|
| `btrfs` | 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。 |
| `mkfs.btrfs` | 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 -m/-d オプションで single, raid0, raid1, raid10 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が変わるため、明示的に指定するとよいでしょう。 |
| `btrfsck` | 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。 |
| `btrfs-image` | いわゆる dump/restore。FS イメージをファイルに落としたり、そこから戻します。 |
| `btrfs-convert` | 現在 ext[234] 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、ext[234] 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。 |
| `btrfs-debug-tree` | デバッグ用。btrfs の内部構造をダンプします。 |

表 3　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 hogehoge > ./v00/v01/hogehoge
# find
.
./v00
./v00/v01
./v00/v01/hogehoge

// これ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*7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の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。

### 5.8.2 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るために、 Linux のカーネルソースをテストでは使いました:

*7 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 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 を使っているので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。

## 5.9 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 x30 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 x30 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ペだからです
# 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
# 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  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分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[*8]。

## 5.10 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[*9]。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[*10] 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[*11][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的ストレージを 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が、nilfs も有力な選択肢として検討してみてはいかがでしょうか?

# 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`


[*8] 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・・・

[*9] 総容量 16GB の Debian-on-USB 箱というささやかなものですが・・・

[*10] 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]

[*11] 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/

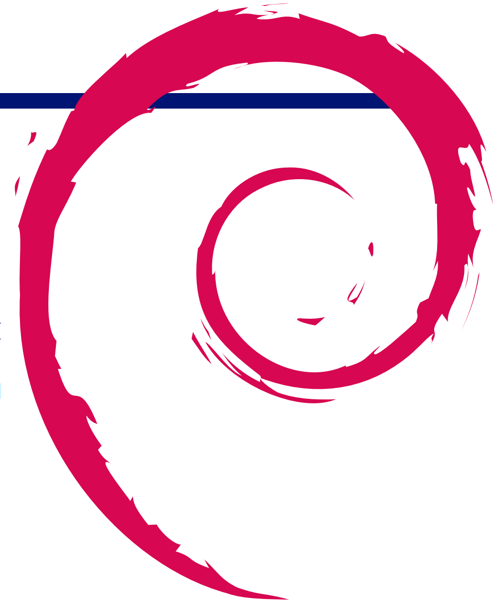
[12] "Btrfs FAQ",
https://btrfs.wiki.kernel.org/index.php/FAQ

[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/



# 6 Btrfs を Debian で活用してみる

鈴木 崇文

## 6.1 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[*12] 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と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
```

## 6.2 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
```

## 6.3 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。

[*12] /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 エクステント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
```

## 6.4 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
```

## 6.5 バックアップ用のファイルシステムとして使ってみる

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
```

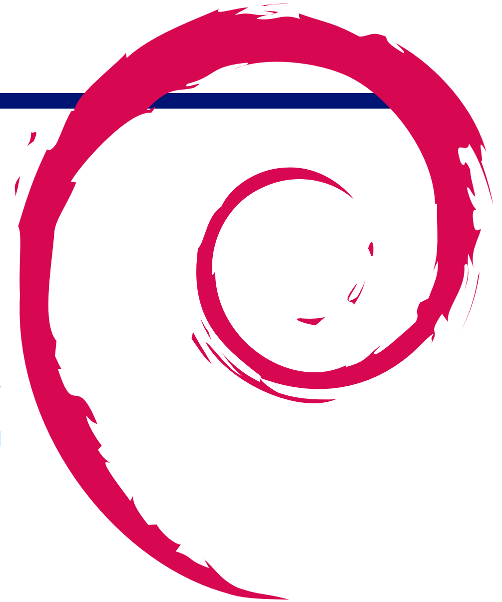
なお、作成したサブボリュームは btrfs subvolume list で表示できるはずでしたが、現時点の Squeeze/Sid ではエラーになってしまいました。

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

## 6.6 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。



# 7 分散ファイルシステム CEPH を Debian で活用してみる

服部 武史

## 7.1 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[*13]。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。

## 7.2 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

## 7.3 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/
```

## 7.4 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。

[*13] ←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/ods$id
        osd journal = /data/osd$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
```

## 7.5 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/
```

## 7.6 テスト

テスト方法として 5 台のマシンそれぞれの HDD を BrtFS 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 BrtFS 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。

### 7.6.1 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
```

### 7.6.2 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
```

### 7.6.3 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
```

### 7.6.4 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
```

### 7.6.5 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のよう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。

## 7.7 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: fffffe4 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
```

# 8 Debian Miniconf 計画検討

山本 浩之

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

## 8.1 本日のミッション

ネタ出しを終了させる。

聞きたい講演のネタは全部吐き出しましょう。来月からは具体的な検討になる予定なので、今がチャンスです。

## 8.2 先月までに決まっていること

### 8.2.1 聴衆のメインターゲット

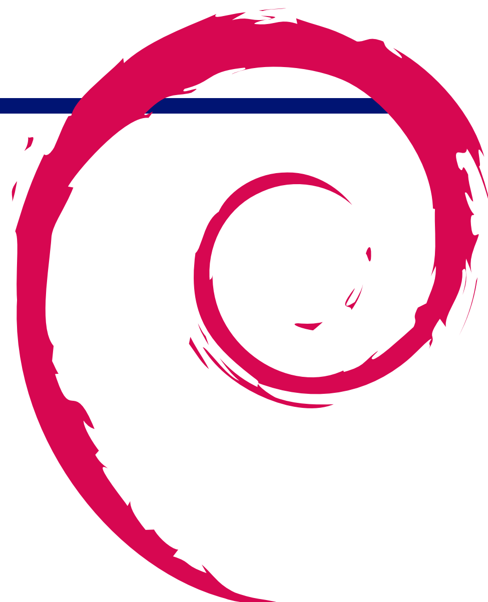
Debian だけに限らない開発者やユーザ。アップストリーム開発者や翻訳者も含む。ビジネス関係の人たちは二の次。

### 8.2.2 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。

### 8.2.3 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 (?)。



# 9 索引

Btrfs, 22

CEPH, 25

ext4, 6

miniconf, 30

NILFS, 7

**Debian 勉強会資料**

2010 年 11 月 20 日　初版第 1 刷発行

東京エリア Debian 勉強会（編集・印刷・発行）