http://people.allbsd.org/~hrs/FreeBSD/sato-FBSD20121207.pdf

FreeBSD 勉強会
ZFSの活用とチューニング

佐藤　広生 <hrs@FreeBSD.org>

東京工業大学/ FreeBSD Project

2012/12/7

講師紹介

佐藤 広生 <hrs@FreeBSD.org>

- *BSD関連のプロジェクトで10年くらい色々やってます
- カーネル開発・ユーザランド開発・文書翻訳・サーバ提供
  などなど
- FreeBSD コアチームメンバ(2006 年から4期目)、
  リリースエンジニア
  (commit 比率は src/ports/doc で 1:1:1くらい)
- AsiaBSDCon 主宰
- 技術的なご相談や講演・執筆依頼は hrs@allbsd.org まで

お話すること

- ZFS
  - 使い方
    - ストレージプール(zpool)の作成
    - データセットの作成
    - スナップショット
    - ZFSのみで運用(rootfs on ZFS)
  - 性能とチューニング
    - 技術詳細と利点/欠点
    - 各設定項目、運用のtips
    - 統計のモニタリング

ZFSの概要

- **ZFS とは?**
  - Solaris 用に開発されたファイルシステム
  - 多数の記憶装置があることを前提に、
    - スケーラビリティ
    - データの冗長性確保
    - データ破壊に対する耐性

    を高めた実装
- 今までのファイルシステムの良いとこ取りをしている

ZFSの概要

- **ZFSの構造**
  - 従来:
    デバイス→ボリュームマネージャ→ファイルシステム
  - ZFS :
    ZPOOL→DMU→ZFS

ZFSの概要

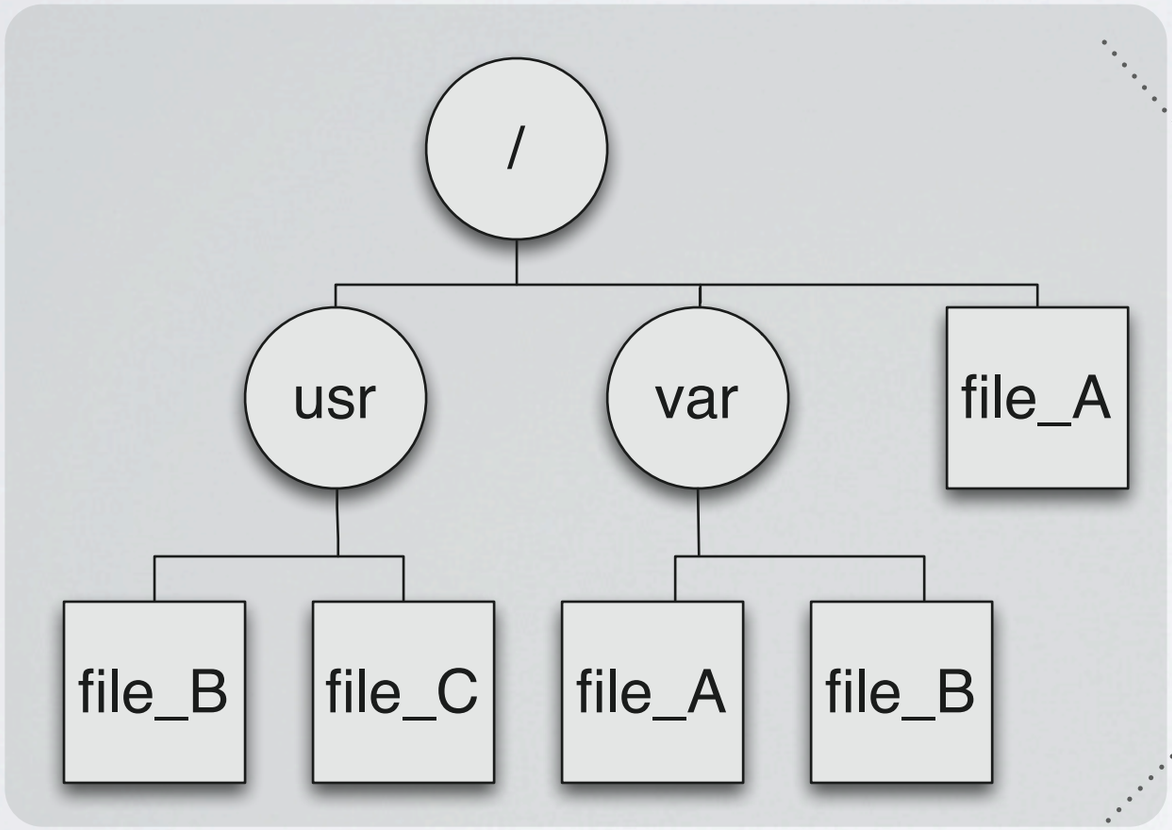
- **用語**
  - **ZPOOL**: ブロックデバイスの集まりに名前を付けたもの（ストレージプール）
    - 例）/dev/ada0 + /dev/ada1 で 1 個の ZPOOL を構成

ZFSの概要

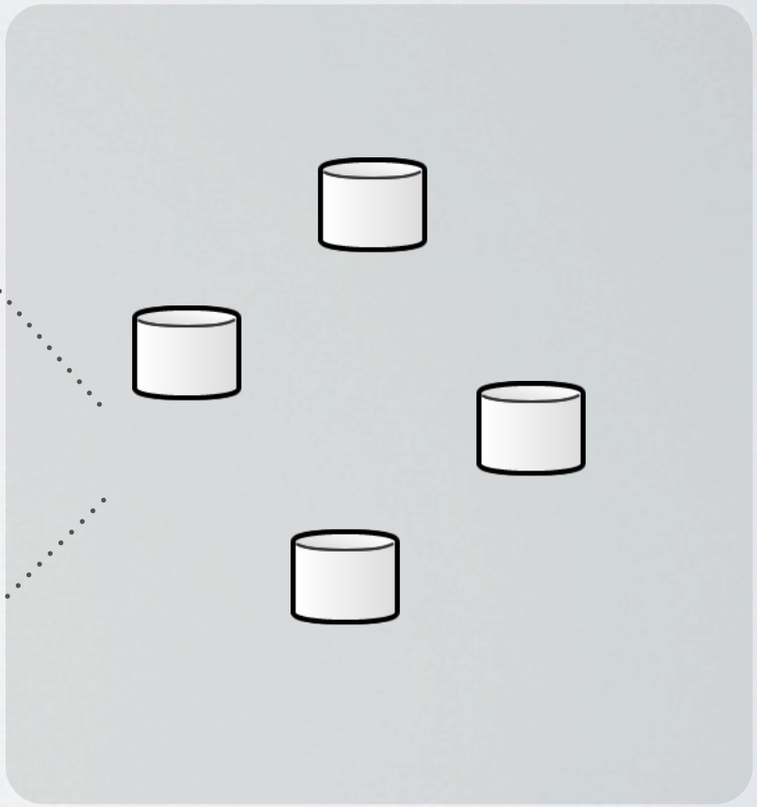
- **用語**
  - **ZPOOL**: ブロックデバイスの集まりに名前を付けたもの（ストレージプール）
    - 例）/dev/ada0 + /dev/ada1 で 1 個の ZPOOL を構成
    - 構成の方法にバリエーションがある
      - /dev/ada0 と /dev/ada1 をミラーしつつ構成
      - /dev/ada{0,1,2,3,4} で RAID5 を組んで構成
      - どれも「1 個の記憶装置」に見える

ZFSの概要

- **用語**
  - **ZFSデータセット**: ZPOOLの上に構成できる名前空間
    ＝ファイルシステム、ボリューム、スナップショット



データセット



ZPOOL

ZFSの概要

- **従来のUFSとの比較**
  - ZPOOLで記憶装置を管理
  - ZFSデータセット1個＝ファイルシステム1個（相当）

ZFSの使い方

- **FreeBSD におけるZFS**
  - OpenSolaris のコードをほぼそのまま導入
    - SPA(ストレージプールアロケータ)バージョン 28
    - ZPL(ZFS POSIX Layer)バージョン 5

ZFSの使い方

- **FreeBSD におけるZFS**
  - OpenSolaris のコードをほぼそのまま導入
    - SPA(ストレージプールアロケータ)バージョン 28
    - ZPL(ZFS POSIX Layer)バージョン 5
  - 現在は illumos project のコードで保守されている
    - illumos = OpenSolaris から分岐
      - OpenIndiana = illumos ベースの OS
      - Nexenta = illumos ベースの software-defined storage (NexentaStor)
      - Joyent = illumos ベースの cloud

ZFSの使い方

- **まずは使ってみよう**
- **条件：**
  - FreeBSD 8.3 以降、もしくは 9 系を使うこと
  - amd64 で 4GB 以上の RAM を載せて使うこと

- これを満たしていない場合は、満足に性能が出ません

ZFSの使い方

- **まずは使ってみよう**
- **期待できる効果：**
  - 多数のHDDを接続している場合の管理が楽になる
  - スナップショットの機能が使える
  - jail の中でファイルシステムの操作ができる

- HDDの数が少なく、上記機能を使わないのであれば、
  あまり嬉しいことはありません

ZFSの使い方

- **まずは使ってみよう**
- **準備：**
  - FreeBSD の ZFS 機能は、カーネルモジュールで提供
    （ライセンスの問題があるので標準では入らない）
  - 次の操作でモジュールがロードされる
    - zfs コマンドを実行する
    - zpool コマンドを実行する
    - /boot/loader.conf に zfs_load=”YES” を入れる
    - /etc/rc.conf に zfs_enable=”YES” を入れる

ZFSの使い方

- **まずは使ってみよう**
- **手順：**
  - ストレージプールをつくる
  - データセットをつくる
  - （最も単純なケースでは明示的にやらなくても可）

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - ZFS で利用する記憶装置を決める作業
  - 論理的な記憶装置＝物理記憶装置の集合体

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - ZFS で利用する記憶装置を決める作業
  - 論理的な記憶装置＝物理記憶装置の集合体
  - ストレージプールの操作は「zpool コマンド」

```
# zpool create pool /dev/da1 /dev/da2
# zpool list -v
NAME           SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
pool          3.97G  92.5K  3.97G     0%  1.00x  ONLINE  -
  da1         1.98G    36K  1.98G         -
  da2         1.98G  56.5K  1.98G         -
```

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - 「pool」という名前のストレージプールができる
  - 「zpool create pool <記憶装置...>」とすると
    - 容量は合計値、冗長性なし
    - アクセスはストライピング

```
# zpool create pool /dev/da1 /dev/da2
# zpool list -v
NAME           SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
pool          3.97G  92.5K  3.97G     0%  1.00x  ONLINE  -
  da1         1.98G    36K  1.98G         -
  da2         1.98G  56.5K  1.98G         -
```

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - ストレージプールをつくると、デフォルトでは勝手にマウントされる
  - マウント位置は変更できる

```
# df
Filesystem            1K-blocks    Used    Avail Capacity  Mounted on

....

pool                    4096450      31  4096419     0%    /pool
```

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - ストレージプールをつくると、デフォルトでは勝手にマウントされる
  - マウント位置は zfs コマンドで変更できる

```
# zfs set mountpoint=/pool-2 pool
# df
Filesystem            1K-blocks   Used   Avail Capacity  Mounted on

....

pool                    4096450     31 4096419     0%    /pool-2
```

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - 実はこのままファイルシステムとして使える
  - ストレージプールをつくると、自動的に同じ名前で「ファイルシステムデータセット」も作成される

```
# zfs set mountpoint=/pool-2 pool
# df
Filesystem           1K-blocks   Used   Avail Capacity  Mounted on

....

pool                   4096450     31 4096419     0%    /pool-2
```

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - いつマウントされるのか?
    - カーネルモジュールがロードされ、ZFSが有効になった時
  - rc.conf に zfs_enable=”YES” を書いておく
  - /etc/fstab は**見ない**

```
# zfs set mountpoint=/pool-2 pool
# df
Filesystem           1K-blocks   Used   Avail Capacity  Mounted on

....

pool                   4096450     31 4096419     0%    /pool-2
```

ZFSの使い方

- **まずは使ってみよう**
- **ストレージプールをつくる**
  - ストレージプールの消去
    - zpool destroy <ストレージプール名>
      - 中のデータも全部消える

```
# zpool destroy pool
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - データセットとは？
    - ファイルシステム
    - ボリューム
    - スナップショット
  - 操作は「zfsコマンド」を使う

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ファイルシステムをつくる
    - zfs create <プール名>/<データセット名>
      - これも自動的にマウントされる

```
# zfs create pool/data-a
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    143K  3.91G    31K  /pool
pool/data-a              31K  3.91G    31K  /pool/data-a
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ファイルシステムをつくる
    - mountpoint を指定してつくることも可能

```
# zfs create -o mountpoint=/a pool/data-a
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    143K  3.91G    31K  /pool
pool/data-a              31K  3.91G    31K  /a
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ファイルシステムをつくる
    - データセットは階層構造を持つが、
      マウントポイントの階層構造と一致していなくても良い

```
# zfs create -o mountpoint=/a pool/data-a
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    143K  3.91G    31K  /pool
pool/data-a              31K  3.91G    31K  /a
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ファイルシステムをつくる
    - 「mountpoint=/a」のような設定項目を、「プロパティ」と呼ぶ。いろいろある。

```
# zfs create -o mountpoint=/a pool/data-a
# zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    143K  3.91G    31K  /pool
pool/data-a              31K  3.91G    31K  /a
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - スナップショットをつくる
    - ファイルシステムの、ある時点の状態を保存する機能
    - zfs snapshot <プール名>/<データセット名>@<スナップショット名>
  - 自動マウントはされない

```
# zfs snapshot pool/data-a@20121207
# zfs list -t all
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    143K  3.91G    31K  /pool
pool/data-a              31K  3.91G    31K  /a
pool/data-a@20121207       0      -    31K  -
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - スナップショットをつくる
    - 中身は snapdir プロパティを visible にすると見える
  - プロパティの操作は zfs set を使う
  - ドットzfsという専用ディレクトリができる（readonly)

```
# zfs set snapdir=visible pool/data-a@20121207
# ls -lad /a/.zfs/snapshot/20121207
drwxr-xr-x  2 root  wheel  2 Dec  7 12:39 /a/.zfs/snapshot/20121207
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - スナップショットは何に使うの？
    - 基本的にバックアップのため
    - あとでスナップショットまで戻せる
    - zfs rollback

```
# touch /a/a
# ls -al /a/a
rw-r--r--  1 root  wheel  0 Dec  7 12:52 /a/a
# zfs rollback pool/data-a@20121207
# ls -al /a/a
ls: /a/a: No such file or directory
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - スナップショットは何に使うの？
    - 同じプール内で複製が簡単にできる（書込可能な複製）
    - zfs cloneコマンド。複製は差分記録なので容量節約に。
    - 「-o mountpoint=/b」を付けることも可能

```
# zfs clone pool/data-a@20121207 pool/data-b
# zfs list -t all
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    204K  3.91G    31K  /pool-2
pool/data-a              32K  3.91G    31K  /a
pool/data-b               1K  3.91G    31K  /pool-2/data-b
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - スナップショットは何に使うの？
    - 外部にバックアップをとる
      - zfs send
      - バイトストリームでデータが全部吸い出せる

```
# zfs send pool/data-a@20121207 > /tmp/data-a.zfs
# ls -al /tmp/data-a.zfs
-rw-r--r--  1 root  wheel  47696 Dec  7 12:56 /tmp/data-a.zfs

# file /tmp/data-a.zfs
/tmp/data-a.zfs: ZFS shapshot (little-endian machine), version 17, type:
ZFS, destination GUID: 00 00 00 00 46 64 01 A0, name: 'pool/data-
a@20121207'
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - スナップショットは何に使うの？
    - バックアップはもちろん元に戻せる
      - zfs recv (receive)
      - スナップショットの状態等も全部もどるが...

```
# zfs recv pool/data-a2 < /tmp/data-a.zfs
# zfs -t all
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    246K  3.91G    32K  /pool-2
pool/data-a              32K  3.91G    31K  /a
pool/data-a@20121207      1K      -    31K  -
pool/data-a2             31K  3.91G    31K  /pool-2/data-a2
pool/data-a2@20121207     0      -    31K  -
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - zfs send/recv の注意点
    - 自動マウントされないようにするには recv に-uを付ける
    - プロパティも全部反映させるには、send に-pを付ける
    - ただし、重複すると困るプロパティもあるので注意

```
# zfs send -p pool/data-a@20121207 > /tmp/data-a.zfs
# zfs recv pool/data-a2 < /tmp/data-a.zfs
# zfs -t all
NAME                      USED  AVAIL  REFER  MOUNTPOINT
pool                      241K  3.91G    31K  /pool-2
pool/data-a                32K  3.91G    31K  /a
pool/data-a@20121207        1K      -    31K  -
pool/data-a2               31K  3.91G    31K  /a
pool/data-a2@20121207       0      -    31K  -
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - zfs destroy と階層構造
    - スナップショットをつくる＝データセットに階層ができる
  - zfs destroy -r オプション

```
# zfs destroy pool/data-a2
cannot destroy 'pool/data-a2': filesystem has children
use '-r' to destroy the following datasets:
pool/data-a2@20121207
# zfs destroy -r pool/data-a2
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ボリューム：もうひとつのデータセット
    - zfs create -V <サイズ>
    - ブロックデバイスとして見える
    - /dev/zvol/<プール名>/<データセット名>

```
# zfs create -V 100m pool/vol0
# ls -al /dev/zvol/pool/vol0
crw-r-----  1 root  operator    0, 112 Dec  7 13:07 /dev/zvol/pool/vol0
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ボリューム：もうひとつのデータセット
    - 単純な記憶装置と同じなので、newfs などができる
    - ZFS をボリュームマネージャとしてだけ使う
    - たとえば/etc/fstab に書いてマウント可

```
# zfs create -V 100m pool/vol0
# ls -al /dev/zvol/pool/vol0
crw-r-----  1 root  operator    0, 112 Dec  7 13:07 /dev/zvol/pool/vol0
# newfs -U /dev/zvol/pool/vol0
/dev/zvol/pool/vol0: 100.0MB (204800 sectors) block size 32768, fragment
size 4096
    using 4 cylinder groups of 25.03MB, 801 blks, 3328 inodes.
super-block backups (for fsck_ffs -b #) at:
 192, 51456, 102720, 153984
# mount /dev/zvol/pool/vol0 /mnt
```

ZFSの使い方

- **まずは使ってみよう**
- **データセットをつくる**
  - ボリューム：もうひとつのデータセット
    - スナップショットをつくったり、
      zfs send/recv がファイルシステムと同じようにできる
  - UFS を使いつつ、スナップショットを使える

```
# zfs snapshot pool/vol0@20121207
# zfs send pool/vol0@20121207 > /tmp/vol0.zfs
# ls -al /tmp/vol0.zfs
-rw-r--r--  1 root  wheel  593840 Dec  7 13:13 /tmp/vol0.zfs
```

ZFSの使い方

- **まずは使ってみよう**
- **ここまでのまとめ**
  - zpool: ストレージプールをつくる
  - zfs: データセットを操作
    - データセット
      - ファイルシステム
      - スナップショット
      - ボリューム
  - スナップショットはsend/recvでバイトストリームにできる
  - スナップショットはロールバックできる
  - clone を使うと書込可能で容量を節約した複製が作れる

ZFSの使い方

- **まずは使ってみよう**
- **zfs_enable="YES" の起動時の処理**
  - /boot/zfs/zpool.cache を読んでストレージプールを認識
  - /etc/fstab を読む前に、データセットが認識、
    ファイルシステムは自動マウントされる

ZFSの使い方

- **まずは使ってみよう**
- **ふつうに使うために必要な設定まとめ**
  - zpool でストレージプールをつくる
  - zfs でデータセットをつくる
  - zfs_enable=”YES” を /etc/rc.conf に書く

進んだ使い方

- 1) zfs のデータセットの作り方
- 2) zfs send/recv を使ったバックアップ
- 3) zfs diff
- 4) zpool ストレージプールの詳細
- 5) zfs scrub
- 6) zpool import/export コマンド
- 7) ZFS をルートファイルシステムとして使う

ZFSデータセットの作り方

- **必ず2レベルにして、プールと同名のデータセットは使わない**
  - 例）pool という名前のプールにファイルシステムを作る
  - zfs set mountpoint=none pool
  - zfs create -o mountpoint=/data-a pool/data-a
  - zfs create -o mountpoint=/data-b pool/data-b

```
# zfs list -t all
NAME                    USED  AVAIL  REFER  MOUNTPOINT
pool                    225K  3.91G    31K  none
pool/data-a              31K  3.91G    31K  /a
pool/data-b              31K  3.91G    31K  /b
```

ZFSデータセットの作り方

- **必ず2レベルにして、プール名と同じデータセットは使わない**
  - 理由：1レベルはスナップショットの扱いが難しいため
  - zfs snapshot pool@1
  - zfs send pool@1 > pool@1.zfs
  - NG: zfs recv pool < pool@1.zfs
  - OK: zfs recv pool/pool2 < pool@1.zfs

- pool のスナップショットを pool/pool2 として
  復元しなければならない

ZFSデータセットの作り方

- **必ず2レベルにして、プール名と同じデータセットは使わない**
  - 2レベルの場合
  - zfs snapshot pool/data@1
  - zfs send pool/data@1 > pool@1.zfs
  - zfs recv pool/data2 < pool@1.zfs

  - pool/data のスナップショットを pool/data2 として書き戻せる＝同じ階層なので直感的に分かりやすい
  - もちろん、recv には -u をいつも付けることを推奨

ZFS send/recv

- **定期的なスナップショットとオフサイトバックアップ**
  - 階層構造を全部とる：

    zfs snapshot -r pool/data@20121205
    zfs snapshot -r pool/data@20121206
    zfs snapshot -r pool/data@20121207
  - 全部のバックアップ

    zfs send pool/data@20121205 > data.20121205
  - 差分バックアップ

    zfs send -I pool/data@20121205 pool/data@20121207 > data.20121205-07
  - 全部＋差分を順番に recv できる

    zfs recv pool/data_new < data.20121205
    zfs recv pool/data_new < data.20121205-07

ZFS send/recv

- **定期的なスナップショットとオフサイトバックアップ**
  - バックアップ戦略の例
    - 全部のバックアップを1週間に1回、差分を1日に1回
    - 復元はストレージプールをつくりなおして、
      順番に zfs recv するだけ
  - オフサイトバックアップ
    - 単なるバイトストリームなので、SSH 等で転送できる
    - dump & restore でやっていた技法がそのまま適用可能

zfs send pool/data@20121201 | ssh somewhere “cat > /backup/data.20121201”

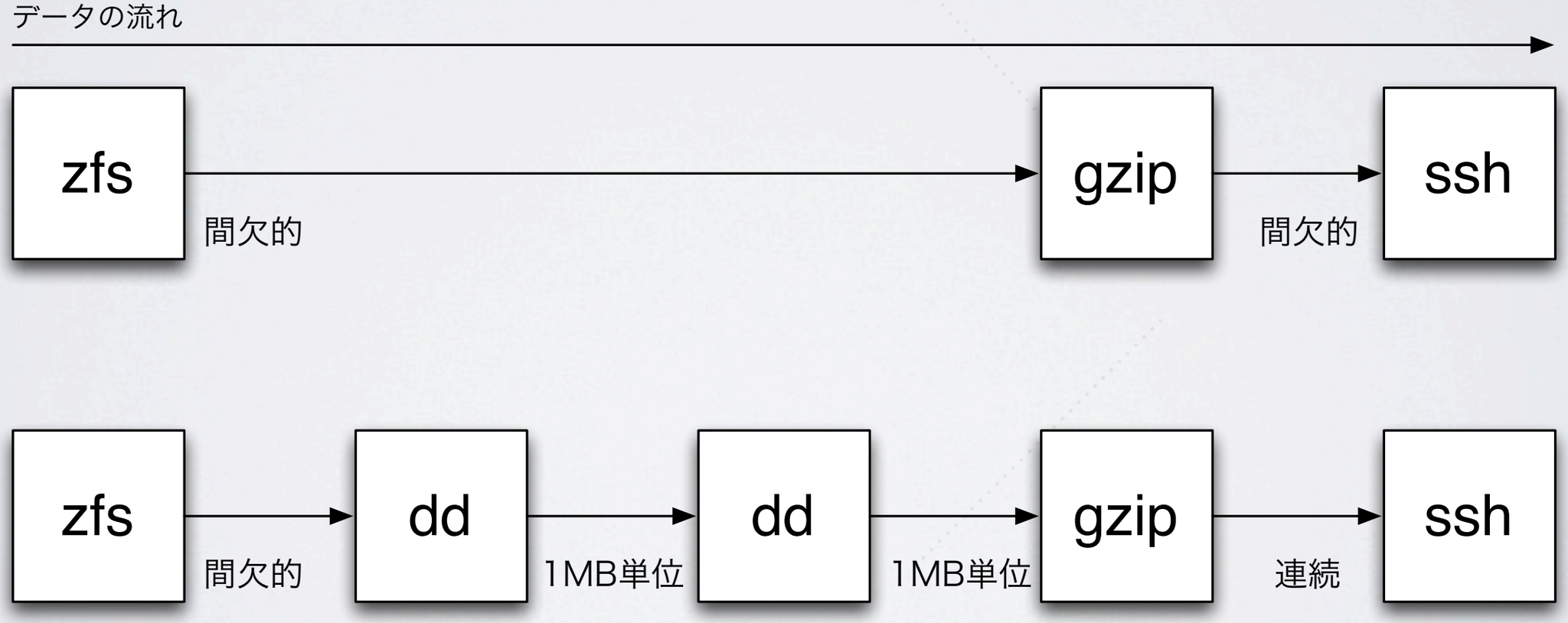
ZFS send/recv

- **定期的なスナップショットとオフサイトバックアップ**
  - Tips: バッファリングと圧縮、暗号化
    - バイトストリームはパイプ処理で圧縮や暗号化できる
    - zfs send はデータの出力が間欠的なので、
      処理時間が遅くなることがある

ZFS send/recv

- **定期的なスナップショットとオフサイトバックアップ**
  - Tips: バッファリングと圧縮、暗号化

```
# zfs send pool/data@20121207 | dd obs=1m | dd obs=1m | gzip -9 | ssh somewhere “cat > file.gz”
```



ZFS send/recv

- **定期的なスナップショットとオフサイトバックアップ**
  - Tips: バッファリングと圧縮、暗号化
    - 圧縮や暗号化はデータをブロック単位で
      入力することが多く、処理が重いので出口が遅い
    - データが連続して供給されないと、さらに遅くなる
    - 入り口のデータをなるべく連続にすることで、
      総合的な処理時間は短くなる

ZFS diff

- **スナップショットの差分 (9系から)**
  - zfs diff pool/root@1 pool/root@2

```
M       /etc
M       /etc/ssh
-       /etc/master.passwd
-       /etc/spwd.db
-       /etc/passwd
-       /etc/pwd.db
M       /etc/ssh/sshd_config
+       /etc/resolv.conf
+       /etc/ssh/ssh_host_key
+       /etc/ssh/ssh_host_key.pub
+       /etc/ssh/ssh_host_dsa_key
+       /etc/ssh/ssh_host_dsa_key.pub
+       /etc/ssh/ssh_host_rsa_key
+       /etc/ssh/ssh_host_rsa_key.pub
+       /etc/ssh/ssh_host_ecdsa_key
+       /etc/ssh/ssh_host_ecdsa_key.pub
+       /etc/master.passwd
+       /etc/pwd.db
+       /etc/spwd.db
+       /etc/passwd
```

ZFS ストレージプール

- **ストレージプールの種類と選択**
  - 最初の例でつくったのはストライププール
  - **ミラープール**
    - zpool create pool mirror da1 da2
    - 冗長性がある。容量は半分。
- **RAID-Z プール**
  - zpool create pool raidz da1 da2 da3
  - シングルパリティの RAID5 相当。2台以上。
  - サイズX, ディスクN台, パリティ数Pだと
    (N-P) x X の容量になる（シングルパリティ：P=1）
  - 3台なら 66% 程度の容量。冗長性あり(P台壊れても可)

ZFS ストレージプール

- **ストレージプールの種類と選択**
  - **RAID-Z2 プール**
    - zpool create pool raidz2 da1 da2 da3
    - ダブルパリティ RAID5 相当。3台以上。
    - 3台なら 33% 程度の容量(P=2)
  - **RAID-Z3 プール**
    - トリプルパリティ RAID5 相当。4台以上(P=3)
    - zpool create pool raidz3 da1 da2 da3 da4

ZFS ストレージプール

- **ストレージプールの種類と選択**
  - 冗長性のないプールは作らないこと
    - 最低でもミラーにする
    - ZFSはデータの信頼性をコピーに頼っている
    - ホットスペアもプールに追加して作成できる

```
# zpool create pool mirror da1 da2 spare da3
```

    - ディスク全体を使うよりも、サイズを指定した
      パーティション(da0p1等)を使った方が良い場合がある
    - gpart create -s gpt da0
    - gpart add -t freebsd-zfs -s 500m da0

同じか大きいディスク容量のものでないとミラーにならない！

ZFS ストレージプール

- **ストレージプールの操作**
  - 冗長性があるプールは、削除・交換・挿入ができる

```
# zpool create pool mirror da1 da2

# zpool replace pool da2 da3 （交換）

# zpool offline pool da2 （停止）
# zpool online pool da2 （有効化）

# zpool add pool da4 （追加）
# zpool attach pool da4 （追加）

# zpool remove pool da4 （削除）
# zpool detach pool da4 （削除）
```

ZFS ストレージプール

- **ストレージプールの操作**
  - zpool add を使うと、プール形式を変えないまま増える

```
# zpool create pool da1
# zpool list -v
NAME            SIZE   ALLOC    FREE     CAP   DEDUP   HEALTH   ALTROOT
pool           1.98G   107K   1.98G      0%   1.00x   ONLINE   -
  da1          1.98G   107K   1.98G           -

# zpool add pool da2
# zpool list -v
NAME            SIZE   ALLOC    FREE     CAP   DEDUP   HEALTH   ALTROOT
pool           3.97G   144K   3.97G      0%   1.00x   ONLINE   -
  da1          1.98G   144K   1.98G           -
  da2          1.98G      0   1.98G           -
```

ZFS ストレージプール

- **ストレージプールの操作**
  - zpool attach を使うと冗長性のないプールがミラーになる

```
# zpool create pool da1
# zpool list -v
NAME           SIZE   ALLOC    FREE     CAP   DEDUP   HEALTH   ALTROOT
pool          1.98G   107K  1.98G      0%   1.00x   ONLINE   -
  da1         1.98G   107K  1.98G           -

# zpool attach pool da2
# zpool list -v
NAME           SIZE   ALLOC    FREE     CAP   DEDUP   HEALTH   ALTROOT
pool          1.98G   156K  1.98G      0%   1.00x   ONLINE   -
  mirror      1.98G   156K  1.98G           -
    da1           -      -      -           -
    da2           -      -      -           -
```

ZFS ストレージプール

- **ストレージプールの操作**
  - zpool attach を使うと冗長性のないプールがミラーになる

```
# zpool create pool da1
# zpool list -v
NAME           SIZE   ALLOC    FREE     CAP   DEDUP   HEALTH   ALTROOT
pool          1.98G   107K  1.98G      0%   1.00x   ONLINE   -
  da1         1.98G   107K  1.98G           -

# zpool attach pool da2
# zpool list -v
NAME           SIZE   ALLOC    FREE     CAP   DEDUP   HEALTH   ALTROOT
pool          1.98G   156K  1.98G      0%   1.00x   ONLINE   -
  mirror      1.98G   156K  1.98G           -
    da1           -      -      -           -
    da2           -      -      -           -
```

ZFS scrub

- **ZFS のデータ破壊検出機能**
  - zpool scrub は、**ZFS 以外の要因によるデータ破壊**を検出し、自動的に修復する機能
  - 実行しなくとも、エラーが検出されればその時に修復される

```
# zpool scrub pool
# zpool status
  pool: pool
 state: ONLINE
  scan: scrub in progress since Fri Dec  7 14:12:32 2012
        396M scanned out of 1.31G at 198M/s, 0h0m to go
        0 repaired, 29.44% done
config:

    NAME          STATE     READ WRITE CKSUM
    pool          ONLINE       0     0     0
      da1         ONLINE       0     0     0
      da2         ONLINE       0     0     0
```

ZFS scrub

- **ZFS のデータ破壊検出機能**
  - どれくらいの頻度でやると良いか？
    - 家庭用PCなら3週間に1回
    - 業務用機なら1-2ヶ月に1回

```
# zpool scrub pool
# zpool status
  pool: pool
 state: ONLINE
  scan: scrub in progress since Fri Dec  7 14:12:32 2012
        396M scanned out of 1.31G at 198M/s, 0h0m to go
        0 repaired, 29.44% done
config:

    NAME          STATE     READ WRITE CKSUM
    pool          ONLINE       0     0     0
      da1         ONLINE       0     0     0
      da2         ONLINE       0     0     0
```

ZFS scrub

- **ZFS のデータ破壊検出機能**
  - periodic に設定しておくと、自動化できる
  - /etc/periodic.conf に次の記述を追加

```
daily_scrub_zfs_enable="YES"　（scrubを行う。デフォルトは無効）
daily_scrub_zfs_pools="pool"　（scrub対象のプール名。デフォルトは全部）
daily_scrub_zfs_default_threshold="35"　（何日おきにscrubするか。デフォルトは35日）
daily_scrub_zfs_<プール名>_threshold="35"（プール単位の日数指定）

daily_status_zfs_enable="YES"　（ZFSの状態を出力）
daily_status_zfs_zpool_list_enable="YES"　（プールの一覧を出力）
```

zpool import/export

- **ストレージプールの認識**
  - zpool export <プール名>
    - そのプールが認識されなくなる
  - zpool import <プール名>
    - 接続されているディスクからプールを探して認識する

- 認識は/boot/zfs/zpool.cacheを読んで行っているため、import/export は、このファイルに登録するための明示的な操作だと考えて良い

RootFS on ZFS

- **ZFS をルートファイルシステムに使う**
  - 9.1 以降はようやく安全になってきた
  - まだ手動で全部やる必要があるので、ちょっと面倒
  - UFS なしの生活が可能

- CDROMから起動して、インストールするところまでを紹介

RootFS on ZFS

- **手順**
  - 起動ディスクに GUID パーティションを使う
  - 起動ローダ(セカンダリ)にgptbootではなくgptzfsbootを使う
  - ローダの段階で ZFS のカーネルモジュールを読ませる
  - データセットを loader.conf に指定する
  - zpool.cache を予めつくっておく
  - ストレージプールに bootfs プロパティを付ける

RootFS on ZFS

- **手順**
  - CDから起動してシングルユーザモードへ。
  - gpartでパーティションを切る
    - 例は da0。SATAだと ada0 になるかも
    - ミラーにしたい場合は、これを台数分繰り返す。

```
gpart create -s gpt da0　（GPTパーティション作成）
gpart add -s 122 -t freebsd-boot -l boot0 da0　（起動パーティション作成。「boot0」）
gpart add -s 128m  -t freebsd-swap -l swap0 da0（スワップ用領域。「swap0」）
gpart add -t freebsd-zfs -l disk0（ストレージプールの領域。「disk0」）
gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1 da0（起動ローダの書き込み）
```

RootFS on ZFS

- **手順**
  - ストレージプールを作成し、データセットをつくる
  - CDROM は書き込みできないため、
    /media にマウントするように指示している
  - 例ではミラーにしていないが、ミラー推奨
  - 他のデータセットは自由につくって良いが....

```
zpool create -o altroot=/media rpool /dev/gpt/disk0 (disk0 から rpool 作成)
zfs create rpool/root
zfs create rpool/root/tmp
zfs create rpool/root/usr
zfs create rpool/root/usr/local
zfs create rpool/root/var
```

RootFS on ZFS

- **手順**
  - rc.conf と loader.conf を作成
  - zpool.cache をそこにコピーする（卵鶏問題）

```
mkdir -p /media/root/boot
mkdir -p /media/root/boot/zfs
mkdir -p /media/root/etc
echo 'zfs_enable="YES"' >> /media/root/etc/rc.conf
echo 'zfs_load="YES"' >> /media/root/boot/loader.conf
echo 'vfs.root.mountfrom="zfs:rpool/root"' >> /media/root/boot/loader.conf

zfs unmount -a   （一度全部アンマウント）
zpool export rpool   （ストレージプールを zpool.cache から消す）
mount -t tmpfs tmpfs /tmp   （/tmp を書き込み可能に）
zpool import -o cachefile=/tmp/zpool.cache -o altroot=/media rpool
                （zpool.cache の作成場所を指定しつつ、もう一度ストレージプールを認識させる）
cp /tmp/zpool.cache /media/root/boot/zfs/   （zpool.cache を中にコピー）
zfs unmount -a
umount /tmp
```

RootFS on ZFS

- **手順**
  - プロパティの設定

```
zpool set bootfs=rpool/root rpool （ルートファイルシステムになるデータセットを指定）
zpool set cachefile="" rpool （cachefile は空にする。これは記録されない）
zfs set mountpoint=none rpool （レベル１はマウントしない）
zfs set mountpoint=/ rpool/root（レベル２より下をマウント）
zfs set mountpoint=/tmp rpool/root/tmp
zfs set mountpoint=/usr rpool/root/usr
zfs set mountpoint=/usr/local rpool/root/usr/local
zfs set mountpoint=/var rpool/root/var
zfs get -r mountpoint rpool （この表示が正しいかチェック）
```

注意：ここで

```
cannot mount '/media/usr': failed to create mountpoint
```

のような警告がたくさん出るが、無視して良い。マウントポイントに対応するディレクトリがないので作ろうとするが、CDROMなので作成できない

RootFS on ZFS

- **手順**
  - CDROMからインストール

```
tar xzpvf /usr/freebsd-dist/base.txz -C /media
tar xzpvf /usr/freebsd-dist/src.txz -C /media
tar xzpvf /usr/freebsd-dist/kernel.txz -C /media
```

これだけで完了！

RootFS on ZFS

- **手順**
  - スワップ領域はZFSの外なので、/etc/fstab を書く

```
# echo “swap0 none swap sw 0 0” > /etc/fstab
```

RootFS on ZFS

- **手順**
  - ZFSで起動できるかチェック（9.1以降）
    - zfsboottest というツールをインストール
    - zfsboottest.sh <プール名> でチェック
    - 実行するのはスクリプトなので注意

```
# cd /media/usr/src/tools/tools/zfsboottest
# make && make DESTDIR=/media install
# env PATH=${PATH}:/media/usr/bin sh ./zfsboottest.sh rpool
The "mountpoint" property of dataset "rpool/root" should be set to "legacy".

  (mountpoint  の指定は、全部アンマウントしないと操作できない)


# zfs unmount -a
# zfs set mountpoint=legacy rpool/root
```

RootFS on ZFS

- **手順**
  - おしまい：reboot で再起動！
  - このインストール方法だと、root のパスワードは空になる
  - df で ZFS になっているかチェック
  - 後の運用は、特別な操作は必要ない

```
# df
Filesystem            1K-blocks   Used   Avail Capacity  Mounted on
rpool/root              5037816 383322 4654493     8%    /
devfs                         1      1       0   100%    /dev
rpool/root/tmp          4655274    781 4654493     0%    /tmp
rpool/root/usr          5646700 992206 4654493    18%    /usr
rpool/root/usr/local    4654524     31 4654493     0%    /usr/local
rpool/root/var          4654755    262 4654493     0%    /var
```

RootFS on ZFS

- **zpoolの操作・アップグレード時の注意**
  - 何らかの要因で zpool が認識できなくなると
    起動せず、操作できなくなってしまう危険性がある
  - システムを更新したら、必ず
    zfsboottest.sh スクリプトでチェック！
  - OKが出ていれば（たぶん）大丈夫

```
# sh /usr/src/tools/tools/zfsboottest/zfsboottest.sh
  pool: rpool
config:
       NAME  STATE
       rpool ONLINE
          gpt/disk0 ONLINE
OK
```

RootFS on ZFS

- **zpoolの操作・アップグレード時の注意**
  - 起動に使うストレージプールは、データを入れるプールと分離しておいたほうが無難。

```
# sh /usr/src/tools/tools/zfsboottest/zfsboottest.sh
  pool: rpool
config:
       NAME  STATE
       rpool ONLINE
          gpt/disk0 ONLINE
OK
```

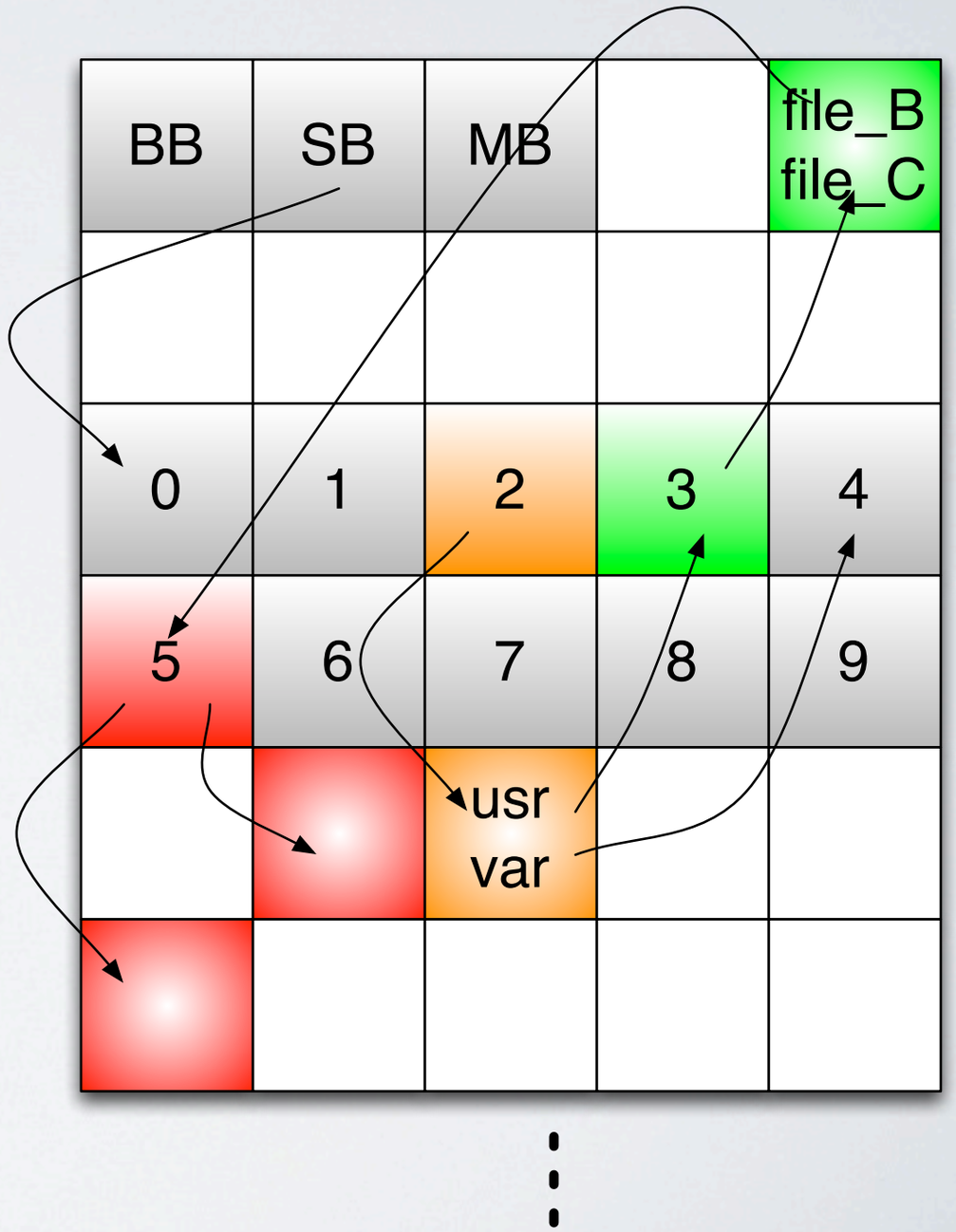
ZFSの構造と性能

- **ZFSの考え方**
  - ZPOOLで記憶装置を管理
  - ZFSデータセット1個＝ファイルシステム1個（相当）

- **dnode:**
  ZPOOL 上にZFSデータセットを構成するために
  作られる、UFS の inode に似たリンク用の情報単位
- **DMU(Data Management Unit)**:
  dnode の操作を担当するカーネルの部分

ZFSの構造

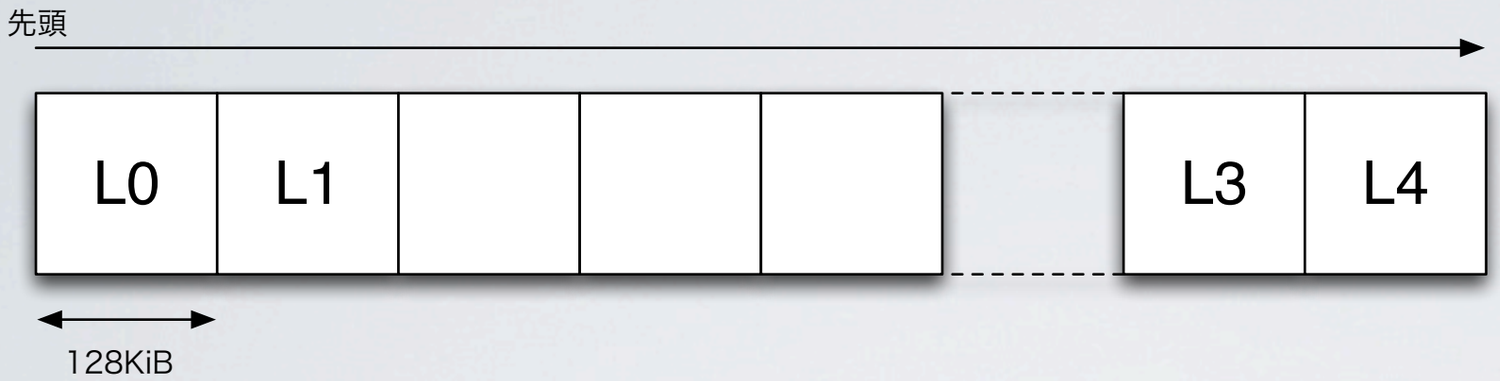
- **UFS の復習**
  - 最初はスーパーブロック。ここには inode の在処がある
  - inode はディレクトリ・ファイルの情報と、データブロックの場所を記録しているところ
  - inode は配列状に並んでいる。ルートディレクトリの場所だけ、予め決まっている
  - あとはとにかくリンクを辿る



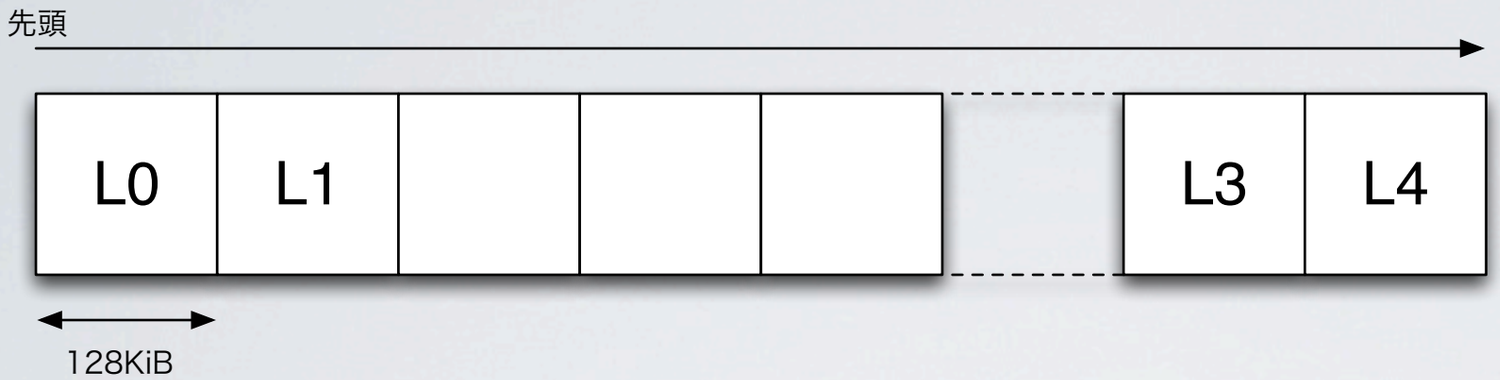
ZFSとUFSの構造の違い

- ストレージプール
  ＝ 記憶装置を管理
- ZFSデータセット1個
  ＝ファイルシステム1個
  = dnode の集合体（厳密にはちょっと違うけれど...）
- dnodeには種類がある
  - ZAPオブジェクト（name=value を格納する）
  - DSLデータセットオブジェクト（ファイルシステムを指す）
  - DSLディレクトリオブジェクト（ファイルシステムの集合体を指す）
  - inode がデータブロックにディレクトリ情報を入れていたのと同じく、dnode も役割を分けていろいろな機能を持たせている

ZFSの構造



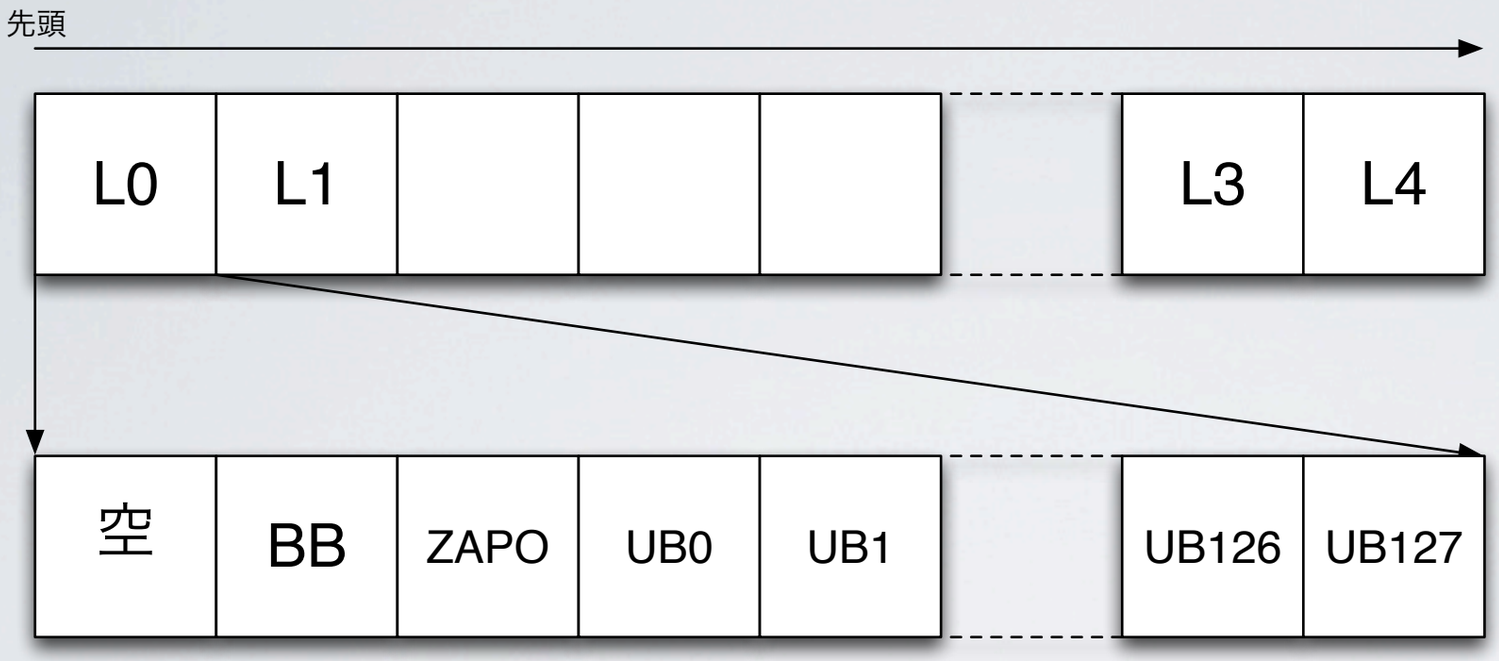
ZFSの構造



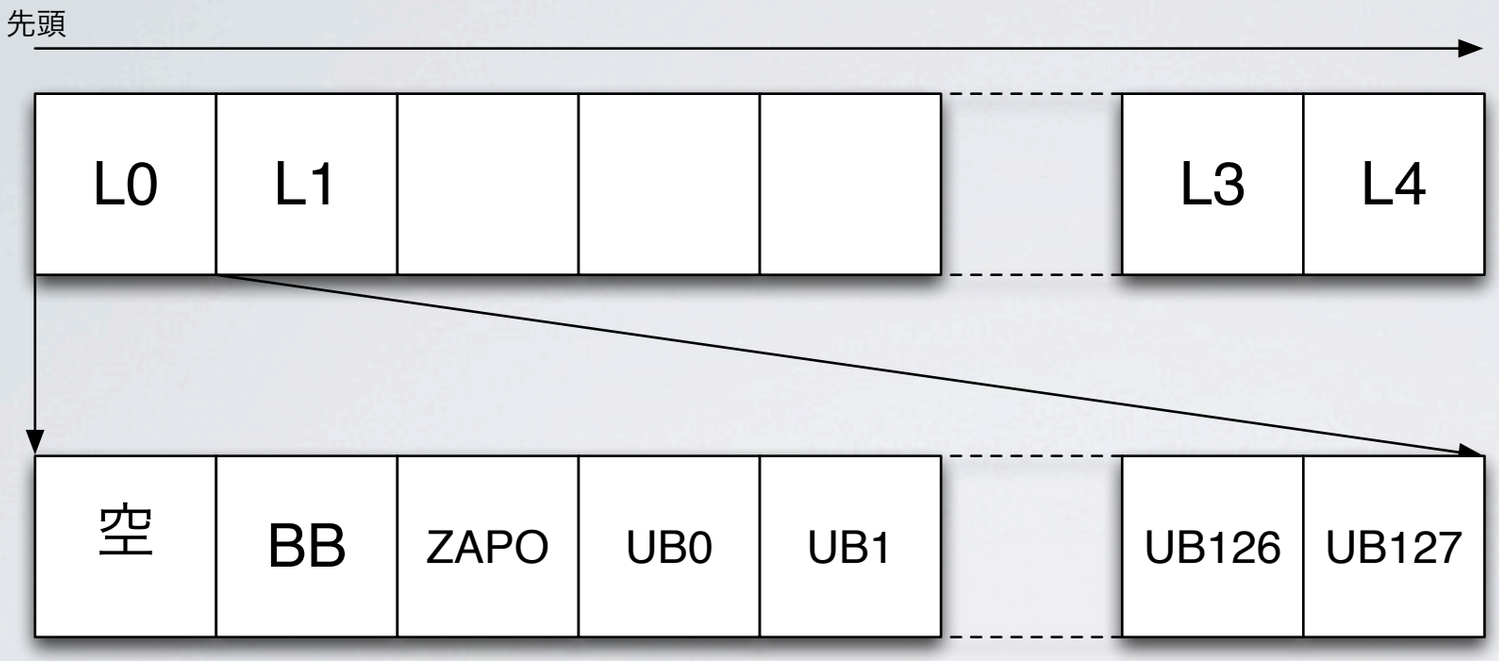
VDEVラベル（128KiB, 4個のコピー）

→ UFS の スーパブロックに相当

ZFSの構造



ZFSの構造



ラベルの中身は、

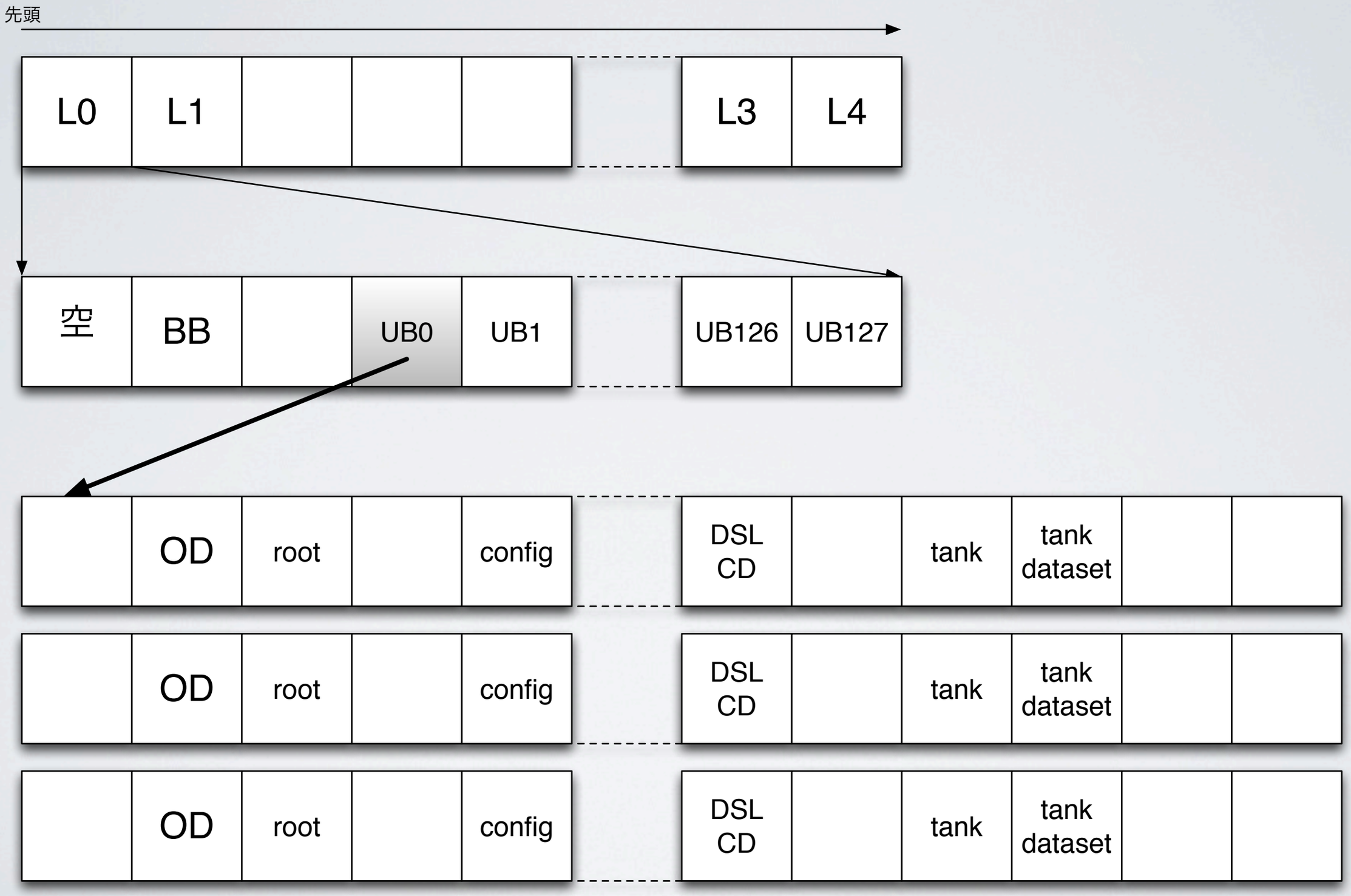
①起動ブロック

②ZAPオブジェクト

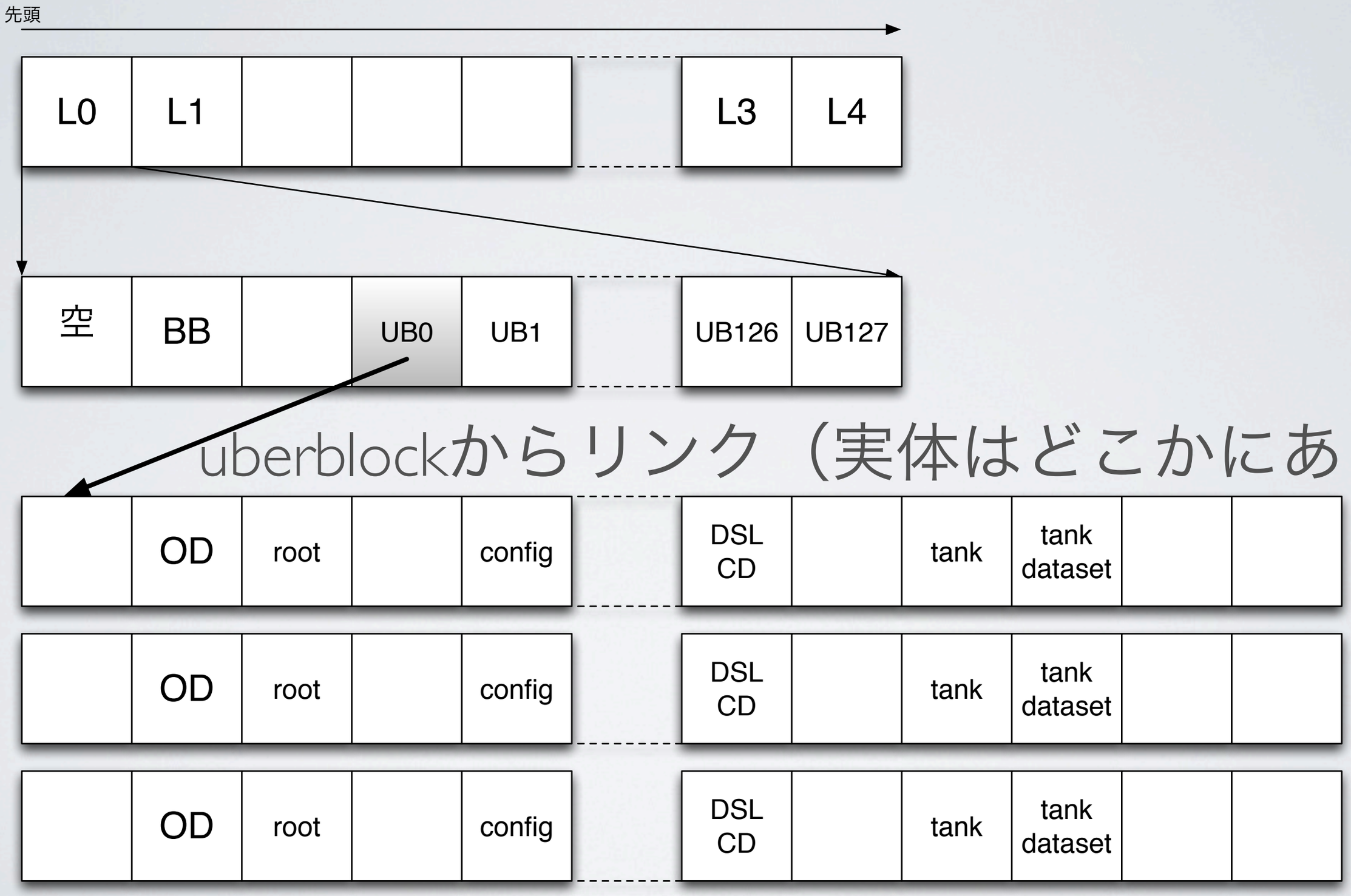
　(name=value を保存できる領域)

③uberblockの配列

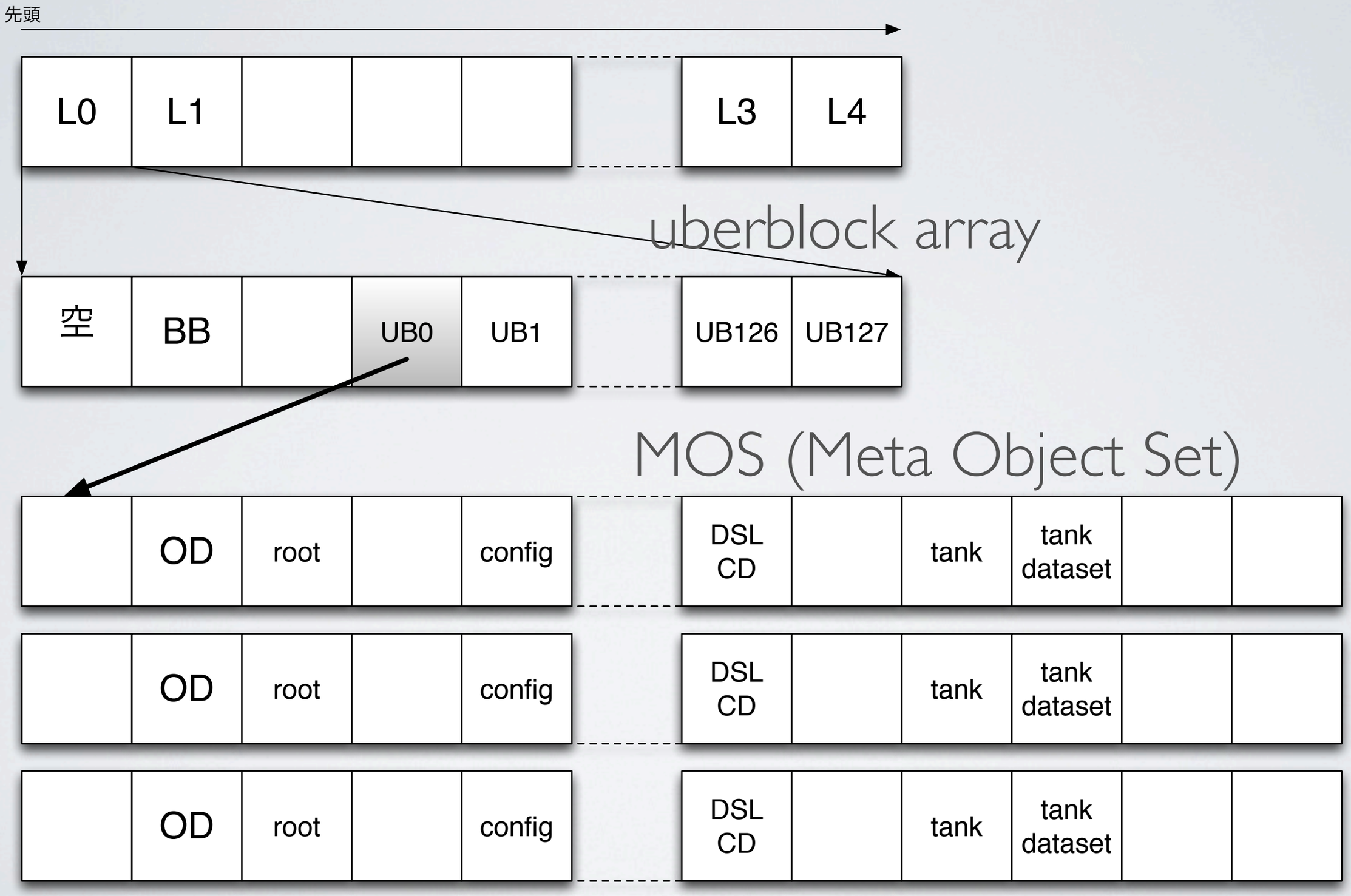
ZFSの構造


先頭
L0
L1
L3
L4
空
BB
UB0
UB1
UB126
UB127
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset

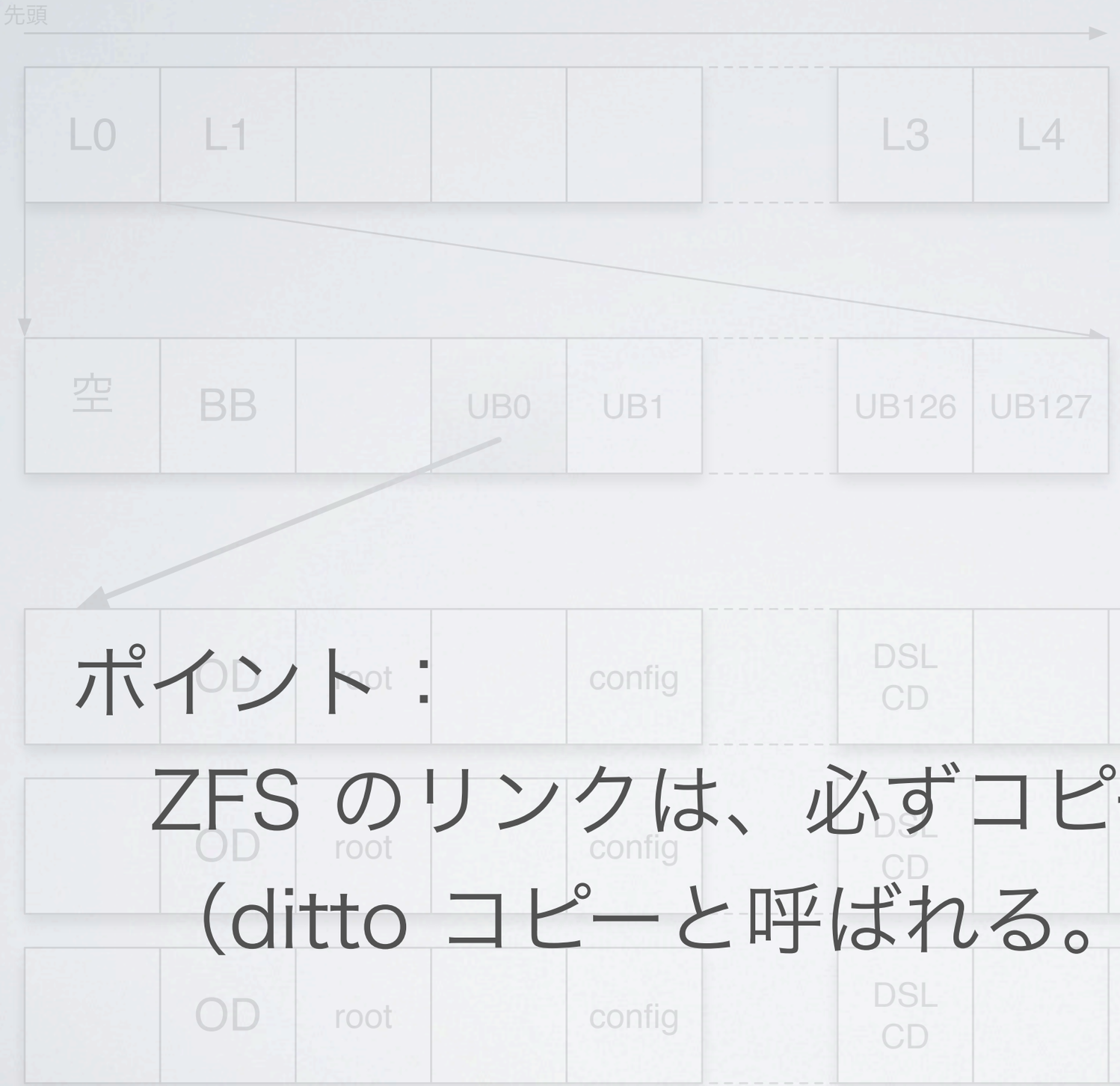
ZFSの構造


先頭
L0
L1
L3
L4
空
BB
UB0
UB1
UB126
UB127
uberblockからリンク（実体はどこかにある）
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset

ZFSの構造


先頭
L0
L1
L3
L4
uberblock array
空
BB
UB0
UB1
UB126
UB127
MOS (Meta Object Set)
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset

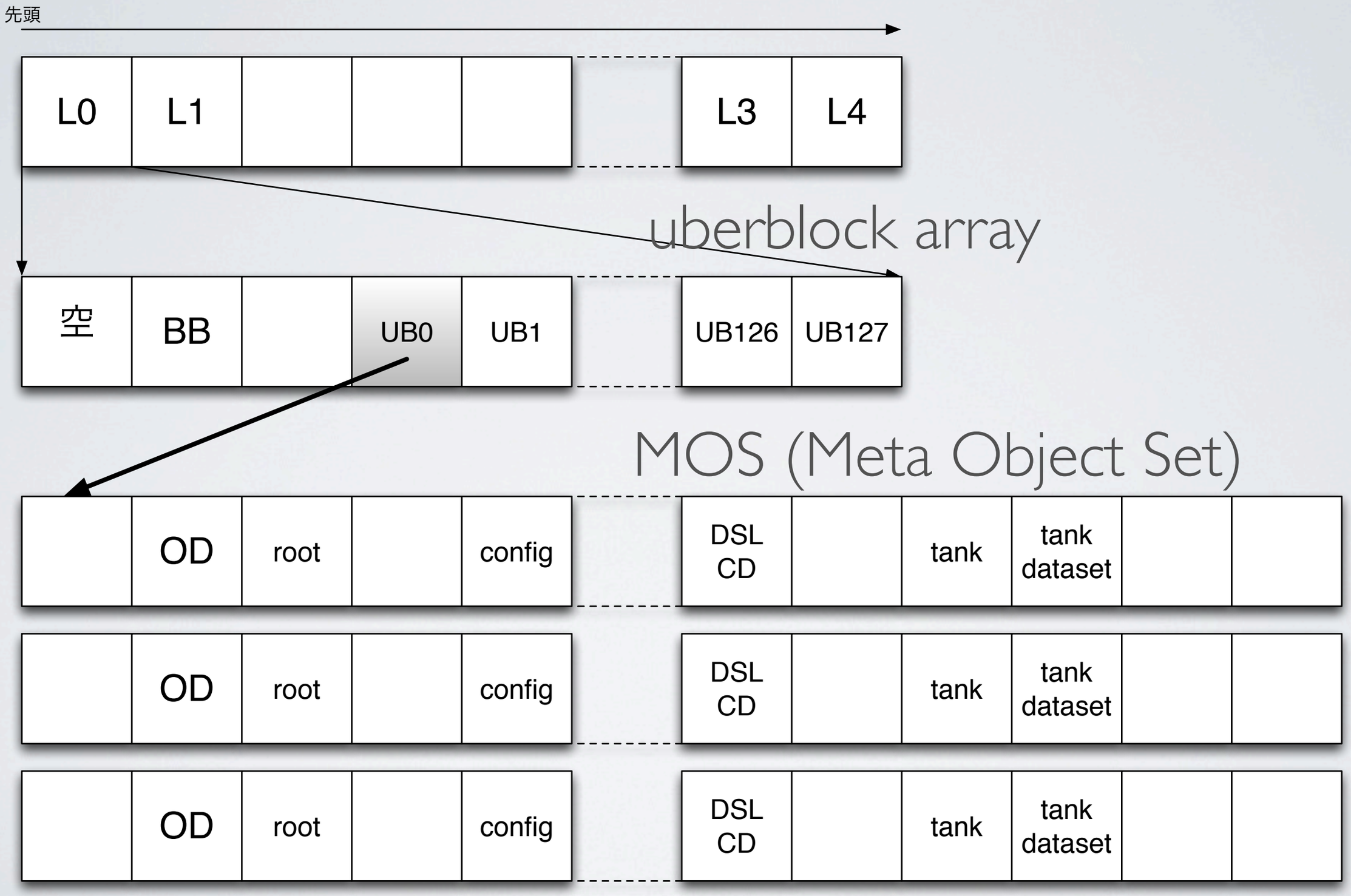
ZFSの構造



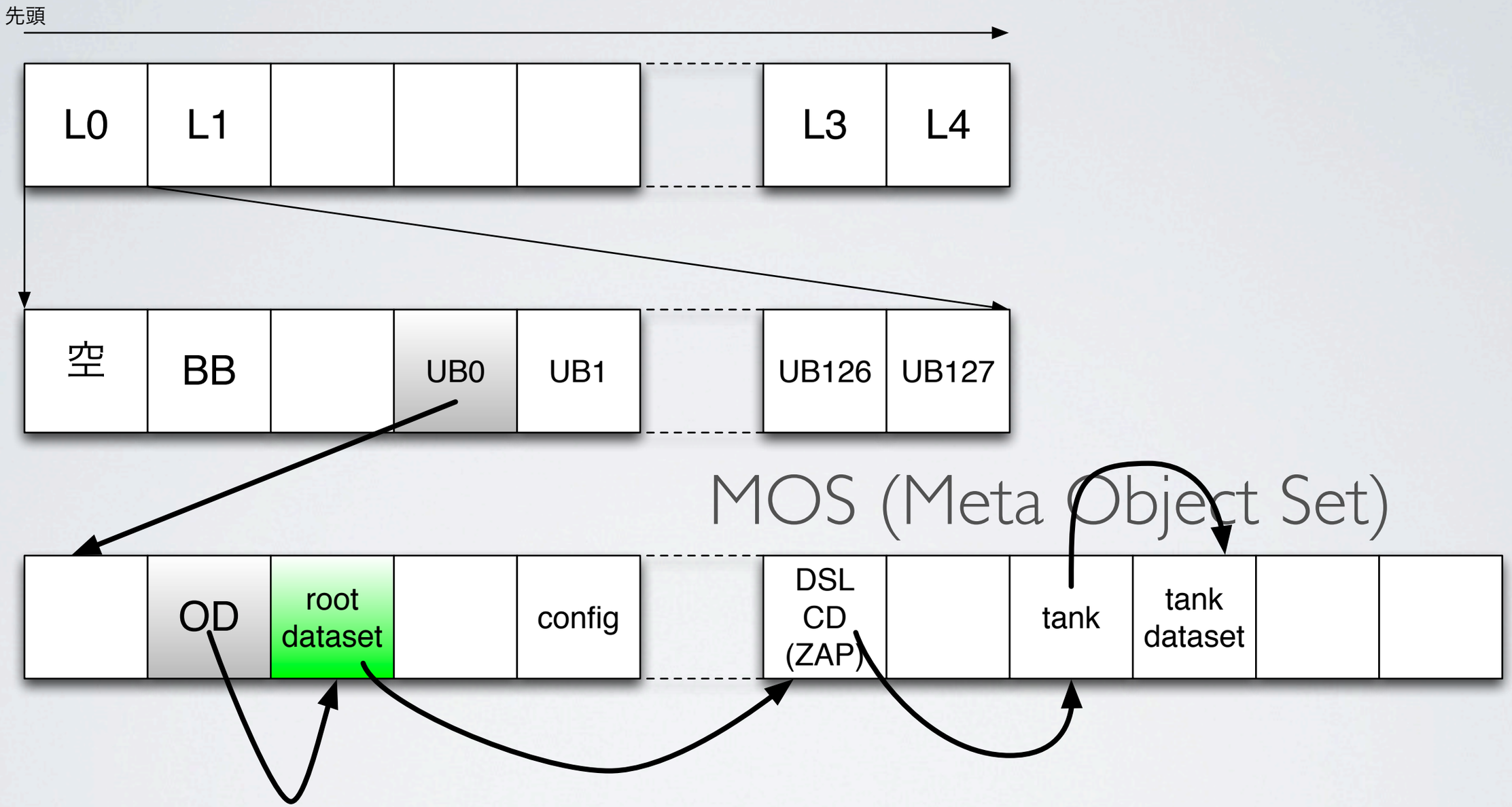
ポイント：

ZFS のリンクは、必ずコピーが作成される
（ditto コピーと呼ばれる。最大3個）

ZFSの構造


先頭
L0
L1
L3
L4
uberblock array
空
BB
UB0
UB1
UB126
UB127
MOS (Meta Object Set)
OD
root
config
DSL CD
tank
tank dataset
OD
root
config
DSL CD
tank
tank dataset
OD
root
config
DSL CD
tank
tank dataset

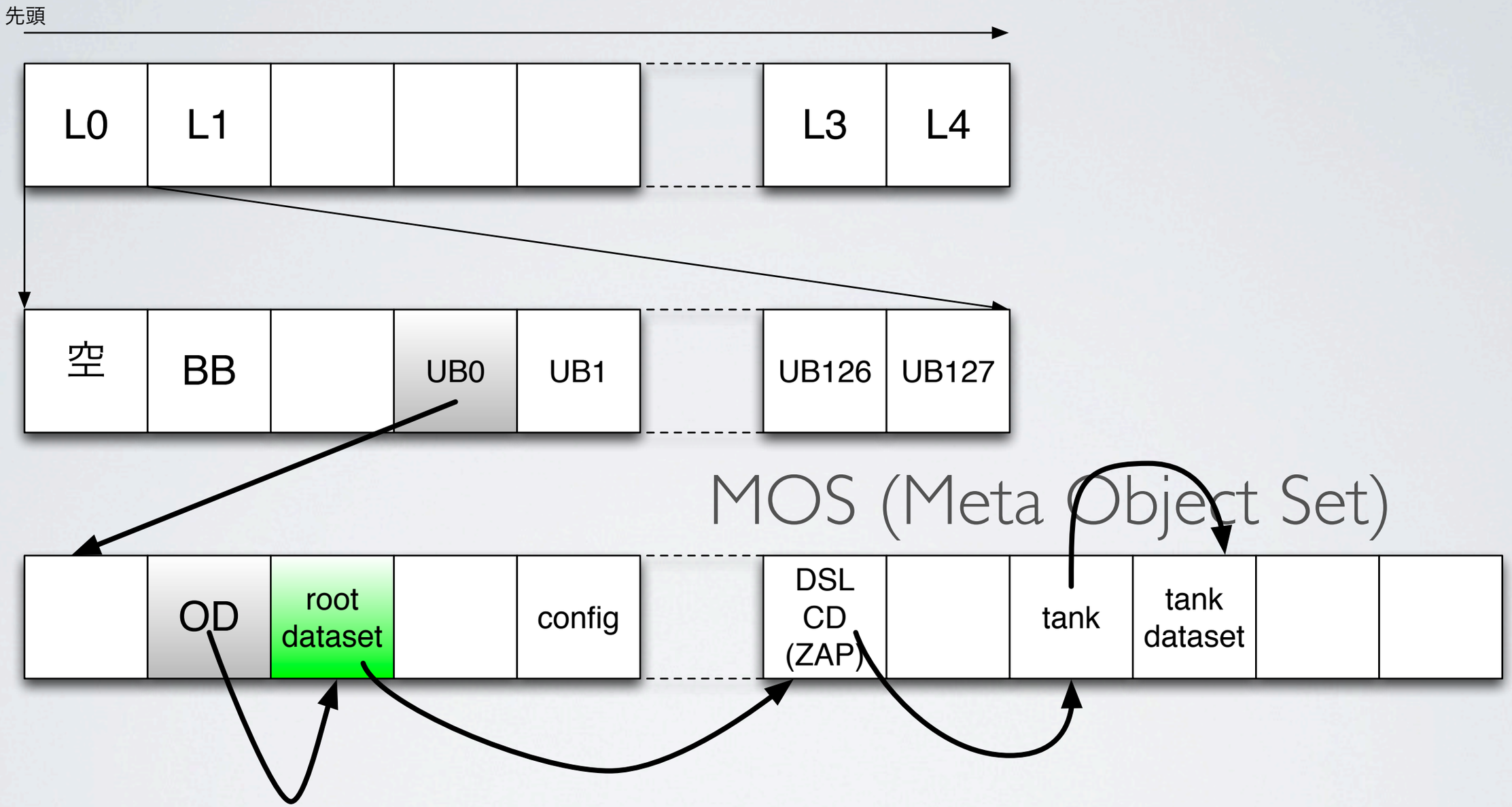
ZFSの構造



MOSの実体は、dnode が並んでいる配列構造

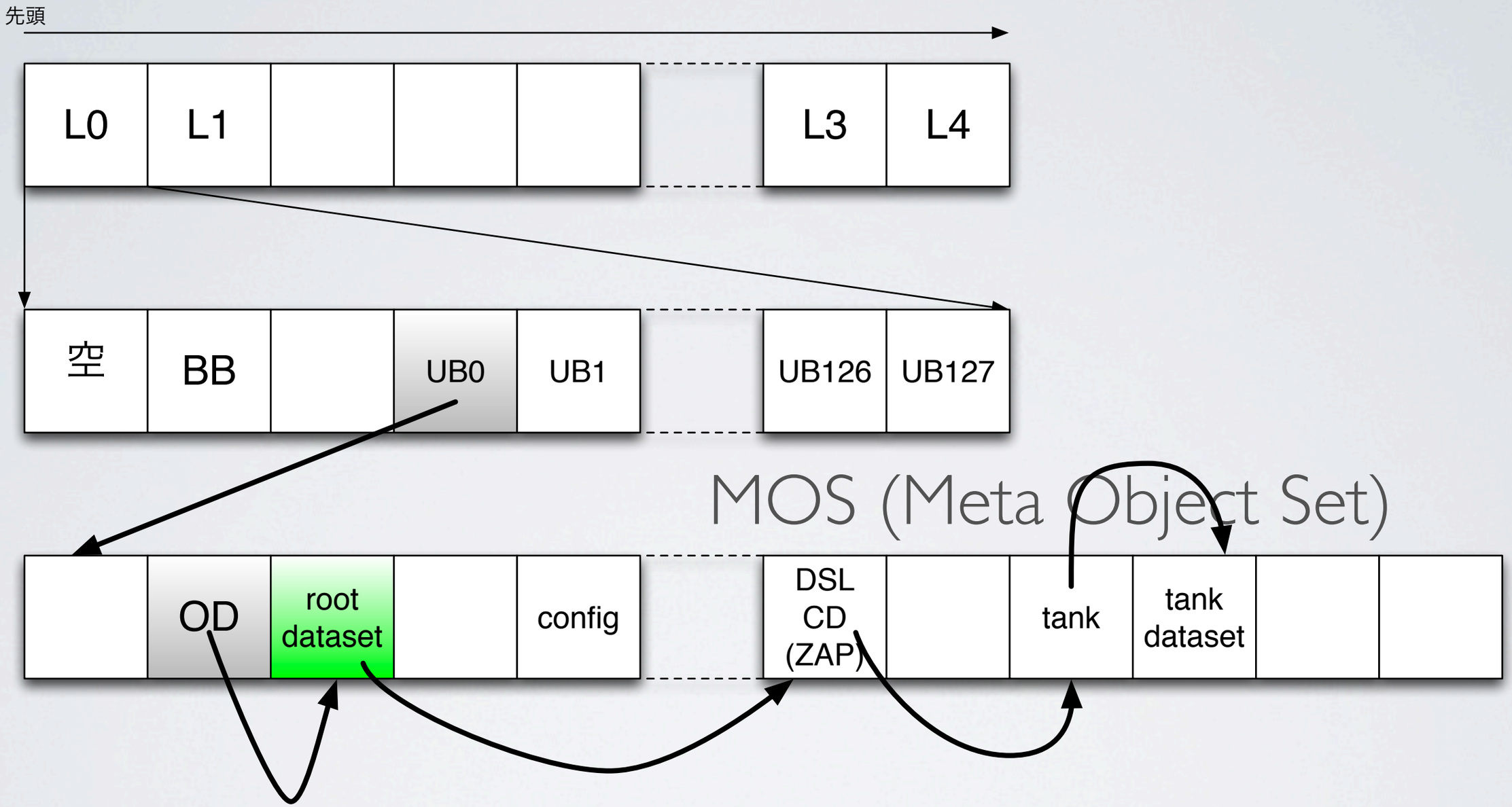
（＝UFSで言うinode領域と同じ）

ZFSの構造



ただし、MOSにはZPOOLに含まれるファイルではなく、
ZFSデータセット（＝ファイルシステム）の情報が格納される

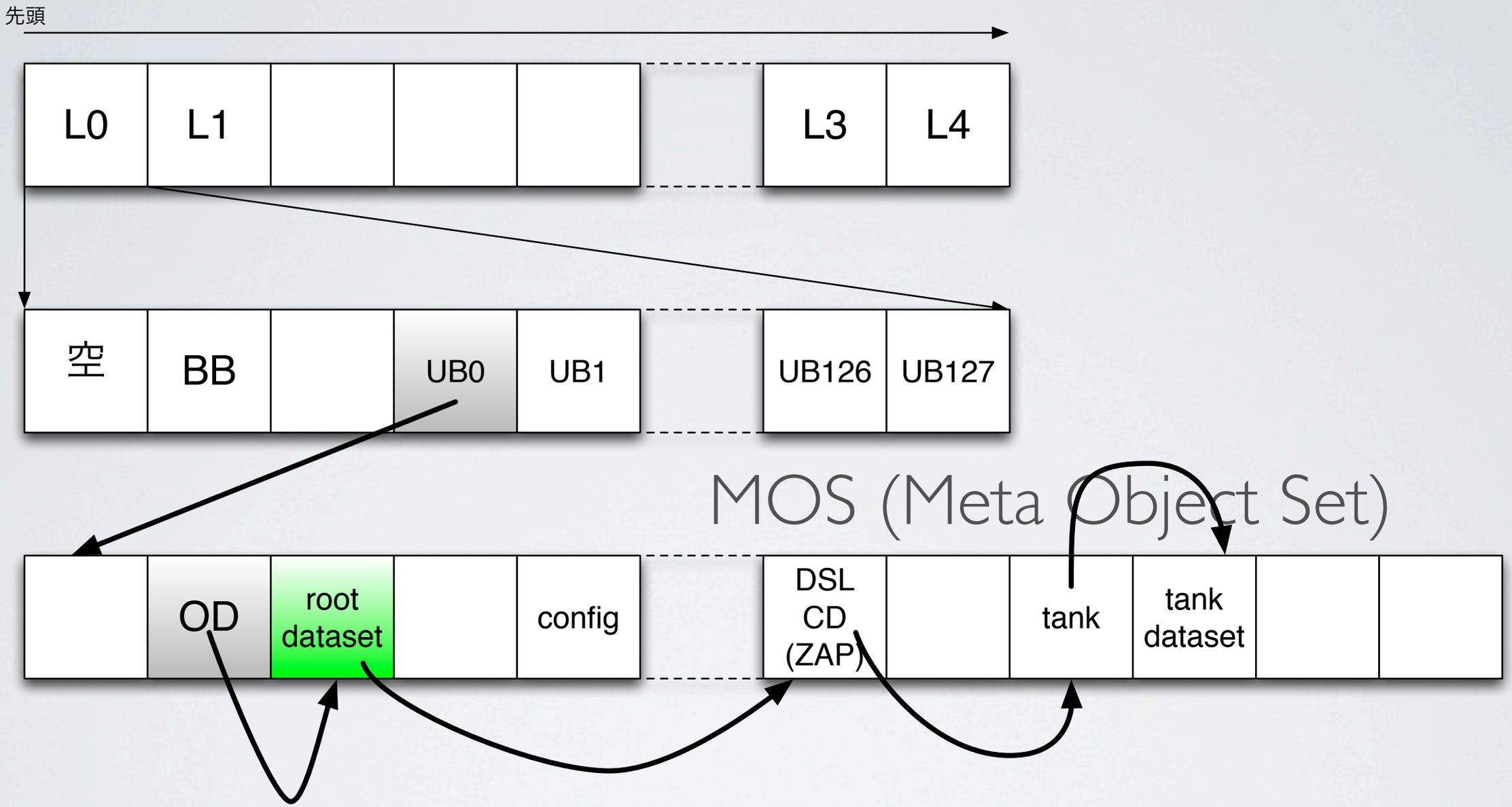
ZFSの構造



ルートデータセット：ZPOOLに必ずあるファイルシステム

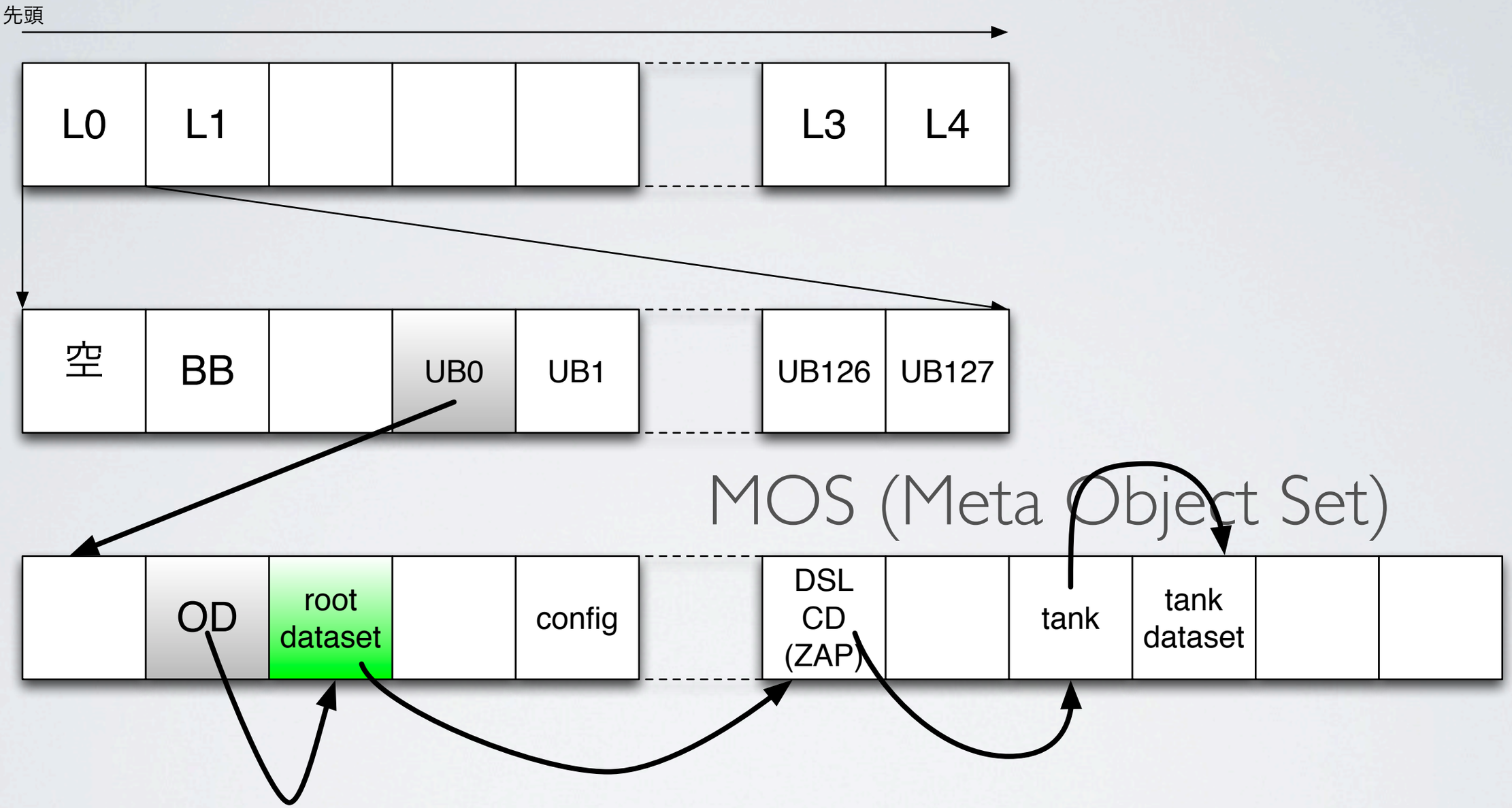
（ちなみにdnode 番号は必ず２）

ZFSの構造



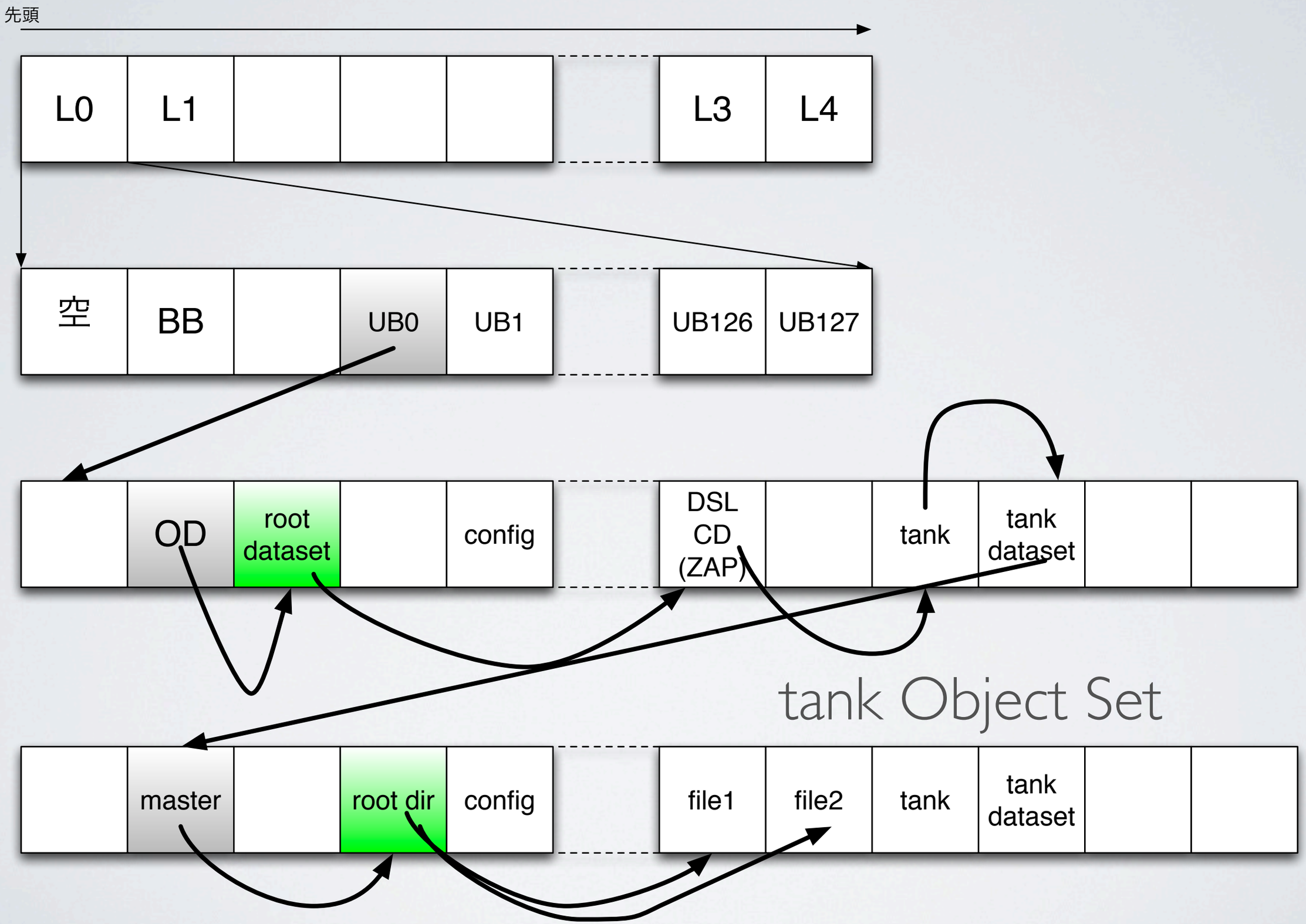
tank という名前のデータセットが登録されている

ZFSの構造

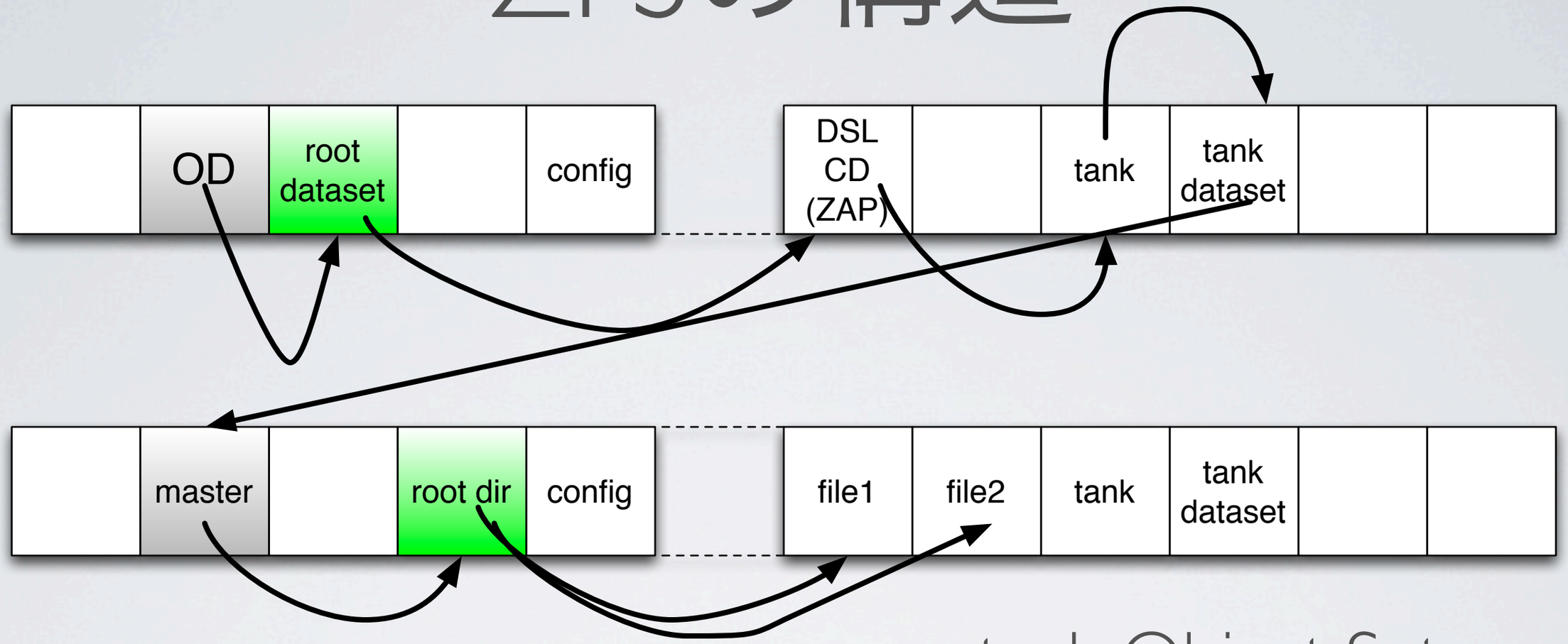


tank dataset にデータセットの中身の情報が。

ZFSの構造


先頭
L0
L1
L3
L4
空
BB
UB0
UB1
UB126
UB127
OD
root dataset
config
DSL CD (ZAP)
tank
tank dataset
tank Object Set
master
root dir
config
file1
file2
tank
tank dataset

ZFSの構造



tank Object Set

tank dataset は、もうひとつのdnode 配列を指している。

このdnode配列は、UFSのinodeとほぼ同じ機能

= tank dataset がスーパーノード

= object set 配列が inode (ルートディレクトリはdnode=3)

ZFSの構造

- **つまり**
  - ファイルシステムを inode っぽい構造で管理
  - ファイルとディレクトリを、inode っぽい構造で管理

という2段構えになっている

（簡単のため中間ブロックをいくつか飛ばして説明しています）

ZFSの構造

- **つまり**
  - ファイルシステムを inode っぽい構造で管理
  - ファイルとディレクトリを、inode っぽい構造で管理

という2段構えになっている

（簡単のため中間ブロックをいくつか飛ばして説明しています）

- **データセット**＝ファイルシステム（だけではないが）相当
- **uberblock** =データセットのスーパブロック
- **データセットオブジェクト**
  ＝ファイルシステムのスーパブロック

ZFSの構造

- **つまり**
  - ファイルシステムを inode っぽい構造で管理
  - ファイルとディレクトリを、inode っぽい構造で管理

という2段構えになっている

（簡単のため中間ブロックをいくつか飛ばして説明しています）

- **データセット**＝ファイルシステム（だけではないが）相当
- **uberblock** =データセットのスーパブロック
- **データセットオブジェクト**
  ＝ファイルシステムのスーパブロック
- 発想的には、今までのファイルシステムの技術を
  素直に拡張した形。**複数の記憶装置にまたがって構成可能**

ZFSのデータ処理

- **ブロックの処理**
  - ブロックへのリンクは、最大3重コピー
  - ブロックのチェックサムを記録できる
  - ブロックが壊れていたら、正常なコピーを使う

ZFSのデータ処理

- **ブロックの処理**
  - ブロックへのリンクは、最大3重コピー
  - ブロックのチェックサムを記録できる
  - ブロックが壊れていたら、正常なコピーを使う
- **書き込みは copy-on-write + トランザクション**
  - 書き換えないでコピーする
  - リンクは下から書き換える（SoftUpdates と同じ）
  - 一番上は uberblock をずらす
  - I/O をトランザクショングループ(txg)に分けてコミット

ZFSの性能とチューニング

- **ZFS の処理の特徴的な部分**
  - UFS以上にリンクを辿りまくる
    ＝dnode アクセスが性能に敏感なのは同じ。
      複数台のディスクに分散させることでIOPSを抑える
  - 書き込みは読み込みが付随する(COW)
    ＝IOPSが読み込みによって制限されることがある

ZFSの性能とチューニング

- **ZFS の処理の特徴的な部分**
  - UFS以上にリンクを辿りまくる
    ＝dnode アクセスが性能に敏感なのは同じ。
    複数台のディスクに分散させることでIOPSを抑える
  - 書き込みは読み込みが付随する(COW)
    ＝IOPSが読み込みによって制限されることがある

  - どちらもキャッシュヒット率が高いと緩和される。
  - キャッシュヒット率を上げる努力をしないと、
    UFS より遅い結果を招くことが多々ある

ZFSの性能とチューニング

- **一般的なベンチマークが役に立つか？**
  - 意図的にキャッシュを回避するようなベンチマークは
    実際のワークロードと大きく異なるため、役に立たない
- 例：
  - da1 da2 da3 の 3 台のストライププール
  - da1 da2 da3 の 3 台の gstripe でつくったRAID0領域
  - 単体で 150 IOPS のHDDの時、IOPS はどっちが良い？

ZFSの性能とチューニング

- **一般的なベンチマークが役に立つか？**
  - 意図的にキャッシュを回避するようなベンチマークは
    実際のワークロードと大きく異なるため、役に立たない
- 例：
  - da1 da2 da3 の 3 台のストライププール
  - da1 da2 da3 の 3 台の gstripe でつくったRAID0領域
  - 単体で 150 IOPS のHDDの時、IOPS はどっちが良い？
  - 答：gstripe は書き込みが 400 IOPS 前後出るが、
    ZFS は読み書きともに150 IOPSしか出ない

ZFSの性能とチューニング

- 一般的なベンチマークが役に立つか？
  - 意図的にキャッシュを回避するようなベンチマークは実際のワークロードと大きく異なるため、役に立たない
  - 例：
    - da1 da2 da3 の 3 台のストライププール
    - da1 da2 da3 の 3 台の gstripe でつくったRAID0領域
    - 単体で 150 IOPS のHDDの時、IOPS はどっちが良い？
    - 答：gstripe は書き込みが 400 IOPS 前後出るが、ZFS は読み書きともに150 IOPSしか出ない

ZFSは、キャッシュが効かない
完全ランダムな読出アクセスに対して
性能が高く出ることはない

＝　単純に他のファイルシステムより
踏むステップが多いから

ZFSの性能とチューニング

- **もう一つの地味な落とし穴**
  - 空き容量が 20% を切ると性能が低下しやすくなる
  - 容量が足りなくなってきたら、早めに増やすしかない
  - 0% まで使い切ると、奇妙な動作をすることがある
    - ファイルが消せないとか

チューニング

- 1）atime
- 2）圧縮
- 3）dedup
- 4）send/recv
- 5）ワークロード：ウェブサーバ
- 6）ワークロード：データベース
- 7）ワークロード：ファイルサーバ
- 8）ARC, ZIL, zfetch, vdev
- 9）txg
- 10) 性能モニタリング

プロパティの操作

```
# zpool get <プロパティ名> <プール名>
# zpool get all <プール名>                    (すべてを列挙)
# zpool set <プロパティ名>=<値> <プール名>

# zfs get <プロパティ名> <プール名>
# zfs get all <プール名>                      (すべてを列挙)
# zfs get -r <プロパティ名> <プール名>        (再帰的に下位階層を全部)
# zfs set <プロパティ名>=<値> <プール名>
```

- ストレージプールプロパティ
- データセットプロパティ
- それぞれ対象となるものが違うが、操作は似ている

atime

```
# zfs get -r atime rpool
NAME                  PROPERTY  VALUE  SOURCE
rpool                 atime     on     default
rpool/root            atime     on     default
rpool/root/tmp        atime     on     default
rpool/root/usr        atime     on     default
rpool/root/usr/local  atime     on     default
rpool/root/var        atime     on     default
```

- アクセス時刻を記録するかどうか
  - データセットプロパティ atime で設定
  - onで有効、offで無効
- ZFSの書き込みはCOWなので、
  書き込み時にも読み出しが必ず発生する。
- 読み出し負荷が高いケースで atime は性能劣化要因
- 書き込まなくてもスナップショットの容量が増えて行く

圧縮

```
# zfs get -r compression rpool
NAME                  PROPERTY     VALUE     SOURCE
rpool                 compression  off       default
rpool/root            compression  off       default
rpool/root/tmp        compression  off       default
rpool/root/usr        compression  off       default
rpool/root/usr/local  compression  off       default
rpool/root/var        compression  off       default
```

- データを圧縮するかどうか
  - データセットプロパティ compression で設定
  - offで無効、gzip, lzjb で有効。
- ログなどテキストファイル主体の場合に有効
- CPU 負荷を調べながら判断する必要あり
- illumos では lz4 というアルゴリズムの追加が検討中

dedup

```
# zfs get -r dedup rpool
NAME                  PROPERTY  VALUE          SOURCE
rpool                 dedup     off            default
rpool/root            dedup     off            default
rpool/root/tmp        dedup     off            default
rpool/root/usr        dedup     off            default
rpool/root/usr/local  dedup     off            default
rpool/root/var        dedup     off            default
```

- 重複データをまとめて、記録領域を節約するかどうか
  - データセットプロパティ dedup で設定
  - on で有効、offで無効

- 絶対に有効にしないこと！

send/recv

- パイプを流れるデータに対して、バッファリングすることをおすすめ。
- やり方は資料の前のページを参照のこと。

ワークロード：ウェブサーバ

- 小さい多数のファイルを読み出し、ネットワークに投げる負荷
- メタデータのキャッシュヒット率が低下していないか
  チェック。vfs.zfs.arc_meta_limit が少なければ上げる。
- 現在の FreeBSD 実装では、ZFS上のファイルに sendfile や mmap を使うとキャッシュを二重にとる可能性がある
  - Apache: EnableMMAP, EnableSendfile で無効に
  - Nginx: sendfile off で無効に
  - Lighthttpd: server.network-backend=”writev” で無効に

ワークロード：データベース

- 固定長レコードへの読み書きが発生する負荷

- ZFS のデフォルトレコード長は 128kB
  ＝ 書き込みのCOW処理が 128kB より小さい単位で性能低下
- データセットプロパティ recordsize を使って
  データベースに合わせる
- 例：
  PostgreSQL: recordsize=8k
  MySQL:　MyISAM = 8k
  　　InnoDB = 16k

ワークロード：ファイルサーバ

- ファイル数が多く、継続的にアクセスが行われる負荷

- スナップショット数を減らす
- atime を無効に
- 書き込みが多ければ、ZIL を SSD に分離
- それ以外は基本的にウェブサーバと変わらない。
  メタデータのキャッシュヒット率はこちらも重要

ARC, ZIL, zfetch, vdev

- **ARC: ZFS の要。**
  - デフォルト：(搭載メモリ量-1GB)
  - メタデータ用ARCのデフォルト：その1/4
- 多ければ多いほど良いが、統計データを見て判断
- **L2ARC: ARC の二次キャッシュ版。SSD等が想定。**
  - 読み込み負荷に対する効果がある（COWを思い出そう）
  - zpool add pool cache da10
- **L2ARC プリフェッチ：キャッシュを埋める速度を**
  向上させるための処理。（デフォルトoff)

  vfs.zfs.l2arc_write_max
  vfs.zfs.l2arc_write_boost
  vfs.zfs.l2arc_noprefetch

ARC, ZIL, zfetch, vdev

- **ZIL: intent log**
  - POSIX が定めている fsync() システムコールのため
  - 不意の電源断が発生した時に、このログをさかのぼる
  - 通常はストレージプールを構成するデバイスそれぞれの小さな領域が使われる
  - 書き込み速度を向上させるために、分離することができる
    - zpool add pool log da10
  - データセット単位で非同期にすることもできるが...
    - zfs set sync=[standard | always | disabled]

ARC, ZIL, zfetch, vdev

- **zfetch**
  - ファイルの読み込みパターンを解析して、動的に無駄な処理を排除する仕組み
  - 例）
    同じファイルを2つのプロセスが読んでいる。
    ＝読み出しアクセスの内部処理を1個にまとめる
    （いわゆるストライドパターンのアクセス最適化）
  - 実質的なI/Oが減少するため、アクセス遅延が小さくなる
  - vfs.zfs.zfetch の設定項目があるが、パラメータは複雑なので、ヒット率を監視しておくと良い

ARC, ZIL, zfetch, vdev

- **vdev cache**
  - ブロックデバイスのアクセス時に、
    シーク時間を考慮したキャッシュを行う（先読みに近い）
  - 非常にシーク遅延が大きい記憶装置にのみ有効
    - 得られるのはシークが減る、という効果だけ
  - デフォルトでは OFF（最近のデバイスは十分に速い...）
  - vfs.zfs.vdev.cache.size が接続台数分だけ
    確保されてしまうので、0 に設定すること

txg

- **トランザクショングループ**
  - ZFSの書き込みは、次のような特徴がある
    - COWがあるので、なるべくひとまとまりで処理
    - 何秒待つか？vfs.zfs.txg.timeout: 5
    - デバイスにどれだけ仕事を与えるか?
      vfs.zfs.vdev.min_pending: 4
      vfs.zfs.vdev.max_pending: 10
    - 書き込みが連続で発生しないため、
      ベンチマークでぼろぼろの結果に見えることがある
    - 瞬間的な性能ではなく、平均的な性能に最適化している

txg

- **トランザクショングループ**
  - vfs.zfs.txg.timeout=1: 連続で書き込むようになる
    - デメリット
      - フラグメンテーションが起こりやすくなる
        - 書き込みのグルーピングが弱いので、
          アクセスのランダム性があがる＝ZFSの弱点を突く
  - 大きいファイルのシーケンシャルアクセスに対しては
    短い方が良い。
  - 多数の小さいファイルへのアクセスに対しては、長めの方
    が良い。
  - 統計データを見ること！

統計データの調べ方

- **zfs-mon と zfs-stats**
  - sysutils/zfs-stats にある
  - zfs-mon: リアルタイムのキャッシュヒット率表示
  - zfs-stats: 各種統計情報を分かりやすい形で一覧表示

zfs-mon

```
% zfs-mon -a
ZFS real-time cache activity monitor
Seconds elapsed:  26

Cache hits and misses:
                                1s    10s    60s    tot
                      ARC hits:  2812   2768   3241   3241
                    ARC misses:     1     19     14     14
          ARC demand data hits:     7      5      3      3
        ARC demand data misses:     0      0      0      0
      ARC demand metadata hits:  2805   2675   3190   3190
    ARC demand metadata misses:     1      1      1      1
      ARC prefetch data hits:     0      2      8      8
    ARC prefetch data misses:     0      3      4      4
   ARC prefetch metadata hits:     0     86     40     40
 ARC prefetch metadata misses:     0     14      8      8
                   L2ARC hits:     1      1      1      1
                 L2ARC misses:     0     18     12     12
                  ZFETCH hits:  3701   3775   3484   3484
                ZFETCH misses:  3668   3472   4326   4326

Cache efficiency percentage:
                          10s    60s    tot
                 ARC:  99.32  99.57  99.57
       ARC demand data: 100.00 100.00 100.00
   ARC demand metadata:  99.96  99.97  99.97
     ARC prefetch data:  40.00  66.67  66.67
ARC prefetch metadata:  86.00  83.33  83.33
                L2ARC:   5.26   7.69   7.69
               ZFETCH:  52.09  44.61  44.61
```

allbsd.org の NFS サーバ

- Supermicro X8DTL, 24GB RAM, 2TB x 12 本
- ZIL, L2ARC は SSD

```
hrs@pool % zpool list -v
NAME          SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
a            18.1T  6.97T  11.2T    38%  1.00x  ONLINE  -
  raidz2     9.06T  3.49T  5.58T          -
    da2p2        -      -      -          -
    da3p2        -      -      -          -
    da4p2        -      -      -          -
    da5p2        -      -      -          -
    da6p2        -      -      -          -
  raidz2     9.06T  3.49T  5.58T          -
    da7p2        -      -      -          -
    da8p2        -      -      -          -
    da9p2        -      -      -          -
    da10p2       -      -      -          -
    da11p2       -      -      -          -
  ada0p2     15.9G  74.8M  15.8G          -
cache            -      -      -      -      -      -
  ada0p1     64.0G  64.0G     7M          -
```

allbsd.org の NFS サーバ

- Supermicro X8DTL, 24GB RAM, 2TB x 12 本
- ZIL, L2ARC は SSD

```
hrs@pool % zfs list
NAME           USED  AVAIL  REFER  MOUNTPOINT
a             4.15T  6.47T  50.0K  none
a/a           4.15T  6.47T  74.5G  /a
a/a/builder   4.58G  6.47T  4.58G  /a/builder
a/a/cvsroot   16.7G  6.47T  16.7G  /a/cvsroot
a/a/ftproot   3.95T  6.47T  3.95T  /a/ftproot
a/a/htdocs    7.82G  6.47T  7.82G  /a/htdocs
a/a/rootfs    56.8G  6.47T  56.8G  /a/rootfs
a/a/svnroot   44.8G  6.47T  44.8G  /a/svnroot
```

allbsd.org の NFS サーバ

- 圧縮：テキスト主体の CVS ミラーと WWW ミラーは x3 程度その他は x 1.5 に行くか行かないか。

```
a/a/cvsroot  compressratio         3.20x                     -
a/a/cvsroot  compression           gzip                      local

a/a/ftproot  compressratio         1.21x                     -
a/a/ftproot  compression           lzjb                      inherited from a

a/a/htdocs   compressratio         2.70x                     -
a/a/htdocs   compression           gzip                      local

a/a/rootfs   compressratio         1.64x                     -
a/a/rootfs   compression           lzjb                      inherited from a

a/a/svnroot  compressratio         1.60x                     -
a/a/svnroot  compression           gzip                      local
```

allbsd.org の NFS サーバ

- 大量のファイルを扱うので、ARC の調整をしないと
  かなり遅い（ミラー同期の時間が2-3倍になる）

```
(ZFS を使っている NFS サーバ)
vfs.zfs.arc_min="15G"
vfs.zfs.arc_max="16G"
vfs.zfs.arc_meta_limit="15G"

vfs.zfs.vdev.cache.size="0"
vfs.zfs.vdev.min_pending=2
vfs.zfs.vdev.max_pending=4

vfs.zfs.txg.timeout=30


（NFS クライアント側）
vfs.nfs.iodmin=48
vfs.nfs.iodmax=64
vfs.nfs.prime_access_cache=1
vfs.nfs.access_cache_timeout=180
vfs.nfs.bufpackets=32
```

まとめ

- 9系の ZFS は、デフォルト設定でもそこそこ動きます
- 開発者が使っているマシンも ZFS に移行しているので、安定性の心配は（昨年に比べると）だいぶなくなりました
- チューニングはキャッシュヒット率を観察するところから。ほとんどの場合、これが低い
- ウェブの情報等は玉石混淆なので、まずはデフォルトからスタートするのが無難
- Rootfs on ZFS は、まだ設定を簡単にできるツールがありません。インストーラが対応するまで待ちましょう

お知らせ1：ABC2013

- AsiaBSDCon 2013 を、3/14-17 で開催します。
- 場所は前回と同じ、東京理科大学（JR飯田橋駅）です
- 基本的に英語での開催ですが、日本語での催しも企画しています。詳しくは http://www.asiabsdcon.org まで。

お知らせ２：ML

- FreeBSDに関する疑問や、勉強会の内容のフォローアップ等のやりとりができるメーリングリストをつくりました。
- もし興味のある方がいらっしゃったら、
  参加してみてください。
- http://lists.allbsd.org にアクセスして、
  freebsd-ja というメーリングリストに入会申請を出してください。手順に従って操作すれば、自動登録されます。

おしまい

- トラブルシュートの話は入れられなかったので、
  またどこか別の機会に...

- 質問はありますか？