Btrfs vs ZFS 实现 snapshot 的差异

目录

Contents

    - 那么快照又是如何记录的呢？
- ZFS 的文件系统（filesystem）、快照（snapshot）、克隆（clone）及其它
    - ZFS 设计中和快照相关的一些术语和概念
        - 数据集（dataset）
        - 文件系统（filesystem）
        - 快照（snapshot）
        - 克隆（clone）
        - 书签（bookmark）
        - 检查点（checkpoint）
    - ZFS 的概念与 btrfs 概念的对比
    - ZFS 中是如何存储这些数据集的呢
        - ZFS 的存储格式概况

Btrfs 和 ZFS 都是开源的写时拷贝（Copy on Write, CoW）文件系统，都提供了相似的子卷管理和 快照 (snapshot）的功能。网上有不少文章都评价 ZFS 实现 CoW FS 的创新之处，进而想说「 Btrfs 只是 Linux/GPL 阵营对 ZFS 的拙劣抄袭」。或许（在存储领域人尽皆知而在领域外）鲜有人知，在 ZFS 之前就有 NetApp 的商业产品 WAFL(Write Anywhere File Layout) 实现了 CoW 语义的文件系统，并且集成了快照和卷管理之类的功能。描述 btrfs 原型设计的 论文 和 发表幻灯片 也明显提到 WAFL 比提到 ZFS 更多一些，应该说 WAFL 这样的企业级存储方案才是 ZFS 和 btrfs 共同的灵感来源，而无论是 ZFS 还是 btrfs 在其设计中都汲取了很多来自 WAFL 的经验教训。

我一开始也带着「Btrfs 和 ZFS 都提供了类似的功能，因此两者必然有类似的设计」这样的先入观念，尝试去使用这两个文件系统，却经常撞上两者细节上的差异，导致使用时需要不尽相同的工作流，或者看似相似的用法有不太一样的性能表现，又或者一边有的功能（比如 ZFS 的 inband dedup，Btrfs 的 reflink）在另一边没有的情况。后来看到了 LWN 的这篇《A short history of btrfs》让我意识到 btrfs 和 ZFS 虽然表面功能上看起来类似，但是实现细节上完全不一样，所以需要不一样的用法，适用于不一样的使用场景。

为了更好地理解这些差异，我四处搜罗这两个文件系统的实现细节，于是有了这篇笔记，记录一下我查到的种种发现和自己的理解。~~（或许会写成一个系列？还是先别乱挖坑不填。）~~ 只是自己的笔记，所有参阅的资料文档都是二手资料，没有深挖过源码，还参杂了自己的理解，于是难免有和事实相违的地方，如有写错，还请留言纠正。

Btrfs 的子卷（subvolume）和快照（snapshot）

先从两个文件系统中（表面上看起来）比较简单的 btrfs 的子卷（subvolume）和快照（snapshot）说起。 关于子卷和快照的常规用法、推荐布局之类的话题就不细说了，网上能找到很多不错的资料，比如 btrfs wiki 的 SysadminGuide 页 和 Arch wiki 上 Btrfs#Subvolumes 页都有不错的参考价值。

关于写时拷贝（CoW）文件系统的优势，我们为什么要用 btrfs/zfs 这样的写时拷贝文件系统， 而不是传统的文件系统设计，或者写时拷贝文件系统在使用时有什么区别之类的，网上同样也能找到很多介绍 ，这里不想再讨论。这里假设你用过 btrfs/zfs 至少一个的快照功能，知道它该怎么用， 并且想知道更多细节，判断怎么用那些功能才合理。

子卷（subvolume）和快照（snapshot）的术语

在 btrfs 中，存在于存储媒介中的只有「子卷」的概念，「快照」只是个创建「子卷」的方式， 换句话说在 btrfs 的术语里，子卷（subvolume）是个名词，而快照（snapshot）是个动词。 如果脱离了 btrfs 术语的上下文，或者不精确地称呼的时候，也经常有文档把 btrfs 的快照命令创建出的子卷叫做一个快照，所以当提到快照的时候，根据上下文判断这里是个动词还是名词， 把名词的快照当作用快照命令创建出的子卷就可以了。或者我们可以理解为， **互相共享一部分元数据**

（metadata）的子卷互为彼此的快照（名词），那么按照这个定义的话，在 btrfs 中创建快照（名词）的方式其实有两种：

1. 用 `btrfs subvolume snapshot` 命令创建快照
2. 用 `btrfs send` 命令并使用 `-p` 参数发送快照，并在管道另一端接收

`btrfs send` 命令的 `-p` 与 `-c`

这里也顺便提一下 `btrfs send` 命令的 `-p` 参数和 `-c` 参数的差异。只看 btrfs-send(8) 的描述的话：

> -p <parent>
>
> send an incremental stream from parent to subvol
>
> -c <clone-src>
>
> use this snapshot as a clone source for an incremental send (multiple allowed)

看起来这两个都可以用来生成两个快照之间的差分，只不过 -p 只能指定一个「parent」， 而 -c 能指定多个「clone source」。在 unix stackexchange 上有人写明了这两个的异同 。使用 -p 的时候，产生的差分首先让接收端用 subvolume snapshot 命令对 parent 子卷创建一个快照， 然后发送指令将这个快照修改成目标子卷的样子，而使用 -c 的时候，首先在接收端用 subvolume create 创建一个空的子卷，随后发送指令在这个子卷中填充内容，其数据块尽量共享 clone source 已有的数据。 所以 `btrfs send -p` 在接收端产生是有共享元数据的快照，而 `btrfs send -c` 在接收端产生的是仅仅共享数据而不共享元数据的子卷。

定义中「互相共享一部分 **元数据** 」比较重要，因为除了快照的方式之外， btrfs 的子卷间也可以通过 reflink 的形式共享数据块。我们可以对一整个子卷（甚至目录）执行 `cp -r --reflink=always` ，创建出一个副本，副本的文件内容通过 reflink 共享原本的数据，但不共享元数据，这样创建出的就不是快照。

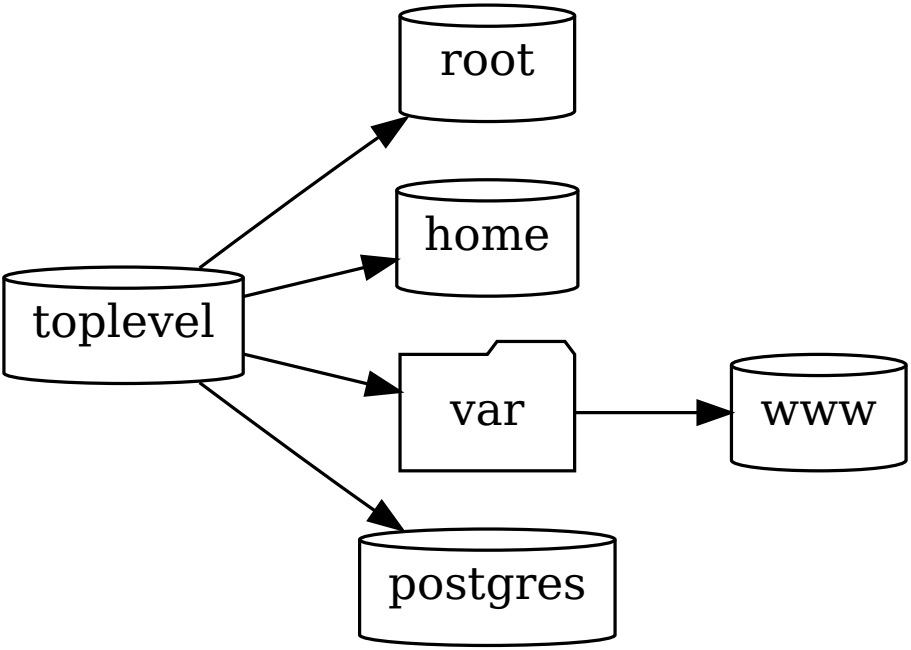
说了这么多，其实关键的只是 btrfs 在传统 Unix 文件系统的「目录/文件/inode」 这些东西之外只增加了一个「子卷」的新概念，而子卷间可以共享元数据或者数据， 用快照命令创建出的子卷就是共享一部分元数据。

于是子卷在存储介质中是如何记录的呢？

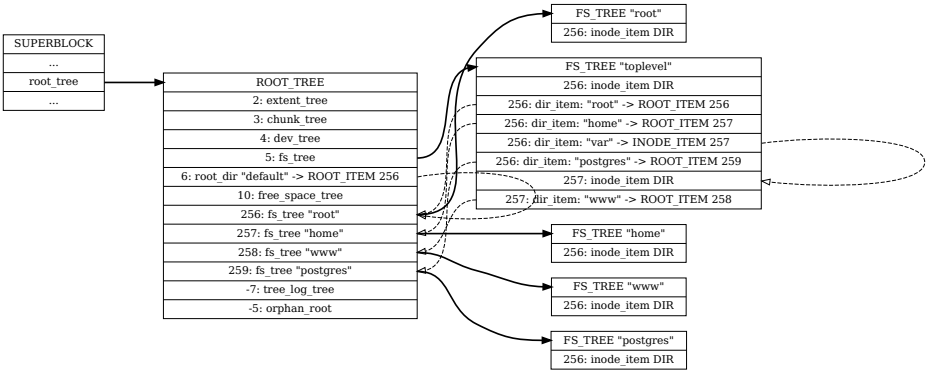
比如在 SysadminGuide 这页的 Flat 布局 有个子卷布局的例子。

```
toplevel         (volume root direc
tory, not to be mounted by default)
    +-- root       (subvolume root
directory, to be mounted at /)
    +-- home       (subvolume root
directory, to be mounted at /home)
    +-- var        (directory)
    |   \-- www    (subvolume root
directory, to be mounted at /var/ww
w)
    \-- postgres   (subvolume root
directory, to be mounted at /var/li
b/postgresql)
```

用圆柱体表示子卷的话画成图大概是这个样子：



首先要说明， btrfs 中大部分长度可变的数据结构都是 CoW B-tree ，一种经过修改适合写时拷贝的B树结构，所以在 on-disk format 中提到了很多个树。这里的树不是指文件系统中目录结构树，而是 CoW B-tree ，如果不关心B树细节的话可以把 btrfs 所说的一棵树理解为关系数据库中的一个表， 和数据库的表一样 btrfs 的树的长度可变，然后表项内容根据一个 key 排序。 有这样的背景之后，上图例子中的 Flat 布局在 btrfs 中大概是这样的数据结构：



上图中已经隐去了很多和本文无关的具体细节，所有这些细节都可以通过 btrfs inspect-internal 的 dump-super 和 dump-tree 查看到。btrfs 中的每棵树都可以看

作是一个数据库中的表，可以包含很多表项，根据 KEY 排序，而 KEY 是 (object_id, item_type, item_extra) 这样的三元组。每个对象（object）在树中用一个或多个表项（item）描述，同 object_id 的表项共同描述一个对象（object）。B树中的 key 只用来比较大小不必连续，从而 object_id 也不必连续，只是按大小排序。有一些预留的 object_id 不能用作别的用途，他们的编号范围是 -255ULL 到 255ULL，也就是表中前 255 和最后 255 个编号预留。

ROOT_TREE 中记录了到所有别的B树的指针，在一些文档中叫做 tree of tree roots 。「所有别的B树」 举例来说比如 2 号 extent_tree ，3 号 chunk_tree ， 4 号 dev_tree ，10 号 free_space_tree ，这些B树都是描述 btrfs 文件系统结构非常重要的组成部分，但是在本文关系不大， 今后有机会再讨论它们。在 ROOT_TREE 的 5 号对象有一个 fs_tree ，它描述了整个 btrfs pool 的顶级子卷，也就是图中叫 toplevel 的那个子卷（有些文档用定冠词称 the FS_TREE 的时候就是在说这个 5 号树，而不是别的子卷的 FS_TREE ）。除了顶级子卷之外，别的所有子卷的 object_id 在 256ULL 到 -256ULL 的范围之间，对子卷而言 ROOT_TREE 中的这些 object_id 也同时是它们的 子卷 id ，在内核挂载文件系统的时候可以用 subvolid 找到它们，别的一些对子卷的操作也可以直接用 subvolid 表示一个子卷。 ROOT_TREE 的 6 号对象描述的不是一棵树，而是一个名叫 default 的特殊目录，它指向 btrfs pool 的默认挂载子卷。最初 mkfs 的时候，这个目录指向 ROOT_ITEM 5 ，也就是那个顶级子卷，之后

可以通过命令 `btrfs subvolume set-default` 修改它指向别的子卷，这里它被改为指向 ROOT_ITEM 256 亦即那个名叫 "root" 的子卷。

每一个子卷都有一棵自己的 FS_TREE （有的文档中叫 file tree），一个 FS_TREE 相当于传统 Unix 文件系统中的一整个 inode table，只不过它除了包含 inode 信息之外还包含所有文件夹内容。在 FS_TREE 中，object_id 同时也是它所描述对象的 inode 号，所以 btrfs 的 **子卷有互相独立的 inode 编号**，不同子卷中的文件或目录可以拥有相同的 inode。 或许有人不太清楚子卷间 inode 编号独立意味着什么，简单地说，这意味着你不能跨子卷创建 hard link，不能跨子卷 mv 移动文件而不产生复制操作。不过因为 reflink 和 inode 无关，可以跨子卷创建 reflink，也可以用 reflink + rm 的方式快速移动文件。

FS_TREE 中一个目录用一个 inode_item 和多个 dir_item 描述， inode_item 是目录自己的 inode，那些 dir_item 是目录的内容。 dir_item 可以指向别的 inode_item 来描述普通文件和子目录， 也可以指向 root_item 来描述这个目录指向一个子卷。有人或许疑惑，子卷就没有自己的 inode 么？其实如果看 数据结构定义 的话 `struct btrfs_root_item` 结构在最开头的地方包含了一个 `struct btrfs_inode_item` 所以 root_item 也同时作为子卷的 inode，不过用户通常看不到这个子卷的 inode，因为子卷在被（手动或自动地）挂载到目录上之后， 用户会看到的是子卷的根目录的 inode。

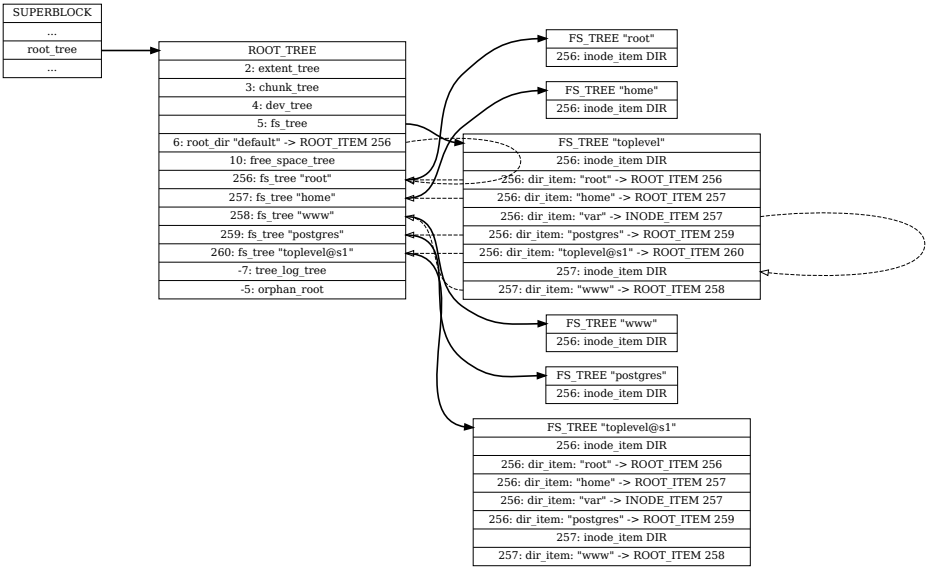
比如上图 FS_TREE toplevel 中，有两个对象，第一个 256 是（子卷的）根目录，第二个 257 是 "var" 目录，256 有4个子目录，其中 "root" "home" "postgres" 这三个指向了 ROOT_TREE 中的对应子卷，而 "var" 指向了 inode 257 。然后 257 有一个子目录叫 "www" 它指向了 ROOT_TREE 中 object_id 为 258 的子卷。

那么快照又是如何记录的呢？

以上是子卷、目录、 inode 在 btrfs 中的记录方式，你可能想知道，如何记录一个快照呢？ 特别是，如果对一个包含子卷的子卷创建了快照，会得到什么结果呢？ 如果我们在上面的布局基础上执行：

```
btrfs subvolume snapshot toplevel t
oplevel/toplevel@s1
```

那么产生的数据结构大概如下所示：


SUPERBLOCK
...
root_tree
...
ROOT_TREE
2: extent_tree
3: chunk_tree
4: dev_tree
5: fs_tree
6: root_dir "default" -> ROOT_ITEM 256
10: free_space_tree
256: fs_tree "root"
257: fs_tree "home"
258: fs_tree "www"
259: fs_tree "postgres"
260: fs_tree "toplevel@s1"
-7: tree_log_tree
-5: orphan_root
FS_TREE "root"
256: inode_item DIR
FS_TREE "home"
256: inode_item DIR
FS_TREE "toplevel"
256: inode_item DIR
256: dir_item: "root" -> ROOT_ITEM 256
256: dir_item: "home" -> ROOT_ITEM 257
256: dir_item: "var" -> INODE_ITEM 257
256: dir_item: "postgres" -> ROOT_ITEM 259
256: dir_item: "toplevel@s1" -> ROOT_ITEM 260
257: inode_item DIR
257: dir_item: "www" -> ROOT_ITEM 258
FS_TREE "www"
256: inode_item DIR
FS_TREE "postgres"
256: inode_item DIR
FS_TREE "toplevel@s1"
256: inode_item DIR
256: dir_item: "root" -> ROOT_ITEM 256
256: dir_item: "home" -> ROOT_ITEM 257
256: dir_item: "var" -> INODE_ITEM 257
256: dir_item: "postgres" -> ROOT_ITEM 259
257: inode_item DIR
257: dir_item: "www" -> ROOT_ITEM 258

在 ROOT_TREE 中增加了 260 号子卷，其内容复制自 toplevel 子卷，然后 FS_TREE toplevel 的 256 号 inode 也就是根目录中增加一个 dir_item 名叫 *toplevel@s1* 它指向 ROOT_ITEM 的 260 号子卷。这里看似是完整复制了整个 FS_TREE 的内容，这是因为 CoW b-tree 当只有一个叶子节点时就复制整个叶子节点。如果子卷内容再多一些，除了叶子之外还有中间节点， 那么只有被修改的叶子和其上的中间节点需要复制。从而创建快照的开销基本上是 O(level of FS_TREE)，而B树的高度一般都能维持在很低的程度，所以快照创建速度近乎是常数开销。

从子卷和快照的这种实现方式，可以看出： **虽然子卷可以嵌套子卷，但是对含有嵌套子卷的子卷做快照的语义有些特别** 。上图中我没有画 *toplevel@s1* 下的各个子卷到对应 ROOT_ITEM 之间的虚线箭头， 是因为这时候如果你尝试直接跳过 *toplevel* 挂载 *toplevel@s1* 到挂载点， 会发现那些子卷没有被自动挂载，更奇怪的是那些子卷的目录项也不是个普通目录， 尝试往它们中放东

西会得到无权访问的错误，对它们能做的唯一事情是手动将别的子卷挂载在上面。 推测原因在于这些子目录并不是真的目录，没有对应的目录的 inode ，试图查看它们的 inode 号会得到 2 号，而这是个保留号不应该出现在 btrfs 的 inode 号中。 每个子卷创建时会记录包含它的上级子卷，用 `btrfs subvolume list` 可以看到每个子卷的 top level subvolid ，猜测当挂载 A 而 A 中嵌套的 B 子卷记录的上级子卷不是 A 的时候， 会出现上述奇怪行为。嵌套子卷的快照还有一些别的奇怪行为，大家可以自己探索探索。

建议用平坦的子卷布局

因为上述嵌套子卷在做快照时的特殊行为， 我个人建议是 **保持平坦的子卷布局** ，也就是说：

1. 只让顶层子卷包含其它子卷，除了顶层子卷之外的子卷只做手工挂载，不放嵌套子卷
2. 只在顶层子卷对其它子卷做快照，不快照顶层子卷
3. 虽然可以在顶层子卷放子卷之外的东西（文件或目录），不过因为想避免对顶层子卷做快照， 所以避免在顶层子卷放普通文件。

btrfs 的子卷可以设置「可写」或者「只读」，在创建一个快照的时候也可以通过 `-r` 参数创建出一个只读快照。通常只读快照可能比可写的快照更有用，因为

`btrfs send` 命令只接受只读快照作为参考点。子卷可以有两种方式切换它是否只读的属性，可以通过 `btrfs property set <subvol> ro` 直接修改是否只读，也可以对只读子卷用 `btrfs subvolume snapshot` 创建出可写子卷，或者反过来对可写子卷创建出只读子卷。

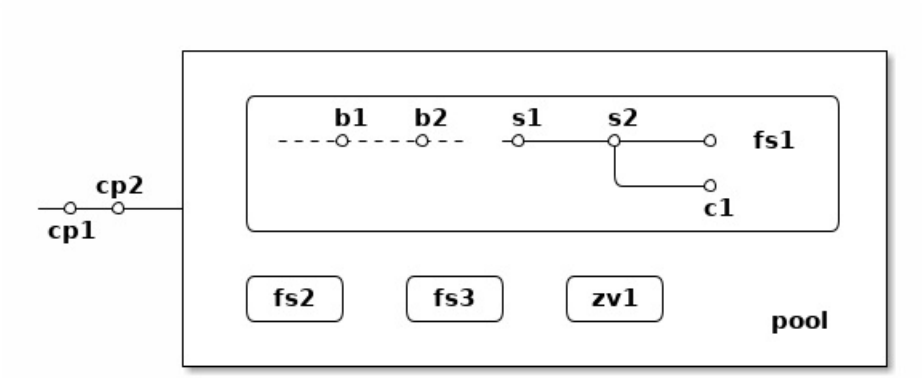
只读快照也有些特殊的限制，在 SysadminGuide#Special_Cases 就提到一例，你不能把只读快照用 mv 移出包含它的目录，虽然你能用 mv 给它改名或者移动包含它的目录 到别的地方。 btrfs wiki 上给出这个限制的原因是子卷中记录了它的上级， 所以要移动它到别的上级需要修改这个子卷，从而只读子卷没法移动到别的上级（ 不过我还没搞清楚子卷在哪儿记录了它的上级，记录的是上级目录还是上级子卷）。不过这个限制可以通过 对只读快照在目标位置创建一个新的只读快照，然后删掉原位置的只读快照来解决。

ZFS 的文件系统（filesystem）、快照（snapshot）、克隆（clone）及其它

Btrfs 给传统文件系统只增加了子卷的概念，相比之下 ZFS 中类似子卷的概念有好几个，据我所知有这些：

- 数据集（dataset）
- 文件系统（filesystem）
- 快照（snapshot）
- 克隆（clone）
- 书签（bookmark）：从 ZFS on Linux v0.6.4 开始
- 检查点（checkpoint）：从 ZFS on Linux v0.8.0 开始

梳理一下这些概念之间的关系也是最初想写下这篇笔记的初衷。先画个简图，随后逐一讲讲这些概念：



上图中，假设我们有一个 pool，其中有 3 个文件系统叫 fs1~fs3 和一个 zvol 叫 zv1，然后文件系统 fs1 有两个快照 s1 和 s2，和两个书签 b1 和 b2。pool 整体有两个检查点 cp1 和 cp2。这个简图将作为例子在后面介绍这些概念。

ZFS 设计中和快照相关的一些术语和概念

数据集（dataset）

ZFS 中把文件系统、快照、克隆、zvol 等概念统称为数据集（dataset）。一些文档和介绍中把文件系统叫做数据集，大概因为在 ZFS 中，文件系统是最先创建并且最有用的数据集。

在 ZFS 的术语中，把底层管理和释放存储设备空间的叫做 ZFS 存储池（pool），简称 zpool，其上可以容纳多个数据集，这些数据集用类似文件夹路径的语法 `pool_name/dataset_path@snapshot_name` 这样来称呼。存储池中的数据集一同共享可用的存储空间，每个数据集单独跟踪自己所消耗掉的存储空间。

数据集之间有类似文件夹的层级父子关系，这一点有用的地方在于可以在父级数据集上设定一些 ZFS 参数，这些参数可以被子级数据集基础，从而通过层级关系可以方便地微调 ZFS 参数。在 btrfs 中目前还没有类似的属性继承的功能。

zvol 的概念和本文关系不大，这里简要介绍一下。zvol 是在 ZFS 存储池中虚拟出的一个块设备，可以在 zvol 上创建别的类型的文件系统，比如 ext4 或者 xfs 这种。很多对 ZFS 文件系统能做的操作，比如快照或者 send 也都能对 zvol 做，从而用 zvol 能把 ZFS 当作一个

传统的卷管理器，绕开 ZFS 的 ZPL（ZFS Posix filesystem Layer） 层。在 Btrfs 中可以用 loopback 块设备某种程度上模拟 zvol 的功能。

文件系统（filesystem）

创建了 ZFS 存储池后，首先要在其中创建文件系统（filesystem），才能在文件系统中存储文件。 容易看出 ZFS 文件系统的概念直接对应 btrfs 中的子卷。文件系统（filesystem）这个术语， 从命名方式来看或许是想要和（像 Solaris 的 SVM 或者 Linux 的 LVM 这样的）传统的卷管理器 与其上创建的多个文件系统（Solaris UFS 或者 Linux ext）这样的上下层级做类比。 从 btrfs 的子卷在内部结构中叫作 FS_TREE 这一点可以看出，至少在 btrfs 早期设计中大概也是把子卷称为 filesystem 做过类似的类比的。 和传统的卷管理器与传统文件系统的上下层级不同的是， ZFS 和 btrfs 中由存储池跟踪和管理可用空间， 做统一的数据块分配和释放，没有分配的数据块算作整个存储池中所有 ZFS 文件系统或者 btrfs 子卷的可用空间。

与 btrfs 的子卷不同的是， ZFS 的文件系统之间是完全隔离的，（除了后文会讲的 dedup 方式之外）不可以共享任何数据或者元数据。一个文件系统还包含了隶属于其中的快照（snapshot）、 克隆（clone）和书签（bookmark）。在 btrfs 中一个子卷和对其创建的快照之间虽然有父子关系， 但是在 ROOT_TREE 的记录中属于平级的关系。

上面简图中 pool 里面包含 3 个文件系统，分别是 fs1~3 。

快照（snapshot）

ZFS 的快照对应 btrfs 的只读快照，是标记数据集在某一历史时刻上的只读状态。 和 btrfs 的只读快照一样，ZFS 的快照也兼作 send/receive 时的参考点。 快照隶属于一个数据集，这说明 ZFS 的文件系统或者 zvol 都可以创建快照。

ZFS 中快照是排列在一个时间线上的，因为都是只读快照，它们是数据集在历史上的不同时间点。 这里说的时间不是系统时钟的时间，而是 ZFS 中事务组（TXG, transaction group）的一个序号。 整个 ZFS pool 的每次写入会被合并到一个事务组，对事务组分配一个严格递增的序列号， 提交一个事务组具有类似数据库中事务的语义：要么整个事务组都被完整提交，要么整个 pool 处于上一个事务组的状态，即使中间发生突然断电之类的意外也不会破坏事务语义。 因此 ZFS 快照就是数据集处于某一个事务组时的状态。

如果不满于对数据集进行的修改，想把整个数据集恢复到之前的状态，那么可以回滚（rollback ）数据集到一个快照。回滚操作会撤销掉对数据集的所有更改，并且默认参数下只能回滚到最近的一个快照。 如果想回滚到更早的快照，可以先删掉最近的几个，或者可以使用 `zfs rollback -r` 参数删除中间的快照并回滚。

除了回滚操作，还可以直接只读访问到快照中的文件。 ZFS 的文件系统中有个隐藏文件夹叫 ".zfs" ，所以如果只想回滚一部分文件，可以从 ".zfs/snapshots/SNAPSHOT-NAME" 中把需要的文件复制出来。

比如上面简图中 fs1 就有 `pool/fs1@s1` 和 `pool/fs1@s2` 这两个快照， 那么可以在 fs1 挂载点下 `.zfs/snapshots/s1` 的路径直接访问到 s1 中的内容。

克隆（clone）

ZFS 的克隆有点像 btrfs 的可写快照。因为 ZFS 的快照是只读的，如果想对快照做写入，那需要先用 `zfs clone` 从快照中建出一个克隆，创建出的克隆和快照共享元数据和数据， 然后对克隆的写入不影响数据集原本的写入点。 创建了克隆之后，作为克隆参考点的快照会成为克隆的依赖，克隆存在期间无法删除掉作为其依赖的快照。

一个数据集可以有多个克隆，这些克隆都独立于数据集当前的写入点。使用 `zfs promote` 命令可以把一个克隆「升级」成为数据集的当前写入点，从而数据集原本的写入点会调转依赖关系， 成为这个新写入点的一个克隆，被升级的克隆原本依赖的快照和之前的快照会成为新数据集写入点的快照。

比如上面简图中 fs1 有 c1 的克隆，它依赖于 s2 这个快照，从而 c1 存在的时候就不能删除掉 s2 。

书签（bookmark）

这是 ZFS 一个比较新的特性，ZFS on Linux 分支从 v0.6.4 开始支持创建书签的功能。

书签特性存在的理由是基于这样的事实：原本 ZFS 在 send 两个快照间的差异的时候，比如 send S1 和 S2 之间的差异，在发送端实际上只需要 S1 中记录的时间戳（TXG id），而不需要 S1 快照的数据， 就可以计算出 S1 到 S2 的差异。在接收端则需要 S1 的完整数据，在其上根据接收到的数据流创建 S2 。 因此在发送端，可以把快照 S1 转变成书签，只留下时间戳元数据而不保留任何目录结构或者文件内容。 书签只能作为增量 send 时的参考点，并且在接收端需要有对应的快照，这种方式可以在发送端节省很多存储。

通常的使用场景是，比如你有一个笔记本电脑，上面有 ZFS 存储的数据，然后使用一个服务器上 ZFS 作为接收端，定期对笔记本上的 ZFS 做快照然后 send 给服务器。在没有书签功能的时候， 笔记本上至少得保留一个和服务器上相同的快照，作为 send 的增量参考点，而这个快照的内容已经在服务器上，所以笔记本中存有相同的快照只是在浪费存储空间。 有了书签功能之后，每次将定期的新快照发送到服务器之后，就可以把这个快照转化成书签，节省存储开销。

检查点（checkpoint）

这也是 ZFS 的新特性， ZFS on Linux 分支从 v0.8.0 开始支持创建检查点。

简而言之，检查点可以看作是整个存储池级别的快照，使用检查点能快速将整个存储池都恢复到上一个状态。 这边有篇文章介绍 ZFS checkpoint 功能的背景、用法和限制 ，可以看出当存储池中有检查点的时候很多存储池的功能会受影响（比如不能删除 vdev 、不能处于 degraded 状态、不能 scrub 到当前存储池中已经释放而在检查点还在引用的数据块）， 于是检查点功能设计上更多是给系统管理员准备的用于调整整个 ZFS pool 时的后悔药， 调整结束后日用状态下应该删除掉所有检查点。

ZFS 的概念与 btrfs 概念的对比

先说书签和检查点，因为这是两个 btrfs 目前完全没有的功能。

书签功能完全围绕 ZFS send 的工作原理，而 ZFS send 位于 ZFS 设计中的 DSL_ 层面，甚至不关心它 send 的快照的数据是来自文件系统还是 zvol 。在发送端它只是从目标快照递归取数据块，判断 TXG 是否老于参照点的快照，然后把新的数据块全部发往 send stream ；在接收端也只是完整地接收数据块， 不加以处理，。与之不同的是 btrfs 的 send 的工作原理是工作在文件系统的只读子卷层面， 发送端在内核代码中根据目标快照

的 b 树和参照点快照的 generation 生成一个 diff （可以通过 `btrfs subvolume find-new` 直接拿到这个 diff ），然后在用户态代码中根据 diff 和参照点、目标快照的两个只读子卷的数据产生一连串修改文件系统的指令， 指令包括创建文件、删除文件、让文件引用数据块（保持 reflink ）等操作；在接收端则完全工作在用户态下， 根据接收到的指令重建目标快照。可见 btrfs send 需要在发送端读取参照点快照的数据（比如找到 reflink 引用），从而 btrfs 没法（或者很难）实现书签功能。

检查点也是 btrfs 目前没有的功能。 btrfs 目前不能对顶层子卷做递归的 snapshot ，btrfs 的子卷也没有类似 ZFS 数据集的层级关系和可继承属性，从而没法实现类似检查点的功能。

除了书签和检查点之外，剩下的概念可以在 ZFS 和 btrfs 之间有如下映射关系：

ZFS 文件系统: btrfs 子卷

ZFS 快照: btrfs 只读快照

ZFS 克隆: btrfs 可写快照

对 ZFS 数据集的操作，大部分也可以找到对应的对 btrfs 子卷的操作。

zfs list: `btrfs subvolume list`

zfs create: `btrfs subvolume create`

zfs destroy: `btrfs subvolume delete`

zfs rename: `mv`

zfs snapshot: `btrfs subvolume snapshot -r`

zfs rollback: 这个在 btrfs 需要对只读快照创建出可写的快照（用 snapshot 命令，或者直接修改读写属性），然后改名或者调整挂载点

zfs diff: `btrfs subvolume find-new`

zfs clone: `btrfs subvolume snapshot`

zfs promote: 和 rollback 类似，可以直接调整 btrfs 子卷的挂载点

可见虽然功能上类似，但是至少从管理员管理的角度而言， zfs 对文件系统、快照、克隆的划分更为清晰，对他们能做的操作也更为明确。这也是很多从 ZFS 迁移到 btrfs ，或者反过来从 btrfs 换用 zfs 时，一些人困惑的起源（甚至有人据此说 ZFS 比 btrfs 好在 cli 设计上）。

不过 btrfs 子卷的设计也使它在系统管理上有了更大的灵活性。比如在 btrfs 中删除一个子卷不会受制于别的子卷是否存在，而在 zfs 中要删除一个快照必须先保证先摧毁掉依赖它的克隆。 再比如 btrfs 的可写子卷没有主次之分，而 zfs 中一个文件系统和其克隆之间有明显的区

别，所以需要 promote 命令调整差异。还有比如 ZFS 的文件系统只能回滚到最近一次的快照， 要回滚到更久之前的快照需要删掉中间的快照，并且回滚之后原本的文件系统数据和快照数据就被丢弃了； 而 btrfs 中因为回滚操作相当于调整子卷的挂载，所以不需要删掉快照， 并且回滚之后原本的子卷和快照还可以继续保留。

加上 btrfs 有 reflink ，这给了 btrfs 在使用中更大的灵活性，可以有一些 zfs 很难做到的用法。 比如想从快照中打捞出一些虚拟机镜像的历史副本，而不想回滚整个快照的时候，在 btrfs 中可以直接 `cp --reflink=always` 将镜像从快照中复制出来，此时的复制将和快照共享数据块； 而在 zfs 中只能用普通 cp 复制，会浪费很多存储空间。

ZFS 中是如何存储这些数据集的呢

要讲到存储细节，首先需要提一下 ZFS 的分层设计。不像 btrfs 基于现代 Linux 内核，有许多现有文件系统已经实现好的基础设施可以利用，并且大体上只用到一种核心数据结构（CoW的B树）； ZFS 则脱胎于 Solaris 的野心勃勃，设计时就分成很多不同的子系统，逐步提升抽象层次， 并且每个子系统都发明了许多特定需求下的数据结构来描述存储的信息。

ZFS 在设计之初背负了重构 Solaris 诸多内核子系统的重任，从而不同于 Linux 的文件系统 只负责文件系统的功能而把其余功能（比如内存脏页管理，IO调度）交给内核更底层的子系统， ZFS 的整体设计更层次化并更独立，很多部分可能和 Linux 内核已有的子系统有功能重叠。 而本文想讲的只是 ZFS 中与快照相关的一些部分，于是先从 ZFS 的整体设计上说一下和快照相关的概念位于 ZFS 设计中的什么位置。

和本文内容密切相关的是 ZPL 、 DSL、 DMU 这些 ZFS 子系统。

ZFS 的存储格式概况

Sun 曾经写过一篇 ZFS 的 On disk format 对理解 ZFS 如何存储在磁盘

Docutils System Messages

System Message: ERROR/3 (/home/travis/build/farseerfc/farse
vs-zfs-difference-in-implementing-snapshots.zhs.rst, line 557); *backlink*

Unknown target name: "dsl".