Btrfs vs ZFS 实现 snapshot 的差异

Table of Contents

- Btrfs 的子卷（subvolume）和快照（snapshot）
- ZFS 的数据集（dataset）、快照（snapshot）、克隆（clone）、书签（bookmark）和检查点（checkpoint）
    - 数据集（dataset）

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。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。

为了更好地理解这些差异，我四处查询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。

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，或者不精确地随口说说的时候，也经常有人把 btrfs 的快照命令创建出的子卷叫做一个快照。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。

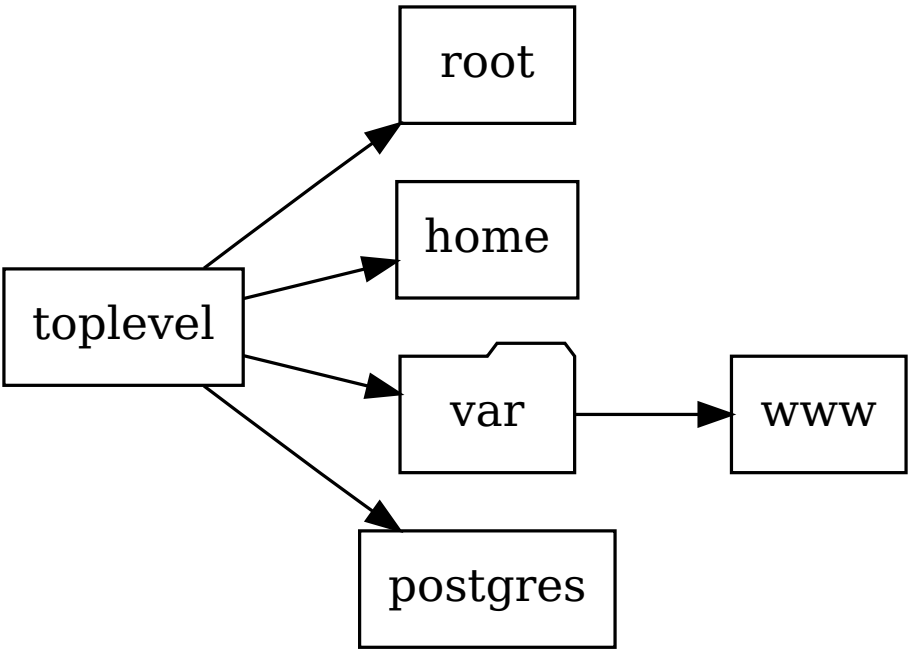
定义中「互相共享一部分 **元数据** 」比较重要，因为除了快照的方式之外， btrfs 的子卷间也可以通过 reflink 的形式共享数据块。我们可以对一整个子卷（甚至目录）执行 `cp -r --reflink=always` ，创建出一个副本，副本的文件内容通过 reflink 共享原本的数据，但不共享元数据，这样创建出的就不是快照。

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。于是这个子卷在文件系统中具体是如何记录的呢？ 举个例子解释可能比较好理解：

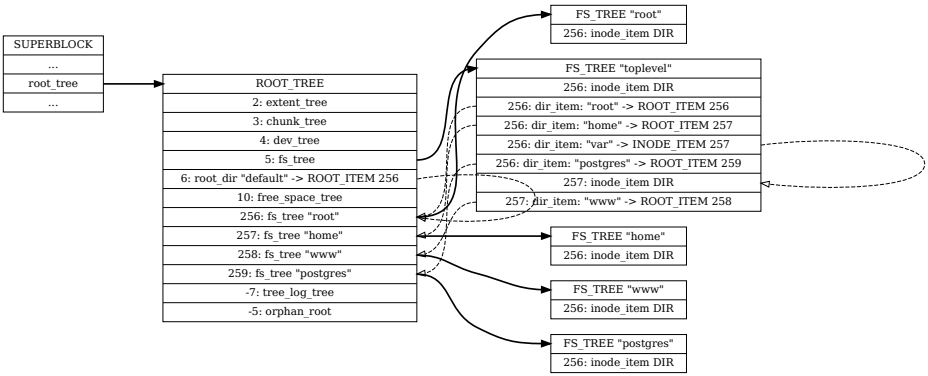
比如在 SysadminGuide 这页的 Flat 布局 有个子卷布局的例子。

```
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表，可以包含很多表项，根据 KEY 排序，而 KEY 是 (object_id, item_type, item_offset) 这样的三元组。每个 object 在树中用一个或多个 item 描述，同 object_id 的 item 共同描述一个对象（object）。B树中的 key 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 中包含了到别的所有 tree 的定义，像 2 号 extent_tree ，3 号 chunk_tree ， 4 号 dev_tree ，10 号 free_space_tree ，这些 tree 都是描述文件系统结构非常重要的 tree 。然后在 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。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 。 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 ，描述普通文件和子目录， 也可以指向 root_item ，描述这个目录指向一个子卷。

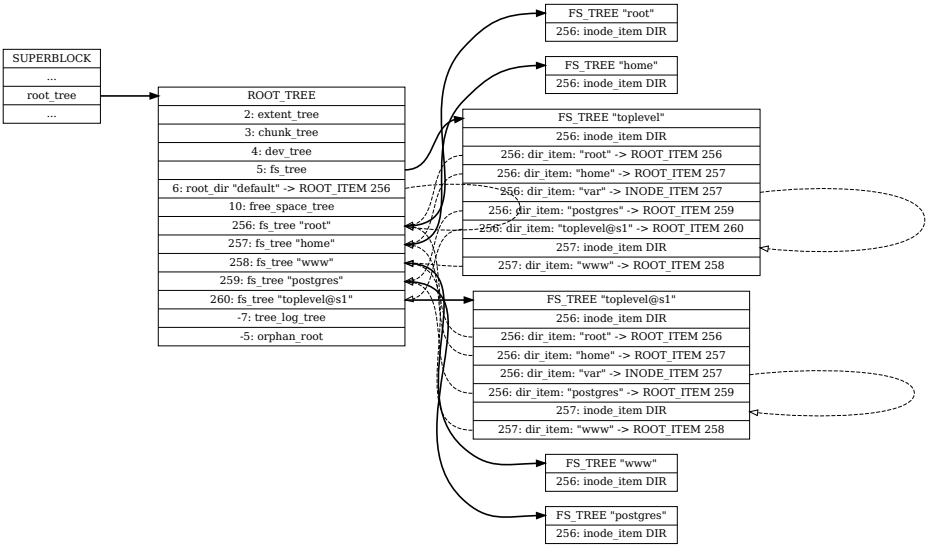
比如上图 FS_TREE toplevel 中，有两个对象，第一个 256 是（子卷的）根目录，第二个 257 是 "var" 目录，256 有4个子目录，其中 "root" "home" "postgres"

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。

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？ 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 "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
FS_TREE "www"
256: inode_item DIR
FS_TREE "postgres"
256: inode_item DIR

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 ，当只有一个叶子时就复制整个叶子。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。

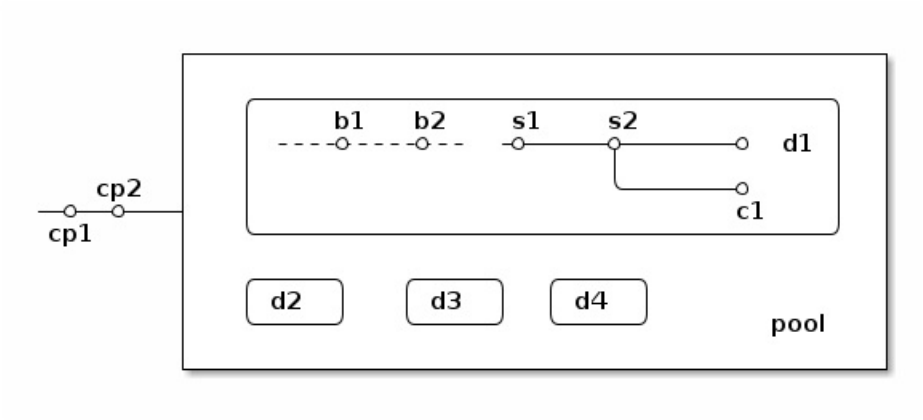
从子卷和快照的这种实现方式，可以看出：**虽然子卷可以嵌套子卷，但是对含有嵌套子卷的子卷做快照难以快速实现**。因此在目前实现的 btrfs 语义中，当子卷 S1 嵌套有别的子卷 S2 的时候，对 S1 做

ZFS 的数据集（dataset）、快照（snapshot）、克隆（clone）、书签（bookmark）和检查点（checkpoint）

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，分别叫：

- 数据集（dataset）
- 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，随后逐一讲讲这些东西：



数据集（dataset）

先从最简单的概念说起。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，其上可以创建多个数据集（dataset）。容

易看出数据集的概念直接对应 btrfs 中的子卷。也有很多介绍 ZFS 的文档中把一个数据集（dataset）叫做一个文件系统（filesystem），这或许是想要和（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。

与 btrfs 的子卷不同的是，ZFS 的数据集之间是完全隔离的，（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。