Btrfs vs ZFS 实现 snapshot 的差异

目次

目录

1 Btrfs 的子卷和快照
1.1 子卷和快照的术语
1.2 于是子卷在存储介质中是如何记录的呢？

1.3 那么快照又是如何记录的呢？
2 ZFS 的文件系统、快照、克隆及其它
2.1 ZFS 设计中和快照相关的一些术语和概念
- 数据集
- 文件系统
- 快照
- 克隆
- 书签
- 检查点

2.2 ZFS 的概念与 btrfs 概念的对比
2.3 ZFS 中是如何存储这些数据集的呢
- ZFS 的块指针
- ZPL 的元对象集

3 创建快照这么简单么？那么删除快照呢？
3.1 日志结构文件系统中用的垃圾回收算法
3.2 WAFL 早期使用的可用空间位图数组
3.3 ZFS 中关于快照和克隆的空间跟踪算法
- 乌龟算法：概念上 ZFS 如何删快照
- 兔子算法：死亡列表算法（ZFS早期）
- 豹子算法：死亡列表的子列表
- 生存日志：ZFS 如何管理克隆的空间占用

3.4 btrfs 的空间跟踪算法：引用计数与反向引用
- EXTENT_TREE 和引用计数
- 反向引用（back reference）

■ 遍历反向引用(backref walking)
4 btrfs vs ZFS 的 dedup 现状
4.1 ZFS 是如何实现 dedup 的？

> zfs 这个东西倒是名不符实。叫 z storage stack 明显更符合。 叫 fs 但不做 fs 自然确实会和 btrfs 有很大出入。
> 我反而以前还好奇为什么 btrfs 不弄 zvol ， 直到我意识到这东西真是一个 fs ，名符奇实。
> —— 某不愿透露姓名的 Ext2FSD 开发者

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 的在线去重（in-band dedup），Btrfs 的 reflink，在另一边没有的情况，进而需要不同细粒度的子卷划分方案。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。

1 Btrfs 的子卷和快照

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。

1.1 子卷和快照的术语

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。

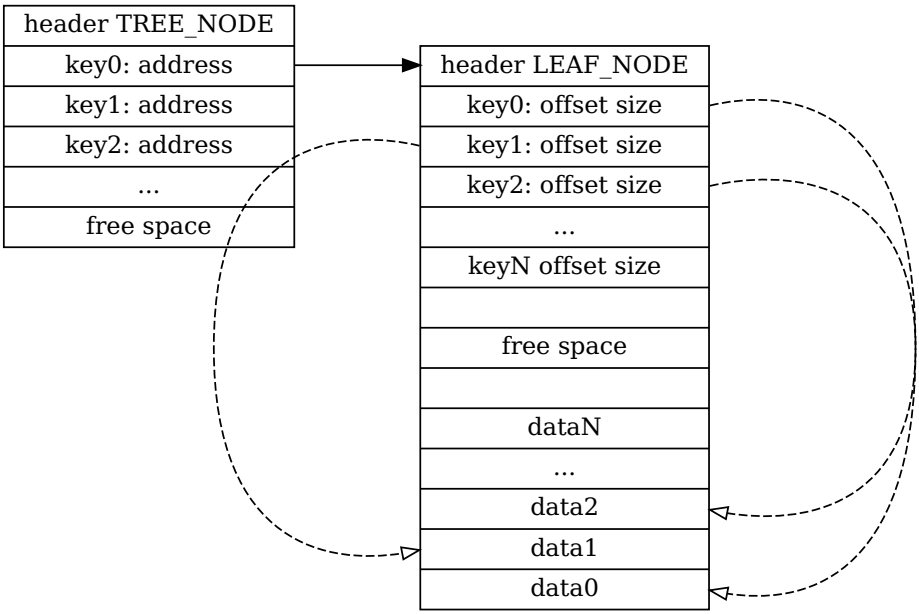
说了这么多，其实关键的只是 btrfs 在传统 Unix 文件系统的「目录/文件/inode」 这些东西之外只增加了一个「子卷」的新概念，而子卷间可以共享元数据或者数据， 用快照命令创建出的子卷就是共享一部分元数据。

1.2 于是子卷在存储介质中是如何记录的呢？

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，而是写时拷贝B树（CoW B-tree，下文简称B树），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。

B树结构由索引 key 、中间节点和叶子节点构成。每个 key 是一个 `(uint64_t object_id, uint8_t item_type, uint64_t item_extra)` 这样的三元组，三元组每一项的具体含义由 item_type 定义。 key 三元组构成了对象的概念，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。B树中的 key 只用来比较大小而不必连续，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。

B树中间节点和叶子节点结构大概像是这个样子：



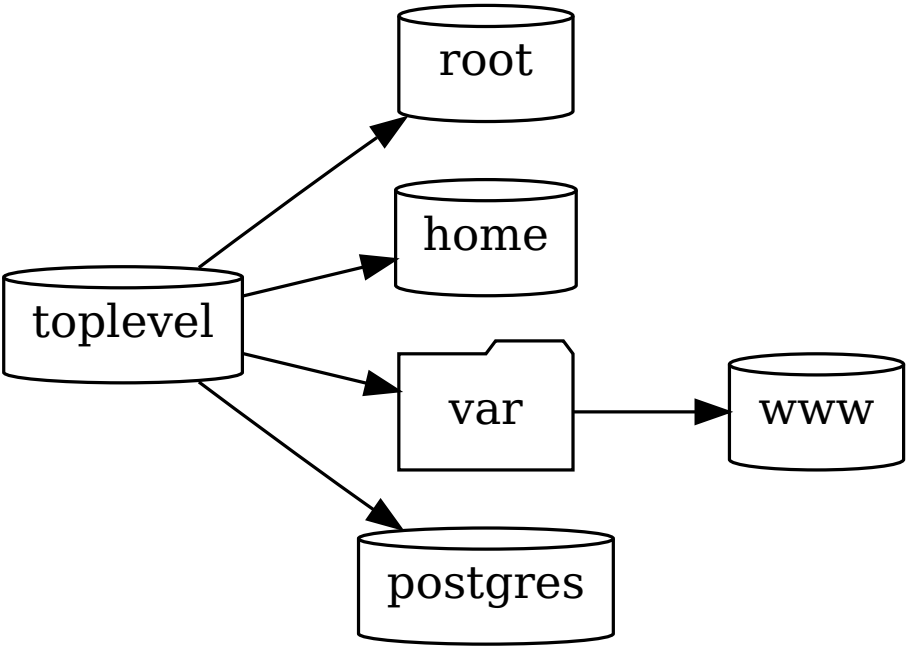
由此，每个中间节点保存一系列 key 到叶子节点的指针，而叶子节点内保存一系列 item ，每个 item 固定大小，并指向节点内某个可变大小位置的 data 。从而逻

辑上一棵B树可以包含任何类型的 Item，每个 Item 都可以有可变大小的附加数据。通过这样的B树结构，可以紧凑而灵活地表达很多数据类型。

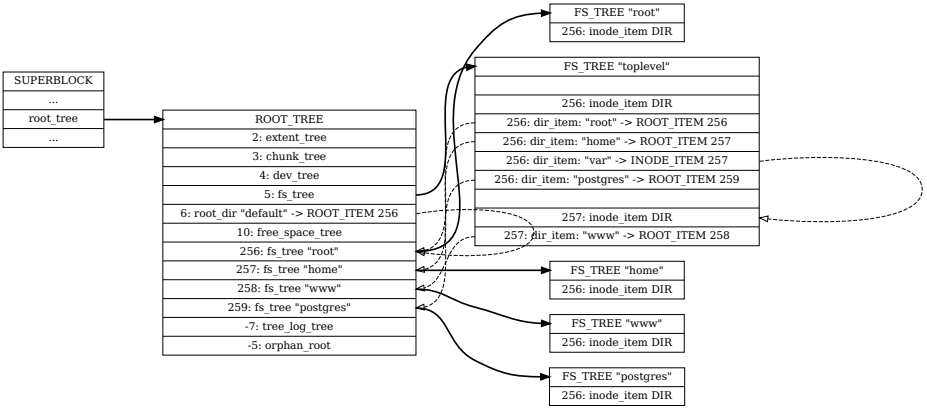
有这样的背景之后，比如在 SysadminGuide 这页的 Flat 布局 有个子卷布局的例子。

```
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：



上图例子中的 Flat 布局在 btrfs 中大概是这样的数据结构， 其中实线箭头是B树一系列中间节点和叶子节点，逻辑上指向一棵B树，虚线箭头是根据 inode 号之类的编号的引用：



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。

ROOT_TREE 中记录了到所有别的B树的指针。在一

ROOT_TREE 中记录了到所有别的B树的指针，在些文档中叫做 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

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（这里移动加引号是因为 inode 变了，传统上不算移动）。

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。

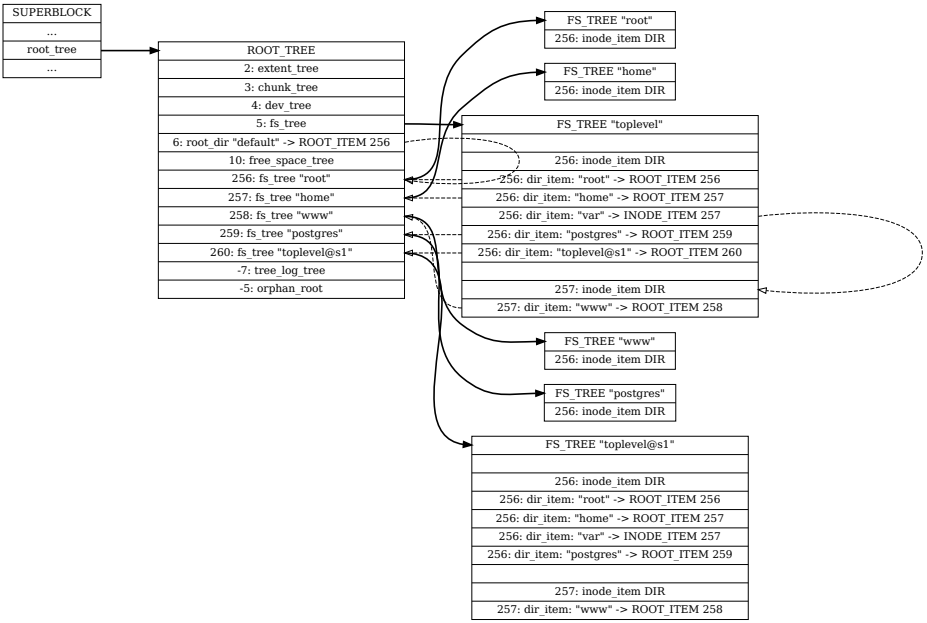
1.3 那么快照又是如何记录的

1.3 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：

```
 1  btrfs subvolume snapshot toplevel to
plevel/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。

2 ZFS 的文件系统、快照、克隆及其它

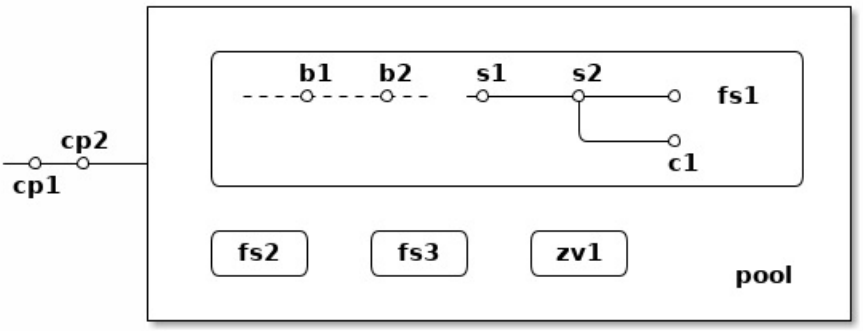
Btrfs 给传统文件系统只增加了子卷的概念，相比之下 ZFS 中类似子卷的概念有好几个，据我所知有这些：

- 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。

2.1 ZFS 设计中和快照相关的一些术语和概念

数据集

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，可以参考我上一篇 ZFS 子系统笔记中 ZVOL 的说明 。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

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

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

ZFS 的克隆（clone）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

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（bookmark）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

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（checkpoint）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。

2.2 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。

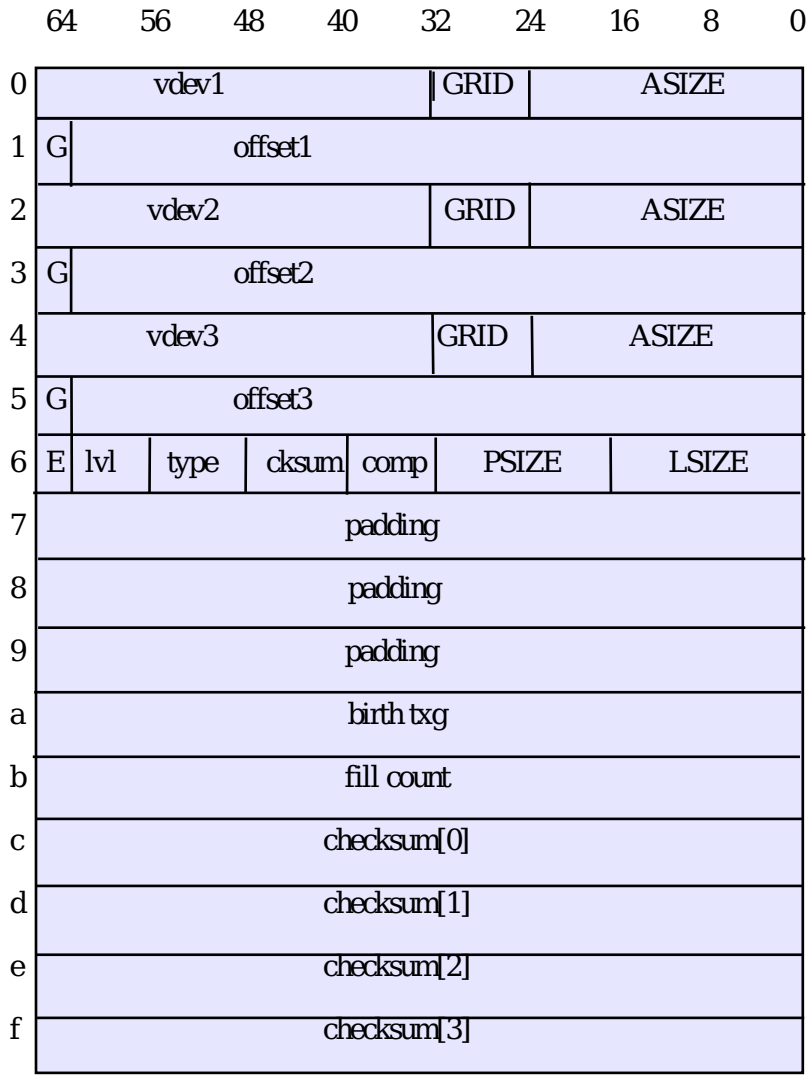
2.3 ZFS 中是如何存储这些数据集的呢

要讲到存储细节，首先需要 了解一下 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。 在这里和本文内容密切相关的是 ZPL 、 DSL 、 DMU 这些 ZFS 子系统。

Sun 曾经写过一篇 ZFS 的 On disk format 对理解 ZFS 如何存储在磁盘上很有帮助，虽然这篇文档是针对 Sun 还在的时候 Solaris 的 ZFS ，现在 ZFS 的内部已经变化挺大，不过对于理解本文想讲的快照的实现方式还具有参考意义。这里借助这篇 ZFS On Disk Format 中的一些图示来解释 ZFS 在磁盘上的存储方式。

ZFS 的块指针

ZFS 中用的 128 字节块指针



要理解 ZFS 的磁盘结构首先想介绍一下 ZFS 中的块指针（block pointer, blkptr_t），结构如右图所

指针（block pointer，blkptr_t），结构如右图所示。 ZFS 的块指针用在 ZFS 的许多数据结构之中，当需要从一个地方指向任意另一个地址的时候都会 插入这样的一个块指针结构。大多数文件系统中也有类似的指针结构，比如 btrfs 中有个8字节大小的逻辑地址（logical address），一般也就是个 4字节 到 16字节 大小的整数写着扇区号、块号或者字节偏移，在 ZFS 中的块指针则是一个巨大的128字节（不是 128bit !）的结构体。

128字节块指针的开头是3个数据虚拟地址（DVA, Data Virtual Address），每个 DVA 是 128bit ，其中记录这块数据在什么设备（vdev）的什么偏移（offset）上占用多大（asize)，有 3个 DVA 槽是用来存储最多3个不同位置的副本。然后块指针还记录了这个块用什么校验算法（ cksum ）和什么压缩算法（comp），压缩前后的大小（PSIZE/LSIZE），以及256bit的校验和（checksum）。

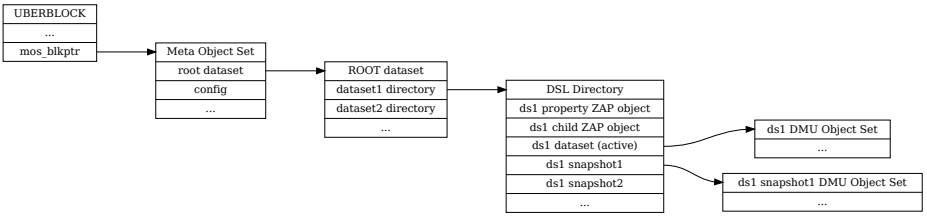
当需要间接块（indirect block）时，块指针中记录了间接块的层数（lvl），和下层块指针的数量（fill）。一个间接块就是一个数据块中包含一个块指针的数组，当引用的对象很大需要很多块时，间接块构成一棵树状结构。

块指针中还有和本文关系很大的一个值 birth txg ，记录这个块指针诞生时的整个 pool 的 TXG id 。一次 TXG 提交中写入的数据块都会有相同的 birth txg ，这个相当于 btrfs 中 generation 的概念。 实际上现在的 ZFS 块指针似乎记录了两个 birth txg ，分别在图中的9行和a行的位置， 一个 physical 一个 logical ，用于 dedup 和 device removal 。值得注意的是块指针里只有 birth txg

device removal。值得注意的是块指针里只有 birth txg，没有引用计数或者别的机制做引用，这对后面要讲的东西很关键。

ZPL 的元对象集

理解块指针和 ZFS 的子系统层级之后，就可以来看看 ZFS 存储在磁盘上的具体结构了。 因为涉及的数据结构种类比较多，所以先来画一张逻辑上的简图，其中箭头只是某种引用关系不代表块指针， 方框也不是结构体细节：

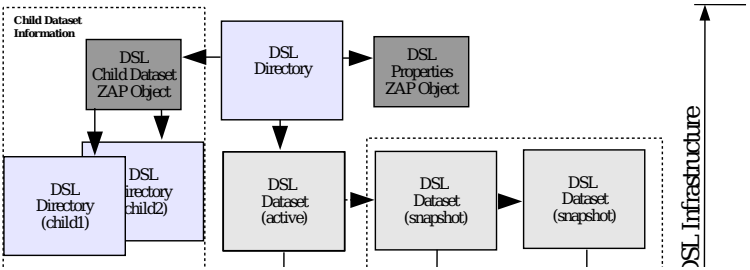


如上简图所示，首先 ZFS pool 级别有个 uberblock，具体每个 vdev 如何存储和找到这个 uberblock 今后有空再聊，这里认为整个 zpool 有唯一的一个 uberblock。从 uberblock 有个指针指向元对象集（MOS, Meta Object Set），它是个 DMU 的对象集，它包含整个 pool 的一些配置信息，和根数据集（root dataset）。根数据集再包含整个 pool 中保存的所有顶层数据集，每个数据集有一个 DSL Directory 结构。然后从每个数据集的 DSL Directory 可以找到一系列子数据集和一系列快照等结

构。最后每个数据集有个 active 的 DMU 对象集，这是整个文件系统的当前写入点，每个快照也指向一个各自

是个文件系统的当前子根点，每个快照也指向一个各自的 DMU 对象集。

DSL 层的每个数据集的逻辑结构也可以用下面的图表达（来自 ZFS On Disk Format ）：



ZFS On Disk Format 中 4.1 节的 DSL infrastructure

ZFS On Disk Format 中 4.2 节的 Meta Object Set

需要记得 ZFS 中没有类似 btrfs 的 CoW b-tree 这样的统一数据结构，所以上面的这些设施是用各种不同的数据结构表达的。 尤其每个 Directory 的结构可以包含一个 ZAP 的键值对存储，和一个 DMU 对象。 可以理解为， DSL 用 DMU 对象集（Objectset）表示一个整数（uinit64_t 的 dnode 编号）到 DMU 对象的映射，然后用 ZAP 对象表示一个名字到整数的映射，然后又有很多额外的存储于 DMU 对象中的 DSL 结构体。如果我们画出不同的指针和不同的结构体，那么会得到一个稍显复杂的图，见右边「ZFS On Disk Format 中 4.2 节的 Meta Object Set」，图中还只画到了 root_dataset 为止。

看到这里，大概可以理解在 ZFS 中创建一个 ZFS 快照的操作其实很简单：找到数据集的 DSL Directory 中当

前 active 的 DMU 对象集指针，创建一个表示 snapshot 的 DSL dataset 结构，指向那个 DMU 对象集，然后快照就建好了。因为今后对 active 的写入会写时复制对应的 DMU 对象集，所以 snapshot 指向的 DMU 对象集不会变化。

3 创建快照这么简单么？那么删除快照呢？

按上面的存储格式细节来看， btrfs 和 zfs 中创建快照似乎都挺简单的，利用写时拷贝，创建快照本身没什么复杂操作。

如果你也听到过别人介绍 CoW 文件系统时这么讲，是不是会觉得似乎哪儿少了点什么。创建快照是挺简单的， **直到你开始考虑如何删除快照** ……

或者不局限在删除单个快照上， CoW 文件系统因为写时拷贝，每修改一个文件内容或者修改一个文件系统结构， 都是分配新数据块，然后考虑是否要删除这个数据替换的老数据块，此时如何决定老数据块能不能删呢？ 删除快照的时候也是同样，快照是和别的文件系统有共享一部分数据和元数据的， 所以显然不能把快照引

用到的数据块都直接删掉，要考察快照引用的数据块是否还在别的地方被引用着， 只能删除那些没有被引用的

数据。

深究「如何删快照」这个问题，就能看出 WAFL 、btrfs 、 ZFS 甚至别的 log-structured 文件系统间的关键区别，从而也能看到另一个问题的答案： **为什么 btrfs 只需要子卷的抽象，而 zfs 搞出了这么多抽象概念？** 带着这两个疑问，我们来研究一下这些文件系统的块删除算法。

3.1 日志结构文件系统中用的垃圾回收算法

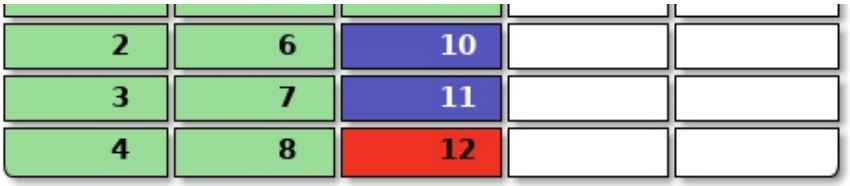
讲 btrfs 和 zfs 用到的删除算法之前，先讲一下日志结构（log-structured）文件系统中的垃圾回收（ GC, Garbage Collection）算法。对熟悉编程的人来说，讲到空间释放算法，大概首先会想到 GC ，因为这里要解决的问题乍看起来很像编程语言的内存管理中 GC 想要解决的问题：有很多指针相互指向很多数据结构，找其中没有被引用的垃圾然后释放掉。

首先要澄清一下 日志结构文件系统（log-structured file system） 的定义，因为有很多文件系统用日志，而用了日志的不一定是日志结构文件系统。 在维基百科上有个页面介绍 日志结构文件系统 ，还有个 列表列出了一些日志结构文件系统 。通常说，整个文件系统的存储结构都组织成一个大日志的样子，就说这个文件系统是日志结构的， 这包括很多早期学术研究的文件

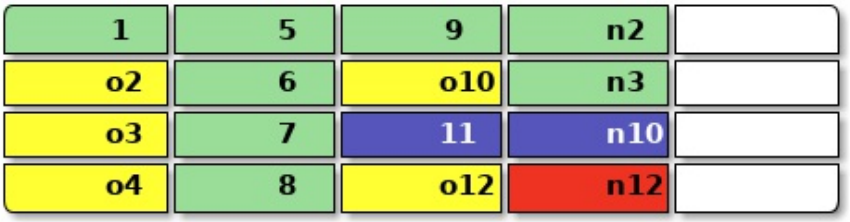
系统，以及目前 NetBSD 的 LFS 、Linux 的 NILFS ，用在光盘介质上的 UDF ，还有一些专门为闪存优化的 JFFS 、 YAFFS 以及 F2FS 。日志结构文件系统不包括那些用额外日志保证文件系统一致性，但文件系统结构不在日志中的 ext4 、 xfs 、 ntfs 、 hfs+ 。

简单来说，日志结构文件系统就是把存储设备当作一个大日志，每次写入数据时都添加在日志末尾， 然后用写时复制重新写入元数据，最后提交整个文件系统结构。因为这里用了写时复制，原本的数据块都还留着，所以可以很容易实现快照之类的功能。从这个特征上来说，写时拷贝文件系统（CoW FS）像 btrfs/zfs 这些在一些人眼中也符合日志结构文件系统的特征， 所以也有人说写时拷贝文件系统算是日志结构文件系统的一个子类。不过日志结构文件系统的另一大特征是利用 GC 回收空间，这里是本文要讲的区别，所以在我看来不用 GC 的 btrfs 和 zfs 不算是日志结构文件系统。

举个例子，比如下图是一个日志结构文件系统的磁盘占用，其中绿色是数据，蓝色是元数据（比如目录结构和 inode），红色是文件系统级关键数据（比如最后的日志提交点），一开始可能是这样，有9个数据块， 2个元数据块，1个系统块：



现在要覆盖 2 和 3 的内容，新写入 n2 和 n3 ，再删除 4 号的内容 ，然后修改 10 里面的 inode 变成 n10 引用这些新数据，然后写入一个新提交 n12 ，用黄色表示不再被引用的垃圾，提交完大概是这样：



日志结构文件系统需要 GC 比较容易理解，写日志嘛，总得有一个「添加到末尾」的写入点，比如上面图中的 n12 就是当前的写入点。空盘上连续往后写而不 GC 总会遇到空间末尾，这时候就要覆盖写空间开头， 就很难判断「末尾」在什么地方，而下一次写入需要在哪里了。 这时文件系统也不知道需要回收哪些块（图中的 o2 o3 o4 o10 和 o12），因为这些块可能被别的地方还继续引用着，需要等到 GC 时扫描元数据来判断。

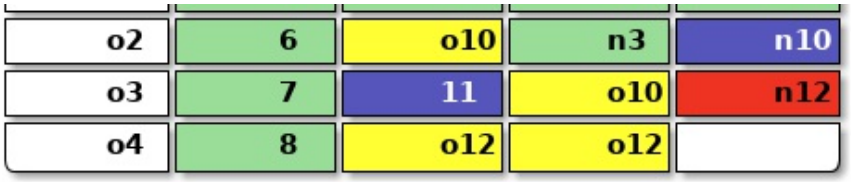
和内存管理时的 GC 不同的一点在于，文件系统的 GC 肯定不能停下整个世界跑 GC ，也不能把整个地址空间对半分然后 Mark-and-Sweep ，这些在内存中还尚可的简单策略直接放到文件系统中绝对是性能灾难。所以文件系统的 GC 需要并行的后台 GC ，并且需要更细粒度

的分块机制能在 Mark-and-Sweep 的时候保持别的地方可以继续写入数据而维持文件系统的正常职能。

通常文件系统的 GC 是这样，先把整个盘分成几个段（segment）或者区域(zone)，术语不同不过表达的概念类似， 然后 GC 时挑一个老段，扫描文件系统元数据找出要释放的段中还被引用的数据块，搬运到日志末尾，最后整个释放一段。 搬运数据块时，也要调整文件系统别的地方对被搬运的数据块的引用。

物理磁盘上一般有扇区的概念，通常是 512B 或者 4KiB 的大小，在文件系统中一般把连续几个物理块作为一个数据块， 大概是 4KiB 到 1MiB 的数量级，然后日志结构文件系统中一个段(segment)通常是连续的很多块，数量级来看大约是 4MiB 到 64MiB 这样的数量级。相比之下 ufs/ext4/btrfs/zfs 的分配器通常还有 block group 的概念， 大概是 128MiB 到 1GiB 的大小。可见日志结构文件系统的段，是位于数据块和其它文件系统 block group 中间的一个单位。段大小太小的话，会显著增加空间管理需要的额外时间空间开销，而段大小太大的话， 又不利于利用整个可用空间，这里的抉择有个平衡点。

继续上面的例子，假设上面文件系统的图示中每一列的4块是一个段，想要回收最开头那个段， 那么需要搬运还在用的 1 到空闲空间，顺带修改引用它的 n10 ，最后提交 n12 ：



要扫描并释放一整段，需要扫描整个文件系统中别的元数据（图中的 n12 和 n10 和 11）来确定有没有引用到目标段中的地址，可见释放一个段是一个 $O(N)$ 的操作，其中 N 是元数据段的数量，按文件系统的大小增长， 于是删除快照之类可能要连续释放很多段的操作在日志文件系统中是个 $O(N^2)$ 甚至更昂贵的操作。 在文件系统相对比较小而系统内存相对比较大的时候，比如手机上或者PC读写SD卡，大部分元数据块（ 其中包含块指针）都能放入内存缓存起来的话，这个扫描操作的开销还是可以接受的。 但是对大型存储系统显然扫描并释放空间就不合适了。

段的抽象用在闪存类存储设备上的一点优势在于，闪存通常也有擦除块的概念，比写入块的大小要大， 是连续的多个写入块构成，从而日志结构的文件系统中一个段可以直接对应到闪存的一个擦除块上。 所以闪存设备诸如U盘或者 SSD 通常在底层固件中用日志结构文件系统模拟一个块设备，来做写入平衡。 大家所说的 SSD 上固件做的 GC ，大概也就是这样一种操作。

基于段的 GC 还有一个显著缺陷，需要扫描元数据，复制搬运仍然被引用到的块，这不光会增加设备写入，还需要调整现有数据结构中的指针，调整指针需要更多

写入，同时又释放更多数据块， F2FS 等一些文件系统设计中把这个问题叫 Wandering Tree Problem ，在 F2FS

设计中是通过近乎「作弊」的 NAT 转换表 放在存储设备期待的 FAT 所在位置，不仅能让需要扫描的元数据更集中，还能减少这种指针调整导致的写入。

不过基于段的 GC 也有一些好处，它不需要复杂的文件系统设计，不需要特殊构造的指针， 就能很方便地支持大量快照。一些日志结构文件系统比如 NILFS 用这一点支持了「连续快照（continuous snapshots）」，每次文件系统提交都是自动创建一个快照，用户可以手动标记需要保留哪些快照， GC 算法则排除掉用户手动标记的快照之后，根据快照创建的时间，先从最老的未标记快照开始回收。 即便如此， GC 的开销（CPU时间和磁盘读写带宽）仍然是 NILFS 最为被人诟病的地方，是它难以被广泛采用的原因。 为了加快 NILFS 这类日志文件系统的 GC 性能让他们能更适合于普通使用场景，也有许多学术研究致力于探索和优化 GC ，使用更先进的数据结构和算法跟踪数据块来调整 GC 策略，比如这里有一篇 State-of-the-art Garbage Collection Policies for NILFS2 。

3.2 WAFL 早期使用的可用空间位图数组

从日志结构文件系统使用 GC 的困境中可以看出，文件系统级别实际更合适的， 可能不是在运行期依赖扫描

元数据来计算空间利用率的 GC ，而是在创建快照时或者写入数据时就预先记录下快照的空间利用情况， 从而可以细粒度地跟踪空间和回收空间，这也是 WAFL 早期实现快照的设计思路。

WAFL 早期记录快照占用数据块的思路从表面上来看也很「暴力」，传统文件系统一般有个叫做「位图（bitmap ）」的数据结构，用一个二进制位记录一个数据块是否占用，靠扫描位图来寻找可用空间和已用空间。 WAFL 的设计早期中考虑既然需要支持快照，那就把记录数据块占用情况的位图，变成快照的数组。 于是整个文件系统有个 256 大小的快照利用率数组，数组中每个快照记录自己占用的数据块位图， 文件系统中最多能容纳 255 个快照。

	block1	block2	block3	block4	block5	...	block N
filesystem	1	2	3	4	5	...	N
snapshot1	1	2	3	4	5	...	N
snapshot2	1	2	3	4	5	...	N
...	1	2	3	4	5	...	N
snapshot255	1	2	3	4	5	...	N

上面每个单元格都是一个二进制位，表示某个快照有没有引用某个数据块。有这样一个位图的数组之后，就可以直接扫描位图判断出某个数据块是否已经占用，可以找出尚未被占用的数据块用作空间分配， 也可以方便地计算每个快照引用的空间大小或者独占的空间大小，估算删除快照后可以释放的空间。

需要注意的是，文件系统中可以有非常多的块，从而位图数组比位图需要更多的元数据来表达。 比如估算

一下传统文件系统中一块可以是 4KiB 大小，那么跟踪空间利用的位图需要 1bit/4KiB ， 1TiB 的盘就需要 32MiB 的元数据来存放位图；而 WAFL 这种位图数组即便限制了快照数量只能有255个，仍需要 256bit/4KiB 的空间开销， 1TiB 的盘需要的元数据开销陡增到 8GiB ，这些还只是单纯记录空间利用率的位图数组，不包括别的元数据。

使用这么多元数据表示快照之后，创建快照的开销也相应地增加了，需要复制整个位图来创建一个新的快照， 按上面的估算 1TiB 的盘可能需要复制 256MiB 的位图，这不再是一瞬能完成的事情， 期间可能需要停下所有对文件系统的写入等待复制完成。 位图数组在存储设备上的记录方式也很有讲究，当删除快照时希望能快速读写上图中的一整行位图， 于是可能希望每一行位图的存储方式在磁盘上都尽量连续， 而在普通的写入操作需要分配新块时，想要按列的方式扫描位图数组，找到没有被快照占用的块， 从而上图中按列的存储表达也希望在磁盘上尽量连续。 WAFL 的设计工程师们在位图数组的思路下，实现了高效的数据结构让上述两种维度的操作都能快速完成， 但是这绝不是一件容易的事情。

位图数组的表达方式也有其好处，比如除了快照之外，也可以非常容易地表达类似 ZFS 的克隆和独立的文件系统这样的概念，这些东西和快照一样，占用仅有的 256 个快照数量限制。 这样表达的克隆可以有数据块和

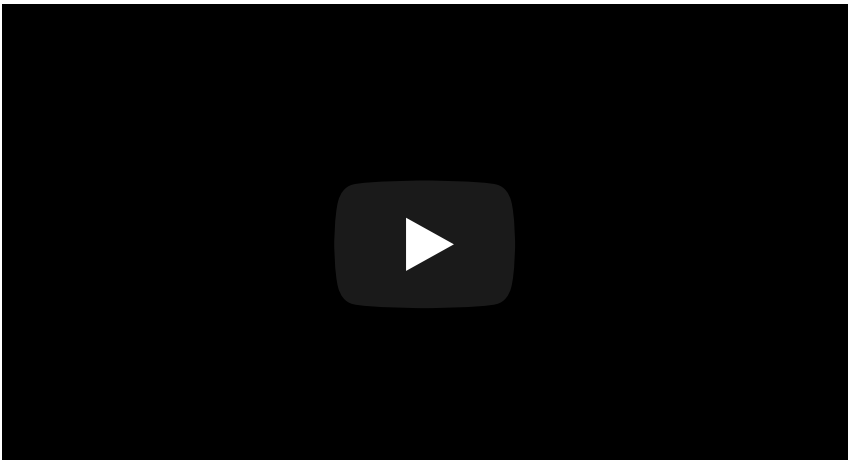
别的文件系统共享，文件系统之间也可以有类似 reflink 的机制共享数据块，在位图数组的相应位置将位置1即

可。

使用位图数组的做法，也只是 WAFL 早期可能采用的方式，由于 WAFL 本身是闭源产品，难以获知它具体的工作原理。哈佛大学和 NetApp 的职员曾经在 FAST10 (USENIX Conference on File and Storage Technologies) 上发表过一篇讲解高效跟踪和使用 back reference 的论文，叫 Tracking Back References in a Write-Anywhere File System，可以推测在新一代 WAFL 的设计中可能使用了类似 btrfs backref 的实现方式，接下来会详细介绍。

3.3 ZFS 中关于快照和克隆的空间跟踪算法

How ZFS snapshots really work And why they perform well (usually)



幻灯片可以从这里下载

OpenZFS 的项目领导者，同时也是最初设计 ZFS 中

OpenZFS 的项目领导者，同时也是最初设计 ZFS 中 DMU 子系统的作者 Matt Ahrens 在 DMU 和 DSL 中设计并实现了 ZFS 独特的快照的空间跟踪算法。他也在很多地方发表演讲，讲过这个算法的思路和细节，比如右侧就是他在 BSDCan 2019 做的演讲 How ZFS snapshots really work And why they perform well (usually) 的 YouTube 视频。

其中 Matt 讲到了三个删除快照的算法，分别可以叫做「乌龟算法」、「兔子算法」、「豹子算法」， 接下来简单讲讲这些算法背后的思想和实现方式。

乌龟算法：概念上 ZFS 如何删快照

乌龟算法没有实现在 ZFS 中，不过方便理解 ZFS 在概念上如何考虑快照删除这个问题，从而帮助理解 后面的兔子算法和豹子算法。

要删除一个快照， ZFS 需要找出这个快照引用到的「独占」数据块，也就是那些不和别的数据集或者快照共享的 数据块。 ZFS 删除快照基于这几点条件：

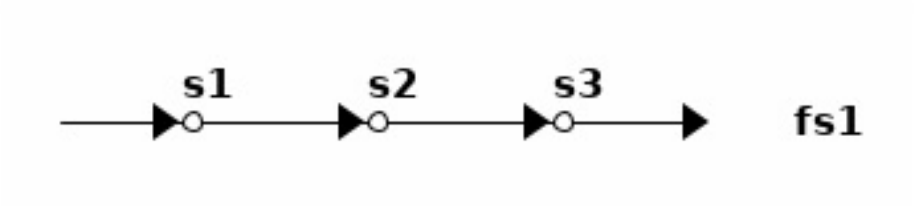
1. ZFS 快照是只读的。创建快照之后无法修改其内容。
2. ZFS 的快照是严格按时间顺序排列的，这里的时间指 TXG id ，即记录文件系统提交所属事务组的严格递增序号。
3. ZFS 不存在 reflink 之类的机制，从而在某个时间

点删除掉的数据块，不可能在比它更后面的快照中「复活」。

第三点关于 reflink 造成的数据复活现象可能需要解释一下，比如在（支持 reflink 的） btrfs 中有如下操作：

```
 1  btrfs subvolume snapshot -r fs s1
 2  rm fs/somefile
 3  btrfs subvolume snapshot -r fs s2
 4  cp --reflink=always s1/somefile fs/s
omefile
 5  btrfs subvolume snapshot -r fs s3
```

我们对 fs 创建了 s1 快照，删除了 fs 中某个文件，创建了 s2 快照，然后用 reflink 把刚刚删除的文件从 s1 中复制出来，再创建 s3 。如此操作之后，按时间顺序有 s1、s2、s3 三个快照：



其中只有 s2 不存在 somefile ，而 s1 、 s3 和当前的 fs 都有，并且都引用到了同一个数据块。 于是从时间线来看， somefile 的数据块在 s2 中「死掉」了，又在 s3 中「复活」了。

而 ZFS (目前还）不支持 reflink ，所以没法像这样让数据块复活。一旦某个数据块在某个快照中「死」

了，就意味着它在随后的所有快照中都不再被引用到了。

ZFS 的快照具有的上述三点条件，使得 ZFS 的快照删除算法可以基于 birth time 。回顾上面 ZFS 的块指针 中讲到，ZFS 的每个块指针都有一个 birth txg 属性，记录这个块诞生时 pool 所在的 txg 。于是可以根据这个 birth txg 找到快照所引用的「独占」数据块然后释放掉它们。

具体来说，乌龟算法可以这样删除一个快照：

1. 在 DSL 层找出要删除的快照（我们叫他 s ），它的前一个快照（叫它 ps ），后一个快照（叫它 ns ），分别有各自的 birth txg 叫 s.birth, ps.birth, ns.birth 。
2. 遍历 s 的 DMU 对象集指针所引出的所有块指针。这里所有块指针在逻辑上构成一个由块指针组成的树状结构，可以有间接块组成的指针树，可以有对象集的 dnode 保存的块指针，这些都可以看作是树状结构的中间节点。
    1. 每个树节点的指针 bp，考察如果 bp.birth <= ps.birth ，那么这个指针和其下所有指针都还被前一个快照引用着，需要保留这个 bp 引出的整个子树。
    2. 按定义 bp.birth 不可能 > s.birth 。
    3. 对所有满足 ps.birth < bp.birtu <= s.birth 的 bp ，需要去遍历 ns 的相应块指针（同样文件的同样偏移位置），看是否还在引用 bp 。
        - 如果有在，继续递归往下考察树状结

- 如果存在，继续递归在下考察树状结构中 bp 的所有子节点指针。因为可能共享了这个 bp 但 CoW 了新的子节点。
- 如果不存在，说明下一个快照中已经删了 bp 。这时可以确定地说 bp 是 s 的「独占」数据块。

3. 释放掉所有找到的 s 所「独占」的数据块。

上述算法的一些边角情况可以自然地处理，比如没有后一个快照时使用当前数据集的写入点， 没有前一个快照时那么不被后一个快照引用的数据块都是当前要删除快照的独占数据块。

分析一下乌龟算法的复杂度的话，算法需要分两次，读 s 和 ns 中引用到的所有 ps 之后创建的数据块的指针，重要的是这些读都是在整个文件系统范围内的随机读操作，所以速度非常慢……

兔子算法：死亡列表算法（ZFS早期）

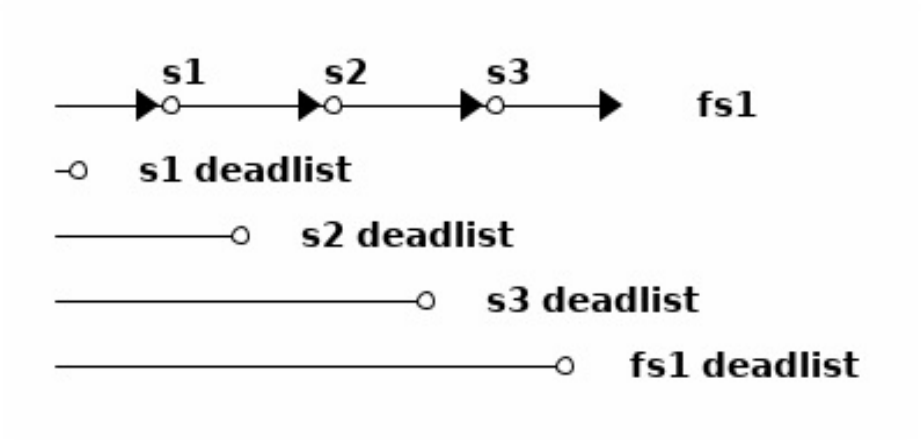
可以粗略地认为乌龟算法算是用 birth txg 优化代码路径的 GC 算法，利用了一部分元数据中的 birth txg 信息来避免扫描所有元数据，但是概念上仍然是在扫描元

数据找出快照的独占数据块， 而非记录和跟踪快照的数据块，在最坏的情况下仍然可能需要扫描几乎所有元数据

据。

兔子算法基于乌龟算法的基本原理，在它基础上跟踪快照所引用数据块的一些信息， 从而很大程度上避免了扫描元数据的开销。ZFS 在早期使用这个算法跟踪数据集和快照引用数据块的情况。

兔子算法为每个数据集（文件系统或快照）增加了一个数据结构，叫死亡列表（dead list）， 记录 **前一个快照中还活着，而当前数据集中死掉了的数据块指针**，换句话说就是在本数据集中「杀掉」的数据块。举例画图大概是这样



上图中有三个快照和一个文件系统，共 4 个数据集。每个数据集维护自己的死亡列表， 死亡列表中是那些在该数据集中被删掉的数据块。于是兔子算法把乌龟算法所做的操作分成了两部分， 一部分在文件系统删除数据时记录死亡列表，另一部分在删除快照时根据死亡列表释放需要释放的块。

在当前文件系统删除数据块（不再被当前文件系统引用）时，负责比对 birth txg 维护当前文件系统的死亡列表。每删除一个数据块，指针为 bp 时，判断 bp.birth

列表。每删除一个数据块，指针为 bp 时，判断 bp.birth 和文件系统最新的快照（上图为 s3）的 birth：

- bp.birth <= s3.birth： 说明 bp 被 s3 引用，于是将 bp 加入 fs1 的 deadlist
- bp.birth > s3.birth：说明 bp 指向的数据块诞生于 s3 之后，可以直接释放 bp 指向的块。

创建新快照时，将当前文件系统（图中 fs1）的死亡列表交给快照，文件系统可以初始化一个空列表。

删除快照时，被删除的快照 s 和前一个快照 ps 、后一个快照 ns ，需要读入后一个快照 ns 的死亡列表：

1. 对 s.deadlist 中的每个指针 bp
    - 复制 bp 到 ns.deadlist
2. 对 ns.deadlist 中的每个指针 bp
    - 如果 bp.birth > ps.birth ，释放 bp 的空间
    - 否则保留 bp

换个说法的话， **死亡列表记录的是每个数据集需要负责删除，但因为之前的快照还引用着所以不能删除的数据块列表** 。从当前文件系统中删除一个数据块时，这个职责最初落在当前文件系统身上，随后跟着创建新快照职责被转移到新快照上。 每个负责的数据集根据数据块的出生时间是否早于之前一个快照来判断现在是否能立刻释放该块， 删除一个快照时则重新评估自己负责的和下一个快照负责的数据块的出生时间。

从所做的事情来看，兔子算法并没有比乌龟算法少做很多事情。乌龟算法删除一个快照， 需要遍历当前快照和后一个快照两组数据块指针中，新写入的部分；

照和后一个快照两组数据块指针中，新导入的部分；兔子算法则需要遍历当前快照和后一个快照两个死亡数组中，新删除的块指针。 但是实际兔子算法能比乌龟算法快不少，因为维护死亡列表的操作只在文件系统删除数据时和删除快照时， 顺序写入，并且删除快照时也只需要顺序读取死亡列表。在磁盘这种块设备上，顺序访问能比随机访问有数量级的差异。

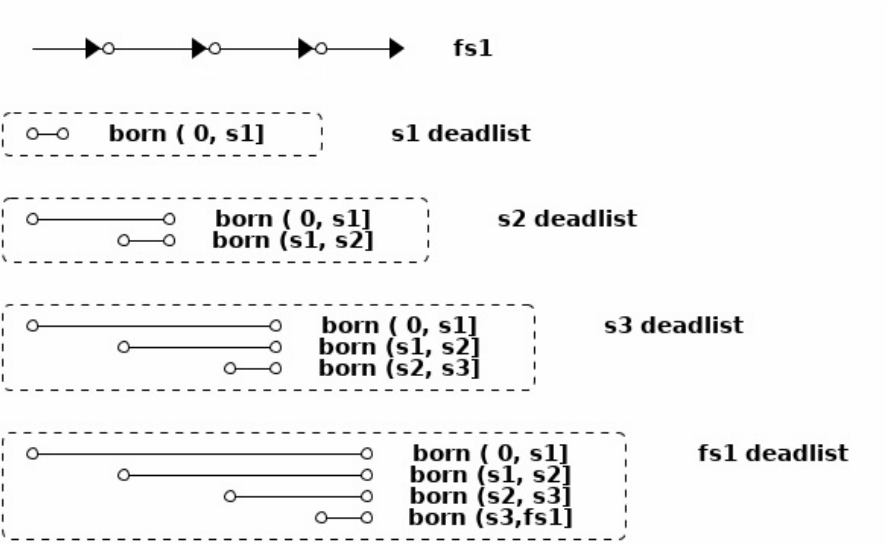
不过记录死亡列表也有一定存储开销。最差情况下，比如把文件系统写满之后，创建一个快照， 再把所有数据都删掉，此时文件系统引用的所有数据块的块指针都要保存在文件系统的死亡列表中。 按 ZFS 默认的 128KiB 数据块大小，每块需要 128 字节的块指针，存储这些死亡列表所需开销可能要 整个文件系统大小的 1/1024 。如果用 4KiB 的数据块大小，所需开销则是 1/32 ， 1TiB 的盘会有 32GiB 拿来存放这些块指针，将高于用位图数组所需的存储量。

豹子算法：死亡列表的子列表

豹子算法是 ZFS 后来在 2009 年左右实现的算法。在兔子算法中就可以看到，每次删除快照操作死亡列表的时候， 都需要扫描死亡列表中的块指针，根据指针中记录的 birth txg 做判断是否能直接释放或是需要保留到另一个快照的死亡列表。 于是豹子算法的思路是，在死亡列表中记录块指针时，就把其中的块指针按 birth txg 分成子列表（sublist）

成了列表（sublist）。

比如上面兔子算法中那4个死亡列表，可以这样拆成子列表：



这样拆成子列表之后，每次从死亡列表中释放数据块都能根据出生时间找到对应的子列表， 然后连续释放整个子列表。每次合并死亡列表时，也能直接用单链表穿起需要合并的子列表，不需要复制块指针。

死亡列表并不在跟踪快照的独占大小，而是在跟踪快照所需负责删除的数据块大小， 从这个数值可以推算出快照的独占大小之类的信息。 有了按出生时间排列的死亡列表子列表之后，事实上给任何一个出生时间到死亡时间的范围， 都可以找出对应的几个子列表，从而根据子列表的大小可以快速计算出每个快照范围的「独

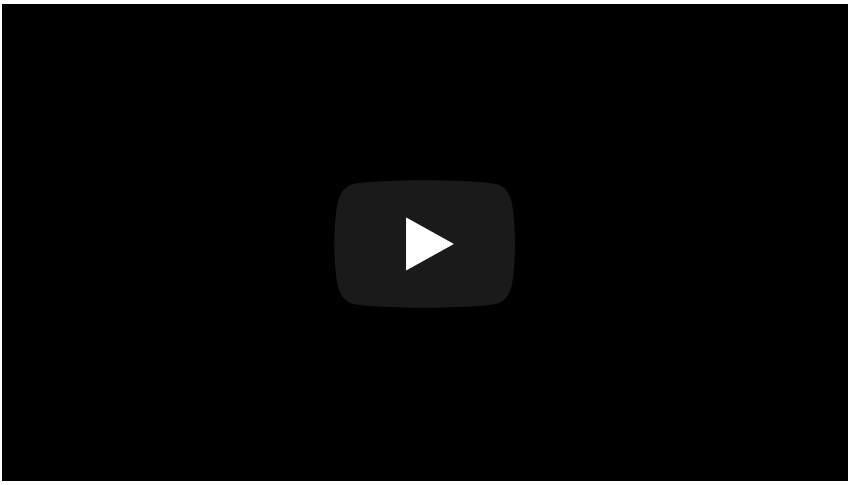
占」数据块、 「共享」数据块等大小，这不光在删除快照时很有用，也可以用来根据大小估算 zfs send 或者别的基于快照操作时需要的时间。

的基于快照操作的需要的时间。

从直觉上理解，虽然 ZFS 没有直接记录每个数据块属于哪个数据集，但是 ZFS 跟踪记录了每个数据块的归属信息，也就是说由哪个数据集负责释放这个数据块。在文件系统中删除数据块或者快照时，这个归属信息跟着共享数据块转移到别的快照中，直到最终被释放掉。

生存日志：ZFS 如何管理克隆的空间占用

Fast Clone Deletion by Sara Hartse



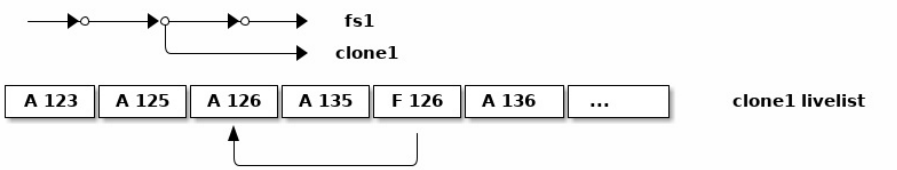
以上三种算法负责在 ZFS 中跟踪快照的空间占用，它们都基于数据块的诞生时间，所以都假设 ZFS 中对数据块的分配是位于连续的快照时间轴上。但是明显 ZFS 除了快照和文件系统，还有另一种数据集可能分配数据块，那就是 克隆 ，于是还需要在克隆中使用不同的算法单独管理因克隆而分配的数据块。OpenZFS Summit

2017 有个演讲 Fast Clone Deletion by Sara Hartse 解释了其中的细节。

首先克隆的存在本身会锁住克隆引用到的快照，不能删除这些被依赖的快照， 所以克隆无须担心靠快照共享的数据块的管理问题。因此克隆需要管理的，是从快照分离之后， 新创建的数据块。

和乌龟算法一样，原理上删除克隆的时候可以遍历克隆引用的整个 DMU 对象集，找出其中晚于快照的诞生时间的数据块，然后释放它们。也和乌龟算法一样， 这样扫描整个对象集的开销很大，所以使用一个列表来记录数据块指针。 克隆管理新数据块的思路和快照的兔子算法维持死亡列表的思路相反， 记录所有新诞生的数据块，这个列表叫做「生存日志（livelist）」。

克隆不光要记录新数据块的诞生，还要记录新数据块可能的死亡，所以磁盘上保存的生存日志虽然叫 livelist，但不像死亡列表那样是列表的形式，而是日志的形式，而内存中保存的生存日志则组织成了棵 自平衡树（AVLTree） 来加速查找。



磁盘上存储的生存日志如上图，每个表项记录它是分配（A）或者删除（F）一个数据块，同时记录数据块的地址。 这些记录在一般情况下直接记录在日志末尾，

随着对克隆的写入操作而不断增长，长到一定程度则从内存中的 AVL Tree 直接输出一个新的生存日志替代掉旧的，合并其中对应的分配和删除操作。

生存日志可以无限增长，从而要将整个生存列表载入内存也有不小的开销，这里的解决方案有点像快照管理中用 豹子算法改进兔子算法的思路，把一个克隆的整个生存日志也按照数据块的诞生时间拆分成子列表。Sara Hartse 的演讲 Fast Clone Deletion 中继续解释了其中的细节和优化方案，感兴趣的可以看看。

3.4 btrfs 的空间跟踪算法：引用计数与反向引用

理解了 ZFS 中根据 birth txg 管理快照和克隆的算法之后，可以发现它们基于的假设难以用于 WAFL 和 btrfs 。 ZFS 严格区分文件系统、快照、克隆，并且不存在 reflink ，从而可以用 birth txg 判断数据块是否需要保留，而 WAFL 和 btrfs 中不存在 ZFS 的那些数据集分工，又想支持 reflink ，可见单纯基于 birth txg 不足以管理 WAFL 和 btrfs 子卷。

让我们回到一开始日志结构文件系统中基于垃圾回收（GC）的思路上来，作为程序员来看， 当垃圾回收的性能不足以满足当前需要时，大概很自然地会想到：引

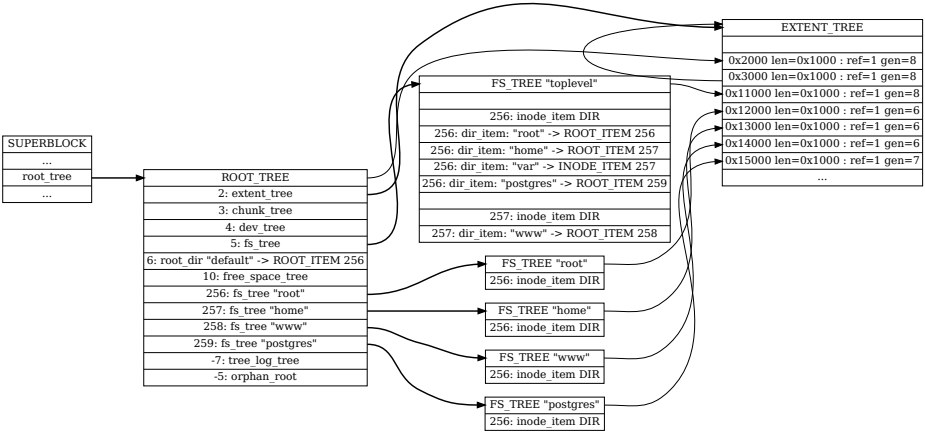
用计数（reference counting）。 编程语言中用引用计数作为内存管理策略的缺陷是：强引用不能成环， 这在文件系统中看起来不是很严重的问题，文件系统总体上看是个树状结构，或者就算有共享的数据也是个 上下层级分明的有向图，很少会使用成环的指针，以及文件系统记录指针的时候也都会区分指针的类型， 根据指针类型可以分出强弱引用。

EXTENT_TREE 和引用计数

btrfs 中就是用引用计数的方式跟踪和管理数据块的。引用计数本身不能保存在 FS_TREE 或者指向的数据块中，因为这个计数需要能够变化，对只读快照来说整个 FS_TREE 都是只读的。 所以这里增加一层抽象，btrfs 中关于数据块的引用计数用一个单独的 CoW B树来记录，叫做 EXTENT_TREE ，保存于 ROOT_TREE 中的 2 号对象位置。

btrfs 中每个块都是按 区块（extent） 的形式分配的，区块是一块连续的存储空间，而非 zfs 中的固定大小。每个区块记录存储的位置和长度， 以及这里所说的引用计数。所以本文最开始讲 Btrfs 的子卷和快照 中举例的那个平坦布局，如果画上 EXTENT_TREE 大概像是下

图这样，其中每个粗箭头是一个区块指针，指向磁盘中的逻辑地址，细箭头则是对应的 EXTENT_TREE 中关于这块区块的描述：


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"
-7: tree_log_tree
-5: orphan_root
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
257: inode_item DIR
257: dir_item: "www" -> ROOT_ITEM 258
FS_TREE "root"
256: inode_item DIR
FS_TREE "home"
256: inode_item DIR
FS_TREE "www"
256: inode_item DIR
FS_TREE "postgres"
256: inode_item DIR
EXTENT_TREE
0x2000 len=0x1000 : ref=1 gen=8
0x3000 len=0x1000 : ref=1 gen=8
0x11000 len=0x1000 : ref=1 gen=8
0x12000 len=0x1000 : ref=1 gen=6
0x13000 len=0x1000 : ref=1 gen=6
0x14000 len=0x1000 : ref=1 gen=6
0x15000 len=0x1000 : ref=1 gen=7
...

包括 ROOT_TREE 和 EXTENT_TREE 在内，btrfs 中所有分配的区块（extent）都在 EXTENT_TREE 中有对应的记录，按区块的逻辑地址索引。从而给定一个区块，能从 EXTENT_TREE 中找到 ref 字段描述这个区块有多少引用。不过 ROOT_TREE 、 EXTENT_TREE 和别的一些数据结构本身不是引用计数的， 这些数据结构对应的区块的引用计数总是 1 ，不会和别的树共享区块；从 FS_TREE 开始的所有树节点都可以共享区块，这包括所有子卷的元数据和文件数据，这些区块对应的引用计数可以大于 1 表示有多处引用。

EXTENT_TREE 按区块的逻辑地址索引，记录了起始地址和长度，所以 EXTENT_TREE 也兼任 btrfs 的空间利用记录，充当别的文件系统中 block bitmap 的指责。比

如上面例子中的 extent_tree 就表示 `[0x2000,0x4000)` `[0x11000,0x16000)` 这两段连续的空间是已用空间，剩下的空间按定义则是可用空间。为了加速空间分配

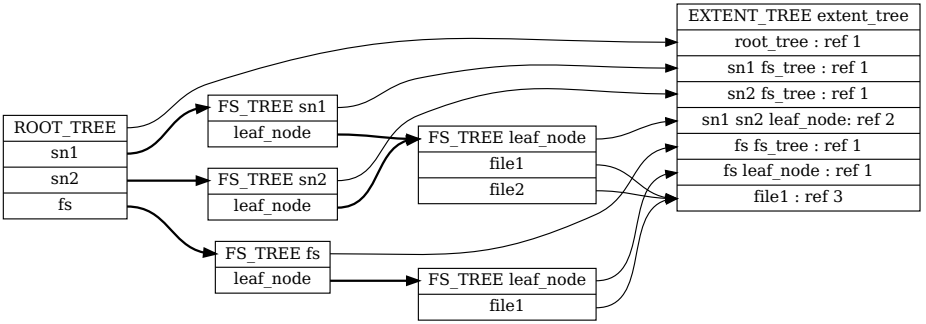
器， btrfs 也有额外的 free space cache 记录在 ROOT_TREE 的 10 号位置 free_space_tree 中，不过在 btrfs 中这个 free_space_tree 记录的信息只是缓存，必要时可以通过 `btrfs check --clear-space-cache` 扔掉这个缓存重新扫描 extent_tree 并重建可用空间记录。

比如我们用如下命令创建了两个文件，通过 reflink 让它们共享区块，然后创建两个快照， 然后删除文件系统中的 file2 ：

```
1  write fs/file1
2  cp --reflink=always fs/file1 fs/file2

3  btrfs subvolume snapshot fs sn1
4  btrfs subvolume snapshot fs sn2
5  rm fs/file2
```

经过以上操作之后，整个 extent_tree 的结构中记录的引用计数大概如下图所示：



图中可见，整个文件系统中共有5个文件路径可以访问到同一个文件的内容，分别是 `sn1/file1`, `sn1/file2`, `sn2/file1`, `sn2/file2`, `fs/file1` ， 在 extent_tree 中， sn1 和 sn2 可能共享了一个 B树 叶子节点，这个叶子节点的引用计数为 2 ，然后每个文件的内容都指向同一个 extent ，这个 extent 的引用计数为 3 。

删除子卷时，通过引用计数就能准确地释放掉子卷所引用的区块。具体算法挺符合直觉的：

1. 从子卷的 FS_TREE 往下遍历

1. 从子卷的 FS_TREE 往下遍历
    - 遇到引用计数 >1 的区块，减小该块的计数即可，不需要再递归下去
    - 遇到引用计数 =1 的区块，就是子卷独占的区块，需要释放该块并递归往下继续扫描

大体思路挺像上面介绍的 ZFS 快照删除的乌龟算法，只不过根据引用计数而非 birth txg 判断是否独占数据块。 扫描 FS_TREE 可能需要耗时良久的，这个递归的每一步操作都会记录在 ROOT_TREE 中专门的结构，也就是说删除一个子卷的操作可以执行很长时间并跨越多个 pool commit。 `btrfs subvolume delete` 命令默认也只是记录下这个删除操作，然后就返回一句类似：`Delete subvolume (no-commit): /subvolume/path` 的输出，不会等删除操作执行结束。 相比之下 ZFS 那边删除一个快照或克隆必须在一个 txg 内执行完，没有中间过程的记录， 所以如果耗时很久会影响整个 pool 的写入，于是 ZFS 那边必须对这些操作优化到能在一个 txg 内执行完的程度。

只有引用计数就足够完成快照的创建、删除之类的功能，也能支持 reflink 了（仔细回想， reflink 其实就是 reference counted link 嘛），普通读写下也只需要引用计数。 但是只有引用计数不足以知道区块的归属，不能用引用计数统计每个子卷分别占用多少空间， 独占多少区块而又共享多少区块。上面的例子就可以看出，所有

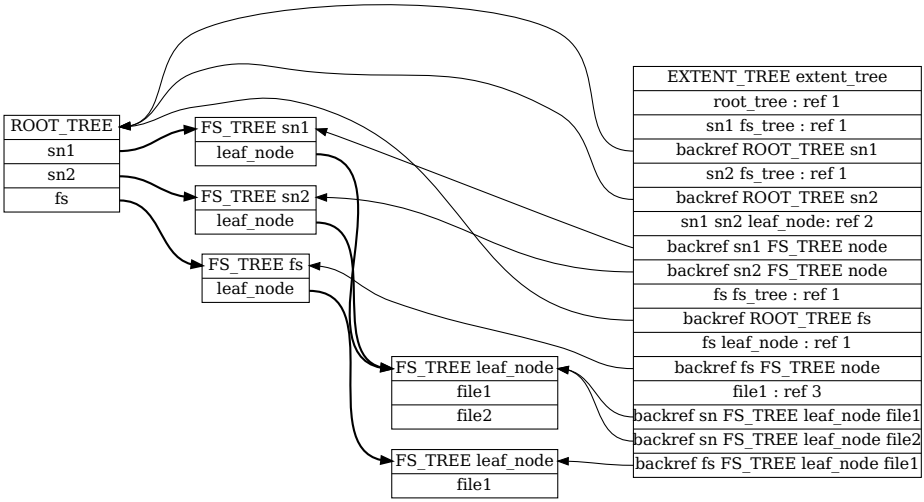
文件都指向同一个区块，该区块的引用计数为 3 ，而文件系统中一共有 5 个路径能访问到该文件。可见从区块根据引用计数反推子卷归属信息不是那么一目了然的。

反向引用（back reference）

单纯从区块的引用计数难以看出整个文件系统所有子卷中有多少副本。 也就是说单有引用计数的一个数字还不够，需要记录具体反向的从区块往引用源头指的引用，这种结构在 btrfs 中叫做「反向引用（back reference，简称 backref）」。所以在上图中每一个单向的箭头，在 btrfs 中都有记录一条反向引用，通过反向引用记录能反过来从被指针指向的位置找回到记录指针的地方。

反向引用（backref）是 btrfs 中非常关键的机制，在 btrfs kernel wiki 专门有一篇页面 Resolving Extent Backrefs 解释它的原理和实现方式。

对上面的引用计数的例子画出反向引用的指针大概是这样：



EXTENT_TREE 中每个 extent 记录都同时记录了引用到这个区块的反向引用列表。反向引用有两种记录方式：

1. 普通反向引用（Normal back references）。记录这个指针来源所在是哪颗B树、 B树中的对象 id 和对象偏移。
    - 对文件区块而言，就是记录文件所在子卷、inode、和文件内容的偏移。
    - 对子卷的树节点区块而言，就是记录该区块的上级树节点在哪个B树的哪个位置开始。
2. 共享反向引用（Shared back references）。记录这个指针来源区块的逻辑地址。
    - 无论对文件区块而言，还是对子卷的树节点区块而言，都是直接记录了保存这个区块指针的上层树节点的逻辑地址。

有两种记录方式是因为它们各有性能上的优缺点：

普通反向引用
因为通过对象编号记录，所以当树节点 CoW 改变了地址时不需要改变， 从而在普通的读写和快照之类的操作下有更好的性能， 但是在解析反向引用时需要额外一次树查找。 同时因为这个额外查找，普通反向引用也叫间接反向引用。

共享反向引用
因为直接记录了逻辑地址，所以当这个地址的节点被 CoW 的时候也许有调整这里记录的地址。 在普通的读写和快照操作下，调整地址会增加写入从而影响性能，但

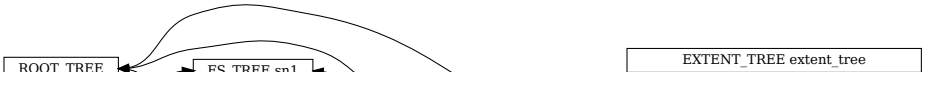
是在解析反向引用时更快。

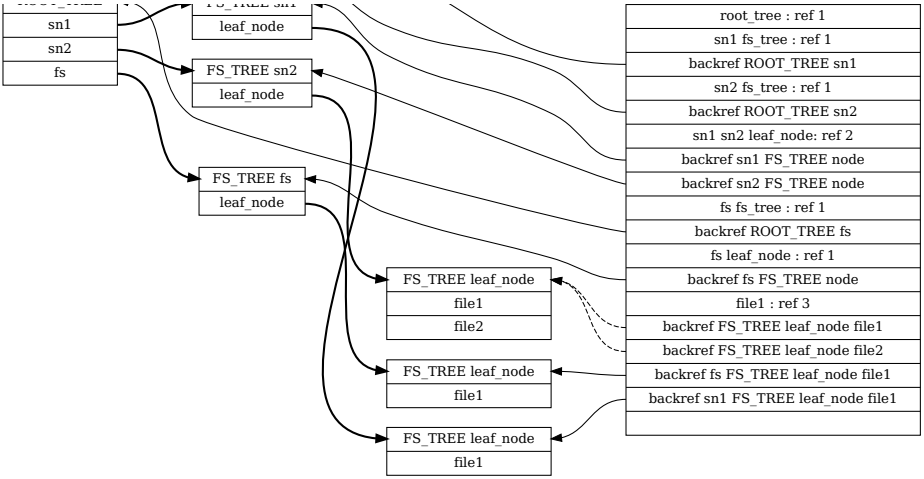
通常通过普通写入、快照、 reflink 等方式创建出来

的引用是普通反向引用， 由于普通反向引用记录了包含它的B树，从而可以说绑在了某棵树比如某个子卷上， 当这个普通反向引用指向的对象不再存在，而这个反向引用还在通过别的途径共享时， 这个普通反向引用会变成共享反向引用；共享反向引用在存在期间不会变回普通反向引用。

比如上图反向引用的例子中，我们先假设所有画出的反向引用都是普通反向引用，于是图中标为 file1 引用数为 3 的那个区块有 3 条反向引用记录，其中前两条都指向 sn1 里面的文件，分别是 sn1/file1 和 sn1/file2 ，然后 sn1 和 sn2 共享了 FS_TREE 的叶子节点。

假设这时我们删除 sn1/file2，执行了代码 `rm sn1/file2` 之后：





那么 sn1 会 CoW 那个和 sn2 共享的叶子节点，有了新的属于 sn1 的叶子，从而断开了原本 file1 中对这个共享叶子节点的两个普通反向引用，转化成共享反向引用

（图中用虚线箭头描述），并且插入了一个新的普通反向引用指向新的 sn1 的叶子节点。

遍历反向引用(backref walking)

有了反向引用记录之后，可以给定一个逻辑地址，从 EXTENT_TREE 中找到地址的区块记录， 然后从区块记录中的反向引用记录一步步往回遍历，直到遇到 ROOT_TREE ，最终确定这个逻辑地址的区块在整个文件系统中有多少路径能访问它。 这个遍历反向引用的操作，在 btrfs 文档和代码中被称作 backref walking 。

比如还是上面的反向引用图例中 sn1 和 sn2 完全共享叶子节点的那个例子，通过 backref walking ，我们能从 file1 所记录的 3 个反向引用，推出全部 5 个可能的访问路径。

backref walking 作为很多功能的基础设施，从 btrfs 相当早期（3.3内核）就有，很多 btrfs 的功能实际依赖 backref walking 的正确性。列举一些需要 backref walking 来实现的功能：

1. qgroup

   btrfs 的子卷没有记录子卷的磁盘占用开销，靠引用计数来删除子卷，所以也不需要详细统计子卷的占用。 不过有些场合下，统计子卷占用很有用。由于 reflink 的存在，显然子卷占用不能靠累计加减法算出来， 所以有了 qgroup 和 quota 功能，用来统计子卷占用空间。为了实现 qgroup ，需要 backref walking 来计算区块共享的情况。

2. send

   btrfs send 在计算子卷间的差异时，也通过 backref walking 寻找能 reflink 的区块，从而避免传输数据。

3. balance/scrub

   balance 和 scrub 都会调整区块的地址，通过 backref walking 能找到所有引用到这个地址的位置并正确修改地址。

可见 backref walking 的能力对 btrfs 的许多功能都非常重要。不过 backref walking 根据区块共享的情况也可能导致挺大的运行期开销，包括算法时间上的和内存占用方面的。 比如某个子卷中有 100 个文件通过 reflink 共享了同一个区块，然后对这个子卷做了 100 个快照，那么对这一个共享区块的 backref walking 结果可能解析出 10000 个路径。可见随着使用 reflink 和快照，backref walking 的开销可能爆炸式增长。

值得再强调的是，在没有开启 qgroup 的前提下，正常创建删除快照或reflink ，正常写入和覆盖区块之类的文件系统操作，只需要引用计数就足够，不需要动用 backref walking 这样的重型武器。

4 btrfs vs ZFS 的 dedup 现状

上面讨论 ZFS 的快照和克隆如何跟踪数据块时，故意避开了 ZFS 的 dedup 功能，因为要讲 dedup 可能需要先理解引用计数在文件系统中的用法，而 btrfs 正好用了引用计数。 于是我们再回来 ZFS 这边，看看 ZFS 的 dedup 是具体如何运作的。

稍微了解过 btrfs 和 ZFS 两者的人，或许有不少 btrfs 用户都眼馋 ZFS 有 in-band dedup 的能力，可以在写入数据块的同时就去掉重复数据，而 btrfs 只能「退而求其次」地选择第三方 dedup 方案，用外部工具扫描已经写入的数据，将其中重复的部分改为 reflink 。又或许有不少 btrfs 用户以为 zfs 的 dedup 就是在内存和磁盘中维护一个类似 Bloom filter 的结构，然后根据结果对数据块增加 reflink ，从而 zfs 内部大概一定有类似 reflink 的东西，进一步质疑为什么 btrfs 还迟迟没有实现这样一个 Bloom filter 。 或许还有 ZFS 用户有疑惑， 为什么 ZFS 还没有暴露出 reflink 的用户空间接口 ，或者既然 ZFS 已经有了 dedup ， 能不能临时开关 dedup 来提供类似 reflink 式的共享数据块 而避免长期开 dedup 的巨大开销。

看过上面 ZFS 中关于快照和克隆的空间跟踪算法 之后我们会发现，其实 ZFS 中并没有 能对应 btrfs reflink 的功能，而是根据数据块指针中的 birth txg 来跟踪快照和克隆的共享数据块的。这引来更多疑惑：

4.1 ZFS 是如何实现 dedup 的？