BTRFS 文件系统

21221074 江大鹏　21221075 程璐　21221077 盛晓昌

Btrfs 被称为下一代 Linux 文件系统。自从 2007 年提出后，它由富士通，Fusion-IO，英特尔，甲骨文，红帽，Strato，SUSE 以及很多其他公司联合开发，是完全开源的文件系统。本文将对该文件系统进行一个较为详细的考察。首先，我们将对 btrfs 做一个简要介绍，了解它的设计目标和已有特性。紧接着我们会详细介绍 btrfs 的设计，包括整体结构和所涉及的主要数据结构。后续几节分别是对 btrfs 的几个特点：写时复制、快照和克隆、多设备支持以及自带碎片整理的重点介绍。

1. Btrfs 简介

该文件系统的主要特点有：

- 维护元数据和数据的 CRC 校验和
- 基于 extents 的文件存储
- 支持最大文件为 16EiB
- 动态的 inode 分配
- 高效的可写快照和克隆
- 集成多设备支持
- 在线的大小调整和碎片整理
- 自带数据压缩
- 小文件的高效存储
- 针对 SSD 优化和 TRIM 支持

该文件系统的设计目标是作为 Linux 的默认文件系统，能够适用于多种场合，而不是只适用于某个特定场合的存储系统。它能在智能手机这样的小设备上很好的工作，也能在企业的生产服务器上表现优异。也就是说它需要适应多种多样的硬件。

Btrfs 文件系统的磁盘结构是一些 b-trees 组成的森林，更新操作采用写时复制的策略。磁盘块的管理是基于 extents 的，使用校验和保持数据的一致性，使用引用计数来回收空间。在文件系统中，btrfs 对于 COW 友好的 b-tree 以及引用计数的使用是独特的。

文件系统的性能依赖于大段连续的 extents 的可用性。但是随着系统的使用，空间碎片会不断增多，因此需要在线的碎片整理。由于快照的使用，磁盘 extents 就很可能被多个文件系统卷所指，这种情况使得碎片这里很有挑战性，因为，（1）只有在所有源指针都更新后，extents 才能被移动；（2）所有的快照都期望文件保持连续。

为了能很好的利用现代的 CPUs，良好的并发性就非常重要了。但是，由于采用写时复

制的更新策略，要达到这个目标就有点困难了，因为所有的更新操作都要传播到文件系统的根。我们将在这篇报告中讲解 BTRFS 是如何解决这些挑战，并实现良好性能的。我们将主要通过图表、例子以及少量的源代码分析来解释 BTRFS 的初衷以及核心概念。

2. Btrfs 设计

Btrfs 整个系统都基于一个非常有名的结构，b-tree，不过这个 b-tree 是进行了专门改进的，以对写时复制这个特性友好。我们先来看看这个文件系统总体的结构，然后再详细介绍其使用的 b-tree 结构。Btrfs b-tree 提供了一个通用的结构来存储各种类型的数据。

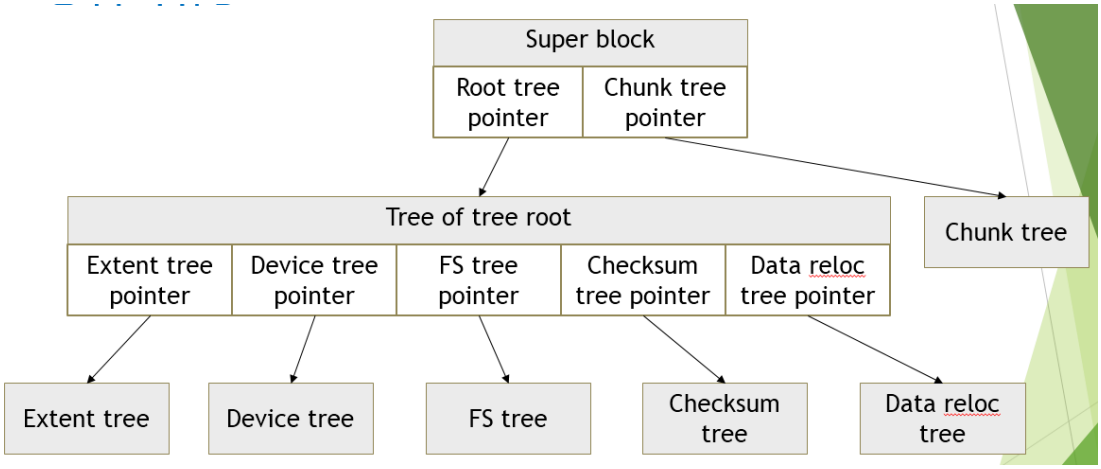

Figure 1 btrfs 结构

2.1 总体结构

每一个 btrfs 文件系统都由很多用于不同功能的 btree 组成，就如图所示。超级块（super block）在磁盘的固定位置，它指向 tree of tree root 以及 chunk tree。Tree of tree root 负责维护其他树的根节点，这里所指的其他树主要包括：

- FS tree：也被称为 sub-volume（子卷） tree，负责管理文件的元数据，如 inode_item，dir_item。每个子卷都用一颗树来实现。子卷可以被快照、被克隆，这样就增加了新的子卷树。也就是说 FS-tree 是可以有多个的。

- Extent tree：管理磁盘空间分配信息。文件系统每分配一段磁盘空间，就将该段磁盘空间的信息插入到 Extent tree，这些信息包括，这段空间起始地址、长度（以块为单位），哪些对象引用了这段空间，该 extent 是用来存储文件数据还是元数据（tree blocks）等等。

- Device tree 和 Chunk tree：为了管理物理设备所设计的中间层。每个磁盘设备都在 Chunk Tree 中有一个 item，维护逻辑 trunks 到物理 trunks 的映射。Device tree 则负责维护从物理 trunks 到逻辑 trunks 的映射。文件系统的其他部分见到的都是逻辑地址。通常 chunk tree 和 device 都比较小，可以缓存在内存中，因此可以降低因增加这样一个

设备管理的中间层所带来的性能损失。

- Checksum tree：每个分配出去的 extent 都有一个对应的 checksum item，这个 item 包含了该 extent 中的每个 block 的 checksum。
- Data reloc tree：主要用于文件系统一些特殊操作，包括 extents 的移动，例如在系统的碎片整理功能就需要用到这种类型的树。

这些树的功能差异很大，但是在实现上，内部采用了一种通用的数据结构，接下来我们将详细的介绍这个关键的数据结构。

2.2 Btree 数据结构

Btrfs 文件系统的这种通用的 btree 结构是通过精心设计的，只包含三种子数据结构，分别是：key，item 以及 block header，通过组合这三个数据结构，并赋予它们相应的值，就能实现上面提到的各种功能的树。这也正是这个文件系统的美感所在，整个系统使用统一的结构。内部节点和叶子节点都包含 bock header，单其余结构是有所区别的。内部节点只包含[key，block pointer]对，在数据查找时发挥路标的作用，block pointer 就是指存储块的逻辑地址。叶子节

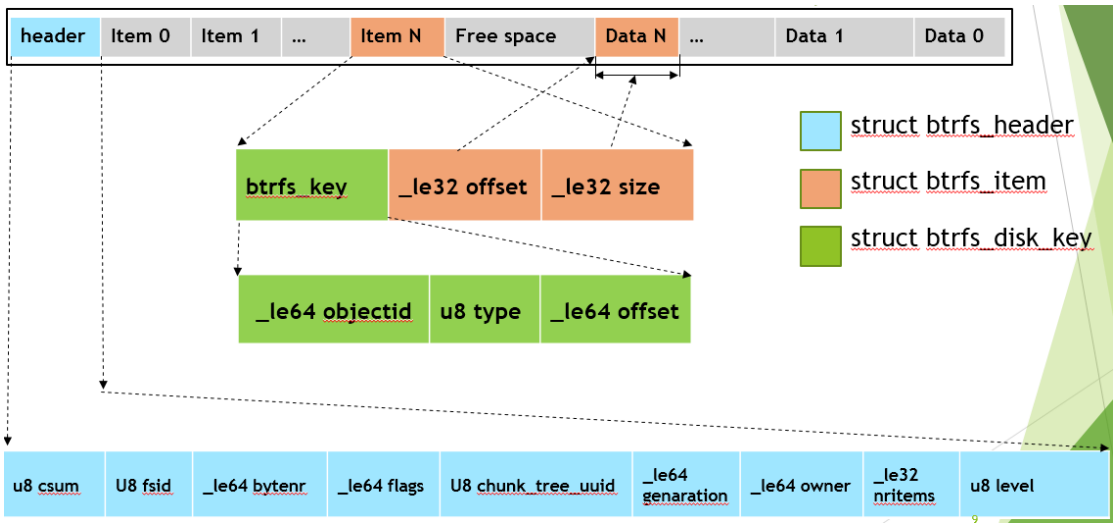

图 2.2.1 叶子节点结构

点则存储具体的 item，具体结构我们通过图 2.2.1 来清楚的展现。图最上面黑色边框圈出来的就是一个叶子节点，在节点开始是 block header，接下来就是 item。每个 item 其实包括两部分，如图中所示，完整的 item 1 其实由 Item1 和 Data1 构成。考虑到每个 item 的前面这一部分是定长的，不管是什么类型的 item，这部分结构是一样的，只是里面填充的数据不同而已，而第二部分 Data 对于不同不同类型的 Item，甚至是同一种类型的 item 长度都有可能不同。Btrfs 文件系统采用了如图所示的存储方式，定长的那部分紧邻着存储，而变长的数据从后往前存储。这样设计的一大好处就是，当需要遍历这个叶子节点中的所有 items 时，可以很方便的定位到每个 item 以及数据。

Block header 主要包括 9 个字段，每个字段所占用的空间大小已经在图中标示（单位：

bit）。Csum 标示这一块的内容的校验和，uuid 标示那个文件系统拥有这一块，该块处于树中的哪一层用 level 标示。当数据被读取的时候，这些字段使得元数据的内容能够被验证。Generation 是在块被插入到树中的时候设置的，对应于分配了这个块的事务的 ID，这个数据使得增量备份变得容易，同时还被用在写时复制事务中。

Btrfs_disk_key 就是前面提到的三大子结构中的 key，它主要用来指定 item 属于哪个对象，存储的是什么类型的数据。属于哪个对象，用 64 位的 objectid 来指定，btrfs 文件系统中的每个对象（这些对象将在下一节“文件系统层面的数据对象”中进行介绍）都有一个 objectid。比如，一个文件就拥有一个 objectid。允许有多种类型的 item 与这个对象相关，比如 inode_item，inode ref 等等，这些类型都是通过 key 中的 type 字段指定。而 key 中的最后 64 位空间所发挥的作用就不能一概而论，它是与 item 的类型相关的。具体的关联关系，我们会在接下来的内容中进行详细的介绍。

2.3 文件系统层面的数据对象

- Inode：存储在 btrfs_indoe_item 结构中（见图 2.3.1），其中 key 中的 objectid 就是通常意义

图 2.3.1 inode_item 结构

上的 inode number，offset 值为 0，type 为 INODE_ITEM（==1）。Inode item 的 data 部分存储的就是文件或者目录的状态信息，具体包括：generation（也就是最近一个使用过该 inode 的事务的 ID），文件大小，链接数，uid，gid，操作模式 mode，标志 flags，创建时间 ctime，修改时间 mtime 等等。

- File：对于小文件，也就是占用空间比一个叶子块还小的文件，系统会直接把文件内容内嵌在元数据 file_extent_item 里面，叫做 inline file。这种时候，item 中的 key 中的 offset 字段存储的就是这段数据在文件中的偏移地址，而 item 中的 size 字段表明内嵌了多少数据。每个小文件可以有多个这种 extent item。对于更大的文件，则采用正常的存储手段。也就是说，分配 extents 存储文件的数据，每个分配出去的 extent 都有一个 btrfs_file_extent_item 表示，记录有这个 extent 的 generation number 以及[逻辑偏移，块数目]对，表明这个文件存储在磁盘上的范围。Extents 还会将这个 extent 的逻辑偏移以及所使用块的数目存储在磁盘上的这个 extent 中。这样做使得 btrfs 修改一个 extent 的中间部分时不需要先把旧文件数据读出来。文件数据的校验和存储在 checksum tree 里，再具体点说，就是存储在 btrfs_csum_item 中。Btrfs_csum_item 的 key 中的 offset 这个时候存储的是对应的 extent 的字节偏移，而 data 部分则是一个校验和序列，因为每个数据块都有一个校验和（目前长度为 4 字节）。数据的校验和在压缩或者加密操作值之后

计算，反映的是写到磁盘的数据状态，因此它用来验证从磁盘读取的数据是否与之前写到磁盘的一致。Btrfs_csum_item 只用来存储文件数据 extents 的校验和，前面说到的内嵌在 file_extent_item 中的小文件并不需要，因为，这个时候数据的校验和已经被 block header 中的校验和覆盖。

■ 目录：btrfs 中的目录同时有两种索引方式，一种是为了文件的快速查找，另一种索引则是为了满足 Linux 文件系统的标准调用 readir，即能按照 inode number 的顺序遍

图 2.3.2 dir_item 的结构

历目录下的文件。因为这种顺序与磁盘上数据块的顺序更接近，通常能提供更好的批量数据读取性能。第一种索引，每个目录项对应一个 dir_item，具体结构如图 4 所示。Objectid 也就是目录的 inode，key 中的 offset 存储的则是该目录下某个文件名的哈希值，哈希方法在 superblock 中指定。dir_item 存储在 FS-tree 上，因此文件搜索自然是利用 b-tree 进行的。第二种索引为每个目录项增加一个 dir_index item，与 dir_item 的不同之处在于，key 中的 offset 存储的则是 inode sequence number。每个目录有一个自己的 inode sequence number，每次有新文件或者目录在此目录下创建时，它的值就增加 1。因而这个值的顺序大致上能代表文件数据在磁盘上的顺序。

还有很多其他的对象，这里不再一一详细介绍了，详细情况可以通过我们引用的参考资料了解。通过对这几个非常重要的对象的介绍，大家应该已经了解，btrfs 文件系统是如何使用统一的内部结构来实现不同功能的树。

3. 写时复制

Btrfs 的数据一致性相关的特性主要是靠 cow 事务技术来保证的。首先要解释一下 cow 事务，cow 是 copy on write 的缩写，顾名思义，它指的是每次写磁盘数据时，先将数据载入到内存中，更新数据后写入一个新的 block。那么事务是什么？cow 只能保证单一数据更新的原子性，但文件系统中很多操作需要更新多个不同的元数据，比如创建文件需要修改以下这些元数据：首先修改 extent tree，分配一段磁盘空间；然后创建一个新的 inode，并插入 FS Tree 中；最后还要增加一个目录项，插入到 FS Tree 中。整个这一完整的过程叫做一个事务。这种不直接在原始位置更新数据避免了一些突发状况，比如突然停电了。

介绍完 cow 事务，接着我们要介绍基于 COW 的 b-tree ，它是 Btrfs 最重要的数据结构方法。它主要思想使用了标准 b+ tree 的构造方法，但也存在不同的地方：1）删除叶节点之间的链接；2）使用一个自顶向下的更新过程；3）用懒惰引用计数进行空间管理。下面用一棵简化的树来介绍具体一个事务的过程。

如图 3.1 所示，假定每个节点占据一个 block，每个节点用一个整数代表一个 key，没有实际的 data。黄色代表没有被改变的页，绿色代表通过 cow 事务创建页。比如我们需要在最右边的叶节点中添加一个新的 key，首先将为了找到这个叶节点所经过的父节点都读到内存里，然后对于这块的数据进行更新，现在的情况是加入一个新的 key，再更新相关的数据结构指向这个节点。所有更新完成后，将这些写入磁盘新的 block，并把旧的 block 删除。如果要在节点中删除一个 key，操作过程也是一样的。如上一节介绍的，btrfs 整个结构如图所示，下面就介绍基于此结构的简化版 btrfs 中的 cow 事务。

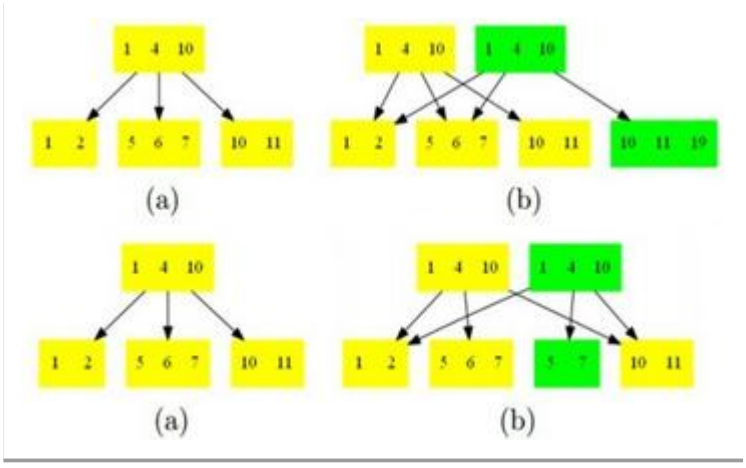

图 3.1

如图 3.2 所示，A 代表结构图中 tree of root tree 里指向 FS Tree 的根节点，B 代表 FS Tree，C 代表 inode tree，E 代表 dictionary tree。我们要在 Btrfs 添加一个新的 inode，它将会被插入到节点 C。首先，btrfs 将 inode 插入一个新分配的 block C ’中，并修改上层节点 B，使其指向新的 block C ’；修改 B 节点也将引发 COW 事务，以此类推，引发一个连锁反应，直到最顶层的 Root A。当整个过程结束后，新节点 A ’变成了 FS Tree 的根。但此时事务并未结束，superblock 依然指向 A。因为 inode 的增加会带来 dictionary tree 的改变，所以我们接着修改目录项（E 节点），同样引发这一过程，从而生成新的根节点 A ’’。此时将 superblock 依然指向 A ’’，整个事务结束，将更新后的节点写会新的磁盘空间。

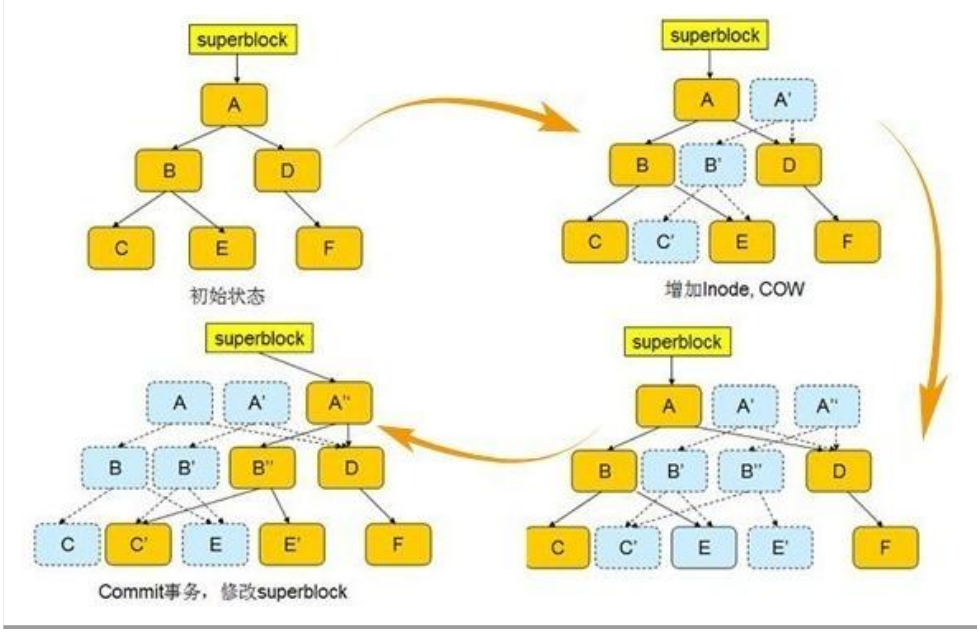

图 3.2

如图 3.3 所示为用户对文件进行写操作时 btrfs 中发生的 cow 事务，图(b)代表更新后结果。为了简单起见，图中省略了 chunk tree，device tree 和 data relo tree，绿色仍然代表更新后的块。首先写操作一定会改变 data extent 的内容，这就会回溯导致 FS tree 的更新。另外写操作还会带来 extent 的分配，这就导致 extent tree 的更新，此外写操作后 checksum 也会改变，所以 checksum tree 也要更新，具体更新的过程依旧遵照 cow 事务进行，在内存中完成。当所有更新完成后被写回到新的磁盘空间。

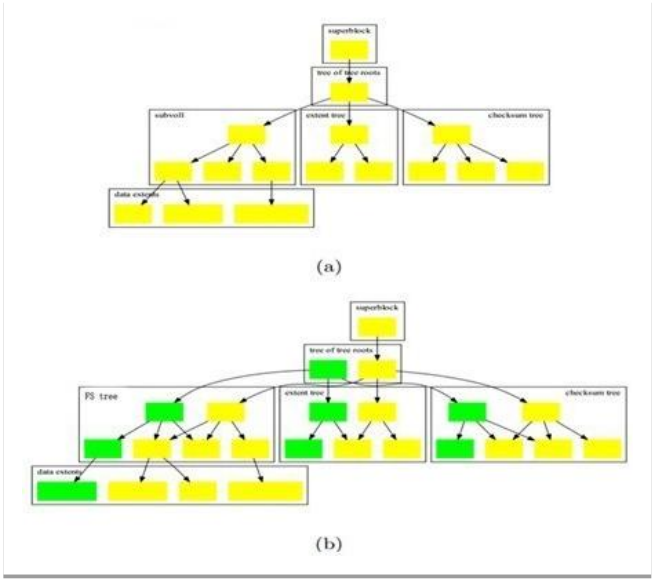

图 3.3

4. 克隆与快照

所谓快照，指的是对文件系统某一时刻的完全备份。建立快照之后，对文件系统的修改不会影响快照中的内容。这是非常有用的一种技术。比如数据库备份。假如在时间点 T1，管理员决定对数据库进行备份，那么他必须先停止数据库。备份文件是非常耗时的操作，假如在备份过程中某个应用程序修改了数据库的内容，那么将无法得到一个一致性的备份。因此在备份过程中数据库服务必须停止，对于某些关键应用这是不能允许的。

利用快照，管理员可以在时间点 T1 将数据库停止，对系统建立一个快照。这个过程一般只需要几秒钟，然后就可以立即重新恢复数据库服务。此后在任何时候，管理员都可以对快照的内容进行备份操作，而此时用户对数据库的修改不会影响快照中的内容。当备份完成，管理员便可以删除快照，释放磁盘空间。快照一般是只读的，当系统支持可写快照，那么这种可写快照便被称为克隆。克隆技术也有很多应用。比如在一个系统中安装好基本的软件，然后为不同的用户做不同的克隆，每个用户使用自己的克隆而不会影响其他用户的磁盘空间。非常类似于虚拟机。

Btrfs 支持快照和克隆，这个特性极大地增加了 btrfs 的使用范围，用户不需要购买和安装昂贵并且使用复杂的卷管理软件。下面简要介绍一下 btrfs 实现快照的基本原理。如前所述 Btrfs 采用 cow 事务技术，cow 事务结束后，如果不删除原来的节点，那么那些节点依然完整的表示着事务开始之前的文件系统。这就是快照实现的基本原理。

如图 4.1 所示，下面看一下快照实现的具体过程。建立一个新节点 Q，指向 P 节点所指的块。现在 P,Q 共享数据，当数据发生变化时，这棵树就会开始分裂，每棵树就会拥有自己的数据。

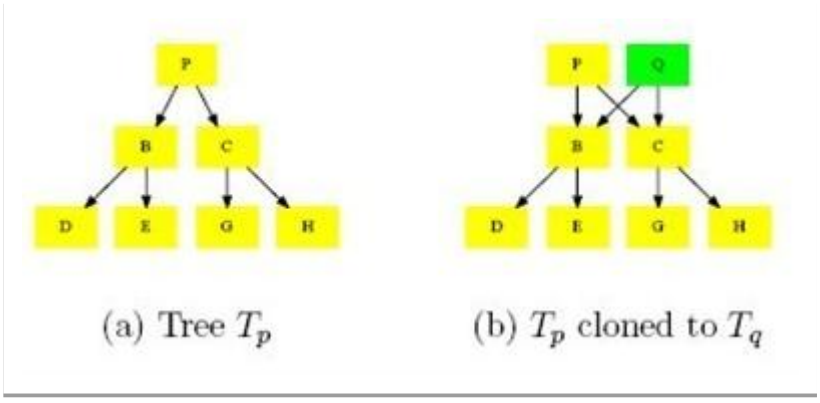

图 4.1

快照和克隆还引入了引用计数的问题。由于树的节点对于多个根节点来说都是可达的，所以有必要对空间回收再利用。 引用计数能并且被用来记录有多少个指针指向这些树的节点，Btrfs 采用引用计数决定是否在事务 commit 之后删除原有节点。对每一个节点，btrfs 维护一个引用计数。当该节点被别的节点引用时，该计数加一，当该节点不再被别的节点引用时，该计数减一。一旦这些计数器的值变为 0，那一个存储块就能被重用了。创建快照时，btrfs 复制根节点，并将其的引用计数设置为 2。在事务 commit 的时候，它的引用计数不会

归零，从而不会被删除，因此用户可以继续通过根节点访问快照中的文件。

只有 FS tree 是支持快照和克隆的，引用计数会被重新计数。其他树只进行普通操作，所以引用计数对他们不是必须的。下面就根据快照的例子来看引用计数的重新计算。对于普通的 Tree Root, 引用计数在创建时被加一，因为 Superblock 会引用这个 Root block。很明显，初始情况下这棵树中的所有其他节点的引用计数都为一。如图 4.2 所示，克隆之后，粉色的节点引用计数变为 2，他们的子节点保持不变。如果此时要在克隆的树中修改数据，如(d)所示，绿色仍旧代表 cow 事务操作创建的节点。C 的引用计数本为 2，分裂成两个节点后，C 与 C'的引用计数均为 1，G 的引用计数本为 1，现在因为被 C 与 C'引用，引用计数变成 2，H 通过 COW 后创建了一个新的节点 H'，H 与 H' 的引用计数均为 1。所有更新完成后，父节点指向 Q'，Q 的引用计数变为 0，会被删除。Q 被删除后是否会带来它子节点的删除，这就会引入节点删除算法。

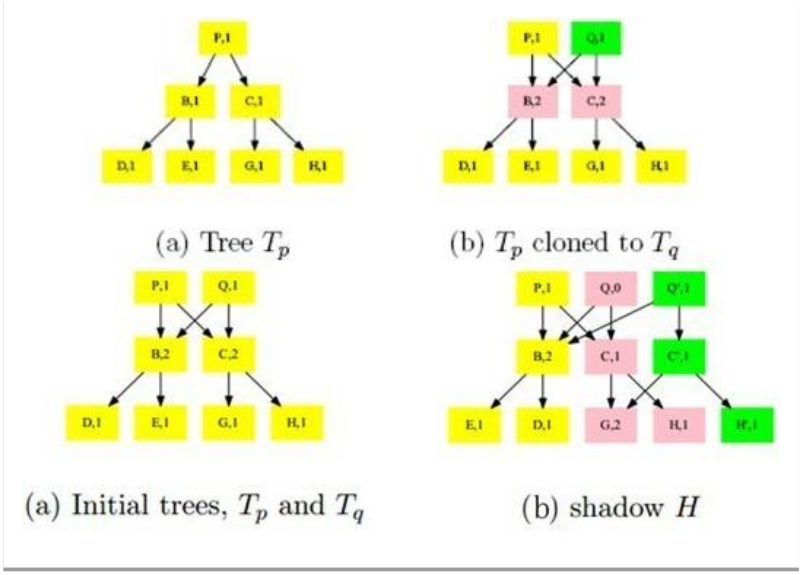

图 4.2

当要删除一棵树时，从根节点开始遍历，如果引用计数大于 1，那么将引用计数减 1 并且停止向下的遍历。如果引用计数为 1，将这个节点删除并继续向下遍历。要把 Q 这棵树删除。如图 4. 3 所示，从根节点 Q 开始，它的引用计数本为 1，现在变成了 0 所以需要删除，继续向下遍历。对于 C 节点，原来引用计数为 2，因为现在还被 P 引用所以引用计数变为 1，停止向下遍历。对于 X 节点，原来引用计数为 1，现在是 0，所以删除这个节点并继续向下遍历，Z 的引用计数从 1 变为 0，被删除。此时整个过程结束，最终结果如(c)所示。

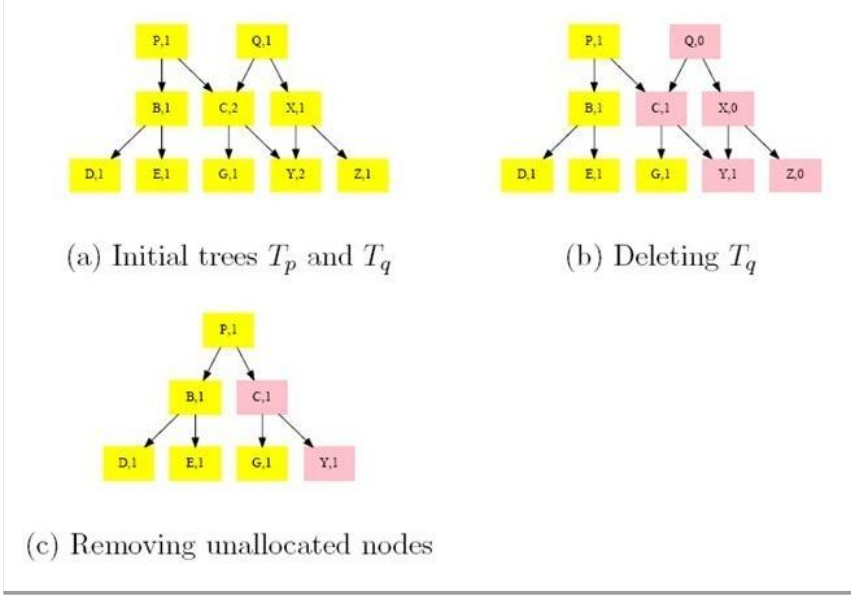

图 4.3

5. BTRFS 的多设备支持

BTRFS 在管理多个存储设备时需要提供对以下 10 项功能的支持：

- 镜像元数据，需要可配置为 N 份镜像（N>2）
- 在单独一个存储设备中镜像元数据
- 镜像数据 extent
- 用镜像恢复校验和错误
- 条带化数据 extent
- 在一个单独存储设备中混合使用镜像策略
- 高效地在不同设备间迁移数据
- 对存储设备高效地重配置
- 对每个子卷动态分配空间
- 对分配的空间记录所有的反向引用

Linux 系统中已经存在着一些管理存储设备的子系统，这些子系统被称为 device-mapper，这些系统的主要功能是管理未经处理过的存储设备，将它们合并映射到连续的虚拟块地址空间。它们可以支持镜像、条带化和 RAID 等功能。这类系统中典型的有 LVM,mdadm 等。

但是这类系统不能在 BTRFS 中用来管理多个存储设备，这是因为它们的错误恢复过程对于上层来说是完全透明的，如果使用这类系统，那么，BTRFS 将不能提供上述恢复校验和、高校迁移数据、条带化数据等功能。举例来说：当使用一个支持 RAID1 的存储设备管理子系统时，磁盘上存储的每份数据都有一份相应的备份，如果系统在读取一份数据时发现一个校验和错误，那么它就会到备用磁盘上去读取相应的正确数据，这一过程被这些设备管理子系统隐藏起来，文件系统无法知道自己读到的数据是源数据还是备份数据，因此，BTRFS 实现

了自己的设备管理子系统。

BTRFS 使用 chunk 来管理存储设备，chunk 分为逻辑 chunk 和物理 chunk 两种。逻辑 chunk 是用逻辑地址表示的一块存储空间，所有的 extent 都使用逻辑 chunk 的地址来访问存取数据，而不使用存储设备的物理地址。BTRFS 将存储设备分成许多物理 chunk，每个物理 chunk 的大小至少为 256M，一般是存储设备大小的 1%。文件系统为这些物理 chunk 建立起与逻辑 chunk 之间的映射，BTRFS 中用于映射逻辑 chunk 和物理 chunk 的是两棵 B 树——chunk tree 和 device tree，其中 chunk tree 用来维护从逻辑 chunk 到物理 chunk 的映射，device tree 用来维护从物理 chunk 到逻辑 chunk 的反向映射。这两棵树所占的空间很小，可以被存储在内存中以提高文件系统的运行效率。

每块逻辑 chunk 只能被一棵 extent allocation tree 拥有，并在 chunk 中存储了到这棵 extent allocation tree 的引用，被用来作为数据和元数据的存储空间。当被使用的存储空间增加时，越来越多的 chunk 被动态添加到 extent allocation tree 中，在这种情况下，extent allocation tree 将会分配比一块 chunk 更小的空间，以此来充分利用每块 chunk 之内未被利用的部分存储空间。

逻辑 chunk 的使用提供了更高效的 chunk 重分配策略，当要移动某块数据时，可以从 extent allocation tree 中查找这块数据所拥有的逻辑 chunk，并从 chunk tree 中查找这些逻辑 chunk 所对应的物理 chunk 地址，然后将这些物理 chunk 上的数据从源地址移动到目标地址，并修改从逻辑 chunk 到物理 chunk 的引用及其反向引用，不用修改文件系统中其他部分到这块 chunk 的引用（因为逻辑 chunk 并没有改变）。每当新的存储设备被加到文件系统中时，这块设备上的存储空间不仅可以被用来当作一块新的未分配的存储池，而且还可以使用上述 chunk 重分配的策略将之用来条带化文件系统中已存在的数据，以使文件系统中的数据均衡存储在各个存储设备中。表 5.1 是在 RAID1 级别下增加一块新的存储设备时，chunk 的条带化过程。

(a) two disks	Logical chunks	Disk1	Disk2	
	L1	C11	C21	
	L2	C12	C22	
	L3	C13	C23	
(b) disk added				Disk3
	L1	C11	C21	
	L2	C12	C22	
	L3	C13	C23	
(c) rebalance				
	L1	C11	C21	
	L2		C22	C12

	L3	C13		C23

表 5.1 增加一个条带化 Disk 的过程：L1,L2,L3 为逻辑 chunk,Cij 为第 i 块存储设备上第 j 块物理 chunk

在初始情况下，一个逻辑 chunk 分别对应两个 Disk 上的物理 chunk，逻辑 chunk L1 由相同大小的物理 chunk C11 和 C21 实现，这两份数据互为备份，L2 和 L3 亦如此。当 Disk3 被添加进来的时候，逻辑 chunk L2 和 L3 的各自一份物理 chunk 被移动到 Disk3 中，这样，L1，L2，L3 的逻辑地址并没有被修改，被修改的只有其对应的物理 chunk，而文件系统中的其他部分使用的地址都是逻辑 chunk 的地址，因此，不必修改其他部分对该部分地址的引用即可条带化文件系统中的数据。

同时，逻辑 chunk 的使用也降低了重新镜像的开销，当一个镜像存储设备被损坏的时候，文件系统将查询 device tree 以确定这块设备上的物理 chunk 所对应的逻辑 chunk，然后独立地修复每块逻辑 chunk，通过这样的方法，文件系统仅需要恢复那些真正被使用的数据块。

6. 碎片整理

在 BTRFS 中，磁盘的碎片整理问题有两种截然不同的实现方法。在较为简单的一种实现方法中，为了整理一个文件，首先读取这个文件，然后使用 Copy-On-Write 的方法写回。因为文件真正写到磁盘上去是等到下一个 checkpoint 到来时才会执行的，而且在 checkpoint 时，存储分配器将会尽可能将数据存储到相邻的一些 extent 中，所以，在大部分情况下，这样的方法会工作得很好，但是这种方法的一个缺点是通过整理，数据之间的共享关系被丢失了。这样，虽然会减少磁盘的碎片，但是，被整理的数据所占用的空间或许反而变得更大了。在一些情况下，更需要一个维护这些共享关系的更加复杂的碎片整理实现。

进行磁盘的碎片整理或从一个磁盘上移除数据的过程是一个复杂的过程，然而，如果忽视数据之间的共享关系，将会增大数据占用的磁盘空间，这与磁盘碎片整理的目的——降低数据的存储空间相反。这一过程的步骤一般分下以下三步：

1. 通过移动 chunk 的方法移出所有有效的 extent。
2. 寻找对所需移动的所有 chunk 的引用。
3. 在维护共享关系的情况下修复数据对 chunk 的引用。

在进行移动 chunk 过程时，同样用到了 Copy-On-Write 的方法。下面通过一个例子来详细解释这一过程：图 6.1 是移动一个 chunk 的初始状态：

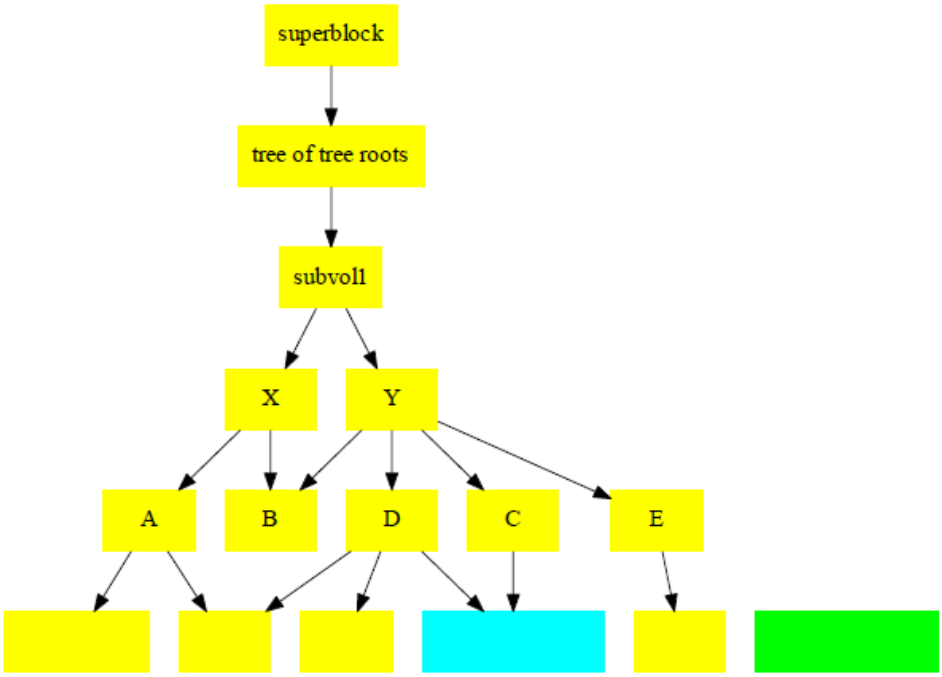

图 6.1 要将蓝色 extent 的数据移动到绿色的 extent 上

根据要移动的数据的反向引用找到所有直接或间接使用这些 extent 的树，并将其存在一个名为 backref_cache 的有向无环图数据结构中。在这之后，所有引用了要移动的块的子卷都将被复制，复制过程如下：

1. 冻结该卷的状态
2. 用 COW 的方法复制跟节点，然后解冻，允许其他操作继续进行。

这一步骤的结果如图 6.2 所示：

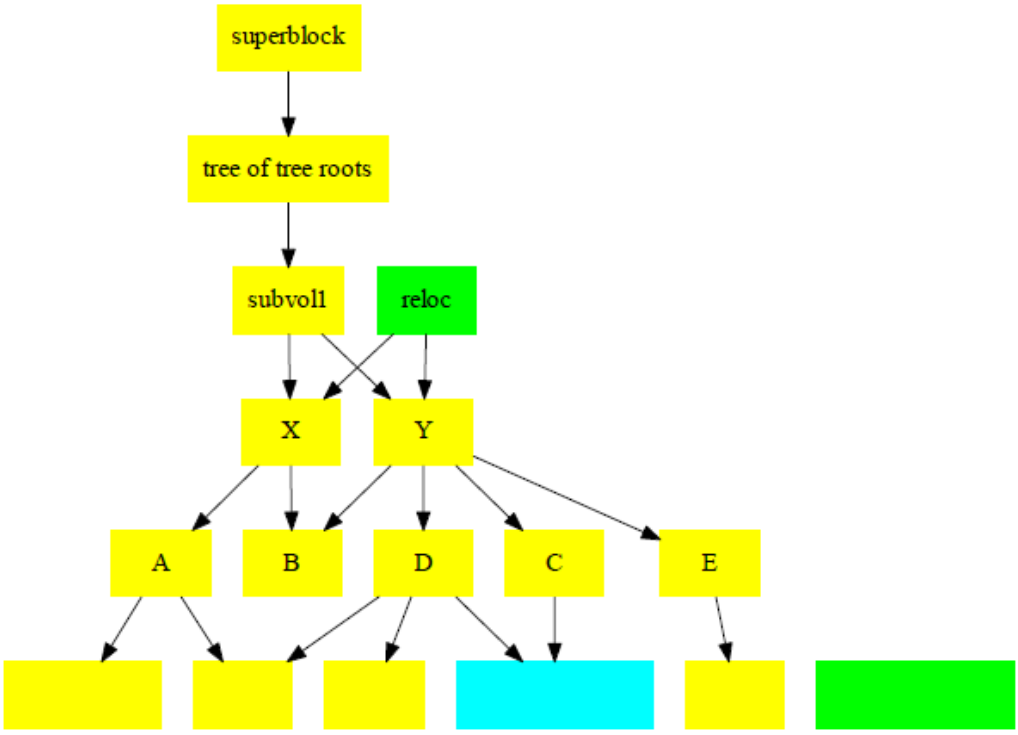

图 6.2 复制了子卷之后

复制出来的新的树为 reloc tree。接着，所有引用了要移动的块的数据都被以 COW 的方法更新到 reloc tree 中，结果如图 6.3 所示：

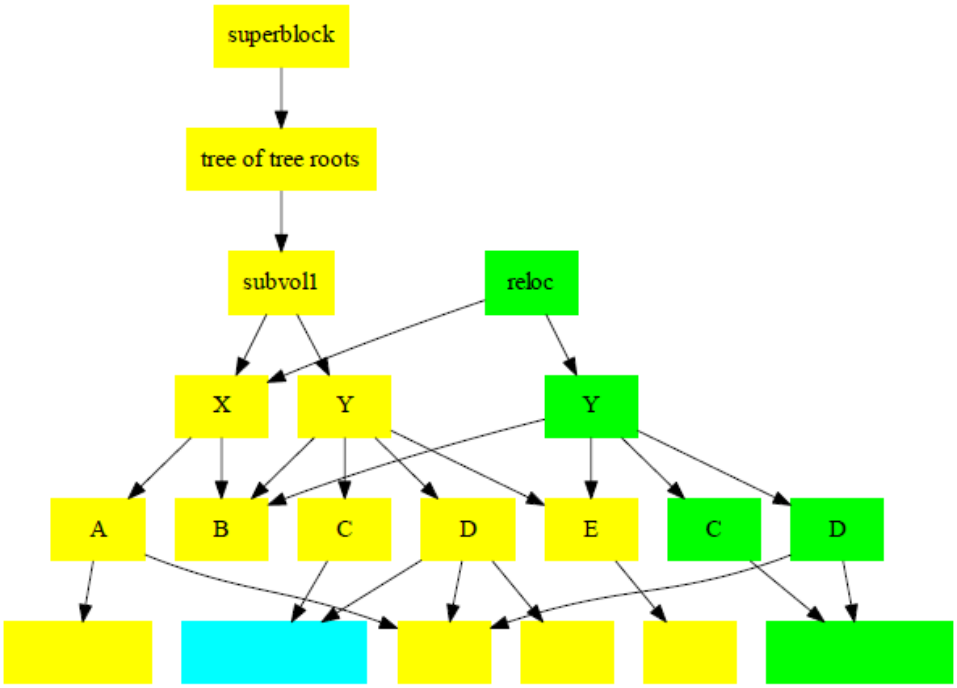

图 6.3 移动数据之后

最后一步是将 reloc tree 和原来的 fs tree 合并。方法为遍历整棵树，寻找那些在 reloc

tree 中被修改的子树，然后用这些已修改的子树替换掉原来树中未修改的部分，得到一棵新的 fs tree，这棵 fs tree 所使用的存储空间是新的 extent，最终，释放不需要的 reloc tree。所得最终结果如图 6.4 所示：

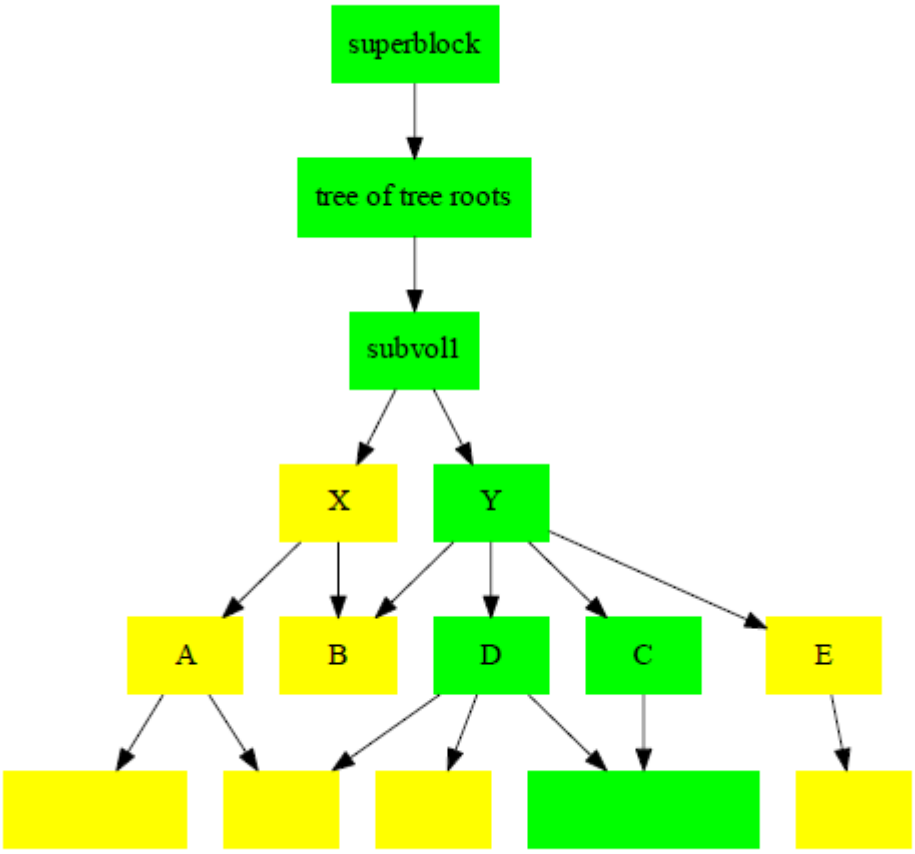

图 6.4 合并后的结果

通过以上步骤，文件系统新的树形结构已经存储在内存中了，并且拥有正确的共享结构，它仍然和原来的树占用同样大小的存储空间，当把这棵树的数据结构写到磁盘上去时，由于写数据时，文件系统将会尽量将数据写在相同，或相邻的 extent 中，然后把这棵树原先占用的旧的 extent 删除，这样，减少了文件系统的磁盘碎片，实现了磁盘碎片的整理。

参考资料：

[1] https://btrfs.wiki.kernel.org/index.php/

[2] Ohad Rodeh, Josef Bacik, Chris Mason. BTRFS: The Linux B-Tree FileSystem, IBM research report, 2012

[3] http://www.ibm.com/developerworks/cn/linux/l-cn-btrfs/新一代文件系统 btrfs 简介