

Lecture15 File System Crash Consistency

1. 文件系统崩溃一致性

文件系统崩溃一致性

- 文件系统保存了多种数据结构
- 各种数据结构之间存在依赖关系与一致性要求：
 1. inode中保存的**文件大小**，应该与其索引中保存的数据块个数相匹配
 2. inode中保存的**链接数**，应与指向其的目录项个数相同
 3. 超级块中保存的**文件系统大小**，应该与文件系统所管理的空间大小相同
 4. 所有inode分配表中标记为**空闲**的inode均未被使用；标记为**已用**的inode均可以通过文件系统操作访问

突发状况(Crash)可能会造成这些一致性被打破

创建文件时崩溃的六种情况

	文件系统结构			是否影响使用	典型问题
	inode 位图	inode 结构	目录项		
情况 1	持久	未持久	未持久	否	空间泄漏
情况 2	未持久	持久	未持久	否	无
情况 3	未持久	未持久	持久	是	信息错乱
情况 4	未持久	持久	持久	是	信息错乱
情况 5	持久	未持久	持久	是	信息错乱
情况 6	持久	持久	未持久	否	空间泄漏

崩溃一致性：用户期望

重启并恢复后...

1. 维护文件系统数据结构的内部不变量
 - 没有磁盘块既在freelist中也在一个文件中
2. 仅有最近的一些操作没有保存到磁盘中
 - 用户只需要关心最近的几次修改还在不在
3. 没有顺序的异常

文件系统操作所要求的三个属性

```
creat("a"); fd = creat("b"); write(fd,...); crash
```

持久化/Durable: 哪些操作可见

a和b都可以

原子性/Atomic: 要不所有操作都可见, 要不都不可见

要么a和b都可见, 要么都不可见

有序性/Ordered: 按照前缀序(Prefix)的方式可见

如果b可见, 那么a也应该可见

崩溃一致性保障方法

我们下面要具体介绍的几种方法

1. 同步元数据写+fsck
2. 日志
3. 写时复制
4. Soft updates

2. 同步元数据写+FSCK

- 同步元数据写
 - 每次元数据写入后, 运行sync()保证更新后的元数据入盘
- 若非正常重启, 则运行fsck检查磁盘, 具体步骤:
 1. 检查superblock
 - 如果出错, 则尝试使用superblock的备份
 - e.g. 保证文件系统大小大于已分配的磁盘块总和
 2. 检查空闲的block
 - 扫描所有inode的所有包含的磁盘块
 - 用扫描结果来检测磁盘块的bitmap
 - 对inode bitmap也用类似的方法
 3. 检查inode的状态
 - 检查类型: 如普通文件、目录、符号连接等
 - 若类型错误, 则清除掉inode以及对应的bitmap
 4. 检查inode链接

- 扫描整个文件系统树，核对文件链接的数量
- 如果某个inode存在但在任何一个目录，则放到 `lost+found` 中

5. 检查重复磁盘块

- 如：两个inode指向同一个磁盘块
- 如果一个inode明显有问题则删掉，否则复制磁盘块一边给另一个

6. 检查坏的磁盘块ID

- 如：超出磁盘空间ID
- 问：这种情况下，fsck能做什么呢？仅仅是移出这个指针嘛？

方法1：把这个指针改成指向全0块

方法2：将这个指针移出，后面的指针往前挪

7. 检查目录

- 这是fsck对数据有更多语义的唯一一种文件
 - 保证 `.` 和 `..` 是位于头部的目录项
 - 保证目录的链接数只能是1个
 - 保证目录中不会有相同的文件名
- fsck的问题：太慢
 - **fsck需要用多长时间？**
 - 对于服务器70GB磁盘（2百万个inode），需要10分钟
 - 时间与磁盘的大小成比例增长
 - **同步元数据写导致创建文件等操作非常慢**
 - 例：解压Linux内核源代码需要多久？
 - 创建一个新文件需要8次磁盘写，每次10ms
 - Linux内核大概有6万个源文件
 - $8 \times 10\text{ms} \times 60000 = 1.3\text{小时}$




3. 日志

步骤

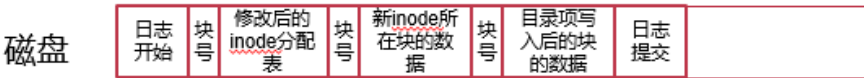
1. 在进行修改之前，先将修改记录到日志中
2. 在所有要进行的修改都记录完毕后，提交日志
3. 此后再进行修改
4. 修改之后，删除日志

创建"/新文件"的修改包括：

- 1. 标记inode为占用
- 2. 初始化inode
- 3. 将目录项写入目录中

 崩溃随时可能发生！

在"日志提交"写入存储设备之前崩溃
➢ 恢复时发现日志不完整，忽略日志，"/新文件"未被创建
在"日志提交"写入存储设备之后崩溃
➢ 将日志中的内容，拷贝到对应位置，"/新文件"被创建成功



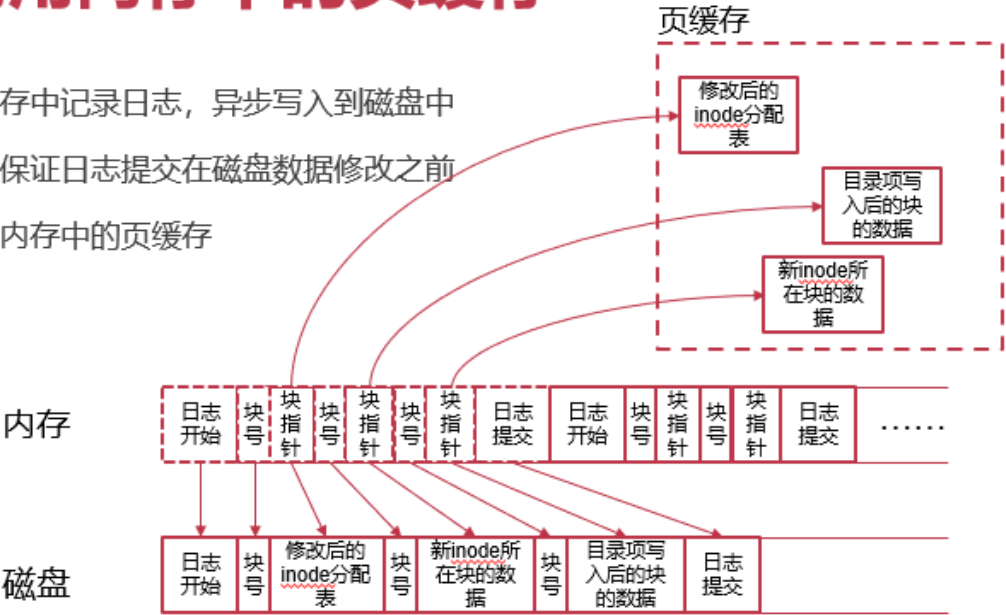
在内存中进行上述操作的同时，在磁盘上记录日志

- Q：这种方法有什么问题？
- A1：每个操作都需要写磁盘，内存缓存优势被抵消
- A2：每个修改需要拷贝新数据到日志
- A3：相同块的多个记录被修改多次

解决方案

利用内存中的页缓存

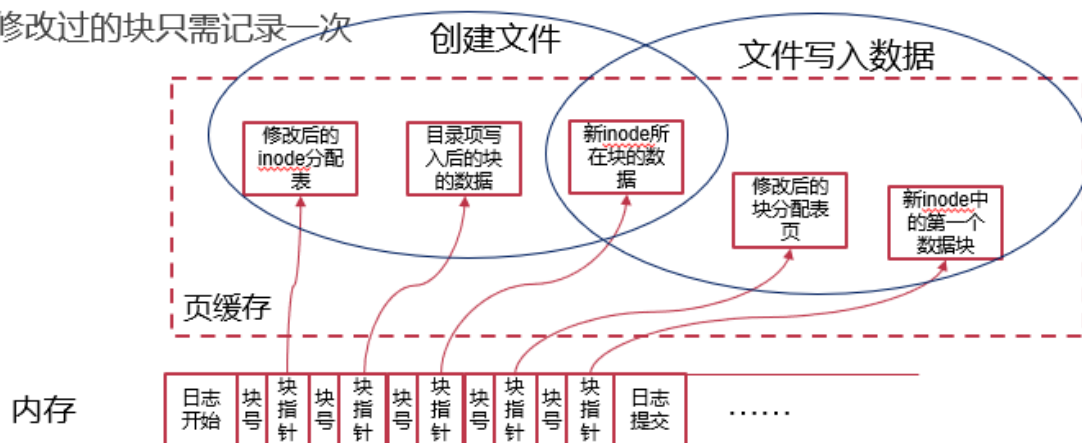
在内存中记录日志，异步写入到磁盘中
仅需保证日志提交在磁盘数据修改之前
利用内存中的页缓存



批量处理日志以减少磁盘写

多个文件操作的日志合并在一起

每个修改过的块只需记录一次



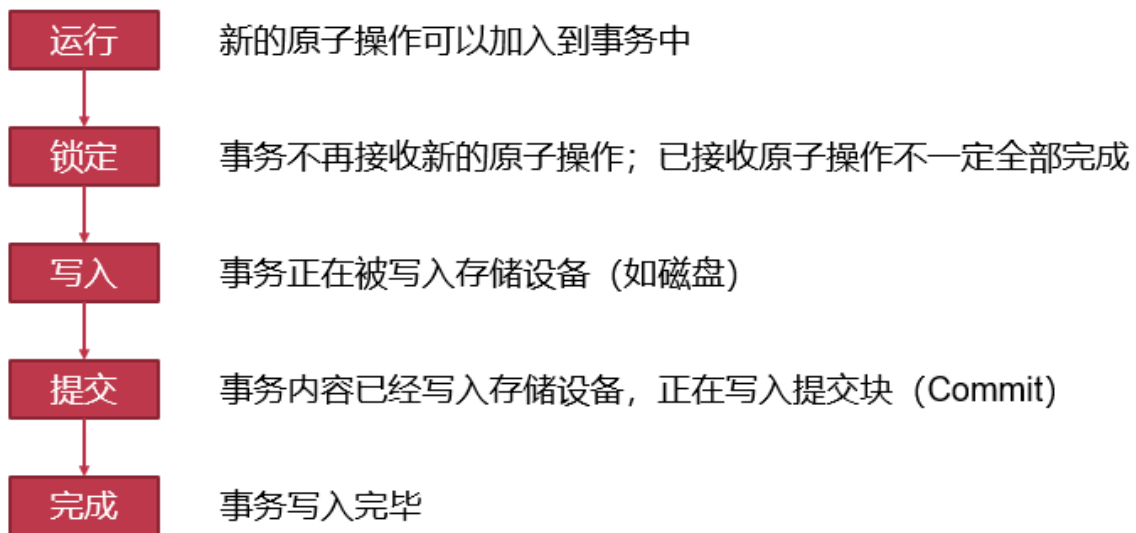
日志提交的触发条件

- 定期触发
 - 每一段时间 (如5s) 触发一次
 - 日志达到一定量 (如500MB) 时触发一次
 - 用户触发
 - 例如: 应用调用fsync()时触发
1. 使用内存中的页缓存
 - 在内存中记录日志, 异步写入到磁盘中
 - 保证日志提交在磁盘数据修改之前
 2. 批量处理日志以减少磁盘写
 - 多个文件操作的日志合并在一起
 - 每个修改过的块只需记录一次
 3. 日志提交出发条件设置
 - 定期触发: 每一段时间/日志大小达到一定量后提交
 - 用户触发: 应用调用fsync()时触发

Case: Linux中的日志系统JBD2

Journal Block Device 2

- 通用的日志记录模块
 - 日志可以以文件形式保存
 - 日志也可以直接写入存储设备块
- 概念
 - Journal: 日志, 由文件或设备中某区域组成
 - Handle: 原子操作, 由需要原子完成的多个修改组成
 - Transaction: 事务, 多个批量在一起的原子操作
- JBD2事务的状态



这里的事务内容应该是指Log中的记录

Ext4的三种日志模式

Writeback Mode: 日志只记录元数据

最快, 但是一致性最差!



Ordered Mode: 日志只记录元数据+数据块在元数据日志前写入磁盘

默认模式



Journal Mode: 元数据和数据均使用日志记录

一致性最好, 但数据写入两次!



思考一下: 三种模式各自有何问题和优势?

1. Writeback Mode: 快, 但是一致性差, 可能元数据指向了错误的数
2. Journal Mode: 最保险, 但是所有写入都要写两遍, 对大文件比较差
3. Ordered Mode: 一种平衡的模式, 能保证元数据不会指错, 但是数据部分可能会出现不一致 (比如写了一个write操作只持久化了一半, 另一个操作的数据写完了, 但是元数据没有修改)

Ordered Mode保证了数据的持久化在metadata持久化之前

- Ordered Mode: 两次Flush保证顺序

应用程序

数据

文件系统

数据

元数据

J元数据

Jcmt

缓存

数据

J元数据

Flush

Jcmt

Flush

元数据

磁盘

盘片

元数据

数据

J元数据

Jcmt

磁盘也有缓存, 积累到一定程度后写入到磁盘中。

这样子做是为了减少磁盘读写的次数, 提高磁盘的寿命。

1. 第一个flush的作用: 保证在Journal commit之前将数据和Journal(元数据)持久化到了磁盘中。否则由于磁盘的缓存的存在, Journal commit操作可能在J(元数据)之前, 若commit后crash掉了, 则会导致日志文件恢复时出现inconsistency (误以为已经写入了J(元数据)), 这是我们所无法接收的。
2. 第二个flush的作用: 保证Journal commit在元数据写入之前。否则新的元数据会覆盖掉旧的元数据, 若在还未commit时crash掉了, 则在恢复时既没办法回滚 (旧的元数据被删掉了, 而且Filesystem根据此时的感知不会选择去从J(metadata)中读取旧数据覆写), 因此回滚后元数据还是新的, 不符合all-or-nothing的定义 (会造成inconsistency)。

In question, 再等一波回放...

Q: 如何优化掉flush?

A1: 第一个flush, 计算数据+Journal(元数据)的hash值加入到Journal(Cmt)中, 恢复时通过计算hash值来判断Journal(cmt)是否是当前版本的

A2: 第二个flush比较难去掉, 可以通过定时提交日志来降低flush对其的影响 (因为新的元数据在Cache中), 但是可能用着用着突然卡一下, 因为有太多东西提交。我们可以通过修改硬件, 在commit时发送一个中断, 提醒OS可以将元数据写入磁盘来解决这个问题。

- **权衡一致性和性能**

- 数据的数量大，只需要写入一次
- 元数据的数量少，写入两次相对可接受

- **可能出现的问题**

- 数据只有一份，若出现问题无法回退（all-or-nothing）
- 部分情况下，一致性还是可以保证的（如新增数据时）
- 部分情况下，数据会丢失，但元数据依然可以保证一致性

4. 写时复制

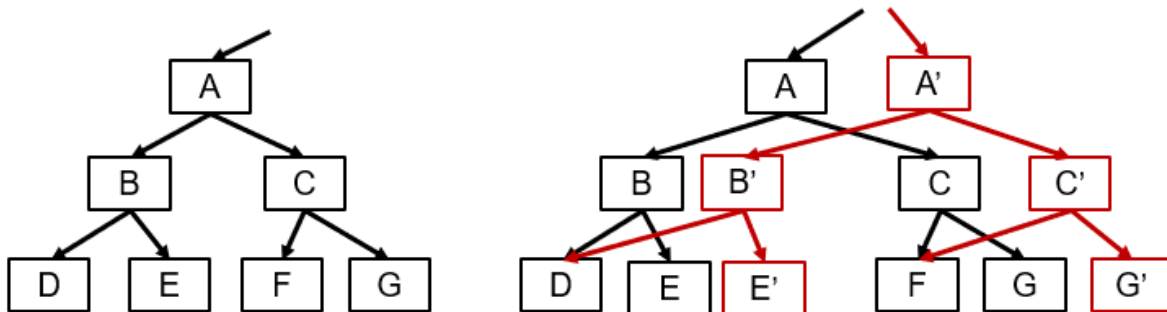
原子更新操作：Copy-on-Write

写时复制(Copy-on-Write)

- 在修改多个数据时，不直接修改数据，而是将数据复制一份，在复制上进行修改，并通过递归的方法将修改变成原子操作

因为最后只用改一个指针...

- 常用于树状结构



文件中的写时复制

- 文件数据散落在多个数据块内
- 使用日志：数据需要写两遍
- 写时复制保证多个数据块原子更新
 1. 将要修改的数据进行复制（分配新的块）
 2. 在新的数据上修改数据
 3. 向上递归复制和修改，直到所有修改能原子完成
 4. 进行原子修改
 5. 回收资源(Garbage Collection)

思考时间

Q1: 对于文件的修改, 写时复制一定比日志更加高效吗?

A1: 不一定, copy-on-write对于小的修改很低效 (至少要copy一个页)

Q2: 写时复制和日志各自的优缺点有哪些?

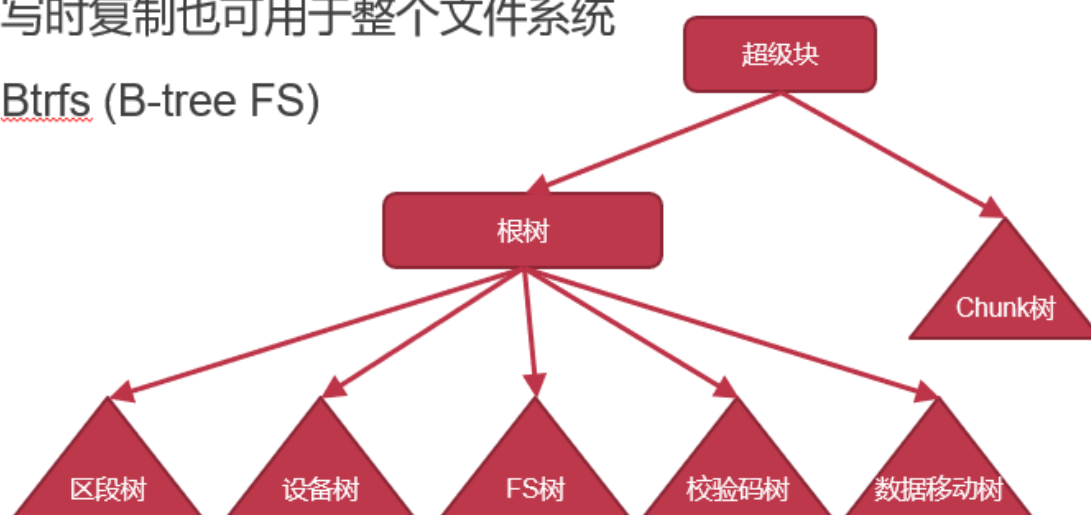
A2: copy-on-write的优点是不用写两遍, 只需要做一次更改, 缺点是小修改的性能较差; 日志的优点是小修改的性能较高, 缺点是要redo, 且平时做更改时要写两次

Q3: 能否只用写时复制来实现一个文件系统?

A3: 可以, B-tree FS

“写时复制”文件系统

- 写时复制也可用于整个文件系统
- Btrfs (B-tree FS)



5. Soft Updates

Soft Updates

- 一些不一致情况是良性的

合理安排修改写入磁盘的次序, 可避免恶性不一致的情况的发生

- 相对于其它方法的优势
 1. 无需恢复便可挂载使用
 2. 无需在磁盘上记录额外信息

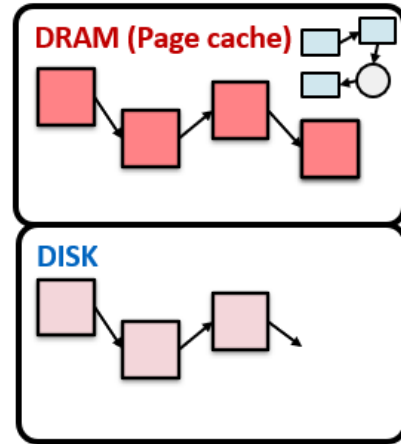
Soft Updates的总体思想

- 最新的元数据在内存中

- 在DRAM中更新，跟踪dependency
- ✓ DRAM 性能更好
- ✓ 无需同步的磁盘写

- 磁盘中的元数据总是一致的

- 在遵循dependency的前提下写入磁盘
- ✓ 一直能保证一致性
- ✓ 发生崩溃后，重启立即可用



Soft Updates的三个次序规则

1. 不要指向一个未初始化的结构

如：目录项指向一个inode之前，该inode结构应该先被初始化

2. 一个结构被指针指向时，不要重用该结构

如：当一个inode指向了一个数据块，这个数据块不应该被重新分配给其他结构

3. 不要修改最后一个指向有用结构的指针

如：Rename文件时，在写入新的目录项前，不应删除旧的目录项

依赖追踪

- Soft Update原理

- 使用内存结构表示还未写回到存储设备的修改，并异步地将这些修改写入存储设备中

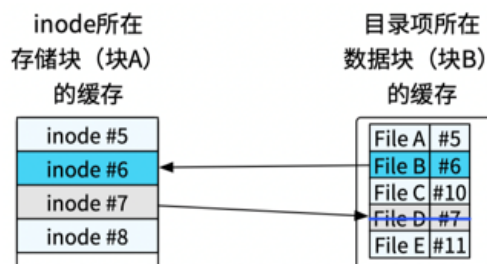
- 依赖追踪

- 根据3条规则，对修改之间需要遵守的顺序进行记录

如果修改A需要在修改B之前写入到存储，则称B依赖于A

- Soft update会将这些修改之间的依赖关系记录下来

依赖追踪的两个问题



- **问题1：环形依赖**

- 一个块通常包含多个文件系统结构
- 环形依赖：块 A 需要在块 B 前写回，同时块 B 需要在块 A 前写回

- **问题2：写回迟滞**

- 当一个结构中的数据被频繁修改时，该结构很可能由于一直产生新的依赖导致长时间无法被写回到存储设备之中

撤销和重做

- 解决环形依赖
 - 将依赖追踪从块粒度细化为结构粒度，使用撤销和重做打破循环依赖
- 记录每个结构上的修改记录
 1. 当需要将某个结构写回到存储设备时，检测是否有环形依赖
 2. 当出现环形依赖时，其先将部分操作撤销
 - 即将内存中的结构还原到此操作执行前的状态
 3. 撤销之后环形依赖被打破，根据打破后的依赖将修改按照顺序持久化
 4. 持久化完毕之后，将此前被撤销的操作恢复，即重做
 5. 在重做完成后，将最新的内存中的结构按照新的依赖关系再次持久化

不能让用户在使用过程中感知到撤销操作，应该对撤销的操作上锁，防止用户对其的访问