ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences

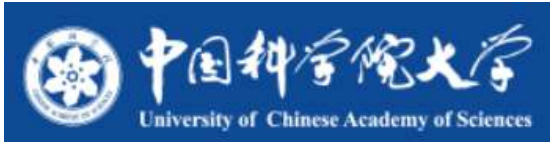
中国科学院大学
University of Chinese Academy of Sciences

文件系统崩溃一致性

李鹏

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，已获得原作者授权，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。

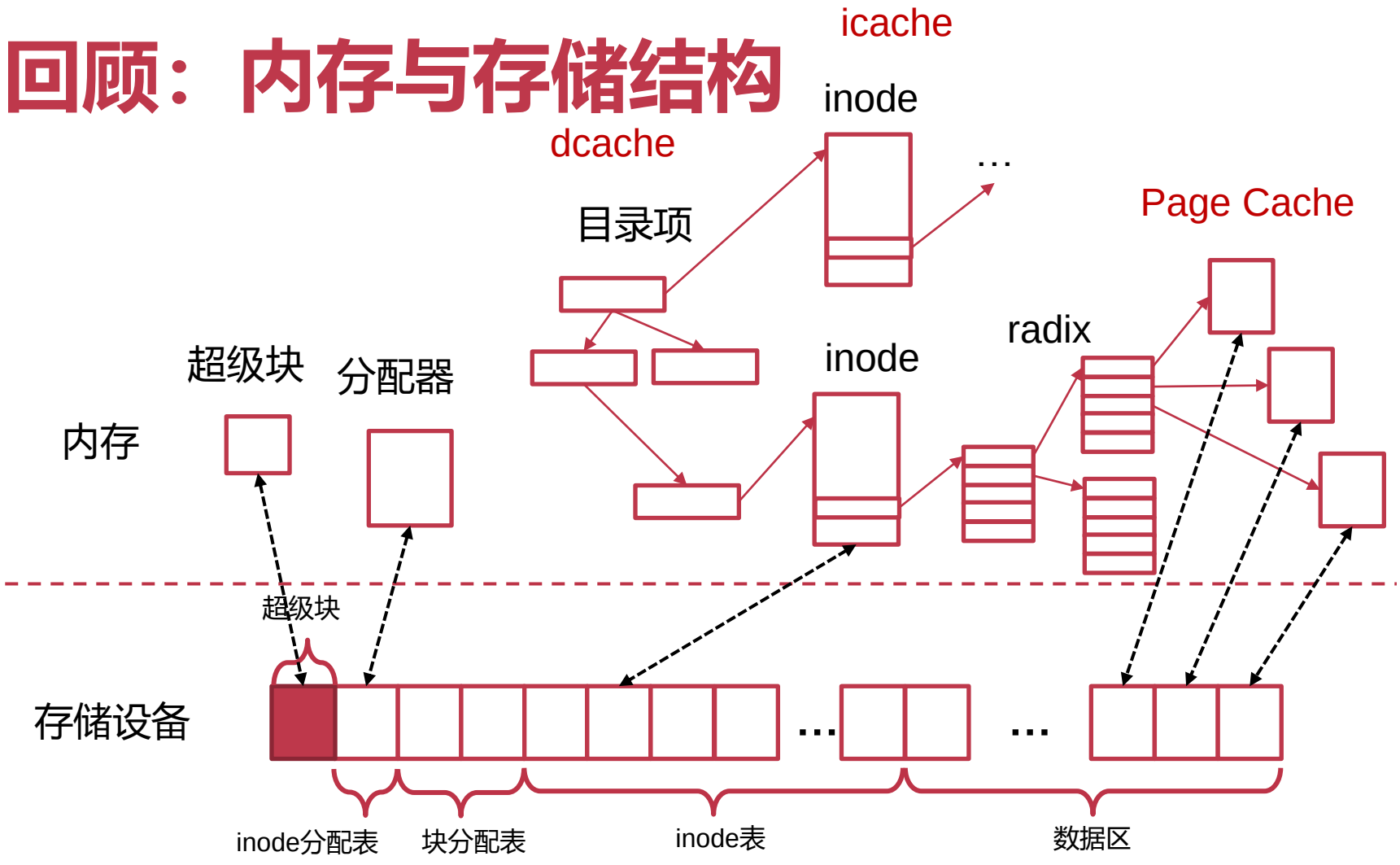


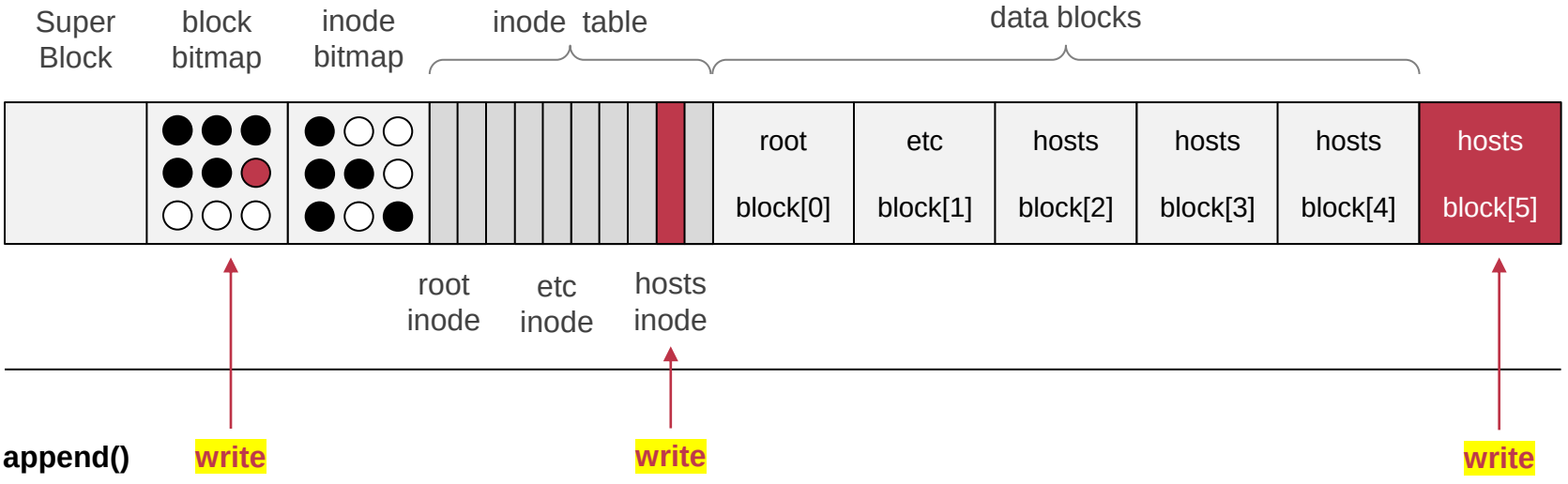
文件系统崩溃一致性

文件系统的崩溃一致性

- **文件系统中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**
  - 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
  - inode中保存的链接数，应与指向其的目录项个数相同
  - 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
  - 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（崩溃）可能会造成这些一致性被打破！**

回顾：内存与存储结构


icache
inode
dcache
…
目录项
Page Cache
radix
超级块
分配器
inode
内存
超级块
存储设备
…
…
inode分配表
块分配表
inode表
数据区



bitmap[5] = 1

inode of /etc/hosts (旧)

- size : 8000
- pointer : block[3]
- pointer : block[4]
- pointer : null

⇨

新的inode

- size : 9000
- pointer : block[3]
- pointer : block[4]
- pointer : block[5]

block[5] =

xxxxxxxxx
xxxxxxxxx
xxxxxxxxx

append包含三个磁盘写

1. 写入新数据
2. 写入新inode
3. 更新block位图

若写入过程发生故障，有6种可能：仅1个写成功（3种），仅2个写成功（3种）
可能的错误：数据错误、空间浪费

原子性丧失的后果

序号	Data Bitmap	inode	Data Block	后果
1	✓	✓	✓	☺
2	✓	✕	✕	Dead block
3	✕	✓	✕	Corrupt filesystem
4	✕	✕	✓	Random writes
5	✕	✓	✓	Corrupt filesystem
6	✓	✕	✓	Dead block
7	✓	✓	✕	Incorrect data
8	✕	✕	✕	☺

更复杂的情况

- 大文件多级索引的更新
- link/unlink
- 跨数据块的目录文件更新

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的不变量

   例如, 没有磁盘块既在free list中也在一个文件中

2. 仅有最近的一些操作没有被保存到磁盘中

   例如：我昨天写的OS Lab的文件还存在

   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

   $ echo 99 > result ; echo done > status

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

同步元数据写

- **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
- **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 x 10ms x 60000 = 1.3小时

崩溃一致性保障方法

- fsck
- 日志
- 写时复制
- Soft updates

原子更新技术

FSCK

fsck

若非正常重启，则运行fsck检查磁盘，具体步骤：

- **1. 检查superblock**
  - 例：保证文件系统大小大于已分配的磁盘块总和
  - 如果出错，则尝试使用superblock的备份
- **2. 检查空闲的block**
  - 扫描所有inode的所有包含的磁盘块
  - 用扫描结果来检验磁盘块的bitmap
  - 对inode bitmap也用类似方法

fsck

- **3. 检查inode的状态**
  - 检查类型：如普通文件、目录、符号链接等
  - 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存在但不在任何一个目录，则放到/lost+found
- **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一个

fsck

- **6. 检查坏的磁盘块ID**
  - 如：指向超出磁盘空间的ID
- **7. 检查目录**
  - 这是fsck对数据有更多语义的唯一的一种文件
  - 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

原子更新技术：Journaling

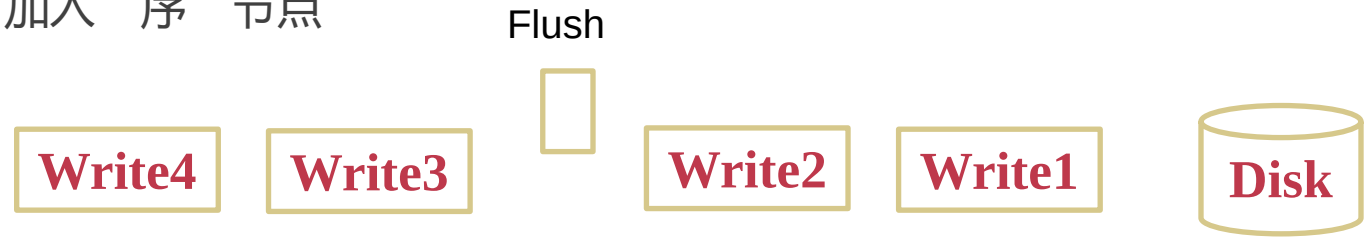
日志

崩溃恢复：Journaling

- **崩溃一致性的原因**
  - 磁盘读取不会导致崩溃一致性
  - 多次磁盘写入的非原子性
- **数据库ACID属性**
  - 日志
- **“原子性”的多次写入操作**
  - 首先将它们“原子性”地写入日志
  - 然后再正式写入数据
  - 如果发生崩溃，则重放日志

日志的原子性保证

- **flush原语**
  - flush之前的所有写操作都实际写入磁盘（落盘）
  - flush之后的所有写操作都不实际写入磁盘
  - 加入“序”节点

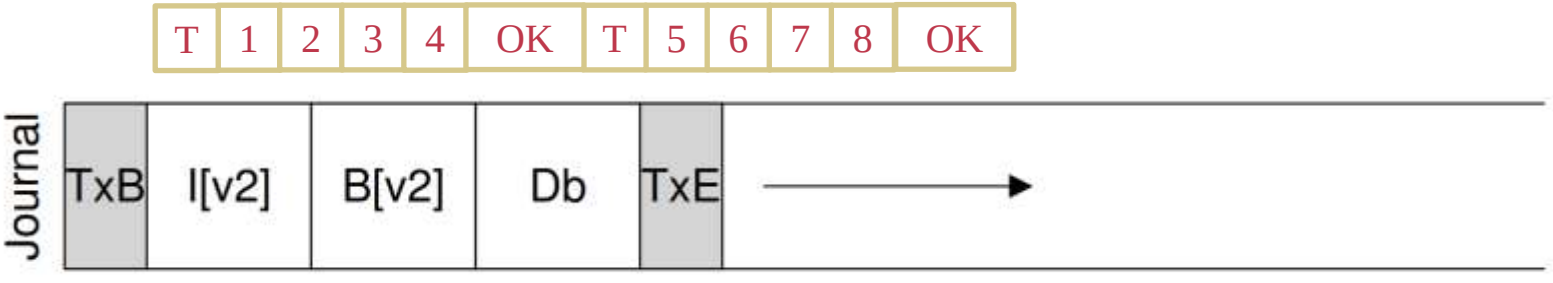


- **假设磁盘提供write（写入数据块）和sync（等待数据落盘）**
  - 能否实现一种crash-safe的数据结构?

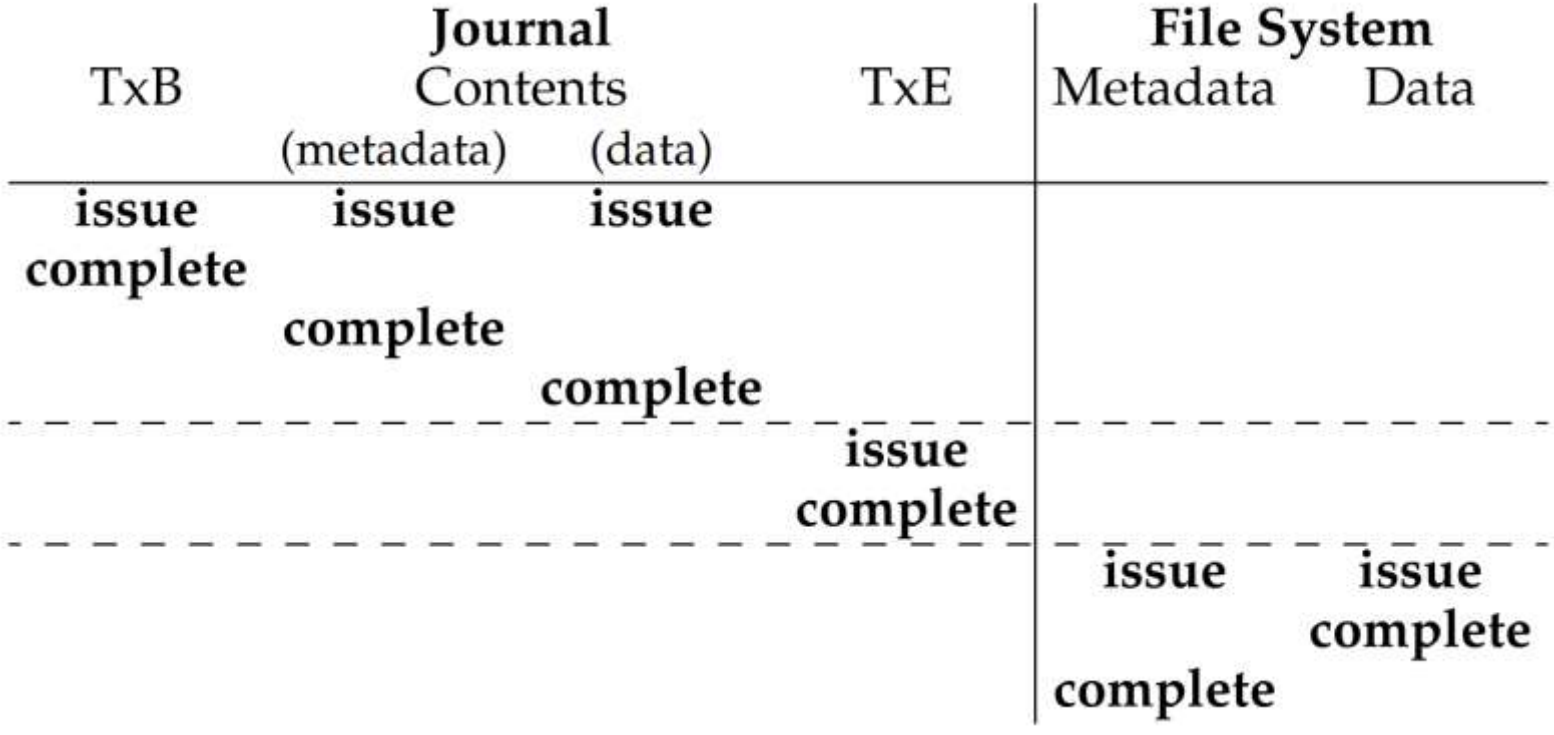
简化的问题：append-only的block vector

- **随机读取**
- **atomic-append (blocks)**

- **实现方法** 原子添加
  - 保证之前的数据落盘后才写入下一份数据

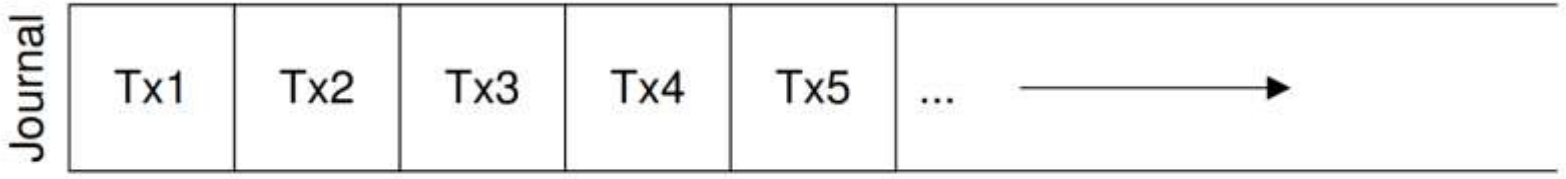


数据日志时间线



发生崩溃怎么办?

崩溃恢复

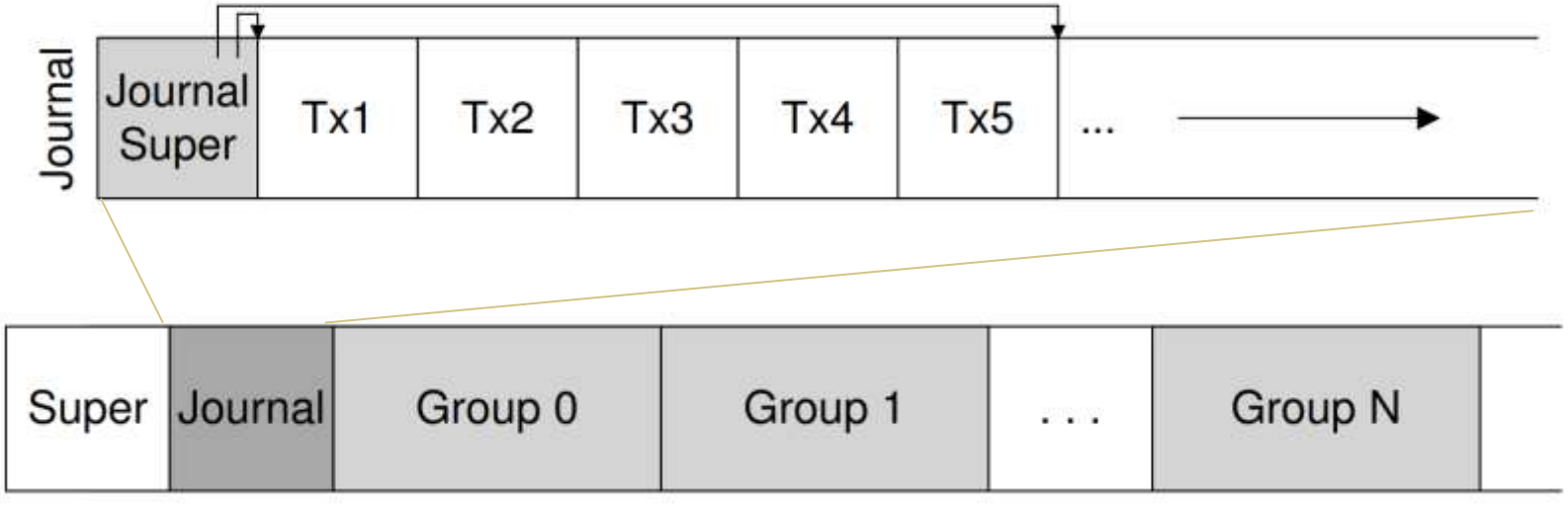


- **Replay所有已经提交的日志**

- **写入的过程**
  - 日志写入
  - 日志提交
  - 检查点
  - 释放

    问题：如果只记录一个日志是否可以?

循环日志（Circular Log）



- **日志超级块**
  - 哪些事务尚未加检查点
- **所有数据块都被写入两次**
  - 有没有其他选择?

性能问题

问题1. 每个操作都写磁盘，内存缓存优势被抵消
问题2. 每个修改需要拷贝新数据到日志
问题3. 相同块的多个修改被记录多次

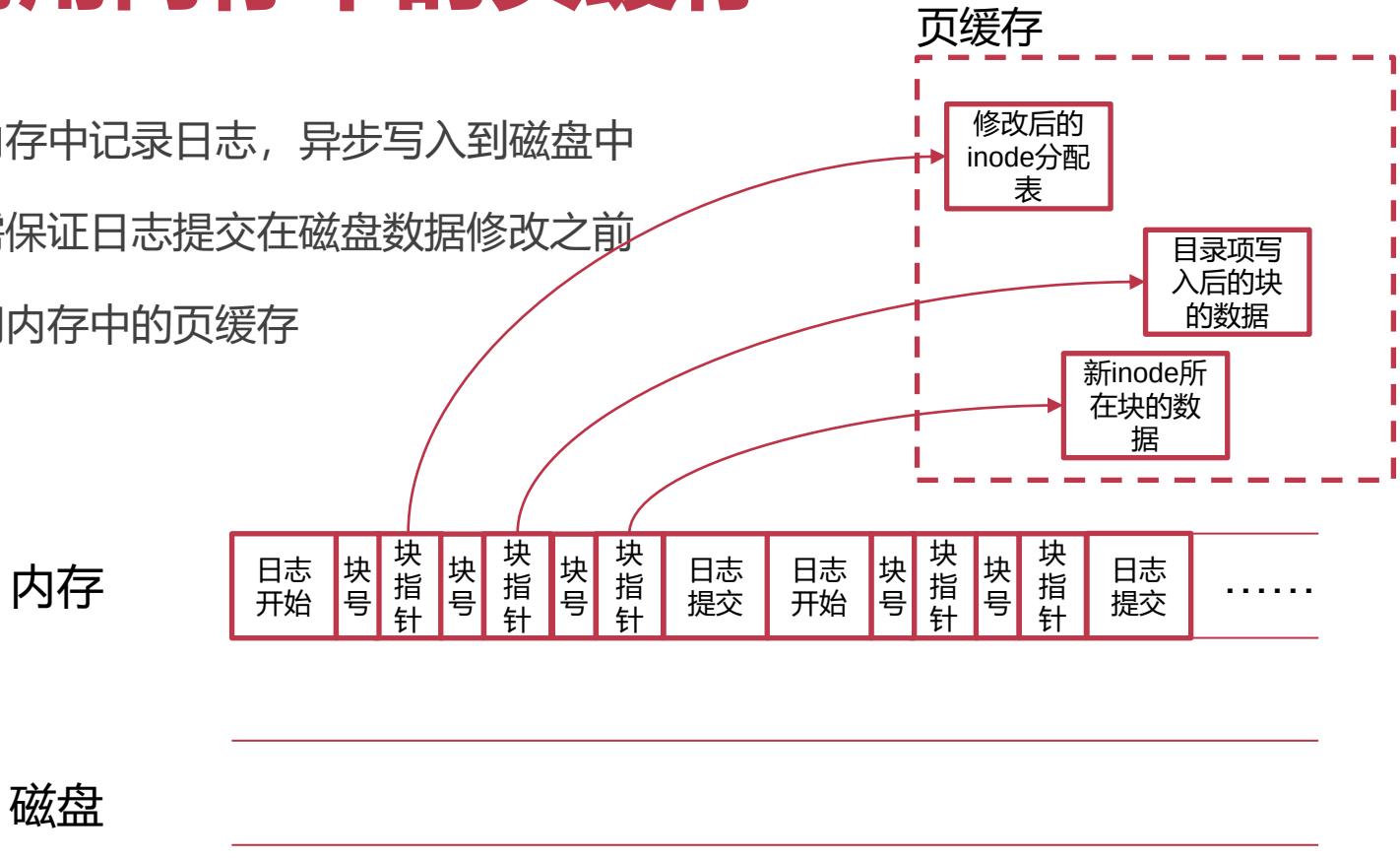
……

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

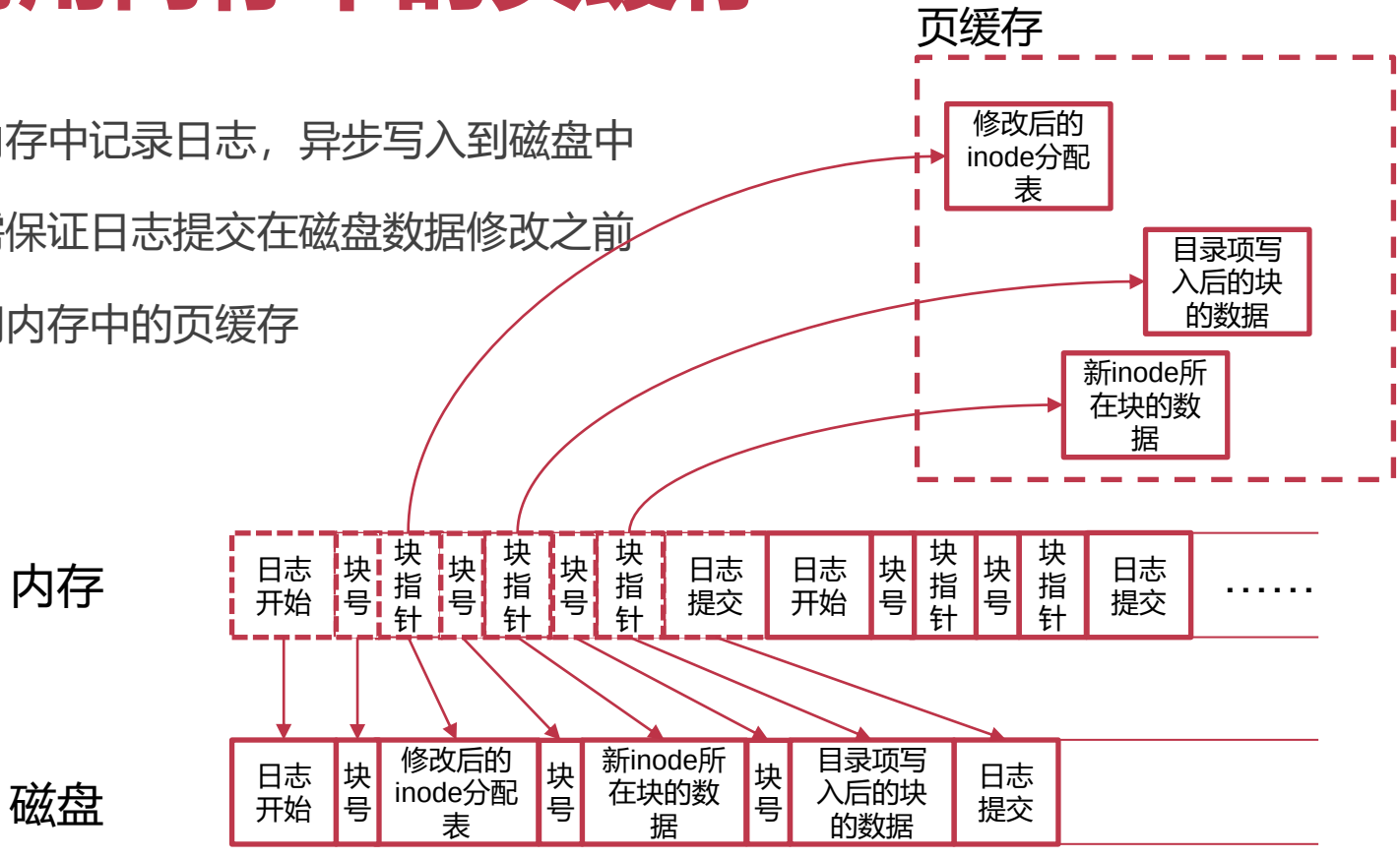


利用内存中的页缓存

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

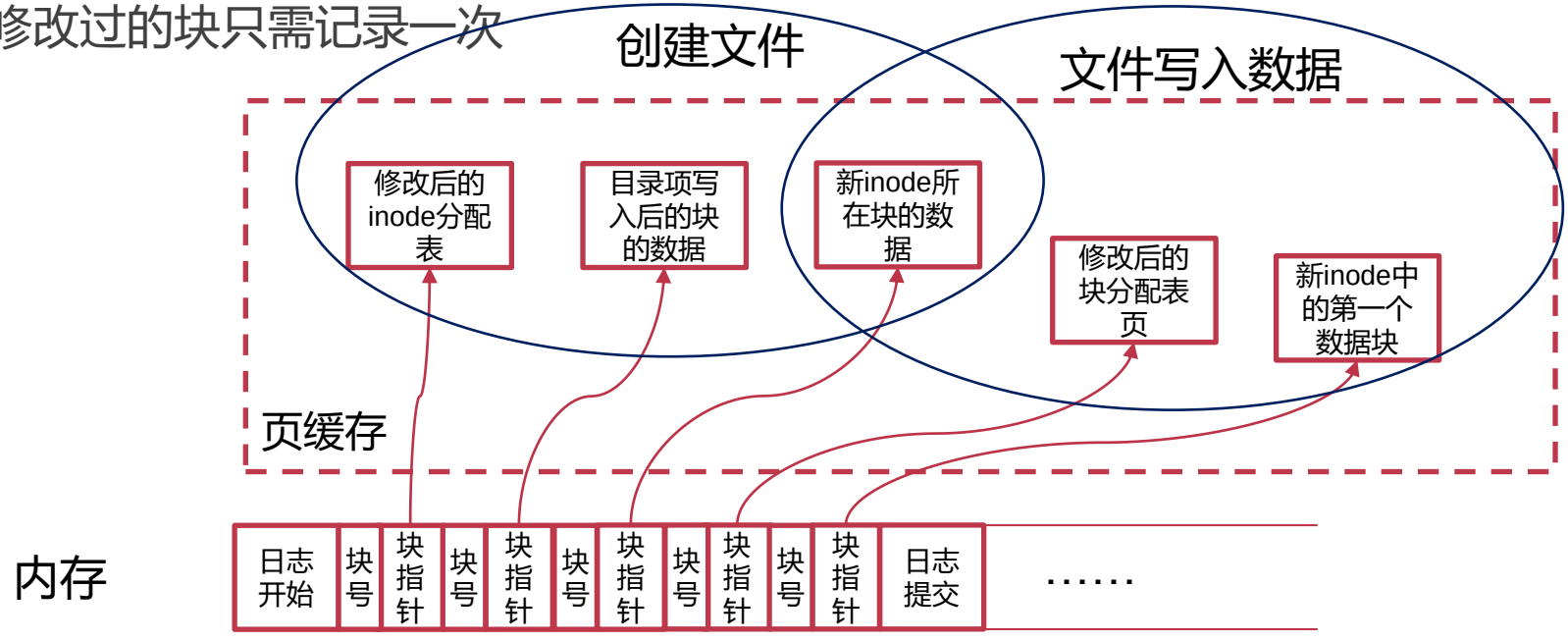
利用内存中的页缓存



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

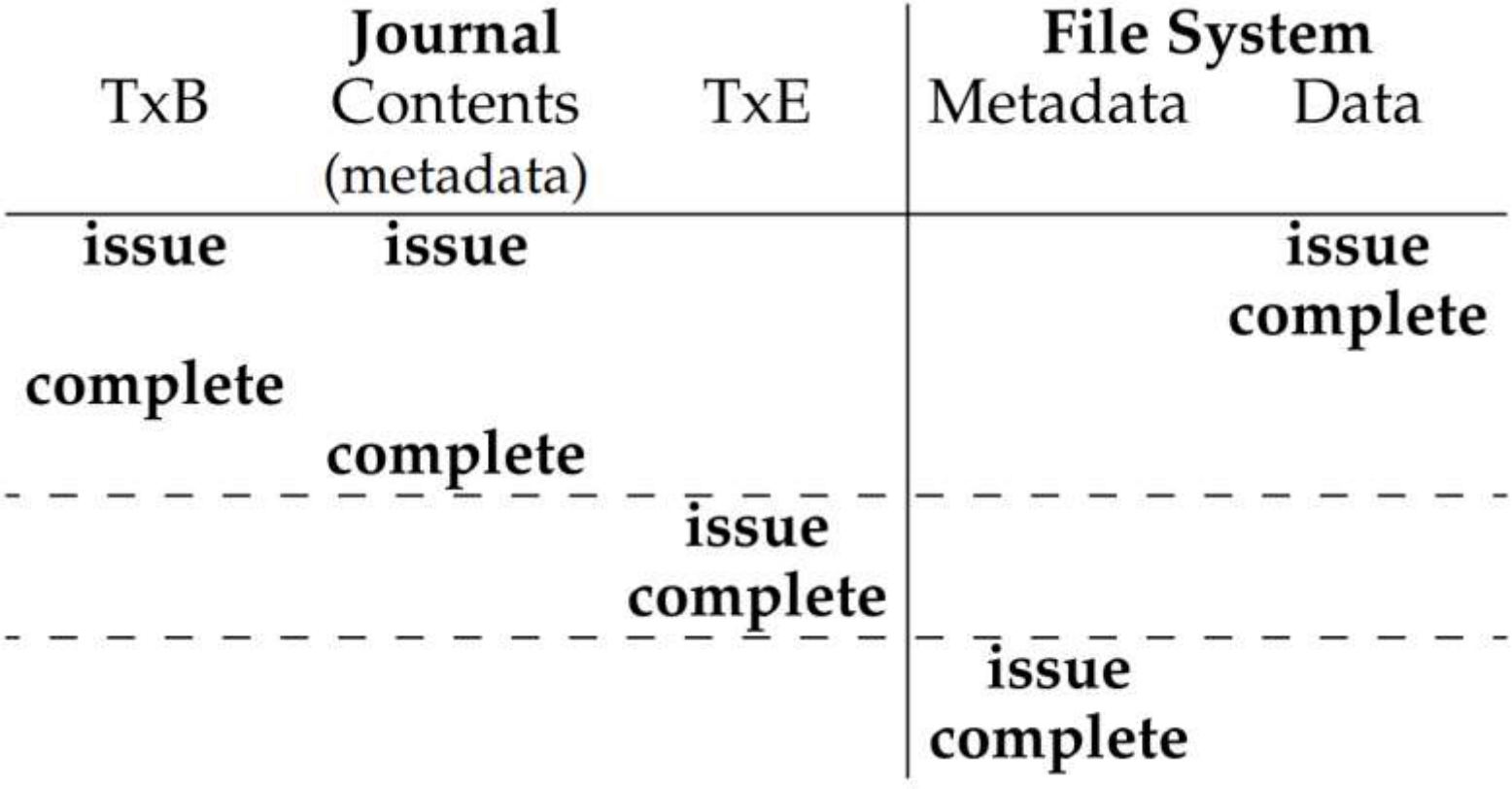
每个修改过的块只需记录一次



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

元数据日志


Journal
File System
TxB
Contents (metadata)
TxE
Metadata
Data
issue
issue
issue
complete
complete
complete
issue
complete
issue
complete

没有flush原语怎么办?

- **在起始块与结束块加入校验和**
  - 当校验和匹配时，日志条目有效
  - 当校验和不匹配时，日志条目无效
- **实现简单、高效**
- **性能更高**
- **EXT4文件系统**

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

新的原子操作可以加入到事务中

事务不再接收新的原子操作；已接收原子操作不一定全部完成

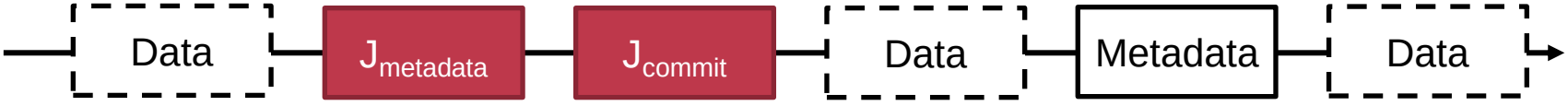
事务正在被写入存储设备（如磁盘）

事务内容已经写入存储设备，正在写入提交块（Commit）

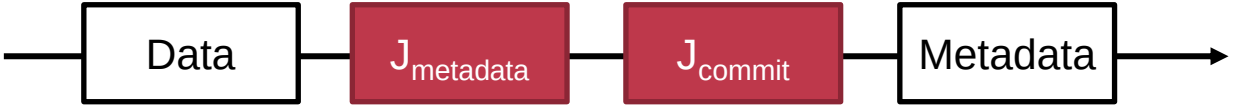
事务写入完毕

Ext4的三种日志模式

Writeback Mode：日志只记录元数据



Ordered Mode：日志只记录元数据+数据块在元数据日志前写入磁盘



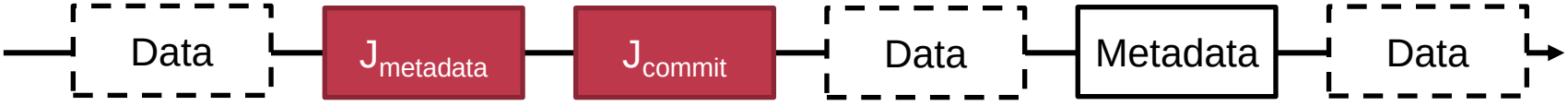
Journal Mode (Full Mode)：元数据和数据均使用日志记录

思考一下：三种模式各自有何问题和优势?

Ext4的三种日志模式

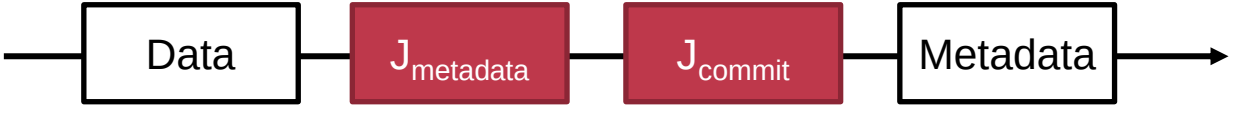
Writeback Mode：日志只记录元数据

最快，但是一致性最差！



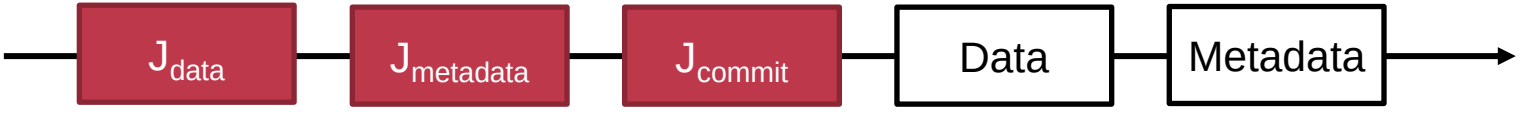
Ordered Mode：日志只记录元数据+数据块在元数据日志前写入磁盘

默认模式



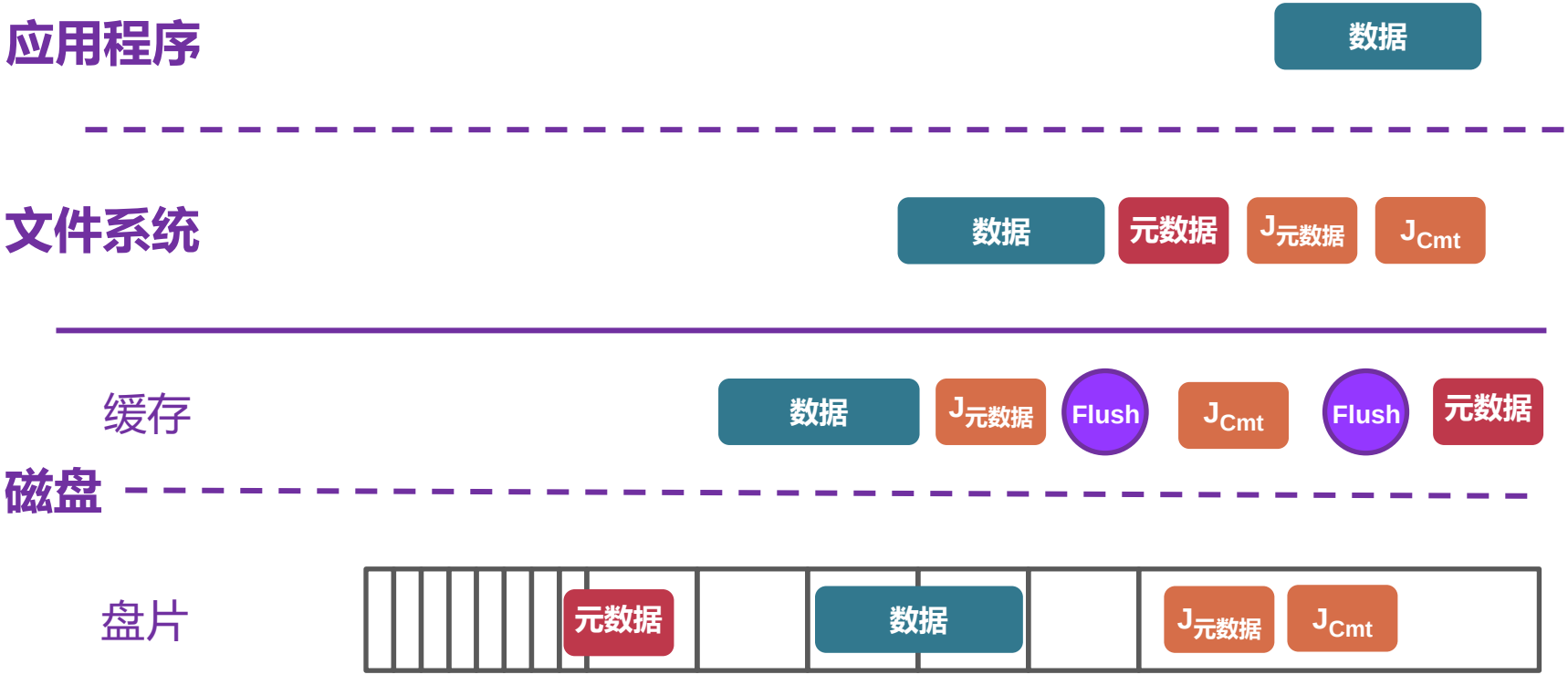
Journal Mode：元数据和数据均使用日志记录

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？

Ordered Mode：两次Flush保证顺序



Ordered Mode

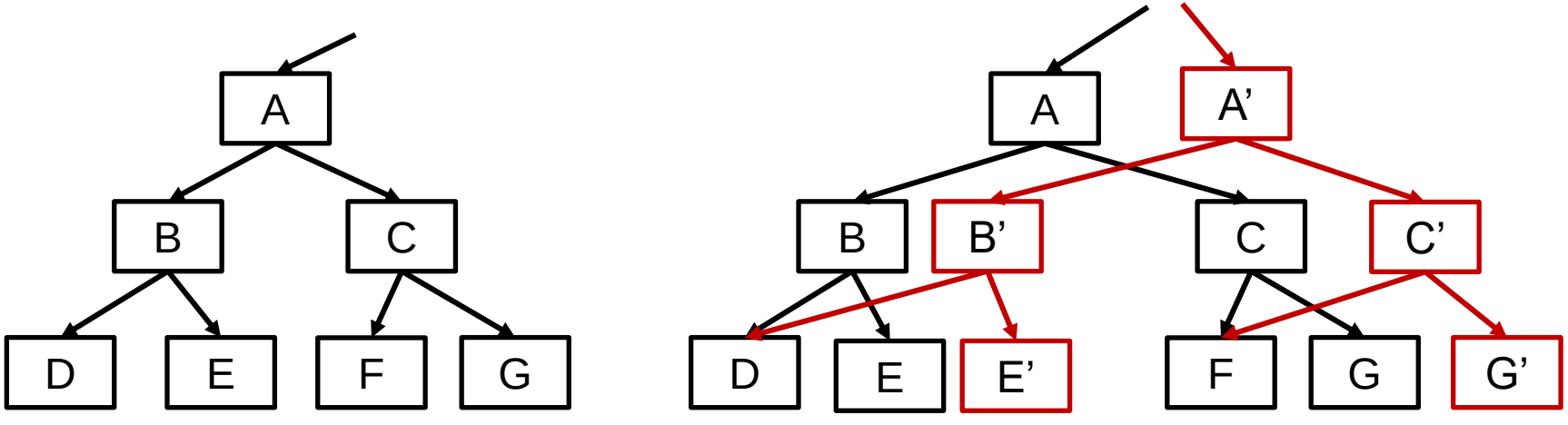
- **权衡一致性和性能**
  - 数据的数量大，只需要写入一次
  - 元数据的数量少，写入两次相对可接受
- **可能出现的问题**
  - 数据只有一份，若出现问题无法回退（all-or-nothing）
  - 部分情况下，一致性还是可以保证的（如新增数据时）
  - 部分情况下，数据会丢失，但元数据依然可以保证一致性

原子更新技术：Copy-on-Write

写时复制

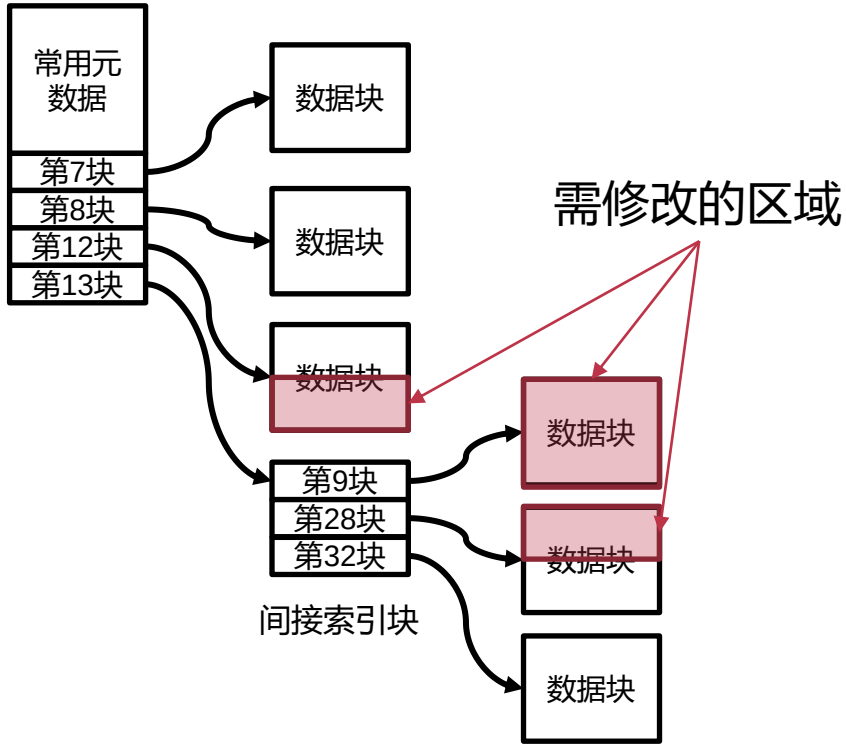
写时复制（Copy-on-Write）

- 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
- 常用于树状结构



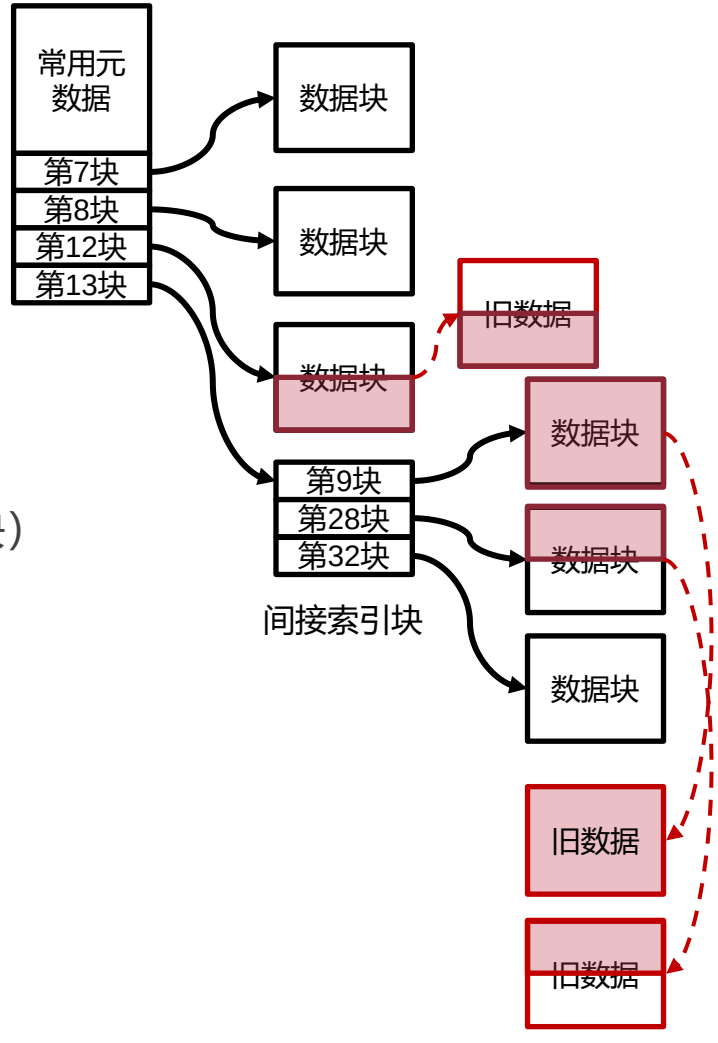
文件中的写时复制

- 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



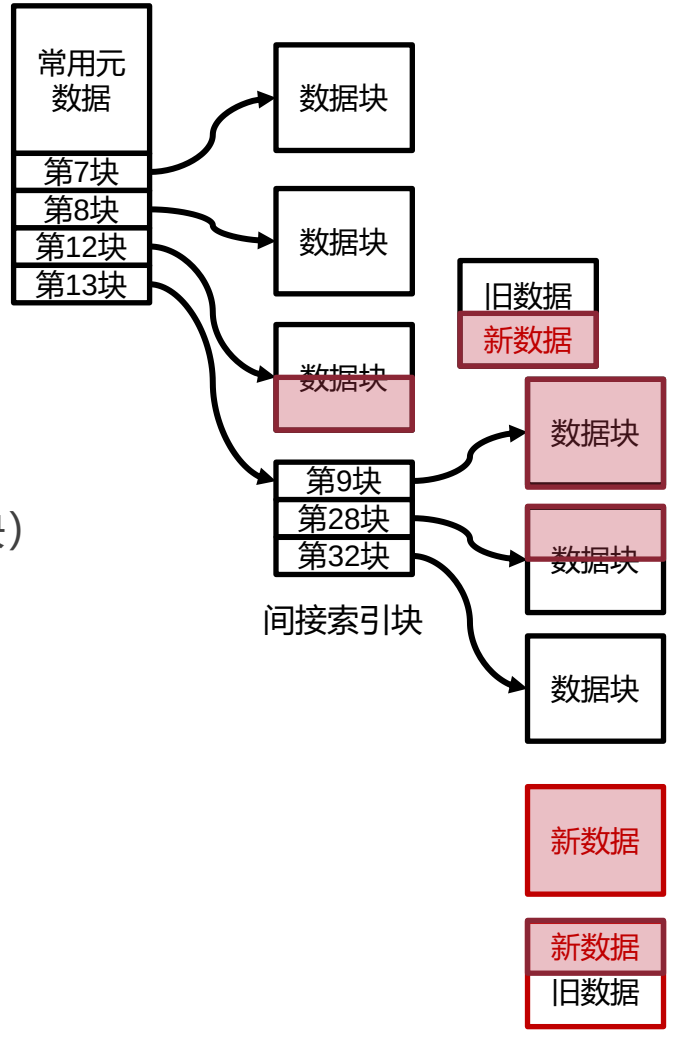
文件中的写时复制

- 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
  - 将要修改的数据块进行复制（分配新的块）



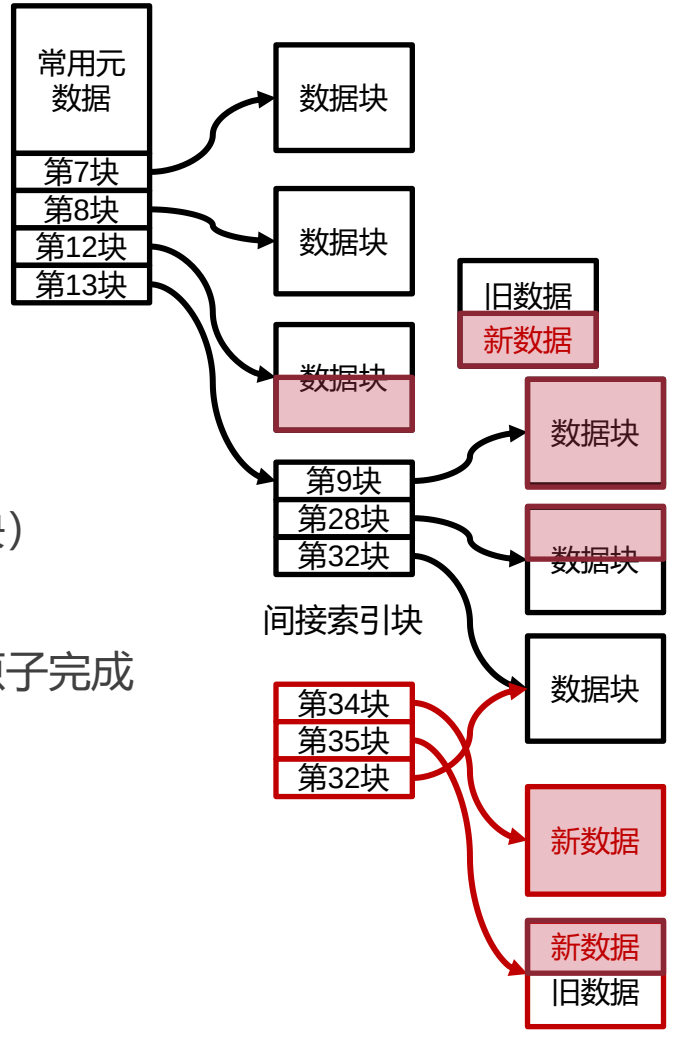
文件中的写时复制

- 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
  - 将要修改的数据块进行复制（分配新的块）
  - 在新的数据块上修改数据



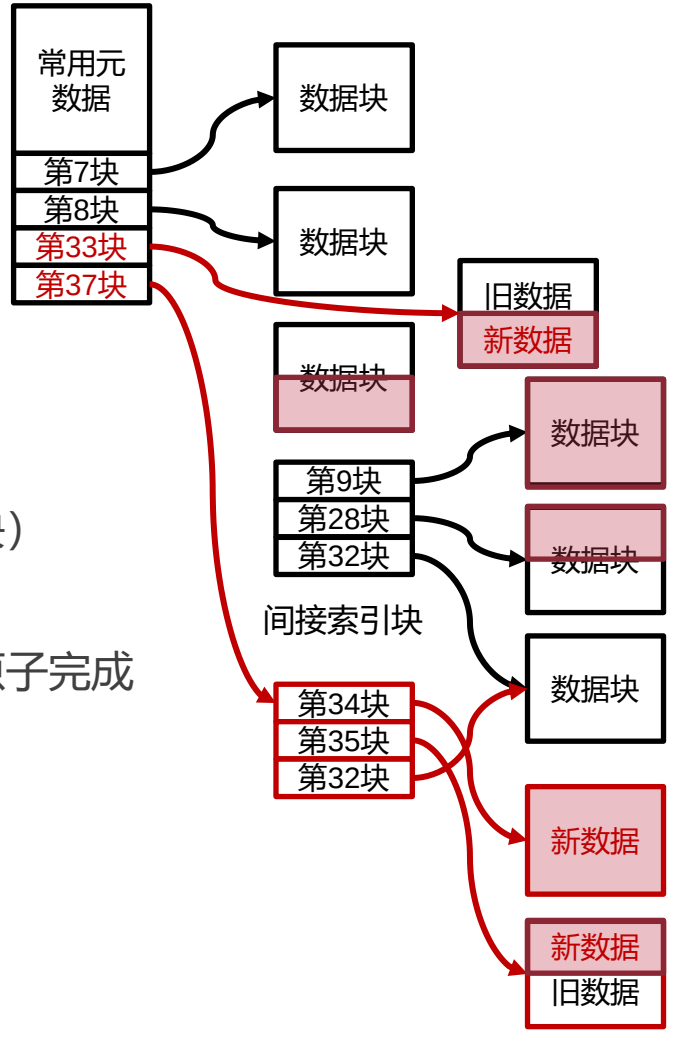
文件中的写时复制

- 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
  - 将要修改的数据块进行复制（分配新的块）
  - 在新的数据块上修改数据
  - 向上递归复制和修改，直到所有修改能原子完成



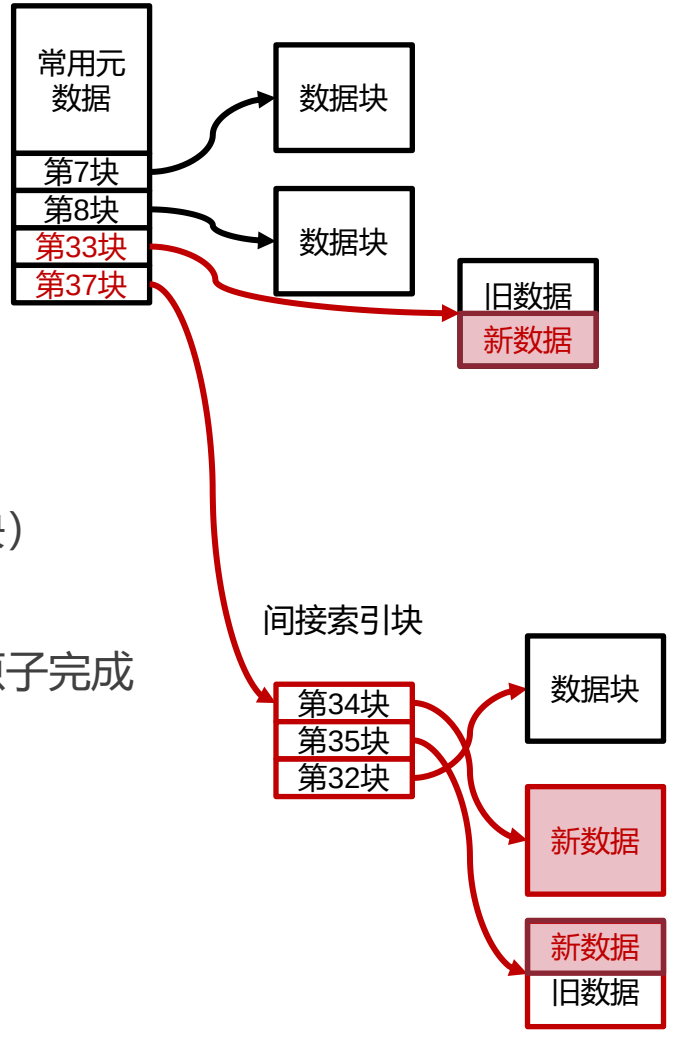
文件中的写时复制

- 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
  - 将要修改的数据块进行复制（分配新的块）
  - 在新的数据块上修改数据
  - 向上递归复制和修改，直到所有修改能原子完成
  - 进行原子修改



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
  - 将要修改的数据块进行复制（分配新的块）
  - 在新的数据块上修改数据
  - 向上递归复制和修改，直到所有修改能原子完成
  - 进行原子修改
  - 回收资源

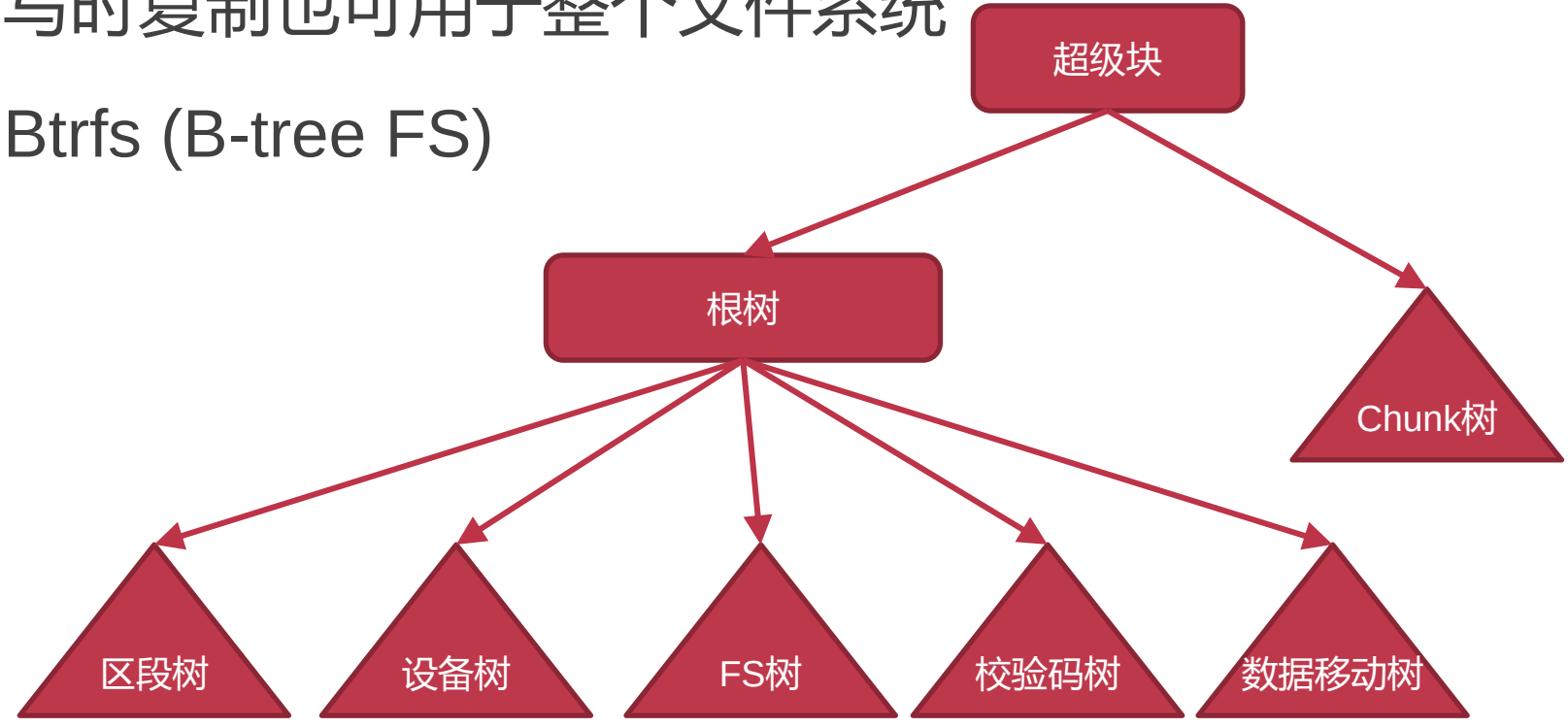


思考时间

- 对于文件的修改，写时复制一定比日志更高效吗?
- 写时复制和日志各自的优缺点有哪些?
- 能否只用写时复制来实现一个文件系统?

"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)



文件复制

MAC的APFS磁盘秒复制超大文件是什么原理 （11）

[收藏]

乐海浮沉
581

... 18-5-20 14:53 #1

MAC的APFS磁盘秒复制超大文件是什么原理

试了下，复制一个10多G的虚拟机文件，不要一秒就成功了，再试两个同时复制，也是一样，再翻倍，四个，八个同时复制，还是那么快...
...我想是不是因为我的是固态硬盘所以快，但是放到另外一个格式化为APFS格式的机械硬盘里边测试，还是一样，就跟发送快捷方式那么快，不过检查了确定是实体文件不是快捷方式，觉得好神奇啊，，，同样的操作，在mac os扩展日志式格式下就没那么快，

http://www.audiobar.cn/forum.php?mod=viewthread&tid=523830

文件复制

MAC的APFS磁盘秒复制超大文件是什么原理 （11）

[收藏]

乐海浮沉
581

... 18-5-20 14:53 #1

MAC的APFS磁盘秒复制超大文件是什么原理

试了下，复制一个10多G的虚拟机文件，不要一秒就成功了，再试两个同时复制，也是一样，再翻倍，四个，八个同时复制，还是那么快...
...我想是不是因为我的是固态硬盘所以快，但是放到另外一个格式化为APFS格式的机械硬盘里边测试，还是一样，就跟发送快捷方式那么快，不过检查了确定是实体文件不是快捷方式，觉得好神奇啊，，，同样的操作，在mac os扩展日志式格式下就没那么快，

read/write

```
$ cp A B

1. 打开文件A
2. 创建并打开文件B
3. 从A中读出数据到buffer
4. 将buffer中的数据写入B
5. 重复3、4直到文件A被读完
```

mmap

```
$ cp A B

1. 打开文件A
2. 获取A的大小为X
3. 创建并打开文件B
4. 改变B的大小为X(fallocate/ftruncate)
5. 将A和B分别mmap到内存空间
6. memcpy
```

http://www.audiobar.cn/forum.php?mod=viewthread&tid=523830

文件复制

MAC的APFS磁盘秒复制超大文件是什么原理 （11）

[收藏]

乐海浮沉
581

... 18-5-20 14:53 #1

MAC的APFS磁盘秒复制超大文件是什么原理

试了下，复制一个10多G的虚拟机文件，不要一秒就成功了，再试两个同时复制，也是一样，再翻倍，四个，八个同时复制，还是那么快...
...我想是不是因为我的是固态硬盘所以快，但是放到另外一个格式化为APFS格式的机械硬盘里边测试，还是一样，就跟发送快捷方式那么快，不过检查了确定是实体文件不是快捷方式，觉得好神奇啊，，，同样的操作，在mac os扩展日志式格式下就没那么快，

read/write

```
$ cp A B

1. 打开文件A
2. 创建并打开文件B
3. 从A中读出数据到buffer
4. 将buffer中的数据写入B
5. 重复3、4直到文件A被读完
```

mmap

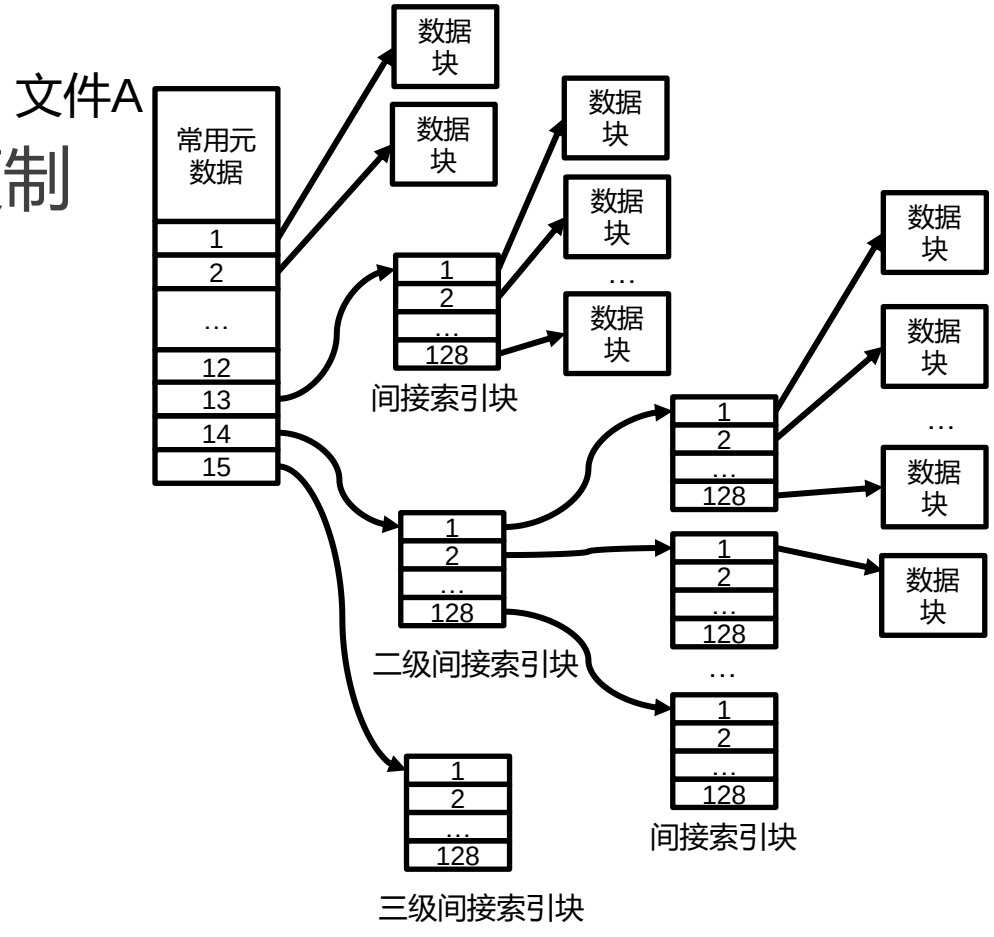
```
$ cp A B

1. 打开文件A
2. 获取A的大小为X
3. 创建并打开文件B
4. 改变B的大小为X(fallocate/ftruncate)
5. 将A和B分别mmap到内存空间
6. memcpy
```

文件越慢，耗时越长！如何做到秒复制?

http://www.audiobar.cn/forum.php?mod=viewthread&tid=523830

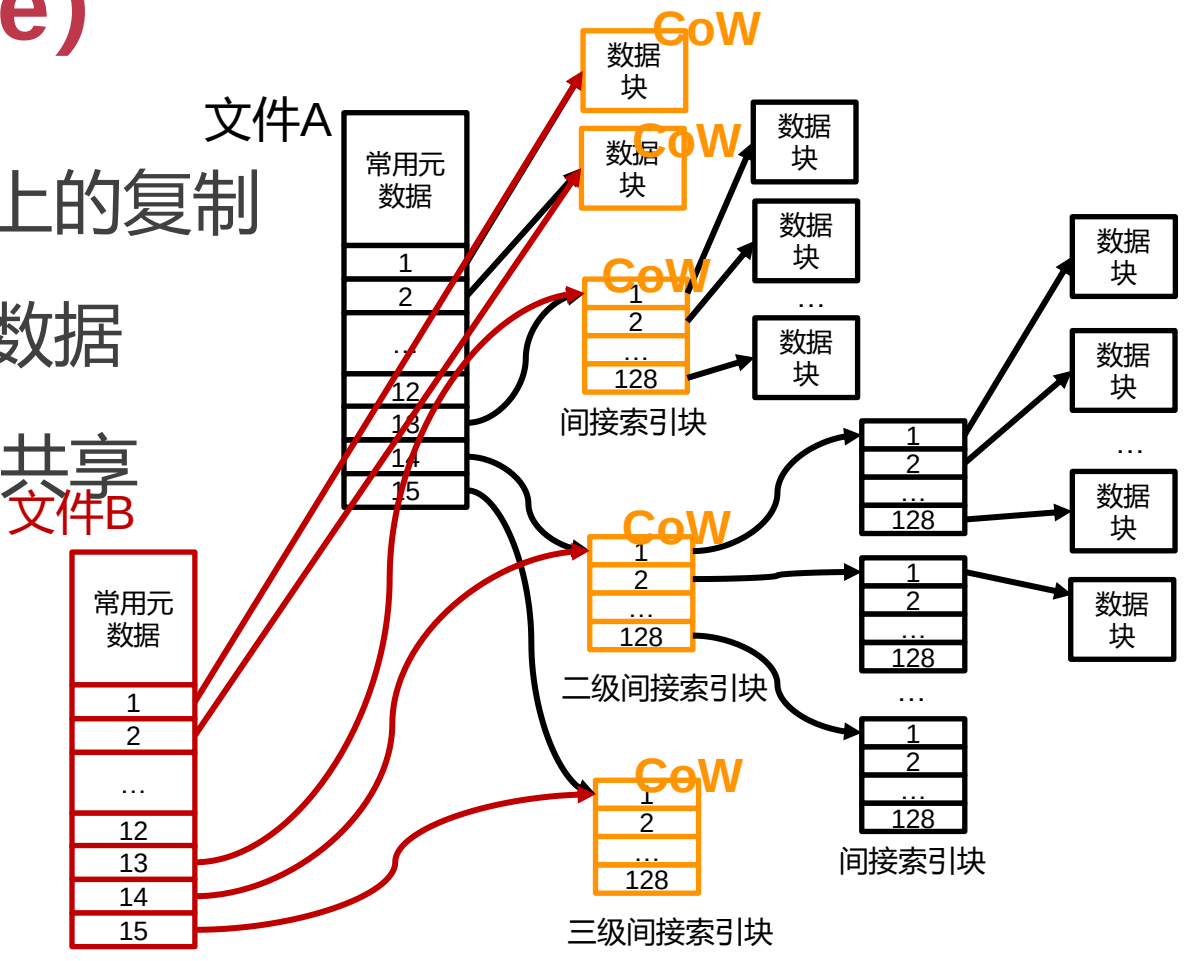
克隆

- 文件系统层面上的复制
- 只复制关键元数据
- 其他部分CoW共享



克隆（Clone）

- 文件系统层面上的复制
- 只复制关键元数据
- 其他部分CoW共享



快照（Snapshot）

- 同样使用CoW
- 对于基于inode表的文件系统
  - 将inode表拷贝一份作为快照保存
  - 标记已用数据区为CoW
- 对于树状结构的文件系统
  - 将树根拷贝一份作为快照保存
  - 树根以下的节点标记为CoW

SOFT UPDATES

Soft Updates

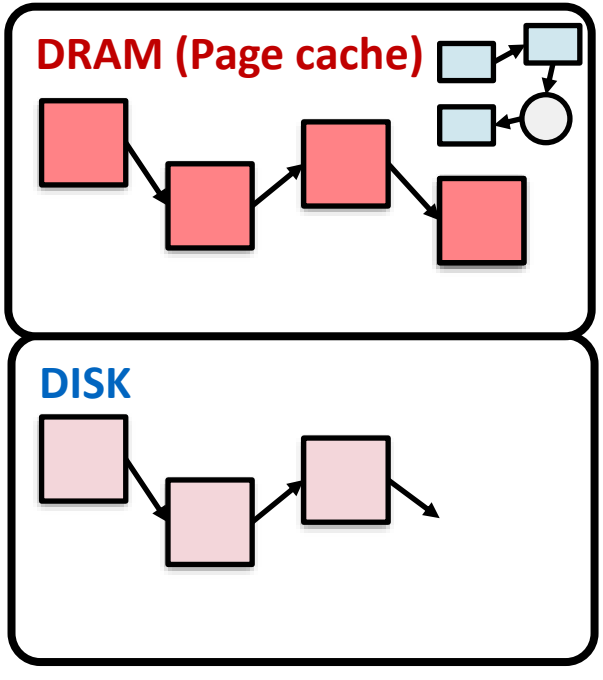
- 一些不一致情况是良性的
  - 某inode被标记为占用，却从文件系统中无法遍历到该inode
    - 如创建文件：
      1. 标记inode为占用
      2. 初始化inode
      3. 将目录项写入目录中
  - 合理安排修改写入磁盘的次序（order），可避免恶性不一致情况的发生
- 相对其它方法的优势
  - 无需恢复便可挂载使用
  - 无需在磁盘上记录额外信息

回顾：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

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



传统的 Soft Updates

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指向了一个数据块时，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

Soft Updates

- **对于每个文件系统请求，将其拆解成对多个结构的操作**
  - 记录对每个结构的修改内容（旧值、新值）
  - 记录这个修改依赖于那些修改（应在哪些修改之后持久化）
  - 如创建文件：
    1. 标记inode为占用（对bitmap的修改）
    2. 初始化inode（对inode的修改，依赖于1）
    3. 将目录项写入目录中（对目录文件的内容修改，依赖于1和2）

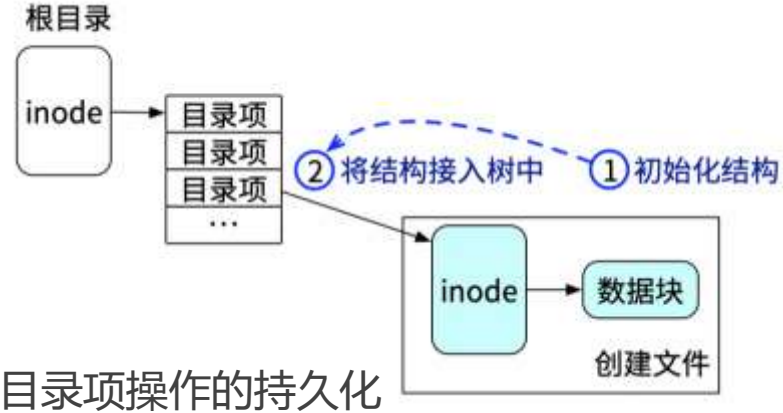
创建文件/目录

包括3个主要步骤：

分配 inode

初始化 inode

增加目录项



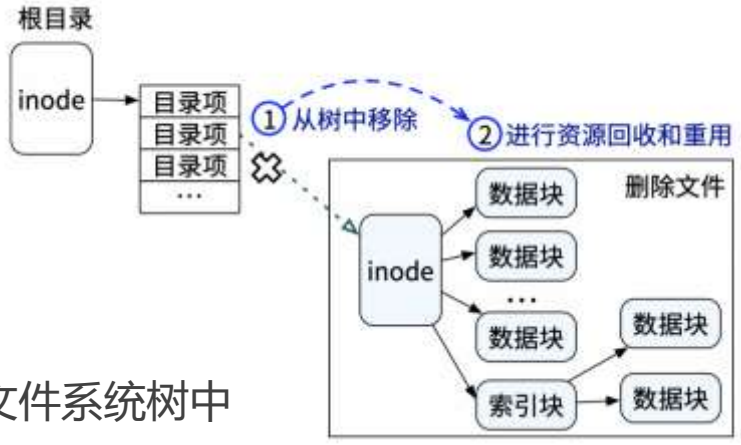
- **操作步骤**
  - 根据规则1，inode 初始化的持久化应早于增加目录项操作的持久化
  - 根据规则2，inode 分配同样应该在增加目录项之前持久化
  - 需同时创建“.”和“..”两个目录项，需要分配并初始化一个新的数据块
  - 根据规则1，数据块的初始化操作应该先 于 inode 中索引的修改持久化
  - 根据规则2，数据块分配信息（如 bitmap）的持久化同样应该在 inode初始化之前
- **异常情况**
  - 依然可能会发生空间泄漏，即 inode 分配信息被持久化但却未被文件系统使用
  - 对文件系统结构没有影响；可通过定期扫描找到未使用的 inode 节点并修复

删除文件

包括2个主要步骤：

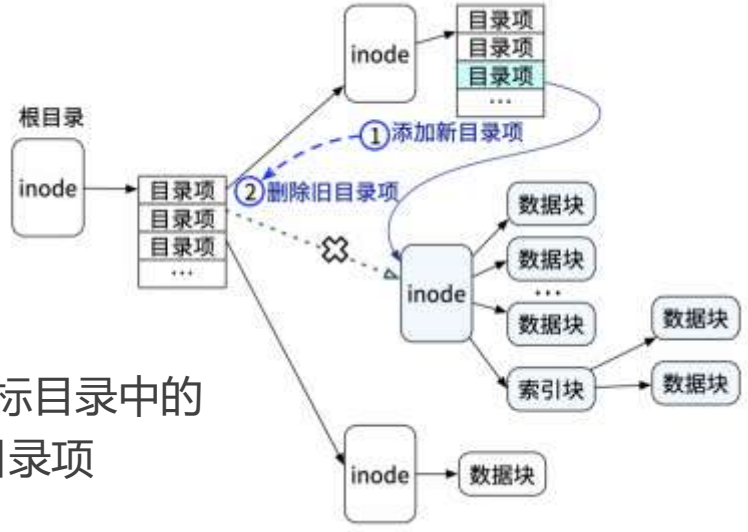
从文件树移除

资源回收和重用



- **操作步骤**
  - 根据规则2，文件系统需要将目标文件先从整个文件系统树中去掉，再进行资源回收等操作
- **异常情况**
  - 删除目录项为删除操作的原子更新点，系统崩溃只可能发生在此操作之前或之后
  - 若崩溃发生在此之前，则没有删除操作被执行，因此不会产生不一致的情况
  - 若崩溃发生在此之后，并不会造成文件系统中其他文件和数据的不一致性
    - 此过程中造成的空间泄漏，也 可以通过定期检查的方法进行修复

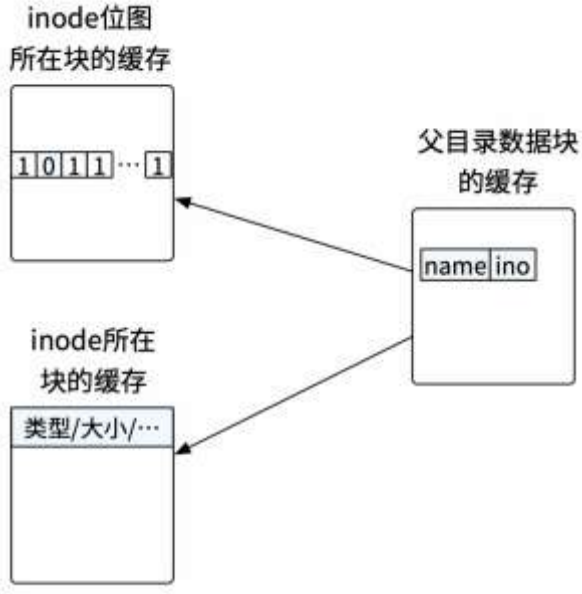
文件重命名



- **操作步骤**
  - 根据规则3，在进行文件移动时，需要先保证目标目录中的目录项被写入完毕，之后才能删除源目录中的目录项
- **异常情况**
  - 目标目录中的目录项写入完毕后发生崩溃
  - 重启后会发现两个目录中的目录项均指向该 inode 结构
  - 并未造成被移动文件的数据丢失， 也不影响文件系统中其他文件的一致性

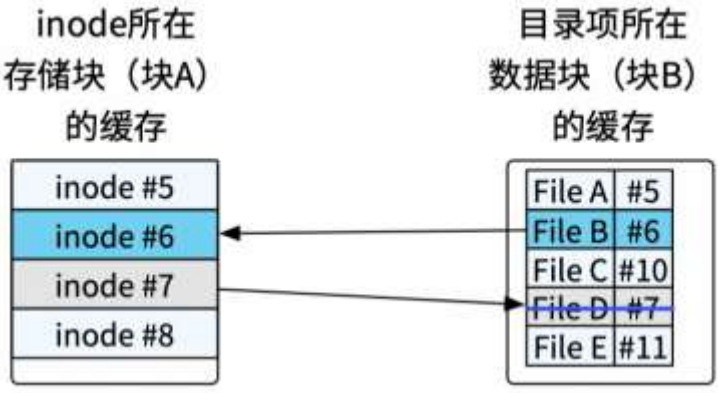
依赖追踪

- **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持久化到 存储设备中
  - 问题：如何保证这些修改的持久化顺序呢?
- **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



基于块的 DAG 依赖关系图

依赖追踪的两个问题



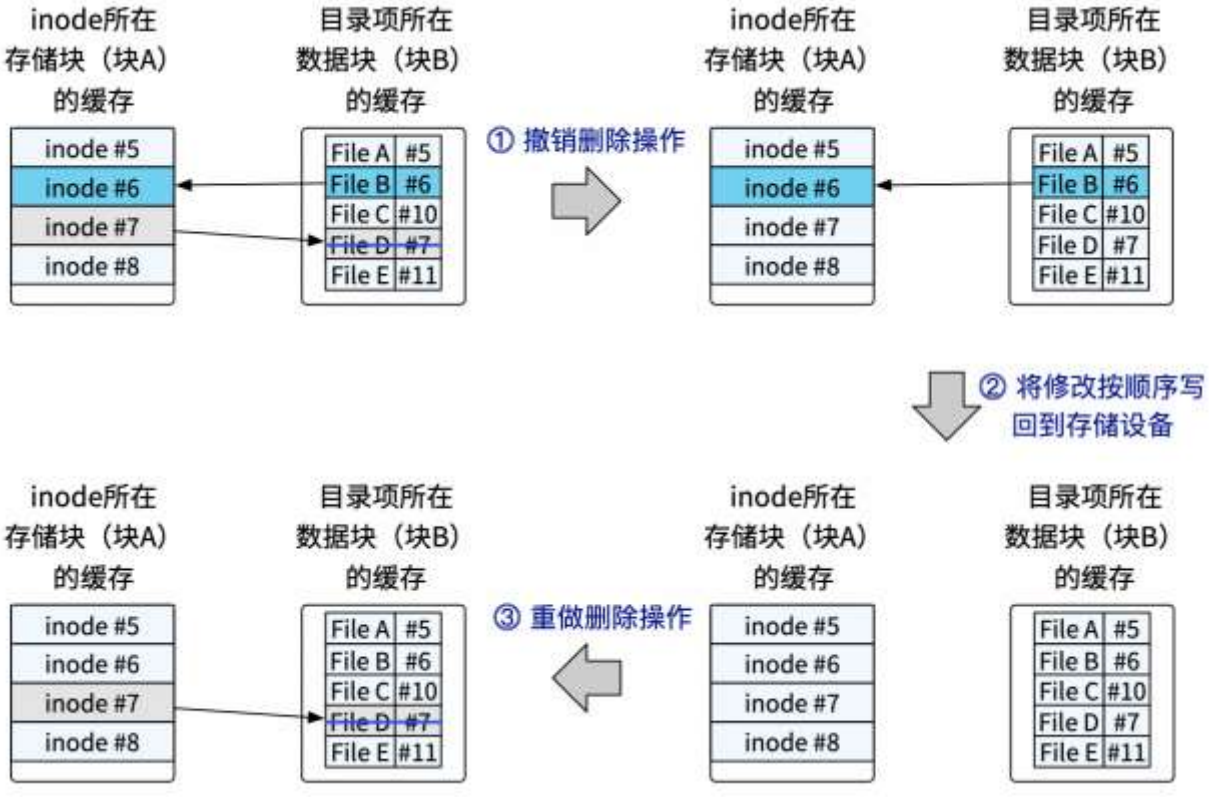
- **问题1：环形依赖**
  - 一个块通常包含多个文件系统结构
  - 环形依赖：块 A 需要在块 B 前写回，同时块 B 需要在块 A 前写回

- **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**
  - 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
  - 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
  - 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
  - 持久化完毕之后，将此前被撤销的操作恢复，即重做。
  - 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

撤销和重做


inode所在
存储块（块A）
的缓存
目录项所在
数据块（块B）
的缓存
inode #5
inode #6
inode #7
inode #8
File A #5
File B #6
File C #10
File D #7
File E #11
① 撤销删除操作
② 将修改按顺序写
回到存储设备
③ 重做删除操作

撤销和重做

- **不仅能够打破环形依赖，还能解决写回迟滞的问题**
  - 若某个结构被频繁修改，导致不断有新的依赖产生时，可将部分新的修改撤销，在快速完成持久化后将修改重做
  - 避免新依赖不断推迟该结构上修改的持久化

参考资料

- 上海交通大学并行与分布式系统研究所操作系统课程
- 操作系统导论，[美] Remzi H. Arpaci-Dusseau / [美] Andrea C. Arpaci-Dusseau，人民邮电出版社，2019
- 2020 南京大学 “操作系统：设计与实现” ，蒋炎岩
- 文件系统技术内幕，大数据时代的海量数据存储之道，张书宁，电子工业出版社，2021
- 网络文章与图片