中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

文件系统崩溃一致性

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

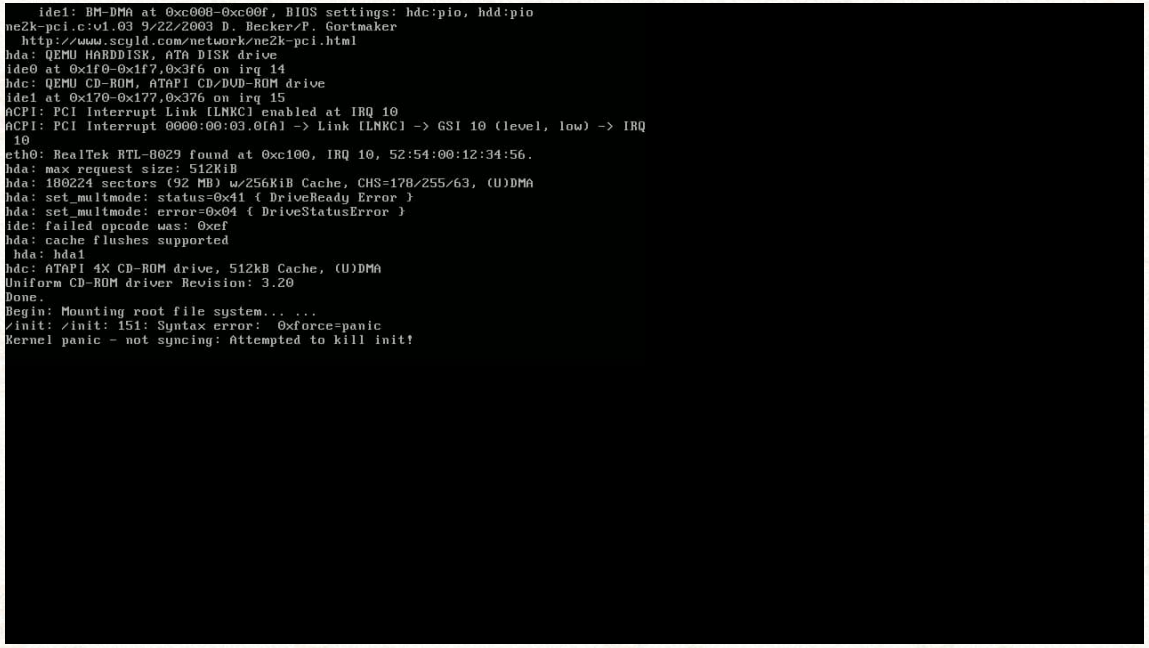
助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

电脑偶尔会崩溃

- 台式机突然断电
- U盘突然拔出
- 数据线接触不良
- 设备老化自己坏了
- 还有很多，靠猜。。。



很多崩溃是由硬盘/文件系统损坏导致的!

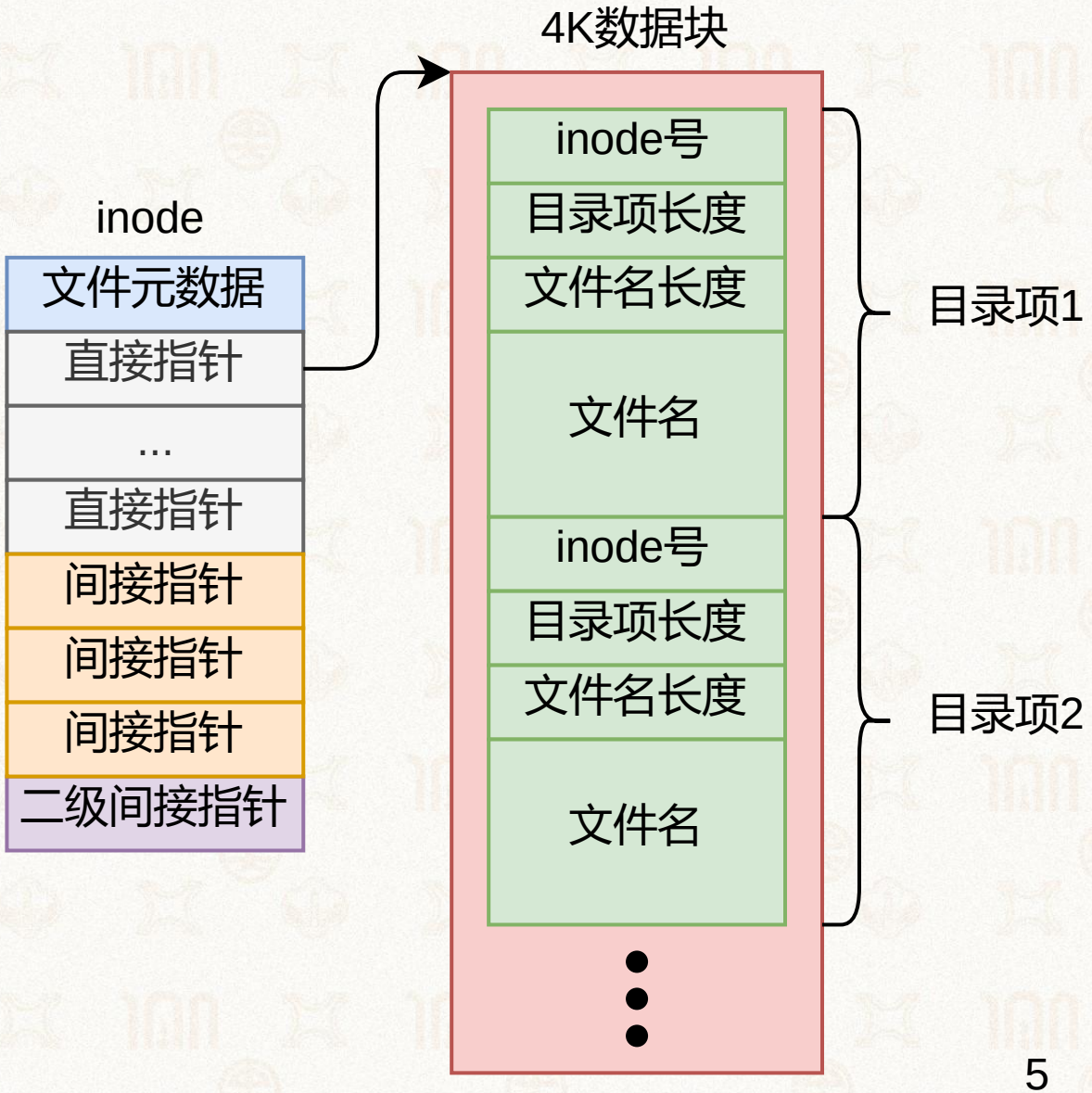
大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
    - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

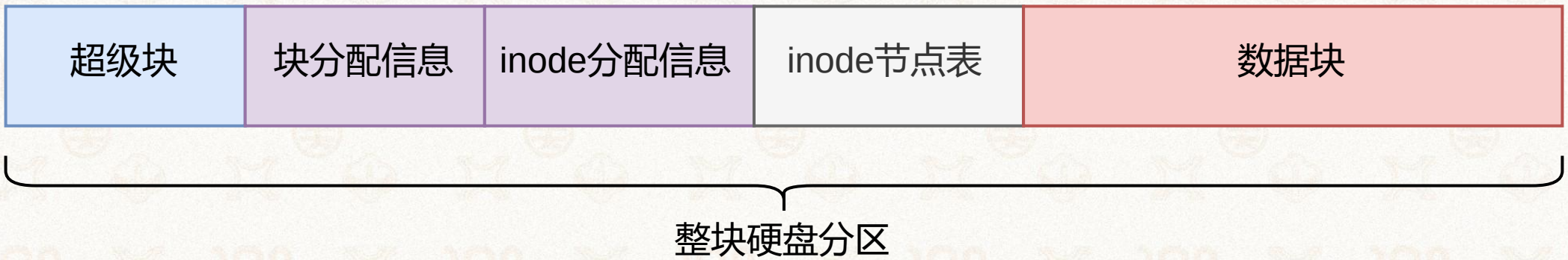
文件的创建

- 创建"/chb"
  - 先找到要创建文件所在目录的inode



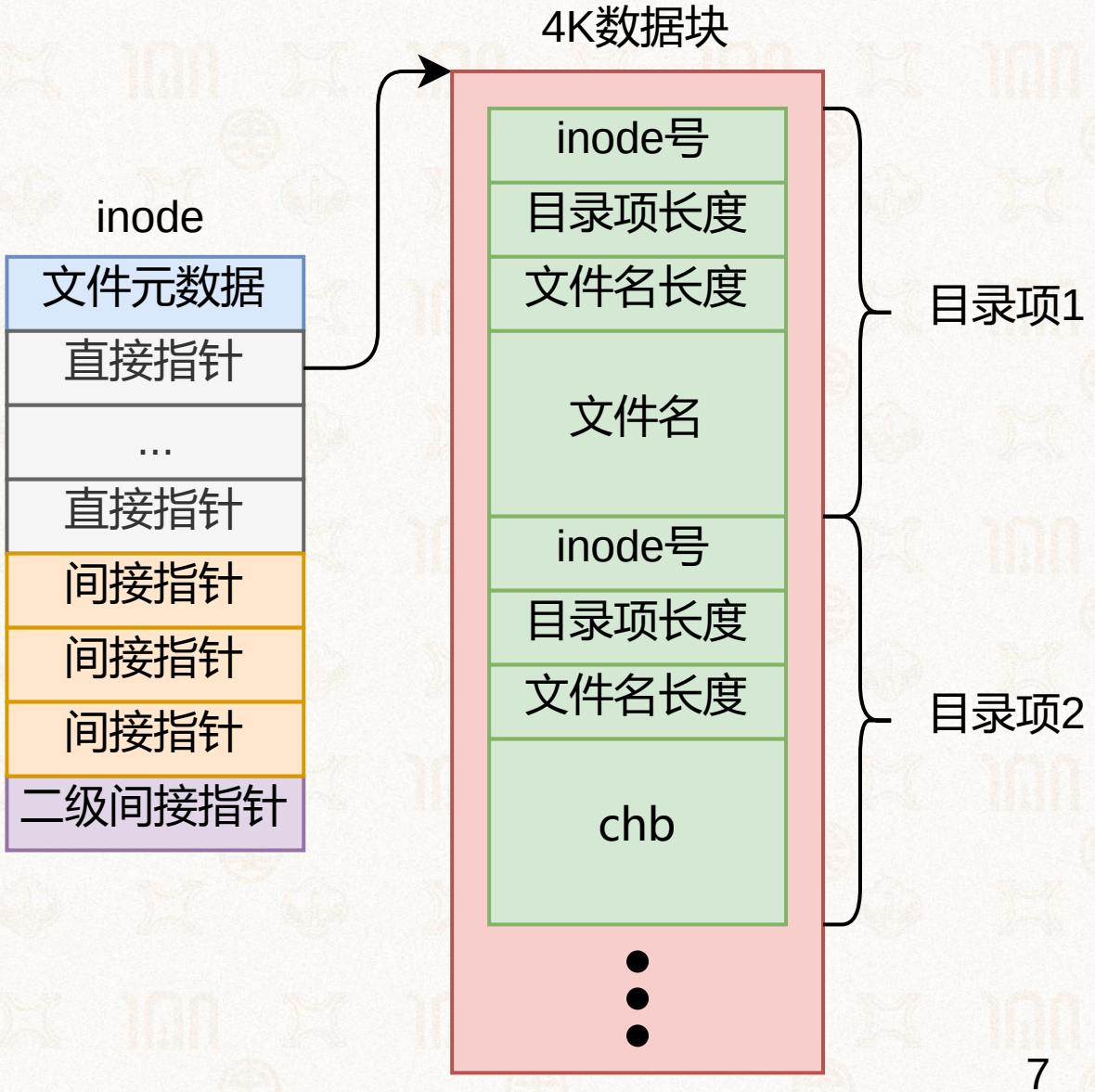
文件的创建

- 创建"/chb"
  - 先找到要创建文件所在目录的inode
  - 从inode分配表中找出一个空闲inode，标记为占用；并初始化该inode



文件的创建

- 创建"/chb"
  - 先找到要创建文件所在目录的inode
  - 从inode分配表中找出一个空闲inode，标记为占用；并初始化该inode
  - 将”chb"和新分配的inode号作为新的目录项写入目录中。



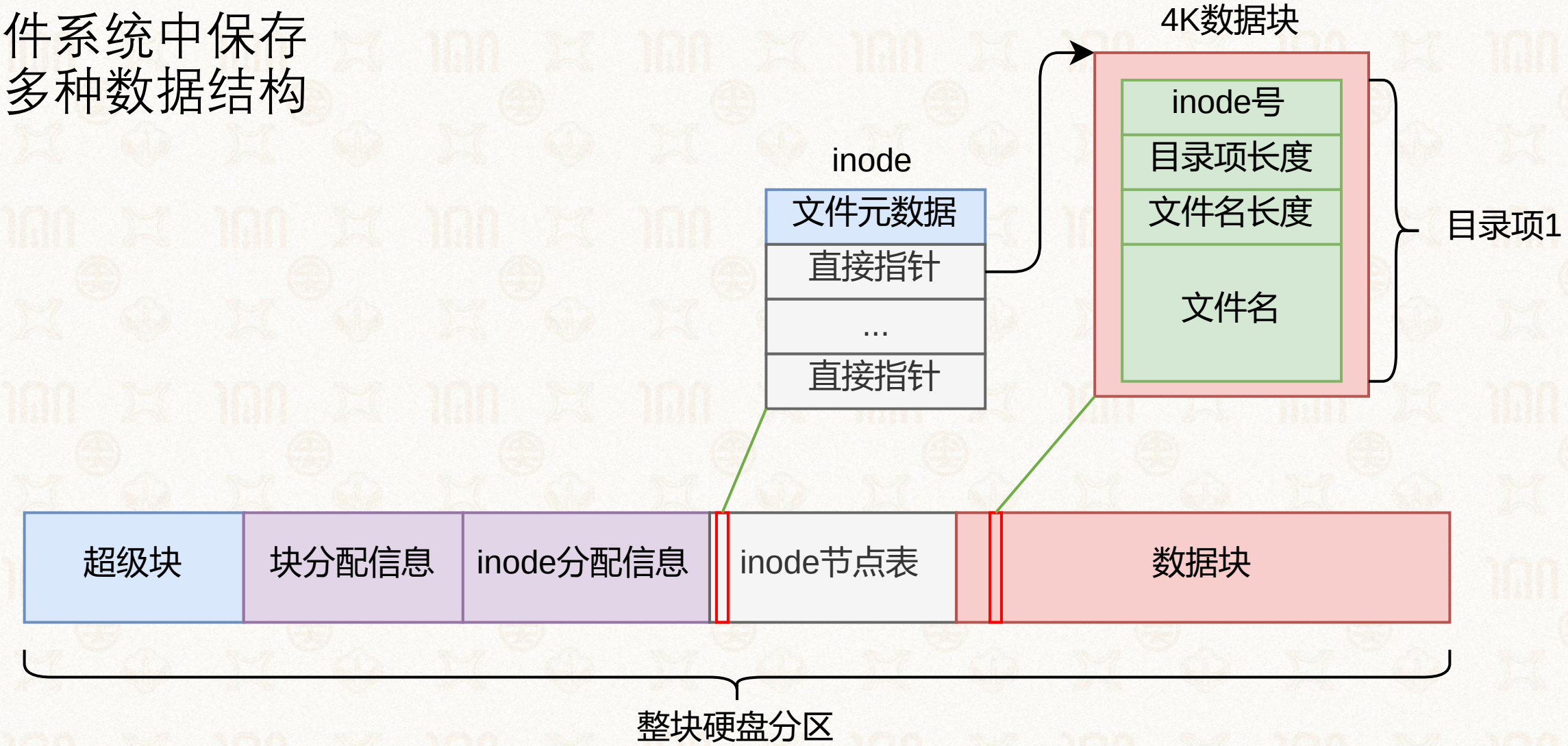
大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
    - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

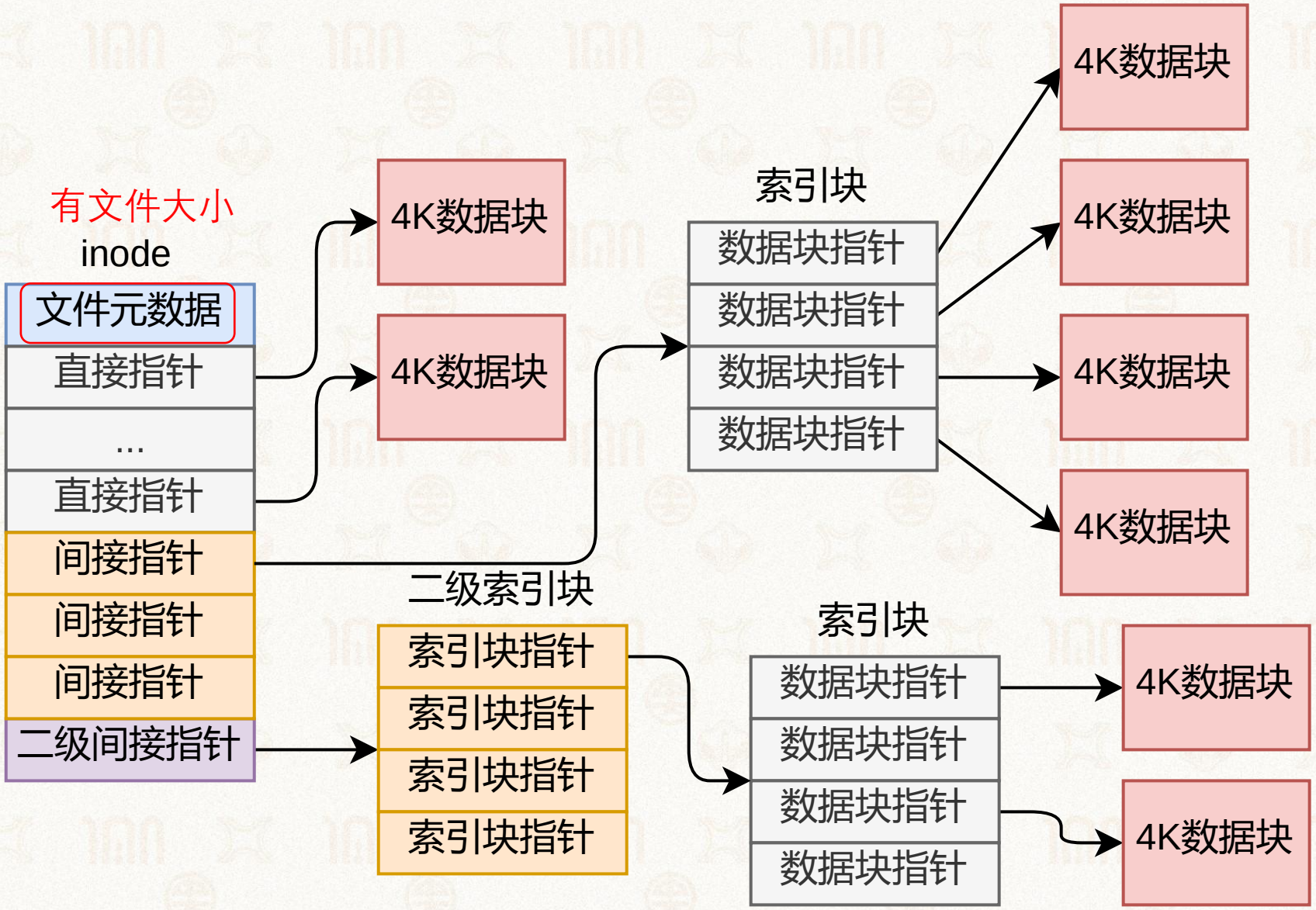
文件系统的一致性约束

- 文件系统中保存了多种数据结构



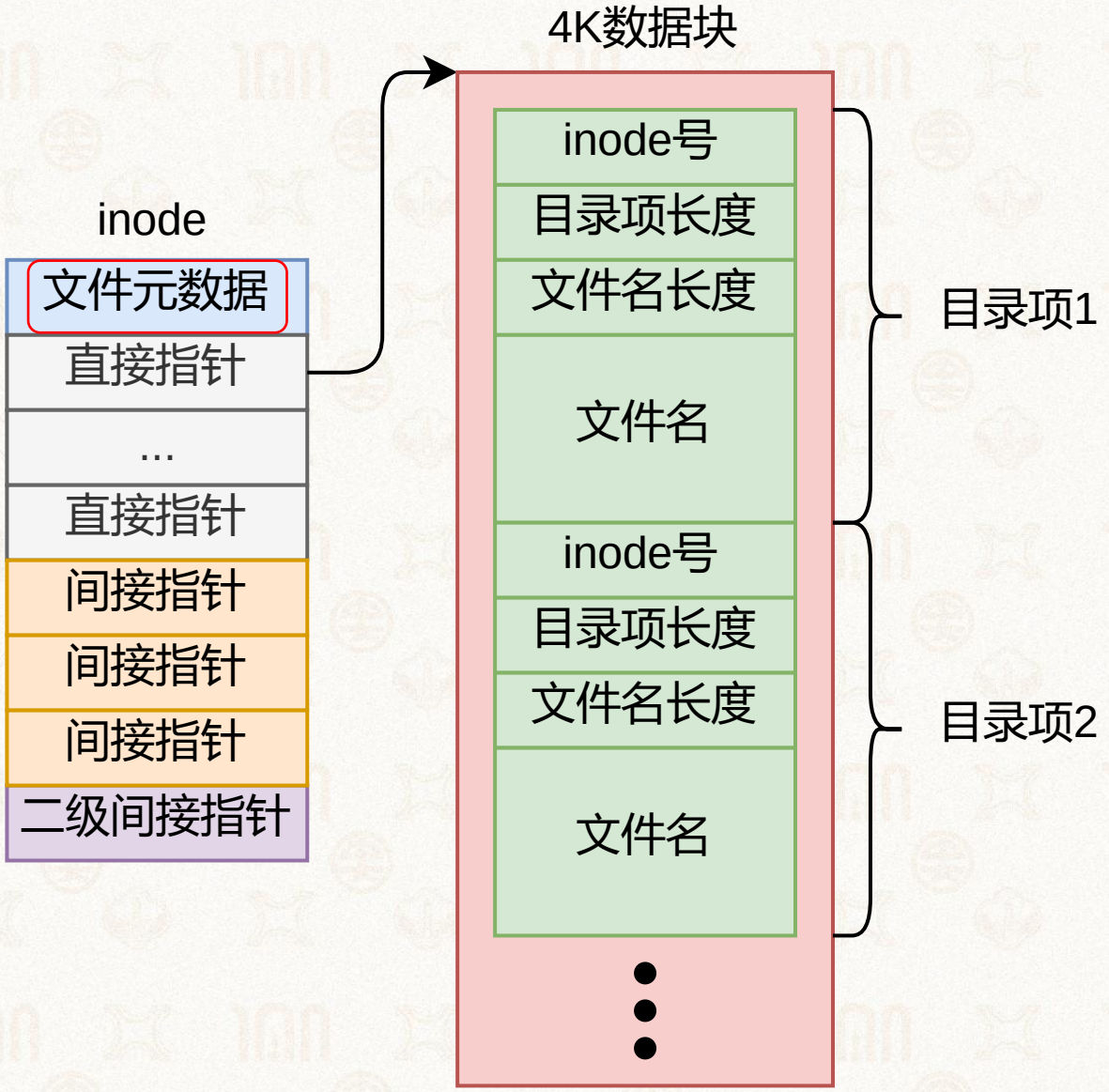
文件系统的一致性约束

- 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



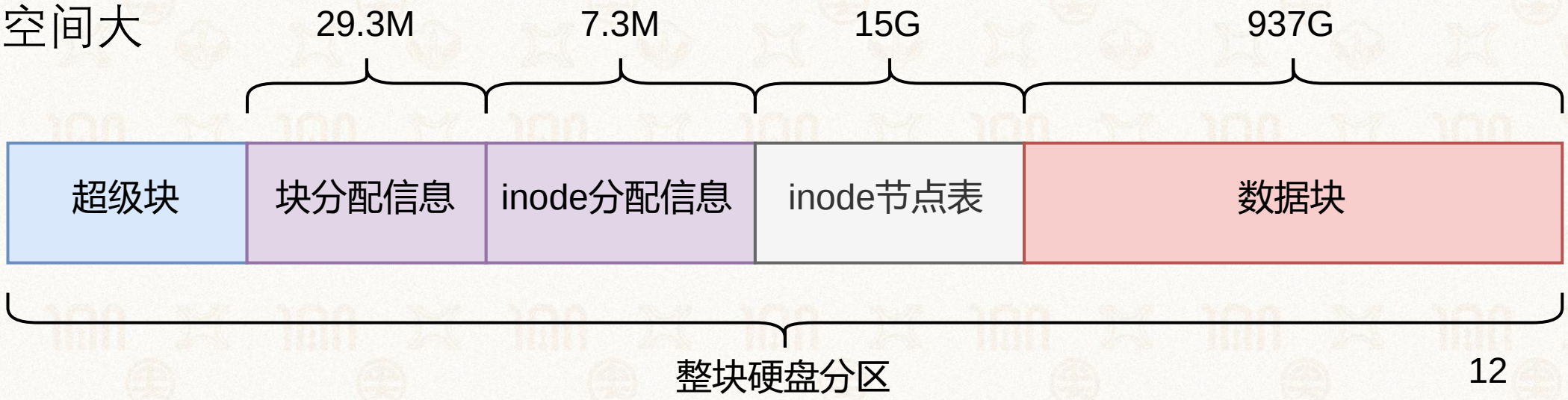
文件系统的一致性约束

- 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中保存的链接数，应与指向其的目录项个数相同



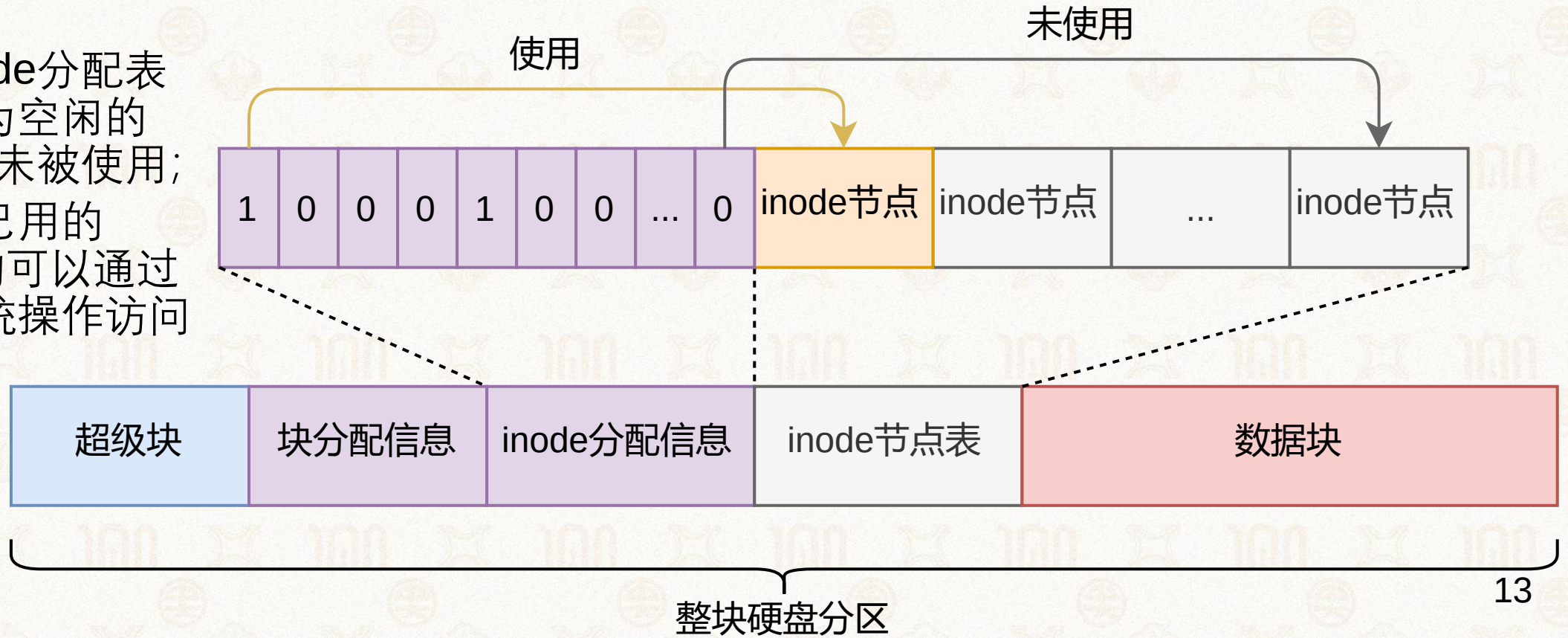
文件系统的一致性约束

- 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
  - 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



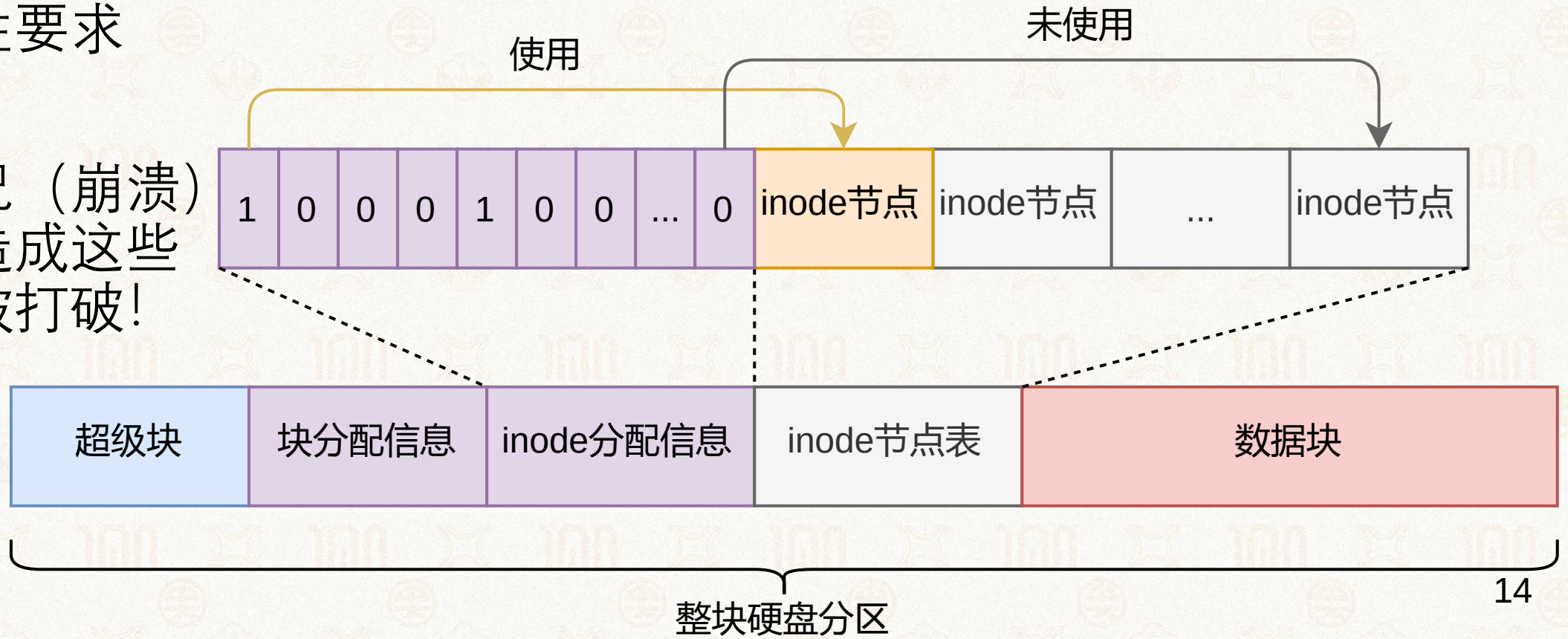
文件系统的一致性约束

- 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
  - 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



文件系统的一致性约束

- 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
- 突发状况（崩溃）可能会造成这些一致性被打破!



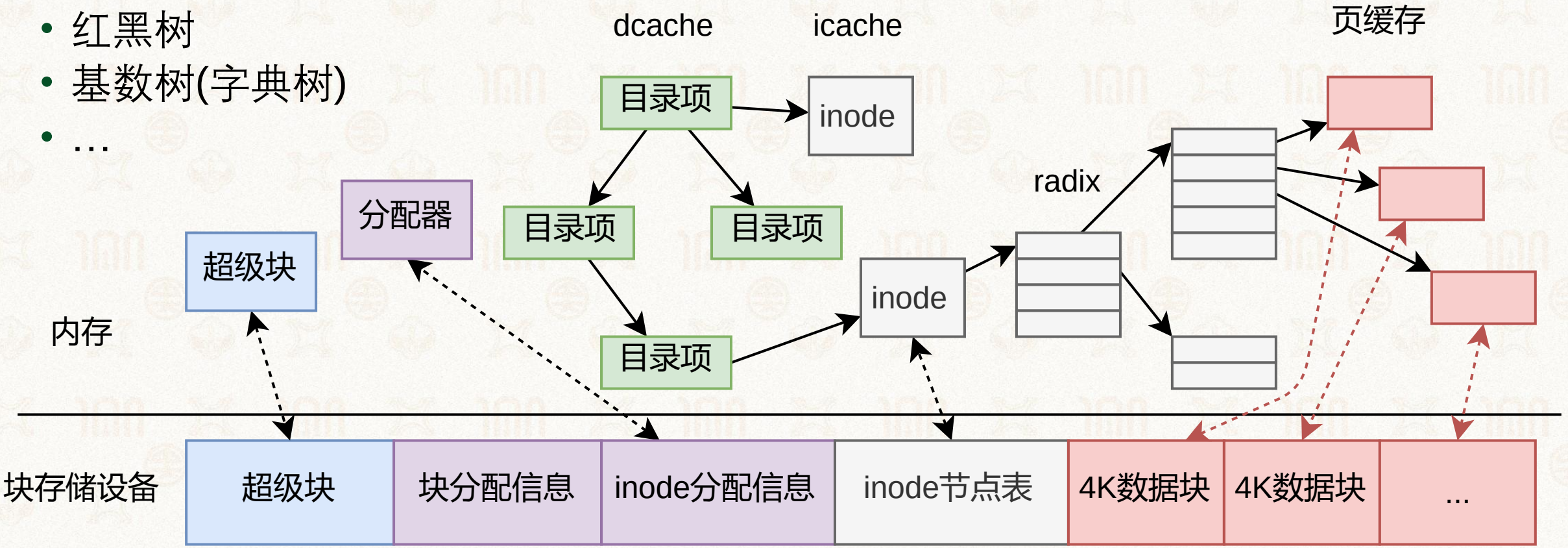
文件的创建

- 创建“/chb”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！

文件系统在内存中有缓存

- 文件系统中多种结构在内存中均有映射
- 利用更多数据结构做优化
  - 红黑树
  - 基数树(字典树)
  - …



大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
  - 不详细讲，太复杂，有兴趣同学自己看

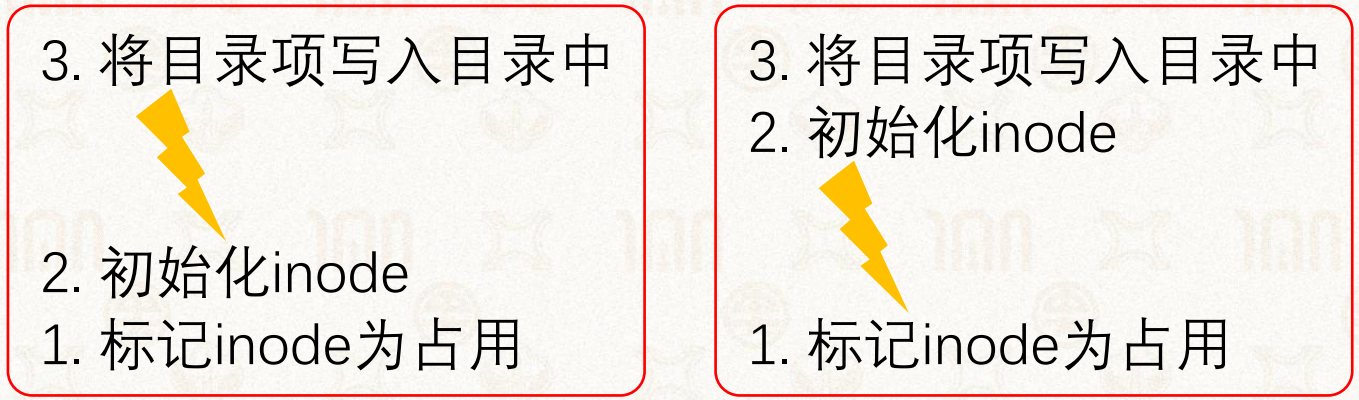
保障文件系统一致性的两大类技术

考虑内存缓存下的崩溃情况

- 创建“/chb”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

考虑存在缓存，共存在多少种的崩溃情况？

两种常见情况：



目录项指向了未分配/未初始化的inode

考虑内存缓存下的崩溃情况

- 创建“/chb”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

考虑存在缓存，共存在多少种的崩溃情况？

共有8种情况

已被持久化的操作(成功保存)	问题
{}	没有操作被持久化
{1}	inode空间泄漏
{2}	后续创建文件时直接覆盖，不产生一致性问题
{3}	访问未初始化的数据，造成错误。错误地被两个不同的文件共享
{1，2}（与 {2，1} 相同）	inode空间泄漏
{1，3}	访问未初始化的数据，造成错误。
{2，3}	错误地指向未分配 inode 结构，产生正确性和安全性问题
{1，2，3}	正常，没有问题

考虑内存缓存下的崩溃情况

- 创建“/chb”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

注意：此处的创建文件还未考虑修改时间戳、写入新目录项需要分配新的数据块、修改超级块中的统计信息等情况。考虑后情况会更复杂!

共有8种情况

已被持久化的操作(成功保存)	问题
{}	没有操作被持久化
{1}	inode空间泄漏
{2}	后续创建文件时直接覆盖，不产生一致性问题
{3}	访问未初始化的数据，造成错误。错误地被两个不同的文件共享
{1，2}（与 {2，1} 相同)	inode空间泄漏
{1，3}	访问未初始化的数据，造成错误。
{2，3}	错误地指向未分配 inode 结构，产生正确性和安全性问题
{1，2，3}	正常，没有问题

单选题 1分

设置

SUN YAT-SEN UNIVERSITY

崩溃情况的讨论是否真实:

手机和笔记本电脑等设备有电池，是否还需要保证文件系统崩溃一致性?

需要

不需要

提交

崩溃一致性：用户期望

- 重启并恢复后...

- 维护文件系统数据结构的内部的不变量
  - 例如，没有磁盘块既处于空闲也在一个文件中

- 仅有最近的一些操作没有被保存到磁盘中
  - 例如：我昨天写的作业文件还存在
  - 用户只需要关心最近的几次修改还在不在

- 没有顺序的异常

一些（简化的）假设

- 磁盘是失效即停(fail-stop)的
- 没失效时，磁盘会忠实执行文件系统下发的命令，不会多做也不会少做
- 如果失效：磁盘可能不会执行最近的几次操作
- 无论是否失效：磁盘不会写飞(wild writes，乱写)

为什么保证崩溃一致性这么困难呢?

- 崩溃可以在任意时刻发生

- 如果系统死机，继续让磁盘完成当前的写操作?
  - CPU能理会你的请求么?

- 如果重启，文件系统可以自动恢复磁盘上的元数据?
  - 只写了一半，如何未卜先知补齐另一半?

方法：在线与离线恢复

- 离线恢复
  - 文件系统检查工具

windows中的chkdsk

```
PS C:\WINDOWS\system32> chkdsk
文件系统的类型是 NTFS。
卷标是 Windows。
阶段 1: 检查基本文件系统结构...
阶段 2: 检查文件名链接...
阶段 3: 检查安全描述符...

Windows 已扫描文件系统并且没有发现问题。
无需采取进一步操作。
每个分配单元中有       4096 字节。
磁盘上共有  249723903 个分配单元。
磁盘上有  163204565 个可用的分配单元。
总持续时间: 1.32 分钟 (79766 毫秒)。
```

Linux中的fsck

```
yxsu@Dell-T6401:~$ sudo fsck -t ext4 /dev/sdb2
fsck，来自 util-linux 2.34
e2fsck 1.45.5 (07-Jan-2020)
/dev/sdb2 已挂载。
e2fsck: 无法继续，已中止。
```

注：fsck常用于系统崩溃时修复，
我的系统好好的，就不演示了

- 在线恢复：运行过程中，检查一些重要的不一致性

大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
    - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

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

大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
    - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

日志(Journaling)

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

- 相当于正式干活之前“打个草稿”

日志(Journaling)

创建"/chb"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

在内存中进行上述操作的同时,
在磁盘上记录日志

日志(Journaling)

创建"/chb"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

日志(Journaling)

创建"/chb"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!

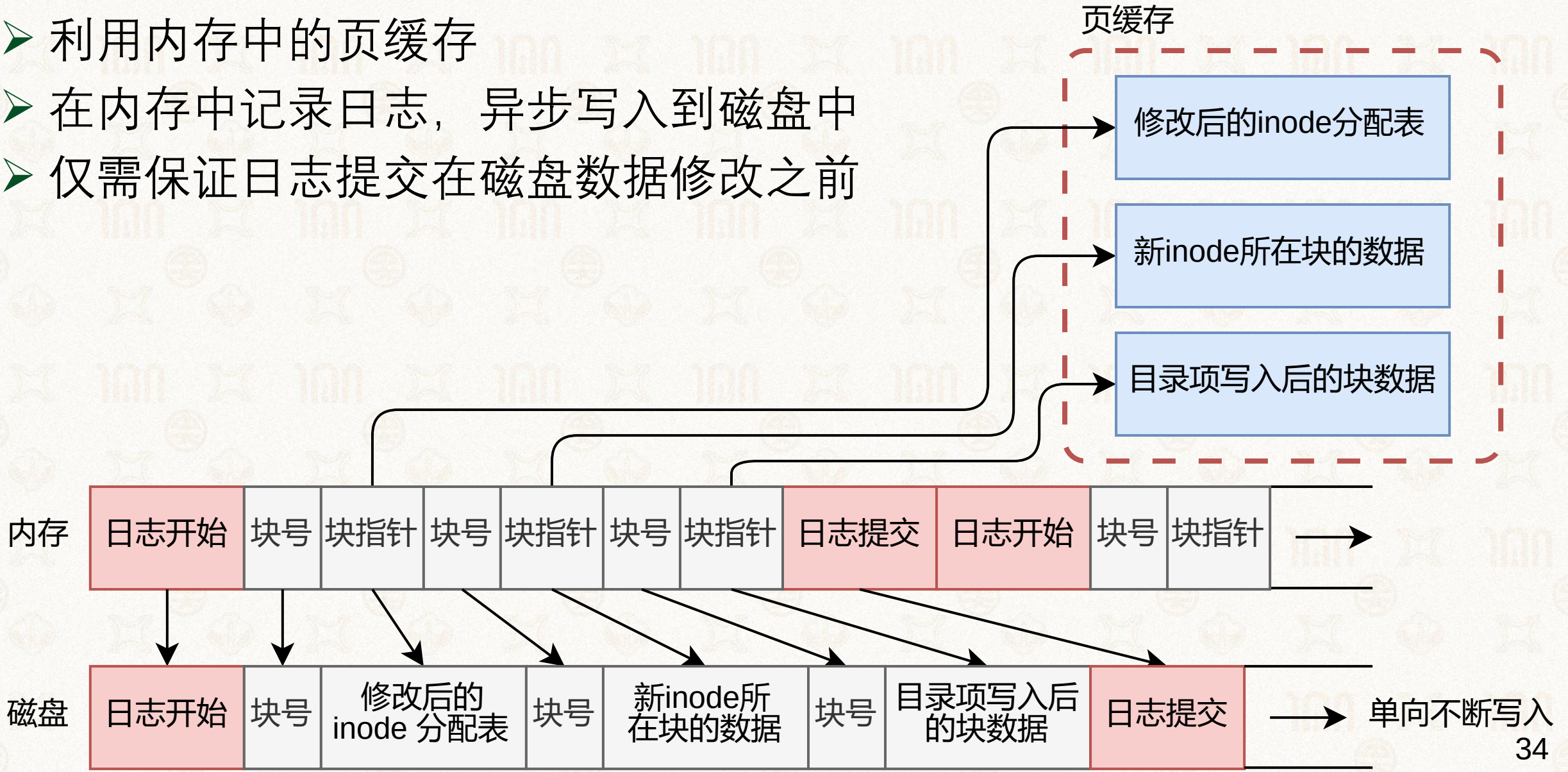
- 在"日志提交"写入存储设备之前崩溃
  - 恢复时发现日志不完整，忽略日志，"/新文件"未被创建
- 在"日志提交"写入存储设备之后崩溃
  - 将日志中的内容，拷贝到对应位置，"/新文件"被创建成功

问题

- 此种方法有什么问题?
- 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
- ……

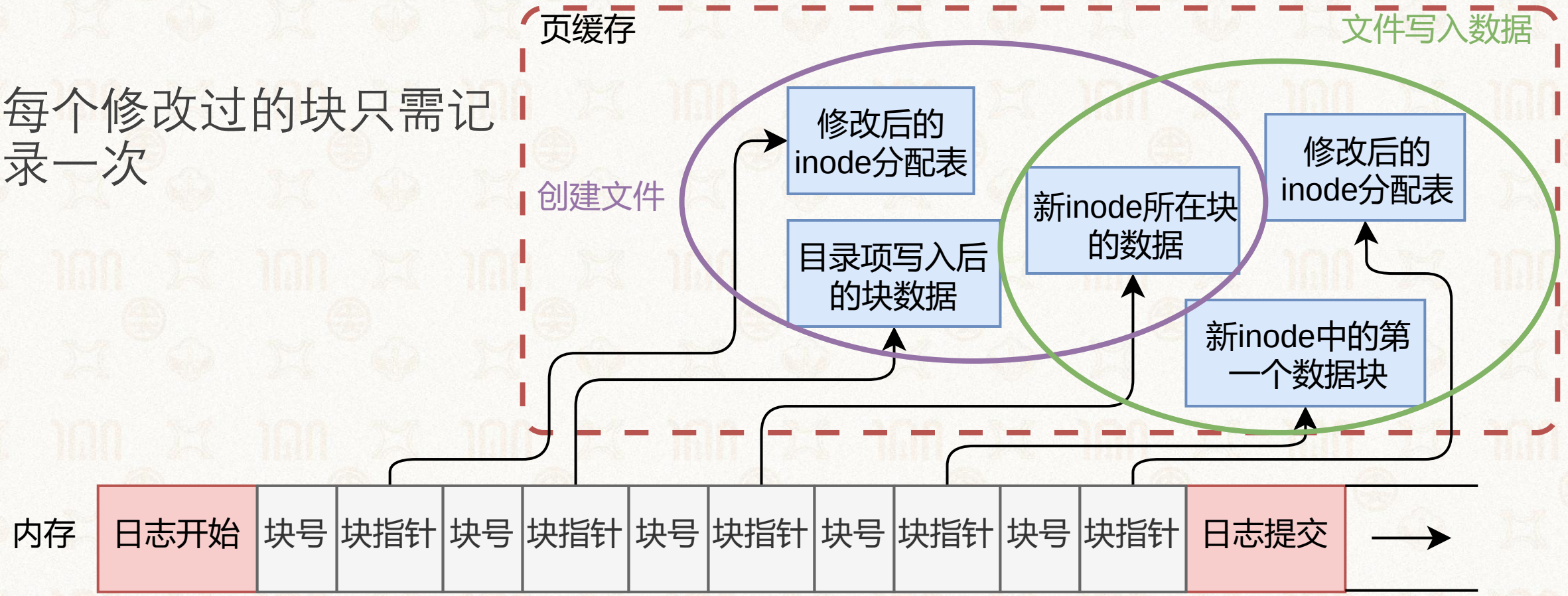
利用内存中的页缓存

- 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



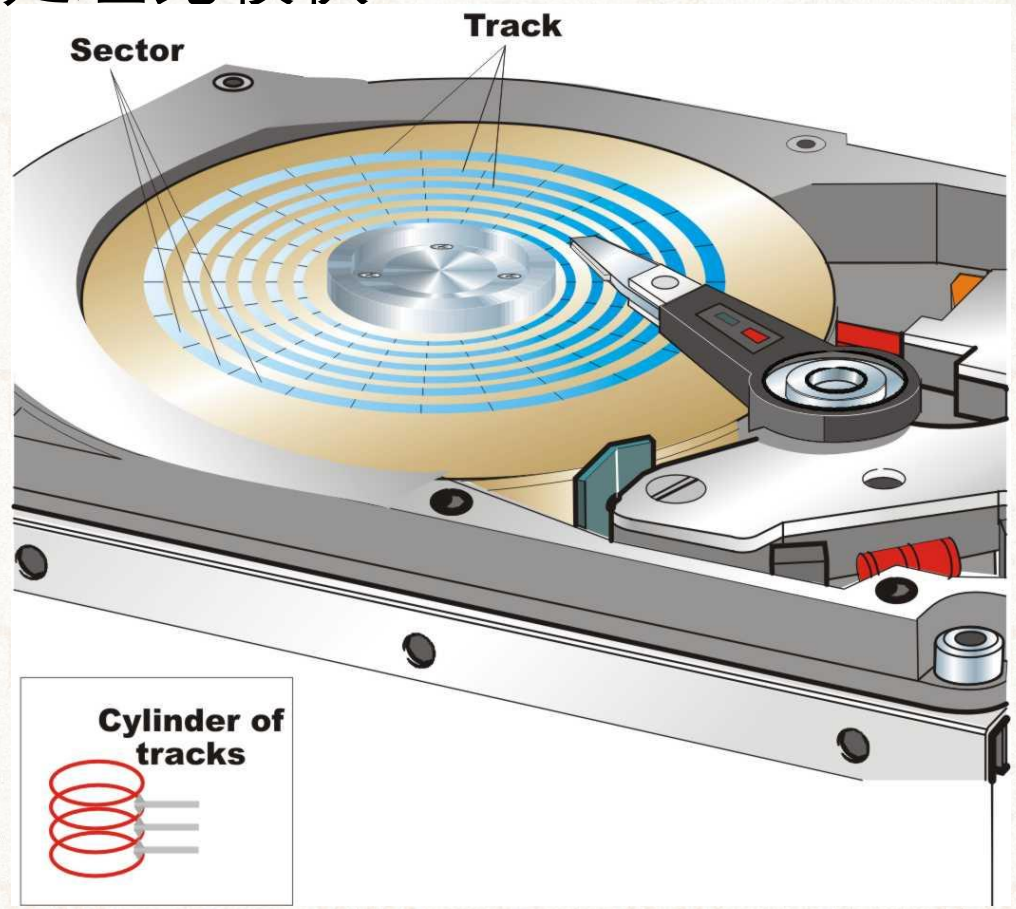
批量处理日志以减少磁盘写

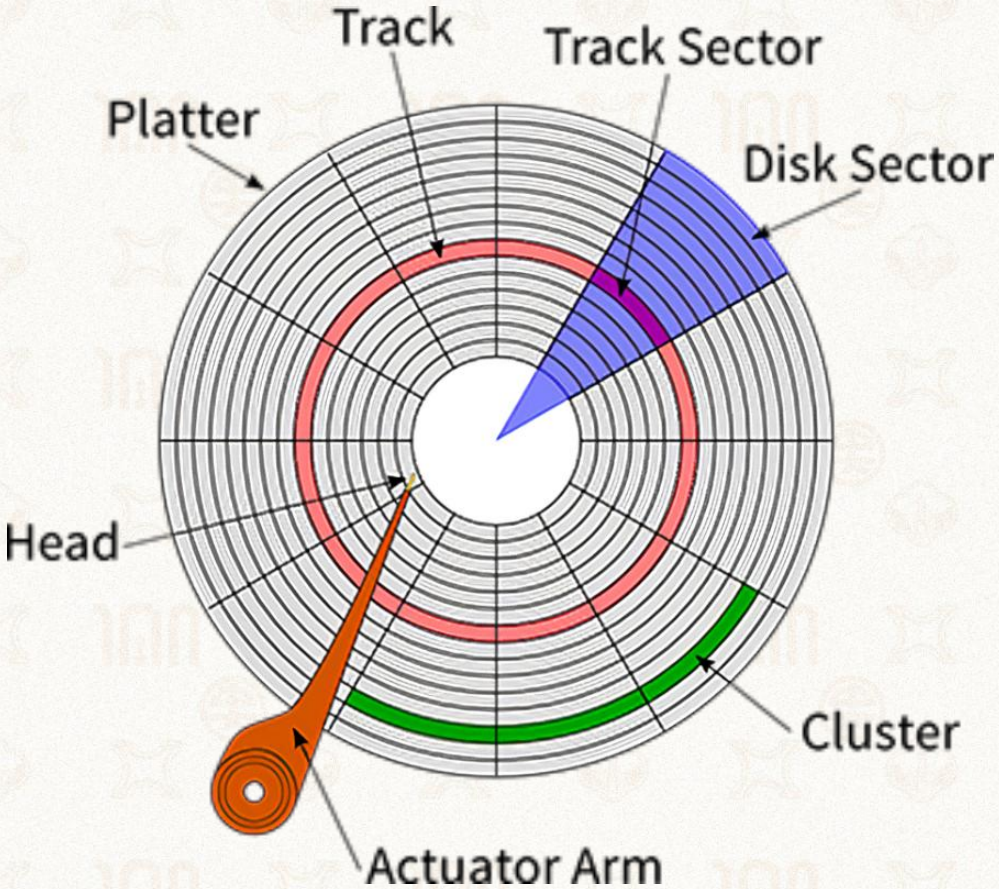
- 多个文件操作的日志合并在一起
- 每个修改过的块只需记录一次



为什么要批量处理日志：块(block)设备

- 磁盘内有多个轨道，探头一次读取一“块”数据
- 块处理比较快





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

大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
    - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

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

运行	• 新的原子操作可以加入到事务中
锁定	• 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
写入	• 事务正在被写入存储设备
提交	• 事务内容已经写入存储设备，正在写入提交块
完成	• 事务写入完毕

JBD2部分接口和使用方法

文件系统挂载时：

```
journal_t journal;
// 初始化日志系统（日志存在文件中）
journal = jbd2_journal_init_inode(inode)
// 读取并恢复已有日志（如果存在）
jbd2_journal_load(journal)
```

后台进程：

```
while (sleep_5s()) {
    // 提交事务和回收日志空间（并开始新的事务）
    jbd2_journal_commit_transaction(journal)
}
```

文件系统卸载时：

```
// 释放日志系统
jbd2_journal_destroy(journal)
```

系统调用处理：

```
// 不使用日志时的创建文件


// 1. 标记inode为占用
// bh: buffer_head 对应存储设备中的最小访问单元
bitmap_bh = read_inode_bitmap(sb, group)
set_bit(ino, bitmap_bh->b_data)
// 2. 初始化inode
inode_bh = get_inode_bh(sb, ino)
init_inode(inode_bh)
// 3. 将目录项写入目录中
data_bh = get_data_page(dir_inode)
add_dentry_to_data(page, filename, ino)
```

JBD2部分接口和使用方法

文件系统挂载时：

```
journal_t journal;
// 初始化日志系统（日志存在文件中）
journal = jbd2_journal_init_inode(inode)
// 读取并恢复已有日志（如果存在）
jbd2_journal_load(journal)
```

后台进程：

```
while (sleep_5s()) {
    // 提交事务和回收日志空间（并开始新的事务）
    jbd2_journal_commit_transaction(journal)
}
```

文件系统卸载时：

```
// 释放日志系统
jbd2_journal_destroy(journal)
```

系统调用处理：

开始新的原子操作

```
handle_t handle;
// 原子操作：创建新文件
handle = jbd2_journal_start(journal, nblocks=8)

// 1. 标记inode为占用
// bh: buffer_head 对应存储设备中的最小访问单元
bitmap_bh = read_inode_bitmap(sb, group)
set_bit(ino, bitmap_bh->b_data)
// 2. 初始化inode
inode_bh = get_inode_bh(sb, ino)
init_inode(inode_bh)
// 3. 将目录项写入目录中
data_bh = get_data_page(dir_inode)
add_dentry_to_data(page, filename, ino)
```

JBD2部分接口和使用方法

文件系统挂载时：

```
journal_t journal;
// 初始化日志系统（日志存在文件中）
journal = jbd2_journal_init_inode(inode)
// 读取并恢复已有日志（如果存在）
jbd2_journal_load(journal)
```

后台进程：

```
while (sleep_5s()) {
    // 提交事务和回收日志空间（并开始新的事务）
    jbd2_journal_commit_transaction(journal)
}
```

文件系统卸载时：

```
// 释放日志系统
jbd2_journal_destroy(journal)
```

系统调用处理：

通知jbd2即将修改bh中的数据

```
handle_t handle;
// 原子操作：创建新文件
handle = jbd2_journal_start(journal, nblocks=8)

// 1. 标记inode为占用
// bh: buffer_head 对应存储设备中的最小访问单元
bitmap_bh = read_inode_bitmap(sb, group)
jbd2_journal_get_write_access(handle, bitmap_bh)
set_bit(ino, bitmap_bh->b_data)
// 2. 初始化inode
inode_bh = get_inode_bh(sb, ino)
init_inode(inode_bh)
// 3. 将目录项写入目录中
data_bh = get_data_page(dir_inode)
add_dentry_to_data(page, filename, ino)
```

JBD2部分接口和使用方法

文件系统挂载时：

```
journal_t journal;
// 初始化日志系统（日志存在文件中）
journal = jbd2_journal_init_inode(inode)
// 读取并恢复已有日志（如果存在）
jbd2_journal_load(journal)
```

后台进程：

```
while (sleep_5s()) {
    // 提交事务和回收日志空间（并开始新的事务）
    jbd2_journal_commit_transaction(journal)
}
```

文件系统卸载时：

```
// 释放日志系统
jbd2_journal_destroy(journal)
```

系统调用处理：

通知jbd2，修改bh完毕

```
handle_t handle;
// 原子操作：创建新文件
handle = jbd2_journal_start(journal, nblocks=8)

// 1. 标记inode为占用
// bh: buffer_head 对应存储设备中的最小访问单元
bitmap_bh = read_inode_bitmap(sb, group)
jbd2_journal_get_write_access(handle, bitmap_bh)
set_bit(ino, bitmap_bh->b_data)
jbd2_journal_dirty_metadata(handle, bitmap_bh)
// 2. 初始化inode
inode_bh = get_inode_bh(sb, ino)
init_inode(inode_bh)
// 3. 将目录项写入目录中
data_bh = get_data_page(dir_inode)
add_dentry_to_data(page, filename, ino)
```

JBD2部分接口和使用方法

文件系统挂载时：

```
journal_t journal;
// 初始化日志系统（日志存在文件中）
journal = jbd2_journal_init_inode(inode)
// 读取并恢复已有日志（如果存在）
jbd2_journal_load(journal)
```

后台进程：

```
while (sleep_5s()) {
    // 提交事务和回收日志空间（并开始新的事务）
    jbd2_journal_commit_transaction(journal)
}
```

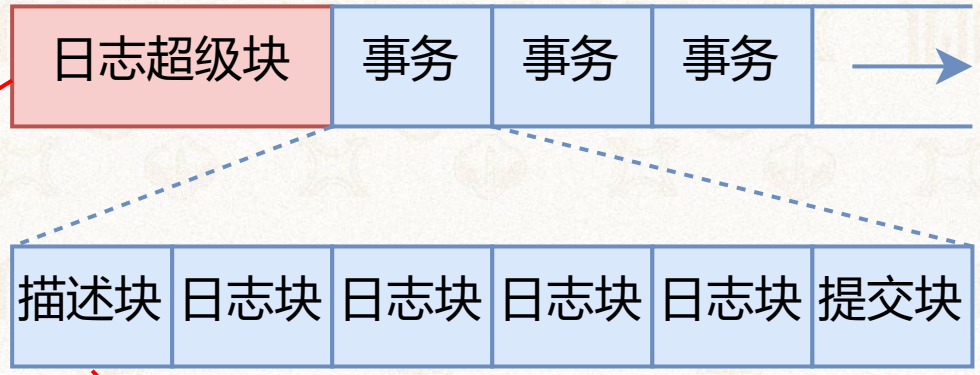
文件系统卸载时：

```
// 释放日志系统
jbd2_journal_destroy(journal)
```

系统调用处理：

```
handle_t handle;
// 原子操作：创建新文件
handle = jbd2_journal_start(journal, nblocks=8)
// 1. 标记inode为占用
// bh: buffer_head 对应存储设备中的最小访问单元
bitmap_bh = read_inode_bitmap(sb, group)
jbd2_journal_get_write_access(handle, bitmap_bh)
set_bit(ino, bitmap_bh->b_data)
jbd2_journal_dirty_metadata(handle, bitmap_bh)
// 2. 初始化inode
inode_bh = get_inode_bh(sb, ino)
jbd2_journal_get_write_access(handle, inode_bh)
init_inode(inode_bh)
jbd2_journal_dirty_metadata(handle, inode_bh)
// 3. 将目录项写入目录中
data_bh = get_data_page(dir_inode)
jbd2_journal_get_write_access(handle, data_bh)
add_dentry_to_data(page, filename, ino)
jbd2_journal_dirty_metadata(handle, data_bh)
jbd2_journal_stop(handle) // 结束原子操作
```

JBD2日志的磁盘结构




```
typedef struct journal_header_s {
__be32 h_magic; /* 魔法数字 */
__be32 h_blocktype; /* 块的类型 */
__be32 h_sequence; /* 块序号 */
} journal_header_t;

typedef struct journal_superblock_s {
journal_header_t s_header;
__be32 s_blocksize; /* 块大小 */
__be32 s_maxlen; /* 日志文件的块总数 */
__be32 s_first; /* 日志信息的第一个块号 */
// ...
__be32 s_sequence; /* first commit ID expected in log */
__be32 s_start; /* 日志的开始块号 */
// ...
__be32 s_checksum; /* 校验码 */
// ...
} journal_superblock_t;
```

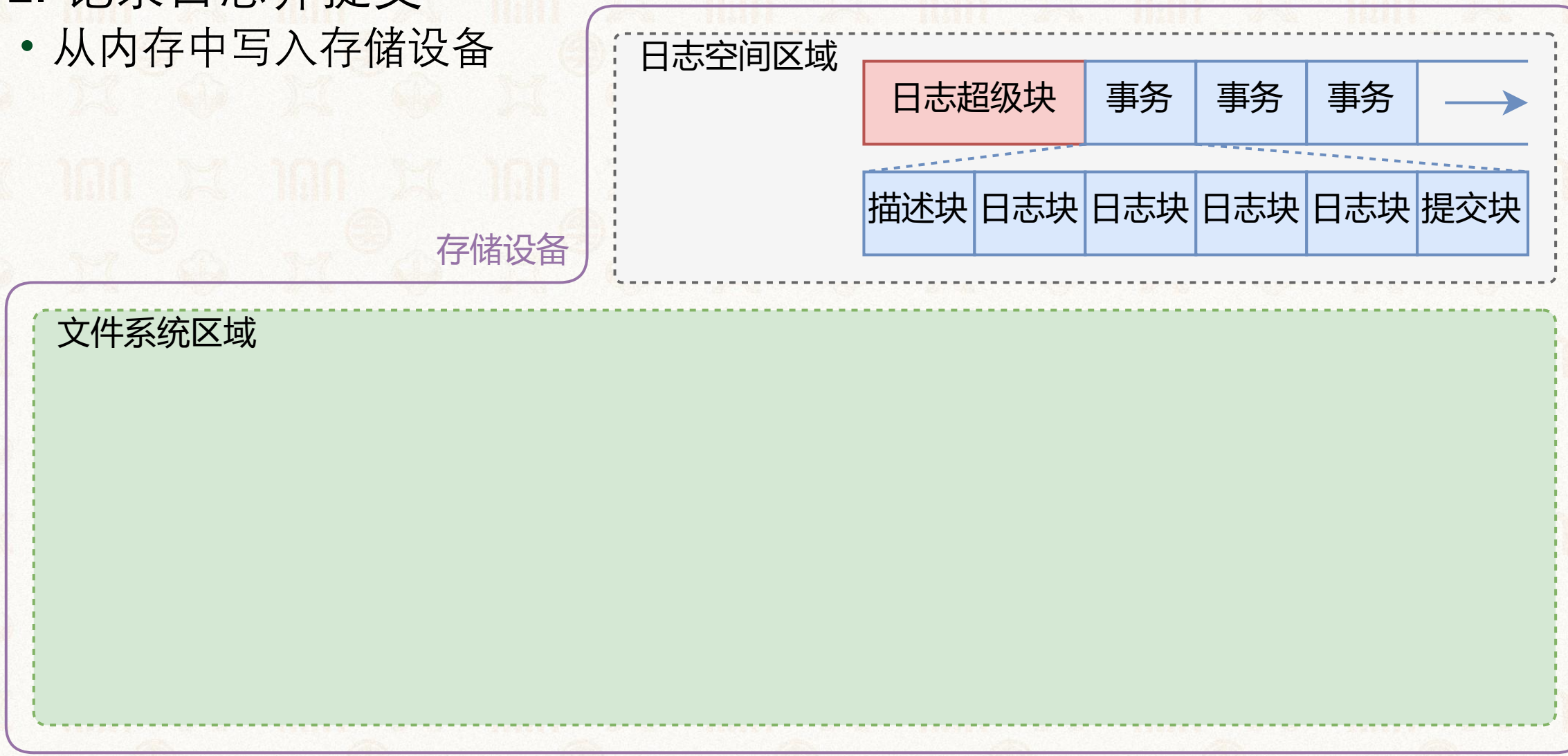
tag数组，每个tag对应后面日志块的信息

```
typedef struct journal_block_tag_s {
__be32 t_blocknr; /* 磁盘上的块号 */
__be16 t_checksum; /* 校验码*/
__be16 t_flags; /* 一些标志位 */
__be32 t_blocknr_high; /* 高32位 */
} journal_block_tag_t;
```

恢复时，根据描述块中记录的tag信息，将后面日志块中的数据写入到对应的磁盘块中

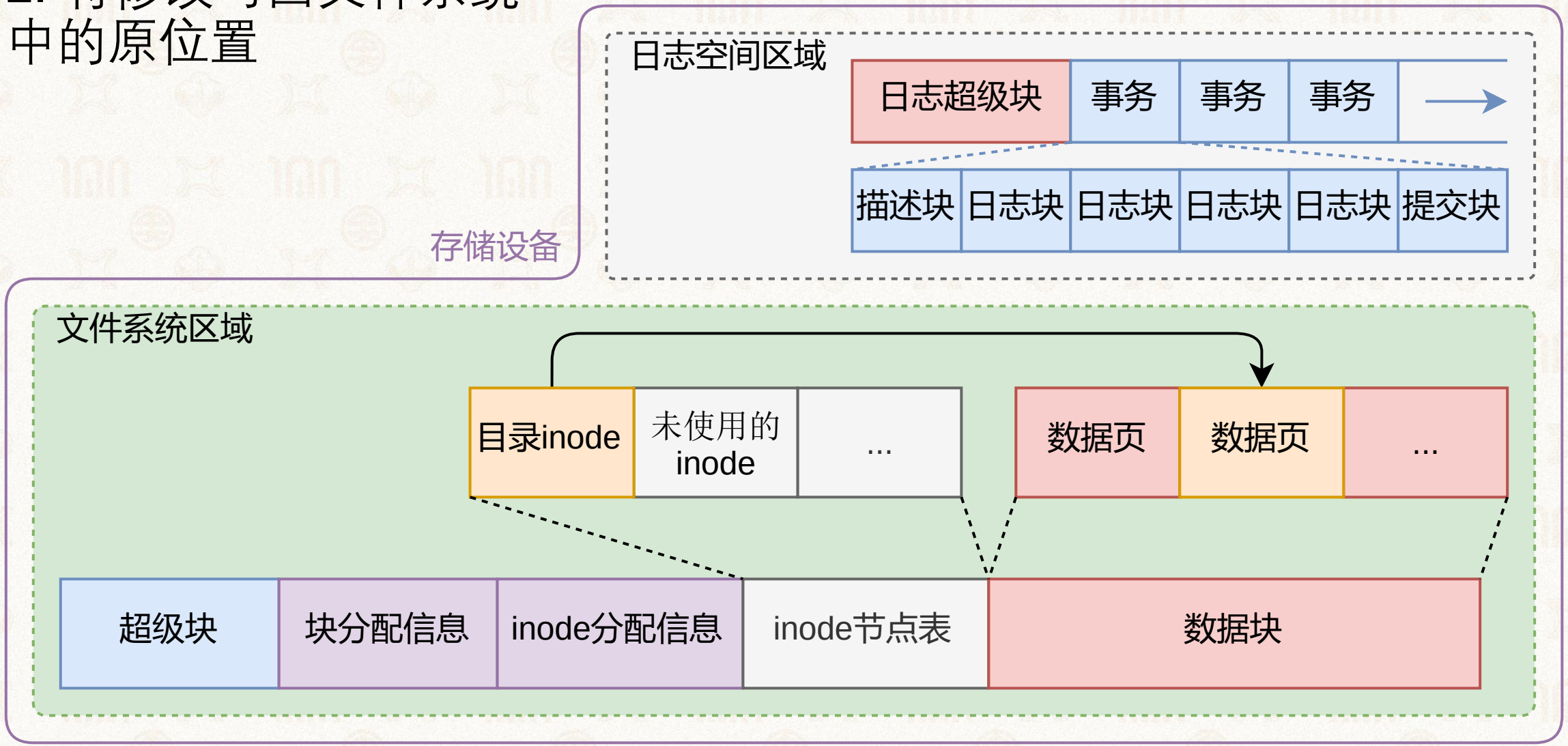
JBD2日志使用过程

- 1. 记录日志并提交
  - 从内存中写入存储设备



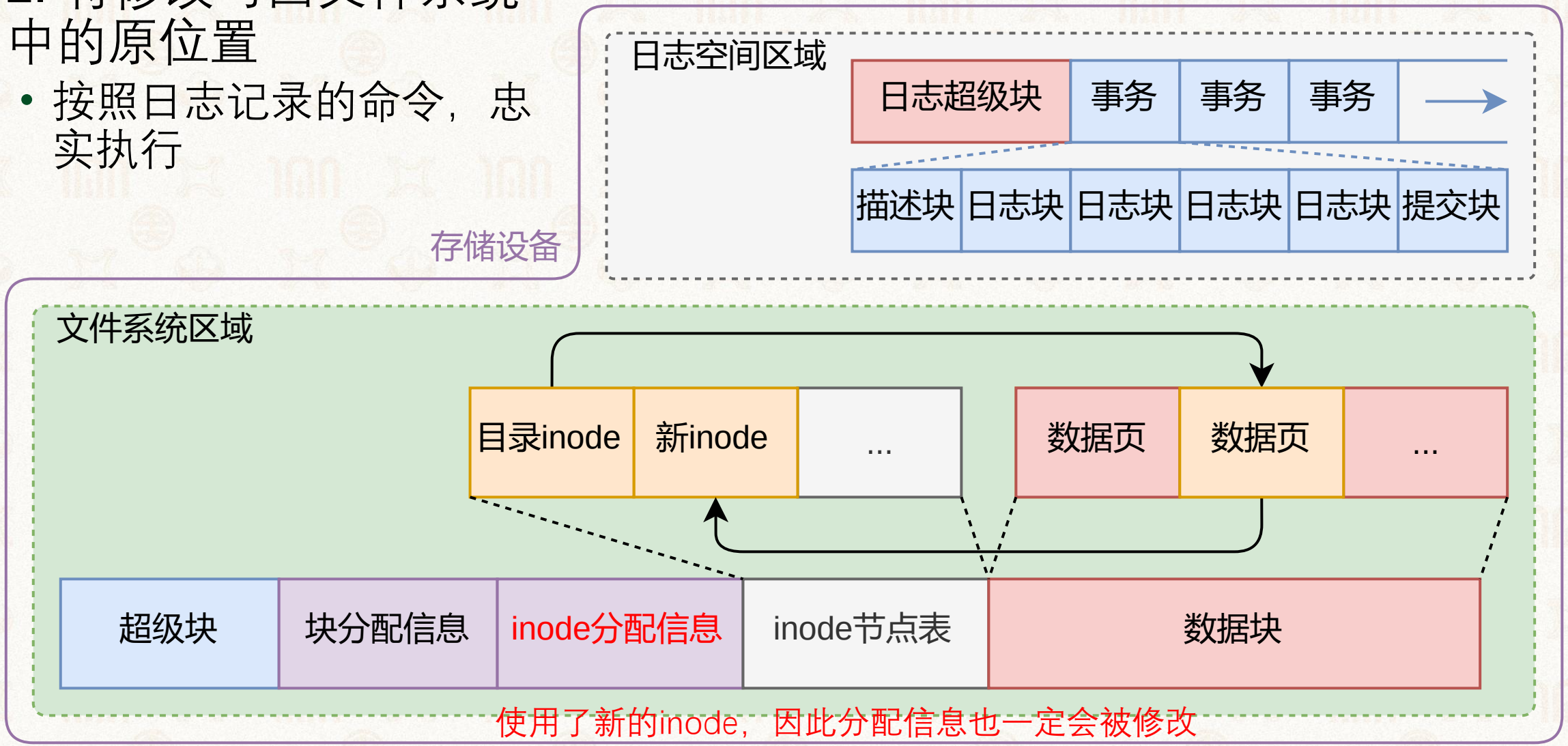
JBD2日志使用过程

- 2. 将修改写回文件系统中的原位置



JBD2日志使用过程

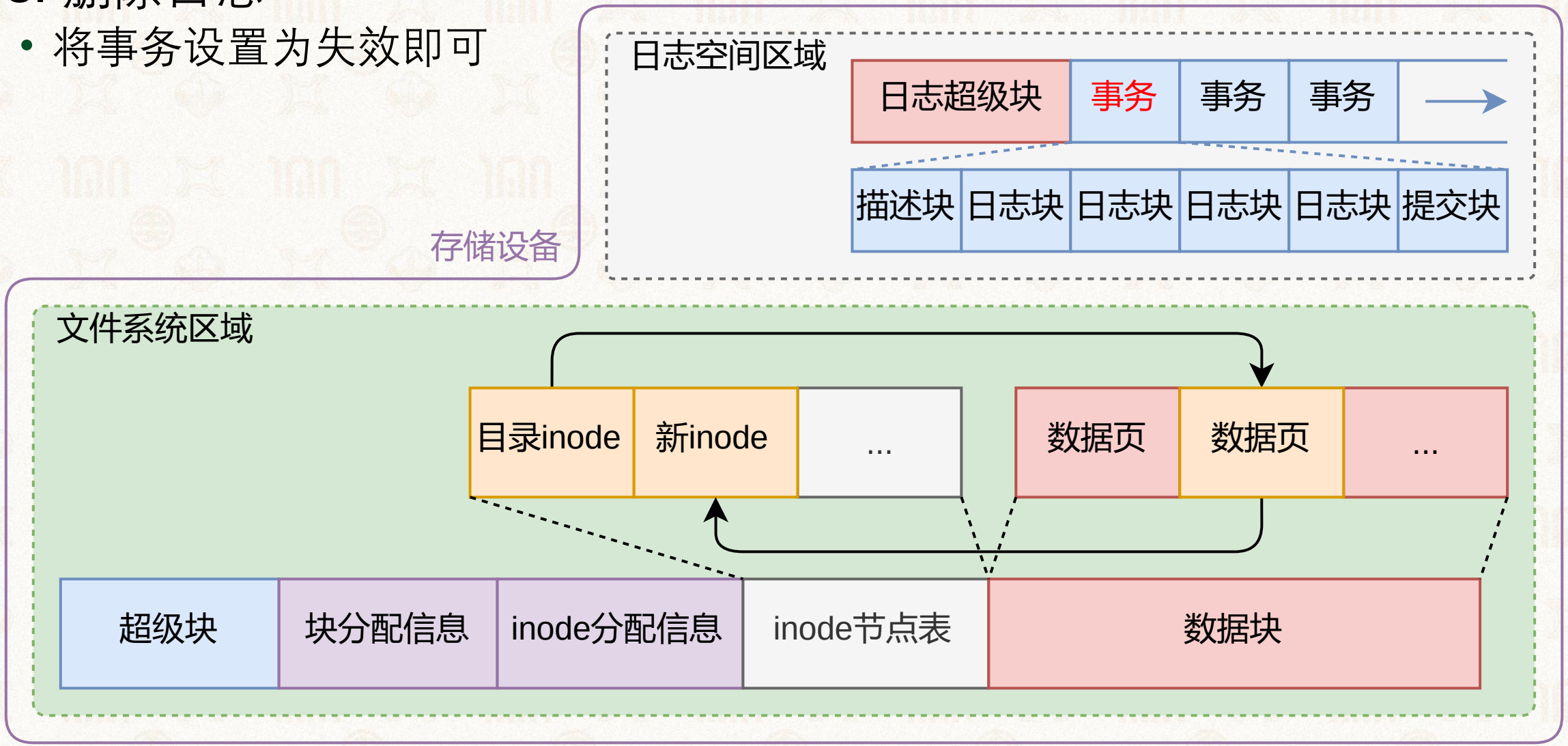
- 2. 将修改写回文件系统中的原位置
  - 按照日志记录的命令，忠实执行



JBD2日志使用过程

3. 删除日志

- 将事务设置为失效即可



JBD2日志使用过程

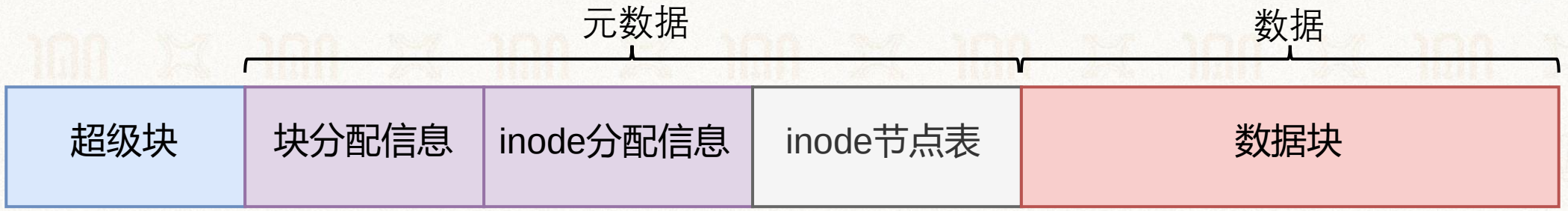
- 和崩溃一致性有什么关系？

- 我们不是追求不丢数据，而是尽量保证文件系统结构不被损坏(一致性)

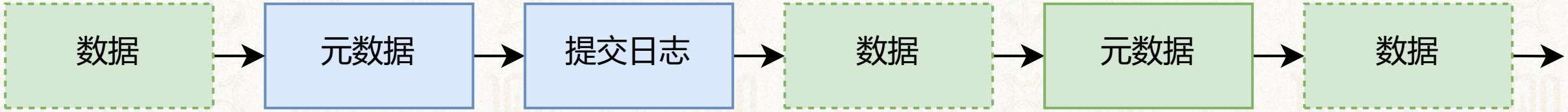
- 事务如果没写完就崩溃了
  - 扔掉不完整的事务就好

- 事务写完之后再崩溃
  - 根据事务的记录，重启时可以恢复对文件系统的更改

Ext4用JBD2实现的三种日志模式

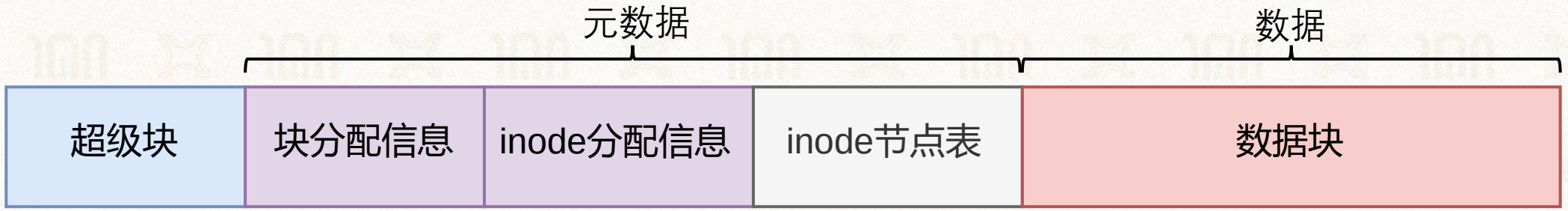


- 写回(writeback)模式：日志只记录元数据
  - 最快，但是一致性最差!
  - 保结构、不保数据

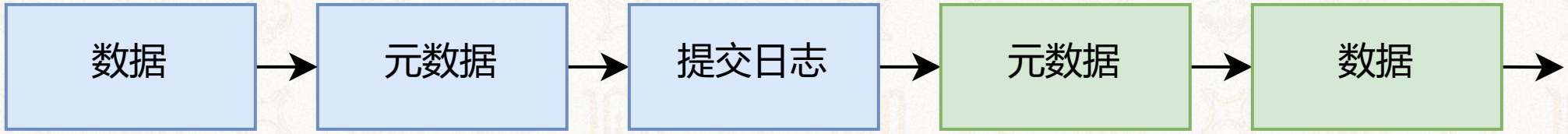


日志　有一致性保证地写入硬盘　无一致性保证地写入硬盘

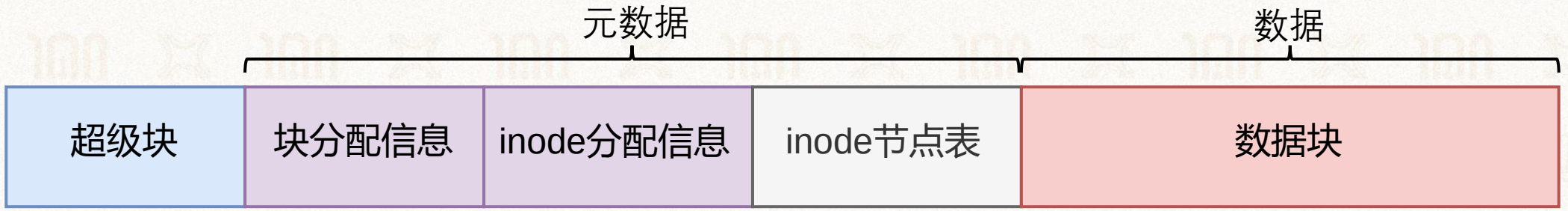
Ext4用JBD2实现的三种日志模式



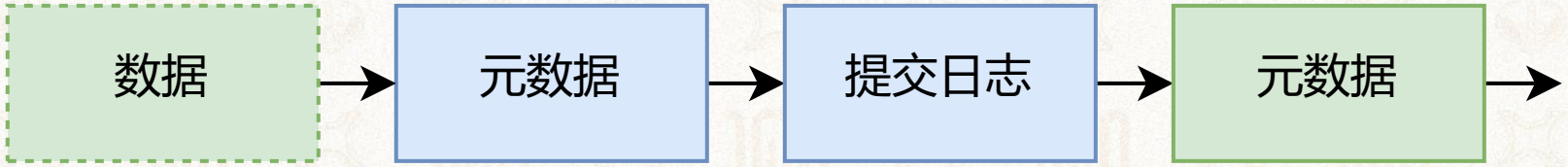
- 日志(journal)模式(完整版)：元数据和数据均使用日志记录
  - 一致性最好，但数据写两次



Ext4用JBD2实现的三种日志模式



- 顺序(ordered)模式：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
  - 新数据可能会被覆盖，但结构不会乱



日志　有一致性保证地写入硬盘　无一致性保证地写入硬盘

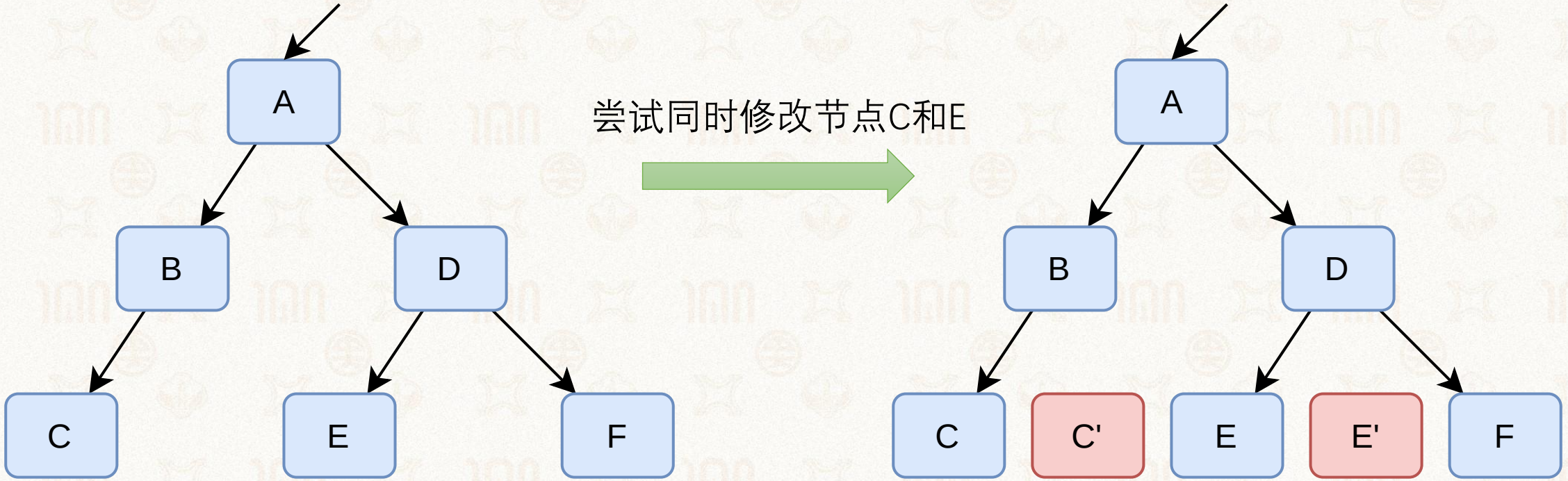
大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
  - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

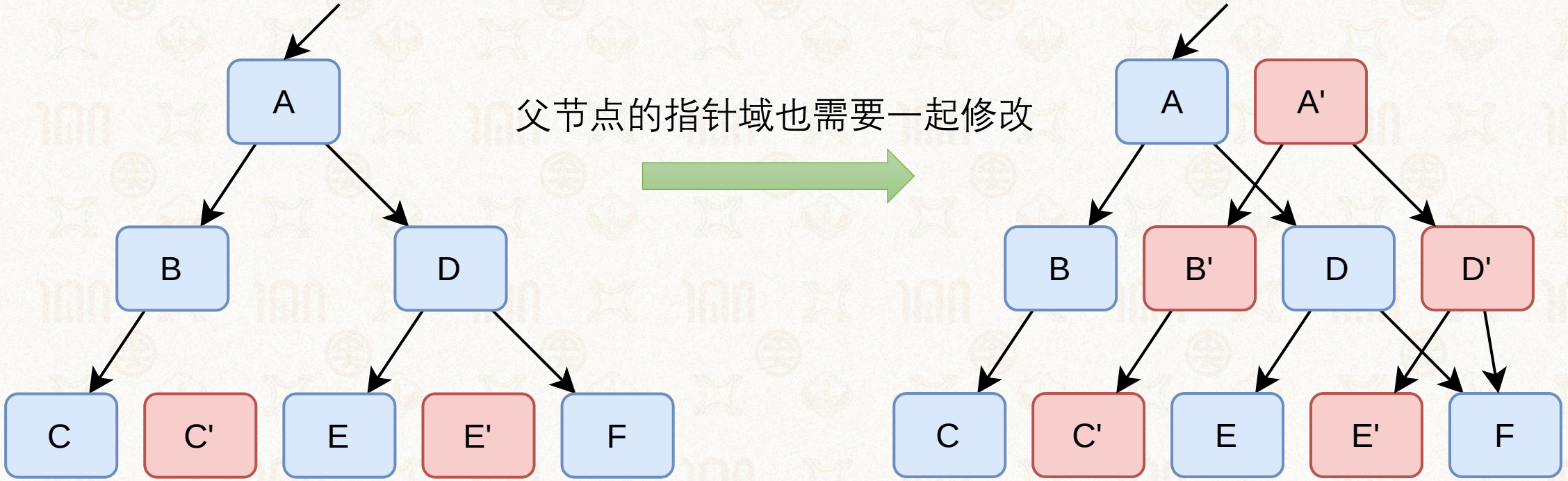
写时复制（Copy-on-Write）

- 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



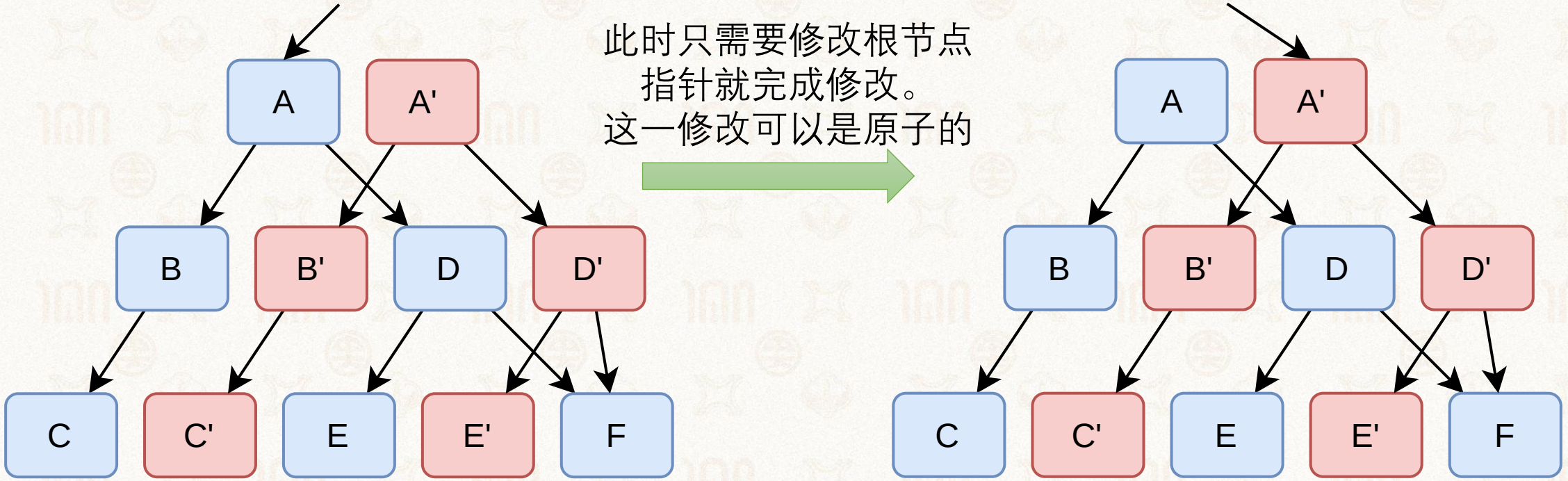
写时复制（Copy-on-Write）

- 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



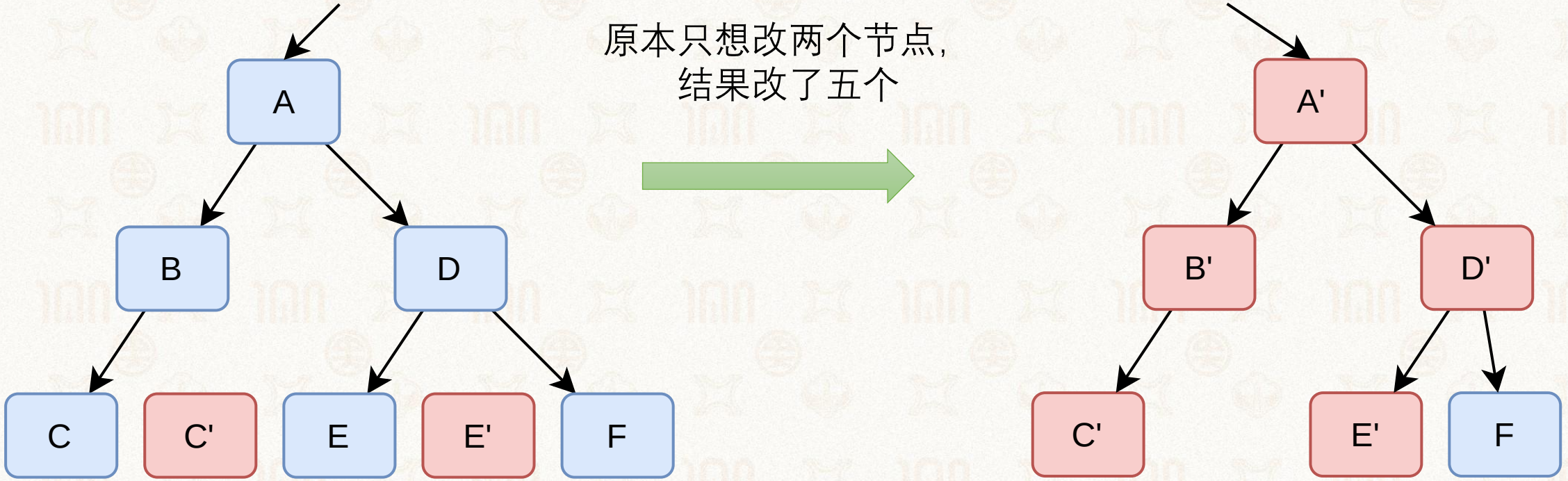
写时复制（Copy-on-Write）

- 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



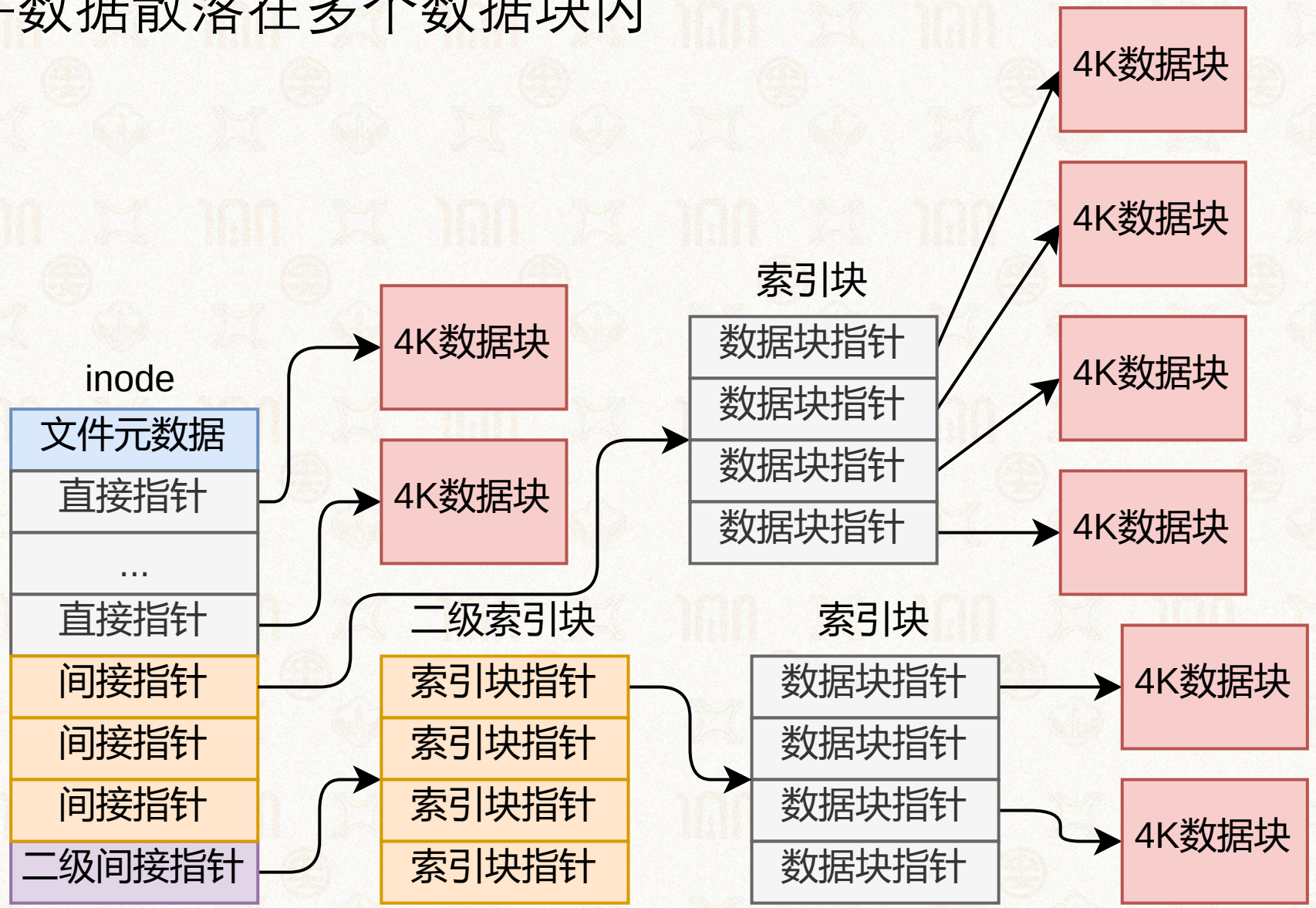
写时复制（Copy-on-Write）

- 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
- 写放大问题：实际修改量比用户预期修改量要大



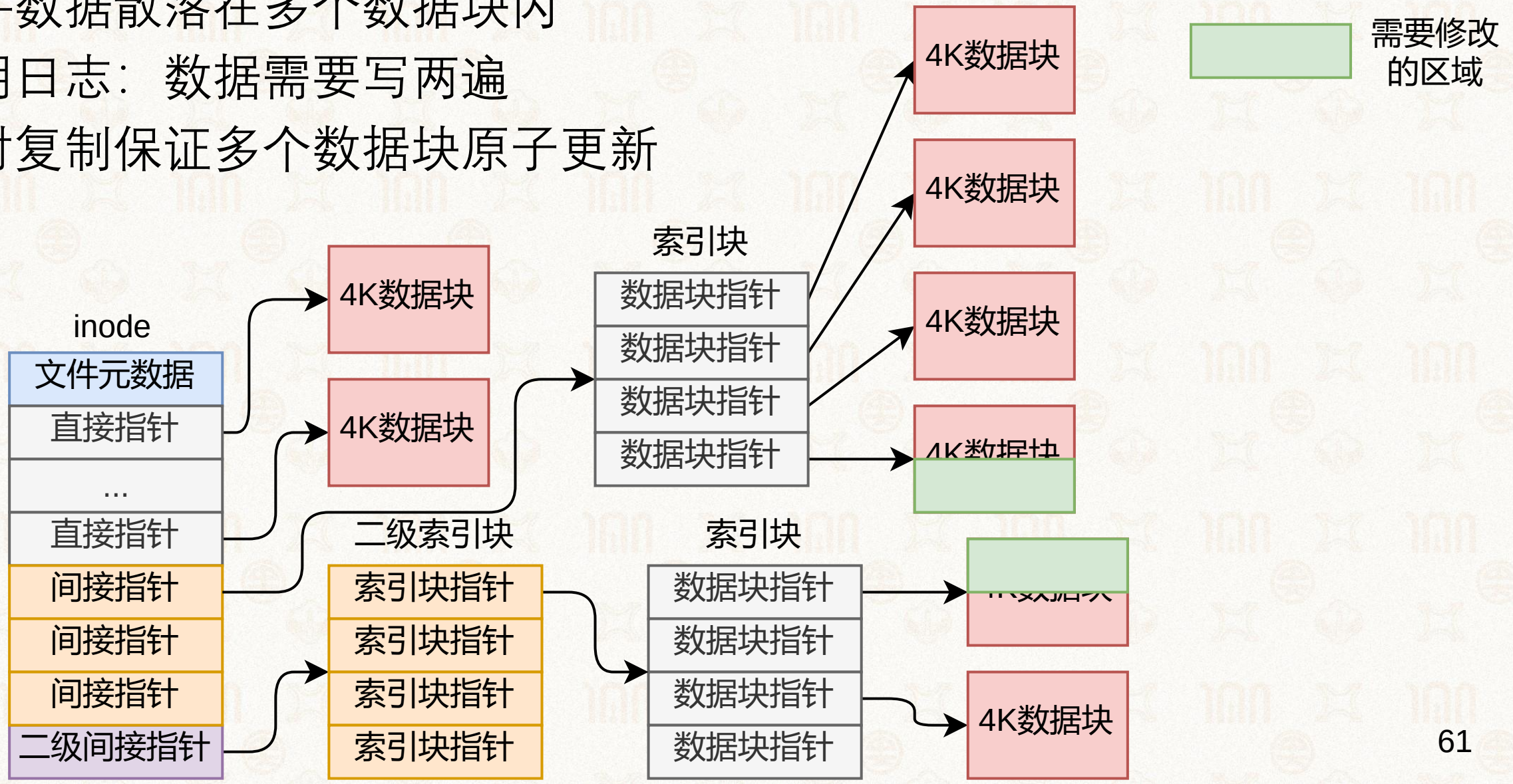
文件中的写时复制

- 文件数据散落在多个数据块内



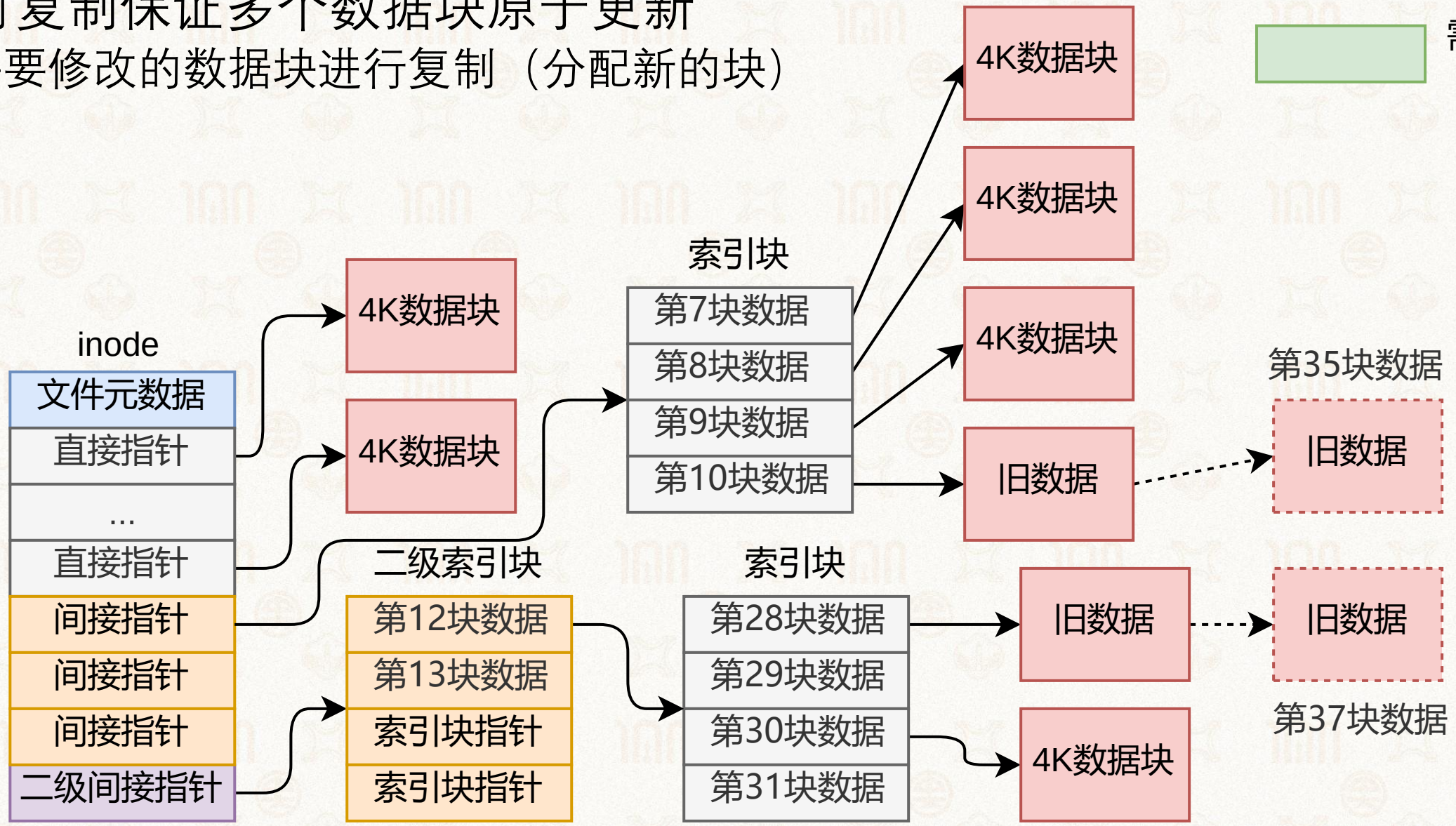
文件中的写时复制

- 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



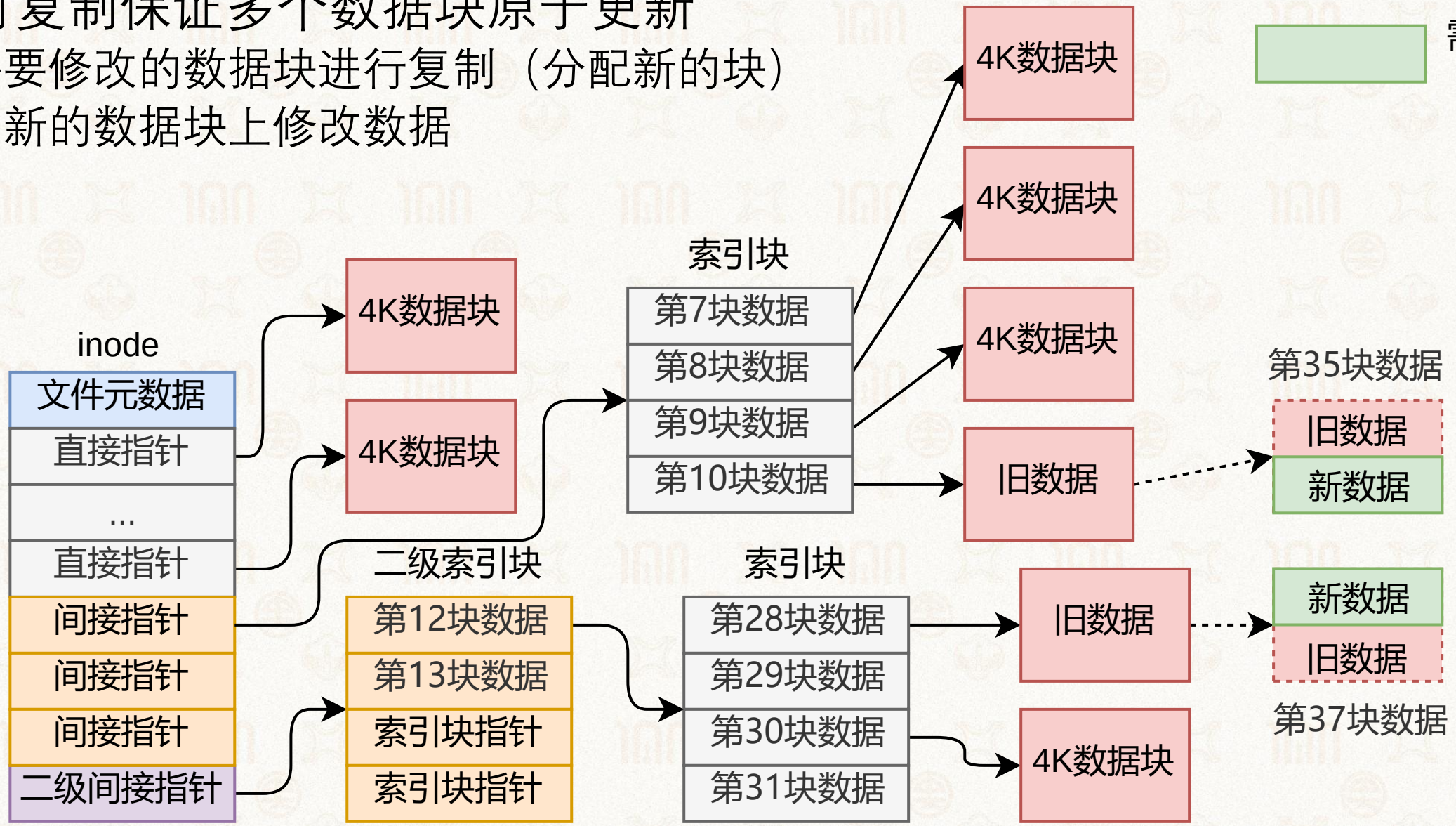
文件中的写时复制

- 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）



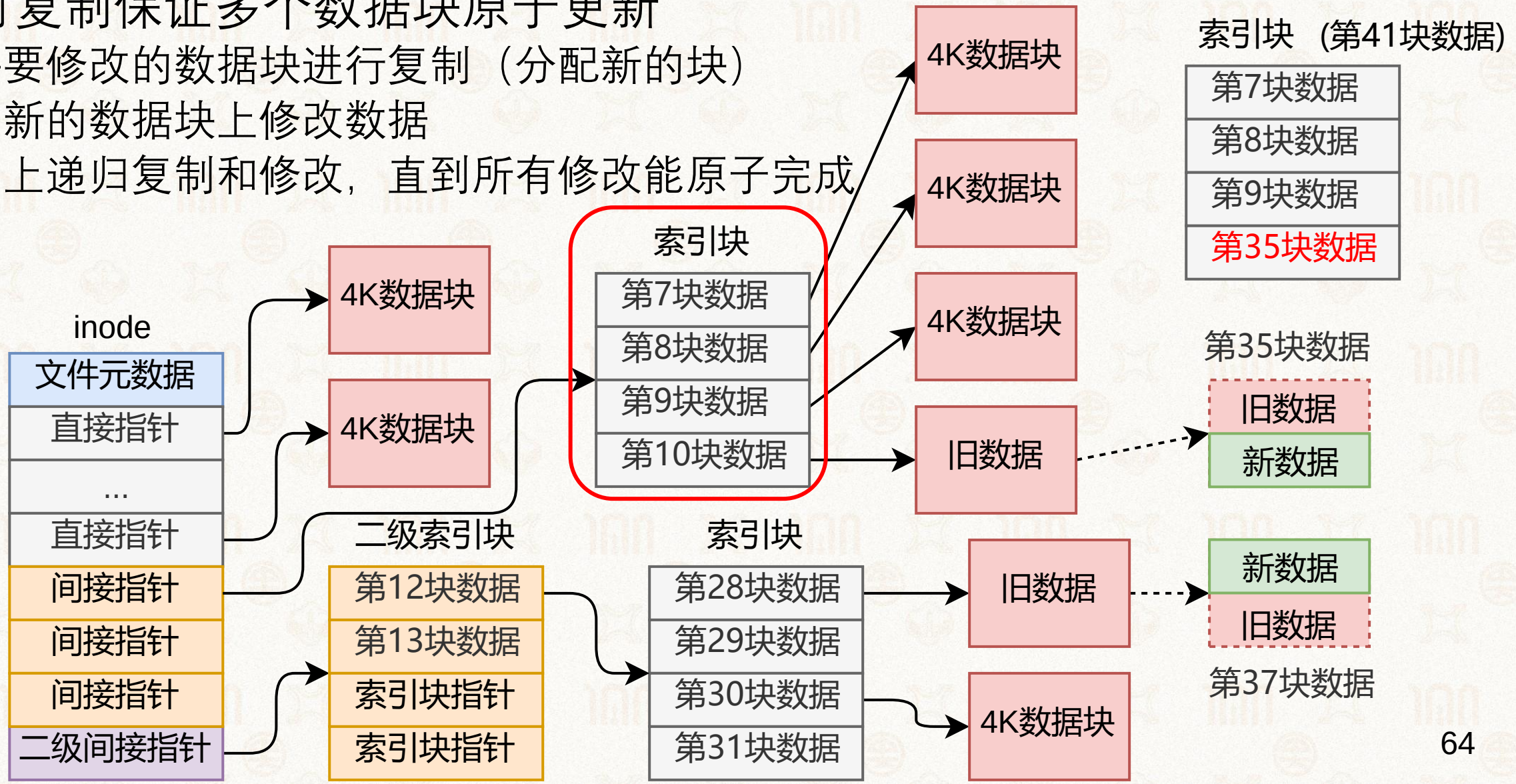
文件中的写时复制

- 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



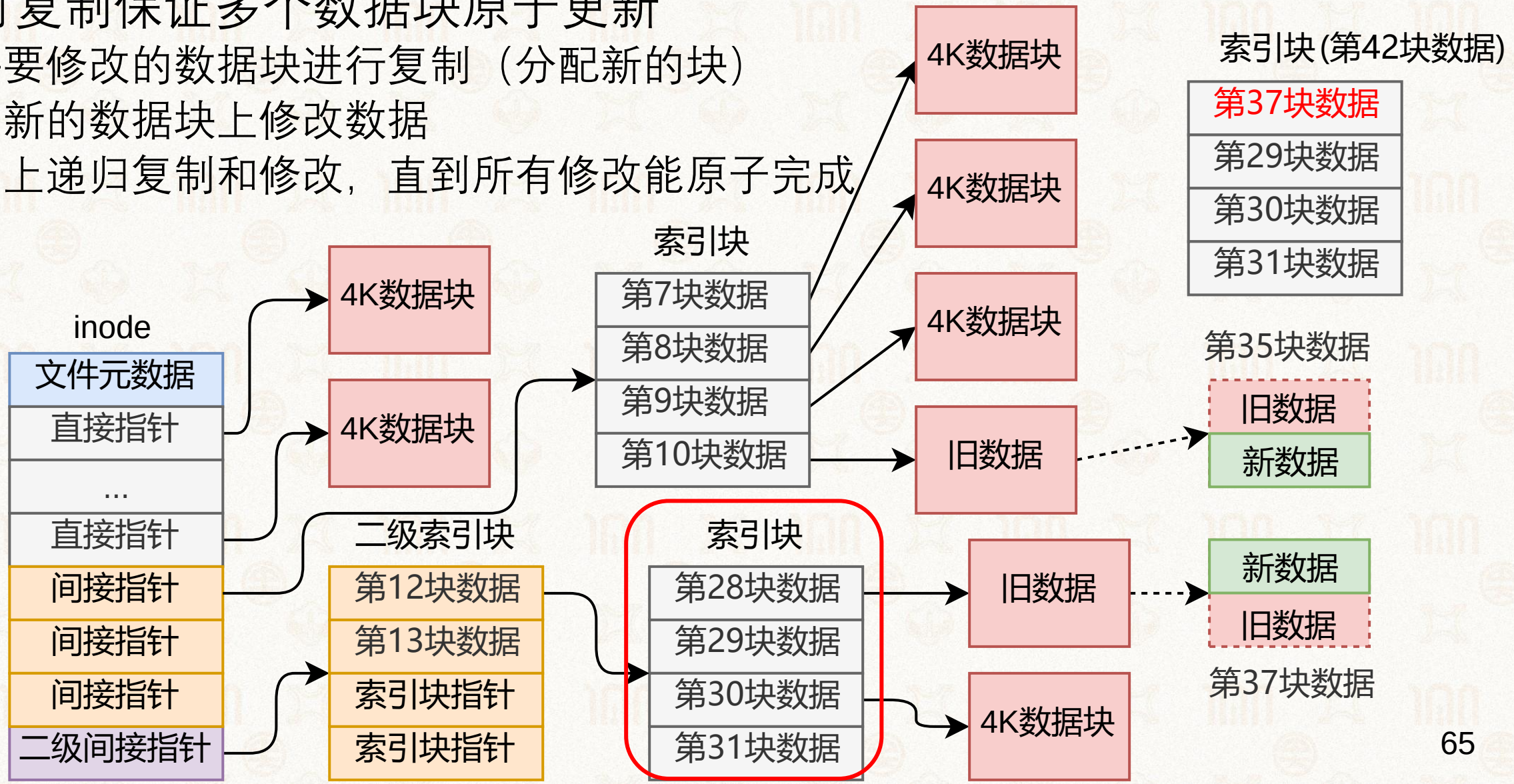
文件中的写时复制

- 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



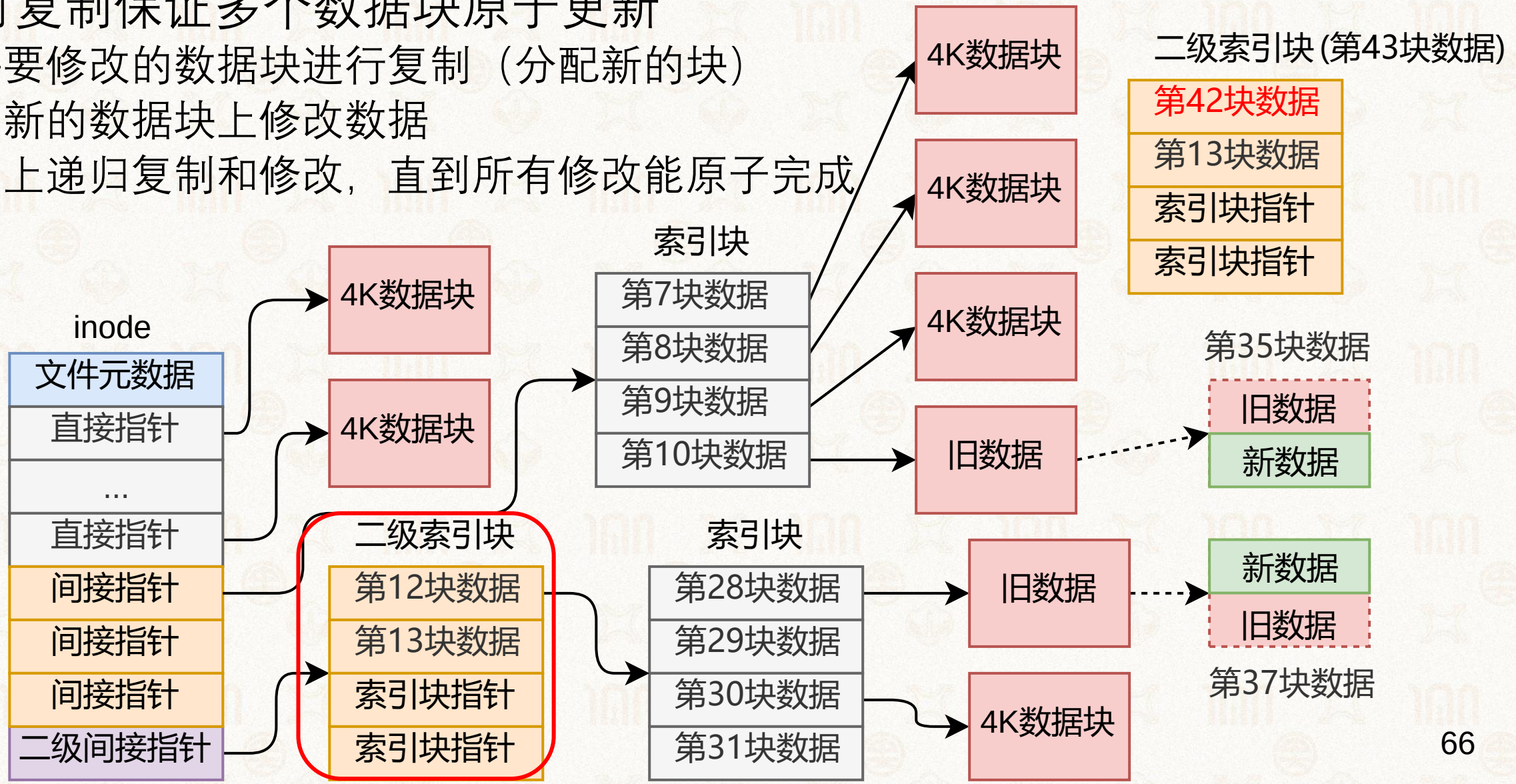
文件中的写时复制

- 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



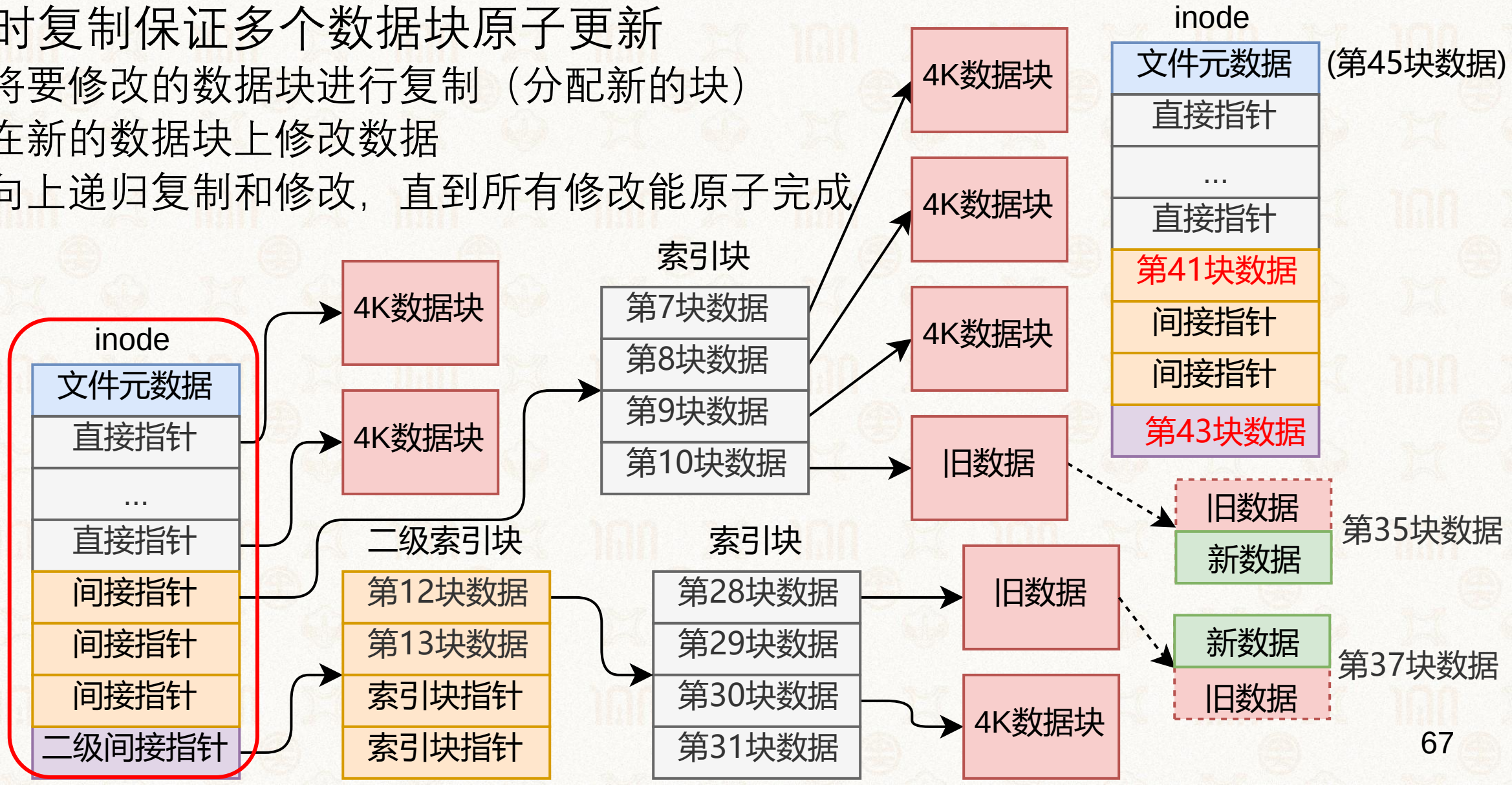
文件中的写时复制

- 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



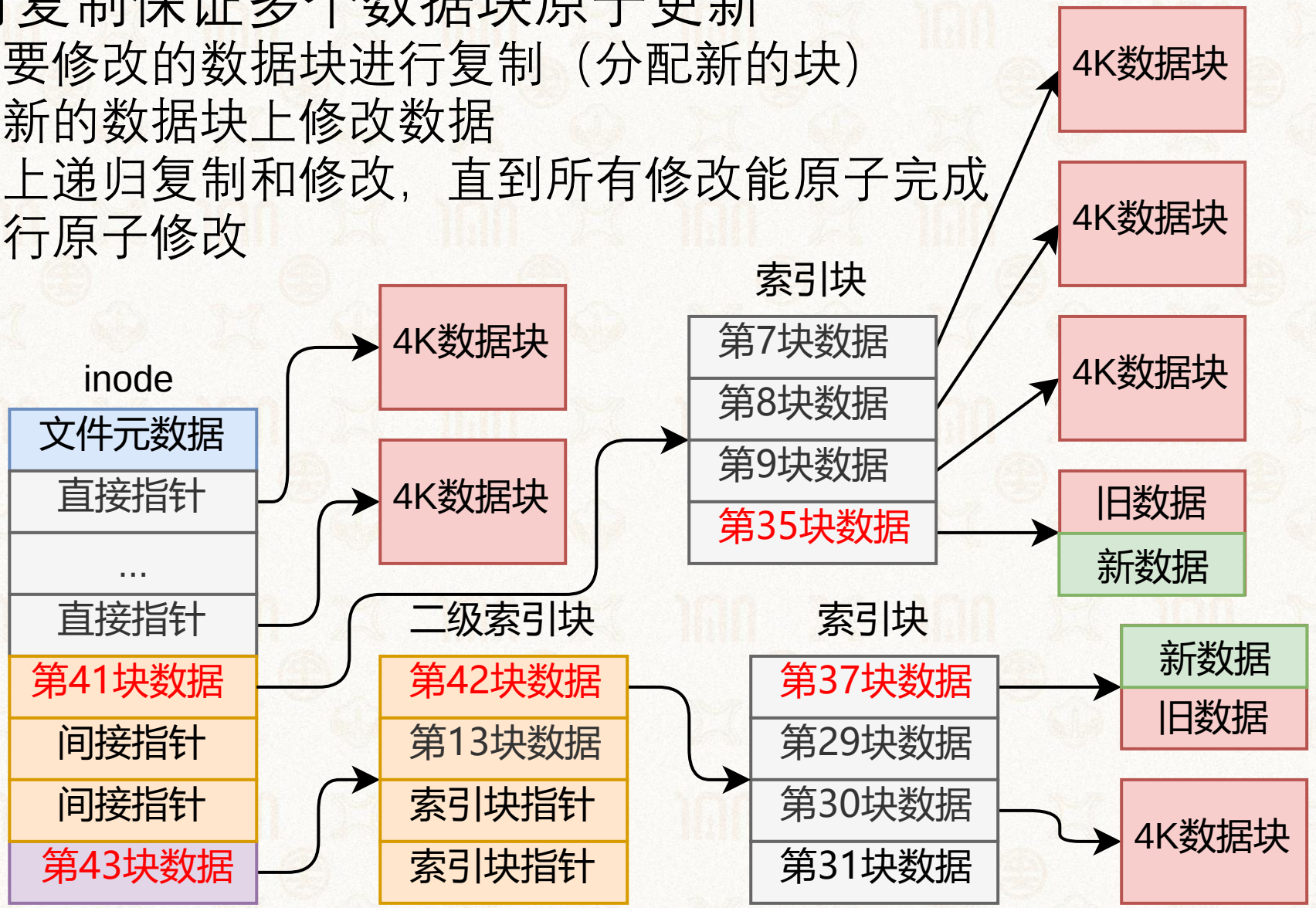
文件中的写时复制

- 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



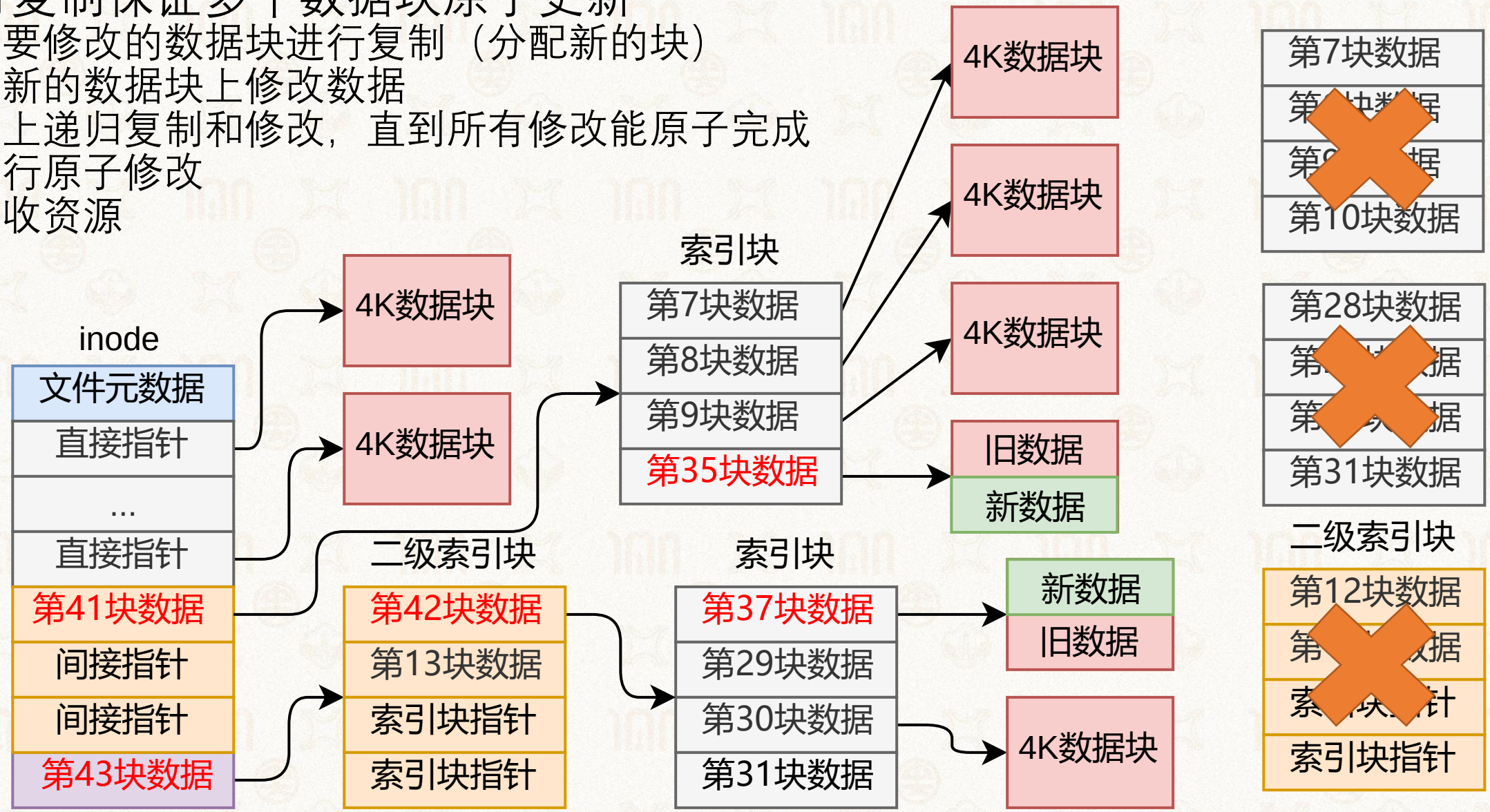
文件中的写时复制

- 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



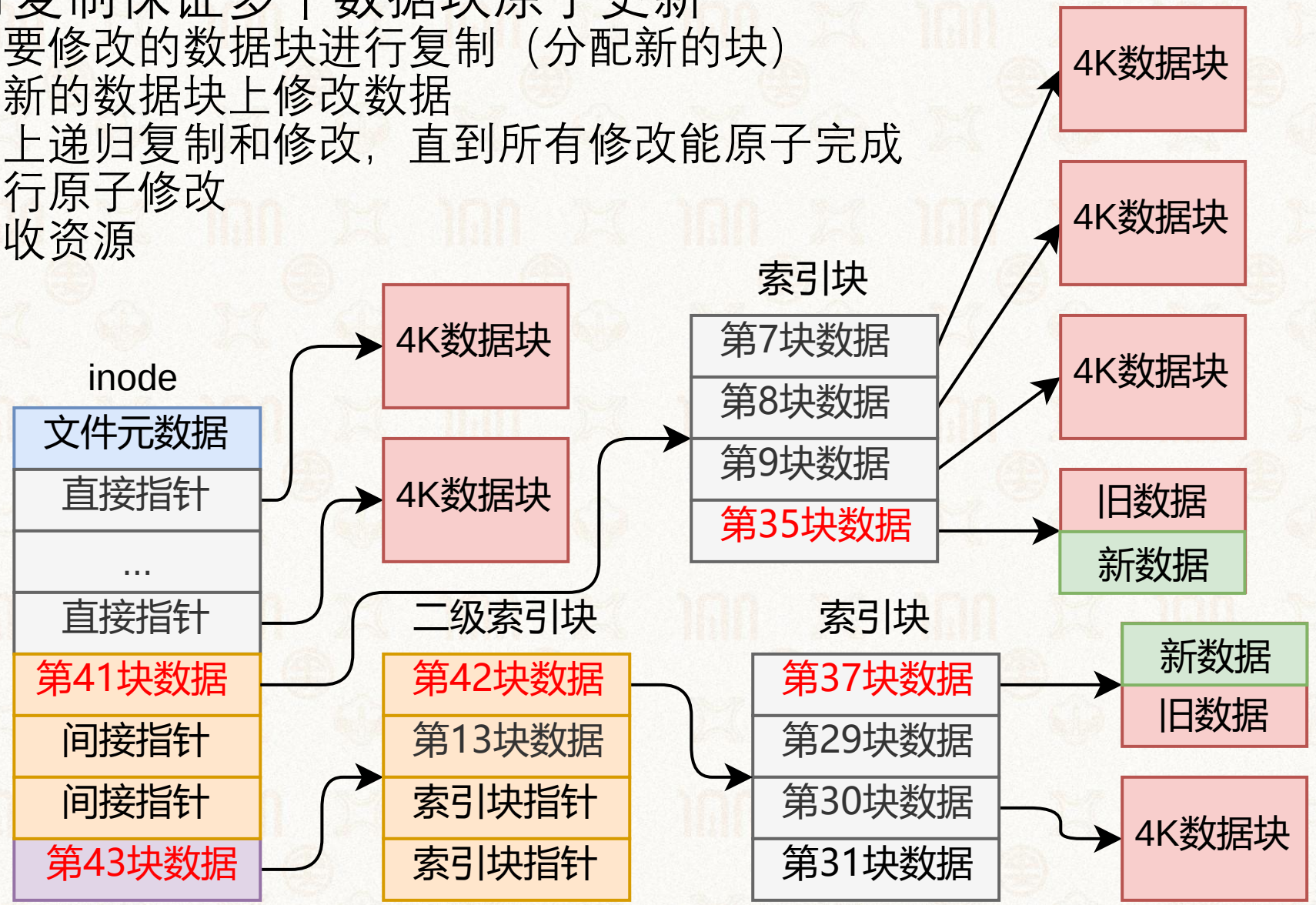
文件中的写时复制

- 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



文件中的写时复制

- 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？

主观题 10分

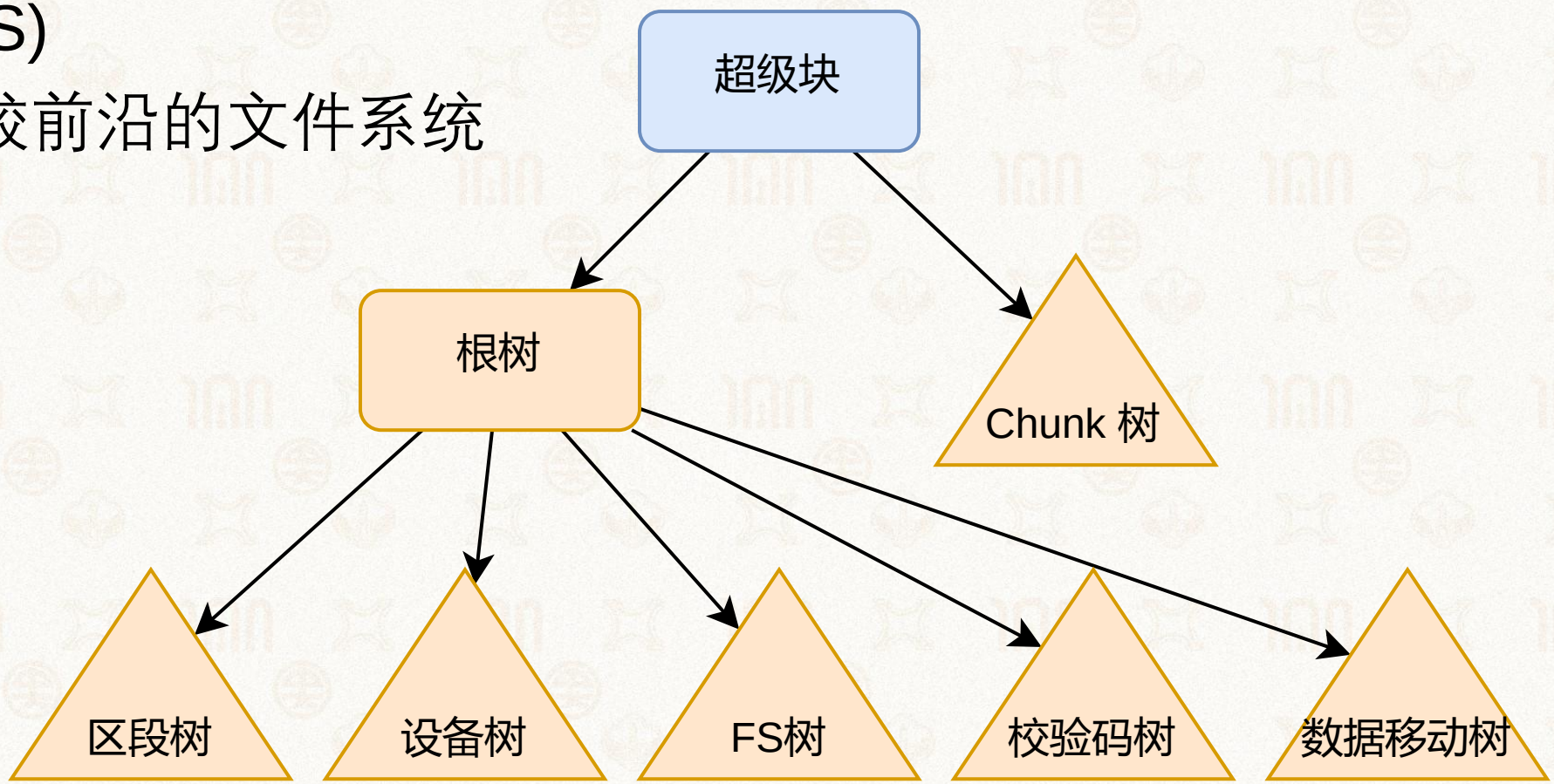
设置

写时复制和日志各自的优缺点有哪些?

作答

"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)
- 目前还是个比较前沿的文件系统
  - 可能不稳定



如何看待 Fedora 33 将默认使用 btrfs 文件系统? (https://www.zhihu.com/question/407386653)

大纲

- 文件系统崩溃一致性是什么
  - 文件系统一致性约束
  - 崩溃与恢复
- 日志文件系统
- 原子更新技术
  - 日志
    - 日志系统JBD2
  - 写时复制
- Soft Updates
  - 不详细讲，太复杂，有兴趣同学自己看

保障文件系统一致性的两大类技术

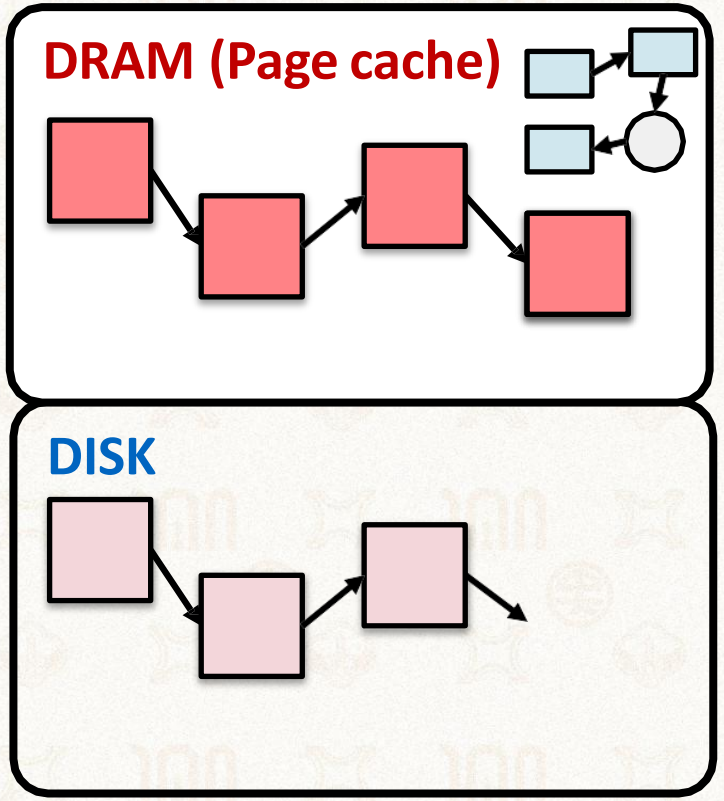
Soft Updates

- 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：
      - 标记inode为占用
      - 初始化inode
      - 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

Soft Updates的总体思想

- 最新的元数据在内存中
  - 在内存中记录数据的依赖关系
    - 有向无环图
  - 读写操作可达到内存的性能
  - 不需要同步执行磁盘写操作
- 磁盘中的元数据总是一致的
  - 有依赖关系的数据结构以原子形式写入
  - 总是一致的
  - 崩溃后可立即使用，不用恢复



Soft Updates

Soft Updates的三个次序规则

- 不要指向一个未初始化的结构
  - 如：目录项指向一个inode之前，改inode结构应该先被初始化
- 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
- 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：
    - 标记inode为占用（对bitmap的修改）
    - 初始化inode（对inode的修改，依赖于1）
    - 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）

- 实现十分复杂
  - 忽略

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫