IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

文件系统崩溃一致性

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

https://ipads.se.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

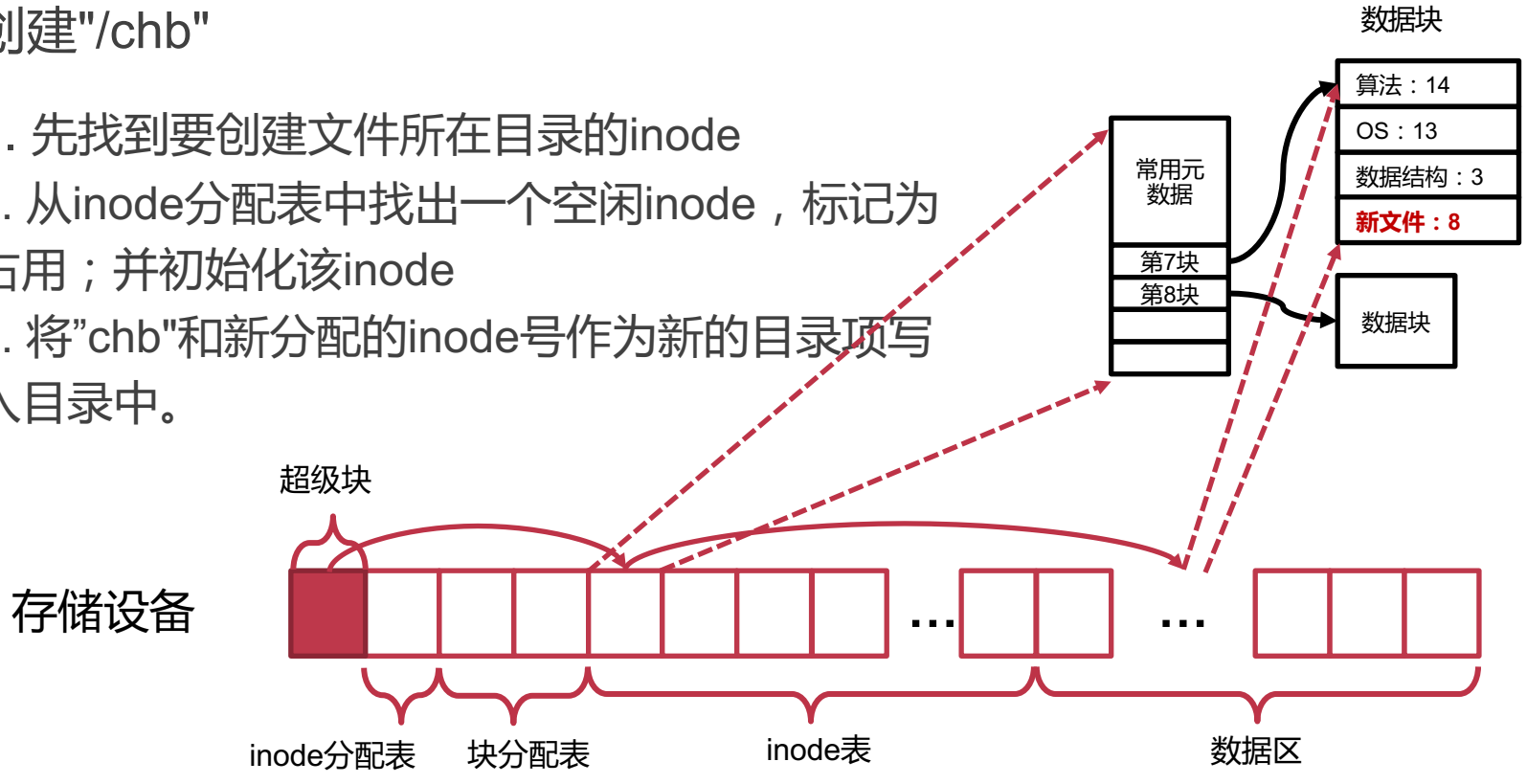
文件系统的崩溃一致性

- 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：文件的创建

创建"/chb"

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



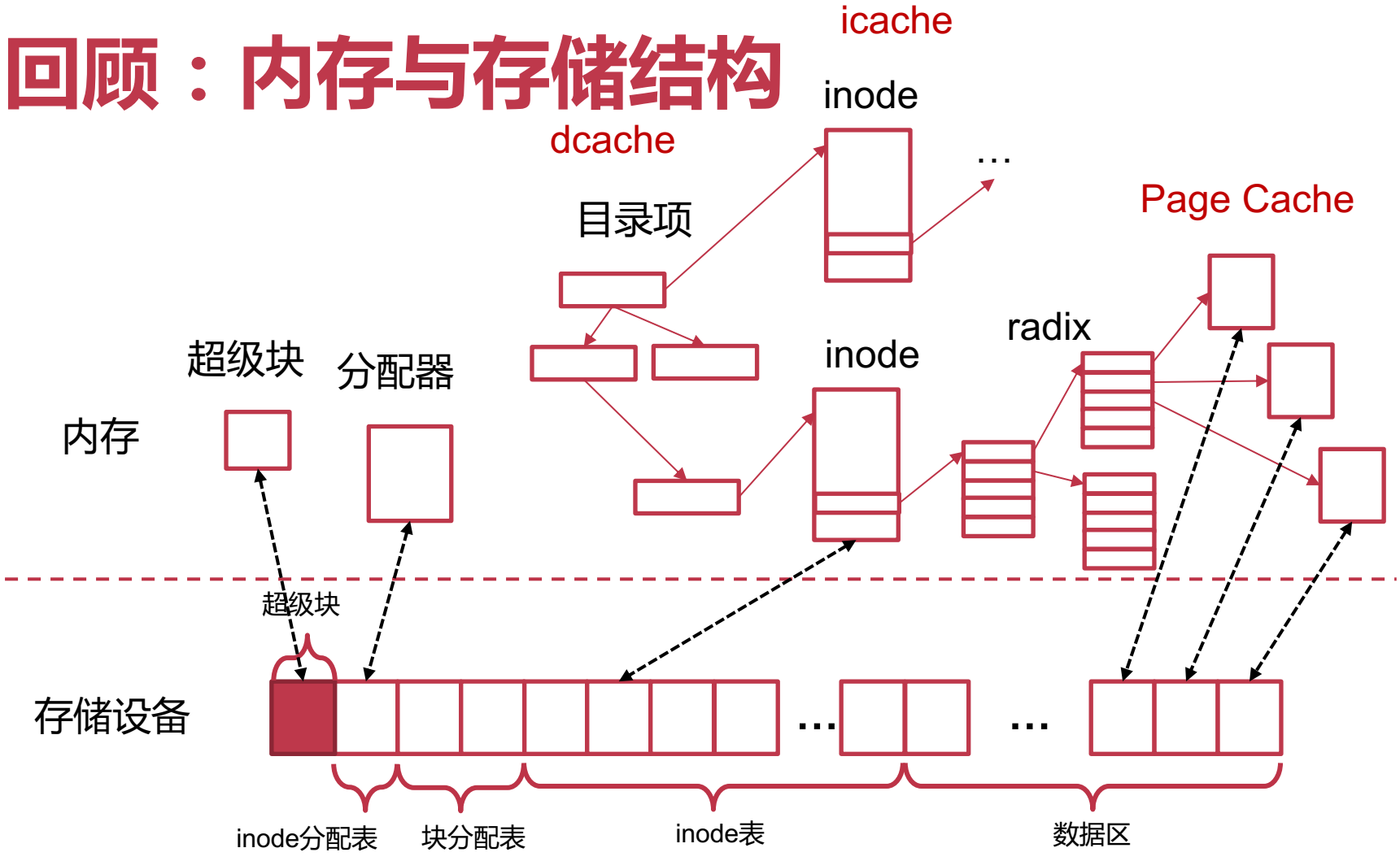
文件的创建

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！

回顾：内存与存储结构


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

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

思考一下：
考虑存在缓存，共存在多少种的崩溃情况？

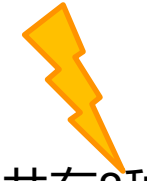
两种常见情况：

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

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

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



崩溃随时可能发生！

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

共有8种情况：
{}, // 没有操作被持久化
{1}
{2}
{3}
{1, 2}（与{2,1}相同）
{1, 3}
{2, 3}
{1, 2, 3}

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！

思考时间🤔

- 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？
- 数据中心一般会配有UPS（不间断电源），是否还需要保证文件系统崩溃一致性？

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
   ```

一些（简化的）假设

- 磁盘是fail-stop, 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
- 磁盘可能不会执行最近的几次操作
- 保障：磁盘不会写飞(wild writes)

为什么保证崩溃一致性这么困难呢？

崩溃 = halt/restart CPU

let disk finish current sector write, assume no h/w damage, no wild write to disk

目标: 自动恢复

Can fs always make sense of on-disk metadata after restart?

Given that the crash could have occurred at any point?

例子：

Crash during mkdir, leave directory without . and ..

Crash during free blocks

方法：在线与离线恢复

离线恢复

文件系统检查工具, 例如： windows中的chkdsk，Linux中的fsck

例如, ext3

在线恢复

运行过程中，检查一些重要的不一致性

例子, ext4 (同时也使用fsck, 但是非常简单)

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

- **原子更新技术**
  - 日志
  - 写时复制

- **Soft updates**

原子更新技术：Journaling

日志

日志

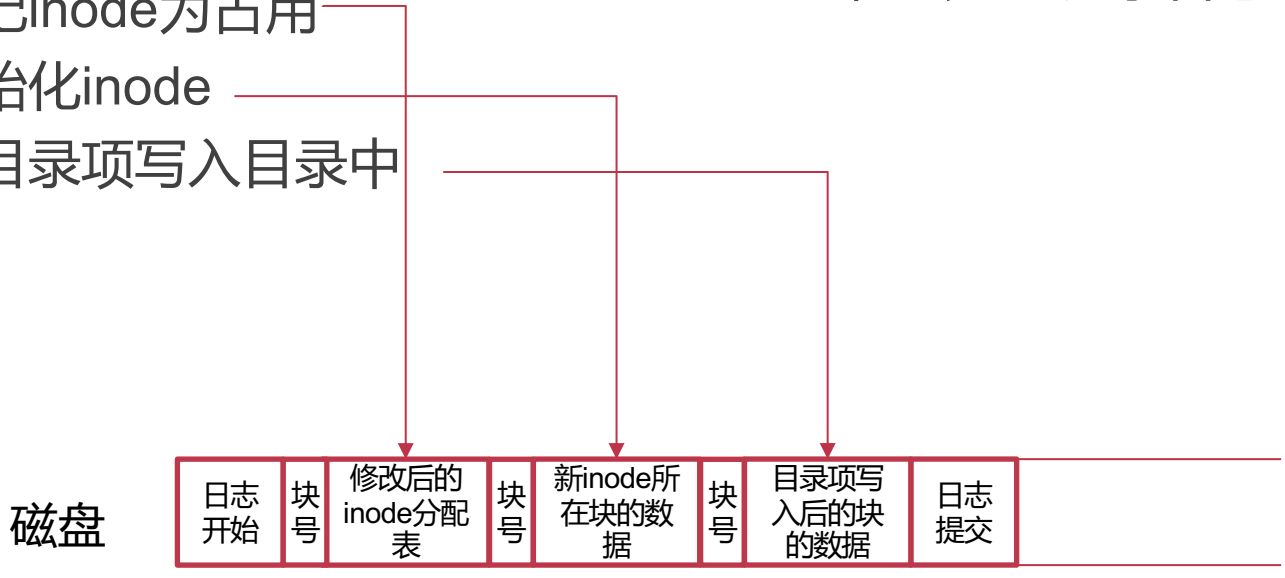
- 在进行修改之前，先将修改记录到日志中
- 所有要进行的修改都记录完毕后，提交日志
- 此后再进行修改
- 修改之后，删除日志

日志

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

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

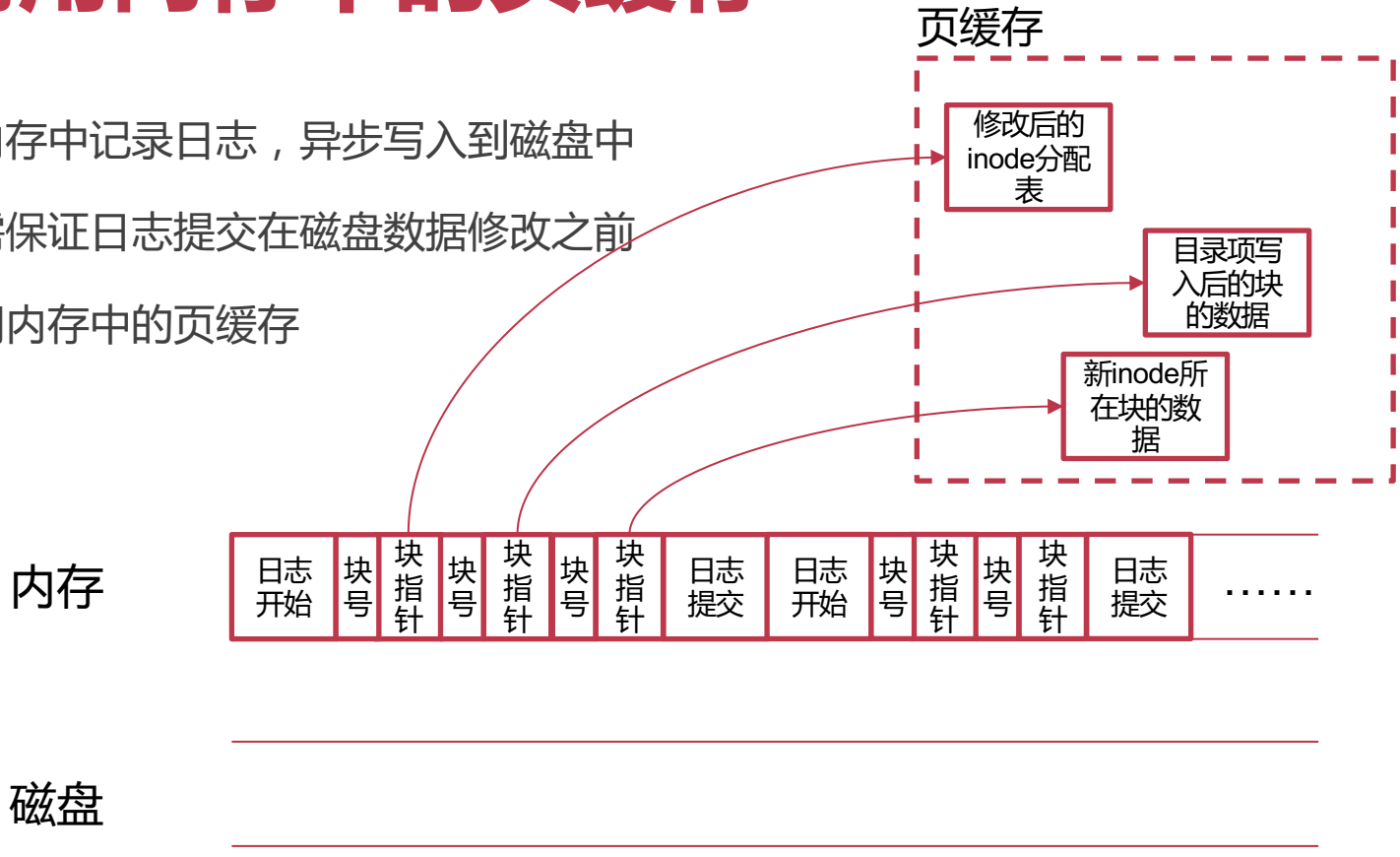
……

利用内存中的页缓存

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

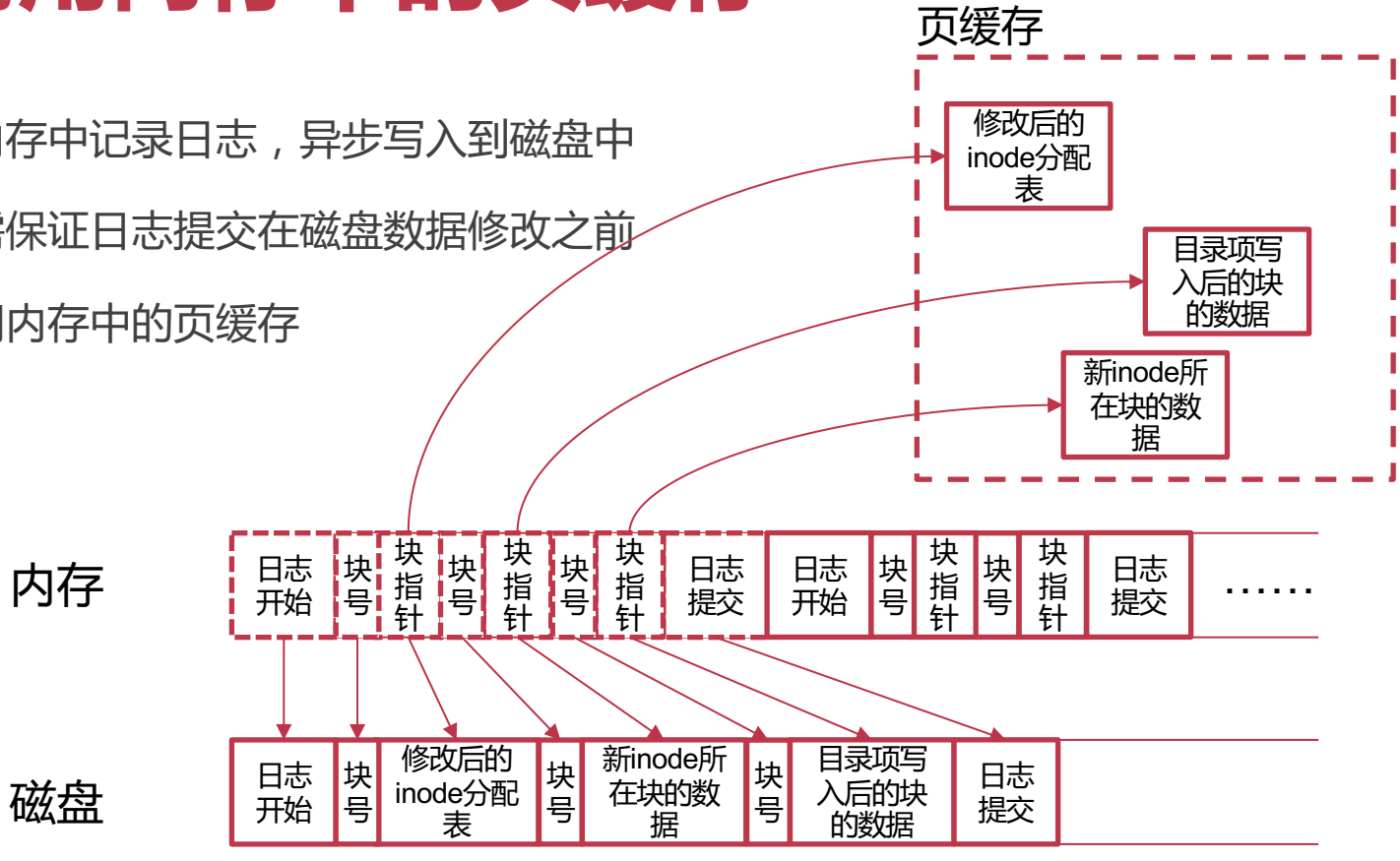


利用内存中的页缓存

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

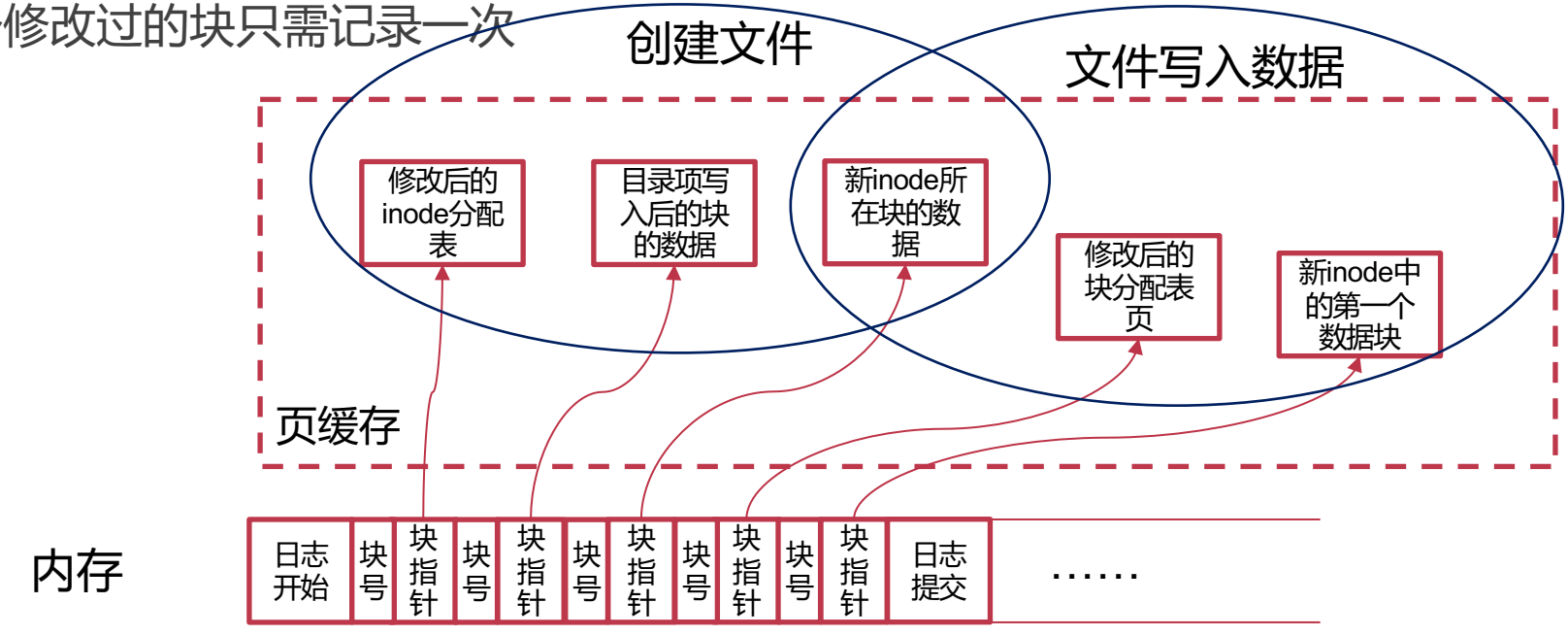
利用内存中的页缓存



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

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

事务内容已经写入存储设备，正在写入提交块

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)
```

通知jbd2即将修改bh中的数据

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)


// 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)

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)
```

通知jbd2，修改bh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)
}
```

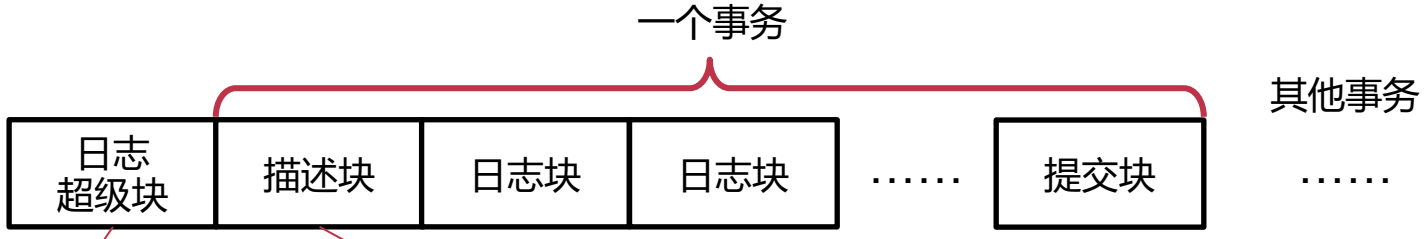
文件系统卸载时：

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

系统调用处理：

```
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;
```

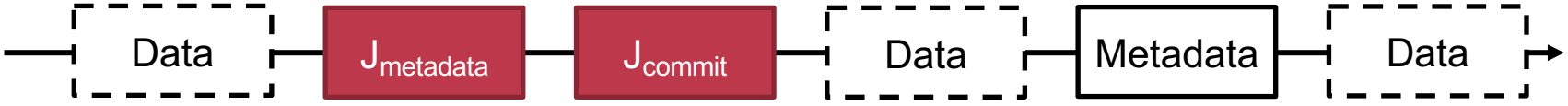
tag数组，每个tag对应后面日志块的信息

```
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;
```

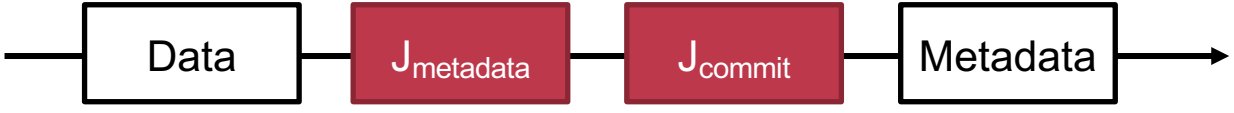
恢复时，根据描述块中记录的tag信息，将后面日志块中的数据写入到对应的磁盘块中

Ext4用JBD2实现的三种日志模式

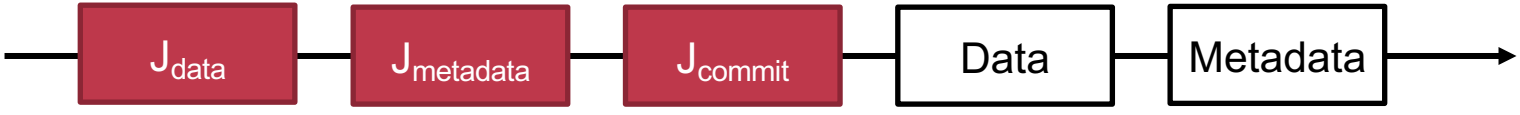
Writeback Mode：日志只记录元数据



Ordered Mode：日志只记录元数据+数据块在元数据日志前写入磁盘



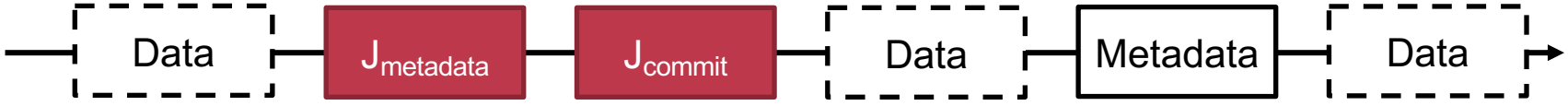
Journal Mode (Full Mode)：元数据和数据均使用日志记录



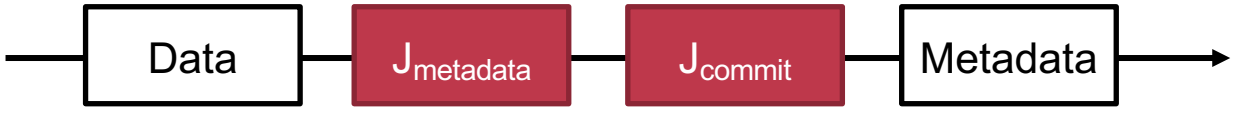
思考一下：三种模式各自有何问题和优势？

Ext4用JBD2实现的三种日志模式

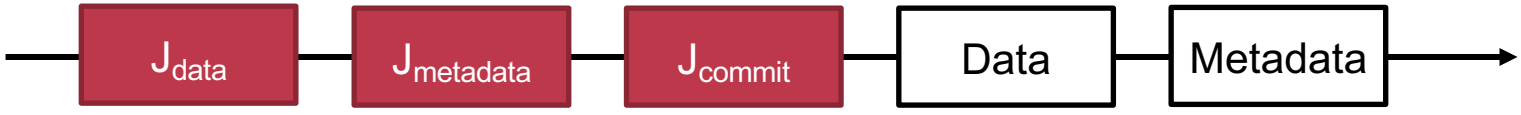
Writeback Mode：日志只记录元数据　最快，但是一致性最差！



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！



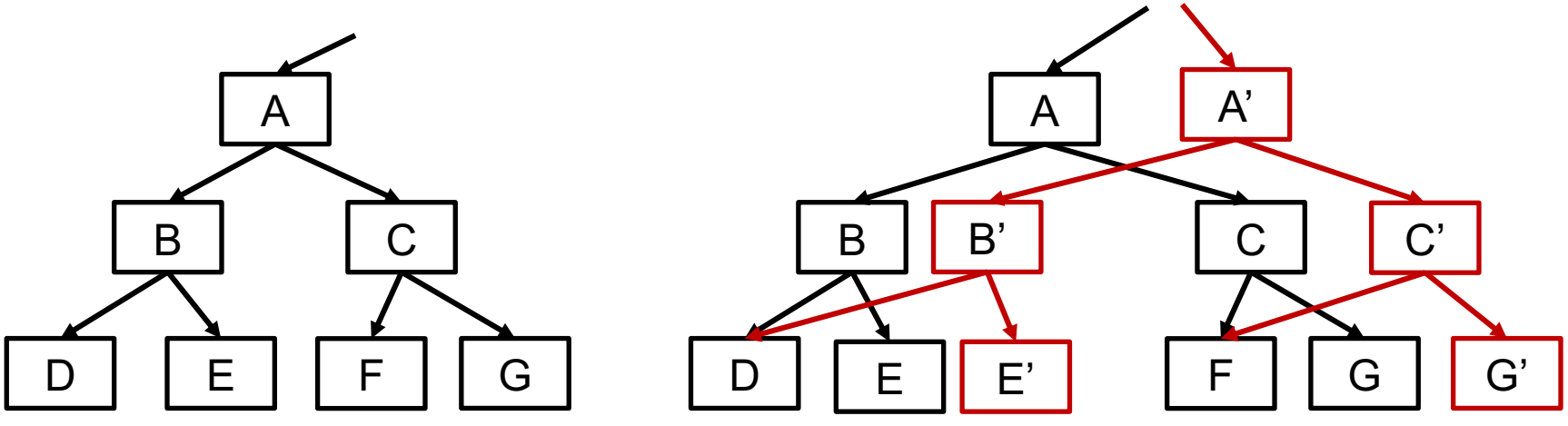
思考一下：三种模式各自有何问题和优势？

原子更新技术：Copy-on-Write

写时复制

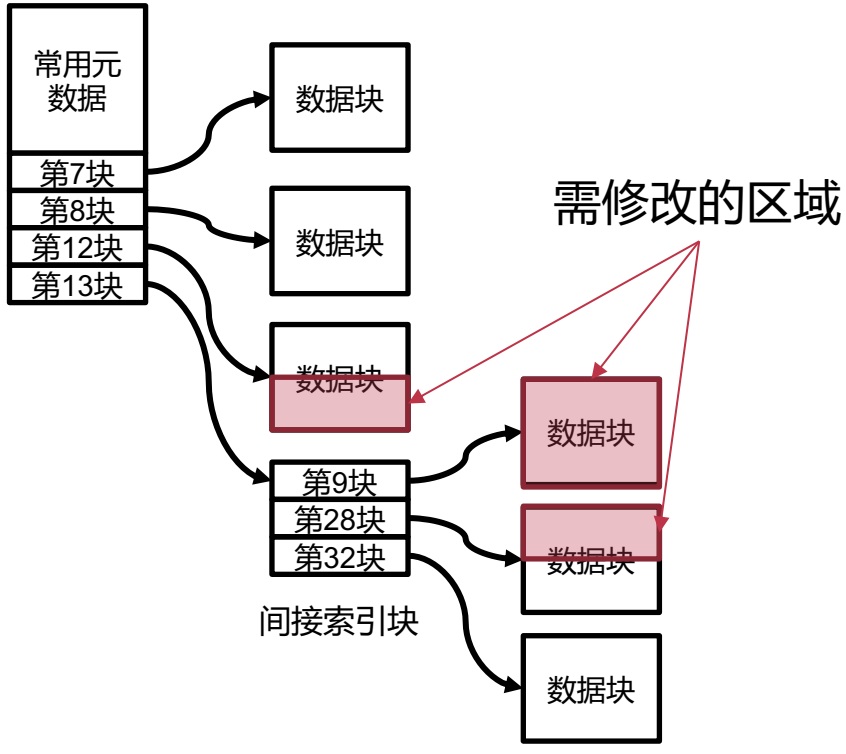
写时复制（Copy-on-Write）

- 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



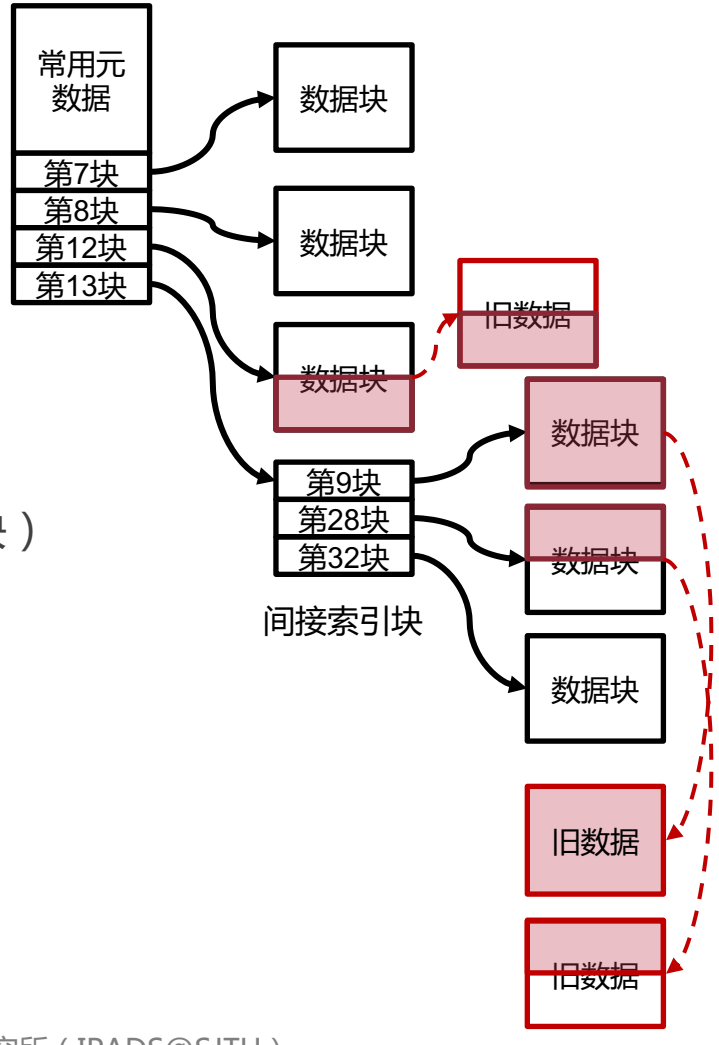
文件中的写时复制

- 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



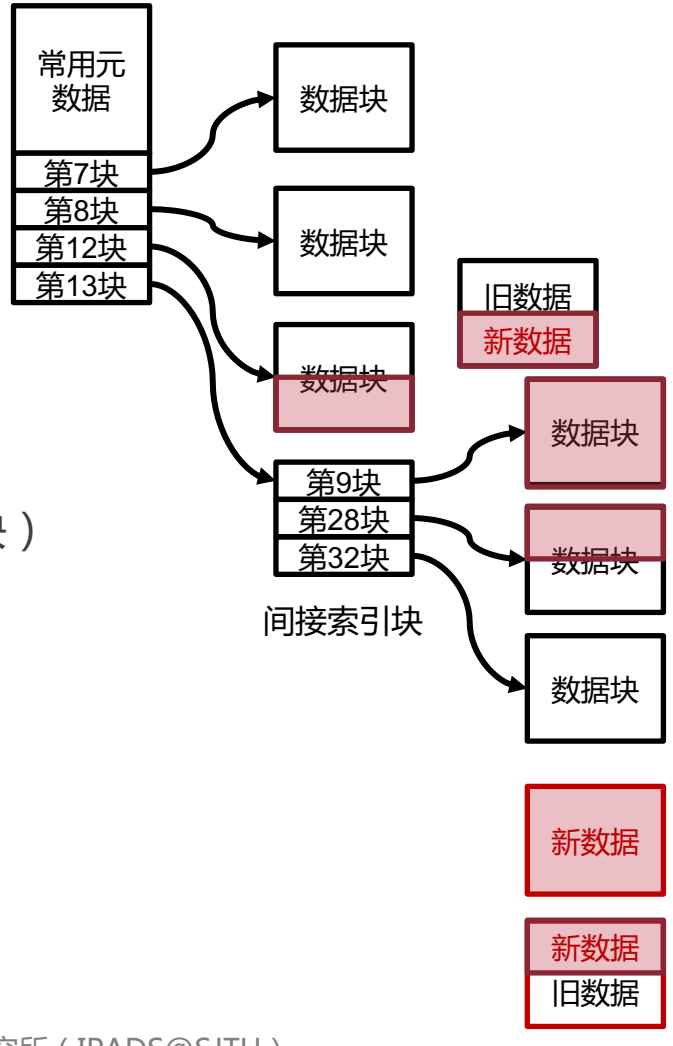
文件中的写时复制

- 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）



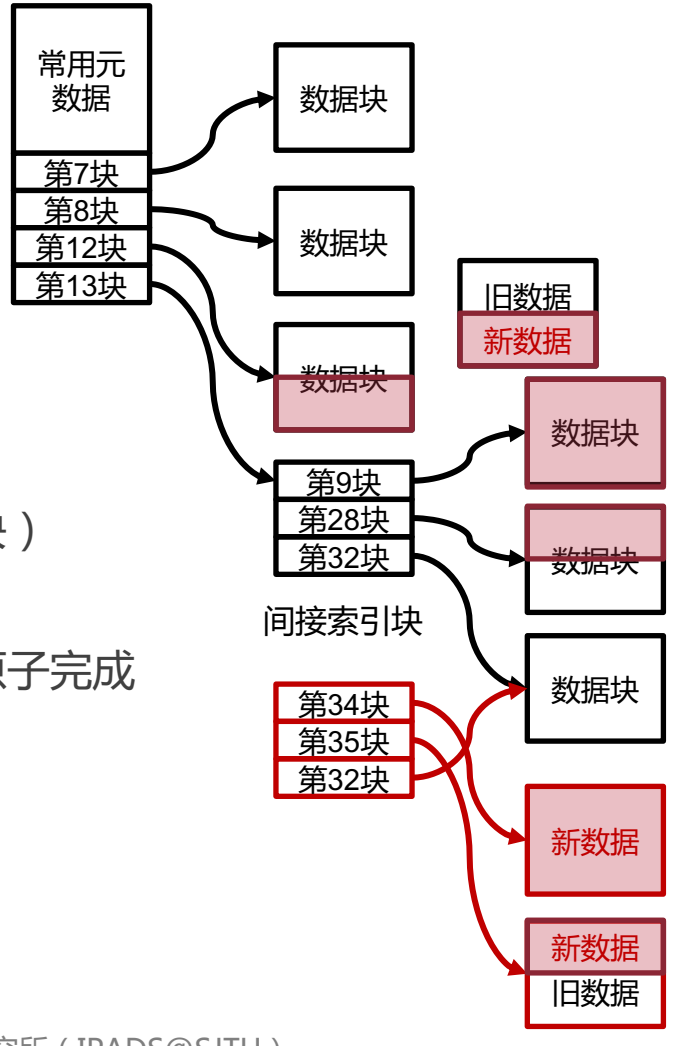
文件中的写时复制

- 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



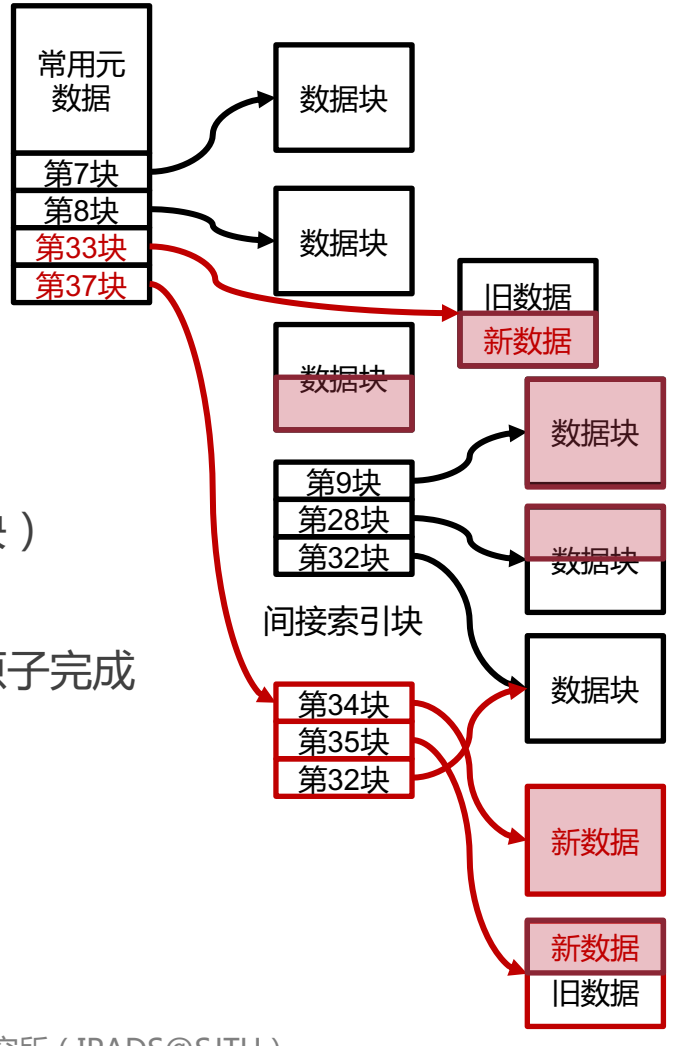
文件中的写时复制

- 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



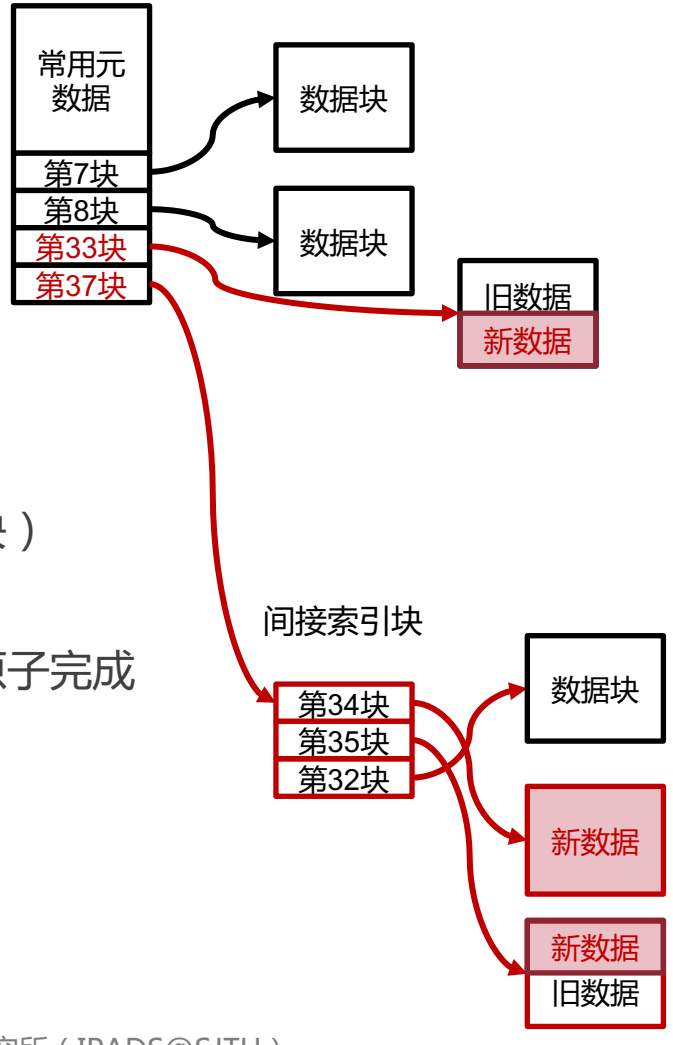
文件中的写时复制

- 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

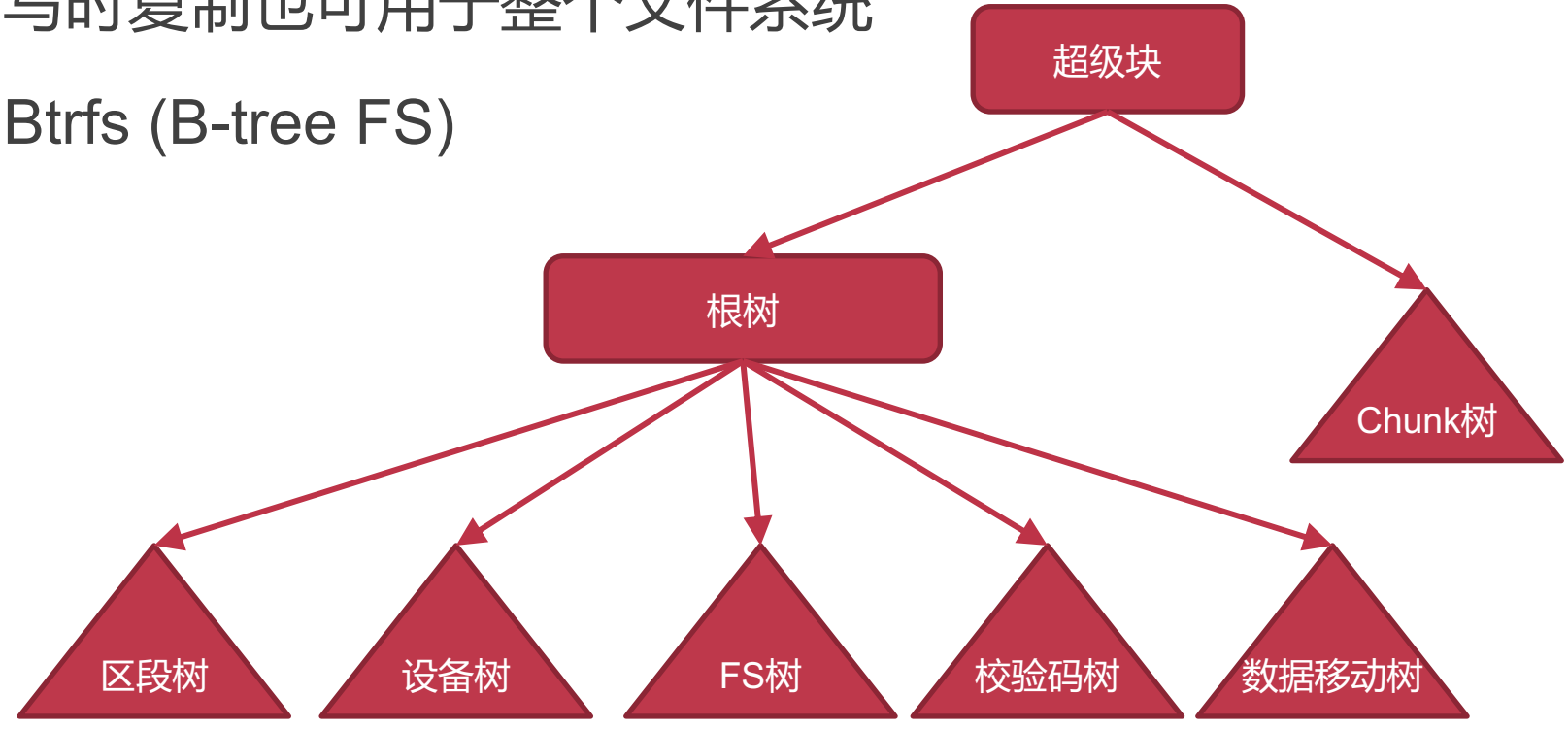


思考时间🤔

- 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)



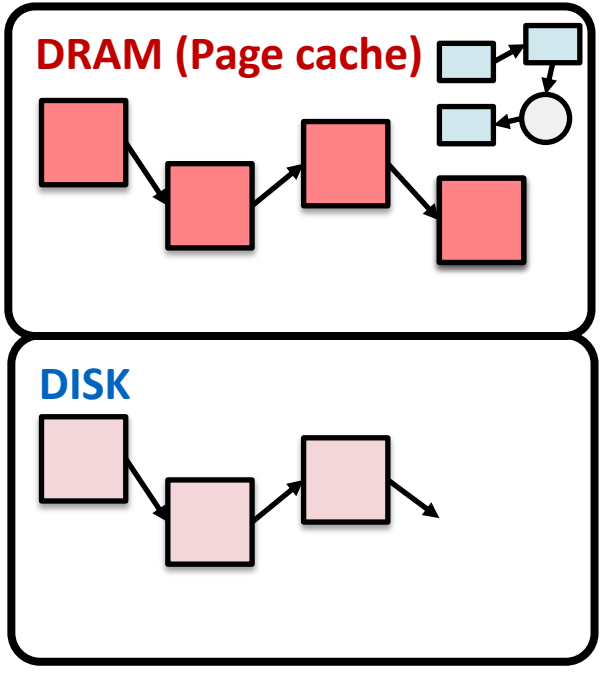
SOFT UPDATES

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

Soft Updates的总体思想

- **最新的元数据在内存中**
  - Updated in DRAM with dependency tracked
  - ✓ DRAM performance
  - ✓ No synchronous disk writes
- **磁盘中的元数据总是一致的**
  - Persisted to disks with dependency enforced
  - ✓ Always consistent
  - ✓ Immediately usable after crash



Traditional 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，改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

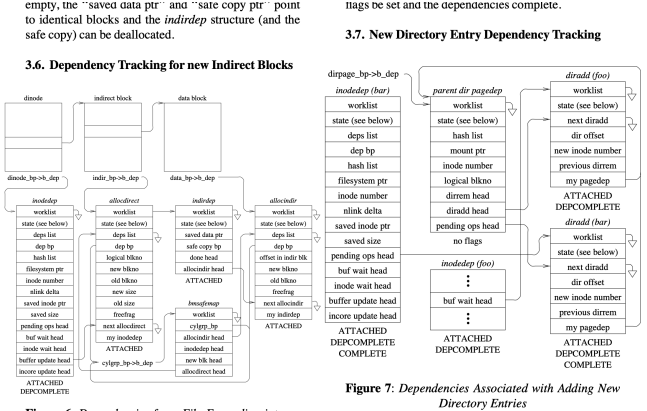
- **对于每个文件系统请求，将其拆解成对多个结构的操作**
  - 记录对每个结构的修改内容（旧值、新值）
  - 记录这个修改依赖于那些修改（应在哪些修改之后持久化）
  - 如创建文件：
    1. 标记inode为占用（对bitmap的修改）
    2. 初始化inode（对inode的修改，依赖于1）
    3. 将目录项写入目录中（对目录文件的内容修改，依赖于1和2）

实现十分复杂

- **Delayed disk writes**
  - Auxiliary structures for each operation
  - More complex dependencies
- **Cyclic dependencies**
  - Rolling back/forward
- **实现**
  - (FreeBSD fs modified on FFS)

“soft updates are, simply put, too hard to understand, implement, and maintain to be part of the mainstream of file system development”

Valerie Aurora, Linux Kernel Developer



empty, the ¨saved data ptr¨ and ¨safe copy ptr¨ point to identical blocks and the *indirdep* structure (and the safe copy) can be deallocated.

3.6. Dependency Tracking for new Indirect Blocks

Figure 6: *Dependencies for a File Expanding into an Indirect Block*

flags be set and the dependencies complete.

3.7. New Directory Entry Dependency Tracking

Figure 7: *Dependencies Associated with Adding New Directory Entries*

Figure 7 shows the dependency structures for a direc-

John K. Ousterhout, Mendel Rosenblum. (1991), *The Design and Implementation of a Log-Structured File System*

日志文件系统

日志文件系统（Log-structured FS）

The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout

Electrical Engineering and Computer Sciences, Computer Science Division
University of California
Berkeley, CA 94720
mendel@sprite.berkeley.edu, ouster@sprite.berkeley.edu

Abstract

This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efficiency of a simple cleaning policy based on cost and benefit. We have implemented a prototype log-

magnitude more efficiently than current file systems.

Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests[1]. As a result, disk traffic will become dominated by writes. A log-structured file system writes all new information to disk in a sequential structure called the *log*. This approach increases write performance dramatically by eliminating almost all seeks. The sequential nature of the log also permits much faster crash recovery: current Unix file systems typically must scan the entire disk to restore consistency after a crash, but a log-structured file system need only examine the most recent portion of the log.

1992 *ACM Transactions on Computer Systems*

John K. Ousterhout

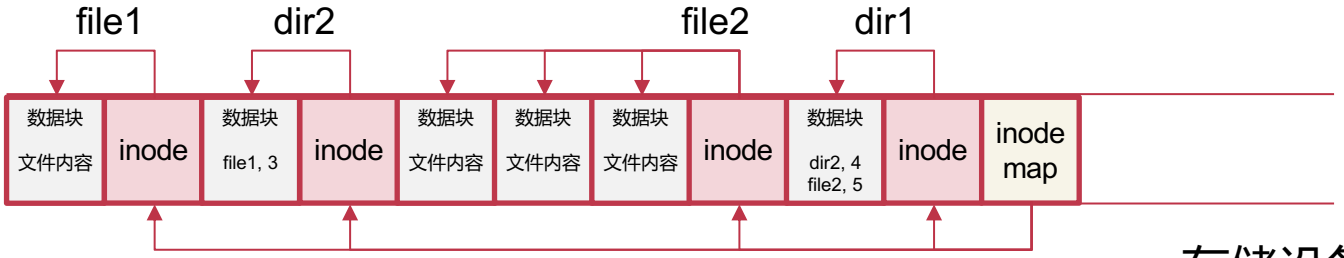
Sprite 负责人
TCL/TK 作者
Magic VLSI CAD 作者
Co-scheduling 提出者
Raft 一致性协议
目前是Stanford大学教授

Mendel Rosenblum

VMware 联合创始人
目前是Stanford大学教授

日志文件系统（Log-structured FS）

- 假设：文件被缓存在内存中，文件读请求可以被很好的处理
  - 于是，文件写成为瓶颈
- 块存储设备的顺序写比随机写速度很块
  - 磁盘寻道时间
- 将文件系统的修改以日志的方式**顺序写入**存储设备

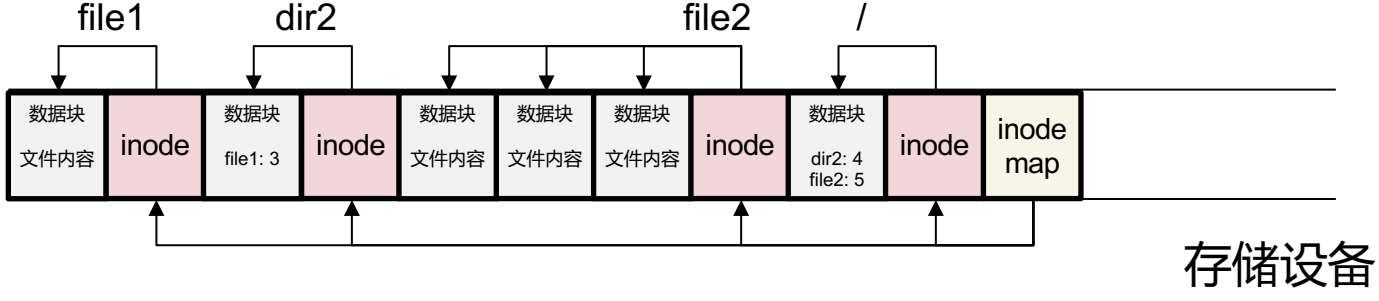


Sprite LFS的数据结构

- 固定位置的结构
  - 超级块、检查点（checkpoint）区域
- 以Log形式保存的结构
  - inode、间接块（索引块）、数据块
  - inode map：记录每个inode的当前位置
  - 段概要：记录段中有效块
  - 段使用表：段中有效字节数、段的最后修改时间
  - 目录修改日志

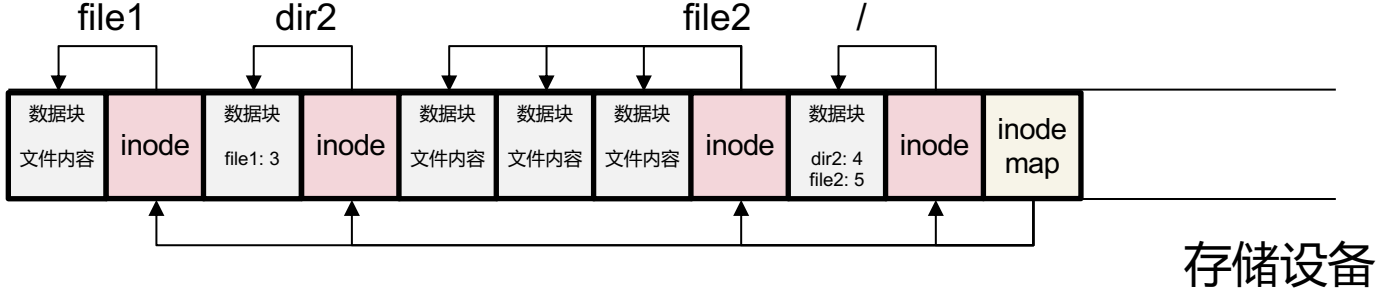
举例

- 一个日志文件系统
  - 有4个inode，位置记录在inode map中
  - 对应4个文件分别为：/，/dir2，/file2，/dir2/file1



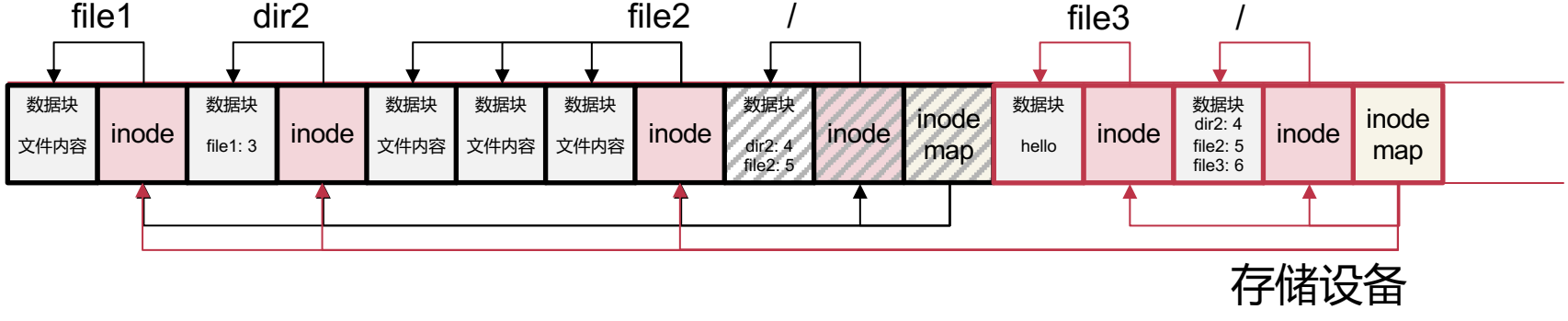
创建文件举例

- echo hello > /file3
  - 创建文件
  - 修改文件数据



创建文件举例

- echo hello > /file3
  - 创建文件
  - 修改文件数据

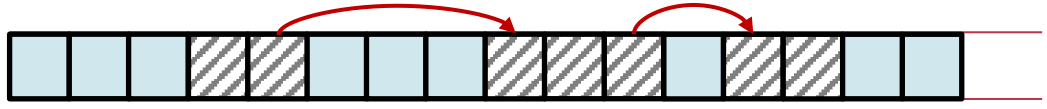


空间回收利用

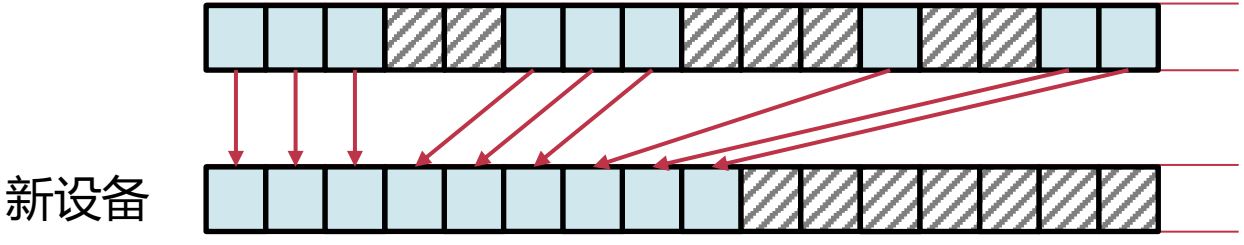
- 存储设备最开始是一个连续的空闲空间
- 随文件系统的使用，日志写入位置会接近存储设备末端
- 需重新利用前面的设备空间
  - 文件系统数据被修改后，此前的块被无效化
  - 如何组织前面无效空间，以重新利用它们？

空间回收管理方法

- 串联：将所有空闲空间用链表串起来
  - 磁盘空间会越来越碎，影响到LFS的大块顺序写的性能

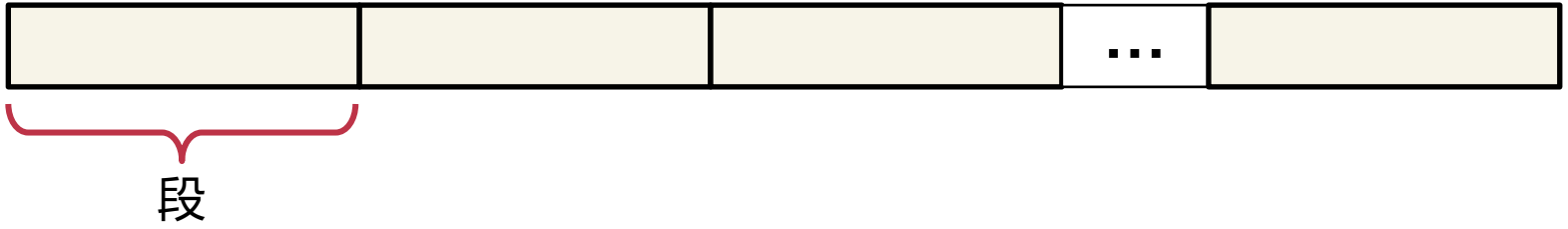


- 拷贝：将所有的有效空间整理拷贝到新的存储设备
  - 数据需要拷贝



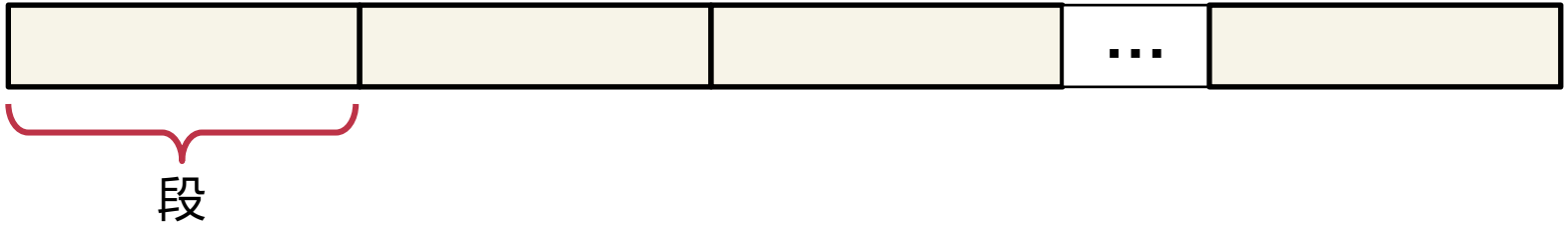
空间回收管理方法：段（Segment）

- 串联
  - 磁盘空间会越来越碎，影响到LFS的大块顺序写的性能
- 拷贝
  - 数据需要拷贝
- 串联和拷贝两种方法的结合：段



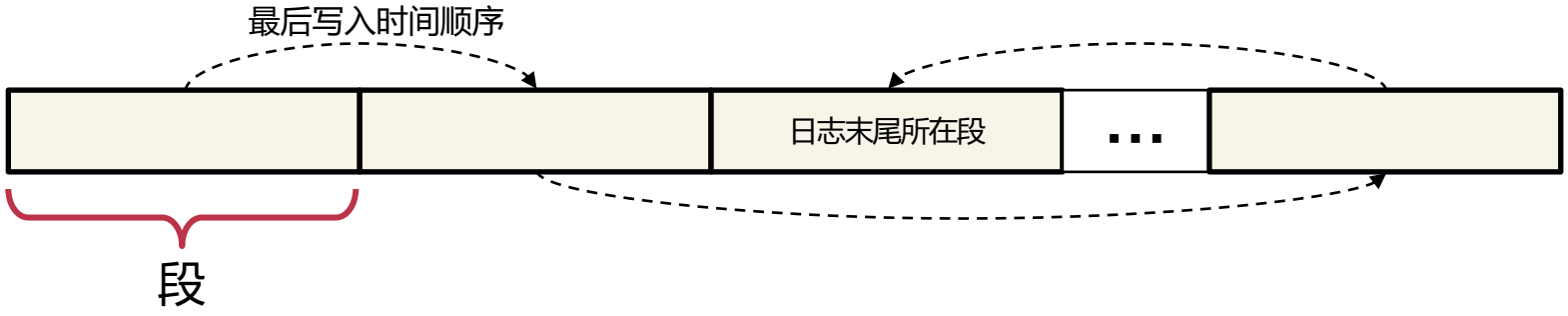
空间回收管理方法：段（Segment）

- 一个设备被拆分为定长的区域，称为段
  - 段大小需要足以发挥出顺序写的优势，512KB、1MB等
- 每段内只能顺序写入
  - 只有当段内全都是无效数据之后，才能被重新使用
- 干净段用链表维护（对应串联方法）



段使用表

- 段使用表
  - 记录每个段中有效字节数
    - 归零时变为干净段
  - 记录了每个段最近写入时间
    - 将非干净段按时间顺序连在一起，形成逻辑上的连续空间

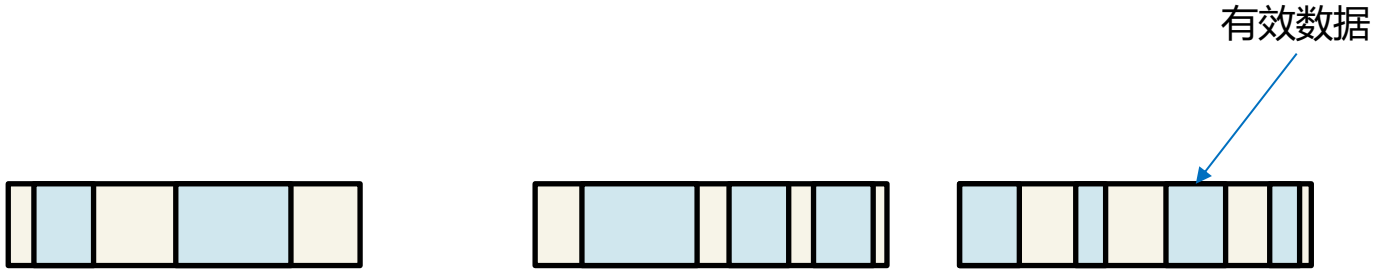


段清理

1. 将一些段读入内存中准备清理

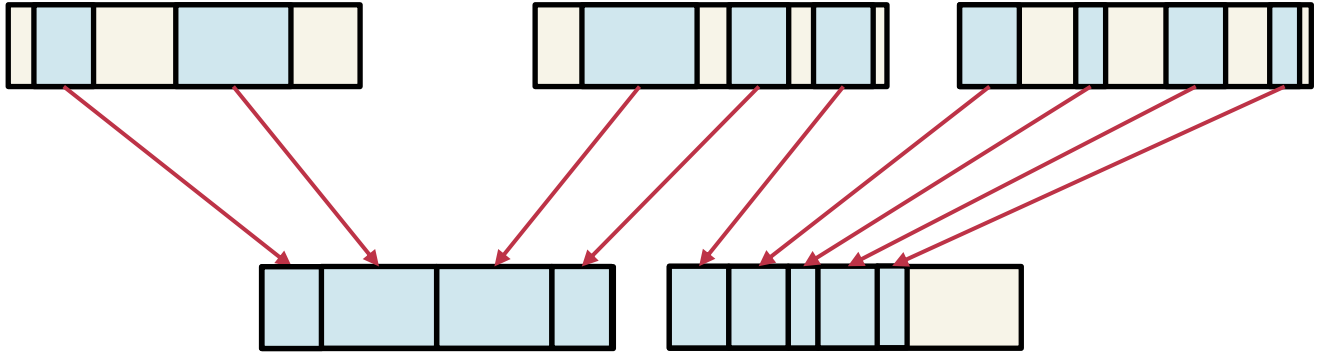
段清理

1. 将一些段读入内存中准备清理
2. 识别出有效数据



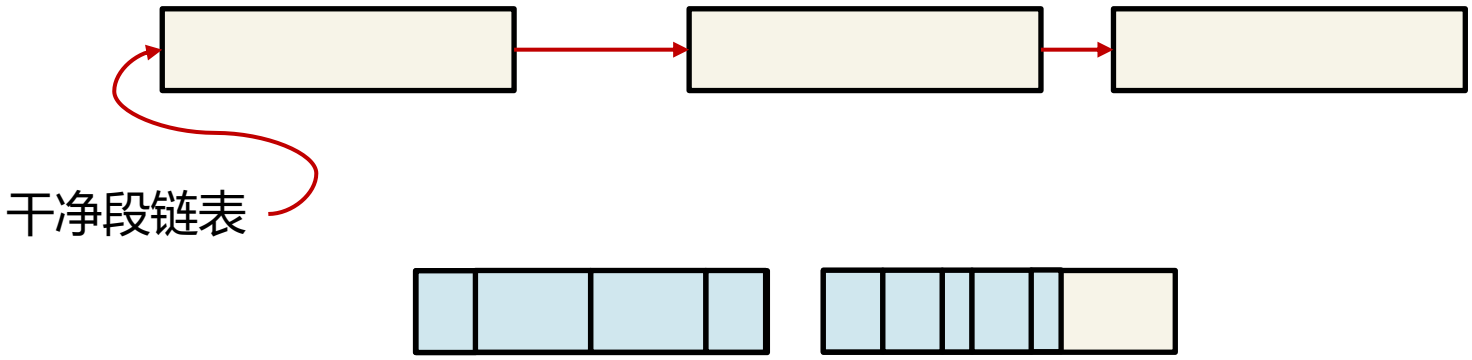
段清理

1. 将一些段读入内存中准备清理
2. 识别出有效数据
3. 将有效数据整理后写入到干净段中（对应拷贝方法）



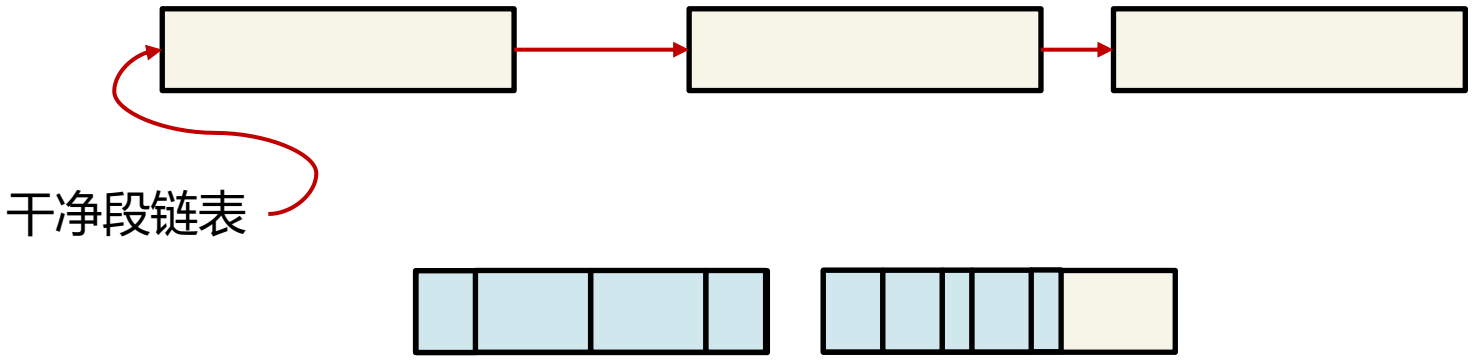
段清理

1. 将一些段读入内存中准备清理
2. 识别出有效数据
3. 将有效数据整理后写入到干净段中（对应拷贝方法）
4. 标记被清理的段为干净



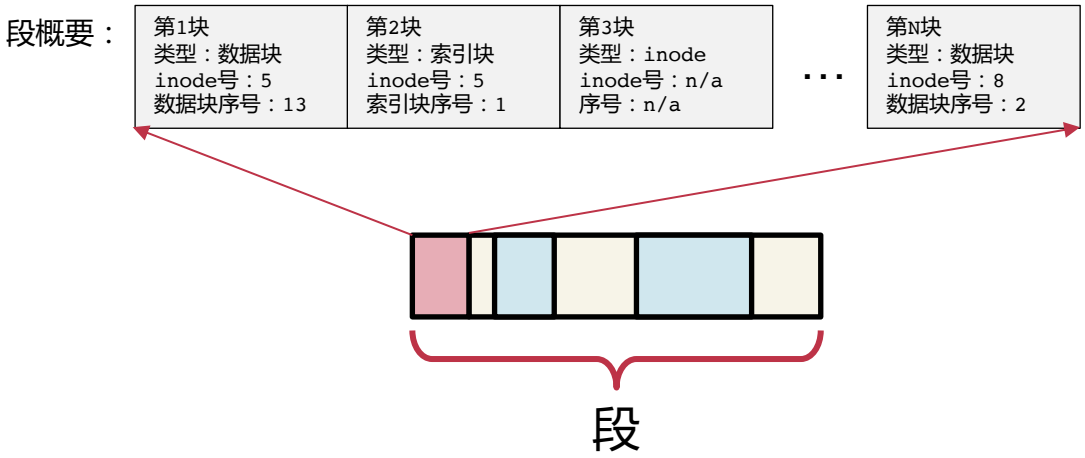
段清理

1. 将一些段读入内存中准备清理 ✓
2. 识别出有效数据 ?
3. 将有效数据整理后写入到干净段中（对应拷贝方法） ✓
4. 标记被清理的段为干净 ✓



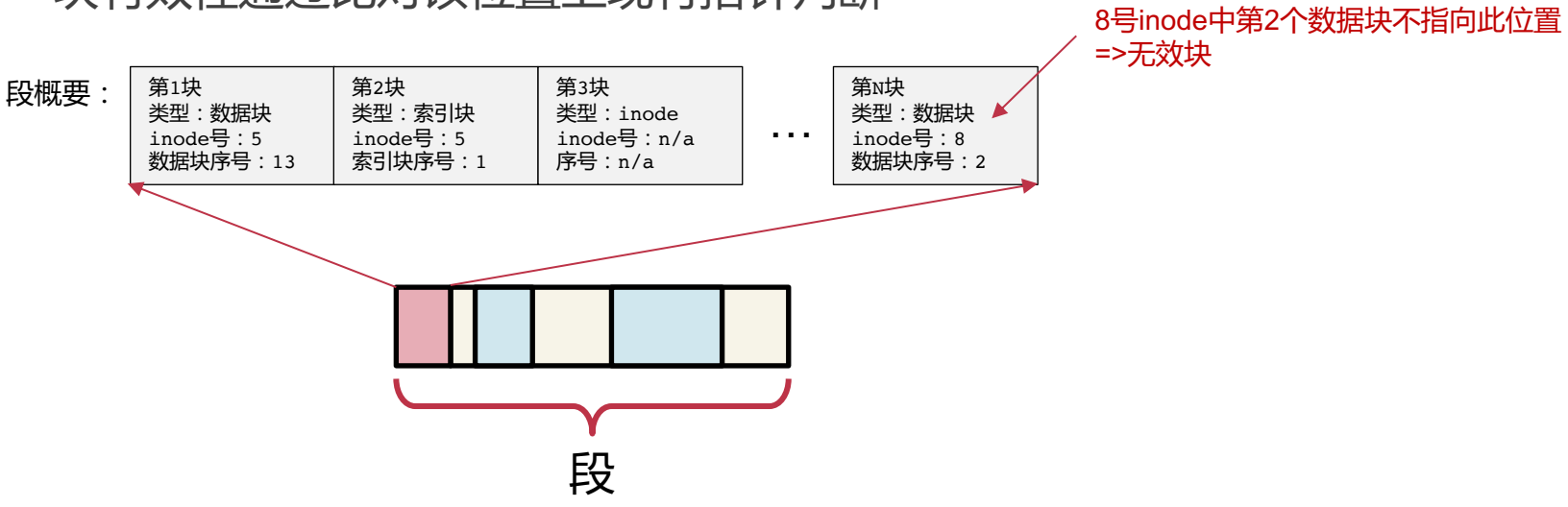
识别有效数据

- 每个段中保存有**段概要**
  - 每个块被哪个文件的哪个位置所使用
    - 如：数据块可使用inode号和第几个数据块来表示位置



识别有效数据

- 每个段中保存有**段概要**
  - 每个块被哪个文件的哪个位置所使用
    - 如：数据块可使用inode号和第几个数据块来表示位置
  - 块有效性通过比对该位置上现有指针判断



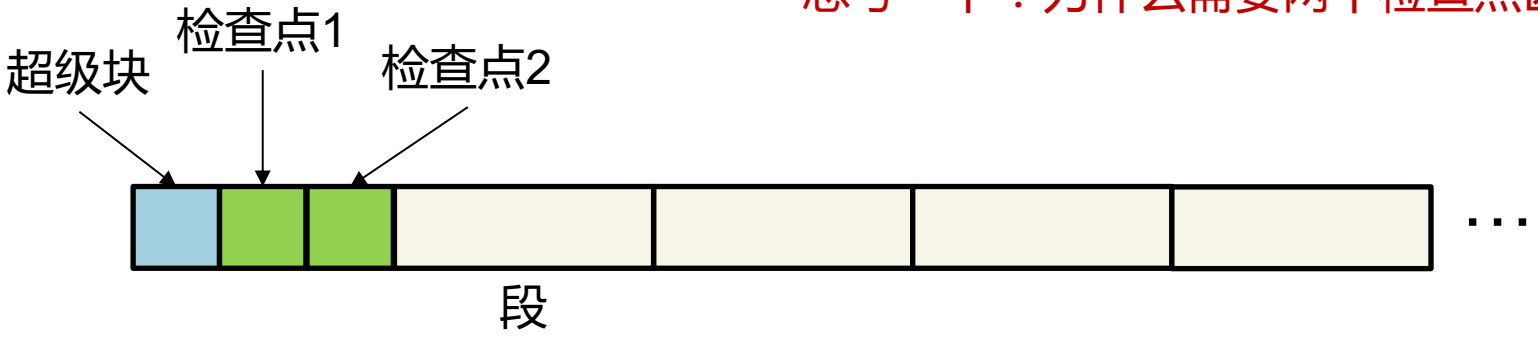
挂载和恢复

- 扫描所有日志，重建出整个文件系统的内存结构
  - 大量无效数据也被扫描
- 定期写入检查点（checkpoint）
  - 写入前的有效数据，可以通过检查点找到
  - 只需扫描检查点之后写入的日志
  - 减少挂载/恢复时间

检查点

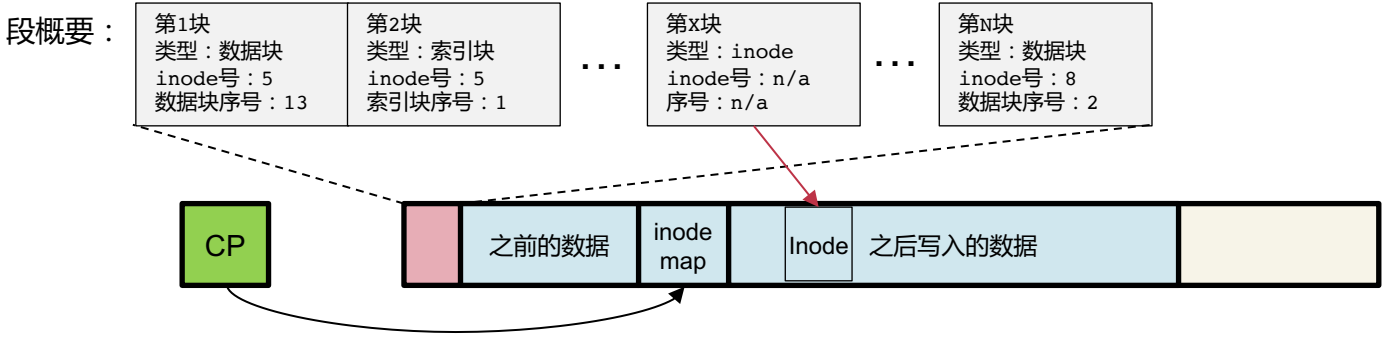
- 检查点内容
  - inode map的位置（可找到所有文件的内容）
  - 段使用表
  - 当前时间
  - 最后写入的段的指针

思考一下：为什么需要两个检查点区域？



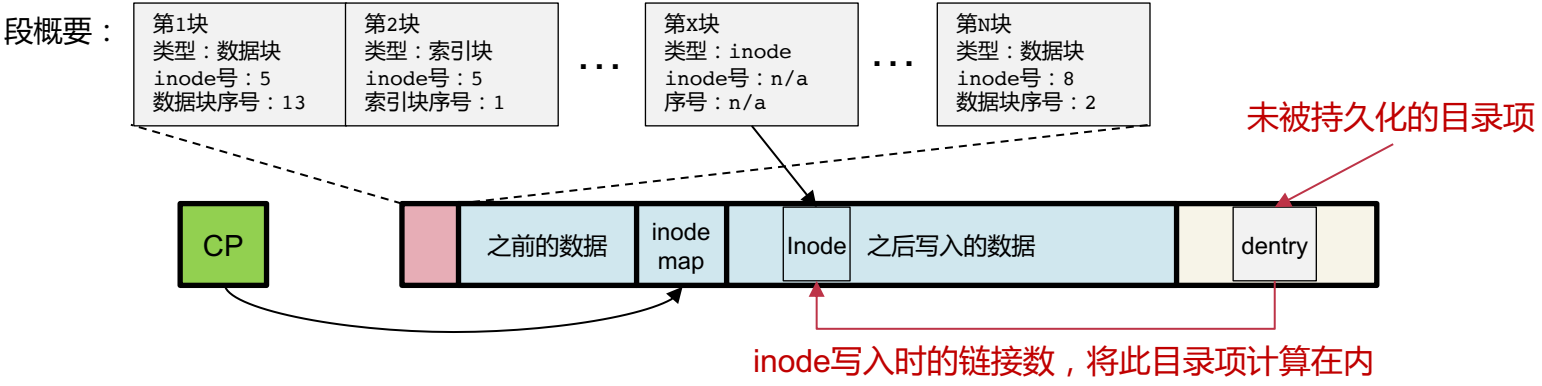
恢复：前滚（roll-forward）

- 尽量恢复检查点后写入的数据
- 通过段概要里面的新inode，恢复新的inode
  - 其inode中的数据块会被自动恢复
- 未被inode"认领"的数据块会被删除



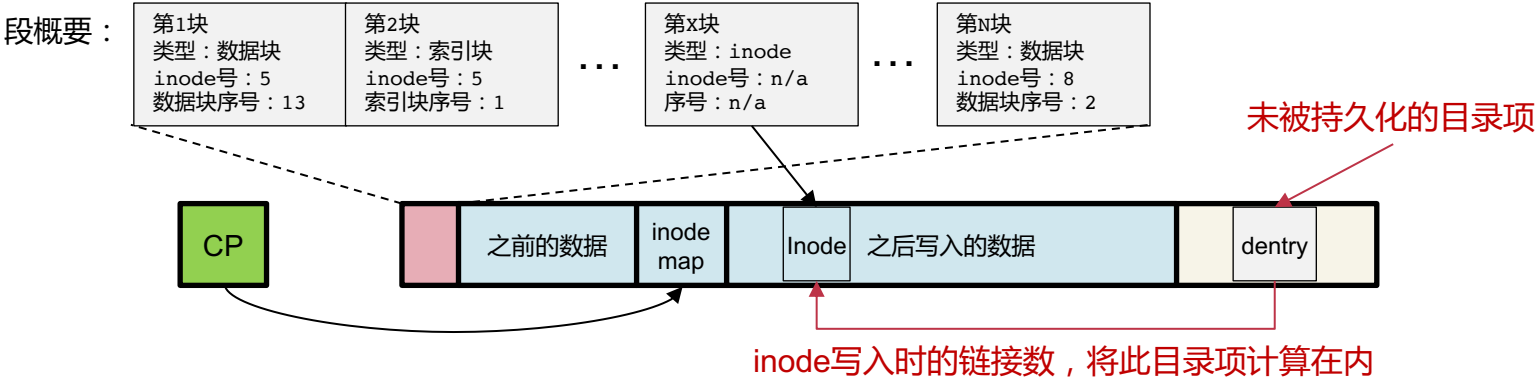
恢复：前滚（roll-forward）

- 段概要无法保证inode的链接数一致性
  - 如：inode被持久化，但是指向其的目录项未被持久化



恢复：前滚（roll-forward）

- 段概要无法保证inode的链接数一致性
  - 如：inode被持久化，但是指向其的目录项未被持久化
- 解决方案：目录修改日志



恢复：目录修改日志

- 目录修改日志
  - 记录了每个目录操作的信息
    - create、link、rename、unlink
  - 以及操作的具体信息
    - 目录项位置、内容、inode的链接数
- 目录修改日志的持久化在目录修改之前
  - 恢复时根据目录修改日志保证inode的链接数是一致的

关于LFS的讨论

- 假设：大多数读请求可以通过内存缓存处理
  - 但是真的去磁盘上读，会比较慢。因为文件会非常分散。
- LFS的segment和Flash的性质很像？

其他LFS的实现

- **光盘**
  - UDF
- **Flash/SSD**
  - JFFS, JFFS2
  - UBIFS
  - LogFS
  - YAFFS, YAFFS2
  - F2FS
- **非易失性内存**
  - NOVA
- **RAID**
  - WAFL (by NetApp)
- **NVM**
  - NOVA
- **Cloud**
  - ObjectiveFS via FUSE uses cloud object stores (e.g. Amazon S3, Google Cloud Storage and private cloud object store)

https://en.wikipedia.org/wiki/List_of_log-structured_file_systems