

新型文件系统

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

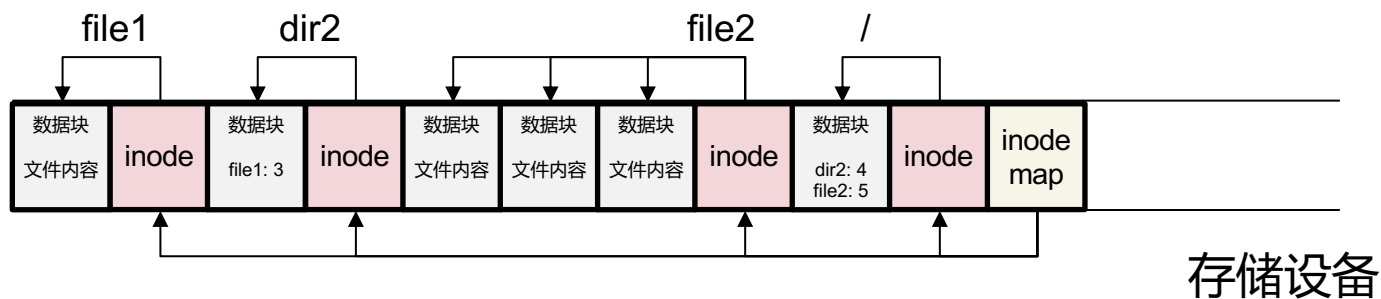
<https://ipads.se.sjtu.edu.cn>

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>

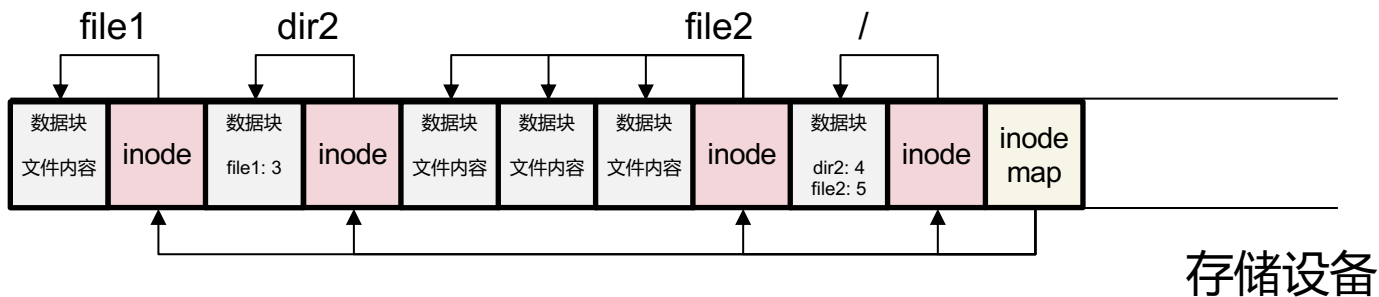
Review : LFS

- 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



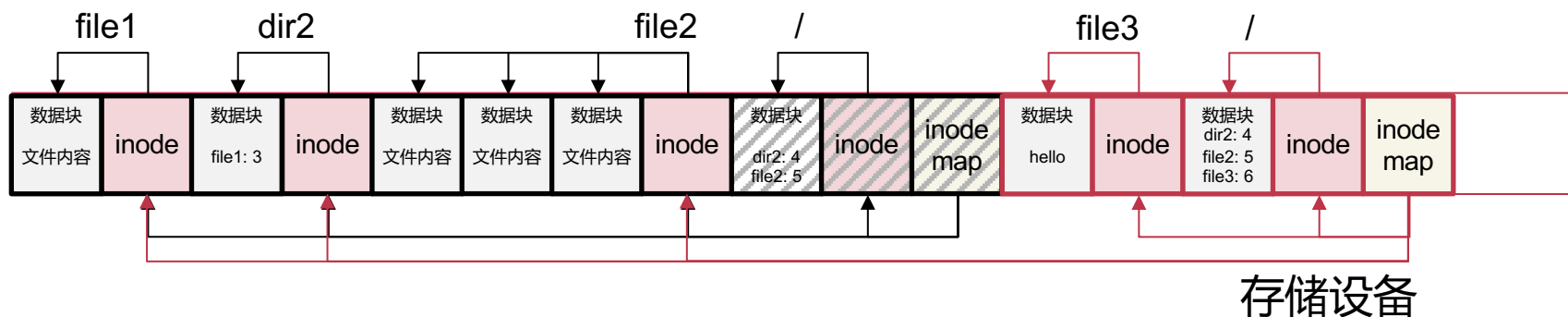
Review : 创建文件举例

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



Review : 创建文件举例

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



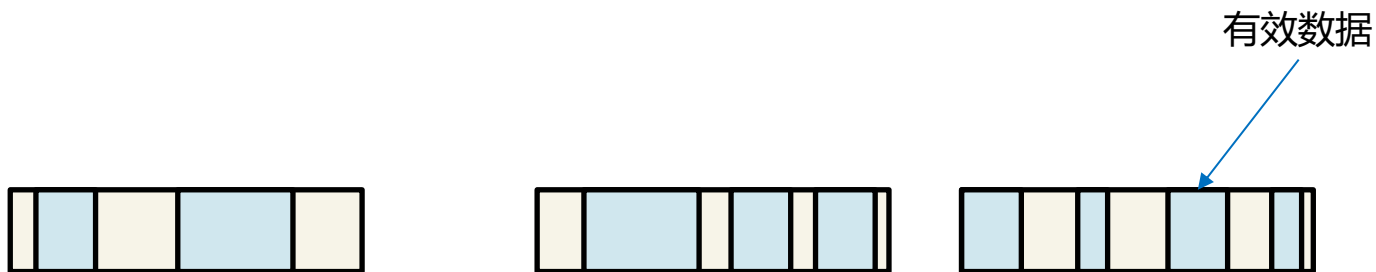
段清理

1. 将一些段读入内存中准备清理



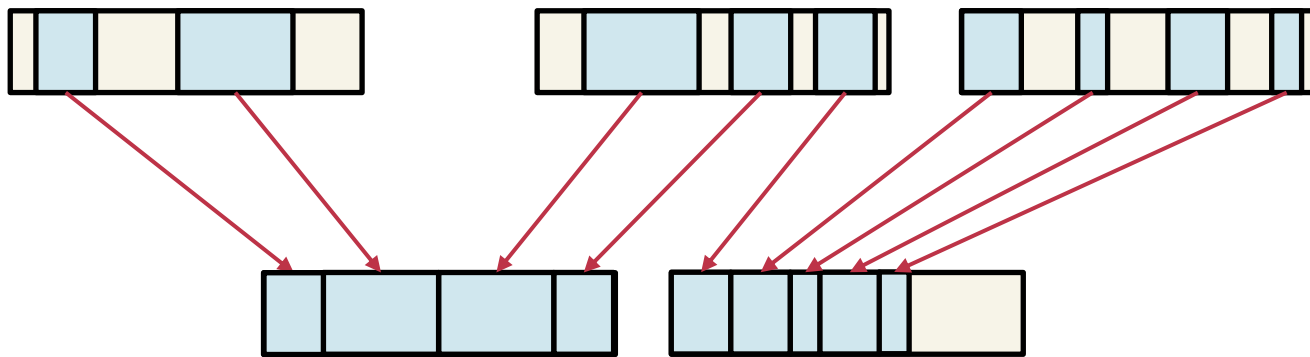
段清理

1. 将一些段读入内存中准备清理
2. 识别出有效数据



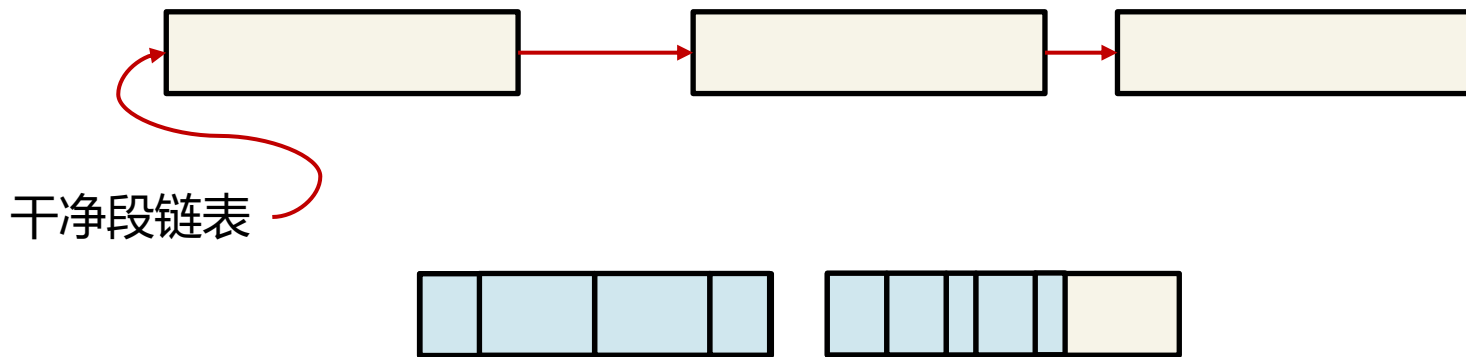
段清理

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）



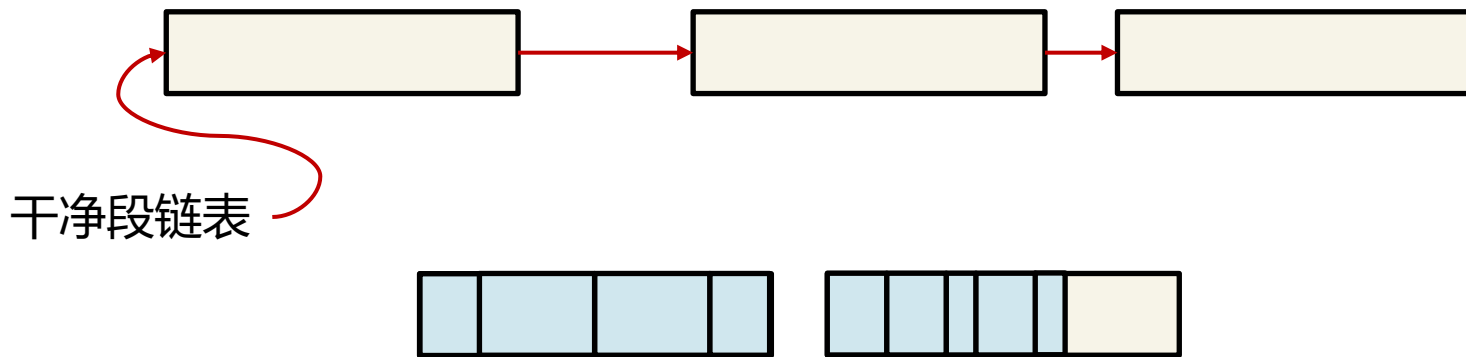
段清理

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



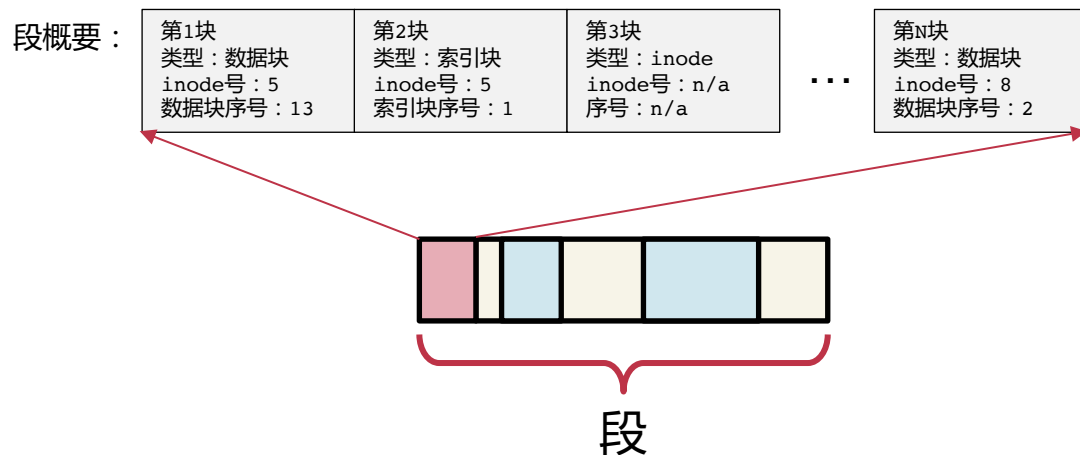
段清理

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 ✓



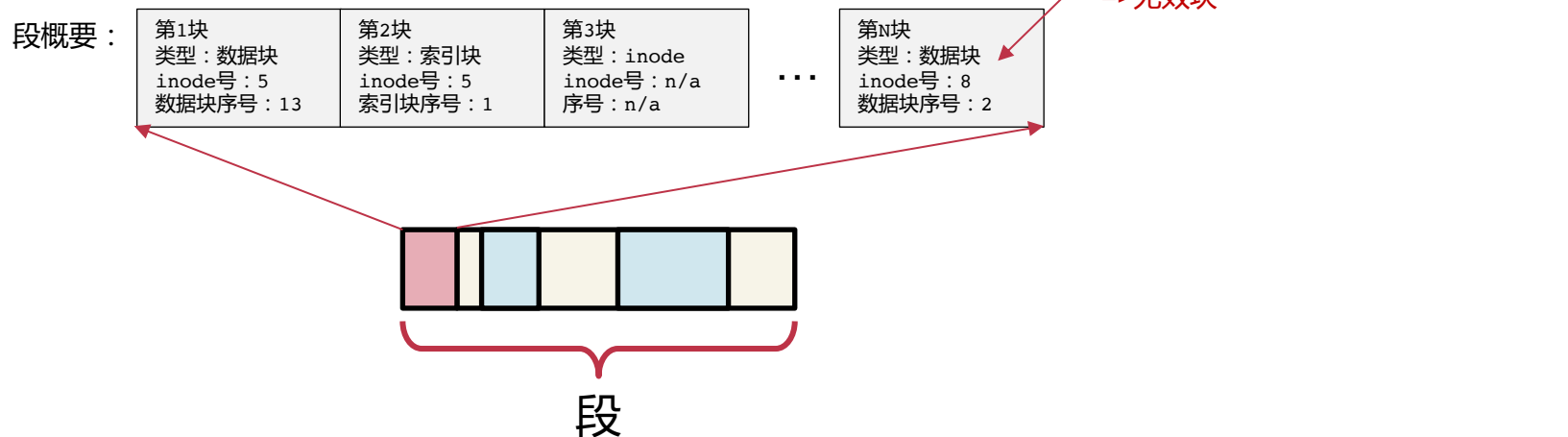
识别有效数据

- 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



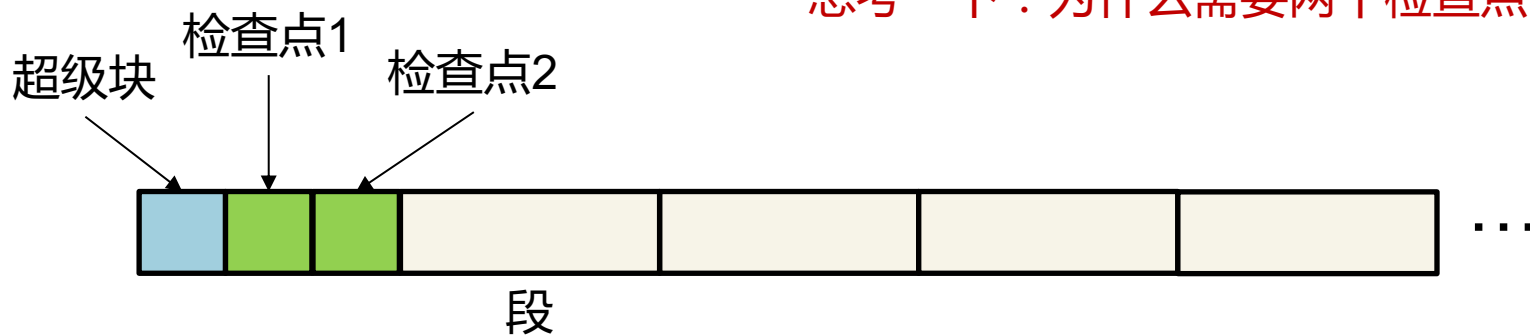
挂载和恢复

- 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

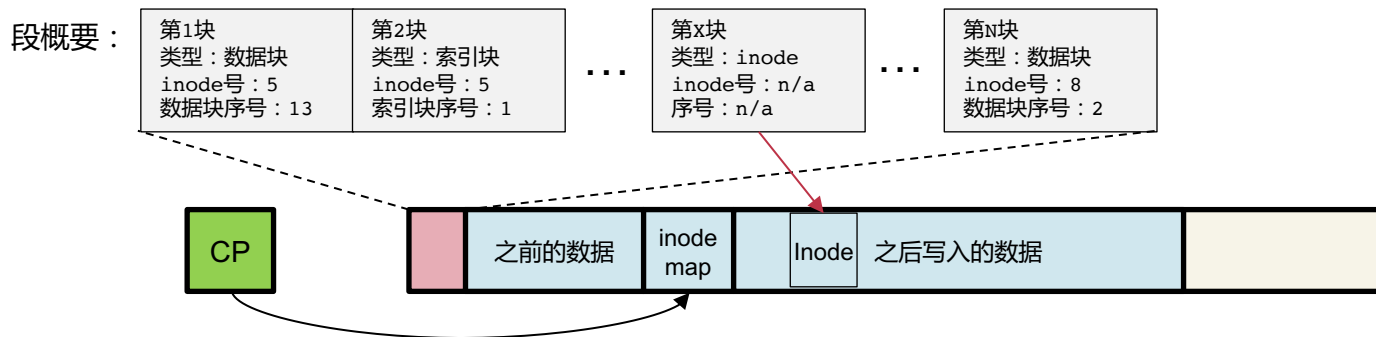
- 检查点内容
 - inode map的位置（可找到所有文件的内容）
 - 段使用表
 - 当前时间
 - 最后写入的段的指针

思考一下：为什么需要两个检查点区域？



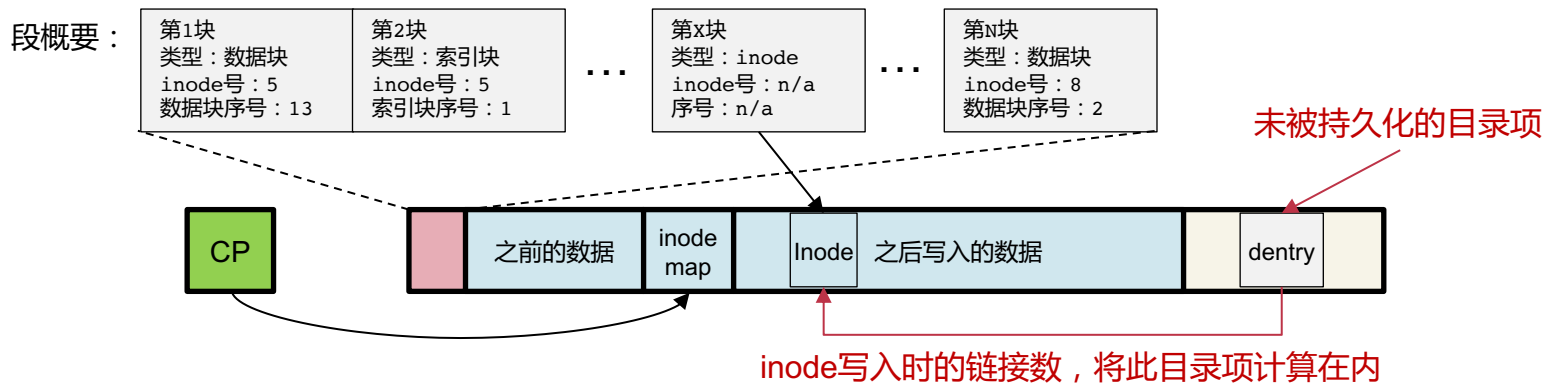
恢复：前滚（roll-forward）

- 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



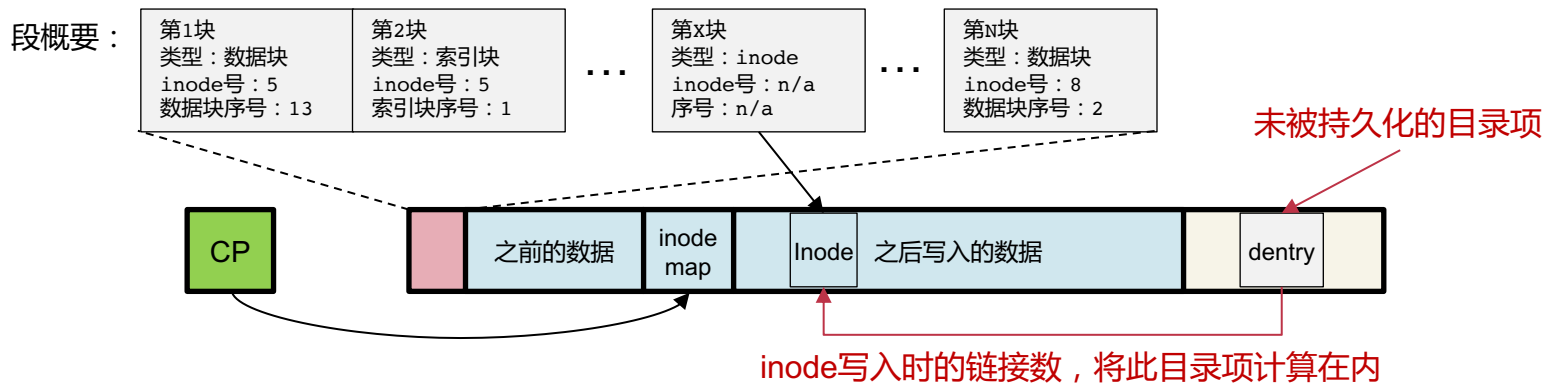
恢复：前滚 (roll-forward)

- 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？

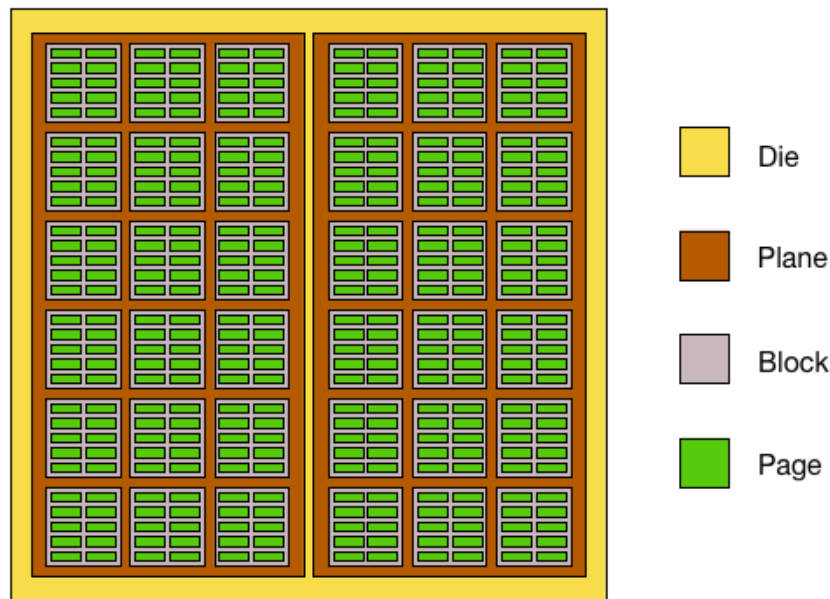
Flash友好的文件系统



F2FS

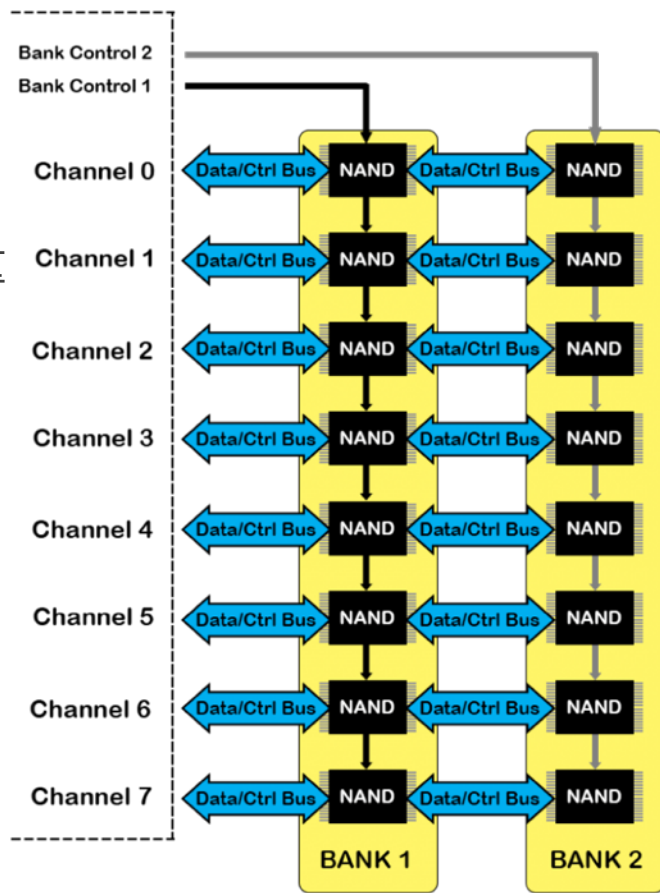
闪存盘的组织

- (NAND) 闪存盘组织结构
 - A chip/package
 - => 1/2/4 dies
 - => 1/2 planes
 - => n blocks (块)
 - => n pages (页)
 - => n cells
 - => 1/2/3/4 levels



闪存盘的组织

- **通道 (Channel)**
 - 控制器可以同时访问的闪存芯片数量
- **多通道 (Multi-channel)**
 - 低端盘有2或4个通道
 - 高端盘有8或10个通道



NAND Banks and Channels Illustration

闪存盘的性质

- **非对称的读写与擦除操作**
 - 页 (page) 是读写单元 (8-16KB)
 - 块 (block) 是擦除单元 (4-8MB)
- **Program/Erase cycles**
 - 写入前需要先擦除
 - 每个块被擦除的次数是有限的
- **随机访问性能**
 - 没有寻道时间
 - 随机访问的速度提升，但仍与顺序访问有一定差距

闪存盘的性质

- **磨损均衡**

- 频繁写入同一个块会造成写穿问题
- 将写入操作均匀的分摊在整个设备

- **多通道**

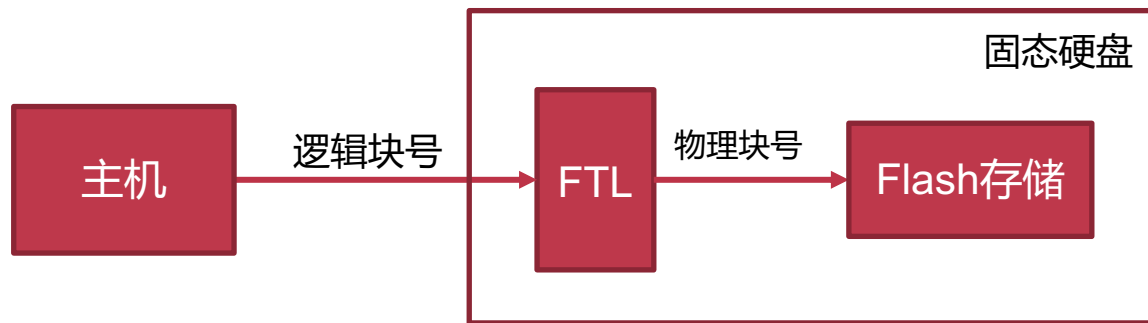
- 高并行性

- **异质Cell**

- 存储1到4个比特：SLC、MLC、TLC、QLC

Flash Translation Layer (FTL)

- 逻辑地址到物理地址的转换
 - 对外使用逻辑地址
 - 内部使用物理地址
 - 可软件实现，也可以固件实现
 - 用于垃圾回收、数据迁移、磨损均衡（wear-levelling）等



问题：如何为Flash设计文件系统？

LFS与Flash很相似？

- LFS

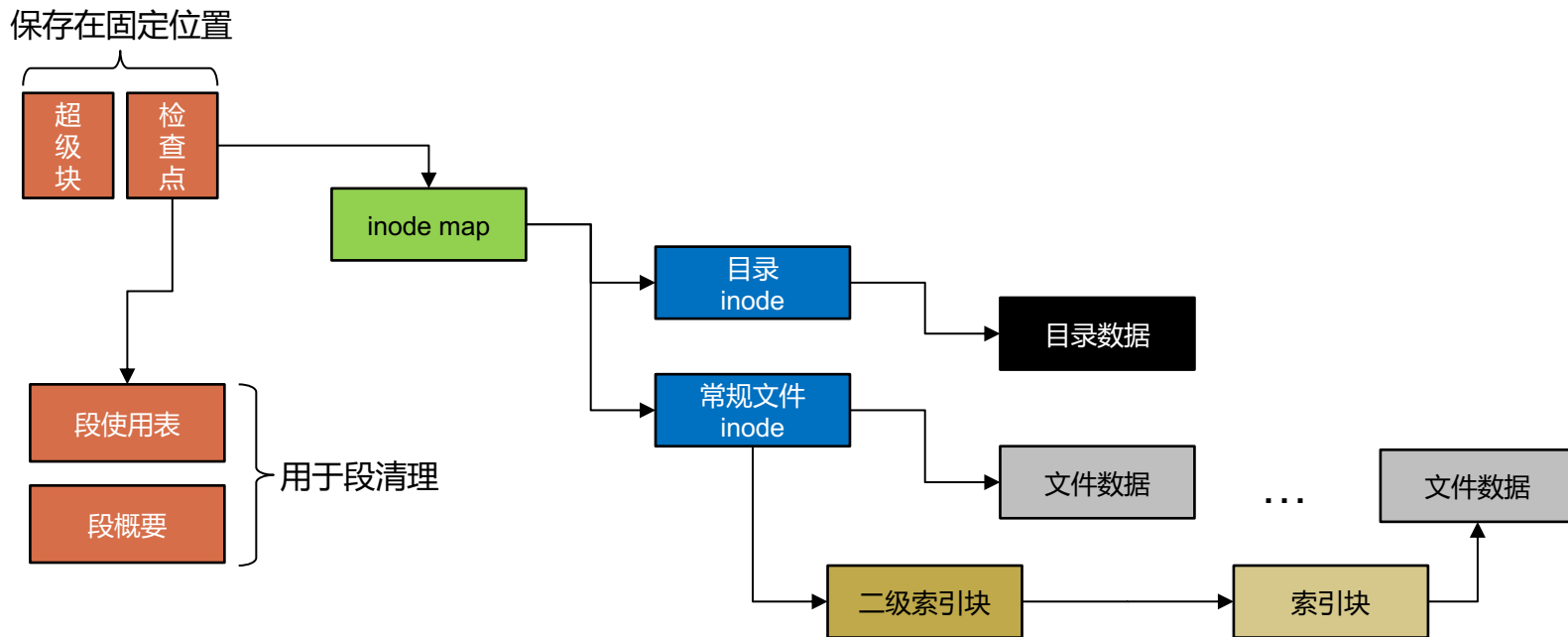
- Segment清理后才能使用
- 顺序写入
- 需要清理（垃圾回收）

- Flash

- Block擦除后才能使用
- 需要考虑磨损均衡
- 需要垃圾回收

LFS的结构

逻辑结构

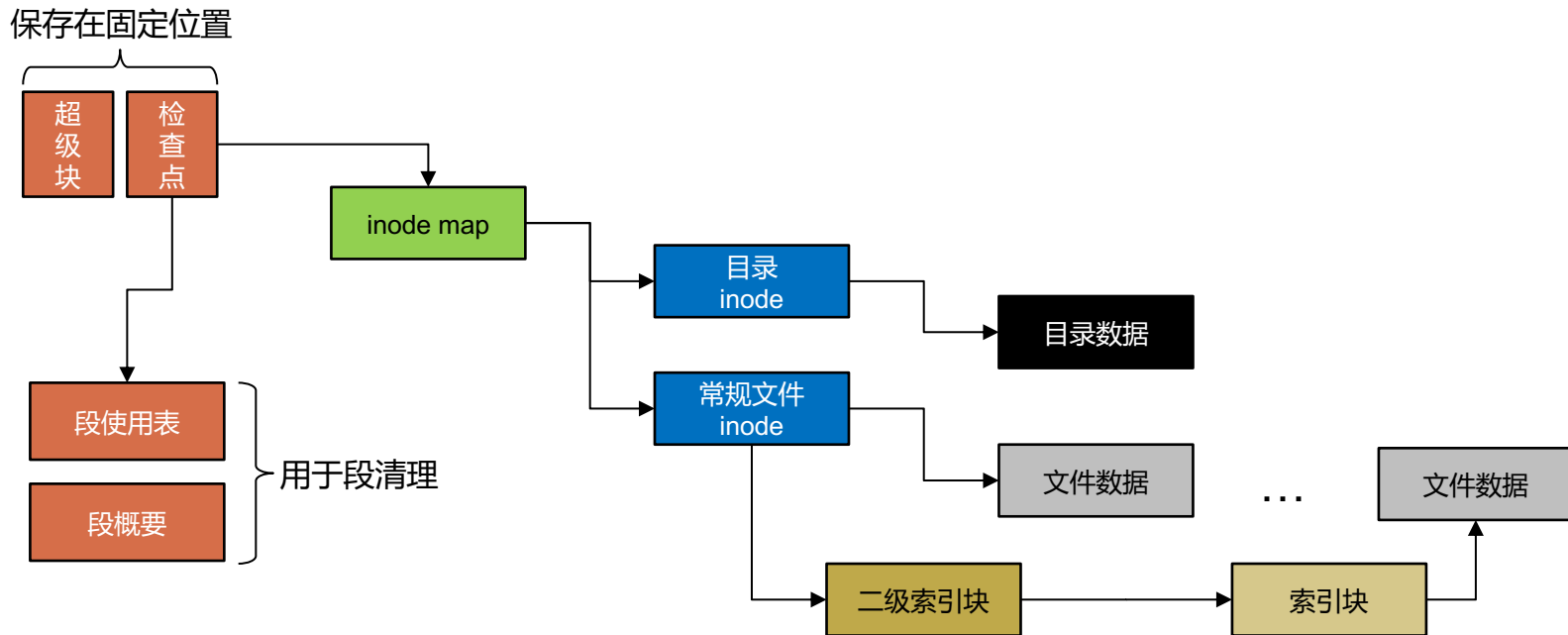


存储结构



LFS的问题1：递归更新问题

逻辑结构

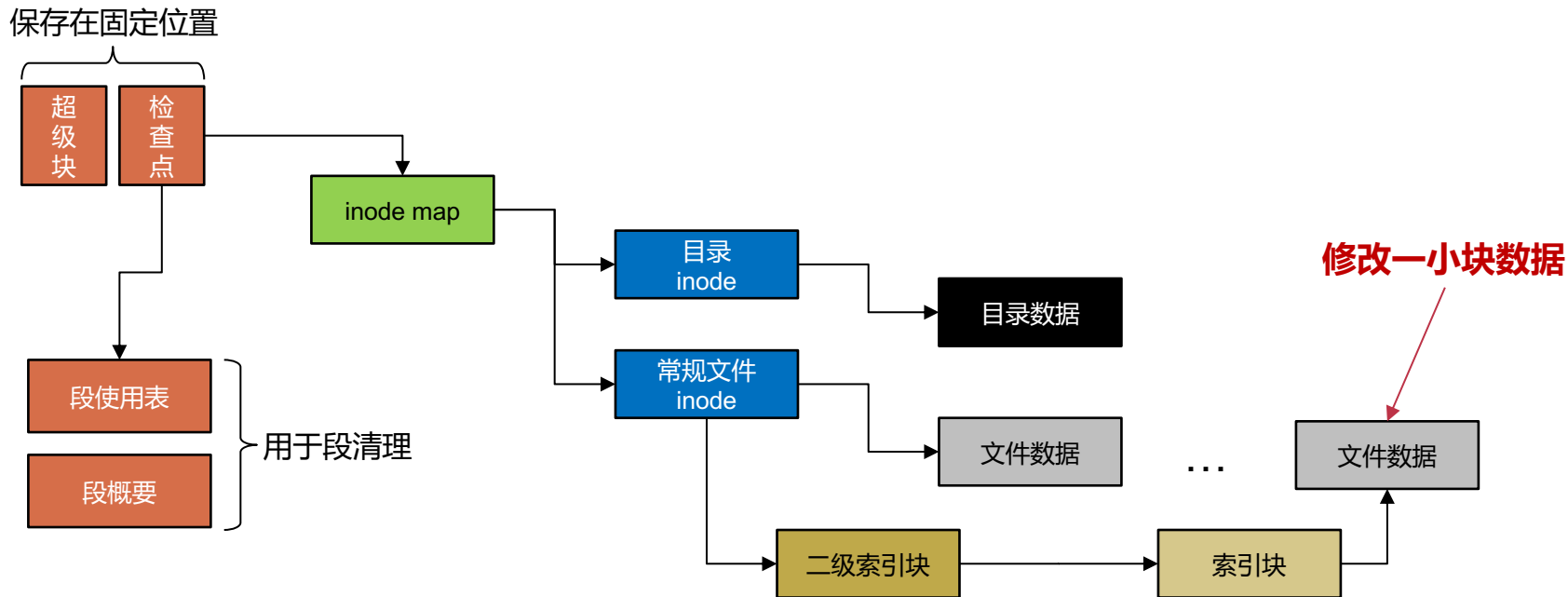


存储结构



LFS的问题1：递归更新问题

逻辑结构



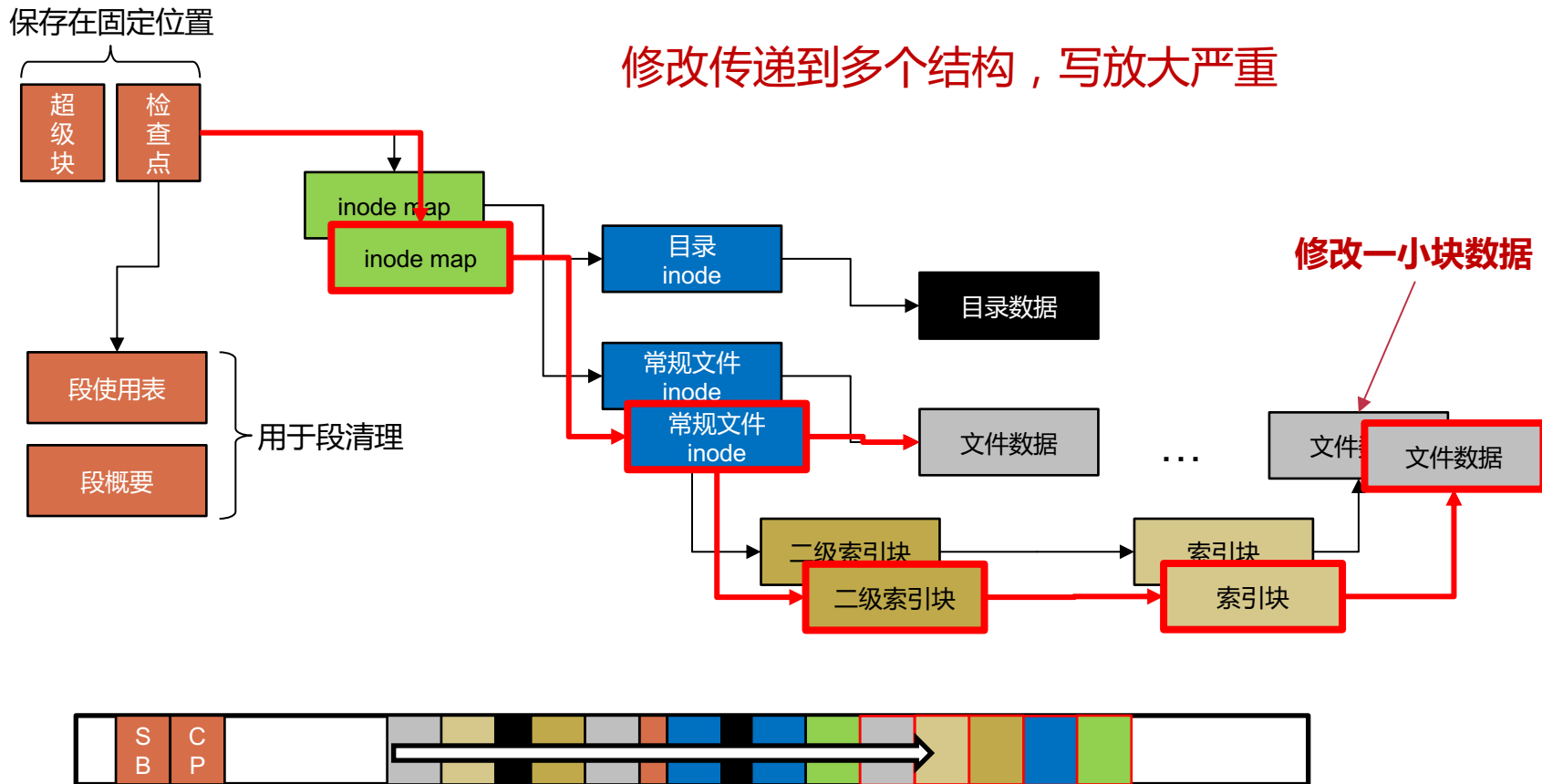
存储结构



LFS的问题1：递归更新问题

逻辑结构

存储结构

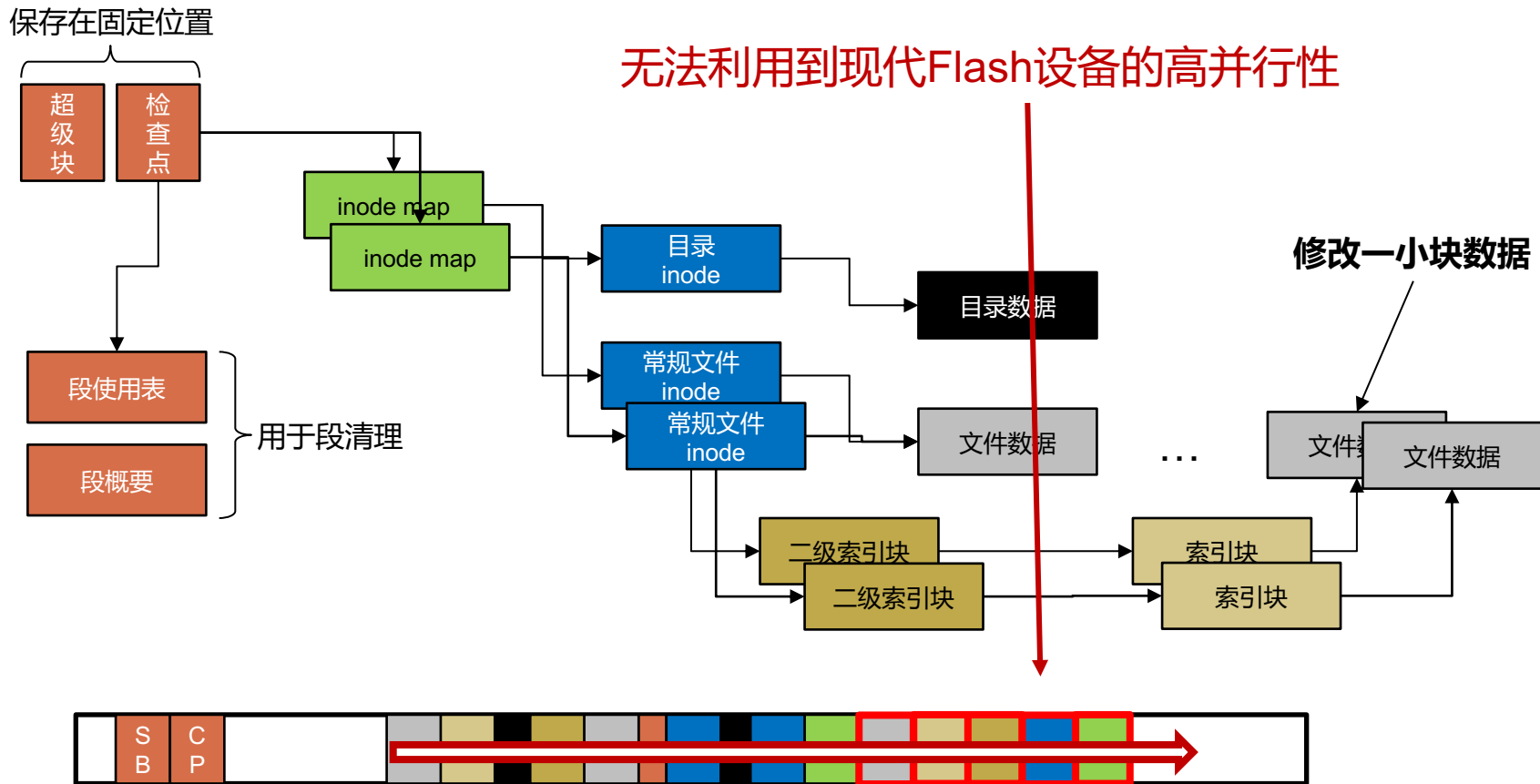


LFS的问题2：单一log顺序写入

无法利用到现代Flash设备的高并行性

逻辑结构

存储结构



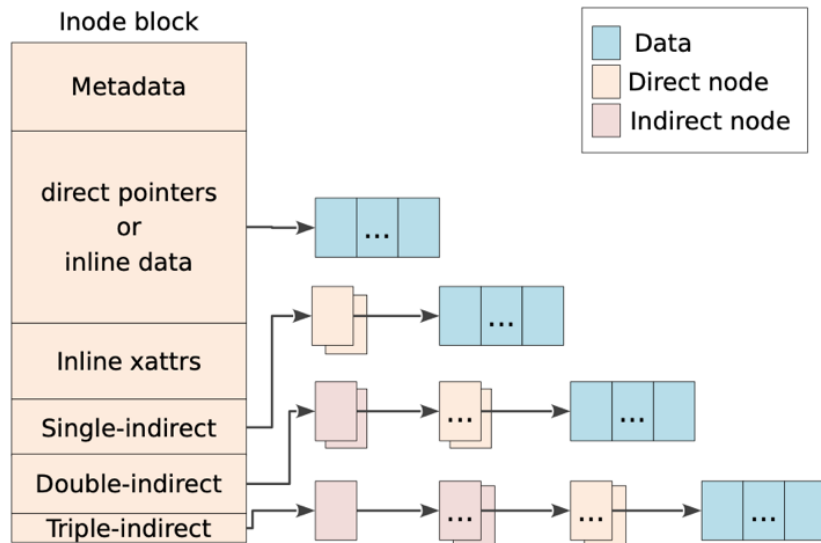
F2FS的改进1：NAT

- 引入一层 indirection：NAT (node地址转换表)

- NAT：Node Address Table
- 维护node号到逻辑块号的映射
- Node号需转换成逻辑块号才能使用

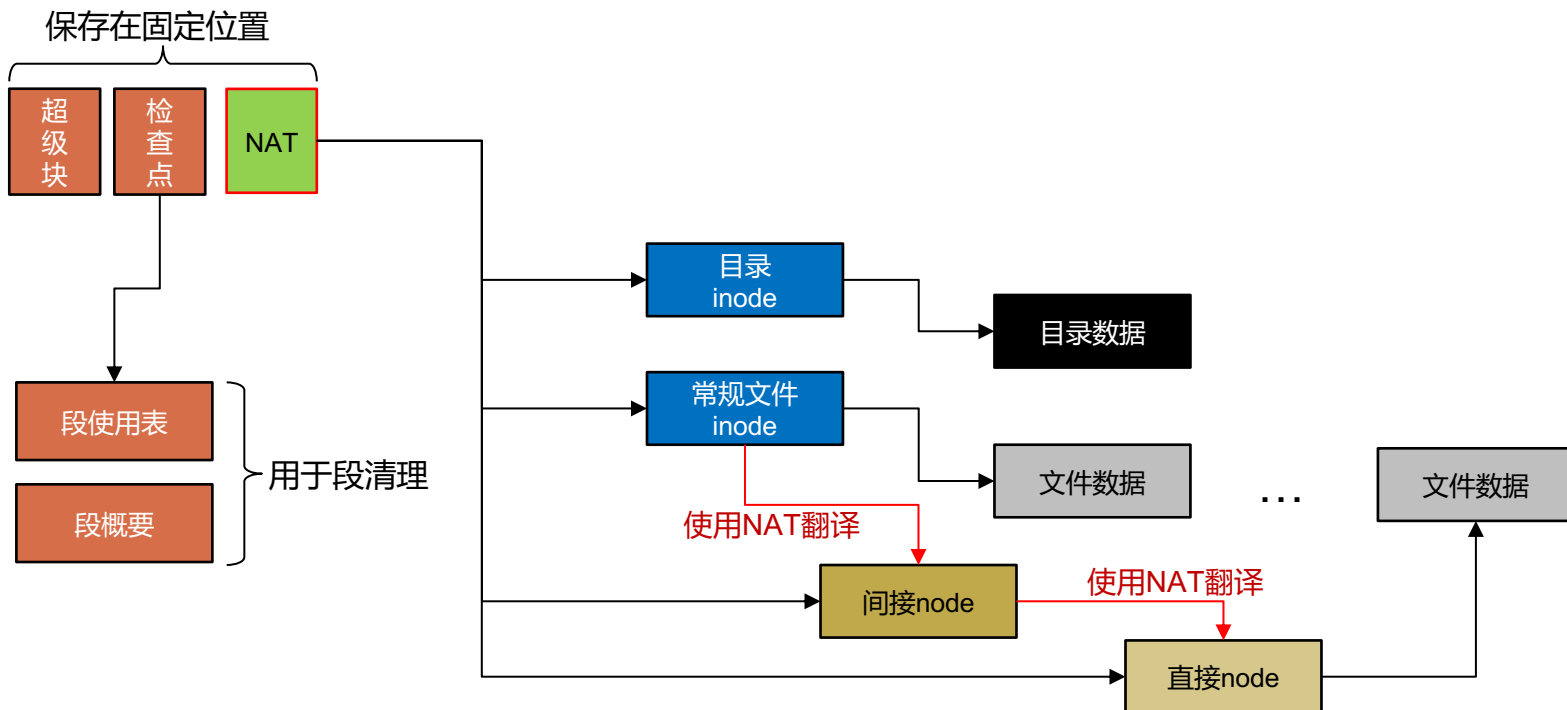
- F2FS中的文件结构

- 直接node：保存数据块的逻辑块号
- 间接node：保存node号
(相当于索引块)
- 数据块：保存数据

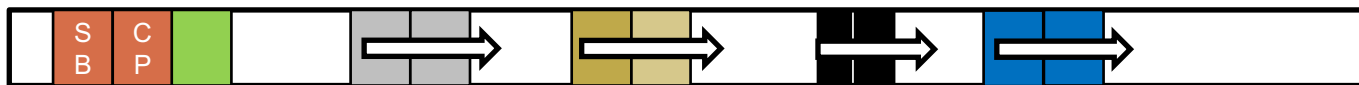


F2FS的改进1 : NAT

逻辑结构



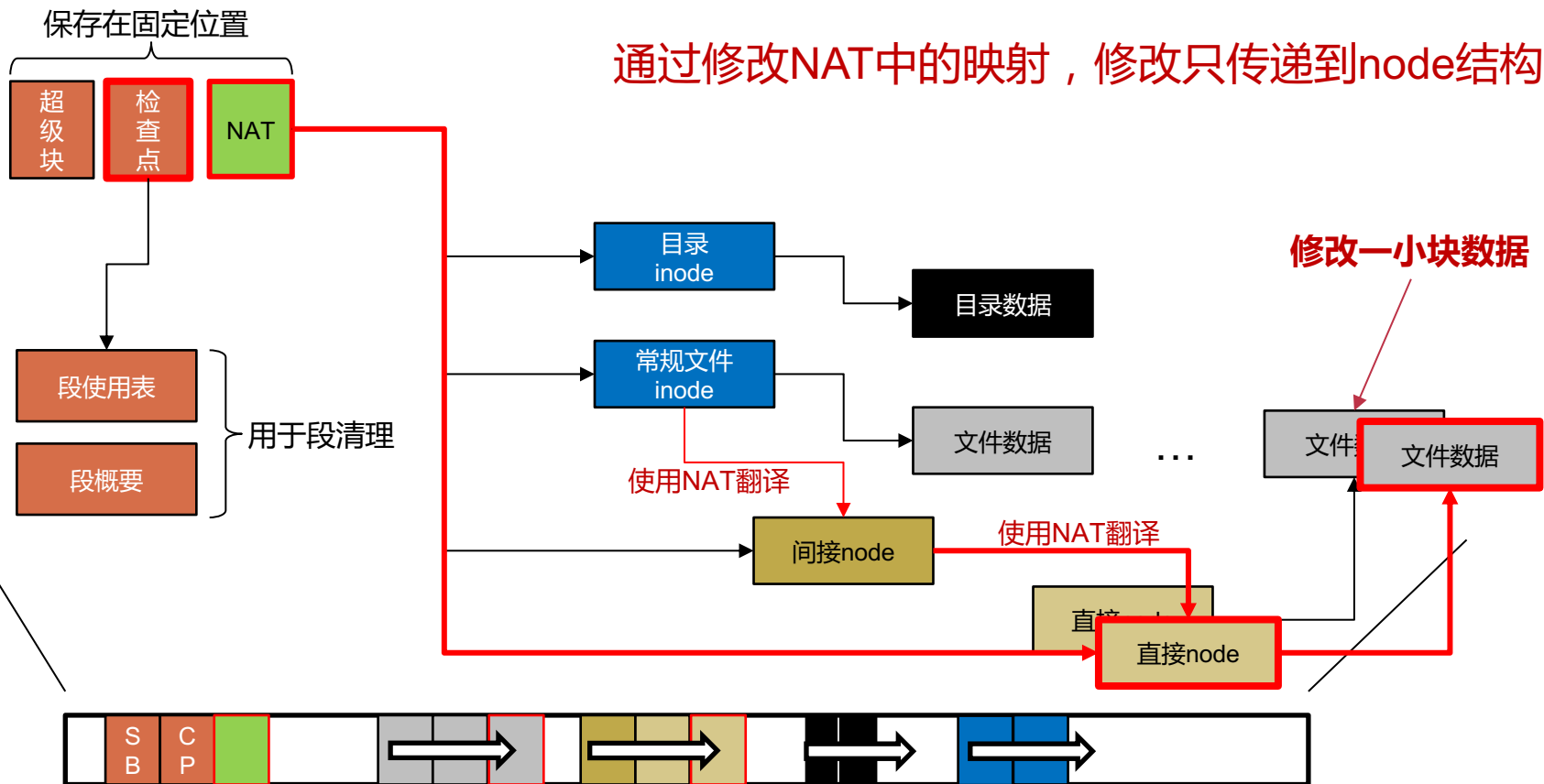
存储结构



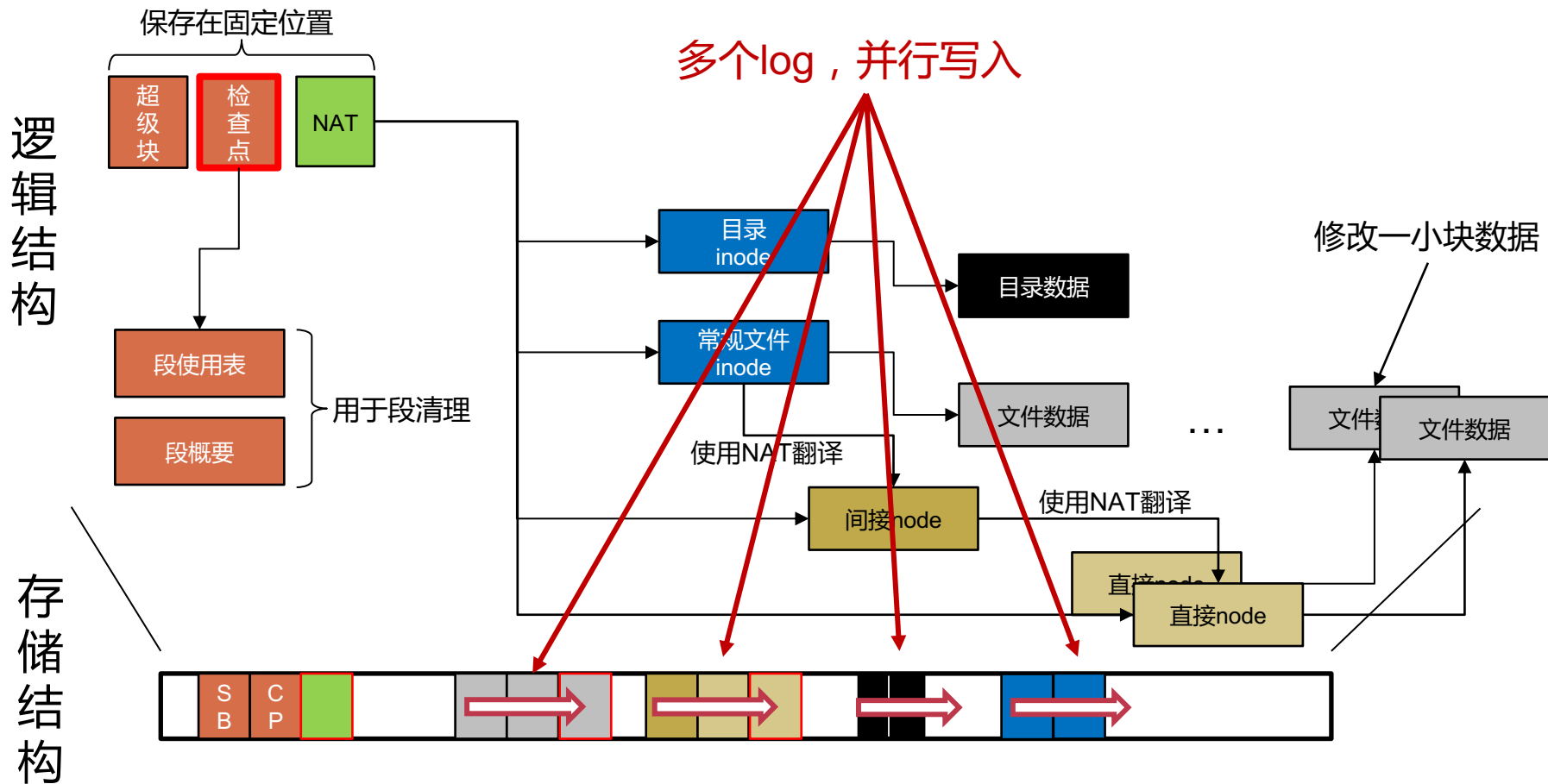
F2FS的改进1 : NAT

逻辑结构

存储结构

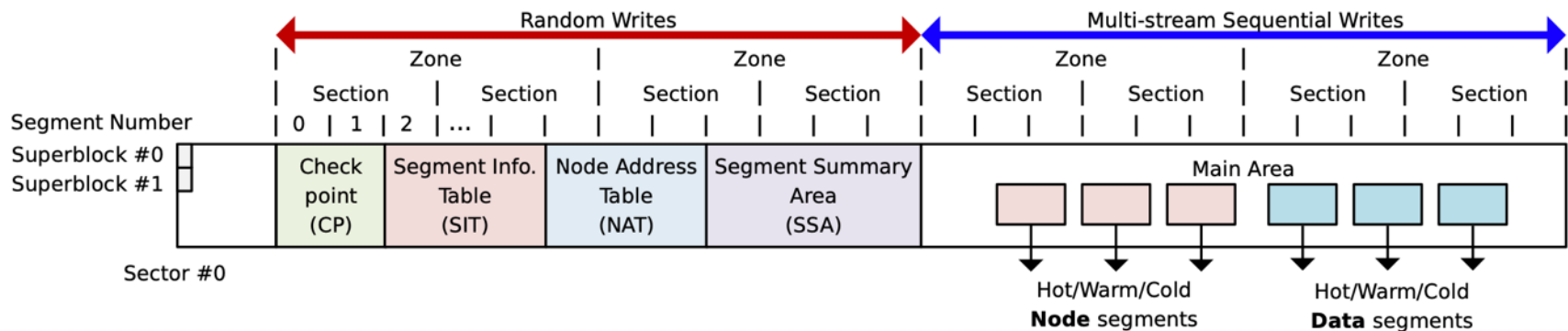


F2FS的改进2：多log并行写入



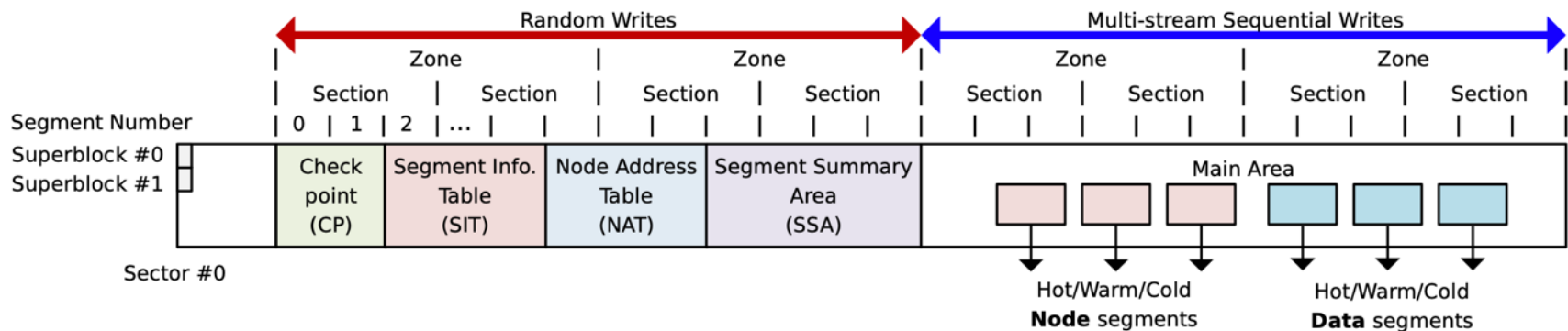
闪存友好的磁盘布局

- 组织层级
 - Block : 4KB , 最小的读写单位
 - Segment : 2MB
 - Section : 多个segment (垃圾回收/GC粒度)
 - Zone : 多个section



闪存友好的磁盘布局

- 系统元数据（随机写入）
 - 存放在一起：局部性更好
 - CP：检查点
 - SIT：段信息表
 - NAT：node地址转换表
 - SSA：段概要区域
- 数据区（多Log顺序写入）
 - 区分冷/温/热数据
 - 区分文件数据（data segment）与元数据（node segment）



多Log写入

- 按热度将结构分类
 - 每个类型和热度对应一个log
 - 默认打开6个log
 - 用户可进一步配置
- 根据硬件信息可以进一步调整
 - 调整zone、section大小
 - 与硬件GC单元对齐等

类型	热度	对象
Node	热	目录的直接node块（包括inode块）
	温	常规文件的间接node块
	冷	间接node块
Data	热	存放目录项的数据块
	温	常规文件的数据块
	冷	被清理过程移动的数据块 用户指定的冷数据块 多媒体文件的数据块

清理 (Cleaning)

回想一下：LFS为何要进行清理？

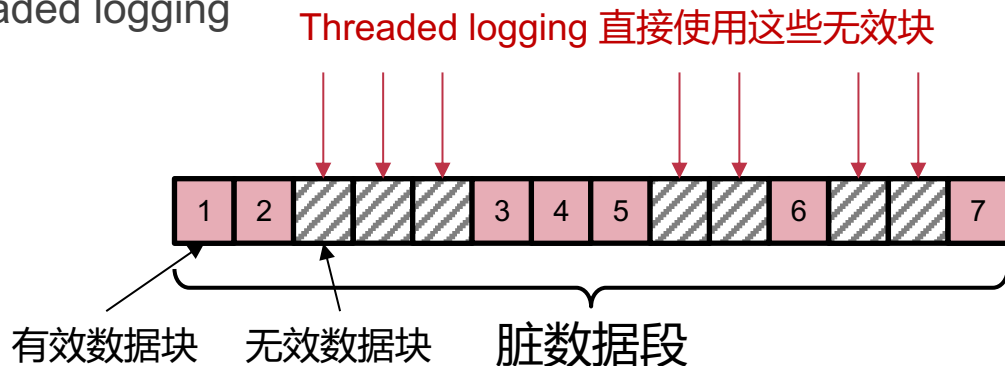
- 以section为粒度
 - 与硬件FTL的GC单位是对齐的

清理 (Cleaning)

- 以section为粒度
 - 与硬件FTL的GC单位是对齐的
 - 过程 (与LFS类似)
 1. 选择需要清理的section
 - Greedy : 选择有效块最少的section
 - Cost-effective : 同时考虑数据修改时间
 2. 识别有效数据
 3. 有效数据拷贝到干净section
 4. 标记被清理的section为pre-free
 - 在下一次checkpoint之后被标记为free
- 思考一下：为什么不直接标记为free？

自适应日志

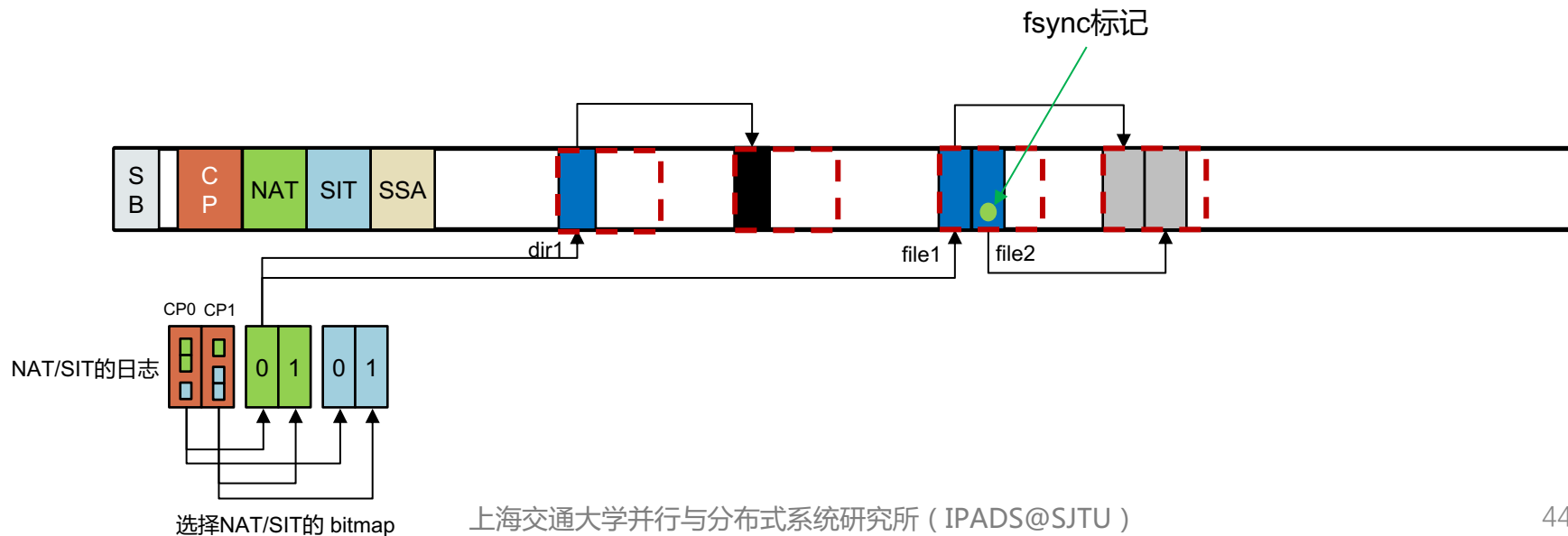
- 文件系统使用一段时间后，干净section不足，需要频繁清理
- F2FS动态调整数据段的日志方法
 - 干净section充足时，使用常规方法
 - 日志写到干净section中
 - 没有干净section时需要进行清理操作
 - 干净section不足时，使用 threaded logging
 - 使用脏段中无效的块
 - 避免清理操作
 - 但会产生一些随机写



检查点

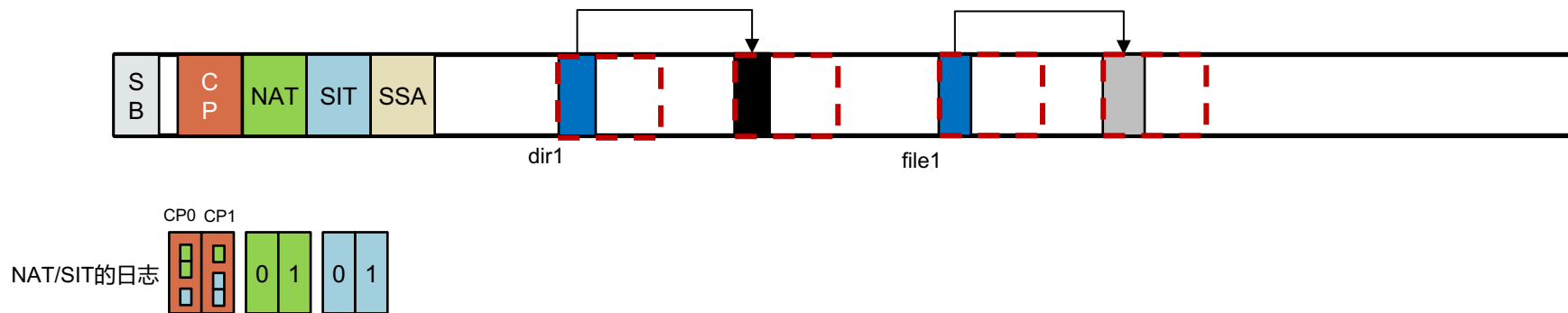
回想一下：为何需要检查点？

- 检查点、NAT、SIT各有两份
- 检查点中保存有NAT和SIT的日志，避免NAT和SIT的频繁更新
- 恢复时回滚到最近的检查点



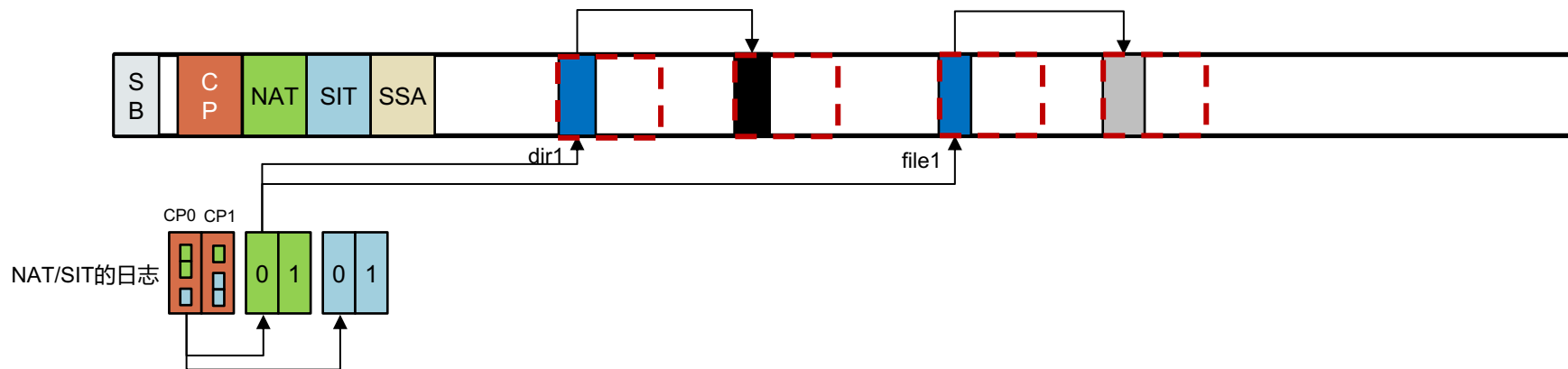
崩溃与恢复

1. 创建dir1和file1



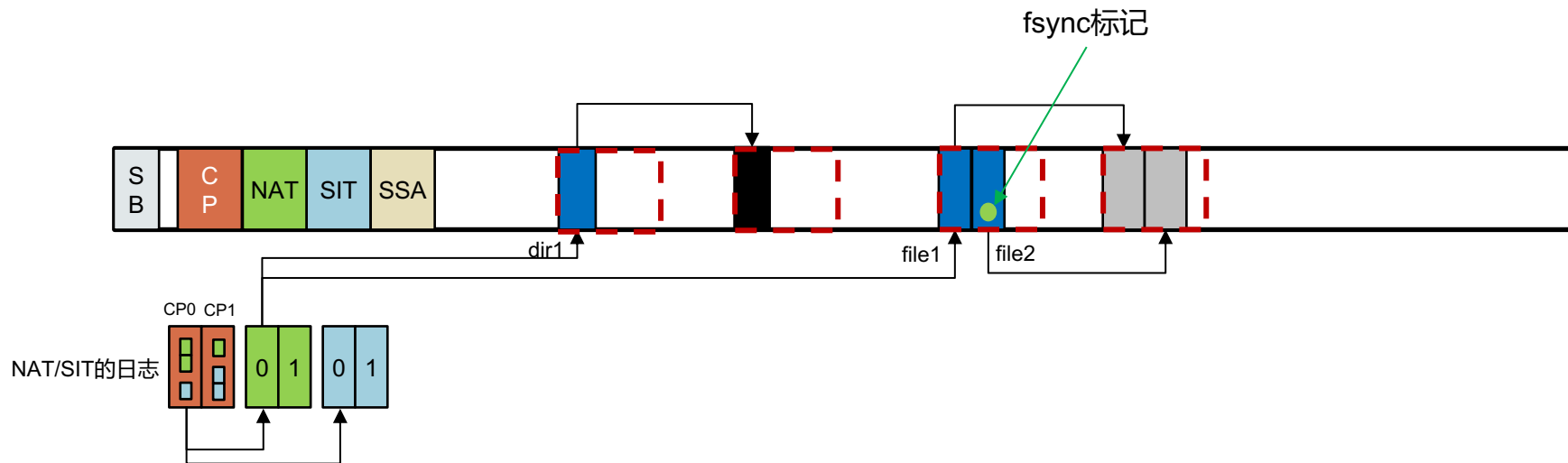
崩溃与恢复

1. 创建dir1和file1
2. 创建检查点0



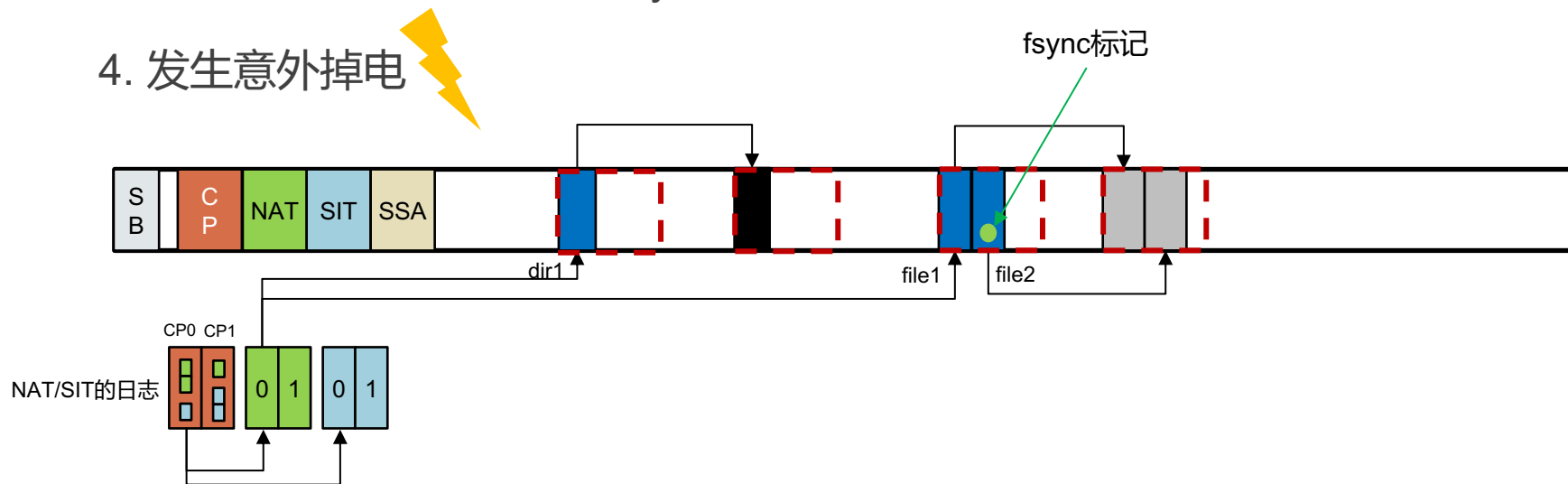
崩溃与恢复

1. 创建dir1和file1
2. 创建检查点0
3. 创建和更新file2，并执行fsync完毕



崩溃与恢复

1. 创建dir1和file1
2. 创建检查点0
3. 创建和更新file2，并执行fsync完毕
4. 发生意外掉电



崩溃与恢复：回滚 (roll-back)

1. 创建dir1和file1

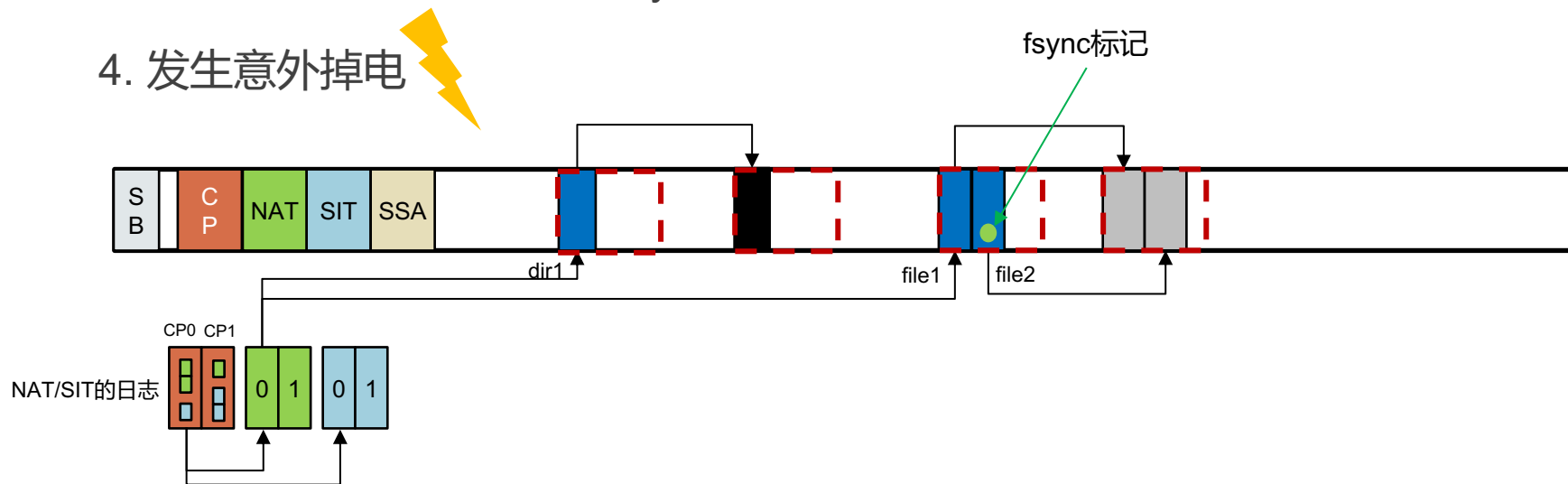
2. 创建检查点0

3. 创建和更新file2，并执行fsync完毕

4. 发生意外掉电

恢复： 1. 回滚到最近的检查点 (CP0)

— 可找到文件dir1和file1



崩溃与恢复：前滚 (roll-forward)

1. 创建dir1和file1

2. 创建检查点0

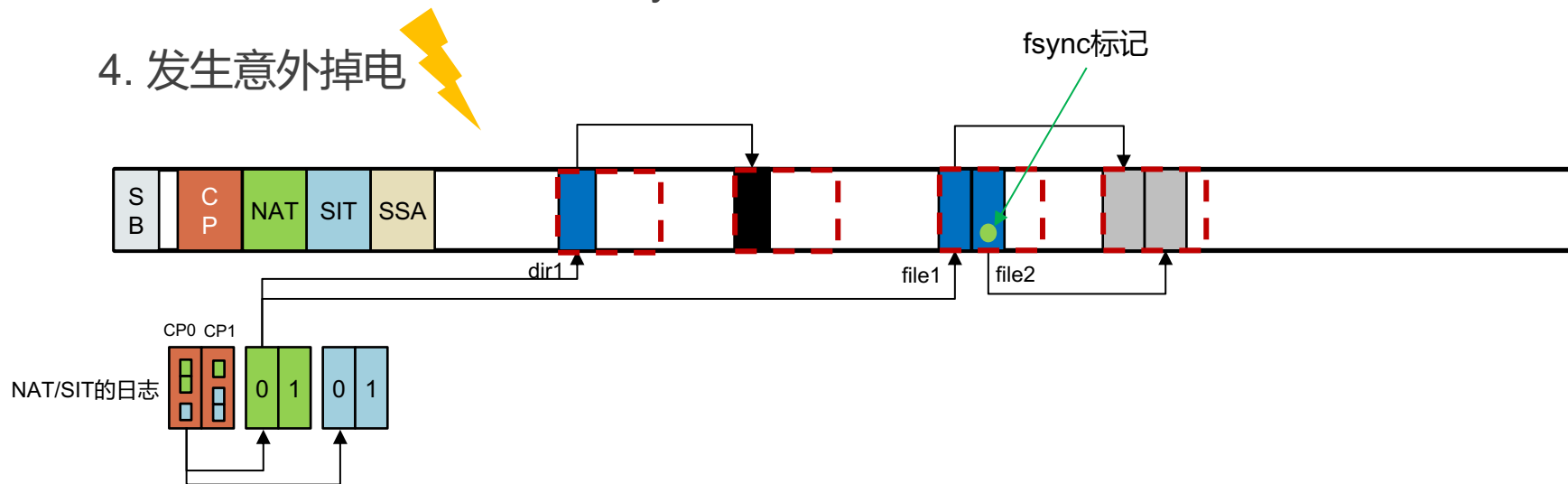
3. 创建和更新file2，并执行fsync完毕

4. 发生意外掉电

恢复： 1. 回滚到最近的检查点 (CP0)

— 可找到文件dir1和file1

2. 前滚恢复fsync过的file2



前滚：fsync()的处理

- 原有LFS
 - 创建新的检查点
- F2FS
 - 无需创建新的检查点
 - 持久化文件数据块和直接node，并在直接node上附带fsync标记
 - 前滚：恢复检查点之后fsync过的数据

前滚：恢复检查点之后fsync过的数据

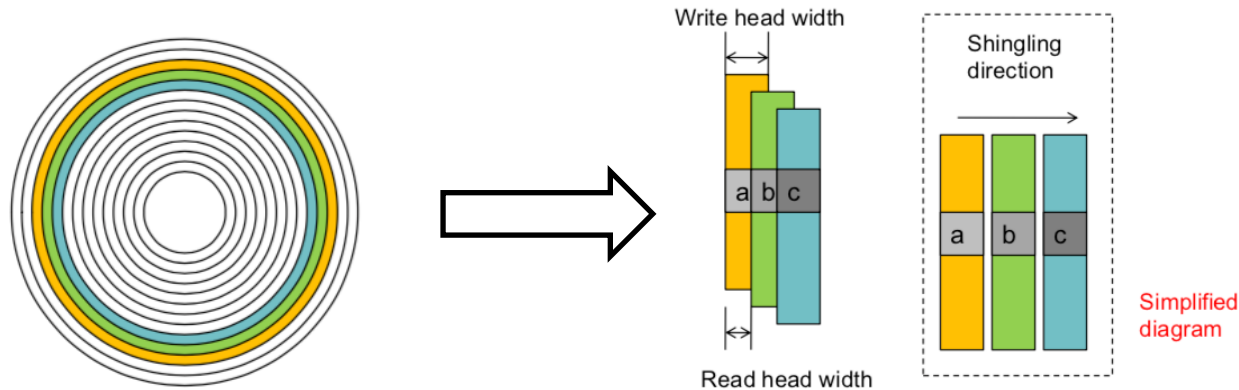
- 前滚恢复
 1. 查找带有fsync标记的直接node
 2. 对于每个直接node，对比其中的数据块指针，识别新旧数据块
 3. 更新SIT，标记旧的数据块为无效
 4. 根据直接node中新数据块的记录，更新NAT和SIT
 5. 创建新的检查点

Shingled Magnetic Recording (SMR) Disk

瓦式磁盘

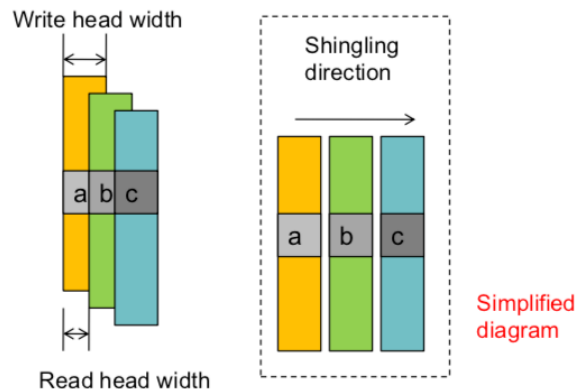
瓦式磁盘

- 传统磁盘密度难以提升
 - 写磁头的宽度难以减小
- 瓦式磁盘将磁道重叠，提升存储密度
 - 减小读磁头的宽度



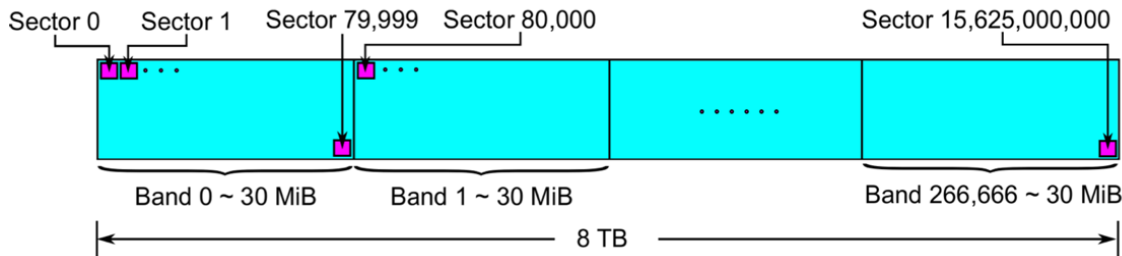
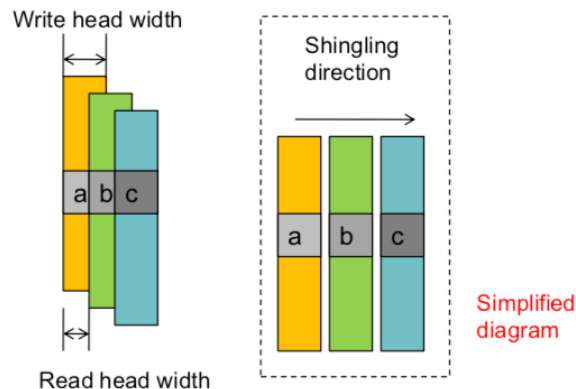
瓦式磁盘的问题：随机写

- 随机写会覆盖后面磁道的数据
 - 只能顺序写入



瓦式磁盘的问题：随机写

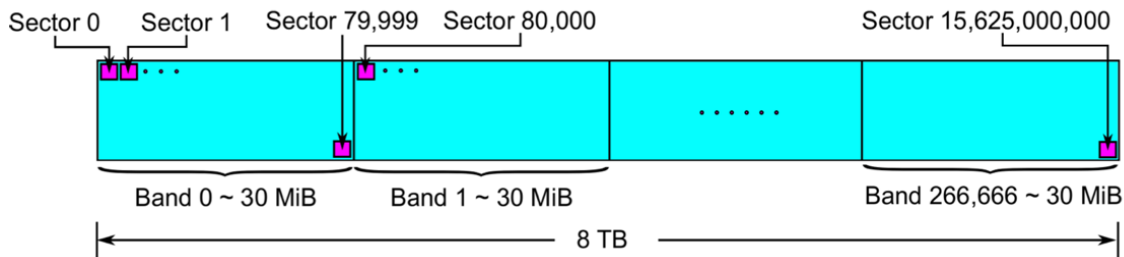
- 随机写会覆盖后面磁道的数据
 - 只能顺序写入
- 避免整个磁盘只能顺序写入
 - 磁盘划分成多个Band，Band间增大距离
 - 每个Band内必须顺序写入



Logical view of an 8 TB statically mapped drive-managed SMR disk

Band内随机写怎么办？

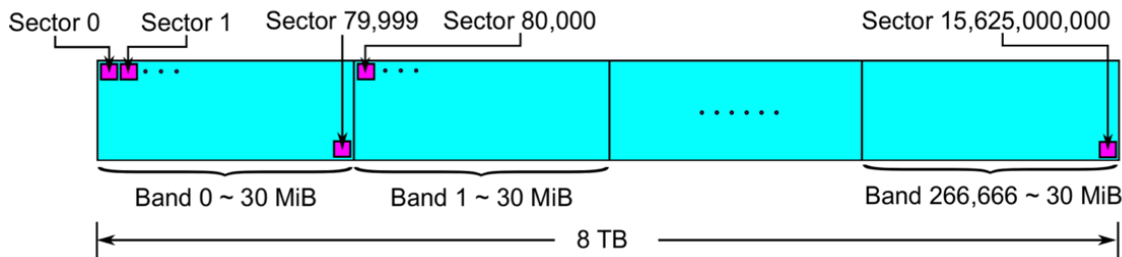
- Band大小 (30MB) > 块设备读写粒度 (4KB)



Logical view of an 8 TB statically mapped drive-managed SMR disk

方法一：多次拷贝

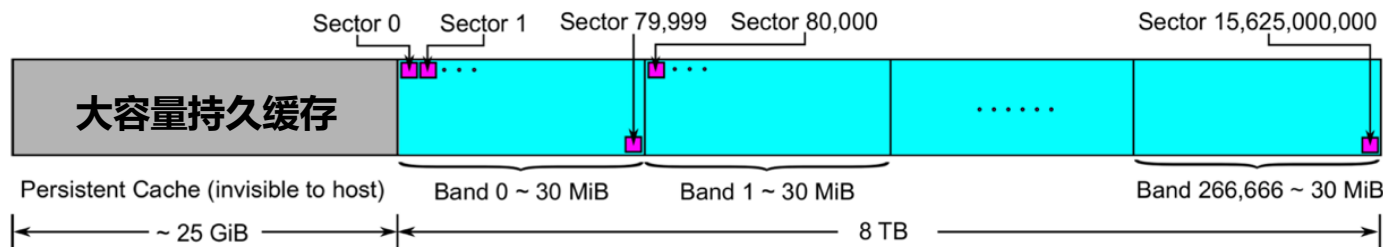
- 修改Band X中的4KB数据
 1. 找到空闲Band Y
 2. 从Band X的数据拷贝到Band Y，拷贝时将4KB修改写入
 3. 将Band Y中的数据拷贝回Band X
- 4KB随机写 → 120MB访问



Logical view of an 8 TB statically mapped drive-managed SMR disk

方法二：缓存+动态映射

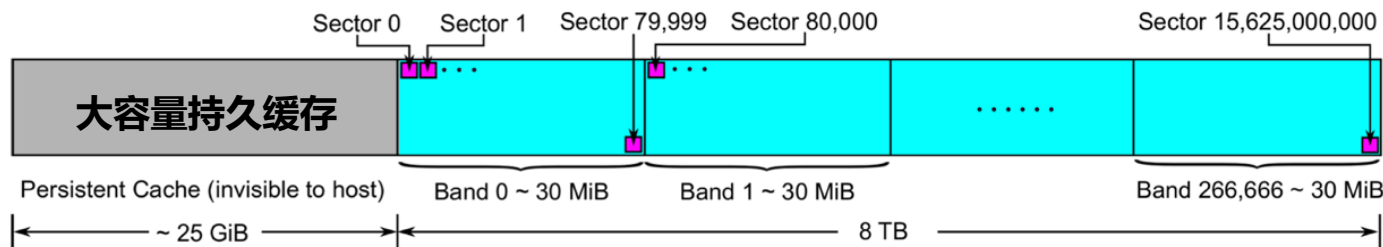
- 大容量持久缓存
 - 在磁盘头部预留的区域，磁道不重叠，可随机写入
 - 给固件（STL）单独使用，外部不可见
- 动态映射：Shingle Translation Layer (STL)
 - 从外部（逻辑）地址到内部（物理）地址的映射



Logical view of an 8 TB statically mapped drive-managed SMR disk

方法二：缓存+动态映射

- 修改Band X中的4KB数据
 1. 将修改写入缓存，标记Band X为dirty
 2. 修改STL映射（让原位置指向持久化缓存）
 3. 空闲时，根据缓存内容，将 dirty Band进行清理
- 4KB随机写 → 修改4KB缓存



Logical view of an 8 TB statically mapped drive-managed SMR disk

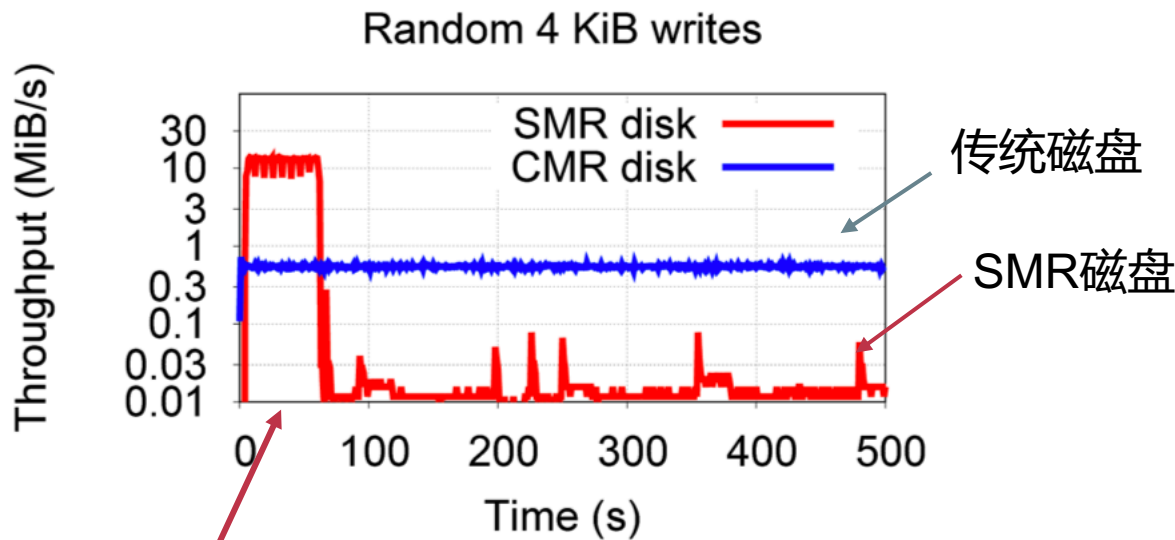
瓦式磁盘种类

SMR磁盘种类	接口	随机写处理方法
Drive-managed SMR (DM-SMR)	普通块设备接口	固件进行缓存和清理
Host-aware SMR (HA-SMR)	特殊指令接口	固件进行缓存和清理
Host-managed SMR (HM-SMR)	特殊指令接口	必须顺序写，随机写请求被拒绝

无需修改软件，可直接替换传统磁盘！

DM-SMR上使用Ext4

- 当随机写入时，Ext4吞吐量非常低！

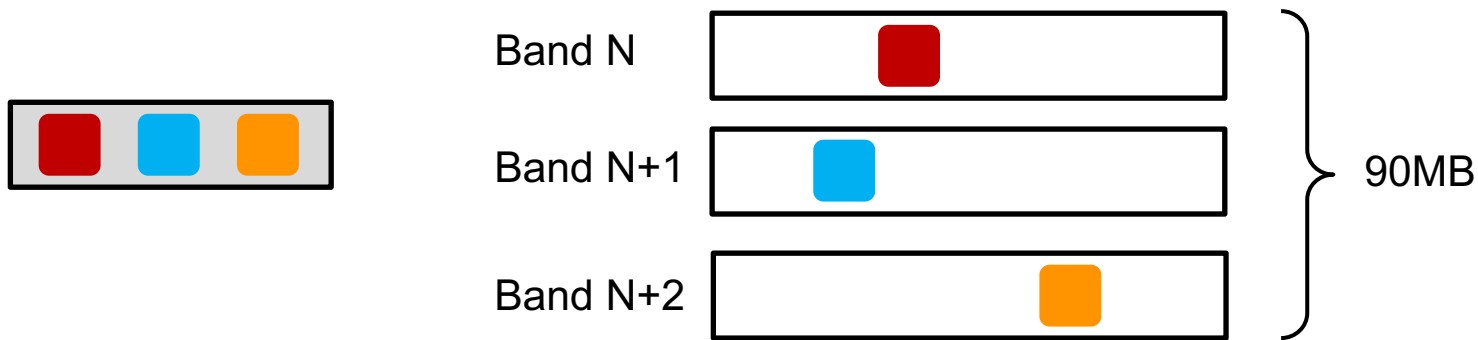


思考一下：为何这段时间内SMR磁盘上吞吐量很高？

如何改进Ext4来适应瓦式磁盘呢？

- **以DM-SMR磁盘为目标**
 - HM-SMR和HA-SMR需要文件系统的设计新的文件系统
 - 在成熟的Ext4上优化：消除元数据写回造成的随机写入
- **修改量小**
 - Ext4 + journaling 代码 ~50,000 行
 - 仅仅修改 ~40 行代码，新文件中添加 ~600 LOC 代码
- **效果显著**
 - 元数据修改较少时(<1%)，在SMR磁盘上有1.7-5.4倍性能提升
 - 有大量元数据修改时，在SMR和普通磁盘上，有2-13倍性能提升
 - 特定场景下达到40倍性能提升

观察：持久缓存对吞吐量的影响



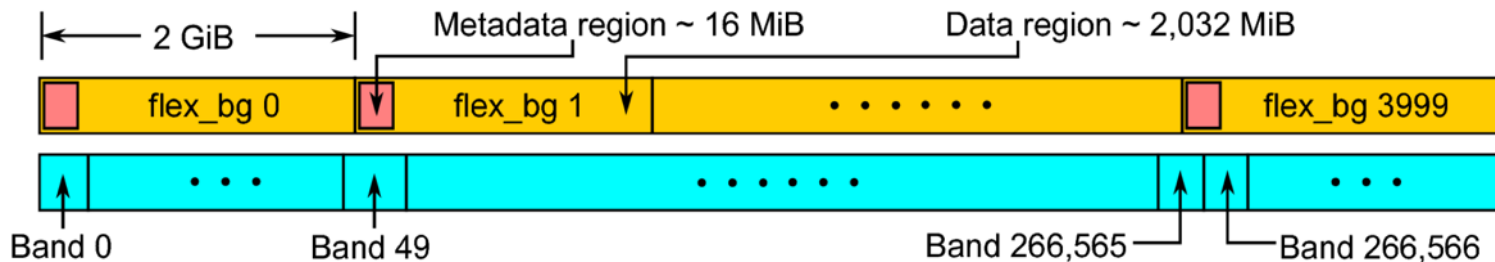
若持久化缓存中的写比较分散，清理时需要清理大量Band
假设清理1个Band需要1秒，吞吐量为1个随机写/秒



若三个随机写在一个Band中，则只需清理一个Band；吞吐量为3个随机写/秒
随机写的跨度 → 脏band数量 → 清理时的工作量 → 吞吐量

然而：Ext4的元数据非常分散

- Ext4使用块组 (flex_bg) 将文件系统分成多个区域
 - 每个块组前16MB用来保存元数据，其余保存数据
- 每次数据修改产生多处元数据的修改
- 8TB分区上有4,000个块组，元数据分散在4,000个Band！
- 分散的元数据随机写 → 脏band数量↓ → 清理工作的负担↓ → 吞吐量↑

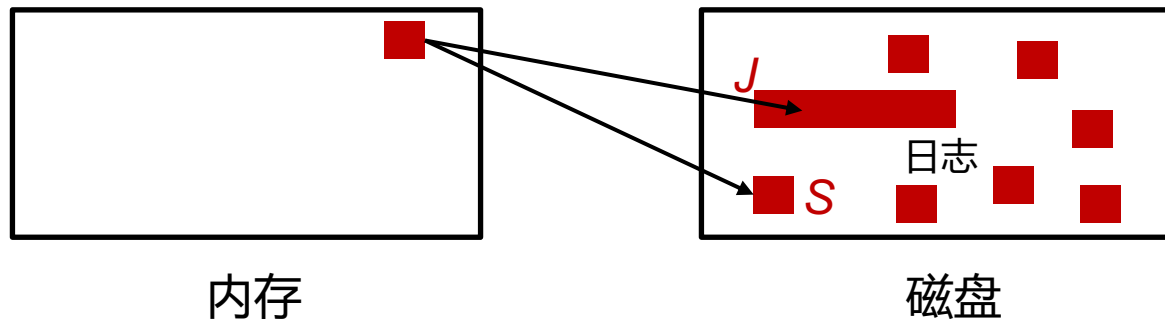


Ext4 partition laid on top of an 8 TB SMR disk

回顾：Ext4上的元数据写回

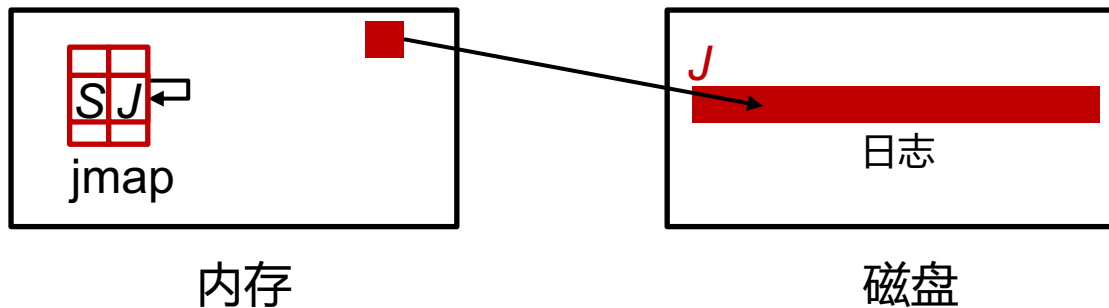
- Ext4使用JBD2记录元数据日志
 - 128MB的日志区域
 - 1. JBD2首先将元数据写入日志区域 J ，标记元数据为脏
 - 2. 脏元数据在日志提交后被写回到其应有位置 S

频繁的元数据写回，造成大量的分散随机写，降低吞吐量



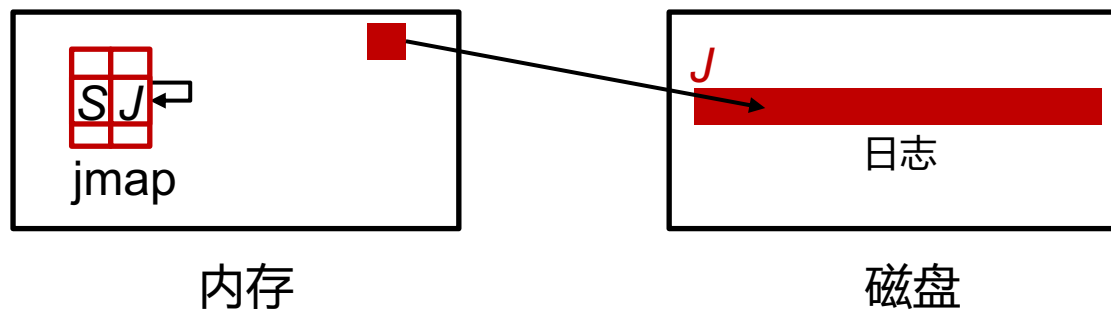
回到问题：Ext4上的元数据分散

- 修改磁盘布局需要大规模修改Ext4
 - 人力成本、新增Bug、破坏原有功能.....
- 怎么办？
- 引入Indirection：以LFS形式增加一个元数据缓存



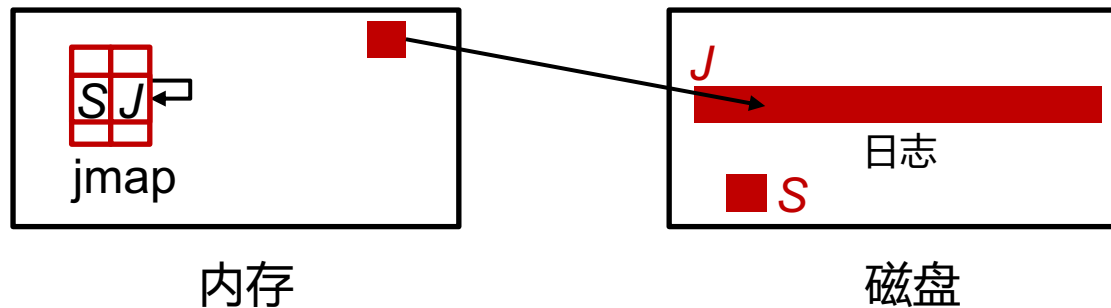
解决方法：以LFS形式增加一个元数据缓存

- 以LFS形式维护10GB日志空间作为元数据缓存
 - JBD2首先将元数据写入日志区域 J ，将元数据标记为clean（无需写回）
 - JBD2在内存中的jmap中将 S 映射到 J
- Indirection: 元数据访问需要通过jmap进行一次地址转换



日志满了怎么办？

- 日志空间清理
 - 无效的元数据（被新修改覆盖过的元数据）可以直接被回收
 - 对于冷元数据，可将其写回到Ext4中其原本的位置S
 - 热元数据继续保留在日志中
- 挂载FS时，读取日志，恢复出jmap



实现：对Ext4的修改

修改的~40行代码：将请求转给jbd2

```
- submit_bh(READ | REQ_META | REQ_PRIO, bh);  
+ jbd2_submit_bh(journal, READ | REQ_META | REQ_PRIO, bh, __func__);
```

新增文件：jmap.c (维护jmap)

```
+void jbd2_submit_bh(journal_t *journal, int rw, struct buffer_head *bh,  
+                  const char *func)  
{  
+    sector_t fsblk = bh->b_blocknr;  
+    ...  
+    je = jbd2_jmap_lookup(journal, fsblk, func);  
+    if (!je) {  
+        submit_bh(rw, bh);  
+        return;  
+    }  
+    ...  
+    logblk = je->mapping.logblk;  
+    read_block_from_log(journal, rw, bh, logblk);  
+}
```

拦截Ext4对block的请求

查找jmap

不在jmap中，直接访问磁盘对应block

根据jmap中记录的地址，在日志中读取block

实现：对Ext4的修改

新增文件：cleaner.c（维护10GB日志空间）

```
+static void do_clean_batch(struct work_struct *work) 将10GB日志空间的元数据写回到其在Ext4中的原本位置
+{
+    struct cleaner_ctx *ctx;
+    handle_t *handle = NULL;
+    int nr_live, err;
+    ...
+    nr_live = find_live_blocks(ctx);
+    if (nr_live == 0)
+        goto done;
+    ...
+    read_live_blocks(ctx, nr_live);
+    handle = jbd2_journal_start(ctx->journal, nr_live);
+    ...
+    attach_live_blocks(ctx, handle, nr_live);
+    err = jbd2_journal_stop(handle);
+done:
+    ...
+}
```

扫描日志中有效块，并加入ctx中临时保存

新的JBD2原子更新

将ctx中的有效块加入到JBD2原子更新中

结束JBD2原子更新