IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

文件系统结构

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

Inode、文件、目录、链接

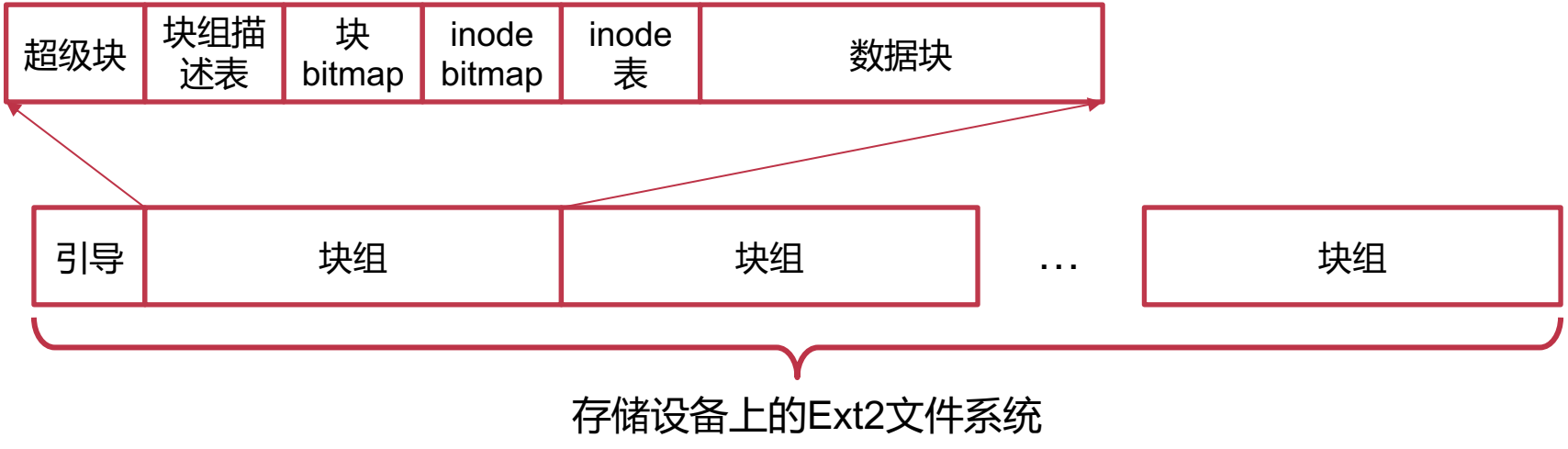
基于INODE的文件系统

Ext2存储布局

将磁盘分为多个块组，每个块组中都有超级块，互为备份

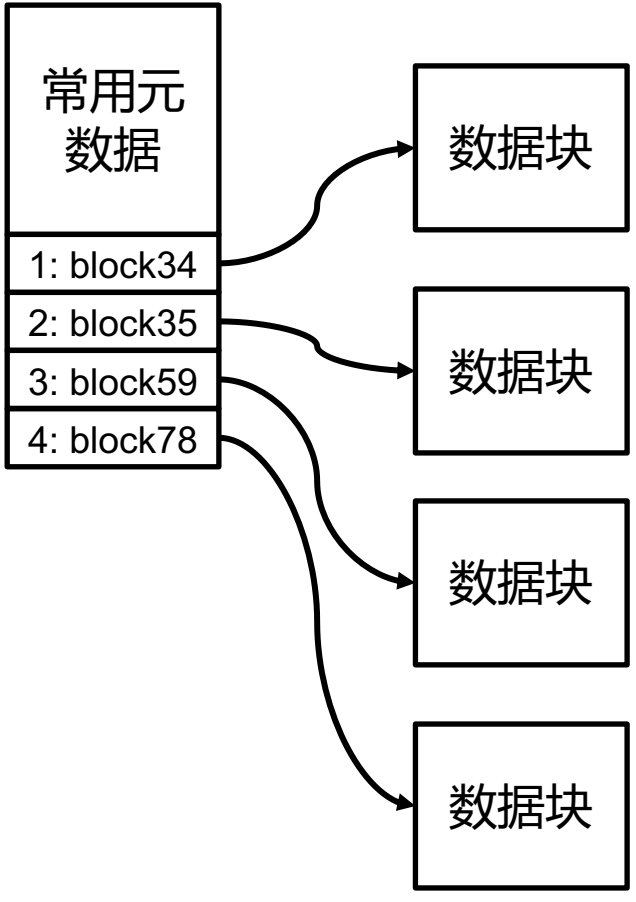
超级块（Super Block）记录了整个文件系统的元数据

块组描述表记录了快速中各个区域的位置和大小



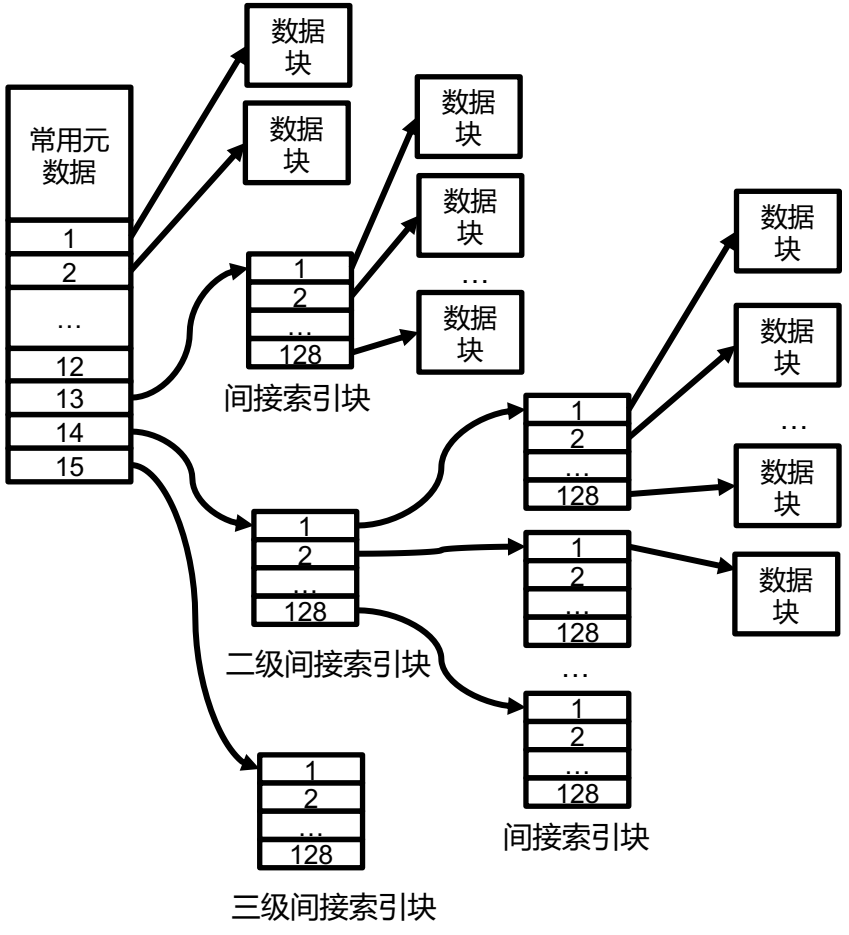
文件的索引节点：inode

- 常用的元数据
  - 文件类型
  - 文件大小
  - 链接数
  - 文件权限
  - 拥有用户/组
  - 时间（创建、修改、访问时间）
- 具体文件数据的位置



Ext2的常规文件

- Ext2的inode中
  - 12个直接指针
  - 1个间接指针
  - 1个二级间接指针
  - 1个三级间接指针



使用区段（Extent）来优化

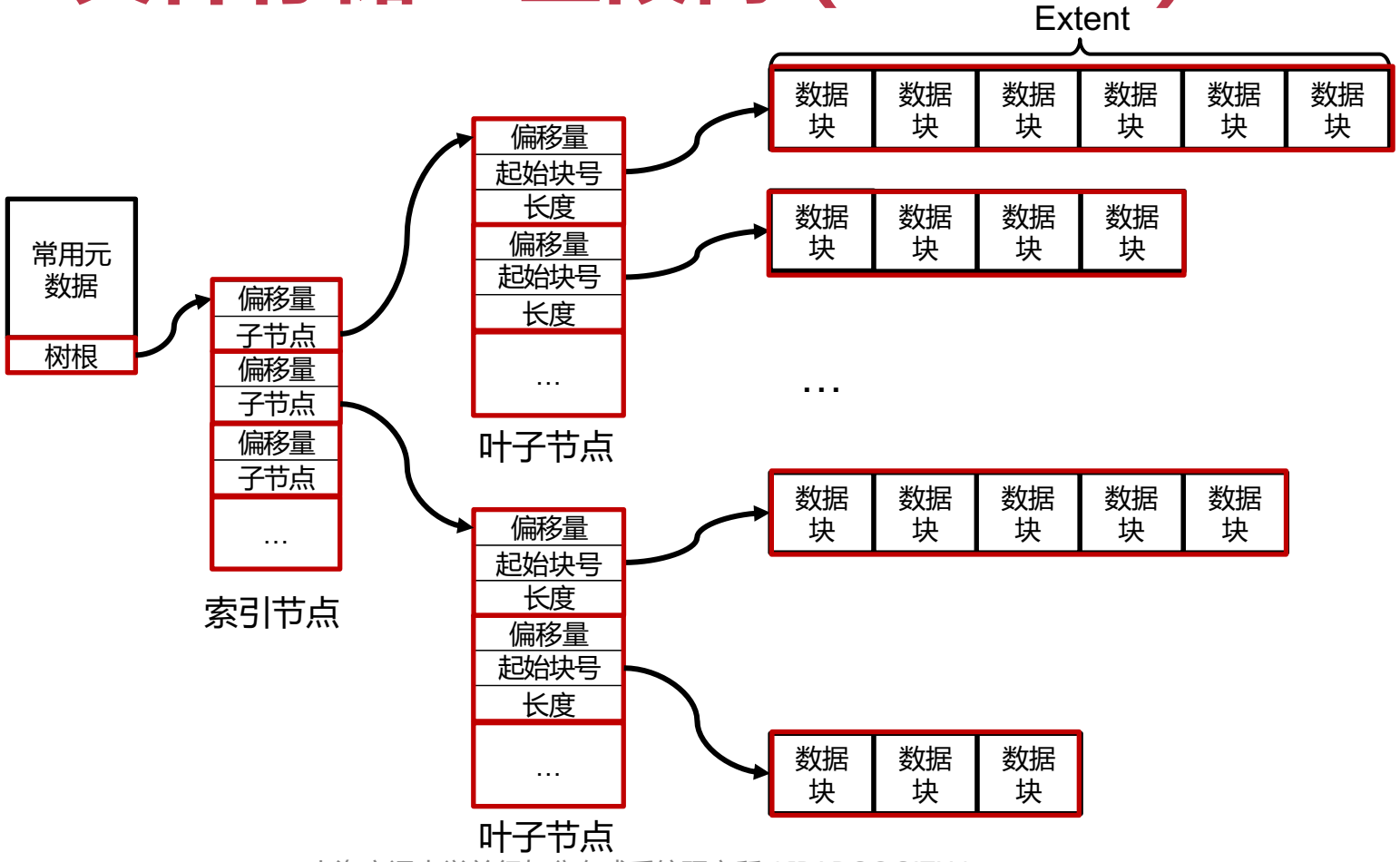
问题：在Ext2的设计中，保存一个1GB的视频文件，文件被拆成多少数据块？需要多少元数据来维护这些数据块？

如果这些数据块物理上连续，只需要保存**起始块地址和长度**即可！

区段（Extent）是由**物理上连续的多个数据块**组成

- 一个区段内的数据可以连续访问，无需按4KB数据块访问
- 可以减少元数据的数量

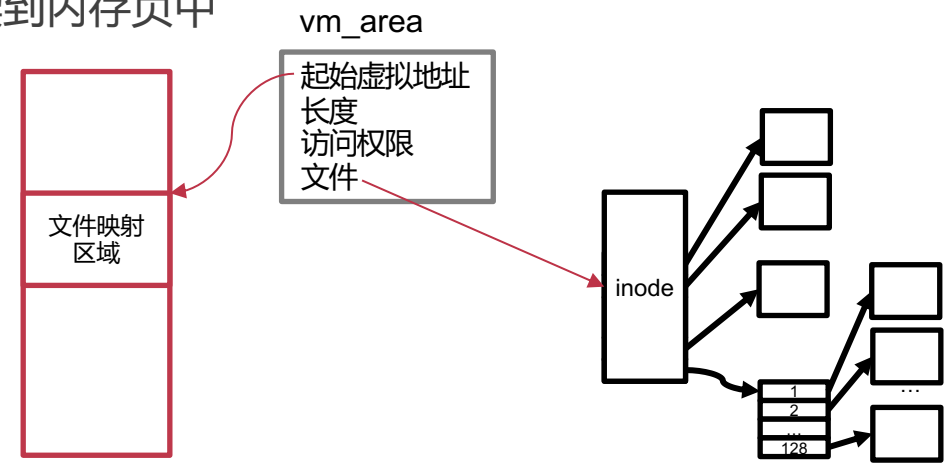
Ext4文件存储 – 区段树（Extent）



文件内存映射：用mmap()来访问文件

- mmap可将文件映射到虚拟内存空间中
  1. mmap时分配虚拟地址，并标记此段虚拟地址与该文件的inode绑定
  2. 访问mmap返回的虚拟地址时，触发缺页中断（page fault）
  3. 缺页中断处理函数，通过虚拟地址，找到该文件的inode
  4. 从磁盘中将inode中对应的数据读到内存页中
  5. 将内存页映射添加到页表中

```
fd = open("/OS/考试", O_RDWR);
addr = mmap(NULL, length, PROT_WRITE,
            MAP_SHARED, fd, 0);
memset(addr, 0, length);
```



文件内存映射的优势

- 对于随机访问，不用频繁lseek
- 减少系统调用次数
- 可以减少数据copy
  - 如拷贝文件，数据无需经过中间buffer
- 访问的局部性更好
- 可以用madvice为内核提供访问提示，提高性能

ChCore中的文件系统

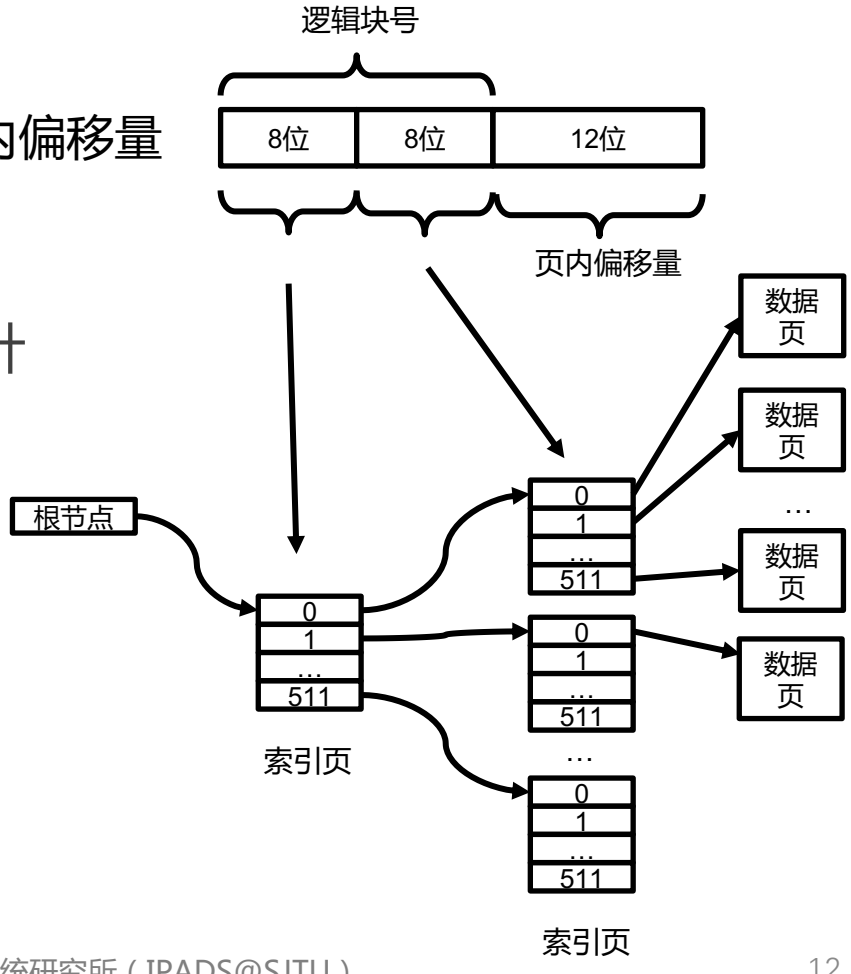
- ChCore中实现了一个内存文件系统
  - 数据存在内存中，无持久化和存储格式

ChCore中的inode

```
struct inode {
    int nlinks;                     /* 链接数 */
    u64 type;                       /* 文件类型 */
    size_t size;                    /* 文件大小 */
    union {
        struct htable dentries;     /* 目录文件的哈希表 */
        struct radix data;          /* 常规文件的基数树 */
    };
};
```

ChCore的常规文件

- 基数树结构
  - 提供<键,值>索引
  - 如前述inode中的多级间接指针
  - 如页表结构
- ChCore中的基数树
  - 保存<逻辑块号，块指针>

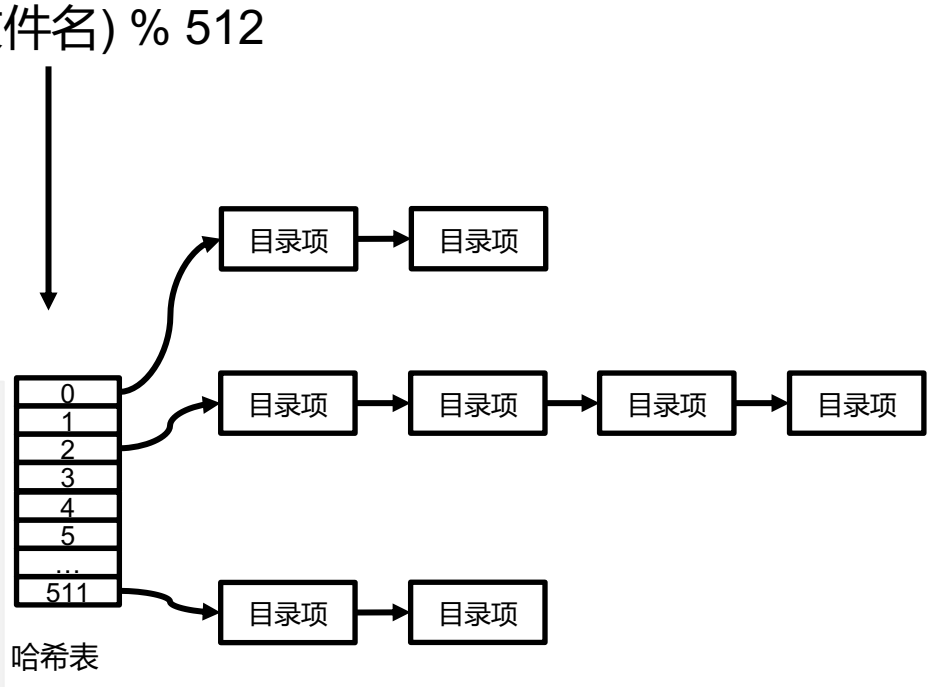


ChCore的目录文件

- 哈希表
  - 保存目录项
  - 使用链表解决冲突

ChCore中的目录项结构

```
struct string {
    char *str;          /* 文件名字符串 */
    size_t len;         /* 文件名长度 */
    u64 hash;           /* 文件名的哈希值 */
};

struct dentry {
    struct string name;  /* 文件名 */
    struct inode *inode; /* 文件对应的inode */
    struct dentry *next; /* 下一个目录项 */
};
```

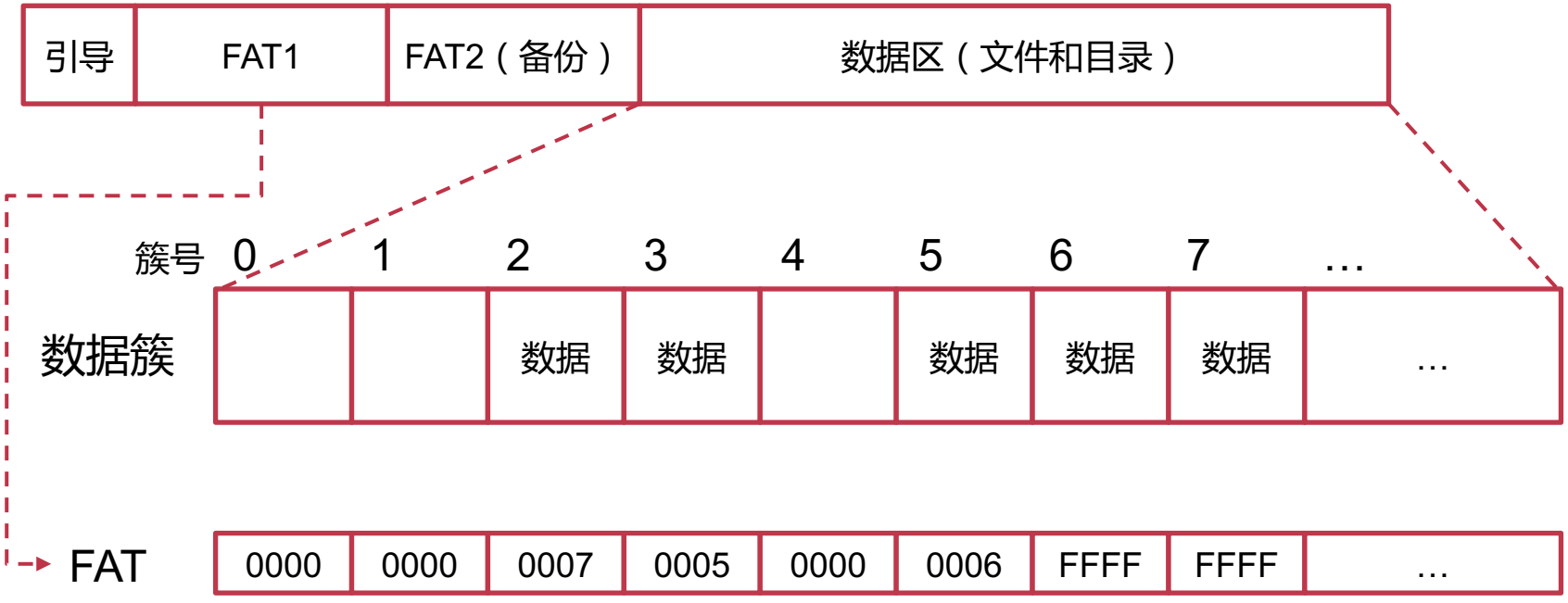


FAT、NTFS

基于TABLE的文件系统

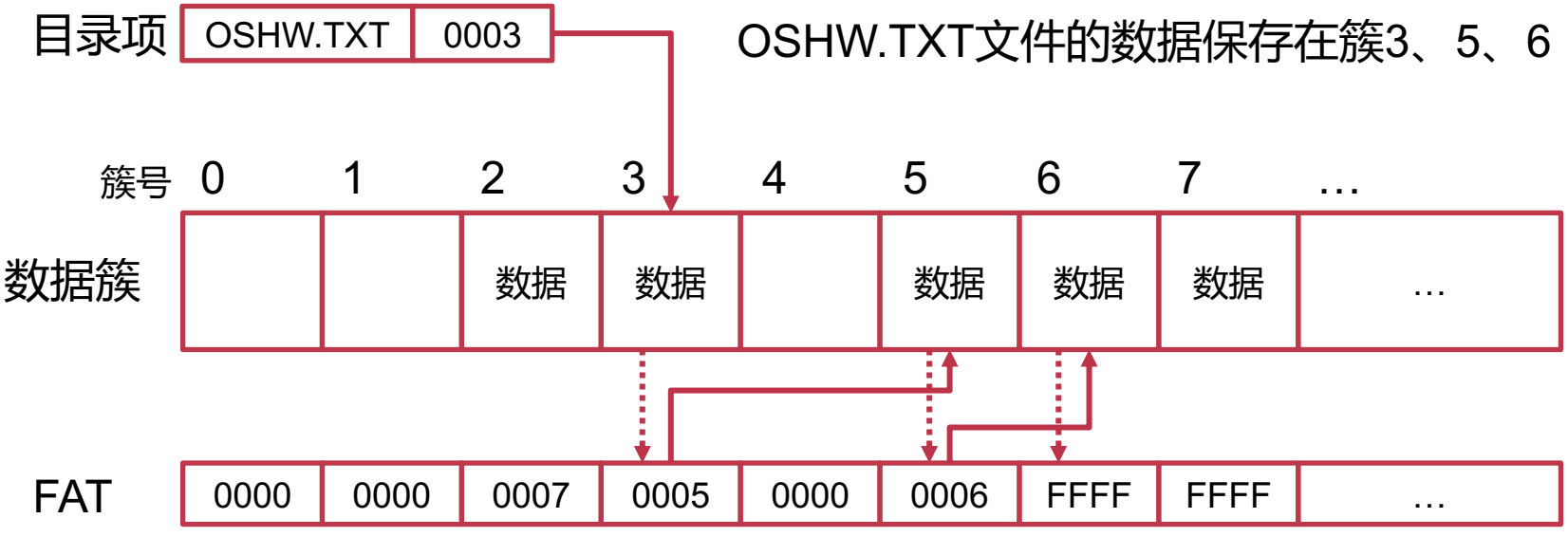
FAT32存储布局

FAT：文件分配表



FAT为每个数据簇增加了一个**next**指针，让簇可以串联在一起

FAT：文件分配表



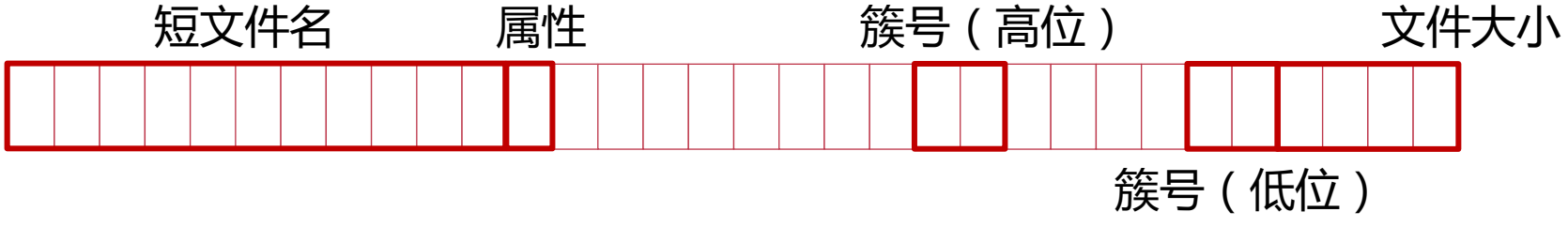
FAT为每个数据簇增加了一个**next**指针，让簇可以串联在一起

FAT32中的目录项

每个目录记录32个字节，作为目录文件的数据保存在数据簇中

四种目录项：

短文件名目录项、长文件名目录项、卷标目录项、"."和".."目录项



思考时间🤔

- 为什么FAT不支持4G以上的文件？
- exFAT如何扩展FAT，使其能支持4G以上的文件？
- 为什么U盘一般用FAT？
- 为什么FAT不支持link？
- 为什么FAT的随机读取文件非常慢？

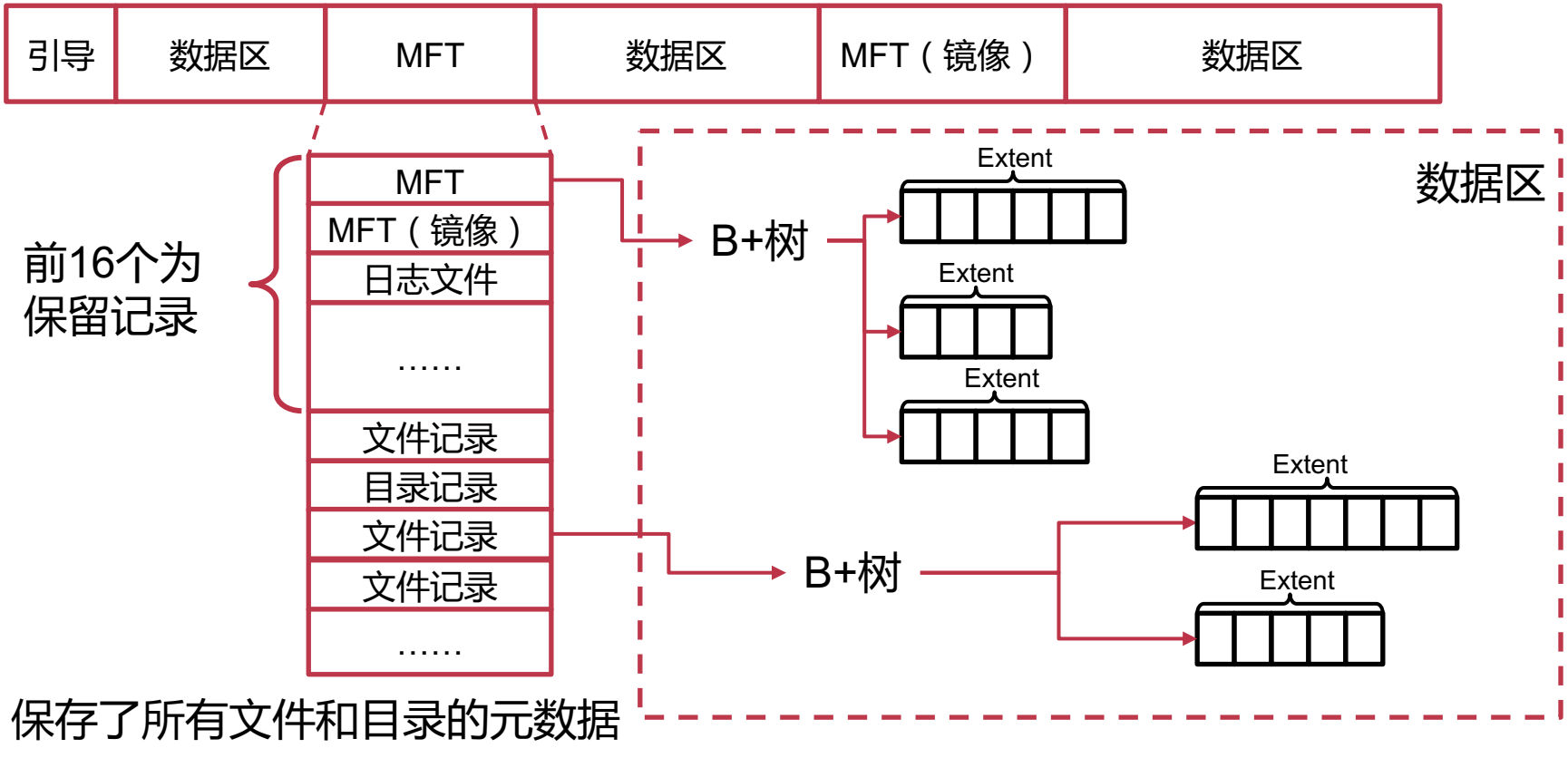
exFAT Highlights

- 与FAT32并不兼容
- 使用Bitmap加快空间分配
- Unicode保存长文件名
- 目录中查找文件时使用哈希对比
- 允许4GB以上文件（新的目录项格式、文件大小用8个字节）

- 使用校验码保证元数据完整性
- 为闪存做优化
  - 可调参数，与存储单元边界对齐
  - OEM域
  - 没有日志

NTFS存储布局

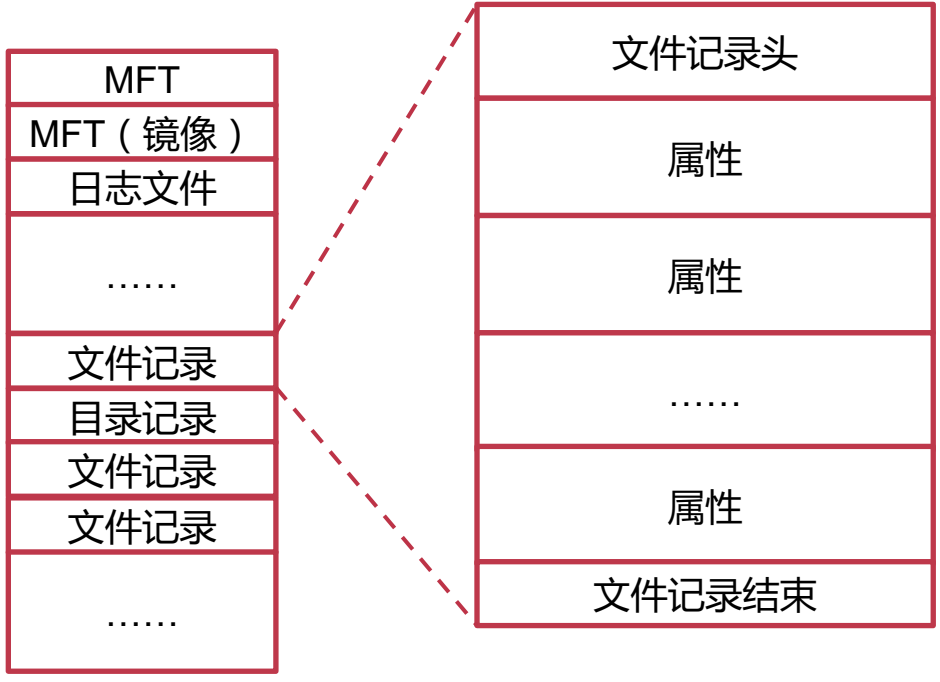
NTFS主文件表MFT



主文件表保留文件

序号	文件名	说明
0	$MFT	主文件表
1	$MFTMirr	主文件表镜像
2	$LogFile	日志文件
3	$Volume	卷文件
4	$AttrDef	属性定义列表
5	$Root	根目录
6	$Bitmap	位图文件
7	$Boot	引导文件
8	$BadClus	坏簇文件
9	$Secure	安全文件
10	$UpCase	大写表
11	$Extend	扩展元数据目录
12	$Extend\$Reparse	重解析点文件
13	$Extend\$UsnJrnl	变更日志文件
14	$Extend\$Quota	配额管理文件
15	$Extend\$ObjId	对象ID文件

主文件表记录



常用属性包括：

- 文件标准元数据（大小、时间等）
- 文件名
- 数据
- 索引根

NTFS数据保存位置和目录项

- 非常驻文件（大文件/目录）
  - 数据区的B+树和区段
- 常驻文件（小文件/目录）
  - 大小不超过MFT记录的最大值（1KB）
  - 内嵌在MFT中保存（在数据属性中）
- 目录项
  - 包含文件名、文件ID（在MFT中的序号）

缓存、多文件系统、FUSE

虚拟文件系统（VFS）

如何在一个系统中同时支持多个文件系统？

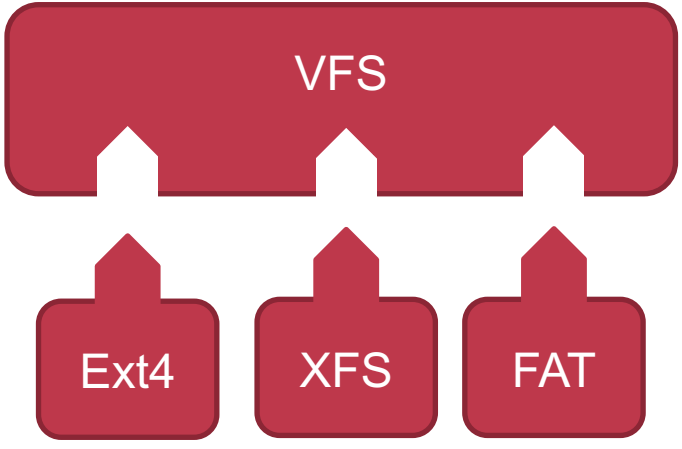
- **计算机中的异构文件系统**
  - Linux和Windows双启动，两个分区有各自的文件系统
  - Mac用APFS，U盘一般用FAT/exFAT，移动硬盘用NTFS

- **如何对用户屏蔽文件系统的异构性？**
  - VFS：Virtual File System
  - 中间层，对上提供POSIX API，对下对接不同的文件系统驱动

Linux中的虚拟文件系统VFS

Linux的VFS定义了一些系列接口

具体的文件系统实现这些接口



如在读取一个inode的文件时

- VFS先找到该inode所属文件系统
- 再调用该文件系统的读取接口

Windows的类似机制

- Installable File System

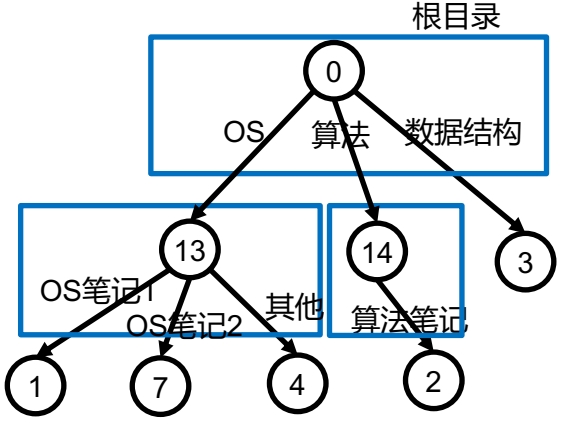
虚拟文件系统 VFS

操作系统同时使用多个文件系统

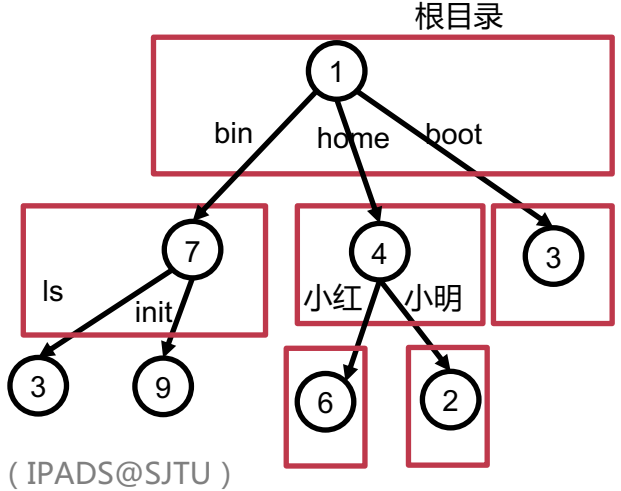
虚拟文件系统提供统一的管理，对应用程序提供统一的视图和抽象

问题：每个文件系统都有自己的根节点，它们如何配合在一起？

文件系统1



文件系统2

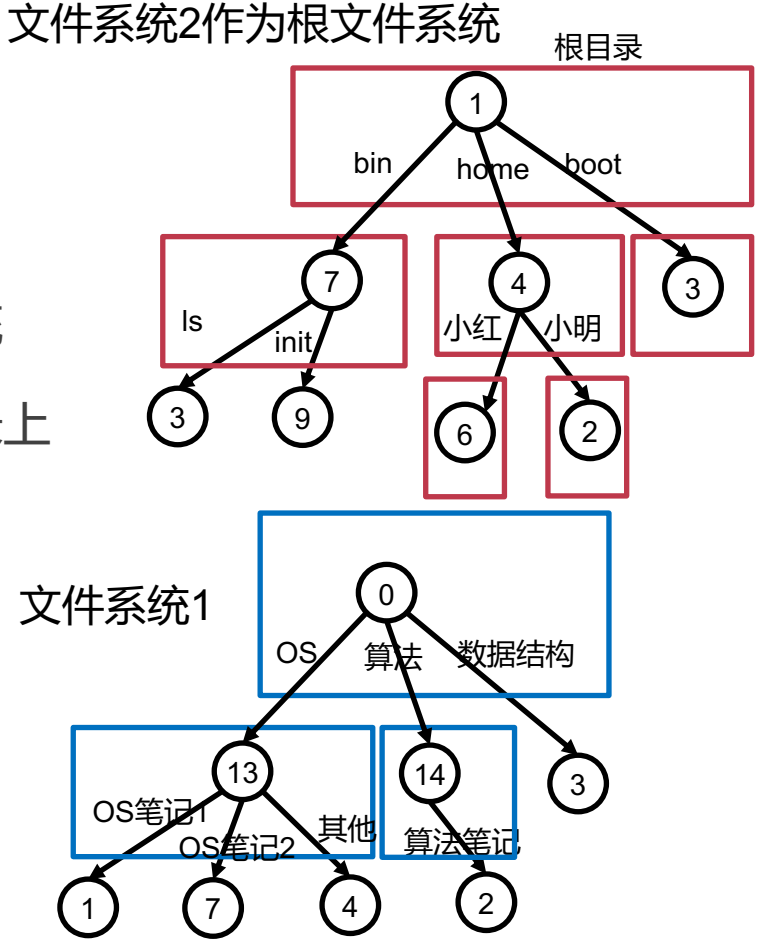


虚拟文件系统 VFS

VFS维护一个统一的文件系统树

操作系统内核启动时会挂载一个根文件系统

其他文件系统可以**挂载**在文件系统树的目录上

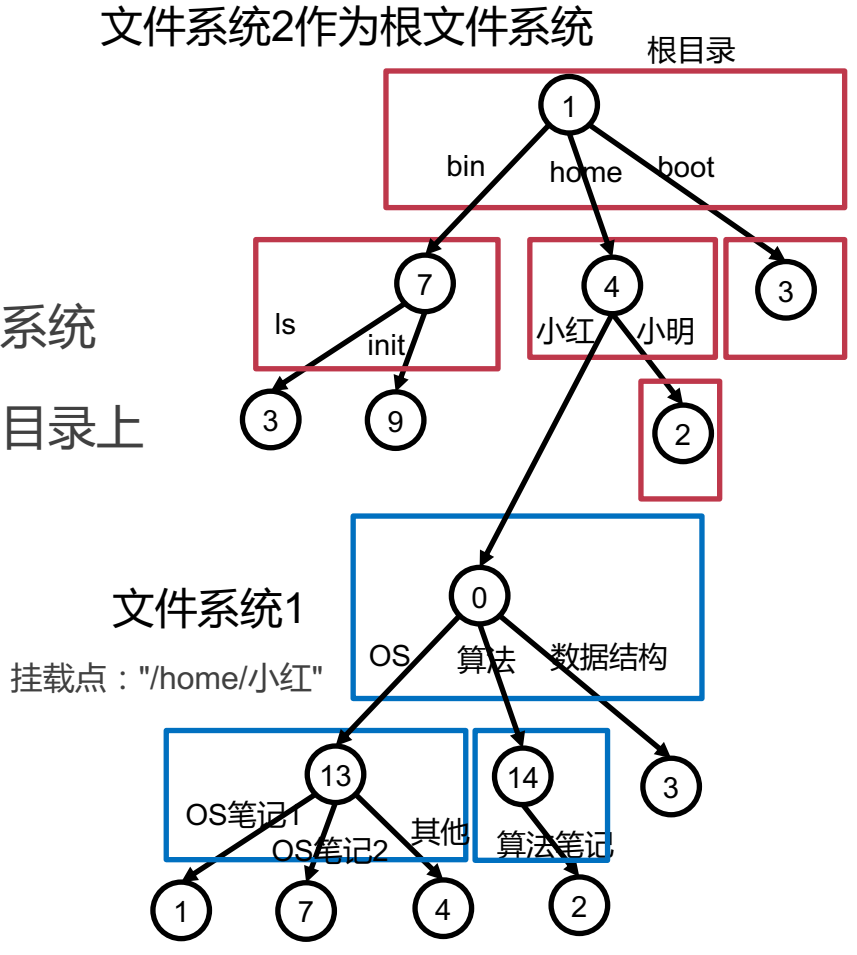


虚拟文件系统 VFS

VFS维护一个统一的文件系统树

操作系统内核启动时会挂载一个根文件系统

其他文件系统可以**挂载**在文件系统树的目录上

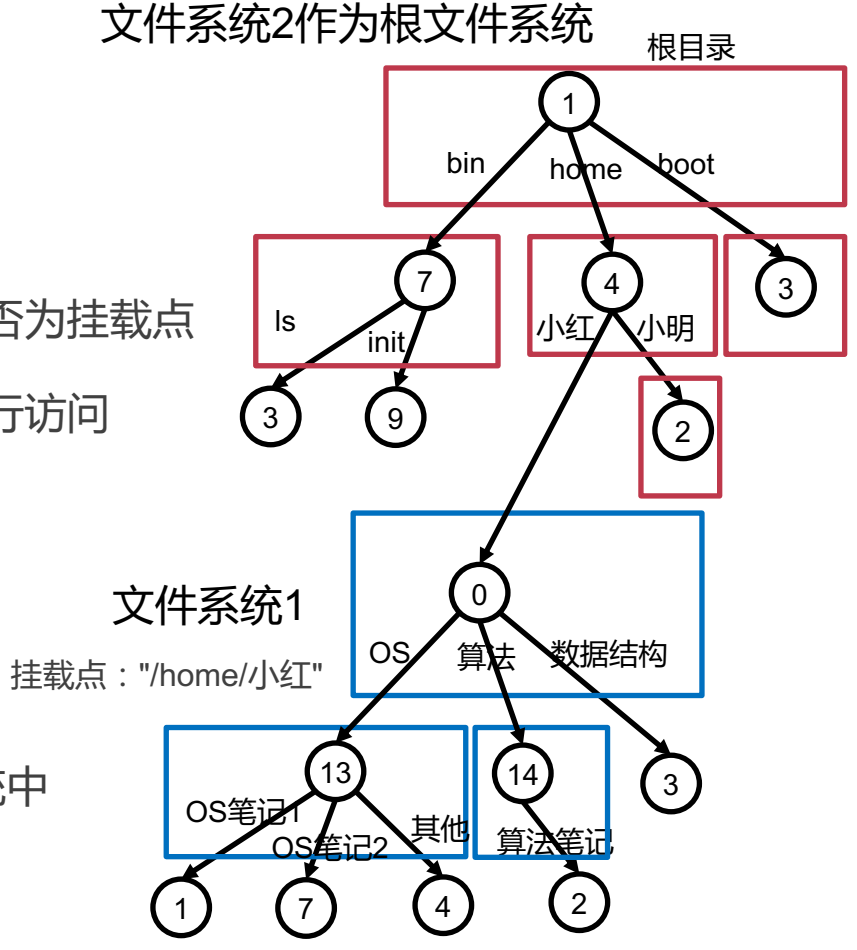


虚拟文件系统 VFS

VFS维护所有的挂载信息

- 查找文件时的每一步，检查当前目录是否为挂载点
- 若是，则使用被挂载的文件系统继续进行访问

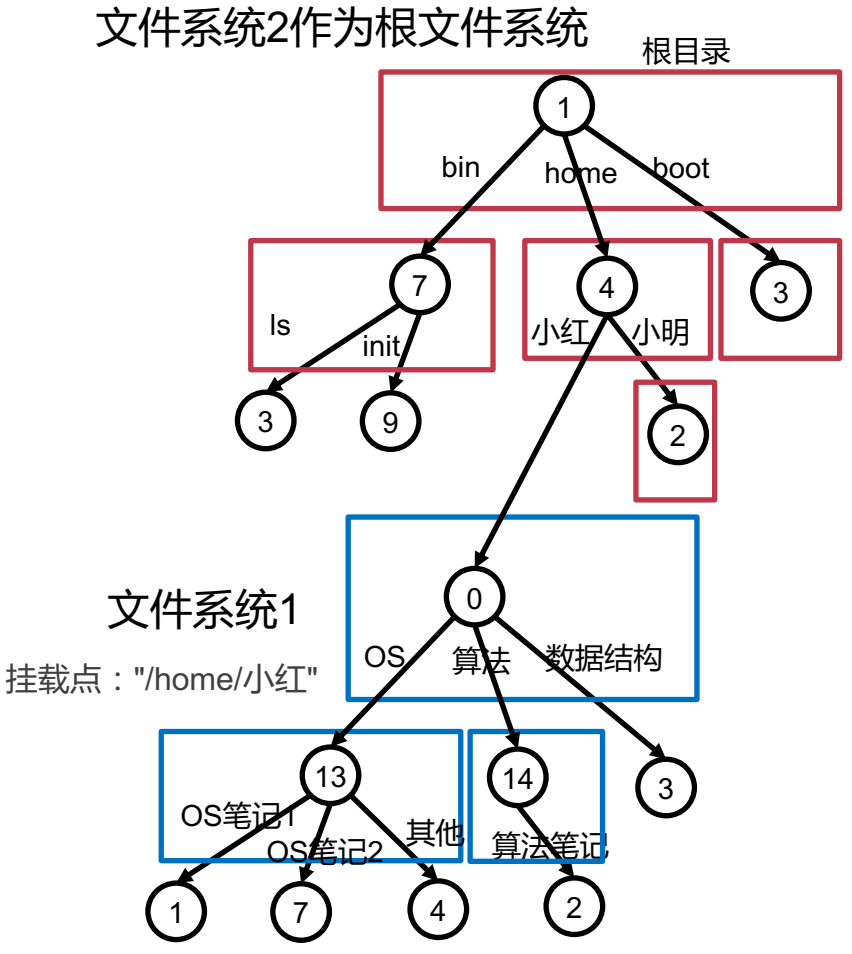
文件系统1中的"/OS/OS笔记1"通过操作系统中的"/home/小红/OS/OS笔记1"访问



虚拟文件系统 VFS

挂载在逻辑上覆盖挂载点原有的结构

- 无法访问文件系统2中的"/home/小红"

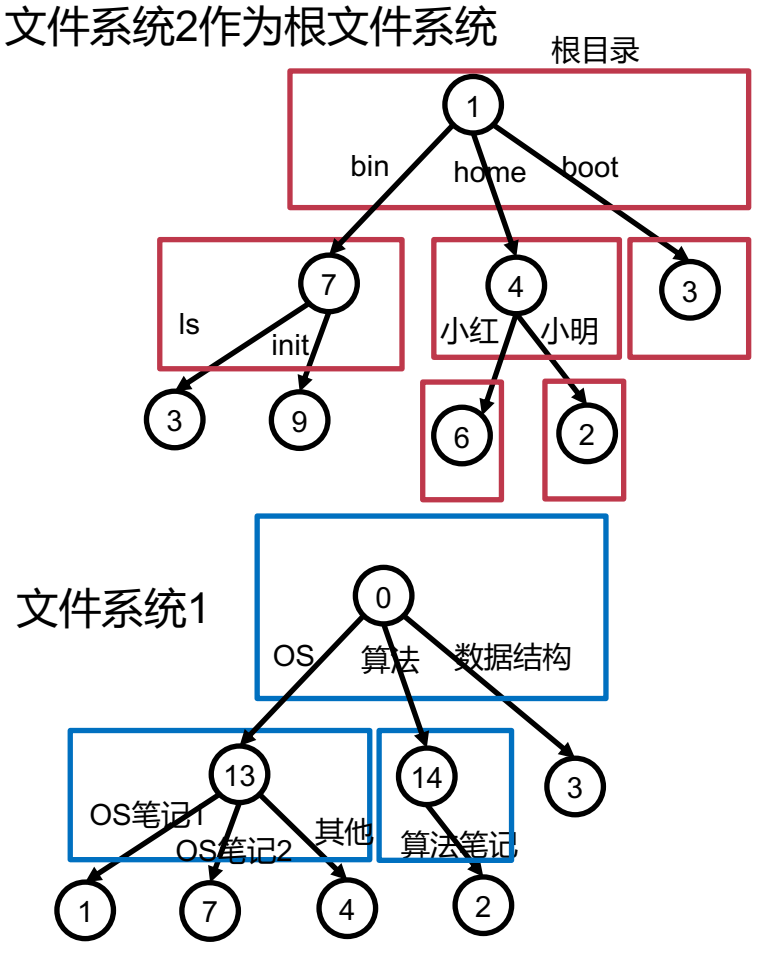


虚拟文件系统 VFS

挂载在逻辑上覆盖挂载点原有的结构

- 无法访问文件系统2中的"/home/小红"

挂载点下的数据在卸载后依然可以访问



Linux中的虚拟文件系统VFS

Linux的VFS定义的一些inode上的操作接口

```
struct inode_operations {
  struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
  ...
  int (*create) (struct inode *,struct dentry *, umode_t, bool);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,umode_t);
  int (*rmdir) (struct inode *,struct dentry *);
  ...
};
```

Ext2对这些接口的实现

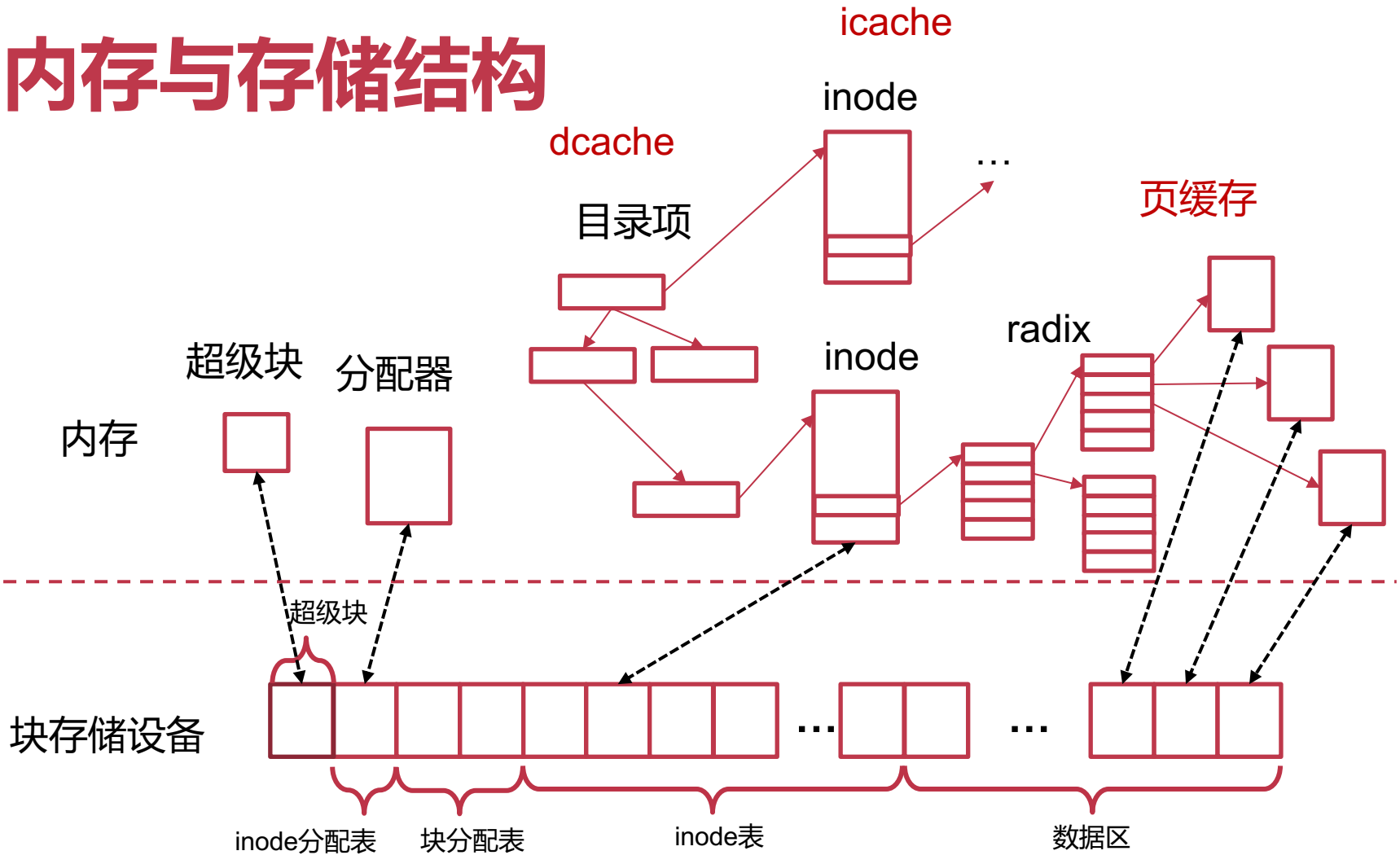
```
const struct inode_operations ext2_dir_inode_operations = {
  .create   = ext2_create,
  .lookup   = ext2_lookup,
  .link     = ext2_link,
  .unlink   = ext2_unlink,
  .symlink  = ext2_symlink,
  .mkdir    = ext2_mkdir,
  .rmdir    = ext2_rmdir,
  ...
};
```

问题

- **FAT没有inode，如何挂载到VFS？**
  - VFS层对上提供的接口，每个文件都有一个inode
  - FAT的inode从哪里来？

- **FAT的驱动需要提供inode**
  - 磁盘上的FAT并没有inode：硬盘上的数据结构
  - 内存中的VFS需要inode：只在内存中的数据结构

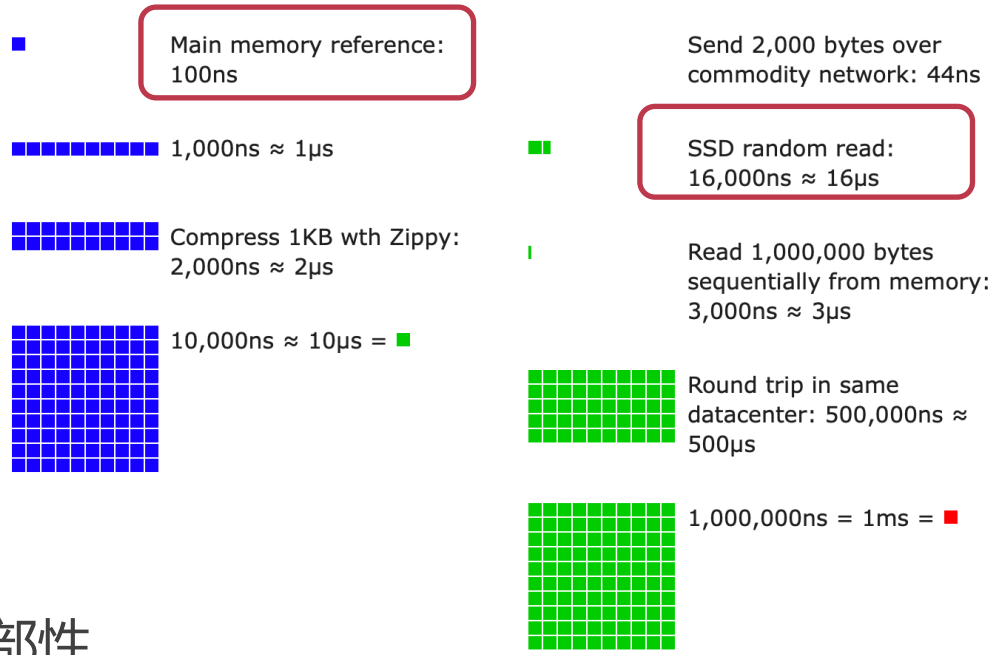
内存与存储结构


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

存储结构与缓存

页缓存（Page Cache）

- 存储访问非常耗时



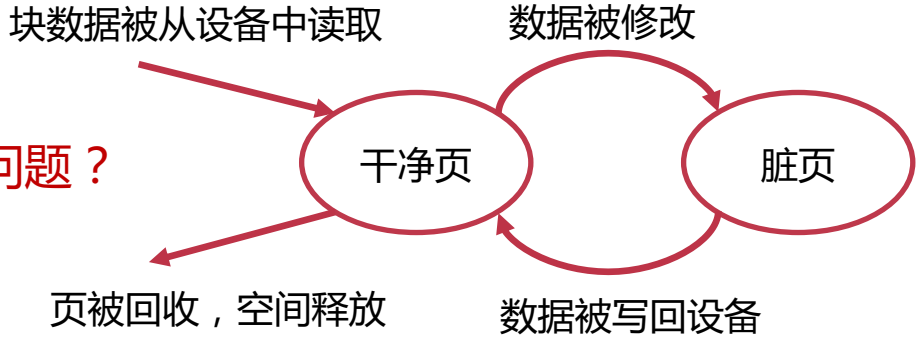
- 文件访问具有时间局部性
  - 一些目录/文件的数据块会被频繁的读取或写入

https://colin-scott.github.io/personal_website/research/interactive_latency.html

页缓存（Page Cache）

- 通过缓存提升文件系统性能
  - 在一个块被读入内存并被访问完成后，并不立即回收内存
    - 将块数据暂时缓存在内存中，下一次被访问时可以避免磁盘读取
  - 在一个块被修改后，并不立即将其写回设备
    - 将块数据暂时留在内存中，此后对于该数据块的写可直接修改在此内存中
  - 定期或在用户要求时才将数据写回设备

问题：数据不及时写回，会造成什么问题？



页缓存之外

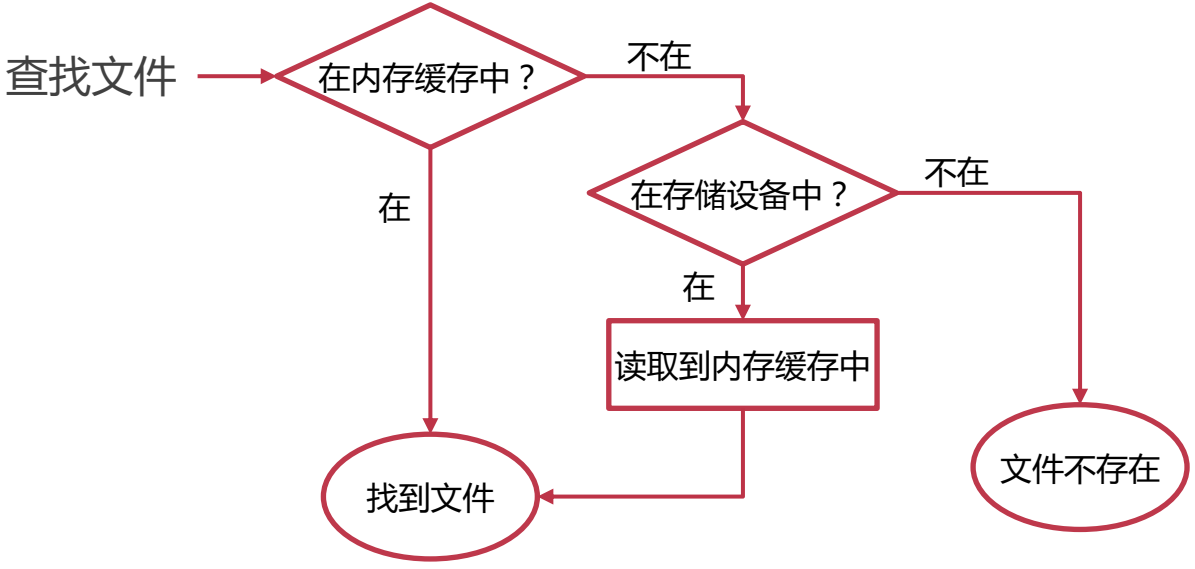
存储中的每个数据结构，在内存中均有对应的结构

- 存储的数据页：页缓存中的内存页
- 存储中的inode：icache中的inode
- 存储中的目录项：dcache中的目录项
- 存储中的超级块：内存中的超级块结构
- 存储中的分配表：内存中的分配器

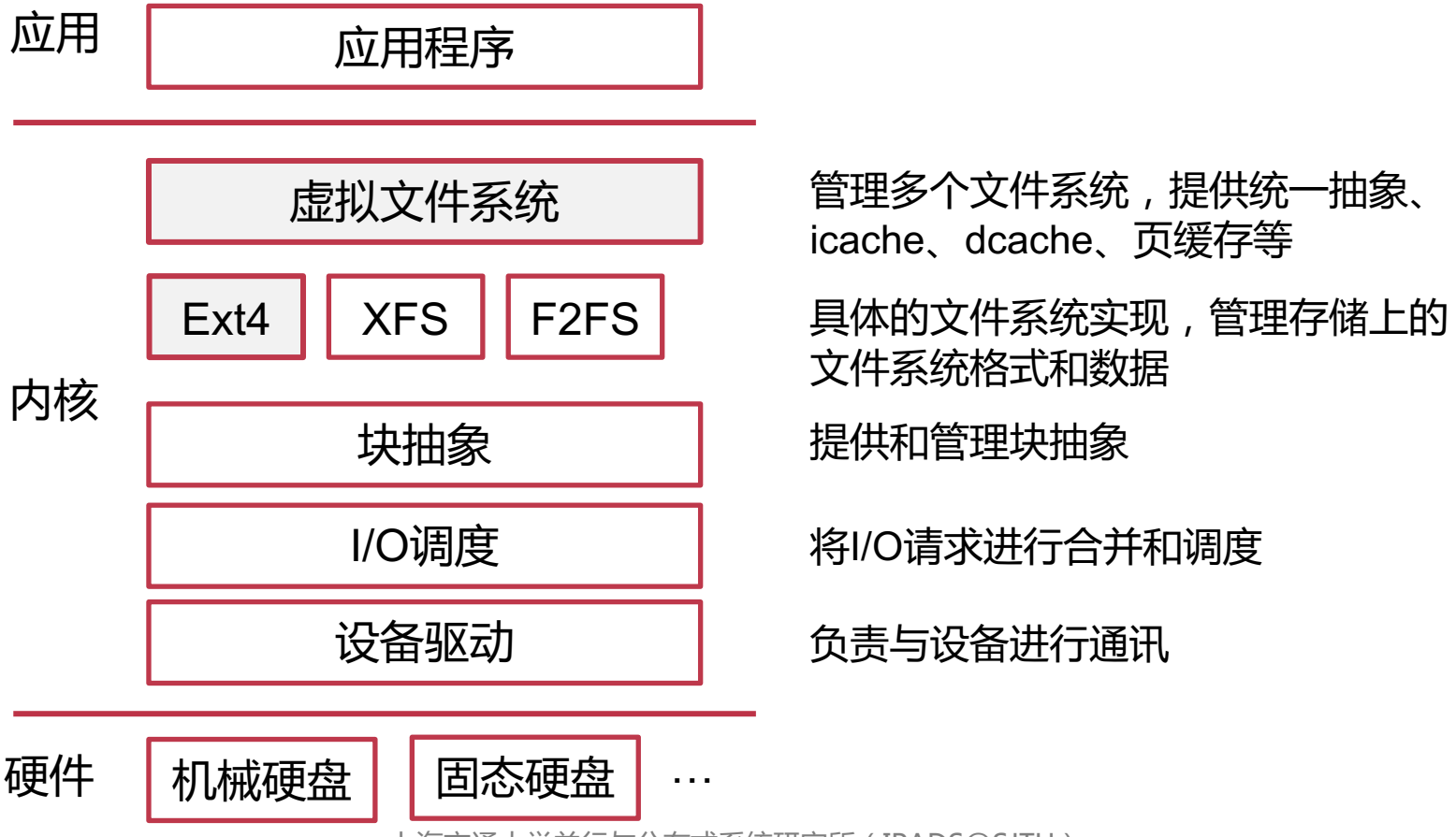
缓存情况下的文件查找

由于内存大小限制，内存中缓存的数据是存储中数据的子集

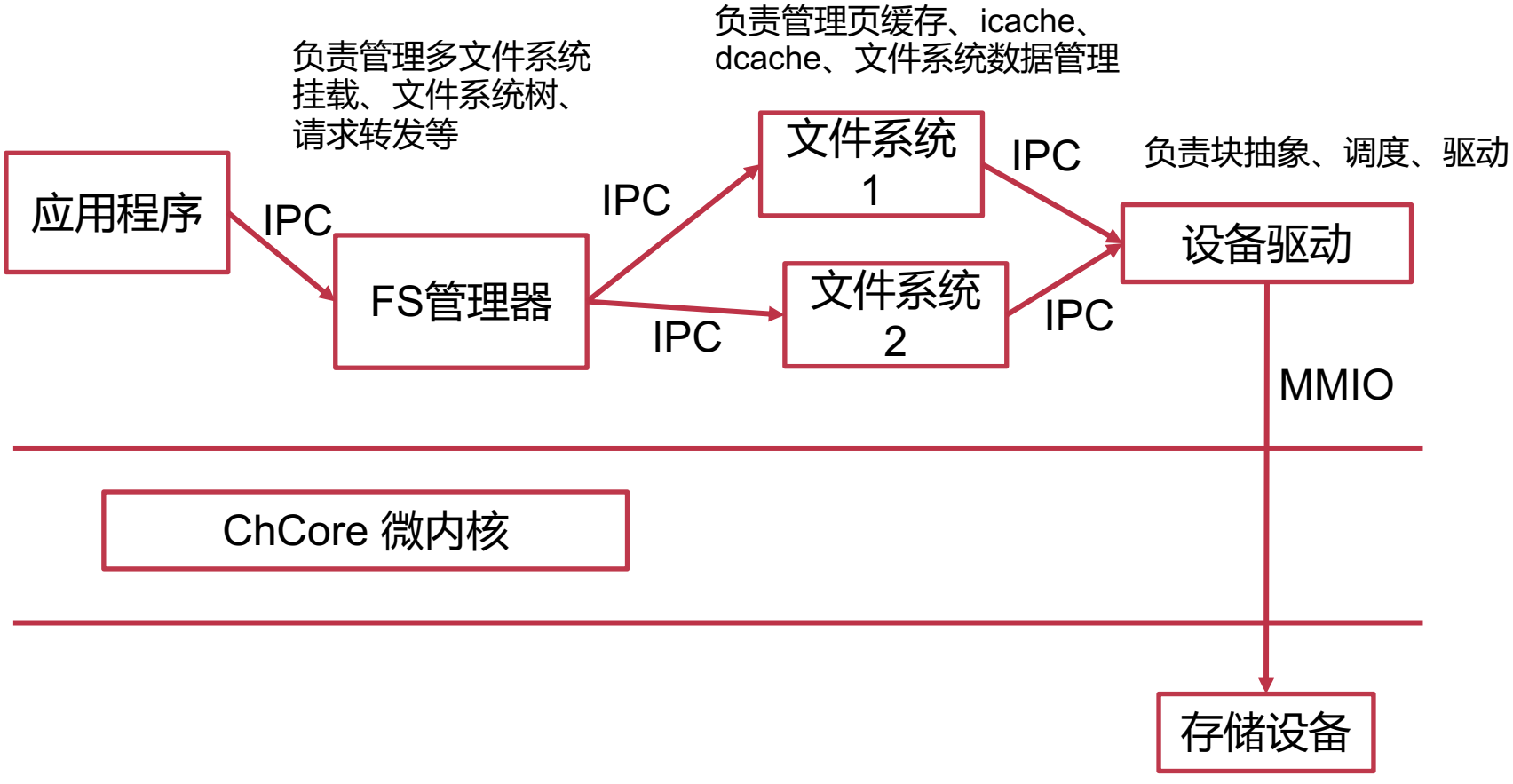
当要访问的数据不在内存中时，会从存储中读取并构造内存中相应的对象



宏内核(Linux)中的存储栈



ChCore中的文件与存储结构



文件系统高级功能

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

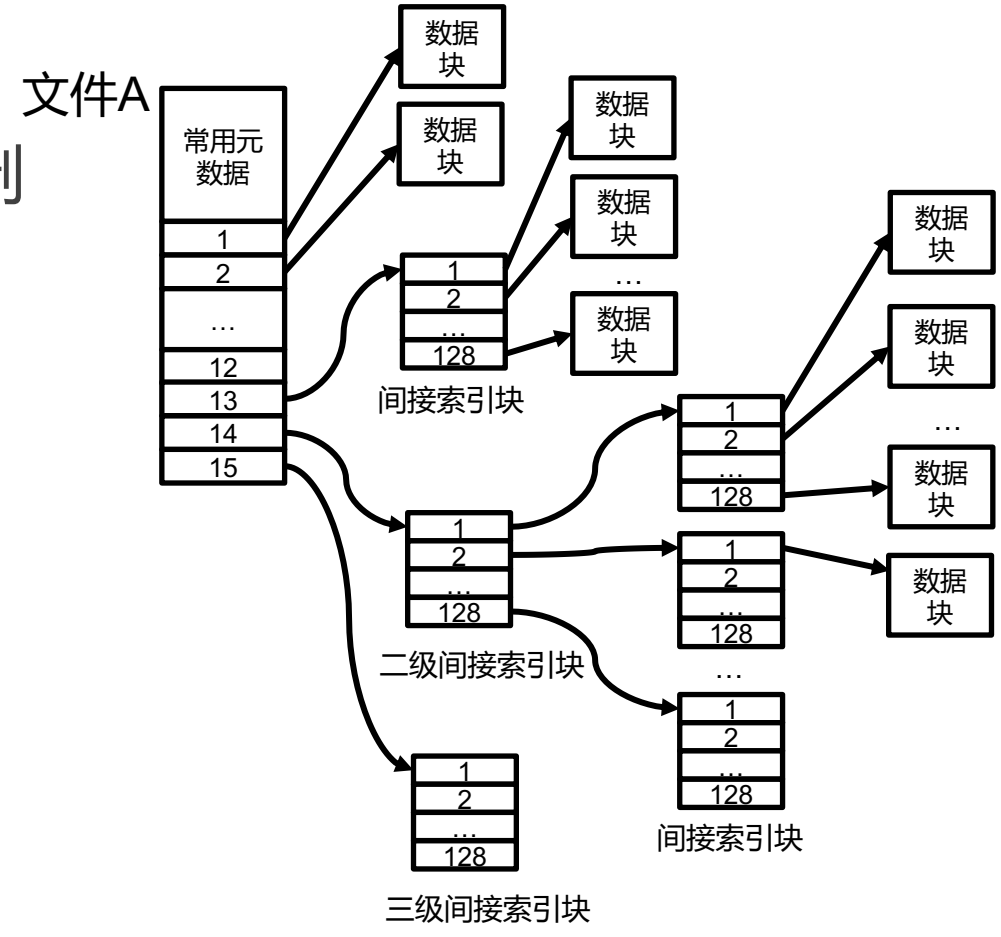
```
$ cp A B

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

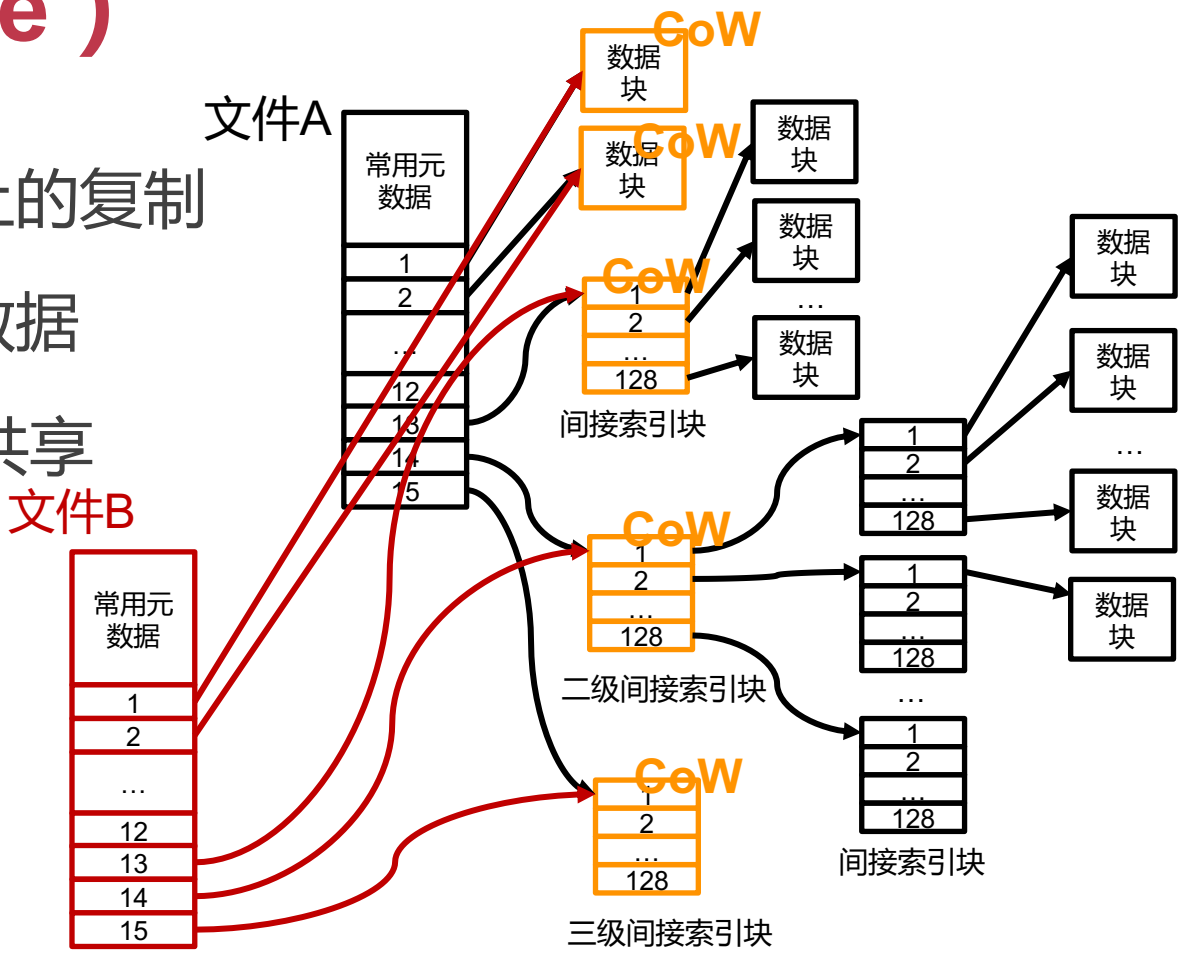
克隆

- 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）

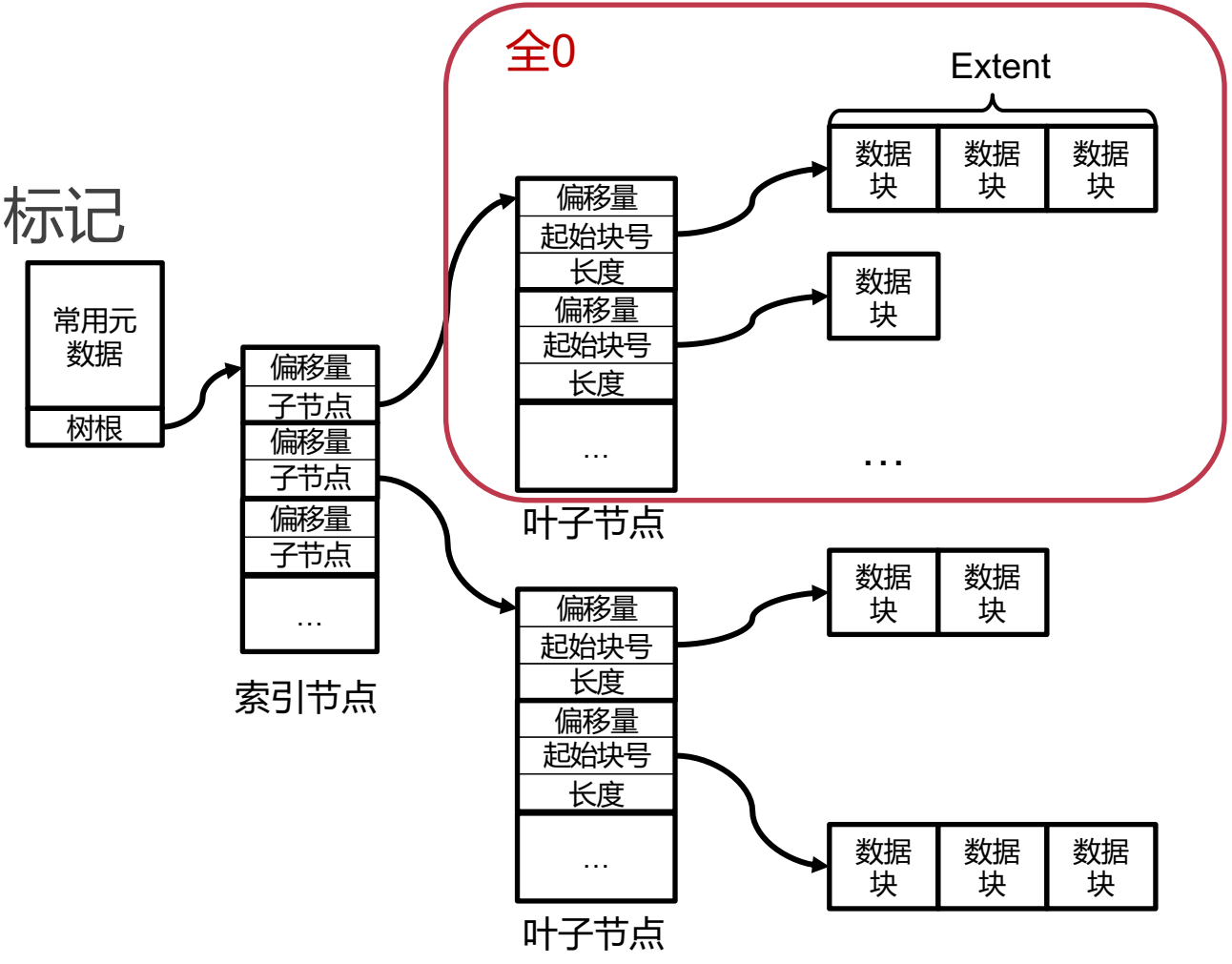
- 同样使用CoW
- 对于基于inode表的文件系统
  - 将inode表拷贝一份作为快照保存
  - 标记已用数据区为CoW
- 对于树状结构的文件系统
  - 将树根拷贝一份作为快照保存
  - 树根以下的节点标记为CoW

稀疏文件

- 一个文件大部分数据为0，则为稀疏文件
  - 如虚拟机镜像文件
- 稀疏文件中大量的0数据，白白消耗空间

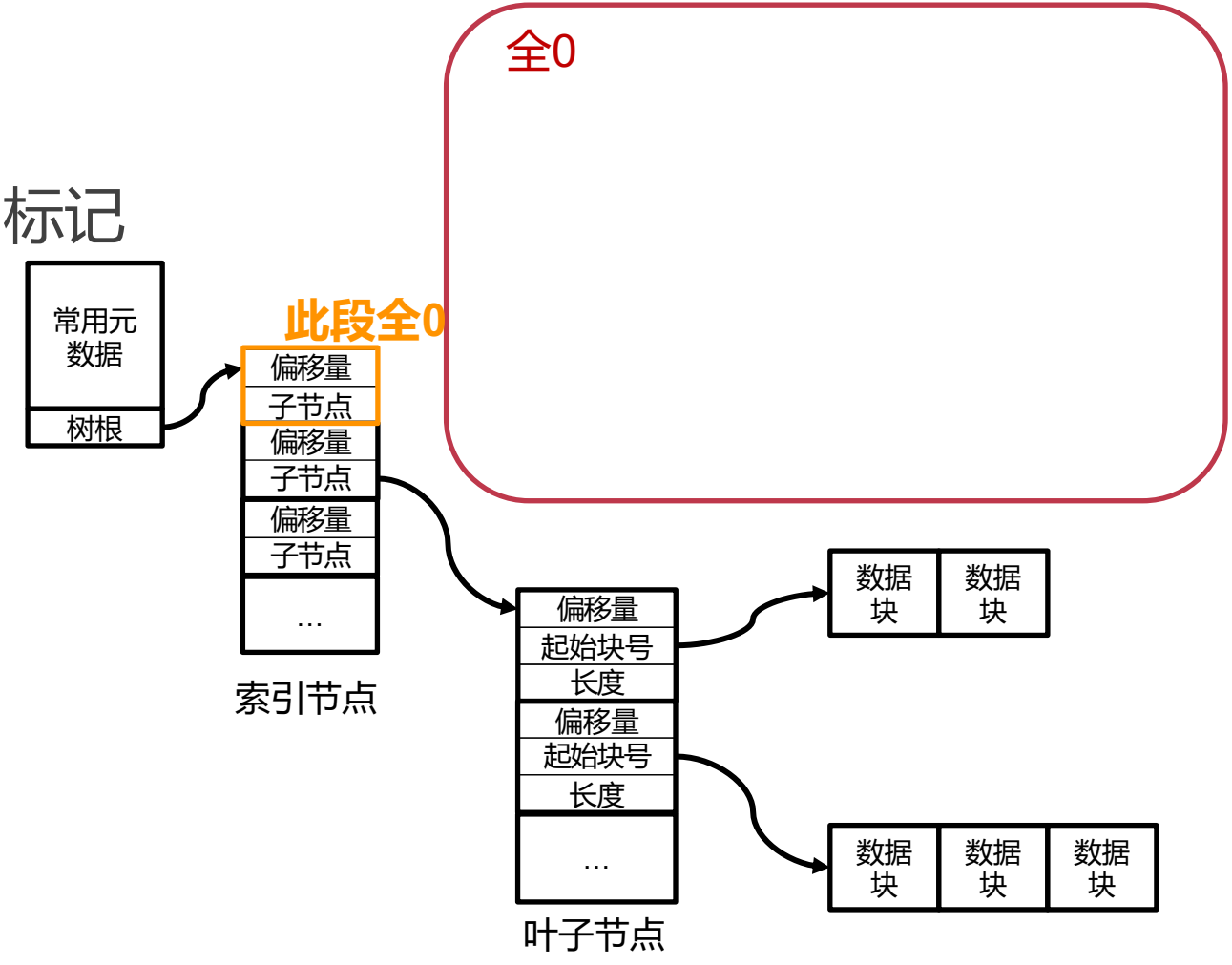
稀疏文件

- 在索引中增加标记
- 删除全0块



稀疏文件

- 在索引中增加标记
- 删除全0块



文件系统的一些其他高级功能

- 加密
- 压缩
- 去重
- 数据和元数据校验
- 配额管理（QoS）
- 软件RAID
- 多设备管理
- 子卷
- 事务（Transaction）

文件系统的多种形式

GIT：内容寻址文件系统

- 表面上GIT是一个版本控制软件
- 但实际上GIT是一个内容寻址的文件系统！
- 其核心是一个键值存储
  - 值：加入GIT的数据
  - 键：通过数据内容算出的40个字符SHA-1校验和
  - 前2个字符作为子目录名，后38个字符作为文件名
  - 所有对象均保存在`.git/objects`目录中（文件内容会被压缩）
- 是一个“文件系统之上的文件系统”

GIT对象与文件系统

- **BLOB对象**：对应文件系统中的文件

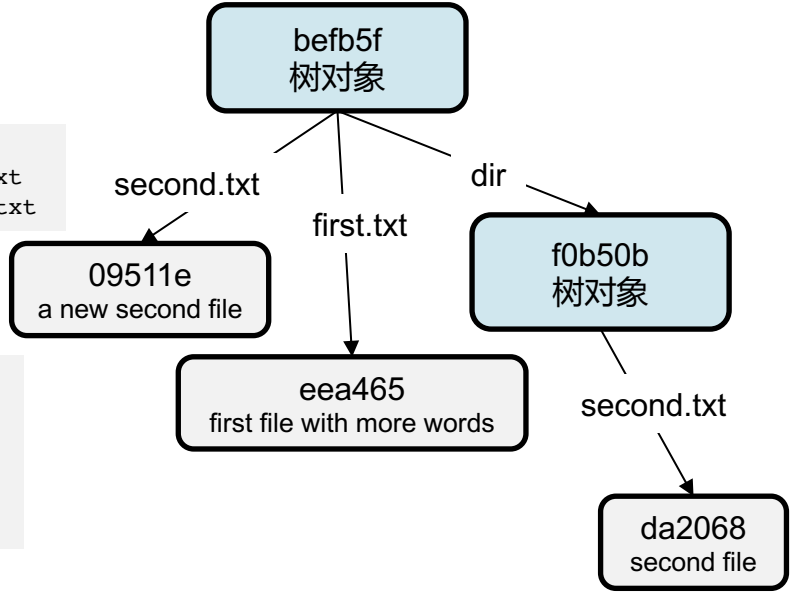
```
first file with more words
```

- **树对象**：对应文件系统中的目录

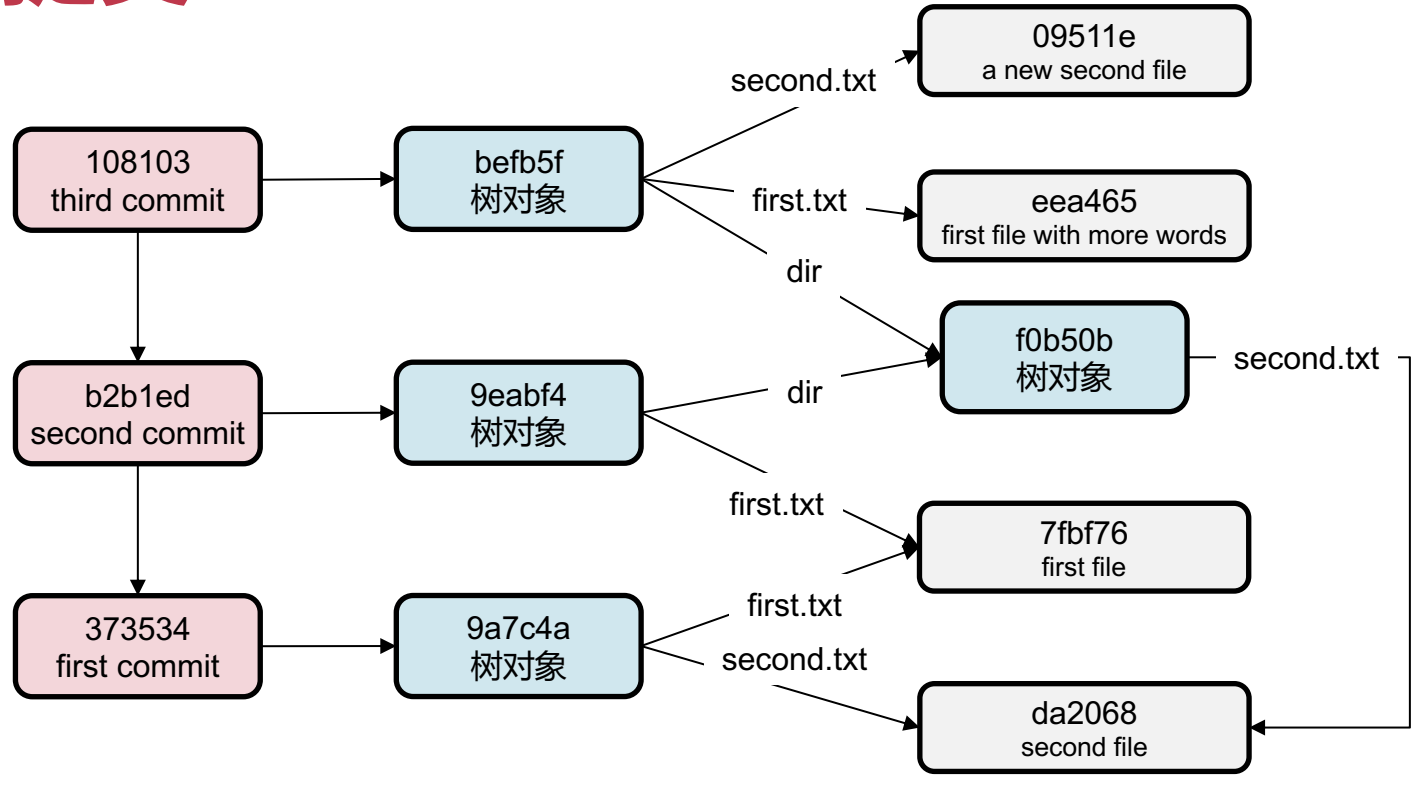
```
040000 tree f0b50bef52478d42d68ff21914c430d8148f23cd  dir
100644 blob eea465ab73dfb5cd24379efd61da949b3b5138ea  first.txt
100644 blob 09511ef0b04fb20dd32f3cb090f9c6de49cd2627  second.txt
```

- **提交对象**

```
tree befb5f375306393e542026391d95104579da48ce
parent b2b1eda3c44d0a2cb10ad7f522ae78f9ded7e95e
author IPADS <ipads@ipads.se.sjtu.edu.cn> 1587414317 +0800
committer IPADS <ipads@ipads.se.sjtu.edu.cn> 1587414317 +0800

third commit
```



GIT的提交



小问题：在执行 git checkout b2b1ed -- first.txt 时都发生了什么？

SQLite：文件系统的竞争者

SQLite

Small. Fast. Reliable.
Choose any three.

Home | About | Documentation | Download | License | Support | Purchase | Search

35% Faster Than The Filesystem

▶ Table Of Contents

1. Summary

SQLite reads and writes small blobs (for example, thumbnail images) 35% faster[1] than the same blobs can be read from or written to individual files on disk using fread() or fwrite().

Furthermore, a single SQLite database holding 10-kilobyte blobs uses about 20% less disk space than storing the blobs in individual files.

The performance difference arises (we believe) because when working from an SQLite database, the open() and close() system calls are invoked only once, whereas open() and close() are invoked once for each blob when using blobs stored in individual files. It appears that the overhead of

https://www.sqlite.org/fasterthanfs.html

SQLite：文件系统的竞争者

- 表面上SQLite是一个数据库
- 但实际上SQLite也可以是一个文件系统！
- 其核心还是一个数据库
  - 在关系型数据库的表中，记录文件名和BLOB类型文件数据
  - 通过查找文件名，获取对应文件数据
  - 存储大量小文件
- 文件系统里的文件里的文件系统里的文件

SQLite引发的思考🤔

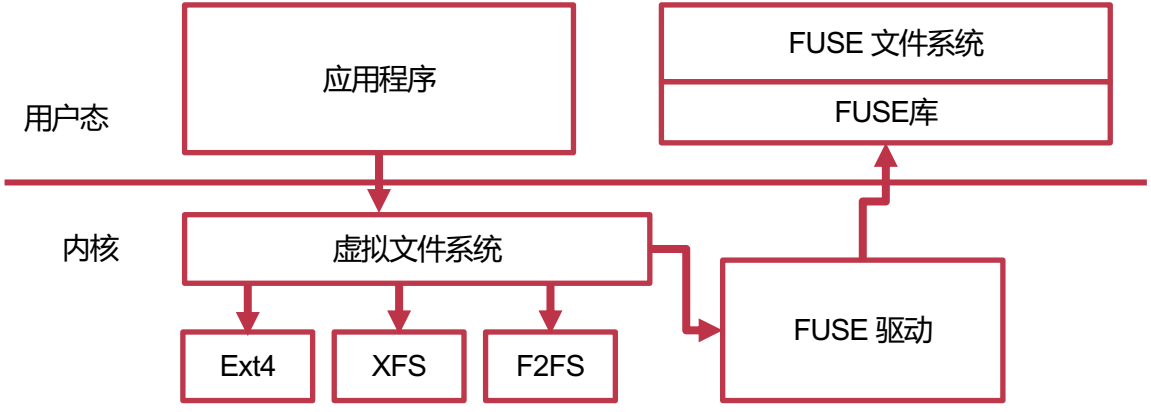
- 对于小文件，为何一般文件系统不如SQLite效率高？
- 文件系统如何针对小文件进行改进？
- 还有哪些针对小文件特殊处理的场景？

FUSE：用户态文件系统框架

FUSE用户态文件系统框架

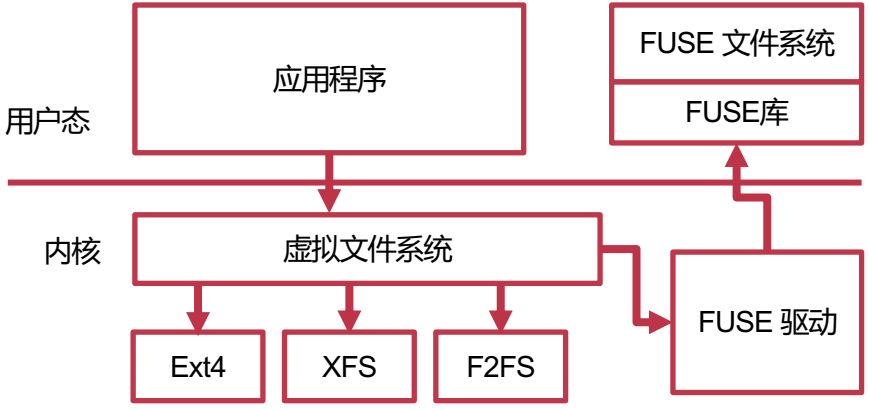
为什么要用户态文件系统？

- 快速试验文件系统新设计
- 大量第三方库可以使用
- 方便调试
- 无需担心把内核搞崩溃
- 实现新功能



FUSE基本流程

1. FUSE文件系统向FUSE驱动注册（挂载）
2. 应用程序发起文件请求
3. 根据挂载点，VFS将请求转发给FUSE驱动
4. FUSE驱动通过中断、共享内存等方式将请求发给FUSE文件系统
5. FUSE文件系统处理请求
6. FUSE文件系统通知FUSE驱动请求结果
7. FUSE驱动通过VFS返回结果给应用程序



问题：从这个流程中可以看出FUSE有什么问题？

FUSE API

- **底层API**
  - 直接与内核交互
  - 需要负责处理inode和查找等操作
  - 需要处理内核版本等差异
- **高层API**
  - 构建于底层API之上
  - 以路径名为参数
  - 无需关注inode、路径和查找

FUSE底层API

```
static void hello_ll_lookup(fuse_req_t req, fuse_ino_t parent, const char *name)
{
  struct fuse_entry_param e;  e.ino = 2;
  hello_stat(e.ino, &e.attr);
  fuse_reply_entry(req, &e);
} // lookup：根据父目录ino和文件名，返回查找结果
static void hello_ll_read(fuse_req_t req, fuse_ino_t ino, size_t size,
                          off_t off, struct fuse_file_info *fi)
{
  assert(ino == 2);
  reply_buf_limited(req, hello_str, strlen(hello_str), off, size);
} // read：根据ino，返回请求

static const struct fuse_lowlevel_ops hello_ll_oper = {
  .lookup = hello_ll_lookup,
  .getattr = hello_ll_getattr,
  .readdir = hello_ll_readdir,
  .open = hello_ll_open,
  .read = hello_ll_read,
}; // 提供函数指针作为回调函数

int main() {
  ...
  se = fuse_session_new(&args, &hello_ll_oper, sizeof(hello_ll_oper), NULL); // 建立连接
  fuse_session_mount(se, opts.mountpoint); // 挂载
  ret = fuse_session_loop(se); // 等待请求和回调
}
```

http://libfuse.github.io/doxygen/example_2hello__ll_8c.html

FUSE高层API

```
static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                      struct fuse_file_info *fi)
{
  const char *content = "Hello World!\n";
  memcpy(buf, content + offset, MIN(size, strlen(content) – offset));
  return size;
} // read：根据path，在buf中填入数据，返回写入字节数

static const struct fuse_operations hello_oper = {
  .init = hello_init,
  .getattr = hello_getattr,
  .readdir = hello_readdir,
  .open = hello_open,
  .read = hello_read,
}; // 提供函数指针作为回调函数

int main() {
  ...
  ret = fuse_main(args.argc, args.argv, &hello_oper, NULL); // 挂载、注册并等待请求和回调
  ...
}
```

http://libfuse.github.io/doxygen/example_2hello_8c.html

现实中，FUSE能用来做什么？

- 出Lab！
- SSHFS（用ssh挂载远端目录到本地）
- NTFS-3G
- GMailFs（以文件接口收发邮件）
- WikipediaFS（用文件查看和编辑Wikipedia）
- 网盘同步
- 分布式文件系统（Lustre、GlusterFS等）
- *Everything is a file;* can everything be done with a filesystem?