# 文件系统模拟

文件系统模拟采用了类似 xv6的实现，但是没有实现同步(控制并发)操作和日志系统。

## 实现的接口

* ls
* mkdir
* touch
* pwd
* cd
* rm
* rmdir

## 基本的操作和使用

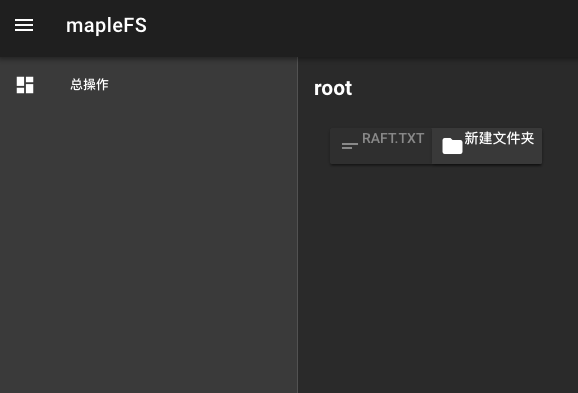
切换到 TongjiFileLab 目录下，运行 mkfs, 来生成文件系统的镜像：

cd TongjiFileLab/  
./mkfs

再运行服务器进程：

export GIN\_MODE=release   
./main # 在 0.0.0.0:8080 运行

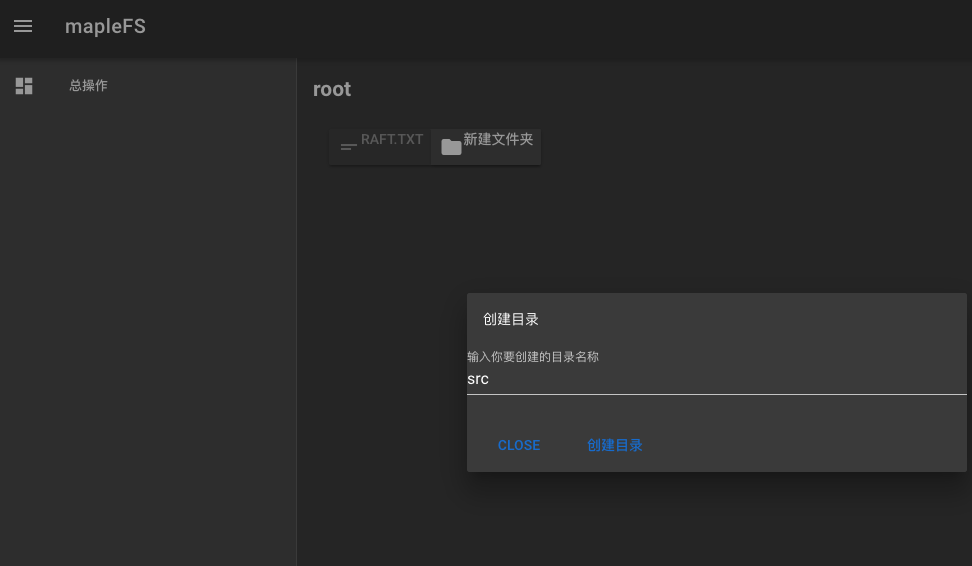
系统基本的使用可以在web 服务器上进行，访问可以得到页面：



你可以选择创建文件和目录，或者回到上一级目录(如果你所在的是根目录，那么你可以点击，但是不会去往新的地方。)



你可以选择创建文件／目录

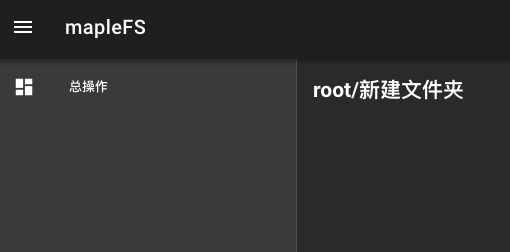


对于创建的目录，你可以选择删除：

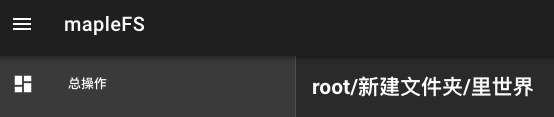


可以选择进入：





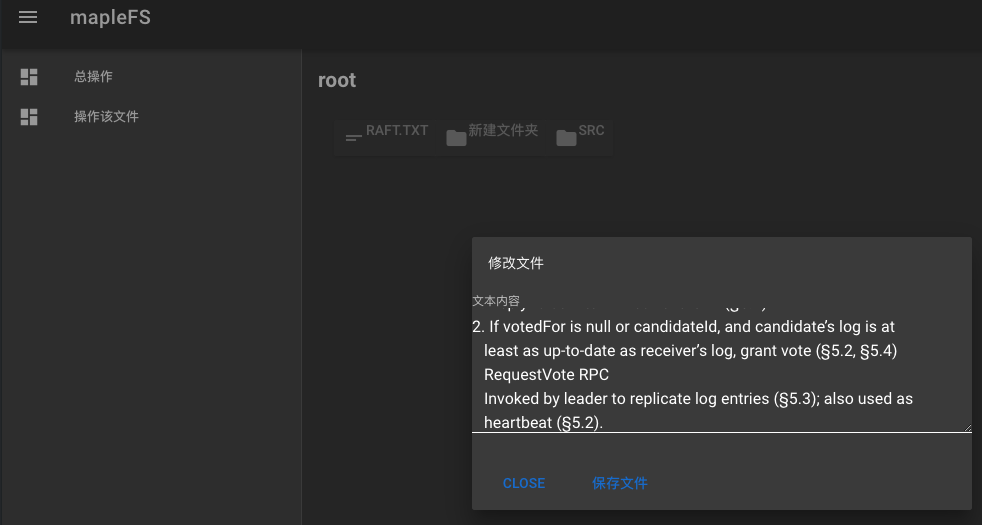
当然，可以建立多级的目录，只要总文件不超过这里的限制(暂定为200):



你可以选择“返回上一级目录”返回出来

对于文件，你可以创建文件。这里文件最大容量是(12 + 1 \* 1024 / (32/8) ) k, 即268k。 大于12k的需要间接索引。

这里存放大小140k的Raft论文：



保存成功后，我们再次打开，仍然能看到这篇论文。

现在我们关闭服务器并重新打开，仍然能看到以上的文件存在。

## 架构：

这里才用了自底向上的实现，上一层调用下一层或者别的层次提供的数据结构和接口。以下是关键的名词和接口的层次。

### BasicIO

基础的给程序提供I/O的包。

### SuperBlock

type superblock struct {  
 Size uint32 // size of blocks  
 Nblocks uint32 // number of datablocks  
 Ninodes uint32 // number of inodes  
}

Superblock 作为程序的超级块，储存着这一部分的文件系统的基本的信息。

### IDE Driver 层

0: unused  
 1: super block (size, ninodes)  
 2: log for transactions  
 32: array of inodes, packed into blocks  
 58: block in-use bitmap (0=free, 1=used)  
 59: file/dir content blocks  
 end of disk

|  |  |
| --- | --- |
| 区域 | 功能 |
| 0 | unused(本来是启动程序的扇区，但是没有实现对应功能) |
| 1 | super block |
| 2 | log for transactions(程序对应的log, 暂未实现) |
| x | inodes |
| y | block in use bitmap |
| z | file/dir content blocks |

我们可以对照代码方面的定义：

const (  
 // 总共大小  
 SIZE = 1024  
 // INODES 的数量  
 NINODES = 200  
 // BLOCKS 的数量  
 NBLOCKS = 995  
)  
  
const ROOT\_INODE\_NUM uint32 = 0  
const BLOCK\_SIZE = 512  
  
// 直接的指针  
const (  
 // 直接指针的数目  
 NDIRECT = 12  
 // 多级索引的最大上限。因为这里只有一个块作为二级索引  
 NINDIRECT = BLOCK\_SIZE / unsafe.Sizeof(uint32(0))  
 // 最多的文件指针？  
 MAX\_FILE = NDIRECT + NINDIRECT  
)  
  
const (  
 FILETYPE\_FREE = iota  
 FILETYPE\_FILE  
 FILETYPE\_DIRECT  
)  
  
// TODO:弄出一个实际的块...? 这个安全吗？  
// 一个BLOCK能存储的INODE的数目  
const IPB = BLOCK\_SIZE / unsafe.Sizeof(Dinode{})  
  
// Bitmap bits per block  
const BPB = BLOCK\_SIZE \* 8  
  
// BITMAP 占有的 BLOCK 的量  
const BITMAP\_BLOCK\_NUM uint32 = SIZE/(BLOCK\_SIZE\*8) + 1  
  
// 目录对应的 bytes  
const DIRSIZ = 28  
  
const MAX\_UINT16 = 65535  
  
type bufferStatus uint8  
  
const (  
 BUF\_BUSY bufferStatus = 1 << iota // buffer is locked by some process  
 BUF\_VALID bufferStatus = 1 << iota // buffer has been read from disk  
 BUF\_DIRTY bufferStatus = 1 << iota // buffer needs to be written to disk  
 BUF\_UNUSED bufferStatus = 1 << iota  
)  
  
const DIRENT\_SIZE = uint(unsafe.Sizeof(Dirent{}))  
  
const MAX\_UINT32 = 4294967295  
  
const bitblocks uint32 = SIZE/(BLOCK\_SIZE\*8) + 1  
const usedblocks = NINODES/uint32(IPB) + 3 + bitblocks

超级块：

type superblock struct {  
 size uint32 // size of blocks  
 nblocks uint32 // number of datablocks  
 ninodes uint32 // number of inodes  
}

日志系统：暂无

### Block Buffer 层

1. 同步对磁盘的访问
2. 缓存常用的块，提升性能

主要接口：bread bwrite 从磁盘中写入缓冲区／把缓冲区一块写到磁盘上正确位置。处理完后要用brelse

块缓冲利用LRU替换

binit 从buf取出

binit 从静态数组buf 构建有NBUF元素的 linkedlist 通过链表访问缓冲区

有 VALID DIRTY BUSY 三种状态

bget获得指定扇区缓冲 从磁盘中读取可能会调用iderw 扫描缓冲区链表，寻找到对应的，不处于BUSY 状态则返回（反之阻塞）

brelse 将缓冲区移动到链表头部，清除B\_BUSY

block在文件系统还是对应sector的一个固定大小的块，这里采用数据结构buffer来为访问具体的block提供接口和抽象。

/\*\*  
disk blocks  
 most o/s use blocks of multiple sectors, e.g. 4 KB blocks = 8 sectors  
 to reduce book-keeping and seek overheads  
 xv6 uses single-sector blocks for simplicity  
  
dev/sector 应该是单一指定的位置？  
\*/  
type buffer struct {  
 statusFlag bufferStatus  
 // dev, sector 是对应的设备、扇区管理  
 dev uint8 // 设备  
 sector uint16 // 扇区 这个程序里面表示所存储的块  
  
 prev, next, qnext \*buffer  
 // 对应的数据，有着固定的大小  
 data [BLOCK\_SIZE]byte  
}

这里buffer标定了 block的数据块，信息。用statusFlag表示这块buffer是否读写。buffer包抽象了具体的位置，提供了这些抽象的信息。

根据空闲块位图，分配新的块。

balloc 分配新的块 bfree 释放。 先用readsb从磁盘读 superblock, balloc寻找对应的块，同时清空对应的位。

### INode/Dinode 层

inode是 dinode 的记录 ialloc 申请新的i节点。 iget 会遍历 inode缓存寻找

bmap 会返回对应序号 inode 的内容

在操作系统中，dinode表示操作系统对文件的标示，一个文件只存在一个inode,inode用inum这个序号来表示。一下，同时表示又正确的文件的的大小。mapleFS中dinode表示磁盘上的inode`, 在文件系统中它的位置是不言自明的。

type Dinode struct {  
 FileType uint16 // 文件的类型  
 Nlink uint16 // link 链接的数量  
  
 Major, Minor uint16 // 对应的major minor, 我这里好像没啥用...好吧我他妈把MAJOR当成LINK链接好了,MINOR当成-s link好了  
 Size uint32 // size of file  
 Addrs [NDIRECT + 1]uint32 // 直接指向的数据块，最后一个+1对应的是二级索引  
  
}

inode是内存中的 inode. 保存了dinode, 我们的外部操作都是针对inode

/\*\*  
inode is dinode in memory  
  
 FS records file info in an "inode" on disk  
 FS refers to inode with i-number (internal version of FD)  
 inode must have link count (tells us when to free)  
 inode must have count of open FDs  
 inode deallocation deferred until last link and FD are gone  
  
\*/  
type INode struct {  
 num uint16 // 对应的序号  
 ref int // 引用计数  
 lock sync.Mutex // 内容的锁，暂时不会用到  
 valid int32 // 是否在disk中被读出  
  
 // copy of disk inode, 指向真实的block信息  
 dinodeData Dinode  
}

### 目录层

目录的 inode类型是 T\_DIR, dirlookup dirlink dirunlink 操作目录

### Dirent

// 存储目录项的条目  
// TODO: 搞清楚导入导出的机制  
type Dirent struct {  
 INum uint16  
 // 是不是到时候改回rune比较好  
 Name [DIRSIZ]byte  
 // 文件的类型  
 FileType uint16  
}

目录项，对应的目录的操作中，会像目录中添加目录项。每个目录项的大小是*32bytes*, 每个block能存储固定的dirent作为目录的记录。

### 文件描述符／系统调用层

sys\_link sys\_unlink nameiparent dirlookup

### File

const (  
 FD\_NONE = iota  
 FD\_PIPE = iota  
 FD\_INODE = iota  
)  
  
type FsFile struct {  
 ref int // ref cnt  
 inodePtr \*INode  
  
 readable bool  
 writeable bool  
}

*FIle*, 作为系统的文件，同样用inode表示，可以往里面同步内容，添加信息。

## 关键实现细节

### 多级文件读写

多级文件的读是在INode 层完成的，这里封装了 buffer 完成了对迭代器的操作，每次生产这个文件占有的buffer, 具体内容由外界的文件／目录来读取信息，完成操作。

func (node \*INode) BufferStream() <-chan \*buffer {  
 bufChan := make(chan \*buffer)  
 // 向 chan 发送信息  
 go func() {  
  
 var index uint16  
 for index = 0; index <= node.dinodeData.Nlink && index < NDIRECT && node.dinodeData.Addrs[index] != 0; index++ {  
 bufChan <- bget(uint16(node.dinodeData.Addrs[int(index)]))  
 }  
 if node.dinodeData.Nlink == NDIRECT && node.dinodeData.Addrs[NDIRECT] != 0 {  
  
 // 读取  
 secondBuf := bget(uint16(node.dinodeData.Addrs[NDIRECT]))  
 // 里面的元素数目  
 nodeSize := int(unsafe.Sizeof(uint32(0)))  
 // 不可能等于0，所以一个个读  
 var readSecondIndex uint32 // 读出来的索引地址  
 var secIndex = 0 // 次级对应的index  
 for secIndex < BLOCK\_SIZE/nodeSize {  
 readObject(secondBuf.data[secIndex\*nodeSize:(secIndex+1)\*nodeSize], &readSecondIndex)  
 if readSecondIndex == 0 {  
 break  
 } else {  
 secIndex++  
 bufChan <- bget(uint16(readSecondIndex))  
 }  
 }  
 }  
 close(bufChan)  
 }()  
 return bufChan  
}

写入文件采用 IAppend IModify接口，往文件中写入内容，先读取修改，占用已经使用的块，释放多余的块，并处理size, nlink等接口。

// 我总觉得这个函数会出事  
// 全部修改一个节点的信息  
func IModify(node \*INode, newData []byte) {  
 var editedBytes uint32 = 0 // 编辑过的byte  
 var remainBytes = uint32(len(newData))  
 var ifEnd = false // 读写是否停止  
 var cnt = 0  
 // 先用掉已经申请的块  
 for buf := range node.BufferStream() {  
 if !ifEnd {  
 cnt++  
 // 先编辑存在的buf  
 if remainBytes > BLOCK\_SIZE {  
 copy(buf.data[:], newData[editedBytes:editedBytes+BLOCK\_SIZE])  
 editedBytes += BLOCK\_SIZE  
 remainBytes -= BLOCK\_SIZE  
 } else {  
 bzero(buf)  
 copy(buf.data[:remainBytes], newData[editedBytes:editedBytes+remainBytes])  
 editedBytes += remainBytes  
 remainBytes = 0  
 // release node data  
 ifEnd = false  
 }  
 brelse(buf)  
 } else {  
 bfree(buf)  
 }  
 }  
 oldSize := node.dinodeData.Size  
 // 如果没有完成，iappend 会妥善修改内容  
 node.dinodeData.Size = editedBytes  
  
 // 内容过剩，处理NLink  
 currentUsed := editedBytes / BLOCK\_SIZE  
 if editedBytes%BLOCK\_SIZE != 0 {  
 currentUsed++  
 }  
 oldUsed := oldSize / BLOCK\_SIZE  
 if oldSize%BLOCK\_SIZE != 0 {  
 oldSize++  
 }  
  
 if !ifEnd {  
 // 使用  
 if currentUsed < NDIRECT {  
 node.dinodeData.Nlink = uint16(currentUsed)  
 } else {  
 node.dinodeData.Nlink = NDIRECT  
 }  
 // 没有结束，添加内容  
 if remainBytes > 0 {  
 IAppend(node, newData[editedBytes:])  
 }  
 } else {  
 // 使用  
 node.dinodeData.Nlink = uint16(currentUsed)  
 if currentUsed >= NDIRECT {  
 var secondUsed = int(currentUsed - NDIRECT)  
 buf := bget(uint16(node.dinodeData.Addrs[NDIRECT]))  
 intSize := int(unsafe.Sizeof(uint32(0)))  
 var secnt int  
 var nodeNum uint32  
 for secnt < int(oldUsed-currentUsed) {  
 readObject(buf.data[intSize\*(secondUsed+secnt):intSize\*(secondUsed+secnt+1)], &nodeNum)  
 // 释放多余的块  
 bfree(bget(uint16(nodeNum)))  
 secnt++  
 }  
 copy(buf.data[intSize\*secondUsed:], zeroBuf[intSize\*secondUsed:])  
 } else {  
 // 初始化 ADDRS 字段  
 node.dinodeData.Nlink = uint16(currentUsed)  
 for i := range node.dinodeData.Addrs {  
 if i > int(node.dinodeData.Nlink) {  
 node.dinodeData.Addrs[i] = 0  
 }  
 }  
 }  
 }  
 fsyncINode(node)  
}

## api系统接口

运行 main 可以打开文件系统的 api 访问文件系统的接口，与前端交互，以下是 对应的接口：

|  |  |  |
| --- | --- | --- |
| 方法 | url | 语义 |
| GET | /dirs | 获得当前目录所有的文件信息 |
| GET | /files/:name | 获得所访问的文件的内容 |
| GET | /cd/\*name | 切换当前目录 |
| POST | /dirs/:name | 在目录添加文件 |
| DELETE | /files/:name | 删除对应文件 |
| DELETE | /dirs/:name | 删除对应目录 |
| POST | /files/:name | 创建对应文件 |
| PUT | /files/:name | 修改文件内容 |

## 命令行接口

cmd 包的 main 是程序的命令行接口，你可以使用 help 来获得提示，访问同一个文件系统。

这里实现了 ls mkdir mkfile cd rmdir等命令.