

《操作系统课程设计》

(2022/2023 学年第一学期)

指导教师：田秋红

郭奕亿

班级：20 计算机科学与技术(4)

学号：2020333503081

姓名：陈伟剑

计划安排：

时间		地点	工作内容	指导教师
11.22	下午	10-414	任务布置，相关要求介绍与选题	田秋红、郭奕亿
	晚上	10-414	完成选题报告和需求分析文档	田秋红、郭奕亿
11.29	下午	10-414	算法分析与基本设计	田秋红、郭奕亿
	晚上	10-414	完成基本设计文档	田秋红、郭奕亿
12.6	下午	10-414	编写代码	田秋红、郭奕亿
	晚上	10-414	编写代码	田秋红、郭奕亿
12.13	下午	10-414	程序测试	田秋红、郭奕亿
	晚上	10-414	程序测试	田秋红、郭奕亿
12.20	下午	10-414	撰写课程设计报告，答辩	田秋红、郭奕亿
	晚上	10-414	撰写课程设计报告，答辩	田秋红、郭奕亿

目 录

1.	引言.....	1
1.1	任务要求.....	1
1.2	选题.....	1
2.	需求分析与设计.....	2
2.1	需求分析.....	2
2.1.1	“吃水果”问题.....	2
2.1.2	文件系统.....	3
2.2	系统框架和流程.....	3
2.2.1	Linux 文件系统源码解读绘制功能框架图.....	3
2.2.2	本系统功能框架和流程图.....	5
2.3	文件系统流程和模块描述.....	13
2.3.1	类图.....	13
2.3.2	用例图.....	15
2.3.3	顺序图.....	16
3.	数据结构.....	17
3.1	“吃水果”问题.....	17
3.1.1	全局变量.....	17
3.1.2	线程类.....	17
3.2	文件系统.....	17
3.2.1	基本数据结构类.....	17
3.2.2	服务类.....	20
4.	关键技术.....	22
4.1	“吃水果”问题.....	22
4.1.1	java 线程类.....	22
4.1.2	Semaphore 线程类.....	23
4.2	文件系统.....	23
4.2.1	数据持久化存储方案.....	23
4.2.2	利用位图管理所有盘块.....	24
4.2.3	基于链表实现文件内容存储.....	25
4.2.4	利用 java 反射机制设计命令处理流程.....	27
4.2.5	路径解析.....	32
4.2.6	文件大小更新.....	35
4.2.7	使用正则表达式约束权限的设置.....	36
5.	运行结果.....	36
5.1	运行环境.....	36
5.2	服务模式.....	37
5.3	运行结果.....	37
5.3.1	“吃水果”问题.....	37

5.3.2 文件系统.....	38
6. 调试和改进.....	56
6.1 持久化存储.....	56
6.2 cd 导致目录类型变化	56
6.3 权限的判断.....	58
6.4 man 方法修改	59
7. 心得和结论.....	60
7.1 结论和体会.....	60
7.1.1 “吃水果”问题	60
7.1.2 文件系统.....	60
7.2 进一步改进方向.....	63
7.3 分析设计方案对系统安全的影响.....	63
主要参考文献.....	63

1. 引言

操作系统是管理计算机硬件、软件资源并为计算机程序提供通用服务的系统软件，它包含进程管理、内存管理、文件管理、I/O 管理等功能。为了深入理解并实现底层操作系统的进程管理和文件管理，本课程设计选择模拟“吃水果”问题(使用 JAVA 实现)，另外选择设计一个包含文件基本操作以及一些拓展功能的多用户文件系统。

1.1 任务要求

为了模拟 B 类题的“吃水果”问题，有以下需求：

- (1) 多线程：需要利用 JAVA 线程类。
- (2) 进程间同步与互斥：通过 jdk 内置的 Semaphore 类实现线程之间的互斥。
- (3) 模拟需求中的“吃水果”：模拟时要打印线程(父母、儿女)的具体动作细节。

为了实现 C 类题的多用户文件系统，有以下需求：

- (1) 文件存储：目前考虑将文件存在本地，基本功能完善后考虑存储到云服务器上。
- (2) 文件状态：设置文件当前状态（是否有人编辑、文件读写权限等）。
- (3) 基本功能：实现用户验证(权限设置)、打开磁盘，读/写文件，创建、打开、删除、重命名文件和目录，更改文件状态等逻辑功能。

1.2 选题

B 类：“吃水果”问题

序号	实验项目名称	内 容 提 要	实验性质	实验种类
2	用 JAVA 语言模仿实现“吃水果”问题	1. 实现多进程的创建、进程间同步与互斥解决具体问题。 2. 问题描述：桌上有一盘子，桌上有一个空盘，允许存放一只水果，爸爸可向盘内放苹果，妈妈可向盘内放桔子，儿子专等吃盘内的桔子，女儿专等吃	专业基础	综合型

		<p>盘中的苹果。</p> <p>3. 桌上有一盘子，桌上有一个空盘，允许存放一只水果，爸爸可向盘内放苹果，妈妈可向盘内放桔子，儿子专等吃盘内的桔子，女儿专等吃盘中的苹果。</p>		
--	--	--	--	--

C 类：多用户文件系统

序号	实验项目名称	内 容 提 要	实验种类
1	文件系统	<p>设计一个多用户文件系统，理解文件系统的层次结构，完成基本的文件系统 create、open、close、read/write 等基本功能，并实现文件保护操作。实现以此为基础加入自己设计功能的小型文件系统。要求：</p> <p>1、解读部分 Linux 文件系统源码，绘出功能结构框图；</p> <p>2、本系统为多用户系统，可定义文件或目录的访问权限；</p> <p>3、模拟文件的存储和索引，整个文件系统为树状结构；</p> <p>4、实现文件/目录的基本命令，类似于 Linux 终端界面；</p> <p>5、实现文件保护操作，满足读读允许、读写互斥、写写互斥；</p> <p>6、友好的用户界面，合理的数据结构设计</p>	综合型

2. 需求分析与设计

2.1 需求分析

2.1.1 “吃水果”问题

通过 java 多线程模拟“吃水果”问题。桌上有一个空盘，允许存放一只

水果，爸爸可向盘内放苹果，妈妈可向盘内放桔子，儿子专等吃盘内的桔子，女儿专等吃盘中的苹果。一个盘子最多放一个水果，且同一时刻只能由一个人占用，这个占用可以是放水果也可以是取水果。程序运行时需要实时打印盘子的占用状态，方便用户了解模拟程序的运行情况。

四个家庭成员可以抽象为四个线程，设置互斥变量 Semaphore (java 自带的类) 来表示盘子类的占用情况，某个线程占用盘子时，先通过操作 Semaphore 对盘子上锁，以实现资源的互斥访问。

家庭成员每执行一次放(取)水果操作后，其线程就随机停顿一段时间，以此使运行过程更加具有随机性。

2.1.2 文件系统

参照 linux 模拟一个多用户文件系统，可执行的功能如下：

- ①注册登录
- ②创建、删除文件
- ③文件权限管理
- ④读写文件
- ⑤指令帮助
- ⑥目录位置切换
- ⑦文件存储

2.2 系统框架和流程

2.2.1 Linux 文件系统源码解读绘制功能框架图

2.2.1.1 linux 文件系统主要的数据结构

Linux 中的文件系统会给每个文件分配两个数据结构：索引节点 (index node) 和目录项 (directory entry)，它们用来记录文件的元信息和目录层次结构，具体介绍如下(整体图示见图 2-1)：

①索引节点：inode，用来记录文件的元信息，如 inode 编号、文件大小、访问权限、创建时间、修改时间、数据在磁盘的位置等。索引节点是文件的唯一标识，它们之间一一对应，都被存储在硬盘当中，索引节点会占用磁盘的存

存储空间。另外，目录也是文件，它也用索引节点唯一标识，不同于普通文件的是，普通文件在磁盘中存放的是文件数据，而目录文件在磁盘中存放的是子目录或文件。

②目录项：dentry，用来记录文件的名称、索引节点指针以及与其他目录项的层级关联关系。多个目录项关联起来，就会形成目录结构，它与索引节点不同的是，目录项是由内核维护的一个数据结构，缓存在内存中而非磁盘。目录项包括文件名和 inode 节点号。

当磁盘格式化时会被分成三个存储区域：超级块(superblock)、索引节点区(inode)、数据块区(block)。具体介绍如下：

①超级块(superblock)：用来存储文件系统的详细信息，包括 inode/block 的总量、使用量、剩余量（块个数、块大小、空闲块等），以及文件系统的格式。

②索引节点(inode)区：存储文件的索引节点。

③数据块(block)区：存储文件/目录的数据。

④inode 映射表：inode bitmap：位图，记录 inode 的使用情况。

⑤block 映射表：block bitmap：记录 block 的使用情况。

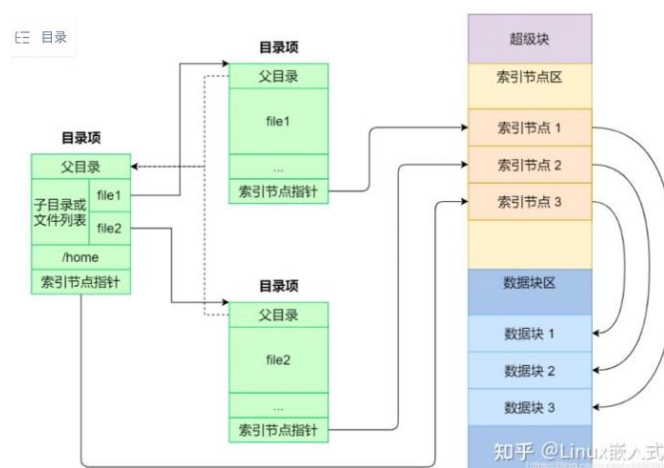


图 2-1 索引节点、目录项以及文件数据之间的关系[1]

2.2.1.2 linux 文件系统功能框架

底层的文件系统有 FAT、EXT2、MINIX 等类型，linux 为了统一操作各种文件系统建立了虚拟文件系统(VirtualFileSystem,VFS),VFS 提供了数十种文件系统，为文件系统提供了一个通用的接口抽象，下图为 linux 文件系统的整体架构

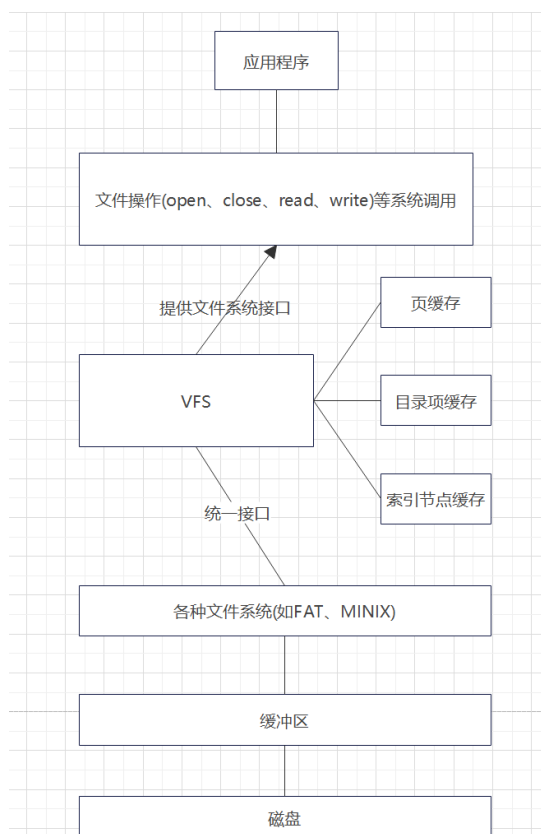


图 1-2 VFS 在 linux 文件系统的整体架构

2.2.2 本系统功能框架和流程图

2.2.2.1 “吃水果”问题

“吃水果”问题的基本流程如下图所示

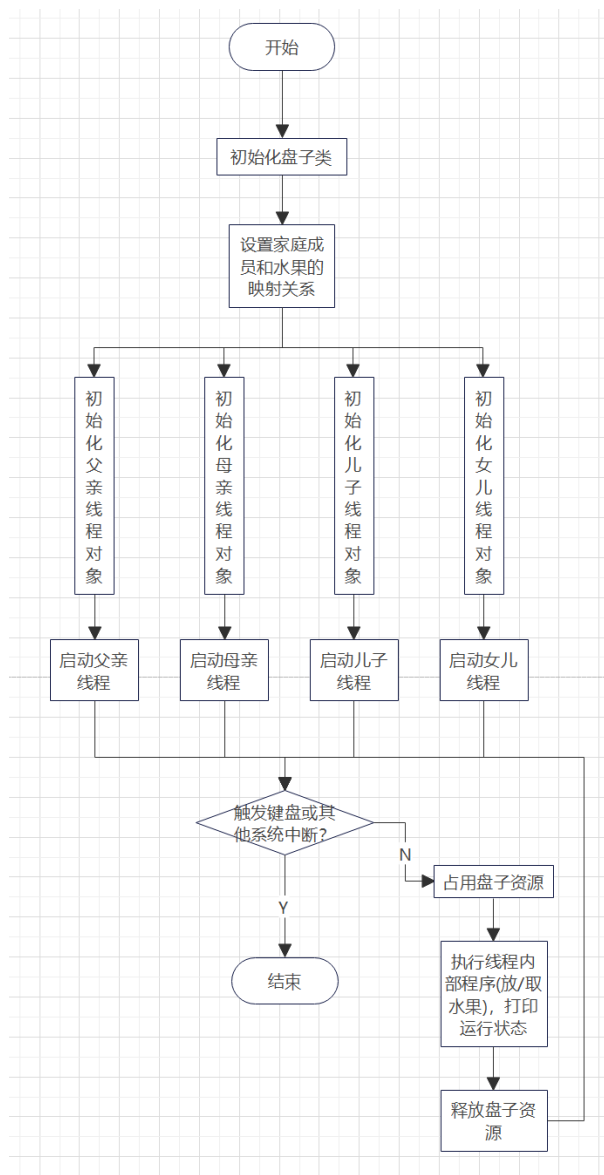


图 2-2 模拟“吃水果”问题流程图

2.2.2.2 文件系统

多用户文件系统的功能结构图如下

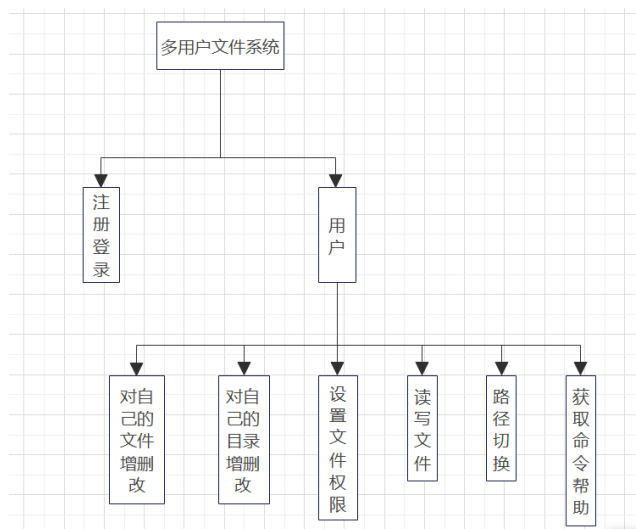


图 2-3 多用户文件系统基本功能架构

整个系统模仿 linux 的命令行进行实现，主程序流程如下：

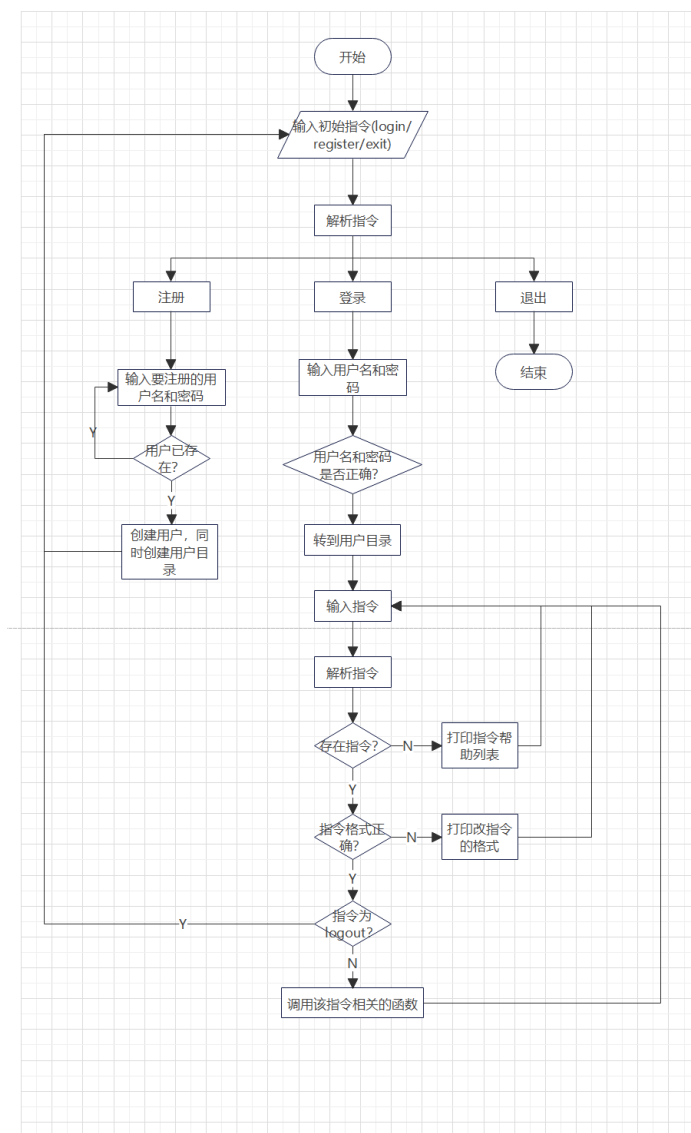


图 2-4 主程序流程图

期望实现的指令列表如下

表 2-1 期望实现的指令列表

指令名	含义
login	登录
register	注册
exit	结束程序
logout	退出登录
mkdir	创建目录
ls	列出当前目录下的文件名
ll	列出当前目录下的文件部分具体属性
pwd	当前路径
touch	创建文件
chmod	修改文件权限
cd	切换路径
open	打开文件
close	关闭文件
delete	删除文件
rename	重命名文件
man	查看给定指令的具体信息

下面给出部分指令的流程图。

mkdir 的流程图如下(touch 命令的流程与此相近)

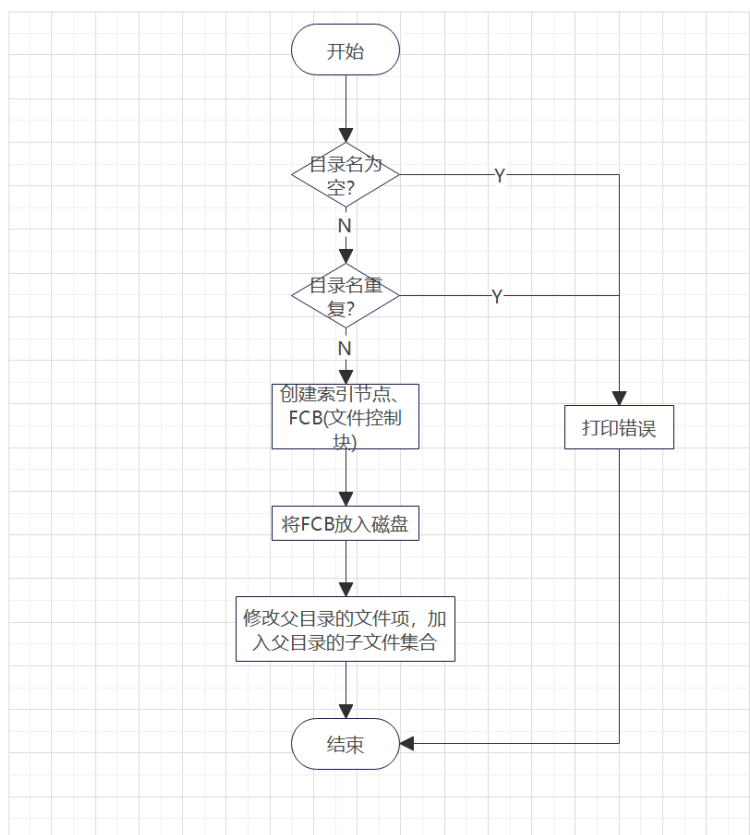


图 2-5 mkdir 流程图

pwd 的流程图如下

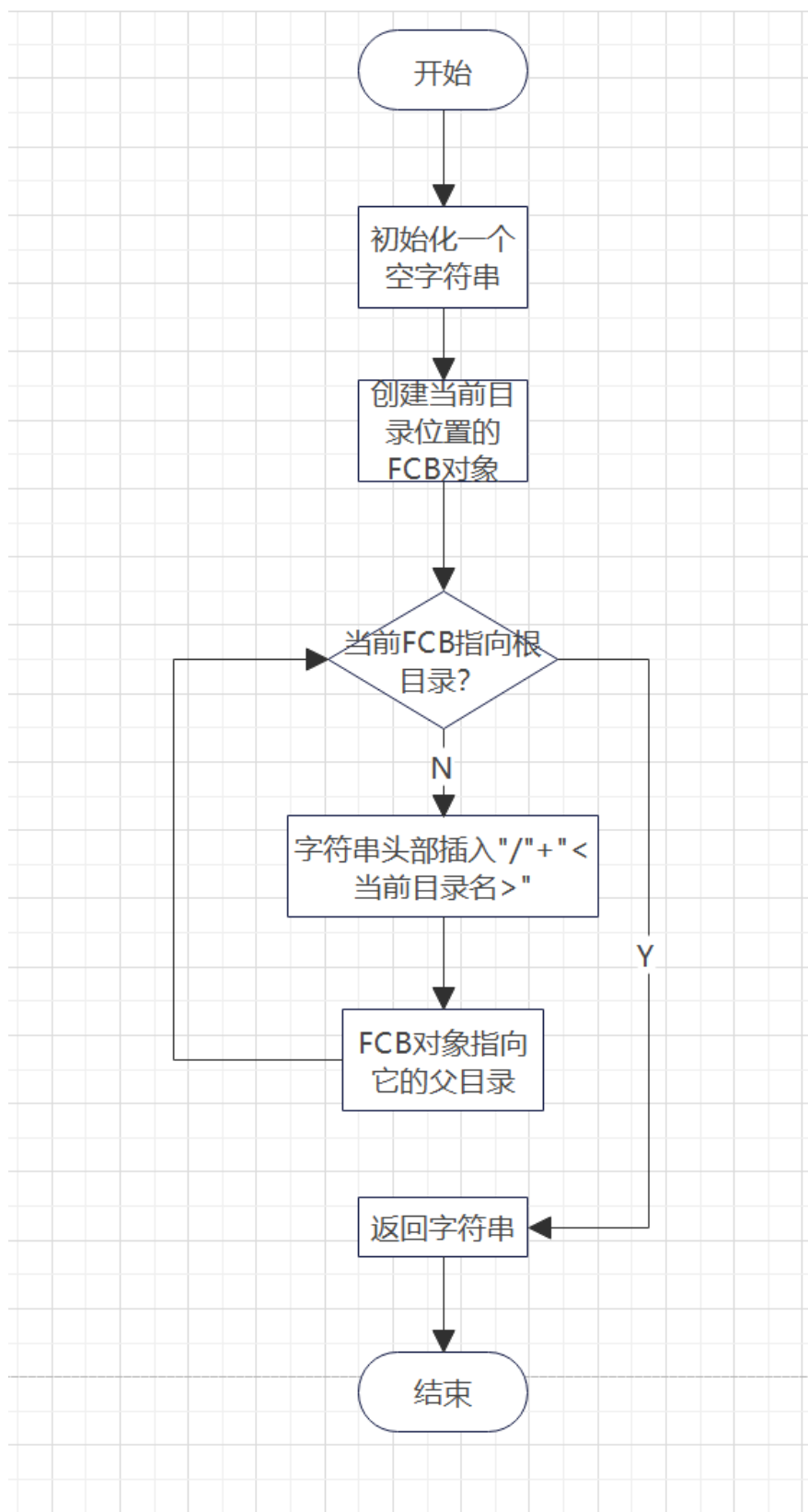


图 2-6 pwd 流程图

cd 的流程图如下

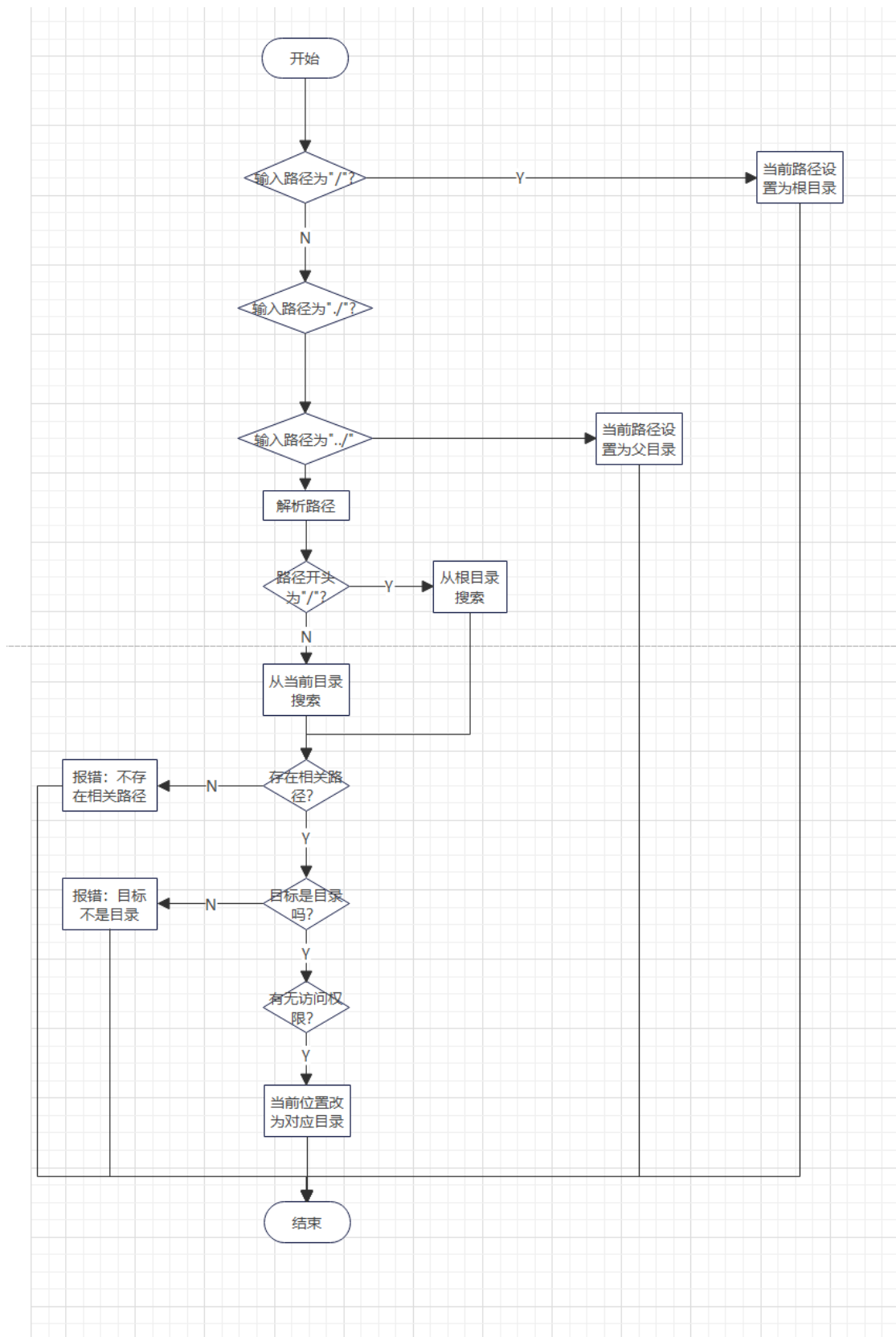


图 2-7 cd 流程图

open 指令的流程如下(close 的流程大体相似)

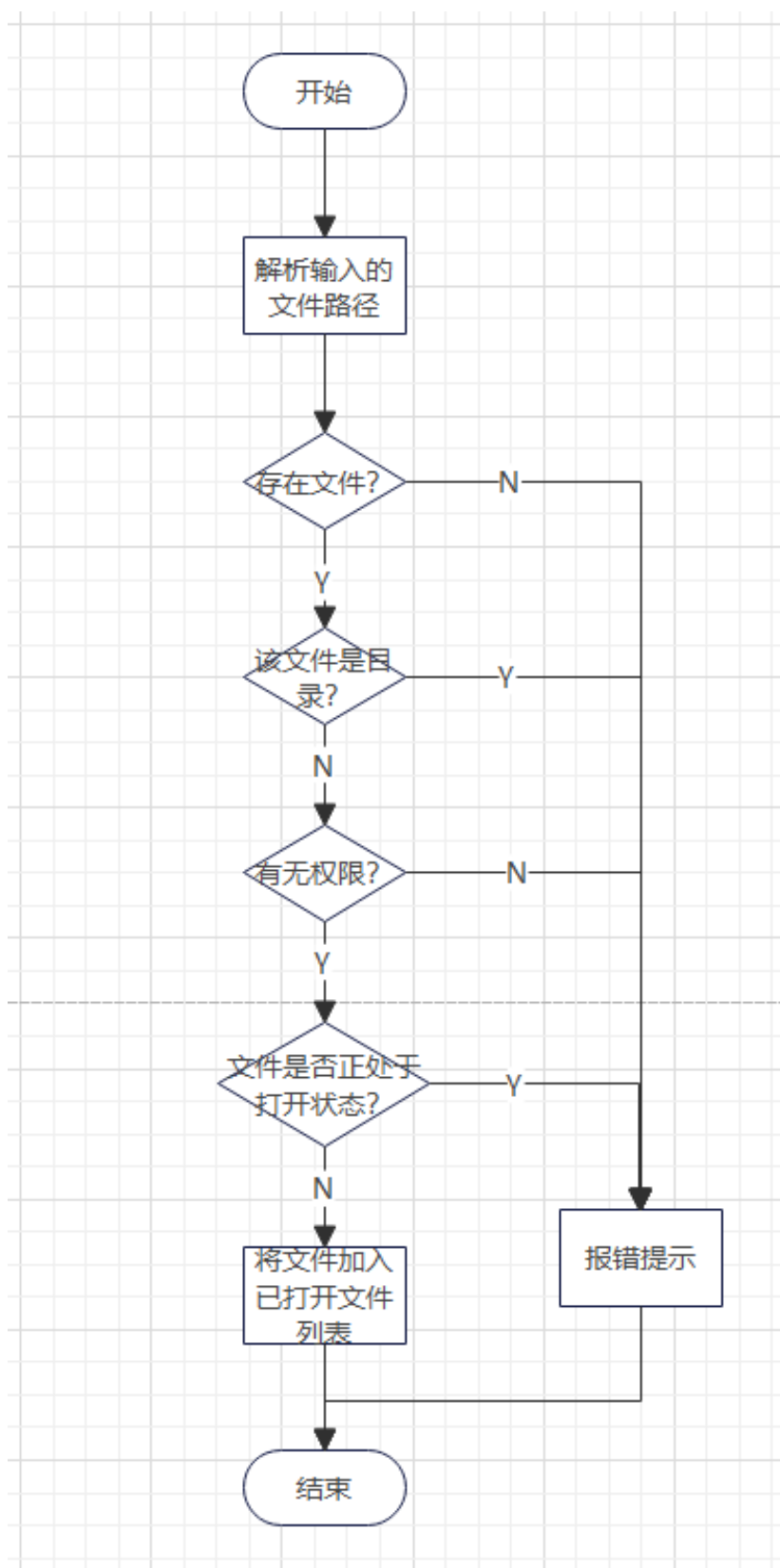


图 2-9 open 流程图

write 指令的流程图如下(read 的流程大体相似)

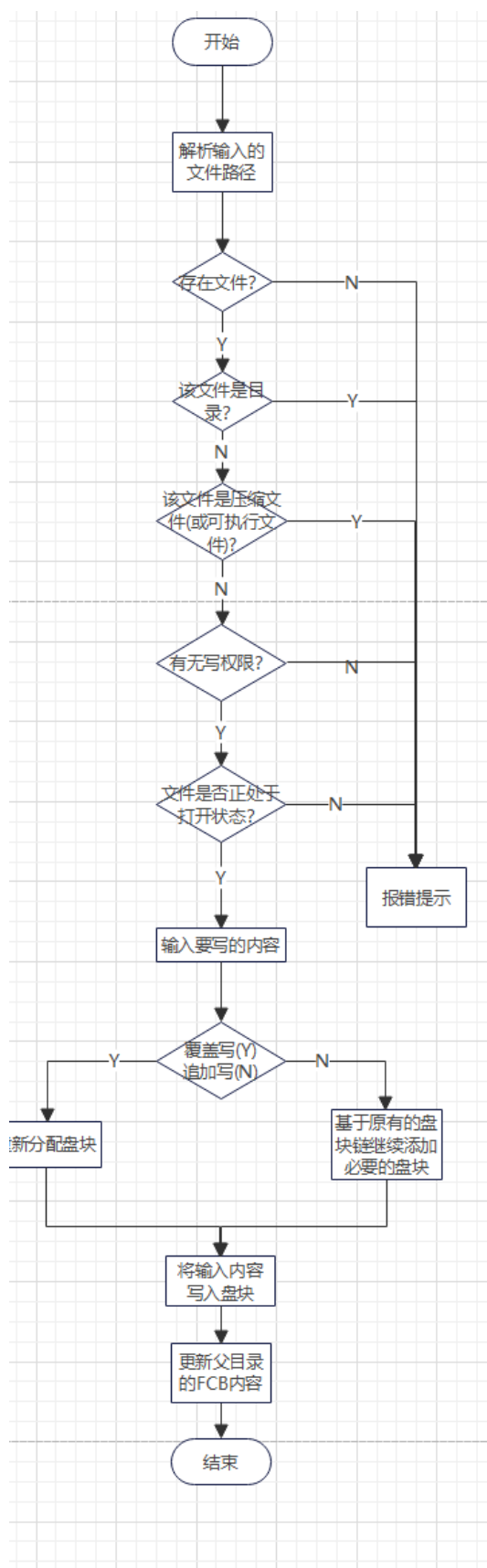


图 2-10 write 流程图

2.3 文件系统流程和模块描述

2.3.1 类图

查阅资料了解到，Linux 文件系统需要对磁盘进行一系列的抽象。一块磁盘除去用户区、所有的文件描述符(FCB)、文件分配表(FAT)这三个部分外，其余部分为空闲区域。为了提高空闲区域的利用效率，需要将空闲区域划分为多个盘块 Block，专门用来存放文件的具体数据。文件系统运行时，内存需要从磁盘读入用户数据、FCB、FAT，这些数据在磁盘中占比很小，但可以用来规划文件具体数据的控制和存储。

关于 linux 文件系统的粗略图示如下

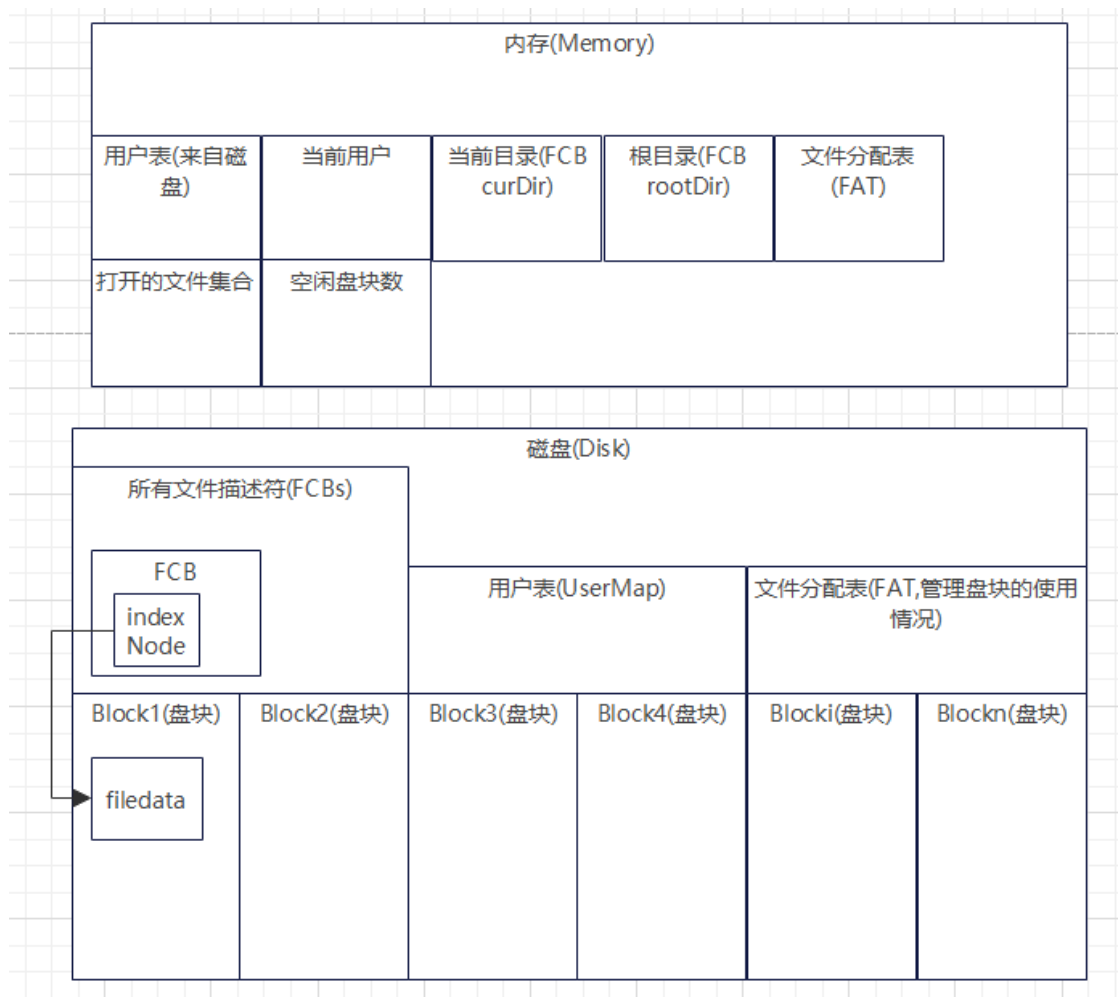


图 2-11 文件系统图示

一个文件拥有一个 FCB，FCB 本身记录着文件名、类型、父 FCB、子 FCB 列表，主要用于描述文件的名称和在文件系统树中的位置。

另外，一个 FCB 内部还拥有一个 IndexNode(在 linux 中是 inode)，indexNode 记录着文件的访问权限、大小、首个盘块号、创建者等更具体一点描述内容。

文件的具体内容需要存在盘块(Block)中，不过由于内容大小的不确定性，本文件系统采用链表的方式为文件灵活分配用于存储的磁盘空间，这也是上文中 `indexNode` 存储首个盘块号，它相当于链表的头结点。

利用文件分配表(FAT,本质是个位图)管理盘块的占用情况,假设磁盘的空闲块划分为n个盘块,那么FAT的长度也为n,当盘块i被占用时,FAT第i位置1,否则置0。这样,每次为文件分配存储空间时,先通过FAT寻找空闲的盘块,然后将获取到的盘块加入改文件的盘块链表。

综上，整理出本文件系统的类图如下

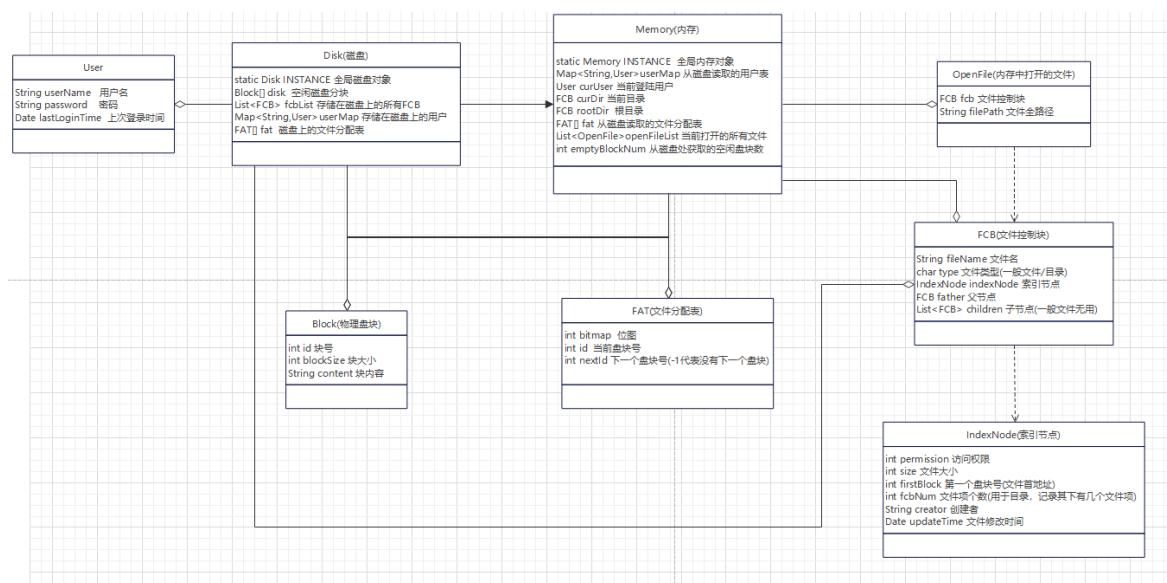


图 2-12 文件系统类图

2.3.2 用例图

基于 2.2.2.2 给出的文件系统架构图，作出如下文件系统用例图

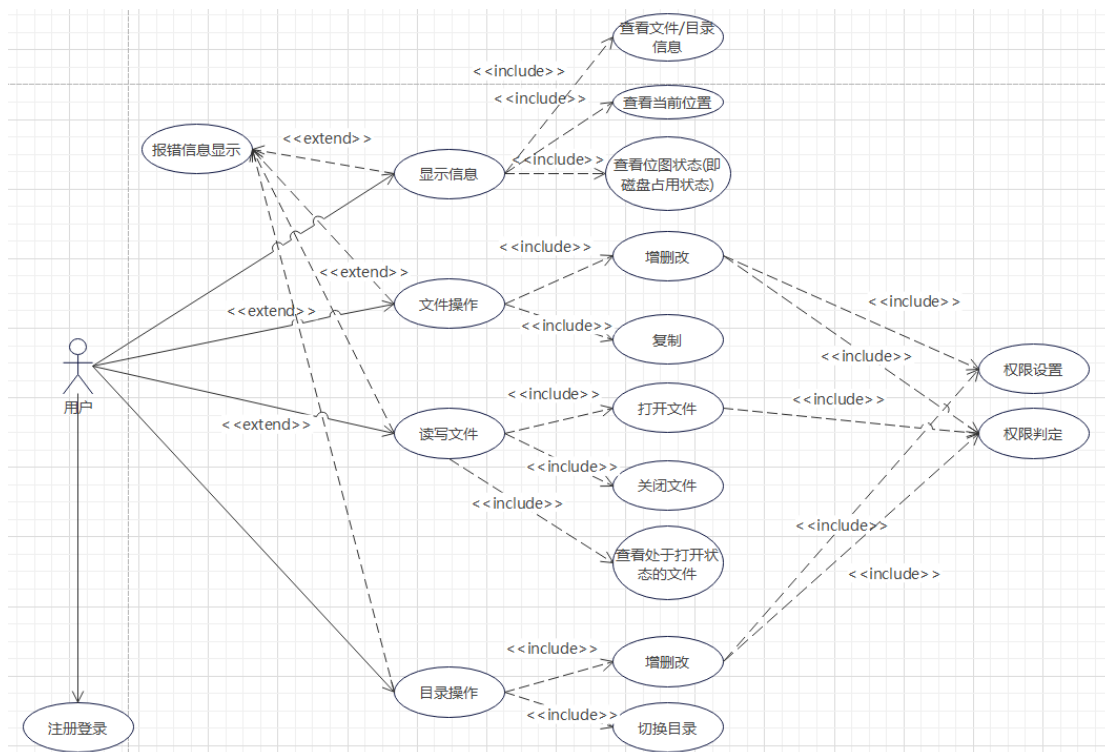


图 2-13 文件系统用例图

2.3.3 顺序图

基于 2.2.2.2 中整体流程的设计，做出如下顺序图

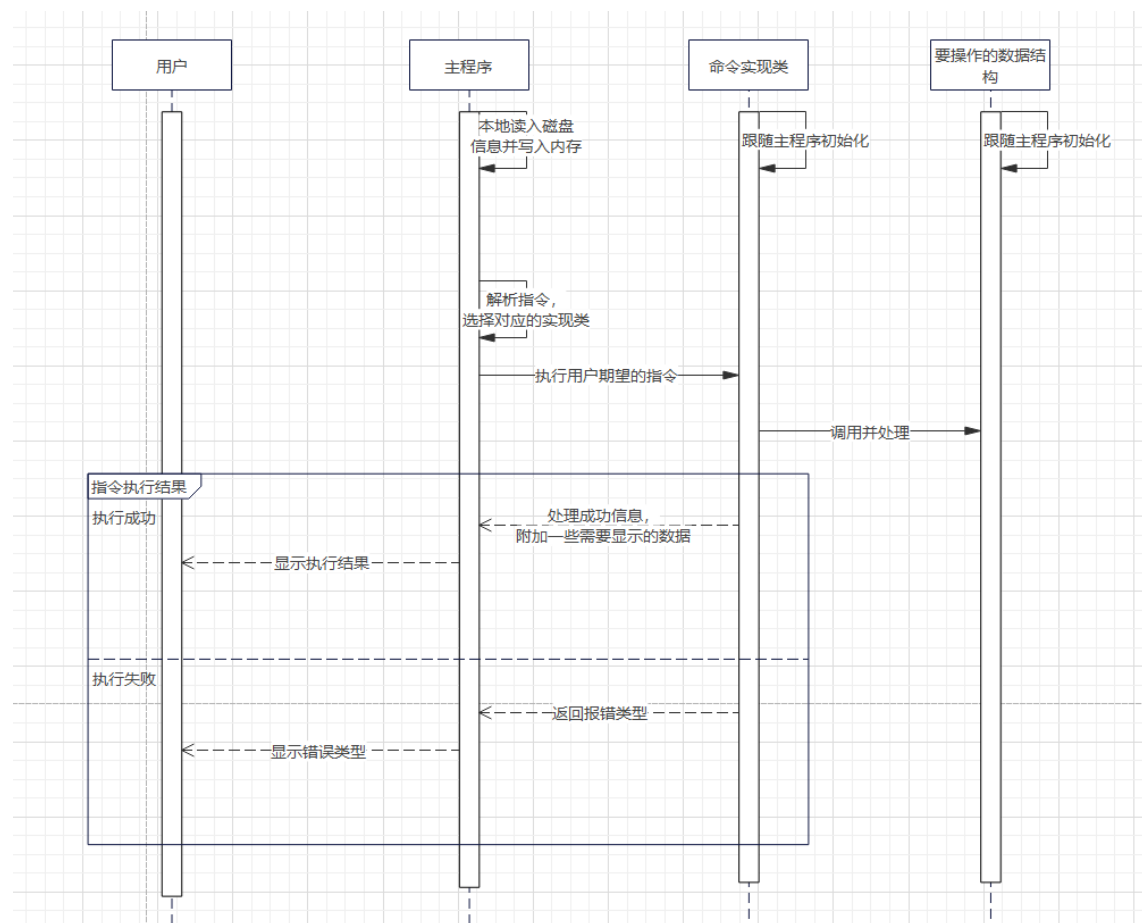


图 2-14 文件系统顺序图

3. 数据结构

3.1 “吃水果”问题

3.1.1 全局变量

①mutex

类型为 java 内置的 Semaphore 类型，线程在占用盘子资源前先通过 `mutex.acquire()` 获取（或等待）盘子的资源，执行完操作之后通过 `mutex.release()` 释放盘子资源。

②Plate

全局类，里面存储家庭成员和水果之间的关系映射。且内置了 `put`、`get` 方法，用来模拟放(取)水果操作。

3.1.2 线程类

设置 Member 类，成员变量如下：

```
String member    //家庭成员名称
Boolean isParent //是不是家长
```

父母、儿子、女儿都通过这个类进行创建，通过 `isParent` 来判断线程要执行放水果操作还是取水果操作，通过 `member` 结合 `Plate` 中的映射表来指定放(取)的水果。

3.2 文件系统

3.2.1 基本数据结构类

3.2.1.1 User(用户)

成员名	类型	含义
<code>userName</code>	<code>String</code>	用户名

password	String	密码
lastLoginTime	Date	上次登录的时间

3.2.1.2 Block(物理盘块)

成员名	类型	含义
id	int	块号
blockSize	Int	块大小
content	String	块内容

3.2.1.3 FAT(文件分配表)

成员名	类型	含义
bitmap	int	位图(0 表示空闲, 1 表示占用)
id	int	当前盘块号
nextId	int	指向下一个盘块号

3.2.1.4 IndexNode(索引节点)

成员名	类型	含义
permission	String	文件访问权限
size	int	文件大小
firstBlock	int	文件首地址(第一个盘块号)
fcNum	int	文件项个数(目录用, 普通文件设为 0)
creator	String	创建者

updateTime	Date	文件修改时间
------------	------	--------

3.2.1.5 FCB(文件控制块)

成员名	类型	含义
fileName	String	文件名
type	Character	文件类型
indexNode	IndexNode	索引结点
father	FCB	父节点
children	List<FCB>	子节点(目录用)

3.2.1.6 OpenFile(内存中打开的文件)

成员名	类型	含义
fcb	FCB	文件控制块
filePath	String	文件全路径

3.2.1.7 Disk(磁盘)

成员名	类型	含义
INSTANCE	Disk(static)	持久化磁盘对象
disk	Block[]	磁盘盘块
fcbList	List<FCB>	存储在磁盘上的 FCB 集合
userMap	Map<String, User>	存在磁盘上的用户
fat	FAT[]	文件分配表

3.2.1.8 Memory(内存)

成员名	类型	含义
INSTANCE	Memory(static)	内存对象（全局变量）
userMap	Map<String, User>	用户表
curUser	User	当前登录的用户
curDir	FCB	当前路径
rootDir	FCB	跟路径
fat	FAT[]	文件分配表
openFileList	List<OpenFile>	已打开的文件集合
emptyBlockNum	Int	空闲盘快数

3.2.2 服务类

3.2.2.1 UserService(用户服务类)

方法名	功能	参数	返回类型
login	登录	用户名、密码	Boolean
register	注册	用户名、密码	Boolean
logout	登出	NULL	Boolean

3.2.2.2 DirService(目录操作类)

方法名	功能	参数	返回类型
dir	显示当前目录的文件项、文件个数、目录大小	NULL	Void
mkdir	创建目录	目录名、权限	Boolean
cd	切换目录	目录名	Boolean

pathResolve	解析路径	路径	FCB
updateSize	递归修改父目录大小	FCB、是否添加文件	Void
ls	简略打印目录的文件名信息	NULL	Void
pwd	显示当前目录全路径	FCB	String
showpath	显示当前目录	NULL	Void
bitmap	显示位图	NULL	Void

3.2.2.3 FileService(文件操作类)

方法名	功能	参数	返回类型
create	创建文件	文件名、权限	Boolean
open	打开文件	文件路径	Boolean
show_open	显示打开的文件	NULL	Void
read	读文件	文件路径	Boolean
write	写文件	文件路径	Boolean
close	关闭文件	文件路径	Boolean
delete	删除文件	文件路径	Boolean
rename	重命名	文件路径、新文件名	Boolean

3.2.2.4 DiskService(磁盘操作类)

方法名	功能	参数	返回类型
-----	----	----	------

freeDir	释放目录中的所有文件	FCB	Boolean
freeFile	释放文件	FCB	Boolean
writeToDisk	写入磁盘	文 件 内 容 (String)	Int(第一个盘块的盘块号)
findEmpty	寻找空闲盘块	NULL	Int(盘块号)

3.2.2.5 DataService(数据存储类)

方法名	功能	参数	返回类型
init	初始化磁盘(若没找到保存的磁盘对象, 则调用这个方法)	NULL	Void
loadData	读入磁盘对象	磁盘对象的存储路径	Boolean
saveData	保存磁盘对象	磁盘对象的存储路径	Boolean

4. 关键技术

4.1 “吃水果”问题

4.1.1 java 线程类

程序创建了 Member 类, 继承 Thread, 重写其 run() 函数, 通过 isParent 成员来判定要执行取水果操作还是放水果操作。家庭成员每执行一次对盘子的操作就在停顿一段时间(停顿时长在 0.2~1s 之间随机选择)。

4.1.2 Semaphore 线程类

程序使用了 Semaphore 信号类，保障同一时刻只有一个线程可以获取盘子资源，每个线程类在获取盘子资源之前都需要先获取(或等待)Semaphore，完成取水果(放水果)操作后释放 Semaphore，以此保证了同一时刻只有一个线程可以进行拿水果(放水果)操作，完成了“吃水果”问题的互斥关系模拟。

4.2 文件系统

4.2.1 数据持久化存储方案

由 3.2 中的系统设计已知，磁盘(Disk)对象存储了所有文件系统的相关内容(如 fcb, block, userMap)，因此数据保存时只需将磁盘对象写入指定文件中即可(这里设定为 system.data)。下一次初始化时，磁盘类需要先寻找 system.data 文件，如果找到，则将磁盘对象从中取出并设为全局变量，否则提示用户新建一个磁盘对象。

另外，为了结合实际的文件存储，考虑限制盘块的数量(设为 256 块)，且每个盘块存储的字符数量固定(设为可以存储 8 个 ascii 字符)。

Java 支持类的读写，前提是类需要实现序列化接口并指定一个序列号。对于磁盘类，可以如下编写

```
public class Disk implements Serializable {  
    private static final long serialVersionUID = 1L; //序列号  
    private static Disk INSTANCE;  
    private Block[] disk; //磁盘  
    private List<FCB> fcbList; //存储在磁盘上的 FCB 集合  
    private Map<String, User> userMap; //存储在磁盘上的用户集合  
    private FAT[] fat; //文件分配表  
}
```

然后通过 java 提供的 ObjectOutputStream 类来执行将对象写入指定文件的操作(读文件操作与之类似)，关键代码如下

```
public Boolean saveData(String savePath) {  
    // 根据给定的路径获取用于存储的文件
```

```
File file = new File(savePath);
// 检查文件是否存在
if (!file.exists()) { // .....}

ObjectOutputStream oos = null;
try {
    oos = new ObjectOutputStream(new FileOutputStream(file));
    // 持久化到要保存的文件中
    oos.writeObject(Disk.getInstance());
    oos.flush();
    System.out.println("[success]:保存数据成功");
    return true;
} catch (IOException e) {
    System.out.println("[error]:数据保存失败");
    return false;
} finally {
    try {
        if (Objects.nonNull(oos)) {
            oos.close();
        }
    } catch (IOException ignored) {
    }
}
}
```

4.2.2 利用位图管理所有盘块

结合 linux 文件系统的设计，一个磁盘可以均分为多个盘块(假定为 n 个)，通过文件分配表(FAT)数据结构进行管理，磁盘对象中存储的是 `FAT[]` 数组，数组元素的下标代表盘块号，数组元素的值代表对应的盘块是否已分配(约定 0 代表未分配，1 代表已分配)。[2]

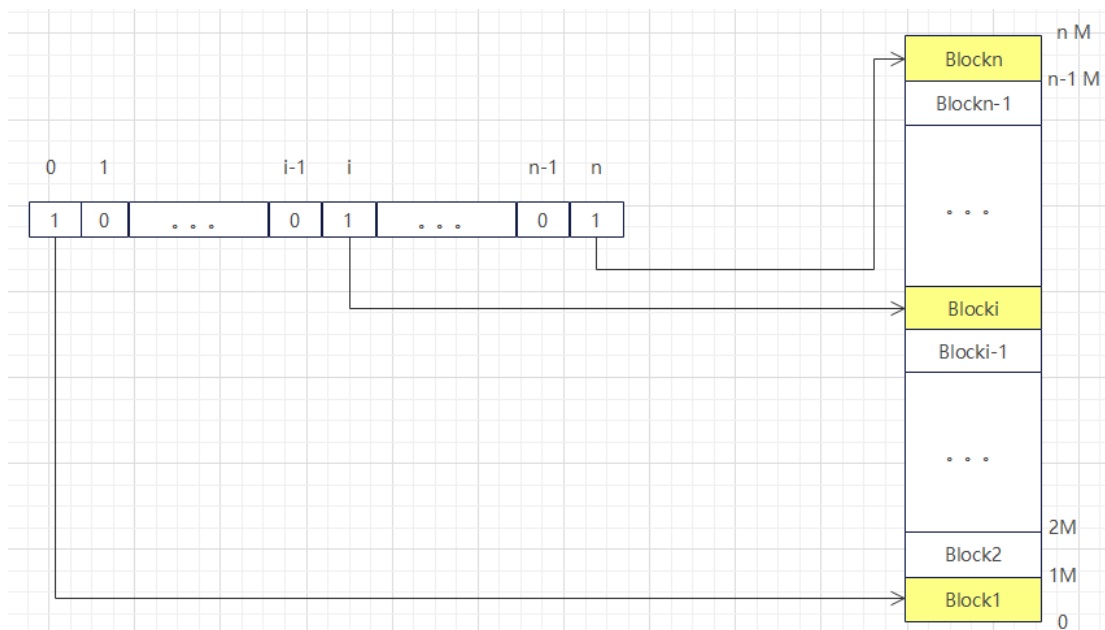


图 4-1 FAT 管理盘块

在类设计中，FAT 不只存储当前盘块号的占用状况，还会存储下一个盘块号，这样也便于 4.2.3 中的利用链表存储文件内容。

4.2.3 基于链表实现文件内容存储

由 3.2 中的类分析已知，文件的元数据由 FCB 管理，更具体一点的数据由 indexNode 管理(相当于 linux 中的 inode)，indexNode 中存储首个盘块节点，这个盘块是链表的一部分，一个文件如果数据量比较大，那么就需要通过文件分配表(FAT)申请足够多的盘块来存储文件数据，申请到的一系列盘块通过链表的形式进行连接，好处是 indexNode 只需要存储链表的头结点即可，而且可以更充分地利用空闲盘块。[2]

下面以覆盖写操作为例描述实现的过程，对于磁盘分配的策略，我假定一个盘块可以存储 BLOCK_SIZE 个 ascii 字符(后续输入也只考虑 ascii 字符)，将写入文件的内容除以单个盘块的大小后向上取整，可以计算出需要分配的盘块数量。如果空闲盘块数量足够，则从 FAT[] 数组中取出空闲盘块、写入内容、拼接到该文件的存储盘块链表中，这个流程的执行次数要根据分配的盘块数量而定。

主流程的代码如下

```
//覆盖写
//如果不是空文件 则清空之前占据的盘块
if(fcb.getIndexNode().getSize() != 0){
    diskService.freeFile(fcb);
```

```
}  
//重新写入  
int first = diskService.writeToDisk(content.toString());  
if(first==-1){  
    return false;  
}  
//将文件指向第一块  
fcb.getIndexNode().setFirstBlock(first);  
//修改索引结点大小  
fcb.getIndexNode().setSize(size);  
//修改父目录项 自底向上修改父目录的大小  
dirService.updateSize(fcb, true, -1);
```

其中，writeToDisk 的具体代码如下

```
public int writeToDisk(String content) {  
    //判断是否有足够的磁盘空间  
    int needNum = Utility.ceilDivide(content.length(),  
Constants.BLOCK_SIZE);  
    if(needNum > Memory.getINSTANCE().getEmptyBlockNum()){  
        System.out.println("[error]: 磁盘空间不足!");  
        return -1;  
    }  
    //利用双指针的方法操作链表  
    FAT[] fats = Memory.getINSTANCE().getFat();  
    int first = -1;  
    //找到第一个空闲盘块  
    first = findEmpty();  
    int temp_1 = first;  
    int temp_2 = -1;  
    Block[] disk = Disk.getINSTANCE().getDisk();  
    int i = 0;  
    for (; i < needNum - 1; i++) {  
        //拆分存储输入的字符  
        String splitString =  
content.substring(i*Constants.BLOCK_SIZE, (i+1)*Constants.BLOCK_SIZE);  
        //存储到盘块中  
        disk[temp_1].setContent(splitString);  
        fats[temp_1].setBitmap(1);  
        temp_2 = temp_1;  
        //寻找下一个空闲块  
        temp_1 = findEmpty();  
        //设置链表的下一个节点  
        fats[temp_2].setNextId(temp_1);  
    }  
}
```

```
    }  
    //设置最后一个盘块  
disk[temp_1].setContent(content.substring((i)*Constants.BLOCK_SIZE));  
    fats[temp_1].setNextId(-1);  
    fats[temp_1].setBitmap(1);  
    //返回首个磁盘块号(链表头部)  
    return first;  
}
```

文件内容释放的操作跟上面的代码大致相近，也是通过双指针的方式将文件占用的盘块链表节点(FAT)逐个重置。

4.2.4 利用 java 反射机制设计命令处理流程

按照常规的设计思路，对于以命令行形式提供用户交互界面的 shell 程序，我会选择直接使用 switch 语句来识别输入的指令，但是如果指令数量超过 10 个或更多时，主函数会显得非常臃肿，比如下面这条 switch 语句

```
switch (inputs2[0]) {  
    case "login":  
        userService.login(null, null);  
        break;  
    case "register":  
        userService.register(null, null);  
        break;  
    case "logout":  
        logout = userService.logout();  
        break;  
    case "mkdir":  
        if(inputs2.length != 2){  
            System.out.println("mkdir [dirName]");  
            break;  
        }  
        String permissionDir=Utility.inputPermission();  
        dirService.mkdir(inputs2[1],permissionDir);  
        break;  
    // 下面还有 10 多条指令,总和超过 70 行  
    // .....  
}
```

为了简化主函数的大小，有必要用其他更好的方法替代一长串的 switch 语句。

事实上，查阅了 linux0.11 版的源码[3]后才发现，linus 通过 fork 结合

函数指针的方式实现了指令的解析与调用，代码量明显低于普通的 switch。下面为 linux0.11 中的 main 函数(部分注释直接翻译自 linux 创建者 linus 的原话)

```
void main(void)
{
    /* 在 startup 程序(head.s)中就是这样假设的。 */

    // 一些内存初始化代码，这里不放出
    //.....

    //还有一些别的初始化函数，包括块设备初始化、字符设备初始化、tty 初始化、设置
    //开机启动时间等。这里不放出
    //.....

    // 下面过程通过在堆栈中设置的参数，利用中断返回指令切换到任务 0。
    move_to_user_mode();    // 移到用户模式。(include/asm/system.h)
    if (!fork()) {          /* we count on this going ok */
        init();
    }
    for(;;) pause();
}
```

进入 init() 函数，结合资料会发现了这一段关键代码

```
void init(void)
{
    int pid,i;

    // 一些初始化代码，这里不放出
    // .....
    while (1) {
        if ((pid=fork())<0) {
            printf("Fork failed in init\r\n");
            continue;
        }
        if (!pid) {
            close(0);close(1);close(2);
            setsid();
            (void) open("/dev/tty0",O_RDWR,0);
            (void) dup(0);
            (void) dup(0);
            // 这一步通过 fork 的子进程运行 shell 程序
            _exit(execve("/bin/sh",argv,envp));
        }
    }
}
```



```

        while (1)
            if (pid == wait(&i))
                break;
            printf("\n\rchild %d died with code %04x\n\r",pid,i);
            sync();
        }
        _exit(0); /* NOTE! _exit, not exit() */
    }
}

```

Linux0.11 源码中的 shell 程序写在/bin/sh 中，不过并没有在源码中找到这一部分的代码，翻阅了源码的其他部分，发现在 linux0.11 源码下的 /include/linux/sys.h 中定义了所有基本的系统调用函数，如下

```

extern int sys_setup (); //系统启动初始化设置函数。(kernel/blk_drv/hd.c, 71)
extern int sys_exit (); // 程序退出。(kernel/exit.c, 137)
extern int sys_fork (); // 创建进程。(kernel/system_call.s, 208)
extern int sys_read (); // 读文件。(fs/read_write.c, 55)
// 还有其他系统调用函数，每个函数在下面的系统调用函数指针表中都有记录
//.....

// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
    sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
    sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
    sys_chown, sys_break, sys_stat, sys_lseek, sys_gepid,.....
};

```

C 语言中可以通过函数指针对函数进行调用，那么可以考虑通过解析输入的命令字符串，然后在函数指针表中取出对应的函数并运行，这样主函数中就可以避免冗长的 switch 语句。

java 中其实有一个叫做反射的机制，它的主要用途就是将 java 类中的各种成分(如成员变量、方法、构造方法、包信息等)映射成一个个 java 对象。为此，可以专门设计一个类来存储指令的执行方法，主函数中只需要通过提取输入的命令然后利用反射获取这个类中对应的方法即可。以 touch 命令为例，这个命令有一个输入参数，格式为 touch [fileName]，下面结合代码讲述一个指令如何在主函数中调用。

首先，设计一个专门存储命令执行方法的类（这里定为 CommandManagement）

```

static class CommandManagement{
    public static void inputCommand(String[] input){
        // 解析输入的指令
        // 指令的参数不确定，不过数量一般不多，故这里用 switch 区分参数不同的指令
    }
}

```

```
switch (input.length) {
    case 1:
        // 流程相似，不再展示
        // .....
    case 2:
        try{
            // 通过反射获取指令对应的方法
            Method method=CommandManagement.class.getMethod(input[0],String.class);
            // 运行方法对象
            callFunc1(method,input);
        } catch (NoSuchMethodException e) {
            System.out.println("不存在相关指令，可用指令如下");
            View.help();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        break;
    case 3:
        // 流程相似，不再展示
        // .....
    default:
        }
}

// 0 参数指令
public static void callFunc0(Method method) throws Exception {
    method.invoke(null);
}

// 1 参数指令
public static void callFunc1(Method method, String[] parm) throws
Exception {
    //运行方法对象
    method.invoke(null, parm[1]);
}

// 2 参数指令
public static void callFunc2(Method method, String[] parm) throws
Exception {
    method.invoke(null, parm[1], parm[2]);
}

//=====具体命令=====
```

```
public static void touch(String fileName){
    String permissionFile=Utility.inputPermission();
    fileService.create(fileName,permissionFile);
}
}
```

对于下面的语句

```
Method method=CommandManagement.class.getMethod(input[0],String.class);
```

Input[0]为输入指令对应的方法名假如输入的指令为 rename，那么就会调用 rename 方法，不过一个方法还需要结合其传入参数来和其他同名方法进行区分。String.class 为传入参数的类型，这需要结合具体的方法的传入参数而定，如果目标方法为 method(String a,int b)，那么 getMethod 的参数就需要改成 (input[0], String.class, Int.class)。

这样，主函数直接调用 CommandManagement 的 inputCommand 方法即可，代码如下

```
while (true){
    System.out.print("[unlogin]");
    String nextLine = scanner.nextLine();
    String[] inputs = Utility.inputResolve(nextLine);
    //外围未登录者只有三个命令可以调用，因此使用 switch 影响不大
    switch (inputs[0]){
        case "help":
            //.....
        case "login":
            System.out.print("用户名: ");
            String username = scanner.nextLine();
            System.out.print("密码: ");
            String pwd = scanner.nextLine();
            login = userService.login(username, pwd);
            if(login){
                System.out.println("进入系统.....");
                while (login){
                    dirService.showPath();
                    String nextLine2 = scanner.nextLine();
                    String[] inputs2 =
Utility.inputResolve(nextLine2);
                    // 解析指令，通过反射实现(linux 源码中采用函数指针实现，原理相近)
                    // 只需要在 CommandManagement 中添加对应的静态函数即可
                    CommandManagement.inputCommand(inputs2);
                }
            }
    }
}
```

```
    }  
    break;  
    case "register":  
        //.....  
        break;  
    case "exit":  
        //.....  
        break;  
    default:  
    }  
}
```

这样，原本主函数中近 100 行的 switch 代码现在只需要调用一个类的方法就能达到相同的效果。另外，这样写也更容易添加新的指令，只需要在 CommandManagement 类中添加一个同名的方法即可。

4.2.5 路径解析

大部分指令的输入参数都带有一个路径比如 cd 指令，路径分为绝对路径和相对路径两种，主要目的还是标出指定文件(或目录)在文件系统中的位置。对于绝对路径，我假设为必须以“/”作为开头(如“/a/b”将从根目录下先寻找 a 目录再寻找 b 目录)，寻找文件时将从根目录的 fcb 开始向下寻找指定文件；对于相对路径，为了模仿 linux 的写法，路径的前缀可以加上一系列的“./”和“../”，前者表示当前目录，后者表示上一级目录，解析路径前先根据“./”和“../”将目录位置从当前目录转移到特定目录下，然后从该目录的 fcb 开始向下寻找指定文件。

后续的寻找则更简单，比如路径 a/b/c，这是一种相对路径的写法，代码中会利用“/”拆分路径字符串以便后续操作。首先需要寻找当前目录下的 a 目录，目录的 fcb 中有一个子文件节点列表，通过枚举这个列表可以找到 a 目录并进入。同理，枚举 a 目录 fcb 中的子文件节点列表可以找到 b 目录，进而寻找 c 目录(c 也可以是普通文件)。

下面为路径解析的具体代码

```
public FCB pathResolve(String path) {  
    path = path.trim();  
    FCB curDir = Memory.getInstance().getCurDir(); //当前所在的目录  
    FCB rootDir = Memory.getInstance().getRootDir(); //根目录
```

```
//判断是否直接切换到根目录
if("/") .equals(path)) {

Memory.getInstance().setCurDir(Memory.getInstance().getRootDir());
    return rootDir;
}
//判断是不是 ./
if("./" .equals(path) || "/" .equals(path)) {
    return curDir;
}
//判断是不是.. ../
if("../" .equals(path) || "/" .equals(path)) {
    //根目录无法再往父节点走了
    if(curDir != Memory.getInstance().getRootDir()) {
        //返回当前目录的父目录
        return curDir.getFather();
    }
    return rootDir;
}

FCB ansFCB=null; //指定文件的 fcb 通过这个变量返回
//判断是不是/开头，不是就从当前目录找
if(!path.startsWith("/")) {
    // 解析所有的 ./和../ 约定这两种字符只出现在路径的前缀
    while(path.startsWith("./") || path.startsWith("../")) {
        if(path.startsWith("./")) { // 从当前目录往下找
            path=path.substring(2);
        } else if(path.startsWith("../")) { // 从父目录往下找
            if(curDir!=rootDir) {
                curDir=curDir.getFather();
            }
            path=path.substring(3);
        }
    }
    if(path.equals("")) {
        return curDir;
    }
    ansFCB = curDir;
} else {
    //以/开头 从根目录逐层往下找
    path = path.substring(1);
    ansFCB = rootDir;
}
String[] splitDir = path.split("/");
```

```
for (String target : splitDir) {
    //找到指定的文件
    for (FCB child : ansFCB.getChildren()) {
        if (child.getFileName().equals(target)) {
            ansFCB = child;
            break;
        }
    }
}

if(!Objects.equals(ansFCB.getFileName(), splitDir[splitDir.length
- 1])){
    ansFCB=new FCB();
    ansFCB.setType(Constants.U_FILE);
}
return ansFCB;
}
```

cd 指令中调用到了这个方法，在类设计中，我将当前所在的目录 fcb 存在 Memory 全局对象中(上述代码第 3 行)，因此，只需要将路径解析获取到的 fcb 保存为 Memory 全局对象中的当前目录位置即可，具体代码如下

```
public Boolean cd(String path) {
    //解析路径
    FCB fcb = dirService.pathResolve(path);
    //null 不存在
    if(fcb.getType() == Constants.U_FILE){
        System.out.println("[error]: 目标目录不存在");
        return false;
    }else if(fcb.getType().equals(Constants.C_FILE)){
        //type N 不是目录文件
        System.out.println("[error]: 无法进入普通文件");
        return false;
    }else {
        //type D 切换到对应目录
        //判断权限
        int permission = Utility.checkPermission(fcb);
        if(permission == 0){
            System.out.println("[error]: 无权限");
            return false;
        }
        Memory.getInstance().setCurDir(fcb);
    }
}
```

```
    return null;
}
```

可以看出，cd 命令的流程主体部分还是一系列的异常判断，实际的切换操作只有 `Memory.getInstance().setCurDir(fcb)` 这一句。

4.2.6 文件大小更新

一个普通文件写入内容后，它的大小就会改变，具体体现在其 fcb 上。同时，与该文件级联的一系列父目录(直到根目录为止)也需要修改大小。由于整个文件系统的 fcb 可以形成树形结构，因此只需要更新从当前文件向上一直到根目录的路径上的所有 fcb 的大小即可(不包括根目录)。具体代码如下，下面的代码还适用于删除文件的情景

```
public void updateSize(FCB fcb, Boolean isAdd, int newAdd) {
    FCB temp = fcb.getFather();
    // 根目录大小不需要修改，因为整个文件树就是从根目录衍伸而出的
    // 而且也没有移动根目录的需求
    while (temp != Memory.getInstance().getRootDir()) {
        // 自底向上修改父目录的大小
        int size = temp.getIndexNode().getSize();
        if (isAdd) {
            if (newAdd == -1) { // 代表覆盖写操作
                // 增加目录大小
                temp.getIndexNode().setSize(size +
fcb.getIndexNode().getSize());
            } else { // newAdd 非-1，此时代表追加内容的大小
                temp.getIndexNode().setSize(size + newAdd);
            }
        } else { // 删除文件
            temp.getIndexNode().setSize(size -
fcb.getIndexNode().getSize());
        }
        temp = temp.getFather();
    }
}
```

4.2.7 使用正则表达式约束权限的设置

模仿 linux 中对权限的设定，本文件系统中的权限也为类似 Drwxrwx，第一位为文件类型(目录为 D，普通文件为 C)，接下来的三位是文件创建者的权限(r：可读，w：可写，x：可运行)，最后三位是其他用户的权限，如果对应为止没有权限则可以用“-”替代。

用户将以字符串的形式输入权限，这就难免会产生异常输入，主要为权限长度不对(必须为 6 位)、使用了除“r、w、x、-”之外的其他字符两类问题。针对第一个问题，直接判断输入字符串的长度即可。针对第二个问题，直接通过多个 if-else 语句进行解析的写法会非常麻烦。比如下面代码

```
if(permission[0]=="r" || permission[3]=="-"){
    if(permission[1]=="w" || permission[4]=="-"){
        if(permission[2]=="x" || permission[5]=="-"){
            //.....
        }
    }
}
```

事实上，对于这类固定格式的字符串匹配的问题，可以利用正则表达式作为解决方案。首先明确权限字符串的格式，第 1、4 位必须为“r”或“-”，第 2、5 位必须为“w”或“-”，第 3、6 位必须为“x”或“-”。于是，正则表达式可以设计为“((r|-)(w|-)(x|-)){2}”，“{2}”代表前面的格式一共出现两次。

于是，输入权限的判定代码可以简化为下面的形式

```
if(!permission.matches("((r|-)(w|-)(x|-)){2}")){
    System.out.println("[error] 权限格式不符合要求");
    return false;
}
```

5. 运行结果

5.1 运行环境

硬件：

CPU	英特尔 11 代芯片 i5-1135G7@2.40Hz
内存	16.00GB

操作系统：windows10

编码工具：jetbrain-IDEA(2022.2)

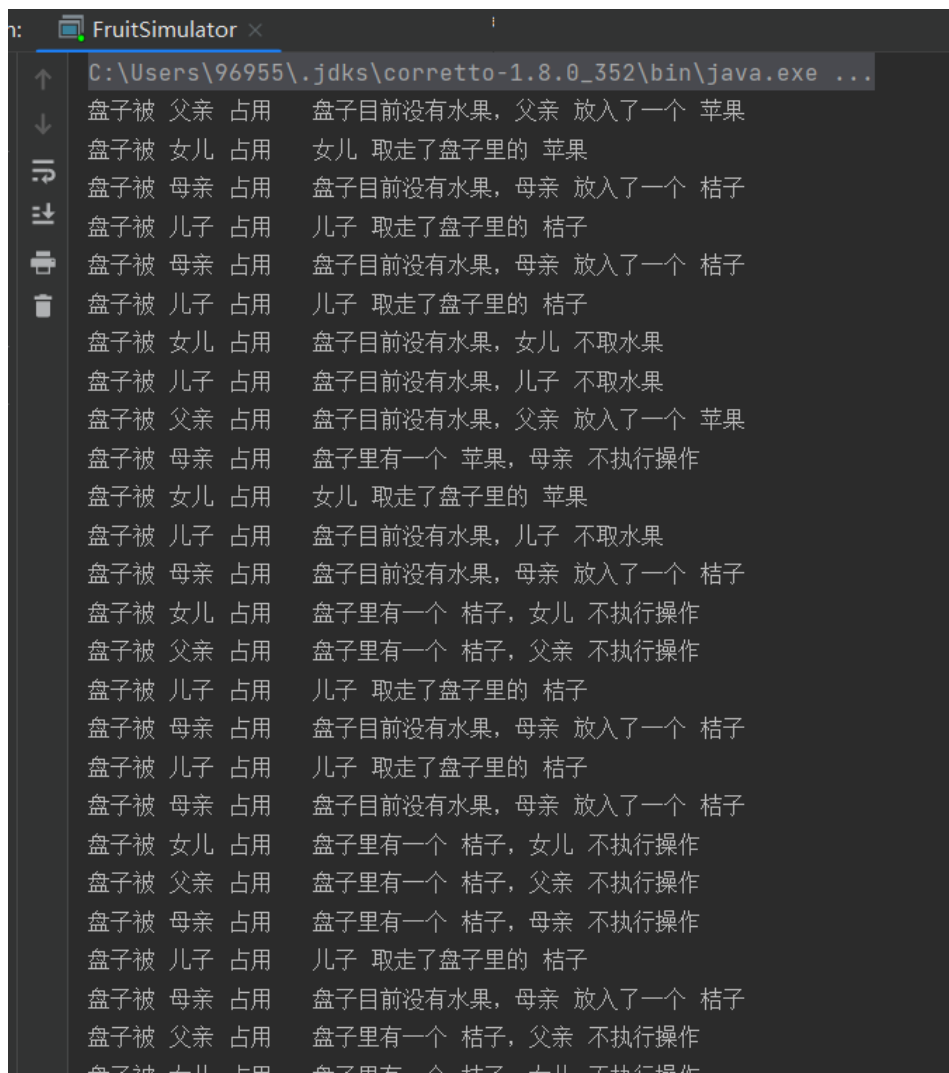
5.2 服务模式

所有数据封装成 Disk 对象并保存在/resource/diskObject/system.data 文件中。当 diskObject 目录下没有该文件时，系统会提示用户创建一个新的 system.data 文件。

5.3 运行结果

5.3.1 “吃水果”问题

运行结果如下图所示



```
C:\Users\96955\.jdk\corretto-1.8.0_352\bin\java.exe ...
盘子被 父亲 占用  盘子目前没有水果, 父亲 放入了一个 苹果
盘子被 女儿 占用  女儿 取走了盘子里的 苹果
盘子被 母亲 占用  盘子目前没有水果, 母亲 放入了一个 桔子
盘子被 儿子 占用  儿子 取走了盘子里的 桔子
盘子被 母亲 占用  盘子目前没有水果, 母亲 放入了一个 桔子
盘子被 儿子 占用  儿子 取走了盘子里的 桔子
盘子被 女儿 占用  盘子目前没有水果, 女儿 不取水果
盘子被 儿子 占用  盘子目前没有水果, 儿子 不取水果
盘子被 父亲 占用  盘子目前没有水果, 父亲 放入了一个 苹果
盘子被 母亲 占用  盘子里有一个 苹果, 母亲 不执行操作
盘子被 女儿 占用  女儿 取走了盘子里的 苹果
盘子被 儿子 占用  盘子目前没有水果, 儿子 不取水果
盘子被 母亲 占用  盘子目前没有水果, 母亲 放入了一个 桔子
盘子被 女儿 占用  盘子里有一个 桔子, 女儿 不执行操作
盘子被 父亲 占用  盘子里有一个 桔子, 父亲 不执行操作
盘子被 儿子 占用  儿子 取走了盘子里的 桔子
盘子被 母亲 占用  盘子目前没有水果, 母亲 放入了一个 桔子
盘子被 儿子 占用  儿子 取走了盘子里的 桔子
盘子被 母亲 占用  盘子目前没有水果, 母亲 放入了一个 桔子
盘子被 女儿 占用  盘子里有一个 桔子, 女儿 不执行操作
盘子被 父亲 占用  盘子里有一个 桔子, 父亲 不执行操作
盘子被 母亲 占用  盘子里有一个 桔子, 母亲 不执行操作
盘子被 儿子 占用  儿子 取走了盘子里的 桔子
盘子被 母亲 占用  盘子目前没有水果, 母亲 放入了一个 桔子
盘子被 父亲 占用  盘子里有一个 桔子, 父亲 不执行操作
盘子被 女儿 占用  盘子里有一个 桔子, 女儿 不执行操作
```

图 5-1 “吃水果”问题测试

5.3.2 文件系统

5.3.1 注册、登录、退出

用户信息存储在 disk 对象的 userMap 中，不可出现重名用户。测试结果如下

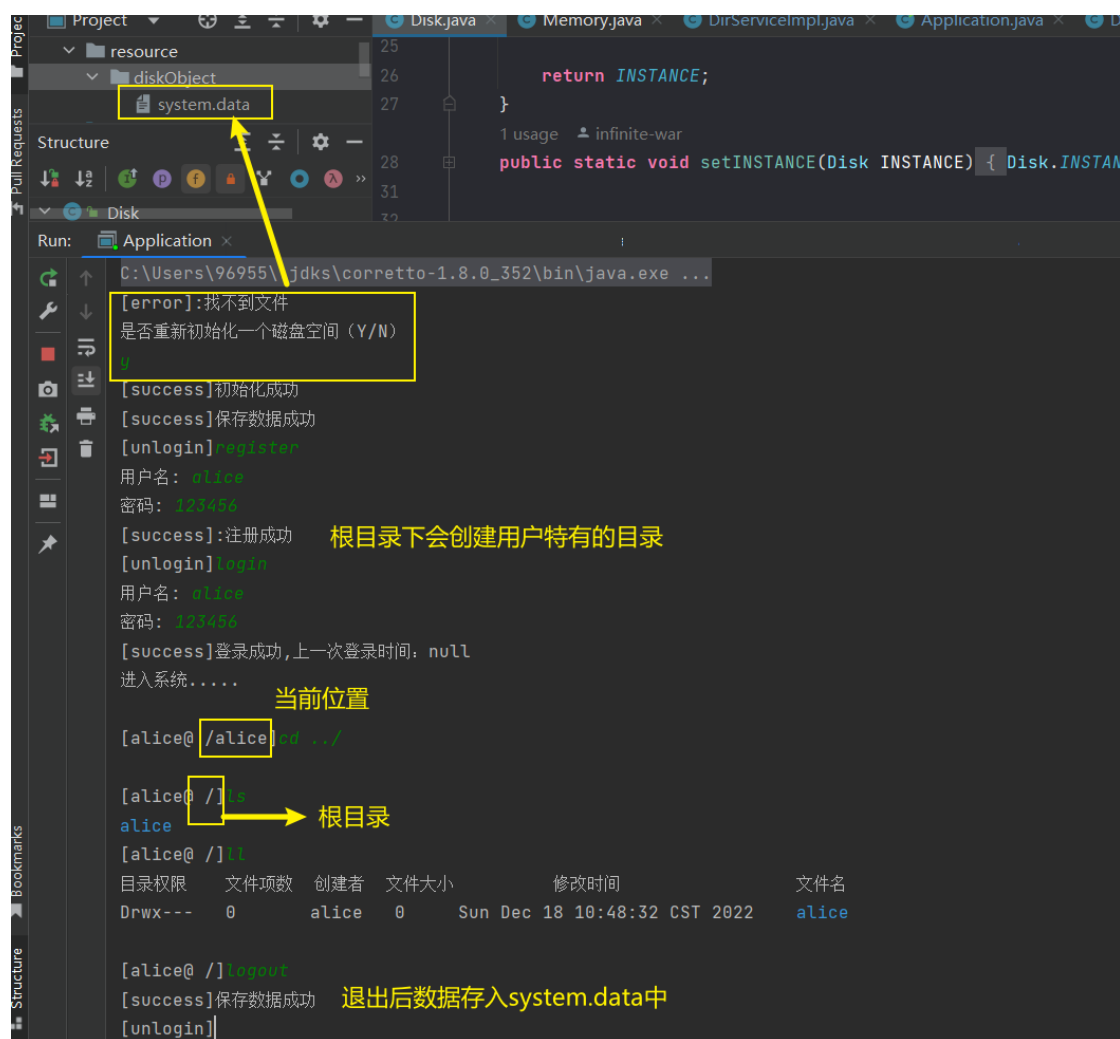


图 5-2 测试登录

下图测试重启后登录

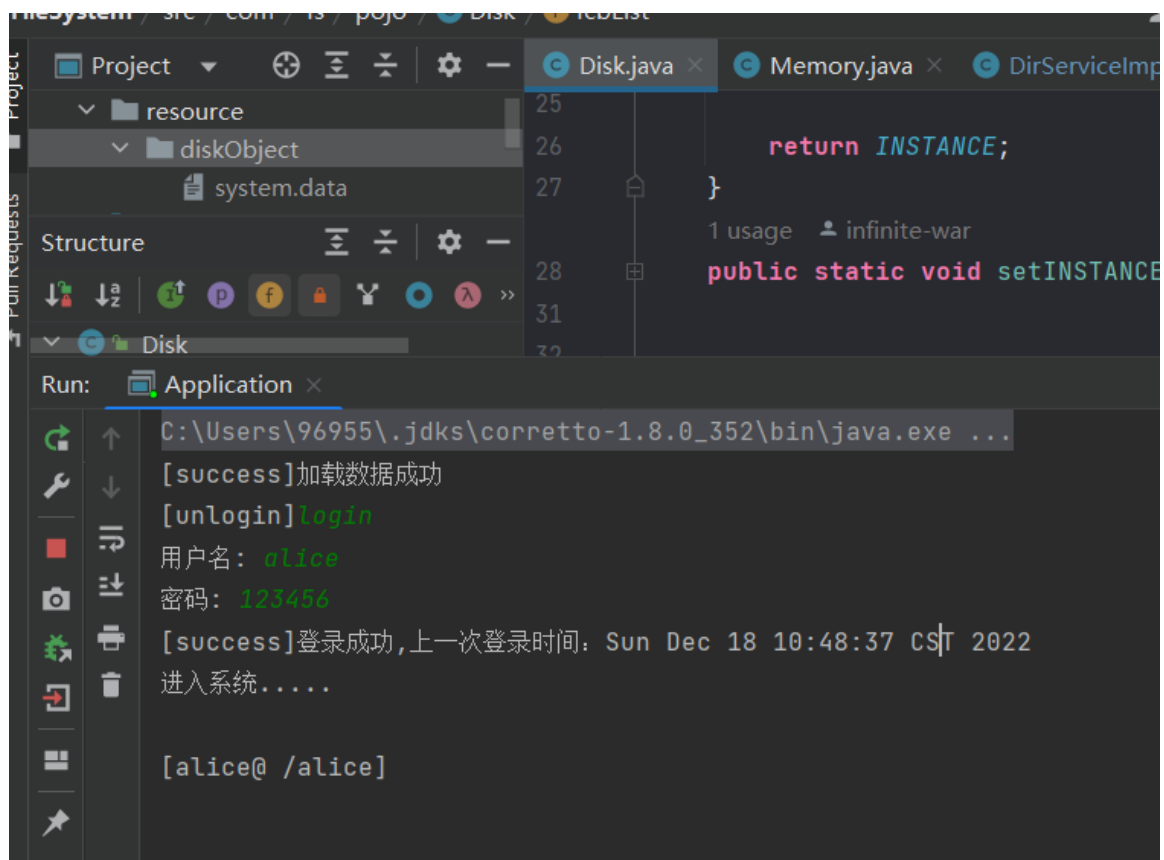


图 5-3 测试重启程序后登录

5.3.2 基本信息展示(ls、ll、help、man、bitmap)

首先测试基础的 ll、ls、help 指令，如下图



```
[alice@ /alice]cd ../  
[alice@ /]ll          显示详细信息  
目录权限  文件项数  创建者  文件大小      修改时间          文件名  
Drwx---   0        alice    0      Sun Dec 18 10:48:32 CST 2022  alice  
[alice@ /]ls          显示简略信息  
alice  
[alice@ /]help          显示帮助  
====command=====  
login  登录(先退出当前用户)  
register 注册(先退出当前用户)  
logout 退出  
mkdir [dirName] 创建目录  
cd [fileName] 切换到指定目录  
touch [fileName] 创建文件  
chmod [fileName] 修改文件权限  
rename [filePath] [newName] 文件重命名  
open [fileName] 打开文件  
close [fileName] 关闭文件  
read [fileName] 读文件  
write [fileName] 写文件  
delete [fileName] 删除文件  
show_open 显示打开的文件  
bitmap 显示位示图  
ls 显示目录文件名  
ll 显示当前目录下所有文件详细信息  
man [command] 显示指令详细信息  
====command=====
```

图 5-4 测试基本信息展示 1

5.3.3 创建目录、普通文件(mkdir、touch)

下图为测试 mkdir 指令的过程

```
[alice@ /alice]mkdir dira
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
afoefaeafe
[error] 权限必须为6位
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
rwxrwx
[error] 权限格式不符合要求
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
rwxrwx
[success]: 创建目录成功

[alice@ /alice]ls
dira
[alice@ /alice]mkdir o/v
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
rwxrwx
不能包含特殊字符(/,\)

[alice@ /alice]mkdir dira
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
rwxrwx
[error]: 目录名重复,请重新命名

[alice@ /alice]
```

图 5-6 测试 mkdir

下图为测试 touch 指令的过程

```
[alice@ /alice]mkdir dir0
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
afae fae fae
[error] 权限必须为6位
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwxrwxrwx
[error] 权限格式不符合要求
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwxrwxrwx
[success]: 创建目录成功

[alice@ /alice]ls
dir0
[alice@ /alice]mkdir a/v
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwxrwxrwx
不能包含特殊字符(/,\)

[alice@ /alice]mkdir dir0
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwxrwxrwx
[error]: 目录名重复,请重新命名

[alice@ /alice]
```

图 5-7 测试 touch

5.3.4 切换目录(cd)

下图为测试 cd 指令的过程

```
[alice@ /]ls
alice
[alice@ /]cd alice

[alice@ /alice]cd ../

[alice@ /]cd alice/dira

[alice@ /alice/dira]cd ../

[alice@ /alice]mkdir dirb
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwxrwx
[success]: 创建目录成功

[alice@ /alice]cd dira

[alice@ /alice/dira]cd ../dirb

[alice@ /alice/dirb]cd ../dira
[error]: 目标目录不存在

[alice@ /alice/dirb]
```

图 5-8 测试 cd

5.3.5 修改文件权限(chmod)

首先查看当前用户 alice 的个人目录的权限

```
[alice@ /]ll
目录权限 文件项数 创建者 文件大小 修改时间 文件名
Drwx--- 6 alice 0 Sun Dec 18 10:48:32 CST 2022 alice
```

图 5-8 alice 的个人用户目录

该权限末三位为“---”，说明其他用户没有任何权限，而且不可访问。下面登录用户 bob 尝试访问 alice 的个人目录


```

[success]加载数据成功
[unlogin]register
用户名: bob
密码: 123456
[success]:注册成功
[unlogin]login
用户名: bob
密码: 123456
[success]登录成功,上一次登录时间: null
进入系统.....

[bob@ /bob]cd ../

[bob@ /]ll
目录权限  文件项数  创建者  文件大小      修改时间      文件名
Drwx---   5        alice   0      Sun Dec 18 10:48:32 CST 2022  alice
Drwx---   0        bob     0      Sun Dec 18 11:50:44 CST 2022  bob

[bob@ /]cd alice  若未经权限设置, 用户不能访问其他用户的个人目录
[error]: 无权限

[bob@ /]

```

图 5-9 测试无权限访问

修改 bob 的个人目录的权限, 看看 alice 能不能访问 bob 的个人目录

```

[bob@ /]chmod bob
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】【后三位表示其他用户】
rwxrwx
确认修改该文件的权限? (Y/N)
y
[success]: 权限修改成功

[bob@ /]ll
目录权限  文件项数  创建者  文件大小      修改时间      文件名
Drwx---   5        alice   0      Sun Dec 18 10:48:32 CST 2022  alice
Drwxrwx   0        bob     0      Sun Dec 18 11:50:44 CST 2022  bob

[bob@ /]logout
[success]保存数据成功
[unlogin]login
用户名: alice
密码: 123456
[success]登录成功,上一次登录时间: Sun Dec 18 11:45:26 CST 2022
进入系统.....

[alice@ /alice]cd ../bob

[alice@ /bob]

```

图 5-10 测试 alice 访问 bob 的个人目录

同理，修改 alice 的个人目录的权限，看看 bob 能不能访问 alice 的个人目录

```
[alice@ /]chmod alice
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwxrwx
确认修改该文件的权限? (Y/N)
Y
[success]: 权限修改成功

[alice@ /]logout
[success]保存数据成功
[unlogin]login
用户名: bob
密码: 123456
[success]登录成功,上一次登录时间: Sun Dec 18 11:50:49 CST 2022
进入系统.....

[bob@ /bob]cd ../alice

[bob@ /alice]
```

图 5-11 测试 bob 访问 alice 的个人目录

下面再测试一下修改普通文件的权限，首先 bob 创建一个普通文件 bob.txt(权限“rwx——”)，看看 alice 能不能打开这个文件

```
[bob@ /bob]touch bob.txt
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 【前三位表示自己】 【后三位表示其他用户】
rwx---
[success]: 创建文件成功

[bob@ /bob]logout
[success]保存数据成功
[unlogin]login
用户名: alice
密码: 123456
[success]登录成功,上一次登录时间: Sun Dec 18 11:53:41 CST 2022
进入系统.....

[alice@ /alice]cd ../bob

[alice@ /bob]open bob.txt
[error]: 无权限

[alice@ /bob]
```

图 5-12 测试 alice 打开 bob 创建的文件

回到 bob 账号，将 bob.txt 的权限修改为 “rwx-wx”，看看 alice 能不能读 bob.txt

```
[bob@ /bob]chmod bob.txt
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
rwx-wx
确认修改该文件的权限? (Y/N)
Y
[success]: 权限修改成功

[bob@ /bob]login
[error]:请先退出登录

[bob@ /bob]logout
[success]保存数据成功
[unlogin]login
用户名: alice
密码:
[error]:用户名或密码错误
[unlogin]login
用户名: alice
密码: 123456
[success]登录成功,上一次登录时间: Sun Dec 18 11:55:16 CST 2022
进入系统.....

[alice@ /alice]cd ../bob

[alice@ /bob]open bob.txt
[success]: 打开成功

[alice@ /bob]read bob.txt
[error]: 无读权限

[alice@ /bob]
```

图 5-13 测试 alice 读 bob 创建的文件

5.3.6 文件重命名(rename)

下图为修改文件名测试

```
[alice@ /alice]ll
目录权限  文件项数  创建者  文件大小      修改时间      文件名
Drwxrwx   0      alice   0      Sun Dec 18 11:08:45 CST 2022  dira
Crwxrwx   0      alice   0      Sun Dec 18 11:09:31 CST 2022  a.txt
Crwxrwx   0      alice   0      Sun Dec 18 11:09:40 CST 2022  b.7z
Crwxrwx   0      alice   0      Sun Dec 18 11:09:48 CST 2022  c.exe
Crwxrwx   0      alice   0      Sun Dec 18 11:09:56 CST 2022  d.rar

[alice@ /alice]rename a.txt ssss.txt
[success]: 文件名修改成功

[alice@ /alice]ls
dira ssss.txt b.7z c.exe d.rar
[alice@ /alice]rename ssss.txt uuu.cpp
[success]: 文件名修改成功

[alice@ /alice]ls
dira uuu.cpp b.7z c.exe d.rar
[alice@ /alice]
```

图 5-14 测试 rename

5.3.7 打开、关闭文件、展示处于打开状态的文件(open、close、show_open)

下图为对文件进行打开、关闭操作的测试

```
[alice@ /alice]show_open
<没有打开的文件>

[alice@ /alice]open a.txt
[success]: 打开成功

[alice@ /alice]show_open
a.txt
[alice@ /alice]logout          退出时会提示有文件未关闭
当前有文件未关闭, 强制退出将关闭所有的文件, 确认退出? (Y/N)
y
[success]: 文件关闭成功
关闭了文件/alice/a.txt
[success]保存数据成功
[unlogin]login
用户名: alice
密码: 123456
[success]登录成功,上一次登录时间: Sun Dec 18 12:00:46 CST 2022
进入系统.....

[alice@ /alice]open a.txt
[success]: 打开成功

[alice@ /alice]close a.txt
[success]: 文件关闭成功

[alice@ /alice]
```

图 5-15 测试打开、关闭文件

5.3.8 读写、删除文件(read、write、delete)

首先针对已有的普通文件 a.txt 进行读写测试，如下图

```
[alice@ /alice]open a.txt
[success]: 打开成功

[alice@ /alice]read a.txt
-----BEGIN-----
<----EMPTY FILE---->
-----END-----

[alice@ /alice]write a.txt
请输入要写入的内容（以$$结尾）：
int main{
    printf("-----");
}
$$
[success]: 写入成功!

[alice@ /alice]read a.txt
-----BEGIN-----
int main{
    printf("-----");
}
-----END-----

[alice@ /alice]bitmap
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

分配了5个盘块

图 5-16 测试读写 a.txt 文件

创建一个文件 b.txt，写入内容，查看位图的变化情况，见下图

图 5-17 测试读写 b.txt 文件

删除 a.txt，查看位图的变化情况，见下图

```
[alice@ /alice]delete a.txt
[error]: 文件被打开,请先关闭

[alice@ /alice]close a.txt
[success]: 文件关闭成功

[alice@ /alice]delete a.txt
确认删除该文件? (Y/N)
y
[success]: 删除成功

[alice@ /alice]bitmap
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

图 5-18 测试删除 a.txt 文件

新建一个文件 cc.txt，写入少量内容，查看位图的变化情况，见下图

```
[alice@ /alice]touch cc.txt
请输入文件权限(必须为6位): r:读 w:写 x:执行 -:否定对应位置的权限 [前三位表示自己] [后三位表示其他用户]
rwxrwx
[success]: 创建文件成功

[alice@ /alice]open cc.txt
[success]: 打开成功

[alice@ /alice]write cc.txt
请输入要写入的内容(以$$结尾):
aaaaaaaaaaaaaaaa
$$
[success]: 写入成功!

[alice@ /alice]bitmap
1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

图 5-19 测试写入 cc.txt 文件

新建一个文件 ddd.txt，写入少量内容，查看位图的变化情况，见下图

图 5-20 测试写入 ddd.txt 文件

由上图测试可以看出，文件所占用的盘块可以是不连续的，因为一个文件占用的盘块之间采用链表进行连接。下面删除 `ddd.txt` 再次查看位图的变化情况

[illegible]

图 5-21 测试删除 ddd.txt 文件

6. 调试和改进

6.1 持久化存储

一开始测试的时候，如果将已有的 system.data 放入 resource/diskObject 中，会报序列号错误，如下图

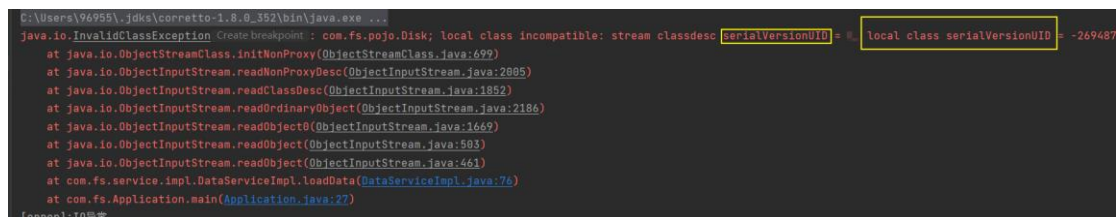


图 6-1 反序列化报错

查阅资料后了解到，一个实现了序列化接口的 java 类在生成对象时会赋予一个随机的序列号(serialVersionUID)，这个序列号会随对象一起序列化并写入指定的文件中。读取文件中的对象时需要将对象解序列化并赋值到给定的类中，这时候类中必须要有与读取对象相同的序列号。因此，需要给磁盘类假如一个固定的序列号，如下

```
public class Disk implements Serializable {  
    private static final long serialVersionUID = 1L;  
    //.....  
}
```

6.2 cd 导致目录类型变化

登录用户后，首先执行“cd ../”指令，目录跳转成功，然而打开后用户目录的文件类型变成了‘U’（未定义类型），后续无法再通过 cd 指令进入用户目录。

主要的问题出现在 cd 的流程中，经过 debug，发现问题出在以下代码中

```
public FCB pathResolve(String path) {  
    //.....  
  
    FCB ansFCB=null;  
    //判断是不是/开头 不是就从当前目录找
```

```

    if(!path.startsWith("/")){
        // 解析所有的 ./和../ 约定这两种字符只出现在路径的前缀
        // .....

        ansFCB = curDir;
        ansFCB.setType(Constants.U_FILE); // 这一部分直接修改了目录类型
    }else {
        //以/开头 从根目录逐层往下找
        path = path.substring(1);
        ansFCB = rootDir;
    }
    String[] splitDir = path.split("/");
    for (String target : splitDir) {
        //找到目标文件所在目录
        for (FCB child : ansFCB.getChildren()) {
            if (child.getFileName().equals(target)) {
                ansFCB = child;
                break;
            }
        }
    }
    return ansFCB;
}

```

java 本身没有显式的指针操作，不过上述代码中 ansFCB=curDir 的操作传递的是对象的引用，并没有为 ansFCB 额外复制一个对象，因此对 ansFCB 的修改会直接改变原文件系统的内容。

作为该进，需要删除 ansFCB.setType(Constants.U_FILE)，将是否找到目标文件的判断留到最后。如果经过一番查找，ansFCB 的文件名与要查找的文件名不同，则说明没有找到指定的文件。另外，在最后针对 splitDir 的循环中，需要设置布尔变量查看路径的每一部分是否都可以查找到。比如路径 “a/b/c”，如果找到了目录 a，再进一步却没有找到目录 b，则返回未找到文件的信息。修改后的关键代码如下

```

public FCB pathResolve(String path) {
    //.....
    FCB ansFCB=null;
    //判断是不是/开头 不是就从当前目录找
    if(!path.startsWith("/")){
        // .....
        ansFCB = curDir;
    }else {

```

```
//以/开头 从根目录逐层往下找
path = path.substring(1);
ansFCB = rootDir;
}
String[] splitDir = path.split("/");
for (String target : splitDir) {
    boolean f=false;
    //找到目标文件所在目录
    for (FCB child : ansFCB.getChildren()) {
        if (child.getFileName().equals(target)) {
            ansFCB = child;
            f=true;
            break;
        }
    }
    if(!f){
        ansFCB=new FCB();
        ansFCB.setType(Constants.U_FILE);
    }
}
//判断有没有找到指定文件
if(!Objects.equals(ansFCB.getFileName(), splitDir[splitDir.length
- 1])){
    ansFCB=new FCB();
    ansFCB.setType(Constants.U_FILE);
}
return ansFCB;
}
```

6.3 权限的判断

权限分为 r(可读)、w(可写)、x(可执行) 三种，可以用三位二进制表示，回顾代码后我发现很多地方的权限判断采用了硬编码的方式，这样不利于代码的维护，比如

```
//判断权限
//checkPermission 将 rwx 转成十进制
int permission = Utility.checkPermission(fcb);
if(permission != 8){
    System.out.println("[error]: 无读权限");
}
```

```
    return false;
}
```

这里的 8 代表读权限，三位二进制一共就八种组合方法，可以在编写在常数类中。比如，只读权限二进制为 100，转为十进制为 8；只写权限二进制位 010，转为十进制位 4；读写权限二进制为 110，转为十进制为 10。

不过直接用“==”来匹配文件计算得到的权限会出现一个问题，假如对某件进行某一种操作，用户只需要读权限，但是用户拥有所有权限，这时候采用 `permission==Constraint.READ` 语句进行判断就会出错。这时候可以采用与运算的方式进行判断，通过将 `permission` 和 `Constraint.READ_AND_WRITE` (二进制为 110) 进行二进制与运算，即可得知该用户有没有读写权限，另外 7 种权限同理。

修改后的代码如下

```
if((permission & Constants.READ)== Constants.READ){
    System.out.println("[error]: 无读权限");
    return false;
}
```

6.4 man 方法修改

测试时发现 `man` 方法也写了一长串的 `switch` 语句，为了简化 `man` 方法的调用以及便于后续 `man` 方法的内容维护，可以借鉴主函数中利用反射机制替代 `switch` 的写法。`man` 方法的具体流程修改为下面代码

```
// 主函数中的 man 指令
public static void man(String command){
    View.getCommandDetail(command);
}

// man 指令的具体流程
public static void getCommandDetail(String command){
    try{
        // 利用反射机制获取方法对象
        Method method=View.class.getMethod(command);
        callFunc(method);
    } catch (NoSuchMethodException e) {
        System.out.println("不存在相关指令，可用指令如下");
        View.help();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

```
    }  
}  
// 执行方法对象  
public static Void callFunc(Method method) throws Exception{  
    return (Void)method.invoke(null);  
}  
// =====指令的详细内容(可通过反射获取的方法)=====  
// .....  
public static void ls(){  
    System.out.println("命令 ls");  
    System.out.println("格式: \n\tls");  
    System.out.println("显示当前目录下所有文件的简要信息");  
}
```

7. 心得和结论

7.1 结论和体会

7.1.1 “吃水果”问题

7.1.1.1 贡献

通过多线程模拟了“吃水果”问题，代码可以进行扩展，可以添加更多家庭成员，模拟更复杂一些的“吃水果”场景。

7.1.2 文件系统

7.1.2.1 贡献

本文件系统实现了基本的文件操作(如创建、读、写、删除、打开、关闭)和目录操作(如切换目录、目录大小级联更新)，同时加入了用户和文件权限相关的内容。另外，本系统采用持久化磁盘对象的方式来保存整个文件系统的信息。

7.1.2.2 系统优点

- ①用户界面友好，用户通过控制台可以快速获取帮助信息和系统处理提示。
- ②数据结构设计合理，本系统的数据结构设计主要借鉴了 linux 文件系统的设计，
- ③用户可以查看磁盘的占用状况。
- ④利用 Memory 全局对象简单模拟了实际计算机系统中内存和磁盘的交互。
- ⑤利用按位与操作和正则表达式简化了权限部分的流程。

7.1.2.3 系统缺点

- ①目录的级联删除没有实现。
- ②没有实现 GUI 界面。
- ③文件只有覆盖写和追加写操作，没有修改功能。
- ④无法调用软件打开本项目中的文件，主要还是因为本项目的 FCB 跟操作系统中的 FCB 不同。记事本软件之所以能打开 windows 下的 txt 文件，是因为 txt 文件已经有一套成型的 FCB 数据结构，记事本软件基于该 FCB 实现文件的打开与关闭。

7.1.2.4 体会

①本次项目目录设计时借鉴了平时写 web 项目的经验，之前利用 springboot 编写 web 项目时，项目大致会分成 pojo、service、controller 三层，pojo 存储实体类、service 存储业务代码、controller 存储路由信息。本项目保留了前两者，将数据和操作分离，这样更容易对业务代码进行分模块设计，本项目的 pojo 层含有基本的数据结构，service 层含有普通文件操作模块、目录文件操作模块、用户模块、数据存储模块、磁盘操作模块五个部分，controller 层就是主函数。另外，service 需要先编写接口，再去实现接口，这样一方面便于其他部分调用 service 层的代码，另一方面可以先通过接口规划好要实现的方法，然后实现接口时可以多一份约束，不需要考虑太多东西。

②采用了 java 反射机制处理输入的指令，主函数的流程非常简单，修改指令对主函数没有任何影响，代码整体的耦合度降低。这一部分也是我第一次具体体会到 java 反射机制的有用之处，原来一直没有想到反射机制的应用场景，但是当我看到我编写了一大串的 switch 语句后，我觉得如果 linux 使用 switch 处理指令的话，那源代码的主函数一定非常特别的长，因为 linux 的指令远远不止 10 个。以此为契机，我查阅了许多文章并结合着看了 linux 源码，才了解到 linux(还有其他操作系统)底层的 shell 程序大都是利用 C 的函数指针来实现的，java 没有显式的指针操作，但是我发现 java 的反射机制可以实现相近的功能，于是立刻修改了主函数并第一次体会到了反射机制的优点。

③文件系统的整体操作其实都和树这一数据结构有关，每个文件都有一个 FCB 存储基本数据，根据树的特性，FCB 中还存储了父节点和子节点列表，通过父节点和子节点列表这两个信息可以唯一确定一个 FCB，这样同一目录下不会出现重名文件，但是不同目录下可以出现重名文件。另外，项目中还额外设置了一个 Memory 全局对象来存储当前所在目录的 FCB。这个全局对象再加上树的操作可以简化很多命令的流程设计，首先是 cd 指令，它的主要流程其实就是根据路径找到指定的目录，然后将该目录的 FCB 赋值给 Memory 全局对象，这样目录的切换就完成了。然后是 pwd 指令，一个文件的位置其实就是从根节点到当前文件的一段路径，而且根据树的特性，这个路径没有回路，因此，可以从 Memory 全局对象获取到当前目录的 FCB，然后向父节点递归(直到根目录位置)，递归的过程中将路径上的目录名通过头插法的方式组合成一段路径字符串即可。

④通过本次文件系统的学习与设计，我实际体会到了从磁盘向上抽象出一个文件系统的过程，具体参照了 linux0.11 源码。首先是将磁盘划分出少部分的空间用于存储 FCB、User 这类具有信息管理性质的数据结构，然后大部分的磁盘空间可以视为空闲盘块，为了更合理地为文件分配磁盘的空闲空间，有必要将磁盘空间离散化成多个小盘块，通过为文件分配一些小盘块可以更充分地利用磁盘空间，再利用链表串联一个文件的盘块，使得文件的内容不必连续化存储，同时利用位图管理盘块的占用情况，可以提高盘块的搜索效率。到了这里就基本完成了从磁盘抽象出文件系统的前期工作，后续将各个文件信息组织成树的结构就是③中的内容了。

⑤设计项目的过程中最常读的还是 linux 的源码，现在的 linux 源码行数特别大(大于 10 万行)，不利于初学者的学习，不过网上有 linux0.11 版本的备份，这是非常早期的版本，总体代码在 2 万行之间，而且可以在现在的 linux 平台上编译运行，其中文件系统的代码写在项目的/fs 目录下。即使代码量少了很多，读起来还是十分吃力，因此我选择先读相关的文章，这里着重阅读了《像小说一样品读 Linux0.11 核心源码》[4]系列文章，它以 linux0.11 为参考

对象，从 linux 启动前的各种初始化操作开始逐步讲解到 linux 的 shell 程序，其中操作系统的许多概念基本都有涉及到，受益颇丰，而且根据这个系列文章我也是第一次具体看到 linux 启动前的汇编操作代码(在 linux0.11 源码的 boot 目录下)。事实上，linux 源码的其他部分也编写了大量的汇编代码，是 C 语言和汇编代码结合编写的一个很好的案例。自此，操作系统在我心中开始变得不再像一个大黑箱了，同时编写文件系统的思路也变得更为清晰。

⑥本次项目整体开发较为顺利，主要还是因为本项目的数据结构参照了 linux 文件系统的结构，这使得业务流程的编写没遇到特别严重的问题。linus 曾说过：“烂程序员关心的是代码。好程序员关心的是数据结构和它们之间的关系。”这一点在本次项目中确实有所体会，前期良好的数据结构设计确实可以让后续代码不用考虑太多交互的情况，因为这些情况大多数在数据结构的设计阶段就考虑好了。

7.2 进一步改进方向

①考虑文件目录删除和子文件权限之间的问题。比如用户 a 想删除目录 b，但是用户 a 对目录 b 下的部分子文件没有任何权限，那就会导致删除失败。上述问题尚未解决。

②虽然设置了 Memory 全局对象用来充当内存，但是本系统没有实现内存的存储管理，而是将其放到了磁盘中，后续可以考虑扩大磁盘的空间，然后 Memory 中也加入 FAT 用来管理内存空间。

③考虑如何像 vim 一样在控制台层面打开一个文件。换句话说，可以将本项目的 FCB 向 linux 的 FCB 靠近。

④可以优化以下根目录下的目录分配，比如 linux 根目录下就分出了 usr、bin、dev 等目录，功能划分更为明确。

7.3 分析设计方案对系统安全的影响

①未设置 root 管理员，普通用户除了文件权限的限制几乎没有其他约束。

②追加写的过程中，程序没有利用好链表末尾节点可能空余的盘块空间，而是直接获取新的盘块直接拼接到链表的末尾。这样容易造成磁盘空间的浪费。

主要参考文献

- [1] Linux 内核—文件系统详解 <https://zhuanlan.zhihu.com/p/505338841>，访问日期：

2022 年 11 月 28 日

- [2] 汤小丹、梁红兵、哲凤屏, 计算机操作系统(第四版) 西安电子科技大学出版社, 2014 年 5 月
- [3] linux0.11 源码(附带注释)<https://github.com/beride/linux0.11-1>, 访问日期 2022 年 12 月 6 日
- [4] 像小说一样品读 Linux 0.11 核心代码 <https://github.com/sunym1993/flash-linux0.11-talk>, 访问日期: 2022 年 11 月 28 日

考核成绩评定表

指导教师考核成绩	
答辩成绩	
总成绩	

签字：

年 月 日

