SmartFileSystem

刘俊伟 18302010042

一、系统结构

系统结构UML图

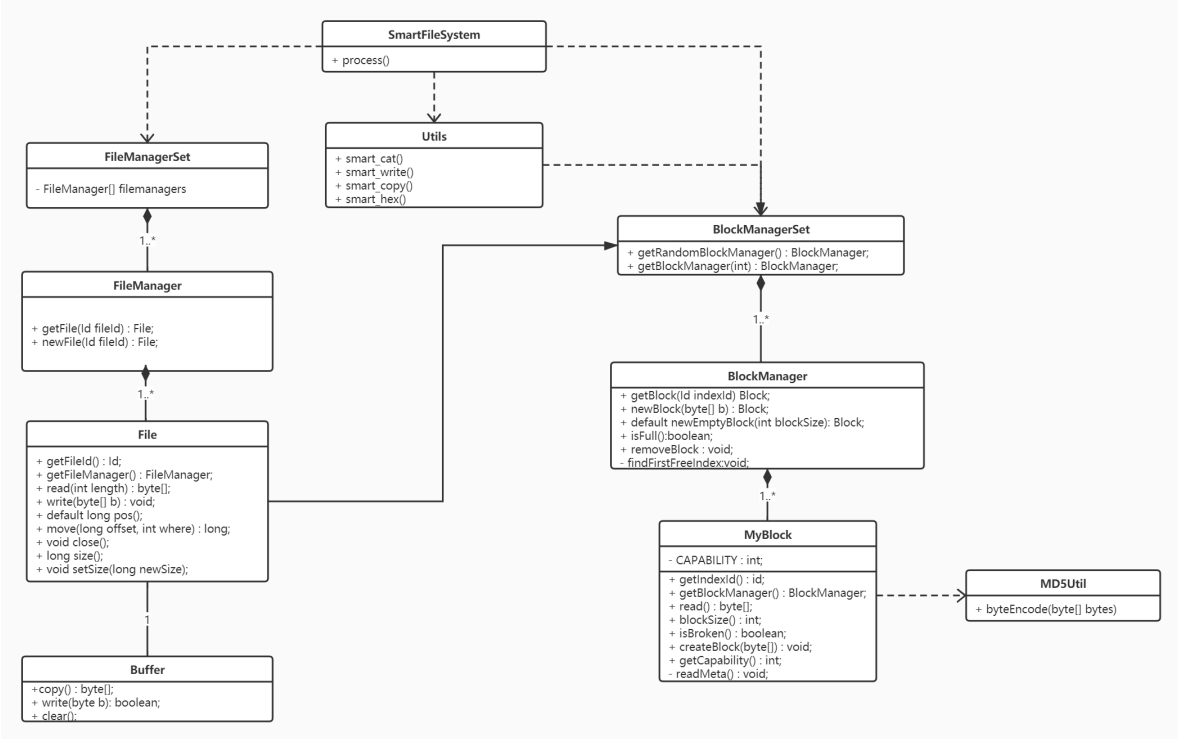

图1 SmartFileSystem结构

二、部分实现细节

1. 控制台程序

```
D:\DevelopTool\Java\jdk1.8.0_191\bin\java.exe ...
Welcome to the SmartFileSystem!
You have 5 file managers to use, and you can refer them by fm1 - fm5
SmartFS >> create fm1 file1
```

图2 控制台程序

- 控制台程序通过SmartFileSystem类实现，通过读取并解析用户的指令执行对应的文件系统操作。
- 用户可以使用的指令如下：

```
SmartFS >> help
[commands list]
    help        - get the help information
    create      - 'create fileManager fileName' creates a new file with the given name in the fileManager
    move-cursor - 'move-cursor fileManager fileName where offset' move the cursor to where + offset, where can be head/curr/tail
    get-cursor  - 'get-cursor fileManager fileName' get the cursor of the file
    set-size    - 'set-size fileManager fileName newSize' reset the fileSize to new size, fill 0 if newSize is bigger than the old
    get-size    - 'get-size fileManager fileName' get the size of the file
    smart-cat   - 'smart-cat fileManager fileName' reads all content of the file
    smart-write - 'smart-write fileManager fileName pos' input some content and insert them into the file from pos
    smart-hex   - 'smart-hex blockManagerId blockId' read the data in the block and represent them in hexadecimal
    smart-copy  - 'smart-copy fileManagerFrom fileNameFrom fileManagerTo fileNameTo'copy the content from a file to another
    close       - 'close fileManager fileName' flush all the content in the buffer of file
```

图3 程序指令集合

2. 文件系统的持久化

- 为了实现文件系统的持久化，系统将必要的元信息保存在了磁盘上，目录结构如下：

```
- SmartFileSystem
    - FM
        - FM-1
            - file
        - ...
        - FM-20
    - BM
        - BM-1
            - meta
                - b1.meta
            - data
                - b1.data
        - ...
        - BM-20
```

可以在系统中设定FileManager和BlockManager以及每一个BlockManager管理的Block数量，在初始化时，系统将根据设定的数目对目录进行扫描和必要的补充，并将信息映射到内存中对应的数据结构之中。

3. 创建新文件

当用户需要创建新文件时，根据用户指定的file manager name和file name，从`FileManagerSet`中调出对应的file manager并执行`newFile`方法，系统会在对应目录下创建文件，并将size设定为0，写入文件中，之后在对应fileManager用来存储file的map中加入这个新文件。filename就成为用户寻找此文件的key/ID。

4. 写入文件

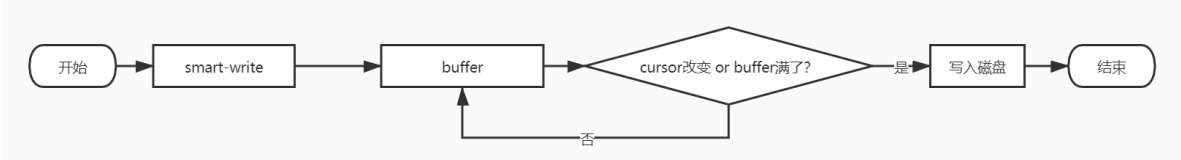

图4 数据写入流程

数据从console到buffer

- 用户可以使用`smart-write fileManager fileName index`指令，在指定的位置插入内容。
- `smart-write`指令会提示用户输入内容，用户输入内容后敲击enter，系统会将用户输入的信息以ASCII编码的方式转为byte数组，并**将其暂时写入对应文件的buffer之中。**
- 用户可以继续使用smart-write在该光标位置输入信息，系统会将其继续写入buffer之中/

数据从buffer到Block

- 三种情况会使得buffer中的内容写入Block：
  - 用户手动使用`close fileManager fileName`将该文件buffer中的数据全数冲出，写入硬盘
  - 在用户下次使用挪动指针，即`move`、`set-size`或者向其他位置进行写入时，系统会先将buffer中的内容写入到当前指针位置。
  - 当buffer写满了，系统也会自行将内容写入buffer之中。
- 数据写入Block过程中，**Block应该满足不可重写**，即一旦内容发现变更，总是创建一个新的Block而不是修改原先的Block。

  为了实现不可重写，每次修改都应该将受影响的数据（包括指针所在块的全部数据和之后的数据）全部取出，然后将插入的数据在内存中拼接到指针处，再另寻块将数据重新写入

图5 红色框线内为受影响的数据

5. 读文件

- 用户可以通过`smart-cat fileManager fileName`从指定fileManager下的文件中，从cursor位置读到文件的尾部，并将数据输入到console中。
- 文件系统实现这一功能，主要依靠计算读取的长度，然后从cursor位置依次读取Block中的数据复制到内存中，然后输出到console中。

6. Set Size

- `set-size fileManger fileName`操作用来给文件重新设定大小，新设定的大小要么比现在的大，要么小于等于现在的大小。
- 如果新设定的大小比现在的大，本质上就是向文件末尾写入一定长度的0x00，因此是一个`samrt-write`的操作，只不过可以跳过buffer直接写入Blcok中。
- 如果新设定的大小比现在的小，也就是一种删除操作，为了保证file和Block的size都正确更新，我选择将受影响的Block全部移除，并新建一个Block存储最后的一些数据。

图6 尾部的数据新建一个Block来保存

7. 复制文件

- `smart-copy fileManagerFrom fileNameFrom fileMangerTo fileNameTo`会将fileNameFrom里的数据复制到fileNameTo中
- 本质上是一次读和一次写。

8. Duplication & Checksum

- 本次lab中的logic block我统一设定为3块，实现细节上，就是每次的写入操作如果需要分配新的block，总是分配3个，并写入一样的内容。
- 在用户读取时，系统会读取block的data用MD5进行加密，然后看密文与meta中保存的是否一致，如果不一致说明文件遭到了人为改动，系统会从duplication中读取数据，如果所有的logic blcok都被损坏了，系统会提示用户，但不会终止进程。

三、异常处理

本系统的异常处理使用ErrorCode实现。

异常类型表

异常类型	ErrorCode码	出现场合
IO_EXCEPTION	0	底层调用Java IO流访问文件时发生IOException
FILE_NOT_FOUND	1	用户输入了一个错误的文件名；底层访问文件时找不到指定文件
FILE_NAME_OCCUPIED	2	供系统使用的文件名被占用了，例如用户在BM下建了一个meta的文件导致系统无法初始化
CREATE_DIRECTORY_OR_FILE_FAILED	4	系统在创建文件或者目录时出错
DATA_FILE_LOST	5	初始化时出现了blcok的meta文件存在但data文件不存在的情况
META_FILE_LOST	6	初始化时出现了blcok的data文件存在但meta文件不存在的情况
INITIAL_FILE_FAILED	7	初始化时读取文件的meta失败导致文件初始化失败
BLOCK_BROKEN	8	文件的所有logic block均出错导致数据受损
NO_MORE_SPACE	9	文件系统全部空间已被使用完全
WRONG_INSTRUCTION	3	用户输入了一条错误的指令
WRONG_FILE_MANAGER_NAME	11	用户输入了一个错误的file manger name
PASSIVE_SIZE	12	用户在输入size的时候输入了一个负值
INVALID_ID	13	创建Block时使用了超出范围的ID
DUPLICATED_ID	14	用户使用了重复的ID创建文件
FILE_ALREADY_USED	15	用户用来复制的目标文件已经有数据了
CURSOR_OUT_OF_RANGE	16	文件的指针超出了文件的范围

- 用户调用方法，如果发生错误，会被捕获并且提示给用户，并且程序不会异常终止。

利用异常保证文件简单一致性

- 在向Block中写入数据，主要包括一下几个步骤

1. 整合受影响的数据和插入的数据。
2. 向Block中写入，并更新内存中对这些新建的block的访问
3. 删除被舍弃不用的Block，包括文件的删除和从内存中去除引用
4. 更新内存中文件的元信息并写入到file的meta文件中

- 这个过程中，一旦2发生错误，3和4便不会执行，因此写在磁盘中的文件信息不会受到影响，于此同时，系统捕获的error后会恢复对原先block的引用，因此整个文件系统回到未执行之前的样子，data和meta都不会被更新。

代码如下：

```
public void writeIntoBlock(byte[] b){
        // 1. 内容整合
        int blockSize = MyBlock.getCapability();
        int pointer = (int) this.cursor % blockSize;
        int blocksIndex = (int)this.cursor / blockSize;

        // 从blockIndex的开头读influenceContentLength大小的数据
        byte[] headContent = readBlock(0,pointer,blocksIndex);
        int tailContentLength = (int)(this.size - this.cursor);
        byte[] tailContent = read(tailContentLength);

        this.cursor -= tailContentLength;//恢复上一步操作导致的cursor改变

        byte[] newContent = new byte[headContent.length + b.length +
tailContentLength];
        System.arraycopy(headContent,0,newContent,0,pointer);
        System.arraycopy(b,0,newContent,pointer,b.length);

 System.arraycopy(tailContent,0,newContent,pointer+b.length,tailContentLength);

        // 2. 开始写入
        // 随机选择一个blockManager，新建一个block（2个duplication），写入数据，不够再建
        int newContentLength = newContent.length;
        int start = 0;
        BlockManagerSet blockManagerSet = BlockManagerSet.getInstance();

        ArrayList<int[]> tmp = this.storeBlocks;
        try {
            // 创建新blocks并填入storeblocks
            while (newContentLength > 0) {
                // 获取data
                byte[] data;
                int min = Math.min(blockSize, newContentLength);
                data = new byte[min];
                System.arraycopy(newContent, start, data, 0, min);

                // 写入block
                int[] blocks = new int[LOGIC_BLOCK_NUM * 2];
                for (int i = 0; i < LOGIC_BLOCK_NUM * 2; i += 2) {
                    MyBlockManager blockManager = (MyBlockManager)
BlockManagerSet.getInstance().getRandomBlockManager();
```

```
                    Block block = blockManager.newBlock(data);
                    blocks[i] = blockManager.getId();
                    blocks[i + 1] = ((BlockId) block.getIndexId()).getId();
                }

                this.storeBlocks.add(blocksIndex, blocks);
                newContentLength -= min;
                start += min;
                blocksIndex++;
            }
        }catch (ErrorCode errorCode){
            this.storeBlocks = tmp; // 如果出错，恢复storeBlocks
            throw  errorCode;
        }

        // 运行到这里说明顺利创建并写入了data文件，接下来才删除data
        // 把原先的data和meta删除腾出空间
        while(blocksIndex < this.storeBlocks.size()){
            int[] oldBlocks = this.storeBlocks.get(blocksIndex);
            for(int i =0; i <oldBlocks.length; i+=2 ){
                MyBlockManager blockManager = (MyBlockManager)
blockManagerSet.getBlockManager(oldBlocks[i]);
                BlockId blockId = new BlockId(oldBlocks[i+1]);
                blockManager.removeBlock(blockId);
            }
            this.storeBlocks.remove(blocksIndex);
            blocksIndex++;
        }

        // 文件元信息放到最后更新，上述过程中出现仍和exception都会上抛，因此不会更改meta，保
证基本的一致性
        this.size += b.length;
        try {
            writeMeta();
        }catch (ErrorCode errorCode){
            this.size -= b.length;
            throw  errorCode;
        }
        this.cursor += b.length;

    }
```

四、补充

本次文件系统我除了实现了一个console程序方便执行以外，还设计了block的回收机制。废弃不用的block，它在内存中的引用以及实际的文件都会被清除，不会影响到文件系统的使用。