CSE设计文档

异常处理

以下是程序中全部异常及其描述，上层类（如File)会捕捉下层异常并产生符合本层描述的错误信息

```
static {
        ErrorCodeMap.put(IO_EXCEPTION, "IO exception");
        //Block类中异常----------------------------------------------------------
---------
        ErrorCodeMap.put(CHECKSUM_CHECK_FAILED, "block checksum check failed");
        ErrorCodeMap.put(BLOCK_DATA_MISSING,"the data of the block is missing");
        ErrorCodeMap.put(INCOMPLETE_READING,"incomplete reading");//没有读入全部
block数据
        //BlockManager中异常-----------------------------------------------------
-----
        ErrorCodeMap.put(BLOCK_NOT_EXIST,"the block doesn't exist");
        ErrorCodeMap.put(DATA_LENGTH_EXCEED_BLOCK_SIZE,"the data is too large
for the block! creation fail...");
        ErrorCodeMap.put(BLOCKMANAGER_CREATION_FAIL,"block manager creation
fail");
        //BlockSystem中异常------------------------------------------------------
------
        ErrorCodeMap.put(DATA_ALLOCATION_FAIL,"data allocation fail");
        //FileMeta中异常---------------------------------------------------------
------
        ErrorCodeMap.put(FILEMETA_CREATION_FAIL,"file meta creation fail");
        ErrorCodeMap.put(FILEMETA_UPDATE_FAIL,"file meta update fail");
        ErrorCodeMap.put(READ_FILEMETA_FAIL,"read file meta fail");
        //File中的异常-----------------------------------------------------------
--------
        ErrorCodeMap.put(OFFSET_OUT_OF_BOUNDARY,"the offset is out of
boundary");
        ErrorCodeMap.put(FILE_IS_BROKEN,"the file is broken");
        ErrorCodeMap.put(GET_FILE_SIZE_FAIL,"get file size fail");
        //FileManager中的异常----------------------------------------------------
---------
        ErrorCodeMap.put(FILEMANAGER_CREATION_FAIL,"file manager creation
fail");
        ErrorCodeMap.put(FILENAME_MAP_CREATION_FAIL,"filename map creation
fail");
        ErrorCodeMap.put(FILENAME_ADD_TO_FILENAME_MAP_FAIL,"filename add to
filename map fail");
        //FileSystem中的异常-----------------------------------------------------
----------
        ErrorCodeMap.put(READ_FILENAME_MAP_FAIL,"read filename map fail");
        ErrorCodeMap.put(DUPLICATE_FILENAME,"the filename has been used");
        ErrorCodeMap.put(FILE_NOT_CREATED,"the file hasn't been created");
        //FileId中异常-----------------------------------------------------------
--------
```

```
        ErrorCodeMap.put(FILEID_GENERATION_FAIL,"file id generation fail");
    }
```

buffer设计

我的buffer设计为链表的结构（因为考虑到buffer数据之间位置变动比较频繁，需要经常进行头部插入和尾部删除，同时还需要将buffer中刚刚被访问过的node放置到链表头部，来实现least-recent-used移除策略的缓存。

我的buffer同时实现了读、写缓存。

缓存node的结构如下图所示：

```
 private static class Buffer{
        int logicBlockNum;
        byte[] data;
        boolean isDirty;
        int addLength;//在isDirty时要读入，在像block中写入时要修改!!!!!!
        Buffer next;

        public Buffer(int logicBlockNum, byte[] data) {
            //..............
        }

    }
```

关于缓存操作有以下几个方法：

1、初始化buffer（主要是先生成10个空node，logicBlockNum设为-1表示没有存放块）

```
private void initBuffer()
```

2、将文件的逻辑块数据存到buffer中（在链表头部插入，将链表末尾的node数据根据isDirty属性写入文件）

```
private void addBlockToBuffer(int logicBlockNum, byte[] data) throws ErrorCode
```

写入文件最大的难点是计算新分配块插入的位置，需要仔细计算[0, logicBlockNum)、[logicBlockNum, logicBlockNum +分配块的块数) 、[logicBlockNum+分配块的块数，....]

将文件大小小心地加上addLength（注意addLength是累加的，因为一个node可能被多次写入）

3、搜索块（如果找到会把该块放到头部）

```
private Buffer search(int logicBlockNum)
```

4、将buffer中的内容强制写入内存（将buffer中标注为isDirty的块都写入）最后必须调用！！！

```
public void flush()
```

与2中操作有一个不同之处在于要把buffer中的这个块恢复成<=BLOCKSIZE的大小（因为这个块已经被认为是未写过的了，只能保存一个逻辑块的数据）

```
//将buffer中的data恢复成<=PublicVars.BLOCKSIZE
if(ptr.data.length > PublicVars.BLOCKSIZE){
    byte[] catData = new byte[PublicVars.BLOCKSIZE];
    System.arraycopy(ptr.data,0,catData,0,PublicVars.BLOCKSIZE);
    ptr.data = catData;
}
ptr.isDirty = false;
ptr.addLength = 0;
```

5、将buffer资源释放

```
public void close()
```

遍历链表，指向node的指针都设为null，将其中的data也设为nuil，释放资源。

关于使用buffer：

在文件read中，每次会先调用search方法搜索想要读取的块，如果没有再到BlockSystem中读取（我自己添加的中间层，下面会进行介绍)。

在文件write中，每次写入时先search，如果buffer中有该logicBlock的缓存，则直接写入buffer，并修改buffer中元信息即可。如果没有，直接写入底层Block，但是同时将该logicBlock缓存。

```
//缓冲写入的整块logicblock（即使它被写得超过size）
int toPutInBufferSize = Math.min(currBlockData.length + b.length,
PublicVars.BLOCKSIZE);
byte[] toPutInBuffer = Arrays.copyOfRange(res,0,toPutInBufferSize);
addBlockToBuffer(currLogicBlock, toPutInBuffer);
fileMeta.setLogicBlock(newLogicBlocks);
```

文件系统基础设计

我的文件系统层次如下：

MyBlock类-->MyBlockManager类-->BlockSystem类-->Myfile类-->MyFileManager类-->FileSystem类

这里写一下加入BlockSystem类和FileSystem类的原因：

BlockSystem

```java
//运用单例模式
public class BlockSystem {
    private static MyBlockManager[] blockManagers;
    private static final int BLOCKMANAGERNUM = 10;
    private static BlockSystem single = null;

    private BlockSystem() {
        //初始化
        blockManagers = new MyBlockManager[BLOCKMANAGERNUM];
        for (int i = 0; i < BLOCKMANAGERNUM; i++) {
            blockManagers[i] = new MyBlockManager(new BlockManagerId(i));
        }
    }

    public static BlockSystem getInstance(){...}

    public String[][] allocateContentToBlocks(byte[] content) throws ErrorCode {...}

    //如果读取失败会返回null，不会抛出错误
    public byte[] readBlock(int blockManagerId, int blockId, int offset) {...}

    public Block getBlock(int blockManagerId, int blockId){...}
```

如上图，我将该类设计为单例模式，因为我只需要它的构造方法调用一次（同时一个文件系统只有一个块系统也很合理）。该类统一管理所有的BlockManager和Block，为上层提供三个方法作为接口（其实真正实用的两个就够了），上层的类只需要依赖于该类就完全可以实现所有功能，降低了耦合性。

FileSystem

```java
public class FileSystem {
    private static MyFileManager[] fileManagers;
    public static HashMap<String,String> filenameToFmAndId;
    private static final int NUMOFFILEMANAGERS = 10;
    private static FileSystem single = null;

    private FileSystem() throws ErrorCode{
        //先初始化十个fileManager
        fileManagers = new MyFileManager[NUMOFFILEMANAGERS];
        for(int i = 0; i < NUMOFFILEMANAGERS; i++){
            fileManagers[i] = new MyFileManager(new FileManagerId(i));
        }
        //将filename与fm和id对应表读入HashMap
        updateFilenameToFmAndId();
    }

    public  static FileSystem getInstance(){...}

    public static void updateFilenameToFmAndId() throws ErrorCode {...}

    public interfaces.File newFile(String filename) throws ErrorCode {...}

    public interfaces.File getFile(String filename) throws ErrorCode {...}
}
```

filenameToFmAndId文件内容如下：

```
file1:1-0
file2:9-0
file3:5-0
```

可以从构造方法以及这个filenameToFmAndId这个HashMap看出来，这个类主要是管理FileManager和File，并且将filename与它在文件系统中的位置记录起来。与BS的理由一样，我也使用了单例模式来设计这个类。

除此之外，还有两个类需要进行说明;

FileMeta

由于file的meta文件包含的内容非常丰富，而且直接修改非常困难，所以我给每个文件对象都维护一个记录了file 的meta信息的数据结构，并且通过它来读取和更新file的meta。

```
public class FileMeta {
    private FileId fileId;
    private  MyFileManager fileManager;
    private long size;
    private int logicBlockNum;
    private String[][] logicBlock;
    private int[] load;

    //此构造方法用于文件已经被创建时使用FileMeta
    public FileMeta(FileId fileId, MyFileManager fileManager) throws ErrorCode {...}

    //此方法用于创建文件时创建meta
    public FileMeta(FileId fileId, MyFileManager fileManager, long size, int logicBlockNum, String[][] logicBlock) throws Erro

    public void updateFileMeta() throws ErrorCode {...}
```

上图展示了主要的方法（略去了setter和getter），构造方法用于读取meta文件内容到文件里。updateFileMeta则将当前FileMeta类实例中的内容按照约定的格式覆盖写到原file的meta文件中，实现file meta的修改。

FileId

该类的构造方法会根据输入的文件名给文件分配id：

```
public FileId(String filename, MyFileManager fileManager) throws ErrorCode{
    //先对filename进行查找，如果有记录则不需要新生成id
    FileSystem.updateFilenameToFmAndId();
    if(FileSystem.filenameToFmAndId.containsKey(filename)){
        String value = FileSystem.filenameToFmAndId.get(filename);
        id = Integer.parseInt(value.substring(value.indexOf("-") + 1));
    }else {
        try {
            id = PublicMethods.chooseSpareId("FM//fm-" +
fileManager.getFileManagerId().getId() +"//meta");
        } catch (IOException e) {
            throw new ErrorCode(ErrorCode.FILEID_GENERATION_FAIL);
        }
    }
    this.filename = filename;
}
```

关于光标

我的光标（currPos）是从1开始的（表示指向文件第一个byte），offset表示移动的字节数。

read操作读取会从包括currPos指向的字节开始读，而write写入会从currPos指向的字节后面开始插入。

如果光标移动到超出文件size的地方，会报错。如果需要这个功能，建议与setSize配合使用。

在这个定义下，currPos = 0时只能写不能读，用户需要注意！！！

关于Block的size

size我认为是最大大小

所以每个block的meta中我还维护了一个变量：load——表示block实际装载的数据大小