CSE Lab1 文件系统lab1

By 黄子豪 俞哲轩 彭超

准备

开始本次lab前，请确保你已经理解第二章的文件系统，理解文件的meta和data block存储方式。

另外如果你了解hadoop的存储原理，做本次lab将会轻松很多，本次lab只要求实现hadoop的partition和duplicate，hadoop官网：http://hadoop.apache.org/

关于lab中不确定的内容，可以与同学或者助教讨论，但是严禁抄袭！动手写代码之前请确保自己理解lab要求，设计正确。

遇到问题可以在课程群中及时询问助教；同时在有自己思考和对问题的整理的基础上向助教提出疑问，避免低效沟通。

情景假设

1. 小明是一个公共文件系统（例如我们学院的ftp）的管理员，他需要采用unix文件系统的方式管理这些文件，即在底层文件的文件名、大小、位置等meta信息和文件的实际内容data是分开存储在不同的block上的
2. 最开始小明把所有的文件放在一台服务器上
3. 但是随着时间流逝用户增多，这台服务器上的文件也越来越多，读写文件的效率也严重下降
4. 于是小明扩展到3台服务器，并且根据某种哈希函数，将所有文件可能均匀地分布到3台服务器中，存储能力和读写速度都提高了很多
5. 但是有一天一台服务器宕机了，上面的所有文件都损坏了，灾难过后小明觉得有必要给每个文件都做个备份，并且备份之间应该在不同的服务器上
6. 假如某个服务器上的一个备份发生错误，应该有能力检测出这种错误，并且能去其他备份上找到正确的文件
7. 但是小明的客户并不关心小明如何实现文件的管理、存储、分区和备份，客户只希望简单的提供文件名就能进行读写，所以小明还应该提供一层抽象，封装底层的实现，向上提供文件操作接口

请帮助小明实现一个这样的文件系统，并且满足以上要求的业务场景。

具体基础要求

1. 类似于unix文件系统，文件的meta和data必须分开存储，通过meta block中的文件位置找到对应的data block
2. 具备两层抽象：File和FileManager，Block和BlockManager，其中File提供文件级别的封装，BlockManager提供Block级别的封装，FileManager和BlockManager只需要提供对应的get和new操作，具体要求实现的接口如下：（语言不限，但建议以面向对象的语言编写，以java为例）FileManager管理File的集合，负责记录File的meta信息
3. Block负责存储数据，Block中的数据分为BlockData和BlockMeta两类信息
    1. 前者以.data为结尾，存储文件数据
    2. 后者以.meta为结尾，存储Block的meta信息
4. Block大小固定，单位是byte，每个Block中可以存储的内容不得大于该size
5. Block应当设置为不可重写的，即在对文件内容进行修改时，应当新建一个Block进行写入，同时修改file的meta信息等

6. BlockManager管理Block的集合，具备创建Block、索引和读取BlockData的能力
7. FileManager和BlockManager支持分组，即不同的Manager只负责自己的FileMeta
8. File需要支持随机读写，允许调整FileData的size；可以直接写入也可以使用buffer，使用buffer写入减少磁盘io次数能获得额外加分
9. File的写入要满足简单的一致性。成功的写入操作需要满足数据写入到block中且fileMeta修改成功。如果写入失败，则不能改变fileData，可以不需要保证FileMeta的不变性，也可以考虑保证。（请在面试的时候和文档中说清楚）
10. 需要完成一个工具系列：
    1. smart-cat：获取File的File内容；能够从文件指定位置，读去指定长度的内容并且打印在控制台。
    2. smart-hex：读取block的data并用16进制的形式打印到控制台。
    3. smart-write：将写入指针移动到指定位置后，开始读取用户数据，并且写入到文件中。
    4. smart-copy：复制File到另一个File。
    5. smart-ls：查看文件系统结构，包括每个FileManager下管理的文件，每个BlockManager下管理的block及其duplication，和每个file使用的block，可以参考下一节“参考”的形式。
11. 需要完成一个异常处理规范，处理异常是使用对应的内容，并且应该整理一个处理规范文档。

参考

BlockManager 和 FileManager 的 Partition 应该可以像下图一样：

```
BlockManager:
BM1 BM2 BM3
b1  b3  b5
b2  b4  b6
FileManager:
FM1 FM2
f1  f3
f2
下面是各个 File 使用的Block
fl:BM1.b1,BM3.b5
f2:BM2.b3,BM1.b2
f3:BM3.b6,BM2.b4
```

一次File读操作的流程：

1. 用户请求读取 FM1.f1 的数据；
2. FileManager通过读取 FM1.f1的meta信息得知它有BM1.b1和 BM3.b5两个block
3. FileManager调用BlockManager1的getBlock，index为1
4. BlockManager1读取BM1.b1的meta信息，得知它的data信息存储的具体物理位置，block读取data信息，返回
5. BM3.b5 同理
6. 如果读取失败（FM不可用，BM不可用，Blk不可用，BIK校验失败等原因）则抛出异常给上层；
7. 如果读取成功，直接返回数据。

一次成功的写操作流程：

1. 用户请求写入数据到FM2.f3的第二个数据块上；

2. 随机选择一个BM,假设选择 BM1;
3. BM1 分配一个新的 Block 编号为 b7;
4. 写入数据到b7;
5. 改变FM2.f3的 FileMeta 为 BM3.b6,BM1.b7(不再引用BM2.b4了）；
6. 不应该从BM2中抹去b4的存在；

一次失败的写入操作：

1. 用户请求写入数据到FM1.f1 的第1个数据块和第2个数据块上；
2. BM3 为其分配b8作为新的第1个数据块，BM2为其分配b9作为新的第2个数据块；
3. 第一个数据块写入成功，第二个数据块写入失败；
4. 维持 FM1.f1 的 FileMeta 不变，然后抛出异常给上层表示写入失败；
5. 不需要删除新分配的b8和b9;
6. 这样就可以保证失败的写操作不会改变File,保证了简单一致性。(注：简单一致性没有定义 FileMeta 写入失败的处理过程，如果FileMeta写入失败，不需要恢复 FileMeta)

支持duplication

现实生活中，磁盘的损坏会使磁盘上的文件丢失，所以我们经常需要使用duplication来保证及时一部分磁盘损坏，也能读取到正确的文件内容。在这个lab中我们可以通过引入logic block来实现。即用户看起来只有一个逻辑上的block，当读取这个logic block时，随机选择一个可用的物理block。

```
size: 200
block size: 32
logic block:
0:["bm-01",13]["bm-02",82]["bm-03",14]
1:["bm-05",31]
...
6:["bm-1",89]["bm-04",21]
```

可以看到，大小为200的文件，占据了六个logic block；且对于每个block而言，存在一定数量的duplicated block；这个“一定数量”既可以是一个固定的数字，也可以是一个在一定范围内的随机数字。

对于每一个logic block，可以首先随机选择一个block，如果block所属的manager存在且数据完整，则可以获取数据，进而一步步获取所有logic block的数据；对于数据信息错误的logic block，应该有相应的异常处理。

注意，duplicated block存在的意义就是防止某一个磁盘损坏，所以不应该在同一个磁盘（对应于BlockManager）下存储相同的duplicated block。

File的写操作

我们这里要实现两种file的写操作，其中一种是write，另一种是setSize。

- write(byte[] b)，将数组b的内容写入到file中
- setSize(int size)，将文件的大小直接进行修改
    - 如果size大于原来的file size，那么新增的字节应该全为0x00
    - 如果size小于原来的file size，注意修改file meta中对应的logic block，且被删除的数据不应该能够再次被访问
    - 主要实现方式合理即可

Block 实现建议

建议Block的data和meta分成两个文件进行存储

由于我们需要检验block是否被损坏，所以我们需要在meta中存储data内容对应的checksum，可以参考的meta存储内容格式为：/path/bm-xx/12.meta

```
size: 512
checksum: 12349192123491912921
```

校验码可以随便选择一种校验/哈希函数（例如MD5函数）

可以参考的存储格式

file.meta

路径:/.../fm-xx/id.meta

内容:

```
{
  file_name: xxx,
  id: xxx,
  block: BM1-b1,BM3-b5,
  ...
}
```

block.meta

路径:/.../bm-xx/id.meta

内容:

```
{
  id: xxx,
  duplication block:BM2-b2,
  physical path:/.../id.data
  checksum: xxx
  ...
}
```

block.data

路径:/.../id.data

内容：实际的用户输入的字节

需要实现的接口

```
public interface Block {
    int getIndex();
    BlockManager getBlockManager();
    byte[] read();
    int getSize();
}
```

```
public interface BlockManager {
    Block getBlock(int index);
    Block newBlock(byte[] b);
    Block newEmptyBlock(int blockSize);
}
```

```
public interface File {
    int MOVE_CURR = 0; //只是光标的三个枚举值，具体数值无实际意义
    int MOVE_HEAD = 1;
    int MOVE_TAIL = 2;
    int getFileId();
    FileManager getFileManager();
    byte[] read(int length);
    void write(byte[] b);
    default int pos() {
        return move(0, MOVE_CURR);
    }
    int move(int offset, int where);//把文件光标移到距离where offset个byte的位置，并返回文件光标所在位置
    int getSize();
    void setSize(int newSize);

    //使用buffer的同学需要实现
    void close();
}
```

```
public interface FileManager {
    File getFile(int fileId);
    File newFile(int fileId);
}
```

一些解释：

- File接口中的`MOVE_CURR`、`MOVE_HEAD`、`MOVE_TAIL`代表的是文件中光标的位置、文件开头和文件结尾，其中光标的位置是File需要维护的一个指针，而用1和2给后两者赋值并无实际意义。
- `close`方法表示的是释放资源，在该lab中如果使用了buffer则需要释放资源，不使用buffer的情况下直接实现为空方法即可。

异常处理

我们的任务是完成一个文件系统，在文件的读取、写入等操作中，很容易引起异常：例如打开不存在的File、创建已经被创建过的文件、Block已经被损坏等。Error Code是一种很简单的异常处理方法，且大家需要整理一份异常处理的文档，解释清楚异常产生的原因即可。如下是一份参考实现。

```
public class ErrorCode extends RuntimeException {
  public static final int IO_EXCEPTION = 1;
  public static final int CHECKSUM_CHECK_FAILED = 2;
  // ... and more
  public static final int UNKNOWN = 1000;

  private static final Map<Integer, String> ErrorCodeMap = new HashMap<>();

  static {
    ErrorCodeMap.put(IO_EXCEPTION, "IO exception");
    ErrorCodeMap.put(CHECKSUM_CHECK_FAILED, "block checksum check failed");
    ErrorCodeMap.put(UNKNOWN, "unknown");
  }

  public static String getErrorText(int errorCode) {
    return ErrorCodeMap.getOrDefault(errorCode, "invalid");
  }

  private int errorCode;

  public ErrorCode(int errorCode) {
    super(String.format("error code '%d' \"%s\"", errorCode, getErrorText(errorCode)))
this.errorCode = errorCode;
  }

  public int getErrorCode() {
    return errorCode;
  }
}
```

工具参考实现

1. smart-cat：直接读取fileData
2. smart-hex：读取block的data并用16进制的形式打印到控制台
3. smart-write：将写入指针移动到指定位置后，开始读取用户数据，并且写入到文件中
4. smart-copy：
    1. 读取已有的file的fileData，写入到新File中
    2. 直接复制File的FileMeta，这个方法的正确性来源于Block是不可重写的，建议使用这个方法实现
5. smart-ls：展示出你的文件系统结构

评分

基础分部，包括上述的基本要求、partition和duplication，满分90分。

bonus额外分数，满分20分，两部分总分不超过100分。

除了基本的要求，其余是否算做bonus视情况而定；在面试之前整理明确好自己的思路。

bonus举例：

- buffer实现

  实际的文件系统一般会先把写操作的脏数据保存在内存中，当文件关闭或者flush时才写回磁盘，这样可以减少磁盘io次数提高效率，你可以参考这一实现方式，把file的字节先缓存在buffer中，close或者flush时再写入到实际的block中

- 文件系统持久化

  你的程序运行时，FileManager和BlockManager，File和Block这些都对象信息都在内存中，假如你的程序结束后你能保存这些信息并且下次成功恢复它，并且保留了原有的文件系统的信息，可以获得一定的加分

- hadoop的其他特性

  我们只要求实现hadoop的最基本的特性之一，如果你能实现其他特性，例如当某个duplication block发生损坏时能自动检测并且恢复，也能获得酌情加分

DDL

10月22日 23:59，提交方式和面试安排另行通知

可参考的测试用例

下面是一个可参考的面试时的测试用例，在面试前请先自己实现这个test类并且确保正确无误（使用其他语言也可，大意一致即可）。面试时将执行类似的测试用例并且通过打断点的方式观察系统状态。如果你提供一个更加user-friendly的命令行界面甚至图形化界面，也可以酌情作为bonus加分。

```
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class Test {
    public static void main(String[] args){
        /*
        * initialize your file system here
        * for example, initialize FileManagers and BlockManagers
        * and offer all the required interfaces
        * */

        // test code
        File file = fileManager0.newFile(1); // id为1的一个file
        file.write("FileSystem".getBytes(StandardCharsets.UTF_8));
        System.out.println(Arrays.toString(file.read(file.getSize())));
        file.move(0,File.MOVE_HEAD);
        file.write("Smart".getBytes(StandardCharsets.UTF_8));
```

```
        System.out.println(Arrays.toString(file.read(file.getSize())));
        file.setSize(100);
        System.out.println(Arrays.toString(file.read(file.getSize())));
        file.setSize(16);
        System.out.println(Arrays.toString(file.read(file.getSize())));
        file.close();
        Tools.smartLs();


        //here we will destroy a block, and you should handler this exception
        File file1 = fileManager0.getFile(1);
        System.out.println(Arrays.toString(file.read(file.getSize())));
        Tools.smartLs();

        File file2 = Tools.smartCopy(1);
        System.out.println(Arrays.toString(file.read(file.getSize())));
        Tools.smartHex(file2.getFileId());
        Tools.smartWrite(0,File.MOVE_HEAD,file2.getFileId());
        file2.close();
        Tools.smartLs();
    }
}

class Tools{
    // implements 4 smart-function
    public static byte[] smartCat(int fileId){};
    public static void smartHex(int blockId){};
    public static void smartWrite(int offset, int where, int fileId){};
    public static File smartCopy(int fileId){};
    public static void smartLs(){};
}
```