CSE Lab1 => α-FileSystem (AlphaFileSystem)

2019年秋CSE, 计算机系统工程 SOFT130058.01, 教师: 冯红伟

助教

1. 黄泠淇 16302010054 (Lab1是这个人出的)
2. 卢振洁 16302010075
3. 袁莉萍 19210240029

描述

计算机系统有一个非常重要的任务叫存储, 更抽象的说法叫数据管理, 存储系统有很多,
比如文件系统和数据库. 大家在本次 Lab 中需要构建一个有多数据副本 (duplication)
和在文件/数据管理层面上分组 (partition) 的文件系统.
虽然用到了分布式系统的两个常用手段 (duplication 和 partition),
但我们仅仅是让这个文件系统具备成为 NFS 的能力 (Lab2才需要大家完成NFS).
同时这个文件系统需要配套的工具, 简单的一致性管理, 以及较为简洁的异常处理规范,
我们给这个文件系统取名为 "α-FileSystem" (或者写作 "AlphaFileSystem").

为了体现 *Worse is better* 的精神, 本次 Lab 必须实现助教要求的 java interface 和给出的特性,
其余没有要求的可以自由发挥, 同学们可以随意选择具体的实现, 也可以随意扩展 interface, 也可以自己添加特性,
如果你不想自由发挥, 可以直接参考下文讨论的实现建议.

> 本次 Lab 修改自18年秋 CSE 课程 (ftp:16ss的文件夹中) 的 SDFS,
> 而 SDFS 受到了 HDFS 的启发.

注: Lab 工作量大, 请尽快构思 AlphaFileSystem 的代码设计, 遇到困难及时询问助教, 当你构思好 Lab 后,
尽量和周围同学或者助教讨论, 以评审设计的正确性, 在与助教交流的时候需要先思考一下, 助教可能不能及时回复,
避免浪费时间.

DDL

DDL: 10月27号23时59分.

将代码打包为 "lab1-名字-学号.zip" 上传至FTP.

在之后会安排面试, 面试主要是询问特性的完成度.

1. AlphaFileSystem 的特性

1. AlphaFileSystem 会尽量利用操作系统自带的文件系统;
2. AlphaFileSystem 分为 "FileManager" 和 "BlockManager" 两类服务;
3. AlphaFileSystem 用 "File" 来管理数据, 用 "PhysicalBlock" (或者 "Block") 来存储数据;
4. File 管理的数据是一个连续且有限的字节序列 (就像常见的文件系统那样), 最小存储单位为 8bit-byte;
5. File 分为 FileData 和 FileMeta 两部分:
    1. FileData 是字节序列的抽象, 用 size 表示字节数, 并要求以某种方式拆分 FileData,
       并将拆分好的数据分散存放在一些 Block 中, 同时 FileData 必须实现 **Duplication**;
    2. FileMeta 表现为 "file-23.meta" 这样的文件,
       至少需要记录 FileData 的 size 以及找到分散的 Block 并重新组成完整的 FileData 的方法;

    3. 之后会讨论 FileData 拆分, 复制 (duplication) 以及 FileMeta 如何描述这些信息的实现方法;
6. FileManager 管理一个 File 的集合, FileManager 只负责记录 FileMeta,
   因此 FileManager 具备 File 创建和检索能力但不具备数据存储的能力;
7. Block (PhysicalBlock) 负责存储数据, 也分为 BlockData 和 BlockMeta 两部分:
    1. BlockData 表现为 "blk-82.data" 这样的文件, 这个文件存放 Block 存储的数据;
    2. BlockMeta 表现为 "blk-82.meta" 这样的文件, 至少需要记录 Block 的大小 size 和校验值 checksum;
8. Block 的 size 一般是2的幂, 单位是 byte, 比如 512B, 4KB, 1MB, 在程序构建早期,
   建议选择 512B 或者更小的 256B (256=16*16),
   这样可以方便在调试的时候打印查看, 在调试好后可以选用更大的 4KB 和 1MB;
9. Block 必须是 **Immutable** 的, 即 Block 一经创建并保存数据后就不能再改变;
10. BlockManager 管理一个 Block 的集合, 具备 Block 创建, 索引和读取 BlockData 的能力;
11. FileManager 和 BlockManager 要支持分组 (**Partition**), 即不同的 Manager 只能管理自己负责的那部分,
    比如一个 FileManager 管理的 FileMeta 不可能从其他 FileManager 读取到;
12. 根据上面的信息, 获取 FileData 的方法可以简单理解为: 从 FileMeta 中获得所有 Block 的定位信息 (Block 归属的 BlockManager 和在该 BlockManager 中的索引), 然后读取对应的 BlockData, 然后将这些数据组合起来复原得到 FileData;
13. File 需要支持随机读写的, 允许任意调整 FileData 的 size,
    在读写的时候使用 Buffer (如果觉得太难可以考虑作为Bonus),
    后面给出的 java interface 会再次讨论这个问题;
14. File 的一次写入操作需要满足简单的一致性:
    一次成功的写入操作需要满足数据写入到 Block 中, 同时 FileMeta 修改成功,
    如果写入失败, 则不能改变 FileData, (不需要保证 FileMeta 不变);
15. 需要一些配套的工具:
    alpha-cat 获取 File 的 FileData 并打印到控制台中;
    alpha-hex 读取 BlockData 并用16进制的形式打印到控制台中;
    alpha-write 指定写入位置, 并从控制台读取用户输入, 然后将数据从 FileData 的指定位置开始写入;
    alpha-copy 将一个 File 复制到新的 File 中;
16. 需要一个异常处理规范, 使用 Error Code 来编号异常, 并且整理出一个异常处理规范.

1.1. 参考

BlockManage 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
    f1: 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. File 先读取 BM1.b1 的数据, 然后读取 BM3.b5 的数据.
3. 如果读取失败(FM不可用, BM不可用, Blk不可用, Blk校验失败等原因)则抛出异常给上层;
4. 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.
7. (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)

2. 索引 File 和 Block

本次Lab需要实现 Partition, 索引一个 File/Block 必须知道所属的 FileManager/BlockManager,
下面将会讨论一个建议的索引方法.

2.1. File 的索引

一个好消息: 在18CSE课程中的 SDFS 使用 Path (比如 "/foo/bar/1.txt") 来索引 File,
由于本次 Lab 要求支持 partition, Path索引工作量会很大, 我们就取消这个东西,
如果你能力极强也是可以考虑做一下的.

如果不用 Path 来索引 File 那么 AlphaFileSystem 没有必要有目录结构,
因此 AlphaFileSystem 也不需要文件夹 (Directory) 这个概念,
AlphaFileSystem 的 File 类似于 Unix/Linux 的 Regular File.

虽然不使用Path, 但还是需要一个方法去索引 File, 那就是用 fileId 索引, fileId是一个字符串,
如果我们知道了 File 所属的 FileManager 那么给出 fileId 就可以索引到这个 File.

那么如何获得 File 所属的 FileManager 呢?

我建议使用字符串作为 Manager 的 Id. 例如, FileManager 的 Id 为 "fm-17",
同时管理了一些File, fileId 分别为 "nihao", "hello", "cse", 那么在你的电脑中应该表现为:

```
/path/to/fm-17/nihao.meta
/path/to/fm-17/hello.meta
/path/to/fm-17/cse.meta
```

这样你只需要知道 `("fm-17", "nihao")` 这个二元组就可以索引到 `/path/to/fm-17/nihao.meta` 了.

注: 特性1告诉我们应该充分使用自带的文件系统, 因此我们把 `/path/to/fm-17/` 作为 "fm-17" 的工作目录,
工作目录需要自己指定, 如果你有更合适的方案也是OK的.

2.2. Block 的索引

Block 使用一个64位整数 indexId 来索引, 在知道所属的 BlockManager 后, 给出 indexId 就可以找到这个
Block; BlockManager 与 FileManager Id 一样使用字符串作为 Id.

Block 使用64位整数索引是因为 BlockManager 需要支持自动分配 Block indexId 的功能,
简单起见, indexId 建议使用自增的方法 (魔鬼的低语: 在工程上更喜欢使用UUID之类的东西),

下面是一个例子:

BlockManager Id: "bm-07", 所属Block的indexId有"1", "2", "3", 于是:

```
/path/to/bm-07/1.data
/path/to/bm-07/1.meta

/path/to/bm-07/2.data
/path/to/bm-07/2.meta

/path/to/bm-07/3.data
/path/to/bm-07/3.meta

/path/to/bm-07/id.count  // id.count 里面是 "4", 表示下一个可用的 Id
```

每次分配 indexId 的时候, 需要从 id.count 中读取, 然后自增1后再写入 id.count, 同时将读取到的 Id,
作为新 Block 的 indexId.

3. FileMeta 和 Duplication

下面仅仅只是一个 FileMeta 和 **Duplication** 实现的一个建议, 如果你对 duplication 有独到的理解,
你可以独立思考一个 duplication 的设计, 实现之后可以在课堂和大家分享一下, 如果你第一次接触 duplication,
助教建议先参考下面的设计.

3.1. 为 FileMeta 引入 LogicBlock 以支持 Duplication

引入 LogicBlock 后的 FileMeta, 需要记录 size 表示 FileData 的字节数, 还需要记录 block size 表示
LogicBlock的字节数, 最后是 LogicBlock List, 下面是一个例子:

```
size: 65300
block size: 512
logic block:
0: ["bm-01",13] ["bm-02",82] ["bm-05",12]
1: ["bm-04",322]
 .
 .
16:    // 文件空洞
 .
 .
127: ["bm-04",322] ["bm-05",12]
```

这个例子中, FileData 的 size 为 65300, block size 指定为统一的 512 Bytes
(一般一个文件的每个 block 的 block size 都是相同的),
有 ceil(65300/512)=128 个 LogicBlock,
最后一个 LogicBlock 没有使用所有的 512B 空间, 但是还是需要占用一个 512B 的 LogicBlock;

每个 LogicBlock 由序列号和一串 Block(PhysicalBlock) 列表 (Block List) 组成, 比如:

```
0: ["bm-01",13] ["bm-02",82] ["bm-05",12]
```

表示第0个 LogicBlock, 有3个副本, 正常情况下这3个 Block 保存的数据应当是一模一样的, Duplication 就是指
将同样的数据多次复制, 数据副本的数量应当可以配置/或者取决于某种分配算法,
不要求每个 LogicBlock 都有相同数量的 Block.

如何索引 LogicBlock 0 的数据呢? 随机选择一个 Block 如果 Block 所属的 BlockManager 存在,
且能从该BlockManager取得该 Block 的数据, 且这个Block校验成功,
则 LogicBlock 的数据与这个 Block 的数据相同, 如果上面任何一步失败, 则随机选择另外的 Block, 直到成功,
如果找完所有的 Block 却没有成功, 则认为这个 LogicBlock 是 unavailable, 需要有相应的异常处理机制.

另外注意到 **文件空洞** 这个注释, 如果 LogicBlock 没有关联的 Block 那么认为这个 LogicBlock 的数据的每个字节都是 0x00: `0x00 0x00 ... 0x00`.

3.2. 引入 LogicBlock 后 File 写操作实现的问题

File 的写操作有 write 和 setSize (类似于 POSIX 接口的 truncate).

LogicBlock 这种设计使得每种写操作都会更改 FileMeta:

执行 write 操作时, 由于要求 Block 是 **Immutatable** 的, 我们不能直接更改旧 Block 中的数据 (BlockData),
所以每次改写一个 LogicBlock 需要将数据写入到新 Block 中,
由于要求 Duplication, 你需要根据配置 (或者分配算法) 创建多个数据相同的 Block,
最后将旧 Block 从该 LogicBlock 的 Block List 中删掉, 并将新 Block 添加到对应的 Block List 中.

另一种 setSize 操作需要改变 FileMeta 中的 size,
如果长度变短到需要删掉多余的 LogicBlock, 则需要从 FileMeta 删掉这写多余的 LogicBlock,
如果长度变长则需要增加 LogicBlock 以及分配 Block,
如果长度变长且某个 LogicBlock 的数据全为 0x00, 那么可以像 **文件空洞** 那样不去分配 Block.

4. Block 实现建议

建议 BlockData 和 BlockMeta 用两个文件分别存储, 一个例子是 size 为 512 bytes 的 Block:

/path/to/bm-xxx/12.data: // 这里使用16进制展示, 实际的文件应该是二进制的

```
0x70 0xc2 0x1f ... 0x23
```

/path/to/bm-xxx/12.meta:

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

校验: 随便选择一种校验/哈希函数(在以前MD5是一种不错的校验函数, 完全可以在这个Lab中使用),
也可以自己写一个, 下面给一个参考:

```
// 这段代码可能不能通过编译
long checksum(byte[] bs) {
    long h = b.length;
    for (byte b: bs) {
        long lb = (long) b;
        h = ((h << 1) | (h >>> 63) |
            ((lb & 0xc3) << 41) | ((lb & 0xa7) << 12)) +
            lb * 91871341 + 1821349192381;
    }
    return h;
}
```

5. 需要实现的接口

```
interface Id {
    // empty interface
}

// Block 没有write操作, immutatable
interface Block {
    Id getIndexId();
    BlockManager getBlockManager();

```

```
        byte[] read();
        int blockSize();
    }

    interface BlockManager {
        Block getBlock(Id indexId);
        Block newBlock(byte[] b);
        default Block newEmptyBlock(int blockSize) {
            return newBlock(new byte[blockSize]);
        }
    }

    public interface File {
        int MOVE_CURR = 0;
        int MOVE_HEAD = 1;
        int MOVE_TAIL = 2;

        Id getFileId();
        FileManager getFileManager();

        byte[] read(int length);
        void write(byte[] b);
        default long pos() {
            return move(0, MOVE_CURR);
        }
        long move(long offset, int where);

        void close();

        long size();
        void setSize(long newSize);
    }

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

稍微解释一下 `File` 的接口的语义:

`File` 应该维护一个指针, 指向 FileData 的某个位置, 每次 `read`/`write` 操作从这个指针指向的位置开始, 一般, 每次成功的 `read`/`write` 操作都会向后移动 `length`/`b.length` 个字节.

`File` 应该是可以随机访问的, 即可以使用 `move` 重新指定指针的位置, 在本次 Lab 中, 指针应该是64位整数, 指针不应该是负值, 指针应该从0开始, 指针的值可以超过 FileData 的 size.

`move` 操作有两个参数: offset 和 where, 语义类似于 POSIX `lseek`.

`close` 表示释放资源, 在本次 Lab 中这个资源应该是指 Buffer, 如果不打算完成 Buffer 可以将这个方法实现为一个空方法.

`size` 表示 FileData 的字节数量, 是64位整数; `setSize` 是重新指定 FileData 的字节数量, 如果 size 变大, 新增的字节应该全为 0x00, 如果 size 变小, 被删除的数据不能再访问到.

6. 异常处理

CSE 计算机系统工程, 顾名思义我们的Lab是一个工程, 所以需要我们去处理异常, 可能的异常: 无法索引到 Block; 打开 File 时, File 不存在; 创建 File 时, File 却在早先时候创建了; 读取 File Meta 时, InputStream 抛出了 IOException; 读取 Block 时校验失败 ...

由于最后会使用 RPC 的方式, Error Code可能是一种简单的异常处理方式, 同时为了体现工程, 这个Lab需要大家整理一份异常处理的文档, 用 Error Code 编码每一个你想到的异常, 同时描述这个异常产生的原因.

由于 java 有良好的 Exception 机制, 下面有一个参考实现, 由于上面的 interface 没有标注 Exception, 所以你需要使用 RuntimeException.

```java

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;
    }

}
```

7. 工具实现参考

1. alpha-cat: 直接读取 FileData;
2. alpha-hex 读取 BlockData 并用16进制的形式打印到控制台中,
   每行输出16个16进制数表示的字节, 如果 Block 的 size 为 256B,
   那么输出应该有16行;
3. alpha-write 将写入指针移动到指定位置后, 开始读取用户数据, 并写入数据到文件中;
4. alpha-copy 有两种实现方式:
    1. 读取已有 File 的 FileData, 然后写入到新 File 中;
    2. 直接复制已有 File 的 FileMeta, 这个方法的正确性依赖于 Block 是 *Immutable* 的,
       建议使用这个方法来实现 copy.