CSE Lab2 => α-FileSystem-v2 (AlphaFileSystemV2)

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
2. 卢振洁 16302010075
3. 袁莉萍 19210240029

Lab2描述

在Lab1中, 同学们实现了 AlphaFileSystem, 以此为基础, 通过引入某种形式的网络通信协议, AlphaFileSystem 可以具备NFS的能力, 为做区分, 我们将其称为 AlphaFileSystemV2.
在具备网络通信能力的同时, NFS必须要考虑更多的限制, 文档在稍后的时候讨论这些限制.

与Lab1一样, 同学们只需要完成要求的特性, 其余不作要求, 自由发挥.

DDL

DDL: 12月1号23时59分.

将代码打包为 "lab2-名字-学号.zip" 上传至FTP.

在之后会安排面试, 和Lab1一样, 面试主要是询问特性的完成度.

1. AlphaFileSystemV2 的特性

1. Lab2包括Lab1的所有特性;
2. 选择某种网络通信协议:
    1. 必须是RPC的形式;
    2. 可以自己实现一个RPC协议, 需要整理一个详细的协议文档;
    3. 选择一个开源的RPC协议, 这些协议一般都有现成的框架, 建议使用Java RMI, 比较省事;
3. 每个 Manager (FileManager 和 BlockManager) 都应该有 Server 端和 Client 端两种形式
    1. 我们用 "Manager(Server)" 表示 Server 端的 Manager;
    2. 首先各个 Manager(Server) 需要具备**异步工作**的能力;
    3. **异步工作**的意思是指各个 Manager(Server) 不需要按照某个时间约定上线和下线;
       比如, 你的 AlphaFileSystemV2 有2个FileManager和3个BlockManager,
       你可以先启动其中1个FileManager和2个BlockManager, 并开始对外提供服务,
       尽管这个时候 AlphaFileSystemV2 没有完全启动, 但是仍能提供不完整的文件管理服务,
       在稍后启动剩下的Manager, AlphaFileSystemV2 就能对外提供完整的服务了;

    4. Manager(Server) 应该具备一定的容错能力, 如果 Client 端发送了一个会引起异常的请求
       (比如向BlockManager请求一个不存在的Block), 你的设计不应该让 Manager(Server)
       异常终止, 相反 Server 应该将这个异常情况告诉 Client 端, 并能继续对外提供服务;
       (需要注意的是, 这一点无法完全做到, 这里只是要求大家尽量多的处理这些异常请求)
    5. 由于第3点无法完全做到, 你的 Server 在异常终止时需要尝试重启,
       如果重启失败, 这个 Server 只能停止对外服务;
    6. Client 需要将 Manager(Server) 返回的异常重新抛出, 如果这些异常是可恢复的,
       外层代码应该尽量恢复这些异常, 而不是让 Client 所在的程序异常终止;
    7. Client 需要设置 RPC 超时时间; 比如, 我们设置 RPC 超时时间为 500ms, 假设正常情况下,
       从 Client 发送请求到 Client 收到 Manager(Server) 发送的回执, 需要花费20ms;
       由于网络异常上述过程可能需要花费 2s, 或者由于服务端异常上述过程永远无法完成,
       Client 需要在等待时间超过 500ms 时切断连接, 并抛出一个异常;
       需要注意的是, 仅仅设置 TCP/IP 的超时时间是不能满足要求的,
       同学们应该自己实现一个计时器(开源RPC框架可能已经有可用的超时配置项)来满足这个特性;
       (自测建议: Manager(Server) 可以调用 sleep(2000) 造成超时)
    8. 由于 Lab2 中的 Manager 可能会异常终止, (而 Lab1 中的 Manager 可能总是在线),
       你需要仔细考虑异常处理的情况; 比如在实现 **Duplication** 时, 你可能有一个选择
       BlockManager 的算法, 如果选择的某些 BlockManager 下线了,
       那么你可以进行新一轮的选择算法以替换这些下线的 BlockManager,
       或者是抛出一个异常, 表示 Block 分配失败;
4. 有独立于网络通信协议的名字系统, 稍后会给出一些实现建议;
5. RPC通信协议必须接入到Lab1中要求的java interface:
    1. 如果你是自己实现的RPC协议, 务必按照Lab1要求的java interface来实现;
    2. 如果你选择已有的RPC协议, 那么这些协议的接口代码形式很可能和
       Lab1 中要求的java interface不同, 你需要编写一些转接器(adapter)代码,
       将这些接口翻译成Lab1中要求的java interface;
6. 更完善的buffer策略 (Bonus)
    1. BlockManager 的 Server 端需要缓存 Block 以减少磁盘IO (Bonus #1);
    2. BlockManager 的 Client 端需要缓存 Block 以减少网络IO (Bonus #2);
    3. FileManager 的 Server 端需要缓存 FileMeta 以减少磁盘IO (Bonus #3),
       注意 FileMeta 是 **Mutable** 的, 你应该实现 **write through**,
       即对 FileMeta 的更改需要同时更改内存中的 FileMeta 和磁盘中的 FileMeta;
    4. 缓存换入换出的策略可以自由选择;
7. 更完善的异常处理规范:
    1. 在Lab1的基础上扩充;
    2. AlphaFileSystemV2 引入了网络编程, 网络本身的异常处理需要整理到规范中;
    3. **异步工作** 中提到的容错机制需要对应的异常处理规范, 也需要整理进文档;
8. Lab2 对一致性有更高的要求:
    1. 对某个`File`的一次成功写入操作(setSize, write等),
       你需要将改动的 FileData 保存在新分配的 Block 中,
       同时将 FileMeta 的变更保存在磁盘中;
    2. 如果这次写入操作失败了, 你应该保证 FileMeta 和执行写入操作前的 FileMeta 一致;
    3. 如果你实现了 "Bonus #3", 如果你先改写内存中的buffer, 然后执行存盘操作,
       如果存盘操作失败, 你应该回滚对buffer的改动;

4. 提示: java 的 `FileOutputStream` 不能保证写入的数据一定能记录在磁盘中,
   java 提供了 `RandomAccessFile` 这个API,
   使用 `rws` 或 `rwd` 模式读写文件可以实现上面的一致性;

1.1. 对 Lab1 的补充:

1. 部分同学们反映Lab1文档没有要求校验 Block 也没有要求其他的异常处理(只在实现建议中提到),
   在 Lab2 中要求在读写 FileData 时需要利用 Block 的备份特性,
   即对于存在多个副本 (duplication) 的 Block, 如果读取的 Block 无法通过校验 (BlockData的checksum与BlockMeta的checksum不一致), 或者根本没有办法读取到这个 Block,
   则需要尝试读取其他 Block, 直到找到一个通过校验的 Block, 如果找不到这样的Block,
   需要抛出异常;
2. Lab1 中要求的 java interface 有一个接口的名字为 `File`, 可能会对编码造成麻烦,
   建议大家将其改名为 `AlphaFile` 或者其他名字;

2. RPC 协议

这里只讨论自己实现的RPC协议和使用Java RMI两种情况:

2.1. 自己实现RPC协议

如果你不愿意去了解现有的RPC协议, 你可以通过socket编程,
自己定义方法调用的编码规则. 假设每个 Manager 独占一个 `IP:port`,
那么还需要维护一个 `ID -> IP:port` 的映射表作为名字管理系统.

下面给出一种参考:

1. 为java interface中的每个有必要编码的方法分配一个32位整数作为 ID,
   我们称这个 ID 为 Method Id;
   比如为BlockManager 的 `getBlock` 分配 `1`, `newBlock` 分配 `2`,
   没有必要为 `newEmptyBlock` 分配 Method Id;
2. 为每个 Manager 建立一个名字映射表;
   比如 ID 为 "m1" 的 Manager, 映射到 `127.0.0.1:12345`;
3. 现在尝试调用Client端 "m1" 的 `getBlock` 方法;
    1. 从映射表中找到 "m1" 对应的 `IP:port`;
    2. 使用socket连接 `127.0.0.1:12345`
    3. 连接成功后, 确定 `getBlock` 的 Method Id, 同时序列化这个方法的参数;
    4. 依次发送以下数据作为request:
        1. 一个整数表示Type (`1`表示BlockManager, `2`表示FileManager);
        2. 被调用方法的Method Id (`getBlock` 对应 `1`);
        3. 发送序列化的函数参数;
    5. Server端的BlockManager收到request后反序列化, 得到RPC调用的方法和参数,
       然后调用本地的 `getBlock`方法, 得到结果;
    6. 然后Server端的根据结果依次发送以下数据作为response:

        1. 首先发送 ErrorCode, 如果没有异常, 发送0, 如果有异常, 这发送该异常的 ErrorCode
        2. 如果没有异常, 那么你需要返回一个 `Block` 对象, 稍后会讨论返回的是什么Block对象;
    7. Client端收到response后, 反序列化, 如果 ErrorCode 不为 0,
       则抛出一个 `ErrorCode` 异常对象,
       如果 ErrorCode 为0, 反序列化 `Block` 对象并返回.
4. 通常需要实现 `BlockManagerClient` 和 `BlockManagerServer` 这个两个类,
   这两个类都实现(`implements`)了 `BlockManager` 接口, 用来负责上面的网络协议编码和解码;
5. 自己实现RPC协议将会依赖于socket编程, 需要将socket编程的异常处理整理到异常处理规范中.

2.2. 选择Java RMI

同学们可以去搜索一下 Java RMI 的基本用法, 这里整理了一些相对于自己实现的RPC协议的优点:

1. Java RMI 的传输协议已经定义好, 不需要自己定义上面讨论的 Type, Method Id 等;
2. 网络传输, 方法调用的编码的代码可以自动生成, 不需要手写;
3. 使用了反射, Server(RMI Skeleton) 和 Client(RMI Stub) 的代码可以在运行时生成, 不需要在运行前生成;
4. 定义了一个完整的发布机制:
    1. RMI 有一个 Registry, 可以把一个 object 注册到上面, 注册时还需要额外提供名字;
    2. RMI Registry 将注册的 object 以URL的形式发布出来:
       比如, 有2个BlockManager, 可以分别发布为:
       `rmi://localhost/bm1` 和 `rmi://localhost/bm2`;
    3. 在这个Lab里面需要维护一个映射表来隐藏这个发布机制;
5. Java RMI 的接口定义语言(IDL)就是Java的interface;

下面是一些需要注意的地方:

1. RMI接口形式和Lab1中要求的接口不同:
    1. RMI接口需要继承 `Remote`, 且每个方法都要抛出 `RemoteException` 这个异常;
    2. Lab1中要求的接口比较干净, 每个方法都没有抛出 Checked Exception, 仅仅会抛出 `ErrorCode`
       这样的 Unchecked Exception, 而且也没有继承额外的接口;
    3. 按照本次lab的要求, 你需要一些转接代码将RMI接口翻译成 Lab1 中要求的 java interface;
2. RMI 需要接口参数和返回值进行序列化, 不要使用无法序列化的参数;
3. 建议大家在AlphaFileSystemV2中使用RMI前, 先做一个简单RMI Demo, 看看如何进行参数序列化和异常处理.
4. RMI 也会引入新的异常处理, 需要整理进异常处理规范中.

3. Block 和 File 的传输

在 Lab1 中, 有些同学在实现 Block 的时候, 采用了一种 "Lazy Load" 的策略,
即 `getBlock` 返回的 `Block` 对象没有包含数据, 在调用`read`方法后才读取 BlockData.

在本次 Lab 中, 这样的方法会徒增烦恼, 建议在 `getBlock` 时同时读取 BlockMeta 和 BlockData, 然后返回给 Client 端.

对于 File, 由于 FileManager 只保存 FileMeta, 你可以将整个 FileMeta 返回给 Client 端, 让 Client 维护文件指针, 如果 File 成功执行了写入操作, 需要将新的 FileMeta 发送给 Server 端以保存变更.
这里不需要考虑多次打开同一个文件的情况,
可以假设每时每刻同一个文件只会被打开一次.