16 级计科 7 班: 分布式期末项目 #SDFS 分布式文件系统

Due on Wednesday, January 9, 2018

陈鹏飞 周一 *9-10* 节

颜彬

16337269

Content

1　项目简介

这次项目实现了名为 SDFS(Simple Distributed File System）的分布式文件系统工具。

其代码包括 server 端和 client 端。在 client 端，用户可以执行文件系统支持的常用操作，例如上传文件（到 SDFS)、下载文件（回原来的 fs)、ls、cat 等。server 端负责响应这些请求并做相关的操作。(图 1为其命令行界面，展示了支持的一些操作)。

本项目使用 Python 3.7 实现，仅仅利用了 Python 的标准库和 rpyc 库，其中后者用于发起 rpc 请求。整个分布式文件系统以 rpc 的方式交互和传递信息。

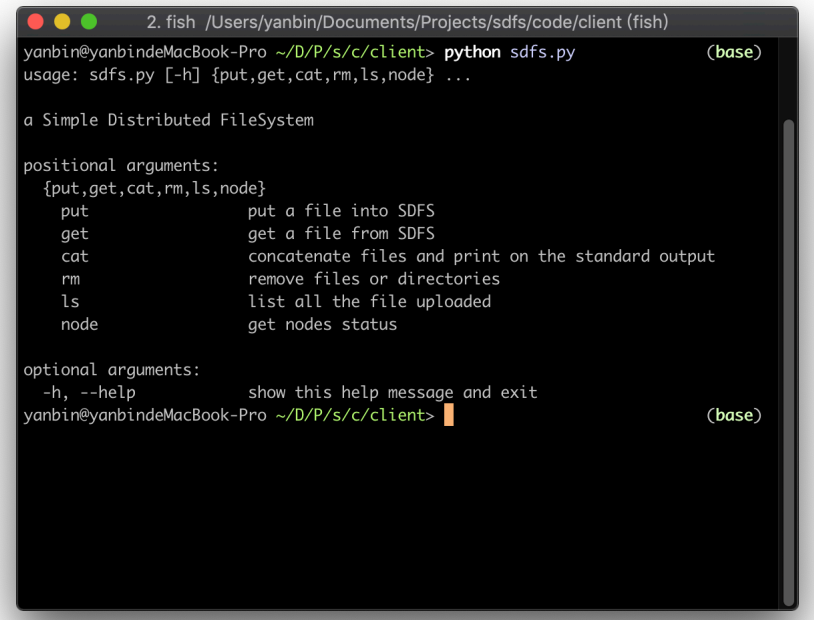

图 1: SDFS 命令行界面

2　使用方式

2.1　开发环境介绍

使用 Python 3.7 开发，使用了 rpyc 第三方库。如果系统有预装 conda，可以按代码 1的方式配置。测试环境下，会同时开启多个终端并运行多个进程，需要使用 tmux 工具。

其他安装方式与之类似。

代码 1: 环境安装方式

```
sudo apt install tmux
conda create --name sdfs python=3.7
conda install rpyc
conda activate sdfs
```

2.2 服务端运行方式

服务端需要运行起注册服务器、NameNode 和 DataNode（见节 3的介绍）。如代码 2所示。

其中 run.sh 程序需要安装 tmux 工具。

代码 2: 测试环境下服务端的运行方式

```
cd code/server
python rpyc_registry.py # 运行注册服务
bash run.sh # 运行NameNode和DataNode
```

2.3 客户端运行方式

客户端的运行方式如代码 3所示。如果环境正确，预计会看到图 1所示的欢迎画面。该画面展示了命令行工具支持的指令，以及他们的简要用法。

代码 3: 客户端运行方式

```
cd code/client
python sdfs.py
```

3 设计架构

由于本项目尽可能地按照 HDFS 架构来设计，故先介绍 HDFS 架构，再比较和本项目的不同之处。

3.1 HDFS 设计架构

本项目的设计类似于 HDFS 架构。图 2介绍了 HDFS 的具体架构。

在 HDFS 是一个 master/slave 架构的文件系统。一个 HDFS 集群包括一个 NameNode 和若干个 DataNode。其中 NameNode 是主服务器，负责管理文件的元数据，例如文件系统的命名空间、文件的访问权限等。其他的若干个 DataNode 负责存储和管理文件的真实数据。

在 HDFS 内部，一个文件会划分为多个块 (block)。这些不同的块会产生若干个（一般是 3 个）副本（replica）。这些副本被分配到不同的 DataNode 中存储。当用户需要检索某个文件时，需要向不同的

DataNode 拿到文件的块，再将块拼接起来得到原始文件。

在这个过程中，NameNode 会负责处理打开文件、关闭文件、文件重命名以及处理文件的路径。除此之外，NameNode 还负责将副本 (replica of blocks) 映射到某个 DataNode 中。

DataNode 的任务简化为创建块、删除块、以及根据 NameNode 的指示复制块等。

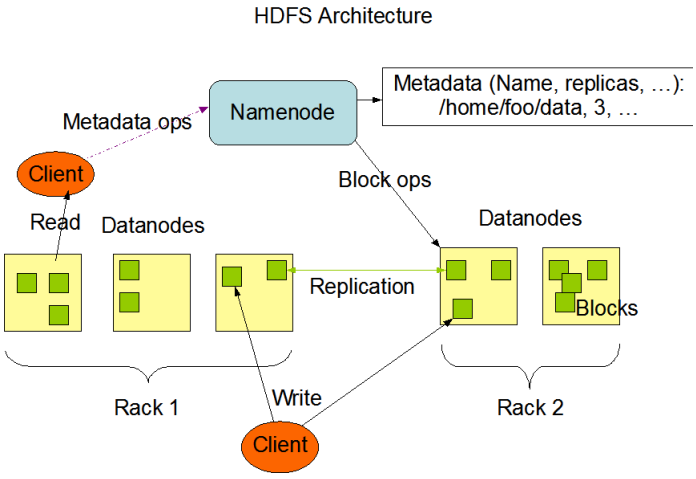

图 2: HDFS 架构示意图

3.2　SDFS 设计架构

本项目（SDFS）的架构与 HDFS 十分类似，但 HDFS 还是过于复杂，作为一次课程项目，做了如下的简化。

- 不存储文件路径。即 SDFS 是 “扁平化” 的文件系统，所有文件存储在根目录上。
- 不进行用户权限管理。即默认所有用户是 root 用户，具有完全的权限。

但 SDFS 支持并实现了 HDFS 的许多功能，包括文件分块、为块创建副本、容错、块状态维护和错误块修复等。这些特性保证了 SDFS 具有比较好的容错能力。

在默认 3 副本的情况下，任何块可以拥有 1-容错的能力。即便一个文件的每个块都有一个副本被损坏了，剩下的两个副本可以通过投票胜出，成为正确 (healthy) 的块，从而被用户访问到。

假设不考虑块损坏，由于块拥有 3 个副本，（最坏下）任何两台服务器的停止失效都不会影响到块的正常访问。

SDFS 还提供了足够的命令行指令，允许用户指定块大小 (--block, -b) 和指定副本个数 (--replica) 等。

3.3　SDFS 架构细节

下面进一步介绍 SDFS 架构的细节。

用户（client) 通过命令行工具与 HDFS 后端交互。后端包括一个注册器 (register), 一个 NameNode 主服务器和若干个 DataNode 数据存储器。

3.3.1 启动 SDFS

注册器必须是最开始运行的进程。在注册器启动完毕后，其他后端服务器才可以创建和运行。这些服务进程创建时，会自动通过广播的方式向注册器进行注册。注册器维护当前活跃的服务进程。

由于注册器的存在，NameNode、DataNode 和 Client 三者的交互可以通过注册器完成。任何一方要访问另一方时，只需要通过 rpc 向注册器询问一个服务，注册器会返回该服务的一系列地址（ip 和端口)。在这里，NameNode、DataNode、Client 都是服务，它们的 ‘‘服务标志符’’ 就是自己的名字。

如果 NameNode、DataNode 或 Client 中有部分进程不在一个子网中，注册器会失效。此时 SDFS 提供了配置文件的机制，可以通过配置文件指定要访问的服务的 ip 和 port。

3.4 发起请求

当 Client 实际上由两部分组成。分别是

- Parser，命令行解析工具，将用户的输入提取为一个 object。
- Connector，连接工具，负责使用与 NameNode 或 DataNode 作连接和数据传输。

当 Client 发起请求时，Connector 拿到解析好的用户输入指令后，要考虑与哪些服务建立连接。

其中 Client 的每个操作都一定要与 NameNode 建立连接。这是因为所有的 ‘‘文件分块信息’’ 都被存储到 NameNode 上，对文件的任何操作在访问到文件的块之前，都必须向 NameNode 询问块的所在位置。

Client 的每个操作都会涉及到一定数量的 DataNode。这取决于文件块所在的位置。

Note: Client 直接将文件块传送给 *DataNode*，而不是先把文件传给 *NameNode* 再由其进行分发。这样做的目的是尽可能地减少带宽（与 *HDFS* 的实现相同)。

具体地说，当执行文件上传 (put) 操作时，首先 Client 会向 NameNode 发起请求，NameNode 会告知 Client 应该把文件的每个块分配到哪些 DataNode 上。Client 拿到这个信息后，会依次联系 DataNode 并向其传送对应的块。每传送成功一个 DataNode，Client 就要向 NameNode 告知一次成功，NameNode 记录下这个副本的所在位置。

如果中途出现了异常，Client 会停止操作，并向用户返回错误信息。但之前传送成功的块不会被丢失。

在最坏情况下，Client 向 DataNode 传送了块后就出现异常，不能成功地向 NameNode 注册。这种情况的最坏后果也只是同一个块多次地上传到同一个 DataNode 中，不会带来数据丢失等恶性后果。这个设计遵循了 ‘‘至少一次’’。

3.5 信息存储

显然 NameNode 和 DataNode 需要具有存储能力。SDFS 的实现是让 NameNode 和 DataNode 将信息存储到本地文件系统的文件中。

当 DataNode 收到某个块后，它会产生一个特殊命名的存储文件。这个存储文件的命名由 (原文件名，块号) 唯一决定。

当 NameNode 收到信息，它会将状态存储到一个对象中，并写入文件以持久化存储。

信息存储有一个隐含的 ‘‘写写冲突’’ 问题。这是因为每个服务事实上是多线程的。在每个 rpc 请求到来时，rpc 库会为此产生一个线程用以响应。而许多数据是不线程安全的。

4 模块间的介绍和联系

4.1 模块介绍

本项目用到了以下的模块

- parser(code/client/parser.py)。该模块解析命令行输入。
- runner(code/client/sdfs.py)。用户的命令行工具。用户直接执行的脚本。
- connector(code/client/connector.py)。连接用具。负责和 DataNode 和 NameNode 作连接。是客户端对复杂逻辑的一层抽象
- register(code/server/rpyc_registry.py)。rpyc 库自带的模块，用于服务注册。
- NameNode(code/server/namenode.py)。NameNod 的核心实现。
- DataNode(code/server/datanode.py)。Datanode 的核心实现

4.2 模块间的联系

用户直接调用 sdfs.py 文件。该文件使用 parser 作命令行解析，然后调用 connector 执行实际的任务。connector 为 sdfs.py 提供了及其简单的抽象。

connector 有比较复杂的逻辑。对于不同的操作（例如 ls、cat、put 等），每个操作构成一个成员函数。它负责实际地与 NameNode 和 DataNode 作数据传送，并进行错误处理，尽可能维持足够高的鲁棒性。

NameNode 和 DataNode 在一般情况下不会直接交互（除了 NameNode 发起的 block 状态检查）。一般情况下，connector 向 NameNode 询问到足够的信息后，直接向 DataNode 请求执行相关的操作。

5 rpyc 库简介

之所以需要简介 rpyc 库，是因为其涉及到多线程和服务注册的概念，还一致性的讨论有关。代码 4是 rpyc 库的一个使用模板，介绍了它的大致用法。

rpyc 库要求将任务以类的方式实现。每个类会成为一个服务 (service)。

rpyc 库要求每个类以 Service 这个单词作为结尾。Service 前的部分即为该服务的服务名。每个服务可以实现 on_connect 和 on_disconnect 函数。在本项目中，NameNode 会在 on_connect 中向需要的 DataNode 作连接，并在 on_disconnect 中对它们作连接的释放。

每个成员函数名如果以 exposed_ 开头，则该函数成为服务提供的 api。其他服务可以远程调用该函数 (不需要 exposed_ 部分)。例如代码 4中暴露了 put 这个 api。

产生该服务器使用的是 ThreadServer 这个函数。它会为每个尝试建立起的 rpc 请求产生一个新的线程。这意味着可能会产生写冲突问题。

代码 4: rpyc 库简介

```
class NameNodeService(rpyc.Service):
    def __init__(self, storage_path=None):
        # ...

    def on_connect(self, conn):
        print('OPEN - {}'.format(conn))
        # ...
    def on_disconnect(self, conn):
        print('COLSE - {}'.format(conn))
        for conn in self.conns:
            conn.close()
    def exposed_put(self, filename, filesize, replica=3, blocksize=1024):
        '''
        NameNode 回应存放文件的方式
        @param filename :: Str 文件名
        ...
        @ret Int, [[Tuple(Str, Int)]],
        ...
        '''
        # ...
        # return ...

if __name__ == '__main__':
    from rpyc.utils.server import ThreadedServer
    t = ThreadedServer(NameNodeService, port=PORT, protocol_config={
        'allow_public_attrs': True,
    }, auto_register=True)
    print('start at port {}'.format(PORT))
    t.start()
```

6 具体代码 - 以 put 为例

由于代码比较复杂，此处仅以一个具体操作作为例子，讲述从客户端到底层的代码实现。

6.1 put 操作的分发

在执行到 put 操作时，首先 sdfs 会调用 parser 解析用户命令，并在 connector 处作分发 (**??**)。

分发实际做的操作是调用 connector 的 put 成员函数，并向该函数提供文件名。该文件名指向本地文件系统中的一个实际存在的文件。对 parser 的具体调用过程略。

代码 5: put 命令的解析和分发

```python
# in code/client/sdfs.py
class Dispatcher:
    def __init__(self):
        # ...
    def parse(self):
        self.args = parser.parse_args()
        self.dispatch()
    def dispatch(self):
        try:
            if self.args.which == 'put':
                success, msg = self.conn.put(self.args.file)
                if not success:
                    self.parser.print_help()
                print(msg)
        # ...
```

6.2 connector 的执行

connector 的代码（代码 6）展示中省去了函数头注释，原始代码见 code/client/connector.py。

首先 put 函数会尝试打开文件 filename，并以二进制的形式读入数据。然后 connector 会尝试远程调用 NameNode 的 put 方法。该方法不会真正执行 put，而是会返回这个文件的每个分块应该存储到的 DataNode 的地址。

datanodes 可以看做列表的列表。datanodes 的第 i 个元素是一个列表，表示文件的第 i 块应该放在哪些 DataNode 中。在代码的外层 for 中，遍历 datanodes，变量 idx 指当前执行到文件的第 idx 块。

wrtie_binary 变量存储的是第 idx 块的具体信息。值得注意的是，文件的最后一块的大小很可能跟前面的 n-1 块不一样，要特殊处理。

对每个块，以及这个块应该存储进的每个 DataNode，connector 会尝试调用 put_block 函数，并向 NameNode 注册这一 put_block 操作。如果全部操作都正常完成了，则程序错误码 0 和提示信息。否则返回非 0 的错误码。

6.3 NameNode 的执行

NameNode 被调用 put 时，应该向 connector 返回文件的每个 block 应该存储到的 DataNode 的地址 (代码 7)。

首先 exposed_put 会判断现有的 DataNode 是否足够请求副本的个数。如果不足，则返回错误码 1。如果足够，则不断地对可用的 DataNode 的索引 (available_idx) 做乱序，然后选择前 replica 个返回。

6.4 DataNode 的执行

DataNode 程序十分简单，如代码 8所示。

代码 6: connector 的具体代码

```
def put(self, filename, blk_sz=16384, replica=3):
    with open(filename, 'rb') as f:
        binary = f.read()
        size = len(binary)
        namenode_conn = rpyc.connect(self.ip, self.port)
        errno, datanodes = namenode_conn.root.put(filename,
            size,
            blocksize=blk_sz,
            replica=replica)
        if errno == 1:
            return False, 'Not enough DataNode for replica'

        # 告知namenode, 并对datanode作IO
        for idx, segment in enumerate(datanodes):
            if (idx + 1) * blk_sz <= size:
                write_binary = binary[idx * blk_sz: (idx + 1) * blk_sz]
            else:
                write_binary = binary[idx * blk_sz :]
            for datanode in segment:
                ip, port = datanode
                conn = rpyc.connect(ip, port)
                # 将该块传给datanode
                errno, msg = conn.root.put_block(filename, idx, write_binary)
                if errno == 1:
                    print('Warning: {}-{} alread exists {} block {}'.
                        format(ip, port, filename, idx))
                namenode_conn.root.put_block_registry(filename, idx, datanode)
    return True, 'success'
```

代码 7: NameNode 的 put 函数实现

```
def exposed_put(self, filename, filesize, replica=3, blocksize=1024):
    ret = []
    datanodes = self.datanodes
    if len(datanodes) < replica:
        # DataNode的数量不足以创建副本
        return 1, []
    # block的数量
    block_num = math.ceil(filesize / blocksize)
    available_idx = [i for i in range(len(datanodes))]
    for _ in range(block_num):
        # 随机挑选replica个可用的DataNode
        random.shuffle(available_idx)
        chosen_idx = available_idx[:replica]
        ret.append([datanodes[i] for i in chosen_idx])
    return 0, ret
```

首先 DataNode 需要检查 self.storage_path 这个目录是否存在，如果不存在则创建目录（利用 mkdir）。然后 DataNode 计算出存储的文件名，它是 “目录/文件名-块号”。如果这个文件名已经存在且没有指定强制执行 (force)，那么这个操作错误，返回错误码 1。否则，DataNode 向文件系统写入文件，并返回错误码 0。至此，put 操作基本完成。connector 收到所有的正确信号后，会向客户端提示操作成功。

代码 8: DataNode 的 put_block 的实现

```
def exposed_put_block(self, filename, blockid, binary, force=False):
    '''
    将block存储到本地文件中
    @param filename :: Str, 存储的文件名
    @param blockid :: Int,  第blockid块
    @param binary :: bStr, 待写入的数据

    @ret Tuple(Int, Str) :: errno and message
        - 0: no error
    '''
    if not os.path.isdir(self.storage_path):
        os.system('mkdir {}'.format(self.storage_path))
    target_filename = '{}/{}-{}'
        .format(self.storage_path, filename, str(blockid))
    if not force and os.path.exists(target_filename):
        return 1, 'block already exists'
    with open(target_filename, 'wb') as f:
        f.write(binary)
    return 0, 'ok'
```

7　结果展示

7.1　help 和 node 指令

可以对 sdfs 工具的任何一个命令作--help 操作，它会输出更详细的命令介绍。同时 sdfs.py node 指令可以返回当前连接正常的 DataNode。(图 3)

7.2　put、get 和 ls 操作

put、get 和 ls 操作的效果展示见图 4。

put 操作完成后，ls 操作成功地看到了 SDFS 中创建了 data-Trie.png 文件，而且看到了分配到 5 个块。在随后的 get 操作中，成功地将该图片传送回了本地，且重命名为 data-Trie2.png。

diff 指令可以证明 SDFS 的正确性，两者的文件内容完全一致。

7.3　容错

7.3.1　副本被篡改

下面尝试手动地篡改副本内容，形成 “数据破坏” 的效果。然后观察 SDFS 如何正确地将文件副本返回。

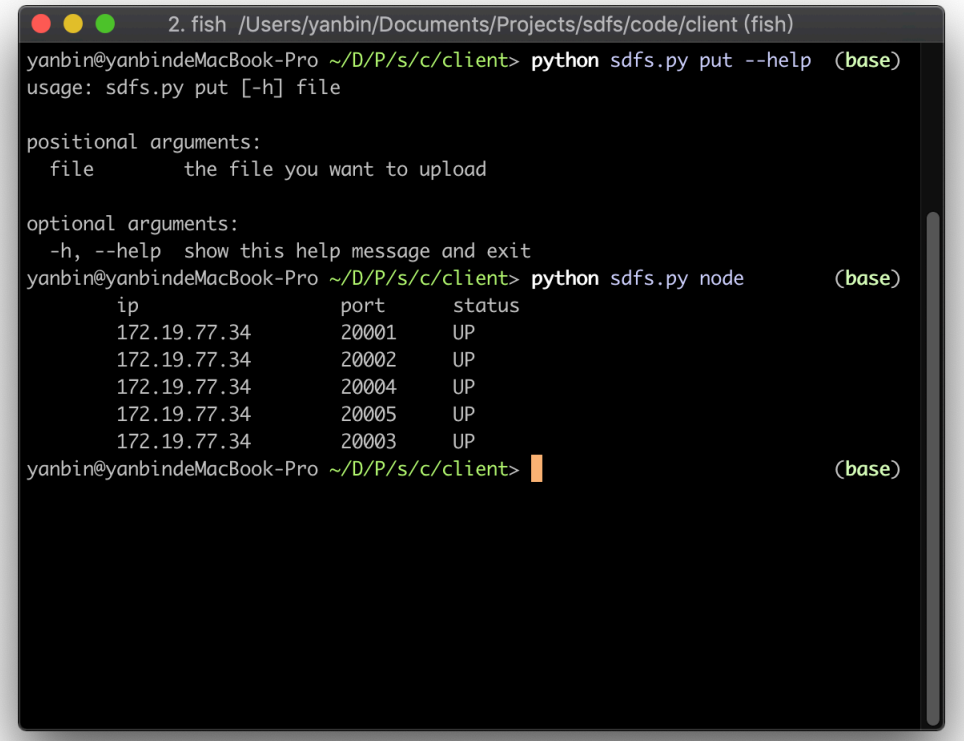

图 3: help 和 node 指令的示意图

如图 6所示，先输入 ls --all。其中--all 指令会触发 “检查全部副本的正确性” 这一操作。从图中可以看到，15 个副本中有 5 个副本已经被破坏了。(同一块中没有 2 个副本被同时破坏，否则就恢复不回来了)。

图 6中可以看到，get 操作依旧正确地把未损坏的副本返回。而且 diff 指令验证了他们的正确性。
Node: 如果所有的副本都是完整的，那么任何 2 台服务器的停止性失效都不会造成问题。这部分的演示省略。

上面所展示的情况是极端情况。每个块恰好有 3 个副本，能允许 1 个副本失效。上面的情况刚刚好是每个块恰有一个副本失效了。如果再有更多的副本失效，SDFS 无法判断哪些块是正确的块，故文件就被彻底地损坏了（图 7)。

Note: 为了找到哪些块已经失效了，NameNode 会向 DataNode 请求所有块的 md5 摘要，然后 NameNode 会以少数服从多数的方法判断哪些块是好块。这个操作的性能代价是很低的，因为 md5 的体积很小。

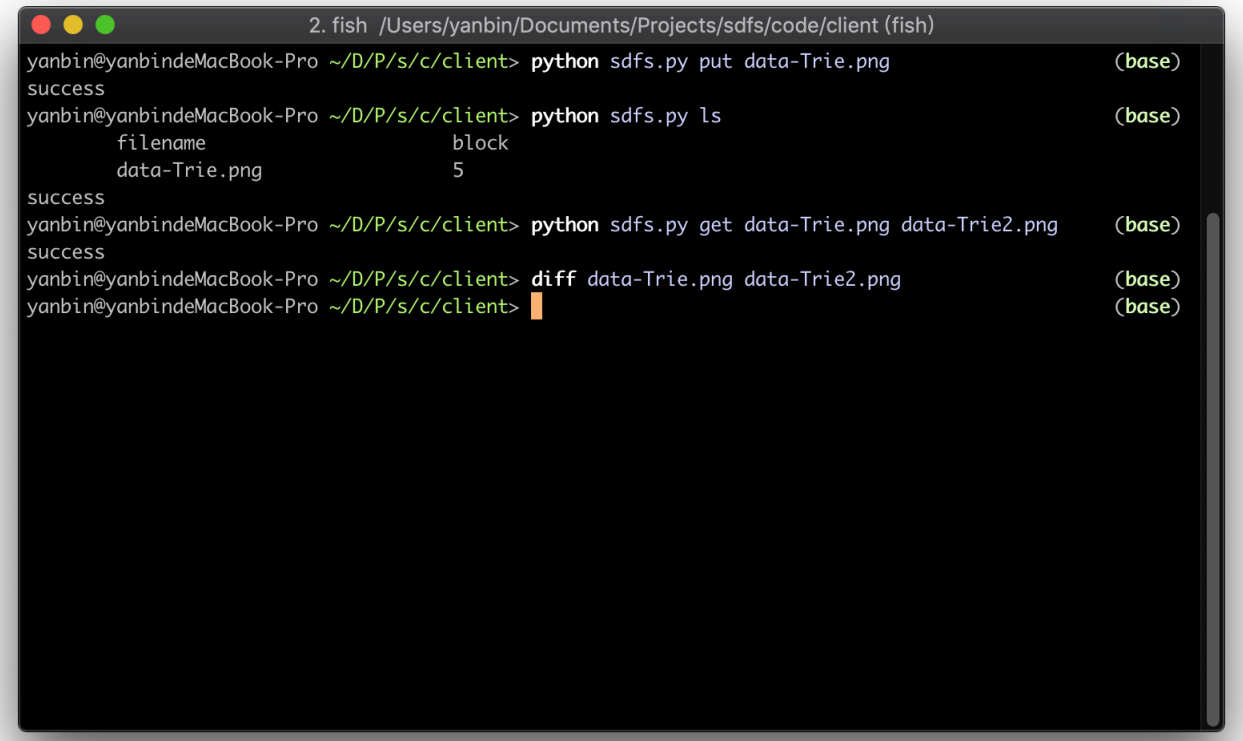

图 4: put、get 和 ls 操作效果展示

图 5: data-Trie.png 和 data-Trie2.png 完全一样

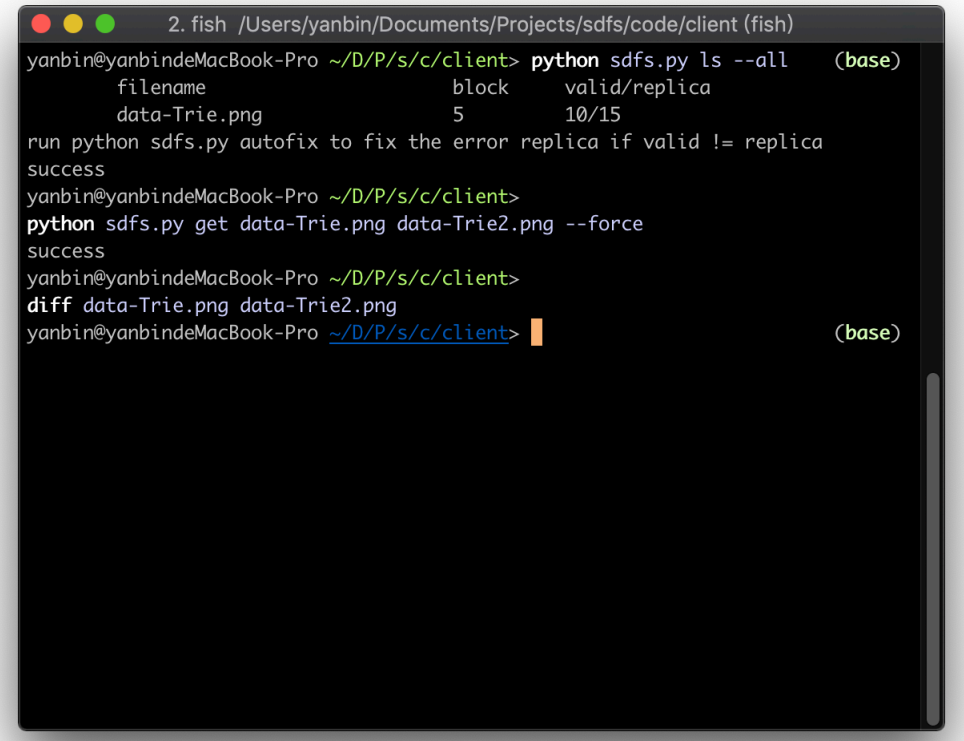

图 6: 副本被破坏时依旧返回正确的内容

8　一致性和容错的细节补充

8.1　读写锁

在本次项目的实现中遇到了需要采用读写锁的问题，否则会产生写冲突。

由于 NameNode 是多线程的，对它的写很可能会产生冲突。如果多个客户端尝试对同一个文件执行命令，NameNode 中的簿记信息很可能会丢失，具体表现为后来的信息覆盖之前的信息。

解决这个的方法是加锁。为了最大限度地提高性能，避免在 NameNode 上完全串行化，这里采用读写锁的机制。

在 SDFS 支持的操作中，get、cat、ls 操作都是非写操作，只需要申请读锁（共享锁）即可。而 put 和 ls 操作是写操作，需要申请写锁（互斥锁）。读写锁机制能最大限度地提升程序的并发度，这是因为现实情况中，读操作比写操作多很多。

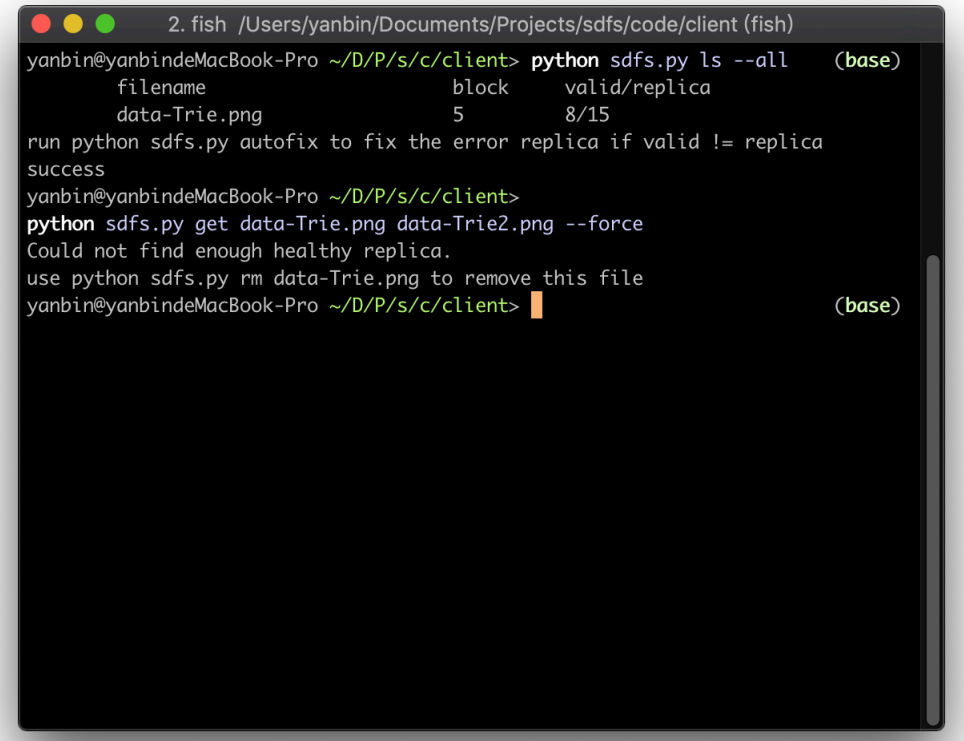

图 7: 更多的副本失效的情况下，无法得到原文件

8.2 坏块发现

NameNode 需要发现坏块，避免用户下载到损坏的文件。

HDFS 的实现是 Namenode 定时查看各个块的损坏情况，并修补坏块。但这个实现过于复杂。在本项目 SDFS 中，采取 “用户指令触发检索坏块” 的操作。

具体地说，每当用户执行 ls --all 操作或 get 操作时，先进行检索坏块，再执行之后的操作。

检索坏块由 NameNode 的辅助函数 fresh_update 完成。它的具体做法是，向所有 DataNode 请求所有的块。考虑到这个操作太耗时，本项目实际请求的是块的 md5 摘要。然后 NameNode 比较所有的 md5 摘要。出现次数多的块认为是好块 (healthy)。

这里有一个细节上的问题。如果出现次数最多的块有两个，那么全部的块都应标记为失效 (invalid)。由于 SDFS 默认副本为 3，于是只要 2 个副本被损坏，则整个块就会被彻底损坏，于是文件将彻底失效 (图 6展示了这种情况)。

还有一个细节上的问题是，如果两个损坏的块恰好其 md5 相同，导致 SDFS 认为他们健康，该怎么解决。实际上，数学上可以保证这种情况出现的概率可以忽略不计。

9 实验总结

本次实验实现了 SDFS（Simple Distributed FileSystem），它是一个有容错、缓存功能的分布式文件系统。它采用类似于 HDFS 的架构，将上传的文件分块存储到各个 DataNode 中，并由 NameNode 统一管理元数据。

SDFS 仅仅基于 Python 的 Rpyc 库，仅利用 RPC 请求的方式做到主机间的信息传输。使用 Python 是因为它的 rpc 库用法比较简单，抽象层次高，能把主要注意力集中在代码逻辑的开发中。

由于本实现各个模块间是高度解耦的，它有很强的拓展性，可以很容易地为其添加新功能。

经过这次实验，我对分布式的许多概念了解得更加透彻，把课本的知识和实践结合到了一起。