实 验 报 告 内 容

一・实验内容

实验题目：HDFS 编程实践

实验内容与要求：

1. 使用 HDFS 文件操作的常用 Shell 命令；
2. 利用 Hadoop 提供的 Java API 进行基本的文件操作。

二・实验目的

1. 理解 HDFS 在 Hadoop 体系结构中的角色。
2. 熟练使用 HDFS 操作常用的 Shell 命令。
3. 熟悉 HDFS 操作常用的 Java API。

三・实验过程截图及说明

1. 实验环境与前提

本实验基于 Hadoop 伪分布式环境进行操作，HDFS 的访问地址为 `hdfs://namenode:9000`。所有命令与示例程序均在 `lab-client` 容器中执行，工作目录为 `/workspace`。

进入容器命令如下：

```
docker exec -it lab-client bash
```

2. 使用 HDFS 文件操作的常用 Shell 命令

在 `/workspace` 目录下依次执行下列命令，完成目录创建、文件生成、上传、列目录与查看内容：

```
cd /workspace

hdfs dfs -mkdir -p /user/root/input

echo "Hello CachyOS Hello Hadoop" > test.txt

hdfs dfs -put test.txt /user/root/input/

hdfs dfs -ls /user/root/input/

hdfs dfs -cat /user/root/input/test.txt
```

图 1 给出了命令执行过程与输出结果，可见 `test.txt` 已成功上传至 HDFS，并能通过 `hdfs dfs -cat` 正确读取。

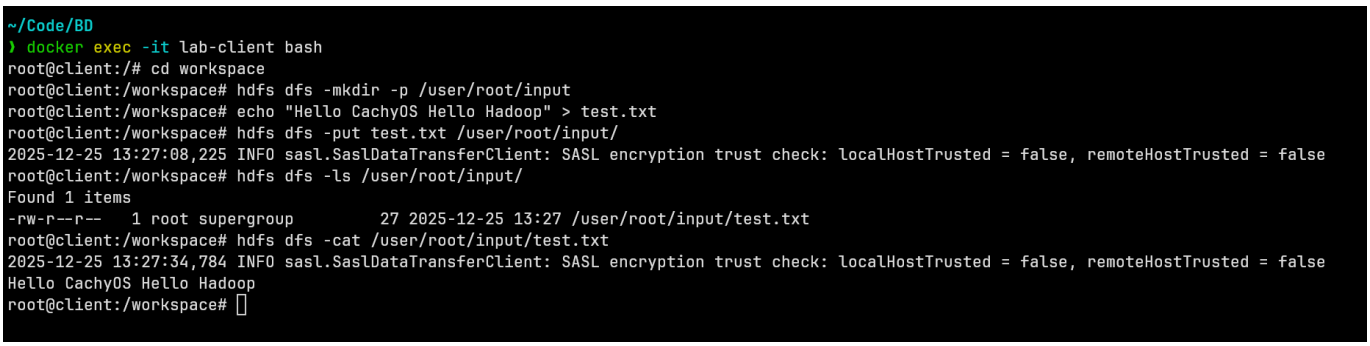

图 1 HDFS Shell 命令操作与输出

3. 利用 Hadoop 提供的 Java API 进行基本的文件操作

本部分使用 Hadoop Java API 对 HDFS 执行一组可重复的文件操作流程，覆盖创建、读取、重命名、状态查看与删除，验证 `FileSystem`、`Path`、`FSDataOutputStream`、`FSDataInputStream` 等核心类的使用方式。

程序 `HDFSTest.java` 如下：

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileStatus;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URI;

public class HDFSTest {

    private static final String HDFS_URI = "hdfs://namenode:9000";
    private static FileSystem fs;

    public static void main(String[] args) {
        try {
            Configuration conf = new Configuration();
            fs = FileSystem.get(URI.create(HDFS_URI), conf, "root");

            Path filePath = new Path("/user/root/input/java_crud_test.txt");
            Path renamedPath = new Path("/user/root/input/java_crud_renamed.txt");

            System.out.println(">>> STARTING HDFS CRUD TEST <<<");

            System.out.println("\n[1] TEST: Create & Write");
            createFile(filePath, "Hello Hadoop! This is a test for CRUD operations.");

            System.out.println("\n[2] TEST: Read");
            readFile(filePath);

            System.out.println("\n[3] TEST: Rename");
            renameFile(filePath, renamedPath);

            System.out.println("\n[4] TEST: List File Status");
            listFileStatus(renamedPath);

            System.out.println("\n[5] TEST: Delete");
            deleteFile(renamedPath);

            System.out.println("\n>>> ALL TESTS PASSED SUCCESSFULLY! <<<");

            fs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void createFile(Path path, String content) throws Exception {
        FSDataOutputStream os = fs.create(path, true);
```

```
        os.write(content.getBytes());
        os.close();
        System.out.println(">>> Success: Created file " + path.toString());
    }

    private static void readFile(Path path) throws Exception {
        if (!fs.exists(path)) {
            System.out.println(">>> Error: File not found!");
            return;
        }
        FSDataInputStream is = fs.open(path);
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        String line = br.readLine();
        System.out.println(">>> Content: " + line);
        br.close();
        is.close();
    }

    private static void renameFile(Path oldPath, Path newPath) throws Exception {
        if (fs.rename(oldPath, newPath)) {
            System.out.println(">>> Success: Renamed " + oldPath.getName() + " to " +
newPath.getName());
        } else {
            System.out.println(">>> Error: Rename failed!");
        }
    }

    private static void listFileStatus(Path path) throws Exception {
        FileStatus status = fs.getFileStatus(path);
        System.out.println(">>> File Path: " + status.getPath());
        System.out.println(">>> Owner: " + status.getOwner());
        System.out.println(">>> Size: " + status.getLen() + " bytes");
        System.out.println(">>> Replication: " + status.getReplication());
    }

    private static void deleteFile(Path path) throws Exception {
        if (fs.delete(path, true)) {
            System.out.println(">>> Success: Deleted file " + path.toString());
        } else {
            System.out.println(">>> Error: Delete failed!");
        }
    }
}
```

编译与运行命令如下（在容器内执行）：

```
javac -cp `hadoop classpath` HDFSTest.java
java -cp .:`hadoop classpath` HDFSTest
```

图 2 给出了代码与运行结果，输出包含文件创建成功、读取到期望内容、重命名成功、状态信息可查询以及最终删除成功，证明 Java API 能完成基本的文件操作链路。

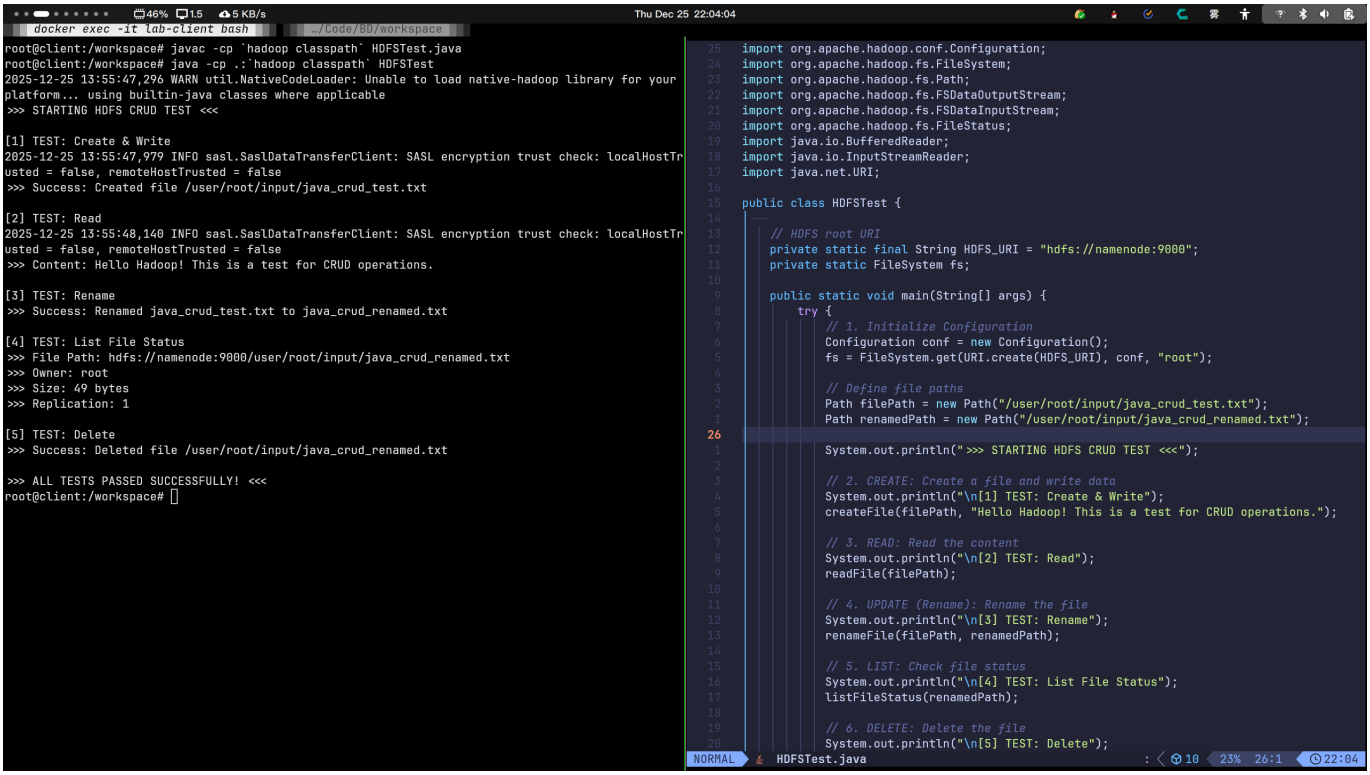

图 2 HDFS Java API（CRUD）代码与运行结果

四・实验总结及心得体会

实验总结：本实验完成了 HDFS Shell 命令的基本文件操作，并通过 Java API 对 HDFS 进行创建、读取、重命名、状态查询与删除等操作验证，满足实验内容与要求。

心得体会：Shell 操作适合快速验证与排障，Java API 更适合将 HDFS 操作嵌入业务程序并形成可复用的逻辑单元。实践中 `hadoop classpath` 的正确配置直接决定编译与运行是否成功；同时在 HDFS 上进行“修改”通常以重命名或重写的方式实现，需要在程序设计时明确其语义与代价。