|  |
| --- |
| 一、实验目的   1. 本次实验的目的是掌握 Java 企业级应用开发工具的使用方法，掌握 Java 面向对象程序编程技术，掌握常用工具类的使用。理解面向对象的分析和设计方法，理解和使用设计模式。 2. 设计开发完成一个基于命令行的文件资源管理软件。熟练掌握文件资源的处理方法和工具类。理解多线程工作机制。提高 OOP 能力。 3. 抄袭计 0 分。 |
| 二、实验项目内容  设计开发完成“基于命令行的文件管理器”软件。要求如下：  1、可以设置当前工作文件夹，默认在当前文件夹下进行文件的增删改查操作，文件夹的增删改查操作。也可以操作指定的文件夹。  2、实现当前文件夹下的内容罗列；可以根据文件名、文件大小、文件类型、文件日期等过滤特定类型的文件。罗列的时候可以排序，依据名称、大小、时间等。罗列文件的时候，显示要规范。  3、可以直接查看和显示一个文本文件的内容。  4、实现文件拷贝和文件夹拷贝（文件夹拷贝指深度拷贝，包括所有子目录和文件）；能指定目标名称和位置，能计算拷贝时间，能显示拷贝进度。5、可以利用线程机制，支持后台长时任务异步执行。不影响前端操作。  例如在拷贝大文件或者文件夹的时候，可以选择后台执行，还是前台执行。如果前台执行，显示时间和进度。  6、可以对指定文件进行加密和解密；加密后形成新的文件，可以指定文件名。加密与解密方式自己选择。  7、可以对文件或者文件夹进行压缩，或者解压。压缩与解压方式自己选择。  8、可以实现自定义功能。注意操作使用的方便性，注意类和类之间的关系。充分利用继承，多态等特性，使用上抽象类，接口，泛型，内部类等设计元素，使用好集合类、多线程、IO 工具类、NIO 工具类等。注意程序的总执行流程和分支执行流程。注意设计思想的表达，注意优化代码结构，优化 |

|  |
| --- |
| 类的职责分工，注意使用设计模式。代码有注释。  9、在报告中注明自己的创新点、特色等。  10、提交：（1）本实验报告，（2）源代码压缩文件 zip，（3）软件演示的 MP4 视频，视频大小不超过 40M，视频请在 **QQ** 浏览器测试能否正常播放。注意源代码加注释。注意文件名称的规范性。文件名：学号姓名 2.docx， 学号姓名 2.zip, 学号姓名 2.mp4。三个文件分别提交。 |
| 三、实验过程或算法（写明：1 创新点或特色、2 设计思想、3 设计模式的使用、4 程序的结构或者架构、5 功能组织图（思维导图）、6 类的说明和类之间的关系图（UML 规范的类图）、7 程序主要执行流程图（符合行业规范）， 8 是核心源代码、截图等）  1 **创新点或特色**   1. 仿Linux终端的命令行界面，且指令大部分与Linux命令一致，例如cd, ls, find, cat, touch, mkdir, rm, mv, cp, gpg, zip, unzip。总计实现35个不同具体功能的指令。指令行上会显示当前所在文件夹。      1. 在命令模式的使用上，新增了一个commandDetail类，专门用于充当命令调用类和命令类之间传递信息的中介，方便扩展命令接口所需要的信息。 2. 操作界面友好   提供了完整的指令帮助文档，输入help指令即可获取，有包括指令模板，指令功能，使用样例，特殊说明。指令的报错也用红色字体与其他系统提示信息分割开，更明显。用仿mysql表格的形式打印文件目录，使用全角字符以实现中英文字符对齐的问题。     1. 使用文件锁和synchronized关键字来确保线程和文件的安全，避免冲突。      1. 使用过滤器模式的设计模式来进行文件过滤，并功能完善。   名称过滤可过滤出所有按顺序含有指定字符串所有字符的文件，并按一定策略计算名称与该字符的相关度来进行过滤后的文件的排序。  大小和日期排序也是可以设置区间来进行过滤，也可以单区间过滤。且文件大小支持不同单位的识别比如MB，KB，B。  文件日期也可自动识别字符串，最高可精确到秒，要求最低输入到日即可。   1. 拷贝、加密解密、压缩解压等均可选择后台执行。前台执行时会在一行内动态更新输出进度的百分比和已花费的时间，使得界面更简洁美观，而不是一行一行的不断输出，使提示信息过于冗余。后台执行时，也可用backstage命令来查看线程的执行情况，是否执行完成。        1. 在实验要求基础上，扩展了一系列功能。   例如，cd命令中包括切换上级目录、切换到主目录、切换到上次访问的目录；查看txt文件内容时可以设置添加行号进行查看；touch创建文件；mkdir创建目录；rm删除文件或文件夹；mv重命名文件或文件夹；backstage显示后台线程的执行情况。   1. 使用了工厂模式，单例模式，命令模式，策略模式，过滤器模式等设计模式。代码共8个packet，46个类。充分进行模块化设计。      1. 充分使用java的各种特性，多线程、IO 工具类、NIO 工具、继承、多态、抽象类、接口、泛型、内部类、集合类、注解、lambda表达式。 2. **设计思想**   （1） 模块化设计  将代码拆分成多个方法或方法类来分模块进行编写，提高代码的可读性和编写代码时的条理性。将代码分为了8个packet，46个类。包括命令包，工厂包，模型包，过滤器包，策略包，单例包，工具包，客户端包。  代码整体结构为主函数下创建三个单例当前目录管理器、命令解析器、线程池。将输入的命令和当前目录管理器以及线程池传给命令解析器。再由命令解析器去分解命令，调用对应的命令类。  （2）封装  数据和对数据的操作都封装起来，限制其他部分的代码直接访问，保证数据的安全性，并使用工厂类进行创建模型。例如下图：    （3）设计模式  使用了工厂模式、单例模式、命令模式、过滤器模式、策略模式等设计模式，根据设计模式的理念去创建大部分的类和划分了代码包结构  （4） 可扩展性和可维护性  由于进行模块化编程，在新添加功能时，几乎不用更改之前的代码，只需创建新功能对应的方法或类就可以实现。代码出现bug时，也只需跟踪到出错的方法或类中，修改对应的方法或类即可，而不用关心一大串连在一起的代码。  （5） 重视用户体验  提供了完整的指令帮助文档，输入help指令即可获取，有包括指令模板，指令功能，使用样例，特殊说明。  针对用户各种可能输入错误指令的情况，设计了指令错误提示的特定输出格式，并提示用户重新输入命令。指令的报错也用红色字体与其他系统提示信息分割开，更明显。  （6）重视代码的复用  由于一直进行模块化编程，很多方法类可以在多个程序中发挥作用。比如第一次实验的打印表格的工具类。不经过修改或进行略微修改就可达成很好地效果。这也是激励一直进行模块化编程的一个动力。  （7）重视对java各种特性和工具的使用，在实践中磨炼对java技术的掌握和理解。充分使用多线程、IO 工具类、NIO 工具、继承、多态、抽象类、接口、泛型、内部类、集合类、注解、lambda表达式。例如下面的举例  线程池、集合类：    IO/NIO    继承，多态，接口，抽象类（可从目录看出来）    泛型    内部类、注解    lambda表达式     1. **设计模式的使用** 2. 工厂模式   用途：更安全和方便地扩展对实体对象的初始化。  实现：创建了文件的抽象类和工厂的接口     1. 单例模式   用途：确保一个类只有一个实例，并提供一个全局访问点来访问该实例。  实现：创建了当前目录管理器，线程池和命令解析器等单例     1. 策略模式   用途：将每个算法封装起来，使它们可以互换使用。  实现：实现当前目录下文件按名称，大小，时间等排序。     1. 命令模式   用途：将请求封装为一个对象，允许用户使用不同的请求对客户端进行参数化。  实现：将对文件进行操作的命令都封装成命令类，继承命令接口。     1. 过滤器模式   用途：用于将对象的筛选过程封装起来，允许使用不同的筛选标准动态地筛选对象。  实现：实现对文件进行名称、时间、大小、类型等进行过滤    4 **程序的结构或者架构**  将代码分为了8个packet，46个类。下面目录树从上到下分别是客户端包，命令包，工厂包，过滤器包，模型包，单例包，策略包，工具包。  程序的的起点是Main.java，在其中使用三个单例：命令解析器CommandParser.java，当前目录管理器CurrentDirectory.java和线程池TaskPool.java，由这三个大模块完成所有的操作。命令解析器解析所有的输入指令并调用方法或命令类进行执行，当前目录管理器管理当前目录下的文件对象、排序策略等，线程池负责处理命令类中支持多线程后台操作的命令。其中，命令解析器是程序的核心。  Java  ├─client  │ Main.java (主函数)  │ Test.java  │  ├─command  │ Command.java (接口)  │ CopyFileCommand.java (拷贝命令)  │ CreateFileCommand.java (创建文件命令)  │ DecryptFileCommand.java (解密命令)  │ DeleteFileCommand.java (删除命令)  │ EncryptFileCommand.java (加密命令)  │ HelpCommand.java (获取指令帮助文档命令)  │ ReadFileCommand.java (读txt文档命令)  │ RenameFileCommand.java (重命名命令)  │ UnzipFileCommand.java (解压命令)  │ ZipFileCommand.java (压缩命令)  │  ├─factory  │ CommandDetailFactory.java  │ DirectoryFactory.java  │ Factory.java (工厂类接口)  │ FileFactory.java  │  ├─filter  │ DateFilter.java (日期过滤)  │ Filter.java (过滤器接口)  │ NameFilter.java (名称过滤)  │ SizeFilter.java (大小过滤)  │ TypeFilter.java (类型过滤)  │  ├─model  │ AbstractFile.java (抽象文件类)  │ CommandDetail.java (为命令类传递信息的类)  │ DirectoryModel.java (文件夹类)  │ FileModel.java (文件类)  │ Model.java (接口)  │ TaskInfo.java (线程类)  │  ├─singleton  │ CommandParser.java (命令解析器（核心）)  │ CurrentDirectory.java (当前目录管理器)  │ TaskPool.java (线程池类)  │  ├─strategy  │ NameStrategy.java (名称排序)  │ SizeStrategy.java (大小排序)  │ Strategy.java (策略接口)  │ TimeStrategy.java (时间排序)  │  └─tool  CalculateDirectorySize.java  CopyFile.java  DecryptFile.java  EncryptFile.java  GetFileType.java  ParseDate.java  PrintTable.java  PrintTable2.java  UnzipFile.java  WarnOut.java  ZipFile.java  5 **功能组织图（思维导图**）    6 **类的说明和类之间的关系图（UML 规范的类图）**  各个类的说明在程序结构的树形图中已做说明。  下面的类图中Main主函数类用红色标注，绿色标注的是接口，标签较大的是程序的几个主要管理类。可以清晰看到命令解析器CommandParser是程序的核心，连接了大部分方法类和管理类。Main主函数类连接了三个主要管理类。  Main类是程序的的起点，在其中使用三个单例：命令解析器CommandParser.java，当前目录管理器CurrentDirectory.java和线程池TaskPool.java，由这三个大模块完成所有的操作分配或信息管理。命令解析器解析所有的输入指令并调用方法或命令类进行执行，当前目录管理器管理当前目录下的文件对象、排序策略等，线程池负责处理命令类中支持多线程后台操作的命令。命令解析器是程序的核心。      7 **程序主要执行流程图（符合行业规范）**    8 **核心源代码、截图等**  **主函数：**    **命令解析器（部分省略）：**  *//命令解析器（单例模式）* public class CommandParser {  *//设置单例* private static CommandParser *commandParser*=new CommandParser();  private CommandParser(){}  public static CommandParser getInstance(){  return *commandParser*;  }   *// ANSI 颜色常量* public static final String *RESET* = "\u001B[0m"; *// 重置颜色* public static final String *RED* = "\u001B[31m"; *// 红色* private String formerPath="";   public void parse(String order,CurrentDirectory currentDirectory,TaskPool taskPool)  {  *// 使用空格分隔指令* String[] wordsArray = order.split("\\s+"); *// 以任意空格为分隔符  // 将指令数组转换为 ArrayList* ArrayList<String> orderList = new ArrayList<>(Arrays.*asList*(wordsArray));  int back=0;*//后台运行* if(orderList.get(orderList.size()-1).equals("-back"))  {  back=1;  orderList.remove(orderList.size()-1);  }  CommandDetail commandDetail=new CommandDetail();  commandDetail.setBack(back);  commandDetail.setTaskPool(taskPool);  switch (orderList.get(0))  {  case "cd":*// 切换目录* cd(orderList,currentDirectory);  break;  case "ls":*// 按策略展示文件夹目录* ls(orderList,currentDirectory);  break;  case "find":*// 过滤文件* find(orderList,currentDirectory);  break;  case "cat":*// 查看txt文件* cat(orderList,currentDirectory);  break;  case "touch":*// 创建文件* case "mkdir":*// 创建文件夹* create(orderList,currentDirectory);  break;  case "rm":*// 删除文件或文件夹* delete(orderList,currentDirectory);  break;  case "mv":*// 重命名文件* mv(orderList,currentDirectory);  break;  case "cp":*// 拷贝* CopyFileCommand copyFileCommand=new CopyFileCommand();  commandDetail.setSourcePath(*processPath*(orderList.get(1),currentDirectory));  commandDetail.setDestinationPath(*processPath*(orderList.get(2),currentDirectory));  copyFileCommand.execute(commandDetail);  break;  case "gpg":*// 加密* EncryptFileCommand encryptFileCommand=new EncryptFileCommand();  commandDetail.setPassword(orderList.get(1));  commandDetail.setSourcePath(*processPath*(orderList.get(2),currentDirectory));  if(orderList.size()==4)commandDetail.setDestinationPath(*processPath*(orderList.get(3),currentDirectory));  else commandDetail.setDestinationPath(*processPath*(orderList.get(2),currentDirectory));  encryptFileCommand.execute(commandDetail);  break;  case "ungpg":*// 解密* DecryptFileCommand decryptFileCommand=new DecryptFileCommand();  commandDetail.setPassword(orderList.get(1));  commandDetail.setSourcePath(*processPath*(orderList.get(2),currentDirectory));  if(orderList.size()==4)commandDetail.setDestinationPath(*processPath*(orderList.get(3),currentDirectory));  else commandDetail.setDestinationPath(*processPath*(orderList.get(2),currentDirectory));  decryptFileCommand.execute(commandDetail);  break;  case "zip":*// 压缩* ZipFileCommand zipFileCommand=new ZipFileCommand();  commandDetail.setSourcePath(*processPath*(orderList.get(1),currentDirectory));  if(orderList.size()==3)commandDetail.setDestinationPath(*processPath*(orderList.get(2),currentDirectory));  else commandDetail.setDestinationPath(*processPath*(orderList.get(1),currentDirectory));  zipFileCommand.execute(commandDetail);  break;  case "unzip":*// 解压* UnzipFileCommand unzipFileCommand=new UnzipFileCommand();  commandDetail.setSourcePath(*processPath*(orderList.get(1),currentDirectory));  if(orderList.size()==3)commandDetail.setDestinationPath(*processPath*(orderList.get(2),currentDirectory));  else commandDetail.setDestinationPath(*processPath*(orderList.get(1),currentDirectory));  unzipFileCommand.execute(commandDetail);  break;  case "help":*// 指令帮助文档* HelpCommand helpCommand=new HelpCommand();  helpCommand.execute(new CommandDetail());  break;  case "backstage":*// 查看后台运行情况* try {  Thread.*sleep*(1000); *// 等待片刻以让一些任务开始执行* } catch (InterruptedException e) {  Thread.*currentThread*().interrupt();  }  for (TaskInfo info : taskPool.getTaskInfoList()) {  System.*out*.println("后台任务: " + info.getTaskName() + " , 是否完成: " + info.isCompleted());  }  break;  default:  warning();  break;  }  }  public void warning() {  System.*out*.println(*RED*+"\*\*指令错误！您可以输入help指令来获取指令帮助文档\*\*"+*RESET*); }  *//处理相对路径* public static String processPath(String inputPath,CurrentDirectory currentDirectory) {  String processedPath;  *// 检查是否是绝对路径（以斜杠开头）* if (inputPath.startsWith("\\")) {  *// 去掉第一个斜杠* processedPath = inputPath.substring(1);  } else {  *// 对于相对路径，直接使用输入的路径* processedPath = currentDirectory.getCurrentPath()+"\\"+inputPath;  }  return processedPath; } *//处理带..的相对路径* public static String resolvePath(String currentPath, String relativePath) {  *// 创建当前路径的 Path 对象  //因为不知道为什么resolveSibling方法会从上一级目录开始计算.. ,所以手动加一层无用的文件夹* Path current = Paths.*get*(currentPath+"\\11");  *// 解析相对路径到当前路径* Path resolvedPath = current.resolveSibling(relativePath);  *// 获取规范化的绝对路径* return resolvedPath.toAbsolutePath().normalize().toString(); }  }  **当前目录管理器：**      **线程池**：  *//线程池* public class TaskPool {  *//设置单例* private static TaskPool *taskPool*=new TaskPool(10);  private TaskPool(int poolSize) {  this.executorService = Executors.*newFixedThreadPool*(poolSize);  this.taskInfoList = new ArrayList<>();  }  public static TaskPool getInstance(){  return *taskPool*;  }   private final ExecutorService executorService;  private final List<TaskInfo> taskInfoList;*//线程队列   //添加线程任务* public void submitTask(String taskName, Runnable task) {  TaskInfo taskInfo = new TaskInfo(taskName);  taskInfoList.add(taskInfo);   executorService.submit(() -> {  try {  task.run();  } finally {  taskInfo.setCompleted(true);  }  });  }   *//关闭线程池* public void shutdown() {  executorService.shutdown();  try {  if (!executorService.awaitTermination(60, TimeUnit.*SECONDS*)) {  executorService.shutdownNow();  }  } catch (InterruptedException e) {  executorService.shutdownNow();  }  }   public List<TaskInfo> getTaskInfoList() {  return taskInfoList;  } }  **命令详情传递器**：    **文件夹对象工厂（工厂示例）**：    **名称过滤器（过滤器示例）**    **名称排序策略（策略示例）：**    **压缩文件工具类（示例）**  *//传入参数1.源地址2.目的地址3.是否允许重写4.是否后台* public class ZipFile {  *// 记录压缩任务状态* private static long *totalSize* = 0; *// 记录要压缩的文件总大小* private static long *compressedSize* = 0; *// 记录已压缩的大小* public static void zipPath(String sourcePath, String zipFilePath, boolean overwrite,int back) throws IOException {  File sourceFile = new File(sourcePath);   *// 检查目标文件是否存在，且不允许覆盖* if (!overwrite && new File(zipFilePath).exists()) {  throw new IOException("目标文件已存在，且不允许覆盖：" + zipFilePath);  }   *// 获取源文件的总大小  totalSize* = *getDirectorySize*(sourceFile);  *compressedSize* = 0; *// 初始化已压缩大小* try (FileOutputStream fos = new FileOutputStream(zipFilePath);  ZipOutputStream zos = new ZipOutputStream(fos)) {  *zipFileOrDirectory*(sourceFile, zos, sourceFile.getName(),back);  }  }   *// 递归压缩文件或目录* private static void zipFileOrDirectory(File fileOrDir, ZipOutputStream zos, String zipEntryName,int back) throws IOException {  if (fileOrDir.isDirectory()) {  File[] files = fileOrDir.listFiles();  if (files != null) {  for (File file : files) {  *zipFileOrDirectory*(file, zos, zipEntryName + "/" + file.getName(),back);  }  }  *// 添加目录条目* zos.putNextEntry(new ZipEntry(zipEntryName + "/"));  zos.closeEntry();  } else {  *// 处理文件压缩* try (FileInputStream fis = new FileInputStream(fileOrDir)) {  ZipEntry zipEntry = new ZipEntry(zipEntryName);  zos.putNextEntry(zipEntry);   byte[] buffer = new byte[1024];  int length;  while ((length = fis.read(buffer)) >= 0) {  zos.write(buffer, 0, length);  *compressedSize* += length; *// 更新已压缩大小* if(back==0)*printProgress*(); *// 打印进度* }  zos.closeEntry();  }  }  }   *// 打印压缩进度与已花费时间* private static void printProgress() {  int progressPercentage = (int) ((*compressedSize* \* 100) / *totalSize*);  long seconds = (System.*currentTimeMillis*() - *startTime*) / 1000;  *// 将耗时转换为秒和毫秒* long milliseconds = (System.*currentTimeMillis*() - *startTime*) % 1000;   *// 打印进度信息* System.*out*.printf("\r压缩进度: %d%%, 已压缩大小: %d 字节, 已花费时间: %d.%03d 秒",  progressPercentage, *compressedSize*, seconds,milliseconds);  }   *// 获取目录的总大小* private static long getDirectorySize(File dir) {  long size = 0;  if (dir.isDirectory()) {  File[] files = dir.listFiles();  if (files != null) {  for (File file : files) {  size += *getDirectorySize*(file);  }  }  } else {  size = dir.length();  }  return size;  }   private static long *startTime* = System.*currentTimeMillis*(); *// 记录开始时间* }  **加密命令（命令示例）**  public class EncryptFileCommand implements Command{  private Object lock;  @Override  public void execute(CommandDetail details) {  String sourcePath=details.getSourcePath();  String destinationPath=details.getDestinationPath();  if(details.getBack()==1)  {  final Object lock=details;  details.getTaskPool().submitTask("加密 "+details.getSourcePath(), () -> {  synchronized (lock) { *// 确保任务的线程安全* try {  *// 在执行文件操作前获得文件锁* try (FileChannel channel = FileChannel.*open*(Paths.*get*(sourcePath))) {  try (FileLock fileLock = channel.lock(0, Long.*MAX\_VALUE*, true)) {  Thread.*sleep*(2000); *// 模拟长时间运行的任务* EncryptFile.*encryptFile*(details.getPassword(), details.getSourcePath(), details.getDestinationPath(), 1);  } catch (InterruptedException | NoSuchAlgorithmException | NoSuchPaddingException |  InvalidKeyException | IOException e) {  Thread.*currentThread*().interrupt();  }   }  }  catch (RuntimeException | IOException e) {  throw new RuntimeException(e);  }  }  });  }  else {  try {  EncryptFile.*encryptFile*(details.getPassword(),details.getSourcePath(),details.getDestinationPath(),0);  } catch (Exception e) {  e.printStackTrace();  }  }   } } |
| 四、实验结果及分析和（或）源程序调试过程（界面截图和文字）、实验总结与体会（必写）  **实验体会：**  由于一开始不熟悉IO和NIO工具类对文件的操作以及多线程的实现，一开始对程序的整体架构的设计有些困难。对我来说难点较多，以至于迟迟不知从何下手。主要难点是不确定怎么更合适地在方法类之间传递文件和文件夹对象使命令类能更好地对文件进行各种操作以及如何在其中融入多线程。  后来是选择先进行小模块的设计，先将压缩解压文件、读文件、加密解密文件的这些工具类写好（因为这个工具类不用太在意文件对象的传递方式，具体的文件操作命令的调用接口在命令类中去实现）。然后，在此基础上对IO，NIO工具类对文件的操作更熟悉后进行基本的设计模式的使用，并设计各个类的接口。再然后融入多线程的使用。这样就有条理地分步解决了实验的几个主要难点。由此感叹，任务总体困难时，能将难点有条理的分开、分布来解决的思想很重要。  在设计仿Linux系统的命令行终端和类似的指令时，也让自己对Linux的指令更加熟悉了解。  另外在上一次的实验基础上，这一次实验对设计模式的使用更加熟悉。每一次实验都在解决新的难点上完成了进步，下一次的实验不知道又有什么新的困难。    **实验结果：**  Help指令正确：    切换上一级目录正确：    切换主目录正确：    切换上次访问目录正确    切换绝对路径正确    切换相对路径正确    压缩指令正确且命令可自动识别相对路径和绝对路径      Ls切换排序策略指令正确    Find过滤指令均正确    Cat指令正确    Backstage查看后台，及后台线程运行均 正确    其他更完整的功能可在演示视频中一一得到功能正确的验证。  综上，程序所有功能均运行正确，实验完成！ |