I. Project structure

1. Catalog structure

The directory structure of a.git repository has changed because branch management and the reflog command have been implemented:

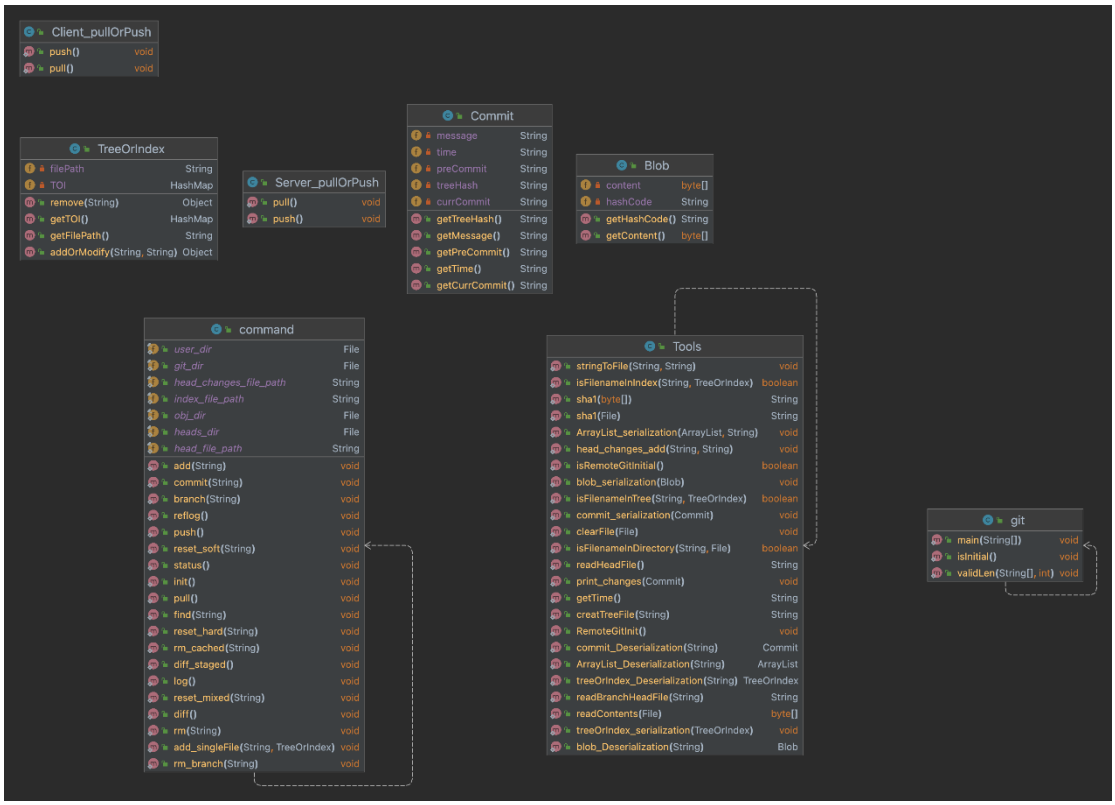
```
/ *
    *    .git
    *        |--objects
    *        |        |--commit /blob /tree
    *        |--refs
    *        |      |--heads
    *        |--head
    *        |--index
    *        |--head_changes
    * /
```

The objects folder is for commit, blob, and tree files. The head file and index file are as described in the requirements document.

There is also a level of heads folders in refs, where each file is named after the branch and contains the hash of the latest commit for that branch. Therefore, the actual purpose of this folder is to hold the table head Pointers for each branch.

The head_changes.txt file is a special file for implementing the reflog command to store the head changes. The contents of head_changes.txt are written as a serialization of an ArrayList that stores the hash of each head change.

2. Class architecture



(1) Blob class

A、 Attributes: byte[] content, String hashCode. Where content is the file content of the corresponding workspace file and hashCode is the hash value of the file content of the corresponding workspace.

B、 Constructor: public Blob(String fileName), role: generate Blob object

C、 Getters: Two automatically generated getter functions: String getHashCode() and byte[] getContent(). What it does: Get the attribute information.

(2) TreeOrIndex class

Because the contents of tree and index are exactly the same for a single-level folder, this project combines them into the same class.

A、 Properties: HashMap TOI and String filePath. Where HashMap TOI (tree or index abbreviation), key is used to store the filename, value is used to store the hash value generated by the corresponding file; filePath is used to store tree or index file path, although tree and index as the same class, but their storage path is different, tree is stored in the objects folder, index is stored in the.git folder, so the path difference as a property, easy to distinguish.

B、 Constructor: public TreeOrIndex(HashMap TOI,String filePath), function: generate index or tree object.

C、 Getters: Two automatically generated getters: String getFilePath() and HashMap getTOI(). What it does: Get property information.

D、 addOrModify(String filename, String hash) function: There are methods to add/modify the index object. The specific implementation operation is to call the put() function of the hashmap.

E、 The remove(String fileName) function does (the title requires) : There is a method to remove in the index object. The specific implementation of the operation is to call the hashmap remove() function.

Commit the class

A、 Properties: String preCommit, String message, String treeHash, String time, String currCommit. Where preCommit is the last commit id, message is the messag entered by the user when committing, treeHash is the id of the tree object generated by this commit, time is the commit time, and currCommit is the commit id of this commit.

B、 Constructor: public Commit(String message, String treeHash), function: generate commit object. In the function, currCommit is the hash value generated by concatenating the string of message+treeHash. Because the time is always changing, the content of the hash value does not include time; If two commits have the same version but different precommits, they should have the same hash value. Therefore, you shouldn't include time and preCommit when calculating the commit id for this commit.

C、 C, getters: Five automatically generated getters: String getPreCommit(), String getMessage(), String getTreeHash(), String getTime(), and String getCurrCommit(). What it does: Gets property information.

The Tools class

There are no attributes, and this class is designed to store highly reusable functions to avoid code redundancy.

A、String sha1(byte[] content) : This takes a byte array as input

B、String sha1(File file) : This is an override of the sha1 method with an input parameter of type File

C、byte[] readContents(File file) : Takes a text file and returns the contents of the file

D、void stringToFile(String s, String filePath) : Write the string into the file, passing in a string and the file path.

E、String getTime() : Gets the current time

F、boolean isFilenameInDirectory(String fileName, File dir) : Determines if the filename is in a folder

G、boolean isFilenameInIndex(String fileName, TreeOrIndex idx) : Determines if a filename exists in the index staging area

H、boolean isFilenameInTree(String fileName, TreeOrIndex tree) : Determines if a filename exists in the tree

I、void blob_serialization(Blob blob) : Serialization of a blob object

J、Blob blob_Deserialization(String filePath) : deserialization of a blob object

K、void commit_serialization(Commit commit) : commit object serialization

L、Commit commit_Deserialization(String filePath) : commit object deserialization

M、void treeOrIndex_serialization(TreeOrIndex treeOrIndex) : Serialize the TreeOrIndex object

N、TreeOrIndex treeOrIndex_Deserialization(String filePath) : Deserialization of the TreeOrIndex object

O、void ArrayList_serialization(ArrayList list,String filePath) : Serialization of an ArrayList

P、ArrayList ArrayList_Deserialization(String filePath) : deserialization of an ArrayList

Q、void head_changes_add(String preHead,String afterHead) : Add a new head change record in head_changes.txt. The first argument is the head value before the change, and the second argument is the head value after the change. Head_changes_txt is written as a serialized ArrayList, so adding a head record simply requires deserializing the head_changes file to an ArrayList with the hash of each head change. We then use list.add(0, afterHead) to add the new head value as the first element in the ArrayList, and then serialize the ArrayList into head_changes.txt.

R、String creatTreeFile(String index_file_path) : Generates a tree file. This function corresponds to the documentation: the tree object is the same as the index object, the tree object is generated from all the entries in the index file, and then the tree object is serialized into the objects folder. The idea of function construction: when git commit, the tree file is generated according to the index file. The content of the two files is the same, but the path and filename are different. The index.txt file is under git_dir, the tree file is under obj_dir, and the file name is the hash value of the file content. Therefore, this function is essentially a file copy, and we need to additionally find the hash value of the file content as the name of the new file. The copy is processed using an object stream so that the tree file can be converted into a tree object later.

S、void print_changes(Commit cmt) : (document requirement/purpose) Prints the changes(additions, deletions, changes) that have been made to the file since the last commit. Idea of function construction: The commit object passed in is the commit object of this commit, and the filename of the last commit file is found through the preCommit attribute of the commit object. After finding the last commit file, it is deserialized to

generate a commit object precmt, which represents the last commit object. Look for the treeHash attribute in precmt and cmt, find the root tree file of the two commits, deserialize the root tree file to get the tree object, in which this_time_tree and last_time_tree are compared, and there are the following four cases:

Case 1: If the fileName in this_time_tree can be found in last_time_tree, but the hash value is different, then this file is a modified file, output prompt.

Case 2: If the fileName in this_time_tree can not be found in last_time_tree, it means this file is a new file, output a message.

Case 3: If the fileName in this_time_tree can be found in last_time_tree and the hash value is the same, then this file is also committed last time, no change, no prompt message is output.

Case 4: If the fileName in last_time_tree cannot be found in this_time_tree, then the file is deleted and output a message.

T、String readHeadFile() : Reads the hashcode in the head file

U、String readBranchHeadFile(String branchPath) : Reads hashcode from the branchHead file in the heads folder

V、void clearFile(File dir) : Clears the files under the folder and the directory remains

W、boolean isRemoteGitInitial() : Determines if the remote repository exists

X、void RemoteGitInit() : This initializes the remote repository, because the remote repository is used to store the history of commits and does not need a workspace for users to edit, so there is no index file in the.git folder. The rest of the directory structure is the same as the local repository.

(5) git classes

The git class exists with three functions: void main(String[] args), void validLen(String[] args, int n), and void isInitial(). No properties exist.

A、Void validLen (String [] args, int n) : whether the input command length for the effective length, this function is used to the length of the illegally in user input command output, the use of the function code can enhance robustness.

B、void isInitial() : Determine whether the repository is initialized. This function is called to determine whether the repository is initialized when the user enters a command other than init. If the repository is not initialized, this function will output a printing prompt message. Similarly, the use of this function can enhance the robustness of the code.

C、void main(String[] args) : The main program for the entire project. String[] args is the command you typed. The body of the main function is a switch statement that sorts commands and executes different command functions. The command functions are all placed in the Command class.

In particular, the possibility of invalid commands entered by users should be taken into account in mian function. Two functions void validLen(String[] args, int n) and void isInitial() were called many times in mian function.

(6) command class

All properties of this class are File type directories or String type file absolute path, all methods are for the implementation of git commands. This part of all the implementation of git command methods briefly introduce the role, the specific implementation principle in the third part of the principle of command implementation and robustness.

A. Attribute File user_dir, is the directory of the user workspace, the code implementation of the workspace path is defined as the absolute path of the desktop.
   The File git_dir property is the local repository directory.
   The property File obj_dir, is the objects directory.
   The property String head_file_path, is the absolute path to the head file.
   The property String index_file_path is the absolute path to the index file.
   The File heads_dir property is the heads directory, which is needed for branch management to hold the branch Pointers. The String head_changes_file_path is the absolute path to the head_changes file, which is used to store each change of head, which is used to implement the reflog command.
B. void init() : Repository initialization. If the local repository already exists, you will be prompted to print a message, if not, you will create a git repository.
C. void add_singleFile(String fileName, TreeOrIndex idx) : Add or modify a file named fileName from the workspace to the staging area, and create the corresponding blob object to serialize into the objects folder. If the file exists only in staging but not in the workspace, remove the entry from staging.
D. void add(String s) : If s is ". ", then perform an add_singleFile operation on all files in the workspace. If s is a filename, then add_singleFile is applied to the file in the workspace.
E. void commit(String message) : Implements the specific requirements in all commits in Tencent docs. This project additionally implements the reflog command, so the method also calls the Tools.head_changes_add(head, cmt.getCurrCommit()) method, so that when the user commits, This will add the new head to head_changes.txt.
F. void rm_cached(String fileName) : This removes only the corresponding entry in index
G. void rm(String fileName) : Deletes the file named fileName from the workspace and the corresponding entry from the index object.
H. void log() : Reads the most recent commit id from head.txt and goes through the list, printing log information for each commit.
I. void reset_soft(String commitID) : Changes the hash in head.txt to the given commitID.
J. void reset_mixed(String commitID) : Resets index.txt on top of soft.
K. void reset_hard(String commitID), based on mixed, resets the workspace so that the contents of the workspace and staging are the same.
L. void pull() : Transfers the entire content from the remote repository to the local repository.
M. void push() : Upload all content from the local repository to the remote repository.
N. void status() : This prints the status information. There are three statuses: Changes not staged for commit, Untracked files, and Changes to be committed.
O. void diff() : The diff command compares changes made to files that were not staged for commit and files that were last added.
P. void diff_staged() : The diff_staged command is used to compare the specific changes between the staging area and the last commit.
Q. void branch(String branchName) : Adds a Branch by adding a new file called branchname to the heads folder with the current commitID content. This does not change the HEAD pointing, it simply adds a Branch.
R. void rm_branch(String branchName) : Removes a Branch that cannot be the Branch pointed to by the current HEAD.

S. void find(String targetMessage) : Prints the ids of all commits that are the same as the input message, one for each line if there are multiple results.

T. void reflog() : The log command can only output information about linked lists where the head pointer is the list head, while the reflog command outputs information about every change in head.

Server_pullOrPush class

There are no attributes in this class, only pull and push methods.

A、 void push() : Indicates the specific operation of the Server side (remote repository side, which is also the receiver of the push command) when the push command is executed.

B、 void pull() : Indicates the specific operation of the Server side (remote repository side, which is also the sending side of the pull command) when the pull command is executed.

The Client_pullOrPush class

Same logic as Server_pullOrPush class, this class has no attributes, only pull and push methods.

A、 void push() : When the push command is executed, the specific operation of the Client side (the local repository side, which is also the sending side of the push command).

B、 void pull() : Indicates the specific operation of the Client side (local repository side, which is also the receiving side of the pull command) when the pull command is executed.

II. Project highlights

Complete the reflog, status, diff, find commands in the optional command requirements and the branch and rm --branch commands in the branch management. This part is a brief introduction to the role of all the highlight commands, and the specific implementation principle is introduced in the third part of command implementation principle and robustness.

1、 reflog command

Command input form: git reflog

The log command can only output information about the linked list whose head pointer is the list head, and the reflog command can output information about every change in head.

2、 The status command

Command input form: git status

Command implementation requirements: Print the status of each file in the workspace and staging area. The status is divided into 3 types: Changes not staged for commit, Untracked files, Changes to be committed.

3、 diff command

Command type: git diff or git diff --staged

The diff command is used to compare the modified contents of the unadded file and the last added file. The diff_staged command is used to compare the specific changes in the staging area and the last commit file.

4、 The find command

git find [commitmessage]

Command implementation requirements: Print the ID of all commits that are the same as typing message, one for each line if there are multiple results.

5、 branch command

Command input form: git branch [branchName]

Command implementation requirements: Add a Branch, that is, add a new file named branchname in the heads folder with the current commitID content. This operation does not change the HEAD pointing, but simply adds a Branch.

6、 rm --branch command

Command input form: git rm --branch [branchname]

Command implementation requirements: Delete a Branch, this Branch cannot be the Branch pointed to by the current HEAD.

III. Command implementation principle and robustness

There are 18 git commands implemented in this project, which are: init, add, commit, rm --cached, rm, log, reset --soft, reset --mixed, reset --hard, pull, push, status, diff, diff --staged, branch, rm --branch, find, reflog.

1、 init command

The code is as follows. Use git_dir.exists() to see if your local repository already exists, and print a message if it does. If not, create a git directory, objects directory, heads directory, index file, head file, and head_changes file. If all are successfully created, a prompt message will be printed.

```
public static void init() {
    if (git_dir.exists()) {
        System.out.println("local repository already exists ");
        System.exit(0);
    }
    git_dir.mkdir();
    obj_dir.mkdir();
    heads_dir.mkdirs();
    Tools.stringToFile("", index_file_path);
    Tools.stringToFile("", head_file_path);
    Tools.ArrayList_serialization(new ArrayList(),head_changes_file_path);
    System.out.println("Local repository was initialized successfully.");
}
```

2, add command

To implement the add command, start by implementing the add _singleFile() method, which means to perform an add operation on a single file. Here's the code for add _singleFile() : First, use file.exists() to check if the file exists in the workspace, and then use Tools.isFilenameInIndex(fileName, idx) to check if the file exists in the staging area. If the file does not exist in either the workspace or the staging area, a prompt message is output. If it is present in the staging area and not in the workspace, remove the entry from the index object using idx.remove(fileName). If it is in the workspace, the object is serialized into a blob file, and idx.addOrModify(fileName, blob.gethashCode ()) is used to add or modify the entry. Note that if

the contents of the add file have been added before, these actions will have no effect, because they will produce an identical blob file that overwrites the previous blob.

```
public static void add_singleFile(String fileName, TreeOrIndex idx) {
        File file = new File(user_dir.getAbsolutePath() + File.separator + fileName
);
        if (!file.exists()) {
            if (Tools.isFilenameInIndex(fileName, idx)) {
                idx.remove(fileName);
            } else {
                System.out.println("does not exist in the workspace and the staging
area the file, please add the files in the workspace.");
                System.exit(0);
            }
        } else {
            Blob blob = new Blob(fileName);
            Tools.blob_serialization(blob);
            idx.addOrModify(fileName, blob.getHashCode());
        }
    }
```

The add command is much easier to implement with the add_singleFile() method. Here's the code: First, we check to see if it's the first add, by using "".equals(indexContent) to see if the index file is empty. If it is empty, then idx (the index object) is newly created; otherwise, idx is the result of deserializing the index file. At this point, you have an index object idx in your program.

Then determine whether the string you entered is "." ", if it is, then the next requirement is "delete from index the records that exist only in staging but not in the workspace." The idea of this requirement is to iterate over the filenames in the staging area, determine if each fileName is in the workspace, and delete the corresponding entry if it is not. This part started with the enhancement of the for traversal whether delete hashmap index entries, using the map, the remove (), but it will throw an exception ConcurrentModificationException, access to information, Iterator, then iterator.remove (), without exception.

Then, implement the requirement "modify the file records in the staging area that exist in the workspace in the index, and add the file records that do not exist in the staging area to the index." The idea of this requirement is: traverse the files in the workspace, call the add_singleFile(filename, idx) method for each file, and output a prompt message after completion.

If the input string is not ". ", then executes add_singleFile(s, idx). add_singleFile() takes into account whether s is the filename in the workspace, a robustness consideration I won't cover here.

When all the entries that need to be added, modified or deleted in the index object are updated, the serialization function of the index object is executed, and

the index object is serialized and written to the index file. At this point, the add command is complete.

```
public static void add(String s) {
        File indexFile = new File(index_file_path);
        String indexContent = new String(Tools.readContents(indexFile));
        TreeOrIndex idx = null;
        if ("".equals(indexContent)) {
            idx = new TreeOrIndex(new HashMap(), index_file_path);
        } else {
            idx = Tools.treeOrIndex_Deserialization(index_file_path);
        }
        if (".".equals(s)) {
            Set keyset = idx.getTOI().keySet();
            Iterator it = keyset.iterator();
            while (it.hasNext()) {
                Object key = it.next();
                String lastFileName = key.toString();
                if (!Tools.isFilenameInDirectory(lastFileName, user_dir)) {
                    it.remove();
                }
            }

            File[] files = user_dir.listFiles();
            for (File f : files) {
                if (f.isFile()) {
                    String filename = f.getName();
                    add_singleFile(filename, idx);
                }
            }
            System.out.println("All files in the workspace have been added to staging.");
        } else {
            add_singleFile(s, idx);
            System.out.println("have the add file "+ s +" to the staging area ");
        }
        Tools.treeOrIndex_serialization(idx);
    }
```

3 commit the command

The code is as follows. First check whether the index file is empty, if it is empty, it means that no add has been done, output a prompt message and exit the program.

If the index file is not empty, then use Tools.creatTreeFile(index_file_path) to get the tree id of this commit and create a tree file in the objects folder. The specific construction idea of this function is explained in the Tools class of the class architecture in Part I, so I won't repeat it here. Then create

a commit object and use Tools.readHeadFile() to get the commit id of the last commit. Use head.equals(cmt.getCurrCommit()) to check if the last commit and message are the same, and if they are, print a Don't commit message and exit the program.

If it is not the same as the last commit, then serialize the commit object and write it to the commit file in the objects folder.

In addition, Tencent documentation requires "print the file changes (add, delete, modify) of this commit relative to the last commit", so use! cmt.getPreCommit().equals("") First determines whether it is the first commit, if it is the first commit, there is no need to print the file changes (robustness). If not, the Tools.print_changes(cmt) function is called to print the changes. How print_changes() is constructed is explained in Part I, in the Tools class.

To implement the reflog command, we need to add a new head to head_changes.txt for each change in head. Therefore, in the commit() function, you need to run Tools.head_changes_add(head, cmt.getCurrCommit()) to write the commit id of this commit to the head_changes file.

Finally, write the commit id of this commit to the head file and output a message saying "commit was successful".

```
public static void commit(String message) {
    if("".equals(new String(Tools.readContents(new File(index_file_path))))){
        System.out.println("There are no files currently in staging to execute the
commit command, please run add first.");
        System.exit(0);
    }
    String treeHash = Tools.creatTreeFile(index_file_path);
    Commit cmt = new Commit(message, treeHash);
    String head = Tools.readHeadFile();
    if (head.equals(cmt.getCurrCommit())) {
        System.out.println("Exactly the same file as the last commit and message.
Do not re-commit.");
        System.exit(0);
    }
    Tools.commit_serialization(cmt);

    if (!cmt.getPreCommit().equals("")) Tools.print_changes(cmt);

    // change the head, adding new head to head_changes. TXT
    Tools.head_changes_add(head, cmt.getCurrCommit());

    Tools.stringToFile(cmt.getCurrCommit(), head_file_path);
    System.out.println("commit succeeded ~");
}
```

rm --cached command

The code is as follows. The Tencent documentation asks to "only delete the corresponding entry in index". So first read the contents of the index file, determine whether it is empty, if it is empty, output a prompt message. If not, deserialize it to produce an index object. Iterate over the

HashMap in the index object, and if there is a corresponding entry in the HashMap, delete the entry, and output the deletion success message. If the traversal is completed and the fileName is not found in the HashMap, the output is a prompt message of "There is no file in the staging area, the deletion failed".

```
public static void rm_cached(String fileName) {
    File indexFile = new File(index_file_path);
    String indexContent = new String(Tools.readContents(indexFile));
    if ("".equals(indexContent)) {
        System.out.println("Scratch area empty, delete failed ");
         System.exit(0);
    }
    TreeOrIndex idx = Tools.treeOrIndex_Deserialization(index_file_path);
    Set keyset = idx.getTOI().keySet();
    for (Object key : keyset) {
        if (fileName.equals(key.toString())) {
            idx.remove(fileName);
            Tools.treeOrIndex_serialization(idx);
            System.out.println(fileName + " file deleted successfully in the
staging area ");
            return;
        }
    }
    //index 中不存在 fileName，给出提示信息
    System.out.println("There is no file with the fileName "+ fileName + "in index
staging area. Delete failed.");
}
```

5, rm command

The code is as follows. The Tencent documentation requires "delete the corresponding entry in the index object and delete the file in the workspace". Therefore, the deletion operation in index can directly refer to the rm_cashed function. To delete the file in the workspace, it is necessary to check whether the file exists in the workspace, and output a prompt message if it does not exist (robustness). If it exists, the file.delete() function is used to delete the file in the workspace, and the prompt message "The file was deleted successfully in the workspace" is output after the deletion is successful.

```
public static void rm(String fileName) {
    rm_cached(fileName);
    // Delete the file in the workspace
    File file = new File(user_dir.getAbsolutePath() + File.separator + fileName);
    if (file.exists()) {
        // Only implement a single folder, so no more checking if file is a
directory
        file.delete();
```

```
        System.out.println(fileName + " file in your workspace deleted
success.");
    } else {
        System.out.println("the workspace is no file called "+ fileName +" file.
Delete failed.");
    }
}
```

6, log command

The code is as follows. First, use `Tools.readHeadFile()` to read the commit id of the last commit from the head file.If the head file is empty, print a message saying "Never committed to local repository, can't log".

If the head file is not empty, find the commit file stored in the objects folder based on the commit id read from the head file, and then deserialize the commit file into a commit object. Print the commit id, message, and commit time of the commit object, and then set commitID = cmt.getPreCommit(). Do this repeatedly until you've printed the first commit, so the duration condition of the loop is! commitID.equals(""). When the list is traversed, the log message is printed.

```
public static void log() {
    String head = Tools.readHeadFile();
    if (head.trim().equals("")) {
        System.out.println("This hasn't been committed to the local repository,
can't log ");
        return;
    }

    String commitID = head;
    while (!commitID.equals("")) {
        Commit cmt = Tools.commit_Deserialization(obj_dir.getAbsolutePath() +
                File.separator + commitID + ".txt");
        System.out.println("commit id:" + cmt.getCurrCommit()
                + "\nmessage:" + cmt.getMessage() + "\ncommit time:" + cmt.getTime(
));
        System.out.println("================================");
        commitID = cmt.getPreCommit();
    }
}
```

The reset_soft command

The code is as follows. The documentation says "check if there is a commit object in the objects folder", so first check if there is a file with this Id in the objects folder, if not, then prompt "There is no file in the objects folder with this commitID name, please enter the correct commitID". If there is, then check if the file is a Commit file. If not, then "this file is a tree or blob file, please enter the correct commitID". (Robustness)

If you are sure that there is a commit file named this commit id in the objects folder, follow the documentation to "change the head file contents to the given commit id", you need to modify head_changes.txt and head.txt, So call the Tools.head_changes_add(head, commitID) function and the Tools.stringToFile(commitID, head_file_path) function. When the head file is modified, print out a message indicating that the change was successful.

```
public static void reset_soft(String commitID) {
    String fileName = commitID + ".txt";
    if (Tools.isFilenameInDirectory(fileName, obj_dir)) {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(obj_dir.getAbsolutePath
() + File.separator + fileName));
            Object o = ois.readObject();
            if (o instanceof Commit) {
                / / change the head, adding new head head_changes. TXT
                String head = Tools.readHeadFile();
                Tools.head_changes_add(head, commitID);

                // modify the HEAD file content for a given commit id
                Tools.stringToFile(commitID, head_file_path);
                System.out.println("HEAD file content was modified as
" + commitID);
            } else {
                System.out.println("this hash value is named the file tree or blob
files, please input the correct commitID ");
                System.exit(0);
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    } else {
        System.out.println("There is no file named commitID in the objects folder,
please enter a correct commitID ");
        System.exit(0);
    }
}
```

8, reset --mixed command

The code is as follows. The documentation asks to "reset the index file based on the commit id on a soft basis". So reset_soft(commitID) is called first to modify the head file. Whether or not the commit id is legal is already considered in the reset_soft command and will not be added here (robustness).

Then find the commit file and deserialize it to get the commit object. Through the cmt.getTreeHash() function in the commit object, you can get the tree id generated at the time of

commit, and find the tree file in the objects folder according to the tree id. Then deserialize the tree file to get the tree object. Because this project only implements a single layer folder, so the tree content and index content are the same, this project defines tree and index as the same class TreeOrIndex, so the tree object is the index object, and then serialize the index object into the index file. After resetting the index file, output the prompt message "index file has been reset".

```
public static void reset_mixed(String commitID) {
    reset_soft(commitID);
    String commitFilePath = obj_dir.getAbsolutePath() + File.separator + commitID +
 ".txt";
    Commit cmt = Tools.commit_Deserialization(commitFilePath);
    String treeFilePath = obj_dir.getAbsolutePath() + File.separator + cmt.getTreeH
ash() + ".txt";
    TreeOrIndex tree = Tools.treeOrIndex_Deserialization(treeFilePath);
    // single level folder with same tree and index contents
    TreeOrIndex idx = tree;
    Tools.treeOrIndex_serialization(idx);
    System.out.println("The index file has been reset.");
}
```

9, reset -hard command

The code is as follows. The documentation requires that "on the basis of mixed, the reset workspace is consistent with the staging area content". The code does this by first calling Tools.clearFile(user_dir) to clear the workspace and then generating files based on the entries in the staging area. Therefore, we deserialize the index file to get the index object, traverse the HashMap of the index object and find the hash value of each entry, which is the filename of the blob file, so we can find the corresponding blob file and deserialize to get the blob object. Then we write the contents of blob.getContent() to the workspace's file. When the traversal is complete, the message "Workspace reset successfully" will be printed.

```
public static void reset_hard(String commitID) {
    reset_mixed(commitID);
    Tools.clearFile(user_dir);
    TreeOrIndex idx = Tools.treeOrIndex_Deserialization(index_file_path);
    Set keyset = idx.getTOI().keySet();
    for (Object fileName : keyset) {
        String hashCode = idx.getTOI().get(fileName).toString();
        String blobFilePath = obj_dir.getAbsolutePath() + File.separator + hashCode
 + ".txt";
        Blob blob = Tools.blob_Deserialization(blobFilePath);
        String txtFilePath = user_dir.getAbsolutePath() + File.separator + fileName
.toString();
        Tools.stringToFile(new String(blob.getContent()), txtFilePath);
    }
    System.out.println("The workspace reset successfully.");
}
```

10, pull command

In the main function of the pull command, in addition to determining whether the local repository is initialized, it also determines whether the remote repository is initialized. If it is not initialized, it needs to output a prompt and exit the program. Make sure that both the local and remote repositories are initialized before executing command.pull(). The pull() function in the command class looks like this:

This project simulates the Server (remote repository) and Client (local repository) two programs to run, so in the pull command need to create two threads, two threads call the pull() function of the Server side and the pull() function of the Client side.

```
public static void pull() {
    Thread threadOne = new Thread(new Runnable() {
        public void run() {
            Server_pullOrPush.pull();
        }
    });

    Thread threadTwo = new Thread(new Runnable() {
        public void run() {
            Client_pullOrPush.pull();
        }
    });
    threadOne.start();
    threadTwo.start();
}
```

The pull command Client is the receiver and Server is the sender. Analyze the code logic of the sending side first. The pull() function on the Server side looks like this: This project directly takes the current running path as the remote repository path, and gets the remote repository path through System.getProperty("user.dir").

Then the server side opens, ip address is the local ip address, port number is 6666, and print the port number and ip address.

Then the server connection, if the connection is successful, the output prompt information.

Then you can start sending files from the remote repository to the Client. The numbers, objects, and strings being sent are all written to the transport stream as ObjectOutputStream.

First, we send the files in the objects folder. We loop through the objects folder and count the number of files in the objects folder. We need to send the count number of files to the client, so that the client knows how many files it has received and closes the connection. We then send each file in the objects folder in turn. We use ObjectInputStream to read an Object from the text in the remote repository and write it to the transport stream. We also need to pass the filename so that the local repository can name the file when it receives it.

After the objects folder is transferred, the head file is sent. The hash value in the head file is read and then written as a string to the transport stream.

Then send the heads folder. The logic is the same as sending the files in the objects folder. The number of files is sent first, followed by the contents.

Finally, we send the head_changes file, deserialize it into an ArrayList of each head value, and

write the ArrayList to the transport stream.

At this point, all files have been transferred, and the prompt message "Transfer completed, server terminates connection" is output. It should be noted that there is no index file set in the remote repository of this project, because when the reset_hard() function is finally executed in the Client side, it will reset the index file and the workspace according to the commit in the head, so there is no need to set the index file in the remote repository.

```
public static void pull() {
    // simply use the current run path as the remote repository path
    String remote_path = System.getProperty("user.dir");
    ServerSocket ss = null;
    Socket sk_server = null;
    ObjectInputStream ois = null;
    ObjectOutputStream oos = null;
    try {
        // Server side open
        ss = new ServerSocket(6666);
        System.out.println("server ip:
" + InetAddress.getLocalHost() + "server port: 6666");
        // server connection
        sk_server = ss.accept();
        System.out.println("Server client successfully connected!");

        // send a file server, remote file sent to the Client in the warehouse
        File remote_obj_dir = new File(remote_path+File.separator+".git"+File.separ
ator+"objects" );
        File[] files = remote_obj_dir.listFiles();
        int count = 0;
        for (File file : files) {
            count++;
        }
        oos = new ObjectOutputStream(sk_server.getOutputStream());
        oos.writeInt(count);// there are count files in the objects folder of the
remote repository, need to send the number of files to the client so it knows when
to close the connection

        // objects folder files
        for (File file : files) {
            String fileName = file.getName();
            ois = new ObjectInputStream(new FileInputStream(file));
            oos.writeObject(ois.readObject());
            oos.writeUTF(fileName);
        }

        // Send the head file
```

```
        File remoteHeadFile = new File(remote_path+File.separator+".git"+File.separ
ator+"head.txt" );
        String head = new String(Tools.readContents(remoteHeadFile));
        oos.writeUTF(head);

        // send heads folder
        File remote_heads_dir = new File(remote_path+File.separator+".git"+File.sep
arator+"refs"+File.separator+"heads");
        File[] heads_files = remote_heads_dir.listFiles();
        int k = 0;
        for (File file : heads_files) {
            k++;
        }
        oos.writeInt(k);

        for (File file : heads_files) {
            String fileName = file.getName();
            String branchHeadString = Tools.readBranchHeadFile(file.getAbsolutePath
());
            oos.writeUTF(branchHeadString);
            oos.writeUTF(fileName);
        }

        // transferring head_changes file
        String filePath = remote_path+File.separator+".git"+File.separator+"head_ch
anges.txt";
        oos.writeObject(Tools.ArrayList_Deserialization(filePath));

    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        try {
            if(ois != null){
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(oos != null){
                try {
                    oos.close();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
            }
            sk_server.close();
            ss.close();
            System.out.println("Server terminates connection after transfer.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The pull command Client is the receiving end. According to the information sent by the sender, it can be received in turn. The code for the Client side is as follows. Start by creating sk_client = new Socket(InetAddress.getLocalHost(), 6666) for the client connection. In a real situation, the ip address of the Client and Server should be different, but because this program is for testing purposes, so the ip address of the Client and Server are set to the local machine address. Then the output prints the IP address of the Client and the port number, and output the message of successful connection. Then receive the information sent by the sender in turn.

First, we receive an integer objectsFileCount, which is the number of files in the objects folder.

Second, the loop takes the object and fileName from the objectsFileCount transport and writes the object to the file named fileName. Finally, the message "objects folder transfer was successful" is printed.

We then receive the head file, read a string (hash) from the transport stream, and write the string to the head file in our local repository. Then the message "head file transferred successfully" is printed.

Then receive the heads folder, the code logic is the same as receiving the objects folder, no more details here.

Finally, it receives the head_changes file, reads an object from the transport stream, strong-converts the object to an ArrayList, and serializes the ArrayList to a local head_changes.txt. Then output the message "head_changes file received successfully".

After all files have been transferred, the message "Client terminated connection" will be printed.

Finally, we call the reset_hard() function from the Client's pull function, passing in the commit id from the head file, so we can reset the staging area and workspace. If the head file is empty, the remote repository is new and empty, and a message that it cannot be reset is printed (robustness). At this point, the pull() function on the Client side is complete.

```
public static void pull(){
        Socket sk_client = null;
        ObjectInputStream ois = null;
        ObjectOutputStream oos = null;
        try {
            // client link
            sk_client = new Socket(InetAddress.getLocalHost(), 6666);
            System.out.println("client ip:
" + InetAddress.getLocalHost() + "client port: 6666");
```

```
            System.out.println("connection successful ~");

            int objectsFileCount = 0;
            ois = new ObjectInputStream(sk_client.getInputStream());
            objectsFileCount = ois.readInt();

            // receiving objects the files in the folder
            for(int i = 1;i <= objectsFileCount;i++){
                Object o = ois.readObject();
                String fileName = ois.readUTF();
                File file = new File(command.obj_dir.getAbsolutePath()+File.separat
or+fileName);
                oos = new ObjectOutputStream(new FileOutputStream(file));
                oos.writeObject(o);
            }
            System.out.println("The objects folder was transferred
successfully.");

            // receive the head file
            Tools.stringToFile(ois.readUTF(),command.head_file_path);
            System.out.println("head file transfer was successful.");

            // receiving heads folder
            int headsFileCount = 0;
            headsFileCount = ois.readInt();
            for(int i = 1;i <= headsFileCount;i++){
                String branchHeadString = ois.readUTF();
                String fileName = ois.readUTF();
                String filePath = command.heads_dir.getAbsolutePath()+File.separato
r+fileName;
                Tools.stringToFile(branchHeadString,filePath);
            }
            System.out.println("heads folder transferred successfully.");

            // receive head_changes file
            Tools.ArrayList_serialization((ArrayList)ois.readObject(),command.head_
changes_file_path);
            System.out.println("head_changes file receive success.");
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }finally {
            try {
                if(ois !=null){
                    try {
```

```
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(oos != null){
                try {
                    oos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            sk_client.close();
            System.out.println("Client terminates connection.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
     if(Tools.readHeadFile().trim().equals("")){
     System.out.println("The remote repository is a new repository and the head
pointer is empty, so the reset operation cannot be performed ");
     System.exit(0);
     }
    command.reset_hard(Tools.readHeadFile());
}
```

11. push command

The push command is sent by the client and received by the server. The client-side code implementation logic of the push command is the same as the server-side logic of the pull command, and they are both the sender. The server-side logic of push command is the same as the client-side logic of pull command, and they are the same as the receiving side. Therefore, no further details will be added here.

12, the status command

The status command requires you to print the status of every file in your workspace and staging area.There are three types of status: Changes not staged for commit, Untracked files, and Changes to be committed.

Changes not staged for commit can be specifically divided into deleted and modified cases. deleted refers to the file records that exist in the staging area but not in the work area; The modified case is a file that exists in the staging area and also exists in the workspace, but has different hashes.

Untracked files are files that have been added to the workspace and haven't yet been added, meaning they don't currently have an entry in the index.

Changes to be committed are files that have been added but not committed, and can be modified, deleted, or new files. Specific code idea: get the commit object of the last commit by deserializing the head pointer, get the tree object of the last commit by deserializing the treeHash in the commit

object, traverse the entries in the tree object, you can get all the files of the last commit, and compare the entries in the index file. For the file entry, there are the following four cases: a. If it is in the index, but not in the last tree, it is new file. b. If it's in the index and it's in the previous tree, but the hash is different, it's a modified file. If it is in the index and in the previous tree and has a different hash, it is unmodified and does not need to output any information. d. If it is not in the index but it is in the last tree, it is deleted.

The specific code is as follows.

```
public static void status() {
    File indexFile = new File(index_file_path);
    String indexContent = new String(Tools.readContents(indexFile));
    TreeOrIndex idx = null;
    if ("".equals(indexContent)) {
        idx = new TreeOrIndex(new HashMap(), index_file_path);
    } else {
        idx = Tools.treeOrIndex_Deserialization(index_file_path);
    }
    System.out.println("===Changes not staged for commit ===");

    // use the Iterator Iterator traverses
    Set keyset1 = idx.getTOI().keySet();
    Iterator it1 = keyset1.iterator();
    while (it1.hasNext()) {
        Object key = it1.next();
        String fileName = key.toString();
        //1、Changes not staged for commit :deleted
        if (!Tools.isFilenameInDirectory(fileName, user_dir)) {
            System.out.println(fileName + "(deleted)");
        } else {
            //2、Changes not staged for commit :modified
            File f = new File(user_dir.getAbsolutePath() + File.separator + fileNam
e);
            if (!idx.getTOI().get(key).toString().equals(Tools.sha1(f))) {
                System.out.println(fileName + "(modified)");
            }
        }
    }

    //3、Untracked files:
    System.out.println("===Untracked files===");
    File[] files = user_dir.listFiles();
    for (File f : files) {
        if (f.isFile()) {
            if (!Tools.isFilenameInIndex(f.getName(), idx)) {
                System.out.println(f.getName());
```

```
            }
        }
    }

    //4、Changes to be committed
    System.out.println("===Changes to be committed===");
    String head = Tools.readHeadFile();
    Set keyset2 = idx.getTOI().keySet();
    Iterator it2 = keyset2.iterator();
    if (head.trim().equals("")) {
        // No commits yet, all files are new files
        while (it2.hasNext()) {
            Object key = it2.next();
            String fileName = key.toString();
            System.out.println(fileName + "(new file)");
        }
    } else {
        String commitFilePath = obj_dir.getAbsolutePath() + File.separator + head +
 ".txt";
        Commit cmt = Tools.commit_Deserialization(commitFilePath);
        String treeFilePath = obj_dir.getAbsolutePath() + File.separator + cmt.getT
reeHash() + ".txt";
        TreeOrIndex tree = Tools.treeOrIndex_Deserialization(treeFilePath);
        while (it2.hasNext()) {
            Object key = it2.next();
            String fileName = key.toString();
            if (!Tools.isFilenameInTree(fileName, tree)) {
                System.out.println(fileName + "(new file)");
            } else {
                String idx_hash = idx.getTOI().get(key).toString();
                String tree_hash = tree.getTOI().get(key).toString();
                if (!idx_hash.equals(tree_hash)) {
                    System.out.println(fileName + "(modified)");
                }
            }
        }

        Set keyset_ = tree.getTOI().keySet();
        for (Object key : keyset_) {
            if (!Tools.isFilenameInIndex(key.toString(), idx)) {
                System.out.println(key.toString() + "(deleted)");
            }
        }
    }
```

```
}
```

13, diff command

The diff command compares the changes made to the unadded file with the last added file.

Code implementation idea: first determine whether the index file is empty, if it is empty, it means that there is no file add to the staging area, output prompt information and exit the program (robustness). If the index file is not empty, then first traverse the entry in the index file to see whether the fileName in the index file has the same fileName in the workspace. If it is, then determine whether the hash value corresponding to the index entry is different from the hash value generated by the content of the workspace file. If it is different, it means that this file is a modified file. Then output the file contents of the last and this time to facilitate the user to compare their differences. The file content of this time is directly output, and the added file needs to find the corresponding blob file through the hash value in the index, and then deserialize it into a blob object. The last file content can be obtained through blob.getContent().

The output result is preceded by - sign, which represents the content before modification, and the output result is preceded by + sign, which represents the content after modification. The code is as follows.

```
public static void diff() {
    File indexFile = new File(index_file_path);
    String indexContent = new String(Tools.readContents(indexFile));
    TreeOrIndex idx = null;
    if ("".equals(indexContent)) {
        System.out.println("No file has been added to the staging area so we can't
see what's changed ");
         System.exit(0);
    } else {
        idx = Tools.treeOrIndex_Deserialization(index_file_path);
    }

    // use the Iterator Iterator traverses
    Set keyset = idx.getTOI().keySet();
    Iterator it = keyset.iterator();
    // Iterate over the entries in the index file
    while (it.hasNext()) {
        Object key = it.next();
        String fileName = key.toString();
        // See if the fileName in the index file has the same fileName in the
workspace
        if (Tools.isFilenameInDirectory(fileName, user_dir)) {
            File f = new File(user_dir.getAbsolutePath() + File.separator + fileNam
e);
            String idx_hash = idx.getTOI().get(key).toString();
            // index whether the hash corresponding to the entry is different from
the hash generated by the contents of the workspace file
```

```
            if (!idx_hash.equals(Tools.sha1(f))) {
                String blobFilePath = obj_dir.getAbsolutePath() + File.separator +
idx_hash + ".txt";
                Blob blob = Tools.blob_Deserialization(blobFilePath);
                System.out.println("-" + new String(blob.getContent()));
                System.out.println("+" + new String(Tools.readContents(f)));
            }
        }
    }
}
```

14, diff --staged command

The diff_staged command compares the specific changes made to the staging area and the last commit. How the code works: If it's in the index and in the previous tree, but the hash is different, it's a modified file. Find the blob file through the hash value, deserialize it into a blob object, and output the content before and after the modification through blob.getContent(). The blob file found with the hash value of the entry in the index is the modified one. The blob file found for the hash of the entry in tree is the one before the change.

The output is preceded by a - sign, which means the content before modification, and the output is preceded by a + sign, which means the content after modification. The code is as follows.

```
public static void diff_staged() {
    String head = Tools.readHeadFile();
    if (head.trim().equals("")) {
        // Nothing has been committed yet, all files are new files, output
message
        System.out.println("There are currently no historical commit versions.
There is no information about staging files compared to the last commit. Please
enter this command after you have committed.");
    } else {
        String commitFilePath = obj_dir.getAbsolutePath() + File.separator + head +
 ".txt";
        Commit cmt = Tools.commit_Deserialization(commitFilePath);
        String treeFilePath = obj_dir.getAbsolutePath() + File.separator + cmt.getT
reeHash() + ".txt";
        TreeOrIndex tree = Tools.treeOrIndex_Deserialization(treeFilePath);

        TreeOrIndex idx = Tools.treeOrIndex_Deserialization(index_file_path);
        ;
        // Iterate over using Iterator
        Set keyset = idx.getTOI().keySet();
        Iterator it = keyset.iterator();
        while (it.hasNext()) {
            Object key = it.next();
            String fileName = key.toString();
```

```
            if (Tools.isFilenameInTree(fileName, tree)) {
                String idx_hash = idx.getTOI().get(key).toString();
                String tree_hash = tree.getTOI().get(key).toString();
                if (!idx_hash.equals(tree_hash)) {
                    Blob preModifiedBlob = Tools.blob_Deserialization(obj_dir.getAb
solutePath() + File.separator + tree_hash + ".txt");
                    Blob afterModifiedBlob = Tools.blob_Deserialization(obj_dir.get
AbsolutePath() + File.separator + idx_hash + ".txt");
                    System.out.println("-
" + new String(preModifiedBlob.getContent()));
                    System.out.println("+" + new String(afterModifiedBlob.getConten
t()));
                }
            }
        }
    }
}
```

15, branch command

The branch command asks to add a Branch, which is a new file called branchname in the heads folder. The file content will be the current commitID. This does not change the HEAD pointing; it simply adds a Branch.

The input branchName also needs to determine if the branch name already exists, and if it does, output a prompt and exit the program (robustness).

```
public static void branch(String branchName) {
    String branchName_ = branchName+".txt";
    if(Tools.isFilenameInDirectory(branchName_,heads_dir)){
        System.out.println("This branch name already exists; you can change
it.");
        System.exit(0);
    }
    String newBranchHeadPath = heads_dir.getAbsolutePath() + File.separator + branc
hName_;
    String curCommitID = Tools.readHeadFile();
    Tools.stringToFile(curCommitID, newBranchHeadPath);
    System.out.println("Branch created successfully.");
}
```

16, rm -branch command

The rm_branch command requires the removal of a Branch that cannot be the Branch pointed to by the current HEAD. The idea is to delete the branchname file in the heads folder. There are two additional robustness issues to consider: a, consider that the entered branchName does not exist. b, the branch to be deleted is the current branch and the hint cannot be deleted.

The second robustness is more complex, the idea is as follows: take the head pointer in the

branchName file as the list head, go through the branch linked list, determine whether each commitID on the list is a commitID in the head file, if it is, it will be prompted that it cannot be deleted, and the program will exit. If the list is traversed and every commitID on the linked list is not the commitID in the head file, it means that the branch is not the current branch and can be deleted. The code is as follows:

```
public static void rm_branch(String branchName) {
     String branchName_ = branchName+".txt";
     if (!Tools.isFilenameInDirectory(branchName_, heads_dir)) {
         System.out.println("This branch does not exist, please enter the correct
branch name.");
     } else {
         String branchPath = heads_dir.getAbsolutePath() + File.separator + branchN
ame + ".txt";
         String branchHead = Tools.readBranchHeadFile(branchPath);
         String head = Tools.readHeadFile();
         String commitID = branchHead;
         while (!commitID.equals("")) {
             if (commitID.equals(head)) {
                 System.out.println(branchName + " branch is master, can't
delete.");
                 System.exit(0);
             }
             Commit cmt = Tools.commit_Deserialization(obj_dir.getAbsolutePath() +

                     File.separator + commitID + ".txt");
             commitID = cmt.getPreCommit();
         }

         File branchFile = new File(heads_dir.getAbsolutePath() + File.separator +
branchName + ".txt");
         branchFile.delete();
         System.out.println("The branch was deleted successfully.");
     }
 }
```

17, find command

The find command requires printing the ids of all commits that are the same as the one you entered message, one for each line if there are multiple results. Code implementation idea: Iterate through the files in the objects folder to determine if the object is a commit class. If it is, it is deserialized into a commit object, and the message information of this commit is obtained by cmt.getMessage(). If the message is the same as the targetMessage, the commitID is output. The code is as follows:

```
public static void find(String targetMessage) {
    File[] files = obj_dir.listFiles();
```

```
    for (File f : files) {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream(f));
            Object o = ois.readObject();
            if (o instanceof Commit) {
                Commit cmt = Tools.commit_Deserialization(f.getAbsolutePath());
                if (cmt.getMessage().equals(targetMessage)) {
                    System.out.println(cmt.getCurrCommit());
                }
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            if (ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

18, reflog command

The log command can only output information about linked lists whose head pointer is the list head, while the reflog command outputs information about every change in head. The code implementation idea is to use ArrayList, when the head is changed, the new hash value in the head file is written to the ArrayList, and then the ArrayList is serialized to the head_changes.txt file in the.gits directory.

head changes are divided into three cases: commit, reset, pull, and the pull command has reset hard command built in.

```
public static void reflog() {
    String head = Tools.readHeadFile();
    if (head.trim().equals("")) {
        System.out.println("No commits have been made so far. There is no version
history of commits.");
        System.exit(0);
    }
    // If head is not empty, then neither is head_changes
    String filePath = head_changes_file_path;
    ArrayList list = Tools.ArrayList_Deserialization(filePath);
    Iterator it = list.iterator();
```

```
    int k = 0;
    while (it.hasNext()) {
        String commitID = (String) it.next();
        String commitFilePath = obj_dir.getAbsolutePath() + File.separator + commit
ID + ".txt";
        Commit cmt = Tools.commit_Deserialization(commitFilePath);
        System.out.println(cmt.getCurrCommit() + "\tHEAD@{" + k + "}" + "\tmessage:
" + cmt.getMessage());
        k++;
    }
}
```