Project 4: SJTU!It′sMyGit!!!!!

> SJTU CS1958-01 2025Fall 第四次大作业
>
> **WARNING:本项目的README较长，请通览一遍后再开始动手写代码**
>
> **WARNING:没有必要也不可能第一次仔仔细细全部读完README，但请务必仔细看文中的要点部分，这会非常有用**
>
> **WARNING:本项目不是一个能够短期内肝出来的项目，请循序渐进完成，DDL赶不完者后果自负**

请先阅读Git傻瓜教程以及仓库里的GITLITE.pptx:)

项目骨架

```
gitlite/
├── include/                        # 头文件
|   ├── Utils.h
|   └── GitliteException.h
├── src/                            # 源代码
|   ├── Utils.cpp
|   ├── GitliteException.cpp
├── testing/                        # 测试文件
|   ├── samples/
|   ├── src/
|   ├── tester.py
|   ├── out.txt
|   └── MakeFile
└── main.cpp
```

在本次大作业中，我们只提供了`Utils`与`GitliteException`两个文件，它们中的实用方法可以来执行一些主要与文件系统相关的任务，以便可以专注于项目的逻辑而不是处理操作系统的特殊性。

同时，我们为您添加了一个`main.cpp`与两个建议类`Commit`和`Repository`以帮助入门；除此之外，你需要自己编写`CMakeList.txt`完成这个项目的编译与运行, 为了评测的正常运行，我们要求您编译后生成的可执行文件需要放在`gitlite/build`目录下.

除此之外，你可以编写其他任意类来支持您的项目，或者根据需要删除我们建议的类。但**请勿**使用任何外部代码，也**不要**使用 C++ 以外的任何编程语言。你可以使用所有你想要的 C++ 标准库，以及我们提供的实用程序，在此我们列举若干可能有用的库函数：

- 常见的`STL`库
- `unistd.h`中的`getcwd()`，用以获取当前目录下的路径
- `<ctime>`中的`std::time`,`std::time_t`,`std::gmtime`,`std::tm`用以获取当前年月日相关信息，以`Unix`纪元开始计算.
- `<string>`中的`size()`,`find()`，`string::npos()`，其可能对这个项目有所简化
- `<sstream>`, `<fstream>`中的`std::istringstream`,`std::ostringstream`，`std::ifstream`其可能对这个项目有所帮助

- 必要的输入输出函数
- 必要的类型转换函数，譬如`str()`,`stoul()`

我们希望你可以在完整阅读`README.md`后能够自行完成项目的剩余部分,这固然是一个极大的挑战，对于整个项目的设计、构建、实现、调试固然是一个难以攀登的高峰，但我们相信你能够战胜这一切，为此，我们提供了一些帮助，譬如：

> 1.设计环节请通览`README.md`，明确每个对象的作用和它们之间的常用交互方法，设计好了再开始实现；如果真的没有想法，可以阅读开头部分的`Git`的基本概念一节，这会对你如何设计有所帮助；
>
> 2.请运用OOP的思想试图对这个问题进行简化与封装，当你想要一个功能的时候，编写一个功能的函数，然后将文件中相关功能的部分解耦合，这样就在修改的时候就不需要分别进行更改；请封装好每一个你想要抽象化的类，上层对象之间的交互一定要避免使用底层操作。
>
> 3.在给变量起名时，请利用对象间的共性与差异性起名：比如起尽可能具体直观的变量名称；在给不同对象间通信的相同接口起相同名称；给具有共性的接口取一个具有泛化能力的名称等等……
>
> 4.仔细打注释，否则会出事：（
>
> 5.请仔细地阅读`Utils.h`与`Utils.cpp`中每一个函数，根据注释明确其用途，避免在操作系统层面的重复实现.
>
> 6.序列化时如果涉及`std::unordered_map`请仔细考虑！由于`std::unordered_map`的底层实现导致其在序列化时会出现问题，如果懒得解决的话建议采用`std::map`。

愿您一切顺利！如果遇到其他问题，我们将会统一收集并公开发布！

任务

总体要求

一、为了使`Gitlite`
工作，它需要一个地方来存储文件的旧副本和其他元数据。所有这些东西都必须存储在名为`.gitlite`的目录中 ，就像这些信息存储在`.git`中，与真正的 git 系统的目录中一样。（前面带有`.`的文件是隐藏文件。在大多数操作系统上，默认情况下你将无法看到它们。在
`Linux`上，该命令`ls -a`将显示它们。）如果`Gitlite`
系统在特定位置有一个`.gitlite`目录，则认为它已在特定位置“初始化”。大多数`Gitlite`
命令（`init`命令除外）需要在已初始化 Gitlet 系统的目录中使用时才有效。

二、某些命令会触发失败情况，并会指定错误消息。这些错误消息的具体格式将在规范的后续部分中指定。所有错误消息都以句点`.`结尾。如果程序遇到这些失败情况之一，它必须打印错误消息，并且不得更改任何其他内容。除了列出的失败情况之外，你无需处理任何其他错误情况。

比如在`main.cpp`中，你有一些故障情况需要处理，它们不适用于特定的命令。具体如下：

> 如果用户没有输入任何参数，则打印消息`Please enter a command.`并退出。
>
> 如果用户输入不存在的命令，则打印消息`No command with that name exists.`并退出。
>
> 如果用户输入的命令的操作数数量或格式错误，则打印消息`Incorrect operands.`并退出。
>
> 如果用户输入的命令(想一想，哪些命令?)要求位于初始化的`Gitlite`
> 工作目录（即包含`.gitlite`子目录的目录）中，但不在这样的目录中，则打印消息`Not in an initialized Gitlite directory.`

部分命令与真实 Git 的区别已列出。规范并未详尽列出与 Git 的所有区别，但列出了一些较大或可能造成混淆和误导的区别；比如
`Gitlite` 整个仓库的文件结构是扁平的（不管子目录），但 `Git` 是要考虑子目录的;

请勿打印任何除规范要求之外的内容。如果你打印任何超出要求的内容，我们的某些自动评分测试可能会崩溃。

三、为了使我们的评测机正常工作，我们要求你将编译生成的可执行文件放到 `gitlite/build` 文件夹下。如果可以的话，请通过修改环境变量 `PATH` 的方式使得我们能够在命令行像
`git` 一样运行 `gitlite add filename` 而不是 `./gitlite add filename`，一个可能的方法是：

```
mkdir -p ~/bin
cp /home/logic/gitlite/build/gitlite ~/bin/
ls -la ~/bin/
echo $PATH
echo 'export PATH=\"\$HOME/bin:\$PATH\"' >> ~/.bashrc
```

如果遇上了问题并在查询资料后无果，欢迎与助教联系。为行文方便，下文中的所有命令均以修改 `PATH` 后的命令为准.

Subtask1

在本子任务中，你需要完成 `init`, `add`, `commit` 和 `rm` 命令.

`init`(用法：`gitlite init`)

在该命令中，你需要在当前目录中创建一个新的 `gitlite` 版本控制系统。

具体地，该命令要求满足：

1. 生成的 `Gitlite` 版本管理系统自动从一次不包含任何文件但包含提交消息 `initial commit`（就是这样，没有标点符号）的提交开始。此初始提交的时间戳将为 `Thu Jan 01 00:00:00 1970 +0000`
   （这称为“Unix纪元”，内部用时间 0 表示。）
2. 生成的 `Gitlite` 版本管理系统有一个分支 `master`，最初指向此初始 `commit`. `master` 将成为当前分支。
3. 生成的 `Gitlite` 版本管理系统创建的所有存储库中的初始提交都具有完全相同的内容，因此所有存储库将自动共享此提交（它们都具有相同的 ID），并且所有存储库中的所有提交都将追溯到它。

除此之外，我们要求：

如果当前目录中已存在 `Gitlite`
版本控制系统，则不应使用新系统覆盖现有系统。请打印错误消息 `A Gitlite version-control system already exists in the current directory.` 后退出。

`add`(用法：`gitlite add [filename]`)

在该命令中，你需要将文件当前存在的副本添加到暂存区（参见命令 `commit`
）。暂存区应位于 `.gitlite` 中的某个位置。由于实现弱化，只要求单次只能上传一个文件。

具体地，该命令要求满足：

1. 如果添加已暂存的文件应当用新内容覆盖暂存区中的先前条目。
2. 如果文件的当前版本与当前提交中的版本相同，则不要将其暂存以待添加；如果此时（文件的当前版本与当前提交中的版本相同）文件已在暂存区，则将其从暂存区中移除（当文件被更改、添加，然后更改回其原始版本时可能会发生这种情况）。
3. 如果文件处于暂存状态且标记为待删除（参见命令`rm`），则将其待删除状态移除。

如果文件不存在，则打印错误消息：`File does not exist.`后退出，不进行任何更改。

`commit`（用法：`gitlite commit [Message]`）

在该命令中，你需要将暂存区中已跟踪的文件保存为一个新的提交（`commit`）。该提交被称为**跟踪**已保存的文件，它记录了这些文件的快照，并为它们创建一个新的历史版本，便于日后恢复。每个提交通常会附带一条提交信息（`message`），描述这次提交的内容。

具体地，该命令要求满足：

1. 默认情况下，每个`commit`的文件快照与其父提交的文件快照完全相同。当然，`commit`的时间和消息可能与父提交不同。
2. `commit`仅更新在提交时已添加到暂存区的文件的内容。更具体的，如果一个文件同时存在于暂存区和父级提交，那么`commit`
   将包含暂存区的文件版本而不是从其父级获取的版本；如果一个文件存在于暂存区但是未被其父级提交跟踪，那么`commit`
   应当将其添加到当前版本中并开始跟踪；如果一个文件在暂存区标记为待删除，那么`commit`应当取消对该文件的跟踪; 在完成对暂存区所有文件的操作后，
   `commit`命令应当清除暂存区。
3. `commit`命令应当忽略在文件暂存后进行的任何添加或删除操作。例如，如果你使用 `Linux` 中的 `rm` 命令（而不是 `gitlite`
   的同名命令）删除已跟踪的文件，则不会对`commit`产生任何影响，因为`commit`仍将包含该文件（现已删除）的快照。
4. `commit` 命令永远不会也不能添加、修改或删除工作目录中`.gitlite`目录以外的文件。
5. `commit` 命令在完成对于文件读写和暂存区的处理后，新的
   `commit`将作为新节点添加到`commit tree`中。最新的`commit`将成为“当前提交”，并且头指针现在指向它。之前的头提交是此提交的父提交。
6. 每一个`commit`命令都应当由其 SHA-1
   id 标识，其内容必须包含其文件的`blob`引用、父引用（可以不止一个）、日志消息和提交时间。

除此之外，我们

- 如果没有文件被暂存，则打印错误消息`No changes added to the commit.`后退出。
- 每个提交都必须有一条非空消息。如果没有，则打印错误消息`Please enter a commit message.`后退出。
- 如果工作目录中的跟踪文件丢失或更改，则不构成失败。只需完全忽略`.gitlite`目录之外的所有内容即可。

`rm`（用法：`gitlite rm [filename]`）

在该命令中，你需要对满足特定要求的文件进行特定的移除操作。

具体地，该命令要求满足：

1. 如果文件当前处于暂存区中且未被当前提交跟踪，则标记取消暂存该文件；
2. 如果文件在当前提交中被跟踪，则将其暂存且标记为待删除，下一次提交时将在版本历史中删除该文件；如果用户没有删除工作目录下的对应文件，则`rm`也需要将该文件从工作目录中移除。

除此之外，我们要求：

如果文件既没有被暂存也没有被`current commit`跟踪，则打印错误消息`No reason to remove the file.`后退出。

WARNING: 这个操作会直接对工作目录的文件进行修改或覆盖，在调试该操作时请谨慎进行。

Subtask2

在本子任务中，你需要完成`log`,`global-log`,`find`和`checkout`命令的最基础功能.

`log`（用法：`gitlite log`）

在该命令中，你需要按照要求打印`commit`的编辑日志。

具体地，该命令要求满足：

1. 该日志需要包含从当前头提交开始，沿着提交树向后显示每个提交的相关信息，直到初始提交，遵循第一个父提交链接，忽略在合并提交中找到的任何第二个父提交。（在常规`Git`中，你可以使用`git log --first-parent`获得此信息）。
2. 对于此日志中的每个节点，它应显示的信息是提交 ID、提交时间和提交消息。以下是它应遵循的确切格式的示例：

通用格式:

```
===
commit a0da1ea5a15ab613bf9961fd86f010cf74c7ee48
Date: Thu Nov 09 20:00:05 2025 +0800
A commit message.

===
commit 3e8bf1d794ca2e9ef8a4007275acf3751c7170ff
Date: Thu Nov 09 17:01:33 2025 +0800
Another commit message.

===
commit e881c9575d180a215d1a636545b8fd9abfb1d2bb
Date: Wed Jan 01 08:00:00 1970 +0800
initial commit
```

每个提交前都有一个`===`，提交后有一个空行。与真正的`Git`一样，每个条目都显示提交对象的唯一`SHA-1` ID。提交中显示的时间戳反映的是**当前时区**，而不是 UTC；因此，初始提交的时间戳不是 1970 年 1 月 1 日星期四 00:00:00，而是等效的北京时间。输出时请保证其按照`EEE MMM dd HH:mm:ss yyyy Z`的格式输出，项与项之间用空格隔开；

对于后续中的合并提交（具有两个父提交的提交），我们要求在第一个提交下方添加一行，如下所示：

```
===
commit 3e8bf1d794ca2e9ef8a4007275acf3751c7170ff
Merge: 4975af1 2c1ead1
Date: Sat Nov 11 12:30:00 2017 -0800
Merged development into master.
```

其中，`Merge:`
后面的两个十六进制数字依次由第一和第二个父提交 ID 的前七位组成。第一个父提交 ID 是你执行合并时所在的分支；第二个父提交 ID 是被合并的分支。这与常规 Git 中的操作相同。

`global-log`（用法：`gitlite global-log`）

在该命令中，你需要按照要求打印`commit`的编辑日志。

具体地，该命令要求满足：

1. 与`log`只显示`commit tree`的某个子树不同，`global-log`要求显示所有`commit`的信息，甚至包括`reset`到某个旧分支后的信息。`commit`的顺序无关紧要。

提示：`Utils.cpp`中有一个实用方法可以帮助你迭代目录中的文件。

`find`（用法：`gitlite find [commit message]`）

这是一个真实`git`中不存在的命令。在该命令中，你需要对满足特定要求的文件进行特定的查找操作。

具体地，该命令要求满足：

1. 打印所有包含指定`commit Message`的`commit ID`，每行一个。如果有多个这样的提交，则将`commit ID`打印在不同的行上，顺序无关紧要;
2. 如果你的`commit Message`很长，请形如`"Add a new file"`用引号将其框起来，就像commit下面的命令一样。

除此之外，我们要求：

如果如果不存在对应提交信息的提交，则打印错误消息`Found no commit with that message.`后退出.

提示：这个命令与`global-log`有何共通之处?

`checkout`（用法：`gitlite checkout ([commit ID]) -- [filename]`）

在该命令中，你需要用当前`commit`/指定`commit ID`的文件版本覆盖工作目录中的文件，具体来说，命令应当支持两种格式：

- `gitlite checkout -- [filename]`表示将文件在`HEAD`中提交的版本放入工作目录中，并覆盖工作目录中已存在的版本（如果存在）。新版本的文件不会被暂存；
- `gitlite checkout [commit ID] -- [filename]`表示获取指定 id 提交的文件版本，并将其放入工作目录，如果工作目录中已有文件版本，则覆盖之。新版本不会被暂存。

其中对于`commit ID`我们特别说明，因为 `commit id` 一般是一个长达 `40` 个字符的十六进制数字；所以我们需要你实现 **缩写**
功能，即用一个唯一的前缀来缩写提交。例如，可以缩写`a0da1ea5a15ab613bf9961fd86f010cf74c7ee48`为`a0da1e`来达到相同的效果。在（可能）没有其他对象具有以相同六位数字开头的
`SHA-1`标识符的情况下，你应该同样支持对缩写形式的支持。

除此之外，我们要求：

- 如果该文件在前一次提交中不存在，则打印错误消息`File does not exist in that commit.`后退出。不要修改当前工作目录。
- 如果不存在具有给定 `commit id`
  的提交，则打印错误信息`No commit with that id exists.`后退出；否则，如果文件在给定的提交中不存在，则打印与上一种相同的消息。不要修改当前工作目录。
- 如果缩写对应多个可能的 `commit id` ，返回列表中第一个找到的即可，无需报错；

WARNING: 这个操作会直接对工作目录的文件进行修改或覆盖，在调试该操作时请谨慎进行。

Subtask3

在本子任务中，你需要完成`status` 和 `checkout` 命令的进阶用法。

`status`（用法：`gitlite status`）

在该命令中，你需要实现通过`status`命令显示当前存在的分支，并用`*`标记当前分支;同时显示已暂存待添加或删除的文件。其格式如下:

```
=== Branches ===
*master
other-branch

=== Staged Files ===
wug.txt
wug2.txt

=== Removed Files ===
goodbye.txt

=== Modifications Not Staged For Commit ===
junk.txt (deleted)
wug3.txt (modified)
```

```
=== Untracked Files ===
random.stuff
```

最后两部分（未暂存的修改和未跟踪的文件）为扩展部分（见`Subtask 6`），你可以留空（只保留文件头）。

具体地，该命令要求满足格式上各部分之间有一个空行，整个状态后无空行结尾。条目应按字符串字典序列出（星号不计算）。

`checkout`（用法：`gitlite checkout [branchname]`）

在该命令中，你需要用指定`branch`的`HEAD`提交的文件覆盖工作目录中的文件，具体地，该命令要求满足：

1. 命令结束时，指定分支将被视为当前分支 (HEAD)。
2. 所有在当前分支中跟踪但在`checkout`分支中不存在的文件都将被删除。除非`checkout`分支是当前分支（请参阅下文的报错案例），否则暂存区将被清空。

除此之外，我们要求：

- 如果不存在同名分支，则打印`No such branch exists.`并退出；请勿更改当前工作目录。
- 如果该分支是当前分支，则打印`No need to checkout the current branch.`并退出；请勿更改当前工作目录。
- 如果当前分支中存在未跟踪的工作文件，且该文件将被`checkout`命令覆盖，则打印`There is an untracked file in the way; delete it, or add and commit it first.`并退出；请勿更改当前工作目录。

WARNING: 这个操作会直接对工作目录的文件进行修改或覆盖，在调试该操作时请谨慎进行。

Subtask4

在本子任务中，你需要完成`branch`，`rm-branch`和`reset`命令。

`branch`（用法：`gitlite branch [branchname]`）

在该命令中，你需要建一个指定名称的新分支，并将其指向当前的主提交。分支只不过是指向提交节点的引用名称（SHA-1 标识符）。此命令不会立即切换到新创建的分支（就像在真实的 Git 中一样）。在调用 branch 之前，你的代码应该在名为“master”的默认分支上运行。

除此之外，我们要求：

如果给定名称的分支已经存在，则打印错误消息`A branch with that name already exists.`后退出.

更具体地，我们希望`branch`功能与之前我们实现的功能能够相互耦合，请确保你的`branch`、`checkout`、`commit`行为与我们上面描述的一致。这是
`Gitlite`的核心功能，许多其他命令都依赖于它。

`rm-branch`（用法：`gitlite rm-branch [branchname]`）

在该命令中，你需要用`rm-branch`删除指定名称的分支。这仅表示删除与该分支关联的指针；并不意味着删除该分支下创建的所有提交，或诸如此类的操作。

除此之外，我们要求：

- 如果指定名称的分支不存在，则打印错误消息`A branch with that name does not exist.`后退出；
- 如果您尝试删除当前所在的分支，则打印错误消息`Cannot remove the current branch.`后退出

`reset`（用法：`gitlite reset [commit-id]`）

在该命令中，你需要用`reset`重置当前分支的`HEAD`指针。具体地，该命令要求满足检出（`checkout`）指定提交所跟踪的所有文件,移除该提交中不存在的跟踪文件,同时将当前分支的头移动到该提交节点。执行该命令后，暂存区将被清空。

除此之外，我们要求：

- 如果不存在具有给定 id 的提交，则打印错误信息`No commit with that id exists.`后退出；
- 如果工作文件在当前分支中未被跟踪并且将被重置覆盖，则打印错误信息`There is an untracked file in the way; delete it, or add and commit it first.`后退出。

HINT:该命令本质上是 checkout一个任意提交，它也会更改当前分支的头。实现该命令时，请学会复用前面的代码。

WARNING: 这个操作会直接对工作目录的文件进行修改或覆盖，在调试该操作时请谨慎进行。

Subtask5

在本子任务中，你需要完成`merge`命令。

`merge`（用法：`gitlite merge [branchname]`）

在该命令中，你需要将指定分支的文件合并到当前分支。

具体地，该命令要求：

如果分割点（`split point`,
`also common ancestor`）与给定分支是同一个提交，那么我们什么都不做；我们宣布合并完成，打印消息`Given branch is an ancestor of the current branch.`
后退出；如果分割点是当前分支，则效果是`checkout`到给定分支，打印消息`Current branch fast-forwarded.`后退出；否则，我们继续以下步骤：

1. 任何自分割点以来在给定分支中被修改过，但在当前分支中未被修改过的文件，都应更改为其在给定分支中的版本。然后，这些文件都将自动暂存。
2. 自分割点以来，在当前分支中已修改但在给定分支中未修改的任何文件都应保持原样。
3. 任何在当前分支和指定分支中以相同方式修改的文件在合并后保持不变。如果某个文件在当前分支和指定分支中都被删除，但工作目录中存在同名文件，则该文件将保持不变，并且在合并后仍处于既不被跟踪也不被暂存的状态。
4. 任何在分割点既不被跟踪也不被暂存且仅存在于当前分支中的文件都应保持原样。

5. 任何在分割点既不被跟踪也不被暂存并且仅存在于给定分支中的文件都应被`checkout`（即取到工作目录中）并暂存。
6. 任何存在于分割点、在当前分支中未修改、在给定分支中既不被跟踪也不被暂存的文件都应被删除并且不被跟踪。
7. 任何存在于分割点、在给定分支中未修改且在当前分支中既不被跟踪也不被暂存的文件都应保持既不被跟踪也不被暂存。
8. 任何在当前分支和指定分支中以不同方式修改的文件都属于冲突文件。“以不同方式修改”可能意味着两个文件的内容均已更改且彼此不同，或者其中一个文件的内容已更改而另一个文件被删除，或者该文件在分割点不存在，并且在指定分支和当前分支中的内容不同。在这种情况下，请将冲突文件的内容替换为（请在具体冲突的地方进行替换）

```
<<<<<<< HEAD
contents of file in current branch
=======
contents of file in given branch
>>>>>>>
```

（将`contents of file`替换为指定文件的内容）并暂存结果并将分支中已删除的文件视为空文件。请注意行终止符和行分隔符的运用，不注意这一点的人将会度过一个失败的人生:）

按照上述步骤更新完成并且分割点不是当前分支或指定分支时，合并操作就会自动提交，并记录日志信息`Merged [given branch name] into [current branch name]`（具体格式详见前文）；

如果合并操作遇到冲突，则会在终端（而不是日志）上打印信息`Encountered a merge conflict.`。

值得注意的是，每一个合并提交与其他提交不同：它们会将当前分支的头（称为“第一个父级”）和命令行中指定的待合并分支的头都都记录为父级。

除此之外，我们要求：

- 如果存在阶段性添加或删除操作，则打印错误消息`You have uncommitted changes.`并退出；
- 如果不存在具有给定名称的分支，则打印错误消息`A branch with that name does not exist.`并退出；
- 如果尝试将分支与其自身合并，则打印错误消息`Cannot merge a branch with itself.`后退出；
- 如果合并会因为提交本身没有更改而生成错误，则只让正常的提交错误消息通过。
- 如果合并会覆盖或删除当前提交中未跟踪的文件，则打印错误消息`There is an untracked file in the way; delete it, or add and commit it first.`并退出；

WARNING: 这个操作会直接对工作目录的文件进行修改或覆盖，在调试该操作时请谨慎进行。

Subtask6 (Bonus)

在本子任务中，你可以选择完善`status`功能，或者完成`remote`功能（包括`add-remote`,`rm-remote`,`push`,`fetch`,`pull`）命令, 或者两者兼有。

`status`（用法：`gitlite status`） {#bonus-status}

在该命令中，你需要实现通过`status`命令显示当前存在的分支，并用`*`标记当前分支;同时显示已暂存待添加或删除的文件。其格式如下:

```
=== Branches ===
*master
other-branch

=== Staged Files ===
wug.txt
wug2.txt

=== Removed Files ===
goodbye.txt

=== Modifications Not Staged For Commit ===
junk.txt (deleted)
wug3.txt (modified)

=== Untracked Files ===
random.stuff
```

最后两部分（未暂存的修改和未跟踪的文件）为扩展部分，您可以留空（只保留文件头）。

具体地，该命令要求满足：

1. 格式上各部分之间有一个空行，整个状态后无空行结尾。条目应按字符串字典序列出（星号不计算）。
2. 如果工作目录中的文件符合以下情况，则表示`Modifications Not Staged For Commit`：
    - 在当前提交中跟踪，在工作目录中更改，但未暂存；
    - 已保存在添加暂存区，但内容与工作目录不同；
    - 已保存在添加暂存区，但在工作目录中已删除；
    - 未在删除暂存区，但在当前提交中被跟踪并已从工作目录中删除。
3. 如果工作目录中的文件符合以下情况，则表示`Untracked Files`：
    - 工作目录中存在但既未暂存待添加也未跟踪的文件。
    - 包括已暂存待删除但随后在 `Gitlite` 不知情的情况下重新创建的文件。
    - 不包括可能已引入的任何子目录，因为 `Gitlite` 不会处理它们。

`Gitlite`中的远程仓库与本地仓库别无二致；由于调试能力限制，我们采用保存在本地的非当前仓库作为远程仓库，当我们提取其分支时，往往采用形如`origin/master`的形式，请注意在实现后文功能的同时，你需要对`branch`做出适当的处理.

`add-remote`（用法：`gitlite add-remote [remotename] [name of remote directory]/.gitlite`）

在该命令中，你需要用将指定的远程仓库地址保存在指定的远程名称下，例如比如`gitlite add-remote other ../testing/otherdir/.gitlite`。您的程序需要将所有正斜杠 `/` 转换为路径分隔符用以正确读取路径.

除此之外，我们要求：如果给定名称的远程已经存在，则打印错误消息：`A remote with that name already exists.`后退出。您不必检查用户名和服务器信息是否合法。

`rm-remote`（用法：`gitlite rm-remote [remotename]`）

在该命令中，你需要删除与给定远程名称关联的信息。此处的思路是，如果您想要更改已添加的远程，则必须先将其删除，然后重新添加。

除此之外，我们要求：如果给定名称的远程不存在，则打印错误消息：`A remote with that name does not exist.`后退出.

`push`（用法：`gitlite push [remotename] [remote branch name]`）

在该命令中，你需要尝试将当前分支的提交附加到指定远程分支的末尾。

具体地，该命令要求满足：

1. 该命令仅当远程分支的`HEAD`位于当前本地分支的历史记录中时才有效，这意味着本地分支包含远程分支未来的一些提交；
2. 在这种情况下，请将未来的提交附加到远程分支。然后，远程分支应重置使得因此其`HEAD`将与本地分支相同)；
3. 如果远程仓库中不存在该分支（即远程第一次接收该分支），则需要在远程新建该分支，并将其 HEAD 设置为当前本地分支的最新提交。

除此之外，我们要求：

- 如果远程分支的头不在当前本地头的历史记录中，则打印错误消息`Please pull down remote changes before pushing.`后退出；
- 如果远程`.gitlite`目录不存在，则打印`Remote directory not found.`后退出

`fetch`（用法：`gitlite fetch [remotename] [remote branch name]`）

在该命令中，你需要将远程 `Gitlite` 仓库中的提交复制到本地 `Gitlite` 仓库。

具体地，该命令要求满足：

将远程仓库中指定分支的所有提交和
`blob`（当前仓库中尚未存在的）复制到本地仓库中`.gitlet`中的分支`[remote name]/[remote branch name]`，并更改`[remote name]/[remote branch name]`使其指向头提交（从而将远程仓库中分支的内容复制到当前仓库）。如果此分支之前不存在，则会在本地仓库中创建。

除此之外，我们要求：

- 如果远程 `Gitlite` 仓库没有给定的分支名，则打印错误信息`That remote does not have that branch.`后退出；

- 如果远程`.gitlite`目录不存在，则打印错误信息`Remote directory not found.`后退出.

`pull`（用法：`gitlite pull [remotename] [remote branch name]`）

在该命令中，你需要通过`fetch`命令获取分支`[remote name]/[remote branch name]`，然后将该获取的分支合并到当前分支中。

除此之外，我们要求：

实现包含`fetch`和`merge`的报错命令.

WARNING: 这个操作会直接对工作目录的文件进行修改或覆盖，在调试该操作时请谨慎进行。

设计文档

你需要提交一个`README.md`来说明你对你项目的设计思路，其可以：

1. 类的定义，可能涉及到的实例变量和静态变量，简要描述变量在其类中的作用
2. 类的工作原理，包括但不限于类的边界情况或是针对复杂任务比如确定合并冲突时如何确定其算法等等
3. 类的持久化实现，你通过怎样的方式在`.gitlite`文件夹中记录程序状态或文件状态？该子文件夹中下有几个文件？您采用的序列化与反序列化方式?

你无需事无巨细，也无需逐行解析代码，上方的三类也只是建议而并非必需；这个`README.md`是希望您建立一个说明文档来帮助我们，也是帮助您了解您的设计思路

须知

截止时间

第十五周周日（12/26）24:00

编译与运行

本项目将在 wsl 中运行，请你在完成对应部分后打开 wsl依次执行以下命令：

```
cd gitlite          # 如果当前目录已在gitlite 下可跳过
rm -rf build/
mkdir build
cd build
cmake ..
make
```

然后打开命令行，用命令`gitlite`就可以开始手动对具体命令具体调试；

同时我们在下发的`testing`文件中提供了部分样例供各位调试，这也会作为最后的量化得分结果，具体操作是：

```
cd gitlite          # 如果当前目录已在gitlite 下可跳过
cd testing
python3 tester.py samples/*.in
```

注意`tester.py`默认调用的是`gitlite/build`文件夹下的可执行文件，你需要先编译后再调试；你也可以通过在命令行运行`python3 tester.py`来查看调用命令行与打开文件的具体方式；如果顺利的话，按照上文代码执行后在终端将会出现形如下方的界面,它会具体显示每个测试点的通过情况与报错信息，并会将错误的期望输出和实际输出保存同目录下的`out.txt`:

```
1-add-01: OK (1pts/1pts)
1-add-02: OK (1pts/1pts)
1-commit-01: OK (1pts/1pts)
1-commit-02: OK (1pts/1pts)
1-init: OK (2pts/2pts)
1-rm: OK (2pts/2pts)
1-robust: OK (2pts/2pts)
2-checkout-01: OK (2pts/2pts)
2-checkout-02: OK (2pts/2pts)
2-find-01: OK (2pts/2pts)
2-find-02: OK (1pts/1pts)
2-global-log-01: OK (1pts/1pts)
2-log: OK (2pts/2pts)
3-checkout-03: OK (2pts/2pts)
3-checkout-04: OK (2pts/2pts)
3-checkout-05: OK (1pts/1pts)
3-status-01: OK (2pts/2pts)
3-status-02: OK (2pts/2pts)
3-status-03: OK (2pts/2pts)
3-status-04: OK (2pts/2pts)
3-status-05: OK (2pts/2pts)
3-status-06: OK (2pts/2pts)
3-status-07: OK (1pts/1pts)
3-status: OK (2pts/2pts)
4-branch-01: OK (3pts/3pts)
4-branch-02: OK (2pts/2pts)
4-branch-03: OK (2pts/2pts)
4-find-03: OK (1pts/1pts)
4-global-log-02: OK (1pts/1pts)
4-reset-01: OK (3pts/3pts)
4-reset-02: OK (3pts/3pts)
4-rm-branch-01: OK (3pts/3pts)
4-rm-branch-02: OK (2pts/2pts)
5-merge-01: OK (3pts/3pts)
5-merge-02: OK (3pts/3pts)
5-merge-03: OK (3pts/3pts)
5-merge-04: OK (5pts/5pts)
5-merge-05: OK (3pts/3pts)
5-merge-06: OK (4pts/4pts)
5-merge-07: OK (4pts/4pts)
5-merge-08: OK (5pts/5pts)
6-remote-01: OK (5pts/5pts)
6-remote-02: OK (5pts/5pts)
6-remote-03: OK (5pts/5pts)
6-status-05: OK (5pts/5pts)

Ran 45 tests.
```

```
Total Score: 110 pts
All tests passed!
```

提交方式

请各位自行测试无误后将`git`仓库提交到 ACMOJ 对应渠道即可.

评分规则

对于项目中的代码部分，我们给出每个`Subtask`的分数占比，具体的分数由上方的评测机给定：

`Subtask 1`: 10 `pts`

`Subtask 2`: 10 `pts`

`Subtask 3`: 20 `pts`

`Subtask 4`: 20 `pts`

`Subtask 5`: 30 `pts`

`Subtask 6`: 20 `pts`

对于 Code Style 部分，我们会根据你的`README.md`以及代码布局与风格进行给分，包括但不限于适当的注释，合理的空行，优秀的板块设计等等，占 5 `pts`.优秀者可适度给6-7 `pts`.

对于 Code Review 部分，占 10 `pts`.

本项目的得分上限是 125 `pts`.

Acknowledgement

特别感谢`CS61B`对这个项目的启发，`UCBerkely`提供了极好的`skeleton`.

感谢`UCBerkely`为原始项目提供的极好的文档以及无数完成这个项目的网友为这个项目撰写的踩坑报告。

感谢2024级蒋欣桐在完成这个项目后提供的反馈以及为README做出的几十条修改，以及2024级ACM 丁宣铭为README提出的宝贵的修改意见。

如有问题请联系本项目的发布者`PhantomPhoenix`, 他的邮箱地址是: `logic_1729@sjtu.edu.cn`