基于Rust的Git系统框架实现

冯洋
南京大学计算机学院
fengyang@nju.edu.cn

助教：燕言言
yanyanyan@smail.nju.edu.cn

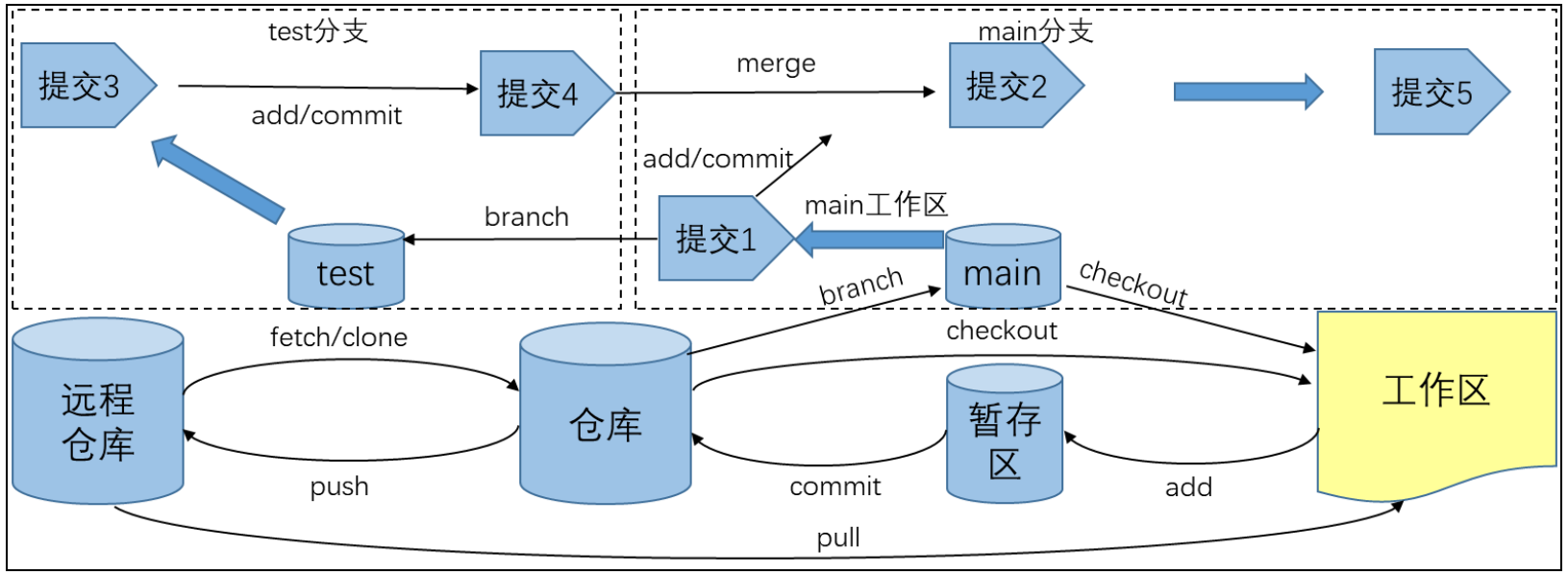
基于Rust的Git系统-要求

- 项目目标
  - 本项目旨在使用 Rust 作为主要开发语言，实现一个**简易的** Git 系统 [1-2]。通过完成该项目，同学们将深入理解 Git 的核心原理，并掌握 Rust 语言在实际项目中的应用。
- 项目内容
  - 基本功能：支持 Git 的基本操作，包括 git init（初始化仓库）、git add（添加文件到暂存区）、git rm（删除文件）、git commit（提交更改）等基础功能。
  - 分支管理：实现分支管理功能，如 git branch（创建/删除分支）、git checkout（切换分支）以及 git merge（合并分支）。
  - 进阶功能：鼓励有兴趣的同学进一步实现远程操作功能，例如 git fetch、git pull 和 git push，以模拟完整的 Git 工作流程。
- 组队要求
  - 组队：1-3人一组，组队后。填报队伍名称、队长信息、队员信息。队长和队员信息可填写姓名、学号等。

[1]. Rust实现Git系统. Online Available. [2025]. https://github.com/GitoxideLabs/gitoxide
[2]. Java实现Git系统. Online Available. [2025]. https://sp21.datastructur.es/materials/proj/proj2/proj2

基于Rust的Git系统-要求

- 预期效果
  - 我们的课程项目是实现基于Rust的Git系统，最终效果是支持常见的Git命令和分支操作，远程操作可选。



课程项目实现效果

基于Rust的Git系统-核心

- Git init初始化
  - ➢Git系统初始化后生成的.git目录是Git版本控制的核心，其内部结构和功能如下:
    - ◘objects目录：
      - ◆功能：存储Git的所有数据对象。包含info和pack子目录

        info子目录：附加信息。

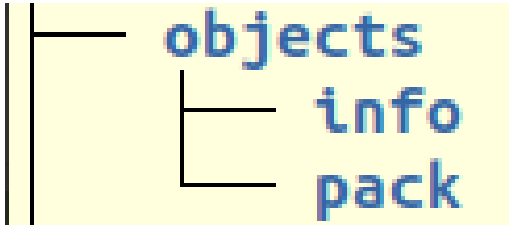
        pack子目录：打包的压缩对象。
      - ◆对象类型：

        blob：文件内容。

        tree：目录结构。

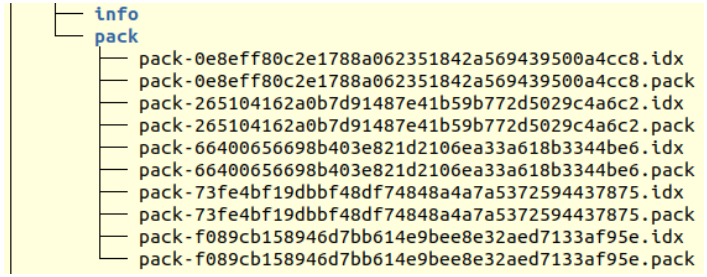
        commit：提交信息。

        tag：标签信息。



直接初始化效果

多次提交效果



基于Rust的Git系统-核心

- Git init初始化
  - ➢Git系统初始化后生成的.git目录是Git版本控制的核心，其内部结构和功能如下:
    - ◘objects目录：
      - ◆形如00-ff的两位十六进制子目录（共256个）及其内部文件共同构成了Git的对象存储系统，其设计原理和作用如下：

        哈希寻址系统：

        每个对象都通过SHA-1哈希值（40位十六进制）唯一标识。

        前2位作为目录名，后38位作为文件名（如/objects/00/140...）。

        这种设计将海量对象分散到256个子目录中，避免单个目录文件过多导致的性能问题。

多次提交效果

基于Rust的Git系统-核心

- Git init初始化
  - ➢Git系统初始化后生成的.git目录是Git版本控制的核心，其内部结构和功能如下:
    - ◘refs目录：
      - ◆功能：存储引用（指针）。包含heads、tags和remotes子目录。

        heads子目录：本地分支指针，如main。

        tags：标签分支。

        remotes：远程跟踪分支。

        tags作用：

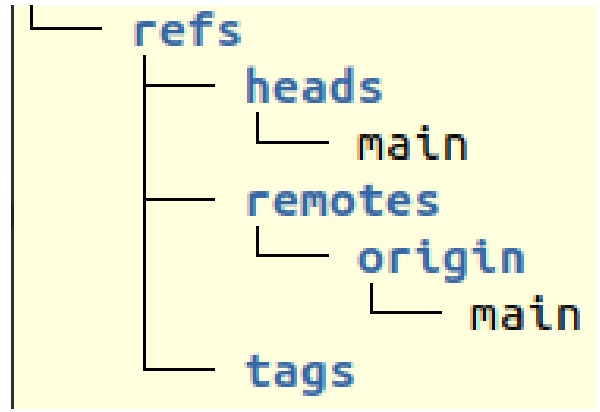
        存储指向特定提交的不可变指针（通常是发布版本，如 v1.0.0）。

        与分支（refs/heads/）不同，标签一旦创建通常不会移动。

        主要用于版本发布：git tag -a v2.1.0 -m “Production release“。

        快速回滚：git checkout v1.0。

        …



基于Rust的Git系统-核心

- Git init初始化
  - ➢Git系统初始化后生成的.git目录是Git版本控制的核心，其内部结构和功能如下:
    - ◘hooks及文件：
      - ◆hooks功能：存放客户端或服务端的Git钩子脚本。
        pre-commit：提交前触发。
        post-receive：推送后触发。
      - ◆HEAD文件功能：指向当前所在的分支或提交。
        通常是ref: refs/heads/[当前分支名]。
      - ◆config文件功能：存储当前仓库的配置信息。
        用户信息、远程仓库地址等。
      - ◆index文件：暂存区（stage）的二进制表示。
        记录已暂存但未提交的文件状态。
      - ◆desciption文件：供GitWeb等工具显示的仓库描述。
      - ◆COMMIT_EDITMSG：临时存储最后一次提交信息。
      - ◆FETCH_HEAD：记录最近一次fetch操作的引用。

```
├── COMMIT_EDITMSG
├── config
├── description
├── FETCH_HEAD
├── HEAD
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── fsmonitor-watchman.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-merge-commit.sample
│   ├── prepare-commit-msg.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   └── update.sample
└── index
```

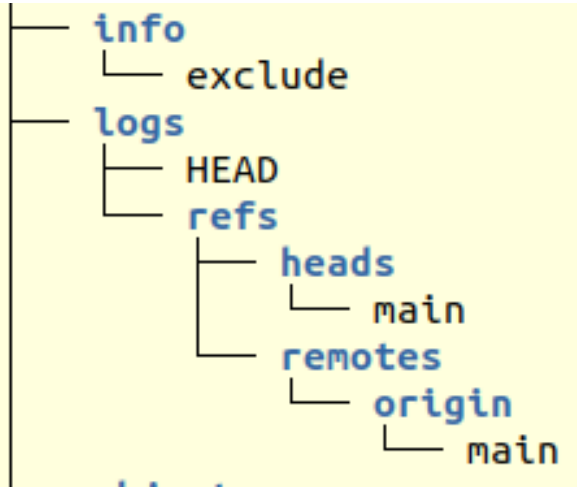
基于Rust的Git系统-核心

- Git init初始化
  - ➢Git系统初始化后生成的.git目录是Git版本控制的核心，其内部结构和功能如下:
    - ◘info目录：
      - ◆功能：存储仓库的元信息。包含exclude文件。

        exclude：本地忽略规则（类似.gitignore但不上传）。
    - ◘logs目录
      - ◆功能：记录所有引用变更历史包含refs/heads以及refs/remotes。

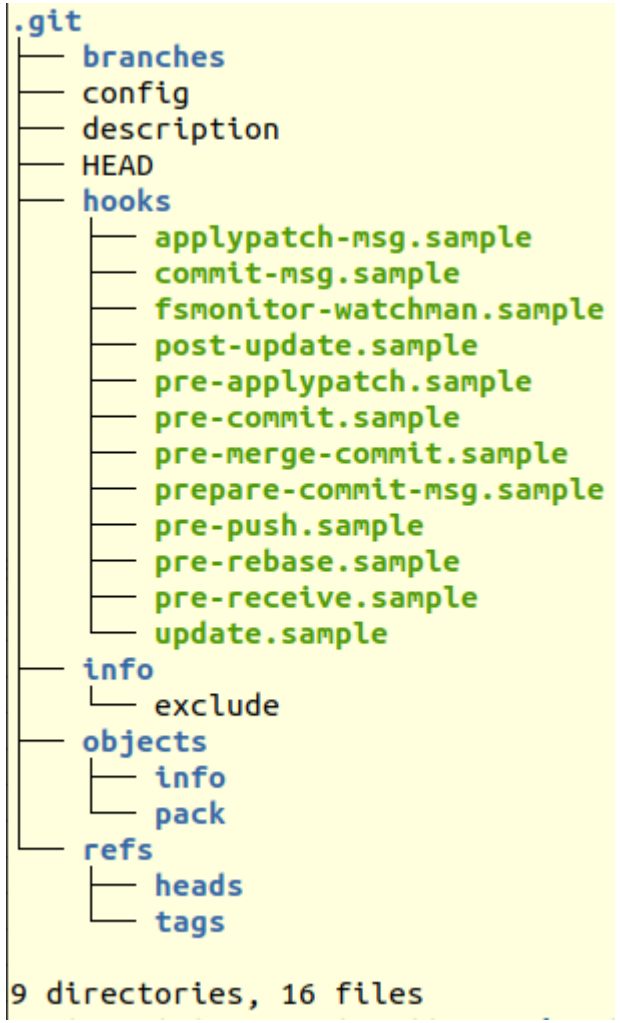
        refs/heads/：分支变更记录。

        refs/remotes：远程引用变更记录。



基于Rust的Git系统-思路

- 功能实现思路
  - ➢基本功能实现思路
    - git init：初始化仓库，创建.git目录及其相关子目录（如objects、refs等），初始化必要的配置文件（如config文件）。
    - git add：将指定文件或目录的内容添加到暂存区（索引文件 index），记录文件的当前状态（如文件名、内容的哈希值等）。
    - git rm：删除指定文件或目录，并从暂存区移除相关记录，同时更新索引文件。
    - git commit：将暂存区的内容提交到仓库，创建一个新的提交对象，记录提交信息（如提交者、提交时间、父提交等），更新分支引用指向新的提交。



Git 初始化

基于Rust的Git系统-思路

- 功能实现思路
  - 分支管理思路
    - git branch：创建新分支，通过在 refs/heads 目录下创建新的引用文件，指向当前分支的最新提交；删除分支，删除对应的引用文件。
    - git checkout：切换分支，更新工作区文件以匹配目标分支的最新提交，同时更新当前分支引用。
  - 分支合并思路
    - git merge：合并分支，将指定分支的更改合并到当前分支，处理冲突（如采用简单的冲突标记方式）。

基于Rust的Git系统-思路

- 功能实现思路
  - 进阶功能思路
    - git fetch：从远程仓库拉取数据，更新本地的远程分支引用，但不自动合并到当前分支。
    - git pull：拉取远程仓库的更改，并尝试自动合并到当前分支。
    - git push：拉取远程仓库的更改，并尝试自动合并到当前分支。

基于Rust的Git系统-结构

- 项目结构
  - ➢可能代码结构如右图所示。各模块功能介绍如下
  - ➢核心模块core介绍
    - repository.rs（仓库）：管理 .git 目录，提供初始化仓库、读取和更新配置文件等功能。
    - object.rs（对象）：表示 Git 对象（如提交、树、blob），提供对象的序列化和反序列化、存储和检索等功能。
    - index.rs（索引）：管理暂存区，记录工作区文件的状态，提供添加、删除文件和更新索引的功能。
    - reference.rs（引用）：表示分支、标签等引用，提供引用的创建、更新和解析等功能。
    - commit.rs（提交）：表示提交对象，包含提交信息、父提交、树对象等信息，提供提交的创建和解析等功能。
    - tree.rs（树）：表示目录结构，包含文件和子目录的引用，提供树的创建和解析等功能。
    - blob.rs（blob）：表示文件内容，提供 blob 创建和解析等功能。

```
.
├── Cargo.toml
└── src
    ├── cli
    │   ├── args.rs
    │   └── command.rs
    ├── commands
    │   ├── add.rs
    │   ├── branch.rs
    │   ├── checkout.rs
    │   ├── commit.rs
    │   ├── fetch.rs
    │   ├── init.rs
    │   ├── merge.rs
    │   ├── pull.rs
    │   ├── push.rs
    │   └── rm.rs
    ├── core
    │   ├── blob.rs
    │   ├── commit.rs
    │   ├── index.rs
    │   ├── object.rs
    │   ├── reference.rs
    │   ├── repository.rs
    │   └── tree.rs
    ├── lib.rs
    ├── main.rs
    └── utils
        ├── error.rs
        ├── fs.rs
        └── hash.rs

5 directories, 25 files
```

项目结构

基于Rust的Git系统-结构

- 项目结构
  - ➢可能代码结构如右图所示。各模块功能介绍如下
  - ➢命令模块commands介绍
    - ☐init.rs：实现 git init 命令，初始化仓库。
    - ☐add.rs：实现 git add 命令，将文件添加到暂存区。
    - ☐rm.rs：实现 git rm 命令，删除文件并更新暂存区。
    - ☐commit.rs：实现 git commit 命令，提交更改。
    - ☐branch.rs：实现 git branch 命令，创建和删除分支。
    - ☐checkout：实现 git checkout 命令，切换分支。
    - ☐merge.rs：实现 git merge 命令，合并分支。
    - ☐fetch.rs：实现 git fetch 命令，拉取远程数据。
    - ☐pull.rs：实现 git pull 命令，拉取并合并远程更改。
    - ☐push.rs：实现 git push 命令，推送本地更改到远程仓库。

可选

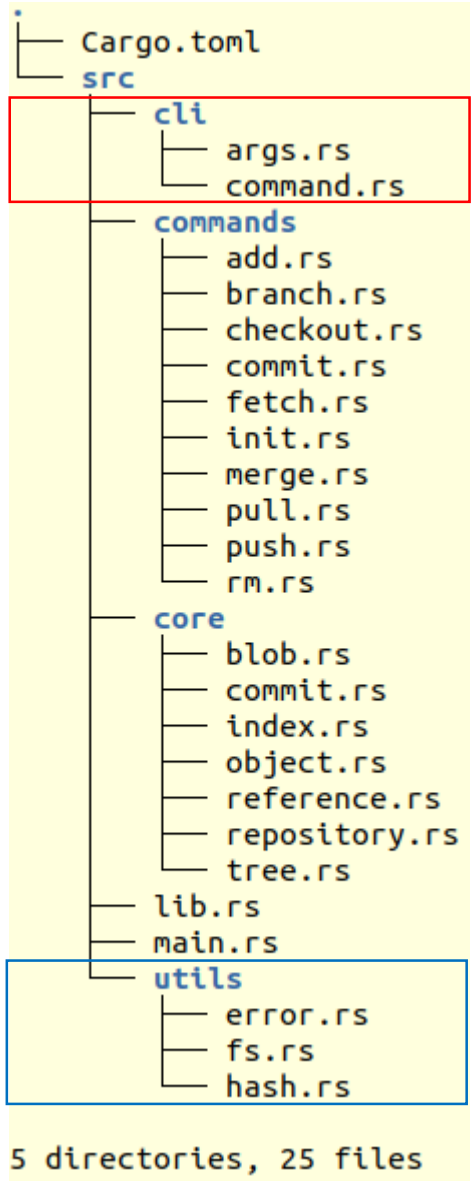
```
.
├── Cargo.toml
└── src
    ├── cli
    │   ├── args.rs
    │   └── command.rs
    ├── commands
    │   ├── add.rs
    │   ├── branch.rs
    │   ├── checkout.rs
    │   ├── commit.rs
    │   ├── fetch.rs
    │   ├── init.rs
    │   ├── merge.rs
    │   ├── pull.rs
    │   ├── push.rs
    │   └── rm.rs
    ├── core
    │   ├── blob.rs
    │   ├── commit.rs
    │   ├── index.rs
    │   ├── object.rs
    │   ├── reference.rs
    │   ├── repository.rs
    │   └── tree.rs
    ├── lib.rs
    ├── main.rs
    └── utils
        ├── error.rs
        ├── fs.rs
        └── hash.rs

5 directories, 25 files
```

项目结构

基于Rust的Git系统-结构

- 项目结构
  - 可能代码结构如右图所示。各模块功能介绍如下
  - 命令行接口cli介绍
    - args.rs：解析命令行参数，提取用户输入的命令和选项。
    - command.rs：根据解析的参数，调用对应的命令模块功能，执行相应的 Git 操作。
  - 工具模块utils介绍
    - fs.rs：提供文件系统操作的封装，如读写文件、创建目录等。
    - hash.rs：提供哈希计算功能，如计算文件内容的 SHA-1 哈希值。
    - error.rs：定义项目中使用的错误类型，提供错误处理和转换的功能。



项目结构

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢入口实现框架
    - ◘main模块
      - ◆程序的入口点，调用 cli::command::git_execute
      - ◆处理命令行参数并执行对应的命令。
      - ◆输入：
        命令行参数。
      - ◆输出：
        命令执行结果。

```rust
// src/main.rs
mod cli;

fn main() {
    cli::command::git_execute();
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢cli各模块实现框架
    - ◘args模块
      - ◆解析命令行参数。
      - ◆提取用户输入的命令和选项。
      - ◆输入：

        命令行参数。
      - ◆输出：

        解析后的命令和选项。

```rust
pub fn git_parse_args() -> ArgMatches {
    // 创建一个新的命令行应用
    创建命令行应用("rust-git")
        .设置版本("0.1.0")
        .设置作者("Your Name <your.email@example.com>")
        .设置描述("A simple Git implementation in Rust")

        // 定义子命令：初始化仓库
        .添加子命令(
            创建子命令("init")
                .设置描述("Initialize a new repository")
                .添加参数("path", "Path to the repository", 可选)
        )

        // 定义子命令：添加文件
        .添加子命令(
            创建子命令("add")
                .设置描述("Add file contents to the index")
                .添加参数("file", "File to add", 必选)
        )

        // 定义子命令：删除文件
        .添加子命令(
            创建子命令("rm")
                .设置描述("Remove files from the working tree and the index")
                .添加参数("file", "File to remove", 必选)
                .添加参数("force", "Force removal", 可选)
        )

        。。。
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢cli各模块实现框架
    - ◘args模块
      - ◆解析命令行参数。
      - ◆提取用户输入的命令和选项。
      - ◆输入：
        命令行参数。
      - ◆输出：
        解析后的命令和选项。

```rust
pub fn git_parse_args() -> ArgMatches {
    // 创建一个新的命令行应用
    创建命令行应用("rust-git")
        .设置版本("0.1.0")
        .设置作者("Your Name <your.email@example.com>")
        .设置描述("A simple Git implementation in Rust")
        。。。
        // 定义子命令：提交更改
        .添加子命令(
            创建子命令("commit")
                .设置描述("Record changes to the repository")
                .添加参数("message", "Commit message", 必选)
        )
        // 定义子命令：分支管理
        .添加子命令(
            创建子命令("branch")
                .设置描述("List, create, or delete branches")
                .添加参数("branch_name", "Branch name", 可选)
                .添加参数("delete", "Delete branch", 可选)
        )
        // 定义子命令：切换分支或恢复工作区文件
        .添加子命令(
            创建子命令("checkout")
                .设置描述("Switch branches or restore working tree files")
                .添加参数("target", "Branch or commit to checkout", 必选)
        )
        // 定义子命令：合并分支
        .添加子命令(
            创建子命令("merge")
                .设置描述("Join two or more development histories together")
                .添加参数("branch_name", "Branch to merge", 必选)
        )
        。。。
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢cli各模块实现框架
    - ◘args模块
      - ◆解析命令行参数。
      - ◆提取用户输入的命令和选项。
      - ◆输入：
        命令行参数。
      - ◆输出：
        解析后的命令和选项。

```rust
pub fn git_parse_args() -> ArgMatches {
    // 创建一个新的命令行应用
    创建命令行应用("rust-git")
        .设置版本("0.1.0")
        .设置作者("Your Name <your.email@example.com>")
        .设置描述("A simple Git implementation in Rust")
        。。。
        // 定义子命令：拉取数据
        .添加子命令(
            创建子命令("fetch")
                .设置描述("Download objects and refs from another repository")
                .添加参数("remote_url", "Remote repository URL", 必选)
        )

        // 定义子命令：拉取并合并
        .添加子命令(
            创建子命令("pull")
                .设置描述("Fetch from and integrate with another repository or a
local branch")
                .添加参数("remote_url", "Remote repository URL", 必选)
        )

        // 定义子命令：推送更改
        .添加子命令(
            创建子命令("push")
                .设置描述("Update remote refs along with associated objects")
                .添加参数("remote_url", "Remote repository URL", 必选)
        )

        // 解析命令行参数并返回匹配结果
        解析命令行参数()
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢cli各模块实现框架
    - ◘command模块
      - ◆根据解析的命令行参数，调用对应的命令模块功能。
      - ◆执行具体的 Git 操作。
      - ◆输入：
        解析后的命令行参数。
      - ◆输出：
        命令执行结果。

```rust
use crate::cli::args::git_parse_args;  // 引入命令行参数解析模块
use crate::commands::git_init::git_init; // 引入其他命令模块
use crate::commands::git_add::git_add;
use crate::commands::git_rm::git_remove;
use crate::commands::git_commit::git_commit;
use crate::commands::git_branch::git_branch;
use crate::commands::git_checkout::git_checkout;
use crate::commands::git_merge::git_merge;
use crate::commands::git_fetch::git_fetch;
use crate::commands::git_pull::git_pull;
use crate::commands::git_push::git_push;

pub fn git_execute() {
    // 解析命令行参数
    解析命令行参数();

    // 根据用户输入的子命令执行对应的操作
    匹配子命令 {
        // 执行 "init" 命令
        ("init", Some(sub_matches)) => {
            获取子命令参数("path", 默认值 = ".");
            执行 git_init(path);
        }
        // 执行 "add" 命令
        ("add", Some(sub_matches)) => {
            获取子命令参数("file", 必选);
            执行 git_add(".", file);
        }
        。。。
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢cli各模块实现框架
    - ◘command模块
      - ◆根据解析的命令行参数，调用对应的命令模块功能。
      - ◆执行具体的 Git 操作。
      - ◆输入：
        解析后的命令行参数。
      - ◆输出：
        命令执行结果。

```rust
pub fn git_execute() {
    // 解析命令行参数
    解析命令行参数();
    // 根据用户输入的子命令执行对应的操作
    匹配子命令 {
        。。。
        // 执行 "rm" 命令
        ("rm", Some(sub_matches)) => {
            获取子命令参数("file", 必选);
            获取子命令参数("force", 可选);
            执行 git_remove(".", file, force);
        }
        // 执行 "commit" 命令
        ("commit", Some(sub_matches)) => {
            获取子命令参数("message", 必选);
            执行 git_commit(".", message);
        }
        // 执行 "branch" 命令
        ("branch", Some(sub_matches)) => {
            获取子命令参数("branch_name", 默认值 = "master");
            获取子命令参数("delete", 可选);
            执行 git_branch(".", branch_name, delete);
        }
        // 执行 "checkout" 命令
        ("checkout", Some(sub_matches)) => {
            获取子命令参数("target", 必选);
            执行 git_checkout(".", target);
        }
        // 执行 "merge" 命令
        ("merge", Some(sub_matches)) => {
            获取子命令参数("branch_name", 必选);
            执行 git_merge(".", branch_name);
        }
        。。。
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢cli各模块实现框架
    - ◘command模块
      - ◆根据解析的命令行参数，调用对应的命令模块功能。
      - ◆执行具体的 Git 操作。
      - ◆输入：
        解析后的命令行参数。
      - ◆输出：
        命令执行结果。

```rust
pub fn git_execute() {
    // 解析命令行参数
    解析命令行参数();

    // 根据用户输入的子命令执行对应的操作
    匹配子命令 {
        。。。
        // 执行 "fetch" 命令
        ("fetch", Some(sub_matches)) => {
            获取子命令参数("remote_url", 必选);
            执行 git_fetch(".", remote_url);
        }
        // 执行 "pull" 命令
        ("pull", Some(sub_matches)) => {
            获取子命令参数("remote_url", 必选);
            执行 git_pull(".", remote_url);
        }
        // 执行 "push" 命令
        ("push", Some(sub_matches)) => {
            获取子命令参数("remote_url", 必选);
            执行 git_push(".", remote_url);
        }
        // 处理未知命令
        _ => {
            打印("Unknown command");
        }
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢core各模块实现框架
    - ◘repository模块
      - ◆管理.git目录的创建和初始化。
      - ◆提供仓库路径的管理。
      - ◆检查当前目录是否为Git仓库。
      - ◆输入：

        初始化路径，如当前目录“.”。

        操作类型（如初始化仓库）。
      - ◆输出：

        初始化成功的提示信息。

        仓库路径。

```rust
// Git 仓库核心模块
mod repository {
    // 仓库结构体
    pub struct Repository {
        path: String,  // 仓库路径
    }

    impl Repository {
        // 初始化新仓库
        pub fn init(path: &str) -> Self {
            // 创建 .git 目录结构
            create_dir(".git");
            create_dir(".git/objects");  // 对象存储
            create_dir(".git/refs");     // 引用存储
            create_file(".git/HEAD");   // 当前分支指针

            Repository { path }
        }

        // 验证Git仓库
        pub fn is_git_repo(path: &str) -> bool {
            check_dir_exists(".git")  // 检测.git目录
        }
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢core各模块实现框架
    - ◘object模块
      - ◆表示 Git 对象（提交、树、blob）。
      - ◆提供对象的序列化和反序列化。
      - ◆将对象存储到 .git/objects 目录。
      - ◆输入：
        - 对象类型（提交、树、blob）。
        - 对象数据（如提交信息、树结构、文件内容）。
      - ◆输出：
        - 对象的 SHA-1 哈希值。
        - 存储路径。

```rust
// 核心对象类型枚举
pub enum Object {
    Commit(String),  // 提交对象：存储提交信息
    Tree(String),    // 树对象：存储目录结构
    Blob(String),    // 数据对象：存储文件内容
}

impl Object {
    /// 对象存储算法
    pub fn save(&self, repo_path: &str) -> String {
        // 1. 序列化对象数据
        let raw_data = match self {
            Object::Commit(data) => format!("commit {}\0{}",
data.len(), data),
            Object::Tree(data)   => format!("tree {}\0{}",
data.len(), data),
            Object::Blob(data)   => format!("blob {}\0{}",
data.len(), data),
        };

        // 2. 计算SHA1哈希值
        let hash = sha1(raw_data);

        // 3. 存储到objects目录
        let dir = format!("{}/.git/objects/{}", repo_path,
&hash[0..2]);
        create_dir(dir);
        write_file(format!("{}/{}", dir, &hash[2..]),
raw_data);

        hash // 返回对象哈希
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢core各模块实现框架
    - ◘index模块
      - ◆管理暂存区（索引文件 index）。
      - ◆添加和删除文件到暂存区。
      - ◆读取和写入索引文件。
      - ◆输入：

        文件路径。

        操作类型（添加或删除）。
      - ◆输出：

        更新后的索引文件。

```rust
// 索引结构体定义
pub struct Index {
    repo_path: String,          // 关联的仓库路径
    staged_files: HashSet<String> // 暂存文件集合
}
```

```rust
impl Index {
    /// 加载现有索引
    pub fn load(repo_path: &str) -> Self {
        let index_file = format!("{}/.git/index", repo_path);

        // 读取逻辑伪代码:
        if 索引文件存在 {
            将文件内容解析为文件路径集合
        } else {
            初始化空集合
        }

        Index { 返回初始化后的实例 }
    }

    /// 添加文件到暂存区
    pub fn stage_file(&mut self, path: &str) {
        // 核心操作:
        1. 将路径插入集合
        2. 持久化到磁盘
    }

    /// 从暂存区移除文件
    pub fn unstage_file(&mut self, path: &str) {
        // 核心操作:
        1. 从集合删除路径
        2. 持久化到磁盘
    }

    /// 内部保存方法
    fn persist(&self) {
        // 存储格式:
        将集合内容序列化为逐行路径文本
        写入.git/index文件
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢core各模块实现框架
    - ◘reference模块
      - ◆管理分支和标签引用。
      - ◆创建、删除和解析引用。
      - ◆输入：
        引用名称（如分支名或标签名）。
        操作类型（创建、删除、解析）。
      - ◆输出：
        引用的 SHA-1 哈希值。

```rust
// 引用管理器（不存储状态，纯操作类）
pub struct Reference;

impl Reference {
    /// 创建引用文件
    pub fn create(repo_path: &str, ref_name: &str, target_hash:
&str) {
        // 实现逻辑：
        1. 构建引用路径："".git/refs/[heads|tags]/[name]""
        2. 将哈希值写入目标文件
        3. 错误时panic（实际项目应返回Result）
    }

    /// 删除引用文件
    pub fn delete(repo_path: &str, ref_name: &str) {
        // 实现逻辑：
        1. 定位引用文件路径
        2. 删除文件系统对应文件
    }

    /// 解析引用内容
    pub fn resolve(repo_path: &str, ref_name: &str) ->
Option<String> {
        // 实现逻辑：
        if 引用文件存在 {
            读取文件内容并去除空白字符
            Some(哈希字符串)
        } else {
            None
        }
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢core各模块实现框架
    - ◘commit模块
      - ◆表示提交对象。
      - ◆创建和解析提交对象。
      - ◆输入：

        提交信息（如提交者、提交时间、父提交、树对象）。
      - ◆输出：

        提交对象的 SHA-1 哈希值。

```rust
// 提交对象构造器
pub struct CommitBuilder;

impl CommitBuilder {
    /// 创建新提交对象
    pub fn create_commit(
        repo_path: &str,
        tree_hash: String,        // 关联的树对象哈希
        parent_commit: Option<String>, // 父提交哈希
        author_info: String,      // 作者信息
        commit_message: String    // 提交信息
    ) -> String {                 // 返回新提交的哈希

        // 1. 构造提交内容：
        let timestamp = 获取当前RFC2822格式时间;
        let commit_content = 格式化字符串包含：
            - tree [树哈希]
            - parent [父提交哈希]（可选）
            - author [作者信息] [时间戳]
            - 空行
            - [提交信息]

        // 2. 存储为Git对象
        let commit_obj = Object::Commit(commit_content);
        commit_obj.save(repo_path)  // 返回对象哈希
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢core各模块实现框架
    - ◘blob模块
      - ◆表示文件内容。
      - ◆创建和解析 blob 对象。
      - ◆输入：

        文件内容。
      - ◆输出：

        blob 对象的 SHA-1 哈希值。

```rust
// 数据对象处理器
pub struct BlobProcessor;

impl BlobProcessor {
    /// 创建Blob对象
    pub fn create_blob(
        repo_path: &str,       // 仓库路径
        content: String        // 文件原始内容
    ) -> String {              // 返回对象哈希

        // 核心处理流程：
        1. 将内容包装为Blob类型对象
        2. 调用object.save存储
        3. 返回生成的SHA1哈希
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘init模块
      - ◆初始化一个新的 Git 仓库。
      - ◆创建 .git 目录及其子目录和文件。
      - ◆输入：

        初始化路径（默认为当前目录 . 或指定路径）。
      - ◆输出：

        初始化成功的消息。

```rust
// Git 初始化命令处理器
pub fn git_init(目标路径: &str) {
    // 核心操作流程：
    1. 调用仓库核心模块的初始化方法
       - 创建标准.git目录结构
       - 生成必要的元文件

    2. 输出标准化成功消息
       "已在[路径]初始化空的Git仓库"
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘add模块
      - ◆将文件添加到暂存区（索引文件 index）。
      - ◆输入：

        文件路径。
      - ◆输出：

        文件成功添加到暂存区的消息。

```
// 文件暂存命令伪实现
pub fn git_add(
    repo_path: &str,    // 仓库根路径
    file_path: &str     // 要添加的文件路径
) {
    // 1. 加载现有暂存区状态
    let mut staging_area = 从.git/index加载索引();

    // 2. 将文件加入暂存区
    staging_area.标记文件为已暂存(file_path);

    // 3. 持久化更新后的索引
    将索引状态保存回.git/index文件;

    // 4. 用户反馈
    打印"Added [file_path] to staging area";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘rm模块
      - ◆从暂存区删除文件
      - ◆并从文件系统中删除文件（可选）。
      - ◆输入：

        文件路径。
      - ◆输出：

        文件成功从暂存区删除的消息。

```
// 文件移除命令伪实现
pub fn git_remove(
    repo_path: &str,     // 仓库根路径
    file_path: &str,     // 要移除的文件路径
    force_mode: bool     // 是否强制删除工作区文件
) {
    // 1. 加载当前暂存区状态
    let mut staging_index = 从.git/index加载索引();

    // 2. 从索引中移除文件记录
    staging_index.删除文件记录(file_path);

    // 3. 可选删除工作区文件
    if force_mode {
        删除工作区文件(file_path);
    }

    // 4. 保存更新后的索引
    将新索引状态写入.git/index;

    // 5. 用户反馈
    打印"Removed [file_path] from tracking";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘commit模块
      - ◆将暂存区的内容提交到仓库
      - ◆创建一个新的提交对象。
      - ◆输入：
        提交信息（如提交者、
        提交时间、父提交、树对象）。
      - ◆输出：
        提交成功的消息，包含提交的哈希值。

```rust
pub fn git_commit(
    repo_path: &str,    // 仓库根路径
    commit_message: &str // 提交信息
) {
    // 1. 初始化仓库对象
    let repo = 初始化仓库(repo_path);

    // 2. 加载当前索引
    let mut staging_index = 从.git/index加载索引();

    // 3. 创建树对象哈希
    let tree_hash = 创建树对象(repo_path, staging_index.获取文件条目());

    // 4. 获取当前分支的最新提交
    let parent_commit = 获取当前分支提交(repo_path, "master");

    // 5. 创建新的提交对象
    let commit_hash = 创建提交(
        repo_path,
        tree_hash,
        parent_commit,
        "Author Name".to_string(),
        commit_message.to_string()
    );

    // 6. 更新当前分支的引用，指向新的提交
    更新分支引用(repo_path, "master", &commit_hash);

    // 7. 输出提交信息
    打印"Committed changes: [commit_hash]";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘branch模块
      - ◆创建、删除或列出分支。
      - ◆输入：

        分支名称。

        操作类型（创建、删除、列出）。
      - ◆输出：

        分支操作成功的消息。

```rust
pub fn git_branch(
    repo_path: &str,    // 仓库根路径
    branch_name: &str,  // 分支名称
    delete: bool        // 是否删除分支
) {
    if delete {
        // 1. 删除分支
        删除分支(repo_path, branch_name);

        // 2. 用户反馈
        打印"Deleted branch [branch_name]";
    } else {
        // 1. 获取当前分支的最新提交哈希
        let commit_hash = 获取当前分支提交(repo_path, "current_branch_name");

        // 2. 创建新分支
        创建分支(repo_path, branch_name, commit_hash);

        // 3. 用户反馈
        打印"Created branch [branch_name]";
    }
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘checkout模块
      - ◆切换到指定的分支或提交。
      - ◆输入：

        分支名称或提交哈希。
      - ◆输出：

        切换成功的消息。

```
pub fn git_checkout(
    repo_path: &str,    // 仓库根路径
    target: &str        // 目标分支名或提交哈希
) {
    // 1. 解析目标，检查是否为分支或提交哈希
    let target_hash = match 解析目标为哈希(repo_path, target) {
        Some(hash) => hash,           // 如果是分支，获取对应的提交哈希
        None => target.to_string(),   // 如果是提交哈希，直接使用
    };

    // 2. 更新 HEAD 指向目标（分支或提交哈希）
    更新HEAD指向(repo_path, &target_hash);

    // 3. 用户反馈
    打印"Switched to branch or commit [target]";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘merge模块
      - ◆合并分支到当前分支。
      - ◆输入：
        分支名称。
      - ◆输出：
        合并成功的消息。

```rust
pub fn git_merge(
    repo_path: &str,          // 仓库根路径
    branch_name: &str         // 目标分支名称
) {
    // 1. 获取当前分支的最新提交哈希
    let current_branch = "master";           // 假设当前分支为 master
    let current_commit_hash = 获取当前分支提交(repo_path, current_branch);

    // 2. 获取目标分支的最新提交哈希
    let target_commit_hash = 获取目标分支提交(repo_path, branch_name);

    // 3. 创建一个新的合并提交
    let merge_commit_hash = 创建合并提交(
        repo_path,                           // 仓库路径
        "tree_hash_placeholder",             // 提交树哈希（占位符）
        Some(current_commit_hash),           // 当前分支提交哈希
        "Author Name".to_string(),           // 作者信息
        格式化"Merge branch [branch_name]"   // 合并信息
    );

    // 4. 更新当前分支的引用，指向新的合并提交
    更新当前分支引用(repo_path, current_branch, &merge_commit_hash);

    // 5. 用户反馈
    打印"Merge branch [branch_name] into [current_branch]";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘fetch模块
      - ◆从远程仓库拉取数据，更新本地的远程分支引用。
      - ◆输入：

        远程仓库地址。
      - ◆输出：

        拉取成功的消息。

```
pub fn git_fetch(
    repo_path: &str,          // 仓库根路径
    remote_url: &str          // 远程仓库的 URL
) {
    // 1. 与远程仓库交互，获取数据（这里假设已经拉取到本地）
    拉取远程数据(repo_path, remote_url);

    // 2. 用户反馈
    打印"Fetched data from remote repository [remote_url]";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘pull模块
      - ◆从远程仓库拉取数据并合并到当前分支。
      - ◆输入：

        远程仓库地址。
      - ◆输出：

        拉取并合并成功的消息。

```
pub fn git_pull(
    repo_path: &str,          // 仓库根路径
    remote_url: &str          // 远程仓库的 URL
) {
    // 1. 拉取远程数据
    执行git_fetch(repo_path, remote_url);

    // 2. 合并远程分支到当前分支
    执行git_merge(repo_path, "remote_branch_name"); // 假设远程分支为 "remote_branch_name"

    // 3. 用户反馈
    打印"Pulled data from remote repository and merged into current branch";
}
```

基于Rust的Git系统-具体实现

- 部分模块代码框架
  - ➢commands各模块实现框架
    - ◘push模块
      - ◆将本地分支的更改推送到远程仓库。
      - ◆输入：

        远程仓库地址。
      - ◆输出：

        推送成功的消息。

```rust
pub fn git_push(
    repo_path: &str,          // 仓库根路径
    remote_url: &str          // 远程仓库的 URL
) {
    // 1. 将本地更改推送到远程仓库
    推送本地更改到远程仓库(repo_path, remote_url);

    // 2. 用户反馈
    打印"Pushed changes to remote repository [remote_url]";
}
```

基于Rust的Git系统-测试

- 评分标准
  - OJ平台评分
    - 占总成绩的50%，基于公开测试用例和隐藏测试用例的通过情况。
    - 小队的代码提交后，会在OJ后台编译，我们将编译后的可执行文件拷贝到测试用例执行目录。假定编译后可执行文件为git，则会用编译后的git来执行Git系统的基本功能、分支管理与合并或进阶功能。
  - 线下演示与问答
    - 占总成绩的30%，小组需要展示项目的功能，并回答老师的提问。
  - 代码质量与文档
    - 占总成绩的20%，评估代码的可读性、模块化设计、注释、文档完整性等。

基于Rust的Git系统-测试

- 评分标准
  - OJ平台评测过程
    - 拉取提交的.zip代码文件；
    - 解压文件并执行`cargo build`进行编译；
    - 拷贝target/debug/rust-git（可执行文件统一命名rust-git，方便评测）到../testcases；
    - 拉取一组测试用例，每个测试用例放到testcases目录下的独立目录中，如test1、test2等等；
    - 对于每一个测试用例，先执行rust-git，将其初始化为Git仓库，再进行一系列测试。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例1： git init
      - ◆描述：初始化一个新的Git仓库。
      - ◆前置条件：

        创建test1目录，并将rust-git拷贝到test1目录。
      - ◆输入：

        rust-git init
      - ◆输出：

        ✓test1目录下包含.git目录，创建这个目录即可，可输出自定义提示信息。
      - ◆验证：

        ✓.git目录存在且不为空。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例1： git init 验证过程

```bash
#!/bin/bash
# 当前目录位于 testcases
cd testcases
# 创建一个空目录 test1
mkdir test1
# 拷贝 rust-git 到 test1 目录
cp rust-git test1/
# 进入 test1 目录
cd test1
# 执行 rust-git init
./rust-git init
# 验证 .git 目录是否存在且不为空
if [ -d ".git" ] && [ "$(ls -A .git)" ]; then
    echo "Test 1 passed: .git directory exists and is not empty"
else
    echo "Test 1 failed: .git directory does not exist or is empty"
    exit 1
fi
```

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例2： git add和git commit
      - ◆描述：添加文件并提交。
      - ◆前置条件：

        创建test2目录，将rust-git拷贝到test2目录，最后执行rust-git init。
      - ◆输入：

```
echo "Hello, Rust!" > test.txt
./rust-git add test.txt
./rust-git commit -m "Initial commit"
```

      - ◆输出：
        - ✓test2目录下包含test.txt文件，输出SHA-1哈希值。 commit之后直接输出hash值到stderr。
      - ◆验证：
        - ✓test.txt文件存在。
        - ✓.git/objects目录下生成新的对象文件，.git/refs/heads/master指向新的提交。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例2： git add和git commit 验证过程

```bash
#!/bin/bash
# 当前目录位于 testcases
# 创建一个空目录 test2
mkdir test2
# 拷贝 rust-git 到 test2 目录
cp rust-git test2/
# 进入 test2 目录
cd test2
# 执行 rust-git init
./rust-git init
```

前置条件

```bash
# 创建文件 test.txt 并添加内容
echo "Hello, Rust!" > test.txt

# 检查 test.txt 文件是否存在
if [ -f "test.txt" ]; then
    echo "test.txt exists"
else
    echo "test.txt does not exist"
    exit 1
fi

# 执行 git add 和 git commit
./rust-git add test.txt
./rust-git commit -m "Initial commit"
```

测试add和commit

```bash
# 验证 .git/objects 目录是否不为空
if [ "$(ls -A .git/objects)" ]; then
    echo ".git/objects directory is not empty"
else
    echo ".git/objects directory is empty"
    exit 1
fi

# 验证 .git/refs/heads/master 文件是否存在且不为空
if [ -s ".git/refs/heads/master" ]; then
    echo ".git/refs/heads/master exists and is not empty"
else
    echo ".git/refs/heads/master does not exist or is empty"
    exit 1
fi

echo "Test 2 passed: git add and git commit succeeded"
```

验证过程

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例3： git branch和git checkout
      - ◆描述：创建分支并切换。
      - ◆前置条件：

        创建test3目录，将rust-git拷贝到test3目录，执行基础测试用例2（创建默认的master分支）。
      - ◆输入：

```
./rust-git branch test
./rust-git checkout test
```

      - ◆输出：

        ✓可输出自定义提示信息。
      - ◆验证：

        ✓.git/refs/heads/test是否存在，.git/HEAD是否指向.git/refs/heads/test。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例3： git branch和git checkout 验证过程

```
# 当前目录位于 testcases
# 创建一个空目录 test3
mkdir test3
# 拷贝 rust-git 到 test3 目录
cp rust-git test3/
# 进入 test3 目录
cd test3
# 执行 rust-git init
./rust-git init
# 创建文件 test.txt 并添加内容
echo "Hello, Rust!" > test.txt
# 执行 git add 和 git commit
./rust-git add test.txt
./rust-git commit -m "Initial commit"
```

前置条件

```
# 执行 git branch test
./rust-git branch test
# 执行 git checkout test
./rust-git checkout test

# 验证 .git/refs/heads/test 文件是否存在
if [ -f ".git/refs/heads/test" ]; then
    echo ".git/refs/heads/test exists"
else
    echo ".git/refs/heads/test does not exist"
    exit 1
fi
# 验证 .git/HEAD 文件是否指向 refs/heads/test
if grep -q "ref: refs/heads/test" ".git/HEAD"; then
    echo ".git/HEAD points to refs/heads/test"
else
    echo ".git/HEAD does not point to refs/heads/test"
    exit 1
fi
echo "Test 3 passed: git branch and git checkout succeeded"
```

验证过程

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例4： git merge
      - ◆描述：合并两个分支。
      - ◆前置条件：

        创建test4目录，将rust-git拷贝到test4目录，执行rust-git init。
      - ◆输入：

        main分支创建，test分支创建，main分支与test分支合并。
      - ◆输出：

        ✓main可执行文件输出结果`测试分支合并\n`。
      - ◆验证：

        ✓可执行文件执行是否成功，结果是否符合预期。

```rust
use std::fs::File;
use std::io::{self, Read};

fn main() -> io::Result<()> {
    // 打开当前目录下的 test.txt 文件
    let mut file = File::open("test.txt")?;

    // 创建一个字符串来存储文件内容
    let mut contents = String::new();

    // 读取文件内容到字符串
    file.read_to_string(&mut contents)?;

    // 打印文件内容
    println!("{}", contents);

    Ok(())
}
```

图1. 代码示例

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例4： git merge 验证过程

```
# 当前目录位于 testcases
# 创建一个空目录 test4
mkdir test4
# 拷贝 rust-git 到 test4 目录
cp rust-git test4/
# 进入 test4 目录
cd test4
# 执行 rust-git init
./rust-git init
```

前置条件

```
# 创建 main 分支并切换到 main 分支
./rust-git checkout -b main
# 创建 main.rs 文件并添加内容
echo 'use std::fs::File;
use std::io::{self, Read};
fn main() -> io::Result<()> {
    // 打开当前目录下的 test.txt 文件
    let mut file = File::open("test.txt")?;
    // 创建一个字符串来存储文件内容
    let mut contents = String::new();
    // 读取文件内容到字符串
    file.read_to_string(&mut contents)?;
    // 打印文件内容
    println!("{}", contents);
    Ok(())
}' > main.rs
# 添加并提交 main.rs
./rust-git add .
./rust-git commit -m "update main.rs"
```

main分支创建

```
# 创建 test 分支
./rust-git branch test
# 切换到 test 分支
./rust-git checkout test
# 创建 test.txt 文件并添加内容
echo "测试分支合并" > test.txt
# 添加并提交 test.txt
./rust-git add .
./rust-git commit -m "update test.txt"
# 切换回 main 分支
./rust-git checkout main
# 合并 test 分支
./rust-git merge test
# 编译 main.rs
rustc main.rs
# 运行 main.rs 并检查输出
if ./main | grep -q "测试分支合并"; then
    echo "Test 4 passed: git merge succeeded and main.rs output is correct"
else
    echo "Test 4 failed: main.rs output is incorrect"
    exit 1
Fi
```

分支合并及验证过程

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例5： git rm
      - ◆描述：从暂存区或工作区删除文件，git通常会保留提交历史，不会直接删除.git/objects下的文件。我们的Git系统可以直接删除文件以及.git/objects的文件。
      - ◆前置条件：

        创建test5目录，将rust-git拷贝到test5目录，执行rust-git init。
      - ◆输入：

        创建main分支，创建delete.txt文件并提交，再创建temp分支执行rm命令删除文件，最后执行merge命令合并temp分支。
      - ◆输出：

        ✓输入命令执行结束，main分支不包含delete.txt文件。
      - ◆验证：

        ✓delete.txt文件不存在。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘基础测试用例5： git rm 验证过程

```
# 当前目录位于 testcases
# 创建一个空目录 test5
mkdir test5
# 拷贝 rust-git 到 test5 目录
cp rust-git test5/
# 进入 test5 目录
cd test5
# 执行 rust-git init
./rust-git init
```

前置条件

```
# 创建 main 分支并切换到 main 分支
./rust-git checkout -b main
# 创建 delete.txt 文件并添加内容
echo "Delete me" > delete.txt
# 添加并提交 delete.txt
./rust-git add .
./rust-git commit -m "add file"
# 创建 temp 分支
./rust-git branch temp
# 切换到 temp 分支
./rust-git checkout temp
# 删除 delete.txt 文件
./rust-git rm delete.txt
./rust-git commit -m "delete file"
```

rm执行过程

```
# 切换回 main 分支
./rust-git checkout main
# 合并 temp 分支
./rust-git merge temp

# 检查当前目录下是否不存在 delete.txt 文件
if [ ! -f "delete.txt" ]; then
    echo "Test 5 passed: git rm succeeded"
else
    echo "Test 5 failed"
    exit 1
```

验证过程

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘高级测试用例1： 冲突合并。
      - ◆描述：测试分支合并时的冲突处理。
      - ◆前置条件：

        创建ad_test1目录，将rust-git拷贝到ad_test1目录，执行rust-git init。
      - ◆输入：

        main分支创建，test分支创建，main分支与test分支合并。
      - ◆输出：

        ✓冲突所在的文件名以及冲突行号。
      - ◆验证：

        ✓是否有冲突，冲突内容是否包含文件名和行号。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘高级测试用例1： 冲突合并验证过程
      - ✓冲突格式为 Merge confilict in 文件名:行号，最重要的是给出文件名以及行号，如果是多行，格式为区间[a, b]。

```
# 当前目录位于 testcases
# 创建一个空目录 test5
mkdir test5
# 拷贝 rust-git 到 test5 目录
cp rust-git test5/
# 进入 test5 目录
cd test5
# 执行 rust-git init
./rust-git init
```

前置条件

```
# 创建 main 分支并切换到 main 分支
./rust-git checkout -b main
# 创建 test.txt 文件并添加内容
echo "main分支修改内容" > test.txt
# 添加并提交 test.txt
./rust-git add .
./rust-git commit -m "main"

# 创建 test 分支
./rust-git branch test
# 切换到 test 分支
./rust-git checkout test
# 修改 test.txt 文件并添加内容
echo "test分支修改内容" > test.txt
# 添加并提交 test.txt
./rust-git add .
./rust-git commit -m "test"
```

main和test分支创建

```
# 切换回 main 分支
./rust-git checkout main
# 在 main 分支上进行额外的提交
echo "additional content in main" > additional.txt
./rust-git add .
./rust-git commit -m "additional commit in main"

# 合并 test 分支并检查是否提示冲突
if ./rust-git merge test 2>&1 | grep -q “Merge conflict in test.txt:1"; then
    echo "Conflict detected correctly"
else
    echo "Conflict not detected"
    exit 1
fi

echo "Advanced Test 1 passed: Conflict merge detected and handled correctly"
```

冲突检测

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘高级测试用例2： 大文件处理。
      - ◆描述：测试大文件的添加和提交。
      - ◆前置条件：

        创建ad_test2目录，将rust-git拷贝到ad_test2目录，执行rust-git init。
      - ◆输入：

```
./rust-git checkout -b main
cp /path/to/large_file.bin .
./rust-git add .
./rust-git commit -m "Add large file"
```

      - ◆输出：
        - ✓ad_test2目录下包含large_file.bin文件，输出SHA-1哈希值。commit之后直接输出hash值到stderr。
      - ◆验证：
        - ✓.git/objects目录下生成新的对象文件。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘高级测试用例2： 大文件处理验证过程

前置条件

```
# 当前目录位于 testcases
# 创建 large_file.bin
dd if=/dev/zero of=large_file.bin bs=1M count=10
# 创建一个空目录 ad_test2
mkdir ad_test2
# 拷贝 rust-git到ad_test2 目录
cp rust-git ad_test2/
# 进入 ad_test2 目录
cd ad_test2
```

大文件提交

```
# 执行 rust-git init
./rust-git init
# 创建 main 分支并切换到 main 分支
./rust-git checkout -b main
# 拷贝large_file.bin 到 ad_test2 目录
cp ../large_file.bin ad_test2/
# 添加 large_file.bin
./rust-git add large_file.bin
# 提交 large_file.bin
commit_hash=$(/usr/bin/git commit -m "Add large file")

# 检查提交是否成功
if [ -z "$commit_hash" ]; then
    echo "Commit hash not found"
    exit 1
fi

echo "Committed changes: $commit_hash"
```

检验过程

```
# 提取 large_file.bin 的 SHA-1 哈希值
file_hash=$(/usr/bin/git hash-object large_file.bin)

# 检查 .git/objects 目录下是否存在对应的对象文件
object_dir=".git/objects/${file_hash:0:2}"
object_file="$object_dir/${file_hash:2}"

if [ -d "$object_dir" ] && [ -f "$object_file" ]; then
    echo "Object file $object_file exists"
else
    echo "Object file $object_file does not exist"
    exit 1
fi

echo "Advanced Test 2 passed: Large file is correctly stored"
```

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘高级测试用例3： 分支删除。
      - ◆描述：测试分支删除功能。
      - ◆前置条件：

        创建ad_test3目录，将rust-git拷贝到ad_test3目录，执行rust-git init。
      - ◆输入：

        创建main分支和temp分支，合并main分支和temp分支，删除temp分支。
      - ◆输出：

        ✓ad_test2目录下main.txt和temp.txt，可选的输出信息。
      - ◆验证：

        ✓.git/refs/heads目录下只有main文件。

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘高级测试用例3： 分支删除验证过程

前置条件

```
# 当前目录位于 testcases
# 创建一个空目录 ad_test3
mkdir ad_test3
# 拷贝 rust-git 到 ad_test3 目录
cp rust-git ad_test3/
# 进入 ad_test3 目录
cd ad_test3
# 执行 rust-git init
./rust-git init
```

main和temp分支创建

```
# 创建 main 分支并切换到 main 分支
./rust-git checkout -b main
# 创建 test.txt 文件并添加内容
echo "main分支创建" > main.txt
# 添加并提交 test.txt
./rust-git add .
./rust-git commit -m "update main"
# 创建 temp 分支
./rust-git branch temp
# 切换到 temp 分支
./rust-git checkout temp
# 创建 temp.txt 文件并添加内容
echo "test分支创建" > temp.txt
# 添加并提交 temp.txt
./rust-git add .
./rust-git commit -m "update temp"
```

检验过程

```
# 切换回 main 分支
./rust-git checkout main
# 合并 temp 分支
./rust-git merge temp)
# 删除 temp 分支
./rust-git branch -d temp

# 检查当前目录下是否存在 test.txt 和 temp.txt 文件
if [ -f "test.txt" ] && [ -f "temp.txt" ]; then
    echo "Both test.txt and temp.txt exist in the working directory"
else
    echo "Files are missing in the working directory"
    exit 1
fi

# 检查 .git/refs/heads 目录下是否只存在 main 分支的引用文件
if [ -f ".git/refs/heads/main" ] && [ ! -f ".git/refs/heads/temp" ]; then
    echo "Only main branch reference exists in .git/refs/heads"
else
    echo "Branch references other than main exist in .git/refs/heads"
    exit 1
fi

echo "Advanced Test 3 passed: Branch temp is deleted and temp.txt is merged into main"
```

基于Rust的Git系统-测试

- 测试用例设计
  - ➢公开测试用例
    - ◘进阶测试用例1：远程操作（可选）。
      - ◆描述：测试远程仓库的拉取和推送。OJ系统中不包含远程操作相关测试用例，实现可选功能的小组，可在线下汇报时演示。
      - ◆步骤：
        1. 在远程仓库中创建一个分支remote_branch。
        2. 执行git fetch，检查是否能够获取远程分支。
        3. 执行git pull origin remote_branch，检查是否能够合并远程分支。
        4. 执行git push origin main，检查是否能够推送本地提交。
      - ◆预期输出：
        - ✓远程操作成功执行。
      - ◆补充信息
        - ✓远程仓库可以用现有的代码仓库，也可以本地部署

谢谢！