Java-12

一、Task1.Git

1. **git操作**
    - **.git** 文件
        - **hooks 存储了 Git 钩子脚本的模板文件** (一些可执行的脚本可以用于自定义和控制 Git 操作的行为)
            - pre-commit：在执行提交操作前运行，可以用于代码检查、格式化等操作，以确保提交的代码符合规范
            - pre-receive：在执行推送操作前执行，可以用于进行服务端校验、权限验证等操作，以控制推送到远程仓库的内容
            - post-commit：在执行提交操作后运行，可以用于发送通知、执行后续操作等
            - post-receive：在执行推送操作后运行，可以用于执行服务器端处理、触发自动部署等操作
            - **注意:** 需要去掉 **.sample** 后缀并赋予可执行权限后才会生效
        - **info 存储了一些额外的 Git 配置文件**
            - `exclude`：全局性排除文件
        - **logs 用于存储 Git 仓库的引用日志信息**
            - refs：存储各个引用（如分支、标签）的引用日志信息。每个应用都对应的一个子目录，如：

              refs/heads 用于存储分支的引用日志

              refs/tags 用于存储标签的引用日志
            - HEAD：存储 HEAD 引用的变动记录
        - **objects(重要) 用于存储 Git 仓库中的所有对象**
            - `info` 目录：存储一些辅助信息和索引文件，用于加快对象访问速度
            - `pack` 目录：存储了使用 Git 的打包机制（packing）压缩的对象文件
            - `哈希` 目录: 用于存储具体的对象文件，每个对象的文件名是由哈希值组成
        - **refs(重要) 用于存储指向提交对象的引用(如分支引用和标签引用)**
            - heads：存储分支引用，每个分支都对应一个文件，文件名与分支名称相同，内容是指向分支最新提交的指针
            - tags：存储标签引用，每个标签都对应一个文件，文件名与标签名称相同，内容是指向标签的对象的指针
            - remotes：存储远程引用，每个远程仓库都对应一个子目录，目录名与远程仓库名称相同。在每个远程仓库目录下，可以存储与该远程仓库相关的引用，如远程分支引用
        - **COMMIT_EDITMSG** 存储最近一次 Git 提交时的提交消息
        - **config 用于存储仓库级别的配置选项**
            - core：包含与 Git 核心功能相关的配置选项，如仓库路径、忽略文件权限等
            - remote: 用于定义与远程仓库的连接和交互的配置选项，可以指定远程仓库的 URL、分支跟踪等
            - branch: 用于定义分支相关的配置选项，如分支的追踪关系、合并策略等
            - user: 用于设置 Git 用户的姓名和邮箱地址，这些信息会出现在提交记录中
            - alias: 用于定义 Git 命令的别名，可以简化常用命令的输入
        - **FETCH_HEAD 用于存储最近一次从远程仓库中 `fetch` 的提交记录**

        - **HEAD(重要) 用于指示当前所在分支或提交**
            - **直接指向某个提交的哈希值**，表示当前处于分离头指针（detached HEAD）状态，即不在任何分支上工作
            - **以 ref: refs/heads/ 的形式**，表示当前所在的分支
        - **index(重要) 暂存区或索引**

          使用 `git status` 可以查看 **index** 的状态
    - 分支
        - Git 分支实际上是指向更改快照的指针
        - 创建分支
            - `git branch <name>` 创建分支
            - `git checkout -b <name>` 创建并切换到该分支
            - `git checkout` 切换分支
        - 查看分支
            - `git branch` 查看本地分支
            - `git branch -r` 查看远程分支
            - `git branch -a` 查看所有分支(本地和远程)
            - `git branch -v` 查看分支和最后提交信息
        - 删除分支
            - `git branch -d` 删除本地分支
            - `git branch -D` 强制删除未合并的分支
            - `git push origin --delete` 删除远程分支
        - 合并分支
            - `git merge` 将其他分支合并到当前分支，**保留历史记录分支结构**
            - `git rebase` 将其他分支合并到当前分支，**改变提交历史，保持线性**
            - **分支冲突**
                - 发生场景:两人对同文件同行进行修改并提交

                  同一文件在不同分支都有重命名
                - `git status` 查看所有冲突文件
                - 能自动解决的冲突(缺少): `git pull` 后再 `git push`
                - 手动解决:**vim** 打开文件进行编辑，选择保留修改(最后删除 `<<<<<<<`、`=======` 和 `>>>>>>>` 这些标记)

                  可以使用 **VSCode** 等图形化工具
    - `HEAD`
        - 状态
            1. 指向一个分支(最常见)
            2. 指向某个提交记录(`git checkout` 直接指向某个提交)
        - 相对引用
            - `HEAD~` 表示 `HEAD` 的前一个提交，加数字表示前第几个提交
            - `HEAD^` 表示父提交

              父提交 指的是一个提交**直接基于**的那个更早的提交

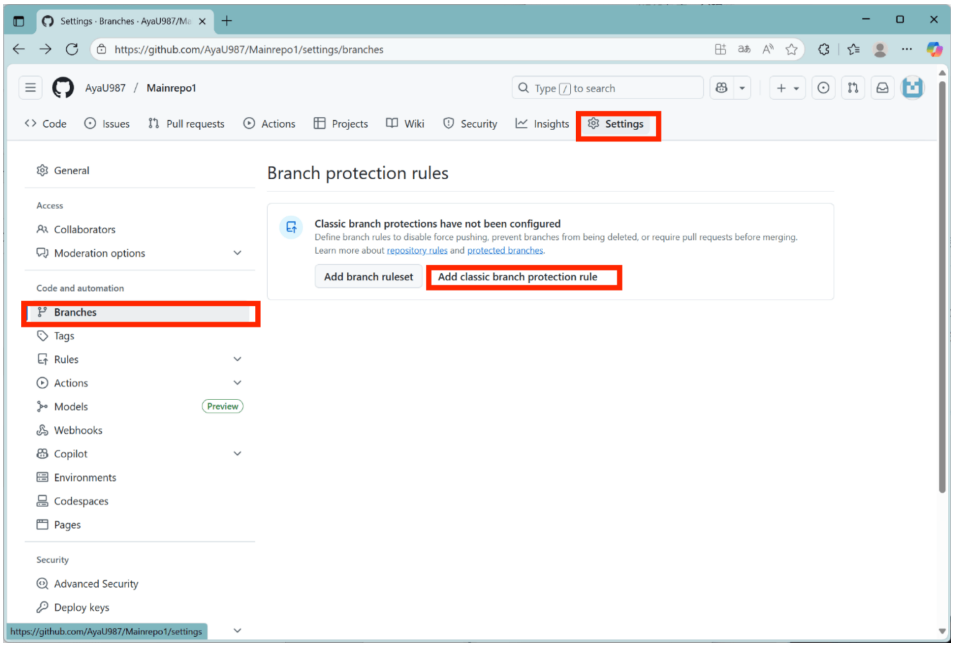
              合并提交 则会有多个 **父提交** ， 此时

```
HEAD^1 主分支的前一个提交
HEAD^2 被合并的分支的前一个提交
```

- 远程分支
  - 远程分支是指向**远程仓库**中分支的引用
  - **main与origin/main**

    `main` 是本地仓库的**主分支**，可以直接在此提交更改

    `origin/main` 是**远程跟踪分支**，也在本地仓库但是只读不能直接提交，代表远程状态，只能通过 `git fetch` `git pull` 进行更新
- `fetch` 和 `pull`
  - `fetch` 下载远程的最新提交和分支信息，不影响工作文件和本地分支(预览)
  - `pull` 下载远程更新并合并到当前分支，会修改工作文件和本地分支

    相当于 `git fetch` + `git merge`
- 锁定的main分支
  - 目的: 防止直接推送，强制代码审查，使所有成员使用相同的代码合并流程
  - **github 锁定分支**
  - 

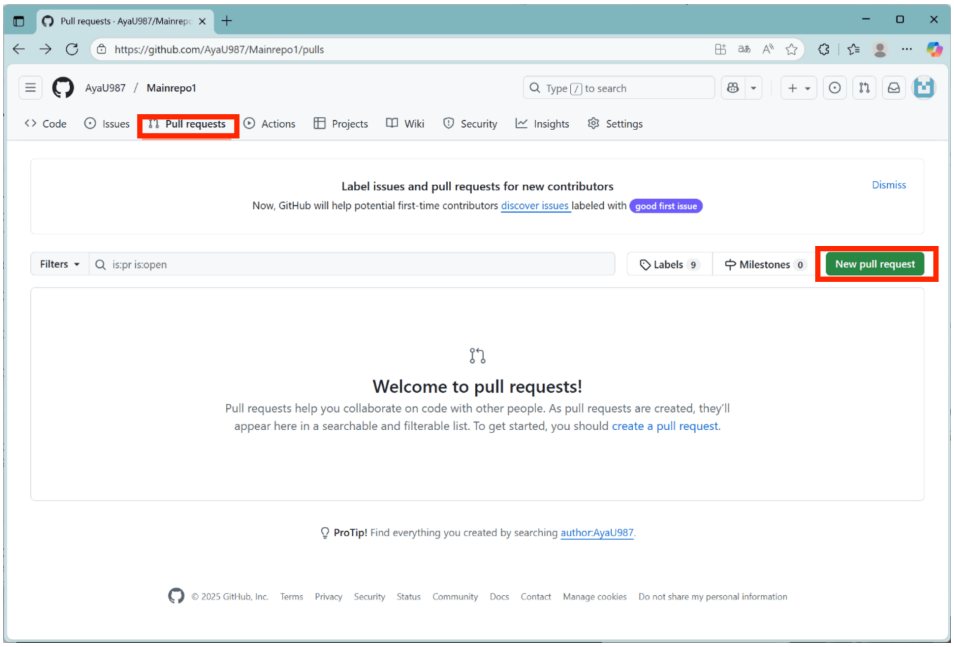
- 添加保护规则即可

Branch name pattern *

Protect matching branches

- [ ] **Require a pull request before merging**
  When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.
- [ ] **Require status checks to pass before merging**
  Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.
- [ ] **Require conversation resolution before merging**
  When enabled, all conversations on code must be resolved before a pull request can be merged into a branch that matches this rule. Learn more about requiring conversation completion before merging.
- [ ] **Require signed commits**
  Commits pushed to matching branches must have verified signatures.
- [ ] **Require linear history**
  Prevent merge commits from being pushed to matching branches.
- [ ] **Require deployments to succeed before merging**
  Choose which environments must be successfully deployed to before branches can be merged into a branch that matches this rule.
- [ ] **Lock branch**
  Branch is read-only. Users cannot push to the branch.
- [ ] **Do not allow bypassing the above settings**

- **PR** 即 **Pull Request**，请求将某个分支的更改合并到另一个分支(main/master)
  - 目的: 多人审查保证代码质量，便于追踪谁为什么做了某个修改(漏洞修复)
  - **push** 操作之后，在 **github** 创建 **pull request**



在 **Comparing changes** 界面填写相关信息并 **Create pull request** 即可

- **git flow** 工作流
  - 安装:
    - Linux `sudo apt-get install git-flow` (Ubuntu)

- Windows 使用 **Chocolatey(使用官网指令安装)**
  输入`choco install git -y` **(Git for Windows)**
- 使用`git flow version`验证是否成功

```
PS C:\Windows\system32> git flow version
1.12.3 (AVH Edition)
```

- 初始化 `git flow init`

  这一步实际上创建了 **Develop** 分支 **(接收其他辅助分支的合入)**
  即 `git branch develop`

  `git push -u origin develop` **(-u创建跟踪分支，建立同名分支联系)**

  功能分支(Feature)

  - 开始 `git flow feature start <feature-name>`
  - 方式一:**本地完成**
    - 合并到本地develop `git flow feature finish <feature-name>`
    - 推送本地develop分支到远程 `git push origin develop`
  - 方式二:**远程PR流程(常见流程)**
    - 推送feature到远程 `git flow feature publish <feature-name>`
    - 创建PR使推送合并到远程develop分支
    - `git flow feature finish <feature-name>`

  发布分支(Release)

  - 开始 `git flow release start <version>`
  - 完成 `git flow release finish version`

    `git push origin develop`

    `git push --tags`

  修复分支(Hotfix)

  - 开始 `git flow hotfix start version`
  - 完成 `git flow hotfix finish version`

    `git push origin develop`

    `git push --tags`

2. **模拟场景01**
   - **原因:虽然第二次把密钥文件加入了 `.gitignore`，但第一次仍在历史记录中而`git push` 实际上推送的是整个 commit 历史**
   - 解决方式:
     - 备份仓库:`git clone --mirror <仓库地址>`
     - 删除该秘钥文件 `bfg --delete-files '<file-name>' <仓库名>.git`
     - 进入克隆的仓库 `cd your-repo.git`
     - 切断引用 `git reflog expire --expire=now --all`
     - 删除冗余对象 `git gc --prune=now --aggressive`
     - 强制推送 `git push --force-with-lease`
     - 最后处理**镜像仓库**和**原仓库**
       - 原仓库 **删除**并**重新克隆**

```
cd <原仓库地址>
rm -rf <原仓库>
git clone <远程仓库>
```

- 镜像仓库 **删除**即可

```
cd <镜像仓库地址>
rm -rf <镜像仓库>
```

3. **模拟场景02**
   - **解决方案:**
     - 在本地 **develop** 分支创建 **feature** 用于备份刚才的开发

       `git checkout -b feature`
     - `git checkout develop` 回到 **develop** 分支
     - `git fetch origin develop` 检查远程 **develop** 分支是否更新
     - 比较本地与远程的不同

       `git log --oneline HEAD..origin/develop` 远程仓库有的
       `git log --oneline origin/develop..HEAD` 本地仓库有的

       `A..B` 表示 **显示在分支 B 中但不在分支 A 中的所有提交**
     - `git reset --hard origin/develop`

       重置本地 **develop** 分支到远程分支的状态

       注意 因为刚才创建了 **feature** 分支，所以这不会使刚才的修改丢失

       `git checkout -b feature` 创建了当前状态的一个完整副本 **feature** 分支
     - `git checkout feature` 切换到 **feature** 分支

       `git add .` `git commit -m “注释”` `git push -u origin feature`

       并创建 **PR** 请求，完成推送