实验一 Git和Markdown基础

班级： 21计科02

学号： B20210906220

姓名： 刘嘉璐

Github地址：https://github.com/Yalerea/pyexperiment

实验目的

1. Git基础，使用Git进行版本控制
2. Markdown基础，使用Markdown进行文档编辑

实验环境

1. Git
2. VSCode
3. VSCode插件

实验内容和步骤

第一部分 实验环境的安装

1. 安装git，从git官网下载后直接点击可以安装：git官网地址
2. 从Github克隆课程的仓库：课程的仓库地址，运行git bash应用（该应用包含在git安装包内），在命令行输入下面的命令（命令运行成功后，课程仓库会默认存放在Windows的用户文件夹下）

```
git clone https://github.com/zhoujing204/python_course.git
```

如果你在使用`git clone`命令时遇到SSL错误，请运行下面的git命令(这里假设你的Git使用了默认安装目录)：

```
git config --global http.sslCAInfo "C:/Program
Files/Git/mingw64/ssl/certs/ca-bundle.crt"
```

或者运行下面的命令：

```
git config --global http.sslVerify false
```

如果遇到错误：`error setting certificate file`，请运行下面的命令重新指定git的安全证书：

```
git config --global --unset http.sslCAInfo
git config --global http.sslCAInfo "C:/Program
Files/Git/mingw64/ssl/certs/ca-bundle.crt"
```

该仓库的课程材料后续会有更新，如果需要更新课程材料，可以在本地课程仓库的目录下运行下面的命令：

```
git pull
```

在本地的仓库内容有更新后，可以运行下面的命令，将本地仓库的内容和远程仓库的内容同步：

```
git push origin main
```

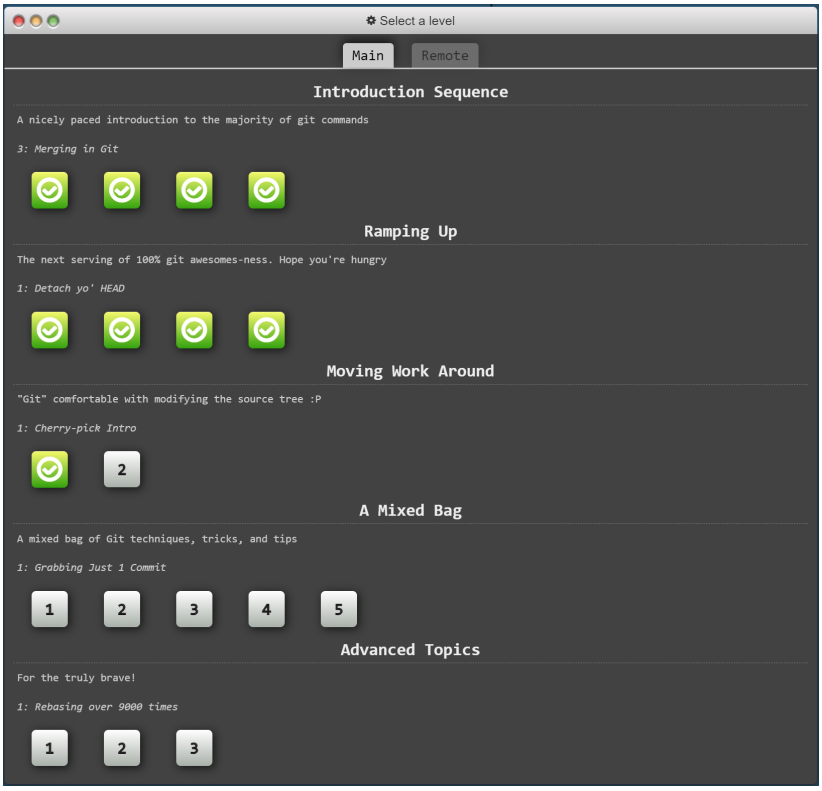
3. 注册Github账号或者Gitee帐号，创建一个新的仓库，使用上面同样的方法将该仓库clone到本地，用于存放实验报告和实验代码，使用 `git pull` 和 `git push` 命令保持远程仓库和本地仓库的同步。
4. 安装VScode，下载地址：Visual Studio Code
5. 安装下列VScode插件
    - GitLens
    - Git Graph
    - Git History
    - Markdown All in One
    - Markdown Preview Enhanced
    - Markdown PDF
    - Auto-Open Markdown Preview
    - Paste Image
    - markdownlint

第二部分 Git基础

教材《Python编程从入门到实践》P436附录D：使用Git进行版本控制，按照教材的步骤，完成Git基础的学习。

第三部分 learngitbranching.js.org

访问learngitbranching.js.org，如下图所示完成Main部分的Introduction Sequence和Ramping Up两个小节的学习。



上面你学习到的git命令基本上可以应付百分之九十以上的日常使用，如果你想继续深入学习git，可以：

- 继续学习learngitbranching.js.org后面的几个小节（包括Main和Remote）
- 在日常的开发中使用git来管理你的代码和文档，用得越多，记得越牢
- 在git使用过程中，如果遇到任何问题，例如：错误删除了某个分支、从错误的分支拉取了内容等等，请查询git-flight-rules

第四部分 Markdown基础

查看Markdown cheat-sheet，学习Markdown的基础语法

使用Markdown编辑器（例如VScode）编写本次实验的实验报告，包括实验过程与结果、实验考查和实验总结，并将其导出为 **PDF格式** 来提交。

实验过程与结果

第二部分：

1.查看Git的版本，配置Git的用户名和邮箱地址，将每个项目中主分支的默认名称设置为main。

```
git --version
git config --global user.name "Yalerea"
git config --global user.email "1301964885@qq.com"
git config --global init.defaultBranch main
```

2.创建项目并忽略文件，在git_practice文件夹下使用`git init`操作初始化空仓库，检查项目的状态

```
git init
git status
git add .
git status
```

3.将文件加入仓库,并执行命令git commit -m "Started project."提交，之后检查状态。

```
git add .
git status
git commit -m "Started project."
git status
```

4.使用git log查看提交历史，之后查看提交历史条目和项目状态，并提交修改之后的文件查看状态后再次打印提交历史条目。

```
git log
git log --pretty=oneline
git status
git commit -am "Extended greeting."
git status
git log --pretty=oneline
```

5.查看文件状态，放弃最后一次提交后对所有文件做出的所有修改，将项目恢复到最后一次提交的状态。使用checkout命令和提交的引用ID的前6个字符，检出以前的提交。之后使用git switch -命令撤销上一步操作。使用git reset --hard和永久要恢复的提交的引用ID的前6个字符，将项目重置到以前的提交。

```
git status
git restore .
git status
git log --pretty=oneline
git checkeout 0c8cd6
git switch -
git status
git log --pretty=oneline
git reset --hard 0c8cd6
git status
git log --pretty=oneline
```

6.使用命令rm -rf .git/删除目录.git，之后新建仓库，加入所有文件并执行第一次提交，最后查看状态。

```
git status
rm -rf .git/
git status
git init
git status
git add .
git commit -m "Starting over."
git status
```

第三部分实验结果

Introduction Sequence:

第一问

```
git commit
git commit
```

执行提交

第二问

```
git branch bugFix
git checkout bugFix
```

使用了'git branch newImage'生成一个新的分支，并且切换到现在的主分支。

第三问

```
git branch bugFix
git checkout bugFix
git commit
git checkout main
git commit
git merge bugFix
```

创建'bugFix'分支并切换到该分支，提交一次'bugFix'分支，切换到'main'分支，之后提交一次'main'分支，最后把'bugFix'分支的修改合入main分支。

第四问

```
git branch bugFix
git checkout bugFix
git commit
git checkout main
git commit
git checkout bugFix
git rebase main
```

创建'bugFix'分支并切换到该分支，提交一次'bugFix'分支，切换到'main'分支，'main'分支提交一次，切换到'bugFix'分支，将'main'分支的当前提交作为父节点，并合并修改点到'bugFix'分支。

Ramping Up:

第一问

```
git checkout c4
```

将'HEAD'切换到提交记录c4上

第二问

```
git chekout bugFix^
```

使用^向上移动1个位置并切换到父节点

第三问

```
git branch -f main c6
git checkout HEAD^
git branch -f bugFix HEAD^
```

把'main'分支切换到提交记录c6上，向上移动'HEAD'1个位置并切换到这个位置，把'bugFix'分支切换到提交记录c6上，

第四问

```
git reset HEAD~1
git checkout pushed
git revert HEAD
```

使用'git reset'向上移动分支，原来指向的提交记录就跟从来没有提交过一样,切换到'pushed'分支，使用'git revert'撤销'HEAD'的修改。

实验考查

请使用自己的语言回答下面的问题，这些问题将在实验检查时用于提问和答辩，并要求进行实际的操作。

1. 什么是版本控制？使用Git作为版本控制软件有什么优点?

版本控制：版本控制是一种管理和跟踪软件项目中各个版本的系统。它有助于团队协作、跟踪代码更改、返回到先前的版本以及解决代码冲突等。

使用Git作为版本控制软件的优点：

1. 分布式版本控制：Git 允许每个开发者在本地拥有完整的代码仓库副本，便于工作离线和并行开发。每个副本都包含完整的历史记录，因此降低了对中央服务器的依赖性。
2. 强大的分支支持：Git 提供了轻松创建和合并分支的功能。开发者可以在不影响主要代码线的情况下并行开发新功能或修复错误。
3. 快速和高效：Git 在处理大型代码库时非常高效。使用了各种压缩和差异算法，因此占用的磁盘空间小，传输速度快。
4. 可定制性：Git 具有灵活的配置选项，可以根据项目的需要进行定制。
5. 大型社区和支持：Git 有庞大的用户社区和广泛的文档资源，便于解决问题和得到帮助。
6. 跨平台：Git 支持多种操作系统，包括Linux、macOS和Windows，可以在不同的开发环境中使用。

Git适用于各种规模的项目，并且能够提高开发团队的协作效率和代码管理能力,是一款强大的版本控制工具。

2. 如何使用Git撤销还没有Commit的修改？如何使用Git检出（Checkout）已经以前的Commit？（实际操作）

使用Git撤销还没有Commit的修改的实际操作步骤：

1. 撤销还没有Commit的修改：

```
git checkout .
```

以上代码会清除工作区中所有尚未Commit的更改。由于会不可逆地删除未保存的更改，需要谨慎使用。

2. 检出已经存在的Commit：

```
git checkout <commit_sha>
```

`<commit_sha>` 是要检出的Commit的哈希值，可以在Git日志中找到。

如果想查看先前的Commit但是不切换，可以使用以下命令来查看Commit的内容：

```
git show <commit_sha>
```

以上代码会显示Commit的详细信息和更改内容。

3. Git中的HEAD是什么？如何让HEAD处于detached HEAD状态？（实际操作）

HEAD：
在Git中，HEAD是一个指向当前所在分支的指针或引用。它可以指向一个分支的最新提交，也可以处于detached HEAD状态，即不指向任何分支，而是直接指向一个提交（commit）。

将HEAD置于detached HEAD状态的实际操作步骤：

1. 使用`git log`命令查看想要切换到的提交的哈希值（commit hash）或唯一标识符。
2. 确保当前的工作区是干净的，没有未提交的更改。如果有未提交的更改，需要先提交或保存它们。
3. 运行命令`git checkout <commit>`，将< commit >替换为想切换的提交的哈希值或标识符。例如：`git checkout abcdefg`，其中`abcdefg`是要切换的提交的哈希值。
4. 此时，HEAD处于detached HEAD状态。

4. 什么是分支（Branch）？如何创建分支？如何切换分支？（实际操作）

分支：
分支（Branch）是指向提交历史的一条独立线索。分支允许开发人员从已有的代码中创建出新的代码副本，并且在不破坏主线的同时继续开发。

创建分支的实际操作步骤：

1. 使用命令`git branch <branch-name>`来创建一个新的分支，将替换为创建的分支的名称。例如：`git branch feature`。
2. 执行命令后，将会在当前的提交上创建一个名为< branch-name >的分支，但是HEAD仍然指向之前所在的分支。

切换分支的实际操作步骤：

1. 使用命令`git checkout <branch-name>`切换到另一个分支，将< branch-name >替换为想要切换到的分支的名称。例如：`git checkout feature`。
2. 执行命令后，HEAD将会指向< branch-name >所在的最新提交，从而切换到了指定的分支。

创建分支的同时切换到该分支：

```
git checkout -b <branch-name>
```

其中< branch-name >是创建和切换到的新分支的名称。

注意：在切换分支之前，确保工作区是干净的，没有未提交的更改。如果有未提交的更改，需要先提交或保存它们，否则切换分支可能会导致未提交的更改的丢失。

5. 如何合并分支？git merge和git rebase的区别在哪里？（实际操作）

合并分支：
在Git中，合并分支是将一个分支的更改合并到另一个分支上的操作。有两种常用的方法进行分支合并：`git merge`和`git rebase`。

`git merge`和`git rebase`的区别：

合并分支（使用`git merge`）：

1. 确保处于接收更改的目标分支上（通常是主分支），使用命令`git checkout <target-branch>`可以切换到目标分支。
2. 运行命令`git merge <source-branch>`，将< source-branch >替换为想要合并的源分支的名称。例如：`git merge feature`。
3. Git会自动将< source-branch >中的更改合并到当前所在的目标分支中。如果存在冲突，需要解决冲突并手动提交合并结果。

合并分支（使用`git rebase`）：

1. 确保处于要进行变基操作的目标分支上，使用命令`git checkout <target-branch>`切换到目标分支。
2. 运行命令`git rebase <source-branch>`，将< source-branch >替换为想要合并的源分支的名称。例如：`git rebase feature`。
3. Git会将< target-branch >中的更改保存为临时文件，并将< source-branch >中的更改应用到< target-branch >上。
4. 如果存在冲突，需要解决冲突并手动更改。
5. 完成冲突解决后，使用命令`git rebase --continue`继续进行变基操作。
6. 使用命令`git branch -f <target-branch>`将目标分支指向最新的提交。

区别：

- git merge会将源分支中的更改合并到目标分支的最新提交上，并在提交历史中保留源分支的完整记录。合并后的提交会有多个父节点。
- git rebase会将源分支中的更改应用到目标分支上，并以目标分支上的最新提交作为基础，形成一系列新的提交。它会重写提交历史，使得合并后的提交线性的排列。

6. 如何在Markdown格式的文本中使用标题、数字列表、无序列表和超链接？（实际操作）

1. 标题：

使用 # 符号加空格来表示标题的级别，一共有 1 到 6 级。例如：`# 标题一`表示一级标题。

2. 数字列表：

使用数字加点号加空格的形式来表示一个有序列表。例如：`1. 列表项一`。

3. 无序列表：

使用减号、加号或星号加空格的形式来表示一个无序列表。例如：`- 列表项一` 或 `* 列表项一`。

4. 超链接：

使用方括号 [] 来表示链接的显示文本，紧接着使用圆括号 () 来表示链接的URL或相对路径。例如：`[链接文本](http://www.example.com)`。

Markdown文本的实际操作示例：

> # 标题一
>
> ## 标题二
>
> ### 标题三
>
> 正文内容。
>
> 有序列表示例：
>
> 1. 列表项一
> 2. 列表项二
> 3. 列表项三
>
> 无序列表示例：
>
> - 列表项一
> - 列表项二
> - 列表项三
>
> 超链接示例：GitHub

实验总结

总结一下这次实验你学习和使用到的知识，例如：编程工具的使用、数据结构、程序语言的语法、算法、编程技巧、编程思想。

本次实验学习和使用到的知识主要包括以下几个方面：

1. Git基础：

- 下载配置了Git。
- 学习了Git的基本概念，如版本控制、仓库、分支等。

- 掌握了Git的常用命令，如`git init`、`git add`、`git commit`、`git branch`、`git merge`等。
- 了解了Git的工作原理和常见的工作流程，如代码合并、引用等。

2. Markdown基础：

- 学习了Markdown的基本语法，如标题、列表、超链接等。
- 掌握了在Markdown中编辑文档的方法，如使用各种标记符号、添加链接等。
- 了解了Markdown的优点和应用场景，如编写README文档、博客文章等。

1. 编程工具的使用：

- 学习了使用VSCode进行代码编辑和文档编写。
- 掌握了VSCode的基本功能和常用插件的安装与使用。

-了解了GitHub这一常用的Git客户端。

- 了解了VSCode的优点和应用场景，如编写代码、编写文档等。

总而言之，本次实验主要学习了Git的基础知识和Markdown的基本语法，通过实践掌握了使用Git进行版本控制和使用Markdown进行文档编辑的能力。同时，还学习了如何使用VSCode这一常用的编程工具，便利了日后的编程和文档编辑。