Git 教程

零基础入门系列教程

作者: 魏明择

2025 年版

https://weimingze.com

目录

第一章 Git 简介与安装

序

1. Git 简介
2. 安装 Git
3. Git 基本设置

第二章、Git 的基本操作

1. 创建本地 Git 仓库
2. 向 Git 仓库提交更新记录
3.工作区和暂存区操作
4. 忽略文件.gitignore
5. 标签操作

第三章、远程仓库

1. Git 远程仓库的应用场景
2. 远程仓库的基础操作
3. 添加现有工程到远程仓库
4. 远程仓库拉取和推送

第四章、Git 分支

1. 分支简介
2. 分支管理
3. 记录的提交和合并
4. Git 可视化合并

第五章、高级用法

1. HEAD 指针
2. 重置 HEAD 指针
3. Git 版本比较
4. Git 可视化版本比较
5. Git 变基操作
6. 交互式变基

总结

Git 总结

第一章 Git 简介与安装

序

用了近一个月的时间我完成了这篇 **Git 教程**。这是我对这些年的使用心得的总结，也分享给喜欢我和关心我的朋友。

自从 2019 年 Git 改版到 v2.23 之后，还是发生了很大的变化。新版本的 Git 增加了 `git restore` 和 `git switch` 来代替之前的万能的 `git checkout`，这样的变化让 Git 更加好用，思路更加清晰且不容易出错。新版本已经不在推荐使用 `git checkout` 这个命令。因此教程回尽力避开过时或者可能踩坑的某些用法。

本教程的目标是让不熟悉使用版本控制的软件开发者或文档编写者快速入门。能够使用 Git 进行版本控制提高自己的工作效率。

当你读懂了本教程，你就可以使用 Git 进行对您的项目开发或文档编写进行版本控制了。如果你需要更深入的了解 Git ，请查看Git 官网来了解更过的功能。

Git 的官网：

https://git-scm.com/

版权声明

1. Git 简介

Git 是由 Linux 创始人 Linus Torvalds 于 2005 年创建，最初是为了更好地管理 Linux 内核的开发，后来被广泛应用于软件开发领域。Git 不仅可以用于软件项目的版本控制，它还可以用于对任何类型的文件进行版本控制。如：财务报表，文章修改等。

版本控制的概念

先说一下版本控制。

我们在工作中经常改写一些文档或程序代码文件。当有多次修改该文件后，突然发现我们原来认为不重要的东西被我们删除了。如何找回来了呢？有经验的开发人员会在改写文件前以日期为后缀对之前的文件进行备份，以备不时之需。当有一天真的出现问题后想回到之前的某个版本时。你发现之前备份了上百个文件夹，占用了大量磁盘空间的同时，查找问题的难度也大大增加。解决这一问题的方法就是使用**版本控制**软件，它可以方便我们管理修改之前的代码。对修改的内容做比较，也可以快速回溯到之前的某个版本在进行重新修改，建立不同分支等操作。同时在多终端用户合作开发时也方便代码的相互传输。

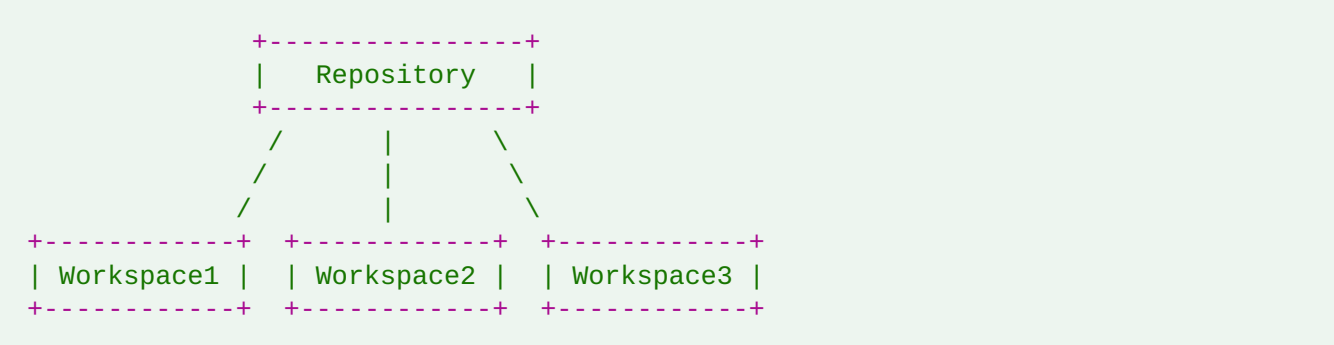
有了版本控制软件，当我们的文件或软件项目出现问题时，你就可以将你的工程文件回溯到之前的某个时间点的状态，你可以比较工程或某个文件的变化细节，查出最后是谁修改了哪个地方，从而找出问题出现的原因。使用版本控制系统通常还意味着，就算你一通乱来把整个项目中的文件删删改改，你也照样可以轻松恢复到原先的样子。 但额外增加的工作量却微乎其微。

目前比较主流的版本控制软件有两种：

1. SVN（Subversion）
2. Git

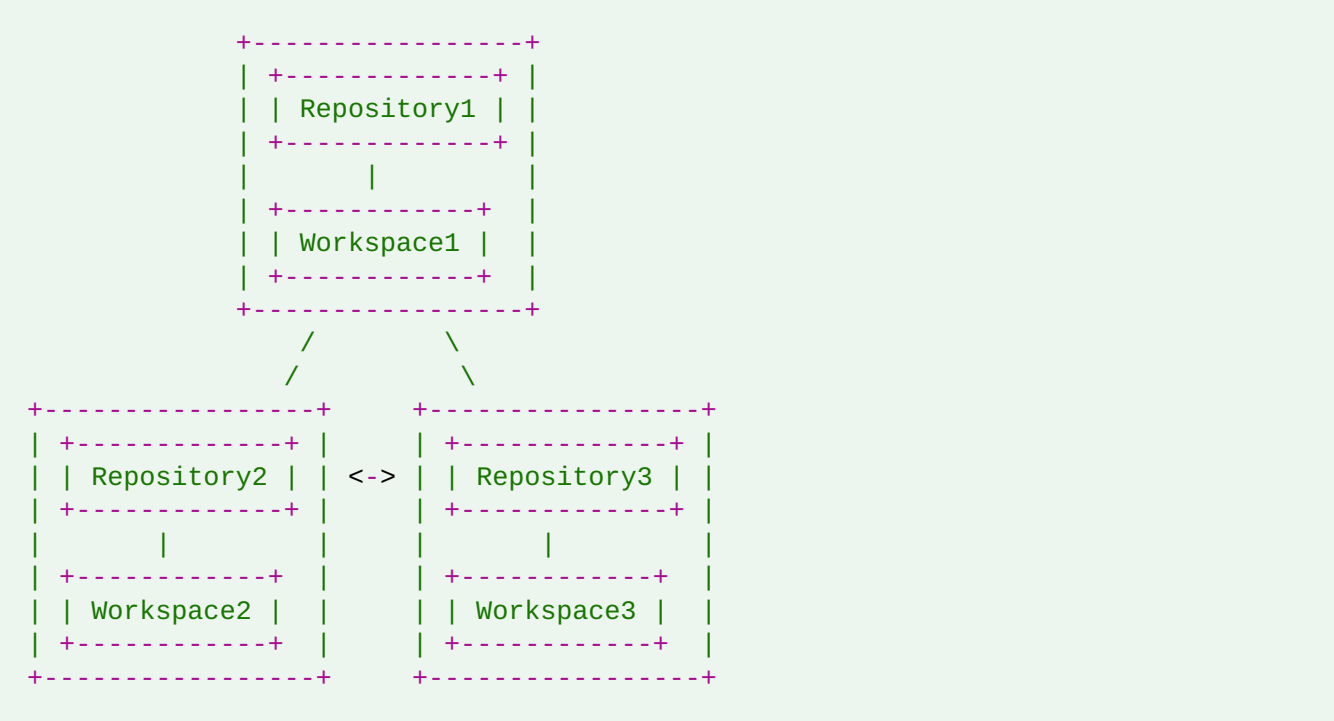
其中 SVN 是**集中式版本控制系统**，即保存版本信息的仓库（Repository）在某一台主机上，所有用户在修改代码时都需要先从仓库拉取（poll）最新文件到本地工作区（Workspace，存放将要编辑的文件的文件夹），待文件修改完毕后提交（Commit）到仓库。这样同一仓库的其它用户在需要更新代码时，只需要重新从仓库拉取最新版本到本地工作区即可。SVN 是比较简单高效的版本控制系统，它的的缺点是在不联网的本地工作区就无法完成版本的提交等工作。后来发展出了 Git 这种 **分布式版本控制系统**。Git 使用分布式的方式来存储仓库，每一用户都可以从远程仓库克隆（Clone）一份完整的仓库形成本地仓库。同时为仓库创建自己的本地工作区。Git 的好处是可以随时提交自己的修改到本地仓库。也可以快速在本地仓库创建分支。工作完成后我们可以把代码提交到本地仓库，如果需要，还可以将本地仓库的内容推送（push）到远程仓库。实现多人合作开发。

SVN 的集中式架构



在使用 SVN 时，每个开发者都可以有自己的工作区，在开发完成后将本地工作区的文件提交到仓库，其它工作区就可以获取仓库中的最新代码了。当然其它工作区也可以将自己的文件提交到仓库中。

Git 的分布式架构



可见 Git 是由多个仓库组成的。仓库用于存储当前工作区的版本信息。同时可以将本地仓库的某个分支合并到其它仓库中。

相比 SVN、Git 是比较复杂的版本控制系统。本教程将带你快速了解和使用 Git，如果你打算深入了解 Git，请查看 Git 的官方网站：https://git-scm.com/ 获取更多的信息。

专业术语：

- **本地仓库（Local Repository）**：用来存储所有文件、版本信息、历史记录、分支信息的数据结构。
- **远程仓库（Remote Repository）**：同本地仓库一样，它是存储于服务器或云端的仓库，可以用来团队协作的中心仓库。
- **工作区（Workspace）**：也称作工作空间或工作文件夹，是在你本地查看和编辑受跟踪文件的文件夹。

实验：

口述 SVN 和 Git 的区别。

2. 安装 Git

Git 是一套跨平台的软件系统，它可以运行在 Windows、Mac OS 和 Linux 系统中。Git 有多种使用方式。原生的 Git 系统有很多个命令组成，你可以使用这套命令来操作 Git 对你的文件进行管理。当然你也可以使用图形用户接口（GUI）的模式使用 Git 的各种功能。如使用 TortoiseGit（仅Windows 可用）通过图形用户界面的方式来代替输入命令来操作 Git，也可以在集成开发环境中，如：Visual Studio Code、PyCharm、XCode 等开发环境通过接口菜单来操作 Git。当你使用图形化的工具来对 Git 进行操作时，这些图形化的工具会间接的调用 Git 相关的命令来完成相关的操作。因此在使用可视化图形工具前你也可能需要安装 Git。

本课程只使用命令对 Git 的原理和操作等进行讲解。了解原理后你就可以随意的通过图形用户界面对 Git 进行操作了。

在不同的操作系统上有不同的命令行工具，本课程推荐使用如下工具进行操作：

1. Mac OS/Linux 用户使用终端（Terminal）
2. Windows 用户使用 **Git Bash**、**命令提示符**（Command Prompt）或 **PowerShell**（推荐）。

当你掌握了本教程中基础理论，你再转去使用图形用户接口的方式使用 Git 就游刃有余了。

下载和安装 Git

你可以在 Git 的官方网站下载最新的版本的 Git 安装包并进行安装。

官方下载页面地址如下：

https://git-scm.com/install/

一、Windows 安装 Git

1. 下载 X64 或 ARM 处理器的安装包

下载地址：

- X64 处理器安装包本地下载地址：Git-2.52.0-64-bit.exe
- ARM 处理器安装包本地下载地址：Git-2.52.0-arm64.exe

2. 安装方法

双击安装包，按着安装向导的提示，一步一步的进行安装即可。安装完毕后打开 **Git Bash** 或 **Windows 的命令提示符** 就可以通过 git 命令来管理你的文件了。

二、Mac OS 安装 Git

Mac OS 下可以有多种方法安装 Git，以下任选其一即可。 MacOS 下安装 Git 需要打开终端，在终端中输入如下命令进行安装。

1. 通过 Homebrew 安装 Git

```
% brew install git
```

> 请确保你已经安装了 Homebrew 并可以运行 brew 命令。

2. 通过 MacPorts 安装 Git

```
% sudo port install git
```

> 请确保你已经安装了 MacPorts 并可以运行 port 命令。

3. 通过 Xcode 命令行工具安装 Git

```
% xcode-select --install
```

三、Linux 安装 Git

在 Linux 上安装 Git 需要使用 Linux 的终端，在终端中输入安装命令来进行安装。

由于 Linux 存在众多的发行版版本，不同的 Linux 使用的安装命令也有所不同。如下提供了不同 Linux 发行版的几种安装方法：

Debian/Ubuntu

```
$ sudo apt install git
```

如果你是 Ubuntu 系统，有可能还需要重新更新 apt 源列表，方法如下：

```
$ sudo add-apt-repository ppa:git-core/ppa
$ sudo apt update
$ sudo apt install git
```

Fedora

```
# yum install git # (up to Fedora 21)
# dnf install git # (Fedora 22 and later)
```

Gentoo

```
# emerge --ask --verbose dev-vcs/git
```

Arch Linux

```
# pacman -S git
```

openSUSE

```
# zypper install git
```

Mageia

```
# urpmi git
```

Nix/NixOS

```
# nix-env -i git
```

FreeBSD

```
# pkg install git
```

Solaris 9/10/11 (OpenCSW)

```
# pkgutil -i git
```

Solaris 11 Express, OpenIndiana

```
# pkg install developer/versioning/git
```

OpenBSD

```
# pkg_add git
```

Alpine

```
$ apk add git
```

验证安装是否成功

打开终端（Windows 系统叫命令提示符），然后输入 `git --version`，如果命令已经存在并能够打印出你安装的 Git 版本的信息则说明你的 Git 安装成功了。我在 Ubuntu Linux 下的显示结果如下：

```
weimingze@mzstudio:~$ git --version
git version 2.43.0
```

实验：

在你使用的电脑上安装 Git。

3. Git 基本设置

在初次使用 Git 时需要设置你的身份信息，其中有两个必须要设置的信息：用户名和邮箱。这些信息会在你每次提交文件时自动添加到仓库中。以便于在库中查看这些用户信息。

一、必选设置

设置用户名和邮箱的命令如下：

```
git config --global user.name "你的名字"
git config --global user.email "你的邮箱"
```

> 其中 `--global` 选项是设置当前用户的 git 的全局信息，即以后此用户默认是你设置的名字和邮箱。 如果改成 `--system` 选项则是设置所有用户的用户名和邮箱。`--local` 选项则是设置此仓库的用户名和邮箱信息。具体使用那个用户信息的规则是就近查找，即优先使用 `--local`，然后是 `--global`， 最后才是 `--system`。

示例：

```
git config --global user.name "weimingze"
git config --global user.email "auth@weimingze.com"
```

查看所有设置项：

使用 `git config --list` 命令可以查看你的当前用户的 Git 设置信息。经过上述设置后我的设置信息打印如下：

```
weimingze@mzstudio:~$ git config --list
user.name=weimingze
user.email=auth@weimingze.com
```

以上这些信息会保存在你的用户的主文件夹下的一个名为 `.gitconfig` 的文件中。我的Linux 系统下使用 `cat` 命令查看文件的内容如下：

```
weimingze@mzstudio:~$ cat .gitconfig
[user]
    name = weimingze
    email = auth@weimingze.com
```

如果删除此 `.gitconfig` 文件则 Git 恢复到初始的设置状态。

你可以通过以下命令查看所有的设置以及它们在哪个文件中：

```
$ git config --list --show-origin
```

如在我的电脑上查看上述设置位置的结果：

```
weimingze@mzstudio:~$ git config --list --show-origin
file:/home/weimingze/.gitconfig user.name=weimingze
file:/home/weimingze/.gitconfig user.email=auth@weimingze.com
```

可见我的设置文件是 `/home/weimingze/.gitconfig`。

除了上述必须进行的设置以外。为方便使用，有时我们也可以进行如下可选的设置。

二、可选设置

1）文本编辑器设置

在使用 git 时可以设置默认的文本编辑器，这个默认的编辑器会在提交文件等场景下自动启动来编写相应的文本信息。这些编辑器可以是 `Visual Studio Code`、`notepad`、`vim`、`nano`等。

设置 Visual Studio Code 作为默认的文本编辑器:

```
git config --global core.editor "code --wait"
```

> `code` 是 Visual Studio Code 对应的启动命令，`--wait` 是 code 启动后的阻塞选项。

设置 Vim 作为默认的文本编辑器:

```
git config --global core.editor "vim"
```

2）difftool 比较软件设置

在使用 `git difftool` 命令做版本比较时可以设置可视化的软件来进行内容对比。如：`Visual Studio Code`、`meld`、`vimdiff` 等。

如：

设置 vimdiff 作为默认的比较软件:

```
git config --global diff.tool vimdiff
```

设置 Meld 作为默认的比较软件:

```
# 1. 配置 meld 命令的启动方式和参数
git config --global difftool.meld.cmd 'meld "$LOCAL" "$REMOTE"'

# 2. 配置默认的 difftool 比较工具是 `meld`
git config --global diff.tool meld
```

`meld` 软件是开源免费的文件和文件夹比较软件，也可以用来做版本合并，它能够运行在 `Windows`、`Mac OS` 和 `Linux` 操作系统中。

`meld` 官方下载地址：http://meldmerge.org/

3）mergetool 合并软件设置

在使用 `git mergetool` 命令做版本合并时可以设置可视化的软件来进行可视化合并。如：`Visual Studio Code`、`meld`、`vimdiff` 等。

如：

设置 vimdiff 作为默认的合并软件:

```
git config --global merge.tool vimdiff
```

设置 Meld 作为默认的比较软件:

```
git config --global merge.tool meld
```

实验

查看你电脑上 Git 的默认设置。

第二章、Git 的基本操作

1. 创建本地 Git 仓库

我们在使用 Git 时需要先创建本地仓库，我们使用本地仓库来存储编写的文件、版本信息，历史记录、分支信息等。本地仓库会自带当前仓库的工作区。我们在工作区内编写代码，然后再提交到本地仓库。这样就完成了文件的管理和版本控制。

创建本地仓库的方法有两种：

1. 使用 `git init` 命令将一个本地的文件夹初始化成为一个最原始的空的本地仓库，它也可以将一个已经存在的仓库重新初始化成为空的仓库。
2. 使用 `git clone` 命令从一个存在的一个远程仓库克隆成为本地仓库，此本地仓库包含远程仓库的全部信息（包括存储的文件、版本信息、历史记录等）。我们不但能够查看过去的修改历史记录，也能够在此基础上进行改写，然后提交到远处仓库。

1、创建空的 Git 仓库

如果你已经在编写一个项目文件夹，其中的所有文件都没有使用任何的版本控制，现在你想使用 Git 对其进行版本控制，那么你可以使用 `cd` 命令先进入这个文件夹，然后使用 `git init` 命令对其初始化成为一个空的 Git 仓库。比如在我的用户主文件夹下，将名称为 `my_project` 的项目文件夹初始化成为 Git 仓库，在不同的操作系统下用法如下：

Linux 系统

```
$ cd /home/weimingze/my_project
```

Mac OS 系统

```
$ cd /Users/weimingze/my_project
```

Windows 系统（建议使用PowerShell）

```
C:\> cd C:\Users\weimingze\my_project
```

进入 `my_project` 文件夹后运行 `git init` 命令，此命令运行成功后，此 `my_project` 文件夹就成为了一个 Git 仓库。

```
$ git init
```

当运行完 `git init` 后，`my_project` 文件夹内会新创建一个 `.git` 的文件夹。这个 `.git` 文件夹就是当前工程 `my_project` 的**本地 Git 仓库**，其中包含仓库所需要的全部文件。而当前的文件夹 `my_project` 则用于存放构成该项目的文件，这里称之为**工作区（workspace）**（不包括 `.git` 文件夹）。工作区是树形文件结构，因此我们又把工作区叫做**工作树（worktree）**。只有工作区内的文件才能纳入 Git 版本控制。

在 Mac OS 和 Linux 系统中以点（`.`）开头的文件和文件夹是隐藏文件夹。默认不会在终端或文件浏览器中显示。如果你需要查看隐藏文件，你可以使用 命令 `ls -a` 进行查看隐藏文件

`my_project` 本地仓库的内部结构如下：

```
my_project/
└── .git
    ├── HEAD
    ├── branches
    ├── config
    ├── description
    ├── hooks
    │   ├── applypatch-msg.sample
    │   ├── commit-msg.sample
    │   ├── fsmonitor-watchman.sample
    │   ├── post-update.sample
    │   ├── pre-applypatch.sample
    │   ├── pre-commit.sample
    │   ├── pre-merge-commit.sample
    │   ├── pre-push.sample
    │   ├── pre-rebase.sample
    │   ├── pre-receive.sample
    │   ├── prepare-commit-msg.sample
    │   ├── push-to-checkout.sample
    │   ├── sendemail-validate.sample
    │   └── update.sample
    ├── info
    │   └── exclude
    ├── objects
    │   ├── info
    │   └── pack
    └── refs
        ├── heads
        └── tags
```

通常我们不需要手动修改 `.git` 文件夹内的内容。要操作此 Git 仓库需要使用 Git 命令。

这样，一个空的 Git 仓库创建完毕。我们接下来就可以将文件加入到此仓库中进行版本控制了。

2、克隆一个已经存在的远程仓库

如果你编写的项目是一个远程多人合作开发的项目。而项目的主仓库是一个远程仓库。如果你需要获取到该项目，并在原有代码的基础上进行修改。你可以使用 `git clone` 命令来为远程仓库创建一个本地的副本，成为本地仓库。你可以在本地仓库为该项目贡献你自己的代码。在需要时，你还可以将本地仓库的内容推送到远程仓库，实现你对此项目的贡献。

`git clone` 是为远程仓库创建一个近乎完整的副本，即便远程仓库由于磁盘损坏等导致远程仓库丢失，你也可以使用此本地仓库将远程仓库恢复为 `git clone` 时的状态。

`git clone` 命令的用法如下：

```
git clone 远程仓库的URL地址
```

如果我要获取我之前用 python 写的一个 2048 的小游戏。这个游戏放在码云平台上。远程仓库的URL地址是 `https://gitee.com/weimz/py2048.git`。使用如下命令就可以完成此开源项目的复制。

```
git clone https://gitee.com/weimz/py2048.git
```

在我的 Ubuntu Linux 上执行效果如下:

```
weimingze@mzstudio:~$ git clone https://gitee.com/weimz/py2048.git
Cloning into 'py2048'...
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 19 (delta 1), reused 0 (delta 0), pack-reused 10 (from 1)
Receiving objects: 100% (19/19), 101.91 KiB | 1.82 MiB/s, done.
Resolving deltas: 100% (1/1), done.
weimingze@mzstudio:~$ cd py2048
weimingze@mzstudio:~/py2048$ tree .
.
├── 2048game.py
├── README.md
└── images
    └── demo.png

2 directories, 3 files
weimingze@mzstudio:~/py2048$ tree -a
.
├── .git
│   ├── HEAD
│   ├── branches
│   ├── config
│   ├── description
│   ├── hooks
│   │   ├── applypatch-msg.sample
│   │   ├── commit-msg.sample
```

```
│   │   ├── fsmonitor-watchman.sample
│   │   ├── post-update.sample
│   │   ├── pre-applypatch.sample
│   │   ├── pre-commit.sample
│   │   ├── pre-merge-commit.sample
│   │   ├── pre-push.sample
│   │   ├── pre-rebase.sample
│   │   ├── pre-receive.sample
│   │   ├── prepare-commit-msg.sample
│   │   ├── push-to-checkout.sample
│   │   ├── sendemail-validate.sample
│   │   └── update.sample
│   ├── index
│   ├── info
│   │   └── exclude
│   ├── logs
│   │   ├── HEAD
│   │   └── refs
│   │       ├── heads
│   │       │   └── master
│   │       └── remotes
│   │           └── origin
│   │               └── HEAD
│   ├── objects
│   │   ├── info
│   │   └── pack
│   │       ├── pack-bd32d13ce9569d66d6046aea24ac1d72e1c2b0b6.idx
│   │       ├── pack-bd32d13ce9569d66d6046aea24ac1d72e1c2b0b6.pack
│   │       └── pack-bd32d13ce9569d66d6046aea24ac1d72e1c2b0b6.rev
│   ├── packed-refs
│   └── refs
│       ├── heads
│       │   └── master
│       ├── remotes
│       │   └── origin
│       │       └── HEAD
│       └── tags
├── 2048game.py
├── README.md
└── images
    └── demo.png

19 directories, 31 files
weimingze@mzstudio:~/py2048$
```

可见，我将远程仓库 `https://gitee.com/weimz/py2048.git` 克隆到本地文件夹 `py2048` 内。在 `py2048` 文件夹内也有一个 `.git` 文件夹，这个文件夹就是存放 Git 本地仓库的所需文件的文件夹。这个名为 `py2048` 的文件夹就是这个项目的工作区。其中 `2048game.py`、`README.md` 和 `images` 文件夹都是工作区中的文件，即最后提交的工作区中的文件内容。

其中 `2048game.py` 是游戏的运行文件。你如果已经安装了 Python 的解释执行器，你可以运行 `python3 2048game.py` 或 `python 2048game.py` 就可以运行这个游戏了

实验：

1. 从 github 上克隆远程仓库 `https://github.com/sqlite/sqlite`（因为服务器在国外，可能克隆失败）。
2. 从 码云 上克隆魏明择写过的电子书的远程仓库 `https://gitee.com/weimz/books.git`。

2. 向 Git 仓库提交更新记录

上节课我们创建了 `my_project` 这个本地仓库，但并没有将任何文件放入到仓库中。这节课我们来学习如何使用 Git 本地仓库对工作区中的文件进行管理。

首先我们在工作区 `my_project` 内创建一个文件 `README.md` 写入一行 `# Git 教程`。在我的 Ubuntu Linux 系统的电脑上操作如下:

```
weimingze@mzstudio:~/my_project$ echo -e -n "# Git 教程\r\n" > README.md
weimingze@mzstudio:~/my_project$ cat README.md
# Git 教程
```

> Windows 用户请尝试使用 PowerShell 进行上述操作或使用记事本手动创建 `README.md` 文件。

我们上述创建的文件 `README.md` 只是在工作区中存在，并没有纳入到该 Git 本地仓库中进行管理。下面我们要做的就是将工作区内新添加的 `README.md` 文件形成一个**快照**提交到 Git 本地仓库中并形成一条提交**记录**，以便在需要时取出当前 `README.md` 文件的快照。在使用 Git 提交记录时会在本地仓库内创建一个**提交对象**，此提交对象用来记录本次提交的内容（当前工作区的变化信息）。

专业术语：

- **快照（Snapshot）：** 是指当前工作区被跟踪的所有文件当前内容、修改时间等状态的保存。比如在编写文件时，你将当前内容保存成文件，你保存的文件就是你当前编写文件时状态的一个快照。
- **提交（Commit）：** 是将改写工作区后，将工作区变化的快照放入仓库的过程。
- **记录（Record）：** 将本次提交的修改内容、提交人，日期等信息形成可保存的信息，在仓库中形成的历史存档。

- **提交对象（Commit Object）**：在仓库中用来存储本次提交具体信息的存储对象。每一个提交对象，都会有一个不会重复的哈希值，用于表示此提交对象，这个哈希值也称为**提交ID** 或**提交哈希**。
  - 提交哈希值示例: `8cb34d10643636d04286624adaa55b6a3967ab9e`。

要了解 Git，你需要先了解 **工作区** 中的每个文件的状态有哪些。

工作区中每个文件都有两种状态中的一个：

1. 已跟踪状态。
2. 未跟踪状态。

已跟踪文件（Tracked files）是指 Git 知道这个文件，并此文件已经在本地仓库中形成**快照**或在**暂存区**中将放入到本地仓库中进行管理的文件。这样的文件是**已跟踪状态**。

未跟踪文件（Untracked files） 是指没有纳入 Git 本地仓库管理的文件。也不在**暂存区**中管理的文件。这样的文件是**未跟踪状态**。

上述我们创建的文件 `README.md` 此时就是**未跟踪状态**。如果我们要将其纳入版本控制中，我们需要将其变成**已跟踪状态**。

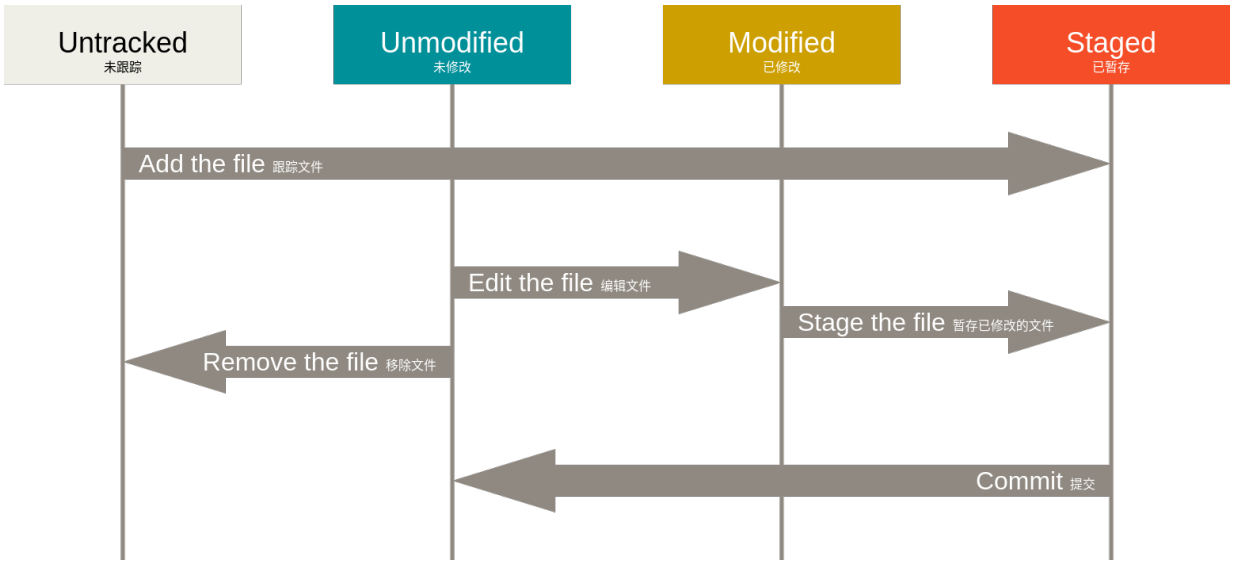
已跟踪状态 又分为三种状态:

1. 未修改状态（Unmodified）。
   - 当前工作区中的文件和本地仓库中的文件内容完全一致。当你首次克隆（复制）一个远程仓库时，你本地工作区中的所有的文件都是已跟踪且处于未修改状态。
2. 已修改状态（Modified）。
   - 是指工作区中的文件内容和暂存区中文件内容不一致或暂存区文件内容和本地仓库中文件内容不一致的文件。
3. 已暂存状态（Staged）。
   - 工作区中文件已经修改，并放入到暂存区中，工作区和暂存区中文件内容一致，但暂存区和本地仓库的内容不一致的文件。

专业术语：

- **暂存区（Staging Area）**：是存放准备提交到本地仓库的文件的 **快照**，是准备放入仓库前的文件的时间节点的映射。你可以理解成即将运往本地仓库数据的**小货车**。

它们的关系如下图所示:



我们刚才创建的 `README.md` 就是未跟踪文件。如果我们要将其纳入版本控制我们需要将其变为 **已跟踪文件**。

查看文件状态

此时我们可以使用 `git status` 查看当前工作区的文件状态。

`git status` 命令

`git status` 是用于查看工作区和暂存区的当前状态。

我的电脑上的查询结果如下：

```
weimingze@mzstudio:~/my_project$ git log
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.md

nothing added to commit but untracked files present (use "git add" to track)
```

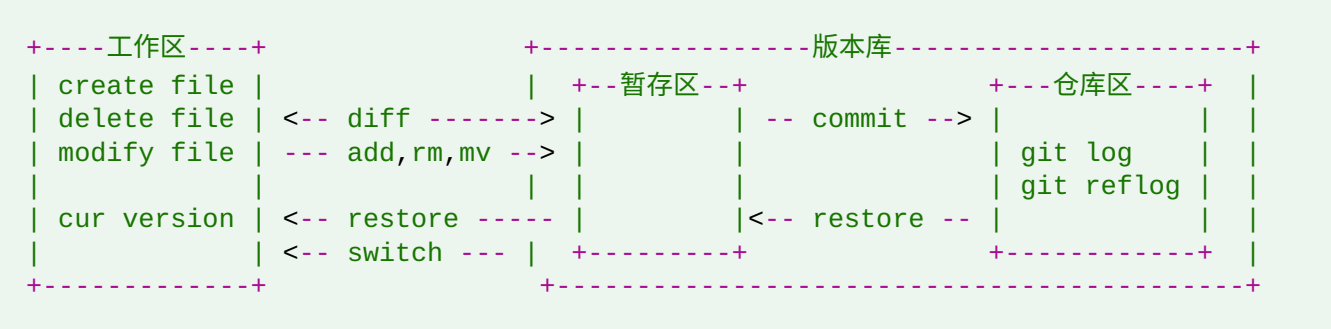
我们可以看出文件 `README.md` 是 **未跟踪文件**（`Untracked files`），并提示使用 `git add` 命令可以将其变为 **已跟踪状态**。

现在我们已经完成整个工作区文件的修改。我们准备将当前编写的 `README.md` 文件的状态做一个快照保存在本地仓库，完成一个备份操作。此时我们还要理解工作区、暂存区、本地仓库之间的关系。

一个本地仓库有三个部分构成：

1. **工作区（Workspace）**：是存放当前代码的区域，是一个树形结构，也称为工作树（`working tree`），你可以理解成你工作的办公室或车间。
2. **暂存区：**你可以理解成存放运往本地仓库的文件的小货车，你要你发个`git commit`命令，这些数据就会发往本地仓库。
3. **本地仓库：**用于存放版本信息的仓库。

它们的关系如下：



`git add` 命令

作用： 将一个或多个文件、文件夹、符号链接等加入**暂存区**，并变为**已跟踪状态**，准备提交。

命令格式

```
git add 文件或文件夹1 文件或文件夹2 ...
```

> 文件可以使用通配符，如`*.c`表示所有以`.c`结尾的文件。

我们将之前`README.md`加入暂存区，执行结果如下：

```
weimingze@mzstudio:~/my_project$ git add README.md
weimingze@mzstudio:~/my_project$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```


现在 `README.md` 的文件已经加入到暂存区，变成已跟踪状态。提示为：新加入的文件（`new file`）。下一步我们就可以使用 `git commit` 命令将其提交到仓库了。

`git commit` 命令

`作用： 用于将暂存区内的数据提交到本地仓库，并形成版本信息。

git commit 常用的选项

选项	说明
`-m "版本说明信息"`	提交时附带当前提交的说明，如果不携带此参数，则会启动默认的文本编辑器，待编辑完成后再提交。
`--amend`	将**暂存区**的内容提交到上一个版本（修订），不形成新的版本信息。

示例：

使用 `git commit` 命令将当前暂存区中的文件 `README.md` 提交到本地仓库，并形成提交对象。

```
weimingze@mzstudio:~/my_project$ git commit -m "魏明择在当前项目中添加了 README.md文件"
[master (root-commit) 8cb34d1] 魏明择在当前项目中添加了 README.md文件
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

如出现上述提示，说明你已将完成了提交。其中 `master` 是你本地提交操作的分支（后面会讲）。其中提示中 `8cb34d1` 是本次提交在本地仓库中产生的提交对象的ID，它代表本次提交在本地仓库内产生的提交对象的唯一标识（后面回溯历史版本时会用到）。

提交对象ID

每次提交到本地仓库都会在本地仓库产生一个提交对象，这个提交对象记录本地提交的所有信息。每个提交对象有一个唯一的标识，即**提交对象ID**，如：`8cb34d10643636d04286624adaa55b6a3967ab9e`，这个哈希值就是本地提交对象的 ID，这个ID相当于这个提交对象的身份证号，并在整个仓库中不会重复。通常我们只用其中的前几位，如 `8cb34d1` 是这个哈希值的前 7 个数字，这几位只要不和其它的 ID 重复，它就可以代表这个提交对象了。

完成了本次提交以后我们在使用 `git status` 命令查看工作区和暂存区的状态如下：

```
weimingze@mzstudio:~/my_project$ git status
On branch master
nothing to commit, working tree clean
```

上述信息指出当前你正处于 `master` 分支（默认分支，后面会讲）上，目前没有任何数据需要提交，工作区是干净的。以上提示信息意味着当前已跟踪的所有文件都没有被修改，Git 也没有发现任何未跟踪的文件。即当前工作区所在的文件夹是一个干净的文件夹。否则这些文件会在运行 `git status` 命令时被列出。

如果你在提交过程中提示如下信息，则说明你需要设置你的用户信息。

```
weimingze@mzstudio:~/my_project$ git commit -m "魏明择在当前项目中添加了 README.md文件"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'weimingze@mzstudio.(none)')
```

你需要使用上述命令来设置当前用户信息，以便 git 记录提交人是谁。设置方法详见《Git 基本设置》这一小节

时隔一天，我又改写了上述 `my_project` 项目。我在 `README.md` 中添加了一行内容 `作者：魏明择`，内容如下：

文件 `README.md` 中的内容：

```
# Git 教程
作者：魏明择
```

同时我有添加了一个文件 `website.txt` ，其中的内容只是记录了一行我网站的网址，其内容如下：

```
https://weimingze.com
```


使用 `git status` 命令查看当前的状态如下:

```
weimingze@mzstudio:~/my_project$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    website.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

可见现在项目中有一个未跟踪的文件 `website.txt` 和 已跟踪并已修改的的文件 `README.md`。

现在我们将两个文件使用 `git add README.md website.txt` 命令重新添加到暂存区并查看状态如下:

```
weimingze@mzstudio:~/my_project$ git add README.md website.txt
weimingze@mzstudio:~/my_project$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.md
    new file:   website.txt
```

可见现在两个文件 `README.md` 和 `website.txt` 都成为了已跟踪状态，并随时可以提交。接下来我们使用 `git commit` 命令进行提交。执行结果如下:

```
weimingze@mzstudio:~/my_project$ git commit -m "魏明择在当前项目中修改了 README.md
文件，并添加了 website.txt 文件"
[master 5dcccb0] 魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件
 2 files changed, 2 insertions(+)
 create mode 100644 website.txt
weimingze@mzstudio:~/my_project$ git status
On branch master
nothing to commit, working tree clean
```

本次提交的哈希值是 `5dcccb0`，本地提交的分支是 `master` 分支。提交完毕后当前工作区是干净的（没有任何修改没有提交）。

上述代码，我经历两天工作，完成了两个版本的提交。我们怎样才能查看我们提交的信息呢？下面我们学习 `git log` 命令。

`git log` 命令

作用： 用来查看当前分支的历史提交记录，以便提取过去存入仓库中的某个版本。

`git log` 常用选项

选项	说明
`--pretty=oneline`	每个提交日志以一行的形式显示，前面显示完整的提交哈希值，后面显示提交信息。
`--oneline`	以一行的形式显示，前面显示简短的提交哈希值，后面显示提交信息。
`-n`	n 为整数，表示最近的 n 次提交信息。
`--abbrev-commit`	使用短哈希值代替完整的 40 位提交哈希（默认 7 位）
`--graph`	在日志左侧以 ASCII 字符绘制提交历史图。
`--all`	显示所有分支的历史提交。

我们已经完成了两次提交，现在我们使用 `git log` 命令来查看我们过去有几次提交信息。

```
weimingze@mzstudio:~/my_project$ git log
commit 5dcccb02bd6e6593cdec017f9af4f910c0c01c3a (HEAD -> master)
Author: weimingze <auth@weimingze.com>
Date:   Sun Jan 11 18:50:28 2026 +0800

    魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件

commit 8cb34d10643636d04286624adaa55b6a3967ab9e
Author: weimingze <auth@weimingze.com>
Date:   Sat Jan 10 17:15:05 2026 +0800

    魏明择在当前项目中添加了 README.md文件
```

如果你的提交次数比较多，你想以最简单的方式来查看提交日志，可以使用 `--pretty=oneline` 选项简化显示内容。如下：

```
weimingze@mzstudio:~/my_project$ git log --pretty=oneline
5dcccb02bd6e6593cdec017f9af4f910c0c01c3a (HEAD, master) 魏明择在当前项目中修改了 REA
DME.md 文件，并添加了 website.txt 文件
8cb34d10643636d04286624adaa55b6a3967ab9e 魏明择在当前项目中添加了 README.md文件
```

使用 `--oneline` 选项显示的结果如下:


```
weimingze@mzstudio:~/my_project$ git log --oneline
5dcccb0 (HEAD, master) 魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件
8cb34d1 魏明择在当前项目中添加了 README.md文件
```

使用 `--graph` 选项显示的结果如下:

```
weimingze@mzstudio:~/my_project$ git log --graph
* commit 5dcccb02bd6e6593cdec017f9af4f910c0c01c3a (HEAD, master)
| Author: weimingze <auth@weimingze.com>
| Date:   Sun Jan 11 18:50:28 2026 +0800
|
|     魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件
|
* commit 8cb34d10643636d04286624adaa55b6a3967ab9e
  Author: weimingze <auth@weimingze.com>
  Date:   Sat Jan 10 17:15:05 2026 +0800

      魏明择在当前项目中添加了 README.md文件
```

使用 `--abbrev-commit` 选项显示的结果如下:

```
weimingze@mzstudio:~/my_project$ git log --abbrev-commit
commit 5dcccb0 (HEAD, master)
Author: weimingze <auth@weimingze.com>
Date:   Sun Jan 11 18:50:28 2026 +0800

    魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件

commit 8cb34d1
Author: weimingze <auth@weimingze.com>
Date:   Sat Jan 10 17:15:05 2026 +0800

    魏明择在当前项目中添加了 README.md文件
```

上述信息显示本地仓库 `my_project` 两次提交。最上面的是最近的提交，完整的哈希值是 `5dcccb02bd6e6593cdec017f9af4f910c0c01c3a`。下面的是稍远时间点的提交，哈希值是 `8cb34d10643636d04286624adaa55b6a3967ab9e`。上述信息还显示了每次提交的提交人、提交时间、提交备注的文字信息。其中 `(HEAD -> master)` 指示我们当前操作的位置指针 `HEAD` 是在 `master` 分支中的 `5dcccb02bd6e6593cdec017f9af4f910c0c01c3a` 提交对象的位置。即当前所处于的位置是在第二次提交中（即最新的一次提交）。

经历了两次提交，目前我们 本地仓库形成了如下的结构：

```
                HEAD
                 |
              .--------.
```

```
               | master |
               `--------`
                   |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
第一次提交      第二次提交
```

其中 `8cb34d1` 是第一次提交生成的提交对象，本次提交添加了一个文件 `README.md`。`5dcccb0` 是第二次的提交对象，本次提交增加了一个文件 `website.txt` 并修改了文件 `README.md` 的内容。

附加信息：

以上两次提交都在主分支 `master`（默认分支）上进行。`master` 分支的分支指针 `master` 指向了第二次提交创建的提交对象 `5dcccb0`。`HEAD` 是当前位置指针，它指向了分支指针 `master`，表示当前分支是 `master` 分支。

关于 `HEAD` 当前位置指针和分支指针后面会讲。

> Git 命令一般有很多的选项，可以实现不同的功能。如 `git log` 命令可以使用 `git log --help` 来查看其所有选项和使用手册。

点击这里下载上述示例 my_project 本地仓库的代码

本节小结：

1. `git status` 命令用于查看当前工作区的状态。
2. `git add` 命令用于将文件加入到暂存区。为提交做准备。
3. `git commit` 命令用于将暂存区的修改记录提交到本地仓库并创建提交记录。
4. `git log` 命令用于显示提交日志。

实验：

再次修改 `README.md` 文件中的内容，然后提交到本地仓库。

3.工作区和暂存区操作

在 Git 中，工作区是用来存储当前编辑文件的文件夹，暂存区是存放将要提交到仓库的文件的快照。以下介绍常用于工作区和暂存区的命令。

命令	说明
`git add`	添加文件内容至索引(暂存区) 文件转为已跟踪状态。
`git mv`	移动或重命名一个已跟踪的文件、文件夹或符号链接
`git rm`	从工作区和索引(暂存区)中删除文件，并记录删除操作，文件转为未跟踪状态。
`git restore`	使用工作区或本地仓库文件恢复工作区和暂存区的修改。

`git add` 命令

作用：添加文件内容至暂存区，准备提交到仓库中，加入暂存区的文件一定是已跟踪状态。

命令格式

```
git add 文件或文件夹1 文件或文件夹2 ...
```

示例:

```
# 将文件 a.txt，b.txt 记录到暂存区
git add a.txt b.txt

# 将所有文件（不包含隐藏文件）记录到暂存区
git add *
```

注意

以上命令记录到暂存区是对当前文件状态的记录，后续如果在工作区对文件进行修改，则暂存区的内容不会改变。

`git mv` 命令

作用：移动或重命名一个已跟踪的文件、文件夹或符号链接。

命令格式

```
git mv 文件或文件夹1 文件或文件夹2
```

示例:

```
# 将已跟踪的文件a.txt 更名为b.txt，同时放入暂存区。
git mv a.txt b.txt

# 将已跟踪的文件c.txt 移动到 mydir 文件夹，同时放入暂存区。
git mv c ./mydir/
```

`git rm` 命令

作用： 从工作区和索引(暂存区)中删除文件，并记录删除操作，文件转为未跟踪状态。

命令格式

```
# 在暂存区里面的添加删除文件记录，同时删除文件
git rm [选项] 文件或文件夹1 文件或文件夹2 ...

# 在暂存区里面的添加删除文件记录，但不会删除文件
git rm --cached 文件或文件夹1 文件或文件夹2 ...
```

git rm 常用选项

选项	说明
`--cached`	在暂存区里面的添加删除文件记录，但不会删除工作区文件。

示例:

```
# 将已跟踪文件 a.txt 和 b.txt 从暂存区删除，但文件a.txt、b.txt 依旧保存在工作区
git rm --cached  a.txt b.txt
# 将已跟踪文件 c.txt 从暂存区删除，同时删除工作区里面的 c.txt
git rm c.txt
```

`git restore` 命令

作用：使用工作区或本地仓库文件恢复工作区和暂存区的修改。

命令格式

```
git restore [选项] [--] 文件或文件夹1 文件或文件夹2 ...
```

说明:

- `--` 是可选项，它是分隔符号，`--` 前面写选项，后面写文件路径。
- 使用 `.` 可以代表当前文件夹即这个工作区。

常用选项

选项	说明
`--source=<提交对象>`	恢复当前工作区的内容为某个提交时的内容。
`--staged`	撤销暂存区文件记录。
`--worktree <文件路径>`	恢复当前工作区的某个路径，`.` 代表当前工作区。

在上一节的 `my_project` 项目 中已经有两次提交。提交日志如下：

```
weimingze@mzstudio:~/my_project$ git log --oneline
5dcccb0 (HEAD, master) 魏明择在当前项目中修改了 README.md 文件，并添加了website.txt文件
8cb34d1 魏明择在当前项目中添加了 README.md 文件
```

我们可以使用 `git restore` 的`--source` 选项将工作区的一个文件或整体恢复成某次提交后的状态。如：

```
# 恢复第一次提交全部文件到当前工作区，参数 . 代表整个工作文件夹（工作树）。
git restore --source=8cb34d1 .

# 恢复第一次提交的 README.md 文件到当前工作区，其它工作区的文件不变
git restore --source=8cb34d1 README.md
```

示例:

```
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
weimingze@mzstudio:~/my_project$ git restore --source=8cb34d1 .
weimingze@mzstudio:~/my_project$ ls
README.md
weimingze@mzstudio:~/my_project$ git restore .
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
weimingze@mzstudio:~/my_project$ git restore --source=8cb34d1 README.md
weimingze@mzstudio:~/my_project$ cat README.md
# Git 教程
weimingze@mzstudio:~/my_project$ cat website.txt
https://weimingze.com
```


从上述示例中 `git restore --source=8cb34d1 .` 整个工作区还原成了 第一次提交时的状态（只有一个文件 `README.md`，当执行 `git restore .` 工作区用恢复了第二次提交时的状态（两个文件 `README.md` 和 `website.txt`。当执行 `git restore --source=8cb34d1 README.md`命令后，只有 `README.md` 恢复成了第一次提交时的内容。而 `website.txt` 文件的内容没有变化，也没有被删除。

练习：

1. 修改 `my_project` 中的 `README.md` 文件。使用 `git status` 查看修改后的状态。
2. 将 `README.md` 文件添加到暂存区。然后删除工作区的 `README.md` 文件。
3. 使用暂存区来恢复你删除的 `README.md` 文件，查看文件的内容是你修改后的文件内容还是没有修改的内容?
4. 删除暂存区添加 `README.md` 文件的记录。
5. 删除工作区 `README.md` 文件。
6. 使用 `git restore` 恢复 `README.md` 文件。并查看文件的内容是修改前的内容还是没有修改前的内容。
7. 使用 `git mv` 命令 修改 `README.md` 文件名为 `README.txt`，然后使用 `git status` 查看其状态。
8. 想办法通过本节课的四个 `git` 命令将 `my_project` 恢复成干净的状态（没有修改前的状态），包括清空暂存区。

> 上述操作请不要使用 `git commit` 命令进行提交。

4. 忽略文件.gitignore

在项目开发过程中，有些文件是项目测试时自动生成的文件，这些文件没有必要加入到仓库中进行版本控制。如：日志文件等。在这种情况下，你可以在工作区根文件夹下创建一个名为 `.gitignore` 的文件，此文件的内容可以列出匹配模式的文件，与这些模式匹配的文件默认不会被纳入版本控制。即使用 `git add *` 等命令也会忽略这些匹配的文件。

如：`.gitignore` 文件的内容如下:

```
*.[oa]
*.obj
```

上述文件第一行告诉 Git 忽略任何以 `.o` 或 `.a` 结尾的文件。第二行告诉 Git 忽略所有文件名以 `.obj` 结尾的文件。

在创建一个新的 Git 仓库时，设置一个 `.gitignore` 文件通常是个好办法，这样你就不会意外提交你确实不想纳入 Git 仓库的文件。

你可以放入 `.gitignore` 文件的模式规则如下：

1. 空行或以 # 开头的行会被忽略。这是 `.gitignore` 文件的注释。
2. 标准的 glob 模式有效，并且会在整个工作树中递归应用。
3. 你可以以正斜杠（`/`）开头模式以避免递归。
4. 你可以以正斜杠（`/`）结尾模式来指定一个文件夹。
5. 你可以通过在模式前加上感叹号（`!`）来取反该模式。

Glob 模式类似于 shell 使用的简化正则表达式。星号（`*`）匹配零个或多个字符；`[abc]` 匹配括号内的任何字符（本例中是 `a`、`b` 或 `c`）；问号（`?`）匹配单个字符；用连字符分隔字符并用括号括起来（`[0-9]`）匹配它们之间的任何字符（本例中是 `0` 到 `9`）。你也可以使用两个星号来匹配嵌套文件夹；`a/**/z` 将匹配 `a/z`、`a/b/z`、`a/b/c/z` 等等。

以下是一个 `.gitignore` 文件的例子：

```
# 忽略所有 .a 文件
*.a

# 但是跟踪 lib.a，即使你上面忽略了 .a 文件
!lib.a

# 只忽略当前文件夹下的 TODO 文件，而不是 subdir/TODO
/TODO

# 忽略所有名为 build 的文件夹下的所有文件
build/

# 忽略 doc/notes.txt，但不忽略 doc/server/arch.txt
doc/*.txt

# 忽略 doc/ 文件夹及其任何子文件夹中的所有 .pdf 文件
doc/**/*.pdf
```

提示

GitHub 维护了一个相当全面的 `.gitignore` 文件示例列表，适用于数十个项目和语言，地址是：https://github.com/github/gitignore，如果你想为你的项目设置一个方便的提交方式，可以参照这个文件的写法。

注意

在通常情况下，仓库可能在根文件夹中有一个 `.gitignore` 文件，它递归地应用于整个仓库。然而，也可以在子文件夹中拥有额外的 `.gitignore` 文件。这些嵌套的 `.gitignore` 文件中的规则仅适用于它们所在文件夹下的文件。Linux 内核源代码仓库就有 206 个 `.gitignore` 文件。

深入探讨多个 `.gitignore` 文件的细节详情请参阅手册 `man gitignore`。

5. 标签操作

标签的概念

鉴于在版本控制当中使用**提交对象哈希值**那样一长串字符并不容易记忆。Git 提供了可以将提交对象哈希值 起一个别名的功能，这个别名就是**标签（TAG）**。标签对应一个提交对象的哈希值。

在项目进行到一个关键节点时进行的提交，我们通常为本次提交打一个标签，以示重要，也方便一个查找和定位当前提交。

通常这个**标签**用来定义版本号，用于版本的迭代。

Git 支持两种标签：

1. 轻量标签（lightweight）
   - 它只是某个特定提交的引用（本文不讲解）。
2. 附注标签（annotated）
   - 是存储在 Git 数据库中的一个完整对象，其中包含打标签者的名字、电子邮件地址、日期时间，此外还有一个标签信息。

本文只讲解**附注标签**。

创建标签

命令格式

```
git  tag -a [标签名] [提交对象ID] [-m  "提交信息"]
```

说明:

1. 提交对象ID可以不写，默认标签表示最新的提交位置。
2. 提交信息也可以不写，但是最好添加。

示例:

使用 `git tag -a v0.1 8cb34d1 -m "版本 0.1"` 命令为 `my_project` 项目中的第一次提交打一个标签 `v0.1`。使用 `git tag -a v0.2 -m "版本 0.2"` 命令为第二次提交打一个标签 `v0.2`。

```
weimingze@mzstudio:~/my_project$ git log --oneline
5dcccb0 (HEAD, master) 魏明择在当前项目中修改了README.md文件，并添加了website.txt文件
8cb34d1 魏明择在当前项目中添加了 README.md文件
weimingze@mzstudio:~/my_project$ git tag -a v0.1 8cb34d1 -m "版本 0.1"
weimingze@mzstudio:~/my_project$ git tag -a v0.2 -m "版本 0.2"
weimingze@mzstudio:~/my_project$ git tag
v0.1
v0.2
```

上述使用 `git tag` 命令时查看当前仓库有哪些标签。

查看标签

命令格式

```
# 查看标签列表
git tag

# 查看标签详细信息
git show [tag_name]
```

示例:

查看标签 `v0.1` 的详细信息

```
weimingze@mzstudio:~/my_project$ git show v0.1
tag v0.1
Tagger: weimingze <auth@weimingze.com>
Date:   Sun Jan 11 22:28:05 2026 +0800

版本 0.1

commit 8cb34d10643636d04286624adaa55b6a3967ab9e (tag: v0.1)
Author: weimingze <auth@weimingze.com>
Date:   Sat Jan 10 17:15:05 2026 +0800

    魏明择在当前项目中添加了 README.md文件

diff --git a/README.md b/README.md
new file mode 100644
index 0000000..d8b46cf
--- /dev/null
+++ b/README.md
```

```
@@ -0,0 +1 @@
+# Git 教程^M
```

上述信息列出了标签 `v0.1` 标签的详细信息，也列出了对应的提交对象的信息，同时显示了本次提交对象和之前版本的不同之处。

回退到某个标签节点

命令格式

```
git restore --source=标签名
```

示例：

使用标签将 v0.1 的版本恢复到工作区。然后再回复 v0.2 的版本到工作区。

```
weimingze@mzstudio:~/my_project$ git restore --source=v0.1 .
weimingze@mzstudio:~/my_project$ ls
README.md
weimingze@mzstudio:~/my_project$ git restore --source=v0.2 .
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
```

删除标签

命令格式：

```
git tag -d [标签]
```

示例

删除 `my_project` 项目中的 `v0.1` 标签和 `v0.2` 标签

```
weimingze@mzstudio:~/my_project$ git log --oneline
5dcccb0 (HEAD, tag: v0.2, master) 魏明择在当前项目中修改了 README.md 文件，并添加了 we
bsite.txt 文件
8cb34d1 (tag: v0.1) 魏明择在当前项目中添加了 README.md文件
weimingze@mzstudio:~/my_project$ git tag -d v0.1
Deleted tag 'v0.1' (was ec4f821)
weimingze@mzstudio:~/my_project$ git tag
v0.2
weimingze@mzstudio:~/my_project$ git tag -d v0.2
Deleted tag 'v0.2' (was abcb88d)
weimingze@mzstudio:~/my_project$ git tag
```

```
weimingze@mzstudio:~/my_project$ git log --oneline
5dcccb0 (HEAD, master) 魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件
8cb34d1 魏明择在当前项目中添加了 README.md文件
```

练习：

给你的 Git 提交信息打标签，然后删除这些标签。

第三章、远程仓库

Git 远程仓库通常是指托管在互联网平台上的 Git 仓库。其实这个仓库也可以是局域网内或你自己主机上的 Git 仓库。通常这些仓库用于多人协作开发，并且可能大部分仓库是对你只读。

1. Git 远程仓库的应用场景

我的 `my_project` 项目 已经有了两次提交。现在为了赶进度。我邀请小张、小李、小赵也来参加项目的开发。为了方便多人合作开发。我需要搭建一个远程仓库。

搭建远程仓容的方法有很多。

- 方法1：

如果我们都是在同一个局域网内开发。我可以将我自己的电脑（Ubuntu Linux操作系统）改造成为 Git 远程服务器，其它人可以把我的本地仓库作为远程仓库访问。

- 方法2（推荐）：

我们可以在公网上购买一台属于自己的服务器或者将自己的服务器放在公网上，在自己的服务器上搭建 Git 远程仓库，这样的服务器整体可控且安全。当然这个 Git 服务器也可以搭建在局域网内。这种做法对安全级别要求较高的项目。

- 方法3：

使用现有的代码托管平台，如国际的 Github（https://github.com） 和国内的 码云（https://gitee.com） 。这种方法成本低，方便，但安全性差，你的代码有可能会拿去训练人工智能的大模型。安全性取决于代码托管平台的安全性和你自己的使用方法。这种做法适合中小企业和个人等对安全级别要求不高的项目。除了上述提到的代码托管平台外，国内和国外还有很多这样的代码托管平台，如果需要请各位朋友自己查找。

本教程将使用国内的 **码云** 作为 Git 远程服务器来实现代码托管。

2. 远程仓库的基础操作

本节将介绍添加、移除远程仓库，以及获取 Git 远程仓库的提交信息等基本操作。

在上一章中，我们使用 `git clone https://gitee.com/weimz/py2048.git` 命令将魏明择用 Python 语言写的 2048 游戏的开源代码从远程仓库克隆到了本地。本地仓容的文件夹是 `py2048`，我们可以使用 `git remote` 命令查看本地仓库 `py2048` 中设置的远程仓库的信息。如下：


```
weimingze@mzstudio:~$ cd py2048/
weimingze@mzstudio:~/py2048$ git remote
origin
```

上面 `origin` 就是远程仓库的名称（Git默认名称，并可以修改）。此名称对应的就是远程仓库 `https://gitee.com/weimz/py2048.git`。我们还可以使用 `git remote -v` 来查看更详细的信息，如下所示:

```
weimingze@mzstudio:~/py2048$ git remote -v
origin  https://gitee.com/weimz/py2048.git (fetch)
origin  https://gitee.com/weimz/py2048.git (push)
```

可见名称 `origin` 就是远程仓库 `https://gitee.com/weimz/py2048.git` 的别名。其中此远程仓库 `origin` 的获取（fetch）地址是 `https://gitee.com/weimz/py2048.git`，推送（push）地址也是 `https://gitee.com/weimz/py2048.git`。

一个本地仓库可以拥有不止一个远程仓库。默认如果你是从一个远程仓库克隆出来的本地仓库，那么这个远程仓库默认的名称是 `origin` 对应的远程仓库就是你指定的远程仓库的链接地址（URL）。我们可以为一个本地仓库添加多个远程仓库，这样我们就可以从远程仓库获取代码，也可以将自己编写的代码文件推送到其它的远程仓库。

如我已经分享在码云上的远程仓库有如下几个:

项目名称	仓库地址
2048	`https://gitee.com/weimz/py2048.git`
飞机大战	`https://gitee.com/weimz/plane_war.git`
编程电子书	`https://gitee.com/weimz/books.git`

> 以上远程仓库对所有人只读。

添加远程仓库

在本地仓库中，使用 `git remote add` 命令可以将远程仓库添加到本地仓库。同时可以为此远程仓库的 URL 取一个别名，方便后续使用。

`git remote add` 命令

格式:


```
git remote add 简短别名 远程仓库URL地址
```

示例：

下面将飞机大战的远程仓库加入到 `py2048` 本地仓库，同时取别名为 `planewar`。

```
weimingze@mzstudio:~/py2048$ git remote add planewar https://gitee.com/weimz/
plane_war.git
weimingze@mzstudio:~/py2048$ git remote -v
origin  https://gitee.com/weimz/py2048.git (fetch)
origin  https://gitee.com/weimz/py2048.git (push)
planewar    https://gitee.com/weimz/plane_war.git (fetch)
planewar    https://gitee.com/weimz/plane_war.git (push)
```

可见项目中多了一个别名为 `planewar` 的远程仓库，这个远程仓库是 `https://gitee.com/weimz/plane_war.git`。

从远程仓库中获取数据

现在我们已经设置了另外一个远程仓库 `planewar`但远程仓库的内容并没有放入到本地仓库中。下面我们使用 `git fetch` 命令来获取 `planewar` 远程仓库的内容并形成本地仓库中的一个或多个分支。

`git fetch` 命令

这个命令的作用是从远程仓库中抓取所有仓库中的内容，并形成本地仓库的一个分支，比如 `planewar` 内、内部只有一个 master 分支。则使用 `git fetch planewar` 命令后，本地就会多一个 `planewar/master` 分支。这个远程分支我们可以使用 `git branch -r` 命令进行查看。

命令格式

```
 git fetch 远程仓库名
```

示例：

从远程仓库抓取数据到本地仓库。在抓取前我们使用 `git branch -r`查看 `py2048` 本地仓库的所有分支，包含远程分支如下。

```
weimingze@mzstudio:~/py2048$ git branch -r
  origin/HEAD -> origin/master
  origin/master
```

我们看到目前本地仓库中只有一个 `origin` 远程仓库的 `master` 分支，其中 `origin/HEAD` 是 远程仓库的默认分支指针，它指向 `origin/master`，这个我们不用理会。

下面我行使用 `git fetch planewar` 将远程仓库 `planewar` 的所有分支抓取到本地。

```
weimingze@mzstudio:~/py2048$ git fetch planewar
remote: Enumerating objects: 174, done.
remote: Counting objects: 100% (35/35), done.
remote: Compressing objects: 100% (34/34), done.
remote: Total 174 (delta 8), reused 2 (delta 0), pack-reused 139 (from 1)
Receiving objects: 100% (174/174), 1.55 MiB | 1.22 MiB/s, done.
Resolving deltas: 100% (31/31), done.
From https://gitee.com/weimz/plane_war
 * [new branch]      master     -> planewar/master
 * [new tag]         v0.1       -> v0.1
 * [new tag]         v0.2       -> v0.2
weimingze@mzstudio:~/py2048$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  planewar/master
```

从运行结果可以看出本地仓库多了一个 `planewar/master` 分支，同时多了两个标签 `v0.1` 和 `v0.2` 这是 `planewar` 远程仓库中的标签也被抓取到了本地仓库中。

从远程仓库提取代码到工作区

执行了上述操作后，我们工作区的代码并没有发生变化。下面我们使用 `git restore` 命令将 `planewar/master` 分支中的最后一次提交的文件还原到工作区。

注意工作区中的文件会被清除，请提前做好备份或保存到暂存区。

命令如下：

```
git restore --source=planewar/master --worktree .
```

执行结果如下:

```
weimingze@mzstudio:~/py2048$ ls
2048game.py  README.md  images
weimingze@mzstudio:~/py2048$ git restore --source=planewar/master --worktree .
weimingze@mzstudio:~/py2048$ ls
LICENSE  README.md  game  images  main.py
```

上述工作区中 `main.py`、`README.md`、`LICENSE` 就是远程仓库 `planewar` 中的文件，其中的`game`、`images`是文件夹。我们使用 `git status` 就能查看到这些文件的状态，并且 `main.py`、`LICENSE`、

`game`、`.gitignore`都是**未跟踪状态**。接下来我们使用 `git restore --worktree .` 还原为原来的工作区。

执行结果如下:

```
weimingze@mzstudio:~/py2048$ git restore --worktree .
weimingze@mzstudio:~/py2048$ ls
2048game.py  LICENSE  README.md  game  images  main.py
```

可见 `2048game.py`、`README.md` 和 `images` 中的文件又回来了。因为当前还是在本地仓库的主分支 `master` 中。

查看某个远程仓库

在使用远程仓库时。我们可以使用 `git remote show <远程仓库名>` 命令来查看远程仓库更过的信息。

如下面使用 `git remote show` 查看 `origin` 的信息。

```
weimingze@mzstudio:~/py2048$ git remote show origin
* remote origin
  Fetch URL: https://gitee.com/weimz/py2048.git
  Push  URL: https://gitee.com/weimz/py2048.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

上述信息会显示当前远程仓库 `origin` 的获取地址和推送地址，并且只有一个分支 `master`，此时当前操作位置指针 `HEAD` 位于 `master` 分支。为 `'git pull'`(后面会讲) 设置的本地分支为本地 `master` 与远程 `master` 合并。为 `git push`（后面会讲）设置的本地引用为 `master` 推送至远程 `master` (最新)。

远程仓库的重命名

使用 `git remote rename` 命令可以将一个已有的远程仓库名改名。

命令用法

```
git remote rename 旧名称 新名称
```

如我将上述远程仓库名 `planewar` 改为 `pw`则命令为 `git remote rename planewar pw`，如果再改回来，只需要将新旧名称对调即可。

示例：

```
weimingze@mzstudio:~/py2048$ git remote
origin
planewar
weimingze@mzstudio:~/py2048$ git remote rename planewar pw
Renaming remote references: 100% (1/1), done.
weimingze@mzstudio:~/py2048$ git remote
origin
pw
weimingze@mzstudio:~/py2048$ git remote rename pw planewar
Renaming remote references: 100% (1/1), done.
weimingze@mzstudio:~/py2048$ git remote
origin
planewar
```

上面的示例改了远程仓库`planewar` 的名字为 `pw`后，又将名字改回来了。

远程仓库的移除

我们在本地仓库中加入了远程仓库。当我们不在使用时，可以使用 `git remote remove <远程仓库名>` 命令移除远程仓库。

需要注意的是，你一旦使用这种方式移除了一个远程仓库，那么所有和这个远程仓库相关的远程分支以也会被一起删除。

示例：

删除远程仓库 `planewar`。

```
weimingze@mzstudio:~/py2048$ git remote
origin
planewar
weimingze@mzstudio:~/py2048$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  planewar/master
weimingze@mzstudio:~/py2048$ git remote remove planewar
weimingze@mzstudio:~/py2048$ git remote
origin
weimingze@mzstudio:~/py2048$ git branch -r
  origin/HEAD -> origin/master
  origin/master
```

可见远程仓库 `planewar` 被移除后，远程分支 `planewar/master` 也同时被移除了。

实验：

1. 将远程仓库 `https://gitee.com/weimz/books.git` 添加到你的本地仓库，取别名为 `books`。
2. 抓取远程仓库 `books` 中的全部内容到工作区中。
3. 删除远程仓库 `books`。

3. 添加现有工程到远程仓库

本小节的目标是将我们已经提交了两次的 `my_project` 项目 放入码云上建立的远程仓库，连同两次提交信息一起同步到远程仓库。同时学习 Git 远程仓库的推送和拉取操作。

现有的 `my_project` 项目在本地已经有了 `master` 分支，并有了两次提交。当前这个本地仓库并没有添加任何远程仓库，当你使用 `git remote` 命令显示时也是显示空的内容。

要将此本地仓库创建一个相同的远程仓库有几种做法。

1. 我们利用本地的仓库 `.git` 文件夹创建一个远程仓库。
2. 我先在云平台上创建一个空的，没有分支和提交的最原始的远程仓库，然后将本地仓库的所有分支和推送到远程仓库，实现远程仓库的同步。

本小节我们使用第二种方法来实现远程仓库的建立。首先我们要在码云平台上创建空的仓库。

在创建空的远程仓库前我们需要在码云平台注册账号，其网址是：https://gitee.com/

建立码云平台的仓库

1、 登录码云平台账户

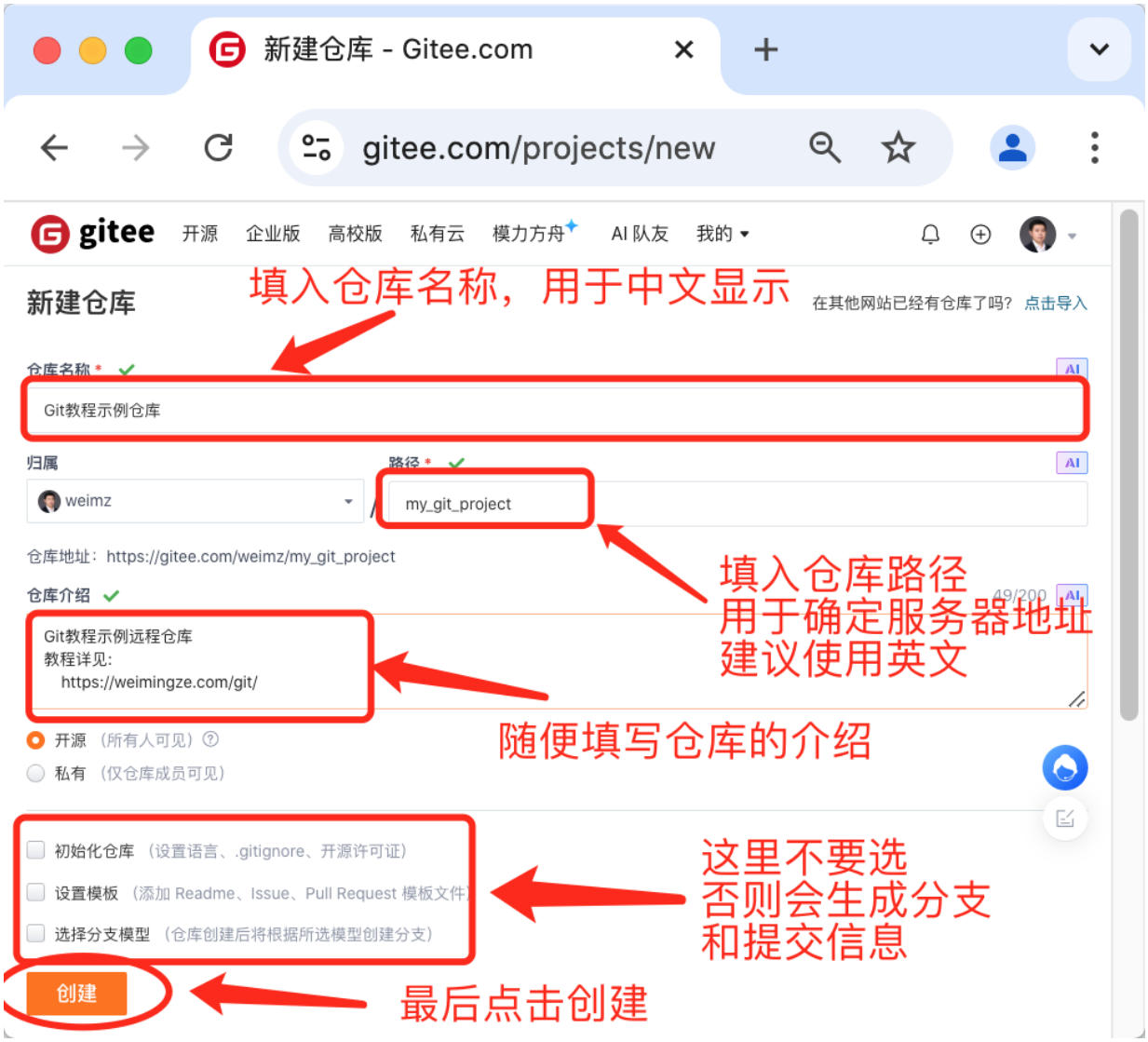
创建并登录码云账户即可。

2、 选择创建仓库

点击**新建仓库**，新建一个远程仓库，如下图所示:

3、 填写新仓库信息

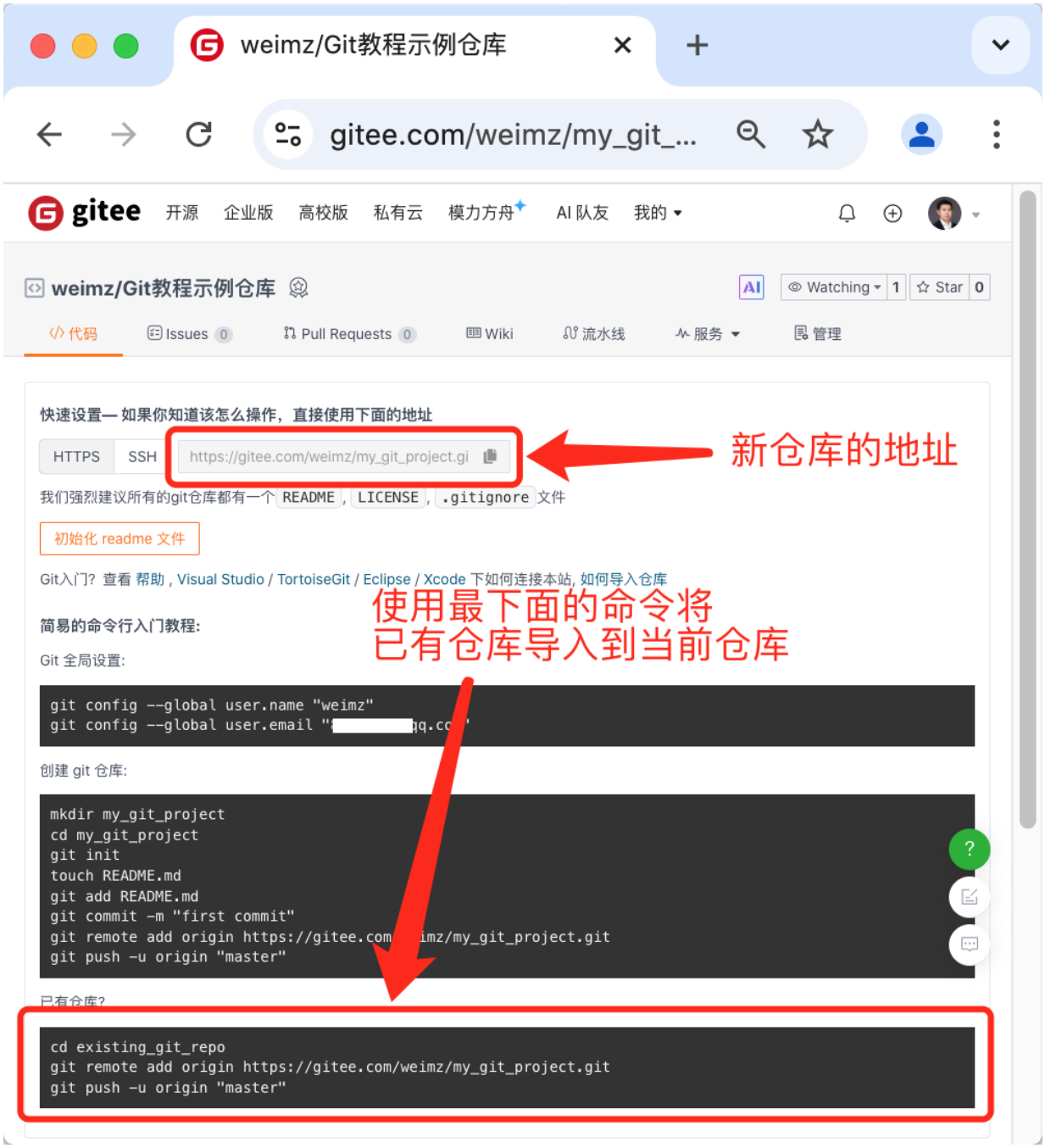
其中最重要的就是仓库的存储路径，这里填写 `my_git_project`，此路径最好选择只用英文填写。如下图所示：



然后点击**创建**。

4、 创建完毕，显示最终的仓库信息

如下图所示，最终创建远程仓库的地址是 `https://gitee.com/weimz/my_git_project.git`。



本页面还给出了对于已有本地仓库用户如何导入本地仓库到此远程仓库的方法如下：

```
cd existing_git_repo
git remote add origin https://gitee.com/weimz/my_git_project.git
git push -u origin "master"
```

这里我们本地仓库的路径是 `~/my_project` 不是 `existing_git_repo`。

5、 导入本地仓库到码云仓库

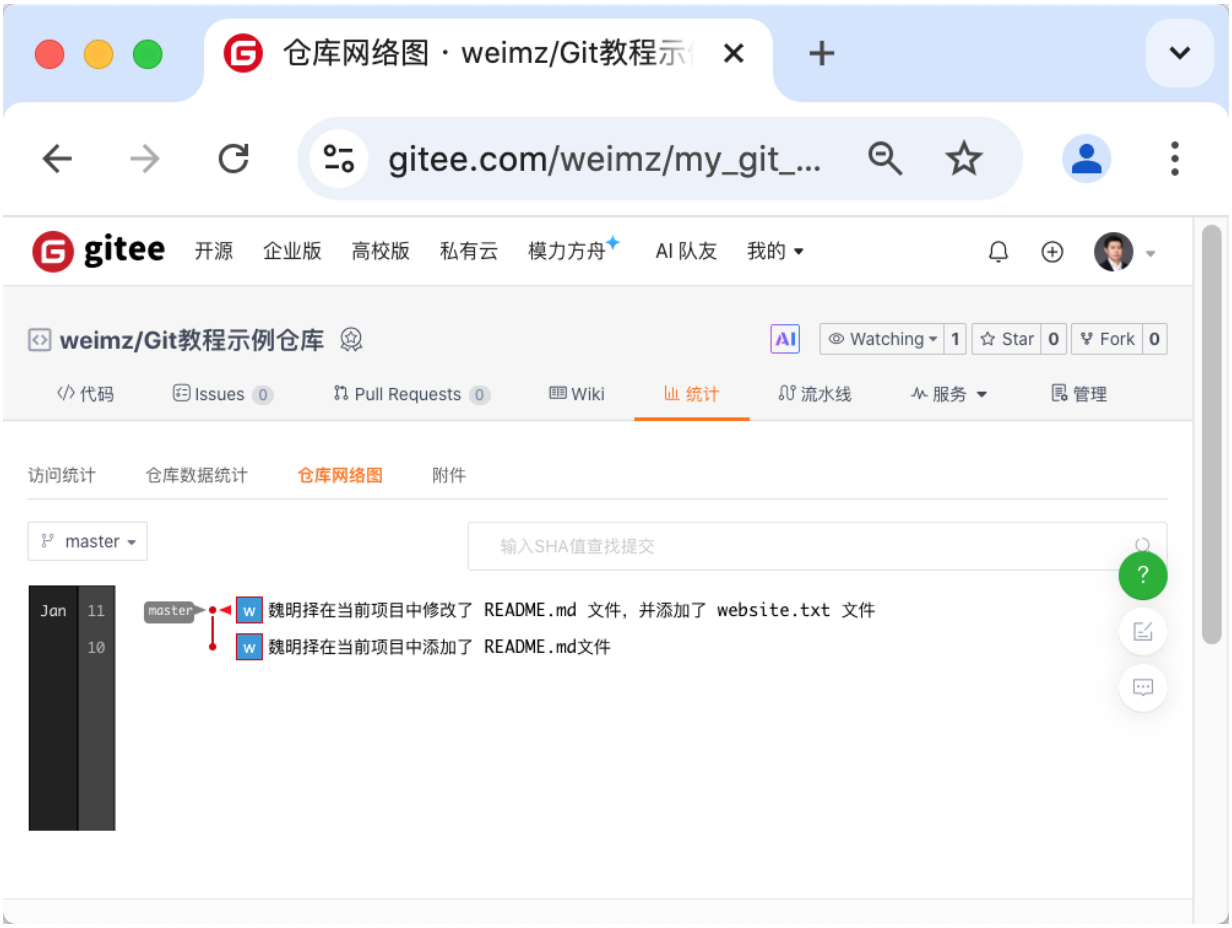
我们改写上述命令如下：


```
cd my_project
git remote add origin https://gitee.com/weimz/my_git_project.git
git push -u origin "master"
```

在我的电脑上的执行后的结果如下：

```
weimingze@mzstudio:~$ cd my_project
weimingze@mzstudio:~/my_project$ git remote add origin https://gitee.com/weimz/
my_git_project.git
weimingze@mzstudio:~/my_project$ git push -u origin "master"
Username for 'https://gitee.com': weimz
Password for 'https://weimz@gitee.com':
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (7/7), 676 bytes | 225.00 KiB/s, done.
Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
remote: Powered by GITEE.COM [1.1.23]
remote: Set trace flag 6d40f28d
To https://gitee.com/weimz/my_git_project.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
weimingze@mzstudio:~/my_project$ git remote
origin
weimingze@mzstudio:~/my_project$ git remote -v
origin  https://gitee.com/weimz/my_git_project.git (fetch)
origin  https://gitee.com/weimz/my_git_project.git (push)
```

现在我们已经完成将本地仓库推送到了远程仓库，此时的远程仓库的内容和本地仓库保持一致。你可以通过链接地址：https://gitee.com/weimz/my_git_project/ 查看远程仓库的内容和提交信息。你也可以通过 `git clone https://gitee.com/weimz/my_git_project.git` 命令来获取此仓库的一个副本。因为你没有加入到我们开发团队，此仓库对你是只读的。从网页中你可以看到项目的中两次提交信息和每次提交的内容，如下图所示：



上面的命令 `git push -u origin "master"` 是将本地分支 `"master"`推送本地的提交到远程仓库 `origin`。其中 `-u` 选项是建立本地分支与远程分支的追踪关系，设置后，以后可以直接使用 `git push` 或 `git pull` 而不需要指定远程仓库和分支名。关于 `git push` 和 `git pull` 命令我们下节课再详细讲解。

提示：

用同样的方法，可以将你的本地仓库导入到你自己的 Git 服务器或 Github 代码托管平台。

实验：

1. 在码云上创建仓库，然后将这个仓库克隆到本地。

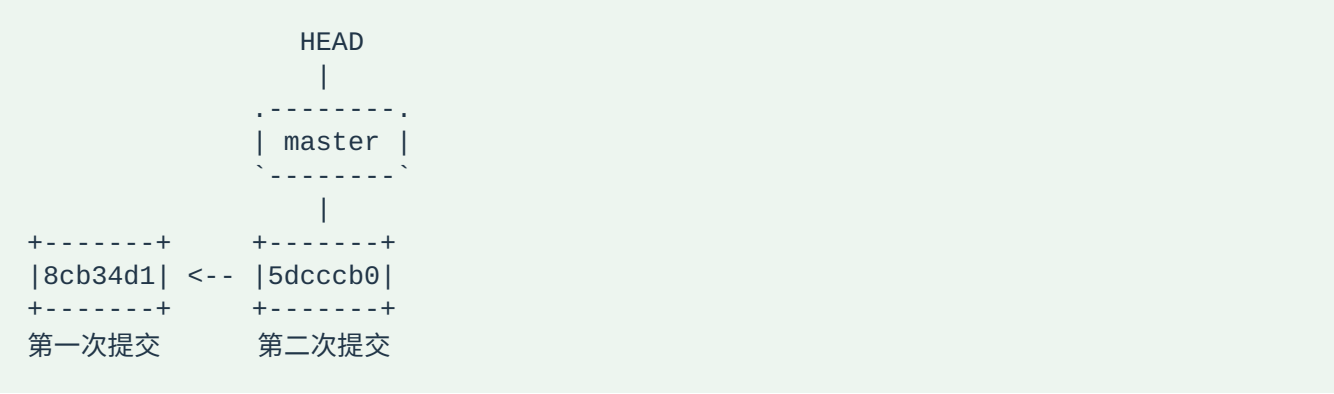
4. 远程仓库拉取和推送

本小节我们来学习如何使用远程仓库进行协作开发。我们将学习如下三个常用与远程操作的命令。

命令	说明
`git fetch`	从远程仓库下载所有的提交对象到本地，但不合并到本地当前分支。
`git pull`	从远程仓库下载所有的提交对象到本地，然后自动合并到本地当前分支。
`git push`	将本地提交上传到远程仓库并更新远程分支。如果此时远程有修改，则需要先拉取`git pull`然后再推送。

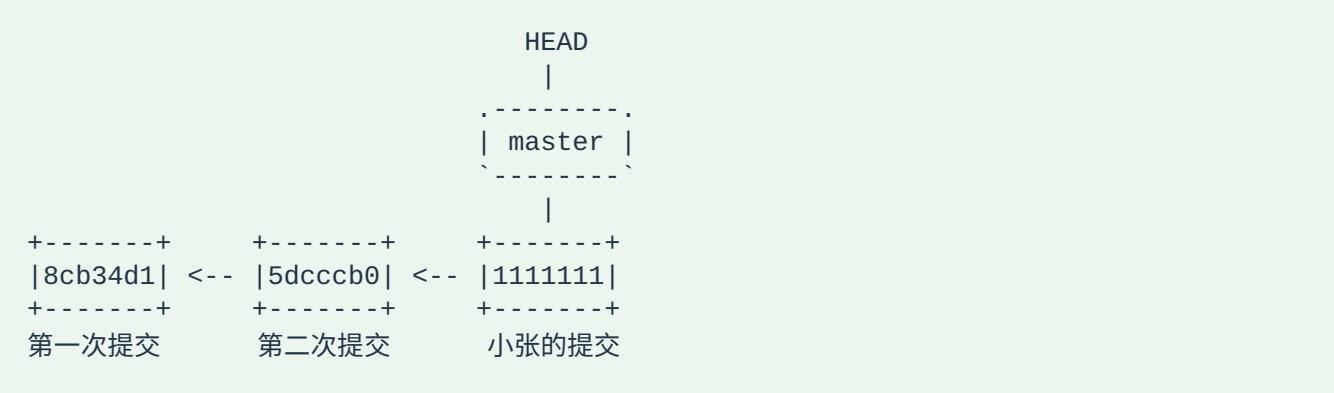
下面我们来解释一下上述命令的应用场景。

上一小节我们已经将`my_project`最新的两次提交都推送到了码云的远程仓库中。现在本地仓库和远程仓库的内容如下：



它们都有两个提交对象，并且提交对象的ID（哈希值）也完全一致。

现在假设你的合作伙伴小张完了一次新的功能，并提交到远程仓库中。远程仓库的内容编程如下状态：



而你本地的`master`分支依旧没有变化。

此时我们可以使用`git fetch`或`git pull`将远程仓库中小张的修改也同步到我的本地仓库中，那这两个命令有什么不同呢？不同之处在于`git fetch`只是将远程仓库的分支下载到本地仓库，形

成一个 `origin/master` 的远程分支，但本地的 `master` 分支不会有任何变化。而 `git pull` 先做 `git fetch` 的工作，然后将 远程分支 `orgin/master` 合并但本地分支 `master` 中，此时的本地分支和远程分支一模一样。

`git fetch` 命令

作用： 从远程仓库下载所有的提交对象到本地，但不合并到本地当前分支。

命令格式如下:

```
git fetch [选项] [远程仓库名] [远程分支名]
```

说明:

- 远程仓库名如果省略不写，默认是 `origin`。
- 远程分支名如果省略不写，默认是 `master`。
- 如果远程仓库名和远程分支名都省略，则默认抓取 `origin` 的所有分支.
- `git fetch` 不会改变你的本地分支。

`git fetch` 常用选项

选项	说明
`--all`	获取所有远程的所有分支。
`--prune` 或 `-p`	获取并删除本地已不存在的远程分支引用。
`-v`	获取并显示详细信息
`--force`	强制更新所有分支
`--tags`	获取所有标签

示例:

在上述远程仓库 已经有个了提交对象 `1111111` 的情况下使用 `git fetch` 获取后的结果。

本地的远程分支 `origin/master` 分支是这样：

```
          HEAD
           |
        .--------.
        | master |
```



```
                              `--------`
                                  |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |1111111|
+-------+     +-------+     +-------+
第一次提交      第二次提交      小张的提交
```

本地的 `master` 分支是这样：

```
                HEAD
                  |
               .--------.
               | master |
               `--------`
                   |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
第一次提交      第二次提交
```

可见本地的 `master` 分支并没有变化。此时如果我们在本地仓库产生新的提交就会在 `master` 分支后创建提交对象。这样会导致本地分支和远程分支不一致的情况。

`git pull` 命令

作用：从远程仓库下载所有的提交对象到本地，然后自动合并到本地当前分支。

命令格式如下:

```
git pull [选项] [远程仓库名] [远程分支名[:本地分支名]]
```

说明:

- 远程仓库名如果省略不写，默认是 `origin`。
- 远程分支名如果省略不写，默认是当前本地分支对应的分支名。
- 本地分支名如果省略不写，默认是当前本地分支。
- 如果远程仓库名和本地分支名都省略，则默认获取 `origin` 的当前分支对应的远程分支。
- `git pull` 命令可能会改变本地分支。

`git pull` 常用选项

选项	说明
`--rebase`	拉取所有远程的所有分支，并使用变基的方式合并到当前分支。
`--all`	拉取所有远程的所有分支。
`--no-commit`	拉取但不自动提交合并。需要手动提交。
`-v`	显示详细信息

示例：

在上述远程仓库 已经有个了提交对象 `1111111` 的情况下使用 `git pull` 获取后的结果。

本地的远程分支 `origin/master` 分支是这样：

```
                              HEAD
                               |
                           .--------.
                           | master |
                           `--------`
                               |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |1111111|
+-------+     +-------+     +-------+
第一次提交      第二次提交      小张的提交
```

本地的 `master` 分支会和本地的远程 `origin/master` 分支合并，即：将 `1111111` 这个提交对象合并到 `5dcccb0` 这个对象之后。最终结果同远程分支一致。如下图所示：

```
                              HEAD
                               |
                           .--------.
                           | master |
                           `--------`
                               |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |1111111|
+-------+     +-------+     +-------+
第一次提交      第二次提交      小张的提交
```

现在如果你完成了某个工作，再次本地提交则会在 `1111111` 提交对象后创建你的提交对象。

git pull 注意事项：

假设小张在远程仓库进行了提交，而你在本地仓库进行了提交。结果如下：

远程仓库的内容:


```
                              HEAD
                               |
                           .--------.
                           | master |
                           `--------`
                               |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |1111111|
+-------+     +-------+     +-------+
第一次提交      第二次提交      小张的提交
```

你的本地仓库的内容:

```
                              HEAD
                               |
                           .--------.
                           | master |
                           `--------`
                               |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |2222222|
+-------+     +-------+     +-------+
第一次提交      第二次提交       你的提交
```

这样在使用 `git pull` 可能就会出现合并失败的情况。因此带提交到远程仓库使用的分支时，最好先使用 `git pull` 将远程仓库的变化合并到本地。然后再进行提交。还有一个好办法就是在本地仓库建立其它的分支进行提交。在需要时在合并到远程仓库协作开发的分支中。

关于分支的管理我们后面会讲。

`git push` 命令

作用： 将本地分支推送到远程分支，让远程仓库的一个或多个分支与本地分支同步。

命令格式如下:

```
git push [选项] [远程仓库名] [本地分支名][:[远程分支名]]
```

说明:

- 远程仓库名如果不写，默认是 `origin`。
- 本地分支名如果不写，默认是 `master`。
- 远程分支名默认是本地分支名，也可是手动指定。

`git push` 常用选项

选项	说明
`--all`	推送所有分支
`-u`	建立本地分支与远程分支的追踪关系，只需要首次设置。
`--force`	强制推送（谨慎使用）
`--delete 分支名`	删除远程分支
`--tags`	推送所有标签

现在假设远程分支是这样：

```
                HEAD
                 |
              .--------.
              | master |
              `--------`
                 |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
第一次提交      第二次提交
```

而我的本地分支经过了一次提交，内容是如下这样：

```
                           HEAD
                            |
                         .--------.
                         | master |
                         `--------`
                            |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |2222222|
+-------+     +-------+     +-------+
第一次提交      第二次提交       你的提交
```

现在要让远程仓库和本地仓库同步，我可以使用 `git push` 命令远程仓库变成本地仓库的样子。

需要注意的是，默认情况下 `git push` 并不会向远程仓库推送标签，如果你需要推送所有的标签可以添加 `--tags` 选项。如果你只是向远程仓库推送某一个标签，你可以使用 `git push origin <标签名>` 的方式推送。这样远程仓库就存在标签并可以被其它使用者拉取到本地仓库，如推送：`v0.2` 这个标签，你可以使用命令 `git push origin v0.2` 命令完成推送。

实验:

在码云上创建远程仓库，然后克隆到本地，在本地编写代码并提交，最后将本地仓库的各个版本提交到码云上的远程仓库。在码云上查看你的提交。

第四章、Git 分支

1. 分支简介

分支就是一个全新的提交路径。它是指在项目开发过程中，为了某个新功能，在不影响开发主线的情况下，脱离原来的开发主线独立创建一条新开发路径的方式。

通常我们把原来的开发主线叫做主分支 `master`。为了方便记忆和管理通常我们把新分离出来的分支取一个具有代表性的名字，如：`develop`、`release`、`main`、`feature`等。

通常在开发过程中有如下情况是比较适合建立分支：

1. 开发新功能。
2. 修复 Bug。
3. 试验性开发
4. 旧版本维护
5. 多人协作开发

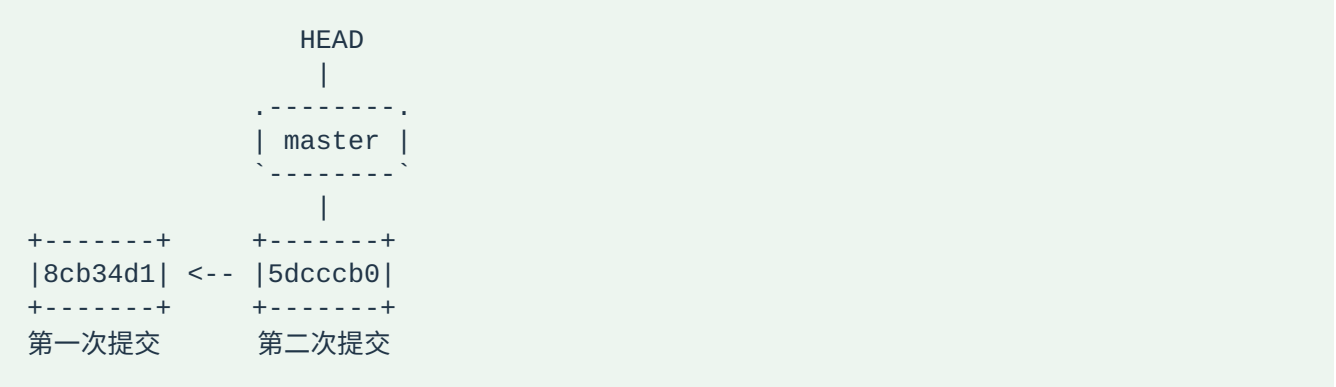
使用分支的好处如下：

1. 并行开发互不影响。
2. 隔离风险，试验性代码不会影响其它分支的稳定。
3. 提高协作效率

Git 分支的基本原理和概念

Git 分支本质上是指向特定提交对象的轻量级可移动指针。你可以把分支理解成一个时间线上的标记点，代表着一个独立的提交对象链。

先来回顾一下我们常见的 `my_project` 工程，其中有两次提交，Git 本地仓库内部结果如下：



上图中，`master` 就是一个指针，它指向最后一个提交对象 `5dcccb0`，每个提交对象都有一个向前指向的指针，指向上一次提交，这样就形成了一个提交对象链。而每个提交对象链记录的完整的提交过程和记录。

HEAD 指针

HEAD 指针是正在操作的分支的指针。它记录了你正在操作的分支。

上图中 `HEAD` 指向了 `master` 分支对指针，也就意味着当前所有的操作都是在 `master` 分支上进行。如果改变正在操作的分支，只需要让 `HEAD` 指向其它的分支指针即可。

创建分支

现在在当前提交对象位置创建一个新的分支 `test`，则创建后的仓库结构如下:

```
                HEAD
                 |
              .--------.
              | master |
              `--------`
                 |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
                 |
              .--------.
              |  test  |
              `--------`
```

下载有两个分支指针，即 `master` 和 `test` 这两个分支指针都指向了 `5dcccb0` 这个提交对象。从总体来看，这两个分支是一样的。它们都有两次提交，即提交历史 都是 `8cb34d1` 和 `5dcccb0`。

现在 `HEAD` 指针还是指向 `master` 分支，因此所有的提交都会操作的是 `master` 指针。

切换分支

如果要操作 `test` 分支，此时需要切换分支，将当前分支切换到 `test`， Git 的做法是将 `HEAD` 指针指向 `test` 即可，如切换到 `test` 分支后，仓库的结构如下:

```
              .--------.
              | master |
              `--------`
                 |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
                 |
```

```
              .--------.
              |  test  |
              `--------`
                  |
                 HEAD
```

提交版本

现在我们在 `test` 分支实现新**功能1**，然后提交一个新的版本。

当进行一次提交时，仓库内会新创建一个提交对象，这个提交对象会插在当前 `HEAD` 指向的分支的最后一个节点后，新创建的提交对象（假设是`3333333`）会指向之前的最后一个节点，然后分支指针指向次提交对象。最后形成这样一个结构。

```
              .--------.
              | master |
              `--------`
                  |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
                  ^
                  |         +-------+
                  `-------- |3333333|
                            +-------+
                               |
                            .--------.
                            |  test  |
                            `--------`
                                |
                               HEAD
```

现在我们又需要在主分支上实现另外一个**功能2**，然后提交一个新的版本，此时需要切换回 `master` 分支（即将 `HEAD` 指针指向 `master` 指针），然后再进行一次提交，假设此次提交创建的提交对象是 `4444444`，则提交后的仓库结构如下：

```
                               HEAD
                                |
                            .--------.
                            | master |
                            `--------`
                               |
+-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |4444444|
+-------+     +-------+     +-------+
                  ^
                  |         +-------+
                  `-------- |3333333|
                            +-------+
```



```
                              |
                          .--------.
                          |  test  |
                          `--------`
```

经过一段时间的测试我们发现 `test` 分支实现的新功能稳定可靠。我们需要将其添加到 `master` 分支，此时需要将 `test` 分支合并到 `master` 分支（当然也可以将 `master` 分支合并到 `test` 分支），合并后会在 `master` 分支上有创建一个新的提交对象，假设是 `555555`。则合并后的结构如下:

```
                                        HEAD
                                         |
                                     .--------.
                                     | master |
                                     `--------`
                                         |
+-------+     +-------+     +-------+     +-------+
|8cb34d1| <-- |5dcccb0| <-- |4444444| <-- |5555555|
+-------+     +-------+     +-------+     +-------+
                  ^                           |
                  |         +-------+  <----+  合并版本
                  `-------- |3333333|
                            +-------+
                               |
                           .--------.
                           |  test  |
                           `--------`
```

合并后，`master`分支d的提交历史是 `8cb34d1`、`5dcccb0`、`4444444`、`5555555`，其中实现了所有的功能。而 `test` 分支的提交历史是 `8cb34d1`、`5dcccb0`、`3333333`。

这样，**功能1**的开发和测试都在 `test` 分支上完成。待开发完成后我们将 test 分支上的某个提交合并到 `master` 即可。而此时 `test` 分支可以继续提交，当然也可以删除该分支。

使用 Git 进行版本控制时，可以创建多个分支，每个分支都可以合并到其它分支中。比较灵活和高效。Git 推荐使用分支进行版本控制。

2. 分支管理

Git 分支操作主要有创建分支、删除分支、列出分支、切换分支、合并分支，提交记录、等操作。

本小节我将学习如下几个关于分支的命令：

命令	说明
`git branch`	创建分支、删除分支、列出分支。
`git switch`	切换分支。

`git branch` 命令

作用： 用于创建分支、删除分支、列出分支。

命令格式

```
git branch [选项] [新分支名] [提交哈希值]
```

说明

1. `git branch` 命令是显示本地所有分支，在当前的分支前显示星号(`*`)。
2. 只给出新分支名则在当前位置创建新分支。
3. 只给出新分支名和提交哈希值则在指定的提交对象位置创建新分支。

常用选项

选项	说明
`-d <分支名>`	删除已合并分支。
`-D <分支名>`	删除未合并分支。
`-r`	查看远程分支。
`-a`	查看所有分支，包括远程分支。
`-v`	查看本地分支的最后一次提交信息。
`-vv`	查看本地分支的最后一次提交信息（包含跟踪关系）。
`--merged`	查看已合并到当前分支的分支。
`--no-merged`	查看未合并到当前分支的分支。
`-m <新分支名>`	重命名当前分支
`-m <旧分支名> <新分支名>`	重命名指定分支

示例：

之前的 `my_project` 工程，其中有两次提交，Git 本地仓库内部结构如下：

```
                HEAD
                  |
              .--------.
              | master |
              `--------`
                  |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
```

1、 查看 `my_project` 项目中本地分支。

```
weimingze@mzstudio:~/my_project$ git branch
* master
```

可见现在只有一个分支 `master` ，前面的星号（`*`） 表示当前分支指向 `master` 分支，即: `HEAD` 指向 `master` 分支指针。

2、 查看 `my_project` 项目中所有分支。

```
weimingze@mzstudio:~/my_project$ git branch -a
* master
  remotes/origin/master
```

`remotes/origin/master` 是远程分支。

3、 查看 `my_project` 项目中所有分支，并显示最后一次提交信息。

```
weimingze@mzstudio:~/my_project$ git branch -a -v
* master                 5dcccb0 魏明择在当前项目中修改了 README.md 文件，并添加了 webs
ite.txt 文件
  remotes/origin/master 5dcccb0 魏明择在当前项目中修改了 README.md 文件，并添加了 webs
ite.txt 文件
```

4、 在当前分支的最后一个提交对象位置创建一个 `myb1` 的分支。

```
weimingze@mzstudio:~/my_project$ git branch myb1
weimingze@mzstudio:~/my_project$ git branch
* master
  myb1
```

此时的仓库内部有两个分支 `master` 和 `myb1`，其中 `HEAD` 指向 `master` 分支指针，结果如下。

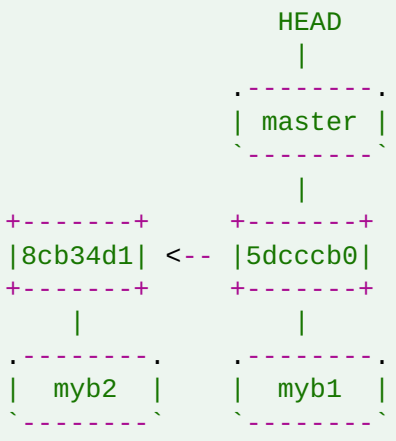
```
                HEAD
                  |
              .--------.
              | master |
              `--------`
                  |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
                  |
              .--------.
              |  myb1  |
              `--------`
```

5、 在当前分支的 `8cb34d1` 这个提交对象位置创建一个 `myb2` 的分支。

```
weimingze@mzstudio:~/my_project$ git branch myb2 8cb34d1
weimingze@mzstudio:~/my_project$ git branch
* master
  myb1
  myb2
weimingze@mzstudio:~/my_project$ git branch -v
* master 5dcccb0 魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件
```

```
  myb1   5dcccb0 魏明择在当前项目中修改了 README.md 文件，并添加了 website.txt 文件
  myb2   8cb34d1 魏明择在当前项目中添加了 README.md文件
```

可见当前本地仓库有三个本地分支，其中 `myb2` 分支指针指向了 `8cb34d1` 这个提交对象。



`git switch` 命令

作用： 用于切换到某个分支。

命令格式

```
git switch [选项] 分支名
```

`git switch` 的常用选项

选项	说明
`-c`	切换到当前分支，如果没有此分支则创建分支。
`-c <新分支名> <提交哈希值>`	在特定的提交对象的位置创建新分支并切换到当前分支。
`-`	切换到来到次分支前的那个分支。

注意：

1. Git v2.23 以前的版本需要使用 `git checkout` 代替 `git switch` 命令。
2. 在切换分支之前，你要保证当前工作区是干净的，如果你的工作区不是干净的工作区，则切换会失败。

切换分区后，工作区的内容会被切换后的分支的最后一次提交的内容覆盖。

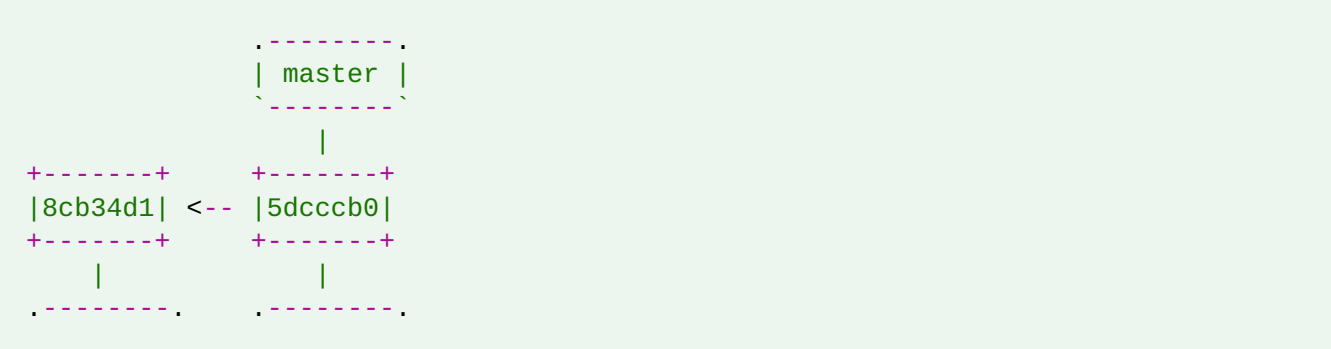
示例：

1、将当前分支切换到 `myb1` 分支。

```
weimingze@mzstudio:~/my_project$ git branch
* master
  myb1
  myb2
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
weimingze@mzstudio:~/my_project$ git switch myb1
Switched to branch 'myb1'
weimingze@mzstudio:~/my_project$ git branch
  master
* myb1
  myb2
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
```

可见，切换前 工作区有两个文件 `README.md` 和 `website.txt` 使用 `git switch myb1` 成功切换到了 `myb1` 分支。工作区内容没有变化。原因是此时 `master` 分支指针和 `myb1` 分支指针都指向了最后一个提交，切换前的结构如下：

切换到 `myb1`分支后只是 `HEAD` 指向了 `myb1` 分支指针。如下图所示：




```
|  myb2  |     |  myb1  |
`--------`     `--------`
                   |
                  HEAD
```

因为这两个分支指针都指向了 `5dcccb0` 这个提交对象。因此工作区的内容也被恢复到了 `5dcccb0` 提交时的状态。

2、 切换到 `myb2` 分支

```
weimingze@mzstudio:~/my_project$ git switch myb2
Switched to branch 'myb2'
weimingze@mzstudio:~/my_project$ git branch
  master
  myb1
* myb2
weimingze@mzstudio:~/my_project$ ls
README.md
```

当切换到 `myb2` 分支后，`HEAD` 指针指向了 `myb2` 分支指针。工作区也被恢复成 `8cb34d1` 提交的状态。状态如下：

```
              .--------.
              | master |
              `--------`
                  |
+-------+     +-------+
|8cb34d1| <-- |5dcccb0|
+-------+     +-------+
    |             |
.--------.    .--------.
|  myb2  |    |  myb1  |
`--------`    `--------`
    |
   HEAD
```

3、 删除 `myb2` 分支。

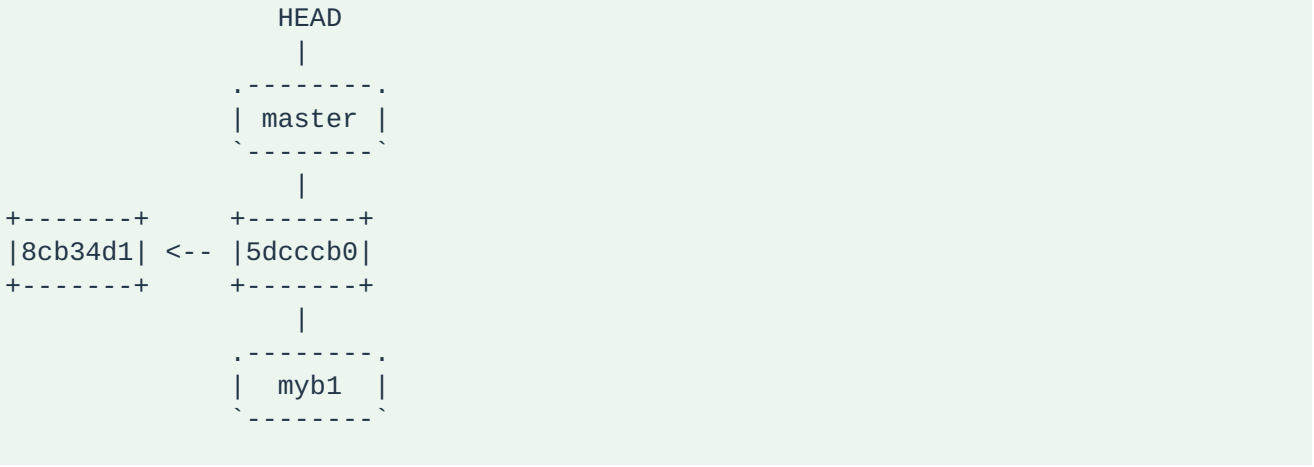
先要确保你的当前分支不是 `myb2` 分支，否则删除失败。提示如下：

```
weimingze@mzstudio:~/my_project$ git branch
  master
  myb1
* myb2
weimingze@mzstudio:~/my_project$ git branch -d myb2
error: cannot delete branch 'myb2' used by worktree at '/home/weimingze/
my_project'
```

下面我们切换当前工作分支为 `master` 分支，然后删除 `myb2` 分支。

```
weimingze@mzstudio:~/my_project$ git switch master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
weimingze@mzstudio:~/my_project$ git branch
* master
  myb1
  myb2
weimingze@mzstudio:~/my_project$ git branch -d myb2
Deleted branch myb2 (was 8cb34d1).
weimingze@mzstudio:~/my_project$ git branch
* master
  myb1
```

此时本地仓库恢复到了以下的状态:



现在之前创建的 `myb2` 分支被删除了。 `myb2` 分支指针也就不存在了。

3. 记录的提交和合并

本节课我们来学习不同分支的记录提交和合并。

本小节我将学习 `git merge` 命令。

命令	说明
`git merge`	合并两个或多个分支的记录。

上节课我们在原有 `my_project` 项目中添加了 `myb1` 分支，加上原有的默认分支 `master` 共有两个分支。本节课我们模拟不同的需求，在两个分支上分别添加功能并提交，然后合并两个分支中的修改记录合并到主分支中形成一个提交。

前面的章节中使用 `git commit` 命令可以将当前暂存区中的修改记录提交到当前分支中形成新的提交对象。提交的同时会移动当前分支指针指向最新生成的提交对象。

需要注意的是在不同的分支上进行提交时，需要使用 `git switch` 命令切换某个分支到当前分支。

假设我们现在要改写 `my_project` 项目的功能。为了不影响主分支的开发进度，我们最后要将此次修改提交到 `myb1` 分支，待修改完成后再将新加入的功能合并到主分支。

我们切换当前分支为 `myb1` 分支，然后在创建一个新文件 `function1.txt` 写入一行文件文字 `新功能1的实现代码`，然后再修改 `website.txt` 文件，在最末尾加入一行 `https://weimingze.com/c/`。

文件：`function1.txt` 内容如下:

```
新功能1的实现代码
```

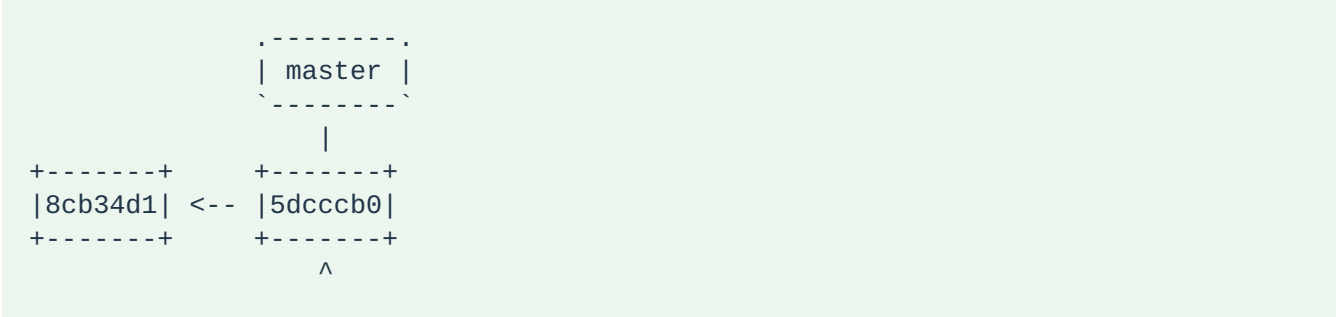
文件：`website.txtt` 内容如下:

```
https://weimingze.com
https://weimingze.com/c/
```

最后提交到 `myb1` 分支，我的操作如下：

```
weimingze@mzstudio:~/my_project$ git switch myb1
Switched to branch 'myb1'
weimingze@mzstudio:~/my_project$ git branch
  master
* myb1
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
weimingze@mzstudio:~/my_project$ echo -e -n "新功能1的实现代码\r\n" > function1.txt
weimingze@mzstudio:~/my_project$ echo -e -n "https://weimingze.com/c/\r\n" >> we
bsite.txt
weimingze@mzstudio:~/my_project$ git add website.txt function1.txt
weimingze@mzstudio:~/my_project$ git commit -m "完成了功能1"
[myb1 8e7f7c7] 完成了功能1
 2 files changed, 2 insertions(+)
 create mode 100644 function1.txt
weimingze@mzstudio:~/my_project$ ls
README.md  function1.txt  website.txt
```

此时在 `myb1` 上新创建了一个提交对象 `8e7f7c7`，这个提交对象记录了新增功能1的状态。 仓库结构如下:




```
                  |         +-------+
                  `-------- |8e7f7c7|
                            +-------+
                               |
                            .--------.
                            |  myb1  |
                            `--------`
                                |
                               HEAD
```

下面我们在主分支`master`上完成一个新功能2的代码并提交。

我们切换当前分支为`master`分支，然后在创建一个新文件`function2.txt`写入一行文件文字`新功能2的实现代码`，然后在修改`website.txt`文件，在最末尾加入一行`https://weimingze.com/git/`（需要注意此时`website.txt`文件的内容应当只有一行，不存在分支`myb1`中的`https://weimingze.com/c/`这一行）。

文件：`function2.txt`内容如下:

```
新功能2的实现代码
```

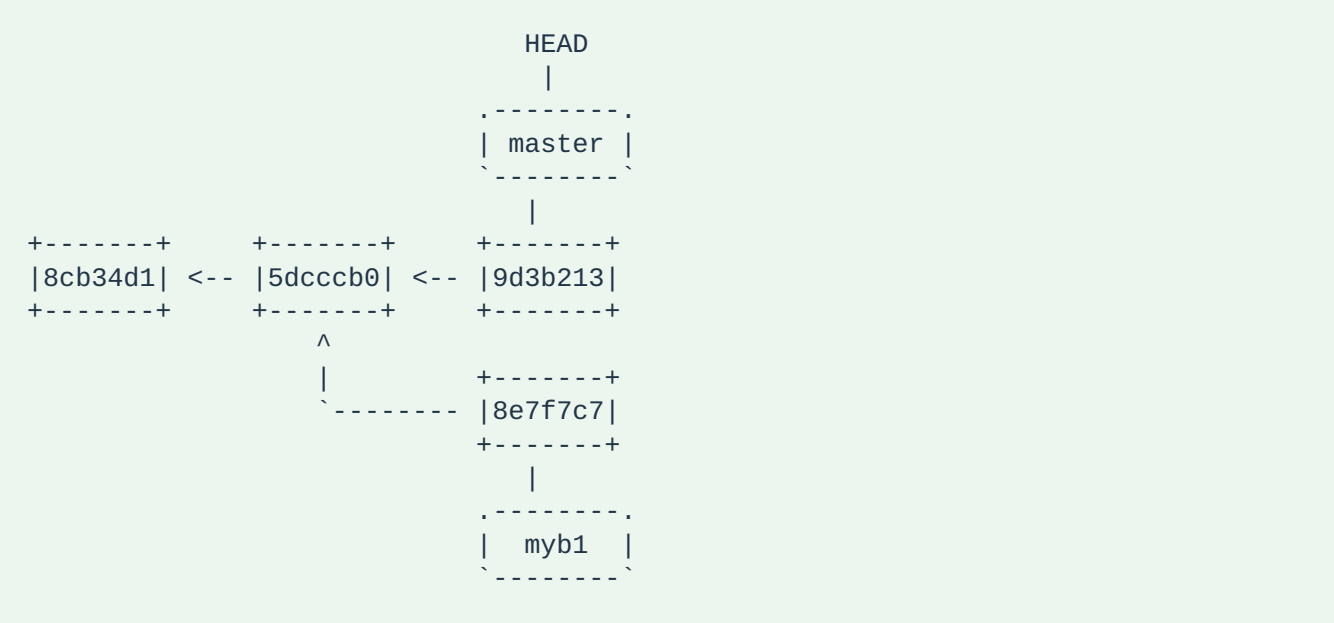
文件：`website.txtt`内容如下:

```
https://weimingze.com
https://weimingze.com/git/
```

完成上述编写后提交到`master`分支，操作如下：

```
weimingze@mzstudio:~/my_project$ git switch master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
weimingze@mzstudio:~/my_project$ ls
README.md  website.txt
weimingze@mzstudio:~/my_project$ echo -e -n "新功能2的实现代码\r\n" > function2.txt
weimingze@mzstudio:~/my_project$ echo -e -n "https://weimingze.com/git/\r\n" >>
website.txt
weimingze@mzstudio:~/my_project$ cat website.txt
https://weimingze.com
https://weimingze.com/git/
weimingze@mzstudio:~/my_project$ cat function2.txt
新功能2的实现代码
weimingze@mzstudio:~/my_project$ git add function2.txt website.txt
weimingze@mzstudio:~/my_project$ git commit -m "完成了功能2"
[master 9d3b213] 完成了功能2
 Date: Wed Jan 14 15:07:32 2026 +0800
 2 files changed, 2 insertions(+)
 create mode 100644 function2.txt
weimingze@mzstudio:~/my_project$ ls
README.md  function2.txt  website.txt
```

此时在 `master` 分支上新创建了一个提交对象 `9d3b213`，这个提交对象记录了新增功能2的状态。仓库结构如下:



到现在为止，`master` 分支最后的提交状态时有三个文件 `README.md`、`website.txt` 和 `function2.txt`, `myb1` 分支最后的提交状态也有三个文件 `README.md`、`website.txt` 和 `function1.txt`, 但 `master` 分支和 `myb1` 分支中的 `website.txt` 文件的内容是不同的。

合并分支

现在 `master` 分支中新增了**功能2**，`myb1` 分支中新增了 **功能1**。现在我们希望这个项目同时使用 **功能1**和**功能2** 两个功能。这个时候有两个比较方便的选择：第一种方法是将 `myb1` 合并到 `master` 分支中，形成一个新的提交对象，此提交对象中将 `myb1` 的修改内容添加进来，最后发布 `master` 分支的功能。第二种方法就是将 `master` 分支的修改合并到 `myb1` 分支中。

> 在 Git 中，分支都是平等的。如何操作分支取决于实际开发需求。

以下讲解 将 `myb1` 合并到 `master` 分支中的方法。合并分支需要使用 `git merge` 命令。

`git merge` 命令

作用： 将两个或两个以上分支的最终提交对象合并到当前分支并提交。

命令格式


```
git merge [选项] 分支名1 分支名2 分支名3 ...
```

`git merge` 的常用选项

选项	说明
`-m "提交说明信息"`	合并的提交说明。
`--no-commit`	仅合并，不提交。
`--commit`	合并后自动提交。
`--tool=<工具命令名>` 或 `-t 工具命令名`	使用指定合并工具，如 meld、vimdiff 等。
`--abort`	取消本次合并。

注意：

如果两个分支有共同修改的文件，则合并过程中可能会产生冲突，需要解决冲突问题，然后再使用 `git commit` 命令提交合并的结果。

示例：

上述两个提交中 `master` 分支和 `myb1` 分支都修改了 `website.txt` 中的内容，因此两个分支的 `website.txt` 合并时就会产生冲突。当产生冲突时，合并工作会中断，此时需要使用编辑软件对冲突的文件进行手动合并。当然也可以使用 `git mergetool` 命令调用可视化的工具进行手动合并。待合并完成后需要使用 `git commit` 来手动提交合并结果。

在 `myb1`分支中的 `website.txt` 内容如下:

```
https://weimingze.com
https://weimingze.com/c/
```

在 `master`分支中的 `website.txt` 内容如下:

```
https://weimingze.com
https://weimingze.com/git/
```

我们希望合并后的 `website.txt`是:


```
https://weimingze.com
https://weimingze.com/c/
https://weimingze.com/git/
```

合并时要注意一下几个可能引发合并失败的问题：

1. 将要并入的分支作为当前分支。
2. 确保工作区是干净的状态。
3. 工作区必须是当前分支的最新版本的状态。

具体操作如下:

```
weimingze@mzstudio:~/my_project$ git branch
* master
  myb1
weimingze@mzstudio:~/my_project$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
weimingze@mzstudio:~/my_project$ git merge myb1
Auto-merging website.txt
CONFLICT (content): Merge conflict in website.txt
Automatic merge failed; fix conflicts and then commit the result.
weimingze@mzstudio:~/my_project$ ls
README.md  function1.txt  function2.txt  website.txt
```

此时已经完成了合并，合并的结果存放在当前工作区，可见其中的四个文件 `README.md`、`function1.txt`、`function2.txt`、`website.txt` 其中提示了 `Auto-merging website.txt` 说明 `website.txt` 文件产生了冲突，并已经自动合并。当前文件内容如下:

```
https://weimingze.com
<<<<<<< HEAD
https://weimingze.com/git/
=======
https://weimingze.com/c/
>>>>>>> myb1
```

其中 `<<<<<<< HEAD` 和 `=======` 之间的内容是当前 `HEAD` 指向的分支 `master` 中的内容。而 `=======` 和 `>>>>>>> myb1` 之间是 `myb1` 分支中 `mywebsite.txt` 的内容。

手动改写 `website.txt` 内容如下:

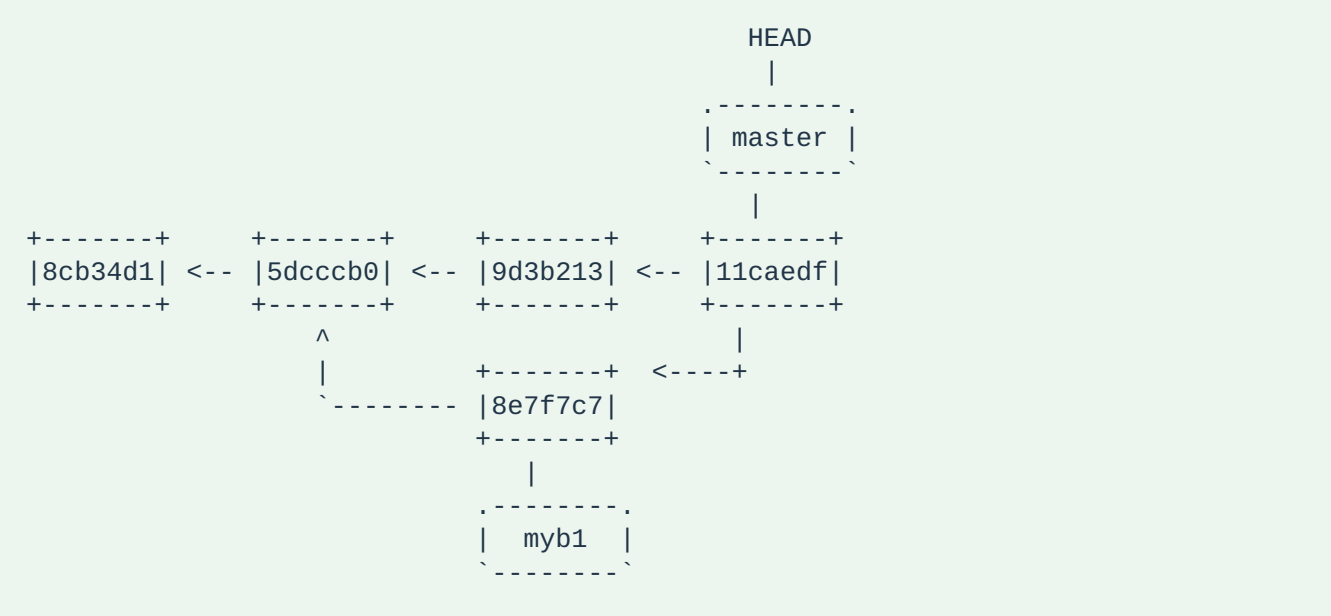
```
https://weimingze.com
https://weimingze.com/c/
https://weimingze.com/git/
```

至此 `myb1` 和 `master` 分支的最终结果已经合并到工作区，我们将所有文件加入到暂存区，然后提交到 `master` 分支。操作如下:

```
weimingze@mzstudio:~/my_project$ git add *
weimingze@mzstudio:~/my_project$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
    new file:   function1.txt
    modified:   website.txt

weimingze@mzstudio:~/my_project$ git commit -m "将功能1合并到 master分支"
[master 11caedf] 将功能1合并到 master分支
weimingze@mzstudio:~/my_project$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

可见在 master 分支上又创建了一个提交对象 `11caedf`，此提交对象是 `9d3b213` 和 `8e7f7c7` 合并后的结果。这两个提交对象也称之为 `11caedf` 的父对象。

此时的仓库内的结构如下:



使用 `git log --graph --abbrev-commit --oneline` 命令查看日志可以看到如下分支结构:

```
weimingze@mzstudio:~/my_project$ git log --graph --abbrev-commit
*   11caedf (HEAD -> master) 将功能1合并到 master分支
|\
| * 8e7f7c7 (myb1) 完成了功能1
* | 9d3b213 完成了功能2
|/
* 5dcccb0 (origin/master) 魏明择在当前项目中修改了 README.md 文件，并添加了 website.tx
t 文件
* 8cb34d1 魏明择在当前项目中添加了 README.md文件
```

上述合并后 `my_project`项目 可以点击此处进行下载。

实验:

1. 将 `master` 分支的功能2 在合并到 `myb1` 分支中，形成一个新的提交对象。
2. 查看 `master` 分支的提交日志。
3. 查看 `myb1` 分支的提交日志。
4. 删除 myb1 分支。

4. Git 可视化合并

在进行分支合并时常因为出现冲突而导致合并终止，并需要手动解决冲突问题。如上节课执行 `git merge myb1` 时出现冲突。提示如下：

```
weimingze@mzstudio:~/my_project$ git merge myb1
Auto-merging website.txt
CONFLICT (content): Merge conflict in website.txt
Automatic merge failed; fix conflicts and then commit the result.
```

其中 `website.txt` 文件出现了冲突，内容如下：

```
https://weimingze.com
<<<<<<< HEAD
https://weimingze.com/git/
=======
https://weimingze.com/c/
>>>>>>> myb1
```

在冲突比较少的情况下，上述文件可以手动修改后再改动提交合并。但文件比较多，冲突也比较多时。使用手动合并的方法就显得捉襟见肘，这时候最好使用 可视化的合并工具进行合并，比如 `Meld`、`Beyond Compare` 或 `vimdiff` 等。

在使用 `git merge` 合并因冲突而中断时，可以使用 `git mergetool` 命令调用已经设置好的可视化合并工具或使用 `-t` 选项临时启动一个可视化的合并工具进行合并。待合并完成后就可以使用 `git commit` 提交合并后的版本。

使用 `git mergetool --tool-help` 可以查询你当前版本的 `git` 所支持的基于图形的比较软件，如下所示：

```
weimingze@mzstudio:~/my_project$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        meld            Use Meld (requires a graphical session) with optional
`auto merge` (see `git help mergetool`'s `CONFIGURATION` section)
        vimdiff         Use Vim with a custom layout (see `git help
mergetool`'s `BACKEND SPECIFIC HINTS` section)
        vimdiff1        Use Vim with a 2 panes layout (LOCAL and REMOTE)
        vimdiff2        Use Vim with a 3 panes layout (LOCAL, MERGED and
REMOTE)
        vimdiff3        Use Vim where only the MERGED file is shown

The following tools are valid, but not currently available:
        araxis          Use Araxis Merge (requires a graphical session)
        bc              Use Beyond Compare (requires a graphical session)
        bc3             Use Beyond Compare (requires a graphical session)
        bc4             Use Beyond Compare (requires a graphical session)
        codecompare     Use Code Compare (requires a graphical session)
        deltawalker     Use DeltaWalker (requires a graphical session)
        diffmerge       Use DiffMerge (requires a graphical session)
        diffuse         Use Diffuse (requires a graphical session)
        ecmerge         Use ECMerge (requires a graphical session)
        emerge          Use Emacs' Emerge
        examdiff        Use ExamDiff Pro (requires a graphical session)
        guiffy          Use Guiffy's Diff Tool (requires a graphical session)
        gvimdiff        Use gVim (requires a graphical session) with a custom l
ayout (see `git help mergetool`'s `BACKEND SPECIFIC HINTS` section)
        gvimdiff1       Use gVim (requires a graphical session) with a 2 panes
layout (LOCAL and REMOTE)
        gvimdiff2       Use gVim (requires a graphical session) with a 3 panes
layout (LOCAL, MERGED and REMOTE)
        gvimdiff3       Use gVim (requires a graphical session) where only the
MERGED file is shown
        kdiff3          Use KDiff3 (requires a graphical session)
        nvimdiff        Use Neovim with a custom layout (see `git help
mergetool`'s `BACKEND SPECIFIC HINTS` section)
        nvimdiff1       Use Neovim with a 2 panes layout (LOCAL and REMOTE)
        nvimdiff2       Use Neovim with a 3 panes layout (LOCAL, MERGED and REM
OTE)
        nvimdiff3       Use Neovim where only the MERGED file is shown
        opendiff        Use FileMerge (requires a graphical session)
        p4merge         Use HelixCore P4Merge (requires a graphical session)
        smerge          Use Sublime Merge (requires a graphical session)
        tkdiff          Use TkDiff (requires a graphical session)
        tortoisemerge   Use TortoiseMerge (requires a graphical session)
        winmerge        Use WinMerge (requires a graphical session)
        xxdiff          Use xxdiff (requires a graphical session)
```

```
Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

上述显示当前可以直接设置并使用的是 `meld` 和 `vimdiff`，支持的合并软件有 `Araxis Merge`、`Beyond Compare` 等。

`git mergetool` 命令

作用： 用于在使用 `git merge` 合并发生冲突时调用可视化工具手动对冲突内容进行合并。

命令格式

```
git mergetool [选项] [提交1] [提交2] [--] [路径]
```

常用选项

选项	说明
`--tool=<可视化工具命令>` 或 `-t <可视化工具命令>`	使用 **可视化工具命令** 代替默认 `merge.tool` 的设置可视化工具对冲突进行合并。
`-y` 或 `--no-prompt`	不提示确认，直接调用 `--tool` 选项指定或 `merge.tool` 设置的可视化工具合并。

示例：

1、 使用 `meld` 解决并冲突：`git mergetool -t meld`

执行结果如下：

```
weimingze@mzstudio:~/my_project$ git mergetool -t meld
Merging:
website.txt

Normal merge conflict for 'website.txt':
  {local}: modified file
  {remote}: modified file
```

图形用户界面如下:

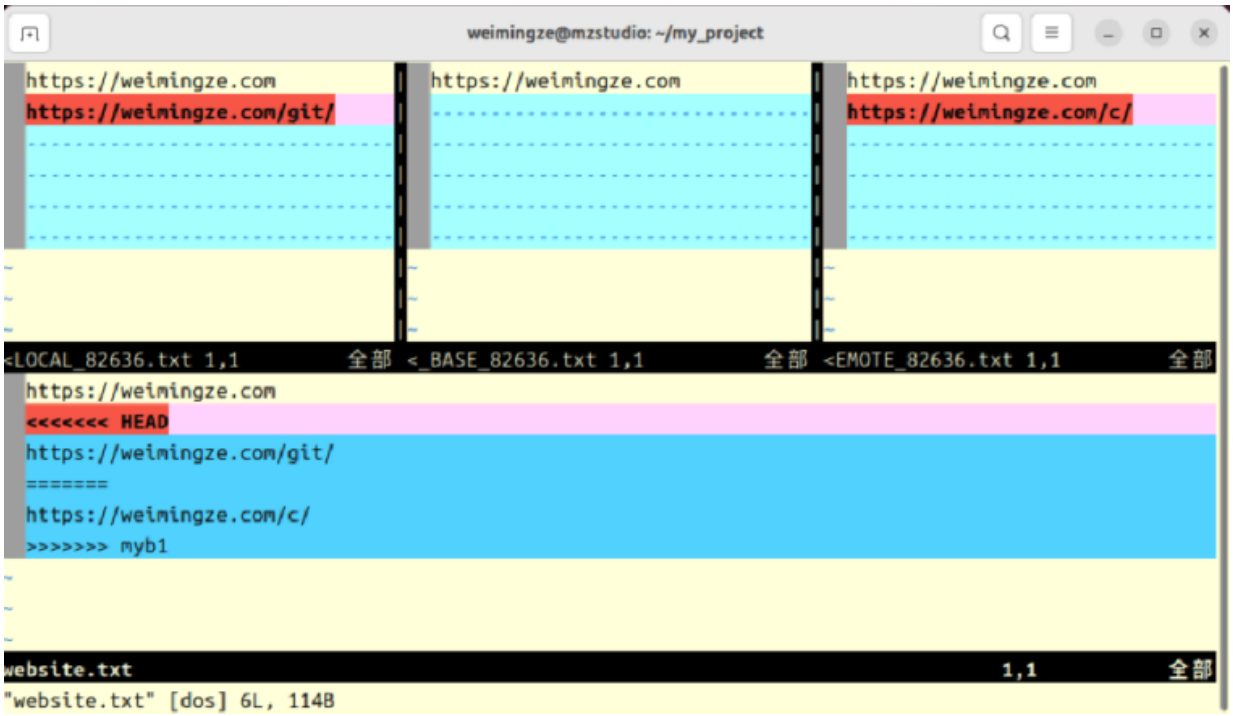
上图中，中间的编辑内容是最终合并后需要提交的内容。左侧内容是合并前当前分支中 `website.txt` 中的内容，右侧是合并前 `myb1` 分支中 `website.txt` 文件中的内容。编辑中间的内容，然后保存退出即可。

2、 使用 `vimdiff` 解决合并冲突：`git mergetool -t vimdiff`

执行结果如下：

```
weimingze@mzstudio:~/my_project$ git mergetool -t vimdiff
Merging:
website.txt

Normal merge conflict for 'website.txt':
  {local}: modified file
  {remote}: modified file
4 files to edit
```

图形用户界面如下:



在上面的操作中，使用 `ctrl + w` 然后再 `ctrl + w` 即可以在不同编辑框中切换光标。上图中上面中间内容就是我们最终合并后的结果。编辑此文档，然后保存退出即可。

总之，使用 mergetool 对文件文件进行编辑更加直观，不容易出错。

实验：

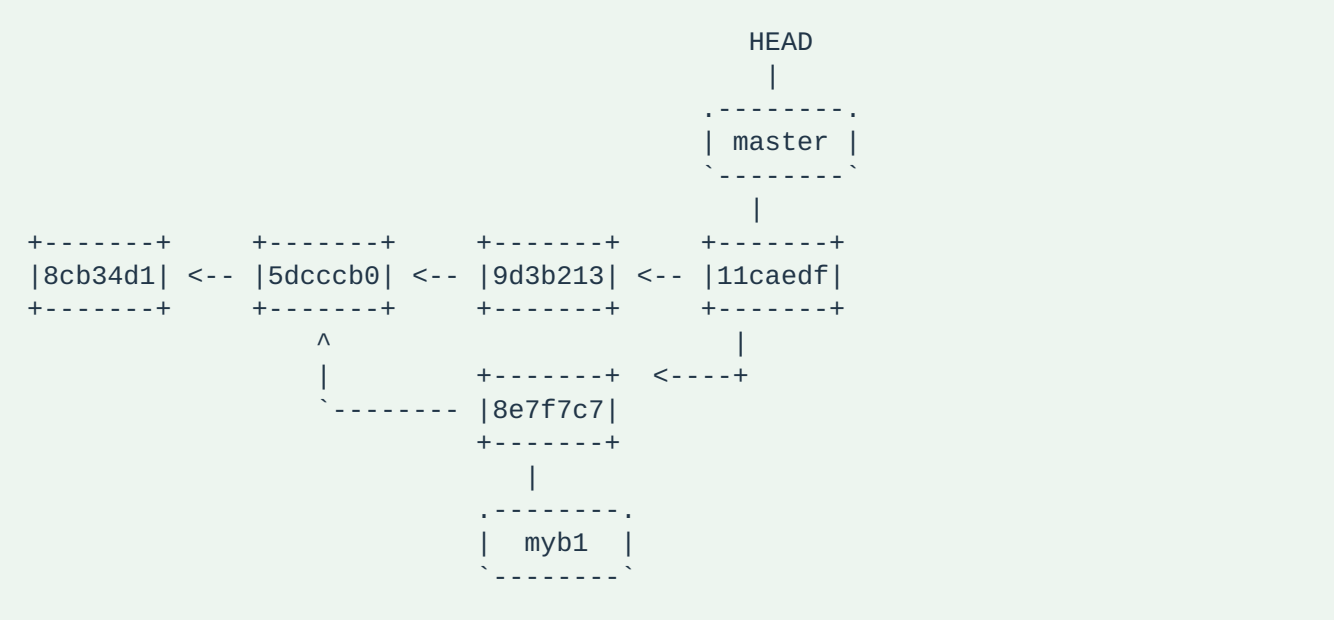
尝试使用 `meld` 或 `vimdiff` 可视化工具来解决合并中的冲突。

第五章、高级用法

1. HEAD 指针

在 Git 版本控制系统中总有一个指向当前操作位置的 **HEAD 指针**，这个指针指向当前分支的分支指针。

这里以之前 经过两个分支合并后的 `my_project` 项目的仓库为例。此时 `HEAD` 指向 `master`， 而 `master` 指向 `11caedf`，即如果再有提交，则提交到 `11caedf` 提交对象之后。其分支结构如下所示：



默认情况下 HEAD 代表当前分支的最后提交对象。我们使用 HEAD 可以取出提交对象的快照。

在使用 `git restore` 命令恢复工作区时可以使用 `--source=<提交对象>` 恢复指定提交对象带边的工作区。这个提交对象也可以用 `HEAD` 或相对于 `HEAD` 的提交对象。

提交对象位置的写法

1. 使用提交哈希值表示具体的提交对象。
   - 长哈希值，如：`11caedf3fd3c82f930421ee319bebab3c5822abe`
   - 短哈希值，如：`11caedf`
2. 使用 `HEAD` 表示当前位置。
3. 使用 `HEAD^n` 表示当前提交对象的第 n 个父对象（合并前的第 n 个提交对象）。
   - 如：`HEAD^1` 等同于 `HEAD^` 表示 `11caedf` 的第一个父对象 `9d3b213`

- 如：`HEAD^2` 表示 `11caedf` 的第二个父对象 `8e7f7c7`（因为 `11caedf` 是由两个对象合并得到的，因此它有两个父对象）。
- 如：`HEAD^2^1` 表示 `11caedf` 的第二个父对象 `8e7f7c7` 的第一个父对象 `5dcccb0`。

4. 使用 `HEAD~` 或 `HEAD~n` 表示当前分支上的第 n 个父对象。
   - 如：`HEAD~1` 等同于 `HEAD~` 表示 `11caedf` 的第一个父对象 `9d3b213`。
   - 如：`HEAD~2` 等同于 `HEAD~~` 表示 `11caedf` 的父对象 `9d3b213`的父对象 `5dcccb0`。

示例：

以使用 `git restore` 恢复项目 `my_project` 的工作区为例:

1. `git restore --source=HEAD .` 等同于 `git restore --source=11caedf .` 和 `git restore .`
2. `git restore --source=HEAD~ .` 或 `git restore --source=HEAD~1 .` 等同于 `git restore --source=9d3b213 .`
3. `git restore --source=HEAD^2 .` 等同于 `git restore --source=8e7f7c7 .`
4. `git restore --source=HEAD~3 .` 等同于 `git restore --source=8cb34d1 .`

执行后的效果如下:

```
weimingze@mzstudio:~/my_project$ ls
README.md  function1.txt  function2.txt  website.txt
weimingze@mzstudio:~/my_project$ git restore --source=HEAD~ .
weimingze@mzstudio:~/my_project$ ls
README.md  function2.txt  website.txt
weimingze@mzstudio:~/my_project$ git restore --source=HEAD^2 .
weimingze@mzstudio:~/my_project$ ls
README.md  function1.txt  website.txt
weimingze@mzstudio:~/my_project$ git restore --source=HEAD~3 .
weimingze@mzstudio:~/my_project$ ls
README.md
```

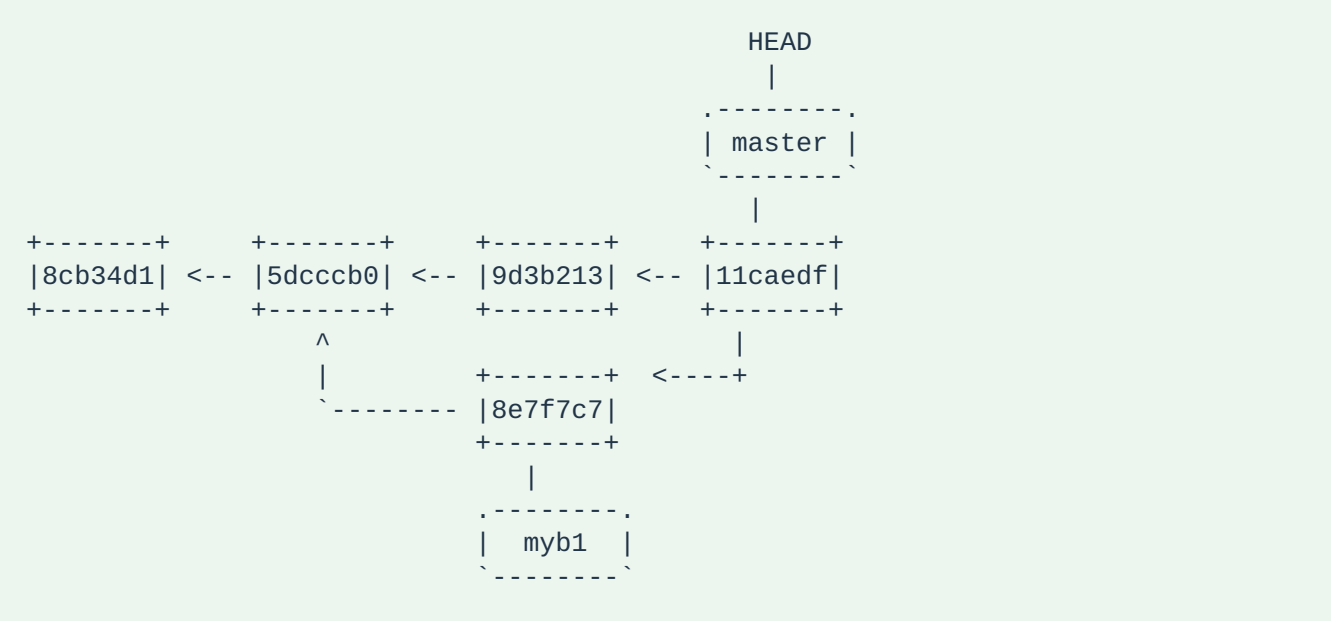
练习：

在 `my_project` 项目中使用 `HEAD` 指针将工作区的文件恢复到任意一个提交时的状态。

2. 重置 HEAD 指针

在使用 Git 进行版本控制时，每次提交后 HEAD 和当前的分支指针都指向最后一次提交的提交对象。下一次提交会将此刚提交的提交对象作为下一此提交的父对象。如果你希望丢弃本地提交。你的提交从当前的父提交重新创建一个新的提交，你可以使用 `get reset` 命令重置 HEAD 指针的位置。

之前我们已经完成了 `my_project` 的分支合并。最终仓库结构如下:



如果我们在主分支上再次进行提交则提交对象会放在 `11caedf` 这个提交对象之后。新创建的提交对象会指向 `11caedf` 让其作为父对象。如果我们希望丢弃 `11caedf` 这次提交，那我们可以将 `HEAD` 指针指向 `9d3b213` 这个提交对象，然后再次提交则新创建的提交对象会以 `9d3b213` 作为父对象。这这样 `master` 分支就丢弃了 `11caedf`, 此对象成为了**悬空对象**。此悬空对象在一定时间后会被 Git 的垃圾回收机制回收，从而达到丢弃 `11caedf` 这次提交的目的。

`git reset` 命令

作用： 移动当前分支的 `HEAD` 指针到指定的提交对象。并可以选择是否重置暂存区和工作区。

命令格式

```
git reset [选项] 提交对象位置
```

> 注意：这是一个比较危险的命令，请谨慎使用。

`git reset` 的常用选项

选项	说明
`--mixed`	移动 HEAD 指针，重置暂存区，不修改工作区。（默认）
`--soft`	移动 HEAD 指针，不修改暂存区和工作区。
`--hard`	移动 HEAD 指针，重置暂存区和工作区（未提交的修改会丢失，危险!）。
`--merge`	移动 HEAD 指针，重置暂存区，保留工作区中未提交的更改与重置提交之间的冲突。
`--keep`	如果工作区的可能更改，则终止执行当前命令。否则移动 HEAD 指针，重置暂存区。
	其它选项
`--no-refresh`	移动 HEAD 指针后，暂存区不变。
`-- 路径`	重置指定路径的文件

示例：

1、 使用 `git reset --hard` 将之前工作区和暂存区的内容都清空，HEAD 指针位置不变，等同于 `git restore --staged --worktree .`，如：

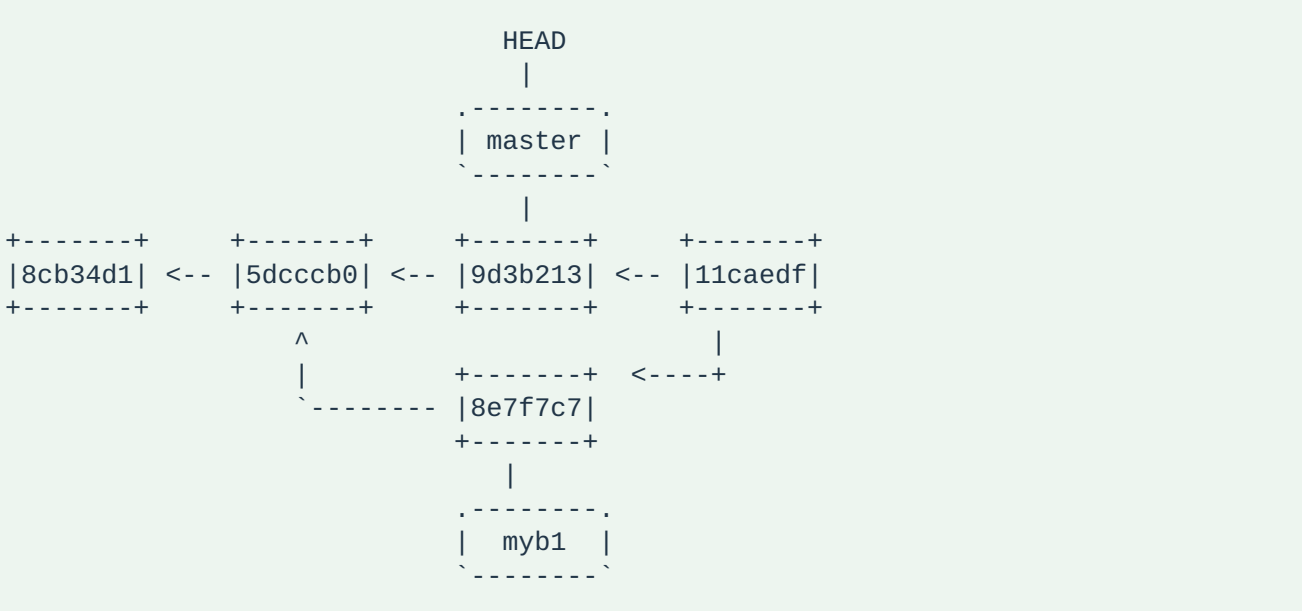
```
weimingze@mzstudio:~/my_project$ git log --graph --all --oneline
*   11caedf (HEAD -> master) 将功能1合并到 master分支
|\
| * 8e7f7c7 (myb1) 完成了功能1
* | 9d3b213 完成了功能2
|/
* 5dcccb0 (origin/master) 魏明择在当前项目中修改了 README.md 文件，并添加了 website.tx
t 文件
* 8cb34d1 魏明择在当前项目中添加了 README.md文件
weimingze@mzstudio:~/my_project$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   website.txt

weimingze@mzstudio:~/my_project$ git reset --hard
HEAD is now at 11caedf 将功能1合并到 master分支
weimingze@mzstudio:~/my_project$ git log --graph --all --oneline
*   11caedf (HEAD -> master) 将功能1合并到 master分支
|\
```

```
| * 8e7f7c7 (myb1) 完成了功能1
* | 9d3b213 完成了功能2
|/
* 5dcccb0 (origin/master) 魏明择在当前项目中修改了 README.md 文件，并添加了 website.tx
t 文件
* 8cb34d1 魏明择在当前项目中添加了 README.md文件
weimingze@mzstudio:~/my_project$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

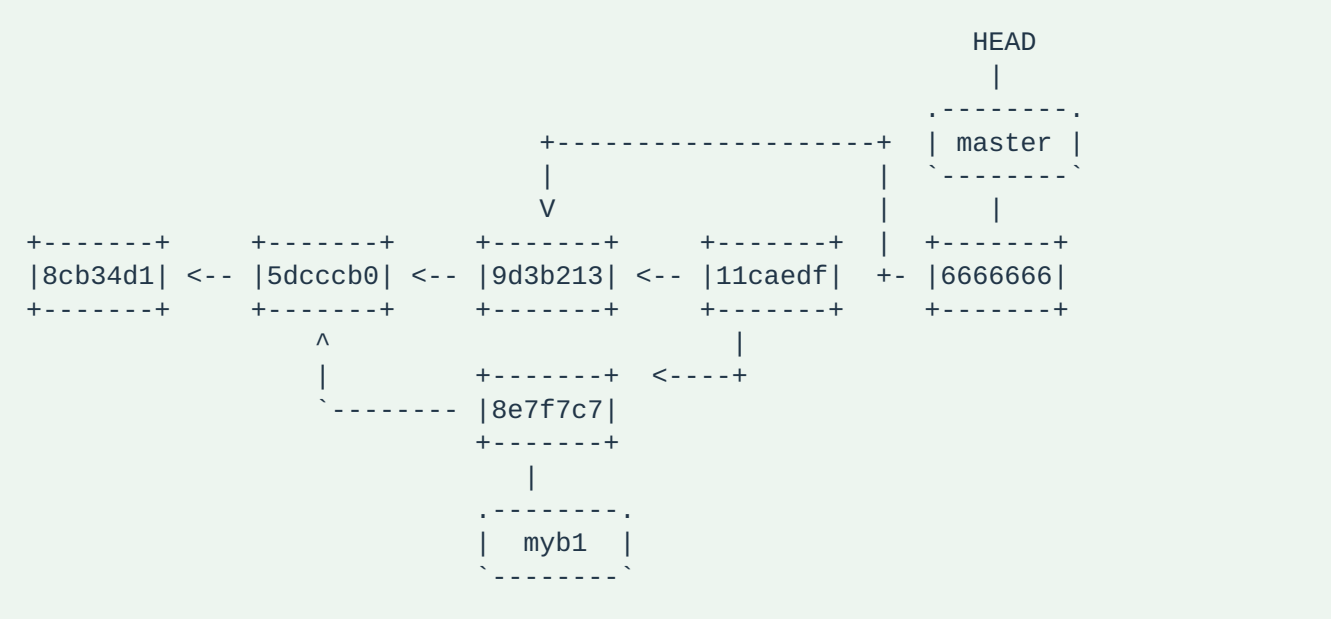
2、 使用 `git reset --hard HEAD~1` 将 当前指针重置到 `9d3b213`，如：

```
weimingze@mzstudio:~/my_project$ git reset --hard HEAD~1
HEAD is now at 9d3b213 完成了功能2
weimingze@mzstudio:~/my_project$ ls
README.md  function2.txt  website.txt
weimingze@mzstudio:~/my_project$ git log --graph --all --oneline
* 9d3b213 (HEAD -> master) 完成了功能2
| * 8e7f7c7 (myb1) 完成了功能1
|/
* 5dcccb0 (origin/master) 魏明择在当前项目中修改了 README.md 文件，并添加了 website.tx
t 文件
* 8cb34d1 魏明择在当前项目中添加了 README.md文件
```

可见当前 `HEAD` 指针指向了 `9d3b213` 提交对象。此时使用 `git log` 命令已经看不到最后一次合并后的节点 `11caedf`。目前的仓库状态如下所示：



假如此时在当前分支 `master` 上创建新的提交 `6666666`，则此提交对象的父节点则是 `9d3b213`。如下图所示



如果我们需要将 `HEAD` 指针再次重定位到之前合并后的节点 `11caedf`， 则必须使用 `git reset 11caedf` 命令来实现。因为此时已经无法通过 `HEAD` 找到它的位置了。

引用日志

在上述示例中，在移动HEAD 指针后，就再也无法通过 `git log` 等命令看到合并后的提交对象 `11caedf`，也无法恢复本次提交后的内容。那我们如何查找到我们之前创建过的提交对象的历史记录呢。如果你有这样的需求，那你需要使用 `git reflog` 命令来查看引用日志了。

引用日志（简称 "reflogs"）记录了分支头指针及其它引用在本地仓库中更新的时间点。它记录了仓库中 HEAD 指针的所有移动历史，可以帮助你找回 **丢失** 的提交、分支或更改。

`git reflog` 命令

作用： 用于查看和管理**引用日志**中的记录信息。

命令格式

```
git reflog [选项] [引用信息]
```

示例：

查看 `git reset --hard HEAD~1` 后的引用日志

```
weimingze@mzstudio:~/my_project$ git reflog
9d3b213 (HEAD -> master) HEAD@{0}: reset: moving to HEAD~1
```

```
11caedf HEAD@{1}: commit (merge): 将功能1合并到 master分支
9d3b213 (HEAD -> master) HEAD@{2}: commit (amend): 完成了功能2
2d700a1 HEAD@{3}: commit: 完成了功能1
5dcccb0 (origin/master) HEAD@{4}: checkout: moving from myb1 to master
8e7f7c7 (myb1) HEAD@{5}: commit: 完成了功能1
5dcccb0 (origin/master) HEAD@{6}: checkout: moving from master to myb1
5dcccb0 (origin/master) HEAD@{7}: checkout: moving from myb1 to master
...
```

上述日志中，第一列是提交对象的 哈希值，如 `9d3b213`。括号中是分支信息。HEAD@{n} 表示距离当前操作的第n次 HEAD指针的变动。冒号后面是具体的操作信息。

如果我们希望将 `HEAD` 再次指向 `11caedf HEAD@{1}: commit (merge): 将功能1合并到 master分支` 这次操作的位置。则我们可以使用 `git reset 11caedf` 或 `git reset HEAD@{1}` 来实现此操作。

实验：

使用 `git reset` 命令任意改动 `HEAD` 指针的位置，然后通过 `git reflog` 查看引用日志。最后通过引用日志中的信息，使用 `git reset` 重新将 `HEAD` 指针定位到你去到某个位置之前的位置。

3. Git 版本比较

在编写文档时，往往需要对比自己的修改情况，经过比较当前修改或过去的某些修改之间的差异来找出问题存在的原因。这时候你可以使用 `git diff` 命令达到上述目的。

`git diff` 命令

作用： 查看工作区和暂存区、工作区和最后一次提交以及过去的两次提交之间的改动情况。它可以比较不同提交、分支、标签之间的差异。

命令格式:

```
git diff [选项] [提交1] [提交2] [--] [路径]
```

说明：

1. 当**提交2**省略不写时，使用**路径**对比暂存区或当前 HEAD 提交对象的内容。

`git diff` 的常用选项

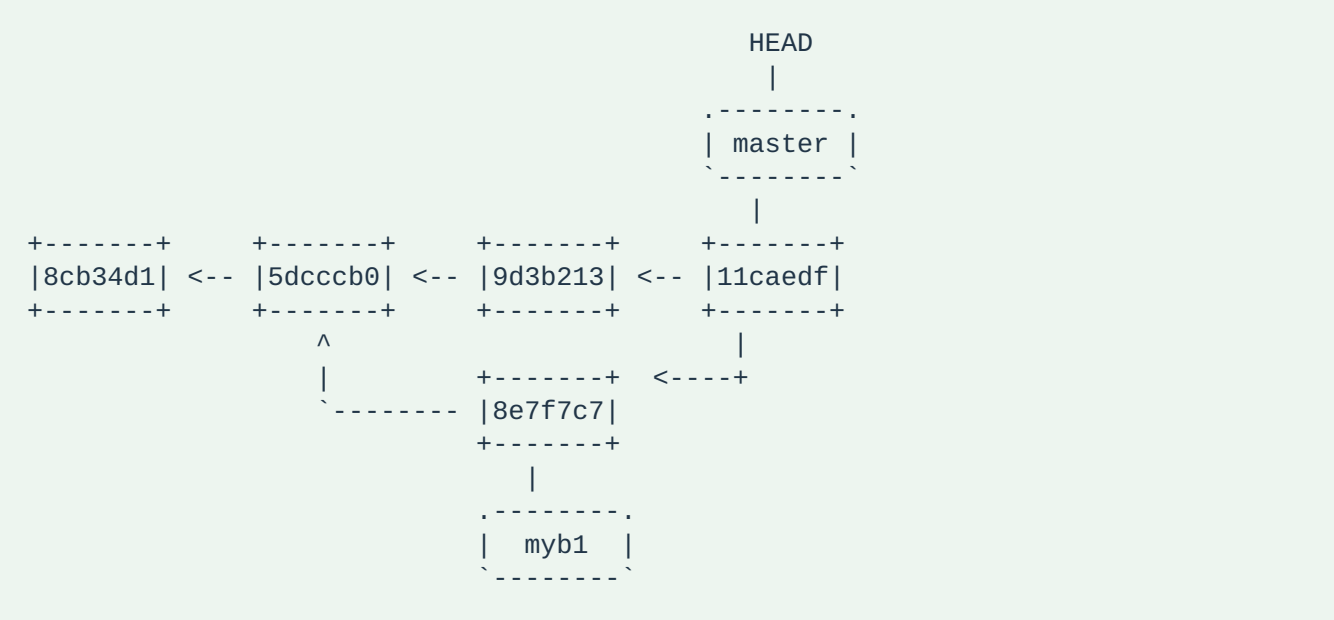
选项	说明
`--staged` 或 `--cached`	比较暂存区（如果暂存）和 HEAD 之间的差别。

命令示例：

- `git diff --staged` 或 `git diff --cached` 是比较当前暂存区和最后提交 `HEAD` 之间的差别。
- `git diff` 是比较当前工作区和暂存区之间的差别，如果暂存区不存起则比较最后提交 `HEAD` 之间的差别。
- `git diff 8e7f7c7` 是比较当前工作区和提交 `8e7f7c7` 之间的差别。
- `git diff v0.1 v0.2` 是比较两个标签之间的差别。
- `git diff 8e7f7c7 9d3b213` 是比较两个提交之间的差别。
- `git diff HEAD~1 HEAD~2` 是比较当前提交的前一次提交和前两次提交之间的差别。

示例：

以当前的 `my_project` 为例，当前的仓库结构如下：



其中每个提交的节点的内容如下:

1) `8cb34d1` 这次提交增加了文件 `README.md` 2) `5dcccb0` 这次提交增加了文件 `website.txt` 其内容是

```
https://weimingze.com
```

3) `8e7f7c7` 这次提交增加了文件 `function1.txt` 并修改 `website.txt` 的内容为


```
https://weimingze.com
https://weimingze.com/c/
```

4) `9d3b213` 这次提交增加了文件 `function2.txt` 并修改 `website.txt` 的内容为

```
https://weimingze.com
https://weimingze.com/git/
```

5) `11caedf` 这次提交合并和 **3** 和 **4** 两次提交，并修改 `website.txt` 的内容为

```
https://weimingze.com
https://weimingze.com/c/
https://weimingze.com/git/
```

现在修改工作区中 `website.txt` 的内容为

```
https://weimingze.com
aaaaaaaa
https://weimingze.com/c/
https://weimingze.com/git/
```

然后通过 `git add website.txt` 添加到 暂存区，然后再次修改工作区的 `website.txt` 内容如下:

```
https://weimingze.com
aaaaaaaa
https://weimingze.com/c/
bbbbbbbb
https://weimingze.com/git/
```

1、 比较暂存区和 HEAD 之间的差别：`git diff --staged`

```
weimingze@mzstudio:~/my_project$ git diff --staged
diff --git a/website.txt b/website.txt
index ee1ef1a..82fe119 100644
--- a/website.txt
+++ b/website.txt
@@ -1,3 +1,4 @@
 https://weimingze.com
+aaaaaaaa^M
 https://weimingze.com/c/
 https://weimingze.com/git/
```

上述输出内容中、`--- a/website.txt` 表示原始文件（HEAD中的文件） `+++ b/website.txt` 表示修改后的文件（暂存区的文件）。`@@ -1,3 +1,4 @@` 表示修改的内容。`-1,3` 表示原始文件的第 1

行开始的 3 行。`+1,4` 表示修改后的文件(暂存区的文件)的第 1 行开始的 4 行。`+aaaaaaaa^M` 表示修改后比修改前增加的一行 `aaaaaaaa` 后面的 `^M` 表示这一行以 Windows 的换行 `CRLF` 结尾。修改内容中第一个字符是 `+` 表示增加的一行，`-` 表示修改后删除的行，空格表示不变。

2、 比较工作区和暂存区 之间的差别：`git diff`

```
weimingze@mzstudio:~/my_project$ git diff
diff --git a/website.txt b/website.txt
index 82fe119..c896361 100644
--- a/website.txt
+++ b/website.txt
@@ -1,4 +1,5 @@
 https://weimingze.com
 aaaaaaaa
 https://weimingze.com/c/
+bbbbbbbb^M
 https://weimingze.com/git/
weimingze@mzstudio:~/my_project$
```

可见当前工作区比暂存区多了一行 `bbbbbbbb`。

3、 比较工作区和 HEAD 之间的差别：`git diff HEAD`

```
weimingze@mzstudio:~/my_project$ git diff HEAD
diff --git a/website.txt b/website.txt
index ee1ef1a..c896361 100644
--- a/website.txt
+++ b/website.txt
@@ -1,3 +1,5 @@
 https://weimingze.com
+aaaaaaaa^M
 https://weimingze.com/c/
+bbbbbbbb^M
 https://weimingze.com/git/
```

可见当前工作区比 `HEAD` 提交 多了 2 行 `aaaaaaaa` 和 `bbbbbbbb`。

4、 比较 `9d3b213` 和 `8e7f7c7` 之间的差别：`git diff 9d3b213 8e7f7c7` 或 `git diff HEAD^1 HEAD^2`

```
weimingze@mzstudio:~/my_project$ git diff HEAD^1 HEAD^2
diff --git a/function1.txt b/function1.txt
new file mode 100644
index 0000000..2f3b80f
--- /dev/null
+++ b/function1.txt
@@ -0,0 +1 @@
+新功能1的实现代码^M
```

```
diff --git a/function2.txt b/function2.txt
deleted file mode 100644
index 7dc6e91..0000000
--- a/function2.txt
+++ /dev/null
@@ -1 +0,0 @@
-新功能2的实现代码
diff --git a/website.txt b/website.txt
index c69f01b..2ddf62f 100644
--- a/website.txt
+++ b/website.txt
@@ -1,2 +1,2 @@
 https://weimingze.com
-https://weimingze.com/git/
+https://weimingze.com/c/^M
```

可见合并之前，提交`9d3b213`没有文件`function1.txt`（`/dev/null`表示没有此文件），提交`8e7f7c7`没有文件`function2.txt`。

实验：

使用`git diff`比较各个版本的不同。

4. Git 可视化版本比较

在使用`git diff`比较版本差异时，提示的字符不宜阅读。我们可以使用`git difftool`命令代替`git diff`。`git difftool`可以启用提前设置好的可视化的比较软件进行比较，也可以使用`-t`选项临时设置比较软件。

使用`git difftool --tool-help`可以查询你当前版本的`git`所支持的基于图形的比较软件，如下所示：

```
weimingze@mzstudio:~$ git difftool --tool-help
'git difftool --tool=<tool>' may be set to one of the following:
        meld            Use Meld (requires a graphical session)
        vimdiff         Use Vim

The following tools are valid, but not currently available:
        araxis          Use Araxis Merge (requires a graphical session)
        bc              Use Beyond Compare (requires a graphical session)
        bc3             Use Beyond Compare (requires a graphical session)
        bc4             Use Beyond Compare (requires a graphical session)
        codecompare     Use Code Compare (requires a graphical session)
        deltawalker     Use DeltaWalker (requires a graphical session)
        diffmerge       Use DiffMerge (requires a graphical session)
        diffuse         Use Diffuse (requires a graphical session)
        ecmerge         Use ECMerge (requires a graphical session)
        emerge          Use Emacs' Emerge
        examdiff        Use ExamDiff Pro (requires a graphical session)
```



```
        guiffy          Use Guiffy's Diff Tool (requires a graphical session)
        gvimdiff        Use gVim (requires a graphical session)
        kdiff3          Use KDiff3 (requires a graphical session)
        kompare         Use Kompare (requires a graphical session)
        nvimdiff        Use Neovim
        opendiff        Use FileMerge (requires a graphical session)
        p4merge         Use HelixCore P4Merge (requires a graphical session)
        smerge          Use Sublime Merge (requires a graphical session)
        tkdiff          Use TkDiff (requires a graphical session)
        winmerge        Use WinMerge (requires a graphical session)
        xxdiff          Use xxdiff (requires a graphical session)

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

上述显示当前可以直接设置并使用的是 `meld` 和 `vimdiff`，支持的比较软件有 `Araxis Merge`、`Beyond Compare` 等。

`git difftool` 命令

作用： 同 `git diff` 一样，它会在图形用户界面启动相应程序进行比较

命令格式

```
git difftool [选项] [提交1] [提交2] [--] [路径]
```

常用选项

选项	说明
`-t <图形工具启动命令>`	使用 启动命令代替默认设置的比较软件对版本进行比较。
`--staged` 或 `--cached`	比较暂存区（如果暂存）和 HEAD 之间的差别。

示例：

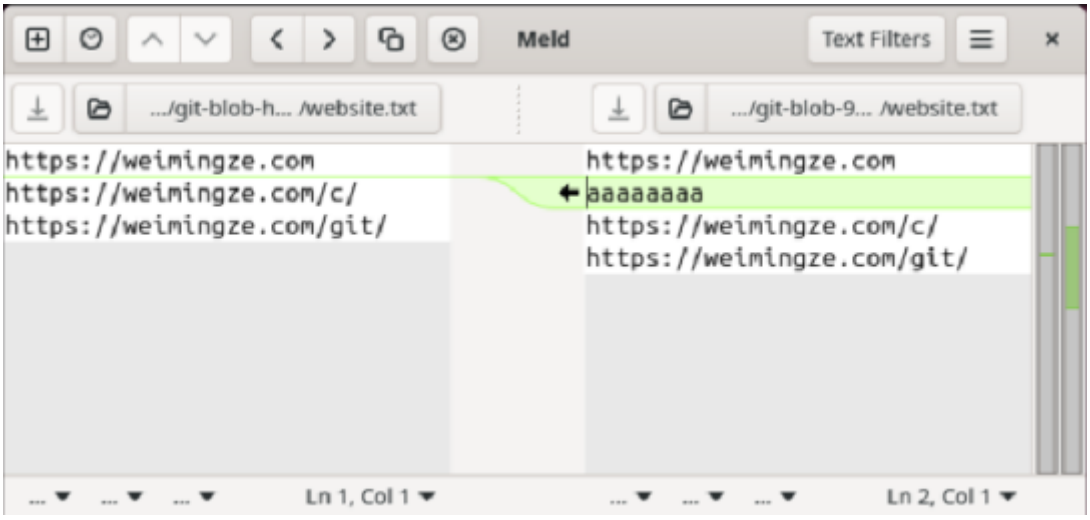
1、 使用 `meld` 比较暂存区和 HEAD 之间的差别：`git difftool --staged -t meld`

执行结果如下：

```
ze@mzstudio:~/my_project$ git difftool -t meld --staged
```

```
Viewing (1/1): 'website.txt'
Launch 'meld' [Y/n]? y
```

图形用户界面如下:

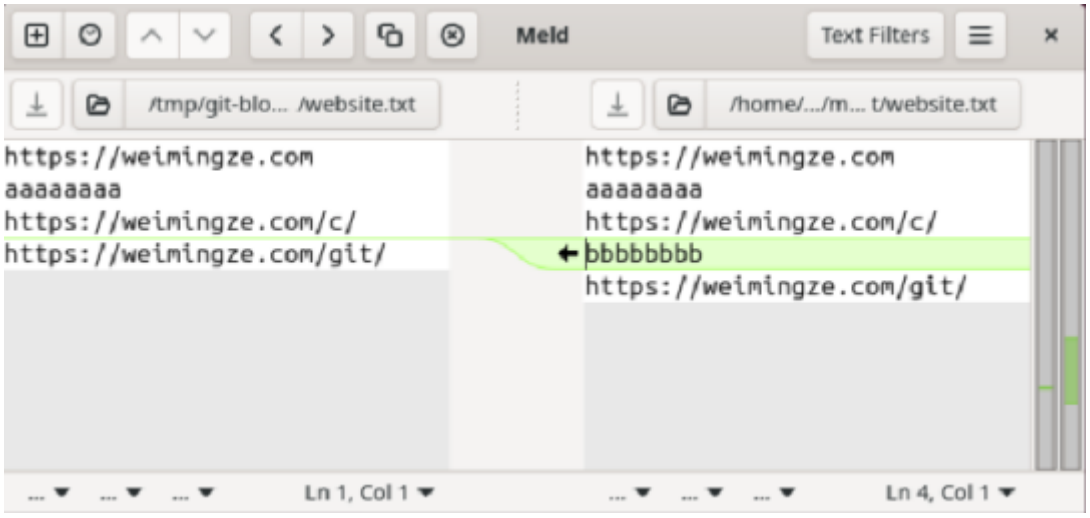


2、 使用 `meld` 比较工作区和暂存区 之间的差别：`git difftool -t meld`

执行结果如下：

```
weimingze@mzstudio:~/my_project$ git difftool -t meld

Viewing (1/1): 'website.txt'
Launch 'meld' [Y/n]? y
```

图形用户界面如下:

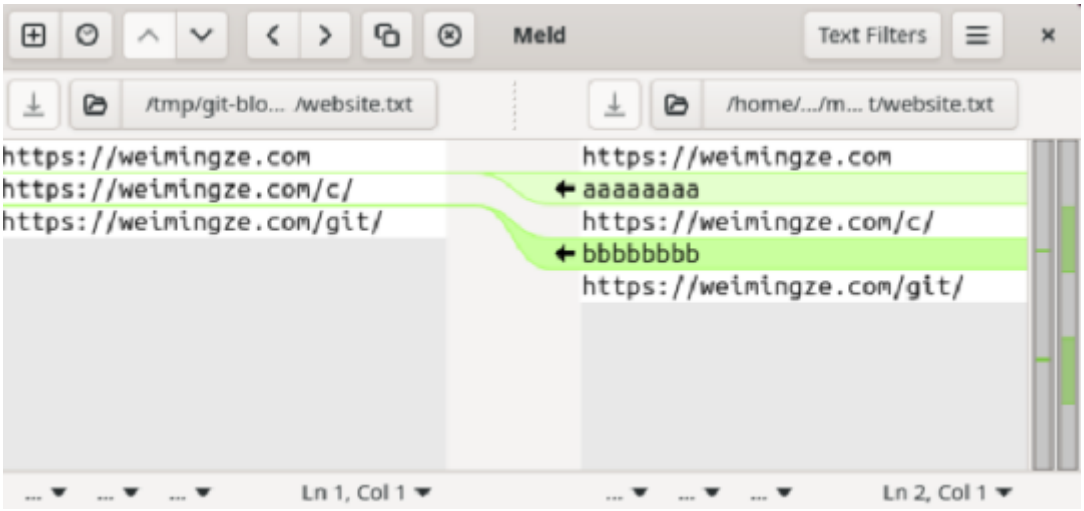


3、 使用 `meld` 比较工作区和 HEAD 之间的差别：`git difftool HEAD -t meld`

执行结果如下：

```
weimingze@mzstudio:~/my_project$ git difftool HEAD -t meld .

Viewing (1/1): 'website.txt'
Launch 'meld' [Y/n]? y
```

图形用户界面如下:



4、 使用 `meld` 比较 `9d3b213` 和 `8e7f7c7` 之间的差别：`git difftool 9d3b213 8e7f7c7 -t meld` 或 `git difftool HEAD^1 HEAD^2 -t meld`

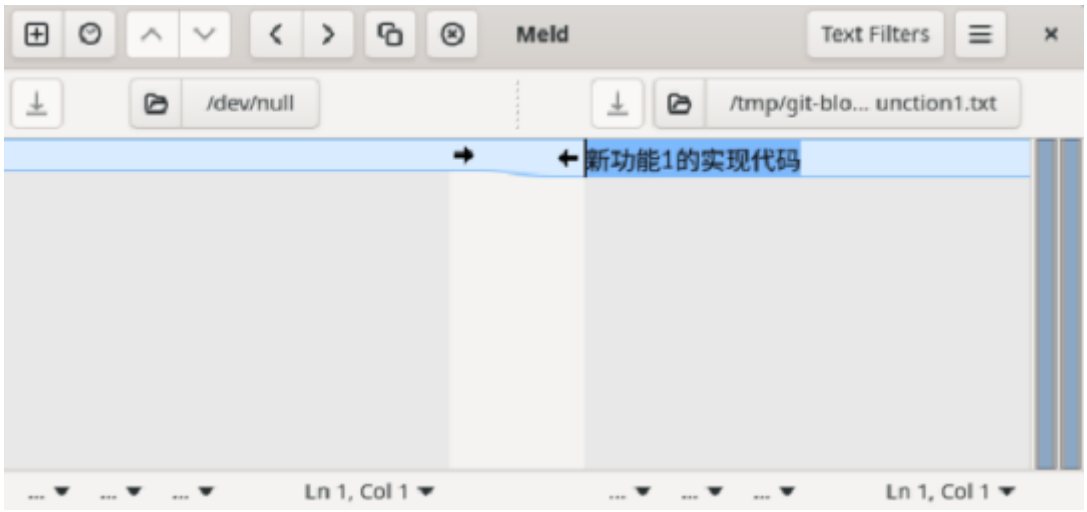
执行结果如下:

```
weimingze@mzstudio:~/my_project$ git difftool HEAD^1 HEAD^2 -t meld

Viewing (1/3): 'function1.txt'
Launch 'meld' [Y/n]? y

Viewing (2/3): 'function2.txt'
Launch 'meld' [Y/n]? y

Viewing (3/3): 'website.txt'
Launch 'meld' [Y/n]? y
```

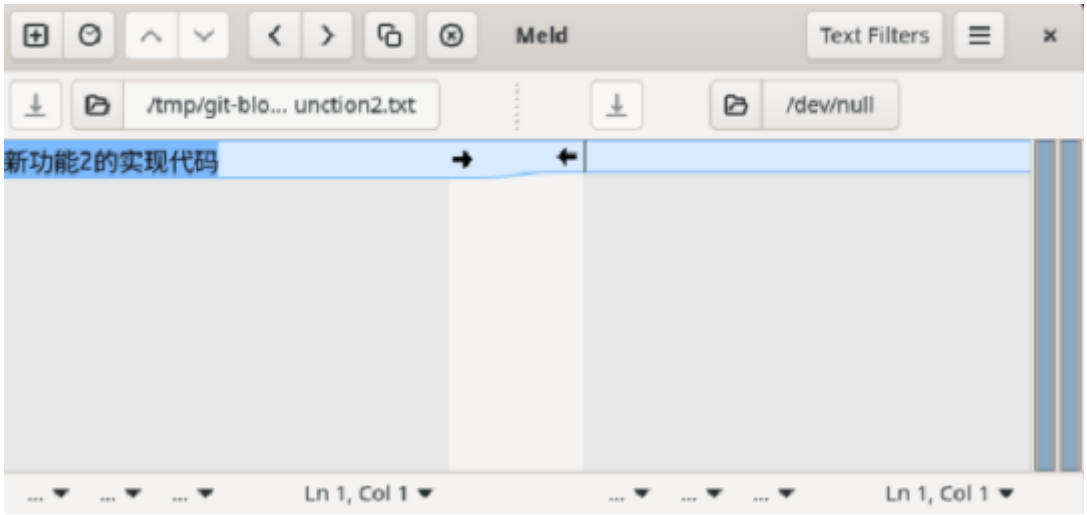
上面会提示三次是否打开比较，输入 `y` 后确认打开则会打开 `meld` 进行比较。

图形用户界面如下:

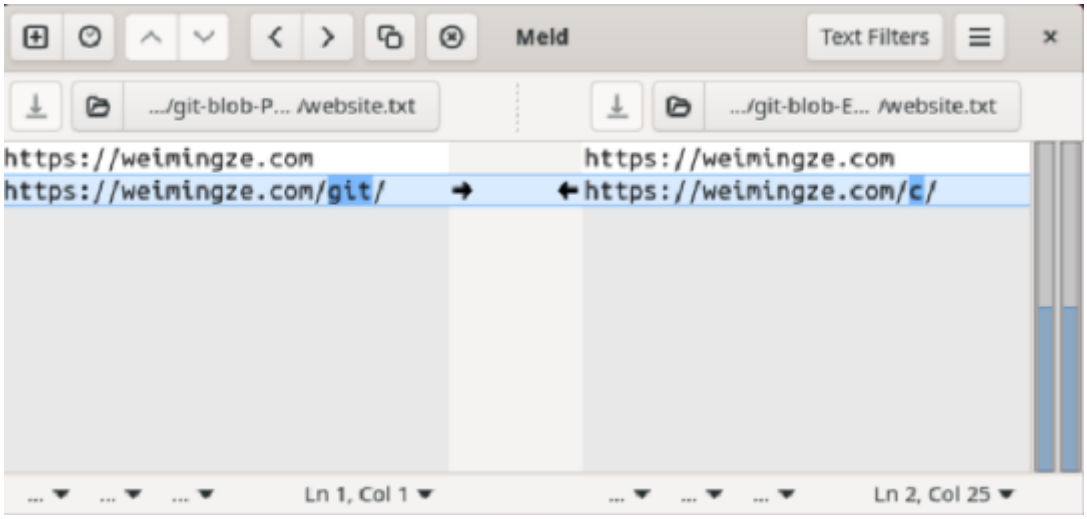
第 1 次确认比较 `function1.txt`:



第 2 次确认比较 `function2.txt`:



第 3 次确认比较 `website.txt`:



实验：

1. 下载 meld，下载地址： http://meldmerge.org/。
2. 安装 meld。

3. 设置 meld 为默认 `git difftool` 的比较工具。

5. Git 变基操作

变基 是指将一个分支开始的一系列提交对象或从某个提交对象开始的一系列提交对象的所有提交重新应用到一个新的提交对象上重新提交以实现整理提交结果的目的。

变基的目的：

1. 合并提交历史：将当前分支的提交转入到目标分支的最新提交之后，形成一条直线的提交历史，方便提交的远程分支中，并能保留分支的提交历史。
2. 整理本地提交：将本地的多个分支整理成一个分支。
3. 合并或删除同一个分支之前的提交节点。

在 Git 中整合来自不同分支的修改主要有两种方法：`git merge`（合并）和 `git rebase`（变基）。

变基操作需要使用 `git rebase` 命令。

`git rebase` 命令

作用： 在另一个分支上重新应用提交。

命令格式

```
git rebase [选项] [旧基准] [基底分支]
```

说明

1. **基底分支**是要作为变基后作为基底的分支，新分支以此分支的最后一个提交对象作为基础。默认是当前分支。
2. **旧基准** 可能是 **需要变基分支** 或者 **提交对象**。
   - 旧基准 是**需要变基分支**时，表示该分支从**基底分支**分叉处的所有提交。
   - 旧基准 是**提交对象**时，表示该提交对象到 **基底分支** 的所有提交。

常用选项

选项	说明
`-i` 或 `--interactive`	交互式变基，重新提交历史。
`--onto <新基底>`	变基到某个新基底（某个分支或提交对象）。
`--abort`	中止变基，恢复到变基之前的状态。
`--continue`	继续没有完成的变基操作。
`--skip`	跳过导致合并冲突的提交。

下面我们举例说明变基的思想和变基的方法。

示例1:

现在我们有如下图的六次提交 `A`、`B`、`C`、`D`、`E`、`F`，其中每次提交 都会提交一个文件。如: `A` 这次提交增加一个 `a.txt`， `A` 这次提交增加一个 `b.txt`，以此类推。

```
A---B---C  master
     \
      D---E---F  myb2
```

现在分支 `master` 的最终状态是有三个文件 `a.txt`、`b.txt`、`c.txt`，而 `myb2` 分支的最终状态则有五个文件 `a.txt`、`b.txt`、`d.txt`、`e.txt`、`f.txt`。

`myb2` 分支是在 `B` 提交后进行的分支，因此 `myb2` 分支不存在 `c.txt` 这个文件。

下载上述示例仓库

将 `myb2` 分支变基到 `master` 分支

如果此时你在 `myb2` 分支中，你可以使用 `git rebase master` 命令将 `myb2` 分支变基到 `master` 分支。

如果你不在 `myb2` 分支中，你可以使用 `git rebase master myb2` 命令同样可以将 `myb2` 分支变基到 `master` 分支。

变基完成后会切换到 `myb2` 分支。

执行结果如下:

```
$ git rebase master myb2
Successfully rebased and updated refs/heads/myb2.
```

变基完成后，即形成如下的结果：


```
A---B---C  master
         \
          D'---E'---F'  myb2
```

变基后的 `D'`、`E'`、`F'` 提交对象是是 `D`、`E`、`F`提交对象复制品，并提交哈希值也不相同。以前的提交对象 `D`、`E`、`F` 依旧存在，可以通过 `git reflog` 查看。

变基后 `myb2` 分支的最终状态已经包含了 `C` 提交的 `c.txt` 这个文件（此时是 6 个文件）。

上述操作如果在 `master` 分支的 `C` 提交和 `myb2` 分支的 `D`、`E`、`F` 提交修改的公共的文件，变基操作可能会失败。出现如下提示。

```
$ git rebase master myb2
Auto-merging b.txt
CONFLICT (content): Merge conflict in b.txt
error: could not apply 8e7f7c7... 修改了 b.txt
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --
abort".
Could not apply 8e7f7c7... 修改了 b.txt
```

上述提示是在 变基时 因为两个分支同时修改了 `b.txt` 而产生了变基的冲突。因此要手动解决变基后 `b.txt` 的冲突。方法有如下几种：

1. 如果你想放弃这次变基，则使用 `git rebase --abort` 恢复到变基之前的状态。
2. 如果你手动解决了 `b.txt` 的冲突，你可以使用 `git add/rm` 将冲突文件提交到暂存区，然后运行 `git rebase --continue` 继续你的变基操作。
3. 也可以使用 `git rebase --skip` 命令跳过导致合并冲突的提交。

将 `master` 分支变基到 `myb2` 分支

> 重新解压缩上述示例代码

如果此时你在 `master` 分支中，你可以使用 `git rebase myb2` 命令将 `master` 分支变基到 `myb2` 分支。

如果你在其它分支中，你可以使用 `git rebase myb2 master` 命令同样可以将 `master` 分支变基到 `myb2` 分支。

变基完成后会切换到 `master` 分支。

> `git rebase myb2 master` 等同于 `git switch master` 然后在执行 `git rebase myb2`。

执行结果如下:

```
$ git rebase myb2
Successfully rebased and updated refs/heads/master.
```

变基前

```
A---B---C  master
     \
      D---E---F  myb2
```

变基后

```
A---B           C‘  master
     \         /
      D---E---F  myb2
```

此次变基后 `myb2` 分支没有变化，但 `master` 分支的的最终结果是含有 6 个文件`a.txt` … `f.txt`。

此时分支 `master` 的历史记录是 `A-B-D-E-F-C'`，`myb1` 分支的提交历史是 `A-B-D-E-F`。

示例2

变基到上游分支

在使用 `git rebase` 进行变基时，可以使用 `--onto <新基底>` 选项将变基后的结果应用到某个新基底的提交对象上。

假设现在有这样一个提交情况:

```
                     H---I---J myb4
                    /
            E---F---G  myb3
           /
A---B---C---D  master
```

下载此示例仓库

则使用 `git rebase --onto master myb3 myb4` 进行变基，变基后的结果如下：

> 如果当前分支是 `myb4` 则可以使用 `git rebase --onto master myb3` 命令。

```
             H'--I'--J'  myb4
            /
            | E---F---G  myb3
            |/
A---B---C---D  master
```

使用 `git log --graph --oneline --all` 可以查看结果如下:

```
weimingze@mzstudio:~/rebase_demo2$ git log --graph --oneline --all
* d474c64 (HEAD -> myb4) J
* c74a2de I
* 640ca4d H
* 5f9fd7d (myb3) G
* 9c3c5c9 F
* 6628d53 E
* 1c200fe (master) D
* 80d8724 C
* f29bb2c B
* 355d87c A
weimingze@mzstudio:~/rebase_demo2$ git rebase --onto master myb3
Successfully rebased and updated refs/heads/myb4.
weimingze@mzstudio:~/rebase_demo2$ git log --graph --oneline --all
* 419fbc3 (HEAD -> myb4) J
* 077ac87 I
* a036bff H
| * 5f9fd7d (myb3) G
| * 9c3c5c9 F
| * 6628d53 E
|/
* 1c200fe (master) D
* 80d8724 C
* f29bb2c B
* 355d87c A
```

如果 原来的 myb4 分支几乎没有在 myb3 分支上进行改动，则此中做法很高效且实用。

上述命令 `git rebase --onto master myb3 myb4` 中 `master` 是指 `master` 分支的最后提交对象 `D` 作为新基底，代替默认的 `myb4`；`myb3` 是指 `myb3` 最后的提交对象 `G`，`myb4` 是指 `myb4` 最后的提交对象 `J`，上述会将从 `G` 开始（不包含`G`）到 `J` 的结束（包含`J`）分支移动到 `D` 提交对象之后。

示例3

同一分枝变基

现在有这样一组分支。

```
A---B---C---D---E---F---G  master
```

下载此示例仓库

上述提交中的 C 中，由于操作失误，错将中间生成的占用空间很大的临时资源文件错误提交到了 C 中。然后在 D 提交时又删除了此资源文件。后续 E、F、G是正常提交。如果将此分支直接推送到远处的合作分支，则远处合作分支会多出一些垃圾文件，到时所有拉取此远处仓库的合作者的本地仓库也会变大。

这种情况可以将 E 及之后的所有提交变基到 B即可，而后的 C 和 D 则会失去引用。

在使用 `git rebase` 进行变基时，可以使用 `--onto <新基底>` 选项将分支的后续节点变基到之前的节点达到上述目的。

在当前分支 `master` 下执行 `git rebase --onto HEAD~5 HEAD~3` 或 `git rebase --onto HEAD~5 HEAD~3 master` 命令。

变基后的结果

```
A---B---E'---F'---G'  master
```

使用 `git log --graph --oneline --all` 可以查看结果如下:

```
weimingze@mzstudio:~/rebase_demo3$ git log --graph --oneline --all
* 8498821 (HEAD -> master) G
* 18b5f87 F
* 0aee6a4 E
* 6753c83 D
* 2052505 C
* 95a244a B
* 297e5a2 A
weimingze@mzstudio:~/rebase_demo3$ git rebase --onto HEAD~5 HEAD~3
Successfully rebased and updated refs/heads/master.
weimingze@mzstudio:~/rebase_demo3$ git log --graph --oneline --all
* 2959bca (HEAD -> master) G
* dd86199 F
* dbc69b1 E
* 95a244a B
* 297e5a2 A
```

变基的限制

不要对已推送到远程仓库且与它人共享的分支进行变基。

实验:

下载上述示例的仓库，尝试使用 `git rebase` 命令达成你自己的变基目标。

6. 交互式变基

交互式变基是指在使用 `git rebase` 命令进行变基时，可以使用 `-i` 选项通过文本编辑器对变基后的即将插入新位置的提交进行重新编辑并提交的操作。

这个操作包括

1. `pick`：使用提交。
2. `squash`：合并到前一个提交。
3. `fixup`：类似squash，但丢弃提交信息。
4. `reword`：使用提交，但重新编辑提交信息。
5. `edit`：使用提交，暂停修改。
6. `drop`：删除提交

示例

现在有这样一组分支。

```
A---B---C---D---E---F---G  master
```

下载此示例仓库

现在希望在 `D` 位置变基到 `B` 位置。变基后 `C` 提交对象删除。变基时 `D` 和 `E` 合并为 `DE'`，`F` 删除，`G` 的提价信息由原来的 `G` 体改为 `G for rebase。`

变基前提交信息

```
weimingze@mzstudio:~/rebase/rebase_demo4$ git log --graph --oneline --all
* 8498821 (HEAD -> master) G
* 18b5f87 F
* 0aee6a4 E
* 6753c83 D
* 2052505 C
* 95a244a B
* 297e5a2 A
```

使用 `git rebase -i --onto HEAD~5 HEAD~4` 命令进行交互式变基。编辑器提示如下：

```
pick 6753c83 D
pick 0aee6a4 E
pick 18b5f87 F
pick 8498821 G

# Rebase 2052505..8498821 onto 95a244a (4 commands)
```

```
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
...
```

编辑器中 `#` 开头的内容是注释内容，提示你如何操作。最上面的四行是 `D`、`E`、`F`、`G` 四个提交的的提交方式，默认是 `pick` 使用提交。

我们可以根据目标，修改上述内容如下：

```
pick 6753c83 D
squash 0aee6a4 E
drop 18b5f87 F
reword 8498821 G
```

其中 `pick 6753c83 D` 是提交 `D`。

`squash 0aee6a4 E` 是将 `E` 和上一次提交合并。此提交回弹出编辑器用来编辑两个提交合并后的提交信息。此时我们重新编辑提交信息为 `DE' for D and E Merge`。

`drop 18b5f87 F` 是删除 `F` 提交。

`reword 8498821 G` 会调用编辑器，重新编辑 `G` 的提交信息，再次我们添加提交信息： `G for rebase。`

保存退出上述编辑器。

执行结果如下：

```
weimingze@mzstudio:~/rebase/rebase_demo4$ git rebase -i --onto HEAD~5 HEAD~4
[detached HEAD 536f3d4] DE` for D and E Merge
 Date: Tue Jan 20 15:40:54 2026 +0800
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 d.txt
 create mode 100644 e.txt
[detached HEAD b2cfb81] G for rebase。
 Date: Tue Jan 20 15:40:54 2026 +0800
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 g.txt
Successfully rebased and updated refs/heads/master.
weimingze@mzstudio:~/rebase/rebase_demo4$ git log --graph --oneline --all
* b2cfb81 (HEAD -> master) G for rebase。
* 536f3d4 DE` for D and E Merge
* 95a244a B
* 297e5a2 A
```

通过上述操作，我们将原来的分支修改为如下结构：

```
A---B---DE'---G  master
```

实验

使用交互式变基的方式对示例中的提交进行操作。

总结

Git 总结

在此教程的讲解中。使用 `my_project` 项目案例几乎使用了所有常用的 Git 操作。在此对用过的命令做一个总结：

命令	简要说明
	工作区创建命令
`git clone`	克隆仓库到一个新文件夹
`git init`	创建一个空的 Git 仓库或重新初始化一个已存在的仓库
	工作区和暂存区修改和恢复
`git add`	添加文件内容至索引
`git mv`	移动或重命名一个文件、文件夹或符号链接
`git restore`	恢复工作区文件
`git rm`	从工作区和索引中删除文件
	检查历史和状态
`git diff`	显示提交之间、提交和工作区之间等的差异
`git difftool`	显示提交之间、提交和工作区之间等的差异，使用图形界面显示
`git log`	显示提交日志
`git show`	显示各种类型的对象
`git status`	显示工作区状态
`git reflog`	显示引用日志
	分支，提交，工作区管理相关
`git branch`	列出、创建或删除分支
`git commit`	记录变更到仓库
`git merge`	合并两个或更多开发历史
`rebase`	在另一个分支上重新应用提交
`git reset`	重置当前 HEAD 到指定状态
`git switch`	切换分支
`git tag`	创建、列出、删除或校验一个 GPG 签名的标签对象
	远程仓库管理
`git fetch`	从另外一个仓库下载对象和引用
`git pull`	获取并整合另外的仓库或一个本地分支

`git push`	更新远程引用和相关的对象