

Git教程

一、Git简介

Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Git 与常用的版本控制工具 CVS, Subversion 等不同，它采用了**分布式版本库**的方式，不必服务器端软件支持。

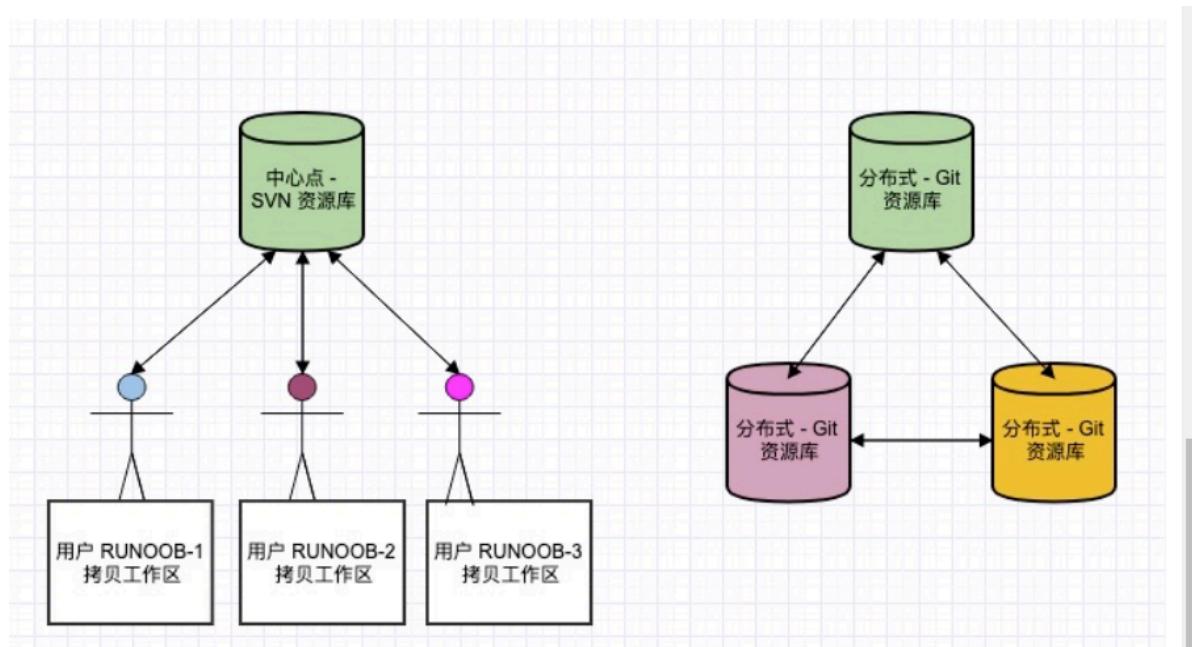
Git与SVN区别

Git 不仅仅是个版本控制系统，它也是个内容管理系统(CMS)，工作管理系统等。

如果你是一个具有使用 SVN 背景的人，你需要做一定的思想转换，来适应 Git 提供的一些概念和特征。

Git 与 SVN 区别点：

- 1、Git 是分布式的，SVN 不是：这是 Git 和其它非分布式的版本控制系统，例如 SVN, CVS 等，最核心的区别。
- 2、Git 把内容按元数据方式存储，而 SVN 是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似 .svn, .cvs 等的文件夹里。
- 3、Git 分支和 SVN 的分支不同：分支在 SVN 中一点都不特别，其实它就是版本库中的另外一个目录。
- 4、Git 没有一个全局的版本号，而 SVN 有：目前为止这是跟 SVN 相比 Git 缺少的最大的一个特征。
- 5、Git 的内容完整性要优于 SVN：Git 的内容存储使用的是 SHA-1 哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。



学习网址

[GIT命令手册](#)

二、Git 安装

在使用 Git 前我们需要先安装 Git。

Git 目前支持 Linux/Unix、Solaris、Mac 和 Windows 平台上运行。

[Git 下载地址](#)

Downloads

The screenshot shows the 'Downloads' section of the GitHub Git repository. It features three prominent download links: 'macOS' (with an Apple icon), 'Windows' (with a Windows icon), and 'Linux/Unix' (with a terminal icon). Below these links, a note states: 'Older releases are available and the Git source repository is on GitHub.' A small preview image of a computer monitor displaying the release notes for version 2.47.0 is visible.



Linux 平台安装

包安装

各大 Linux 平台可以使用包管理器（apt-get、yum 等）进行安装。

Debian/Ubuntu Git 安装最新稳定版本命令为：

```
sudo apt-get install git
```

Centos/RedHat

如果你使用的系统是 Centos/RedHat 安装命令为：

```
yum -y install git-core
```

Fedora 安装命令：

```
# yum install git (Fedora 21 及之前的版本)  
# dnf install git (Fedora 22 及更高版本)
```

FreeBSD 安装命令：

```
pkg install git
```

OpenBSD 安装命令：

```
pkg_add git
```

Alpine 安装命令：

```
apk add git
```

```
sun@sun-Lenovo-Legion-R7000P2021:~$ sudo apt install git  
[sudo] sun 的密码：  
正在读取软件包列表... 完成  
正在分析软件包的依赖关系树  
正在读取状态信息... 完成  
git 已经是最新版 (1:2.25.1-1ubuntu3.13)。  
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 404 个软件包未被升  
级。  
sun@sun-Lenovo-Legion-R7000P2021:~$
```

安装系统时已经自带

源码安装

[源码包地址](#)

Latest source Release
2.45.2
[Release Notes \(2024-05-31\)](#)

Download for Mac

macOS Windows

Linux/Unix

Older releases are available and the Git source repository is on GitHub.

也可以在 GitHub 上克隆源码包：

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

解压安装下载的源码包：

```
1 $ tar -zxf git-1.7.2.2.tar.gz
2 $ cd git-1.7.2.2
3 $ make prefix=/usr/local all
4 $ sudo make prefix=/usr/local install
```

Windows安装

在 Windows 平台上安装 Git 同样轻松，有个叫做 msysGit 的项目提供了安装包，可以到 GitHub 的页面上下载 exe 安装文件并运行：

[安装包下载地址](#)

[直接官网下载也可以](#)

下载完整直接双击安装



完成安装之后，就可以使用命令行的 git 工具（已经自带了 ssh 客户端）了，另外还有一个图形界面的 Git 项目管理工具。

在开始菜单里找到 "Git"->"Git Bash"，会弹出 Git 命令窗口，你可以在该窗口进行 Git 操作

此外，

如果你已经安装了 winget，可以使用以下命令来安装：

```
winget install --id Git.Git -e --source winget
```

Mac端安装

通过 Homebrew 安装：

```
brew install git
```

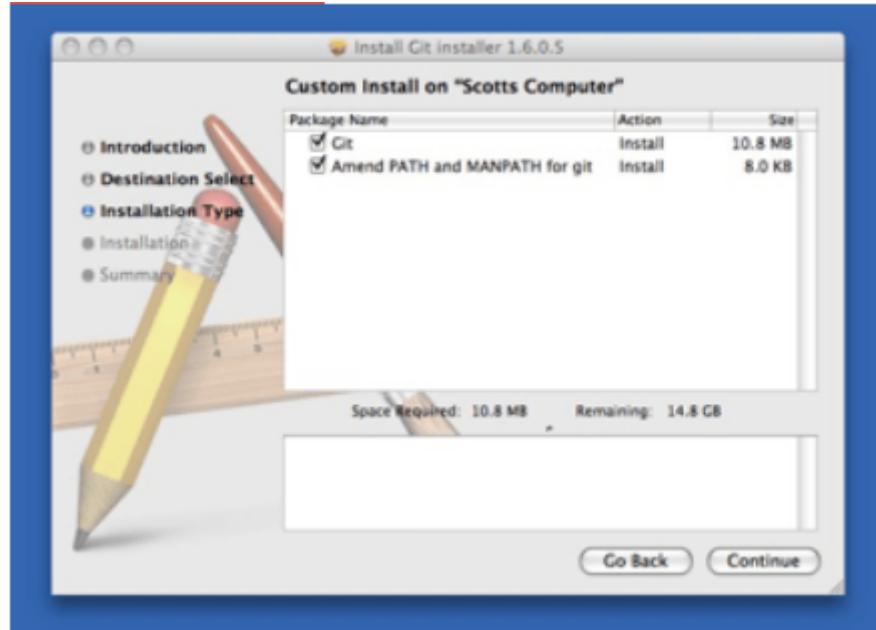
如果您想要安装 git-gui 和 gitk (git 的提交 GUI 和交互式历史记录浏览器)，您可以使用 homebrew 进行安装：

```
brew install git-gui
```

也可以使用图形化的 Git 安装工具

[下载地址为](#)

安装界面如下所示：



Git 配置

配置文件

Git 提供了一个叫做 `git config` 的命令，用来**配置或读取相应的工作环境变量**。

这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。

这些变量可以存放在以下三个不同的地方：

- 1 `/etc/gitconfig` 文件：系统中对所有用户都普遍适用的配置。
若使用 `git config` 时用 `--system` 选项，读写的就是这个文件。
- 2 `~/.gitconfig` 文件：用户目录下的配置文件只适用于该用户。
若使用 `git config` 时用 `--global` 选项，读写的就是这个文件。
- 3 当前项目的 `Git` 目录中的配置文件（也就是工作目录中的 `.git/config` 文件）：这里的配置仅仅针对当前项目有效。
- 4 每一个级别的配置都会覆盖上层的相同配置，所以 `.git/config` 里的配置会覆盖 `/etc/gitconfig` 中的同名变量。

在 Windows 系统上，Git 会找寻用户主目录下的 `.gitconfig` 文件。主目录即 `$HOME` 变量指定的目录，一般都是 `C:\Documents and Settings\$USER`。

此外，Git 还会尝试找寻 `/etc/gitconfig` 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

用户信息

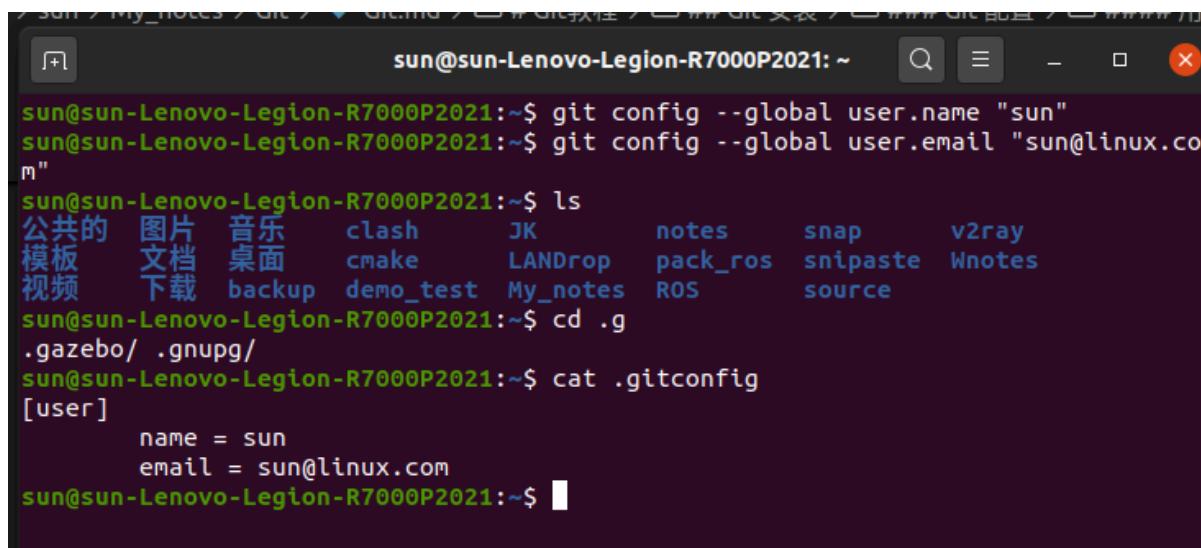
配置个人的用户名称和电子邮件地址，这是为了在每次提交代码时**记录提交者的信息**，信息会保存在你的配置文件中

```
1 git config --global user.name "runoob"  
2 git config --global user.email test@runoob.com
```

如果用了 `--global` 选项，那么更改的配置文件就是位于你用户主目录下的那个。以后你所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮。

只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。



```
sun@sun-Lenovo-Legion-R7000P2021:~$ git config --global user.name "sun"
sun@sun-Lenovo-Legion-R7000P2021:~$ git config --global user.email "sun@linux.co
m"
sun@sun-Lenovo-Legion-R7000P2021:~$ ls
公共的 图片 音乐 clash JK notes snap v2ray
模板 文档 桌面 cmake LANDrop pack_ros snipaste Wnotes
视频 下载 backup demo_test My_notes ROS source
sun@sun-Lenovo-Legion-R7000P2021:~$ cd .g
.gazebo/ .gnupg/
sun@sun-Lenovo-Legion-R7000P2021:~$ cat .gitconfig
[user]
    name = sun
    email = sun@linux.com
sun@sun-Lenovo-Legion-R7000P2021:~$
```

文本编辑器

设置 Git 默认使用的文本编辑器,一般可能会是 Vi 或者 Vim, 如果你有其他偏好, 比如 VS Code 的话, 可以重新设置

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

core.editer 表示需要Git编辑文件时使用的编辑器

code --wait指定了使用vscode来进行编辑 --wair表示Git会确保用户在完成VScode编辑后才会继续操作

差异分析工具

还有一个比较常用的是，在解决合并冲突时使用哪种差异分析工具。

比如要改用 vimdiff 的话：

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

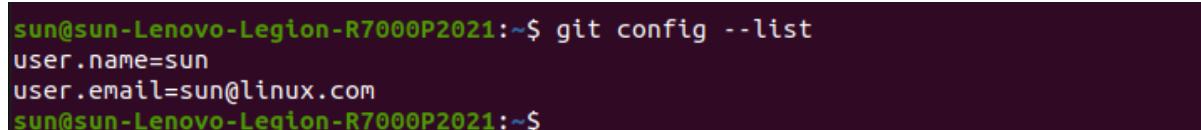
Git 可以理解 kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, 和 opendiff 等合并工具的输出信息。

当然，你也可以指定使用自己开发的工具，具体怎么做可以参阅第七章。

查看配置信息

要检查已有的配置信息，可以使用 git config --list 命令：

```
1 $ git config --list
2 http.postbuffer=2M
3 user.name=runoob
4 user.email=test@runoob.com
```



```
sun@sun-Lenovo-Legion-R7000P2021:~$ git config --list
user.name=sun
user.email=sun@linux.com
sun@sun-Lenovo-Legion-R7000P2021:~$
```

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig）不过最终 Git 实际采用的是最后一个。

这些配置我们也可以在 ~/.gitconfig 或 /etc/gitconfig 看到，如下所示：

```
vim ~/.gitconfig
```

显示如下所示：

```
1 [http]
2   postBuffer = 2M
3 [user]
4   name = runoob
5   email = test@runoob.com
```

```
sun@sun-Lenovo-Legion-R7000P2021:~$ cat .gitconfig
[user]
  name = sun
  email = sun@linux.com
sun@sun-Lenovo-Legion-R7000P2021:~$
```

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

```
1 $ git config user.name
2 runoob
```

```
sun@sun-Lenovo-Legion-R7000P2021:~$ git config user.name
sun
sun@sun-Lenovo-Legion-R7000P2021:~$
```

生成SSH密钥

如果你需要通过 SSH 进行 Git 操作。

可以生成 SSH 密钥并添加到你的 Git 托管服务（如 GitHub、GitLab 等）上。

```
ssh-keygen -t rsa -b 4096 -C "your.email@example.com"
```

按提示完成生成过程，然后将生成的公钥添加到相应的平台。

验证安装

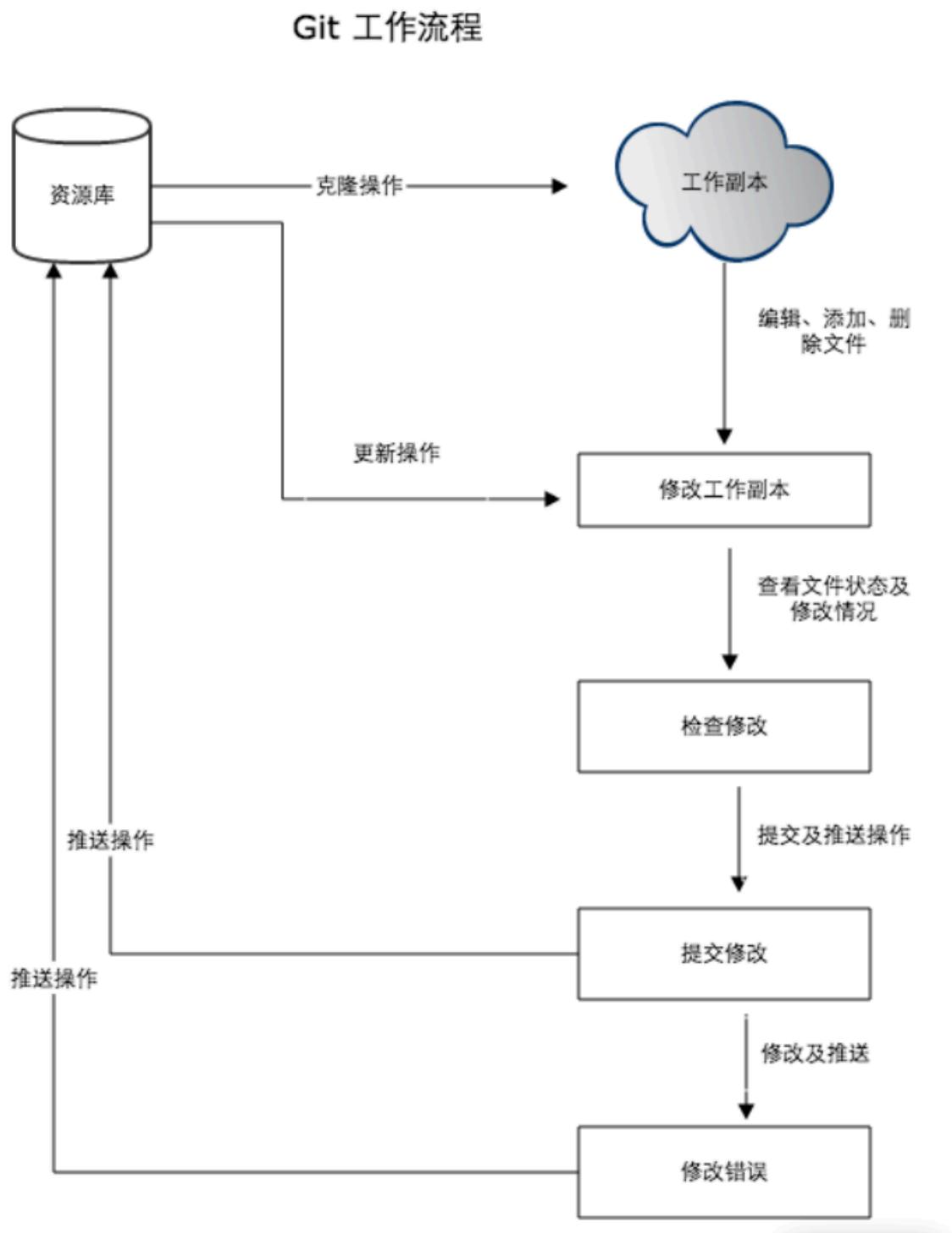
在终端或命令行中运行以下命令，确保 Git 已正确安装并配置：

```
1 git --version
2 git config --list
```

三、Git工作流程

本章节我们将为大家介绍 Git 的工作流程。

下图展示了 Git 的工作流程：



1. 克隆仓库

如果你要参与一个已有的项目，首先需要将远程仓库克隆到本地：

```
1 | git clone https://github.com/username/repo.git
2 | cd repo
```

2. 创建新分支

为了避免直接在 main 或 master 分支上进行开发，通常会创建一个新的分支：

```
1 | git checkout -b new-feature
```

3. 工作目录

在工作目录中进行代码编辑、添加新文件或删除不需要的文件。

4. 暂存文件

将修改过的文件添加到暂存区，以便进行下一步的提交操作：

```
1 | git add filename  
2 | # 或者添加所有修改的文件  
3 | git add .
```

5. 提交修改

将暂存区的更改提交到本地仓库，并添加提交信息：

```
1 | git commit -m "Add new feature"
```

6. 拉取最新修改

在推送本地更改之前，最好从远程仓库拉取最新的更改，以避免冲突：

```
1 | git pull origin main  
2 | # 或者如果在新的分支上工作  
3 | git pull origin new-feature
```

7. 推送更改

将本地的提交推送到远程仓库：

```
1 | git push origin new-feature
```

8. 创建Pull Request (PR)

在 GitHub 或其他托管平台上创建 Pull Request，邀请团队成员进行代码审查。

PR 合并后，你的更改就会合并到主分支。

9. 合并修改

在 PR 审核通过并合并后，可以将远程仓库的主分支合并到本地分支：

```
1 | git checkout main  
2 | git pull origin main  
3 | git merge new-feature
```

10. 删除分支

如果不再需要新功能分支，可以将其删除：

```
git branch -d new-feature
```

或者从远程仓库删除分支：

```
git push origin --delete new-feature
```

四、Git工作区、暂存区和版本库

基本概念

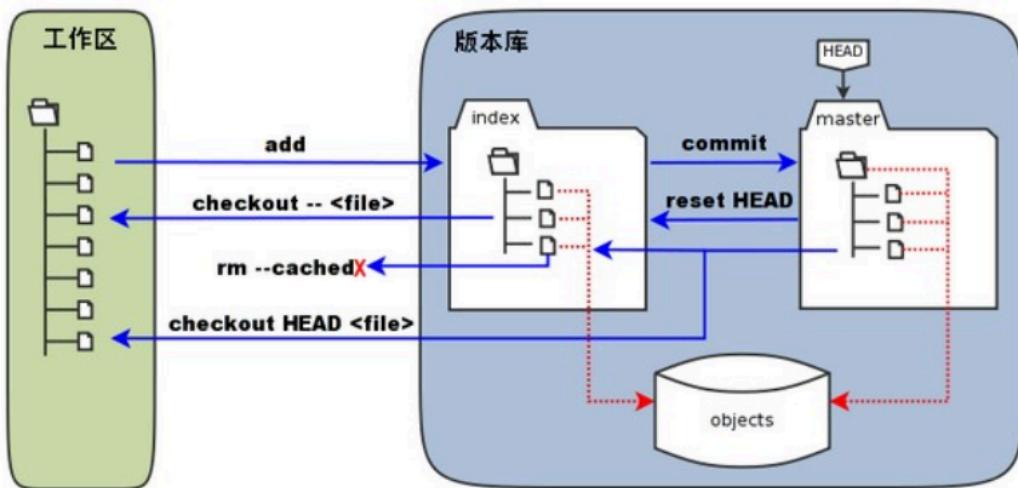
我们先来理解下 Git 工作区、暂存区和版本库概念：

工作区：就是你在电脑里能看到的目录。

暂存区：英文叫 stage 或 index。一般存放在 .git 目录下的 index 文件 (.git/index) 中，所以我们把暂存区有时也叫作索引 (index) 。

版本库：工作区有一个隐藏目录 .git，这个不算工作区，而是 Git 的版本库。

关系图解：



解释：

图中左侧为工作区，右侧为版本库。

在版本库中标记为 "index" 的区域是暂存区 (stage/index)，标记为 "master" 的是 master 分支所代表的目录树。

图中我们可以看出此时 "HEAD" 实际是指向 master 分支的一个"游标"。

所以图示的命令中出现 HEAD 的地方可以用 master 来替换。

图中的 objects 标识的区域为 Git 的对象库，实际位于 ".git/objects" 目录下，里面包含了创建的各种对象及内容

当对工作区修改（或新增）的文件执行 git add 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。

当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。

即 master 指向的目录树就是提交时暂存区的目录树。

当执行 git reset HEAD 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。

当执行 git rm --cached 命令时，会直接从暂存区删除文件，工作区则不做出改变。

当执行 git checkout . 或者 git checkout -- 命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区中的改动。

当执行 `git checkout HEAD .` 或者 `git checkout HEAD` 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。

这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

工作区

工作区是你在本地计算机上的项目目录，你在这里进行文件的创建、修改和删除操作。

工作区包含了当前项目的所有文件和子目录。

特点：

- 1 显示项目的当前状态。
- 2 文件的修改在工作区中进行，但这些修改还没有被记录到版本控制中。

暂存区

暂存区是一个临时存储区域，它包含了即将被提交到版本库中的文件快照。

在提交之前，你可以选择性地将工作区中的修改添加到暂存区。

特点：

- 1 暂存区保存了将被包括在下一个提交中的更改。
- 2 你可以多次使用 `git add` 命令来将文件添加到暂存区，直到你准备好提交所有更改。

常用命令：

- 1 `git add filename` # 将单个文件添加到暂存区
- 2 `git add .` # 将工作区中的所有修改添加到暂存区
- 3 `git status` # 查看哪些文件在暂存区中

版本库

版本库包含项目的所有版本历史记录。

每次提交都会在版本库中创建一个新的快照，这些快照是不可变的，确保了项目的完整历史记录。

特点：

- 1 版本库分为本地版本库和远程版本库。这里主要指本地版本库。
- 2 本地版本库存储在 `.git` 目录中，它包含了所有提交的对象和引用。

常用命令：

- 1 `git commit -m "Commit message"` # 将暂存区的更改提交到本地版本库
- 2 `git log` # 查看提交历史
- 3 `git diff` # 查看工作区和暂存区之间的差异
- 4 `git diff --cached` # 查看暂存区和最后一次提交之间的差异

工作区、暂存区与版本库之间关系

1. 工作区 -> 暂存区

使用 git add 命令将工作区中的修改添加到暂存区。

```
git add filename
```

2. 暂存区 -> 版本库

使用 git commit 命令将暂存区中的修改提交到版本库。

```
git commit -m "Commit message"
```

3. 版本库 -> 远程仓库

使用 git push 命令将本地版本库的提交推送到远程仓库。

```
git push origin branch-name
```

4. 远程仓库 -> 本地版本库

使用 git pull 或 git fetch 命令从远程仓库获取更新。

```
1 | git pull origin branch-name
2 | # 或者
3 | git fetch origin branch-name
4 | git merge origin/branch-name
```

示例

假设你在工作目录中修改了 file.txt:

1、工作区

修改 file.txt 并保存。

2、暂存区

将修改添加到暂存区：

```
git add file.txt
```

3、版本库

将暂存区的修改提交到本地版本库：

```
git commit -m "Update file.txt"
```

4、远程仓库

将本地提交推送到远程仓库：

```
git push origin main
```

通过理解工作区、暂存区和版本库的作用及其相互关系，你可以更加高效地使用 Git 进行版本控制和协同开发。

五、Git创建仓库

本章节我们将为大家介绍如何创建一个 Git 仓库。

你可以使用一个已经存在的目录作为 Git 仓库。

Git init

Git 使用 git init 命令来初始化一个 Git 仓库，Git 的很多命令都需要在 Git 的仓库中运行，所以 git init 是使用 Git 的第一个命令。

在执行完成 git init 命令后，Git 仓库会生成一个 .git 目录，该目录包含了资源的所有元数据，其他的项目目录保持不变。

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ cd Myproject/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git init
已初始化空的 Git 仓库于 /home/sun/My_notes/Git/codes/Myproject/.git/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls -a
.  ..  .git
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ cd .git/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject/.git$ ls
branches  config  description  HEAD  hooks  info  objects  refs
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject/.git$
```

使用方法

进入你想要创建仓库的目录，或者先创建一个新的目录：

```
1 | mkdir my-project
2 | cd my-project
```

使用当前目录作为 Git 仓库，我们只需使它初始化。

```
git init
```

该命令执行完后会在当前目录生成一个 .git 目录。

当然我们也使用我们指定目录作为 Git 仓库。

```
git init newrepo
```

初始化后，会在 newrepo 目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。

如图：

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ ls
lyproject
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ mkdir newrepo
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ git init newrepo/
已初始化空的 Git 仓库于 /home/sun/My_notes/Git/codes/newrepo/.git/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ cd newrepo/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ ls -a
.  ..  .git
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$
```

如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

```
1 $ git add *.c  
2 $ git add README  
3 $ git commit -m '初始化项目版本'
```

以上命令将目录下以 .c 结尾及 README 文件提交到仓库中。

注意：

- 1 在 Linux 系统中，commit 信息使用单引号 '
- 2 windows 系统，commit 信息使用双引号 "
- 3 所以在 git bash 中 git commit -m '提交说明' 这样是可以的
- 4 在 windows 命令行中就要使用双引号 git commit -m "提交说明"

Git clone

我们使用 git clone 从现有 Git 仓库中拷贝项目（类似 svn checkout）。

克隆仓库的命令格式为：

```
git clone <repo>
```

如果我们需要克隆到指定的目录，可以使用以下命令格式：

```
git clone <repo> <directory>
```

参数说明：

- 1 **repo:**Git 仓库。
- 2 **directory:**本地目录。

比如，要克隆 Ruby 语言的 Git 代码仓库 Grit，可以用下面的命令：

```
$ git clone git://github.com/schacon/grit.git
```

执行该命令后，会在当前目录下创建一个名为grit的目录，其中包含一个 .git 的目录，用于保存下载下来的所有版本记录。

如果要自己定义要新建的项目目录名称，可以在上面的命令末尾指定新的名字：

```
$ git clone git://github.com/schacon/grit.git mygrit
```

示例：

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ git clone https://github.com/Embedfire/ebf_6ull_linux.git IMX6ULL
正克隆到 'IMX6ULL'...
remote: Enumerating objects: 54418, done.
remote: Counting objects: 100% (172/172), done.
remote: Compressing objects: 100% (103/103), done.
remote: Total 54418 (delta 79), reused 105 (delta 44), pack-reused 54246 (from 1)
接收对象中: 100% (54418/54418), 149.01 MiB | 11.37 MiB/s, 完成.
处理 delta 中: 100% (5315/5315), 完成.
正在更新文件: 100% (51413/51413), 完成.
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ ls
IMX6ULL Myproject newrepo
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ cd IMX6ULL/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/IMX6ULL$ ls
arch      crypto      init      MAINTAINERS      README.md      tools
block     Documentation  ipc      Makefile      REPORTING-BUGS  usr
build.sh   drivers      Kbuild    mm      samples      virt
COPYING    firmware      Kconfig   Module.symvers  scripts
copy.sh    fs          kernel    net      security
CREDITS    include      lib      OFFICIAL-README sound
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/IMX6ULL$
```

配置

git 的设置使用 git config 命令。

显示当前的 git 配置信息：

```
1 $ git config --list
2 credential.helper=osxkeychain
3 core.repositoryformatversion=0
4 core.filemode=true
5 core.bare=false
6 core.logallrefupdates=true
7 core.ignorecase=true
8 core.precomposeunicode=true
```

```
5sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/IMX6ULL$ ls
5arch      crypto      init      MAINTAINERS      README.md      tools
5block     Documentation  ipc      Makefile      REPORTING-BUGS  usr
5build.sh   drivers      Kbuild    mm      samples      virt
5COPYING    firmware      Kconfig   Module.symvers  scripts
5copy.sh    fs          kernel    net      security
5CREDITS    include      lib      OFFICIAL-README sound
5sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/IMX6ULL$ git config --list
5user.name=sun
5user.email=sun@linux.com
5core.repositoryformatversion=0
5core.filemode=true
5core.bare=false
5core.logallrefupdates=true
5remote.origin.url=https://github.com/Embedfire/ebf_6ull_linux.git
5remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
5branch.master.remote=origin
5branch.master.merge=refs/heads/master
5sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/IMX6ULL$
```

编辑 git 配置文件：

```
$ git config -e      # 针对当前仓库
```

或者

```
$ git config -e --global  # 针对系统上所有仓库
```

```
GNU nano 4.8      /home/sun/My notes/Git/codes/IMX6ULL/.git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[remote "origin"]
url = https://github.com/Embedfire/ebf_6ull_linux.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master
```

已读取 11 行

^G 求助 ^O 写入 ^W 搜索 ^K 剪切文字 ^J 对齐 ^C 游标位置
^X 离开 ^R 读档 ^\ 替换 ^U 粘贴文字 ^T 拼写检查 ^ ^ 跳行

默认编辑器为nano 如果不喜欢

git config core.editor "vim" 可以修改为vim

git config core.editor "code --wait" 可以修改为VScode

默认都是在当前目录下修改，记录在.gitconfig文件中

加入--global 修改~/下的.gitconfig

加入--system 修改/etc/git/下的.gitconfig

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config core.editor "vim"
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config -e
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

The screenshot shows a terminal window with the following command history:

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config core.editor "vim"
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config -e
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

Below the command history, there is a code editor window titled "config" showing the contents of the ".git/config" file. The file contains the following configuration:

```
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
editor = code --wait
```

After the configuration command, the terminal shows the output of the command and a note about the editor configuration:

```
CREDITS include lib OFFICIAL-README sound
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/IMX6ULL$ cd ..
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ cd Myproject/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config -e
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config core.editor "vim" --wait
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config -e
提示：等待您的编辑器关闭文件... VIM - Vi IMproved 8.1 (2018 May 18, compiled Sep 25 2024 05:18:33)
未知的选项参数："--wait"
更多信息请见："vim -h"
error: There was a problem with the editor 'vim --wait'.
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config core.editor "vim"
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git config -e
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ cd ..
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes$ cd newrepo/
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ ls
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ git config -e
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ git config core.editor "code" --wait
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ git config -e
提示：等待您的编辑器关闭文件...
```

设置提交代码时的用户信息：

```
1 $ git config --global user.name "runoob"
2 $ git config --global user.email test@runoob.com
```

如果去掉 --global 参数只对当前仓库有效。

六、Git基本操作

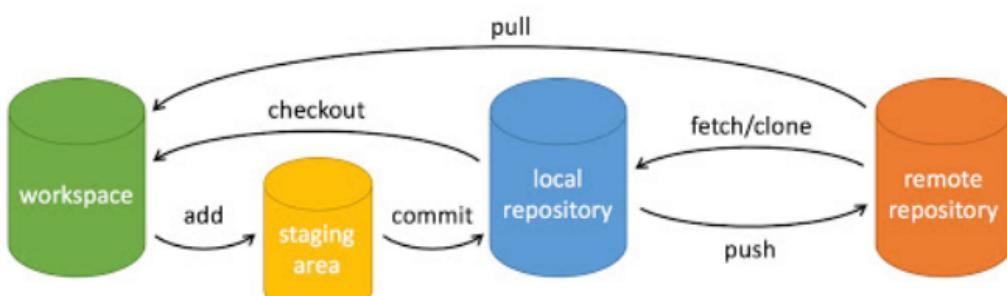
Git 的工作就是创建和保存你项目的快照及与之后的快照进行对比。

本章将对有关创建与提交你的项目快照的命令作介绍。

Git 常用的是以下 6 个命令：

git clone、git push、git add、git commit、git checkout、git pull

后面我们会详细介绍。



说明：

- 1 **workspace:** 工作区
- 2 **staging area:** 暂存区/缓存区
- 3 **local repository:** 版本库或本地仓库
- 4 **remote repository:** 远程仓库

一个简单的操作步骤：

```
1 $ git init  
2 $ git add .  
3 $ git commit
```

- git init - 初始化仓库。
- git add . - 添加文件到暂存区。
- git commit - 将暂存区内容添加到仓库中。

基本操作

创建仓库

下表列出了 git 创建仓库的命令：

命令	说明
git init	初始化仓库
git clone	拷贝一份远程仓库，也就是下载一个项目。

提交与修改

Git 的工作就是创建和保存你的项目的快照及与之后的快照进行对比。

下表列出了有关创建与提交你的项目的快照的命令：

命令	说明
git add	添加文件到暂存区
git status	查看仓库当前的状态，显示有变更的文件。
git diff	比较文件的不同，即暂存区和工作区的差异。
git difftool	使用外部差异工具查看和比较文件的更改。
git range-diff	比较两个提交范围之间的差异。
git commit	提交暂存区到本地仓库。
git reset	回退版本。
git rm	将文件从暂存区和工作区中删除。
git mv	移动或重命名工作区文件。
git notes	添加注释。
git checkout	分支切换。

命令	说明
git switch (Git 2.23 版本引入)	更清晰地切换分支。
git restore (Git 2.23 版本引入)	恢复或撤销文件的更改。
git show	显示 Git 对象的详细信息。

```

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add .
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master

尚无提交

要提交的变更:
  (使用 "git rm --cached <文件>..." 以取消暂存)
    新文件:  project.txt

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git diff
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git commit -m 'test'
[master (根提交) d76e891] test
 1 file changed, 2 insertions(+)
  create mode 100644 project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master

无文件要提交，干净的工作区
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ 
```

提交日志

命令	说明
git log	查看历史提交记录
git blame	以列表形式查看指定文件的历史修改记录
git shortlog	生成简洁的提交日志摘要
git describe	生成一个可读的字符串，该字符串基于 Git 的标签系统来描述当前的提交

```

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log
commit d76e891bd60eb3070b88bcc675d6aa937ccfbc0c (HEAD -> master)
Author: sun <sun@linux.com>
Date:   Fri Nov 15 13:54:24 2024 +0800

  test
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git blame
.git/      project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git blame project.txt
^d76e891 (sun 2024-11-15 13:54:24 +0800 1) This is a file for test the Git.
^d76e891 (sun 2024-11-15 13:54:24 +0800 2) It has no sense!
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git shortlog
sun (1):
  test

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git describe
fatal: 没有发现名称，无法描述任何东西。
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ 
```

远程操作

命令	说明
git remote	远程仓库操作

命令	说明
git fetch	从远程获取代码库
git pull	下载远程代码并合并
git push	上传远程代码并合并
git submodule	管理包含其他 Git 仓库的项目

Git文件状态

Git 的文件状态分为三种：

- 1 | 工作目录 (Working Directory)
- 2 | 暂存区 (Staging Area)
- 3 | 本地仓库 (Local Repository)

了解这些概念及其交互方式是掌握 Git 的关键。

工作目录

工作目录是你在本地计算机上看到的项目文件。

它是你实际操作文件的地方，包括查看、编辑、删除和创建文件。

所有对文件的更改首先发生在工作目录中。

在工作目录中的文件可能有以下几种状态：

- 1 | 未跟踪 (Untracked) : 新创建的文件，未被 Git 记录。
- 2 | 已修改 (Modified) : 已被 Git 跟踪的文件发生了更改，但这些更改还没有被提交到 Git 记录中。

暂存区

暂存区，也称为索引 (Index)，是一个临时存储区域，用于保存即将提交到本地仓库的更改。

你可以选择性地将工作目录中的更改添加到暂存区中，这样你可以一次提交多个文件的更改，而不必提交所有文件的更改。

使用 `git add <filename>` 命令将文件从工作目录添加到暂存区。

使用 `git add .` 命令将当前目录下的所有更改添加到暂存区。

- 1 | `git add < filename >` # 添加指定文件到暂存区
- 2 | `git add .` # 添加所有更改到暂存区

本地仓库

本地仓库是一个隐藏在 `.git` 目录中的**数据库**，用于存储项目的所有提交历史记录。

每次你提交更改时，Git 会将暂存区中的内容保存到本地仓库中。

使用 `git commit -m "commit message"` 命令将暂存区中的更改提交到本地仓库。

```
git commit -m "commit message" # 提交暂存区的更改到本地仓库
```

-m参数用于指定说明文档

文件状态的转换流程

未跟踪 (Untracked) : 新创建的文件最初是未跟踪的。它们存在于工作目录中，但没有被 Git 跟踪。

```
1 | touch newfile.txt # 创建一个新文件
2 | git status          # 查看状态，显示 newfile.txt 未跟踪
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ touch newfile.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
未跟踪的文件：
  (使用 "git add <文件>..." 以包含要提交的内容)
    newfile.txt

提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

已跟踪 (Tracked) : 通过 git add 命令将未跟踪的文件添加到暂存区后，文件变为已跟踪状态。

```
1 | git add newfile.txt # 添加文件到暂存区
2 | git status          # 查看状态，显示 newfile.txt 在暂存区
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add newfile.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
要提交的变更：
  (使用 "git restore --staged <文件>..." 以取消暂存)
    新文件： newfile.txt

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

已修改 (Modified) : 对已跟踪的文件进行更改后，这些更改会显示为已修改状态，但这些更改还未添加到暂存区。

```
1 | echo "Hello, World!" > newfile.txt # 修改文件
2 | git status                          # 查看状态，显示 newfile.txt 已修改
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ echo "Hello world" > newfile.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
stage  stash  status
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
要提交的变更：
  (使用 "git restore --staged <文件>..." 以取消暂存)
    新文件： newfile.txt

尚未暂存以备提交的变更：
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git restore <文件>..." 丢弃工作区的改动)
    修改： newfile.txt

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

已暂存 (Staged) : 使用 git add 命令将修改过的文件添加到暂存区后，文件进入已暂存状态，等待提交。

```
1 | git add newfile.txt # 添加文件到暂存区
2 | git status          # 查看状态，显示 newfile.txt 已暂存
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add newfile.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
要提交的变更:
  (使用 "git restore --staged <文件>..." 以取消暂存)
    新文件:    newfile.txt

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

已提交 (Committed)： 使用 git commit 命令将暂存区的更改提交到本地仓库后，这些更改被记录下来，文件状态返回为已跟踪状态。

```
1 | git commit -m "Added newfile.txt" # 提交更改
2 | git status                      # 查看状态，工作目录干净
```

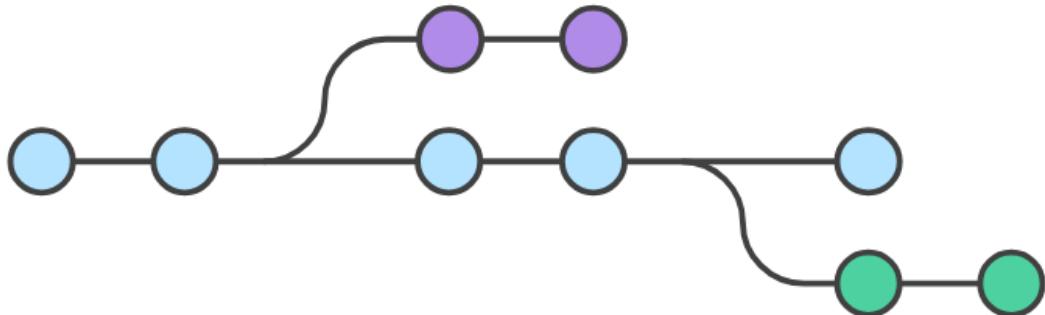
```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git commit -m "add newfile.txt"
[master 3ae9f3e] add newfile.txt
 1 file changed, 1 insertion(+)
 create mode 100644 newfile.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
无文件要提交，干净的工作区
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

七、Git分支管理

Git 分支管理是 Git 强大功能之一，能够让多个开发人员并行工作，开发新功能、修复 bug 或进行实验，而不会影响主代码库。

几乎每一种版本控制系统都以某种形式支持分支，一个分支代表一条独立的开发线。

使用分支意味着你可以从开发主线分离出来，然后在不影响主线的同时继续工作。



Git 分支实际上是指向更改快照的指针。

有人把 Git 的分支模型称为必杀技特性，而正是因为它，将 Git 从版本控制系统家族里区分出来。

创建分支

创建新分支并切换到该分支：

```
git checkout -b <branchname>
```

git checkout：这是用于切换到已存在的分支或文件的命令。

-b：这是一个选项，表示创建并切换到新的分支。

< branchname >：这是你想要创建的新分支的名称。

示例：

```
git checkout -b feature-xyz
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
newfile.txt project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git checkout -b feature-xyz
切换到一个新分支 'feature-xyz'
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
newfile.txt project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

切换分支命令

```
git checkout (branchname)
```

例如：

```
git checkout main
```

当你切换分支的时候，Git 会用该分支的最后提交的快照**替换你的工作目录**的内容。
所以多个分支不需要多个目录。

查看分支

查看所有分支：

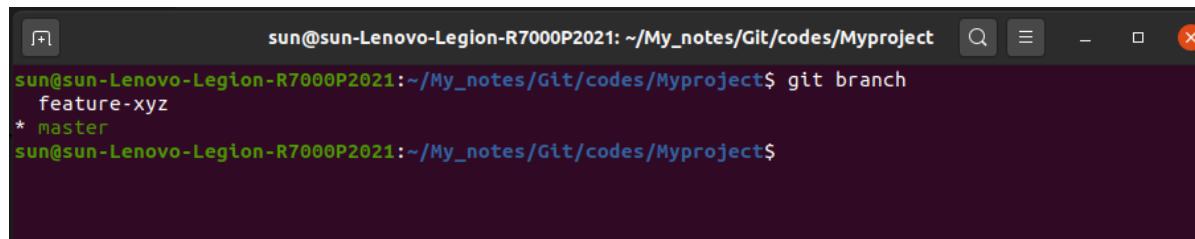
```
git branch
```

查看远程分支：

```
git branch -r
```

查看所有本地和远程分支：

```
git branch -a
```



```
sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/Myproject$ git branch
  feature-xyz
* master
sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/Myproject$
```

命令详解：

执行效果：

```
1 | $ git branch
2 | * master
```

此例的意思就是，我们有一个叫做 master 的分支，并且该分支是当前分支。

当你执行 git init 的时候，默认情况下 Git 就会为你创建 master 分支。

如果我们要手动创建一个分支。执行 git branch (branchname) 即可。

```
1 | $ git branch testing
2 | $ git branch
3 | * master
4 |   testing
```

现在我们可以看到，有了一个新分支 testing。

当你以此方式在上次提交更新之后创建了新分支(新分支相当于当前master的快照,可以继续开发)

如果后来master中又有更新提交。

然后又切换到了 testing 分支, Git 将还原你的工作目录到你创建分支时候的样子 (快照)

示例用 git checkout (branch) 切换到我们要修改的分支

```
1 $ ls
2 README
3 $ echo 'runoob.com' > test.txt
4 $ git add .
5 $ git commit -m 'add test.txt'
6 [master 3e92c19] add test.txt
7 1 file changed, 1 insertion(+)
8 create mode 100644 test.txt
9 $ ls
10 README      test.txt
11 $ git checkout testing
12 Switched to branch 'testing'
13 $ ls
14 README
```

当我们切换到 testing 分支的时候，我们添加的新文件 test.txt 被移除了。

切换回 master 分支的时候，它们又重新出现了。

```
1 $ git checkout master
2 Switched to branch 'master'
3 $ ls
4 README      test.txt
```

使用 git checkout -b (branchname) 命令来创建新分支并立即切换到该分支下

从而在该分支中操作

示例:

```
1 $ git checkout -b newtest
2 Switched to a new branch 'newtest'
3 $ git rm test.txt
4 rm 'test.txt'
5 $ ls
6 README
7 $ touch runoob.php
8 $ git add .
9 $ git commit -am 'removed test.txt、add runoob.php'
10 [newtest c1501a2] removed test.txt、add runoob.php
11 2 files changed, 1 deletion(-)
12 create mode 100644 runoob.php
13 delete mode 100644 test.txt
14 $ ls
15 README      runoob.php
16 $ git checkout master
17 Switched to branch 'master'
18 $ ls
19 README      test.txt
```

如你所见，我们创建了一个分支，在该分支上移除了一些文件 test.txt，并添加了 runoob.php 文件，然后切换回我们的主分支，删除的 test.txt 文件又回来了，且新增加的 runoob.php 不存在主分支中。使用分支将工作切分开来，从而让我们能够在不同开发环境中做事，并来回切换。

合并分支

将其他分支合并到当前分支：

```
git merge <branchname>
```

例如，切换到 main 分支并合并 feature-xyz 分支：

```
1 | git checkout main
2 | git merge feature-xyz
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git checkout master
切换到分支 'master'
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git merge feature-xyz
已经是最新的。
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git branch
  feature-xyz
* master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

合并分支不会消除原有分支

默认状态下git log 各个分支有自己的记录

当完成合并分支后，master中就拥有了其他分支的git commit记录

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log
commit 9e5c82d7bac57ded71d0a16adc9d7493796336dc (HEAD -> master)
merge: 2bid62f b19c4df
author: sun <sun@linux.com>
date:   Fri Nov 15 15:19:06 2024 +0800

  Merge branch 'Feature'
  through this merge,we finish touch two files!

commit b19c4df308e08e74fea4550690b5241eed225b03 (Feature)
author: sun <sun@linux.com>
date:   Fri Nov 15 15:18:33 2024 +0800

  new file 2

commit ded709b4ac57fb925704b7c708c3b5ea983f16c0
author: sun <sun@linux.com>
date:   Fri Nov 15 15:18:01 2024 +0800

  new file one

commit 2bid62f69a54ac0d47c845d7aeeb52ed3b933c4a
merge: 1bc61d0 1acf61
author: sun <sun@linux.com>
date:   Fri Nov 15 15:15:08 2024 +0800

  master 3

commit 1acf61bf7cae5d1f324716ad22f057ec41bb1c5 (feature)
author: sun <sun@linux.com>
date:   Fri Nov 15 15:13:26 2024 +0800

  feature 1

commit 1bc61d08f1475ca113946cf522511f21cc08ba10
author: sun <sun@linux.com>
date:   Fri Nov 15 15:12:11 2024 +0800
```

合并详解

一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。

你可以使用以下命令将任何分支合并到当前分支中去：

```
git merge
```

```
1 | $ git branch
2 | * master
```

```
3 newtest
4 $ ls
5 README      test.txt
6 $ git merge newtest
7 Updating 3e92c19..c1501a2
8 Fast-forward
9  runoob.php | 0
10 test.txt   | 1 -
11 2 files changed, 1 deletion(-)
12 create mode 100644 runoob.php
13 delete mode 100644 test.txt
14 $ ls
15 README      runoob.php
```

以上实例中我们将 newtest 分支合并到主分支去， test.txt 文件被删除。
合并完后就可以删除分支：

```
1 $ git branch -d newtest
2 Deleted branch newtest (was c1501a2).
```

删除后，就只剩下 master 分支了：

```
1 $ git branch
2 * master
```

合并冲突

当合并过程中出现冲突时，Git 会标记冲突文件，你需要手动解决冲突。
打开冲突文件，按照标记解决冲突。

标记冲突解决完成：

```
git add <conflict-file>
```

提交合并结果：

```
git commit
```

合并详解：

合并并不仅仅是简单的文件添加、移除的操作，Git 也会合并修改。

```
1 $ git branch
2 * master
3 $ cat runoob.php
```

我们创建一个叫做 change_site 的分支，切换过去，我们将 runoob.php 内容改为：

```
1 <?php
2 echo 'runoob';
3 ?>
```

创建 change_site 分支：

操作如下

```
1 #创建切换分支
2 $ git checkout -b change_site
3 Switched to a new branch 'change_site'
4 #修改内容
5 $ vim runoob.php
6 $ head -3 runoob.php
7 <?php
8 echo 'runoob';
9 ?>
10 #提交分支到本地仓库
11 $ git commit -am 'changed the runoob.php'
12 [change_site 7774248] changed the runoob.php
13 1 file changed, 3 insertions(+)
```

将修改的内容提交到 change_site 分支中。

现在，假如切换回 master 分支我们可以看内容恢复到我们修改前的(空文件，没有代码)
我们再次修改 runoob.php 文件。

```
1 #切回原来分支
2 $ git checkout master
3 Switched to branch 'master'
4 #修改内容
5 $ cat runoob.php
6 $ vim runoob.php      # 修改内容如下
7 $ cat runoob.php
8 <?php
9 echo 1;
10 ?>
11 #检测差异
12 $ git diff
13 diff --git a/runoob.php b/runoob.php
14 index e69de29..ac60739 100644
15 --- a/runoob.php
16 +++ b/runoob.php
17 @@ -0,0 +1,3 @@
18 +<?php
19 +echo 1;
20 +?>
21 #提交代码到本地库
22 $ git commit -am '修改代码'
23 [master c68142b] 修改代码
24 1 file changed, 3 insertions(+)
```

现在这些改变已经记录到我的 "master" 分支了。接下来我们将 "change_site" 分支合并过来。

```
1 $ git merge change_site
2 Auto-merging runoob.php
3 CONFLICT (content): Merge conflict in runoob.php
4 Automatic merge failed; fix conflicts and then commit the result.
5
6 $ cat runoob.php      # 打开文件，看到冲突内容
7 <?php
8 <<<<< HEAD
9 echo 1;
10 =====
11 echo 'runoob';
12 >>>>> change_site
13 ?>
```

我们将前一个分支合并到 master 分支，一个合并冲突就出现了，接下来我们需要手动去修改它。

```
1 $ vim runoob.php
2 $ cat runoob.php
3 <?php
4 echo 1;
5 echo 'runoob';
6 ?>
7 $ git diff
8 diff --cc runoob.php
9 index ac60739,b63d7d7..0000000
10 --- a/runoob.php
11 +++ b/runoob.php
12 @@@ -1,3 -1,3 +1,4 @@
13     <?php
14     +echo 1;
15     + echo 'runoob';
16     ?>
```

在 Git 中，我们可以用 git add 要告诉 Git 文件冲突已经解决

```
1 $ git status -s
2 UU runoob.php
3 $ git add runoob.php
4 $ git status -s
5 M runoob.php
6 $ git commit
7 [master 88afe0e] Merge branch 'change_site'
```

现在我们成功解决了合并中的冲突，并提交了结果。

删除分支

删除本地分支

```
git branch -d <branchname>
```

强制删除没有合并的分支

```
git branch -D <branchname>
```

删除远程分支

```
git push origin --delete <branchname>
```

示例

```
1 $ mkdir gitdemo
2 $ cd gitdemo/
3 $ git init
4 Initialized empty Git repository...
5 $ touch README
6 $ git add README
7 $ git commit -m '第一次版本提交'
8 [master (root-commit) 3b58100] 第一次版本提交
9   1 file changed, 0 insertions(+), 0 deletions(-)
10  create mode 100644 README
```

命令手册

- `git branch`

列出、创建或删除分支。

它不切换分支，只是用于管理分支的存在。

示例	解释
<code>git branch</code>	列出所有分支
<code>git branch new-branch</code>	创建新分支
<code>git branch -d old-branch</code>	删除分支

- `git checkout`

切换到指定的分支或恢复工作目录中的文件。

也可以用来检出特定的提交。

示例	解释
<code>git checkout branch-name</code>	切换分支
<code>git checkout file.txt</code>	恢复文件到工作区
<code>git checkout <commit-hash></code>	检出特定提交

- `git switch`

专门用于切换分支，相比 `git checkout` 更加简洁和直观，主要用于分支操作。

示例	解释
<code>git switch branch-name</code>	切换到指定分支

示例	解释
git switch -c new-branch	创建并切换到新分支

- `git merge`

合并指定分支的更改到当前分支。

示例	解释
<code>git merge branch-name</code>	将指定分支的更改合并到当前分支

- `git mergetool`

启动合并工具，以解决合并冲突。

示例	解释
<code>git mergetool</code>	使用默认合并工具解决冲突
<code>git mergetool --tool=< tool-name></code>	指定合并工具

- `git log`

显示提交历史记录。

示例	解释
<code>git log</code>	显示提交历史
<code>git log --oneline</code>	以简洁模式显示提交历史

- `git stash`

保存当前工作目录中的未提交更改，并将其恢复到干净的工作区。

示例	解释
<code>git stash</code>	保存当前更改
<code>git stash pop</code>	恢复最近保存的更改
<code>git stash list</code>	列出所有保存的更改

- `git tag`

创建、列出或删除标签。标签用于标记特定的提交。

示例	解释
git tag	列出所有标签
git tag v1.0	创建一个新标签
git tag -d v1.0	删除标签

- git worktree

允许在一个仓库中检查多个工作区，适用于同时处理多个分支。

示例	解释
git worktree add < path> branch-name	在指定路径添加新的工作区并切换到指定分支
git worktree remove < path>	删除工作区

八、Git查看提交历史

查看 Git 提交历史可以帮助你了解代码的变更情况和开发进度。

Git 提供了多种命令和选项来查看提交历史，从简单的日志到详细的差异对比。

Git 提交历史一般常用两个命令：

- 1 | `git log` - 查看历史提交记录。
- 2 | `git blame <file>` - 以列表形式查看指定文件的历史修改记录。

Git Log

在使用 Git 提交了若干更新之后，又或者克隆了某个项目，想回顾下提交历史，我们可以使用 `git log` 命令查看。

`git log` 命令用于查看 Git 仓库中提交历史记录。

`git log` 显示了从最新提交到最早提交的所有提交信息，包括提交的哈希值、作者、提交日期和提交消息等。

`git log` 命令的基本语法：

```
git log [选项] [分支名/提交哈希]
```

常用的选项包括：

选项	解释
-p	显示提交的补丁（具体更改内容）。
--oneline	以简洁的一行格式显示提交信息。
--graph	以图形化方式显示分支和合并历史。
--decorate	显示分支和标签指向的提交。
--author=<作者>	只显示特定作者的提交。
--since=<时间>	只显示指定时间之后的提交。

选项	解释
--until=<时间>	只显示指定时间之前的提交。
--grep=<模式>	只显示包含指定模式的提交消息。
--no-merges	不显示合并提交。
--stat	显示简略统计信息，包括修改的文件和行数。
--abbrev-commit	使用短提交哈希值。
--pretty=<格式>	使用自定义的提交信息显示格式。

git log --oneline

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --oneline
9e5c82d (HEAD -> master) Merge branch 'Feature' through this merge,we finish touch two files!
b19c4df (Feature) new file 2
ded709b new file one
2b1d62f mastor 3
1acfca61 feature 1
1bc61d0 master 2
9570605 master 1
a55bcd5 mastor deal merge problem
015b34f Feature modify
05aa8cd modify again
a87b3b2 modify newfile.txt
4bd6944 modify project.txt
3ae9f3e add newfile.txt
d76e891 test
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

git log --graph

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --graph
*   commit 9e5c82d7bac57ded71d0a16adc9d7493796336dc (HEAD -> master)
| \ Merge: 2b1d62f b19c4df
| | Author: sun <sun@linux.com>
| | Date:   Fri Nov 15 15:19:06 2024 +0800
|
|   Merge branch 'Feature'
|   through this merge,we finish touch two files!
|
* commit b19c4df308e08e74fea4550690b5241eed225b03 (Feature)
| Author: sun <sun@linux.com>
| Date:   Fri Nov 15 15:18:33 2024 +0800
|
|   new file 2
|
* commit ded709b4ac57fb925704b7c708c3b5ea983f16c0
| Author: sun <sun@linux.com>
```

此外，你也可以用 --reverse 参数来逆向显示所有日志。

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --reverse
commit d76e891bd60eb3070b88bcc675d6aa937ccfbc0c
Author: sun <sun@linux.com>
Date:   Fri Nov 15 13:54:24 2024 +0800
    test
commit 3ae9f3e46823c6cbc64ebaa35397140017fbfc10
Author: sun <sun@linux.com>
Date:   Fri Nov 15 14:22:09 2024 +0800
    add newfile.txt
commit 4bd694433372a3bbd64a89c97cf050f1cb670857
Author: sun <sun@linux.com>
Date:   Fri Nov 15 14:45:44 2024 +0800
```

如果只想查找指定用户的提交日志可以使用命令：git log --author
例如，比方说我们要找 Git 源码中 Linus 提交的部分：

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --author=sun
commit 9e5c82d7bac57ded71d0a16adc9d7493796336dc (HEAD -> master)
Merge: 2b1d62f b19c4df
Author: sun <sun@linux.com>
Date:   Fri Nov 15 15:19:06 2024 +0800

    Merge branch 'Feature'
    through this merge,we finish touch two files!

commit b19c4df308e08e74fea4550690b5241eed225b03 (Feature)
Author: sun <sun@linux.com>
Date:   Fri Nov 15 15:18:33 2024 +0800

    new file 2

commit ded709b4ac57fb925704b7c708c3b5ea983f16c0
```

如果你要指定日期，可以执行几个选项：--since 和 --before
但是你也可以用 --until 和 --after。

```
$ git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merges
5469e2d Git 1.7.1-rc2
d43427d Documentation/remote-helpers: Fix typos and improve language
272a36b Fixup: Second argument may be any arbitrary string
b6c8d2d Documentation/remote-helpers: Add invocation section
5ce4f4e Documentation/urls: Rewrite to accomodate transport::address
00b84e9 Documentation/remote-helpers: Rewrite description
03aa87e Documentation: Describe other situations where -z affects git diff
77bc694 rebase-interactive: silence warning when no commits rewritten
636db2c t3301: add tests to use --format="%N"
```

常用选项：

限制显示的提交数：

```
git log -n <number>
```

例如，显示最近的 5 次提交：

```
git log -n 5
```

显示自指定日期之后的提交：

```
git log --since="2024-01-01"
```

显示指定日期之前的提交：

```
git log --until="2024-07-01"
```

只显示某个作者的提交：

```
git log --author="Author Name"
```

更多配置项 请看[官方文档](#),或使用git log --help 寻求帮助。

git blame

git blame 命令用于逐行显示指定文件的每一行代码是由谁在什么时候引入或修改的。

git blame 可以追踪文件中每一行的变更历史，包括作者、提交哈希、提交日期和提交消息等信息。

如果要查看指定文件的修改记录可以使用 git blame 命令，格式如下：

```
git blame [选项] <文件路径>
```

常用选项：

选项	解释
-L <起始行号>,<结束行号>	只显示指定行号范围内的代码注释。
-C	对于重命名或拷贝的代码行，也进行代码行溯源。
-M	对于移动的代码行，也进行代码行溯源。
-C -C 或 -M -M	对于较多改动的代码行，进行更进一步的溯源。
--show-stats	显示包含每个作者的行数统计信息。

显示文件每一行的代码注释和相关信息：

```
git blame <文件路径>
```

只显示指定行号范围内的代码注释：

```
git blame -L <起始行号>,<结束行号> <文件路径>
```

对于重命名或拷贝的代码行进行溯源：

```
git blame -C <文件路径>
```

对于移动的代码行进行溯源：

```
git blame -M <文件路径>
```

显示行数统计信息：

```
git blame --show-stats <文件路径>
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git blame newfile.txt
3ae9f3e4 (sun 2024-11-15 14:22:09 +0800 1) Hello world
a55bcd5a (sun 2024-11-15 15:04:08 +0800 2) <<<<< HEAD
a87b3b2b (sun 2024-11-15 14:51:45 +0800 3) this was modified by Feature!
a87b3b2b (sun 2024-11-15 14:51:45 +0800 4) hello whis
2b1d62f6 (sun 2024-11-15 15:15:08 +0800 5) <<<<< HEAD
05aa8cdb (sun 2024-11-15 14:59:53 +0800 6) modify again
a55bcd5a (sun 2024-11-15 15:04:08 +0800 7) =====
015b34fc (sun 2024-11-15 15:02:43 +0800 8) I(Feature)modify again
a55bcd5a (sun 2024-11-15 15:04:08 +0800 9) >>>>> Feature
2b1d62f6 (sun 2024-11-15 15:15:08 +0800 10) =====
1acf61b (sun 2024-11-15 15:13:26 +0800 11) I ,feature, have modified this file ,haha!
2b1d62f6 (sun 2024-11-15 15:15:08 +0800 12) >>>>> feature
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

恢复和回退

Git 提供了多种方式来恢复和回退到之前的版本，不同的命令适用于不同的场景和需求。

以下是几种常见的方法：

- 1 `git checkout`: 切换分支或恢复文件到指定提交。
- 2 `git reset`: 重置当前分支到指定提交（软重置、混合重置、硬重置）。
- 3 `git revert`: 创建一个新的提交以撤销指定提交，不改变提交历史。
- 4 `git reflog`: 查看历史操作记录，找回丢失的提交。

git checkout: 检查出特定版本的文件

`git checkout` 命令用于切换分支或恢复工作目录中的文件到指定的提交。

恢复工作目录中的文件到某个提交：

```
git checkout <commit> -- <filename>
```

例如，将 `file.txt` 恢复到 `abc123` 提交时的版本：

```
git checkout abc123 -- file.txt
```

切换到特定提交：

```
git checkout <commit>
```

例如：

```
git checkout abc123
```

这种方式切换到特定的提交时，处于分离头指针（detached HEAD）状态。

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git checkout 2b1d62f69a54ac0d47c845d7aeeb52ed3b933c4a
注意：正在切换到 '2b1d62f69a54ac0d47c845d7aeeb52ed3b933c4a'。
您正处于分离头指针状态。您可以查看、做试验性的修改及提交，并且您可以在切换回一个分支时，丢弃在此状态下所做的提交而不对分支造成影响。
如果您想要通过创建分支来保留在此状态下所做的提交，您可以通过在 switch 命令中添加参数 -c 来实现（现在或稍后）。例如：
git switch -c <新分支名>
或者撤销此操作：
git switch -
通过将配置变量 advice.detachedHead 设置为 false 来关闭此建议
HEAD 目前位于 2b1d62f master 3
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
```

如果回退的是文件，会直接修改源文件到指定版本位置。

但是此时文件没有上传到本地库，仍可以返回源文件版本。

git reset: 重置当前分支到特定提交

`git reset` 命令可以更改当前分支的提交历史

它有三种主要模式：`--soft`、`--mixed` 和 `--hard`。

`--soft`: 只重置 HEAD 到指定的提交，暂存区和工作目录保持不变。

```
git reset --soft <commit>
```

`--mixed` (默认) : 重置 HEAD 到指定的提交，暂存区重置，但工作目录保持不变。（工作目录不变，暂存区跳到当前版本）

```
git reset --mixed <commit>
```

`--hard`: 重置 HEAD 到指定的提交，暂存区和工作目录都重置。（直接跳到指定提交版本）

```
git reset --hard <commit>
```

例如，将当前分支重置到 `abc123` 提交：

```
git reset --hard abc123
```

git revert: 撤销某次提交

git revert 命令创建一个新的提交，用来撤销指定的提交，它不会改变提交历史，适用于已经推送到远程仓库的提交。

```
git revert <commit>
```

例如，撤销 abc123 提交：

```
git revert abc123
```

git reflog: 查看历史操作记录

git reflog 命令记录了所有 HEAD 的移动。即使提交被删除或重置，也可以通过 reflog 找回。

```
git reflog
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git reflog
b19c4df (HEAD -> master, Feature) HEAD@{0}: reset: moving to b19c4df
28f0a9d HEAD@{1}: commit: huitui
b19c4df (HEAD -> master, Feature) HEAD@{2}: reset: moving to b19c4df308e08e74fea4550690b5241eed225b03
f272e6b HEAD@{3}: reset: moving to f272e6bdbf7354e9793d73c174c56623ee0e44ec
adc7da8 HEAD@{4}: commit: huitui version
f272e6b HEAD@{5}: commit: add project
9e5c82d HEAD@{6}: checkout: moving from NewVersion to master
2b1d62f (NewVersion) HEAD@{7}: checkout: moving from 2b1d62f69a54ac0d47c845d7aeeb52ed3b933c4a to NewVersion
2b1d62f (NewVersion) HEAD@{8}: checkout: moving from master to 2b1d62f69a54ac0d47c845d7aeeb52ed3b933c4a
9e5c82d HEAD@{9}: checkout: moving from feature to master
5f6f7ec (feature) HEAD@{10}: commit: xiugai project
1acf6a1 HEAD@{11}: checkout: moving from master to feature
9e5c82d HEAD@{12}: merge Feature: Merge made by the 'recursive' strategy.
2b1d62f (NewVersion) HEAD@{13}: checkout: moving from Feature to master
b19c4df (HEAD -> master, Feature) HEAD@{14}: commit: new file 2
ded709b HEAD@{15}: commit: new file one
015b34f HEAD@{16}: checkout: moving from master to Feature
2b1d62f (NewVersion) HEAD@{17}: commit (merge): mastor 3
1bc61d0 HEAD@{18}: checkout: moving from feature to master
1acf6a1 HEAD@{19}: commit: feature 1
a87b3b2 HEAD@{20}: checkout: moving from master to feature
1bc61d0 HEAD@{21}: commit: master 2
9570605 HEAD@{22}: commit: master 1
a55bcd5 HEAD@{23}: checkout: moving from feature to master
a87b3b2 HEAD@{24}: checkout: moving from Feature to feature
015b34f HEAD@{25}: checkout: moving from master to Feature
```

利用 reflog 可以找到之前的提交哈希，从而恢复到特定状态。例如：

```
git reset --hard HEAD@{3}
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git reset --hard b19c4df
HEAD 现在位于 b19c4df new file 2
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
newFeature newFeature2 newfile.txt project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ vim project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

实例：

以下是一个综合示例，演示如何使用这些命令恢复历史版本：

查看提交历史：

```
git log --oneline
```

假设输出如下：

```
1 abc1234 Commit 1
2 def5678 Commit 2
3 ghi9012 Commit 3
```

切换到 Commit 2 (处于分离头指针状态) :

```
git checkout def5678
```

重置到 Commit 2, 保留更改到暂存区:

```
git reset --soft def5678
```

重置到 Commit 2, 取消暂存区更改:

```
git reset --mixed def5678
```

重置到 Commit 2, 丢弃所有更改:

```
git reset --hard def5678
```

撤销 Commit 2:

```
git revert def5678
```

查看 reflog 找回丢失的提交:

```
git reflog
```

找到之前的提交哈希并恢复:

```
git reset --hard HEAD@{3}
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git reset --hard HEAD@{7}
HEAD 现在位于 f272e6b add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
newFeature newFeature2 newfile newfile.txt project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ vim project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ █
```

HEAD指针与git版本变化

HEAD指针

在 Git 中, HEAD 是一个非常重要的指针, 它指向当前检出的分支或者当前检出的某个提交。

可以将它视作一个指示器, 告诉 Git 当前工作目录中正在操作的分支或提交。

1. HEAD 指针的作用

指向当前分支:

当你在某个分支上工作时, HEAD 指向该分支的最新提交。

例如, 在 master 分支上, HEAD 指向 master 分支的最新提交。

指向某个特定提交:

在某些情况下, HEAD 可能直接指向某个具体的提交哈希 (如在“detached HEAD”状态下)。

此时, HEAD 不再指向分支, 而是指向某个提交对象, 这种状态下你可以查看或修改该提交, 但不会对任何分支产生影响。

2. HEAD 的常见形式

指向分支:

例如, 假设你当前在 main 分支上, 那么 HEAD 会指向 main 分支的最新提交。

你可以通过以下命令查看:

```
git log --oneline -n 1
```

这会显示 HEAD 当前指向的提交。

Detached HEAD (游离 HEAD) :

当你检出一个特定的提交（而不是分支）。

HEAD 将处于游离状态，直接指向那个提交，而不是某个分支。

例如：

```
git checkout < commit-hash >
```

在这种情况下，HEAD 不再指向任何分支，而是指向你检出的那个提交。

这通常是一个临时状态，你可以在这里进行查看、修改或创建新的分支。

如果不创建新的分支，修改将不会保存到任何分支。

版本回退

Git 的版本回退功能是基于其 快照存储 和 指针管理 机制实现的。

以下是具体原理：

1. Git 的数据存储模型

Git 不是直接存储文件的差异，而是将项目在某一时刻的完整快照存储下来。

每次提交（commit）都会生成一个唯一的哈希值（SHA-1），用于标识该快照。

一个提交包含以下信息：

- 1 提交哈希值
- 2 指向该提交的父提交（parent commit）的指针
- 3 提交时的树对象（保存文件和目录结构）
- 4 提交信息（message）
- 5 作者和时间戳

补充：父提交

父提交的指针是指向当前提交之前的一个或多个提交的引用，用于记录当前提交的历史来源。

它是 Git 提交对象中的一个重要组成部分。

主要作用有：

记录历史：每个提交会记录它的上一个提交（即父提交），这样所有提交会以链表的形式连接起来，形成一个完整的历史记录。

- 1 构建提交图： Git 的提交历史以有向无环图（DAG）的形式存在，每个提交通过父提交指针建立关联。
- 2
- 3 支持回溯： 基于父提交指针， Git 可以快速回溯到任意一个历史版本。

单个父提交：

在大多数情况下，普通的提交只有一个父提交。比如：

```
A --> B --> C
```

A 是 B 的父提交。

B 是 C 的父提交。

每个提交通过其父指针知道自己是从哪一个提交演变而来的。

多个父提交（合并时）：

当分支进行合并操作时，产生的合并提交会有两个或多个父提交。

```
1 |   A
2 |   / \
3 |   B   C
4 |   \ /
5 |   D
```

提交 D 是一个合并提交，拥有两个父提交：B 和 C。

Git 会通过这两个父提交来记录分支的合并点。

可以通过以下命令查看某个提交的父提交：

```
git log --graph --oneline
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --graph
--oneline
* f272e6b (HEAD -> master) add project
* 9e5c82d Merge branch 'Feature' through this merge,we finish touch two files!
|\ \
| * b19c4df (Feature) new file 2
| * ded709b new file one
* | 2b1d62f (NewVersion) mastor 3
| \ \
| * | 1acfa61 feature 1
* | | 1bc61d0 master 2
* | | 9570605 master 1
* | | a55bcd5 mastor deal merge problem
| \ \
| | |
| | |
| | *
| | | 015b34f Feature modify
* | | 05aa8cd modify again
| | |
| | |
| | *
| | | a87b3b2 modify newfile.txt
| |
* 4bd6944 modify project.txt
```

或者：

```
git show <commit-hash>
```

查看父提交指针：

- 直接查看父提交的哈希值

```
git log --pretty=format:"%h %p"
```

%h 显示提交的哈希值。

%p 显示父提交的哈希值。

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --pretty=format:"%h %p"
f272e6b 9e5c82d
9e5c82d 2b1d62f b19c4df
b19c4df ded709b
ded709b 015b34f
2b1d62f 1bc61d0 1acfa61
1acfa61 a87b3b2
1bc61d0 9570605
9570605 a55bcd5
a55bcd5 05aa8cd 015b34f
015b34f 4bd6944
05aa8cd a87b3b2
a87b3b2 4bd6944
4bd6944 3ae9f3e
3ae9f3e d76e891
d76e891
```

- 访问父提交

HEAD^ 表示当前提交的第一个父提交。

HEAD^2 表示当前提交的第二个父提交（在合并提交中有意义）。

例如：

```
git show HEAD^
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git show HEAD^
commit 9e5c82d7bac57ded71d0a16adc9d7493796336dc
Merge: 2b1d62f b19c4df
Author: sun <sun@linux.com>
Date:   Fri Nov 15 15:19:06 2024 +0800

    Merge branch 'Feature'
    through this merge, we finish touch two files!

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

父提交在版本回退中的作用:

Git 的版本回退依赖父提交指针来找到之前的版本

- 1 | `git reset` 会根据父提交指针移动 `HEAD` 到指定的父提交。
- 2 | `git revert` 会根据父提交生成一个反向变更并创建新的提交。

2. HEAD 指针

Git 通过一个叫 `HEAD` 的指针来跟踪当前所在的提交。

`HEAD` 通常指向当前分支的最新提交。

3. 版本回退的本质

版本回退实际上是操作 `HEAD` 和分支指针的位置。常见的版本回退方式有以下几种:

(1)git reset

`git reset` 通过移动 `HEAD` 和分支指针来回退版本

分为以下三种模式:

- soft 模式:

仅移动 `HEAD` 和分支指针, 不修改暂存区 (staging area) 和工作区 (working directory)。

适合需要保留更改但重新组织提交的场景。

```
git reset --soft <commit-hash>
```

- mixed 模式 (默认)

移动 `HEAD` 和分支指针, 同时重置暂存区, 使其与目标提交一致。

工作区不受影响。

```
git reset --mixed <commit-hash>
```

- hard 模式

移动 `HEAD` 和分支指针, 同时重置暂存区和工作区。

不可恢复, 谨慎使用!

```
git reset --hard <commit-hash>
```

(2)git revert

`git revert` 通过创建一个新的提交。

回滚指定提交的更改, 而不会更改提交历史。

原理: 基于目标提交的内容生成反向更改, 并作为一个新提交应用。

```
git revert <commit-hash>
```

(3)git checkout 和 git switch

临时查看某个提交的内容，不会更改分支指针。

```
git checkout <commit-hash>
```

推荐使用Switch

4. 底层数据结构支撑

Git 的回退操作依赖于其数据模型：

对象存储：Git 通过 blob（文件内容）、tree（目录结构）和 commit（版本信息）对象进行管理。
有向无环图（DAG）：Git 的提交历史以有向无环图组织，使得版本回退、合并等操作高效且易管理。

三种版本回退的应用场景

Git reset

应用场景：

- 当需要彻底移除某些提交记录，并且不希望在项目历史中保留痕迹。
- 适用于修改最近几次提交内容，重新组织提交记录。
- 有时用于回退代码到一个稳定的状态。

实际操作：

```
git reset [--soft | --mixed | --hard] <commit-hash>
```

--soft模式：

- 场景：回退版本，但保留代码的所有更改到暂存区。适合调整提交内容。
- 操作：`git reset --soft HEAD^`

将 HEAD 指针回退到上一个提交，保留代码在暂存区。

- 解释：对上次的提交不满意，直接reset到HEAD^ 更改的代码会保留在暂存区可以选择性再次提交。

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit f272e6dbf7354e9793d73c174c56623ee0e44ec (HEAD -> master)
Author: sun <sun@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ touch bumanyifile
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add .
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git commit -m "commit of bumanyi"
[master e4b2624] commit of bumanyi
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 bumanyifile
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit e4b2624b101c522c228250ffead200fc234a03b (HEAD -> master)
Author: sun <sun@linux.com>
Date:   Sat Nov 16 13:36:51 2024 +0800

    commit of bumanyi
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
bumanyifile  newFeature  newFeature2  newfile  newfile.txt  project  project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
无文件要提交，干净的工作区
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git reset --soft HEAD^
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
bumanyifile  newFeature  newFeature2  newfile  newfile.txt  project  project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
要提交的变更：
(使用 "git restore --staged <文件>..." 以取消暂存)
  新文件:  bumanyifile

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit f272e6dbf7354e9793d73c174c56623ee0e44ec (HEAD -> master)
Author: sun <sun@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ rm bumanyifile
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add .
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
无文件要提交，干净的工作区
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

类似撤回提交操作

--mixed模式(默认)

- 场景：回退版本，保留代码更改在工作区，但清空暂存区。适合修复暂存区不一致问题。

- 操作：`git reset --mixed HEAD~2`

HEAD~2 表示当前提交 (HEAD) 的第 2 个祖先提交。

(回退到倒数第二次提交，保留代码更改。)

- 解释：回退目标版本，HEAD指向，暂存区清理，工作目录不变

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit f272e6bdbf7354e9793d73c174c56623ee0e44ec (HEAD -> master)
Author: sun <sung@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ touch bumanyi
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add .
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git commit -m "bumanyi de tijiao"
[master 5f1384c] bumanyi de tijiao
 1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bumanyi
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit 5f1384c5f127944eaef9b34bfdf9709a9aa4d42 (HEAD -> master)
Author: sun <sung@linux.com>
Date:   Sat Nov 16 13:43:58 2024 +0800

    bumanyi de tijiao
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git re
base          reflog      remote      repack      replace      request-pull      reset      restore      revert
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git reset HEAD^
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
bumanyi  newfeature  newfeature2  newfile  newfile.txt  project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git status
位于分支 master
未跟踪的文件：
  (使用 "git add <文件>..." 以包含要提交的内容)

  bumanyi

提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit f272e6bdbf7354e9793d73c174c56623ee0e44ec (HEAD -> master)
Author: sun <sung@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

暂存区清零，工作目录不变，bumanyi文件属于未跟踪状态

--hard模式：

- 场景：彻底删除提交记录，同时丢弃所有代码更改。适合需要完全恢复到某版本的场景。危险操作！
- 操作：`git reset --hard <commit-hash>`
- 解释：彻底退回版本

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit f272e6bdbf7354e9793d73c174c56623ee0e44ec (HEAD -> master)
Author: sun <sung@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ echo helloworld > bumanyi
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add bumanyi
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git commit -m "bumnayi"
[master e598ae4] bumnyai
 1 file changed, 1 insertion(+)
create mode 100644 bumnyai
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit e598ae42571a024fdccfa2d9e25b91802b82a8 (HEAD -> master)
Author: sun <sung@linux.com>
Date:   Sat Nov 16 13:47:21 2024 +0800

    bumnyai
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
bumnyai  newFeature  newFeature2  newfile  newfile.txt  project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git reset --hard HEAD^
HEAD 现在位于 f272e6bb add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
Commit f272e6bdbf7354e9793d73c174c56623ee0e44ec (HEAD -> master)
Author: sun <sung@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ls
newFeature  newFeature2  newfile  newfile.txt  project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

Git revert

- 应用场景：
 - 当需要保留提交历史但撤销某次提交的更改时使用。
 - 常用于团队协作，避免因版本回退影响其他开发人员的历史。
- 实际操作

```
1 | # 生成一个新的提交，用于撤销指定提交的更改
2 | git revert <commit-hash>
```

```
sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/Myproject$ ls
bumanyi newFeature newFeature2 newfile newfile.txt project.txt
sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/Myproject$ git revert HEAD^
[master eaaec89] Revert "add project"
 1 file changed, 4 deletions(-)
sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/Myproject$ git log -n 2
commit eaaec89d726fce7e08975551f49476ba2f9395fa (HEAD -> master)
Author: sun <sun@linux.com>
Date:   Sat Nov 16 13:54:14 2024 +0800

    Revert "add project"

    This reverts commit f272e6bdbf7354e9793d73c174c56623ee0e44ec.

此次回退，表名放弃了不满意的上一次提交

commit fc29437b3946888f5f3d57160032f1fdc7db1393
Author: sun <sun@linux.com>
Date:   Sat Nov 16 13:50:48 2024 +0800

    bumanyi
sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/Myproject$
```

```
1 Revert "add project"
2
3 This reverts commit f272e6bdbf7354e9793d73c174c56623ee0e44ec.
4
5 # 请为您的变更输入提交说明。以 '#' 开始的行将被忽略，而一个空的提交
6 # 说明将会终止提交。
7 #
8 # 位于分支 master
9 # 要提交的变更：
10 #   修改： project.txt
11 #
12 此次回退，表名放弃了不满意的上一次提交
```

使用此回退不会修改git log日志内容

Git checkout / Git switch

- 应用场景：

用于临时查看代码在某个提交的状态，而不更改当前分支的提交历史。

适合调试或查看特定提交的状态，不会修改历史记录。

- 实际操作

```
1 # 检出到某个提交（进入“分离头指针”状态）
2 git checkout <commit-hash>
3 git checkout HEAD^
```

可以再次创建新的分支进行开发。

对比

回退方式	应用场景	是否保留历史	操作是否可逆
git reset	彻底删除提交记录，重新组织历史	不保留历史	部分可逆
git revert	撤销某次提交内容，但保留提交记录	保留历史	可逆
git checkout	临时查看某次提交的状态，不影响历史记录	无更改历史	无需回滚

HEAD的几种格式

HEADⁿ:表示当前分支的第n个父提交

HEAD[^]:当前分支的父提交

HEAD⁻ⁿ: 向上寻找第n层父提交

HEAD⁻¹等价于HEAD[^]

HEAD@{}:表示引用reflog中记录的HEAD位置

可以用追踪HEAD的历史变动

HEAD@{0}: 当前 HEAD 所在的位置。

HEAD@{n}: HEAD 在引用日志中的第 n 个历史位置 (n 越大, 时间越久远)。

HEAD:path/to/file

表示 某次提交中的特定文件。

用于查看或操作提交中的某个文件内容。

```
git show HEAD:src/main.c
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log -n 1
commit eaaec89d726fce7e08975551f49476ba2f9395fa (HEAD -> master)
Author: sun <sun@linux.com>
Date:   Sat Nov 16 13:54:14 2024 +0800

    Revert "add project"

    This reverts commit f272e6bdbf7354e9793d73c174c56623ee0e44ec.

}
此次回退, 表名放弃了不满意的上一次提交
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ echo "hello" > bumanyi
[sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git add .
[sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git commit -m "na"
[master cad64b5] na
 1 file changed, 1 insertion(+)
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git show HEAD^:./bumanyi
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

HEAD^{}

表示将提交对象解析为 对应的提交类型 (例如标记对象解析到实际的提交)。

通常用于操作标签 (tag) 对象, 转换为指向的实际提交。

HEAD^{^^}

表示 当前提交的第二级父提交 (等价于 HEAD⁻²)。

用于合并提交的更复杂父关系。

(commit):path/to/file

表示从特定提交中获取某个文件的内容。

可用于比较或恢复历史版本的文件。

```
(commit)^{tree}
```

表示 当前提交对应的树对象。

用于查看或操作整个目录结构。

示例：

```
git ls-tree HEAD^{tree}
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git ls-tree HEAD^{tree}
100644 blob ce013625030ba8dba906f756967f9e9ca394464a    bumanyi
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    newFeature
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    newFeature2
100644 blob abd5734314a7bfccfb871d478cd0fec60ac6d46a    newfile
100644 blob d5e48e6680088e020d03ea4cf40785108d050db8    newfile.txt
100644 blob 343160726ef35527ab812043f7b9ceea6e63ef1    project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git show HEAD^{tree}
tree HEAD^{tree}

bumanyi
newFeature
newFeature2
newfile
newfile.txt
project.txt
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

.git 目录结构

在 Git 中，.git 是一个隐藏目录，用于存储版本控制相关的所有数据和元信息。了解 .git 的目录结构对深入理解 Git 的工作原理非常有帮助

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ ll .git/
总用量 52K
drwxrwxr-x  2 sun sun 4.0K 11月 15 12:36 branches
-rw-rw-r--  1 sun sun     3 11月 16 14:10 COMMIT_EDITMSG
-rw-rw-r--  1 sun sun   106 11月 15 14:44 config
-rw-rw-r--  1 sun sun    73 11月 15 12:36 description
-rw-rw-r--  1 sun sun  1.7K 11月 15 17:39 gitk.cache
-rw-rw-r--  1 sun sun    23 11月 15 17:04 HEAD
drwxrwxr-x  2 sun sun 4.0K 11月 15 12:36 hooks
-rw-rw-r--  1 sun sun   529 11月 16 14:10 index
drwxrwxr-x  2 sun sun 4.0K 11月 15 12:36 info
drwxrwxr-x  3 sun sun 4.0K 11月 15 13:54 logs
drwxrwxr-x 60 sun sun 4.0K 11月 16 14:10 objects
-rw-rw-r--  1 sun sun    41 11月 16 13:53 ORIG_HEAD
drwxrwxr-x  4 sun sun 4.0K 11月 15 12:36 refs
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

```
1 .git/
2 └── HEAD
3 └── config
4 └── description
5 └── hooks/
6 └── info/
7 └── objects/
8 └── refs/
9 └── logs/
10 └── index
11 └── packed-refs
```

1. HEAD

作用：指向当前分支或提交的位置。

内容：

当你在某个分支上时，HEAD 指向对应分支的引用，例如：

```
ref: refs/heads/main
```

当你处于分离头指针状态时，HEAD 直接指向某个提交哈希值：

```
1b6d2aeabc1234567890abcdef1234567890abcd
```

2. config

作用：存储当前仓库的配置信息（如用户名、邮件地址、远程仓库 URL 等）。

内容：

```
1 [core]
2   repositoryformatversion = 0
3   filemode = true
4   bare = false
5 [remote "origin"]
6   url = https://github.com/user/repo.git
7   fetch = +refs/heads/*:refs/remotes/origin/*
```

3. description

作用：提供仓库的描述信息。

主要在裸仓库（bare repository）中使用，对普通 Git 操作无影响。

4. hooks/

作用：包含各种客户端或服务器端的钩子脚本。

常见钩子：

pre-commit: 提交前的检查脚本。

post-commit: 提交完成后的操作。

pre-push: 推送到远程仓库前的操作。

```
1 hooks/
2   └── pre-commit.sample
3   └── pre-push.sample
4   └── post-commit.sample
```

5. info/

作用：包含与当前仓库相关的额外信息。

文件：

exclude: 定义忽略的文件（类似 .gitignore）

```
1 # Patterns to exclude
2 *.log
3 *.tmp
```

6. objects/

作用：存储所有 Git 的数据对象，包括提交（commit）、树（tree）、文件（blob）等。

结构：

```
1 objects/
2   └── 1b/
3     └── 6d2aeabc1234567890abcdef1234567890abcd
4   └── 3c/
5     └── 9f1ab1234567890abcdef1234567890abcdef
6   └── pack/
7     └── pack-xxxx.pack
8     └── pack-xxxx.idx
```

前两位为子目录名，剩余的部分为文件名。

pack/ 子目录：存储被压缩的对象，用于提高效率。

7. refs/

作用：存储分支、标签等引用信息。

结构：

```
1 | refs/
2 |   └── heads/          # 本地分支
3 |     └── main
4 |       └── feature
5 |   └── remotes/         # 远程分支
6 |     └── origin/
7 |       └── main
8 |         └── feature
9 |   └── tags/            # 标签
10 |     └── v1.0
```

heads/：保存本地分支的引用。

remotes/：保存远程分支的引用。

tags/：保存标签的引用。

8. logs/

作用：保存引用日志（reflog），记录分支、HEAD 等引用的变动历史。

结构：

```
1 | logs/
2 |   └── HEAD          # 记录 HEAD 的变动历史
3 |   └── refs/
4 |     └── heads/      # 记录本地分支的变动历史
5 |       └── main
6 |         └── feature
7 |     └── remotes/    # 记录远程分支的变动历史
8 |       └── origin/
9 |         └── main
10 |           └── feature
```

9. index

作用：Git 的暂存区（staging area）的二进制文件，记录哪些文件被暂存。

细节：

包含文件路径、权限、内容哈希值等信息。

在每次执行 git add 时更新。

10. packed-refs

作用：存储被压缩的引用信息，减少磁盘占用。

内容示例：

```
1 | # pack-refs with: peeled fully-peeled sorted
2 | abc1234567890abcdef1234567890abcdef refs/heads/main
3 | def1234567890abcdef1234567890abcdef refs/tags/v1.0
```

九、Git标签

标签的使用

如果你达到一个重要的阶段，并希望永远记住提交的快照，你可以使用 git tag 给它打上标签。

Git 标签 (Tag) 用于给仓库中的特定提交点加上标记，通常用于发布版本（如 v1.0, v2.0）。

比如说，我们想为我们的 runoob 项目发布一个 "1.0" 版本，我们可以用 git tag -a v1.0 命令给最新一次提交打上 (HEAD) "v1.0" 的标签。

-a 选项意为"创建一个带注解的标签"，不用 -a 选项也可以执行的，但它不会记录这标签是啥时候打的，谁打的，也不会让你添加个标签的注解，我们推荐一直创建带注解的标签。

标签语法格式：

```
git tag <tagname>
```

示例：

```
git tag v1.0
```

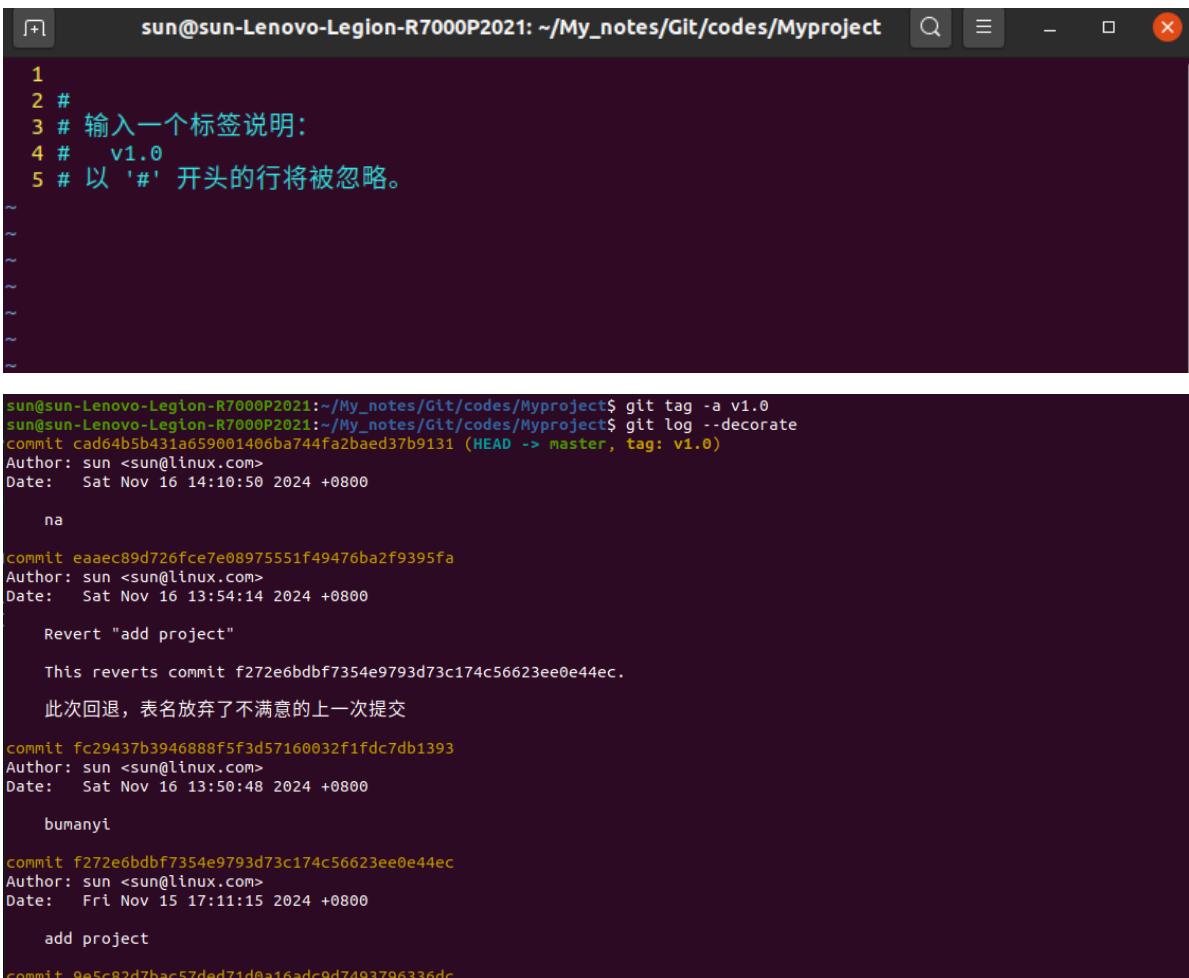
-a 选项可以添加注解

```
$ git tag -a v1.0
```

当你执行 git tag -a 命令时，Git 会打开你的编辑器，让你写一句标签注解，就像你给提交写注解一样。

现在，注意当我们执行 git log --decorate 时，我们可以看到我们的标签了

示例：



```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git tag -a v1.0
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --decorate
commit cad64b5b431a659001406ba744fa2baed37b9131 (HEAD -> master, tag: v1.0)
Author: sun <sun@linux.com>
Date:   Sat Nov 16 14:10:50 2024 +0800

    na

commit eaacec89d726fce7e08975551f49476ba2f9395fa
Author: sun <sun@linux.com>
Date:   Sat Nov 16 13:54:14 2024 +0800

    Revert "add project"

    This reverts commit f272e6bdbf7354e9793d73c174c56623ee0e44ec.

    此次回退，表名放弃了不满意的上一次提交

commit fc29437b3946888f5f3d57160032f1fdc7db1393
Author: sun <sun@linux.com>
Date:   Sat Nov 16 13:50:48 2024 +0800

    bumanyi

commit f272e6bdbf7354e9793d73c174c56623ee0e44ec
Author: sun <sun@linux.com>
Date:   Fri Nov 15 17:11:15 2024 +0800

    add project

commit 9e5c82d7bac57ded71d0a16adc9d7493796336dc
```

如果我们忘了给某个提交打标签，又将它发布了，我们可以给它追加标签。

例如，假设我们发布了提交 85fc7e7(上面实例最后一行)，但是那时候忘了给它打标签。我们现在也可以：

```
git tag -a v0.9 <commit-hash>
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git tag -a v0.5 9e5c82d
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --decorate --oneline --graph
* cad64b5 (HEAD -> master, tag: v1.0) na
* eaaec89 Revert "add project"
* fc29437 bumanyi
* f272e6b add project
* 9e5c82d (tag: v0.5) Merge branch 'Feature' through this merge,we finish touch two files!
|\ \
| * b19c4df (Feature) new file 2
| * ded709b new file one
* | 2b1d62f (NewVersion) master 3
|\ \
| * | 1acf61 feature 1
| * | 1bc61d0 master 2
* | | 9570605 master 1
* | | a55bcds master deal merge problem
|\ \
| |
| / \
| / \
| * | 015b34f Feature modify
* | | 05aa8cd modify again
| |
| / \
* | a87b3b2 modify newfile.txt
|
* 4bd6944 modify project.txt
* 3ae9f3e add newfile.txt
* d76e891 test
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

如果我们要查看所有标签可以使用以下命令：

```
git tag
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git tag
v0.5
v1.0
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

推送标签到远程仓库

默认情况下，git push 不会推送标签，你需要显式地推送标签。

```
git push origin <tagname>
```

推送所有标签

```
git push origin --tags
```

删除标签

本地删除：

```
git tag -d <tagname>
```

远程删除：

```
git push origin --delete <tagname>
```

附注标签

附注标签存储了创建者的名字、电子邮件、日期，并且可以包含标签信息。

附注标签更为正式，适用于需要额外元数据的场景。

创建附注标签语法：

```
git tag -a <tagname> -m "message"
```

示例：

```
git tag -a <tagname> -m "runoob.com标签"
```

PGP 签名标签命令：

```
git tag -s <tagname> -m "runoob.com标签"
```

查看标签信息：

```
git show <tagname>
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git tag -a v1.1 -m "sun release"
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git log --decorate --oneline
cad64b5 (HEAD -> master, tag: v1.1, tag: v1.0) na
eaaec89 Revert "add project"
fc29437 bumanyi
f272e6b add project
9e5c82d (tag: v0.5) Merge branch 'Feature' through this merge,we finish touch two files!
b19c4df (Feature) new file 2
ded709b new file one
2b1d62f (NewVersion) mastor 3
1acf61 feature 1
1bc61d0 master 2
9570605 master 1
e55bcd5 mastor deal merge problem
015b34f Feature modify
05aa8cd modify again
a87b3b2 modify newfile.txt
4bd6944 modify project.txt
3ae9f3e add newfile.txt
d76e891 test
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git show v1.1
tag v1.1
Tagger: sun <sun@linux.com>
Date:   Sat Nov 16 15:35:22 2024 +0800

sun release

commit cad64b5b431a659001406ba744fa2baed37b9131 (HEAD -> master, tag: v1.1, tag: v1.0)
Author: sun <sun@linux.com>
Date:   Sat Nov 16 14:10:50 2024 +0800

na

diff --git a/bumanyi b/bumanyi
index e69de29..ce01362 100644
--- a/bumanyi
+++ b/bumanyi
@@ -0,0 +1 @@
+hello
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

示例：

以下是一个综合示例，演示如何创建、查看、推送和删除标签。

创建轻量标签和附注标签：

```
1 | git tag v1.0
2 | git tag -a v1.1 -m "runoob.com标签"
```

查看标签和标签信息：

```
1 | git tag
2 | git show v1.1
```

推送到远程仓库：

```
1 | git push origin v1.0
2 | git push origin v1.1
3 | git push origin --tags # 推送所有标签
```

删除标签

本地删除：

```
git tag -d v1.0
```

远程删除：

```
git push origin --delete v1.0
```

十、Git Flow

Git Flow 是一种基于 Git 的分支模型，旨在帮助团队更好地管理和发布软件。

Git Flow 由 Vincent Driessen 在 2010 年提出，并通过一套标准的分支命名和工作流程，使开发、测试和发布过程更加有序和高效。

Git Flow 主要由以下几类分支组成：master、develop、feature、release、hotfix。

Git Flow安装

包安装：

Linux

Debian/Ubuntu:

```
sudo apt-get install git-flow
```

Fedora:

```
1 sudo dnf install gitflow  
2 sudo apt-get install git-flow
```

macOS

在 macOS 上，你可以使用 Homebrew 来安装 Git Flow:

```
brew install git-flow
```



```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ sudo apt install git-flow  
[sudo] sun 的密码：  
正在读取软件包列表... 完成  
正在分析软件包的依赖关系树  
正在读取状态信息... 完成  
下列【新】软件包将被安装：  
git-flow  
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 404 个软件包未被升级。  
需要下载 38.8 kB 的归档。  
解压缩后会消耗 330 kB 的额外空间。  
获取：1: https://mirrors.ustc.edu.cn/ubuntu focal/universe amd64 git-flow all 1.12.3-1 [38.8 kB]  
已下载 38.8 kB, 耗时 0 秒 (112 kB/s)  
正在选中未选择的软件包 git-flow。  
(正在读取数据库 ... 系统当前共安装有 277408 个文件和目录。)  
准备解压 .../git-flow_1.12.3-1_all.deb ...  
正在解压 git-flow (1.12.3-1) ...  
正在设置 git-flow (1.12.3-1) ...  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$ git version  
git version 2.25.1  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/Myproject$
```

源码安装

```
1 git clone https://github.com/nvie/gitflow.git  
2 cd gitflow  
3 sudo make install
```

验证安装

```
git flow version
```

Windows下安装

在 Windows 上，你可以通过以下方式安装 Git Flow：

- 使用 Git for Windows: Git for Windows 包含了 Git Flow。你可以从 Git for Windows 安装 Git，然后使用 Git Bash 来使用 Git Flow。

- 使用 Scoop: 如果你使用 Scoop 包管理工具, 可以通过以下命令安装 Git Flow:

```
scoop install git-flow
```

- 使用 Chocolatey: 如果你使用 Chocolatey 包管理工具, 可以通过以下命令安装 Git Flow:

```
choco install gitflow
```

Git Flow分支模型

master 分支:

永远保持稳定和可发布的状态。

每次发布一个新的版本时, 都会从 develop 分支合并到 master 分支。

develop 分支:

用于集成所有的开发分支。

代表了最新的开发进度。

功能分支、发布分支和修复分支都从这里分支出去, 最终合并回这里。

feature 分支:

用于开发新功能。

从 develop 分支创建, 开发完成后合并回 develop 分支。

命名规范: feature/feature-name。

release 分支:

用于准备新版本的发布。

从 develop 分支创建, 进行最后的测试和修复, 然后合并回 develop 和 master 分支, 并打上版本标签。

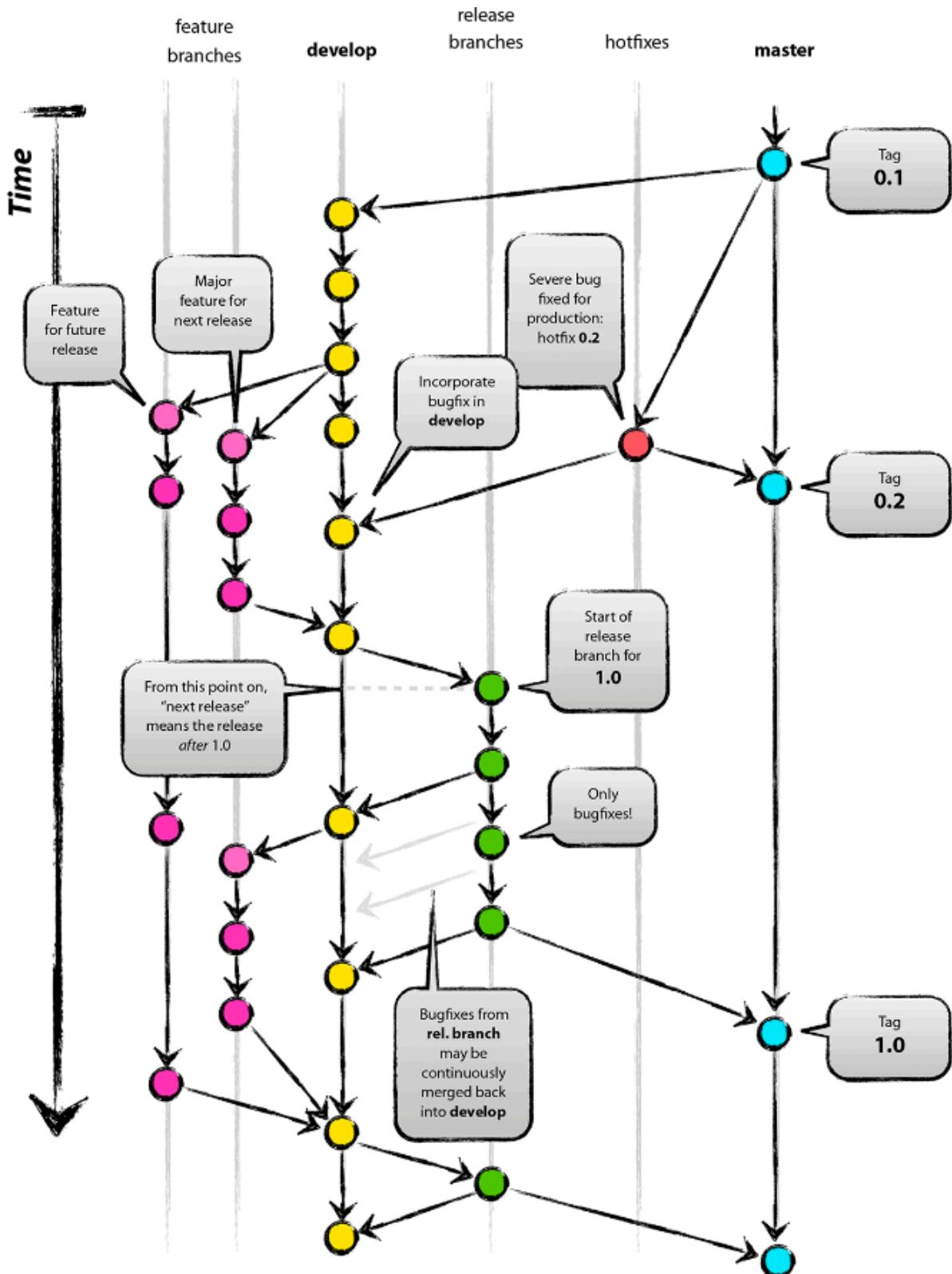
命名规范: release/release-name。

hotfix 分支:

用于修复紧急问题。

从 master 分支创建, 修复完成后合并回 master 和 develop 分支, 并打上版本标签。

命名规范: hotfix/hotfix-name。



分支操作原理

- Master 分支上的每个 Commit 应打上 Tag, Develop 分支基于 Master 创建。
- Feature 分支完成后合并回 Develop 分支，并通常删除该分支。
- Release 分支基于 Develop 创建，用于测试和修复 Bug，发布后合并回 Master 和 Develop，并打 Tag 标记版本号。
- Hotfix 分支基于 Master 创建，完成后合并回 Master 和 Develop，并打 Tag 1。

Git Flow命令示例：

开始 Feature 分支: `git flow feature start MYFEATURE`

完成 Feature 分支: `git flow feature finish MYFEATURE`

开始 Release 分支: `git flow release start RELEASE [BASE]`

完成 Release 分支: 合并到 Master 和 Develop, 打 Tag, 删除 Release 分支。

开始 Hotfix 分支: `git flow hotfix start HOTFIX [BASE]`

完成 Hotfix 分支: 合并到 Master 和 Develop, 打 Tag, 删除 Hotfix 分支。

Git Flow工作流程

工作流程

1. 初始化 Git Flow

首先, 在项目中初始化 Git Flow。可以使用 Git Flow 插件 (例如 git-flow) 来简化操作。

```
git flow init
```

初始化时, 你需要设置分支命名规则和默认分支。

必须先初始化git init

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow init
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [/home/sun/My_notes/Git/codes/.git/hooks]
```

2. 创建功能分支

当开始开发一个新功能时, 从 develop 分支创建一个功能分支。

```
git flow feature start feature-name
```

完成开发后, 将功能分支合并回 **develop** 分支, 并删除功能分支。

```
git flow feature finish feature-name
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch
* develop
  master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow feature start feature-c
切换到一个新分支 'feature/feature-c'

Summary of actions:
- A new branch 'feature/feature-c' was created, based on 'develop'
- You are now on branch 'feature/feature-c'

Now, start committing on your feature. When done, use:

  git flow feature finish feature-c

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ touch hello.c
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ vim hello.c
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ gcc hello.c -o chengfa
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ ./chengfa
15 4
60sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ vim he
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ vim hello.c
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ gcc hello.c -o chengfa
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ ./chengfa
45 8
360
98 5
490
147 56
8232
^C
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ ls
chengfa  hello.c
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow feature finish feature-c
切换到分支 'develop'
已经是最新的。
已删除分支 feature/feature-c (曾为 09df72f)。

Summary of actions:
- The feature branch 'feature/feature-c' was merged into 'develop'
- Feature branch 'feature/feature-c' has been locally deleted
- You are now on branch 'develop'

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch
* develop
  master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$
```

自动完成删除与合并功能

3. 创建发布分支

当准备发布一个新版本时，从 develop 分支创建一个发布分支。

```
git flow release start release-name
```

在发布分支上进行最后的测试和修复，准备好发布后，将发布分支合并回 **develop** 和 **master** 分支，并打上版本标签。

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow release start release-1.0
切换到一个新分支 'release/release-1.0'

Summary of actions:
- A new branch 'release/release-1.0' was created, based on 'develop'
- You are now on branch 'release/release-1.0'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

    git flow release finish 'release-1.0'

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ ./chengfa
45 5
225
^C
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow release finish release-1.0
切换到分支 'master'
已删除分支 release/release-1.0 (曾为 09df72f) 。

Summary of actions:
- Release branch 'release/release-1.0' has been merged into 'master'
- The release was tagged 'release-1.0'
- Release branch 'release/release-1.0' has been locally deleted
- You are now on branch 'master'

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch
  develop
* master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$
```

测试完成后，打上版本标签会自动合并到develop和master

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch
  develop
* master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git log
commit 09df72fa98b48f01daef92c6e95aa4fe32529b6b (HEAD -> master, tag: release-1.0, develop)
Author: sun <sun@linux.com>
Date:   Sat Nov 16 16:00:34 2024 +0800

  Initial commit
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$
```

4. 创建修复分支

当发现需要紧急修复的问题时，从 master 分支创建一个修复分支。

```
git flow hotfix start hotfix-name
```

修复完成后，将修复分支合并回**master** 和 **develop** 分支，并打上版本标签。

```
git flow hotfix finish hotfix-name
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow hotfix start hotfix-changeadd  
切换到一个新分支 'hotfix/hotfix-changeadd'  
  
Summary of actions:  
- A new branch 'hotfix/hotfix-changeadd' was created, based on 'master'  
- You are now on branch 'hotfix/hotfix-changeadd'  
  
Follow-up actions:  
- Start committing your hot fixes  
- Bump the version number now!  
- When done, run:  
    git flow hotfix finish 'hotfix-changeadd'  
  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ ls  
chengfa  hello.c  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ vim hello.c  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ gcc hello.c -o add  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git flow hotfix finish hotfix-changeadd  
切换到分支 'develop'  
已删除分支 hotfix/hotfix-changeadd (曾为 09df72f)。  
  
Summary of actions:  
- Hotfix branch 'hotfix/hotfix-changeadd' has been merged into 'master'  
- The hotfix was tagged 'hotfix-changeadd'  
- Hotfix branch 'hotfix/hotfix-changeadd' has been locally deleted  
- You are now on branch 'develop'  
  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch  
* develop  
  master  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git log  
commit 09df72fa98b48f01daf992c6e95aa4fe32529b6b (HEAD -> develop, tag: release-1.0, tag: hotfix-changeadd, master)  
Author: sun <sun@linux.com>  
Date:   Sat Nov 16 16:00:34 2024 +0800  
  
    Initial commit  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$
```

示例

以下是一个实际使用 Git Flow 的综合实例。

- 初始化 Git Flow:

```
git flow init
```

- 创建和完成功能分支:

```
1 | git flow feature start new-feature # 开发新功能  
2 | git flow feature finish new-feature
```

- 创建和完成发布分支:

```
1 | git flow release start v1.0.0 # 测试和修复  
2 | git flow release finish v1.0.0
```

- 创建和完成修复分支:

```
1 | git flow hotfix start hotfix-1.0.1. # 修复紧急问题  
2 | git flow hotfix finish hotfix-1.0.1
```

优点和缺点

优点

```
1 | 明确的分支模型: 清晰的分支命名和使用规则, 使得开发过程井然有序。  
2 | 隔离开发和发布: 开发和发布过程分离, 减少了开发中的不确定性对发布的影响。  
3 | 版本管理: 每次发布和修复都会打上版本标签, 方便回溯和管理。
```

缺点

- 1 复杂性：对于小型团队或简单项目，**Git Flow** 的分支模型可能显得过于复杂。
- 2 频繁的合并：在大型团队中，频繁的分支合并可能导致合并冲突增加。

Git Flow 是一种结构化的分支管理模型，通过定义明确的分支和工作流程，帮助团队更好地管理软件开发和发布过程。

虽然它增加了一定的复杂性，但对于大型项目和团队协作，Git Flow 提供了强大的支持和管理能力。

十一、Git进阶操作

在掌握了 Git 的基础操作之后，进阶操作可以帮助你更高效地管理和优化你的代码库。

以下是一些常见的进阶操作及其详细说明：

- 1 交互式暂存：逐块选择要暂存的更改，精细控制提交内容。
- 2 **Git Stash**：临时保存工作进度，方便切换任务。
- 3 **Git Rebase**：将一个分支上的更改移到另一个分支之上，保持提交历史线性。
- 4 **Git Cherry-Pick**：选择特定提交并应用到当前分支。

交互式暂存

git add 命令可以选择性地将文件或文件的一部分添加到暂存区，这在处理复杂更改时非常有用。

使用 git add -p：逐块选择要暂存的更改

```
git add -p
```

执行此命令后，Git 会逐块显示文件的更改，你可以选择是否暂存每个块。常用选项包括：

选项	描述
y	暂存当前块
n	跳过当前块
s	拆分当前块
e	手动编辑当前块
q	退出暂存

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git add -p
diff --git a/pystr.py b/pystr.py
index 8d90b86..5d9d51f 100755
--- a/pystr.py
+++ b/pystr.py
@@ -1,3 +1,16 @@
#!/bin/python3

print("a + b = a+b")
+print("a + b = a+b")
(1/1) Stage this hunk [y,n,q,a,d,e,?]? s
Sorry, cannot split this hunk
@@ -1,3 +1,16 @@
#!/bin/python3

print("a + b = a+b")
+print("a + b = a+b")
```

git stash 临时保存工作进度

git stash 命令允许你临时保存当前工作目录的更改，以便你可以切换到其他分支或处理其他任务。

保存当前工作进度：

```
git stash
```

查看存储的进度：

```
git stash list
```

应用最近一次存储的进度：

```
git stash apply
```

应用并删除最近一次存储的进度：

```
git stash pop
```

删除特定存储：

```
git stash drop stash@{n}
```

清空所有存储：

```
git stash clear
```

```

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch
* develop
* feature/feature-jian
  master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git checkout master
error: 您对下列文件的本地修改将被检出操作覆盖:
      pystr.py
请在切换分支前提交或贮藏您的修改。
正在终止
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git branch
* develop
* feature/feature-jian
  master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git stash
保存工作目录和索引状态 WIP on feature-jian: a8dde6b xiuh
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git checkout master
切换到分支 'master'
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ 

```

不提前保存存储，是无法切换工作分支的

Git Rebase

git rebase 命令用于将一个分支上的更改移到另一个分支之上。它可以帮助保持提交历史的线性，减少合并时的冲突。

变基当前分支到指定分支：

```
git rebase <branchname>
```

例如，将当前分支变基到 main 分支：

```
git rebase main
```

交互式变基：

```
git rebase -i <commit>
```

交互式变基允许你在变基过程中编辑、删除或合并提交。常用选项包括：

选项	操作
pick	保留提交
reword	修改提交信息
edit	编辑提交
squash	将当前提交与前一个提交合并
fixup	将当前提交与前一个提交合并，不留提交信息
drop	删除提交

工作场景：

在多人协作开发中，主分支（如 main）经常更新，而你的功能分支（如 feature）可能落后于主分支。你需要将主分支的最新更改合并到功能分支中。

传统方法：merge

```

1 | git checkout feature
2 | git merge main

```

结果：

历史记录中会出现一个合并提交（merge commit）。

提交记录显示了分支的分叉和合并。

rebase 方法

```
1 | git checkout feature  
2 | git rebase main
```

结果：

功能分支的提交会被重新定位到主分支的最新状态。

历史记录变得线性，没有额外的合并提交。

适用场景：

希望保持历史记录清晰、线性，便于审查代码和回溯问题。

Git Cherry-Pick:挑选提交

git cherry-pick 命令允许你选择特定的提交并将其应用到当前分支。

它在需要从一个分支移植特定更改到另一个分支时非常有用。

在该分支获取其他分支的特定提交部分。

拣选提交：

```
git cherry-pick <commit>
```

例如，将 abc123 提交应用到当前分支：

```
git cherry-pick abc123
```

处理拣选冲突：如果拣选过程中出现冲突，解决冲突后使用

```
git cherry-pick --continue
```

继续拣选。

```
Initial commit  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ git cherry-pick 68d4881c059dc824d0fc8672f7a693e1df4b31f4  
[name d86cc84] add name  
Date: Sat Nov 16 17:25:33 2024 +0800  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 name.b  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$ ls  
add chengfa hello.c lianxi name.b pystr.py  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/gitflow$
```

master的提交的name.b

被name分支，通过cherry-pick加载到当前分支

综合示例：

以下是一个综合示例，展示了如何使用这些进阶操作：

交互式暂存：

```
git add -p
```

保存工作进度：

```
git stash
```

查看存储的进度：

```
git stash list
```

应用存储的进度：

```
git stash apply
```

变基当前分支到 main 分支：

```
git rebase main
```

交互式变基，编辑提交历史：

```
git rebase -i HEAD~3
```

编辑提交历史，如合并和重命名提交。

拣选 feature 分支上的特定提交到 main 分支：

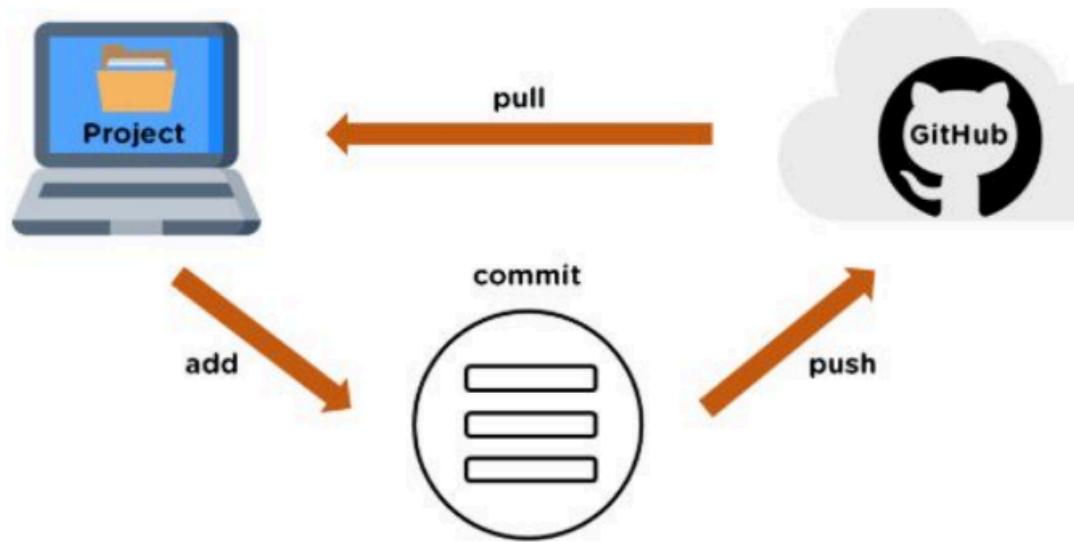
```
1 | git checkout main  
2 | git cherry-pick abc123
```

十二、远程仓库Github

Git 并不像 SVN 那样有个中心服务器。

目前我们使用到的 Git 命令都是在本地执行，如果你想通过 Git 分享你的代码或者与其他开发人员合作。你就需要将数据放到一台其他开发人员能够连接的服务器上。

(需要一台其他人员能够ping的服务器)



添加远程库

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来引用，命令格式如下：

```
git remote add [shortname] [url]
```

本例以 Github 为例作为远程仓库，如果你没有 Github 可以在官网 <https://github.com/> 注册。

由于你的本地 Git 仓库和 GitHub 仓库之间的传输是通过SSH加密的，所以我们需要配置验证信息：(采用 ssh 加密)

使用以下命令生成 SSH Key：

```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

-t 用于指定加密算法 -C 为注释 -f 可以指定路径文件名

```

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/SSH$ ssh-keygen -t rsa -C "sun@github.com" -f "github_rsa"
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in github_rsa
Your public key has been saved in github_rsa.pub
The key fingerprint is:
SHA256:s7970pwFHHh5uGzxlRkW1XTDmIfMSVmW8CRUHgn2aLE sun@github.com
The key's randomart image is:
+---[RSA 3072]----+
|          =0/@B|
| . o.X@0=|
| . * .E=+ |
| + *..   |
| S * .    |
| + .      |
| . .      |
| o o.    |
| .O=     |
+---[SHA256]-----+
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/SSH$ ls
codes  github_rsa  github_rsa.pub  images  ssh.md

```

注释的内容最后会加入到公钥的结尾，常用语标识所有者

至此，我们拥有了一对SSH密钥。我们需要把公钥上传到Github上

进入，Github设置 (Settings)

选择new SSH keys

**hongtiansun (hongtiansun)**

Your personal account

[Go to your personal profile](#)[Public profile](#)[Account](#)[Appearance](#)[Accessibility](#)[Notifications](#)[Access](#)[Billing and plans](#)[Emails](#)[Password and authentication](#)[Sessions](#)**SSH and GPG keys**[Organizations](#)[Enterprises](#)[Moderation](#)

SSH keys

[New SSH key](#)

There are no SSH keys associated with your account.

Check out our guide to [connecting to GitHub using SSH keys](#) or troubleshoot [common SSH problems](#).

GPG keys

[New GPG key](#)

There are no GPG keys associated with your account.

Learn how to [generate a GPG key and add it to your account](#).

Vigilant mode

 Flag unsigned commits as unverified

This will include any commit attributed to your account but not signed with your GPG or S/MIME key.

Note that this will include your existing unsigned commits.

[Learn about vigilant mode](#).

title 可以随便起

Key 粘贴公钥

SSH keys

[New SSH key](#)

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

Authentication keys

**Linux_Untuntu**

SHA256:s7976pwFHHh5uGzx1RkW1XTDmIfMSVmW8CRUHgn2aLE

[Delete](#)

Added on Nov 16, 2024

Never used — Read/write

Check out our guide to [connecting to GitHub using SSH keys](#) or troubleshoot [common SSH problems](#).

至此，成功在Github里添加公钥。

然后我们可以进行验证连接是否成功：

```
ssh -T git@github.com
```

其中-T表示无需进入交互式终端

```
sun@sun-Lenovo-Legion-R7000P2021:~/ssh$ ssh -T git@github.com
The authenticity of host 'github.com (20.205.243.166)' can't be established.
ECDSA key fingerprint is SHA256:p2QAMXNIC1TJYWeI0ttrVc98/R1BUFWu3/LiyKgUfQM.
Are you sure you want to continue connecting (yes/no/[fingerprint])? 
```

连接成功后获得github的服务器指纹，存储在known_hosts中

```
sun@sun-Lenovo-Legion-R7000P2021:~/ssh
```

```
1 |1|ORM90HcTC+jIhJBeRoVEWqryAsk=|+8zUffTYf34YdE7t0zkAs68V5Qc= ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTibmlzdHAyNTYAAAIBmlzdHAyNTYAAABBBBYaOaAIbWniCWJu8EAIk5e+hZYnF1cZnWqvrV2C/mfWMw4WPArfzBGxI5ZYWtb83oQciVs2W1aW9R1uwkWt8=
2 |1|QsLWXtzW39sJCoaU//C18eTJ8VI=|3hfz0a/Q7/MeEbJU25bqCBqpxPM= ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTibmlzdHAyNTYAAAIBmlzdHAyNTYAAABBBEmKSENjQEez0mxkZMy7opKgwFB9nkt5YRrYMjNuG5N87uRgg6CLRboSwAdT/y6v0mKV0U2w0WZ2YB/+TpoCKg=
3 |1|8s9JB6/D1TrnxzSu4Y0T4bHf08M=|3j0VqyJwrkz5fBpdv/+w/kTUK1M= ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTibmlzdHAyNTYAAAIBmlzdHAyNTYAAABBBEmKSENjQEez0mxkZMy7opKgwFB9nkt5YRrYMjNuG5N87uRgg6CLRboSwAdT/y6v0mKV0U2w0WZ2YB/+TpoCKg=
```

到此为止。我们就完成了与github的连接。

接下来，我们可以创建远程库。

The screenshot shows the GitHub Dashboard. On the left, there's a sidebar with 'Create your first project' and two buttons: 'Create repository' (highlighted in green) and 'Import repository'. The main area is titled 'Home' and has a section 'Start a new repository for hongtiansun'. A dropdown menu is open on the right, listing several options: 'New repository' (highlighted with a red box), 'Import repository', 'New codespace', 'New gist', 'New organization', and 'New project'. The 'New repository' option is the top item in the list.

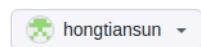
进行库描述

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner *



Repository name *

/ My_notes

My_notes is available.

Great repository names are short and memorable. Need inspiration? How about [curly-invention](#) ?

Description (optional)

This is a note made by myself.All content comes from network.Only help me to use!

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None ▾

接下来，我们就有了自己的库
我们可以使用ssh连接，或者http进行连接

Quick setup — if you've done this kind of thing before

HTTPS SSH https://github.com/hongtiansun/My_notes.git

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# My_notes" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/hongtiansun/My_notes.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/hongtiansun/My_notes.git
git branch -M main
git push -u origin main
```

我们接下来就可以把本地库直接上传到Github了

```
1 # 提交到 Github
2 $ git remote add origin git@github.com:tianqixin/runoob-git-test.git
3 $ git push -u origin master
```

远程服务器命名规则：

git@github.com:账户名/库名.git

```
git remote add lib名 服务器地址 创建远水库
git push -u lib名 master 把master分支push上去
```

-u：这是一个选项，用于设置默认的上游仓库。

这样以后就可以直接使用 git push 命令而不需要每次都指定远程仓库和分支。

The screenshot shows a GitHub repository named 'Test'. It contains a single file, 'README.md', which has the content '测试实验' and '无实际意义'. The repository statistics on the right indicate 0 stars, 1 watching, and 0 forks. There are no releases published.

成功push。

查看当前远程库

git remote

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git remote  
Test  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git remote -v  
Test    git@github.com:hontiansun/Test.git (fetch)  
Test    git@github.com:hontiansun/Test.git (push)  
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$
```

Test是我们在给远程库起的别名

提取远程仓库

Git 有两个命令用来提取远程仓库的更新。

- ## 1. 从远程仓库下载新分支与数据：

git fetch

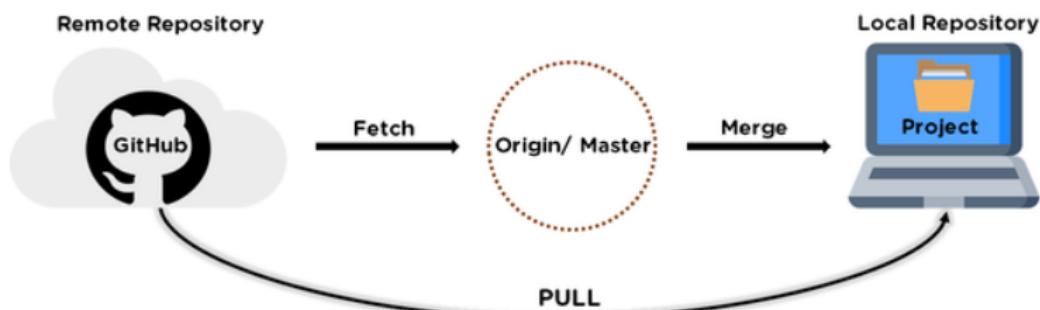
该命令执行完后需要执行 git merge 远程分支到你所在的分支。

- ## 2. 从远端仓库提取数据并尝试合并到当前分支：

git merge

该命令就是在执行 git fetch 之后紧接着执行 git merge 远程分支到你所在的任意分支。

总体过程就是，获取远程分支，合并到主分支。



假设你配置好了一个远程仓库，并且你想要提取更新的数据，你可以首先执行 `git fetch [alias]` 告诉 Git 去获取它有你没有的数据。

然后你可以执行 `git merge [alias]/[branch]` 以将服务器上的任何更新（假设有人这时候推送到服务器了）合并到你的当前分支。

示例：

Github 上远程修改了文档

Preview	Code	Blame	Raw	Copy	Download	Edit	More
Older  Newer						 Contributors	1
12 minutes ago	Test		1	*****	测试实验		
			2		无实际意义		
			3				
			4	*****			
now	 Update README.md		5	*****	继续修改		
			6		无实际意义		
			7				
			8				
			9	*****			

本地修改

```

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git fetch Test
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
展开对象中: 100% (3/3), 953 字节 | 953.00 KiB/s, 完成.
来自 github.com:hongtiansun/Test
  83d1c0c..f4dee2a master -> Test/master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git branch
* master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git merge Test/master
更新 83d1c0c..f4dee2a
Fast-forward
 README.md | 5 +++++
 1 file changed, 5 insertions(+)
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ cat README.md
*****
    测试实验
    无实际意义
*****
*****
    继续修改
    无实际意义
*****
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$
```

远程分支不会直接加到本地分支上，但是可以进行合并。

推送远程仓库

```
git push [alias] [branch]
```

alias 代表远程仓库名

branch 代表你的本地分支

可以使用 -u 记录分支和远程仓库名。

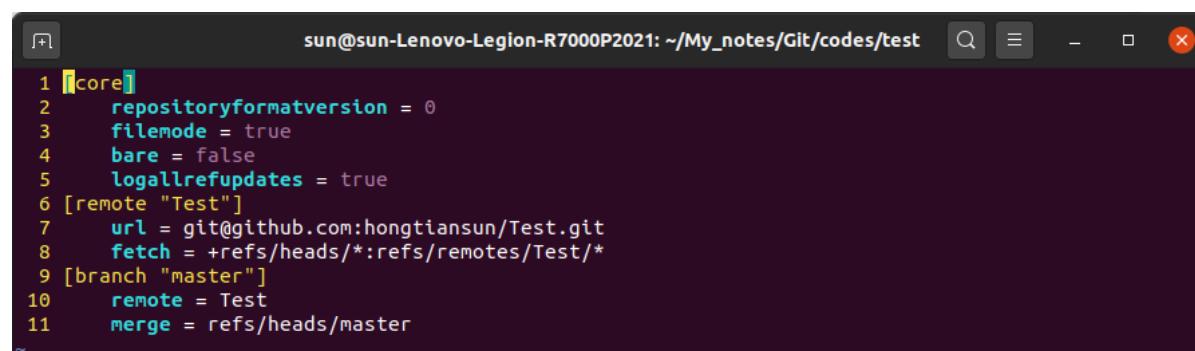
下次push直接git push即可

```

sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ vim README.md
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git add .
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git commit -m "xiugaierci"
[master 783eb24] xiugaierci
 1 file changed, 5 insertions(+)
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git push
枚举对象: 5, 完成.
对对象计数中: 100% (5/5), 完成.
使用 12 个线程进行压缩
压缩对象中: 100% (2/2), 完成.
写入对象中: 100% (3/3), 288 字节 | 288.00 KiB/s, 完成.
总共 3 (差异 1), 复用 0 (差异 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:hongtiansun/Test.git
  f4dee2a..783eb24 master -> master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$
```

原因在于

-u选项可以修改config进行配置



```

sun@sun-Lenovo-Legion-R7000P2021: ~/My_notes/Git/codes/test
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[remote "Test"]
url = git@github.com:hongtiansun/Test.git
fetch = +refs/heads/*:refs/remotes/Test/*
[branch "master"]
remote = Test
merge = refs/heads/master
~
```

删除远程仓库

```
git remote rm [库名]
```

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ vim .git/config
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git remote rm Test
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git remote
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git remote -v
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$
```

成功删除。

十三、Github教程

github是一个基于git的代码托管平台，付费用户可以建私人仓库。
我们一般的免费用户只能使用公共仓库，也就是代码要公开。

Github由Chris Wanstrath, PJ Hyett与Tom Preston-Werner三位开发者在2008年4月创办。
迄今拥有59名全职员工，主要提供基于git的版本托管服务。

目前看来，GitHub这场冒险已经胜出。

Github配置

见远程仓库上文

需要利用ssh连接，将公钥贴在Github上用以身份验证。

检出仓库

```
git clone /path/to/repository
```

可以克隆出一个本地仓库的克隆版本

```
git clone username@host:/path/to/repository
```

可以直接clone一个远程仓库

获取仓库

上文提到第一种方式，fetch and merge

```
git fetch origin <remote_branch_name>
```

现在我们使用第二种方式。

```
git pull origin <remote_branch_name>
```

从远程仓库origin中拉取remote_branch_name，并与当前本地分支进行合并

推送仓库

```
git push [alias] [branch]
```

即可

可以用-u参数设置remote，

下此使用直接git pull / git push 即可

查看远程仓库的分支

查看远程仓库分支

```
git branch -r
```

结果示例：

```
1 | origin/main
2 | origin/feature-branch
```

查看本地和远程分支

```
git branch -a
```

示例：

```
1 | * main
2 |   feature-branch
3 |   remotes/origin/main
4 |   remotes/origin/feature-branch
```

如果要指定远程仓库：

```
git ls-remote <remote-name>
```

即可

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$ git ls-remote Test
783eb24ca2db76e0800f427669927b9fffd3acd0      HEAD
783eb24ca2db76e0800f427669927b9fffd3acd0      refs/heads/master
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/test$
```

refs/heads/master

1. refs

refs 是 Git 中存储引用的顶层目录，引用是指向特定对象（通常是提交）的指针。

引用的类型：

refs/heads/：表示本地分支。

refs/remotes/：表示远程分支。

refs/tags/：表示标签。

refs/stash：表示临时存储的更改。

2. heads

heads 是一个子目录，表示当前存储的是本地分支。

在 Git 中，每一个本地分支的名称都会存储在 refs/heads/ 目录中。例如：

refs/heads/main：表示本地的 main 分支。

refs/heads/feature：表示本地的 feature 分支。

3. master

master 是分支的名字，这是 Git 默认的主分支名称（早期版本的 Git 默认主分支名为 master，较新的版本已经改为 main，但 master 依然常见）。

在路径中，master 表示一个具体的分支名称。

remote config 配置：

```
1 | 复制代码
2 | [remote "Test"]
3 |   url = git@github.com:hongtiansun/Test.git
4 |   fetch = +refs/heads/*:refs/remotes/Test/*
```

[remote "Test"]

定义了一个名为 Test 的远程仓库。

url = git\@github.com:hongtiansun/Test.git

指定了 Test 远程仓库的 URL，使用的是 SSH 协议。

这里表示远程仓库的地址为：git\@github.com:hongtiansun/Test.git。

fetch = +refs/heads/:refs/remotes/Test/

定义了从远程仓库拉取 (fetch) 的引用映射规则：

+：表示强制更新本地引用，即使远程分支被强制推送 (rebase 或修改历史)。

refs/heads/：匹配远程仓库的所有分支。

refs/remotes/Test/：将这些远程分支映射到本地的 Test 名称空间中（远程分支的本地跟踪分支）。

```
1 | [branch "master"]
2 |   remote = Test
3 |   merge = refs/heads/master
```

[branch "master"]

定义了本地分支 master 的相关配置。

remote = Test

指定本地 master 分支与远程 Test 仓库关联。

merge = refs/heads/master

表示本地 master 分支默认跟踪远程仓库 Test 中的 master 分支。

当你执行 git pull 或 git push 时，默认操作的是 Test 仓库的 master 分支。

十四、Git服务器搭建

我们远程仓库使用了 Github，Github 公开的项目是免费的，2019 年开始 Github 私有存储库也可以无限制使用。

当然我们也可以自己搭建一台 Git 服务器作为私有仓库使用。

使用裸存储库 (Bare Repository)

登录服务器

1. 安装git

Ubuntu 服务器上安装 Git：

```
sudo apt install git
```

其余系统请见上文

2. 创建git用户组与用户，运行git服务(任意用户均可)

```
1 | $ groupadd git
2 | $ useradd git -g git
```

```
sun@JK-Linux:~$ groupadd git
groupadd: Permission denied.
groupadd: cannot lock /etc/group; try again later.
sun@JK-Linux:~$ sudo groupadd git
sun@JK-Linux:~$ sudo useradd git -g git
sun@JK-Linux:~$
```

3. 创建裸存储库

登录到指定用户

```
$ sudo su - git
```

创建仓库目录为gitrepo

初始化裸仓库

```
1 $ cd gitrepo
2 $ git init --bare runoob.git
```

以上命令Git创建一个空仓库，服务器上的 Git 仓库通常都以 .git 结尾。

然后，把仓库所属用户改为 git (如果是其他用户操作，比如 root)：

```
$ chown -R git:git runoob.git
```

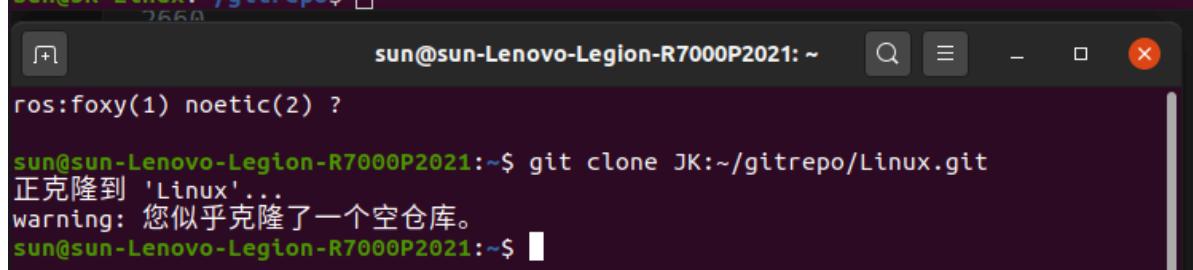
4. 创建证书登录

见SSH笔记

5. 测试仓库

git clone

```
sun@JK-Linux:~/gitrepo$ ls
Linux.git
sun@JK-Linux:~/gitrepo$ cd Linux.git/
sun@JK-Linux:~/gitrepo/Linux.git$ ls
HEAD branches config description hooks info objects refs
sun@JK-Linux:~/gitrepo/Linux.git$ cd ..
sun@JK-Linux:~/gitrepo$ ls
Linux.git
sun@JK-Linux:~/gitrepo$ 
```



构建远程仓库

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ git remote add Sun JK:~/gitrepo/Linux.git
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ git remote -v
Sun      JK:~/gitrepo/Linux.git (fetch)
Sun      JK:~/gitrepo/Linux.git (push)
```

可以进行push / pull /fetch merge操作

```
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$ git push -u Sun master
枚举对象: 3, 完成.
对对象计数中: 100% (3/3), 完成.
使用 12 个线程进行压缩
压缩对象中: 100% (2/2), 完成.
写入对象中: 100% (3/3), 406 字节 | 406.00 KiB/s, 完成.
总共 3 (差异 0) , 复用 0 (差异 0)
To JK:~/gitrepo/Linux.git
 * [new branch]      master -> master
 分支 'master' 设置为跟踪来自 'sun' 的远程分支 'master'.
sun@sun-Lenovo-Legion-R7000P2021:~/My_notes/Git/codes/newrepo$
```

注意：

裸仓库没有工作区，因此没有文件的实际内容。

而是通过 .git 文件夹的内部结构存储所有的版本数据。

各个文件存储内容，见.git目录结构

Gitlab服务

GitLab 是一个功能强大的 Git 服务管理工具，适合中大型团队，提供了丰富的用户管理、CI/CD、代码审查等功能。

安装Gitlab

```
1 sudo apt-get update
2 sudo apt-get install -y curl openssh-server ca-certificates tzdata perl
3 curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-
ee/script.deb.sh | sudo bash
4 sudo EXTERNAL_URL="http://yourdomain" apt-get install gitlab-ee
```

EXTERNAL_URL="http://yourdomain" 要设置自己的域名，或者公网 IP，比如：

```
sudo EXTERNAL_URL=101.132.xx.xx yum install -y gitlab-ee
```

解释：

```
sudo apt-get install -y curl openssh-server ca-certificates tzdata perl
sudo apt-get install -y curl openssh-server ca-certificates tzdata perl 安装多个包
-y
```

作用：自动回答 "yes" 以确认安装过程中的所有提示。

好处：避免交互式提示，适合脚本或批量操作。

包清单：

curl

作用：命令行工具，用于通过 HTTP、HTTPS、FTP 等协议传输数据。

常见用途：

下载文件。

测试 API 接口。

如：curl <https://example.com>。

openssh-server

作用：OpenSSH 的服务器端实现，允许远程用户通过 SSH 登录这台机器。

用途：

设置远程登录服务。

允许通过 ssh 访问和管理系统。

ca-certificates

作用：安装并维护系统中可信的证书颁发机构（CA）的证书。

用途：

验证 HTTPS 连接的安全性。

确保 curl 等工具能正确验证 SSL/TLS 证书。

tzdata

作用：时区数据库，用于设置和管理系统时区。

用途：

确保时间和日期在不同地区显示正确。

安装时可能会提示选择时区。

perl

作用：Perl 编程语言的解释器。

用途：

支持运行许多系统管理脚本。

许多系统工具（如 Git）依赖于 Perl 运行。

```
curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.deb.sh  
| sudo bash
```

下载官方安装脚本并直接输出bash执行

```
sudo EXTERNAL_URL="http://yourdomain" apt-get install gitlab-ee
```

配置外部访问的URL

示例：

```
sun@JK-Linux:~/gitrepo/Linux.git$ sudo apt install -y ca-certificates tzdata perl  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
ca-certificates is already the newest version (20240203~20.04.1).  
ca-certificates set to manually installed.  
perl is already the newest version (5.30.0-9ubuntu0.5).  
perl set to manually installed.  
The following packages were automatically installed and are no longer required:  
  linux-azure-5.15-cloud-tools-5.15.0-1060 linux-azure-5.15-cloud-tools-5.15.0-1061  
  linux-azure-5.15-cloud-tools-5.15.0-1063 linux-azure-5.15-cloud-tools-5.15.0-1064  
  linux-azure-5.15-cloud-tools-5.15.0-1067 linux-azure-5.15-cloud-tools-5.15.0-1068  
  linux-azure-5.15-cloud-tools-5.15.0-1070 linux-azure-5.15-cloud-tools-5.15.0-1071  
  linux-azure-5.15-cloud-tools-5.15.0-1072 linux-azure-5.15-headers-5.15.0-1060  
  linux-azure-5.15-headers-5.15.0-1061 linux-azure-5.15-headers-5.15.0-1063  
  linux-azure-5.15-headers-5.15.0-1064 linux-azure-5.15-headers-5.15.0-1067  
  linux-azure-5.15-headers-5.15.0-1068 linux-azure-5.15-headers-5.15.0-1070  
  linux-azure-5.15-headers-5.15.0-1071 linux-azure-5.15-headers-5.15.0-1072  
  linux-azure-5.15-tools-5.15.0-1060 linux-azure-5.15-tools-5.15.0-1061  
  linux-azure-5.15-tools-5.15.0-1063 linux-azure-5.15-tools-5.15.0-1064  
  linux-azure-5.15-tools-5.15.0-1067 linux-azure-5.15-tools-5.15.0-1068
```

```
sun@JK-Linux:~/gitrepo/Linux.git$ curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.deb.sh | sudo bash
% Total    % Received % Xferd  Average Speed   Time     Time      Time  Current
          Dload  Upload   Total   Spent    Left  Speed
100  6865  100  6865    0     0  25332      0 --:--:-- --:--:-- 25332
Detected operating system as Ubuntu/focal.
Checking for curl...
Detected curl...
Checking for gpg...
Detected gpg...
Running apt-get update... done.
Installing apt-transport-https... done.
Installing /etc/apt/sources.list.d/gitlab_gitlab-ee.list...done.
Importing packagecloud gpg key... done.
Running apt-get update... done.

The repository is setup! You can now install packages.
sun@JK-Linux:~/gitrepo/Linux.git$ sudo EXTERNAL_URL=74.226.155.16 apt-get install gitlab-ee
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-azure-5.15-cloud-tools-5.15.0-1060 linux-azure-5.15-cloud-tools-5.15.0-1061
  linux-azure-5.15-cloud-tools-5.15.0-1063 linux-azure-5.15-cloud-tools-5.15.0-1064
  linux-azure-5.15-cloud-tools-5.15.0-1067 linux-azure-5.15-cloud-tools-5.15.0-1068
  linux-azure-5.15-cloud-tools-5.15.0-1070 linux-azure-5.15-cloud-tools-5.15.0-1071
  linux-azure-5.15-cloud-tools-5.15.0-1072 linux-azure-5.15-headers-5.15.0-1060
  linux-azure-5.15-headers-5.15.0-1061 linux-azure-5.15-headers-5.15.0-1063
  linux-azure-5.15-headers-5.15.0-1064 linux-azure-5.15-headers-5.15.0-1067
  linux-azure-5.15-headers-5.15.0-1068 linux-azure-5.15-headers-5.15.0-1070
  linux-azure-5.15-headers-5.15.0-1071 linux-azure-5.15-headers-5.15.0-1072
  linux-azure-5.15-tools-5.15.0-1060 linux-azure-5.15-tools-5.15.0-1061
  linux-azure-5.15-tools-5.15.0-1063 linux-azure-5.15-tools-5.15.0-1064
  linux-azure-5.15-tools-5.15.0-1067 linux-azure-5.15-tools-5.15.0-1068
  linux-azure-5.15-tools-5.15.0-1070 linux-azure-5.15-tools-5.15.0-1071
  linux-azure-5.15-tools-5.15.0-1072 linux-cloud-tools-5.15.0-1060-azure
  linux-cloud-tools-5.15.0-1061-azure linux-cloud-tools-5.15.0-1063-azure
  linux-cloud-tools-5.15.0-1064-azure linux-cloud-tools-5.15.0-1067-azure
  linux-cloud-tools-5.15.0-1068-azure linux-cloud-tools-5.15.0-1070-azure
  linux-cloud-tools-5.15.0-1071-azure linux-cloud-tools-5.15.0-1072-azure

linux-tools-5.15.0-1072-azure
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed:
  gitlab-ee
0 upgraded, 1 newly installed, 0 to remove and 40 not upgraded.
Need to get 1430 MB of archives.
After this operation, 4046 MB of additional disk space will be used.
Get:1 https://packages.gitlab.com/gitlab/gitlab-ee/ubuntu focal/main amd64 gitlab-ee amd64 17.5.2-ee.0 [1430 MB]
36% [1 gitlab-ee 644 MB/1430 MB 45%] 58.1 MB/s 13s
```

本人服务器安装失败，后续不在演示：

```
[+] sun@JK-Linux: ~/gitrepo/Linux.git Q E - x
stacktrace.out
[2024-11-16T13:49:42+00:00] FATAL: -----
[2024-11-16T13:49:42+00:00] FATAL: PLEASE PROVIDE THE CONTENTS OF THE stacktrace.out FILE (above)
IF YOU FILE A BUG REPORT
[2024-11-16T13:49:42+00:00] FATAL: -----
[2024-11-16T13:49:42+00:00] FATAL: Chef::Exceptions::MultipleFailures: Multiple failures occurred:
* Mixlib::ShellOut::ShellCommandFailed occurred in Cinc Client run: rails_migration[gitlab-rails]
(gitlab::database_migrations line 51) had an error: Mixlib::ShellOut::ShellCommandFailed: bash_hid
e_env[migrate gitlab-rails database] (gitlab::database_migrations line 20) had an error: Mixlib::S
hellOut::ShellCommandFailed: Expected process to exit with [0], but received '137'
---- Begin output of "bash" ----
STDOUT:
STDERR:
---- End output of "bash" ----
Ran "bash" returned 137
* Mixlib::ShellOut::ShellCommandFailed occurred in delayed notification: execute[clear the gitlab-
rails cache] (gitlab::gitlab-rails line 566) had an error: Mixlib::ShellOut::ShellCommandFailed: E
xpected process to exit with [0], but received ''
---- Begin output of /opt/gitlab/bin/gitlab-rake cache:clear ----
STDOUT:
STDERR:
---- End output of /opt/gitlab/bin/gitlab-rake cache:clear ----
Ran /opt/gitlab/bin/gitlab-rake cache:clear returned

dpkg: error processing package gitlab-ee (--configure):
 installed gitlab-ee package post-installation script subprocess returned error exit status 1
Errors were encountered while processing:
 gitlab-ee
E: Sub-process /usr/bin/dpkg returned an error code (1)
sun@JK-Linux:~/gitrepo/Linux.git$
```

配置Gitlab

安装完成后，打开浏览器访问 <http://yourdomain>，设置管理员账户。

当出现类似如下回显信息，表示 GitLab 已经安装成功。

创建项目

登录 GitLab，创建一个新的项目，用户名为 root。

获取登录密码：

```
sudo cat /etc/gitlab/initial_root_password
```

首次登录使用root用户名

生成密钥对文件并获取公钥

不在赘述，详情请见SSH

创建项目

在 GitLab 的主页中，点击 Create a project 选项：

点击 Create blank project，设置 Project name 和 Project URL，然后点击 Create project：

添加SSH key

在当前 project 页面，点击 Add SSH key：

将公钥文件 id_rsa.pub 中的内容粘贴到 Key 所在的文本框中：

点击 Add key，SSH Key 添加完成后，如下图所示：

复制 Clone 链接，该链接在进行克隆操作时需要使用：

其余使用与裸仓库，Github仓库完全一致。

十五、Sourcetree 使用教程

SourceTree 是一个 Git 客户端管理工具，适用于 Windows 和 Mac 系统。

SourceTree 简化了开发者与代码仓库之间的 Git 操作方式，我们可以通过界面菜单很方便的处理 Git 操作，而不需要通过命令。

通过 SourceTree，我们可以管理所有的 Git 库，无论是远程还是本地的。SourceTree 支持 Bitbucket、GitHub 以及 Gitlab 等远程仓库。

Sourcetree安装

[官网安装](#)

Souretree远程仓库

Souretree可以连接Github账户，管理远程仓库

Sourcetree创建本地仓库

当然也可以创建本地仓库

不进行过多赘述！