



Git使用培训

质量管理总部 俞红娟



目录



一、Git是什么？



二、Git与SVN的区别



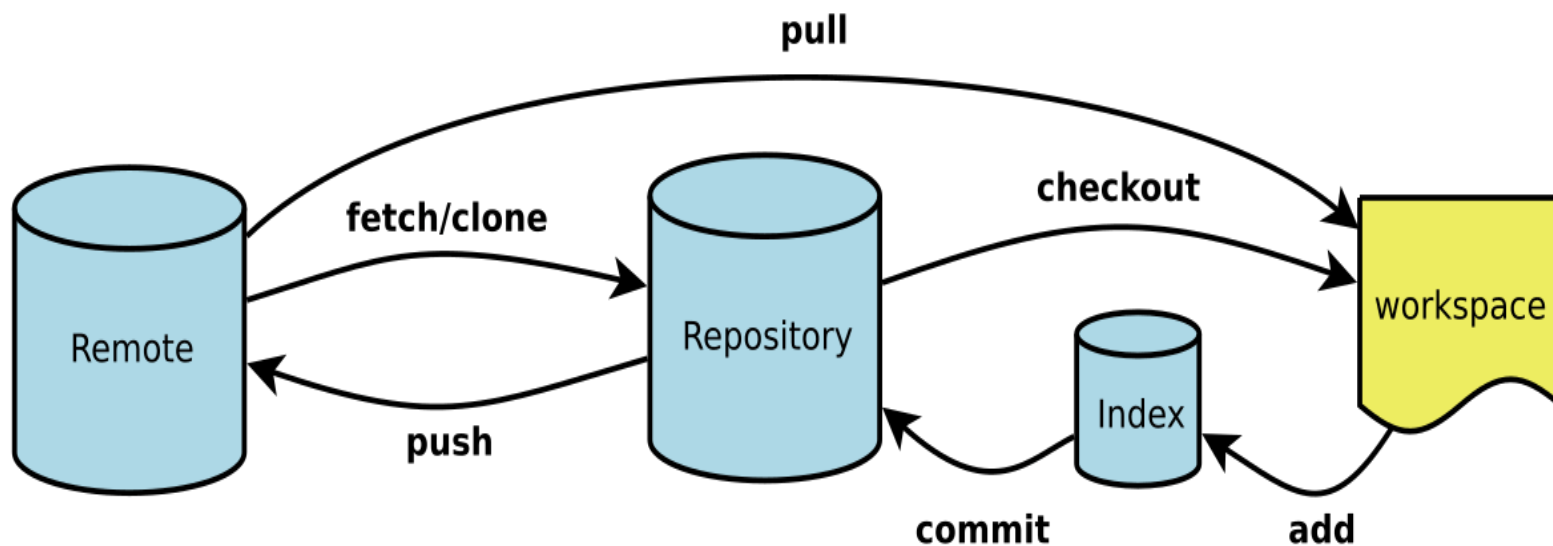
三、Gitlab可以做什么？



四、我们怎么用？

一、Git是什么？

1、Git是目前世界上最先进的分布式版本控制系统



Workspace: 工作区。就是你在电脑上看到的目录 (.git目录除外)

Index/Stage: 暂存区，也叫索引。

Repository: 仓库区（本地仓库），也称作存储库。工作区有一个隐藏目录.git,这个不属于工作区，这是版本库。其中版本库里面存了很多东西，其中最重要的就是stage(暂存区)，还有Git为我们自动创建了第一个分支master,以及指向master的一个指针HEAD。

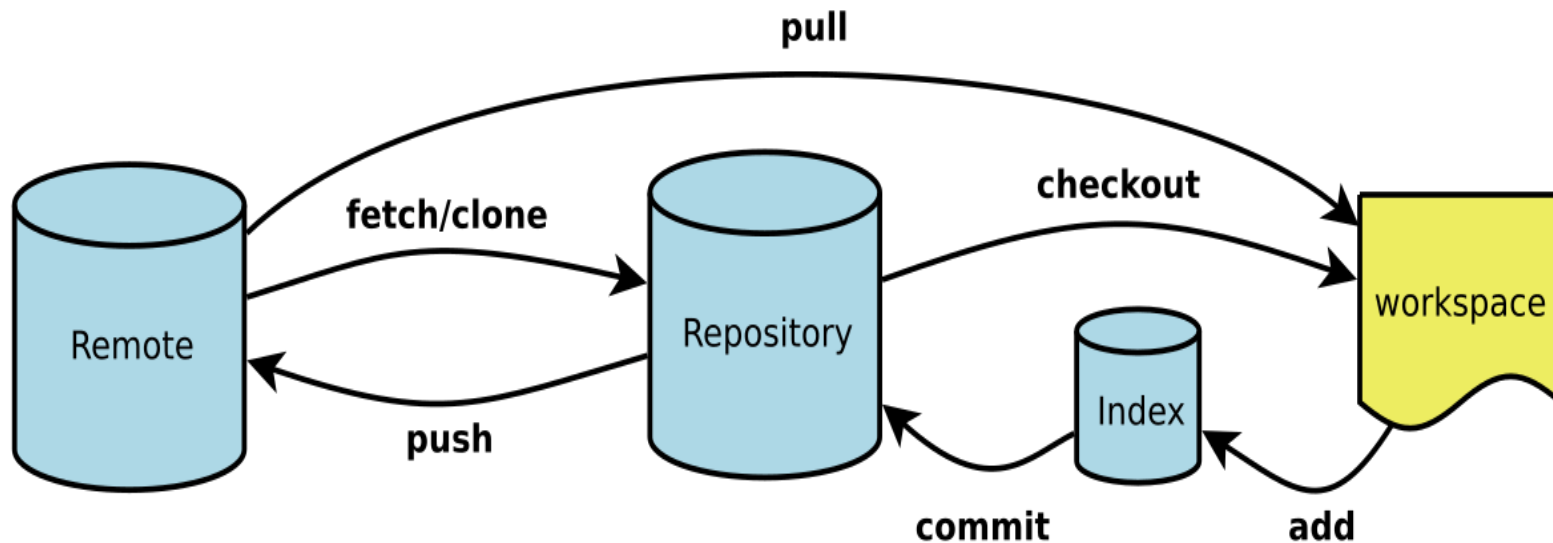
Remote: 远程仓库



一、Git是什么？

2、工作原理 / 流程

一般来说，日常使用只要记住下图6个命令，就可以了。



- (1) 对代码进行修改
- (2) 完成了某项功能，提交（commit，只是提交到本地代码库），1-2可以反复进行，直到觉得可以推送到服务器上时，执行3
- (3) 拉取（pull，或者用获取fetch然后再手动合并merge）
- (4) 如果存在冲突，解决冲突
- (5) 推送（push），将数据提交到服务器上的代码库



二、Git 与SVN的区别

对比项	Git	SVN	优势方
操作速度	常用简单命令，速度快，技术人员喜欢。	常用鼠标，非技术人员更喜欢。	
离线工作	可在本地查看所有版本历史，可在本地做所有操作，提交代码、查看历史、合并、创建分支等等。	不连网用户就看不到历史版本，也无法切换版本验证问题。	Git
分支功能	GIT的分支简单、有趣，可以从同一个工作目录下快速的在几个分支间切换，很容易发现未被合并的分支，能简单而快捷的合并这些文件。	对分支当做路径来授权，造成管理负担，需手工运行命令来确认代码是否被合并，会发生有些分支被遗漏的情况。	Git
合并的便捷性	基于对文件名追踪，遇到一方或双方对文件名更改时，SVN会产生树冲突，解决起来很麻烦。	基于对内容的追踪，基于DAG（有向非环图）的设计比SVN的线性提交提供更好的合并追踪，避免不必要的冲突，提高工作效率。	Git
隔离开发与提交审核	自建分支、本地版本库，审核新成员提交时从其个人版本库或个人分支获取（fetch）提交，审核通过执行GIT merge 命令合并到开发主线中。	集中式开发，开发人员不能随意建分支与配置库，他人做代码审核不便。	Git
学习周期	学习周期长，相对SVN更难。	简单易学	SVN
权限控制	GIT的授权模型只能实现非零即壹式的授权，要么拥有全部的写权限，要么没有写权限，要么拥有整个版本库的读权限，要么禁用。	可控制到任何一级目，不能在分支中集成，权限脚本工作量大，不能对单个文件授权。	SVN
项目类型	适用于开源或纯代码开发，不适合对二进制文件（Word文档、PPT演示稿）的管理。	SVN能清楚的按目录进行分类管理，使项目组的管理处于有序高效的状态。有阻止多人同时编辑一个文件的功能。	SVN



二、Git与SVN的区别

总结：

Git在操作的便携性上更胜SVN一筹，但不适合管理二进制文件；开源的理念，针对目录无详细权限控制（可每个模块一个库来控制模块的访问权限）。

SVN在管理工作上更胜Git，二进制文件、权限控制等都不惧怕。

三、Gitlab 可以做什么？

1、Gitlab 是 Git 服务端的集成管理平台

GitLab - 基于Git的项目管理软件。使用Git作为代码管理工具，并在此基础上搭建起来的web服务。



提供了：

- 1、代码托管服务
- 2、访问权限控制
- 3、问题跟踪，bug的记录、跟踪和讨论
- 4、Wiki，项目中一些相关的说明和文档
- 5、代码审查，可以查看、评论代码



三、Gitlab 可以做什么？

2、gitLab工作方式及流程

1、普通开发人员

第一步：pull项目

第二步：创建issue，创建后注意生成的issue编号，这个很重要。

第三步：切换分支干活

第四步：提交代码

第五步：等待项目管理员code review 然后合并到master。

第六步：若该issue已被关闭可以将本地的分支删掉。

2、master开发人员

第一步：创建project

第二步：创建milestone，评估工作量和时间

第三步：创建issue关联到milestone中

第四步：code review

第五步：若有冲突，需要解决。

第六步：合并分支

三、Gitlab 可以做什么？

3、gitLab权限举例



组角色

行为	Guest	Reporter	Developer	Master	Owner
浏览组	✓	✓	✓	✓	✓
编辑组					✓
创建项目				✓	✓
管理组成员					✓
移除组					✓

组名：部门缩写+代号

All Projects



组内项目

项目角色

- Guest: 访客，只能提交问题和评论内容。
- Reporter: 报告者，只有项目的读权限，可以创建代码片段，但是不能 push 到仓库的默认分支。
- Developer: 开发人员，能够推送和删除没有保护的分支，刚创建的分支默认都是没有保护的，对受保护内容无权限。
- Master: 项目管理人员，可以对没有保护和有保护的所有分支进行操作，几乎拥有所有权限，除更改、删除项目元信息外其它操作均可；可以创建项目、添加 tag、保护分支、添加项目成员、编辑项目。
- Owner: 系统管理员或项目所有者，拥有所有的操作权限。可以设置项目访问权限 - Visibility Level、删除项目、迁移项目、管理组成员。

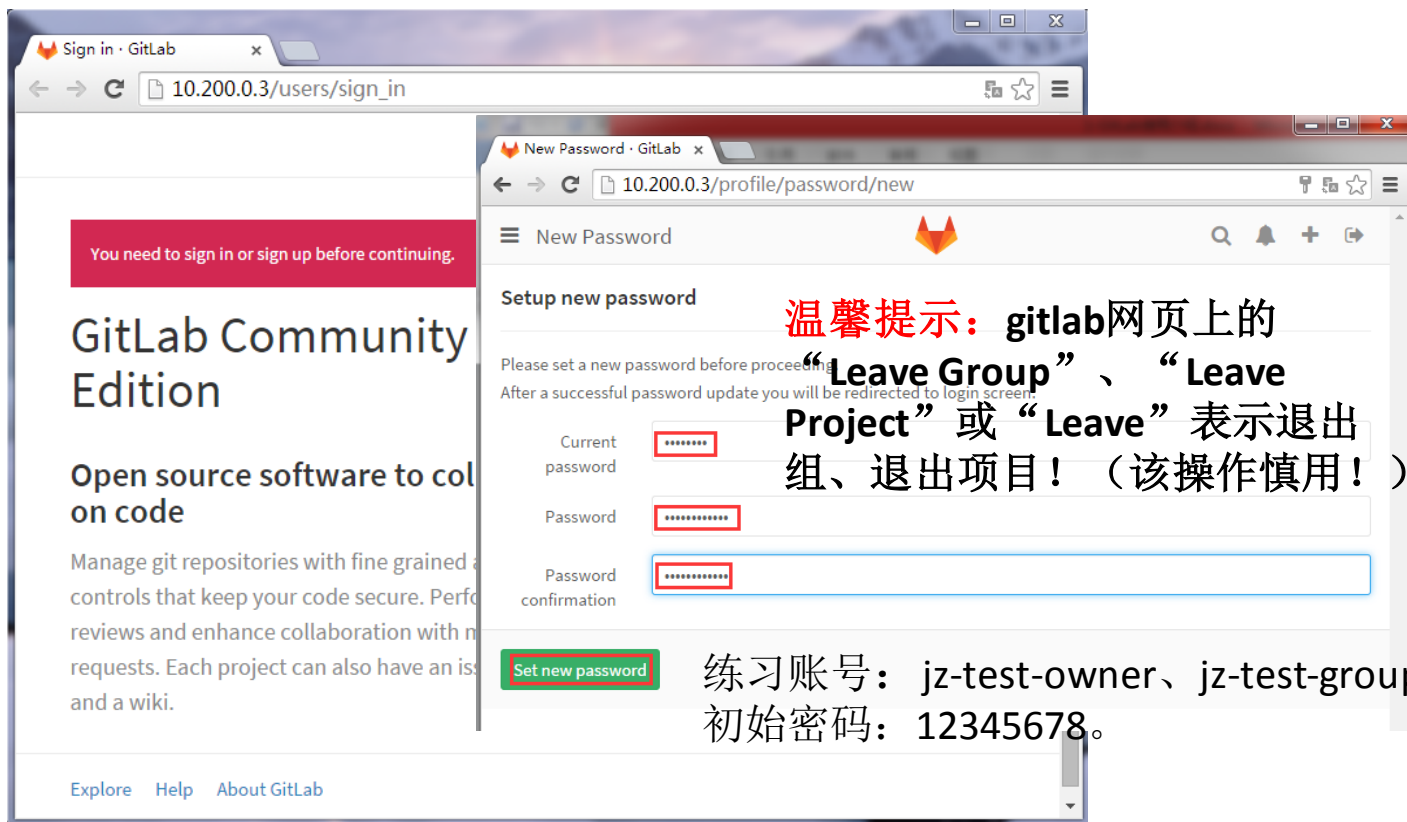
项目的默认权限继承组的权限，如果组的权限不合理，可以进一步更改。

温馨提示：目前仅系统管理员为项目的owner，组及组内项目由系统管理员创建。

四、我们怎么用？

1、收到gitLab账号密码后如何操作？--修改密码

用火狐或谷歌浏览器访问公司gitlab网址（http://10.200.0.3），输入gitlab账号密码，页面会跳转到新密码设置界面，输入初始密码、新密码，确认即可。



The image displays two overlapping browser windows from a GitLab instance at 10.200.0.3. The background window, titled 'Sign in · GitLab', shows the login page with a red banner stating 'You need to sign in or sign up before continuing.' and the 'GitLab Community Edition' logo. The foreground window, titled 'New Password · GitLab', shows the 'Setup new password' page. It includes fields for 'Current password', 'Password', and 'Password confirmation', each with a red box around it. A green 'Set new password' button is at the bottom. A red text overlay on the right side of the foreground window reads: '温馨提示：gitlab网页上的“Leave Group”、“Leave Project”或“Leave”表示退出组、退出项目！（该操作慎用！）'. Below this, black text provides practice credentials: '练习账号：jz-test-owner、jz-test-group、jz-test-project' and '初始密码：12345678.'.

温馨提示：gitlab网页上的“Leave Group”、“Leave Project”或“Leave”表示退出组、退出项目！（该操作慎用！）

练习账号：jz-test-owner、jz-test-group、jz-test-project
初始密码：12345678。

四、我们怎么用？

2、项目master如何操作？

2-1、设置项目组成员权限

在Gitlab页面里，点击Project，找到指定的Project，下拉菜单点击Members，可以添加人员，并给指定的人员设置权限。

Members

Groups

Deploy Keys

Webhooks

Services

Protected branches

Runners

Variables

Triggers

Badges

Edit Project

Leave Project

Add new user to project

Import members

Users with access to this project are listed below.

People

× fisheye_crucible

Search for users by name, username, or email, or invite new ones using their email address.

Project Access

Reporter

Read more about role permissions [here](#)

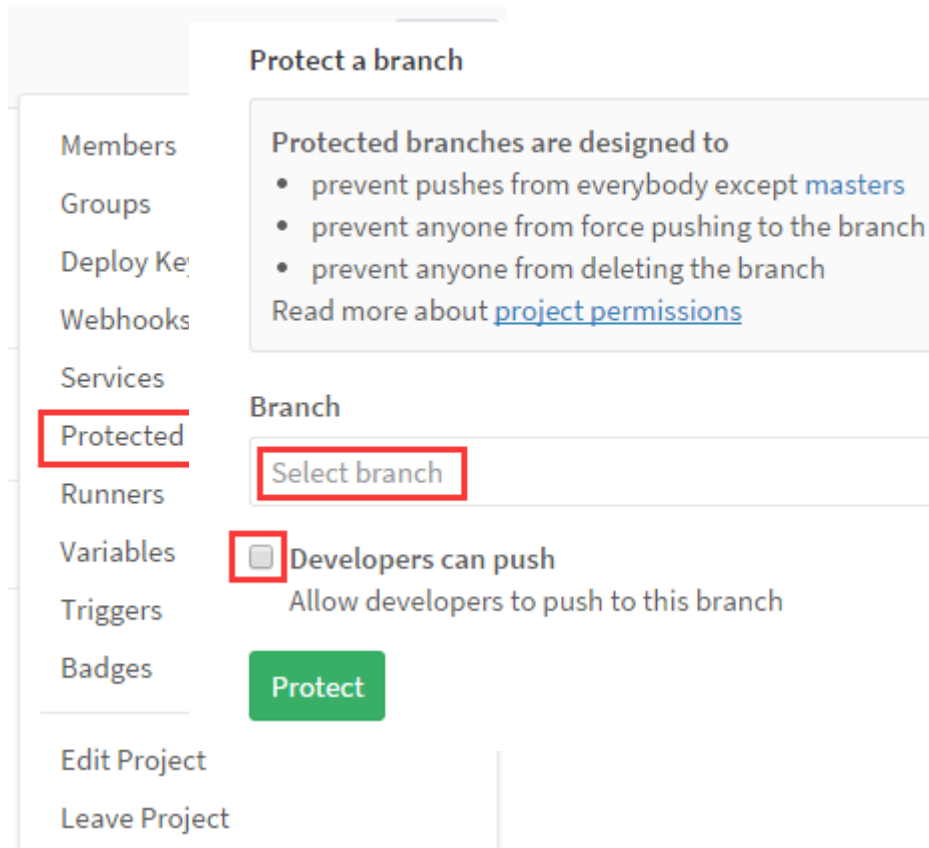
Add users to project

四、我们怎么用？

2、项目master如何操作？

2-2、设置分支保护

关于保护分支的设置，可以进入Settings->Protected branches进行管理。



该功能可用于：

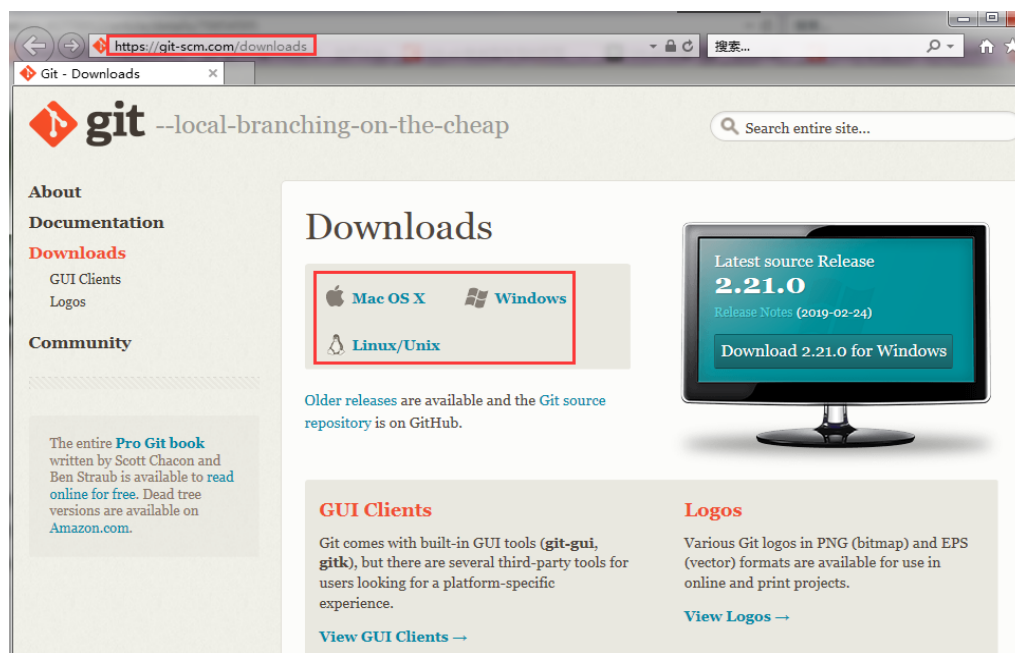
- 阻止 Master 角色以外的开发人员直接向此类分支推送代码，保持稳定分支的安全性；
 - 在向受保护分支合并代码前，强制进行代码审查。
- 例如：锁定我们的受保护分支——主分支 master 和预发布分支 release-*，以阻止 Developer 直接向这两类分支中推送代码。

四、我们怎么用？

3、Git 下载及安装

3-1命令操作工具

- Git（Git 主程序） <https://git-scm.com/downloads>



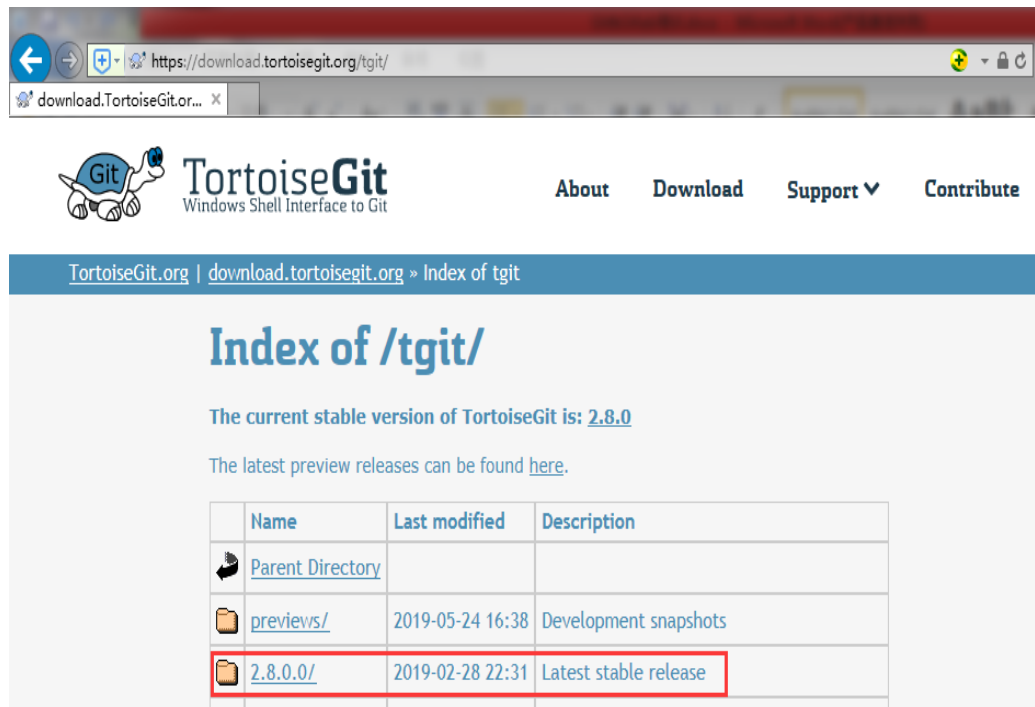
根据自身电脑的操作系统选择下载的git版本（点上图红圈那里）。默认安装，一路next就安装好了。安装完成后，本地目录鼠标右键，左键单击“Git Bash Here”，即可弹出Git命令框。

四、我们怎么用？

3、Git 下载及安装

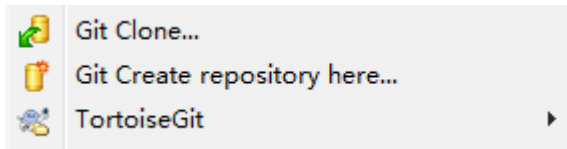
3-2Git 图形界面操作工具

- TortoiseGit <https://download.tortoisegit.org/tgit/>



The screenshot shows the TortoiseGit website. The browser address bar displays <https://download.tortoisegit.org/tgit/>. The page header includes the TortoiseGit logo and navigation links: About, Download, Support, and Contribute. The main content area is titled "Index of /tgit/" and states: "The current stable version of TortoiseGit is: [2.8.0](#)". It also mentions: "The latest preview releases can be found [here](#)." Below this is a table listing directory contents:

Name	Last modified	Description
Parent Directory		
previews/	2019-05-24 16:38	Development snapshots
2.8.0.0/	2019-02-28 22:31	Latest stable release



TortoiseGit只有 Windows 版本，有32位和64位版本，请根据自己的电脑选择相应的版本，同时下载对应版本的中文语言包。先安装主程序，再安装语言包。安装完语言包之后，右键→TortoiseGit→setting，把language项改为中文，确定即可。

四、我们怎么用？

4、初次运行 Git 前的配置

4-1、配置git

配置过程

```
[root@gitlab ~]# git config --global user.name "张三" #配置git使用用户
[root@gitlab ~]# git config --global user.email "zhangs@szkingdom.com"
" #配置git使用邮箱
```

```
[root@gitlab ~]# git config --global color.ui true #语法高亮
```

```
[root@gitlab ~]# git config --list # 查看全局配置
```

```
user.name=张三
```

```
user.email=zhangs@szkingdom.com
```

```
color.ui=true
```

生成的配置文件

```
[root@gitlab ~]# cat .gitconfig # win7 git bash 下执行 cat ~/.gitconfig
```

```
[user]
```

```
name = 张三
```

```
email = zhangs@szkingdom.com
```

```
[color]
```

```
ui = true
```

注意：git config --global 参数，有了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然你也可以对某个仓库指定的不同的用户名和邮箱。

清除保存好的账号密码：删除.git-credentials
(一般在c盘-->用户下面有，删除即可)

四、我们怎么用？

4、初次运行 Git 前的配置

4-2、获取帮助

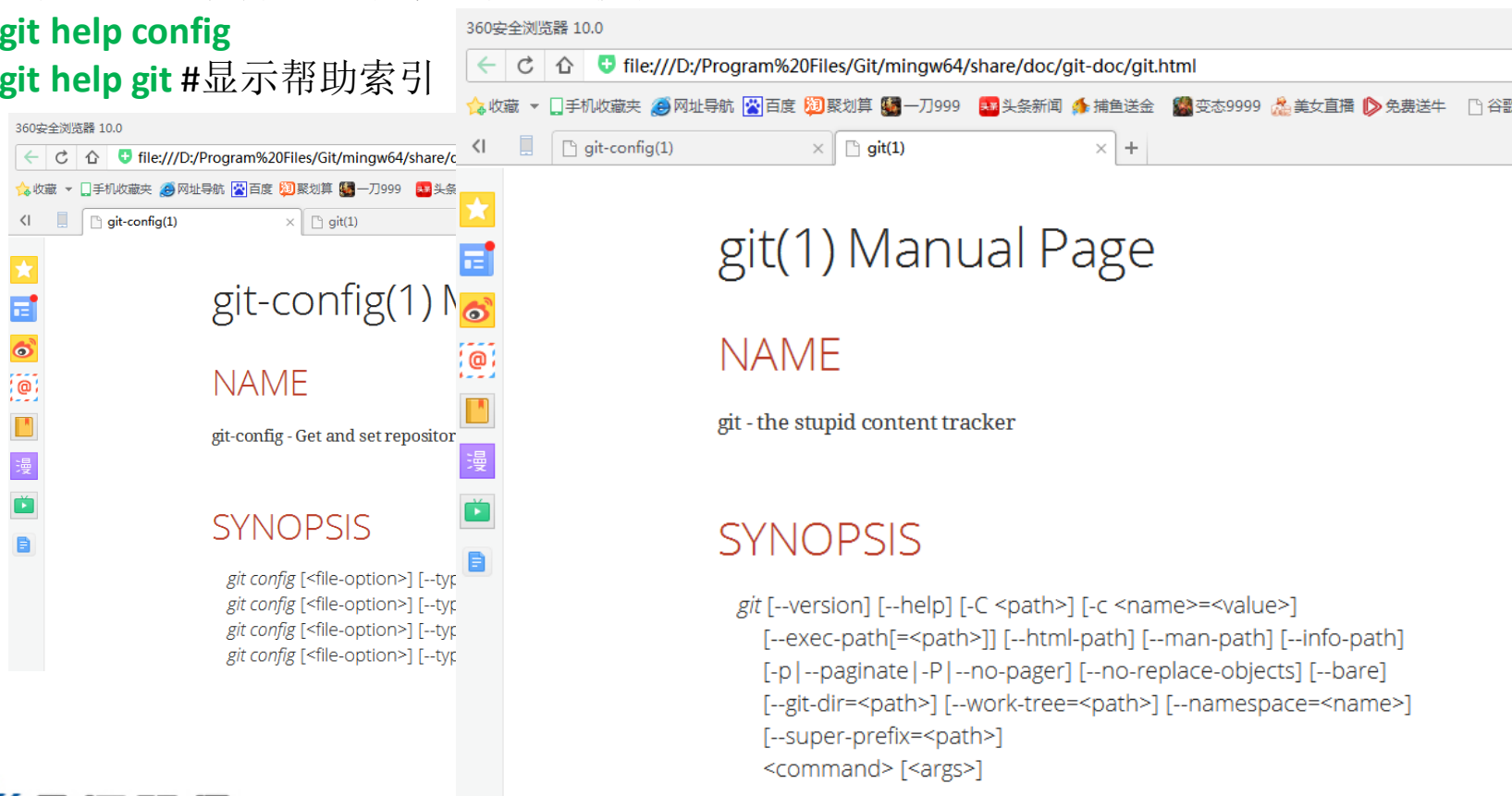
使用Git时需要获取帮助，使用如下方法可以找到Git命令的使用手册：

`git help <verb>`

例如，要想获得配置命令的手册，执行

`git help config`

`git help git` #显示帮助索引



The screenshot shows a 360 browser window with two tabs. The left tab is titled 'git-config(1)' and displays the manual page for the 'git-config' command. The right tab is titled 'git(1)' and displays the manual page for the 'git' command. The 'git-config(1)' page shows the title 'git-config(1) NAME', the description 'git-config - Get and set repository configuration', and the synopsis 'git config [<file-option>] [--type=<type>] [<key>=<value>]'. The 'git(1)' page shows the title 'git(1) Manual Page', the description 'git - the stupid content tracker', and the synopsis 'git [--version] [--help] [-C <path>] [-c <name>=<value>] [--exec-path=<path>] [--html-path] [--man-path] [--info-path] [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare] [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>] [--super-prefix=<path>] [<command>] [<args>]'. The browser's address bar shows the file path 'file:///D:/Program%20Files/Git/mingw64/share/doc/git-doc/git.html'.

四、我们怎么用？

5、获取 Git 仓库（初始化仓库）

操作过程

```
[root@gitlab ~]# mkdir git_data      # 创建目录
[root@gitlab ~]# cd git_data/        # 进入目录
[root@gitlab git_data]# git init      # 初始化（初始化后多了一个隐藏的.git目录）
初始化空的 Git 版本库于 /root/git_data/.git/
[root@gitlab git_data]# git status    # 查看工作区状态
# 位于分支 master
#
# 初始提交
#
无文件要提交（创建/拷贝文件并使用 "git add" 建立跟踪）
```

补充：

有两种取得 Git 项目仓库的方法。**第一种**是从一个服务器**克隆**一个现有的 Git 仓库。**第二种**是在现有项目或目录下**导入**所有文件到 Git 中。

克隆仓库的命令格式是 `git clone [url]`。

例如：\$ `git clone http://git.oschina.net/yiibai/git-start.git`

如果想在克隆远程仓库的时候，自定义本地仓库的名字，可以使用如下命令：

\$ `git clone http://git.oschina.net/yiibai/git-start.git mygit-start`

Git 支持 HTTP、SSH 数据传输协议，如果要使用 SSH 的方式连接，需要确保自己的 IP 有访问 Gitlab 服务器 22 端口的权限。**--目前公司使用 http（开放 80 端口）**

四、我们怎么用？

6、Git命令常规操作

6-1创建文件

```
[root@gitlab git_data]# touch README
```

```
[root@gitlab git_data]# git status
```

```
# 位于分支 master
```

```
#
```

```
# 初始提交
```

```
#
```

```
# 未跟踪的文件:
```

```
# （使用 "git add <file>..." 以包含要提交的内容）
```

```
#
```

```
# README
```

```
提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
```

四、我们怎么用？

6、Git命令常规操作

添加文件跟踪

```
[root@gitlab git_data]# git add ./*
```

```
[root@gitlab git_data]# git status
```

```
# 位于分支 master
```

```
#
```

```
# 初始提交
```

```
#
```

```
# 要提交的变更:
```

```
# （使用 "git rm --cached <file>..." 撤出暂存区）
```

```
#
```

```
# 新文件:  README
```

```
#
```

文件会添加到.git的隐藏目录

```
[root@gitlab git_data]# tree .git/
```

```
.git/
```

```
├── objects├── e6└── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

四、我们怎么用？

6、Git命令常规操作

由工作区提交到本地仓库

```
[root@gitlab git_data]# git commit -m 'first commit'
```

```
[master (根提交) bb963eb] first commit
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 README
```

查看git的状态

```
[root@gitlab git_data]# git status
```

```
# 位于分支 master
```

无文件要提交，干净的工作区

提交后的git目录状态

```
[root@gitlab git_data]# tree .git/
```

```
.git/
```

```
├── objects├── 54├── 3b9bebd6c6bd5c4b22136034a95dd097a57d3dd├── bb├── 963eb32ad93a72d9ce93e4bb55105087f1227d├── e6├── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

四、我们怎么用？

6、Git命令常规操作

6-2添加新文件

修改或删除原有文件

`git add *` 添加到暂存区域

`git commit` 提交git仓库 -m 后面接上注释信息，内容关于本次提交的说明，方便自己或他人查看

简便方法 `git commit -a -m "注释信息"`

-a 表示直接提交 #Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

6-3删除git内的文件

命令说明：

- 没有添加到暂存区的数据直接rm删除即可。
- 已经添加到暂存区数据：

`git rm --cached database` #→将文件从git暂存区域的追踪列表移除(并不会删除当前工作目录内的数据文件)

`git rm -f database` #→将文件数据从git暂存区和工作目录一起删除

四、我们怎么用？

6、Git命令常规操作

命令实践：

```
[root@gitlab git_data]# touch 123 # 创建新文件
```

```
[root@gitlab git_data]# git status
```

```
# 位于分支 master
```

```
# 未跟踪的文件:
```

```
# （使用 "git add <file>..." 以包含要提交的内容）
```

```
#
```

```
# 123
```

```
提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
```

```
[root@gitlab git_data]# git add 123 # 将文件添加到暂存区域
```

```
[root@gitlab git_data]# git status
```

```
# 位于分支 master
```

```
# 要提交的变更:
```

```
# （使用 "git reset HEAD <file>..." 撤出暂存区）
```

```
#
```

```
# 新文件: 123
```

四、我们怎么用？

6、Git命令常规操作

命令实践：

```
[root@gitlab git_data]# rm 123 -f # 删除文件
[root@gitlab git_data]# ls
[root@gitlab git_data]# git status
# 位于分支 master
# 要提交的变更：
# （使用 "git reset HEAD <file>..." 撤出暂存区）
#
# 新文件： 123
#
# 尚未暂存以备提交的变更：
# （使用 "git add/rm <file>..." 更新要提交的内容）
# （使用 "git checkout -- <file>..." 丢弃工作区的改动）
#
# 删除： 123
#
[root@gitlab git_data]# git reset HEAD ./*
[root@gitlab git_data]# git status
# 位于分支 master
无文件要提交，干净的工作区
```

四、我们怎么用？

6、Git命令常规操作

6-4重命名暂存区数据

- 没有添加到暂存区的数据直接mv/rename改名即可。
- 已经添加到暂存区数据： `git mv README NOTICE`

6-5查看历史记录

- `git log` #→查看提交历史记录(列出部分日志，回车继续显示，输入q则退出！)
- `git log -2` #→查看最近几条记录
- `git log -p -1` #→-p显示每次提交的内容差异,例如仅查看最近一次差异
- `git log --stat -2` #→--stat简要显示数据增改行数，这样能够看到提交中修改过的内容，对文件添加或移动的行数，并在最后列出所有增减行的概要信息
- `git log --pretty=oneline` #→--pretty根据不同的格式展示提交的历史信息
- `git log --pretty=fuller -2` #→以更详细的模式输出提交的历史记录
- `git log --pretty=format:"%h %cn"` #→查看当前所有提交记录的简短SHA-1哈希字符串与提交者的姓名。

四、我们怎么用？

6、Git命令常规操作

6-6还原历史数据

Git服务程序中有一个叫做HEAD的版本指针，当用户申请还原数据时，其实就是将HEAD指针指向到某个特定的提交版本，但是因为Git是分布式版本控制系统，为了避免历史记录冲突，故使用了SHA-1计算出十六进制的哈希字符串来区分每个提交版本，另外默认的HEAD版本指针会指向到最近的一次提交版本记录，而上一个提交版本会叫HEAD^，上上一个版本则会叫做HEAD^^，当然一般会用HEAD~5来表示往上数第五个提交版本。

`git reset --hard hash`

`git reset --hard HEAD^` #→还原历史提交版本上一次

`git reset --hard 3de15d4` #→找到历史还原点的SHA-1值后，就可以还原(值不写全，系统会自动匹配)

6-7还原未来数据

什么是未来数据？就是你还原到历史数据了，但是你后悔了，想撤销更改，但是git log已经找不到这个版本了。

`git reflog` #→查看未来历史更新点

四、我们怎么用？

6、Git命令常规操作

6-8标签使用

前面回滚使用的是一串字符串，又长又难记。

`git tag v1.0` #→当前提交内容打一个标签(方便快速回滚)，每次提交都可以打个tag。

`git tag` #→查看当前所有的标签

`git show v1.0` #→查看当前1.0版本的详细信息

`git tag v1.2 -m "version 1.2 release is test"` #→创建带有说明的标签,-a指定标签名字，-m指定说明文字

`git tag -d v1.0` #→我们为同一个提交版本设置了两次标签,删除之前的v1.0

6-9对比数据

`git diff`可以对比当前文件与仓库已保存文件的区别，知道了对README作了什么修改

后，再把它提交到仓库就放心多了。

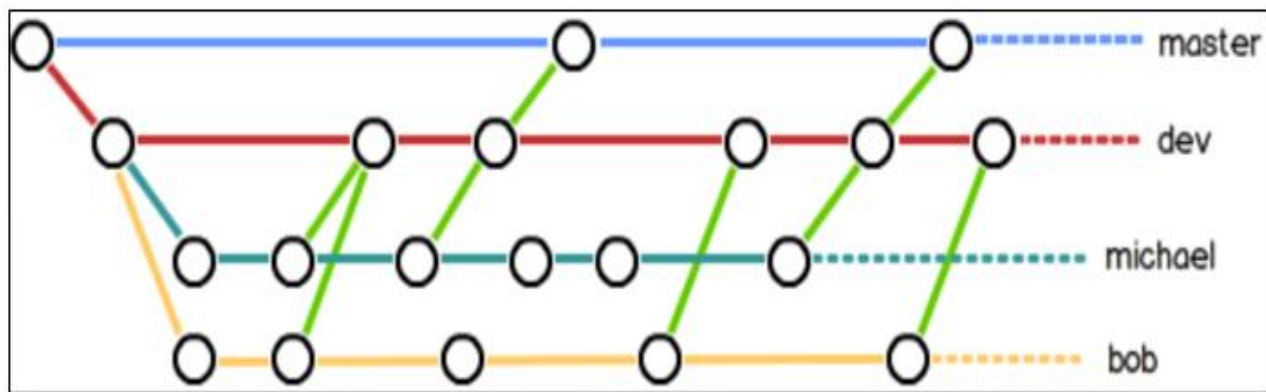
`git diff README`

四、我们怎么用？

7、分支结构

在实际的项目开发中，尽量保证master分支稳定，仅用于发布新版本，平时不要随便直接修改里面的数据文件。

那在哪干活呢？干活都在dev分支上。每个人从dev分支创建自己个人分支，开发完合并到dev分支，最后dev分支合并到master分支。所以团队的合作分支看起来会像下图那样。



四、我们怎么用？

7、分支结构

7-1、分支切换

```
[root@gitlab git_data]# git branch linux
[root@gitlab git_data]# git branch
linux
* master
[root@gitlab git_data]# git checkout linux
切换到分支 'linux'
[root@gitlab git_data]# git branch
* linux
Master
```

可以使用git checkout -b develop实现创建+切换的功能
在linux分支进行修改

```
[root@gitlab git_data]# cat README
[root@gitlab git_data]# echo "2017年11月30日" >> README
[root@gitlab git_data]# git add .
[root@gitlab git_data]# git commit -m "2017年11月30日09点10分"
[linux 5a6c037] 2017年11月30日09点10分
1 file changed, 1 insertion(+)
[root@gitlab git_data]# git status
# 位于分支 linux
无文件要提交，干净的工作区
```

四、我们怎么用？

7、分支结构

回到master分支

```
[root@gitlab git_data]# git checkout master
切换到分支 'master'
[root@gitlab git_data]# cat README
[root@gitlab git_data]# git log -1
commit 7015bc7b316cc95e2dfe6c53e06e3900b2edf427
Author: clsn <admin@znix.top>
Date: Wed Nov 29 19:30:57 2017 +0800
```

123

合并代码

```
[root@gitlab git_data]# git merge linux
更新 7015bc7..5a6c037
Fast-forward
 README | 1 +
 1 file changed, 1 insertion(+)
[root@gitlab git_data]# git status
# 位于分支 master
无文件要提交，干净的工作区
[root@gitlab git_data]# cat README
2017年11月30日
```

四、我们怎么用？

7、分支结构

7-2、合并失败解决

模拟冲突，在文件的同一行做不同修改

在**master**分支进行修改

```
[root@gitlab git_data]# cat README
```

2017年11月30日

```
[root@gitlab git_data]# echo "clsn in master">> README
```

```
[root@gitlab git_data]# git commit -a -m "clsn 2017年11月30日 09点20分 "
```

```
[master 7ab71d4] clsn 2017年11月30日 09点20分
```

1 file changed, 1 insertion(+)

切换到**linux**分支

```
[root@gitlab git_data]# git checkout linux
```

切换到分支 'linux'

```
[root@gitlab git_data]# cat README
```

2017年11月30日

```
[root@gitlab git_data]# echo "clsn in linux" >> README
```

```
[root@gitlab git_data]# git commit -a -m "2017年11月30日 03"
```

```
[linux 20f1a13] 2017年11月30日 03
```

1 file changed, 1 insertion(+)

四、我们怎么用？

7、分支结构

回到master分区，进行合并，出现冲突

```
[root@gitlab git_data]# git checkout master
```

切换到分支 'master'

```
[root@gitlab git_data]# git merge linux
```

自动合并 README

冲突（内容）：合并冲突于 README

自动合并失败，修正冲突然后提交修正的结果。

解决冲突

```
[root@gitlab git_data]# vim README
```

2017年11月30日

clsn in master

clsn in linux

手工解决冲突

```
[root@gitlab git_data]# git commit -a -m "2017年11月30日 03"
```

```
[master b6a097f] 2017年11月30日 03
```

四、我们怎么用？

7、分支结构

7-3、删除分支

因为之前已经合并了linux分支，所以现在看到它在列表中。在这个列表中分支名字前没有 * 号的分支通常可以使用 `git branch -d` 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。

查看所有包含未合并工作的分支，可以运行 `git branch --no-merged:`

```
git branch --no-merged
```

```
testing
```

这里显示了其他分支。因为它包含了还未合并的工作，尝试使用 `git branch -d` 命令删除它时会失败：

```
git branch -d testing
```

```
error: The branch 'testing' is not fully merged.
```

If you are sure you want to delete it, run 'git branch -D testing'.

如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 **-D** 选项强制删除它。



四、我们怎么用？

8、常用命令

Create a new repository

```
git clone git@10.200.0.3:JZ-TEST/client.git  
cd client  
touch README.md  
git add README.md  
git commit -m "add README"  
git push -u origin master
```

Existing folder or Git repository

```
cd existing_folder  
git init  
git remote add origin git@10.200.0.3:JZ-TEST/client.git  
git add .  
git commit  
git push -u origin master
```

四、我们怎么用？

9、命令列表参考

配置	Git的设置文件为 .gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。	
	\$ git config --list	# 显示当前的Git配置
	\$ git config -e [--global]	# 编辑Git配置文件
	\$ git config [--global] user.name "[name]" \$ git config [--global] user.email "[email address]"	# 设置提交代码时的用户信息
新建版本仓库	\$ git init	# 在当前目录新建一个Git代码库（初始化后：多了一个隐藏的.git目录） 【常用】
	\$ git init [project-name]	# 新建一个目录，将其初始化为Git代码库
	\$ git clone [-o faker] [url]	# 下载一个项目和它的整个代码历史，-o 给远程仓库起名:faker,默认origin 【常用】
增加/删除文件	\$ git add [file1] [file2] ...	# 添加指定文件到暂存区
	\$ git add [dir]	# 添加指定目录到暂存区，包括子目录
	\$ git add .	# 添加当前目录的所有文件到暂存区 【常用】
	\$ git add -p	# 添加每个变化前，都会要求确认 #对于同一个文件的多处变化，可以实现分次提交
	\$ git rm [file1] [file2] ...	# 删除工作区文件，并且将这次删除放入暂存区
	\$ git rm --cached [file]	# 停止追踪指定文件，但该文件会保留在工作区
代码提交	\$ git mv [file-original] [file-renamed]	# 改名文件，并且将这个改名放入暂存区
	\$ git commit -m [message]	# 提交暂存区到仓库区 【常用】
	\$ git commit [file1] [file2] ... -m [message]	# 提交暂存区的指定文件到仓库区
	\$ git commit -a	# 提交工作区自上次commit之后的变化，直接到仓库区
	\$ git commit -v	# 提交时显示所有diff信息
	\$ git commit --amend -m [message]	# 使用一次新的commit，替代上一次提交 #如果代码没有任何新变化，则用来改写上一次commit的提交信息
	\$ git commit --amend [file1] [file2] ...	# 重做上一次commit，并包括指定文件的新变化

四、我们怎么用？

9、命令列表参考

分支	\$ git branch	# 列出所有本地分支（当前分支将以*突出显示）
	\$ git branch -r	# 列出所有远程分支
	\$ git branch -a	# 列出所有本地分支和远程分支
	\$ git branch -v	# 列出所有本地分支，并展示没有分支最后一次提交的信息
	\$ git branch -vv	# 列出所有本地分支，并展示没有分支最后一次提交的信息和远程分支的追踪情况
	\$ git branch [branch-name]	# 新建一个分支，并切换到该分支
	\$ git checkout -b [branch]	# 新建一个分支，并切换到该分支
	\$ git checkout -b [branch] [origin branch]	# 基于某个分支创建一个新的分支，并且切换到该分支【常用】
	\$ git checkout --track [branch-name]	# 新建一个与远程分支同名的分支，并切换到该分支
	\$ git branch [branch] [commit]	# 新建一个分支，指向指定commit
	\$ git branch --track [branch] [remote-branch]	# 新建一个分支，与指定的远程分支建立追踪关系
	\$ git checkout [branch-name]	# 切换到指定分支，并更新工作区
	\$ git checkout -	# 切换到上一个分支
	\$ git branch --set-upstream [branch] [remote-branch]	# 建立追踪关系，在现有分支与指定的远程分支之间
	\$ git merge [branch]	# 合并指定分支到当前分支
	\$ git merge --no-ff [branch]	# 合并指定分支到当前分支，不要fast forward合并，首先注意切换分支【常用】
	\$ git cherry-pick [commit]	# 选择一个commit，合并进当前分支
	\$ git branch -d [branch-name]	# 删除分支
	\$ git push origin [branch-name]	# 新增远程分支 远程分支需先在本地创建,再进行推送
	\$ git push origin --delete [branch-name]	# 删除远程分支
	\$ git branch -dr [remote/branch]	

四、我们怎么用？

9、命令列表参考

标签	\$ git tag	# 列出所有tag【常用】
	\$ git tag [tag]	# 新建一个tag在当前commit【常用】
	\$ git tag [tag] [commit]	# 新建一个tag在指定commit
	\$ git tag -d [tag]	# 删除本地tag
	\$ git push origin :refs/tags/[tagName]	# 删除远程tag
	\$ git show [tag]	# 查看tag信息
	\$ git push [remote] [tag]	# 提交指定tag
	\$ git push [remote] --tags	# 提交所有tag【常用】
	\$ git checkout -b [branch] [tag]	# 新建一个分支，指向某个tag
查看信息	\$ git status	# 显示有变更的文件
	\$ git log	# 显示当前分支的版本历史
	\$ git log --stat	# 显示commit历史，以及每次commit发生变更的文件
	\$ git log -S [keyword]	# 搜索提交历史，根据关键词
	\$ git log [tag] HEAD --pretty=format:%s	# 显示某个commit之后的所有变动，每个commit占据一行
	\$ git log [tag] HEAD --grep feature	# 显示某个commit之后的所有变动，其"提交说明"必须符合搜索条件
	\$ git log --follow [file]	# 显示某个文件的版本历史，包括文件改名
	\$ git whatchanged [file]	
	\$ git log -p [file]	# 显示指定文件相关的每一次diff
	\$ git log -5 --pretty --oneline	# 显示过去5次提交
	\$ git shortlog -sn	# 显示所有提交过的用户，按提交次数排序
	\$ git blame [file]	# 显示指定文件是什么人在什么时间修改过
	\$ git diff	# 显示暂存区和工作区的差异
	\$ git diff --cached [file]	# 显示暂存区和上一个commit的差异
	\$ git diff HEAD	# 显示工作区与当前分支最新commit之间的差异
	\$ git diff [first-branch]...[second-branch]	# 显示两次提交之间的差异
	\$ git diff --shortstat "@{0 day ago}"	# 显示今天你写了多少行代码
	\$ git show [commit]	# 显示某次提交的元数据和内容变化
	\$ git show --name-only [commit]	# 显示某次提交发生变化的文件
	\$ git show [commit]:[filename]	# 显示某次提交时，某个文件的内容
	\$ git reflog	# 显示当前分支的最近几次提交

四、我们怎么用？

9、命令列表参考

远程同步	\$ git fetch [shortname]	# 下载远程仓库的所有变动 [shortname] 为远程仓库的shortname, 如origin, 为空时:默认origin
	\$ git remote -v	# 显示所有远程仓库
	\$ git remote add origin [url]	# 推送前要先添加一个远程仓库地址
	\$ git ls-remote [shortname]	# 显式地获得远程引用的完整列表 [shortname] 为远程仓库的shortname, 如origin, 为空时:默认origin
	\$ git remote show [shortname]	# 显示某个远程仓库的信息 [remote] 为远程仓库的shortname, 如origin
	\$ git remote add [shortname] [url]	# 增加一个新的远程仓库, 并命名
	\$ git remote rm [shortname] [url]	# 删除一个远程链接
	\$ git remote set-url origin [shortname] [url]	# 修改远程仓库地址
	\$ git pull [remote] [branch]	# 取回远程仓库的变化, 并与本地分支合并
	\$ git push [remote] [branch]	# 上传本地指定分支到远程仓库【常用】
撤销	\$ git push [remote] --force	# 强行推送当前分支到远程仓库, 即使有冲突--慎用
	\$ git push [remote] --all	# 推送所有分支到远程仓库
	\$ git checkout [file]	# 恢复暂存区的指定文件到工作区
	\$ git checkout [commit] [file]	# 恢复某个commit的指定文件到暂存区和工作区
	\$ git checkout .	# 恢复暂存区的所有文件到工作区
	\$ git reset [file]	# 重置暂存区的指定文件, 与上一次commit保持一致, 但工作区不变
	\$ git reset --hard	# 重置暂存区与工作区, 与上一次commit保持一致
	\$ git reset [commit]	# 重置当前分支的指针为指定commit, 同时重置暂存区, 但工作区不变
	\$ git reset --hard [commit]	# 重置当前分支的HEAD为指定commit, 同时重置暂存区和工作区, 与指定commit一致
	\$ git reset --keep [commit]	# 重置当前HEAD为指定commit, 但保持暂存区和工作区不变
其他	\$ git revert [commit]	# 新建一个commit, 用来撤销指定commit # 后者的所有变化都将被前者抵消, 并且应用到当前分支
	\$ git stash	# 暂时将未提交的变化移除, 稍后再移入
	\$ git stash pop	
其他	\$ git archive	# 生成一个可供发布的压缩包



结束语

谢谢观看

金证
学院