

# Git使用教程

## 第一章 Git安装

windows安装：进入网站<https://git-scm.com/>下载安装，然后在cmd命令行配置

直接去腾讯软件中心下载也可以！

```
1 > git config --global user.name "itnanls"
2 > git config --global user.email "itnanls@163.com"
3 #检查信息是否写入成功
4 git config --list
```

ubuntu配置：apt-get install git

centos配置：yum install git

## 第二章 理论基础

### (1) Git 是什么？

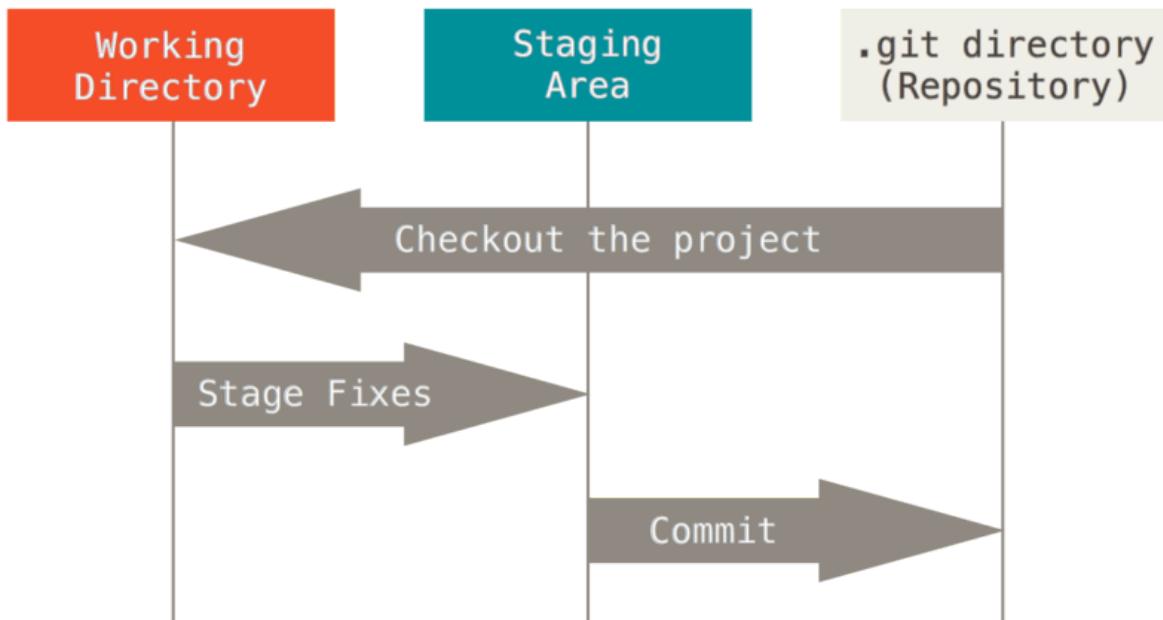
那么，简单地说，Git 究竟是怎样的一个系统呢？请注意接下来的内容非常重要，若你理解了 Git 的思想和基本工作原理，用起来就会知其所以然，游刃有余。在学习 Git 时，请尽量理清你对其它版本管理系统已有的认识，如 CVS、Subversion 或 Perforce，这样能帮助你使用工具时避免发生混淆。尽管 Git 用起来与其它的版本控制系统非常相似，但它在对信息的存储和认知方式上却有很大差异，理解这些差异将有助于避免使用中的困惑。

### (2) 三种状态

现在请注意，如果你希望后面的学习更顺利，请记住下面这些关于 Git 的概念。Git 有三种状态，你的文件可能处于其中之一：**已提交 (committed)**、**已修改 (modified)** 和 **已暂存 (staged)**。

- 已修改表示修改了文件，但还没保存到数据库中。
- 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。
- 已提交表示数据已经安全地保存在本地数据库中。

这会让我们的 Git 项目拥有三个阶段：工作区、暂存区以及 Git 目录。



工作目录、暂存区域以及 Git 仓库。

工作区是对项目的某个版本独立提取出来的内容。这些从 Git 仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

暂存区是一个文件，保存了下次将要提交的文件列表信息，一般在 Git 仓库目录中。按照 Git 的术语叫做“索引”，不过一般说法还是叫“暂存区”。

Git 仓库目录是 Git 用来保存项目的元数据和对象数据库的地方。这是 Git 中最重要的部分，从其它计算机克隆仓库时，复制的就是这里的数据。

基本的 Git 工作流程如下：

1. 在工作区中修改文件。
2. 将你想要下次提交的更改选择性地暂存，这样只会将更改的部分添加到暂存区。
3. 提交更新，找到暂存区的文件，将快照永久性存储到 Git 目录。

如果 Git 目录中保存着特定版本的文件，就属于 **已提交** 状态。如果文件已修改并放入暂存区，就属于 **已暂存** 状态。如果自上次检出后，作了修改但还没有放到暂存区域，就是 **已修改** 状态。在 [Git 基础](#) 一章，你会进一步了解这些状态的细节，并学会如何根据文件状态实施后续操作，以及怎样跳过暂存直接提交。

如上，如果每个版本中有文件发生变动，Git 会将整个文件复制并保存起来。这种设计看似会多消耗更多的空间，但在分支管理时却是带来了很多的益处和便利。

### (3) Git 保证完整性

Git 中所有的数据在存储前都计算校验和，然后以校验和来引用。这意味着不可能在 Git 不知情时更改任何文件内容或目录内容。这个功能建构在 Git 底层，是构成 Git 哲学不可或缺的部分。若你在传送过程中丢失信息或损坏文件，Git 就能发现。

Git 用以计算校验和的机制叫做 **SHA-1 散列** (hash, 哈希)。这是一个由 40 个十六进制字符 (0-9 和 a-f) 组成的字符串，基于 Git 中文件的内容或目录结构计算出来。SHA-1 哈希看起来是这样：

Git 中使用这种哈希值的情况很多，你将经常看到这种哈希值。实际上，Git 数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。

## 第三章 实战

### 1、初始化Git

#### (1) 初次运行 Git 前的配置

既然已经在系统上安装了 Git，你会想要做几件事来定制你的 Git 环境。每台计算机上只需要配置一次，程序升级时会保留配置信息。你可以在任何时候再次通过运行命令来修改它们。

Git 自带一个 `git config` 的工具来帮助设置控制 Git 外观和行为的配置变量。

在 Windows 系统中，Git 会查找 `$HOME` 目录下（一般情况下是 `C:\users\$USER`）的 `.gitconfig` 文件。

你可以通过以下命令查看所有的配置以及它们所在的文件：

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

#### (2) 用户信息

安装完 Git 之后，要做的第一件事就是设置你的用户名和邮件地址。这一点很重要，因为每一个 Git 提交都会使用这些信息，它们会写入到你的每一次提交中，不可更改：

```
1 | $ git config --global user.name "itnans"
2 | $ git config --global user.email "510180298@qq.com"
```

再次强调，如果使用了 `--global` 选项，那么该命令只需要运行一次，因为之后无论你在该系统上做任何事情，Git 都会使用那些信息。当你想针对特定项目使用不同的用户名与邮件地址时，可以在那个项目目录下运行没有 `--global` 选项的命令来配置。

很多 GUI 工具都会在第一次运行时帮助你配置这些信息。

#### (3) 检查配置信息

如果想要检查你的配置，可以使用 `git config --list` 命令来列出所有 Git 当时能找到的配置。

```
1 | $ git config --list
2 | user.name=John Doe
3 | user.email=johndoe@example.com
4 | color.status=auto
5 | color.branch=auto
6 | color.interactive=auto
7 | color.diff=auto
8 | ...
```

你可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置（例如：`/etc/gitconfig` 与 `~/.gitconfig`）。这种情况下，Git 会使用它找到的每一个变量的最后一个配置。

你可以通过输入 `git config <key>`：来检查 Git 的某一项配置

```
1 | $ git config user.name  
2 | John Doe
```

## (4) 尝试

在自己方便的盘中新建一个文件夹，这里以MyProject为例，注意路径中不要含有中文字符。打开cmd命令窗口，操作如下：

找一个空文件夹：

点击鼠标右键----》git bash here

```
1 // 初始化 仓库  
2 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study  
3 $ git init  
4 Initialized empty Git repository in C:/Users/51018/Desktop/git-study/.git/  
5 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
6  
7 // 添加一个文件  
8 $ touch a.txt  
9 $ echo 123 > a.txt  
10 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
11  
12 // 提交至缓存区  
13 $ git add a.txt  
14 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
15  
16 // 提交到本地仓库  
17 $ git commit -m 'first'  
18 [master (root-commit) ac41d06] first  
19 1 file changed, 0 insertions(+), 0 deletions(-)  
20 create mode 100644 a.txt  
21  
22 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
23
```

## (5) 获取帮助

若你使用 Git 时需要获取帮助，有三种等价的方法可以找到 Git 命令的综合手册（manpage）：

```
1 | $ git help <verb>  
2 | $ git <verb> --help
```

例如，要想获得 `git config` 命令的手册，执行

```
1 | $ git help config
```

此外，如果你不需要全面的手册，只需要可用选项的快速参考，那么可以用 `-h` 选项获得更简明的“help”输出：

```
1 $ git add -h
2 usage: git add [<options>] [--] <pathspec>...
3
4     -n, --dry-run          dry run
5     -v, --verbose          be verbose
6
7     -i, --interactive      interactive picking
8     -p, --patch            select hunks interactively
9     -e, --edit              edit current diff and apply
10    -f, --force             allow adding otherwise ignored files
11    -u, --update            update tracked files
12    --renormalize          renormalize EOL of tracked files (implies -u)
13    -N, --intent-to-add    record only the fact that the path will be added
14    later
15        -A, --all           add changes from all tracked and untracked files
16        --ignore-removal     ignore paths removed in the working tree (same as
17        --no-all)
18        --refresh           don't add, only refresh the index
19        --ignore-errors      just skip files which cannot be added because of
                           errors
20        --ignore-missing     check if - even missing - files are ignored in dry
                           run
21        --chmod (+|-)x       override the executable bit of the listed files
```

## 2、基础命令

我们怎么知道哪些文件是新添加的，哪些文件已经加入了暂存区域呢？总不能让我们自己拿个本本记下来吧？

当然不，作为世界上最伟大的版本控制系统，你能遇到的困境，Git 早已有了相应的解决方案。随时随地都可以使用`git status`查看当前状态

```
1 $ git status
2 On branch master
3 nothing to commit, working tree clean
```

```
1 | $ git add b.txt
```

如果代码报错：git上传代码报错-The file will have its original line endings in your working directory  
原因是因为文件中换行符的差别导致的。

这里需要知道CRLF和LF的区别：

windows下的换行符是CRLF而Unix的换行符格式是LF。git默认支持LF。

上面的报错的意思是会把CRLF（也就是回车换行）转换成Unix格式（LF），这些是转换文件格式的警告，不影响使用。

一般commit代码时git会把CRLF转LF，pull代码时LF换CRLF。

解决方案：

```
1 | git rm -r --cached .
2 | git config core.autocrlf false
```

然后重新上传代码即可。

为true时，Git会将你add的所有文件视为文本文件，将结尾的CRLF转换为LF，而checkout时会再将文件的LF格式转为CRLF格式。

为false时，line endings不做任何改变，文本文件保持其原来的样子。

为input时，add时Git会把CRLF转换为LF，而check时仍旧为LF，所以Windows操作系统不建议设置此值。

输入git status命令，提示如下：

```
1 | 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
2 | $ echo 1234 > b.txt
3 |
4 | 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
5 | $ git add b.txt
6 |
7 | $ git status
8 | On branch master
9 | Changes to be committed:
10 |   (use "git reset HEAD <file>..." to unstage)
11 |
12 |       new file:   b.txt
13 |
14 |
15 | 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
```

**Untracked files** 说明存在未跟踪的文件（下边红色的那个）

所谓的“未跟踪”文件，是指那些新添加的并且未被加入到暂存区域或提交的文件。它们处于一个逍遥法外的状态，但你一旦将它们加入暂存区域或提交到Git仓库，它们就开始受到Git的“跟踪”。

这里圆括号中的英文是git给我们的建议：使用git add命令将待提交的文件添加到暂存区域。

```
1 F:\MyProject>git add LICENSE
2
3 F:\MyProject>git status
4 On branch master
5 Changes to be committed:
6   (use "git reset HEAD <file>..." to unstage)
7
8       new file:   LICENSE(绿色)
```

再次添加到暂存区域，然后执行 git commit -m "b.txt" 命令：

## 修改数据

```
1 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
2 $ echo 123 > b.txt
3
4 $ git status
5 On branch master
6 Changes not staged for commit:
7   (use "git add <file>..." to update what will be committed)
8   (use "git checkout -- <file>..." to discard changes in working directory)
9
10      modified:   b.txt
11
12 no changes added to commit (use "git add" and/or "git commit -a")
13
14 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
15
```

修改文件后，使用git status查看数据。

## git log 查看历史操作记录

```
1 $ git log
2 commit 5da78a44017dda030d1fe273e2a470792534ba9a (HEAD -> master)
3 Author: zhangnan <510180298@qq.com>
4 Date:   Sat Mar 13 16:01:01 2021 +0800
5
6 123
7
8 commit c7c0e3bf6d64404e3e68632c24ca13eac38b02e2
9 Author: zhangnan <510180298@qq.com>
10 Date:  Sat Mar 13 15:53:38 2021 +0800
11
12 first
13
14 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
```

```
1 * d5a12d8a966da5bf36c1f4a080c5d507398f5f59 (HEAD -> master) first
```

结果中：有head代表当前所处的分支，master代表当前是master分支。可以按下不表。

两次的提交记录看到了。--pretty=oneline

head git 中的分支，其实本质上仅仅是个指向 commit 对象的可变指针。git 是如何知道你当前在哪个分支上工作的呢？

其实答案也很简单，它保存着一个名为 HEAD 的特别指针。在 git 中，它是一个指向你正在工作中的本地分支的指针，可以将 HEAD 想象为当前分支的别名。

```
1 | $ git log --graph
```

## 3、时光回退

有关回退的命令有两个：**reset** 和 **checkout**

### (1) 回滚快照

注：快照即提交的版本，每个版本我们称之为一个快照。

现在我们利用 reset 命令回滚快照，并看看 Git 仓库和三棵树分别发生了什么。

执行 **git reset HEAD~** 命令：

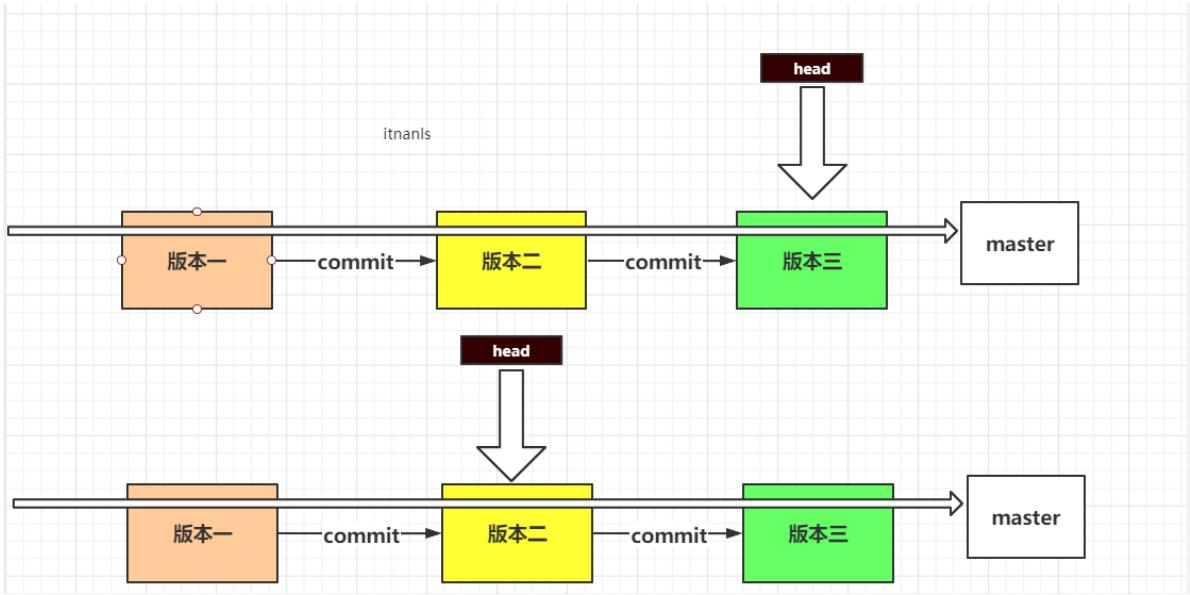
注：HEAD 表示最新提交的快照，而 HEAD~ 表示 HEAD 的上一个快照，HEAD~~ 表示上上个快照，如果表示上 10 个快照，则可以用 HEAD~10

此时我们的快找回滚到了第二棵数（暂存区域）

记住：head永远指向当前分支的当前快照

```
1 | $ git hard reset head  git reset -hard head~  
2 |  
3 | 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
4 | $ git log  
5 | commit c7c0e3bf6d64404e3e68632c24ca13eac38b02e2 (HEAD -> master)  
6 | Author: zhangnan <510180298@qq.com>  
7 | Date:   Sat Mar 13 15:53:38 2021 +0800  
8 |  
9 |     first  
10|  
11| 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
12|
```

可以看到，只剩下一个记录了。



### 参数选择

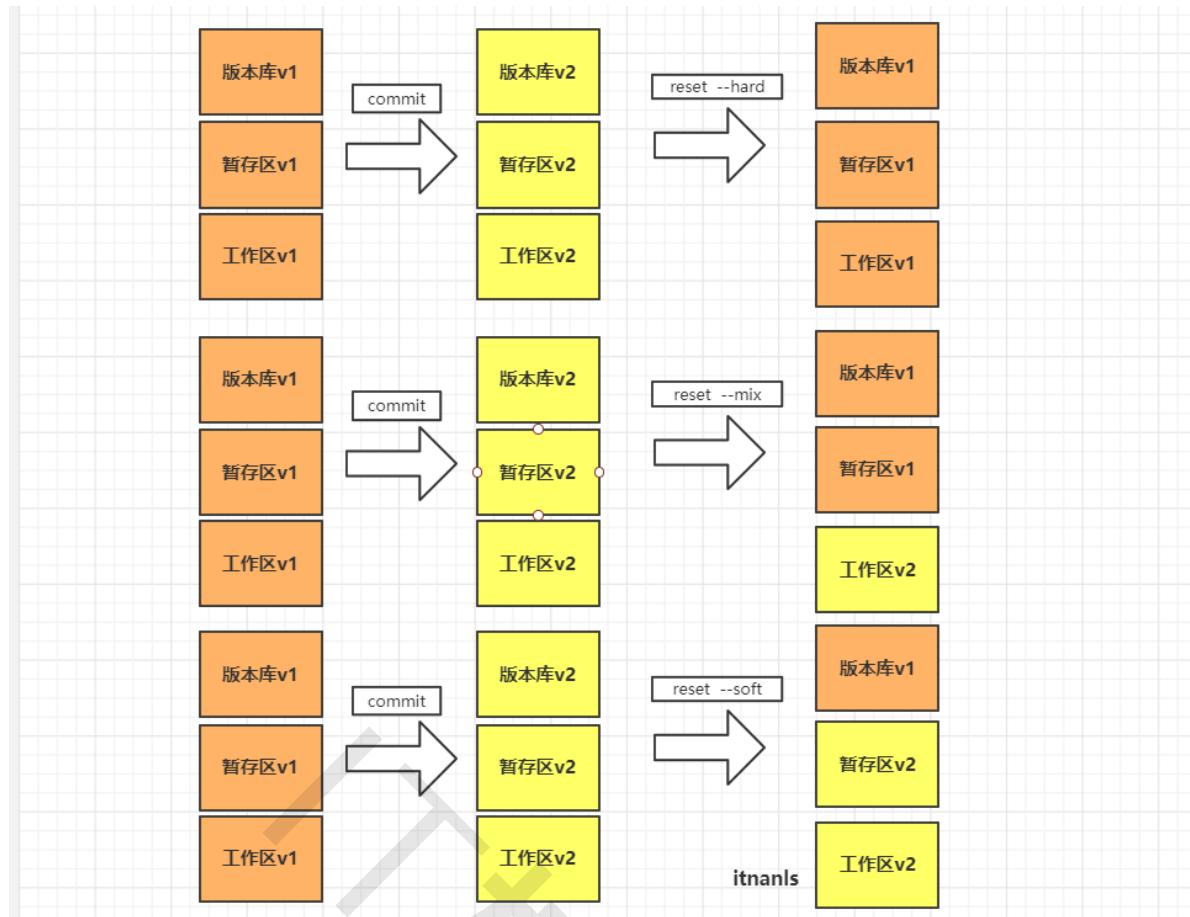
`-hard`: 回退版本库, 暂存区, 工作区。 (因此我们修改过的代码就没了, 需要谨慎使用)

`reset` 不仅移动 `HEAD` 的指向, 将快照回滚动到暂存区域, 它还将暂存区域的文件还原到工作目录。

`-mixed`: 回退版本库, 暂存区。 (`--mixed`为`git reset`的默认参数, 即当任何参数都不加的时候的参数)

`-soft`: 回退版本库。

就相当于只移动 `HEAD` 的指向, 但并不会将快照回滚到暂存区域。相当于撤销了上一次的提交  
(commit)。



## (2) 回滚指定快照

reset 不仅可以回滚指定快照，还可以回滚个别文件。

命令格式为：

```
1 | git reset --hard c7c0e3bf6d64404e3e68632c24ca13eac38b02e2
```

这样，它就会将忽略移动 HEAD 的指向这一步（因为你只是回滚快照的部分内容，并不是整个快照，所以 HEAD 的指向不应该发生改变），直接将指定快照的指定文件回滚到暂存区域。

**不仅可以往回滚，还可以往前滚！**

这里需要强调的是：reset 不仅是一个“复古”的命令，它不仅可以回到过去，还可以去到“未来”。

唯一的一个前提条件是：你需要知道指定快照的 ID 号。

**那如果不小心把命令窗口关了不记得ID号怎么办？**

命令：

```
1 | git reflog
```

Git记录的每一次操作的版本ID号

```
1 | $ git reset --hard 7ce4954
```

## 4、版本对比

### (1) 暂存区与工作树

目的：对比版本之间有哪些不同

在已经存在的文件b.txt中添加内容：

```
1 $ git diff
2 diff --git a/b.txt b/b.txt
3 index 9ab39d5..4d37a8a 100644
4 --- a/b.txt
5 +++ b/b.txt
6 @@ -2,3 +2,4 @@
7 1212
8 123123123
9 234234234
10 +手动阀手动阀
```

现在来解释一下上面每一行的含义：

**第一行：**diff --git a/b.txt b/b.txt

表示对比的是存放在暂存区域和工作目录的b.txt

**第二行：**index 9ab39d5..4d37a8a 100644

表示对应文件的 ID 分别是 9ab39d5 和 4d37a8a，左边暂存区域，后边当前目录。最后的 100644 是指定文件的类型和权限

**第三行：**---

--- 表示该文件是旧文件（存放在暂存区域）

**第四行：**+++

+++ 表示该文件是新文件（存放在工作区域）

**第五行：**@@ -2,3 +2,4 @@

以 @@ 开头和结束，中间的“-”表示旧文件，“+”表示新文件，后边的数字表示“开始行号，显示行数”

内容：+代表新增的行 -代表少了的行

直接执行 git diff 命令是比较暂存区域与工作目录的文件内容：

### (2) 工作树和最新提交

```
1 $ git diff head
2 warning: LF will be replaced by CRLF in b.txt.
3 The file will have its original line endings in your working directory
4 diff --git a/b.txt b/b.txt
5 new file mode 100644
6 index 0000000..4d37a8a
7 --- /dev/null
8 +++ b/b.txt
9 @@ -0,0 +1,5 @@
```

```
10 +123
11 +1212
12 +123123123
13 +234234234
14 +手动阀手动阀
15
16 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
17
```

### (3) 两个历史快照

```
1 $ git diff 5da78a4 c7c0e3b
2 diff --git a/b.txt b/b.txt
3 deleted file mode 100644
4 index 81c545e..0000000
5 --- a/b.txt
6 +++ /dev/null
7 @@ -1 +0,0 @@
8 -1234
```

### (4) 比较仓库和暂存区

```
1 $ git diff --cached c7c0e3b
2 diff --git a/b.txt b/b.txt
3 new file mode 100644
4 index 0000000..9ab39d5
5 --- /dev/null
6 +++ b/b.txt
7 @@ -0,0 +1,4 @@
8 +123
9 +1212
10 +123123123
11 +234234234
```

## 5、删除文件

不小心删除文件怎么办？

现在从工作目录中手动删除 b.txt 文件，然后执行 git status 命令：

```
1 $ git status
2 On branch master
3 Changes not staged for commit:
4   (use "git add/rm <file>..." to update what will be committed)
5   (use "git checkout -- <file>..." to discard changes in working directory)
6
7       deleted:    b.txt
8
9 no changes added to commit (use "git add" and/or "git commit -a")
10
```

提醒使用 checkout 命令可以将暂存区域的文件恢复到工作目录：

```
1 $ git checkout -- b.txt
```

文件就会重新返回。

那么如何彻底删除一个文件呢？

假如你不小心把小黄图下载到了工作目录，然后又不小心提交到了 Git 仓库：

新增一个c.txt文件

```
1 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
2 $ echo 123 > c.txt
3
4 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
5 $ git add .
6 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
7 $ git commit -m 'third'
8 [master 3bd84d8] third
9   1 file changed, 1 insertion(+)
10  create mode 100644 c.txt
11
```

还有方法：

执行 git rm a.txt 命令：

```
1 $ git rm c.txt
2 rm 'c.txt'
```

此时工作目录中的c.txt已经被删除.....

```
1 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
2 $ ls
3 a.txt b.txt mintty.exe.stackdump
```

但执行 git status 命令，你仍然发现 Git 还不肯松手：

```
1 $ git status
2 On branch master
3 Changes to be committed:
4   (use "git reset HEAD <file>..." to unstage)
5
6       deleted:    c.txt
```

意思是说它在仓库的快照中发现有个叫 c.txt 的东西，但似乎在暂存区域和当前目录不见了！

此时可以执行 git reset --soft HEAD~ 命令将快照回滚到上一个位置，然后重新提交，就好了：

**注意：rm 命令删除的只是工作目录和暂存区域的文件（即取消跟踪，在下次提交时不纳入版本管理）**

| 缓冲区和工作树的内容不一致，怎么删除

- 1、修改b.txt 添加至缓冲区
- 2、再修改b.txt
- 3、git rm c.txt

```
1 $ echo 123 > b.txt
2
3 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
4 $ git add b.txt
5
6 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
7 $ echo 123 > b.txt
8
9 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
10 $ git rm b.txt
11 error: the following file has changes staged in the index:
12     b.txt
13 (use --cached to keep the file, or -f to force removal)
14
```

因为两个不同内容的同名文件，谁知道你是不是搞清楚了都要删掉？还是提醒一下好，别等一下出错了又要赖机器.....

根据提示，执行 git rm -f b.txt命令就可以把两个都删除。

| 我只想删除暂存区域的文件，保留工作目录的，应该怎么操作？

执行 git rm --cached 文件名 命令。

## 6、重命名文件

直接在工作目录重命名文件，执行git status出现错误：

```
1 $ git status
```

```
2 on branch master
3 changes to be committed:
4   (use "git reset HEAD <file>..." to unstage)
5
6       modified:   b.txt
7
8 changes not staged for commit:
9   (use "git add/rm <file>..." to update what will be committed)
10  (use "git checkout -- <file>..." to discard changes in working directory)
11
12      deleted:   b.txt
13
14 untracked files:
15   (use "git add <file>..." to include in what will be committed)
16
17       n.txt
18
```

正确的姿势应该是：

```
git mv |旧文件名 新文件名
```

```
1 $ git mv b.txt c.txt
2
3 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
4 $ git status
5 On branch master
6 Changes to be committed:
7   (use "git reset HEAD <file>..." to unstage)
8
9       renamed:   b.txt -> c.txt
```

## 7、忽略文件

如何让Git识别某些格式的文件，然后自主不跟踪它们？

比如工作目录中有三个文件1.temp、2.temp 和 3.temp，我们不希望后缀名为 temp 的文件被追踪，可是每次执行git status都会出现：

解决办法：在工作目录创建一个名为 .gitignore 的文件。

然后你发现 Windows 压根儿不允许你在文件管理器中创建以点（.）开头的文件。windows需要在命令行窗口创建（.）开头的文件。执行 echo \*.temp > .gitignore 命令，创建一个 .gitignore 文件，并让 Git 忽略所有 .temp 后缀的文件：

```
1 $ echo *.temp > .gitignore
```

```
1 $ echo *.temp > .gitignore
```

在工作目录创建 a.temp

```
1 $ git status
```

```
2 on branch master
3 changes to be committed:
4   (use "git reset HEAD <file>..." to unstage)
5
6       renamed:    b.txt -> c.txt
7
8 untracked files:
9   (use "git add <file>..." to include in what will be committed)
10
11 .gitignore
12
13
14 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
```

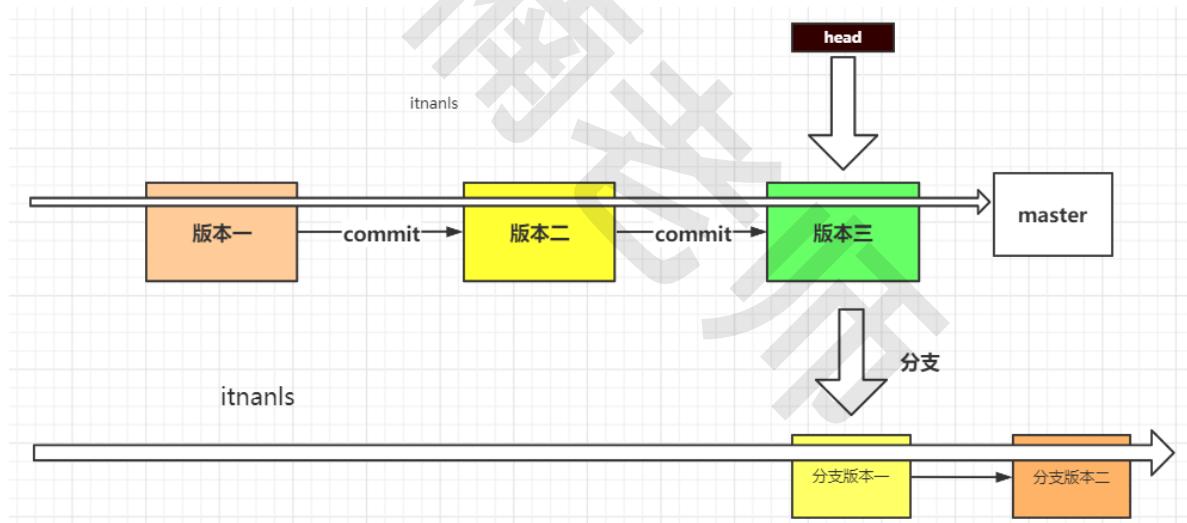
好了，Git 已经忽略了所有的 \*.temp 文件（你还可以把 .gitignore 文件也一并忽略）。

## 8、创建和切换分支

### (1) 分支是什么？

假设你的大项目已经上线了（有上百万人在使用），过了一段时间

你突然觉得应该添加一些新的功能，但是为了保险起见，你肯定不能在当前项目上直接进行开发，这时候你就有创建分支的需要了。



对于其它版本控制系统而言，创建分支常常需要完全创建一个源代码目录的副本，项目越大，耗费的时间就越多；而 Git 由于每一个结点都已经是一个完整的项目，所以只需要创建多一个“指针”（像 master）指向分支开始的位置即可。

### (2) 创建分支

执行 git status 查看状态：

```
1 $ git status
2 On branch master
```

创建分支，使用 git branch 分支名 命令：

```
1 | $ git branch feature01
```

没有任何提示说明分支创建成功（一般也不会失败啦，除非创建了同名的分支会提醒你一下），此时可以执行 git log --decorate 命令查看：

如果希望以“精简版”的方式显示，可以加上一个--oneline 选项（即git log --decorate --oneline），这样就只用一行来显示一个快照记录。

```
1 | $ git log
2 | commit 432621d36faf270eae133cfe2e976fc99df479a5 (HEAD -> master, feature01)
3 | Author: zhangnan <510180298@qq.com>
4 | Date:   Sat Mar 13 17:43:53 2021 +0800
5 |
6 |     1
7 |
8 | commit 4c9e83b6d4ca3ca3d8b0b77bb5aca614dd755413
9 | Author: zhangnan <510180298@qq.com>
10 | Date:  Sat Mar 13 17:11:51 2021 +0800
11 |
12 |     123
13 |
```

可以看到最新的快照后边多了一个 (HEAD -> master, feature01)

它的意思是：目前有两个分支，一个是主分支（master），一个是刚才我们创建的新分支（feature），然后 HEAD 指针仍然指向默认的 master 分支。

```
1 | $ git log --decorate --oneline
2 | 432621d (HEAD -> master, feature01) 1
3 | 4c9e83b 123
4 | 8af2e68 secong
5 | c7c0e3b first
```

所以目前仓库中的快照应该是这样：head -» master feature01

### (3) 切换分支

现在我们需要将工作环境切换到新创建的分支（feature）上，使用的就是之前我们欲言又止的 checkout 命令。执行 git checkout feature 命令：

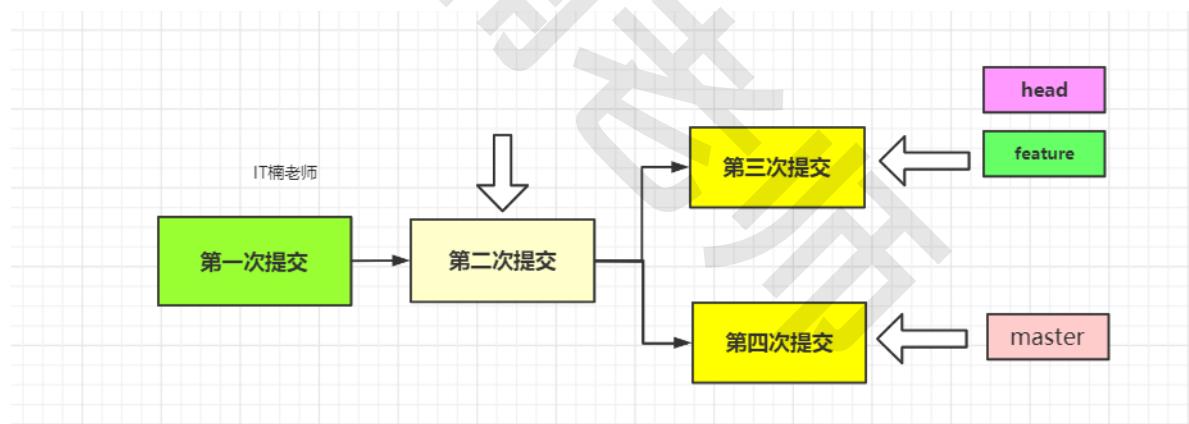
```
1 $ git checkout feature01
2 Switched to branch 'feature01'
3
4 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
5 $ git status
6 On branch feature01
7 nothing to commit, working tree clean
```

现在我们进行一次提交（暂存区域还有一个更改的文件没有提交呢）：

创建d.txt文件

```
1 $ git add d.txt
2
3 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
4 $ git commit -am 'feature01'
5 [feature01 f5e0b68] feature01
6   1 file changed, 1 insertion(+)
7   create mode 100644 d.txt
8
```

现在仓库中的快照应该是酱紫（提交的快照由当前HEAD指针指向的分支来管理）：



然后我们将 HEAD 指针切回 master 分支：

```
1 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
2 $ ls
3 a.temp a.txt c.txt d.txt mintty.exe.stackdump
4
5 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
6 $ git checkout master
7 Switched to branch 'master'
8
9 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
10 $ ls
11 a.temp a.txt c.txt mintty.exe.stackdump
12
```

细心的朋友会发现上一次对 README.md 文件的修改已经荡然无存了，这是因为我们的工作目录已经回到 master 分支的状态中：

在不同的分支分别提交

```
1 $ git status
2 On branch master
3 nothing to commit, working tree clean
4
5 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
6 $ echo 333 >> c.txt
7
8 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
9 $ git add c.txt
10 warning: LF will be replaced by CRLF in c.txt.
11 The file will have its original line endings in your working directory
12
13 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
14 $ git commit -m 'master'
15 [master baccb7f] master
16 1 file changed, 1 insertion(+)
17
18 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
19 $ git checkout feature01
20 Switched to branch 'feature01'
21
22 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
23 $ echo 333 >> c.txt
24
25 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
26 $ git add c.txt
27 warning: LF will be replaced by CRLF in c.txt.
28 The file will have its original line endings in your working directory
29
30 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
31 $ git commit -m 'feature01'
32 [feature01 b134862] feature01
33 1 file changed, 1 insertion(+)
34
```

```
35 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
36 $ git log --graph
37
38 | Author: zhangnan <510180298@qq.com>
39 | Date:   Sat Mar 13 18:00:03 2021 +0800
40 |
41 |     feature01
42 |
43 * commit f5e0b68217b66d959bf9eed7ad7631e4365f355
44 | Author: zhangnan <510180298@qq.com>
45 | Date:   Sat Mar 13 17:50:20 2021 +0800
46 |
47 |     feature01
48 |
49 | * commit baccb7f29da8143adebc79ca4c10e15204e79411 (master)
50 |/ Author: zhangnan <510180298@qq.com>
51 | Date:   Sat Mar 13 17:59:20 2021 +0800
52 |
53 |     master
54 |
55 * commit 432621d36faf270eae133cfe2e976fc99df479a5
56 | Author: zhangnan <510180298@qq.com>
57 | Date:   Sat Mar 13 17:43:53 2021 +0800
58 |
59 |     1
60 |
61 * commit 4c9e83b6d4ca3ca3d8b0b77bb5aca614dd755413
62 | Author: zhangnan <510180298@qq.com>
63 | Date:   Sat Mar 13 17:11:51 2021 +0800
64 |
65 |     123
66 |
67
68
```

## 9、合并分支

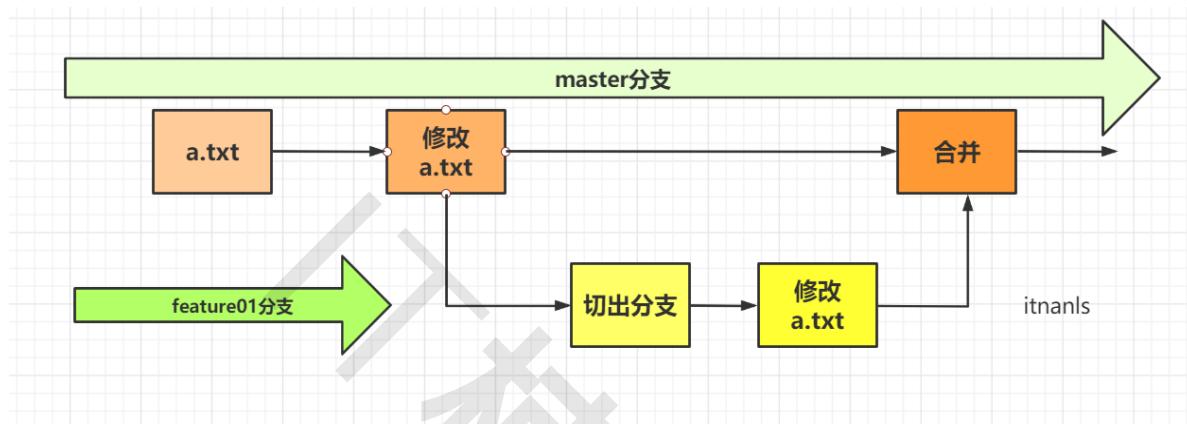
新建一个仓库

```
1 // 初始化一个仓库
2 $ git init
3 // 创建一个a.txt 文件，并且修改他的内容
4 $ touch a.txt
5
6 // 提交该分支
7 $ git add a.txt
8 $ git commit -m 'master'
9
10 // 切出一个分支
11 $ git branch feature1
12 // 切换到该分支
13 $ git checkout feature1
14 Switched to branch 'feature1'
```

```

15 // 随意修改a.txt的内容
16 .
17 // 切换回主分支
18 $ git checkout master
19 Switched to branch 'master'
20 // 合并分支
21 $ git merge feature1
22 Updating 540e027..cae5dfc
23 Fast-forward
24 a.txt | 8 ++++++-
25 1 file changed, 7 insertions(+), 1 deletion(-)
26

```



当一个子分支的使命完结之后，它就应该回归到主分支中去。

```

1 $ git log --decorate --all --graph --oneline
2 fatal: unrecognized argument: --online
3
4 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (feature01)
5 $ git log --decorate --all --graph --oneline
6 * b134862 (HEAD -> feature01) feature01
7 * f5e0b68 feature01
8 | * baccb7f (master) master
9 |/
10 * 432621d 1
11 * 4c9e83b 123
12 * 8af2e68 seong
13 * c7c0e3b first
14

```

合并分支我们使用 merge 命令，执行 git merge feature01命令，将 feature 分支合并到 HEAD 所在的分支（master）上：

第一步 切出一个feature2分支，修改master分支中a.txt第一行数据，

```

1 // 先切出一个分支
2 $ git branch feature2
3
4 // 在master分支做修改，修改a.txt的第一行数据

```

```

5 $ git add a.txt
6
7 // 提交 master 分支
8 $ git commit -m 'master'
9
10 // 切换到 feature2 分支
11 $ git checkout feature2
12 Switched to branch 'feature2'
13
14 // 同样修改 a.txt 的第一行数据
15 $ git add a.txt
16 // 提交
17 $ git commit -m feature2
18 [feature2 0ebb84a] feature2
19   1 file changed, 1 insertion(+), 1 deletion(-)
20
21 // 切换到 master 分支
22 $ git checkout master
23 Switched to branch 'master'
24
25 // 将 feature2 合并到 master 分支上
26 $ git merge feature2
27 // 发生了问题
28 Auto-merging a.txt
29 CONFLICT (content): Merge conflict in a.txt
30 Automatic merge failed; fix conflicts and then commit the result.

```

a.txt 内容变成了如下：

```

1 <<<<< HEAD
2 123123
3 =====
4 123345
5 >>>>> feature2

```

意思是说现在你需要先解决冲突的问题，Git 才能进行合并操作。所谓冲突，无非就是像两个分支中存在同名但内容却不同的文件，Git 不知道你要舍弃哪一个或保留哪一个，所以需要你自己来决定。

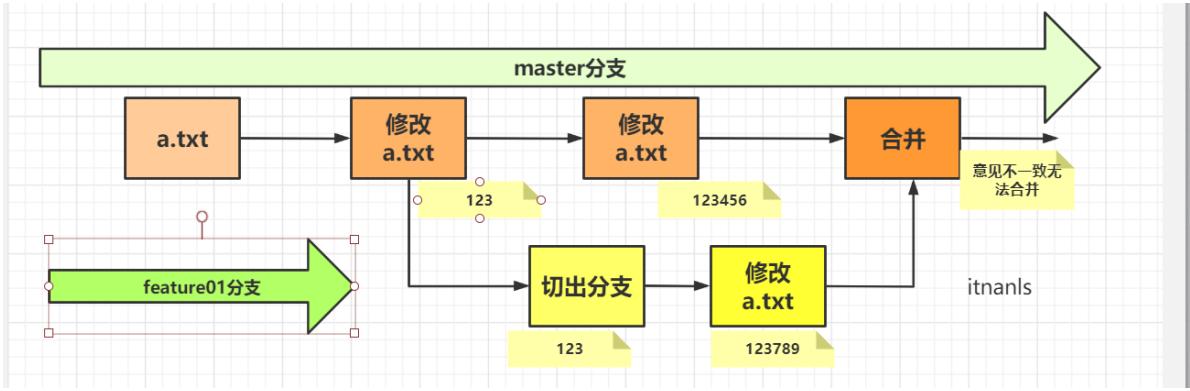
此时执行 git status 命令也会显示需要你解决的冲突：

```

1 $ git status
2 On branch master
3 You have unmerged paths.
4   (fix conflicts and run "git commit")
5   (use "git merge --abort" to abort the merge)
6
7 Unmerged paths:
8   (use "git add <file>..." to mark resolution)
9
10      both modified:  a.txt
11
12 no changes added to commit (use "git add" and/or "git commit -a")

```

以“=====”为界，上到“<<<<< HEAD”的内容表示当前分支，下到“>>>>> feature”表示待合并的 feature 分支，之间的内容就是冲突的地方。



我们就需要手动修改：

```
1 | 123123
```

```
1 | 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master|MERRGING)
2 | $ git add a.txt
3 |
4 | 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master|MERRGING)
5 | $ git commit -m '解决冲突'
6 | [master 569943e] 解决冲突
```

## 10、删除分支

当一个功能开发完成，并且成功合并到主分支，我们应该删除分支

使用 `git branch -d 分支名` 命令：

执行 `git log --decorate --all --graph --oneline` 命令：

由于 Git 的分支原理实际上只是通过一个指针记载，所以创建和删除分支都几乎是瞬间完成。

**注意：如果试图删除未合并的分支，Git 会提示你“该分支未完全合并，如果你确定要删除，请使用`git branch -D 分支名`命令。**

## 11、变基

当我们开发一个功能时，可能会在本地有无数次commit，而你实际上在你的master分支上只想显示每一个功能测试完成后的一次完整提交记录就好了，其他的提交记录并不想将来全部保留在你的master分支上，那么rebase将会是一个好的选择，他可以在rebase时将本地多次的commit合并成一个commit，还可以修改commit的描述等

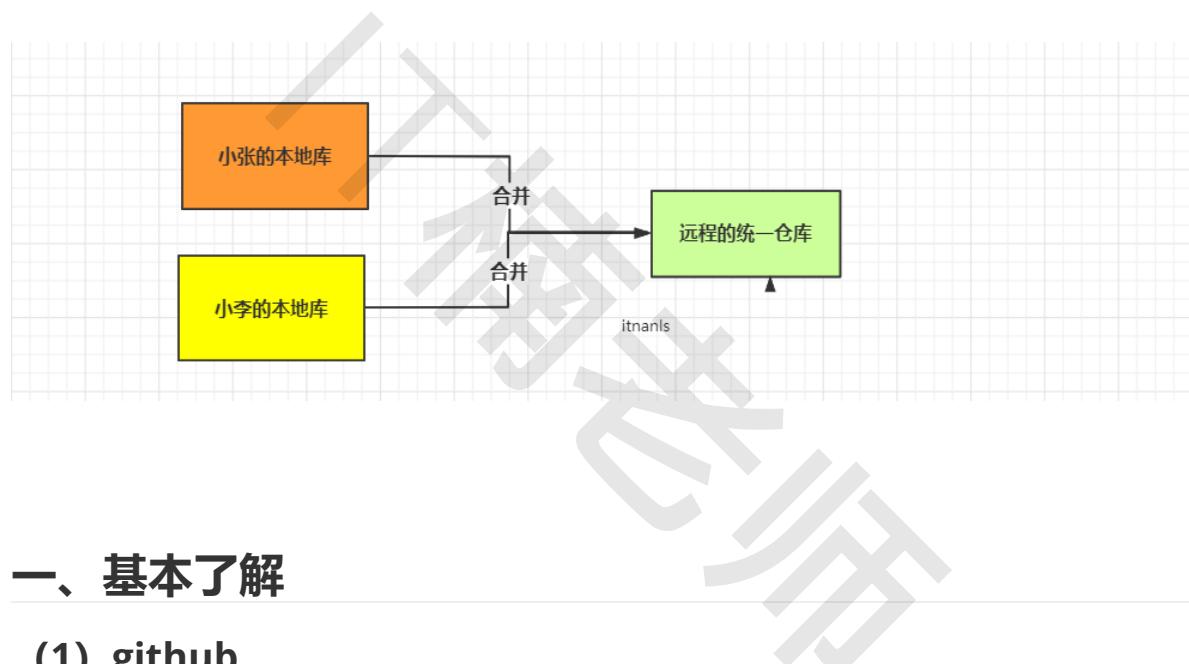
```
1 // 合并前两次的commit  
2 git rebase -i head~~  
3  
4 // 合并此次commit在最新commit的提交  
5 git rebase -i hash值
```

## 第四章 码云的使用

既然git是一个团队合作开发的工具，那本地的仓库肯定不能满足团队开发的需求！就必须要有一个远程仓库统一管理我们的代码。

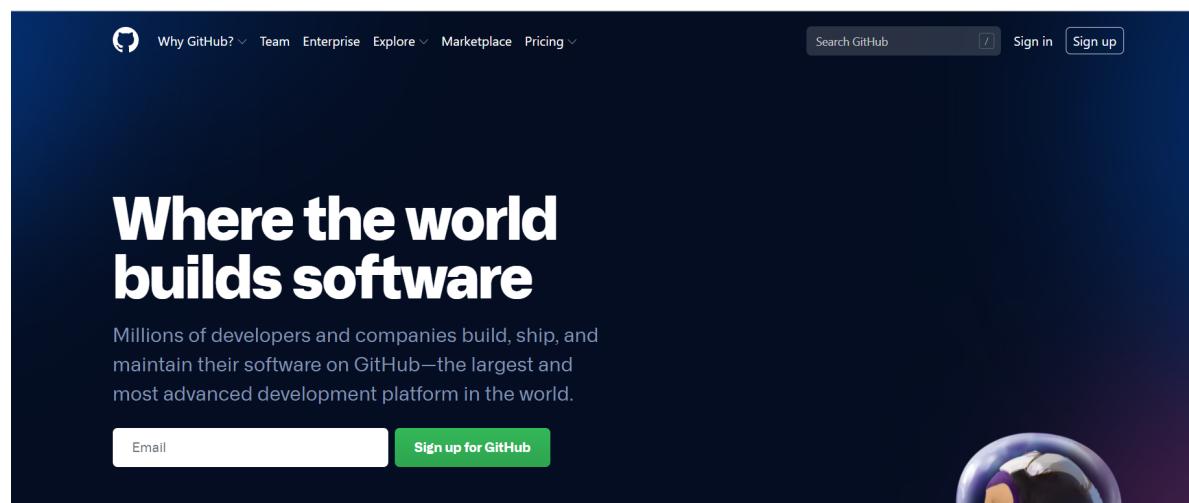
这类的工具有很多，公网上的有github，国内的有码云 gitee

公司内部直接使用 gitlab（等学习了docker后我们部署一个gitlab）



### 一、基本了解

#### (1) github



## (2) 码云

The screenshot shows the Gitee homepage. At the top, there's a navigation bar with links for '开源软件' (Open Source Software), '企业版' (Enterprise Edition), '高校版' (University Edition), '博客' (Blog), and '我的' (My). A search bar is on the right. Below the navigation, the user profile 'Alm张楠' is displayed, along with repository statistics: 19仓库 (Repositories), 0 Pull Requests, 0 任务 (Tasks), 0 代码片段 (Code Snippets), and 6 我 Star 的仓库 (Starred Repositories). The main area is divided into sections: '企业' (Enterprise), '组织' (Organization), and '仓库' (Repository). The '仓库' section shows a repository named '公开的' (Public) with 1+ commits. On the right, the '动态' (Activity) feed shows recent events like '删除了 Alm张楠/boke' and '推送到了 Alm张楠/IT楠老师的java学习资料全套 的 master 分支'. A '推荐关注' (Recommended Follow) section lists users like 'yann', 'yutons', 'Forever J 顾北', and '程序猿小码'. A '推荐仓库' (Recommended Repository) section lists repositories like 'zTree.v3' and 'bootstrap-addTab'.

## (3) gitlab独立部署

The screenshot shows the GitLab landing page. The top navigation bar includes 'Projects', 'Groups', 'Activity', 'Milestones', 'Snippets', and a search bar. The main content area is titled 'Welcome to GitLab' with the tagline 'Code, test, and deploy together'. It features five call-to-action boxes: 'Create a project' (Projects are where you store your code, access issues, wiki and other features of GitLab.), 'Create a group' (Groups are a great way to organize projects and people.), 'Add people' (Add your team members and others to GitLab.), 'Configure GitLab' (Make adjustments to how your GitLab instance is set up.), and 'Unlock more features with GitLab Ultimate' (GitLab is free to use. Many features for larger teams are part of our paid products. You can try Ultimate for free without any obligation or payment details.). A 'Start free trial' button is located in the bottom right of the 'Ultimate' section. A '查看原图' (View Original) button is at the bottom center.

但是基于网路的原因和学习成本，咱们使用码云。会一个就都回了。

## 二、基本使用

### 1、注册个账号

### 2、认识界面

开源软件--》里边有很多的优秀的开源软件学习

The screenshot shows the Gitee homepage with a dark theme. At the top, there's a navigation bar with links for '开源软件', '企业版', '高校版', '博客', and '我的'. A search bar and user profile icons are also present. The main content area features a banner for 'GVP - Gitee最有价值开源项目' (Gitee Most Valuable Project). Below the banner, three project cards are displayed: 'MaxKey Java', 'oui JavaScript', and 'hutool Java'. Each card includes a star rating, commit count, and a brief description. A button labeled '加入GVP计划' (Join GVP Plan) is visible. To the right, there's a purple sidebar for 'Gitee Go' with the text '持续集成流水线 免费内测' (Continuous Integration Pipeline Free Internal Testing) and a '马上体验' (Experience Now) button.

## 我的控制台

This screenshot shows the 'My Control Panel' (我的控制台) page. At the top, there's a navigation bar with '我的' highlighted. The main area includes a user profile section for 'Alm张楠', a '动态' (Activity) feed showing recent events like pushing code and commenting on issues, and a '推荐关注' (Recommended Follow) section listing several users with their profiles and follow buttons. On the left, there are sections for '企业' (Enterprise), '组织' (Organization), and '仓库' (Repositories), each with a '+' button to add new items.

This screenshot shows the user profile page for 'Alm张楠'. It displays basic user stats: 6 stars, 23 watches, 115 followers, and 0 following. Below this, there are sections for '个人设置' (Personal Settings) and '贡献度和动态' (Contribution and Activity). The '贡献度' section shows a chart with four segments: '活跃度' (Activity), '贡献度' (Contribution), '活跃度' (Activity), and '贡献度' (Contribution). The '动态' section shows a timeline of recent contributions and interactions. The '贡献度' section shows a chart with four segments: '活跃度' (Activity), '贡献度' (Contribution), '活跃度' (Activity), and '贡献度' (Contribution).

## 贡献度和动态



## 动态

今天 2021-03-16

## 认识仓库

- 1、pull Request 开发者在本地对源代码进行修改之后，想仓库提交请求合并的功能
- 2、Wiki 该功能通常用作文档手册的编写当中
- 3、Issues：是将一个任务或问题分配给一个issue进行跟踪和管理，可以当做bug管理系统使用，每一个功能的更正或修改都应该对应一个issue，只要看issues就能看到关于这个更改的所有信息
- 4、统计就是仓库各项数据的数据统计，devOPs是持续继承、持续交付的服务，服务：其他码云提供的  
一些服务。
- 5、管理：对仓库的一些修改删除等操作：

## 3、新建仓库

The header of the Gitee website. It features the Gitee logo, navigation links for '开源软件' (Open Source), '企业版' (Enterprise Edition) with a red badge labeled '特惠' (Special Offer), '高校版' (University Edition), '博客' (Blog), '我的' (My Profile), a search bar with placeholder '搜开源', and user profile icons.

## 新建仓库

仓库名称 ✓  
git-study

归属 路径 ✓  
Alm张楠 / git-study

仓库地址: <https://gitee.com/zhangnan716/git-study>

仓库介绍 非必填  
用来学习git的基本操作

是否开源  
 私有  公开

私有仓库的非仓库成员无法访问该仓库的代码和其他任何形式的资源  
私有仓库最多支持 5 人协作 (如拥有多个私有仓库, 所有协作人数总计不得超过 5 人)  
企业仓库, 更适合使用 Gitee 企业版, [了解更多 >>](#)

选择语言 添加 .gitignore  
请选择语言 请选择 .gitignore 模板

是否开源  
 私有  公开

私有仓库的非仓库成员无法访问该仓库的代码和其他任何形式的资源  
私有仓库最多支持 5 人协作 (如拥有多个私有仓库, 所有协作人数总计不得超过 5 人)  
企业仓库, 更适合使用 Gitee 企业版, [了解更多 >>](#)

选择语言 添加 .gitignore  
Java 请选择 .gitignore 模板

使用 README 文件初始化这个仓库  
 使用 Issue 模板文件初始化这个仓库 ⓘ  
 使用 Pull Request 模板文件初始化这个仓库 ⓘ

选择分支模型 (仓库初始化后将根据所选分支模型创建分支)  
单分支模型 (只创建 master 分支)

导入已有仓库

**创建**

尝试 Gitee 企业版?

- 专业研发管理平台
- 有序规划和管理软件研发全流程

与他们一起提升研发效能

已有超过 180,000 家企业客户



[了解更多](#)

[Gitee 企业版介绍](#)

[社区版与企业版功能对比](#)



[了解更多](#)

[Gitee 企业版介绍](#)

[社区版与企业版功能对比](#)

全部不勾, 创建一个空仓库就行了

第一次进入

快速设置—如果你知道该怎么操作，直接使用下面的地址

HTTPS SSH <https://gitee.com/zhangnan716/git-study.git>

我们强烈建议所有的git仓库都有一个 `README`, `LICENSE`, `.gitignore` 文件

**初始化 readme 文件**

Git入门? 查看 帮助 , Visual Studio / TortoiseGit / Eclipse / Xcode 下如何连接本站 如何导入仓库

简易的命令行入门教程:

Git 全局设置:

```
git config --global user.name "Alm张楠"
git config --global user.email "510180298@qq.com"
```

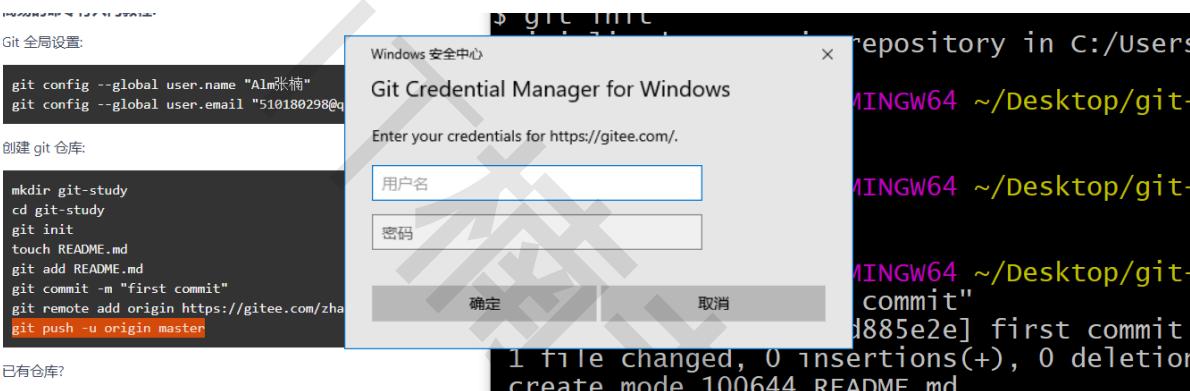
创建 git 仓库:

```
mkdir git-study
cd git-study
git init
touch README.md
git add README.md
git commit -m "first commit"
git remote add origin https://gitee.com/zhangnan716/git-study.git
git push -u origin master
```

已有仓库?

```
cd existing_git_repo
git remote add origin https://gitee.com/zhangnan716/git-study.git
git push -u origin master
```

## 会让你输入密码



## 4、建立本地仓库

```

1
2 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study
3 $ git config --global user.name "Alm张楠"
4
5 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study
6 $ git config --global user.email "510180298@qq.com"
7
8 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study
9 $ cd git-study
10 bash: cd: git-study: No such file or directory
11 touch README.md
12 git add README.md
13 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study
14 $ git init
15 Initialized empty Git repository in C:/Users/51018/Desktop/git-study/.git/
16
17 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
18 $ touch README.md
19
20 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)
21 $ git add README.md

```

```
22  
23 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
24 $ git commit -m "first commit"  
25 [master (root-commit) d885e2e] first commit  
26 1 file changed, 0 insertions(+), 0 deletions(-)  
27 create mode 100644 README.md  
28  
29 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
30 $ git remote add origin https://gitee.com/zhangnan716/git-study.git  
31  
32 51018@DESKTOP-6R8BLO2 MINGW64 ~/Desktop/git-study (master)  
33 $ git push -u origin master  
34 Enumerating objects: 3, done.  
35 Counting objects: 100% (3/3), done.  
36 Writing objects: 100% (3/3), 215 bytes | 215.00 KiB/s, done.  
37 Total 3 (delta 0), reused 0 (delta 0)  
38 remote: Powered by GITEE.COM [GNK-5.0]  
39 To https://gitee.com/zhangnan716/git-study.git  
 * [new branch] master -> master  
40 Branch 'master' set up to track remote branch 'master' from 'origin'.  
41
```

### 第一个关键：添加一个远程仓库

```
1 | $ git remote add origin https://gitee.com/zhangnan716/git-study.git
```

### 把代码推送到远程仓库

```
1 | $ git push -u origin master
```

### 列出所有的远程仓库

```
1 | git remote -v
```

### 显示某个远程仓库信息

```
1 | git remote show [remote]
```

### git push 的其他命令\*\*

这几个常见的用法已足以满足我们日常开发的使用了，还有几个扩展的用法，如下：

```
1 | git push -u origin master
```

如果当前分支与多个主机存在追踪关系，则可以使用 -u 参数指定一个默认主机，这样后面就可以不加任何参数使用git push，

不带任何参数的git push，默认只推送当前分支，这叫做simple方式，还有一种matching方式，会推送所有有对应的远程分支的本地分支，Git 2.0之前默认使用matching，现在改为simple方式

如果想更改设置，可以使用git config命令。git config --global push.default matching OR git config --global push.default simple；可以使用git config -l 查看配置

```
1 | git push --all origin
```

当遇到这种情况就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要 -all 选项

```
1 | git push --force origin
```

git push的时候需要本地先git pull更新到跟服务器版本一致，如果本地版本库比远程服务器上的低，那么一般会提示你git pull更新，如果一定要提交，那么可以使用这个命令。

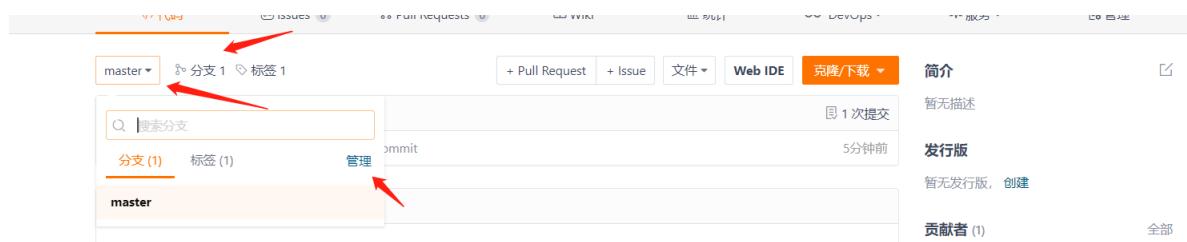


## 5、其他功能

### 打标签



### 分支选择



### 创建分支

The screenshot shows a top navigation bar with tabs: '代码' (Code), 'Issues 0', 'Pull Requests 0', 'Wiki', '统计' (Statistics), 'DevOps', '服务', and '管理'. Below the navigation is a section titled '所有分支' (All Branches) with a note about branch protection rules. A modal dialog is open for creating a new branch, with fields for '起点' (Start) set to 'master' and '新分支名称' (New Branch Name) set to 'dev'. A large orange '提交' (Submit) button is at the bottom right of the dialog.

The screenshot shows a top navigation bar with tabs: '代码' (Code), 'Issues 0', 'Pull Requests 0', 'Wiki', '统计' (Statistics), 'DevOps', '服务', and '管理'. Below the navigation is a branch details page for 'dev'. It shows a commit history with one entry by 'Alm张楠' (commit d885e2e, first commit, 5 minutes ago). On the right side, there are sections for '简介' (Description), '发行版' (Release), '贡献者' (Contributors), and '近期动态' (Recent Activity).

在统计模块，你还可以发布发型版，提供下载

The screenshot shows a top navigation bar with tabs: '访问统计' (Access Statistics), '仓库数据统计' (Repository Data Statistics), '仓库网络图' (Repository Network Graph), '发行版' (Release), '标签' (Tags), '提交' (Commits), and '附件' (Attachments). Below the navigation is a '创建发行版' (Create Release) form. The '版本' (Version) field contains 'v1.0', which is highlighted in green with a checkmark indicating it's a valid release candidate. The '标签' (Label) field contains 'sdf'. To the right of the form are several informational boxes: '标签命名建议' (Label Naming Suggestions), '语义化版本' (Semantic Versioning), and '附件大小说明' (Attachment Size Instructions). At the bottom is a large file upload area with the placeholder '拖拽文件或点击此处上传文件' (Drag files or click here to upload) and a large orange '创建发行版' (Create Release) button.

v1.0 d885e2e  
2021-03-16 14:51

sdf

Alm张楠  
a阿斯蒂芬

最后提交信息为: first commit

下载

下载 Source code (zip)  
 下载 Source code (tar.gz)

## 三、一般的协同开发流程

### 1、仓库

#### (1) 源仓库(线上版本库)

在项目的开始,项目的发起者构建起一个项目的最原始的仓库,称为**origin**。

源仓库的有两个作用:

- 汇总参与该项目的各个开发者的代码
- 存放趋于稳定和可发布的代码

源仓库应该是受保护的,开发者不应该直接对其进行开发工作。只有项目管理者能对其进行较高权限的操作。

#### (2) 开发者仓库(本地仓库)

任何开发者都不会对源仓库进行直接的操作,源仓库建立以后,每个开发者需要做的事情就是把源仓库的“复制”一份,作为自己日常开发的仓库。这个复制是gitlab上面的**fork**。

每个开发者所fork的仓库是完全独立的,互不干扰,甚至与源仓库都无关。每个开发者仓库相当于一个源仓库实体的影像,开发者在这个影像中进行编码,提交到自己的仓库中,这样就可以轻易地实现团队成员之间的并行开发工作。而开发工作完成以后,开发者可以向源仓库发送pull request,请求管理员把自己的代码合并到源仓库中,这样就实现了**分布式开发工作**和**集中式的管理**。

### 2、分支划分 (Branch)

#### (1) master branch: 主分支

**master**: 主分支从项目一开始便存在,它用于存放经过测试,已经完全稳定的代码;在项目开发以后的任何时候当中, **master** 存放的代码应该是可作为产品供用户使用的代码。所以,应该随时保持**master** 仓库代码的清洁和稳定,确保入库之前是通过完全测试和代码review的。**master** 分支是所有分支中最不活跃的,大概每个月或每两个月更新一次,每一次**master** 更新的时候都应该用**git** 打上**tag**,来说明产品有新版本发布。

## (2) develop branch: 开发分支

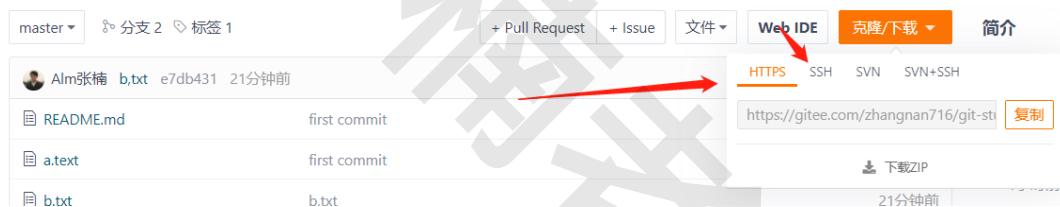
**develop**: 开发分支，一开始从**master**分支中分离出来，用于开发者存放基本稳定代码。每个开发者的仓库相当于源仓库的一个镜像，每个开发者自己的仓库上也有**master**和**develop**。开发者把功能做好以后，是存放到自己的**develop**中，当测试完以后，可以向管理者发起一个pull request，请求把自己仓库的**develop**分支合并到源仓库的**develop**中。所有开发者开发好的功能会在源仓库的**develop**分支中进行汇总，当**develop**中的代码经过不断的测试，已经逐渐趋于稳定了，接近产品目标了。这时候，就可以把**develop**分支合并到**master**分支中，发布一个新版本。

注:任何人不应该向**master**直接进行无意义的合并、提交操作。正常情况下，**master**只应该接受**develop**的合并，也就是说，**master**所有代码更新应该源于合并**develop**的代码。

## (3) feature branch: 功能分支

**feature**: 功能性分支，是用于开发项目的功能的分支，是开发者主要战斗阵地。开发者在本地仓库从**develop**分支分出功能分支，在该分支上进行功能的开发，开发完成以后再合并到**develop**分支上，这时候功能性分支已经完成任务，可以删除。功能性分支的命名一般为**feature-\***，|\*为需要开发的功能的名称。

## (4) 协议选择



### http好还是ssh好

- git可以使用四种主要的协议来传输资料: 本地协议 (Local) , HTTP 协议, SSH (Secure Shell) 协议及 git 协议。其中，本地协议由于目前大都是进行远程开发和共享代码所以一般不常用，而git协议由于缺乏授权机制且较难架设所以也不常用。
- 最常用的便是SSH和HTTP(S)协议。git关联远程仓库可以使用http协议或者ssh协议。

### HTTPS优缺点

- 优点1: 相比 SSH 协议，可以使用用户名 / 密码授权是一个很大的优势，这样用户就不必须在使用 Git 之前先在本地生成 SSH 密钥对再把公钥上传到服务器。对非资深的使用者，或者系统上缺少 SSH 相关程序的使用者，HTTP 协议的可用性是主要的优势。与 SSH 协议类似，HTTP 协议也非常快和高效
- 优点2: 企业防火墙一般会打开 80 和 443 这两个常见的http和https协议的端口，使用http和https的协议在架设了防火墙的企业里面就可以绕过安全限制正常使用git，非常方便
- 缺点: 使用http/https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令。但是现在操作系统或者其他git工具都提供了 **keychain** 的功能，可以把你的账户密码记录在系统里，例如 OSX 的 Keychain 或者 Windows 的凭证管理器。所以也只需要输一次密码就搞定了。

### SSH的优缺点

- 优点1: 架设 Git 服务器时常用 SSH 协议作为传输协议。因为大多数环境下已经支持通过 SSH 访问——即时没有也比较很容易架设。SSH 协议也是一个验证授权的网络协议；并且，因为其普遍性，架设和使用都很容易。
- 缺点1: SSH服务端一般使用22端口，企业防火墙可能没有打开这个端口。
- 缺点2: SSH 协议的缺点在于你不能通过他实现匿名访问。即便只要读取数据，使用者也要有通过 SSH 访问你的主机的权限，这使得 SSH 协议不利于开源的项目。如果你只在公司网络使用，SSH 协议可能是你唯一要用到的协议。如果你要同时提供匿名只读访问和 SSH 协议，那么你除了为自己推送架设 SSH 服务以外，还得架设一个可以让其他人访问的服务。

## 总结

HTTPS利于匿名访问，适合开源项目可以方便被别人克隆和读取(但他没有push权限)；毕竟为了克隆别人一个仓库学习一下你就要生成个ssh-key折腾一番还是比较麻烦，所以github除了支持ssh协议必然提供了https协议的支持。

而SSH协议使用公钥认证比较适合内部项目。当然了现在的代码管理平台例如github、gitlab，两种协议都是支持的，基本上看自己喜好和需求来选择就可以了。

## 生成/添加SSH公钥

### [SSH Key SSH 公钥](#)

Gitee 提供了基于SSH协议的Git服务，在使用SSH协议访问仓库仓库之前，需要先配置好账户/仓库的 SSH公钥。

你可以按如下命令来生成 sshkey:

```
1 | ssh-keygen -t rsa -C "510180298@qq.com"
2 | # Generating public/private rsa key pair...
```

注意：这里的 `xxxxx@xxxxx.com` 只是生成的 sshkey 的名称，并不约束或要求具体命名为某个邮箱。

现网的大部分教程均讲解的使用邮箱生成，其一开始的初衷仅仅是为了便于辨识所以使用了邮箱。

按照提示完成三次回车，即可生成 ssh key。通过查看 `~/.ssh/id_rsa.pub` 文件内容，获取到你的 public key

```
1 | cat ~/.ssh/id_rsa.pub
2 | # ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQC6eNtGpNGwstc....
```

```

[root@VM_3_177_centos ~]# ssh-keygen -t rsa -C "root@VM_3_177_centos.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:KusL8juVg3THVybmvRx5z21VWhIqNk6vEo/NzLh20 n
The key's randomart image is:
+---[RSA 2048]---+
| ..o... .oo |
| * = o . |
| . @ + . |
| . . . . |
| . o . |
| . . . . |
| . o++ + + |
+---[SHA256]---+
[root@VM_3_177_centos ~]# cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDEasR1exU62AwGauGXamfomk+OwagYzTeyGaIGeAvkd0GR5951PqgZ9k_
vmmpwv
/vqqca0
wxw1 TChs
ma...com
[root@VM_3_177_centos ~]#

```

生成SSH Key

三次回车

查看获取 public key

此处输出内容为 public key 内容

复制生成后的 ssh key，通过仓库主页「管理」->「部署公钥管理」->「添加部署公钥」，添加生成的 public key 添加到仓库中。

部署公钥管理

+添加部署公钥

部署公钥允许以只读的方式访问项目，主要用于项目在生产服务器的部署上，免去HTTP方式每次操作都要输入密码和普通SSH方式担心不小心修改项目代码的麻烦。

部署公钥配置后的机器，只支持clone或pull等只读操作。如果您想要对仓库进行写操作，请[添加个人公钥](#)

**已启用公钥** 可部署公钥

添加新的公钥 或 启用 可部署公钥

添加后，在终端（Terminal）中输入

```
1 | ssh -T git@gitee.com
```

首次使用需要确认并添加主机到本机SSH可信列表。若返回 `Hi XXX! You've successfully authenticated, but Gitee.com does not provide shell access.` 内容，则证明添加成功。

```

[root@VM_3_177_centos ~]# ssh -T git@gitee.com
The authenticity of host 'gitee.com (120.55.226.24)' can't be established.
ECDSA key fingerprint is SHA256:FQGC...
ECDSA key fingerprint is MD5:27:ea:a7:20:4...
Are you sure you want to continue connecting (yes/no)? yes
warning: permanently added "gitee.com,120.55.226.24" (ECDSA) to the list of known hosts.
Hi ! You've successfully authenticated, but Gitee.com does not provide shell access.

```

添加成功后，就可以使用SSH协议对仓库进行操作了。

```

1 | $ ssh -T git@gitee.com
2 | Hi Alm张楠 (DeployKey)! You've successfully authenticated, but GITEE.COM does
   | not provide shell access.
3 | Note: Perhaps the current user is DeployKey.
4 | Note: DeployKey only supports pull/fetch operations

```

为了便于用户在多个项目仓库下使用一套公钥，免于重复部署和管理的繁琐，Gitee 推出了「可部署公钥」功能，支持在一个仓库空间下使用当前账户名下/参与的另一个仓库空间的部署公钥，实现公钥共用。

部署公钥允许以只读的方式访问仓库，主要用于仓库在生产服务器的部署上，免去HTTP方式每次操作都要输入密码和普通SSH方式担心不小心修改仓库代码的麻烦。

部署公钥配置后的机器，只支持clone与pull等只读操作。如果您想要对仓库进行写操作，请[添加个人公钥](#)

部署公钥允许以只读的方式访问仓库，主要用于仓库在生产服务器的部署上，免去HTTP方式每次操作都要输入密码和普通SSH方式担心不小心修改仓库代码的麻烦。

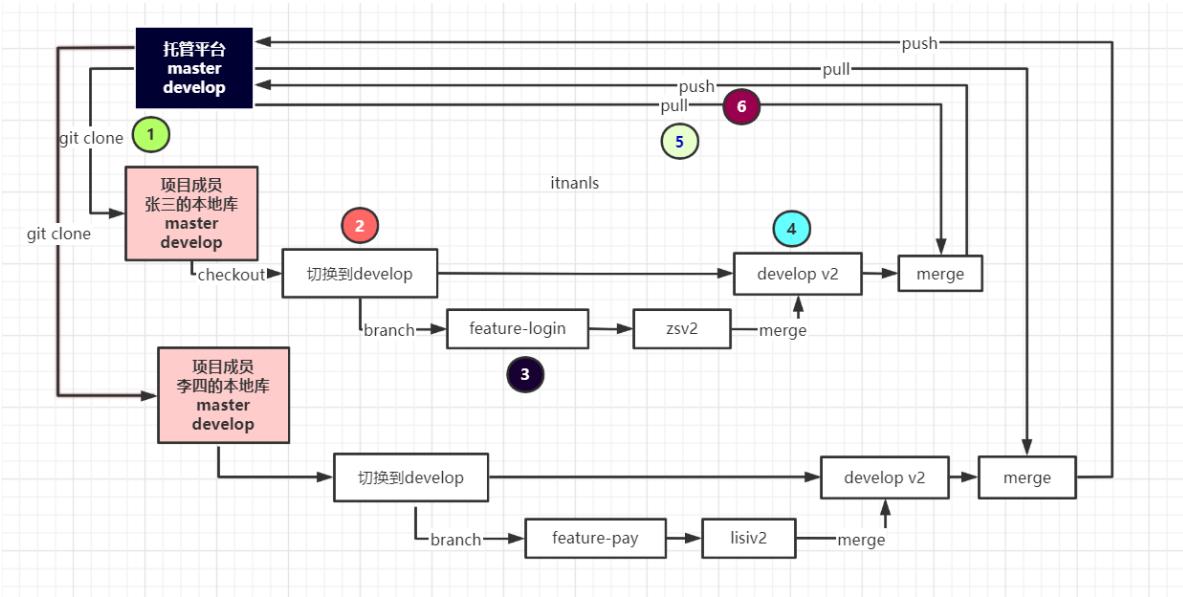
部署公钥配置后的机器，只支持clone与pull等只读操作。如果您想要对仓库进行写操作，请[添加个人公钥](#)

个人公钥的添加地址：

The screenshot shows the Gitee user profile page for 'Alm张楠'. On the left sidebar, there is a red arrow pointing to the 'SSH公钥' (SSH Public Key) link under the '安全设置' (Security Settings) section. The main content area is titled 'SSH公钥' (SSH Public Key). It contains a note: '使用SSH公钥可以让你在你的电脑和 Gitee 通讯的时候使用安全连接 (Git的Remote要使用SSH地址)' (Using an SSH key allows you to communicate with Gitee using a secure connection (Git's Remote must use an SSH address)). Below this, it says '您当前的SSH公钥数: 0' (Current number of SSH keys: 0) and '你还没有添加任何SSH公钥' (You have not added any SSH keys yet). A form for adding a new key is shown, with a red arrow pointing to the '标题' (Title) input field which contains '公钥标题(key)'. The '公钥' (Key) input field has placeholder text: '把你的公钥粘贴到这里, 查看 [怎样生成公钥](#)' (Paste your public key here, see [How to generate a public key](#)). Below the input fields is an orange '确定' (Confirm) button.

### 3、实战

#### (1) 多人合作开发流程



1、首先项目经理初始化仓库建立好分支。一般会建立两个，一个master分支，一个develop分支。当然，也可能建立一个预发布版本的分支用于测试不如 realse分支。

2、对个分支设置保护行为。

3、添加项目成员。

### 小张的开发

- 1、将项目克隆到本地。
- 2、切换至开发分支
- 3、在开发分支上新建一个单独的功能分支，进行开发。
- 4、开发完成，合并到开发分支，如果功能分支没用了，可以删除。
- 5、先拉取新代码（git pull），其实就是合并，发生冲突，解决冲突。
- 6、解决完冲突，将代码推送至代码托管平台。

### 1、新建仓库

### 选择项目分支保护

**新建保护分支规则**

设置分支/通配符  
master

例如：设置为“master”，则对名称为“master”的分支生效；设置为“\*.stable”或“release\*\*”，则对名称符合此通配符的所有保护分支生效

可推送代码成员  
Alm张楠

可合并 Pull Request 成员  
Alm张楠

保存 取消

添加开发者（如果你的仓库是公共的，直接搜也行）

**开发者管理**

仓库成员配额说明  
个人私有仓库最多支持 5 人协作 (如个人拥有多个私有仓库，所有协作人数总计不得超过 5 人)

开发者 (1) [赵登凯](#)

赵 登凯 zhao-dengkai (1210003591@qq.com)

权限说明：开发者能推送代码，新建和删除分支，创建Issue,Pull Request,Wiki 等

开发者 移出仓库

此时开发者的账户会出现在该仓库

将项目克隆到本地，进行开发

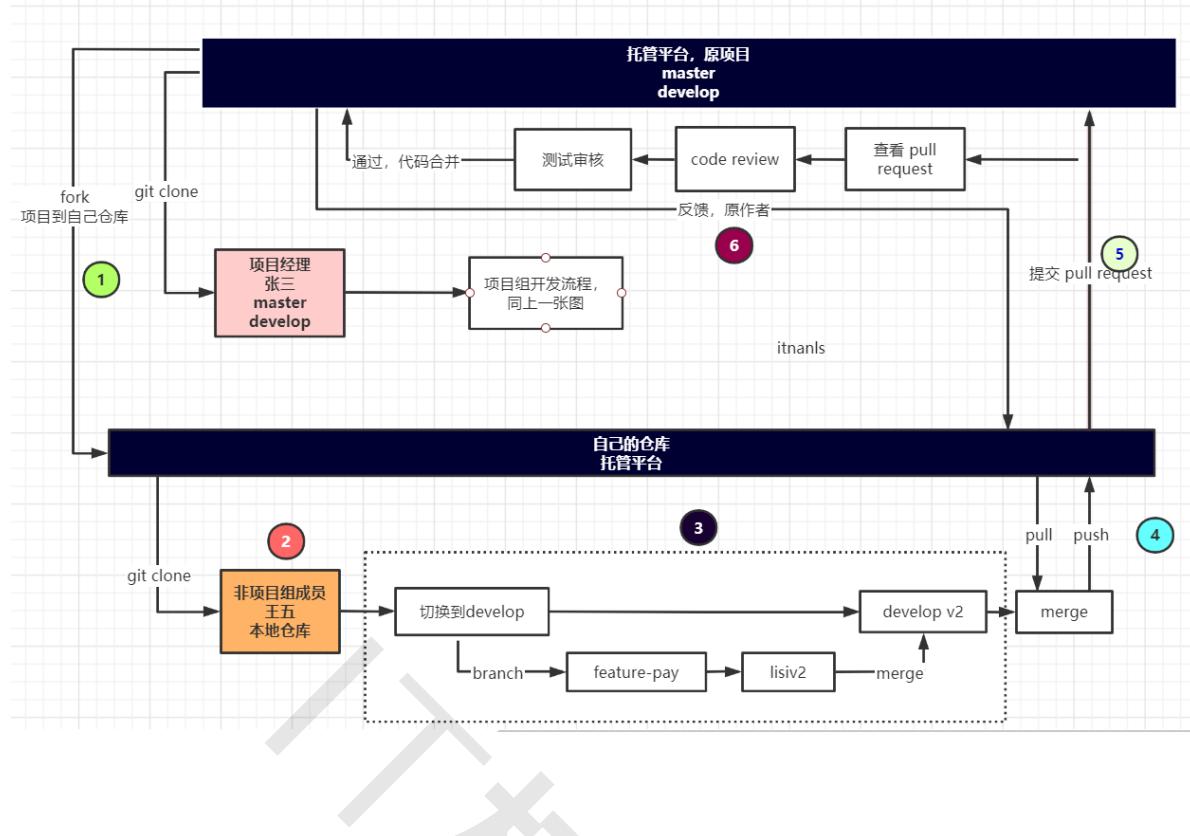
开发完成推送至远程仓库

```

1 >>> git clone 仓库地址
2 >>> git checkout develop
3 # 切换到`develop`分支
4 >>> git checkout -b feature-discuss
5 # 分出一个功能性分支
6 >>> touch discuss.java
7 # 假装discuss.java就是我们要开发的功能
8 >>> git add .
9 >>> git commit -m 'finish discuss feature'
10 # 提交更改，多次测试以后
11 >>> git checkout develop
12 # 回到develop分支
13 >>> git merge feature-discuss
14 # 把做好的功能合并到develop中
15 >>> git branch -d feature-discuss
16 # 删除功能性分支
17 >>> git push origin develop
18 # 把develop提交到远程仓库中

```

## (2) 跨团队合作开发



项目组成员的开发保持不变。

### 跨团队成员的合作方式

- 1、将代码fork到自己的仓库，同样可以进行相关的配置。
- 2、项目克隆到本地。
- 3、可以担任开发也可以多人开发。
- 4、开发完成后合并到自己的仓库
- 5、发起pull request请求给源仓库管理员
- 6、源仓库管理员进行code review（重新检查代码，审核代码），测试审核，通过则进行合并。

### 源仓库的构建

这一步通常由项目发起人(项目管理员)来操作，源仓库为op/Chanjet\_Asset\_Management，并初始化两个分支master和develop.

#### 1、新建仓库

仓库设置

- 基本设置
- 转移仓库
- 清空仓库
- 存储库 GC
- 删除仓库
- 代码审查设置

仓库成员管理

部署公钥管理

环境变量管理

仓库挂件

WebHooks

### 代码审查设置

本功能用于设置 Pull Request 默认审查/测试人员与合并规则  
指派人员将作为 Pull Request 创建时的预设指派，且不允许 Pull Request 创建者修改，如未设置，则允许创建者自行指派。该设置项不会影响已创建指派的 Pull Request

指派审查人员 Alm张楠 (zhangnan716)

至少需要 1 名审查人员审查通过后可合并

指派测试人员 Alm张楠 (zhangnan716)

至少需要 1 名测试人员测试通过后可合并

**保存**

## 选择项目分支保护

仓库设置

仓库成员管理

部署公钥管理

环境变量管理

仓库挂件

WebHooks

只读文件管理 **New**

保护分支设置

### 新建保护分支规则

设置分支/通配符  
master

例如：设置为“master”，则对名称为“master”的分支生效；设置为“\*-stable”或“release\*”，则对名称符合此通配符的所有保护分支生效

可推送代码成员  
 Alm张楠

可合并 Pull Request 成员  
 Alm张楠

**保存** **取消**

## 开发者fork仓库到自己的账户下，作为自己开发所用的仓库。



Watching 2 Star 0 Fork 0

+ Pull Request + Issue 文件 Web IDE 克隆/下载

3 次提交 2小时前

简介 暂无描述

发行版 (1) 全部

## 把自己开发者仓库clone到本地

```
1 | >>> git clone https://gitee.com/zhao-dengkai/git-study.git
```

## 修改内容，并提交，这里直接在码云上修改

A screenshot of a GitHub repository page for 'git-study'. The top navigation bar shows the repository name '赵登凯 / git-study' with a star icon. Below it, a message says 'forked from Alm张楠 / git-study'. On the right, there are buttons for 'Watching' (1), 'Star' (0), 'Fork' (1), and 'Issues' (0). The main menu includes '代码' (Code) which is underlined, 'Issues' (0), 'Pull Requests' (0), 'Wiki', '统计' (Statistics), 'DevOps' (DevOps), '服务' (Services), and '管理' (Management). The 'README.md' file is open in the code editor, showing the text '阿达算法阿斯蒂芬阿斯蒂芬撒地方撒地方撒地方'. A toolbar with various icons is visible above the editor area.

构建功能分支进行开发

假设现在要开发一个“讨论”功能：

```
1 >>> git checkout develop
2 # 切换到`develop`分支
3 >>> git checkout -b feature-discuss
4 # 分出一个功能性分支
5 >>> touch discuss.java
6 # 假装discuss.java就是我们要开发的功能
7 >>> git add .
8 >>> git commit -m 'finish discuss feature'
9 # 提交更改,多次测试以后
10 >>> git checkout develop
11 # 回到develop分支
12 >>> git merge feature-discuss
13 # 把做好的功能合并到develop中
14 >>> git branch -d feature-discuss
15 # 删除功能性分支
16 >>> git push origin develop
17 # 把develop提交到自己的远程仓库中
```

此时，上自己gitlab的项目主页中**develop**分支中查看，已经有**discuss.java**这个文件了：

向项目经理提交pull request

## 提交pullrequest请求页面

代码 Issues ① Pull Requests ① Wiki 统计 DevOps 服务 管理

开启的 搜索 pull requests 搜索 重置过滤条件 +新建 Pull Request

所有 开启的 ① 已合并 ① 已关闭 ① 创建者 测试人员 审查人员 标签 里程碑 排序

dev分支添加用户新增功能功能 刚刚  
by 赵登凯 赵登凯/dev → Alin张楠/dev 审查 测试

在完成了“讨论”功能（当然，也可能对自己的develop进行了多次合并，完成了多个功能），经过测试以后，觉得没问题，就可以请求管理员把自己仓库的develop分支合并到源仓库的develop分支中。

### 管理员测试、合并

管理员登陆gitlab，看到了开发者对源仓库发起的pull request。

管理员需要做的事情就是：对开发者的代码进行`review`。

在他的本地测试新建一个测试分支，测试开发者的代码：

```

1 >>> git checkout develop
2 # 进入管理员本地的develop分支
3 >>> git checkout -b manager-develop
4 # 从develop分支中分出一个叫manager-develop的测试分支测试开发者的代码
5 >>> git pull
6 http://gitlab.rd.chanjet.com/op/Chanjet_Asset_Management.git develop
7 # 把开发者的代码pull到测试分支中，进行测试

```

判断是否同意合并到源仓库的develop中，如果经过测试没问题，可以把开发者的代码合并到源仓库的develop中：

### 审核通过

表态: +②  
共0条评论, 1人参与  
Alm张楠 审查通过 刚刚

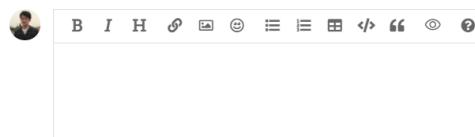
展开详细操作日志

此Pull Request暂时不能合并。当前状态: 未测试 可自动合并  
至少需要1名审查人员审查通过后和至少需要1名测试人员测试通过后才能合并

测试通过

[评论 ①](#) [提交 ①](#) [文件 ①](#)

暂无评论



### 接受请求

源分支与目标分支没有冲突  
可自动合并(如果你仍然想手动合并 - 点击这里 查看怎样操作)

评论 ① 提交 ① 文件 ① 合并

暂无评论

B I H ⌂ 回退 编辑

**已合并** #1 dev分支添加用户新增功能功能

赵登凯:dev → Alm张楠:dev

赵登凯 创建于: 2分钟前 审查 测试

我写了一个用户添加的功能，想提交到dev分支。

表态: +①

共 0 条评论, 1 人参与

Alm张楠 合并了 Pull Request (通过 Web 合并) 刚刚 展开详细操作日志

评论 ① 提交 ① 文件 ①

### 总结:

- 1、自己先fork代码到自己账户
- 2、拉倒本地，写代码
- 3、推到远程仓库
- 4、提交issue
- 5、管理员测试，review后统一，就合并了
- 6、如发生冲突，解决冲突即可

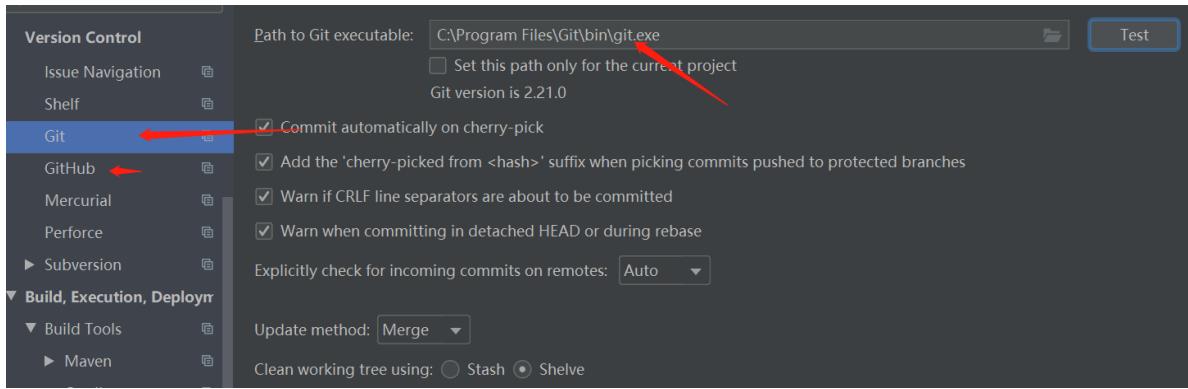
## 4、重点

- 不要随便动别人的代码，即使要动也要商量！
- 不要随便动别人的代码，即使要动也要商量！
- 不要随便动别人的代码，即使要动也要商量！
- 记住一点，写代码和提交之前先拉去最新的代码！必须记住！
- 记住一点，写代码和提交之前先拉去最新的代码！必须记住！
- 记住一点，写代码和提交之前先拉去最新的代码！必须记住！

能够很大程度的避免冲突！

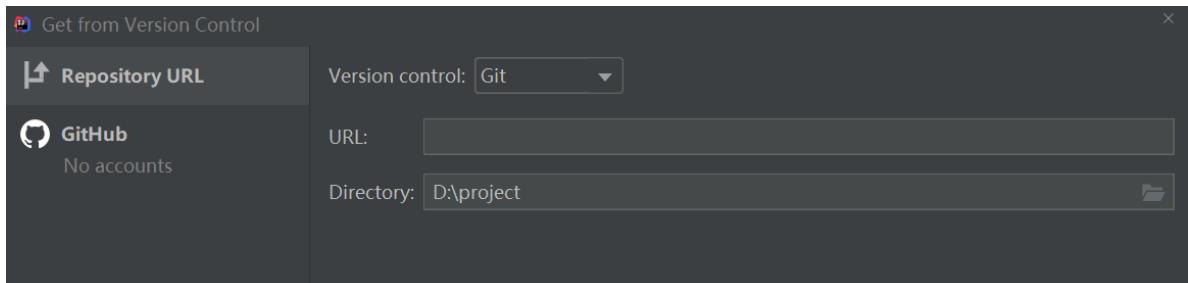
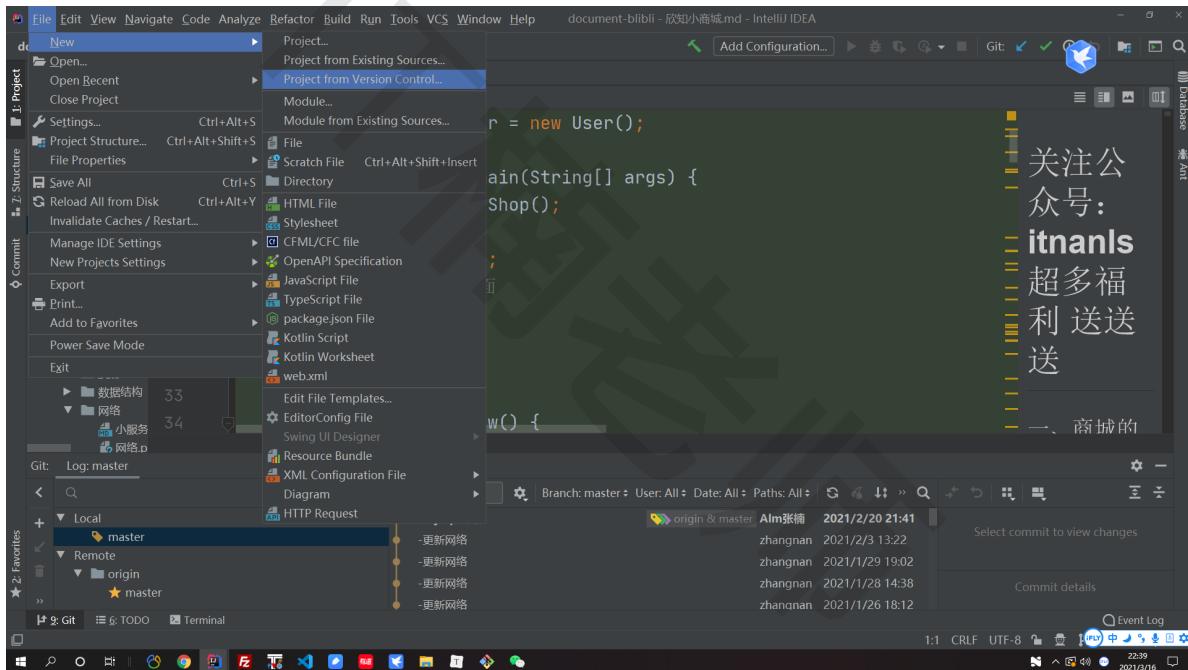
# 第五章 idea使用git

## 配置

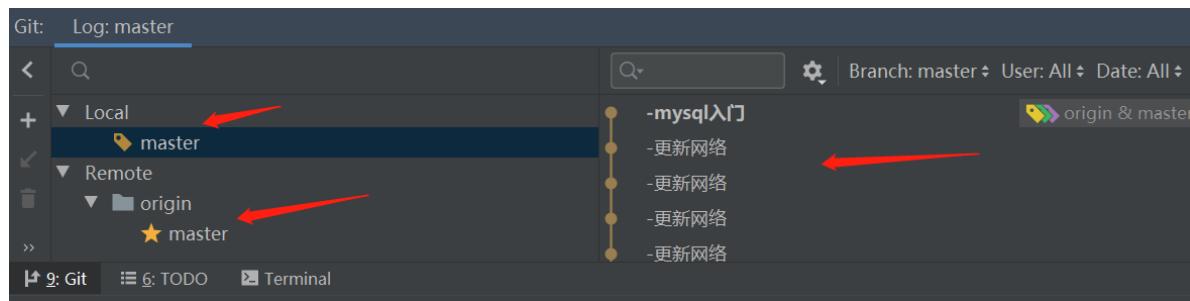


可以下载码云插件 gitee

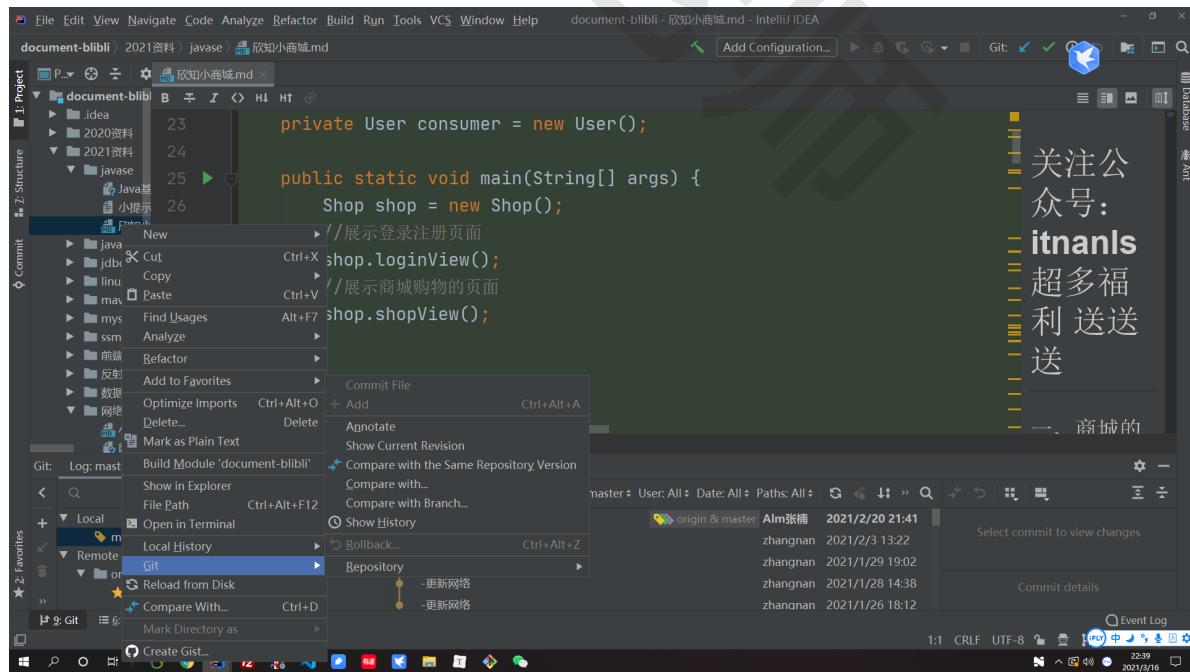
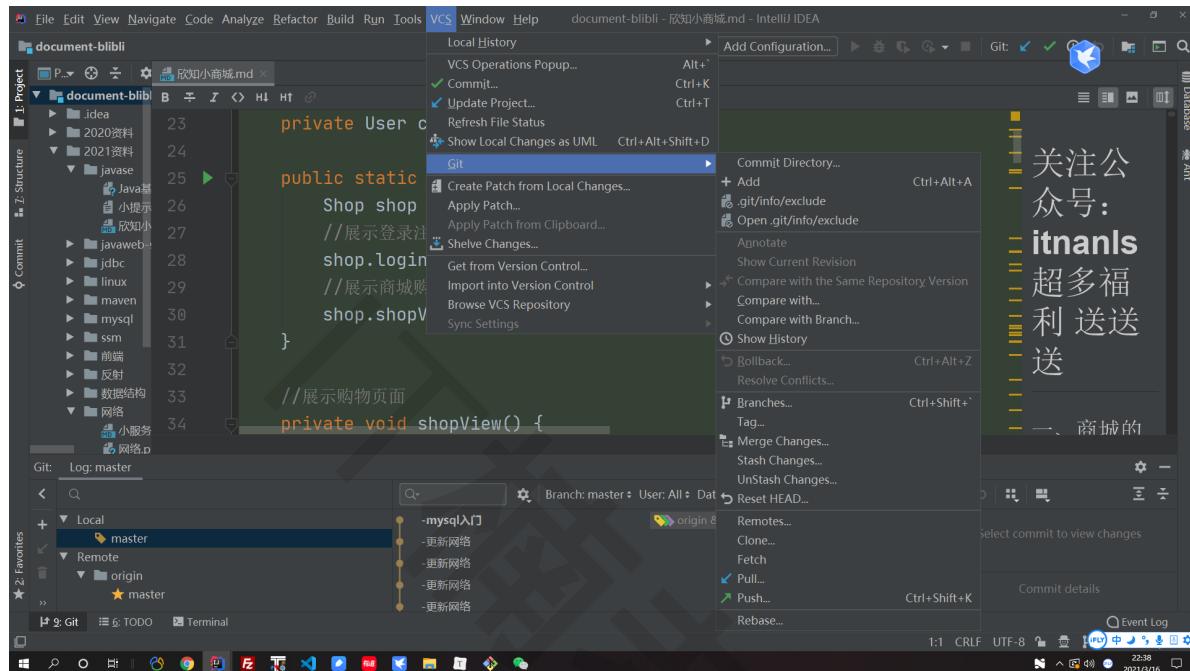
## 从远程仓库拉项目



控制台查看分支提交等信息



## 提交代码



IT楠老用