目 录

[## Git第一轮学习 3](#_Toc508868525)

[**1.1** **果壳中的git** 3](#_Toc508868526)

[1.1.1什么是Git 3](#_Toc508868527)

[1.1.2性能 3](#_Toc508868528)

[1.1.3安全性 3](#_Toc508868529)

[1.1.4柔软性 3](#_Toc508868530)

[1.1.5使用Git进行版本管理 4](#_Toc508868531)

[**2.1快速指南** 5](#_Toc508868532)

[2.1.1安装Git 5](#_Toc508868533)

[2.1.2检出仓库 6](#_Toc508868534)

[2.1.3工作流 6](#_Toc508868535)

[2.1.4添加与提交 6](#_Toc508868536)

[2.1.5推送改动 6](#_Toc508868537)

[**2.2创建代码仓库** 6](#_Toc508868538)

[2.2.1 git init 7](#_Toc508868539)

[2.2.2 git clone 8](#_Toc508868540)

[2.2.3 git config 8](#_Toc508868541)

[**2.3保存更改** 9](#_Toc508868542)

[2.3.1 git add 9](#_Toc508868543)

[2.3.2 git commit 9](#_Toc508868544)

[**2.4检查仓库状态** 10](#_Toc508868545)

[2.4.1 git status 10](#_Toc508868546)

[2.4.2 git log 10](#_Toc508868547)

[**2.5检出之前的提交** 11](#_Toc508868548)

[2.5.1 git checkout 11](#_Toc508868549)

[**2.6回滚错误的修改** 12](#_Toc508868550)

[2.6.1 git revert 12](#_Toc508868551)

[2.6.2 git reset 12](#_Toc508868552)

[2.6.3 git clean 14](#_Toc508868553)

[**2.7重写项目历史** 14](#_Toc508868554)

[2.7.1 git commit –amend 14](#_Toc508868555)

[2.7.3 git rebase -i 15](#_Toc508868556)

[2.7.4 git reflog 16](#_Toc508868557)

[**3.2保持同步** 16](#_Toc508868558)

[3.2.1 git remote 16](#_Toc508868559)

[3.2.2 git fetch 17](#_Toc508868560)

[3.2.3 git pull 17](#_Toc508868561)

[3.2.4 git push 18](#_Toc508868562)

[3.4.1 git branch 20](#_Toc508868563)

[3.4.2 git checkout 20](#_Toc508868564)

[3.4.3 git merge 21](#_Toc508868565)

[3.4.4快速向前合并 21](#_Toc508868566)

[3.4.5三路合并 21](#_Toc508868567)

[3.5.1中心化的工作流 21](#_Toc508868568)

[3.5.2给予功能分支的工作流（Feature分支的工作流） 22](#_Toc508868569)

[3.5.3 Gitflow工作流 23](#_Toc508868570)

[3.5.4 Fork工作流 23](#_Toc508868571)

[**4.1图解Git命令** 23](#_Toc508868572)

[**5.1代码合并：Merge、Rebase的选择** 24](#_Toc508868573)

[5.1.1 Merge 24](#_Toc508868574)

[5.1.2 Rebase 24](#_Toc508868575)

[5.1.3交互式的rebase 24](#_Toc508868576)

[5.1.4 Rebase黄金法则：绝对不要再公共的分支上使用它 25](#_Toc508868577)

[**1集中式VS分布式** 25](#_Toc508868578)

[**2创建版本库** 25](#_Toc508868579)

[**2.1把文件添加到版本库** 25](#_Toc508868580)

[**3时光穿梭机** 26](#_Toc508868581)

[**3.1版本回退** 28](#_Toc508868582)

[**3.2工作区和暂存区** 29](#_Toc508868583)

[**3.3管理修改** 31](#_Toc508868584)

[**3.4撤销修改** 31](#_Toc508868585)

[**3.5删除文件** 32](#_Toc508868586)

[**4远程仓库** 32](#_Toc508868587)

[**4.1添加远程库** 32](#_Toc508868588)

[**4.2从远程库克隆** 34](#_Toc508868589)

[**5分支管理** 34](#_Toc508868590)

[**5.1创建与合并分支** 35](#_Toc508868591)

[**5.2解决冲突** 37](#_Toc508868592)

[**5.3分支管理策略** 38](#_Toc508868593)

[**5.4 Bug分支** 39](#_Toc508868594)

[**5.5 Feature分支** 39](#_Toc508868595)

[**5.6多人协作** 39](#_Toc508868596)

[**6标签管理** 41](#_Toc508868597)

[**6.1创建标签** 41](#_Toc508868598)

[**6.2操作标签** 41](#_Toc508868599)

[**7使用GitHub** 42](#_Toc508868600)

[**8使用码云** 42](#_Toc508868601)

[**9自定义Git** 42](#_Toc508868602)

[**10总结** 42](#_Toc508868603)

# Git笔记

## Git第一轮学习

以下笔记主要来源：https://github.com/geeeeeeeeek/git-recipes/wiki

2018-3-7

* 1. **果壳中的git**

### 1.1.1什么是Git

Git是目前世界上被最广泛使用的现代软件版本管理系统，Git本身也是一个成熟并处于活跃开发状态的开源项目。

### 1.1.2性能

Git在决定代码修改历史以及保存形式的时候不会被文件名的变化所愚弄，Git关注的是文件的内容本身。Git使用一种混合了差分编码（delta encoding, 仅保存代码修改的差分）、压缩、直接保存、以及版本元数据（version metadata objects）的管理方式。

分散式的架构也给 Git 带来了极大的性能优势。

### 1.1.3安全性

Git将保持所管理代码的整合性作为首要要务。加密哈希校验算法（SHA1）

### 1.1.4柔软性

Git的关键设计目标之一就是保持柔软性。支持各种非线性的开发工作流程，对或大或小的软件项目都可以良好支持，以及兼容各种操作系统和协议。

Git支持将分支和标签作为一级基本对象（不同于SVN），所以多有对分支和标签的操作也都会被保存到修改历史中。

### 1.1.5使用Git进行版本管理

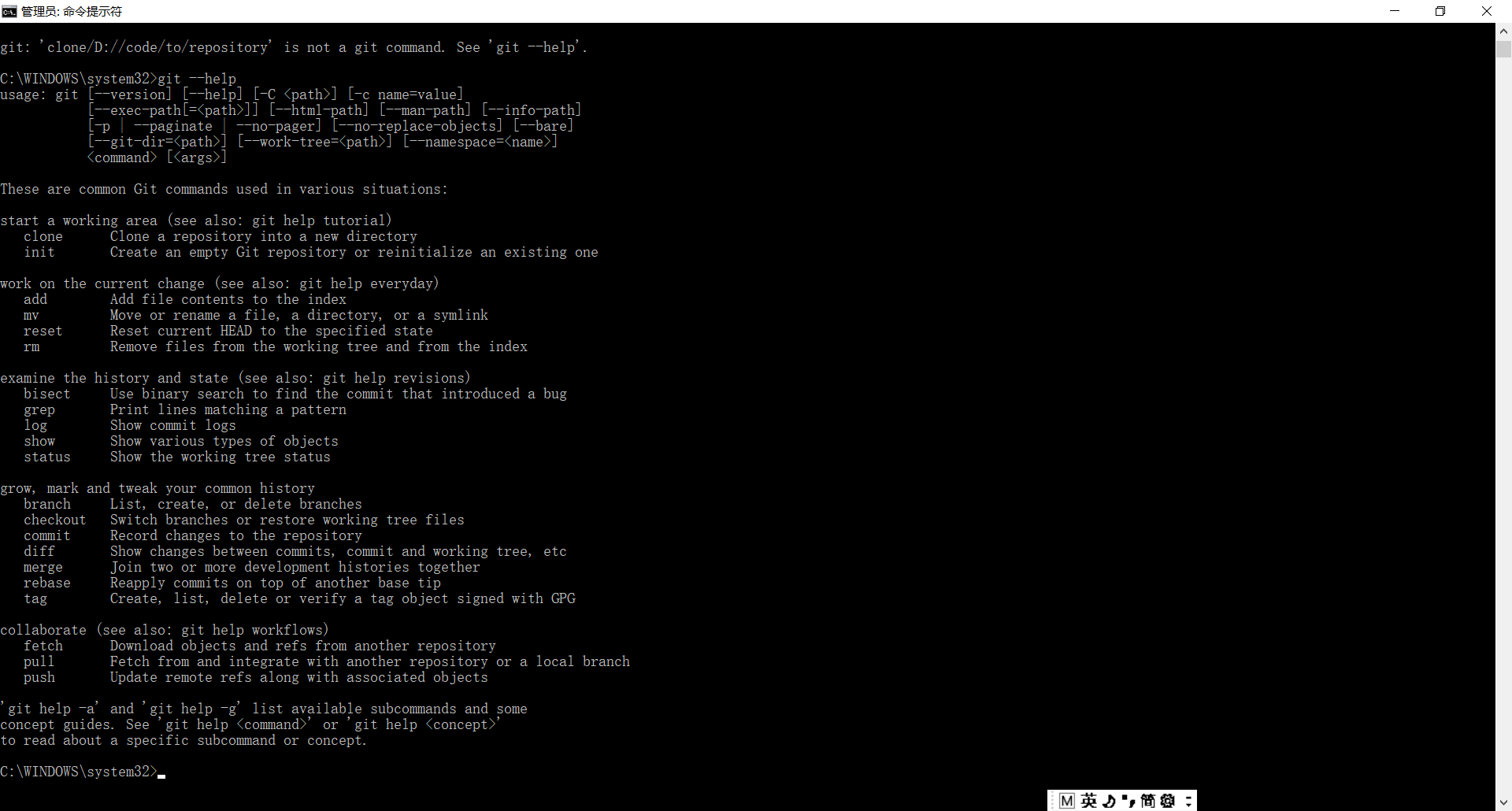
(1) Git很棒

兼备了功能性、高性能、安全性和柔软性

(2) Git已经成为了默认的行业标准

(3) Git是一个高质量的开源项目

2018-3-8



**2.1快速指南**

### 2.1.1安装Git

Windows用户：下载Git SCM，安装后如果希望在全局的cmd中使用git，需要把git.exe加入PATH环境变量中，或在Git Bash中使用Git

### 2.1.3工作流

### 2.1.4添加与提交

（1）可以计划改动（把它们添加到缓存区），使用如下命令：

Git add <filename>

Git add \*

（2）使用如下命令以实际提交改动

Git commit -m “代码提交信息”

### 2.1.5推送改动

（1）你的改动现在已经在本地仓库的 HEAD 中了。执行如下命令以将这些改动提交到远端仓库：

Git push origin master （可以把master换成想要推送的任何分支）

（2）如果你还没有克隆现有仓库，并欲将你的仓库连接到某个远程服务器，你可以使用如下命令添加：

Git remote add origin <server>

这样就可以将改动推送到所添加的服务器上去。

### 2.1.2检出仓库

（1）创建一个本地仓库的克隆版本：git clone/path/to/repository

（2）如果是远端服务器上的仓库：git clone username@host:/path/to/repository（通过SSH）或者git clone https:/path/to/repository.git（通过https）

比如说 git clone https://github.com/geeeeeeeeek/git-recipes.git 可以将 git clone 到你指定的目录。

**2.2创建代码仓库**

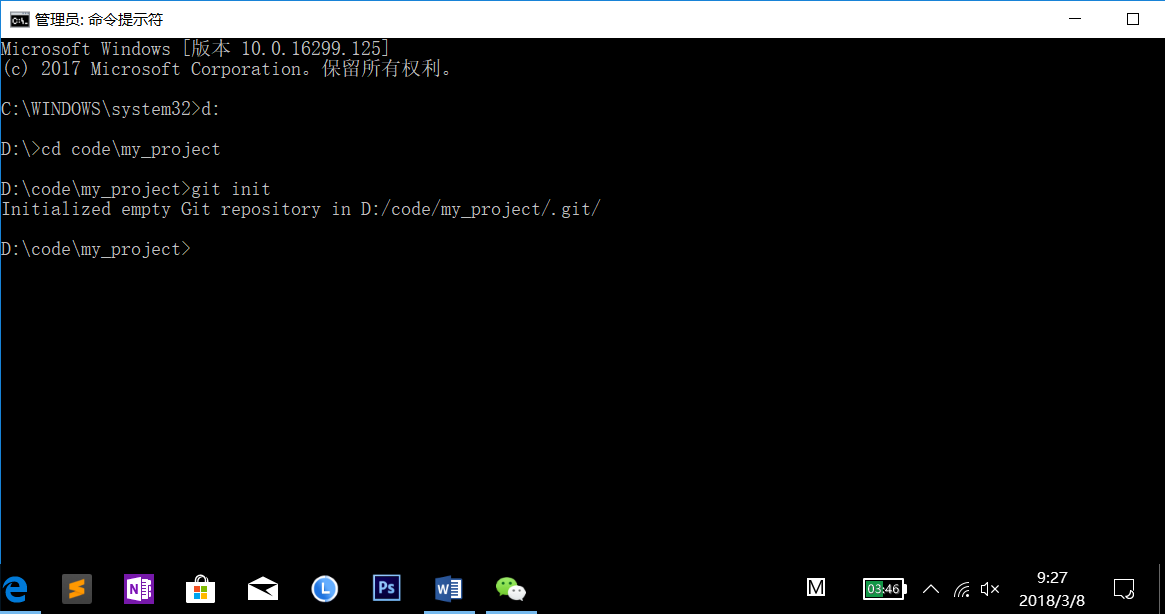
### 2.2.1 git init

创建一个新的Git仓库，它用来将已存在但还没有版本控制的项目转换成一个Git仓库，或者创建一个空的新仓库。大多数Git命令在未初始化的仓库中都是无法使用的，所以这就是你运行新项目的第一个命令了。

运行git init会在项目的根目录下创建一个新的.git目录，其中包含了项目必需的所有元数据。

1. 用法

git init 将当前的目录转换成一个Git仓库。它在当前的目录下增加一个.git目录，于是就可以开始记录项目版本了。



Git init <directory> 在指定目录创键一个空的Git仓库，会创建一个名为directory，只包含.git子目录的空目录。

Git init –bare <directory>

（2）首先，你用SSH连入存放中央仓库的服务器。然后，来到任何你想存放项目的地方，最后，使用 -—bare 标记来创建一个中央存储仓库。开发者会将 my-project.git 克隆到本地的开发环境中。

Ssh <user>@<host>

Cd path/above/repo

Git init –bare my-project.git

### 2.2.2 git clone

拷贝整个Git仓库，除了【工作副本】是一个完备的Git仓库——它包含自己的历史，管理自己的文件，以及环境和原仓库完全隔离。

1. 用法

git clone <repo> 将位于<repo>的仓库克隆到本地机器。原仓库可以在本地文件系统中，或是通过HTTP或SSH连接的远程机器

git clone <repo> <directory> 将位于<repo>的仓库克隆到本地机器上的<directory>目录

1. git clone是用户获取开发副本最常见的方式。只要开发者获得了一份工作副本，所有版本控制操作和协作管理都是在本地仓库中完成。
2. 用SSH用户名john连接到example.com，获取远程服务器上中央仓库的本地副本

git clone ssh://john@example.com/path/to/my-project.git

cd my-project

# 开始工作

### 2.2.3 git config

允许你在命令中配置你的Git安装，这个命令定义了所有配置，从用户信息到仓库行为等等。

# 告诉Git你是谁

git config --global user.name "John Smith"

git config --global user.email john@example.com

# 选择你喜欢的文本编辑器

git config --global core.editor vim

# 添加一些快捷方式(别名)

git config --global alias.st status

git config --global alias.co checkout

git config --global alias.br branch

git config --global alias.up rebase

git config --global alias.ci commit

**2.3保存更改**

### 2.3.1 git add

将工作目录中的变化添加到缓存区。它告诉Git你想要在下一次提交时包含这个文件的更新。在运行git commit前更改都不会被记录。使用git status来查看工作目录和缓存区的状态。

1. 用法

git add <file> 将<file>中的更改加入下次提交的缓存

git add <directory> 将<directory>下的更改加入下次提交的缓存

git add -p 开始交互式的缓存（y：将这块更改加入缓存; n：忽略这块更改; s：分割成更小的块; e：手动编辑这块更改; q：退出）

1. 缓存区

你可以更改各种不相关的文件，然后会过去将他们按逻辑切分，将相关的更改添加到缓存，一份一份提交。在任何修改控制系统中，提交必须时原子性的，以便追踪bug，并用最小的代价回滚更改。

### 2.3.2 git commit

将缓存的快照提交到项目历史。

（1）用法

Git commit他会运行文本编辑器，等待你输入提交信息。当输入信息后，保存文件，关闭编辑器，创建实际的提交

Git commit -m “<messae>”提交已经缓存的快照。但将<message>作为提交信息，而不是运行文本编辑器

Git commit -a提交一份包含工作目录所有更改的快照。它只包含跟踪过的文件的更改

（2）eg

它会打开一个文件编辑器（可以通过 git config 设置) 询问提交信息，同时列出将被提交的文件。

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

#modified: hello.py

Git 对提交信息没有特定的格式限制，但约定俗成的格式是：在第一行用 50 个以内的字符总结这个提交，留一空行，然后详细阐述具体的更改。比如：

Change the message displayed by hello.py

- Update the sayHello() function to output the user's name

- Change the sayGoodbye() function to a friendlier message

**2.4检查仓库状态**

### 2.4.1 git status

显示工作目录和缓存区的状态。

如果想要看任何已提交到项目历史的信息，就应该使用git log命令

Git status列出已缓存、未缓存、未追踪的文件

### 2.4.2 git log

显示已提交的快照。可以列出项目历史，筛选，以及搜索特定更改

Take care: 与git status的区别

1. 用法

Git log 使用默认格式显示完整地项目历史。如果输出超过一屏，你可以用 空格键 来滚动，按 q 退出

Git log -n <limit> 用 <limit> 限制提交的数量。比如 git log -n 3 只会显示 3 个提交。

Git log –oneline 将每个提交压缩到一行。当你需要查看项目历史的上层情况时这会很有用。

Git log –stat 除了 git log 信息之外，包含哪些文件被更改了，以及每个文件相对的增删行数。

Git log -p 显示代表每个提交的一堆信息。显示每个提交全部的差异（diff），这也是项目历史中最详细的视图。

Git log –author = “<pattern>” 搜索特定作者的提交。<pattern> 可以是字符串或正则表达式。

Git log –grep = “<pattern>” 搜索提交信息匹配特定 <pattern> 的提交。<pattern> 可以是字符串或正则表达式。

Git log <since>..<util> 只显示发生在 <since> 和 <until> 之间的提交。两个参数可以是提交 ID、分支名、HEAD 或是任何一种引用。

Git log <file> 只显示包含特定文件的提交。查找特定文件的历史这样做会很方便。

Git log –graph –decorate –online --graph 标记会绘制一幅字符组成的图形，左边是提交，右边是提交信息。--decorate 标记会加上提交所在的分支名称和标签。--oneline 标记将提交信息显示在同一行，一目了然。

2018-3-9

**2.5检出之前的提交**

### 2.5.1 git checkout

（1）三个作用：检出文件，检出提交，检出分支

（2）用法

Git checkout master

回到项目[当前]状态的一种方式

一旦回到master分支之后，你可以使用git rever或git reset来回滚任何不想要的更改

Git checkout <commit><file>

查看文件之前的版本，它将工作目录中的<file>变成<commit>中那个文件的拷贝，并将它加入缓存区

Git checkout <commit>

更新工作目录中的所有文件，使得和某个特定提交中的文件一致。你可以将提交的哈希字串，或是标签作为<commit>参数。这会使你处在分离HEAD的状态

**2.6回滚错误的修改**

### 2.6.1 git revert

撤销一个已经提交的快照。但是，它是通过搞清楚如何撤销这个提交引入的更改，然后在最后加上一个撤销了更改的 新 提交，而不是从项目历史中移除这个提交。这避免了Git丢失项目历史。

（1）用法：git revert <commit> 生成一个撤销了<commit>引入的修改的新提交，然后应用到当前分支

（2）eg

# 编辑一些跟踪的文件

# 提交一份快照

git commit -m "Make some changes that will be undone"

# 撤销刚刚的提交

git revert HEAD

### 2.6.2 git reset

相对git revert来说，git reset是一个危险的方式。这个撤销是永远的。Be careful to use this tool.

1. 用法：

Git reset <file> 从缓存区移除特定文件，但不改变工作目录。它会取消这个文件的缓存，而不覆盖任何更改。

Git reset 重设缓冲区，匹配最近的一次提交，但工作目录不变。它会取消 所有 文件的缓存，而不会覆盖任何修改，给你了一个重设缓存快照的机会。

Git reset –hard 重设缓冲区和工作目录，匹配最近一次提交。除了取消缓存之外，--hard 标记告诉 Git 还要重写所有工作目录中的更改。换句话说：它清除了所有未提交的更改，所以在使用前确定你想扔掉你所有本地的开发。

Git reset <commit> 将当前分支的末端移到 <commit>，将缓存区重设到这个提交，但不改变工作目录。所有 <commit>之后的更改会保留在工作目录中，这允许你用更干净、原子性的快照重新提交项目历史。

Git reset –hard <commit> 将当前分支的末端移到 <commit>，将缓存区和工作目录都重设到这个提交。它不仅清除了未提交的更改，同时还清除了 <commit> 之后的所有提交。

**取消文件缓存**

# 编辑了hello.py和main.py

# 缓存了目录下所有文件

git add .

# 意识到hello.py和main.py中的修改

# 应该在不同的快照中提交

# 取消main.py缓存

git reset main.py

# 只提交hello.py

git commit -m "Make some changes to hello.py"

# 在另一份快照中提交main.py

git add main.py

git commit -m "Edit main.py"

**移除本地修改**

# 创建一个叫`foo.py`的新文件，增加代码

# 提交到项目历史

git add foo.py

git commit -m "Start developing a crazy feature"

# 再次编辑`foo.py`，修改其他文件

# 提交另一份快照

git commit -a -m "Continue my crazy feature"

# 决定废弃这个功能，并删除相关的更改，将当前分支向前倒退两个提交，这种重设只能用在 非公开 的提交中

git reset --hard HEAD~2

### 2.6.3 git clean

将未跟踪的文件从你的工作目录中移除。和一般的rm命令一样，git clean是无法撤销的。

经常与git reset –hard一起使用。reset只影响被跟踪的文件，clean影响未跟踪的文件。这两个命令结合，可以将工作目录回到之前特定提交时的状态。

**(1)用法：**

Git clean -n 执行一次git clean的『演习』。它会告诉你那些文件在命令执行后会被移除，而不是真的删除它。

Git clean -f 移除当前目录下未被跟踪的文件。-f（强制）标记是必需的，除非 clean.requireForce 配置项被设为了 false（默认为 true）。它 不会 删除 .gitignore 中指定的未跟踪的文件。

Git clean -f <path> 移除未跟踪的文件，但限制在某个路径下。

Git clean -df 移除未跟踪的文件，以及目录。

Git clean -xf 移除当前目录下未跟踪的文件，以及 Git 一般忽略的文件。

**(2)eg**

# 编辑了一些文件

# 新增了一些文件

# 『糟糕』

# 将跟踪的文件回滚回去

git reset --hard

# 移除未跟踪的文件

git clean -df

**2.7重写项目历史**

### 2.7.1 git commit –amend

是修复最新提交的便捷方式。它允许你将缓存的修改和之前的提交合并到一起，而不是提交一个全新的快照。它还可以用来简单地编辑上一次提交地信息而不改变快照。

# 编辑 hello.py 和 main.py

git add hello.py

git commit

# 意识到你忘记添加 main.py 的更改

git add main.py

git commit --amend --no-edit

2.7.2 git rebase

变基是将分支移到一个新的基提交地过程

**（1）用法**：git rebase <base> 将当前分支rebase到<base>,这里可以是任何类型的提交引用（ID、分支名、标签、或是HEAD的相对引用）

（2）不要rebase公共历史

### 2.7.3 git rebase -i

用 -i 标记运行 git rebase 开始交互式 rebase。交互式 rebase 给你在过程中修改单个提交的机会，而不是盲目地将所有提交都移到新的基上。你可以移除、分割提交，更改提交的顺序。它就像是打了鸡血的 git commit --amend 一样。

**(1)用法：**git rebase -i <base> 将当前分支 rebase 到 base，但使用可交互的形式

**(2)Eg:**

# 开始新的功能分支

git checkout -b new-feature master

# 编辑文件

git commit -a -m "Start developing a feature"

# 编辑更多文件

git commit -a -m "Fix something from the previous commit"

# 直接在 master 上添加文件

git checkout master

# 编辑文件

git commit -a -m "Fix security hole"

# 开始交互式 rebase

git checkout new-feature

git rebase -i master

### 2.7.4 git reflog

Git 用引用日志这种机制来记录分支顶端的更新。它允许你回到那些不被任何分支或标签引用的更改。在重写历史后，引用日志包含了分支旧状态的信息，有需要的话你可以回到这个状态。

1. 用法

Git reflog 显示本地仓库的引用日志

Git reflog --relative-data 用相对的日期显示引用日志

1. 每次当前的 HEAD 更新时（如切换分支、拉取新更改、重写历史或只是添加新的提交），引用日志都会添加一个新条目。

**3.2保持同步**

### 3.2.1 git remote

创建、查看和删除和其他仓库之间的连接

**(1)用法：**

Git remote 列出你和其他仓库之间的远程连接。

Git remote -v 和上个命令相同，但同时显示每个连接的 URL。

Git remote add <name> <url> 创建一个新的远程仓库连接。在添加之后，你可以将 <name> 作为 <url> 便捷的别名在其他 Git 命令中使用。

Git remote rm <name> 移除名为<name>的远程仓库的连接。

Git remote rename <old-name> <new-name> 将远程连接从 <old-name> 重命名为 <new-name>。

(2)仓库的URL

Git支持多种方式来引用一个远程仓库。其中两种最简单的方式便是HTTP和SSH协议。HTTP允许匿名、只读访问仓库，like: http://host/path/to/repo.git

如果希望对仓库进行读写，你需要使用 SSH 协议：ssh://user@host/path/to/repo.git

### 3.2.2 git fetch

将提交从远程仓库导入到你的本地仓库。拉取下来的提交储存为远程分支，而不是我们一直使用的普通的本地分支。你因此可以在整合进你的项目副本之前查看更改。

1. 用法

Git fetch <remote> 拉取仓库中所有的分支。同时会从另一个仓库中下载所有需要的提交和文件。

Git fetch <remote> <branch> 只拉取指定的分支

1. 远程分支

远程分支和本地分支一样，只不过它们代表这些提交来自于其他人的仓库。你可以像查看本地分支一样查看远程分支，但你会处于分离 HEAD 状态（就像查看旧的提交时一样）。你可以把它们视作只读的分支。如果想要查看远程分支，只需要向 git branch 命令传入 -r 参数。远程分支拥有 remote 的前缀，所以你不会将它们和本地分支混起来。

1. 可以用git checkout和git log命令来查看这些分支
2. 同步本地仓库和远程仓库事实上是以恶分两步的操作：先fetch，然后merge。Git pull命令是这个过程的快捷方式

### 3.2.3 git pull

在基于 Git 的协作工作流中，将上游更改合并到你的本地仓库是一个常见的工作。我们已经知道应该使用 git fetch，然后是 git merge，但是 git pull 将这两个命令合二为一。

1. 用法

Git pull <remote> 拉取当前分支对应的远程副本中的更改，并立即并入本地副本。效果和 git fetch 后接 git merge origin/. 一致

Git pull -rebase <remote> 和上一个命令相同，但使用 git rebase 合并远程分支和本地分支，而不是使用 git merge。

1. Eg

git checkout master

git pull --rebase origin

### 3.2.4 git push

Push 是你将本地仓库中的提交转移到远程仓库中时要做的事。它和 git fetch 正好相反，fetch 将提交导入到本地分支，而 push 将提交导出到远程分支。它可以覆盖已有的更改，所以你需要小心使用。

1. 用法

Git push <remote> <branch>

将指定的分支推送到 <remote> 上，包括所有需要的提交和提交对象。它会在目标仓库中创建一个本地分支。为了防止你覆盖已有的提交，如果会导致目标仓库非快速向前合并时，Git 不允许你 push。

Git push <remote> --force

和上一个命令相同，但即使会导致非快速向前合并也强制推送。除非你确定你所做的事，否则不要使用 --force 标记。

git push <remote> --all

将所有本地分支推送到指定的远程仓库。

git push <remote> --tags

当你推送一个分支或是使用 --all 选项时，标签不会被自动推送上去。--tags 将你所有的本地标签推送到远程仓库中去。

3.3创建Pull Reauest



3.4使用分支

### 3.4.1 git branch

分支代表了一条独立的开发流水线。该命令允许创建、列出、重命名和删除分支。它不允许切换分支或是被fork的历史放回去。因此，git branch和git checkout、git merge这两个命令通常紧密地结合在一起使用。

1. 用法

Git branch 列出仓库中所有分支

Git branch <branch> 创建一个名为<branch>的分支。不会自动切换到那个分支去。

Git branch -d <branch> 删除指定分支。这是一个安全的操作，git会阻止你删除包含未合并更改的分支

Git branch -D <branch> 强制删除指定分支，即使包含未合并更改。这将永远删除某条开发线的所有提交

Git branch -m <branch> 将当前分支命名为<branch>

1. take care: 当想要添加一个新的功能或是修复一个bug时，不管bug是大是小，你都应该新建一个分支来封装你的修改。这确保了不稳定的代码永远不会被提交到主代码库中，它同时给了你机会，在并入主分支前清理你 feature 分支的历史。

### 3.4.2 git checkout

允许你切换到用git branch创建的分支

（1）用法：

Git checkout <existing -branch> 查看特定分支

Git checkout -b <new -branch> 创建并查看<new -branch>，-b选项是一个方便的标记，告诉git在运行git checkout <new -branch>之前运行git branch <new -branch>

Git checkout -b <new -branch> <existing -branch> 和上一条相同，但将<existing -branch>作为新分支的基，而不是当前分支。

（1）eg

git branch new-feature

git checkout new-feature

### 3.4.3 git merge

合并是Git将fork的历史放回到一起的方式。Git merge命令允许你将git branch创建的多条分支合并成一个。

用法：

Git merge <branch> 将指定分支并入当前分支。Git会决定使用哪种合并算法

Git merge –no-ff <branch> 将指定分支并入当前分支，但总是生成一个合并提交（即使是快速向前合并）。这可以用来记录仓库中发生的所有合并。

### 3.4.4快速向前合并

### 3.4.5三路合并

2018-3-10

3.5常见工作流比较

### 3.5.1中心化的工作流

（1）一人初始化中央仓库

ssh user@host git init --bare /path/to/repo.git

（2）所有人将仓库克隆到本地

git clone ssh://user@host/path/to/repo.git

1. John在开发他的功能

git status # 查看仓库状态

git add <some-file> # 缓存一个文件

git commit # 提交一个文件</some-file>

（4）Mary在开发她的功能

同时，Mary 在她自己的本地仓库用相同的编辑/缓存/提交流程开发她的功能。和 John 一样，她不需要关心中央仓库的进展，她也 完全 不关心 John 在他自己仓库中做的事，因为所有本地仓库都是私有的。

（5）John发布了他的功能

git push origin master

origin 是 John 克隆中央仓库时指向它的远程连接。master 参数告诉 Git 试着将 origin的 master 分支变得和他本地的 master 分支一样。中央仓库在 John 克隆之后还没有进展，因此这个推送如他所愿，没有产生冲突。

（6）

（7）

### 3.5.2给予功能分支的工作流（Feature分支的工作流）

（1）Mary开始了一个新功能，创建一个新分支

Git checkout -b marys-feature master

一个基于master、名为marys-feature的分支将会被checkout，-b标记告诉Git在分支不存在时创建它。

（2）Mery将她的分支推送到中央仓库，这是一种方便的备份，但如果Mary和其他开发者一起协作，他们也可以看到她的初始提交了。

Git push -u origin marys-feature

将marys-feature推送到中央仓库（origin），-u将它添加为远程跟踪的分支。之后mary调用不带任何参数的git push来推送她的功能。

（3）Mary完成了她的工作

Git push 首先，确保中央仓库包含了她最新的提交

发起一个Pull Request，请求将marys-feature合并进master，

（4）bill收到了Pull Request，决定做一些小修改

（5）Mary作了修改

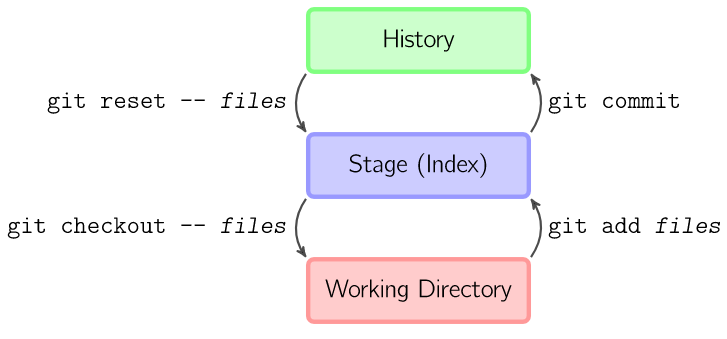
（6）Mary发布了她的功能

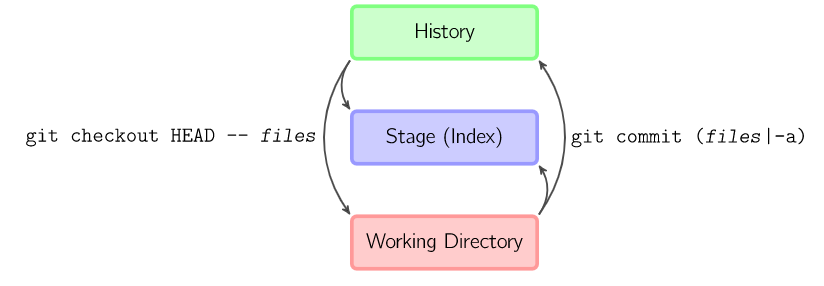
（7）同时，John以同样的方式工作着

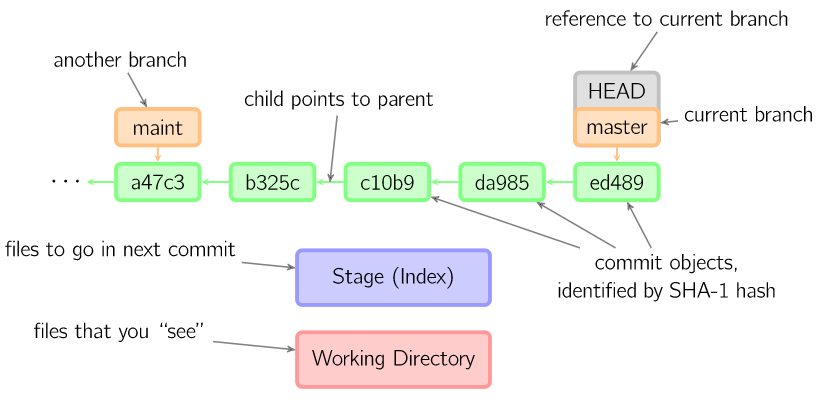
### 3.5.3 Gitflow工作流

### 3.5.4 Fork工作流

**4.1图解Git命令**







**5.1代码合并：Merge、Rebase的选择**

### 5.1.1 Merge

将master分支合并到feature分支最简单的办法就是用下面这些命令：

git checkout feature

git merge master

将上面合并为一行：git mere master feature

### 5.1.2 Rebase

作为merge的替代选择，你可以像下面这样将feature分支并入master分支：

Git checkout feature

Git rebase master

### 5.1.3交互式的rebase

允许你更改并入新分支的提交，提供了分支提交历史完整的控制。

Git checkout feature

Git rebase -i master #开始一个交互式的rebase过程

它会打开一个文本编辑器，显示所有将被移动的提交

### 5.1.4 Rebase黄金法则：绝对不要再公共的分支上使用它

## Git第二轮学习

笔记大部分来源于：https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001373962845513aefd77a99f4145f0a2c7a7ca057e7570000

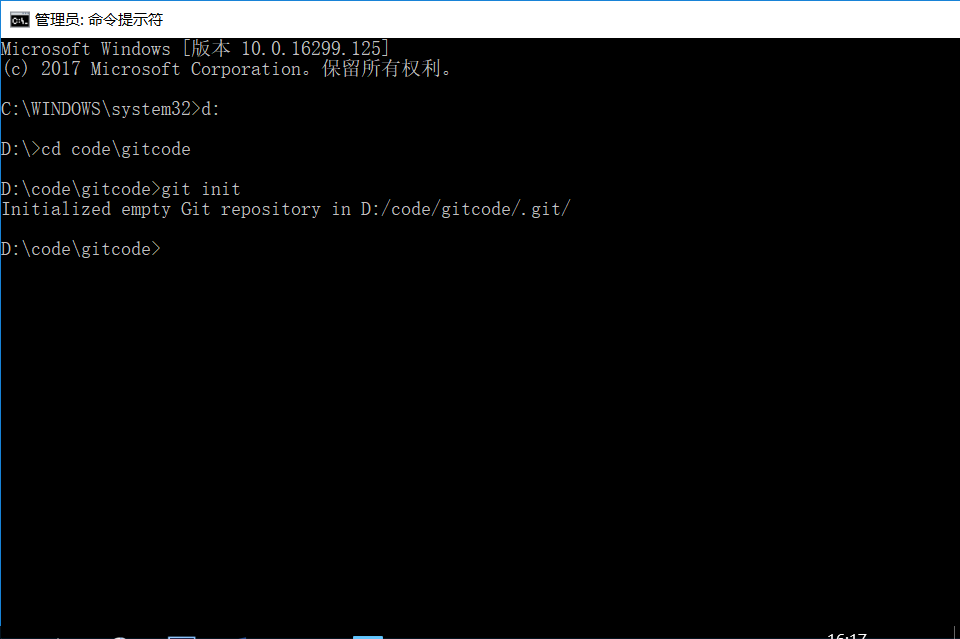
**1集中式VS分布式**

CVS和SVN是集中式的版本控制系统，而Git是分布式版本控制系统

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多

**2创建版本库**

版本库又名仓库，

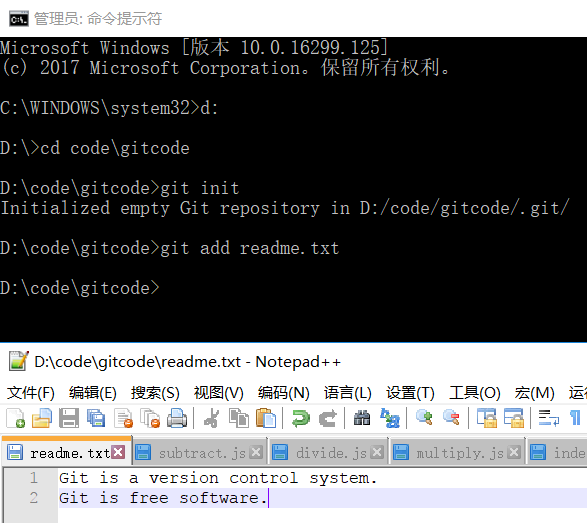




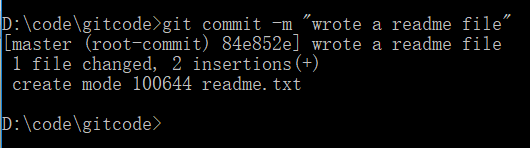
**2.1把文件添加到版本库**

所有的版本控制系统，只能跟踪文本文件的改变。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化。Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的

1. 在gitcode目录下编写readme.txt
2. 用git add告诉Git，把文件添加到仓库

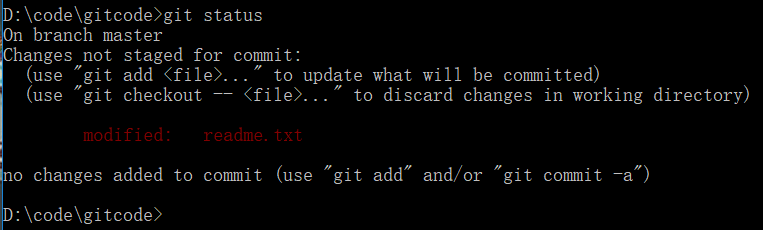


1. 用git commit告诉Git，把文件提交到仓库（-m后面输入的是本次提交的说明）



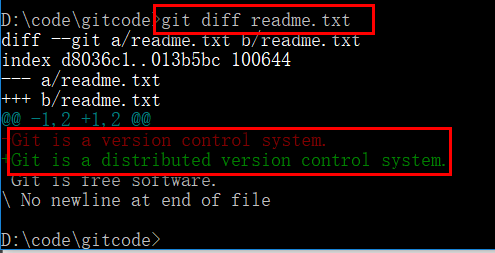
**3时光穿梭机**

（1）修改readme.txt，然后运行git status命令看看结果

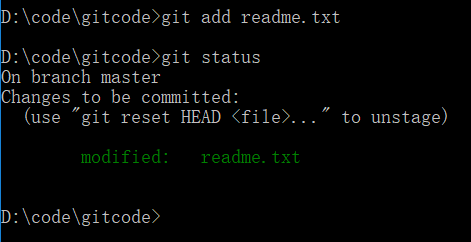


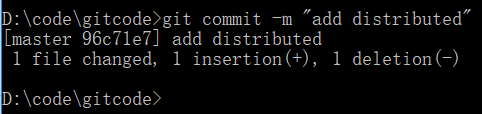
上面的命令告诉我们，readme.txt被修改过了，但是还没有准备提交修改

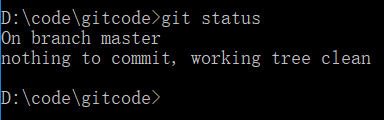
（2）用git diff查看具体修改了什么内容



（3）git add readme.txt git status

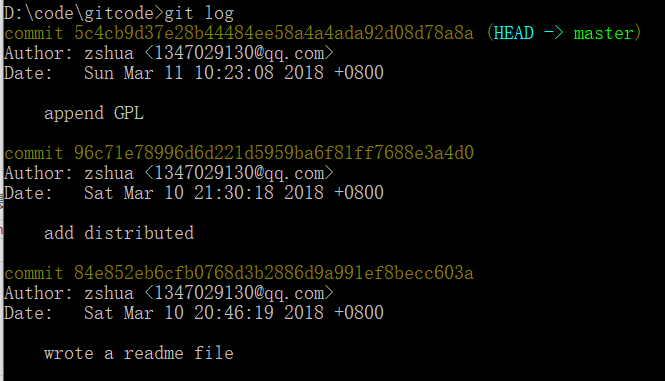






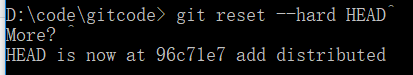
**3.1版本回退**

（1）git log查看历史记录

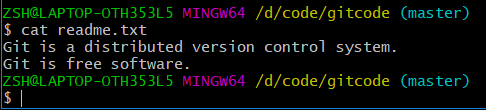


（3）回退

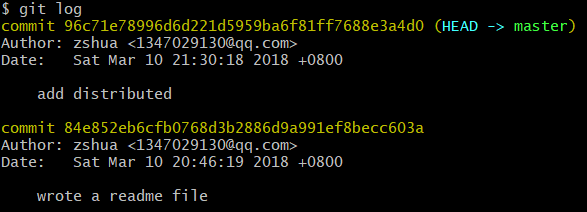
git reset



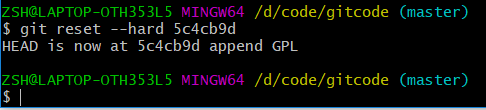
Cat



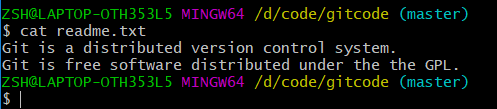
Git log



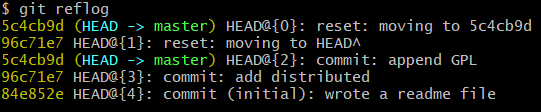
回到最新的那个版本



查看内容是否是之前最新版本的内容



当你想要恢复到最新版本，却找不到新版本的commit id时，可以用git reflog（记录你的每一次命令）



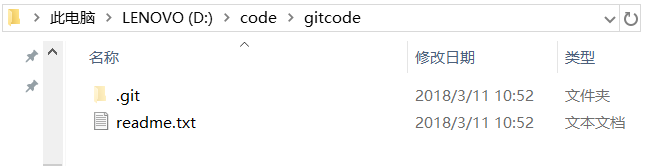
（4）小结

* HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。
* 穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。
* 要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

**3.2工作区和暂存区**

1）工作区（Working Directory）

就是在电脑里能看到的目录，比如gitcode文件夹就是一个工作区



2）版本库（Repository）

（1）工作区有一个隐藏目录.git，这个不算工作去，而是Git的版本库。

（2）版本库里存了很多东西，其中最重要的就是称为stage（index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

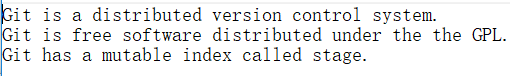
第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

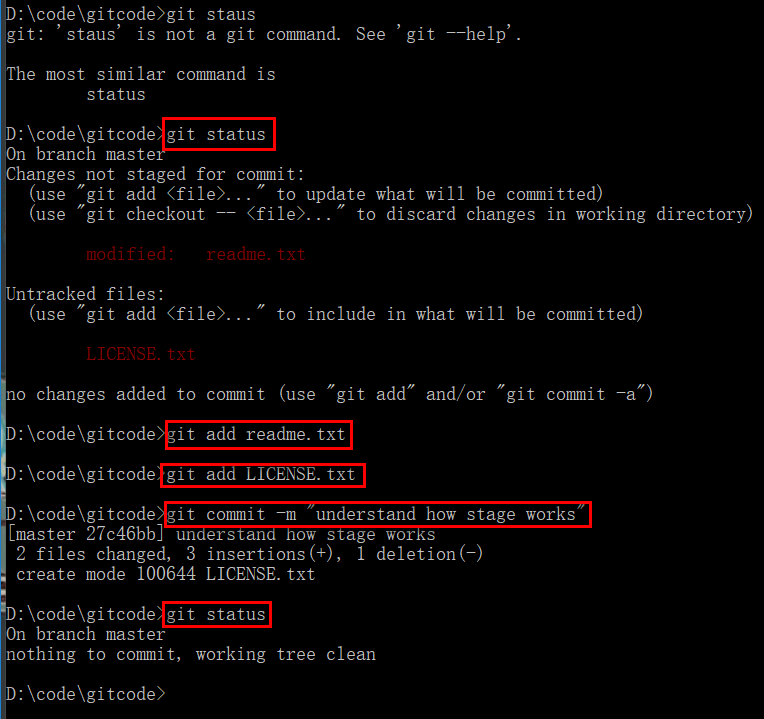
因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

（3）实操

修改readme.txt





**3.3管理修改**

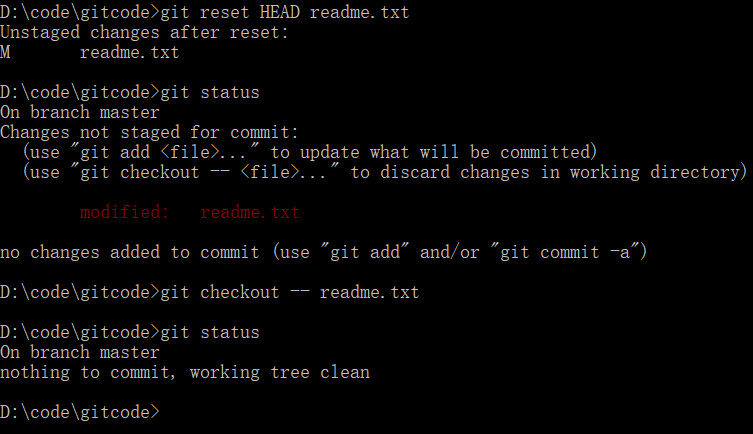
每次修改，如果不add到暂存区，那就不会加入到commit中。

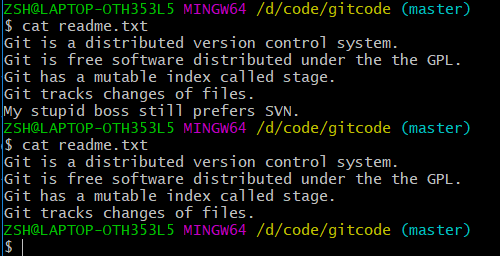
**3.4撤销修改**

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考[版本回退](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000" \t "_blank)一节，不过前提是没有推送到远程库。





**3.5删除文件**

命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。

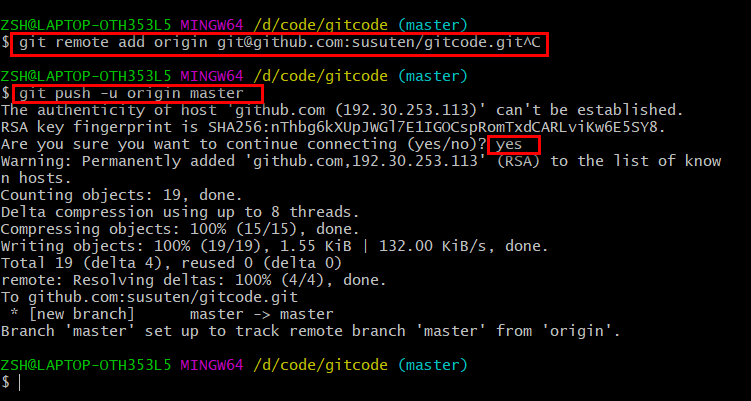
**4远程仓库**

Git是分布式控制版本系统，同一个git仓库，可以分布到不同的机器

**4.1添加远程库**

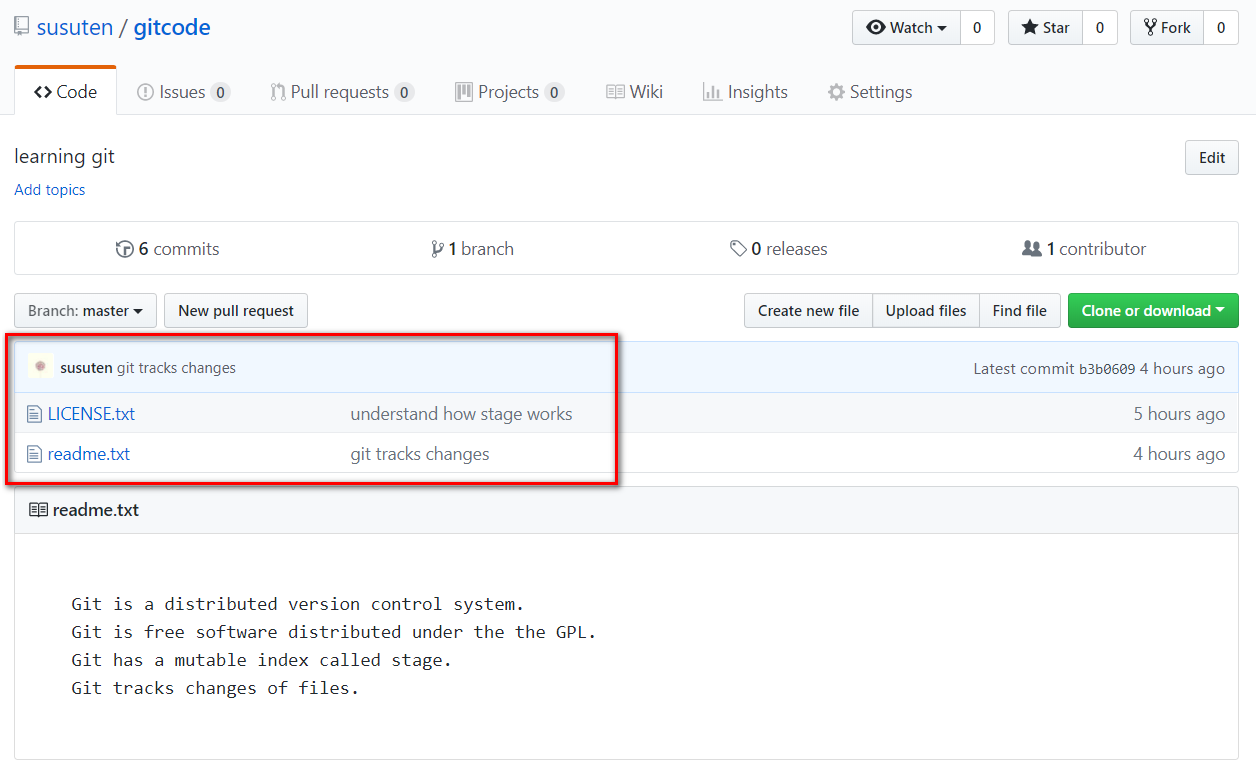
（1）git remote add origin [git@github.com:susuten/gitcode.git](mailto:git@github.com:susuten/gitcode.git)

——>Git push -u origin master——>yes



由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

（2）推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：



（3）从现在起，只要本地作了提交，就可以通过命令：git push origin master把本地master分支的最新修改推送至GitHub。

（4）小结

要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git；

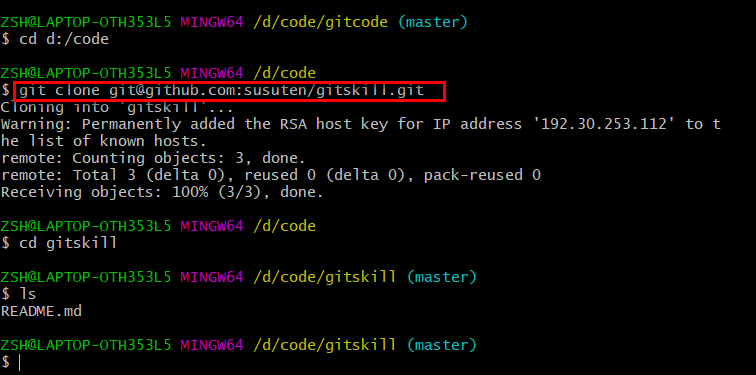
关联后，使用命令git push -u origin master第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最新修改；

分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

**4.2从远程库克隆**

（1）git clone git@github.com:susuten/gitskill.git



或者用<https://git.github.com/susuten/gitskill.git>进行克隆

GitHub给出的地址不止一个，还可以用https://github.com/michaelliao/gitskills.git这样的地址。实际上，Git支持多种协议，默认的git://使用ssh，但也可以使用https等其他协议。

使用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，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。

（2）小结

要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆。

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

**5分支管理**

你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

**5.1创建与合并分支**

（1）创建dev分支并切换到dev分支

Git checkout -b dev # -b参数表示创建并切换，相当于以下两条

Git branch dev

Git checkout dev

（2）用git branch查看当前分支

该命令会列出所有分支，当前分支会标一个\*号

（3）提交

Git add readme.txt

Git commit -m “branch test”

（4）切换回到master分支

Git checkout master

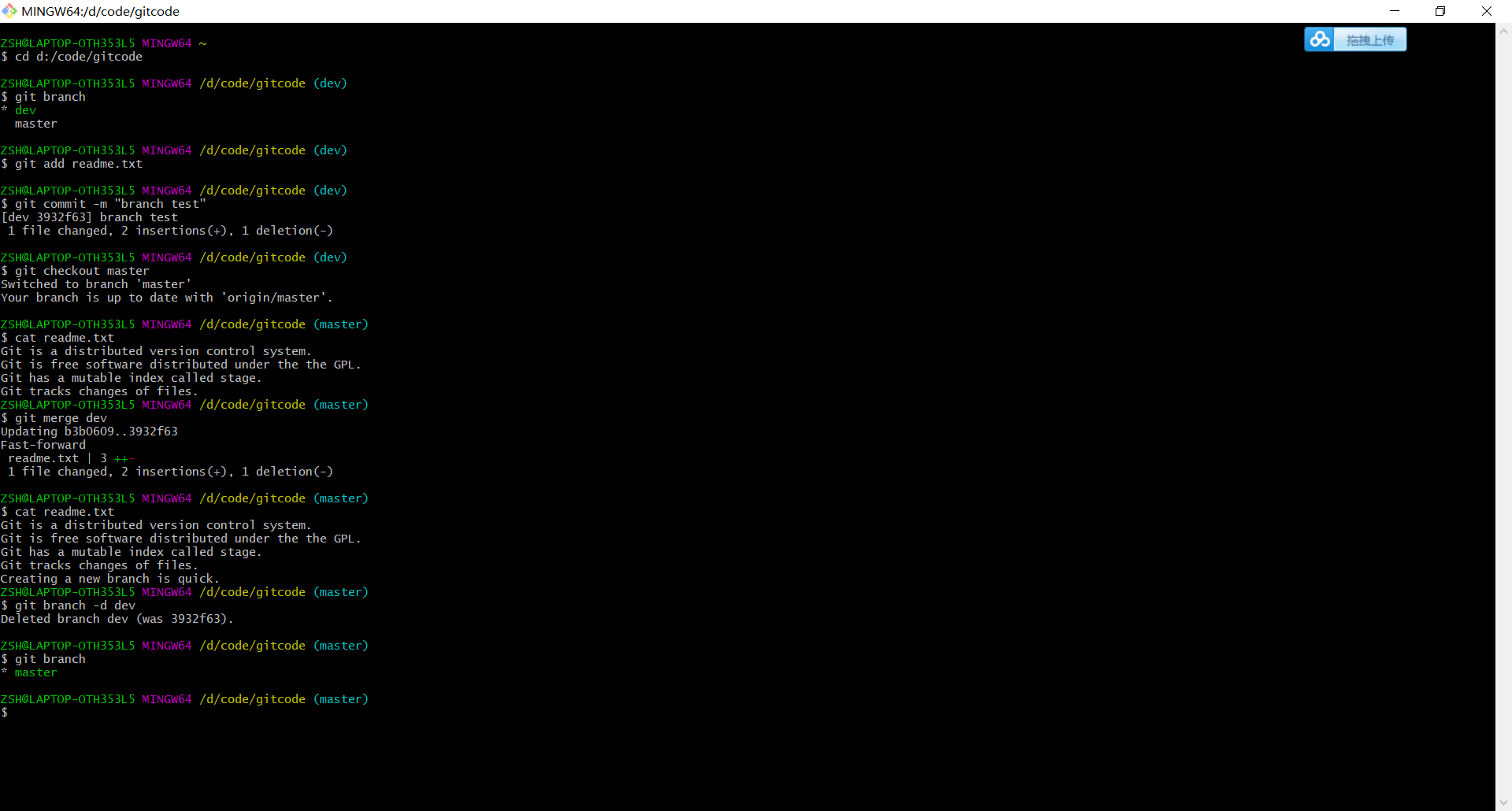
（5）切换回到master分支，再查看一个readme.txt，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有改变

（6）把dev分支的工作成果合并到master分支上

Git merge dev

（7）合并完成后，就可以删除dev分支

Git branch -d dev



（8）小结

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

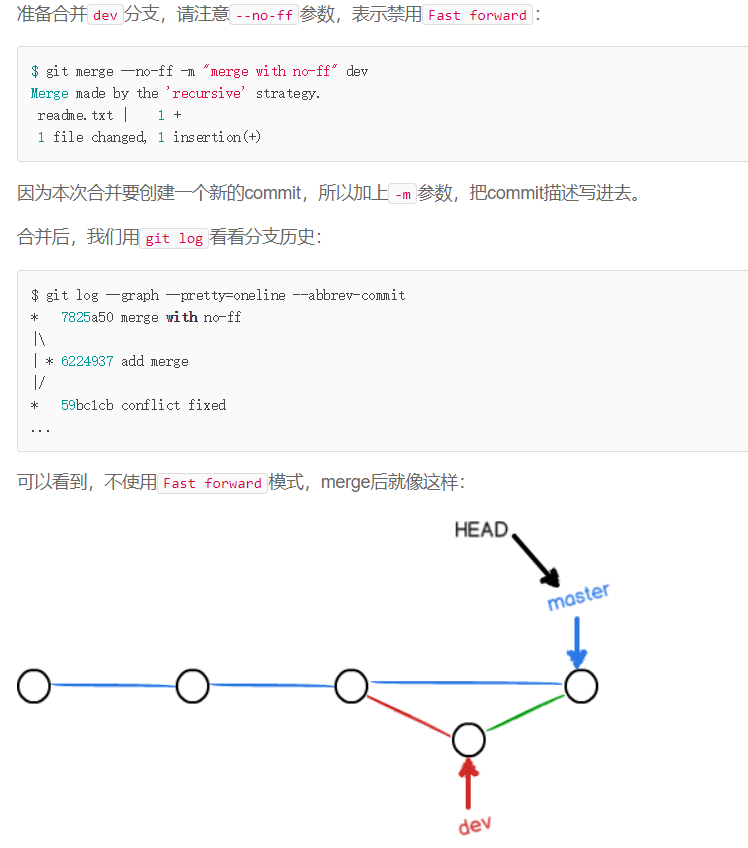
创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

**5.2解决冲突**



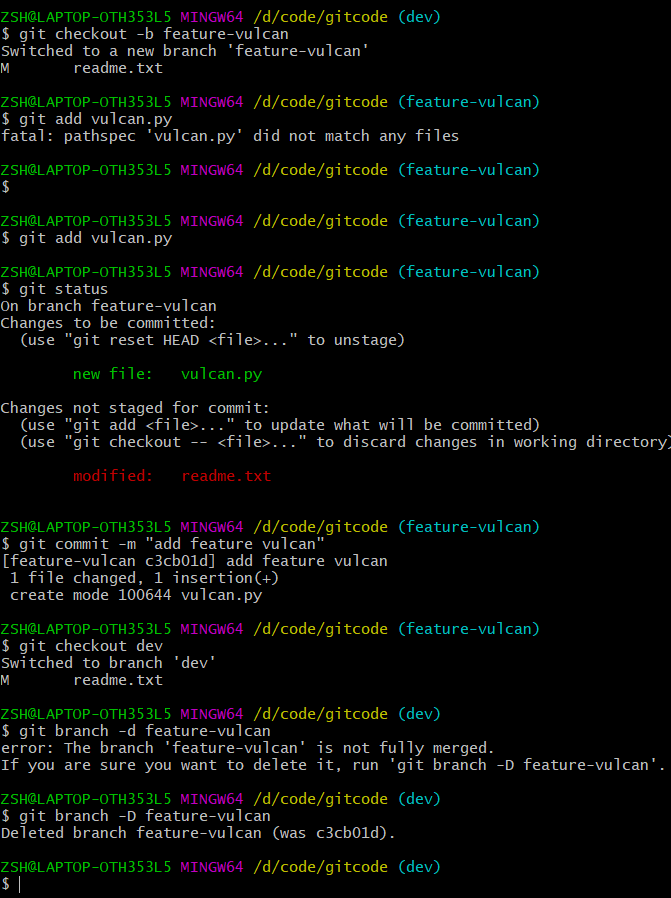




**5.3分支管理策略**

**5.4 Bug分支**

**5.5 Feature分支**



**5.6多人协作**

（1）当你从远程仓库克隆时，实际上Git自动把本地的master和远程的master分支对应起来了，并且，远程仓库的默认名称是origin

（2）查看远程库的信息

Git remote

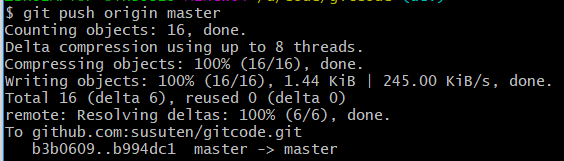


Git remote -v显示更详细的信息



（3）推送分支

Git push origin master 指定分支master，如果要推送其他分支，比如dev，就改成dev



1. master分支是主分支，要时刻与远程同步；dev是开发分支，也需与远程同步；bug分支只用于在本地修复bug，就没有必要推送到远程了；feature分支是否推送到远程，取决于你是否和你的伙伴在上面开发。
2. 抓取分支

Git clone [git@github.com:susuten/gitcode.git](mailto:git@github.com:susuten/gitcode.git)

Git branch 查看分支

Git checkout -b dev origin/dev 创建本地dev

1. 把dev分支push到远程

Git commit -m “add/usr/bin/env”

Git push origin dev

1. Git pull

Git branch –set-upstream dev origin/dev

Git pull

Git commit -m “merge & fix hello.py”

Git push origin dev

**6标签管理**

**6.1创建标签**

（1）git tag <name> 创建一个新的标签

（2）git tag 查看所有标签

（3）git tag v0.9 6224937为commit id为6224937打上标签，名为v0.9

（4）git show <tagname> 查看标签信息

（5）git tag -a v0.1 -m “version 0.1 released” 628146

（6）通过-s用私钥签名一个标签

Git tag -s v0.2 -m “signed version 0.2 released” f4c145a

（7）小结

a. 命令git tag <name>用于新建一个标签，默认为HEAD，也可以指定一个commit id；

b. git tag -a <tagname> -m "blablabla..."可以指定标签信息；

c. git tag -s <tagname> -m "blablabla..."可以用PGP签名标签；

d. 命令git tag可以查看所有标

**6.2操作标签**

(1) 删除标签 git tag -d v0.1\

因为创建的标签都只存储在本地，不会自动推送到远程，所以打错的标签可以在本地安全删除

(2) git push origin <tagname> 推送标签到远程

(3) git push origin –tags 一次性推送全部标签

(4) 已推送到远程时的删除操作

Git tag -d v0.9

Git push origin:refs/tags/v0.9

(5) 小结

命令git push origin <tagname>可以推送一个本地标签；

命令git push origin --tags可以推送全部未推送过的本地标签；

命令git tag -d <tagname>可以删除一个本地标签；

命令git push origin :refs/tags/<tagname>可以删除一个远程标签。

**7使用GitHub**

**8使用码云**

国内Git托管服务——码云（gitee.com）

**5人以下小团队免费**

**9自定义Git**

Git config –global alias.st status

**10总结**