Git 快速入门

目录

[第一章 Git 简介 1](#_Toc4605775)

[第一节 Git 基础 2](#_Toc4605776)

[第二节 git 安装、配置 3](#_Toc4605777)

[一、Windows 安装Git 3](#_Toc4605778)

[二、Ubuntu 安装 Git 8](#_Toc4605779)

[第二章 Git 常用操作 1](#_Toc4605780)

[第三章 Git 服务器搭建 1](#_Toc4605781)

[参考资料 2](#_Toc4605782)

第一章 Git 简介

版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。

Git是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

集中式版本管理系统

在**集中式**版本管理系统中（比如SVN），有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的**文件**或者提交更新。这样，每个人都可以在一定程度上看到项目中的其他人正在做些什么。而管理员也可以轻松掌控每一个开发者的权限。通过中央服务器，可以快速的获取别人提交的最新代码和将自己的更新推送到服务器。每次提交自己修改到服务器前先拉去服务器上的最新代码，如果没有和其他人修改冲突的地方，则直接提交就行；如果很不幸，你的修改和别人冲突则需要先解决冲突再提交。集中式版本控制系统，给代码管理带来很多便利，尤其是提供了全局统一的代码视图。但是，这种方式管理代码有一个非常明显的缺点，那就是中央服务器的故障可能会导致整个历史更新记录丢失。

分布式版本管理系统

于是分布式版本控制系统（Distributed Version Control System，简称 DVCS）面世了。 在这类系统中，像Git等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。 这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的克隆操作，实际上都是一次对代码仓库的**完整备份（clone到本地的仓库和服务器端完全一致）。**其实，git中服务端和客户端仓库地位是平等的，只是为了方便大家将修改推送到一个共同的服务器。所谓的客户端、服务端也只是为了叙述方便而加的。

## 第一节 Git 基础

直接记录快照，而非差异比较

大多数版本控制系统存储的数据由原始文件和一系列随时间逐步累积的差异补丁组成。而**Git**保存数据的方式为：对提交时的**全部文件**制作一个**快照**并保存这个快照的索引。为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个快照流。

近乎所有操作都是本地执行

因为克隆到本地的git仓库拥有完整的历史记录，所以大多是操作在本地可以完成。比如查看历史修改，在集中式版本控制系统中必须链接到版本服务器才可以查看，git中在本地就可以完成（仅限于最近一次git pull 之前提交到服务器的修改历史）。

Git 保证完整性

Git中所有数据在存储前都计算校验和（使用SHA-1算法），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。若你在传送过程中丢失信息或损坏文件，Git就能发现。

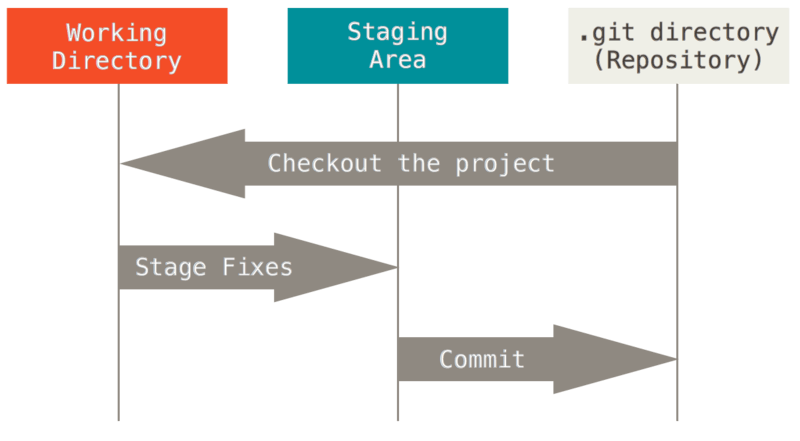
Git 中文件的三种状态

**已提交：**表示数据已经安全的保存在本地数据库（即Git仓库.git文件夹）中。

**已修改：**表示自上次提交后（准确来说应该是暂存后，一般情况下暂存后立即提交）在工作目录中修改了文件，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。

与之相对应的关于Git项目目录的三个概念：**Git 仓库、工作目录以及暂存区域**



**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，拷贝的就是这里的数据。

**工作目录**：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。

**暂存区域**：是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。 有时候也被称作‘**索引**’，不过一般说法还是叫暂存区域。

基本的 Git 工作流程如下：

1. 在工作目录中修改文件。

2. 暂存文件，将文件的快照放入暂存区域。

3. 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

4. 将本地修改推送至远端。

如果 Git 目录中保存着的特定版本文件，就属于已提交状态。 如果作了修改并已放入暂存区域，就属于已暂存状态。 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

## 第二节 git 安装、配置

### 一、Windows 安装Git

1. 下载安装包

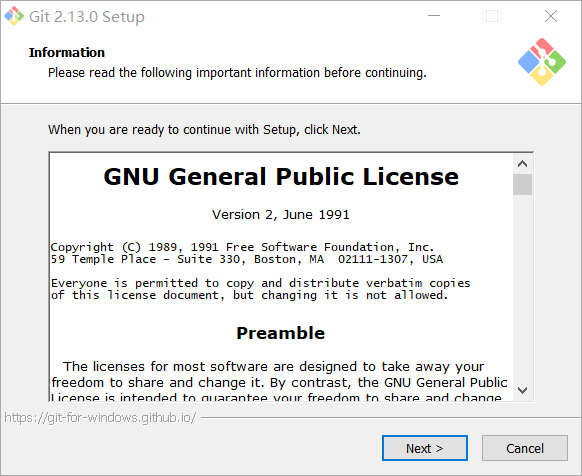
到[Git官网](https://git-scm.com/download/win)下载Git for windows；

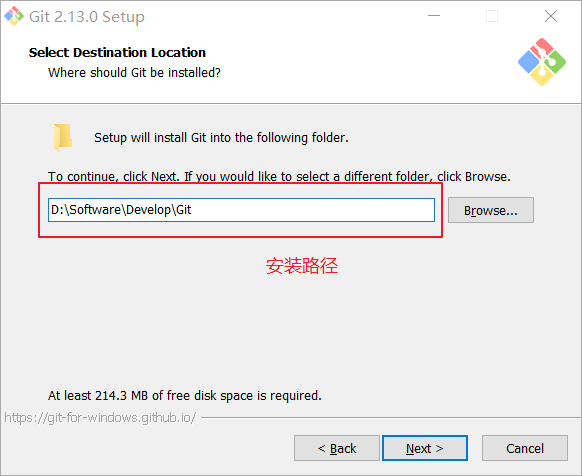
如果需要使用GitHub ，则需要下载[Git Large File Storage](https://git-lfs.github.com/)；

如果需要使用图形界面，则需要下载[TortoiseGit](https://tortoisegit.org/).

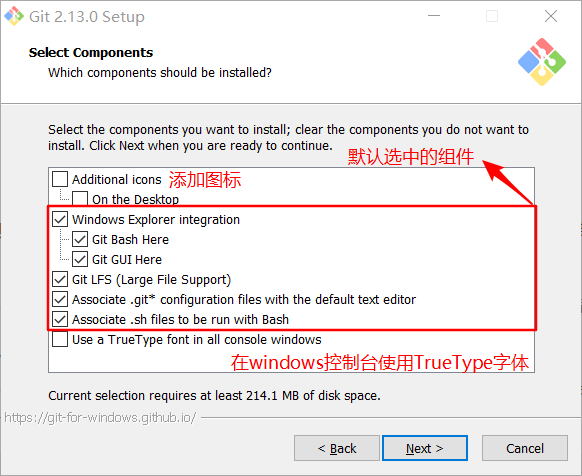
1. 安装步骤

Git 安装基本步骤如下图所示，TortoiseGit和Git Large File Storage默认安装即可。TortoiseGit 提供Git的图形界面操作，喜欢的话可以装。Git Large File Storage 用于向GitHub上传大于100M的文件。

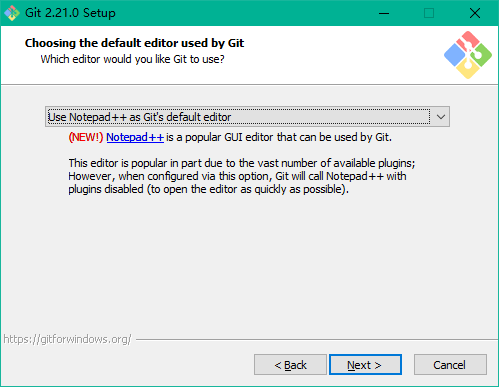


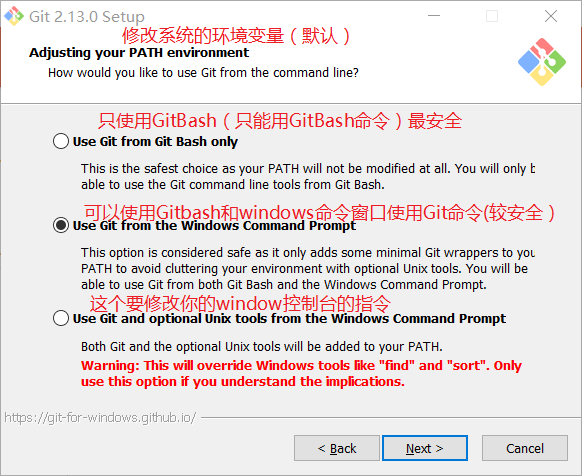


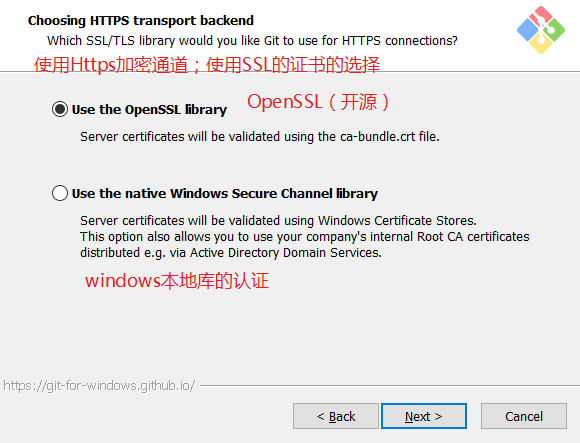
这里可以Bash GUI选项去掉，很简单的界面没啥用，如果要用图形界面可以安装TortoiseGit.

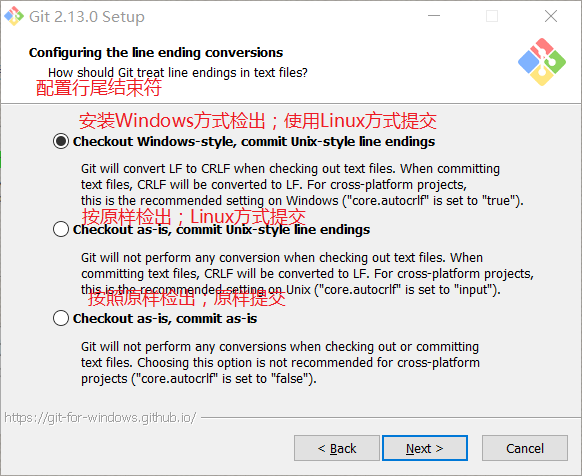


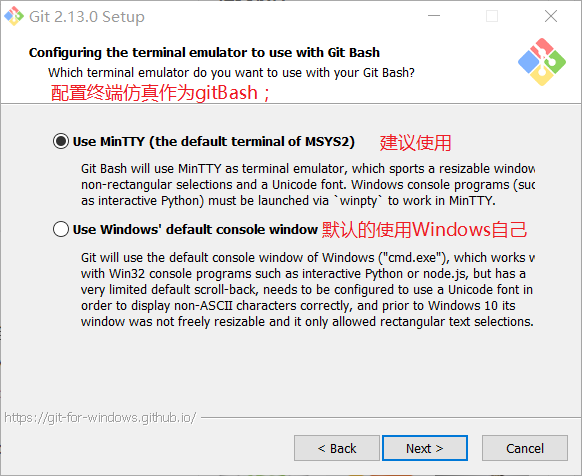
在这里选择notepad++作为默认的文本编辑器，提交代码时如果没有-m选项，则会用这里选的文本编辑器打开一个文件来写入我们对本次提交的说明信息。

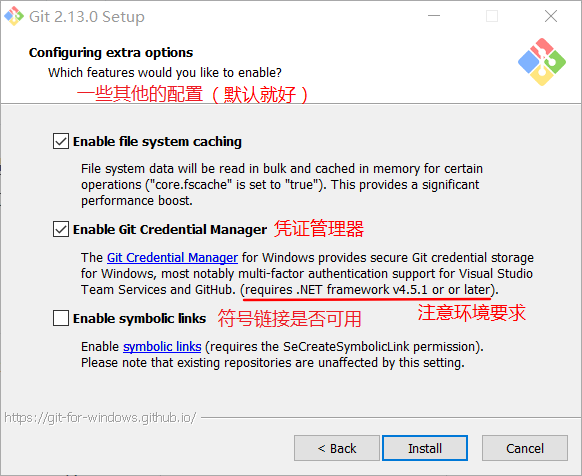






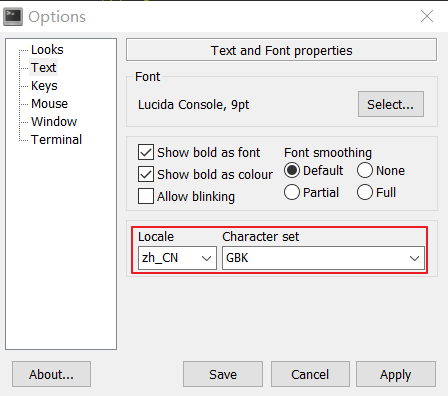






然后，等待安装完成。

安装完成后，运行Git Bash 右键-》选项，如下配置可解决中文乱码问题。



1. 安装完成后的基本配置

#配置用户名，GitHub注册时的用户名

$ git config --global user.name "Your Name"

#设置邮箱，使用github注册时的邮箱

$ git config --global user.email "email@example.com"

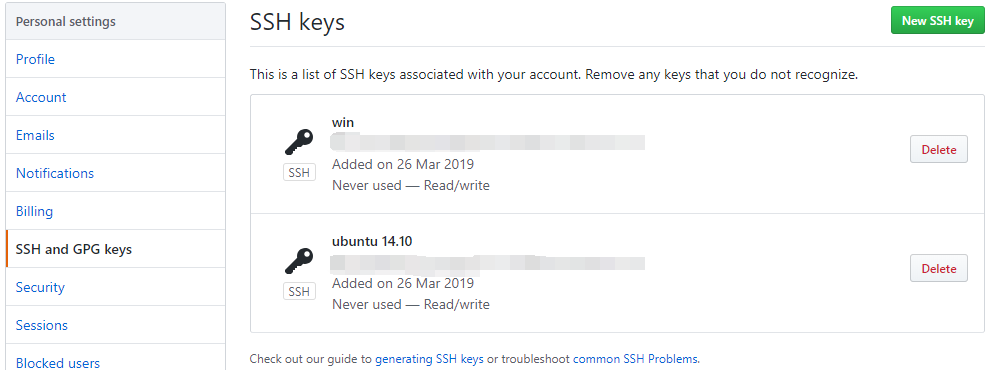
#git关联notepad++

git config --global core.editor "&apos;C:\Program Files\Notepad++\notepad++.exe&apos; -multiInst -notabbar -nosession -noPlugin"

#生成公钥和私钥，id\_rsa.pub公钥，id\_rsa为私钥，这两个文件在~/.ssh目录下

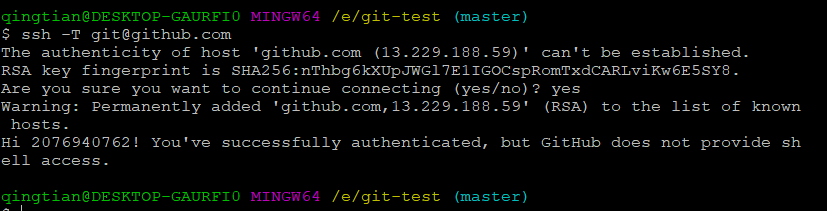
$ ssh-keygen -t rsa -C [1234\*@qq.com](mailto:1234*@qq.com)

将上一步中生成的公钥复制到GitHub https://github.com/settings/keys



#测试配置，出现如下界面则配置成功。

$ ssh -T git@github.com



### 二、Ubuntu 安装 Git

$ sudo apt-get install git

配置和Windows基本相同。

设置默认编辑器：

git config --global core.editor notepadqq

git config --global core.editor notepad-plus-plus

Ubuntu下的安装方法:

sudo add-apt-repository ppa:notepadqq-team/notepadqq

sudo apt-get update

sudo apt-get install notepadqq

Ubuntu下的卸载方法:

sudo apt-get remove notepadqq

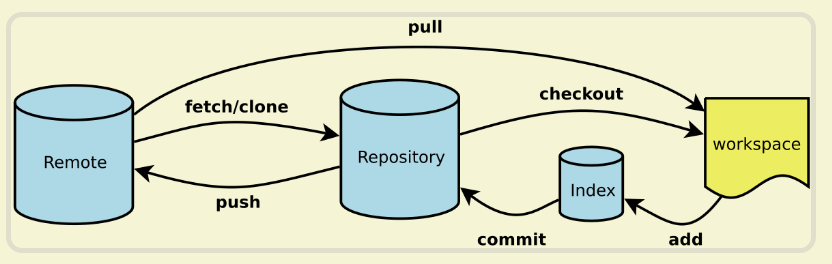
sudo add-apt-repository --remove ppa:notepadqq-team/notepadqq

#Notepad++安装

sudo snap install notepad-plus-plus

第二章 Git 常用操作

第一节 git基本操作



**Workspace：工作区 Index / Stage：暂存区 Repository：仓库区（或本地仓库） Remote：远程仓库**

Git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）。

# 显示当前的Git配置

$ git config –list

git config -l #列举所有配置

# 编辑Git配置文件

$ git config -e [--global]

# 设置提交代码时的用户信息

$ git config [--global] user.name "[name]"

$ git config [--global] user.email "[email address]"

--system #系统级别

--global #用户全局

--local #单独一个项目

下层的配置会覆盖上层的配置

git config --global user.name "xxxx" #用户名

git config --global user.email "xxxx@xxx.com" #邮箱

git config --global core.editor vim #编辑器

#检查 Git 的某一项配置

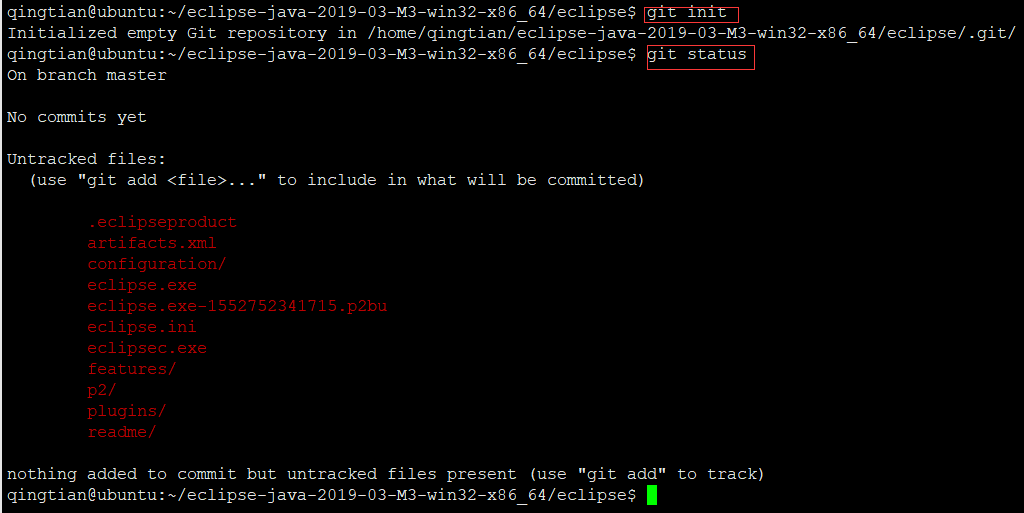
git config <key>

git config --global alias.st status #按这种方法，配置别名

创建仓库

# 在当前目录新建一个Git代码库，执行完成后，会在当前文件夹下.git目录初始化一个空的仓库。如果该目录原来有文件，这些文件不会被删除，这些文件处于未跟踪状态。

$ git init



# 新建一个目录，将其初始化为Git代码库

$ git init [project-name]

# 下载一个项目和它的整个代码历史

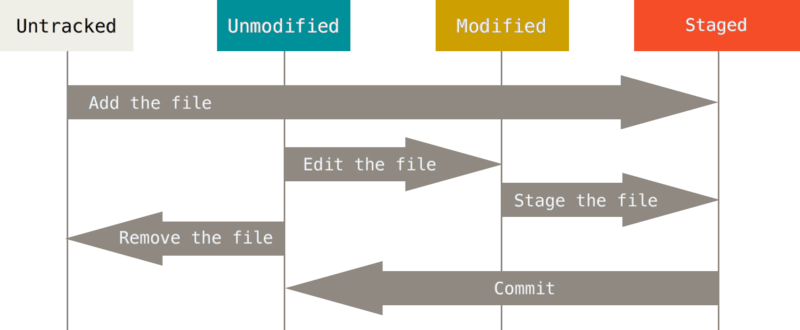
$ git clone [url]

#克隆并重命名

$ git clone <url> <新仓库名>

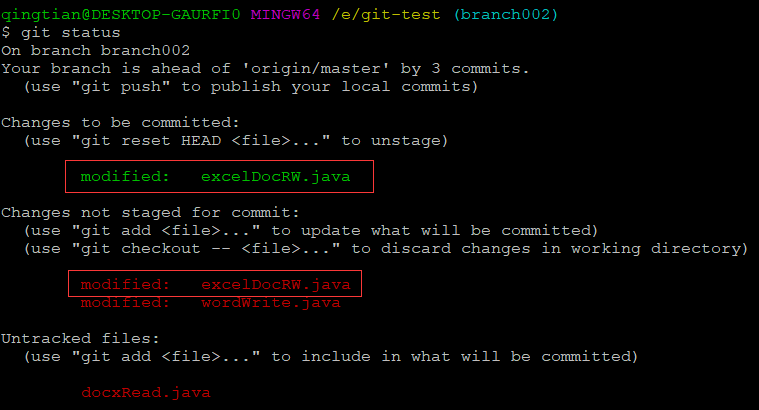
文件状态

**已跟踪的文件**是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。 工作目录中除已跟踪文件以外的所有其它文件都属于**未跟踪文件**，它们既不存在于上次快照的记录中，也没有放入暂存区。 初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为已修改文件。 我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：



#查看当前仓库简要状态信息

$ git status



**Changes to be committed:**表示文件自上次提交后，有过修改，并且已经add到暂存区，但是没有提交到仓库，可以使用"git reset HEAD <file>..."命令放弃暂存，或者使用”git commit ”命令将暂存提交到仓库。

**Changes not staged for commit:**表示文件自上次提交后有过修改，但是没有add到暂存区。使用"git add <file>..."命令将修改后的文件快照添加到暂存区；或者使用"git checkout -- <file>..."命令用暂存区中的文件快照覆盖工作目录中的文件，也就是说放弃了当前修改。

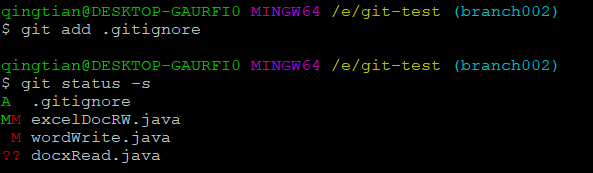
**Untracked files:**为跟踪文件，使用"git add <file>..."命令会开始跟踪文件。

**Your branch is ahead of 'origin/master' by 3 commits.**表示自上从origin/master仓库中pull后，本地仓库已经有了三次提交。可以使用git push origin master 将这三次提交推送到远端。

上图中文件excelDocRW.java同时出现在暂存区和非暂存区。出现在暂存区中的excelDocRW.java是上一次运行git add命令时生成的文件快照。在执行玩add后，没有执行git commit 而是继续修改excelDocRW.java文件，工作目录中的文件和暂存区中的快照不一致，所以excelDocRW.java文件同时出现在已修改和已暂存状态。

#以简短的方式输出状态信息

$ git status -s



??：未跟踪文件

A:新添加到暂存区的文件

MM:左边的M表示文件被修改过，并且已经add到暂存区；右边的M表示在工作目录修改了但是没有放入暂存区。

#以文件补丁形式显示具体的修改内容

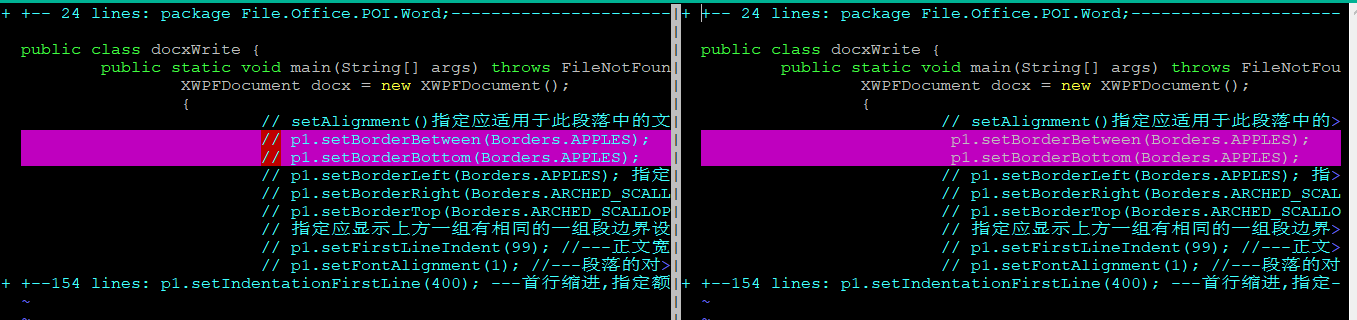
#此命令比较的是工作目录中当前文件和暂存区域快照之间的差异， 也就是修改之后还没有暂存起来的变化内容。

$ git diff [文件名]

#对比暂存区和仓库中的文件快照差异，查看已暂存的将要添加到下次提交里的内容。

git diff –cached

git difftool 命令将以如下分屏方式显示文件差异，在Windows下可以继承Beyond Compare 工具更加友好的查看文件差异。



# 显示工作区与当前分支最新commit之间的差异

$ git diff HEAD

# 显示两次提交之间的差异

$ git diff [first-branch]...[second-branch]

# 显示今天你写了多少行代码

$ git diff --shortstat "@{0 day ago}"

增加/删除文件

# 如果文件尚未被跟踪，则开始跟踪文件并将文件添加到暂存区；如果文件处于已修改状态，则将文件快照添加到暂存区；合并（merge）时把有冲突的文件标记为已解决状态。

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录

$ git add [dir]

# 添加当前目录的所有文件到暂存区

$ git add .

# 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

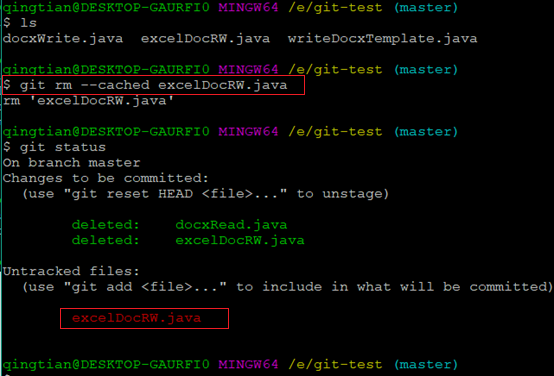
$ git add -p

#从已跟踪文件清单中删除，并删除工作目录中的磁盘文件。如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f（译注：即 force 的首字母）。如果手工在工作目录中删除，git status 会出现“Changes not staged for commit”。

$ git rm [file1] [file2] ...

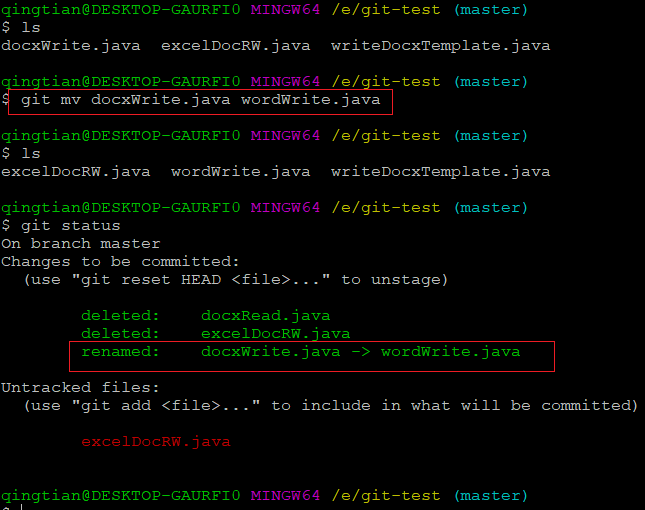
# 停止追踪指定文件，但该文件会保留在工作区

$ git rm --cached [file]



#重命名文件，并且将这个改名放入暂存区

$ git mv [file-original] [file-renamed]



撤销

#这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息。

$ git commit –amend

#你提交后发现忘记了暂存某些需要的修改，可以像下面这样操作

$ git add forgotten\_file

$ git commit --amend

# 恢复暂存区的指定文件到工作区

$ git checkout -- [file]

# 恢复某个commit的指定文件到暂存区和工作区

$ git checkout [commit] [file]

# 恢复暂存区的所有文件到工作区

$ git checkout .

# 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 [file]

# 重置暂存区与工作区，与上一次commit保持一致

$ git reset --hard

# 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 [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 --hard [commit]

# 重置当前HEAD为指定commit，但保持暂存区和工作区不变

$ git reset --keep [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 revert [commit]

# 暂时将未提交的变化移除，稍后再移入

$ git stash

$ git stash pop

代码提交

# 提交暂存区到仓库区，如果没有-m选项则会用第一章中设置的默认文本编辑器打开一个文本，在其中编辑本次提交的描述信息，关闭文本编辑器后自动完成提交。

$ git commit -m [message]

# 提交暂存区的指定文件到仓库区

$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区，**跳过暂存区**

$ git commit -a

# 提交时显示所有diff信息

Show unified diff between the HEAD commit and what would be committed at the bottom of the commit message template to help the user describe the commit by reminding what changes the commit has. This diff will not be a part of the commit message.

If specified twice, show in addition the unified diff between what would be committed and the worktree files, i.e. the unstaged changes to tracked files.

$ git commit -v

# 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 -m [message]

# 重做上一次commit，并包括指定文件的新变化

$ git commit --**amend** [file1] [file2] ...

标签

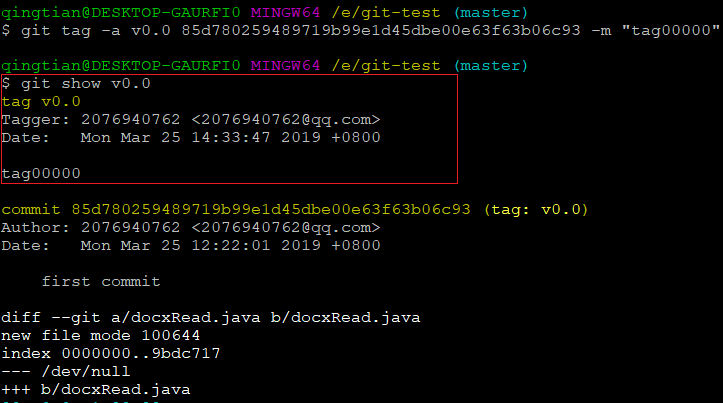
Git 可以给历史中的某一个提交打上标签，以示重要。 比较有代表性的是人们会使用这个功能来标记发布结点（v1.0 等等）。Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）。

一个**轻量标签**很像一个不会改变的分支，它只是一个特定提交的引用。然而，**附注标签**是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU Privacy Guard （GPG）签名与验证。**创建附注标签需要指定-a选项。** 通常建议创建附注标签，这样你可以拥有以上所有信息；但是如果你只是想用一个临时的标签，或者因为某些原因不想要保存那些信息，轻量标签也是可用的。

#打**附注标签**v1.0 ,-a 指定标签名； -m 选项指定了一条将会存储在标签中的信息。

#-a 表示创建**附注标签**，-m 选项指定了一条将会存储在标签中的信息。

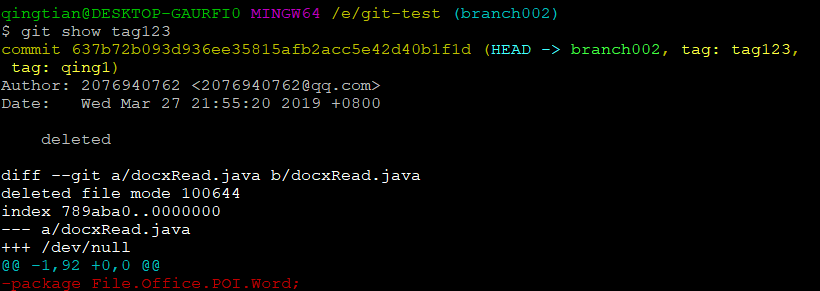
$ git tag **-a** v1.0 -m "1.0tag"



注意轻量级标签没有红色方框内的信息

#轻量标签本质上是将提交校验和存储到一个文件中没有保存任何其他信息。 创建轻量标签，不需要使用 -a、-s 或 -m 选项，只需要提供标签名字。但它不会记录这标签是啥时候打的，谁打的，也不会让你添加个标签的注解。git show，你不会看到额外的标签信息。 命令只会显示出提交信息。

$ git tag <tagname>



# 列出所有tag

$ git tag

# 新建一个轻量级标签在当前commit

$ git tag [tag]

#对过去的提交打标签

git tag -a <标签名> <校验和/部分校验和>

#删除本地标签

git tag -d <tagname>

#推送本地轻量级标签删除到远程

git push <remote> :refs/tags/<tagname>

# 查看tag信息

$ git show [tag]

#默认情况下，git push 命令并不会传送标签到远程仓库服务器上。 在创建完标签后你必须显式地推送标签到共享服务器上。 这个过程就像共享远程分支一样,运行

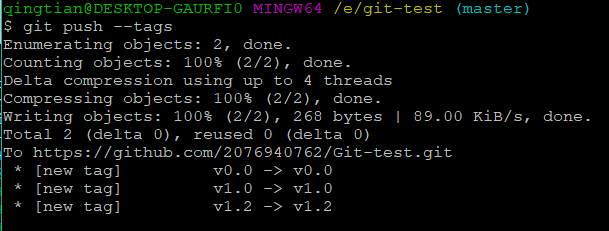
$ git push origin [tagname]

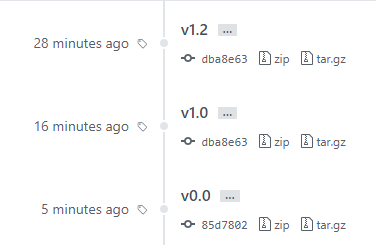
$ git push [remote] [tag]

#把所有不在远程仓库服务器上的标签全部传送到那里。

$ git push origin --tags

$ git push [remote] –tags





# 新建一个分支，指向某个tag

$ git checkout -b [branch] [tag]

查看提交历史

# 显示当前分支的版本历史

$ git log

# 显示commit历史，以及每次commit发生变更的文件。在每次提交的下面列出所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。 在每次提交的最后还有一个总结。

$ git log --stat

# 搜索提交历史，根据关键词

$ git log -S [keyword]

# 显示某个commit之后的所有变动，每个commit占据一行

$ git log [tag] HEAD --pretty=format:%s

# 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 [tag] HEAD --grep feature

# 显示某个文件的版本历史，包括文件改名

$ git log --follow [file]

$ git whatchanged [file]

# 显示指定文件相关的每一次diff

$ git log -p [file]

# 显示过去5次提交，将每个提交放在一行显示

$ git log -5 --pretty –oneline

git log --pretty=format 常用的选项 列出了常用的格式占位符写法及其代表的意义。

git log --pretty=format 常用的选项

|  |  |
| --- | --- |
| 选项 | 说明 |
| %H | 提交对象（commit）的完整哈希字串 |
| %h | 提交对象的简短哈希字串 |
| %T | 树对象（tree）的完整哈希字串 |
| %t | 树对象的简短哈希字串 |
| %P | 父对象（parent）的完整哈希字串 |
| %p | 父对象的简短哈希字串 |
| %an | 作者（author）的名字 |
| %ae | 作者的电子邮件地址 |
| %ad | 作者修订日期（可以用 --date= 选项定制格式） |
| %ar | 作者修订日期，按多久以前的方式显示 |
| %cn | 提交者（committer）的名字 |
| %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 |
| %cr | 提交日期，按多久以前的方式显示 |
| %s | 提交说明 |

#列出所有最近两周内的提交

$ git log --since=2.weeks

#查看 Git 仓库中，2008 年 10 月期间，Junio Hamano 提交的但未合并的测试文件

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" --before="2008-11-01" --no-merges

限制 git log 输出的选项

|  |  |
| --- | --- |
| 选项 | 说明 |
| -(n) | 仅显示最近的 n 条提交 |
| --since, --after | 仅显示指定时间之后的提交。 |
| --until, --before | 仅显示指定时间之前的提交。 |
| --author | 仅显示指定作者相关的提交。 |
| --committer | 仅显示指定提交者相关的提交。 |
| --grep | 仅显示含指定关键字的提交 |
| -S | 仅显示添加或移除了某个关键字的提交 |

# 显示所有提交过的用户，按提交次数排序

$ git shortlog -sn

# 显示指定文件是什么人在什么时间修改过

$ git blame [file]

# 显示某次提交的元数据和内容变化

$ git show [commit]

# 显示某次提交发生变化的文件

$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容

$ git show [commit]:[filename]

# 显示当前分支的最近几次提交

$ git reflog

远程同步

#列出你指定的每一个远程服务器的简写

$ git remote

# 显示所有远程仓库

$ git remote -v

# 显示某个远程仓库的信息

$ git remote show [remote]

# 增加一个新的远程仓库，并命名

$ git remote add [shortname] [url]

#将远程仓库 pb 重命名为 paul

$ git remote rename pb paul

#删除远程服务器

$ git remote rm <远程服务器名，像origin>

# 取回远程仓库的变化，并与本地分支**合并**

$ git pull [remote] [branch]

#从远程仓库中拉取所有你还没有的数据。 执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。并不会自动合并或修改你当前的工作。 当准备好时你必须手动将其合并入你的工作。

$ git fetch [remote-name]

# 上传本地指定分支到远程仓库

$ git push [remote] [branch]

# 强行推送当前分支到远程仓库，即使有冲突

$ git push [remote] --force

# 推送所有分支到远程仓库

$ git push [remote] –all

GitHub上传大文件

GitHub大于100M的文件需要用Git Large File Storage上传，操作步骤如下：

To get started with Git LFS, the following commands can be used.

1. Setup Git LFS on your system. You only have to do this once per repository per machine:

git lfs install

2. Choose the type of files you want to track, for examples all ISO images, with git lfs track:

git lfs track "\*.iso"

3. The above stores this information in gitattributes(5) files, so that file need to be added to the repository:

**git add .gitattributes**

3. Commit, push and work with the files normally:

git add file.iso

git commit -m "Add disk image"

git push

忽略某些文件

大多时候我们只需上传代码，编译产生的中间文件以及日志文件并不需要上传至远程仓库。这时需要在工作目录的根目录下创建一个.gitignore文件，并提交到仓库。以后生成的日志等文件，如果在.gitignore中指定过则会直接忽略。

**.gitignore** 的格式规范如下：

所有空行或者以 ＃ 开头的行都会被 Git 忽略。

可以使用标准的 glob 模式匹配。

匹配模式可以以（/）开头防止递归。

匹配模式可以以（/）结尾指定目录。

要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。

glob 模式是指 shell 所使用的简化了的正则表达式。 星号（\*）匹配零个或多个任意字符；[abc] 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；问号（?）只匹配一个任意字符；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如 [0-9] 表示匹配所有 0 到 9 的数字）。 使用两个星号（\*) 表示匹配任意中间目录，比如`a/\*\*/z` 可以匹配 a/z, a/b/z 或 `a/b/c/z`等。

GitHub上已经有针对大多数开发语言写好的.gitignore 文件仓库，下载后更改文件名，复制到你自己的仓库根目录下，提交即可。

<https://github.com/github/gitignore>

获取帮助

Git获取帮助的三种方式：

$ git <verb> --help

$ man git-<verb>

$ git help <verb>

git help config 会打开网页版帮助文档。也可以直接在git安装目录下打开用户手册，路径如下：

D:\Program Files\Git\mingw64\share\doc\git-doc\user-manual.html

其他

# 生成一个可供发布的压缩包

$ git archive

#git push origin master

推送到远程库

#git remote -v

显示远程仓库的URL

第二节 分支管理

分支

在其它版本控制系统，创建新分支常常需要完全创建一个源代码目录的副本。效率比较低，所以很少创建分支。而Git创建分支几乎瞬间完成，所以Git鼓励频繁的创建分支与合并。在实际开发中，可以为每个新特性、待修复的bug创建一个分支，完成后再合并到主线分支。

Git 保存的不是文件的变化或者差异，而是一系列不同时刻的**文件快照**。在进行提交操作时，Git 会保存一个**提交对象**（commit object）。该提交对象会包含一个指向暂存内容快照的指针。除此之外，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象，而由多个分支合并产生的提交对象有多个父对象。**暂存操作会为每一个文件计算校验和（使用SHA-1 哈希算法），然后会把当前版本的文件快照保存到Git仓库中（Git 使用 blob 对象来保存它们），最终将校验和加入到暂存区域等待提交**。

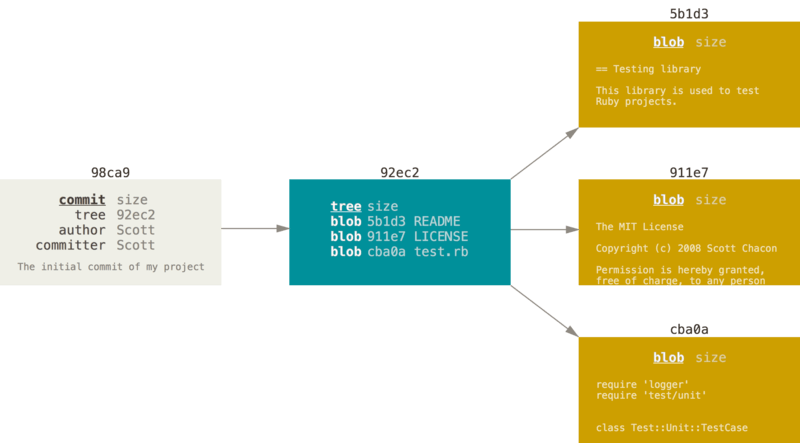
**当使用 git commit 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在Git 仓库中这些校验和保存为树对象。 随后，Git 便会创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。如此一来，Git 就可以在需要的时候重现此次保存的快照。**

举个例子：设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件（README test.rb LICENSE）。将它们提交到本地仓库，

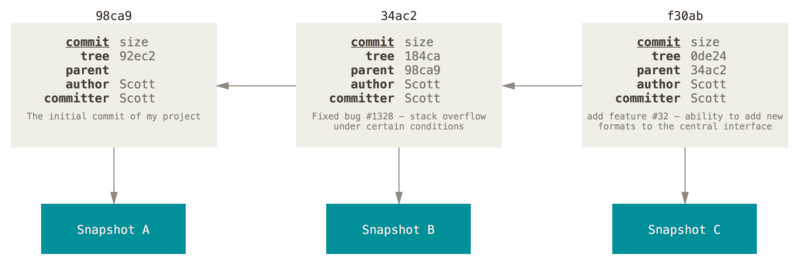
$ git add README test.rb LICENSE

$ git commit -m 'The initial commit of my project'

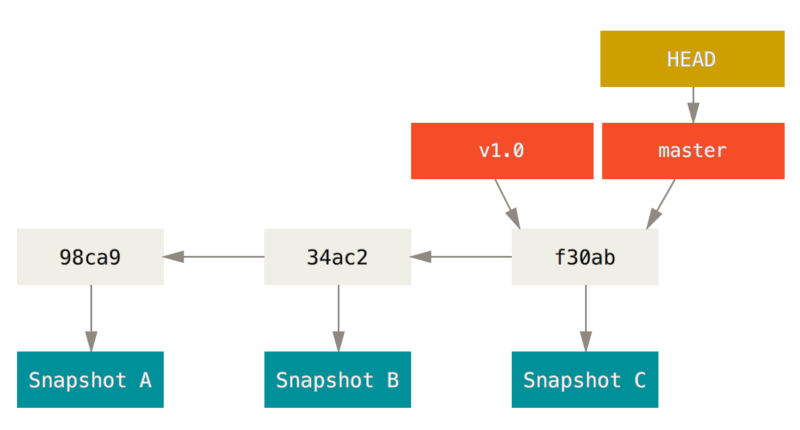
现在，Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。



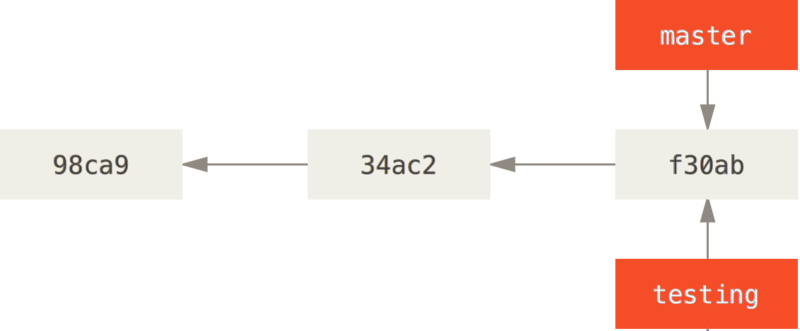
做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



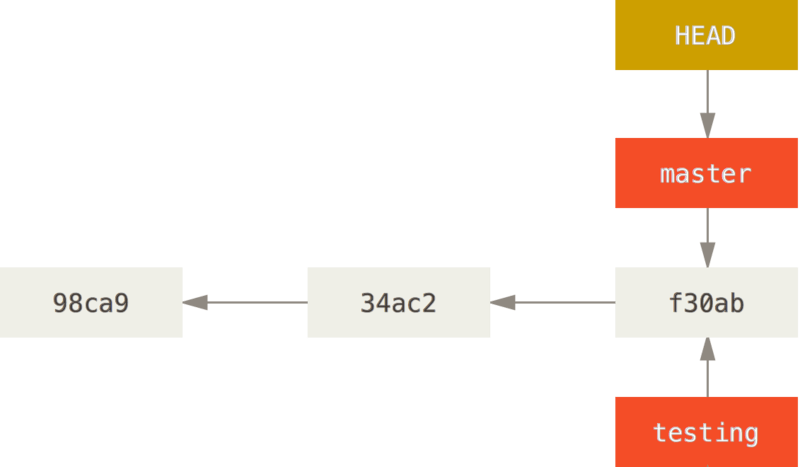
**Git 的分支，其实本质上仅仅是指向提交对象的可变指针。** Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 它会在每次的提交操作中自动向前移动。



Git 是怎么创建新分支的呢？ 很简单，它只是为你创建了一个可以移动的新的指针。

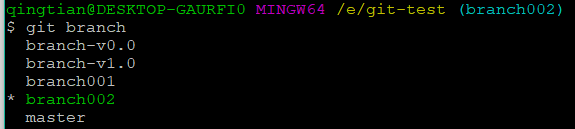


Git 又是怎么知道当前在哪一个分支上呢？ 也很简单，它有一个名为 HEAD 的特殊指针，指向当前所在的本地分支（将 HEAD 想象为当前分支的别名）。



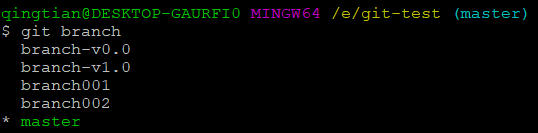
可以使用git log 或者git branch查看当前分支。





# 列出所有本地分支，\*表示当前分支

$ git branch



# 列出所有远程分支

$ git branch -r

# 列出所有本地分支和远程分支

$ git branch -a

# 新建一个分支，但依然停留在当前分支

$ 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，指向指定commit

$ git branch [branch] [commit]

# 新建一个分支，与指定的远程分支建立追踪关系

$ git branch --track [branch] [remote-branch]

# 切换到指定分支，并更新工作区

$ git checkout [branch-name]

# 切换到上一个分支

$ git checkout -

# 建立追踪关系，在现有分支与指定的远程分支之间

$ git branch --set-upstream [branch] [remote-branch]

# 合并指定分支到当前分支

$ git merge [branch]

# 选择一个commit，合并进当前分支

$ git cherry-pick [commit]

# 删除分支

$ git branch -d [branch-name]

# 删除远程分支

$ git push origin --delete [branch-name]

$ git branch -dr [remote/branch]

合并与变基

第三章 Git 服务器搭建

参考资料

[Pro git 2](https://progit.bootcss.com/#_pro_git)

<http://www.cnblogs.com/schaepher/p/5561193.html>