

# Git 学习笔记

---

# 目录

第一章 Git 简介.....	1
第一节 Git 基础.....	2
第二节 git 安装、配置.....	3
一、Windows 安装 Git.....	3
二、Ubuntu 安装 Git.....	8
第二章 Git 常用操作.....	10
第一节 git 基本操作.....	10
Git 配置.....	10
创建仓库.....	11
文件状态.....	11
增加/删除文件.....	14
撤销.....	15
将多个提交压缩.....	16
代码提交.....	19
标签.....	20
查看提交历史.....	22
远程同步.....	24
GitHub 上传大文件.....	25
忽略某些文件.....	26
获取帮助.....	26
其他.....	26
第二节 分支管理.....	27
分支简介.....	27
分支管理常用命令.....	32
合并与变基.....	35
第三节 其它命令.....	39
第三章 Git 服务器.....	43
第一节 Git 支持的协议.....	43
本地协议.....	43
HTTP 协议.....	44
SSH 协议.....	45
Git 协议.....	45
第二节 配置 Git 服务器.....	45
一、使用本地协议的服务器.....	45
二、使用 SSH 协议的服务器.....	46
三、Smart HTTP 服务器.....	48
参考资料.....	52

---

## 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。

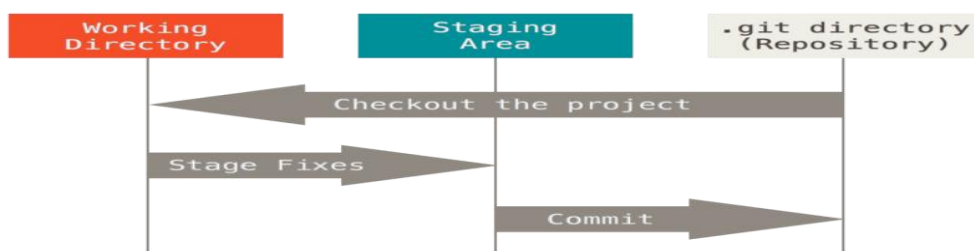
### Git 中文件的三种状态

**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

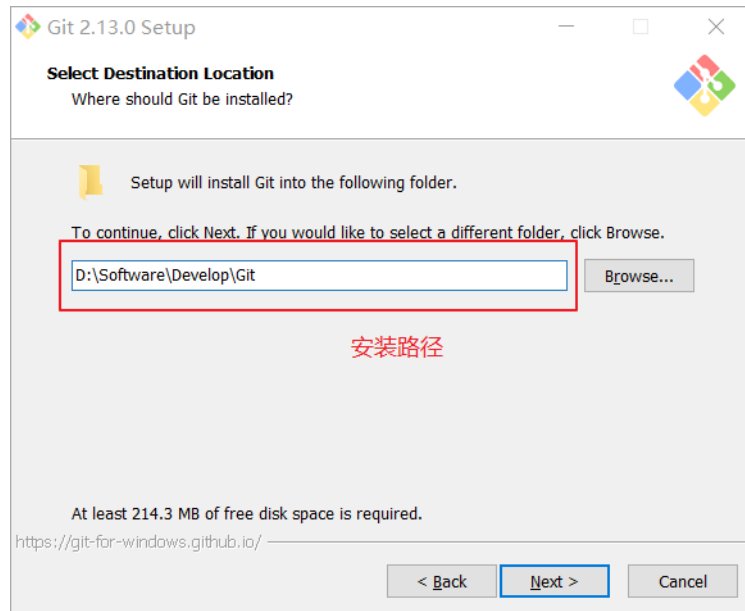
到 [Git 官网](#) 下载 Git for windows;

如果需要使用 GitHub , 则需要下载 [Git Large File Storage](#);

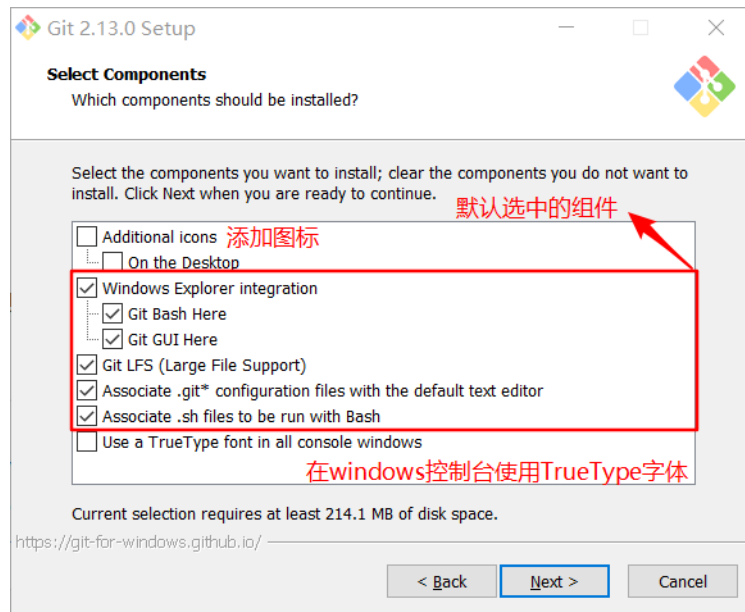
如果需要使用图形界面, 则需要下载 [TortoiseGit](#).

#### 2. 安装步骤

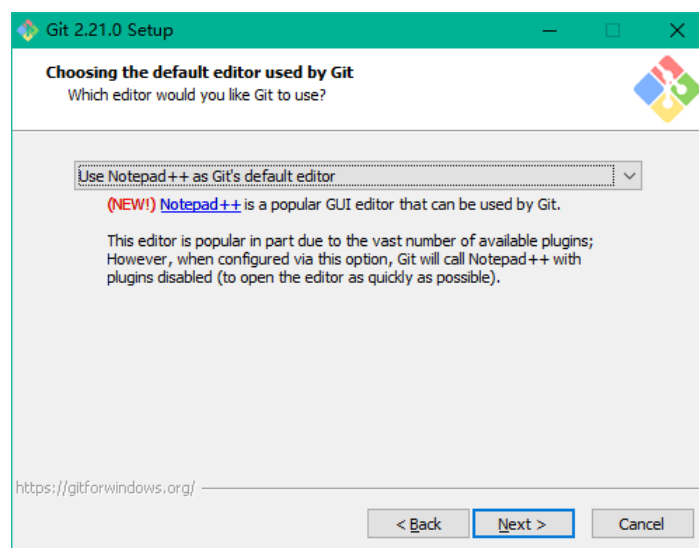
Git 安装基本步骤如下图所示, TortoiseGit 和 Git Large File Storage 默认安装即可。TortoiseGit 提供 Git 的图形界面操作, 喜欢的话可以装。Git Large File Storage 用于向 GitHub 上传大于 100M 的文件。

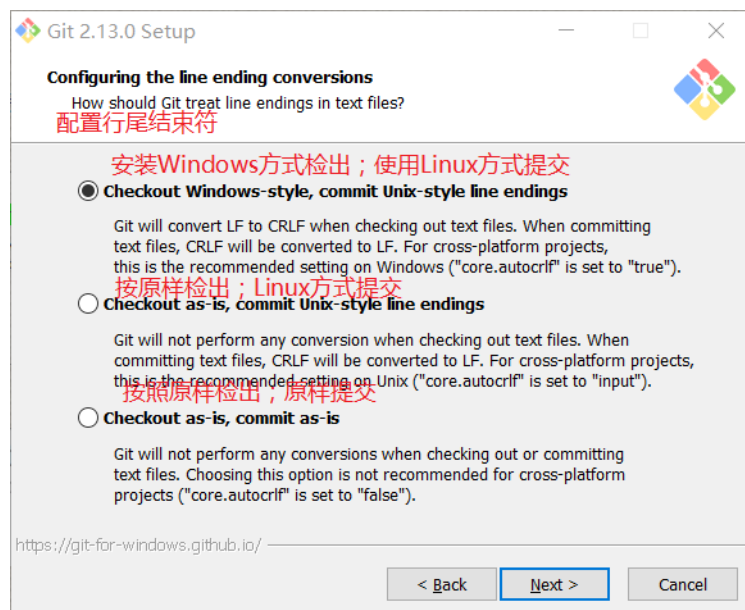
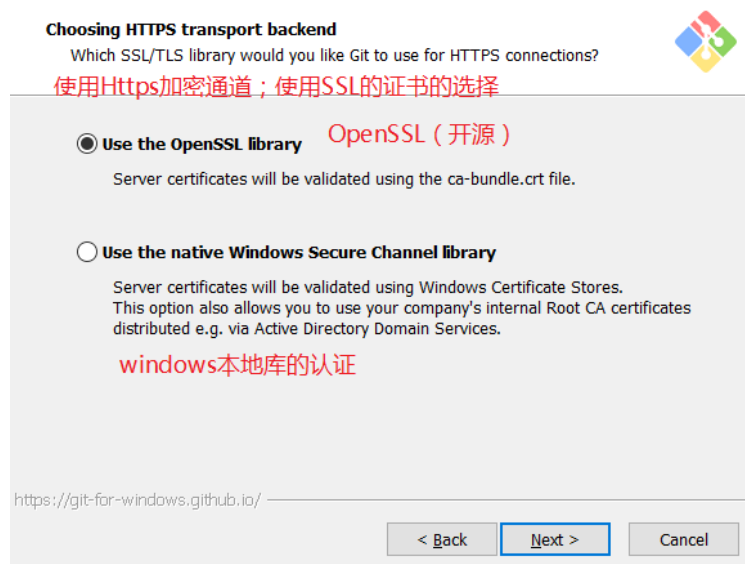
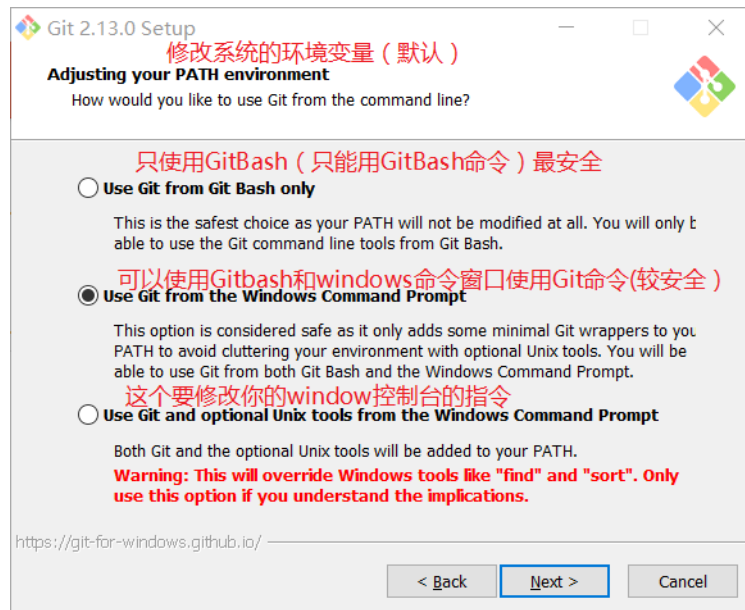


这里可以 Bash GUI 选项去掉，很简单的界面没啥用，如果要用图形界面可以安装 TortoiseGit.

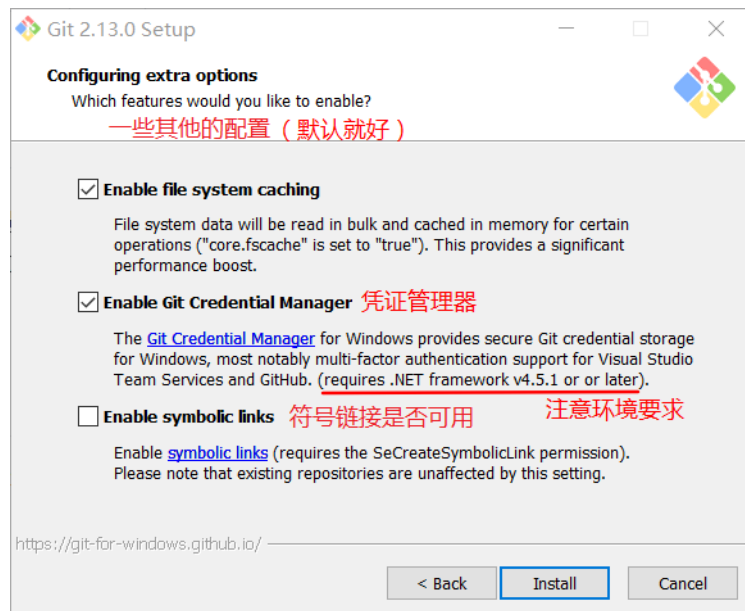
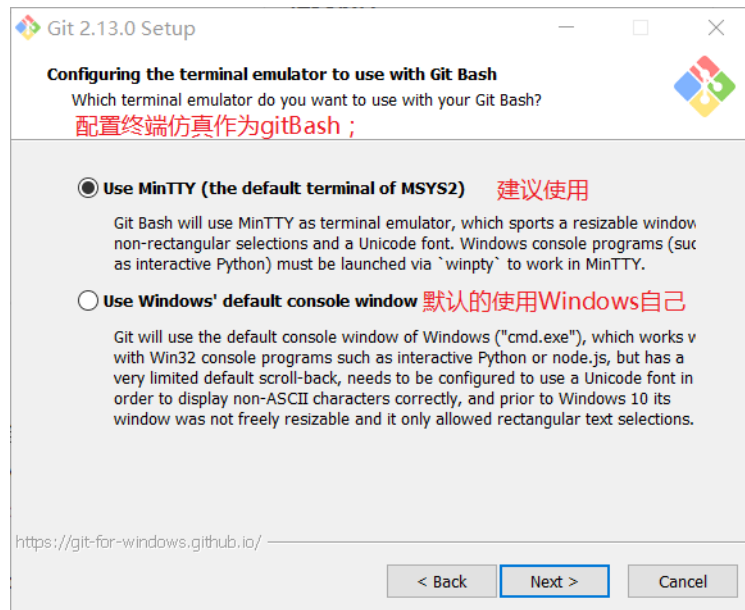


在这里选择 notepad++作为默认的文本编辑器，提交代码时如果没有-m 选项，则会用这里选的文本编辑器打开一个文件来写入我们对本次提交的说明信息。



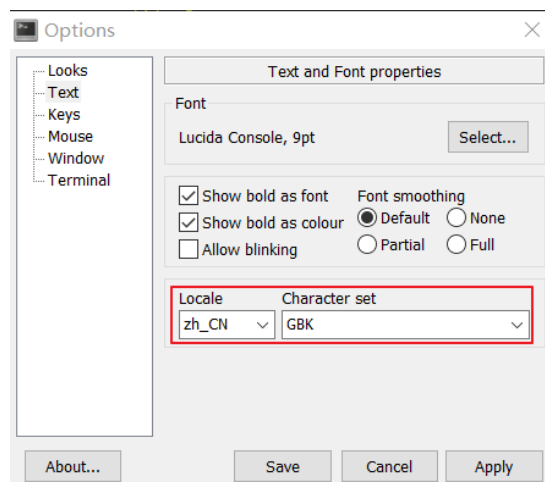






然后，等待安装完成。

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。



### 3. 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++

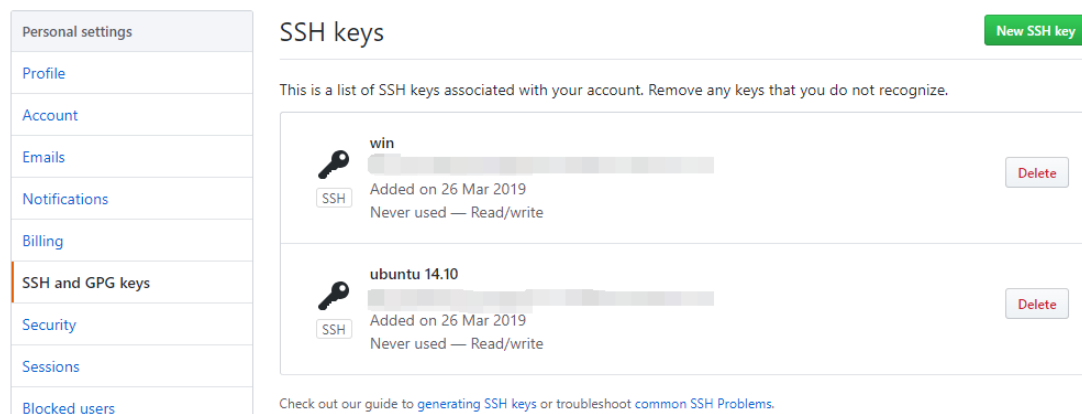
```
git config --global core.editor "C:\Program Files\Notepad++\notepad++.exe -multiInst -notabbar -nosession -noPlugin"
```

#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
```

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
```

```
qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (master)
$ ssh -T git@github.com
The authenticity of host 'github.com (13.229.188.59)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,13.229.188.59' (RSA) to the list of known
hosts.
Hi 2076940762! You've successfully authenticated, but GitHub does not provide sh
ell access.
qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (master)
```

## 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 notepadqq 方法:

```
sudo add-apt-repository ppa:notepadqq-team/notepadqq  
sudo apt-get update  
sudo apt-get install notepadqq
```

Ubuntu 下的卸载 notepadqq 方法:

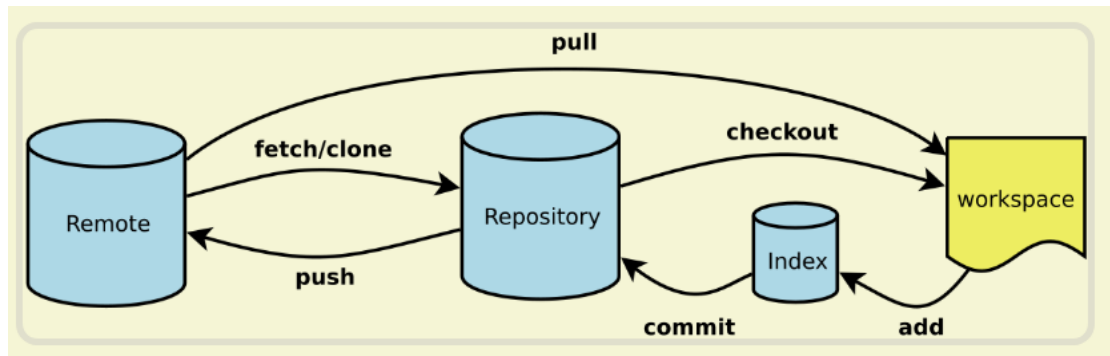
```
sudo apt-get remove notepadqq  
sudo add-apt-repository --remove ppa:notepadqq-team/notepadqq
```

#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
```

```
qingtian@ubuntu:~/eclipse-java-2019-03-M3-win32-x86_64/eclipse$ git init
Initialized empty Git repository in /home/qingtian/eclipse-java-2019-03-M3-win32-x86_64/eclipse/.git/
qingtian@ubuntu:~/eclipse-java-2019-03-M3-win32-x86_64/eclipse$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .eclipseproduct
        artifacts.xml
        configuration/
        eclipse.exe
        eclipse.exe-1552752341715.p2bu
        eclipse.ini
        eclipsesec.exe
        features/
        p2/
        plugins/
        readme/

nothing added to commit but untracked files present (use "git add" to track)
qingtian@ubuntu:~/eclipse-java-2019-03-M3-win32-x86_64/eclipse$
```

# 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

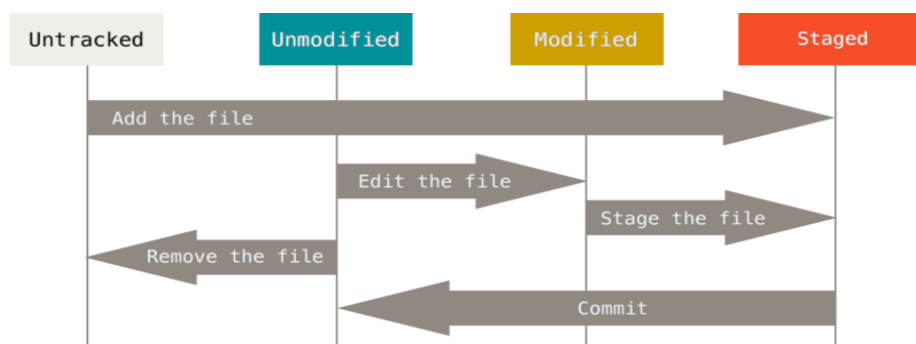
```
$ git clone [url]
```

#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
```

```

qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   excelDocRW.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   excelDocRW.java
    modified:   wordWrite.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    docxRead.java
  
```

**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

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git add .gitignore

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git status -s
A .gitignore
MM excelDocRW.java
M wordWrite.java
?? docxRead.java
```

??：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

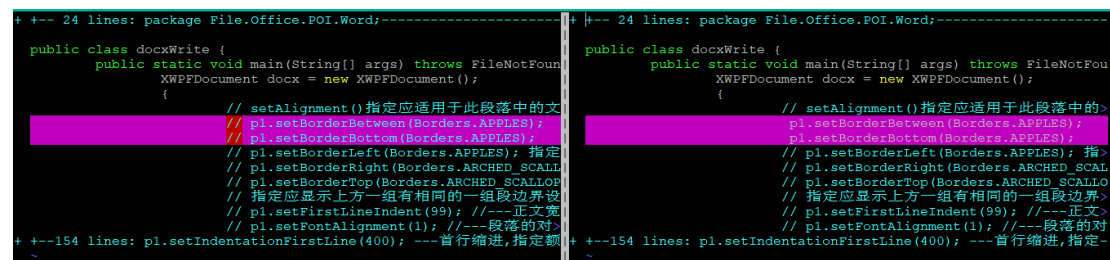
#此命令比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。

\$ 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。



```
+ +-+ 24 lines: package File.Office.POI.Word;-----+ +-+ 24 lines: package File.Office.POI.Word;-----
public class docxWrite {
    public static void main(String[] args) throws FileNotFoun
        XWPFDocument docx = new XWPFDocument();
        {
            // setAlignment() 指定应适用于此段落中的文
            pl.setBorderBetween(Borders.APPLES);
            pl.setBorderBottom(Borders.APPLES);
            pl.setBorderLeft(Borders.APPLES); 指定
            pl.setBorderRight(Borders.ARCHED_SCALL
            pl.setBorderTop(Borders.ARCHED_SCALLOP
            指定应显示上方一组有相同的一组段边界
            pl.setFirstLineIndent(99); ---正文宽
            pl.setPonAlignment(1); ---段落的对
+ +-+154 lines: pl.setIndentationFirstLine(400); ---首行缩进, 指定额
+ +-+154 lines: pl.setIndentationFirstLine(400); ---首行缩进, 指定-
```

# 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]
```



```

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ ls
docxWrite.java  excelDocRW.java  writeDocxTemplate.java

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git rm --cached excelDocRW.java
rm 'excelDocRW.java'

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    docxRead.java
        deleted:    excelDocRW.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        excelDocRW.java

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)

```

#重命名文件，并且将这个改名放入暂存区

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

```

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ ls
docxWrite.java  excelDocRW.java  writeDocxTemplate.java

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git mv docxWrite.java wordWrite.java

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ ls
excelDocRW.java  wordWrite.java  writeDocxTemplate.java

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    docxRead.java
        deleted:    excelDocRW.java
        renamed:    docxWrite.java -> wordWrite.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        excelDocRW.java

gingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)

```

## 撤销

## 重置

HEAD 是当前分支引用的指针，它总是指向该分支上的最后一次提交。这表

---

示 HEAD 将是下一次提交的父结点。

索引(暂存区)是你的 预期的下一次提交。

工作目录会将它们解包为实际的文件以便编辑。你可以把工作目录当做 沙盒。在你将修改提交到暂存区并记录到历史之前，可以随意更改。

reset 命令会以特定的顺序重写这三棵树，在你指定以下选项时停止：

1. 移动 HEAD 分支的指向 （若指定了 --soft ，则到此停止）
2. 使索引看起来像 HEAD （若未指定 --hard ，则到此停止）
3. 使工作目录看起来像索引

# 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

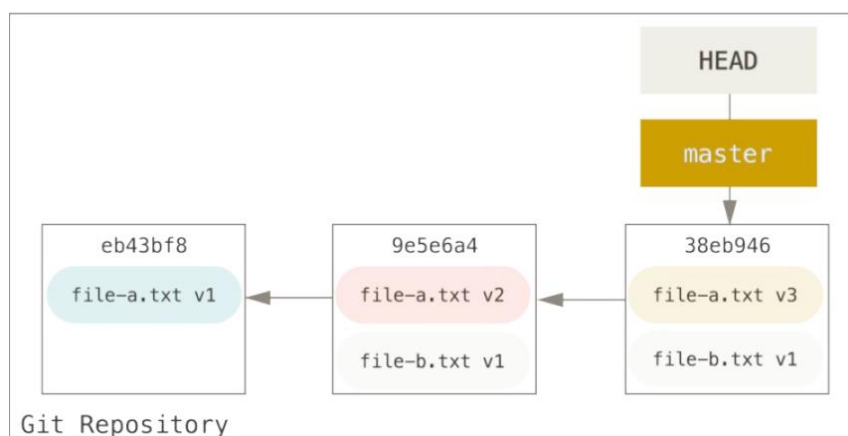
```
$ git reset --hard [commit]
```

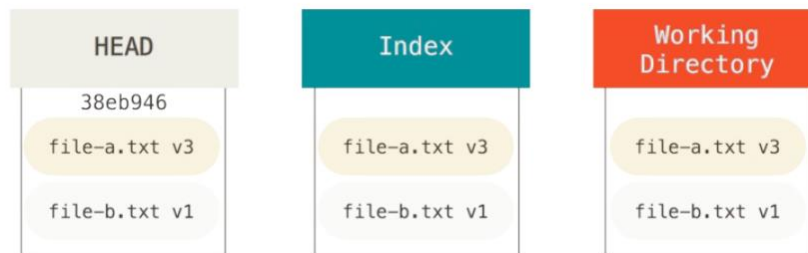
# 重置当前 HEAD 为指定 commit，但保持暂存区和工作区不变

```
$ git reset --keep [commit]
```

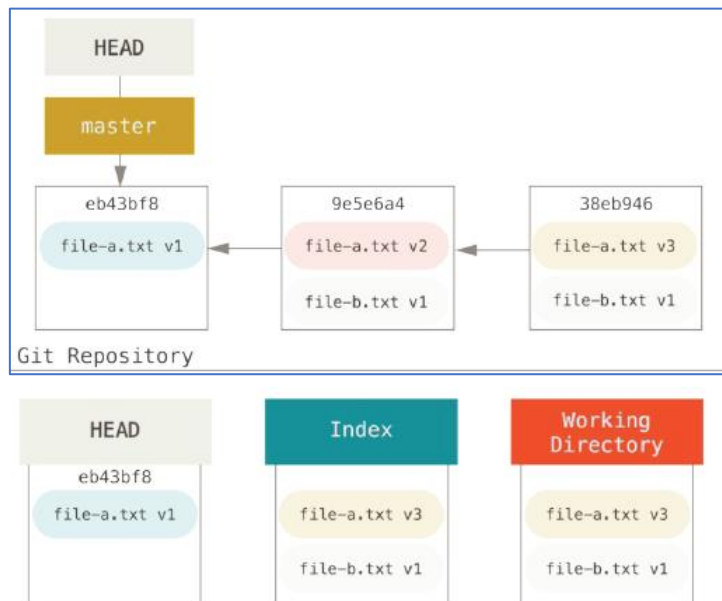
## 将多个提交压缩

假设你有一个项目，第一次提交中有一个文件，第二次提交增加了一个新的文件并修改了第一个文件，第三次提交再次修改了第一个文件。 由于第二次提交是一个未完成的工作，因此你想要压缩它。



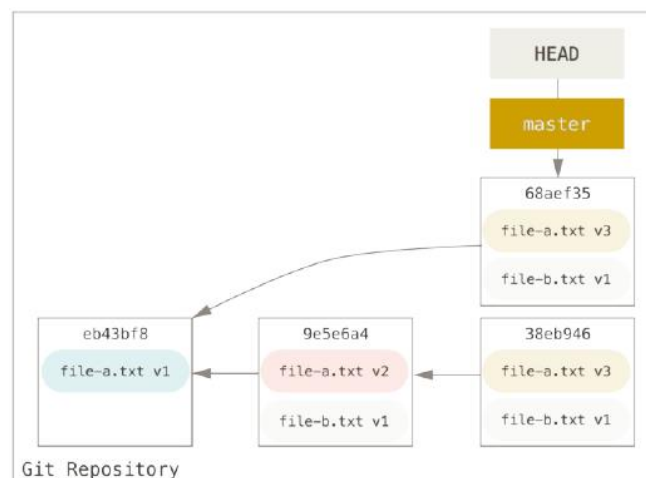


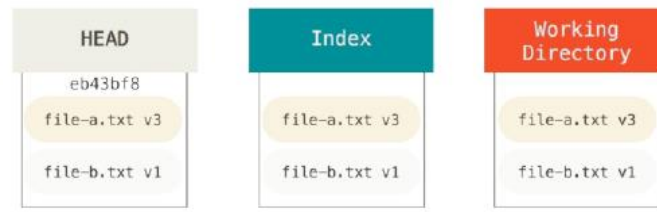
那么可以运行 `git reset --soft HEAD~2` 来将 HEAD 分支移动到一个旧一点的提交上（即你想要保留的第一个提交）：



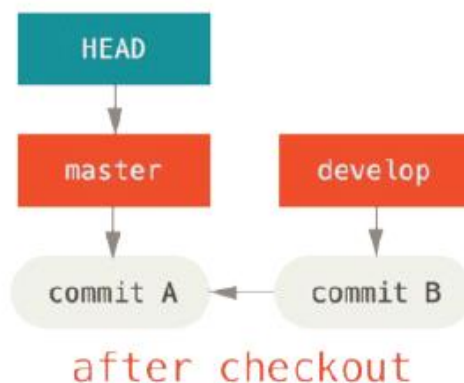
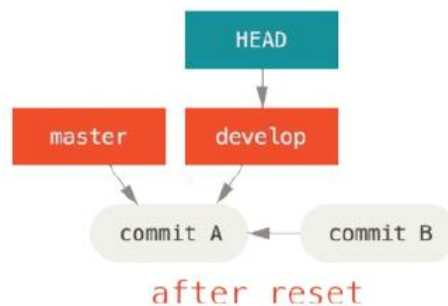
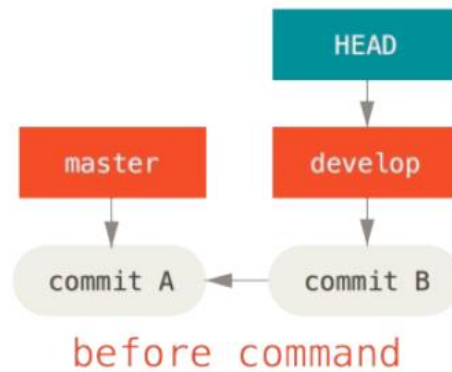
#这个命令会将暂存区中的文件提交。如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息。

然后只需再次运行 `git commit` :





运行 `git checkout [branch]` 与运行 `git reset --hard [branch]` 非常相似，不过有两点重要的区别。首先不同于 `reset --hard`，`checkout` 对工作目录是安全的，它会通过检查来确保不会将已更改的文件吹走。其实它还更聪明一些。它会在工作目录中先试着简单合并一下，这样所有还未修改过的文件都会被更新。而 `reset --hard` 则会不做检查就全面地替换所有东西。第二个重要的区别是如何更新 HEAD。reset 会移动 HEAD 分支的指向，而 `checkout` 只会移动 HEAD 自身来指向另一个分支。



---

checkout <文件名> 它就像 `git reset [branch] file` 那样不会移动 HEAD, 用该次提交中的那个文件来更新索引, 但是它也会覆盖工作目录中对应的文件。

# 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 .
```

# 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]
```

#重新提交

```
$ 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
```

## 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] ...
```

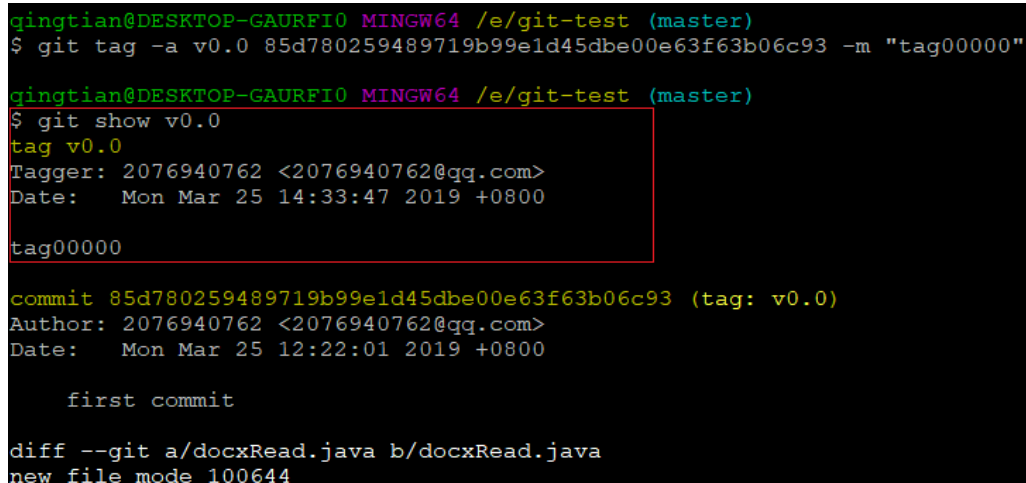
## 标签

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"
```



```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git tag -a v0.0 85d780259489719b99e1d45dbe00e63f63b06c93 -m "tag00000"

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git show v0.0
tag v0.0
Tagger: 2076940762 <2076940762@qq.com>
Date: Mon Mar 25 14:33:47 2019 +0800
tag00000

commit 85d780259489719b99e1d45dbe00e63f63b06c93 (tag: v0.0)
Author: 2076940762 <2076940762@qq.com>
Date: Mon Mar 25 12:22:01 2019 +0800

    first commit

diff --git a/docxRead.java b/docxRead.java
new file mode 100644
```

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>
```

```
qingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (branch002)
$ git show tag123
commit 637b72b093d936ee35815afb2acc5e42d40b1fld (HEAD -> branch002, tag: tag123,
tag: qing1)
Author: 2076940762 <2076940762@qq.com>
Date:   Wed Mar 27 21:55:20 2019 +0800

    deleted

diff --git a/docxRead.java b/docxRead.java
deleted file mode 100644
index 789aba0..0000000
--- a/docxRead.java
+++ /dev/null
@@ -1,92 +0,0 @@
-package File.Office.POI.Word;
```

# 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。

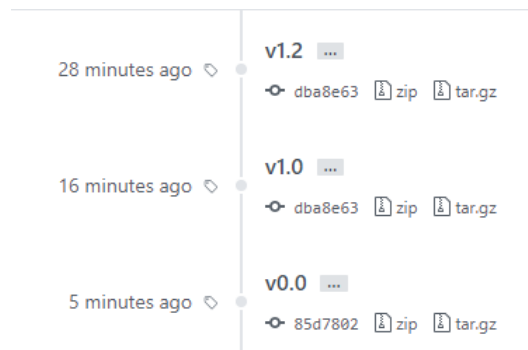
```
$ git push origin --tags
```

```
$ git push [remote] --tags
```

```

qingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git push --tags
Enumerating objects: 2, done.
Counting objects: 100% (2/2), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 268 bytes | 89.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/2076940762/Git-test.git
* [new tag]          v0.0 -> v0.0
* [new tag]          v1.0 -> v1.0
* [new tag]          v1.2 -> v1.2

```



# 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
<code>%H</code>	提交对象 (commit) 的完整哈希字符串
<code>%h</code>	提交对象的简短哈希字符串
<code>%T</code>	树对象 (tree) 的完整哈希字符串
<code>%t</code>	树对象的简短哈希字符串
<code>%P</code>	父对象 (parent) 的完整哈希字符串
<code>%p</code>	父对象的简短哈希字符串
<code>%an</code>	作者 (author) 的名字
<code>%ae</code>	作者的电子邮件地址
<code>%ad</code>	作者修订日期 (可以用 <code>--date=</code> 选项定制格式)
<code>%ar</code>	作者修订日期，按多久以前的方式显示
<code>%cn</code>	提交者 (committer) 的名字
<code>%ce</code>	提交者的电子邮件地址
<code>%cd</code>	提交日期
<code>%cr</code>	提交日期，按多久以前的方式显示
<code>%s</code>	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
<code>-(n)</code>	仅显示最近的 n 条提交
<code>--since, --after</code>	仅显示指定时间之后的提交。
<code>--until, --before</code>	仅显示指定时间之前的提交。
<code>--author</code>	仅显示指定作者相关的提交。
<code>--committer</code>	仅显示指定提交者相关的提交。
<code>--grep</code>	仅显示含指定关键字的提交
<code>-S</code>	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]
```

#只有当你有所克隆服务器的写入权限, 并且之前没有人推送过时, 这条命令才能生效。当你和其他人在同一时间克隆, 他们先推送到上游然后你再推送到上游, 你的推送就会毫无疑问地被拒绝。你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。

```
$ git push [remote-name] [branchname]
```

# 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

### 分支简介

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。

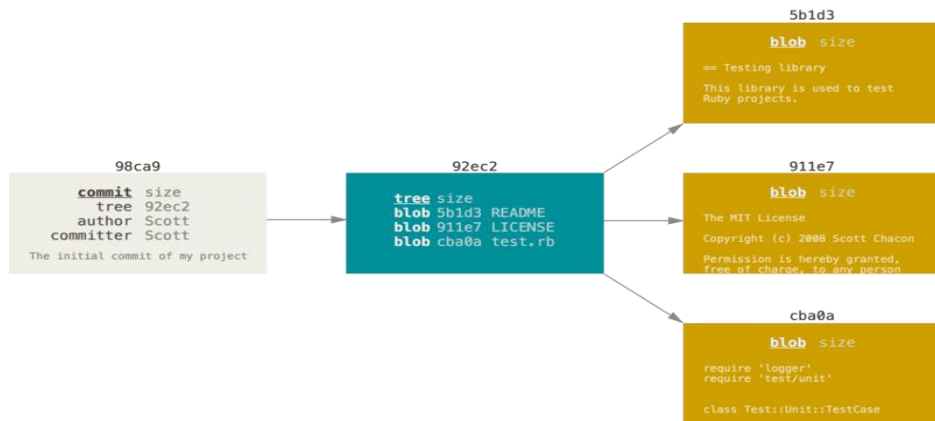
Git 保存的不是文件的变化或者差异, 而是一系列不同时刻的**文件快照**。在进行提交操作时, Git 会保存一个**提交对象** (commit object)。该提交对象会包含一个指向暂存内容快照的指针。除此之外, 该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。首次提交产生的提交对象没有父对象, 普通提交操作产生的提交对象有一个父对象, 而由多个分支合并产生的提交对象有多个父对象。暂存操作会为每一个文件计算校验和 (使用 SHA-1 哈希算法), 然后会把当前版本的文件快照保存到 Git 仓库中 (Git 使用 blob 对象来保存它们), 最终将校验和加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

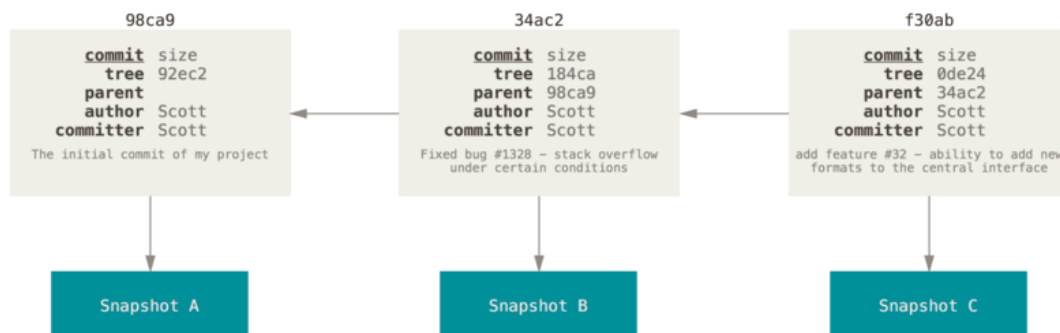
举个例子: 设现在有一个工作目录, 里面包含了三个将要被暂存和提交的文件 (README test.rb LICENSE)。将它们提交到本地仓库,

```
$ git add README test.rb LICENSE  
$ git commit -m 'The initial commit of my project'
```

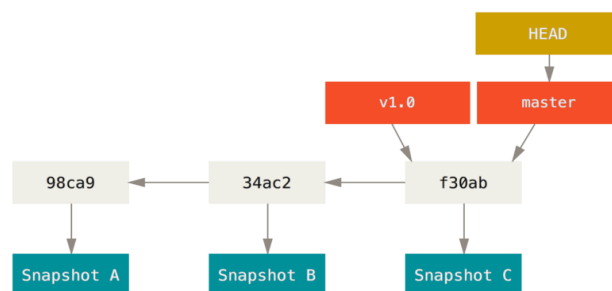
现在, Git 仓库中有五个对象: 三个 blob 对象 (保存着文件快照)、一个树对象 (记录着目录结构和 blob 对象索引) 以及一个提交对象 (包含着指向前述树对象的指针和所有提交信息)。



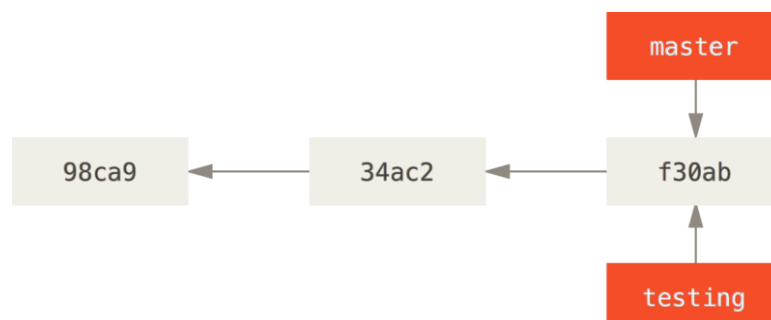
做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



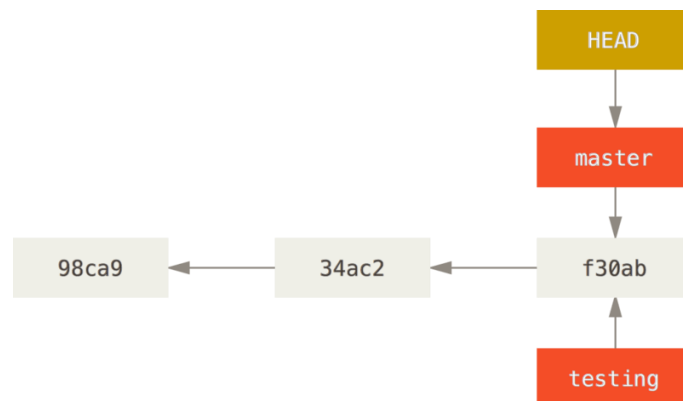
Git 的分支，其实本质上仅仅是指向提交对象的可变指针。Git 的默认分支名字是 `master`。在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 `master` 分支。它会在每次的提交操作中自动向前移动。



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）。

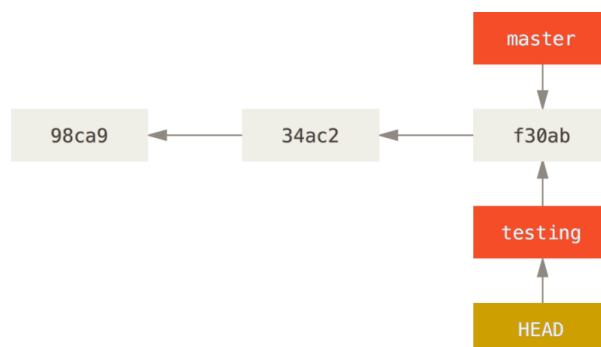


可以使用 `git log` 或者 `git branch` 查看当前分支。

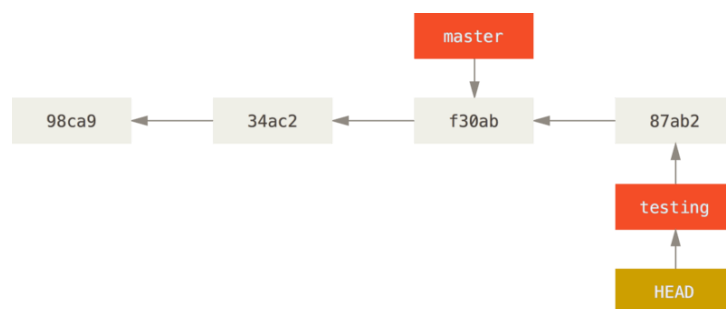
```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git log
commit 637b72b093d936ee35815afb2acc5e42d40b1f1d (HEAD -> branch002, tag: tag123, tag: qing1)
Author: 2076940762 <2076940762@qq.com>
Date: Wed Mar 27 21:55:20 2019 +0800
```

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git branch
branch-v0.0
branch-v1.0
branch001
* branch002
master
```

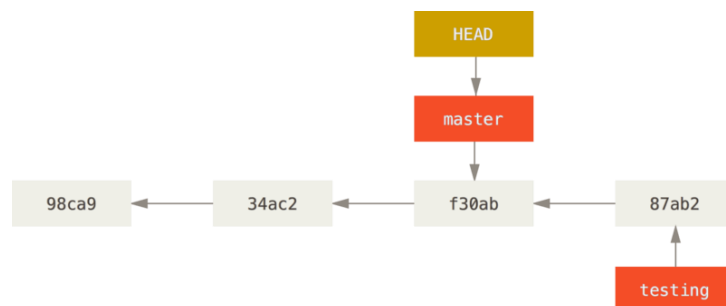
如果我们这时使用切换到分支 `testing`，这时 HEAD 指针会指向 `testing` 分支。



然后，我们在 `testing` 分支修改文件并提交，HEAD 分支随着提交操作自动向前移动。此时，仓库状态如下：



如果这时我们再次切回到分支 master, 得到的状态如下:



git checkout master 命令做了两件事。一是使 HEAD 指回 master 分支, 二是将工作目录恢复成 master 分支所指向的快照内容。也就是说, 你现在做修改的话, 项目将始于一个较旧的版本。本质上来讲, 这就是忽略 testing 分支所做的修改, 以便于向另一个方向进行开发。

!!! 分支切换会改变你工作目录中的文件。在切换分支时, 一定要注意你工作目录里的文件会被改变。如果是切换到一个较旧的分支, 你的工作目录会恢复到该分支最后一次提交时的样子。

1. 工作目录中有未跟踪文件时切换分支, 不会导致未跟踪文件丢失;

```
qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 4 commits.
(use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    docxRead.java

nothing added to commit but untracked files present (use "git add" to track)
qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (branch002)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 4 commits.
(use "git push" to publish your local commits)

qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (master)
$ ls
docxRead.java  writeDocxTemplate.java

qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (master)
$ git checkout -
Switched to branch 'branch002'
Your branch is ahead of 'origin/master' by 4 commits.
(use "git push" to publish your local commits)
```



```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ ls
docxRead.java  writeDocxTemplate.java

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git checkout -
Switched to branch 'branch002'
Your branch is ahead of 'origin/master' by 4 commits.
(use "git push" to publish your local commits)

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ ls
docxRead.java  excelDocRW.java  wordWrite.java  writeDocxTemplate.java

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 4 commits.
(use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        docxRead.java

nothing added to commit but untracked files present (use "git add" to track)

```

2. 有首次跟踪的文件处于暂存区时（刚刚将未跟踪的文件 add 到暂存区）切换分支也不会丢失暂存区文件；

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   docxRead.java

```

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git checkout -
Switched to branch 'master'
A       docxRead.java
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ ls
docxRead.java  writeDocxTemplate.java

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git checkout -
Switched to branch 'branch002'
A       docxRead.java
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   docxRead.java

```

3. 工作区有已修改状态的文件时禁止分支切换

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   excelDocRW.java

no changes added to commit (use "git add" and/or "git commit -a")

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git checkout -
error: Your local changes to the following files would be overwritten by checkout:
        excelDocRW.java
Please commit your changes or stash them before you switch branches.
Aborting

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)

```

4. 已经提交过的文件再次修改后处于暂存区时禁止分支切换:

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git status
On branch branch002
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   excelDocRW.java

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
        excelDocRW.java
Please commit your changes or stash them before you switch branches.
Aborting

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)

```

总之，在切换分支前先 `git commit -a` 将所有本地修改提交到仓库。

## 分支管理常用命令

# 列出所有本地分支，\*表示当前分支

\$ `git branch`

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git branch
branch-v0.0
branch-v1.0
branch001
branch002
* master

```

# 只显示已经合并到当前分支的分支

\$ `git branch --merged`

# 只显示没有合并到当前的分支

\$ `git branch --no-merged`

# 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]
```

#用指定标签对应的快照创建新分支

# If -B is given, <new\_branch> is created if it doesn't exist; otherwise, it is reset.

```
$ git checkout -b <分支名> <标签名>
```

跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 `git pull`, Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

# 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]
```

#设置上游分支

```
$ git branch -u origin/上游分支名
```

#查看所有的跟踪分支

# When in list mode, show sha1 and commit subject line for each head, along with relationship to upstream branch (if any). If given twice, print the name of the upstream branch, as well

```
$ git branch -vv
```

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git branch -v
  branch-v0.0 5d0447e adfa
  branch-v1.0 6e44f7f 1111
  branch001   941a2fa b001
* branch002   dad3f54 [ahead 6] aa
  master      be51f24 [ahead 4] Merge branch 'master' of https://github.com/2076940762/Git-test

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git branch -vv
  branch-v0.0 5d0447e adfa
  branch-v1.0 6e44f7f 1111
  branch001   941a2fa b001
* branch002   dad3f54 [origin/master: ahead 6] aa
  master      be51f24 [origin/master: ahead 4] Merge branch 'master' of https://github.com/2076940762/Git-test
```

#获得远程引用的完整列表

```
git ls-remote
```

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)
$ git remote show origin
* remote origin
Fetch URL: https://github.com/2076940762/Git-test.git
Push URL: https://github.com/2076940762/Git-test.git
HEAD branch: master
Remote branches:
  bran002      tracked
  branch-v0.0 tracked
  branch-v1.0 tracked
  branch001   tracked
  master      tracked
  newBranch   tracked
Local branches configured for 'git pull':
  branch002 merges with remote master
  master    merges with remote master
Local refs configured for 'git push':
  branch-v0.0 pushes to branch-v0.0 (up to date)
  branch-v1.0 pushes to branch-v1.0 (up to date)
  branch001   pushes to branch001   (up to date)
  master      pushes to master      (up to date)

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch002)

```

#Gives some information about the remote <name>.

\$ git remote show origin

#如果本地分支 sf 不存在，则用远程分支 origin/serverfix 创建本地分支 sf. 如果本地分支存在则重置 sf 分支。

\$ git checkout -b sf origin/serverfix

# 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，如果两个分支修改有冲突，需要手动解决冲突后 git add .

\$ 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]

本地的分支并不会自动与远程仓库同步，你必须显式地推送想要分享的分支。

#推送本地分支到远端

\$ git push origin 本地分支名

#讲本地分支推送到远端并重命名

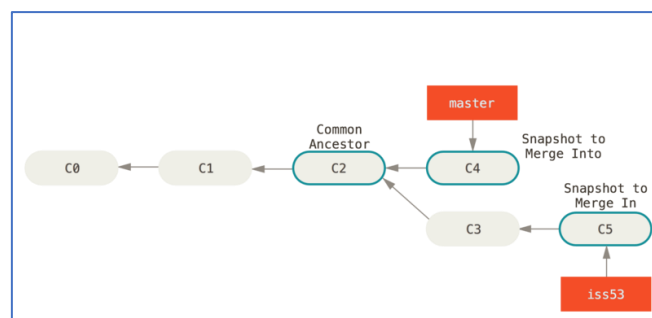
\$ git push origin 本地分支名:远程仓库分支名

```
qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (branch002)
$ git push origin branch002:bran002
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 280 bytes | 93.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'bran002' on GitHub by visiting:
remote:   https://github.com/2076940762/Git-test/pull/new/bran002
remote:
To https://github.com/2076940762/Git-test.git
 * [new branch]      branch002 -> bran002
qingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (branch002)
```

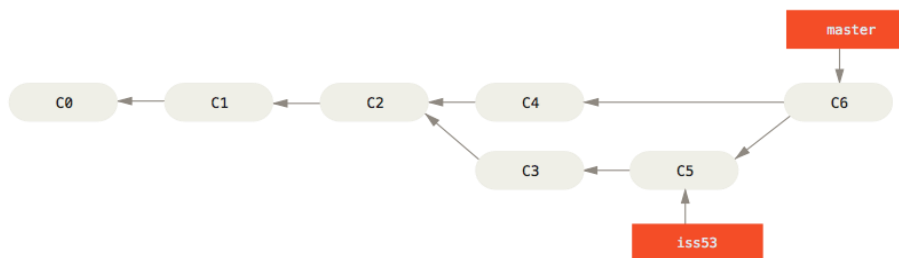
## 合并与变基

### 一、分支合并

假设你在 `iss53` 分支解决了一个问题，现在要讲修改合并到 `master` 分支。只需要切换到分支 `master`，然后执行 `git merge iss53` 命令即可。`master` 分支所在提交并不是 `iss53` 分支所在提交的直接祖先。这时，会使用两个分支的末端所指的快照（`C4` 和 `C5`）以及这两个分支的工作祖先（`C2`），做一个简单的三方合并。



Git 将此次三方合并的结果做了一个新的快照（`C6`）并且自动创建一个新的提交指向它。这个被称作一次合并提交，它的特别之处在于他有不止一个父提交。



合并完成后，可以删除 iss53 分支。

## 解决合并冲突

如果要合并的两个分支有冲突，比如两个分支同时修改了同一个文件的相同部分，Git 会做合并，但是不会自动提交创建的合并。这时需要我们手动解决冲突。解决后使用 `git add <file>` 命令将未合并文件，标记为冲突已解决。最后，`git commit` 提交合并结果。

```

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch-v0.0)
$ git merge master
Auto-merging writeDocxTemplate.java
CONFLICT (content): Merge conflict in writeDocxTemplate.java
Automatic merge failed; fix conflicts and then commit the result.

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch-v0.0|MERGING)
$ git status
On branch branch-v0.0
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   writeDocxTemplate.java

no changes added to commit (use "git add" and/or "git commit -a")

qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (branch-v0.0|MERGING)
$
  
```

```

public static void main(String[] args) {
    //11111111111111111111
    //22222222222222222222
    <<<<<< HEAD
    //*****????????????
    //33333333333333333333
    =====
    //*****
    //33333333333333333333
    >>>>>> master

    //aaaaaaaaaaaaaa
    //bbbbbbbbbbbbbbbbbb
    //cccccccccccccccccc
  
```

## 手动解决冲突：

选择<<<<<< ， ===== ， >>>>>>中=====上面或者下面保存，删除其余，包括<<<<<, =====, >>>>>>。

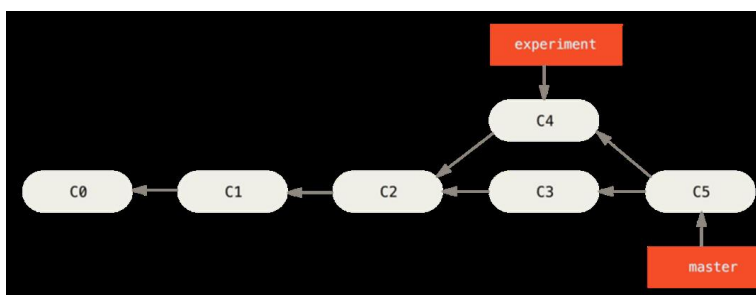
## 二、变基

Git 中合并其他分支的修改到当前分支的方法除了前面的 merge 外还有 rebase. 变基的原理是首先找到这两个分支的最近共同祖先 C2, 然后对比当前分支相对于该祖先的历次提交, 提取相应的修改并存为临时文件, 然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用。相对 merge 变基结果的提交历史更简洁一些。

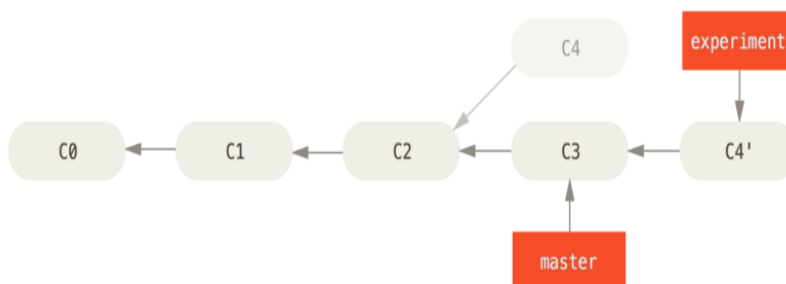
```
$ git checkout experiment
```

```
$ git rebase master
```

Merge 结果如下:



变基结果如下:



此时, C4' 指向的快照就和上面使用 merge 命令的例子中 C5 指向的快照一模一样了。这两种整合方法的最终结果没有任何区别, 但是变基使得提交历史更加整洁。经过变基的分支的历史记录, 看上去就像是穿行的一样。

一般我们这样做的目的是为了确保在向远程分支推送时能保持提交历史的整洁——例如向某个其他人维护的项目贡献代码时。在这种情况下, 你首先在自己的分支里进行开发, 当开发完成时你需要先将你的代码变基到 origin/master 上, 然后再向主项目提交修改。这样的话, 该项目的维护者就不再需要进行整合工作, 只需要快进合并便可。

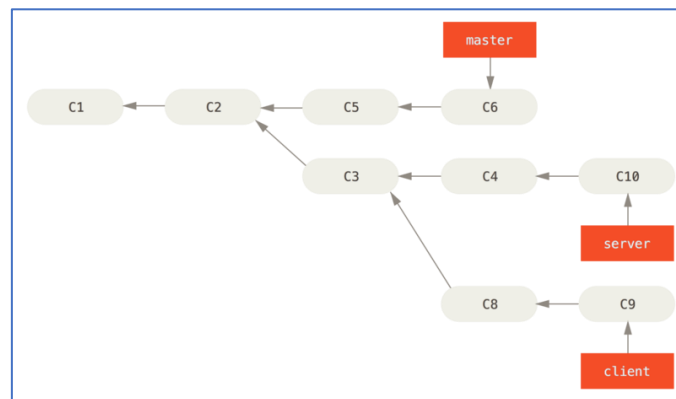
```
#将 server 分支中的修改变基到 master 分支上
```

```
$ git rebase master server
```

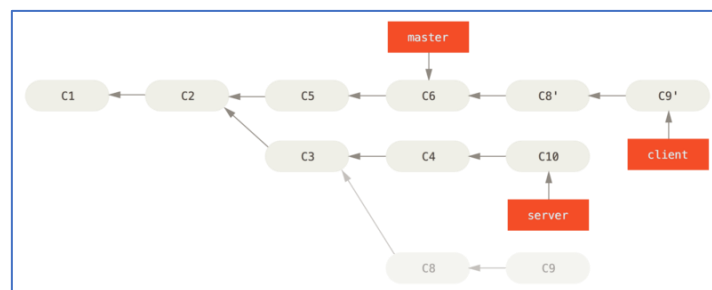
```
#选中在 client 分支里但不在 server 分支里的修改 (即 C8 和 C9), 将它们在 master
```

分支上重放

```
$ git rebase --onto master server client
```

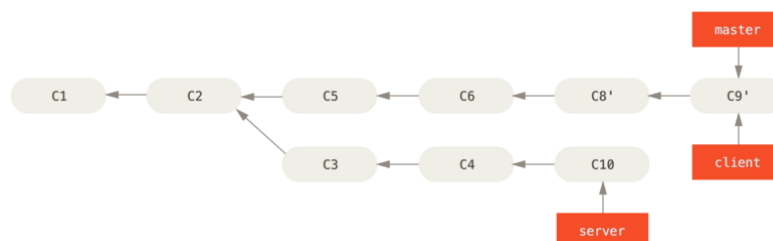


变基结果：



```
$ git checkout master
```

```
$ git merge client
```



变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。**不要对在你的仓库外有副本的分支执行变基。**如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用 `git rebase` 命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

#如果有人强行对已经上传到服务器的分支变基，覆盖了服务器上的一些提交历史，而你的工作正好是基于这些提交的，请执行如下命令：

```
git pull --rebase
```

变基完成后，在推送服务器前将不用的分支删除。

与合并一样，如果修改有冲突，则需要手动解决冲突。



---

```
git add <文件名>
git rebase --continue
```

## 第三节 其它命令

#查看某个分支当前指向的 SHA-1 值

```
$ git rev-parse <分支名>
```

Git 会在后台保存一个引用日志(reflog), 引用日志记录了最近几个月你的 HEAD 和分支引用所指向的历史。每当你的 HEAD 所指向的位置发生了变化, Git 就会将这个信息存储到引用日志这个历史记录里。通过这些数据, 你可以很方便地获取之前的提交历史。引用日志只存在于本地仓库。

#来查看引用日志

```
$ git reflog
```

```
yingtian@DESKTOP-GAURF10 MINGW64 /e/Document (master)
$ git reflog
6906fd0 (HEAD -> master, origin/master) HEAD@{0}: commit: backup
61d9606 HEAD@{1}: commit: backup git doc
fb0feec HEAD@{2}: commit: a
1d131fd HEAD@{3}: commit: <E5><BF><BD><E7><95><A5><E6><96><87><E4><BB><B6>
bfaca65 HEAD@{4}: commit: <E5><A4><87><E4><BB><BD>
b86f604 HEAD@{5}: commit: <E5><A4><87><E4><BB><BD> git <E5><BF><AB><E9><80><9F>
<E5><85><A5><E9><97><A8>
a06c128 HEAD@{6}: commit: <E4><B8><8A><E4><BC><A0><E5><A4><A7><E6><96><87><E4>
<BB><B6><E3><80><8A>linux <E5><86><85><E6><A0><B8><E3><80><8B>
9641c83 HEAD@{7}: commit: <E5><A4><A7><E6><96><87><E4><BB><B6>
29219d8 HEAD@{8}: commit: <E6><B7><BB><E5><8A><A0><E5><85><A8><E9><83><A8><E6>
<96><87><E4><BB><B6>
0a5941b HEAD@{9}: commit: <E8><B7><9F><E8><B8><AA><E5><A4><A7><E6><96><87><E4>
<BB><B6>
d0b94cf HEAD@{10}: commit (initial): first commit
```

# HEAD 在五次前的所指向的提交

```
$ git show HEAD@{5}
```

#查看类似于 git log 输出格式的引用日志信息

```
$ git log -g
```

#查看上一次提交

```
$ git show HEAD^
```

#查看 experiment 分支中还有哪些提交尚未被合并入 master 分支。

```
$ git log master..experiment
```

#查看你即将推送到远端的内容

```
$ git log origin/master..HEAD
```

#查看所有被 refA 分支 或 refB 包含的但是不被 refC 包含的提交

```
$ git log refA refB ^refC
```

```
$ git log refA refB --not refC
```

#查看 master 或者 experiment 中包含的但不是两者共有的提交。--left-right , 它会显示每个提交到底处于哪一侧的分支

```
$ git log --left-right master...experiment
```

#交互式提交

```
$ git add -i
```

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git add -i

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now>
```

从交互式提示符中, 输入 5 或 p (补丁)。Git 会询问你想要部分暂存哪些文件; 然后, 对已选择文件的每一个部分, 它都会一个个地显示文件区别并询问你是否想要暂存它们。

git add -p 或 git add --patch 来启动补丁方式暂存

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git add -p
diff --git a/writeDocxTemplate.java b/writeDocxTemplate.java
index d4e1a52..803bbf7 100644
--- a/writeDocxTemplate.java
+++ b/writeDocxTemplate.java
@@ -4,10 +4,8 @@ public class writeDocxTemplate {
     public static void main(String[] args) {
         //11111111111111111111
         //22222222222222222222
         //*****
         //33333333333333333333
-
-
-
+
+
+
         //aaaaaaaaaaaaaa
         //bbbbbbbbbbbbbbbbbbbb
         //cccccccccccccccccc
Stage this hunk [y,n,q,a,d,s,e,?]?
```

你在当前分支工作一段时间后, 有更为紧迫的问题需要处理, 你不得不切换分支, 但因为你还没有完成工作不想提交, 这时可以 git stash 会把所有未提交的修改(包括暂存的和非暂存的)都保存起来, 用于后续恢复当前工作目录。stash 是本地的, 不会通过 git push 命令上传到 git server 上。

#储藏当前工作

```
$ git stash
```

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git stash
Saved working directory and index state WIP on master: be51f24 Merge branch 'master' of https://github.com/2076940762/Git-test
```

默认情况下, git stash 会缓存下列文件:

- 添加到暂存区的修改 (staged changes)

- Git 跟踪的但并未添加到暂存区的修改 (unstaged changes)

但不会缓存一下文件:

- 在工作目录中新的文件 (untracked files)

- 被忽略的文件 (ignored files)

---

`git stash` 命令提供了参数用于缓存上面两种类型的文件。使用 `-u` 或者 `--include-untracked` 可以 `stash untracked` 文件。使用 `-a` 或者 `--all` 命令可以 `stash` 当前目录下的所有修改。

当你在一个分支上完成工作后，返回上一个分支 (`git checkout -`)，然后用 `git stash apply` 命令恢复刚才储藏的工作。

如果你储藏了一些工作，暂时不去理会，然后继续在你储藏工作的分支上工作，你在重新应用工作时可能会碰到一些问题。如果尝试应用的变更是针对一个你那之后修改过的文件，你会碰到一个归并冲突并且必须去化解它。如果你想用更方便的方法来重新检验你储藏的变更，你可以运行 `git stash branch <新分支名>`，这会创建一个新的分支，检出你储藏工作时的所处的提交，重新应用你的工作，如果成功，将会丢弃储藏。

#将缓存堆栈中的第一个 stash 删除，并将对应修改应用到当前的工作目录下。

# Applying the state can fail with conflicts; in this case, it is not removed from the stash list. You need to resolve the conflicts by hand and call "git stash drop" manually afterwards.

```
$ git stash pop
```

#恢复上上一次储藏的工作，但不会删除缓存堆栈中的 stash

```
$ git stash apply stash@{2}
```

# 移除储藏

```
$ git stash drop stash@{0}
```

#查看储藏

```
$ git stash list
```

```
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
$ git stash list
stash@{0}: WIP on master: be51f24 Merge branch 'master' of https://github.com/2076940762/Git-test
stash@{1}: WIP on master: be51f24 Merge branch 'master' of https://github.com/2076940762/Git-test
qingtian@DESKTOP-GAURFI0 MINGW64 /e/git-test (master)
```

# 删除所有储藏

```
$ git stash clear
```

# If the `--index` option is used, then tries to reinstate not only the working tree's changes, but also the index's ones. However, this can fail, when you have conflicts (which are stored in the index, where you therefore can no longer apply the changes as they were originally).

```
$ git stash apply --index
```

`--keepindex` 选项。它告诉 Git 不要储藏任何你通过 `git add` 命令已暂存的东西。

`--include-untracked` 或 `-u` 标记，Git 也会储藏任何创建的未跟踪文件。

---

#从储藏创建分支

\$ git stash branch testchanges

#删除所有没有忽略的未跟踪的文件

\$ git clean

-x 删除被忽略的文件

-i 交互式删除

```
mingtian@DESKTOP-GAURF10 MINGW64 /e/git-test (master)
$ git clean -i
Would remove the following item:
123
*** Commands ***
1: clean          2: filter by pattern  3: select by numbers
4: ask each       5: quit              6: help
What now>
```

#命令来移除工作目录中所有未追踪的文件以及空的子目录。

\$ git clean -f -d

\$ git stash --all 来移除每一样东西并存放在栈中。

---

## 第三章 Git 服务器

在 Git 中，你完全可以将一起合作的某一个个人仓库设置为远端，通过从该个人仓库 push 和 pull 来修改内容。为了代码管理方便，搭建一个服务器可能会更好一点。

一个远程仓库通常只是一个裸仓库 (bare repository) — 即一个没有当前工作目录的仓库。因为该仓库仅仅作合作媒介，不需要从磁碟检查快照；存放的只有 Git 的资料。简单的说，裸仓库就是你工程目录内的 .git 子目录内容，不包含其他资料。远程仓库目录名称可以为任何合法的路径名，一般情况下使用“.git”结尾。

```
#把现有仓库导出为裸仓库——即一个不包含当前工作目录的仓库，  
$ git clone --bare my_project my_project.git
```

### 第一节 Git 支持的协议

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 (SecureShell) 协议及 Git 协议。

#### 本地协议

本地协议就是将本地磁盘上另一个目录作为远程仓库来使用。

```
#增加一个本地版本库到现有的 Git 项目  
$ git remote add local_proj /opt/git/project.git
```

然后，就可以像在网络上一样从远端版本库推送和拉取更新了。

```
#克隆仓库  
$ git clone /opt/git/project.git
```

配置简单适合一个人开发的小项目。

---

## HTTP 协议

### 智能 (Smart) HTTP 协议

“智能” HTTP 协议的运行方式和 SSH 及 Git 协议类似，只是运行在标准的 HTTP/S 端口上并且可以使用各种 HTTP 验证机制，这意味着使用起来会比 SSH 协议简单的多，比如可以使用 HTTP 协议的用户名 / 密码的基础授权，免去设置 SSH 公钥。

智能 HTTP 协议或许已经是最流行的使用 Git 的方式了，它即支持像 `git://` 协议一样设置匿名服务，也可以像 SSH 协议一样提供传输时的授权和加密。而且只用一个 URL 就可以都做到，省去了为不同的需求设置不同的 URL。如果你要推送到一个需要授权的服务器上（一般来讲都需要），服务器会提示你输入用户名和密码。从服务器获取数据时也一样。GitHub 就是使用此协议。

### 哑 (Dumb) HTTP 协议

如果服务器没有提供智能 HTTP 协议的服务，Git 客户端会尝试使用更简单的“哑” HTTP 协议。哑 HTTP 协议里 web 服务器仅把裸版本库当作普通文件来对待，提供文件服务。哑 HTTP 协议的优美之处在于设置起来简单。基本上，只需要把一个裸版本库放在 HTTP 根目录，设置一个叫做 `post-update` 的挂钩就可以了（见 Git 钩子）。此时，只要能访问 web 服务器上你的版本库，就可以克隆你的版本库。

### 智能 HTTP 协议优点

不同的访问方式只需要一个 URL 以及服务器只在需要授权时提示输入授权信息，这两个简便性让终端用户使用 Git 变得非常简单。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。

另一个好处是 HTTP/S 协议被广泛使用，一般的企业防火墙都会允许这些端口的数据通过。

### 智能 HTTP 协议缺点

在一些服务器上，架设 HTTP/S 协议的服务端会比 SSH 协议的棘手一些。

---

除了这一点，用其他协议提供 Git 服务与 “智能” HTTP 协议相比就几乎没有优势了。

## SSH 协议

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。

## Git 协议

接下来是 Git 协议。这是包含在 Git 里的一个特殊的守护进程；它监听在一个特定的端口 (9418)，类似于 SSH 服务，但是访问无需任何授权。要让版本库支持 Git 协议，需要先创建一个 `git-daemon-export-ok` 文件 —— 它是 Git 协议守护进程为这个版本库提供服务的必要条件 —— 但是除此之外没有任何安全措施。要么谁都可以克隆这个版本库，要么谁也不能。这意味着，通常不能通过 Git 协议推送。由于没有授权机制，一旦你开放推送操作，意味着网络上知道这个项目 URL 的人都可以向项目推送数据。不用说，极少会有人这么做。

# 第二节 配置 Git 服务器

## 一、使用本地协议的服务器

Git 没有中央仓库的概念，只是一般情况下服务端的 Git 仓库目录中没有工作目录（即裸仓库）。使用本地协议时，将访问 GitHub 时的 URL 换成本地文件系统路径即可。

在下图中，我的 E 盘根目录下，有一个从 GitHub 上 clone 下来的本地仓库，将此仓库作为远端在 F 盘创建副本。

```
qingtian@DESKTOP-GAURFI0 MINGW64 /f
$ git clone /e/git-test/
Cloning into 'git-test'...
done.

qingtian@DESKTOP-GAURFI0 MINGW64 /f
$ cd git-test/

qingtian@DESKTOP-GAURFI0 MINGW64 /f/git-test (master)
$ git branch
* master

qingtian@DESKTOP-GAURFI0 MINGW64 /f/git-test (master)
$ git remote -v
origin  E:/git-test/ (fetch)
origin  E:/git-test/ (push)
```

### #克隆仓库

```
$ git clone /opt/git/project.git
```

```
$ git clone file:///opt/git/project.git
```

如果在 URL 开头明确的指定 `file://`，那么 Git 的行为会略有不同。如果仅是指定路径，Git 会尝试使用硬链接 (hard link) 或直接复制所需要的文件。如果指定 `file://`，Git 会触发平时用于网路传输资料的进程，那通常是传输效率较低的方法。

### #要增加一个本地版本库到现有的 Git 项目

```
$ git remote add local_proj /opt/git/project.git
```

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。如果你的团队已经有共享文件系统，建立版本库会十分容易。只需要像设置其他共享目录一样，把一个裸版本库的副本放到大家都可以访问的路径，并设置好读/写的权限，就可以了。

这种方法的缺点是，通常共享文件系统比较难配置，并且比起基本的网络连接访问，这不方便从多个位置访问。

最终，这个协议并不保护仓库避免意外的损坏。每一个用户都有“远程”目录的完整 shell 权限，没有方法可以阻止他们修改或删除 Git 内部文件和损坏仓库。

## 二、使用 SSH 协议的服务器

如果你有一台所有开发者都可以用 SSH 连接的服务器，你只需要设置仓库目录的访问权限 (linux 文件系统的权限)。如果你的 linux 发行版没有默认安装 ssh, 使用 `sudo apt-get install openssh-server` 安装即可。

访问服务器仓库的 URL 路径格式为：

`ssh://linux 操作系统用户名@服务器 IP 或者域名:linux 文件系统路径`



例如: `ssh://qingtian@192.168.43.154:/git-home/git-test`

在下面的例子中, 首先创建 `/git-home/` 目录, 并修改目录所有者为 `qingtian` 用户。然后在 `/git-home/` 目录下创建 `git-test` 空仓库。然后在 Windows 上创建仓库副本。

```
$ git init --bare git-test
```

```
qingtian@ubuntu:/git-home$ sudo chown -R qingtian /git-home/
```

```
$ git clone ssh://qingtian@192.168.43.154:/git-home/git-test git-test-ubuntu
```

```
qingtian@DESKTOP-GAURFIO MINGW64 /f
$ git clone ssh://qingtian@192.168.43.154:/git-home/git-test git-test-ubuntu
Cloning into 'git-test-ubuntu'...
The authenticity of host '192.168.43.154 (192.168.43.154)' can't be established.
ECDSA key fingerprint is SHA256:bKLW27vD8zFxmZFMYkYN2yCzoTfs5lxQsL3pw+Inq7pc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.43.154' (ECDSA) to the list of known hosts.
qingtian@192.168.43.154's password:
warning: You appear to have cloned an empty repository.
qingtian@DESKTOP-GAURFIO MINGW64 /f
```

在这种方式中, 有三个问题: 一是如果要多个用户访问则需要创建多个用户, 并且要设置相应的读写权限; 二是使用的用户使用 `bash`, 不能限制用户操作范围; 三是每次都需要输入密码。

针对第一个问题, 我们创建一个专门作为访问 `git` 服务的用户 `git` (或者 `git-root`), 并修改文件权限使 `git` 用户拥有读写权限。

```
$ sudo useradd git
```

```
$ sudo chown -R git /git-home/
```

```
qingtian@DESKTOP-GAURFIO MINGW64 /f/git-test-ubuntu (bran1)
$ git remote -v
origin  ssh://qingtian@192.168.43.154:/git-home/git-test (fetch)
origin  ssh://qingtian@192.168.43.154:/git-home/git-test (push)
ubuntu-server  ssh://git@192.168.43.154:/git-home/git-test (fetch)
ubuntu-server  ssh://git@192.168.43.154:/git-home/git-test (push)
```

```
qingtian@DESKTOP-GAURFIO MINGW64 /f/git-test-ubuntu (bran1)
$ git push ubuntu-server bran1:bran00001
git@192.168.43.154's password:
Total 0 (delta 0), reused 0 (delta 0)
To ssh://192.168.43.154:/git-home/git-test
 * [new branch]      bran1 -> bran00001
qingtian@DESKTOP-GAURFIO MINGW64 /f/git-test-ubuntu (bran1)
```

针对问题二, 我们可以修改 `git` 用户使用的 `shell` 为 `git shell`, 把用户的操作限制在与 `git` 相关的范围内;

```
qingtian@ubuntu:~$ sudo chsh git
正在更改 git 的 shell
请输入新值, 或直接敲回车键以使用默认值
登录 Shell [/usr/bin/git-shell]:
```

针对问题三, 我们在 `/home/git/.ssh/authorized_keys` 文件中加入各个用户生成的 `ssh` 公钥。

```
#生产 ssh key, 三次回车
```

```
$ ssh-keygen.
```

---

```
# cat id_rsa.pub >> /home/git/.ssh/authorized_keys
```

不再需要密码:

```
qingtian@DESKTOP-GAURFI0 MINGW64 /f/git-test-ubuntu (bran1)
$ git push origin bran1
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 668 bytes | 334.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://192.168.43.154:/git-home/git-test
 * [new branch]      bran1 -> bran1
qingtian@DESKTOP-GAURFI0 MINGW64 /f/git-test-ubuntu (bran1)
```

其它相关命令:

#把现有仓库导出为裸仓

```
$ git clone --bare my_project my_project.git
```

```
qingtian@DESKTOP-GAURFI0 MINGW64 /f
$ git clone --bare git-test/ git-test.git
Cloning into bare repository 'git-test.git'...
done.
qingtian@DESKTOP-GAURFI0 MINGW64 /f
```

#新建服务端空仓库（没有工作目录）

```
$ git init --bare
```

#自动修设置仓库目录的组权限为可写。

```
git init --shared
```

## 三、Smart HTTP 服务器

### 原理

通过 Apache Web 服务器接受 HTTP 请求，并将请求转发到 Git 自带的一个名为 git-http-backend 的 CGI 脚本。

### 简单 Smart HTTP 服务器搭建

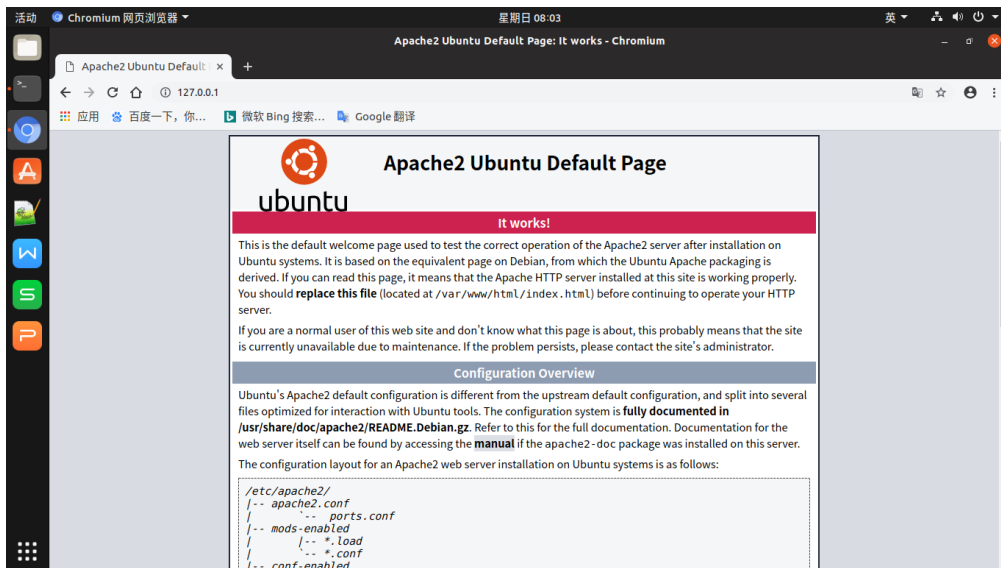
#### 1. 安装相关软件

```
sudo apt-get install apache2
```

```
sudo apt-get install apache2-utils
```

```
sudo a2enmod cgi alias env
```

在浏览器中输入 <http://127.0.0.1> 出现如下界面则安装成功。



## 2. 放置 git 裸仓库

### a. 创建一个空目录用来存放 git 仓库

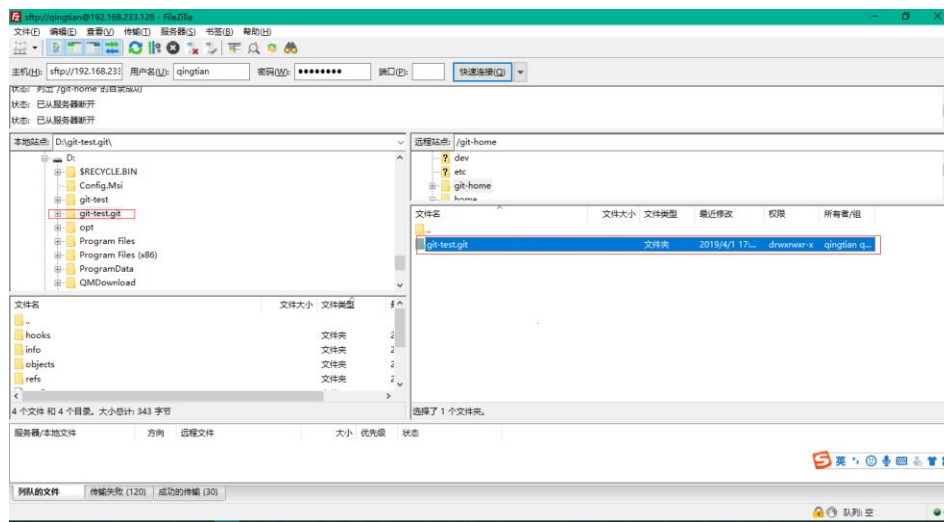
`sudo mkdir /git-home`

### b. 为了方便上传已有的裸仓库修改权限

`sudo chown qingtian:qingtian /git-home/`

### c. 创建裸仓库并上传服务器

`$ git clone --bare git-test/ git-test.git`



在我电脑上使用 `http://192.168.233.128/git-home/git-test.git` 路径会报错，修改为 `http://192.168.233.128/git-home/git-test`。

## 3. 修改 Apache 配置

在 Apache 配置文件 (`/etc/apache2/apache2.conf`) 中添加如下配置：

`SetEnv GIT_PROJECT_ROOT /git-home`

`SetEnv GIT_HTTP_EXPORT_ALL`

---

```
ScriptAlias /git-home /usr/lib/git-core/git-http-backend/
```

```
<Directory "/usr/lib/git-core*">
Options ExecCGI Indexes
Order allow,deny
Allow from all
Require all granted
</Directory>

<LocationMatch "^/git/.*/git-receive-pack$">
AuthType Basic
AuthName "Git Access"
AuthUserFile /git-home/.htpasswd
Require valid-user
</LocationMatch>
```

然后重启 Apache 服务。

```
$sudo service apache2 restart。
```

#### 4. 添加用户 2076940762 并设置密码

```
$ htdigest -c /git-home/.htpasswd "Git Access" 2076940762
```

#### 5. 使能 receivepack

到此，我们的服务器只能执行 pull，不能执行 push 操作。

By default, only the upload-pack service is enabled, which serves git fetch-pack and git ls-remote clients, which are invoked from git fetch, git pull, and git clone. If the client is authenticated, the receive-pack service is enabled, which serves git send-pack clients, which is invoked from git push.

在 git 仓库目录下

```
qingtian@ubuntu:/git-home/git-test$ git config http.receivepack
true
```

## 验证

```
git clone http://192.168.233.128/git-home/git-test
```

```
qingtian@DESKTOP-GAURFIO MINGW64 /e
$ git clone http://192.168.233.128/git-home/git-test
Cloning into 'git-test'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
```

```
qingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git remote -v
origin http://192.168.233.128/git-home/git-test (fetch)
origin http://192.168.233.128/git-home/git-test (push)
```

修改后提交:

```
qingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 277 bytes | 138.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To http://192.168.233.128/git-home/git-test
6fef18d..2e91eeb master -> master
```

pull

```
qingtian@DESKTOP-GAURFIO MINGW64 /e/git-test (master)
$ git pull origin master
From http://192.168.233.128/git-home/git-test
* branch          master      -> FETCH_HEAD
Already up to date.
```

---

## 参考资料

[Pro git 2](#)

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