

# 1. 什么是版本控制

版本控制是指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。在软件开发中，我们通常对代码源文件作版本控制，但实际上，你可以对任何类型的文件进行版本控制。

版本控制最主要的功能就是追踪文件的变更。它将什么时候、什么人更改了文件的什么内容等信息忠实地记录下来。每一次文件的改变，文件的版本号都将增加。除了记录版本变更外，版本控制的另一个重要功能是并行开发。软件开发往往是多人协同作业，版本控制可以有效地解决版本的同步以及不同开发者之间的开发通信问题，提高协同开发的效率。并行开发中最常见的不同版本软件的错误(Bug)修正问题也可以通过版本控制中分支与合并的方法有效地解决

如果你是位图形或网页设计师，可能会需要保存某一幅图片或页面布局文件的所有修订版本（这或许是你非常渴望拥有的功能），采用版本控制系统（VCS）是个明智的选择。有了它你就可以将选定的文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等。使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改的改删的删，你也照样可以轻松恢复到原先的样子。但额外增加的工作量却微乎其微。

## 2. VCS介绍

### 主流的版本控制系统

VCS: Version Control System, 版本控制系统

当前主流的两类代码版本管理系统: SVN和Git



SVN的全称是Subversion，即版本控制系统，开源软件。

Git是Linux之父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。

### SVN与Git的区别对比

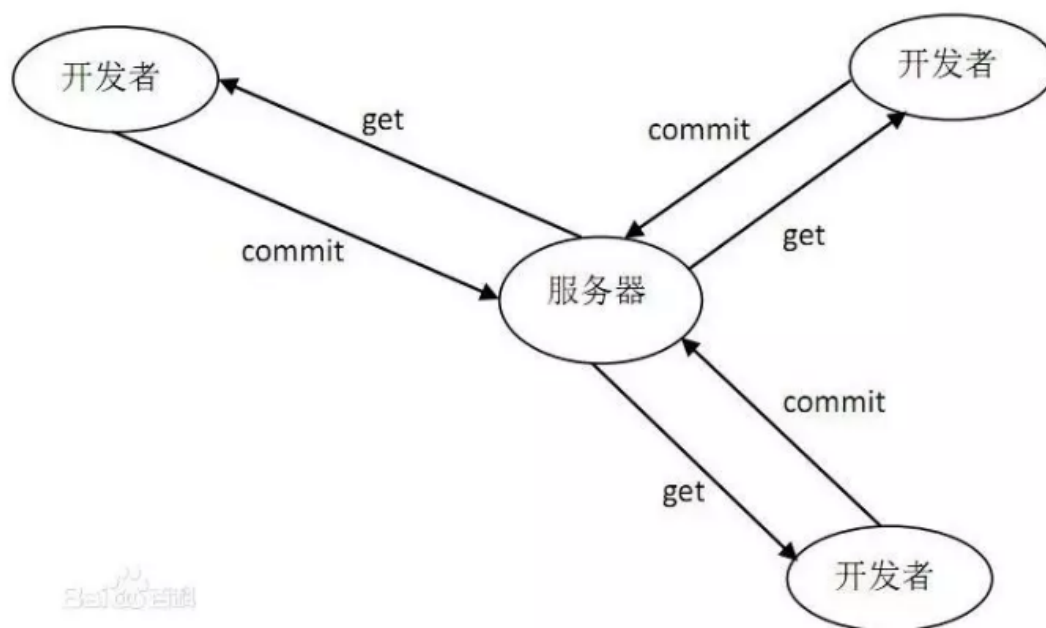
#### 1. SVN属于集中式的版本控制系统，Git属于分布式版本控制系统

SVN: SVN属于集中式的版本控制系统，而集中式的版本控制系统都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。

- 优点: 代码一致性非常高，每个人都可以一定程度上看到项目中的其他人正在做些什么，管理员也可以轻松掌控每个开发者的权限。
- 缺点: 必须联网，中央服务器不可靠，易出现单点故障。

Git: Git属于分布式版本控制系统, 分布式没有“中央服务器”, 每个人的电脑上都是一个完整的版本库, 意味着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。

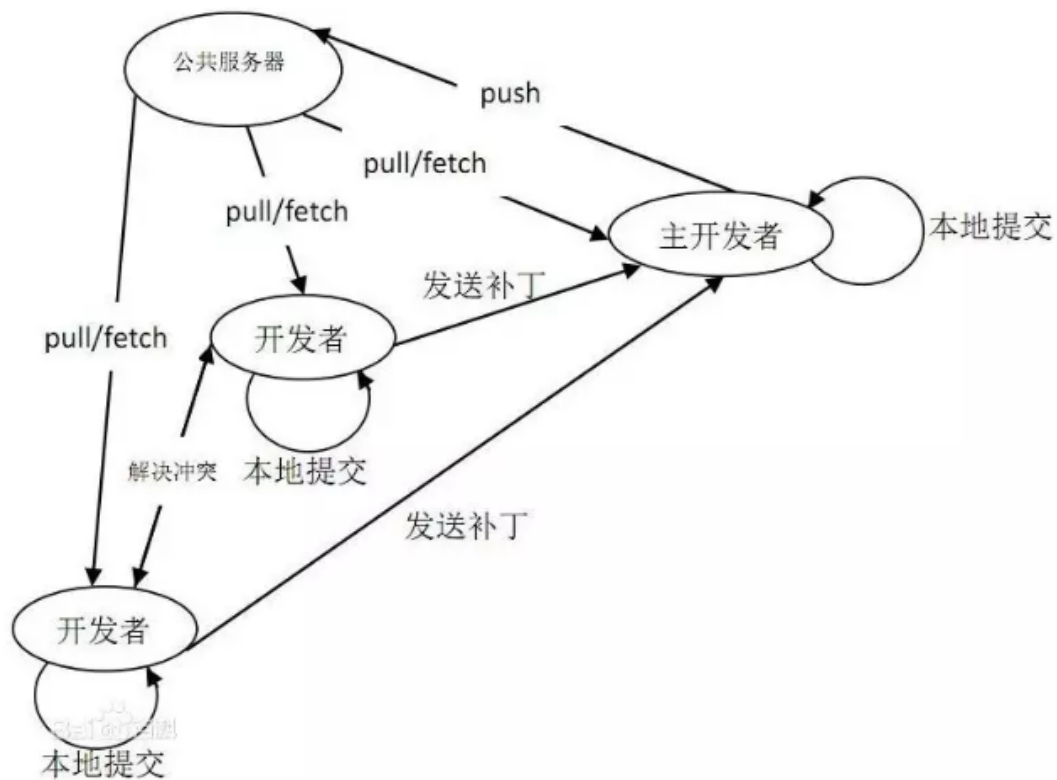
使用集中式版本控制系统的工作流程:



开发人员的一天:

- 1、开始一天的工作之前从服务器下载项目组最新代码。
- 2、进入自己的分支, 进行工作, 每隔一个小时向服务器自己的分支提交一次代码 (很多人都有这个习惯。因为有时候自己对代码改来改去, 最后又想还原到前一个小时的版本, 或者看看前一个小时自己修改了哪些代码, 就需要这样做了)。
- 3、下班时间快到了, 把自己的分支合并到服务器主分支上, 一天的工作完成, 并反映给服务器。

使用分布式版本控制系统的工作流程:



从一般开发者的角度来看，Git有以下功能：

- 1、从服务器上克隆完整的Git仓库（包括代码和版本信息）到单机上。
- 2、在自己的机器上根据不同的开发目的，创建分支，修改代码。
- 3、在单机上自己创建的分支上提交代码。
- 4、在单机上合并分支。
- 5、把服务器上最新版的代码fetch下来，然后跟自己的主分支合并。
- 6、生成补丁（patch），把补丁发送给主开发者。
- 7、看主开发者的反馈，如果主开发者发现两个一般开发者之间有冲突（他们之间可以合作解决的冲突），就会要求他们先解决冲突，然后再由其中一个人提交。如果主开发者可以自己解决，或者没有冲突，就通过。
- 8、一般开发者之间解决冲突的方法，开发者之间可以使用pull 命令解决冲突，解决完冲突之后再向主开发者提交补丁。

从主开发者的角度（假设主开发者不用开发代码）看，Git有以下功能：

- 1、查看邮件或者通过其它方式查看一般开发者的提交状态。
- 2、打上补丁，解决冲突（可以自己解决，也可以要求开发者之间解决以后再重新提交，如果是开源项目，还要决定哪些补丁有用，哪些不用）。
- 3、向公共服务器提交结果，然后通知所有开发人员。

## 2. SVN与Git版本号区别

Git分布式版本管理系统，采用 40 位长的哈希值作为版本号，不会出现重复，比如Git的版本号：  
cc09c29b705b99de5298d22a2c0e30262e83a7d2

了解：hash算法就是把任意长度的输入（又叫作预映射，pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。常用的hash算法有SHA-1、SHA-256、MD5等，Git使用的是SHA-1算法。

SVN 的版本号是连续的，可以预判下一个版本号。

比如SVN的版本号：

文件	扩展名	版本	作者	大小	日期	锁定	锁定备注
.idea		763	test_jenkins		2022/2/14 15:02:19		
base		767	test_jenkins		2022/2/15 10:28:16		
common		763	test_jenkins		2022/2/14 15:02:19		
dataconfig		763	test_jenkins		2022/2/14 15:02:19		
testcases		763	test_jenkins		2022/2/14 15:02:19		

### 3. SVN与Git代码检出区别

SVN中，每个子目录下都维护着自己的.svn目录，记录着该目录中文件的修改情况以及和服务器端仓库的对应关系。SVN可以checkout部分路径下的内容（部分检出），而不用 checkout整个版本库或分支。

Git 没有部分检出，并且Git的本地仓库信息完全维护在项目根目录的.git目录下，（不像 SVN一样，每个子目录下都有单独的.svn目录）。

### 4. SVN与Git分支实现区别

本质上SVN的分支和标签都是来自目录拷贝，通常约定是拷贝在branches/和tags/目录下。所谓分支、tag等概念都只是仓库中不同路径上的一个对象或索引而已。

Git中的分支实际上仅是一个包含所指对象校验和（40个字符长度SHA-1哈希值）的文件。Git的分支是完全隔离的，一个提交一般只能发生在一个分支中。

## SVN与Git优缺点分析、适用场景

SVN的优缺点：

- 优点及适用场景：
  - 管理方便，逻辑明确，符合一般人思维习惯。
  - 易于管理，集中式服务器更能保证安全性。
  - 代码一致性非常高。
  - 适合开发人数不多的项目开发。
- 缺点：
  - 服务器压力太大，数据库容量暴增。
  - 如果不能连接到服务器上，基本上不可以工作
  - 不适合开源项目或者大型团队协作开发的项目。

Git的优缺点：

- 优点及适用场景：
  - 适合分布式团队多人协作，敏捷开发。
  - 公共服务器压力和数据量都不会太大。
  - 速度快、灵活、可离线工作。
  - 任意两个开发者之间可以很容易的解决冲突。
- 缺点：
  - 学习周期相对而言比较长。
  - 代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

## 3. Git 的安装与配置

### 安装Git

官方网站: <https://git-scm.com/downloads>, 根据自己的系统选择。

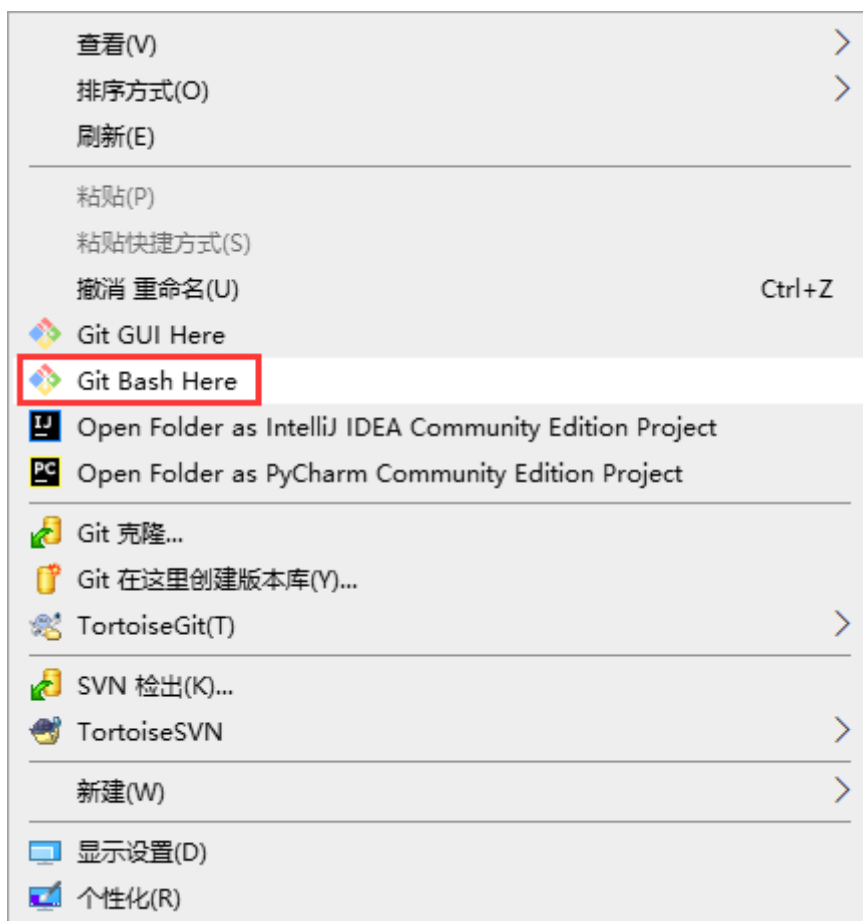


检查是否安装成功?

在Dos窗口中输入: `git --version`, 如果打印了Git的版本号, 则安装成功。

```
C:\Users\Charles.Shen>git --version
git version 2.35.1.windows.2
```

也可以通过点击鼠标右键看是否有Git Bash Here的选项来判断安装成功。



## 配置Git用户和邮箱

安装好Git后需要先配置用户名和邮箱，用户名和邮箱地址是本地git客户端的一个变量，每次commit都会用用户名和邮箱纪录。Github和Gitee上的贡献者统计就是按邮箱来统计的。具体配置命令如下：

- 设置用户姓名: `git config --global user.name "姓名"`，比如`git config --global user.name "Charles.Shen"`
- 设置邮箱地址: `git config --global user.email "联系邮箱"`，比如`git config --global user.email "charles_shen2009@163.com"`
- 查看设置信息: `git config --global --list`

## 4. Git的核心概念

### Git的4个区：工作区、暂存区、本地仓库、远程仓库











相比于SVN等传统的版本管理工具，Git多引入了一个暂存区(Stage)的概念：

- **工作区**：英文叫Workspace，就是你在电脑里能看到的目录（除了隐藏的.git目录）。
- **暂存区**：英文叫 stage 或 index。一般存放在 **.git** 目录下的 index 文件（.git/index）中，所以我们把暂存区有时也叫作索引（index）。
- **本地仓库区（版本库）**：英文叫Local Repository，也叫版本库，就是**.git**目录（隐藏的目录）
- **远程仓库**：英文叫Remote Repository，如果你想通过Git分享你的代码或者与其他开发人员合作。你就需要将数据放到一台其他开发人员能够连接的服务器上，这就是远程仓库。

2个区举例说明：

<input type="checkbox"/> 名称	修改日期	类型	大小
 .git <b>本地仓库区</b>	2022/3/20 16:57	文件夹	
 README.en.md <b>工作区</b>	2022/3/20 16:57	Markdown File	1 KB
 README.md	2022/3/20 16:57	Markdown File	1 KB

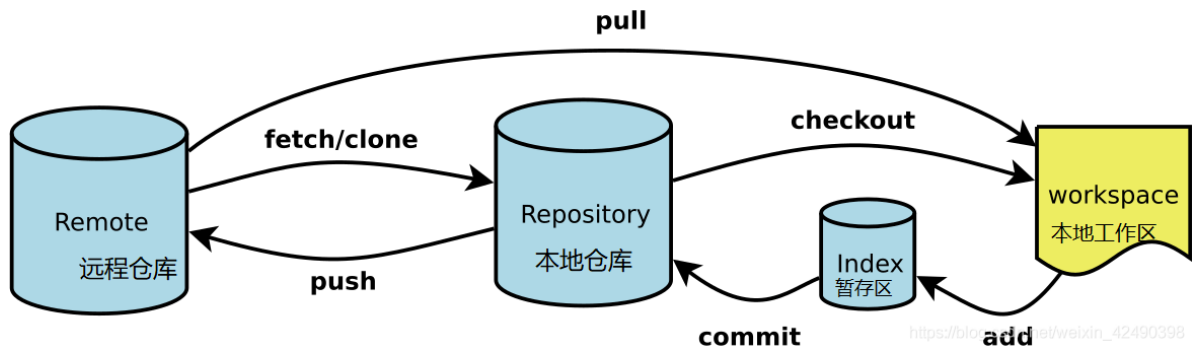
下面是暂存区：

<input type="checkbox"/> 名称	修改日期	类型	大小
 hooks	2022/3/20 16:57	文件夹	
 info	2022/3/20 16:57	文件夹	
 logs	2022/3/20 16:57	文件夹	
 objects	2022/3/20 16:57	文件夹	
 refs	2022/3/20 16:57	文件夹	
 config	2022/3/20 16:57	文件	1 KB
 description	2022/3/20 16:57	文件	1 KB
 HEAD	2022/3/20 16:57	文件	1 KB
 index <b>暂存区</b>	2022/3/20 16:57	文件	1 KB
 packed-refs	2022/3/20 16:57	文件	1 KB

4个区之间的关系说明：

- 把工作区的代码添加到暂存区
- 把暂存区的代码提交到本地仓库
- 把本地仓库的代码推到远程仓库

- 从远程仓库取代码到本地仓库，有fetch、clone、pull三种方式



## Git的3个步骤

正常情况下，我们的工作流就是3个步骤，对应上图中的3个箭头线：

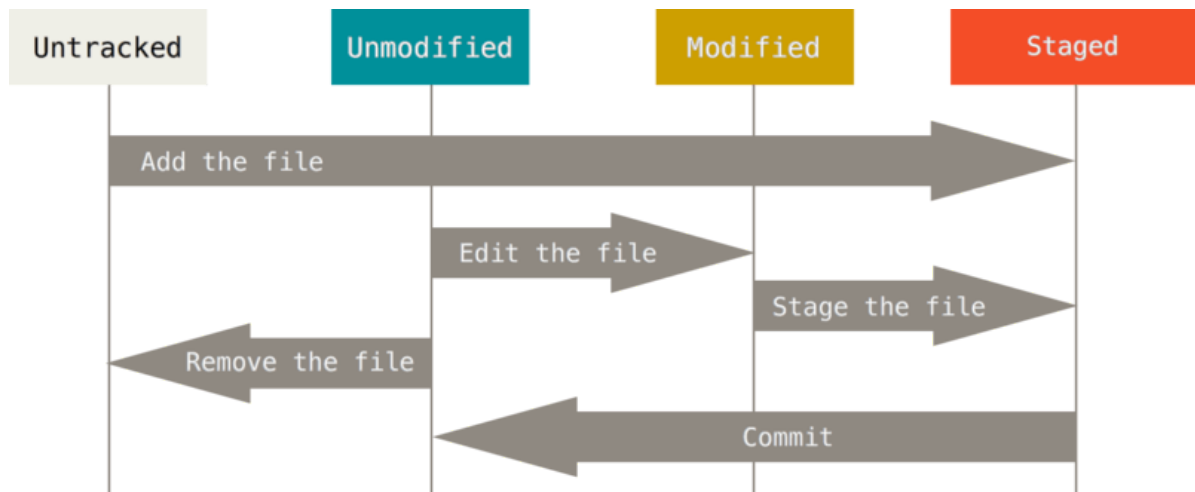


- `git add`：把所有文件放入暂存区
- `git commit`：把所有文件从暂存区提交进本地仓库
- `git push`：把所有文件从本地仓库推送进远程仓库

## Git的5种状态

Git对应的4个区，进入每一个区成功之后会产生一个状态，再加上最初始的一个状态，一共是5种状态：

- 未修改(Origin)
- 已修改(Modified)&未追踪(Untracked) 或者叫未暂存
- 已暂存(Staged)
- 已提交(Committed)
- 已推送(Pushed)





## 5. 创建Git仓库

通常有两种获取Git项目仓库的方式：

1. 将尚未进行版本控制的本地目录转换为 Git 仓库
2. 从其它服务器 **克隆** 一个已存在的 Git 仓库

两种方式都会在你的本地机器上得到一个工作就绪的 Git 仓库。

### git init

**git init** 命令用于在目录中创建新的Git仓库，在目录中执行 **git init** 就可以创建一个Git仓库了。执行这个命令后会在目录下生成一个.git的隐藏目录，这是普通目录和Git仓库的区别，如果删除.git目录就又变成了一个普通目录。

**#举例：创建一个demo\_repo的目录，然后进入到demo\_repo目录，执行git init命令创建Git仓库**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo
```

```
$ mkdir demo_repo
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo
```

```
$ cd demo_repo/
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo
```

```
$ git init
```

```
Initialized empty Git repository in D:/GitDemo/demo_repo/.git/
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
```

```
$ ll
```

```
total 0
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
```

```
$ ls -al
```

```
total 4
```

```
drwxr-xr-x 1 Charles.Shen 197121 0 Mar 20 19:58 ./
```

```
drwxr-xr-x 1 Charles.Shen 197121 0 Mar 20 19:58 ../
```

```
drwxr-xr-x 1 Charles.Shen 197121 0 Mar 20 19:58 .git/
```

### git clone

**git clone** 命令用于拷贝（克隆）一个远程仓库到本地，让自己能够查看该项目，或者进行修改。

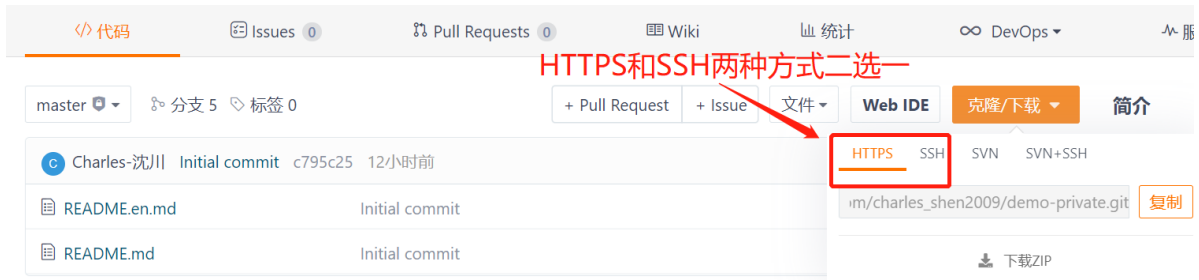
克隆仓库的几种命令：

- 1、克隆仓库到本地：**git clone 远程仓库URL**，当远程仓库有多个分支时，git clone命令克隆的是默认分支。例如：git clone [https://gitee.com/charles\\_shen2009/demo-private.git](https://gitee.com/charles_shen2009/demo-private.git) 会把远程仓库克隆到本地的当前工作目录下，并且本地仓库的名称默认与远程仓库相同



```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo
$ git clone https://gitee.com/charles_shen2009/demo-private.git
Cloning into 'demo-private'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

在Gitee上查看仓库URL地址：



2、克隆仓库到本地，并修改名称：**git clone 远程仓库URL 本地仓库名称**，例如：`git clone git@gitee.com:charlessshenchuan/rong-hua-test32.git rong-hua-test-demo` 会把远程仓库克隆到本地的当前工作目录下，并且命名成rong-hua-test-demo

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo
$ git clone git@gitee.com:charlessshenchuan/rong-hua-test32.git rong-hua-test-demo
Cloning into 'rong-hua-test-demo'...
remote: Enumerating objects: 32, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 32 (delta 8), reused 0 (delta 0), pack-reused 10
Receiving objects: 100% (32/32), 7.31 KiB | 534.00 KiB/s, done.
Resolving deltas: 100% (11/11), done.
```

3、clone克隆指定分支：**git clone 远程仓库URL -b 分支名称 本地仓库名称**，分支的概念后面再详细讲解，例如：`git clone git@gitee.com:charlessshenchuan/rong-hua-test32.git -b test rong-hua-test-demo-test`

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo
$ git clone git@gitee.com:charlessshenchuan/rong-hua-test32.git -b test rong-hua-test-demo-test
Cloning into 'rong-hua-test-demo-test'...
remote: Enumerating objects: 32, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 32 (delta 8), reused 0 (delta 0), pack-reused 10
Receiving objects: 100% (32/32), 7.31 KiB | 1.22 MiB/s, done.
Resolving deltas: 100% (11/11), done.
```

## 两种克隆方式：HTTPS和SSH

在Git中clone项目有两种方式：HTTPS和SSH，它们的区别如下：

- HTTPS：不管是谁，拿到URL就可以随便clone，但是在Push的时候需要验证用户名和密码。
- SSH：clone的项目你必须是拥有者或者管理员，并且需要在clone前添加SSH公钥。SSH在Push的时候，不需要输入用户名，如果配置SSH公钥的时候设置了密码，则需要输入密码的，否则直接是不需要输入密码的。

## SSH方式的配置

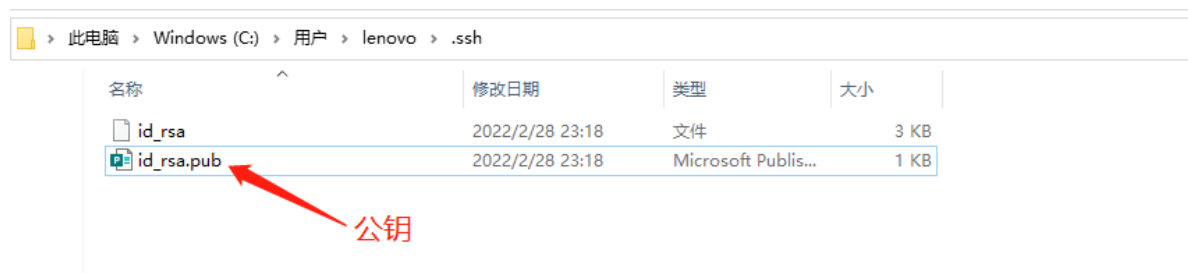
如果要使用SSH的方式clone项目，需要在Windows上生成SSH密钥对，然后把公钥提供给Git服务器进行认证授权，一个密钥对分为私钥和公钥，公钥放到服务器端提供认证。

生成密钥对的命令：**ssh-keygen -t rsa -C 描述信息**，其中-C表示添加说明，-t代表类型。

建议一路按Enter回车，不修改密钥对文件的路径和名称，这样会默认保存密钥对文件在/c/Users/用户名/.ssh/目录下，其中id\_rsa 文件是私钥，一定不能泄露；id\_rsa.pub 文件是公钥，内容放在服务端提供验证。

```
$ ssh-keygen -t rsa -C "测试密钥"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/lenovo/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/lenovo/.ssh/id_rsa
Your public key has been saved in /c/Users/lenovo/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:cXPCLWmMiTRXrDz/GD8E1Ue2fri9bwd+q6rh4hKgyfk 测试密钥
The key's randomart image is:
+---[RSA 3072]-----+
|      ..  o.  . .o |
|      . .  .+.o. .o |
|      . .oo.X.o  o |
|      . . .++.= . . |
| . + .   So .   . . |
| = .      o . .o. |
| . .      * . . . . |
|   E . .  o + . = |
|      o..o...o.== |
+-----[SHA256]-----+
```

私钥与公钥：



复制公钥的内容，配置到Gitee的 **安全设置——SSH公钥** 里面

添加公钥

标题

测试用途

公钥

把你的公钥粘贴到这里，查看 [怎样生成公钥](#)

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQGCs++eZDnl5jeHEBAiDw+vUZdo+LIUcfOlB RAMhooeQPG90axYB8yb8+hRx1x2CfDbvcPm/0
gDk9melh2ZMjQ0KdYGLQK3kcttPDY7ATEyID+I3D+Dhs6D5dbDY6N4S3TdLQk/qisv5eycsHKjKr20XL+CBX7FolFgY5jX/I2fSiognouevNoovP+N
VZF6BW4cnq+sYjdk8H2iK6F/g/9/Ye8G8KYvf9n8ezR3yyhVNVPF7vLF/lyiQ0TEqNQLR/di1slel2SfFO+R2u1OeCLxPMZuzQvXgaUeLccUqPf4G0F
NDRQEZralZirMQe55+eFVoAj/vy1VrmuwgPBj6MUI5pn7er9IYqnKijeH9dhyeZXpt73KEs0Kvr8MGXbQ1+CyxS5dTDX+voDuuK0p4jrpSX4p04Yt
7sRcrlVGfpQzrb44HRv+MBIHx5NvYpMM7Paw/r82uFt.../xY0gxhZllxbHn+Jls7yKUAYpkr9Rphil1cJvA1suzM= 测试密钥
```

确定

## 6. 提交与修改

### git status

**git status** 命令用于查看在你上次提交之后是否有对文件进行再次修改或删除，通常我们使用 **-s** 参数来获得简短的输出结果。

如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

这说明你当前的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。最后，该命令还显示了当前所在分支是 master，并告诉你这个分支同远程服务器上对应的分支没有偏离。

如下例子表示有更新、删除和新增的文件：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   test1.py
    deleted:    test2.py
    new file:   test4.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status -s
M test1.py
D test2.py
A test4.py
```

git status的3种状态：

- Untracked files: 这种情况出现在新建几个文件（版本中不存在的），但是没有提交到暂存区的时候
- Changes not staged for commit: 这种情况出现在版本已有文件遭到修改但是还没提交到暂存区的时候
- Changes to be committed: 这种情况出现在提交到暂存区之后的时候

## git add

**git add** 命令可将文件添加到暂存区，也叫追踪文件。添加后可以执行命令git status来查看文件的状态，只要在 Changes to be committed 这行下面的，就说明是已暂存状态。

添加当前目录的所有文件到暂存区：**git add .**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git add .
warning: LF will be replaced by CRLF in test33.py.
The file will have its original line endings in your working directory
```

添加一个或多个文件到暂存区：**git add 文件1 文件2 ... 文件n**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git add test1.py test2.py test3.py
warning: LF will be replaced by CRLF in test1.py.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test2.py.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in test3.py.
The file will have its original line endings in your working directory
```

添加整个目录到暂存区：**git add 目录**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git add dir01/
```

## git ls-files

**git ls-files** 命令用来查看暂存区的文件，常用参数：

- **--cached 或者 -c**: 查看暂存区中文件，如果不指定参数，默认就是使用的这个参数

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git ls-files -c
demo.py
hello_world2.py
test1.py
test4.py
test5.py
test6.py
test7.py
```

- **--modified 或者 -m**: 查看修改的文件, 在add后被修改的文件 (修改后还没有被再次暂存)

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ vi test6.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git ls-files -m
demo.py
test6.py
```

- **--delete 或者 -d**: 查看删除过的文件, 在add后被删除的文件

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ rm -f test7.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git ls-files -d
test7.py
```

- **--other 或者 -o**: 查看没有被Git追踪的文件, 也就是在工作区中但还没有被add到暂存区的文件

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ touch test7.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git ls-files -o
test7.py
```

- **--stage 或者 -s**: 显示暂存的条目的相关信息 (模式位mode, 文件哈希后的值, 暂存号和文件名)

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git ls-files -s
100644 1f9580a3cff61cb8f0dbed353a4ba0150d0162a8 0      demo.py
100644 bcb7afa68a48159accff1e006b2c9a29bb3361dc 0      hello_world2.py
100644 0eee9f0d14fd9dea0a811dfda761ae76676847cc 0      test1.py
100644 e48b640c6cd50eebf46b412dfbf409eb85673a36 0      test4.py
100644 24646e1b1c2ac7a50a1e300d62fe2f83821f6c51 0      test5.py
100644 b917a726c93f902e43291d9009d6488385133b67 0      test6.py
100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0      test7.py
```

其中100644是mode: 100代表常规文件, 644代表文件权限

**注意: 查看工作区的文件用 ls 命令, 查看暂存区的文件用 git ls-files 命令。**

## git commit

**git commit** 命令将暂存区内容添加到本地仓库中。

提交暂存区代码到本地仓库: **git commit -m 备注信息**, 例如: `git commit -m "2022年2月28日提交"`

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git commit -m "2022年第一次提交"
[test 3e77a0e] 2022年第一次提交
2 files changed, 2 insertions(+)
rename test34.py => test38.py (100%)
```

**注意: 在提交代码的时候, 一定要通过参数-m添加描述, 如果没有加参数-m, 会弹出一个编辑窗口添加描述后才能提交。**

直接提交工作区自上次 commit 之后的变化 (修改或删除) 到本地仓库: **git commit -a -m 备注信息**, 针对修改或删除的文件, 可以通过这个方式直接提交, 省略git add这一步, 但针对新文件还是要先git add, 不然就是 untracked 状态。

举例: 修改test33.py文件后, 直接执行**git commit -a -m**命令就可以提交修改后的代码

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git commit -a -m "2022年第二次提交"
warning: LF will be replaced by CRLF in test33.py.
The file will have its original line endings in your working directory
[test b23b3d5] 2022年第二次提交
1 file changed, 1 insertion(+)
```

### 追加提交 (修改上一次的提交信息)

命令: **git commit --amend -m 备注信息**, 比如 `git commit --amend -m "2022年2月28日提交更新"`, 当你提交后发现描述信息有误, 可以使用这个命令来更新描述信息, 追加提交后上一次提交记录被删除了, 只保留新的提交记录。

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (test)
$ git commit --amend -m "2022年第二次提交（追加描述）"
[test 13192e5] 2022年第二次提交（追加描述）
Date: Tue Mar 1 23:45:14 2022 +0800
1 file changed, 1 insertion(+)
```

## git rm

如果要删除工作区的文件，可以用 **rm** 命令，如果要删除暂存区和版本库的文件，则需要使用 **git rm** 命令。把一个文件从工作区删除，并不会影响暂存区和版本库。

将文件从暂存区和工作区中删除：**git rm 文件名**，删除文件的几种场景：

1、如果文件提交到了版本库，且工作区、暂存区、版本库中的文件一致，则可以成功删除：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git rm test2.py
rm 'test2.py'
```

2、如果文件没有在暂存区，执行 **git rm** 命令会报如下错误：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git rm test4.py
fatal: pathspec 'test4.py' did not match any files
```

3、如果文件在暂存区但没有提交到版本库，执行 **git rm** 命令会报如下错误：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git rm test1.py
error: the following file has changes staged in the index:
    test1.py
(use --cached to keep the file, or -f to force removal)
```

4、如果工作区与**暂存区**、**版本库**中文件不一致，就删除不了，除非强制删除，如下：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git rm test1.py
error: the following file has local modifications:
    test1.py
(use --cached to keep the file, or -f to force removal)
```

5、如果**工作区**、**暂存区**与版本库中文件不一致，就删除不了，除非强制删除，如下：



```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git rm test1.py
error: the following file has changes staged in the index:
    test1.py
(use --cached to keep the file, or -f to force removal)
```

6、从暂存区删除，但文件仍保留在工作区：**git rm --cached 文件名**，如果是删除目录需要加上参数 **-r**

```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (test)
$ git rm --cached -r dir1/
rm 'dir1/file1'
```

## git mv

**git mv** 命令用于移动或重命名一个文件、目录或软链接。重命名后，新的文件放在暂存区中，需要重新提交到版本库。

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git mv hello_world.py hello_world2.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    hello_world.py -> hello_world2.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git commit -m "the 3rd commit"
[master 8c331a7] the 3rd commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename hello_world.py => hello_world2.py (100%)

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

## git diff

**git diff** 命令比较文件的不同，即比较文件在暂存区和工作区的差异。这个命令显示的已写入暂存区和已经被修改但尚未写入暂存区文件的区别。暂存区的文件叫源文件，工作区的文件叫目标文件。

**git diff** 有两个主要的应用场景。

- 显示工作区中已经修改但未添加至暂存区的文件和已经添加至暂存区文件的修改：**git diff**
- 查看最后一次提交到版本库环境中文件和暂存区中文件的修改对比：**git diff --cached**
- 查看已经提交到版本库环境中的文件和未提交到版本库环境中文件（包括工作区和暂存区的文件）的所有修改对比：**git diff HEAD**

- 只显示摘要而非整个差异: **git diff --stat**

举例:

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git diff
diff --git a/demo.py b/demo.py
index 4a2964c..1f9580a 100644
--- a/demo.py
+++ b/demo.py
@@ -9,7 +9,7 @@
# 命名规范: 文件名全小写+下划线, 类名大驼峰, 方法和变量小写+下划线连接,
# 常量大写, 变量和常量用名词, 方法用动词
# -----
-a = ['a', 'b', 'c', 'a', 'c']
+a = ['a', 'b', 'c', 'a', 'c', 'f']
for item in enumerate(a):
    print(item)
    if item[1] == 'a':
@@ -19,10 +19,10 @@ for item in enumerate(a):
    print(string)
# str1="hello"          #定义一个变量str1, 赋值成字符串hello; 声明了一个字符串类的对象str1
# print(type(str1))
-# class DemoClass:
-#     pass
+class DemoClass:
+    pass
#
-# dc=DemoClass()
+dc=DemoClass()
# print(type(dc))
#
# ''
@@ -51,3 +51,5 @@ for item in enumerate(a):
# str1="hello world"
# print(str1[1])
# print(str1[-10])
+for i in range(11):
+    print(i)
```

**git diff** 结果分析:

1. 第一行diff --git a/demo.py b/demo.py表示结果为git格式的diff, a版本的demo.py表示变动前的版本(暂存区中的文件, 源文件), b版本的demo.py表示变动后的版本(工作区中的文件, 目标文件);
2. 第二行index 4a2964c..1f9580a 100644表示两个版本的git哈希值(index区域的1f9580a对象, 与工作目录区域的100644对象进行比较), 最后的六位数字是对象的模式(普通文件,644权限);
3. 第三行和第四行表示进行比较的两个文件. --- a/demo.py表示变动前的版本, +++ b/demo.py表示变动后的版本;
4. @@ -9,7 +9,7 @@表示在源文件第9行开始的7行和目标文件第9行开始的7行构成一个差异小结, 后面是详细的差异。

## git restore、git reset

git restore 命令的作用是撤销文件的修改，通常有以下几种场景：

- 1、工作区的文件修改了，但还没添加到暂存区：**git restore 文件**
- 2、工作区的文件修改了，也添加到了暂存区，但还没提交到版本库：先执行**git restore --staged 文件**再执行**git restore 文件**来撤销文件的修改
- 3、已经提交到了版本库：
  - git reset --soft HEAD~1，其中HEAD~1表示退回到上一个版本，如果要退回到上2个版本则用HEAD~2；--soft表示不删除工作区的改动代码，只是把本地仓库提交撤销了，撤销commit，不撤销add
  - git reset --hard HEAD~1，其中--hard表示删除工作空间的改动代码，同时撤销commit和撤销add

举例：

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ vi test6.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test6.py

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

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git add .
warning: LF will be replaced by CRLF in test6.py.
The file will have its original line endings in your working directory

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   test6.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git restore --stage test6.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test6.py

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

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ cat test6.py
print(1)
#New comment
```

#这个代码有毒

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git restore test6.py
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ cat test6.py
print(1)
#New comment
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git reset --soft HEAD~1
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   test6.py
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ cat test6.py
print(1)
#New comment
#这个代码有毒
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git commit -m "20220322第6次提交"
[master e2e2ece] 20220322第6次提交
1 file changed, 1 insertion(+)
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git reset --hard HEAD~1
HEAD is now at 09ccf2c 20220322第3次提交
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ cat test6.py
print(1)
#New comment
```

# git log

查看版本库修改记录: git log, 执行这个 命令可以查看版本库的提交记录

MINGW64:/c:/Users/lenovo/Desktop/GitDemo38\_2/rhctest33

```
commit 13192e535b65d66d1213fd271d6970e94390db64 (HEAD -> test)
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 23:45:14 2022 +0800
```

2022年第二次提交 (追加描述)

```
commit 3e77a0ecaea845d94f19c76b02d52a4f44c3dbb0
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 23:40:19 2022 +0800
```

2022年第一次提交

```
commit 8742d1aea98bc27ec8b18f0e13fd40f63e332751 (origin/test)
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 22:32:26 2022 +0800
```

xxx

```
commit 0ceb114f0a16d0567f31913762525044c2b6114e
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 22:11:41 2022 +0800
```

第一次提交

```
commit e68e5e3ed512d436afb2c3b9746b220972430aa8
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 07:19:21 2022 +0000
```

在Gitee上更新2

```
commit 27d8c6848cd75f248ff11c89f426cd8cacb4507e
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 05:56:35 2022 +0000
```

update demo\_test.

```
commit 1139af6701361ce144d5bd231123150637243be5
Author: Charles.Shen <charles_shen2009@163.com>
Date: Tue Mar 1 13:52:25 2022 +0800
```

test分支更新demo\_test文件后提交

```
commit cc09c29b705b99de5298d22a2c0e30262e83a7d2 (origin/rhctest)
Author: Charles.Shen <charles_shen2009@163.com>
Date: Sun Sep 26 16:23:32 2021 +0800
```

1.0版本

```
commit a5fa1a7ab463621b12d651ad833864f3ccb773e8
Author: Charles.Shen <charles_shen2009@163.com>
Date: Sun Sep 26 16:14:15 2021 +0800
```

第一次提交代码

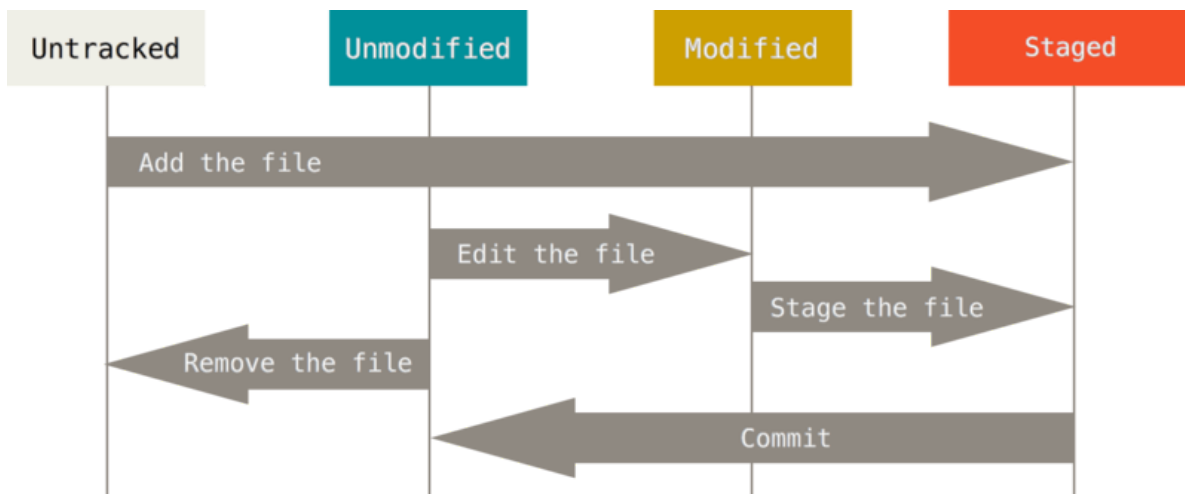
```
commit ccad95d74a444cdef95857d897f6c1ff49c91a1a
:|
```

- 查看版本库修改记录: git log
- 单行显示最近5条提交历史记录: git log --oneline -n5
- 图形化显示当前分支的提交日志: git log --graph --oneline
- 图形化显示所有分支的提交日志: git log --graph --oneline --all
- 查看某人提交记录: git log --author=Charles.Shen
- 查看所有最近两周内的提交: git log --since=2.weeks

# Git状态流转演示

Git对应的4个区，进入每一个区成功之后会产生一个状态，再加上最初始的一个状态，一共是5种状态：

- 未修改(Origin)
- 已修改(Modified)&未追踪(Untracked)
- 已暂存(Staged)
- 已提交(Committed)
- 已推送(Pushed)



## 1、检查当前文件状态

可以用 `git status` 命令查看哪些文件处于什么状态。如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

```
$ git status
On branch test
Your branch is up to date with 'origin/test'.

nothing to commit, working tree clean
```

这说明你当前的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上对应的分支没有偏离。现在，分支名是“test”，这是默认的分支名。

## 2、在工作目录下新建一个文件，这时这个文件的状态是未追踪

```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (test)
$ git status
On branch test
Your branch is up to date with 'origin/test'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  test38.py

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

3、跟踪新文件，使用命令 `git add` 跟踪上一步新建的文件，这时文件的状态变成已暂存，只要在 `Changes to be committed` 这行下面的，就说明是已暂存状态。

```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhctest33 (test)
$ git add test38.py

lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhctest33 (test)
$ git status
On branch test
Your branch is up to date with 'origin/test'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   test38.py
```

4、修改一个已被跟踪的文件，再执行git status命令，出现在 Changes not staged for commit 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行 git add 命令。

```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhctest33 (test)
$ vi test33.py

lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhctest33 (test)
$ git status
On branch test
Your branch is up to date with 'origin/test'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   test38.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test33.py
```

5、执行git add追踪已修改的文件，这时文件的状态会变成已暂存

```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhctest33 (test)
$ git add test33.py
warning: LF will be replaced by CRLF in test33.py.
The file will have its original line endings in your working directory

lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhctest33 (test)
$ git status
On branch test
Your branch is up to date with 'origin/test'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   test33.py
    new file:   test38.py
```

6、如果这时修改其中一个已暂存文件test33.py的内容，修改后发现这个文件同时出现在暂存区和非暂存区。实际上 Git 只不过暂存了你运行 git add命令时的版本。如果你现在提交，test33.py 的版本是你最后一次运行 git add 命令时的那个版本，而不是你运行 git commit 时，在工作目录中的当前版本。所以，运行了git add 之后又作了修订的文件，需要重新运行 git add 把最新版本重新暂存起来。



```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (test)
$ git status
On branch test
Your branch is up to date with 'origin/test'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   test33.py
    new file:   test38.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test33.py
```

7、再次执行git add命令追踪修改后的文件，执行后两个最新版本的文件被放到了暂存区

```
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (test)
$ git add test33.py
warning: LF will be replaced by CRLF in test33.py.
The file will have its original line endings in your working directory

lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (test)
$ git status
On branch test
Your branch is up to date with 'origin/test'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   test33.py
    new file:   test38.py
```

## Git查询类命令

- 查看引用日志: **git reflog**, 通过这个命令可以看到当前分支所有操作历史记录

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git reflog
09ccf2c (HEAD -> master) HEAD@{0}: reset: moving to HEAD~1
e2e2ece HEAD@{1}: commit: 20220322第6次提交
09ccf2c (HEAD -> master) HEAD@{2}: reset: moving to HEAD~1
4a20c2b HEAD@{3}: commit: 20220322第5次提交
09ccf2c (HEAD -> master) HEAD@{4}: commit: 20220322第3次提交
0a88c6c HEAD@{5}: commit: 20220322第二次提交
d664e8d HEAD@{6}: commit: 20220322提交
e2500d7 HEAD@{7}: commit: the 4th commit
8c331a7 HEAD@{8}: commit: the 3rd commit
24fb681 HEAD@{9}: commit: the 2nd commit
056ee75 HEAD@{10}: commit (initial): the 1st commit
```

- 显示指定文件是什么人在什么时间修改过: **git blame [file]**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git blame test1.py
^056ee75 (Charles.Shen 2022-03-20 21:33:06 +0800 1) print("hello world")
24fb681c (Charles.Shen 2022-03-20 21:52:56 +0800 2) print(123)
```

- 查看某个文件的历史具体修改内容: `git log -p [file]`

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
```

```
$ git log -p test1.py
```

```
commit 24fb681c7f6e299a23b27fa119a4f7e762a30e96
```

```
Author: Charles.Shen <charles_shen2009@163.com>
```

```
Date: Sun Mar 20 21:52:56 2022 +0800
```

```
the 2nd commit
```

```
diff --git a/test1.py b/test1.py
```

```
index 8cde782..0eee9f0 100644
```

```
--- a/test1.py
```

```
+++ b/test1.py
```

```
@@ -1,2 @@
```

```
    print("hello world")
```

```
+print(123)
```

```
commit 056ee7588d952a5a6cf3612b405d547fbec9ac19
```

```
Author: Charles.Shen <charles_shen2009@163.com>
```

```
Date: Sun Mar 20 21:33:06 2022 +0800
```

```
the 1st commit
```

```
diff --git a/test1.py b/test1.py
```

```
new file mode 100644
```

```
index 0000000..8cde782
```

```
--- /dev/null
```

```
+++ b/test1.py
```

```
@@ -0,0 +1 @@
```

```
+print("hello world")
```

```
(END)
```

```
the 2nd commit
```

```
diff --git a/test1.py b/test1.py
```

```
index 8cde782..0eee9f0 100644
```

```
--- a/test1.py
```

```
+++ b/test1.py
```

```
@@ -1,2 @@
```

```
    print("hello world")
```

```
+print(123)
```

```
commit 056ee7588d952a5a6cf3612b405d547fbec9ac19
```

```
Author: Charles.Shen <charles_shen2009@163.com>
```

```
Date: Sun Mar 20 21:33:06 2022 +0800
```

```
the 1st commit
```

```
diff --git a/test1.py b/test1.py
```

```
new file mode 100644
```

```
index 0000000..8cde782
```

```
--- /dev/null
```

```
+++ b/test1.py
```

```
@@ -0,0 +1 @@
```

```
+print("hello world")
```

- 查看某次提交具体修改内容: `git show [commit]`

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git show 24fb681c7f6e299a23b27fa119a4f7e762a30e96
commit 24fb681c7f6e299a23b27fa119a4f7e762a30e96
Author: Charles.Shen <charles_shen2009@163.com>
Date: Sun Mar 20 21:52:56 2022 +0800
```

the 2nd commit

```
diff --git a/test1.py b/test1.py
index 8cde782..0eee9f0 100644
--- a/test1.py
+++ b/test1.py
@@ -1,2 @@
 print("hello world")
+print(123)
diff --git a/test2.py b/test2.py
deleted file mode 100644
index 8cde782..0000000
--- a/test2.py
+++ /dev/null
@@ -1,0,0 @@
-print("hello world")
diff --git a/test4.py b/test4.py
new file mode 100644
index 0000000..e69de29
```

- 图形化显示当前分支的提交日志及每次提交的变更内容: `git log --graph --patch`
- 图形化显示所有分支的提交日志及每次提交的变更内容: `git log --graph --patch --all`

## 7. 分支操作

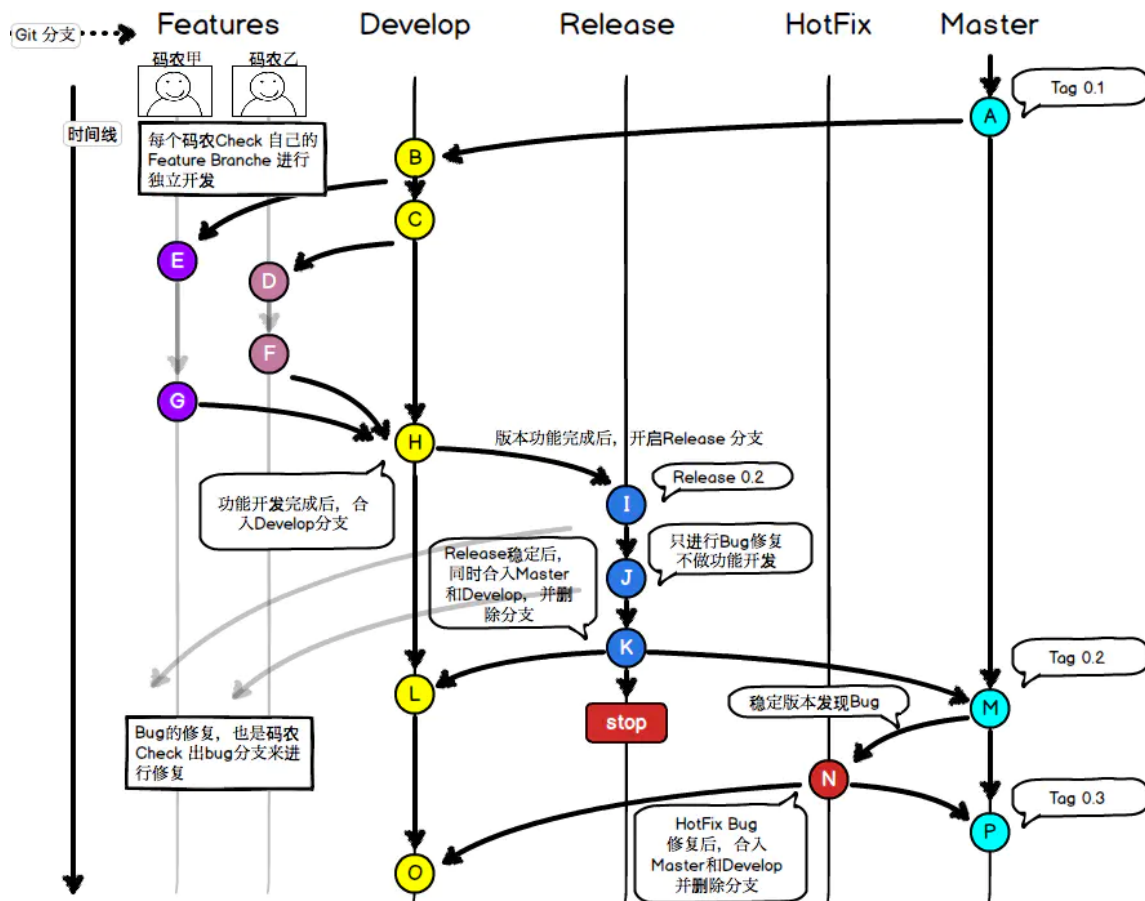
### 项目开发中为什么需要分支

项目开发中的烦恼:

- 维护线上项目, 同时开发新功能模块
- 尝试开发一个不确定上线的功能模块
- 渠道、定制项目, 代码上大同小异
- .....

针对这些烦恼, 怎么解决? 答案: 起分支。

在软件项目开发中, 我们通常是多人协同开发, 不同的功能分支就成了家常便饭的事情了。咱先来看一副图:



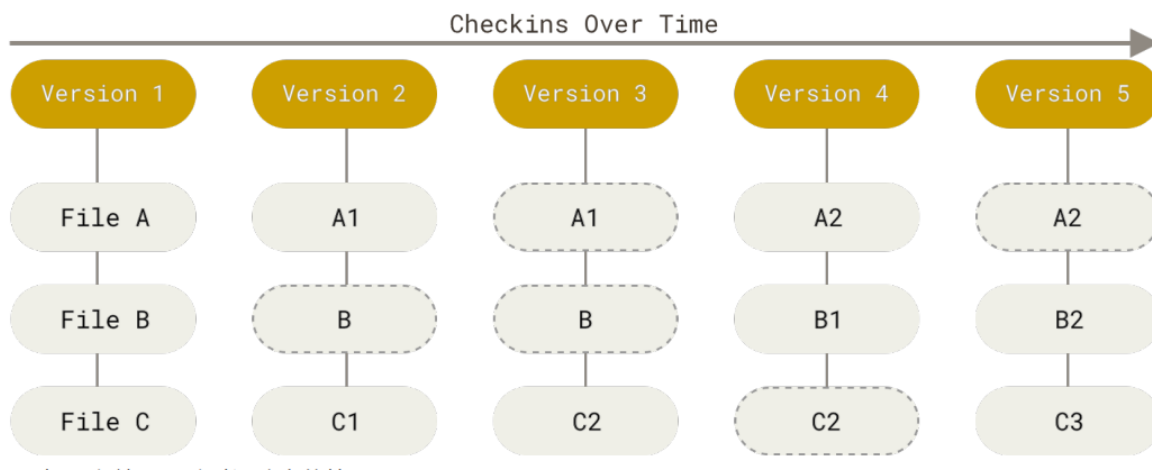
代码管理中的常见分支：

- Master: 主分支；主要是稳定的版本分支，正式发布的版本都从Master拉。
- Develop: 开发分支；更新和变动最频繁的分支，正常情况下开发都是在Develop分支上进行的。
- Release: 预发行分支；一般来说，代表一个版本的功能全部开发完成后递交测试，测试出Bug后进行修复的分支。
- Features: 功能分支；其实Features不是一个分支，而是一个分支文件夹。里面包含了每个程序员开发的功能点。Feature开发完成后合入Develop分支。
- HotFix: 最希望不会被创建的分支；这个分支的存在是在已经正式上线的版本中，发现了重大Bug进行修复的分支。

## Git保存数据的原理

**Git保存数据是直接记录快照，而非差异比较。**这是 Git 与几乎所有其它版本控制系统的重要区别。

在Git中，每当你提交更新或保存项目状态时，它基本上就会对当时的全部文件创建一个快照并保存这个快照的索引。为了效率，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**。



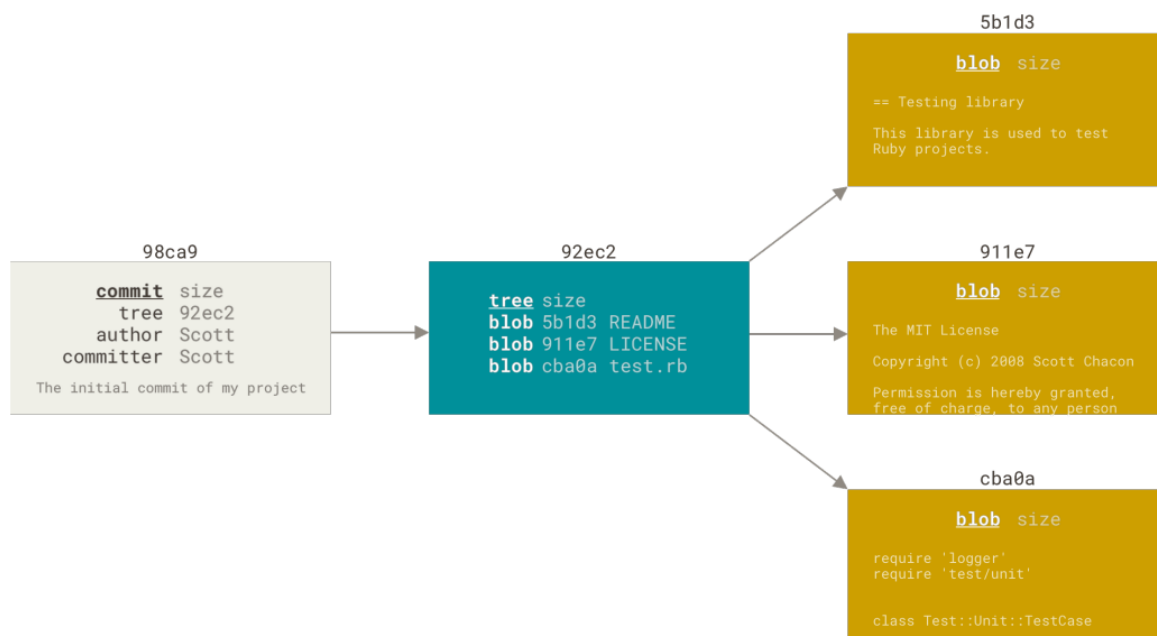
**Git的三大对象：commit对象（提交对象）、tree对象（树对象）、blob对象。**

举例：工作区有README、test.rb、LICENSE三个文件，暂存并提交。

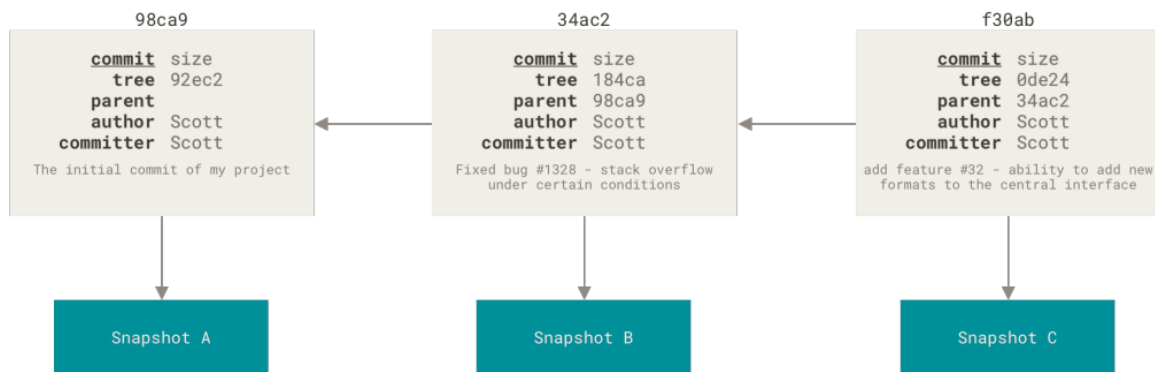
```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

当执行git commit操作后，Git会创建以下5个对象：1个提交对象，1个树对象，以及3个blob对象。其中：

- 提交对象包括：树对象的hash值、作者、以及父对象的hash值（首次提交没有父对象）
- 树对象包括：blob对象的hash值、文件名
- blob对象包括：文件快照



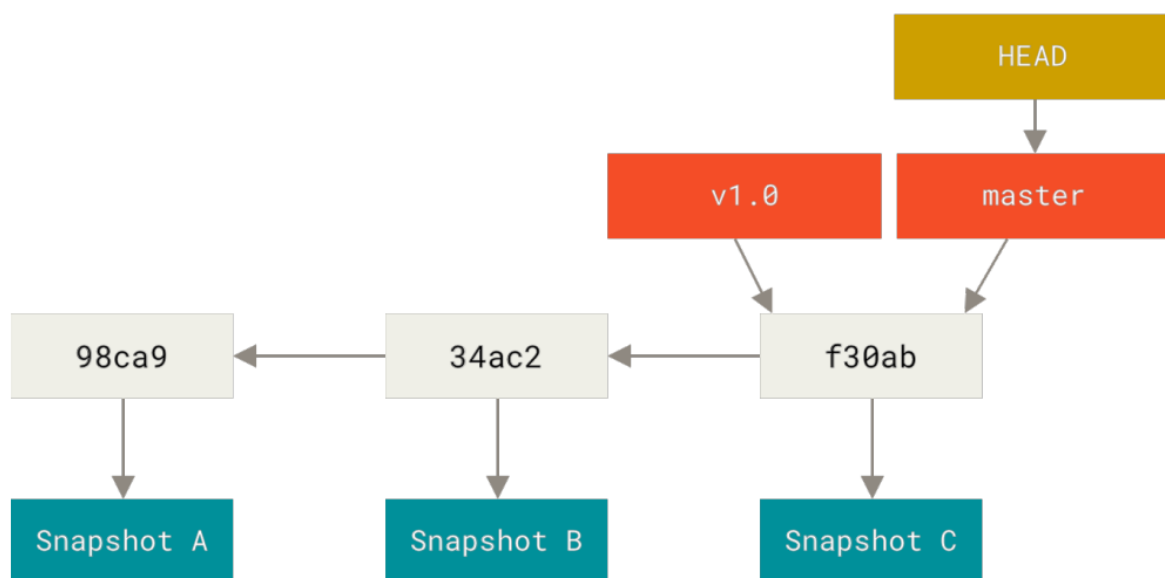
当对文件做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



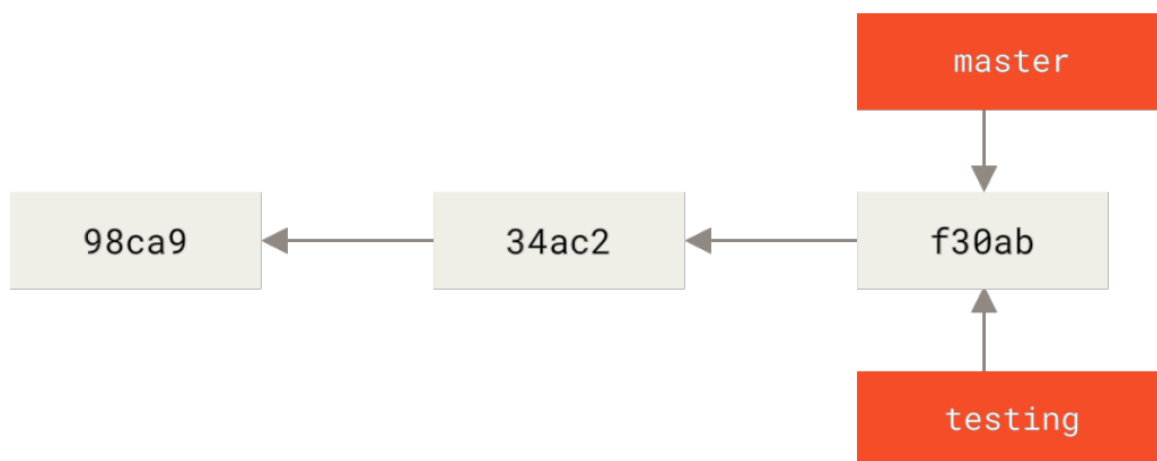
## Git分支的实现

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。master 分支会在每次提交时自动向前移动。

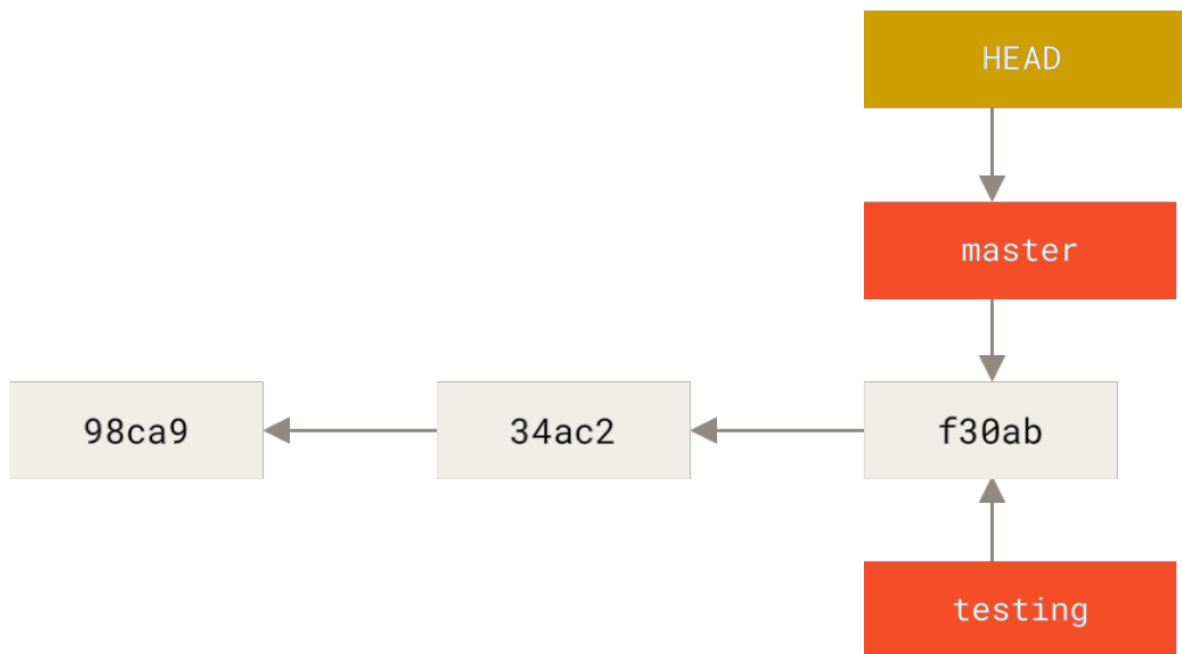
分支及其提交历史：



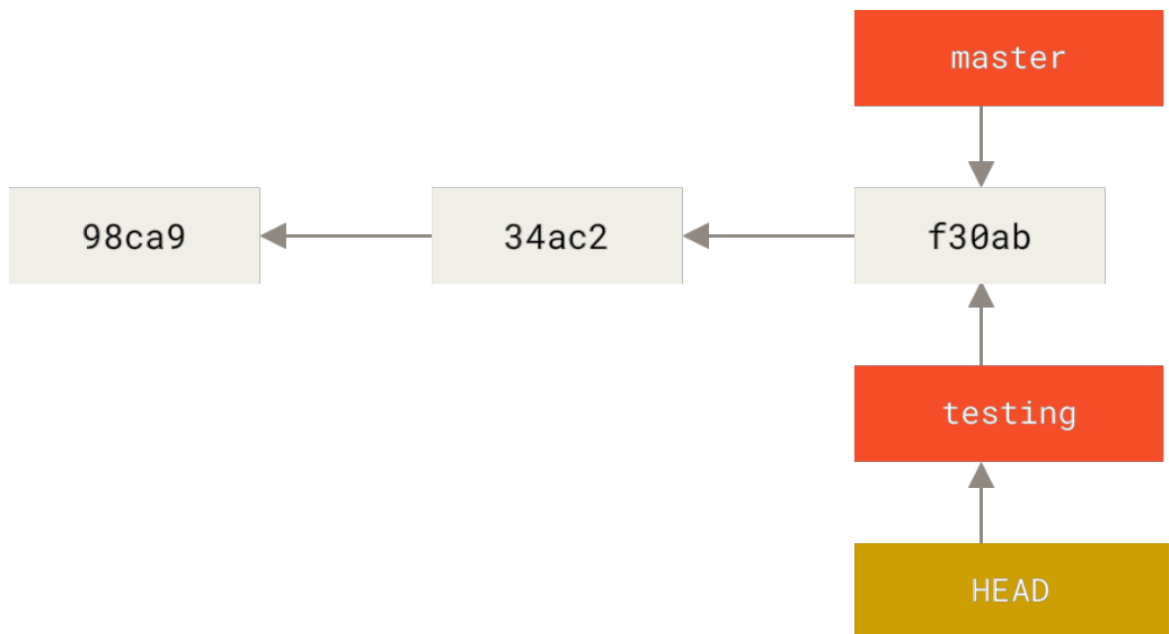
Git创建分支，只是为你创建了一个可以移动的新的指针，比如创建了一个testing分支，这会在当前所在的提交对象上创建一个指针。



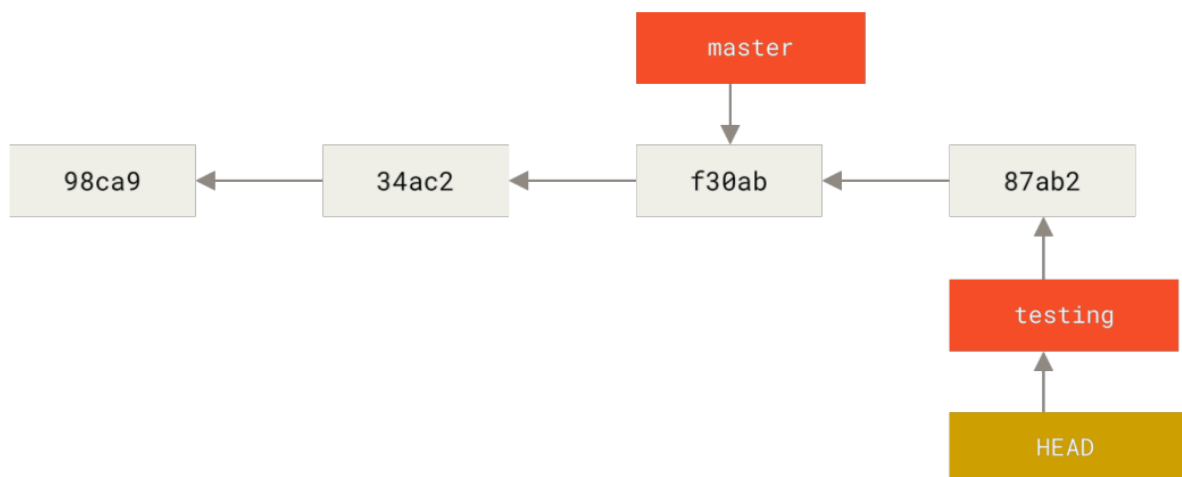
HEAD指针：当Git有多个分支的时候，通过HEAD指针来表示当前所在分支，HEAD是一个特殊指针，指向当前所在的本地分支（可以将 HEAD想象为当前分支的别名）。如下表示当前所在分支是master：



切换到testing分支，HEAD指针指向testing分支：

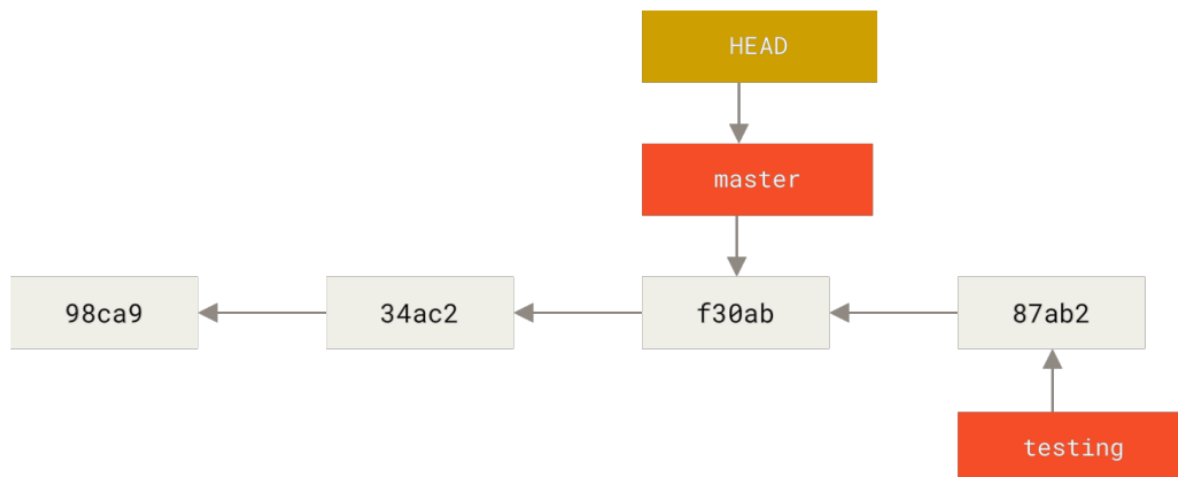


在testing分支提交代码，testing分支会向前移动，但master分支不变：

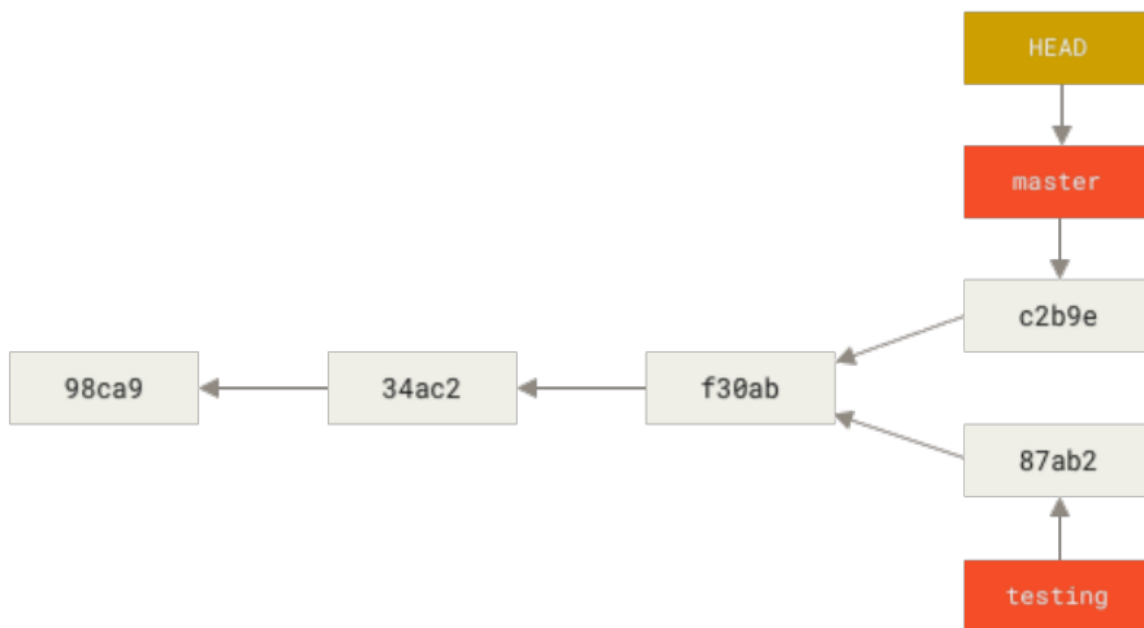




这时再切换回master分支，HEAD指针指向master分支，同时将工作目录恢复成 master 分支所指向的快照内容。也就是说，你现在做修改的话，项目将始于一个较旧的版本。本质上来讲，这就是忽略 testing 分支所做的修改，以便于向另一个方向进行开发。



项目分叉：切换回master分支后，做一些修改后提交，现在，这个项目的提交历史已经产生了分叉。因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。



## git branch

- 列出所有本地分支：**git branch**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git branch
* master
```

前面带\*的表示是当前分支

- 新建一个分支，但不切换到新建的分支：**git branch 新分支名称**，如下表示新建一个rhtest分支

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git branch rhtest
```

- 删除本地分支: **git branch -d 分支名称**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (ronghuatest)
$ git branch -d rhtest
Deleted branch rhtest (was fff1313).
```

注意: 不能直接删除当前所在的分支

## git checkout

- 切换到指定分支, 并更新工作区: **git checkout 已有分支名称**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (ronghuatest)
$ git checkout rhtest
Switched to branch 'rhtest'
```

- 新建一个分支, 并切换到该分支: **git checkout -b 新分支名称**, 这个命令相当于 **git branch 新分支名称** 和 **git checkout 新分支名称** 两条命令的简写

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git checkout -b ronghuatest
Switched to a new branch 'ronghuatest'
```

- 切换到上一个分支: **git checkout -**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (rhtest)
$ git checkout -
Switched to branch 'ronghuatest'
```

## git merge

- 合并指定分支到当前分支: **git merge 分支名称**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (test)
$ git merge hotfix
Updating 3be115a..bd04598
Fast-forward
 test33.py | 2 + -
 1 file changed, 1 insertion(+), 1 deletion(-)
```

# 冲突与解决冲突

git合并分支的时候，如果存在以下情况就会出现冲突：

1. 两个分支中修改了同一个文件（不管什么地方）
2. 两个分支中修改了同一个文件的名称

如果两个分支中分别修改了不同文件中的内容，则不会产生冲突，可以直接将两部分合并。

git产生冲突的场景：

1. 场景一：多个分支代码合并到一个分支时；
2. 场景二：多个分支向同一个远端分支推送代码时，先推送的不会冲突，后推送的会冲突；

解决冲突的方法：找到冲突的文件，直接修改冲突代码，然后add，再commit，最后再merge。注意要删除git自动生成的冲突代码分隔符。

演示：出现冲突，并解决冲突。

```
#1、在master分支上修改demo_1.py文件，并提交
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (master)
$ vi demo_1.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (master)
$ git add .

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (master)
$ git commit -m "2022040601"
[master d9b681e] 2022040601
 1 file changed, 1 insertion(+), 1 deletion(-)

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (master)
$ git checkout -
Switched to branch 'hotfix'

#2、在hotfix分支上修改demo_1.py文件，并提交
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (hotfix)
$ vi demo_1.py

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (hotfix)
$ git add .

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (hotfix)
$ git commit -m "2022040602"
[hotfix 2e3d0f3] 2022040602
 1 file changed, 1 insertion(+), 1 deletion(-)

#3、合并master分支到hotfix分支，出现冲突
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (hotfix)
$ git merge master
Auto-merging demo_1.py
CONFLICT (content): Merge conflict in demo_1.py
Automatic merge failed; fix conflicts and then commit the result.

#可以通过git status命令查看冲突的文件
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh_testing_40 (hotfix|MERGING)
$ git status
```

```
On branch hotfix
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:   demo_1.py
```

#通过cat命令查看文件内容时，可以看到冲突的具体代码，这时发现文件中自动生成了冲突代码分隔符  
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/rh\_testing\_40 (hotfix|MERGING)

```
$ cat demo_1.py
```

#输出hello丹尼尔

```
print("hello world")
```

```
a=100
```

```
<<<<<<< HEAD
```

```
a+=3
```

```
=====
```

```
a+=2
```

```
>>>>>>> master
```

#4、解决冲突：首先要判断以哪个分支的代码为准，然后修改文件，再次add、commit、merge

## 8. 远程操作

### 本地仓库关联远程仓库

#### git remote

对于使用 **git init** 命令初始化的本地仓库，一开始是没有与之关联的远程仓库的，需要使用 **git remote** 命令来关联远程仓库，对于 **git clone** 克隆的本地仓库，默认是与克隆的远程仓库保持关联的。

关联本地仓库与远程仓库: **git remote add origin 远程仓库URL**，例如: **git remote add origin git@gitee.com:charlesshenchuan/rong-hua-test.git**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git remote add origin git@gitee.com:charlesshenchuan/rong-hua-test.git
```

在Gitee上获取远程仓库URL的方法：

Charles.Shen / RongHuaTest

代码 Issues Pull Requests Wiki 统计 DevOps 服务

master 分支 2 标签 0

+ Pull Request + Issue 文件 Web IDE 克隆/下载 简介

Charles.Shen 2.0版本代码提交 c90f30b 5个月前

.gitignore Initial commit

Demo.txt 第一次提交代码

HTTPS SSH SVN SVN+SSH

git@gitee.com:charlesshenchuan/rc 复制

下载ZIP

查看关联的远程仓库：**git remote -v**，关联远程仓库后可以通过这个命令来查看关联的远程仓库的详细信息，如下表示fetch和push对应的远程仓库都是rong\_hua\_test

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git remote -v
origin  git@gitee.com:charlessenchuan/rong-hua-test.git (fetch)
origin  git@gitee.com:charlessenchuan/rong-hua-test.git (push)
```

## 本地分支追踪远程分支

建立本地分支与远程分支的追踪关系之后，在执行git pull或git push命令时，就不必每次都要指定从远程的哪个分支拉取合并和推送到远程的哪个分支。

查看当前分支与远程分支对应关系：**git branch -vv**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git branch -vv
* master fff1313 [origin/master] 修改了hellow world添加了第三行
```

新建一个本地分支，并与远程分支建立追踪关系：**git branch --track 本地分支名称 远程仓库名称/远程分支名称**，通常保持本地分支名称和远程分支名称相同

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (ronghuatest)
$ git branch --track test origin/test
Branch 'test' set up to track remote branch 'test' from 'origin'.
```

新建一个本地分支，并与远程分支建立追踪关系，新建的本地分支名称与远程分支保持相同，并切换到该分支：**git checkout --track origin/远程分支名称**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git checkout --track origin/dev
Switched to a new branch 'dev'
branch 'dev' set up to track 'origin/dev'.
```

对现有分支与远程分支，建立分支追踪关系：**git branch --set-upstream-to=远程仓库名称/远程分支名称 本地分支名称**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (ronghuatest)
$ git branch --set-upstream-to=origin/dev ronghuatest
Branch 'ronghuatest' set up to track remote branch 'dev' from 'origin'.
```

以上表示把本地的分支ronghuatest追踪到远程分支dev

查看远程分支：**git branch -r**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git branch -r
origin/dev
origin/master
```

列出所有本地分支和远程分支：**git branch -a**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git branch -a
* master
remotes/origin/dev
remotes/origin/master
```

查看远程仓库与本地仓库的关系：**git remote show origin**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (master)
$ git remote show origin
* remote origin
Fetch URL: git@gitee.com:charlesshenchuan/rong-hua-test.git
Push URL: git@gitee.com:charlesshenchuan/rong-hua-test.git
HEAD branch: master
Remote branches:
  dev      tracked
  master   tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (up to date)
```

## 从远程仓库拉取代码

### git fetch

获取远程仓库的指定分支更新：**git fetch 远程主机名 分支名**，拉取下来后需要执行 **git merge** 命令合并远程分支到你所在的分支

```
#获取更新
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (hotfix)
$ git fetch origin hotfix
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 955 bytes | 50.00 KiB/s, done.
From gitee.com:charlesshenchuan/rong-hua-test32
 * branch                hotfix      -> FETCH_HEAD
   20375ef..e5dcf4b hotfix      -> origin/hotfix

#合并到当前分支
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (hotfix)
```

```

$ git merge origin/hotfix
Updating 20375ef..e5dcf4b
Fast-forward
 test38.py | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
#合并后可以看到最新的代码
lenovo@LAPTOP-OUBMTCD1 MINGW64 ~/Desktop/GitDemo38_2/rhtest33 (hotfix)
$ cat test38.py
在Gitee上更新后的文件
222222222222

```

对于本地分支追踪了远程分支的场景，可以直接指定**git fetch**命令来获取远程分支的更新。

```

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (release)
$ git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 998 bytes | 83.00 KiB/s, done.
From gitee.com:charlessenchuan/rong-hua-test
   c90f30b..04e51cc  release    -> origin/release

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (release)
$ git merge origin/release
Updating c90f30b..04e51cc
Fast-forward
 login_bpm_1.csv | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 login_bpm_1.csv

```

## git pull

拉取指定分支的更新并直接合并: **git pull 远程主机名 远程分支名 : 本地分支名**

```

Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (hotfix)
$ git pull origin hotfix:hotfix
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 957 bytes | 53.00 KiB/s, done.
From gitee.com:charlessenchuan/rong-hua-test32
   e5dcf4b..b127e1e  hotfix      -> hotfix
   e5dcf4b..b127e1e  hotfix      -> origin/hotfix
warning: fetch updated the current branch head.
fast-forwarding your working tree from
commit e5dcf4bfe1dde9048725bf0073bbba7a5e1e8117.
Already up to date.

```

注意:

1. git pull命令实际上是git fetch和git merge的合并写法。



2. 如果远程分支是与当前分支合并，则冒号后面的部分可以省略，比如：`git pull origin master`表示拉取远程的master分支与本地当前分支合并。
3. 对于本地分支追踪了远程分支的情况，可以直接执行`git pull`命令来拉取远程分支的更新到本地。

## 推送代码到远程仓库

### git push

将本地分支的更新，推送到远程主机: **git push 远程主机名 本地分支名:远程分支名**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (test)
$ git push origin test:rhtest
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 723 bytes | 361.00 KiB/s, done.
Total 7 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Powered by GITEE.COM [GNK-6.2]
remote: Create a pull request for 'rhtest' on Gitee by visiting:
remote:   https://gitee.com/charlesshenchuan/rong-hua-test32/pull/new/charlesshenchuan:rhtest...charlesshenchuan:master
To gitee.com:charlesshenchuan/rong-hua-test32.git
* [new branch]      test -> rhtest
```

说明:

1. 如果指定的远程分支不存在，则会自动创建远程分支。
2. 如果本地分支名与远程分支名相同，则可以省略冒号，比如`git push origin test`等效于`git push origin test:test`。
3. 对于本地分支追踪了远程分支的情况，可以直接执行`git push`命令推动本地分支的更新到远程分支。

- 将本地分支推送到远程仓库，并让本地分支追踪远程分支: **git push -u 远程主机名 本地分支名**

```
Charles.Shen@HuaweiLaptop MINGW64 /d/GitDemo/demo_repo (test)
$ git push -u origin rhtest
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote: Powered by GITEE.COM [GNK-6.2]
remote: Create a pull request for 'rhtest' on Gitee by visiting:
remote:   https://gitee.com/charlesshenchuan/rong-hua-test32/pull/new/charlesshenchuan:rhtest...charlesshenchuan:master
To gitee.com:charlesshenchuan/rong-hua-test32.git
* [new branch]      rhtest -> rhtest
Branch 'rhtest' set up to track remote branch 'rhtest' from 'origin'.
```

## 9. GitLab的介绍

### 什么是GitLab

GitLab是一个利用Ruby on Rail开发的开源应用程序，用于实现一个自托管的Git项目仓库，可以通过Web界面进行访问公开的或者私人项目。使用Git作为代码管理工具，并在此基础上搭建起来的Web服务。GitLab拥有与GitHub类似的功能，能够浏览源代码，管理缺陷和注释，可以管理团队对仓库的访问。GitLab是专门为Unix操作系统开发的，不可运行在Windows操作系统上面，所以需要在Linux服务器或者Docker上安装。

## Git、GitHub、Gitee、GitLab的关系



- Git: Git是一种基于命令的版本控制系统，全命令操作，没有可视化界面。
- GitHub: GitHub是一个基于Git实现的**在线代码托管仓库**，提供可视化管理界面，同时提供社区版和企业版，提供开放和私有的仓库，大部分的开源项目都选择GitHub作为代码托管仓库。
- Gitee: 中国版的GitHub，是与GitHub类似的在线代码托管仓库，因为受网络影响，访问GitHub不稳定，所以国内越来越多的开发者选择Gitee托管开源代码。

Gitee也分为社区版和企业版，它们的服务对比：

功能特性	社区版		企业版	
使用场景	开源项目	个人私有仓库	免费版	协作开发
总计协作人数	不限	≤ 5 人	≤ 5 人	20 ~ 100 人+
仓库总容量	5G	5G	5G	20 ~ 100G
单仓库大小	1G	500M	500M	1 ~ 3G
单文件大小	50M	50M	100M	100 ~ 300M
附件总容量	3G	3G	3G	10 ~ 50G +

- GitLab是一个基于Git实现的在线代码仓库软件，提供Web可视化管理界面，通常用于企业团队内部协作开发。当企业不想把代码托管到公共的仓库平台时，就会选择私有部署GitLab。

## GitLab的安装（课后自己练习安装）

如下是在CentOS7上安装GitLab的详细步骤。

#### 1、安装所需依赖

```
[root@localhost ~]# yum install -y curl policycoreutils-python openssh-server postfix
```

#### 2、启动postfix服务，并设置开机自启动

```
[root@localhost ~]# service postfix start  
[root@localhost ~]# chkconfig postfix on
```

#### 3、下载GitLab（CentOS7.X版本，如果为CentOS6.x，将el7修改为el6）

```
[root@localhost ~]# cd /usr/local/src  
[root@localhost ~]# wget https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el7/gitlab-ce-14.9.2-ce.0.el7.x86_64.rpm --no-check-certificate
```

#### 4、安装GitLab

```
#先安装依赖包policycoreutils-python  
[root@localhost ~]# yum install -y policycoreutils-python  
#安装gitlab  
[root@localhost ~]# rpm -ivh gitlab-ce-14.9.2-ce.0.el7.x86_64.rpm
```

#### 5、修改GitLab端口号、存储仓库修改为自定义路径

```
[root@localhost ~]# vi /etc/gitlab/gitlab.rb
```

#### 6、配置GitLab(配置完自动启动，默认帐号为root)

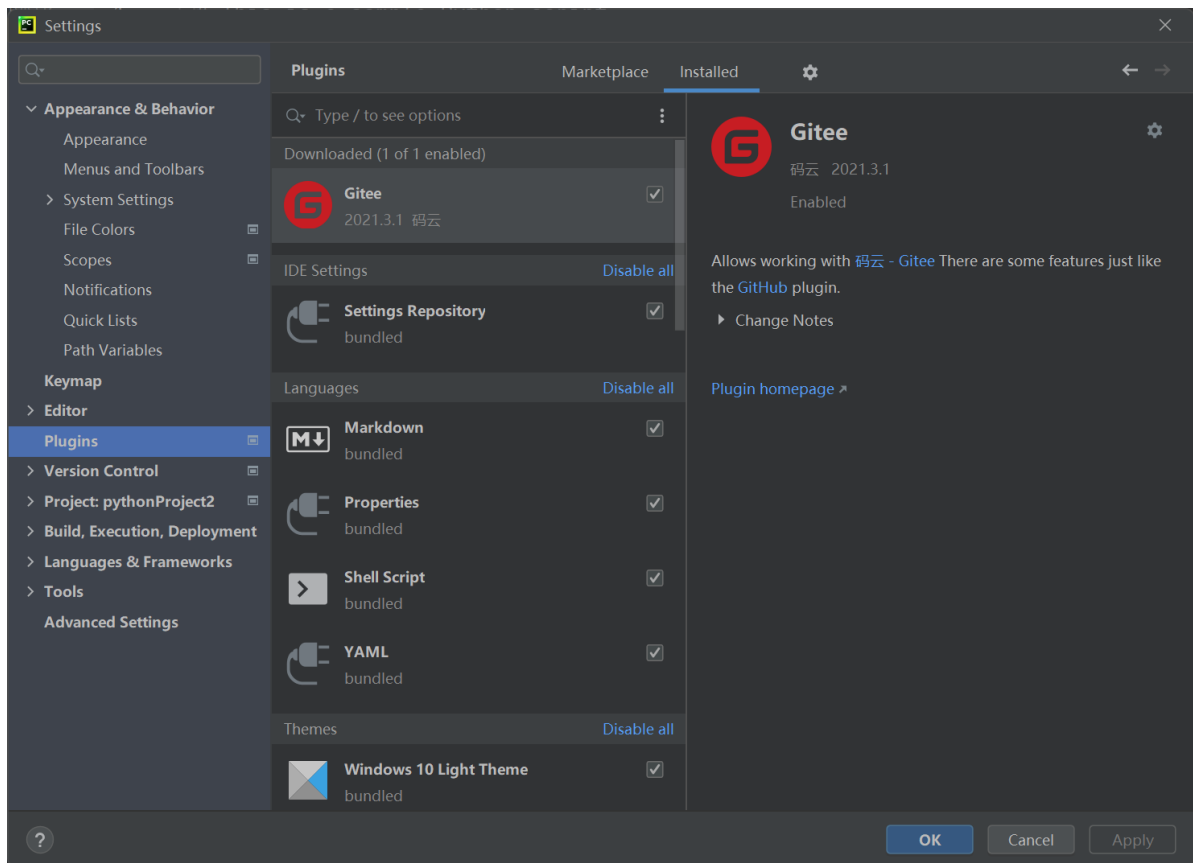
```
[root@localhost ~]# gitlab-ctl reconfigure
```

#### 7、重启关闭启动

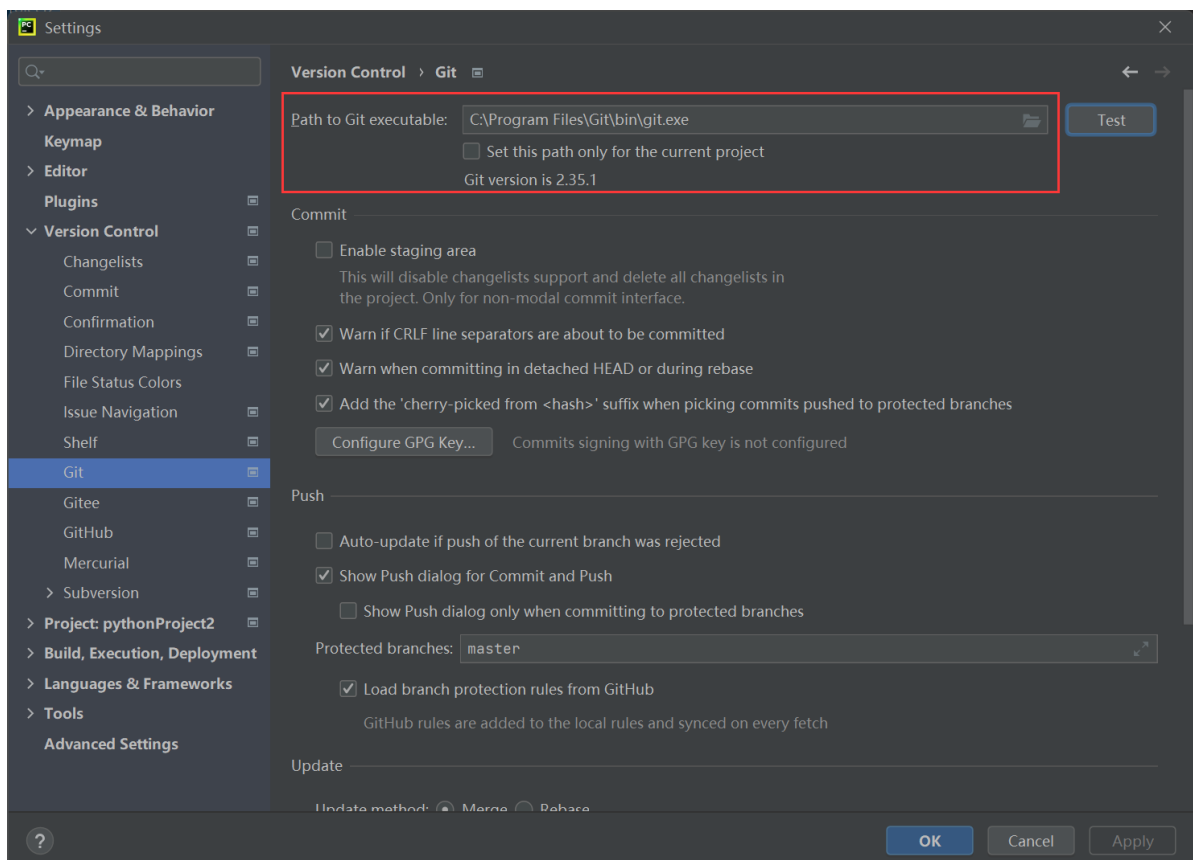
```
[root@localhost ~]# gitlab-ctl restart  
[root@localhost ~]# gitlab-ctl stop  
[root@localhost ~]# gitlab-ctl start
```

## 10. 配置PyCharm集成Git

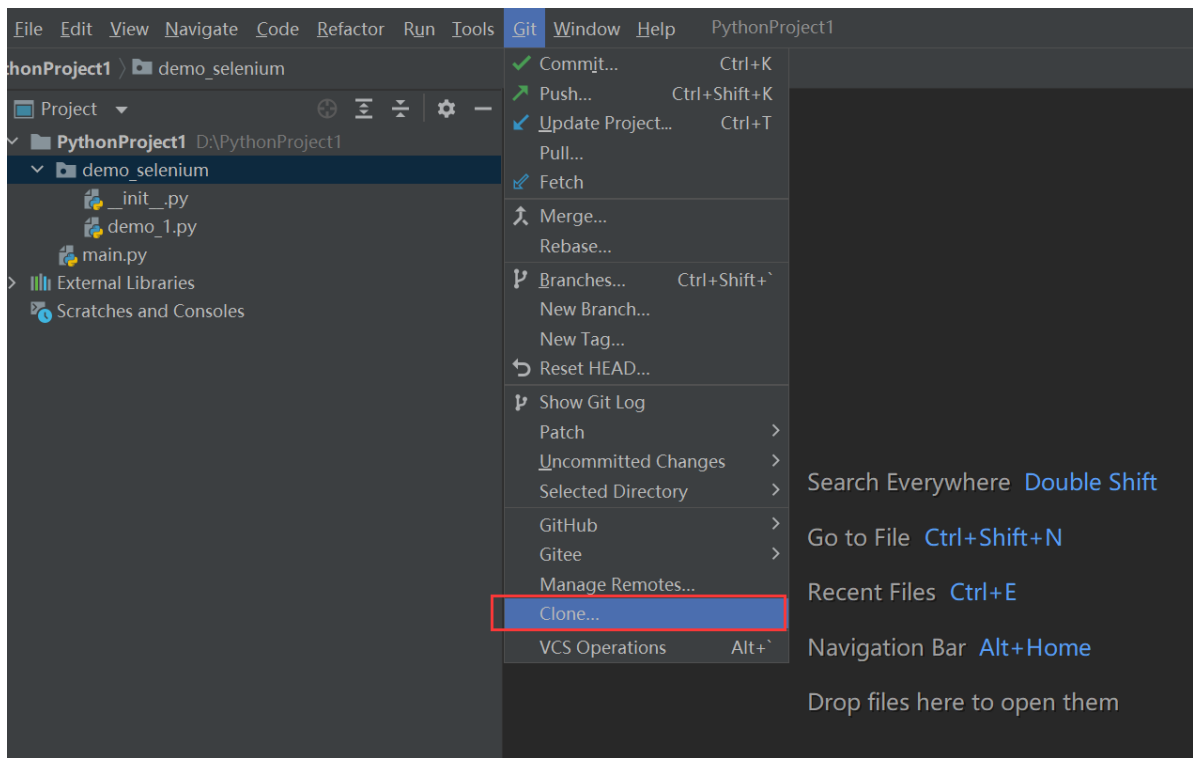
在PyCharm上安装Gitee插件：



设置Git:



从Gitee克隆项目到本地:



演示：Add、Commit、Push、Pull

