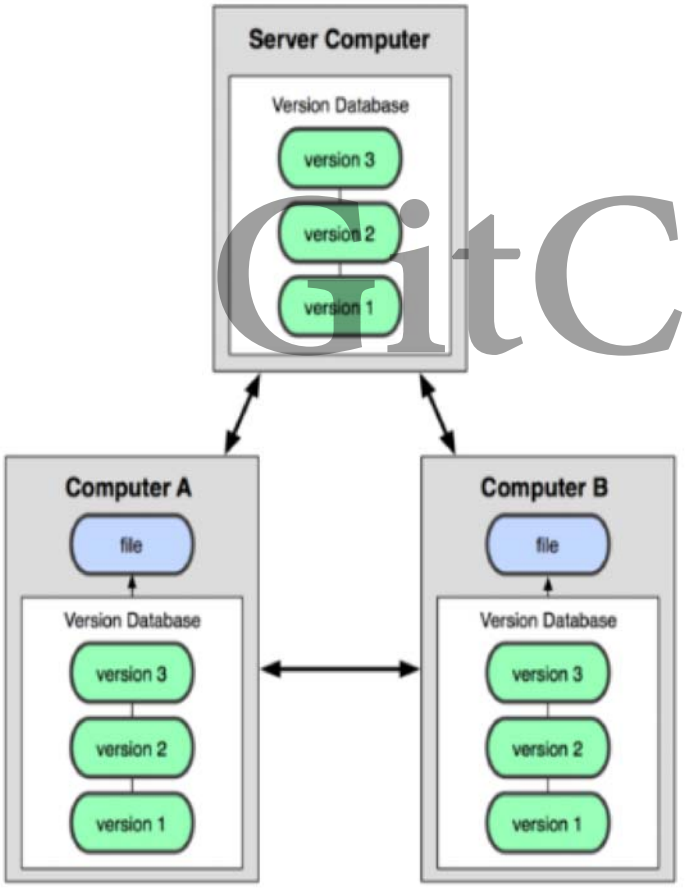
让你一场 Chat 学会 Git

一、Github基础

什么是Git?

git是一个分布式版本控制软件，最初由林纳斯·托瓦兹（Linus Torvalds）（Linux之父）创作，于2005年发布。最初目的是为更好地管理Linux内核开发。Git 在本地磁盘上就保存着所有有关当前项目的历史更新，处理速度快；Git 中的绝大多数操作都只需要访问本地文件和资源，不用实时联网。



Git客户端

TortoiseGit是一个Git版本控制客户端，作为Microsoft Windows的外壳扩展实现，用户界面友好，大多数人应该用过TortoiseSvn；

MsysGit是一个轻量级的 Git 工具集，可以进行各种 Git 操作，MsysGit又分为简单的界面 Git GUI，和命令行Git Bash，我们这节课主要通过Git Bash来演示。

Git服务端：GitHub 和Gitlab

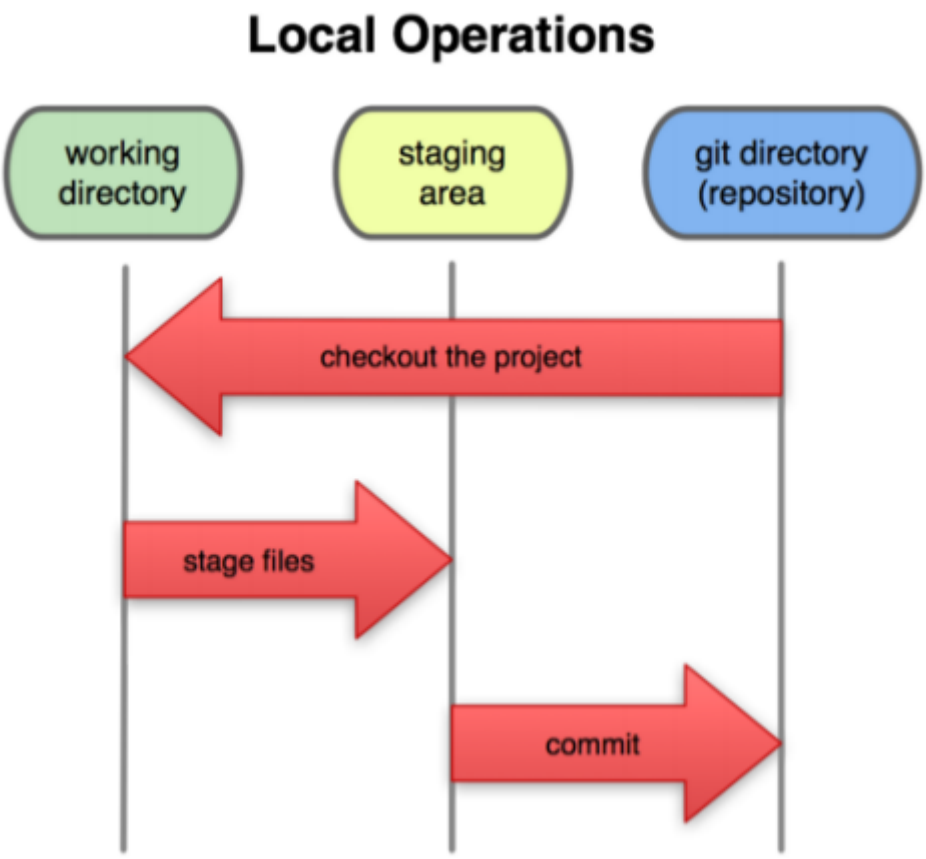
GitHub 是一个共享虚拟主机服务，用于存放使用Git版本控制的软件代码和内容项目；允许用户跟踪其他用户、组织、软件库的动态，对软件代码的改动和 bug 提出评论。

GitLab是一个利用 Ruby on Rails 开发的开源应用程序，实现一个自托管的Git项目仓库，可通过Web界面进行访问公开的或者私人项目，拥有与Github类似的功能，能够浏览源代码，管理缺陷和注释。

Git基本概念

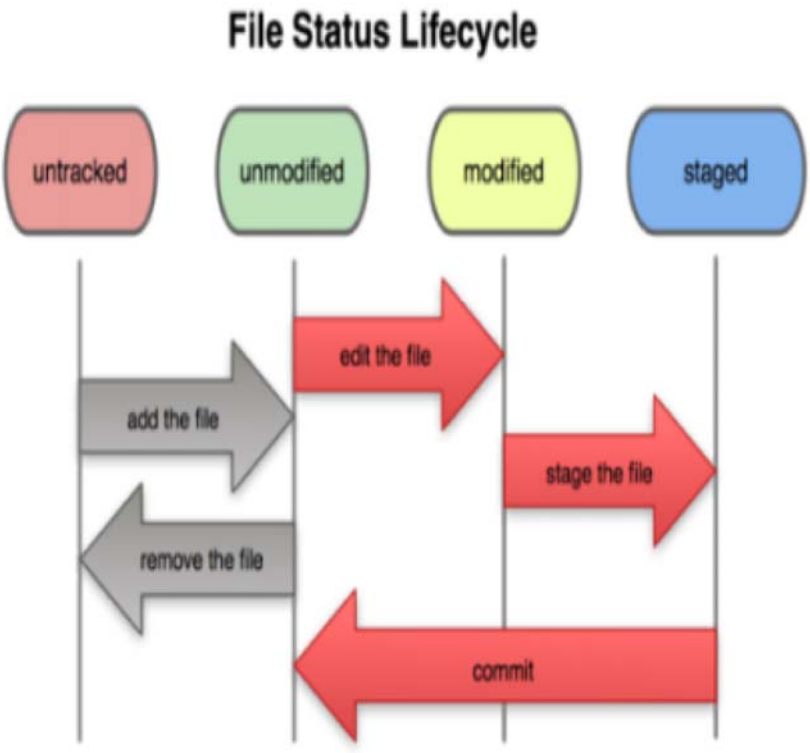
三种工作区域

1. Git 的本地仓库：在.git 目录中
2. 工作区：用户操作目录
3. 暂存区:在.git 目录中

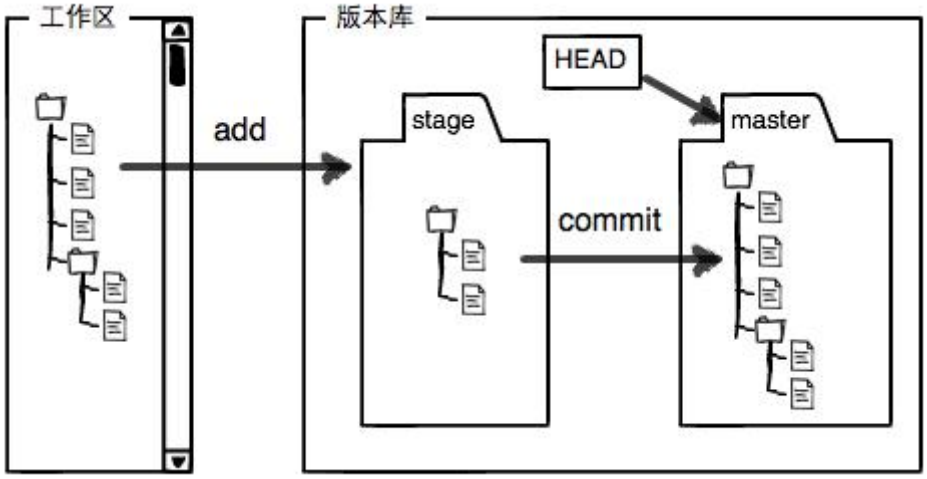


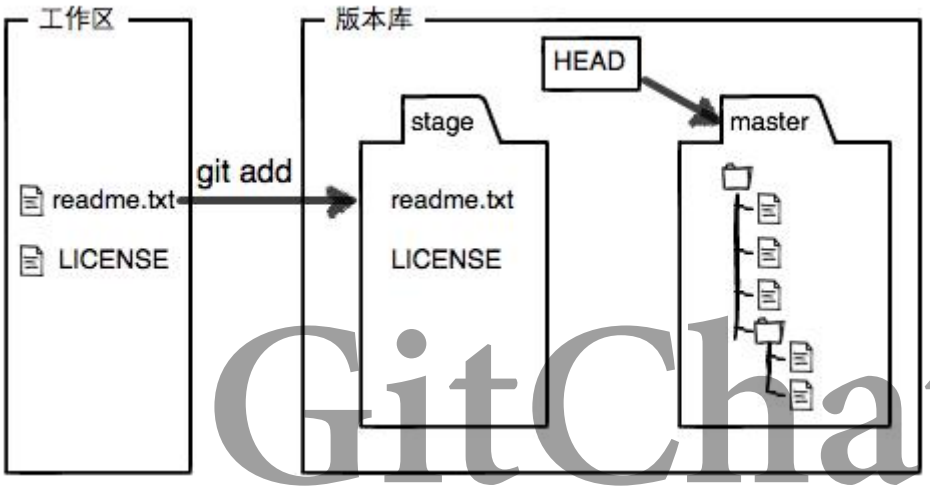
三种状态

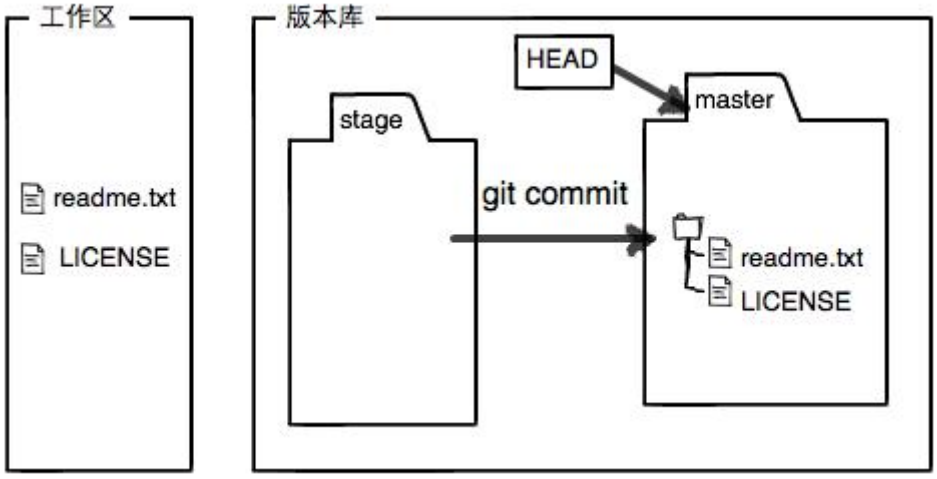
1. 已提交（committed）:该文件已经被安全地保存在本地仓库中
2. 已修改（modified）:修改了某个文件，但还没有提交保存
3. 已暂存（staged）:把已修改的文件放在下次提交时要保存的清单中



注：状态转换图示如下







Git 分支初识

1. Git 中的分支，其实本质上仅仅是个指向 commit 对象的可变指针。
2. Git 会使用 master 作为分支的默认名字。在若干次提交后，你其实已经有了一个指向最后一次提交对象的 master 分支，它在每次提交的时候都会自动向前移动。
3. Git 鼓励在工作流程中频繁使用分支与合并。

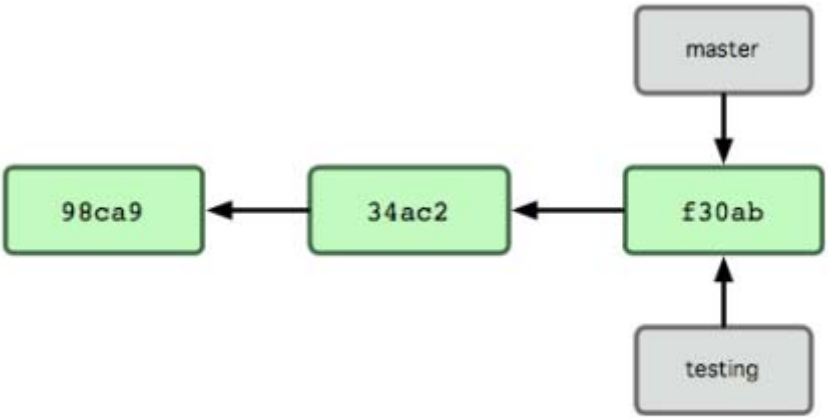


图 a ： 多个分支指向提交数据的历史

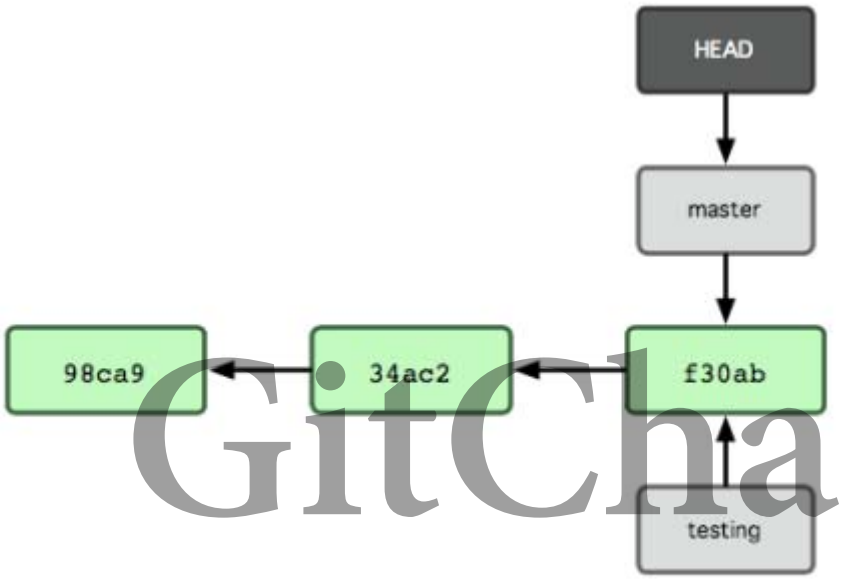


图 b ： HEAD 指向当前所在的分支

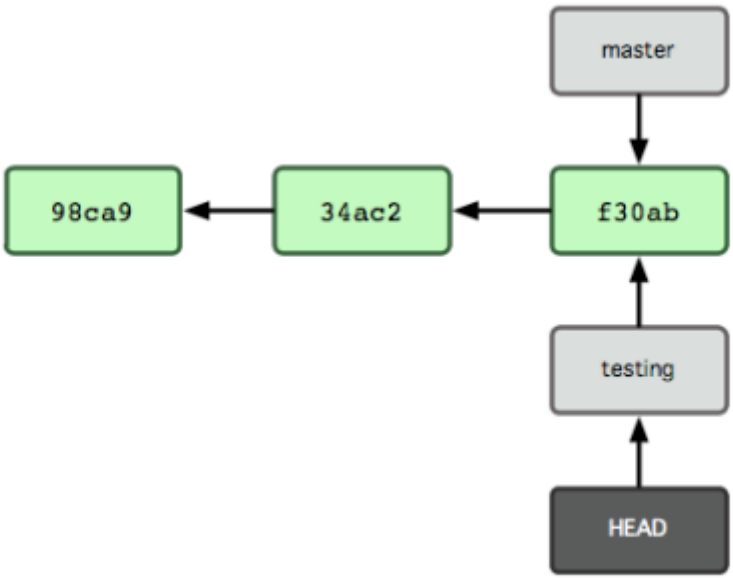


图 c ： HEAD 在你转换分支时指向新的分支

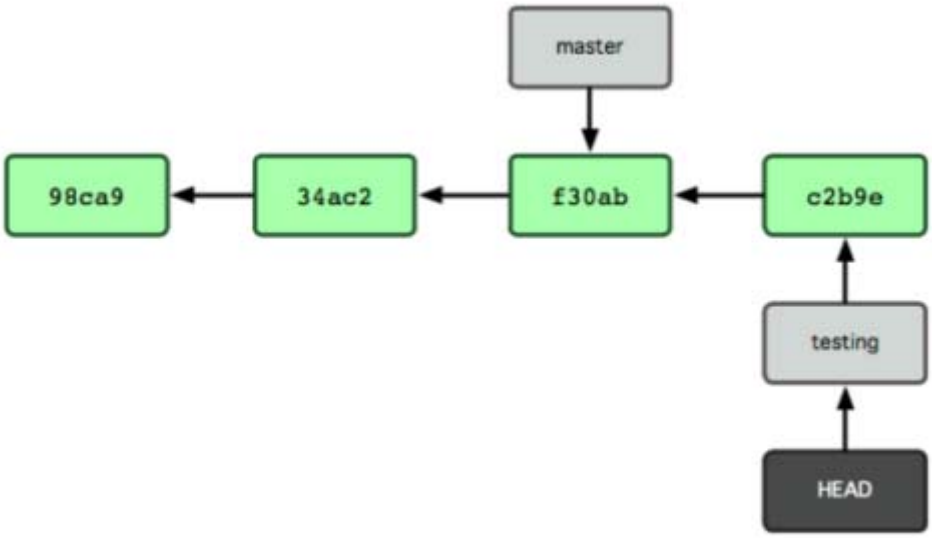


图 d：每次提交后 HEAD 随着分支一起向前移动

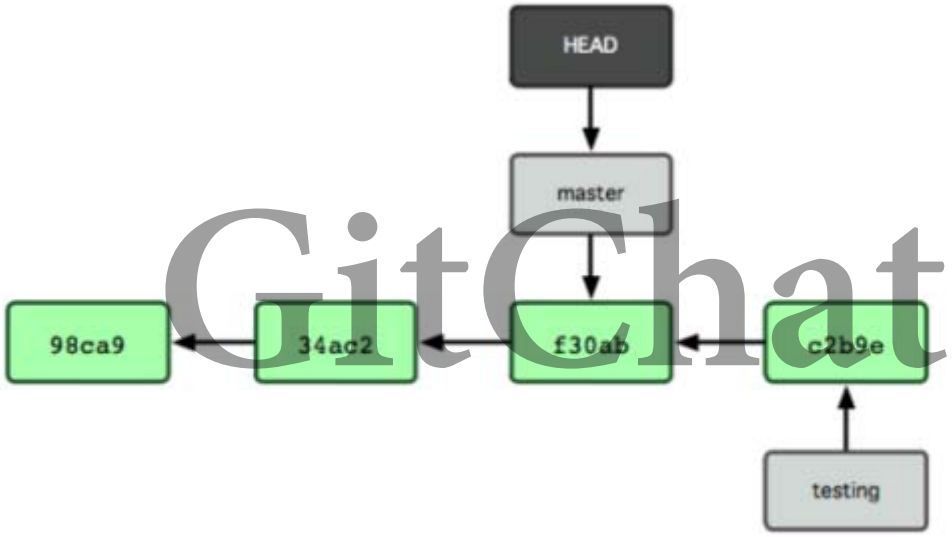
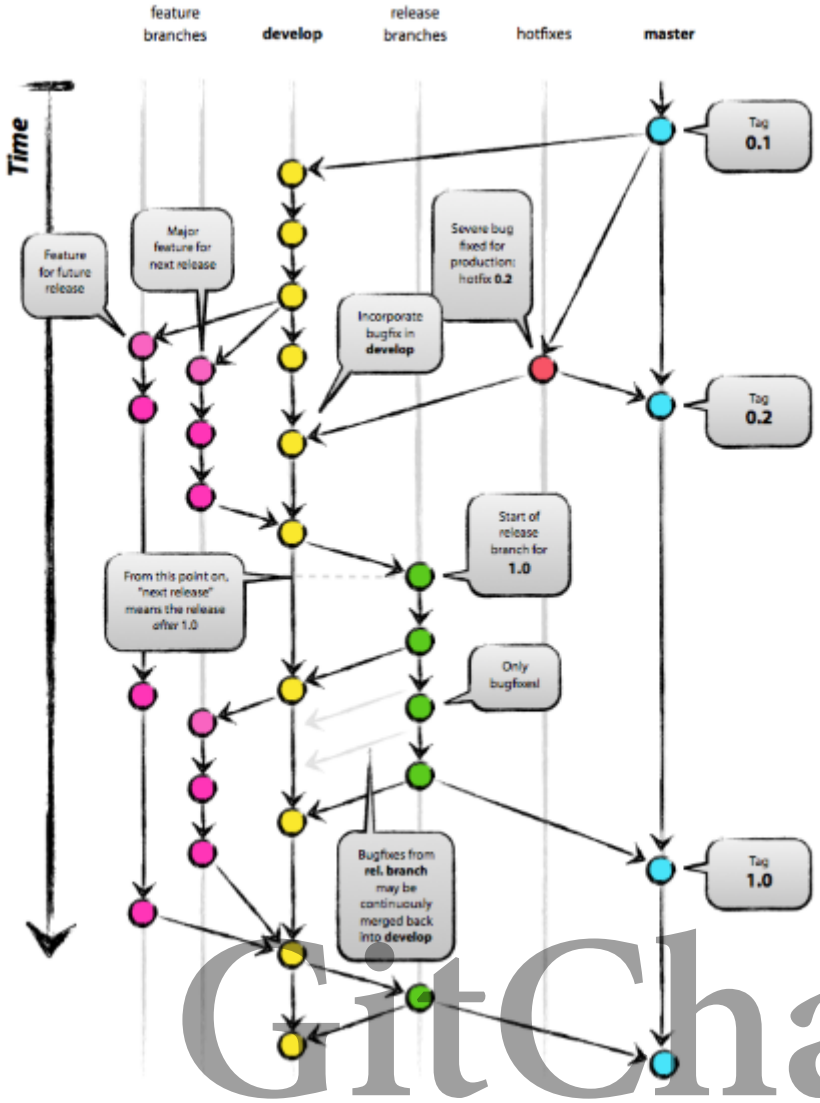


图 e：HEAD 在一次 checkout 之后移动到了另一个分支

二、Git 工作流程

git的工作流程见下图，这是最标准的git使用流程，我们可以看到git是通过对各分支的维护来规范工作的。

首先有一条master分支，这是发布稳定版，一般不会对master进行修改，当有新需求的时候，从master上分出一条develop分支，具体开发人员在从develop上分出自己具体feature分支，开发完毕后合并回develop分支，测试人员从develop分支分出release分支进行测试，没有问题了合并到master分支发布。如果线上有bug需要处理，则从master上分出hotfix分支，用来解决bug。



三、Git 基本操作

配置用户名和邮件

打开git bash，配置命令：

```
git config --global user.name "your name"
git config --global user.email "your email"
```

创建版本库

版本库又名仓库，英文名repository，你可以简单理解成一个目录，这个目录里面的所有文件都可以被

Git管理起来：

1. 新建目录初始化

```
mkdir testgit
cd testgit
```

2. 从当前目录初始化

```
$ git init
```

查看仓库状态：

```
git status
```

添加到暂存区：

```
git add fileA fileB fileC ....
```

提交到本地仓库：

```
git commit -m “remarks”
```

查看修改内容（工作区和仓库的区别）：

```
git diff （file）
```


查看版本（参数可以简化版本信息，commit id和备注）：

```
git log --pretty=oneline
```

版本回退

在 Git中，用HEAD表示当前版本，也就是最新的提交commit id，上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

回退到上一个版本：

```
git reset --hard HEAD^
```

丢弃工作区的修改（撤销）：

```
git checkout -- file
```

删除文件:

1. 正确操作：git rm file git commit -m “remove file”（文件被删除）
2. 操作失误：git checkout – file（文件被恢复）

查看当前分支：

```
git branch (-a)
```

新建分支：

```
git branch develop（只是新建了一条分支，并未切换）
```

切换分支：

```
git checkout develop
```

新建并切换分支：

```
git checkout -b feature（相当于3.10和3.11两步操作）
```

删除分支：

```
git branch -d feature（注意：不能删除当前所在分支）
```


合并分支(–no-ff参数，表示禁用Fast forward)：

```
git checkout develop  &&  git merge feature（把feature分支合并到develop分支）
```

注：

1. 因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以默认git commit就是往master分支上提交更改。
2. 如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <branch>` 强行删除。
3. 建立本地分支和远程分支的关联，使用git branch –set-upstream branch-name origin/branch-name；

Git 基本操作—和远程服务器交互（一）

添加远程服务器：

```
git remote add dcmsStatics4.5git(别名）
http://gitlab.cephchina.com/ccod_project/dcmsstatics4-5git.git
```

查看远程服务器的相关信息：

```
git remote -v
git remote show dcmsStatic4.5git
```

重命名远程仓库信：

```
git remote rename demo test
```

删除远程仓库：

```
git remote rm test
```

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

之后，只要本地作了提交，就可以通过命令把本地master分支的最新修改推送至GitLab：

```
git push dcmsStatic4.5git master
```


远程仓库

从远程仓库获取数据：

- git fetch origin develop — 只是获取远程仓库的数据至 .git 目录，并未merge本地
- git merge origin/develop — 把获取的远程仓库的数据手工merge至当前分支
- git pull origin develop —- 获取远程仓库的数据，并自动merge至当前的分支，相当于以上两步

合并两个不同的项目：

```
--allow-unrelated-histories
```

把本地仓库的内容推送到远程库上：

```
git push (-u) demo develop（从svn迁移到gitlab注意路径，要确保路径正确）
```

注：

从远程分支 checkout 出来的本地分支，称为跟踪分支(tracking branch)。跟踪分支是一种和远程分支有直接联系的本地分支。在跟踪分支里输入git push，Git 会自行推断应该向

哪个服务器的哪个分支推送数据。反过来，在这些分支里运行git pull 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

在克隆仓库时，Git 通常会自动创建一个名为 master 的分支来跟踪 origin/master。这正是git push 和 git pull 一开始就能正常工作的原因。当然，你可以随心所欲地设定为其它跟踪分支，比如origin 上除了 master 之外的其它分支。

Git 基本操作：和远程服务器交互（二）

从远程库克隆：

1. 从svn克隆 git svn clone 地址
2. 从git远程库上克隆：git clone 地址

在本地创建和远程分支对应的分支，使用 `git checkout -b branch-name origin/branch-name`

四、Git 标签

git标签，通常用来给分支做标记，如标记一个版本号。

创建

标签分类

轻量级标签：

```
git tag <tagname> commit id
```

带说明标签：

```
git tag -a  <tagname> commit id
git tag -m <msg> <tagname> commit id
```

带签名的标签（GPG加密，需安装配置）：

```
git tag -s  <tagname> commit id
git tag -u  <key-id> commit id
```

查看和删除

查看标签：

```
git tag
git tag -n
git show <tagname>
```

删除标签：

```
git tag -d <tagname>
```

共享标签

向上游版本库提交标签：

```
git push origin <tagname>
git push origin  --tags
```

删除远程版本库的标签：

```
git push origin :tag2
```

五、补充

Git 分支冲突解决

如果在不同的分支中都修改了同一个文件的同一部分，Git 就无法干净地把两者合到一起 Git 作了合并，但没有提交，它会停下来等你解决冲突。可以用 git status 查阅哪些文件在合并时出现冲突。

Git 会在有冲突的文件里加入标准的冲突解决标记，可以通过它们来手工定位并解决这些冲突。

注：用带参数的git log也可以看到分支的合并情况：

```
git log --graph --pretty=oneline --abbrev-commit
```

冲突标记<<<<<<<（7个<）与=======之间的内容是我的修改，=======与>>>>>>>之间的内容是别人的修改。 最简单的编辑冲突的办法，就是直接编辑冲突了的文件，把冲突标记删掉，把冲突解决正确。

特殊场景

场景：当接到一个新的bug，急需解决，但是目前工作想保留。

方法：Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：比如你正在dev分支开发，突然接到master上有一个特别急的bug需要解决，这时就可以把当前dev的工作现场“储藏”起来。

首先“储藏”dev的工作现场git status，然后从master创建临时分支：

```
git checkout master
git checkout -b issue-101
```

现在修复bug，然后提交：

```
git add readme.md
git commit -m "fix bug 101"
```

修复完成后，切换到master分支，并完成合并，最后删除issue-101分支：

```
git checkout master
git merge --no-ff -m "merged bug fix 101" issue-101
```

接着回到dev分支干活了！

```
git checkout dev
git status
```


工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：

```
git stash list
```

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；另一种方式是用git stash pop，恢复的同时把stash内容也删了：

```
git stash pop
```

再用git stash list查看，就看不到任何stash内容了。你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

```
git stash apply stash@{0}
```

远程分支

提交本地test分支作为远程的develop的分支：

```
git push origin develop:test
```

删除远程的test分支，但是本地还会保存的：

```
git push origin :test
```

忽略特殊文件

我们在开发过程中，有一些文件是不需要提交的，但是git总显示这部分文件会让人很不舒服，这时我们就可以通过编辑.gitignore文件来使不需要提交的文件不在提示，编写要忽略的文件，下列内容是Java开发者经常用到的：

```
# java:
*.class
# My configurations:
db.ini
deploy_key_rsa
```

注：# 此为注释，这行内容将被 Git 忽略


```
# 忽略所有 .a 结尾的文件
*.a
# 但 lib.a 除外
!lib.a
# 仅仅忽略项目根目录下的 TODO 文件，
/TODO
# 忽略 build/ 目录下的所有文件
build/
# 忽略 doc/notes.txt
doc/*.txt
```

配置别名

git命令可不可以根据自己的特点配置别名呢，当然是可以的，这样做能提高工作效率，下列是一些简单的例子。

```
git config --global alias.st status（git st=git status）
git config --global alias.co checkout（git co=git checkout）
git config --global alias.ci commit(git ci=git commit)
git config --global alias.br branch（git br=git branch）
git config --global alias.last 'log -1'(git lat= git log -l)
git config --global alias.unstage 'reset HEAD'(git unstage=git reset
HEAD)
git config --global alias.lg "log --color --graph --
```

```
pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
%C(bold blue)<%an>%Creset' --abbrev-commit"(这个大家可以亲自试试，特别炫)
```

切换用户

查看当前配置（用户）：

```
git config --list
```

修改配置：打开全局的.gitconfig文件：vi ~/.gitconfig;然后在文件中直接修改。