Git常用命令

## 一、创建版本库、提交文件

### ①git init:

初始化（或重新初始化已存在的）一个Git仓库（repository），在当前目录下生成.git目录，用于实现当前目录所有文件的版本控制。

### ②git add <name>:

添加多个文件。

eg.

**git add readme.txt:**

### ③git commit –m <msg>:

一次性提交之前add的文件。

eg.

**git commit –m “commit\_message”**

## 二、查看仓库状态

### ①git status:

查看仓库的状态。

**1）创建新文件，未添加到仓库中时，查看仓库状态：**

*# On branch master*

*# Untracked files:*

*# (use "git add <file>..." to include in what will be committed)*

*#*

*#* ***readme.txt***

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

**2）文件已添加到仓库中，并修改之后，查看仓库状态：**

*# On branch master*

*# 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:*** ***readme.txt***

*#*

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

**3）执行git add，查看仓库状态：**

*# On branch master*

*# Changes to be committed:*

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

*#*

*#* ***modified: readme.txt***

*#*

**4）执行git commit，查看仓库状态：**

*# On branch master*

*nothing to commit (working directory clean)*

### ②git diff <name>:

查看文件修改的具体内容。

eg.

**git diff readme.txt**

## 三、版本回退

### ①git log:

显示从最近到最远的提交日志。

每一次commit对应一个当时的快照，有一个commit id作为标识。

1. **git log --pretty=oneline：**

一行显示一个提交

1. **git log -p**

显示每次提交的修改内容

1. **git log --graph --oneline**

显示图形化分支拓扑

1. **git log --before="2 weeks ago" --after="2009-01-26"**

按时间段过滤提交

1. **git log --author=123**

按提交者过滤提交

1. **git log --grep=123**

按提交信息进行过滤

### ②git reset --<mode> [<commit>]:

将HEAD（指向当前版本）重置到特定的版本。

eg.

1. **git reset –hard HEAD^**

回退到上个版本。

1. **gti reset –hard HEAD^^**

回退到上上个版本。

1. **git reset –hard HEAD~100**

回退到前100个版本。

1. **git reset –hard commit\_id**

重置到commit\_id对应的版本。commit\_id可以不写全，git会自动查找。

注：

回退到旧版本时，通过git log将不能查看新版本的commit，可以通过git reflog从历史命令中找到commit\_id。

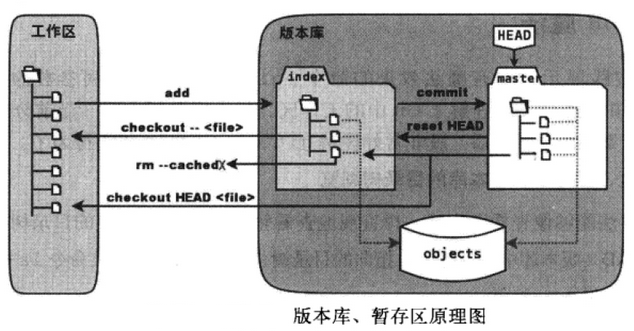
### ③git reflog:

记录每一次命令，从中可以查询git log中查询不到的commit\_id。

## 四、工作区和暂存区

工作区（Working Directory）：进行版本控制时的工作目录。

版本库（Repository）：工作区中的.git目录。



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

## 五、撤销修改

### ①git checkout -- <name>:

丢弃工作区的修改，让这个文件回到最近一次git commit或git add时的状态。

eg.

**git checkout -- readme.txt:**

### ②git reset HEAD <name>:

把暂存区的修改撤销（unstage）掉，把修改重新放回工作区。

eg.

**git reset HEAD readme.txt:**

## 六、删除文件

### git rm <name>:

从版本库中删除该文件。

eg.

**git rm test**

**1）删除工作区的文件，查看仓库状态：**

*git status*

*# On branch master*

*# Changes not staged for commit:*

*# (use "git add/rm <file>..." to update what will be committed)*

*# (use "git checkout -- <file>..." to discard changes in working directory)*

*#*

*#* ***deleted: test***

*#*

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

**若该文件是被误删的，使用git checkout -- test从版本库恢复（git checkout其实是用版本库里暂存区的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”）；**

**若确实要从版本库中删除该文件**

**2）执行git rm，查看仓库状态：**

*# On branch master*

*# Changes to be committed:*

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

*#*

*#* ***deleted: test***

*#*

**3）执行git commit，查看仓库状态：**

*# On branch master*

*nothing to commit (working directory clean)*

## 七、远程仓库

### ①git remote add <name> <url>:

将本地库关联到一个远程库。

以github为例，**git remote add origin git@server-name:path/repo-name.git:**origin即为远程仓库的name，后接远程仓库的url。path为github账户名。

eg.

**git remote add origin git@github.com:houzhiwei/learngit.git**

### ②git push -u <repository> [<refspec>...]:

把当前分支master推送到远程。<repository>可以为url或①中关联的name。

Eg.

**git push -u origin master**

若远程库是空的，关联后，第一次推送master分支的所有内容时，使用-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去掉-u参数。

### ③git clone <repository>:

从远程库克隆到本地。

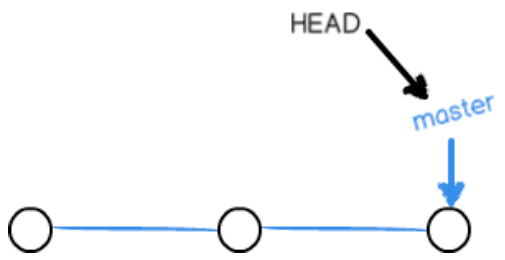
eg.

**git clone git@github.com:houzhiwei/learngit.git**

## 八、分支管理: Git鼓励大量使用分支

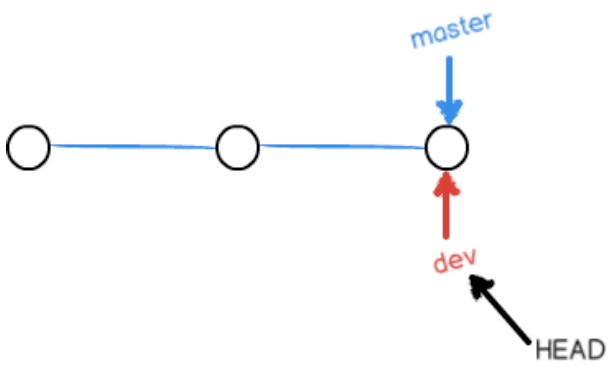
每次提交（commit），Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即 master分支。

HEAD严格来说不是指向提交，而是指向master，master才是指向提交的。一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：

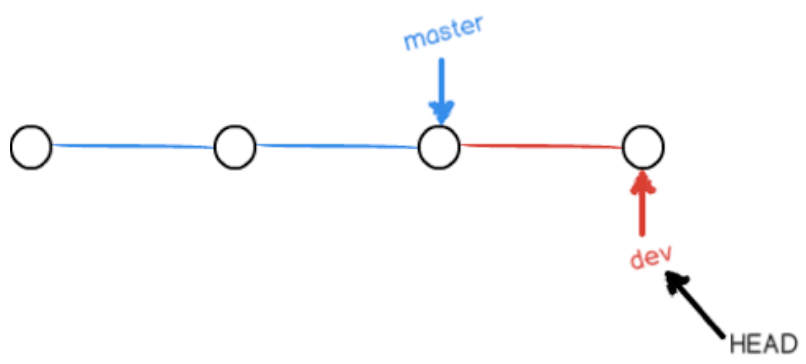


每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

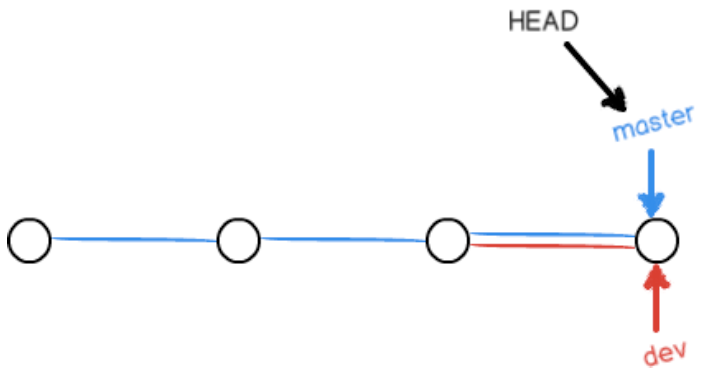
当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前在dev分支上：



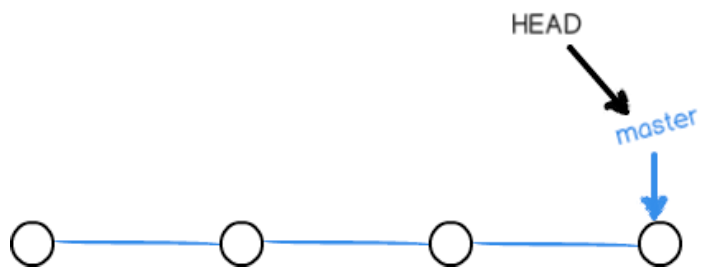
从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



在dev上的工作完成了，就可以把dev合并到master上。最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



合并完分支后，可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，就剩下了一条master分支：



### ①git checkout -b <new\_branch>:

创建新分支，并切换到该分支上。

eg.

**git checkout -b dev**

等同于：

**git branch dev**

**git checkout dev**

### ②git branch (-a):

查看当前分支。当前分支前面会标一个\*号。

eg.

**$ git branch**

*\* dev*

*master*

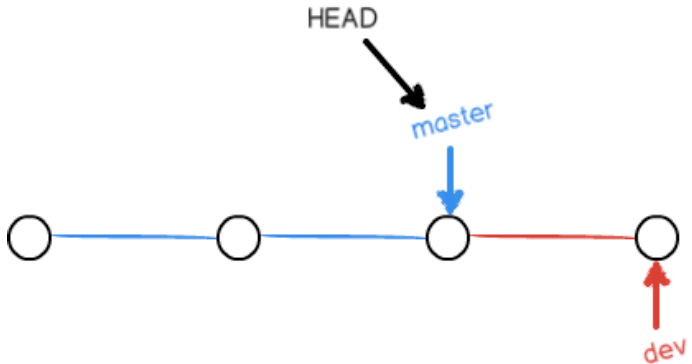
### ③git checkout <branch>:

切换分支。

eg.

**git checkout master**

当前分支情况:

****

### ④git merge <branch>:

合并指定分支到当前分支。

eg.

**$ git merge dev**

*Updating d17efd8..fec145a*

*Fast-forward*

*readme.txt | 1 +*

*1 file changed, 1 insertion(+)*

Fast-forward信息表示这次合并是“快进模式”，也就是直接把master指向dev的当前提交。

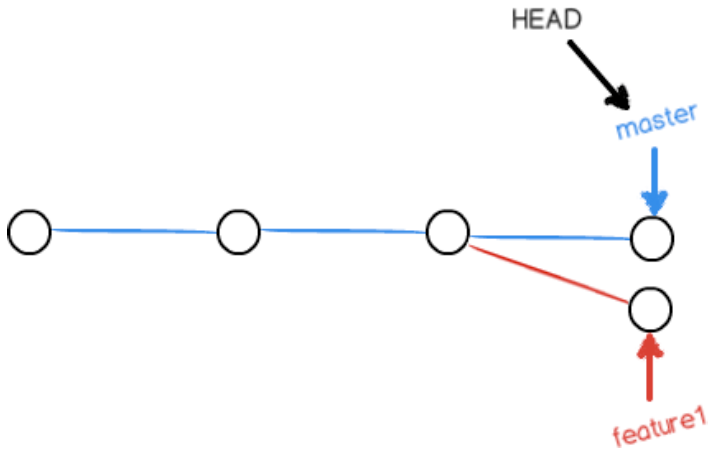
### ⑤git branch -d <branch>:

删除分支。

eg.

**git branch -d dev**

如果存在一个feature1分支，master分支和feature1分支各自都分别有新的提交：

****

这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突：

**$ git merge feature1**

*Auto-merging readme.txt*

*CONFLICT (content): Merge conflict in readme.txt*

*Automatic merge failed; fix conflicts and then commit the result.*

Git提示readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以看到冲突的文件：

**$ git status**

*# On branch master*

*# Your branch is ahead of 'origin/master' by 2 commits.*

*#*

*# Unmerged paths:*

*# (use "git add/rm <file>..." as appropriate to mark resolution)*

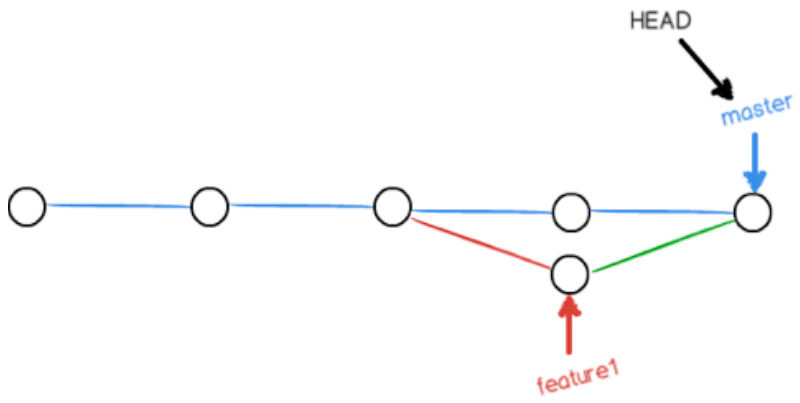
*#*

*# both modified: readme.txt*

*#*

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

Git会在该文件中用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，修改后再提交（git add、git commit）：



### ⑥git log –graph:

查看分支合并图。

eg.

**$ git log --graph --pretty=oneline --abbrev-commit**

*\* 59bc1cb conflict fixed*

*|\*

*| \* 75a857c AND simple*

*\* | 400b400 & simple*

*|/*

*\* fec145a branch test*

*...*

## 九、删除文件

### ①清除未跟踪文件:

**$ git clean -dxf**