Git是目前世界上最先进的分布式版本控制系统，Git管理的是修改，而不是文件

* **创建版本库：**
  + $ mkdir 文件名 //创建文件
  + $ cd 文件名 //进入该目录
  + $ pwd //显示当前目录
  + $ git init //把该目录变成git仓库，当前目录下多了一个.git的目录

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编写readme.txt\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

* **把文件添加到版本库**
  + $ git add readme.txt //文件添加到仓库
  + $ git commit -m “\*\*\*\*\*\*\*\*” //文件提交到仓库，\*\*\*为本次提交的说明

（commit可以一次提交很多文件，可以多次add不同的文件）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*修改readme.txt\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

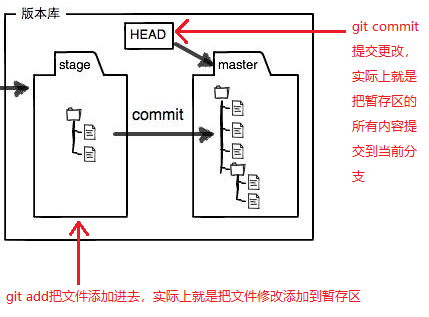
* **掌握仓库当前的状态**
  + $ git status //可以看见提示readme.txt被修改，文件是否提交到仓库
  + $ git diff readme.txt //看看具体修改了什么内容
* **版本回退**
  + $ git log //显示从最近到最远的提交日志（获取历史commit id）
  + 或者 $ git log --pretty=online //显示的提交日志一行显示
  + $ git reset --hard HEAD^ //HEAD表示当前版本，上一个版本就是HEAD^，上上一个版本就是HEAD^^，往上100个版本写成HEAD~100。此时readme.txt的内容已经回退成之前版本的内容。
  + $ cat readme.txt //查看readme.txt的内容

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*在版本回退后，命令行窗口还没有被关掉时，返回最新版本\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

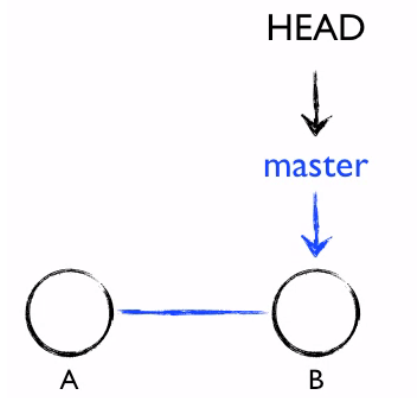
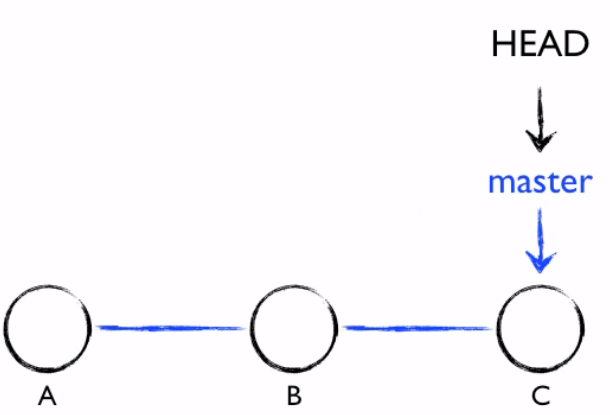
* **指定回到未来的某个版本**
  + $ git reset --hard [commit id] //版本号没必要写全，前几位就可以了
  + $ cat readme.txt //查看readme.txt的内容
* **命令行窗口已关掉状况下查看未来版本commit id**：$ git reflog

## 工作区与暂存区

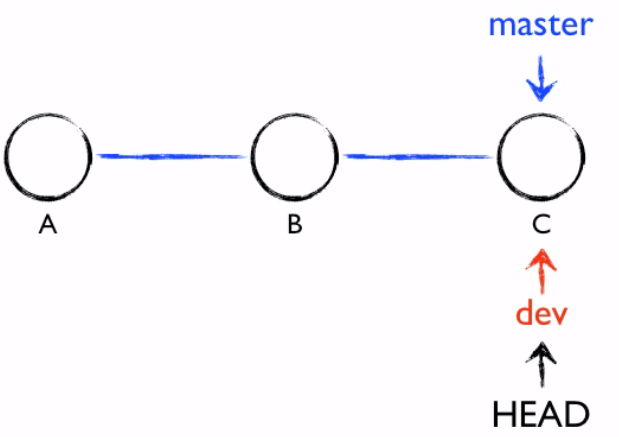
* **工作区：**电脑中的git目录
* **版本库：**工作区有一个隐藏目录.git，是Git的版本库，Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD



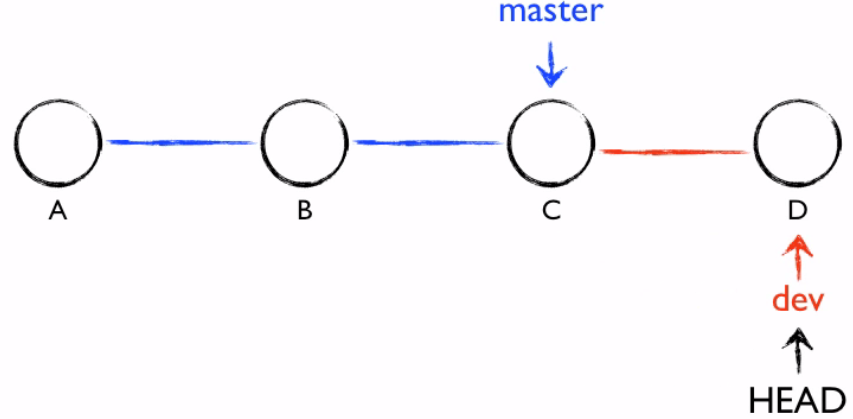
* **撤销修改**（让这个文件回到最近一次git commit或git add时的状态）：
  + 修改后还没add：$ git checkout --readme.txt
  + add后再修改：$ git checkout --readme.txt
  + 修改后已经add，还没commit：
    - $ git reset HEAD readme.txt //暂存区的修改回退到工作区
    - $ git checkout --readme.txt //丢弃工作区的修改
* **删除文件**
  + $ rm readme.txt
  + $ git status //工作区和版本库就不一致，会告诉你哪些文件被删除了
  + 确定要从版本库中删除
    - $ git rm readme.txt
    - $ git commit -m “\*\*\*\*”
  + 误删的文件恢复到最新版本：$ git checkout --readme.txt
* **本地库的所有内容推送到远程库：**$ git push -u origin master
* **推送其他分支**：$ git push origin dev
* **克隆远程库：**$ git clone [git@github.com:\*\*/\*\*.git](mailto:git@github.com:**/**.git)
* **查看远程库信息：**$ git remote 或 $ git remote -v
* **分支管理**：HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支，每次提交，master分支都会向前移动一步

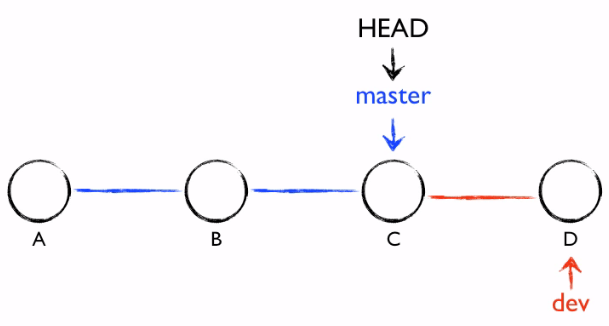
* + 创建新分支，从现在开始，对工作区的修改和提交就是针对dev分支
    - $ git checkout -b dev //-b参数表示创建并切换



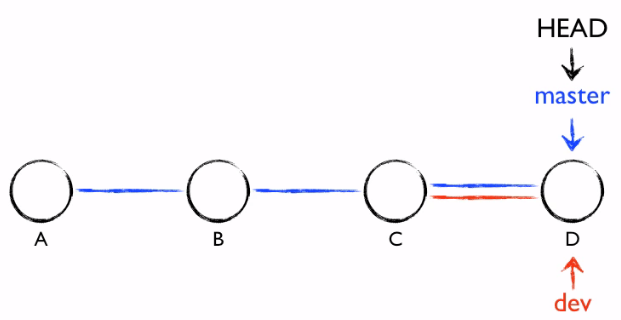
* + - $ git commit -m “D” //对文档进行修改，并提交



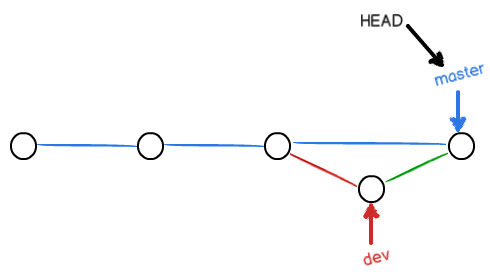
* + 合并新分支
    - $ git checkout master //切换回master分支



* + - $ git merge dev //dev分支的工作成果合并到master分支上

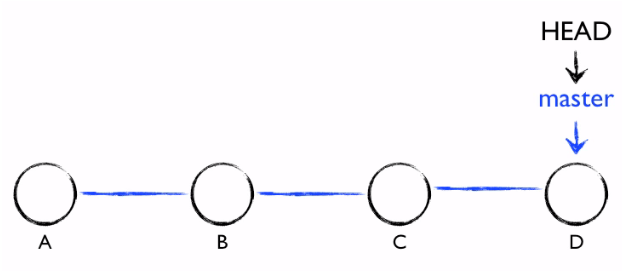


* + - 或者 $ git merge --no-ff -m “\*\*” dev //表示禁用Fast forward

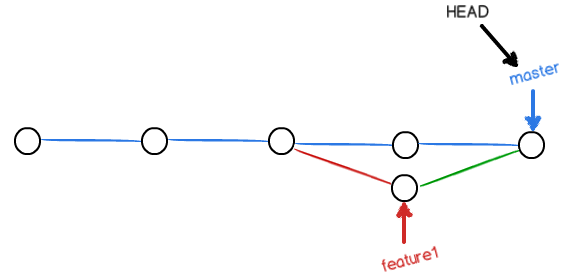


（合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并）

* + 删除新分支：
    - $ git branch -d dev



* + - 切换分支后，未合并：$ git branch -D \*\*\* //丢弃没有被合并过的分支
  + 查看当前分支：$ git branch //列出所有分支，当前分支前面会标一个\*号
  + 解决新分支和master分支冲突
    - git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容
    - $ git log --graph --pretty=online --abbrev-commit //查看分支的合并情况



### **分支策略**

在实际开发中，我们应该按照几个基本原则进行分支管理：

master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

* **Bug分支**：正在dev上进行的工作还没有提交，bug需要通过一个新的临时分支来修复：
  + $ git stash //把当前工作现场“储藏”起来，等以后恢复现场后继续工作
  + 可以创建bug分支，修改后并合并到master分支上，最后删除bug分支
  + $ git checkout dev //回到dev分支
  + $ git stash lish //查看工作现场
  + 恢复工作现场
    - $ git stash apply //恢复后，stash内容并不删除
    - 或$ git stash pop //恢复的同时把stash内容也删了
* **标签管理**
  + 创建标签：
    - $ git checkout master //切换到需要打标签的分支上
    - $ git tag \*\*\* //认标签是打在最新提交的commit上
    - 或者通过commit id创建标签：$ git tag \*\*\* [commit id]
  + 查看所有标签：$ git tag //标签不是按时间顺序列出，而是按字母排序
  + 查看标签具体信息：$ git show [标签名]
  + 创建带有说明的标签：$ git tag -a [标签名] -m “\*\*\*\*” [commit id]
  + 删除标签：
    - 未推送至远程：$ git tag -d [标签名]
    - 已推送至远程：$ git push origin:refs/tags/[标签名]
  + 推送标签至远程：$ git push origin [标签名]或一次性推送全部 $ git push origin --tags