**Github:**进行文档版本管理的软件

**（版本管理：**便于多人协同开发项目的技术**）**

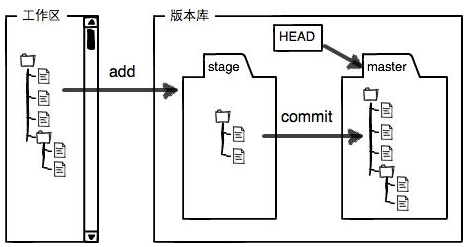
**工作区和暂存区：**

工作区：当前工作的**目录**

**版本库**：仓库，课理解为一个目录，工作区里面隐藏的**.git**目录（不算工作区），目录里的所有文件都可以被git**管理**起来，每一个文件的修改、删除，都可以被追踪到，**随时复原**。

最重要的是**stage**（又叫**index**）：**暂存区**，还有git自动创建的第一个分支**master**，以及指向**master**的一个指针**HEAD**。

\*\*\***git add + git commit** ：把文件修改添加到暂存区，然后**一次性**提交暂存区的所有分支。

****

**\*\*git** 跟踪并管理的是**修改**，而非文件。**git commit**只负责把暂存区的修改提交。每次修改，如果不使用git add到暂存区，就不会加入到**commit**中。

提交后，用**git diff HEAD -- readme.txt**命令可以查看工作区和版本库里**最新版**的区别。

补充：**git diff** #是工作区(work dict)和暂存区(stage)的**比较**，比较的是工作区文件与暂存区文件的区别（上次**git add**后的内容）的区别。

**git diff --cached** #是暂存区(stage)和分支(master)的比较，比较的是暂存区的文件与仓库分支里（上次**git commit**后的内容）的区别 。

**主要命令：**

**git init** 创建仓库，成功后会在仓库文件夹中生成**.git目录**，此目录用来跟踪管理版本库，**不可修改**。当创建git版本库的时候，git会自动创建**唯一一个**master分支

**git add <filename>** 添加文件，将文件放进**暂存区**，可反复多次使用，也可**同时添加多个文件**

**git commit -m <message>** 添加到版本库，就是把**暂存区**的所有内容提交到当前分支，message是本次提交的描述

**git status** 查看仓库状态（掌握工作区/仓库的状态：显示有**更改**的文件），有3种结果：

1. changes not staged for commit：文件**更改了**，**还未**进入暂存区，需要**add**；
2. changes to be committed：文件**已进入**暂存区，但还**未提交**到版本库，需要**commit**；
3. Untracked files：表示该文件还**从来没有**被添加进版本库，这是**新添加**的文件；

**add**和**commit** 提交和修改，需要提交的文件修改通通放到暂存区，**一次性**提交所有修改。

**版本回退：**

当文件修改到一定程度，可以选择保存一个“快照”，在git中被称为**commit**。当不小心把文件改乱了，或者误删了文件，就可以从最近的一个**commit**恢复，然后继续工作，不会把工作成果全部丢失。

1. **git log** 可以查看提交的**历史**记录（查看版本库的状态），显示从最近到最远的提交的日志；**git log --pretty=oneline** 使每个日志**单独成行**（简化版输出）
2. Git中，用**HEAD**表示**当前版本**（commit id版本号）
3. 回退到上一个版本：**git reset --hard HEAD^**，**HEAD^**表示回退1个版本，**HEAD^^**表示回退2个版本，**HEAD~100**表示回退100个版本。**HEAD**是个指向当前版本的**指针**。
4. **git reset --hard** （新版本的**commit id** ）：返回过去/未来的版本。其中“新版本的commit id”可以只写id号的前几位，Git会自动去找。
5. **git reflog** 查看**历史**每一次命令，可以查看每一次提交时的**commit id**。

**撤销修改：**

1. 当不小心改乱了工作区某个文件的内容，想直接丢掉工作区的修改时，用命令**git checkout -- <file>**，总之，就是让这个文件回到**最近一次git commit**或**git add**时的状态。

2. 当不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令**git reset HEAD <file>**，把暂存区的修改撤销掉（**unstage**），**重新放回工作区** ，回到了1。第二步按1操作，丢弃工作区的修改即可。

注：**git reset**命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用**HEAD**时，表示最新的版本。

3. 已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退那一节，可以回退到上一个版本，不过**前提**是你还没有把自己的本地版本库推送到远程，否则就不行了。

补充：

Unix/Linux 命令中，**-** 后一般跟**短命令选项**（通常是单字母，如**-m**），**--** 后一般跟**长命令选项**。如果只有一个单独的**--**，后面不紧跟任何选项，则表示命令选项**结束**，后续的都作为命令的参数而不是选项。

例如：

**git checkout -- filename**（**git checkout**其实是用版本库里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”。）

**filename**作为**git checkout** 的参数，而不是选项。

**删除文件：**

手动删除或者用命令**rm <fileName>**删除文件后，执行**git** **add/rm <fileName>**和**git commit-m <message>**就可以了。

另一种情况是删错了，因为版本库里还有存档，所以可以很轻松地把误删的文件恢复到**最新**版本**git checkout -- file** 。

**git checkout**其实是用版本库里的版本**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”。

**远程仓库：**

**Git**是分布式版本控制系统，同一个Git仓库，可以分布到**不同的机器**上。 每个人都从“服务器”仓库上**克隆**一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

1.创建SSH Key：

ssh-keygen -t rsa -C "mail@example.com" # 一路回车即可，无需设置密码

此时可以在用户**主目录**里找到.ssh目录，里面有**id\_rsa**和**id\_rsa.pub**两个文件，这两个就是SSH Key的**秘钥对**，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是**公钥**，可以放心地告诉任何人。

2.打开GitHub，在个人主页找到setting，找到**SSH and GPG keys**，然后，点“New SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容，完后确定添即可。 注：GitHub允许你**添加多个**Key。

**添加远程库：**

让本地和远程仓库**同步**。在GitHub上新建仓库，此时既可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

1.在本地的**learnGit**仓库下运行如下命令（先有本地库时，将本地库和远程库关联），此时远程库的名字就是**origin**。

git remote add origin git@github.com:ft-sunshine/learngit.git # 需改

2.把本地库的所有内容推送到远程库上 ，实际上是把当前分支**master**推送到远程。 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。 （去掉**-u**）

git push -u origin master # 第一次推送

git push origin master # 本地提交后，如有必要，可推送本地的最新修改。

**从远程库克隆：**

git clone git@github.com:ft-sunhine/gitskills.git

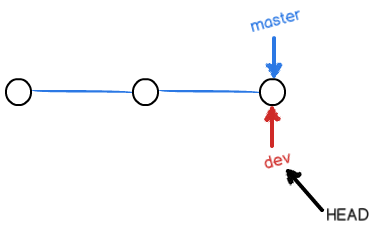
如果有多个人协作开发，那么每个人各自从远程库**克隆**一份就可以了。

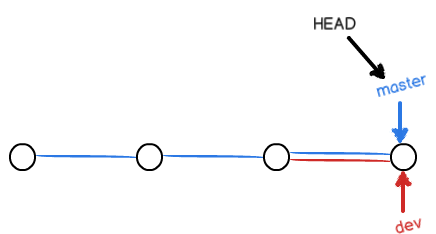
**分支管理：**

如果创建了一个属于自己的分支，别人看不到，还继续在原来的分支上正常工作，而在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既**安全**，又不影响别人工作。

**分支与管理：**

每次提交，Git都把它们串成一条时间线，这条**时间线**就是一个分支。严格来说**HEAD**并不是指向提交，而是指向**master**（**HEAD**指向**当前**分支），**master**（当前分支）才是指向提交的（指向**当前分支**的提交点）。所以，**HEAD**指向的就是当前分支，其可以确定当前的分支。





Git鼓励大量使用分支：

  1. 查看分支：**git branch**

  2. 创建分支（相当于增加了一个**dev**指针）：**git branch <name>** ，切换分支：**git checkout <name>**

  3. 创建+切换分支：**git checkout -b <name>**

  4. 合并指定分支到当前分支：**git merge <name>**

  5. 删除分支：**git branch -d <name>**

例子如下：

创建分支**dev**：**git checkout -b dev**

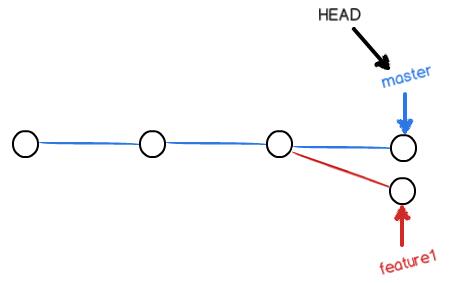
在**dev**分支上进行一通操作

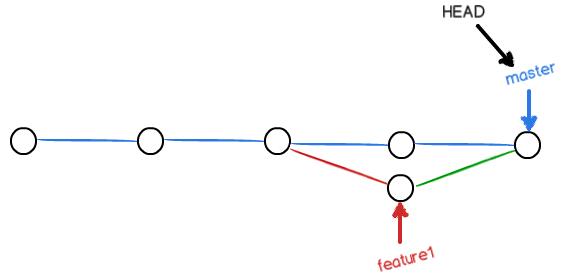
合并分支 ，先**git checkout master**切换到主分支，然后执行**git merge dev**进行合并。然后可以再执行**git branch -d dev**删除分支

**解决冲突：**

合并分支时就可能出现**代码冲突**，任何合并冲突都必须去解决。

其实冲突的本质是：不同的分支**修改**了代码**相同**的部分，从而导致合并分支时出现冲突（虽然冲突，但还是会合并，会**自动标注**出冲突的地方）。所以解决冲突就需要把Git合并失败的文件进行手动修复，改为想要的内容再提交（**add + commit**）即可。





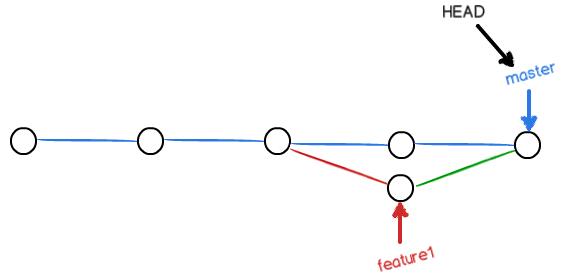
可以用带参数的git log查看分支的合并情况（分支合并图）： **git log --graph --pretty=oneline --abbrev-commit**

**分支管理策略：**

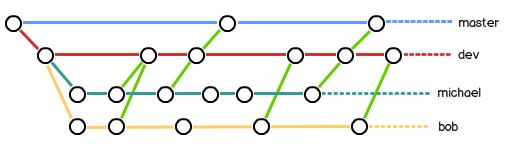
合并分支时，默认采取的为Fast farword模式，这种合并看不到合并历史，删除分支会丢掉分支信息。

  禁用Fast farword模式时，合并分支时会产生**一个新的commit**，这样，从分支历史上就可以看出分支信息。 用**git merge --no-ff -m "merge with no-ff" dev** 命令**合并**，其中，**--no-ff**参数，表示禁用Fast forward模式。 因为本次合并要创建一个新的**commit**，所以加上**-m**参数，把**commit**描述写进去。

不使用Fast forward模式，**merge**后就像这样 ：



分支策略（分支**管理原则**）图如下：



**强制删除：**

开发一个新feature，最好**新建**一个分支； 如果要丢弃一个没有被**合并过**的分支（已提交），可以通过**git branch -D <name>强行删除**。

**多人协作：**

当从远程仓库克隆时，实际上Git自动把本地的**master**分支（或**指定**的某分支）和远程的**master**分支（或指定的某分支）对应起来了，并且，远程仓库的默认名称是**origin**，用**git remote**或git **remote -v**可查看**远程库**的信息。

**推送分支：**

一般来说，master主分支和dev开发分支，都要时刻与远程库**保持同步**，所以需要推送。其他分支，比如bug分支和feature分支等，是否需要推动到远程可依具体情况而定。

# 推送时，要**指定**本地分支，这样，Git就会把该分支（的所有提交）推送到远程库对应的远程分支上

git push origin master

git push origin dev

**多人协作的工作模式：**

多人协作的工作模式通常是这样：

1.首先，可以试图用**git push origin <branch-name>**推送自己的修改；

2.如果推送失败，则因为远程分支比你的本地更新，需要先用**git pull**试图**合并**；

3.如果合并有冲突，则**解决冲突**，并在本地提交；

没有冲突或者解决掉冲突后，再用**git push origin <branch-name>**推送就能成功

4.如果**git pull**提示**no tracking information**，则说明本地分支和远程分支的链接关系**没有创建**，用命令**git branch --set-upstream-to <branch-name> origin/<branch-name>**。

**小结:**

* 查看远程库信息，使用**git remote -v**；
* 本地新建的分支如果不推送到远程，对其他人就是**不可见的**；
* 从本地推送分支，使用**git push origin branch-name**，如果推送失败，先用**git pull**抓取远程的**新提交**；
* 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**，本地和远程分支的名称最好**一致**；
* 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**；
* 从远程抓取分支，使用**git pull**，如果有冲突，要先处理冲突

**标签管理:**

Git的标签是版本库的**快照**，它就是指向某个commit的指针（跟分支很像，但是分支可以**移动**，标签不能移动）。

1.创建标签

* 首先，**切换**到需要打标签的分支上：
* 命令**git tag <tagname> [commit id]**用于新建一个标签，默认标签是打在**最新**提交的**commit**上的，也可以指定一个**commit id**（找到历史提交的**commit id：git log --pretty=oneline --abbrev-commit）**；
* 命令**git tag -a <tagname> -m "describe" [commit id]**可以**指定标签信息**；
* 命令**git tag**可以**查看所有**标签。
* 可以用**git show <tagname>**查看标签信息，

**\*\*\***标签总是和某个**commit**挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上**都可以**看到这个标签。

2.操作标签

默认创建的标签都**只存储**在本地，不会自动推送到远程，所以，打错的标签可以在本地**安全删除**。

* 命令**git push origin <tagname>**可以推送一个**本地标签**到远程库；
* 命令**git push origin --tags**可以推送**全部**未推送过的本地标签到远程库；
* 命令**git tag -d <tagname>**可以**删除**一个本地标签；
* 命令**git push origin :refs/tags/<tagname>**可以删除一个远程标签（先从本地删除）。