**工作区和暂存区：**

**工作区（Working Directory）:**

就是存放git源码的目录。

**版本库（Repository）:**

工作区有一个隐藏目录.git，这是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自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



往Git版本库里加文件，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到**暂存区**；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支，也就是说commit针对的是暂存区的内容。

**放弃未提交的文件修改：**

场景1：改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令

git checkout -- <file> //注意，--和<file>之间必须有空格。

场景2：当你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>，就回到了场景1，第二步按场景1操作。

**git删除文件：**

git rm可以删除一个文件，并从git仓库中移除。注意：要执行commit后才真正提交。

git rm <file> //删文件

git rm -r <folder> //删文件夹

git rm --cached <file> //删除已经提交到暂存区的文件，但不会对文件本身做改动

git rm的误删的文件被提交之前可以用checkout恢复。

**git分支管理：**

git branch <new branch> //创建新分支

git checkout <branch name> //切换分支

git checkout -b <new branch> //创建新分支并切换过去

git checkout -b <new branch> --track <remote>/<branch> //创建新分支并切换过去，并指定对应的远端分支

git branch -d <branch name> //删除分支

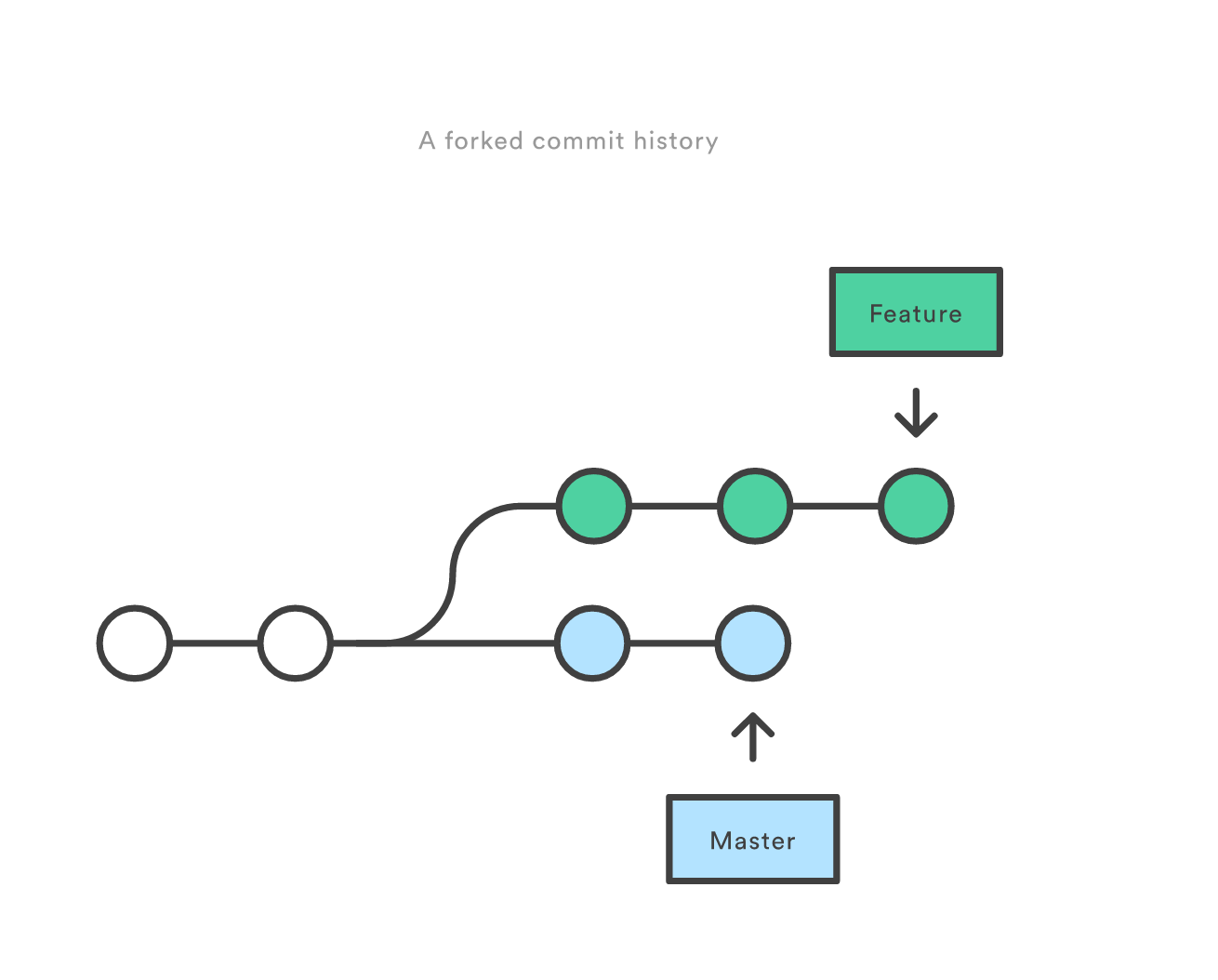
git branch //查看本地分支

git branch -a //查看本地和远端分支

git branch -r //查看远端分支

**git rebase 和 git merge 的区别：**

在一个feature分支进行新特性的开发，与此同时，master 分支的也有新的提交。



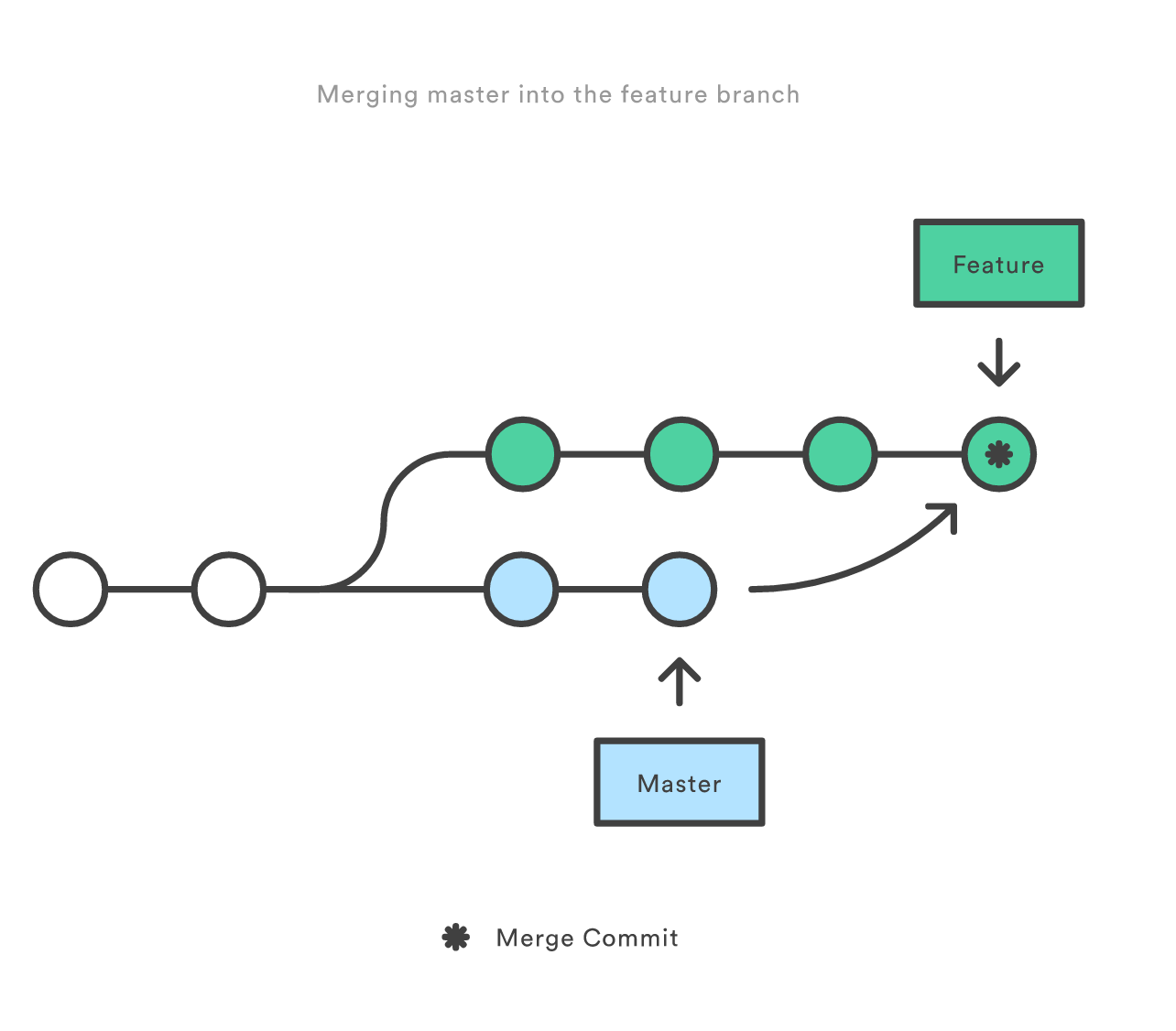
为了将master 上新的提交合并到你的feature分支上，有两种选择：merging or rebasing：

merge:

git checkout feature

git merge master

此时在feature上git 自动会产生一个新的commit(merge commit):



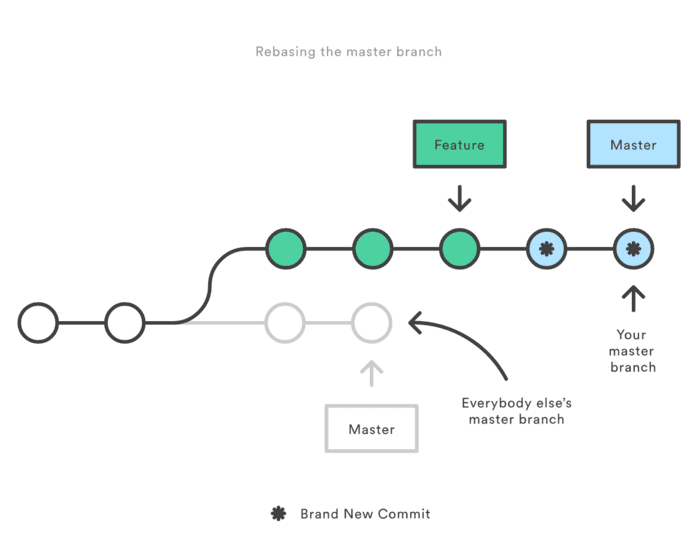
merge 特点：自动创建一个新的commit  
如果合并的时候遇到冲突，仅需要修改后重新commit  
优点：记录了真实的commit情况，包括每个分支的详情  
缺点：因为每次merge会自动产生一个merge commit，所以在使用一些git 的GUI tools，特别是commit比较频繁时，看到分支很杂乱。

rebase:

rebase本质是变基。

git checkout feature

git rebase master



rebase 将所有master的commit移动到你的feature 的顶端。

合并时如果出现冲突需要按照如下步骤解决：

1. 修改冲突部分
2. git add 不要在git add 之后习惯性的执行 git commit命令
3. git rebase --continue

如果第三步无效可以执行 git rebase --skip

**git add详解：**

-u, --update update tracked files. 已经被tracked的文件的所有修改加入暂存区。

-A, --all add changes from all tracked and untracked files

--ignore-removal 添加新文件和被修改的文件，忽略被删除的文件。

**为本地的分支设置远端分支：**

git branch --set-upstream <local branch> <remote branch>

git branch --set-upstream-to <remote branch>

创建本地分支的同时指定对应的远端分支：

git checkout -b <local branch> <remote branch>

第一次向远端推送时指定对应的远端branch：

git push --set-upstream origin <remote branch>

**修改上一次提交的信息：**

git commit –-amend

git commit –-amend --no-edit 表示不改变提交信息，不会弹出编辑器

**撤销git commit –-amend:**

如果只 amend 了一次, 那么直接用 git reset HEAD@{1} 就可以撤销这次 amend. 如果 amend 多次, 就参考 git reflog 进行撤销:

首先使用 git reflog 命令查看操作记录，如：

625ba04 (HEAD -> empty\_photo\_info) HEAD@{0}: commit (amend): 从leaf获取的photo info为空，需要记录到grafana

52b8e81 HEAD@{1}: commit (amend): 从leaf获取的photo info为空，需要记录到grafana

用git reset HEAD@{你要的}即可。

**比较两个分支：**

git diff <branch a> <branch b>

git diff <branch a> <branch b> --stat // 显示有差异的文件列表

或：

git diff <branch a>..<branch b>

比较两个分支之间某个文件的不同：

git diff <branch a>..<branch b> 文件名

git diff <branch a>..<branch b>

git log branch1 ^branch2 // 查看branch1上有，而branch2上没有的提交

git log branch1..branch2 // 查看branch2比branch1多了哪些提交

git log branch1...branch2 // 查看branch1和branch2差异

git log -lefg-right branch1...branch2 // 查看差异的提交分别在哪个分支

**git fetch与git pull的区别：**

**FETCH\_HEAD**： 是一个版本链接，记录在本地的一个文件.git/FETCH\_HEAD中，指向着目前已经从远程仓库取下来的分支的最新commit。

**git fetch有四种基本用法:**

1. git fetch 这将更新git remote中所有的远程repo 所包含的所有分支的最新commit-id, 将其记录到.git/FETCH\_HEAD文件中

2. git fetch remote\_repo 这将更新名称为remote\_repo 的远程repo上的所有branch的最新commit-id，将其记录。

3. git fetch remote\_repo remote\_branch 这将这将更新名称为remote\_repo 的远程repo上的分支： remote\_branch

4. git fetch remote\_repo remote\_branch:local\_branch 这将这将更新名称为remote\_repo 的远程repo上的分支： remote\_branch，并在本地创建local\_branch本地分支保存远端分支的所有数据。

**用fetch合并将远端master分支的改动合并到本地dev分支的流程：**

git fetch origin master:temp //在本地新建一个temp分支，并将远程origin仓库的master分支代码下载到本地temp分支

git checkout dev

git merge temp

git branch –d temp

git pull**的工作流程:**

首先，基于本地的FETCH\_HEAD记录，比对本地的FETCH\_HEAD记录与远程仓库的版本号，然后git fetch 获得当前指向的远程分支的后续版本的数据，然后再利用git merge将其与本地的当前分支合并。所以可以认为git pull是git fetch和git merge两个步骤的结合。

小结：

与git pull相比git fetch相当于是从远程获取最新版本到本地，但不会自动merge。如果需要有选择的合并git fetch是更好的选择。否则git pull将更为快捷。

**git删除文件：**

git rm <文件名>

git rm –-cached <文件名> // 只是取消对文件的跟踪，文件不会被删除

**git下载远端分支：**

拉取远程分支并创建本地分支，自动切换到这条本地分支：

git checkout -b local\_branch origin/remote\_branch

拉取远程分支并创建本地分支，不会自动切换到这条本地分支：

git fetch origin remote\_branch:local\_branch

**git删除远端分支：**

git push origin --delete <branch>

git branch --delete --remotes <remote>/<branch>

git branch -dr <remote>/<branch> # Shorter

**git创建远端分支：**

git push origin local\_branch:remote\_branch

git push --set-upstream origin local\_branch // 用本地分支创建远端分支同时设置关联

**git查看本地分支对应的远端分支：**

切换到本地分支，执行git branch –vv

**git删除远端分支：**

git push origin -d remote\_branch

git push origin : remote\_branch // 推空分支到远端来删除远端分支

**git获取最后一次提交的id：**

git rev-parse HEAD

**git打标签：**

git可以给仓库历史中的某一个提交打上标签，以示重要。通常会使用这个功能来标记发布结点，如： v1.0 、 v2.0等。

git tag: 列出已有的标签;

-l或—list: 按照特定的模式查找标签，如：

$ git tag -l deploy\_v2020\*

deploy\_v2020-01-02\_12-58-27

deploy\_v2020-01-06\_10-26-25

deploy\_v2020-01-06\_18-04-08

deploy\_v2020-01-08\_09-47-48

deploy\_v2020-01-15\_11-18-56

deploy\_v2020-01-17\_11-34-02

deploy\_v2020-02-12\_10-37-17

git支持两种标签：轻量标签（lightweight）与附注标签（annotated）。

轻量标签相当于某个特定提交的引用。

附注标签是存储在Git 数据库中的一个完整对象，它们是可以被校验的，其中包含打标签者的名字、电子邮件地址、日期时间，此外还有一个标签信息，并且可以使用 GNU Privacy Guard （GPG）签名并验证。通常会建议创建附注标签，这样你可以拥有以上所有信息。但是如果你只是想用一个临时的标签， 或者因为某些原因不想要保存这些信息，那么也可以用轻量标签。

**创建附注标签：**

指定 -a 选项即可：git tag -a v1.4 -m "my version 1.4"。-m 选项指定了一条将会存储在标签中的信息。如果没有指定信息，Git会启动编辑器要求输入信息。

通过使用 git show 命令可以看到标签信息和与之对应的提交信息,如：

$git show deploy\_v2020-01-02\_12-58-27

tag deploy\_v2020-01-02\_12-58-27

Tagger: shanyanmei <shanyanmei@kuaishou.com>

Date: Thu Jan 2 12:58:28 2020 +0800

deploy gen tag

commit e9bcb98a792e336abd666a3d7b7a28e076865a1d

Author: kouyan <kouyan@kuaishou.com>

Date: Thu Jan 2 12:45:27 2020 +0800

输出显示了打标签者的信息、打标签的日期时间、附注信息，然后显示具体的提交信息。

**创建轻量标签：**

创建轻量标签，不需要使用 -a、-s 或 -m 选项，只需要提供标签名字：

$ git tag v1.4-lw

如果在标签上运行 git show，你不会看到额外的标签信息。 命令只会显示出提交信息：

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

**对指定的提交打标签：**

git tag -a {tag\_name} {commit id}

**推送标签：**

默认情况下，git push命令并不会推送到远端。在创建完标签后必须显式地推送标签到远端：

git push origin <tagname>

要一次性推送很多标签，可使用带有--tags选项的git push命令，会把所有不在远端的标签推上去：

git push --tags

**删除标签:**

git tag -d <tagname>

**删除远程标签:**

$ git push origin --delete <tagname>

**检出标签：**

git checkout <tagname>

会使仓库处于“分离头指针（detacthed HEAD）”的状态,不在任何一个分支上。

**远端分支强行覆盖本地：**

git reset --hard origin/{branch name}

**git查看被删除文件日志：**

git log --{file name}

**git获取当前分支**

git symbolic-ref --short HEAD

git rev-parse --abbrev-ref HEAD

git describe --contains --all HEAD

**git更新远程分支列表**

git remote update origin --prune

git remote update origin -p

**git log输出短commit id**

git log --abbrev-commit

**git commit id转短commit id**

git rev-parse --short 5d63088d11748feebc90cfae36af3bb8b7bd0b3a

git rev-parse --short HEAD

**git 短commit id转commit id**

git rev-parse 5d63088d1

**git清理untrack file**

# 删除 untracked files

git clean -f

# 连 untracked 的目录也一起删掉

git clean -fd

# 连 gitignore 的untrack 文件/目录也一起删掉

#（慎用，一般这个是用来删掉编译出来的 .o之类的文件用的）

git clean -xfd

# 在用上述 git clean 前，建议加上 -n 参数来先看看会删掉哪些文件，防止重要文件被误删

git clean -nxfd

git clean -nf

git clean -nfd

**git 生成及合入patch**

# 生成patch

# 生成两个commit之间的patch(包含两个commit)

git format-patch <commit\_id\_1>..<commit\_id\_2>

# 单个commit

git format-patch < commit\_id >

# 检查patch文件

git apply --stat <patch\_file>

# 检查该PATCH是否能合入  
git apply --check <patch\_file> #如果没有任何输出，则表示可合入。如有错误会在终端输出

# 打patch

git apply <patch\_file>