说明

Git 常用、不常用、实用的命令，这些命令都是在日常使用过程中遇到过整理下来的，留作备忘，如果对这些命令还不明白什么意思，请参考学习：Git 新手入门 。

在日常开发中，由于采用 Git 作为版本控制，经常跟命令行打交道，记录整理下使用到的一些命令，方便回顾。

Concept

```
master : default development branch
origin : default upstream repository
HEAD : current branch and commit
HEAD^ : parent of HEAD
HEAD~4 : the great-great grandparent of HEAD
```

Alias

下面的只是例子，想改成什么跟随自己的意愿即可。

```
git config --global alias.st status //status 缩写成 st
git config --global alias.co checkout //checkout 缩写成 co
git config --global alias.br branch //branch 缩写成 br
git config --global alias.ci commit //commit 缩写成 ci
git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset
```

如果不想使用了，删除掉的话，直接删除 conf 配置文件中的行即可，global 的在当前用户下 `vim ~/.gitconfig` 删除 `alias` 下你配置的内容接口；若是当前仓库则在 `.git/config` 中。

Git Config

Git 配置文件分为三级，系统级(--system)、用户级(--global)和目录级(--local)，三者的使用优先级以离目录(repository)最近为原则，如果三者的配置不一样，则生效优先级目录级>用户级>系统级，可以通过 `git config --help` 查看更多内容。

+ 系统级配置存储在 `/etc/gitconfig` 文件中，可以使用 `git config --system user.name "jim"` ，`git config --sytem user.email "jim.jim@gmail.com"` 来进行配置，该配置对系统上所有用户及他们所拥有的仓库都生效的配置值。

+ 用户级存储在每个用户的 `~/.gitconfig` 中，可以使用 `git config --global user.name "jim"` ，`git config --global user.email "jim.jim@gmail.com"` 来进行配置，该配置对当前用户上所有的仓库有效。

+ 目录级存储在每个仓库下的 `.git/config` 中，可以使用 `git config --local user.name "jim"` ，`git config --local user.email "jim.jim@gmail.com"` 来进行配置，只对当前仓库生效。

Basic Usage

- 添加文件到暂存区（staged）：`git add filename` / `git stage filename`
- 将所有修改文件添加到暂存区（staged）：`git add --all` / `git add -A`
- 提交修改到暂存区（staged）：`git commit -m 'commit message'` / `git commit -a -m 'commit message'` 注意理解 -a 参数的意义
- 从Git仓库中删除文件：`git rm filename`
- 从Git仓库中删除文件，但本地文件保留：`git rm --cached filename`
- 重命名某个文件：`git mv filename newfilename` 或者直接修改完毕文件名，进行 `git add -A && git commit -m 'commit message'` Git会自动识别是重命名了文件
- 获取远程最新代码到本地：`git pull (origin branchname)` 可以指定分支名，也可以忽略。pull 命令自动 fetch 远程代码并且 merge，如果有冲突，会显示在状态栏，需要手动处理。更推荐使用：`git fetch` 之后 `git merge --no-ff origin branchname` 拉取最新的代码到本地仓库，并手动 merge

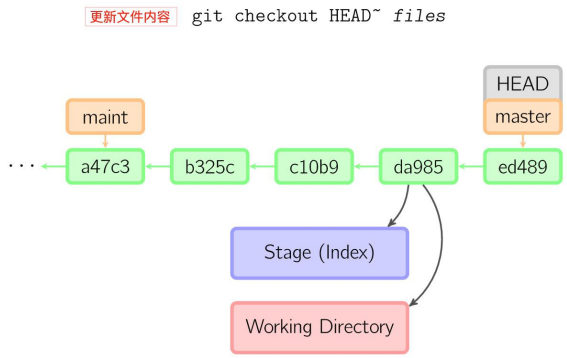
。

Repository

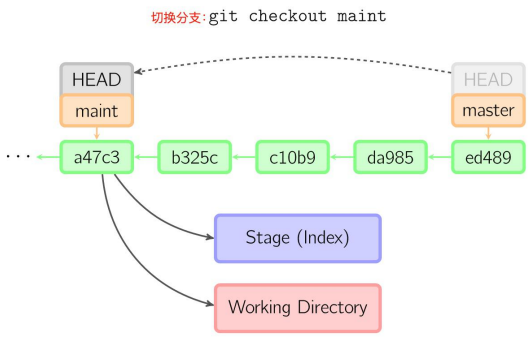
- 检出（clone）仓库代码： `git clone repository-url` / `git clone repository-url local-directoryname`
    - 例如，clone jquery 仓库到本地： `git clone git://github.com/jquery/jquery.git`
    - clone jquery 仓库到本地，并且重命名为 my-jquery ： `git clone git://github.com/jquery/jquery.git my-jquery`
- 查看远程仓库： `git remote -v`
- 添加远程仓库： `git remote add [name] [repository-url]`
- 删除远程仓库： `git remote rm [name]`
- 修改远程仓库地址： `git remote set-url origin new-repository-url`
- 拉取远程仓库： `git pull [remoteName] [localBranchName]`
- 推送远程仓库： `git push [remoteName] [localBranchName]` 例： `git push -u orgin master` 将当前分支推送到远端master分支
- 将本地 test 分支提交到远程 master 分支： `git push origin test:master` （把本地的某个分支 test 提交到远程仓库，并作为远程仓库的 master 分支）提交本地 test 分支作为远程的 test 分支 ： `git push origin test:test`

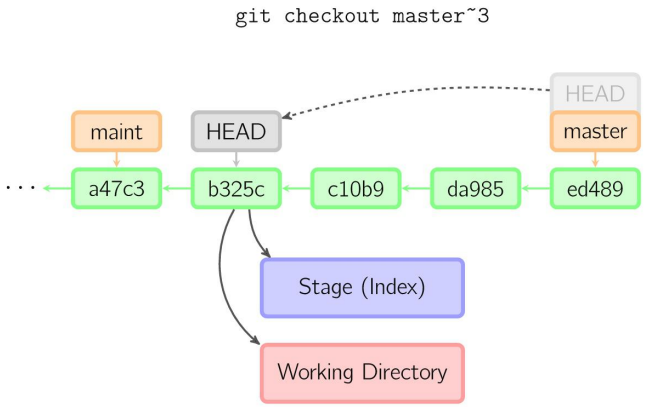
Checkout

checkout命令用于从历史提交（或者暂存区域）中拷贝文件到工作目录，也可用于切换分支。



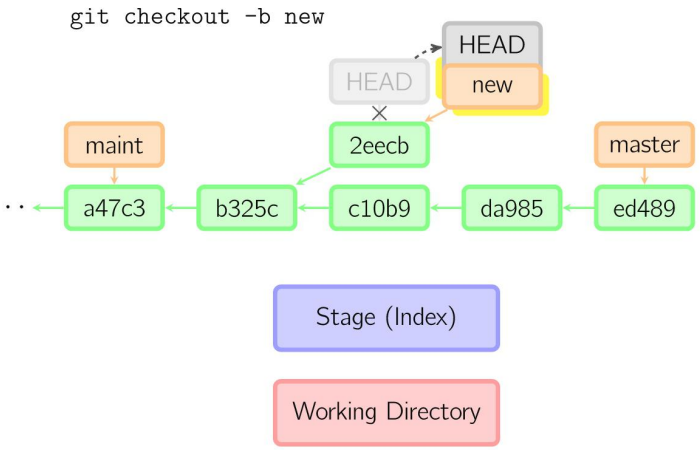
匿名分支：如果既没有指定文件名，也没有指定分支名，而是一个标签、远程分支、SHA-1值或者是像 master~3 类似的东西，就得到一个匿名分支，称作 detached HEAD（被分离的 HEAD 标识）。





当HEAD处于分离状态（不依附于任一分支）时，提交操作可以正常进行，但是不会更新任何已命名的分支。（你可以认为这是在更新一个匿名分支。）一旦此后你切换到别的分支，比如说 `master`，那么这个提交节点（可能）再也不会被引用到，然后就会被丢弃掉了。注意这个命令之后就不会有东西引用 `2eecb`。详细查看：visual-git-guide#detached

但是，如果你想保存这个状态，可以用命令 `git checkout -b name` 来创建一个新的分支。



Log

> Description : Shows the commit logs.
>
> The command takes options applicable to the git rev-list command to control what is shown and how, and options applicable to the git diff-* commands to control how the changes each commit introduces are shown.
>
> git log [options] [revision range] [path]

常用命令整理如下：

- 查看日志： `git log`
- 查看日志，并查看每次的修改内容： `git log -p`
- 查看日志，并查看每次文件的简单修改状态： `git log --stat`
- 一行显示日志： `git log --pretty=oneline` / `git log --pretty='format:"%h - %an, %ar : %s'`
- 查看日志范围：
    - 查看最近10条日志： `git log -10`
    - 查看2周前： `git log --until=2week` 或者指定2周的明确日期，比如： `git log --until=2015-08-12`
    - 查看最近2周内： `git log --since=2week` 或者指定2周明确日志，比如： `git log --since=2015-08-12`
    - 只查看某个用户的提交： `git log --committer=user.name` / `git log --author=user.name`
    - 只查看提交msg中包含某个信息的历史，比如包含'测试'两个字的： `git log --grep '测试'`
    - 试试这个 ： `git log --graph --pretty=format:'%Cred%h%Creset - %C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit` 感觉好用就加成 alias ，方便日后用，方法： `git config --global alias.aliasname 'alias-content'`
    - 更多用法：Viewing the History -- 《Pro Git2》

`log` 的目的就是为了查看改动点来排查问题，除了 `git log` 还可以使用 `git show` 、 `git blame` 来查看文件的改动。

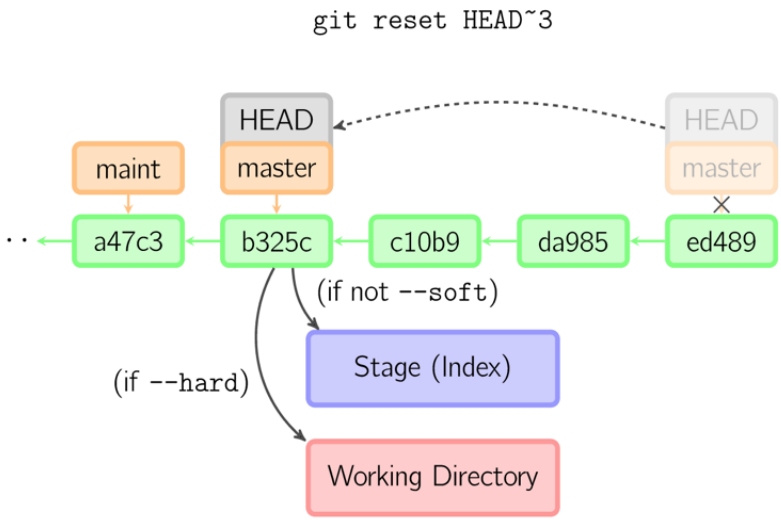
- Who changed what and when in a file : `git blame $file`
- 查看一次 commit 中修改了哪些文件: `git show --pretty="" --name-only <sha1-of-commit>` 或者 `git diff-tree --no-commit-id --name-only -r <sha1-of-commit>`

Undo things

- 上次提交 msg 错误/有未提交的文件应该同上一次一起提交，需要重新提交备注： `git commit --amend -m 'new msg'`
- 一次 `git add -A` 后，需要将某个文件撤回到工作区，即：某个文件不应该在本次commit中： `git reset HEAD filename`
- 撤销某些文件的修改内容： `git checkout -- filename` 注意：一旦执行，所有的改动都没有了，谨慎！谨慎！谨慎！
- 将工作区内容回退到远端的某个版本： `git reset --hard <sha1-of-commit>`

Reset

reset命令把当前分支指向另一个位置，并且有选择的变动工作目录和索引，也用来在从历史仓库中复制文件到索引，而不动工作目录。



将工作区内容回退到远端的某个版本：`git reset --hard <sha1-of-commit>`

- `git reset --hard HEAD^` reset index and working directory ，以来所有的变更全部丢弃，并将 HEAD 指向
- `git reset --soft HEAD^` nothing changed to index and working directory ,仅仅将 HEAD 指向，所有变更显示在 “changed to be committed” 中
- `git reset --mixed HEAD^` default,reset index ,nothing to working directory 默认选项，工作区代码不改动，添加变更到index区

Revert

> `git revert` will create a new commit that's the opposite (or inverse) of the given SHA. If the old commit is "matter", the new commit is "anti-matter"——anything removed in the old commit will be added in the new commit and anything added in the old commit will be removed in the new commit. This is Git's safest, most basic "undo" scenario, because it doesn't alter history—so you can now git push the new "inverse" commit to undo your mistaken commit.

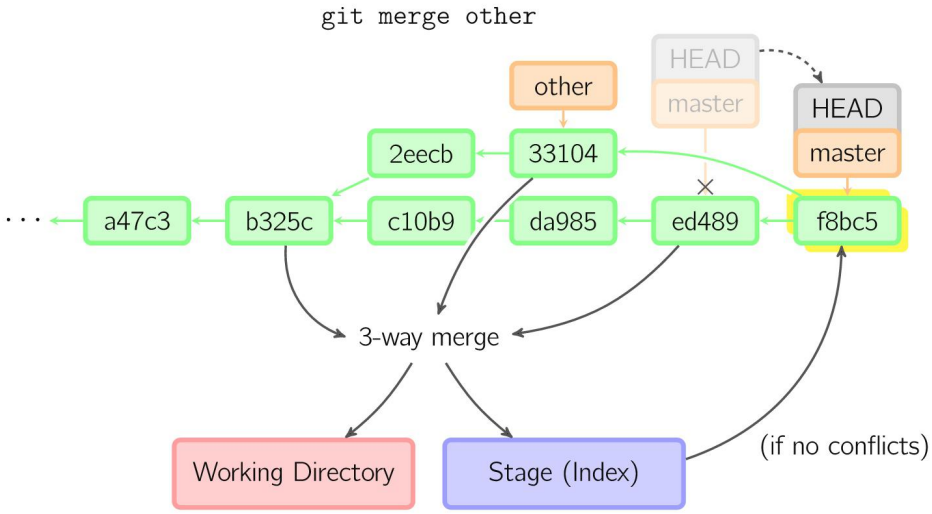
```
git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>…
git revert --continue
git revert --quit
git revert --abort
```

Revert VS Reset

Diff

- 查看工作区（working directory）和暂存区（staged）之间差异：`git diff`
- 查看工作区（working directory）与当前仓库版本（repository）HEAD版本差异：`git diff HEAD`
- 查看暂存区（staged）与当前仓库版本（repository）差异：`git diff --cached` / `git diff --staged`
- 不查看具体改动，只查看改动了哪些类：`git diff --stat`

Merge



- 解决冲突后/获取远程最新代码后合并代码：`git merge branchname` ，将 branchname 分支上面的代码合并到当前分支
- 保留该存在版本合并log：`git merge --no-ff branchname` 参数 `--no-ff` 防止 fast-forward 的提交，详情参考：the difference，fast-forward：分支内容一致，指针直接移动，并未能看出分支信息

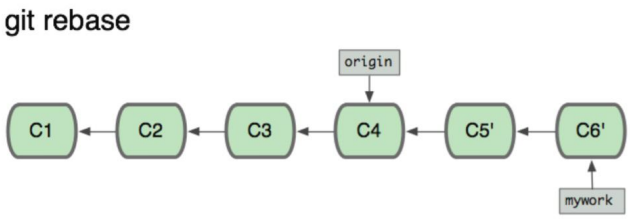
Rebase

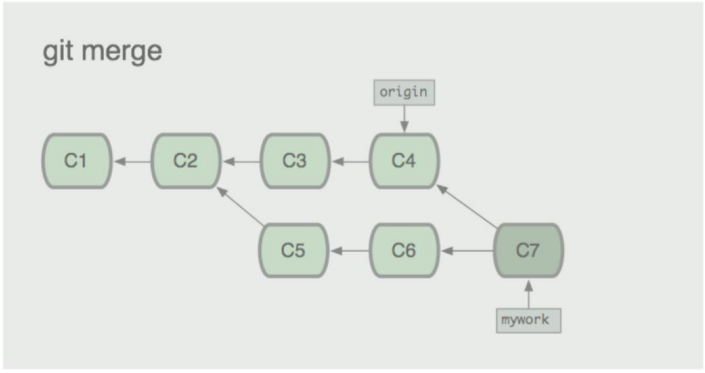
Rebase 同 Merge 的结果是一样的，就是合并本地、远程的改动，但过程中还有区别。

```
git checkout mywork
git rebase origin
```

这些命令会把你的"mywork"分支里的每个提交(commit)取消掉,并且把它们临时 保存为补丁(patch)(这些补丁 放到".git/rebase"目录中),然后把"mywork"分支更新 到最新的"origin"分支,最后把保存的这些补丁应用 到"mywork"分支上。

一张图分清 rebase 和 merge 的区别

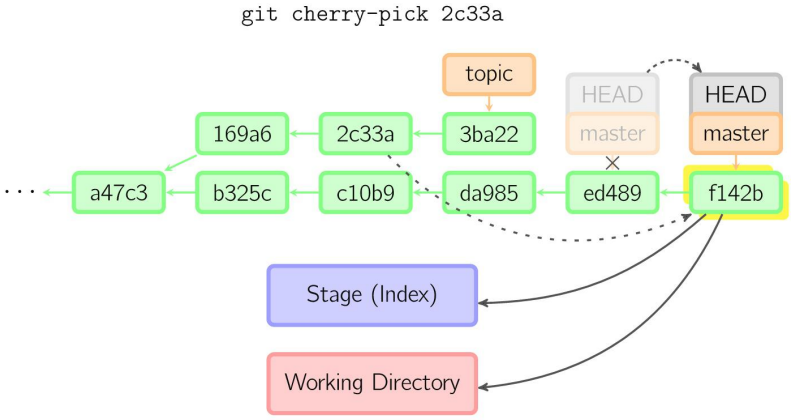




在rebase的过程中,也许会出现冲突(conflict). 在这种情况,Git会停止rebase并会让你去解决冲突;在解决完冲突后,用 `git-add` 命令去更新这些内容的索引(index), 然后,你无需执行 git-commit,只要执行: `git rebase --continue` 这样git会继续应用(apply)余下的补丁。在任何时候,你可以用 `--abort` 参数来终止rebase的行动,并且"mywork" 分支会回到rebase开始前的状态。 `git rebase --abort`

Cherry Pick

cherry-pick 命令"复制"一个提交节点并在当前分支做一次完全一样的新提交。



Branch workflow

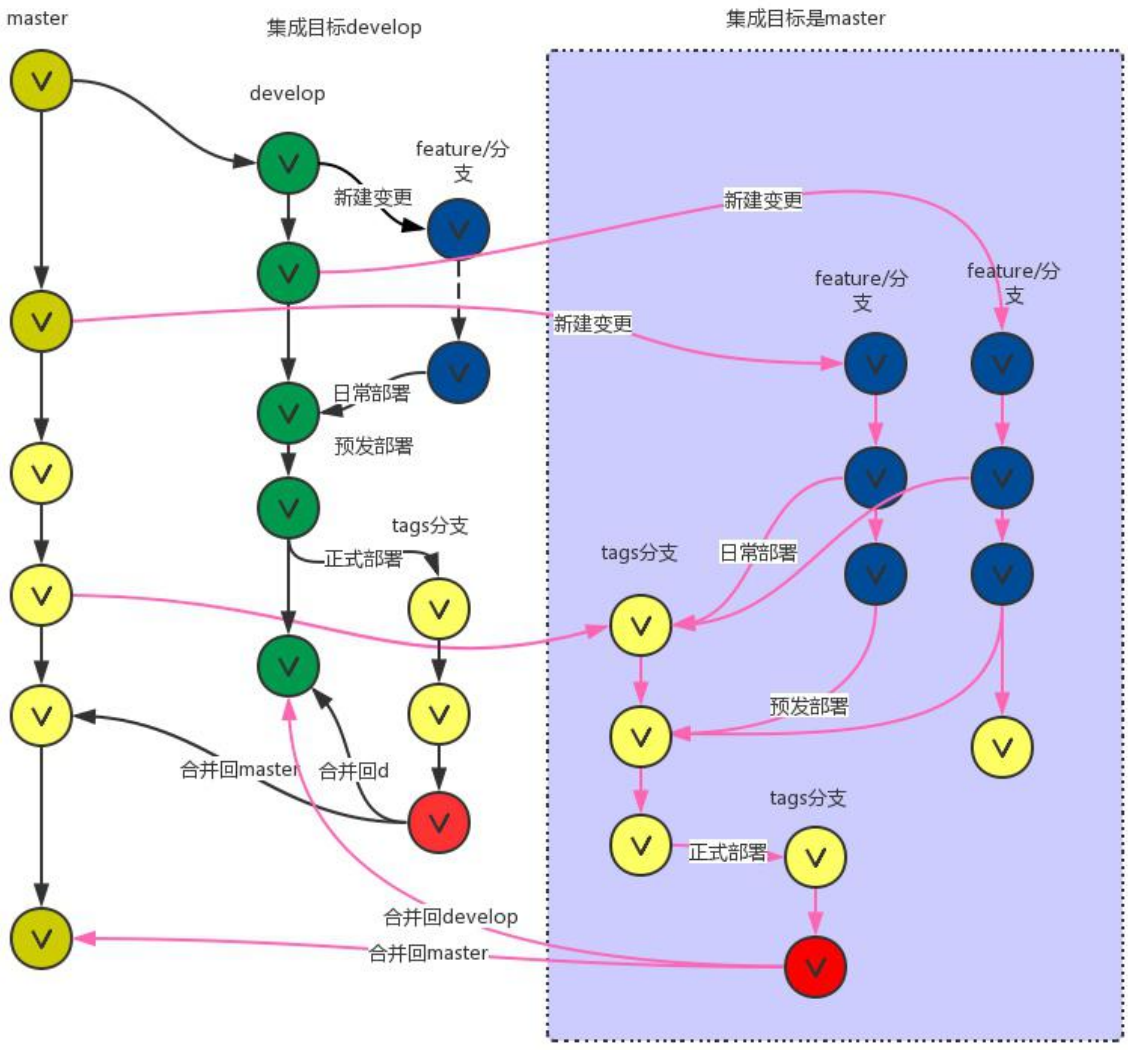
Aone2 Git 分支开发部署模型详细解读 http://docs.alibaba-inc.com:8090/pages/viewpage.action?pageId=194872297

分支情况 origin

- master
- develop

- release
  - 20161129163217010_r_release_yingyongming
  - 20161029163217010_r_release_yingyongming
- feature
  - 20161129_163448_newfeature_1
  - 20161129_163448_newfeature_2
- hotfix
  - 20161129_163448_hotfix_1
- tags
  - 20161129163217010_r_release_newfeature_yingyongming

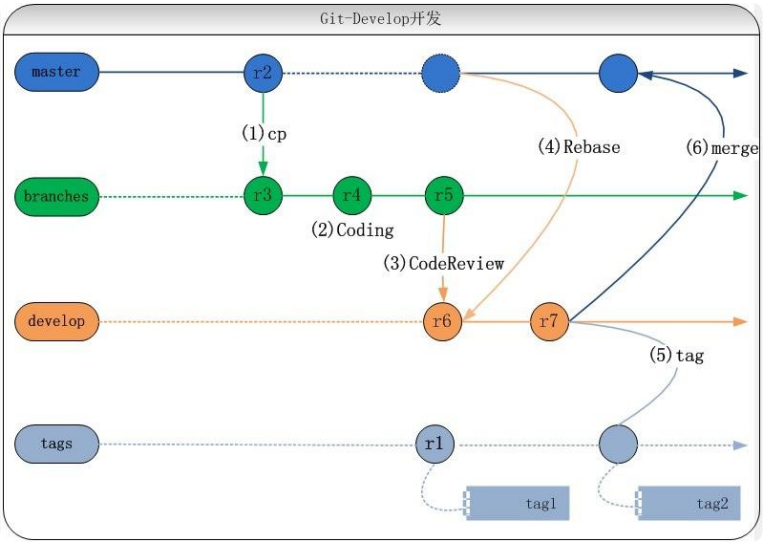
创建分支的时候直接操作： `git checkout -b feature/20161129_163448_newfeature_1`



- master：master永远是线上代码，最稳定的分支，存放的是随时可供在生产环境中部署的代码，当开发活动告

一段落，产生了一份新的可供部署的代码时，发布成功之后，代码才会由 aone2 提交到 master，master 分支上的代码会被更新。应用上 aone2 后禁掉所有人的 master的写权限

- develop：保存当前最新开发成果的分支。通常这个分支上的代码也是可进行每日夜间发布的代码，只对开发负责人开放develop权限。
- feature：功能特性分支，每个功能特性一个 feature/ 分支，开发完成自测通过后合并入 develop 分支。可以从 master 或者develop 中拉出来。
- hotfix：紧急bug分支修复分支。修复上线后，可以直接合并入master。



Git-Develop 分支模式是基于 Git 代码库设计的一种需要严格控制发布质量和发布节奏的开发模式。develop 作为固定的持续集成和发布分支，并且分支上的代码必须经过 CodeReview 后才可以提交到 Develop 分支。它的基本流程如下：

- 每一个需求/变更都单独从Master上创建一条Branch分支；
- 用户在这个Branch分支上进行Codeing活动；
- 代码达到发布准入条件后aone上提交Codereview，Codereview通过后代码自动合并到Develop分支；
- 待所有计划发布的变更分支代码都合并到Develop后，系统再 rebase master 代码到Develop 分支，然后自行构建，打包，部署等动作。
- 应用发布成功后Aone会基于Develop分支的发布版本打一个“当前线上版本Tag”基线；
- 应用发布成功后Aone会自动把Develop分支的发布版本合并回master；

Branch 命令

- 查看分支：`git branch` 、`git branch -v` 、`git branch -vv` (查看当前分支 tracking 哪个远端分支)、`git branch --merged` 、`git branch --no-merged`
- 创建分支：`git branch branchname`
  - 例： 基于 master 分支新建 dev 分支 ： `git branch dev`
- 基于之前的某个 Commit 新开分支： `git branch branchname <sha1-of-commit>`
  - 例： 基于上线的的提交 a207a38d634cc10441636bc4359cd8a18c502dea 创建 hotfix 分支 ： `git branch hotfix a207a38`

    - 例：基于 remoteBranch、localBranch、commitId、tag 创建分支均可以 `git checkout -b newbranch localBranch/remoteBranch/commitId/tag`
    - 例：创建一个空的分支 `bash git checkout --orphan gh-pages # 创建一个orphan的分支，这个分支是独立的 Switched to a new branch \'gh-pages\' git rm -rf . # 删除原来代码树下的所有文件 rm \'.gitignore\' #注意这个时候你用git branch命令是看不见当前分支的名字的，除非你进行了第一次commit。添加新的文件，并且 commit 掉，就能看到分支了。 '`
- 切换分支： `git checkout branchname`
    - 例： 由分支 master 切换到 dev 分支： `git checkout dev`
- 创建新分支并切换到下面： `git checkout -b branchname` 或者 `git branch branchname && git checkout branchname`
    - 例：基于 master 分支新建 dev 分支，并切换到 dev 分支上： `git checkout -b dev` 或 `git branch dev && git checkout dev`
- 查看分支代码不同： `git diff branchname` 比较 branchname 分支与当前分支的差异点
- 合并分支： `git merge branchname` 将 branchname 分支代码合并到当前分支
- 删除分支： `git branch -d branchname` 强制删除未合并过的分支： `git branch -D branchname`
- 重命名分支: `git branch -m dev development` 将分支 dev 重命名为 development
- 查看远程分支： `git branch -r` 或 `git branch -r -v`
- 获取远程分支到本地： `git checkout -b local-branchname origin/remote-branchname`
- 推送本地分支到远程： `git push origin remote-branchname` 或 `git push origin local-branchname:remote-branchname`
    - 将本地 dev 代码推送到远程 dev 分支： `git push (-u) origin dev` 或 `git push origin dev:dev`
    - （技巧）将本地 dev 分支代码推送到远程 master 分支： `git push origin dev:master`
- 删除远程分支： `git push origin :remote-branchname` 或 `git push origin --delete remote-branchname`
- 手动跟踪分支，master分支追踪origin/next分支: `git branch --set-upstream master origin/next` 或者 `git branch --set-upstream-to=origin/master` 看 git 的版本是否支持。
- TrackingBranch，可以通过 `git branch -vv` 来查看当前 track 的分支情况。新建立分支时会自动 track 相应远程分支， `git checkout -b sf origin/serverfix` (Branch sf set up to track remote branch serverfix from origin. Switched to a new branch 'sf'). 也可以手动 track: `git branch -u origin/serverfix` (Branch serverfix set up to track remote branch serverfix from origin). 等同于命令 `git checkout --track origin/serverfix`

> “Checking out a local branch from a remote branch automatically creates what is called a “tracking branch” (or sometimes an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and branch to merge into.
> When you clone a repository, it generally automatically creates a master branch that tracks origin/master. However, you can set up other tracking branches if you wish – ones that track branches on other remotes, or don’t track the master branch. The simple case is the example you just saw, running git checkout -b [branch] [remotename]/[branch]. This is a common enough operation that git provides the --track shorthand:”

Tag

- 查看 tag：`git tag`
- 查找指定 tag，比如查找 V1.0.* ：`git tag -l 'V1.0.*'` 会列出匹配到的，比如 V1.0.1,V1.0.1.1,V1.0.2 等
- 创建轻量级 tag（lightweight tags）：`git tag tag-name` ，例如：`git tag v1.0`
- 创建 tag（annotated tags）：`git tag -a tag-name -v 'msg'` ，例如：`git tag -a v1.0.0 -m '1.0.0版本上线完毕打tag'`
    - annotated tags VS lightweight tags 可以通过命令真实查看下：`git show v1.0` / `git show v1.0.0`
    - “A lightweight tag is very much like a branch that doesn’t change – it’s just a pointer to a specific commit. Annotated tags, however, are stored as full objects in the Git database. They’re checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).”
- 查看指定 tag 信息：`git show tag-name`
- 基于历史某次提交（commit）创建 tag ：`git tag -a tagname <sha1-of-commit>`
    - 例：基于上线时的提交 a207a38d634cc10441636bc4359cd8a18c502dea 创建tag：`git tag -a v1.0.0 a207a38`
- 删除 tag ：`git tag -d tagname`
- 拉取远程 tag 到本地：`git pull remotename --tags` 例如：`git pull origin --tags`
- 推送 tag 到远程服务器：`git push remotename tagname` 例如：`git push origin v1.0.0`
- 将本地所有 tag 推送到远程：`git push remotename --tags` 例如：`git push origin --tags`
- 删除远程 tag ：`git push origin :tagname` 或者 `git push origin --delete tagname`

Submodule

添加子模块：`$ git submodule add [url] [path]`

如：`$ git submodule add git://github.com/soberh/ui-libs.git src/main/webapp/ui-libs`

初始化子模块：$ git submodule init ----只在首次检出仓库时运行一次就行

更新子模块：$ git submodule update ----每次更新或切换分支后都需要运行一下

删除子模块：（分4步走哦）

1）$ git rm --cached [path]

2）编辑“.gitmodules”文件，将子模块的相关配置节点删除掉

3）编辑“ .git/config”文件，将子模块的相关配置节点删除掉

4）手动删除子模块残留的目录

Stash

经常有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是 `git stash` 命令。

`stash` 可以获取你工作目录的中间状态，也就是你修改过的被追踪的文件和暂存的变更，并将它保存到一个未完结变更的堆栈中，随时可以重新应用。

```
usage: git stash list [<options>] 查看当前 stash 的列表
   or: git stash show [<stash>] 查看某一个版本的详细内容
   or: git stash drop [-q|--quiet] [<stash>] 删除 stash 中内容
   or: git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>] 将 stash 中的代码应用到工作区中
   or: git stash branch <branchname> [<stash>]
   or: git stash [save [--patch] [-k|--[no-]keep-index] [-q|--quiet]
                [-u|--include-untracked] [-a|--all] [<message>]]
   or: git stash clear  清空 stash 中所有内容
```

oh-my-zsh 常用命令

```
alias g='git'
alias ga='git add'
alias gco='git checkout'
alias gcb='git checkout -b'
alias gcm='git checkout master'
alias gcd='git checkout develop'
alias gd='git diff'
alias gf='git fetch'
alias gfo='git fetch origin'
alias gl='git pull'
alias gp='git push'
```