Git命令使用与分享V3

1. Git简介
2. Git是什么

|  |
| --- |
| 是一个开源的[分布式](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F/19276232?fromModule=lemma_inlink" \t "_blank)版本控制系统，可以有效、高速地处理从很小到非常大的项目[版本管理](https://baike.baidu.com/item/%E7%89%88%E6%9C%AC%E7%AE%A1%E7%90%86/2511538?fromModule=lemma_inlink" \t "_blank)。 |

1. 集中式版本控制系统 VS分布式版本控制系统
2. 集中式版本控制系统 代表 SVN

|  |
| --- |
| central-repo |

集中式版本控制系统最大的毛病就是必须联网才能工作

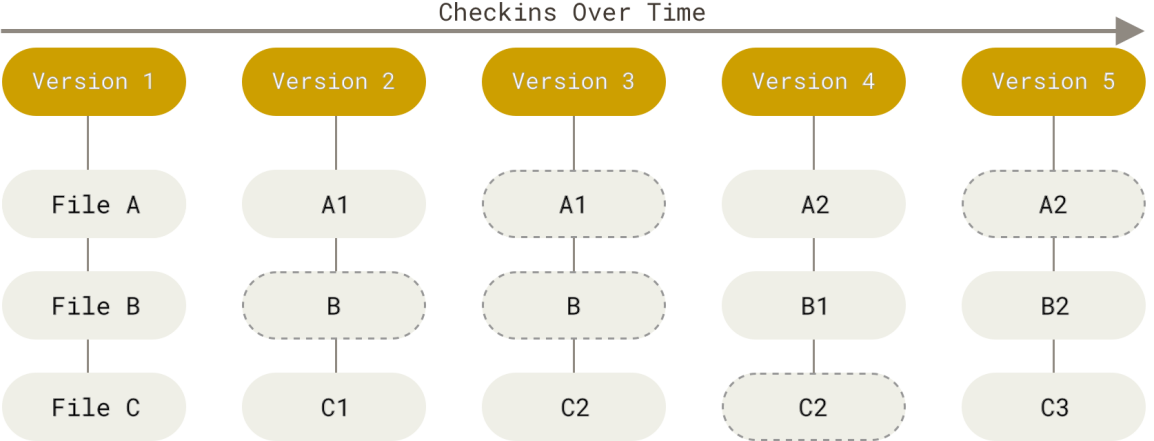
1. 分布式版本控制系统

|  |
| --- |
| distributed-repo |

每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。

1. Git 特点

直接记录快照，而非差异比较。



|  |
| --- |
|  |

1. Git 优缺点

|  |
| --- |
| * 优点：   适合[分布式开发](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E5%BC%80%E5%8F%91/4143301?fromModule=lemma_inlink" \t "_blank)，强调个体。  公共服务器压力和数据量都不会太大。  速度快、灵活。  任意两个开发者之间可以很容易的解决冲突。  离线工作。   * 缺点：   学习周期相对而言比较长。  不符合[常规思维](https://baike.baidu.com/item/%E5%B8%B8%E8%A7%84%E6%80%9D%E7%BB%B4/9532113?fromModule=lemma_inlink" \t "_blank)。  代码[保密性](https://baike.baidu.com/item/%E4%BF%9D%E5%AF%86%E6%80%A7/4928247?fromModule=lemma_inlink" \t "_blank)差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。 |

1. Git 基础
2. 四个工作区域

|  |
| --- |
| https://img2018.cnblogs.com/blog/1090617/201810/1090617-20181008211557402-232838726.png |

* Workspace： 工作区，就是你平时存放项目代码的地方
* Index / Stage： 暂存区，用于临时存放你的改动，事实上它只是一个文件，保存即将提交到文件列表信息
* Repository： 仓库区（或版本库），就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中HEAD指向最新放入仓库的版本
* Remote： 远程仓库，托管代码的服务器，可以简单的认为是你项目组中的一台电脑用于远程数据交换

1. 文件四种状态

|  |
| --- |
| https://img2018.cnblogs.com/blog/1090617/201810/1090617-20181008212040668-1339848607.png |

* Untracked: 未跟踪, 此文件在文件夹中, 但并没有加入到git库, 不参与版本控制. 通过git add 状态变为Staged。
* Unmodify: 文件已经入库, 未修改, 即版本库中的文件快照内容与文件夹中完全一致. 这种类型的文件有两种去处, 如果它被修改, 而变为Modified. 如果使用git rm --cached移出版本库, 则成为Untracked文件。
* Modified: 文件已修改, 仅仅是修改, 并没有进行其他的操作. 这个文件也有两个去处, 通过git add可进入暂存staged状态, 使用git checkout 则丢弃修改过, 返回到unmodify状态, 这个git checkout即从库中取出文件, 覆盖当前修改。
* Staged: 暂存状态. 执行git commit则将修改同步到库中, 这时库中的文件和本地文件又变为一致, 文件为Unmodify状态. 执行git reset HEAD filename取消暂存,文件状态为Modified。

1. 文件四种状态变迁

|  |
| --- |
| https://img2018.cnblogs.com/blog/1090617/201810/1090617-20181008212245877-52530897.png |

新建文件--->Untracked

使用add命令将新建的文件加入到暂存区--->Staged

使用commit命令将暂存区的文件提交到本地仓库--->Unmodified

如果对Unmodified状态的文件进行修改---> modified

如果对Unmodified状态的文件进行remove操作--->Untracked

1. 检查当前文件状态

git status

|  |
| --- |
|  |
|  |
|  |

git status 不光是查看当前文件状态，更重要的是提示下一次如何操作。如果是git 初学者，不知道下一次做什么的时候直接使用git status 查看下。

1. 查看提交历史

git log

|  |
| --- |
|  |

git log --oneline / git log --pretty=oneline

|  |
| --- |
|  |
| 非常简洁输出commit log |

git log --pretty=fuller / git log --pretty=full

|  |
| --- |
|  |
| 将作者和提交人全部打印出来了 |

git log --author="Ketor" –oneline

|  |
| --- |
|  |
| 指定某个人提交的commit记录。 |

git log --name-status

|  |
| --- |
|  |
| 查看提交记录 文件状态。 |

1. 查看文件差异

git diff

|  |
| --- |
|  |
| git diff : 是查看working tree与index file的差别的。 |

git diff --cached/staged

|  |
| --- |
|  |
| git diff –cached/staged :是查看index file与commit的差别的。 |

git diff HEAD

|  |
| --- |
|  |
| Git diff HEAD : 是查看working tree 和commit的差别的 。（你一定没有忘记，HEAD代表的是最近的一次commit的信息）。 |

1. 文件撤销

文件未做commit操作。

git restore/git checkout --

|  |
| --- |
| 文件仅仅是做了修改，未做其他。将文件撤销到上一次提交状态。  注意修改的文件内容被抹掉，无法找回。 |

git restore --staged

|  |
| --- |
|  |
| 将文件从暂存区恢复到了工作区，文件内容不变。 |

更为复杂的撤销 git reset 留到后面章节再讲。

1. 重用之前的commit 记录

git commit --amend

|  |
| --- |
|  |
| 针对于忘记提交文件或者记录需要再次修正下。 |

1. 远程仓库的使用
2. 克隆远程仓库

|  |
| --- |
| git clone [git@github.com:yuhaijun999/dingo-store.git](mailto:git@github.com:yuhaijun999/dingo-store.git)  带分支克隆  git clone -b wip-KvPutIfAbsent [git@github.com:yuhaijun999/dingo-store.git](mailto:git@github.com:yuhaijun999/dingo-store.git) |

1. 查看远程仓库

|  |
| --- |
|  |

1. 添加远程仓库

|  |
| --- |
| git remote add zetyun git@github.com:dingodb/dingo-store.git |

1. 从远程仓库中抓取与拉取

|  |
| --- |
| git fetch origin |

1. 推送到远程仓库

|  |
| --- |
| git push origin main |

1. git 分支

由于git 采用快照方式做版本控制，切换到任何commit速度快。git 分支本质上是切换到某一个commit.git 鼓励我们频繁的切换分支。

1. HEAD

说简单一点，HEAD就是当前活跃分支的游标。

1. 分支创建

|  |
| --- |
| git branch testing |
| 当前HEAD指针仍然指向master  git checkout testing这样 HEAD 就指向 testing 分支了 |
| 实际上我们会将上面两条指令合并成一条指令。  git checkout -b testing |
| 根据远程分支创建本地分支  git checkout -b fix-scan origin/fix-scan  或者  git checkout --track origin/fix-scan  或者  git checkout origin/fix-scan  本地分支与远程分支建立映射关系  git push  git push --set-upstream origin fix-scan |

1. 分支删除

|  |
| --- |
| git branch -d testing  如果当前分支上有未提交的内容  git branch -D testing  删除远程分支  git push origin --delete testing |

1. 分支合并

git 鼓励频繁的创建分支，最终分支也需要合并。另外git 合并仅会自动合并确定内容。不确定的内容留给用户自己合并。一直以来我一直卡在这里。

1. git merge

|  |
| --- |
| git merge不对仓库历史做任何改变，它会保留分支上所有的历史commit，然后简单地创建一个合并的commit。  在这里插入图片描述 |
| **使用**git merge**将**master**分支合并到**feature**分支。**  git checkout feature  git merge master  上面的操作会在feature分支上产生一个新的commit，它是c4和c7的合并为c8，c8包含了master分支上的所有修改，结果如下图所示：  在这里插入图片描述 |
| 产生的是三方合并C2, C4, C7 结果为C8 |

例子：

|  |
| --- |
|  |
|  |
| feature 分支上修改 |
|  |
|  |
|  |
|  |
|  |
|  |
| git merge master |
|  |
|  |
|  |
|  |
| HEAD -> feature  master -> master |
| ours -> feature  theirs -> master |
|  |
|  |
| git commit |
|  |
|  |

还原为merge前状态

|  |
| --- |
|  |
|  |

1. git rebase

|  |
| --- |
| 在这里插入图片描述 |
| **使用**git rebase**将**master**分支合并到**feature**分支**  git checkout feature  git rebase master  rebase将feature分支的基移到master分支的开头(即c4)，将所有的新提交合并到主分支中。与在feature分支中创建合并提交不同，rebase通过在feature分支中为每个commit创建全新的commit来重写提交历史，\*c5、\*c6、\*c7是在feature分支上的新commit。换句话说，就是在c4的基础上，依次重新提交在feature分支上的每一次修改。  在这里插入图片描述 |
| 可以看到提交是线性的，我们可以[回溯](https://so.csdn.net/so/search?q=%E5%9B%9E%E6%BA%AF&spm=1001.2101.3001.7020" \t "_blank)从feature分支顶端开始到master分支的第一次提交的任何提交。rebase不会创建不必要的合并提交，因此，项目的提交历史将更清晰，更容易被其他开发人员理解。   * 注意：不建议在公共分支(master分支)上执行rebase合并，这会创建不同的master分支。当然你也可以同步这两个不同的master分支，然后merge为一个，但是这会导致重复提交，从而让主分支的commit十分混乱。 |

例子：

|  |
| --- |
|  |
|  |
| feature 分支上修改 |
|  |
|  |
|  |
|  |
|  |
|  |
| git rebase master |
| git rebase --skip # 丢弃当前分支的修改 慎用 |
|  |
|  |
|  |
| HEAD -> master  feature -> feature |
| ours -> master  theirs -> feature |
|  |
|  |
|  |
|  |
| git rebase --continue |
|  |
| 保存即可。 |
|  |
|  |
|  |
|  |
|  |
| 保存即可。 |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| 变基： C2->C4 （master） |

还原为 rebase前状态

|  |
| --- |
|  |
|  |

1. git merge squash

|  |
| --- |
| 在git merge中，每一次merge就会产生一个额外的commit，如果在工程中有许多bug和一些微小改变的commit，那么可以使用--squash选项将feature分支中的所有commit取出，然后压缩这些commit为一个并加入到master分支中。  https://img-blog.csdnimg.cn/43b6a78f4dfa49109cf37aeb6007856d.png#pic_center |
| git checkout master  git merge --squash feature   * 注意：上述的合并命令并不会创建一个commit，实际上就是将feature中的每一次commit的更改应用到本地，所以还需要手动提交该commit：git commit -m <"your commit message">   合并结果如下  在这里插入图片描述 |

例子：

|  |
| --- |
|  |
|  |
| feature 分支上修改 |
|  |
|  |
|  |
|  |
|  |
|  |
| git checkout master  git merge --squash feature |
|  |
|  |
|  |
|  |
|  |
| HEAD -> master  feature -> feature |
| ours -> master  theirs -> feature |
|  |
|  |
|  |
| git commit |
|  |
|  |
|  |
|  |

还原为merge --squash前状态

|  |
| --- |
|  |

1. git 工具
2. git 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。储藏(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。 需要说明一点，stash是本地的，不会通过git push命令上传到git server上。 常用git stash命令：  （1）**git stash** save "save message"  : 执行存储时，添加备注，方便查找，只有git stash 也要可以的，但查找时不方便识别。  （2）**git stash list** ：查看stash了哪些存储  （3）**git stash show** ：显示做了哪些改动，默认show第一个存储,如果要显示其他存贮，后面加stash@{$num}，比如第二个 git stash show stash@{1}  （4）**git stash show -p** : 显示第一个存储的改动，如果想显示其他存存储，命令：git stash show  stash@{$num}  -p ，比如第二个：git stash show  stash@{1}  -p  （5）**git stash apply** :应用某个存储,但不会把存储从存储列表中删除，默认使用第一个存储,即stash@{0}，如果要使用其他个，git stash apply stash@{$num} ， 比如第二个：git stash apply stash@{1}  （6）**git stash pop** ：命令恢复之前缓存的工作目录，将缓存堆栈中的对应stash删除，并将对应修改应用到当前的工作目录下,默认为第一个stash,即stash@{0}，如果要应用并删除其他stash，命令：git stash pop stash@{$num} ，比如应用并删除第二个：git stash pop stash@{1}  （7）**git stash drop** stash@{$num} ：丢弃stash@{$num}存储，从列表中删除这个存储  （8）**git stash clear** ：删除所有缓存的stash |

1. git reset
2. 三棵树

|  |  |
| --- | --- |
| **树** | **用途** |
| HEAD | 上一次提交的快照，下一次提交的父结点 |
| Index | 预期的下一次提交的快照 |
| Working Directory | 沙盒 |

1. 工作流程

|  |
| --- |
|  |

1. 增加文件 版本1 和版本2

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
| 接着运行 git commit，它会取得索引中的内容并将它保存为一个永久的快照， 然后创建一个指向该快照的提 交对象，最后更新 master 来指向本次提交。 |
|  |
| 编辑 file.txt 文件 |
|  |
|  |
|  |
|  |
|  |
|  |
| 再次修改文件 生成V3版本 |
|  |

1. 重置文件

|  |
| --- |
| git reset --soft HEAD~ |
|  |
|  |
|  |
| 仅仅将HEAD 移到上一个版本。它本质上是撤销了上一次 git commit 命令。 |
| git reset --mixed HEAD~ |
|  |
|  |
|  |
| git reset --hard HEAD~ |
|  |
|  |
|  |
| 必须注意，--hard 标记是 reset 命令唯一的危险用法，它也是 Git 会真正地销毁数据的仅有的几个操作之一。 其他任何形式的 reset 调用都可以轻松撤消，但是 --hard 选项不能，因为它强制覆盖了工作目录中的文件。 |

1. git checkout

|  |
| --- |
| 运行 git checkout [branch] 与运行 git reset --hard [branch] 非常相似，它会更新所有三棵树使 其看起来像 [branch]，不过有两点重要的区别。   1. 如果当前分支有未提交的文件，则禁止切换 2. 仅仅将HEAD指针指向另一个分支，而不像git reset 会移动HEAD 指针。 |

1. git rebase –i

|  |
| --- |
| pick：正常选中  reword：选中，仅仅是修改提交信息；  edit：选中，rebase时会暂停，允许你修改文件和提交信息。  squash：选中，会将当前commit与上一个commit合并。  而squash命令，则会在git rebase --continue之后打开一个文件，该文件中将会出现所有设置为squash的commit，这时删除掉多余的commit message，留下（或者修改）一行作为合并之后的commit的commit message。  fixup：与squash相同，但不会保存当前commit的提交信息.  fixup命令，rebase的时候将会直接忽略掉它的commit message。  drop:命令表示你要丢弃这个commit以及它的修改 |
| 准备环境 |
|  |
|  |
| 场景1：修改某次提交的commit 信息 |
|  |
|  |
|  |
|  |
|  |
|  |
| 场景2：多条日志合并成一条 |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

1. git reflog

|  |
| --- |
| git reflog根本不遍历HEAD的祖先。它是HEAD所指向的一个顺序的提交列表。 |
|  |
|  |

1. git cherry-pick

|  |
| --- |
| 对于多分支的代码库，将代码从一个分支转移到另一个分支是常见需求。  这时分两种情况。一种情况是，你需要另一个分支的所有代码变动，那么就采用合并（git merge）。另一种情况是，你只需要部分代码变动（某几个提交），这时可以采用 Cherry pick。  git cherry-pick命令的作用，就是将指定的提交（commit）应用于其他分支。  git cherry-pick commit-hash  例如： |
|  |
|  |
|  |
|  |
|  |
|  |
| 多个提交  git cherry-pick A..B  上面的命令可以转移从 A 到 B 的所有提交。它们必须按照正确的顺序放置：提交 A 必须早于提交 B，否则命令将失败，但不会报错。  这个就不演示了。 |

1. git blame

|  |
| --- |
| 谁动了我的代码 |

1. git 工作流
2. git flow

|  |
| --- |
| 长期分支：   * 主分支master * 开发分支develop   短期分支：   * 功能分支（feature branch） * 补丁分支（hotfix branch） * 预发分支（release branch） |

1. github flow

是git的简化版。

仅仅只有一个master(main)分支。

1. git 提交代码总结

针对git 初学者 本着由浅入深的模式提交代码。可能提交代码流程不规范，但前提可以搞定问题。不会因为git 影响工作。

1. 剑英建议的提交流程

|  |
| --- |
| 假设我们现在git 刚入行 仅仅会几个简单的命令，可以使用此方式，相对简单。  我们每次提交代码使用默认的main 分支提交，当前自己的代码仓库仅仅只有main 分支。  git clone [git@github.com:yuhaijun999/dingo-store.git](mailto:git@github.com:yuhaijun999/dingo-store.git)  cd dingo-store  git remote add zetyun [git@github.com:dingodb/dingo-store.git](mailto:git@github.com:dingodb/dingo-store.git)  git remote -v |
| git fetch --all  git rebase origin/main  git rebase zetyun/main  编写代码：  可能我们改了很多代码了  这个时候我们想提交一个PR  先将我们的代码暂存起来  git stash  将zetyun 和我们的代码同步下  git rebase zetyun/main  这个时候是没有冲突的  git stash apply  或者  git stash pop  如果有冲突 则解决冲突。  解决完冲突  git add .  git commit -m “[feat][store] add scan feature”  git push origin main  然后就可以提交PR了。 |

1. 杰哥建议的提交流程

|  |
| --- |
| git 入门 可以对 git rebase 有一定理解，但是每次提代码还是在main 分支上。  git clone [git@github.com:yuhaijun999/dingo-store.git](mailto:git@github.com:yuhaijun999/dingo-store.git)  cd dingo-store  git remote add zetyun [git@github.com:dingodb/dingo-store.git](mailto:git@github.com:dingodb/dingo-store.git)  git remote -v |
| git fetch --all  git rebase origin/main  git rebase zetyun/main  编写代码：  git add .  git commit -m “[feat][store] add scan feature”  git push origin main  编写代码：  git add .  git commit --amend “[feat][store] add scan feature”  git push origin main -f  我们始终只有一个提交  ……  我们写的很多代码 想提交PR了。  git fetch --zetyun  git rebase zetyun/main  如果有冲突  解决冲突  解决完一个  git add ./src/somefile.cc  git rebase --continue  一直反复 知道没有冲突 并最后提示成功了。  git commit --amend “[feat][store] add scan feature”  git push origin main -f  最后提交一个PR。 |

1. 子辉建议的提交流程

|  |
| --- |
| git 仓库中不会出现公司的仓库。当前仓库中仅仅只有我们自己的仓库。当时我们的仓库中会有两个分支 main develop  main 分支用来同步公司仓库 develop 分支用于 checkout 分支用。  假设我们现在有个 feature-scan.  git clone [git@github.com:yuhaijun999/dingo-store.git](mailto:git@github.com:yuhaijun999/dingo-store.git)  git remote -v |
| git checkout -b develop main  git checkout -b feature-scan develop  我们现在在feature-scan 分支上提交代码  git add .  git commit -m “[feat][store] add scan feature1”  git push -u origin feature-scan  然后写代码  git add .  git commit -m “[feat][store] add scan feature2”  git push origin feature-scan |
| 拉取公司的最新的代码 同步main |
| git rebase main  git checkout develop  git rebase main  git merge --squash feature-scan  解决完冲突后  git push origin develop  然后提交PR。 |

1. 圣智建议的提交流程

|  |
| --- |
| git 入门 可以对 git rebase 有一定理解，但是每次提代码还是在main 分支上。  建议的分支提交代码，但是会把自己仓库的main 删除 ，直接使用公司的main 分支，每个分支开头是[wip](work in process)  git clone [git@github.com:yuhaijun999/dingo-store.git](mailto:git@github.com:yuhaijun999/dingo-store.git)  cd dingo-store  git remote add zetyun [git@github.com:dingodb/dingo-store.git](mailto:git@github.com:dingodb/dingo-store.git)  git remote -v |
| git fetch --all  git rebase origin/main  git rebase zetyun/main  删除origin/main 对应的本地分支 main    建立本地 main 和 zetyun/main 对应关系。      git checkout -b [wip]-feature-scan main  编写代码：  git add .  git commit -m “[feat][store] add scan feature”  git push origin main  编写代码：  git add .  git commit --amend “[feat][store] add scan feature”  git push origin main -f  我们始终只有一个提交  ……  我们写的很多代码 想提交PR了。  git fetch --zetyun  git rebase zetyun/main  如果有冲突  解决冲突  解决完一个  git add ./src/somefile.cc  git rebase –continue  一直反复 知道没有冲突 并最后提示成功了。  git commit --amend “[feat][store] add scan feature”  git push origin main -f  最后提交一个PR。 |

1. git 参考资料

|  |  |
| --- | --- |
| Title | Link |
| Pro Git | http://git-scm.com/book |
| 使用git和github进行协同开发流程 | https://github.com/livoras/blog/issues/7 |
| GitHub秘籍 | https://github.com/tiimgreen/github-cheat-sheet/blob/master/README.zh-cn.md |
| Git分支管理策略 | http://www.ruanyifeng.com/blog/2012/07/git.html |
| git cherry-pick 教程 | https://www.ruanyifeng.com/blog/2020/04/git-cherry-pick.html |
| 图解Git | http://marklodato.github.io/visual-git-guide/index-zh-cn.html |
| 自学Git，有哪些书籍或者好的学习资源？ | https://www.zhihu.com/question/38008771/answer/74558769 |
| git-workflow-tutorial | https://github.com/xirong/my-git/blob/master/git-workflow-tutorial.md |
|  |  |