# 基本命令

1. 进入要管理的目录
2. git init 初始化，即让git帮助我们管理当前文件夹
3. git status 检测 当前目录下文件的状态，新增和修改过后的文件都是红色。
4. 三种状态的变化
   1. 红色：新增的文件/修改了的老文件，（红色变绿色：通过git add 文件名/. ，dot是把所有未管理起来的文件都管理起来）
   2. 绿色：已经管理的文件，生成版本通过git commit -m ‘描述信息’
   3. 生成版本。
   4. 查看版本，git log.
5. 注意的问题
   1. 若执行commit时，报错，那么可能是没有个人配置。提交方必须有自己的标识
   2. git config --global user.email [you@example.com](mailto:you@example.com)
   3. git config --global user.name yourname
   4. 配置信息只需执行一次即可。

# 分区



# 反复回滚

1. 回滚到之前(某一历史的)版本

git log

git reset – –hard 版本号

2）取消回滚，回滚至之后（某一新的）版本

查看曾经的历史操作：git reflog

恢复至某一版本：git reset --hard 版本号

# 基本命令总结

|  |
| --- |
| git init  git config --global user.email “your email addr”  git config --global user.name “your name”  git add <file>/.  git commit -m ‘描述’  git log  git reflog  git reset --hard 版本号 |



git reset HEAD <file> 由暂存状态回到未暂存状态。

git checkout -- <file> 回到最原始状态

# 初识分支

若没有分支功能，只能单基线管理，那么以下场景不方便：

1. 多人同时开发部分功能，然后再合并；
2. 基于当前上线版本正在开发新功能，但上线版本发现bug需要紧急修复。



查看分支：git branch

在当前分支中，创建分支：git branch <分支名如dev>

切换到别的分支：git checkout <分支名如dev>

切换回主分支：git checkout master

将其他分支合并到主分支：先确认切换到master分支，然后git merge <分支名如bug>

删除无用的分支：git branch -d <分支名如bug>

合并分支时，可能产生冲突，此时需要找到冲突的文件位置，手动解决，然后再add及commit，或者你也可以选择通过git merge –abort进行取消合并。

总结：

|  |  |
| --- | --- |
| 查看分支 | git branch |
| 创建分支 | git branch <分支名称> |
| 切换分支 | git checkout <分支名称> |
| 分支合并（可能产生冲突） | git merge <要合并的分支>  注意：先确认切换分支再合并 |
| 删除分支 | git branch -d <分支名称> |

# 工作流

注意开发规范（工作流规范）：

分支至少得有两个，包括主分支、开发分支。

定义好管理规范，master上不要瞎搞，只保留上线版本/稳定版。其他正在开发的或公测版等都在开发分支上搞。

# gitHub基本操作

|  |  |
| --- | --- |
| 为远程仓库起别名 | git remote add <别名> 远程仓库地址 |
| 向远程仓库推送代码 | git push -u <别名> 分支名 |
| 克隆远程仓库的代码 | git clone <远程仓库地址>  注：内部已实现git remote add <别名> 远程仓库地址 |
| 切换分支 | git checkout <分支名> |

* + 1. 在甲方dev分支开发；
    2. 提交代码：
       - 1. git add .
         2. git commit -m ‘xxxxx’
         3. git push <仓库别名> dev
    3. 在乙方继续开发dev分支并提交
       - 1. 拉代码：git pull <仓库别名> dev
         2. 继续开发
         3. 提交

git add .

git commit -m ‘xxx’

git push <仓库别名> dev

* + 1. 两地开发测试完毕，将dev分支合并到master，进行上线
       - 1. git checkout master
         2. git merge dev
         3. git push <仓库别名> master
    2. 把dev分支也推送到远程仓库
       - 1. git checkout dev
         2. git merge master (比较好的习惯，保持当前dev和master版本一样)
         3. git push <仓库别名> dev



# rebase（变基）命令

场景1）整合多次提交的版本记录变成一次提交记录，使git记录变简洁，让别人只看想看的。

|  |  |
| --- | --- |
| git rebase -i 版本号 | 指将此版本号开始直到最新的版本一并合并 |
| git rebase -i HEAD~3 | 指将最近的3次整合成一次提交  命令执行过程中的符号s：指该版本合并到其上一版本记录 |
| 注意：尽量不要把已经push到远程仓库的版本记录进行rebase，否则版本管理显得更乱 | |

场景2）多个分支记录合并



操作示例：

1）同在C2基础上，dev 分支开发并提交c3，同时master分支开发并提交至C4；

2) 可以图形化查看下，git log --graph

3）切换至dev分支，执行变基命令：git rebase master

4）切换至master分支，需要继续执行：git merge dev

5）可以图形化查看下是否变基成功：git log --graph --pretty=format:”%h %s”, 这里的%h指版本号哈希值，%s指版本说明。

场景3）不想产生分叉



注意：当git rebase产生冲突时：

1. 解决冲突；
2. git add <冲突文件>或.
3. git rebase –continue
4. 可以通过git log --graph --pretty=format:”%h %s”查看操作结果。

# 借助BeyondCompare解决冲突

1. 安装BeyondCompare
2. 在git中配置

|  |
| --- |
| git config --local merge.tool bc3  git config --local mergetool.path ‘路径名’  git config --local mergetool.keepBackup false  注意：--local指的是仅针对当前项目有效 |

1. 启动该工具

git mergetool

1. 操作Beyond Comparer软件解决冲突，完毕后保存。

# 阶段总结

|  |  |
| --- | --- |
| 添加远程连接 | git remote add 仓库名 地址 |
| 推送代码 | git push 仓库 分支 |
| 下载代码（第一次） | git clone 地址  注意：包含了所有代码 |
| 拉取（更新）代码 | git pull 仓库 分支  等价于  git fetch仓库 分支  git merge仓库/分支 |
| 保持代码提交记录整洁 | git rebase 分支 |
| 提交记录图形化展示 | git log –graph –pretty=format:”%h %s” |

# 多人协作



设置仓库以支持多人协作（适用于个人项目）

1. 进入setting
2. 在合作者处邀请其他合作者

在公司，一般使用组织的形式。

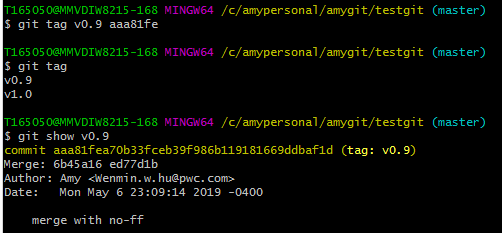
1. 设置公司名称
2. 设置邮箱等
3. 在一个组织内包含多个项目仓库（新建项目就是在一个组织内操作了）
4. 可以在组织内进行权限细分

打标签

1. git tag -a v1.2.3.4 -m “第1版”
2. git push <仓库> --tags

注意：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。

注意：默认标签是打在最新提交的commit上的有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？方法是通过git log找到历史提交的commit id，然后打上就可以了，可以用git show <tagname>查看标签信息。



查看标签git tag

删除标签

1. git tag -d 标签名
2. 如果标签已经推送到远程，要删除远程标签需先从本地删除，然后从远程删除。删除命令也是push，格式为：git push <仓库> :refs/tags/标签名

示例过程-邀请成员

1）创建分支dev,并切换到dev分支（git checkout -b dev）

2）推送到远程：git push origin dev

3）各开发成员在github或gitlab或coding等仓库平台中注册各自的账号；

4）邀请成员（选择角色是成员还是owner）;

5）成员同意后，邀请成员进入组织成功。

示例过程-让邀请进入组织的成员参与开发

1. 进入settings，对于组织，默认成员的权限仅是只读
2. 除了组织，也可以对某一项目设置权限
3. 新加入的成员获取原有代码：git clone 地址
4. git checkout dev
5. 从dev分支基础上接着创建成员自己的开发分支：git checkout -b xgnbranch
6. 可以查看下当前所有分支：git branch
7. git add . ; git commit -m “xxx”; git push origin origin xgnbranch
8. 成员在新功能的分支开发已完成，让team leader做code review（基于pull request 或merge request 来做code review）。
9. 在github的settings中的branches中设置branch protection rules规则（require pull request reviews before merging等规则），对其他分支如master也可以增加设置相关规则。
10. 成员想合并自己的分支到dev分支时，必须先提交pull request.
11. review完成后合并分支
12. 将合并后的origin/dev也更新到本地dev: git pull origin dev。

示例过程-测试合并各成员后的dev

1. 新创建个release分支：git checkout -b release
2. 对release进行测试，并做bug修复；
3. 测试完成后，提出pull request；（可以指定Reviewer）
4. 管理者审查后合并。
5. 将release分支合并到dev分支（git checkout dev; git merge release）
6. release到master时，可以选择pull request 或merge，但在review时，还是要让成员遵守规范，要经过review才能合并分支，不要直接merge。
7. 此时，可以选择删除release分支。
8. 在master分支打标签
   1. git tag -a v2 -m “第二版，增加XX功能”；
   2. git push origin –tags
9. 运维人员可以去下载代码做上线了。

git clone -b v2 仓库地址

注意：在review、merge过程中可能产生冲突，注意解决完冲突再合并。

# 给开源社区贡献代码

1. fork源代码

将别人的源代码拷贝到我自己的仓库

1. 在自己的仓库进行代码修改
   1. 从自己仓库中下载代码

git clone 仓库项目地址

* 1. 自己修改并提交至自己的仓库

1. 给源代码的作者提交修复bug或贡献新代码的申请（pull request）；

在自己的仓库中进行new pull request

1. 源代码作者收到申请，审核可能接收、拒绝或不理你。

# 其他

1. 配置

|  |  |
| --- | --- |
| git config --local user.name ‘zhangsan’  git config --local user.email ‘zhangsan@xxx.com’ | 项目配置文件  有效范围：项目  位置：项目/.git/config |
| git config --global user.name ‘zhangsan’  git config -- global user.email ‘zhangsan@xxx.com’ | 全局配置文件  有效范围：  位置：~/.gitconfig |
| git config --system user.name ‘zhangsan’  git config -- system user.email ‘zhangsan@xxx.com’ | 系统配置文件  位置：/etc/.gitconfig  注意：需要root权限。 |

先在本地找，然后全局，然后系统。

应用场景：

* 1. 配置用户名、邮箱
  2. 配置mergetool
  3. git remote add origin 地址 （默认添加在本地配置文件中--local）
  4. 等等。

1. 免密码登录
   1. 在URL中体现

|  |
| --- |
| 如原来的地址：<https://github.com/xxx/learnspa.git>  修改的地址： <https://用户名:密码@github.com/xxx/learnspa.git>  git remote add origin https://用户名:密码@github.com/xxx/learnspa.git（直接修改配置文件也可以）  git push origin master时就不用输用户名和密码了 |

* 1. SSH实现（企业常用）

|  |
| --- |
| * + - 1. 生成公钥和私钥（默认放在~/.ssh目录下，id\_rsa.pub为公钥，id\_rsa为私钥）       2. 拷贝公钥的内容，并设置到github；(settings中的SSHandGPGKeys)       3. 在git本地中配置ssh地址   gitremote add origin git@github.com:xxx/learnspa.git   * + - 1. 以后使用就可以免密了   如可以直接操作gitpush origin master |

* 1. git可以自动管理凭证（本地用较方便）。

基于操作系统实现的。

1. git忽略文件

让git 不再管理当前目录下的某些文件。在平时工作中一定注意使用，避免敏感数据上传。

项目目录下，创建gitignore文件：vim .gitignore

编辑其中内容如：

\*.h （所有后缀为h的文件）

！a.h （但排除a.h文件）

xdir/ (排除xdir文件夹下的所有文件)

\*.py[c|a|d] （排除以\*.pyc 、\*.pya、\*.pyd结尾的）

更多的参考https://github.com/github/gitignore

1. 任务管理相关
   1. issues(讨论、话题等)，用于文档及任务管理

有的公司使用github做任务管理，可以提出问/话题指派给某个人回答。利用这个功能可以汇总问题，方便给新来的看。也可以给问题贴上bug标签，方便汇总管理出现过的bug。分配给谁的任务状态也能清晰看出来是open还是closed。

* 1. wiki，用于项目文档

通过wiki可以创建、汇总、管理项目相关文档。

通过issues和wiki，出现如bug或项目进度等需要解答或了解的问题时可以先查下issues或wiki。