# Git学习笔记

## Git Bash的配置

git config --global user.name ＂your name＂

git config --global user.name ＂your email＂

--global：这个参数表示这台机器上的所有Git仓库都会使用这个配置。也可以对某个仓库指定不同的用户名和Email地址。

**代理设置**

git config --global http.proxy　＂代理地址:端口号＂

git config --global https.proxy ＂代理地址:端口号＂

**代理查看**

git config --global --get http.proxy

git config --global --get https.proxy

**代理取消**

git config --global --unset http.proxy

git config --global --unset https.proxy

--global:全局 --local:当前目录

## 创建版本库

1. **git init +文件夹名**：将建立的文件夹初始化为git仓库，会在目录下生成一个.git的隐藏文件夹。（ls –ah指令用于显示目录下所有文件）
2. **git add +文件名**：将文件从工作区添加至暂存区。
3. **git commit -m ＂注释内容＂**：将文件从暂存区提交至本地仓库。
4. **git commit -a -m “注释内容”：**直接提交（前提是文件之前已经add至暂存区）。

**注：git add .、git add -u、git add -A的区别：**（使用git add –help指令获取）

**git add . ：**（不包括删除）监控工作区的状态树，把工作时的所有变化提交到暂存区，包括文件内容的修改(modified)以及新文件(new)，但不包括被删除的文件。

**git add -u ：**（不包括未跟踪）仅监控已经被add的文件（即tracked file），将被修改的文件提交到暂存区，但不会提交untracked file。

**git add -A ：**（包括所有）提交所有tracked和untracked文件的变动。

## 版本回退

1. **git status：**查看仓库当前状态
2. **git diff：**查看区别，修改了哪些文件。
3. **git log：**显示从最近到最远的提交日志，可以查看版本号。简化版log可以加--pretty=oneline参数。
4. **git reset --hard HEAD^：**回退至上一个版本。（HEAD表示当前版本，HEAD^表示上个版本，HEAD^^表示上个版本，HAED~100表示之前第100个版本）

**git reset --hard +版本号：**回退至指定版本。

1. **git reflog：**查看历史命令，获取版本号，以撤销版本回退。

## 撤销修改

1. **git checkout -- 文件名：**撤销工作区的修改。

例：git checkout -- readme.txt ，这里分两种情况：

一种是readme.txt修改后还没有被放到暂存区，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt添加到暂存区后，又作了修改，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次**git commit**或**git add**时的状态。

**注：**git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令。

1. **git reset HEAD <file>：**把暂存区的修改（未commit）撤销掉，重新放回工作区
2. **git restore --staged +文件名：**撤销暂存区文件至工作区。
3. **git rm <flie>：**删除指定文件。

**注：**从来没有被添加到版本库就被删除的文件，是无法恢复的！

## 远程仓库

1. **git remote add +远程仓库名+https链接/ssh地址：**将本地仓库与远程仓库关联。

例： git remote add origin <https://github.com/Miao-Tea/learngit.git> （https）

git remote add origin [git@github.com:Miao-Tea/learngit.git](mailto:git@github.com:Miao-Tea/learngit.git) （ssh）

origin为远程仓库的默认名字。

1. **git remote –v：**查看远程仓库配置信息。
2. **git remote rm +远程仓库名：**删除远程仓库的配置。
3. **git push -u origin master：**本地仓库推送至远程仓库。-u参数作用不但把本地master分支推送至远程分支，而且也把这两个分支关联起来。以后再推送时可以省略-u。
4. **git clone +https链接/ssh地址：**克隆远程仓库。

例：git clone <https://github.com/Miao-Tea/learngit.git>

注：git支持多种协议，ssh协议速度快，不用输入密码，但在只开放http端口的公司内部就无法使用ssh协议而只能用https。

### 生成SSH秘钥

（注:ssh协议没能成功实现）

**注：**使用ssh协议时需要生成ssh-key，使用https协议不需要ssh-key。

1. **ssh-keygen -t rsa -C "youremail@example.com"：**生成ssh-key秘钥指令，-C代表拷贝到某个地址下。

例：ssh-keygen -t rsa -C "2451195182@qq.com"

注：新建过程中，passphrase（密码）设置为空即可，新建成功后，在.ssh目录下会分别生成公钥和私钥文件：id\_rsa.pub，id\_rsa。

1. **在github首页：**点头像右侧下三角→settings→SSH and GPG keys→SSH keys→New SSH key→打开本地id\_rsa.pub，将内容复制进去→Add SSH key。
2. **本地添加私钥：**（在~/.ssh目录下）
   1. SSH agent启动bash：ssh-agent bash --login –i
   2. 增加私钥：ssh-add

## 分支管理

**分支的概念**

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。如果只有一条时间线，该分支就是master分支。HEAD与master的关系如下图所示，HEAD指向当前分支，master指向提交。



创建一个dev分支。Git创建一个分支很快，因为除了增加一个dev指针，改变HEAD的指向。



从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变。



在dev上的工作完成了，就可以把dev合并到master上，Git合并分支也很快！同样是改变指针。



删除dev分支就是把dev指针删掉，最后就只剩下一条master分支。



1. **git branch +分支名：**建立分支
2. **git branch：**查看分支，列出所有分支，并会在当前分支名前加\*号标注。
3. **git checkout +分支：**切换到指定分支上。新版本git中**git switch +分支** 指令与其作用相同。
4. **git checkout -b +分支名：**建立并切换分支。该指令等价于建立分支和切换分支两条指令。同样也等价于**git switch -c +分支名**。

例： git checkout -b dev等价于

git branch dev

git checkout dev

1. **git merge +分支名：**合并某分支到当前分支。

例：将dev合并到master。切换至master，执行**git merge dev**

**注**：**Fast-forward**表示“快进模式”合并，即直接将master指向dev的当前提交，优点是合并速度快，缺点是删除分支时会丢掉分支信息，git log时看不到合并的历史信息。

禁用该模式需加参数**--no-ff**，使用时会产生一个提交，需配合**-m**参数使用。

例：**git merge --no-ff -m "merge with no-ff" +分支名**

1. **git branch -d +分支名：**删除指定的分支。**-D**表示强制删除。

### 合并冲突

1. 当两个分支都对同一个文件做了不同的修改，合并时会发生冲突，这时候需要手动解决冲突，提交后再进行合并。
2. 发生冲突时，使用git status可以查看冲突的文件，并且在冲突文件中git用<<<<<<<，========，>>>>>>>标记出不同分支的内容。
3. 查看分支合并图可以使用以下指令：

**git log --graph --pretty=oneline --abbrev-commit**（版本号简化显示，单词abbreviation：简写）

**git log --graph --pretty=oneline**（版本号非简化显示）

**git log –graph**（详细显示）

### 分支策略

master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活，干活都在dev分支上。每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。所以，团队合作的分支看起来就像这样：



### Bug分支

修复bug时，我们需要新建专门用于修复bug的分支，但我们正在干活的分支上的任务没有完成时是不能提交的，因而无法建立用于修复bug的分支。若要建立用于修复bug的分支，需要先保存当前工作分支上的工作进度。可在当前分支下执行指令：

1. **git stash：**保存当前分支的工作进度，使之“隐藏”。

**注：**此时执行**git status**，工作区是干净的。因而可以建立用于修复bug的分支，在该分支上修复完bug提交合并后，可切回原来的工作分支，恢复保存的工作进度。

1. **git stash list：**查看被隐藏的工作进度。（编号越大离当前时间越近）

恢复工作进度时，可用下面指令：

1. **git stash apply：**但是恢复后，stash的内容并不删除，需要用**git stash drop**来清除日志；
2. **git stash pop：**恢复的同时把stash内容也删了。
3. **git cherry-pick +版本号：**会获取某一个分支的单笔提交，并作为一个新的提交引入到当前分支上。（比如master分支的bug修复后需要cherry-pick到工作分支上）

**注：git cherry-pick　<options>　<commit-ish>**

常用options:

--quit 　退出当前的chery-pick序列

--continue 继续当前的chery-pick序列，冲突修复后引用

--abort 取消当前的chery-pick序列，恢复当前分支

-n, --no-commit 不自动提交

-e, --edit 编辑后提交信息

### 多人协作

**推送分支**

**git push origin +本地分支名：**将本地分支推送至远程仓库的对应分支上，远程仓库的名称默认为origin。

例：git push origin master ：将本地仓库master分支推送至远程仓库master分支上。

git push origin dev ：将本地仓库dev分支推送至远程仓库dev分支，若远程不存在该分支，推送时自动创建。

**抓取分支**

1. **git checkout -b 分支名 origin/分支名：**抓取远程库的指定分支至本地库，前提是该分支在远程库存在。

例： git checkout -b dev origin/dev：抓取远程库dev分支，并在本地库建立dev分支。若提示有误，执行git pull后再执行该指令。

1. **git pull：**拉取远程分支并与本地分支合并。当远程分支比本地分支新时，可用该指令更新本地分支。
2. **git pull <远程主机名> <远程分支名>:<本地分支名> ：**若远程分支是与当前分支合并，则冒号后面的部分可以省略。
3. **git branch --set-upstream-to <branch-name> origin/<branch-name>：**将本地分支和远程对应的分支建立链接关系。

例：git branch --set-upstream-to dev origin/dev：将本地dev分支与远程dev分支建立链接关系。

1. **git rebase：**将分叉的提交历史变成一条线，含有commit操作。将本地提交与远程提交作对比，生成查分提交信息，具体提交内容我们未知。

## 标签管理

标签相当于提交的别名，可以理解成便于理解和记忆的提交版本号。

### 创建、查看标签

1. **git tag +标签名：**创建标签，默认打在最新提交的commit上，即当前提交HEAD。

**注：**标签总是和某个commit相关。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。

1. **git tag +标签名+ID：**给指定的commit ID打标签。

例：**git tag v1.0：**给当前提交打标签为v1.0

**git tag v1.1 f52c633：**给ID 为f52c633的提交打标签为v1.1。

还可以指定参数，**-a指定标签名，-m指定说明文字**。

例：**git tag -a v0.1 -m "version 0.1 released" 1094adb**

1. **git tag：**查看所有标签。
2. **git show +标签名：**查看指定标签的详细信息，包括说明文字。

### 删除、推送标签

1. **git tag -d +标签名：**只删除本地标签。
2. **git push origin :refs/tags/标签名：**只删除指定的远程标签。

**注：**若要删除远程标签，首先应删除本地标签，然后再用该指令删除远程标签，否则本地标签依然存在。

1. **git push origin +标签名：**推送本地标签至远程。
2. **git push origin --tags：**一次性推送所有本地标签至远程。

## Patch的生成与应用

git提供了**git diff**和**git format-patch**两种补丁生成方案。

**git diff** ：生成的.diff补丁文件只是记录文件改变的内容，不带有commit信息,多个commit可以合并成一个diff文件。该补丁是标准补丁，git、svn均可以使用。

**git format-patch** ：生成的.patch补丁文件不仅带有文件改变的内容，也带有commit记录信息,每个commit对应一个patch文件。该补丁文件是git专属补丁，只能应用于git。

### 生成patch

#### 使用git diff生成标准patch

git diff > patch

git diff --cached > patch

git diff branchname --cached > patch

git diff 【commit ID1】【commit ID2】 > patch

指令执行后就会在当前目录下生成一个patch文件。

例：git diff 2a2fb4539925bfa4a141fe492d9828d030f7c8a8 89aebfcc73bdac8054be1a242598610d8ed5f3c8 > patch.diff

#### 使用git format-patch生成git专用patch

$ **git format-patch HEAD^** 　 #生成最近1次commit的patch

$ **git format-patch HEAD^^**　　 #生成最近2次commit的patch

$ **git format-patch HEAD^^^** 　 #生成最近3次commit的patch

$ **git format-patch HEAD^^^^**  #生成最近4次commit的patch

$ **git format-patch <r1>...<r2>** #生成两次commit间的patch（包含两个commit ，<r1>和<r2>都是具体的commit号)

$ **git format-patch -1 <r1>** #生成单个commit的patch

$ **git format-patch <r1>** #生成从某个commit以来的patch（不包含该commit）

$ **git format-patch --root <r1>** #生成从根到r1提交的所有patch

同样以上指令执行后会在当前目录下生成补丁文件。

#### 两种patch的比较

**兼容性：**git diff生成的patch兼容性强，可以应用于非git平台，而使用git format-patch生成的patch只能应用于git。

**除错功能：**git diff生成的patch，可以用git apply --check 查看补丁能否干净顺利地应用到当前分支中。而git format-patch 生成的补丁若在应用时出现问题，git am会给出提示，并协助完成打补丁工作。

**版本库信息：**git diff生成的patch不包含版本库信息。而git format-patch 生成的patch含有补丁开发者的名字，在应用补丁时，这个名字会被记录进版本库。

### 应用patch

在应用patch文件打补丁时，主要用到**git am +patch文件**和**git apply +patch文件**这两个指令。

1. **git am +patch文件：**只适用于git format-patch指令生成的补丁文件。

git am常用的指令如下：

1. **git am 0001.patch：**# 将名字为0001.patch的patch打上。
2. **git am --signoff 0001.patch：**# 添加-s或者--signoff，可以把自己的名字添加为signed off by信息，作用是注明打patch的人是谁。
3. **git am ~/patch-set/\*.patch：** # 将路径~/patch-set/中的patch按照先后顺序打上。
4. **git am --abort：**# 当git am失败时，将已经在am过程中打上的patch废弃掉(比如打三个patch，到第三个patch时有冲突，那么这条命令会把打上的前两个patch丢弃掉，返回没有打patch的状态)。
5. **git am --resolved：**#当git am失败，解决完冲突后，这条命令会接着打patch
6. **git am --skip：**# 发生冲突时，跳过此次冲突。

**注：在使用git am之前，首先要执行git am --abort 一次，来放弃掉之前的am信息，这样才可以进行一次全新的am。否则可能会遇到错误提示：.git/rebase-apply still exists but mbox given.**

1. **git apply +patch文件：**通用于所有的patch补丁文件，包括git diff和git format-patch生成的补丁文件。

git apply有以下的辅助指令：

1. **git apply --stat +patch文件**： # 查看patch的情况

例：git apply --stat 0001.patch

1. **git apply --check +patch文件**：# 检查patch是否能够打上，如果没有任何输出，则说明无冲突，可以打上。

例：git apply --check 0001.patch

### 冲突解决

**方案一（推荐）：**

1. 根据git am失败的信息，找到发生冲突的具体patch文件，然后用命令git apply --reject <patch\_name>，强行打这个patch，发生冲突的部分会保存为.rej文件（命名为patch文件+.rej后缀，保存在.git/rebase-apply），未发生冲突的部分会成功打上patch。
2. 根据.rej文件，通过编辑该patch文件的方式解决冲突。
3. 使用git am --abort废弃上一条am命令已经打完的patch。
4. 使用git am ~/patch-set/\*.patch重新打patch。

**方案二：**

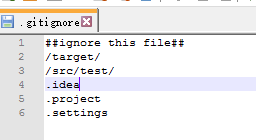
1. 根据git am失败的信息，找到发生冲突的具体patch文件，然后用命令git apply --reject <patch\_name>，强行打这个patch，发生冲突的部分会保存为.rej文件，未发生冲突的部分会成功打上patch。
2. 根据.rej文件，通过编辑发生冲突的code文件的方式解决冲突。
3. 通过命令git add <file\_name>将该patch涉及到的所有文件（不仅仅是发生冲突的文件）添加到工作区中。
4. 使用git am --resolved命令继续打patch。(git am --resolved 和 git am --continue是一样的)

## Git自定义配置

### 忽略指定文件

工作区中的某些文件，当我们不希望被提交时，需要在工作区编写.gitignore文件，其过程如下。

1. 在**工作区目录下**建立**.gitignore**文件；
2. 编辑.gitignore文件，可以是具体文件名、路径、通配符等；



**注：/target/ ：**过滤文件设置，表示过滤这个文件夹；

**\*.mdb ，\*.ldb ，\*.sln ：**表示过滤某种类型的文件；

**/mtk/do.c ，/mtk/if.h** ：表示指定过滤某个文件下具体文件；

**!\*.c , !/dir/subdir/ ：** !开头表示不过滤；

**\*.[oa]  支持通配符：**过滤repo中所有以**.o**或者**.a**为扩展名的文件；方括号“[]”包含单个字符的匹配列表。

1. 利用git status查看是否忽略了指定的文件。

**注：.gitignore**只能忽略那些原来**没有被track**的文件，如果某些文件已经被纳入到版本管理中，则修改**.gitignore**是无效的。解决方法是先把本地缓存删除（改变成未track状态），然后再提交：

**git rm -r --cached .**

**git add .**

**git commit -m 'update .gitignore'**

### 配置别名

当某些指令过长或者容易敲错时，可以通过配置别名来简化指令。配置别名的指令如下：

**git config --global alias.别名 原名：**

**注：**

1. 原名为一个单词时，不需要加引号；当有多个单词时需要加引号，单双引号（‘’、“”）均可。
2. “**alias.别名**”两者要写在一起。
3. --global可有可无。加--global表示别名对所有仓库起作用，不加仅对当前仓库起作用。

加--global时配置文件**.gitconfig**在用户主目录下的一个隐藏文件中，不加时在每个仓库的**.git/config**文件中。也可以通过修改配置文件直接配置别名，别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

**例：git config --global alias.st status：** 配置后git st = git status

**git config --global alias.co checkout：** 配置后git co = git checkout

**git config --global alias.unstage 'reset HEAD'：** 配置后git unstage = git reset HEAD

**git config --global alias.last 'log -1'：**(查看上一条log)配置后git last = git log -1

**※git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"**

### 搭建Git服务器

如果不使用类似github或码云（gitee.com，国内的代码仓库）的第三方远程仓库，则需要搭建自己的git服务器。搭建Git服务器需要准备一台运行Linux的机器，通常使用的是Ubuntu和Debian。在Ubuntu系统上搭建Git服务器的前提条件是有sudo权限，其详细步骤如下。

1. 安装git。

sudo apt-get install git

1. 创建一个git用户，用来运行git服务。

sudo adduser git

1. 创建证书登录

收集所有需要登录的用户的公钥id\_rsa.pub，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

1. 初始化Git仓库

先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

sudo git init --bare sample.git

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库是为了共享，不能让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git：

sudo chown -R git:git sample.git

1. 禁用shell登录

出于安全考虑，第二步创建的git用户不允许登录shell，可以通过编辑/etc/passwd文件完成。

将 **git:x:1001:1001:,,,:/home/git:/bin/bash**

改为： **git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell**

1. 克隆远程仓库

现在，通过git clone命令就可以克隆远程仓库了。

## 附录

### 解决每次git push需要输入用户名密码的问题

**解决方案**

**1.执行下面命令**

git config --global credential.helper store

这个时候~/.gitconfig文件中会多一行

[credential]

helper = store

**2.执行git push再次输入用户名和密码**

此时/root/.git-credentials文件中会多一行类似https://{username}:{password}@github.com这种形式的内容。

### UI界面多色化显示

**git config --global color.ui true：** UI界面多色化显示开。

**git config --global color.ui false：** UI界面多色化显示关。