Git

# 入门

git命令行操作跟Linux一样。

# 1 安装

1. Use git from git bash only...，其他默认 下一步
2. 查看环境变量配置，没有则手动配置，path: E:\programs\Git\bin
3. 配置git，桌面右键git bash...，配置用户名和邮箱。

git config --global user.name hua

git config --global user.email “2827890646@qq.com”

该配置文件在本地计算机c:\user\用户\.gitconfig

# 2 git status，git commit等中文信息乱码问题

# 3 本地git与远程github保持登录联通

Git的远程仓库就github一个（仓库托管）。

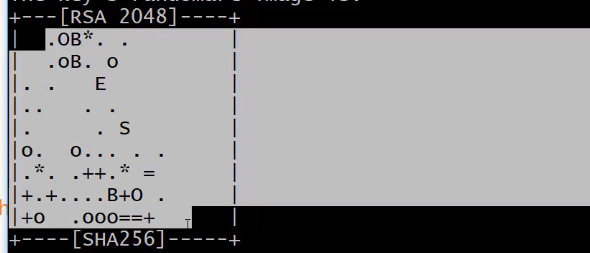
https://github.com

为了在本地和远程仓库之间进行操作，可采取免密钥登录，配置ssh。现在本地配置，然后把公钥发送给远程仓库github。

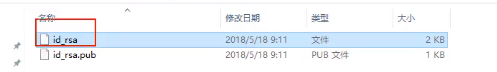
1. 在本地git命令行中输入：

ssh-keygen -t rsa -C 28278906464@qq.com

1. 然后一直回车，出现下图则表示成功。



1. 发送给远程github
2. github -> settings -> SSH and ... -> New SSH -> title & key

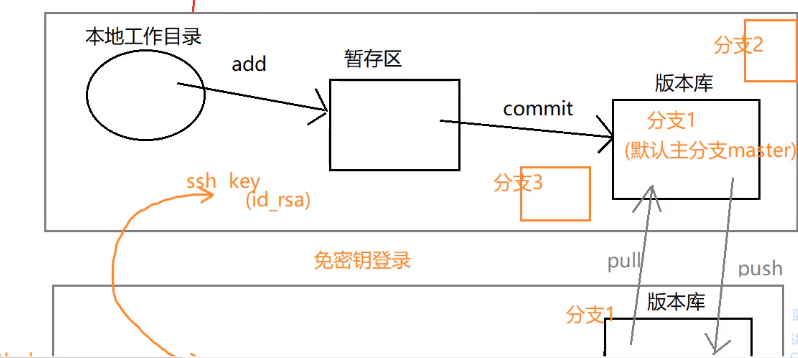


1. 复制本地公钥到key中，删掉最后的回车占位符。
2. 如果本地和远程成功通信，则在本地./ssh目录中自动生成known-hosts文件。
3. 测试本地仓库与远程仓库的登录联通性，git命令行：

ssh -T git@github.com

# 4 本地创建git项目与远程项目进行关联

1. 先创建一个文件夹demo。
2. 进入该文件夹，鼠标右键选择git bash...，打开git命令端，输入git init，会自动在demo目录下生成.git文件夹（该文件夹是隐藏的）。
3. github上创建一个Repository。



1. 第一次发布项目（本地 -> 远程）

git add . // 文件 -> 暂存区，“.”表示当前目录下的所有文件提交暂存区

git commit -m “第一次提交” // 暂存区 -> 本地分支（默认为master）

git remote add origin 远程项目所在的ssh地址（或http协议地址）

git push -u origin master // 本地分支 -> 远程项目

1. 第一次下载项目（远程 -> 本地）

本地创建个文件夹，进入文件夹，打开git命令终端，输入下面命令：

git clone git@github.com:yanse/mygitdemo.git

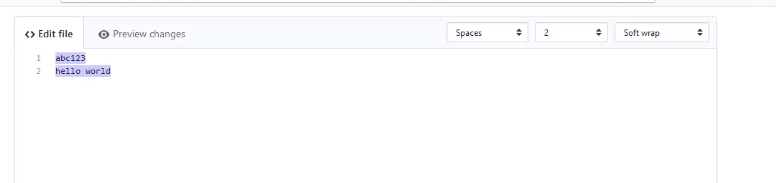
1. 非第一次提交（本地 -> 远程）

git add .

git commit -m “第二次提交”

git push origin master //注意：这里去掉了 -u（其实可写成git push）

1. 更新（远程 -> 本地）
2. 在远程项目中修改文件



1. 本地项目中，git命令行输入git pull即可更新。

# 进阶

# 1 git优势

1. 本地版本控制；
2. 重写提交说明；
3. 每一次操作都可以回滚，还原；
4. 分支系统，分布式。

# 2 git 三种状态

1. 已修改（modified）
2. 已暂存（staged）
3. 已提交（committed）

# 3 设置用户和邮箱

Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在三个不同的位 置：

1. /etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果在执行 git config 时带上--system 选项，那么它就会读写该文件中的配置变量。 （由于它是系统配置文件，因此你需要管理员或超级用户权限来修改它。）

2. ~/.gitconfig 或 ~/.config/git/config 文件：只针对当前用户。 你可以传递 --global 选项让 Git读写此文件，这会对你系统上 所有 的仓库生效。

3. 当前使用仓库的 Git 目录中的 config 文件（即 .git/config）：针对该仓库。 你可以传递 --local 选 项让 Git 强制读写此文件，虽然默认情况下用的就是它。。 （当然，你需要进入某个 Git 仓库中才能让该选项生效。）

每一个级别会覆盖上一级别的配置，所以 .git/config 的配置变量会覆盖 /etc/gitconfig 中的配置变量。

# 4 将某个目录纳入git管理

git init // 此时，git命令行中显示master，项目目录下生成get文件夹。

.git // git版本控制的目录

git rm --cached 文件名 // 暂存区 -> 工作区

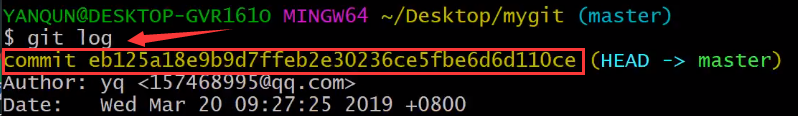
git commit // 进入vi操作界面，写提交备注，适合较长的备注

简写：git commit -m “备注内容”

1. 已提交到对象区（分支区），然后修改该已提交的文件，此时，这个文件会退回到工作区，如果还想提交的话，再经历一次操作：工作区 - > 暂存区 -> 对象区（分支区）。

# 5 git log

1. 查看提交日志，会发现如下内容：



commit后面的那一串字符：sha1计算的结果。

sha1、md5：加密算法、随机数，用于区分是那一次提交。

分布式ID生成器。

1. 自定义查看日志

git log -最近的提交次数

git log --pretty=oneline

git log --pretty=format:”%h -%an, %ar : %s”

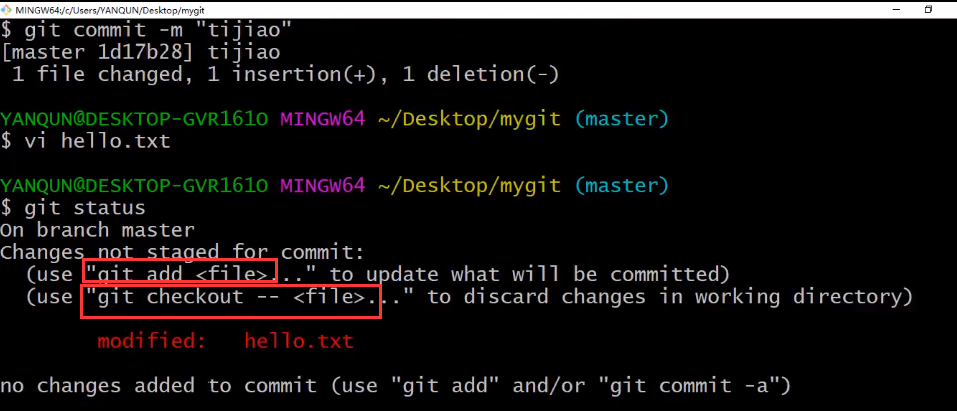
# 6 删除用户配置信息

1. 找到.gitconfig文件直接删掉。
2. 或通过命令删除

git config --global(或--system或--local) --unset user.name

git config --global(或--system或--local) --unset.email

# 7 已提交的文件-修改后-撤销回滚



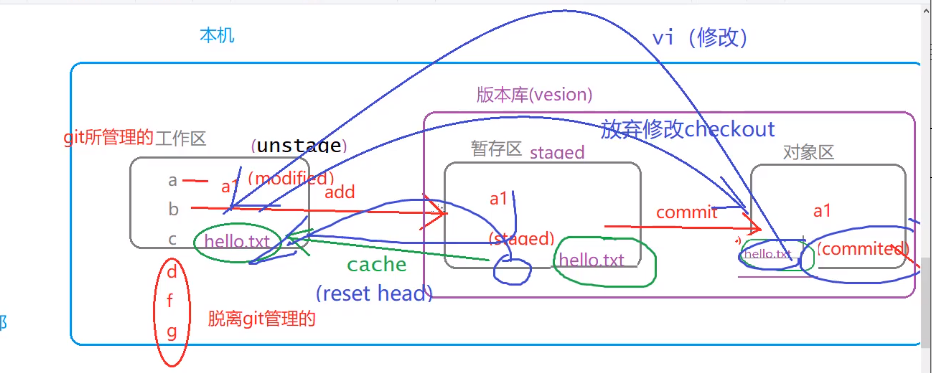
如果某个文件，已提交，并对其进行了修改，可以放弃修改（还原到已提交状态）

git checkout <file>

# 8 工作区文件 -> 暂存区后：撤销add操作

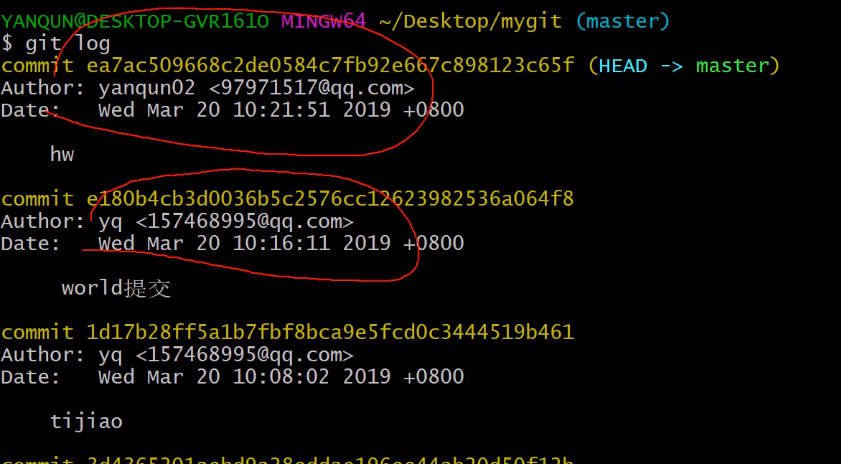
git reset HEAD <file>

小结



# 9 提交问题与用户邮箱配置

场景：第一个邮箱用户git提交了文件，第二个邮箱用户（把第一个邮箱用户改成第二个的）在同一个本地环境（跟第一个用户环境保持一致，就用第一个的项目，说白了就是改了个邮箱用户名而已，其他都没变）的项目中进行git提交操作（如新建了一个文件，修改了文件后提交到对象区）。此时，git log查看到，之前的配置没变，记录此次的配置。



# 10 删除已提交的文件（对象区/分支区）

1. git rm操作删除：删除 -> 暂存区

git rm <file>

删除之后的文件会被放到暂存区，如果想彻底删除，再输入git命令：

git commit -m “彻底删除”

1. 操作系统删除rm ：删除 -> 工作区

rm <file>

git add .

git commit -m “彻底删除”

# 11 撤销已提交文件的删除

对象区（分支区）-git rm <file> ->

暂存区-git reset HEAD<file> -> （注意：这里抛出了delete信息）

工作区-git checkout -- <file>

# 12 重命名文件

1. git文件的重命名

git mv <file1> <file2>

git status // renamed: <file1> -> <file2>

git reset HEAD <file1> // new file: <file2>, deleted: <file1>

git checkout -- <file1>

ls // <file1>、<file2>都在

git status // new file: <file2>, 可提交->对象区，也可撤销->工作区

1. 操作系统的重命名

mv <file1> <file2>

git status // deleted: <file1>, untracked files: <file2>

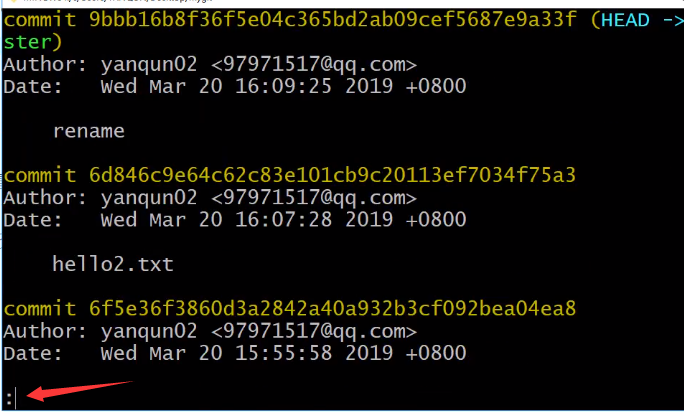
git add .

git commit -m “操作系统的重命名”

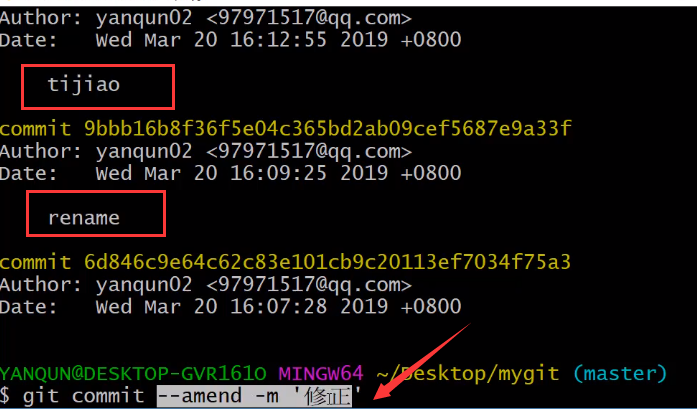
git status // 每次操作可以用此行命令查看文件所在哪个区需执行什么操作

# 13 注释重写提交说明

git log // 出现冒号，按 q 退出模式



修改最近一次注释：git commit --amend -m “注释内容”



# 14 忽略文件：.gitignore

在git项目根目录下创建.gitignore文件，编辑内容vi .gitignore。

a.properties // 列出忽略的文件即可

保存该.gitignore文件，接着，退出vi编辑模式，返回git命令行，输入git status查看文件状态，已经没有了a.properties文件。

# 15 通配符

使用场景：如.gitignore文件编写需要排除的文件。以下是常见的通配符：

1. \*：匹配任意字符；
2. \*.xxx：匹配后缀.xxx的文件；
3. !xx.xxx：在后缀.xxx的文件中，排除xx.xxx文件，结合\*.xxx使用；
4. dir(目录)/：忽略dir中的所有文件；
5. dir(目录)/\*/\*.xxx：能够忽略如dir/t1/a.xxx，dir/t2/b.xxx，但不能忽略含2级以上的目录中的\*.xxx文件（如：dir/a/b/j.xxx），这跟通配符 \* 有关。
6. dir/\*\*/\*.xxx：忽略任意级别目录中的\*.xxx文件。
7. git项目中创建的空目录，默认自动忽略：git status没有此目录信息。

# 16 分支

1. 查看分支：git branch
2. 创建分支：git branch <分支名>
3. 切换分支：git checkout <分支名>
4. 删除分支：git branch -d <分支名>
5. 自己不能删除自己（当前分支），可切换主分支master，再删除当前分支。
6. 如果要被删除的分支，包含“未合并”的内容，切换到master进行删除会报错。建议先合并再通过master删除。



1. 创建新分支，并且切换到该分支：git checkout -b <分支名>

结合 4. 进行操作：

// 场景：git项目已存在文件

git checkout -b new\_branch

git rm \*.xxx // 删除\*.xxx文件

git commit -m “delete files” // 在new\_branch分支中没有了\*.xxx文件

git checkout master // new\_branch删除的\*.xxx文件还在，此时没有受到new\_branch操作影响，注意到这，已经切换到了master

git branch -d new\_branch // 报错，没删掉new\_branch分支，并且提示new\_branch需要合并(merge)到master，才能通过master删掉new\_branch。

git merge new\_branch // new\_branch合并到master，注意观察new\_branch之前删除的\*.xxx在master中没有了。

git branch -d new\_branch // 成功删除分支new\_branch

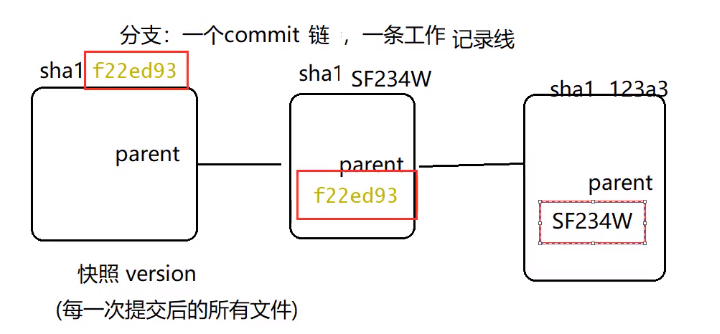
强行删除分支：git branch -D <分支名>

1. 查看分支最近提交(commit)的sha1值：git branch -v

**细节问题：**

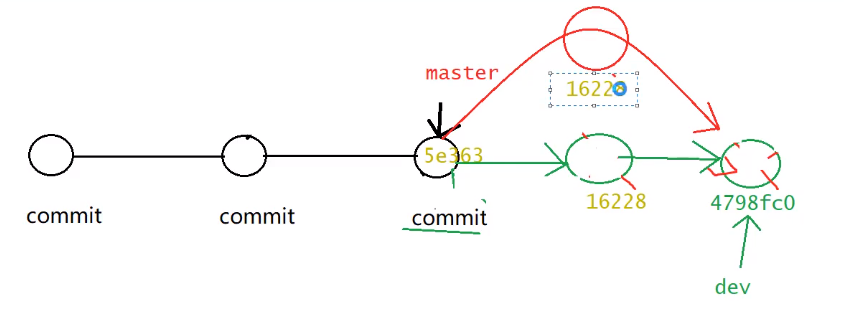
1. 如果在非主分支中，file1文件进行了写操作（即增删改），但此操作局限在工作区中进行（没有add，commit）。在master中能够看到该操作。
2. 如果在非主分支中，file1文件进行了写操作（即增删改），并且add，commit。在master中无法观察该操作，看不到file1文件。
3. 如果在非主分支中，file1文件进行了写操作（即增删改），但此操作局限在工作区中进行（没有add，commit）。在master中删除该非主分支，可直接删除。
4. 如果在非主分支中，file1文件进行了写操作（即增删改），并且add，commit。在master中无法删除该非主分支。

# 17 分支-提交链（分支合并与冲突）



分支名（master）：指向当前的提交（commit）；

HEAD:指向当前分支（HEAD -> 分支名。



// 场景：git空项目，分支合并

git init

git checkout -b dev // 当前分支dev，指向：HEAD -> dev

cd .git

ls // HEAD位置：.git -> HEAD

cat HEAD // ref: refs/heads/dev

cd ..

cd ..

git log // 查看sha1值 HEAD -> dev, master

vi a.txt //编辑内容：a，保存，退出vi模式，返回git命令

git add .

git commit -m “dev1”

git log // 观察dev的sha1值 -> dev1，此时，分支dev领先master一步

vi a.txt // 修改内容

git add .

git commit -m “dev2”

git log // 观察dev的sha1值 -> dev2，此时，分支dev已经领先master两步了

git checkout master // 观察master的sha1值，没有变，此时master落后dev两步

get merge dev // master中合并dev分支，注意：从初始状态跨越到第二步状态，其中第一步直接被覆盖了，并没有忽略第一步。

git log // 观察master的sha1值已经赶上dev第二次commit -> dev（称为fast forward，本质就是fast forward分支指针的移动。注意：跳过的中间commit，仍然会保存的）

git log --graph // 查看fast forward模式下的图形展示提交信息

/\* fast forward：

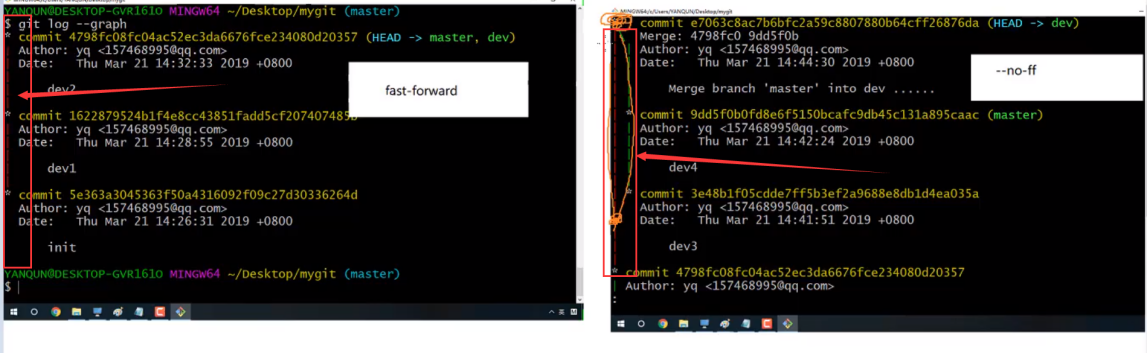
1. 两个分支fast forward归于一点commit
2. 没有分支信息（丢失分支信息）

git在merge时，默认使用fast forward；当然，也可以设置禁止：

git merge --no-ff ，禁止后会有如下情况：

1. 两个分支fast forward 不归于一点，主动发起合并的分支会前进一步。

2. 分支信息完整（保留分支信息）。



--no-ff：左列有竖着的虚线

fast-forward：左列没有虚线

\*/

vi a.txt // 修改内容

git add .

git commit -m “master3”

vi a.txt //修改内容

git add .

git commit -m “master4”

git checkout dev

git merge --no-ff master // 分支dev主动向master发起合并，会跳转类似commit注释信息，这里是分支合并注释

git log // 观察主动发起合并的分支dev向前领先了一步

git log --graph // 查看--no-ff模式下的图形模式提交信息

git checkout master

git merge dev // master又赶上dev的sha1值状态

// 合并：制造冲突。 接上面git命令，往下继续

ls

vi a.txt //修改第二行内容：hello master

git add .

git commit -m “master5”

git checkout dev

ls

vi a.txt // 也修改第二行内容：hello dev ，此时，2个分支都改了相同的行，冲突产生了

git add .

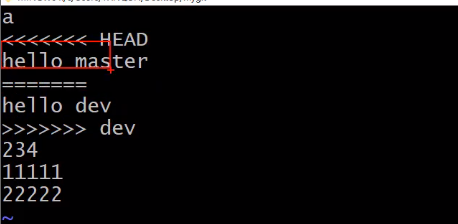
git commit -m “dev6”

git checkout master

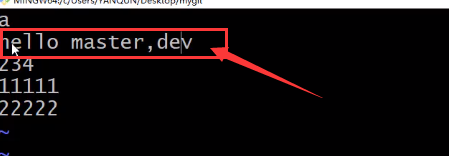
git merge dev // 报错：comment冲突，并提示冲突文件路径位置

// 合并：解决冲突。接上面git命令，往下继续

vi a.txt // vi a.txt，修改内容，再次提交



// 修改内容如下：删掉<<<<<< HEAD、======、>>>>>> dev，然后随便改第二行。



git status

git add . // 此时的add，提示冲突已解决

git commit -m “maste5,dev6解决冲突”

git log // 观察master的sha1值

// 注意：master在merge时，如果遇到冲突，并解决，则解决冲突会进行2次commit，1次是将对方dev的提交信息也拿过来了，接着1次是最终提交。

git checkout dev

cat a.txt // master的a.txt值改了，这里的值没变

git merge master // 成功合并

/\* 注意：

1. 2个分支中，如果一方落后，另一方领先，则落后方可以直接通过merge合并到领先方。
2. 如果2个分支在同一提交点，即sha1值相同，修改任一方分支的文件内容，再合并分支就会有冲突。

\*/

查看日志技巧：

git log --graph --pretty=online --abbrev-commit

# 18 版本回退