学习笔记之 — Git版本库搭建和使用

袁肖

目录

[一、 Git简介 3](#_Toc404860173)

[1.1、 Git是什么？ 3](#_Toc404860174)

[1.2、 集中式 VS 分布式 3](#_Toc404860175)

[二、 安装Git 3](#_Toc404860176)

[2.1、 Linux（Ubuntu）安装 3](#_Toc404860177)

[2.2、 Windows安装 4](#_Toc404860178)

[2.3、 配置Git全局信息 4](#_Toc404860179)

[三、 创建版本库 4](#_Toc404860180)

[3.1、 创建Git远端版本库（Linux系统） 4](#_Toc404860181)

[3.2、 创建Git本地版本库 4](#_Toc404860182)

[3.1.1） Windows系统下 4](#_Toc404860183)

[3.1.2） Linux系统下 4](#_Toc404860184)

[3.3、 添加文件到本地版本库 4](#_Toc404860185)

[四、 时光穿梭机 5](#_Toc404860186)

[4.1、 版本回退 5](#_Toc404860187)

[4.2、 工作区和暂存区 6](#_Toc404860188)

[4.3、 管理修改 7](#_Toc404860189)

[4.3.1） 提交修改 7](#_Toc404860190)

[4.3.2） 撤销修改 7](#_Toc404860191)

[4.3.3） 删除文件 8](#_Toc404860192)

[五、 远端版本库 8](#_Toc404860193)

[5.1、 添加远端版本库 8](#_Toc404860194)

[5.2、 关联远端版本库 8](#_Toc404860195)

[5.3、 克隆远端版本库 9](#_Toc404860196)

[六、 分支管理 9](#_Toc404860197)

[6.1、 创建与合并分支 9](#_Toc404860198)

[6.2、 解决冲突 12](#_Toc404860199)

[6.3、 分支管理策略 13](#_Toc404860200)

[6.4、 BUG分支 14](#_Toc404860201)

[6.5、 Feature分支 15](#_Toc404860202)

[6.6、 多人协作 15](#_Toc404860203)

[6.6.1） 推送分支 15](#_Toc404860204)

[6.6.2） 抓取分支 16](#_Toc404860205)

[七、 标签管理 16](#_Toc404860206)

[7.1、 创建标签 17](#_Toc404860207)

[7.2、 操作标签 17](#_Toc404860208)

[八、 自定义Git 17](#_Toc404860209)

[8.1、 修改全局变量 17](#_Toc404860210)

[8.2、 忽略特殊文件 18](#_Toc404860211)

[8.3、 配置别名 18](#_Toc404860212)

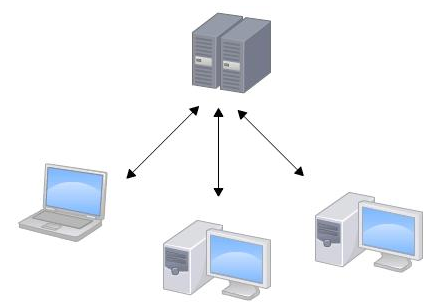
1. Git简介
2. Git是什么？

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

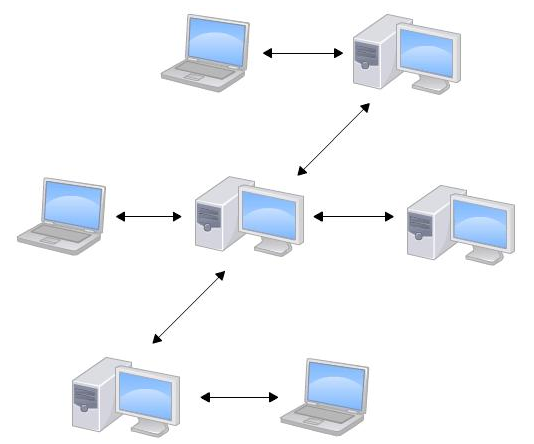
Git有什么特点？简单来说就是：高端大气上档次！

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！

1. 集中式 VS 分布式
2. 集中式版本控制系统，版本库是集中存放在中央服务器的。



1. 分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，安全性高。



1. 安装Git
2. Linux（Ubuntu）安装

打开终端（Ctrl + Alt + T），输入以下命令：

sudo apt-get install git-core openssh-server openssh-client

sudo apt-get install git-core git-gui git-doc

sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext libz-dev git-core

1. Windows安装

安装msysgit + TortoiseGit，例如，安装在C:\Program Files\Git目录下，

将路径C:\Program Files\Git\bin将入系统环境变量Path中，

打开终端，运行->cmd，输入git --version可以查看git版本。

1. 配置Git全局信息

#设置与中文显示有关（SCRT中用到）

git config --global core.quotepath false

# 设定用户名（例如yuanxiao，注：建议使用姓名全称的拼音）

git config --global user.name yuanxiao

# 设定用户邮箱（例如yuan.xiao5@zte.com.cn）

git config --global user.email yuan.xiao5@zte.com.cn

# 查看配置信息

git config --list

1. 创建版本库
2. 创建Git远端版本库（Linux系统）

# 创建远端版本库目录，比如/home/yuanxiao10164257/

cd /home/yuanxiao10164257/

sudo git --bare init

1. 创建Git本地版本库
2. Windows系统下

参考Linux系统下的本地版本库创建。

1. Linux系统下

# 创建文件放置目录，比如/home/

cd /home

mkdir git

cd git

# 初始化Git本地版本库

git init

1. 添加文件到本地版本库

# 将修改的文件加入到本地版本库的暂存区（index或stage）

git add readme.txt ... ....

# 提交改动至本地版本库分支master（-m "type in change log"必须有）

git commit -m "type in change log"

# 查看历史提交日志，从最近之最远

git log

1. 时光穿梭机
2. 版本回退

# 查看工作区文件修改状态（强烈推荐使用）

git status

# 查看文件修改内容

git diff readme.txt

# 从本地版本库中删除文件

git rm readme.txt

# 查看工作区和本地版本库里面最新版本的区别

git diff HEAD -- readme.txt

# git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD

时，表示最新的版本

# 1. 回退版本，在Git中，用HEAD表示当前版本，上一个版本就是HEAD^，上上一

个版本就是HEAD^^，往上100个版本写成HEAD~100。commit\_id（版本号）。

git reset --hard HEAD^

git reset --hard commit\_id

# 2. 把暂存区（index或stage）的修改撤销掉（unstage），重新放回工作区

git reset HEAD readme.txt

# 用来记录你的每一次命令

git reflog

# HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用

# 命令git reset --hard commit\_id。

# 穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。

# 要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

# git log和git reflog可用q退出

1. 工作区和暂存区

**工作区**（Working Directory）：就是你在电脑里能看到的目录，例如/home/git/yuanxiao。

**版本库**（Repository）：工作区有一个隐藏目录.git，这个是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的**暂存区**。

Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

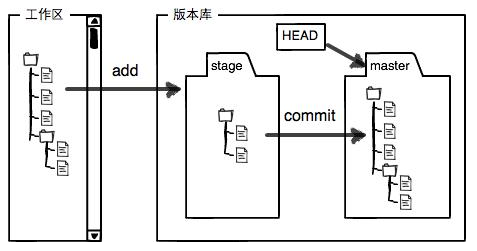
第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

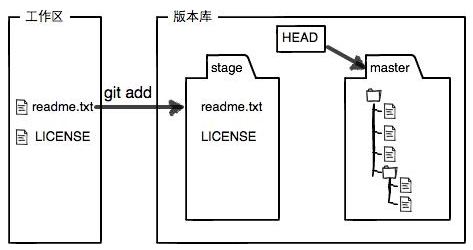
第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

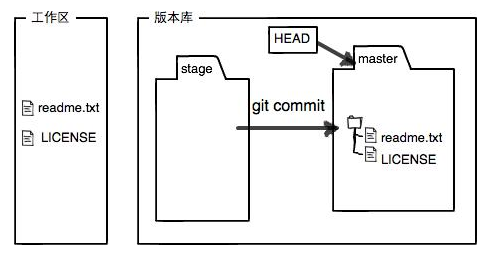
因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，

git commit就是往master分支上提交更改。你可以简单理解为，需要提交的文件修改

通通放到暂存区，然后，一次性提交暂存区的所有修改。







1. 管理修改
2. 提交修改

# 查看工作区文件修改状态（强烈推荐使用）

git status

# 查看文件修改内容

git diff readme.txt

# 查看工作区和本地版本库里面最新版本的区别

git diff HEAD -- readme.txt

# 第一次修改 -> git add -> git commit

# 第一次修改 -> git add -> 第二次修改 -> git add -> git commit

1. 撤销修改

# 把readme.txt文件在工作区的修改全部撤销

# 这里有两种情况：

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

# 一种是readme.txt已经添加到暂存区后，又作了修改，现在，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。

git checkout -- readme.txt

# 场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

# 场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

# 场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

# git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本

# 1. 回退版本，在git中，用HEAD表示当前版本，上一个版本就是HEAD^，上上一个版本就是HEAD^^，往上100个版本写成HEAD~100。

git reset --hard HEAD^

git reset --hard commit\_id

# 2. 把暂存区（index或stage）的修改撤销掉（unstage），重新放回工作区

git reset HEAD readme.txt

1. 删除文件

# 从版本库中删除文件

git rm readme.txt

# 命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

1. 远端版本库
2. 添加远端版本库

# 创建远端版本库目录，比如/home/yuanxiao10164257/

cd /home/yuanxiao10164257/

sudo git --bare init

1. 关联远端版本库

# 关联一个远端版本库

git remote add origin **root@10.63.152.144:/home/ZX\_CSS**

# 更新你的本地版本库至远端版本库的最新改动

git pull(在Windows上等价于使用git fetch + git merge origin master)

# 将本地的改动推送到远端版本库

git push -u origin master第一次推送master分支的所有内容

git push origin master

# 要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git

# 关联后，使用命令git push -u origin master第一次推送master分支的所有内容

# 此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最

# 新修改

1. 克隆远端版本库

# 要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆

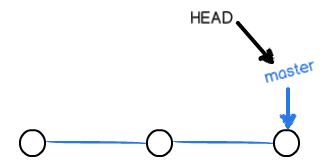
# Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快

# 下载远程仓库

git clone **root@10.63.152.144:/home/ZX\_CSS**

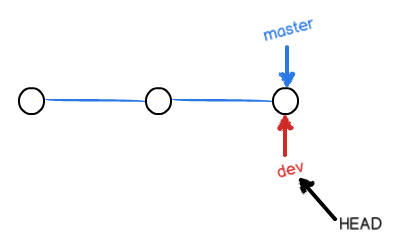
1. 分支管理
2. 创建与合并分支

# 一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指#向master，就能确定当前分支，以及当前分支的提交点，每次提交，master分支都会# 向前移动一步，这样，随着你不断提交，master分支的线也越来越长



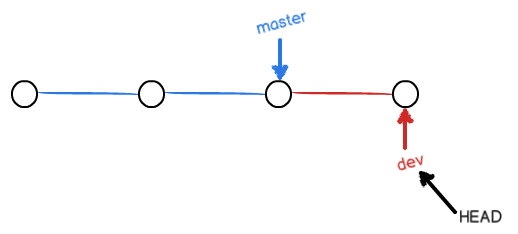
# 当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的

# 提交，再把HEAD指向dev，就表示当前分支在dev上



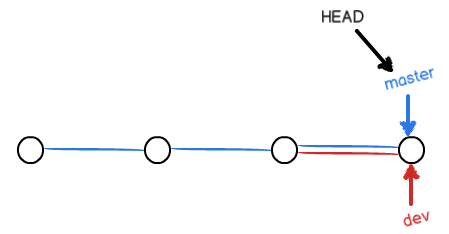
# 从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev

# 指针往前移动一步，而master指针不变



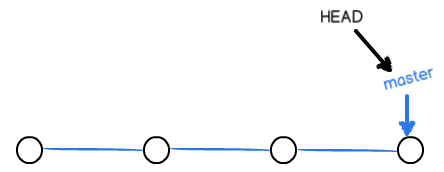
# 假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？

# 最简单的方法，就是直接把master指向dev的当前提交，就完成了合并



# 合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，

# 删掉后，就剩下了一条master分支



# 创建并切换到分支new\_branch\_dev

git checkout -b new\_branch\_dev

等价于git branch new\_branch\_dev + git checkout new\_branch\_dev

# 列出所有分支，当前分支前面会标一个\*号

git branch

# 切换回master分支

git checkout master

# 合并指定分支到当前分支

# 注意Fast-forward信息，git告诉我们，这次合并是“快进模式”，

# 也就是直接把master指向new\_branch\_dev的当前提交，所以合并速度非常快。

git merge new\_branch\_dev

# --no-ff参数，表示禁用Fast forward

# 强制禁用Fast forward模式，git就会在merge时生成一个新的commit，这样，从分

支历史上就可以看出分支信息。

# 合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出

来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

git merge --no-ff -m "merge with no-ff" new\_branch\_dev

# 删除new\_branch\_dev分支

git branch -d new\_branch\_dev

# git用<<<<<<< HEAD 内容 ======= 内容 >>>>>>>new\_branch\_dev标记出不同分

支的内容

# 查看分支合并图

git log --graph

# Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

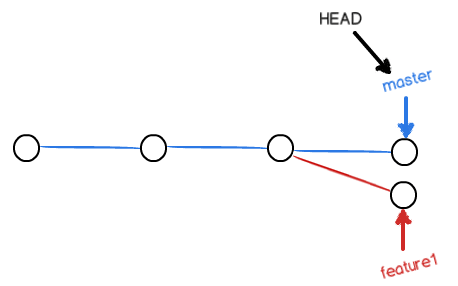
合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

1. 解决冲突

# 准备新的feature1分支，继续我们的新分支开发

# master分支和feature1分支各自都分别有新的提交



# 切换回master分支

git checkout master

# 合并指定分支到当前分支，发生冲突

git merge feature1

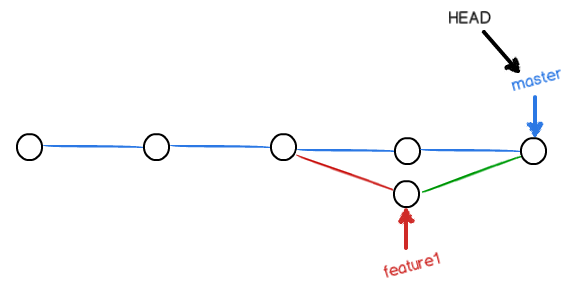
# git用<<<<<<< HEAD 内容 ======= 内容 >>>>>>> feature1标记出不同分支的内容

并修改冲突

# 当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成

# 查看分支合并图

git log --graph



# 删除feature1分支

git branch -d feature1

1. 分支管理策略

# 在实际开发中，我们应该按照几个基本原则进行分支管理：

#首先，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。那在哪干活呢？

# 干活都在dev分支上，也就是说，new\_branch\_dev分支是不稳定的，到某个时候，比

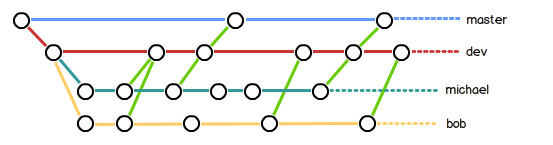
如1.0版本发布时，再把new\_branch\_dev分支合并到master上，在master分支发布1.0

版本

# 你和你的小伙伴们每个人都在new\_branch\_dev分支上干活，每个人都有自己的分支，

时不时地往new\_branch\_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



# Git分支十分强大，在团队开发中应该充分应用。

# 合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出

来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

1. BUG分支

# 软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是

如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，

然后将临时分支删除。

# 当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支

issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交

# Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继

续工作

# 把当前工作现场“储藏”起来，等以后恢复现场后继续工作

git stash

# 查看stash队列，确定要恢复哪个工作现场到当前分支

git stash list

# 工作现场还在，git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

# 1. 恢复后，stash内容并不删除，你需要用git stash drop来删除，num 就是你要恢复

的工作现场的编号

# 注意使用git stash apply是恢复stash队列中的stash@{0}即最上层的那个工作现场

git stash apply

git stash apply stash@{num}

# 2. 恢复的同时把stash内容也删了

# 注意使用git stash pop命令是恢复stash队列中的stash@{0}即最上层的那个工作现场

git stash pop

git stash pop stash@{num}

# 清空stash队列

git stash clear

1. Feature分支

# 软件开发中，总有无穷无尽的新的功能要不断添加进来。

# 添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，

每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，

删除该feature分支。

# 新建新的分支

git checkout -b feature

# 切回dev，准备合并

git checkout new\_branch\_dev

# 就在此时，接到上级命令，因经费不足，新功能必须取消

# 如果要丢弃一个没有被合并过的分支

git branch -D new\_branch\_dev

# 开发一个新feature，最好新建一个分支；

# 如果要丢弃一个没有被合并过的分支，可以通过git branch -D <name>强行删除。

1. 多人协作

# 当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支

对应起来了，并且，远程仓库的默认名称是origin。

# 要查看远程库的信息，-v显示更详细的信息

git remote

git remote –v

1. 推送分支

# 推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，git就会把该分支推送到远程库对应的远程分支上

# master分支是主分支，因此要时刻与远程同步；

# new\_branch\_dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；

# bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug

# feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

git push origin master

git push origin new\_branch\_dev

1. 抓取分支

# 多人协作时，大家都会往master和dev分支上推送各自的修改

# 创建远程origin的new\_branch\_dev分支到本地

git checkout -b new\_branch\_dev origin/new\_branch\_dev

# 指定本地new\_branch\_dev分支与远程origin/new\_branch\_dev分支的链接

git branch --set-upstream new\_branch\_dev origin/new\_branch\_dev

# 多人协作的工作模式：

首先，可以试图用git push origin branch-name推送自己的修改；

如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；

如果合并有冲突，则解决冲突，并在本地提交；

没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！

如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream branch-name origin/branch-name

# 多人协作的工作模式

# 查看远程库信息，使用git remote -v；

# 本地新建的分支如果不推送到远程，对其他人就是不可见的；

# 从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；

# 在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；

# 建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；

# 从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

1. 标签管理

# 发布一个版本时，我们通常先在版本库中打一个标签，这样，就唯一确定了打标签时

刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版

本取出来。所以，标签也是版本库的一个快照。

# Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像

对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

1. 创建标签

# 首先，切换到需要打标签的分支上

git branch

git checkout master

# 打一个新标签v1.0，默认标签是打在最新提交的commit上的

git tag v1.0

git tag v1.0 commit\_id

# 查看标签

git tag

# 查看标签信息

git show <tagname>

# 创建带有说明的标签，用-a指定标签名，-m指定说明文字

git tag -a v0.1 -m "version 0.1 released" commit\_id

# 通过-s用私钥签名一个标签

git tag -s v0.2 -m "signed version 0.2 released" commit\_id

1. 操作标签

# 删除标签

git tag -d v0.1

# 如果要推送某个标签到远程

git push origin <tagname>

# 一次性推送全部尚未推送到远程的本地标签

git push origin --tags

# 删除远程标签

git push origin :refs/tags/v0.9

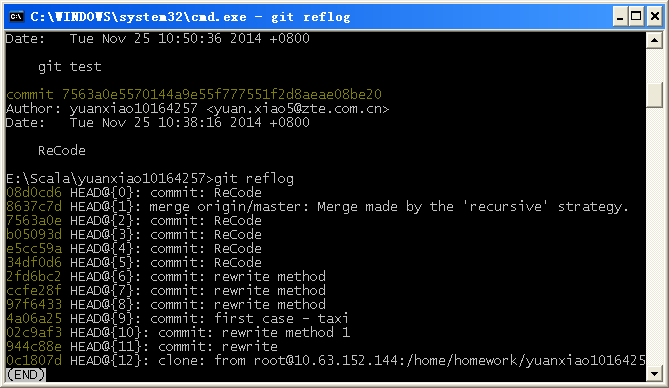
1. 自定义Git
2. 修改全局变量

# 在安装Git一节中，我们已经配置了user.name和user.email，实际上，Git还有很多

可配置项

# 让git显示颜色，让命令输出看起来更醒目，可用git log/ git status等命令查看

git config --global color.ui true



1. 忽略特殊文件

# 在git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进

去，git就会自动忽略这些文件

# 忽略文件的原则是：

忽略操作系统自动生成的文件，比如缩略图等；

忽略编译生成的中间文件. 可执行文件等，

也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放

进版本库，比如Java编译产生的.class文件；

忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

# 忽略某些文件时，需要编写.gitignore；

# .gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理！

1. 配置别名

# 配置别名，st就表示status

git config --global alias.st status

# 显示最后一次提交信息

git log -1

# 配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前

的仓库起作用。配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中。

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。