## git config —— 配置git环境和提交者的信息

* git config [--system –-global --local] 配置项

--system 此电脑的所有用户的所有仓库

--global 当前用户的所有仓库

--local 当前用户的当前仓库

多次重复配置的话，以最低配置有效，即--local会屏蔽--global，--global会屏蔽--system，就像同名的局部变量屏蔽全局变量那样。

以配置机器的某一用户的所有仓库举例：

1. git config --global user.name "Leif Lee" //配置提交者姓名

2. git config --global user.email [print\_sync@qq.com](mailto:print_sync@qq.com) //配置提交者邮箱

3. git config –-global alias.init it //配置命令别名 init --- it

* 查看配置信息

git config -–system –-list

git config --global --list

git config --local list

* 三种级别的配置文位置，可通过直接编辑配置文件去config

git config --system –e //编辑系统的配置文件

git config --global –e

git config --local -e

D:\Program Files\Git\etc\gitconfig //系统的config文件位置

C:\Users\Leif\.gitconfig //用户的config文件位置

D:\TestGit\.git\config //仓库的config文件位置

## 添加文件

git add file.c

作用

1. 已经追踪的文件，从工作区转存到暂存区
2. 未追踪的文件加入git追踪范围

## 文件提交

1. 把暂存区的修改提交到版本库

git commit

git commit –m “message”

1. 把工作区和暂存区的所有修改上传到仓库，如果某个文件存在于暂存区和工作区，且内容不一致，以工作区为最终版本上传到仓库

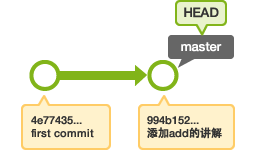
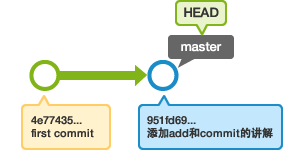
git commit –a

git commit –am “message”

git commit –a

1. 修改最后一次的commit

git commit - -amend

最后一个版本是B，上一版本是A，修改文件然后使用该命令，生成版本C，但是C替代B

最终结果是A-🡪C，而不是A-🡪B-🡪C

不修改文件直接使用该命令，是修改最后一次版本的提交的提示信息

## 重命名文件

git mv 原文件 新文件

1. 如果原文件是版本库的文件，执行该命令后，存于暂存区
2. 如果原文件是工作区的文件，执行该命令后，存于工作区

## 忽略文件

让某些文件永远不会提交到版本库，适合工作区的临时文件，调试文件，日志等

vim .gitignore

\*.o

\*.a

\*.bin

在工作目录下创建.gitignore，后缀是.o .a .bin文件将一直在工作区，但永远不会被转存到暂存区和版本库

不想忽略某个后缀是.bin的文件，例mian.bin，在.gitignore文件中加 ！main.bin

忽略某个目录 /debug

忽略目录下的某个文件 /debug/text

忽略目录下的某个类型的文件 /debug/\*.bin

## 差异比较

比较工作区和暂存区的差别，显示的工作区文件的变化

git diff

比较暂存区和版本库的差别

git diff –staged

git diff –staged SHA

比较工作区和版本库

git diff HEAD

git diff SHA

比较两个版本库的差别

git diff SHA1 SHA2

## 分支管理

分支名指向该分支最新版本，多个分支就有多个分支名。

head只有一个，只能指向当前分支的分支名，切换分支时，head也改变指向。

git branch 新分支名 //创建新分支但是不切换到新分支

git branch 新分支名 旧分支名（哈希值） //在指定版本上创建新分支

git checkout 分支 //切换到某个分支

git branch 查看当前所处的分支

git branch -a 查看所有的分支

git branch -m 旧名 新名 //修改分支的名字

git branch –b 分支名 //删除分支

切换分支之前确保工作区和暂存区是干净的

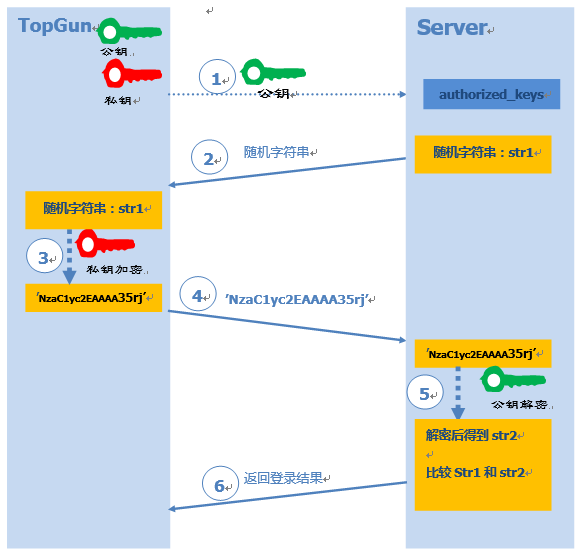
## 远程仓库

<http://www.cnblogs.com/dzblog/p/6930147.html>

SSH协议：加密安全，不能匿名访问，必须授权才能读或写仓库，适合私密项目。

HTTP协议：提供只读的仓库。

SSH协议，免密登陆示意图如下：



公钥认证流程：

1. Client端用户生成公钥和私钥，把公钥拷贝到Server上。

2. Server收到登录请求后，随机生成一个字符串str1，并发送给Client。

3. Client用自己的私钥对字符串str1进行加密。

4. 将加密后字符串发送给Server。

5. Server用之前存储的公钥进行解密，比较解密后的str2和str1。

6. 根据比较结果，返回客户端登陆结果

名词解释：

/home/Leif/.ssh

/home/Leif/.ssh/ id\_rsa 保存私钥

/home/Leif/.ssh/ id\_rsa.pub 保存公钥

/home/Leif/.ssh/ known\_hosts 保存已认证的远程主机公钥

/home/Leif/.ssh/authorized\_hosts 存储已经认证的客户端的公钥，在里面添加主机的公钥，即可被访问。

ssh-keygen是用于生产密钥的工具，ssh-keygen -t rsa -C "你的email地址"，-C设置注释文字，比如邮箱，-f指定密钥文件存储文件名。

一台主机可能既是Client，也是Server。所以会同时拥有authorized\_keys和known\_hosts。



生成密钥操作

1. 打开.ssh目录，一般是隐藏的，无该目录自己创建 cd ~/.ssh

2. 生成密钥文件,默认名字为id\_rsa和id\_rsa.pub

ssh-keygen -t rsa -C "YOUR\_EMAIL@YOUREMAIL.COM"

(说明：YOUR\_EMAIL@YOUREMAIL.COM换成你自己的邮箱)

3. 打开公钥文件（id\_rsa.pub），并把内容复制至代码托管平台上 或者 需要访问的服务器(host)上的authorized\_keys里。

vim ~/.ssh/id\_rsa.pub

a.代码托管平台

进入 GitLab-->账户设置-->Profile Settings --> SSH Keys，点击Add keys 按钮，在 Title 文本框中输入任意字符，在 Key 文本框粘贴刚才复制的公钥字符串，按保存按钮完成操作。

b.目标服务器

cd ~/.ssh/

vim authorized\_keys

4. 测试

ssh -T [git@gitlab.com](mailto:git@gitlab.com) 或者 ssh -T user@host