**基础**

**git config：配置相关信息**

**git clone：复制仓库**

**git init：初始化仓库**

**git add：添加更新内容到索引中**

**git diff：比较内容**

**git status：获取当前项目状况**

**git commit：提交**

**git branch：分支相关**

**git checkout：切换分支**

**git merge：合并分支**

**git reset：恢复版本**

**git log：查看日志**

**git help**

git help –a

**git初始化**

当我们安装了git, 需要配置一些环境变量, 环境变量分为三个级别

系统级别

用户级别

项目级别

> 一般情况下我们配置最多的是 `用户级别`

用户名

git config --global user.name "Magina BMP"

用户邮箱

git config --global user.email "yin32167@aliyun.com"

列出自己的环境变量

git config --global list

> 当我们配置了这些环境变量之后, 除了通过 `git config --global list` 查看之外, 还可以通过文件查看

# mac/linux

vi ~/.gitconfig

# windows

打开 C:\Users\Magina\.gitconfig (其中Magina为我的用户名)

**git config**

git config --global user.name ‘XXX’

git config --global user.email ‘XXX’

git config --unset --global user.name

git config list

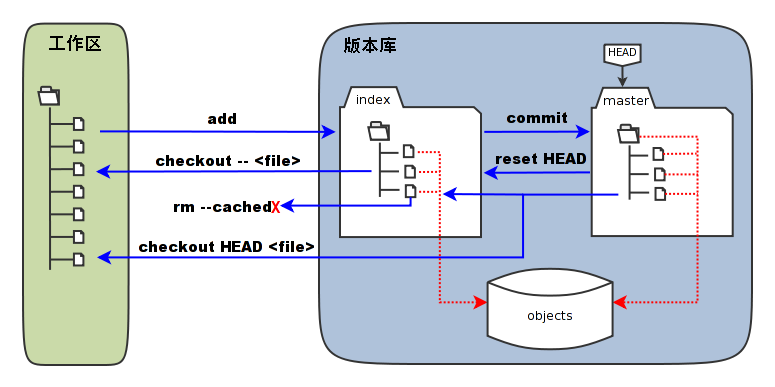
**git工作区与版本库**

**[工作区与版本库关系]**

**> index/stage: 暂存区**

**objects: 对象库**

**master: 主分支**

******

**git init**

**git add**

**git add . //.号的意思是当前目录，即将当前目录所有文件都添加进来。**

git add \***将修改内容添加到本地缓存区，通配符可以把当前目录下所有修改的新增的文件都自动添加：**

$ git add \*

**git commit -am ""**

**对于一般的 git 提交, 通常我们会这样做:**

git add .

git commit -m "添加了 a.html"

**有一种比较简单的写法可以合并上面的两种**

git commit -am "添加了 a.html"

**但这种写法存在问题, 它并不能将新创建的文件提交, 比如这样**

touch b.html

git commit -am "添加了 b.html"

**此时会报错, 因为这种写法无法将 b.html 添加到暂存区, 所以你还是需要通过下面命令提交**

git add b.html

git commit -m "添加了 b.html"

**git commit**

**git commit -m “XXXX” //提交后的日志记录为XXXX**

git status

**日志统计**

**如果用--stat选项使用'git log',它会显示在每个提交(commit)中哪些文件被修改了, 这些文件分别添加或删除了多少行内容，这个命令相当于打印详细的提交记录：**

**$ git log --stat**

git log --🡪 git log --oneline --decorate --all --graph

git log --oneline //日志简略显示

**--graph 选项可以可视化你的提交图(commit graph)，会用ASCII字符来画出一个很漂亮的提交历史(commit history)线：**

$ git log --graph --oneline

**git diff**

**git mv //在暂存区里修改文件名称，mv用来在还未提交时的工作区里修改文件名称**

**git rm 与git rm --cached**

**当我们需要删除`暂存区`或`分支`上的文件, 同时工作区也不需要这个文件了, 可以使用**

**git rm file\_path**

**当我们需要删除`暂存区`或`分支`上的文件, 但本地又需要使用, 只是不希望这个文件被版本控制, 可以使用**

**git rm --cached file\_path**

**> file\_path 为文件路径**

**git rm //删除暂存区里的文件，不删除本地**

**//用于工作区的文件已经添加到版本库，如果没有添加，这个操作是没有效果的。**

**//如果已经添加到版本库，就尽量不用 rm和mv这样的命令，会多几个操作。**

**git rm –cached //删除记录log，不会影响本地文件和版本库文件**

**git rm –f //强制从暂存区和本地一起删除**

**git revert 与git reset**

**首先介绍一下 HEAD 指针**

**当前的版本 HEAD**

**上一个版本 HEAD^**

**上两个版本 HEAD^^**

**上三个版本 HEAD^^**

**上多个版本 HEAD~n**

**git revert**

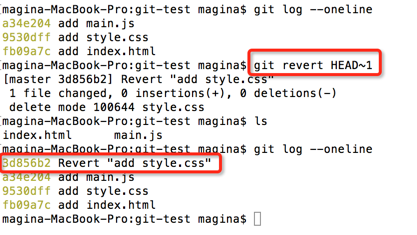
**---**

**有时候我们修改了文件, 并且提交了, 但后来发现, 之前做的是对的, 不需要修改, 这个时候我们就需要撤回那次提交.**

**> 我们先通过git log --oneline 去查看提交的记录**



**> 假如我们不想提交style.css, 可以使用 `git revert HEAD~1` 撤销这次提交**



**可以发现, 它会将这条撤回消息的动作也写入提交记录, 并且这次撤回除了影响暂存区和分支, \*\*还会修改工作区\*\***

**git reset**

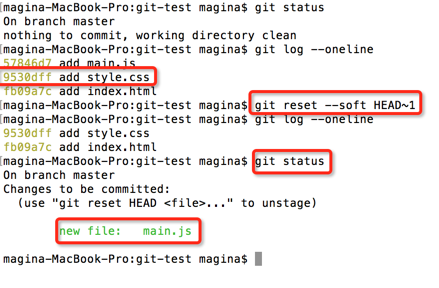
**---**

**如果我们想回滚到前面的版本, 比如说添加style.css那个时候, 可以使用git reset, 其中它有三个参数**

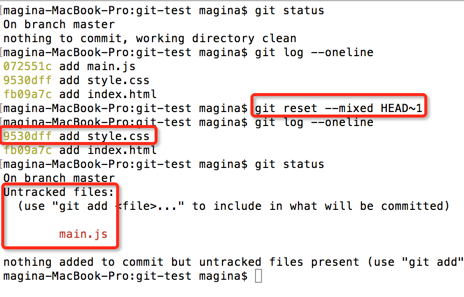
**soft 只改变分支的提交**

**mixed 改变分支和暂存区, 不影响工作区**

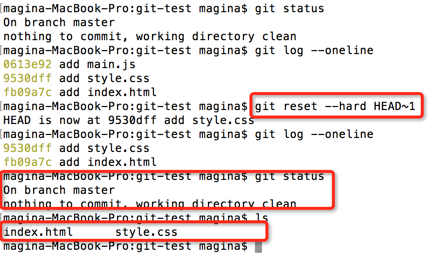
**hard 分支,暂存区和工作区都会影响, 这个操作需要慎重**



**> 软删除只会移除分支上的提交, 在其之后的修改都会撤回`暂存区`**



**> 混合删除会将其之后的修改全部撤回工作区, 不会影响工作区**



**> 硬删除会将其之后的修改全部撤回, 并且会影响到工作区**

**git revert//回撤操作，**

**git revert HEAD //回撤最后一条操作。**

**git revert HEAD^^ //回撤上两个版本**

**git revert HEAD~n //回撤上n个版本**

**git reset //不会有日志记录操作，去掉提交记录**

**git reset HEAD^ --soft //将分支中的文件撤销到暂存区里**

**git reset HEAD^ --mixed //mixed 为reset默认参数,操作不仅将文件从分支撤回，还将文件从//暂存区内撤回。**

**git reset HEAD^ --hard //将文件硬删除。从工作区，暂存区，分支中全部删除。**

**git reset XXXXX :重置到某个版本号**

**分支：**

**git branch //查看当前分支**

**git branch tag1//创建一个名叫tag1的分支**

**git checkout tag1//切换到tag1分支**

**git diff mastert..tag1**

**git merge tag1 ————Fast-Forward**

**git branch -m oldname newname**

**git branch -d tag1**

**git diff命令的作用是比较修改的或提交的文件内容。**

$ git diff

diff --git a/README.md b/README.md

index 21781dd..410e719 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,3 @@

gitproject

==========

+new line

**上面的命令执行后需要使用q退出。命令输出当前工作目录中修改的内容，并不包含新加文件，请注意这些内容还没有添加到本地缓存区。**

**可以使用 git diff 命令再加上 --cached 参数，看看缓存区中哪些文件被修改了。进入到git diff --cached界面后需要输入q才可以退出：**

**$ git diff --cached**

**如果没有--cached参数，git diff 会显示当前你所有已做的但没有加入到索引里的修改。**

**git diff // 查看变化，**

**//**-menglei insert it 删除一行

+menglei insert+dev2 it 添加一行

**如何查看缓存区内与上次提交之间的差别呢？需要使用--cached参数：**

$ git diff --cached

diff --git a/README.md b/README.md

index 21781dd..410e719 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,3 @@

gitproject

==========

+new line

diff --git a/file1 b/file1

new file mode 100644

index 0000000..fa49b07

--- /dev/null

+++ b/file1

@@ -0,0 +1 @@

+new file

**可以看到输出中已经包含了新加文件的内容，因为file1已经添加到了缓存区。**

**可以用 git diff 来比较项目中任意两个分支的差异。**

**我们查看test分支和master之间的差别：**

**$ git diff master test**

diff --git a/file1 b/file1

index fa49b07..17059cd 100644

--- a/file1

+++ b/file1

@@ -1 +1,2 @@

new file

+branch test

diff --git a/file2 b/file2

new file mode 100644

index 0000000..80e7991

--- /dev/null

+++ b/file2

@@ -0,0 +1 @@

+new file2

**git diff 是一个难以置信的有用的工具，可以找出你项目上任意两个提交点间的差异。可以使用git help diff详细查看其他参数和功能。**

**git diff master tag1..tag2 //比较两个分支的异同**

**--stat 参数可以统计一下有哪些文件被改动，有多少行被改动：**

$ git diff test --stat

file1 | 1 -

file2 | 1 -

2 files changed, 2 deletions(-)

**分支合并：**

**git merge tag1 -m “XXX”**

**{**

**Fast-forward 快进。顺利**

**解决冲突**

**--no-ff //禁用快进，显示更多合并细节。**

**git merge tag1 ---no-ff -m “XXX”//通过git merge命令来合并tag1到主线分支master,** **-m参数仍然是需要填写合并//的注释信息。**

**}**

**如果有冲突，比如两个分支都改了一个文件file3，则合并时会失败。首先我们在master分支上修改file3文件并提交：**

# 切换到master分支

$ git checkout master

# 修改file3文件

$ echo "master: update file3" >> file3

# 提交到master分支

$ git commit -a -m 'update file3 on master'

**然后切换到experimental，修改file3并提交：**

# 切换到experimental分支

$ git checkout experimental

# 修改file3文件

$ echo "experimental: update file3" >> file3

# 提交到master分支

$ git commit -a -m 'update file3 on experimental'

**切换到master进行合并：**

$ git checkout master

$ git merge experimental

Auto-merging file3

CONFLICT (content): Merge conflict in file3

Automatic merge failed; fix conflicts and then commit the result.

**合并失败后先用git status查看状态，会发现file3显示为both modified，查看file3内容会发现：**

$ cat file3

test

<<<<<<< HEAD

master: update file3

=======

experimental: update file3

>>>>>>> experimental

**上面的内容也可以使用git diff查看，先前已经提到git diff不加参数可以显示未提交到缓存区中的修改内容。**

**REDME.md为Markdown文件**

**如果你觉得你合并后的状态是一团乱麻，想把当前的修改都放弃，你可以用下面的命令回到合并之前的状态：**

**$ git reset --hard HEAD^**

**# 查看file3的内容，已经恢复到合并前的master上的文件内容**

**$ cat file3**

**分支删除：**

**当我们完成合并后，不再需要tag1时，可以使用下面的命令删除：**

**git branch -d tag1**

**分支名称修改：**

**git branch -m oldname newname**

**git stash //将本分支内的文件封存，不再修改。**

**git stash pop //还原**

**git stash list //封存列表**

**git branch -r //查看远程的gitgub分支**

**git push -u origin tag2 //将自己的分支更新推送**

**使用下面的命令将本地仓库同步到远端服务器：**

**$ git push origin master**

**git branch -a //查看本地与远程的所有分支及定位**

**git checkout -b tag2 origin/tag2 //在本地创建tag2分支并且与远程tag2分支相连接**

**git remote add origin** [**https://github.com/XXXX**](https://github.com/XXXX)**... //与远程版本库相连接**

**git remote remove origin //删除与远程版本库的相关联**

**git remote -v //查看git 与哪一个远程版本库相连接 ：（fetch和push可以不同）**

**git commit -am “XXX” //直接将文件区内容添加到分支版本库中去**

**git pull origin tag2//获取tag2 分支的更新，并且将分支合并**

**可以在仓库/home/shiyanlou/gitproject中把myrepo的修改给拉 (pull)下来。执行下面几条命令:**

$ cd /home/shiyanlou/gitproject

$ git pull /tmp/myrepo master

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From /tmp/myrepo

\* branch master -> FETCH\_HEAD

Updating 8bb57aa..866c452

Fast-forward

newfile | 1 +

1 file changed, 1 insertion(+)

create mode 100644 newfile

# 查看当前目录文件

$ls [8:28:02]

README.md file1 newfile

**这就把myrepo的主分支合并到了gitproject的当前分支里了。**

**如果gitproject在myrepo修改文件内容的同时也做了修改的话，可能需要手工去修复冲突。**

**如果你要经常操作远程分支(remote branch),你可以定义它们的缩写:**

**$ git remote add myrepo /tmp/myrepo**

**git pull命令执行两个操作: 它从远程分支(remote branch)抓取修改git fetch的内容，然后把它合并git merge进当前的分支。**

**gitproject里可以用git fetch 来执行git pull前半部分的工作， 但是这条命令并不会把抓下来的修改合并到当前分支里：**

$ git fetch myrepo

From /tmp/myrepo

\* [new branch] master -> myrepo/master

**git pull 与git fetch的区别：抽空在博客上写一篇文章**

**git fetch tag1//用来更新远程分支，不合并**

**git merge tag1 --no-ff //更新后再合并**

**获取后，我们可以通过git log查看远程分支做的所有修改，由于我们已经合并了所有修改，所以不会有任何输出：**

**$ git log -p master..myrepo/master**

**当检查完修改后，gitproject可以把修改合并到它的主分支中：**

**$ git merge myrepo/master**

Already up-to-date.

**如果我们在myrepo目录下执行git pull会发生什么呢？**

**myrepo会从克隆的位置拉取代码并更新本地仓库，就是把gitproject上的修改同步到本地:**

# 进入到gitproject

$ cd /home/shiyanlou/gitproject

# 添加一行内容到newfile

$ echo "gitproject: new line" >> newfile

# 提交修改

$ git commit -a -m 'add newline to newfile'

[master 8c31532] add newline to newfile

1 file changed, 1 insertion(+)

# 进入myrepo目录

$ cd /tmp/myrepo

# 同步gitproject的所有修改

$ git pull

remote: Counting objects: 6, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From /home/shiyanlou/gitproject

8bb57aa..8c31532 master -> origin/master

Updating 866c452..8c31532

Fast-forward

newfile | 1 +

1 file changed, 1 insertion(+)

**因为myrepo是从gitproject仓库克隆的，那么他就不需要指定gitproject仓库的地 址。因为Git把gitproject仓库的地址存储到myrepo的配置文件中，这个地址就是在git pull时默认使用的远程仓库：**

$ **git config --get remote.origin.url**

/home/shiyanlou/gitproject

**如果myrepo和gitproject在不同的主机上，可以通过ssh协议来执行clone 和pull操作：**

$ git clone localhost:/home/shiyanlou/gitproject test

**远程公共Git仓库**

**开发过程中，通常大家都会使用一个公共的仓库，并clone到自己的开发环境中，完成一个阶段的代码后可以告诉目标仓库的维护者来pull自己的代码。**

**如果你和维护者都在同一台机器上有帐号，那么你们可以互相从对 方的仓库目录里直接拉所作的修改，git命令里的仓库地址也可以是本地的某个目录名：**

**$ git clone /path/to/repository**

**$ git pull /path/to/other/repository**

**也可以是一个ssh地址：**

**$ git clone ssh://yourhost/~you/repository**

**github介绍**

**git remote ————git remote -v**

**git remote add origin https://...**

**git push ————git push -u origin master**

**clone一个练习项目gitproject:**

**$ git clone** [**http://git.shiyanlou.com/shiyanlou/gitproject**](http://git.shiyanlou.com/shiyanlou/gitproject)

**将修改推到一个公共仓库**

**通过http或是git协议，其它维护者可以通过远程访问的方式抓取(fetch)你最近的修改，但是他们 没有写权限。如何将本地私有仓库的最近修改主动上传到公共仓库中呢？**

**最简单的办法就是用git push命令，推送本地的修改到远程Git仓库，执行下面的命令:**

**$ git push ssh://yourserver.com/~you/proj.git master:master**

**或者**

**$ git push ssh://yourserver.com/~you/proj.git master**

**git push命令的目地仓库可以是ssh或http/https协议访问。**

**当推送代码失败时要怎么办**

**如果推送(push)结果不是快速向前fast forward，可能会报像下面一样的错误：**

error: remote 'refs/heads/master' is not an ancestor of

local 'refs/heads/master'.

Maybe you are not up-to-date and need to pull first?

error: failed to push to 'ssh://yourserver.com/~you/proj.git'

**这种情况通常是因为没有使用git pull获取远端仓库的最新更新，在本地修改的同时，远端仓库已经变化了（其他协作者提交了代码），此时应该先使用git pull合并最新的修改后再执行git push：**

**$ git pull**

**$ git push ssh://yourserver.com/~you/proj.git master**

Git标签

**1.轻量级标签**

**我们可以用 git tag不带任何参数创建一个标签(tag)指定某个提交(commit):**

# 进入到gitproject目录

$ cd /home/shiyanlou/gitproject

# 查看git提交记录

$ git log

# 选择其中一个记录标志位stable-1的标签，注意需要将后面的8c315325替换成仓库下的真实提交内，commit的名称很长，通常我们只需要写前面8位即可

$ git tag stable-1 8c315325

# 查看当前所有tag

$ git tag

stable-1

**这样，我们可以用stable-1 作为提交 8c315325 的代称。**

**前面这样创建的是一个“轻量级标签”。**

**如果你想为一个tag添加注释，或是为它添加一个签名, 那么我们就需要创建一个 "标签对象"。**

**标签对象**

**git tag中使用-a， -s 或是 -u三个参数中任意一个，都会创建一个标签对象，并且需要一个标签消息(tag message)来为tag添加注释。 如果没有-m 或是 -F 这些参数，命令执行时会启动一个编辑器来让用户输入标签消息。**

**当这样的一条命令执行后，一个新的对象被添加到Git对象库中，并且标签引用就指向了一个标签对象，而不是指向一个提交，这就是与轻量级标签的区别。**

**下面是一个创建标签对象的例子:**

$ git tag -a stable-2 8c315325 -m "stable 2"

$ git tag

stable-1

stable-2

**2.签名的标签**

**签名标签可以让提交和标签更加完整可信。如果你配有GPG key，那么你就很容易创建签名的标签。首先你要在你的 .git/config 或 ~/.gitconfig 里配好key。**

**下面是示例:**

[user]

signingkey = <gpg-key-id>

**你也可以用命令行来配置:**

$ git config (--global) user.signingkey <gpg-key-id>

**现在你可以在创建标签的时候使用-s 参数来创建“签名的标签”：**

$ git tag -s stable-1 1b2e1d63ff

**如果没有在配置文件中配GPG key,你可以用-u参数直接指定。**

$ git tag -u <gpg-key-id> stable-1 1b2e1d63ff

**git clone**

**git fetch**

**git pull**

**fork**

**pull request**

**实例：**

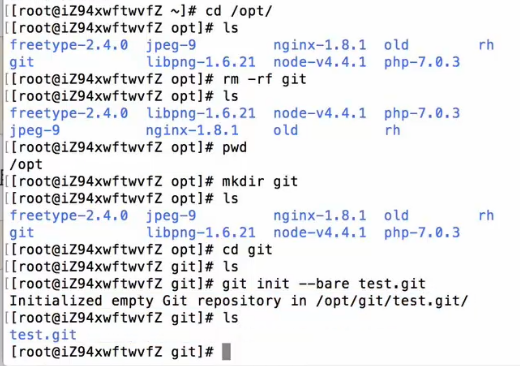
**别名 配置远程服务器 免密登陆 忽略文件**

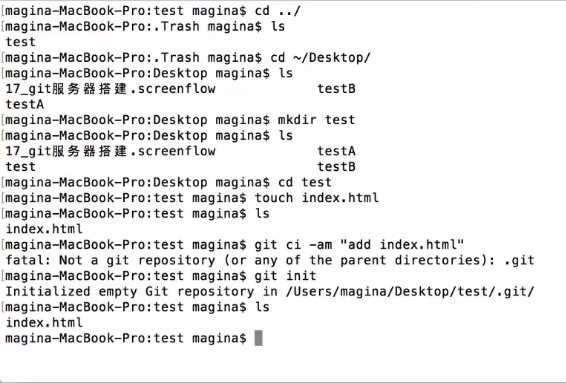
**修改别名：**

**git config --global alias.ci commit //将commit命令修改为ci的别名**

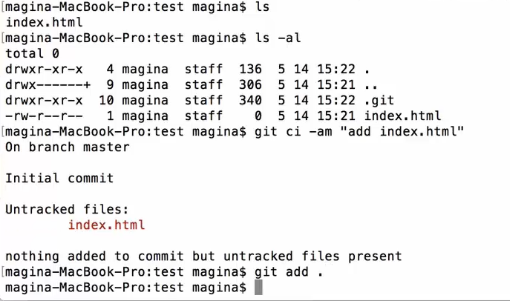
**搭建一个git服务器：**

1. **要有一台自己的服务器（云服务器）**
2. **登陆自己的远程服务器（ssh** [**root@XXX.XXX.cn**](mailto:root@XXX.XXX.cn)**）**
3. **yum install git\***
4. **cd /opt/ ; ls**
5. **建立一个自己的git仓库，一般建在 /opt 目录下；**
6. **初始化git 仓库git init --bare test.git**

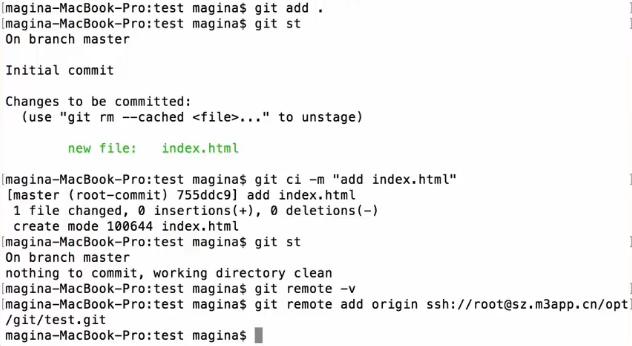
****

1. **打开客户端的终端，在客户端创建文件夹test；**
2. **使用git init命令来创建一个git仓库初始化。**
3. **不可以直接用git commit -am “XXX”来提交到版本控制中推送，会出现文件未被跟踪，没有任何文件被提交。也就是说如果直接使用 -am,而文件没有被添加到版本控制，是无法执行commit操作的，这一点与自己使用github有所不同，因为这是使用的自己的云主机。**
4. 
5. **解决方案,老老实实的使用git add “XXXX”或者将当前目录的所有文件添加进来即**

**(git add . //.号的意思是当前目录)，先将文件添加进来。**



1. **git commit -m “XXXX”将文件提交**
2. **关联到远程服务器，先查看这个项目是否已经关联到远程，使用git remote -v命令查看，若无回应说明没有关联到远程。**



1. **若无关联使用git remote add origin ssh://XXXX 命令将项目关联到自己云主机，注意地址不要写错。并且需要填写主机密码。**

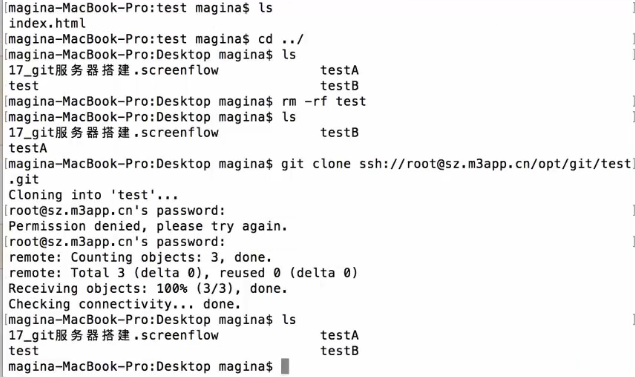
**git remote -v 查看是否关联成功。**

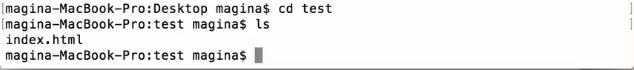
1. **将自己的文件push 到云主机上使用 git push -u origin master 命令**



**10．验证是否push成功删除本地的test文件：rm -f test**

**clone 远程服务器端的test到本地 ：git clone ssh://XXXX 并输入密码。**





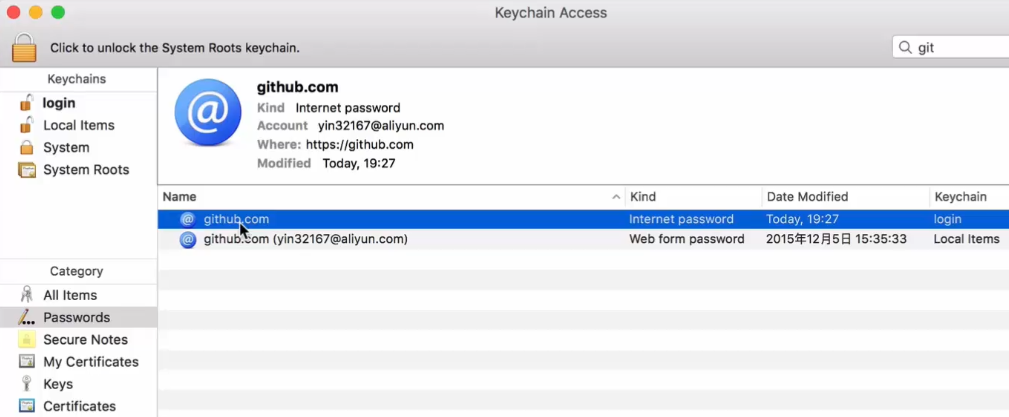
**git的免密码登录，以及创建ssh用户：**

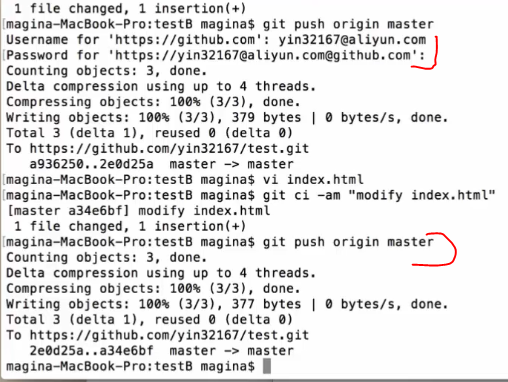
**让服务器记住这台电脑**

**https登陆的话并非是服务器记住了用户名和密码，而是每次主动填充用户名和密码来进行登陆。**

**在keychain里面保存了相应的用户名和密码**

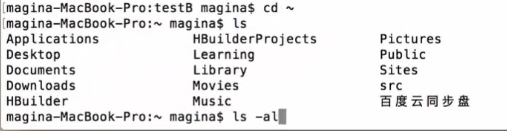
**(1)实现github网站上的免密登录**





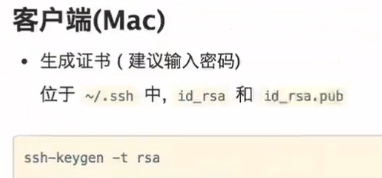
**如果使用的是SSH方式**

1. **生成一对密钥**
2. **安装openssl工具**
3. **一般情况下生成的密钥都会在当前用户下的.ssh文件夹内使用 ls -al来查找**



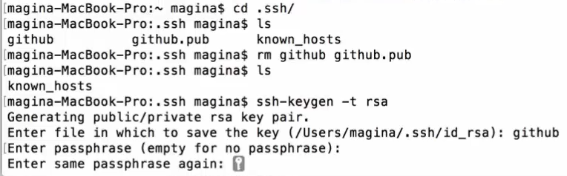


1. **通过客户端生成SSH证书命令来执行密钥生成**



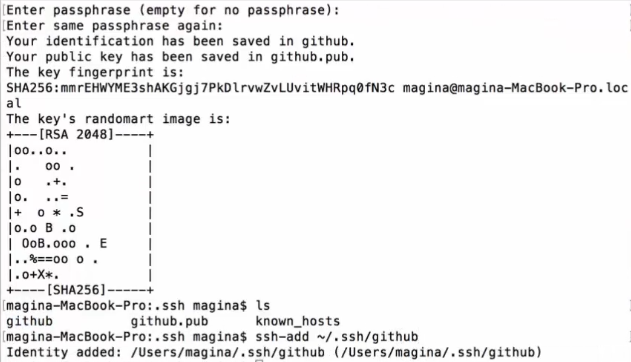
1. **为输入的保存文件取一个名称比如叫github**

**输入密码可以不指定**



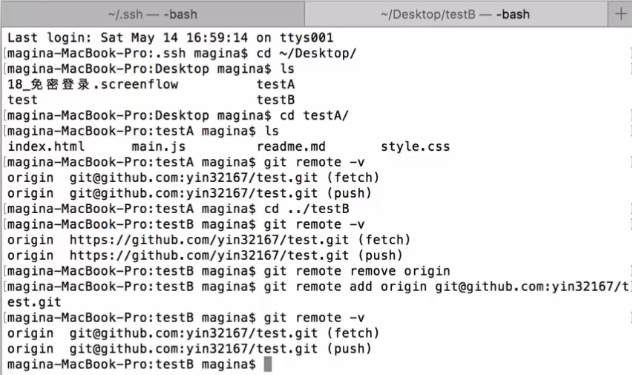
**这样就生成了一个以ISA加密的公钥和私钥**

**如下 github为私钥 github.pub为公钥**



**通过ssh-add ~/.ssh/github 命令将名字叫做github的私钥添加到当前目录下**

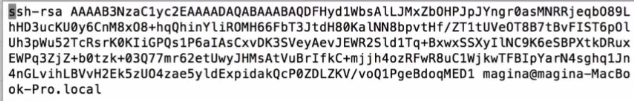
**改变一下test里面的远程路径**



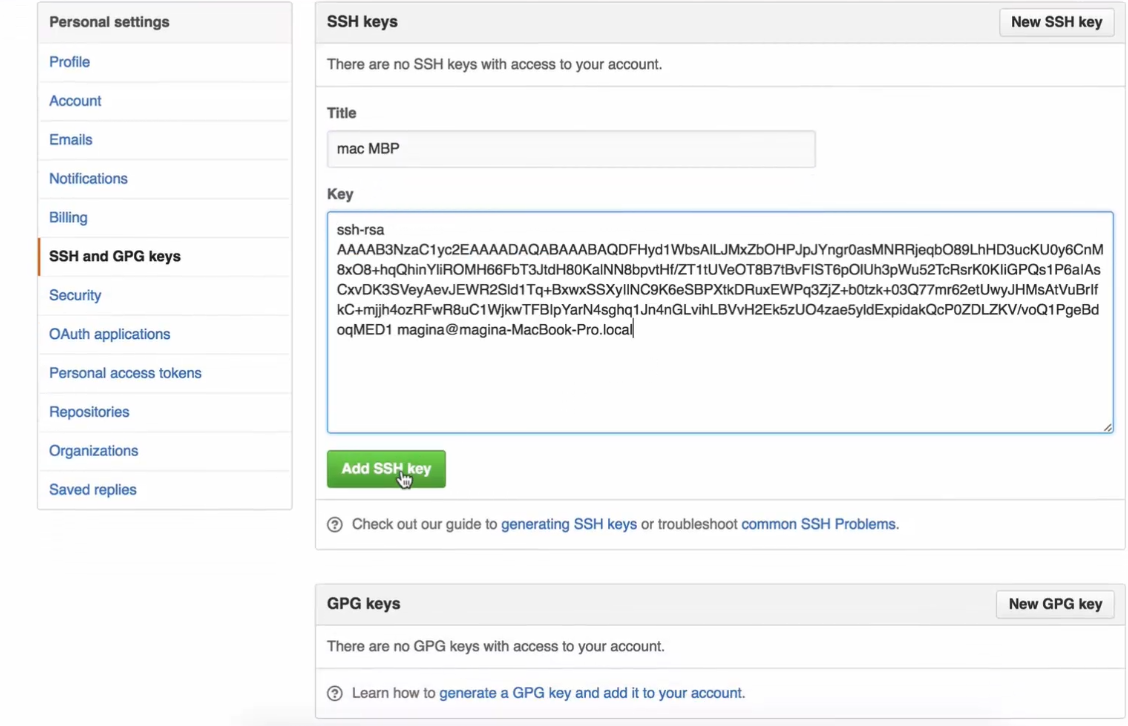
1. **最重要一步：把公钥上传到gihub后台**

**vi github.pub 打开后复制里面内容，粘贴到github**

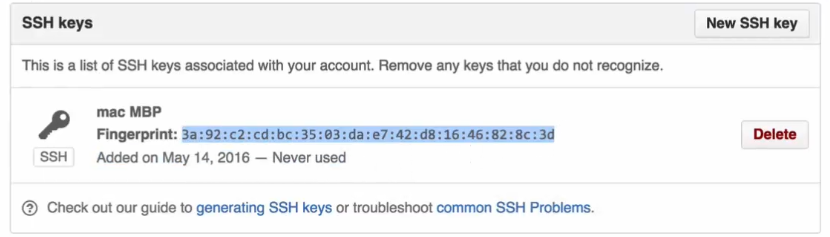




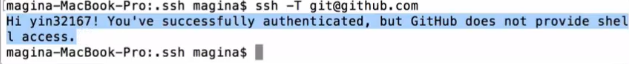
**在github的SSH and GPG keys 中点击 New SSH key 创建公钥，**

**在tittle 中填写标题，key中粘贴公钥，点击Add SSH key 提交**

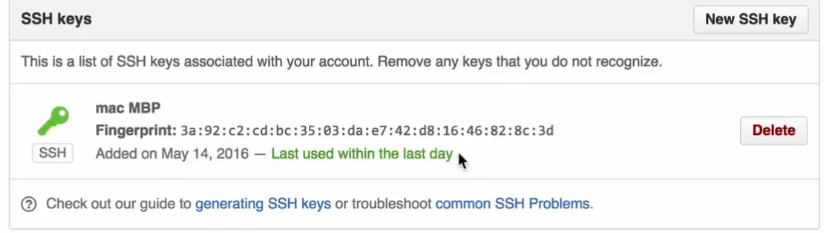
**添加成功**



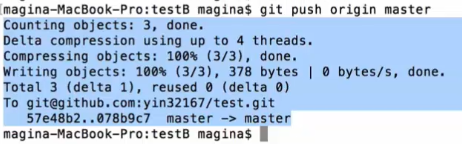
**通过ssh -T** [**git@github.com**](mailto:git@github.com) **命令来校验一下 ，提示成功。**



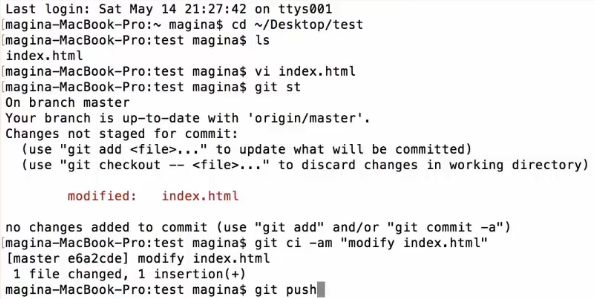
**在github网站上提示公钥最近被使用**

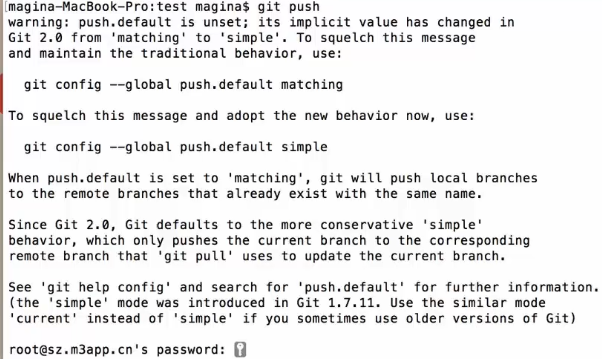


**push时不会再出现错误**



**（2）搭建自己的服务器的免密登录**





**可以看到当我们push的时候需要我们输入服务器密码**

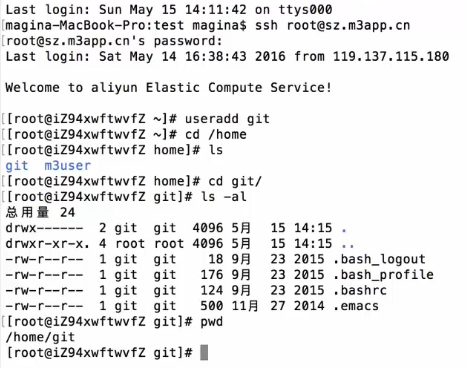


**我们要做一些设置来实现push到自己的远程云服务器的时候不需要再输入密码;**

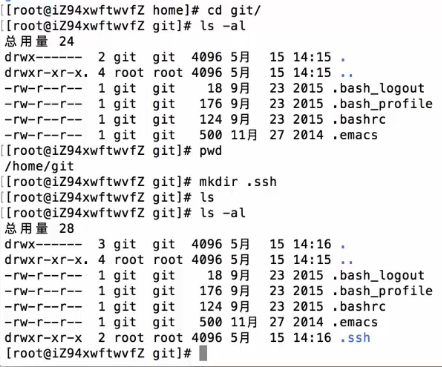
**首先登录到自己的远程云服务器**

**使用命令useradd git 来在服务器创建一个git用户**

**创建成功后在/home目录下会有一个git文件夹，git用户相关的文件都会在这个目录下面**

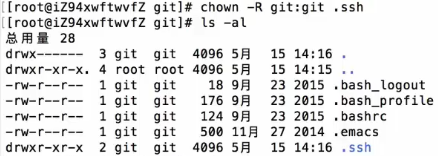


**因为git里边并没有.ssh的文件夹，所以使用**mkdir .ssh**来创建一个。**



**将这个文件夹的所属者设为git用户的git用户组下面的文件，使用命令**

chown -R git:git .ssh

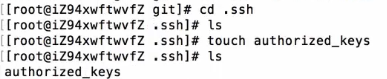


**创建 authorized\_keys**

**1.** cd /home/git/.ssh

**2.** touch authorized\_keys

**如果没有找到 .ssh 文件夹, 则自己创建, 记住将这些文件的所属用户和用户组都改为 git**



**初始化Git仓库**

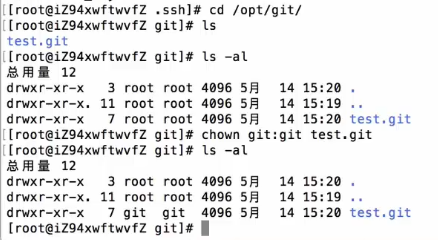
mkdir /opt/git

cd /opt/git

git init --bare test.git

**更改文件或目录的属主和属组**

chown -R git:git test.git

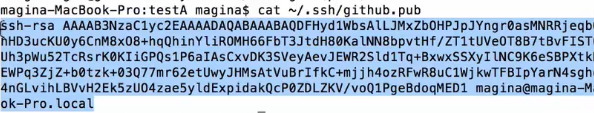


**客户端(Mac)**

**生成证书 ( 建议输入密码)**

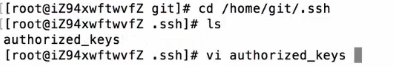
**位于 ~/.ssh 中, id\_rsa 和 id\_rsa.pub**

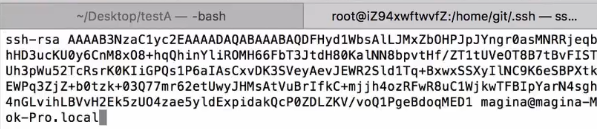
**使用**ssh-keygen -t rsa **命令**



**复制客户端的这个证书再回到远程服务器**

**将 id\_rsa.pub 的内容复制到服务器 /home/git/.ssh/authorized\_keys 中**



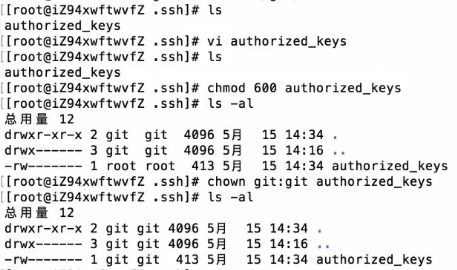


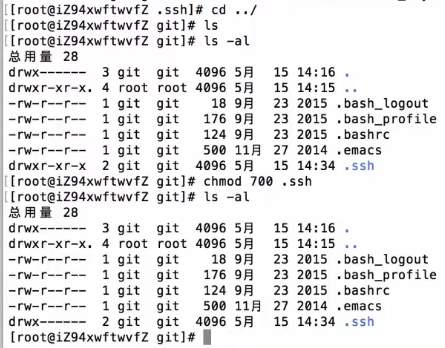
**克隆远程仓库**

git clone [git@yinnote.com:22/opt/git/test.git](mailto:git@yinnote.com:22/opt/git/test.git)

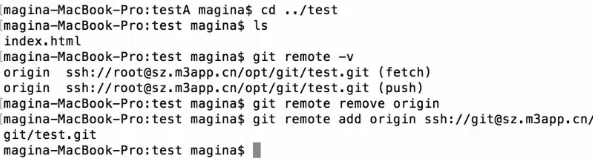
**远程登陆不要密码，authorized\_keys不生效的解决方法**

1. **.ssh目录的权限必须是700**
2. **.ssh/authorized\_keys文件权限必须是600**

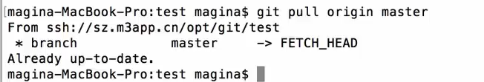




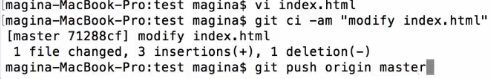
**修改客户端的远程**



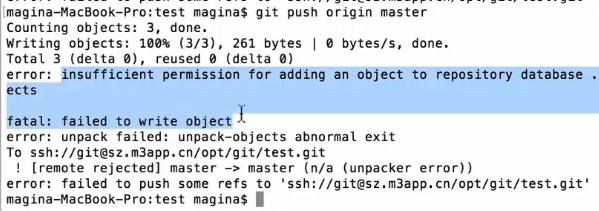
**这个时候再使用pull命令就不用再输入密码了**



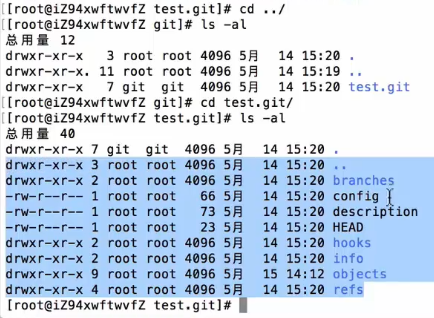
**将修改push到origin上的master分支**



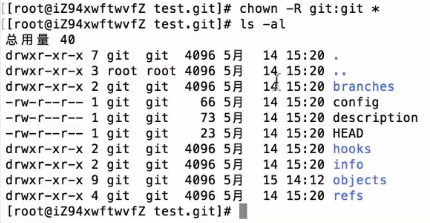
**提示我们权限不足，无法写入对象中的远程库当中。**



**看一下远程服务器，发现test.git是git权限，而test.git文件夹里边则是root用户权限。因为其他用户是有read权限的，所以可以执行pull操作，但是无法执行push操作。**



**使用**chown -R git:git \* **操作将test.git里边的所有文件都更改权限**



**在客户端可以push成功**



**7\_版本控制 Git**

**服务器**

**创建用户**

useradd git

**禁用 git 用户 shell登录 (可选)**

vi /etc/passwd

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

**改为：**

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

**创建 authorized\_keys**

1. cd /home/git/.ssh

2. touch authorized\_keys

**如果没有找到 .ssh 文件夹, 则自己创建, 记住将这些文件的所属用户和用户组都改为 git**

**初始化Git仓库**

mkdir /opt/git

cd /opt/git

git init --bare test.git

**更改文件或目录的属主和属组**

**chown -R git:git test.git**

**客户端(Mac)**

**生成证书 ( 建议输入密码)**

**位于 ~/.ssh 中, id\_rsa 和 id\_rsa.pub**

ssh-keygen -t rsa

**将 id\_rsa.pub 的内容复制到服务器 /home/git/.ssh/authorized\_keys 中**

**克隆远程仓库**

git clone git@yinnote.com:22/opt/git/test.git

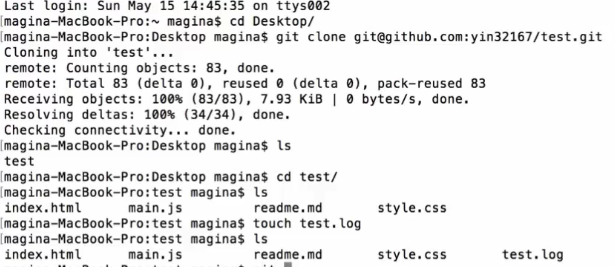
**远程登陆不要密码，authorized\_keys不生效的解决方法**

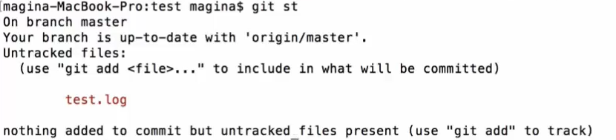
**.ssh目录的权限必须是700**

**.ssh/authorized\_keys文件权限必须是600**

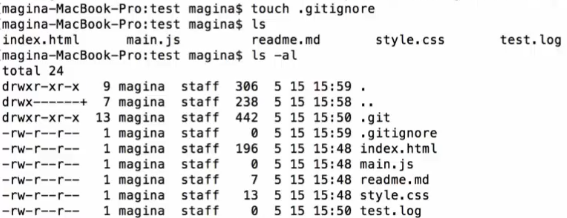
**文件忽略：**

**如下忽略一个 .log文件**





**创建一个.gitignore文件**

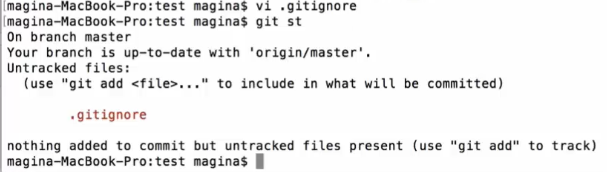


**用vi对.gitignore文件进行相应设置**



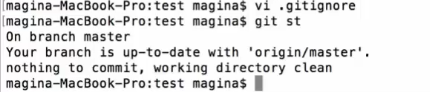
**在里面指定要忽略的文件**



**此时git会先读取.gitignore文件，在查找忽略文件中指定的要忽略的文件，并不再跟踪指定文件，** 

**可以提交.gitignore文件，也可以选择不提交（最好是提交）。如果不提交，就在.gitignore文件中把自己添加进来**





**如果选择忽略一些文件，就千万不要把这些文件添加到版本控制中，一旦添加到版本控制中，忽略文件就不再起作用了。**

**万一添加了，就用git rm --cache XXX 将其从版本控制中删除掉。**

**.gitignore文件只能作用于当前的目录下面，不能作用于其他的文件夹**



**全局忽略：**

**git -忽略文件**

**全局忽略**

**---**

touch ~/.gitignore\_global

git config --global core.excludesfile ~/.gitignore\_global

**> 在 .gitignore\_global 文件中, 添加规则**

**可以通过git config --global --list 命令查看其所有配置**



**项目忽略**

**---**

**1. 对于每一级工作目录，创建一个 `.gitignore` 文件**

**2. 向该文件中添加要忽略的文件或目录**

**3. 在创建并编辑这个文件之前，一定要保证要忽略的文件没有添加到git索引中**

**4. `.gitignor`文件同样可以像其它文件一样加到项目仓库里**

**提交空文件夹**

**---**

**一般情况下, 向空文件夹中添加 `.gitkeep` 文件, 并将其添加到版本控制, 即所在的文件夹也会被跟踪, 对文件这个文件名称, 可以任意指定, 一般习惯指定这个名字**

**忽略规范**

**---**

**1. 所有空行或者以注释符号 # 开头的行都会被 Git 忽略**

**2. 匹配模式最后跟反斜杠(/)说明要忽略的是目录**

**3. 要忽略指定模式以外的文件或目录,可以在模式前加上惊叹号(!)取反**

**4. 星号(\*)匹配零个或多个任意字符**

**5. 问号(?)只匹配一个任意字符**

**6. [abc] 匹配任何一个列在方括号中的字符**

**7. 如果在方括号中使用短划线分隔两个字符, 表示所有在这两个字符范围内的都可以匹配(比如[0-9])**