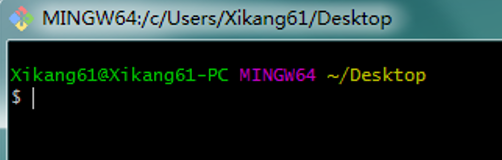
Git的快速使用：

# Git使用的软件

1. Git

在安装完 Git 后，会有一个叫做 Git Bash 的程序，打开之后会出现一个跟 Linux 命令行操作界面神似的命令行窗口，其中可以输入各种Git指令和一些 Linux 指令



1. TortoiseGit
2. Github

# Git的传输方式

## https方式:

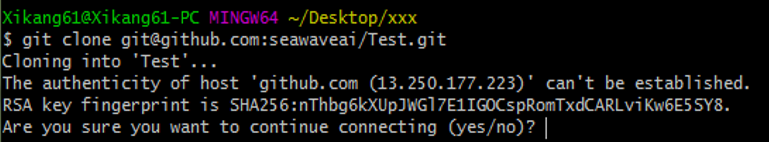
克隆代码方便；但是每次提交代码要输入账号和密码，这就显得不安全也不方便了。

如：从github上拉代码( <https://github.com/seawaveai/Test.git> )，https就可以直接下载。但是要提交代

码时就要账号和密码，你的github账号和密码能随便给人吗！即使给他人能分清是谁提交的吗！

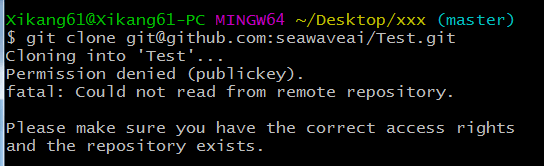
## ssh方式:

只要配置了RSA之后就没有验证的问题了。(git服务器上配置公钥，配置一次就ok)



（上图是**第一次**用ssh克隆代码的时候，询问你是否下载github.com的rsa的指纹。当然要下载了，

果断yes，然后在C:\Users\XXXX\.ssh文件夹下会多出known\_hosts文件）

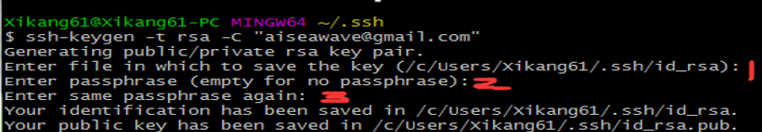


（上图是克隆代码时没有配置rsa出现的提示）

# 简单使用SSH方式

1. $ ssh-keygen -t rsa -C "zhongnanhai@gmail.com" （邮箱填自己的）

2. 默认点击三次就会生成rsa秘钥对：



说明：

1. 在那儿打开bash，相应位置就出现两个文件：id\_rsa（私钥）， id\_rsa.pub（公钥）
2. 默认三次点击：第一次---询问你是否设置秘钥对的名字，否则默认起名id\_rsa；

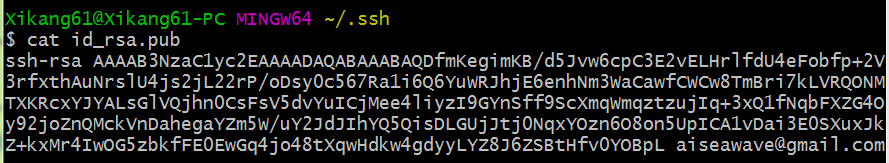
第二次---询问你给秘钥对设置一个密码；第三次---让再确认秘钥对的密码

(开始使用的时候建议三次都默认点击)

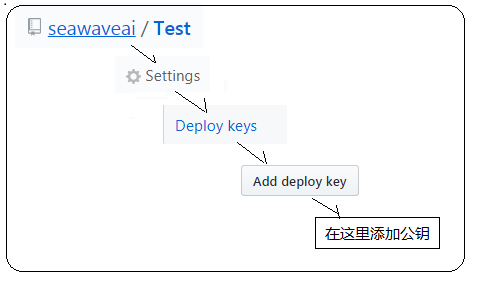
1. 默认情况下只能生成一对秘钥，如果本机管理多个项目，就要生成多个SSH并且区分
2. 查看秘钥

cat ~/.ssh/id\_rsa.pub (查看公钥, ~/.ssh/是路径)

cat id\_rsa.pub



1. 添加公钥(某个项目中)



把查看的公钥，添加到github上。

1. 测试配置是否成功

ssh -T [git@github.com](mailto:git@github.com)

有关ssh更详细问题可以查看<https://help.github.com/articles/checking-for-existing-ssh-keys/>

# 问题

## ssh配置生效的范围

如上生成一个rsa后：

如果配置在某个项目下，则就只能对该项目生效；

如果配置在个人设置中，则就可以对该github用户所有的项目都生效，就是你可以下载和提交代码到任何项目，这就不安全了！

## 如何管理项目对成员开放呢！

在同一账号下，公钥在某个项目下使用了，在其他项目下或个人设置那就不能使用。配置一个公钥，来开放多个项目无法实现。

如：张三要向服务器(管理员)请求使用两个项目，如果只发送一个公钥，就无法实现。这就要通过使用多个SSH-Key来实现。

默认情况下只生成一对秘钥，即id\_rsa。

## Git 配置多个SSH-Key

1. 生成一个SSH-Key，取名work\_rsa

$ ssh-keygen -t rsa -C "xxxx@163.com" 第一次点击后设置名字：work\_rsa (不再默认为id\_rsa)

1. 生成一个SSH-Key，取名github\_rsa

$ ssh-keygen -t rsa -C "xxxx@gmail.com" 第一次点击后设置名字：github\_rsa

(此时.ssh目录下应该有4个文件：work\_rsa和work\_rsa.pub，github\_rsa和github\_rsa.pub，分别将他们的公钥文件（work\_rsa.pub，github\_rsa.pub）内容配置到对应的code仓库上。)

1. 添加私钥 ( 因为默认只读取id\_rsa私钥，为了让SSH识别新的私钥 )

$ ssh-add ~/.ssh/work\_rsa ( ~/.ssh/是路径 )

$ ssh-add ~/.ssh/github\_rsa

(如果执行ssh-add时提示”Could not open a connection to your authentication agent”，

可以先执行命令 $ ssh-agent bash ；然后再运行ssh-add命令添加私钥。

  可以使用使用命令 $ ssh-add –l 检查已保存的公钥

可以通过 $ ssh-add -D 来清空私钥列表 )

1. 修改配置文件

# 若.ssh目录下无config文件，那么创建

touch config

# config添加以下内容



1. 把公钥配置到服务器上

把work\_rsa.pub 与 github\_rsa.pub里面的公钥分别配置到项目，然后两个项目都可以使用了。

1. 测试配置

$ ssh -T [git@work.github.com](mailto:git@work.github.com) (@后面的就是config配置中的Host)

$ ssh -T [git@github.com](mailto:git@github.com) (码云默认就是git@gitee.com)

1. 克隆代码

[git@github.com:seawaveai/Test.git](mailto:git@github.com:seawaveai/Test.git) 地址要改成[git@work.github.com:seawaveai/Test.git](mailto:git@work.github.com:seawaveai/Test.git)

[git@github.com:seawaveai/TestSSH.git](mailto:git@github.com:seawaveai/TestSSH.git) 这个无需修改

( Git会根据~/.ssh/config配置下的Host去寻找授权信息，

比如Host设置为github.com，那么git就会使用~/.ssh/github\_rsa进行授权

比如Host设置为work.github.com，那么git就会使用~/.ssh/work\_rsa进行授权 )

Host就是HostName的别名，说白了就是使用这个别名来区分授权。

1. 也可以直接设置路径和名字

$ ssh-keygen -t rsa -C "xxxx@163.com" -f /c/Users/Xikang61/.ssh/work\_rsa 这样就直接跳过了第一次点击

$ ssh-keygen -t rsa -C "xxxx@gmail.com" -f /c/Users/Xikang61/.ssh/github\_rsa

## 多账户的使用

既然实现了多SSH-Key，那么依葫芦画瓢，稍微修改下config文件就可以了。



(公司的是gitlab，自己的在github上，因为gitlab与github是两个平台，其实使用同一个秘钥也可以)

## 参考博客

参考blog

<https://blog.csdn.net/gdutxiaoxu/article/details/53573426>

<https://my.oschina.net/stefanzhlg/blog/529403>

<https://blog.csdn.net/wangpingfang/article/details/53117087>

# Git命令的使用

首先，你需要配置名字和邮箱

git config --global user.name "you\_name" (不输入名字、邮箱就可以查看是否配置成功)

git config --global user.email "xxx@163.com"

git init (初始化本地仓库)

git clone url (克隆远程仓库)

git add xxx (添加单个文件或文件夹)

git add . (添加全部文件)

git commit -m "first commit" (一定要跟-m添加描述信息，否则非法)

git pull origin master (origin部分指定的是远程仓库的git地址，master部分指定的是同步到哪个分支上)

git push origin master

xxx.gitignore 忽略文件

git status (查看自上次提交后文件的修改情况)

git diff

git diff xxx (查看具体文件前后的不同)

git checkout xxx (撤销修改—没执行add命令的情况)

git reset HEAD xxx (撤销修改—执行了add命令的情况)

git log (查看日志---日志包括：id，提交人，提交日期，提交描述)

git log -1 (看一条) git log -2看两条

git log id -1 (也可以加id)

git log -1 -p (加-p查看这条记录具体修改了什么内容)

git branch -a (查看版本库中有那些分支, 前面有\*标识的代表当前分支)

git branch version1.0 (创建分支version1.0)

git checkout version1.0 (切换当前分支到version1.0)

git checkout master

git merge version1.0 (此两条指令,实现version1.0分支合并到master分支)

git branch -D version1.0 (删除version1.0分支---删除的时候,version1.0不能是当前版本)

git fetch origin master (执行后,代码会存在origin/master分子上)

git diff origin/master (查看远程版本到底修改了哪些东西)

git merge origin/master (将origin/master上的修改合并到主分支上)

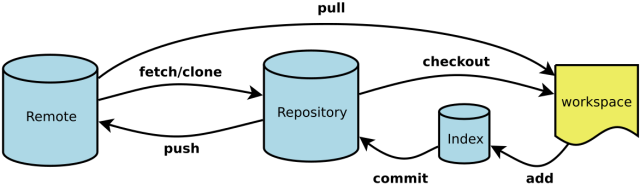
==== git pull origin master (fetch与pull之间的区别)

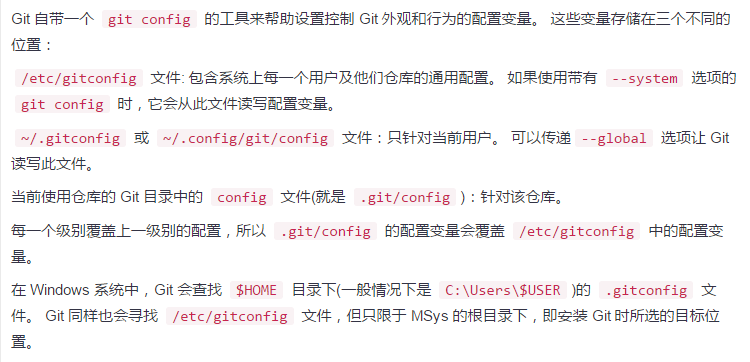
# 整理

**Workspace**：工作区  
**Index/Stage**：暂存区，也叫索引  
**Repository**：仓库区（或本地仓库），也存储库  
**Remote**：远程仓库

**有关几个名词解释**

**工作区**: 通过git init创建的代码库的所有文件但是不包括.git文件(版本库)  
**暂存区**: 通过git add ./\*/\*Xxx/Xxxx\* 添加的修改,都是进入到暂存区了,肉眼不可见 通过 git status 可以看到修改的状态。





git config --list

git version

git config --list 命令来列出所有 Git 当时能找到的配置

git config user.name 检查 Git 的某一项配置

git clone http://git.oschina.net/yiibai/git-start.git mygit-start克隆远程仓库时自定义本地仓库的名字mygit-start

### 状态简览

git status 命令的输出十分详细，但其用语有些繁琐。 如果你使用 git status -s 命令或 git status --short 命令，将得到一种更为紧凑的格式输出。 运行 git status -s，状态报告输出如下：

$ git status -s

M README.md

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

Shell

新添加的未跟踪文件前面有 ?? 标记，新添加到暂存区中的文件前面有 A 标记，修改过的文件前面有 M 标记。 你可能注意到了 M 有两个可以出现的位置，出现在右边的 M 表示该文件被修改了但是还没放入暂存区，出现在靠左边的 M 表示该文件被修改了并放入了暂存区。 例如，上面的状态报告显示： README 文件在工作区被修改了但是还没有将修改后的文件放入暂存区,lib/simplegit.rb 文件被修改了并将修改后的文件放入了暂存区。 而 Rakefile 在工作区被修改并提交到暂存区后又在工作区中被修改了，所以在暂存区和工作区都有该文件被修改了的记录。

Changes to be committed: 网络已暂存;

*Changes not staged for commit*:已跟踪文件的内容发生了变化，但还没有放到暂存区