Git学习笔记

廖雪峰老师网站学习总结的Git笔记，感谢廖雪峰老师的知识分享！

Git介绍

- 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是由Linus写的一个分布式版本控制系统。
- 集中式VS分布式： SVN是集中式版本控制，GIT跟SVN一样有自己的集中式版本库或服务器，但GIT更倾向于被使用于分布式模式，也就是每个开发人员从中心版本库/服务器上chect out代码后会在自己的机器上克隆一个自己的版本库。Git本地仓库包含代码库还有历史库，在本地的环境开发就可以记录历史而SVN的历史库存在于中央仓库，每次对比与提交代码都必须连接到中央仓库才能进行。 优点：
    1. 个人开发可以在无中央仓库连接下查看开发的版本历史并提交。
    2. 假如Git的中央仓库挂了，可以创建一个新的中央仓库然后同步就能恢复中央库。

以上只是简单的说明了下Git分布式的特点相对于SVN集中式的优势，其实Git的优点远远不止于这些。

Git官方的帮助提示

```
add         添加文件内容至索引
bisect      通过二分查找定位引入bug的变更
branch      列出、创建或删除分支
checkout    检出一个分支或路径到工作区
clone       克隆一个版本库到一个新目录
commit      记录变更到版本库
diff        显示提交之间、提交和工作区;之间等的差异
fetch       从另外一个版本库下载对象和引用
grep        输出和模式匹配的行
init        创建一个空的Git版本库或重新初始化一个已存在的版本库
log         显示提交日志
merge       合并两个或更多开发历史
mv          移动或重命名一个文件、目录或符号链接
pull        获取并合并另外的版本库或一个本地分支
push        更新远程引用和相关的对象
rebase      本地提交转移至更新后的上游分支中
reset       重置当前HEAD到指定状态
rm          从工作区和索引中删除文件
show        显示各种类型的对象
status      显示工作区状态
tag         创建、列出、删除或校验一个GPG签名的tag对象
```

Git使用

Git配置

```
git config --global user.name "name"

git config --global user.email "email@example.com"
```

`git config`命令的`--global`参数，全局配置，说明这台机器上的所有Git仓库都会使用这个配置，也可以对某个仓库指定不同的用户名和邮箱地址，其实可以直接修改.gitconfig配置文件。

创建版本库

初始化一个Git仓库，使用`git init`命令。

```
git init
```

添加文件到Git仓库

添加文件到Git仓库，分两步： 第一步，使用命令`git add <;file>;`，注意，可反复多次使用，添加多个文件；

```
git add <file>
```

第二步，使用命令`git commit`，完成。

```
git commit -m "description"
```

`git add`可以反复多次使用，添加多个文件， `git commit`可以一次提交很多文件， `-m`后面输入的是本次提交的说明，可以输入任意内容。

查看工作区状态

要随时掌握工作区的状态，使用`git status`命令。

```
git status
```

查看修改内容

用`git status`告诉你有文件被修改过，用`git diff`可以查看修改内容。 工作区(work dict)和暂存区(stage)的比较：

```
git diff
```

暂存区(stage)和分支(master)的比较：

```
git diff --cached
```

工作区和版本库里面最新版本的比较：

```
git diff HEAD -- <file>
```

查看提交日志

```
git log
```

简化日志输出信息

```
git log --pretty=oneline
```

查看命令历史

```
git reflog
```

版本回退

回退上个版本

`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 reset --hard HEAD^
```

以上命令是返回上一个版本，在Git中，用`HEAD`表示当前版本，上一个版本就是`HEAD^`，上上一个版本是`HEAD^^`，往上100个版本写成`HEAD~100`。

回退指定版本号

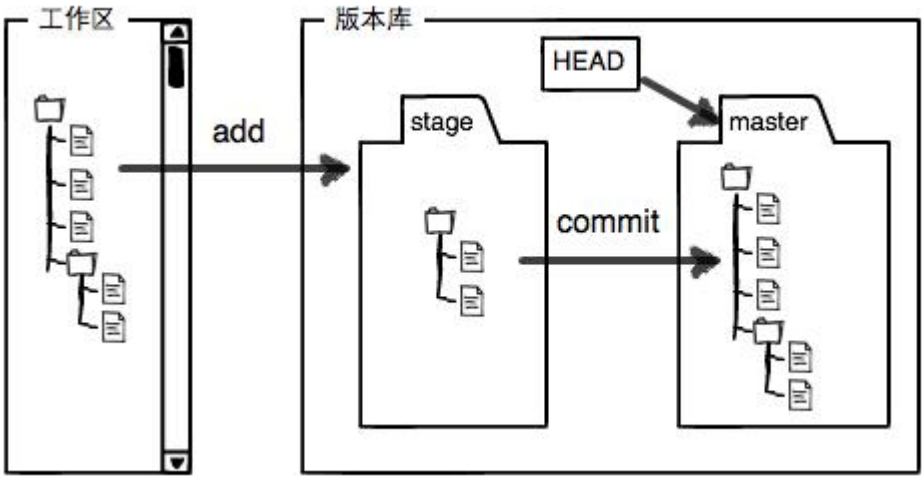
```
git reset --hard commit_id
```

commit_id是版本号，是一个用SHA1计算出的序列

工作区、暂存区和版本库

工作区：在电脑里能看到的目录；

版本库：在工作区有一个隐藏目录`.git`，是Git的版本库。 Git的版本库中存了很多东西，其中最重要的就是称为stage（或者称为index）的暂存区，还有Git自动创建的`master`，以及指向`master`的指针`HEAD`。



- `git add`实际上是把文件添加到暂存区
- `git commit`实际上是把暂存区的所有内容提交到当前分支

撤销修改

这里有两种情况：

1. 一种是file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；
2. 一种是file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。

丢弃工作区的修改

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`。

```
git checkout -- <file>
```

丢弃暂存区的修改

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`，再用命令`git checkout -- file`。 第一步，把暂存区的修改撤销掉(unstage)，重新放回工作区：

```
git reset HEAD <file>
```

第二步，撤销工作区的修改

```
git checkout -- <file>
```

- 结论：

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>;`。
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>;`，就回到了第一步，第二步按第一步操作。
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，进行版本回退，前提是没有推送到远程库。
   总之，就是让这个文件回到最近一次git commit或git add时的状态。

删除文件

命令`git rm`用于删除一个文件

```
git rm <file>
```

`git rm <;file>;`相当于执行

```
rm <file>
git add <file>
```

误删恢复

- 误操作执行了`rm test.txt`，恢复：

执行`git checkout -- test.txt`，把版本库的东西重新写回工作区就行了

- 误操作执行了`git rm test.txt`,工作区内容也删除了，恢复：

先撤销暂存区修改，重新放回工作区，然后再从版本库写回到工作区

```
git reset HEAD test.txt
git checkout -- test.txt
```

- 确实需要删除文件？

执行`git commit -m ";delete test.txt";`，提交后最新的版本库将不包含这个文件

分支

创建分支

```
git branch <branchname>
```

查看分支

```
git branch
```

- `git branch`命令会列出所有分支，当前分支前面会标一个*号。

切换分支

```
git checkout <branchname>
```

创建+切换分支

```
git checkout -b <branchname>
```

合并分支

Fast forward合并分支

```
git merge <branchname>
```

合并分支时，如果可能，Git会用`Fast forward`模式，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用`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。

普通模式合并分支

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 "description" <branchname>
```

因为本次合并要创建一个新的commit，所以加上`-m`参数，把commit描述写进去。合并分支时，加上`--no-ff`参数就可以用普通模式合并，能看出来曾经做过合并，包含作者和时间戳等信息，而fast forward合并就看不出来曾经做过合并。

查看分支合并图

合并后，我们用`git log`看看分支历史：

```
git log --graph --pretty=oneline --abbrev-commit
```

删除分支

```
git branch -d <branchname>
```

丢弃一个没有合并过的分支

git 中未合并的分支无法删除，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。

```
git branch -D <branchname>
```

删除远程分支：

```
git push origin : branch_name
```

可以使用这种语法，推送一个空分支到远程分支，其实就相当于删除远程分支。 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。用`git log --graph`命令可以看到分支合并图。

推送分支

```
git push origin branch-name
```

重命名分支

不会覆盖已经存在的分支：

```
git branch -m branch_name branch_new_name
```

会覆盖已经存在的分支：

```
git branch -M branch_name branch_new_name
```

远程仓库

创建SSH Key

```
ssh-keygen -t rsa -C "email@example.com"
```

关联远程仓库

要关联一个远程库，使用命令`git remote add origin git@server-name:path/repo-name.git`； 这里以github仓库为例：

```
git remote add origin https://github.com/username/repositoryname.git
```

如果使用命令git remote add时报错：

```
git remote add origin git@github.com:username/repositoryname.git
fatal: remote origin already exists.
```

说明本地已经关联了名称为origin的远程库。

查看远程库信息

- 用`git remote` 查看远程库的信息 `git remote -v`可查看详细信息

```
git remote -v
```

可以看到，本地库关联的origin的远程库,可以选择删除该远程库的关联,也可以换个名称关联

删除关联的远程库

先删除已关联的名为`origin`的远程库

```
git remote rm origin
```

推送到远程仓库

关联后，就可以使用命令`git push origin master`推送最新修改； 第一次推送添加参数`-u`：

```
git push -u origin master
```

`-u` 表示第一次推送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。

从远程克隆

从github仓库克隆：

```
git clone https://github.com/username/repositoryname.git
```

在本地创建和远程分支对应的分支

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；

```
git checkout -b branch-name origin/branch-name，
```

建立本地分支和远程分支的关联

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`

```
git branch --set-upstream branch-name origin/branch-name；
```

从本地推送分支

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；

```
git push origin branch-name
```

如果推送失败，先用git pull抓取远程的新提交；

从远程抓取分支

```
git pull
```

如果有冲突，要先处理冲突。

工作现场

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`git stash`

保存工作现场

```
git stash
```

查看工作现场

```
git stash list
```

恢复工作现场

恢复后不删除

一是用`git stash apply`恢复，但是恢复后，stash内容并不删除，你需要用`git stash drop`来删除；二种方式是用`git stash pop`，恢复的同时把stash内容也删了：

- 保存后恢复：

```
git stash apply
```

- 如果需要删除：

```
git stash drop
```

恢复后删除

```
git stash pop
```

标签

tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

新建一个标签

命令`git tag <;name>;`用于新建一个标签，默认为`HEAD`，也可以指定一个commit id

```
git tag <tagname>
```

指定标签信息

```
git tag -a <tagname> -m <description> <branchname> or commit_id
```

`git tag -a <;tagname>; -m ";blablabla...";`可以指定标签信息；

PGP签名标签

```
git tag -s <tagname> -m <description> <branchname> or commit_id
```

`git tag -s <;tagname>; -m ";blablabla...";`可以用PGP签名标签。

查看所有标签

```
git tag
```

推送一个本地标签

```
git push origin <tagname>
```

推送全部未推送过的本地标签

```
git push origin --tags
```

删除一个本地标签

```
git tag -d <tagname>
```

删除一个远程标签

```
git push origin :refs/tags/<tagname>
```

自定义git

Git显示颜色

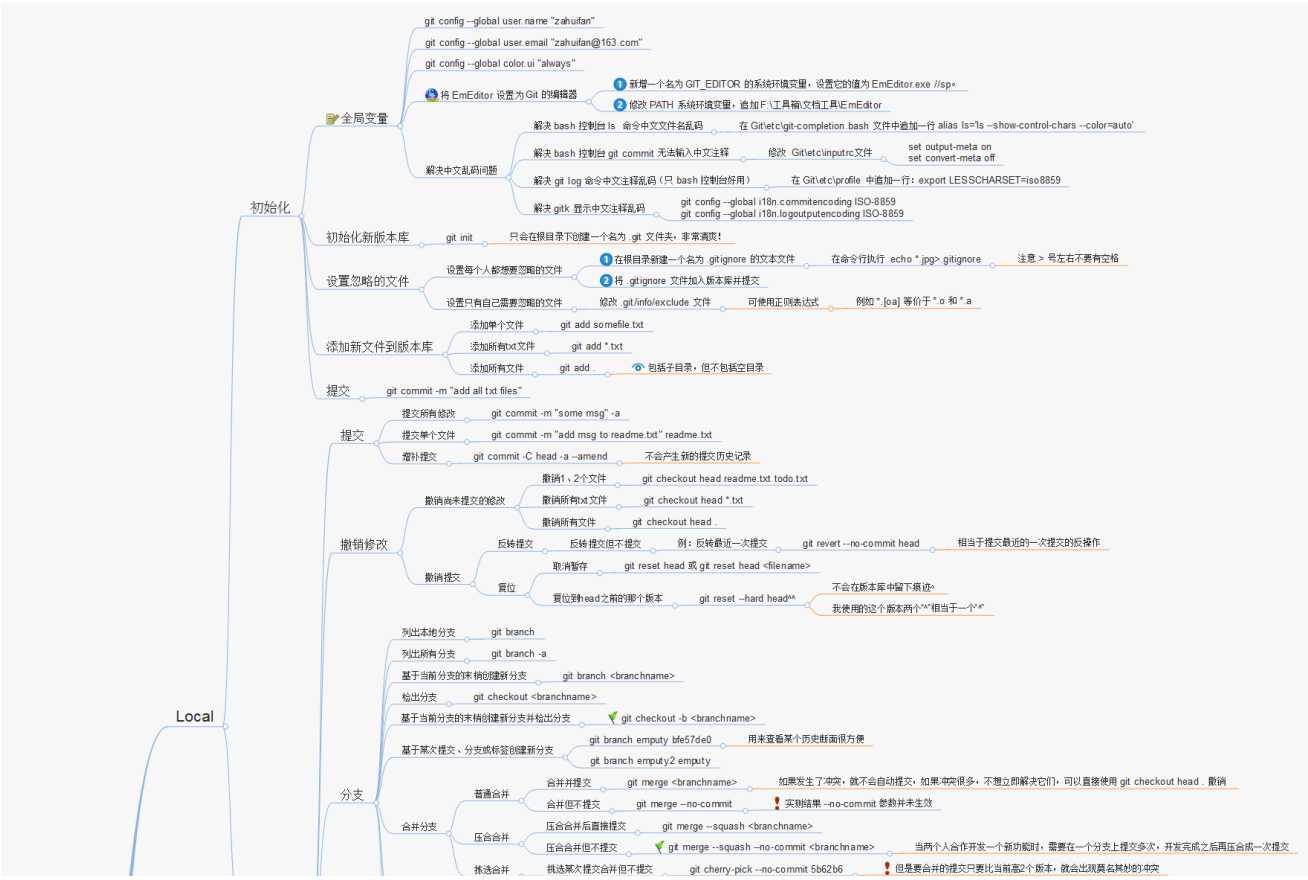
```
git config --global color.ui true`
```

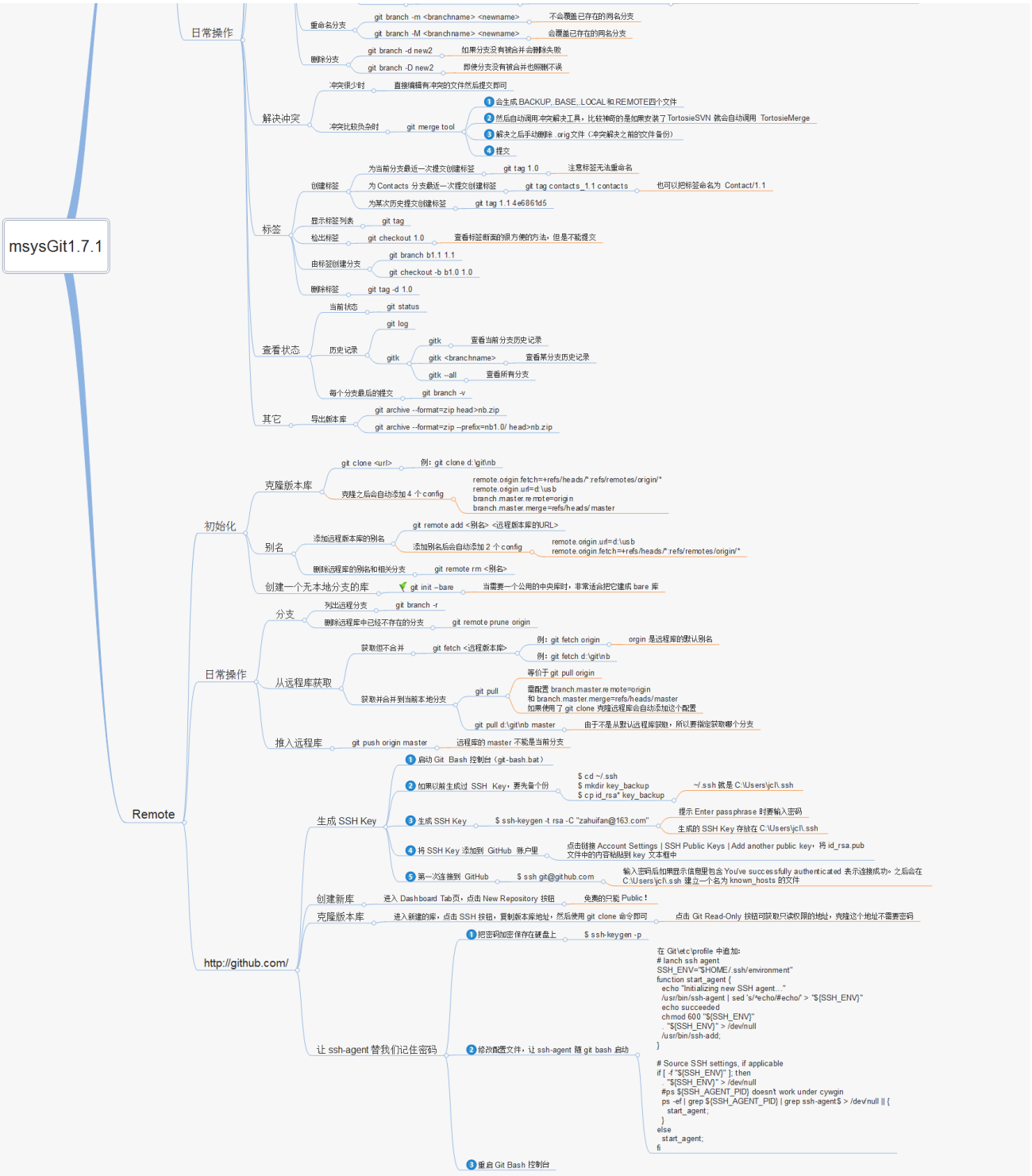
配置别名

配置git lg命令

```
git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold
```

更多git命令见此图：




msysGit1.7.1
日常操作
重命名分支
git branch -m <branchname> <newname>
不会覆盖已存在的同名分支
git branch -M <branchname> <newname>
会覆盖已存在的同名分支
删除分支
git branch -d new2
如果分支没有被合并会删除失败
git branch -D new2
即使分支没有被合并也照删不误
解决冲突
冲突很少时
直接编辑有冲突的文件然后提交即可
冲突比较负杂时
git merge tool
1 会生成 BACKUP, BASE, LOCAL 和 REMOTE 四个文件
2 然后自动调用冲突解决工具，比较神奇的是如果安装了 TortosieSVN 就会自动调用 TortosieMerge
3 解决之后手动删除 .orig 文件（冲突解决之前的文件备份）
4 提交
标签
创建标签
为当前分支最近一次提交创建标签
git tag 1.0
注意标签无法重命名
为 Contacts 分支最近一次提交创建标签
git tag contacts_1.1 contacts
也可以把标签命名为 Contact/1.1
为某次历史提交创建标签
git tag 1.1 4e6861d5
显示标签列表
git tag
检出标签
git checkout 1.0
查看标签断面的很方便的方法，但是不能提交
由标签创建分支
git branch b1.1 1.1
git checkout -b b1.0 1.0
删除标签
git tag -d 1.0
查看状态
当前状态
git status
历史记录
git log
gitk
查看当前分支历史记录
gitk <branchname>
查看某分支历史记录
gitk --all
查看所有分支
每个分支最后的提交
git branch -v
其它
导出版本库
git archive --format=zip head>nb.zip
git archive --format=zip --prefix=nb1.0/ head>nb.zip
Remote
初始化
克隆版本库
git clone <url>
例：git clone d:\git\nb
克隆之后会自动添加 4 个 config
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
remote.origin.url=d:\usb
branch.master.remote=origin
branch.master.merge=refs/heads/master
别名
添加远程版本库的别名
git remote add <别名> <远程版本库的URL>
添加别名后会自动添加 2 个 config
remote.origin.url=d:\usb
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
删除远程库的别名和相关分支
git remote rm <别名>
创建一个无本地分支的库
git init --bare
当需要一个公用的中央库时，非常适合把它建成 bare 库
日常操作
分支
列出远程分支
git branch -r
删除远程库中已经不存在的分支
git remote prune origin
从远程库获取
获取但不合并
git fetch <远程版本库>
例：git fetch origin
orgin 是远程库的默认别名
例：git fetch d:\git\nb
获取并合并到当前本地分支
git pull
等价于 git pull origin
需配置 branch.master.remote=origin
和 branch.master.merge=refs/heads/master
如果使用了 git clone 克隆远程库会自动添加这个配置
git pull d:\git\nb master
由于不是从默认远程库获取，所以要指定获取哪个分支
推入远程库
git push origin master
远程库的 master 不能是当前分支
http://github.com/
生成 SSH Key
1 启动 Git Bash 控制台（git-bash.bat）
2 如果以前生成过 SSH Key，要先备个份
$ cd ~/.ssh
$ mkdir key_backup
$ cp id_rsa* key_backup
~/.ssh 就是 C:\Users\jc\.ssh
3 生成 SSH Key
$ ssh-keygen -t rsa -C "zahuifan@163.com"
提示 Enter passphrase 时要输入密码
生成的 SSH Key 存放在 C:\Users\jc\.ssh
4 将 SSH Key 添加到 GitHub 账户里
点击链接 Account Settings | SSH Public Keys | Add another public key，将 id_rsa.pub 文件中的内容粘贴到 key 文本框中
5 第一次连接到 GitHub
$ ssh git@github.com
输入密码后如果显示信息里包含 You've successfully authenticated 表示连接成功。之后会在 C:\Users\jc\.ssh 建立一个名为 known_hosts 的文件
创建新库
进入 Dashboard Tab 页，点击 New Repository 按钮
免费的只能 Public！
克隆版本库
进入新建的库，点击 SSH 按钮，复制版本库地址，然后使用 git clone 命令即可
点击 Git Read-Only 按钮可获取只读权限的地址，克隆这个地址不需要密码
让 ssh-agent 替我们记住密码
1 把密码加密保存在硬盘上
$ ssh-keygen -p
2 修改配置文件，让 ssh-agent 随 git bash 启动
在 Git\etc\profile 中追加：
# lanch ssh agent
SSH_ENV="$HOME/.ssh/environment"
function start_agent {
echo "Initializing new SSH agent..."
/usr/bin/ssh-agent | sed 's/^echo/#echo/' > "${SSH_ENV}"
echo succeeded
chmod 600 "${SSH_ENV}"
. "${SSH_ENV}" > /dev/null
/usr/bin/ssh-add;
}
# Source SSH settings, if applicable
if [-f "${SSH_ENV}"]; then
"${SSH_ENV}" > /dev/null
#ps ${SSH_AGENT_PID} doesn't work under cywgin
ps -ef | grep ${SSH_AGENT_PID} | grep ssh-agent$ > /dev/null || {
start_agent;
}
else
start_agent;
fi
3 重启 Git Bash 控制台