# Anaconda

# 确认conda已安装

conda --version

# 更新conda版本

conda update conda

conda update anaconda

# 查看当前环境

conda info –envs

conda config --append channels conda-forge

=============================================================

# 创建新环境

conda create --name snowflakes *biopython*

### **NOTES**：snowflakes代指环境的名称，biopython指要在新环境中添加的软件包，使用当前环境使用的Python版本

* + 1. # 创建环境时指定Python版本

conda create --name bunnies **python=3** *astroid babel*

### **NOTES**：在创建环境指定软件包时，可以使用package\_name=version\_number 的方式来指定要使用的软件版本

# 复制环境

conda create --name flowers --clone snowflakes

# 1.1导出配置文件

conda env export --name snowflakes > snowflakes.yml

# 1.2根据配置文件导入环境

conda env create -f snowflakes.yml

# 查看所有已安装的软件包

conda list

# 2.1 查找软件包,罗列出所有可用的版本并在已经安装的版本前加\*

conda search beautifulsoup4

# 2.2 安装软件包

conda install --name beautifulsoup4=4.4.1

pip install XXX

# 更新软件包

conda update --name snowflakes beautifulsoup4=4.5.1

# 查找可用python版本，python对conda也是一个包

conda search --full-name python

conda create -n snakes python=3.4

# 卸载包，删除指定环境中的指定包

conda remove --name snowflakes biopython

# 卸载环境，--all参数表示移除环境中的所有软件包，即删除整个环境

conda remove --name snakes --all

# Jupyter

# Apollo

* Create a docker environment

docker ps # to verify docker works without sudo

bash docker/scripts/dev\_start.sh

bash docker/scripts/dev\_into.sh

* Build Apollo from source

# To get a list of build commands

./apollo.sh

# To make sure you start clean

./apollo.sh clean

# This will build the full system and requires that you have an nVidia GPU with nVidia drivers loaded

bash apollo.sh build

* Bootstrap start Apollo

# start module monitor

bash scripts/bootstrap.sh

* Download the demonstration loop and run it

# get rosbag note that the command download is required

bash ./docs/demo\_guide/rosbag\_helper.sh download

# You can now replay this demo "bag" in a loop with the '-l' flag

rosbag play -l ./docs/demo\_guide/demo\_2.0.bag

* Start a browser session and see the Dreamview user interface

# git

## **配置**

# git config –global user.name “XX”

# git config –gloaba user.email “[XX@163.com](mailto:XX@163.com)”

# **密钥配置**

# ssh-keygen -t rsa -C “邮箱”

# ssh-add id\_rsa

# 打开id\_rsa.pub复制到github的setting的SSH keys当中

# ssh [git@github.com](mailto:git@github.com) #测试

### **创建版本库**

mkdir testgit && cd testgit

git init

touch readme.md

git add readme.md

git commit -m "wrote a readme file"

git status

### 版本回退

git log

git log –pretty=oneline # git log --graph --pretty=oneline --abbrev-commit

git reset --hard HEAD^

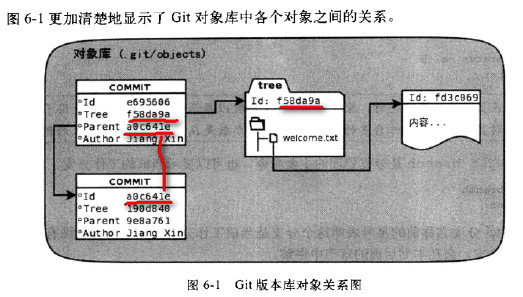
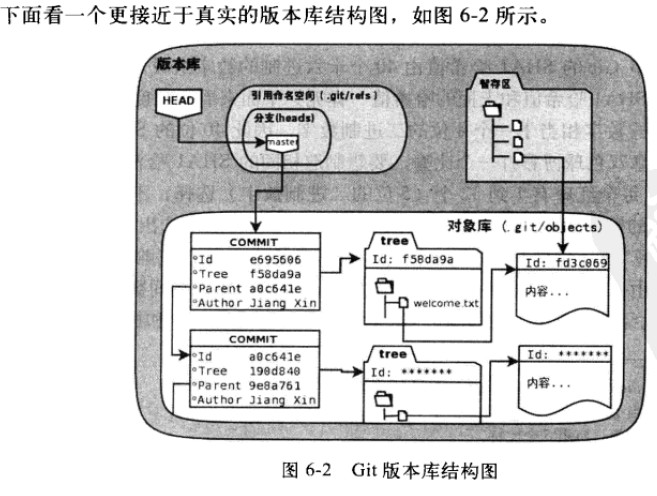
git reset --hard 2e70fdf # commit id

git reflog # Git提供了一个命令git reflog用来记录你的每一次命令

* 在 Git中，用HEAD表示当前版本，也就是最新的提交commit id，上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

### 工作区和暂存区

### 



* 工作区有一个隐藏目录.git，是Git的版本库。Git的版本库里存了称为stage（或者叫index）的暂存区，还有Git自动创建的第一个分支master，以及指向master的一个指针叫HEAD。

### **修改与撤销**

git diff HEAD -- readme.md # 查看工作区和版本库里面最新版本的区别

git checkout -- readme.md # 丢弃工作区的修改

### 删除文件

git rm readme.md

git commit -m "remove readme.md"

### 远程服务器

首先在github上new一个repo

git remote add origin https://github.com/xhzhang76/testgit.git

git push -u origin master

git remote -v *# shows URLs of each remote server*

git remote show origin(别名) *# gives more details about origin*

### 从远程库克隆

git clone git@github.com:xhzhang76/testgit.git

* Git支持多种协议，默认的git://使用ssh，也可以使用https等其他协议。使用https除了速度慢以外，最大的麻烦是每次推送都必须输入口令，在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。

### 创建与合并分支

git checkout -b dev # 等同于

git branch dev

git checkout dev

git branch # 命令查看当前分支

git checkout master

git merge dev

* 注意到git merge的信息里面可能有Fast-forward字样，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。不是每次合并都能Fast-forward

git branch -d dev

git branch -D <branch> #强行删除

git checkout -b branch-name origin/branch-name # 创建本地分支并与远程分支关联

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

### 分支管理策略

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。

git merge --no-ff -m "merge with no-ff" dev

### Bug分支

Git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

首先确定要在哪个分支上修复bug，如master分支：

git checkout master

git checkout -b issue-101

修复bug，提交：

git add readme.md

git commit -m "fix bug 101"

修复完成后，切换到master分支，并完成合并，最后删除issue-101分支：

git checkout master

git merge --no-ff -m "merged bug fix 101" issue-101

回到dev分支：

git checkout dev

git status

git stash list

工作现场还在，Git把stash内容存在某个地方了，恢复一下，有两个办法：

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；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

|  |
| --- |
| git stash pop |

再用git stash list查看，就看不到任何stash内容了。

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令

|  |
| --- |
| git stash apply stash@{0} |

### 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。

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

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

还可以通过-s用私钥签名一个标签：

|  |
| --- |
| git tag -s v0.5 -m "signed version 0.2 released" fec145a |

git tag可以查看所有标签。

用命令git show <tagname>可以查看某个标签的详细信息。

如果标签打错了，也可以删除：

|  |
| --- |
| git tag -d v0.1 |

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令git push origin <tagname>：

|  |
| --- |
| git push origin v1.0 |

或者，一次性推送全部尚未推送到远程的本地标签：

|  |
| --- |
| git push origin --tags |

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

|  |
| --- |
| git tag -d v0.9 |

然后，从远程删除。删除命令也是push，但是格式如下：

|  |
| --- |
| git push origin :refs/tags/v0.9 |

### 忽略特殊文件

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。

比如，让Git显示颜色，会让命令输出看起来更醒目：

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

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次git status都会显示Untracked files…，有强迫症的童鞋心里肯定不爽。

好在Git考虑到了大家的感受，这个问题解决起来也很简单，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。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>

当然也可以配置全局忽略的文件，这样就不用每个项目都加gitignore了：

|  |
| --- |
| git config --global core.excludesfile '~/.gitignore' |

### 配置别名

有没有经常敲错命令？比如git status？status这个单词真心不好记。

如果敲git st就表示git status那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后st就表示status：

|  |
| --- |
| git config --global alias.st status |

当然还有别的命令可以简写：

|  |
| --- |
| git config --global alias.co checkout  git config --global alias.ci commit  git config --global alias.br branch |

--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在撤销修改一节中，我们知道，命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：

|  |
| --- |
| git config --global alias.unstage 'reset HEAD' |

配置一个git last，让其显示最后一次提交信息：

|  |
| --- |
| git config --global alias.last 'log -1' |

甚至还有人把lg配置成了：

|  |
| --- |
| git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit" |

**配置文件**

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。

# Docker必须使用sudo操作的解决方案

### 操作步骤如下

* 1.创建docker组：sudo groupadd docker
* 2.将当前用户加入docker组：sudo gpasswd -a ${USER} docker
* 3.重启服务：sudo service docker restart
* 4.刷新docker成员：newgrp - docker