Git-Tutorial

@powered by 邓泽晖

github上不去?

- 挂梯子
- 修改hosts

在C:\Windows\System32\drivers\etc找到hosts文件（可以用记事本打开），然后随便找个dns查询工具查一下github.com的IP。添加到hosts之后在cmd中刷新一下dns：

```
ipconfig /flushdns
```

大部分时候都可以解决，不然就再换个ip。

```
#52.192.72.89              github.com
52.69.186.44               github.com
#13.114.40.48               github.com
```

这三个试一下就差不多了。github的ip隔一段时间会变化，所以时不时需要重新改一下。学了计网就知道为啥了（

Git学习资料

建议先看demo了解简单用法，资料作为文档等需要深度使用的时候再去查。

一个小游戏：对git有一定了解之后会发现这个对git的可视化和解释是非常好的。

廖雪峰的Git教程：随便找个类似的教程作为查询文档即可。

git开发流程

本地修改提交到远端demo

git中有几个概念：工作区、暂存区、本地仓库、远程仓库。

从github中clone一个库之后，会建立一个本地仓库。对其中的文件进行的修改会同步到工作区中，使用add指令可以将工作区中的修改保留到暂存区，然后再使用commit提交到本地仓库，最后用push指令提交到远端（origin）。

假设：

1. 有一个文件demo.cpp和一个文件夹/demo需要提交到远端
2. 当前在dev分支下

```
git add demo.cpp /demo      # 将文件（实际上是相对于上个版本的修改）从工作区添加到暂存区
git commit -m "你的提交信息" # 将修改提交到本地仓库
git pull origin dev         # 解决冲突
git push origin dev         # 推送到远端
```

单人开发

一般master/main分支作为主分支，发布到线上运行，所以一般不直接修改。要增加新功能的时候，一般检出一个develop/dev分支。开发完成后再新建一个release分支，在release上做测试，最后merge到dev和main上。

```
git checkout main
git checkout -b dev main
...
coding
...
git checkout -b release dev
...
testing
...
git checkout dev
git merge release
git push origin dev

git checkout main
git merge release
git push origin main

git tag v0.0.1
git push origin --tags

git branch
```

多人开发

多人开发时，流程和上面类似。区别就在于使用dev分支时，不要直接在dev上做修改，而是建一个自己的新分支比如[name]。

每个人建一个新分支，在上面进行修改，修改完之后merge到dev里。此时的dev可能和你之前使用的dev不一样了（已经有其他人更新过了），所以需要rebase（或merge）一下，然后再做合并。

如果rebase的时候有冲突，比如两个人同时修改了同一文件的同一行，那就需要手动解决冲突。然后rebase --continue。

举个例子，A从dev拉了一个分支feature后，进行了一些修改。同时B从dev拉了另一个分支，修改并提交到dev中了，此时：

```
git checkout dev
git pull origin dev
git checkout feature
git rebase dev
...
如果有冲突就解决冲突
...
git add .
git rebase --continue
git checkout dev
git merge --no-ff feature
git push origin dev
```

注意这个feature分支创建在本地即可，每次直接提交dev。

在多个feature同时merge到dev上之后，仍然可能存在逻辑错误，所以需要进一步测试。

Github+Hexo搭建个人网站

markdown编辑+hexo编译发布在[username].github.io上。教程

访问https://[username].github.io可以看到自己的博客站。

分支和提交

分支其实就是对某个提交的引用，所以多用分支！！多模块同步开发时非常有用。

首先是将当前暂存区中所有修改提交到本地仓库：

```
git commit -m "..."
```

然后是新建分支的操作

```
git branch [name]
git checkout -b [name]
```

查看当前所有分支

```
git branch
```

如果想同时使用提交树的两个不同的分支上的修改，可以用merge。比如将dev分支的所有修改加入到main分支中。（初学最好用这个）

```
git checkout main
git merge dev
```

合并分支也可以用rebase。rebase之后新提交只会指向一个旧提交，这样提交树会更清晰。

rebase就是把当前分支带来的所有历史提交建立在指定分支的所有提交的基础上。比如c<-c1<-main，c<-c2<-dev，想把dev分支的提交合并到main里面。

```
git checkout main
git rebase dev
```

结果是c<-c2<-dev，c-<c2<-c1'<-main。

HEAD和提交树

git将历史提交记录抽象成了一个多叉树，可以通过相关操作回溯到之前的提交。

HEAD

HEAD指向的提交也就是显示在当前工作区下的内容。

通常HEAD会指向一个分支，实际上是间接指向提交：HEAD->main->commit_id。

```
git checkout main
```

也可以不用分支管理，直接指向提交：HEAD->commit_id。

```
git checkout commit_id
```

log指令可以查看在git上做的历史提交记录，可以理解为是查看commit_id，一般是个$fed2da64c0efc5293610bdd892f82a58e8cbc5d8$这样的40位的哈希值。

知道此哈希值之后就可以找到对应的commit，当然Git还是比较人性化的，一般输入哈希值的前4位就能锁定对应的commit_id。

相对引用

相对引用主要基于^操作来实现，表示指定提交记录的父结点。^操作支持迭代使用，比如main^^等。

如果需要找到main分支的上一个提交：

```
git checkout main^
```

^只能用于分支名或HEAD。

如果需要多次在提交树上向上移动的话，^操作就显得特别麻烦，git也提供了多步移动的操作符~。使用的时候和^类似，就是需要在~后面加上待回溯的提交数。

```
git checkout main~2
```

强制修改分支位置

之前提到的几个操作都是移动HEAD指向的版本，如果想移动某个分支的版本的话：

```
git branch -f main HEAD~3
```

撤销变更

如果不仅仅想回溯分支的提交，同时还想删除回溯到的分支后面的所有提交，当其完全没有提交过。可以使用reset指令。

reset之后，c2的commit记录删除了，但是所做的变更还会存在于暂存区中。

```
git reset HEAD^
```

还有一个类似功能的指令revert。revert就是创建一个新的提交，将回溯的提交作为一个新的提交加入到提交树的下方。

```
git revert HEAD^
```

远程仓库

远程仓库（Repo）就是你在github上看到的那些文件。

远程仓库下载到本地

可以选择https和ssh两种方式，一般来说ssh不用多次输密码，且速度较快。

```
git clone [仓库的网址或ssh链接]
```

如果是在github上新建了一个仓库，想跟本地某个文件夹关联起来。

```
git init
git remote add origin [仓库的网址或ssh链接]
```

跟先clone再修改的效果一样。

将本地修改推送到远端

远端的分支可以用$git\ branch\ -a$查看，通常是/remotes/origin/...的形式。如果想将本地的仓库的当前分支推送到远端的main分支。

```
git push origin main
```

从远端获取最新修改

首先是将远程分支同步到本地的/remotes/origin/...中，然后再做一个merge。这两个操作可以分步进行，也可以用一个指令同时进行。

先看一下分步：

```
git fetch
git merge /remotes/origin/main
```

再看一下一步到位的操作：

```
git pull origin main
```

具体效果完全一样。

如果存在冲突，可以用--rebase选项

```
git pull --rebase
```

Tag

tag主要用于发布版本的管理，发布之后可以为git打上v0.0.1之类的标签。

Tag和Branch的区别

tag对应某次commit。

branch对应一系列的commit。

举个例子。如果想在main分支上做修改，可以先检出一个新分支dev，在dev上不断commit。觉得比较完善了之后，就可以对最后一个commit打一个tag。

也就是说tag用来标记开发过程中若干个commit中比较重要的那些。

一般来说，需要改动代码时用branch，只需查看是用tag（当然也可以把某个tag来出来做一个新分支然后开发）。

tag常用方法

```
git tag v0.0.1 [commit-id]
```

用来创建本地tag，默认（也就是不加commit-id）将为当前分支的最后一个commit创建tag（一般默认就够了）。

```
git tag -l
```

可以查看当前已有的tag。

```
git tag -d v0.0.1
```

删除本地的tag。

```
git tag v0.1 -m "tag messages"
```

在打tag的时候添加提示信息。

```
git push origin v0.0.1
```

将标签提交到远程库。

```
git push origin -d v0.0.1
```

删除远程库中的tag。

```
git show v0.0.1
```

可以查看该版本对应commit所做的修改。

```
git checkout v0.0.1
```

切换标签。