git

1.简介

- git是目前最先进的分布式版本控制系统，不必联网、强大分支管理等
- CVS、SVN是免费的集中式的版本控制系统，但其速度慢、需联网。

1.1安装

- `linux:`

`sudo apt-get install git`

- `mac:`

  安装Xcode->"Preferences"->"Download"->"Command Line Tools"

- `windows:`

  下载安装->在开始菜单里找到“Git”->“Git Bash”，蹦出命令行窗口->安装成功->设置：

  ```
  $ git config --global user.name "Your Name"
  $ git config --global user.email "email@example.com"
  ```

1.2创建版本库

版本库又名仓库repository，可理解为一个目录，该目录下的所有文件都能被git管理跟踪，以便任何时候都可以追踪历史。

创建版本库

- 创建一个空目录
- 到其目录下，通过`git init`命令变成git可以管理的仓库，可看到`.git`目录

把文件添加到版本库

所有的版本控制系统只能跟踪文本文件的改动，二进制格式文件只能跟踪大小变换

- 编写一个文件readme.md
- 使用命令`git add readme.md`把文件**添加**到仓库
- 使用命令`git commit -m "readme"`提交**提交**到仓库，`-m`后面输入的是本次的提交说明

为什么添加问价需要`add、commit`共两步？因为`commit`可以一次提交很多文件，所有可以多次`add`再`commit`

2.时光穿梭

2.1版本回退

- `git status`查看仓库当前状态
- `git diff filename`查看具体修改什么内容

每当觉得文件修改到一定程度的时候，就可以保存一个“快照”，这个快照在Git中被称为`commit`

- `git log`显示从最近到最远的提交日志，加上`--pretty=online`简化输出
  - `1094adb...`是`commit id`版本号，和SVN不同，其不是递增的数字，SHA1计算出来的一个非常大的数字，用十六进制表示。
  - 为什么`commit id`需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

Git必须知道当前版本是哪个版本，在Git中，用`HEAD`表示当前版本，也就是最新的提交`1094adb...`，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`。

- `git reset --hard commit_id`回退版本：`git reset --hard HEAD^`
- `git resert --hard commit _id`回到未来：`git reset --hard 1094a`版本号没必要写全
- `git reflog`用来记录每一次命令，可用来确定要回到未来哪个版本

2.2工作区和暂存区

git个其他版本控制系统SVN的一个不同指出就是有暂存区的概念。

工作区

就是在电脑能看到的目录

版本库

工作区有一个隐藏目录`.git`这个不算工作区，而是git的版本库。Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支`master`，以及指向`master`的一个指针叫`HEAD`。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

- 第一步是用`git add`把文件添加进去，实际上就是把文件修改添加到暂存区；
- 第二步是用`git commit`提交更改，实际上就是把暂存区的所有内容提交到当前分支。
- `git add`命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行`git commit`就可以一次性把暂存区的所有修改提交到分支。

2.3管理修改

为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

- 第一次修改 -> `git add` -> 第二次修改 -> `git commit`

Git管理的是修改，当你用`git add`命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，`git commit`只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

- 第一次修改 -> `git add` -> 第二次修改 -> `git add` -> `git commit`

第二次修改提交了。

2.5 撤销修改

`git checkout -- file`

可以丢弃工作区的修改，这里有两种情况：

- 一种是`readme.txt`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；
- 一种是`readme.txt`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。

总之，就是让这个文件回到最近一次`git commit`或`git add`时的状态。

`git reset HEAD <file>`

修改只是添加到了暂存区，还没有提交.用命令`git reset HEAD <file>`可以把暂存区的修改撤销掉（unstage），重新放回工作区

summary

场景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`。

场景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>`，就回到了场景1，第二步按场景1操作。

场景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，参考版本回退一节，不过前提是没有推送到远程库。

2.6删除文件

从版本库中删除文件，那就用命令`git rm`删掉，并且`git commit`

若删错了，因为版本库里还有呢，所以可以`git checkout -- file`很轻松地把误删的文件恢复到最新版本

3.远程仓库

3.1添加远程库

在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作。

- 本地git仓库关联远程仓库

```
git remote add origin git@server-name:path/repo-name.git
git remote rm origin #删除已有的远程库
```

- 本地初次推送若已经有ssh-key，并添加到GitHub账户SSH Keys中，则直接进行推送

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

- 若本地无ssh，向远程仓库推送会报错，提示无权限，此时本地cmd指令

```
ssh-keygen -t rsa -C "邮箱地址"
```

完成操作可生成ssh，打开生成的`id_ras.pub`文件，将生成ssh密匙存入远程仓库帐号，即可向远程仓库推送。

- 验证`ssh -T git@github.com`,显示`Hi ...! You've successfully authenticated, but GitHub does not provide shell access.`

3.2从远程库克隆

- 用命令`git clone`克隆一个本地库，例如：

```
$ git clone git@github.com:userName/projectName.git
```

- github允许你本地仓库有的东西，远程仓库里没有，但不允许远程仓库有的东西，你本地仓库没有。同步命令：

```
$ git pull --rebase origin master
```

3.3summary

- 克隆一个仓库，首先必须知道仓库的地址，然后使用`git clone`命令克隆。

- Git支持多种协议，包括`https`，但`ssh`协议速度最快。

4.分支管理

假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

SVN的分支管理速度很慢，Git的分支管理速度极快。

4.1创建与合并分支

- 首先，我们创建`dev`分支，然后切换到`dev`分支：

```
$ git checkout -b dev
```

`git checkout`命令加上`-b`参数表示创建并切换，相当于以下两条命令：

```
$ git branch dev
$ git checkout dev
Switched to branch 'dev'
```

- 然后，用`git branch`命令查看当前分支：

```
$ git branch
* dev
  master
```

- 在该分支上进行正常提交修改，不会影响`master`分支上的 内容。
- `dev`分支的工作完成，我们就可以切换回`master`分支：

```
$ git checkout master
Switched to branch 'master'
```

- 将`dev`分支合并到当前分支

```
$ git merge dev
```

- 删除分支

```
$ git branch -d dev
```

summary

- 查看分支：`git branch`
- 创建分支：`git branch <name>`
- 切换分支：`git checkout <name>`或者`git switch <name>`
- 创建并切换分支：`git checkout -b <name>`·或者`git switch -c <name>`\
- 合并分支到当前分支(记得切换至当前分支如master)：`git merge <name>`

- 删除分支：`git branch -d <name>`

4.2 解决冲突

- `git merge <name>`合并时，`master`分支和`feature1`分支各自都分别有新的提交,比如有相同的readme文件但内容不同，会发生冲突。查看冲突内容，比如：

```
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
Creating a new branch is quick AND simple.
>>>>>>> feature1
```

- Git用`<<<<<<<`，`=======`，`>>>>>>>`标记出不同分支的内容，我们修改对应冲突位置的内容使其相同，再次提交即可。
- 用`$ git log --graph --pretty=oneline --abbrev-commit`命令可以看到分支合并图。

4.3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。

准备合并`dev`分支，请注意`--no-ff`参数，表示禁用`Fast forward`：

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

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。合并后，我们用`git log`看看分支历史。

分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

- 首先，`master`分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；
- 那在哪干活呢？干活都在`dev`分支上，也就是说，`dev`分支是不稳定的，到某个时候，比如1.0版本发布时，再把`dev`分支合并到`master`上，在`master`分支发布1.0版本；
- 你和你的小伙伴们每个人都在`dev`分支上干活，每个人都有自己的分支，时不时地往`dev`分支上合并就可以了。

4.4Bug分支

- Git还提供了一个`git stash`功能，可以把当前工作现场“储藏”起来得到干净工作区，等以后恢复现场后继续工作
- 回到主分支创建分支修复bug,合并后删除hug分支
- 用`git stash list`命令看看工作现场，恢复现场
  - 一是用`git stash apply stashid`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。
- 在master分支上修复的bug，想要合并到当前dev分支，可以用`git cherry-pick <commit>`命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

4.5Feature分支

开发新功能

- 开发一个新feature，最好新建一个分支；
- 如果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。

4.6多人协作

推送分支

Git自动把本地的`master`分支和远程的`master`分支对应起来了，并且，远程仓库的默认名称是`origin`.

- `git remote -v`显示远程库详细信息

```
$ git remote -v
origin  git@github.com:michaelliao/learngit.git (fetch)
origin  git@github.com:michaelliao/learngit.git (push)
```

上面显示了可以抓取和推送的`origin`的地址。如果没有推送权限，就看不到push的地址。

- 推送分支，就是把该分支上的所有本地提交推送到远程库`git push origin master`;推送其他分支：`git push origin dev`
  - `master`分支是主分支，因此要时刻与远程同步；
  - `dev`分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
  - bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
  - feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

抓取分支

你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用`git pull`把最新的提交从`origin/dev`抓下来，然后，在本地合并，解决冲突，再推送

- `git branch --set-upstream-to=origin/dev dev`指定`dev`分支和远程`origin/dev`分支的链接
- 再`git pull`
- `git commit -m "..."`解决冲突并提交
- `git push`

因此，多人协作的工作模式通常是这样：

1. 首先，可以试图用`git push origin <branch-name>`推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用`git pull`试图合并；
3. 如果`git pull`也失败，原因时没有指定分支和远程分支的链接，使用`git branch --set-upstream-to=origin/dev dev`
4. 如果合并有冲突，则解决冲突，并在本地提交；
5. 没有冲突或者解决掉冲突后，再用`git push origin <branch-name>`推送就能成功！

如果`git pull`提示`no tracking information`，则说明本地分支和远程分支的链接关系没有创建，用命令`git branch --set-upstream-to <branch-name> origin/<branch-name>`。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

summary

- `git remote -v` 查看远程库信息
- 本栋新建的分支如果不推送到远程，对其他人不可见
- 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
- 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` 本地和远程分支名称最好一致
- 建立本地分支和远程分支的关联，使用 `git branch --set-upstream-to=origin/dev dev` 建立连接
- `git pull` 抓取远程分支，有冲突就先处理冲突

4.7 rebase

- rebase操作可以把本地未push的分叉提交历史整理成直线；
- rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比。

标签管理

- 命令 `git tag <tagname> <commit_id>` 用于新建一个标签，默认为 `HEAD`，也可以指定一个commit id；
- 命令 `git tag -a <tagname> -m "blablabla..."` 创建带有说明的标签，用 `-a` 指定标签名，`-m` 指定说明文字
- 命令 `git tag` 可以查看所有标签。

操作标签

- 命令 `git push origin <tagname>` 可以推送一个本地标签；
- 命令 `git push origin --tags` 可以推送全部未推送过的本地标签；
- 命令 `git tag -d <tagname>` 可以删除一个本地标签；
- 命令 `git push origin :refs/tags/<tagname>` 可以删除一个远程标签。

5.Gitee

一个本地库能不能既关联GitHub，又关联Gitee呢? git给远程库起的默认名称是 `origin`，如果有多个远程库，我们需要用不同的名称来标识不同的远程库

- 先删除已关联origin远程库 `git remote rm origin`
- `git remote add github git@githun.com:username/proj.git` 先关联github远程库，这个远程库名为 `github`,不叫 `origin`
- 关联Gitee的远程库 `git remote add gitee git@gitee.com:username/proj.git`
- `git remote -v` 可看到两个远程库
- `git push github master` 推送到 `github`，`git push gitee master` 推送到 `gitee`

6.自定义Git

我们已经配置了 `user.name` 和 `user.email`，实际上，Git还有很多可配置项

- `git config --global color.ui true` 显示颜色

6.1忽略特殊文件

- 忽略某些文件时，需要编写 `.gitignore`；
- `.gitignore` 文件本身要放到版本库里，并且可以对 `.gitignore` 做版本管理!

6.2配置别名

告诉Git，以后`st`就表示`status`：

```
$ git config --global alias.st status
```

`--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。

6.3搭建Git服务器

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的`apt`命令就可以完成安装。

- 第一步，安装`git`：`sudo apt-get install`
- 第二步，创建`git`用户：用来运行`git`服务器: `sudo adduser git`
- 第三步，创建证书登录：收集所有需要登录的用户的公钥，就是他们自己的`id_rsa.pub`文件，把所有公钥导入到`/home/git/.ssh/authorized_keys`文件里，一行一个。
- 第四步，初始化Git仓库：先选定一个目录作为Git仓库，假定为`/srv/sample.git`，在`/srv`目录下输入命令：`sudo git init --bare sample.git`。Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以`.git`结尾
- 把owner改为`git`：`sudo chown -R git:git sample.git`
- 第五步，禁用shell登录：

  出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑`/etc/passwd`文件完成。找到类似下面的一行：

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

  改为：

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

  这样，`git`用户可以正常通过ssh使用git，但无法登录shell，因为我们为`git`用户指定的`git-shell`每次一登录就自动退出。

- 第六步，克隆远程仓库：

  现在，可以通过`git clone`命令克隆远程仓库了，在各自的电脑上运行：

  ```
  $ git clone git@server:/srv/sample.git
  Cloning into 'sample'...
  warning: You appear to have cloned an empty repository.
  ```

  剩下的推送就简单了。