Guide

repository (仓库)

repository 简介

- repository 是一个数据结构，用于储藏project的信息，即metadata。
- repository 一般存放在project的根目录下，以.git命名的隐藏文件中

repository 安装

- 在linux下安装
  - Ubuntu: $ apt-get install git
  - Mac: $ brew install git
  - Windows: 去官网下载吧LOL

预备案例

- mkdir testGit------>在当前路径下创建一个用于测试的testGit文件夹
- cd testGit-------> 切换到文件夹下
- echo HelloWorld > file1.txt ------>将字符串'HelloWorld'写入file.txt文件中
- git init ---------> 在当前文件夹（testGit）下初始化git 的repository
- tree git 用tree命令看.git 中的文件结构

```
chenruideMacBook-Pro:Github chenrui$ tree .git
.git
├── HEAD
├── branches
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

- 所有的命令包括二级命令都可以通过输入 **--help**参数来查询

.git/objects文件夹

- objects：git将所有project中的信息存储在objects 子文件夹下
  - 在初始化刚开始时objects 文件夹下只有两个空目录，表明当前project还没有任何添加和提交
  - 此时，虽然project根目录下有一个文件file1.txt，但是没有添加到git下，因此objects中还是空
  - 返回project根目录
  - 输入 **git add file1.txt**
  - 此时objects文件夹下变为
  - ```
    objects/
    ├── 3d
    │   └── a1ec26e9c8512eae062868a9ff9bae47e5625b
    ├── info
    └── pack
    ```
  - 此时objects文件夹下多出一个3d文件夹，其下有一个文件，名为：a1ec26e9c8512eae062868a9ff9bae47e5625b
  - 用命令file 该文件，观察该文件的信息为
  - ```
    chenruideMacBook-Pro:3d chenrui$ file a1ec26e9c8512eae062868a9ff9bae47e5625b
    a1ec26e9c8512eae062868a9ff9bae47e5625b: VAX COFF executable not stripped - version 15360
    ```
  - VAX COFF 可执行文件，是linux下一种虚拟地址映射文件
  - 实际上 执行git add file1.txt 会在objecs目录下添加一个blob object。
  - 该blob object 没有以文件名：a1ec26e9c8512eae062868a9ff9bae47e5625b来命名，而是以3da1ec26e9c8512eae062868a9ff9bae47e5625b命名（目录名+文件名）。其中3d是目录名，a1ec26e9c8512eae062868a9ff9bae47e5625b是文件的文件名。
  - 3da1ec26e9c8512eae062868a9ff9bae47e5625b 这个对象名是根据file1.txt文件中的内容通过SHA1散列得到的
  - git就是用这种散列，配合这种二级目录创建规则，来管理project中的文件的，方便快捷
  - 注意：文件file1.txt没有存储在repository中，文件file1.txt在repository中的存储是一个经过压缩后的对象。objects下的二级目录名管理只是查找该对象的一种方式，但无法通过cat这种方式打开。
  - 要想观察该对象可以用命令 git cat-file -p 3da1ec26e9c8512eae062868a9ff9bae47e5625b 来观察
  - **git cat-file** 命令
    - -p(print 意思) 根据文件类型，合理显示文件内容
    - -s 显示文件大小
    - -t 显示文件类型 (一般回事blob类型)
    - --help 显示cat-file 命令的帮助
  - 注意：如果你在同一个文件夹下，创建内容相同的两个文件，并add到git中，objects保持不变，因为objects下的blob对象是根据文件内容通过SHA1散列过来到
  - 例如：
    - 先执行cp file1.txt copied_file.txt
    - 然好添加文件 git add copied_file.txt
    - 在用tree 命令看objects文件夹，-->文件夹没变化
    - ```
      chenruideMacBook-Pro:Github chenrui$ git add copied_file.txt
      chenruideMacBook-Pro:Github chenrui$ tree .git/objects/
      .git/objects/
      ├── 3d
      │   └── a1ec26e9c8512eae062868a9ff9bae47e5625b
      ├── info
      └── pack
      ```
    - 同理，将这个文件拷贝到子目录下（同名或不同名），objects依然没变化。
    - 结论：根据文件内容SHA1文件到objects下
    - **如果手动删除objects目录下的生成文件，再用git add file命令，不会再进行SHA1 散列了-->所以不要手动删objects下的文件:)**
      - 原因可能是因为第一次add文件后文件就会被cache，然后写入objects。当你删除objects下文件后，git检测cache，你的文件没被修改而且还在cache中，因而不会被写入objects
      ```

中。
    - 解决方法1：修改源文件file1.txt，这样git会检测到文件修改并重新cache和SHA1到一个新的不一样的:) objects子文件中
    - 解决方法2：用命令git rm --chached file1.txt 则文件就会返回未被跟踪的状态，如下

```
chenruideMacBook-Pro:Github chenrui$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   copied_file.txt
        new file:   dir/copied_file1.txt
        new file:   longtext.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1.txt
```

    - 这样以后在用git add file1.txt 就会再次生成blob对象

git原理

- 由上面就知道，git每add一个文件，无论是新建文件还是被修改的文件，都会SHA1散列文件内容到objects下，这样很久很久以前的，被修改前的文件的内容都会保存在objects下不同路径下，不同文件中。因而就实现了保存所有历史版本的功能。当然SHA1散列还有压缩功能。

commit命令

- 如上所说，add只是对文件内容进行SHA1散列，并放入objects下。但是并没有对文件名和文件内容进行关联
- commit命令就是解决关联问题
- **git commit -m "commit file1.txt"** 通过执行该命令，则就把file1.txt和objects下的相应内容进行了关联
- 如果不带参数-m git会强行让你进入commit编辑模版，即每次commit必须留下备注，没有备注无法commit。
- 执行完了发现objects下多了两个文件。
- 通过git cat-file 命令来观察这两个新文件
    - 其中一个为关联文件或叫**tree文件**（也就是记录每一次commit的文件），其中列出了关联信息

```
chenruideMacBook-Pro:Github chenrui$ tree .git/objects/
.git/objects/
├── 09
│   └── 9e3427fc491cb39bf67f2ea6ce65b09c90b142
├── 15
│   └── 78822b737a6411ff3866aae3197f7e8e6256e2
├── b5
│   └── f5b8869d9cd1eb3d998a3c05cf222efadf1996
├── info
└── pack

5 directories, 3 files
chenruideMacBook-Pro:Github chenrui$ git cat-file -p 1578822b737a6411ff3866aae3197f7e8e6256e2
100644 blob 099e3427fc491cb39bf67f2ea6ce65b09c90b142    file1.txt
```

        - 从图中看出，用-p参数看到文件中的内容：第一个参数为linux文件模式100644（表示普通文件，644表示文权限件，此处去看linux文件代码）；第二个参数为关联文件内容的blob对象名，第三个为文件路径和文件名（相对project根目录）
        - 可见文件名和内容的关联也是存储于objects下的
    - 另一个文件为commit文件

        - 其中首先给出了绑定的commit文件（即上面那个在objects下的关联文件）
        - commiter信息
        - commit时的备注
- 修改了文件后再次提交，则在objects下会有6个文件：2个commit文件，2个comment文件，2个内容文件。
- 命令参数git commit -a -m "comment" 自动把所有被修改（新创建文件不算）的文件，添加并提交commit
- **最近提交的记录会在.git/refs/master文件中，其中记录了最近commit的blob对象名**
    - 如果你不是第一次提交，则用cat-file上面那个文件你会看到一行parents，parents指向本次提交的上一次提交，因而所有的提交的逻辑结构就是一个指向前驱的链表

```
chenruideMacBook-Pro:.git chenrui$ git cat-file -p 0bf8c959935315657b3831fb2e110
318262e226f
tree ad606f09f61eb1a42fc1c670ac5f9ae172e7836c
parent b5f5b8869d9cd1eb3d998a3c05cf222efadf1996
author swordkeeper <chenrui.apple@gmail.com> 1522126468 +1100
committer swordkeeper <chenrui.apple@gmail.com> 1522126468 +1100

first modify
```

    - 因而所有的commit文件（tree文件）就是一个链表.git/refs/head/master---->最近被修改---->parents（前一次修改）----->parents----->none

branch 分支

- 到现在，所有的commit都在master分支下
- 建立分支就是在上面所说的commit链条上选一个点，复制那个点的commit blob对象号，并用你自己的commit链条，指向它。
- 你可以直接复制那个blob对象好，也可以通过git log 命令来查看所有commit记录,如图

```
chenruideMacBook-Pro:Github chenrui$ git log
commit 0bf8c959935315657b3831fb2e110318262e226f (HEAD -> master)
Author: swordkeeper <chenrui.apple@gmail.com>
Date:   Tue Mar 27 15:54:28 2018 +1100

    first modify

commit b5f5b8869d9cd1eb3d998a3c05cf222efadf1996
Author: swordkeeper <chenrui.apple@gmail.com>
Date:   Tue Mar 27 15:23:02 2018 +1100

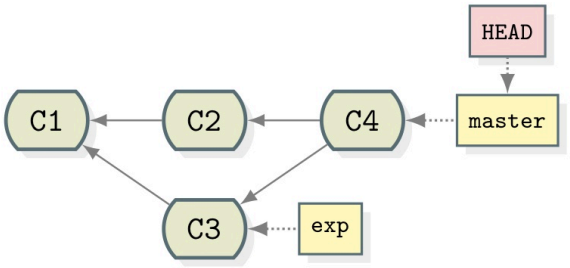
    Commit first time
```

- 最上面表示链表头，即master的head指向最近依次提交，以此类推
- **git branch newline(新分支名) b5f5b8869d9cd1eb3d998a3c05cf222efadf1996** 命令就创建了一个名为newline并且head指向 b5f5b8869d9cd1eb3d998a3c05cf222efadf1996 的新分支。
- 注意：创建新分支不会在objects下创建对象，只会在refs/heads下添加一个新的分支，其中指向你选择的commit点
- refs/heads下记录的是所有分支，而.git/HEAD下则记录的是当前正在使用的分支信息
- 通过命令，git status 或是 git branch 都可以看到当前正在使用的分支。
- **切换分支用 git checkout 分支名** 命令来实现；切换的前提是，本分支没有对其他分支包含的文件进行改动或删除，否则切换分支前必须commit
- **一条分支一旦commit了某个修改或是新建文件以后，另外的分支就看不到这种改变了。**
    - 例子，两条分支master和newline
    - 在newline分支下创建新文件 file2
    - 切换分支到master，用ls可以看到这个新建的文件(原因是，newlinew新建文件没有对master已知的文件进行改动)
    - newline git add 该文件，master仍能看到并也添加到cache中了。
    - 一旦newline commit， master就看不到这个文件了（神奇之处）

融合merging

- 融合merge使用命令 git merge newline

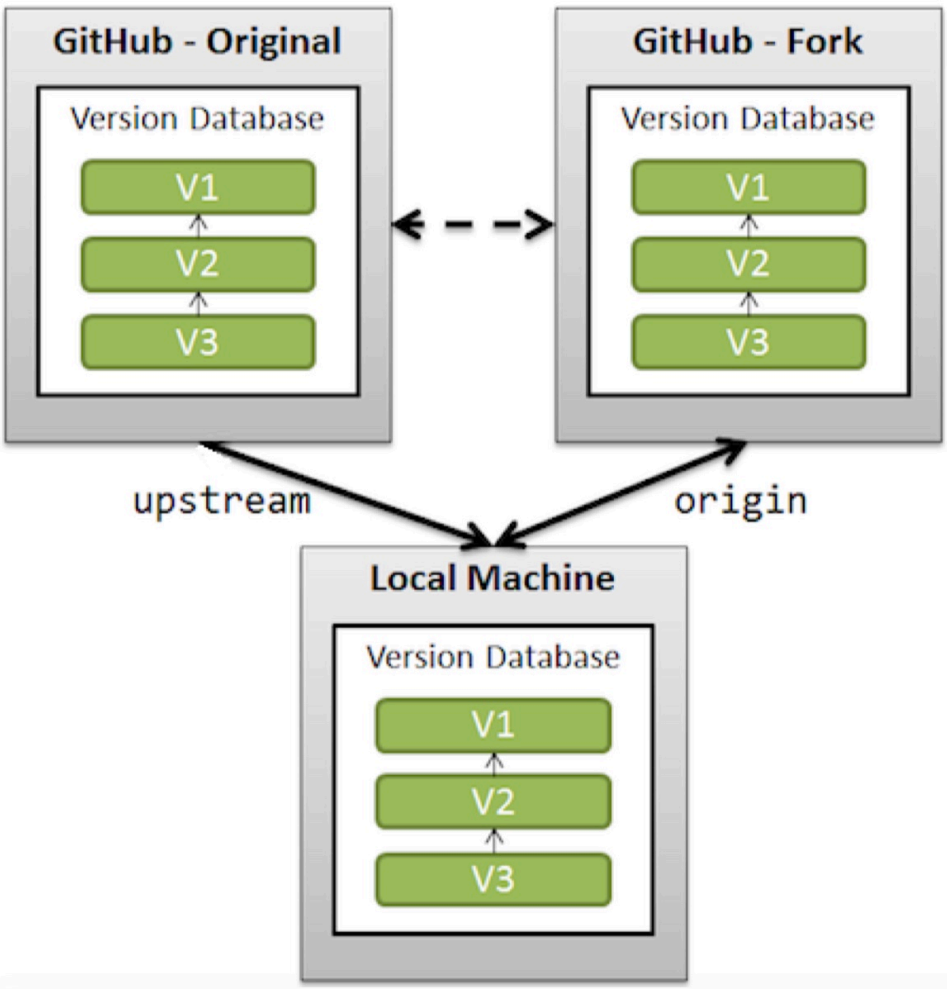
- 首先你要切换到master线上，这样融合后master被改变，newline线不变
- 融合后会标注冲突文件。
- <<<<< HEAD
  sdsd //本分支里的版本
  ====== 分割线
  cscsdcscs //newline分支里的版本
  》》》》》 newline
- 当修改了冲突后，再commit后，新的commit对象里会有两个parents。类似下图



-
- C4提交同时指向C3和C2
- 这样如果在执行 git branch -d 分支名，则删掉了本次分支，所有版本又回归主线master

远程repository

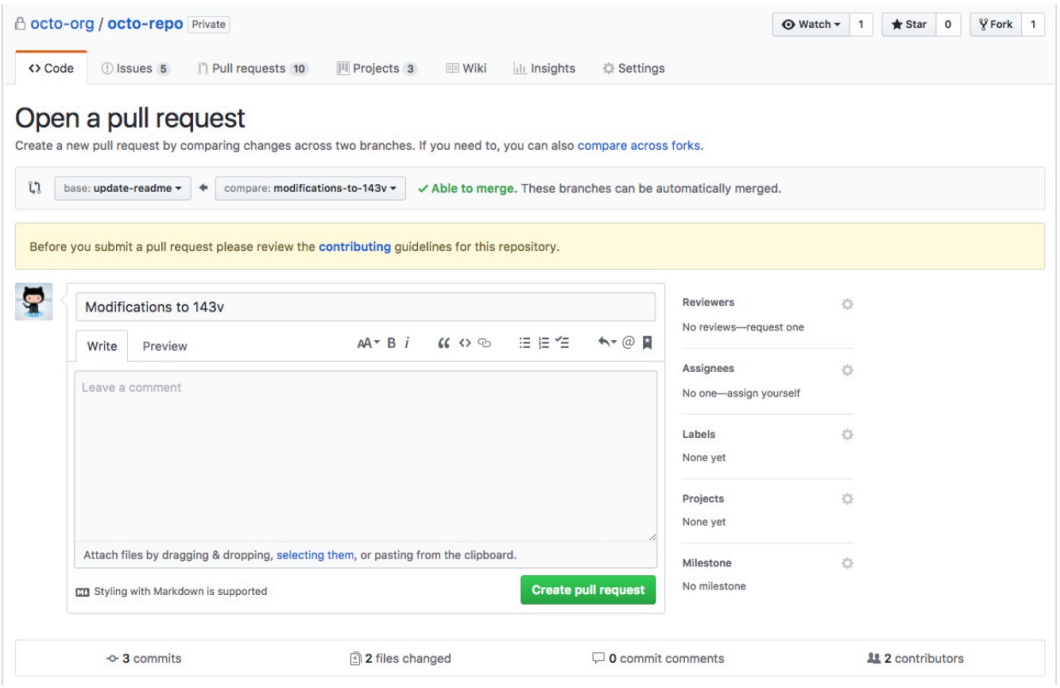
- 下载远程project
  - 使用命令git clone URL 通长URL以.git 结尾
  - 这样就会把一个远程的包含git的project拷贝到当前目录下
- 上载project
  - 切换到project目录下，输入命令git remote add [远程服务器名（自己七个名字用于标识URL）默认叫origin] [远程URL]，这样就建立了你起的名字和远程URL之间的映射
  - 通过命令git remote -v 来查看本project关联的所有远程URL包括给他们起的名字
  - **git push -u origin master** 该命令意思是，将master分支推送到名为origin的远程URL地址，这样就是实现了上传
    - -u参数表示对本次推送做追踪记录
    - --all参数表示推送所有分支
- 下载project的更新
  - 命令为**git fetch origin**，这样就会把origin所指url的project更新新全部拿到本地
  - 区别本地的master分支和远程fetch回来的master分支则看前缀。远程fetch下来的master名字为origin/master
  - git 会自动在.git/refs下添加一个文件叫remotes，其中包含origin/master分支头信息。
  - 如果分支没有新的commit，本地head和origin/master下的head会指向同一个blob对象
  - 如果远程origin/master有跟新，则origin/master下的head会变，head会指向新的blob对象（这个新的对象已经被创建到objects下）
  - 使用cat-file打开这个远程下载下来的新head所指的blob对象，会发现他的tree 也就是这个头所指的内容（content-blob对象）不存在。
  - 同时输入git status命令，命令会提示你，你本地master落后（behind）origin/master几次commit
  - 然后你再在本地master下merge，origin/master 对比查重后这样就可以跟新本地project了
  - **git pull origin** 命令相当于 git fetch 和merge的结合，它会把任何origin中的变化直接融合到本地分支中
- 关系图
  - 最好的方式是对源repository（拷贝源码处）命名为上游upstream
  - 自己推送的git中心repository（自己的远程管理中心）命名为origin，三者关系图如下



- 这样就能很好的管理三方代码。
- 当然另一种三方关系可以是：先fork其他repository到你自己的repository，然后在clone你自己的repository到本机。不过还是推荐第一种关系

好的合作流程

- 如果你们小组合作开发一个project
  a. 将远程master作为主版本进行控制
  b. 任何人添加，修改，测试都需要从主master里面进行分支，然后对分支进行操作
  c. 如果一个人需要向远程master里提交修改，则需要发送**pull-request**来通知其他小组成员，pull-request实际就相当于修改前的声明

d. 

e. pull-request会通知其他成员你要对主master进行了修改。小组成员也可以通过这里的讨论板对你所做的修改进行讨论。

f. 在进行检查代码并讨论后，主要检查人员可以选择拒绝本次pull-request（驳回更改请求，让其继续修改）也可以通过本次pull-request

g. 在通过pull-request以后，你就可以push本地master到远程的主master上了（也可以在远程合并，即远程master合并你远程的分支）