## 1.1Git结构

工作区（写代码）——》git add到暂存区（临时存储）——》git commit到本地库（历史版本）

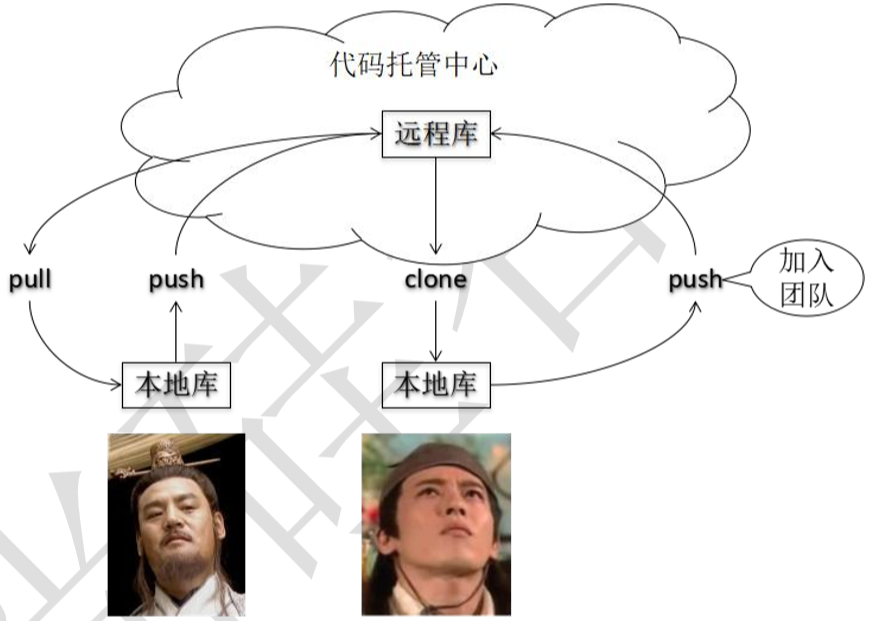
## 1.2Git和代码托管中心

代码托管中心的任务：维护远程库。

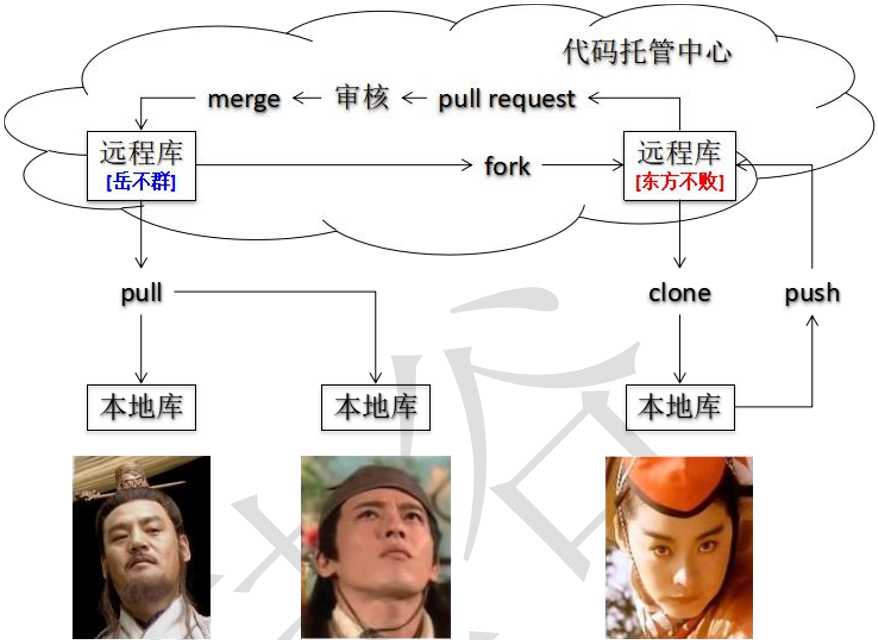
* 局域网环境下：
  + GitLab服务器
* 外网环境下：
  + GitHub
  + 码云

## 1.3本地库和远程库

* 团队内部协助



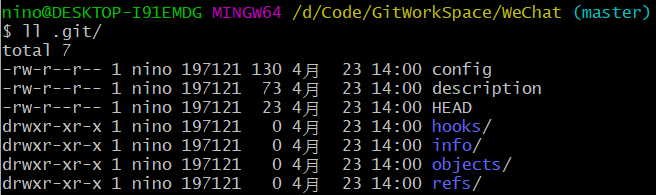
* 跨团队协作



* + fork操作：复制一份成为为一个新的远程库。
  + pull request操作：发起一个拉取请求。
  + merge操作：审核之后拉取。

## 1.4本地库初始化

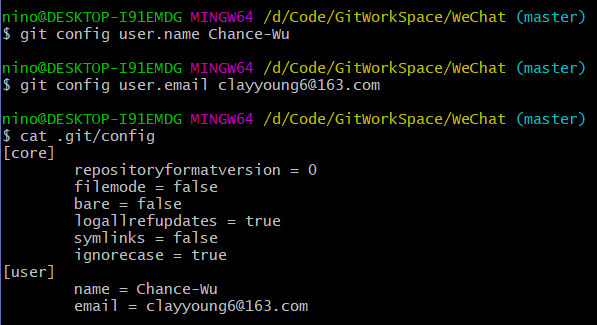
### 1.4.1git init



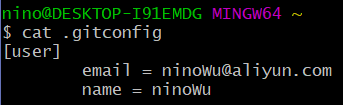
注：.git目录中存放的是本地库相关的子目录和文件，不能删除也不要胡乱修改。

### 1.4.2设置签名

* 形式：
  + 用户名：Chance-Wu
  + Email地址：[clayyoung6@163.com](mailto:clayyoung6@163.com)
* 作用：区分不同开发人员的身份
* 辨析：这设置的签名和登录远程库（代码托管中心）的账号、密码没有任何关系
* 命令：
  + 项目级别/仓库级别：仅在当前本地库范围内有效。
    - git config user.name Chance-Wu
    - git config user.email [clayyoung6@163.com](mailto:clayyoung6@163.com)



* + 系统用户级别：登录当前操作系统的用户范围。
    - git config --global user.name ninoWu
    - git config --global user.email [ninoWu@aliyun.com](mailto:ninoWu@aliyun.com)



* + 优先级别：
    - 就近原则
    - 如果只有系统用户级别的签名，就以系统用户级别的签名为准
    - 二者都没有不允许

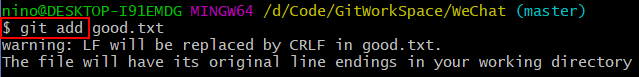
## 1.5基本操作

### 1.5.1查看工作区，暂存区状态

git status

### 1.5.2将工作区的“新建/修改”添加到暂存区

git add [file name]



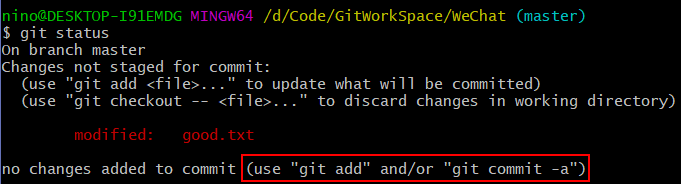
删除暂存区或分支上的文件，但本地又需要使用，只是不希望这个文件被版本控制，使用git rm --cached [filename]



### 1.5.3将暂存区的内容提交到本地库

git commit -m ”commit message” [file name]

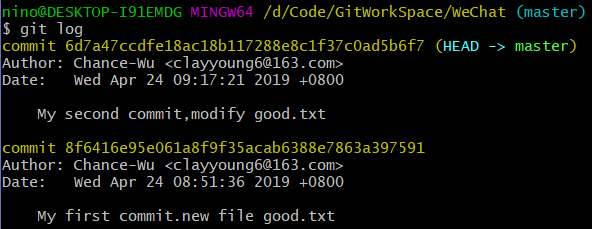
### 1.5.4修改文件后的操作



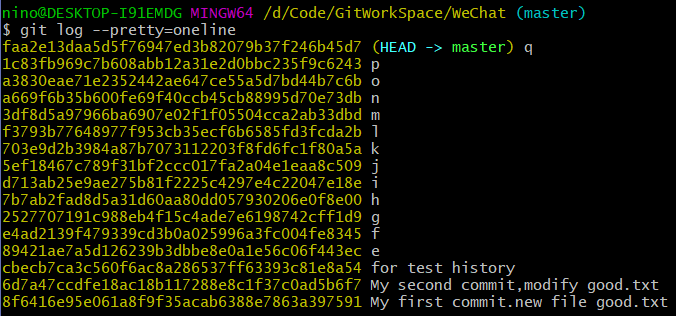
两种方法：添加、提交或者直接提交（git commit -a）——文件不能提交

### 1.5.5查看历史记录（版本穿梭测试的准备工作）

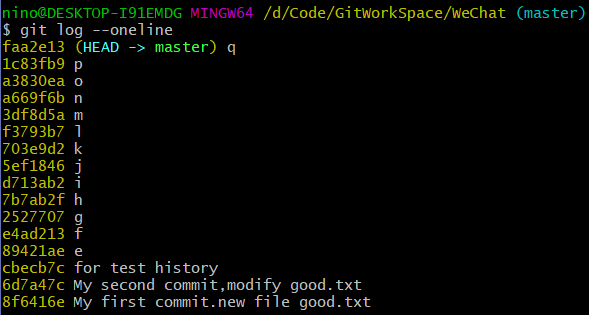
1. git log



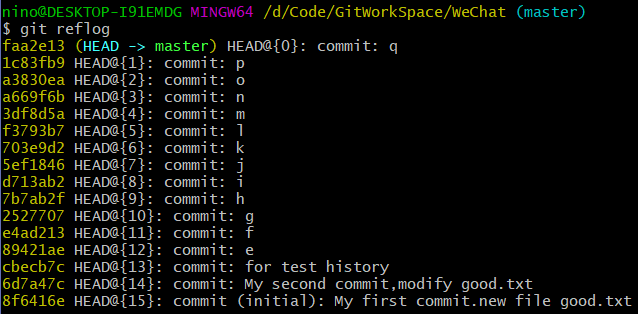
1. git log --pretty=oneline 每一条日志只显示一行



1. git log –oneline 简洁的形式显示



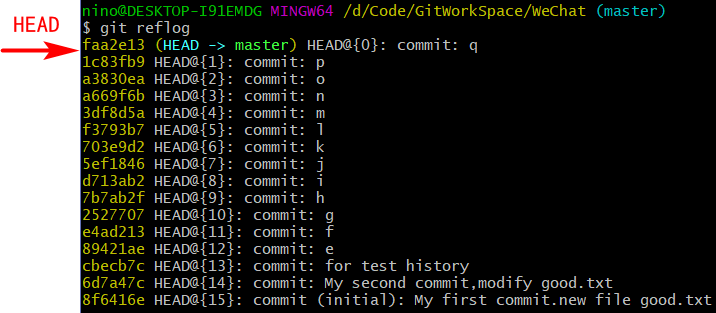
1. git reflog查看所有分支的所有操作记录（包括commit和reset的操作），包括已经被删除的commit记录



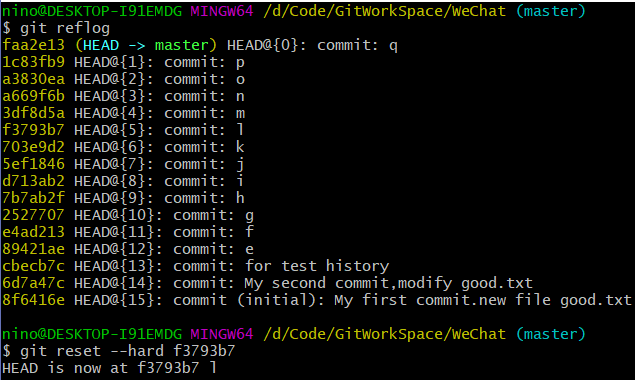
HEAD@{移动到当前版本需要多少步}

### 1.5.6前进后退版本

* 本质：



* 基于索引值操作[推荐]
  + git reset --hard [局部索引值]
  + git reset --hard f3793b7

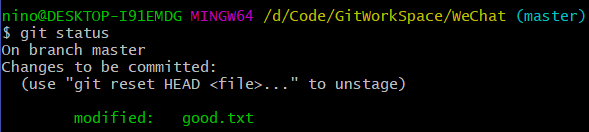


* 使用^符号：只能后退
  + git reset --hard HEAD^
  + 注：一个^表示后退一步，n个表示后退n步
* 使用~符号：只能后退
  + git reset --hard HEAD^n
  + 注：表示后退n步

### 1.5.7reset命令的三个参数对比

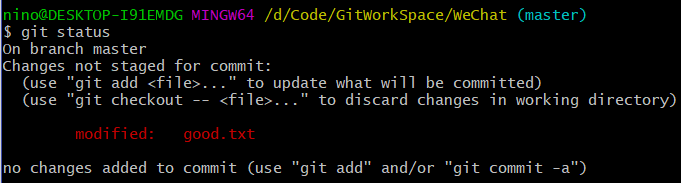
* --soft 参数
  + 仅在本地库移动HEAD指针





* --mixed 参数
  + 在本地库移动HEAD指针
  + 重置暂存区

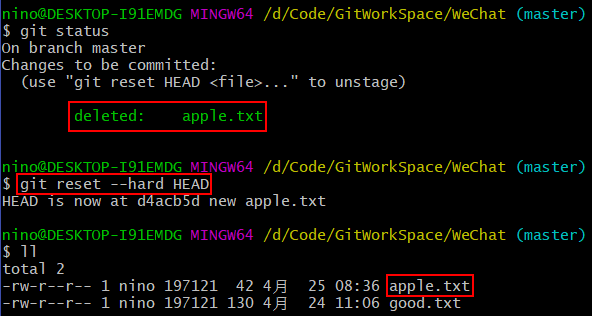




* --hard 参数
  + 在本地库移动HEAD指针
  + 重置暂存区
  + 重置工作区

### 1.5.8删除文件并找回

* 前提：删除前，文件存在时的状态提交到了本地库。
* 操作：git reset –hard [指针位置]
  + 删除操作已经提交到本地库：指针位置指向历史记录
  + 删除操作尚未提交到本地库：指针位置使用HEAD



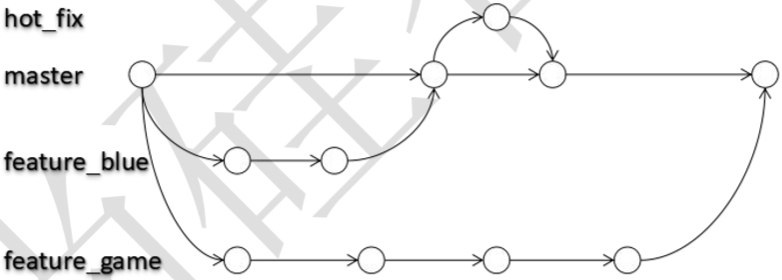
### 1.5.9比较文件差异

* git diff [文件名]
  + 将工作区中的文件和暂存区进行比较
* git diff [本地库中历史版本] [文件名]
  + 将工作区中的文件和本地库历史记录比较
* 不带文件名比较多个文件

## 1.6分支管理

### 1.6.1什么是分支？

在版本控制过程中，使用多条线同时推进多个任务。



### 1.6.2分支的好处

* 同时并行推进多个功能开发，提高开发效率
* 各个分支在开发过程中，如果某一个分支开发失败，不会对其他分支有任何影响。失败的分支删除重新开始即可。

### 1.6.3分支操作

* 创建分支

git branch [分支名]



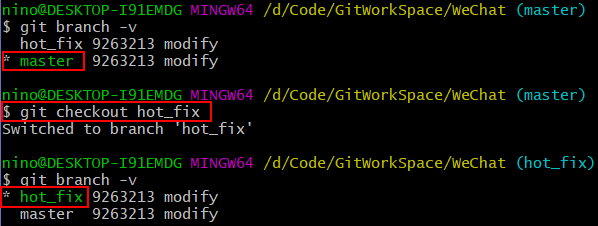
* 查看分支

git branch -v



* 切换分支

git checkout [分支名]

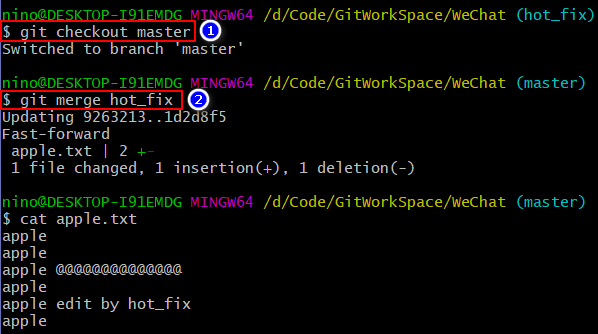


* 合并分支
  + 第一步：切换到接受修改的分支（被合并，增加新内容）上

git checkout [被合并分支名]

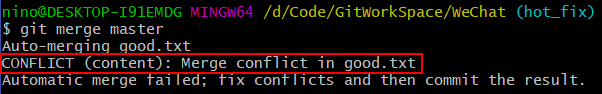
* + 第二步：执行merge命令

git merge [有新内容分支名]

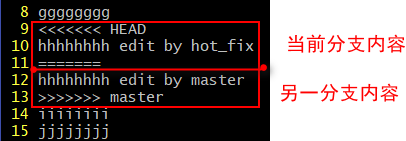


* 解决冲突
  + 产生冲突原因

多个开发者同时使用或者操作git中的同一个文件，最后在以此提交和push的时候，第一个操作的是可以正常提交的，而之后的开发者想要执行pull（拉）和pull（push）操作的时候，就会报冲突异常conflict。



* + 冲突的表现



* + 冲突的解决
    - 第一步：编辑文件，删除特殊符号
    - 第二步：把文件修改到奥满意的程度，保存退出
    - 第三步：git add [文件名]
    - 第四步：git commit -m “日志信息”
      * 注意：此时commit一定不能带具体文件名

## 1.7Git基本原理

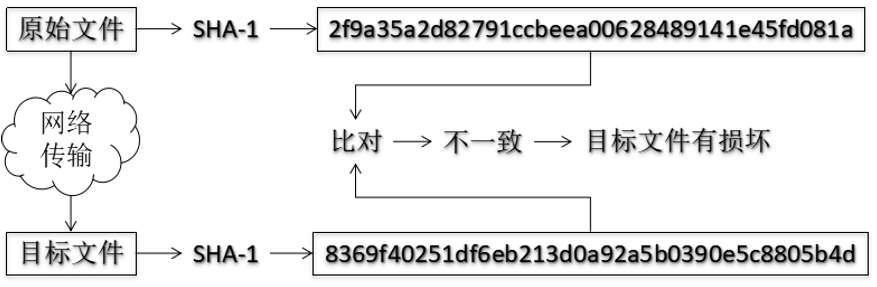
### 1.7.1哈希

哈希是一个系列的加密算法，各个不同的哈希算法虽然加密强度不同，但是有以下几个共同点：

* 不管输入数据的数据量有多大，输入同一个哈希算法，得到的加密结果长度固定。
* 哈希算法确定，输入数据确定，输出数据能够保证不变。
* 哈希算法确定，输入数据有变化，输出数据一定有变化，而且通常变化很大
* 哈希算法不可逆。

Git底层采用的是SHA-1算法。

哈希算法可以被用来验证文件。原理如下图所示：

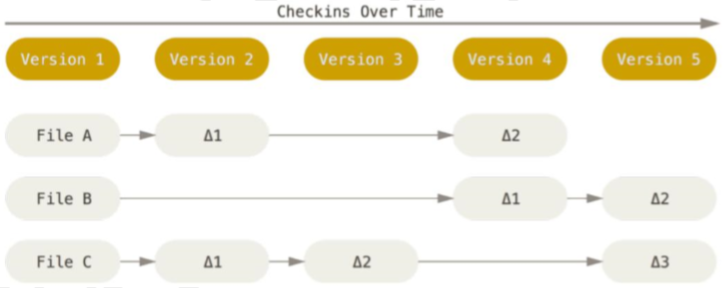


Git就是靠这种机制来从根本上保证数据完整性的。

### 1.7.2Git保存版本的机制

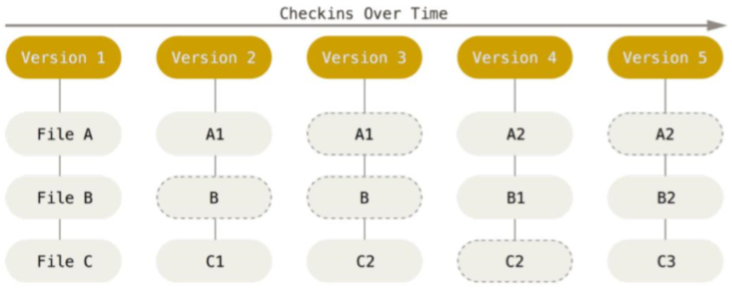
* 集中式版本控制工具的文件管理机制

以文件变更列表的方式存储信息。这类系统将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。



* Git的文件管理机制

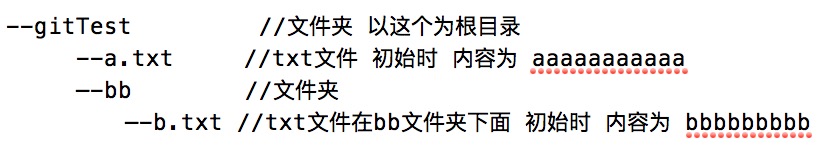
Git把数据看作是小型文件系统的一组快照。每次提交更新时Git都会对当前的全部文件制作一个快照并保存这个快照的索引。为了高效，如果文件没有修改，Git不再重新存储该文件，而是只保留一个链接指向之前存储的文件。所以Git的工作方式可以称之为快照流。



* Git文件管理机制细节

git会将所有改动的文件存到.git/objects/的文件夹下。

例如创建以下文件：



git的文件对象分为四类分别为blob、tree、commit、tags。

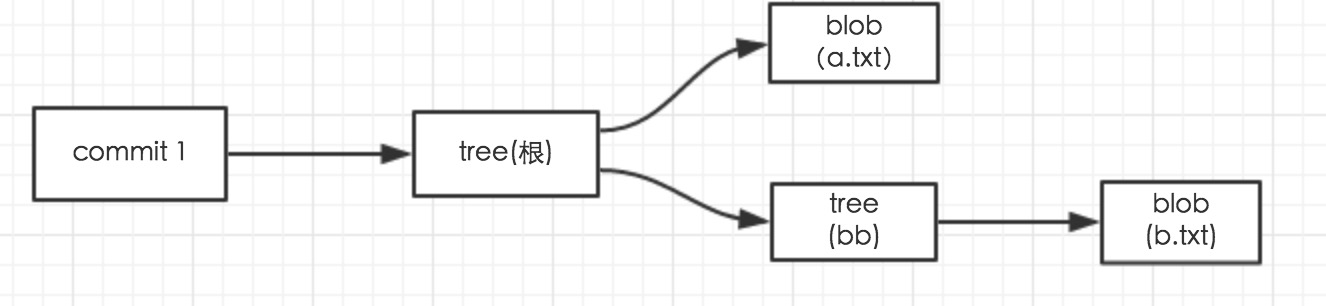
git管理的文件都以blob类型存在，而目录文件则以tree类型存在，tree类型内部可以引用若干个blob和tree文件。

当我们执行了commit 之后git 会依照我们工程的目录结构在objects里相应的建立一套blob和tree文件，可是我们的示例工程只有两个文件和一个文件夹，为什么objects里确有五个文件那，这里就要说说commit类型了，commit类型内部可以引用一个tree类型文件和一个commit类型文件。

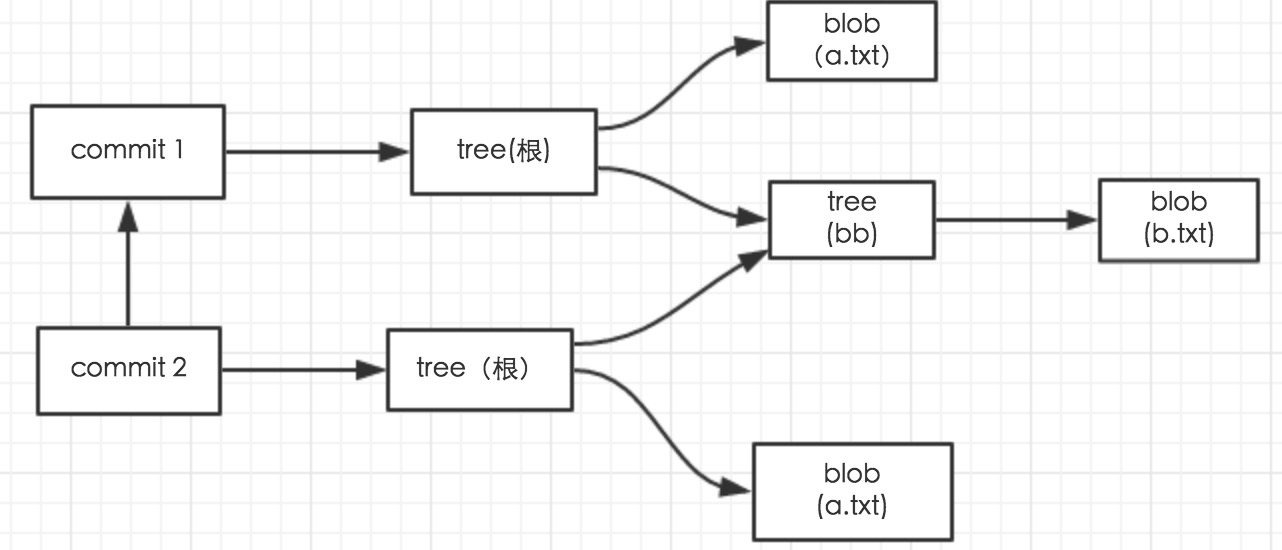
关于文件的名字，objects内部的文件全部是以sha-1命名，如果是blob类型则以文件内容做sha-1计算得出40字符的校验和，然后为了不让这个objects内部文件过多，所以使用40字符的前两位来建立一个文件夹，在以后38位为文件名，包括commit和tree全部使用sha-1校验和来命名。

接下来说一下这五个文件是以什么结构关联的：

每个commit引用一个tree，根据工程目录结构这个tree在引用一个blob(a.txt)和一个tree（bb），这个tree（bb）则引用一个blob（b.txt）如图：

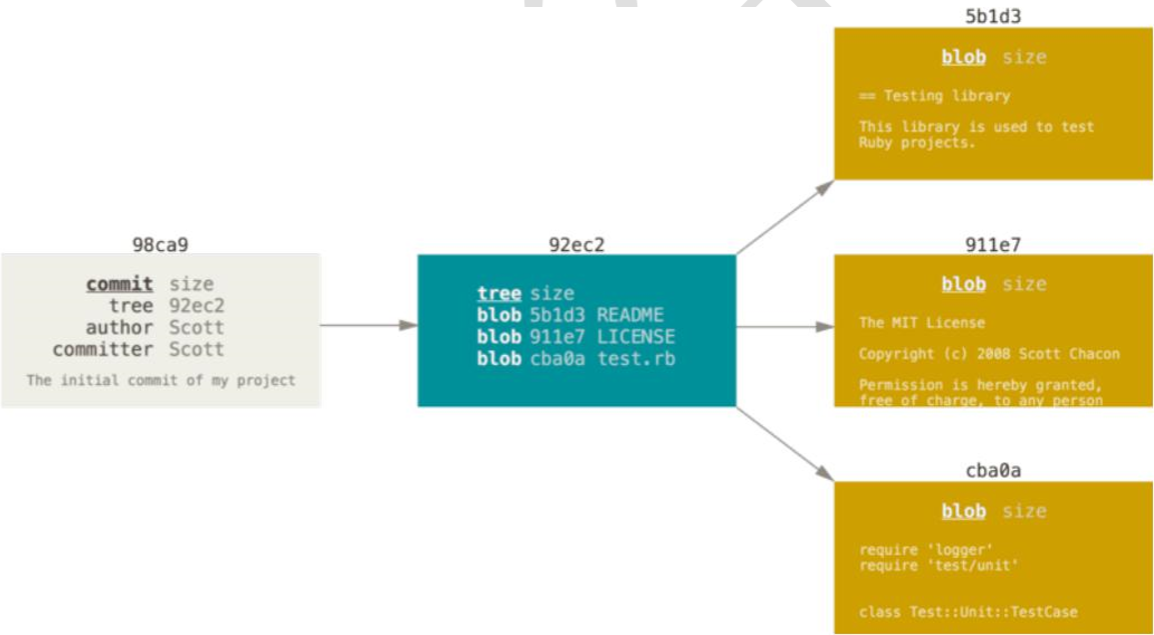


改动以下a.txt然后再提交一个版本，在a.txt添加22222222222内容执行add和commit，添加一个版本，比刚刚多了三个文件。因为我们改动了a.txt文件，所以这里在保留上一个版本的a.txt的基础上新增加一个a.txt文件，git只对有改动的文件进行备份保留。另外两个文件分别是commit2和一个根tree，现在这8个文件的关系如图：

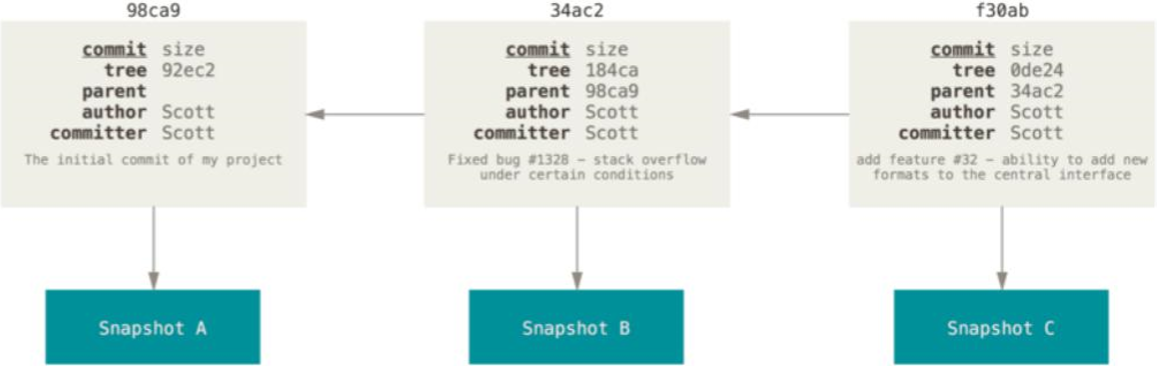


commit文件内部可以引用一个commit，这样commit之间就可以建立关系了，因为只有a.txt文件做了改动所以只有a.txt文件新建了一个，然后被commit2引用，bb和b.txt文件未做改动则commit2依然引用之前的文件。

* + Git的“提交对象”



* + 提交对象及其父对象形成的链条



### 1.7.3Git分支管理机制

git branch可以获取当前的分支列表，这个分支列表会保存在./git/refs/heads/这个路径下，这里包含master和其他一些分支文件，以master文件为例查看master文件内容如下：



其实这是一个commit类型的文件名，这样每个分支都可以拥有一个自己的commit引用，从上面的图可以看出只要拿到commit的文件名就可以找到所有跟他关联的文件，还有个问题是./git/refs/heads/下有所有分支信息，总要有一个当前分支，其实这个当前分支被记录在./git/HEAD文件内部：



git checkout某某分支来切换分支的时候就是在修改HEAD这个文件内容。基本可以理解为我们首先选择一个分支为当前分支，每个分支里记录着当前分支最顶端的commit对象，这个commit对象又可以找到所有跟它关联tree和blob，同时commit对象又和它的历史commit关联。我们可以任意切换当前分支，同时又可以修改当前分支指向的commit对象，比如我们执行reset可以选择回退到任意一个commit。这些就可以顺理成章切换任意分支并且找到任意版本的文件了。

* 分支的创建



* 分支的切换



