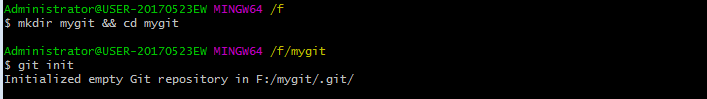
# 基础入门

创建版本库：mkdir mygit && cd mygit

git init



# 1.Git基本概念

## 1.1 基本概念

### 1.1.1 版本库

Git版本库是一个简单的数据库，其中包含所有用来维护与管理项目的修订版本和历史信息。一个版本库维护整个项目的生命周期的完全副本。

Git在每一个版本库中维护一组配置信息，如：版本库的用户名和email地址。

在版本库中，Git主要维护两个主要数据结构：对象库（object stroe）和索引（index），所有这些版本库数据放在工作目录根目录下一个名为.git的隐藏子目录中。

### 1.1.2 Git对象类型

对象库是Git版本库实现的心脏。它包括你的原始数据文件和所有日志消息，作者消息，日期，以及其他用来重建项目任意版本或分支信息。

Git方在对象库里的对象只有4中类型：块（Blob），目录树（tree），提交（commmit）和

标签（tag）。这四种原子对象构成了Git高层数据结构的基础。

**块（Blob）：**文件的每一个版本表示为一个块。Blob是“二进制大对象”缩写（binaray large object）。一个blob保存了一个文件的数据，但不包括关于这个文件的元数据，甚至连文件名都没有。

git hash-object file：计算和输出一个文件的sha-1值。

如： echo "test content" | git hash-object -w --stdin

输出值：d670460b4b4aece5915caf5c68d12f560a9fe3e4

-w：将一个对象写入到对象数据库中，如果不使用，该命令只返回键值。

--stdin：从标准输入中读取一个对象。

我们可以从.git/objects查看我们：find .git/objects –type f

**目录树（tree）：**一个目录树对象代表一层目录信息。它记录了blob标识符，路径名和在一个目录里的所有文件的一些元数据。它也可以递归引用其他目录树或子树对象，从而建立一个包含文件和子目录的完整层次结构。

git cat-file -p master^{tree}：指定了master分支中最后一次提交所指向的树对象。

**提交（commit）：**一个提交对象保存版本库中每一次变化的元数据，包括作者，提交者，提交日期和日志信息。每一个提交对象指向一个目录树对象，这个目录树对象在一张完整的快照中捕获提交时版本库的状态。最初的提交或根提交时没有提交父提交的。

创建提交对象：git commit-tree

标签（tag）：一个标签对象分配一个任意的且我们可读的名字个一个特定的对象，通常是一个提交对象。

### 1.1.3 索引

索引是一个临时的，动态的二进制文件，它描述了真个版本库的目录结构。更具体的来说，索引捕获了项目在某个时刻的整体结构的一个版本。项目的状态可以用一个提交和一棵目录树表示，它可以来自项目历史中的任意时刻，或者它可以是你正在开发的未来的状态。

### 1.1.4 标签

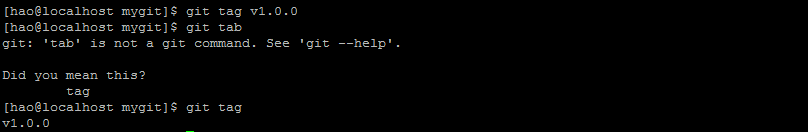
Git基本标签类型为：轻量级的（lightweight）和带附注的（annotated）

轻量级标签：只是一个对象的引用，通常被版本库视为私有的，并且不在版本库中创建永久的对象。

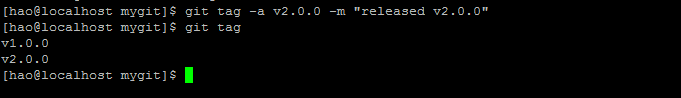
带附注标签：会在版本库中创建一个对象并包含你提供的一条信息，并根据REC 4880来使用GnuPG密钥进行数字签名。

Git在命名一个提交时，对轻量级标签和带附注标签同等对待。

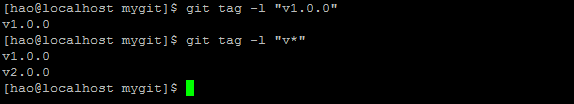
创建一个轻量级标签：git tag v1.0.0



创建一个带附注标签：git tag –a v2.0.0 –m “v2.0.0 released”



查找标签：git tag –l ‘v1.0.0’ 或者 git tag –l “v\*”

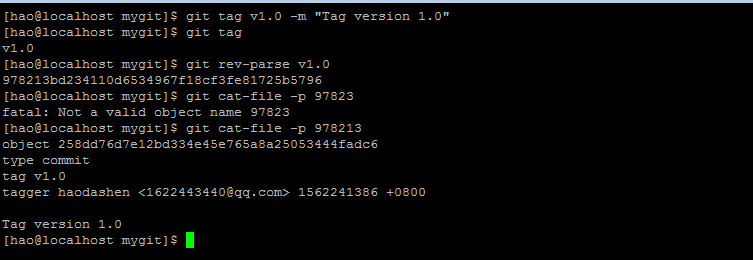


删除指定的标签：git tag –d v1.0.0



查看标签对应的SHA1值：git rev-parse v1.0

查看标签对象：git cat-file –p 978213



我们可以看出标签指定的提交对象为：258dd76d

共享标签：

默认情况下，git push不会把标签传输到远程仓库中，在创建了子标签后，你必须明确地将标签推送到远程仓库中

git push origin tag\_name：如git push origin v1.0.0

git push origin –tags：一次性将所有 的标签推送到远程仓库。

检出标签：

由于标签无法移动，如果需要将某个版本的仓库放入像是标签的工作目录中，使用命令：

git checkout –b [branch\_name] [tag\_name]在特定的标签上创建分支。

## 1.2 Git工作时的概念

# 2 文件管理

工作目录和版本库之间加设了一层索引，用来暂存，收集或者修改。

版本库

暂存区

工作区

## Git中文件分类

Git将文件分为三类：已追踪，被忽略和未追踪。

已追踪（Tracked）：是指已经在版本库中的文件，或者已经暂存到索引中的文件。

被忽略文件（Ignored）：被忽略的文件必须在版本库中明确声明为不可见或被忽略，即使它可以在你的版本库中可见。

未追踪的（Untracked）：是指不在前两类中的文件。

# 3 索引

# 4 .gitignore文件

被忽略的文件必须在版本库中明确声明为不可见或被忽略，即使它们可能在你的工作目录中出现。Git维护一个默认忽略文件列表，也可以配置版本库来识别其他文件。

特殊文件.gitignore，只要将想要忽略的文件名添加到同一目录下的.gitignore中即可，此外，可以通过将文件名添加到版本库顶层目录下的.gitignore文件中来忽略它。

一个.gitignore文件可以包含一个文件名模式列表，指定哪些文件要忽略。

**文件格式：**

* 空行要被忽略，而以#开头的行可以用于注释，如果#跟在其他文件后面，它就不表示注释。
* 一个简单的字面置文件名匹配任何目录中的同名文件。
* 目录名由末尾的反斜杠（/）标记。
* 以斜杠（/）开头的模式可用于禁止递归匹配
* 起始的感叹号（！）会对该行其余部分的模式进行取反。此外，被之前模式排除但被取反规则匹配的文件是要包含的。取反模式会覆盖低优先级的规则。

Git允许在版本库中任何目录下有.gitignore文件，每个文件都只影响该目录及其所有子目录。

.gitignore的规则也是级联的：可以覆盖高层目录中的规则，只要在其子目录包含一个取反模式（使用起始的！）

如：

\*.a #忽略.a类型文件

!li2.a #仍然追踪lib2.a文件，上一行命令要忽略.a类型文件

build/ #忽略build/目录下的所有文件

/hao.txt #只忽略当前目录的hao.txt，而不忽略目录下的hao.txt

doc/\*.txt #忽略doc目录下的所有.txt类型文件

doc/\*\*/\*.java #忽略doc目录下的所有.java文件

# 提交

Git中提交是用来记录版本库的变更的。

在提交时，Git’会记录索引的快照把快照放进版本库中，这个快照不包含该索引中任何文件或目录的副本，因为这样的策略会需要巨大的存储空间。Git会将当前索引的状态与之前的索引的状态进行比较，并派生出一个受影响的文件和目录列表。Git会为任何有变化的文件创建一个新的Blob对象，对有变化的目录创建新的树对象，对于没有变化的文件和目录会引用之前的Blob和树对象。

提交命令：git commit –a -m “”：对于已追踪的所有文件，直接跳过暂存区，直接提交。

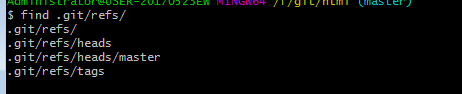
## 引用和符号引用

引用是一个SHA1散列值，指向Git对象库中的对象，虽然一个引用可以指向任何Git对象，但是它通常指向提交对象。

符号引用或称symref，简介指向Git对象，它仍然是一个引用。

本地特性分支名称，远程跟踪分支名称和标签名都是引用。

每一个符号引用都有一个以ref/开头的明确名称，并且都分层存储在版本库的.git/refs/目录中，该目录有三种不同命名空间代表不同的引用：refs/heads/ref代表本地分支，refs/remotes/ref代表远程分支，refs/tags/ref代表标签。



Git自动维护几个用于特定目的的特殊符号引用：

HEAD：始终指向当前分支的最近提交。当切换分支时，HEAD会更新为指向新分支的最近提交。

ORIG\_HEAD：

FETCH\_HEAD：当使用远程库时，git fetch命令将所有抓取分支的头记录到.git/FETCH\_HEAD中。FETCH\_HEAD是最近抓取的分支HEAD的简写，并且仅在刚刚抓取操作之后才有效。使用这个符号引用时，哪怕是一个对没有指定分支名的匿名抓取操作，都可以在

git fetch时找到提交的HEAD。

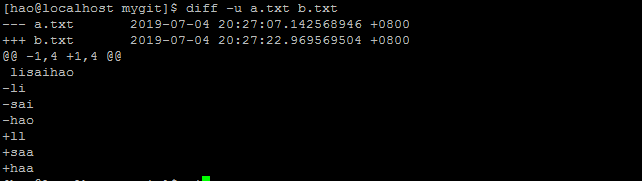
MERGE\_HEAD：当一个合并操作正在进行时，其他分支的头暂时记录在MERGE\_HEAD中，

MERGE\_HEAD是正在合并进HEAD的提交。

# diff使用

diff：用来比较两个事物之间的不同

在Unix系统上使用diff命令进行比较：diff –u source target

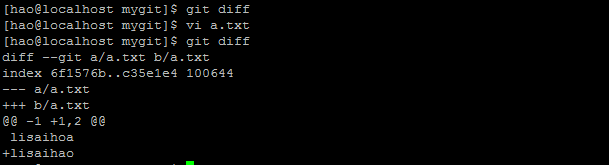


原始文件被“---”符号标记，目标文件被“+++”符号标记，@@之间表示两个不同文件版本的上下文行号。以“-”减号开头的行表示从原始文件删除该行以得到新文件，以“+”加号开始的行表示熊原始文件添加该行以产生新文件，而以空格开头的行表示两个文件中都有。

## git diff命令格式

git diff：显示工作空间与索引空间之间的差异。同时它会显示工作目录里什么是“脏的”，并把这个“脏”文件作为下个提交暂存的候选。

创建a.txt文件，并添加到暂存区内，之后修改a.txt文件，然手使用git diff进行比较：

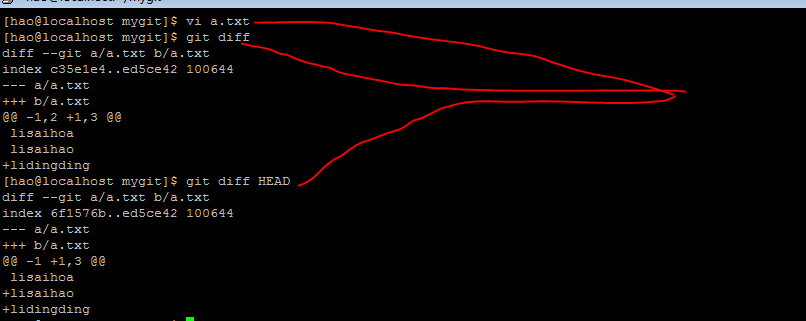


我们可以看出工作区域暂存区的差别，这里a/a.txt代表暂存区中的内容，b/b.txt代表工作

区内的内容。

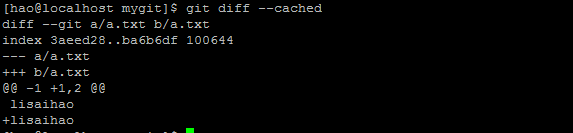
git diff commit\_id：显示工作目录与给定提交间的差别。常用的一种做法是使用HEAD或者一个特定的分支名作为commit\_id.

我们对a.txt进行修改并没有提交：



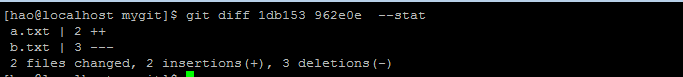
提交之后进行比较：

git diff –cached commit\_id：显示索引（暂存区）中变更与给定提交之间的差异。如果省略commit\_id，这默认为HEAD。

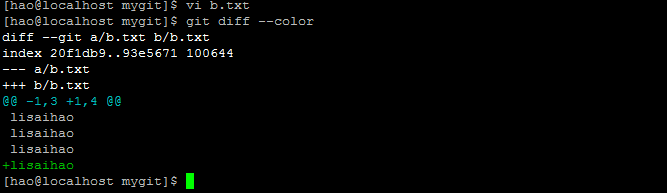


git diff commit1 commit2：比较指定两个提交之间的差异。这个命令会忽略索引和工作目录

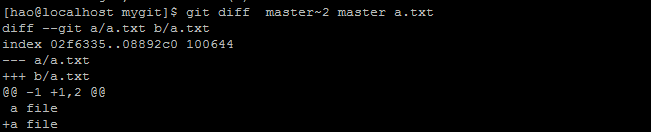
加上—stat：使用简介的方式显示有多少行发生了改变，多少行添加了，多少行删除了。



使用—color参数



我们也可以指定文件来比较文件发生的变化：git diff commit1 commit2 filename



# 6. 分支

分支时在软件项目中启动一条单独的开发线的基本方法，分支从一种统一的，原始的状态分离出来的，是开发能够在多个方向上同时进行，并可能产生项目的不同版本。通常情况下，分支会调节和其他分支合并。

Git允许很多分支，因此一个版本库中可以有许多不同的分支。Git的分支系统是轻量级的，简单的，因此，Git对合并的支持是一流的。

使用分支的原因：（1）一个分支通常代表一个单独的客户发布版

（2）一个分支可以封装一个开发阶段，比如原型，测试，稳定或临近发布。

（3）一个分支可以隔离一个特性的开发或者研究特别复杂的bug。

（4）每一个分支可以代表单个贡献者的工作。

版本库中默认分支命名为master，大多数开发人员在这个分支上保持版本库中最强大和最可靠的开发路线。

分支命名注意事项：（1）可以使用斜杠（/）创建一个分层的命名方案。但是，该分支名不能以斜线结尾。

（2）分支名不能以减号（-）开头。

（3）以斜杠分割的组件不能以点（.）开头。如hao/.sai这样的分支无效

（4）分支名的任何地方不能包含两个连续的点（..）

（5）分支名中不能包含空格或其他空白字符，Git的特殊含义的字符。和ASCLL码控制字符。

在版本库中可能有许多不同的分支，但最多只有一个当前的或活动的分支。活动分支决定工作目录中检出哪些文件。当前分支往往是Git命令中的隐含操作数，默认情况下，master分支作为活动分支，但可以把任何分支设为当前分支。

分支允许版本库中每一个分支的内容向许多不同的方向发散。当一个版本库分出至少一个分支时，把每次提交应用到某个分支，取决于哪个分支时活动的。

每个分支在一个特定的版本库中必须有唯一的名字，这个名字始终指向该分支上最近提交的版本。一个分支的最近提交称为该分支的头部。

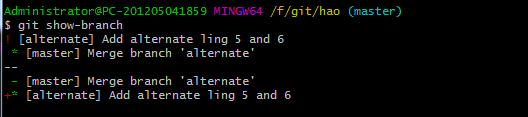
## 6.1创建分支

命令：git branch branch [starting-commit]

如果在创建分支时没有指定starting-commit，则默认认为当前分支的最近提交作为新分支的开始。注意：git branch命令只是把分支名引入版本库中，并没有改变工作目录去使用新的分支，没有工作目录文件发生变化，没有隐式分支环境发生变化，也没有做出新的提交。

命令：git branch列出版本库中的分支名。

git show-branch按时间以递序的形式列出一个或多个分支有贡献的提交，-r显示远程追踪分支，-a显示所有分支。



git show-branch输出被一排破折号分为两部分，**上部分**列出分支名，并用方括号括起来，每行一个，每个分支名跟着一行输出，前面用感叹号或星号标记。上部分的每一个分支都列出该分支最近提交日志消息的一列。**下部分**是一个表示每个分支中提交矩阵。同样，每个提交后面跟着该提交中日志消息的第一行。如果有一个加号（+），星号（\*），或者减号（-）在分支的列中，对应的提交就会在该分支中显示。加号表示提交在一个分支，星号突出显示存在于活动分支的提交，减号表示一个合并分支。如果同一个提交存在于多个分支中，那么每个分支将有一个加号或星号作为标识。c

查看分支：git branch

git show-branch

git show-branch branch\_name1…

## 6.2 检出分支

工作目录一次只能反映一个分支，要在不同的分支上开始工作，要发出git checkout命令，给定一个分支名，git checkout会使该分支变成新的当前分支，它改变了工作树文件和目录结构来匹配给定分支的状态。

选择一个新的当前分支可能会对工作树文件和目录结构产生巨大影响，影响程度的大小取决当前分支和你想检出的目标分支之间的差异。

改变分支的影响有：（1）在要被检出的分支中单不在当前分支中的文件和目录，会从对象库中检出并放置到工作树中。

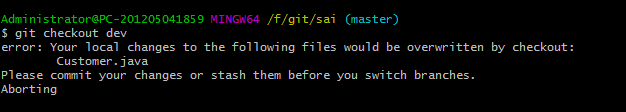
（2）在当前分支中但不在要被检出的分支中的文件和目录，会从工作树中删除。

（3）这两个分支都有的文件会被修改为要被检出的分支的内容。

### 6.2.1 在未提交的更改时进行检出

如果不明确请求，Git会排除本地工作树中数据的删除和修改。工作目录中的未被追踪的文件和目录始终会置之不管，Git不会删除或修改他们，但是，如果一个分支的本地修改不同于新分支上的变更时，Git会报错误信息。

在master分支和dev分支存在相同文件Customer.java，在master分支修改Customer.java后检出dev分支，Git报错：



如果你在dev分支上进行开发时，可能你接到一个紧急任务，对系统出现的bug及时的修复，我们可能会创建一个新的分支，对bug进行修复，但是在dev分支上我们的任务还没有提交，并不是我们不想提交，而是工作才进行一半，还没发进行提交，如果我们检出，则Git会报错，不让我们检出，解决这种问题了有两种方法：第一钟：对当前分支“储藏”。第二种：合并变更到不同的分支。第一种方法稍后详解，第二种方法，我们可以使用 git checkout –m 分支名 命令，Git通过在你的本地修改和对目标分支之间进行一次合并操作，尝试将你的本地修改加入加入到新工作目录中。

## 分离HEAD分支

我们可以检出任何提交，Git会自动创建一个匿名分支，称为一个分离的HEAD（detached HEAD）

在如何下的情况下，Git会自动创建一个HEAD：

1. 检出的提交不是分支的头部。
2. 检出一个追踪分支。
3. 检出标签引用的提交。
4. 启动一个git bisect操作
5. 使用git submodule update命令。

# 合并

一次合并会结合两个或多个历史提交，尽管Git还支持同时合并三个，四个或多个分支，但大多数情况下，一次合并只结合两两个分支。在Git中，合并必须在同一个版本库中，当一个分支中的修改与另一个分支中的修改不发生冲突时，Git会计算合并结果，并会创建一个新提交来代表新的统一状态。但是当分支冲突时，Git不解决冲突，这通常出现在对同一个文件的同一行进行修改的时候。相反，git把这种争议性的修改在索引中标记为“未合并”，留给开发人员来处理。

在开始合并之前，最好整理一下工作目录，在正常合并结束的时候，Git会创建新版本的文件并把它们放在工作目录中，并且Git在操作的时候还用索引来存储文件的中间版本。

## 无冲突合并

### 案例一：

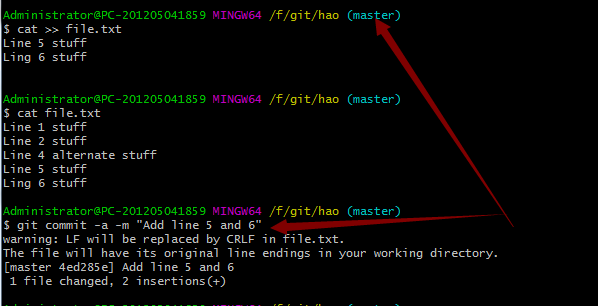
|  |
| --- |
| 第一步：创建版本库    第二步：在master分支上创建文件。        第三步：创建分支    alternate分支是从master^提交初始派生来的，也就是从当前头指针后面的一个提交。  第四步：在alternate分支下修改file.txt    第五步：合并    现在有了两个分支，每个分支都有不同的开发工作，在alternate分支上修改文件，并不影响相同文件的相同部分，所以合并可以顺利进行，不会发生冲突。  使用命令：git log –graph –pretty=oneline –abbrev-commit    两个分支在46417e9处分开，每一个分支显示一个提交（1c80172和a30d130），在849df69处合并。 |

## 有冲突合并

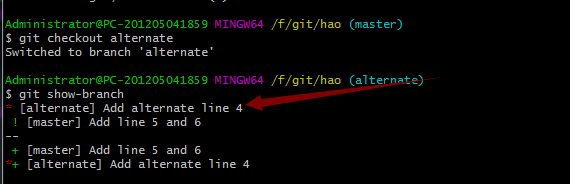
一个分支的修改可能与一个不同分支上的相似或完全不同，修改可能会改变相同文件或无关文件。

继续上面的例子：

在master分支上修改文件file.txt.

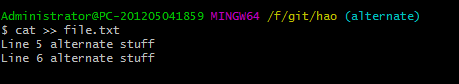


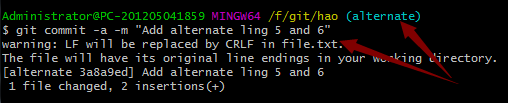
转换到alternate分支上，



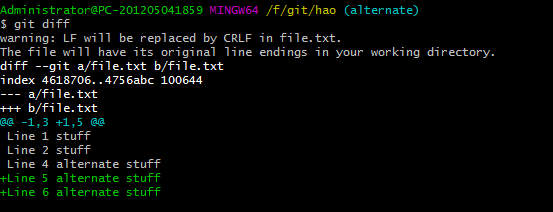
在这个分支上，我们可以看出master分支的提交并没有影响alternate分子，该分支还停在上一次提交时刻。

在alternate分支修改file.txt文件并提交。

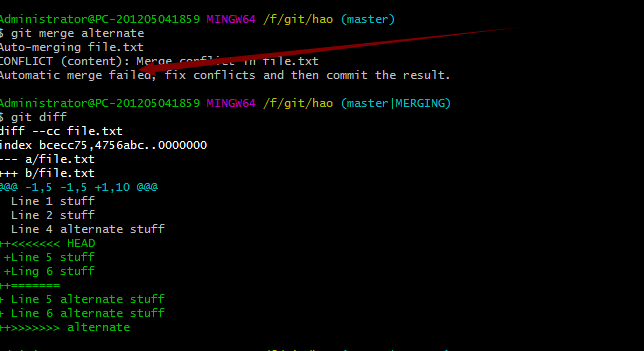




同过命令git diff查看修改前后的差异。



回到master分支，进行合并。

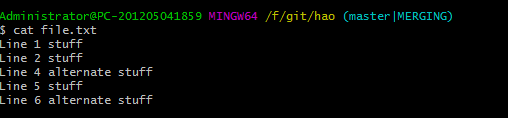


合并失败，发生冲突，使用命令git diff来调查冲突的程度。

git diff命令显示文件在工作目录和索引间的差异，<<<<<<<和===========之间，显示改变内容，==============和>>>>>>>>之间，显示替代的内容。使用额外的加号和减号来表示相对最终版本的来自多个源的变化。

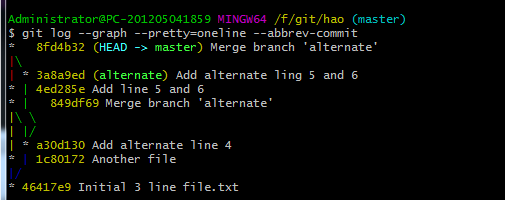
当解决合并冲突时，我们可以自由选择自己喜欢的任何方案，这包括只取一边或另一边的版本，或者两边的混合，甚至全新的内容。

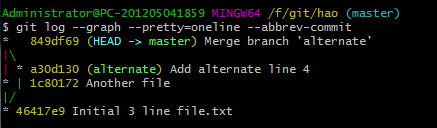
在这里我选择个分支选取一行作为我的解决版本，编译过的文件现如下截图：



如果对自己解决冲突很满意，就可以使用命令git add将文件添加到索引中，并为合并提交而暂存它。

在提交后，使用命令git log –graph –pretty=oneline –abbrev-commit查看提交结果。





# 更改提交

有很多的正当理由让我们去修改某个提交或整个提交序列：

## 使用git reset

git reset命令可以吧版本库和工作目录改变为已知的状态。git reset调整HEAD引用指向给定的提交，默认的情况下还会更新索引已匹配该提交。

Git reset有三个选项：--soft --mixed --hard

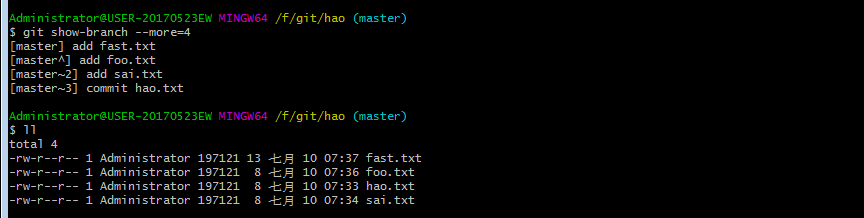
--soft：会将HEAD引用指向给定的提交。索引和工作目录的内容保持不变。这个命令有“最小”影响，只改变一个符合引用的状态使其指向一个新提交。

--mixed：会将HEAD指向给定的提交。索引内容也跟着改变以符合给定提交的树结构，但是工作目录中的内容保持不变。Git默认模式

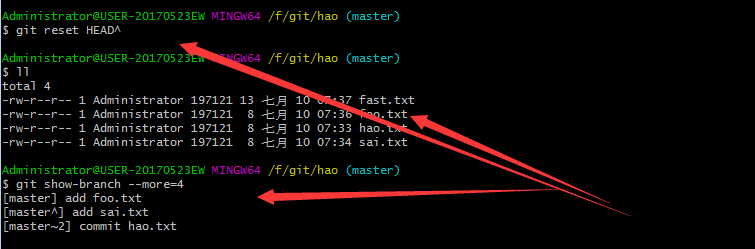
--hard：将HEAD引用指向给定提交。索引的内容也跟着改变以符合给定提交的树结构。此外，工作目录的内容也随之改变以反映给定提交表示的树的状态。

|  |  |  |  |
| --- | --- | --- | --- |
| 选 项 | HEAD | 索 引 | 工 作 目 录 |
| --soft | 是 | 否 | 否 |
| --mixed | 是 | 是 | 否 |
| --hard | 是 | 是 | 是 |

在工作目录中四个文件，共四次提交:

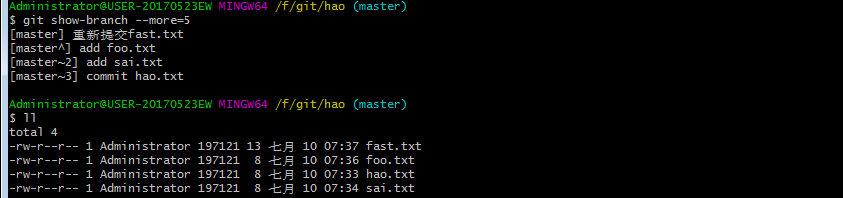


使用git reset命令回退版本库：git reset HEAD^



工作目录没发生变化，然而提交历史记录已经发生回退。

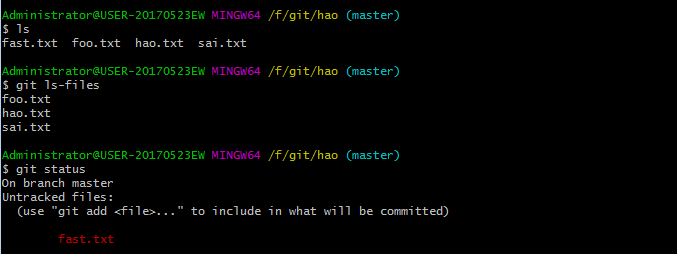
重新添加fast.txt并进行提交：



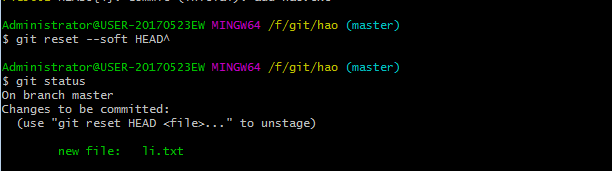
使用—mixed来重置索引：



索引发生改变。



如果你不需要改变索引，但是要调整提交信息，我们可以使用—soft选项。



Git reset –soft HEAD^命令把你挪回之前的提交图中的位置，但是保持索引不变。一切都像咩

有执行该命令一样那样暂存。

# 远程版本库

一个克隆时版本库的副本，一个克隆包含所有原始对象，所以每个克隆都是独立，自治的版本库，与原始版本库时真正对称，地位相同的。

远程版本库是一个引用或句柄，通过文件系统或者网路指向另一个版本库。

Git使用远程追踪分支跟踪其他版本库中的数据。

## 版本库概念

开发版本库：用于常规日常开发。

裸版本库：没有工作目录，并且不应该用于正常开发，没有检出分支的概念。

git clone –bare URL：创建裸版本库。

# 传输协议

Git支持四种不同的传输协议：本地协议，HTTP（S）协议，SSH（Secure Shell）协议和

Git协议。

## 本地协议

本地协议是Git最基本的协议，主要用于本地做一些Git实验。

git clone /opt/soft/public\_html

优点：基于文件仓库的优点在于它的简单，同时保留了现存文件的权限和网络访问权限。

缺点：与基本的网络连接访问相比，难以控制从不同位置来的访问权限。

## SSH协议

安装openssh-server。

SSH协议是一个授权验证的协议

SSH协议是唯一个既支持写操作，有支持读操作的协议。

git clone ssh://GitResosity@qq.com

优点：通过 SSH 进行访问是安全的 — 所有数据传输都是加密和授权的。

缺点：SSH 的限制在于不能通过它实现仓库的匿名访问。

## Git协议

Git协议是一个包含在 Git 软件包中的特殊守护进程； 它会监听一个提供类似于 SSH 服务的特定端口（9418），而无需任何授权。

优点：Git 协议是现存最快的传输协议。

缺点：缺少授权机制。

## HTTP（S）协议

# 储存

储存可以捕获你的工作进度，允许你保存工作进度并且当你方便时再回到该进度。

储存是一个快捷的方式，它让你仅通过一条简单的命令就全面彻底的捕获工作目录和索引，它使你的版本库保持干净。

## 基本命令

保存储存：git stash

查看储存列表：git stash list

保存储存：git stash save "save\_log"

恢复储存：git stash pop。注意：在pop操作成功后，Git会自动将储存状态栈中保存的状态删除，然而，当需要解决冲突时，Git将不会自动丢弃状态，以防你想要尝试不同的方法或还原到不同的提交。在我们解决冲突后，可以使用git stash drop将状态从储存状态栈中删除。

git stash apply

git stash apply stash@{n}：恢复指定的储存

git stash drop stach@{n}：删除指定的储存

git stash show：显示给定的储存条目相对于它的父提交的索引和文件变更记录。

git stash show –p stash@{n}：详细查看某n储存

形成储存状态的变更是相对于某个特定的提交的，所以显示的状态适合于git diff的转态比较的，而不是适合于git log的提交状态序列。

# 引用日志

引用日志记录非裸版本库中分支头的改变。每次对引用的更新，包括对HEAD的，引用日志都会记录这些引用。

默认的 情况下，引用日志在非裸版本库中是启用的，在裸版本库中是禁用的，我们可以使用命令：git config core.logAllRefUpdates ture|false启动或禁用引用日志。

git reflog show

git reflog branch\_name：查看指定分支的引用日志。

# 合并项目

## 子模块（git submodule）

git submodule add sub\_module\_addr sub\_module\_category

切记sub\_module\_category目录在工作目录中不存在，应交给git创建

如： git submodule add https://github.com/haodashen/git\_parent.git parent

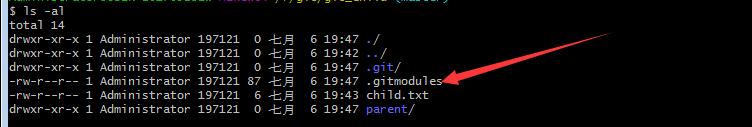
git submodule status：总结这个项目的所有子模块的提交引用和脏状态。

git submodule init：使用子模块信息长期储存的.gitmodules来更新开发人员版本库的.git/config文件。

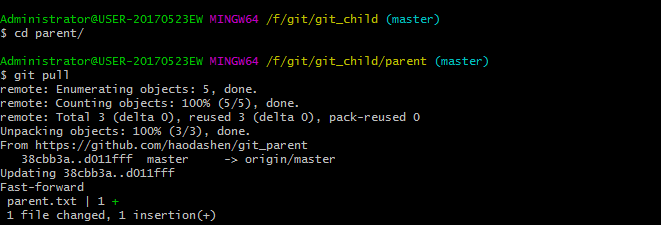
git submodule update：使用.git/config中的地址抓取子模块的内容，并在分离的HEAD指针状态下检出上层项目的子模块记录引用。

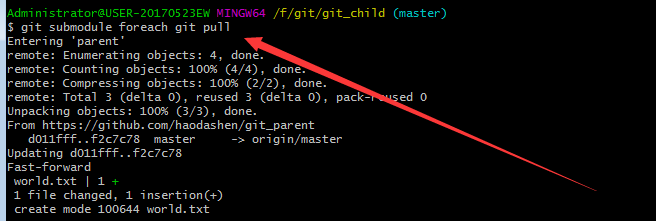
git submodule foreach command：对每一个子模块执行一条shell命令并提供$path，$sha1和其他有用的标识符。

git submodule summary：展示每一个



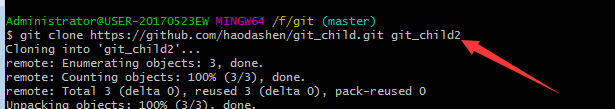


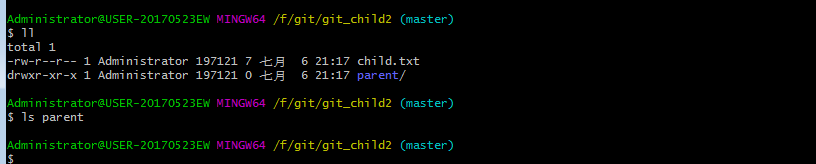




克隆子模块

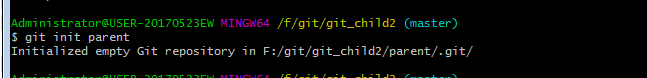
方法一：git clone remote\_repository\_url local\_repository\_name





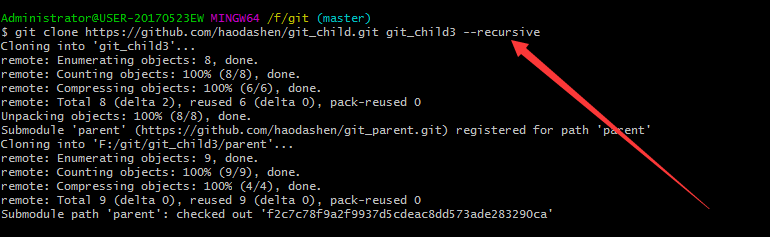
该方法并不能将子模块的代码克隆下来，解决方法：

初始化子模块：git init parent



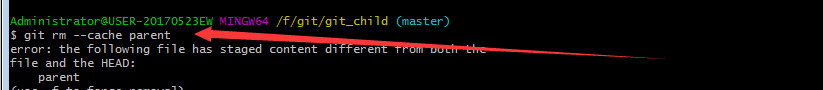
拉取子模块：git submodule update –recursive

方法二：克隆时添加—recursive参数

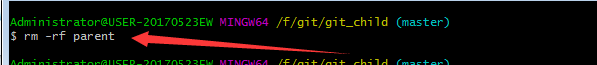


删除子模块：

第一步：从储存区删除：git rm –cached parent



第二步：在工作目录删除：rm –rf parent



第三步：删除.gitmodules文件

第四步：保存变更git add .

第五步：推送远程git push origin

# GitLab安装

