

关于版本控制：

记录一个或若干文件内容变化，以便将来查阅特定版本修订情况

发展：

1. 本地版本控制系统：复制整个项目文件夹从而形成新版本。

优点：简单

缺点：容易搞错

2. 集中化的版本控制系统：通过多台客户端连接服务器。

优点：每个人可以看到项目上其他人做了什么，管理员容易管理权限

缺点：一旦服务器挂了，谁都无法协同工作

3. 分布式版本控制系统：每次将代码仓库完整镜像下来，每个协作者都可在本地拥有一份完整的代码，并可在本地进行修改然后提交到远程仓库

Git和其他版本控制系统的差异

- 差异一：

Git：记录的是上一次每个变动文件的快照，直到文件产生新的变化git再进行新的快照保存，否则就建立一个指向上一文件的指针。

其他：记录变化文件与上一次对比产生了哪些差量，是增量式

- 差异二：

Git：本地数据资源完备，不需要网络便可频繁更新；等到有网络再提交远程仓库，本地资源完备体现在：譬如查看提交历史，版本回退等。

其他：有些需频繁请求网络以期得到数据资源进行数据更新

- 差异三：

Git：通过SHA1算法计算出指纹字符串记录文件内容或目录结构，可由远程代码仓库和本地版本文件进行比较，远程和本地都无更新状态下能验证文件的完整性。

关于Git认知

一、工作区、版本库、暂存区的区分

前提：初始化一个git仓库

- 工作区：本地可以看得见的目录
- 版本库：在隐藏目录.git下就是版本库

- 暂存区：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。

二、判断文件的三种状态

前提：想要判断文件状态首先要将文件加入追踪

- 已提交：.git目录保存特定版本文件就属于已提交状态。
- 已暂存：如果对文件做了修改并放入暂存区属于已暂存状态。
- 已修改：对文件做了修改但未放入暂存区就属于已修改状态

Git对象模型

一、对象名：每个对象名都是对对象内容做SHA1哈希计算得到而来

- 优点1：git只要比较对象名就可以很快知道两个对象是否相同
- 优点2：同样的内容在不同仓库拥有相同对象名
- 优点3：通过对象内容得到SHA1哈希值与对象名比较判断对象内容是否正确

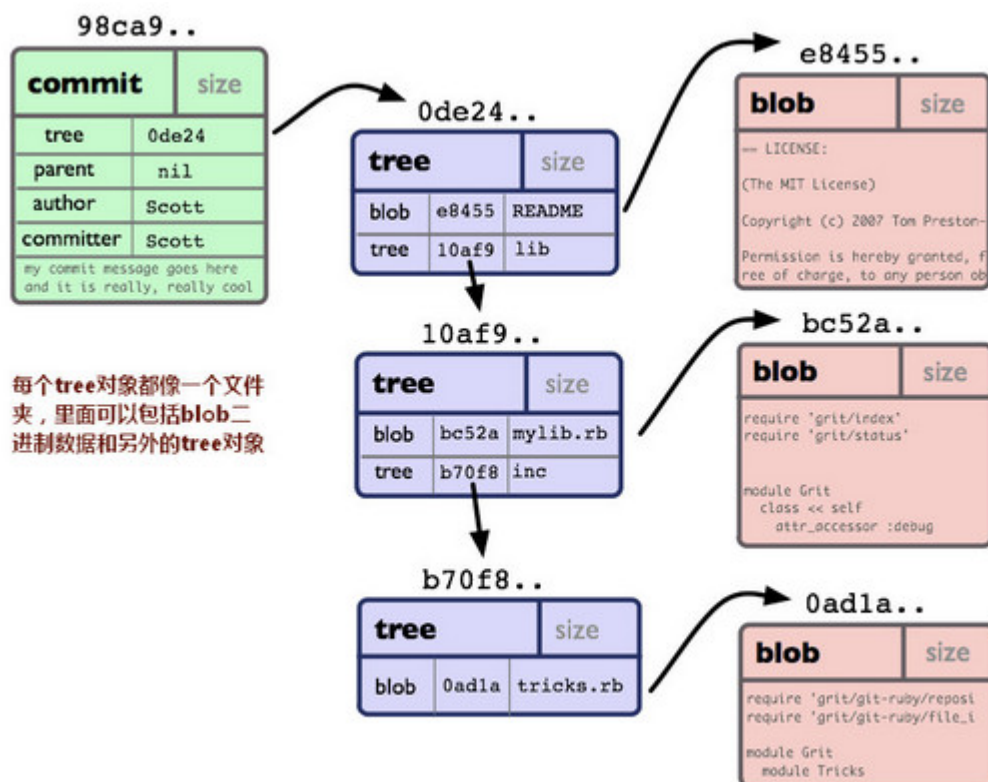
二、对象组成

- 对象的类型包括：

blob：用于存储文件数据，通常是一个文件（通常是二进制文件，没有其他任何属性，文件名修改对其无影响）tree：目录，管理tree和blob(子目录和文件) commit：一个commit指向一个tree，标记项目时间节点，（通常包含一个tree对象，父对象，作者，提交者，注释）tag：标记某一个提交方法

- 大小：内容大小
- 内容：取决于对象的类型

示意图如下：



Git工作流程

- 建立版本库

1. `git init`
2. `git clone + 远程仓库地址`

- 配置 Git

查看配置信息：`git config --list`

配置用户

1. 单个项目配置用户名：`git config user.name + <your_setting_username>`
2. 全局配置用户名：`git config --global user.name + <your_setting_username>`
3. 系统配置用户名：`git config --system user.name + <your_setting_username>`

配置邮箱

1. 单个项目配置邮箱：`git config user.email + <your_email_address>`
2. 全局配置邮箱：`git config --global user.email + <your_email_address>`
3. 系统配置邮箱：`git config --system user.email + <your_email_address>`

ps：配置的用户名有就近原则，例如：在一个项目中，配置的用户名将覆盖全局配置的用户名，优先级：1>2>3；以上配置的作用，Git 用此区分不同的开发人员的身份。

配置文本编辑器：`git config --global core.editor emacs`

配置差异分析工具：`git config --global merge.tool vimdiff`

配置别名：`git config --global alias.st status`

配置 `.gitignore` 文件（作用：列出要忽略的文件列表，不被 Git 纳入版本管理）

`.gitignore` 格式规范：

1. 空行和#都会被 git 忽略
2. 可使用正则模式
3. ' / ' 后说明要忽略的目录
4. ' ! ' 在忽略文件中除了此文件

例如：

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory
doc/**/*.pdf
```

命令详解

添加文件

`git add <file>` 将文件添加至暂存区，即将文件变为追踪状态

`git add .` 将所有文件添加至暂存区

撤销修改

`git checkout <file>` 将已追踪文件从 `modify` 状态变为未修改状态

`git reset HEAD <file>` 添加到了暂存区时，但想丢弃修改

`git reset HEAD~3` 将HEAD指针拨到指定commit_id处

`git reset --hard HEAD` 将文件恢复至之前未修改的工作区状态（`--hard` 参数是将工作区和暂存区强制一致到 commit_id处）

提交到版本库

`git commit -m 'comment'` 提交到版本库并添加注释

`git commit -am 'comment'` 相当于 `git add .` 和 `git commit` 的结合，此时你就省去一条命令，直接将文件添加至暂存区

提交修改

`git commit --amend` 如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息。

如果执行顺序如下：

```
$ git commit -m 'initial commit'
```

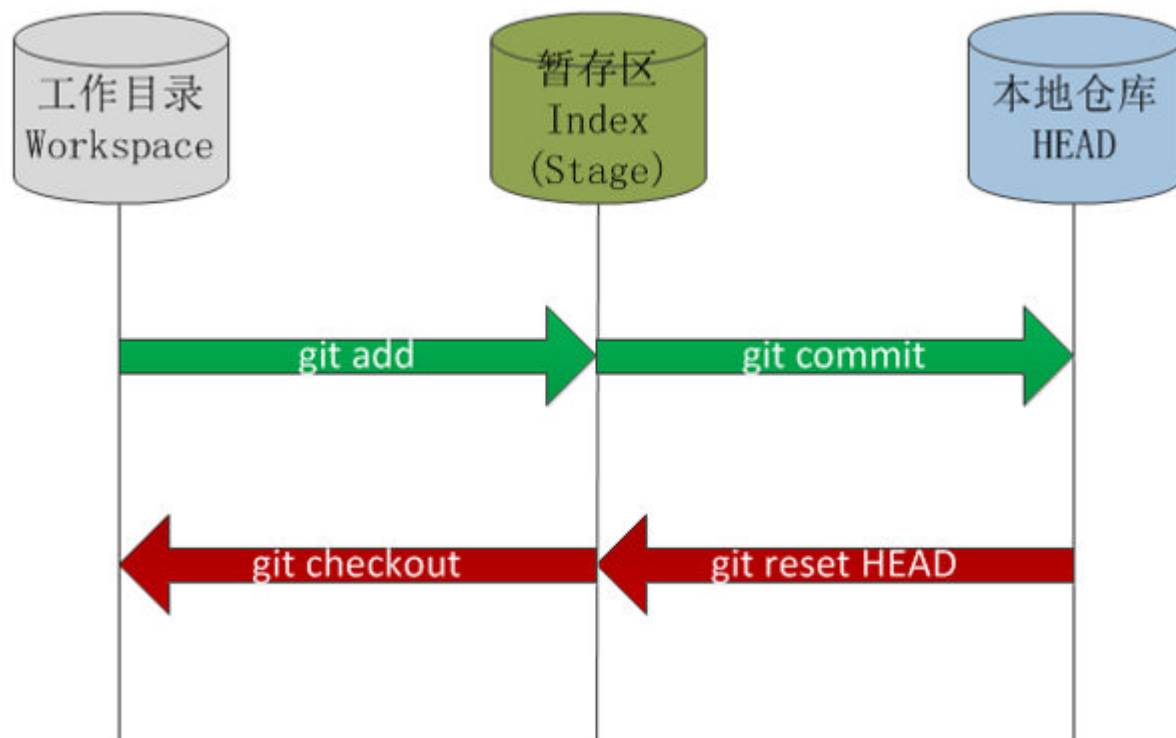
```
$ git add forgotten_file
```

```
$ git commit --amend
```

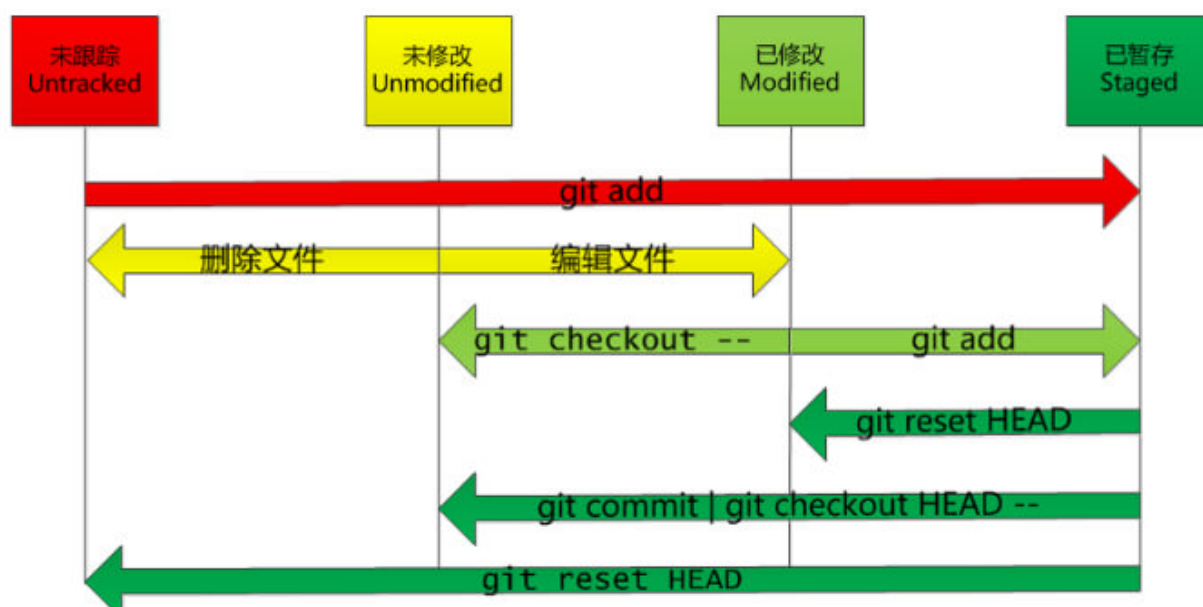
那么将会将后来添加的文件一起提交。

小结

- 简单 Git 工作流示意图如下所示：



- Git 文件状态



查看差异

`git diff file` 查看尚未暂存的文件更新了哪些部分

`git diff --cached||statged file` 查看已添加至暂存区的文件与版本库的区别

移除文件

情况一：未添加至暂存区（可以用 `git rm` 命令完成此项工作，连带从工作目录中删除指定的文件）

```
步骤1: rm filename (手工删除本地文件)
步骤2: git rm filename (记录移除操作)
```

情况二：已添加至暂存区（可以用 `git rm` 命令完成此项工作，连带从工作目录中删除指定的文件）

```
步骤1: git rm -f filename
或
步骤1: rm filename
步骤2: git rm filename
```

情况三：把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪

```
git rm --cached filename
```

文件改名

`git mv old_name new_name`

Git 非常聪明能够意识到这是一次改名，实际上这条命令相当于进行了三条命令的操作，即

```
$ mv old_name new_name
$ git rm old_name
$ git add new_name
```

查看 log

`git log -number #` 显示最近几条

`git log --oneline (-abbrev-commit --pretty=oneline) #` 单行显示，显示简短commit id

`git log --graph #` 以树形展示

`git log --decorate #` 显示分支名等

`git log --first-parent #` 显示第一父元素（不显示merge进来的树形结构）

`git log --all #` 显示全部分支

标签

- 列出标签列表：`git tag`

- 搜索相关标签：`git tag -l 'v1.8.5*'`

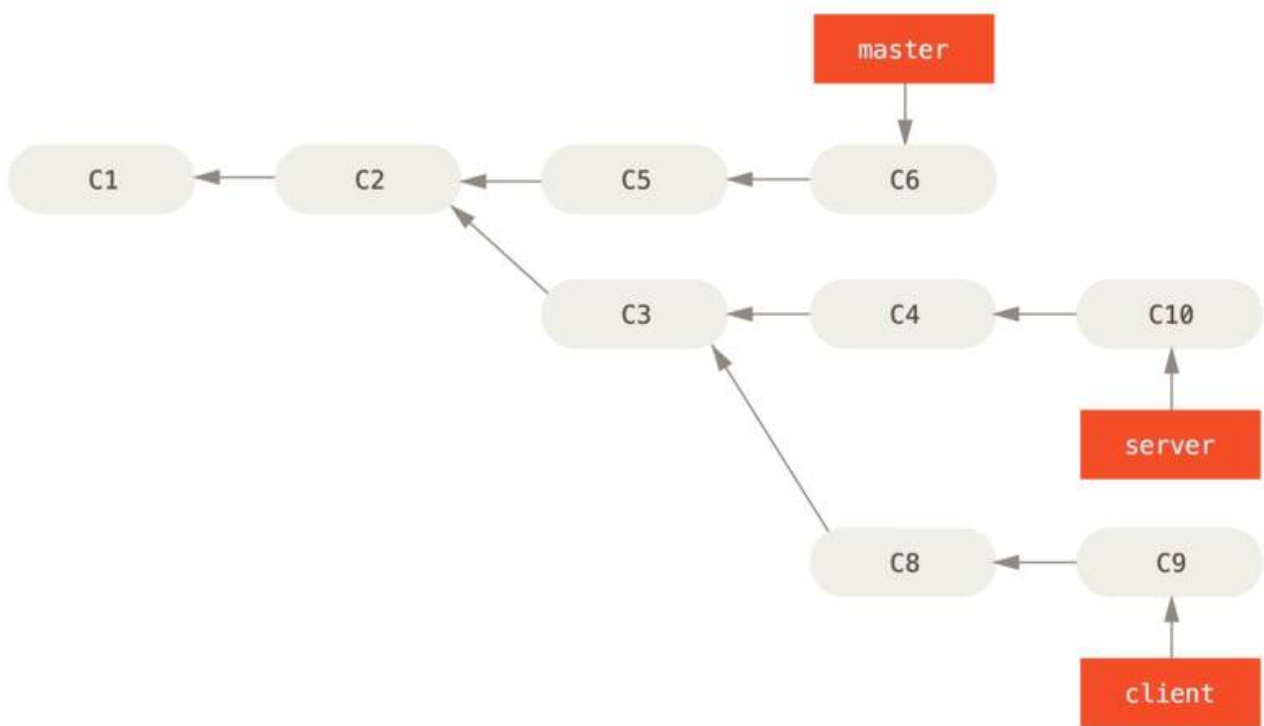
v1.8.5 v1.8.5-rc0 v1.8.5-rc1 v1.8.5-rc2 v1.8.5-rc3 v1.8.5.1 v1.8.5.2 v1.8.5.3 v1.8.5.4 v1.8.5.5

- 附注标签：`git tag -a v14 -m 'my version 14'`
- 轻量标签：`git tag v1.4-ew`

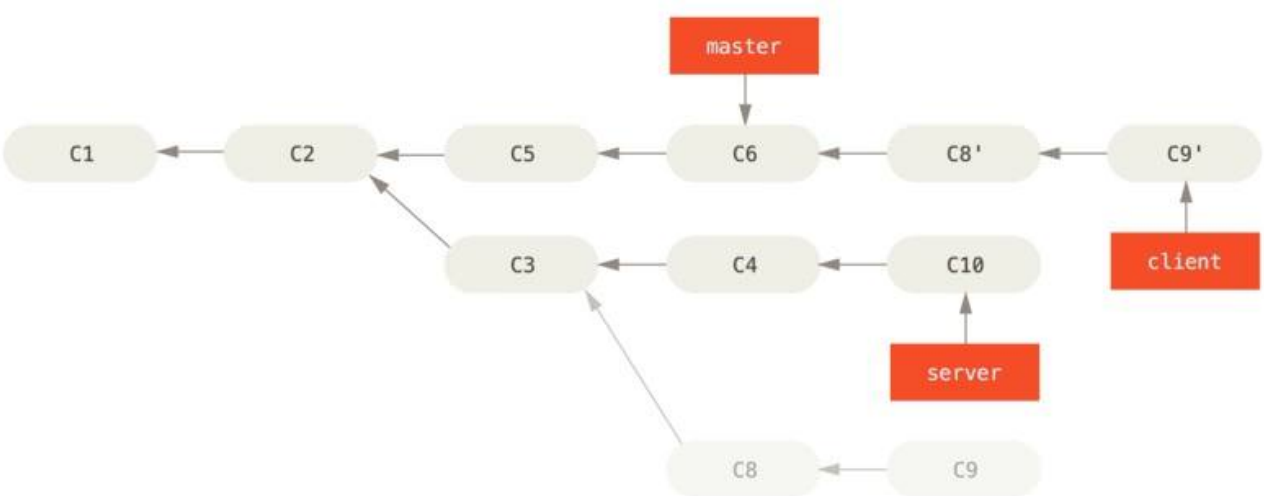
通过这些操作我们就可以给我们的提交打上标签了。但是有时候我们在提交完后才想起打上标签，所以我们可以利用 `git tag a v12 9fceb02` 给某个特定的提交打上标签。

`git merge` 和 `git rebase`

- `$ git rebase --onto master server client`



取出 client 分支，找出处于 client 分支和 server 分支的共同祖先之后的修改，然后把它们在 master 分支上重演一遍



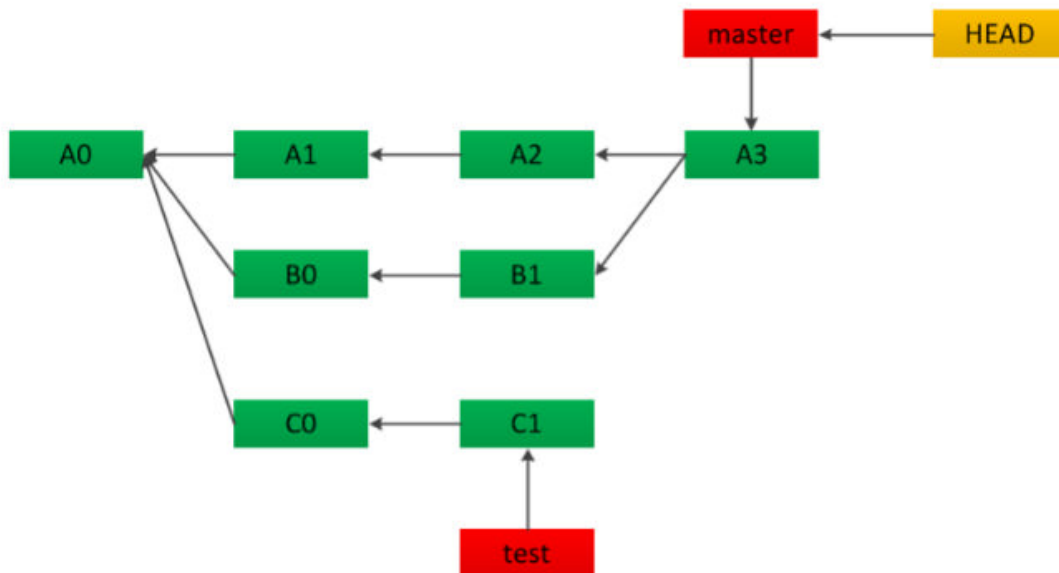
续上步骤，若master进行快速合并后，执行下面命令

- `$ git rebase master server` : 会将分支server放到master后

Git 进阶

选择版本

假设我们的版本如下所示，但是我们有时候想找到当前分支的父提交或是祖父提交。~ 和 ^ 就能大展神威了。



```
$ git log HEAD^
A2
$ git log HEAD^^
A1
$ git log HEAD^2
B1
$ git log HEAD~
A2
$ git log HEAD~~
A1
$ git log HEAD~2
A1
```

又或者我们想要选择一个区间的时候，下面可以看 .. 和 ... 还有 ^ 的区别。


```
$ git log master..test
C0 C1
$ git log ^master test
C0 C1
$ git log master...test
A1 A2 A3 C0 C1
```

git rebase -i HEAD~

提交重排

step1 : 修改文件提交

```
Administrator@GWCVLNP9LEXHQK4 MINGW64 /e/Git-log (dev)
$ git reflog
443e160 (HEAD -> dev) HEAD@{0}: rebase -i (finish): returning to refs/heads/dev
443e160 (HEAD -> dev) HEAD@{1}: rebase -i (pick): submit 2.txt
463e856 HEAD@{2}: rebase -i (pick): submit 3.txt
0869967 HEAD@{3}: rebase -i (start): checkout HEAD~2
f337b35 HEAD@{4}: commit: submit 3.txt
fbccbd3 HEAD@{5}: commit: submit 2.txt
0869967 HEAD@{6}: commit (initial): submit 1.txt
```



step2 : `$ git rebase -i HEAD~2` 修改之前的两个提交

```
Administrator@GWCVLNP9LEXHQK4 MINGW64 /e/Git-log (dev)
$ git rebase -i HEAD~2
```

step3 : 重新排序

```
MINGW64:/e/Git-log
pick 463e856 submit 3.txt
pick 443e160 submit 2.txt

# Rebase 0869967..443e160 onto 0869967 (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
```

step4 : 完成

```
Administrator@GWCVLNP9LEXHQK4 MINGW64 /e/Git-log (dev)
$ git log
commit 443e1608a5fc470038406484589702e5c9212cf9 (HEAD -> dev)
Author: smap <smap@233.com>
Date:   Wed Sep 19 22:11:43 2018 +0800

    submit 2.txt

commit 463e856e7e12ab6d471b0c833675dae22ebe5aed
Author: smap <smap@233.com>
Date:   Wed Sep 19 22:12:10 2018 +0800

    submit 3.txt

commit 086996791b8f5a80ca0870a8a2ac9c10fb312254
Author: smap <smap@233.com>
Date:   Wed Sep 19 22:11:04 2018 +0800

    submit 1.txt
```

删除：在执行 `$ git rebase -i HEAD~2` 后进入的界面将想要删除的记录去除

修改：在执行 `$ git rebase -i HEAD~2` 后进入的界面将想要修改的记录前面的标识改成edit

核弹级选项: filter-branch

- 从**所有提交**中删除一个文件

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

- 全局性地更换电子邮件地址

```
$ git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
    then
        GIT_AUTHOR_NAME="Scott Chacon";
        GIT_AUTHOR_EMAIL="schacon@example.com";
        git commit-tree "$@";
    else
        git commit-tree "$@";
    fi' HEAD
```

搜索调试

blame可以快速显示**每一行**最后一次修改是谁

```
git blame README.md
```

```
$ git blame README.md
^1ea2864 (Syunc      2018-09-04 23:30:11 +0800 1) ##### 关于版本控制:
^1ea2864 (Syunc      2018-09-04 23:30:11 +0800 2) >    记录一个或若干文件内容变化，以便将来查阅特定版本
修订情况
^1ea2864 (Syunc      2018-09-04 23:30:11 +0800 3)
^1ea2864 (Syunc      2018-09-04 23:30:11 +0800 4) ##### 发展:
^1ea2864 (Syunc      2018-09-04 23:30:11 +0800 5) 1. 本地版本控制系统: 复制整个项目文件夹从而形成新版本
```

也可以指定搜索范围

```
$git blame -L10,15 README.md #查看10-15行的信息
```

fast-forward

如果待合并的分支在当前分支的下游，也就是说没有分叉时，会发生快速合并，从test分支切换到master分支，然后合并test分支

p1-9

如果我们不想要快速合并，那么我们可以强制指定为非快速合并，只需加上--no-ff参数 p1-10

git checkout 和 **git reset** 的区别

首先进行概念辨析！

Git里有三个区域很重要

HEAD：指向最近一次commit里的所有snapshot（版本区）

Index：缓存区域，只有Index区域里的东西才可以被commit（暂存区）

Working Directory 用：户操作区域（工作区）

当你checkout分支的时候，git做了这么三件事情

- 1. 将HEAD指向那个分支的最后一次commit
- 2. 将HEAD指向的commit里所有文件的snapshot替换掉Index区域里原来的内容
- 3. 将Index区域里的内容填充到Working Directory里

此时，你的版本库，暂存区，工作区内容是一致的。

然后后~~~！

当你文件进行了修改的时候，Index区域和Working Directory的内容是不一致，即Working Directory的时间节点比Index区域要新，此时 `git add file` 之后Working Directory和 Index区域的内容就是一致的了。再进一步，你对Index区域进行了提交，那么这三个区域又是一致的了！

又然后后~~~！

```
补充
git revert :
git resvert HEAD #赋值前一个节点当做新的提交，相当于回到上一个提交的状态
```

ps：已经提交到远程仓库的 `commit` 不允许被 `git reset`

`git reset` 的一个栗子：

target指的是想要移动到哪去：

working index HEAD target					working index HEAD		
A	B	C	D	--soft	A	B	D
				--mixed	A	D	D
				--hard	D	D	D
				--merge	(disallowed)		

working index HEAD target					working index HEAD		
A	B	C	C	--soft	A	B	C
				--mixed	A	C	C
				--hard	C	C	C
				--merge	(disallowed)		

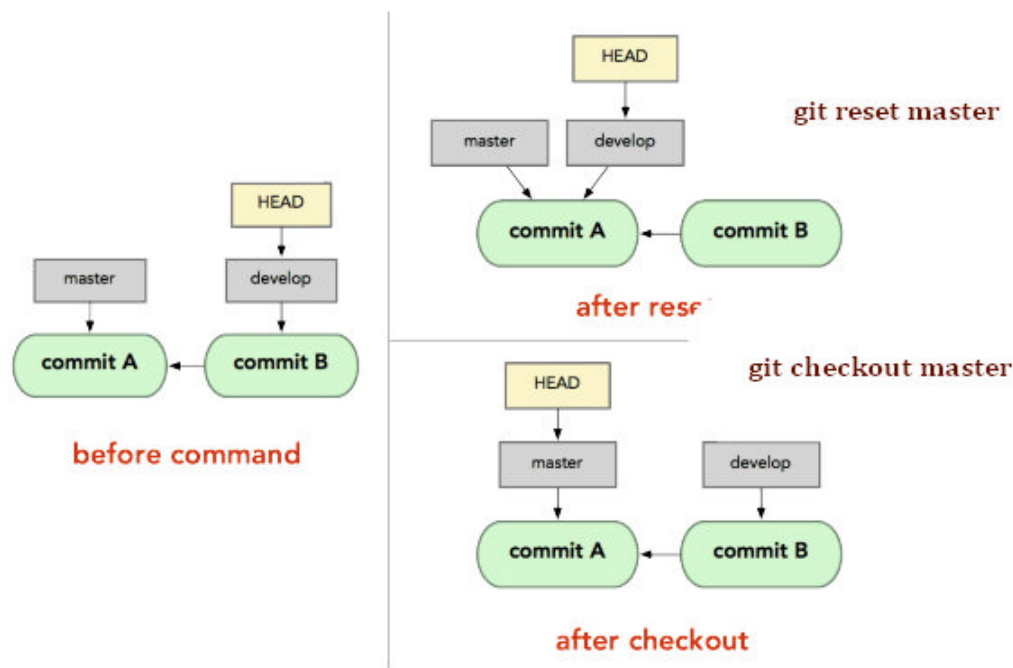
#####完美的分割线#####

checkout

checkout是会修改HEAD的指向，变更Index区域里的内容，修改Working Directory里的内容。这看上去很像reset --hard，但和reset --hard相比有两个重要的差别

- 1.
 - o reset会把working directory里的所有内容都更新掉
 - o checkout不会去修改你在Working Directory里修改过的文件
- 2.
 - o reset把branch移动到HEAD指向的地方
 - o checkout则把HEAD移动到另一个分支

一个栗子:



LAST 来个小结

	head	index	work dir	wd safe
Commit Level				
reset --soft [commit]	REF	NO	NO	YES
reset [commit]	REF	YES	NO	YES
reset --hard [commit]	REF	YES	YES	NO
checkout [commit]	HEAD	YES	YES	YES
File Level				
reset (commit) [file]	NO	YES	NO	YES
checkout (commit) [file]	NO	YES	YES	NO

“head”一列中的“REF”表示该命令移动了HEAD指向的分支引用，而“HEAD”则表示只移动了HEAD自身。
特别注意 “wd safe?” 一列，YES表示不会动你在work dir的修改，NO代表会动你在work dir的修改

参考资料

颜海静博客：<http://yanhaijing.com/git/2017/02/09/deep-git-4/>
图解Git：<https://marklodato.github.io/visual-git-guide/index-zh-cn.html?no-svg>
Pro Git，第二版，简体中文：<https://bingohuang.gitbooks.io/progit2/content/>
chanjarster博客：<https://segmentfault.com/a/1190000006185954>