

Git深入浅出



UGC前端组 冯伟

Trip.com Group™

2023/12/28

什么是版本控制(VCS)

版本控制是一种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的系统

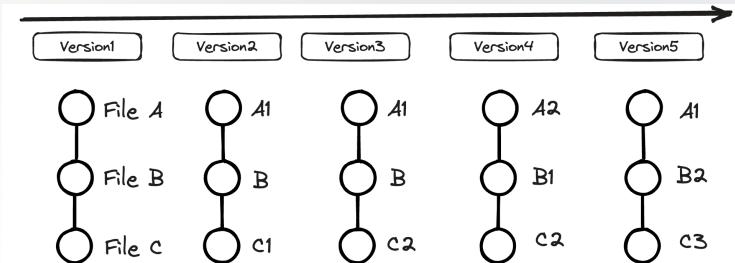
- 本地版本控制系统
- 集中化版本控制系统 (CVCS, centralized version control systems)
- 分布式版本控制系统 (DVCS, distributed version control system)

Git与其他版本控制系统的差别

最主要差别在于Git对待数据的方式

其他大部分系统以文件变更列表的方式存储信息，这类系统将他们存储的信息看作一组基本文件和每个文件随时间逐步累积的差异。他们通常被称作基于差异（delta-based）或增量（incremental）的版本控制系统。

Git更像是一个小型的文件系统，提供了许多以此为基础构建的超强工具，而不是一个简单的VCS。



实验: 创建一条提交

```
$ git commit -a -m 'Initial commit'
```

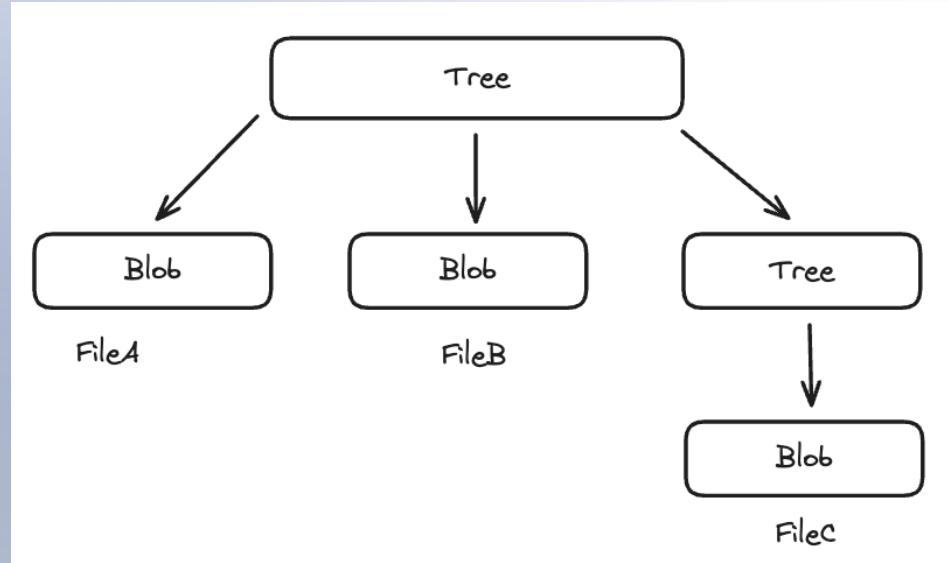


创建数据对象

1. `echo "some text" | git hash-object -w --stdin` 写入文件内容到Git数据库
2. `git cat-file -p <hash>` 查看对象内容(`-p` 选项指示自动判断文件类型)
3. `git hash-object -w <file>` 将文件内容写入对象数据库, 并返回该对象的哈希值
4. `git cat-file -t <hash>` 查看对象类型(`blob`)
5. `tree .git/objects` 查看存入的数据

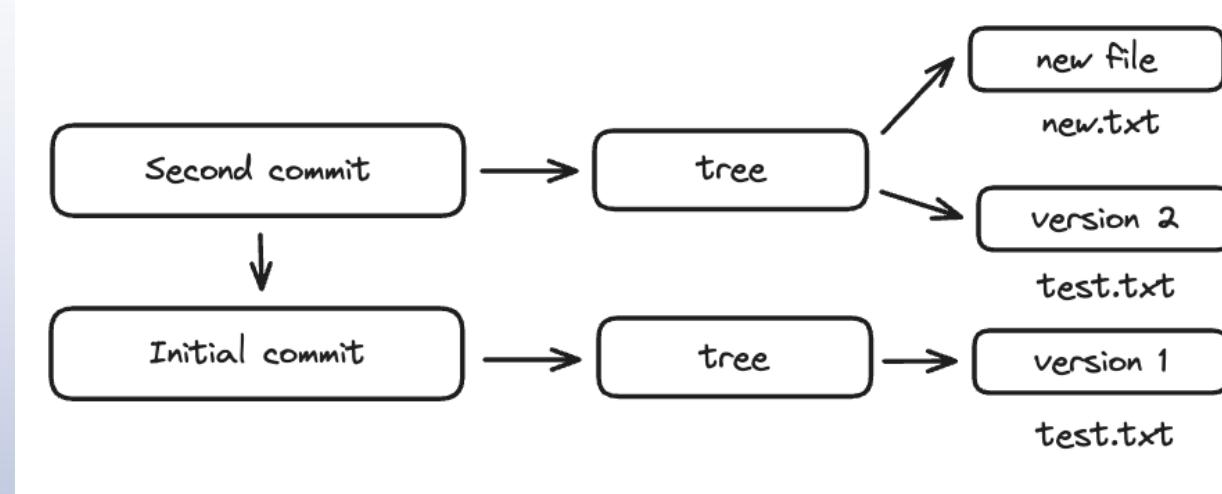
创建树对象

1. `git update-index --add --cacheinfo 100644 <hash> <file>` 将数据对象添加到暂存区
2. `git ls-files --stage` 查看暂存区内容
3. `git write-tree` 将暂存区内容写入对象数据库，并返回树对象的哈希值
4. `git cat-file -t <hash>` 查看对象类型(`tree`)



创建提交对象

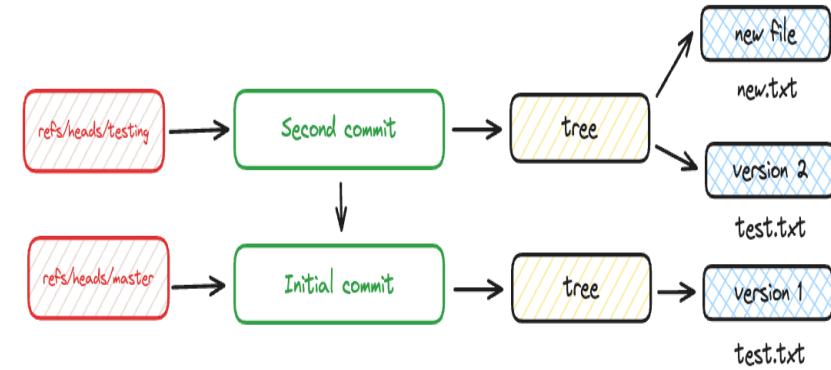
1. `git commit-tree <hash> -m 'Initial commit'` 创建提交对象,并返回提交对象的哈希值
2. `git log --stat <hash>` 查看提交对象内容



`references`引用

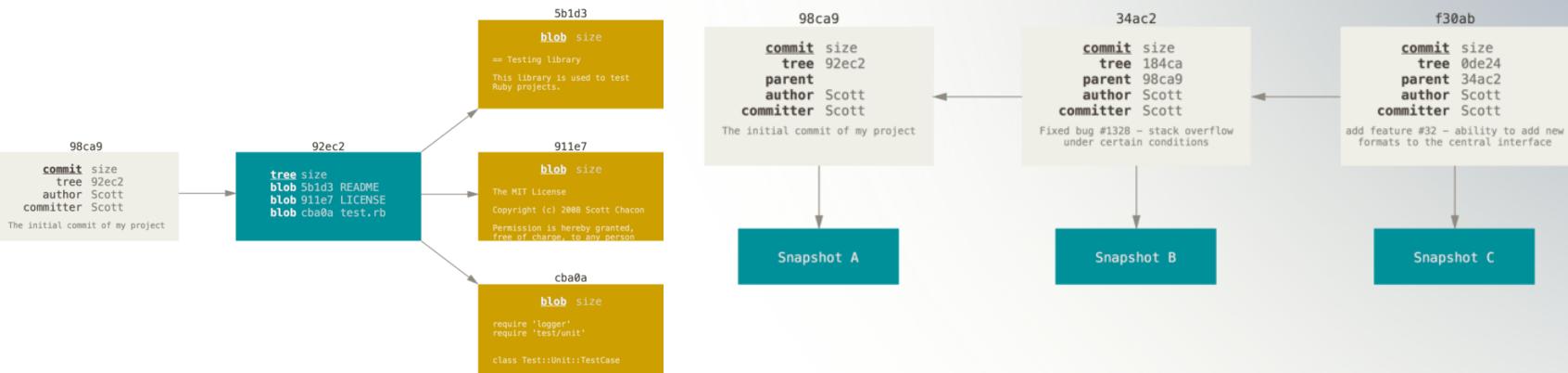
```
$ echo 4e5825 > .git/refs/heads/master
```

```
$ git update-ref refs/heads/master 4e5825
```



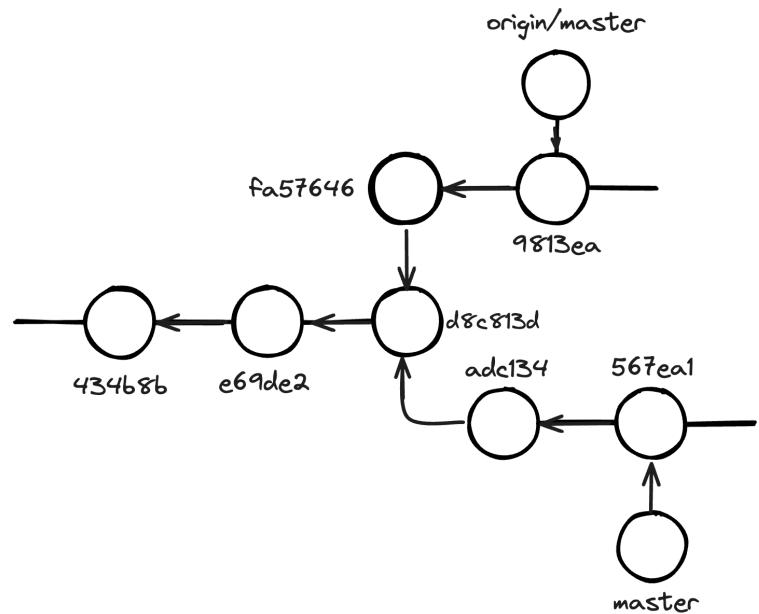
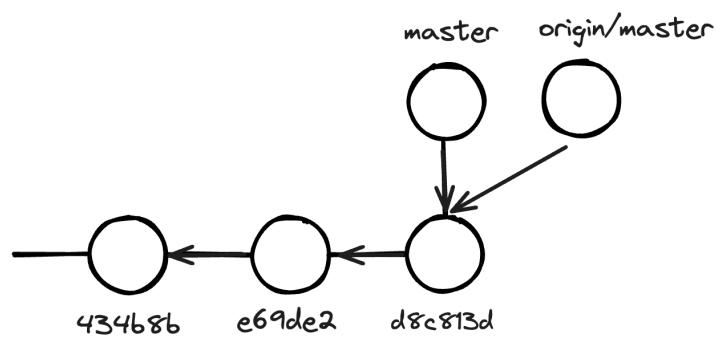
Git分支原理

Git保存的不是文件的差异或变化,而是一系列不同时刻的快照



Git的分支,本质上仅仅是指向提交对象的可变指针

远程分支



跟踪分支

跟踪分支是与远程分支有直接关系的本地分支

```
$ git checkout -b issue233 origin/issue233  
$ git checkout --track origin/issue233
```

```
$ git branch -u origin/server_issue233
```

```
$ git merge @{u}  
$ git rebase @{u}
```

```
$ git branch -vv  
* develop 0997a9f [origin/develop: behind 147] fix: 确认跳转发布页参数  
  release 65304cb [origin/release] Merge branch 'feat233' into 'release'
```

拉取

```
`git pull` = `git fetch` + `git merge`
```

```
A---B---C master on origin  
/  
D---E---F---G master
```

你仓库中的 origin/master

```
A---B---C origin/master  
/     \  
D---E---F---G---H master
```

变基 vs. 合并

```
A---B---C feat1  
/  
D---E---F---G master
```

变基

```
A'--B'--C' topic  
/  
D---E---F---G master
```

合并

```
A---B---C feat1  
/           \  
D---E---F---G---H master
```

变基的风险

变基操作的实质是丢弃一些现有的提交,然后相应地新建一些内容一样但实际上不同的提交

```
A--- feat1  
/  
D--- E--- master
```

userA (rebase)

```
A' --- feat1  
/  
D--- E--- master
```

userB (HEAD)

```
A---B--- feat1  
/  
D--- E--- master  
 \  
 A' --- origin/feat1
```

变基 vs. 合并

只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作

cherry-pick / rebase / revert

- cherry-pick: 获得在单个提交中引入的变更, 然后尝试将作为一个新的提交引入到当前分支上
- rebase: 自动化的 cherry-pick 命令, 计算出一系列的提交, 然后再以它们在其他地方以同样的顺序一个一个的 cherry-picks 出它们。
- revert: 本质上就是一个逆向的 git cherry-pick 操作, 将提交中的变更以完全相反的方式应用到一个新创建的提交中, 本质上就是撤销或者倒转。

重置

树

用途

HEAD

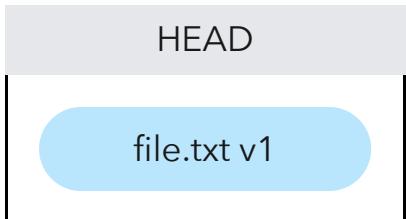
上次提交的快照,下次提交的父节点

索引

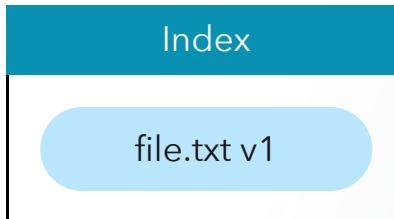
预期的下次提交快照

工作目录

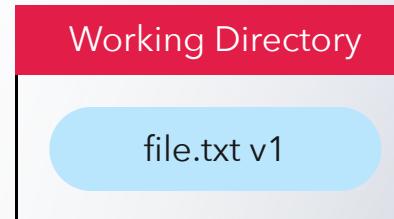
沙盒



`git commit`



`git add`



重置

- `--soft` 仅移动HEAD指针指向
- `--mixed` 移动HEAD指针和重置索引(取消暂存文件,与`git add`相反)
- `--hard` 移动HEAD指针、重置索引和重置工作目录

误 `reset` 后的数据恢复:

1. `git reflog` 引用日志

```
$ git reflog  
$ git reset --hard HEAD@{1}
```

2. `git fsck` 检查数据库完整性

```
$ git fsck --full  
$ git reset --hard HEAD@{1}
```

``checkout`` 与 ``reset`` 的区别

1. ``git checkout <branch>`` 与 ``git reset --hard <branch>`` 类似, 但是 ``checkout`` 对工作目录是安全的, 它会通过检查来确保不会将以更改文件丢掉(先试着简单合), 而 ``reset`` 则会强制覆盖工作目录
2. ``git reset master``, develop自身和master指向同一个提交; ``git checkout master``, develop不会移动, HEAD自身会移动(指向master)

撤销合并

1. 移动分支 `git reset --hard <hash>`
2. 还原提交 `git revert -m 1 HEAD~`

重写历史

- 重写最后一次提交
- 修改多个提交信息

重写最后一次提交

- 修改上次的提交信息
- 添加遗漏的文件
- 修改上次提交的文件

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

修改多个提交信息

```
$ git rebase -i HEAD~3
```

- `edit` 修改提交信息
- `reword` 仅修改提交信息
- `squash` 与前一个提交合并
- `fixup` 与前一个提交合并, 但丢弃提交信息

批量改写历史提交 ⚠

- 误提交了涉密内容、大文件 (alternative: BFG Repo-Cleaner)

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
```

- 修改提交人、作者的姓名、邮箱

```
$ git filter-branch --commit-filter '
  if [ "$GIT_AUTHOR_EMAIL" = "mekhi@work" ];
  then
    GIT_AUTHOR_NAME="Mekhi El";
    GIT_AUTHOR_EMAIL="mekhi@personal.com";
    git commit-tree "$@";
  else
    git commit-tree "$@";
  fi' -- --all
```

- 将子目录独立成单独的仓库

```
$ git filter-branch --subdirectory-filter apps/miniapp HEAD
```

Git工具

提供给Git的SHA-1字符数量不少于4个且无歧义, 即可获得此次提交

- 查看引用日志: `git reflog`
- 使用祖先引用: `^``(脱字符), ``~``(波浪号), 例: `git reset HEAD~3^2``
- 提交区间
 - ``git log master..develop``: 在develop而不在master上的提交
 - ``git log develop HEAD ^master``: 在当前分支和develop不在master上的提交
 - ``git log master...develop --left-right --oneline``: 被其中之一包含但不被同时包含的提交
- 贮藏与清理: `git stash apply`, `git stash pop`, `git stash drop`
- 清理工作目录: `git clean -f -d` 从工作目录中移除未跟踪的文件和目录, `-x`选项会移除被忽略的文件

调试

- `git bisect` 二分查找提交历史
 1. `git bisect start`
 2. `git bisect bad` 标记当前提交为坏提交
 3. `git bisect good <hash>` 标记一个已知的好提交
 4. Git检出中间提交, 使用`git bisect <good | bad>`标记为好或坏
 5. `git bisect reset` 退出二分查找
- `git blame` 文件标注, 查看文件的每一行是谁修改的
- `git grep` 从提交历史、工作目录、索引中查找文本、正则表达式

配置 Git

- 配置文件
 - `/etc/gitconfig` : 系统级配置文件
 - `~/.gitconfig` 或 `~/.config/git/config` : 用户级配置文件, 对应 `git config --global`
 - `.git/config` : 仓库配置文件, 对应 `git config --local`
- 属性: `.gitattributes`
 - `*.docx diff=word` 匹配 `.docx` 文件使用 `word` 过滤器(将 word 文档转为可读文本文件, 便于查看差异),
增加配置 `git config diff.word.textconv docx2txt`
 - `*.pbxproj binary` 将所有该文件作为二进制文件处理, 不进行差异比较
- 忽略文件: `.gitignore`
- 环境变量
- hooks
 - 客户端钩子: 提交工作流钩子、电子邮件工作流钩子、其他钩子
 - 服务器端钩子: `pre-receive`、`update`、`post-receive`

END