

# 前端工程化

—Git基础

# ■ 01 什么是Git

Git是一款用来执行版本控制的软件，和Subversion、CVS具有相似的特性，例如可以保存修改历史、能恢复某些历史的修改等等。

同时，Git与传统的版本控制软件在设计思想和实现上差异比较大。

# ■ Git与早期版本控制软件区别

- 1、早期版本控制软件，例如Subversion，其依赖中心服务器以及客户端到服务器的网络连通性，服务器充当中心结点，所有客户端通过网络协议执行版本控制操作，如果客户端无法连接网络服务，则无法执行相关操作，这种软件架构称为**集中式版本控制**，其优点在于部署简便，不需要考虑多个客户端如何协作。
- 2、与集中式对应的则是**分布式版本控制**，git就是分布式版本控制软件的一种，每个客户端都拥有完整的仓库代码，除非需要远程协作，大部分操作是发生在本地文件系统，摆脱对于网络连通性的强依赖，甚至中心服务器故障时，可以使用某个客户端充当新的中心节点。

# Git与早期版本控制软件区别

除了分布式之外，git对待版本更新的方法和Subversion有些不同，Subversion会将它们存储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，基于差异去控制版本。

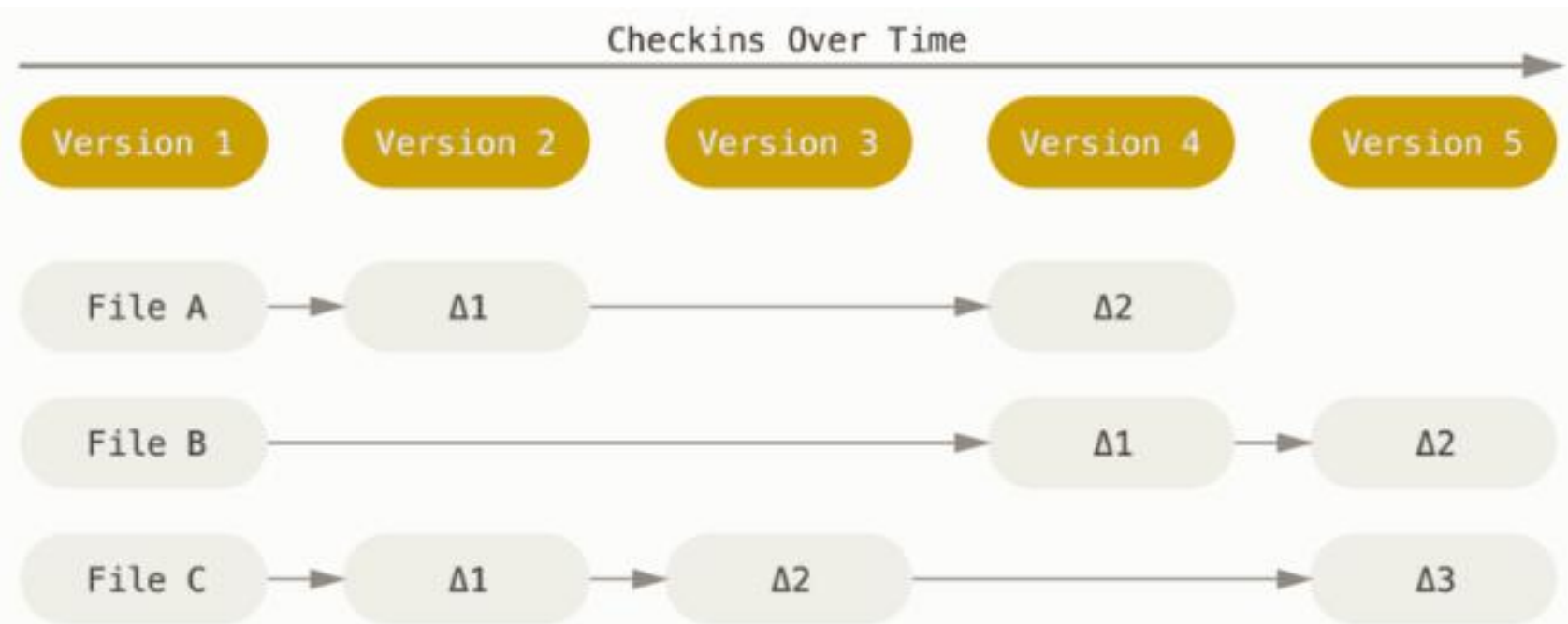


Figure 4. 存储每个文件与初始版本的差异.

# Git与早期版本控制软件区别

Git 对待数据更像是一个快照流，Git把每次提交保存为文件系统的快照，出于效率考虑，如果没有修改则不保存，而是只保留一个链接指向之前存储的文件。

整体上看Git更像是个内容寻址文件系统。

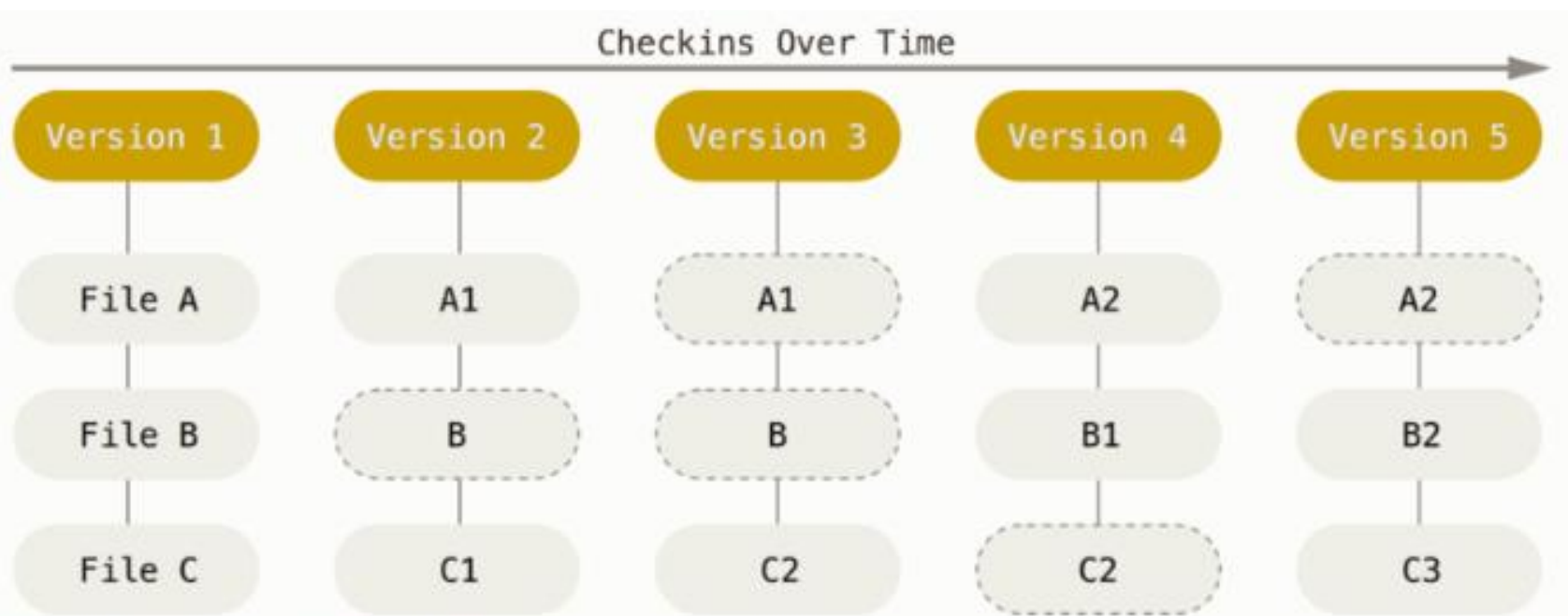


Figure 5. 存储项目随时间改变的快照.

# Git简史

Linux内核开源社区从2002年开始使用BitKeeper，同样是一款分布式版本控制软件，在2005年BitKeeper的商业公司同Linux内核开源社区的合作关系结束，于是Linux内核开源社区不得不另起炉灶。起初他们对Git有以下期待：

1. 速度
2. 简单的设计
3. 支持非线性开发模式即允许大量并行开发分支
4. 完全分布式
5. 有能力高效管理类似 Linux 内核一样的超大规模项目

## ■ 02 基本原理

1. 对象数据库

2. 树对象

3. 提交对象

# ■ 对象状态

在Git中对象包括三种状态，分别是已修改(modified)、已暂存(staged)、已提交(commited)

- 已修改表示已经修改了文件，但是还没有保存到本地文件数据库；
- 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中；
- 已提交表示数据已经安全地保存在本地数据库中。



# 对象状态

以上三种状态会让我们的 Git 项目拥有三个阶段：工作区、暂存区以及 Git 目录。其中已修改（Modified）对应工作区，已暂存（Staged）对应暂存区，已提交（Committed）对应 Git 目录。

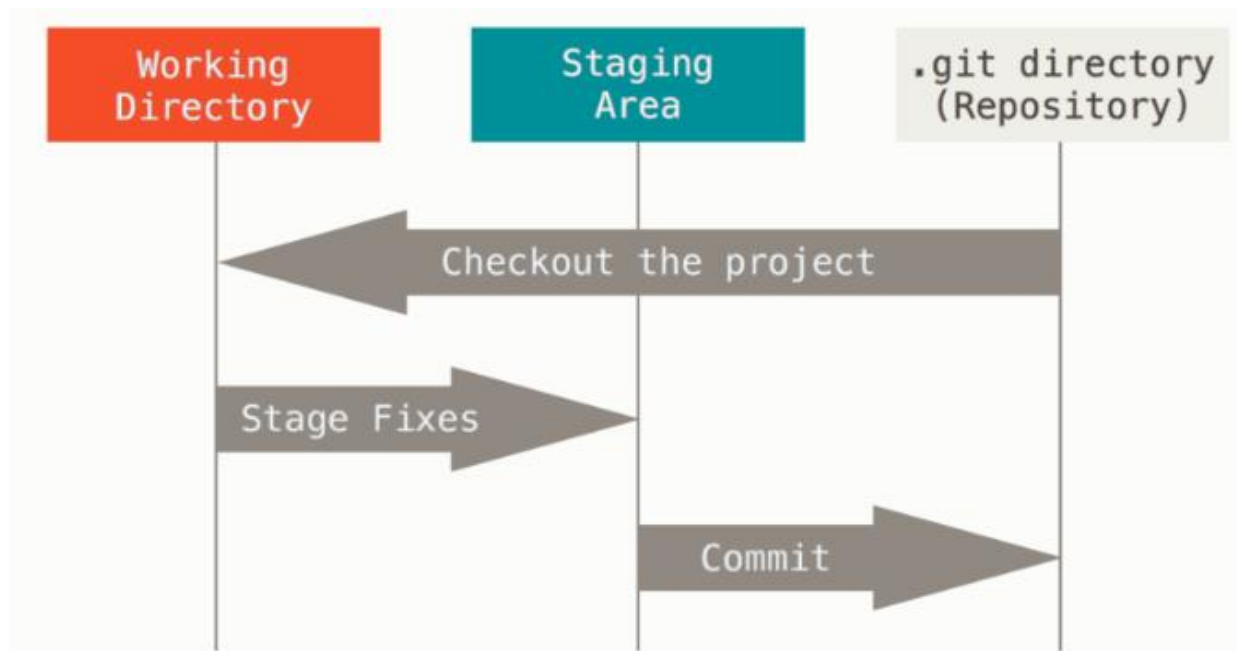


Figure 6. 工作目录、暂存区域以及 Git 仓库.

# 对象数据库

所有Git相关的数据都是存储在.git目录下，其中Git对象存储在.git/objects即对象数据库。

可以通过git hash-object，实现插入一个对象到对象数据库中。

```
$ echo 'An example' > Demo.txt
```

```
$ git hash-object -w Demo.txt
```

```
warning: LF will be replaced by CRLF in Demo.txt.
```

```
The file will have its original line endings in your working directory
```

```
6d833c88d580e266cc6f8c553d50a53305d451b4
```

这时系统会返回长度为40的字符哈希校验和

# 对象数据库

此时执行find指令，可以看到对象数据库新增一个对象

```
zha@WEDESKTOP: /C:/zha/jw/git/objects/0524 (master)$ find .git/objects -type f
.git/objects/6d/833c88d580e266cc6f8c553d50a53305d451b4
```

# 对象数据库

如果想要查看这个对象的值，则可以使用git cat-file 输入刚刚返回的哈希校验和获取：

```
ZHUJW@DESKTOP-ER1QLE8V MINGW64 /C:/ZHUJW/gitee/0524 (mas
$ git cat-file -p 6d833c88d580e266cc6f8c553d50a53305d451b4
An example
```

# 对象数据库

如果修改demo.txt文件内容并再次添加到对象数据库中，这时查看对象数据库则会发现又新增1个对象

```
$ echo 'Add another example' > Demo.txt
```

```
$ git hash-object -w Demo.txt
```

查看对象数据库


```
$ find .git/objects -type f
```

```
.git/objects/6d/833c88d580e266cc6f8c553d50a53305d451b4  
.git/objects/d5/467539e337051268e46a6c65a4f6731f26fde5
```

# 对象数据库

依然可以使用之前的哈希校验和获取第一个版本对象

```
$ git cat-file -p 6d833c88d580e266cc6f8c553d50a53305d451b4  
An example
```



**结论：**将数据保存到本地对象数据库中，其中key是哈希校验和，只要有哈希校验和和完整对象数据库就可以获取任何一个版本的数据对象

# ■ 树对象

类似于 UNIX 文件系统的方式存储文件路径，所有内容均以树对象和数据对象的形式存储。

其中树对象对应了 UNIX 中的目录项，数据对象则大致上对应了 inodes 或文件内容。

一个树对象包含了一条或多条树对象记录，每条记录含有一个指向数据对象或者子树对象的指针以及相应的模式、类型、文件名信息。

# 树对象

```
$ mkdir -p localpath
```

```
$ echo 'Add localpath object' > localpath/Demo.txt
```

```
$ git hash-object -w localpath/Demo.txt
```

先通过git update-index将文件提交到暂存区，再通过git write-tree将暂存区对象写入树对象。

```
$ git update-index --add --cacheinfo 100644 39ced83d8b09f321ef3c3ab547d17cdc10a09c13 localpath/Demo.txt
$ git update-index --add --cacheinfo 100644 d5467539e337051268e46a6c65a4f6731f26fde5 Demo.txt
$ git write-tree
9560757f4609f01c3f6ed83ca5c520b341740ca9
```

git write-tree执行之后会在系统中生成一个新的树对象，并且系统返回该对象的Hash校验和



# ■ 树对象

依然可以通过git cat-file看到对象类型是树对象：

```
$ git cat-file -p 9560757f4609f01c3f6ed83ca5c520b341740ca9
100644 blob d5467539e337051268e46a6c65a4f6731f26fde5    Demo.txt
040000 tree 76470c893368ced796298fc5c1f64b74e7c3a5ec    localpath
```

树对象可以引用另一个树对象，继续使用git cat-file可以查看：

```
$ git cat-file -p 76470c893368ced796298fc5c1f64b74e7c3a5ec
100644 blob 39ced83d8b09f321ef3c3ab547d17cdc10a09c13    Demo.txt
```

# 提交对象

提交对象是引用树对象并记录谁在什么时候增加树对象以及父提交，相当于树对象的操作记录。

```
$ echo 'New tree object' | git commit-tree 76470c893368ced796298fc5c1f64b74e7c3a5ec  
67d8989a8236d1e07b919992be55d83d1b38af68
```

通过git log就可以看到提交的记录

```
$ git log 67d8989a8236d1e07b919992be55d83d1b38af68  
commit 67d8989a8236d1e07b919992be55d83d1b38af68  
Author: sinkingwen <[REDACTED]@com>  
Date: Tue May 23 09:41:53 2023 +0800  
  
New tree object
```

# 引用对象

引用对象就是提交对象的别名，我们可以用git update-ref创建引用对象，如下所示：

```
$ git update-ref refs/demo 67d8989a8236d1e07b919992be55d83d1b38af68  
zhuju@DESKTOP-LRIQL8V MINGW64 /c/zhuju/gitProject/0524 (master)  
$ git cat-file -t refs/demo  
commit
```

这时就可以使用refs/demo来引用提交对象，这样的表示方式更加方便记忆，而这基本就是Git分支的本质。

# Git对象

Git所做的工作实质就是将被改写的文件保存为数据对象，更新暂存区，记录树对象，最后创建一个指明了顶层树对象和父提交的提交对象。

这三种主要的 Git 对象——数据对象、树对象、提交对象——最终均以单独文件的形式保存在 `.git/objects` 目录下。

## 03 Git常见操作

- 初始化git: `git init`
- 获取远程git目录到本地: `git clone git@github.com:yourname/newrepository.git`
- 回顾变更: `git log --pretty=raw`
- 检查工作区和暂存区状态:
  1. `echo 'hello,new world' > hello.txt`
  2. `git status`

```
On branch master
Changes not staged for commit:
  (use "git add <file> ..." to update what will be committed)
  (use "git restore <file> ..." to discard changes in working directory)
        modified:   hello.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

# 如何提交变更

1. `echo 'hello,world' > hello.txt`
2. `git add hello.txt` (将文件增加到暂存区)
3. `git commit -m "hello,world"` (给文件生成一个提交对象)

在一些受管控的项目上，提交代码到 git 服务器后，还需要经过审核确认才正式合入版本。如果提交的代码审核不通过，需要再次修改提交。由于是修改同一个问题，我们可能不希望生成多个 commit 信息，会显得改动分散，看起来改动不完善，所以想要在本地的已有 commit 信息上再次提交改动，而不是在已有的 commit 上再新增一个 commit。使用命令：

- `git commit --amend`

## 如何提交变更

```
$ echo 'Add ccc content' > ccc.txt
```

```
zhuju@DESKTOP-LRIQL8V MINGW64 /c/zhuju/gitProject/0524/localpath (master)
```

```
$ git add . && git commit -m 'Add ccc to localpath/ccc.txt'
```

```
warning: LF will be replaced by CRLF in localpath/ccc.txt.
```

```
The file will have its original line endings in your working directory
```

```
[master 33137b1] Add ccc to localpath/ccc.txt
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 localpath/ccc.txt
```

# ■ 临时保留修改

有时在工作区的修改还没有达到提交状态，这时如果有一个临时的想法想去实现，可以使用`git stash`来临时保存工作区状态。

```
echo 'hello,new world' > hello.txt
```

```
git stash
```

当临时想法实现完毕后，可以使用`git stash apply`来恢复刚刚的工作区。

```
git stash apply
```

恢复完毕，建议清理：

```
git stash clean
```





# 撤销修改

## ◆ 撤销工作区修改

1. `echo 'hello,new world' > hello.txt`
2. `git checkout -- hello.txt`

## ◆ 撤销提交到暂存区

`echo 'hello,new world' > hello.txt`

`git add hello.txt`

使用 **`git reset HEAD hello.txt`** (从暂存区退回到工作区)

## ◆ 撤销提交

`git reset --hard HEAD^1`

回滚到上一个版本, 也可以通过提供哈希校验和回滚到任意提交对象

# ■ 分支管理

新建分支: `git switch -c feature`

切换分支: `git switch feature`

查看分支: `git branch`

(检查git的分支以及当前目录所在的分支, 当前分支前面会标一个\*号)

删除分支: `git branch -d feature`

(删除分支还能找回吗? )

# 分支合并

在大型项目开发中，合并分支往往非常频繁，这是因为需要经常集成其他人的贡献和代码，最常用的合并是通过git merge实现。

1. `git switch master`
2. `git merge feature`

合并并不总是很顺利，有时会伴随冲突，如果当前分支的父提交不是要合并的分支，则无法使用快速合并，如下为快速合并场景：

```
git switch master
git switch -c test
echo 'hello,new world' > hello.txt
git add hello.txt
git commit -m "merge branch"
git switch master
git merge test
```

```
Updating 0a7d684..44d6a4c
Fast-forward
 hello.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

可以看到由于test分支的父提交和当前master的提交一致，符合快速合并原则，可以使用test分支的最新提交并设置父提交为master的最新提交

# 分支合并

如果test分支创建后，master分支又产生新的提交对象，就不符合快速合并原则，需要人工处理。

```
git switch master
git switch -c test
echo 'hello,new world' > hello.txt
git add hello.txt
git commit -m "merge branch"
git switch master
echo 'hello,master world' > hello.txt
git add hello.txt
git commit -m "master branch"
git merge test
```

冲突的问题会将两个版本进行标记

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result
```

```
<<<<<< HEAD
hello,master world
=====
hello, new world
>>>>>> test
```

# 分支合并

## ➤ 查看冲突：

1、git status来查看冲突文件

2、用编辑器打开冲突文件，查看冲突内容。

3、冲突内容分隔线怎么看？

未冲突的内容（两个分支都未改动） 在分隔线外面

<<<<<<< HEAD

Git当前所在分支修改的内容（准确来说是HEAD指针指向的分支修改的内容）

=====

要合并过来的分支修改的内容

>>>>>>> branch\_to\_merge

4、手动编辑冲突文件内容，删除三行标记线，保存文件

5、回到master分支，git add 文件名

6、git commit -m 'commit message'

# 远程协作

将本地的分支关联到远程分支，实际上就是创建一类特殊的远程引用对象，这个引用对象是只读的。

可以通过`git push`将本地的对象推送到远程，如下就是把本地的`master`分支推送到远程，前提是需要对目标仓库有写权限。

1. `git remote add origin git@github.com:yourname/newrepository.git`
2. `git push origin master`

注意，如果使用`git commit --amend`则在push时需要使用-f参数

```
git push -f origin master
```

# ■ 与远程协作

使用git pull来拉取远程数据到本地，自动merge

git pull

使用git fetch来拉取远程数据到本地，用户触发merge

git fetch

相当于：

git pull = git fetch + git merge