

Git应用开发详解



讲师：张龙

版权所有 圣思园

正确发音

- git [ɡɪt]
- 官方网站: <http://www.git-scm.com>



The image is a screenshot of the Git website's header and main content area. It features the Git logo (a red diamond with a white 'g') and the tagline '--everything-is-local'. A search bar is located in the top right corner. The main text describes Git as a free and open source distributed version control system. To the right, there is a diagram illustrating Git's distributed nature with multiple repositories connected by lines. At the bottom left, there is a 'Try Git' button and a link to learn Git in a browser.

git --everything-is-local

Search entire site...

Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient **staging areas**, and **multiple workflows**.

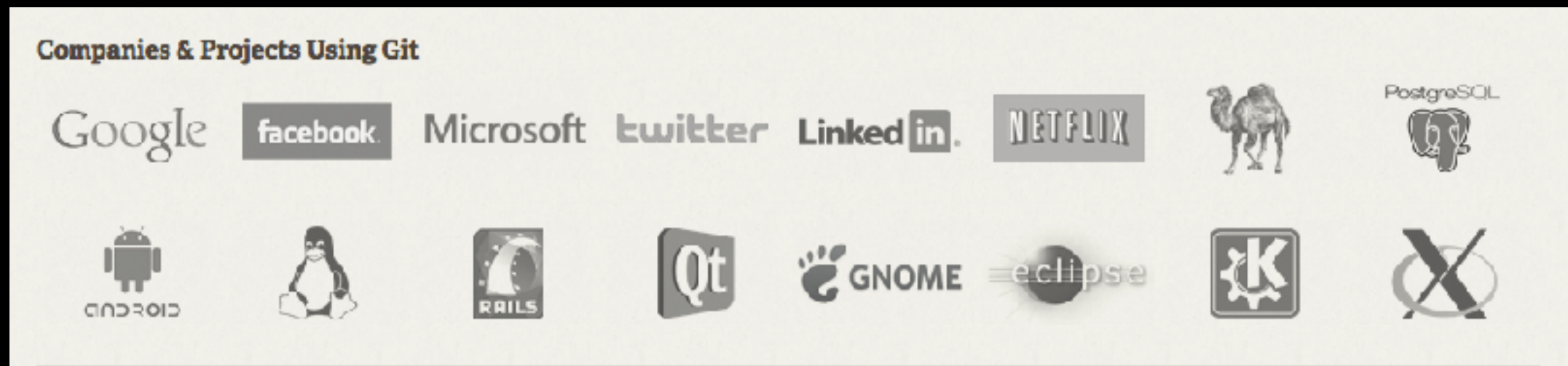
 Learn Git in your browser for free with **Try Git**.

Git简史

- Linux内核开源项目有着为数众广的参与者。一开始整个项目组使用BitKeeper来管理和维护代码。2005年，BitKeeper不再能免费使用，这就迫使Linux开源社区开发一套属于自己的版本控制系统。
- 自诞生于2005年以来，Git日臻成熟完善，它的速度飞快，极其适合管理大型项目，它还有着令人难以置信的非线性分支管理系统，可以应付各种复杂的项目开发需求。

谁在使用Git

- 众多的开源、非开源项目已经逐步由SVN迁移到了Git



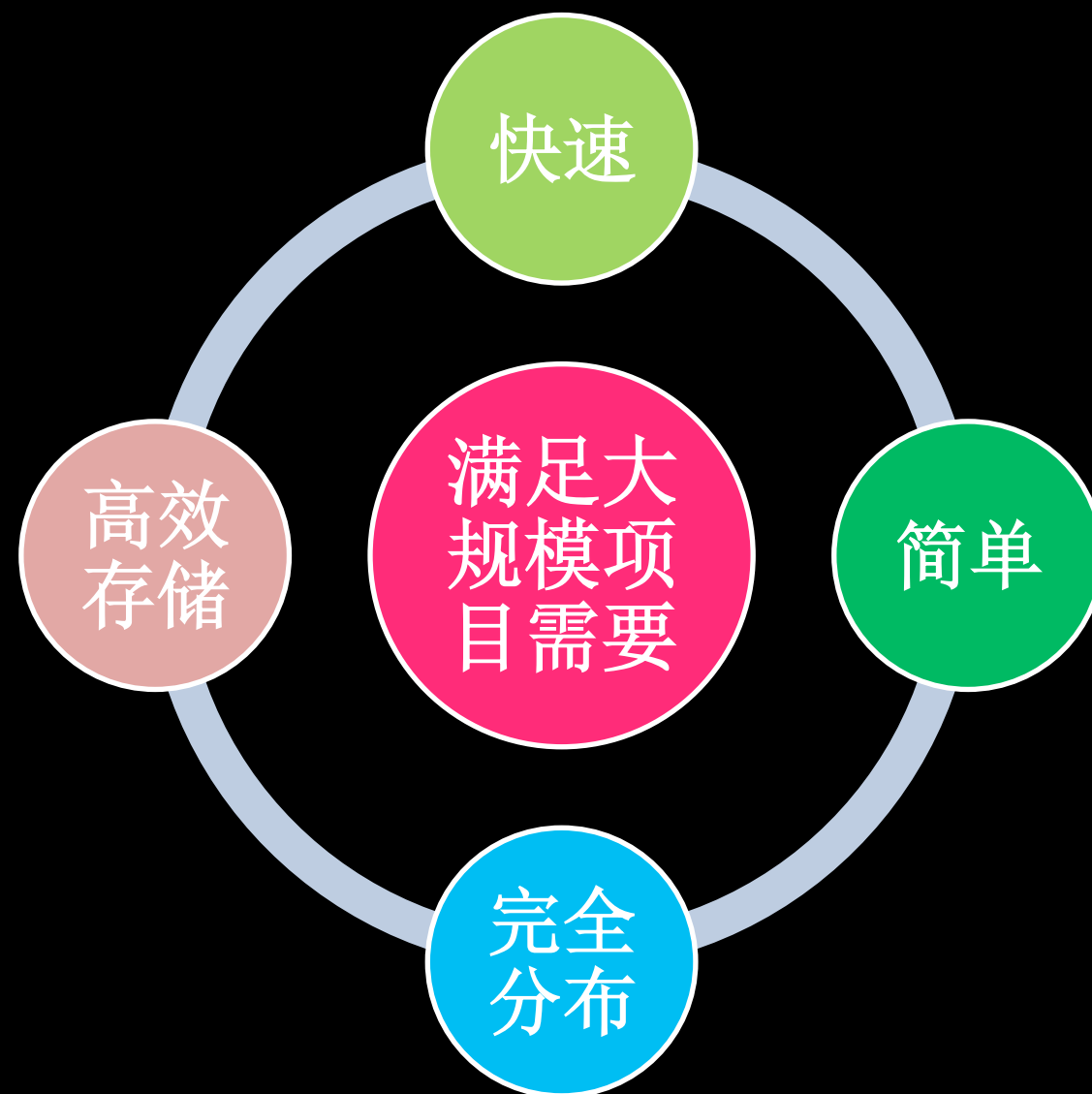
CVS、SVN与Git

- 集中式版本控制系统 (CVCS)
- 分布式版本控制系统 (DVCS)
- 有了Git, 编程真正成为了一种乐趣

Git、GitHub与GitLab

- Git是一个版本控制软件
- GitHub与GitLab都是用于管理版本的服务端软件
- GitHub提供免费服务（代码需公开）及付费服务（代码为私有）
- GitLab用于在企业内部管理Git版本库，功能上类似于GitHub

Git设计目标

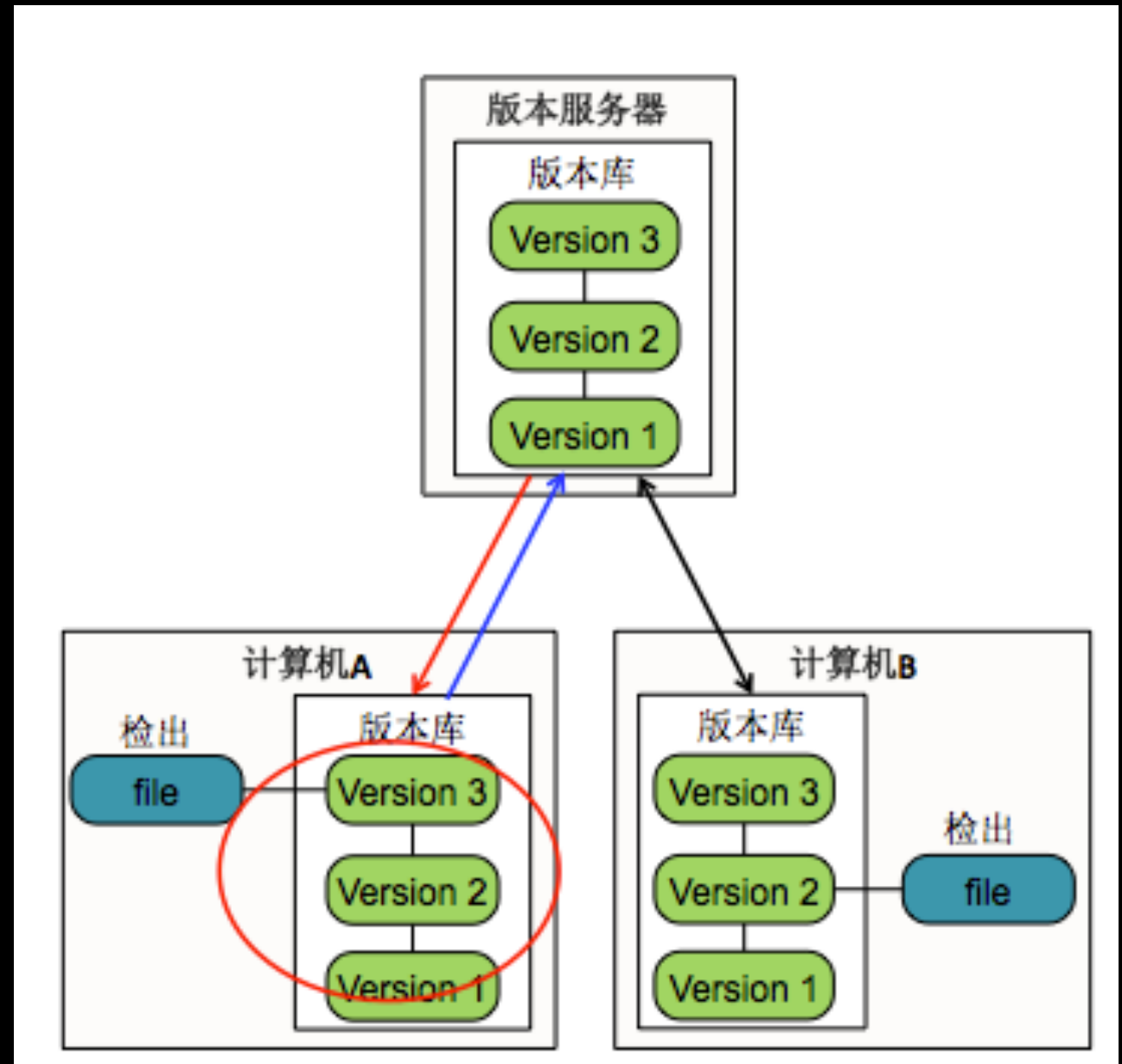


为什么要使用Git

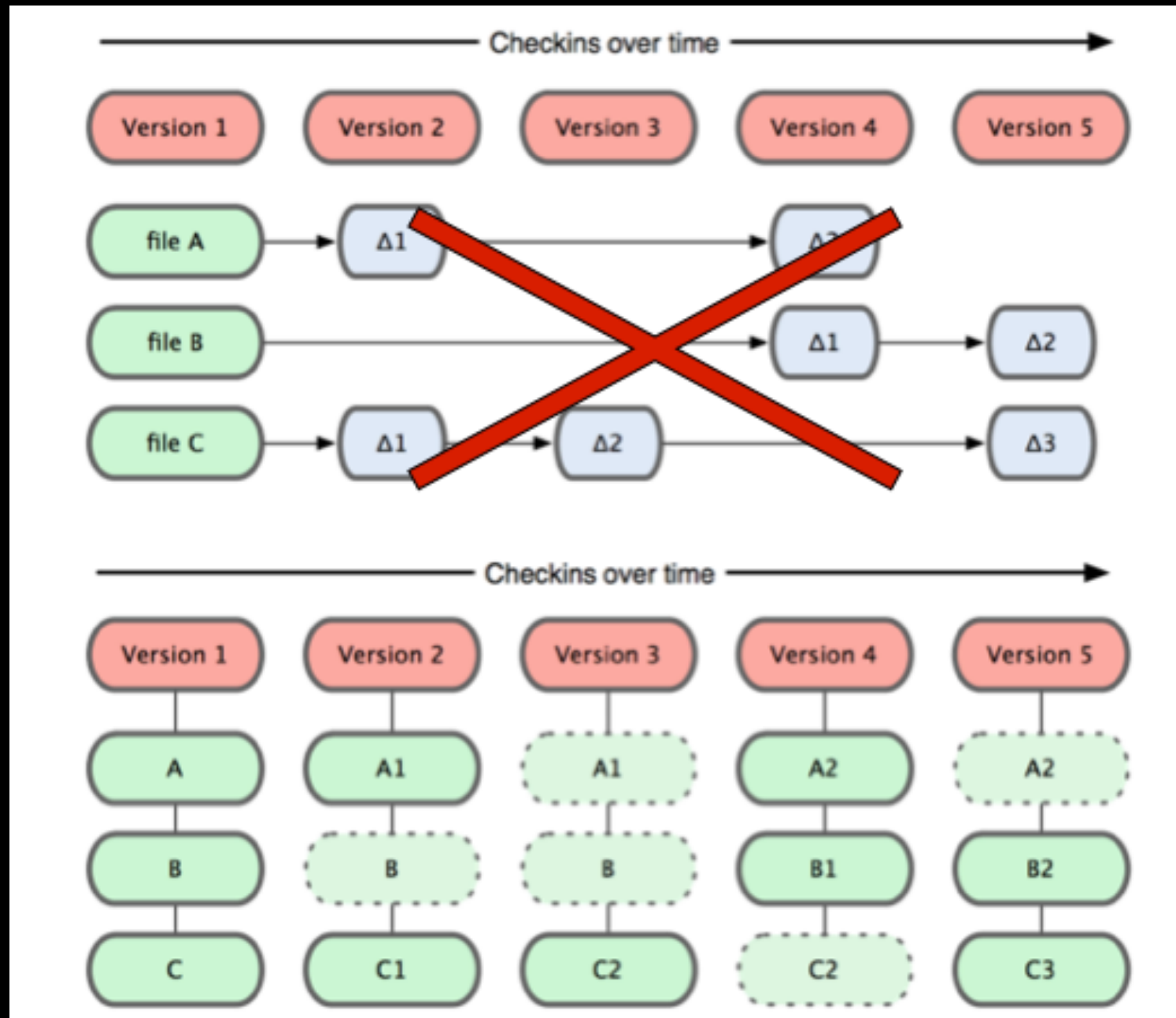
- 本地建立版本库
- 本地版本控制
- 多主机异地协同工作
- 重写提交说明
- 有后悔药可以吃
- 更好用的提交列表
- 更好的差异比较
- 更完善的分支系统
- 速度极快

Git工作模式

- 版本库初始化
 - 个人计算机从版本服务器同步
- 操作
 - 90%以上的操作在个人计算机上
 - 添加文件
 - 修改文件
 - 提交变更
 - 查看版本历史等
- 版本库同步
 - 将本地修改推送到版本服务器



Git文件存储

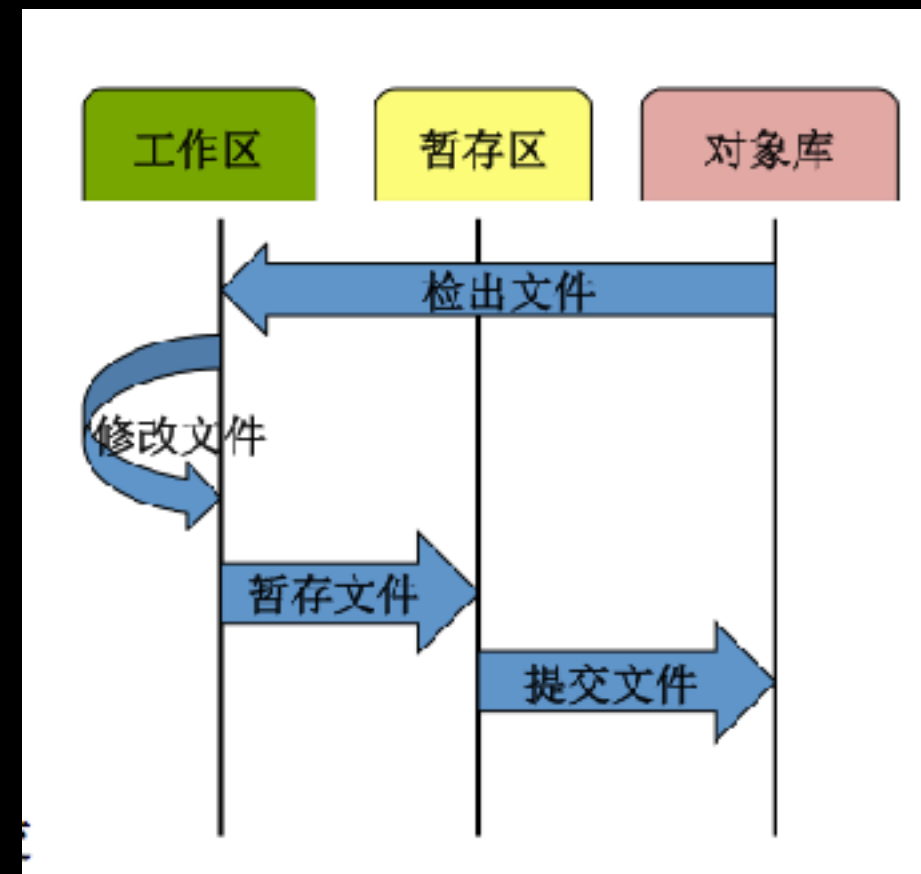


Git基础

- 直接记录快照，而非差异比较
- 近乎所有操作都在本地执行
- 时刻保持数据完整性
- 多数操作仅添加数据
- 文件的三种状态
 - 已修改 (modified)
 - 已暂存 (staged)
 - 已提交 (committed)

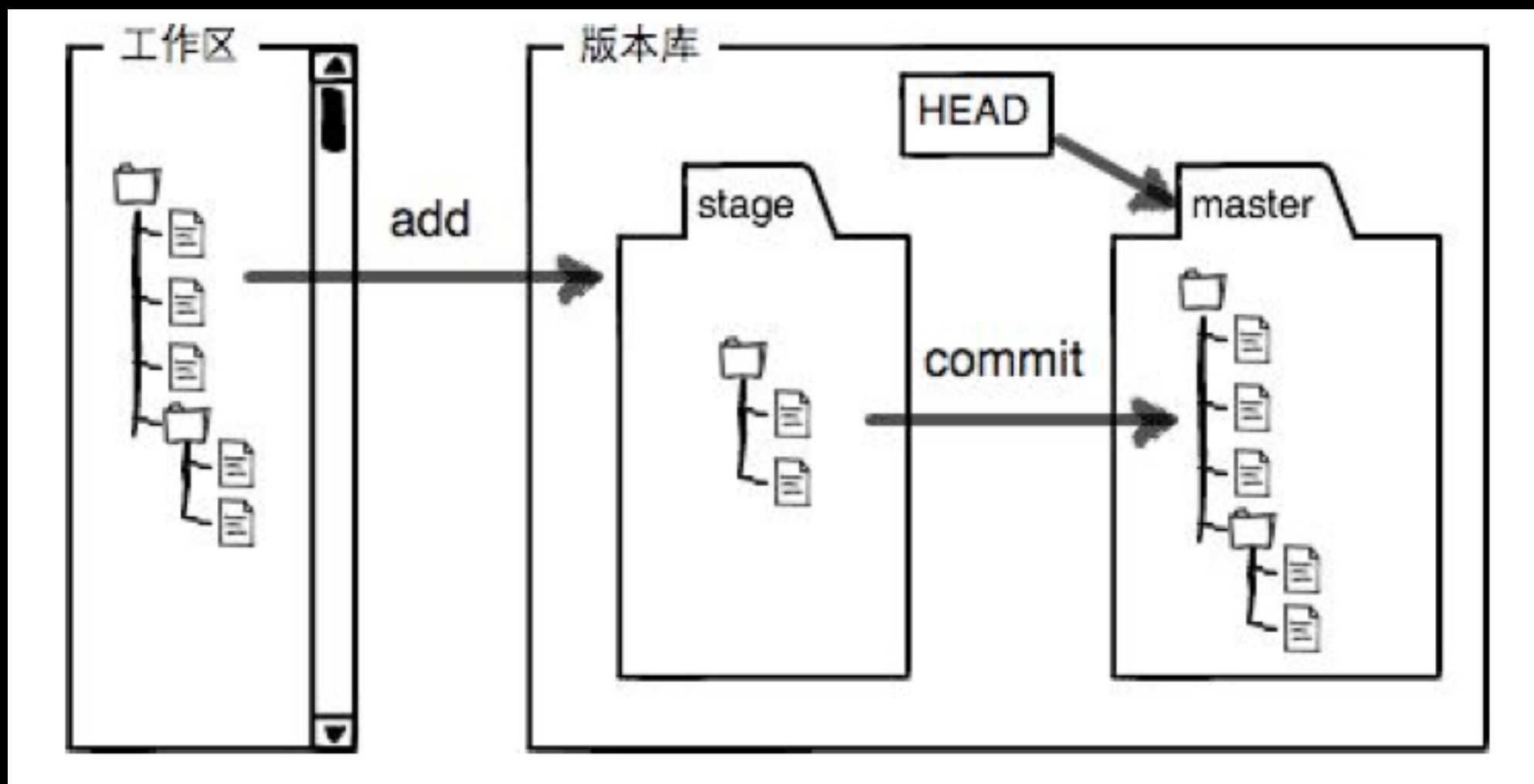
Git文件状态

- Git文件
 - 已被版本库管理的文件
- 已修改
 - 在工作目录修改Git文件
- 已暂存

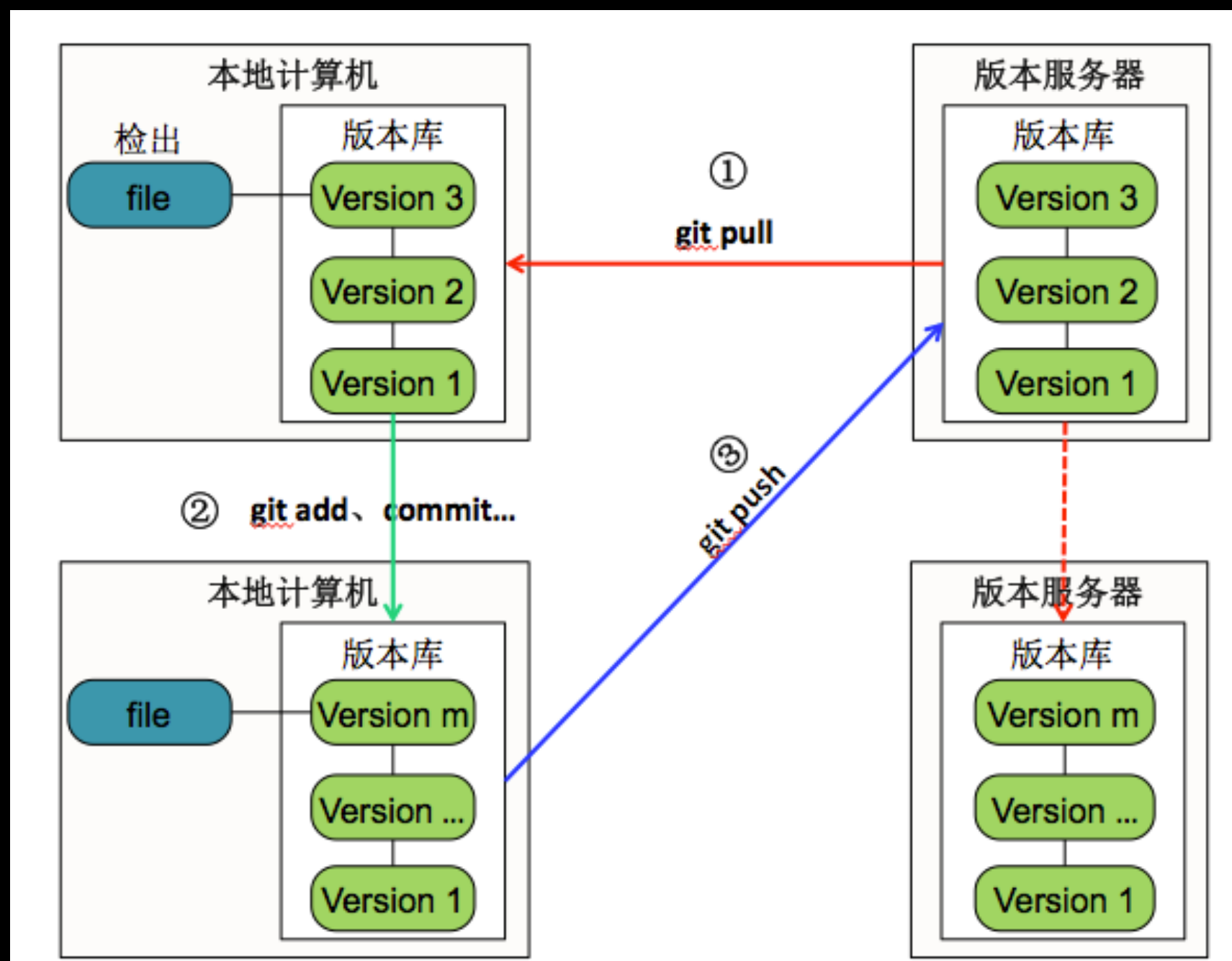


- 对已修改的文件执行Git暂存操作，将文件存入暂存区
- 已提交
 - 将已暂存的文件执行Git提交操作，将文件存入版本库

Git文件状态



本地版本库与服务器版本库



Git安装

- Linux (Ubuntu)
 - `sudo apt-get install git`
- Mac
 - 安装命令行工具（如已安装Xcode，命令行工具会在首次启动Xcode时提示安装）
 - homebrew
 - macports
- Windows
 - 通过msysGit (<http://code.google.com/p/msysgit>)

Git安装

- Windows
 - 完成安装之后，就可以使用命令行的 git 工具（已经自带了ssh 客户端）了，另外还有一个图形界面的 Git 项目管理工具
- 建议使用Git命令行，方便又快捷，GUI反而繁琐
- 如果需要使用GUI，推荐使用SourceTree，拥有Mac与Windows版本；此外，Windows下还可以使用TortoiseGit

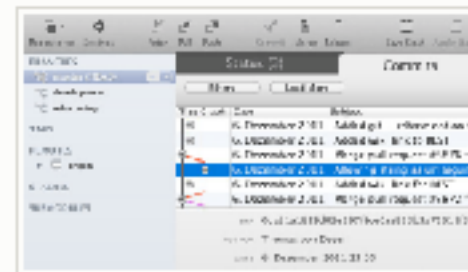
Git GUI



GitHub for Mac

Platforms: Mac

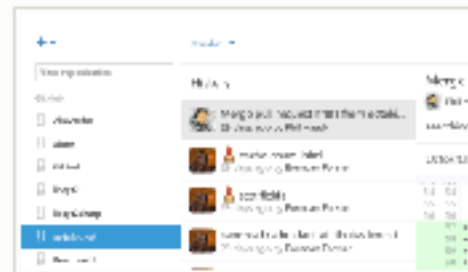
Price: Free



Tower

Platforms: Mac

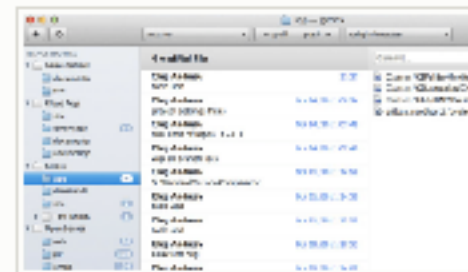
Price: \$59/user (Free 30 day trial)



GitHub for Windows

Platforms: Windows

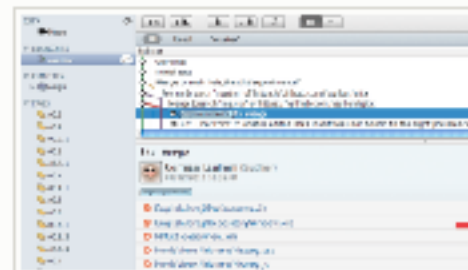
Price: Free



Gitbox

Platforms: Mac

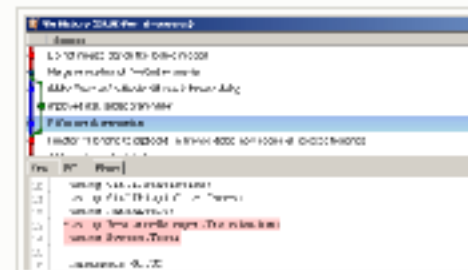
Price: \$14.99



GitX-dev

Platforms: Mac

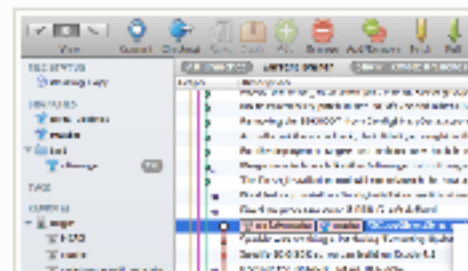
Price: Free



Git Extensions

Platforms: Windows

Price: Free



SourceTree

Platforms: Mac, Windows

Price: Free



git-cola

Platforms: Windows, Mac, Linux

Price: Free

Git常用命令

- 获得版本库

- git init
- git clone

- 版本管理

- git add
- git commit
- git rm

- 查看信息

- git help
- git log
- git diff

- 远程协作

- git pull
- git push

Git配置

- 查看版本
 - `git - -version`
- 首次使用前的配置
 - `git config - -global user.name 'zhangsan'`
 - `git config - - global user.email 'xxx@xxx.com'`

Git配置

- 查看配置信息
 - `git config --list`
 - `git config --user.name`
 - `git config --user.email`

Git配置

- 获取帮助
 - `git help config`
 - `git config - - help`
 - `man git-config`

Git初始化新仓库

- `mkdir git_training && cd git_training`
- `git init` # 初始化git仓库
- 从现有仓库克隆，克隆完整数据，包括版本信息
 - `git clone git://github.com/zhanglong/zi.git`
 - `git clone git://github.com/zhanglong/zi.git helloworld`
- 检查当前文件状态
 - `git status`

Git添加文件与提交

- touch readme
- # 将文件添加到暂存区，每次修改之后都需要将文件放到暂存区中
- git add readme
- # 将暂存区中的文件提交到Git版本库中
- git commit -m 'initial import'

忽略文件

- `.gitignore`
 - `*.a` # 忽略所有 `.a` 结尾的文件
 - `!lib.a` # 但 `lib.a` 除外
 - `/TODO` # 仅仅忽略项目根目录下的 `TODO` 文件, 不包括 `subdir/TODO`
 - `build/` # 忽略 `build/` 目录下的所有文件
 - `doc/*.txt` # 会忽略 `doc/notes.txt` 但不包括 `doc/server/arch.txt`

Git的比较

- #查看尚未暂存的文件更新了哪些部分
 - `git diff`
- #查看暂存区文件和上次提交的快照之间的差异
 - `git diff --cached`
- #查看工作区与版本库最新版本之间的差异
 - `git diff HEAD -- file_name`

Git的提交更新

- # 提交更新，每次准备提交前，先用 `git status` 看下，是不是都已暂存起来了，然后再运行提交命令（好习惯）
- `git commit`
- `git commit -m 'initial project version'`

Git删除文件

- 删除文件
- `git rm readme`

Git重命名文件

- `git mv from_file to_file`
- IDE中的文件名重构使用的就是该命令

Git查看提交历史

- # 查看提交历史
- git log
- -p 展开显示每次提交的内容差异
- -n 仅显示最近的n次更新
- - -stat 仅显示简要的增改行数统计
- - -pretty=oneline
- - -pretty=format:"%h - %an, %ar : %s"

Git修改最后一次提交

- # 修改最后一次提交（后悔药）
- `git commit -m 'initial import'`
- `git commit --amend -m 'correct comments'`
- # 上面2条命令最终只会产生一条提交

Git取消已暂存文件

- # 取消已暂存文件
 - `git reset HEAD file_name`
- # 取消对文件的修改
 - `git checkout -- file_name`

Git标签

- 显示已有的标签
 - `git tag`
- 列出符合条件的标签
 - `git tag -l 'v1.0.1'`

Git标签

- 新建标签，标签有两种：轻量级标签（lightweight）与带有附注标签（annotated）
- 创建一个轻量级标签
 - `git tag v1.0.1`
- 创建一个带有附注的标签
 - `git tag -a v1.0.2 -m 'release version'`
- 删除标签
 - `git tag -d tag_name`

Git标签

- 分享标签
- 默认情况下，git push并不会将标签推送到远程服务器
- `git push origin v1.0.1`
- 一次推送所有本地新增的标签
- `git push origin - - tags`

Git标签

- 查看标签详细信息
 - `git show tag_name`
- 给某个commit id 打标签
 - `git tag tag_name commit_id`

Git回退版本

- 回退到上一版本
 - `git reset --hard HEAD^`
 - `git reset --hard HEAD~1`
 - `git reset --hard commit_id`
- 返回到某一个版本
 - `git reflog`

暂存区与对象库

- 暂存区
 - 一个文件: `.git/index`
- 对象库
 - 一个目录: `.git/objects`
 - 用于存放版本库的各种对象
- 暂存区仅保留对象库当前分支的快照, `git commit`将会替换当前快照, 使得暂存区与对象库的当前分支最新更新一致

对象

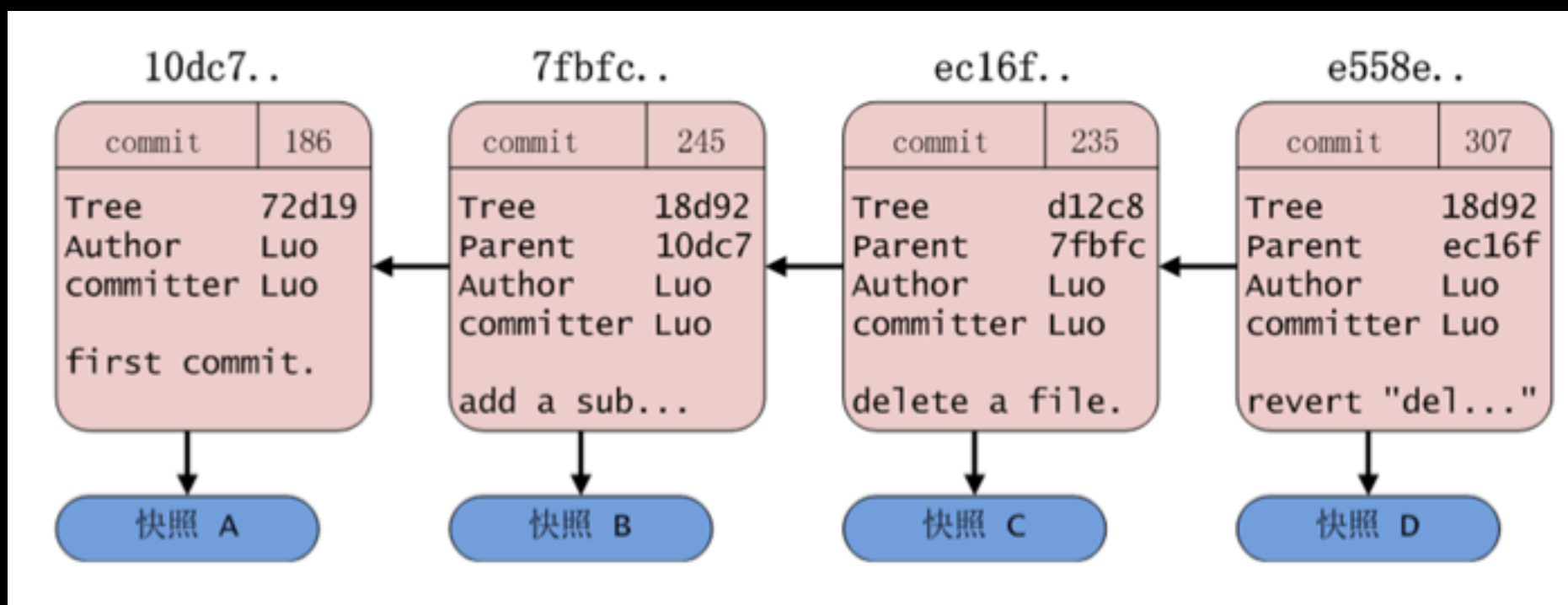
- 对象
 - 每个对象都是一个文件
 - 每个对象都用40位的SHA-1值标识：
6ff87c4664981e4397625791c8ea3bbb5f2279a3
- 对象组成
 - 类型：blob、tree、commit和tag
 - 内容：对象文件的内容
 - 大小：内容的大小

对象类型

- blob
 - 是一个文件
 - 存储文件的内容
- tree
 - 是一个文件
 - 类似一个目录
 - 包含其它tree和blob
- commit
 - 是一个文件
 - 包含时间、作者、一个tree的标识、父commit的标识
- tag
 - 是一个文件
 - 包含一个commit的标识

分支

- 一个commit对象链：一条工作记录线



master

- 主分支
 - 默认分支
 - 稳定分支，项目发布与部署分支
- 综合信息
 - 一个文件： `.git/refs/heads/master`
 - 文件内容： commit对象的SHA-1值

HEAD

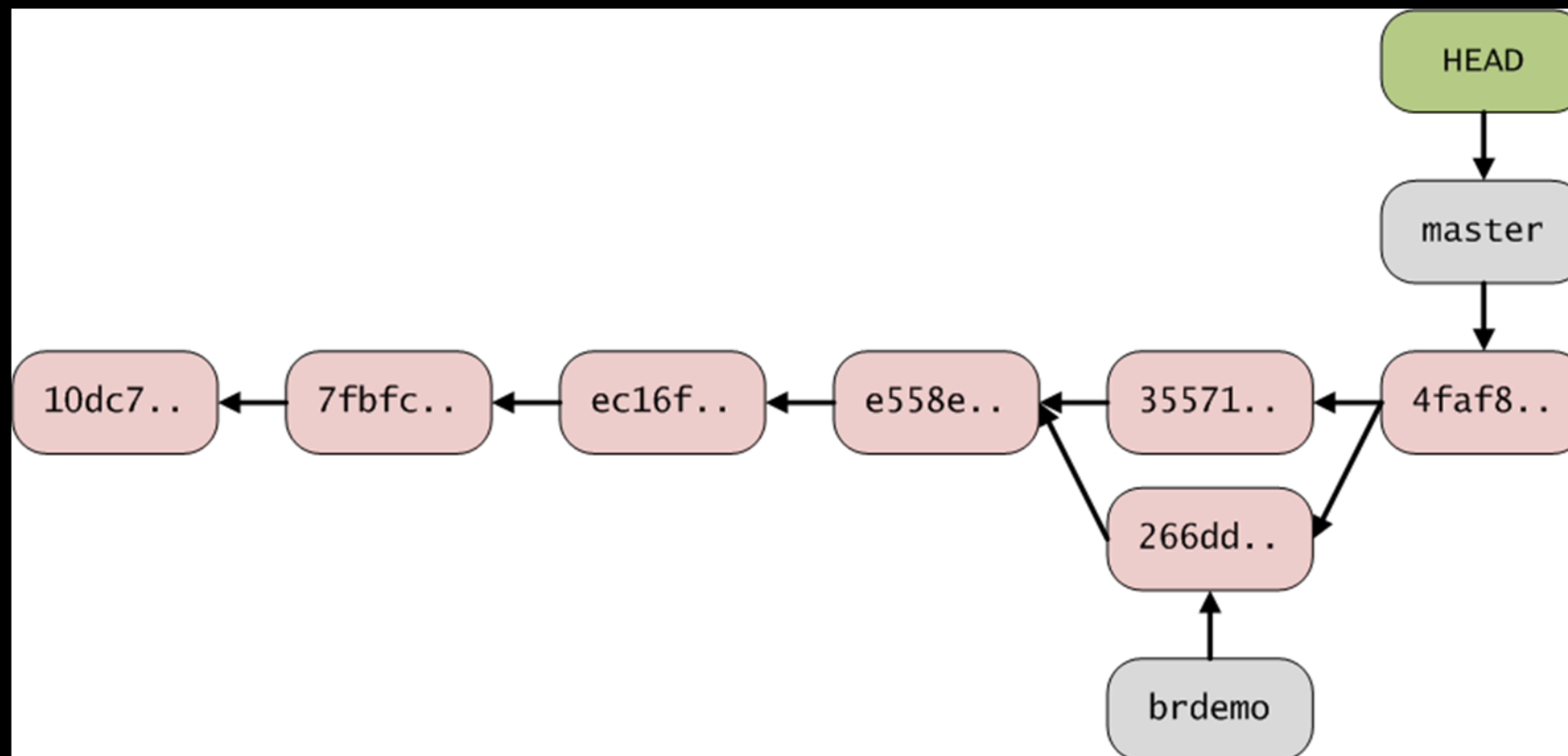
- 当前commit对象引用
 - 一个文件: `.git/HEAD`
 - 文件内容:
 - 分支引用 (`ref: refs/heads/master`)
 - commit对象的SHA-1值 (`4c9f4...`)

分支管理

- 列出当前所有分支
 - `git branch`
- 创建分支：
 - `git branch branch_name`
- 切换分支
 - `git checkout branch_name`
- 创建并转向所创建的分支
 - `git checkout -b branch_name`
- 查看分支最后一次提交的信息
 - `git branch -v`
- 删除分支
 - `git branch -d branch_name`

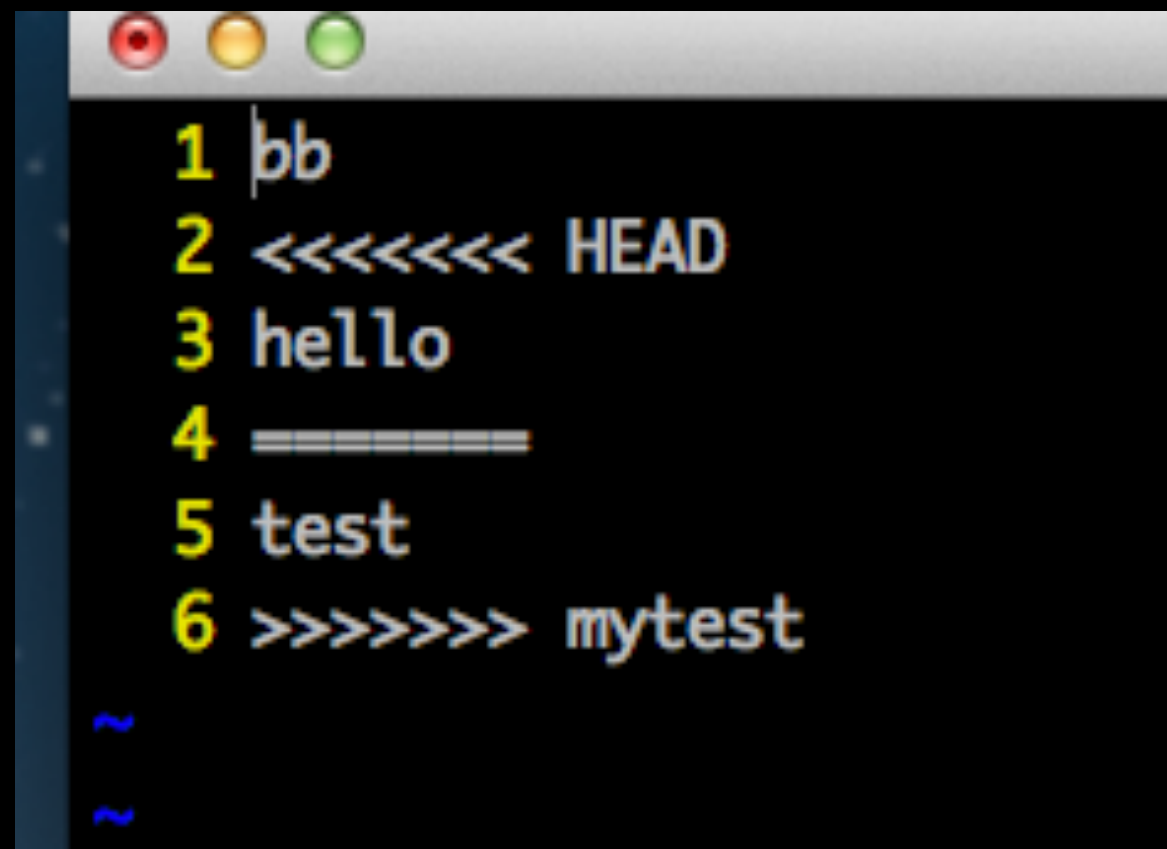
分支合并

- `git merge branch_name`
- Git的自动合并能力非常出色，但有时也需要开发者自行解决冲突



分支合并冲突

- 两个分支修改了同一文件的同一部分内容
- 需要先解决冲突，然后再提交



```
1 bb
2 <<<<<<< HEAD
3 hello
4 =====
5 test
6 >>>>>>> mytest
~
~
```

远程版本库

- 管理

- `git clone <url>`: 生成一个叫origin的远程版本库
- `git remote -v`: 查看远程版本库信息
- `git remote add <remote repo name> <url>`: 指定版本库名字

- 路径

- `.git/refs/remotes/`

远程分支

- 路径
 - `.git/refs/remotes/<remote repo name>/`
- 访问方式
 - `<remote repo name>/<branch name>: origin/master`
- 跟踪分支 (tracking branch)
 - `git checkout <remote repo name>/<remote branch name>`
 - `git checkout -b <branch name> <remote repo name>/<remote branch name>`

从远程版本库抓取数据

- `git fetch [remote repo name]`
 - `git fetch = git fetch origin`
- 从远程仓库中拉取本地仓库中还没有的数据（所有分支）
- 需要执行`git checkout`跟踪远程分支，从而产生本地分支

克隆版本库

- `git clone <url>`
- `git init <repo name>`
- `git remote add origin <url>`
- `git fetch origin`
- `git checkout origin/master`

推送数据到远程版本库

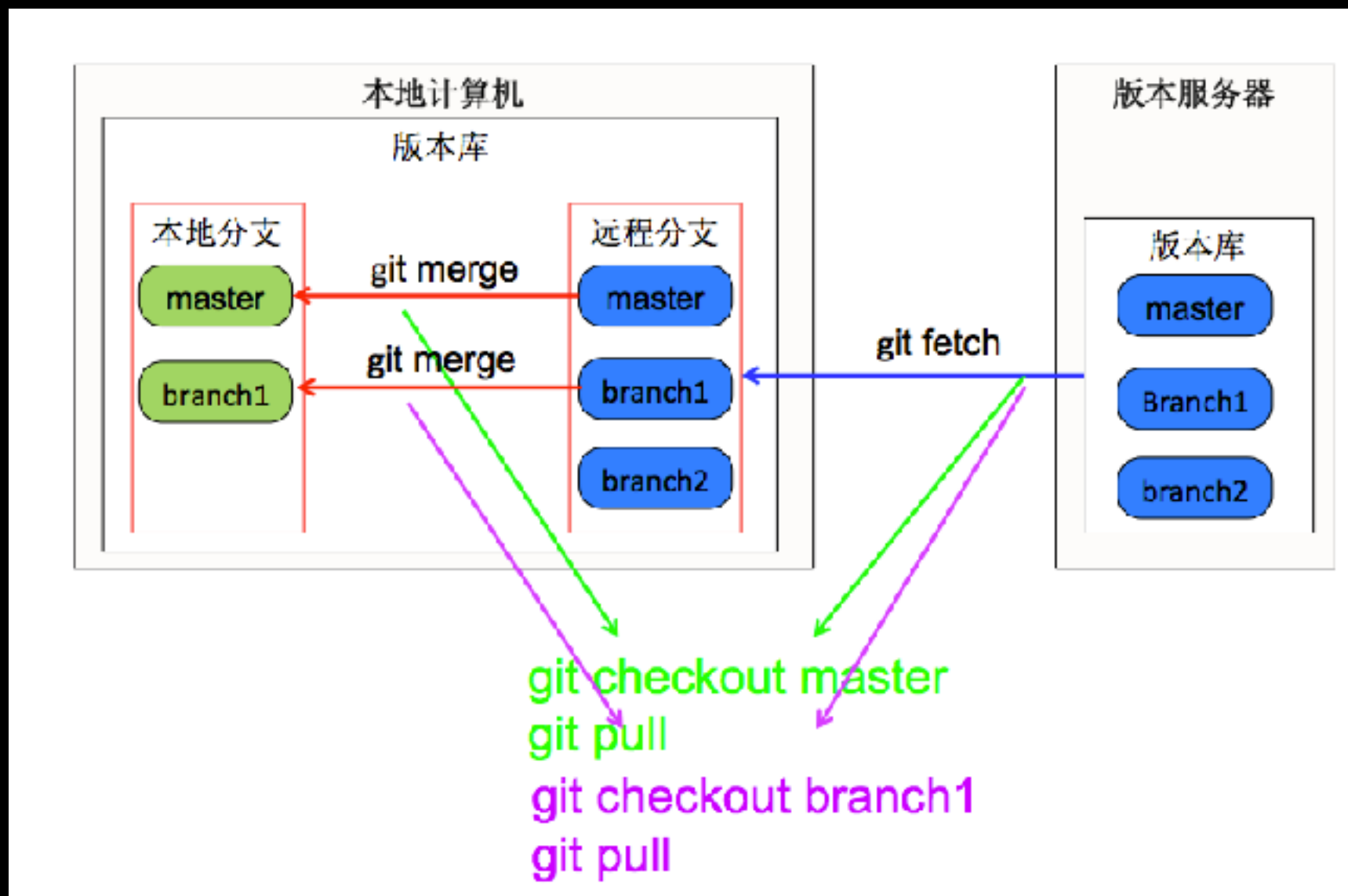
- `git push [remote repo name] [branch name]`
 - 将本地分支<branch name>推送给<remote repo name>的<branch name>分支
- `git push`
 - 如果当前分支为跟踪分支，则推送至其跟踪的远程分支
 - 否则=`git push origin`，将当前分支推送给origin，在origin上的分支名称与当前分支名称相同

推送数据到远程版本库

- 推送本地分支
 - `git push origin server_branch`
- 推送本地分支local_branch为origin的remote_branch
 - `git push origin local_branch:remote_branch`

从远程仓库拉取

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

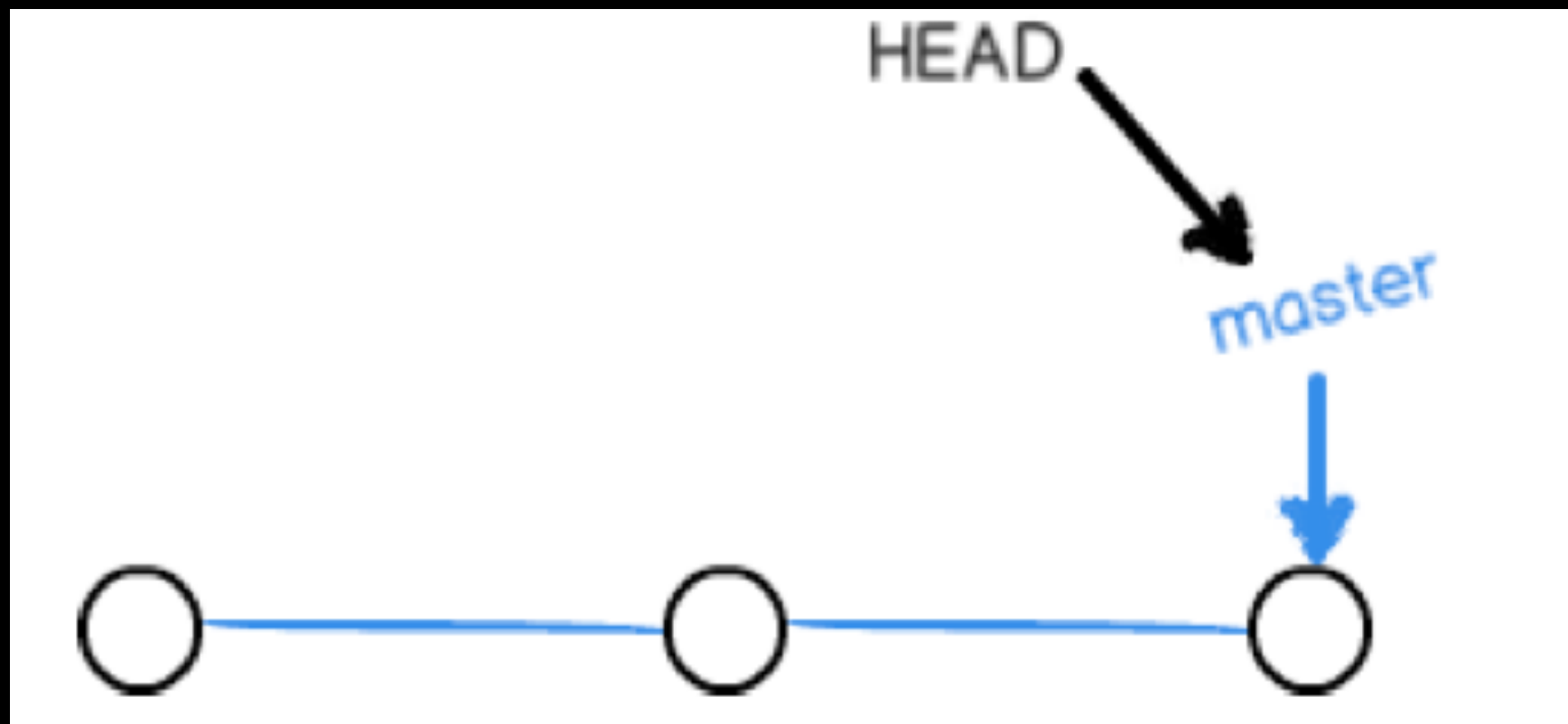


删除分支

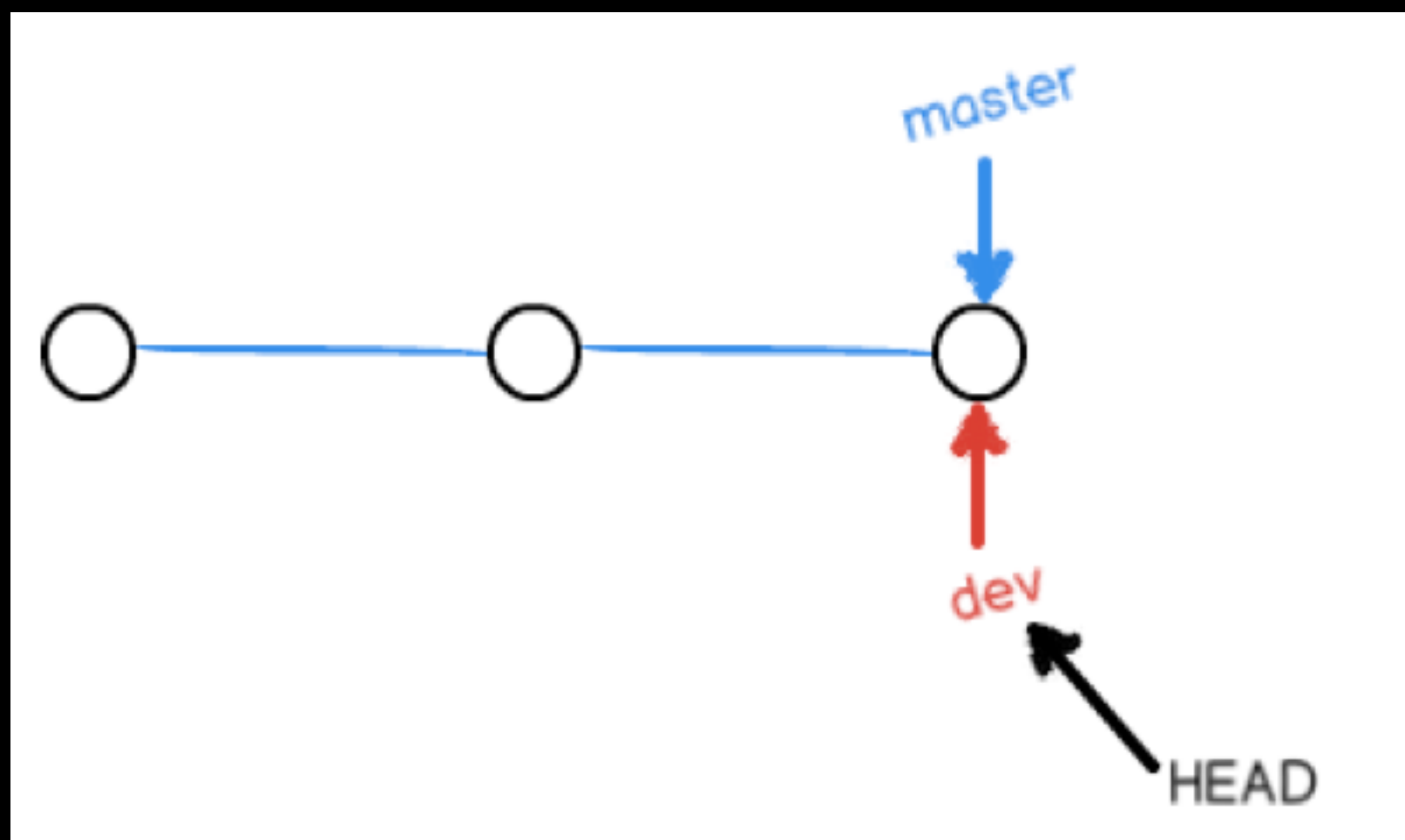
- 删除本地分支
 - `git branch -d branch_name`
- 删除远程分支
 - `git push origin :branch_name`

再谈分支

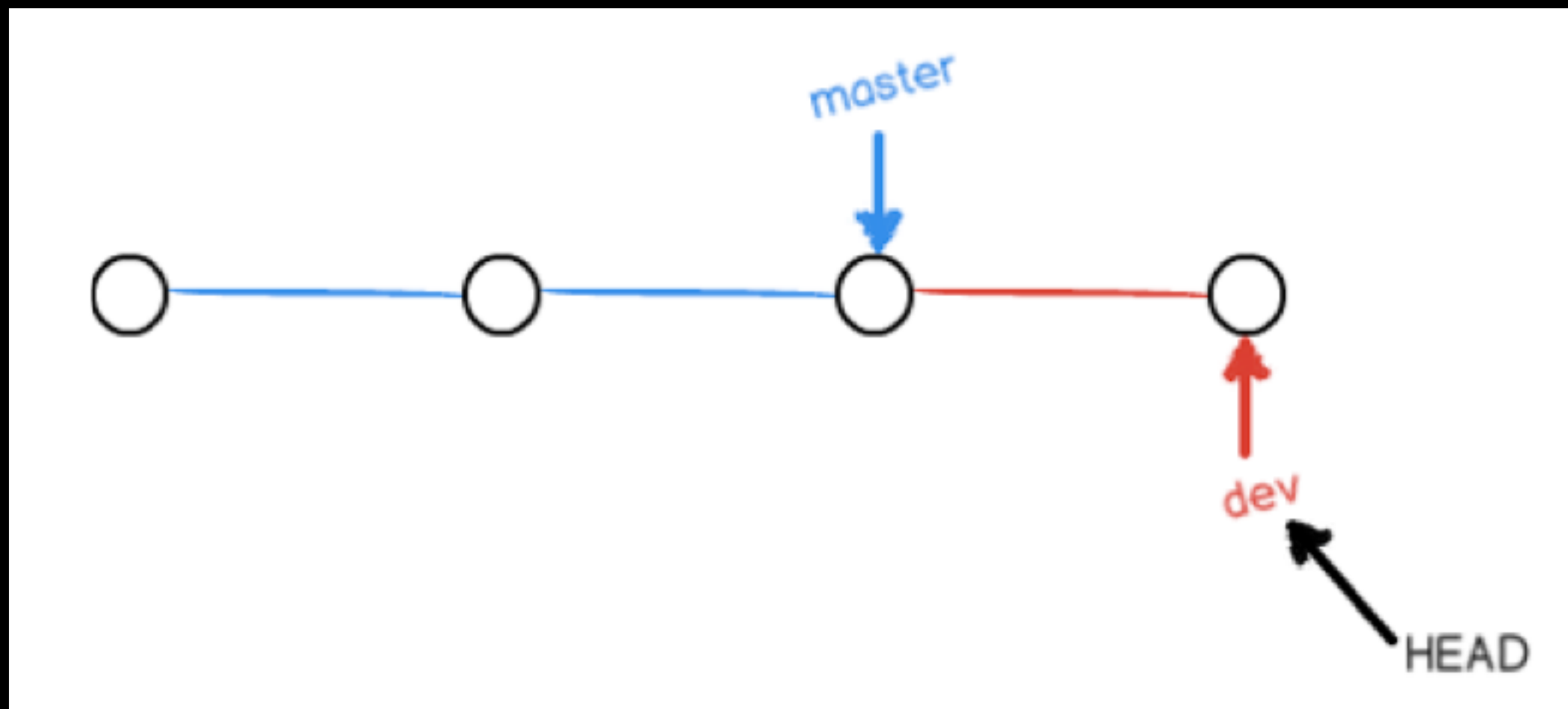
- HEAD指向的是当前分支
- master指向提交



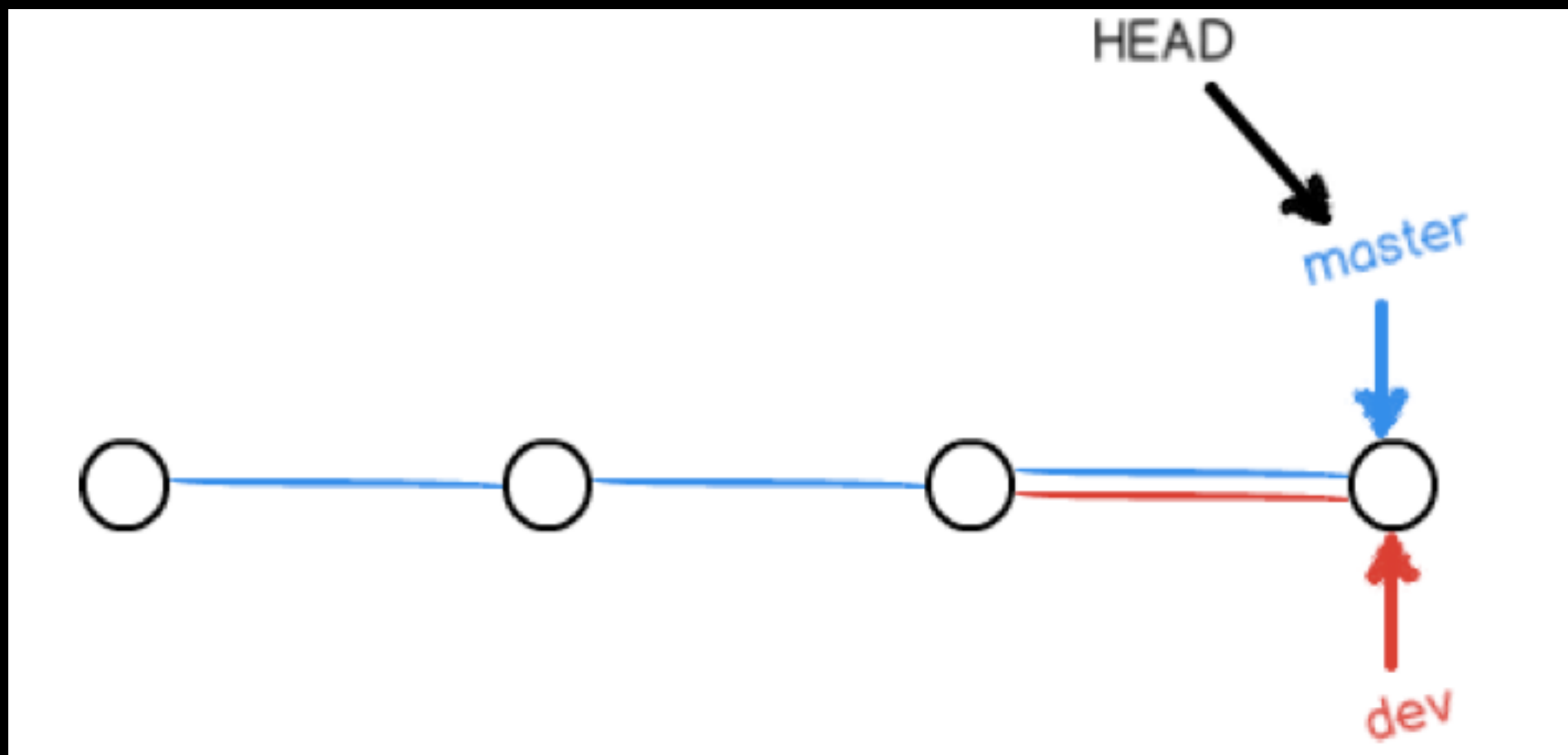
再谈分支



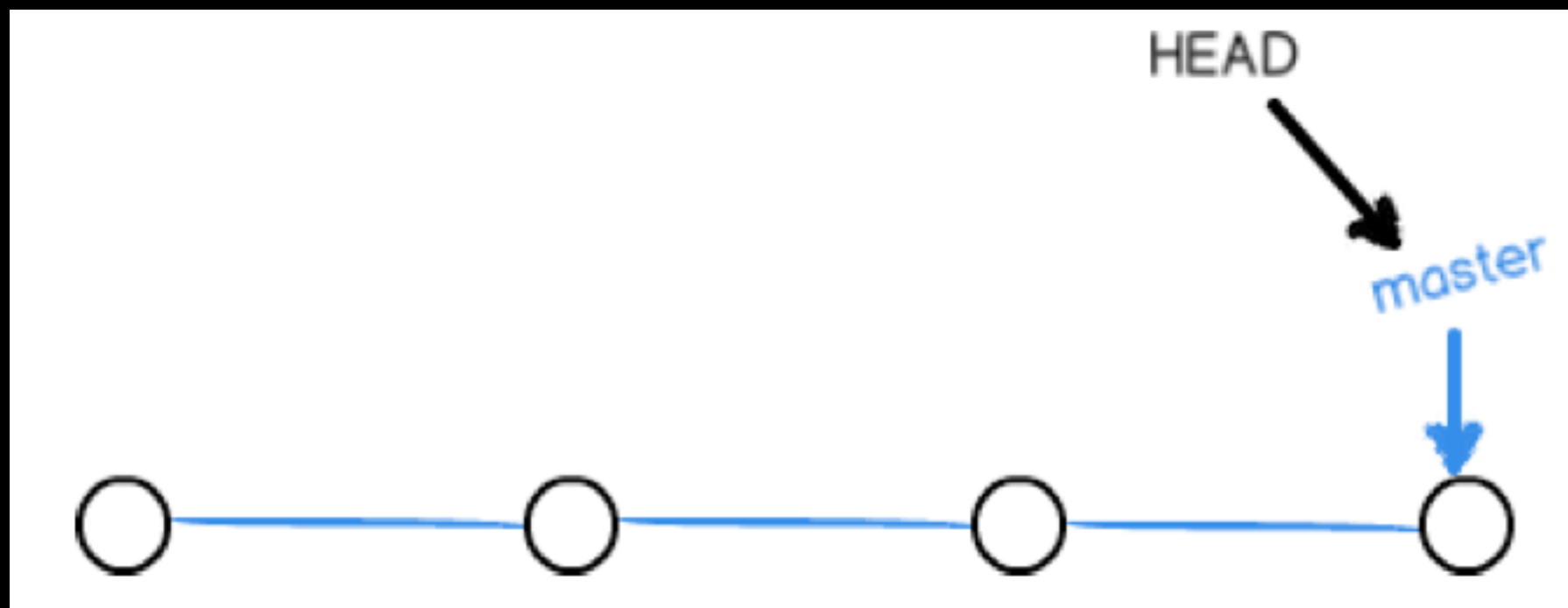
再谈分支



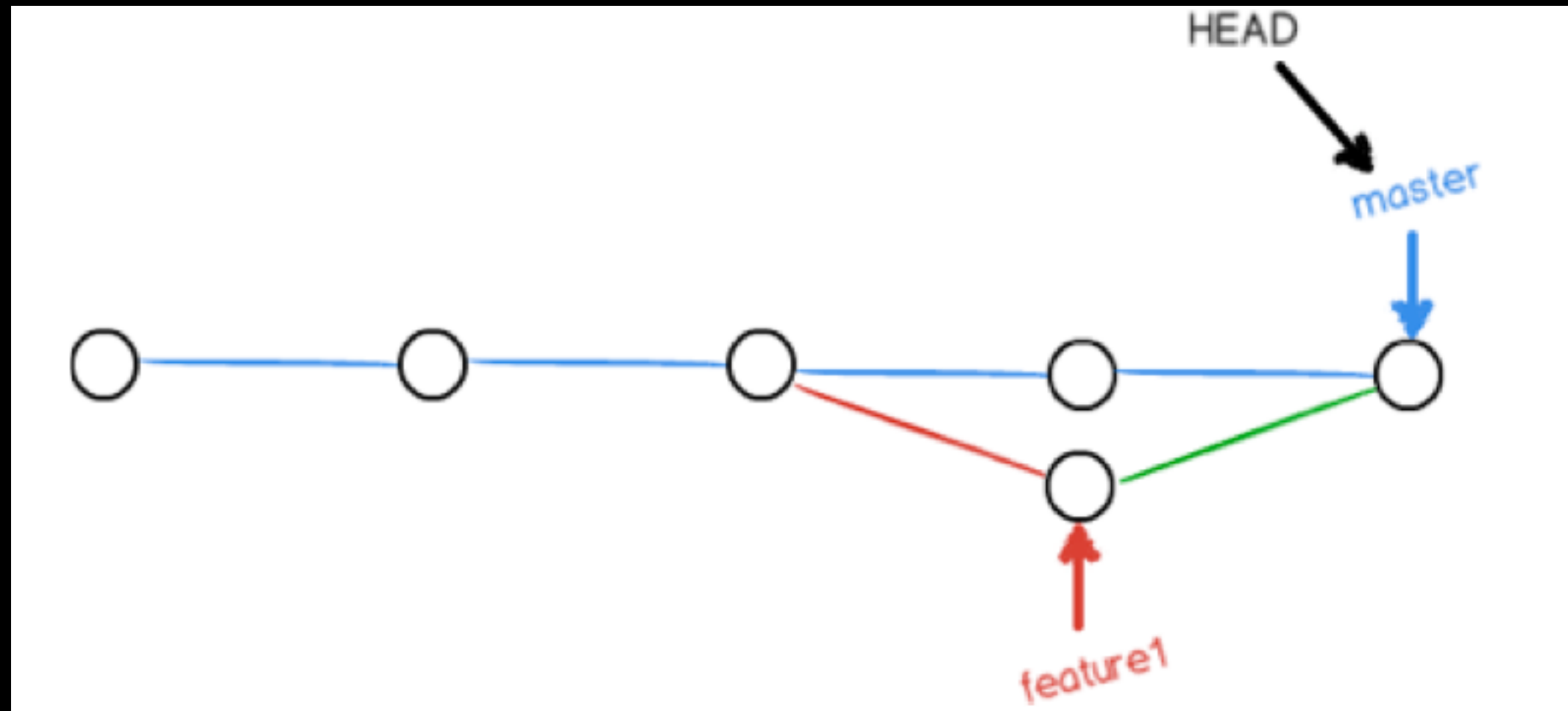
再谈分支



再谈分支



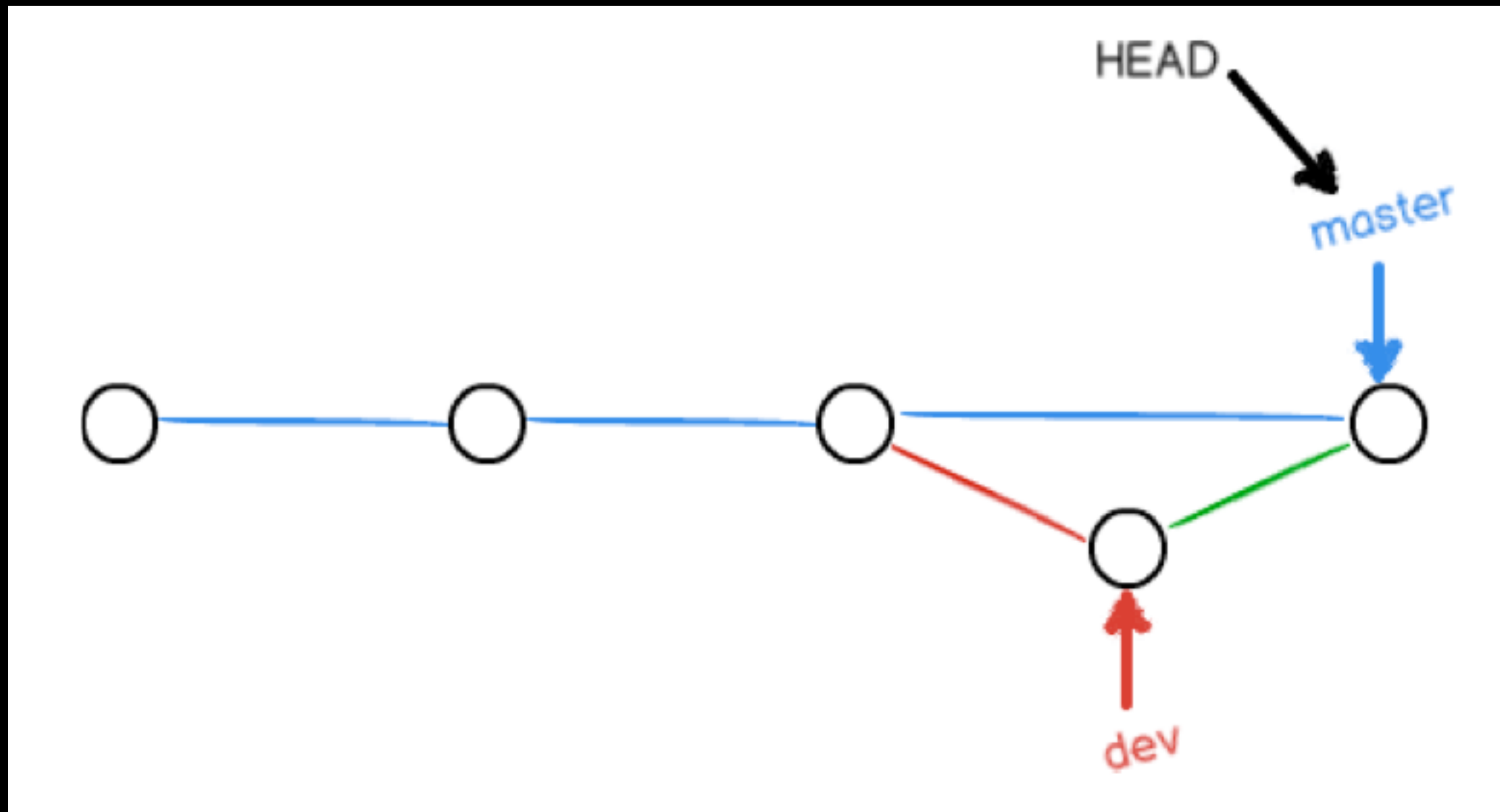
分支合并冲突



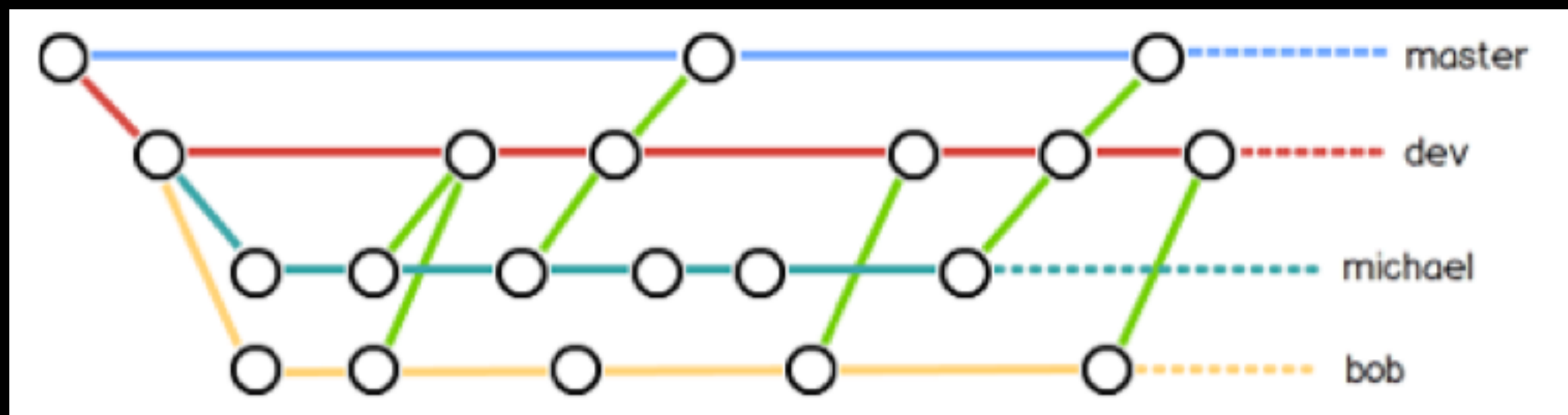
fast-forward

- 如果可能，合并分支时Git会使用fast-forward模式
- 在这种模式下，删除分支时会丢掉分支信息
- 合并时加上 `--no-ff` 参数会禁用fast-forward，这样会多出一个commit id
 - `git merge --no-ff dev`
- 查看log
 - `git log --graph`

不使用fast-forward



分支管理最佳实践



Bug修复

- 修复Bug时，我们需要创建一个新的分支来处理，处理完毕后再将其合并到master上，最后删除该bug分支
 - `git checkout -b issue-001`
 - `git merge - -no-ff issue-001`

保存工作现场

- 保存现场
 - `git stash`
 - `git stash list`
- 恢复现场
 - `git stash apply` (stash内容并不删除, 需要通过`git stash drop stash@{0}`手动删除)
 - `git stash pop` (恢复的同时也将stash内容删除)
 - `git stash apply stash@{0}`

建立分支关联

- `git checkout -b branch_name origin/
branch_name`
- `git branch - -set-upstream branch_name origin/
branch_name`

再谈标签

- 发布版本时通常需要打一个标签
- 标签也是版本库的一个快照
- 标签实际上就是指向某个commit的指针（类似于分支）
- 分支可以移动，标签无法移动（静态的）

删除标签

- 首先删除本地标签
 - `git tag -d tag_name`
- 接下来再删除远程标签
 - `git push origin :refs/tags/tag_name`

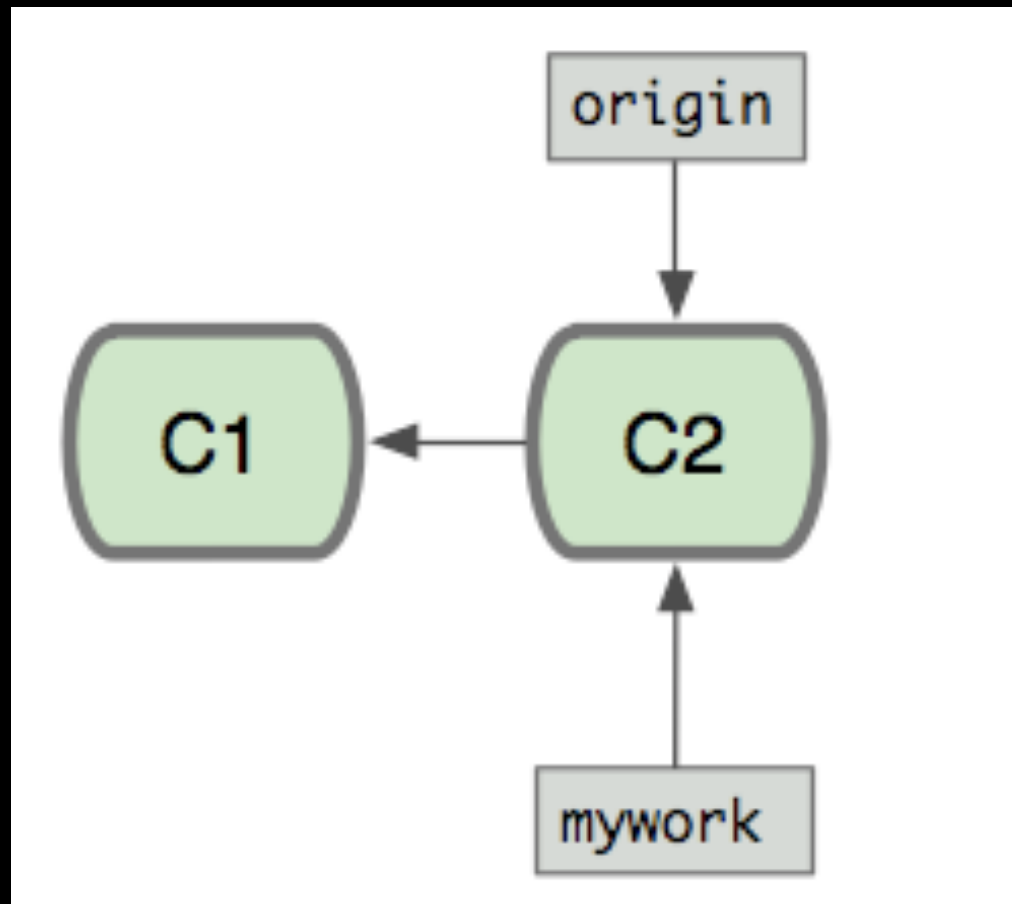
Git裸库

- 初始化Git仓库
 - `git init`
- 初始化Git裸库（主要用于服务器端）
 - `git init - -bare test.git`

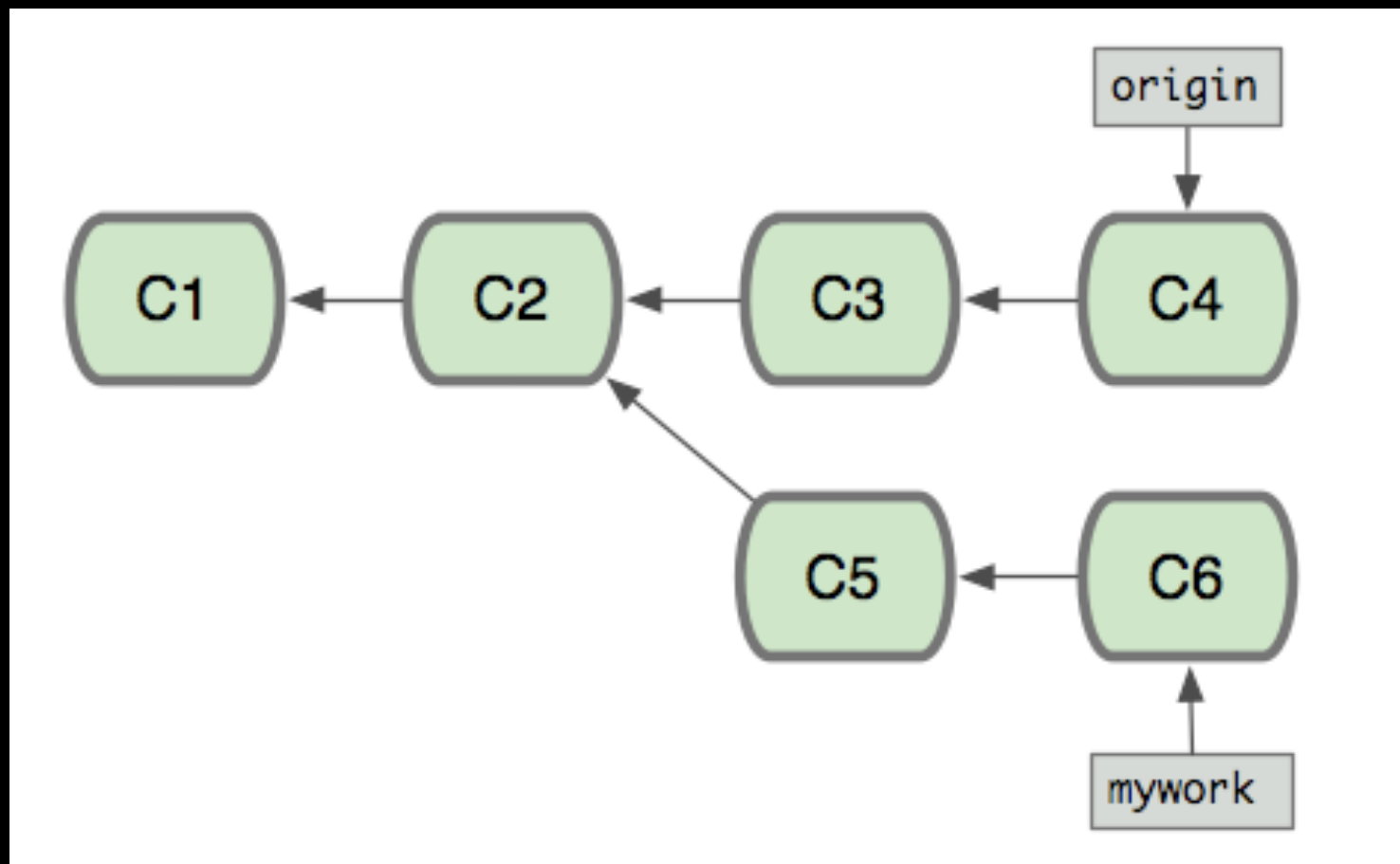
Git rebase

- rebase: 变基, 意即改变分支的根基
- 从某种程度上来说, rebase与merge可以完成类似的工作, 不过二者的工作方式有着显著的差异

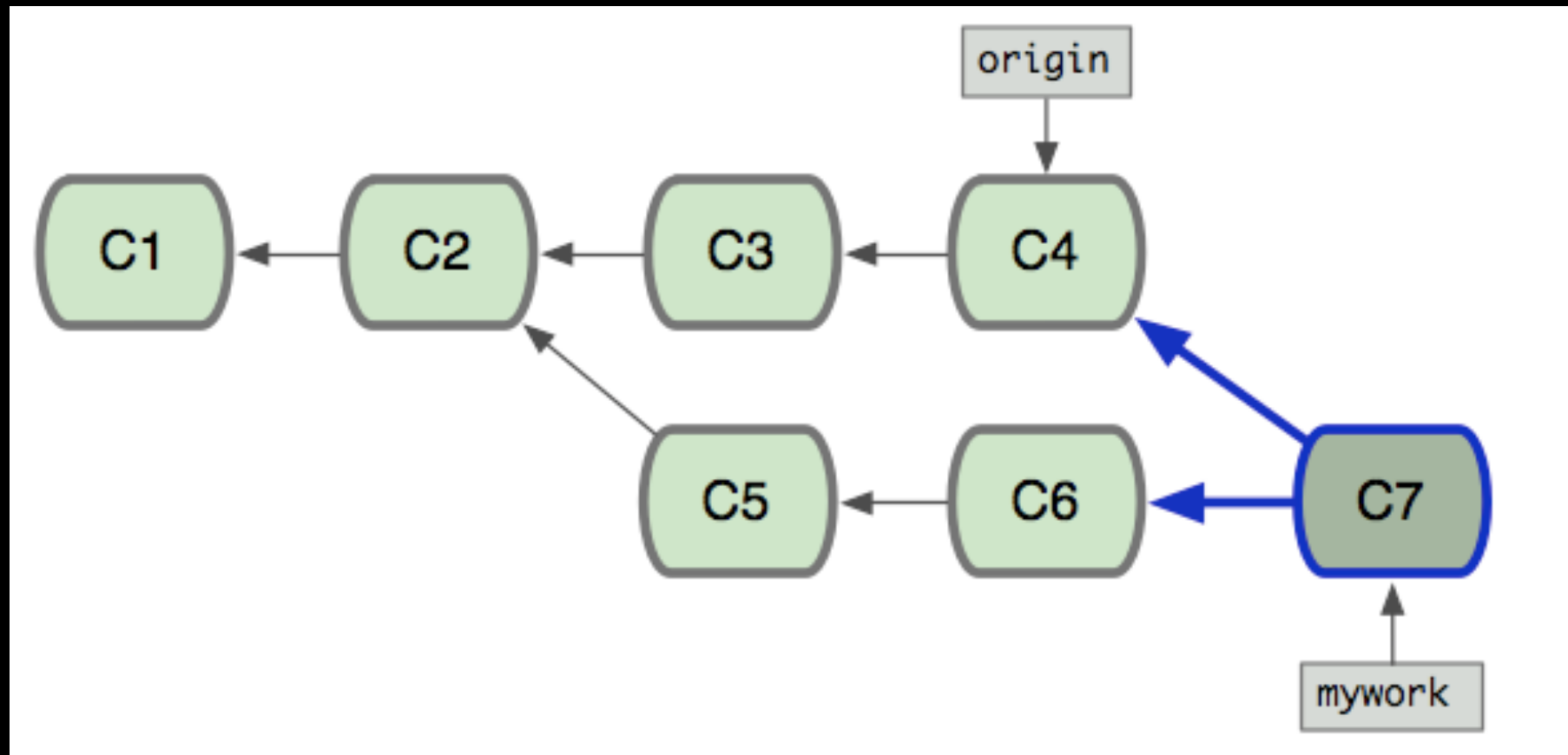
merge



merge

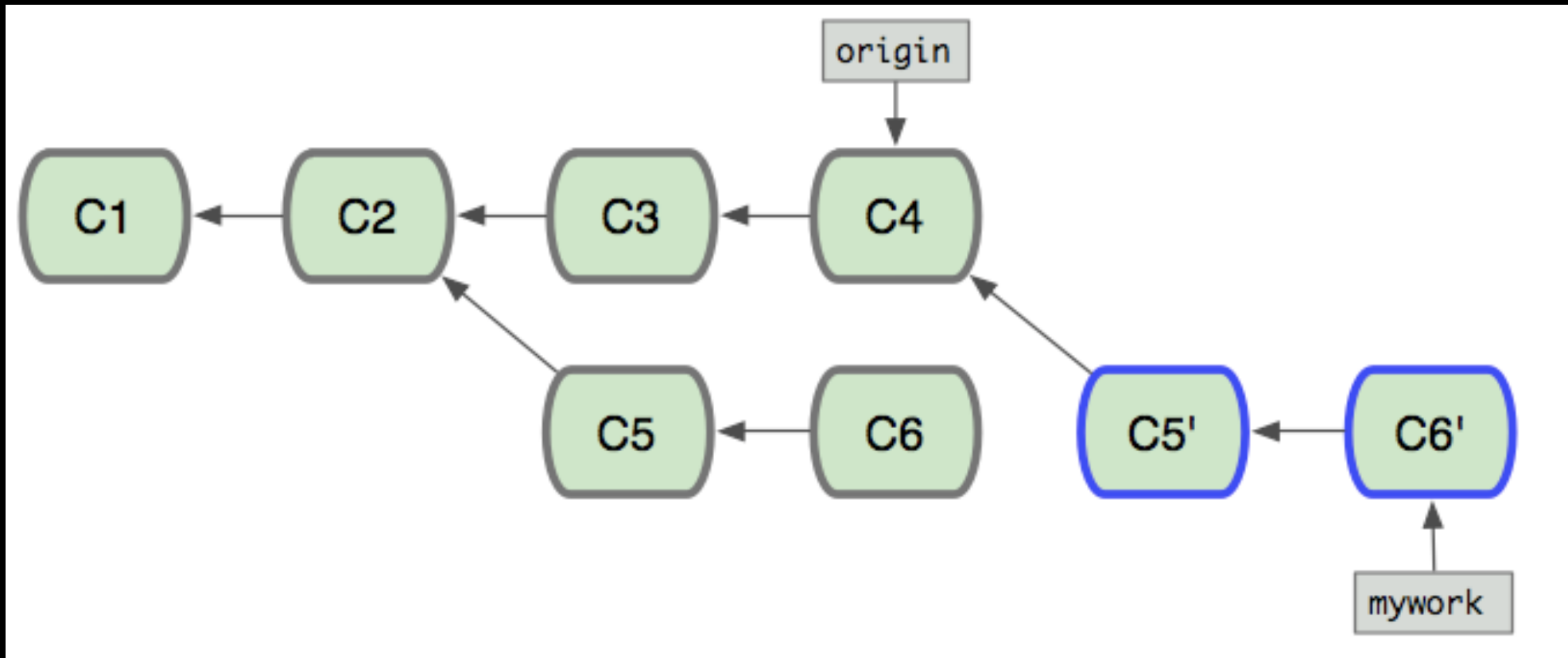


merge

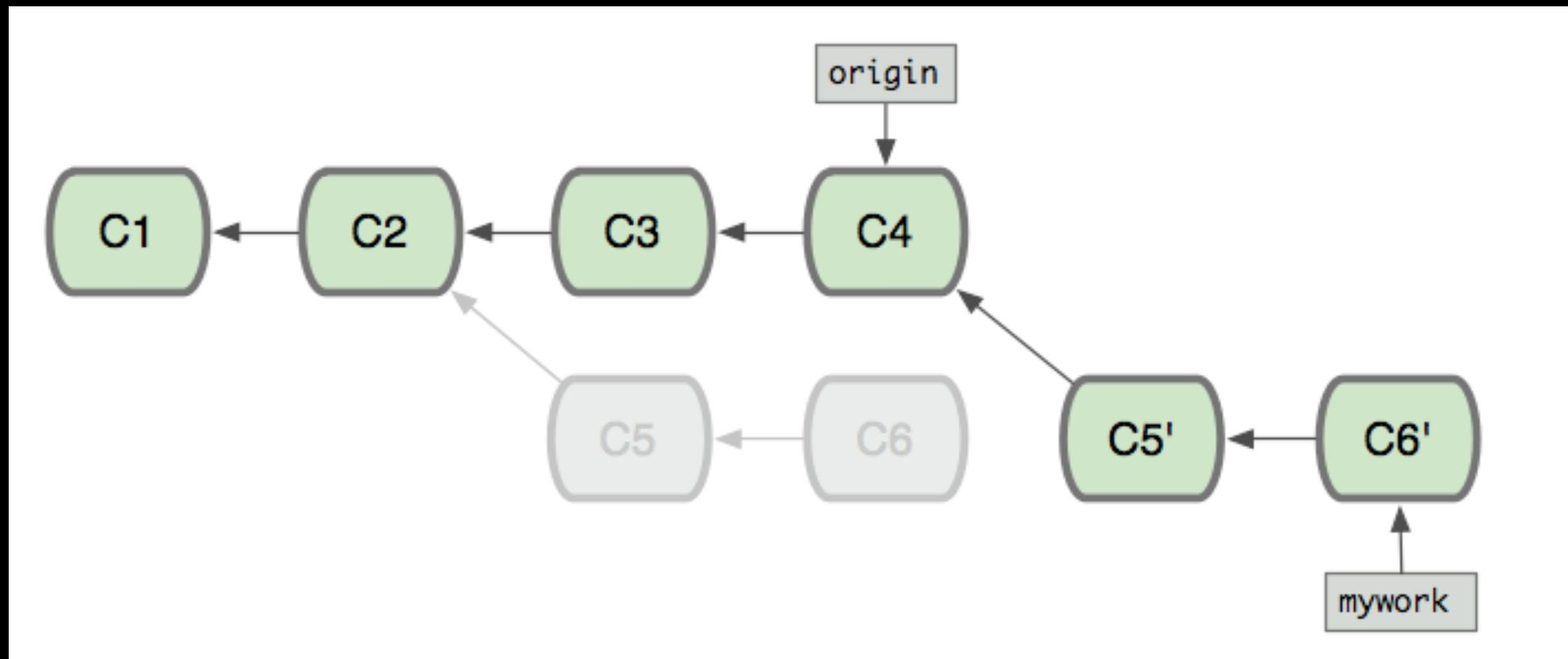


rebase

- git checkout mywork
- git rebase origin

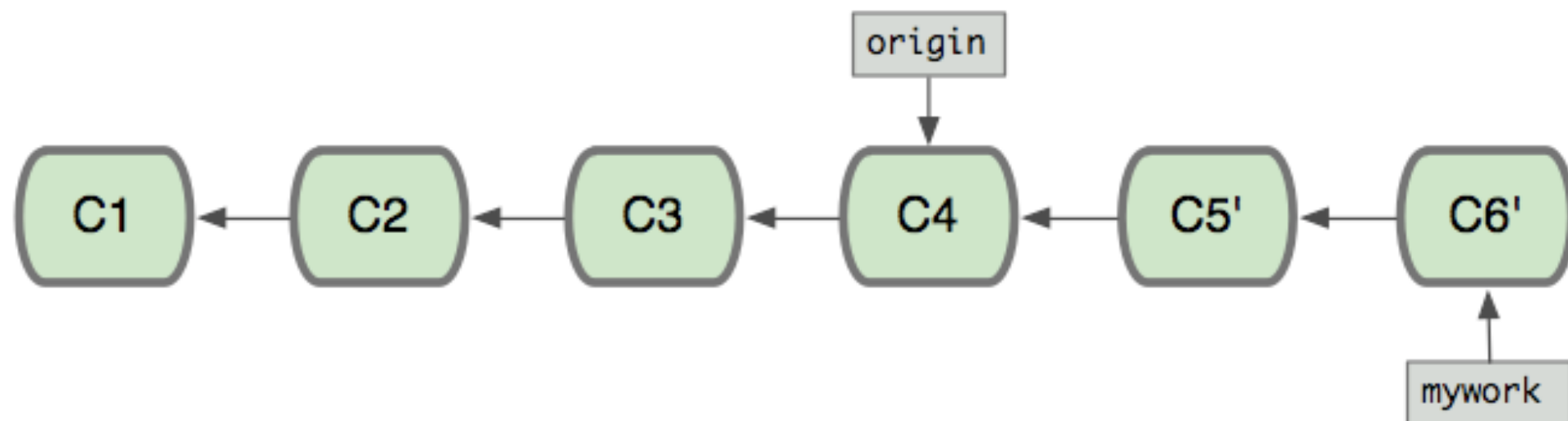


rebase



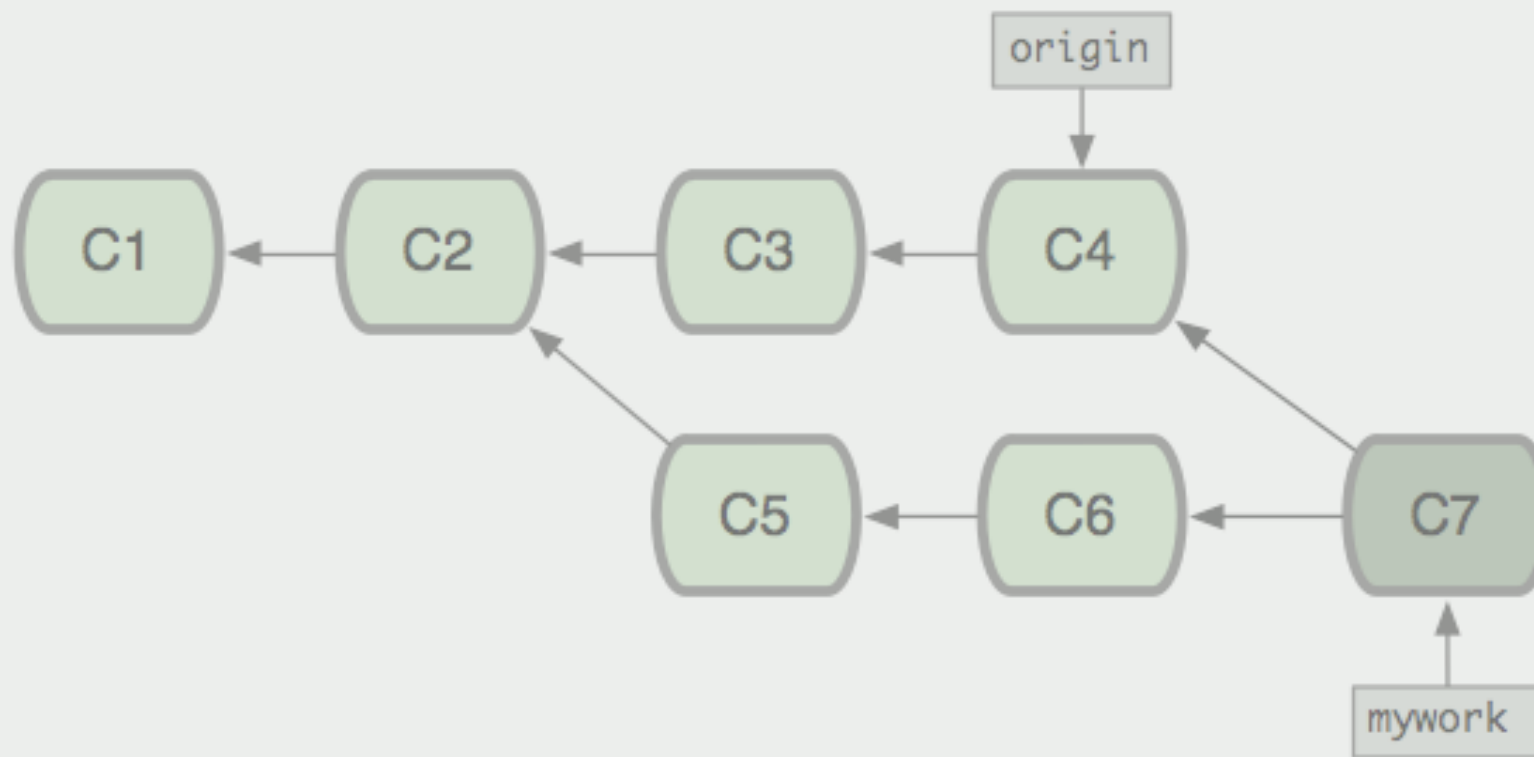
rebase与merge的历史区别

git rebase



rebase与merge的历史区别

git merge



rebase注意事项

- rebase过程中也会出现冲突
- 解决冲突后，使用git add添加，然后执行
 - `git rebase -- continue`
- 接下来Git会继续应用余下的补丁
- 任何时候都可以通过如下命令终止rebase，分支会恢复到rebase开始前的状态
 - `git rebase -- abort`

rebase最佳实践

- 不要对master分支执行rebase，否则会引起很多问题
- 一般来说，执行rebase的分支都是自己的本地分支，没有推送到远程版本库

谢谢

