

Git零基础实战

新浪微博 @宅学部落

作者: wit

视频教程: 51CTO学院或CSDN学院搜 “Git零基础实战”



课程计划

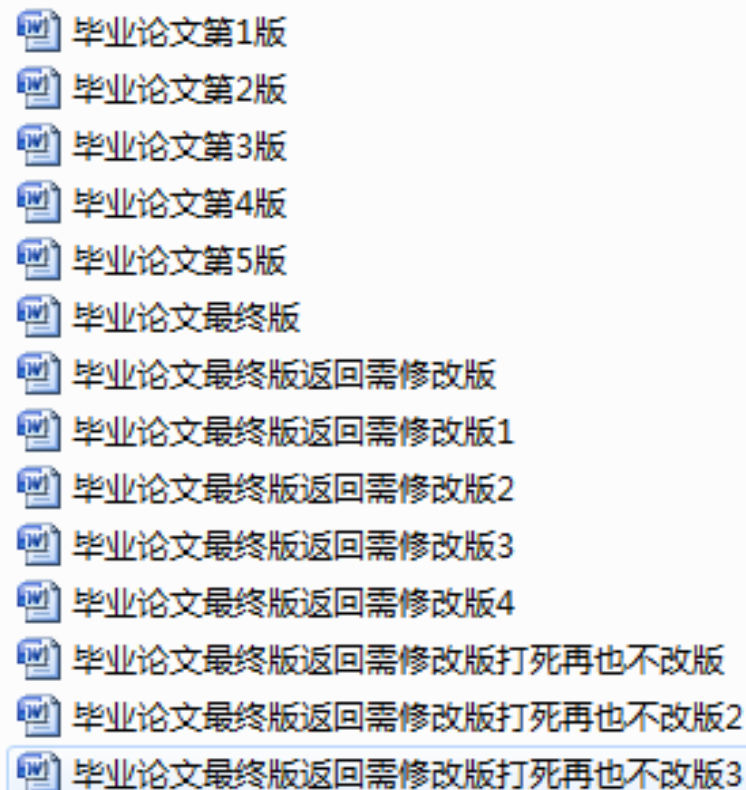
- Git简介
- Git安装
- 快速开始
- Git基本概念
- 翻仓倒海：文件管理
- 时光穿梭：改写历史
- 分支管理
- 标签管理
- 远程仓库
- Git实战

第一章：Git简介

- 版本的基本概念
- 版本控制系统发展历程
- Git特点
- 为什么要学习Git
- 本教程简介

版本的基本概念

- 版本控制系统VCS
 - 银行柜台的“会计”
 - 跟踪、记录文件变化
- 版本控制的重要性
 - 提高工作协作效率
 - 甚至决定项目的成败
- 版本库
 - VCS的核心组成部分
 - 用来存储历史数据的地方
- 分类
 - 集中式版本控制系统
 - 分布式版本控制系统

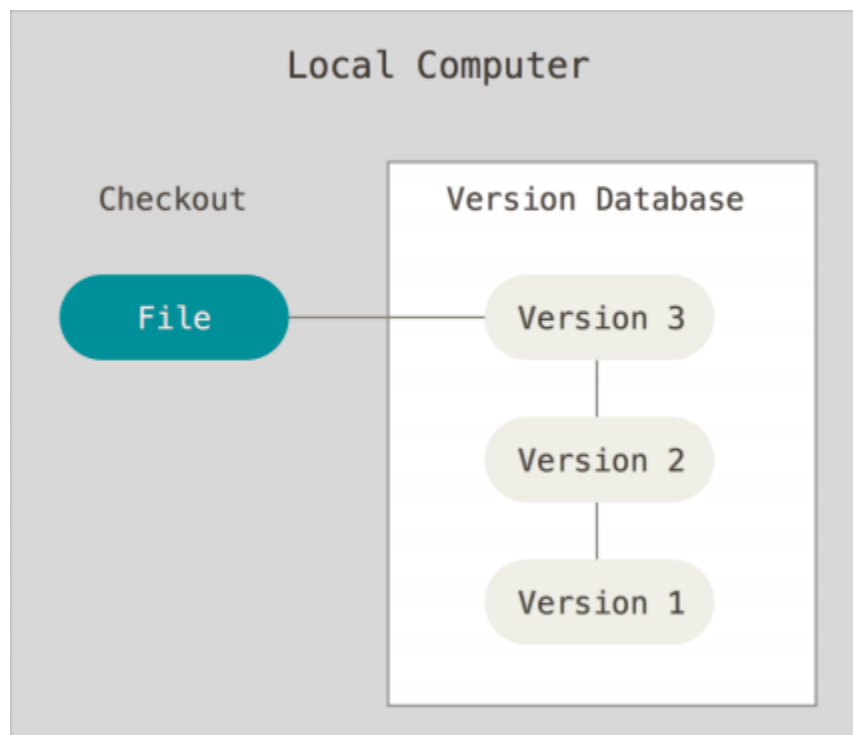


毕业论文第1版
毕业论文第2版
毕业论文第3版
毕业论文第4版
毕业论文第5版
毕业论文最终版
毕业论文最终版返回需修改版
毕业论文最终版返回需修改版1
毕业论文最终版返回需修改版2
毕业论文最终版返回需修改版3
毕业论文最终版返回需修改版4
毕业论文最终版返回需修改版打死再也不改版
毕业论文最终版返回需修改版打死再也不改版2
毕业论文最终版返回需修改版打死再也不改版3

VCS发展历程—本地版本控制系统

- 本地版本控制系统

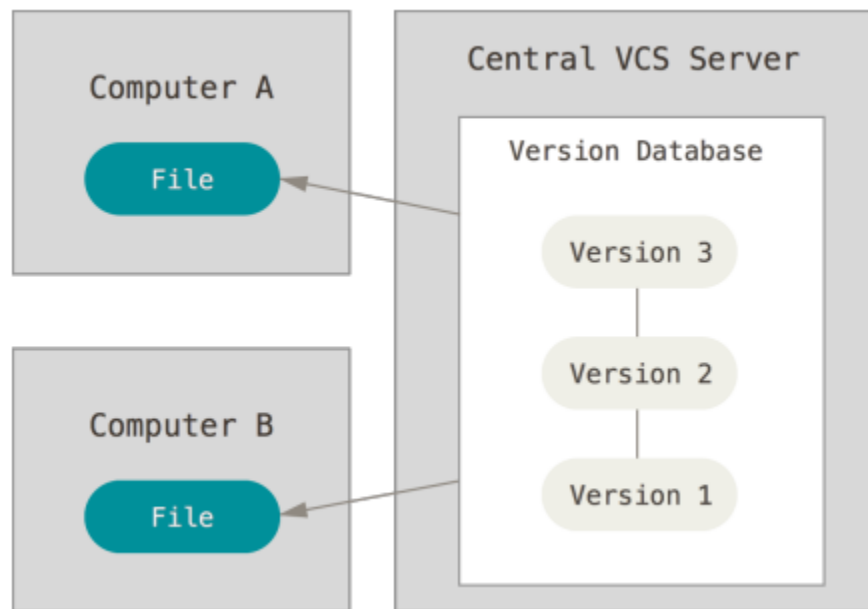
- 复制整个项目工作目录来保存不同的版本
- 简单易用但不易管理追踪且数据丢失不易恢复
- RCS：在本地使用简单数据库记录文件的修改



VCS发展历程—集中版本控制系统

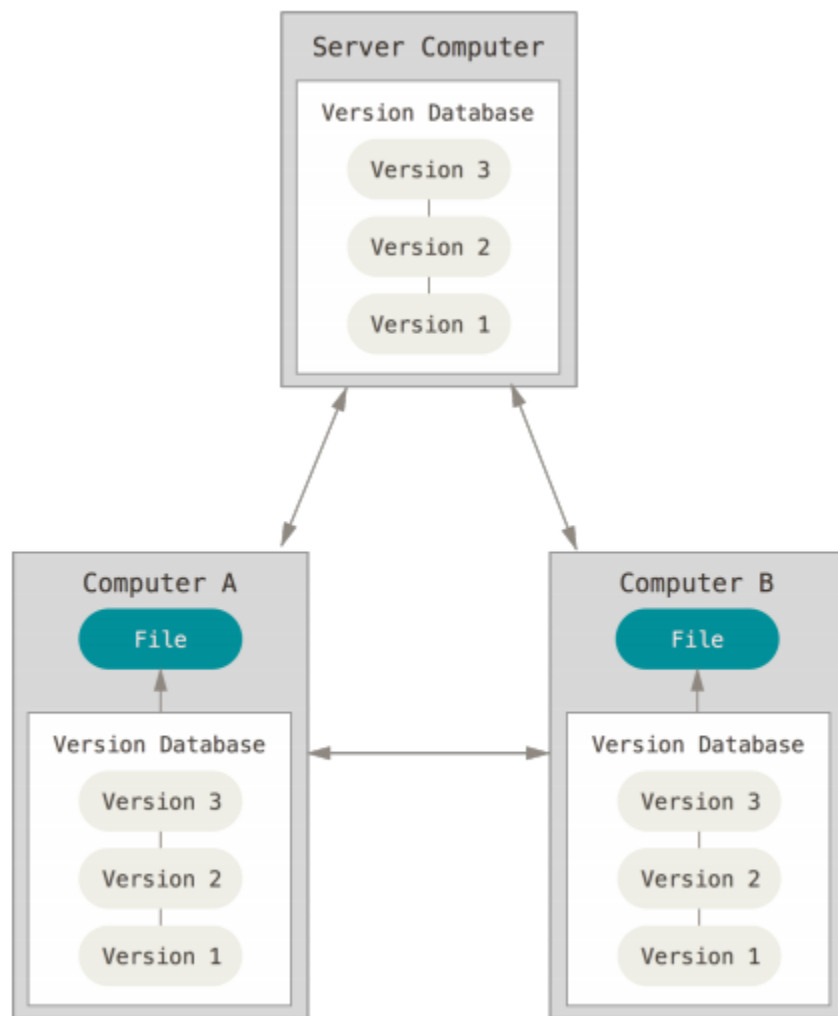
- 集中版本控制系统

- CVCS：集中化的版本控制系统
- 服务器保存文件修改历史，客户端通过网络连接服务器，取出最新文件或者将本地提交更新到服务器
- 版本库只保存在服务器，全球仅此一家！
- 缺点：断网后无法工作、数据丢失风险大



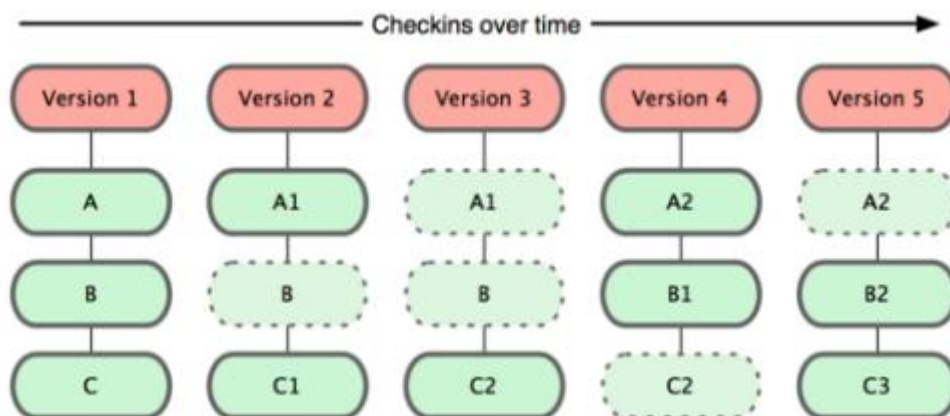
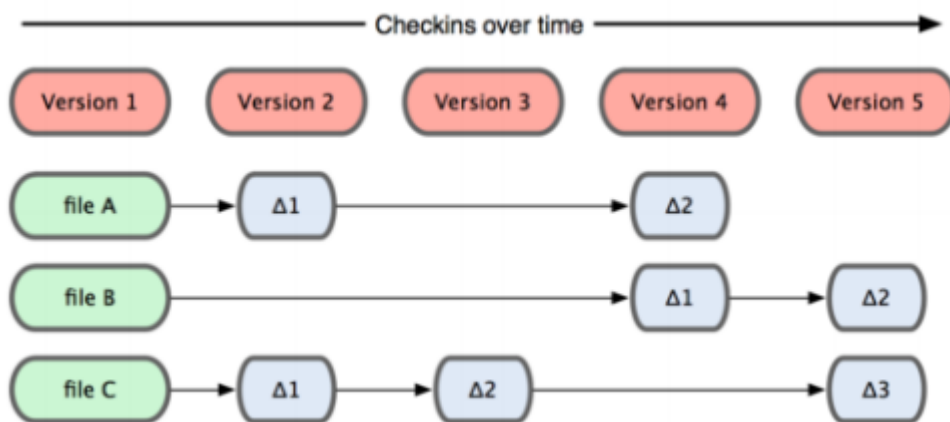
VCS发展历程—分布式版本控制系统

- 分布式版本控制系统
 - 版本库**不仅仅**存放在服务器，**客户端都有一份完整拷贝**
 - 主服务器出现故障，从任何一个客户端都可以恢复版本
 - 有了分布式，再也不用担心服务器崩溃了！



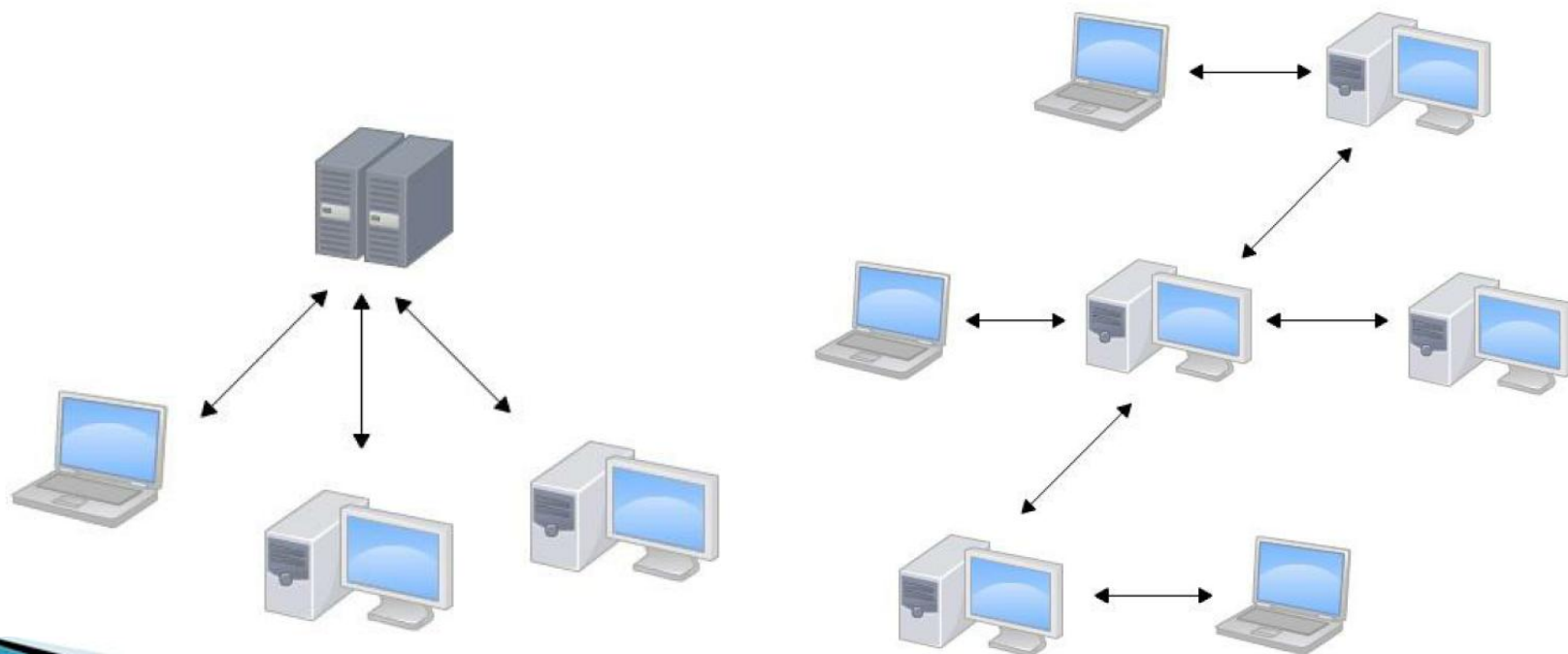
Git特点

- 存储直接快照，而非比较差异



Git特点

- 断网不罢工，也能照常工作
 - 本地保存完整版本库，而集中式版本库保存在服务器，需要网络才能工作



Git特点

- 杀手锏：分支管理
 - 分支切换几乎瞬间完成(内部原理后面会细讲)



VS



为什么要学习Git

- Git与其它VCS优势
 - 版本采用快照而不是补丁
 - 分支切换灵活，速度快
 - 断网后也可以继续干活
 - 管理维护更加方便快捷
 - 高效率、安全性高，而且还开源免费
- Git越来越流行
 - 越来越多的开源项目开始迁移到Git
 - Linux、Android、Perl、Eclipse、Gnome、KDE、Qt、ruby on Rails、PHP、Debian、jQuery...
- Git对学习和工作上的帮助
 - 学会使用Git跟踪、参与开源项目
 - 公司版本控制必须掌握的技能
 - 打开开源世界的大门、学习提高的捷径
 - Git不仅仅是一个下载工具，更是项目管理、学习前沿技术的利器！

本教程简介

- 教程简介

- 30个常用命令的熟练掌握
- Git内部原理理解：一个仓库、二个引用、三大工作区、四大Git对象
- Git实战：如何利用Git去开展自己的工作、团队协作

- 学习环境

- Ubuntu14.04 + Git-1.9
- Ubuntu10.04 + Git-2.8
- Fedora + Git-1.5
- Win7 + Git-2.7
- Github、OSChina远程仓库代码托管
- 本地Git服务器搭建

- 课程目标

- 熟练掌握Git的使用，能处理各种复杂情况
- 熟悉企业版本控制流程
- 熟练使用Git跟踪开源项目、与他人协作、管理自己的开发工作

第二章 Git安装

在Linux/unix环境下安装

- 自动安装(安装成熟稳定版本)
 - Ubuntu\ Debian下安装: `apt-get install git(git-core)`
 - Fedora\ RedHat下安装: `yum install git`
 - Mac下安装: `sudo brew install git`
- 源码包安装(安装最新版本)
 - 最新git源码下载地址:
 - <https://github.com/git/git/releases>
 - 安装git之前, 需要安装其依赖的包
 - `apt-get install curl curl-devel zlib-devel openssl-devel perl cpio expat-devel gettext-devel`
 - 需要安装libiconv编码转换库
 - 源码下载、配置、编译、安装
 - `curl -O https://github.com/git/git/releases/git-2.8.1.tar.gz`
 - `make configure;./configure --prefix=/usr/git`
 - `make prefix=/usr/git`
 - `make install`
 - 配置环境变量PATH

在Windows下安装Git

- Cygwin模拟环境
 - 模拟了Unix的系统调用API，在Windows下可以运行Linux工具和shell命令
 - 在该环境下安装git
- Wine模拟环境
 - 模拟了Windows的系统调用API，可以在Linux下面运行Windows应用程序
- Msysgit
 - Git的“Windows版本”
 - 将git和cygwin一起打包，集成了Git所需要的运行环境和组件，直接安装即可

第3章 快速开始

第3章 快速开始

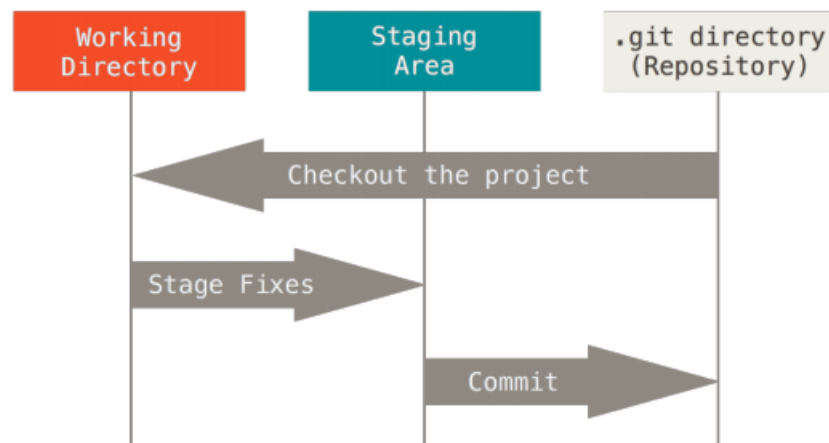
- 新建仓库: `git init`
- git配置: `git config`
- 工作区修改、保存修改、提交修改
 - `git add`、`git commit`
- 查看状态: `git status`
- 查看提交历史: `git log`
- 查看提交差异: `git diff`
- 查看某个提交具体修改: `git show`
- 克隆一个远程仓库:
 - `git clone repo_url`

Git仓库创建及配置

- 创建一个仓库
 - `git init`
- Git配置
 - 姓名: `git config --global user.name "wit.wang"`
 - 邮箱: `git config --global user.email "wit.wang@qq.com"`
 - 差异颜色显示: `git config --global color.ui true`
 - 设置别名: `git config --global alias.it init`
- 命令自动补全
 - 下载配置脚本
 - `https://github.com/markgandolfo/git-bash-completion.git`
 - 拷贝到用户目录并生效
 - `$ echo source ~/git-completion.bash >> ~/.bashrc`
 - 参数选项:
 - `--global`: 修改全局配置文件, 去配置`~/.gitconfig`
 - `--system`: 修改所有用户的配置文件:`/etc/gitconfig`
 - 无参数: 修改本仓库配置文件: `.git/config`

Git基本流程

- 在工作目录(工作区)中修改某些文件
- 对已修改文件作快照, 并保存到暂存区域
 - `git add file.c`
- 将保存在暂存区的文件快照提交到版本库
 - `git commit -m "commit info"`
- 查看提交历史
 - `git log`



分支和标签

- 创建分支
- 切换分支
- 查看分支
- 创建标签
- 查看标签
- 删除标签

第4章 Git基本概念

一个仓库
二个引用
三大工作区
四个对象

Git基本概念

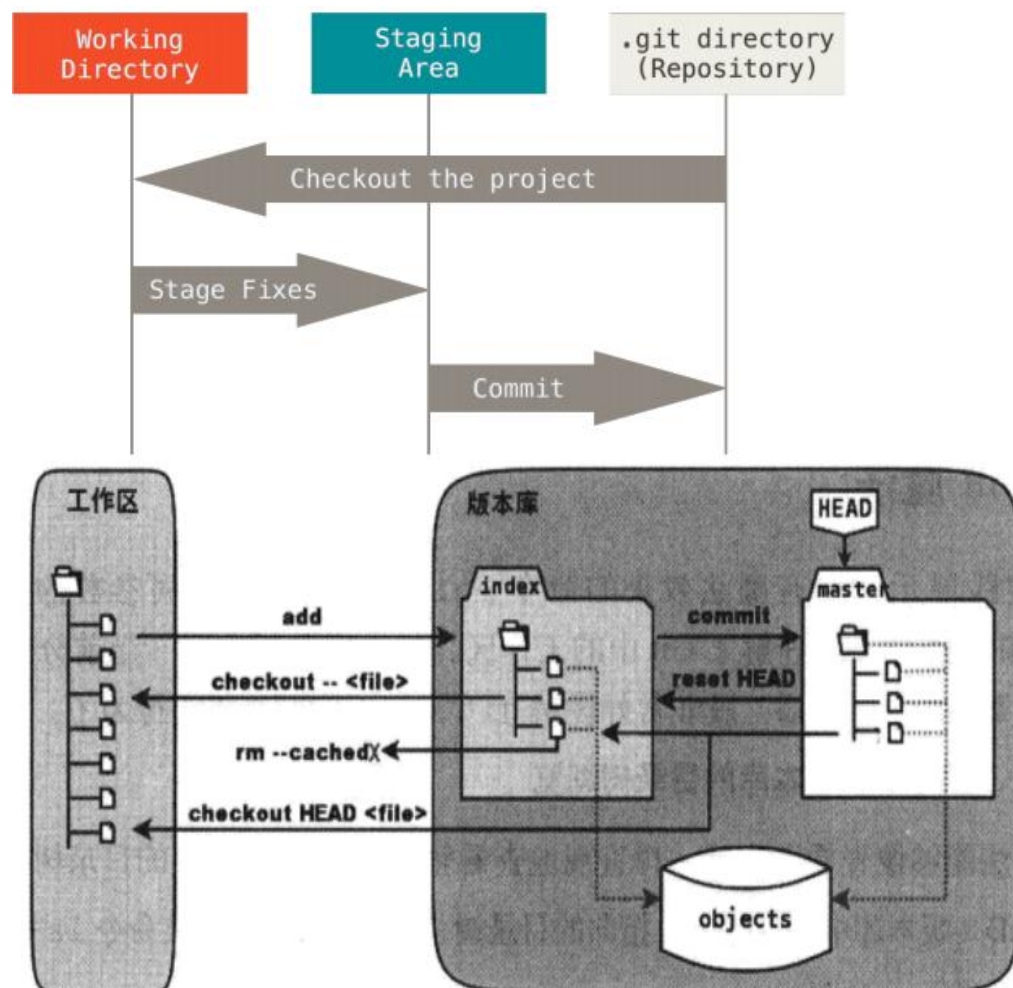
- 专业术语中英文对照
- 工作目录、暂存区和版本库
- .git目录
- git对象模型
 - blob
 - tree
 - commit
 - tag
- 引用

专业术语中英文对照

英文	中文	英文	中文	英文	中文
repository	仓库、版本库	merge	分支合并	snapshot	快照
ref	引用	rebase	分支衍合	staging	暂存区
bare	裸仓库	cherry-pick	条件分支	SHA1	哈希值
SVN	Subversion	squash	压合分支	commit	提交
GIT	分布式RCS	checkout	检出	branch	分支
SSH	安全传输协议	revert	反转提交	tag	标签
HEAD	当前分支	stash	储藏	index	索引
pull	拉取代码	master	主分支	origin	远程仓库
push	推送代码	Gerrit	代码审核		

工作目录、暂存区和版本库

- Git本质
 - 一套内容寻址的文件系统
- 版本库中文件的三种状态
 - 已修改: modified
 - 已暂存: staged
 - 已提交: committed
- 三大工作区
 - 工作目录
 - 暂存区域
 - 版本库(仓库)
- 工作区下的文件状态
 - 未被追踪: untracked
 - 被追踪: tracked



Git索引(index)

- 什么是索引

- 存储了一个tree对象所有信息的二进制文件
- 里面有很多条目，分别指向不同blob、tree哈希值

- 索引是一种暂存区域(staging area)

- 我们文件修改内容并没有保存到该区域
- 索引实际上是一个包含文件索引的目录树
 - 记录了文件名和文件状态信息(时间戳、文件长度等)
 - 文件的内容并没有保存到其中
 - 文件索引建立了文件和对象库中对象之间的关联

.git目录

仓库子目录	功能描述
branches	项目分支信息
hooks	默认的hooks脚本，由特定事件触发
info	内有exclude文件：指定git要忽略的文件
logs	历史记录，删除的commit对象等
objects	Git数据对象：commit、tree、blob、tag
refs	Git引用：指向(远程)分支、标签的指针
config	Git项目配置信息
HEAD	指向当前分支的末端
index	Staging area 暂存区
COMMIT_EDITMSG	最后一次提交的注释
description	Git项目描述信息

```
branches
COMMIT_EDITMSG
config
description
HEAD
hooks
index
info
logs
objects
refs
```

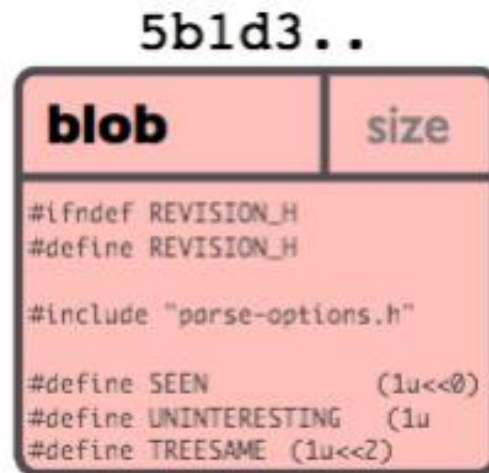
```
branches
COMMIT_EDITMSG
config
description
HEAD
hooks
  applypatch-msg.sample
  commit-msg.sample
  post-update.sample
  pre-applypatch.sample
  pre-commit.sample
  prepare-commit-msg.sample
  pre-push.sample
  pre-rebase.sample
  update.sample
index
info
  exclude
logs
  HEAD
  refs
objects
  21
  8c
  96
  b2
  b8
  d1
  d4
  db
  info
  pack
refs
  heads
  tags
```

Git对象模型

- 对象(objects)
 - 类型：4种数据对象:blob、tree、commit、tag
 - 大小：对象数据内容的大小
 - 文件名：如果SHA1算法生成
- SHA1对象数据
 - 用40个字符的字符串用来表示对象名：目录+名
 - 字符串由对象内容做SHA-1哈希计算得来
 - 通过比较SHA-1值来比较两个文件的内容：快！
- 不同类型的对象用途
 - blob：存储文件数据，通常是一个文件
 - tree：类似一个目录，用来管理tree和blob
 - commit：指向一个tree，标记项目某个特定时间点状态
 - tag：用来标记某一个提交(commit)

blob

- blob对象
 - 对象内容全部是二进制格式数据
- 查看blob文件内容
 - `git show SHA1`
- 对象名
 - 由哈希计算结果生成



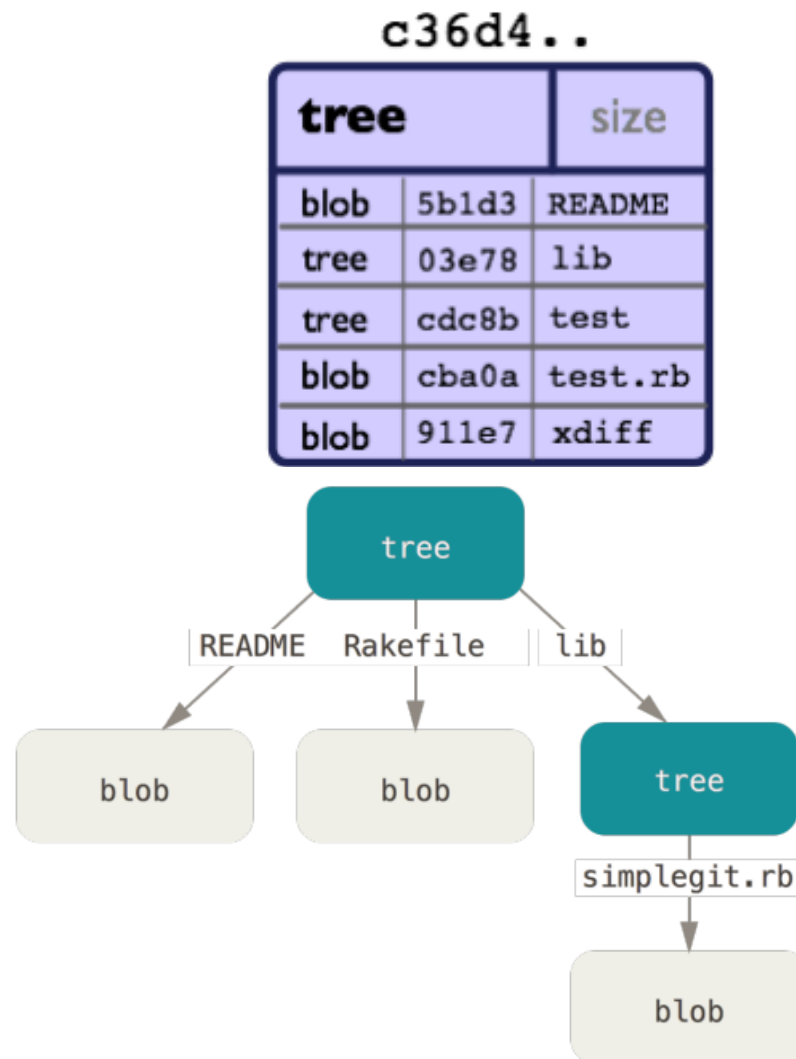
tree

- tree对象

- 包括：mode、对象类型、SHA1值、名字
- 一串指向blob或其它tree对象的指针
- 表示目录树的内容、内容之间的层次目录关系

- 查看tree对象内容

- `git ls-tree SHA1`
- `git cat-file -p master^{tree}`
- `git cat-file -p SHA1`



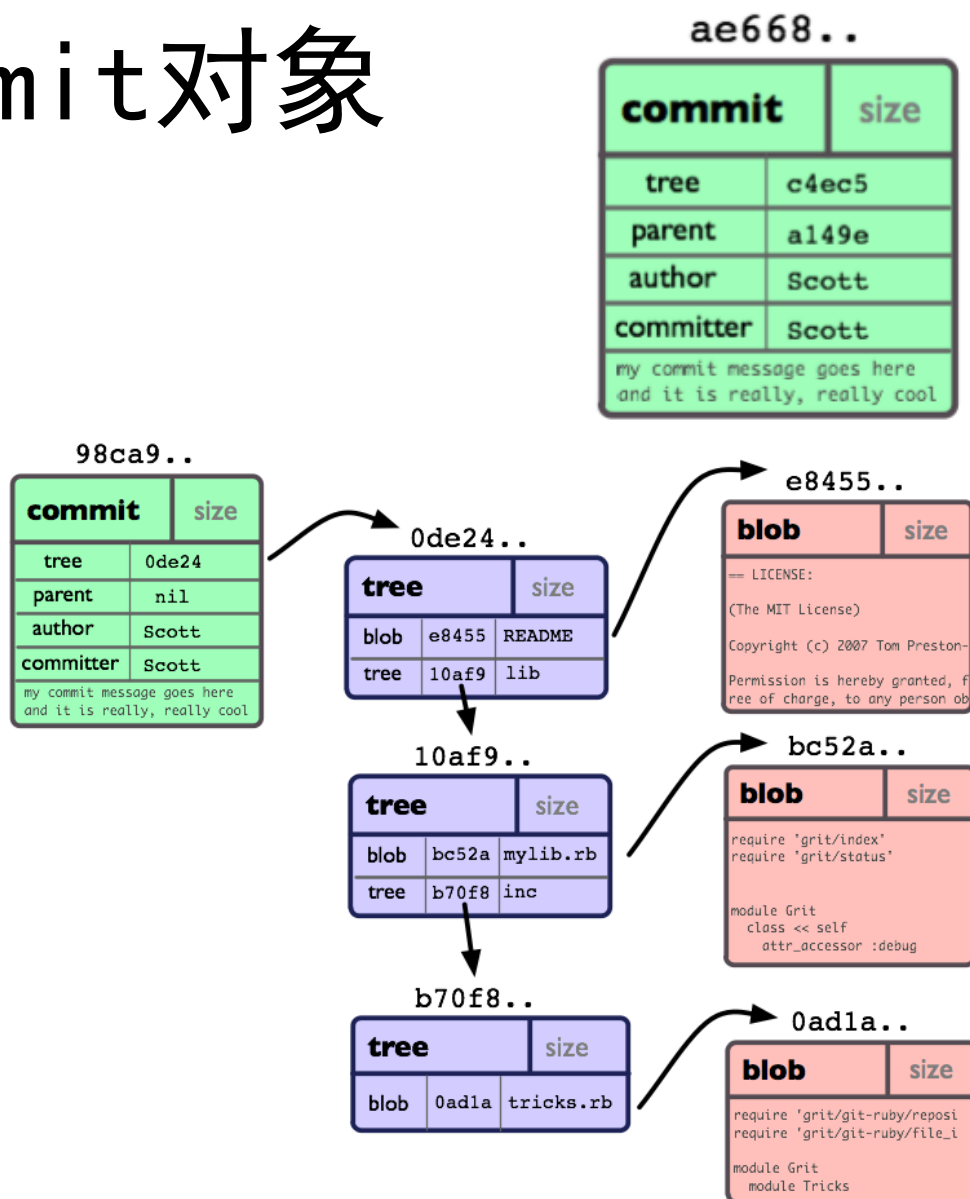
commit对象

- commit对象

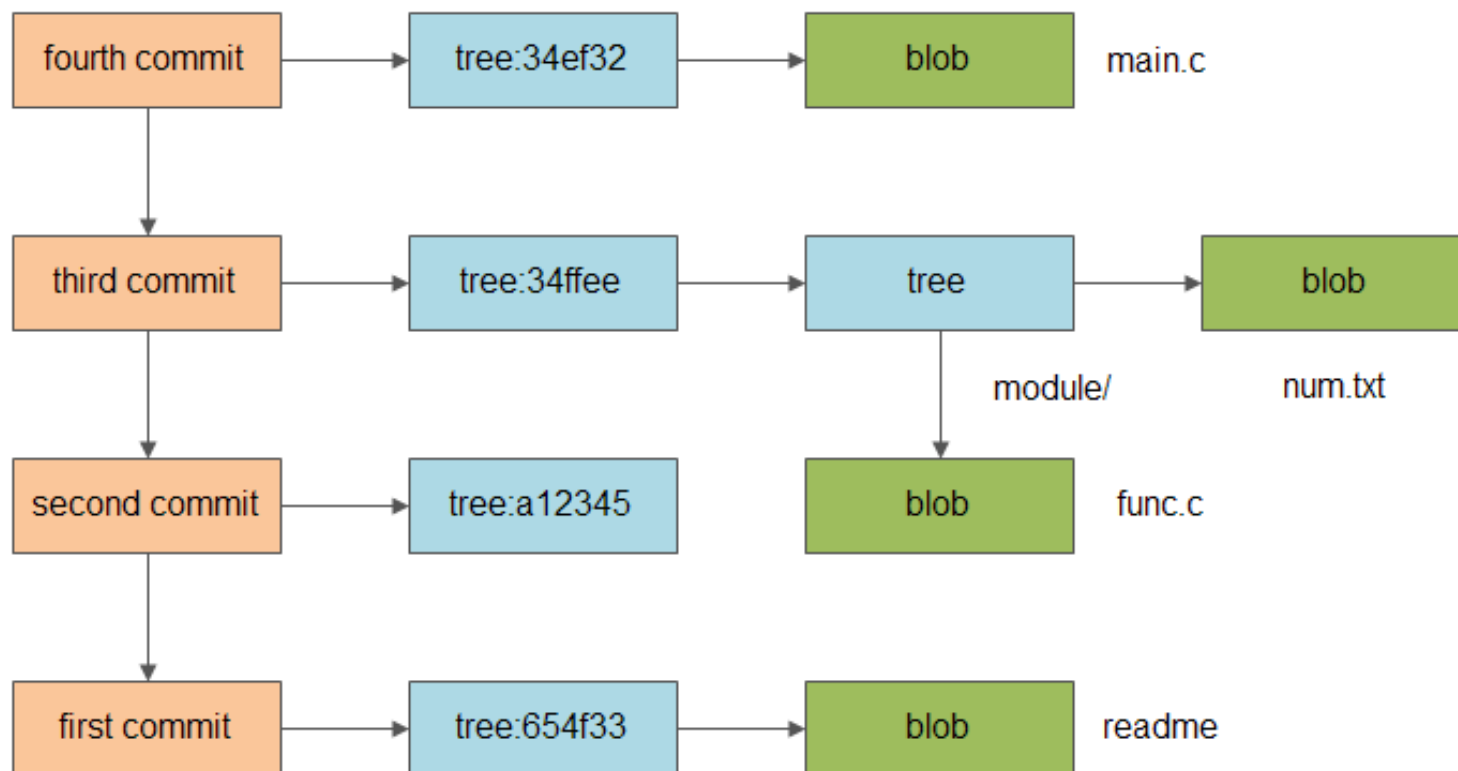
- 用来指向一个tree对象
- 组成
 - 一个tree对象
 - 父对象：一个项目必须有根提交
 - 作者：修改人名字、日期
 - 提交者：实际创建提交的名字、日期
- 一个提交本身并没有包含任何信息说明其做了哪些修改，所有的修改都是通过与父提交比较得来

- 提交commit对象

- 使用git commit命令提交
- 将存储在暂存区的index全部提交
- 提交的父对象为当前分支HEAD



Commit提交版本库库视图



Git引用

- 引用的本质：指针
- 分支、标签都是对提交的引用（指向commit的指针）
- 引用存放路径：`.git/refs`

tag

- 标签对象

- 组成

- 对象名、对象类型、标签名、标签创建者名字、

- 指向一个commit的SHA1

- 与分支比较

- 都是指向一个commit
 - 标签可以看做是一个常量指针，不能改变
 - 分支是一个变量指针

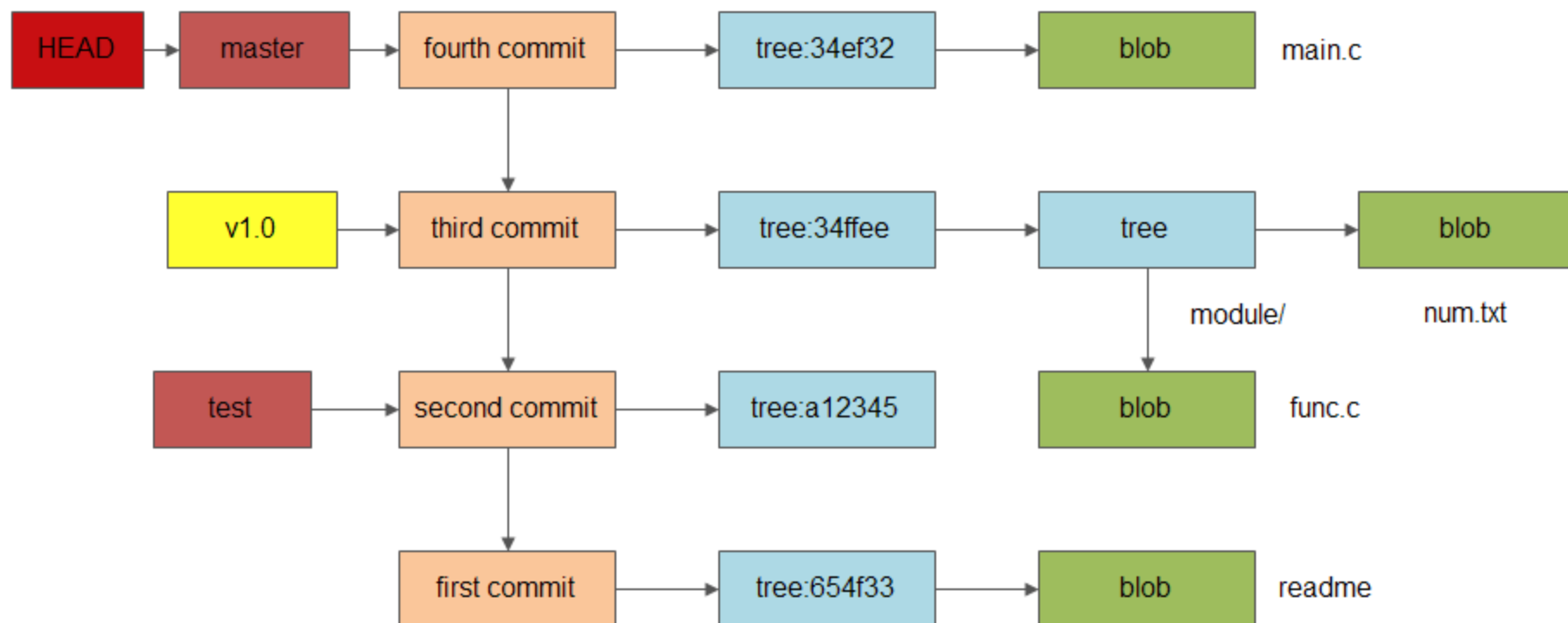
- 查看标签信息

- `git cat-file -p v1.0`

```
49e11..
```

tag		size
object	ae668	
type	commit	
tagger	Scott	
my tag message that explains this tag		

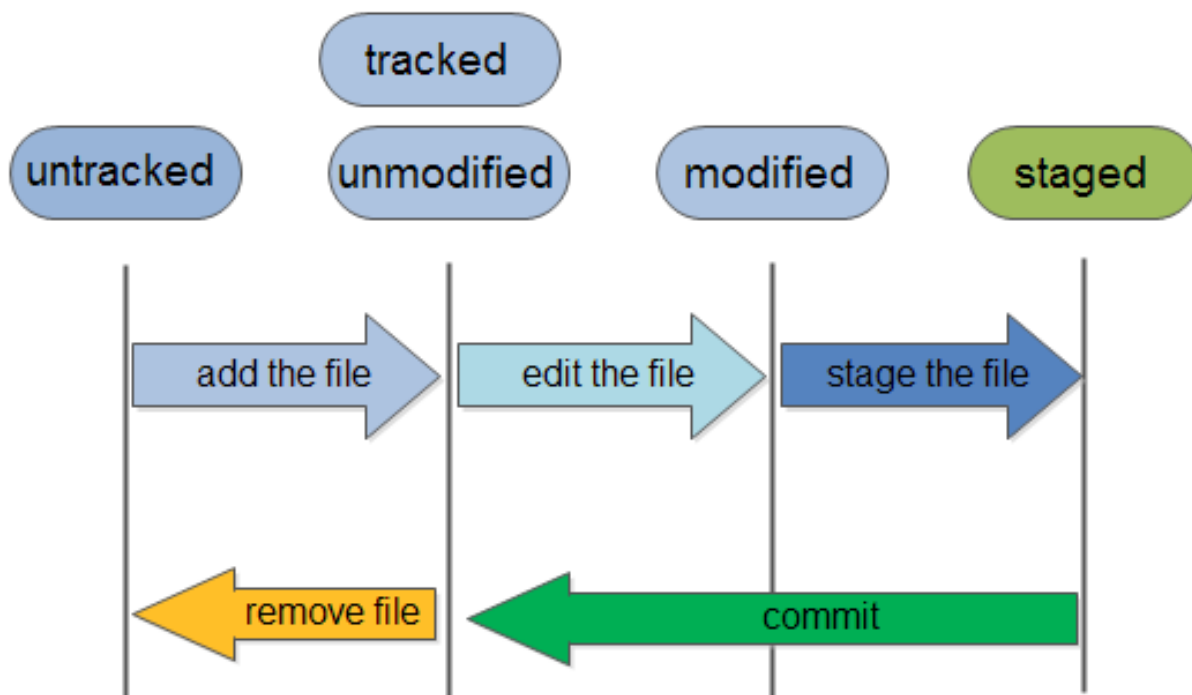
版本库引用视图



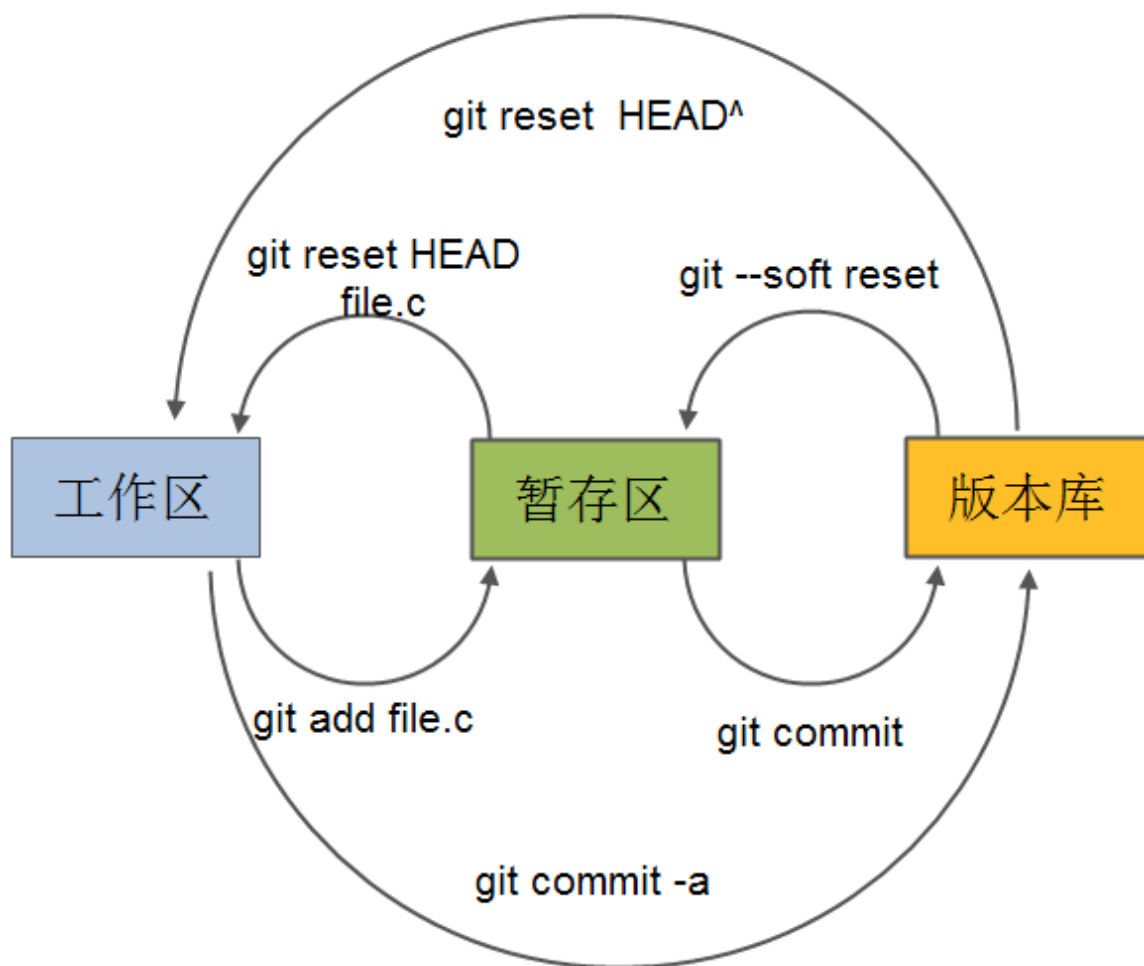
第5章 翻仓倒海：文件管理

文件管理

- 文件添加及提交
- 删除文件
- 重命名文件
- 忽略文件
- 撤销修改
- 比较差异
- 压缩仓库



三大工作区状态转换



文件添加及提交

- 文件添加

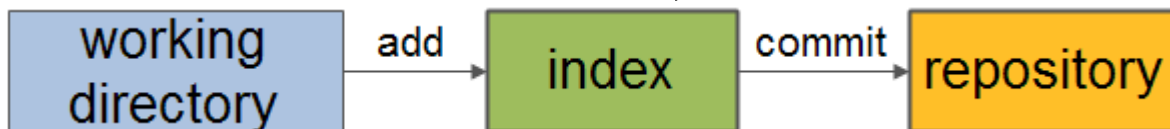
- 添加某个文件: `git add file.c`
- 添加所有文件: `git add .`
- 作用
 - 已经追踪的文件: 将修改从工作区保存到暂存区
 - 未追踪的文件: 加入git追踪范围

- 文件提交

- 将暂存区的所有内容提交到版本库
- 单步提交: `git commit -m “commit info”`
- 一次性提交: `git commit -a`
- 修改最后一次提交: `git commit --amend`

- 注意事项:

- 如果修改没有保存到暂存区, 是不会提交到版本库中的



文件删除

- 从工作目录中删除
- 从工作目录和暂存区中删除
 - `git rm -f file.c`
 - 避免文件出现在未跟踪清单中
 - 删除并提交后，该文件就不再被版本库追踪，但是版本库中仍然有这些文件的快照
- 从暂存区中删除，工作目录中保留
 - 命令：`git rm --cached file.c`
 - 包括：库文件、可执行文件、日志、临时文件
- 从版本库中删除：版本回退到错误提交前的版本
 - 撤销提交到暂存区：`git reset -soft SHA1`
 - 撤销提交到工作区：`git reset -mixed SHA1`
 - 将工作区、暂存区和版本库恢复到指定版本：`git reset --hard SHA1`
- 小结
 - 删除也算一个修改，也可以从版本库中还原

文件重命名

- 文件重命名：
 - `git mv old_file new_file`
- 重命名实际操作
 - 文件移动：
 - `mv old_file new_file`
 - 删除旧文件：
 - `git rm old_file`
 - 添加新文件：
 - `git add new_file`

忽略文件

- 有些文件不必提交到版本库中
 - 可执行文件、日志文件、临时文件、库文件
- 忽略文件模式：glob模式匹配
 - 忽略以#开始的行
 - 忽略某种格式结尾文件：*. [ao]
 - 某个库文件除外，不忽略：! c lib. a
 - 忽略临时文件：*~
 - 忽略根目录下的某个文件：/text
 - 忽略某个目录下的所有文件：libs/、libs/*. a
- 手动创建. gitignore文件

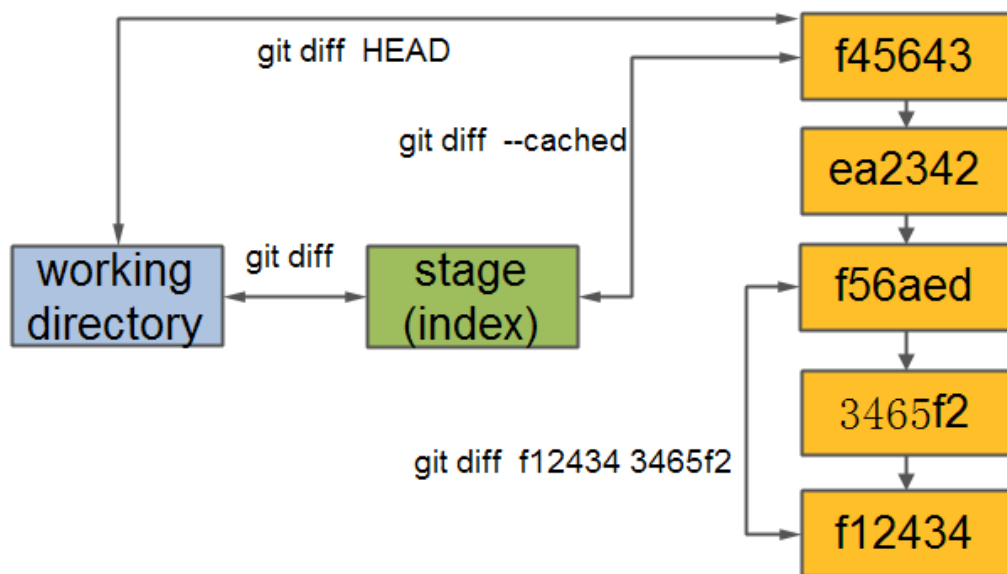
撤销修改

- 撤销工作区的修改
 - `git checkout --file.c`
 - 让文件回到最近一次commit或add的状态
 - 若文件还没添加到暂存区：撤销后和版本库一样
 - 若文件已添加到暂存区且做了修改：回到暂存区状态
- 撤销暂存区的内容
 - `git reset HEAD file.c`
 - 将暂存区的修改撤销掉, 重新放回工作区
- 撤销版本库的提交
 - `git reset --hard SHAI (HEAD^)`
 - 回退版本, 并刷新到工作区中
 - 先前的提交对象还在版本库中孤独地存在着



差异比较

- 比较工作区和暂存区差异
 - `git diff`
 - 查看尚未暂存文件有哪些新的修改
- 比较暂存区和版本库差异
 - `git diff --cached [HEAD]`
 - `git diff --staged SHA`
 - 查看已暂存文件和上次提交的快照之间的差异
- 比较工作区和版本库差异
 - `git diff HEAD (SHA1)`
 - 查看未暂存文件和最新提交文件快照的区别
- 比较两个版本之间差异
 - `git diff SHA1 SHA2`
 - 查看不同版本之间的差异



压缩仓库

- 压缩版本库

- 在日常运行中，git快照会占用磁盘空间
- Git会在增量存储单元中存储修改
- 通过git gc命令压缩增量存储单元，节省磁盘空间

- 快照的存储

- 对于修改的内容：做快照处理并保存
- 对于未修改的文件：做引用处理

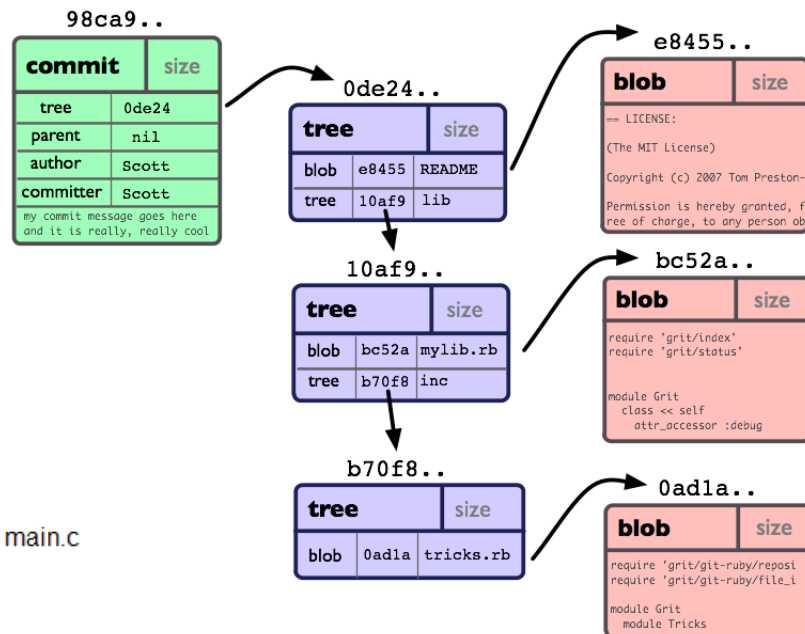
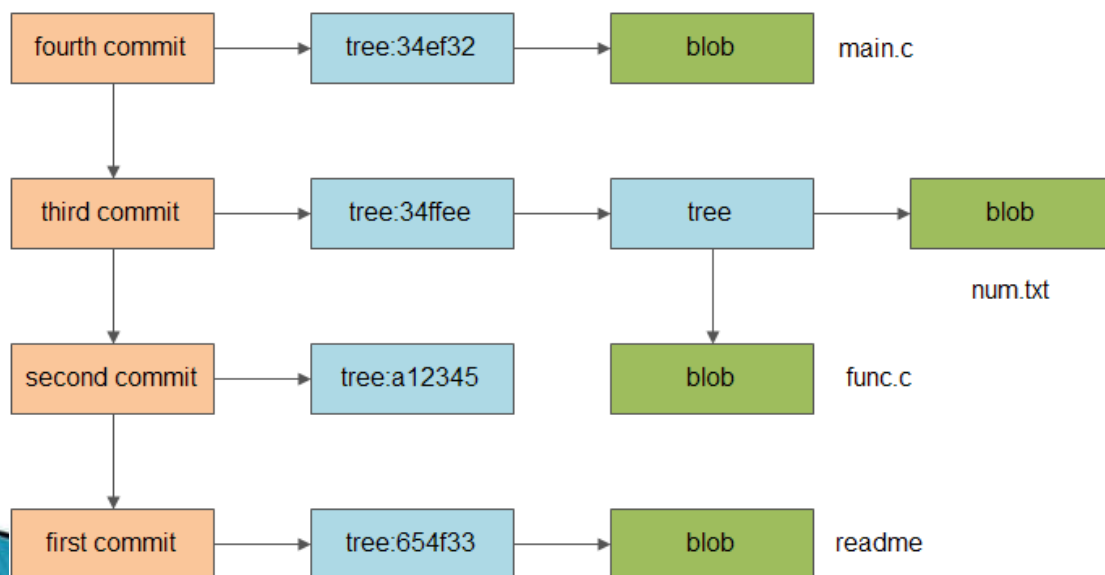
第6章 时光穿梭：历史也可以重写

本章大纲

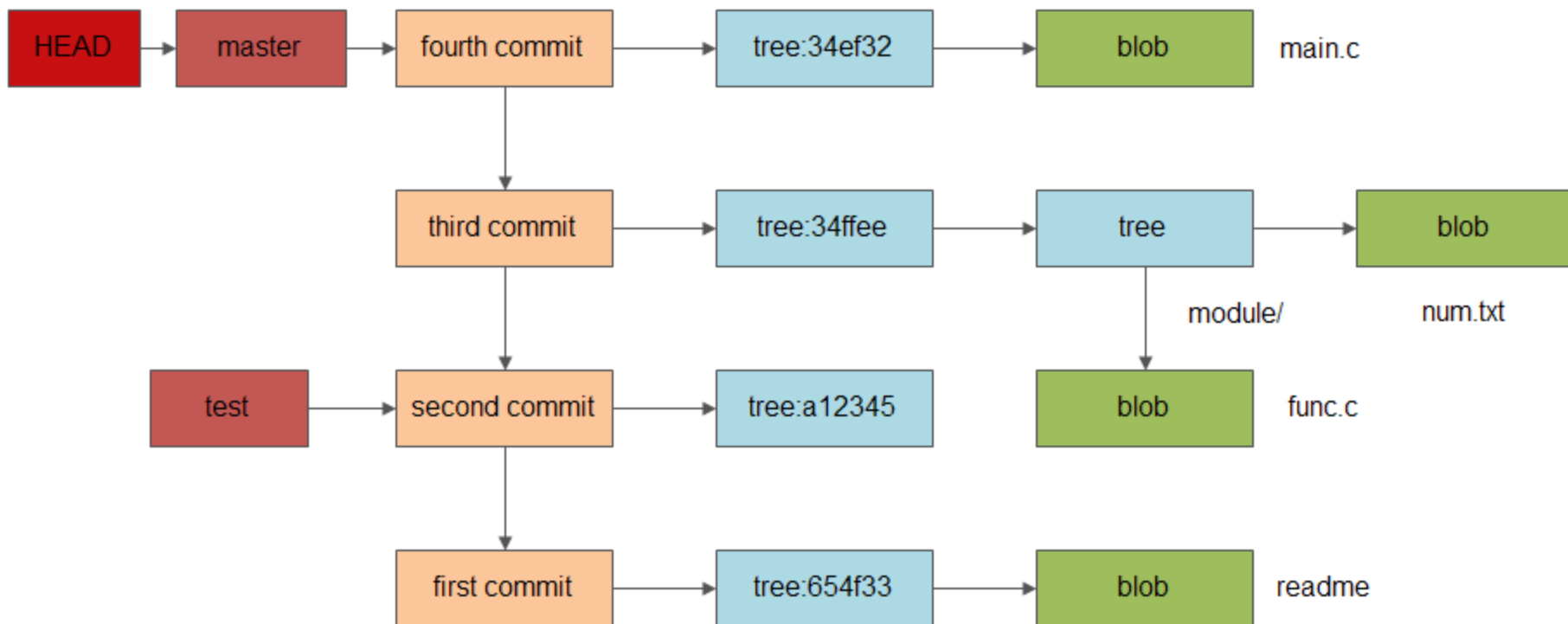
- commit对象与存储
- 查看提交历史
- 提交查找
- 修改提交
- Git置位
- 给提交重新排序
- 将多个提交合成一个提交
- 将一个提交分解为多个提交
- 恢复隐藏的历史

commit对象与存储

- 什么是commit对象
 - 版本库如何保存历史?
 - commit对象存储
- 提交演示



Commit对象与存储



查看提交历史

- 参数选项

- 显示每个版本的修改差异:

- `git log -p -2` | `git shortlog`

- 单行显示3条记录:

- `git log -3 --pretty=oneline` | `git log --oneline`

- 图形化显示:

- `git log --graph` | `gitk`

- 缉凶利器: `git blame`

- 查看一个文件每一行的提交记录

- 查看第6到11行代码是谁的提交:

- `Git blame file.c -L 6,+5`

- 查看某个指定文件的提交修改历史

- `Git log -C file.c`

查看提交历史

- 更多的参数供我们选择…
 - 展开版本之间的内容变化: `-p`
 - 显示最近的两次更新: `-n`
 - 指定位置开始的n个提交: `git log HEAD^~2 -n`
 - 两个版本之间的提交: `git log HEAD~3..HEAD`
 - 仅显示行数变化: `--stat`
 - 显示新增、增改、删除文件清单: `--name-status`
 - 仅显示跟指定作者相关的提交: `--author`
 - 仅显示跟指定提交者相关提交: `--committer`
 - 指定日志的起点: `git log SHA1`

提交查找

- 关键字查找

- 不同的参数灵活使用

- 查找所有包含hello字符串的文件: `git grep hello`
 - 显示关键字所在文件的行号: `git grep -n hello`
 - 不显示内容, 只显示文件名: `git grep --name-only hello`
 - 查看每个文件有多少个匹配: `git grep -c hello`

- 在特定版本里查找

- 在V1.0版本里查找: `Git grep string v1.0`

- 组合查找

- 与查找、或查找

- 查找hello world: `git grep -e hello --and -e world`
 - 查找hello或world: `git grep --e hello --or -e world`

修改提交

- 增补提交

- 修改最后一次提交：修改提交信息或者内容

- `git commit --amend`

- 的

- 反转提交

- 将原来的提交取消掉

- `git revert SHA1`

- 反转多个提交：要先反转最后提交，防止冲突。此时原操作的逆操作都在暂存区，然后重新手动提交

- `git revert -n HEAD`

- `git revert -n SHA1`

- `git commit -m “revert HEADand SHA1”`

Git置位

- 复位

- 版本回退到工作区

- `git reset [--mixed] SHA1`

- 软复位

- 回退版本到暂存区

- `git reset --soft SHA1`

- 硬复位

- 会从版本库和工作目录中同时删除提交

- `git reset --hard SHA1`

给提交重新排序

- 修正主义历史学家会篡改历史
- 完美主义程序员改写提交历史
- 即使已经“载入史册”（提交到版本库），我们也可以“翻案”，也可以修改。
- 通过人机交互模式改写历史
 - 22
 - `git rebase -i HEAD~3`
 - 编辑提交信息
 - 退出，git会重新排序提交
- 改写历史会生成新的commit

将多个提交合成一个提交

- 把大象装到冰箱里分三步
 - 打开冰箱门
 - 把大象放到冰箱里
 - 关上冰箱门
- 操作步骤
 - 首先定位要合并的三个提交
 - `git rebase -i HEAD~3`
 - 将最新的2个提交合并到他们的parent
 - `HEAD\HEAD^` : pick→squash
 - 修改提交信息
 - 保存退出，查看提交历史
 - `git log -oneline`
 - 会生成一个新的commit，而不是原来的那个`HEAD^^`

将一个提交分解成多个提交

- 讲述你把大象装进冰箱的故事…
- 主要分三步
 - 打开冰箱门
 - 把大象装进冰箱里
 - 关上冰箱门
- 操作步骤
 - 首先进入交互模式
 - `git rebase -i SHA1`
 - 定位需要修改的commit
 - `pick→edit`
 - 检出到工作区，重新修改，多次提交
 - 继续rebase
 - `git rebase --continue`
 - 大功告成，查看提交记录

恢复隐藏的历史

- 找回“迷途的大雁”
 - 删除的提交对象其实并没有真正删除
 - 称为悬垂提交对象，仍在保存在仓库里
 - 通过`git reflog`命令查看当前状态
- 穿越历史：
 - 直接回退到第N次改变之前的版本：
 - `Git reset --hard master@{n}`

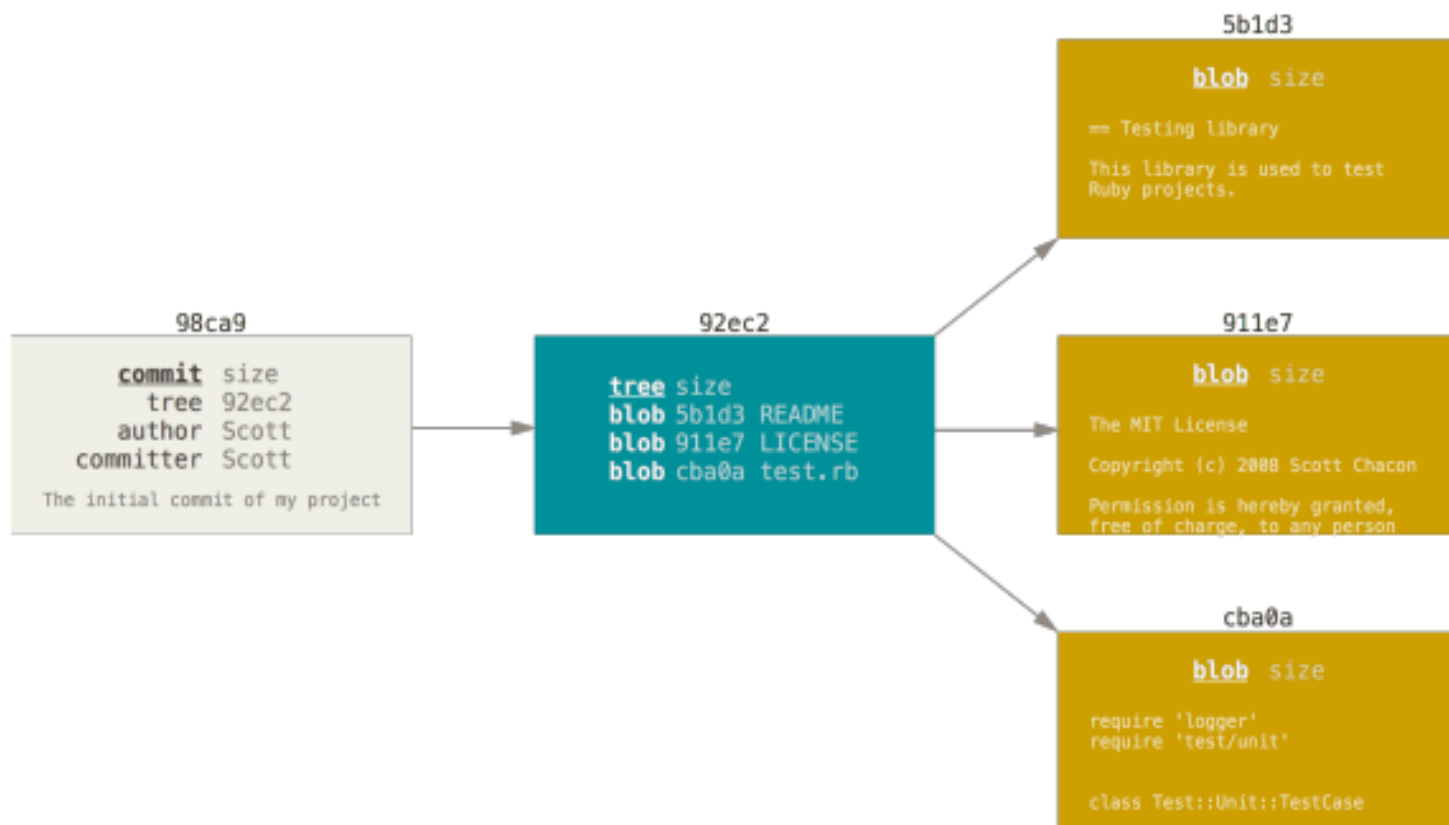
第7章 分支管理

分支管理

- 分支的概念
- 分支基本使用
- 分支合并
- 分支衍合
- 可选择分支
- 分支修改储藏
- 恢复已删除分支的提交
- 一个脱离了组织的提交：no branch

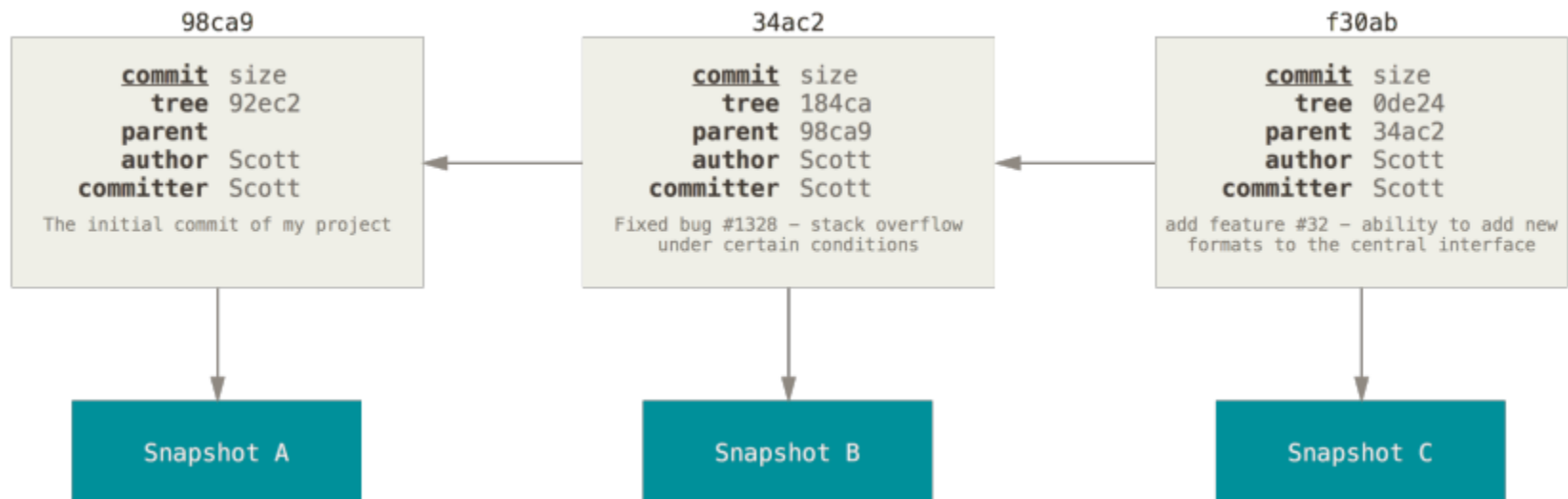
分支的概念(1)

- 一个提交的仓库视图



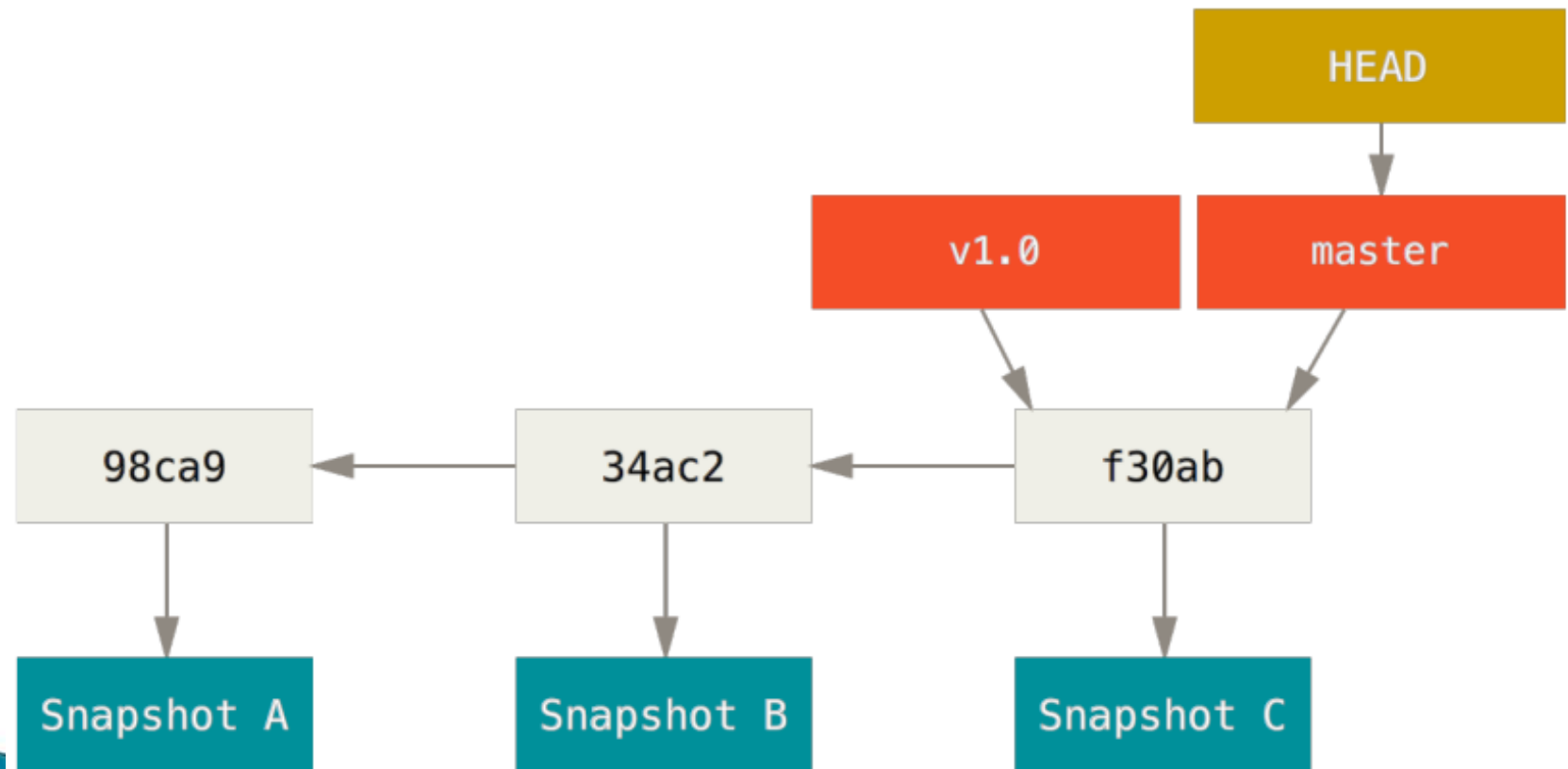
分支的概念 (2)

- 多个提交组成一个链



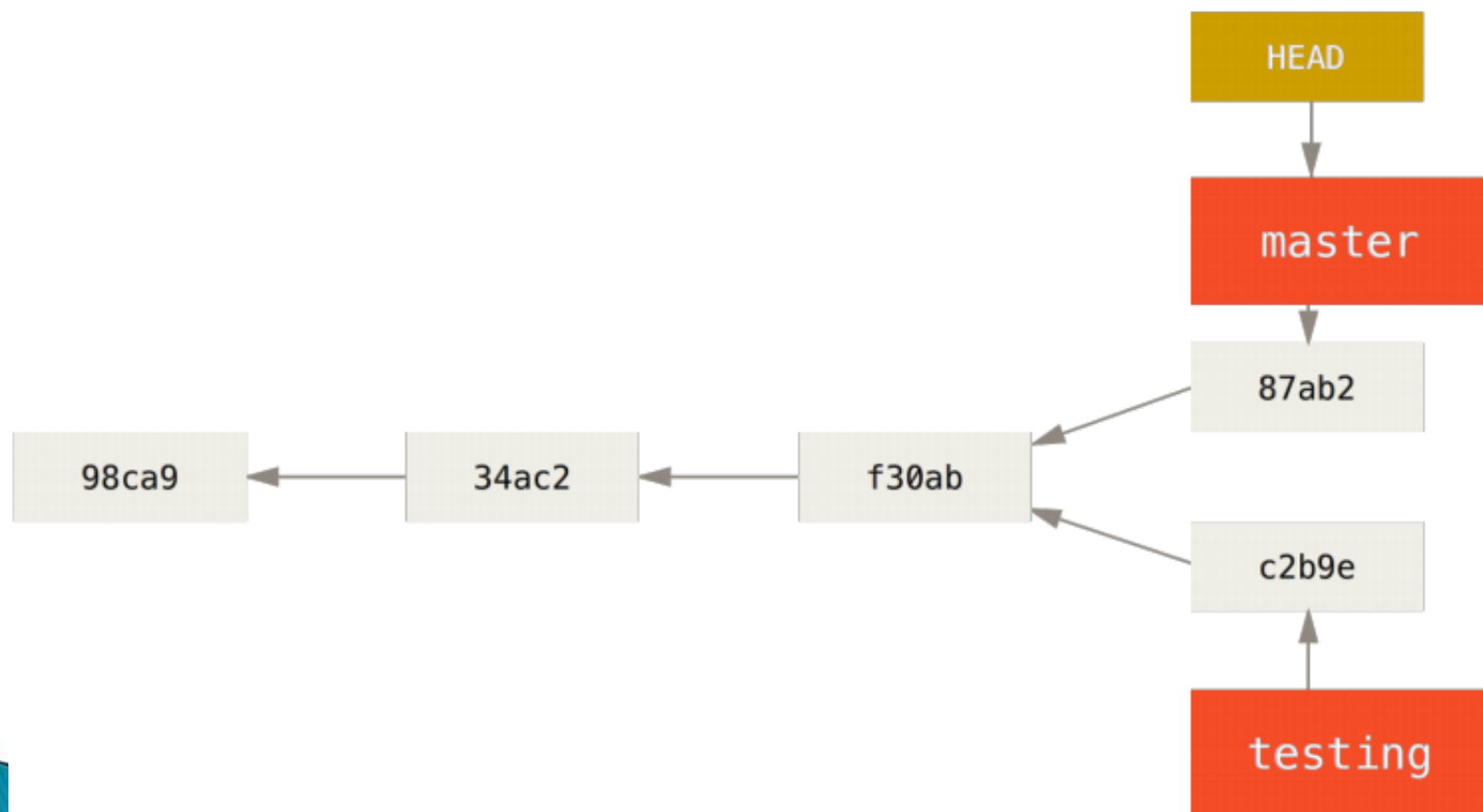
分支的概念 (3)

- 分支的本质：指向一个commit的指针



分支的概念(4)

- 切换分支时，HEAD如何变化？



分支的概念 (5)

- 什么时候使用分支？
 - 增加新功能
 - 实验性修改
 - 多人协作开发
 - Bug修复
- 除了分支，还有那些引用？
 - tag

分支使用初体验

- 创建分支
 - 法一: `git branch new_branch [start_point]`
 - 法二: `git checkout -b new_branch`
 - 在某分支上创建新分支:
 - `git checkout -b new_branch old_branch`
- 查看分支
 - 查看分支: `git branch [-a] [--merged] [--no-merged]`
- 切换分支
 - 直接分支检出: `git checkout branch`
- 分支检出很危险, 注意!
 - 分支检出其实看作是提交的逆操作
 - 检出之前一定要保证工作区和暂存区的清洁

分支使用初体验

- 分支重命名

- 对于未冲突的分支名

- `git branch -m oldbranch newbranch`

- 对于已经存在的分支

- `git branch -M oldbranch newbranch`

- 删除分支

- 对于已经合并的分支，直接删除：

- `git branch -d branch`

- 对于未合并的分支，强制删除：

- `git branch -D branch-name`

分支合并(1)

- 直接合并
 - 快进提交
 - 合并提交
 - 合并test分支到**当前**分支
 - `git merge test`
- 冲突解决
 - 手工解决
 - 图形化工具
 - `git config --global diff.tool vimdiff`
 - `git config --global difftool.prompt false`
 - 分支合并产生冲突后, 使用`git mergetool`解决冲突
 - 使用`:xa`退出, Git会依次打开下一个冲突文件
 - 解决完冲突后, 提交, 这时候会生成一个新的commit

分支合并(2)

- 压合合并

- 将一个分支上的所有历史合并为一个提交，然后合并到另一个分支上。
- 一般bug或新功能分支都可以使用这种方式合并
 - `git merge --squash test`

分支合并 (3)

- 挑选合并

- 挑选一个提交，添加到当前分支末梢

- `Git cherry-pick SHA`

- 挑选多个提交，添加到当前分支末梢

- `Git cherry-pick -n SHA`

- `Git commit`

- 这里您会看到合并并更新到版本库，而是在暂存区保存，可以继续cherry-pick，最后一次性提交即可

分支衍合

- 分支衍合
 - 让提交历史更加清晰
- 注意事项
 - 衍合会更改提交历史
 - 永远不要衍合已经push到公共仓库的更新

衍合与合并的区别

- 合并顺序
 - 按照提交时间依次合并
- 应用场景
 - 新功能开发
 - Bug修复
- 衍合顺序
 - 将其它分支提交线性合并到本分支
- 应用场景
 - pull/push冲突解决
 - 基于主分支开发

分支修改储藏

- 保存当前分支数据
 - 在当前分支工作被打断，进度不适合提交
 - 保存当前分支工作目录和暂存区的数据
 - `git stash`
 - 切换到别的分支做其它工作
- 恢复当前分支数据
 - 别的分支完成后，再切换到被打断的分支
 - 恢复工作区和暂存区的数据
 - `git stash apply (pop)`
 - 继续工作
- 储藏队列
 - 多次使用stash命令
 - 查看储藏队列：`git stash list`
 - 清空储藏队列：`git stash clear`
 - 恢复某次储藏：`git stash apply stash@{2}`

恢复已删除分支的提交

- 分支删除的本质
 - 分支这个指针已经删除，但commit对象还存在
 - 只是脱离了commit链表而已，变成了悬空对象
- 具体步骤
 - 找回提交：`git fsck --lost-found`
 - 查看修改：`git show SHA`
 - 衍合提交：`git rebase SHA`
 - 合并提交：`git merge SHA`
- 注意事项
 - 提交可以找回来，但是分支找不回来
 - Commit删除并不是真正的删除

一个脱离了组织的提交：no branch

- 造成no branch的原因
 - 与远程存在冲突，push/pull后可能会切换到此状态
 - 其实是指在某个commit上，可以看作是一个匿名的branch
- 解决方法：
 - 如果改动较小
 - 看一下提交ID：`git log`
 - 切换到要合并的分支上master： `git checkout master`
 - 直接将该提交ID合并到当前分支： `git merge SHA`
 - 如果改动较大
 - 先建一个新分支： `git checkout -b temp SHA1`
 - 再对该分支进行提交或合并处理： `git merge temp`
 - 如果你已经不在 no branch上，切换不到no branch上了
 - 找回丢失的commit_ish： `git reflog | git fsck -lost-found`
 - 检出提交并备份到新分支上： `git checkout SHA; git checkout -b tmp ;`
 - 或者直接合并到master分支： `git checkout master; git merge SHA`
 - 此时的SHA可以看作一个匿名的branch

第8章 远程仓库

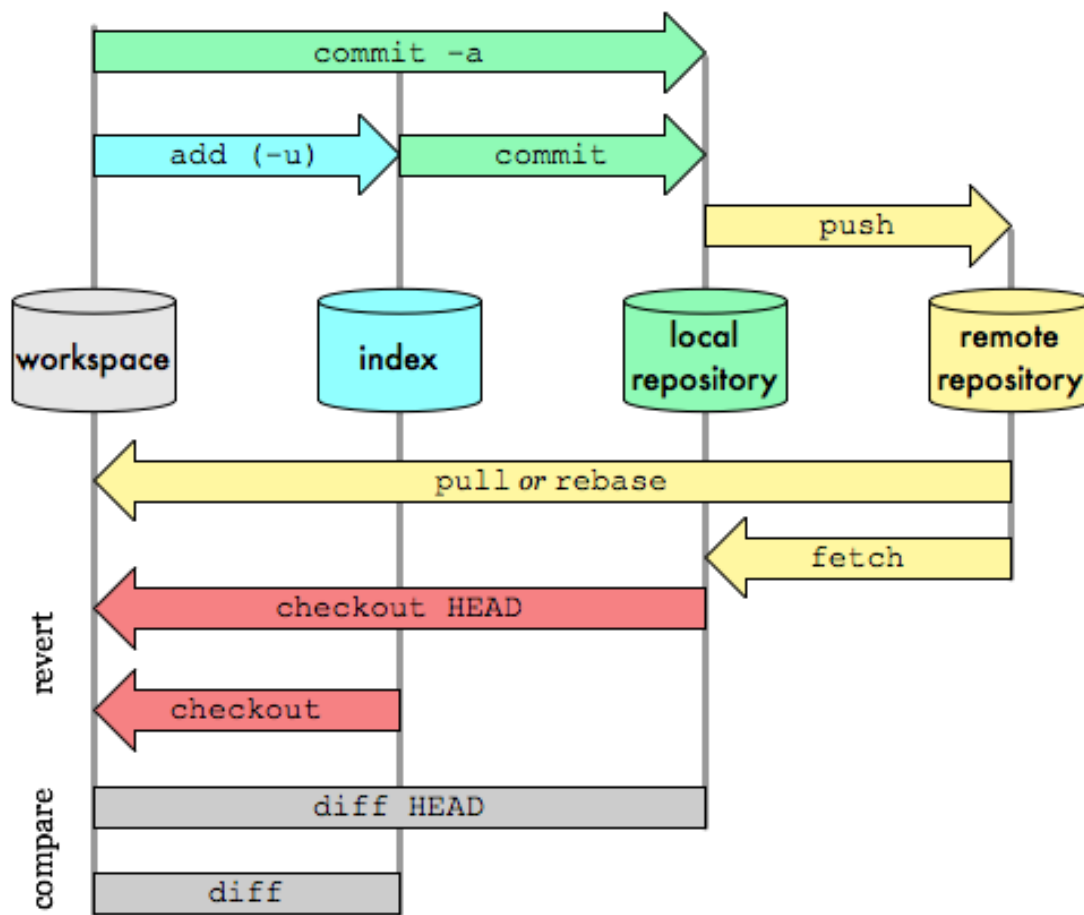
远程仓库

- 远程仓库的概念
- 使用Github代码托管
- 远程仓库基本操作
- 远程分支
- 添加新的远程版本库

远程仓库的概念

Git Data Transport Commands

<http://osteele.com>



远程仓库的概念 (2)

- 协议

- SSH协议

- 同时支持读和写的网络协议，Git默认使用的网络协议
 - 优点：安全性高，数据传输都是加密和授权的
 - 缺点：不能匿名访问，读也要授权，不利于开源的项目

- Git协议

- Git自带的网络协议：适用于不需要对读进行授权的大型项目
 - 优点：传输速度最快
 - 缺点：授权机制不灵活，要么不能推送，要么都能推送

- HTTP/HTTPS协议

- 优点：容易架设，适用于提供只读的仓库、防火墙穿透强
 - 缺点：传输速度慢、开销大

远程仓库的概念 (3)

- SSH key
 - 服务器的一种授权访问机制



- 类似的..
 - 天王盖地虎 \leftrightarrow 宝塔镇河妖
 - 天王盖地虎 \leftrightarrow 小鸡炖蘑菇
 - 地振高冈，一派溪山千古秀 \leftrightarrow 门朝大海，三河合水万年流
 - 地振高冈，一派溪山千古秀 \leftrightarrow 门朝大海，三合河水万年流

远程仓库的概念(4)

- SSH公钥生成

- 先看用户目录下有没有：`~/.ssh`
- 没有的话，使用命令生成：
 - `ssh-keygen -t rsa -C "wit.wang@qq.com"`
- 私钥保存在用户目录下，SSH连接服务器时使用
- 公钥放到服务器，服务器不同配置管理不一样
 - 在服务器上添加一个git用户
 - 将公钥加入到`~/.ssh/authorized_keys`文件

- 多台电脑提交

- 分别在自己的多台电脑上生成多对密钥
- 然后将公钥分别添加到服务器上即可

使用Github代码托管

- 注册、创建仓库
- 添加SSH-key
 - 验证key是否添加成功
 - `ssh -T git@github.com`
- 克隆仓库到本地
 - `git@github.com:wangwit/git_lesson.git`
 - `ssh://git@github.com:wangwit/git_lesson.git`
 - `https://github.com/wangwit/git_lesson.git`
 - `git://github.com/koke/grit.git`
- 注意
 - 一个账户可以添加多个pub key，一个pub key只能添加到一个账户上

远程仓库基本操作

- 从远程仓库克隆 `git clone repo_addr`
- 添加远程仓库
 - `Git remote add origin git@github.com:wangwit/test.git`
- 从远程仓库拉取数据
 - 自动本地master分支并跟踪远程仓库master分支: `git clone origin`
 - 仅仅是拉取, 不合并到本地: `git fetch`
 - 自动拉取某个分支更新: `git pull`
- 推送数据到远程仓库
 - 默认是origin和master分支: `git push origin master`
- 查看远程仓库信息
 - 参数: `git remote -v`
 - `Git remote show origin`
- 远程仓库的删除
 - `Git remote rm respority`
- 远程仓库重命名
 - `Git remote rename old new`

远程分支

- 远程分支
 - 是对远程仓库状态的索引，用origin/master表示
- 远程分支创建
 - 将本地新建分支直接推送到远程仓库
 - `git push origin mybranch:mybranch2`
 - 远程仓库没有这个分支，将创建这个分支origin/mybranch2
 - 并将本地分支和远程分支建立关联，以后可以直接push
- 跟踪远程分支
 - 基于远程分支建立一个本地关联分支
 - `git checkout -b local_branch origin/mybranch2`
 - `git push local_branch` 建立关联后，就可以直接进行pull和push
- 删除远程分支
 - 命令：`git push origin :mybranch2` (远程分支)
 - 远程分支origin/mybranch2就会嗖的一声不见了

远程分支

- 指定分支提交

- `git push origin mybranch:master`

远程分支与本地分支区别

- Master VS origin/master

添加新的远程版本库

- 添加新的远程版本库
 - 其实就是给远程版本库添加一个别名
 - `Git remote add origin
git@github.com:test/test.git`

第9章 标签管理

标签管理

- 标签的基本概念
- 标签的基本操作
- 标签到远程操作

标签的基本概念

- 什么是标签

- 标签是一个引用，标签对象可以指向任何对象
- 通常情况下，是指向一个提交(commit)
- 包括一个标签、一组数据、一个消息、一个指针

- 标签有什么用

- 使用标签可以方便地标记里程碑
- 发布软件的版本标识(内核Linux2.6.30)

标签的基本操作

- 打标签

- 在当前分支的末端创建轻量级标签: `git tag V1.0`
- 给指定某个commit打标签: `git tag v1 SHA`
- 创建重量级标签:
 - `git tag -a v1.0 -m "version1.0" SHA`

- 查看标签

- 查看所有标签: `git show v1.0`

- 使用标签检出其标记的版本库状态

- 检出: `git checkout v1.0`
- 标签指向一个commit, 跟branch类似, 也是一个引用

- 删除标签

- 删除: `git tag -d v1.0`

标签的远程操作

- 推送本地标签到远程(发布版本)
 - 推送某个标签:
 - `git push origin v1.0`
 - 推送所有标签:
 - `git push origin --tags`
- 删除远程标签
 - 先删除本地标签:
 - `git tag -d v1.0`
 - 再删除远程标签:
 - `git push origin :refs/tags/v1.0`

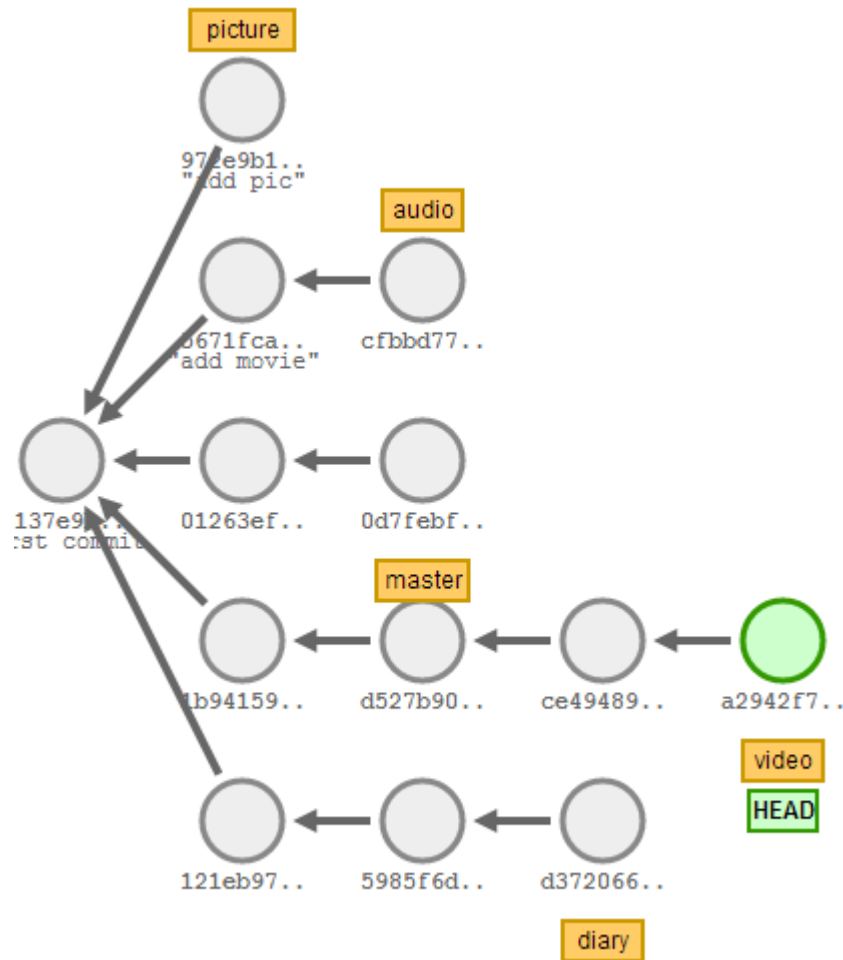
第10章 Git实战

Git实战

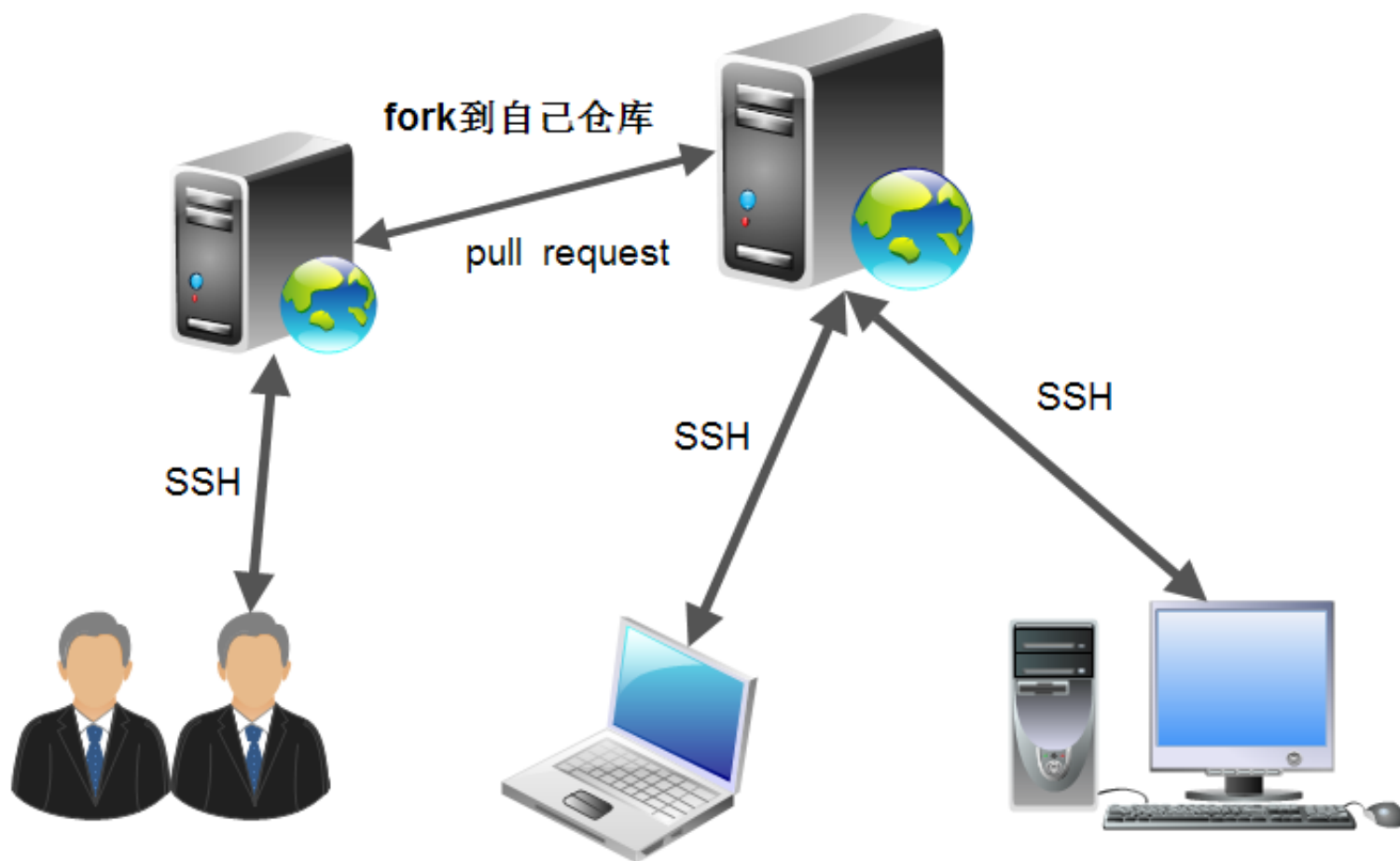
- 管理文件，保护隐私
- 在github上发起一个公开项目
- 在OSC上发起一个私人项目
- 在本地搭建git服务器
- Git+repo+gerrit企业环境介绍

Git实战(1)

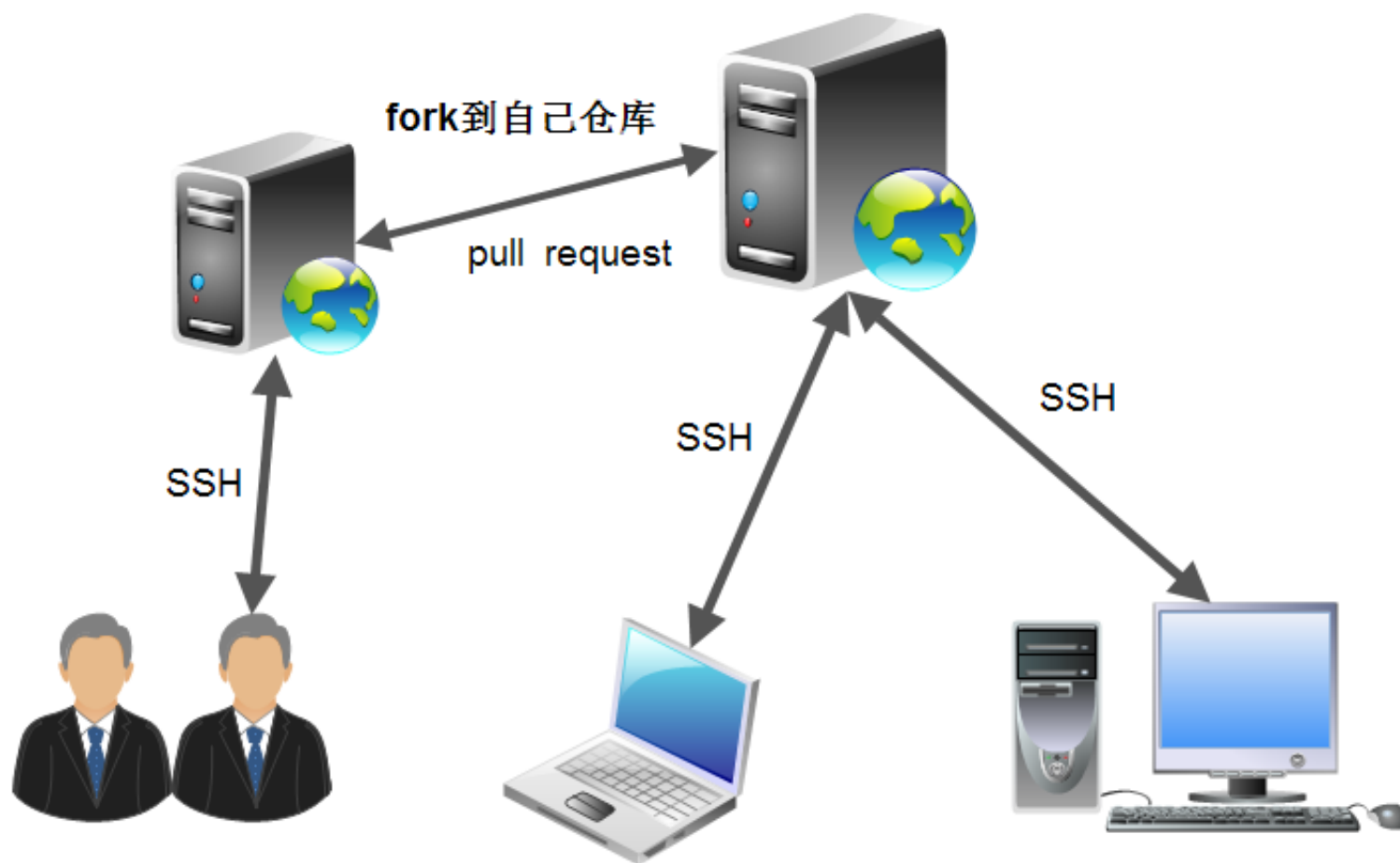
管理本地文件，保护隐私



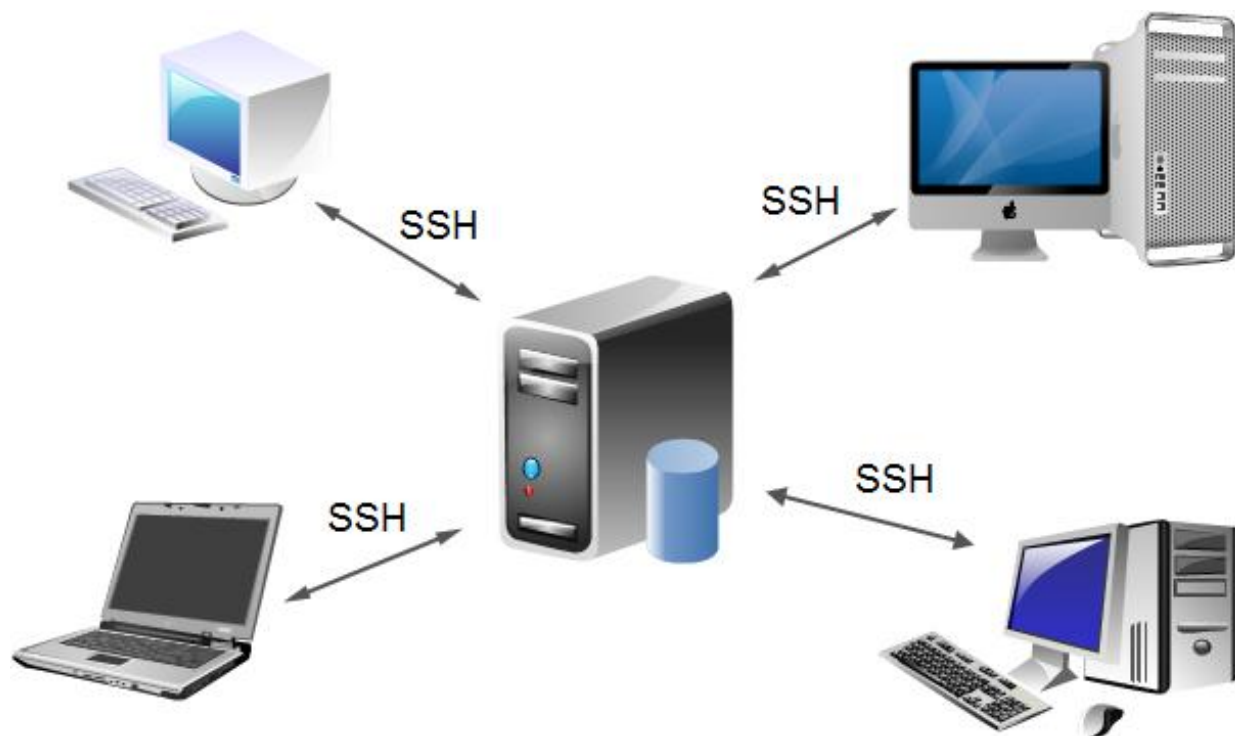
发起一个公开项目



发起一个私人项目



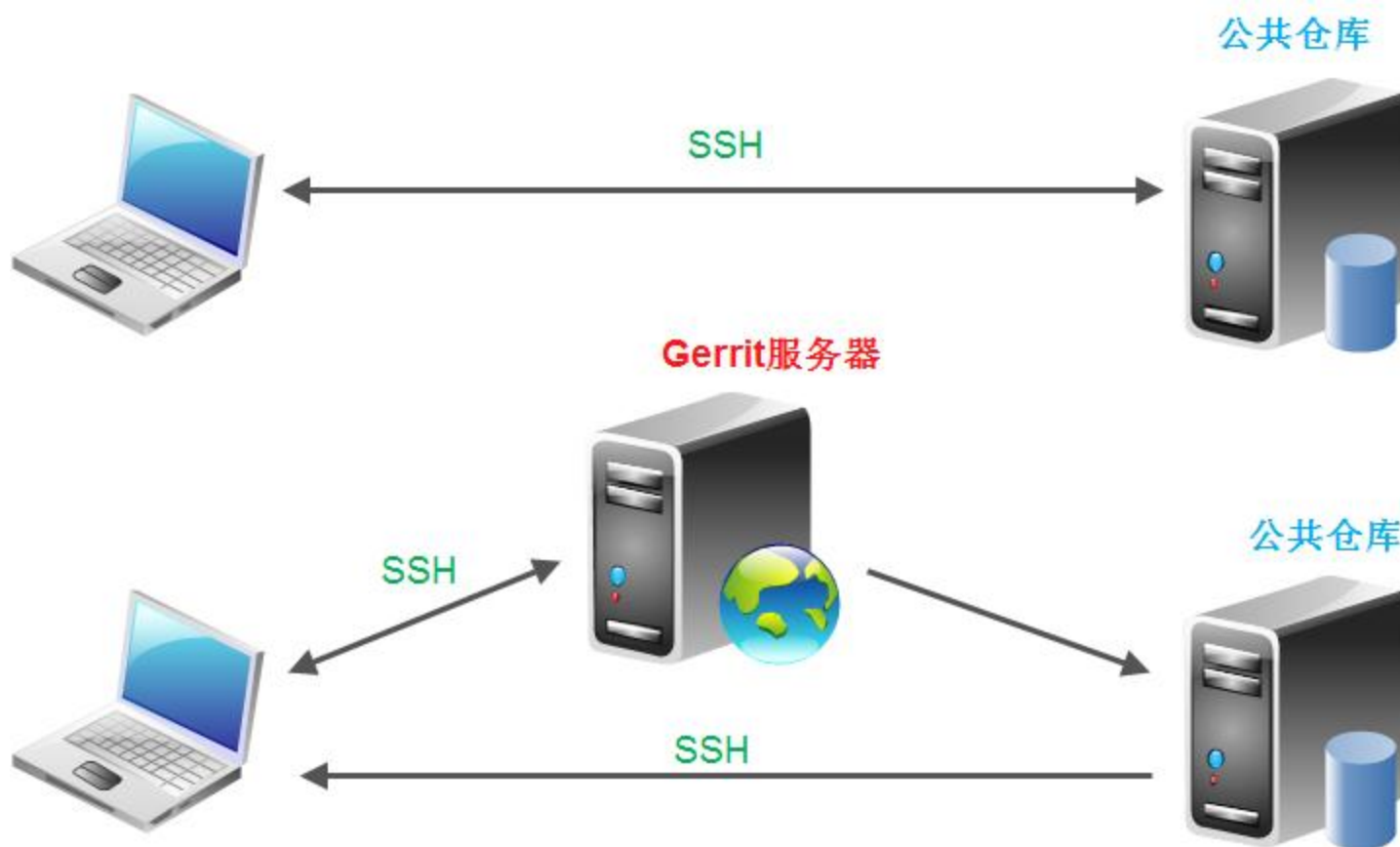
搭建本地Git服务器



更高级的应用

- 更高级的权限管理
 - Gitsis、Gitolite、
- 公网服务器
- 更多…
 - gitlab
 - Gitcaf
 - Coding

Git+repo+gerrit环境介绍



代码审核流程

- **第一步：Review**
 - 资深工程师+2，一般工程师+1
 - 总分+2才能评审通过
- **第二步：Verify**
 - 手工编译、专门测试人员测试
 - 自动化编译、自动化测试
 - Verify+1：才能通过
- **第三步：Merge**
 - 把通过审核的修改合并到公共仓库中

谢谢

- 更多教程，请关注：
- 视频教程：
 - 51CTO学院
 - CSDN学院
 - 搜” **GIT零基础实战**”

