用 Git 进行版本控制

李博杰 2012-05-20
bojieli@gmail.com

版本控制的史前时代

- 用存储介质拷贝代码
  - 代码相互覆盖，不知道哪个版本是正确的
  - 搞错了无法恢复，需要定期手工备份
- diff & patch
  - 1991~2002 ， Linux 内核
  - 能看到文件之间的差异，知道哪里修改了
  - 更改历史需要手工维护
  - GNU diff 不支持二进制文件

版本控制的诞生

- 将 diff 和 patch 的过程自动化
  - 单一文件版本管理工具 RCS
  - 只保留一个版本的完全拷贝，其他历次更改仅保留差异： V3=V1+△1+△2
- CVS ：脚本实现的 RCS 文件容器 (1985, 1986 publish, 1989 rewriten in C)
  - 版本库中任意一个目录拿出来是一个新的版本库
  - Commit log, checkin, checkout, tag, branch
  - 文件的版本号相互独立，全局版本号只能不停地打 tag

CVS 原理示意

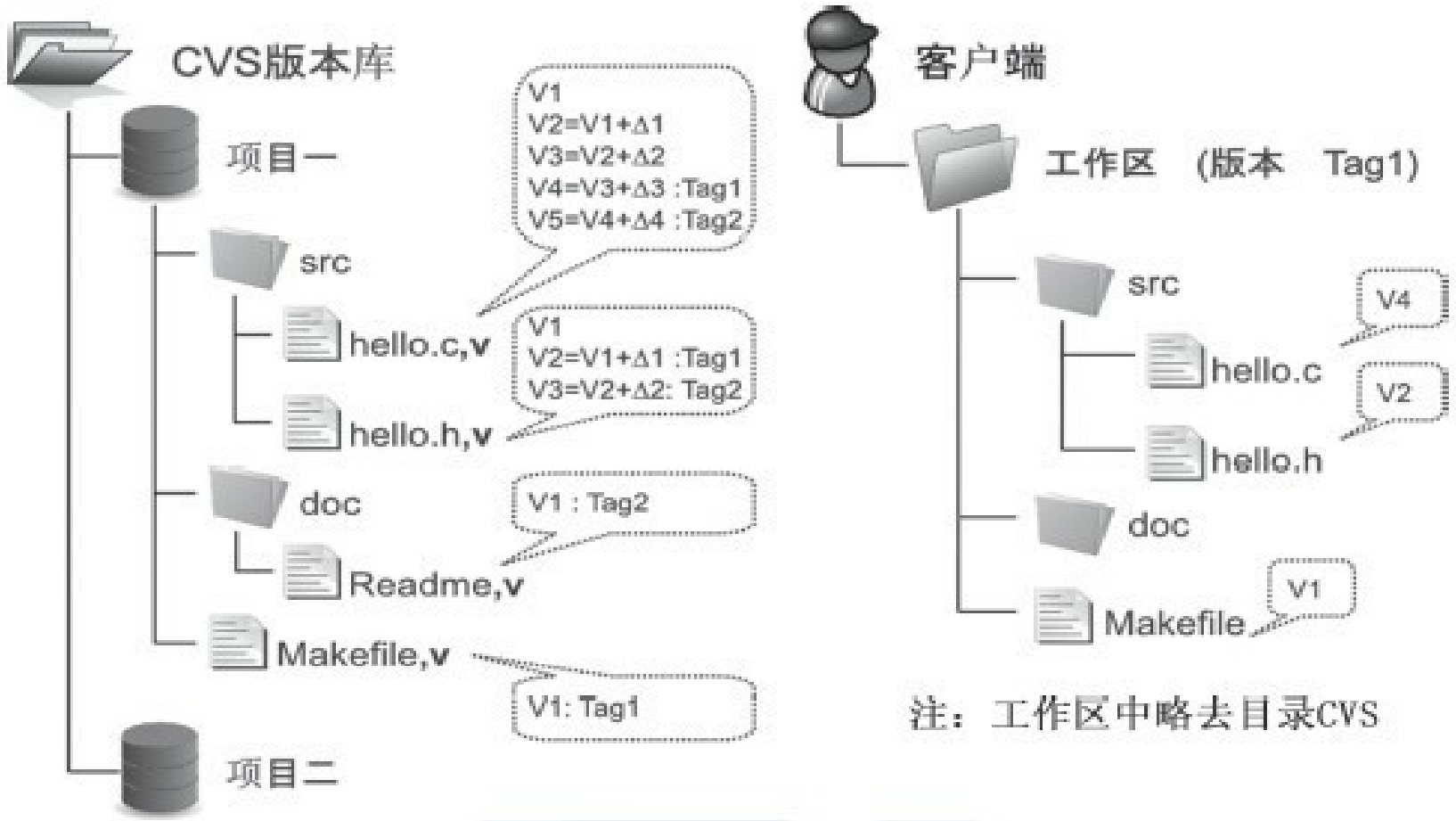


图 1-1 CVS 版本控制系统示意图

SVN

- 索引的是二维信息：文件、版本
- CVS ：工作区中的每个文件对应版本库中的一个文件
- SVN ：每个版本的一个差异文件，一个信息文件
  - 以顺序数字编号命名
  - db/revs 下的：与上一提交的差异
  - db/revprogs 下的：提交日志、作者、提交时间
- SVN 实现了全局版本号、原子提交、跟踪重命名

SVN 原理示意

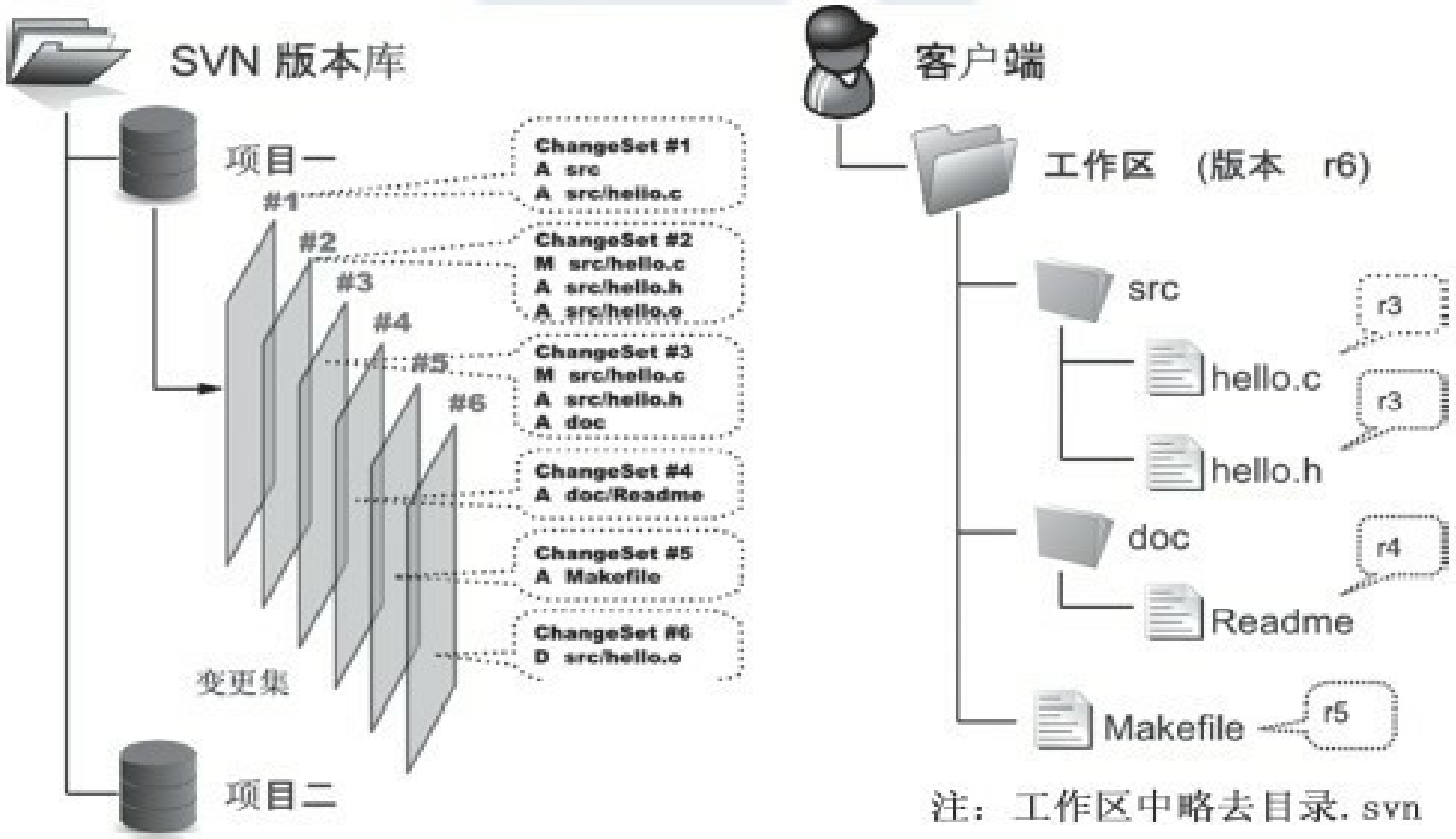


图 1-2 SVN 版本控制系统示意图

Linus 为何迟迟不用版本控制

- 集中式版本控制系统
  - 版本库存储在服务器端，每次提交、查看日志能操作都要与服务器连接
  - 代码的稳定性高度依赖中心服务器
  - 历史不容篡改，无法做试验性提交，“一失足成千古恨”
- Linus 认为这不符合开源项目的精神，因此直到 2002 年才使用商业版本控制系统 BitKeeper 管理 Linux 代码

分布式版本控制系统

- 每个人拥有一个完整的版本库
- 查看日志、提交、创建 tag 和分支等操作可以在本地完成，不再时刻需要网络连接
- 在推送到远程版本库前可以做很多试验性的提交，反复悔改而不必担心干扰其他人
- 多样的协同工作模型使开源项目的参与度爆发式增长

Git 是逼出来的

- 2005 年 4 月 ， Andrew Tridgell 试图对 BitKeeper 进行反向工程，以开发一个能与之交互的开源工具
- BitMover 公司要求收回对 Linux 社区的免费授权
- Linus 只好“自力更生”：
  - 2005-04-03 ，开始开发
  - 2005-04-06 ，项目发布
  - 2005-04-07 ， Git 作为自身的版本控制工具
  - 此时 git 代码只有 1244 行，只有一些底层命令

Git 是逼出来的 (cont'd)

- 2005-04-18 ，第一次分支合并
- 2005-04-29 ， Git 的性能达到 Linus 的预期
- 2005-06-16 ， Linux 2.6.12 开始采用 Git
- 2005-07-26 ， Linus 功成身退，将 Git 的维护交给另一位主要贡献者 Junio C Hamano

- 最初的 Git 除了一些核心命令，都用脚本语言写成（现在为了效率大多用 c 语言重写）
- Android, Debian, Eclipse, Git, Gnome, KDE, Linux kernel, Perl, PHP, PostgreSQL, Qt, Ruby on Rails, X.org...

Git config

- git 是分布式版本控制系统，不存在“验证”用户名、密码的中央版本库
- git 根据提交者设置的 name 和 email 记录更改是谁做出的，因此安装 git 后的第一件事就是设置个人信息：
  - git config --global user.email “bojieli@gmail.com”
  - git config --global user.name “boj”
  - 不要逐字照抄，不然功劳就算到我头上了 ~

Git 初始化

- 初始化版本库： git init
- 得到隐藏的 .git ，版本库信息都存储在 .git 中
- 版本库在服务器端？
- 版本库在本地？
  - 集中存储在全局的“数据库”中？
  - 在工作区每个目录中？（CVS ， Subversion）
  - 在工作区根目录下？（git）

首次提交

- git init 只是建立了一个空的版本库
- 要将已有的内容纳入版本控制，需要
- 将当前目录中的所有文件添加到索引
  - git add .
- 然后提交到版本库。
  - git commit -m “initial commit”
- 查看提交历史是否正常
  - git log

git commit

- git commit 的输出信息
  - 此次提交是在 master （主分支）上的，是该分支上的根提交（首个提交），提交 ID 为……
  - 此次提交修改了 11 个文件，增加了 1244 行
  - 创建了若干个文件，其权限均为 644
- Git 规定提交必须输入提交说明
  - git commit -m “commit message”
  - 或 git commit ，进入编辑器编辑提交说明（至少学会一种编辑器吧，不然保存退出都成问题）

Git config 的配置哪里去了

git config --system	/etc/gitconfig	系统
git config --global	~/.gitconfig	当前用户
git config	.git/config	当前版本库

- git config 默认只针对当前版本库
  - 当前版本库是如何确定的？
  - 没有全局“数据库”，只能找 .git 哦
- Git 配置文件使用 ini 文件格式
  - 读： git config <section>.<key>
  - 写： git config <section>.<key> <value>

Git config 试验

- 如果不设置 user, email 会怎么样？
  - git config --unset --global user.name
  - git config --unset --global user.email
  - git commit --allow-empty -m “anonymous commit”（加参数以允许空提交）
- 如果设置了当前版本库的 user, email 会怎么样？
  - git config user.email “bojieli@gmail.com”
  - git commit --amend --allow-empty --reset-author
    - amend ：重新进行最近一次提交
    - reset-author ：默认不更新提交日志中的 Author

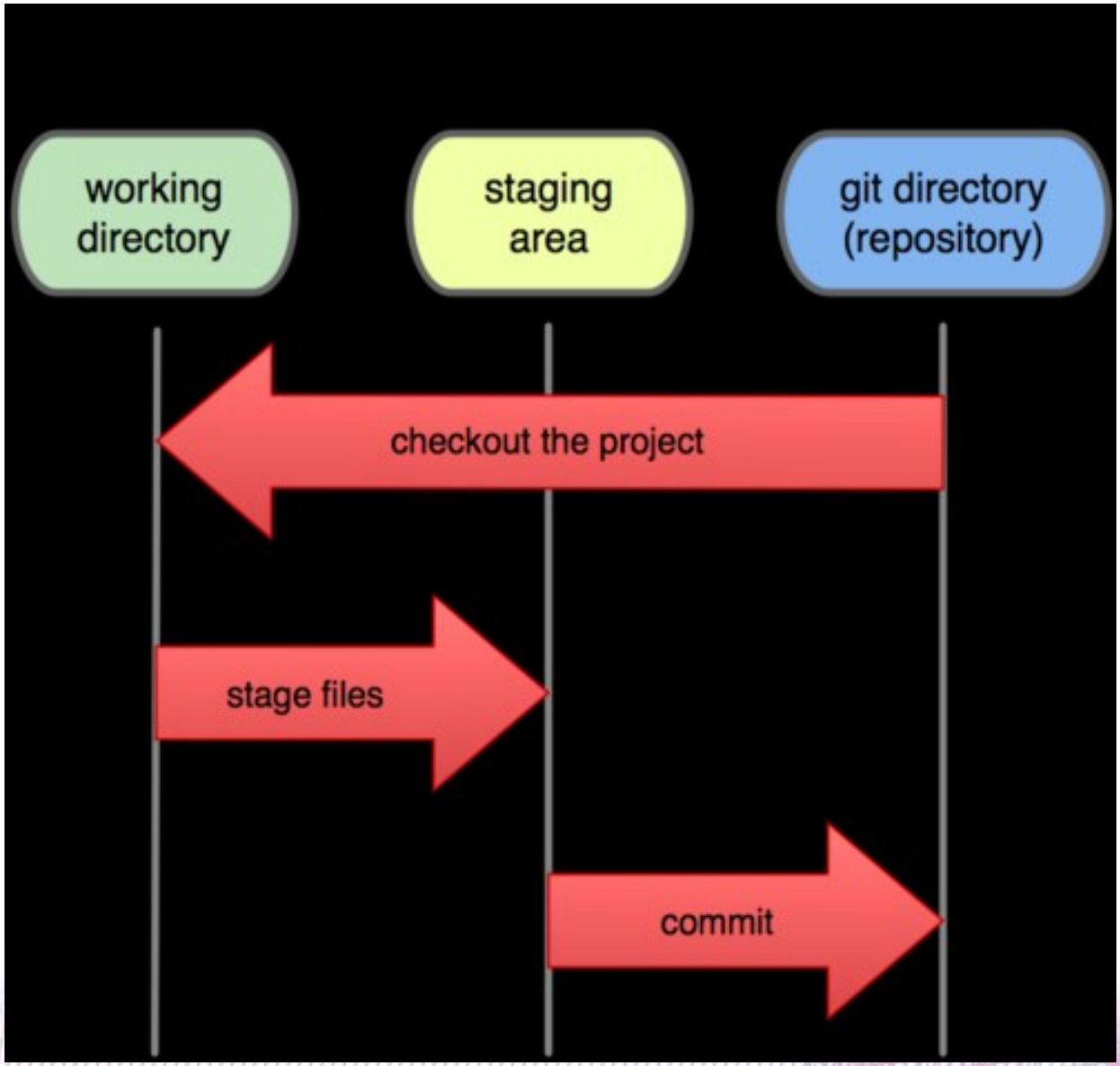
git add 与暂存区

- 首先对版本库做一些修改
- 用 git diff 查看有哪些改动
  - 输出就是 GNU diff 的样式
  - 支持二进制文件哦
- git commit -m “unstaged commit” ，成功了吗？
  - git log 没有增加新的提交日志
  - git status 有点似曾相识的感觉
  - git diff 与 commit 前相比没有变化

git add 与暂存区 (cont'd)

- Changes not staged for commit?
- 只有暂存的修改才会被提交。
  - 如果同时在修改两个不同功能模块，一起 commit 不如把两个模块的修改分开 commit
  - 如果正在开发的两个模块一个已经编好，另一个尚未完成，如何提交阶段性成果？
- git 的“暂存区”设计赋予用户对提交内容进行控制的能力，使得“按需提交”成为可能。
- 工作区 (working dir) => 暂存区 (index) => 版本库 (repo)

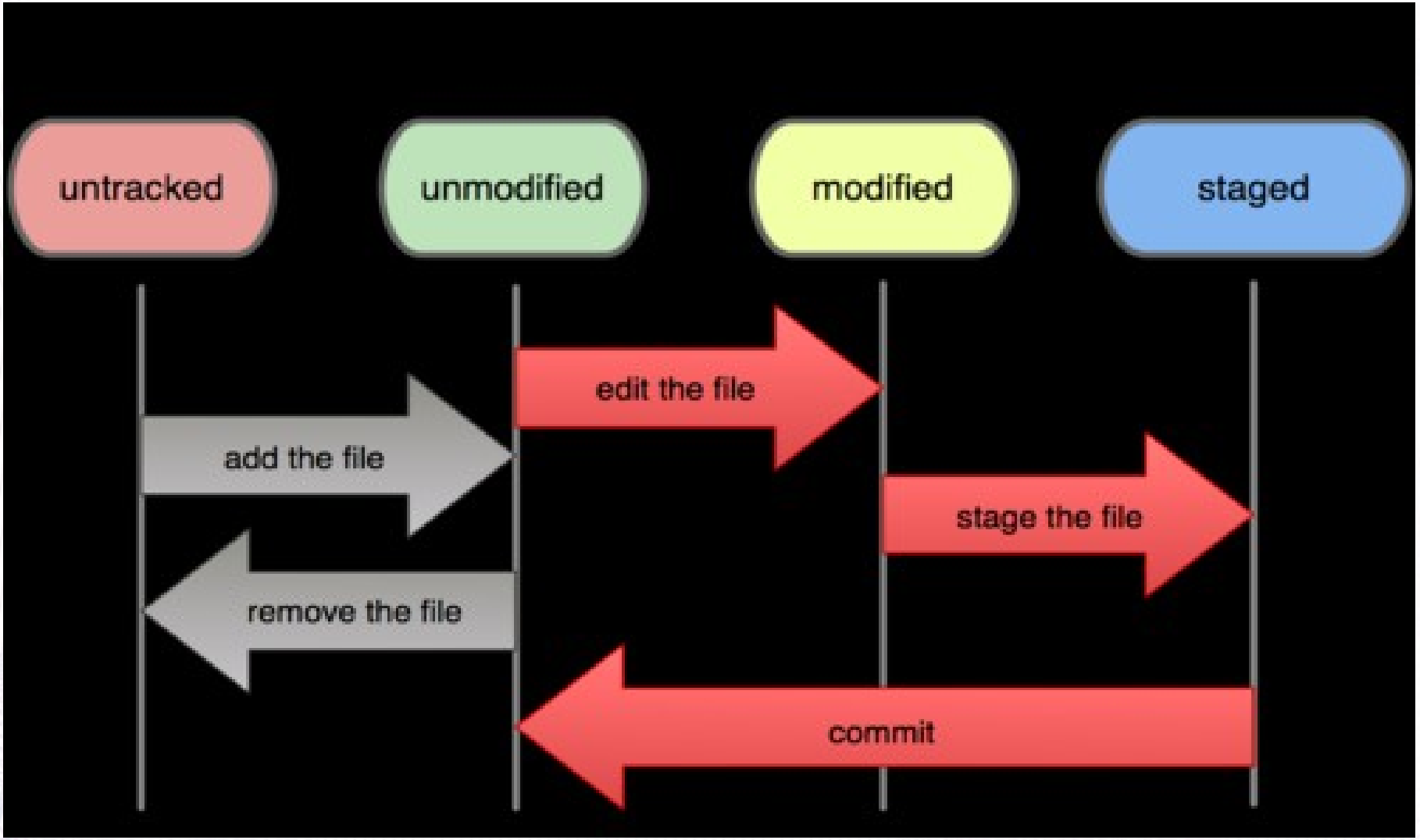
git add 与暂存区 (cont'd)



查看状态： git status

- 工作目录中的每个文件可能处于四种状态之一：
  - untracked ：未被跟踪，新添加的文件不会被自动跟踪哦 ~
  - unmodified ：已被跟踪，但未被修改
  - modified ：已被修改，但尚未被暂存
  - staged ：修改已被暂存
- git add 就是把文件或目录加入暂存区的方法

文件的四种状态及转化


untracked
unmodified
modified
staged
edit the file
add the file
stage the file
remove the file
commit

查看状态： git status (cont'd)

- git add 的几种偷懒方法
  - git add . 将当前目录下的所有文件变更（包括新文件）放入暂存区
  - git add -u 将当前目录下被版本库跟踪的所有文件变更（不包括新文件）放入暂存区
  - git add -A 将当前目录下所有文件变更（包括新文件）放入暂存区，并查找重命名情况
  - git add -i 使用交互式界面选择需要添加的文件

查看状态： git status (cont'd)

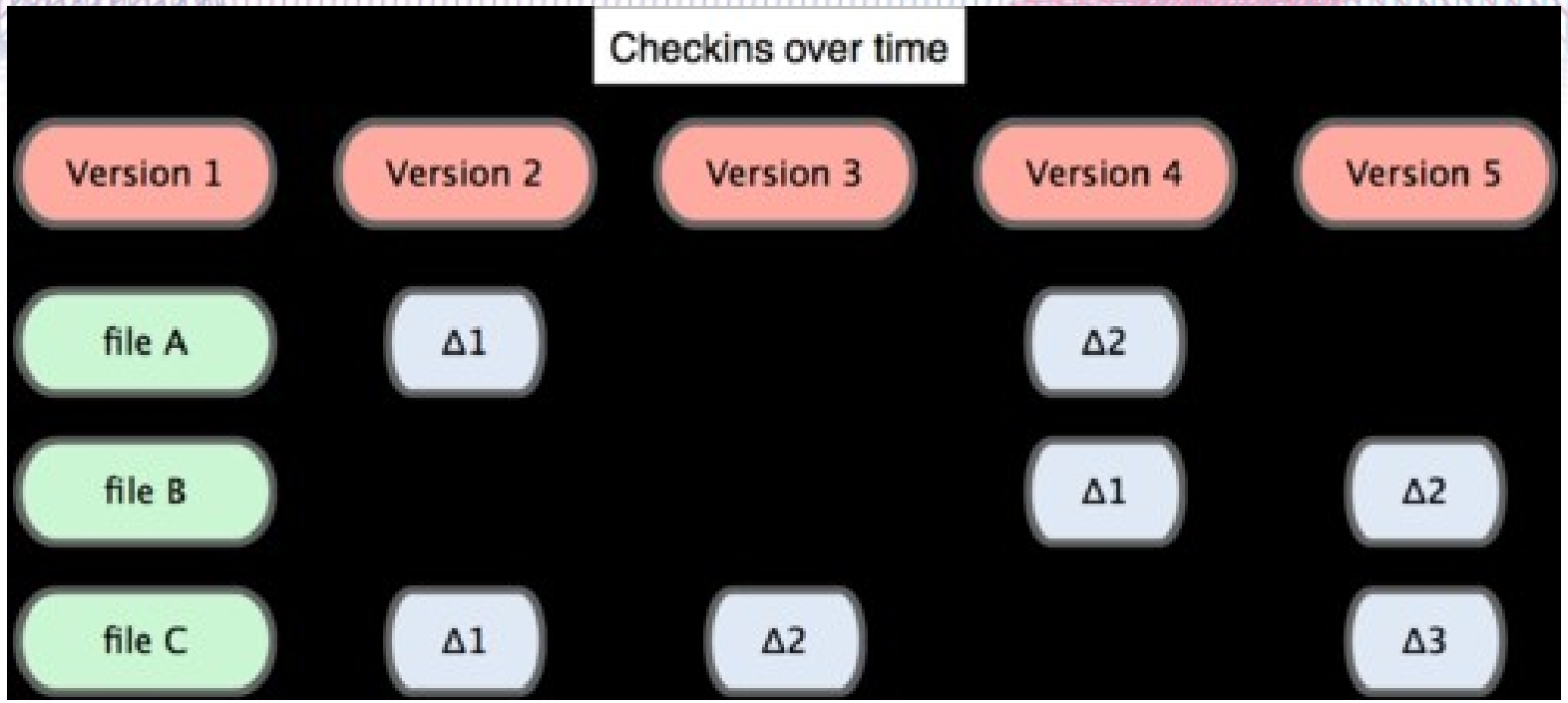
- 每次提交前 add 太麻烦？
  - git commit -a = git add -u + git commit
- Git 可以跟踪文件的移动
  - git mv ，直接将文件移动写入暂存区
  - 使用 mv 移动，只要使用 git add -A 添加， git 也能检测到文件移动
- git rm 可以将文件的删除写入暂存区

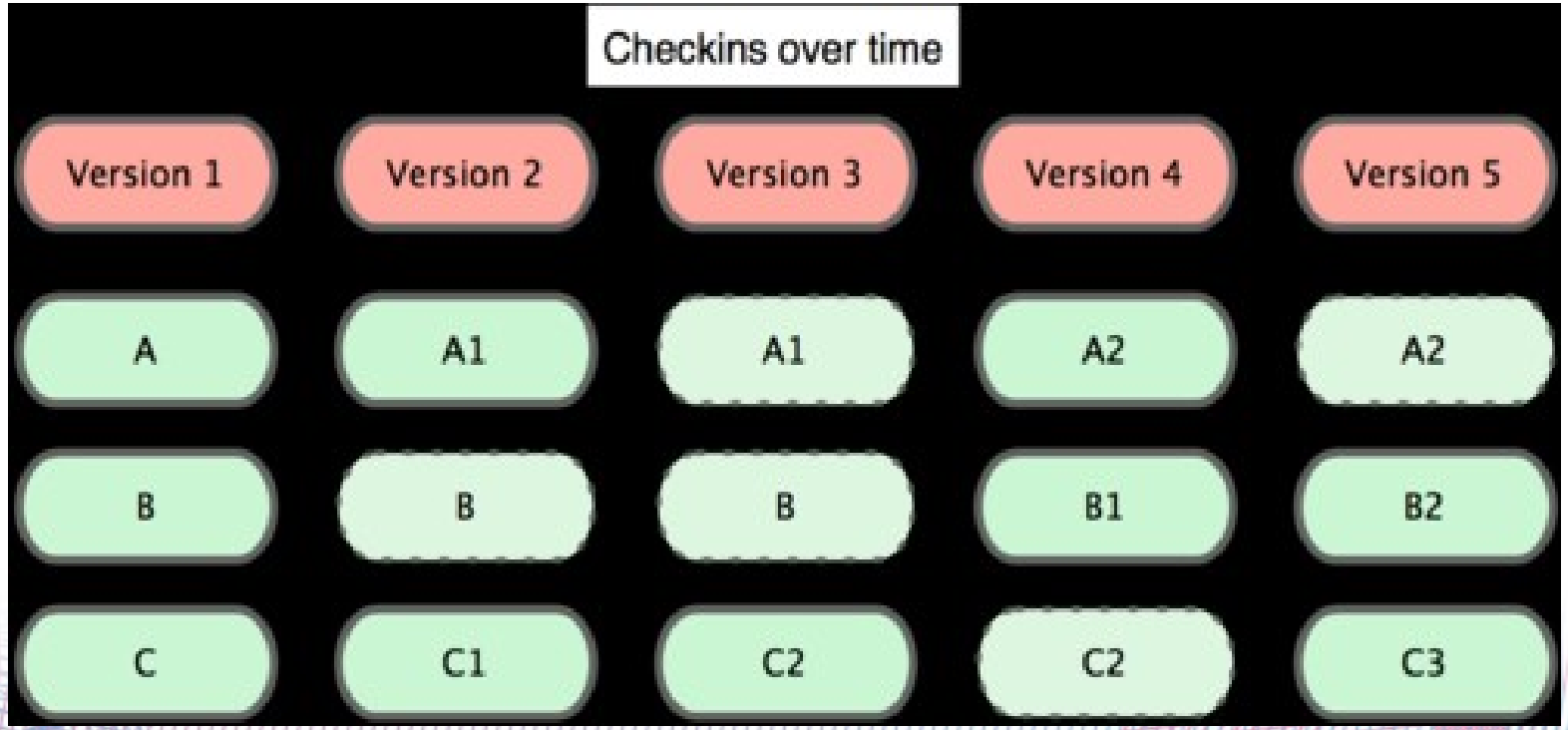
查看状态： git status (cont'd)

- git status 扫描工作区改动时
  - 根据 .git/index 记录的时间戳、长度等信息判断工作区文件是否被改变
  - 如果时间戳改变了，需要读取文件内容，计算其 SHA1 hash 值，与版本库中的相比较；如果文件内容没有改变，则更新 .git/index 时间戳
- .git/index 是包含文件索引的目录树，记录文件名、时间戳、文件长度等。文件内容存储于 git 对象库 .git/objects 中，文件索引建立了文件与对象库中文件内容间的对应。

查看状态： git status (cont'd)

- git status
  - 扫描暂存区相对工作区的改动（只看索引）
  - 扫描暂存区相对版本库的改动
  - 扫描未被版本库跟踪的文件（扫描目录）
- 修改一个文件，添加到暂存区；再修改它一下，执行 git status 。根据前面的原理，你能想到会发生什么吗？
- 暂存区在 .git/index ，版本库在哪里？


Checkins over time
Version 1
Version 2
Version 3
Version 4
Version 5
file A
Δ1
Δ2
file B
Δ1
Δ2
file C
Δ1
Δ2
Δ3


Checkins over time
Version 1
Version 2
Version 3
Version 4
Version 5
A
A1
A1
A2
A2
B
B
B
B1
B2
C
C1
C2
C2
C3

走进版本库的老巢

- .git/HEAD: “ref: refs/heads/master”
- .git/refs/heads/master: 一个提交 ID
- 对象存储在 .git/objects/ID 的前两位 /ID 的后 38 位
- 强大的 git cat-file ：
  - git cat-file -t 3eb72cf (commit)
  - git cat-file -p 3eb72cf (tree, parent)
- tree 对象：本次提交的文件列表（文件名、属性、内容所在对象）

走进版本库的老巢 (cont'd)

- blob 对象存储的就是文件内容
  - 由于对象是按照 hash 值存储的，相同内容的文件只会出现一个副本。
- commit 对象中的 parent 指向上一提交
  - Git 中每个提交的 tree 是对于上一提交的增量
  - 通过 commit 对象间的 parent 关联，可以识别出一条跟踪链： git log --pretty=raw --graph

Git object hash

- Commit, tree, blob 对象的 40 位 ID 是如何生成的？
- 首先看 commit 的 ID 生成规则
- git cat-file -p HEAD 等于 git cat-file commit HEAD
- git rev-parse HEAD
- (echo -n "commit "; echo -n `git cat-file -p HEAD | wc -c`; printf "\000"; git cat-file -p HEAD) | sha1sum

Git object hash (cont'd)

- 文件内容 blob 对象
- git rev-parse HEAD:Makefile
- (echo -n "blob "; echo -n `git cat-file -p HEAD:Makefile | wc -c`; printf "\000"; git cat-file -p HEAD:Makefile) | sha1sum
- tree 对象
- git cat-file -p HEAD^{tree} 不等于 git cat-file tree HEAD^{tree}
- git rev-parse HEAD^{tree}
- (echo -n "tree "; echo -n `git cat-file tree HEAD^{tree} | wc -c`; printf "\000"; git cat-file tree HEAD^{tree}) | sha1sum

Git object hash (cont'd)

- 如果提交采用顺序编号，在分布式版本控制系统中无法做到不同人的提交编号不同
- Git 通过 SHA1 密码学意义上的无重复特性使得不同内容、不同时间、不同作者的提交“全球唯一”
- 每个对象的 ID 与对象类型、长度、内容关联
- 一个提交的 ID 与其父提交关联，使得修改历史会产生连锁效应，谁也不能篡改历史

Hash collision

- 160 位的 SHA1 hash 冲突的概率微乎其微，因而 Git 并没有考虑 hash 冲突的问题
- Linus 在 Git 邮件列表中回应此问题时说：
- 在 commit 时， git 如果发现待添加的新对象 SHA1 值已经存在，则会认为这个对象已经存在，因而引起冲突的新文件不会被保存。用户执行 checkout 时才会发现得到的文件与所期望的不同。

Hash collision

- 因此，即使恶意者提交了引起冲突的文件，也不会改变历史。（构造 SHA1 冲突是很难的！）
- 如果真的在很偶然的情况下发生了冲突，则只要修改待添加的文件（如加个空行）即可避开。
- 如果提前获知一个 patch ，并构造冲突包抢先发给 Linus ，则能让真正的 patch 被“虚假”提交进去
  - 例如 commit message 在显示时被认为是 \0 结尾的，因此可以用 hash-object 手工制作 commit 对象，在 \0 后面藏一些好玩的东西

访问对象的方法

- “基址”
  - 在不引起冲突的情况下，只要把 SHA1 hash 值的前几位写出即可
  - master = refs/heads/master = heads/master
  - HEAD
- “偏移”
  - HEAD^, HEAD^^, HEAD^ ：父提交
  - HEAD^2 ：第二个父提交（有多个父提交时用）
  - HEAD~5 ：祖先提交

访问对象的方法

- 内容
  - 树对象： HEAD^{tree}
  - 文件对象： HEAD:Makefile
  - 暂存区中的文件对象： :Makefile
- 命令
  - git show ：查看提交或对象的详细信息
  - git rev-parse ：查询对象 ID
  - git cat-file ：查看对象内容和类型

研究 Git 常用命令

- Git cat-file
- Git rev-parse
- Git show
- Git ls-tree
- Git ls-files
- Git hash-object
- Git rev-list

分支游标

- cat .git/refs/heads/master
- Echo “Hello World” >> README
- Git commit -a -m “Does master follow this new commit?”
- cat .git/refs/heads/master
- refs/heads/master 就像一个游标，总是指向 master（主分支）上“最新”的提交。
  - refs/tags/tagxxx 指向 tag xxx （里程碑）
  - refs/heads/branchxxx 指向 branch xxx （分支）

git reset

- 有了游标，我们就可以在 git 历史中任意穿梭了
  - 1. 将 HEAD 游标指向新的提交 ID
  - 2. 用游标所指向的提交替换暂存区
  - 3. 用暂存区替换工作区
- git reset
  - git reset --soft <commit> (1)
  - git reset --hard <commit> (1,2,3) 危险！
  - git reset <commit> (1,2)
  - git reset --mixed <commit> (1,2)

git reset (cont'd)

- 不指定 commit ，默认为 HEAD
- git reset ：丢弃已暂存的更改
- git reset --hard 丢弃工作区和已暂存的更改（危险）
- git reset -- filename ： git add 的反向操作
- git commit --amend = git reset --soft HEAD^ + git commit -e -F
  - .git/COMMIT_EDITMSG 保存了上次的提交说明
- git reset --hard HEAD^ ：丢弃最近一次提交及此后的更改（危险）

最后一道防线： git reflog

- 上张 slides 中带 --hard 的都被标为“危险”，因为工作区没有备份，一旦覆盖永久丢失；那么版本库中被 reset 丢弃的提交还能找回来吗？
- .git/logs/HEAD, .git/logs/refs/heads/master
  - 游标的更改是有历史记录的，因此这些被“丢弃”的提交事实上记录在册
  - 不过不能高枕无忧，默认 90 天过期，过期后的更改记录会被清理掉
- git reflog ：查看游标历史

最后一道防线： git reflog (cont'd)

- 访问对象的方法又多了一种：
  - HEAD@{n} ： HEAD 游标的第 n 次变化
  - master@{n} ： master 分支游标的第 n 次变化
  - HEAD@{0} 就是刚刚进行的操作
  - git reset HEAD@{1} 就能把刚刚被 reset 的提交找回来啦
- 利用 git reset 对分支游标和工作区的强大控制能力，我们可以在历史中自由穿梭，甚至改变历史

破坏 git 的防线

- 不小心 commit 了一个大文件，无论如何 reset --hard 版本库也不会变小
  - 只要被引用，对象就会一直保留
  - 直接删除对应的 object ，版本库的一致性被破坏
  - git fsck --no-reflog ： dangling 的对象就是没有被引用的对象，随时可能被清理掉
  - git reflog expire --expire=now --all
  - git fsck
  - git prune

Git checkout

- git reset 虽然强大，但针对的都是 HEAD ，而 HEAD 指向的是 refs/heads/master ，这就意味着只能修改当前分支。那么 HEAD 本身该如何修改呢？
- git checkout <commit>
  - 将某个特定的提交检出：更新 HEAD 指向 <commit> ，用此提交更新暂存区和工作区
  - 危险，暂存区和工作区的未提交改动会被覆盖
- git checkout <commit> [--] <path>
  - 不更新 HEAD ，只覆盖 <path> 指定路径的文件

git checkout (cont'd)

- You are in 'detached HEAD' state?
- 分离头指针，就是 HEAD 头指针指向了一个具体的提交，如果再次执行 checkout ，就会覆盖掉这个 HEAD ，从而丢失这一串提交。
  - 事实上可以通过 reflog 这个神器找回来
- 推荐的方式是创建新的分支：
  - git checkout -b new_branch_name <commit>
  - 省略 commit ，则默认为 HEAD

Git stash

- 如果有一些未提交的改动，现在希望参考一下原来的版本，怎么办？
- git stash 可以保存当前进度，把工作区和暂存区尚未提交的改动照个快照保存起来，然后 reset --hard
- git stash list 查看保存的进度列表（栈）
  - 又一种访问对象的方法： stash@{n}
- git stash pop 把栈顶的进度“出栈”
- git stash apply 应用栈顶的进度，但不出栈

git stash 原理

- stash 之后，发现多了个 refs/stash ，内有提交 ID
  - 一个提交如何同时表示工作区和暂存区？
- 顺藤摸瓜，发现一个 commit ，它有两个 parent
  - git cat-file 不要这么快就忘了哦 ~
  - 这是一个合并提交，内容为工作区进度 （Work In Progress, WIP）
  - 一个 parent 是原来的 HEAD
  - 另一个 parent 是暂存区进度（它的 parent 是原来的 HEAD）

git checkout (cont'd)

- git checkout [--] <paths>
  - 用暂存区覆盖工作区
- git checkout <branch>
  - 切换到已有分支：更新 HEAD 指向 refs/heads/branch ，用分支的最新提交覆盖暂存区和工作区
- git checkout -b <new_branch> [<commit>]
  - 从 <commit> 创建新分支，覆盖暂存区和工作区
- git checkout 之前忘记 git stash ，欲哭无泪

git 中的时光穿梭机

- 如果说 git log 是版本库历史的一张平面图，那么 git 图形工具就是一张全息图，能更直观地展示各提交间的相互关系。
  - gitk （原生）
  - qgit （ QT ）
  - gitg （ GTK+ ）
- 在命令行下， git log --graph 也可以显示提交关系

查看指定范围的历史

- git log 中不仅可以指定一个提交，还可以指定提交范围。
- git log --oneline A ： A 的所有历史提交（一棵树）
- git log --oneline D F ：两棵树的并集
- git log --oneline ^G D ： ^ 是取反，即不包括树 G
- git log --oneline G..D ：与上面相同
- git log --oneline D..G ：与上面不同

查看指定范围的历史 (cont'd)

- git log --oneline B...C ：两个版本能够共同访问到的除外 = B C --not $(git merge-base --all B C)
- git log --oneline B^@ ：不包括自身的历史提交
- git log --oneline B^! ：只包括自身，不包括历史提交
- 使用 git rev-list 可列出匹配的提交 ID

定制 git log 的输出

- git log -3 ：显示最近的 3 条日志
- git log -p ：显示日志时同时显示 GNU diff 样式改动
  - 如： git log -p -1 head
- git log --stat ：显示日志时同时显示 diffstat 改动摘要
- git log --pretty=raw ：显示 commit 的原始数据
- git log --pretty=fuller ：同时显示 Author 和 Committer
- git show ：显示单个提交

git diff

- git diff B A ：比较两个 commit 或 tag
- git diff A ：比较工作区和 A
- git diff --cached A ：比较暂存区和 A
- git diff ：比较工作区和暂存区
- git diff --cached ：比较暂存区和 HEAD
- git diff HEAD ：比较工作区和 HEAD

git blame

- 查看这个文件的每行最早是在什么版本、由谁引入的，以便定位引起 bug 的版本和开发者
- git blame <filename>
- 只查看某几行：（6,10 中间不能有空格）
  - git blame -L 6,10 README
  - git blame -L 6,+5 README

多步悔棋

- 前面提到修补最近提交使用 git commit --amend ，根据其原理，“多步悔棋”也不难实现。
- 例如，我们希望把过去的多个提交压缩成一个提交，隐藏反复试验的过程，使版本库更干净：
  - git reset --soft HEAD^^
  - git status （看看现在成什么样了）
  - git commit -m “finish the new feature”

git rebase

- 如果开发进行了一段时间才想到要整理之前的提交，该怎么办？
- git rebase --onto <newbase> <since> <till>
  - git checkout D （要把 C 和 D 融为一体）
  - git reset --soft HEAD^^
  - git commit -C C （使用 C 的提交说明）
  - git tag newbase （新提交打上标签多方便）
  - git rebase --onto newbase E^ master （这里用 master 取代 F ，就能直接修改 master 的指向而无须再对其进行 reset HEAD@{1} ）

git rebase -i

- 使用 git rebase -i ，可以通过编辑文件的方式，方便地“定制”提交历史
- git rebase -i <since> <till>
  - <till> 可以省略，默认为 HEAD
- 将希望合并的提交的 pick 修改为 squash （或 fixup ），保存退出即可完成 rebase 操作。

git revert

- 在合作开发的过程中，一旦推送到了远程版本库，就无法改变历史了。如何修正错误提交呢？
- 重新做一次新的提交，即错误的历史提交的反向提交，这样就达到了 git reset HEAD^ 的效果。
  - git revert HEAD

Git clone

- 通过 clone 的方式实现版本库的备份
- git clone
  - 生成一个“看起来一样”的版本库
  - 向有工作区的版本库中推送（ push ）是不允许的，因为这样会搞乱工作区和暂存区（除非设置 receive.denyCurrentBranch=ignore ）
- git clone --bare
  - 生成一个裸版本库，即不包含工作区的版本库

git 协议

- 不同 git 版本库间进行数据交换的方式：
- 智能协议（在数据传输过程中有进度显示）
  - SSH
  - Git
  - 本地协议（ file:// ）
  - HTTP （ git-http-backend ）

git 协议（cont'd)

- 哑协议（远程版本库方没有运行程序，全靠客户端主动发现）
  - FTP
  - rsync
  - HTTP （普通）
  - 哑协议的传输速度较慢，因为客户端需要通过网络获得 .git/info/refs 获取当前版本库的引用列表，再根据提交 ID 访问对象库目录下的文件。

Git pull & push

- 从远程服务器获取版本库的更新
  - git pull = git fetch + git merge
- 把本地的版本库推送到远程版本库
  - git push
- pull 和 push 时如何知道远程版本库在哪里？
  - .git/config: remote “origin”
- pull 时如何知道该和本地的哪个分支合并？
  - .git/config: branch “master”

非快进式推送

- 如果当前分支的每一个提交都已经存在于另一个分支， git 执行“ fast-forward” 操作，即不创建新的提交，只是将当前分支指向合并进来的分支。
- non-fast-forward 推送，即远程版本库有一个 commit ，而自己没有这个 commit ；亦即在上次 git pull 之后有人推送了代码
  - git push -f （危险，会覆盖他人的修改）
  - git pull; merge and resolve conflict; git push;

git 分支

- git branch ：分支列表
- git checkout <branch> ：切换到分支
- git checkout -b <branch> ：新建分支
- git branch -d <branch> ：删除已被当前分支合并的其他分支
- git branch -D <branch> ：强制删除分支

git 分支合并

- git merge <branch>
  - 将 <branch> 分支合并到当前分支
- 如果发生冲突，且自动合并没有成功，则暂存区和工作区内有一个特殊的状态，必须手动解决冲突并提交它到暂存区，否则 commit 会失败
  - 只需编辑发生冲突的文件（像 diff 的样式）
  - git add
  - git commit -m “resolve conflict”

git 分支合并 (cont'd)

- 如果发现不小心合并错了怎么办？
  - git reset --hard HEAD
  - HEAD 指向当前 commit
- 如果已经把合并后的代码 commit ，但还想撤销：
  - git reset --hard ORIG_HEAD （危险）
  - ORIG_HEAD 指向“危险操作”前的 HEAD 。执行 merge 后， ORIG_HEAD 就是 HEAD@{1}

Git tag

- git tag ：列出版本库中的现有标签
  - git tag -l <exp> ：模糊匹配某些标签
- git tag <tag> [<commit>] ：新建轻量级标签
  - git tag <tag> ：以 HEAD 建立标签，最常用
- git tag -m <tag> <commit> ：为新建的标签添加消息
- git tag -a <tag> <commit> ：创建一个标签对象，并需要标签消息。此时标签引用指向一个标签对象，而不是一个 commit 。
- git tag -d <tag> ：删除标签

tag 为什么 push 不上去

- tag 要 push 到远程版本库，与分支操作没有任何区别。
  - git push origin <tagname>
  - git push origin :<tagname> （删除标签）
- 如果 tag 名字和分支名字相同，需要指定 refs ：
  - refs/tags/<tagname>
  - refs/heads/branches/<branchname>
- 把所有的 tags 都 push 到远程版本库：
  - git push origin --tags

tag 为什么 pull 不下来

- 与 push 同理。
  - git pull origin <tagname>
  - git pull origin --tags

Git hooks

- 在 .git/hooks 目录下有一些脚本，在特定的事件被触发后调用。
- 远程版本库在 pre-receive 时检查提交者的用户名和 E-mail 的合法性， gitosis 和 gitolite 就是这样做的
- 利用 git 部署 Web 站点：
  - 建立一个 bare 版本库作为 remote origin
  - Web 目录做一个 clone ，并禁止 Web 访问 .git
  - 在 origin 的 post-update 脚本中：进入 Web 目录执行 git checkout ，更新代码

Git 基本工作流程

```
git config;
git clone;
while (project not finished) {
    cd  工作目录 ;
    git pull;
    编写代码 ;
    git add  修改的文件 ;
    git status;
    git commit -m “message”;
    git pull;
    if (conflict) {
        手动修改发生冲突的文件 ;
        git add  发生冲突的文件 ;
        git commit -m “resolve conflict”;
    }
    git push;
    sleep(); // 休息一会儿
} //end while
```

Git 基本操作

- Git config
- Git init
- Git add
- Git status
- Git commit
- Git checkout
- Git reset
- Git log
- Git clone
- Git pull
- Git push

Git 常用操作

- Git tag
- Git stash
- Git reflog
- Git diff
- Git grep
- Git blame
- Git mv
- Git rm
- Git show
- Git fetch
- Git merge
- Git branch
- Git revert
- Git rebase

推荐资料

- Git Community Book
- Pro Git (progit.org)
- 《Git 权威指南》，蒋鑫著
  - 本 slides 中的大部分内容是从此书中摘抄重组的
- Google