

Git 历险记

概要：

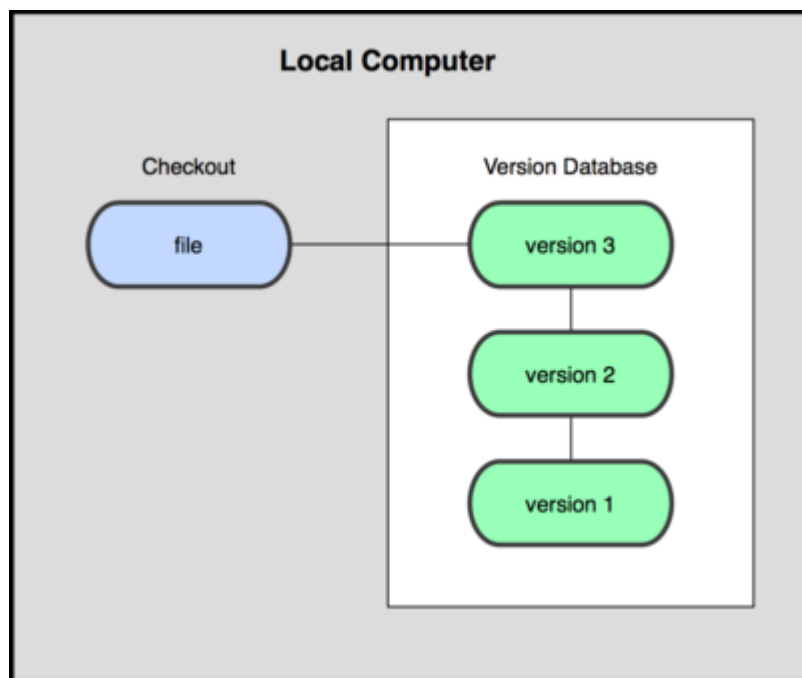
- 一 • 起步
- 二 • Git常用操作
- 三 • 实战
- 四 • 团队开发工作流程
- 五 • 其他

一 起步

- 1 关于版本控制
- 2 Git思想和原理
- 3 安装 Git
- 4 初次运行Git前的配置
- 5 获取帮助

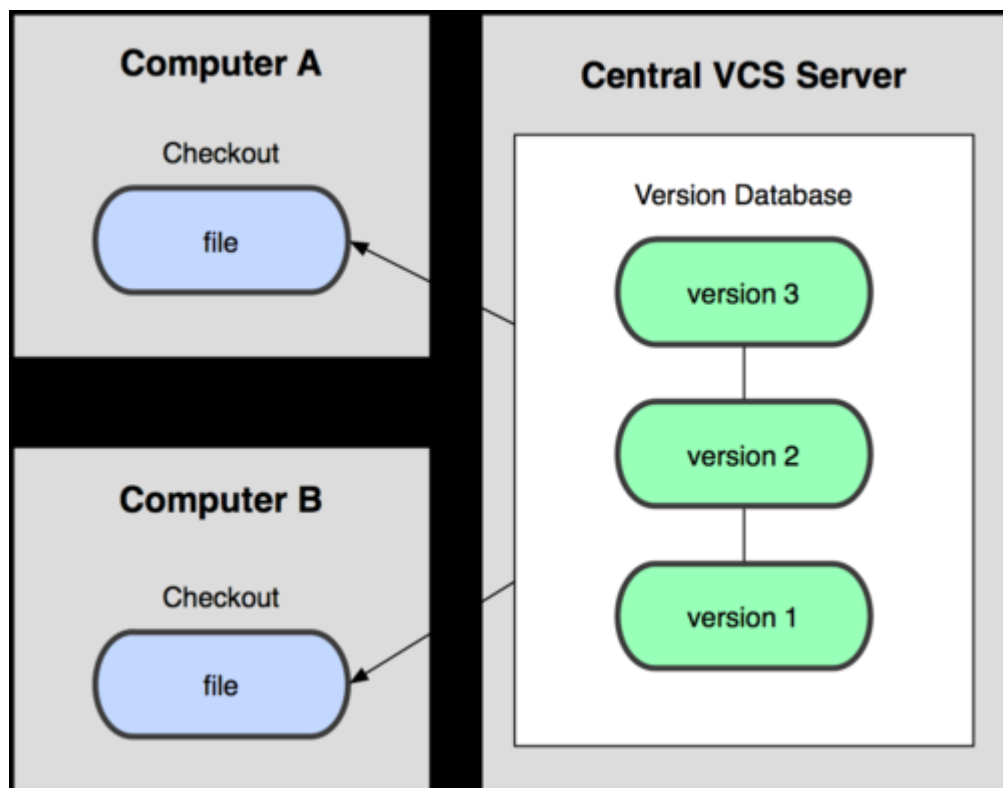
1 本地化版本控制系统

Revision Control System (RCS) 是一个Linux/UNIX 下的版本控制系统



工作原理基本上就是保存并管理文件补丁（**patch**）。文件补丁是一种特定格式的文本文件，记录着对应文件修订前后的内容变化。所以，根据每次修订后的补丁，**rsc** 可以通过不断打补丁，计算出各个版本的文件内容。

2 集中化版本控制系统



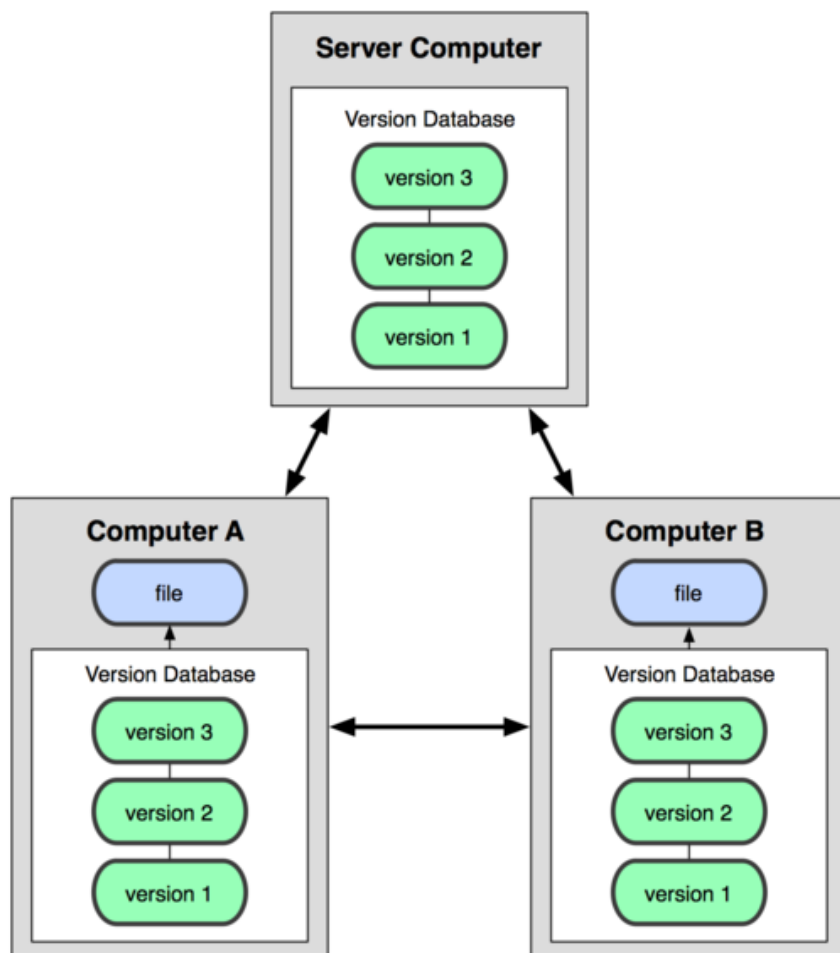
CVS

SVN

Perforce

都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新

3 分布式版本控制系统



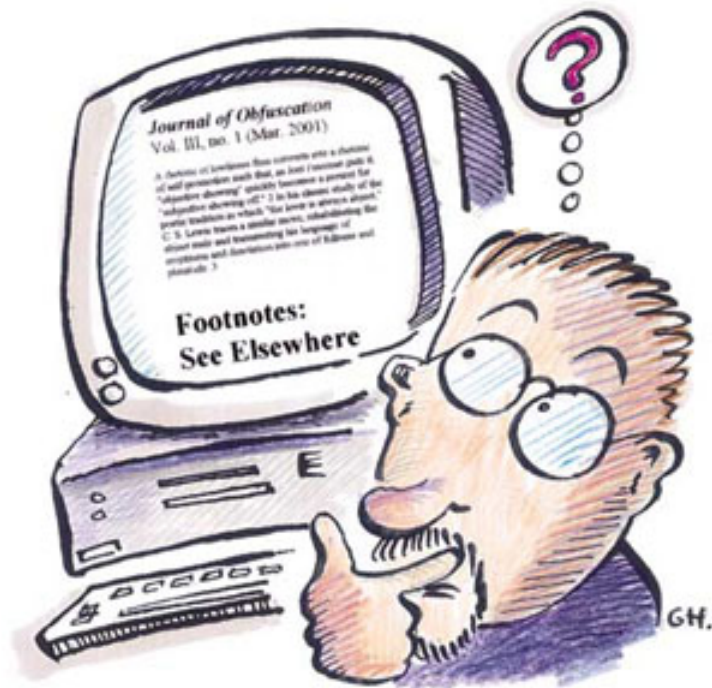
Git

Mercurial

Bazaar

Darcs

客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么一来，任何一处协同工作的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的提取操作，实际上都是一次对代码仓库的完整备份



What is Git?



优势



工作原理

优势

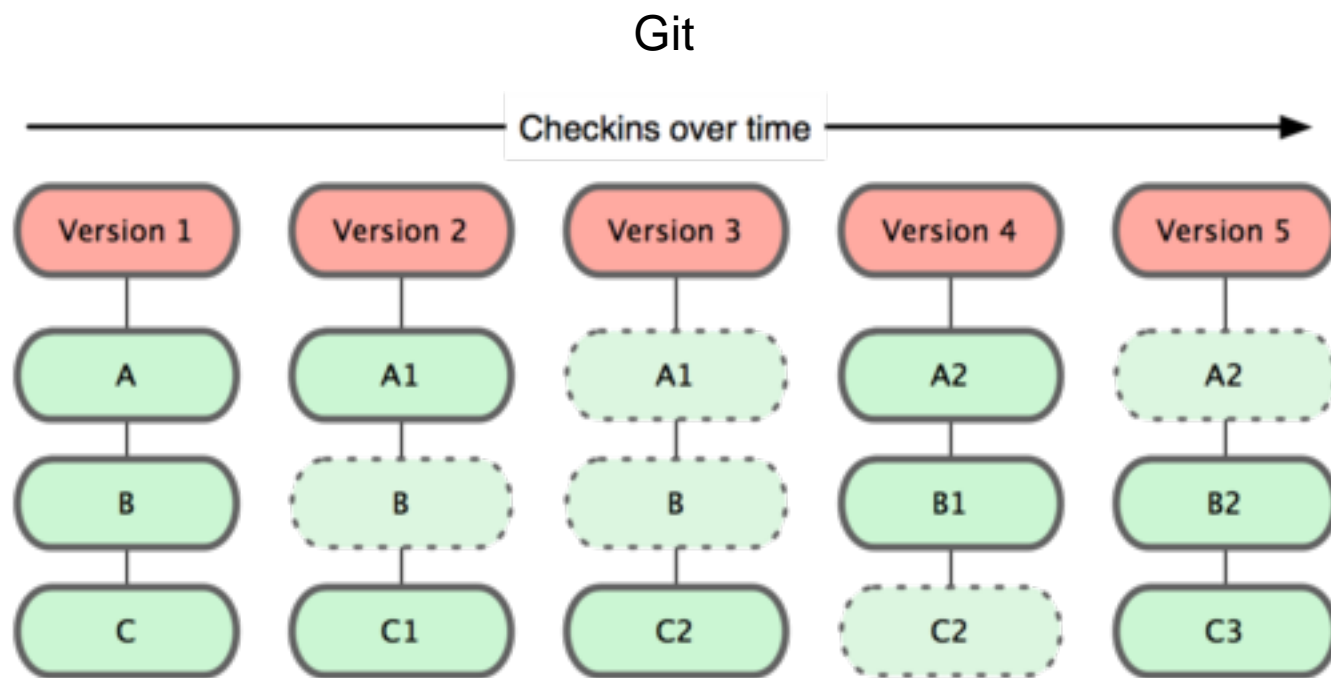
直接记录快照，而非差异比较

近乎所有操作都是本地执行

时刻保持数据完整性

多数操作仅添加数据

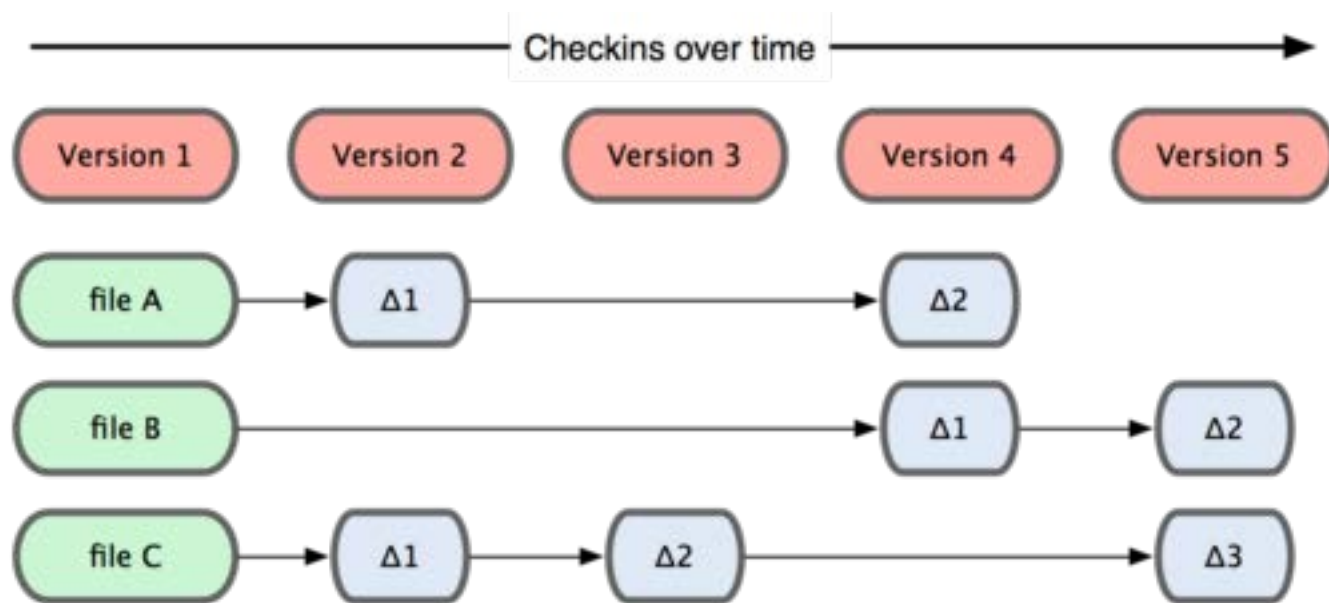
直接记录快照，而非差异比较



保存每次更新时的文件快照

Git 和其他版本控制系统的主要差别

其他系统



在每个版本中记录着各个文件的具体差异

近乎所有操作都是本地执行

绝大多数操作都只需要访问本地文件和资源，不用连网

举个例子，如果要浏览项目的历史更新摘要，**Git** 不用连到外面的服务器上去取数据回来，而直接从本地数据库读取后展示给你看

时刻保持数据完整性

在保存到 **Git** 之前，所有数据都要进行内容的校验和（**checksum**）计算，并将此结果作为数据的唯一标识和索引

24b9da6552252987aa493b52f8696cd6d3b00373

SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 **SHA-1** 哈希值，作为指纹字符串

所有保存在 **Git** 数据库中的东西都是用此哈希值来作索引的，而不是靠文件名

多数操作仅添加数据

常用的 **Git** 操作大多仅仅是把数据添加到数据库。因为任何一种不可逆的操作，比如删除数据，都会使回退或重现历史版本变得困难重重。在别的 **VCS** 中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 **Git** 里，一旦提交快照之后就完全不用担心丢失数据

文件的三种状态

已被安全地保存
到本地数据库中

已提交

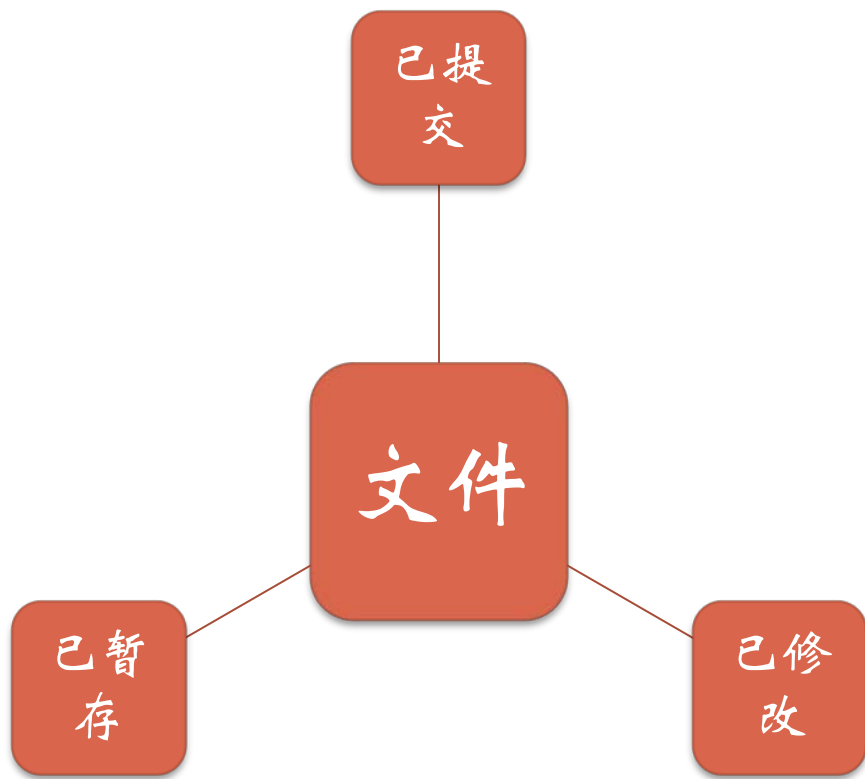
文件

已暂存

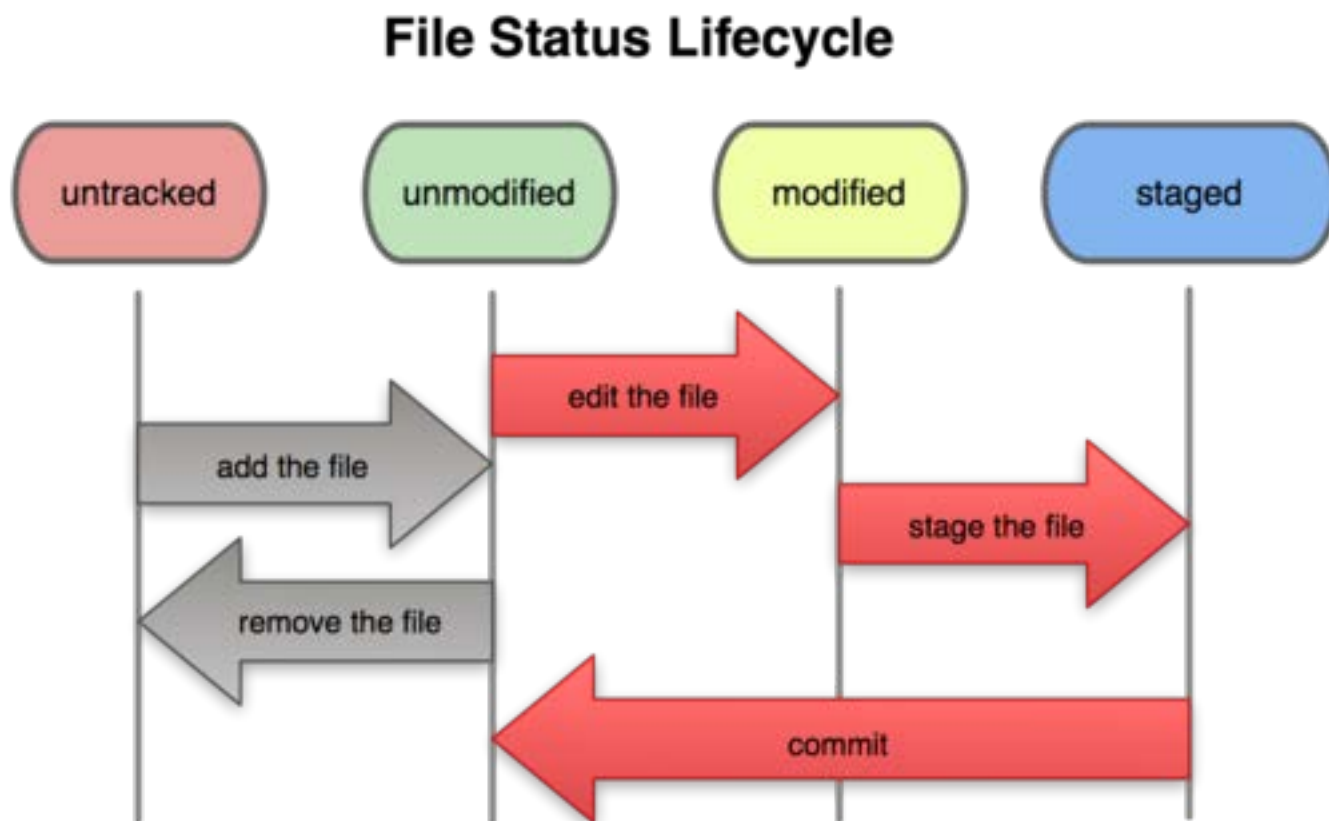
已修改

把已修改的文件
放在下次提交时
要保存的清单中

修改了某个文件，
但还没有保存



文件状态周期图



工作原理



工作目录



暂存区域
(也叫索引文件)



Git本地仓库

文件流转的三个工作区域

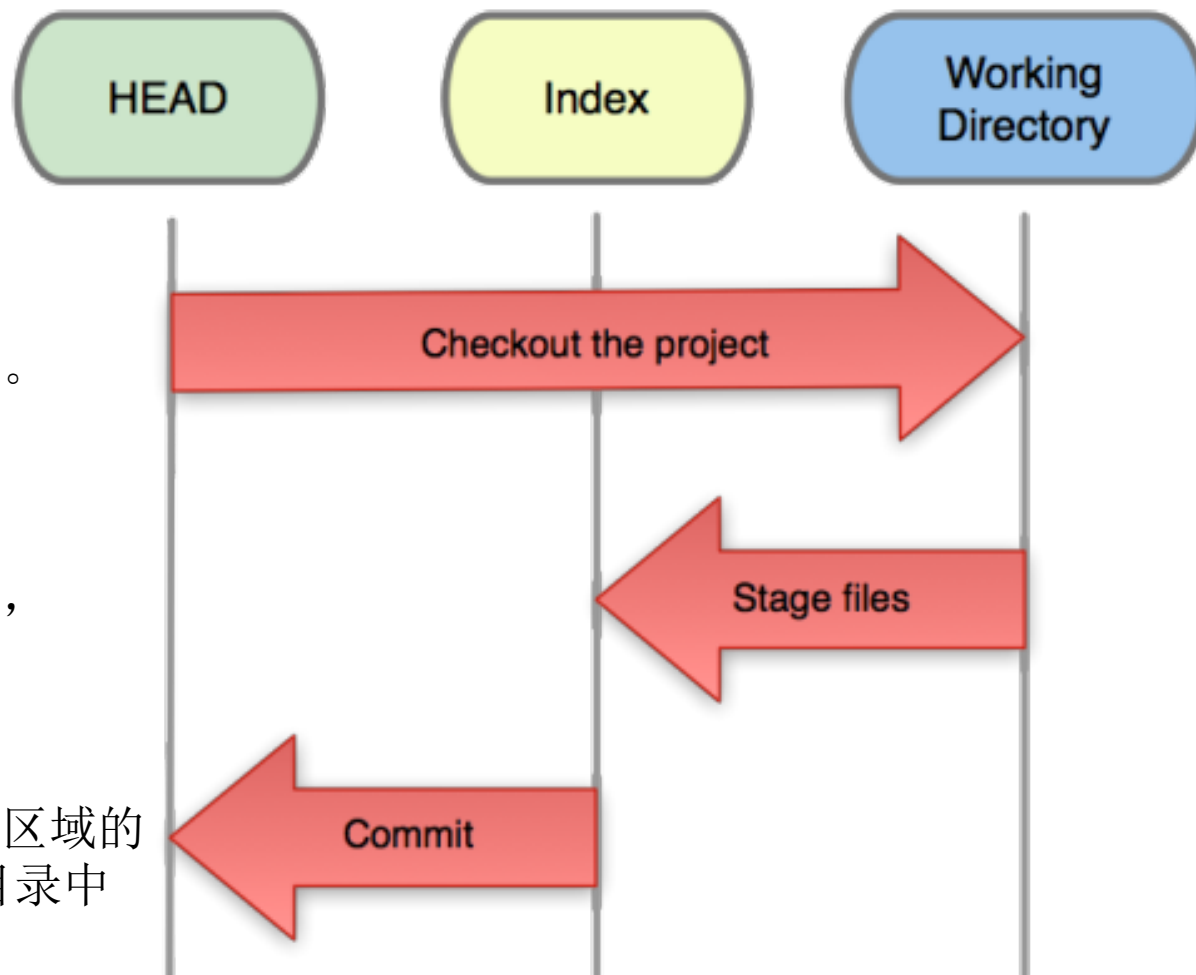
基本的Git工作流程如下：

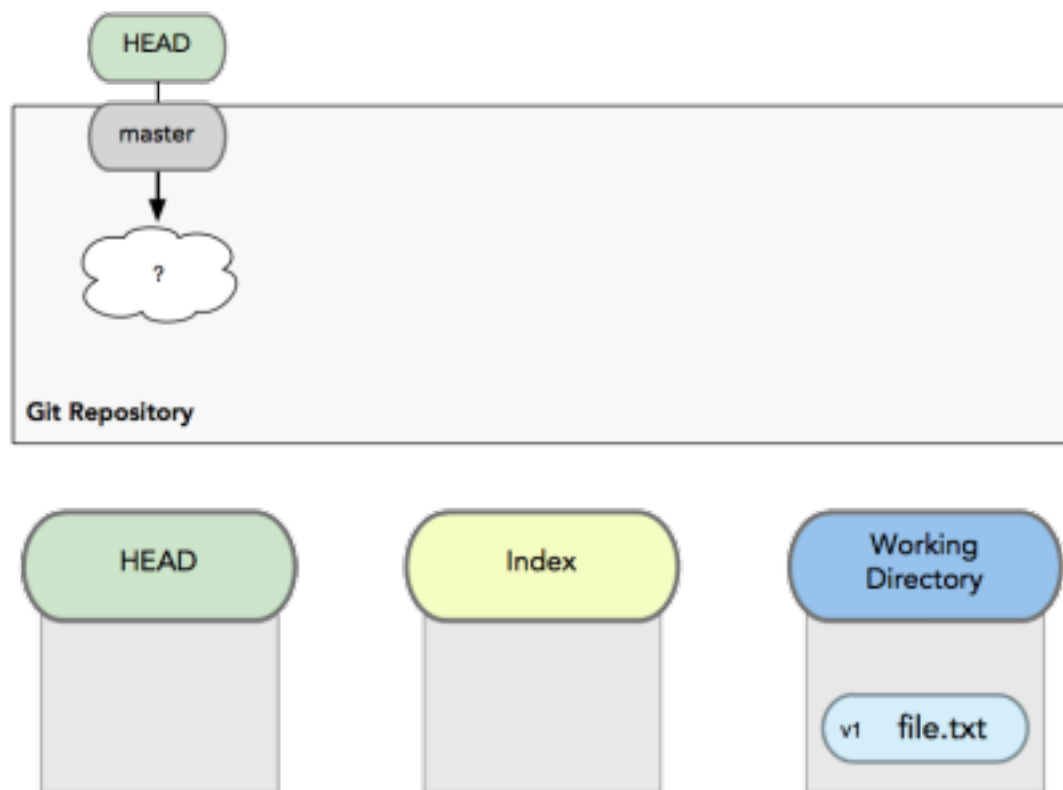
1 从git仓库中checkout项目到工作目录。

2 在工作目录修改某些文件。

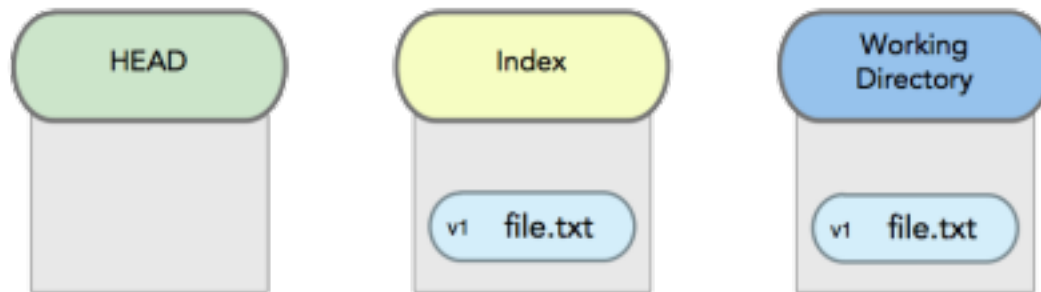
3 对修改后的文件进行快照，然后保存到暂存区域。

4 提交更新，将保存在暂存区域的文件快照永久转储到Git目录中

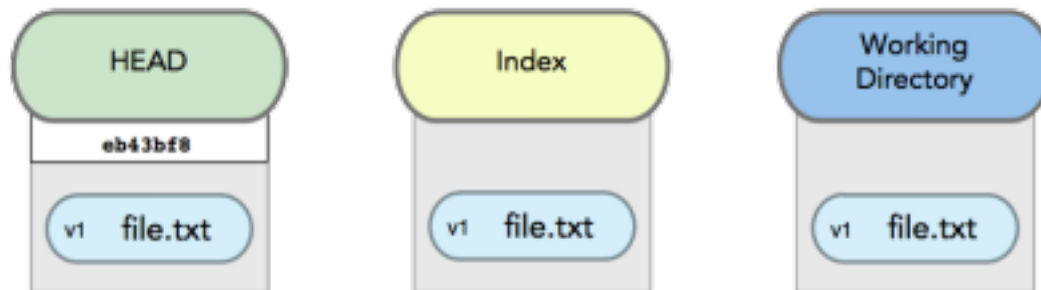
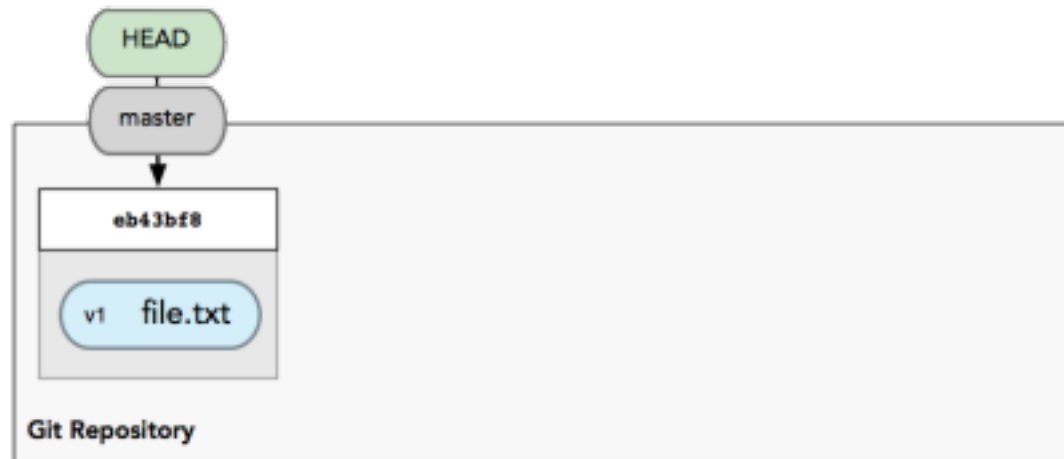




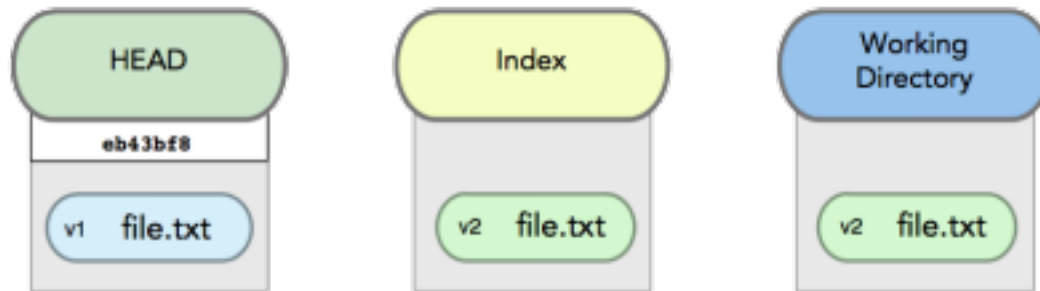
Add File



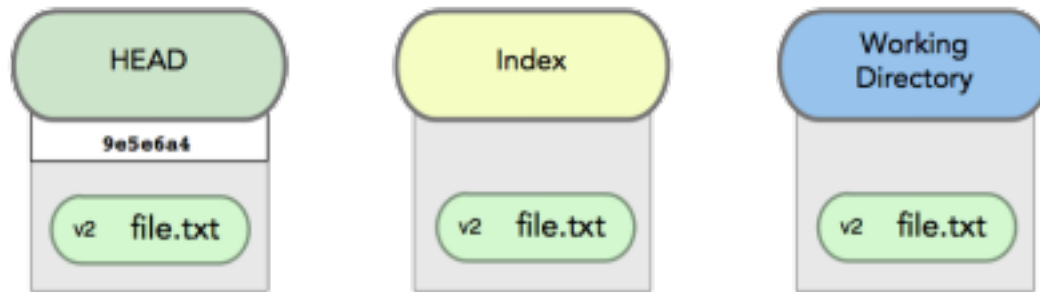
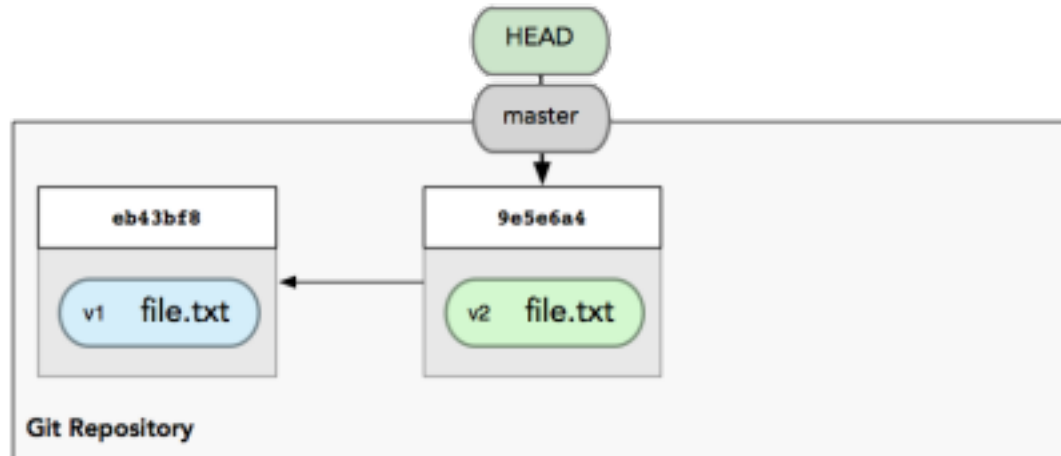
git add



git commit



git add



git commit

安装Git

1 从源代码安装

依赖库

```
$ yum install curl-devel expat-devel gettext-devel \
    openssl-devel zlib-devel
```

```
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
    libz-dev libssl-dev
```

源码下载地址

<http://git-scm.com/download>

编译并安装：

```
$ tar -zxf git-1.7.2.2.tar.gz
```

```
$ cd git-1.7.2.2
```

```
$ make prefix=/usr/local all
```

```
$ sudo make prefix=/usr/local install
```

安装 Git

2 Linux上安装

在 Fedora 上用 yum 安装：

```
$ yum install git-core
```

在 Ubuntu 这类 Debian 体系的系统上，可以用 apt-get 安装：

```
$ apt-get install git-core
```


安装Git

3 在MAC上的安装

安装Xcode后自动装上Git

使用图形化的Git 安装工具Git OS X

<http://code.google.com/p/git-osx-installer>

4 在 Windows 上安装

在 Windows 上安装 Git 同样轻松，有个叫做 `msysGit` 的项目提供了安装包<http://code.google.com/p/msysgit> 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TortoiseGit

5 Eclipse下的插件安装

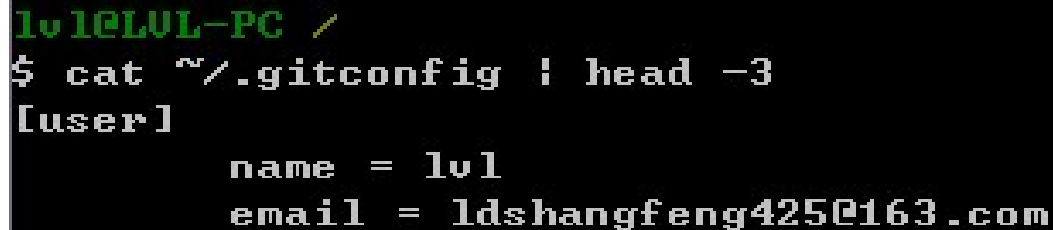
<http://192.168.9.15/wiki/index.php?doc-view-210>

初次运行 Git 前的配置

```
$ git config --global user.name lvi
```

```
$ git config --global user.email ldshangfeng425@163.com
```

```
$ cat ~/.gitconfig | head -3
```



```
lv1@LUL-PC /  
$ cat ~/.gitconfig | head -3  
[user]  
    name = lvi  
    email = ldshangfeng425@163.com
```

如果你想使项目里的某个值与前面的全局设置有区别，可以在`git config`里不带`--global`选项来设置. 这会在你项目目录下的`.git/config`文件增加一节`[user]`内容(如上所示).

获取帮助

```
$ git help <verb>  
$ git <verb> --help  
$ man git-<verb>
```

比如，要学习 `config` 命令可以怎么用，运行：

```
$ git help config
```

小结

至此，你该对 **Git** 有了点基本认识，包括它和以前你使用的 **CVCS** 之间的差别，如何在个操作系统下安装**git**，以及初始化**git**的配置。

二 Git 基础

如何初始代码库化，跟踪文件，暂存和提交更新



如何让Git忽略某些文件



如何既快且容易地撤销犯下的小错误



如何浏览项目的更新历史，查看某两次更新之间的差异



如何从远程仓库拉数据下来或者推数据上去

取得项目的Git 仓库

1 在工作目录中初始化新仓库

```
$ git init
```

```
$ echo "Hello." > README
```

```
$ git add README
```

```
$ git commit -m 'initial project version'
```

2 从现有仓库克隆

```
git clone [url]
```

取得项目的Git 仓库

1 在工作目录中初始化新仓库

```
$ git init
```

```
$ echo "Hello." > README
```

```
$ git add README
```

```
$ git commit -m 'initial project version'
```

2 从现有仓库克隆

```
git clone [url]
```

```
user@server:/path.git
```

Git目录

git init

```
|-- HEAD      # 这个git项目当前处在哪个分支里
|-- config    # 项目的配置信息， git config命令会改动它
|-- description # 项目的描述信息
|-- hooks/    # 系统默认钩子脚本目录
|-- index     # 索引文件
|-- logs/     # 各个refs的历史信息
|-- objects/  # Git本地仓库的所有对象 (commits, trees, blobs, tags)
`-- refs/     # 标识你项目里的每个分支指向了哪个提交(commit)。
```


检查当前文件状态

```
$ git status
```

```
# On branch master
```

```
nothing to commit (working directory clean)
```

```
$ vim README
```

```
$ git status
```

```
# On branch master
```

```
# Untracked files:
```

```
# (use "git add <file>..." to include in what will be committed)
```

```
#
```

```
#      README
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

跟踪新文件

```
$ git add README
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to  
unstage)
```

```
#
```

```
#       new file:   README
```

```
#
```

暂存已修改文件

```
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
#       modified:   benchmarks.rb
#
```

忽略某些文件

`$ cat .gitignore` #作用于.gitignore所存放的目录

glob 模式是指 shell 所使用的简化了的正则表达式

此为注释 – 将被 Git 忽略

`*.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

`*.[oa]` # 忽略所有以 .o 或 .a 结尾的文件

查看已暂存和未暂存的更新---Git diff 魔法

\$ git diff # 工作区和暂存区比较

\$ git diff --cached # HEAD和暂存区比较

\$ git diff HEAD # HEAD和工作区比较

\$ git diff HEAD HEAD^ # HEAD和HEAD的父版本比较

\$ git diff HEAD~2 HEAD^ # HEAD父父版本和HEAD的父版本比较

提交更新

git commit

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
".git/COMMIT_EDITMSG" 10L, 283C
```

git commit -m "-m用来直接增加注释提交"

git commit -a

自动把所有已经跟踪过的文件暂存起来一并提交，
从而跳过 **git add** 步骤，尽量少用，将失去对提交内容进行控制的能力。

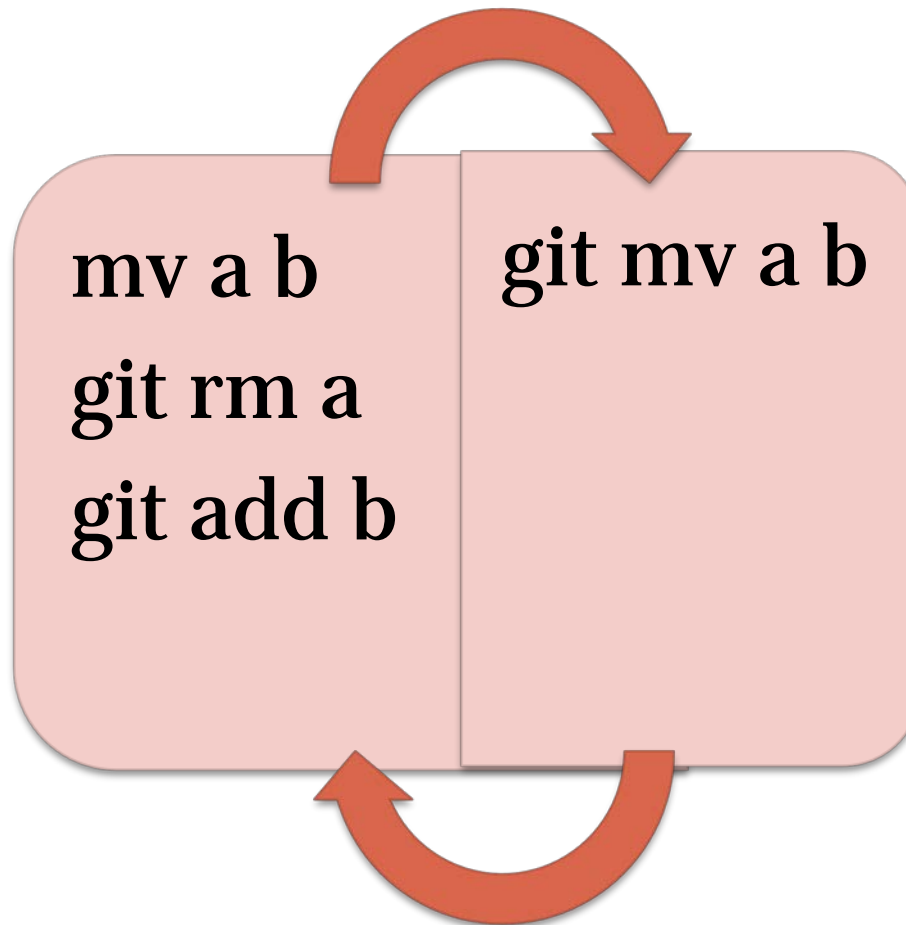
移除文件

`rm filename` #仅工作目录删除

`git rm filename` # 删除并存入暂存区

`git rm --cached filename` # 删除暂存区内的文件

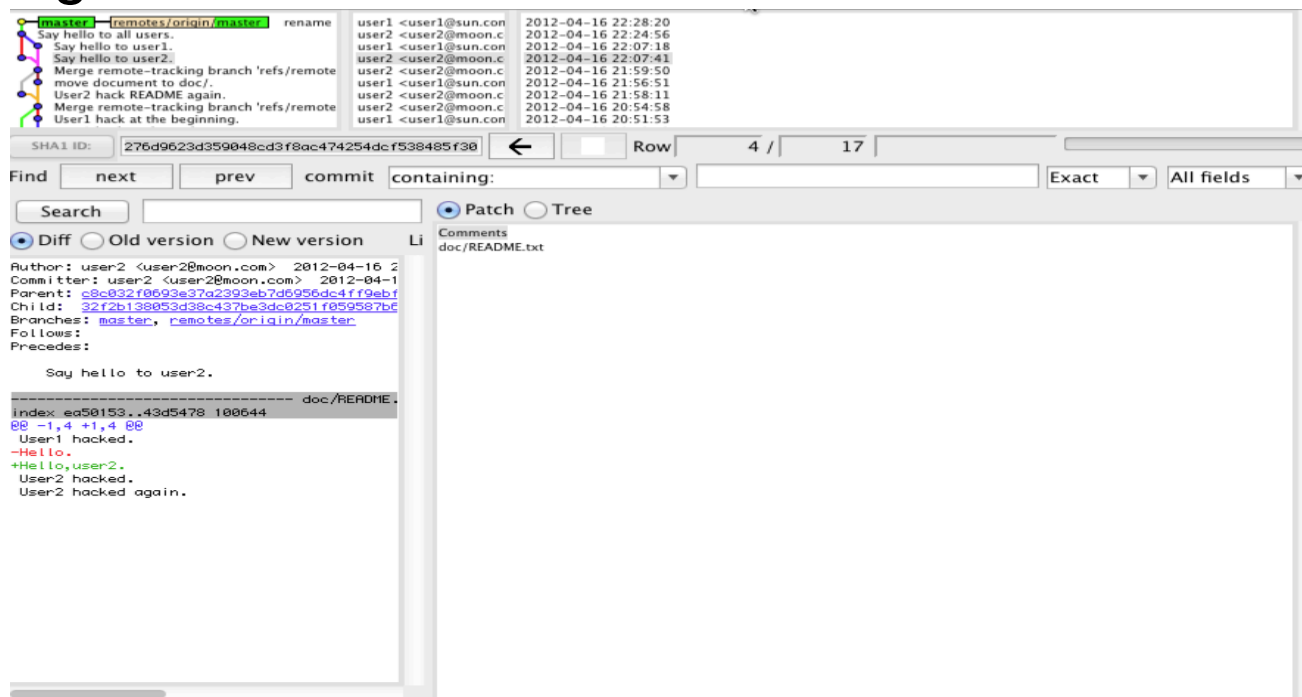
移动文件



查看提交历史

\$ git log # 控制台查询

\$ gitk # 使用图形化工具查阅提交历史



上半个窗口显示的是历次提交的分支祖先图谱，下半个窗口显示当前点选的提交对应的具体差异。

撤销操作

修改最后一次提交

```
$ git commit -m 'initial commit'  
$ git add forgotten_file  
$ git commit --amend
```

取消已经暂存的文件

```
$ git reset HEAD filename
```

取消对文件的修改

```
$ git checkout -- filename
```

远程仓库的使用

查看当前的远程库

```
$ git clone lvlin@192.168.60.45:repos/share.git
```

```
$ git remote -v
```

添加远程仓库

```
$ git remote add temp lvlin@192.168.60.45:repos/share.git
```

```
$ git fetch pb
```

推送数据到远程仓库

```
$ git push origin master
```

查看远程仓库信息

```
$ git remote show origin
```

远程仓库的删除和重命名

```
$ git remote rename temp lvlin
```

```
$ git remote rm lvlin
```

小结

到目前为止，你已经学会了最基本的 **Git** 操作：创建和克隆仓库，做出更新，暂存并提交这些更新，以及查看所有历史更新记录。接下来，我们将通过本地协议来模拟与远程版本库进行交互—交换数据、协同办公以及冲突解决

三 实战模拟

四 **Git**团队开发工作流

集中式 workflow

通常，集中式 workflow 使用的都是单点协作模型。一个存放代码仓库的中心服务器，可以接受所有开发者提交的代码。所有的开发者都是普通的节点，作为中心集线器的消费者，平时的工作就是和中心仓库同步数据（见图 5-1）。

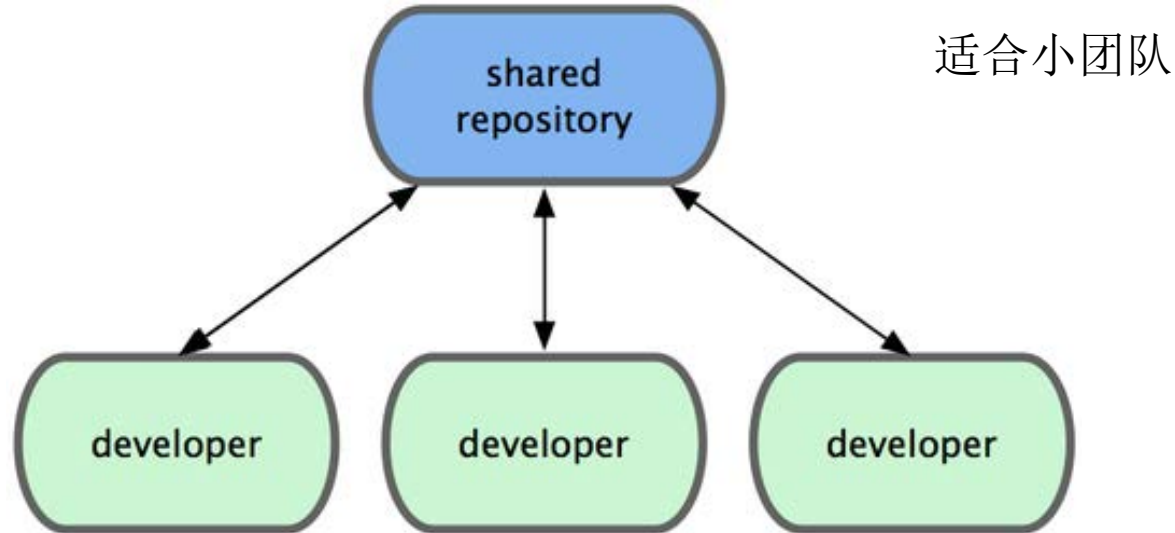


图 5-1. 集中式 workflow

集成管理员 workflow

1. 项目维护者可以推送数据到公共仓库 **blessed repository**。
2. 贡献者克隆此仓库，修订或编写新代码。
3. 贡献者推送数据到自己的公共仓库 **developer public**。
4. 贡献者通知维护者，请求拉取自己的最新修订。
5. 维护者在自己本地的 **integration manger** 仓库中，将贡献者的仓库加为远程仓库，合并更新并做测试。
6. 维护者将合并后的更新推送到主仓库 **blessed repository**。

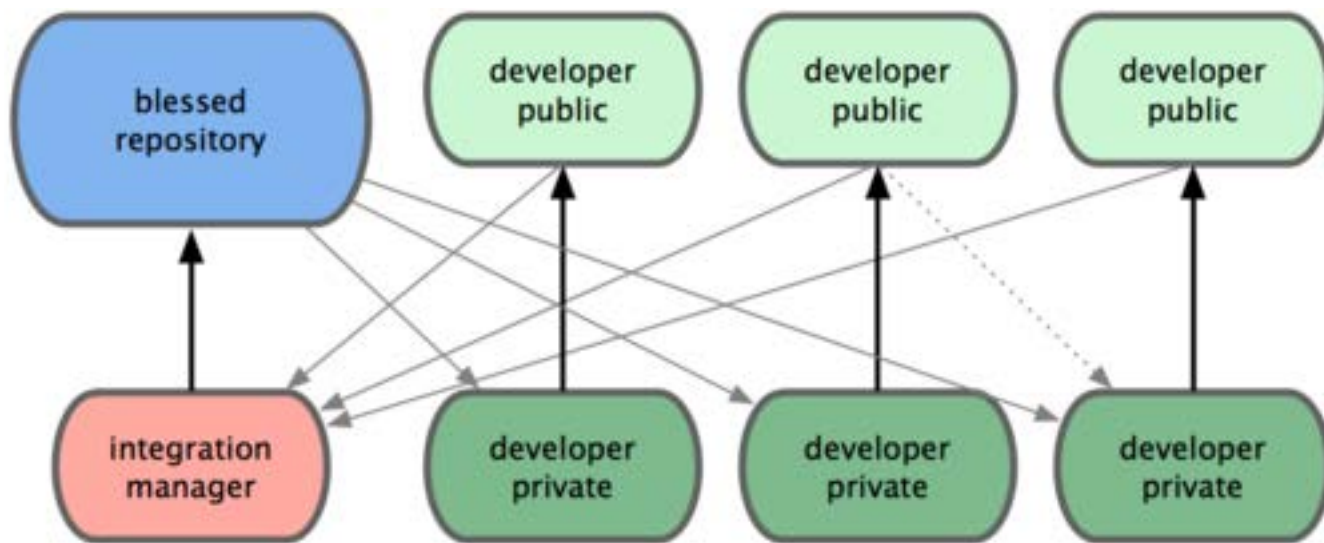


图 5-2. 集成管理员 workflow

五 其他

分支管理

权限管理

GUI工具

正在准备中。。。有相关研究的人可以一起来分享^_^

参考资料:

Pro Git <http://progit.org/book/zh/>

Git Community Book <http://gitbook.liuhui998.com/index.html>

Wiki <http://192.168.9.15/wiki/>

Git权威指南 我桌子上有实体书

A stage with red curtains and the text "Thank You". The stage floor is wooden and has four spotlights. The curtains are red and have tassels. The text "Thank You" is written in white with a yellow glow effect.

Thank You