Blog:<http://omiga.org/blog/archives/1896>

Git笔记-基础

2011-08-10 17:03:10[7](http://omiga.org/blog/archives/1896" \l "comments)条评论»

Git是什么？

Git是一个分布式版本控制系统（Distributed Version Control System，简称 DVCS）。

Git与svn有什么关系？

对于大多数人而言，或许对svn更为熟悉，svn属于集中化的版本控制系统（ Centralized Version Control Systems，简称 CVCS ），在CVCS中会有一个对版本进行集中管理的服务器，协同工作的人都通过客户端连接到该服务器，**检出最新文件或提交更新**。CVCS在协同开发中有两个比较主要的缺点：

1. 如遇服务器宕机，整个协同工作无法进行，因为此时无法进行代码更新提交，当然也不能checkout最新代码
2. 如果服务器数据丢失，整个版本数据也将丢失，除非刻意为版本管理服务器备份

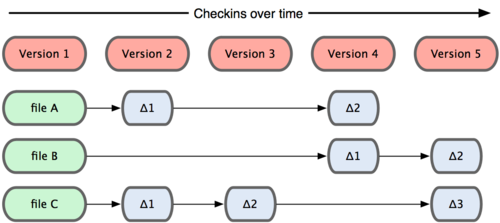
对于这两个主要问题，DVCS都有比较好的解决方案：

首先，DVCS可以方便地在本地进行版本管理，就如同在你本地有一个版本管理服务器一样。你可以选择在合适的时候将本地版本推送到统一的版本管理服务器。

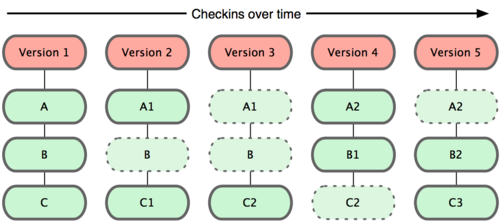
其次，DVCS每次都会提取整个代码仓库的完整镜像，即相当于对整个代码仓库进行了一次备份。这样即使版本管理服务器出现意外，也可以轻松地采用任一本地仓库恢复。结合本地版本管理功能，在远程版本管理服务器出现故障的情况下，你依然可以放心的进行工作，当远程服务器恢复工作时，再提交你的本地版本。

直接记录快照，而非差异比较

Git只关心文件数据的整体是否发生变化，而大多数其他系统则关心文件内容的具体差异，并记录这些差异。git则是更像一个微型文件系统，保存更新文件的快照，并为之创建一个索引。



CVS，Subversion，Perforce，Bazaar 等等）每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容



Git 保存每次更新时的文件快照

如何开始?

你需要安装一个Git客户端来开始使用Git，你可以使用[msysgit](http://code.google.com/p/msysgit/)作为你windows上的git客户端，msysgit包含一个命令行工具Git Bash和一个gui工具Git GUI。对于习惯TortoiseSVN的同学来说，或许Git GUI太简陋了，没关系，安装完msysgit后你依然可以安装[TortoiseGit](http://code.google.com/p/tortoisegit/)，实现svn到git的平滑过渡。TortoiseGit不单独介绍，本文主要介绍通过Git Bash使用git进行版本管理，在你熟悉Git Bash后相信你对TortoiseGit也会有更深入的掌握。

安装好msysgit后，运行Git Bash，在开始所有工作之前我们需要先做一些配置，现在我们只需做一些基础配置，详细的配置后面再讲：

# 用户名

$ git config --global user.name 'omiga'

# email

$ git config --global user.email 'omiga@sample.com'

# 文本编辑器，默认vim

$ git config --global core.editor vim

# 差异分析工具

$ git config --global merge.tool vimdiff

因为每次提交git都会记录committer信息，完成上述配置后，通过cd命令进入到任意文件目录，然后使用git init命名即可初始化一个git版本库。

$ cd /d/ohmygod

$ git init

这样ohmygod目录下的任何改动都处于git版本库的管理下了。在该目录下创建一个README文件，再运行git status命令，将会看到git提示README文件处于为“Untracked files”列表中，并给出了“(use git add <file>… to include in what will be committed)”的建议。此时可以使用git add README命名将README文件加入到暂存区。

$ git add README

此时再执行git status，会看到“new file: README”的提示信息。继续执行git commit README -m “create README”

$ git commit README -m "create README"

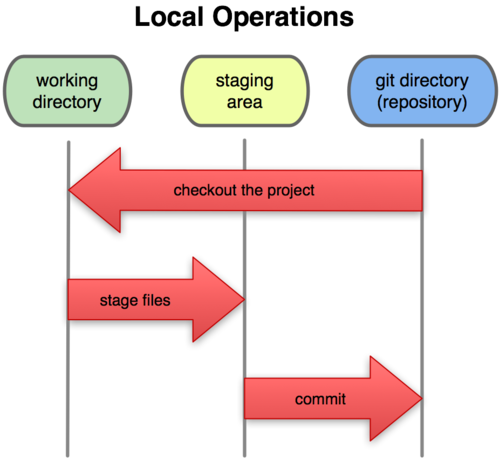
这样README文件就被提交到了本地版本库，完成了一个文件从创建到提交的完整过程：

1. git status – 查看git版本库的状态（这一步不是必须的，但是个好习惯）
2. git add <file> – 将文件提交到暂存区（通常是在通过git status命令后，可以很清楚地看到文件的变更信息，决定哪些文件需要提交到暂存区）
3. git commit – 提交更改（只有处于暂存区的文件会被提交），直接使用git commit命令会调用提交说明窗口。

文件的三种状态

处于git跟踪下的文件具有三种状态：

* 已修改(modified) 已修改过的文件
* 已暂存(staged) 通过git add命令添加到暂存区域的文件
* 已提交(committed) 通过git commit命令提交过的文件



工作目录，暂存区域，以及本地仓库

查看帮助

$ git help <verb>

$ git <verb> --help

$ man git-<verb> # windows下不可用

如查看init命令的帮助信息：

$ git help init

$ git init --help

Git基本操作

创建Git版本库

初始化Git版本库

git init命名将在当前目录新建一个版本库

克隆Git版本库

从你的git版本服务器上clone版本库到本地开展工作，或者从github上clone一个开源项目的代码库，这时候你就需要git clone命令：

# git clone url [newname]

# 克隆到当前目录

$ git clone https://github.com/octocat/Spoon-Knife.git

# 在当前目录新建目录knife 将Spoon-Knife克隆到knife目录

$ git clone https://github.com/octocat/Spoon-Knife.git knife

不管是git init还是git clone都会在你的本地创建一个包含.git目录的git版本库

提交更新

通过git init命名或git clone命令都可以在本地创建一个git版本库，版本库创建成功后便可以在本地进行暂存文件，提交更新等操作了。

在git中可以使用git status查看文件的更改信息，但这个信息比较概要。如果想获取更为详细的更改信息，可以使用git diff命令：

# 查看未暂存文件的变化（与最近一次的暂存/提交比较）

$ git diff

# 查看已暂存文件的变化（与最近一次提交比较）

$ git diff --cached

# 查看与版本库中任一版本的变化

$ git diff 2bd094a

# 查看任意两个版本间的变化

$ git diff 2bd094a 78ab3d1

# 具体到某个文件

$ git diff 2bd094a 78ab3d1 README

关于diff再补充一点内容：如果暂存区/已暂存区都不存在任何未提交的文件，那么diff将对最新版本中与上一版本进行比较。

清楚文件更改信息之后，便可使用git commit对暂存区文件进行提交操作。

如果想提交未暂存文件，可以使用git commit -a命令：

$ git commit -a -m 'all changes'

删除文件

删除Git版本库中的某个文件

可以使用git rm <file>：

$ git rm myfile

当然其实你也可以直接在文件目录中手动删除，这两者的区别在于：使用git rm命令相当于手动删除后使用add命令将更改添加到暂存区域。

删除暂存区的某个文件

$ git reset HEAD <file>

回退版本

如果想撤销某（几）次提交，回退到某个版本，可以使用git reset [--mode] <commit>：

$ git reset [--(**mixed|soft|hard**)] cec8506

* –mixed 默认模式，回退到某次提交后的未暂存状态
* –soft 回退到某次提交后的暂存状态
* –hard 回退到某次提交完整状态，舍弃提交后的所有修改

文件重命名

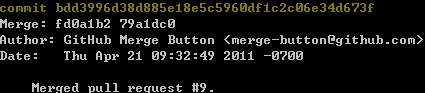
git不会跟踪在文件目录中手动的文件重命名操作，如果手动重命名了某个文件，git会认为这是一次delete-create操作。但是，你可以使用git mv命令完成重命名（文件目录亦使用该命令）：

$ git mv oldfile newfile

查看提交历史

这是git中使用平率非常高的一个操作，git中查看提交历史的功能也非常强大，提供各种筛选和输出格式定制功能。

最简单的，运行git log命令，你将看到一个详细的提交日志：



git-log

# 当然也可以只查看某个版本

$ git log fd0a1b2

信息内容都很好理解，重点说说第一行commit后这个40个字符的字符串，这是该次提交的对应的SHA-1值，在git中，会对提交（commit）、文件（blob）、目录（tree）、标签（tag）生成一个唯一的SHA-1值，git就是基于此来得知文件或目录的改动，因为这四类对象计算得到的SHA-1值都是唯一的，同时你也可以直接使用SHA-1值来指代相应的对象。比如：

$ git show bdd3996

# 查看某个版本下具体某个文件

$ git show bdd3996 README

git log还有很多命令选项来定制历史记录

|  |  |
| --- | --- |
| 选项 | 说明 |
| -(n) | 仅显示最近的 n 条提交 |
| –since,–after | 仅显示指定时间之后的提交 |
| –until,–before | 仅显示指定时间之前的提交 |
| –author | 仅显示指定作者相关的提交 |
| –committer | 仅显示指定提交者相关的提交 |
| –reverse | 按时间倒序显示 |
| -p | 按补丁格式显示每个更新之间的差异 |
| –stat | 显示每次更新的文件修改统计信息 |
| –shortstat | 只显示 –stat 中最后的行数修改添加移除统计 |
| –name-only | 仅在提交信息后显示已修改的文件清单 |
| –name-status | 显示新增、修改、删除的文件清单 |
| –abbrev-commit | 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符 |
| –relative-date | 使用较短的相对时间显示（比如，“2 weeks ago”） |
| –graph | 显示 ASCII 图形表示的分支合并历史 |
| –pretty | 使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式） |

可以通过对上述选项进行组合定制出更为个性化的日志信息，比如：

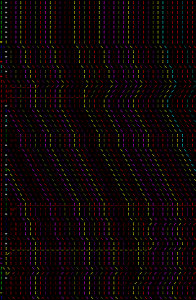
$ git log --committer 'god' --shortstat --pretty=oneline

该命令将以单行模式显示由god提交的统计信息。

$ git log -p -5

显示最近5次提交的，并显示其差异

除此之外，git log –graph也很好玩。我git log –graph了一下git项目的日志，非常壮观。



git graph log

–pretty=format

单独介绍下–pretty=format选项。使用format和占位符可以定制出更为个性化的显示格式。

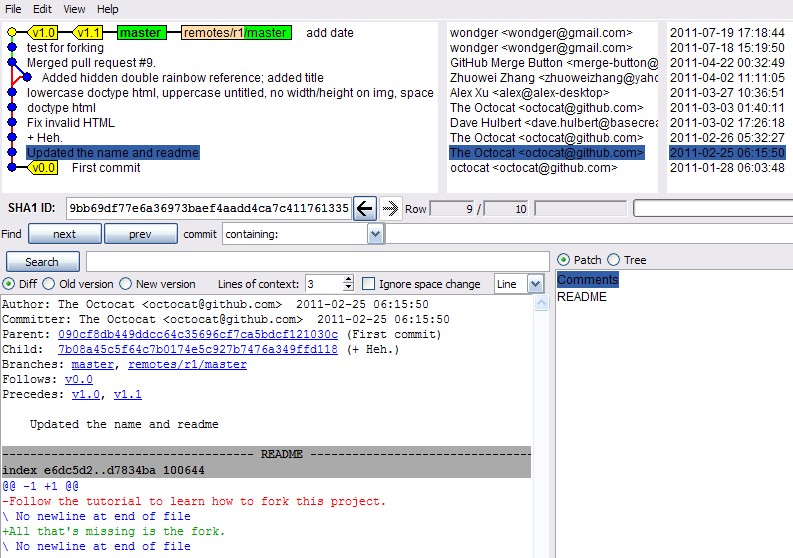
|  |  |
| --- | --- |
| 选项 | 说明 |
| %H | 提交对象（commit）的完整哈希字串 |
| %h | 提交对象的简短哈希字串 |
| %T | 树对象（tree）的完整哈希字串 |
| %t | 树对象的简短哈希字串 |
| %P | 父对象（parent）的完整哈希字串 |
| %p | 父对象的简短哈希字串 |
| %an | 作者（author）的名字 |
| %ae | 作者的电子邮件地址 |
| %ad | 作者修订日期（可以用 -date= 选项定制格式） |
| %ar | 作者修订日期，按多久以前的方式显示 |
| %cn | 提交者(committer)的名字 |
| %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 |
| %cr | 提交日期，按多久以前的方式显示 |
| %s | 提交说明 |

$ git log --pretty=format:'%h by %ce at %cd'

该命令将以“简短SHA-1 by 提交者 at 提交时间”的格式显示日志

图形化日志界面gitk

gitk命名会启用图形化的日志界面



gitk

撤销操作

前面已经介绍过使用[git reset](http://omiga.org/blog/archives/1896" \l "git-reset)来撤销暂存区的文件，以及回退整个版本。但如果只想恢复某个文件，则需要使用checkout — <file>命名：

$ git checkout -- README

它只有在修改文件还没有暂存的情况下，使用最近的提交版本进行恢复。如果文件已经暂存，则需要先使用git reset HEAD <file>从暂存区删除文件，再使用该命令。

修改最后一次提交

git commit –amend命名允许你对最后一次提交信息重新编辑。相当于重新进行一次提交，覆盖掉上一次提交。

远程版本库的使用

尽管git在本地也可以方便地进行版本管理，但是多人协作，或者是多地操作时，总需要使用到远程仓库来进行版本维护。在前面创建版本库的内容中其实使用clone命名就已经是在与远程版本库进行交互了，clone远程库后便会自动创建一个名为origin的远程库，可以使用git remote -v命名查看远程库的详细信息。

$ git remote -v

而在实际工作中，我们可能需要频繁地与某一个或几个远程库交互，那么更好的办法是使用一个别名把远程库保存起来。git添加远程库的方法很简单：

# $ git remote add <name> <remote-url>

$ git remote add pro-git https://github.com/progit/progit.git

这样便将https://github.com/progit/progit.git这个远程git版本库添加并命名为pro-git，后续你只需要使用pro-git便可以指代progit的远程库。

push

当你完成本地工作，并将改动提交到本地版本库后，你便可以使用push将本地提交推送到远程仓库了：

# $ git push <remote-name> <branch-name>

$ git push pro-git master

默认会使用origin和master作为远程仓库和本地分支的名称。

当然你也可以将本地分支推送到远程仓库作为一个分支：

# $ git push <remote-name> <local-branch>:<remote-branch>

$ git push pro-git master:git-branch

当<local-branch>为空时，会尝试删除远程分支：

$ git push pro-git :git-branch

上述命令会删除远程仓库中的”git-branch”分支

fetch与pull

fetch与pull命令都会将一个远程仓库抓取到本地，不同的是fetch仅仅是将远程仓库抓取到本地，以供进行后续操作；pull除了将远程仓库抓取到本地，还会试图与本地当前分支进行合并。

他们与clone不同的是，clone会copy一份版本仓库到本地，如果本地已存在版本仓库，则会被clone后的仓库替换。而fetch和pull都需要在已有本地仓库的条件下操作，不能作为创建本地仓库的方法，即是必须先git init或是git clone后才能使用fetch和pull。

删除远程仓库

$ git remote -d <remote-name>

准确一点说，这只是删除远程仓库在本地的别名，而不是真正删除远程服务器上的git仓库。

重命名远程仓库

$ git remote rename oldname newname

Git别名

虽然git中很多命名都简单易记，但每次都手动输入这些命令确实会浪费不少时间，而且也有那么些命令选项非常冗长，这时就可以使用别名来简化命令的输入了。

别名属于配置项内容，所以需要使用git config命令，如可以为“checkout -b”命令配置别名“cob”

$ git config --global alias.cob 'checkout -b'

为“commit -a -m”配置别名“cam”:

$ git config --global alias.cam 'commit -a -m'

为单行图像化显示log命令“log –pretty=oneline –graph”配置别名“lol”:

$ git config --global alias.lol 'log --pretty=oneline --graph'

至此，git基础篇结束。你已经可以使用git进行日常的代码管理维护，下一篇进阶篇将着重介绍分支，git配置，git原理以及github等内容。