版本管理

GIT 版本管理基本使用

2011 年 07 月

关于本文档

主　　题	GIT版本管理基本使用说明
说　　明	
适用对象	

修 订 历 史					
版 本	**章 节**	**类 型**	**日 期**	**作 者**	**说 明**

目录

1 前言

1.1 目的

本文档对 GIT 版本管理常用命令进行说明，这里只说明了常用的几个命令，如果要了解更多，可以参考官方文档。

1.2 术语与缩写词

暂无。

1.3 参考资料

Pro Git Book.

NetWork.

2 中文环境配置

2.1 Git Bash 窗口 ls 正常显示中文

在$GitHome\etc\git-completion.bash 文件中添加：

alias ls='ls --show-control-chars --color=auto'

2.2 Git Bash 窗口正常输入中文

修改在$GitHome\etc\inputrc 文件中的两项配置：

set output-meta on

set convert-meta off

2.3 Git-log 正常显示中文

在$GitHome\etc\profile 文件中添加：

export LESSCHARSET=utf-8

2.4 Git-gui 正常显示中文

在$GitHome \etc\gitconfig 文件中修改或添加如下配置：

[gui]

encoding = utf-8

[i18n]

commitencoding = GB2312

如果没有这一条，虽然我们在本地用$ git log 看自己的中文修订没问题，但，一、我们的 log 推到服务器后会变成乱码；二、别人在 Linux 下推的中文 log 我们 pull 过来之后看起来也是乱码。这是因为，我们的 commit log 会被先存放在项目的.git/COMMIT_EDITMSG 文件中；在中文 Windows 里，新建文件用的是 GB2312 的编码；但是 Git 不知道，当成默认的 utf-8 的送出去了，所以就乱码了。有了这条之后，Git 会先将其转换成 utf-8，再发出去，于是就没问题了。

[core]

quotepath = false

作用：没有这一条，$git status 输出中文会显示为 UNICODE 编码。

3 Git 配置文件的优先级

Git 加载配置文件首先是 git 安装目录的 etc/gitconfig 文件，其次是用户目录（windows 下是 C:\Documents and Settings\Administrator[如果用户为 Administrator]）下的.gitconfig 文件，最后是 Git 工作目录的.git/config 文件，并且后边的配置会覆盖前边的。

因此，Git 配置的优先级为：

1——git 工作目录.git/config

2——用户目录.gitconfig

3——安装目录 etc/gitconfig

他们的配置方式也不相同，当然都可以直接修改文件达到修改配置的目的。

Git config 是配置 git 工作目录的选项；

Git config –global 是配置用户目录下的选项；

安装目录下的选项需要手工修改文件。

4 基本操作命令

4.1 建立版本库

建立空版本库：

Mkdir repo

Cd repo

git init

建立空版本库，版本库只有版本记录，没有文件：

git init –bare

（假定版本库的绝对路径是：d:/repo）

修改版本库属性，允许提交：

git config receive. denyCurrentBranch ignore 没有这一条无法提交到版本库。

如果是从头建立一个版本库，需要从其他目录上传文件到版本库，假如资源文件在目录 d:/source，把 source 目录资源上传的过程如下：

Cd source

Git init

git remote add repo file:///d:/repo

git add .

git commit –m “comment”

git push

4.2 克隆版本库

git clone file://d:/repo repo_clone

如果没有 repo_clone，将建立一个 repo 版本库，带上建立 repo_clone 版本库

4.3 忽略

如果要忽略某些文件或目录可以：

1、 在.git/conf 中增加忽略的文件或忽略模式

2、 在工作目录增加.ignore 文件，内容同上

4.4 查看更新状态

Git status

4.5 比较

比较不在暂存区的文件与上个版本的区别

Git diff

比较在暂存区的文件与上个版本的区别

Git diff –cached

比较不在和在暂存区的文件与上个版本的区别

Git diff head

4.6 移除文件

Rm　file

处理过程与修改一个文件相同

Git rm file

相当于：rm file

Git add file

以上两个命令都会把文件从库和本地同时删除。

Git rm –cached file

次命令只会从库删除，不会删除本地文件。

4.7 移动文件

Git mv file_from file_to

命令：git mv README.txt README

相当于

$ mv README.txt README

$ git rm README.txt

$ git add README

这其实是个重命名操作

4.8 查看提交历史

Git log——命令查看方式；

Gitk——调用可视化界面查看。

默认不用任何参数的话，git log 会按提交时间列出所有的更新，最近的更新排在最上面。

Git log

我们常用-p 选项展开显示每次提交的内容差异，用-2 则仅显示最近的两次更新：

Git log –p -2

--stat，仅显示简要的增改行数统计，每个提交都列出了修改过的文件，以及其中添加和移除的行数，并在最后列出所有增减行数小计：

Git log –stat

一下是 git log 的常用选项：

选项说明

-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）。

git log 还有许多非常实用的限制输出长度的选项，也就是只输出部分提交信息。

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。

来看一个实际的例子，如果要查看Git 仓库中，2008 年10 月期间，Junio Hamano 提交的但未合并的测试脚本（位于项目的t/ 目录下的文件），可以用下面的查询命令：

$ git log --pretty="%h:%s" --author=gitster --since="2008-10-01" --before="2008-11-01" --no-merges -- t/

4.9 修改历史

4.9.1 修改最后一次提交

参见 3.10.1

4.9.2 修改多次提交

git rebase -i HEAD~3 修改最近三次提交

把对应的 pick 改为 edit，关闭后，执行：

git commit --amend

可以修改历史记录。

如果还要增加文件可以在 git commit –amend 之前执行：

Git add file

一切修改完成后，执行：

Git rebase –continue 回到分支。

4.9.3 重排提交

你也可以使用交互式的衍合来彻底重排或删除提交。如果你想删除“added cat-file”这个提交并且修改其他两次提交引入的顺序，你将rebase脚本从这个

pick f7f3f6d changed my name a bit

pick 310154e updated README formatting and added blame

pick a5f4a0d added cat-file

改为这个：

pick 310154e updated README formatting and added blame

pick f7f3f6d changed my name a bit

当你保存并退出编辑器，Git 将分支倒回至这些提交的父提交，应用310154e，然后f7f3f6d，接着停止。你有效地修改了这些提交的顺序并且彻底删除了“added cat-file”这次提交。

4.9.4 合并多次提交为一次提交

交互式的衍合工具还可以将一系列提交压制为单一提交。脚本在rebase 的信息里放了一些有用的指示：

#

Commands:

p, pick = use commit

e, edit = use commit, but stop for amending

s, squash = use commit, but meld into previous commit

#

If you remove a line here THAT COMMIT WILL BE LOST.

However, if you remove everything, the rebase will be aborted.

#

如果不用“pick”或者“edit”，而是指定“squash”，Git 会同时应用那个变更和它之前的变更并将提交说明归并。因此，如果你想将这三个提交合并为单一提交，你可以将脚本

修改成这样：

pick f7f3f6d changed my name a bit

squash 310154e updated README formatting and added blame

squash a5f4a0d added cat-file

当你保存并退出编辑器，Git 会应用全部三次变更然后将你送回编辑器来归并三次提交说明。

This is a combination of 3 commits.

The first commit's message is:

changed my name a bit

This is the 2nd commit message:

updated README formatting and added blame

This is the 3rd commit message:

added cat-file

当你保存之后，你就拥有了一个包含前三次提交的全部变更的单一提交。

4.9.5 拆分提交

拆分提交就是撤销一次提交，然后多次部分地暂存或提交直到结束。例如，假设你想将三次提交中的中间一次拆分。将“updated README formatting and added blame”拆分成两次提交：第一次为“updated README formatting”，第二次为“added blame”。你可以在rebase -i脚本中修改你想拆分的提交前的指令为“edit”：

pick f7f3f6d changed my name a bit

edit 310154e updated README formatting and added blame

pick a5f4a0d added cat-file

然后，这个脚本就将你带入命令行，你重置那次提交，提取被重置的变更，从中创建多次提交。当你保存并退出编辑器，Git 倒回到列表中第一次提交的父提交，应用第一次提交（f7f3f6d），应用第二次提交（310154e），然后将你带到控制台。那里你可以用git reset HEADˆ 对那次提交进行一次混合的重置，这将撤销那次提交并且将修改的文件撤回。

此时你可以暂存并提交文件，直到你拥有多次提交，结束后，运行git rebase --continue。

$ git reset HEAD^

$ git add README

$ git commit -m 'updated README formatting'

$ git add lib/simplegit.rb

$ git commit -m 'added blame'

$ git rebase --continue

这会修改你列表中的提交的SHA 值，所以请确保这个列表里不包含你已经推送到共享仓库的提交。

4.10 撤销操作

4.10.1修改最后一次提交

有时候我们提交完了才发现漏掉了几个文件没有加，或者提交信息写错了。想要撤消刚才的提交操作，可以使用--amend 选项重新提交：

Git commit --amend

如果刚才提交完没有作任何改动，直接运行此命令的话，相当于有机会重新编辑提交说明，而所提交的文件快照和之前的一样。

启动文本编辑器后，会看到上次提交时的说明，编辑它确认没问题后保存退出，就会使用新的提交说明覆盖刚才失误的提交。

如果刚才提交时忘了暂存某些修改，可以先补上暂存操作，然后再运行--amend 提交：

$ git commit -m 'initial commit'

$ git add forgotten_file

$ git commit --amend

4.10.2取消已经暂存的文件

git reset HEAD file

4.10.3取消对文件的修改

Git checkout -- file

4.11 使用远程仓库

4.11.1 查看远程仓库

Git remote -v

4.11.2 添加远程仓库

git remote add [shortname] [url]

git remote add pb git://github.com/paulboone/ticgit.git

现在可以用字串 pb 指代对应的仓库地址了。比如说，要抓取所有 Paul 有的，但本地仓库没有的信息，可以运行 git fetch pb。

4.11.3 从远程仓库抓取数据

git fetch [remote-name]

此命令会到远程仓库中拉取所有你本地仓库中还没有的数据。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。如果是克隆了一个仓库，此命令会自动将远程仓库归于 origin 名下。所以，git fetch origin 会抓取从你上次克隆以来别人上传到此远程仓库中的所有更新（或是上次 fetch 以来别人提交的更新）。有一点很重要，需要记住，fetch 命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支，只有当你确实准备好了，才能手工合并。如果设置了某个分支用于跟踪某个远端仓库的分支，可以使用 git pull 命令自动抓取数据下来，然后将远端分支自动合并到本地仓库中当前分支。在日常工作中我们经常这么用，既快且好。实际上，默认情况下 git clone 命令本质上就是自动创建了本地的 master 分支用于跟踪远程仓库中的 master 分支（假设远程仓库确实有 master 分支）。所以一般我们运行 git pull，目的都是要从原始克隆的远端仓库中抓取数

据后，合并到工作目录中当前分支。

4.11.4 推送数据到远程仓库

git push [remote-name] [branch-name]

只有在所克隆的服务器上有写权限，或者同一时刻没有其他人在推数据，这条命令才会如期完成任务。如果在你推数据前，已经有其他人推送了若干更新，那你的推送操作就会被驳回。你必须先把他们的更新抓取到本地，并到自己的项目中，然后才可以再次推送。

4.11.5 查看远程仓库的信息

git remote show [remote-name]

4.11.6 远程仓库的删除和重命名

重命名：

git remote rename origin_name new_name

删除：

git remote rm origin_name

4.12 标签

4.12.1 新建标签

Git tag –a tagName –m “comment”

4.12.2 查看标签

Git show tagName

4.12.3 分享标签

git push origin [tagname]

默认情况下，git push 并不会把标签传送到远端服务器上，只有通过显式命令才能分享标签到远端仓库。

如果要一次推送所有（本地新增的）标签上去，可以使用--tags 选项：

git push origin --tags

4.13 分支

4.13.1 创建分支

Git branch branch_name

4.13.2 切换到新分支

Git checkout branch_name

4.13.3 创建并切换到新分支

Git checkout –b branch_name

4.13.4 合并分支

Git merge

切换到 master 分支，把 hotfix 分支合并到 master 分支：

Git checkout master

Git merge hotfix

4.13.5 衍合分支

如果把衍合当成一种在推送之前清理提交历史的手段，而且仅仅衍合那些永远不会公开的 commit，那就不会有任何问题。如果衍合那些已经公开的 commit，而与此同时其他人已经用这些 commit 进行了后续的开发工作，那你有得麻烦了。

永远不要衍合那些已经推送到公共仓库的更新。

Git rebase

4.13.6 删除分支

Git branch –d branch_name

4.13.7 冲突合并

如果两个分支 git merge 出险了冲突，可以手工打开冲突文件解决后，运行 git add，之后 git 就会认为已经解决了冲突。

4.13.8 查看分支

Git branch

前边带*的是当前分支。

4.13.9 查看各个分支最后一次 commit 信息

Git branch -v

4.13.10 查看哪些分支已经并入当前分支

Git branch --merge

4.13.11 查看哪些分支未并入当前分支

Git branch –no-merge

4.13.12 推送本地分支到远程服务器

git push [远程名] [本地分支]:[远程分支]

Git push origin local_branch:remote_branch

Origin：远程分支在本地的名称

local_branch：本地分支名称

remote_branch：远程分支名称

如果远程分支命名为和本地名称相同，也可以：

Git push origin local_branch

此时，如果协作开发者执行：

Git fetch origin

将得到新分支 local_branch 或 remote_branch（根据你推送的名称）。

值得注意的是，在 fetch 操作抓来新的远程分支之后，你仍然无法在本地编辑该远程仓

库。换句话说，在本例中，你不会有一个新的 serverfix 分支，有的只是一个你无法移动的 origin/serverfix 指针。

如果要把该内容合并到当前分支，可以运行 git merge origin/serverfix。如果想要一份自己的 serverfix 来开发，可以在远程分支的基础上分化出一个新的分支来：

git checkout -b local_branch origin/remote_branch

4.13.13 跟踪分支

git checkout --track origin/remote_branch

此命令执行后，创建本地分支 remote_branch，同时跟踪远程分支。

此命令，相当于：

Git checkout –b brancheName origin/remote_branch

4.13.14 删除远程分支

git push [远程名] :[分支名]

git push [远程名] [本地分支]:[远程分支] 语法，如果省略[本地分支]，那就等于是在说“在这里提取空白然后把它变成[远程分支]”。

4.13.15 查看分支差异

比如有分支 master、experiment

你想要查看你的试验分支上哪些没有被提交到主分支，那么你就可以使用 master..experiment 来让 Git 显示这些提交的日志——这句话的意思是“所有可从 experiment 分支中获得而不能从 master 分支中获得的提交”。为了使例子简单明了，我使用了图标中提交对象的字母来代替真实日志的输出，所以会显示：

$ git log master..experiemnt

4.13.16 查看当前分支和远程分支的差异

git log origin/master..HEAD

这条命令显示任何在你当前分支上而不在远程origin 上的提交。如果你运行git push并且的你的当前分支正在跟踪origin/master，被git log origin/master..HEAD 列出的提交就是将被传输到服务器上的提交。你也可以留空语法中的一边来让Git 来假定它是HEAD。

例如，输入git log origin/master.. 将得到和上面的例子一样的结果—— Git 使用HEAD来代替不存在的一边。

4.13.17 多点查询

你也许会想针对两个以上的分支来指明修订版本，比如查看哪些提交被包含在某些分支中的一个，但是不在你当前的分支上。Git允许你在引用前使用^字符或者--not指明你不希望提交被包含其中的分支。因此下面三个命令是等同的：

$ git log refA..refB

$ git log ^refA refB

$ git log refB --not refA

这样很好，因为它允许你在查询中指定多于两个的引用，而这是双点语法所做不到的。例如，如果你想查找所有从refA或refB包含的但是不被refC包含的提交，你可以输入下面中的一个

$ git log refA refB ^refC

$ git log refA refB --not refC

这建立了一个非常强大的修订版本查询系统，应该可以帮助你解决分支里包含了什么这个问题。

4.13.18 三点查询

你可以指定被两个引用中的一个包含但又不被两者同时包含的分支。如果你想查看master或者experiment中包含的但不是两者共有的引用，你可以运行

$ git log master...experiment

这种情形下，log命令的一个常用参数是--left-right，它会显示每个提交到底处于哪一侧的分支。这使得数据更加有用。

4.14 储藏(Stashing)

经常有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是git stash命令。

4.14.1 储藏当前工作

Git stash

4.14.2 查看现有储藏

Git stash list

4.14.3 应用储藏

Git stash apply

Git stash 可以多次储藏，如果没有参数，默认使用最近的储藏，并应用之。

如果要应用某个储藏，可以加上储藏的名字：

Git stash apply stash@{2}

4.14.4 重新应用储藏

重新应用储藏，同时立刻将其从堆栈中移走，此命令不同于 git stash apply

Git stash pop

Git stash pop stash@{2}

4.14.5 删除储藏

Git stash drop

Git stash drop stash@{2}

此命令会删除储藏内的所有变更，慎用！

4.14.6 在储藏上创建分支

如果你储藏了一些工作，暂时不去理会，然后继续在你储藏工作的分支上工作，你在重新应用工作时可能会碰到一些问题。如果尝试应用的变更是针对一个你那之后修改过的文件，你会碰到一个归并冲突并且必须去化解它。如果你想用更方便的方法来重新检验你储藏的变更，你可以运行 git stash branch，这会创建一个新的分支，检出你储藏工作时的所处的提交，重新应用你的工作，如果成功，将会丢弃储藏。

git stash branch branch_name

4.15 全局修改的核武器

filter-branch

如果你想用脚本的方式修改大量的提交，还有一个重写历史的选项可以用——例如，全局性地修改电子邮件地址或者将一个文件从所有提交中删除。这个命令是filter-branch，这个

会大面积地修改你的历史，所以你很有可能不该去用它，**除非你的项目尚未公开，没有其他人在你准备修改的提交的基础上工作**。尽管如此，这个可以非常有用。

4.15.1 从所有提交中删除一个文件

要从整个历史中删除一个名叫password.txt的文件，你可以在filter-branch上使用--tree-filter选项：

$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

--tree-filter选项会在每次检出项目时先执行指定的命令然后重新提交结果。在这个例子中，你会在所有快照中删除一个名叫password.txt 的文件，无论它是否存在。如果你想删除所有不小心提交上去的编辑器备份文件，你可以运行类似git filter-branch --tree-filter 'rm -f *~' HEAD的命令。

你可以观察到Git 重写目录树并且提交，然后将分支指针移到末尾。一个比较好的办法是在一个测试分支上做这些然后在你确定产物真的是你所要的之后，再hard-reset 你的主分支。

要在你所有的分支上运行filter-branch的话，你可以传递一个--all给命令。

4.15.2 将一个子目录设置为新的根目录

假设你完成了从另外一个代码控制系统的导入工作，得到了一些没有意义的子目录（trunk, tags等等）。如果你想让trunk子目录成为每一次提交的新的项目根目录，filterbranch也可以帮你做到：

$ git filter-branch --subdirectory-filter trunk HEAD

Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)

Ref 'refs/heads/master' was rewritten

现在你的项目根目录就是trunk子目录了。Git 会自动地删除不对这个子目录产生影响的提交。

4.15.3 全局性的更换电子邮件地址

另一个常见的案例是你在开始时忘了运行git config来设置你的姓名和电子邮件地址，也许你想开源一个项目，把你所有的工作电子邮件地址修改为个人地址。无论哪种情况你都可

以用filter-branch来更换多次提交里的电子邮件地址。你必须小心一些，只改变属于你的电子邮件地址，所以你使用--commit-filter：

```
$ git filter-branch --commit-filter '
   if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
   then
      GIT_AUTHOR_NAME="Scott Chacon";
      GIT_AUTHOR_EMAIL="schacon@example.com";
      git commit-tree "$@";
   else
      git commit-tree "$@";
   fi' HEAD
```

这个会遍历并重写所有提交使之拥有你的新地址。因为提交里包含了它们的父提交的SHA-1值，这个命令会修改你的历史中的所有提交，而不仅仅是包含了匹配的电子邮件地址的那些。

4.16 Git 调试

4.16.1 文件标注

Git blame

如果你在追踪代码中的缺陷想知道这是什么时候为什么被引进来的，文件标注会是你的最佳工具。它会显示文件中对每一行进行修改的最近一次提交。

```
$ git blame -L 12,22 simplegit.rb
```

此命令查看 12 至 22 行的修改情况。

4.16.2 二分查找

Git bitsect

标注文件在你知道问题是哪里引入的时候会有帮助。如果你不知道，并且自上次代码可用的状态已经经历了上百次的提交，你可能就要求助于 bisect 命令了。bisect 会在你的提交历史中进行二分查找来尽快地确定哪一次提交引入了错误。

首先你运行git bisect start启动，然后你用git bisect bad来告诉系统当前的提交已经有问题了。然后你必须告诉bisect已知的最后一次正常状态是哪次提交，使用git bisect good [good_commit]：

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git 发现在你标记为正常的提交(v1.0)和当前的错误版本之间有大约12次提交，于是它检出中间的一个。在这里，你可以运行测试来检查问题是否存在于这次提交。如果是，那么它是在这个中间提交之前的某一次引入的；如果否，那么问题是在中间提交之后引入的。假设这里是没有错误的，那么你就通过git bisect good来告诉Git 然后继续你的旅程：

```
$ git bisect good
```

当你完成之后，你应该运行git bisect reset来重设你的HEAD到你开始前的地方，否则你会处于一个诡异的地方：

```
$ git bisect reset
```

5 从 Subversion 迁移到 Git

Git svn clone svn_repo_url