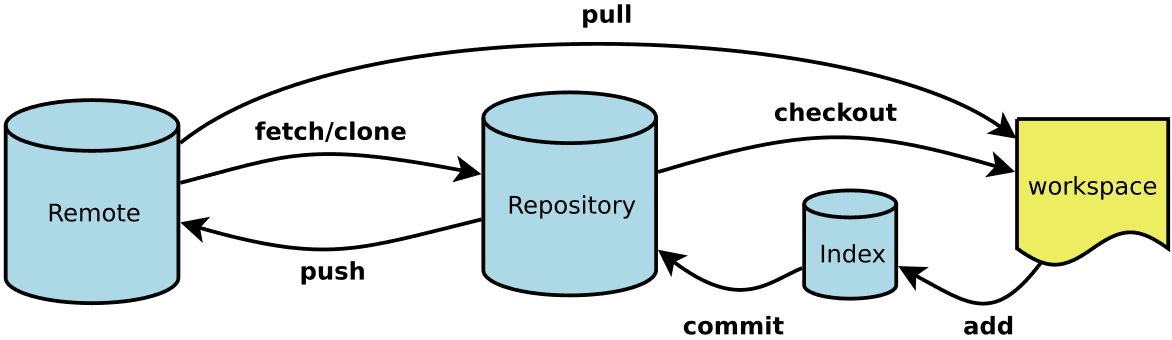
**Git\_常用命令**

一般来说，日常使用只要记住下图6个命令，就可以了。但是熟练使用，恐怕要要记住60~100个命令。



**专用名词**

**Workspace**：工作区

**Index / Stage**：暂存区

**Repository**：仓库区（或本地仓库）

**Remote**：远程仓库

**文件流转的三个工作区域：Git 的工作目录(**工作区**)，暂存区域，以及本地仓库。**

**远程仓库**是指托管在网络上的项目仓库，可能会有好多个，其中有些你只能读，另外有些可以写。

管理远程仓库的工作，包括添加远程库，移除废弃的远程库，管理各式远程库分 支，定义是否跟踪这些分支，等等

[**一、新建代码库**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader0)

[**二、配置**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader1)

[**三、增加/删除文件**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader2)

[**四、代码提交**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader3)

[**五、分支**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader4)

[**六、标签**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader5)

[**七、查看信息**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader6)

[**八、远程同步**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader7)

[**九、撤销**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader8)

[**十、其他**](http://www.open-open.com/lib/view/open1476675065784.html#articleHeader9)

### **一、新建代码库**

**# 在当前目录新建一个Git代码库**

**$ git init**

**# 新建一个目录，将其初始化为Git代码库**

**$ git init [project-name]**

**# 下载一个项目和它的整个代码历史**

**$ git clone git://github.com/schacon/grit.git**

**$ git clone git://github.com/schacon/grit.git mygrit** # **命令末尾指定新的名字**

### ****二、配置****

Git的设置文件为**.gitconfig**，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。

**# 显示当前的Git配置**

**$ git config --list**

**# 编辑Git配置文件**

**$ git config -e [--global]**

**# 设置提交代码时的用户信息**

**$ git config [--global] user.name "[name]"**

**$ git config [--global] user.email "[email address]"**

**# 设定你喜欢的编辑软件**

**git config --global core.editor**

### ****三、增加/删除文件****

**# 添加指定文件到暂存区**

**$ git add [file1] [file2] ...**

**$ git add README.txt**

**# 添加指定目录到暂存区，包括子目录**

**$ git add [dir]**

**# 添加当前目录的所有文件到暂存区**

**$ git add .**

这是个多功能命令，根据目标文件的状态不同，此命令的效果也不同：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等

**# 删除工作区文件，并且将这次删除放入暂存区**

**$ git rm [file1] [file2] ...**

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changed but not updated” 部分（也就是\_未暂存\_清单）看到

如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项**-f**（译注：即 force 的首字母），以防误删除文件后丢失修改的内容。

**# 停止追踪指定文件，但该文件会保留在工作区**

想把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。换句话说，仅是从跟踪清单中删除。

**$ git rm --cached [file]**

**$ git rm --cached readme.txt**

后面可以列出文件或者目录的名字，也可以使用 **glob 模式**。

**# 删除所有log/ 目录下扩展名为 .log 的文件。**

**$ git rm log/\\*.log**

注意到星号 \* 之前的反斜杠 \

实际上不加反斜杠也可以运行，只不过按照 shell 扩展的话，仅仅删除指定目录下的文件而不会递归匹配。上面的例子本来就指定了目录，所以效果等同，但下面的例子就会用递归方式匹配，所以必须加反斜杠。

递归删除当前目录及其子目录中所有 ~ 结尾的文件。

**$ git rm \\*~**

**# 改名文件，并且将这个改名放入暂存区**

**$ git mv [file-original] [file-renamed]**

**# 改名**

**$ git mv first.txt second.txt**

**# 移动文件**

**$ git mv first.txt myFolder**

**# 移动文件并改名**

**$ git mv first.txt myFolder/second.txt**

其实，运行 git mv 就相当于运行了下面三条命令：

**$ mv README.txt README**

**$ git rm README.txt**

**$ git add README**

用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

**# 忽略某些文件**

**$ cat .gitignore \*.[oa] \*~**

第一行告诉 Git 忽略所有以 .o 或 .a 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的，我们用不着跟踪它们的版本。

第二行告诉 Git 忽略所有以波浪符（~）结尾的文件，许多文本编辑软件（比如 Emacs）都用这样的文件名保存副本。

此外，你可能还需要忽略 log，tmp 或者 pid 目录，以及自动生成的文档等等。

文件 **.gitignore 的格式规范**如下：

● 所有空行或者以注释符号 **＃** 开头的行都会被 Git 忽略。

● 可以使用标准的 glob 模式匹配。 **\*** 匹配模式最后跟反斜杠**（/）**说明要忽略的是目录。 \* 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号**（!）**取反。

所谓的 **glob 模式**是指 **shell 所使用的简化了的正则表达式**。

星号（\*）匹配零个或多个任意字符；

[abc] 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；

问号（?）只匹配一个任意字符；

如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如[0-9] 表示匹配所有 0 到 9 的数字）。

一个 .gitignore 文件的例子：

**# 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**

### ****四、代码提交****

**# 每次准备提交前，先用git status 看下，是不是都已暂存起来了**

**# 提交后，会启动文本编辑器以便输入本次提交的说明。**

**# 提交暂存区到仓库区**

-m 参数后跟提交说明的方式，在一行命令中提交更新

**$ git commit -m "Story 182: Fix benchmarks for speed"**

**$ git commit -m [message]**

**# 提交暂存区的指定文件到仓库区**

**$ git commit [file1] [file2] ... -m [message]**

**# 提交工作区自上次commit之后的变化，直接到仓库区**

给 git commit 加上-a 选项，Git 就会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

**$ git commit -a**

**$ git commit -a -m 'added new benchmarks'**

**$ git commit -am 'added new benchmarks'**

**# 提交时显示所有diff信息**

-v 选项将修改差异的每一行都包含到注释中来。

**$ git commit -v**

**# 使用一次新的commit，替代上一次提交**

**# 如果代码没有任何新变化，则用来改写上一次commit的提交信息**

**$ git commit --amend -m [message]**

**# 重做上一次commit，并包括指定文件的新变化**

**$ git commit --amend [file1] [file2] ...**

**记住，提交时记录的是放在暂存区域的快照，任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。**

**每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。**

### ****五、分支****

**# 列出所有本地分支**

分支前的 \* 字符：它表示当前所在的分支

**$ git branch**

**# 列出所有远程分支**

**$ git branch -r**

**# 列出所有本地分支和远程分支**

**$ git branch –a**

**# 查看各个分支最后一个提交对象的信息**

**$ git branch -v**

**# 从该清单中筛选出你已经（或尚未）与当前分支合并的分支**

可以用 –merge 和 –no-merged 选项（Git 1.5.6 以上版本）。

比如用git branch –merge 查看哪些分支已被并入当前分支

**$ git branch --merged**

**# 查看尚未合并的分支**

由于这些分支中还包含着尚未合并进来的工作成果，所以简单地用 git branch -d 删除该分支会提示错误，因为那样做会丢失数据

**$ git branch --no-merged**

**# 新建一个分支，但依然停留在当前分支**

**$ git branch [branch-name]**

**$ git branch testing**

**# 新建一个分支，并切换到该分支**

**$ git checkout -b [branch]**

切换到新建的 serverfix 本地分支，其内容同远程分支 origin/serverfix 一致

**$ git checkout -b serverfix origin/serverfix**

**# 新建一个分支，指向指定commit**

**$ git branch [branch] [commit]**

**# 新建一个分支，与指定的远程分支建立追踪关系**

**$ git branch --track [branch] [remote-branch]**

**# 跟踪远程分支**

从远程分支 checkout 出来的本地分支，称为**跟踪分支**(tracking branch)。跟踪分支是一种和远程分支有直接联系的本地分支。在跟踪分支里输入git push，Git 会自行推断应该向哪个服务器的哪个分支推送数据。反过来，在这些分支里运行 git pull 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

**$ git checkout --track origin/serverfix**

要为本地分支设定不同于远程分支的名字，只需在前个版本的命令里换个名字：

**$ git checkout -b sf origin/serverfix**

本地分支 sf 会自动向 origin/serverfix 推送和抓取数据

**# 切换到指定分支，并更新工作区**

**$ git checkout [branch-name]**

**$ git checkout testing**

**# 建立追踪关系，在现有分支与指定的远程分支之间**

**$ git branch --set-upstream [branch] [remote-branch]**

**把一个分支整合到另一个分支的办法有两种：merge （合并）和 rebase（衍和）**

**衍合是按照每行的修改次序重演一遍修改，而合并是把最终结果合在一起。**

**# 合并指定分支到当前分支**

**$ git merge [branch]**

回到master分支，运行 git merge 命令指定要合并进来的分支

**$ git checkout master**

**$ git merge hotfix**

快进（**Fast forward**）：如果顺着一个分支走下去可以到达另一个分支的话，那么 Git 在合并两者时，只会简单地把指针右移，因为这种单线的历史分支不存在任何需要解决的分歧

**# rebase 命令，就可以把在一个分支里提交的改变移到另一个分支里重放一遍**

**$ git checkout experiment**

**$ git rebase master**

把基于 server 分支而非 master 分支的改变，跳过 server 直接放到master 分支中重演一遍，但这需要用 git rebase 的 –onto 选项指定新的基底分支master

**$ git rebase --onto master server client**

git rebase [主分支] [特性分支] 命令会先取出特性分支server，然后在主分支 master 上重演

**$ git rebase master server**

**一旦分支中的提交对象发布到公共仓库，就千万不要对该分支进行衍合操作。**

**# 选择一个commit，合并进当前分支**

**$ git cherry-pick [commit]**

**# 删除分支**

**$ git branch -d [branch-name]**

**$ git branch -d hotfix**

**# 删除远程分支**

**$ git push origin --delete [branch-name]**

**$ git branch -dr [remote/branch]**

在服务器上删除serverfix 分支

**$ git push origin :serverfix**

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]”。

**# 图形界面工具**

用一个有图形界面的工具来解决这些问题，不妨运行git mergetool，它会调用一个可视化的合并工具并引导你解决所有冲突

**$ git mergetool**

**merge tool candidates**: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff

在”**merge tool candidates**”里找到可用的合并工具列表，输入你想用的工具名。

### ****六、标签****

Git 使用的标签有两种类型：轻量级的（lightweight）和含附注的（annotated）

**轻量级标签**就像是个不会变化的分支，实际上它就是个指向特 定提交对象的引用。

**含附注标签**，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用 GNU Privacy Guard (GPG) 来签署或验证。

**# 列出所有已有的tag**

**$ git tag**

显示的标签按字母顺序排列，所以标签的先后并不表示重要程度的轻重。

可以用特定的搜索模式列出符合条件的标签

只对 1.4.2 系列的版本感兴趣

**$ git tag -l 'v1.4.2.\*'**

**# 新建一个tag在当前commit**

**$ git tag [tag]**

**# 新建一个tag在指定commit**

**$ git tag [tag] [commit]**

**# 含附注的标签**

-a （译注：取 annotated 的首字母）指定标签名字

**$ git tag -a v1.4 -m 'my version 1.4'**

-m 选项则指定了对应的标签说明，Git 会将此说明一同保存在标签对象中。

如果没有给出该选项，Git 会启动文本编辑软件供你输入标签说明。

可以使用 git show 命令查看相应标签的版本信息，并连同显示打标签时的提交对象。

**$ git show v1.4**

**# 签署标签**

如果你有自己的私钥，还可以用 GPG 来签署标签，只需要把之前的 -a 改为 -s （译注： 取 signed 的首字母）

**$ git tag -s v1.5 -m 'my signed 1.5 tag'**

**# 轻量级标签**

**$ git tag v1.4-lw**

现在运行 git show 查看此标签信息，就只有相应的提交对象摘要

**# 验证标签**

可以使用 git tag -v [tag-name] （译注：取 verify 的首字母）的方式验证已经签署的标签。此命令会调用 GPG 来验证签名，所以你需要有签署者的公钥，存放在 keyring 中，才能验证

**$ git tag -v v1.4.2.1**

**# 后期加注标签**

以在后期对早先的某次提交加注标签。

**$ git log --pretty=oneline**

忘了在提交 “updated rakefile” 后为此项目打上版本号 v1.2，没关系，现在也能做。只要在打标签的时候跟上对应提交对象的校验和（或前几位字符）即可：

**$ git tag -a v1.2 9fceb02**

**# 查看tag信息**

**$ git show [tag]**

**# 提交指定tag**

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。其命令格式如同推送分支，运行git push origin [tagname] 即可

**$ git push origin v1.5**

**$ git push [remote] [tag]**

**# 提交所有tag**

**$ git push [remote] --tags**

**$ git push origin --tags**

**# 新建一个分支，指向某个tag**

**$ git checkout -b [branch] [tag]**

### ****七、查看信息****

**# 检查当前文件状态：仅仅列出了修改过的文件**

**$ git status** # On branch master nothing to commit (working directory clean)

# Untracked files

# Changes to be committed 已暂存状态

# Changed but not updated 已跟踪文件的内容发生了变化，但还没有放到暂存区

# Changes to be committed 已暂存

**# 显示当前分支的版本历史（详细见后文）**

**$ git log**

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。

在做代码审查，或者要快速浏览其他协作者提交的更新都作了哪些改动时，就可以用这个选项。

**# 显示commit历史，以及每次commit发生变更的文件**

--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**

# 显示某个文件的版本历史，包括文件改名

**$ git log --follow [file]**

**$ git whatchanged [file]**

# 显示指定文件相关的每一次diff

-p 选项展开显示每次提交的内容差异

-2 则仅显示最近的两次更新

**$ git log -p -2**

**$ git log -p [file]**

在 contrib 特性分支上打了两个补丁，仅查看这两个补丁的提交信息，可以用--not 选项指定要屏蔽的分支 master，这样就会剔除重复的提交历史：

**$ git log contrib --not master**

**# 显示指定文件是什么人在什么时间修改过**

**$ git blame [file]**

**# 显示暂存区和工作区的差异**

比较工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。会使用文件补丁的格式显示具体添加和删除的行。

**$ git diff**

**# 显示暂存区域快照 和 仓库区的差异**

**$ git diff --staged**

**# 显示暂存区和上一个commit的差异**

**$ git diff --cached [file]**

**# 显示工作区与当前分支最新commit之间的差异**

**$ git diff HEAD**

**# 显示两次提交之间的差异**

**$ git diff [first-branch]...[second-branch]**

**# 显示某次提交的元数据和内容变化**

**$ git show [commit]**

**# 显示某次提交发生变化的文件**

**$ git show --name-only [commit]**

**# 显示某次提交时，某个文件的内容**

**$ git show [commit]:[filename]**

**# 显示当前分支的最近几次提交**

**$ git reflog**

**$ git log 详细参数如下：**

--pretty 选项，可以指定使用完全不同于默认格式的方式展示提交历史。

oneline 将每个提交放在一行显示，这在提交数很大时非常有用

short，full 和fuller 可以用，展示的信息或多或少有些不同

**$ git log --pretty=oneline**

format，可以定制要显示的记录格式，这样的输出便于后期编程提取分析

**$ git log --pretty=format:"%h - %an, %ar : %s"**



作者（author）和提交者（committer）之间究竟有何差别，其实作者指的是实际作出修改的人，提交者指的是最后将此 工作成果提交到仓库的人。

用 oneline 或 format 时结合 --graph 选项，可以看到开头多出一些 ASCII 字符串表示的简单图形，形象地展示了每个提交所在的分支及其分化衍合情况。

**$ git log --pretty=format:"%h %s" --graph**

表 2-2 还列出了一些其他常用的选项及其释义。

**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）。**

**限制输出长度**

-2 只显示最近的两条提交，实际上，这是 - 选项的写法，其中的 n 可以是任何自然数，表示仅显示最近的若干条提交。

按照时间作限制的选项，比如 --since 和 –until

最近两周内的提交

**$ git log --since=2.weeks**

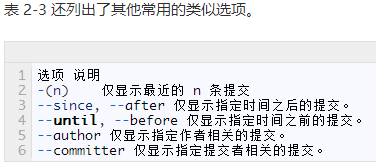
可以给出各种时间格式，比如说具体的某一天（“2008-01-15”），或者是多久以前（“2 years 1 day 3 minutes ago”）。

--author 选项显示指定作者的提交

--grep 选项搜索提交说明中的关键字。

（请注意，如果要得到同时满足这两个选项搜索条件的提交，就必须用--all-match 选项。）

如果只关心某些文件或者目录的历史提交，可以在 git log 选项的最后指定它们的路径。因为是放在最后位置上的选项，所以用两个短划线（--）隔开之前的选项和后面限定的路径名。



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）

**$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \**

**--before="2008-11-01" --no-merges -- t/**

### ****八、远程同步****

**# 从远程仓库抓取数据/ 下载远程仓库的所有变动**

**$ git fetch [remote]**

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。

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。

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）。

**# 显示所有远程仓库**

-v 选项（译注：此为 --verbose 的简写，取首字母），显示对应的克隆地址

**$ git clone git://github.com/schacon/ticgit.git**

**$ cd ticgit**

**$ git remote -v**

**# 远程仓库的删除和重命名**

想把 pb 改成paul

**$ git remote rename pb paul**

注意，对远程仓库的重命名，也会使对应的分支名称发生变化，原来的 pb/master 分支现在成了 paul/master。

碰到远端仓库服务器迁移，或者原来的克隆镜像不再使用，又或者某个参与者不再贡献代码，那么需要移除对应的远端仓库，可以运行 git remote rm 命令：

**$ git remote rm paul**

**# 显示某个远程仓库的信息**

它告诉我们，运行 git push 时缺省推送的分支是什么。

还显示了有哪些远端分支还没有同步到本地

哪些已同步到本地的远端分支在远端服务器上已被删除

运行git pull 时将自动合并哪些分支

**$ git remote show [remote-name]**

**$ git remote show origin**

# 增加一个新的远程仓库，并命名

**$ git remote add [shortname] [url]**

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

用字串 pb 指代对应的仓库地址

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

**$ git fetch pb**

现在，Paul 的主干分支（master）已经完全可以在本地访问了，对应的名字是 pb/master，你可以将它合并到自己的某个分支，或者切换到这个分支，看看有些什么有趣的更新。

**# 取回远程仓库的变化，并与本地分支合并**

**$ git pull [remote] [branch]**

**# 推送数据到远程仓库/ 上传本地指定分支到远程仓库**

**$ git push [remote] [branch]**

**$ git push origin master**

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，合并到自己的项目中，然后才可以再次推送。

**# 强行推送当前分支到远程仓库，即使有冲突**

**$ git push [remote] --force**

**# 推送所有分支到远程仓库**

**$ git push [remote] –all**

### ****九、维护与恢复****

**# 维护**

**$ git gc --auto**

gc 指垃圾收集 (garbage collect)，此命令会做很多工作：收集所有松散对象并将它们存入 packfile，合并这些 packfile 进一个大的 packfile，然后将不被任何 commit 引用并且已存在一段时间 (数月) 的对象删除。

**# 修改最后一次提交：撤消刚才的提交操作**

**$ git commit --amend**

此命令将使用当前的暂存区域快照提交。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，但将要提交的文件快照和之前的一样。

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**

**# 取消对文件的修改**

记住，任何已经提交到 Git 的都可以被恢复。即便在已经删除的分支中的提交，或者用 --amend 重新改写的提交，都可以被恢复（关于数据恢复的内容见第九章）。所以，你可能失去的数据，仅限于没有提交过的，对 Git 来说它们就像从未存在过一样。

（**--**）指当前文件，而非分支

**$ git checkout 3a87227 -- benchmarks.rb**

**# 恢复暂存区的指定文件到工作区**

**$ git checkout [file]**

**# 恢复某个commit的指定文件到工作区**

**$ git checkout [commit] [file]**

**$ git checkout efs001 -– text.txt**

**# 恢复上一个commit的所有文件到工作区**

**$ git checkout .**

**# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变**

**$ git reset [file]**

**# 重置暂存区与工作区，与上一次commit保持一致**

**$ git reset --hard**

**# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变**

**$ git reset [commit]**

**# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致**

**$ git reset --hard [commit]**

**# 重置当前HEAD为指定commit，但保持暂存区和工作区不变**

**$ git reset --keep [commit]**

**# 取消已经暂存的文件**

**$ git reset HEAD benchmarks.rb**

**# 新建一个commit，用来撤销指定commit**

**# 后者的所有变化都将被前者抵消，并且应用到当前分支**

**$ git revert [commit]**

### ****十、其他****

### ****10.1****

**# 生成一个可供发布的压缩包**

**$ git archive**

### ****10.2****

**# 自动完成**

如果你用的是 Bash shell，可以试试看 Git 提供的自动完成脚本。下载 Git 的源代码，进入 contrib/completion 目录，会看到一个git-completion.bash 文件。将此文件复制到你自己的用户主目录中（译注：按照下面的示例，还应改名加上点：cp git-completion.bash ~/.git-completion.bash），并把下面一行内容添加到你的.bashrc 文件中：

**source ~/.git-completion.bash**

也可以为系统上所有用户都设置默认使用此脚本。

Mac 上将此脚本复制到 /opt/local/etc/bash\_completion.d 目录中，Linux 上则复制到/etc/bash\_completion.d/ 目录中。这两处目录中的脚本，都会在 Bash 启动时自动加载。

如果在 Windows 上安装了 msysGit，默认使用的 Git Bash 就已经配好了这个自动完成脚本，可以直接使用。

在输入 Git 命令的时候可以敲两次跳格键（Tab），就会看到列出所有匹配的可用命令建议：

**$ git co commit config**

此例中，键入 git co 然后连按两次 Tab 键，会看到两个相关的建议（命令） commit 和 config。继而输入 m 会自动完成git commit 命令的输入。

命令的选项也可以用这种方式自动完成，其实这种情况更实用些。比如运行 git log 的时候忘了相关选项的名字，可以输入开头的几个字母，然后敲 Tab 键看看有哪些匹配的：

**$ git log --s**

### ****10.3****

**# Git 命令别名**

Git 并不会推断你输入的几个字符将会是哪条命令，不过如果想偷懒，少敲几个命令的字符，可以用 git config 为命令设置别名。来看看下面的例子：

**$ git config --global alias.co checkout**

**$ git config --global alias.br branch**

**$ git config --global alias.ci commit**

**$ git config --global alias.st status**

现在，如果要输入 git commit 只需键入 git ci 即可。

使用这种技术还可以创造出新的命令，比方说取消暂存文件时的输入比较繁琐，可以自己设置一下：

**$ git config --global alias.unstage 'reset HEAD --'**

这样一来，下面的两条命令完全等同：

**$ git unstage fileA**

**$ git reset HEAD fileA**

另外，我们还经常设置**last 命令**：

**$ git config --global alias.last 'log -1 HEAD'**

然后要看最后一次的提交信息，就变得简单多了：

**$ git last**

运行某个外部命令，而非 Git 的附属工具，这个好办，只需要在命令前加上 ! 就行。

如果你自己写了些处理 Git 仓库信息的脚本的话，就可以用这种技术包装起来。作为演示，我们可以设置用 git visual 启动gitk：

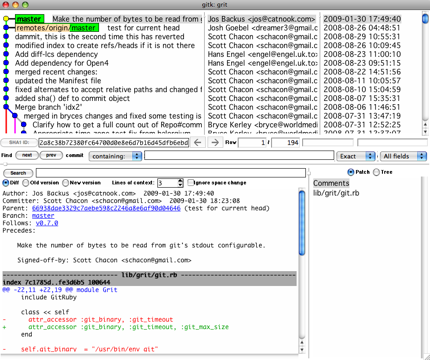
**$ git config --global alias.visual "!gitk"**

### ****10.4****

**# 使用图形化工具查阅提交历史**

gitk工具。它是用 Tcl/Tk 写成的，基本上相当于 git log 命令的可视化版本，凡是git log 可以用的选项也都能用在 gitk 上。

在项目工作目录中输入 gitk 命令后，就会启动

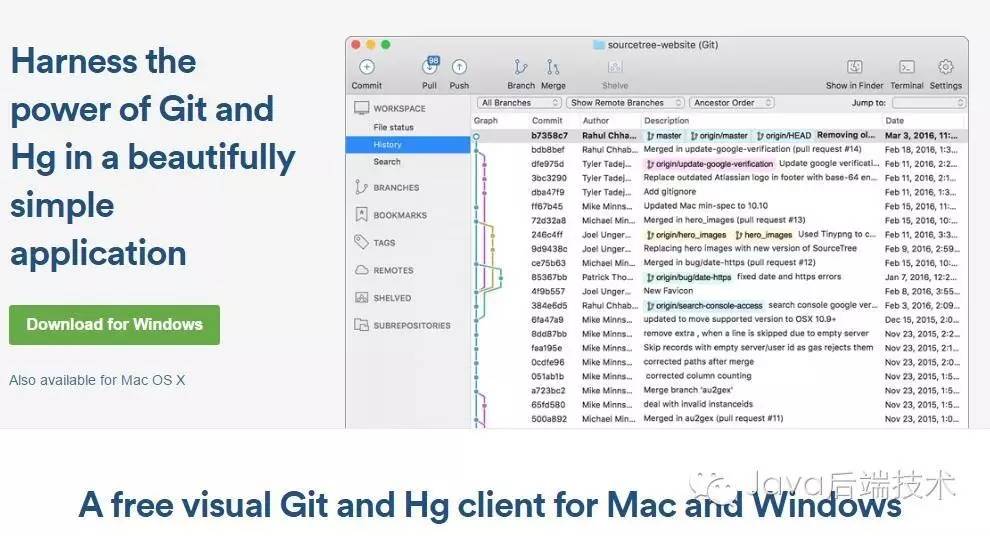


上半个窗口显示的是历次提交的分支祖先图谱

下半个窗口显示当前点选的提交对应的具体差异。

**# 可视化Git工具**

http://static.open-open.com/lib/uploadImg/20161017/20161017113105_500.jpg



### ****10.5****

**# 检查仓库文件个数及大小**

**git count-objects –v [commit]**