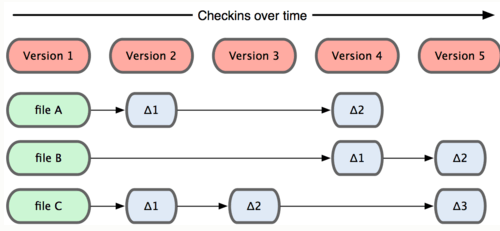
**1 Git与其他版本控制系统的重要区别**

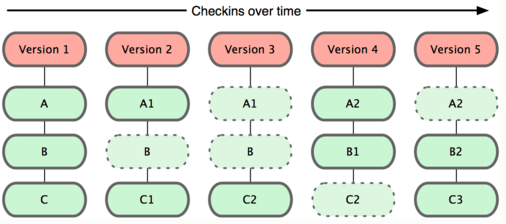
1）分布式

2）直接记录快照，而非比较差异

git和其他版本控制系统的主要差别在于，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只关心文件内容的具体差异。这类系统（CVS，Subversion，Perforce，Bazaar等等）每次记录有哪些文件做了更新，以及都更新了哪些行的什么内容，如下图：



Git并不保存这些前后变化的差异数据，它是一套按内容寻址的文件系统。实际上，git更像是把变化的文件作快照后，记录在一个微型的文件系统中，每次提交更新时，它会纵览一遍所有的文件的指纹信息并对文件做个快照，然后保存一个指向这次快照的索引。为了提高性能，若文件没有变化，Git不会再次保存，而只对上次保存的快照作一链接，如下图：



基本的git工作流程如下：

（1）在工作目录中修改某些文件

（2）对修改后的文件进行快照，然后保存到暂存区域

（3）提交更新，将保存在暂存区域的文件快照永久转到git目录中

**2 git运行配置**

Git提供了一个叫做git config的工具，专门用来配置或读取相应的工作环境，决定了git在各个环节的具体工作方式和行为，这些变量可以存放在以下三个不同的地方：

/etc/gitconfig文件：系统中对所有用户都普遍使用的配置。若使用git config时用—system选项，读写的就是这个文件

~/.gitconfig文件：用户目录下的配置文件只适用于该用户。若使用git config时用

--global选项，读写的就是这个文件

当前项目的git目录中的配置文件（也就是工作目录中.git/config文件）：这里配置仅仅针对当前项目有效。每个级别的配置都会覆盖上层的相同配置，所以.git/config里的配置会覆盖/etc/gitconfig中的同名变量

在windows系统中，git会找寻用户主目录下的.gitconfig文件。主目录即$HOME变量指定的目录，一般都是C:\Documents and Settings\&USER。此外，git还会尝试找寻/etc/gitconfig文件，只不过看当初git装在什么目录，就以此作为根目录定位

**用户信息**

第一个要配置的是你个人的名户名称和电子邮件地址。这两条配置很重要，每次git提交时都会引用这两条信息，说明是谁提交了更新，所以会随更新内容一起被永久纳入历史记录：



如果使用了—global选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这个配置的用户信息。如果要在某个特定的项目中使用其他名字或者电子邮件，只要去掉—global选项重新配置即可，新的设定保存在当前项目中的.git/config文件中（已经测试过）

**查看所有的配置**



有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如/etc/gitconfig）和~/.gitconfig），不过最终git采用的是最后一个

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可



**获取帮助**

想了解git的各式工作怎么用，可以采用下面的三种方法：

Git help <verb>

Git <verb> --help

Man git-<verb>

**3 git基础**

**取得git的项目仓库**

有两种取得git项目仓库的方法。第一种是在现存在目录下，通过导入所有文件来创建新的git仓库。第二种是从已有的git仓库克隆出一个新的镜像仓库。

1）在工作目录中初始化新仓库

采用命令git init

2）从现有的仓库克隆

采用命令git clone [url]

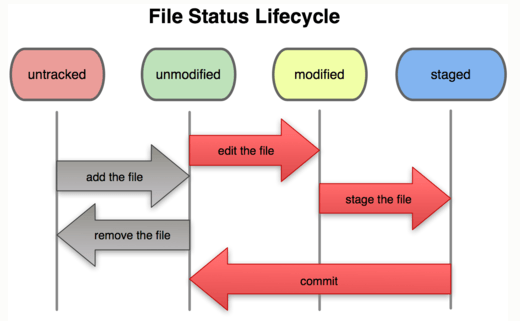
同时如果你想给本地的git仓库取别名，可以使用如下命令



这样远程版本库中的GitDoc就变为本地版本的myGitDoc了

**记录每次更新到仓库**

记住：工作目录下面的所有文件都不外乎两种状态：已跟踪或未跟踪。已跟踪的文件是指本来就被纳入版本控制管理的文件，在上次快照中有它们的记录，工作一段时间后，它们的状态可能未更新，已修改或已放入暂存区。而所有其他文件都属于未跟踪文件，它们既然没有上次更新时的快照，也不在当前的暂存区域中。初次克隆某个仓库时，工作目录的所有文件都属于已跟踪文件，且状态为未修改。



跟踪文件使用git add命令，在git add后面可以致命要跟踪的文件或者目录路径。如果是目录的话，就说明要递归跟踪目录下所有文件（其实git add的潜台词就是把目标文件快照放入暂存区域，也就是add file into staged area，同时未曾跟踪过的文件标记为需要跟踪）

Git add是一个多功能命令，根据目标文件状态的不同，此命令的效果也不同：可以用它来跟踪新文件，或者把已跟踪文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态

**忽略某些文件**

一般我们总会有些文件无需纳入管理，也不希望它们总出现未跟踪文件列表。通常都是些自动生成文件，比如日志文件，或者编译过程中创建的临时文件等，我们可以创建一个名为.gitigonre的文件，列出要忽略的文件模式。比如下面的：

\*.[oa]

\*~

第一行告诉git忽略所有以.a或.o结尾的文件，第二行告诉git忽略以波浪号结尾的文件，此外，你可能还需要忽略log，tmp或pid目录，以及自动生成文档等。要养成一开始就设置好.gitignore文件的习惯，以免将来误提交这类无用的文件

文件.gitignore的格式规范如下：

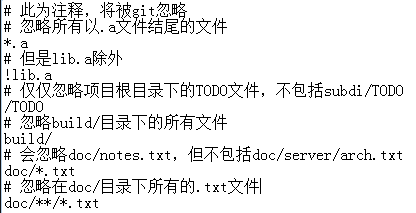
1. 所有空行或者以注释符号#开头的行都会被Git忽略
2. 可以使用标准的glob模式匹配
3. 匹配模式最后跟反斜杠(/)说明要忽略的是目录
4. 要忽略指定模式以外的文件或者目录，可以在模式钱加上惊叹号(!)取反

小技巧：在win7中创建以.开头的文件

采用命令行)

或者创建的文件后面添加.，然后系统会提示文件可能不可识，确认是就可以了

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）



上面为一个示例（在git add的时候已证实）

注意：如果将某个文件已经加入到版本库中了，然后再创建.gitignore是不起作用的。

.gitignore只对还没有加入版本管理的文件起作用

解决办法：在git库中删除该文件并更新

**查看已暂存和未暂存的更新**

实际上git status的显示比较简单，仅仅是列出了修改过的文件，如果要查看具体修改了哪些地方，可以用git diff命令，git diff会使用文件补丁的格式显示具体添加和删除的行。此命令比较的是工作目录中当前文件盒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

若要看已经暂存起来的文件和上次提交时的快照之间的差异，可以用git diff –cached命令（Git 1.6.1及更高版本还允许使用git diff –staged，效果是相同的，但更好记些）

**提交更新**

现在暂存区域中已经准备妥当。在此之前，请一定要确认还有什么修改过的新建的文件还没有git add过，否则提交的时候不会记录这些还没有暂存起来的变化。所以，每次准备提交前，先用git status看下，是不是所有的文件都暂存起来了，然后再运行提交命令

git commit

如果直接git commit为不加-m做说明，会启动文件编辑器以便输入本次提交说明（默认说启动shell的环境变量$EDITOR所指定的软件，一般都是vim或emacs，当然也可以使用之前的git config –global core.editor命令指定你喜欢的编辑软件）



更改了默认的vim变为notepad.exe（已测试），**注意：这里的为linux风格的/**

**跳过使用暂存区域**

尽管使用暂存区域的方式可以静心准备要提交的细节，但有时候这么做略显繁琐。Git提供了一个跳过暂存区域的方式，只要在提交的时候，给git commit添加-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，从而跳过了git add步骤

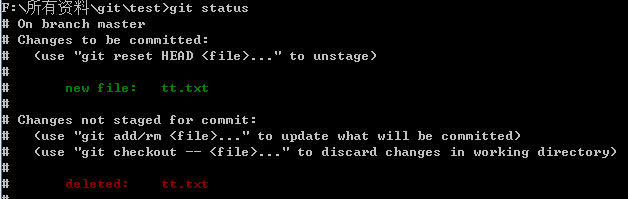
**移除文件**

要从git中移除某个文件，就必须要从已跟踪文件清单中移除（确切的说，是从暂存区移除），然后提交。可以用git rm命令完成此项工作，并连带从工作目录中删除指定文件，这样以后就不会出现在未跟踪文件清单中了。

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“Changes not staged for commit”部分（也就是未暂存清单）

如果文件没有被跟踪，直接删除，没有问题（已验证）

如果文件被跟踪了，直接从工作目录中删除：



然后运行git rm就可以直接删除（也就是手动从工作目录中删除，再调用git rm成功）



没有手动从工作区中删除，如果直接运行git rm（文件在工作区与暂存区相同）



**注意，如果文件夹中有子文件，还需要加上-r表示递归**

没有手动从工作区中删除，如果直接运行git rm（文件在工作区与暂存区不同）



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



不管现在工作区中的qq.txt与暂存区中的qq.txt是否相同，都可以删除

还有一种情况是，我们想从git仓库中删除（即从暂存区中移除），但仍然希望保留在当前工作目录中，换句话说，仅仅是从跟踪清单中删除。比如一些大型日志文件或者一堆.a编译文件，不小心纳入仓库后，要移除跟踪但不删除文件，以便稍后在.gitignore文件中补上，用—cached选项即可：

在工作区目录和暂存区目录不一样的情况下：



在工作区目录和暂存区目录一样的情况下：



**移动文件**

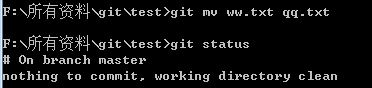
不像其他VCS系统，git并不跟踪文件移动操作，如果在git中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作，不过git非常聪明，它会推断出究竟发生了什么。当你看到git的mv命令时，一定会困惑不已。要在git中对文件改名，可以这么做：

git mv file\_from file\_to

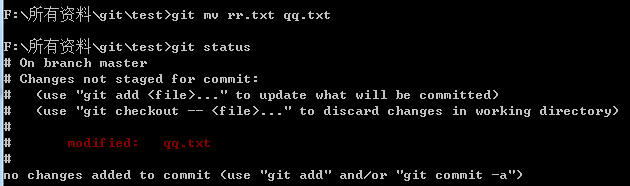




但是换成以前的名字之后：



如果换名字之后修改了文件，又换回以前的文件名字之后：



其实运行git mv就相当于运行了下面三条命令：

Mv a.txt b.txt

Git rm a.txt

Git add b.txt

如此分开操作，git也会意识到这是一次改名操作，所以不管何种方式都一样，当然用git mv轻便的多

**查看提交历史**

想回顾一下提交历史，可以使用git log命令查看，默认不用任何参数，git log会按提交时间列出所有的更新，最后的更新排在最上面，每次更新都有一个SHA-1校验和、作者的名字和电子邮件地址，提交时间，最后一个段落显示提交说明。

有一些常用的参数可以设置，-p选项展开显示每次提交的内容差异，用-2则仅仅显示最近的两次更新



该选项除了显示基本的信息之外，还在附带了每次commit的变化，当进行代码审查时，或者快速浏览某个搭档提交的commit的变化时，这个参数就非常有用了。在层序代码中进行单词层面的对比常常是没有什么用的。不过当你需要在书籍、论文这种很大的文本文件上进行对比时，这个就很有用了，比如：



还有一个常用的—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，比如用oneline将每个提交放在一行显示，这在提交数量很大的时候非常有用，另外还有short、full和fuller可以用，展示的信息或多或少有些不同。

最有意思的是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，比如：









注意作者和提交者的区别：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。所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那个核心成员就是提交者

**限制输入长度**

除了定制输出格式的选项之外，git log还有许多非常实用的限制长度的选项，比如只输出最近的n次提交-<n>的写法

另外，还有按照时间作限制的选项，比如—since和—until，下面的命令列出所有最近两周内的提交。



还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”）

**撤销操作**

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。想要撤销刚才的提交操作，可以用-amend选项重新提交

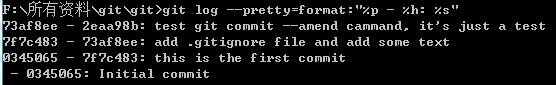
git commit --amend

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。

启动文版编辑器后，会看到上次提交时的说明，编辑它确认没有问题后保存退出，就会使用新的提交说明覆盖刚才失误的提交，如果刚才提交忘了暂存某些修改，可以先不上暂存操作，然后再运行—amend提交：

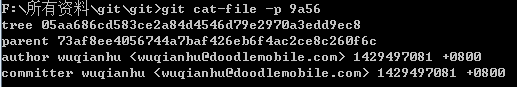


添加一段文字，it’s just a test，查看

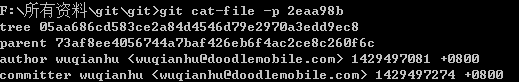


发现只产生了一个提交





我们查看9a56的那次提交，也就是我们撤销的那次提交，发现它的parent为最新的一次提交



上面两次的提交的parent是一样的

这里实际上相当于执行了下面两条命令：

Git reset –soft HEAD^

Git commit –e –F .git/COMMIT\_EDITMSG

2）取消已暂存的文件

下面的命令就是撤销暂存区中的a文件，也就是a文件将会处于unstaged状态



这样，a文件就回到了之前已修改未暂存的状态

3）撤销对文件的修改（

如果觉得对刚才的a文件修改没有必要，该如何取消修改，回到之前的状态（也就是修改之前的版本）呢？



通过上面的这条命令就可以将当前修改的a文件恢复到暂存区中的a文件（已测试）

这条命令有些危险，请务必确定真的不再需要保留刚才的修改。如果只是想回退版本，同时保留刚才的修改以便将来继续工作，可以使用stashing和分支处理

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。所以，你可能失去的数据，仅仅限于没有提交过的，对git来说它们就像从未存在过一样。

Git push <远程主机名> <本地分支名>:<远程分支名>

如果省略远程分支名，则表示将本地分支推送与之存在“追踪关系”的远程分支（通常两者同名），如果该远程分支不存在，则会被创建

Git pull <远程主机名> <远程分支名>:<本地分支名>

查看本地分支：git branch

创建本地分支：git branch test

删除本地分支：git branch –d test 强制删除：git branch –D test

查看远程分支：git branch –a git branch -r

查看本地分支与远程分支的对应关系：git branch –vv

创建本地分支之后添加跟踪：git branch –u origin\_fps/master fps\_master

创建本地分支的同时添加跟踪：git checkout –b test –track origin\_fps/master

创建远程分支：git push origin test:test 将本地的test分支作为远程test

删除远程分支：git push origin :test 注意origin后面的空格

Git push origin –delete test

添加远程仓库：git remote add origin url地址

删除远程仓库：git remote rm 远程仓库名字

重命名远程仓库名字：git remote rename 旧名字 新名字

查看远程仓库的地址：git remote –v git remote只显示远程仓库的地址

查看远程仓库的详细信息：git remote show origin

设置pull默认rebase：git config branch.fps\_master.rebase true

设置所有的分支默认为rebase：git config –local branch.autosetuprebase always

Git config –global push.default (simple/match/upstream/current/nothing)

Nothing-不提交任何内容

Matching-提交所有的分支到相同名字的分支，2.0以前的默认值

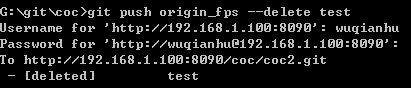
Upstream-提交当前的分支到对应的跟踪分支

Simple-提交当前的分支到相同名字的分支，2.0以后默认值

Current-提交当前的分支到相同名字的分支

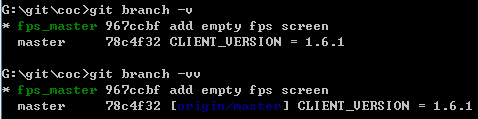
将当前分支的push默认设置为不同的远程服务器的分支

1. 通过设置git config –local push.default upstream
2. 设置配置文件：git config branch.fps\_master.push origin\_fps/master

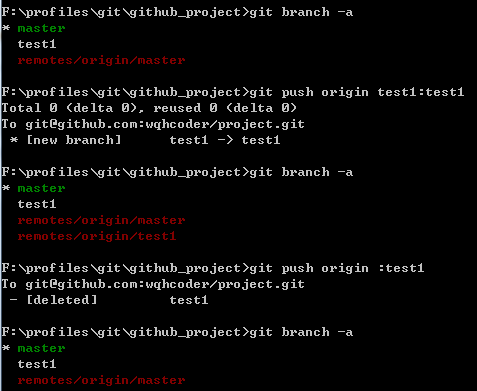












远程删除文件

Git rm –r .

Git commit –a –m “”

Git push origin master

对于上面两者的区别：

Reset：

1 将staged状态修改为unstaged状态

2 –hard：重新设置index和working，同时将HEAD指向commit（其实HEAD仍然引用当前的分支，但是当前的分支指向了新的commit）

-soft：index和working中的内容均不改变，仅仅将HEAD指向commit。这个模式执行完的效果就是，自<commit>以来的所有变化都会显示在git status的“change to be commited”中

-mixed：重新设置index，但不设置working，这个模式是默认模式

Checkout：

1 切换分支

Stash的冲突

git checkout <commit> [--] <paths>

1 <commit>是可选项，如果省略则相当于从暂存区进行检出，和reset命令大不相同，重置的默认值是HEAD，而检出的默认值是暂存区

2 重置一般用于重置暂存区（除非使用—hard，否则不重置工作区），而检出命令主要是覆盖工作区（如果commit不省略，也会替换暂存区中相应的内容）

3 checkout命令不会改变HEAD的头指针，主要用于指定版本文件覆盖工作区中对应的文件，如果省略commit，则会用暂存区的文件覆盖工作区的文件，否则用指定期脚肿的文件覆盖暂存区和工作区中的对应文件

git checkout <branch>

1 会改变HEAD头指针，之所以后面的参数写作<branch>，是因为只有HEAD切换到一个分支才可以对提交进行跟踪，否则仍然会进入”分离头指针”的状态，在”分离头指针”的状态下的提交并不能被引用关联到，从而可能丢失，所以该命令主要作用是切换分支

2 如果省略<branch>则相当于对工作区进行状态检查

通过checkout有可能会导致HEAD处于游离状态，这时候可以强制检出到master分支

Git checkout –f master

git revert命令

git reset是撤销某此提交，但是此次之后的修改都会被退回到暂存区

git revert：是撤销某次操作，此次操作之前的commit都会保留

有可能有冲突，解决冲突之后运行

git add .

git revert --continue

git patch

在git中，我们没有必要直接使用diff和patch来做补丁，这样做既危险又麻烦。Git提供了两种简单的patch方案，一是用git diff生成标准的patch，二是fit format-patch生成的git专用patch

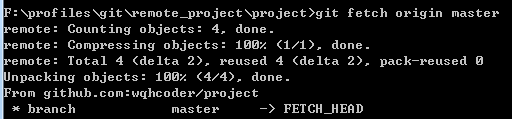
1）git diff



git fetch

理解git fetch关键是理解FETCH\_HEAD

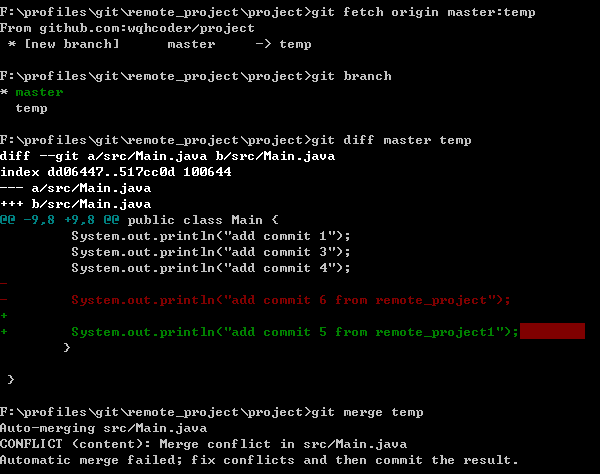
FETCH\_HEAD:指的是某个branch在服务器上最新状态，每一个执行过fetch操作的项目都会存在一个FETCH\_HEAD列表



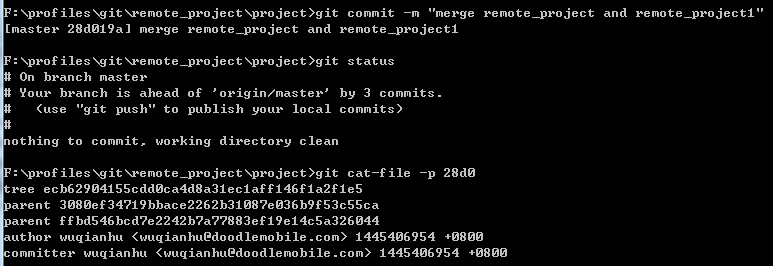
git merge

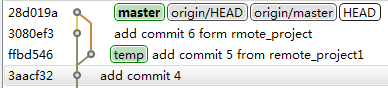
先fetch后merge相当于pull

我们其实可以先fetch远程服务器的数据变为一个临时的分支，饭后比较当前分支与临时分支的差别，再进行合并



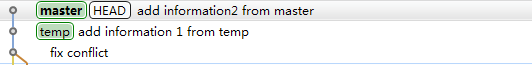
采用fetch和merge的方式，发现当前的commit有两个parent





如果采用rebase合并两个分支





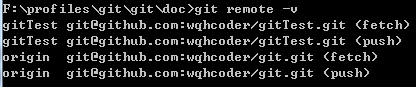
**远程仓库的使用**

要查看当前配置有哪些远程仓库，可以用git remote命令，它会列出每个远程库的简短名字。在克隆完某个项目后，至少看到一个名为origin的远程库，git默认使用这个名字来标识你所克隆的原始仓库

也可以加上-v选项（verbose的简写），显示对应的克隆地址：



如果有多个远程仓库，此命令将全部列出，比如



**添加远程仓库**

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来使用，运行git remote add [shortname] [url]：

**从远程仓库中抓取数据**

git fetch [remote-name]

此命令会到远程仓库中拉去所有你本地仓库中还没有的数据，运行完成后，你就可以再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，或者只是区区某个分支。

需要注意的是，fetch命令只是将远程的数据拉到本地仓库，并不自动合并到当前的工作分支，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程分支，可以使用git pull命令自动抓取数据下来，然后将远程分支自动合并到本地仓库中当前分值。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（假设远程仓库中确实有master分支）。所以我们一般运行git pull，目的都是要从原始克隆的远程仓库中抓取数据后，合并到工作目录中的当前分支

**推送数据到远程仓库**

git push [remate-name] [branch-name]

如果要把本地的master分支推送到origin服务器上（再次说明，克隆操作会自动使用默认的master和origin名字），可以运行下面的命令：

git push origin master

只有在所克隆的服务器上有写权限，且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。如果在你推送数据前，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。



比如上面一条命令，如果远程分支没有test分支，则会创建一个test分支

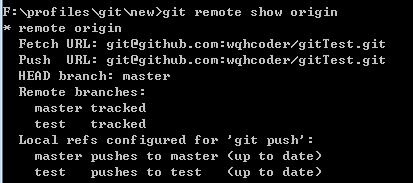
如果远程推送的远程分支不存在，同时本地分支也没有推送的这个分支的名字：



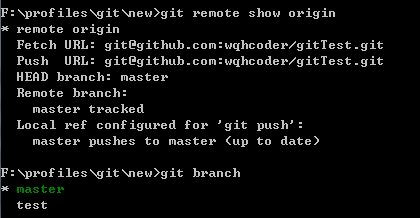
它会告诉你当前推送的分支和本地的分支无法匹配

**查看远程仓库信息**

可以通过git remote show [remote-name]查看某个远程仓库的详细信息，比如要看所克隆的origin仓库，可以运行

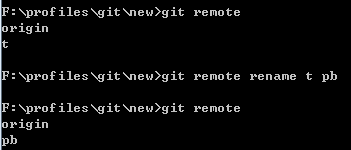


它所显示的是远程分支：



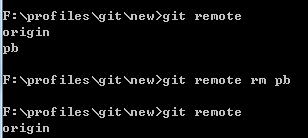
**远程仓库的删除和重命名**

在新版的git中可以用git remote rename命令修改某个远程仓库在本地的简称，



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t/master现在变为了pb/master

碰到远程参股服务区迁移，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程仓库，可以运行git remote rm命令：



删除远程分支：



但是本地的test分支仍然保存没有删除

**4 git分支**

**5 服务器上的git**

远程仓库通常只是一个裸仓库，即一个没有当前工作木库的仓库。因为该仓库只是一个合作媒体，所以不需要从硬盘上取出最新版本的快照；仓库里存放的仅仅是git的数据。简单地说，裸仓库就是你工作目录中.git子目录内的内容

**协议**

Git可以使用四种主要的协议来传输数据：本地传输、SSH协议，Git协议和HTTP协议。

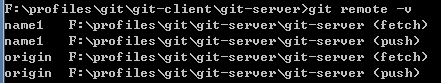
1）本地协议

最基本的就是本地协议，所谓远程仓库在该协议中的表示，就是硬盘上的另一个目录。这常见于团队每一个成员都对一个共享的文件系统拥有访问权，或者比较少见的多人共用同一台电脑的情况，后面一种情况并不安全，因为所有代码仓库实例都存储在同一台电脑里，增加了灾难性数据损失的可能性。



上面的命令即在本地创建了一个本地仓库





优点在于简单，缺点就是不方便，很难控制从不同的位置来访问权限

2）SSH协议

Git使用的传输协议中最常见的可能就是SSH协议了。这是因为大多数环境已经支持通过SSH对服务器的访问。SSH也是唯一一个同时支持读写操作的网络协议。另外两个网络协议（HTTP和Git）通常都是只读的，所以虽然二者对大多数人都可用，但执行写操作时还是需要SSH。SSH同时也是一个验证授权的网络协议；而因为其普遍性，一般架设和使用都很容易。

通过SSH克隆一个git仓库，可以像下面这样给树ssh：//的URL：

git clone ssh://user@server/project.git

或者不指定某个协议，这时git会默认使用SSH

git clone [user@server:project.git](mailto:user@server:project.git)

如果不指明用户，git会默认使用当前登陆的用户名连接服务器

比如：git@github.com:wqhcoder/git.git

**优点：**首先，如果你想拥有对网络仓库的写权利，基本上不可能不使用SSH。其次，SSH架设相对比较简单-SSH守护进程很常见。再次，通过SSH进行访问是安全的-所有数据传输都是加密和授权的，最后，和git及本地协议一样，SSH也是很搞笑，会在传输之前尽可能压缩数据

**缺点：**SSH的限制在于你不能通过它实现仓库的匿名访问。即使仅为读取数据，人们也必须在能通过SSH访问主机的前提下才能访问仓库，这使得SSH不利于开源的项目。如果仅仅在公司网络中使用，SSH可能是你唯一需要使用的协议。如果想允许对项目的匿名只读访问，那么除了为自己推送而假设SSH协议之外，还需要支持其他协议以便他人进行读取

3）git协议

这是一个包含在git软件包中的特殊守护进程；它会箭筒一个提供类似于SSH服务的特定端口（9418），而无须任何授权。它是协议进程提供仓库服务的必要条件，但除此之外该服务没有什么安全措施。要么所有人都能克隆git仓库，要么谁也不能。这也意味着该协议通常不能用来进行推送，你可以允许推送操作；然而由于没有授权机制，一旦允许该操作，网络上任何一个知道项目URL的人将都有推送权限。不用说，这是十分罕见的的情况。

**优点：**git协议是璇存最快的传输协议，它使用与SSH协议相同的数据传输机制，但是省去了加密和授权的开销

**缺点：**缺少授权机制。用git协议作为访问项目的唯一方法通常是不可取的。一般做法是，同时提供SSH接口，让几个开发者拥有推送（写）权限，其他人通过git://拥有只读权限，git协议可能也是最难假设的协议，它要求有单独的守护进程，需要定制。

4）HTTP/S协议

HTTP协议的优美之处在于假设的简便性，基本上只要把git的裸仓库文件放在HTTP的根目录下，配置一个特定的post-update挂钩（hook）就可以搞定。伺候，每个能访问git仓库所在服务器的人都可以进行克隆操作

**生成SSH公匙**

大多数git服务器都会选择使用SSH公匙来进行授权。系统中的每个用户都必须提供一个公匙用于授权，没有的话就要生成一个，SSH公匙默认储存在账户的主目录下的~/.ssh目录中，关键是看有没有用something和something.pub来命名的一对文件，这个something通常就是id\_dsa和id\_rsa。有.pub后缀的文件就是公匙，另一个文件则是密匙。假如没有这些文件，或者干脆连.ssh目录都没有，可以用ssh-keygen来创建。该程序在linux/mac系统上由SSH包提供，而在windows上则包含在MSysGit包里。

命令行：ssh-keygen

它先要求你确认保存公匙的位置，然后它会让你重复一个密码两次，如果不想在使用公匙的时候输入密码，可以留空

现在，所有做过这一步的用户都得把他们的公匙给你或者git服务器管理员（架设SSH服务被设定为使用公匙机制）。他们只需要复制.pub文件的内容然后发邮件给管理员。

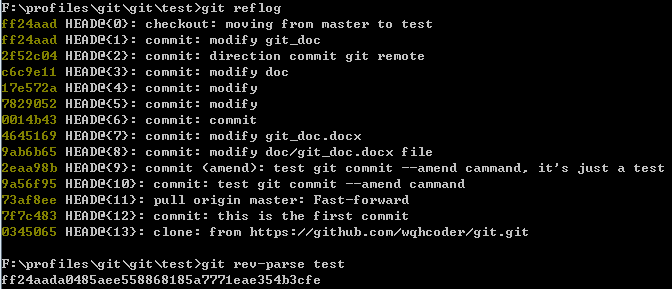
将所有用户的公匙保存在authorized\_keys文件的做法只能凑合一阵子，当用户数量达到几百人的规模时，管理起来就会很痛苦。每次改删用户都必须登录服务器不说，这种做法还缺少必要的权限管理，每个人都对所有项目拥有完整的读写权限，幸好我们我们可以选择应用广泛的gitosis项目。简单地说，gitosis就是一套用来管理authorized\_keys文件和时间简单连接限制的脚本。有趣的是，用来添加用户和设定权限的并非通过网页程序，而只是管理一个特殊的git仓库，你只需要在这个特殊仓库中做好相应的设定，然后推送到服务器上，gitosis就会随之改变运行策略。

**6 分布式git**

**7 git工具**

**修订版本的选择**

分支引用：如果想知道某个分支指向哪个特定的SHA，或者想看任何一个例子被简写的SHA-1，你可以使用一个叫做rev-parse的Git探测工具，该命令是为了底层操作而不是日常操作设计的，不过，有时你想看Git现在到底处于什么状态时，它可能会很有用



如果我们想查看仓库中HEAD在三次前的值，我们可以使用引用日志的输出中的@{n}

引用：



还可以使用这个语言来查看某个分支在一定时间前的位置，例如，如果你想看你的master分支昨天在哪里，可以输入：



它会显示昨天分支的顶端在哪。这项技术只对还在你医用日志里的数据有用，所以不能用来查看比几个月前还早的提交。

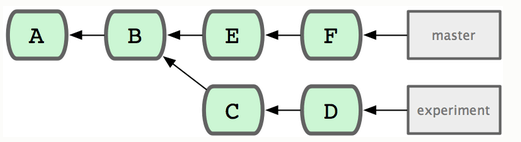
需要注意的是，引用日志信息只存在于本地—-这是一个记录你在自己的仓库中做过什么的日志，其他人拷贝的仓库的引用不会和你的相同；而你新克隆一个仓库时，引用日志是空的，因为你在仓库里还没有操作。git show [HEAD@{2.months.ago}](mailto:HEAD@%7b2.months.ago%7d)这条命令只有在你克隆了一个项目至少两个月时才会有用—-如果你是五分钟前克隆的仓库，那么它就不会有返回

**祖先引用**

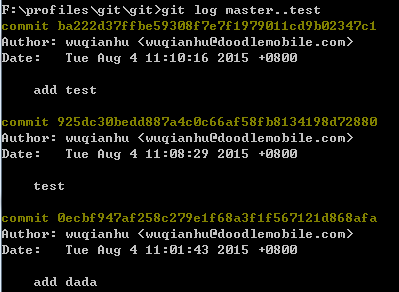
另外一种指明某次提交的常用方法是通过它的祖先。如果你在引用最后加上一个^，git将其理解为此次提交的父提交

**提交范围**

最常用的的指明范围的方法是双点语法，这种语法主要是让git区分出从一个分支中获得而不能从另一个分支获取的提交。假设有类似于下图的历史提交：



如果你想查看实验分支上哪些没有被提交到主分支，那么就可以使用master..experiment来放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



另一方面，如果你想看相反的—-所有在master而不在experiment中分支—

-你可以交换分支的名字,experiment..master显示所有可在master获得而不在experiment中不能的提交



**交互式暂存**

**储藏**

经常有这样的事情发生，当你正在进行某一部分工作，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，否则以后无法回到这个工作点，解决这个问题的办法就是git stash命令

储藏可以获取工作目录的中间状态—也就是修改过的被追踪的文件和暂存的变更—并将它保存到一个未完结变更的堆栈中，随时可以重新应用

git stash 储藏，从最近的一次提交中读取相关内容，让工作区和上次提交的内容一致，同时，将当前的工作区内容保存到git栈中，会发现工作区变成了上一次提交的内容

git stash apply 重新应用刚才的储藏，如果想应用更早的厨房，可以通过名字指定它，类似于这样git stash apply stash@{2}，如果不知名，git默认使用最近的储藏并尝试应用它

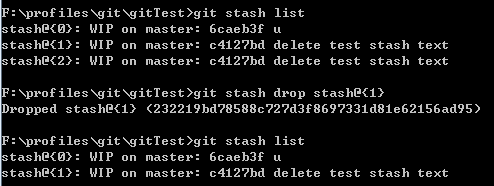




git stash发生了冲突，可以解决冲突之后，add到暂存区，然后commit到代码库中

git stash list 显示储藏列表

git stash drop 移除储藏在栈上的内容，如果希望移除某一个储藏，在后面加上名字



git stash pop 应用储藏，并同时立刻将其从堆栈中移除

**重写历史**

**改变最近一次提交**：git commit --amend

这样会进入文本编辑器，里面包含了最近一次提交说明

如果完成提交后又想修改被提交的快照，增加或者修改其中的文件，可能因为最初提交时，忘了添加一个新建的文件，这个过程基本上一样，通过修改文件然后对其运行git add或对一个已被记录的文件运行git rm，随后的git commit –ament会获取你当前的暂存区并将它作为新提交对应的快照

使用这项技术的时候必须小心，因为修正会改变提交的SHA-1值，这个很像是一次小的rebase---不要再你最近一次提交被推送后还去修正它

**修改多次提交说明**

**Git内部原理**

Git的核心文件：

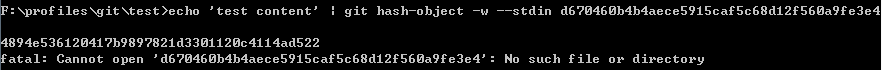
HEAD：指向当前分支

Index：保存了暂存区信息

Objects：存储所有数据内容

Refs：存储指向数据（分支）的提交对象的指针

Git是一套内容寻址的文件系统，这种说法的意思是，从内部来看，git是简单的key-value数据存储。它允许插入任意类型的内容，并返回一个键值，可以用底层命令hash-object命令来示范这一点

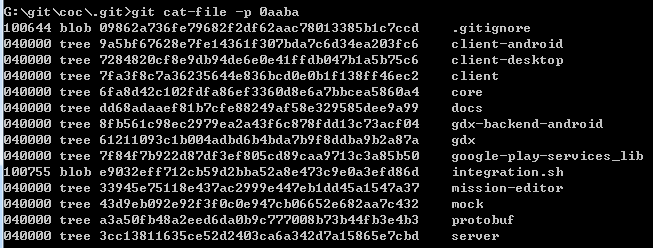


发现object文件夹中创建了一个文件，这便是git存储数据内容的方式—为每一份内容生成一个文件，取得该内容与头信息的SHA-1检验和，创建以该检验和前两个字符为名称的子目录，并以（检验和）剩下38个字符为文件命名（保存至子目录下）



通过cat-file可以将数据内容取回，该命令是查看git对象的瑞士军刀，

Coc项目中的一个例子：



同时我们可以通过参数-t返回任何对象的类型



当我们执行add进入暂存区时，objects文件夹下就多了一个文件

Git中的数据类型 blob tree commit

Tag对象

Tag对象非常像一个commit对象—包含一个标签、一组数据、一个消息和一个指针，对主要的区别是tag对象指向一个commit而不是一个tree。它就像一个分支引用，但是不会变化—永远指向同一个commit，仅仅是提供一个更加友好的名字

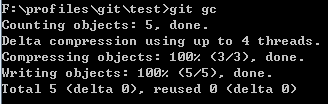
Remotes

远程引用，如果添加一个remote然后推送过去，git会将最后一次推送到这个remote的每个分支的值都记录在refs/remotes目录下，remote应用和分支的主要区别在于他们是不能被checkout的，git把他们当做是标记这些了的这些分支在服务器上最后状态的一种书签。

Packfiles

即使我们只是修改大文件的一小部分，git会用一个全新的对象来保存新的文件内容，这样磁盘上就有了两个几乎相同的文件，如果git只保存其中一个，并保存另一个对象的差异内容，岂不更好？

事实上，git可以这么做，git往磁盘保存对象时默认使用的格式叫做松散对象格式，git是不是地将这些对象打包至一个叫packfile的二进制文件以节省空间并提高效率，当仓库中有太多松散对象时，或是手工调用git gc命令。或推送至远程服务器时，git都会这么做。



会发现pack包中多出了两个新文件pack-628acf2183194addef41a5abe63315d4e8cea603.idx与pack-628acf2183194addef41a5abe63315d4e8cea603.pack

其中pack文件包含了刚才从文件系统中移除的所有对象，索引文件包含了pack文件的偏移信息，这样就可以快速定位任意一个指定对象，有意思的是运行git gc命令前磁盘上的对象大约为12k，而这个新生成的packfile仅为6k，git是如何做到这一点的呢？

Git打包对象时，会查找命名及尺寸相近的文件，并只保存文件不同版本之间的差异内容。