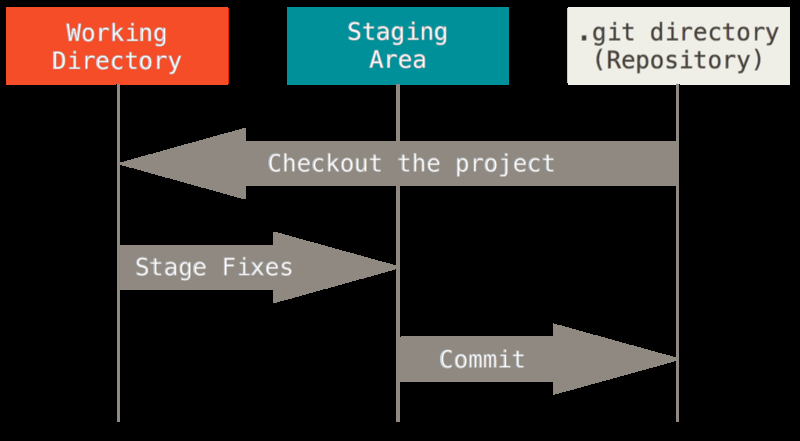
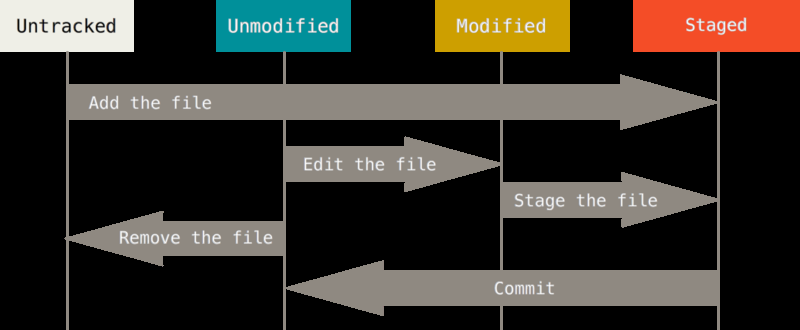
# Git学习笔记

1. **Git具备的特点：**
   1. **快速**
   2. **开发简单**
   3. **对非线性开发模式的强力支持（允许成千上万个并行开发的分支）**
   4. **完全分布式**
   5. **有能力高效管理类似linux内核的超大规模项目（速度和数据量）**
2. **Git的三种状态：**
   1. **已提交（committed）：数据已经安全的保存在本地数据库中。**
   2. **已修改（modified）：修改了文件还没有保存到数据库中**
   3. **已暂存（staged)：对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。**
3. **Git项目的三个工作区域：**



* 1. **Git仓库（Repository）：是git用来保存项目的元数据和对象数据库的地方。从其他计算机克隆仓库时，拷贝的就是这里的数据。**
  2. **工作目录：是对项目的某个版本独立提取出来的内容。这些从Git仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。**
  3. **暂存区域：是一个文件，保存了下次将提交的文件列表信息，一般在Git仓库目录中。有时又被称作‘索引’。**

1. **Git基础**
   1. **git init：在现有目录中初始化Git仓库**
   2. **git add [file name]：添加内容到下一次提交中（开始跟踪新文件、把已跟踪的文件放到暂存区、合并时把有冲突的文件标记为已解决）**
   3. **使用git时文件的生命周期：**



* 1. **git status查看当前文件状态：git status -s 或git status --short 将得到一种更加紧凑的格式输出**

$ git status -s

M README

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

??：表示新添加的未跟踪文件

A：表示新添加到暂存区中的文件

M：出现在右边的M表示文件被修改了但是还没有放入暂存区，出现在左边的M表示文件被修改了并放入了暂存区。

* 1. **忽略文件.gitignore，格式规范如下:**
     1. 所有空行或者以#开头的行都会被Git忽略。
     2. 可以使用标准的glob模式匹配。
     3. 匹配模式可以以（/）开头防止递归。
     4. 匹配模式可以以（/）结尾指定目录。
     5. 要忽略指定模式以外的文件或者目录，可以在模式前加上！取反。eg:

# no .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf

* 1. **git diff：查看尚未暂存的文件更新了哪些部分，git diff --staged：查看已暂存的将要添加到下次提交里的内容**
  2. **git commit -a -m：自动把所有已经跟踪过的文件暂存起来一并提交。**
  3. **git rm：移除文件。如果删除之前文件修改过并且已经放到了暂存区，就必须用强制删除选项-f。如果想把文件从Git仓库中删除（亦从暂存区域移除），但人希望保存在当前工作目录中，换句话说，想让文件保留在磁盘，但并不想让Git继续跟踪。使用--cached选项。**
  4. **git mv：移动文件（可以修改名称）**
  5. **git log：查看提交历史。常用选项：**







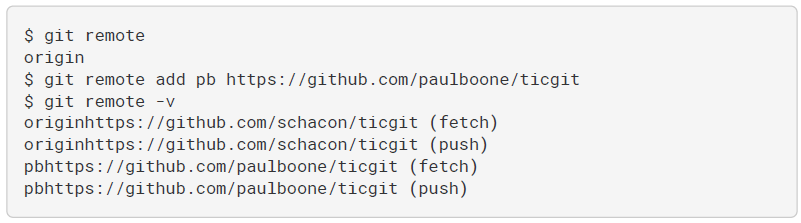
* 1. **git commit --amend：提交完了发现漏掉了几个文件没有添加，或者添加信息写错了。此时此命令可以尝试重新提交：**

$ git commit -m 'initial commit'

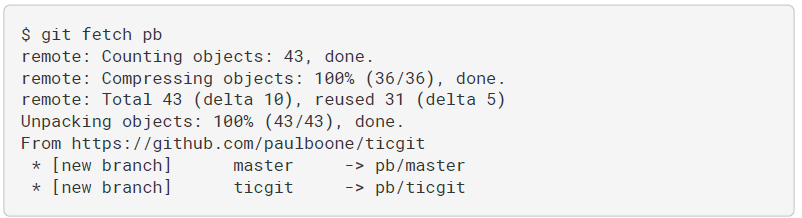
$ git add forgotten\_file

$ git commit --amend

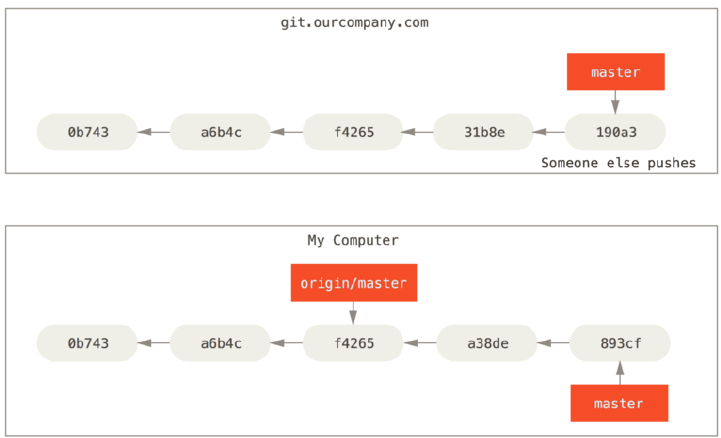
* 1. **git reset HEAD <file>：取消暂存的文件**
  2. **git checkout -- <file>：撤销对文件的修改（这是一个危险的命令，你对那个文件做的任何修改都将消失）**
  3. **在Git中任何已提交的东西几乎都是可以恢复的，甚至那些被删除的分支中的提交或使用--amend选项覆盖的提交也是可以恢复的。然而，任何未提交的东西丢失后很有可能再也找不回来。**
  4. **git remote：查看已经配置的远程仓库服务器。-v选项会显示需要读写的远程仓库使用的Git保存的简写与其对应的URL，如果你的远程仓库不止一个，该命令会将他们全部列出。**
  5. **git remote add <shortname> <url>：添加一个新的远程Git仓库，同事指定一个你可以轻松引用的简写：**

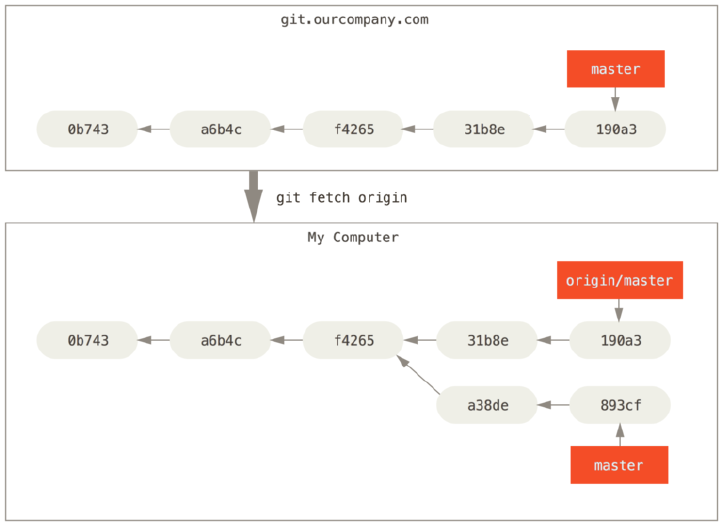


可以在命令行中使用字符串pb来代替整个URL。例如，如果想要拉取Paul的仓库中有但你没有的信息，可以运行git fetch pb:



* 1. **git fetch [remote-name]：从远程仓库中获得数据，这个命令会访问远程仓库，从中拉取所有你还没有的数据。执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。如果你使用clone命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以“origin”为简写。所以，git fetch origin 会抓取克隆（或上一次抓取）后新推送的所有工作。必须注意git fetch命令会将数据拉取到你的本地仓库-它并不会自动合并或修改你当前的工作。当准备好时你必须手动将其合并入你的工作。**
  2. **如果有一个分支设置为跟踪一个远程分支，可以使用git pull命令来自动的抓取然后合并远程分支到当前分支。默认情况下，git clone命令会自动设置本地master分支跟踪克隆的远程仓库的master分支。运行git pull通常会从最初克隆的服务器上抓取数据并自动合并到当前所在的分支。**
  3. **git push [remote-name] [branch-name]：推送到远程仓库。**
  4. **git remote show [remote-name]：查看远程仓库的更多信息**
  5. **git remote rename：修改一个远程仓库的简写名称**
  6. **git remote rm [remote-name]：移除一个远程仓库**
  7. **git tag：列出已有的标签，git tag -l ‘v1.8.5\*’，可以查找匹配的标签**
  8. **Git使用两种主要类型的标签：**
     1. **轻量标签（lightweight)：一个轻量标签很像一个不会改变的分支-它只是一个特定提交的引用。**
     2. **附注标签（annotated)：是存储在Git数据库中的一个完整对象。**
  9. **git tag -a v1.4 -m ‘my version 1.4’ 创建一个附注标签。**
  10. **git show：查看标签信息与对应的提交信息。**
  11. **git tag v1.4-lw：创建一个轻量标签**
  12. **git tag -a v1.2 9fceb02：后期打标签**
  13. **默认情况下，git push命令并不会传送标签到远程仓库服务器。你可以运行git push origin [tagname]来显示的推送标签到共享服务器上。**
  14. **git branch testing：创建一个新分支**
  15. **git log --decorate：查看各个分支当前所指向的对象**
  16. **git checkout testing：切换到已存在的分支，加上-b参数会新建一个分支同时切换到这个分支上**
  17. **git merge hotfix：把hotfix合并到当前分支**
  18. **git branch -d hotfix：删除分支**
  19. **git checkout -b [branch] [remotename]/[branch]：快捷方式为git checkout --track。跟踪分支**
  20. **git fetch origin:会查找‘origin’是哪一个服务器，从中抓取本地没有的数据，并且更新本地数据库，移动origin/master指针指向新的、更新后的位置。**





* 1. **git push (remote) (branch)：当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。本地的分支并不会自动与远程仓库同步，你必须显示的推送想要分享的分支。eg:git push origin serverfix**
  2. **当其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支origin/serverfix,指向服务器的serverfix分支的引用，要特别注意的是，当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本。换句话说，这种情况下不会有一个新的serverfix分支，只有一个不可以修改的origin/serverfix指针。可以运行git merge origin/serverfix 将这些工作合并到所在的分支。如果想要在自己的serverfix分支上工作，可以将其建立在远程跟踪分支上：git checkout -b serverfix origin/serverfix，这会给你一个用于工作的本地分支，并且起点位于origin/serverfix。**
  3. **git checkout --track origin/serverfix: git checkout -b [branch] [remotename] /[branch]的快捷方式**
  4. **设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，你可以在任意时间使用-u或者--set-upstream-to选项运行git branch来显示的设置：git branch -u origin/serverfix.**
  5. **git branch --v:查看设置的所有跟踪分支。**
  6. **git push origin --delete serverfix:删除远程分支**

**变基：使用rebase命令将提交到某一分支上的所有修改都移至另一个分支上。** $ git checkout experiment

$ git rebase master

$ git checkout master

$ git merge experiment

* 1. **git rebase --onto master server client:取出client分支，找出处于client分支和server分支的共同祖先之后的修改，然后把他们在master分支上重演一遍。**
  2. **变基的准则：不要对在你的仓库外有副本的分支执行变基。**
  3. **只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作。**

1. **服务器上的Git**
   1. **Git可以使用四种主要的协议来传输资料：本地协议（Local），HTTP协议，SSH（Secure Shell）协议及Git协议。**
   2. **HTTP协议的优点：不同的访问只需要一个URL以及服务器只在需要授权时提示输入授权信息，这两个简便性让终端用户使用Git变得非常简单。相比SSH协议，可以使用用户名/密码授权是一个很大的又是，这样用户就不必须在使用Git之前先在本地生成SSH密钥对再把公钥上传到服务器。对非资深的使用者，或者系统上缺少SSH相关程序的使用者，HTTP协议的可用性是主要的优势。**
   3. **SSH协议的优势：SSH架设相对简单 ——SSH守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。其次，通过SSH访问是安全的——所有传输的数据都要经过授权加密。与HTTP/S协议、Git协议与本地协议一样，SSH协议很高效，在传输前也会尽量压缩数据。**
   4. **SSH协议的缺点：不能通过他实现匿名访问。即便只要读取数据，使用者也要有通过SSH访问你的主机的权限，这使得SSH协议不利于开源的项目。**
   5. **Git协议的优点：是Git使用的网络传输协议里最快的。**
   6. **Git协议的缺点：缺乏授权机制。把Git作为访问项目版本库的唯一手段实不可取的。一般的做法里，会同时提供SSH或者HTTPS协议的访问服务，只让少数几个开发者有推送权限，其他人通过git://访问只有只读权限。Git协议也许是最难架设的。他要求有自己的守护进程，还要求开放防火墙9418端口，但是企业防火墙一般不会开放这个非标准端口。而大型的企业防火墙通常会封锁这个端口。**
   7. **上次看到113**