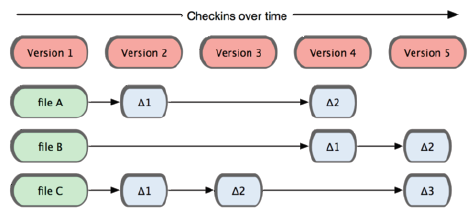
Git使用说明

管理控制发展：

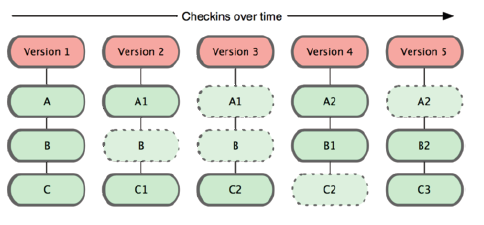
1. 本地版本控制系统
2. 集中化版本控制系统，代表为CVS（Concurrent Version System），以及Subversion等，多年以来这已成为版本控制系统的标准做法，这种做法的缺陷在于服务器的单点问题，服务器的宕机会影响到所有的版本。
3. 分布式版本控制，如Git，客户端并不只是提取最新文件的快照，而是把原始的代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的提取操作，实际上都是一次对代码仓库的完整备份

在开始学习Git 的时候，请不要尝试把各种概念和其他的版本控制系统诸如Subversion 和Perforce 等相比拟，否则容易混淆每个操作的实际意义。Git在保存和处理各种信息的时候，虽然操作起来的命令形式非常相近，但它与其他版本控制系统的做法颇为不同。理解这些差异将有助于你准确地使用Git 提供的各种工具。

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作了更新，以及都更新了哪些行的什么内容。



Git 并不保存这些前后变化的差异数据。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，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照的索引。为提高性能，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连接。



这是Git 同其他系统的重要区别。它完全颠覆了传统版本控制的套路，并对各个环节的实现方式作了新的设计。Git 更像是个小型的文件系统，但它同时还提供了许多以此为基础的超强工具，而不只是一个简单的VCS。

对于任何一个文件，在Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。已提交表示该文件已经被安全地保存在本地数据库中了；已修改表示修改了某个文件，但还没有提交保存；已暂存表示把已修改的文件放在下次提交时要保存的清单中。

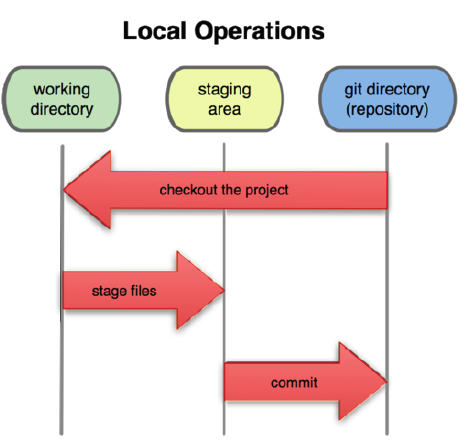
由此我们看到Git 管理项目时，文件流转的三个工作区域：Git 的本地数据目录，工作目录以及暂存区域。每个项目都有一个git 目录，它是Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。

基本的Git 工作流程如下所示：

1. 在工作目录中修改某些文件。

2. 对这些修改了的文件作快照，并保存到暂存区域。

3. 提交更新，将保存在暂存区域的文件快照转储到git 目录中。



红色=当前工作分支所处分支；

绿色=本地分支；

粉红色=服务器上分支；

Git常见命令

|  |  |  |
| --- | --- | --- |
| 命令 | 命令说明 |  |
| git init | 创建本地仓库 |  |
| Git add | 文件添加到仓库 |  |
| Git commit | 文件提交到仓库 |  |
| git commit -m "wrote a readme file" | 文件提交到仓库并且添加此次提交的备注说明 |  |
| Git status | 当前仓库的状态 |  |
| Git diff readme.txt | 查看readme.txt被修改的内容 |  |
| Git log | 显示最近到最晚的提交日志 |  |
| git log --pretty=oneline | 显示更简洁的提交日志 |  |
| git reflog | 显示git的所有操作记录 |  |
| git reset --hard HEAD^ | 恢复上一个版本，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交3628164...882e1e0（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。 |  |
| git reset --hard 12456 | 回退到12456对应的版本 |  |
| git checkout -- file | 直接丢弃工作区的修改，当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。  **其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。** |  |
| git remote add origin git@github.com:michaelliao/learngit.git | 与远程仓库建立关联 |  |
| git push -u origin master | 将本地内容推送到远程，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。 |  |
| git push -u origin master -f | 当远程repository和本地的repository存在冲突，不管远程，强制将本地的版本提交到远程，这样会使远程修改丢失，一般是不可取的，尤其是多人协作开发的时候，另外一种解决方案是将远程的版本pull下来，如果不希望和本地的进行merge可以新建一个分支，然后再将本地push到远程。 |  |
| git push origin master | 日常的提交 |  |
| git clone git@github.com:michaelliao/gitskills.git | 从远程克隆一份 |  |
| git checkout -b dev | git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：  git branch dev –创建分支  git checkout dev—切换分支 |  |
| git branch | git branch命令会列出所有分支，当前分支前面会标一个\*号。 |  |
| git merge dev | 把dev分支的工作成果合并到master分支上，git merge命令用于合并指定分支到当前分支 |  |
| git branch -d dev | 删除分支 |  |
| git branch -D dev | 强行删除分支 |  |
| git stash | 保存工作现场 |  |
| git stash list | 查看存储的工作现场 |  |
| git stash apply | 恢复工作现场，配合git stash drop使用 |  |
| git stash pop | 同时恢复工作现场 |  |
| git remote -v | 查看远程库信息 |  |
| git push origin dev | 推送本地分支 |  |
| git checkout -b dev origin/dev | 在本地创建dev分支，对应到远程的dev分支 |  |
| git pull | 把最新的提交从origin/dev抓下来 |  |
| git tag v1.0 | 给分支打上标签 |  |
| git tag git tag | 查看已经打上分支的标签 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |





Pycharm使用git：

首先要将本地与远程github进行关联，点击菜单栏的VCS，选择check out from version control，即可将远程github与本地进行关联，本地也会程成已给.git文件夹。

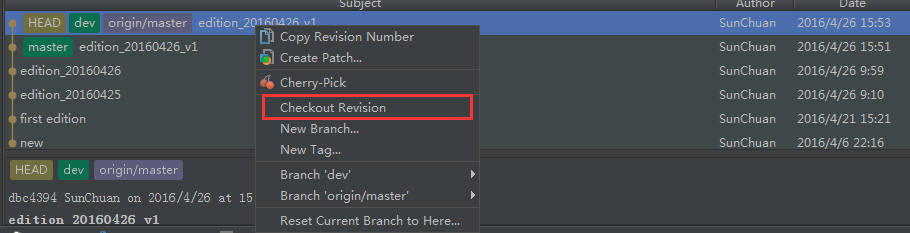
然后可以将文件添加到git中，在文件上右键选择git然后选择添加，之后再使用commit

添加到git的文件名呈现蓝色，而没有添加到git中的文件名呈现红色。

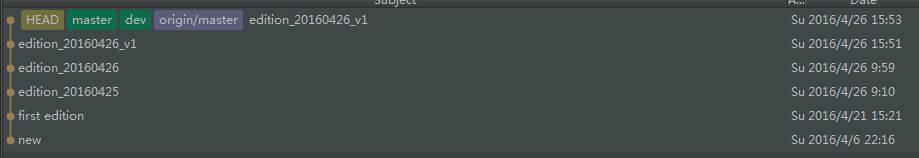
创建本地的git仓库可以通过VCS | Import into Version Control | Create Git Repository进行本地仓库的创建。

将本地的仓库上传到github可以使用VCS | Import into Version Control | Share Project on GitHub进行文件的上传。

Gist作为github推出的一个方便黏贴文本的记录工具

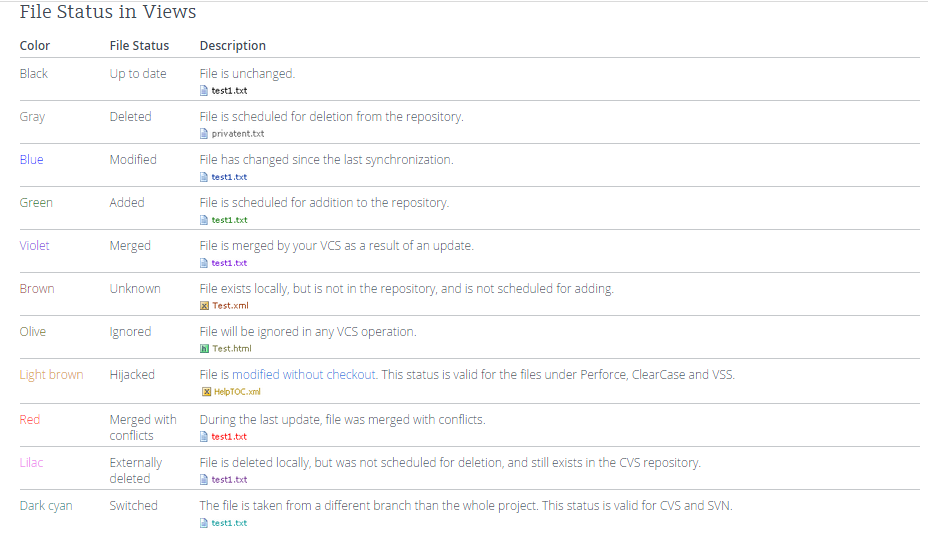
Pycharm中使用checkout Revision进行分支的切换，但是使用过程中发现有点问题，在分支切换的过程中经常会切换到一个head detached分支（游离分支），尚不清楚具体原因。

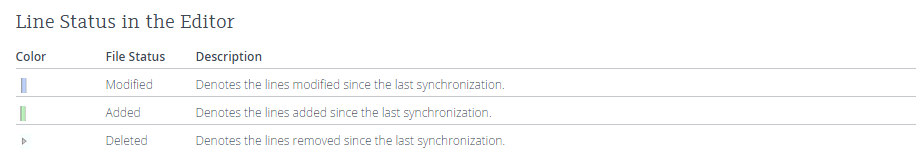
使用reset current branch to here相当于执行了git reset –hard 123456命令



Git fetch 和git pull的区别在于pull相当于在fetch的基础上加了一个merge的动作

上述分支图表示当前有两个分支，master、dev分支，head指向当前的状态，head必须指向一个分支头上





心理一定要有个明确的分支图

