**Git系统原理及管理机制**

## Git简介

Git就是一种分布式版本控制系统。那么什么是版本控制呢？版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。专业来说，版本控制就是对文件变更过程的管理，简单来说，版本控制就是要把一个文件或一些文件的各个版本按一定的方式管理起来，目的是需要用到某个版本的时候可以随时拿出来。在实际应用中，Git能够为我们解决版本控制方面的大多数问题，如：

●可以为每一次变更提交版本更新并且备注更新的内容；

●我们可以在项目的各个历史版本之间自如切换；

●可以一目了然的比较出两个版本之间的差异；

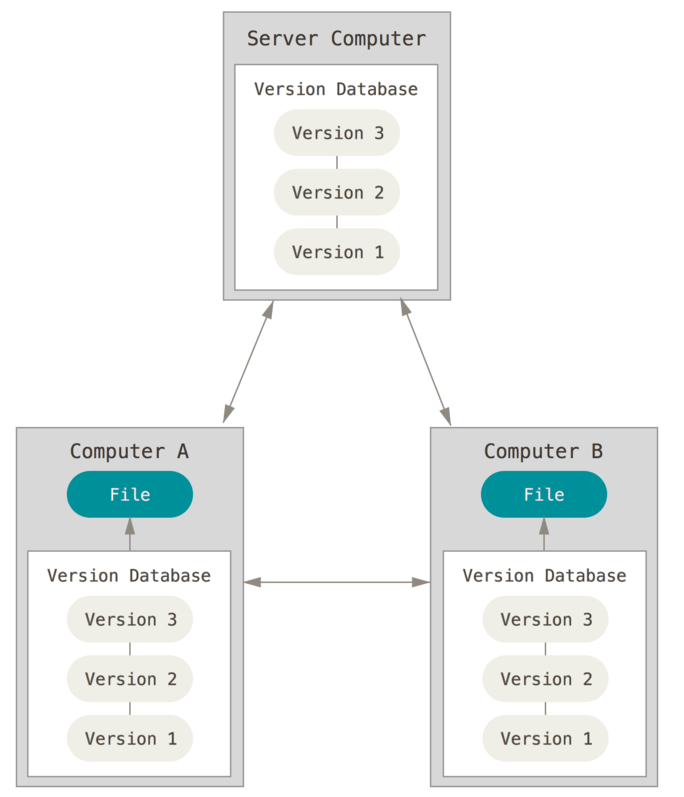
●可以从当前的修改中撤销一些操作；

●可以自如的创建分支、合并分支；

●可以和多人协作开发；

●可以采取自由多样的开发模式。

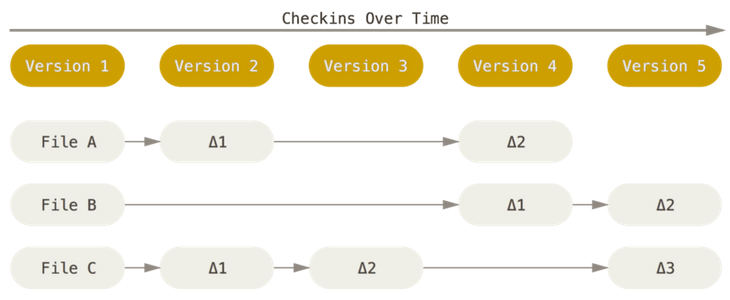
在分布式版本控制系统中，像Git、Mercurial、Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。 因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。



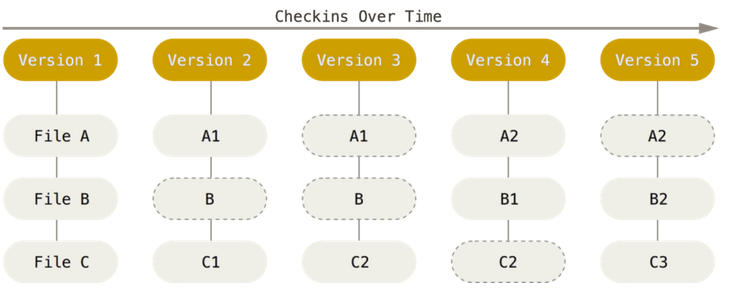
## 2 Git工作原理

### 2.1 Git对数据的处理

Git 和其它版本控制系统(包括 Subversion 和近似工具)的主要差别在于 Git 对待数据的方法。 概念上来区分，其它大部分系统以文件变更列表的方式存储信息。 这类系统(CVS、Subversion、Perforce、Bazaar 等等)将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。存储每个文件与初始版本的差异，如下图所示：



Git 不按照以上方式对待或保存数据。反之，Git 更像是把数据看作是对小型文件系统的一组快照。 每次你提交更新，或在Git中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个快照流。如下图所示：

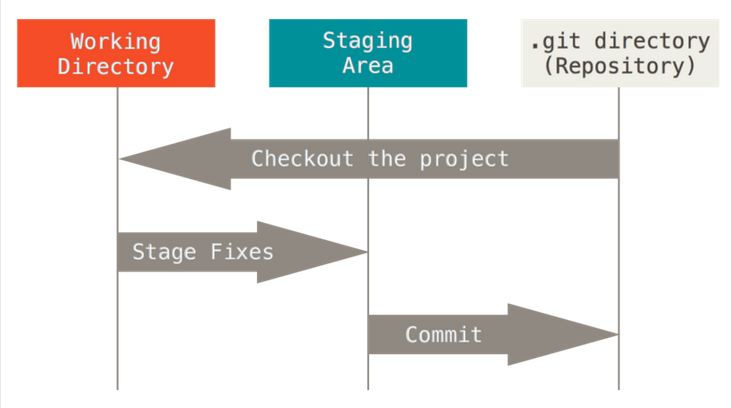


Git 重新考虑了以前每一代版本控制系统延续下来的诸多方面，使他更像是一个小型的文件系统，提供了许多以此为基础构建的超强工具，而不只是一个简单的 VCS。

在执行的 Git 操作时，几乎只往 Git 数据库中增加数据，很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。一旦你提交快照到Git 中，就难以再丢失数据，特别是如果定期的推送数据库到其它仓库的话。这使得我们使用 Git 成为一个安心愉悦的过程，因为我们深知可以尽情做各种尝试，而没有把事情弄糟的危险。 而且Git 中所有数据在存储前都计算校验和，然后以校验和来引用。这意味着不可能在 Git 不知情时更改任何文件内容或目录内容，这个功能建构在 Git 底层，是构成 Git中不可或缺的部分。

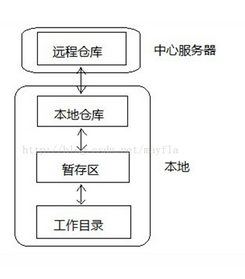
### 2.2 Git存储状态

Git 有三种状态：已提交(committed)、已修改(modified)和已暂存(staged)。 已提交表示数据已经安全的保存在本地数据库中。 已修改表示修改了文件，但还没保存到数据库中。 已暂存表示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。由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。工作目录、暂存区域以及 Git 仓库，如下图所示：



Git 仓库目录是 Git 用来保存项目的元数据和对象数据库的地方。这是 Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。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。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。 有时候也被称作‘索引’，不过一般说法还是叫暂存区域。如果 Git 目录中保存着的特定版本文件，就属于已提交状态。 如果作了修改并已放入暂存区域，就属于已暂存状态。 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

Git的工作总共分四层，其中三层是在自己本地也就是说Git仓库，包括了工作目录、暂存区和本地仓库。工作目录就是我们执行命令git init时所在的地方，也就是我们执行一切文件操作的地方；暂存区和本地仓库都是在.git目录下，因为它们只是用来存储数据的。远程仓库在中心服务器，也就是我们做好工作之后推送到远程仓库，或者从远程仓库更新下来最新代码到本地。Git所存储的都是一系列的文件快照，然后Git 来跟踪这些文件快照，发现哪个文件快照有变化他就会提示你需要添加到暂存区或是提交到本地仓库来保证你的工作目录是干净的。如下图所示为Git的工作层：



## 3 创建Git仓库

什么是版本库呢？版本库又名仓库，英文名repository，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。有两种取得 Git 项目仓库的方法。第一种是在现存的目录下，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来。

### 3.1 在工作目录中初始化新仓库

通过git init命令把目录变成Git可以管理的仓库，到此项目所在的目录，执行：

$ git init

初始化后，在当前目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。不过目前，仅仅是按照既有的结构框架初始化好了里边所有的文件和目录，但我们还没有开始跟踪管理项目中的任何一个文件。如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交。

### 3.2 从现有仓库克隆

如果想对某个开源项目出一份力，可以先把该项目的 Git 仓库复制一份出来，这就需要用到 git clone 命令。Git 收取的是项目历史的所有数据（每一个文件的每一个版本），服务器上有的数据克隆之后本地也都有了。实际上，即便服务器的磁盘发生故障，用任何一个克隆出来的客户端都可以重建服务器上的仓库，回到当初克隆时的状态（虽然可能会丢失某些服务器端的挂钩设置，但所有版本的数据仍旧还在）。克隆仓库的命令格式为：

$ git clone [url]

## 4 Git分支

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。在很多版本控制系统中，这是个昂贵的过程，常常需要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。

有人把 Git 的分支模型称为“必杀技特性”，而正是因为它，将 Git 从版本控制系统家族里区分出来。Git 有何特别之处呢？Git 的分支可谓是难以置信的轻量级，它的新建操作几乎可以在瞬间完成，并且在不同分支间切换起来也差不多一样快。和许多其他版本控制系统不同，Git 鼓励在工作流程中频繁使用分支与合并，哪怕一天之内进行许多次都没有关系。理解分支的概念并熟练运用后，你才会意识到为什么 Git 是一个如此强大而独特的工具，并从此真正改变你的开发方式。

### 4.1 何谓分支

Git存储数据时，保存的不是文件差异或者变化量，而只是一系列文件快照。在 Git 中提交时，会保存一个提交（commit）对象，该对象包含一个指向暂存内容快照的指针，包含本次提交的作者等相关附属信息，包含零个或多个指向该提交对象的父对象指针：首次提交是没有直接祖先的，普通提交有一个祖先，由两个或多个分支合并产生的提交则有多个祖先。

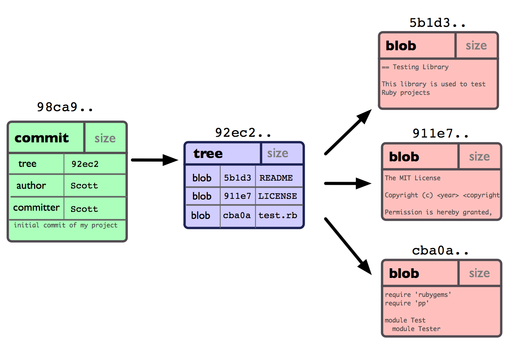
为直观起见，我们假设在工作目录中有三个文件，准备将它们暂存后提交。暂存操作会对每一个文件计算校验和，然后把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 类型的对象存储这些快照），并将校验和加入暂存区域，具体操作如下：

$ git add README test.rb LICENSE

$ git commit -m 'initial commit of my project

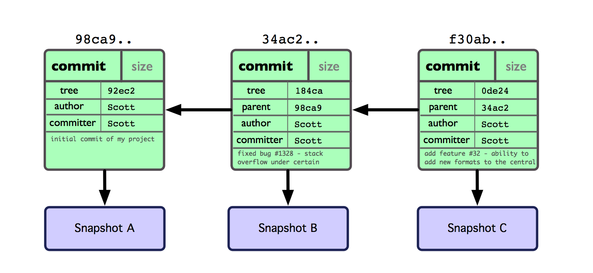
当使用 git commit 新建一个提交对象前，Git 会先计算每一个子目录的校验和，然后在 Git 仓库中将这些目录保存为树（tree）对象。之后 Git 创建的提交对象，除了包含相关提交信息以外，还包含着指向这个树对象（项目根目录）的指针，如此它就可以在将来需要的时候，重现此次快照的内容了。

现在，Git 仓库中有五个对象：三个表示文件快照内容的 blob 对象；一个记录着目录树内容及其中各个文件对应 blob 对象索引的 tree 对象；以及一个包含指向 tree 对象（根目录）的索引和其他提交信息元数据的 commit 对象。概念上来说，仓库中的各个对象保存的数据和相互关系看起来如图所示：



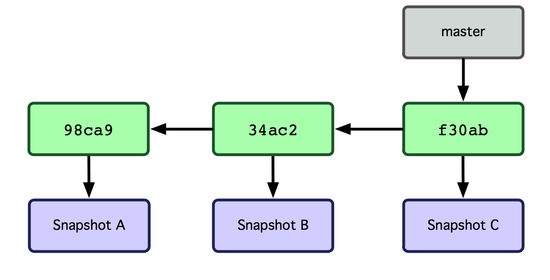
如上图所示，单个提交对象在仓库中的数据结构。

如果作些修改后再次提交，那么这次的提交对象会包含一个指向上次提交对象的指针（下图中的 parent 对象）。两次提交后，仓库历史会变成如下图的样子：



如上图所示为多个提交对象之间的链接关系。

现在来谈分支。Git 中的分支，其实本质上仅仅是个指向 commit 对象的可变指针。Git 会使用 master 作为分支的默认名字。在若干次提交后，你其实已经有了一个指向最后一次提交对象的 master 分支，它在每次提交的时候都会自动向前移动。

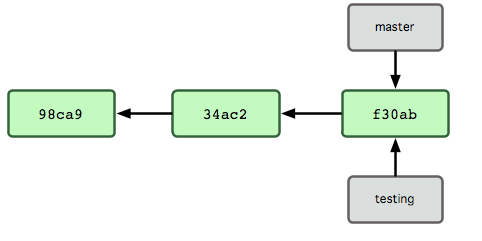


如上图所示，分支其实就是从某个提交对象往回看的历史

那么，Git 又是如何创建一个新的分支的呢？答案很简单，创建一个新的分支指针。比如新建一个 testing 分支，可以使用 git branch 命令：

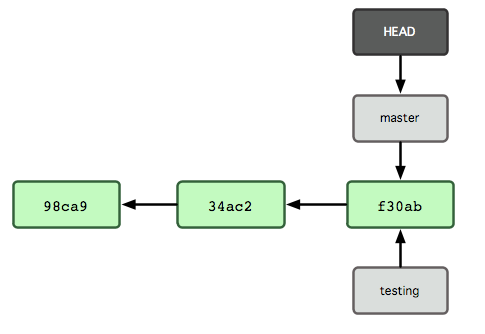
$ git branch testing

这会在当前 commit 对象上新建一个分支指针，如图所示：



如上图所示，多个分支指向提交数据的历史

那么又一个问题来了，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 HEAD 的特别指针。在 Git 中，它是一个指向你正在工作中的本地分支的指针（可将 HEAD 想象为当前分支的别名。）。运行 git branch 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在操作时，我们依然还在 master 分支里工作，如下图所示：

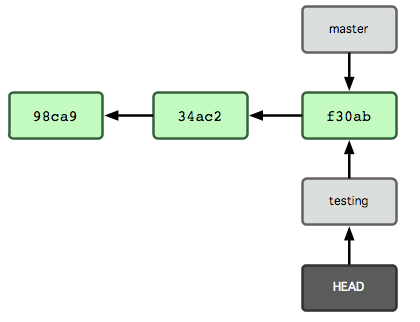


如上图，HEAD 指向当前所在的分支

若要切换到其他分支，可以执行 git checkout 命令。如转换到新建的 testing 分支：

$ git checkout testing

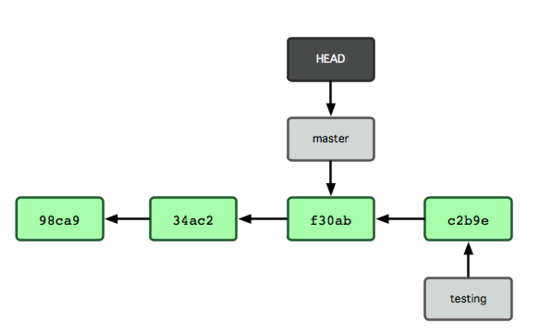
这样 HEAD 就指向了 testing 分支，如下图所示：



如上图，HEAD 在你转换分支时指向新的分支，每次提交后 HEAD 随着分支一起向前移动。

master 分支仍然指向git checkout 时所在的 commit 对象。如果现在我们需要回到 master 分支上去，我们可以执行如下操作：

$ git checkout master



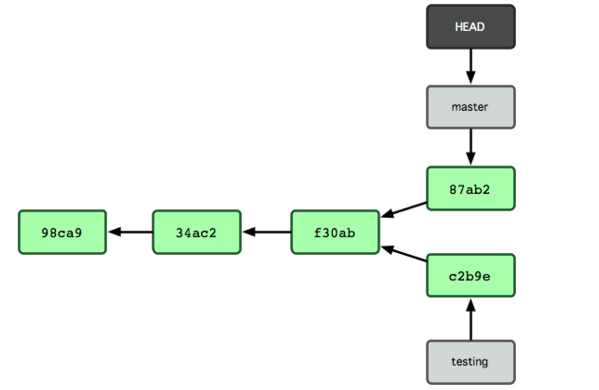
如上图所示，HEAD 在一次 checkout 之后移动到了另一个分支master上。

上条命令做了两件事。它把 HEAD 指针移回到 master 分支，并把工作目录中的文件换成了 master 分支所指向的快照内容。也就是说，现在开始所做的改动，将始于本项目中一个较老的版本。它的主要作用是将 testing 分支里作出的修改暂时取消，这样你就可以向另一个方向进行开发。我们作些修改后再次提交：

$ vim test.rb

$ git commit -a -m 'made other changes'

现在我们的项目提交历史产生了分叉，如下图，因为刚才我们创建了一个分支，转换到其中进行了一些工作，然后又回到原来的主分支进行了另外一些工作。这些改变分别孤立在不同的分支里：我们可以在不同分支里反复切换，并在时机成熟时把它们合并到一起。而所有这些工作，仅仅需要 branch 和 checkout 这两条命令就可以完成。



由于 Git 中的分支实际上仅是一个包含所指对象校验和（40 个字符长度 SHA-1 字串）的文件，所以创建和销毁一个分支就变得非常廉价。说白了，新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）那么简单，当然也就很快了。

这和大多数版本控制系统形成了鲜明对比，它们管理分支大多采取备份所有项目文件到特定目录的方式，所以根据项目文件数量和大小不同，可能花费的时间也会有相当大的差别，快则几秒，慢则数分钟。而 Git 的实现与项目复杂度无关，它永远可以在几毫秒的时间内完成分支的创建和切换。同时，因为每次提交时都记录了祖先信息（即 parent 对象），将来要合并分支时，寻找恰当的合并基础（即共同祖先）的工作其实已经自然而然地摆在那里了，所以实现起来非常容易。

### 4.2 分支的新建与合并

在实际工作和学习中大体会用到这样的工作流程：

（1）开发某个网站。

（2）为实现某个新的需求，创建一个分支。

（3）在这个分支上开展工作。

假设此时，你突然接到一个电话说有个很严重的问题需要紧急修补，那么可以按照下面的方式处理：

（1）返回到原先已经发布到生产服务器上的分支。

（2）为这次紧急修补建立一个新分支，并在其中修复问题。

（3）通过测试后，回到生产服务器所在的分支，将修补分支合并进来，然后再推送到生产服务器上。

（4）切换到之前实现新需求的分支，继续工作。

首先，假设你正在项目中愉快地工作，并且已经提交了几次更新。现在需要修补系统中出现的某一问题，就需要新建一个分支并切换到该分支，运行 git checkout 并加上 -b 参数：

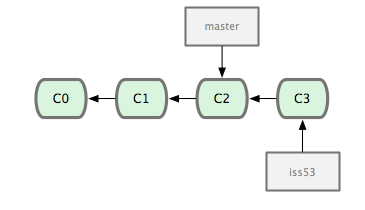
$ git checkout -b dev

上述这条命令相当于执行这两条命令：

$ git branch dev

$ git checkout dev

接着你开始尝试修复问题，在提交了若干次更新后，dev分支的指针也会随着向前推进，因为它就是当前分支，换句话说，当前的 HEAD 指针正指向 dev。



master

dev

有了 Git ，我们就不需要同时发布这个补丁和 dev 里作出的修改，也不需要在创建和发布该补丁到服务器之前花费大力气来复原这些修改。唯一需要的仅仅是切换回 master 分支。

不过在此之前，需要留心你的暂存区或者工作目录里，那些还没有提交的修改，它会和你即将检出的分支产生冲突从而阻止 Git 为你切换分支。切换分支的时候最好保持一个清洁的工作区域。目前已经提交了所有的修改，所以接下来可以正常转换到 master 分支，可以输入如下命令：

$ git checkout master

当我们所有的问题解决完成后，确保修补是成功的，然后需要回到 master 分支并把它们合并进来，然后发布到生产服务器。用 git merge 命令来进行合并，合并后，再查看dev和master里的内容，就可以看到，和dev分支的最新提交是完全一样的。

$ git checkout master

$ git merge dev

提交成功后，我们可以注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。当然，也不是每次合并都能Fast-forward：

### 4.3遇到冲突时的合并

有时候合并操作并不会如此顺利。如果在不同的分支中都修改了同一个文件的同一部分，Git 就无法干净地把两者合到一起。如果你在解决问题的过程中修改了dev中修改的部分，将得到类似下面的结果：

$ git merge dev

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

做了合并，但没有提交，系统会停下来等你解决冲突。要看看哪些文件在合并时发生冲突，可以用 git status 查阅：

$ git status

然后我们可以查看自己编写的文件的内容，任何包含未解决冲突的文件都会以未合并（unmerged）的状态列出。Git 会在有冲突的文件里用<<<<<<<，=======，>>>>>>>标记出不同分支的内容解决标记，可以通过它们来手工定位并解决这些冲突。

### 4.4 分支管理

以上讲述了分支的创建与合并，现在我们需要对分支进行管理，以便在日后的工作中更加的方便。git branch 命令不仅仅能创建和删除分支，如果不加任何参数，它会给出当前所有分支的清单：

$ git branch

dev

\* master

注意看 master 分支前的 \* 字符：它表示当前所在的分支。也就是说，如果现在提交更新，master 分支将随着开发进度前移。若要查看各个分支最后一个提交对象的信息，运行指令 git branch -v

如果要从该清单中筛选出你已经与当前分支合并的分支，可以用 --merge 和 --no-merged 选项。比如用 git branch --merge 查看哪些分支已被并入当前分支：

$ git branch –merged

一般来说，列表中没有 \* 的分支通常都可以用 git branch -d 来删掉。原因很简单，既然已经把它们所包含的工作整合到了其他分支，删掉也不会损失什么。另外可以用 git branch --no-merged 查看尚未合并的工作：

$ git branch --no-merged

## 5 Git常用指令集合

（1）创建git库

$ git init #在当前目录中生成一个.git 目录（含有.git目录的目录即是git仓库）

（2）注册git用户，用于在团队合作开发中，表明代码作者。

$ git config --global user.name XXX #用户名

$ git config --global user.email XXX #用户邮箱

$ git config --list #查看用户信息

注：加--global，全局设置。

（3）向git库添加修改

$ git add [path］ #会把对应目录或文件，添加到stage状态

$ git add . #会把当前所有的untrack files和changed but not updated添加到stage状态实际上是为修改内容添加index索引。

（4）向版本库提交修改

$ git commit –m “XXXX” #提交修改,添加注释

注：git 提示： 未有add红色字体，未有commit绿色字体，已提交则worktree是干净的

（5）查看当前代码库的状态

$ git status

（6）查看版本信息，实际是查看修改提交信息

$ git log

$ git log --graph #以图形化（节点）展示当前git库的提交信息。

（7）查看指定版本信息

$ git show sdjf974654dd…. #(show后面为每次提交系统自动生成的一串哈希值)

$ git show sdji97 #一般只使用版本号的前几个字符即可

（8）撤销修改

$ git reset

（9）撤销整体修改

$ git reset --hard #回到原来编辑的地方,改动会丢失。（同样适用于团队对于其他人的修改恢复）

$ git reset --hard sdv143kvf…... #可回到指定的版本#(hard后面为每次提交系统自动生成的一串哈希值)，git reset [path] 会改变path指定的文件或目录的stage状态，到非stage状态，git reset 会将所有stage的文件状态，都改变成非stage状态。

（10）撤销某次修改

回退1个change的写法就是git reset HEAD^，2个为HEAD^^，3个为HEAD~3，以此类推。

（11）向远端库推送修改（提交修改）

$ git push origin 分支名

10、暂存修改

$ git stash可以把当前的改动（stage和unstage，但不包括untrack的文件）暂存，然后通过git stash list查看，并通过git stash apply重新取出来。但apply之前要保证worktree是干净的。

（12）获取远端库项目

$ git clone/pull

（13）团队开发的基本流程（多分支合并一个分支）

$ git add . #添加改动的文件

$ git commit #（提交至本地）

$ git pull --rebase #（将服务器项目与本地项目合并）

$ git push #（将本地项目上传至远端库）

　　（在提交前要git pull --rebase 一下，确保当前的本地的代码为最新。）

（14）建立分支

$ git branch AAA #建立分支AAA

（15）分支切换

$ git checkout AAA #从当前分支切换到AAA分支 (若AAA分支不存在，则自动新建)

（16）将分支与主枝master合并

$ git checkout master #（首先切换回主枝）

$ git merge AAA #（将分支AAA与主枝合并）

注：git merge：默认情况下，Git执行"快进式合并"（fast-farward merge），会直接将master分支指向Develop分支。

　　使用--no-ff参数后，会执行正常合并，在Master分支上生成一个新节点。为了保证版本演进的清晰（保持提交曲线为直线），建议采用这种方法。

（17）当前分支查看

$ git branch #默认有master（也称为主枝）

$ git branch -r #查看远端库分支

$ git branch –a #查看当前所有分支（包括本地分支和远端库分支）

（18）删除分支

$ git branch –d AAA #删除分支AAA

（19）切下远端库A分支到本地库A分支

$ git checkout -b A origin/A （若本地A分支不存在，则自动新建）

维护人：张秋平

更新日志：2019-7-5