**第一部分：git常用命令**

<1>下载服务器代码到本地： **git clone 服务器代码仓库地址**

<2>查看本地分支：**git branch**

<3>查看本地＋远程分支：**git branch -a**

<4>分支跳转：**git checkout 分支名**

<5>将文件加入git管理：**git add 文件名**

<6>查看提交状态：**git status**

<7>提交代码：**git commit 文件名 －m “说明”**

<8>在父分支上下拉服务器（正本）代码：**git pull**

<9>推送代码到服务器（正本）：**git push**（注意，这个命令执行前提是当前所在分支的分支名等于服务器分支名）

<10>查看提交历史：**git log －10** （10是想查看的历史条数，可任意设置）

<11>版本回退：**git reset －－hard** 提交历史哈希串

<12>创建新分支：

［方式1］**git branch** 新分支名（只创建，不跳转进去）

［方式2］**git checkout －b** 新分支名 （创建，且直接跳转进去）

—————————————————————————————————————————————————————————————————————————————————————————————————————————————

**第二部分：mac系统上安装git**

因为Xcode自带git中的rebase命令实际操作效果等同于merge命令，而无冲突提交需要的最重要的命令恰好是rebase，所以在mac系统上需要下载完整的git

步骤1：安装包管理工具HomeBrew

     ruby -e "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/master/install>)”

     这一行代码可能因为复制格式问题在终端上直接执行不成功，去百度搜索homebrew，第一个就是HomeBrew的官网，其中有这条命令，直接拷贝下来执行

步骤2：通过brew命令安装git

    brew install git

—————————————————————————————————————————————————————————————————————————————————————————————————————————————

**第三部分：git开发的四个分支介绍**

【1】服务器代码（远端）正本分支，\*一级提交分支

         真正的服务器代码仓库，用于正式提交代码，一般不可逆

【2】服务器代码（本地）副本分支，\*分布式存储、远程跟踪、协助提交

         服务器代码仓库在本地的备份（副本），跟踪服务器最新代码，一套代码在服务器和每个开发者的本地都有多份拷贝，这就实现了git的分布式存储，如果服务器坏了，任何一个开发者本地的副本都可以安全的保存完整代码。

【3】父分支，\*二级提交分支

         父分支是本地的提交分支，开发者在本地做的开发先提交到父分支上，保证本地父分支无冲突（提交历史为一条直线），然后再把父分支提交到服务器正本分支上

【4】子分支，\*开发分支，解决冲突

         子分支用来做开发或者bug修改，子分支和本地副本分支配合，通过rebase命令，实现无冲突的提交，子分支的修改要通过父分支提交到正本分支上

—————————————————————————————————————————————————————————————————————————————————————————————————————————————

**第四部分：提交流程（五个步骤 + 冲突解决）**

1、（子分支）git fetch

       将远端正本分支的最新代码同步到本地副本分支上来

2、（子分支）git rebase origin/父分支

       将子分支和本地副本分支无冲突合并

3、（子分支）git checkout 父分支

       切换到父分支

4、（父分支）git merge 子分支

        将父分支和子分支无冲突合并，父分支上获得子分支和远端正本分支的最新修改

5、（父分支）git push

        将父分支上子分支的最新修改提交给远端正本分支

  冲突解决：

正常情况下，会在以上步骤中的2、5两步发生冲突：

（子分支）git rebase origin/父分支冲突：

        协商解决冲突，解决完冲突之后执行：

        git add .  把当前解决完冲突的文件添加至提交缓存

        git rebase --continue  继续执行未执行完毕的rebase命令

（父分支）git push冲突：

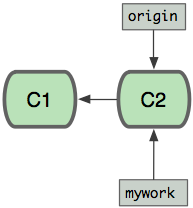
        返回子分支，重新执行一遍1-5五个步骤，并在步骤2解决冲突

—————————————————————————————————————————————————————————————————————————————————————————————————————————————

**第五部分：git rebase 命令的原理解释：（参考[http://gitbook.liuhui998.com/4\_2.html](http://gitbook.liuhui998.com/4_2.html" \t "_blank)）**

假设你现在基于远程分支"origin"，创建一个叫"mywork"的分支。

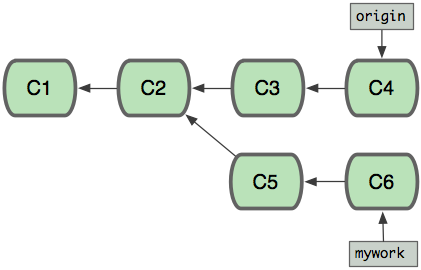
$ git checkout -b mywork origin



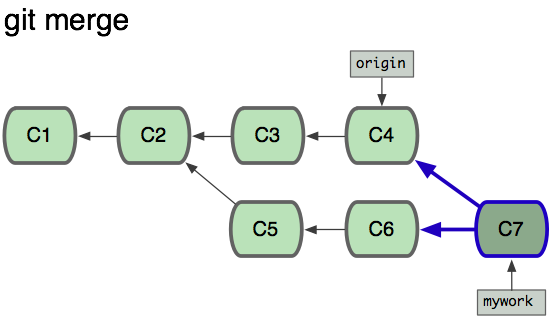
现在我们在这个分支做一些修改，然后生成两个提交(commit).

1. $ vi file.txt
2. $ git commit
3. $ vi otherfile.txt
4. $ git commit
5. ...

但是与此同时，有些人也在"origin"分支上做了一些修改并且做了提交了. 这就意味着"origin"和"mywork"这两个分支各自"前进"了，它们之间"分叉"了。



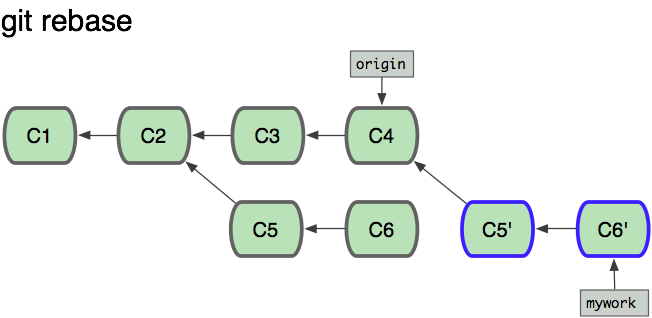
在这里，你可以用"pull"命令把"origin"分支上的修改拉下来并且和你的修改合并； 结果看起来就像一个新的"合并的提交"(merge commit):



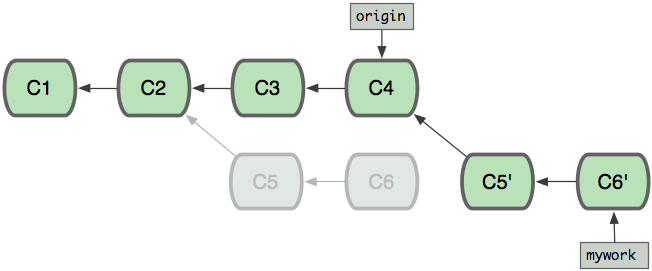
但是，如果你想让"mywork"分支历史看起来像没有经过任何合并一样，你也许可以用 [git rebase](http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html):

1. $ git checkout mywork
2. $ git rebase origin

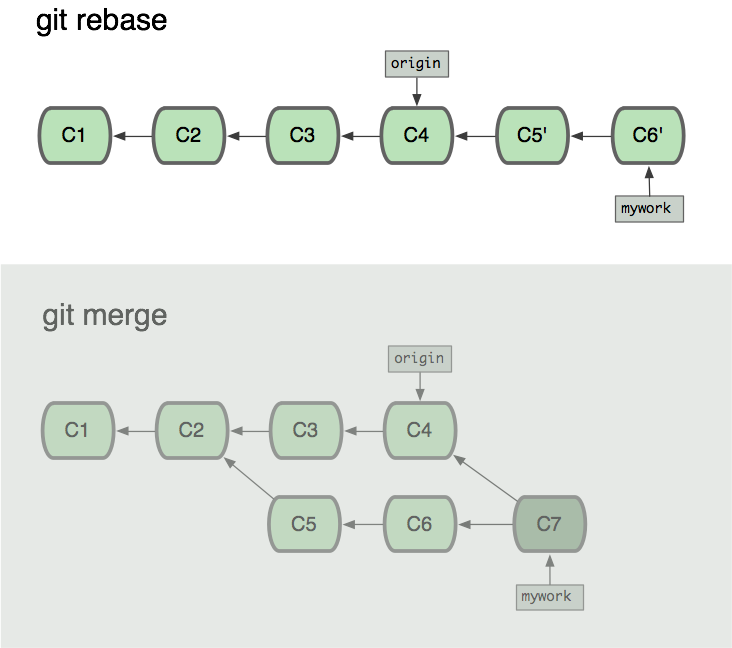
这些命令会把你的"mywork"分支里的每个提交(commit)取消掉，并且把它们临时 保存为补丁(patch)(这些补丁放到".git/rebase"目录中),然后把"mywork"分支更新 到最新的"origin"分支，最后把保存的这些补丁应用到"mywork"分支上。



当'mywork'分支更新之后，它会指向这些新创建的提交(commit),而那些老的提交会被丢弃。 如果运行垃圾收集命令(pruning garbage collection), 这些被丢弃的提交就会删除. （请查看 [git gc](http://www.kernel.org/pub/software/scm/git/docs/git-gc.html))



现在我们可以看一下用合并(merge)和用rebase所产生的历史的区别：



在rebase的过程中，也许会出现冲突(conflict). 在这种情况，Git会停止rebase并会让你去解决 冲突；在解决完冲突后，用"git-add"命令去更新这些内容的索引(index), 然后，你无需执行 git-commit,只要执行:

$ git rebase --continue

这样git会继续应用(apply)余下的补丁。

在任何时候，你可以用--abort参数来终止rebase的行动，并且"mywork" 分支会回到rebase开始前的状态。

$ git rebase *--abort*

—————————————————————————————————————————————————————————————————————————————————————————————————————————————

第六部分：尝试提交

现在有一个给大家尝试提交的远端仓库：[https://github.com/zhch1991/TestGitServer.git](https://github.com/zhch1991/TestGitServer.git" \t "_blank)    这个远端仓库即为：服务器代码（远端）正本分支

本地测试步骤：

1、创建一个新文件夹，进入新文件夹中，执行git初始化

mkdir test

cd test

git init

2、将远端仓库下载到本地

git clone <https://github.com/zhch1991/TestGitServer.git>

此时，执行ls查看，发现当前文件夹中多了：TestGitServer，这个TestGitServer就是从远端仓库上下载的代码

3、进入到文件夹中进行代码和分支查看

cd TestGitServer

git branch -a  此时应该能在分支列表中查看到分支 ：remotes/origin/father，这个分支即为：服务器代码（本地）副本分支，\*分布式存储、远程跟踪、协助提交

4、根据 服务器代码（本地）副本分支 创建 父分支

git checkout -b father origin/father

git branch -a  此时应该能在分支列表中查看到分支 ：father，这个分支即为父分支

5、根据父分支创建子分支

git checkout -b son

 git branch -a  此时应该能在分支列表中查看到分支 ：son，这个分支即为子分支

准备工作完成，在子分支上进行修改，然后按照第四部分提交流程提交即可

—————————————————————————————————————————————————————————————————————————————————————————————————————————————

 第七部分：可能出现的问题

mac系统终端上执行git push或 git pull时，可能出现如下问题：

See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

    git branch --set-upstream-to=origin/<branch> father

原因是mac上不能根据父分支的分支名直接关联到远端同名分支上，遇到这个问题，执行：

 git branch --set-upstream-to=origin/father father

之后就可以正常push和pull了