# Git的基本使用

### 撤销修改

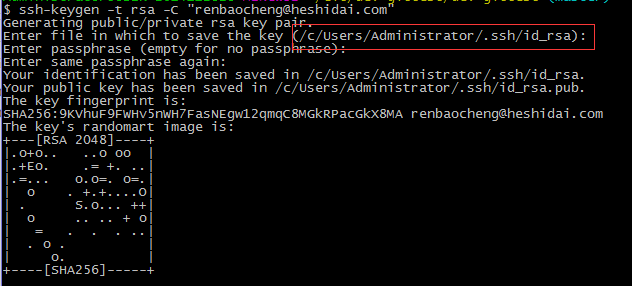
分3种情况

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file **或者** 手动修改。  
场景2：当你不但改乱了工作区某个文件的内容，并且还添加到了暂存区（即已经git add了）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。  
场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，可以用版本回退（参考版本回退一节），**不过前提是没有推送到远程库**。

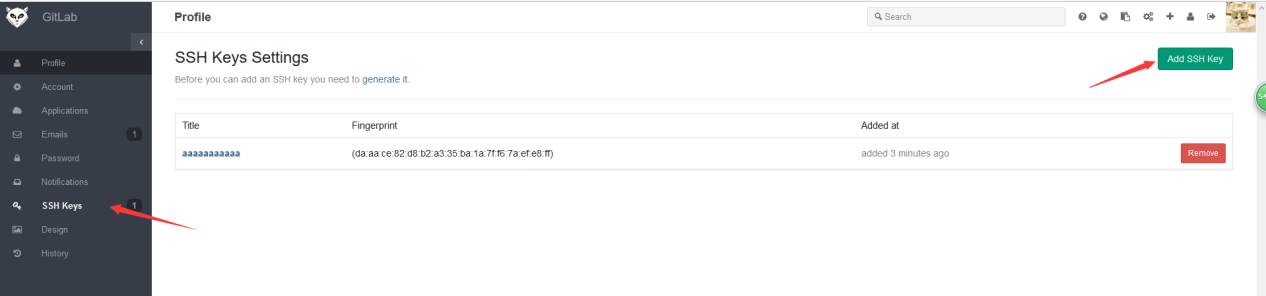
## 配制ssh

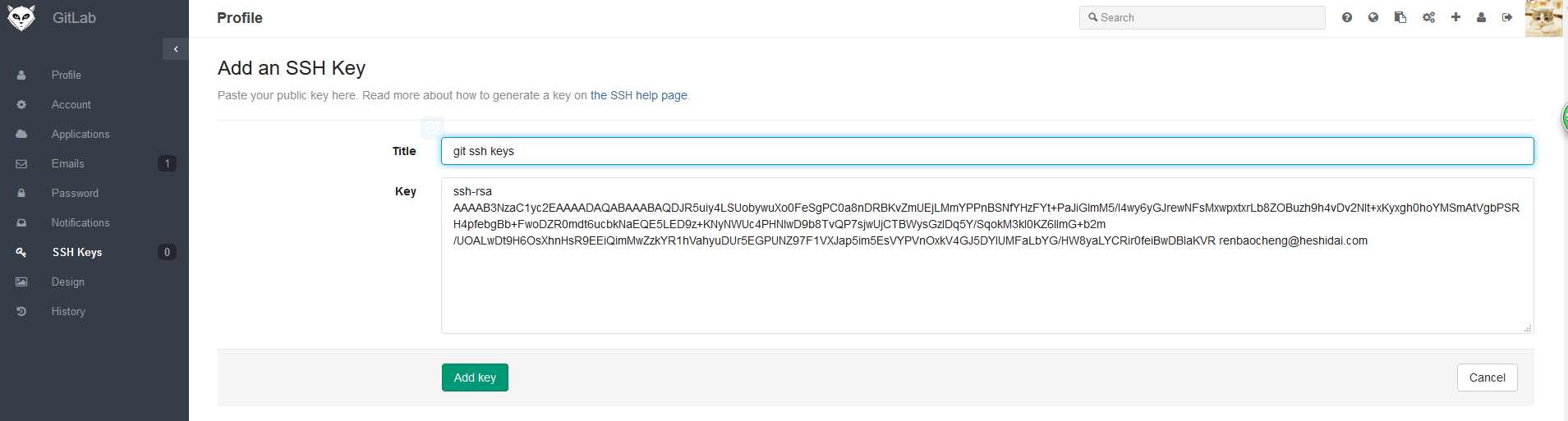
执行命令：ssh-keygen -t rsa -C "renbaocheng@heshidai.com"

可以不用设置密码：直接三次回车



找到文件id\_rsa，复制里面的内容添加到远程仓库的SSH key中





可以测试设置ssh是否成功：ssh [git@192.168.1.239](mailto:git@192.168.1.239)

## 更新代码

1. 首次下载源码：

git clone <http://192.168.1.239:8936/duxu/dev-gittest.git>

1. 配置ssh后首次下载源码：

git clone [git@192.168.1.239:duxu/dev-gittest.git](mailto:git@192.168.1.239:duxu/dev-gittest.git)

1. 更新远程仓库代码：git pull origin <branchname>

## 提交代码

1. 查看修改内容：git diff <filename>
2. 添加代码到暂存区： git add <filename>
3. 提交代码到版本库：git commit -m  "注释"
4. 添加并提交到版本库：git commit -a -m '注释'
5. 本地仓库代码提交到远程仓库：git push origin <branchname>
6. 查看提交代码的结果：git status
7. 查看历史记录：git log （参数：--pretty=oneline）
8. 查看历史命令记录：git reflog

## 还原文件

1. 修改了工作区或删除了某个文件，想还原时：

git checkout -- <fileName>

1. 修改的文件添加加到暂存区时（即执行了add操作），想放弃修改时：

需要两步：第一步：git reset HEAD <fileName>

第二步：git checkout -- <fileName>

1. 修改的文件已经提交了修改（即执行了commit操作），想要撤销本次提交，前提是还没有推送到远程库：

git reset --hard HEAD^

或 git reset –-hard HEAD~1

1. 还原文件到某个版本:

git reset --hard <版本号> （git log 或 git reflog获取版本号）

## 分支管理

1. 创建基于master的本地分支：git branch <branchname> master
2. 修改本地分支名称：git branch -m <oldname> <newname>
3. 删除本地分支：git branch -d <branchname>
4. 强制删除分支：git branch -D <branchname>
5. 查看分支：git branch （-a参数查看到远程分支，远程分支红色）
6. 切换分支：git checkout <branchname>
7. 创建+切换分支：git checkout -b <branchname>
8. 合并某分支到当前分支：

先切换到主分支：git checkout master

把分支<branchname>合并到主分支：

git merge <branchname> 或 git merge --no-ff <branchname>

(默认情况下，Git执行快进式合并,会直接将Master分支指向Develop分支

加上--no-ff参数后，会执行正常合并，在Master分支上生成一个新节点,

merge的时候如果遇到冲突，就手动解决，然后重新add，commit即可)

1. 查看合并的分支图表：

git log --graph --pretty=oneline --abbrev-commit

10、新建一个基于远程的dev分支：git checkout -b dev origin/dev

## 远程分支管理

1. 创建远程分支：git push origin <branchname>
2. 删除远程分支：git push origin --delete <branchname>
3. 查看远程分支(红色的)：git branch -a 或 git branch -r
4. 本地分支和远程分支相一致：git fetch origin --prune

## 解决冲突

1. 查看冲突的文件：git status
2. 打开冲突的文件，修改对应的冲突部分
3. 提交修改后的代码即可

## tag操作

1. 添加标签：git tag <tagname> <branchName>

(<tagname>是标签名称，<branchName>是希望打标签的点分支的末梢)

或 git tag <tagname> <commit版本号>

或 git tag -a <tagname> -m "Version 1.1"

1. 查看标签：git tag
2. 查询标签详细信息：git show <tagname>
3. 提交单个标签到远程仓库：git push origin <tagname>
4. 提交全部未提交的标签到远程仓库：git push origin --tags
5. 删除本地标签：git tag -d <tagname>
6. 删除远程标签：git push origin --delete tag <tagname>

## stash的使用

 Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：  
     1、把当前工作线程存储起来：git stash

     2、查看当前存储起来的文件：git stash list

     3、恢复当前工作现场： git stash pop  （直接会删除储藏的文件）

     4、git stash apply（不会删除储藏的文件，需要手动删除）

5、git stash drop(手动删除命令)

## rebase 的使用

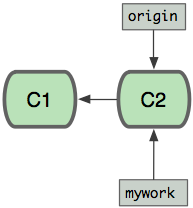
### 1、基本

git rebase用于把一个分支的修改合并到当前分支。

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

假设远程分支"origin"已经有了2个提交，如图

[](http://blog.chinaunix.net/attachment/201209/19/26952464_1348017759qs4k.png)

现在我们在这个分支做一些修改，然后生成两个提交(commit).

$ vi file.txt

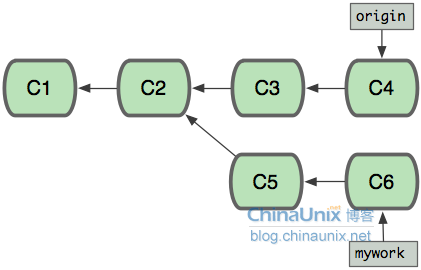
$ git commit

$ vi otherfile.txt

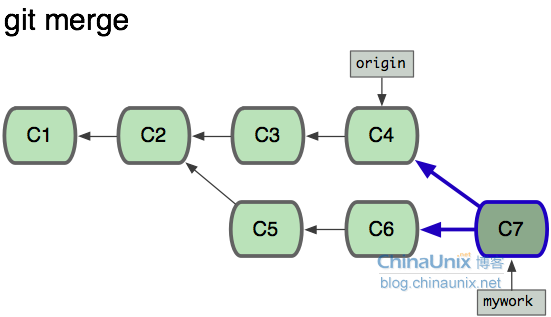
$ git commit

...

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。

[](http://blog.chinaunix.net/attachment/201209/19/26952464_1348017823wiQB.png)

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):

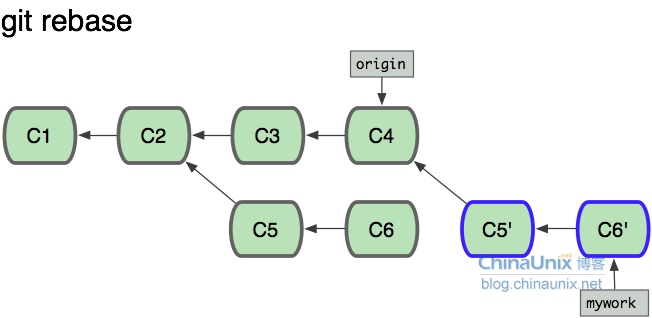
[](http://blog.chinaunix.net/attachment/201209/19/26952464_13480178579bG6.png)

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:

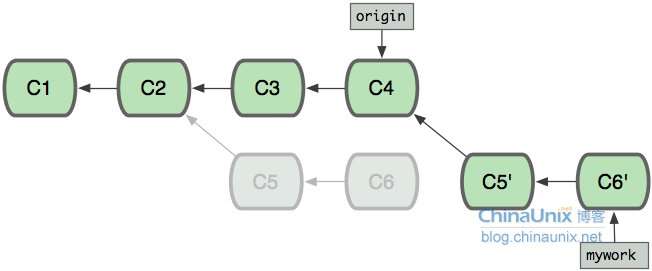
$ git checkout mywork

$ 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 为最新的"origin"分支，最后把保存的这些补丁应用到"mywork"分支上。

[](http://blog.chinaunix.net/attachment/201209/19/26952464_13480178888X41.png)

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://blog.chinaunix.net/attachment/201209/19/26952464_1348017931Waw1.png)

### 2、解决冲突

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

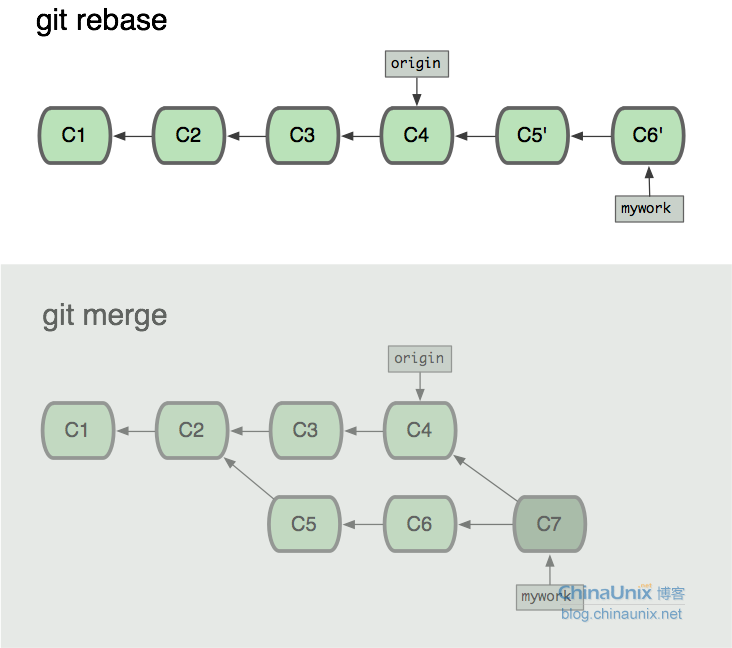
这样git会继续应用(apply)余下的补丁。

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

三、git rebase和git merge的区别

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：

[](http://blog.chinaunix.net/attachment/201209/19/26952464_1348018047J6ss.png)

当我们使用Git log来参看commit时，其commit的顺序也有所不同。

假设C3提交于9:00AM,C5提交于10:00AM,C4提交于11:00AM，C6提交于12:00AM,

对于使用git merge来合并所看到的commit的顺序（从新到旧）是：C7 ,C6,C4,C5,C3,C2,C1

对于使用git rebase来合并所看到的commit的顺序（从新到旧）是：C7 ,C6‘,C5',C4,C3,C2,C1

 因为C6'提交只是C6提交的克隆，C5'提交只是C5提交的克隆，

从用户的角度看使用git rebase来合并后所看到的commit的顺序（从新到旧）是：C7 ,C6,C5,C4,C3,C2,C1

## reset 的使用

1：git reset –mixed：此为默认方式，不带任何参数的git reset，即时这种方式，它回退到某个版本，只保留源码，回退commit和index信息

2：git reset –soft：回退到某个版本，只回退了commit的信息，不会恢复到index file一级。如果还要提交，直接commit即可

3：git reset –hard：彻底回退到某个版本，本地的源码也会变为上一个版本的内容

git reset HEAD^ 回退所有内容到上一个版本

git reset HEAD^ <fileName> 回退文件名<fileName>这个文件的版本到上一个版本

git reset –soft HEAD~3 向前回退到第3个版本

git reset –hard origin/master 将本地的状态回退到和远程的一样

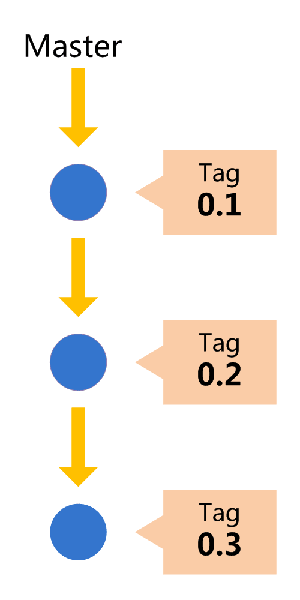
git reset 057d 回退到某个版本

git revert HEAD 回退到上一次提交的状态，按照某一次的commit完全反向的进行一次commit

## Git分支管理策略

### 1、主分支Master

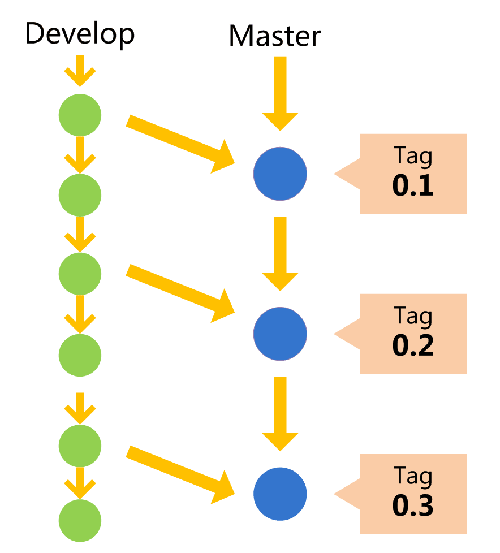
首先，代码库应该有一个、且仅有一个主分支。所有提供给用户使用的正式版本，都在这个主分支上发布。



Git主分支的名字，默认叫做Master。它是自动建立的，版本库初始化以后，默认就是在主分支在进行开发。

### 2、开发分支Develop

主分支只用来分布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做Develop。



这个分支可以用来生成代码的最新隔夜版本（nightly）。如果想正式对外发布，就在Master分支上，对Develop分支进行"合并"（merge）。

Git创建Develop分支的命令：

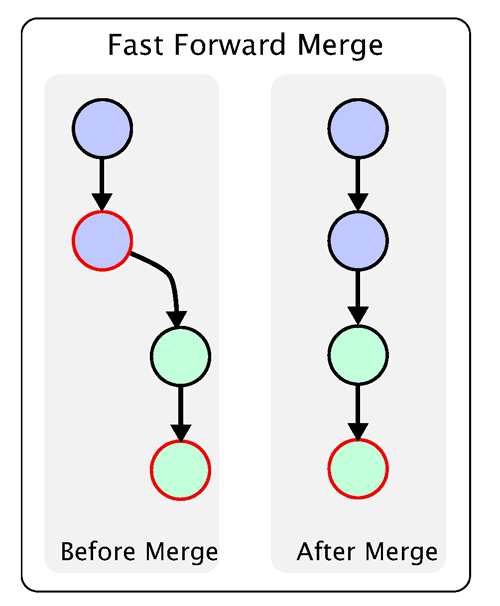
　　git checkout -b develop master

将Develop分支发布到Master分支的命令：

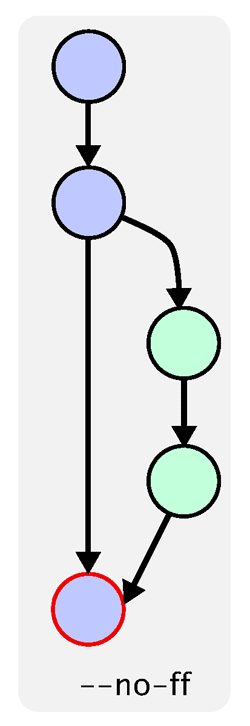
　　# 切换到Master分支  
　　git checkout master

　　# 对Develop分支进行合并  
　　git merge --no-ff develop

这里稍微解释一下，上一条命令的--no-ff参数是什么意思。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，我们希望采用这种做法。关于合并的更多解释，请参考Benjamin Sandofsky的[《Understanding the Git Workflow》](http://sandofsky.com/blog/git-workflow.html" \t "http://www.ruanyifeng.com/blog/2012/07/_blank)。



### 3、临时性分支

前面讲到版本库的两条主要分支：Master和Develop。前者用于正式发布，后者用于日常开发。其实，常设分支只需要这两条就够了，不需要其他了。

但是，除了常设分支以外，还有一些临时性分支，用于应对一些特定目的的版本开发。临时性分支主要有三种：

　　\* 功能（feature）分支

　　\* 预发布（release）分支

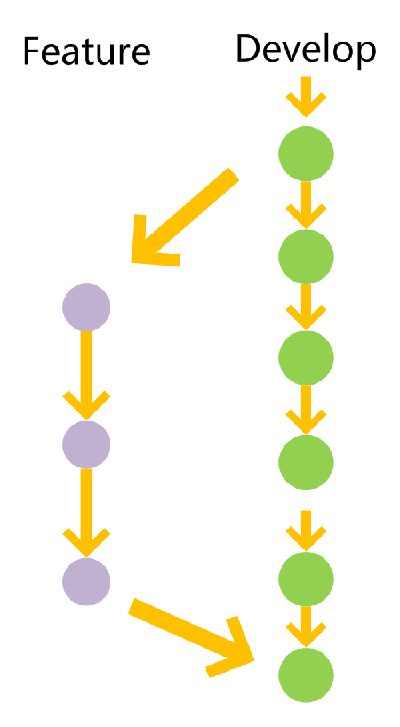
　　\* 修补bug（fixbug）分支

这三种分支都属于临时性需要，使用完以后，应该删除，使得代码库的常设分支始终只有Master和Develop。

### 4、功能分支Feature

接下来，一个个来看这三种"临时性分支"。

第一种是功能分支，它是为了开发某种特定功能，从Develop分支上面分出来的。开发完成后，要再并入Develop。



功能分支的名字，可以采用feature-\*的形式命名。

创建一个功能分支：

　　git checkout -b feature-x develop

开发完成后，将功能分支合并到develop分支：

　　git checkout develop

　　git merge --no-ff feature-x

删除feature分支：

　　git branch -d feature-x

### 5、预发布分支Release

第二种是预发布分支，它是指发布正式版本之前（即合并到Master分支之前），我们可能需要有一个预发布的版本进行测试。

预发布分支是从Develop分支上面分出来的，预发布结束以后，必须合并进Develop和Master分支。它的命名，可以采用release-\*的形式。

创建一个预发布分支：

　　git checkout -b release-1.2 develop

确认没有问题后，合并到master分支：

　　git checkout master

　　git merge --no-ff release-1.2

　　# 对合并生成的新节点，做一个标签  
　　git tag -a 1.2

再合并到develop分支：

　　git checkout develop

　　git merge --no-ff release-1.2

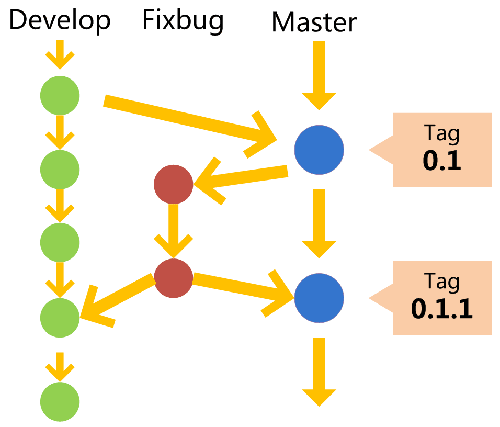
最后，删除预发布分支：

　　git branch -d release-1.2

### 6、修补bug分支Fixbug

最后一种是修补bug分支。软件正式发布以后，难免会出现bug。这时就需要创建一个分支，进行bug修补。

修补bug分支是从Master分支上面分出来的。修补结束以后，再合并进Master和Develop分支。它的命名，可以采用fixbug-\*的形式。



创建一个修补bug分支：

　　git checkout -b fixbug-0.1 master

修补结束后，合并到master分支：

　　git checkout master

　　git merge --no-ff fixbug-0.1

　　git tag -a 0.1.1

再合并到develop分支：

　　git checkout develop

　　git merge --no-ff fixbug-0.1

最后，删除"修补bug分支"：

　　git branch -d fixbug-0.1

## Git上的项目管理

### 1、新建repository

本地目录下，在命令行里新建一个代码仓库（repository）  
　　里面只有一个README.md  
　　命令如下：  
　　**touch README.md**  
　　**git init**  
　　初始化repository

　　**git add README.md**  
　　将README.md加入到缓存区

　　（可以用**git add --a**将所有改动提交到缓存（注意是两个杠））

**git commit -m "first commit"**  
　　提交改变，并且附上提交信息"first commit"

### 2、Push

**git remote add origin https://github.com/XXX(username)/YYYY(projectname).git**

　　加上一个remote的地址，名叫origin，地址是github上的地址（Create a new repo就会有）  
　　因为Git是分布式的，所以可以有多个remote.

　　**git push -u origin master**  
　　将本地内容push到github上的那个地址上去。

**参数-u**  
　　用了参数-u之后，以后就可以直接用不带参数的git pull从之前push到的分支来pull。

　　此时如果origin的master分支上有一些本地没有的提交,push会失败.

　　所以解决的办法是, 首先设定本地master的上游分支:

**git branch --set-upstream-to=origin/master**

　　然后pull:  
　　**git pull --rebase**

　　最后再push:

**git push**

### 3、分支

　　新建好的代码库有且仅有一个主分支（**master**），它是自动建立的。  
　　可以新建分支用于开发：  
　　**git branch develop master**  
　　新建一个叫develop的分支，基于master分支

　　切换到这个分支：  
　　**git checkout develop**  
　　现在可以在这个develop分支上做一些改动，并且提交。  
　　**注意**：切换分支的时候可以发现，在Windows中的repository文件夹中的文件内容也会实时相应改变，变成当前分支的内容。

### 4、push方法1：

　　现在如果想直接Push这个develop分支上的内容到github

**git push -u origin**

　　如果是新建分支第一次push，会提示：  
　　fatal: The current branch develop has no upstream branch.  
　　To push the current branch and set the remote as upstream, use  
　　git push --set-upstream origin develop  
　　输入这行命令，然后输入用户名和密码，就push成功了。

以后的push就只需要输入**git push origin**

### 5、push方法2：

　　比如新建了一个叫dev的分支，而github网站上还没有，可以直接：

**git push -u origin dev**

这样一个新分支就创建好了。

### 6、push方法3：

　　提交到github的分支有多个，提交时可以用这样的格式：

**git push -u origin local:remote**

　　比如：**git push -u origin master:master**  
　　表明将本地的master分支（冒号前）push到github的master分支（冒号后）。  
　　**如果左边不写为空，将会删除远程的右边分支。**

### 7、创建分支的另一种方法

　　用命令**git checkout -b develop2 develop**  
　　可以新建一个分支develop2，同时切换到这个分支

### 8、删除分支

**git branch**可以查看所有的分支  
　　**git branch -d develop2** 将develop2分支删除

### 9、Clone

　　使用git clone+github地址的方法，项目默认只有master分支。**git branch**也只有master

　　要看所有的分支：**git branch -a**或者是**git branch -r**

　　这时候要新建一个分支，叫做dev，基于远程的dev分支：**git checkout -b dev origin/dev**

### 10、加Tag

**git tag tagname develop**  
　　git tag中的两个参数，一个是标签名称，另一个是希望打标签的点develop分支的末梢。

### 11、合并分支

**git checkout master**

　　先转到主分支  
　　**git merge --no-ff develop**

　　然后把develop分支merge过来

**参数意义：**  
　　不用参数的默认情况下，是执行快进式合并。  
　　使用参数--no-ff，会执行正常合并，在master分支上生成一个新节点。  
　　merge的时候如果遇到冲突，就手动解决，然后重新add，commit即可。