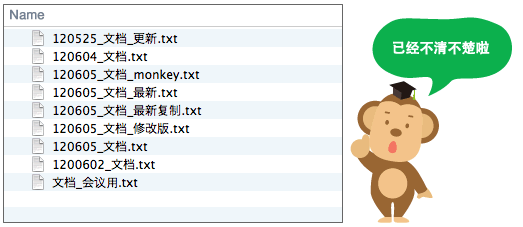
# 入门

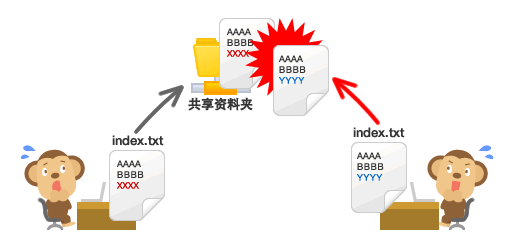
## Git基础

保存文件的多个版本，或者保存项目的多个版本的做法是，改个名字，拷贝。

一旦版本过多非常麻烦

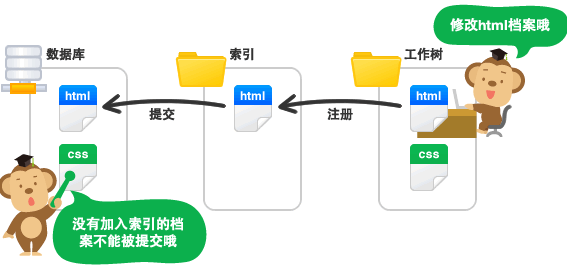


如果两个人共享操作一份代码，则你改的我又改，改来改去乱七八糟。而且我改的东西往往把你的覆盖了，想要还原也非常麻烦



Git下，操作的目录叫做工作树

数据库（版本库）和工作树之间有索引。索引是为了向数据库提交作准备的区域



### 安装

安装命令行工具

### 初期设定

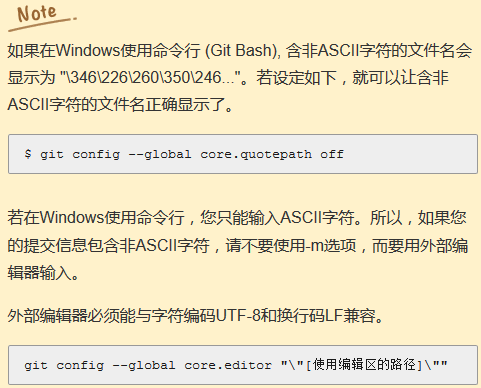
配置你的个人信息

$ git config --global user.name "<用户名>"

$ git config --global user.email "<电子邮件>"

让git以彩色显示

$ git config --global color.ui auto



### 新建数据库

首先在任意一个地方创建tutorial目录。然后使用init命令把该tutorial目录移动到本地 Git数据库。

$git init

$ mkdir tutorial

$ cd tutorial

$ git init

### 提交文件

查看状态

git status

将文件加入到索引，要使用add命令。在<file>指定加入索引的文件。

用空格分割可以指定多个文件。

git add <file> ..

指定参数 . ，可以把所有文件加入到索引

git add .

提交文件

git commit -m “”

查看历史记录

git log

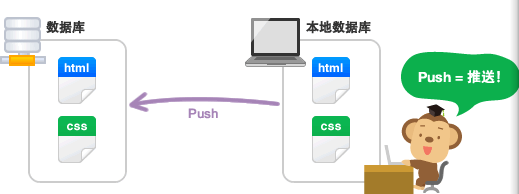
## 共享数据库

以上是本地的数据库操作，如果个人使用当然最好。但是又不安全因素。如果你是一个小说家写了几百万的小说，并且由很多版本。突然有可能家里来贼了，地震了，等等。总之你电脑不见了你的文件也就不见了。因此本地数据库不太安全。最好是放在远程。这样本地一份远程一份非常安全。

### push到远程数据库

为了将本地数据库的修改记录共享到远程数据库，必须上传本地数据库中存储的修改记录。

需要在Git执行推送(Push)操作。执行Push之后，本地的修改记录会被上传到远程数据库。**所以远程数据库的修改记录就会和本地数据库的修改记录保持同步。**



### 克隆远程数据库

进行克隆（Clone）操作就可以复制远程数据库。

共享数据库

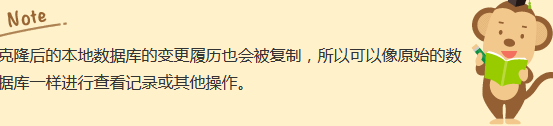
# 克隆远程数据库

如果远程数据库中有他人的修改记录，那么把它完整地复制下来您就可以接着进行工作了。

## 克隆

进行克隆（Clone）操作就可以复制远程数据库。

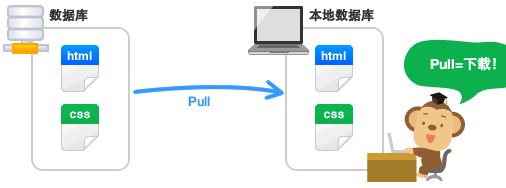
执行克隆后，远程数据库的全部内容都会被下载。之后您在另一台机器的本地数据库上进行操作。



### 从远程数据库pull

若是共享的远程数据库由多人同时作业，那么作业完毕后所有人都要把修改推送到远程数据库。然后，自己的本地数据库也需要更新其他人推送的变更内容。

进行拉取(Pull) 操作就可以把远程数据库的内容更新到本地数据库。进行拉取(Pull) 操作，就是从远程数据库下载最近的变更日志，并覆盖自己本地数据库的相关内容。



### 在贝格乐建立远程数据库

首先在贝格乐创建远程数据库。

Tips

在贝格乐，您的帐户需要具有管理员的权限才能创建Git数据库。如果您的帐户没有管理员的权限，可以委托有此权限的人来创建数据库。友情提示：您可以使用免费方案的贝格乐空间哟。

[从这里获取空间](http://backlogtool.com/cn/pricing/" \t "https://backlog.com/git-tutorial/cn/intro/_blank)

请登录到贝格乐，然后，从创建数据库的项目菜单中点击“Git”。如果您的空间还没有任何项目，请先创建一个项目。



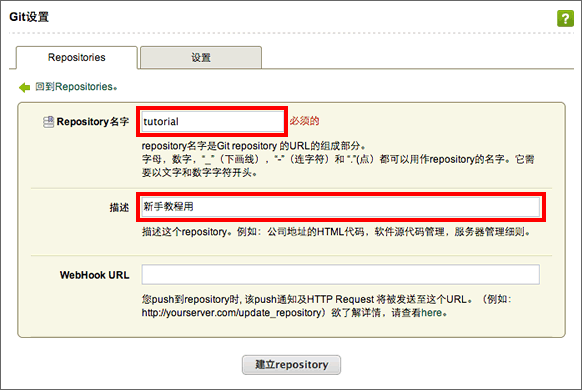
如果页面没有显示“Git”标签，请先激活Git功能。“项目设置” > 在 “Git设置”里激活Git功能。



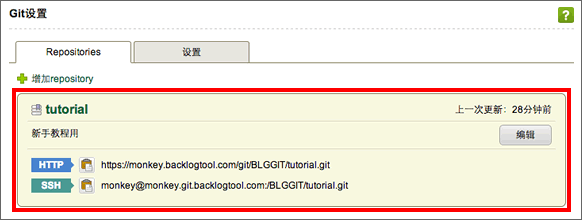
激活Git功能后，在同一个Git设置页面中点击“增加 repository”。



请输入数据库的名字及其描述说明，然后点击“建立repository”按钮。这里我们在名字栏输入了“tutorial”，在描述说明栏输入了“新手教程用”。



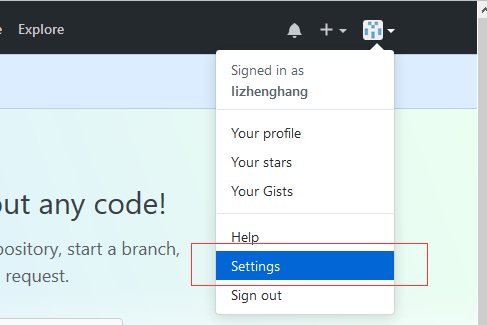
请看下图，我们成功地添加了新的数据库。

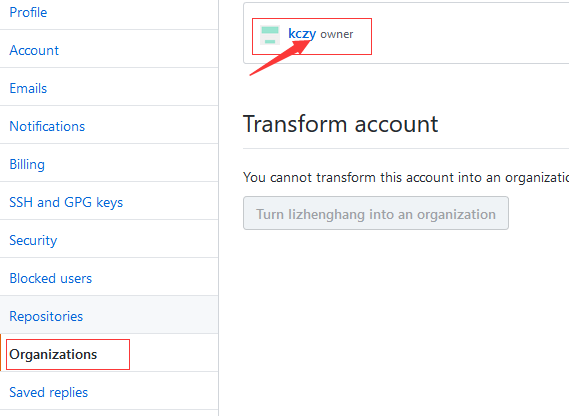


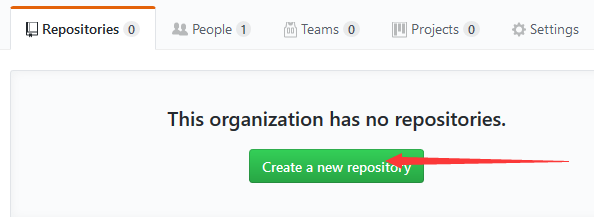
接下来就把修改推送到这个远程数据库吧！

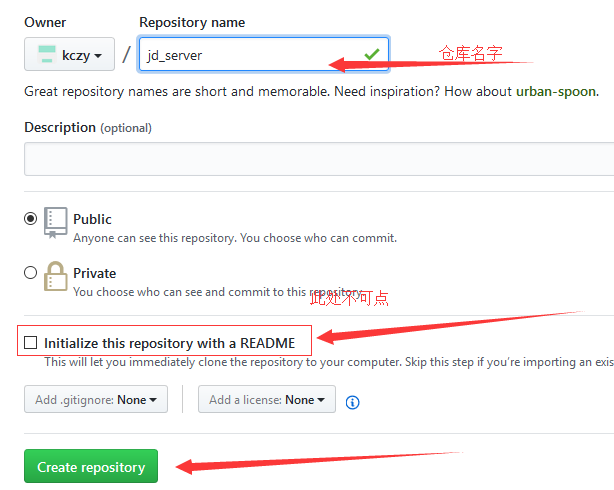
### push

因为是公司的项目，贝格乐收费，于是我托管在github上









初始化仓库后你就不能推送了，必须克隆后才能推送。我们要的效果是将我们已有的代码推送上来。也就是本地有了仓库了。所以不能初始化远程仓库，否则推送不到

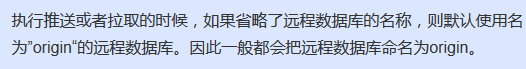


向远程推送本地修改,请使用remote指令添加远程数据库

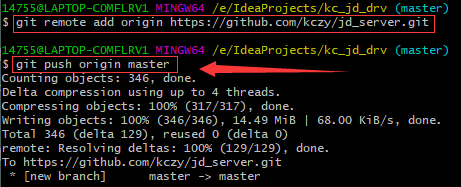
$ git remote add <name> <url>

name是给远程数据库取得别名,这样下次推送的时候就不需要输入长串的远程数据库地址了。在这里我们的远程数据库命名别名为origin

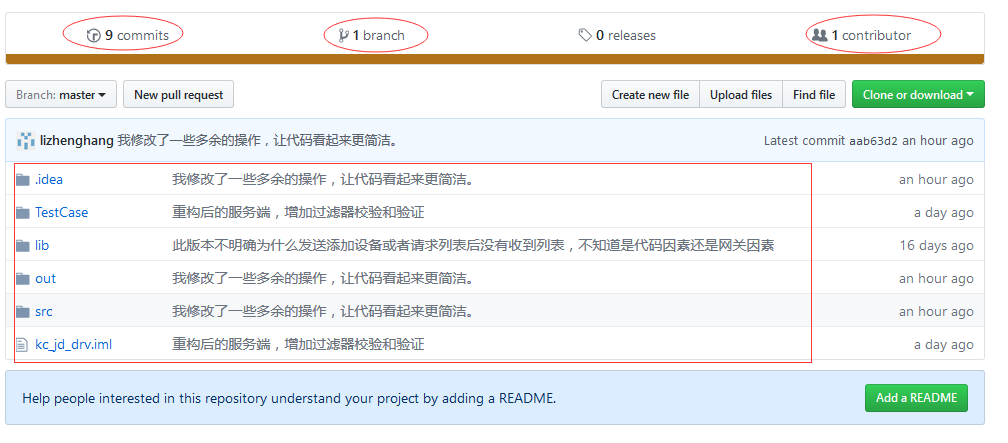




然后将本地代码推送到远程



然后再看看你的github



恭喜你，你已经将本地代码和本地版本库以及一对你的提交历史全部推送到网站托管了

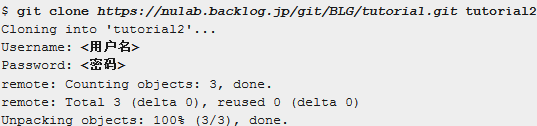
删除远程仓库命令：

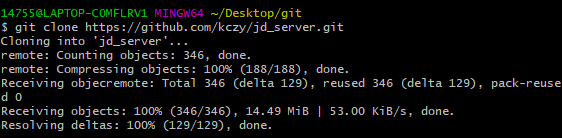
git remote rm 远程仓库名称

### 克隆

假设现在B程序员要参与开发，他应该从远程克隆代码

$ git clone <repository> <directory>





### 从克隆的数据库进行push

上面已经将本地版本库推送到了远程，

后面如果我新增加修改了版本还需要推送就可以省略数据库名和分支名称

add 把变更录入到索引中

$ git add sample.txt

$ git commit -m "添加add的说明"

$ git push

Username: <用户名>

Password: <密码>

### 从远程数据库pull

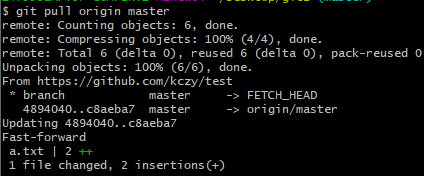
当B程序员跟新修改后需要在本地提交，然后推送到远程

A程序员要获得B的修改就需要pull 拉拔拖

B:



A:



$ git pull <repository> <refspec>...

$ git pull origin master

## 整合修改记录

以上是理想情况下一个人push后你再pull。

如果你pull之后修改，准备push的时候如果有人也push了，那么你的push就会被拒绝

实验，远程库上的文件被两个终端克隆 A端push修改后B端pull没问题，但是A端push后B端push被拒绝！

因为别人push后你也push，如果你们是修改的同一个文件就很有可能把别人的修改给覆盖。这是非常不安全的。因此git禁止。

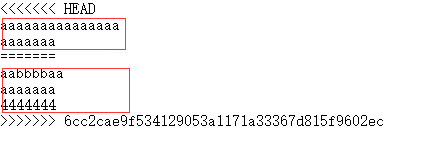
所以你要push就要先合并别人的代码

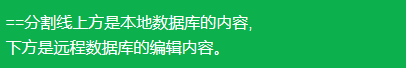
### 合并修改记录

A修改了文件的第一行，push

B 也修改了文件的第一行，因为A推送了所以B要推送首先得pull。然后文件合并就冲突了

因为两个人都修改了第一行，系统就将两人的修改都保留给你处理





修正后再push就没问题了

### 解决冲突

内容同上

# 高级

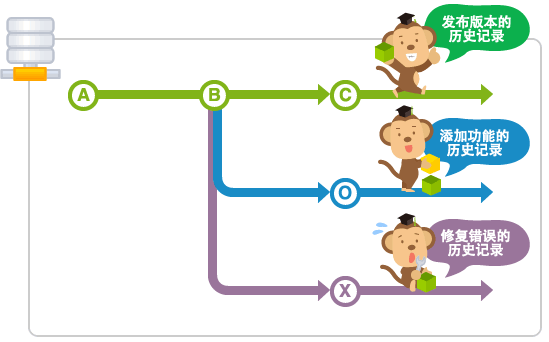
## 分支

开发中，以网安公司的云监管平台为例。同样的系统可能分为四川和新疆两个业务。因此需要两个团队来维护。也就是说一个系统存在两个版本。网安使用SVN分支出来新疆系统。结果出现很多问题。最后不得不合并在主分支上操作。

SVN不推荐分支，对于SVN来说是很麻烦的。

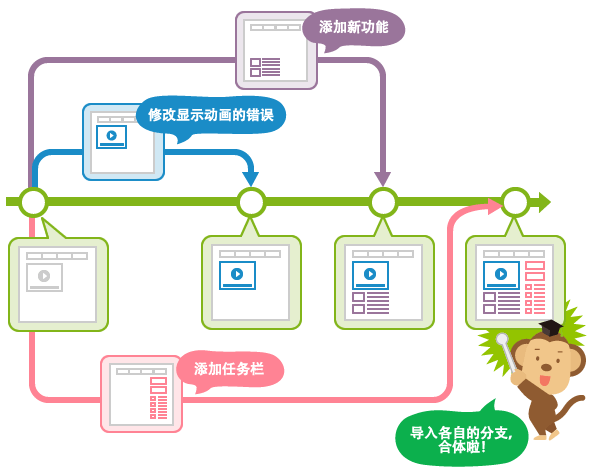
但是git就是轻而易举了。

### 什么是分支



讲一个版本分叉来开发。既可以分叉也可以合并。

工作中如果很多人都要操作一个版本库，提交更新拉取你觉得很麻烦你也可以给自己创建一个分支。自己在分支上操作，互不干扰。等你操作好了再去合并即可



### master分支

当数据库进行最初的提交时，GIT就会创建一个名为master的主分支。

### 分支的运用

分支主要有如下两种类型：

Merge分支

此分支是为了随时可以发布release而创建的分支，

一般大家都把master分支当Merge分支使用

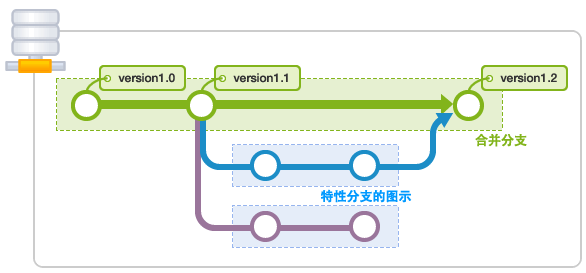
merge分支常作为topic分支的源分支使用

Topic分支

此分支通常是为了开发新功能或修复bug等任务而创建的分支

若要进行多个任务的开发请创建多个topic分支

topic分支是从最稳定的merge分支创建的。完成作业后请将topic合并到merger



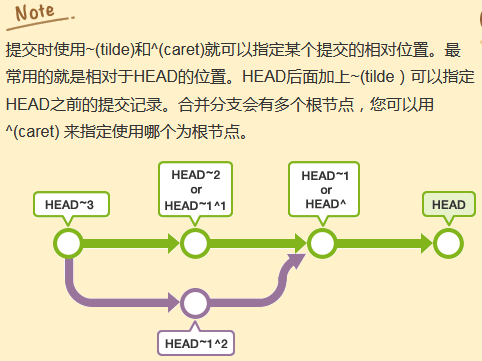
### 分支的切换

若要切换作业的分支，就要进行checkout操作。

HEAD

HEAD指针指向的是现在使用的分支的最后一次更新。

通常默认指向master分支的最后一次更新。通过移动HEAD指针的指向来变更操作的分支

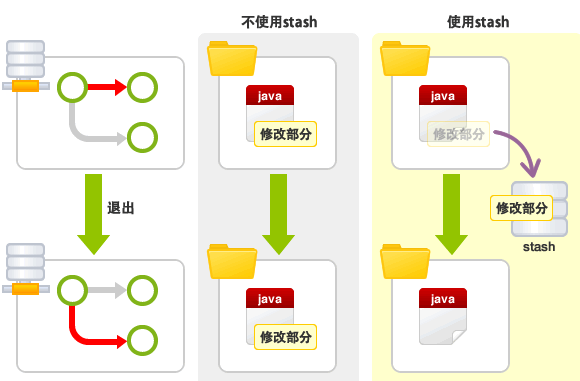


stash

还未提交的修改内容及新添加的文件。如果留在索引区或者工作树中的时候切换到其他分支，那么你的修改会移动到目标分支。

如果你checkout的目标分支中有相同的文件也有修改，那么checkout就会失败。这时候要么先提交修改内容，要么就使用stash保存修改内容后再checkout

stash就是一个保存文件修改内容的区域。可以暂存工作树和索引中没有提交的修改内容。你可以事后再取出暂存的修改放回到原先的分支或者其他分支中

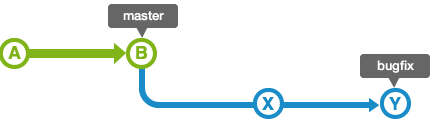


### 分支的合并

习惯：人们通常把master分支当做merge分支使用，人们怕麻烦。

合并分支有两种办法：

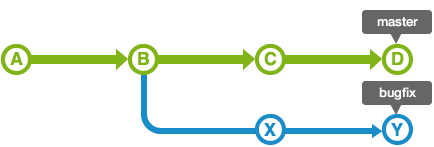
**merge**



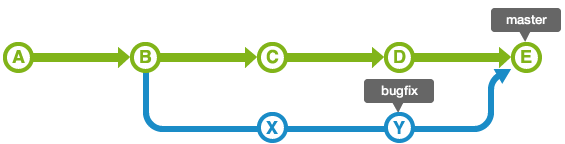
如图所示，bugfix是从master分支中分叉出来的。合并到主分支的时候如果master的状态没有更改过。那么这个合并是非常简单的。bugfix分支的历史记录包含master分支的所有历史记录。所以通过吧master分支的文职移动到bugfix分支的最新分支上，Git就会合并。这样的合并叫做fast-forward（快进）

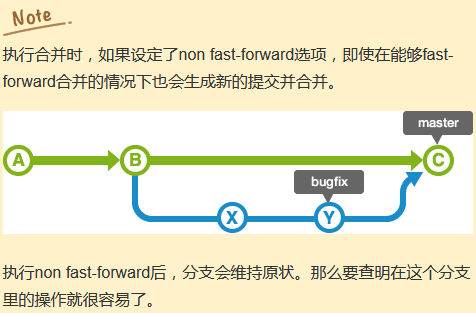


如果master分支在bugfix分支分叉出去后有新的更新

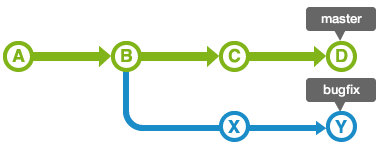


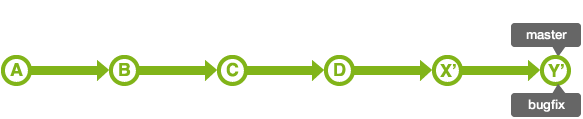
这时候两个分支合并就会生成一个新的提交。master的分支就会移动到该提交上

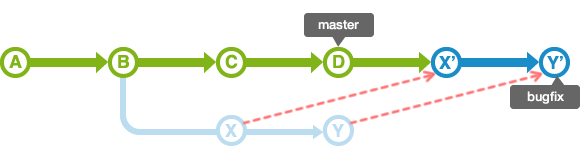


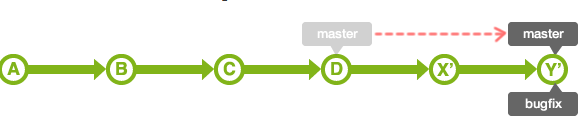


**rebase**









总结：

merge保持修改内容的历史记录，但是历史记录会很复杂

rebase历史记录简单，但是在原有的提交基础上将差异内容反映进去。因此可能导致原本的提交内容无法正常运行

### topic分支和merge分支的运用实例

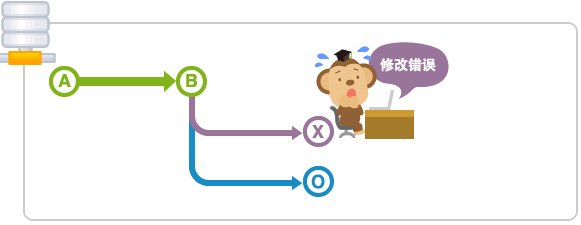
绿色为merge分支

蓝色为topic任务分支

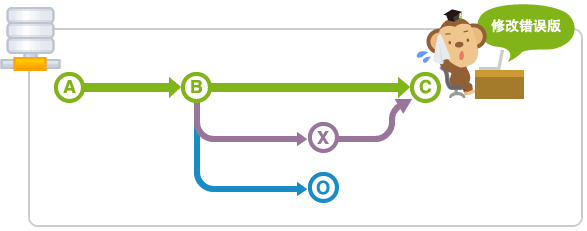


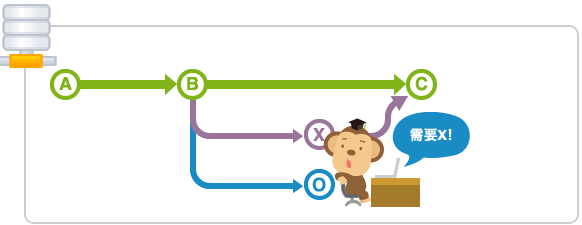
例如，在开发功能的topic分支途中需要修改bug

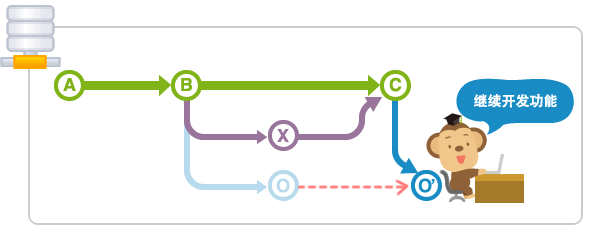
紫色为需要修改的分支



修改错误后就把分支合并到原本的merge/master分支中







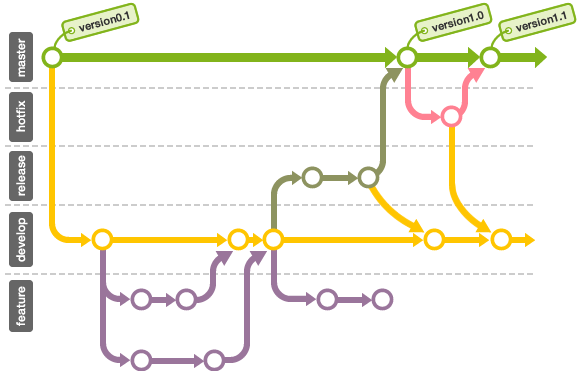
有效的利用分支就可以同时进行不同的作业

作为Git的分支的用例 ，这里介绍 A successful Git branching model

原文:  
[http://nvie.com/posts/a-successful-git-branching-model/](http://nvie.com/posts/a-successful-git-branching-model/" \t "https://backlog.com/git-tutorial/cn/stepup/_blank)

这个用例主要分为

* **主分支**
* **特性分支**
* **release分支**
* **hotFix分支**



主分支

主分支有两种：master分支和develop分支

* **master**  
  master分支只负责管理发布的状态。在提交时使用标签记录发布版本号。
* **develop**  
  develop分支是针对发布的日常开发分支。刚才我们已经讲解过有合并分支的功用。

特性分支

**特性分支就是我们在前面讲解过的topic分支的功用**。

**这个分支是针对新功能的开发，在bug修正的时候从develop分支分叉出来的。**基本上不需要共享特性分支的操作，所以不需要远端控制。完成开发后，把分支合并回develop分支后发布。

release分支

release分支是为release做准备的。通常会在分支名称的最前面加上release-。release前需要在这个分支进行最后的调整，而且为了下一版release开发用develop分支的上游分支。

一般的开发是在develop分支上进行的，到了可以发布的状态时再创建release分支，为release做最后的bug修正。

到了可以release的状态时，把release分支合并到master分支，并且在合并提交里添加release版本号的标签。

要导入在release分支所作的修改，也要合并回develop分支。

hotFix分支

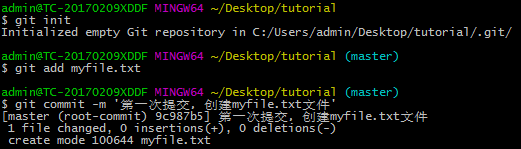
hotFix分支是在发布的产品需要紧急修正时，从master分支创建的分支。通常会在分支名称的最前面加上 hotfix-。

例如，在develop分支上的开发还不完整时，需要紧急修改。这个时候在develop分支创建可以发布的版本要花许多的时间，所以最好选择从master分支直接创建分支进行修改，然后合并分支。

修改时创建的hotFix分支要合并回develop分支。

## 操作分支

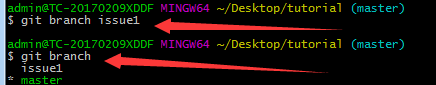
### 事前准备



目前的历史记录是这样的

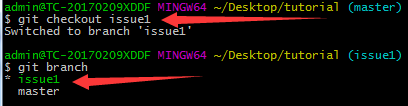


### 建立分支

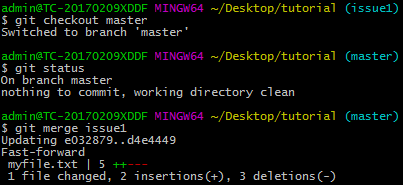


git branch [branch\_name] 如果不加分支名就是显示分支列表，\*指向的分支就是当前操作分支。

### 切换分支

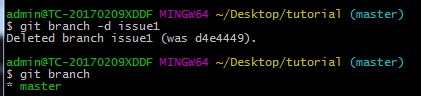


### 合并分支



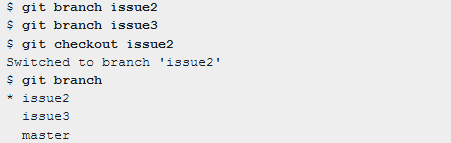
### 删除分支

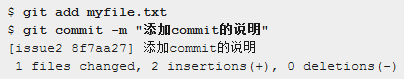
git branch -d <branchname>



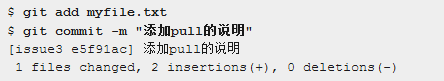
### 并行操作

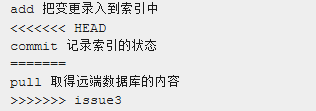
创建两个分支来尝试并行操作









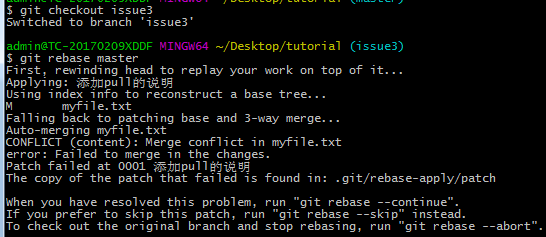


### rebase合并

在合并issue3分支的时候使用rebase分支可以使提交的历史记录显得更简洁

现在暂时取消刚才的合并

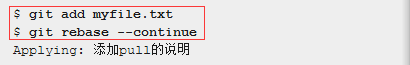
git reset --hard HEAD~

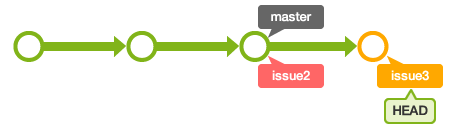


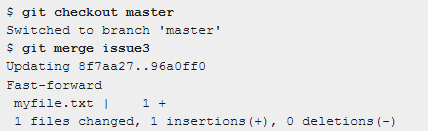
解决冲突后提交

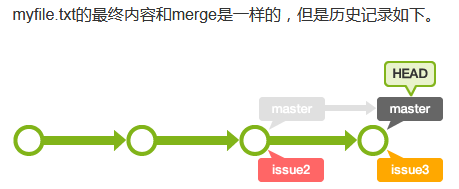
rebase的时候，修改冲突后的提交不是使用commit命令。而是使用rebase命令

指定--continue选项。若要取消rebase，指定--abort选项



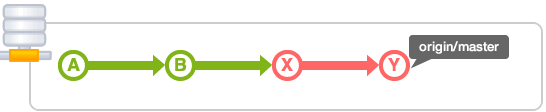


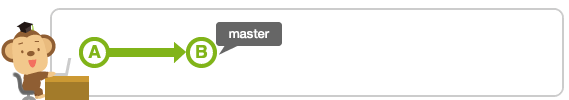




## 远端数据库

### pull





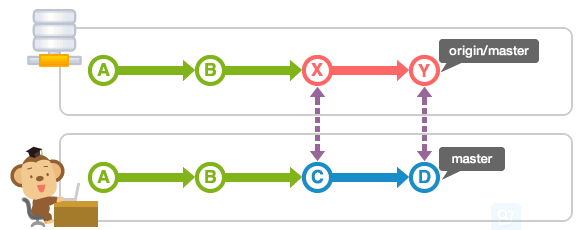
这时只执行fast-forward合并。图中的master是本地数据库的master分支，origin/master是远程数据库的origin的master分支。



显然pull会进行远程master和本地master的合并，这没有问题。

但是远程和本地都有新的历史记录就需要双方合并修改。

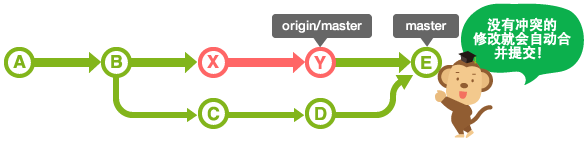
执行pull合并



执行pull合并

如果没有冲突就会自动创建合并提交

吐过发生重入就要先解决冲突再手动提交



### fetch

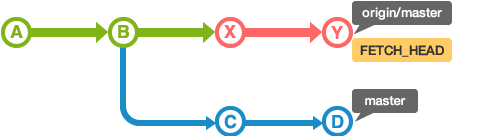
执行pull，远程数据库内容就会合并到本地。

但是有时候只想确认本地数据库的内容而不想合并。

这种情况下就用fetch

**执行fetch就可以取得远程数据库的最新历史记录**

**取得的提交会导入到没有名字的分支，这个分支可以从名为FETCH\_HEAD的退出**



远程主分支B的时候创建本地分开展到C,D

远程继续发展到X,Y

如果想把远程数据库的内容合并到本地数据库可以合并FETCH\_HEAD，或者重新执行pull

合并后历史记录和pull相同

**实际上pull=fetch+merge**

### push

没什么可说的

## 标签

标签是为了方便的参考提交而给他标上通俗易懂的名称

Git标签的分类

轻标签和注解标签。

打上的标签是固定的，不能像分支那样可以移动位置。

轻标签

添加名称

注解标签

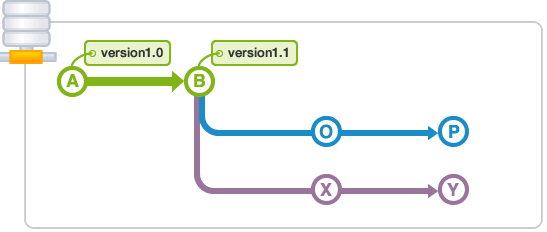
添加名称

添加注解

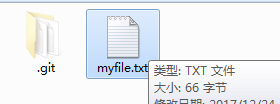
添加签名

一般，轻标签是为了在本地暂时使用或一次性使用的

注解标签是发布的时候使用的标签



### 事前准备







### 添加轻标签

$ git tag <tagname>

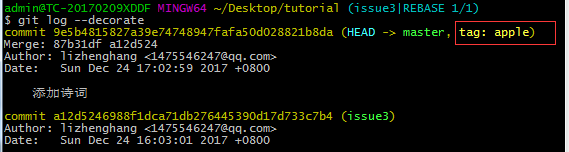
$ git tag apple

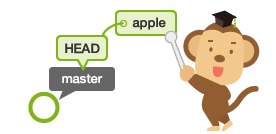
如果没有加参数就会显示标签列表

$ git tag

apple

如果在log命令添加 --decorate选项执行，就可以包含标签资料的历史记录





### 添加注解标签

如果要添加注解标签，可以加上参数-a选项执行。

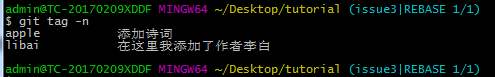
$ git tag -a <tagname>

执行后会启动编辑区，请输入注解，也可以使用-m参数来添加注解

$ git tag -am "连猴子都懂的Git" banana

显示标签列表和注解

$git tag -n 就会



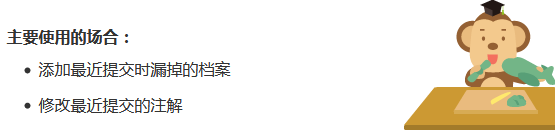
### 删除标签

$ git tag -d <tagname>

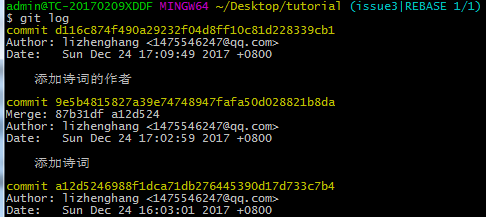
## 改写提交

### 修改最近的提交

修改同一个分支最近的提交内容和注解



确认历史记录



改写提交

git commit -amend 会显示最近一次提交的提交消息。你可以修改你想要的

然后再查看日志记录

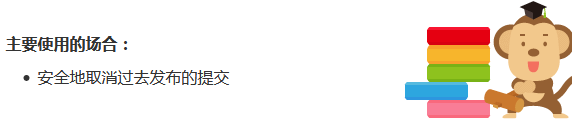


### 取消过去的提交

使用revert可以取消指定的提交内容。

使用后面要提到的rebase -i 或reset也可以删除提交

但是不能删除已经发布的提交



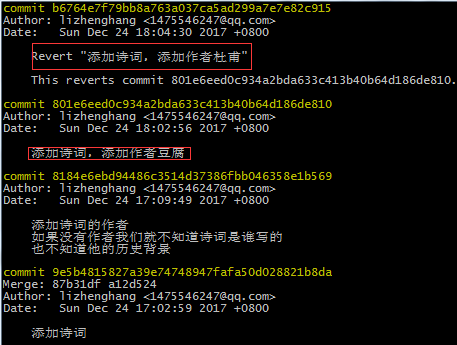
首先我们有如下历史记录



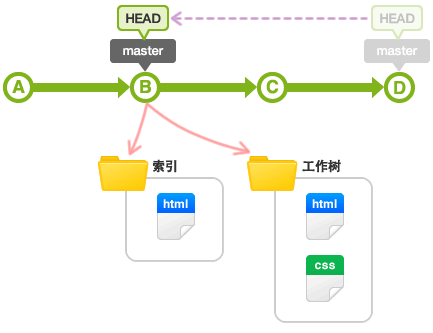
git revert HEAD

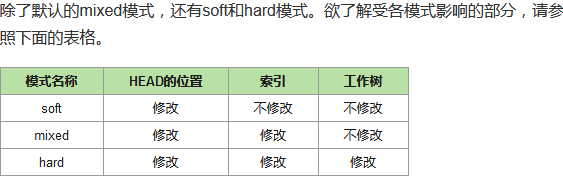


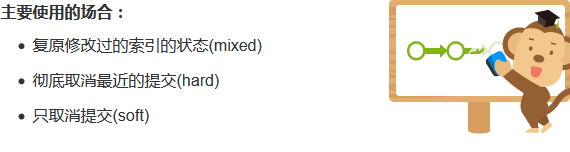
查看历史记录



### 遗弃提交





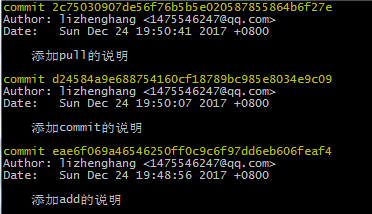


reset可以遗弃不再使用的提交

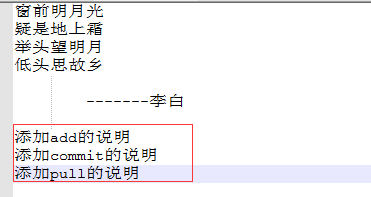
执行遗弃时需要根据影响的范围而指定不同的模式。

可以指定是否复原索引或工作树的内容

首先假设有如下提交



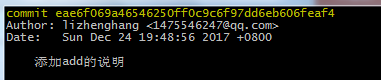
三个提交的内容如下



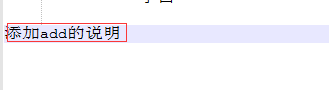
向前移动两个提交

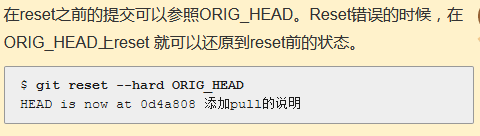


git log



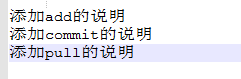
果然向前退回了两个提交

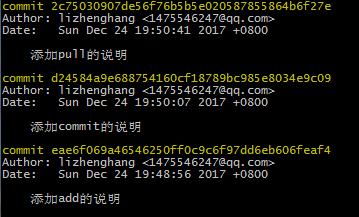




git reset --hard ORIG\_HEAD

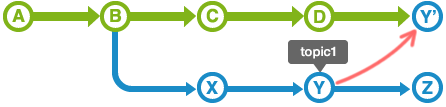


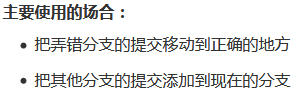




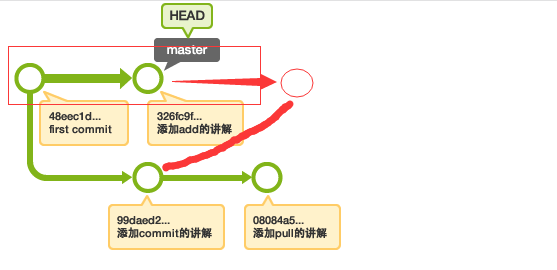
### 提取提交

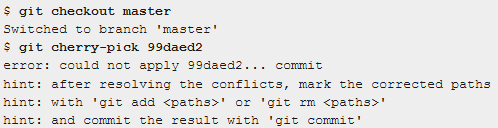
在cherry-pick，您可以从其他分支复制指定的提交，然后导入到现在的分支





案例：

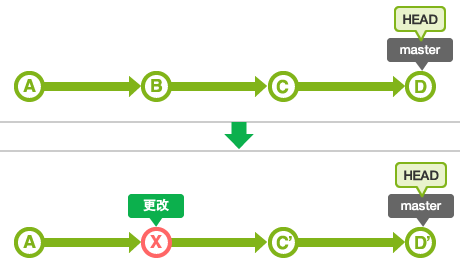




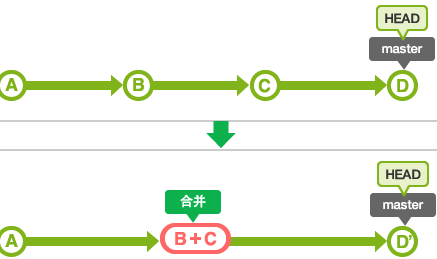


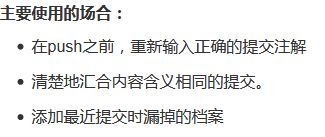
### 改写提交的历史记录

在rebase指定i选项，您可以改写、替换、删除或合并提交。



还可以





案例：

假设有如下三个提交



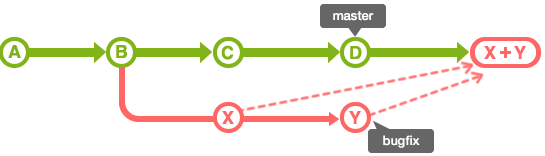
汇合过去的提交

git rebase -i HEAD~~

### 汇合分支上的提交，然后一同合并到分支

我们介绍一下merge的特殊选项：squash

用这个选项指定分支的合并，就可以把所有汇合的提交添加到分支上。



**主要使用的场合：**

* 汇合主题分支的提交，然后合并提交到目标分支。

### commit --amend

### revert

### reset

### cherry-pick

### 用rebase -i 汇合提交

### 用rebase -i 修改提交

### merge -squash

# Git索引