**Git使用规范**



**Version Control**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Change Description** |
| 1.0 | 07/24/2015 | Jonathan Dai | 文档创建 |



1 Git 2

1.1 Git简介 2

1.2 Git与Svn差别比较 2

1.3 Git的本质及优势 2

1.3.1 版本数据存储结构 2

1.3.2 分布式设计 2

2 Git使用规范 2

2.1 客户端选择 2

2.2 正确的注释 3

2.3 使用合适的粒度进行提交 3

2.4 Git分支使用规范 3

2.4.1 适当命名分支 3

2.4.2 保持分支目标明确 3

2.4.3 适时关闭分支 3

2.4.4 禁止使用fast-forward 3

2.4.5 在合并提交之前进行review 5

2.5 Git tag清理规范 6

2.5.1 原因 6

2.5.2 流程 6

2.6 其他版本控制软件中不常见的概念 7

2.6.1 rebase 7

# 



# Git

公司的所有项目资源管理，含源代码（前后端）、静态资源（html、js、css）、美术资源（psd）、美术导出（swf），等等系列的资源和成品，都使用Git进行版本管理。

下文会简单介绍Git是什么，及简单的使用规范。

Git中文教程书：<https://git-scm.com/book/zh/v1>

## Git简介

和Svn一样属于版本管理工具，但是远胜过Svn，无论是功能还是速度。

## Git与Svn差别比较

Svn：

* 中心集中式版本控制工具，服务器挂掉大家都没得玩，回家无法工作
* 文件式版本管理，历史版本信息粒度细到文件，导致大量碎片文件产生
* 创建分支困难，每一次创建分支就是产生了一次分支文件的整个拷贝，大量耗时
* 切换分支困难，同上，大量文件切换，耗时
* 速度慢，大的版本库速度将非常缓慢

GIT：

* 分布式架构，离开服务器一样可以工作，当然最后还是要同步到服务器上
* 数据库模式版本管理，历史信息集中管理，对文件系统负担小
* 创建、切换分支基本上能达到秒级，非常快
* 速度快，使用体验好

## Git的本质及优势

### 版本数据存储结构

上文也提到了GIT的版本控制是使用数据库形式的，而不是以文件为单位进行数据管理的。这样对文件系统的负担非常小，切换版本，创建分支等操作非常快。

细节请参考：[https://git-scm.com/book/zh/v1/起步-Git-基础](https://git-scm.com/book/zh/v1/%E8%B5%B7%E6%AD%A5-Git-%E5%9F%BA%E7%A1%80)

### 分布式设计

简单来说就是每次在Git库上的操作都会落在本地，不需要每次都和服务器通讯。也就是说无论是断网，还是你在家办公，你都可以无需担心版本控制问题。

同样的，在本地的操作完成了之后也需要和真正的线上版本库同步，向上提交你的改动称为push。向你的本地库获取线上别人的最新改动称为pull。

细节请看上文提到的Git教程书。



# Git使用规范

基础的使用方法请参考Git教程书。这里提几点平时比较容易出现的问题。

阅读：[https://git-scm.com/book/zh/v1/Git-基础](https://git-scm.com/book/zh/v1/Git-%E5%9F%BA%E7%A1%80)

## 客户端选择

请选择sourcetree作为你进行Git使用的客户端：<https://www.sourcetreeapp.com/>

## 正确的注释

请在你每次进行提交的时候写上本次提交的注释。注释内容必须让阅读的人了解你这次提交是做什么的，大致上改了哪些内容。

错误：

* Update code
* Update document
* Fix bug
* Review code

大部分的程序员写的注释都是上述级别之列，这种注释毫无意义，看完之后仍旧不知道这个提交究竟做了点什么。

正确：

* Add SQL security check in SqlBuilder class
* Add documents of error handling part in README.md
* Fix SzRedisModel serializing bug, since the return value from SzVoSerializer has been changed from JSON string to array
* Code fix after the review of PlantService part, fixed part: …

## 使用合适的粒度进行提交

不要改一行代码就进行一次提交，也不要每次写个一个礼拜代码才进行一次提交。“粒度”，或者说“度”的把控，也是一名出色程序员需要掌握的能力。

最佳的粒度是在完成一小块功能块改动之后，进行一次提交。

比如说我改框架的报错系统，我将该系统内的所有内容重构完，然后把周边受到影响的内容也一并修正，测试完成，就可以做一次提交了。

## Git分支使用规范

同样这里仅提几点问题。

阅读：[https://git-scm.com/book/zh/v1/Git-分支](https://git-scm.com/book/zh/v1/Git-%E5%88%86%E6%94%AF)

### 适当命名分支

以bug-xxx或feature-xxx、milestone-xxx，这样以特定功能为前缀的命名方式命名新打开的分支，这样能保证观察的人一目了然了解当前的分支究竟是用来做什么的。

### 保持分支目标明确

每次从一个主干上分出一个分支用于工作的时候，必须非常明确在这个分支里应该做的工作是什么，然后在完成后就进行合并，保证工作尽量敏捷，迅速，避免不必要的合并工作。

不要长期保持一个以自己姓名为命名的分支，这样其他工作者会不知道这个分支里的代码究竟包含了哪些东西，也不知道究竟自己应不应该合并该分支。分支应该目标清晰。

### 适时关闭分支

除了master主分支之外，其他的任何分支都是非长期分支，应该在其负责的工作完成之后被关闭。

### 禁止使用fast-forward

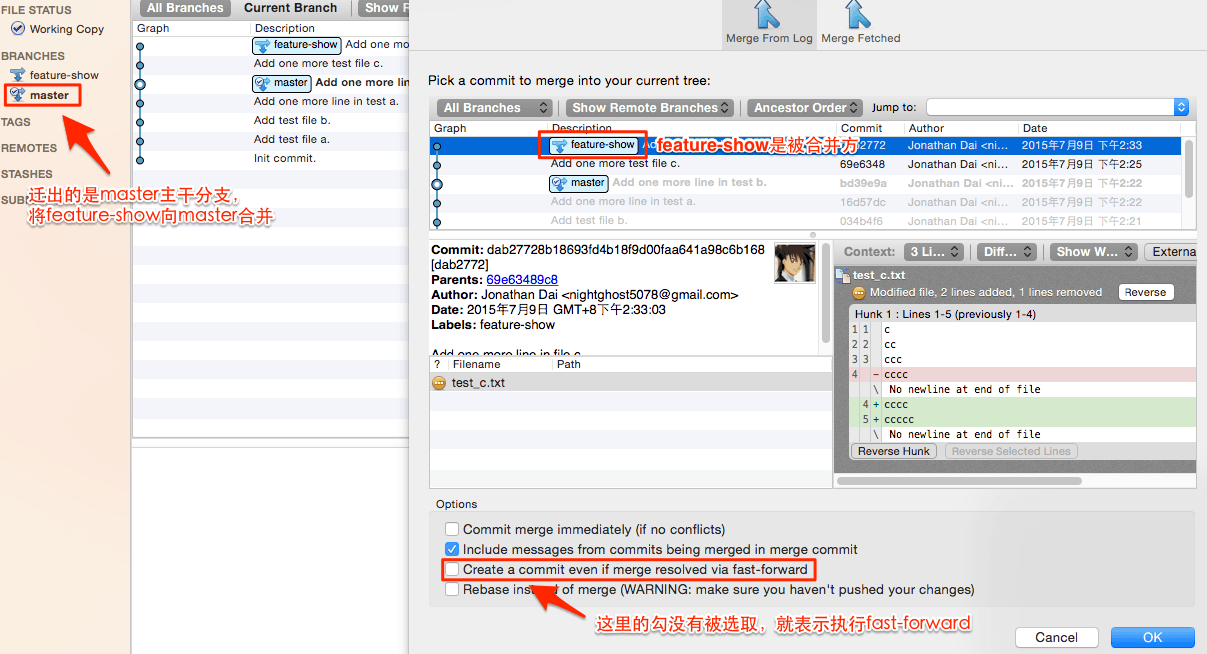
阅读：[https://git-scm.com/book/zh/v1/Git-分支-分支的新建与合并](https://git-scm.com/book/zh/v1/Git-%E5%88%86%E6%94%AF-%E5%88%86%E6%94%AF%E7%9A%84%E6%96%B0%E5%BB%BA%E4%B8%8E%E5%90%88%E5%B9%B6)，搜索：“Fast forward”，理解什么是Fast forward。

简单来说Fast forward意味着你分支的合并工作不会在主分支上留下任何打开过分支的痕迹，工作分支在合并并关闭之后就像完全没有被打开过一样。主分支的提交记录会非常“光滑”。但这样做会造成分支创建关闭等提交信息的丢失，一般不建议这么做。

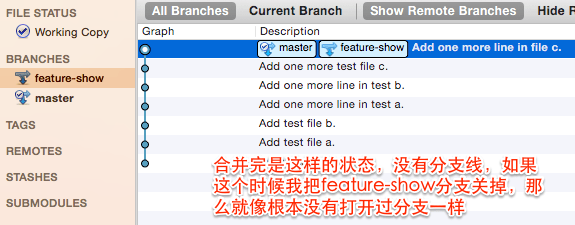
sourcetree范例：

#### 使用fast-forward的例子

合并操作界面：

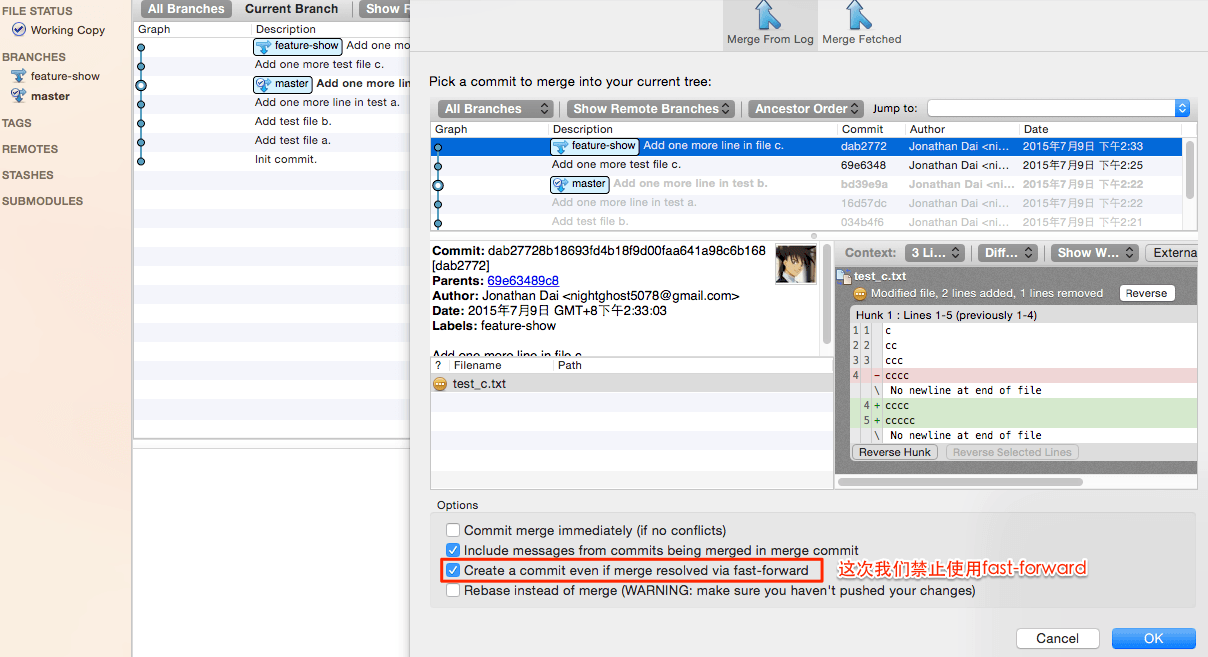


合并完成：

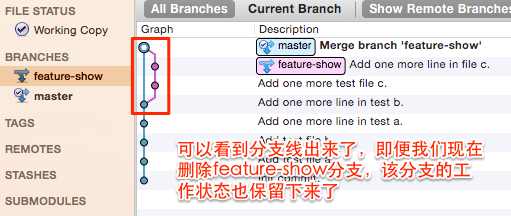


#### 禁止fast-forward的例子

合并操作界面：



合并完成：

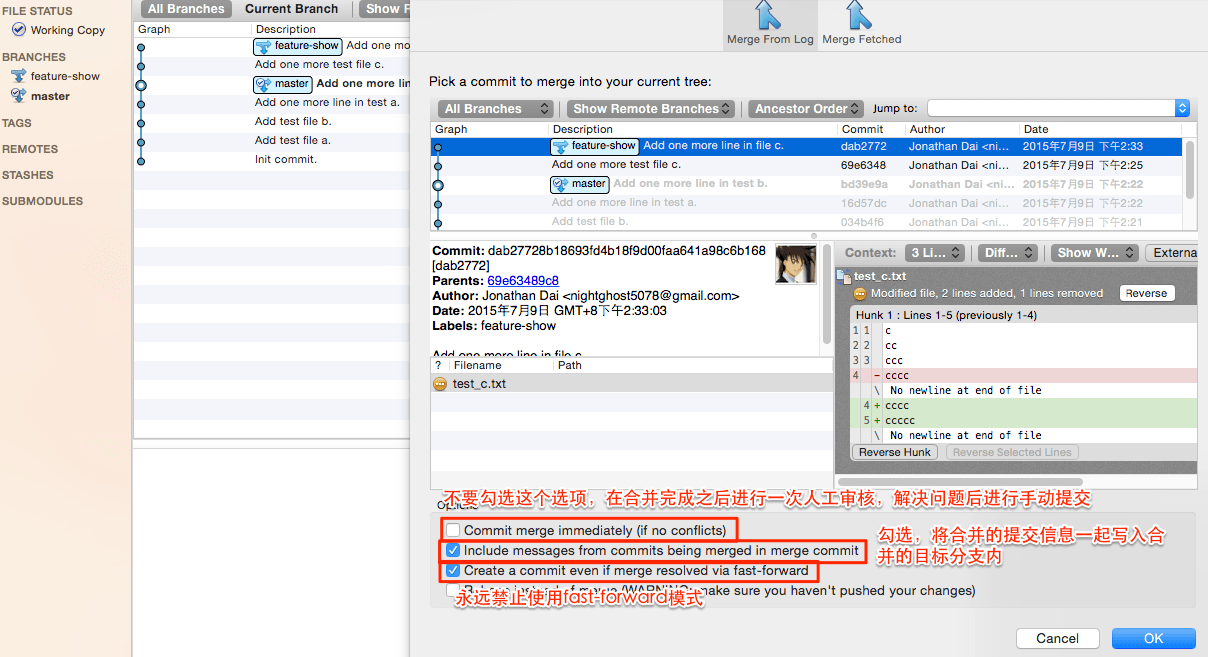


### 在合并提交之前进行review

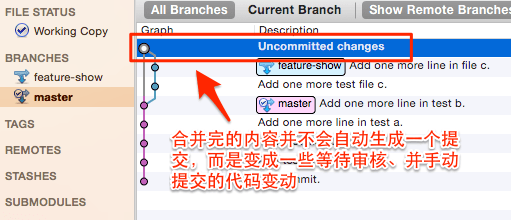
合并的时候需要了解的3个选项：

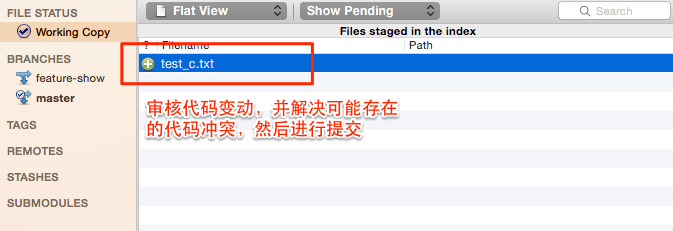
* 不要在合并后让软件自动帮你提交，反选该选项
* 将合并的所有提交信息一并写入到目标分支的提交内，勾选
* 禁用fast-forward，勾选

范例：



合并完成后不会自动提交，需要人工最后审核、解决冲突后进行手动提交：





## Git tag清理规范

### 原因

因为每次版本发布都需要针对当前发布的版本进行一次tag标记，可以想象在项目持续一段时间之后，版本tag将会泛滥。所以有必要定期对无用的tag进行清理。这里我们制定一下清理的规范。

### 流程

* 所有工作中的人员，将需要提交的内容全部提交并推送到服务器
* 由Leader手动删除大量无用的tag，并推送提交到服务器上
* 所有使用该版本库的人员，都删除本地的版本库，再次clone一份新的版本库（这个操作是为了防止那些没有手动删除tag的人员的本地库再次将tag提交上去，客户端的某些设置会强制每次提交都推送tag，这样就会造成反复操作）
* 所有工作中的人员继续中断之前的工作

## 其他版本控制软件中不常见的概念

### rebase

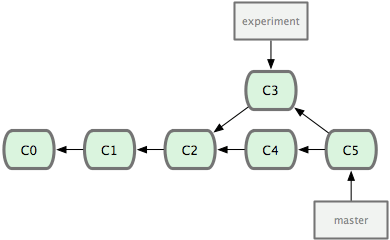
简易说明，rebase和merge基本上服务于同一个目的：即不同分支之间的代码融合。

详细的文档参见：[https://git-scm.com/book/zh/v1/Git-分支-分支的衍合](https://git-scm.com/book/zh/v1/Git-%E5%88%86%E6%94%AF-%E5%88%86%E6%94%AF%E7%9A%84%E8%A1%8D%E5%90%88)

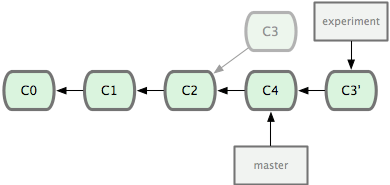
这里简单说下几点结论：

#### rebase和merge的差别

merge的情况：



rebase的情况：



在master分支里rebase入experiment分支的改动内容，等于是根据master分支当前的状态（以master分支当前的代码为基底），将experiment分支的改动做了一个patch，补丁入master分支中。

优势在于进行合并的目标分支会非常干净（master），被合并的分支会直接消失，而内容已经进入目标分支内。这个操作有点类似于合并的时候使用fast-forward，然后将被合并的分支删除。

当然，我们之前也说了，为了保留操作痕迹，fast-forward我们是禁止使用的，这里的rebase也一样。

#### 冲突解决上的差异

从刚才一点的描述中，大家其实可以看出，rebase是以目标分支为基础进行的补丁。也就是说，代码合并的冲突解决，是由合并方解决的（experiment），而不是被合并方（master）。即把解决分支补丁同最新主干代码之间冲突的责任，化转为由提交补丁的人来解决。这样维护者就不需要做任何整合工作了。

这在github等开源软件协同工作中非常有用，因为pull request每次都由维护者来合并会对维护者造成大量的工作请求。

