Git

第 1 章 ： Git简介

1.1:git安装

从 [http://git-scm.com/](http://www.zhufengpeixun.cn/course/20/learn) 下载，然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！

查看帮助

git config help

git gelp config

Man git-config

查看配置

git config --local //当前仓库，级别最高

git config --global nuse.name //当前用户，级别第二

git config --system //当前系统,级别最低

修改全局配置(安装完成后第一时间配置)

git config --global user.name "duanshanghanqing" //添加用户名属性

git config --global user.email ["duanshanghanqing@163.com"](mailto:\"duanshanghanqing@163.com\") //添加邮箱号属性

git config user.name //查看用户名

git config --get user.name //获取用户名

git config --list --global //列出global所有的变量

git config --global --unset user.name "duanshanghanqing" //删除用户名属性

git config --global --unset user.name //删除所有值

git config --list --global //列出所有的值

因为Git是分布式版本控制系统，所以每个机器都必须自报家门：你的名字和Email地址。

2.1:创建版本库

选择文件目录,鼠标右键选择Git bash(安装git-scm后会有此选项)打开命令行。

创建gitTest文件夹

mkdir gitTest

打开文件夹

cd gitTest

查看当前所在目录

pwd

初始化仓库

git init

查看文件目录

ls 或 ls -al

删除git管理

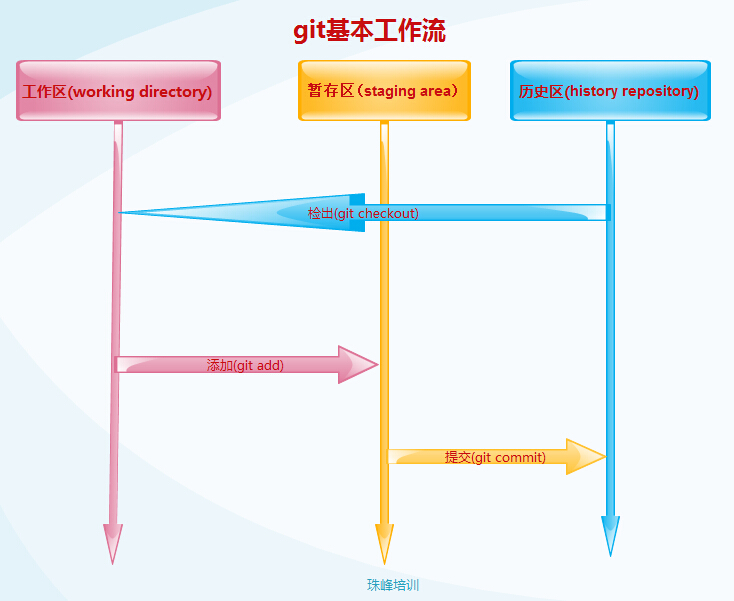
rm .git

2.2工作流

第一个是你的  ****工作区 （****工作目录），对应你的物理目录,树型结构，也就是编辑代码的地方，写代码的地方；

第二个是 ****暂存区****，工作区和历史提交的中间缓存，代表需要提交的工作状态，维护虚拟树型结构,临时保存你的改动；

最后是 **历史区**，树型结构，历史仓库,指向你最近一次提交后的结果。



2.3添加文件

创建文件

touch index.html 创建空文件

echo 1 > index.html 创建并向空文件写入1 这个数字

echo 2 >> index.html 向文件中追加 2 这个数字

编辑文件

vi index.html

随便在键盘上按个字符，等待变成 **INSERT** 输入模式，就可以编辑了

按一下**Esc**后。按 **:wq** 推出编辑

1.在工作区(本地)编辑修改文件，没提交到暂存区

使用git checkout -- 撤销以上的全部修改

git checkout -- index.html

查看文件内容

cat index.html

查看当前目录下文件的状态

git status

红色：代表在本地工作区，等待提交到暂存区。

绿色：代表在暂存区，等待提交到历史区

或

git status -s

M :表示已经被git跟踪的

??:没有被跟踪的

把****工作区文件**** 添加到 ****仓库中的暂存区****

git add index.html //把文件添加到暂存区

git add . //把git仓库下所有文件添加到暂存区

warning: LF will be replaced by CRLF in index.html.

The file will have its original line endings in your working directory.

//警告: index.html 中的 /n将会被/r/n替换掉，在原始目录中文件将保留最原始的换行符

//这个设置跟git-scm安装时的设置有关，因为在window下换行是\r\n,在linux下是\n

删除暂存区里的文件,回到在工作区的状态

git rm --cache index.html

把****暂存区文件**** 添加到 ****历史区****

git commit index.html -m"在index.htm 添加了34567890" //添加指定文件到历史区

git commit -m"添加全部文件到历史区" //添加全部文件到历史区

查看提交历史纪录

git log

控制台打印

commit 5955e37b1afcd2212e942049a1fb58ca727a4789（提价时记录的id）

Author: gaojunfeng <490771426@qq.com> (作者，提交人)

Date: Tue Jan 19 00:11:40 2016 +0800 (提交时间)

+ 添加了一个index.html文件 (提交文件的注释)

英文状态下按 q键 推出查看log

使用git客户端工具查看提交记录

gitk

**总结**

可以多次add不同的文件,然后commit一次提交很多文件

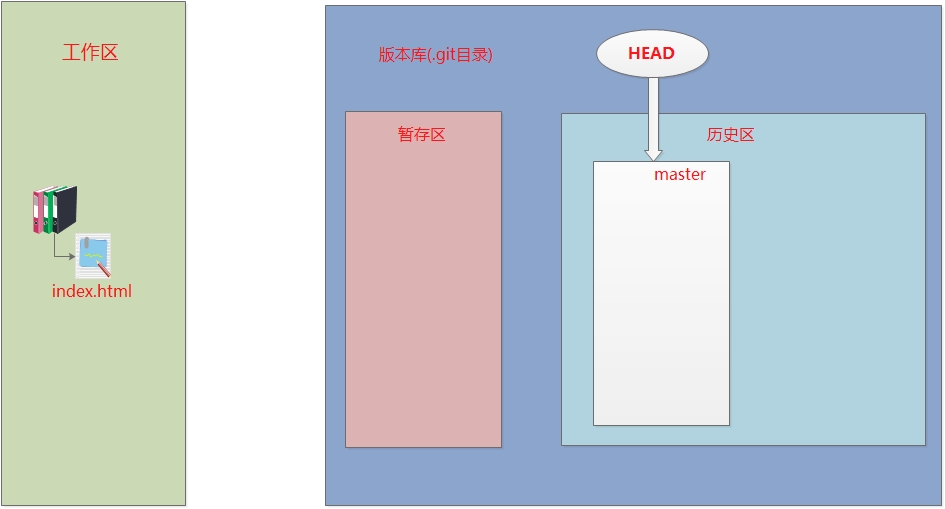
git add index1.html

git add index2.html

git commit -m"添加多个文件到历史区"

2.4三个区域

版本库（Repository）



分支和HEAD的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

练习一下

1.修改index.html文件

vi index.html

End :wq //退出

1. 创建text.js文件,并添加内容

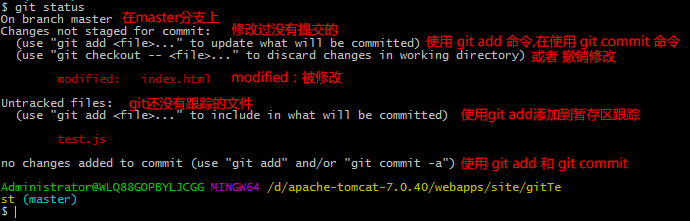
touch test.js

vi test.js

End :wq //退出

1. 查看文件状态

git status



git add命令把所有修改添加到暂存区（Stage），然后，执行git commit就可以一次性把暂存区的所有修改提交到历史区（默认master分支）。

2.5版本比较

# **查看内容差异:**

git diff

# $ git diff //查看工作区和暂存区中的文件区别

**diff --git a/index.html b/index.html** //比较两个index.html

**index e53eaa1..97b3d1a 100644**

**--- a/index.html** //--- 代表源文件

**+++ b/index.html** //+++ 代表目标文件

@@ -7,4 +7,9 @@ //差异按照差异区域进行组织，每个差异区域的第一行都是定位语句，由@@开头，@@结尾

7

8

9

-0

+10

+11

+12

+13

+14

+15

添加到暂存区，在提交到历史区,并查看状态

git add index.html

git add test.js

git commit -m"添加index.html 和 test.js文件"

git status

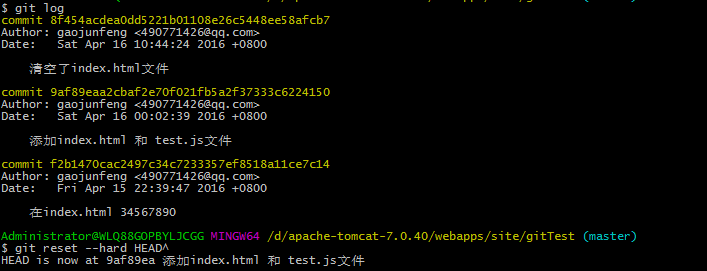
# On branch master

nothing to commit (working directory clean)

//Git告诉我们当前没有需要提交的修改，而且，工作目录是干净\*\*（working directory clean）\*\*的。

**版本回退**

在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log命令查看：

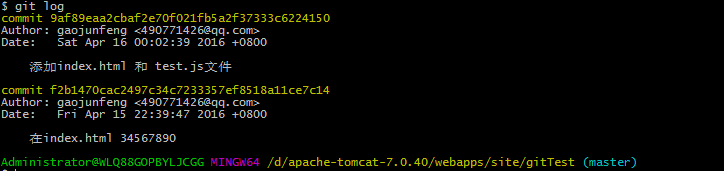


git log命令显示从最近到最远的提交日志，我们可以看到2次提交  
如果嫌输出信息太多，看得眼花缭乱的，可以试试加上**git log --oneline**命令

准备把index.html回退到上一个版本，也就是“**添加index.html 和 test.js文件”**的那个版本，怎么做呢？

使用git reset命令

git reset --hard HEAD^ //把历史区重置到上一个提交

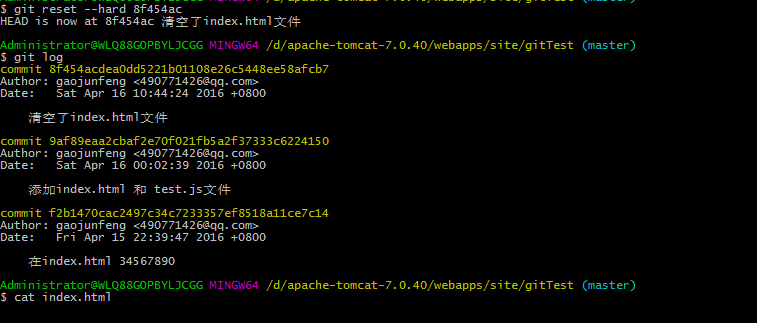


**已经把历史区重置到上一个版本提交，就没有 “清空了index.html文件” 的记录**

想再回到**“清空了index.html文件”**记录，怎么办？？？？？

只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到**“清空了index.html文件”** 的记录的commit id是8f454acdea0dd5221b01108e26c5448ee58afcb7，于是就可以指定回到未来的某个版本：

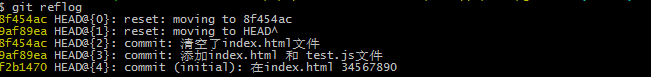
git reset --hard 8f454ac //前7位就可以了



就回到了**“清空了index.html文件”**的记录

Git提供了一个命令git reflog用来记录你的每一次命令：

git reflog



就可以看到对文件历史记录的操作。使用 git reset --hard 8f454ac 命令操作每一条记录

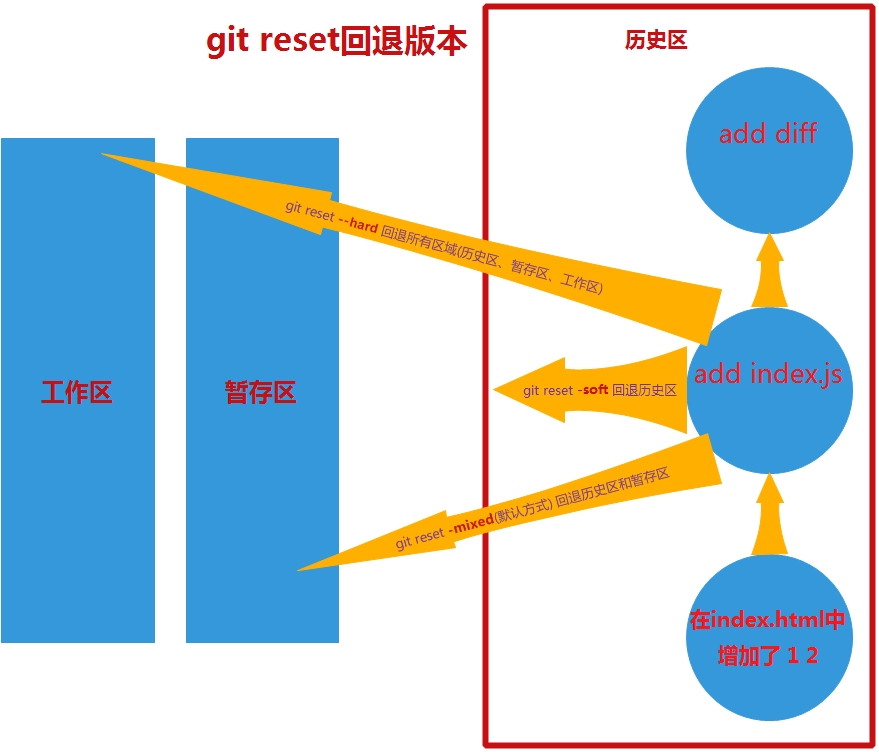
总结：HEAD 指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。

# git reset扩展

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, 工作区 不变，回退历史区 和 暂存区

git reset –soft：回退到某个版本，只回退了历史区的信息，工作区 和 暂存区 都不变

git reset –hard：彻底回退到某个版本，回退 工作区 、历史区 和 暂存区 。



# 过滤日志

git log --oneline -2 显示多少行

git log --oneline -2 --grep='index.html' 过滤

git log --oneline -2 --author='root' 作者

git log --oneline -2 --author='root' --before='2015-6-6'

git log --oneline -2 --author='root' --before='one week'

git log --oneline -2 --author='root' --before='three days' 日期之前

git log --oneline -2 --author='root' --graph 图形化

**撤销修改**

**例子:**

**1.先查看原文件**

**cat index.html**

1. **修改文件，向文件里添加hello world**

**vi index.html**

**3.查看文件状态**

git status

1. 查看文件内容，你会发现都多了hello world

cat index.html

5.撤销到没有添加hello worldzhuangtai

git checkout -- index.html

6.查看一下文件内容，hello world已经不见了

cat index.html

小结

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。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，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。

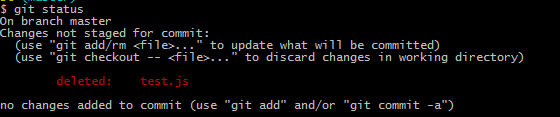
场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

**删除文件**

实战一下，先把前面添加test.js删除,可以在资源管理器中删除也可以用 rm

rm index.js

git status命令会立刻告诉你哪些文件被删除了：



现在你有两个选择！！！！！！！！！！！！！！！！

# 一．是误删除了：

但暂存区里还有呢，所以可以很轻松地把误删的文件从暂存区恢复到工作区：

git checkout -- test.js

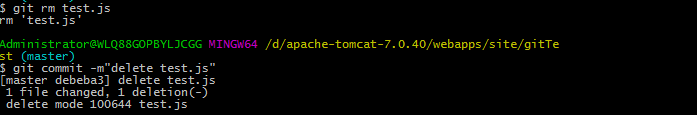
无论工作区是修改还是删除，git checkout都可以“一键还原”。

# 二．是确实要删除：

要是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

git rm test.js

git commit -m"delete test.js"



## 最新的历史

git log --onelin

2.6分支管理

**创建与合并分支**

1. 查看分支

git branch

\* master

会列出所有分支，当前分支的前面会标一个\*号。

1. 创建分支

git branch dev

1. 切换分支

git checkout dev //dev为要切换的分支名

1. 创建dev分支，然后切换到dev分支：

git checkout -b dev

1. 合并分支(把dev分支上的工作成果合并到merge 支上)

git merge dev

git merge命令用于合并指定分支到当前分支。合并后，再查看index.html的内容，就可以看到，和dev分支的最新提交是完全一样的。

例子

git checkout -b dev //切换分支到dev分支

vi index.html //编辑文件

git add index.html //添加到暂存区

git commit -m"add 学习git分支" //添加到历史区

现在，dev分支的工作完成，我们就可以切换回master分支

git checkout master //切换到master主分支

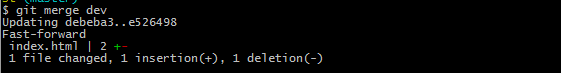
切换回master分支后，再查看一个index.html文件，刚才添加的内容不见了！

因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：

我们把dev分支的工作成果合并到master分支上：

git merge dev //合并dev分支到当前分支

再查看一个index.html文件，就看见在dev分支上添加的内容了！



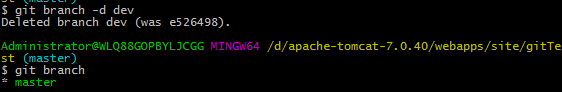
git merge命令用于合并指定分支到当前分支。合并后，再查看index.html的内容，就可以看到，和dev分支的最新提交是完全一样的。注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

当然，也不是每次合并都能Fast-forward，我们后面会将其他方式的合并。

1. 删除分支

git branch -d dev

删除后，查看git branch，就只剩下master分支了：



1. 放弃这次合并

git merge --abort

1. 小结

Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

放弃这次合并: git merge --abort

**解决冲突**

例子:

新建分支,并切换分支

git checkout -b feature

修改index.html,在文件内追加 “解决冲突” 四个字

vi index.html

在feature分支上提交

git add index.html

git commit -m"feature分支追加 解决冲突"

切换回master分支

git checkout master

master分支修改index.html文件， 追加“abcde” 字符

vi index.html

master分支提交

git add index.html

git commit -m "master分支追加 abcde"

**问题就出来了，master分支和feature分支都有提交**

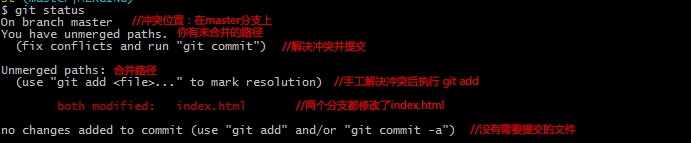
分支合并

git merge feature

**V$(7[`L{H6UY(IC7(7B{0ZK**

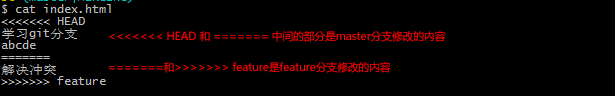
果然冲突了！Git告诉我们，index.html文件存在冲突，必须手动解决冲突后再提交。

git status也可以告诉我们冲突的文件：



手动决绝冲突

打开index.gtml文件



删除调用 不想要的部分，在提交。

git add index.html

git commit -m"master分支添加 abcde"

用带参数的git log也可以看到分支的合并情况：

git log --graph --oneline

最后，删除feature分支

git branch -d feature

Ok:练习完成

**分支策略**

通常，合并分支时，如果可能，Git会用Fast forward(快速前进)模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用Fast forward(快速模式)模式，Git就会在merge(合并)时生成一个新的commit(事务提交)，这样，从分支历史上就可以看出分支信息。

**下面我们实战一下--no-ff方式的git merge：**

练习：

1. 在master分支下清空index.html文件，添加到暂存区，在添加到历史区
2. 创建并切换dev分支

git checkout -b dev

1. 修改index.html文件，并提交一个新的commit(事务提交)

vi index.html //添加hello world

git add index.html

git commit -m"在dev分支上给index.html文件添加了 hello world"

1. 我们切换回master 分支上去

git checkout master

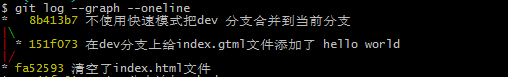
1. 准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward(快速模式)：

git merge --no-ff -m "不使用快速模式把dev 分支合并到当前分支" dev

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

1. 合并后，我们用git log --graph --oneline看看分支历史：

**git log --graph --oneline**



在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



小结:

Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

工作中就使用这种方式合并分支：git merge --no-ff -m "不使用快速模式把dev 分支合并到当前分支" dev

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**隐藏工作台**

使用场景：

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支 来修复，修复后，合并分支，然后将临时分支删除。

创建模拟环境

git branch //查看当前有没有dev分支，没有就创建dev分支(git checkout -b dev)

touch a.js //创建一个a.js文件

vi a.js //添加hello world 文本

git add a.js//添加到暂存区

git status //查看工作区文件状态 。此时已经模拟出我们正在dev分支进行工作

接到一个修复一个bug的任务很自然的想到创建一个issue(问题分支)来修复它。

**Git还提供了一个stash功能，可以把当前工作环境“储藏”起来，等bug修复以后在恢复过来：**

git stash //把当前工作分支(dev)“储藏”起来

git status //查看工作区文件状态。就没有任何东西了，工作区干干净净，就可以创issue(问题分支)来建修复bug了

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时问题分支：

git checkout master //切换到主分支

git checkout -b issue //创建问题分支，并切换到issue分支

vi index.html //编辑index.html文件，添加 “修复bug”

git add index.html //添加暂存区

git commit -m"在index.html文件中添加修改bug文本" //添加历史区

# 到此bug修复完成，切换到master分支，并完成合并

git checkout master //切换到主分支

git merge --no-ff -m "把issue问题分支和并到当前master主分支上" issue //把issue(问题分支)合并到当前切换回来的主分支

git log --graph --oneline //查看分支历史

# 最后删除issue分支

git branch -d issue //删除问题分支

Bug修复了，要切换到dev分支干活了！

git checkout dev//切换到最开始的工作开发分支

git status //查看文件状态，发现什么文件都没有



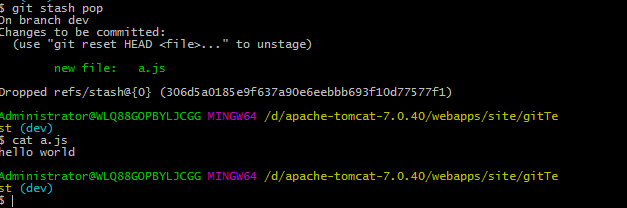
工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：

git stash list //查看 “储藏”起来的文件，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

**一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；**

**二是用git stash pop，恢复的同时把stash内容也删了：**

git stash pop //恢复被“储藏”起来的文件



就和当初正在开发的一样了

# 小结

* git stash list 查看一下有哪些stash
* git stash apply 应用某个stash而不删除stash
* git stash pop 应用某个stash并且删除stash
* git stash clear 一次性清理所有的stash  
  从而回到工作现场

2.7什么是标签

发布一个版本时，我们通常先在版本库中打一个标签，这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那 个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和 删除标签都是瞬间完成的。

# **在Git中打标签非常简单**

首先，切换到需要打标签的分支上：

git checkout master

# 用命令git tag查看所有标签

git tag

敲命令git tag 就可以打一个新标签：

git tag v1.0 //v1.0为版本号

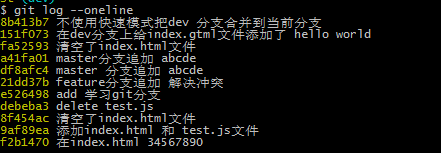
删除标签

git tag -d v1.0

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了：

git log --oneline



比方说要对 151fo73 这次提交打标签，它对应的commit id是151fo73 ，敲入命令：

git tag v0.9 151fo73 //v0.9 :为指定的版本号 151fo73：为要打标签的记录

再用命令git tag查看标签：

git tag

注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签详细信息：

git show v0.9 //v0.9为要查看的标签版本

# 还可以创建带有说明的标签

git tag -a v0.8 -m "merge with no-ff" beebf73 //-a v0.8：指定标签名为 v0.8 。 -m "xxx" :-m指定说明文字。 beebf73:为要打标签的记录

# IMG_256

# **切换标签**

git checkout v0.8  因为tag指向了commit,所以也可以checkout, 他会提示你从head上进行了脱离

git checkout -b v2.0 创建一个叫2.0 名字的分支

git branch -d v2.0 删除标签名，也就是删除2.0分支

2.7远程仓库

**github站点和本地电脑的配置**

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经 有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

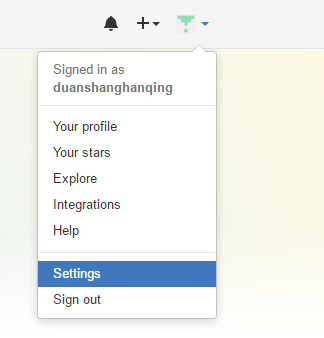
ssh-keygen -t rsa -C "490771426@qq.com"

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

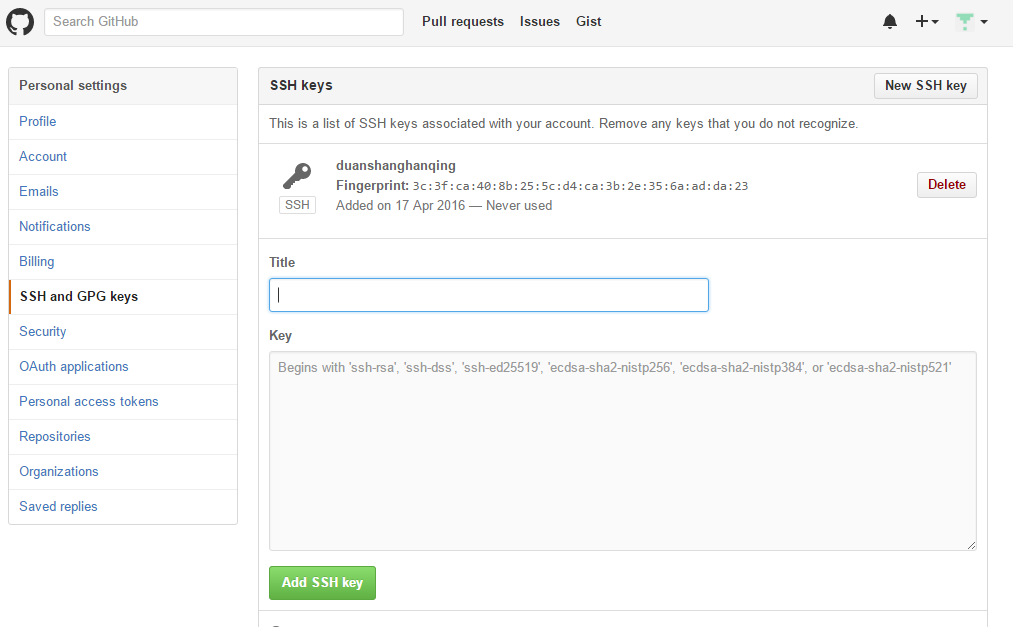
首先，登陆GitHub，没有账号要创建账号，在右上角找到New repository按钮，创建一个新的仓库：

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥， 不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：



然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴电脑里.ssh文件夹下的 id\_rsa.pub文件的内容：



因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

**添加远程库**

****

告诉本地git仓库要关联的远仓库

git remote add origin https://github.com/duanshanghanqing/gitTest.git

下一步，就可以把本地库的所有内容推送到远程库上：

git push -u origin master

Counting objects: 36, done. //推送了36个对象

Delta compression using up to 4 threads.

Compressing objects: 100% (17/17), done.

Writing objects: 100% (36/36), 3.24 KiB | 0 bytes/s, done.

Total 36 (delta 3), reused 0 (delta 0)

To https://github.com/duanshanghanqing/gitTest.git

\* [new branch] master -> master //把本地的master分支推送到远程服务器

Branch master set up to track remote branch master from origin. //让master分支跟踪远程的服务器origin(源)的master分支

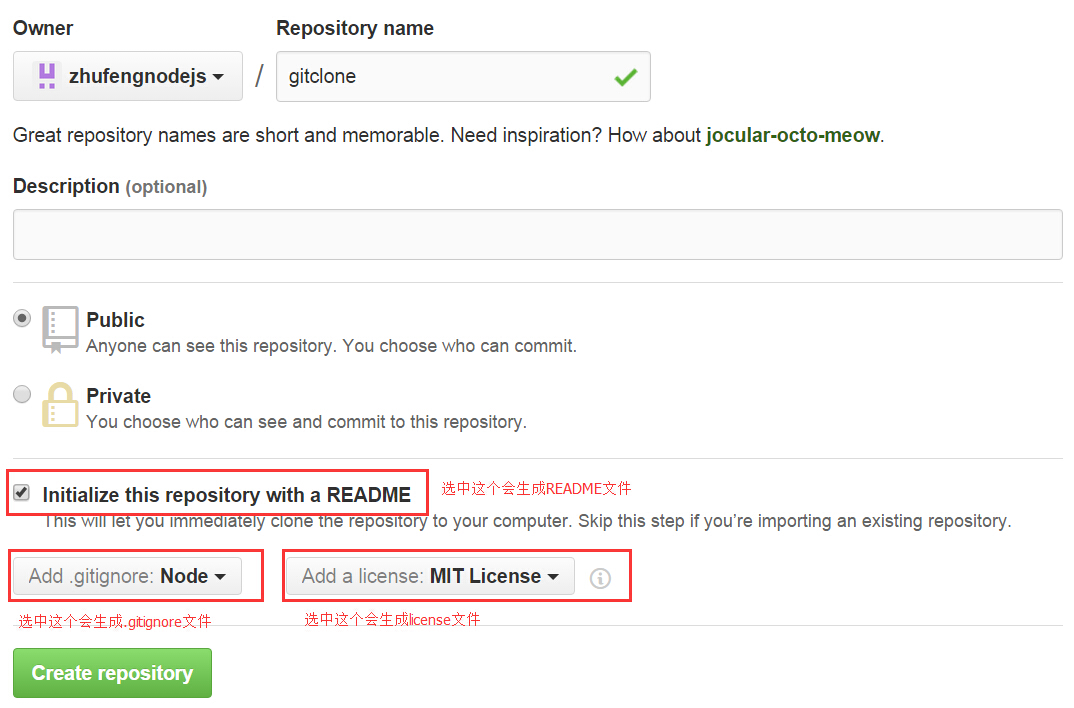
由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

**从现在起，只要本地作了提交，就可以通过命令：**

git push origin master

把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

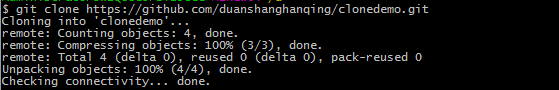
**从远程库克隆**



**克隆仓库命令**

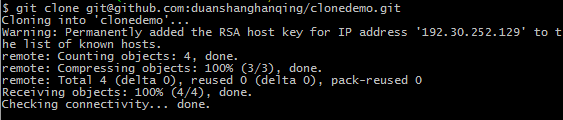
**1.使用https类型的连接**

git clone https://github.com/duanshanghanqing/clonedemo.git



**2.使用SSH类型的连接**

git clone git@github.com:duanshanghanqing/clonedemo.git



使用https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https

**多人协作**

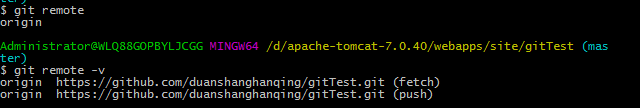
# 要查看远程库的信息，用git remote

**git remote //远程仓库名称**

**或者，用**

**git remote -v //显示更详细的信息：**

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。



上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

# **推送分支到远程库（本人）**

推送分支，就是把本地分支的提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

**git push origin master //把本地master分支推送到远程库**

如果要推送其他分支，比如dev，就改成

**git checkout -b dev //创建并切换到dev分支**

**git push origin dev //把本地dev分支推送到远程库**



但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

* master分支是主分支，因此要时刻与远程同步；
* dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
* bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
* feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

# **从远程库拉去分支（他人）**

# 多人协作时，大家都会往master和dev分支上推送各自的修改。

# 现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

练习：模拟小伙伴

在d盘下克隆项目。在上面我们已经向远程推送了dev分支

**git clone [git@github.com:duanshanghanqing/gitTest.git](mailto:git@github.com:duanshanghanqing/gitTest.git) //克隆项目**

**git branch //查看分支**



现在，小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

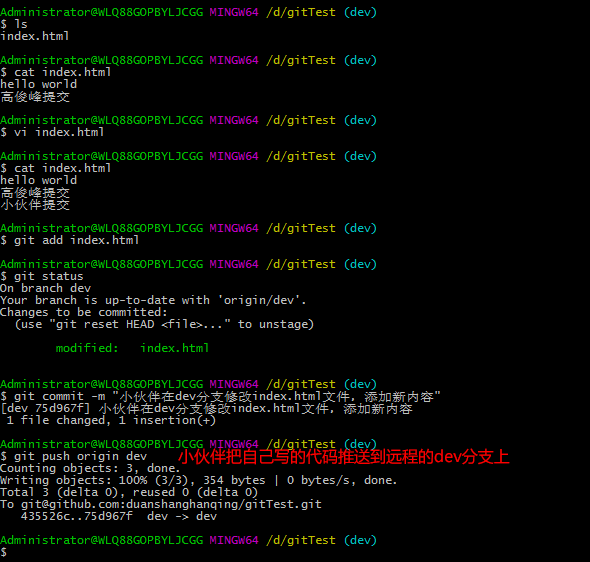
**git fetch origin dev //命令来把远程dev分支拉到本地**

**git checkout -b dev origin/dev //在本地创建分支dev并切换到该分支。并且把本地dev分支跟踪远程origin的dev分支**

**git pull origin dev //把远程dev分支上的内容都拉取到本地了**



现在，小伙伴就可以在dev上继续修改，然后，时不时地把dev分支push到远程：



**多人协作的工作模式**

通常是这样：

首先，可以试图用git push origin branch-name推送自己的修改；

如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；

如果合并有冲突，则解决冲突，并在本地提交；

没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！

如果git pull提示no tracking information，则说明本地分支和远程分支的链接关系没有创建，用命令

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

把本地分支和远程分支关联起来

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

总结：在团队开发中，每个人早上去了，都先拉去一下dev远程端代码。

下班了就向自己的分支上提交代码

**推送标签**

首先，切换到需要打标签的分支上：

git checkout master

# 用命令git tag查看所有标签

git tag

敲命令git tag 就可以打一个新标签：

git tag v1.0 //v1.0为版本号

删除标签

git tag -d v1.0 //v1.0为版本号

如果要推送某个标签到远程

git push origin v1.0 //v1.0为版本号

例子1：（向远程端推送标签）

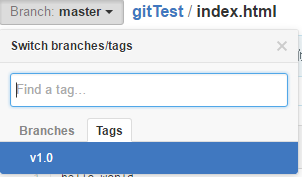
git tag v1.0 //创建v1.0 标签

git push origin v1.0 //把本地 v1.0 标签推送到远程端

或者

git push origin --tags //一次性推送全部尚未推送到远程的本地标签

推送成功到github网站上查看标签



例子2：(删除远程端的标签)

git tag -d v1.0 //先删除本地标签

git push origin :refs/tags/v1.0 //在删除远程端标签

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

**2.8自定义git**

在安装Git一节中，我们已经配置了user.name和user.email，实际上，Git还有很多可配置项。

比如，让Git显示颜色，会让命令输出看起来更醒目

git config --global color.ui true

这样，Git会适当地显示不同的颜色，比如git status命令：

文件名就会有颜色

**忽略特殊文件**

git全局忽略

git config --global core.excludesfile ~/.gitignore\_global

或者在git根目录下创建 .gitignore 文件

# 配置示例

# node\_modules/ //忽略跟node\_modules文件夹

.[oa] //忽略任何以o和a结尾的文件 ~ vim临时文件 vim~

.js //忽略以js结尾的文件

!test.js //除了test.js文件都可以忽略

\*/mk //任何目录 下的res目录 匹配0或多个目录

.idea //忽略.idea问价夹

最后一步就是把.gitignore也提交到Git

**配置别名**

有没有经常敲错命令？比如git status？status这个单词真心不好记。

如果敲git st就表示git status那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后st就表示status：

git config --global alias.st status

然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch

$ git config --global alias.co checkout  
 $ git config --global alias.ci commit  
 $ git config --global alias.br branch