|  |
| --- |
| **Git入门指南 V1.3** |
|  |
|  |
|  |
|  |
| 参加编撰人员：张婷 琚建飞  徐玲博 |
|  |
| **2017年11月17日** |
|  |

**文档修订记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 修改时间 | 修 改 人(编撰人） | 审 核 人 | 修改内容（编撰内容） |
| V1.0 | 2017-02-18 | 张婷 | 米老师、刘文彬 | 拟定初稿，列出大纲 |
| V1.1 | 2017-02-26 | 琚建飞、张婷 | 米老师、刘文彬 | [补充git研究意义、由来、命令、分支等内容](#_三、Git下载、安装、配置) |
| V1.2 | 2017-03-01 | 张婷、琚建飞 | 米老师、刘文彬 | [文档排版的编辑、各项内容的调整等](#_八、远程仓库管理) |
| V1.3 | 2017-11-17 | 徐玲博 | 米老师  李坤 | [IDEA配置Git](#_5.) |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**目录**

[为什么要用Git？ 5](#_Toc476146086)

[版本控制系统？ 5](#_Toc476146087)

[集中式版本控制系统？ 5](#_Toc476146088)

[分布式版本控制系统？ 5](#_Toc476146089)

[Git的由来？ 6](#_Toc476146090)

[Git下载、安装、配置？ 7](#_Toc476146091)

[1. 下载 7](#_Toc476146092)

[2. 安装 7](#_Toc476146093)

[3. 软件简介 8](#_Toc476146094)

[4．Git配置 8](#_Toc476146095)

[4.1用户信息 8](#_Toc476146096)

[4.2查看配置信息 9](#_Toc476146097)

[名词解释？ 10](#_Toc476146098)

[分支管理？ 12](#_Toc476146099)

[1．基本概念 12](#_Toc476146100)

[2．基本命令 13](#_Toc476146101)

[3．合并冲突 14](#_Toc476146102)

[4．Bug分支 14](#_Toc476146103)

[标签管理？ 15](#_Toc476146104)

[1.创建标签 16](#_Toc476146105)

[2.操作标签 16](#_Toc476146106)

[GitHub简介？ 17](#_Toc476146107)

[自定义Git？ 19](#_Toc476146108)

[忽略特殊文件 19](#_Toc476146109)

[配置别名 19](#_Toc476146110)

[配置文件 20](#_Toc476146111)

[Git基本命令？ 20](#_Toc476146112)

[简单应用？ 21](#_Toc476146113)

[版本管理 22](#_Toc476146114)

[1. 版本回退 22](#_Toc476146115)

[工作原理 26](#_Toc476146116)

[1.git的分层结构 26](#_Toc476146117)

[2.git的对象 27](#_Toc476146118)

[3. git的操作流程： 29](#_Toc476146119)

[远程仓库管理 29](#_Toc476146120)

[1.从远程库克隆 30](#_Toc476146121)

[2.将修改后的文件推送到远端服务器 30](#_Toc476146122)

[3.查看远程库 32](#_Toc476146123)

[4.新建远程仓库 33](#_Toc476146124)

[5.修改远程仓库 33](#_Toc476146125)

[6.删除远程仓库 34](#_Toc476146126)

[权限管理\*\*（见gitlab服务器搭建） 34](#_Toc476146127)

# 一、为什么要用Git？

## 版本控制系统

所有的版本控制系统，其实只能跟踪文本文件的改动，比如txt文件，网页，程序代码等等。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100kb改成了200kb，但到底改了啥，版本控制系统不知道，也没法知道。需要注意的是：Word格式的文件是二进制格式，因此，版本控制系统是没法跟踪word文件的改动的。因为文本是有编码的，比如中文常用的GBK编码，日文有Shift\_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

## 集中式版本控制系统

有CVS、SVN。速度慢，必须联网才能使用。SVN是目前用的最多的集中式版本控制系统。

## 分布式版本控制系统

Git、BitKeeper、Mercurial和Bazaar等。这些分布式版本控制系统各有特点，但最快，最简单的也最流行的依然是Git。

分布式版本控制系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是没有联网都可以正常工作，当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了。而SVN在没有联网的时候是拒绝干活的！

# 二、Git的由来

很多人都知道，Linus在1991年创建了开源的Linux，从此Linux系统不断的发展，已经成为最大的服务器系统软件了。Linus虽然创建了Linux，但linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发送给Linus，然后由Linus本人通过手工方式合并代码。

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈的不满，于是Linus选择了一个商业的版本控制系统BitKeeper，它的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干其实也不只他一个），被BitMover公司发现了（监控做的挺好），于是BitMover公司怒了，要收回Linux社区的免费试用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，恩，这是不可能的，实际情况是这样的：Linus花了两周的时间自己用C语言写了一个分布式版本控制系统，就是Git！一个月之内，Linux系统的源码已经由Git管理了！什么是厉害？这才叫厉害！

Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费且超级好用的Git了。

# 三、Git下载、安装、配置

## 下载

在使用Git前我们需要先下载Git，Git目前支持Linux/Unix、Mac和Windows平台。Git各平台的安装包下载地址为：<http://git-scm.com/downloads>。

## 2. 安装

[在Windows平台上安装Git](https://wenku.baidu.com/view/0c479e141eb91a37f1115c7b.html)（文库），双击运行即可。

<https://wenku.baidu.com/view/e9029bd5162ded630b1c59eef8c75fbfc67d9452.html>

这里安装的是msysGit

## 3. 软件简介

Git是一款免费、开源的分布式版本控制系统，最早有Linus创建，用于管理Linux内核开发，现已成为分布式版本控制的主流工具。

Windows平台下的Git软件致力于提供一种具有特色的轻量级本地工具集，可以让新手和经验丰富的大牛都很容易上手。安装完成后，我们会看到三个图标：Git Bash、Bit GUI、Git CMD。Git Bash是以命令行的形式来操作，会使\*NIX用户感到有家的温馨，就像git命令在Linux和Unix环境下运行。Git GUI是为了方便普通用户操作，所有的命令行操作都可以在GUI中找到对应的图形化操作。安装完成后，只需在开始菜单中单击Git Bash和Git GUI来操作它们。

## 4．Git配置

Git提供了一个叫做git config的工具，专门用来配置或读取相应的工作环境变量。这些环境变量，决定了Git在各个环节的具体工作方式和行为。

### 4.1用户信息

配置个人的用户名称和电子邮件地址：

$ git config --global user.name “Your Name”



$ git config --global user.email “Your eamil address”



如果用了—global选项，那么更改的配置文件就是位于你用户主目录下的.gitconfig文件中，以后你所有的项目都会默认使用这里配置的用户信息。如果要在某个特定的项目中使用其他名字或电邮，只要去掉—global选项重新配置即可，新的设定保存在当前项目的.git/config文件中。

**修改新的用户名和邮箱**

|  |
| --- |
| $  git config --global --replace-all user.email "输入你的邮箱"  $  git config --global --replace-all user.name "输入你的用户名" |

### 4.2查看配置信息

要检查已有的配置信息，可以使用git config –list命令：

$ git config –list

user.name=Micheal

user.email=15732625435@163.com

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如/etc/gitconfig和~/.gitconfig）,不过最终Git实际采用的是最后一个。

这些配置我们也可以在~/.gitconfig或/etc/gitconfig看到，如下所示：

vim ~/.gitconfig

[user]

Name = Micheal

Emai = 15732625435@163.com

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

$ git config user.name

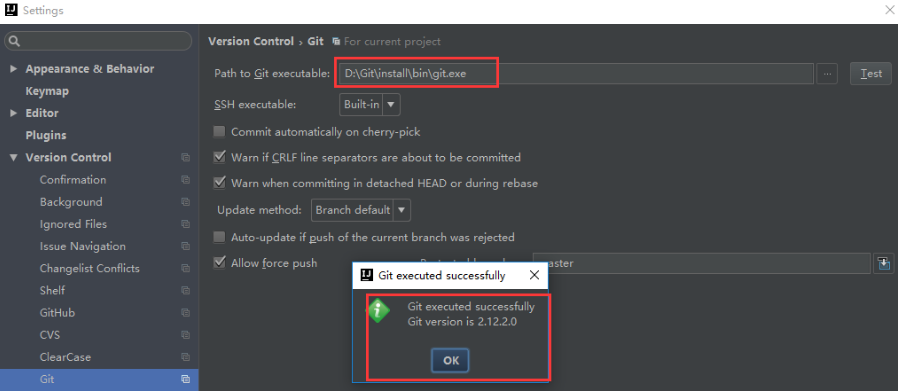
Micheal

## 5. IDEA配置Git

注册github或者码云，二选一，两个都是代码托管平台，一个是国外一个国内的。

#### 5.1idea设置git

Ctrl+Shift+s（File-->Setting）->Version Control-->Git-->Path to Git executable选择安装的git.exe文件，点击Test，测试设置成功如下：



#### 5.2idea中设置github

在官网上注册账号：<https://github.com/>

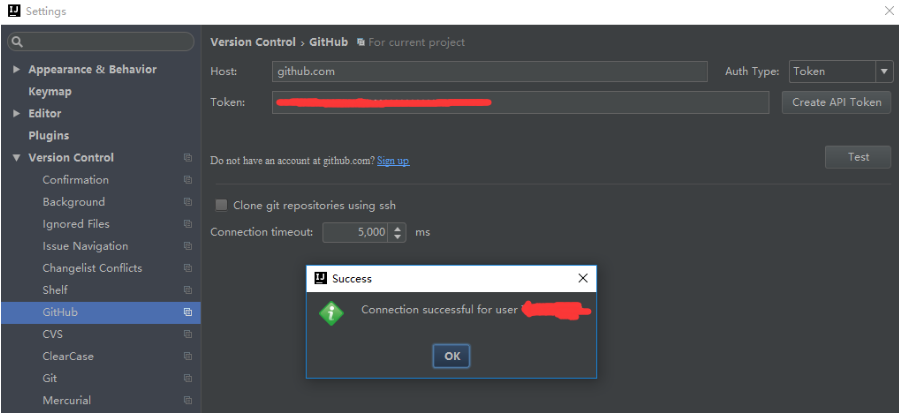
##### 5.2.1连接github

在IDEA中设置GitHub，File-->Setting->Version Control-->GitHub

　　Host：github.com

　　Token：点击Create API Token，输入在github中注册的用户名和密码生成token

　　点击Test，测试是否连接成功



##### 5.2.2 多人协作开发

参考

《git多人协作开发V1.0.0-徐玲博-2017-11-17》

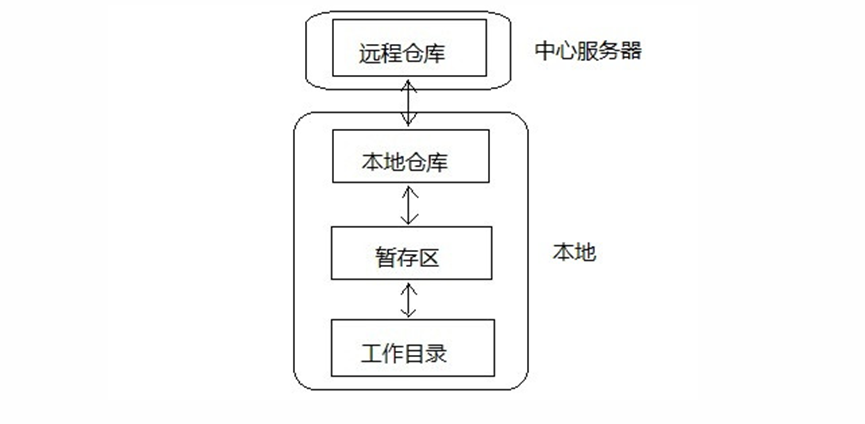
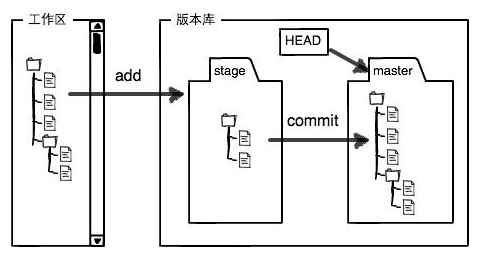
#### 5.3idea设置码云

在官网上注册账号：<https://gitee.com/>

和github步骤一样，大同小异

# 四、工作原理

## 1.git的分层结构



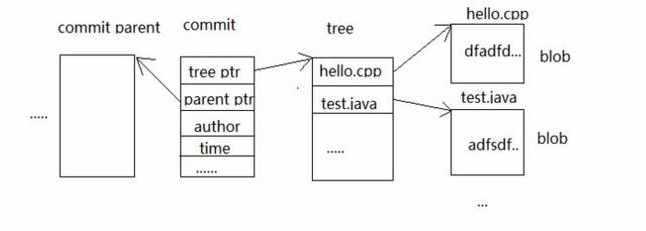
git的工作总共分四层，其中三层是在自己本地也就是说git仓库，包括了工作目录、暂存区和本地仓库。工作目录就是我们执行命令git init时所在的地方，也就是我们执行一切文件操作的地方；暂存区和本地仓库都是在.git目录下，因为它们只是用来存数据的。远程仓库在中心服务器，也就是我们做好工作之后推送到远程仓库，或者从远程仓库更新下来最新代码到本地。Git所存储的都是一系列的文件快照，然后git 来跟踪这些文件快照，发现哪个文件快照有变化他就会提示你需要添加到暂存区或是提交到本地仓库来保证你的工作目录是干净的。

这个要怎么理解呢？git中的文件有两种状态，一种是被跟踪的，也就是提交到本地仓库的文件，因为本地仓库要保管它们当然要跟踪他们，对他们负责，还有一种就是未被跟踪的。那么当我们添加新的文件时，他不是被跟踪的，因为本地仓库里面没有这个文件，他是外来的，本地仓库目前还不需要对他们负责。但是如果是对仓库已经存在的文件进行修改，那么这些文件就是被跟踪的文件，就可以通过git status查看他们的状态来进行相应的操作。当然我们也可以生成一个.gitignore文件，里面指定要忽略的文件类型，然后这些文件就不会被跟踪，不管怎么改变他们，git status都不会提示你需要做什么操作。

所以当我们在工作目录中进行文件操作后，要先添加到暂存区，然后再将暂存区中刚添加的文件快照提交到本地仓库，然后再将本地仓库的最新状态文件快照推送到远程仓库。这个文件快照其实就是各个文件在被添加到暂存区时的状态，就和照相的一样，留下每个不同时刻的快照，方便以后查询，而git存储的就是这些一系列的快照。说到这个快照就要说说git的对象了。

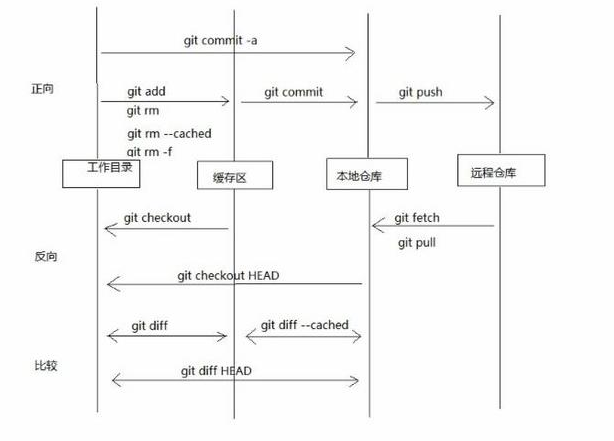
## 2.git的对象

从根本上讲，git是一套内容寻址的文件系统，它存储的也是key-value键值对，然后根据key值来查找value的，说到寻址就会想到指针吧，不错，git也是根据指针来寻址的，这些指针就存储在git的对象中。Git一共有三种对象，commit对象，tree对象和blob对象，下面便是这三个对象：



这个blob对象对应的就是文件快照中那些发生变化的文件内容，而tree对象则记录了文件快照中各个目录和文件的结构关系，它指向了被跟踪的快照，commit对象则记录了每次提交到本地仓库的文件快照，从上图可以看出其中有两个指针，一个指向tree对象，一个则指向上一个commit对象。这个怎么理解呢，怎么还有上一个commit对象，在开发过程中，我们会提交很多次文件快照，那么第一次提交的内容会用一个commit来记录，这个commit 没有指针指向上一个commit对象，因为没有上一个commit,他是第一个，当第二次提交时，又会有另外一个commit对象来记录，那么这次commit对象中就会有一个指针指向上一次提交后的commit对象，经过很多次提交后就会有很多的commit对象，它们组成了一个链表，当我们要恢复哪个版本的时候，只要找到这个commit对象就能恢复那个版本的文件。而我们所谓的HEAD对象其实就是指向最近一个提交的commit对象，也就是最后一个commit对象。

## 3.git的操作流程：



# 五、Git基本命令

1. **新建版本库**：git init
2. **将文件添加到版本库**：git add filename
3. **查看版本库状态**：git status
4. **提交**：git commit –m “tip information”，提交时除了目录和文件内容的实际变化，Git还会在每次提交的时候记录其他一些元数据，包括日志消息和做出本次变更的作者。一条完全限定的git commit命令必须提供日志消息和作者。
5. **查看提交**：一旦版本库里有了提交，就可以通过多种方式查看它们。Git log命令会产生版本库里一些列单独提交的历史。为了查看特定提交的更加详细的信息，可以使用git show命令带一个提交码，如 git show 9a581d。
6. **查看提交差异**：git diff
7. **版本库文件的删除**：git rm filename
8. **查看远程仓库有什么分支：**git branch -r
9. **版本库文件的重命名**：

$ mv foo.html bar.html

$ git rm foo.html

rm ‘foo.html’

$ git add bar.html

### 1.命令的简单应用

环境：安装有Windows Git的电脑

1. 在D盘根目录下建一个文件夹，命名为GitTest。
2. 在GitTest文件夹下右击鼠标打开Git Bash，并输入git init。
3. 这时候，GitTest文件夹下会出现一个.git的隐藏文件夹。这就说明我们建立了一个Git仓库。
4. 在GitTest目录下新建一个文本文件Project.txt，输入两行信息：Git is a version control system.
5. 未完待续（服务器好了之后弄）

# 六、分支管理

## 1．基本概念

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。有人把Git的分支模型称为“必杀技特性”，而正是因为它，将Git从版本控制系统家族中区分出来。

分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没有写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的风险。现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全又不影响别人工作。其他版本控制系统，如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。但Git的分支与众不同，无论创建、切换和删除分支，Git在1秒钟之内就能完成。无论你的版本是1个文件还是1万个文件。

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删除分支，这和直接在master分支上工作效果是一样的，但过程更安全。

## 2．基本命令

git branch branchname，创建分支。

git checkout branchname，切换分支。

git checkout –b branchname，创建新分支并立即切换到该分支。

当你切换到分支的时候，git会用该分支的最后提交的快照替换你的工作目录的内容，所以多个分支不需要多个目录。

git merge branchname，合并分支命令。你可以多次合并到统一分支，也可以选择在合并之后直接删除被并入的分支。

一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支，你可以使用以下命令将任何分支合并到你当前分支中去。

$ git branch

\*master

newtest

$ git merge newtest

git branch ,列出所有分支，前面有\*号的分支为当前分支。

$ git branch

\*master

此例的意思就是，我们有一个叫做master的分支，并且该分支是当前分支。当你执行git init的时候，默认情况下git就会为你创建master分支。如果我们要手动创建一个分支。执行git branch branchname即可。

$ git branch testing

$ git branch

\*master

Testing

现在我们可以看到，有了一个新分支testing。当你以此方式在上次提交更新之后创建了新分支，如果后来又有了更新提交，然后又切换到了“testing”分支，Git将还原你的工作目录到你创建分支时候的样子。

git branch –d branchname，删除分支命令。

## 3．合并冲突

合并不仅仅是简单的文件添加、移除的操作，Git也会合并修改。比如我们的版本库中有两个分支：master和testing。我们在testing分支下修改了test.txt文件并提交到testing分支上。现在切换回master分支，我们可以看到内容恢复到我们修改前的，我们再次修改test.txt文件并提交。然后合并的时候就会出现冲突的问题，因为两次修改不一致。这个问题就需要手动解决，并再次提交。

当git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

## 4．Bug分支

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。

当你接到一个任务，让你修复一个代号为101的bug，很自然的，你想创建一个复制issue-101来修复它。但是，等等，当前正在dev分支上进行的工作还没有提交。并不是你不想提交，而是工作进行到了一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？幸好，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。Bug分支和开发新功能的分支一样，你必须做完就销毁。

# 七、标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就惟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。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。

Git有commit，为什么还要引入tag呢？让我们来假设一个场景。

“请把上周一的那个版本打包发布，commit号是6a587hr”

“一串乱七八糟的数字不好找！”

换一个办法：“请把上周一的那个版本打包发布，版本号是v1.2”。

“好的，按照tag v1.2查找commit就行！”

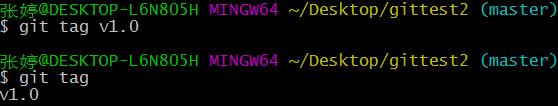
所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

## 1.创建标签

首先，切换到需要打标签的分支上，使用命令git branch和git checkout命令：



然后，敲命令git tag<name>就可以打一个新标签，然后可以用命令git tag查看所有标签：

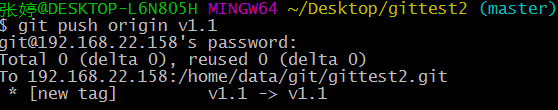


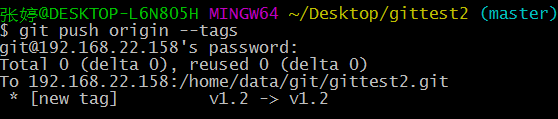
## 2.操作标签

如果标签打错了，也可以删除。



因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令 git push origin <tagname>

 或者，一次性推送全部尚未推送到远程的本地标签：

如果标签已经推送到远程，此时删除远程标签，先从本地删除，然后从远程删除，删除命令也是push，参考下图：

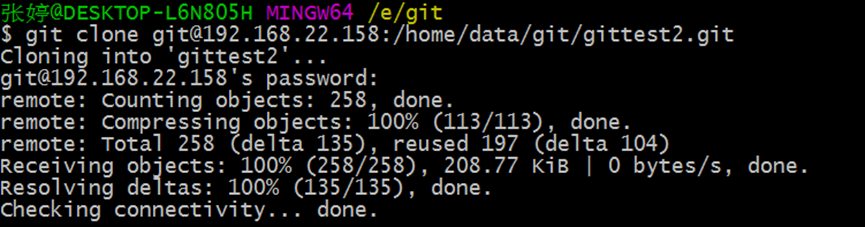


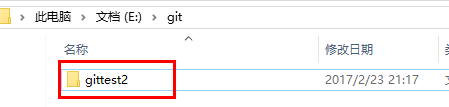
# 八、远程仓库管理

Git 是分布式版本控制系统。对于一个分布式节点来说，其它节点的 Git 仓库都可以作为本地仓库的远程仓库。

## 1.从远程库克隆

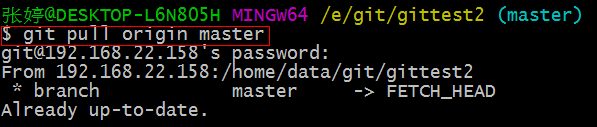
打开本地git客户端：git bush

 输入命令：git clone URL(URL支持git,ssh,http,https等各种协议) ，以git协议示例如下：

克隆过来的文件：

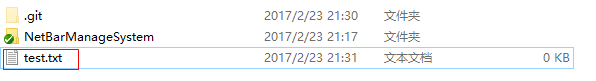
## 2.将修改后的文件推送到远端服务器

提交之前需要先拉取远端的文件，相当于svn的update操作，使客户端的文件状态与服务器端的最新状态保持一致：



注：Git默认使用origin 这个名字来标识原始的远程仓库。

当你在本地添加了一个文件，如下图：

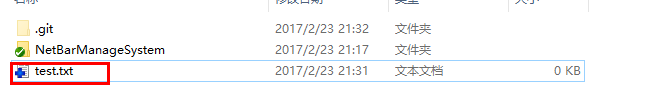


需要提交本地修改到远程仓库中，使用以下命令：

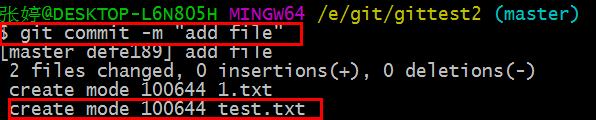
先用添加（add）命令 git add filename



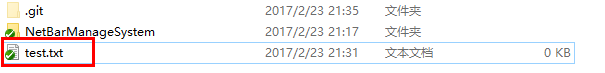
效果如图：



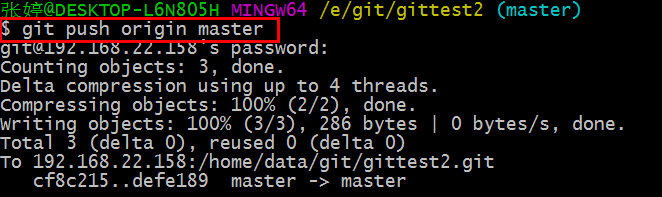
接着使用提交（commit）命令 git commit –m ”注释”



效果如图：



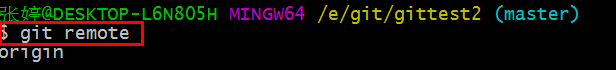
最后需要推送(push)到远程仓库中，使用命令 git push origin master



## 3.查看远程库

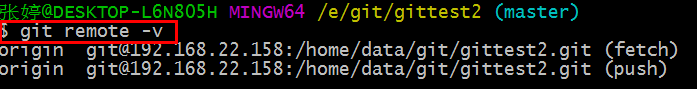
简单查看-所有仓库

git remote (只能查看远程仓库的名字)



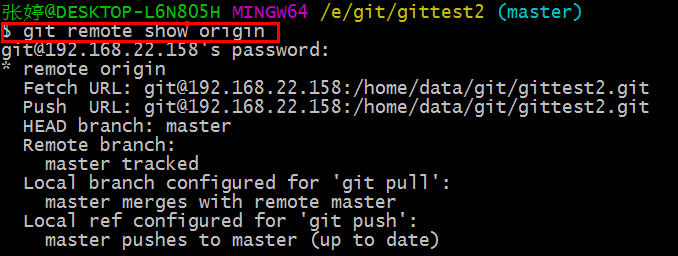
查看更多内容-所有仓库

git remote –v (远程仓库的名字及git地址)



查看单个仓库的信息

git remote show [remote-name]



## 4.新建远程仓库

git remote add [shortname] [url]

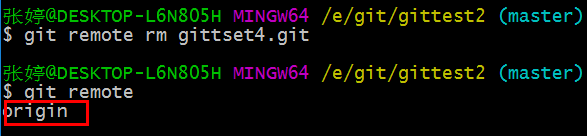


## 5.修改远程仓库

git remote rename [oldname] [newname]

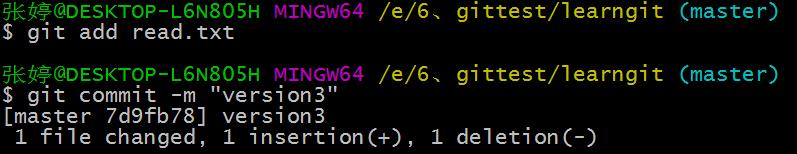


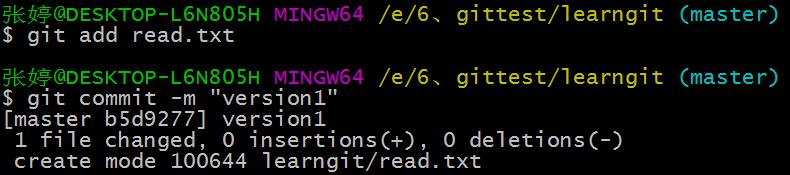
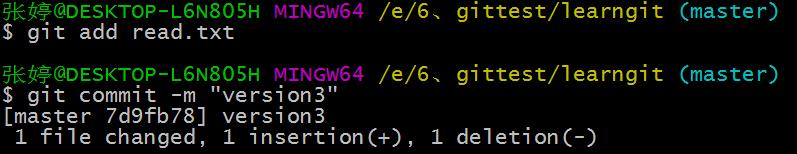
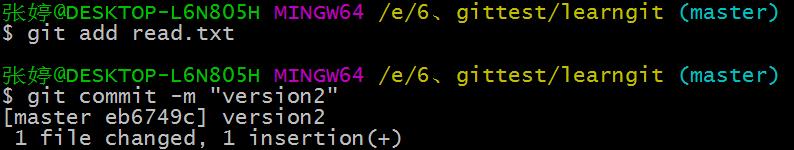
## 6.删除远程仓库

git remote rm [remote-name]

# 九、版本管理

## 1. 版本回退

 如图，我们对同一个read.txt 进行修改提交三次之后，



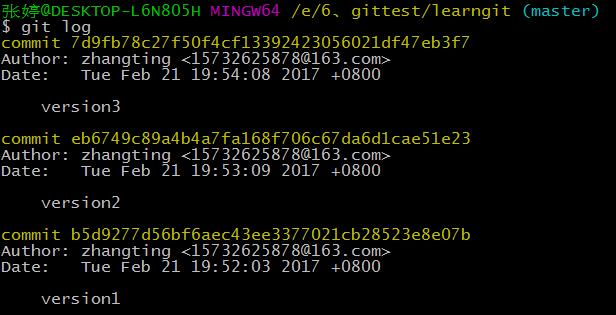
git仓库中便有了三个版本，分别是：

版本1：version1

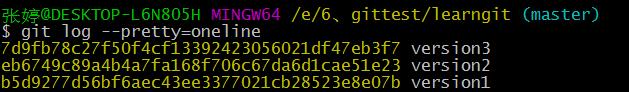
版本2：version2

版本3：version3

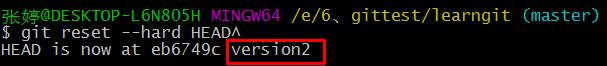
在实际工作中，我们想要查看每次都改了什么内容，有个命令可以告诉我们历史记录，那就是git log :



Git log 命令用于显示从最近到最远的提交日志，我们可以看到3次提交，最近的一次是version3 ,上一次是version 2 ,z最早的一次是version 1。如果嫌输出的信息太多，我们可以试试加上--pretty=oneline参数



注：这里的7d9fb8c…7eb3f7、eb6749c…ae51e23等由字母和数字组成的大长串是版本号（commit id），git这里的版本号和svn的不一样，因为这一长串字母和数字是一个SHA1(哈希算法)计算出来的一个非常大的数字，用十六进制表示。那你肯定会问了，为什么不用数字序列1，2，3……来表示呢？因为git是分布式的版本控制系统，很多的研究者在同一个版本库里工作，如果大家都用1,2,3……作为版本号，肯定就会发生冲突。

现在由于某种原因，你想要把read.txt会退到上一个版本，也就是“version2”的那个版本，你需要使用命令git reset:

进行版本回退时，git必须知道当前版本是哪个版本，在git中，用HEAD 表示当前版本，也就是最新的提交7d9fb8c…7eb3f7，上一个版本就是HEAD^，上上一个版本就是HEAD^^,当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

回退到上一版本version2之后，让我们看看现在版本库的状态（git log）：



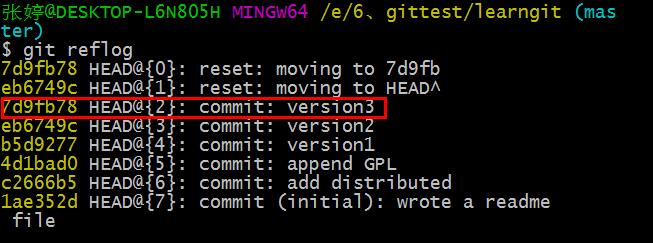
现在发现最新的那个版本version 3 不见了，好比你从现代21世纪穿越回唐朝，玩够了想回去却发现回不去了，怎么办呢？

别急，别急，办法来了只要上面的命令行窗口还没有被关掉，就可以顺着往上找，找到那个version3的版本号是7d9fb8c…7eb3f7 ，就可以指定回到未来的某个版本：



注：版本号没必要写全，前几位就可以了，git会自动去找，但是也不能只写一两位，因为git会找到多个版本号，就无法确定是哪一个了。

当然了，如果你把窗口关闭了，想恢复到新版本怎么办？ Git提供了一个命令git reflog 来记录你的每一次命令：



现在可以看到第三行显示version3的版本号7d9fb8c，现在可以回到21世纪了！

总结：Git log，查看提交详细信息；git log –pretty=oneline，查看简要信息。Git reset –hard 提交id，回退到特定的提交。回退之后，查找回退前的提交id，git reflog命令。

# 十、权限管理（参见gitlab操作文档）

# 十一、自定义Git

## 1.忽略特殊文件

有些时候，你必须把某些文件放到git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件。每次git status都会显示Untracked files….. ，有强迫症的同学肯定心里不爽。

好在git考虑到了大家的感受，这个问题解决起来也很简单，在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

不需要从头写.gitignore文件，github已经为我们准备了各种配置文件，只需要组合一下就可以使用了，所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>

忽略文件的原则是：1.忽略操作系统自动生成的文件，比如缩略图等；2.忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件。3.忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

## 2.配置别名

有没有经常敲错命令?比如git status？如果敲git st就表示git status那就简单多了。我们只需要敲一行命令：git config –global alias.st status，告诉git，以后st就表示status了。

## 3.配置文件

配置git的时候，加上—global是针对当前用户起作用的，如果不加，那只针对当前仓库起作用。配置文件放哪了？每个仓库的git配置文件都放在.git/config文件中。别名就在[alias]后面，要删除别名，直接把对应的行删除即可。而当前用户的git配置文件放在用户主目录下的一个隐藏文件.gitconfig中，配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

# 十二、名词解释

Repository，版本库。

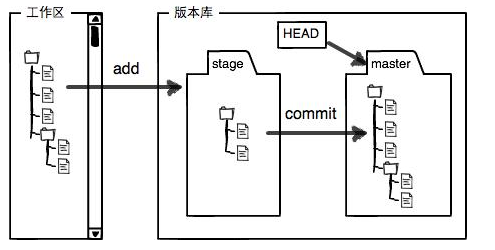
Init：initialize，初始化。

Distributed，分布式的。

在一个目录下，运行git init命令后，这个目录就变成了一个git仓库。把文件添加到仓库分两步：git add ；git commit 。

工作区，就是存在.git文件的文件夹。

版本库，工作区有一个隐藏的.git文件，这个不算工作区内容，而是Git的版本库。Git的版本库中存了很多东西，其中最重要的是称为stage（或者index）的暂存区，还有git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



前面我们讲了把文件往git版本库里添加的时候，是分两步执行的。第一步用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。为什么Git比其他版本控制工具设计的优秀？因为Git跟踪并管理的是修改，而非文件。你会问，什么是修改？比如你增加一行，这就是一个修改；删除了一行，也是一个修改；更改了某些字符，也是修改；删除了一些又加了一些，也是一个修改，甚至创建一个新文件，也算是一个修改。

Tracks，跟踪。

Checkout，检查。

Git checkout –readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销。

Remote，远程的；远程操作，遥控器；向远方延伸

Origin，起源。

Conflict，冲突。

Graph，曲线图。

Pretty，漂亮的。

Abbrev，缩写。

Hint，线索，提示。

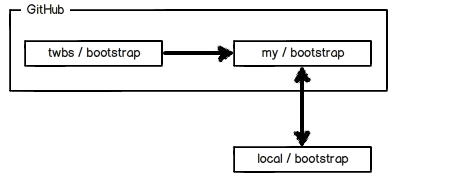
Upstream，向上。

Fatal，致命的。

Fork，分叉，叉。

# 十三、GitHub简介

我们一直用github作为免费的远程仓库，如果是个人的开源项目，放到github上是完全没有问题的。其实github还是一个开源协作社区，通过github，既可以让别人参与你的开源项目，也可以参与别人带的开源项目。如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页：<https://github.com/twbs/bootstrap>，点Fork就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone。一定要从自己的账号下clone仓库，这样你才能推送修改。如果从bootstrap的作者的仓库地址 [git@github.com:twbs/bootstrap.git](mailto:git@github.com:twbs/bootstrap.git) 克隆，因为没有权限，你将不能推送修改。Bootstap的官方仓库、你在github上克隆的仓库，以及你自己克隆到本电脑的仓库，它们的关系就像下图显示的那样:



如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。如果你希望bootstrap的官方库能接受你的修改，你就可以在github上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

在github上，可以任意fork开源仓库。自己拥有fork后的仓库的读写权限，可以推送pull request给官方仓库来贡献代码。