[TortoiseGit用户手册 1](#_Toc353801205)

[说明 1](#_Toc353801206)

[1 安装MsysGit 1](#_Toc353801207)

[2 安装TortoiseGit 6](#_Toc353801208)

[3 配置TortoiseGit 6](#_Toc353801209)

[3.1 生成公钥 6](#_Toc353801210)

[3.2 配置用户（重要） 7](#_Toc353801211)

[3.3 配置msysgit 7](#_Toc353801212)

[4 TortoiseGit的使用 8](#_Toc353801213)

[4.1 克隆远程的版本库 8](#_Toc353801214)

[4.2 使用TortoiseGit本地建库 10](#_Toc353801215)

[4.3 向仓库中添加内容 10](#_Toc353801216)

[4.4 推到服务器 13](#_Toc353801217)

[4.5 更新版本库 15](#_Toc353801218)

[4.5.1 git pull 15](#_Toc353801219)

[4.5.2 git fetch 16](#_Toc353801220)

[4.6 分支 16](#_Toc353801221)

[4.6.1 创建分支 17](#_Toc353801222)

[4.6.2 分支切换 17](#_Toc353801223)

[4.6.3 分支合并 18](#_Toc353801224)

[4.7 版本差异 18](#_Toc353801225)

[4.7.1 两个版本的差异 18](#_Toc353801226)

[4.7.2 查看未提交的修改 19](#_Toc353801227)

[4.8 撤销某次操作 20](#_Toc353801228)

[4.9 Git Resolve 20](#_Toc353801229)

[5 Git中冲突的解决 21](#_Toc353801230)

[5.1 逻辑冲突 21](#_Toc353801231)

[5.2 树冲突 22](#_Toc353801232)

[5.3 内容冲突 23](#_Toc353801233)

[6 Git分支管理策略 25](#_Toc353801234)

[6.1 Master分支（主分支） 26](#_Toc353801235)

[6.2开发分支Develop 26](#_Toc353801236)

[6.3 临时性分支 26](#_Toc353801237)

[功能分支 27](#_Toc353801238)

[预发布分支 27](#_Toc353801239)

[修补Bug分支 27](#_Toc353801240)

# TortoiseGit用户手册

## 说明

我们可以把CVS、SVN拿来和GIT比较但是不要拿它们来思考GIT

本文档只是日常工作中GIT的一些使用的简单归纳，旨在帮助大家快速用上GIT，先能够用起来，知其然后再去究其所以然。本文档主要是关于Git图形界面客户端（TortoiseGit）的使用，但是安装TortoiseGit前必须安装windows下的Git版本——MsysGit。

注意：本人安装过其他版本的msysgit和tortoisegit，安装界面略有不同。

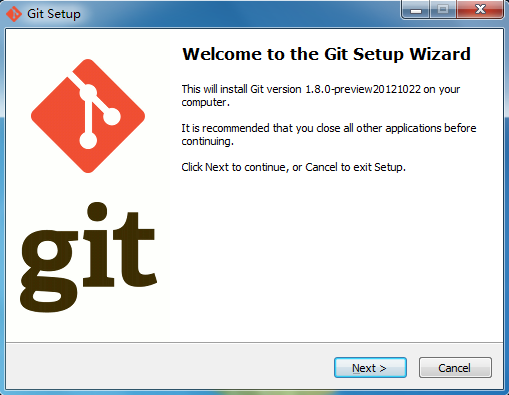
## 1 安装MsysGit

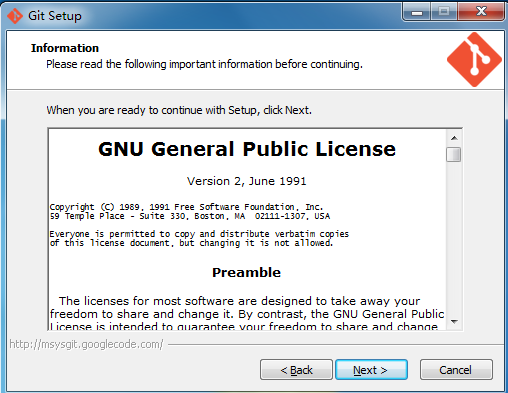
Msysgit是 [Git](http://baike.baidu.com/view/1531489.htm) 版本控制系统Windows下的版本。msysGit 有个简单的 GUI 工具，及简单的 Explorer 集成；但它自带的Bash（命令行式）非常好用，深得Linux 的真传。

**下载地址**（Git-1.8.0-preview20121022.exe）：

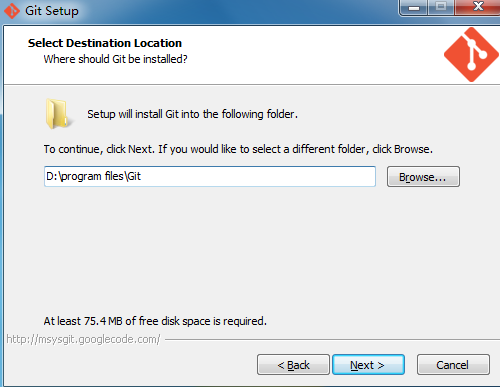
<http://code.google.com/p/msysgit/downloads/list>

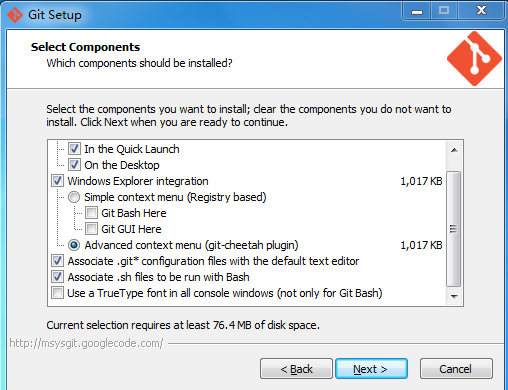
**开始安装：**

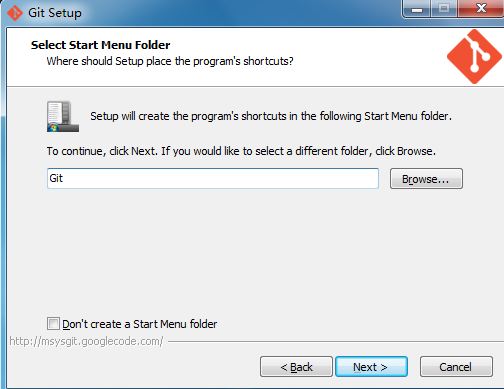




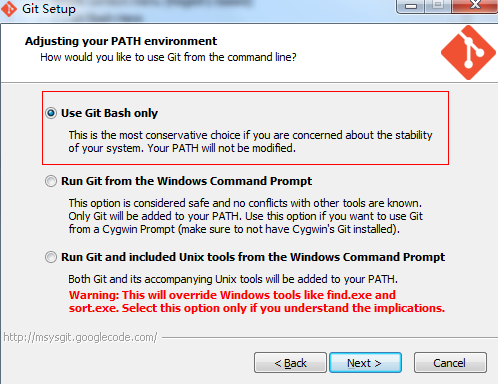
**选择安装目录：**



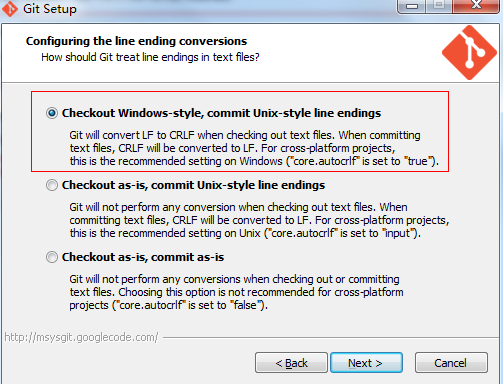




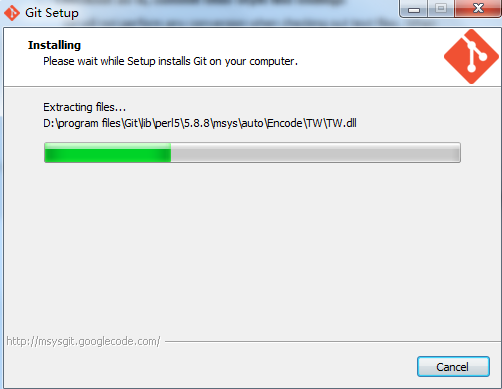
**设置如何使用GIT命令行，按默认设置，使用软件自带的BASH风格的命令行即可：**



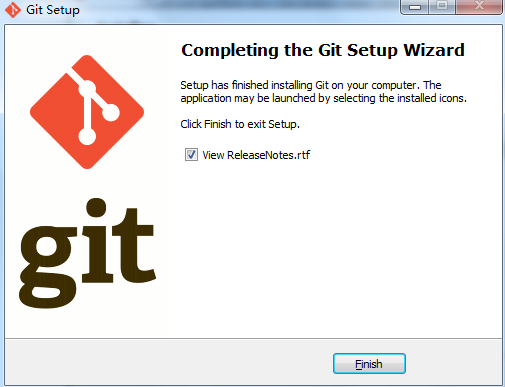
**配置版本控制中文本文件的换行风格，按系统默认设置即可（即从版本库中取出时使用Windows系统的“回车+换行”风格，存入版本库时使用Unix系统的“换行”风格）**



**正在安装：**



**安装完成：**



## 2 安装TortoiseGit

TortoiseGit 只是 GUI 工具，使用它需要先安装 MsysGit，这是正宗的 Git 之 Windows 版本。其用法类似于TortoiseSVN。

**下载地址**（TortoiseGit-1.8.0.0-32bit）：

<http://code.google.com/p/tortoisegit/downloads/list>

**开始安装：**

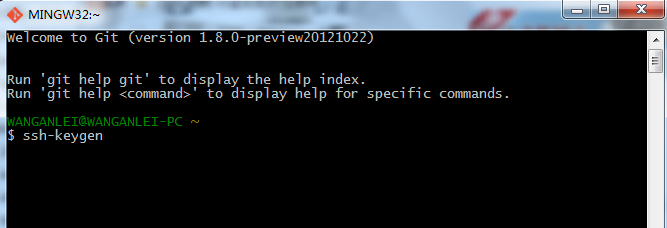
**一路点击“next”即可。**

## 3 配置TortoiseGit

### 3.1 生成公钥

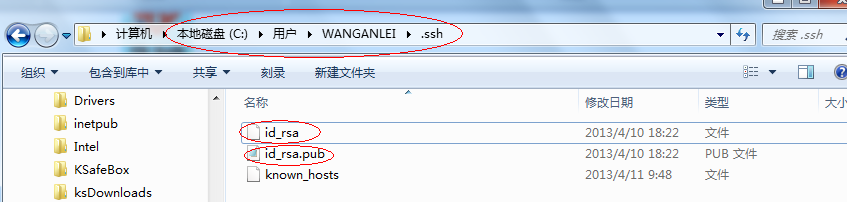
生成SSH安全密钥，提供给GIT版本库管理员以访问Git 版本库，点击桌面上生成的图标

然后执行执行“ssh-keygen”生成自己的公钥：



一路回车即可完成操作，这么操作的话就是没有给自己的私钥设置密码，也可以在执行过程中设置私钥密码的。

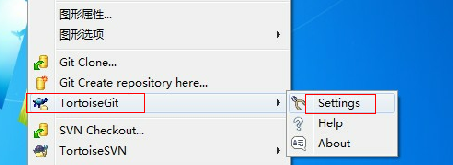
生成的公钥和私钥的目录是：



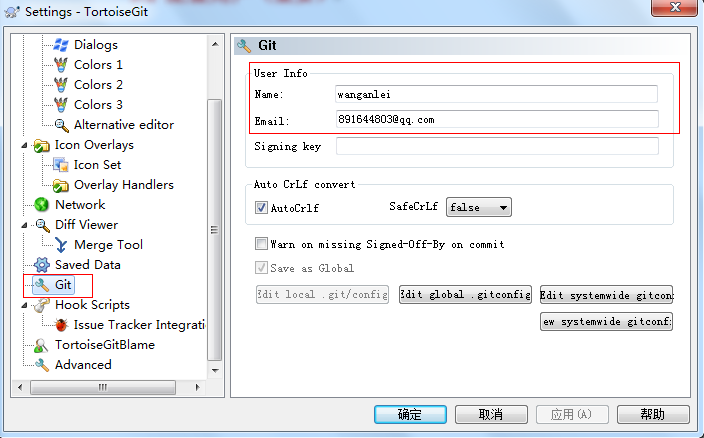
看见没，id\_rsa就是私钥，id\_rsa.pub就是公钥，将id\_rsa.pub 复制一份，然后重命名，发送给GIT版本库的管理员即可。

### 3.2 配置用户（重要）

在桌面的空白处右键——TortoiseGit——setings：

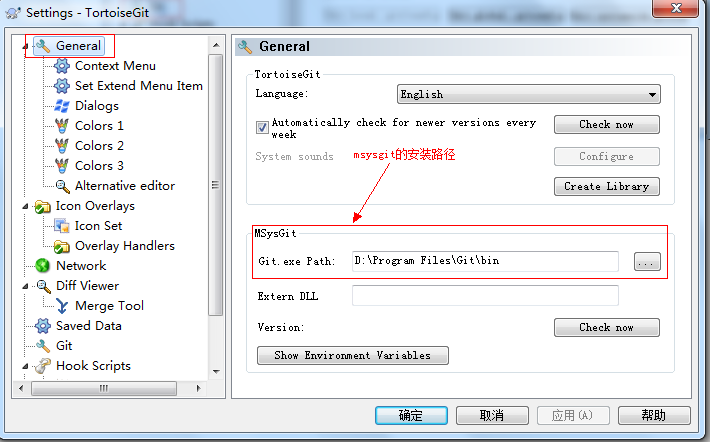


选择“Git”，然后在“Name”和“Email”中填入相关用户信息即可：



### 3.3 配置msysgit

这一步就是为什么要安装msysgit了，TortoiseGit是在它上运行的。完成了msysgit的配置我们就可以开始使用TortoiseGit进行clone、pull、create branch、push等操作了。



## 4 TortoiseGit的使用

本例中GIT版本库相关信息如下：

GIT版本库地址：“10.17.15.162”

操作GIT版本库的用户名：root

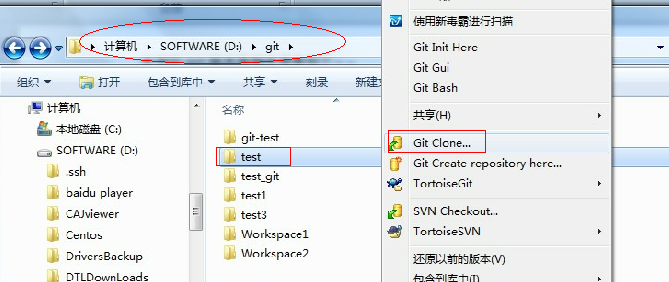
测试项目的名称：project1

本地测试目录：D：\git\test

### 4.1 克隆远程的版本库

例：服务器的project1仓库clone到D:\git\test：

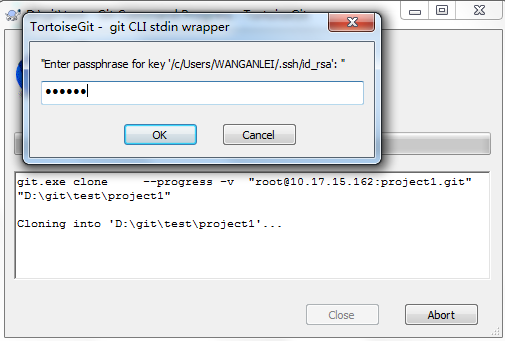
在test目录上右键选择“Git clone”：



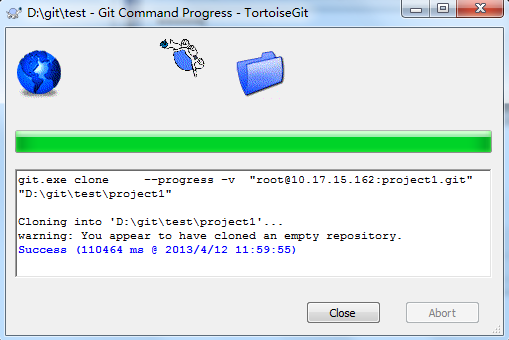
然后URL输入：[root@10.17.15.162:project1.git](mailto:root@10.17.15.162:project1.git)，选择“Web”，会自动生成存放仓库的目录：

D:\git\test\project1；

最后“OK”开始clone（如果的你的私钥设置了密码还有下面的提示，输入密码即可）：



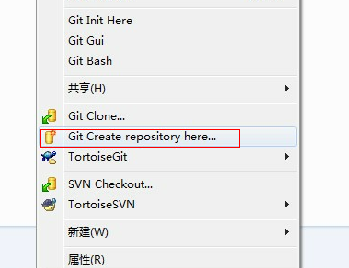
然后就看见了success的提示窗口：



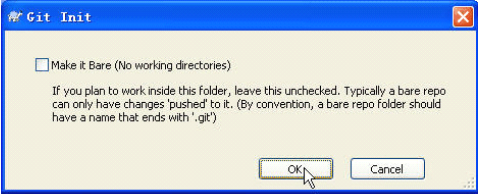
这样我们就完成了一次远程仓库的clone，因为我的远程仓库是新建的里面是空的，所以给了个警告说是我克隆了一个空仓库，没关系的。（看下你的D:\git\test\project1里面是不是有个.git的目录？如果没有就是隐藏了）

### 4.2 使用TortoiseGit本地建库

选中某文件夹按右键,选择Git Create repository here就可以创建库了：



在出现的窗口中,不勾选选项,直接按OK：



在目录中就会出现一个名为.git的隐藏文件夹,所有库的相关内容都会存在这个文件夹中.以后不管这个项目添加多少个文件夹,整个库只会有这一个管理文件夹,这和CVS和SVN有较大差异。

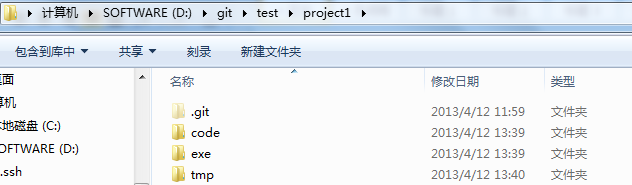
提示：这里所建的仓库只是用说明Git的一个本地建库的功能，与前面和后面的内容是相互独立的，所以大家别误解这一步到底是做什么的。

### 4.3 向仓库中添加内容

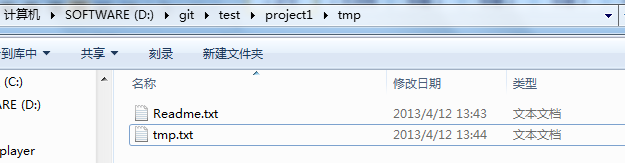
这一步是在本机完成的，完全可以不需要网络脱离服务器的。

4.1那一步我们clone了一个空的版本库，现在为这个库添加目录和文件。

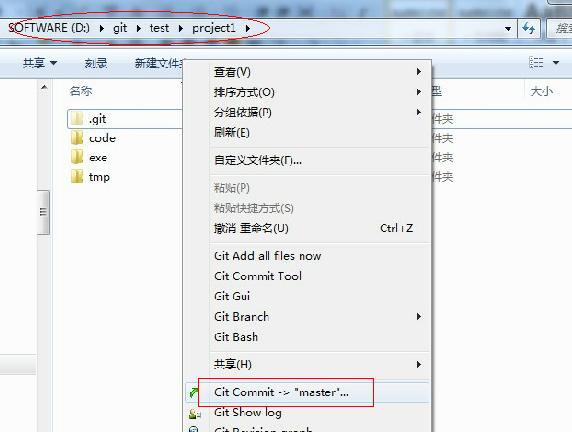
先给D:\git\test\project1创建三个目录：



然后在tmp添加几个文件：

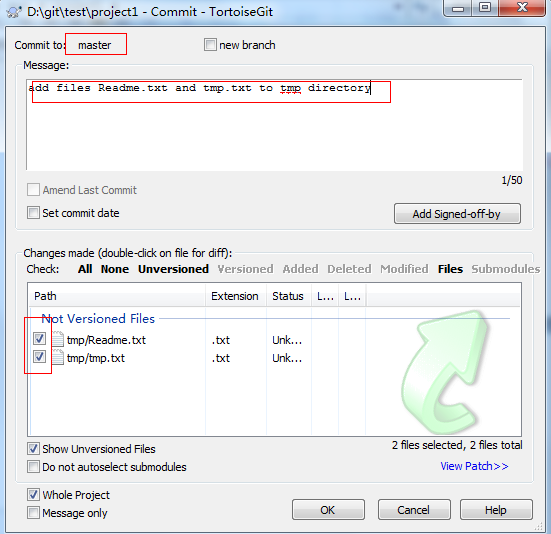


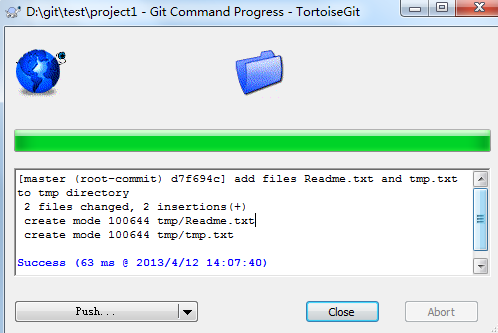
这一步我们就需要把这些文件和目录commit到本地的版本库project1中生成版本快照，在D:\git\test\project1目录的空白处右键选择：Git commit->“master”



注意：我们目前只有一个分支所以只能commit到master分支，当我们创建了其他分支以后我们可以commit到别的分支。

然后把新建文件前面的钩给打上，录入相关操作说明点击“OK”：



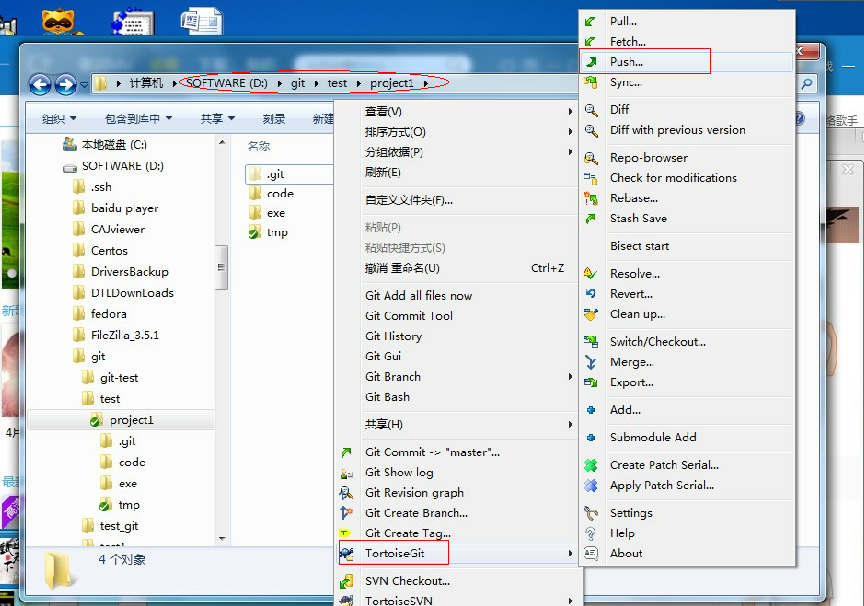


注意：这一步的操作只是commit了tmp目录和它下面的那两个文件，其他的两个空目录并没有commit进来，要一起commit的话必须给那两个目录添加文件。

### 4.4 推到服务器

从远程的clone到本地的commit，那么现在就可以把本地的版本库push到服务器了。

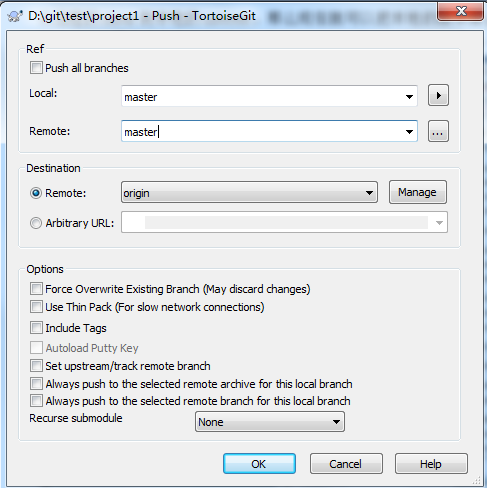
在project1目录的空白处右键——》“TortoiseGit”——》“Push”：

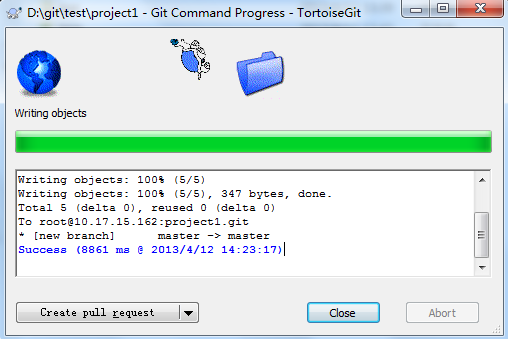


看下图，可以选择要Push的分支和要Push到服务器的哪个分支，这些都是根据实际的需要来定的，这里它会默认的推送到它clone的服务器（10.17.15.162）的project1仓库的。

在Push时还会弹出输入私钥密码的对话框，输入即可：

提醒：master是Git默认的主要分支（主干）,适合单人独自开发，若是多人开发强烈建议每人创建自己的分支。

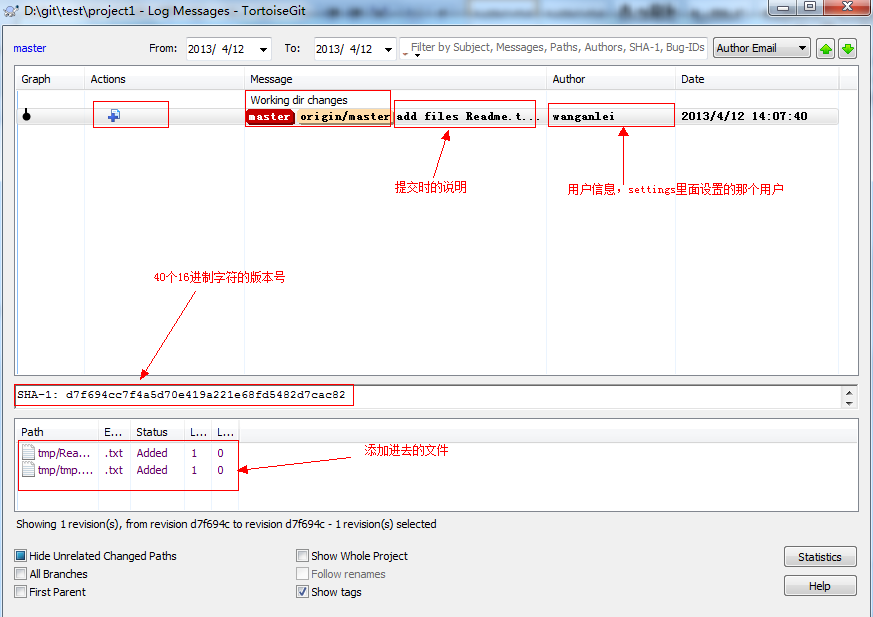




推送成功！

看下log吧，右键选择“Git show log“：

这个日志是随时都能查看的，能看见版本号、提交的文件、时间、由谁提交的等信息。

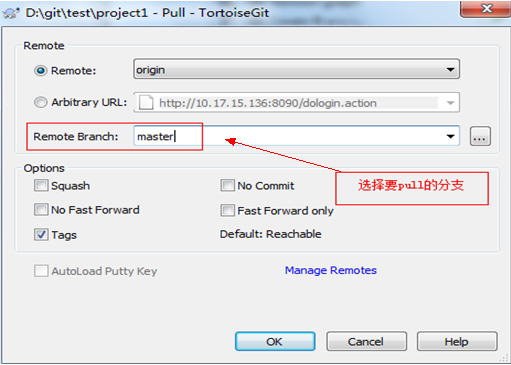


### 4.5 更新版本库

#### 4.5.1 git pull

前一天服务器的版本库经过若人的push以后现在是最新的了，那么今天的准备工作就是把最近的版本库合并到本机，我们采用的是pull命令。

更新操作：在D:\git\test\project1目录下，选中project1仓库右键，然后选择“TortoiseGit“——》”Pull“：



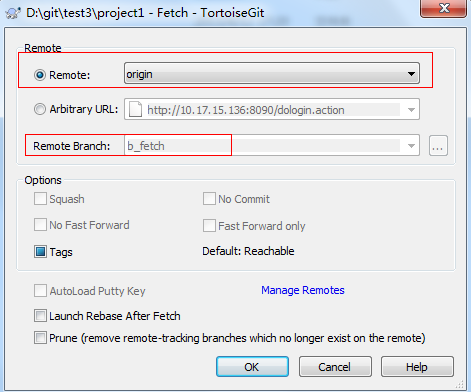
点击“OK“就能把服务器的版本更新到本地的master分支，同步的结果是我发现仓库里面多了一个文件夹和文件。然后新建一个分支在本分支开始今天工作。

#### 4.5.2 git fetch

Git Pull能实现本机和服务器的同步，但是它不够安全，还有一种更加安全的做法就是git fetch，我的做法是创建一个新的分支master/b\_fetch，先把代码fetch到master/b\_fetch分支，然后比较master/b\_fetch与master分支的区别（就是两个版本库的区别），最后进行merge操作。这样具有安全性！分支的创建和比较差异后面介绍！

Git Pull=git Fetch+git merger

演示一下fetch吧！切换到b\_fetch分支，然后右键选择“TortoiseGit“——》”fetch“：



### 4.6 分支

**Git分支简介：**

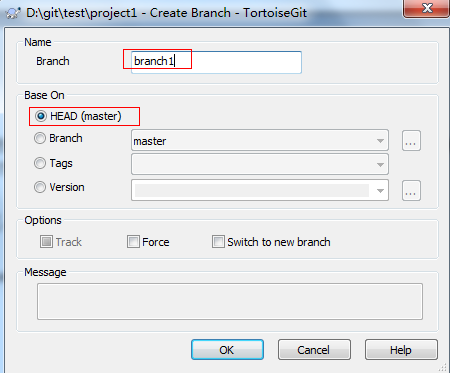
Git 的分支可谓是难以置信的轻量级，它的新建操作几乎可以在瞬间完成，并且在不同分支间切换起来也差不多一样快。和许多其他版本控制系统不同，Git 鼓励在工作流程中频繁使用分支与合并，哪怕一天之内进行许多次都没有关系。

Git 中的分支实际上仅是一个包含所指对象校验和（40 个字符长度 SHA-1 字串）的文件，所以创建和销毁一个分支就变得非常廉价。说白了，新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）那么简单，当然也就很快了。这和大多数版本控制系统形成了鲜明对比，它们管理分支大多采取备份所有项目文件到特定目录的方式，所以根据项目文件数量和大小不同，可能花费的时间也会有相当大的差别，快则几秒，慢则数分钟。而 Git 的实现与项目复杂度无关，它永远可以在几毫秒的时间内完成分支的创建和切换。同时，因为每次提交时都记录了祖先信息（译注：即 parent 对象），所以以后要合并分支时，寻找恰当的合并基础（译注：即共同祖先）的工作其实已经完成了一大半，实现起来非常容易。Git 鼓励开发者频繁使用分支，正是因为有着这些特性

作保障。

#### 4.6.1 创建分支

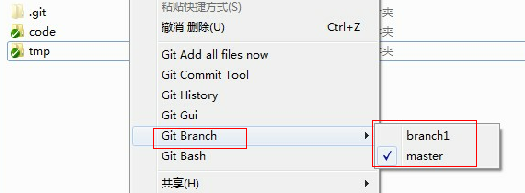
右键选择“Git Create Branch“：



这里是我们创建的第一条分支，所以只能创建在master分支上面了。

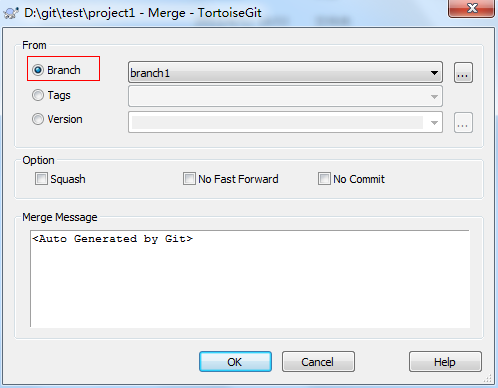
#### 4.6.2 分支切换

右键选择“Git Branch“就能看见所有的分支，显示的当前所在分支是master（注意前面的对勾），branch1是新建分支，这样我们就能随意的切换分支。



#### 4.6.3 分支合并

在branch1上的开发完成以后我们需要合并到master分支，首先切回master分支，然后右键选择“TortoiseGit“——》”merge“：



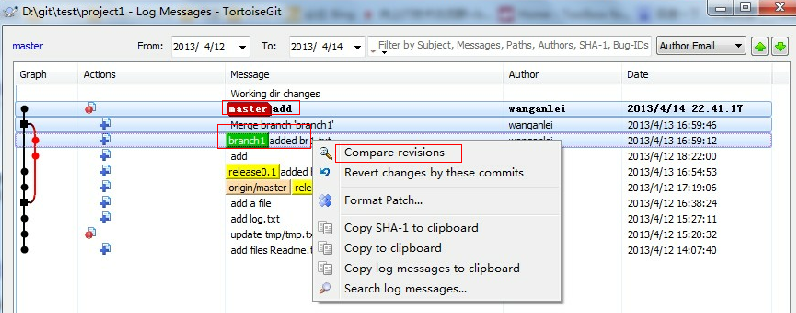
当前分支是master，所以得选择branch1，就是把branch1合并到master分支上。

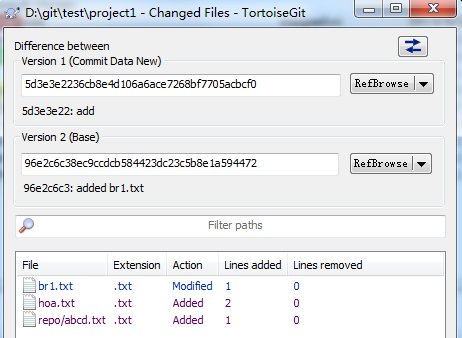
### 4.7 版本差异

#### 4.7.1 两个版本的差异

比如我现在想看一下我master分支和branch1这两个版本的差异，该怎么看呢？

首先我们在“Git Show log”下看到所有的提交日志，然后按住shift用鼠标选择最新的master分支和branch1分支，右键选择“Compare revisions”就能看见这两个版本的差异：



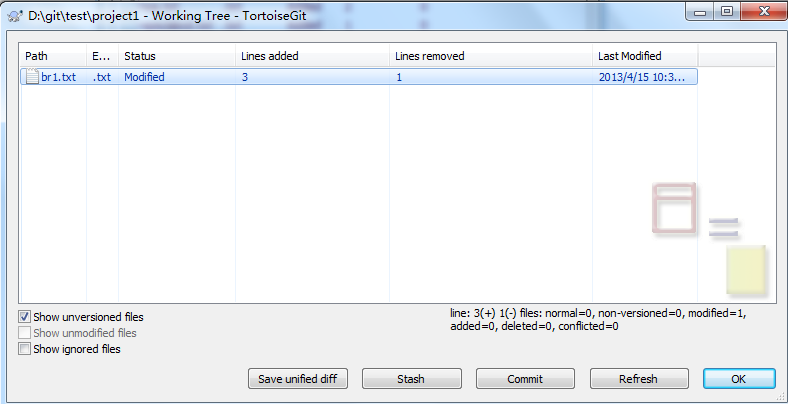


由上图我们发现master分支相对branch1分支修改了br1.txt，增加了hoa.txt和repo文件夹以及repo下面的abcd.txt文件。

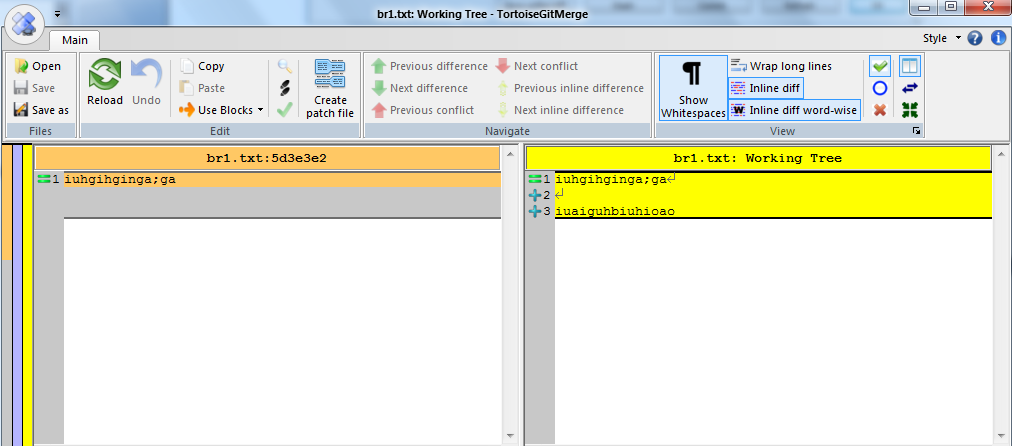
#### 4.7.2 查看未提交的修改

当你修改了一些文件后，这时还没有提交，你可能意识到自己的某些修改是错误的，那么你就想再次的审查一遍，那么怎么看你修改了哪些文件的哪些位置呢？

首先右键选择“TortoiseGit”，然后选择diff：

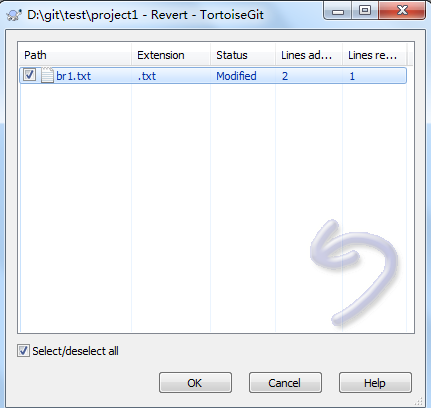


这时就能看见修改过的文件，如果想看修改的具体位置可以双击文件继续查看详细内容：



### 4.8 撤销某次操作

revert 是撤销某次操作，此次操作之前的commit都会被保留，比如我们发现自己修改的错误后就可以撤销，右键选择“TortoiseGit”，然后选择“Revert”：



选中要撤销的文件，点击“OK”就能实现撤销操作。

### 4.9 Git Resolve

这个命令主要是在解决冲突时才用的，具体用法看5.3中内容冲突的解决。

## 5 Git中冲突的解决

Git中很多命令都可能出现冲突，但从根本上来讲，都是merge 和 patch（应用补丁）时产生冲突。而rebase就是重新设置基准，然后应用补丁的过程，所以也会冲突。一般产生冲突的类型有逻辑冲突、内容冲突、树冲突。

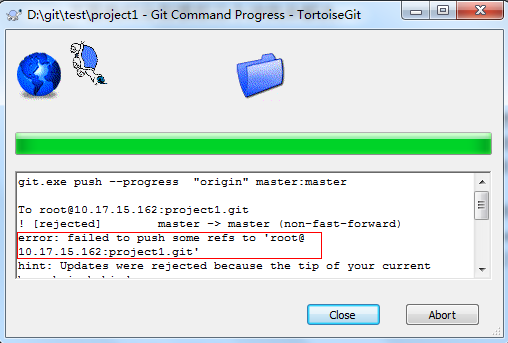
团队协作中冲突是不可避免的，应该尽最大努力的避免冲突发生，比如模块的分工化，然后就是每个开发人员遵守一定的的规则，必要的沟通是解决冲突最有效最好的方法！

下面我们分别模拟三个冲突并解决。

### 5.1 逻辑冲突

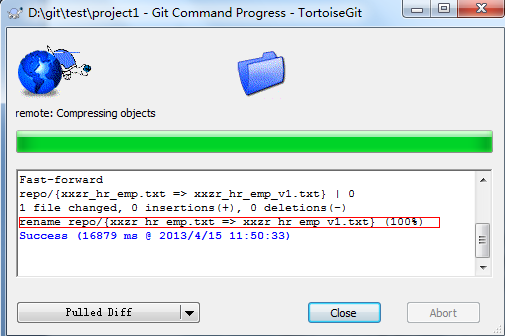
什么事逻辑冲突？Git自动处理（合并、应用补丁）成功，但是逻辑上是有问题的。比如另外一个人修改了文件名，但我还使用老的文件名，这种情况下自动处理是能成功的，但实际上是有问题的。又比如，函数返回值含义变化，但我还使用老的含义，这种情况自动处理成功，但可能隐藏着重大BUG。我们做个实际的模拟：

一个同事在没有告诉我的情况下把我们共同使用的一个文件xxzr\_hr\_emp.txt修改成xxzr\_hr\_emp\_v1.txt，那么他已经把xxzr\_hr\_emp\_v1.txt新文件commit到服务器了，而我依旧使用xxzr\_hr\_emp.txt文件名，那么在我工作完commit时会发生什么呢？



在我commit时就报错了，说是提交失败，这时失败的原因就是冲突导致的，看来解决的办法在服务器是不行了，只能在本机了，这时我就需要做一次Pull操作，把远程库拉下来，合并完解决冲突后再commit上去。

拉到本机：



这样逻辑冲突自己解决了，他会把我以前的xxzr\_hr\_emp.txt文件名更新为xxzr\_hr\_emp\_v1,然后这是我就可以正常的commit了。

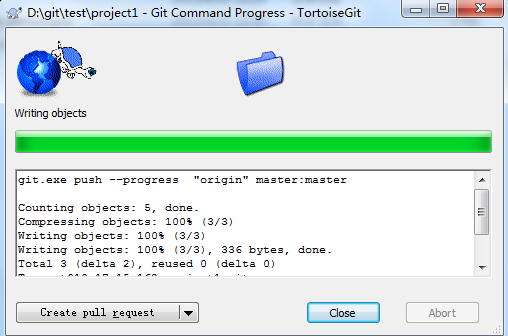
### 5.2 树冲突

文件名修改造成的冲突，称为树冲突。

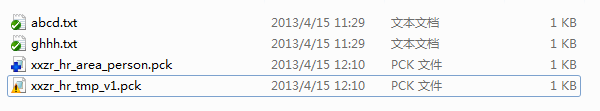
比如，a用户把文件改名为a.c，b用户把同一个文件改名为b.c，那么b将这两个commit合并时，会产生冲突。模拟一下吧：

开始我和另一个同事的版本库是一样的我们都有一个xxzr\_hr\_v1.pck的文件，但是我们两个没有互相沟通都是把这个文件名给改了，我改成xxzr\_hr\_area\_person.pck,他改成xxzr\_hr\_tmp\_v1.pck，我一个一个人改了以后是可以正常push的，那么在我完成后他再push毫无疑问他push时遇到麻烦了，遇到的错误和逻辑错误的报错是一样的，这时看一下该怎么解决：

首先和前面一样的他做一次Pull操作，此时没有显示任何的异常：



但是到他的工作目录一看，就不一样了：



他就发现他重命名的文件名上多了一个叹号，又在该目录下面多了一个新文件，这样的原因在于两人同时修改了同意文件名，这时他发现了错误的原因，就找到我和我沟通到底该用谁的文件名，最终我们通过沟通决定使用我的xxze\_hr\_area\_personp.pck文件名（我们的需求是关于地域人员分布的统计），把他的那个有叹号的文件直接删除再做一次提交就好了。

要是我们在修改文件名时做一次沟通是不是冲突直接就避免了？所以说团队协

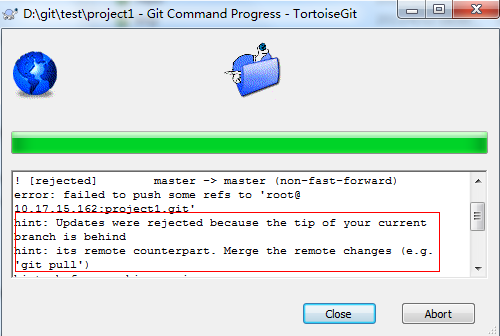
作中沟通真的很重要！

### 5.3 内容冲突

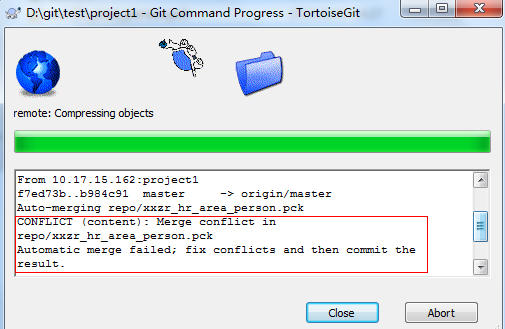
内容冲突时最常见的，也是一种不可避免的冲突现象。两个用户修改了同一个文件的同一块区域，git会报告内容冲突。

模拟并解决：

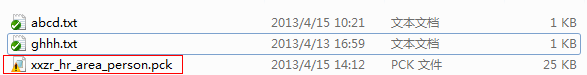
我和另一同事对同一文件的同一块做了修改，那么他提交完毕我提交时就会产生冲突，我们修改的文件是xxzr\_hr\_area\_person.pck，先看一下报的错误吧：



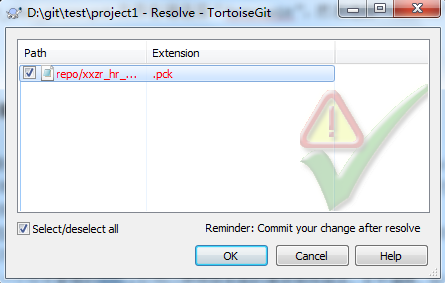
看见错误没提示我要执行一次pull操作，那么我就执行以下看看结果是什么：



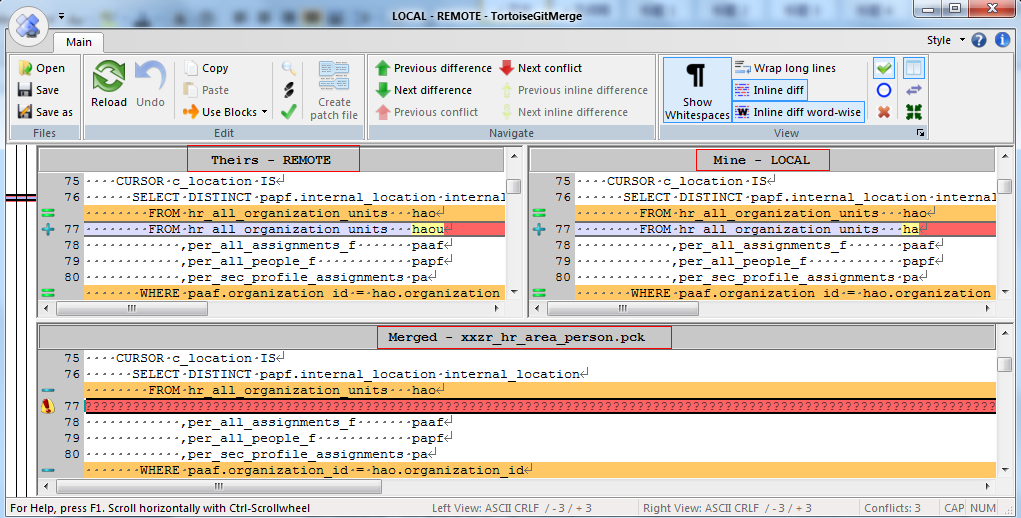
提示了冲突在repo/xxzr\_hr\_area\_person.pck文件这，解决了冲突再commit结果。



发现xxzr\_hr\_area\_person.pck文件上面有个黄色的叹号，这就表示文件有冲突，现在开始解决冲突，选中xxzr\_hr\_area\_person.pck文件右键选择“TortoiseGit”，然后选择“Resolve”开始解决冲突：



这是双击一下那个xxzr\_hr\_area\_person.pck文件会看到详细的改动情况：



左上部分是别人改动的结果，右上部分是我改动的，下面是需要我手工做的最终的合并情况，这里就是我要把改动的结果写到带有红色问号那行就行了，合并完成后保存退出，然后先提交到本机再push操作，这样内容冲突的解决完毕。

提示：如果这个文件是大家共同的，那么请千万记住当你每次有一个小的改动时，立即push到服务器，这样的话冲突解决起来也简单，快捷。

## 6 Git分支管理策略

相比同类软件，Git有很多优点。其中很显著的一点，就是版本的分支（branch）和合并（merge）十分方便。有些传统的版本管理软件，分支操作实际上会生成一份现有代码的物理拷贝，而Git只生成一个指向当前版本（又称”快照”）的指针，因此非常快捷易用。

但是，太方便了也会产生副作用。如果你不加注意，很可能会留下一个枝节蔓生、四处开放的版本库，到处都是分支，完全看不出主干发展的脉络。

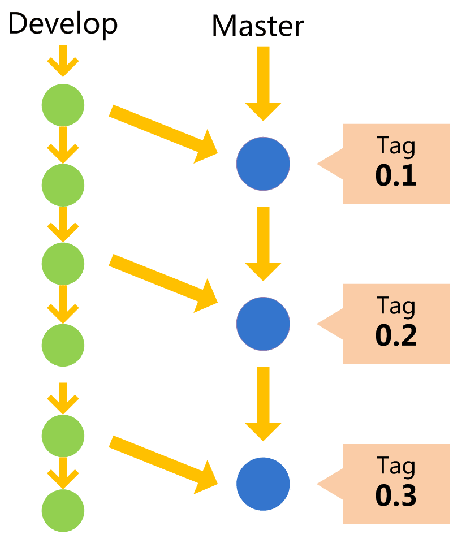
[Vincent Driessen](http://nvie.com/)提出了一个分支管理的[策略](http://nvie.com/posts/a-successful-git-branching-model/)，它可以使得版本库的演进保持简洁，主干清晰，各个分支各司其职、井井有条。理论上，这些策略对所有的版本管理系统都适用，Git只是用来举例而已。

### 6.1 Master分支（主分支）

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。我们可以给主分支上的每个版本库打上tag，为了更好的管理。

### 6.2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。



如果想正式对外发布，就在Master分支上，对Develop分支进行”合并”（merge）。

### 6.3 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。

#### 功能分支

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。

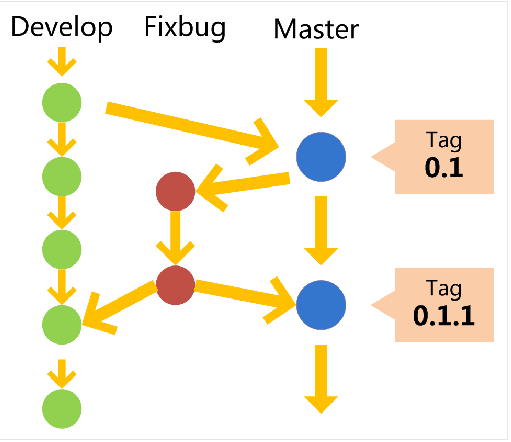
#### 预发布分支

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。

#### 修补Bug分支

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。



合并完成以后的修补Bug分支要删除的。