**GIT使用指南**

**目录**

[1、GIT介绍 4](#_Toc514922511)

[1.1、分布式仓储 4](#_Toc514922512)

[1.2、GIT快照（Snapshot） 5](#_Toc514922513)

[1.2.1 SVN记录差异 5](#_Toc514922514)

[1.2.2 GIT记录快照 5](#_Toc514922515)

[1.3、GIT存储压缩 7](#_Toc514922516)

[1.4、分支特性 7](#_Toc514922517)

[1.5、数据完整型 8](#_Toc514922518)

[1.6、GIT储藏（Stashing） 8](#_Toc514922519)

[1.7、GIT克隆（Clone） 9](#_Toc514922520)

[1.8、GIT对持续构建的影响 9](#_Toc514922521)

[1.9、托管平台与GUI工具 9](#_Toc514922522)

[1.10 GIT与SVN详细对比 10](#_Toc514922523)

[2、GIT学习使用 11](#_Toc514922524)

[2.1 Gitlab新建项目（代码仓库） 12](#_Toc514922525)

[2.2 SourceTree克隆远程代码库 13](#_Toc514922526)

[2.3 分支模型实践 14](#_Toc514922527)

[3、分支策略分析 25](#_Toc514922528)

[3.1、目前我司SVN分支策略 25](#_Toc514922529)

[3.2、基于Git分支模型-功能分支 26](#_Toc514922530)

[3.3、基于Git分支模型-版本分支 27](#_Toc514922531)

[3.4、基于Git分支模型-任务分支 28](#_Toc514922532)

[4、在开发云平台如何使用GIT 28](#_Toc514922533)

[4.1、开发云中的GITWorkFlow 28](#_Toc514922534)

[4.1.1、分支开发模式 29](#_Toc514922535)

[4.1.2、自定义开发模式 30](#_Toc514922536)

[4.2、创建代码库 30](#_Toc514922537)

[4.3、创建分支 31](#_Toc514922538)

[4.4、使用开发云进行持续交付 32](#_Toc514922539)

[4.4.1、规划版本 32](#_Toc514922540)

[4.4.2、Clone代码 35](#_Toc514922541)

[4.4.3、创建特性分支 38](#_Toc514922542)

[4.4.4、代码提交 38](#_Toc514922543)

[4.4.5、个人级(特性分支)构建 40](#_Toc514922544)

[4.4.6、特性分支merge代码到开发分支 42](#_Toc514922545)

[4.4.8、评审人员进行代码评审 43](#_Toc514922546)

[4.4.9、接收合并请求 45](#_Toc514922547)

[4.4.10、创建tag 45](#_Toc514922548)

[4.4.11、版本级（开发分支）构建 46](#_Toc514922549)

[4.4.12、版本测试 46](#_Toc514922550)

[4.4.13、测试阶段缺陷处理 48](#_Toc514922551)

[4.4.14、测试通过后进行封板 48](#_Toc514922552)

[4.4.15、发布员进行生产发布 48](#_Toc514922553)

[4.4.16、开发分支合并master 49](#_Toc514922554)

[4.4.17、生产缺陷解决 49](#_Toc514922555)

[4.4.18、冲突解决 49](#_Toc514922556)

[4.5、SVN迁移GIT 50](#_Toc514922557)

[5、配置SSH密钥 52](#_Toc514922558)

[6、Git常用指令 53](#_Toc514922559)

[7、附录-GIT客户端工具介绍 54](#_Toc514922560)

[7.1、bash使用 55](#_Toc514922561)

[7.2、可视化图形管理 56](#_Toc514922562)

[7.3、插件 61](#_Toc514922563)

# 1、GIT介绍

GIT是一个开源的分布式版本控制系统，可以有效、高速的处理从很小到非常大的项目版本管理；亦非常适合用于多分支并行开发，可灵活定制各种开发Workflow。

## 1.1、分布式仓储

GIT跟SVN一样有自己的集中式版本库或服务器， GIT更倾向于被使用于分布式模式，也就是每个开发人员从中心版本库/服务器上check out代码后会在自己的机器上克隆一个自己的版本库。可以这样说，如果在一个不能连接网络或者网络不稳定的情况下，就像在飞机上、地下室等，你仍然能够在本地仓库上进行开发工作。

 开发者可以从本地库查看完整的历史版本记录，无需从中心库加载；创建分支，提交修改也是本地行为，从而加快操作速度，让使用者感觉GIT各种操作都很高效。

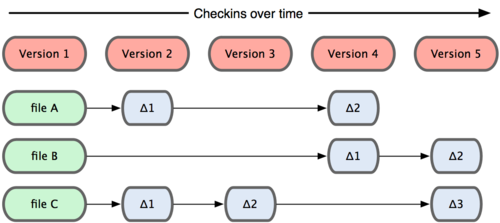
相反，如SVN等集中式代码仓库，开发者所做的任何操作都会对中央仓库发生数据同步，会导致网络负载增加，导致网络抖动；任何分支数据都会产生在服务器上，导致中央仓库容量暴增。

## 1.2、GIT快照（Snapshot）

有不少人认为Git存储的是每次文件相对于“基本文件”或“上一个提交中文件”的差异，其实这是很多人的误区。的确有不少版本控制软件就是这样做的，如SVN等等，但Git不是，它记录的是快照（Snapshot）。但重点是这快照究竟是什么呢？对记录差异和记录快照进行了对比。

### 1.2.1 SVN记录差异

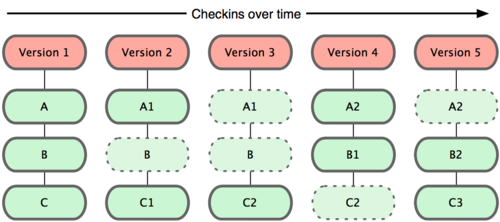
记录差异的原理图：



如图：版本2中记录的是文件A和C的变化，而版本3中记录文件C的变化，以此类推；这类版本控制软件记录的是一组基本文件和每个文件随时间逐步积累的差异，这个看起来非常好理解。这种记录差别的方法，通常对于文本文件比较有效，而对于一些二进制文件如图片、音视频文件，则无法记录其差异，再次提交将会产生另一个完整的文件。

### 1.2.2 GIT记录快照

而Git中，并不保存这些文件前后变化的差异数据，而是对整个缓存区中的所有文件采用了一种叫快照（Snapshot）的技术，有变化的文件保存，没变化的文件不保存；并对上一次的保存的快照做一个链接！



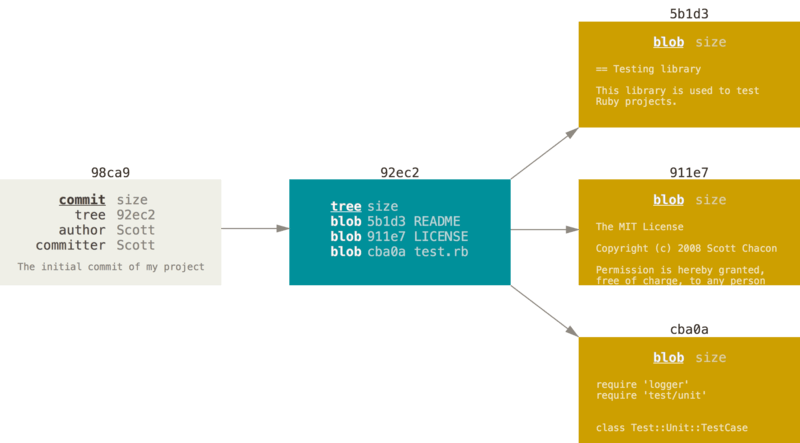
为了更加形象地说明，我们假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。 暂存操作“git add”会为每一个文件计算校验和（SHA-1 哈希算法），然后会把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 对象来保存它们），最终将校验和加入到暂存区域等待提交：

$ git add README test.rb LICENSE

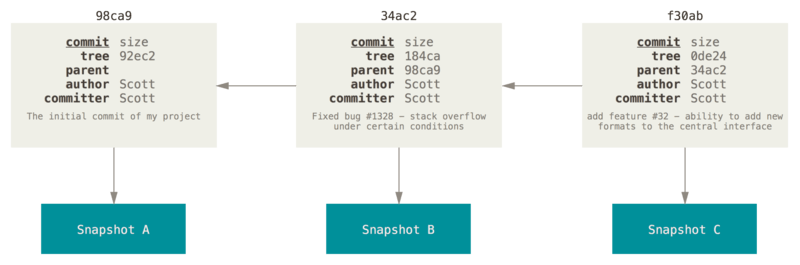
$ git commit -m 'The initial commit of my project'

当使用 ‘git commit’ 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在 Git 仓库中这些校验和保存为树对象。 随后，Git 便会创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引）以及一个提交对象（包含着指向前一个树对象的指针和所有提交信息）。



做些修改后再次提交， 那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



那么GIT使用Snapshot快照存储文件有什么意义呢？

原因很简单，你可以理解为GIT为每个修改过的每个文件生成了一份新的完整副本（快照），当我们下次要检出此次提交的文件时，我们只需要直接加载这个快照就可以了。相反，有些版本管理工具如SVN是保存每个版本之间的变化， 这样虽然总的文件体积小，但是每检出一个文件都要从最开始的版本一个个修改叠加上去，效率很低。

所以GIT在做一些分支检出或切换操作时，速度非常快。

## 1.3、GIT存储压缩

前面提到，GIT存储文件的快照可以理解为一个新的文件副本，那么随着代码修改提交次数的增加，GIT仓库的容量是否会随之增加呢？而像SVN这种只记录文件差异的仓库是否存储效率更好呢？

但实际上，在使用GIT管理自己的代码仓库时，随着代码更改次数的增加，代码仓库的实际大小并不会有太大变化，反而比具有相同文件数和提交数的SVN仓库还要小！这的确是很神奇。

原因是GIT提供了相应的delta压缩算法； 进入任何一个Git项目的.git/objects目录，那些以两位十六进制数命名的子目录里保存的就是未压缩的原始对象（文件快照），但这只是最近创建的还没来得及压缩的一小部分对象。在pack目录中你会看见以压缩形态存储的大部分对象。

通过隐藏“文件的版本之间的delta”这一细节而只暴露“文件快照”这一概念，Git底层反而实现了更高的存储效率。因为delta不再局限于一个文件的相邻版本之间，如果你的项目里有两个文件内容相同，Git也只会存储一份对象；如果两个文件内容相似，Git也可以对它们作delta压缩。

所以，虽然GIT快照保存的是完整的文件，但是其压缩存储却只保存了文件的差异。

而SVN则没有这种delta压缩存储机制。

## 1.4、分支特性

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情况下持续工作。在很多版本控制系统中，这是个昂贵的过程，常常需要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。

在SVN这类的版本控制系统上，分支（Branch）是一个完整的目录，且这个目录拥有完整的实际文件。如果工作成员想要开启新的分支，那将会影响“全世界”！每个人都会拥有和你一样的分支，就是说你创建的分支都会被别人看到。

GIT上，每个工作成员可以任意在自己的本地版本库开启无限个分支。举例：当我尝试破坏自己的程序（安检测试） [仅仅做一些尝试性的工作，并且是以分支方式保存的]，并且想保留这些被修改的文件供日后使用，我可以开一个分支，做我喜欢的事。完全不需担心妨碍其他工作成员，只要我不合并及提交（Push）到主要/中心版本库，没有一个工作成员会被影响。等我不需要这个分支时，我只要把它从我的本地版本库删除即可，无关痛痒。

由于GIT支持本地仓库，使得GIT创建、删除和切换分支时无需从中心仓库加载任何数据；以及Snapshot存储技术运用，使得GIT分支检出非常快；这些特性都使得开发人员在进行分支操作时，显得非常高效和便捷。例如，在一个拥有数万个以上源码文件和大小在几G以上代码仓库中，GIT分支切换只需短短几秒钟的时间。

GIT的分支特性可谓是“必杀技特性”，而正是因为它，将 Git 从版本控制系统家族里区分出来。Git 分支的新建操作几乎也在瞬间完成，因为并不会生成一个新的仓库副本， 而仅仅是将新分支的标签打在某个已有分支的节点上。

GIT支持完整的分支操作，如分支合并（rebase，merge），回归reset，恢复revert等。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。

理解分支的概念并熟练运用后，你才会意识到为什么 Git 是一个如此强大而独特的工具，并从此真正改变你的开发方式。熟练使用GIT分支功能，对于多版本并行开发、单人多任务开发以及尝试性开发是强有力的支撑。

## 1.5、数据完整型

Git 使用 SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1 哈希值，作为指纹字符串，即commit 标识符。这能确保代码内容的完整性，在遇到磁盘故障和网络问题时降低对版本库的破坏。比如，在本地仓库和中央仓库使用pull/push进行数据传送时，接收方可以通过重新计算某个commit节点的sha1值，与收到的commit节点标识符进行比较，来判断网络传送时，数据是否有损坏。

## 1.6、GIT储藏（Stashing）

经常有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是Git 储藏（stashing）。

“储藏”可以获取你工作目录的中间状态——也就是你修改过的、被追踪的文件和暂存的变更——并将它保存到一个未完结变更的堆栈中，随时可以重新应用。

## 1.7、GIT克隆（Clone）

Git克隆（Clone）一个完整项目的速度非常快， 而SVN非常慢（SVN export/import）。

我们克隆一个拥有5个分支的完整项目以及版本库来说，SVN是同时复制5个版本的文件，也就是说重复五次同样的动作。而GIT只是获取文件的每个版本的元素，然后只载入主要的分支（Master）。 克隆一个拥有将近一万个提交（commit），五个分支，每个分支有大约1500个文件的SVN，耗了近一个小时，而GIT只用了区区的1分钟。

## 1.8、GIT对持续构建的影响

通常，‘持续构建’会检测某代码分支的更新，也就是检测是否有新的commit节点；如有更新，会下载该分支代码进行静态检查、编译和单元测试操作。

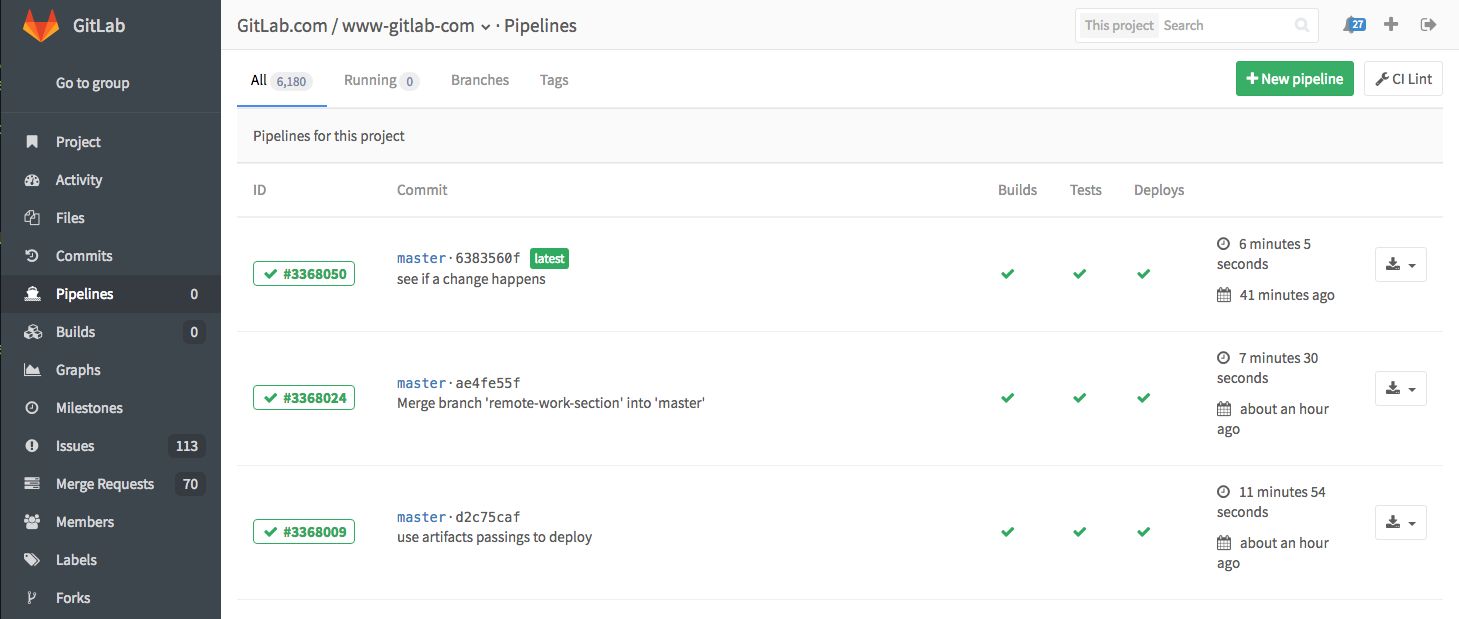
我司目前多数项目都采用SVN单分支开发模式，项目中所有成员都同时工作在这个分支上，对代码的所有改动都会提交到这个单分支上，造成该分支上commit节点过多，几乎每天都会有产生commit节点。这样会导致频繁触发该分支上的‘持续构建’操作，造成CI服务器和中心代码仓库的巨大压力；然而绝大多数被自动触发的‘持续构建’是不必要，比如某个开发人员只是提交了一个简单的改动。

在GIT多分支开发模式下，开发者通常会有自己的开发分支，而‘持续构建’往往只应用于项目的主分支或版本发布分支上； 所以开发人员在自己的分支上所作的任何commit提交都不会触发项目级‘持续构建’动作。 只有当新的功能开发分支合并回（Merge）项目的主分支或版本分支上，才会触发‘持续构建’。

开发人员可以在自己本地环境下执行对自己的开发分支的持续构建和测试， 以确保代码提交质量。通常，开发者自己的分支都仅仅存在于本地仓库里。

## 1.9、托管平台与GUI工具

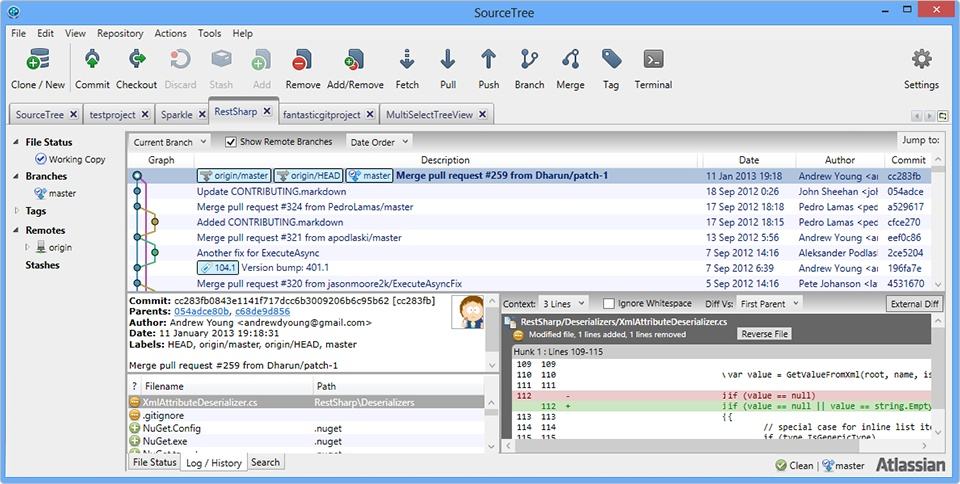
GTI托管平台有：GitLab，GitHub，BitBucket。



这些平台可方便用于：

* 创建项目Project，仓库Repo；
* 管理访问权限、读写权限、分支权限；
* 支持在线评审，定制分支merge策略（需多人评审，或主管评审）；
* 设置Commit Hook等；
* Dashboard；
* 相应API接口支持，便于外围开发。

GIT下拥有众多的客户端GUI工具，如SourceTree, GitKraken，TortoiseGit等，界面友好，易用性强；下图是SourceTree使用界面：



## 1.10 GIT与SVN详细对比

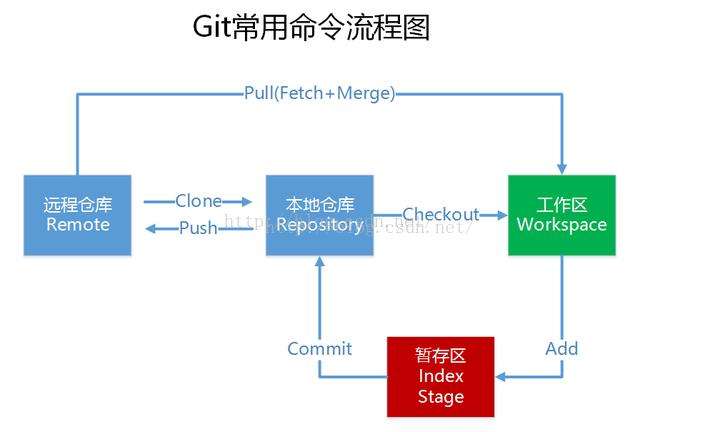
|  |  |  |
| --- | --- | --- |
| 版本工具  差异 | SVN | GIT |
| 系统特点 | 1.集中式版本控制系统（文档管理很方便）  2.企业内部并行集中开发  3.windows系统上开发推荐使用  4.克隆一个拥有将近一万个提交(commit),五个分支,每个分支有大约1500个文件，用时将近一个小时 | 1.分布式系统（代码管理很方便）  2.开源项目开发  3.mac,Linux系统上开发推荐使用  4.克隆一个拥有将近一万个提交(commit),五个分支,每个分支有大约1500个文件，用时1分钟 |
| 灵活性 | 1.搭载svn的服务器出现故障，无法与之交互  2.所有的svn操作都需要中央仓库交互（例：拉分支，看日志等） | 1.可以单机操作，git服务器故障也可以在本地git仓库工作  2.除了push和pull（或fetch）操作，其他都可以在本地操作  3.根据自己开发任务任意在本地创建分支  4.日志都是在本地查看，效率较高 |
| 安全性 | 较差，定期备份，并且是整个svn都得备份 | 较高，每个开发者的本地就是一套完整版本库，记录着版本库的所有信息（gitlab集成了备份功能） |
| 分支方面 | 1.拉分支更像是copy一个路径  2.可针对任何子目录进行branch  3.拉分支的时间较慢，因为拉分支相当于copy  4.创建完分支后，影响全部成员，每个人都会拥有这个分支  5.多分支并行开发较重（工作较多而且繁琐） | 1.我可以在Git的任意一个提交点（commit point）开启分支！  2.拉分支时间较快，因为拉分支只是创建文件的指针和HEAD  3.自己本地创建的分支不会影响其他人  4.比较适合多分支并行开发  5.git checkout hash值(切回之前的版本，无需版本回退)  6.强大的cherry-pick |
| 版本控制 | 1. 保存前后变化的差异数据，作为版本控制  2. 版本号进行控制，每次操作都会产生一个高版本号（svn的全局版本号，这是svn一个较大的特点，git是hash值） | 1. git只关心文件数据的整体发生变化，更像是把文件做快照，文件没有改变时，分支只想这个文件的指针不会改变，文件发生改变，指针指向新版本  2. 40 位长的哈希值作为版本号，没有先后之分  3. git rebase操作可以更好的保持提交记录的整洁 |
| 工作流程 | 1. 每次更改文件之前都得update操作，有的时候修改过程中这个文件有更新，commit不会成功  2. 有冲突，会打断提交动作（冲突解决是一个提交速度的竞赛：手快者，先提交，平安无事；手慢者，后提交，可能遇到麻烦的冲突解决。） | 1. 开始工作前进行fetch操作，完成开发工作后push操作，有冲突解决冲突  2. git的提交过程不会被打断，有冲突会标记冲突文件  3. gitflow流程（经典） |
| 内容管理 | svn对中文支持好，操作简单，适用于大众 | 对程序的源代码管理方便，代码库占用的空间少，易于分支化管理 |
| 学习成本 | 使用起来更方便，svn对中文支持好，操作简单，适用于大众 | 更在乎效率而不是易用性，成本较高（有很多独有的命令，rebase，远程仓库交互的命令，等等） |
| 权限管理 | svn的权限管理相当严格，可以按组、个人针对某个子目录的权限控制（每个目录下都会有个.svn的隐藏文件） | git没有严格的权限管理控制，只有账号角色划分（在项目的home文件下有且只有一个.svn目录） |
| 管理平台 | SVNBucket，AliSVN | gitlab（建议使用，集成的功能较多，API开发），gerrit，github等 |

# 2、GIT学习使用

GIT中心仓库托管平台：Gitlab，内网地址：<http://opensource.cnsuning.com>,这个托管平台是内部交流学习使用，是大家学习使用Git，内部知识分享的代码托管平台，可以理解为公司的GitHub。不作为正式项目的代码托管使用。

客户端工具： Atlassian SourceTree(其它客户端工具可以参考8、附录-GIT客户端工具介绍)

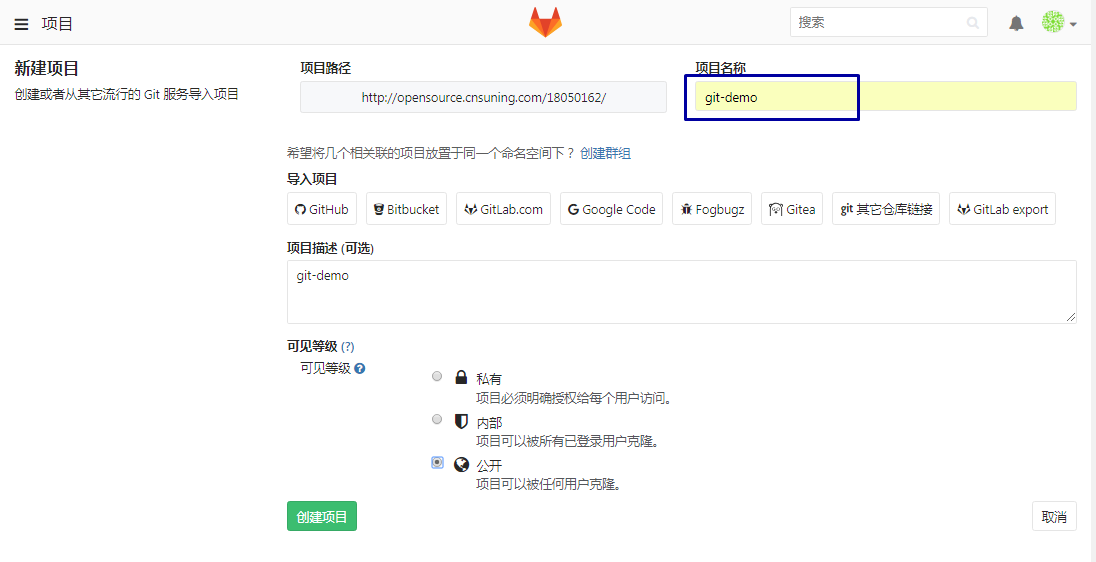
在本章中，我们将以实例演示一边GIT常用操作流程，以便我们对GIT的使用有个完整的概念。



## 2.1 Gitlab新建项目（代码仓库）

登入Gitlab [<http://opensource.cnsuning.com>]，账号为员工OA账号。

新建一个项目（代码仓库），操作如下：



项目创建成功后，在项目中创建一个新文件，并自动生成一个默认主分支Master。如下图所示：



## 2.2 SourceTree克隆远程代码库

使用SourceTree克隆（Clone）运程中心代码仓库，并生成本地仓库。

1. 打开Gitlab中相关项目页面，选择SSH方式，并复制生成的URL。

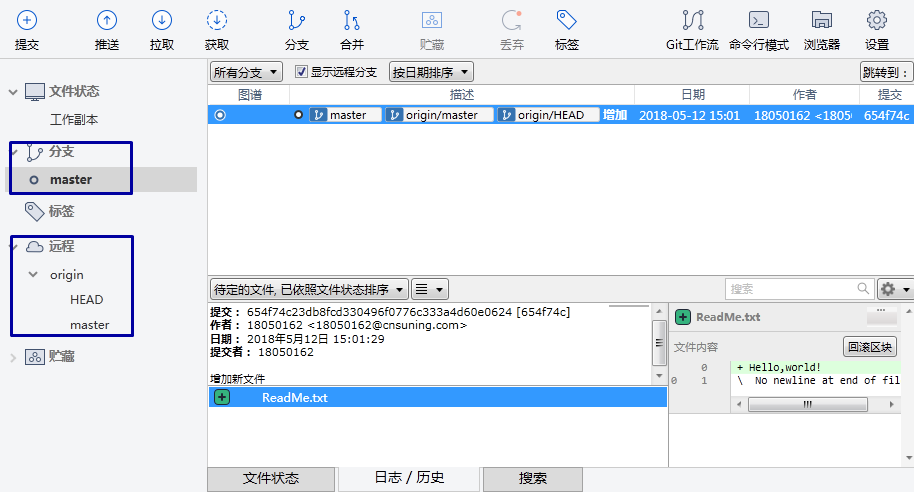


1. 在SourceTree中点击Clone‘克隆’按钮，在生成的界面中输入运程库URL，以及设置本地库的路径。



点击‘克隆’。

1. **Clone**结束，可以在SourceTree看到本地代码库及远程代码库信息.

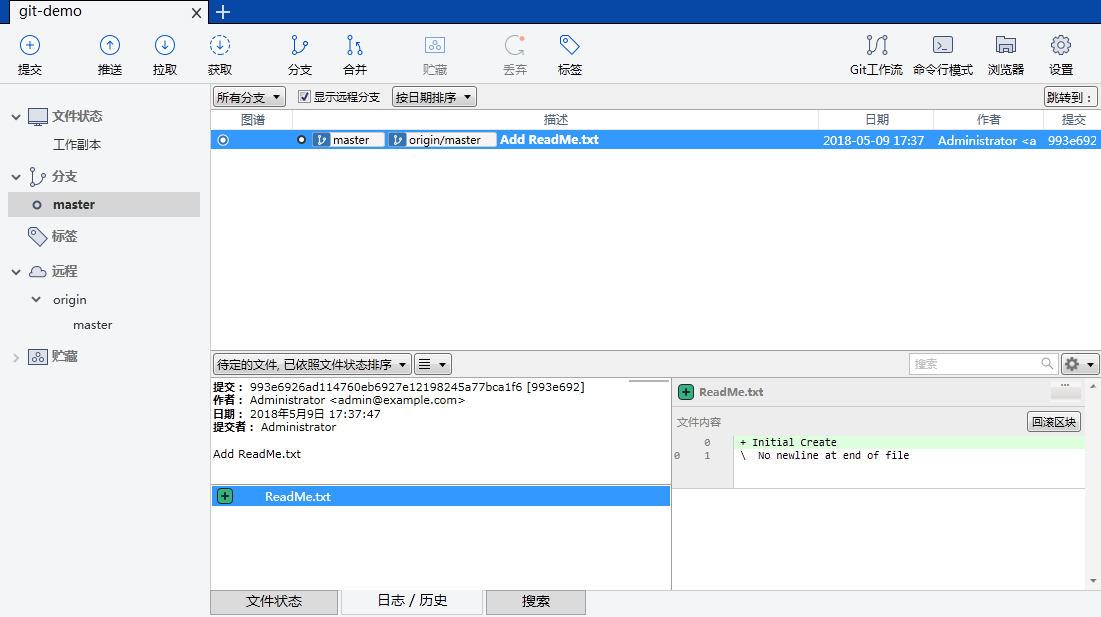


初次**Clone**成功后，本地库所有数据与分支信息和远程库是一致的。

## 2.3 分支模型实践

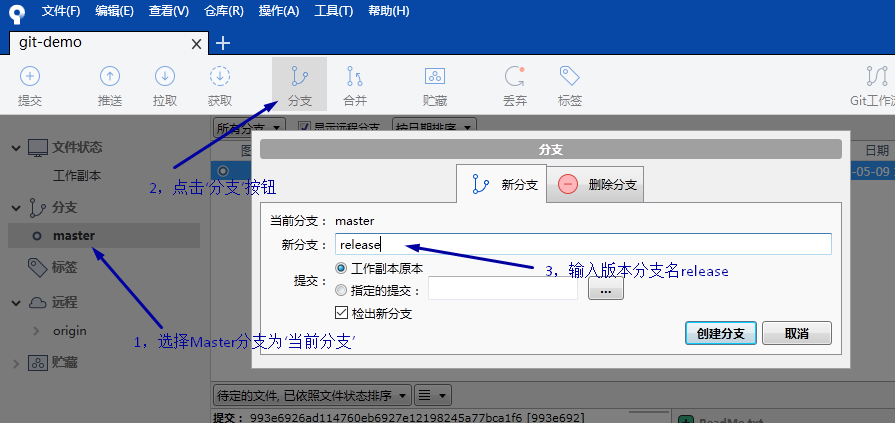
以多版本并行开发分支使用策略为例，演示如何创建分支，提交及合并修改。

我们以一个新创建的仓库为例，该仓库中只有ReadMe.txt文件。所以从中心仓库Clone下来后，只看到仅有一个Master分支。



**1，依据多版本开发的分支策略，我们需要建立相关分支。**

首先基于Master分支创建一个版本发布分支Release。在本地分支数中选定Master分支，在工具栏中点击‘分支’按钮，



版本分支“release”创建成功后，会显示在本地仓库分支树中；

依次我们再基于‘release’分支创建开发主干分支‘develop’；

基于‘develop’分支创建‘功能1’主分支‘feature-1/main’；

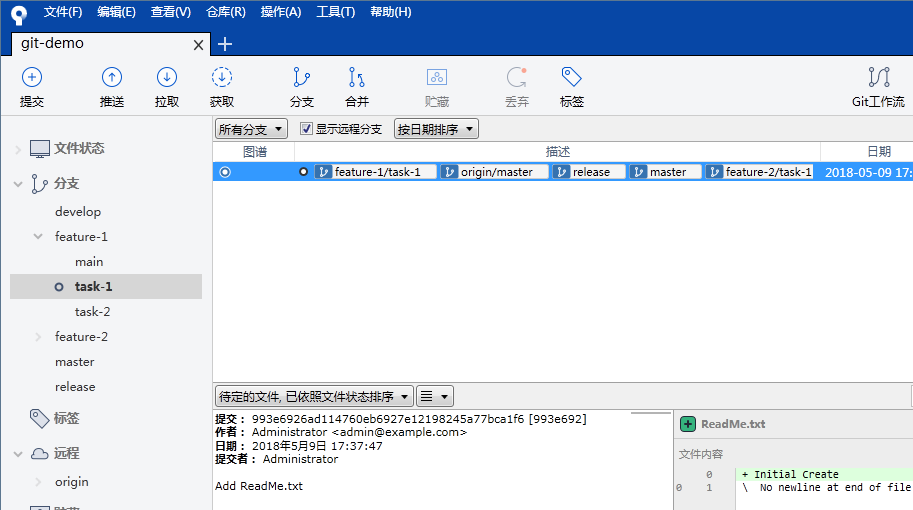
基于‘feature-1/main’分支创建‘任务1’分支‘feature-1/task-1’，

基于‘feature-1/main’分支创建‘任务2’分支‘feature-1/task-2’；（这里我们假定‘功能1’开发工作比较大，可分为多个子任务，由不同开发人员完成。）

基于‘develop’分支创建‘功能2’主分支‘feature-2’；

在feature-2分支上，不再创建子任务分支； （这里我们假定‘功能1’开发工作比较小，可由单个开发人员来完成。）

所有分支建好后，其本地分支树结构如下：



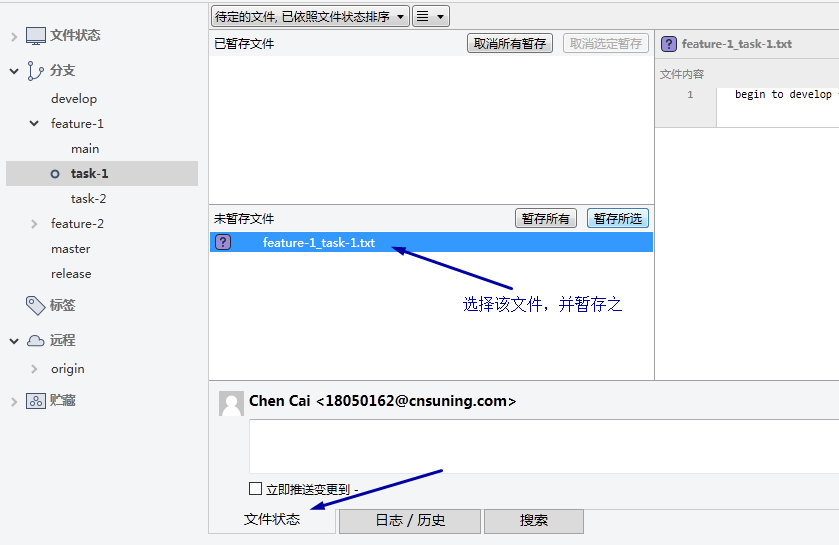
**2，实施feature-1的开发工作。**

双击feature-1/task-1分支，检出（checkout）该分支，我们开始在该分支进行开发工作；首先，创建一个新文件如feature-1\_task-1.txt；

保存文件后，SourceTree界面会有提示‘未提交的更改’。如图，



点击下方‘文件状态’tab键，进入文件操作界面；可以看到feature-1\_task-1.txt出现在‘未暂存文件’列表中，选择该文件，并点击‘暂存所选’。



feature-1\_task-1.txt将会被移至‘已暂存文件’列表中。

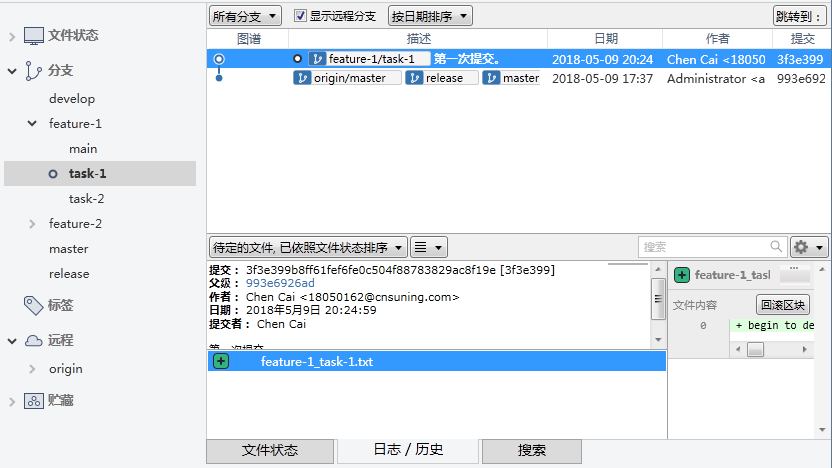
此处的‘暂存’操作其实就是执行‘git add’命令，将修改文件存入暂存区。



填写提交信息，不勾选‘立即推送变更到 -’（此处文本应为‘立即推送变更到运程仓库’）；点击提交。

此处的‘提交’操作其实就是执行‘git commit’命令，将暂存区的数据提交到本地代码仓库。

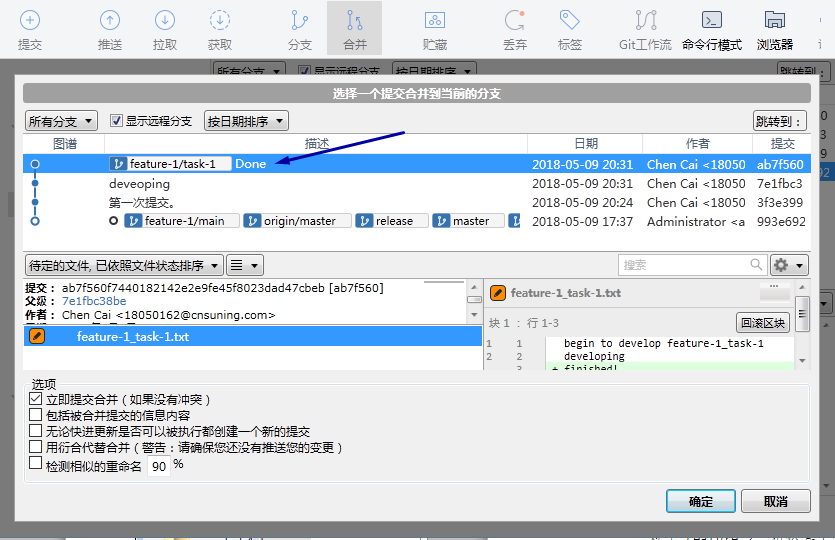
提交后将会在feature-1/task-1分支上生成一个新的commit节点。



可继续修改feature-1\_task-1.txt文件，并提交更改，生成多个commit节点。

现在我们要将feature-1/task-1分支合并（Merge）回feature-1/main分支，首先双击feature-1/main分支，选定为‘当前分支’（即执行checkout命令）；点击功能栏‘合并’按钮。

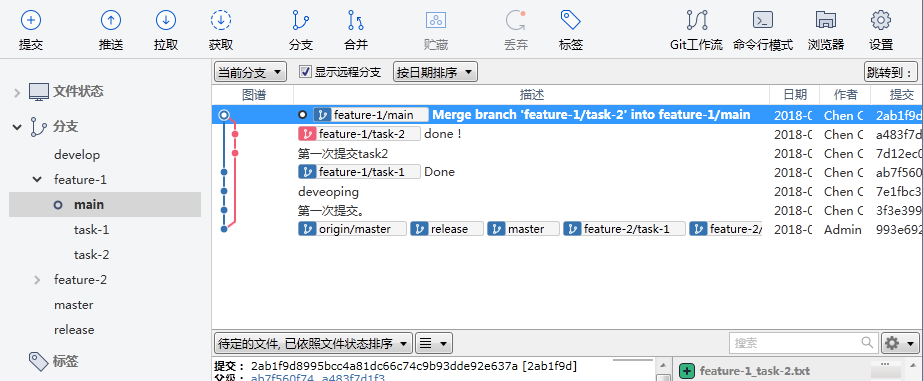
在弹出的对话框中，选择‘feature-1/task-1’分支，点击‘确定’。



合并结束后，可以在分支图谱查看到合并信息。

对feature-1/task-2分支进行同样的操作，生成多个commit节点，然后合并回feature-1/main主分支。

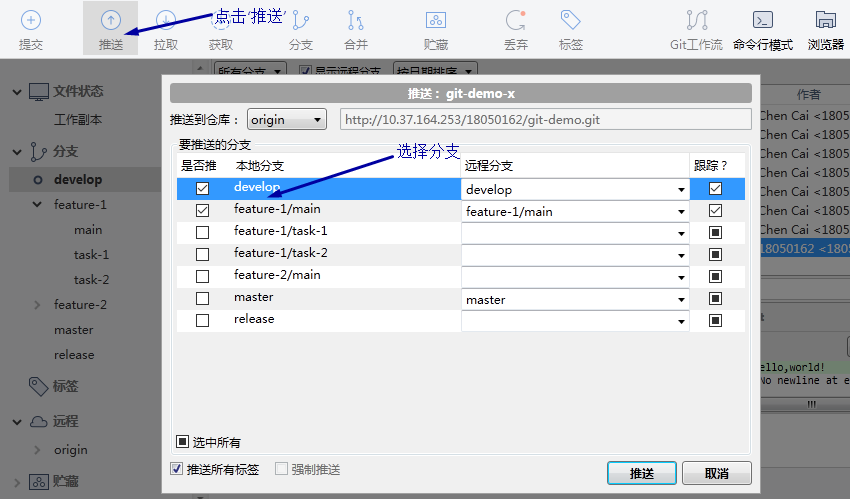
完成后，commit图谱显示状态如下：



此时，由于feature-1的两个task都已完成并merge回feature-1的主分支，可认为feature-1开发完毕，接着需要将feature-1分支merge回develop主分支。

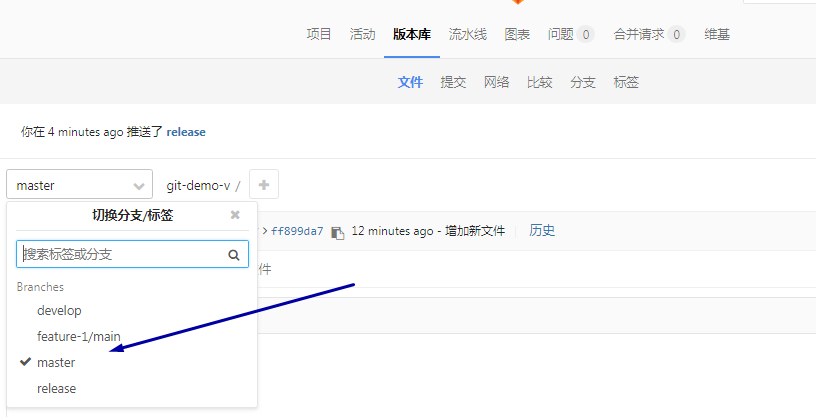
**3，将feature-1的更新合并回‘merge’开发主干分支‘develop’**

通常，功能分支合并回开发主分支‘develop’，需要经过代码评审‘review’。此时需要将feature-1/main分支push到远端中心仓库。



点击“推送”，选择需要推送到远程仓库的分支， 这里我们选择了develop和feature-1/main分支。理论上说，develop应该在远程中心仓库上创建，在本地第一次克隆远程仓库时，就能够获取到；如果这样，这里就不需要选择develop。但在本实践中，develop分支在本地创建，所以需要勾选。

在推送分支过程中，会提示你输入gitlab的账号和密码。推送成功后，就可以在gitlab看到被推送的分支。



然后就可以在Gitlab上发起在线评审；发起在线评审操作可以通过创建一个pull request（Gitlab中叫做合并请求“merge request”），选择“来源分支”为：feature-1/main, “目标分支”为：develop，点击“比较分支后继续”按钮。



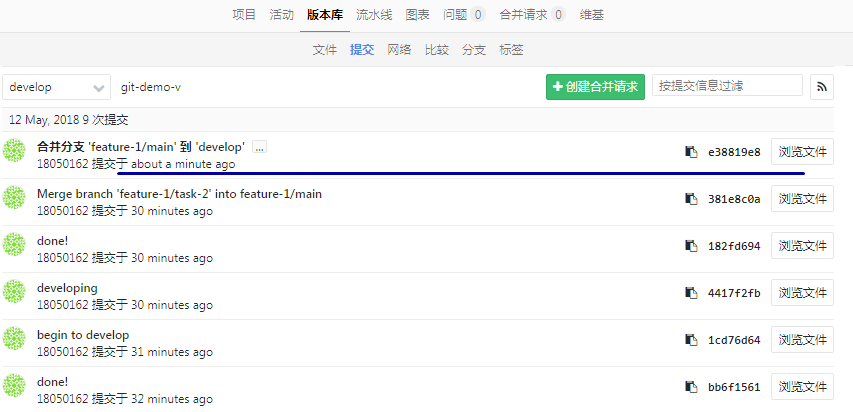
在下一个页面中添加相关信息，并加入相关评审人员，点击“提交新的合并请求”。如下图：



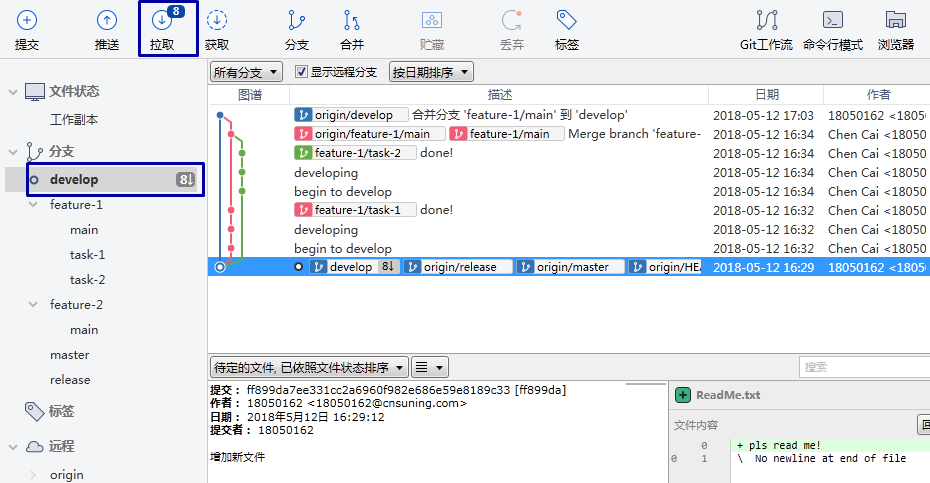
在合并请求页面，评审人员可以看到此次合并的修改，可以没代码行插入评审意见；如评审通过，则点击按钮“接受合并请求”。



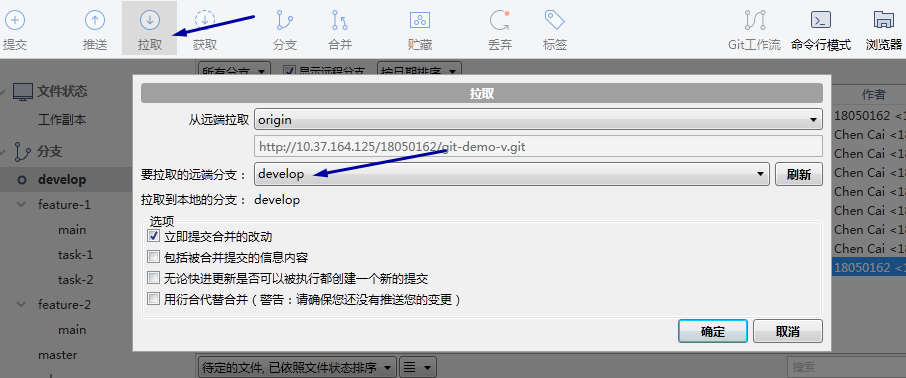
合并后将会在中心代码库的develop分支上生成一个新的commit，如下图：



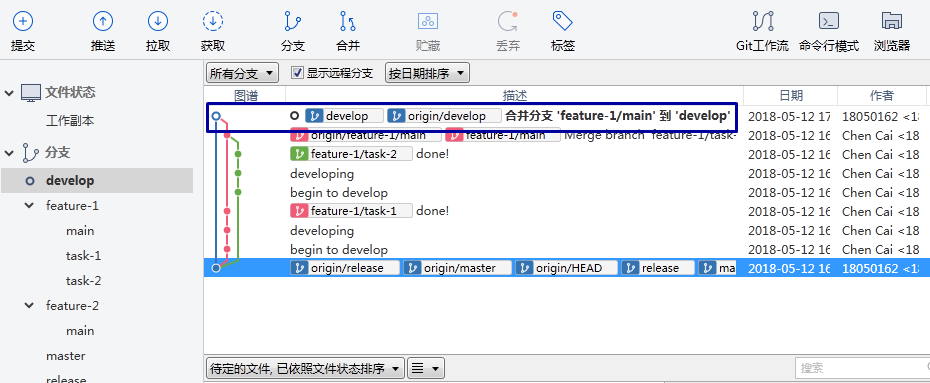
此时在本地仓库中，我们是看不到develop分支有相关更新的；但我们可以在SourceTree界面上看到远程仓库有更新，在下图中可以看到本地仓库develop分支已落后远程仓库8次提交。



我们需要将远程仓库中develop分支的更新“拉取”（pull）到本地仓库； 切换develop为当前分支，点SourceTree功能栏‘拉取’，在弹出的对话框中选择远程仓库develop分支，点击确定。



‘拉取’（pull）操作执行完成后，可以在本地仓库中看到develop分支图谱中已经有了相应的合并信息。

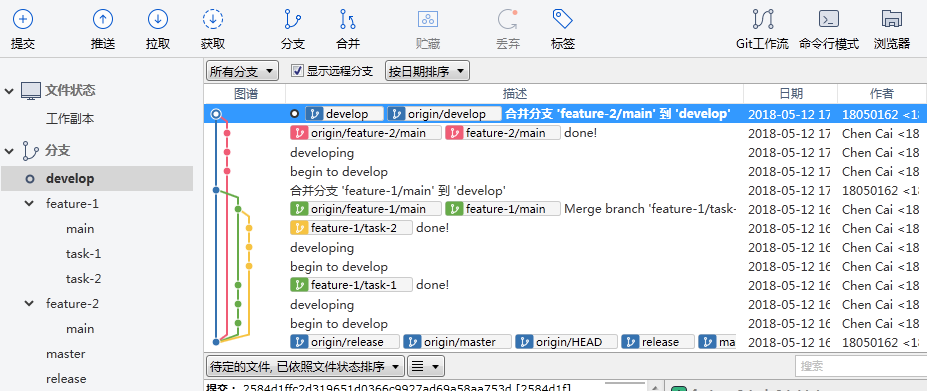


**4，实施feature-2的开发工作。**

同样，完成对feature-2的功能开发，push到远程中心仓库，通过创建pull request合并回develop主分支。然后本地仓库中拉取（pull）远程仓库中develop分支的更新。

对于feature-2，虽然其工作量较少，可由单个开发人员完成；但强烈建议为其创建功能主分支如feature-2/main，以及开发者分支如feature-2/task-x；因为如果开发者直接工作在feature-x分支上，会在该分支上进行频繁的commit操作，生成很多的分支节点；而由于feature-x分支需要push到运程中心代码仓库进行评审‘Review’及合并， 那么这个feature-x所有的commit都会push到中心仓库，这是不必要的，也会造成中心仓库数据量增大；其他团队成员在Pull/Fetch该功能分支时，也会看到这些过多的Commit节点。

操作完成后，本地仓的分支图谱如下：



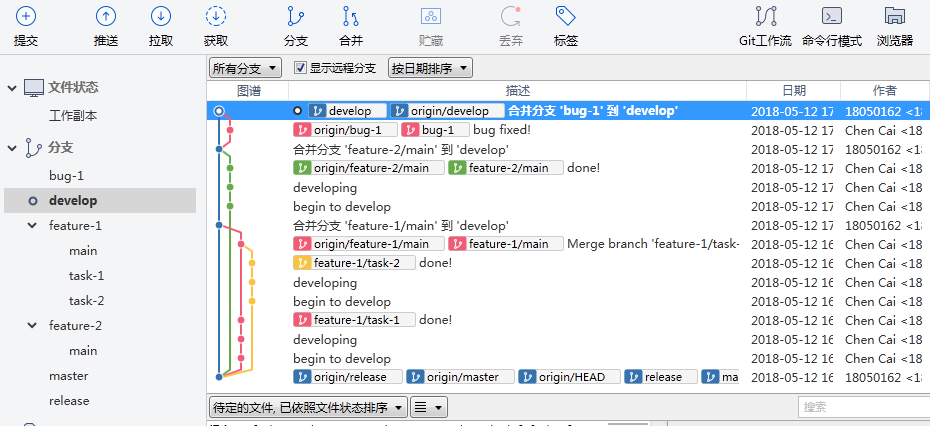
可以看到，feature-1和feature-2的功能分支已合并到develop分支中了。

**5，实施测试工作**

当所有开发成果合并回develop分支后，该版本的测试工作就可以开始执行了。

我们需要从中心仓库的develop分支拉出代码，执行构建、部署等操作将版本发布到对应的测试环境中，例如SIT环境，而后开始执行测试。

测试过程发现的任何bug，应在develop分支上拉出bug-x分支进行修改。Bug的修改同样需要push到远程中心仓库，并在gitlab中创建合并请求并评审。通过下图可以看到，bug-1已修改并合并会develop分支。



**6，执行版本发布**

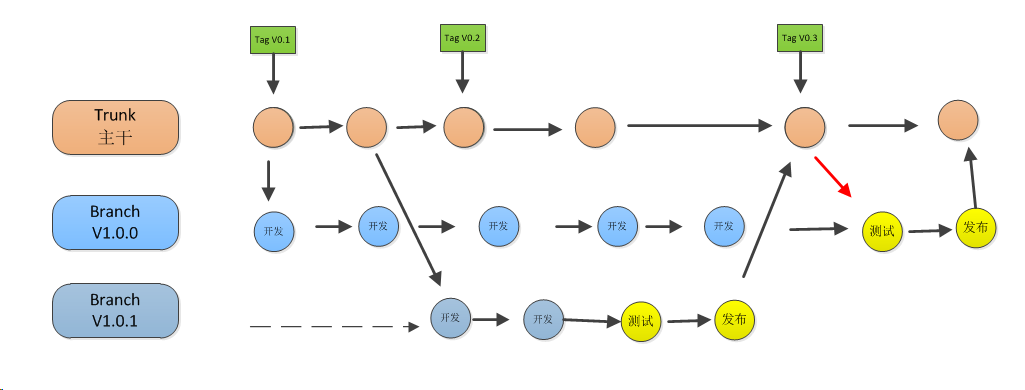
当develop分支在经过测试、bug修改后，版本代码处于可发布状态；这个时候就需要将develop分支合并会版本分支release。同样，需要在Gitlab托管平台上创建pull request来完成在线评审、合并操作；通常这个合并‘merge’操作需要经过版本负责人或产品经理的审批。

在版本代码进入release分支后，我们就可以在该分支上构建正式版本，并部署到生产环境；并合并到master主分支归档。

# 3、分支策略分析

## 3.1、目前我司SVN分支策略

目前我司使用的SVN版本管理方式主要是分支开发模式，在分支开发模式中，trunk与生产保持一致，一个新版本开始的时候，需要基于trunk拉取一个新的分支做开发分支，并且基于这个分支进行开发、测试、发布。这种集中式分支的开发，开发人员往一个集中式分支提交代码。如果在多分支并行开发的情况下，需要经常解决冲突和代码回合。如下图所示分支策略。



目前这种集中式分分支管理策略，策略相对简单，分支管理成本较低，使用是比较简单。随着业务的快速发展，业务上要求快速交付，这样集中式的分支开发模式对于一些多版本并发开发，持续集成支持效果上看还存在一定提升空间。

开发方面：单独分支开发，开发人员往一个分支上提交代码，每个人提交代码并没有进行代码质量检查及单元测试，提交后基于集中式的分支进行持续集成，分支构建，分支构失败率较高，效率不高。

测试方面：测试基于集中式分支进行测试，由于开发过程任何人都可以提交代码，这时并不能保证一个稳定的测试分支，测试过程效率不高。

多版本并行方面：基于svn的分支策略，多版本分支是从trunk上拉下来的，分支上线后系统合并trunk以及回合其它未上线分支，因为SVN架构上的限制，合并，回合冲突解决，效率较低。

## 3.2、基于Git分支模型-功能分支

当一版本开发中存在多个功能点（feature），且这些功能点由不同的开发人员来完成。这种情况下，可以为每个功能点的开发建一个单独的feature分支，如下图所示：

Tag V1.6

Tag V1.5

Hotfix-1

Pull latest from **Master**

Bug-1

主分支/**Master**

开发主分支/**Develop**

**Feature**分支-1

**Feature**分支-2

如上图所示，该应用开发涉及多feature的并行开发，开发者可以工作在不同的feature分支上，提交代码时则互不影响；每个feature开发完成后，才合并回（merge）开发主分支develop，这个时候即便有冲突（conflict），也仅仅只需要在合并的时候修改一次。

当该阶段所有功能开发完成并合并回develop分支后，就开始在develop分支上执行构建版本，部署并测试。在测试过程中，任何bug都直接在develop拉出bug分支来修改；在bug分支上进行修改并自测后，再把bug分支合并回develop分支，以便进行正式的测试验证； 而同时，可以继续从develop分支继续拉出新的feature分支，进行新功能开发。

当前版本经过充分测试而稳定后，可以合并回Master主分支。

## 3.3、基于Git分支模型-版本分支

当同一应用存在多个版本进行开发时，可以为每个版本建立一个单独的版本分支Release。版本的发布及hotfix将发生在release分支上；而主分支Master则只是用于多版本的归档与共享。

Tag V1.0

Tag V1.1

Merge back to Master

Hotfix

Pull latest from **Master**

Bug-1

主分支/**Master**

版本分支/**Release**

开发主分支/**Develop**

**Feature**分支-1

**Feature**分支-2

1. 通过创建release（版本）分支，可有效的进行权限控制，普通开发人员无法随意在该分支上执行修改，所有hotfix的pull request都需要版本经理审批；
2. 新功能持续开发和已发版本的维护在不同分支上，比较清晰；Hotfix打在Release版本分支上，可以无需打在Develop开发主分支上，降低冲突风险。
3. Release分支发布对应于正式版本，适用于Pre，Prd环境；而Develop分支发布的版本为测试版本，适用于SIT等测试环境。

## 3.4、基于Git分支模型-任务分支

主分支/Master

版本分支/**Release**

开发主分支/Dev

**Feature**分支-1

**Feature**分支-2

**Task**分支-1

**Task**分支-2

项目团队可以根据所开发版本中功能点（feature）的大小，灵活创建新的任务（task）子分支开发；如果某个功能点太大，需要多人参与开发，则需建立多个任务分支，这样每个开发人员可以独立工作在一个单独的任务分支上。

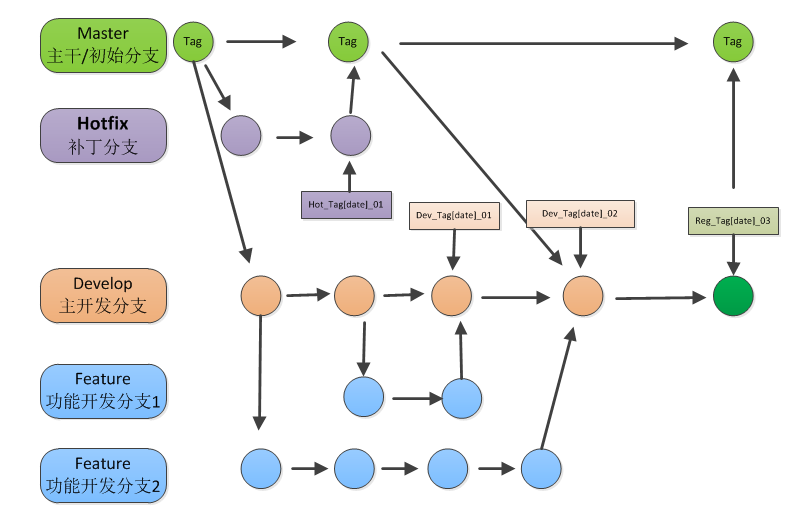
研发中心可以根据实际情况，灵活选择分支开发策略。

# 4、在开发云平台如何使用GIT

## 4.1、开发云中的GITWorkFlow

在开发云平台中针对GIT代码库提供了两种工作模式，即分支开发模式和自定义开发模式。

### 4.1.1、分支开发模式



分支模式包含三种分支类型：开发分支、特性分支、紧急分支

1）Master分支(主干分支)

此分支创建代码库时默认生成，这个分支只能从其他分支合并代码，不能在这个分支上直接操作。Master保持和当前生产运行的代码同步，所以每次版本发布后要将代码合并到Master。

2）Develop 分支（开发分支）

开发分支也是发布分支，一般一个独立项目或者一个版本需要拉取一个开发分支，开发分支主要接收特性分支合并的代码，不能被直接提交代码。develop分支可多个并存，应用负责人、版本经理和技术经理拥有创建权限。

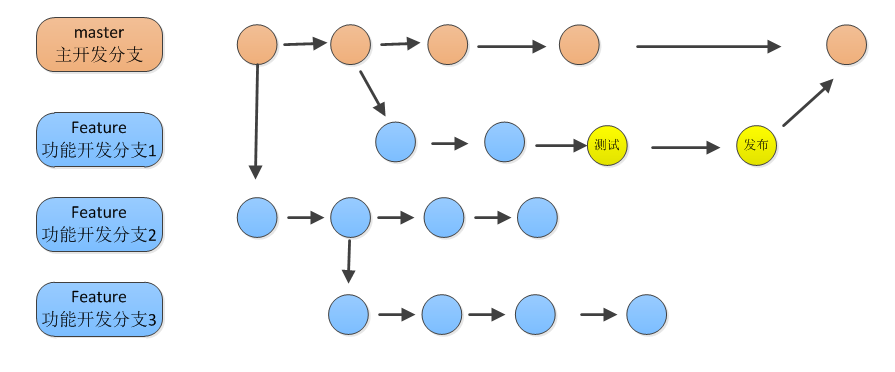
3）Feature 分支(特性分支)

特性分支主要是进行具体功能或任务开发，功能一旦开发完成，要及时合并到Develop分支进行集成测试。所有开发人员都拥有拉取特性分支的权限。开发团队可针对每个独立的子任务拉取一个特性分支，亦可每人使用一个特性分支，建议每个开发人员使用一个特性分支。

4）Hotfix分支(紧急分支)

这是一个特殊的开发分支，当我们在生产环境发现Bug的时候，我们需要创建一个Hotfix,来紧急修复缺陷，完成Hotfix后，需要合并到Master和Develop分支，所以Hotfix的改动会进入下一个Release。

### 4.1.2、自定义开发模式



自由模式为比较自由的一种，只包含两种分支类型：主干分支（master）、开发分支。

master默认为受保护分支，创建代码库时默认生成，无法直接提交代码，只能通过合并请求来合并代码，master分支无法被关联到版本进行发布，master永远和生产最新版本的代码保持同步。

开发分支可以做发布和质量构建，开发分支可以多个并存，一般在多个版本并行开发的情况下会有多个开发分支并存。开发分支的拉取和使用有项目组自行设定策略，开发人员都有拉取开发分支的权限，项目组可共用一个开发分支，亦可根据需要每人拉取一个分支。

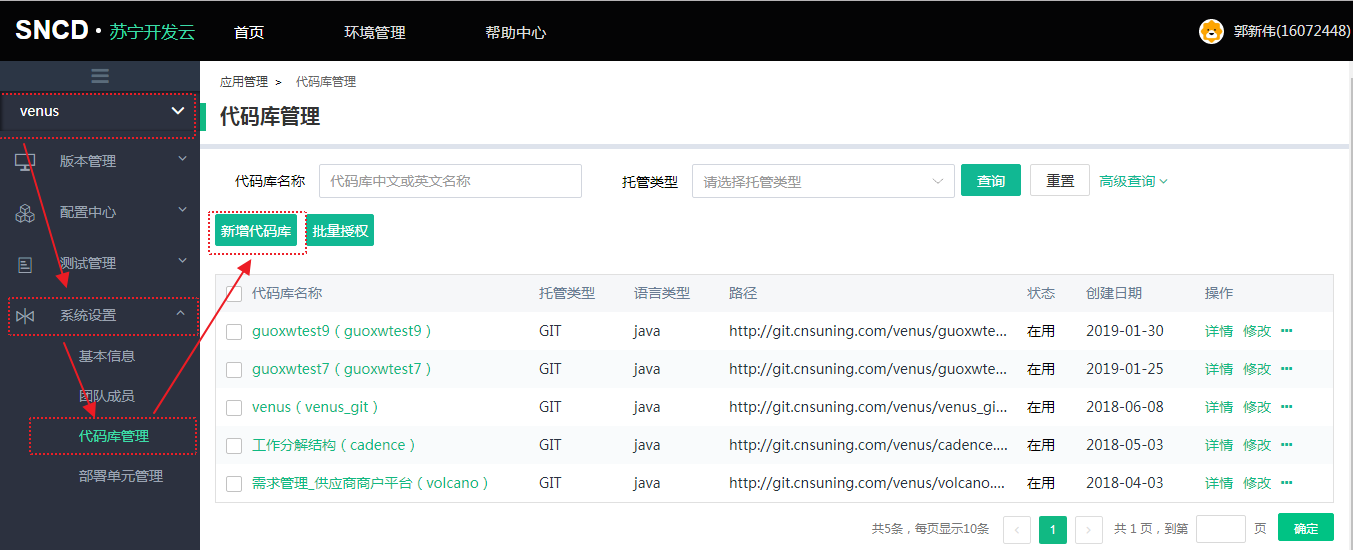
一般在实际开发过程技术经理会拉出一个分支作为主开发分支，每个开发人员拉出一个自己的开发分支，版本发布前将其他开发分支的内容最终合并到此主开发分支一起进行发布。

## 4.2、创建代码库

入口：<http://phoebus.cnsuning.com/>

菜单：应用管理🡪系统设置🡪代码库管理。

(若已有SVN代码库,想转换成GIT库可参考《4.5SVN迁移GIT》)



填写代码库信息



代码库创建成功后会默认生成两个分支（master 和 Dev\_Bryyyymmdd）, Dev\_Bryyyymmdd 为默认的开发分支（以Dev\_Br开头 + 当前日期）。

技术总监或应用负责人可以在开发云平台基于系统创建一个或多个git代码库。如果没有系统信息可以在itp平台进行系统信息注册。

## 4.3、创建分支

菜单：应用管理🡪系统设置🡪代码库管理



点击“新增分支”后，根据代码库选择的开发模式不同创建的分支类型也有所不同。具体分支模式详见《4.1开发云中的GITWorkFlow》

自定义开发模式：



分支开发模式：



## 4.4、使用开发云进行持续交付

本章节将以一个版本交付过程为例，示例说明如何用开发云基于git进行版本交付。

示例：某项目组在月初接到一些日常需求，需要在月底30号上线，经过规划这些需求需要分两个迭代（版本1.0、版本2.0）上线，版本1.0在20号上线，版本2.0在30号上线。经系统分析设计版本1.0包含了5个子需求 (task), 版本2.0包含了5个子需求（task）。若项目团队恰好五个开发人员每人领取一个开发任务。

### 4.4.1、规划版本

首先版本经理需在开发云(phoebus)中创建一个版本，命名为V1.0。版本创建后会同步到jira，产品经理可以在jira中将对应的需求或任务归入到当前V1.0版本。则版本的“工作项”中则会展示出来当前版本所涉及的需求或任务。

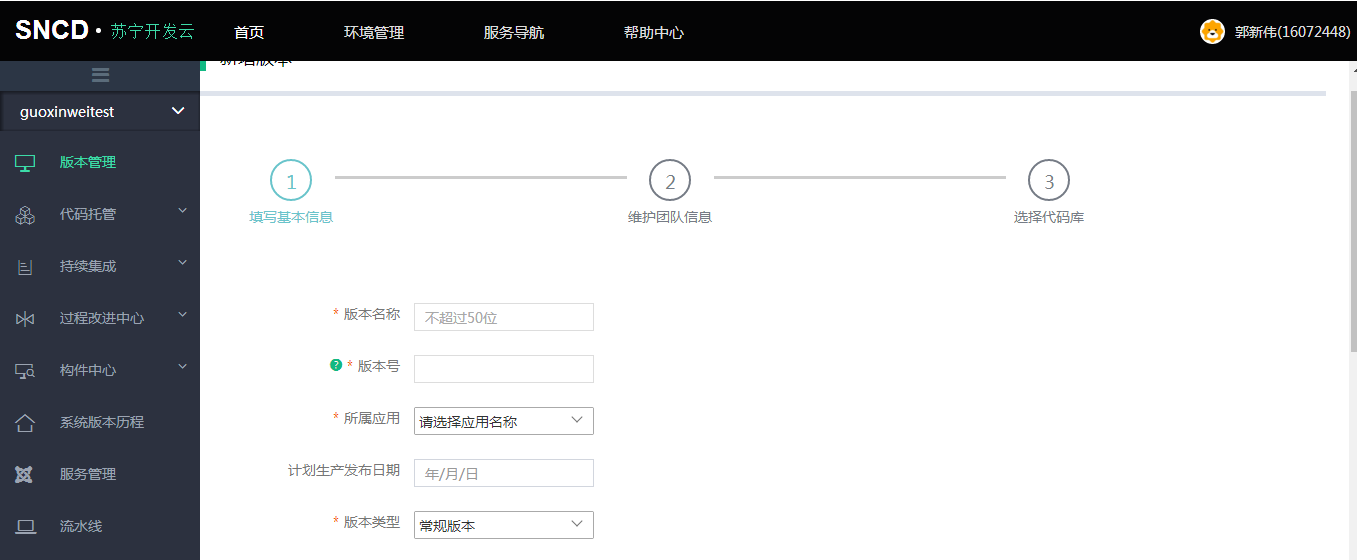
菜单入口：应用管理🡪版本管理

版本管理界面会有“+创建版本”按钮，若无创建按钮则说明不是版本经理，需要添加“版本经理”角色，祥见下面赋权介绍。

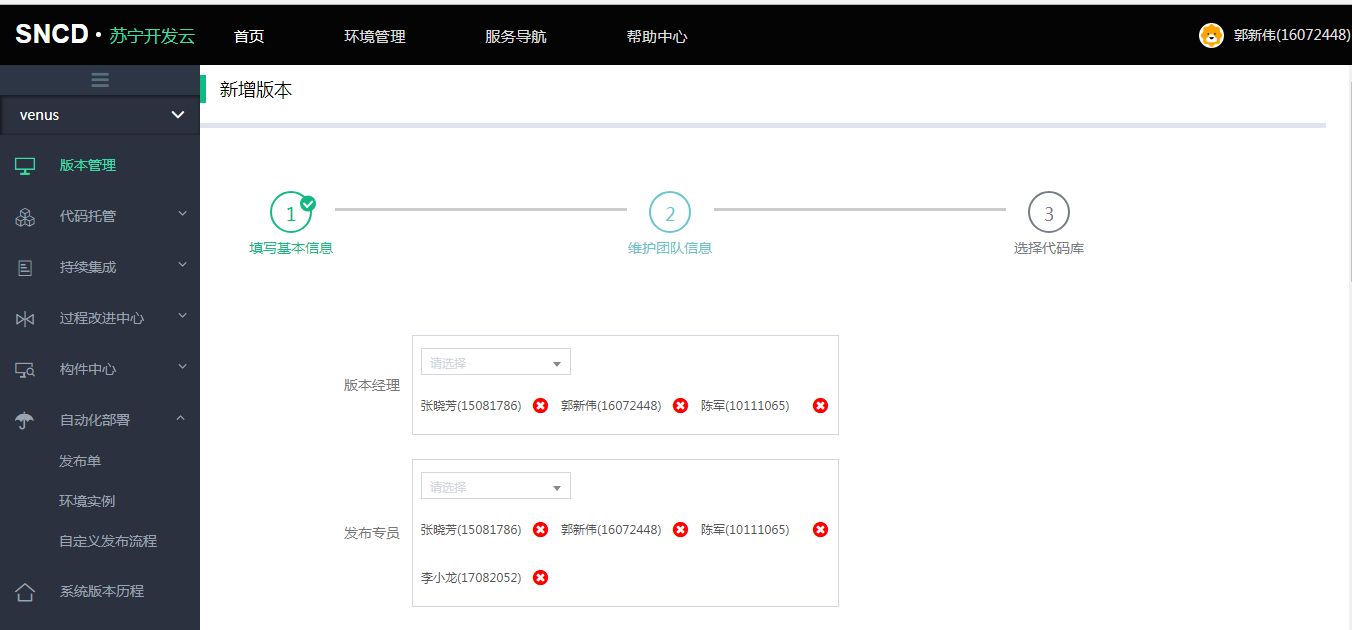
进入菜单后，点击“+创建版本”



点击“+创建版本”后，创建页面，填写版本基本信息

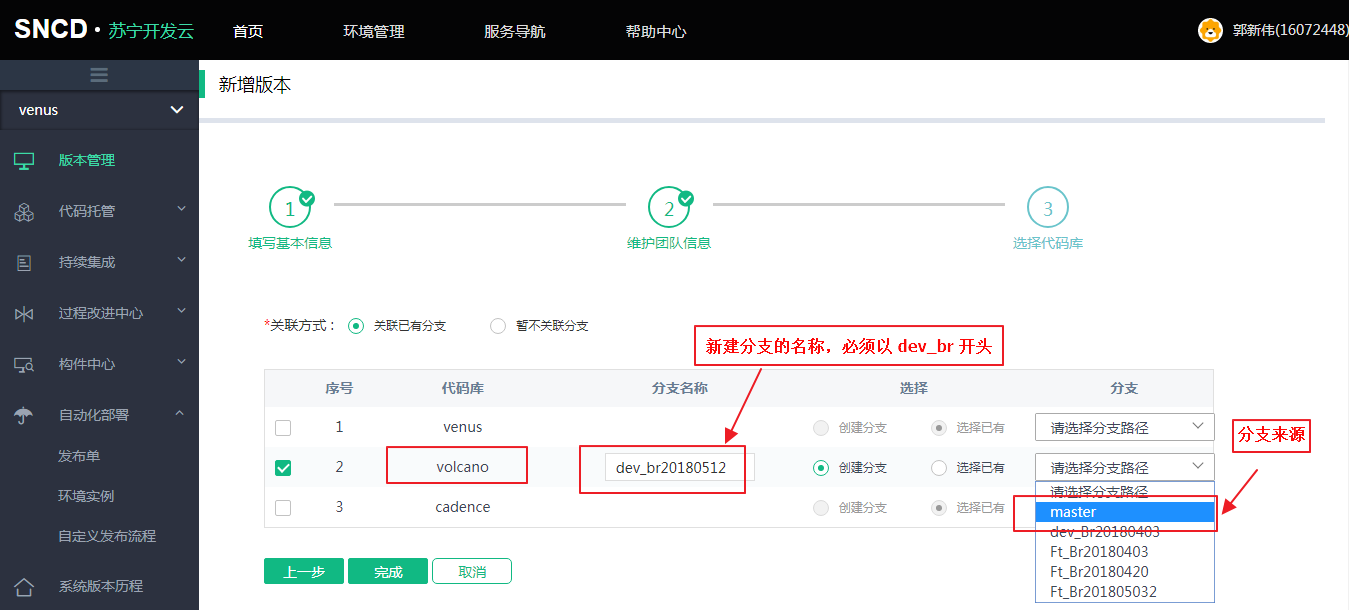


点击“下一步”，设置团队成员



点击“下一步”， 关联分支。

所有涉及开发任务的版本都需要关联一个开发分支，若无代码库可暂不关联，创建代码库后再做关联。创建代码库参考《4.2、创建代码库》章节。



【注】：版本团队中的“开发工程师”会默认带入到当前版本所关联的代码库的开发组中，即版本中的开发工程师默认会有读写权限。

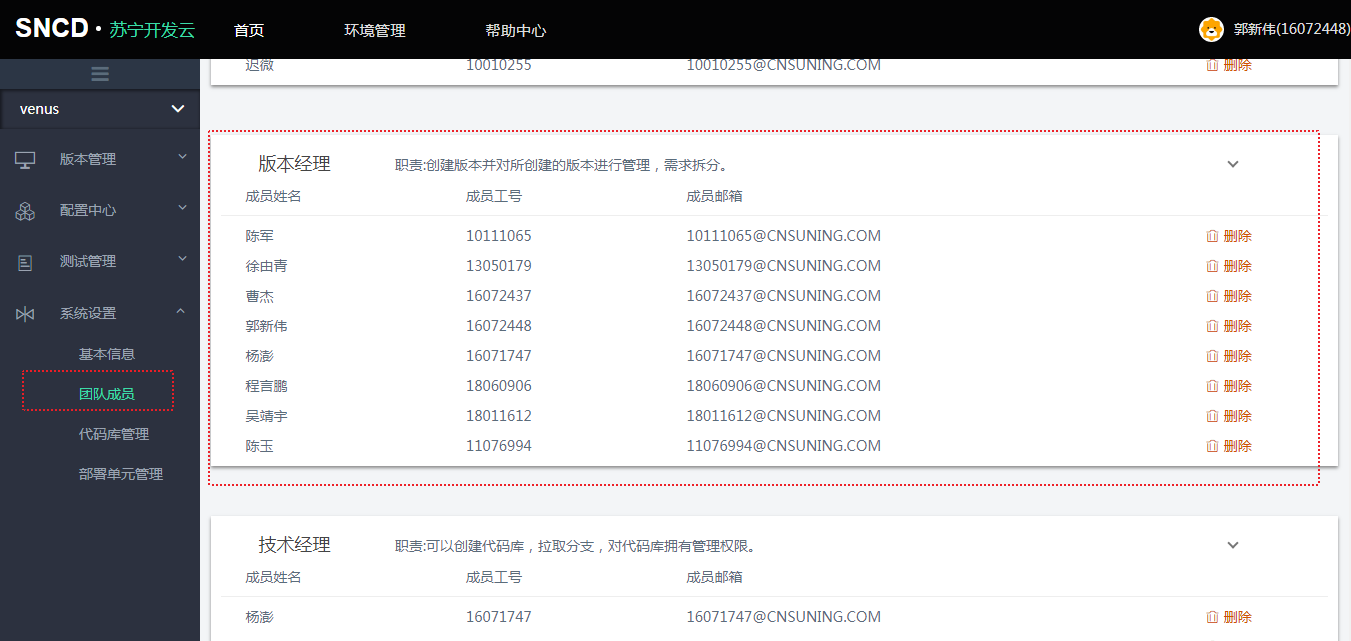
版本创建后若想关联分支，可直接在版本的“配置库”中添加。



若当前登录人员无创建版本的按钮，则说明当前登录人员不是版本经理。可到团队成员添加“版本经理”角色。



到应用的“系统设置🡪团队成员”查询是否版本经理，若不是则需要添加。



添加一个版本经理的角色（此处需要技术总监或者应用负责人添加）

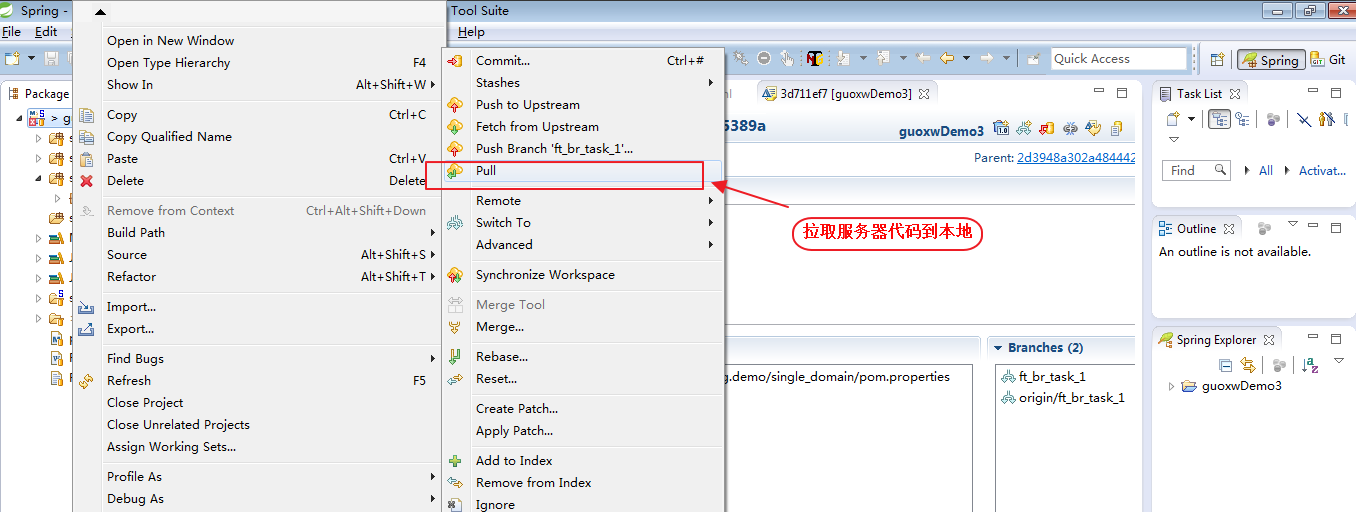


注：在原持续交付(SNCD)平台中一个版本就是一个分支，版本和分支是同一个概念，但是在开发云平台(phoebus)中版本和分支是两个不同的概念，在开发平台一个版本可以看做是敏捷中的一个迭代(sprint),一个迭代包含了团队成员、工作项(需求、任务、缺陷)、配置库（要发布的代码、配置文件、脚本等）、持续集成、发布管理、版本报告等，所以在开发云平台中分支只是版本的一部分，包含在配置库中。

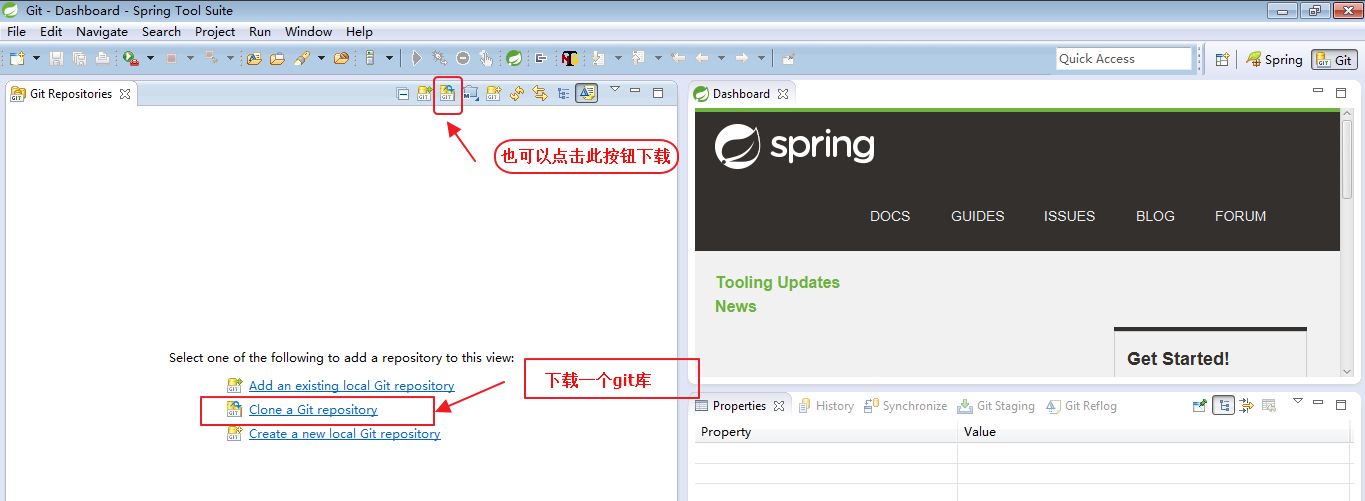
### 4.4.2、Clone代码

开发人员在功能开发前，需要检出代码，开发人员可以查看本地是否有当前待开发任务的工程，若本地已有此工程则直接更新（拉取）代码。

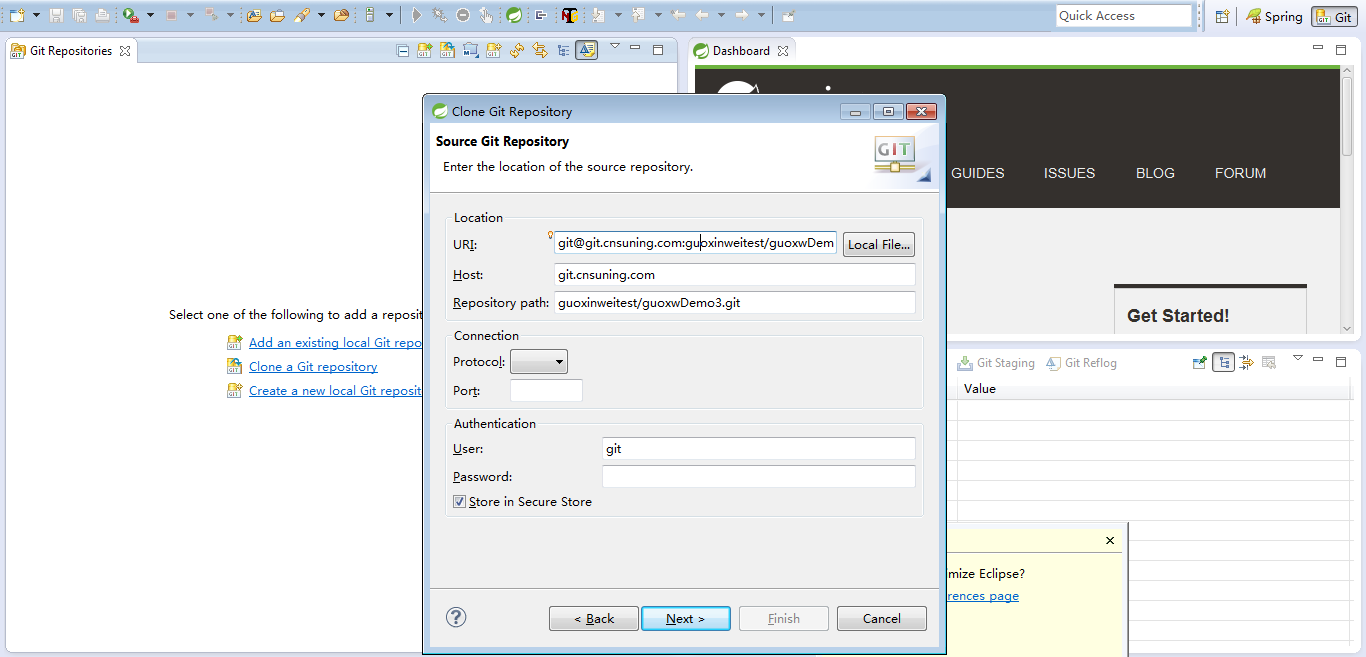
拉取服务器代码到本地：



若本地无当前开发任务的代码，需要Clone代码库。

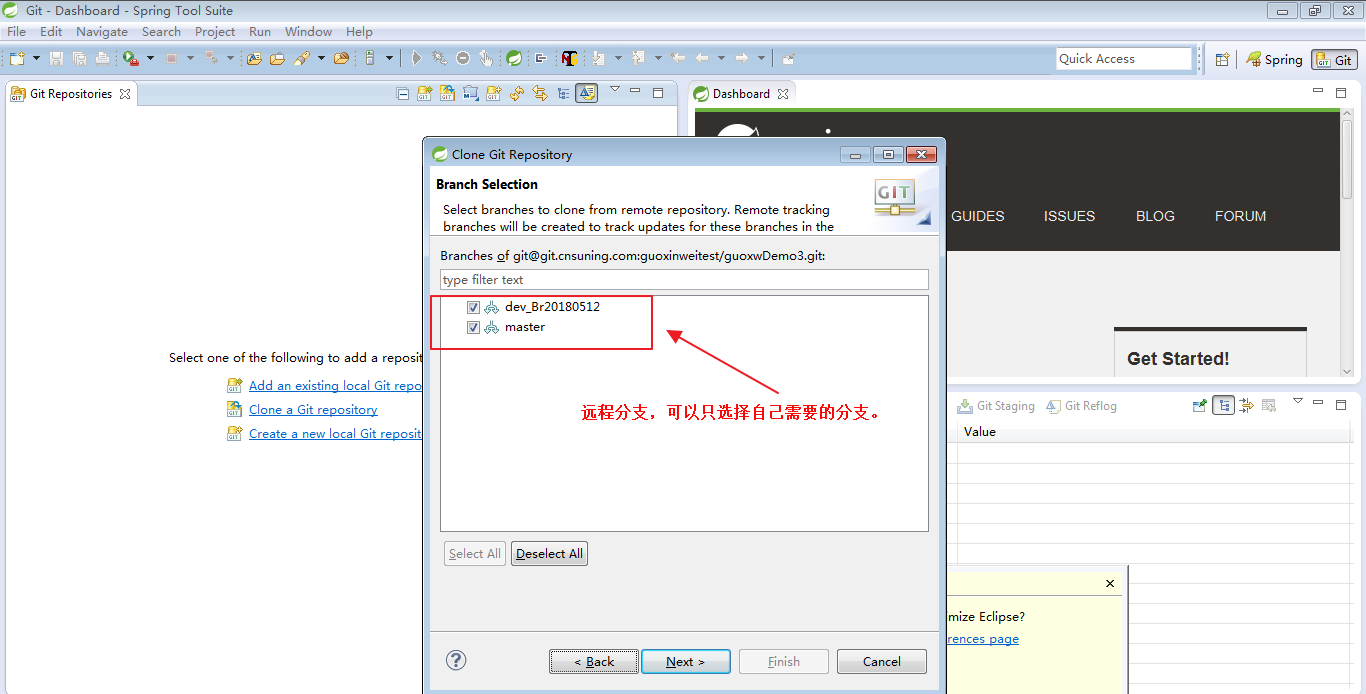


点击“clone代码库”后，会弹出提示框，输入库的URL

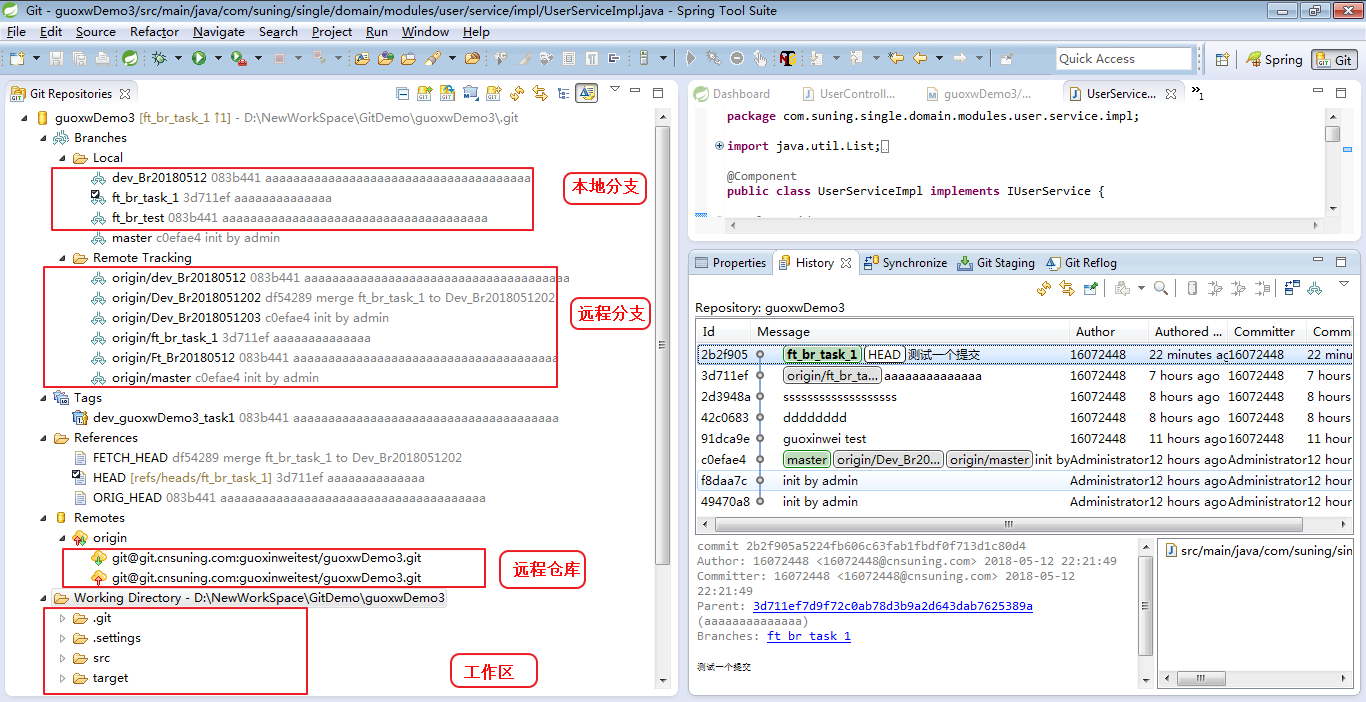


注：此处配置的git地址为ssh类型，ssh密钥配置参考下面章节《配置ssh密钥》。

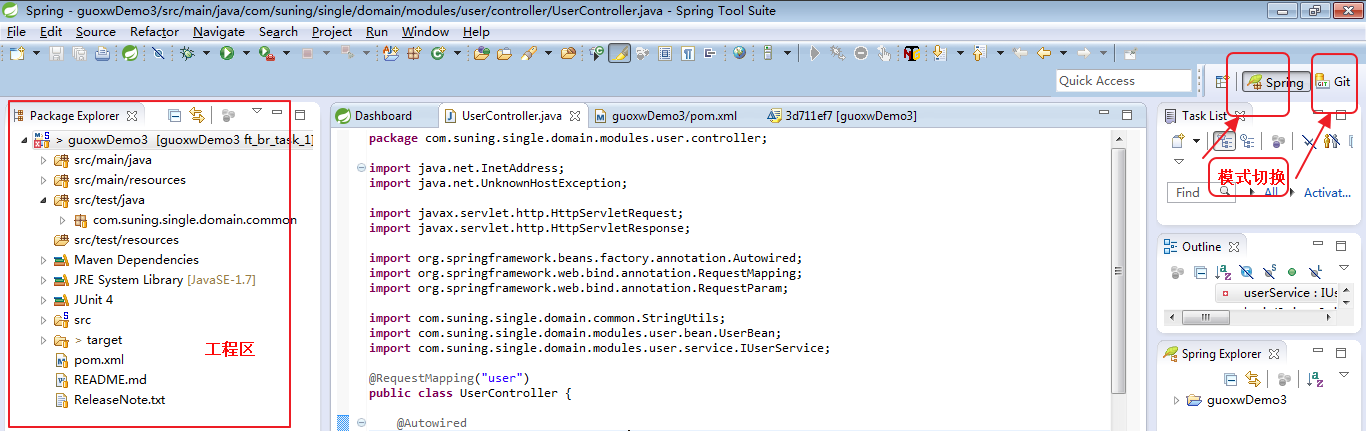
点击“下一步”选择自己需要下载的分支



点击 “完成”后，代码库会加载到工程内。



切换到开发视图：



以上是插件操作，亦可直通过bash命令操作，：

#下载代码库

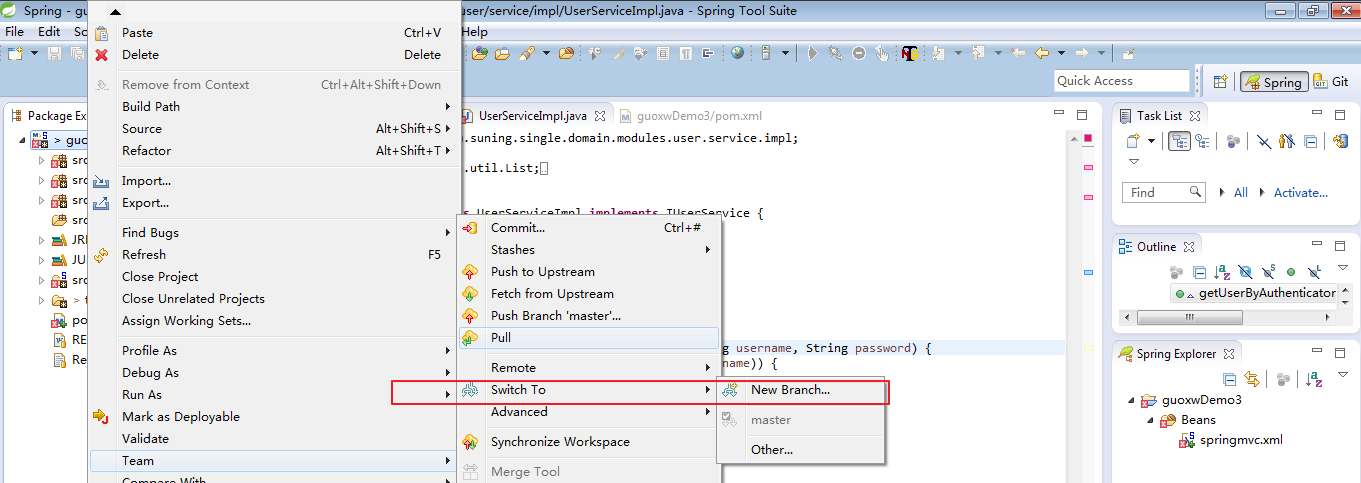
$ git clone [git@git.cnsuning.com:venus/cadence.git](mailto:git@git.cnsuning.com:venus/cadence.git)

#更新代码库

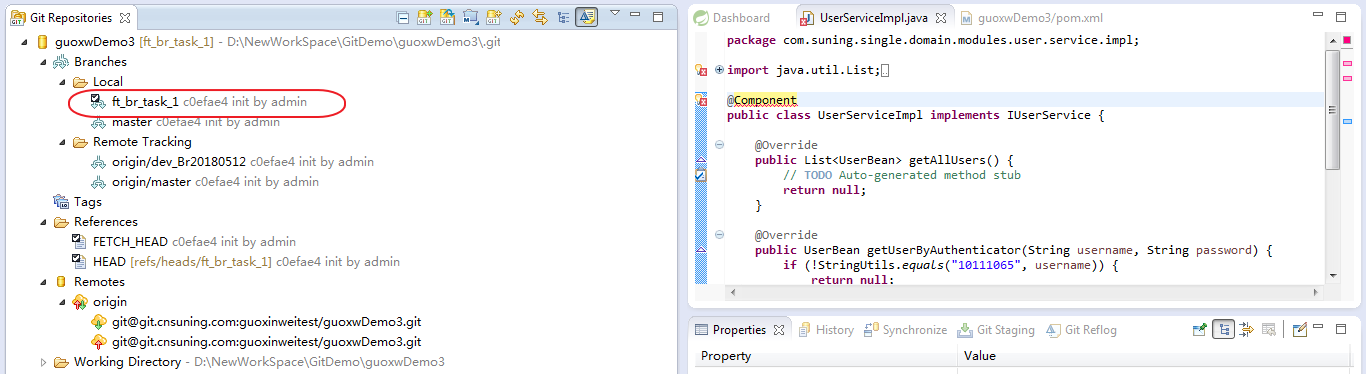
$ git pull

### 4.4.3、创建特性分支

当开发人员接到开发任务后，可根据开发的需要在本地创建一个特性分支进行任务的开发。建议一个开发任务创建一个特性分支，每个开发任务尽可能拆分的颗粒度较小，开发任务之间尽量减少依赖，每个开发任务可独立发布。



创建一个新分支 取名 ft\_br\_${需求编号}\_01



分支命名必须遵循命名规则，建议以ft\_br开头，后面跟开任务或需求编号。

以下是bash命令操作方式：

$ 创建一个特性分支

$ git branch ft\_br\_task\_1

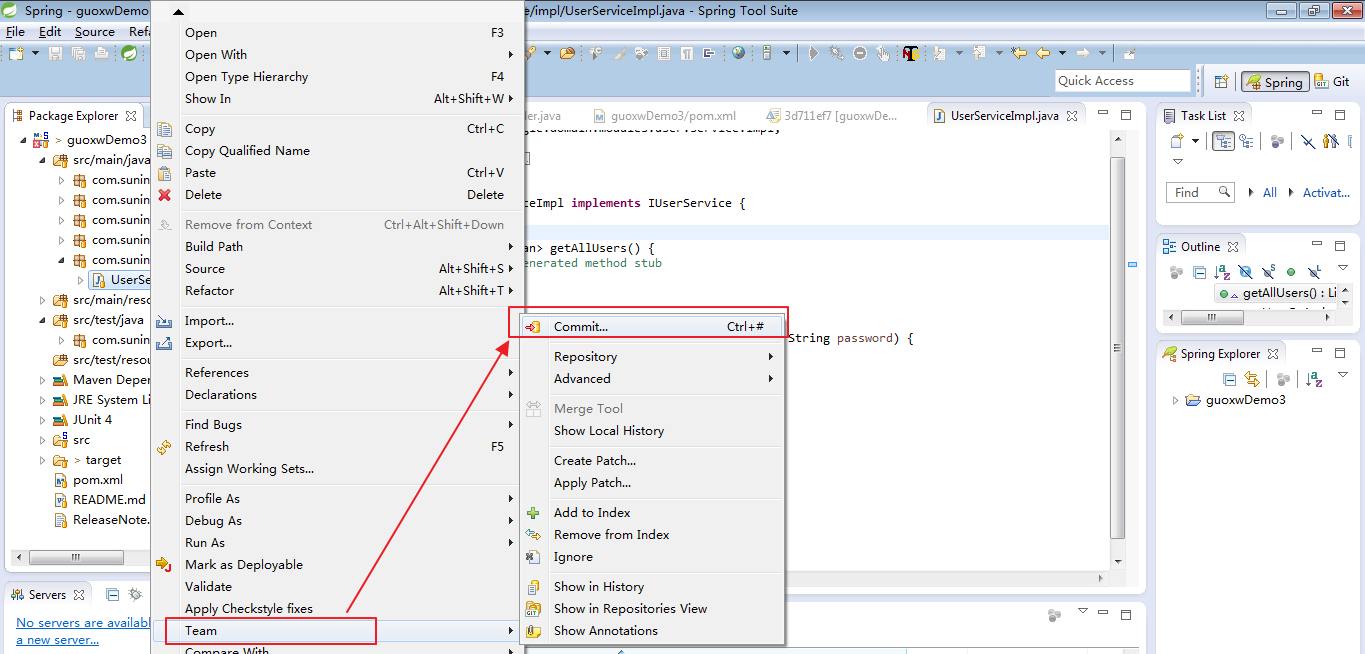
$ 切换到特性分支

$ git checkout ft\_br\_task\_1

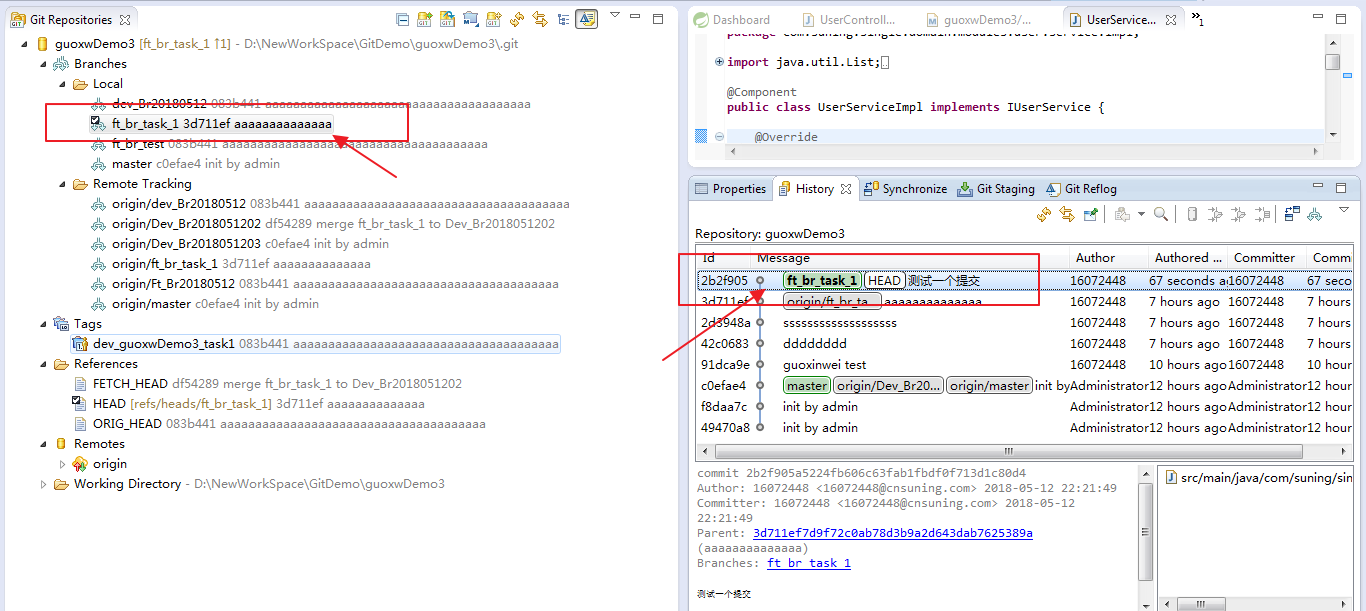
### 4.4.4、代码提交

开发人员功能开发完毕，可以直接保存在本地亦可直接推送本地分支ft\_br\_task\_1到远程。

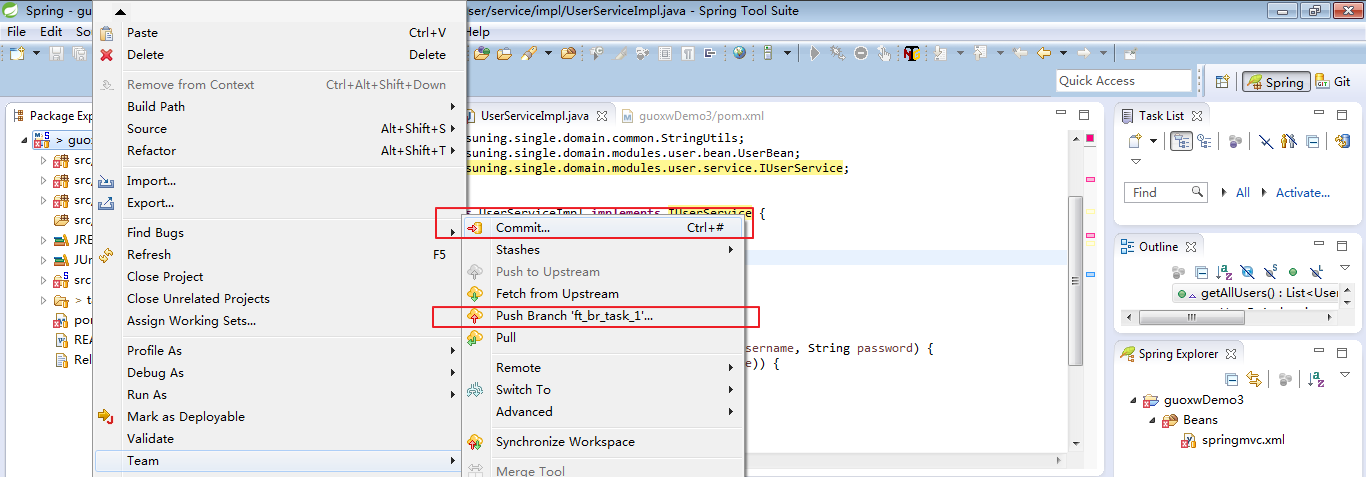
本地开发后可先保存代码到本地，在功能未全部开发完成或者功能未本地调试通过的情况下，代码在本地存储便于版本的比较和回退。



查询提交记录



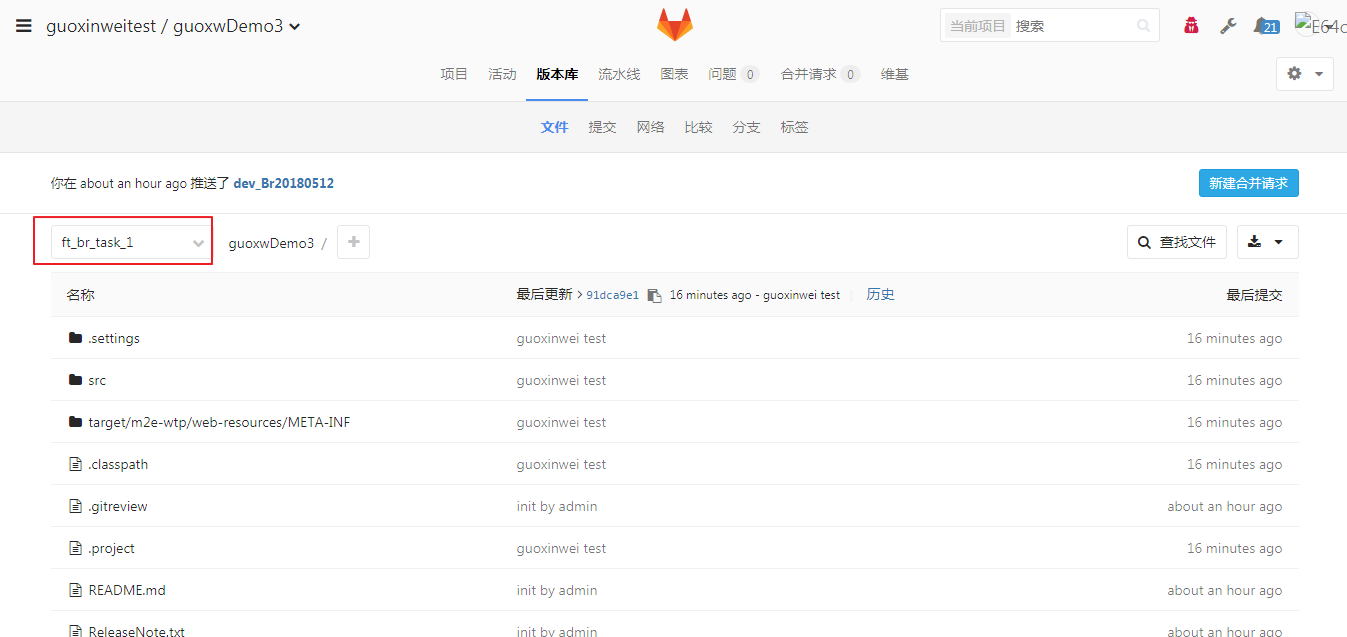
提交代码到远程分支（push），当某个功能开发完毕可提交到远程分支，并进行个人级构建。参考《4.4.5、个人级(特性分支)构建》章节。



在开发云平台可直接看到本地推送的分支



也可以登录Gitlab平台查询



通过bash命令行操作如下：

$ git add 文件

$ git commit xxxx.txt -m "gxwtest"

$ git push origin ft\_br\_task\_1

### 4.4.5、个人级(特性分支)构建

每个开发人员都要保证自己的特性分支（ft\_br\_task\_1）构建通过（代码质量检查，单元测试符合要求），不允许质量不达标的代码合并到开发分支。

入口：应用管理🡪持续集成🡪构建任务管理

创建构建任务：



Cron表达式用法：

|  |  |
| --- | --- |
| MINUTE | Minutes within the hour (0–59) |
| HOUR | The hour of the day (0–23) |
| DOM | The day of the month (1–31) |
| MONTH | The month (1–12) |
| DOW | The day of the week (0–7) where 0 and 7 are Sunday. |

Examples:

# every fifteen minutes (perhaps at :07, :22, :37, :52)

H/15 \* \* \* \*

# every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24)

H(0-29)/10 \* \* \* \*

# once every two hours every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)

H 9-16/2 \* \* 1-5

# once a day on the 1st and 15th of every month except December

H H 1,15 1-11 \*

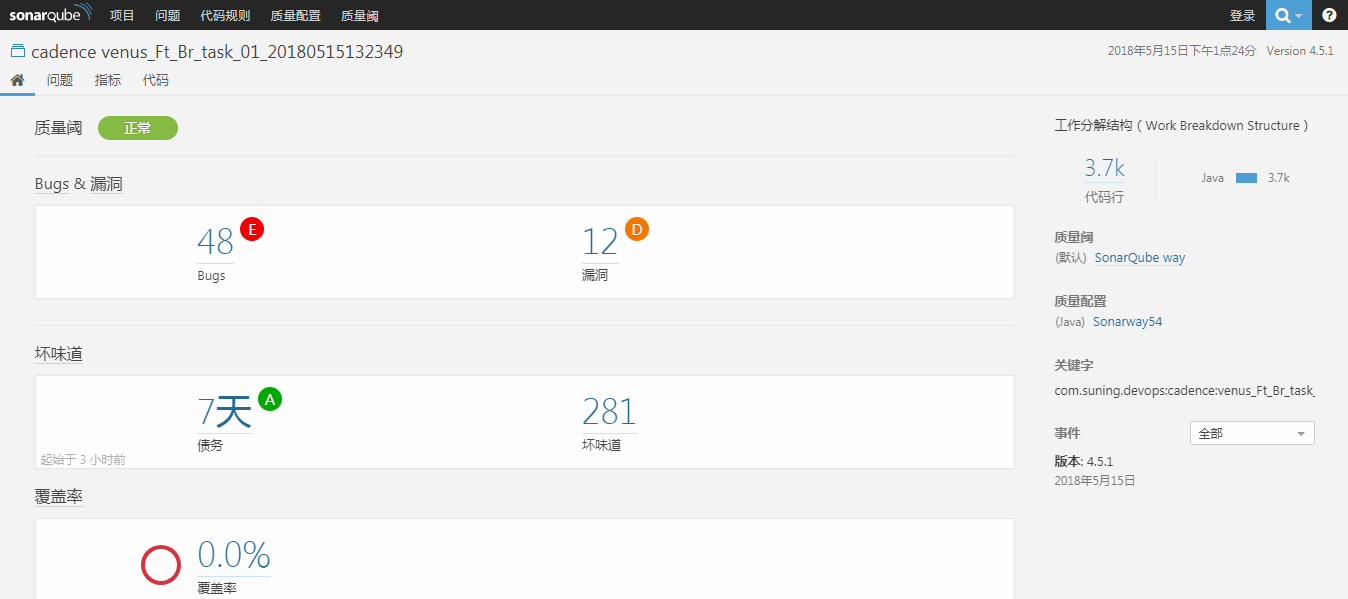
查询自己的构建任务：



点击“执行”后，可查询执行记录



点击“sonar分析”可以直接查询质量分析数据。



### 4.4.6、特性分支merge代码到开发分支

每个开发人员的特性分支（ft\_br\_task\_1）单测通过后，准备进入测试验证阶段时，需要合并到开发分支Dev\_Br20180512,合并代码须在开发云平台通过创建Pull Request请求来完成。若有冲突，参考《冲突解决》。

发起一个pull Request请求：



点击“确定” Pull request创建完成。



Pull request创建完成后处于代码评审状态。

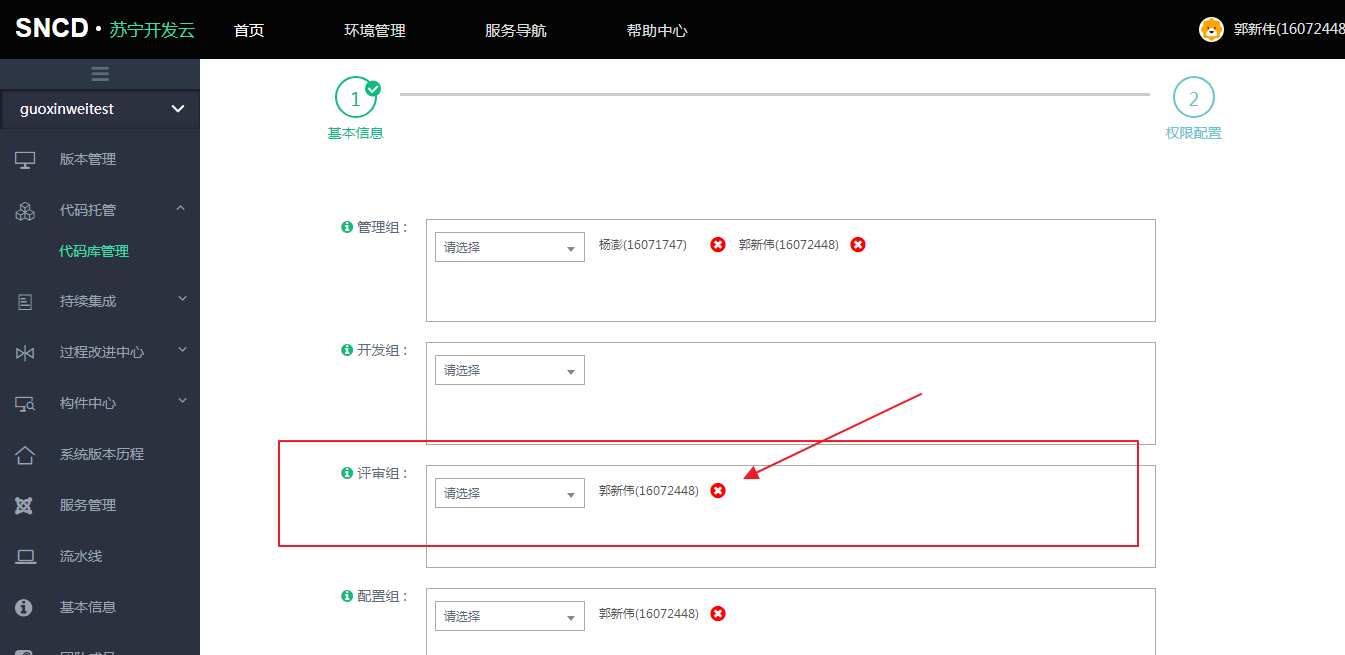


处于代码评审阶段时，“接收请求”置灰不可用，此时可以选择“关闭”来拒绝此次申请。

### 4.4.8、评审人员进行代码评审

若代码库使用了gerrit评审（若当前代码库未使用gerrit评审则直接跳过此步骤），则所有的合并请求必须经过评审步骤，否则无法合并。评审人必须在代码库的评审组里面，否则无评审权限。当开发人员提交合拼请求后需要及时通知评审人员进行代码评审，或者评审人员收到评审邮件后及时进行代码评审。

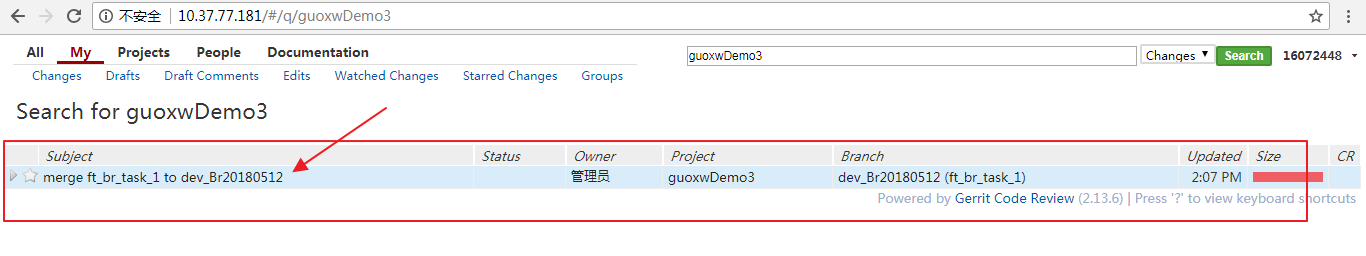
查询评审人员：



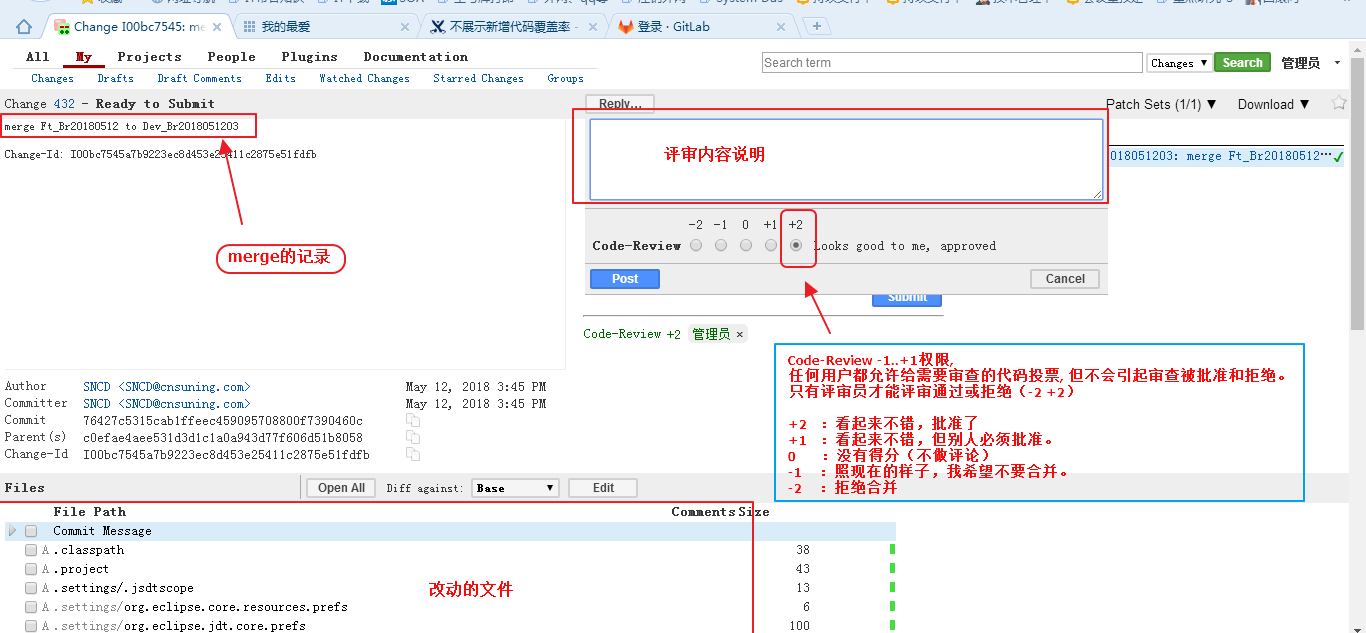
评审人登录gerrit



Gerrit地址：http://10.37.77.181/#/dashboard/self



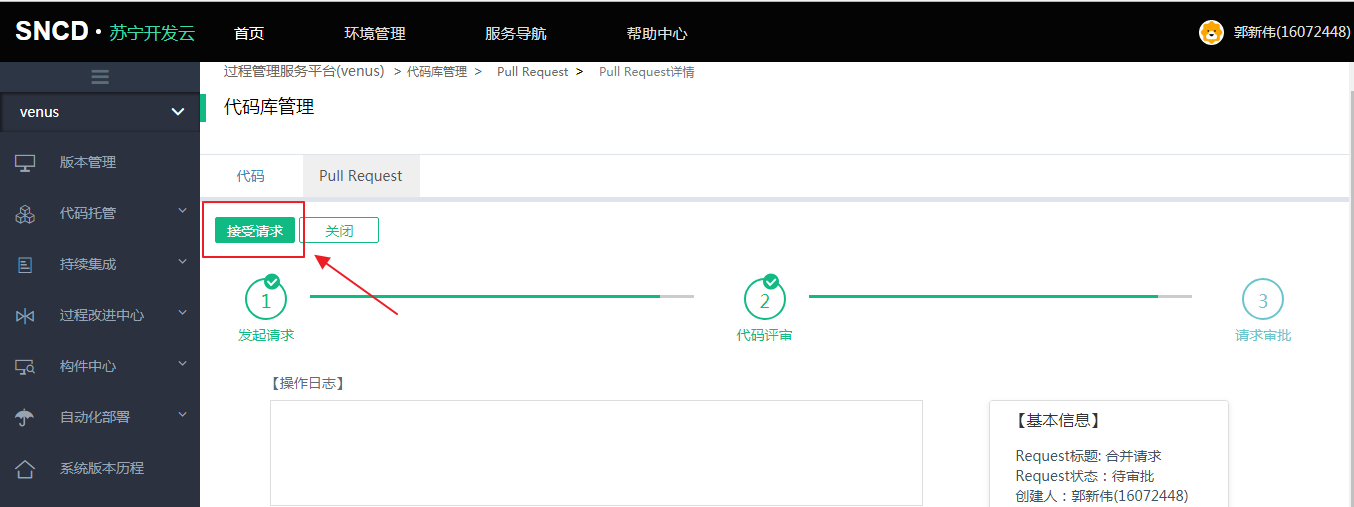
评审人进行代码评审：



### 4.4.9、接收合并请求

所有的合并请求必须经过版本经理接收才能最终合并到目标分支中。版本经理根据特性分支的持续构建情况（单测，规范）及代码评审情况及版本测试计划，来决定是否接收合并请求。

代码库的配置组及管理组成员有接收请求的权限。



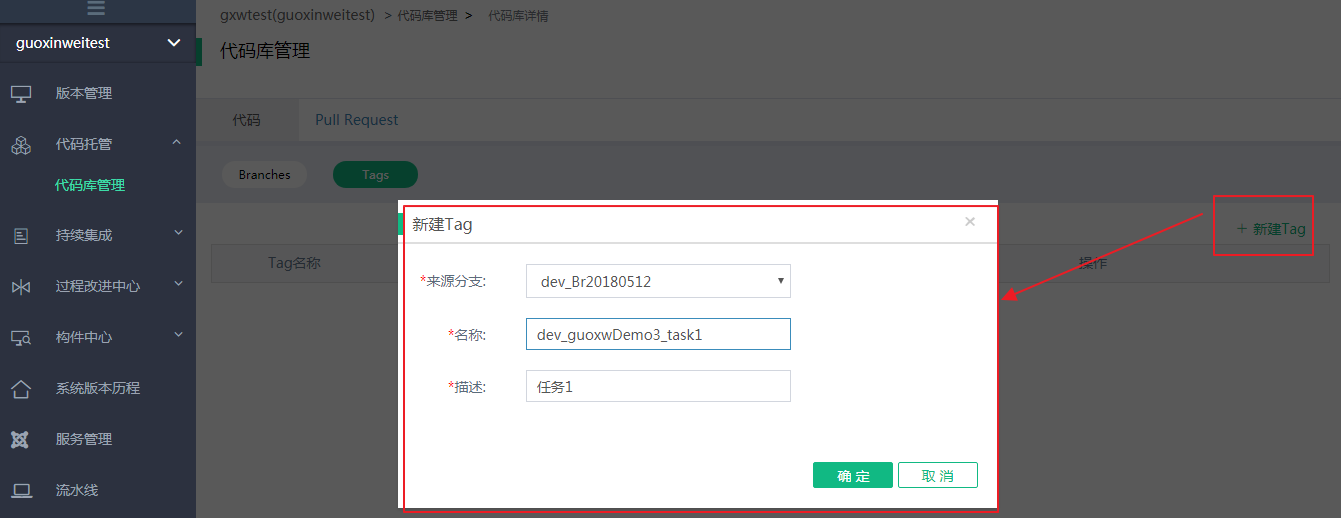
若拒绝请求，则直接关闭即可。



### 4.4.10、创建tag

开发人员在开发完成某个任务或者修改某个缺陷后，在将代码推送到远程后可基于最后提交的commitid创建一个tag，以便于测试人员根据tag来逐个的进行功能验证。测试人员在测试过程中亦可在测试的某个阶段打出一个tag来标识一个阶段的稳定版本。

tag的创建可直接在本地操作然后推送到远程，亦可在服务器直接创建。



注：每个tag用于标识某个任务的完成点或者用于标识某个缺陷的修改。

bash命令操作如下：

# 新建一个tag在指定commit

$ git tag [tag] [commit]

# 提交指定tag

$ git push [remote] [tag]

### 4.4.11、版本级（开发分支）构建

为保证开发分支总是处于可测试的状态及开发分支的质量总是处于质量管控的范围之内，版本经理可以将当前版本的开发分支（如：dev\_br20180512）设置成定时构建，以便于监控版本质量。构建配置参考《4.4.5个人级构建》。

### 4.4.12、版本测试

测试人员根据测试计划及代码合并情况来进行测试，测试时可直接对版本V1.0进行测试验证亦可对某个tag进行测试验证。

* **版本发布**

版本发布针对所有非容器环境进行发布。

菜单：应用管理🡪版本管理🡪操作🡪发布





* **容器发布**

容器发布主要针对docker环境进行发布

入口一：应用管理🡪版本管理🡪版本详情🡪配置库🡪操作🡪容器发布



【注意】：版本须关联了分支才能看到

入口二：应用管理🡪代码托管🡪代码库管理🡪进入代码库🡪分支列表🡪操作🡪容器发布



### 4.4.13、测试阶段缺陷处理

在测试验证阶段发现的缺陷开发人员可直接在特性分支（ft\_br\_task1）修改，亦可直接在开发分支拉取一个新的特性分支修改（此适合于pre测试阶段），针对pre阶段建议每个缺陷打一个tag以便于测试人员根据tag来逐个验证缺陷。

### 4.4.14、测试通过后进行封板

当测试人员在测试环境验证通过后，需要对版本进行封版操作，只有封版后才可进行prd的发布，封板后开发分支不能再接收任何的合并请求。



### 4.4.15、发布员进行生产发布

当测试通过并封板后，发布员可直接进行生产发布。

菜单：应用管理🡪版本管理🡪操作🡪发布





### 4.4.16、开发分支合并master

发布生产后若无问题需要将开发分支（Dev\_Br20180512）合并到master， 合并操作需要在开发云平台发起一个pullRequest请求，参考《4.4.6特性分支merge代码到开发分支》章节。

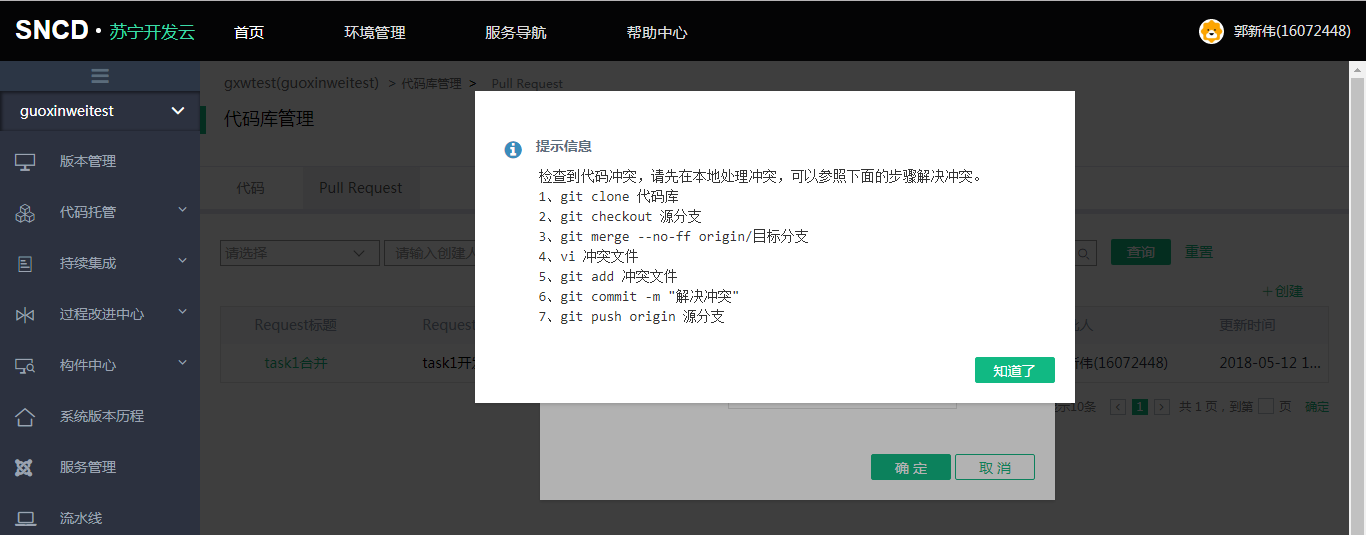
### 4.4.17、生产缺陷解决

若生产环境发生问题可以直接从master拉取一个紧急分支(hotfix)，并在紧急分支上拉取特性分支进行修改。若上线后第二天（非上线当晚，上线当前的缺陷直接在特性分支修改或者回滚代码）开发分支未合并到master时发生的生产问题可以直接在开发分支上拉取紧急分支(hotfix)。紧急分支的上线流程可参考开发分支的使用。

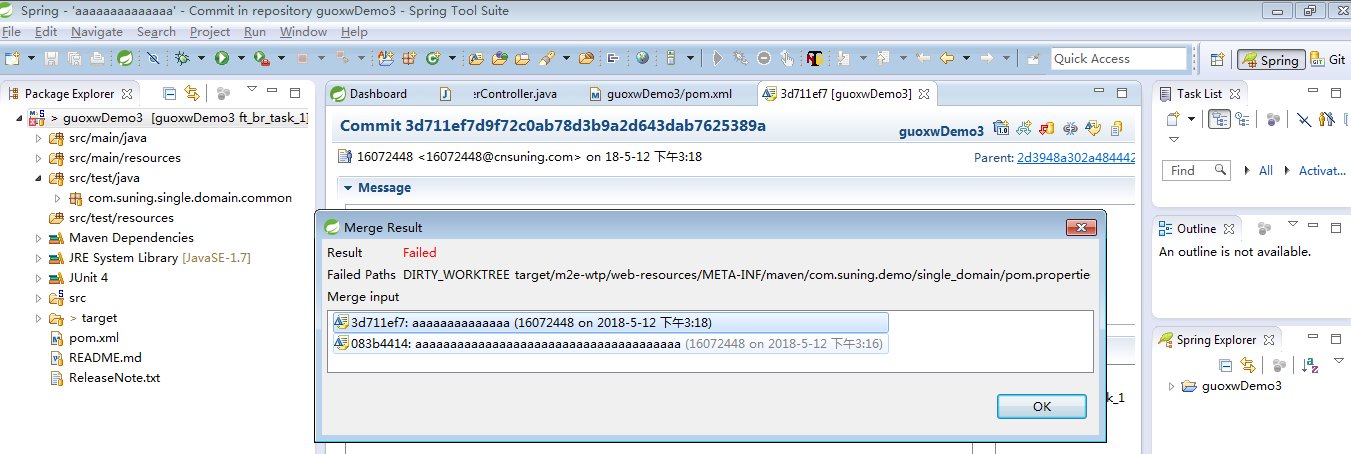
### 4.4.18、冲突解决

如A分支合并到B分支存在冲突，需要先将B分支push到本地，然后将B分支merge到A分支上，重新发起合并。

场景一、创建pullrequest请求出现冲突

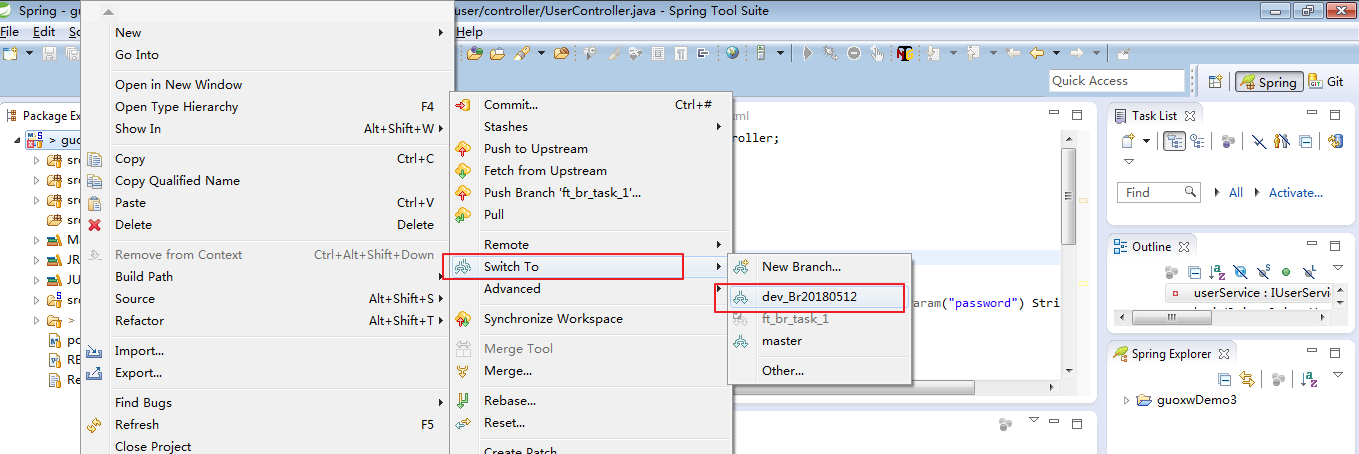


场景二、本地合并出现冲突

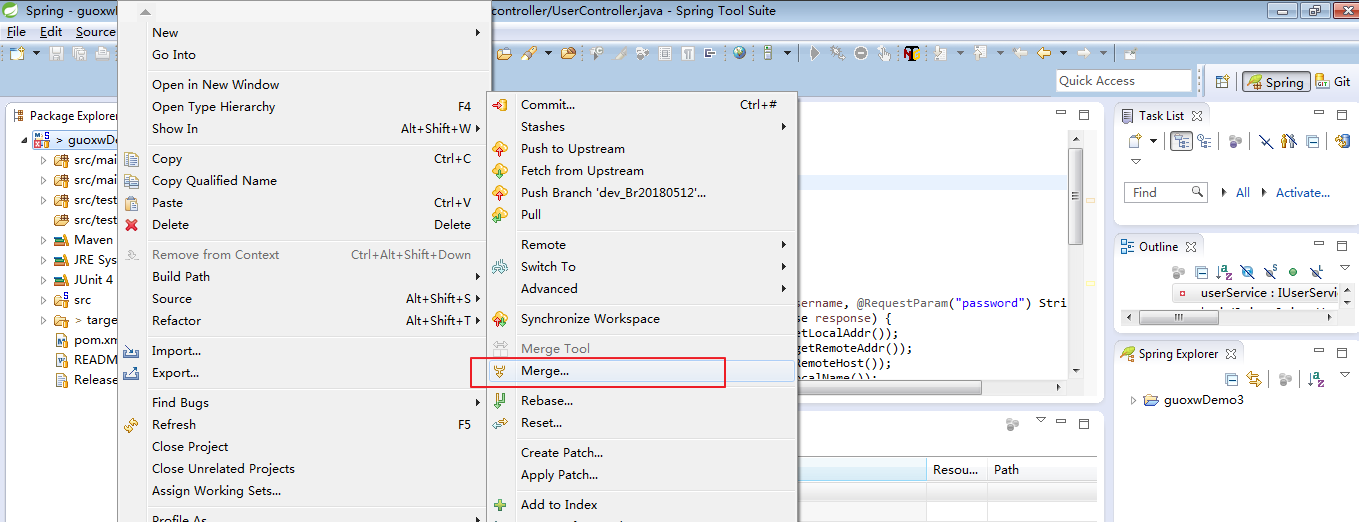


解决方案：

首先将开发分支更新到本地



其次将开发分支merge到特性分支



最后重新合并。

## 4.5、SVN迁移GIT

若某系统在开发云平台中已经存在SVN代码库，想转换成GIT库来试用，可以使用平台提供的一键迁移功能。SVN迁移GIT后原SVN下所有分支都会自动迁移过去，但是为了保证迁移的效率建议尽可能的减少SVN下分支的数量，即能合并到trunk的尽量合并，不用的分支全部废弃然后在做迁移 因为分支太多迁移会比较慢。

入口：<http://phoebus.cnsuning.com/>

菜单：应用管理-🡪代码托管-🡪代码库管理。

进入代码库管理：



点击“迁移git”



点击“确定”





点击底部的“保存”按钮后开始迁移。

迁移是个过程，根据分支数量不等耗时不同，需要耐心等待。

迁移完成后，SVN代码库会废弃，而新增一个git代码库。

# 5、配置SSH密钥

第一步、设置Git的user name和email：(如果是第一次的话)

*$ git config --global user.name "16072448"*

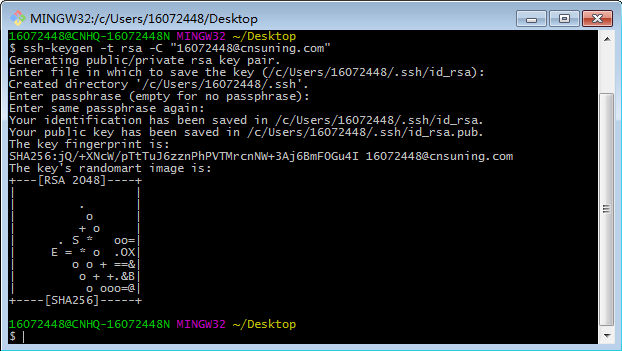
*$ git config --global user.email* [*16072448@cnsuning.com*](mailto:16072448@cnsuning.com)

第二步、创建 SSH Key

*$ ssh-keygen -t rsa -C* [*16072448@cnsuning.com*](mailto:16072448@cnsuning.com)

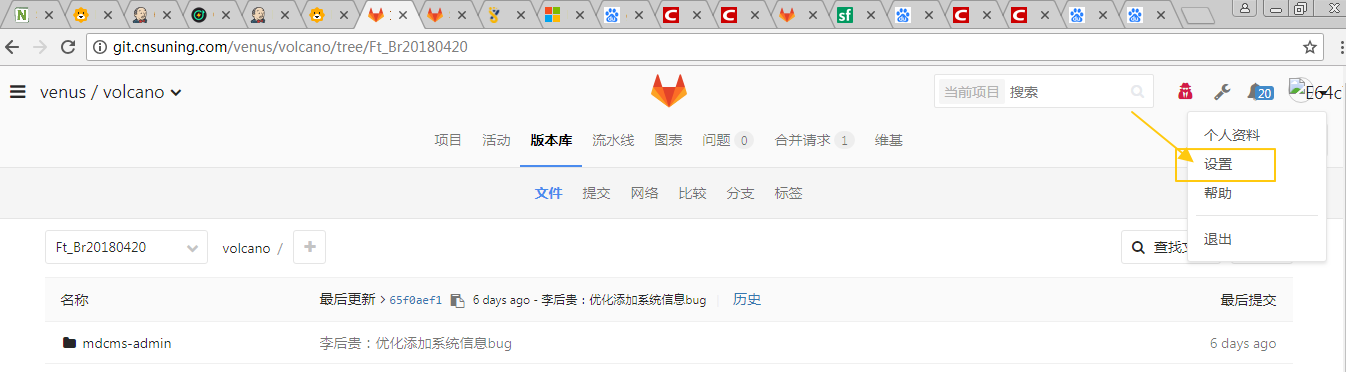
连续3个回车。

在C:\Users\16072448\.ssh 目录下最后得到了两个文件：id\_rsa和id\_rsa.pub。



第三步、在服务端配置公钥

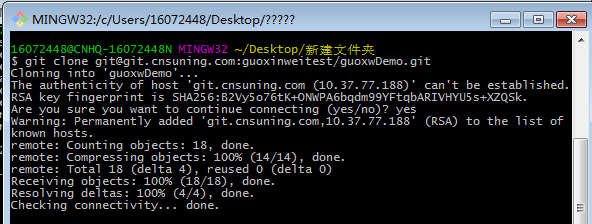
登录git：http://git.cnsuning.com



点击“设置”



第四步：验证是否配置成功



# 6、Git常用指令

$在当前目录新建一个Git代码库

$ git init

$新建一个目录，将其初始化为Git代码库

$ git init [project-name]

$下载一个项目和它的整个代码历史

$ git clone [url]

# 显示当前的Git配置

$ git config –list

# 编辑Git配置文件

$ git config -e [--global]

# 设置提交代码时的用户信息

$ git config [--global] user.name "[name]" $ git config [--global] user.email "[email address]"

# 添加指定文件到暂存区

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录

$ git add [dir]

# 添加当前目录的所有文件到暂存区

$ git add .

# 添加每个变化前，都会要求确认

# 对于同一个文件的多处变化，可以实现分次提交

$ git add -p

# 删除工作区文件，并且将这次删除放入暂存区

$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区

$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区

$ git mv [file-original] [file-renamed]

# 提交暂存区到仓库区

$ git commit -m [message]

# 提交暂存区的指定文件到仓库区

$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区 $ git commit -a

# 提交时显示所有diff信息

$ git commit -v

# 使用一次新的commit，替代上一次提交

# 如果代码没有任何新变化，则用来改写上一次commit的提交信息

$ git commit --amend -m [message]

# 重做上一次commit，并包括指定文件的新变化

$ git commit --amend [file1] [file2]

# 列出所有本地分支

$ git branch

# 列出所有远程分支

$ git branch -r

# 列出所有本地分支和远程分支

$ git branch -a

# 新建一个分支，但依然停留在当前分支

$ git branch [branch-name]

# 新建一个分支，并切换到该分支

$ git checkout -b [branch]

# 新建一个分支，指向指定commit

$ git branch [branch] [commit]

# 7、附录-GIT客户端工具介绍

工欲善其事，必先利其器。作为一名开发人员，你不可能不知道git，无论你是开发自己的开源项目还是和团队一起进行大规模产品的开发，git都已经是源代码管理工具的首选。下面推荐几款常用的客户端工具。

* Git-2.7.2-32-bit\_setup.exe

官网：<https://gitforwindows.org/>

安装包下载（内网）：

<ftp://10.37.77.182/GIT-Client/Git-2.7.2-32/Git-2.7.2-32-bit_setup.1457942412.exe>

账号/密码:匿名访问

* SourceTreeSetup-2.4.8.0.exe

官网：<http://sourcetree.software.informer.com/>

安装包下载(内网)：

<ftp://10.37.77.182/GIT-Client/SourceTree/SourceTreeSetup-2.4.8.0.exe>

账号/密码:匿名访问

* TortoiseGit-1.8.14.0\_64bit.msi

官网：<https://tortoisegit.org/>

安装包下载(内网)：

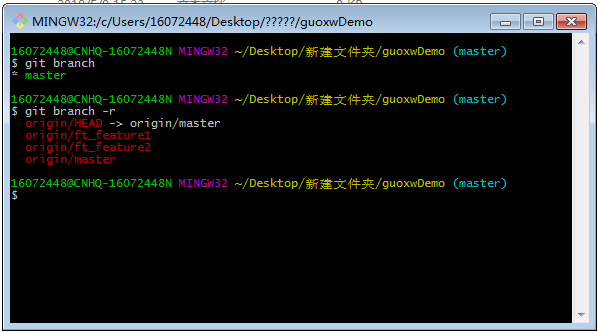
<ftp://10.37.77.182/GIT-Client/TortoiseGit/TortoiseGit-1.8.14.0_64bit.1436148947.msi>

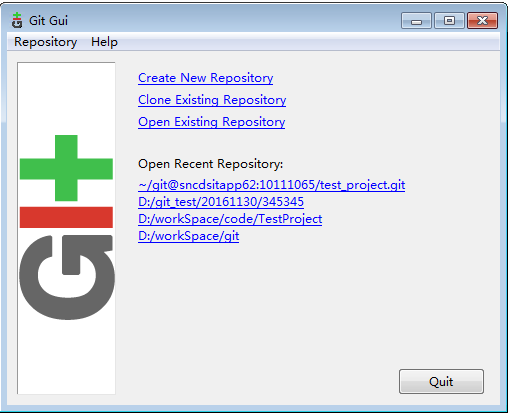
账号/密码:匿名访问

## 7.1、bash使用

包含 git gui 和 git Bash

简易可视化的命令行操作界面、轻巧、简单。可右键随时打开一个命令窗口。





下载地址：ftp://10.37.77.182/GIT-Client/Git-2.7.2-32/Git-2.7.2-32-bit\_setup.1457942412.exe

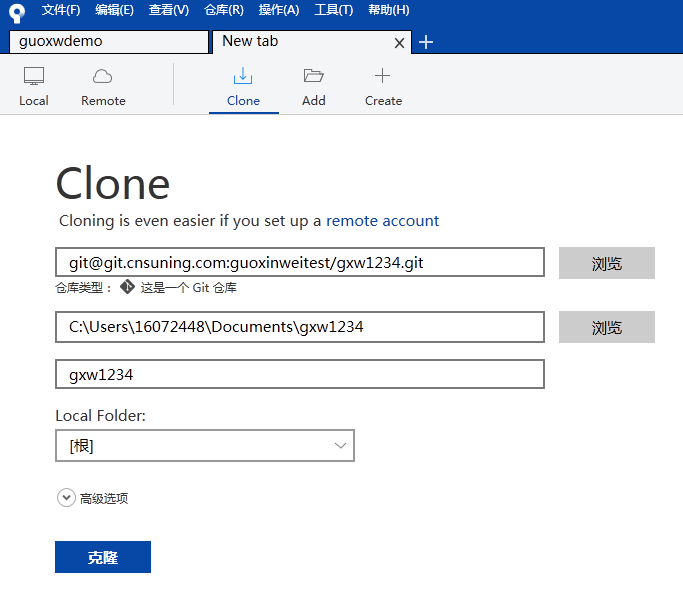
安装包版本：Git-2.7.2-32-bit.exe（根据操作系统位数选择版本）

## 7.2、可视化图形管理

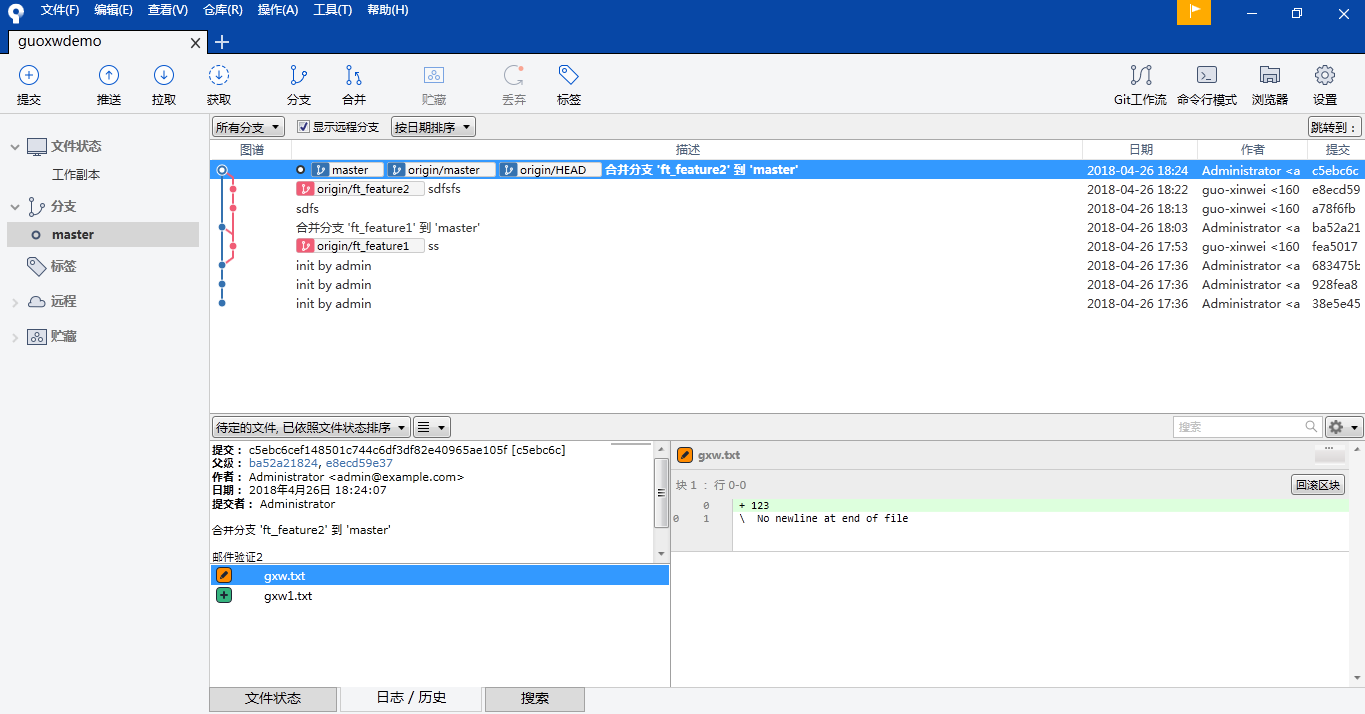
**1）SourceTree：**

SourceTree是老牌的Git GUI管理工具了，也号称是最好用的Git GUI工具。我的体验是确实强大，功能丰富，基本操作和高级操作都设计得非常流畅，适合初学者上手。这个工具很有特色的一个功能就是支持Git Flow，你可以一键创建Git Flow的工作流。Git Flow是非常高效的团队协作模型和流程，Git的一大特色就是灵活轻量的分支，但如何在自己的团队中用好这个功能来匹配自己的研发流程是个问题。内置Git Flow让那些不太熟悉的开发人员也可以很快上手，并且将研发的业务流程固化在工具中，可以说是非常贴心的设计。

* **特点：**
* 免费
* 功能强大：无论你是新手还是重度用户，SourceTree 都会让你觉得很顺手。
* 同时支持 Windows 和 Mac 操作系统
* **使用：**



（加载项目）



**（**操作界面**）**

* **安装：**

下载路径： ftp://10.37.77.182/GIT-Client/SourceTree/SourceTreeSetup-2.4.8.0.exe

安装包：SourceTreeSetup-2.4.8.0.exe（根据操作系统位数选择版本）

假设路径为（ C:\Users\16072448\AppData\Local\SourceTree）

安装过程：（依赖 .net Framework）

Framework安装包（若本机已经安装可以忽略）：ftp://10.37.77.182/GIT-Client/Net%20Framework/NDP461-KB3102438-Web.exe

第一步：



第二步：

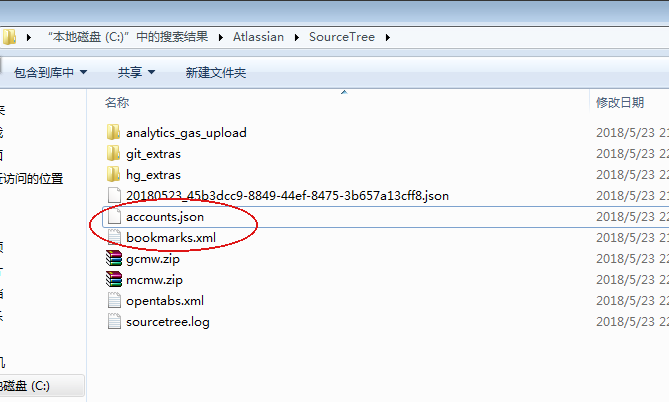


第三步：此时关闭安装窗口，进入 本地安装目录

C:\Users\11076994\AppData\Local\Atlassian\SourceTree

上面红色标注要替换成自己的账号 放入以下文件





第四步：此时本地桌面或者启动栏可以看到sourcetree已安装



点击启动。

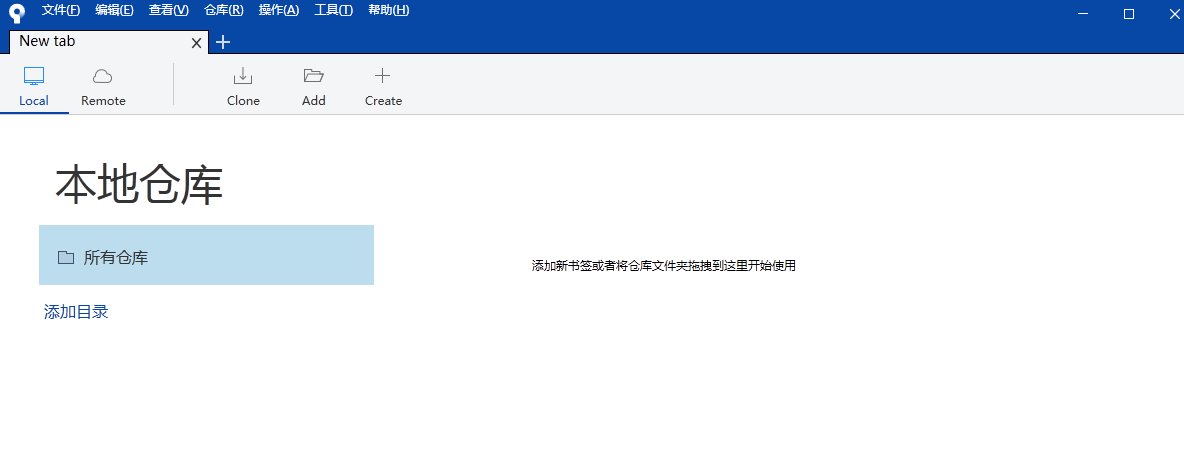
第五步：出现以下页面，选择 2 或者 4，若本地已经安装git则直接选择第二个。



第六步：选择“我不想使用Mercurial”



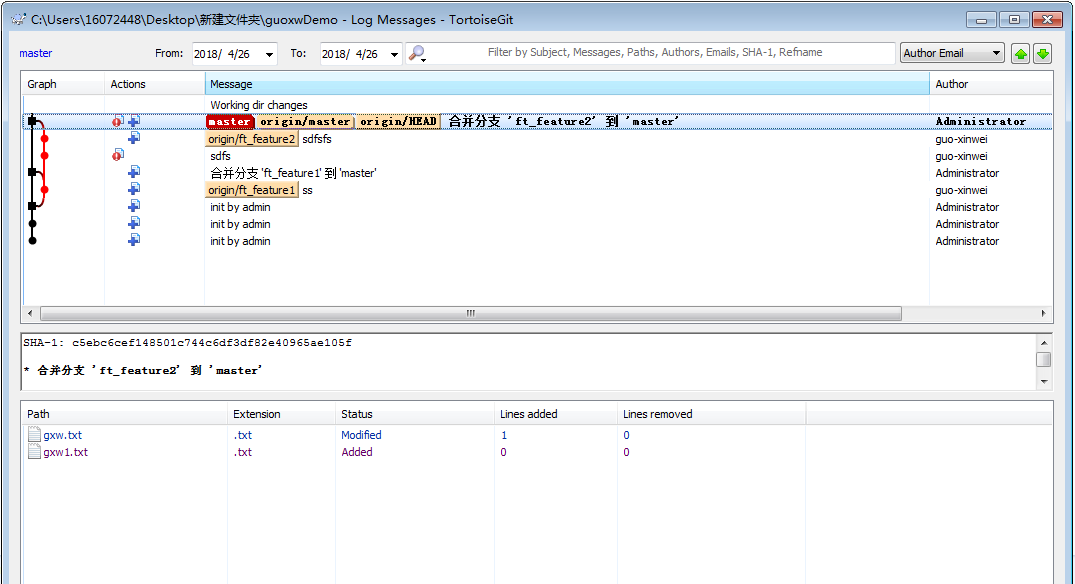
第七步：安装完成



**2）TortoiseGit：**

对这只小乌龟估计没有开发人员会不认识，SVN的超广泛使用也使得这个超好用的Svn客户端成了几乎每个开发人员的桌面必备软件。小乌龟的文件管理器右键菜单的操作方式对于新手来说非常的容易上手，而且容易理解。

* **特点：**
* 免费
* 只支持Windows操作系统：与文件管理器的良好集成
* 中文界面
* 与TortoiseSVN一脉相承的操作体验
* **使用：**





* **安装：**

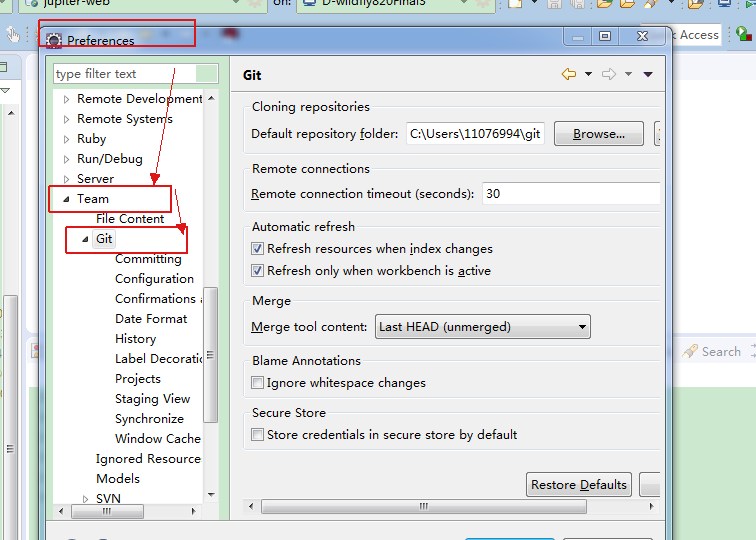
下载地址：ftp://10.37.77.182/GIT-Client/TortoiseGit/TortoiseGit-1.8.14.0\_64bit.1436148947.msi

安装包：TortoiseGit-1.8.14.0\_64bit.1436148947.msi（根据操作系统位数选择版本）

## 7.3、插件

* **安装**

若eclipse中已经安装了git插件的可跳过这一步。

第一步 查看：eclipse—window-preferences-team-git  


如果无，则执行如下步骤进行安装。

插件下载地址:http://download.eclipse.org/egit/updates/

全部勾选，然后一路next

Next-安装-重启eclipse。

第二步：安装好后，配置账号。

window--Preferences--Team--Git—Configuration

点击Add Entry,添加账号和邮箱



可以参考官网:http://wiki.eclipse.org/EGit/User\_Guide

* **使用**

参考文档  ****