|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **兆日科技Git操作指南**  **(仅供内部使用)**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **项目名称** |  | | | | | | | **子项目名称** |  | | | | | | | **产品名称** |  | | **产品代码** | |  | | | **软\硬件版本** |  | | | | | | | **文档编号** |  | **密级** | | **机密** | **页数** |  |     **作 者： 赵 昭 　　 日期： 2019年 4 月 30 日**  **审 核： 　　 日期： 年 月 日**  **标准化： 　　 日期： 年 月 日**  **批 准： 　　 日期： 年 月 日**    **版权所有 不得复制** |

**修订历史：**

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **修改描述** | **修改人** |
| 2019.4.30 | 1.0 | 完成初版 | 赵昭 |
| 2019.5.14 | 1.1 | 1、2.4.1章节完善开发人员修改指南，  2、2.4.1.5章节中增加rebase的使用  3、2.4.2章节增加TortoiseGit 使用 | 赵昭 |
| 2019.9.4 | 1.2 | 1、增加2.8 章节源代码目录图  2、增加2.4.6 章节开发人员(IDEA) | 赵昭 |
| 2019.9.18 | 1.3 | 1、增加2.1.1以及2.1.2中Git权限审批以及仓库、group创建  2、增加2.2.4 项目研发以及发布流程  3、增加2.9 银行代码同步方法  4、增加2.10 Git日志编写规范 | 赵昭 |
| 2019.10.25 | 1.4 | 1、增加git tag中版本号定义  2、增加 git submodule 强制更新远端命令  3、新增2.10 银行项目在Git中的创建 | 赵昭 |
| 2019.11.1 | 1.5 | 1、增加2.3.5 gitlab上仓库的手动fork  2、增加2.2.4.3 tortoiseGit 配置ssh | 赵昭 |

**目录**

[第一章 Git简介 1](#_Toc24641239)

[1.1 Git简介 1](#_Toc24641240)

[1.2 Git分支 1](#_Toc24641241)

[1.2.1 创建分支 1](#_Toc24641242)

[1.2.2 分支的合并和回滚 1](#_Toc24641243)

[1.2.2.1 分支的合并 1](#_Toc24641244)

[1.2.2.2 分支回滚 2](#_Toc24641245)

[1.3 Git工作流 3](#_Toc24641246)

[1.3.1 集中式工作流 3](#_Toc24641247)

[1.3.2 功能分支工作流 4](#_Toc24641248)

[1.3.3 GitFlow 6](#_Toc24641249)

[1.3.4 GitHub Flow 6](#_Toc24641250)

[1.3.5 GitLab Flow 7](#_Toc24641251)

[1.3.6 Forking工作流 8](#_Toc24641252)

[第二章 兆日科技Git规范 10](#_Toc24641253)

[2.1 角色和权限 10](#_Toc24641254)

[2.1.1 Gitlab中的权限审批 11](#_Toc24641255)

[2.1.2 Gitlab中group的创建以及仓库的创建 11](#_Toc24641256)

[2.2 工作流 11](#_Toc24641257)

[2.2.1 GitFlow工作流 11](#_Toc24641258)

[2.2.2 GitFlow分支 13](#_Toc24641259)

[2.2.2.1 工作方式 13](#_Toc24641260)

[2.2.2.2 历史（master、develop）分支 13](#_Toc24641261)

[2.2.2.3 功能（feature）分支 14](#_Toc24641262)

[2.2.2.4 发布（release）分支 15](#_Toc24641263)

[2.2.2.5 维护（hotfix）分支 16](#_Toc24641264)

[2.2.3 GitFlow基本操作流程 17](#_Toc24641265)

[2.2.3.1 创建开发分支 17](#_Toc24641266)

[2.2.3.2 开发者开始开发新功能 18](#_Toc24641267)

[2.2.3.3 开发者完成功能开发 19](#_Toc24641268)

[2.2.3.4 开发者开始准备发布版本 20](#_Toc24641269)

[2.2.3.5 开发者完成发布 20](#_Toc24641270)

[2.2.3.6 最终用户发现Bug 21](#_Toc24641271)

[2.2.4 项目研发以及发布流程 22](#_Toc24641272)

[2.2.4.1 Git项目研发流程 22](#_Toc24641273)

[2.2.4.2 研发分支与版本部署环境的关系 24](#_Toc24641274)

[2.3 项目组织 26](#_Toc24641275)

[2.3.1 Git Submodule 26](#_Toc24641276)

[2.3.2 Git SubTree 27](#_Toc24641277)

[2.3.3 二进制大文件的处理 29](#_Toc24641278)

[2.3.4 项目从svn迁移到Git 29](#_Toc24641279)

[2.3.5 仓库的fork 30](#_Toc24641280)

[2.4 分角色操作指南 31](#_Toc24641281)

[2.4.1 开发人员（命令行操作） 31](#_Toc24641282)

[2.4.1.1 准备工作 31](#_Toc24641283)

[2.4.1.2 初始化 31](#_Toc24641284)

[2.4.1.3 连接到远程仓库 34](#_Toc24641285)

[2.4.1.4 新建功能分支 34](#_Toc24641286)

[2.4.1.5 更新远端代码到本地 35](#_Toc24641287)

[2.4.1.6 在分支上开发 35](#_Toc24641288)

[2.4.1.7 分支合并 37](#_Toc24641289)

[2.4.1.8 删除本地以及远端分支 38](#_Toc24641290)

[2.4.1.9 发布测试版本 38](#_Toc24641291)

[2.4.2 开发人员（TortoiseGit） 39](#_Toc24641292)

[2.4.2.1 下载地址 39](#_Toc24641293)

[2.4.2.2 账户以及Git配置 39](#_Toc24641294)

[2.4.2.3 gitbash中已经配置过ssh场景(推荐使用该方法) 40](#_Toc24641295)

[2.4.2.4 使用TortoiseGit 生成ssh密钥对(不推荐使用) 41](#_Toc24641296)

[2.4.2.5 示例 44](#_Toc24641297)

[2.4.3 项目管理人员（项目经理） 49](#_Toc24641298)

[2.4.3.1 分支管理 50](#_Toc24641299)

[2.4.3.2 发布版本完成后的处理 50](#_Toc24641300)

[2.4.3.3 为master分支打tag 50](#_Toc24641301)

[2.4.3.4 线上紧急bug的处理 50](#_Toc24641302)

[2.4.3.5 Code Review 51](#_Toc24641303)

[2.4.3.6 cherry pick 53](#_Toc24641304)

[2.4.3.7 GitLab分支的设置 53](#_Toc24641305)

[2.4.4 技术管理人员（技管部） 54](#_Toc24641306)

[2.4.4.1 创建项目 54](#_Toc24641307)

[2.4.4.2 添加项目成员并赋予权限 55](#_Toc24641308)

[2.4.5 运维人员 56](#_Toc24641309)

[2.4.2 开发人员（IDEA） 56](#_Toc24641310)

[2.5 Git中大文件的处理 58](#_Toc24641311)

[2.5.1 查看项目中特定分支 58](#_Toc24641312)

[2.5.2 处理巨大二进制资产的库 59](#_Toc24641313)

[2.5.2.1 Git gc 59](#_Toc24641314)

[2.5.2.2 使用子模块 59](#_Toc24641315)

[2.5.3 Git-LFS 59](#_Toc24641316)

[2.5.4 稀疏检出 60](#_Toc24641317)

[2.6 Git中的常用工具 60](#_Toc24641318)

[2.6.1 Git Bash 60](#_Toc24641319)

[2.6.2 Git中的gui工具 60](#_Toc24641320)

[2.6.3 Git与IDE工具 60](#_Toc24641321)

[2.7 项目中Git的常用配置 61](#_Toc24641322)

[2.7.1 gitignore 61](#_Toc24641323)

[2.7.2 git hook 61](#_Toc24641324)

[2.8 仓库源代码目录图 62](#_Toc24641325)

[2.9 Git日志编写规范 63](#_Toc24641326)

[2.9.1 Git 中日志的作用 63](#_Toc24641327)

[2.9.2 Commit message的格式 63](#_Toc24641328)

[2.9.2.1 标题 63](#_Toc24641329)

[2.9.2.2 提交描述 64](#_Toc24641330)

[2.9.2.3 提交编号 64](#_Toc24641331)

[2.9.2.4 撤销提交 65](#_Toc24641332)

[2.9.3 日志举例 65](#_Toc24641333)

[2.10 银行产品在Git中操作 65](#_Toc24641334)

[2.10.1 银行产品环境准备 65](#_Toc24641335)

[2.10.1.1 脚本环境初始化 66](#_Toc24641336)

[2.10.1.2 从arsenal产品中fork出银行产品代码 68](#_Toc24641337)

[2.10.1.3 银行代码自动打包、部署、测试 69](#_Toc24641338)

[2.10.1.4 银行代码项目外带 69](#_Toc24641339)

[2.10.1.5 产品代码上传到银行仓库 69](#_Toc24641340)

[2.10.1.6 银行环境验证 70](#_Toc24641341)

[2.10.2 银行产品研发场景 70](#_Toc24641342)

[2.10.2.1 银行现场研发场景 70](#_Toc24641343)

[2.10.2.2 银行现场代码如何与兆日公司Git仓库同步 71](#_Toc24641344)

# 第一章 Git简介

## 1.1 Git简介

Git是一个免费并且开源的分布式版本控制系统。关于Git的资料，网上有很多，本文不做具体介绍，这里推荐几个链接：

[Git系列教程： https://blog.csdn.net/qq\_32452623/article/details/78276451](https://blog.csdn.net/qq_32452623/article/details/78276451)

[Git工作流指南：http://blog.jobbole.com/76843/](http://blog.jobbole.com/76843/)

[Git在团队中的最佳实践--如何正确使用Git Flow：https://www.cnblogs.com/wish123/p/9785101.html](file:///C:\Users\zhaozhao\Desktop\Git规范\Git在团队中的最佳实践--如何正确使用Git%20Flow：https:\www.cnblogs.com\wish123\p\9785101.html)

[pro git电子书（中文版）：https://git-scm.com/book/zh/v2](https://git-scm.com/book/zh/v2)

## 1.2 Git分支

在Git中使用分支的成本很低，Git鼓励在工作流程中频繁使用分支与合并。本节不对分支的概念做详细的介绍，只介绍Git分支的合并以及Git工作流。

### 1.2.1 创建分支

创建分支的命令：git branch <branch name>

切换分支的命令：git checkout <branch name>

创建并切换分支的命令：git checkout -b <branch name>

### 1.2.2 分支的合并和回滚

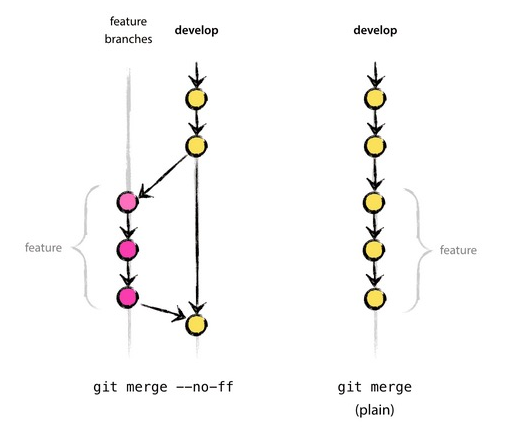
#### 1.2.2.1 分支的合并

合并分支的命令：

git merge <branch name> // Fast forward合并

git merge –-no-ff <branch name> // 普通合并

使用git merge --no-ff可以保存之前的分支历史。能够更好的查看merge历史，以及branch状态。git merge 则不会显示 feature，只保留单条分支记录。对比如下图所示：



#### 1.2.2.2 分支回滚

**本地分支回滚：**

确定要回滚到哪个提交（commit），找到该提交的 commit id，执行以下命令即可：

git reset --hard <commitId>

**远程分支回滚**

远程分支不存在回滚的概念，要想达到回滚的效果，就是删除之前的远程分支，然后把本地回滚好的本地分支，push 到远程。

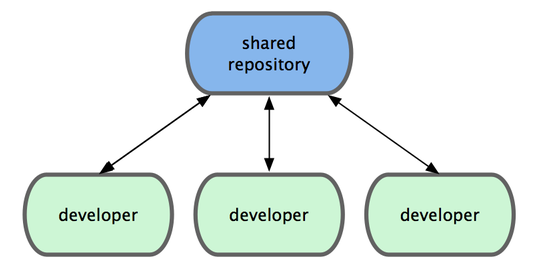
分支回滚是一个危险操作，在回滚之前建议先创建一个备份分支，可以避免很多出错，参考链接：https://blog.csdn.net/qq\_32452623/article/details/79521461

## 1.3 Git工作流

### 1.3.1 集中式工作流

在集中式工作流中，将Git看做是Subversion，用Subversion的方式来管理项目。

但使用Git比SVN有几个优势。首先，每个开发可以有属于自己的整个工程的本地拷贝。隔离的环境让各个开发者的工作和项目的其他部分（修改）独立开来 —— 即自由地提交到自己的本地仓库，先完全忽略上游的开发，直到方便的时候再把修改反馈上去。

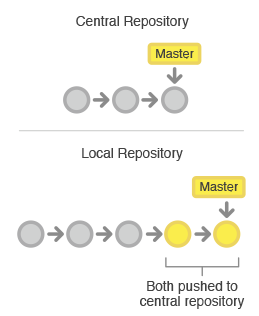


**工作方式**

像Subversion一样，集中式工作流以中央仓库作为项目所有修改的单点实体。相比SVN缺省的开发分支trunk，Git叫做master，所有修改提交到这个分支上。集中式工作流只用到master这一个分支。

开发者开始先克隆中央仓库。在自己的项目拷贝中，像SVN一样的编辑文件和提交修改；但修改是存在本地的，和中央仓库是完全隔离的。

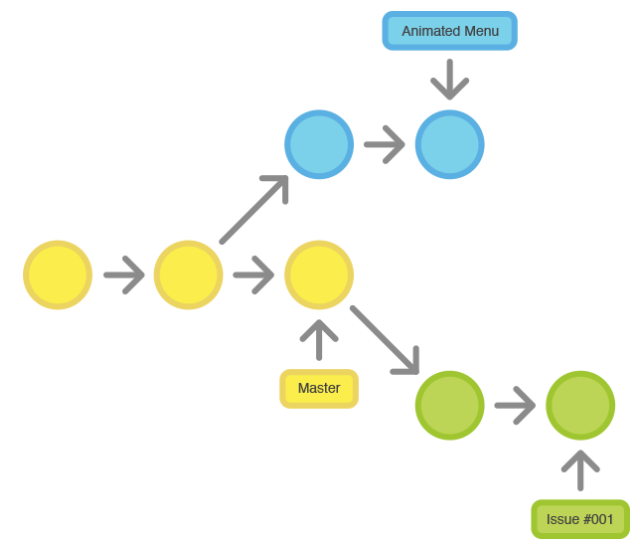
要将本地修改提交到正式项目中，开发者需要把本地master分支的修改“推（push）”到中央仓库中，这相当于svn commit操作。



### 1.3.2 功能分支工作流

功能分支（feature branch）工作流背后的核心思路是，所有的功能开发在各自专门的分支上进行，而不是在master分支上。这个隔离可以方便多个开发者在各自的功能分支上开发而不会弄乱主干代码。

功能开发隔离带来的另外一个好处就是Pull Requests机制。通过Pull Request可以为每个分支发起一个讨论，在分支合入master之前，其它开发者可以对代码进行审查，来决定这次修改是“接纳”还是“拒绝”。同时，如果你在功能开发中有问题卡住了，也可以提一个Pull Request来向同学们征求建议。



**工作方式**

功能分支工作流仍然使用中央仓库，并且master分支还是代表了正式项目的历史。但不是直接提交本地历史到各自的本地master分支，而是开发者每次在开始新功能前先创建一个新分支。功能分支应该有个有描述性的名字，比如animated-menu-items或issue-#1061，这样可以让分支有个清楚且高聚焦的用途。

在master分支和功能分支之间，Git没有技术上的区别，所以开发者可以用和集中式工作流中完全一样的方式编辑、暂存和提交修改到功能分支上。

功能分支可以push到中央仓库中。这样不修改正式代码（master分支）就可以和其它开发者分享提交的功能。在中央仓库上存多个功能分支不会有任何问题。当然，这样做也可以很方便地备份各自的本地提交。

**Pull Requests**

一旦某个开发人员完成一个功能，不是立即合并到master，而是首先将本地功能分支push到中央仓库的功能分支上，然后发起一个Pull Request请求去合并修改到master。在修改成为主干代码前，这让其它的开发者有机会先去Review变更，如Code Review。

一旦Pull Request被接受了，发布功能要做的就和集中式工作流就很像了。首先，确定本地的master分支和上游的master分支是同步的。然后合并功能分支到本地master分支并push已经更新的本地master分支到中央仓库。

### 1.3.3 GitFlow

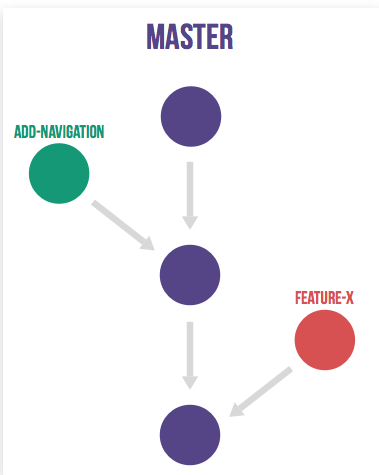
对于大型团队，常常需要给不同分支分配一个更具体的角色。Gitflow工作流是管理功能开发、发布准备和维护的常用模式。

Gitflow工作流定义了一个围绕项目发布的严格分支模型。虽然比功能分支工作流复杂一些，但提供了用于一个健壮的用于管理大型项目的框架。

Gitflow工作流没有用超出功能分支工作流的概念和命令，而是为不同的分支分配一个很明确的角色，并定义分支之间如何和什么时候进行交互。Gitflow当然也具备功能分支工作流的所有好处：Pull Requests、隔离实验性开发和更高效的协作。

Gitflow工作流在第二章中进行详细描述。

### 1.3.4 GitHub Flow



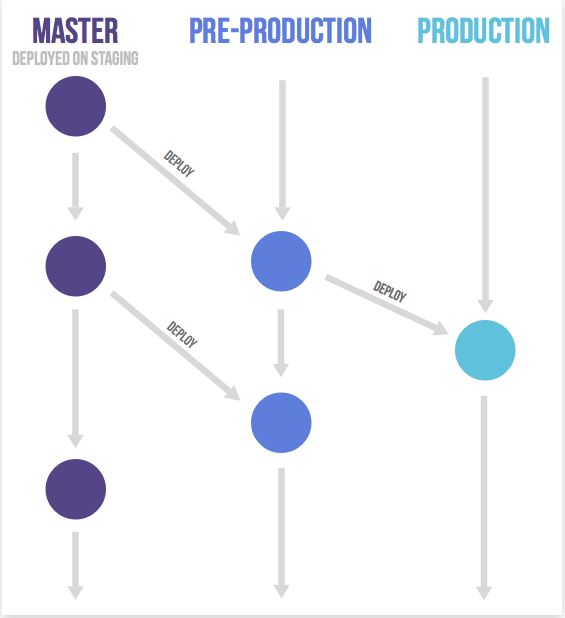
GitHub 工作流十分简单，只有两个分支，master和feature。

master是一个长期分支，而且master分支上的代码，永远是可发布状态。一般master会设置为保护分支，只有有权限的人才能推送代码到master分支。使用步骤如下：

1. 如果有新功能开发，可以从master分支上检出新分支；
2. 在本地分支提交代码，并且保证按时向远程仓库推送；
3. 当需要反馈或者帮助，或者想合并分支时，可以发起一个Pull Request；
4. 当Pull Request讨论并通过后，代码会合并到目标分支（master）。

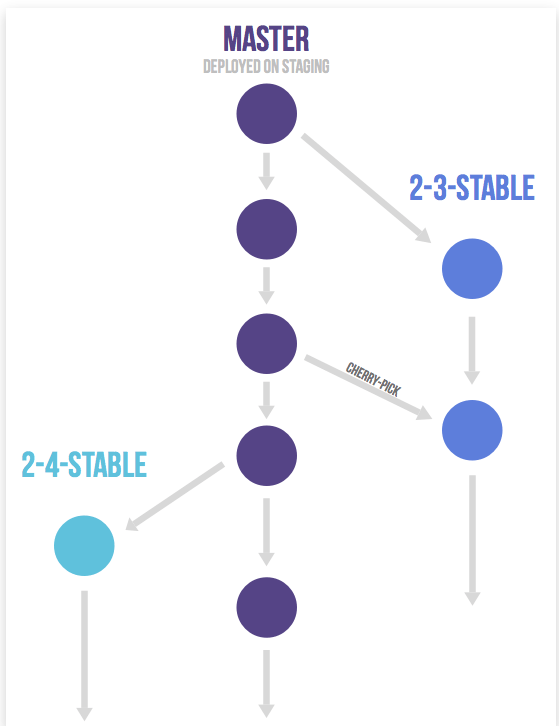
### 1.3.5 GitLab Flow

Gitlab flow 的最大原则叫做"上游优先"（upsteam first），即只存在一个主分支master，它是所有其他分支的"上游"。只有上游分支采纳的代码变化，才能应用到其他分支。



对于"持续发布"的项目，它建议在master分支以外，再建立不同的环境分支。比如，"开发环境"的分支是master，"预发环境"的分支是pre-production，"生产环境"的分支是production。

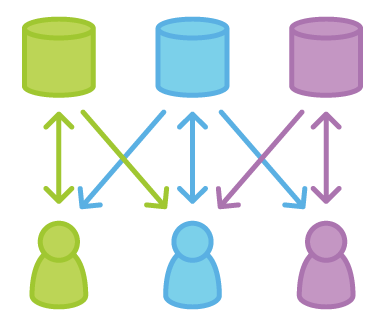
开发分支是预发分支的"上游"，预发分支又是生产分支的"上游"。代码的变化，必须由"上游"向"下游"发展。比如，生产环境出现了bug，这时就要新建一个功能分支，先把它合并到master，确认没有问题，再cherry-pick到pre-production，验证没有问题，才进入production。只有紧急情况，才允许跳过上游，直接合并到下游分支。



对于"版本发布"的项目，建议的做法是每一个稳定版本，都要从master分支拉出一个分支，比如2-3-stable、2-4-stable等等。 以后只有修补bug，才允许将代码合并到这些分支，并且此时要更新小版本号。

### 1.3.6 Forking工作流

Forking工作流和前面讨论的几种工作流有根本的不同。这种工作流不是使用单个服务端仓库作为“中央”代码基线，而让各个开发者都有一个服务端仓库。这意味着各个代码贡献者有2个Git仓库：一个本地私有的，另一个服务端公开的。



Forking工作流的一个主要优势是，贡献的代码可以被集成，但不需要所有人都能push代码到仅有的中央仓库中。开发者push到自己的服务端仓库，而只有项目维护者才能push到正式仓库。这样项目维护者可以接受任何开发者的提交，但无需给他正式代码库的写权限。

# 第二章 兆日科技Git规范

兆日科技使用Gitlab作为Git管理平台，因此本规范都是基于Gitlab进行叙述。

Git工作流采用Git Flow。

大文件采用Git-LFS进行管理。

## 2.1 角色和权限

Gitlab中的角色分为Guest、Reporter、Developer、Maintainer、Owner，它们对应的权限如下：

1. Guest(匿名用户) - 写留言薄，不能读写版本库；
2. Reporter（报告人）-写留言薄、拉项目、下载项目、不能提交代码；

QA，PM可以赋予该权限；

1. Developer（开发者）-写留言薄、拉项目、下载项目、创建代码片段、创建合并请求、创建新分支、推送不受保护的分支、移除不受保护的分支 、创建标签、编写wiki；

该项目研发人员赋予该权限；

1. Maintainer（管理者）- 创建项目、写留言薄、拉项目、下载项目、创建代码片段、创建合并请求、创建新分支、推送不受保护的分支、移除不受保护的分支 、创建标签、编写wiki、增加团队成员、推送受保护的分支、移除受保护的分支、编辑项目、添加部署密钥、配置项目钩子；

产品研发经理、研发负责人可以赋予该权限；

1. Owner（所有者）- 创建项目、写留言薄、拉项目、下载项目、创建代码片段、创建合并请求、创建新分支、推送不受保护的分支、移除不受保护的分支 、创建标签、编写wiki、增加团队成员、推送受保护的分支、移除受保护的分支、编辑项目、添加部署密钥、配置项目钩子、开关公有模式、将项目转移到另一个名称空间、删除项目；

产品总监、技管部Git管理者可以赋予该权限。

### 2.1.1 Gitlab中的权限审批

Gitlab的权限审批在T信中提交“配置管理”，填写申请事由、项目名称、权限明细等，由研发同事发起 --> 产品研发经理审批 --> 产品总监审批 --> 技管部同事开通并审批。

虽然Mantainer和Owner均有添加某个用户为项目成员并可以赋予其一定权限，为了权限的统一管理，均由技管部同事开通。

### 2.1.2 Gitlab中group的创建以及仓库的创建

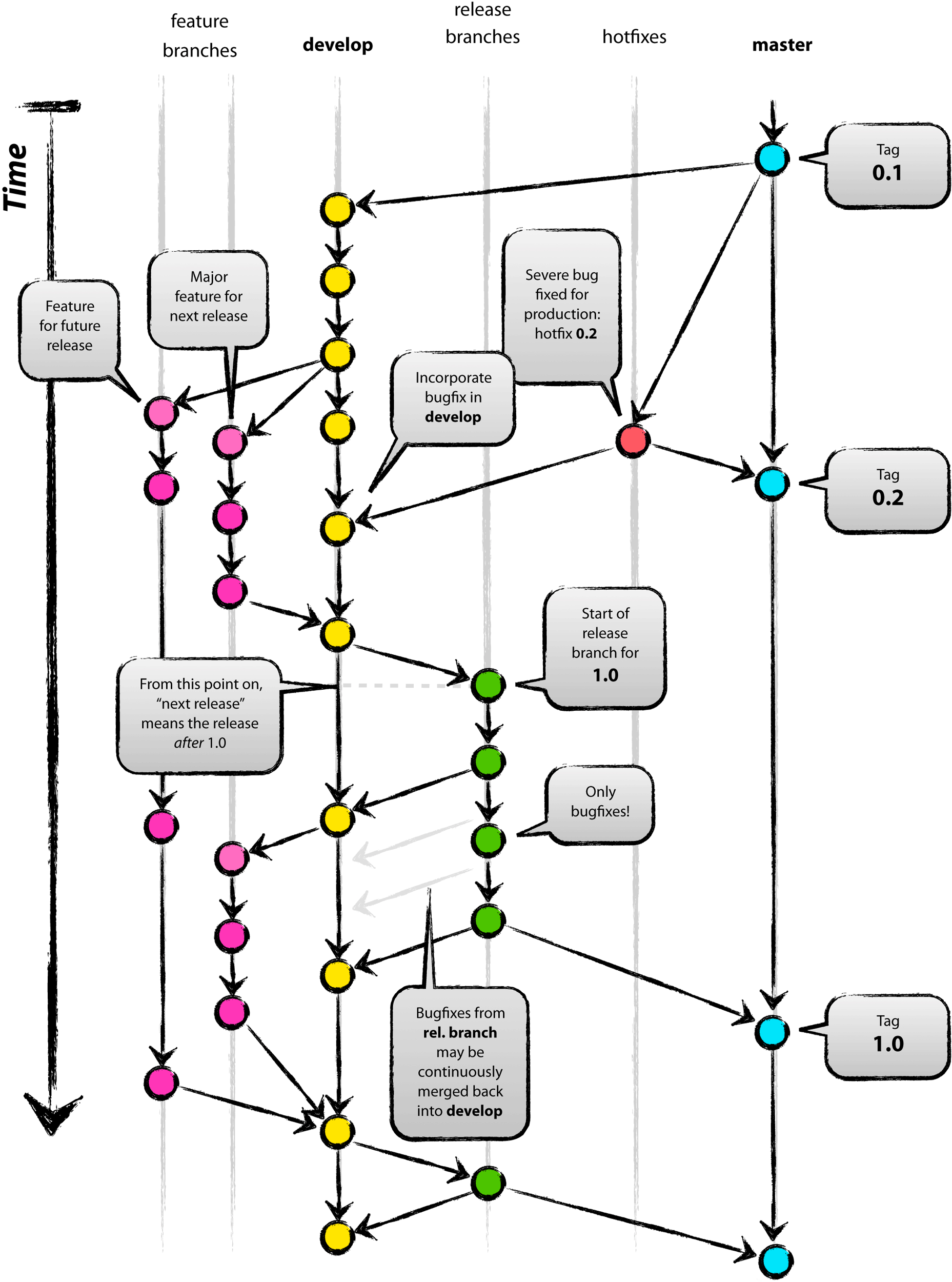
Gitlab产品级别group的创建由技管部同事创建，如目前仓库中已经创建的product, devops, paperprint, yqt, arsenal, bjbank, tchat, xmbank等。创建完成后将赋予对应产品的产品总监Owner权限。

产品级别group内的git仓库或者subgroup由产品总监创建。

## 2.2 工作流

### 2.2.1 GitFlow工作流

采用GitFlow做为代码管理的工作流，GitFlow如下图所示：



关于GitFlow工作流，可参考：<http://blog.jobbole.com/76867/>

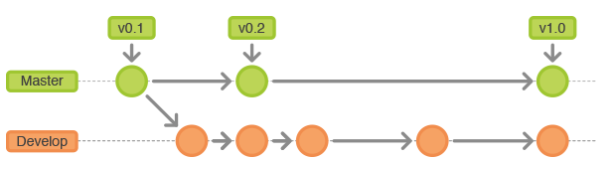
### 2.2.2 GitFlow分支

#### 2.2.2.1 工作方式

Gitflow仍然用中央仓库作为所有开发者的交互中心。和其它的工作流一样，开发者在本地工作并push分支到中央仓库中。

#### 2.2.2.2 历史（master、develop）分支

Gitflow使用两个分支来记录项目的历史。master分支存储了正式发布的历史，而develop分支作为功能的集成分支。这样也方便master分支上的所有提交分配一个版本号。



**1. master分支规范**

存放的是随时可以在生产环境中部署的代码，在项目启动时创建。当开发活动告一段落，产生了一份新的可供部署的代码时，master分支上的代码会被更新。同时，每一次更新，都有对应的版本号标签（tag）。

分支命名：master

该分支由项目管理员（如Team Leader）负责维护，其它人只有拉取（只读）权限。只有release和hotfixes分支的代码可以合并到master分支。

生命周期：伴随着整个项目的生命周期，项目结束时结束。

**2. develop分支规范**

develop分支是开发分支，在项目启动时由项目管理员创建。当develop分支上的代码已实现了软件需求说明书中所有的功能，派生出release分支（管理员操作）

分支命名：develop

分支来源：

1. 开发人员在各自的feature分支完成开发，并通过了Pull Request后，合并至该分支（管理员操作）；
2. release分支的故障处理完成后，合入到该分支（管理员操作）；
3. hotfixes分支完成后，合并到该分支（管理员操作）。

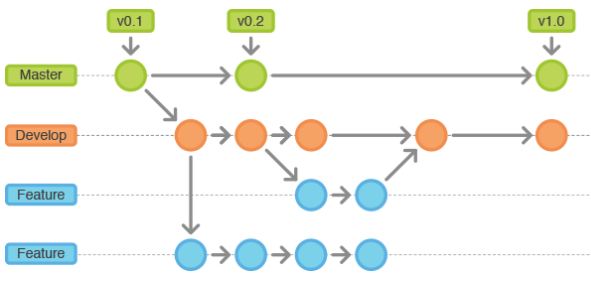
分支去向：

1. 派生出release分支（管理员操作）；
2. 派生出feature分支。

生命周期：伴随着整个项目的生命周期，项目结束时结束。

#### 2.2.2.3 功能（feature）分支

feature分支在开发一项新的软件功能的时候使用，每个新功能都使用自己的独立分支，可以push到中央仓库以备份和协作。但feature分支不是从master分支上拉出新分支，而是使用develop分支作为父分支。当新功能完成时，合并回develop分支。新功能提交应该从不直接与master分支交互。



**feature分支规范：**

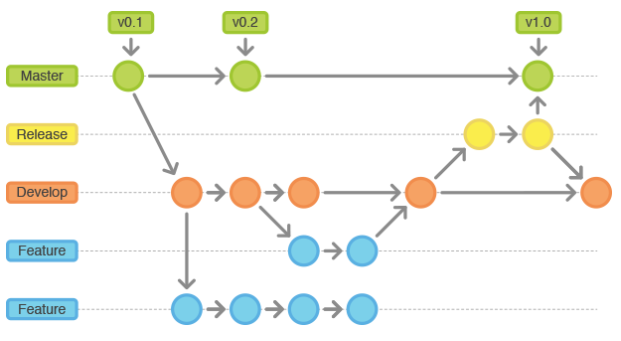
分支命名惯例：feature-分支说明，如：feature-secure-keyboard。

分支来源：由develop分支派生。

分支去向：合并到develop分支。

生命周期：开发一个新功能开始，完成新功能开发并合并回develop分支结束。

#### 2.2.2.4 发布（release）分支



一旦develop分支上有了做一次发布（或者说快到了既定的发布日）的足够功能，就从develop分支上创建一个发布（release）分支。新建的release分支用于开始发布循环，所以从这个时间点开始之后新的功能不能再加到这个分支上（这个分支只应该做Bug修复、文档生成和其它面向发布的任务）。一旦对外发布的工作都完成了，release分支合并到master分支并分配一个版本号打好tag。另外，release分支的修改要合并回develop分支。

使用release分支专门用于准备发布，使得一个团队可以在完善当前的发布版本的同时（在release分支），另一个团队可以继续开发下个版本的功能（在feature分支）。

**release分支规范：**

从develop分支派生（管理员操作）。测试环境中出现的bug，统一在该分支下进行修改，并推送至远程分支。修改内容必须合并回develop分支和master分支。

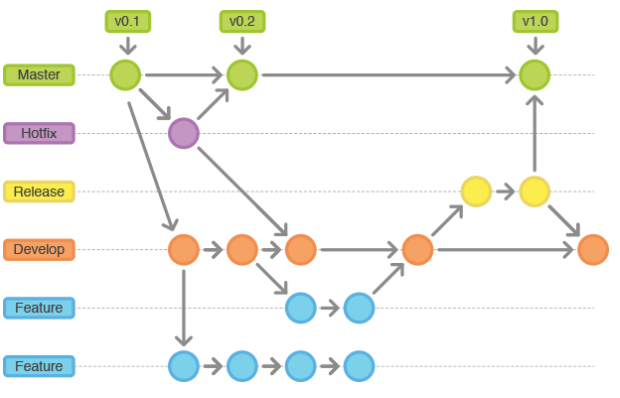
分支命名惯例：release-版本号，如release-1.0。

分支来源：由develop分支派生。

分支去向：合并到develop分支以及master分支。

生命周期：一个阶段功能开发结束开始，完成阶段功能测试并修复所有发现bug，合并到develop和master分支结束。

#### 2.2.2.5 维护（hotfix）分支



维护分支又称为热修复（hotfix）分支，用于给产品发布版本（production releases）打补丁，修复完成，修改应该马上合并回master分支和develop分支（当前的发布分支），master分支应该用新的版本号打好tag。

通过hotfixes为Bug修复使用专门分支，让团队可以处理问题而不用打断其它工作或是等待下一个发布循环。你可以把维护分支想成是一个直接在master分支上处理的临时发布。

**hotfix分支规范**

当master分支发现bug时，在master的分支上派生出一个hotfix分支进行修改；修改完成后，合并至master分支以及develop分支，合并完成，删除该hotfix分支。

分支命名惯例：hotfix-分支说明，如hotfix-#bug100。

分支来源：

1）master分支。

分支去向：

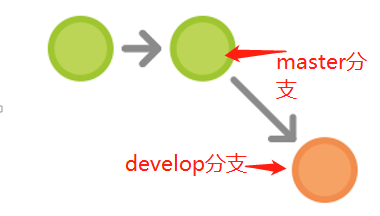
1. master分支（管理员操作）；
2. develop分支（管理员操作）。

生命周期：发现master分支bug开始，合并到develop以及master分支结束。

### 2.2.3 GitFlow基本操作流程

下面的示例演示GitFlow工作流如何用于管理单个发布循环。假设你已经创建了一个中央仓库。

#### 2.2.3.1 创建开发分支



第一步为master分支配套一个develop分支。简单来做可以本地创建一个空的develop分支，push到服务器上：

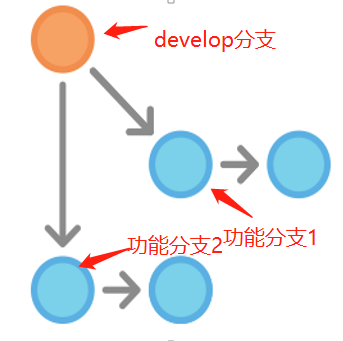
|  |
| --- |
| git branch develop # 创建develop分支 git push -u origin develop # 本地develop push到服务 器，如果远端没有该分支则创建之 |

以后master分支存储了正式发布的历史，而develop分支保存功能的集成历史。其它开发者这时应该克隆中央仓库，建好develop分支的跟踪分支：

|  |
| --- |
| git clone https://git.sinosun.com.cn/user/repo.git # 从仓库中clone最新内容 git checkout develop # 本地切换到develop分支 |

提醒一下，这个操作（创建develop分支）是由项目管理员在创建项目的时候一并完成的，开发人员不需要关心。项目管理员创建develop分支后，需要把develop分支设置为保护分支，只允许有权限的人合入develop分支代码。

#### 2.2.3.2 开发者开始开发新功能



这个示例中，两位开发者开始各自的功能开发。他们需要为各自的功能创建相应的feature分支，这个feature分支基于develop分支：

|  |
| --- |
| git checkout -b feature-1 develop # 本地创建功能分支（基于develop分支） |

开发者提交完成的功能代码到各自功能分支上：编辑、暂存、提交：

|  |
| --- |
| git status # git状态 git add # 将需要提交的文件添加到暂存区 git commit # 将文件提交到本地功能分支 |

##### 2.2.3.2.1 功能分支的粒度

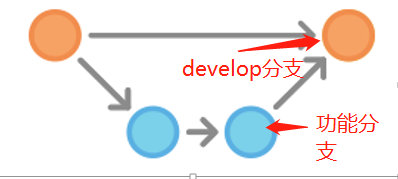
如果项目中同时开发多个“大功能”，应当基于develop创建一个单独的feature分支，项目开发人员基于这个单独的feature分支再创建自己的任务分支。

|  |
| --- |
| git checkout -b feature/v1 develop # 基于develop分支创建feature/v1分支 git push -u origin feature/v1 # 将feature/v1分支提交到远端  ## 其他研发人员基于feature/v1分支拉取代码并开发 |

请注意，合并到上游分支的功能必须相对独立而且是可用的，分支任务工作量0.5-1个工作日，不宜超过2个工作日。

对于小功能，如预计在0.5-1个（不超过2个）工作日工作量的开发任务，则直接基于develop分支创建特性分支即可。

#### 2.2.3.3 开发者完成功能开发



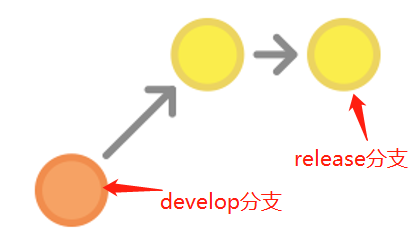
开发者完成功能开发，并且本地自测通过后，需要先将分支同步到远程仓库，然后发起Pull Request请求：

|  |
| --- |
| git pull origin develop # 获取远端（orign）库上 develop 分支的内容，合并到当前分支（feature-1）,有可能会有冲突，需要解决冲突  git add . #  git commit -m "feature finished" # 冲突解决后，再次提交到本地库 git push -u origin feature-1 # 将 some-feature分支提交到远端  # 然后发起 Pull Request：将feature-1 合并到 develop 分支 |

项目管理员审核Pull Request并通过后，将feature-1分支的内容合并到develop分支，然后push到中央仓库：

|  |
| --- |
| git pull origin develop # 同步服务器develop分支 git checkout develop # 切换到develop分支 git merge –no-ff feature-1 # develop分支合并功能分支 git push # 推送本地develop分支到服务器 git branch -d feature-1 # 删除本地分支  注：由于在本规范中，采用了Pull Request模式，develop分支是保护分支，只有项目管理员才能进行合并。 |

#### 2.2.3.4 开发者开始准备发布版本

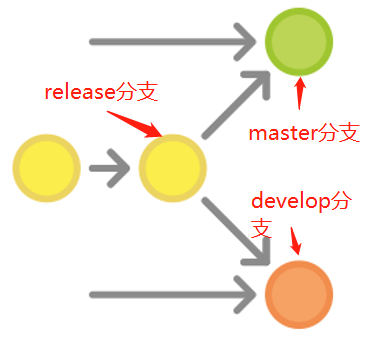


开发者开始准备正式发布项目，创建一个release分支来做版本发布

|  |
| --- |
| git checkout -b release-0.1 develop # 从develop分支创建release-0.1分支并发布 |

这个分支用于版本发布、执行所有测试、更新文档和为下一个发布做准备。

#### 2.2.3.5 开发者完成发布



release的版本经过测试和故障修复后，开发者将其合并修改到master分支和develop分支上，删除该release分支。如果团队要求Code Review，这是一个发起Pull Request的理想时机。

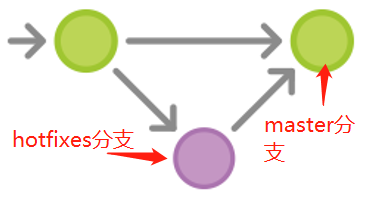
|  |
| --- |
| git checkout master # 本地切换到master分支 git merge --no-ff release-0.1 # master分支合并release-0.1分支 git push # 推送master分支到服务器 git checkout develop # 本地切换到develop分支 git merge –-no-ff release-0.1 # develop分支合并release-0.1分支 git push # 推送develop分支到服务器 git branch -d release-0.1 # 删除release-0.1分支 |

发布分支是作为功能开发（develop分支）和对外发布（master分支）间的缓冲。只要有合并到master分支，就应该打好tag以方便跟踪。

|  |
| --- |
| git tag -a 0.1 -m "Initial public release" master # 本地master分支上打0.1版本的tag  git push --tags # 推送到远端服务器 |

Git提供各种勾子（hook），即仓库有事件发生时触发执行的脚本。可以配置一个勾子，在中央仓库的master分支合并操作完成后，自动构建好对外发布的版本。

#### 2.2.3.6 最终用户发现Bug



对外发布后，开发者去做下一个发布的新功能开发。这时有最终用户开了一个Ticket报告当前版本的一个Bug。为了处理Bug，开发者从master分支上拉出了一个维护分支，提交修改以解决问题，然后直接合并回master和develop分支：

|  |
| --- |
| git checkout -b hotfix-issue-#001 master # 在master分支上创建一个hotfix分支用来解决bug  ## Fix the bug git checkout master  git merge –-no-ff hotfix-issue-#001 # master分支上合并故障解决 git push # 推送到远端服务器(只有管理员可以操作，非管理员请 提 Pull Request) |

维护分支中新加这些重要修改需要合并到develop分支中，所以开发者要执行一个合并操作。然后就可以安全地删除这个分支了：

|  |
| --- |
| git checkout develop # 切换到本地develop git merge -–no-ff issue-#001 # 合并故障分支 git push git branch -d issue-#001 # 删除故障分支 |

注意，在hotfix分支合并到master分支之前，需要通过Pull Request进行审核。

### 2.2.4 项目研发以及发布流程

#### 2.2.4.1 Git项目研发流程

在Gitlab中有Guest，Reporter，Developer，Maintainer，Owner等角色，这些角色与Gitflow项目研发流程的结合，请参考下图



#### 2.2.4.2 研发分支与版本部署环境的关系

主要流程如下：



##### 2.2.4.2.1 研发自测

研发工程师完成功能开发后开始启动研发自测，研发自测的版本放入项目版本库中开发环境目录，并在开发环境部署。

研发自测分为以下步骤：

* 1. 自动化测试：编写功能代码的同事完成自动化测试代码
  2. 功能测试：自行编写或获取测试人员编写的功能测试用例，进行功能测试
  3. 代码静态检查：checkstyle以及findbugs代码静态检查

研发自测通过后在gitlab提交pull Request，由项目经理审核后，将代码合入到develop分支。

若研发自测不通过，在功能分支上故障修改完成后才能提交pull Request，经项目经理审核后，将代码合入到develop分支。

##### 2.2.4.2.2 发布黑盒测试

版本经过项目经理审核后合入develop分支，项目经理可以基于develop分支创建release分支并打包版本，打包的版本放入项目版本库中黑盒环境目录，并在黑盒环境部署，部署完成后提交版本测试由软件测试团队测试。

软件测试团队功能或自动化测试完成后提交Bug到研发团队，研发工程师在release分支解决完bug后再交由测试团队测试，直到通过测试。

通过测试的release分支版本合入develop分支，如果有bugfix，需要提交Pull Request，经过项目经理审核后，将代码合入到develop分支。

通过测试的release分支版本合入master分支，需要提交Pull Request，经过项目经理审核后，将代码合入到master分支。

##### 2.2.4.2.3 发布沙盒测试

经过项目经理审核后，release分支完成的功能合入到master分支。项目经理在master分支上发布沙盒测试版本，发布的版本放入项目版本库中沙盒环境目录，并在沙盒环境部署，部署完成后提交版本测试由软件测试团队测试或前方测试人员。

软件测试团队功能或自动化测试完成后提交Bug到研发团队，研发工程师需要创建hotfix分支，在需要在hotfix分支上解决bug并重新发布版本通过测试。hotfix分支需要测试通过后合入到master和develop分支。

##### 2.2.4.2.4 发布生产版本

通过沙盒测试的版本，项目经理可以放入到版本库生产环境的目录，并由运维人员升级到生产环境。

## 2.3 项目组织

有这样的场景：

1. 一些通用的模块希望能够被抽离出来作为单独的项目，以便其他项目也可以使用；
2. 项目中使用了一些第三方库，但是并不想将代码直接拷贝到项目中，而仅仅只是单纯的引用；
3. 项目中包含大量二进制格式的大文件（比如release包）。

本节叙述了解决这些问题的方法。

### 2.3.1 Git Submodule

Git Submodule允许你将一个 Git 仓库作为另一个 Git 仓库的子目录。它可以将另一个仓库引用到自己的项目中，同时还保持仓库之间提交的独立。Git Submodule适合于项目中引用第三方库的场景。具体操作流程如下：

1. 增加子模块

项目管理员在master分支下增加子模块：

|  |
| --- |
| git submodule add # 添加子模块 git status # 查看状态 cat .gitmodules # 出现的gitmodules配置文件记录了子模块与项目URL的映射 git commit -m "added 3rd module" # 提交到master分支 |

1. 克隆子模块

当克隆一个含有子模块的项目时，默认会包含该子模块目录，但其中还没有任何文件。

方法一：

|  |
| --- |
| git submodule init # 用来初始化本地配置文件。 git submodule update # 从该项目中抓取所有数据并检出父项目中列出的合适的提交 |

方法二：

给git clone命令传递--recursive选项，git会自动初始化并更新仓库中的每一个子模块。

1. 更新子模块

|  |
| --- |
| git submodule update --remote # 进入子模块然后抓取并更新 |

如果子模块无法更新，可以尝试使用以下命令：

|  |
| --- |
| git submodule update --remote -f # 强制更新 |

1. 提交子模块

方法一：进入每个子模块推送更新到远程仓库

方法二：--recurse-submodules提交

|  |
| --- |
| git push --recurse-submodules=on-demand # 提交子模块 |

1. 删除子模块

|  |
| --- |
| git submodule deinit {MOD\_NAME} # 逆初始化模块，其中{MOD\_NAME}为模块目录，执行后可发现模块目录被清空  git rm --cached {MOD\_NAME} # 删除.gitmodules中记录的模块信息（--cached选项清除.git/modules中的缓存）  git commit -am "Remove a submodule." # 提交更改到代码库，可观察到'.gitmodules'内容发生变更 |

### 2.3.2 Git SubTree

Git在1.8.0版本引入了git subtree，它使用Git的subtree merge策略来得到类似git submodule的结果。但本质上，它是将子项目的代码全部merge进父项目。使用git subtree，不仅可以将其他项目合并为父项目的一个子目录，而且可以从父项目提取某个子目录的全部历史作为一个单独的项目。

Git SubTree适合项目中引入第三方源码的处理。

相比Git submodule，Git SubTree的特点：

* 管理和更新流程比较方便；
* 不再有.gitmodules文件；
* 克隆仓库不再需要init和update等操作；
* 删除时不再像git submodule那样费劲。

Git SubTree的操作流程如下：

**1. 添加Subtree**

将一个已存在的Git仓库以Subtree方式添加为子项目:

|  |
| --- |
| git subtree add --prefix=<prefix> <repository> <ref> # 项目添加Subtree git commit # 本地提交 git push origin master # 提交到远程master分支 |

其中--prefix选项指定了子项目对应的子目录，--squash选项用以压缩Subtree的提交为一个提交，这样父项目的历史记录里就不会出现子项目完整的历史记录。添加完成后需要在master分支commit以及push。

**2. 更新Subtree**

|  |
| --- |
| git subtree pull # 更新Subtree |

**3. 提取Subtree**

|  |
| --- |
| git subtree split # 提取Subtree |

**4. 提交子项目**

|  |
| --- |
| git subtree push # 在Subtree上提交 |

### 2.3.3 二进制大文件的处理

在项目中，可能会包含一些较大的、二进制格式的文件，如项目发布时的打包出来的二进制发布包（xxx.tar 或 yyy.rpm），或者是一些资源文件。这些比较大的二进制文件，如果有很多历史记录的话会造成客户端拉取版本耗时过长且占用磁盘过大。对于这些大文件，可以创建专门的仓库来存放。具体可使用如下两种方式（推荐使用方法一）：

方法一：

所有产品源代码使用Git仓库管理，项目发布时打包出来的的二进制包使用SVN仓库管理。这样做的好处就是与目前发布包的方式相同，同时也能避免客户端拉取时间过长和占用磁盘过大的问题。

方法二：

对项目版本库的管理，每个产品创建一个Git仓库主要用于存放发布版本，例如mstp的项目可以创建仓库 [http://host/release/mstp/](http://host/release/mstp/v3.0.1) 来保持发布包。大文件的管理可以使用Gitlab + Git LFS+ 稀疏检出的方式，防止客户端拉取过多的历史版本大文件。具体步骤如下：

1.创建存放发布版本的项目仓库：[http://host/release/mstp/](http://host/release/mstp/v3.0.1)

2.项目仓库中创建存放产品版本的文件夹，文件夹命名如：v1.0.0，v1.0.1等，项目发布的版本需要归档到对应目录;

3.项目仓库配置Git-LFS;

4.项目拉取时采用稀疏检出的方式，指定单独的目录如 v1.0.0，进行版本拉取。

关于git中如何处理大文件，可参考2.5节。

### 2.3.4 项目从svn迁移到Git

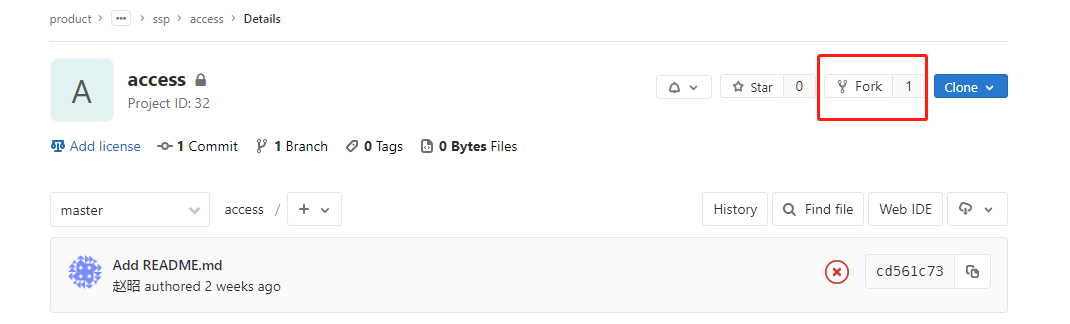
Git 中虽然提供了git svn的命令用于迁移svn项目到Git，但这种做法会将svn所有提交历史导入Git仓库中，造成Git仓库过于庞大，因此建议从svn中export一份最新的代码然后上传到Git仓库。主要流程如下

1. 项目管理者从svn仓库中export最新项目代码；
2. 项目管理者规划项目中的大文件以及子项目，重新组织适合Git管理的目录结构；
3. 技管部在Gitlab上创建对应项目；
4. 项目管理者本地git init后，添加Git LFS大文件管理；
5. 项目管理者使用git push origin上传项目到Git服务器；
6. 项目管理者配置子项目后使用git push origin上传到Git服务器；
7. 开发者git clone该项目，并在本地检查是否完成迁移。

### 2.3.5 gitlab上仓库的手动fork

项目的fork是指将gitlab上的其他仓库以及其所有仓库历史复制一份到你希望的目录下，本节只介绍如何单独fork一个仓库，批量复制仓库的操作请参考2.10章节银行项目操作。

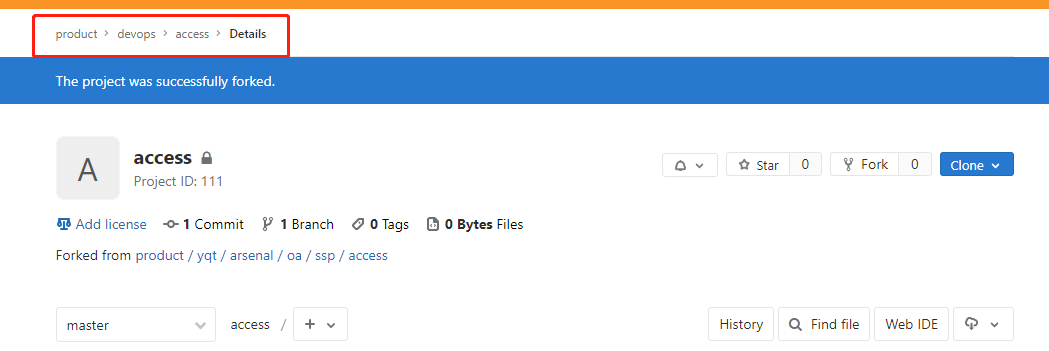
在gitlab上单独fork一个仓库，例如fork product/yqt/arsenal/oa/ssp/下的access仓库。



点击fork按钮，选择你想fork的目的group，例如想复制到product/devops下，点击选择该项。



等待fork完成后会进入fork好的仓库界面。



## 2.4 分角色操作指南

这里主要描述项目在使用Git flow工流以及Git lab工具下的操作。

### 2.4.1 开发人员（命令行操作）

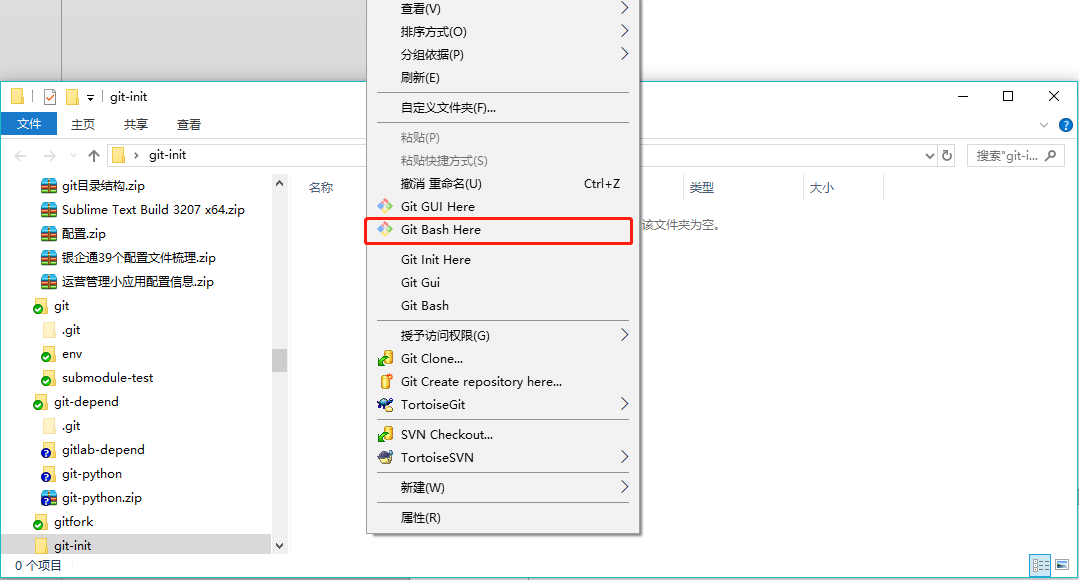
#### 2.4.1.1 准备工作

首先需要安装git，地址：

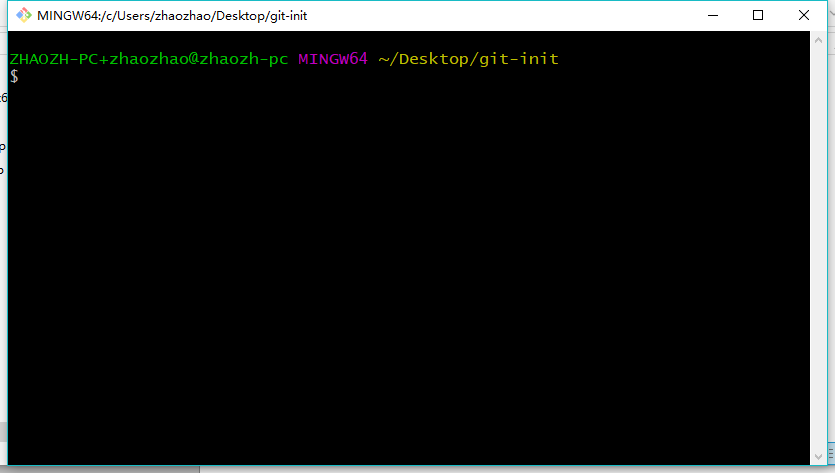
|  |
| --- |
| [\\fs\install\git\Git-2.21.0-64-bit.exe](file:///\\fs\install\git\Git-2.21.0-64-bit.exe) |

#### 2.4.1.2 初始化

开发人员新建一个文件夹，右键在选项中选择Git Bash Here。



选择后会出现如下提示框：



开发人员在该提示框中执行初始化命令（若无特殊说明，git的所有命令均在git bash中执行）。

|  |
| --- |
| git init # 本地创建项目目录并进行git初始化  git config --global user.name "\*\*\*" # 设置用户名 --global是全局生效  git config --global user.email "\*\*\*" # 设置用户email |

开发人员在该提示框中生成ssh秘钥并配置ssh：

1. 在git bash提示框中执行 ssh-keygen -t rsa -C “zhao.zhao@sinosun.com.cn”（邮箱为自己的Email，全部按enter，生成ssh秘钥对）；

这个密钥对一共有两个文件：

id\_rsa 你的私钥

id\_rsa.pub 你的公钥

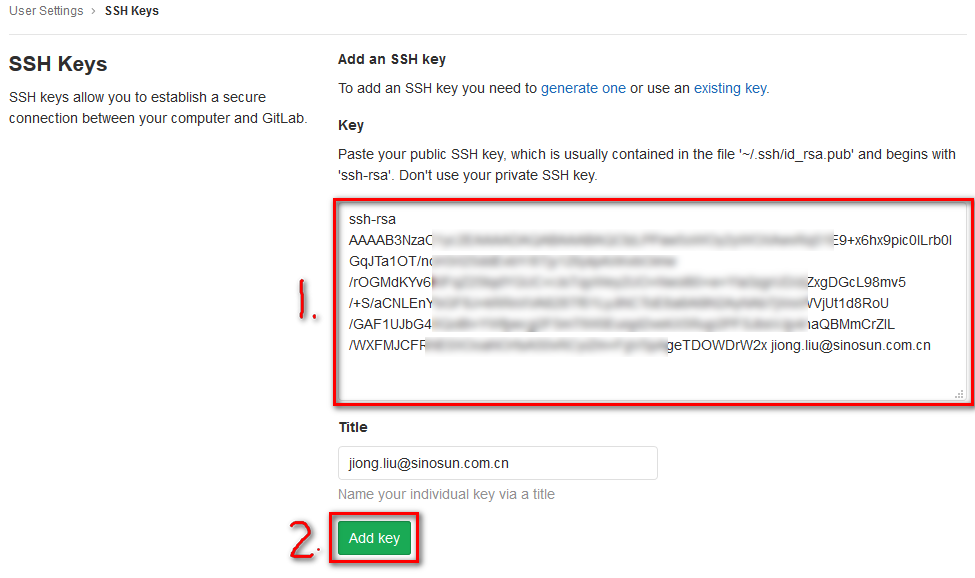
2. 进入秘钥所在目录，cd ~/.ssh;

3. 用 cat 命令显示出公钥内容，cat id\_rsa.pub在命令行打开这个文件，会直接输出密钥；

4. 用你的域账号登录公司git网站：

<https://git.sinosun.com.cn> （注意 URL 中是 https 而不是 http）

5.登录后，直接输入链接：<https://git.sinosun.com.cn/profile/keys>将刚才显示的 id\_rsa.pub 的内容，粘贴到 Key 中，如下图所示：



粘贴后，选择“Add key”，添加公钥。

#### 2.4.1.3 连接到远程仓库

初始化完毕后，可以clone远程仓库。开发人员在开发项目时，默认项目管理人员已经创建好了master和develop分支。

注意，使用仓库地址格式为： [git@git.sinosun.com.cn:test/test/src/test1.git](mailto:git@git.sinosun.com.cn:test/test/src/test1.git) 不要用 http协议的地址。

以一个测试仓库为例，在git bash中执行以下命令。

|  |
| --- |
| git clone [git@git.sinosun.com.cn:test/test/src/test1.git](mailto:git@git.sinosun.com.cn:test/test/src/test1.git) ## clone远程仓库 |

clone仓库完成后，可以使用：

|  |
| --- |
| git branch -av ## 查看所有分支，包含远程分支和本地分支 |

#### 2.4.1.4 新建功能分支

|  |
| --- |
| git checkout develop # 切换到 develop 分支 git pull # 同步代码 git checkout -b feature-1 develop # 基于 develop 分支，创建新的名为 feature-1 的分支，在此分支上开发新功能 |

#### 2.4.1.5 更新远端代码到本地

如果需要更新远端代码到本地，可以使用下面的命令：

|  |
| --- |
| git fetch # 获取远端仓库的所有内容存放到本地  git pull # 获取远端仓库的所有内容，并自动进行merge，  相当于：get fetch + git merge  git fetch origin develop # 同步远端仓库中develop分支的内容存放到本地  git pull origin develop # 同步远端仓库中develop分支的内容到本地的develop分支，  并自动merge |

#### 2.4.1.6 在分支上开发

日常工作：将本地修改加入到stage中，并且用git commit命令将修改提交到本地库中：

|  |
| --- |
| git status # 查看当前本地分支状态 git add . # 将所有文件加入到 stage 中 git add file # 将某个文件加入到 stage 中 git commit -m "xxxxx" # 将 stage 中的内容，提交到本地库中 |

开发人员在本地开发过程中常常会提交一些无意义的提交历史，可以使用git-rebase的方法来修改自己的提交颗粒度。

**git rebase –i (整理同一分支的提交日志)**

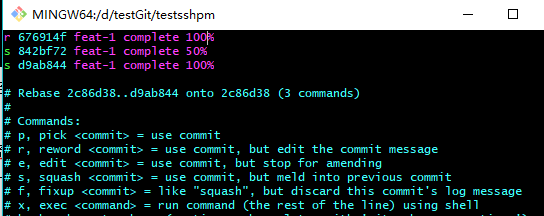
使用git rebase -i命令整理同一个分支上的提交。举例如下：



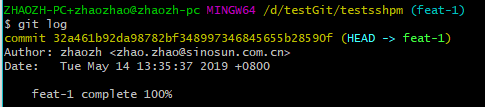
本地使用了三个commit来完成了feat-1功能的开发。提交到远程仓库之前，可以将三个commit合并成一个，过滤掉开发的中间过程

|  |
| --- |
| git rebase -i HEAD~3 # ~表示当前提交的前几个提交 |

进入修改界面，详细的命令可以参考修改界面显示。



只保留feat-1 complete这个commit，其他的合并到这个commit中。整理日志之后的git log显示如下。



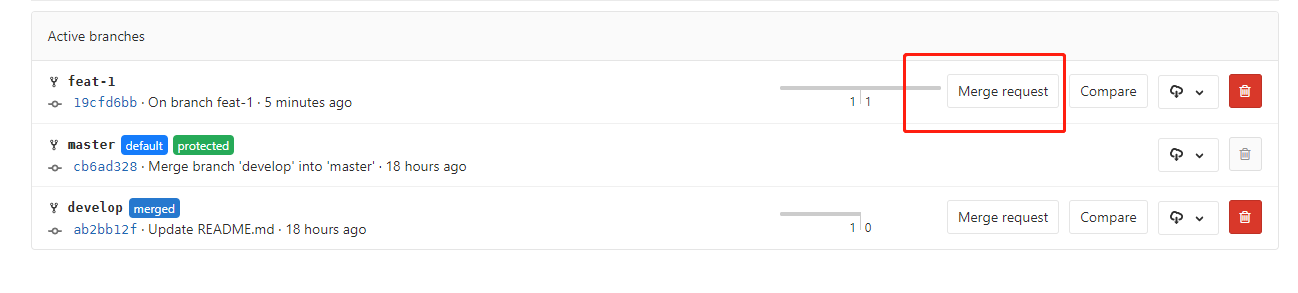
#### 2.4.1.7 分支合并

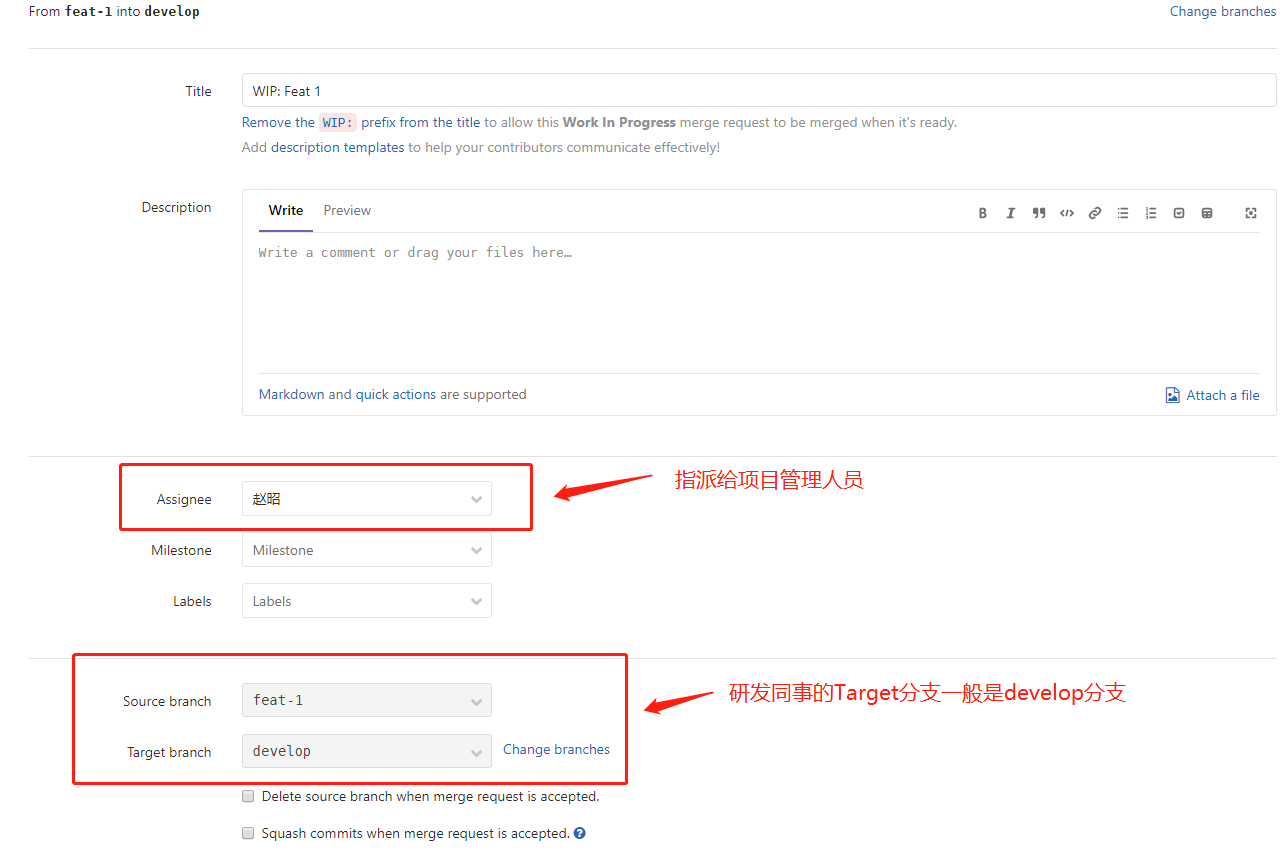
feature分支开发完成后，需要发起Pull Request（Gitlab中叫Merge Request），将分支内容合并到develop分支。步骤如下：

1、将此分支的内容提交到远端仓库：

|  |
| --- |
| git fetch origin develop # 获取远端（orign）库上 develop 分支的内容，合并到当 前分支有可能会有冲突，需要解决冲突  git merge --no-ff # 合并并解决冲突  git push -u origin feature-1 # 将 feature-1分支提交到服务器（服务器中原本没有，因 此需要要加上 -u参数，在服务器上新建此分支） # 然后发起 pull request，审核通过后管理员将 feature-1合并到 develop 分支，完成后删除feature分支 |

2、在Gitlab上提交merge Request，由管理员审核代码合并。





#### 2.4.1.8 删除本地以及远端分支

功能分支合入到develop分支后，本地的feature-1分支以及远端的feature-1分支就可以删除掉。

|  |
| --- |
| git branch -d feature-1 # 删除本地feature分支  git push origin -d feature-1 # 删除远端feature分支 |

#### 2.4.1.9 发布测试版本

develop分支合入功能点后，准备创建release分支并发布测试。

|  |
| --- |
| git checkout develop # 切换到 develop 分支 git pull origin develop # 更新内容 git checkout -b release-0.1 develop # 从 develop 产生一个分支 release-0.1  git push -u origin release-0.1 # 将 release-0.1 推到远端仓库 |

### 2.4.2 开发人员（TortoiseGit）

#### 2.4.2.1 下载地址

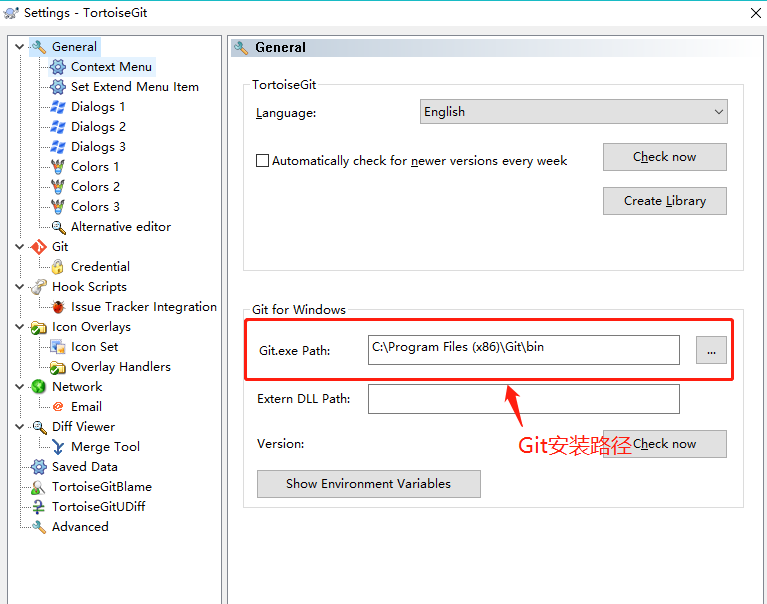
下载地址：<https://tortoisegit.org/download/>

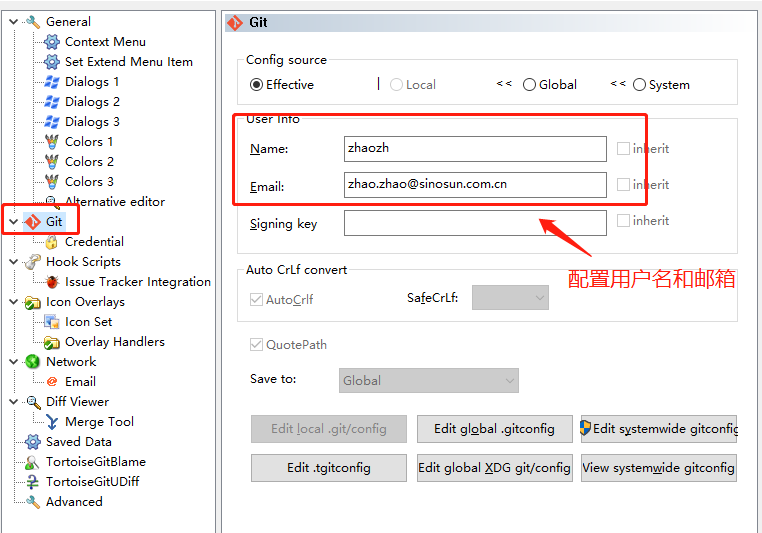


请根据自己的电脑系统下载。

#### 2.4.2.2 账户以及Git配置

选定一个存放Git项目的目录，如：D:\TortoiseGitTest，右键在TortoiseGit菜单栏中打开setting选项进行如下配置。

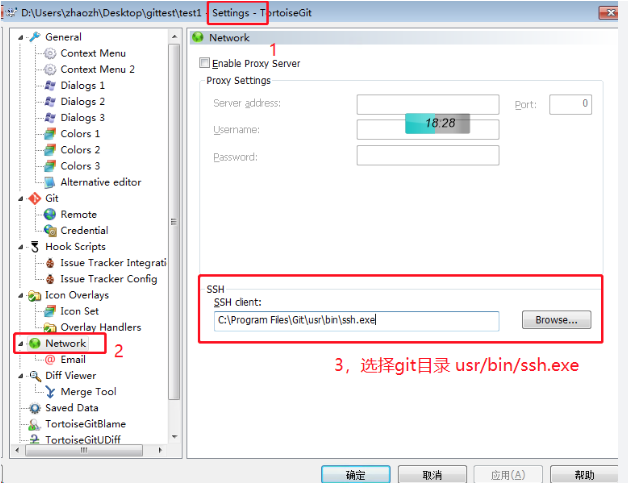




#### 2.4.2.3 gitbash中已经配置过ssh场景(推荐使用该方法)

用户在使用TortoiseGit前必须完成2.4.1 命令行配置。

如果用户已经在git bash中配置过ssh的公私钥，那么只需要在TortoiseGit -> setting ->Network SSH配置中将 SSH Client设置为git目录下的ssh.exe，即可。请参考下图所示：

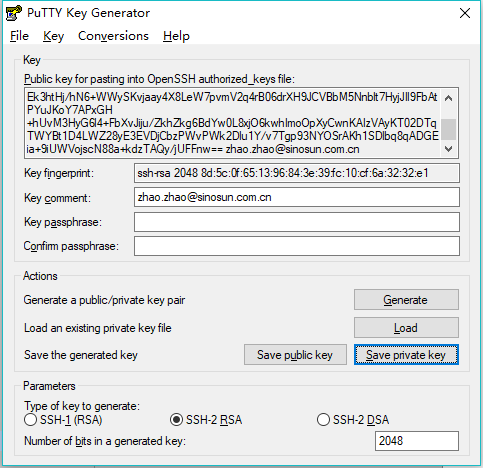


#### 2.4.2.4 使用TortoiseGit 生成ssh密钥对(不推荐使用)

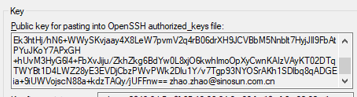
TortoiseGit默认使用putty格式的私钥，与Gitbash中生成的秘钥格式不同，所以不能直接将gitbash产生的私钥直接导入到TortoiseGit中使用。可以使用TortoiseGit自带的puttygen.exe工具生成ssh秘钥对。

ssh秘钥对生成：

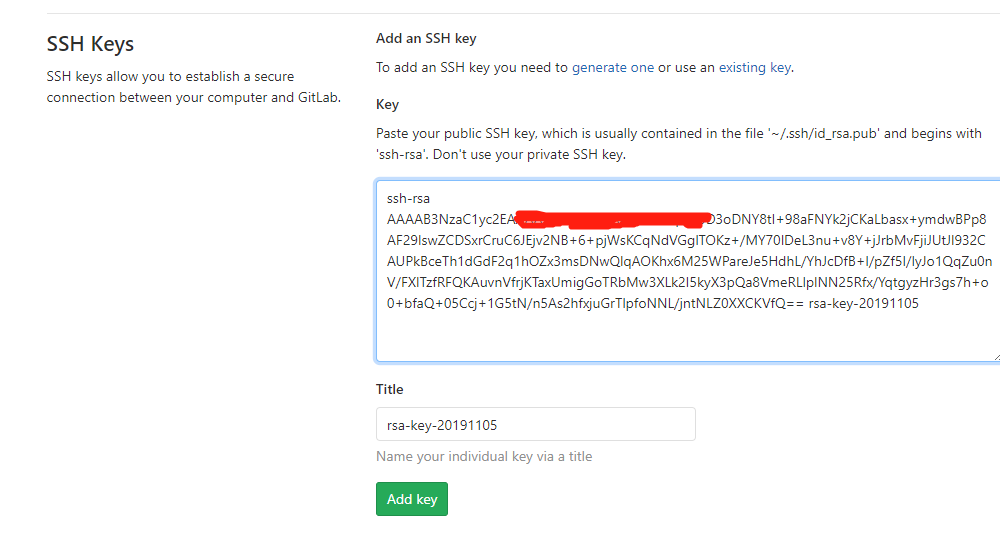
1、双击puttygen.exe工具



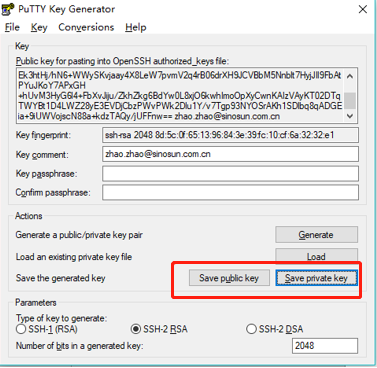
在该页面首先点击Generate按钮，等待生成完成后请将公钥



复制到<https://git.sinosun.com.cn/profile/keys>下。请注意复制时公钥长度较长，请完整复制，否则添加到gitlab中会报错。



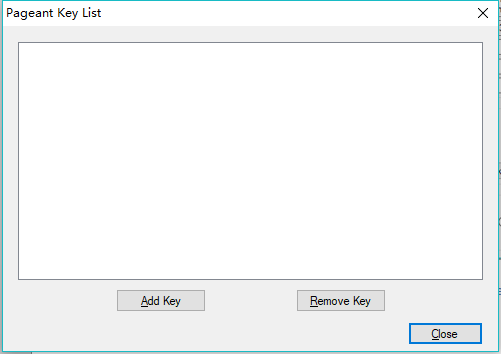
2、保存公私钥，请点击：



保存该公私钥，命名可以自拟如” xx-tortoisegit-privatekey”、 ” xx-tortoisegit-publickey”。

3、添加私钥到TortoiseGit

使用TortoiseGit自带的pageant.exe工具，该工具在安装目录\TortoiseGit\bin\pageant.exe下，双击执行该程序，在电脑右下角会看到 图标，点击该图标

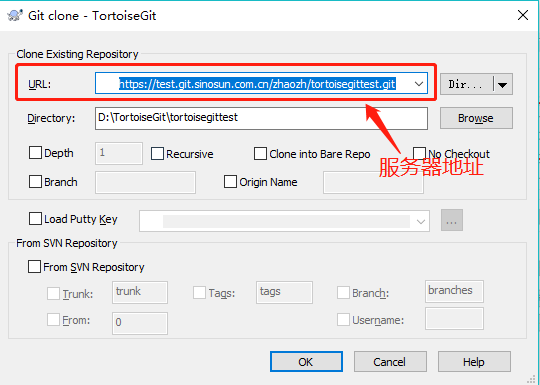


将生成的私钥添加到该程序中。即可以完成配置。

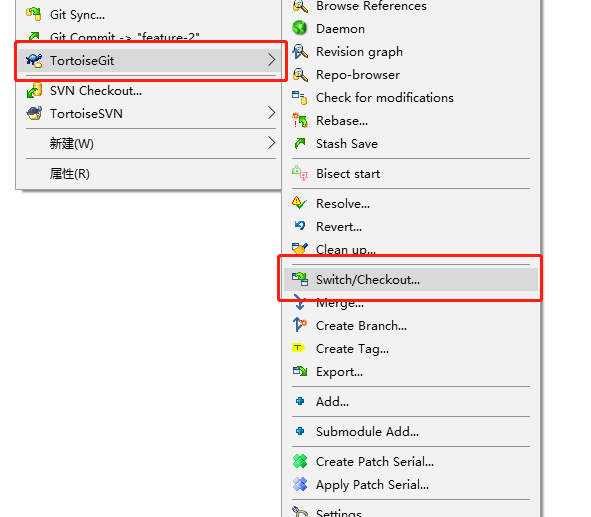
#### 2.4.2.5 示例

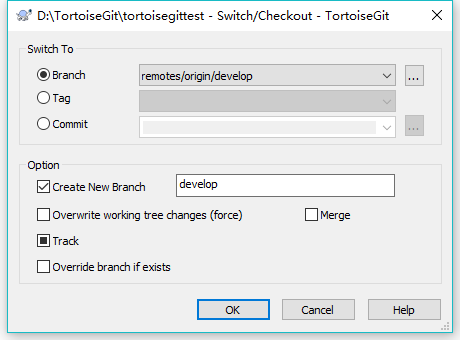
##### 2.4.2.5.1 克隆一个项目

右键->Git clone

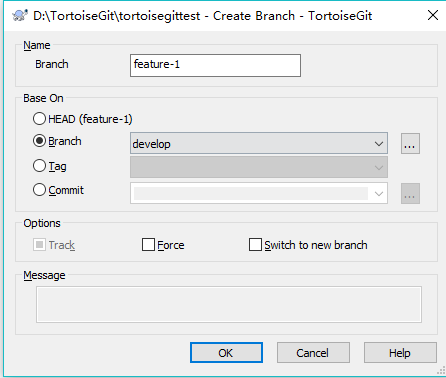


##### 2.4.2.5.2 切换本地develop分支



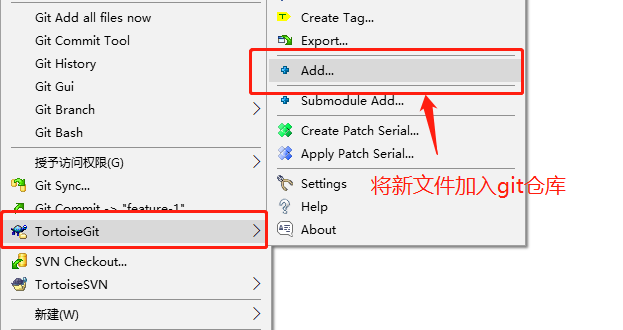


##### 2.4.2.5.3 基于develop分支新建feature分支



##### 2.4.2.5.4 在新建的功能分支上进行开发

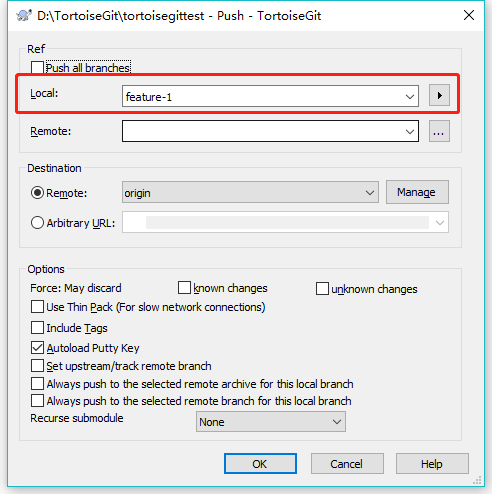
开发完成后将新文件加入仓库，可以查看修改文件的比对，然后将文件commit到本地仓库。





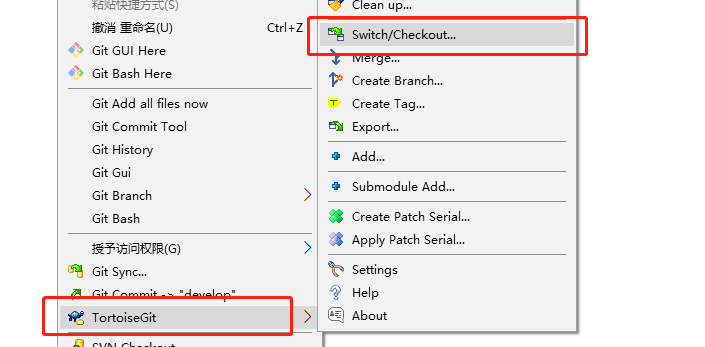
##### 2.4.2.5.5 分支合并

分支开发完成后将此分支的内容提交到远端仓库

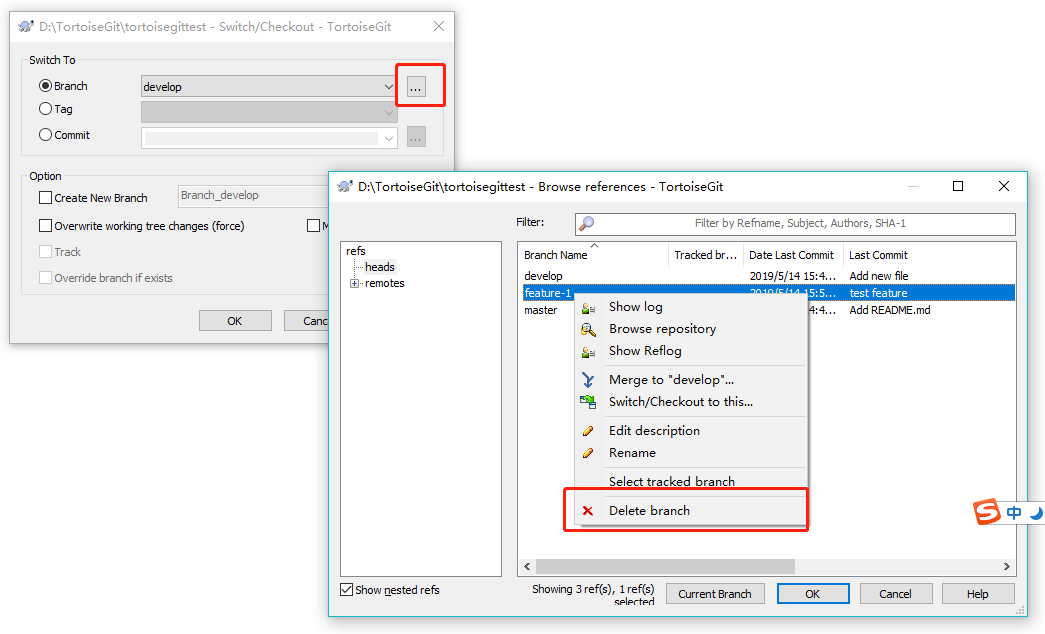


push完成后提交一个pull request，项目管理员通过Pull Request后代码合入develop分支。

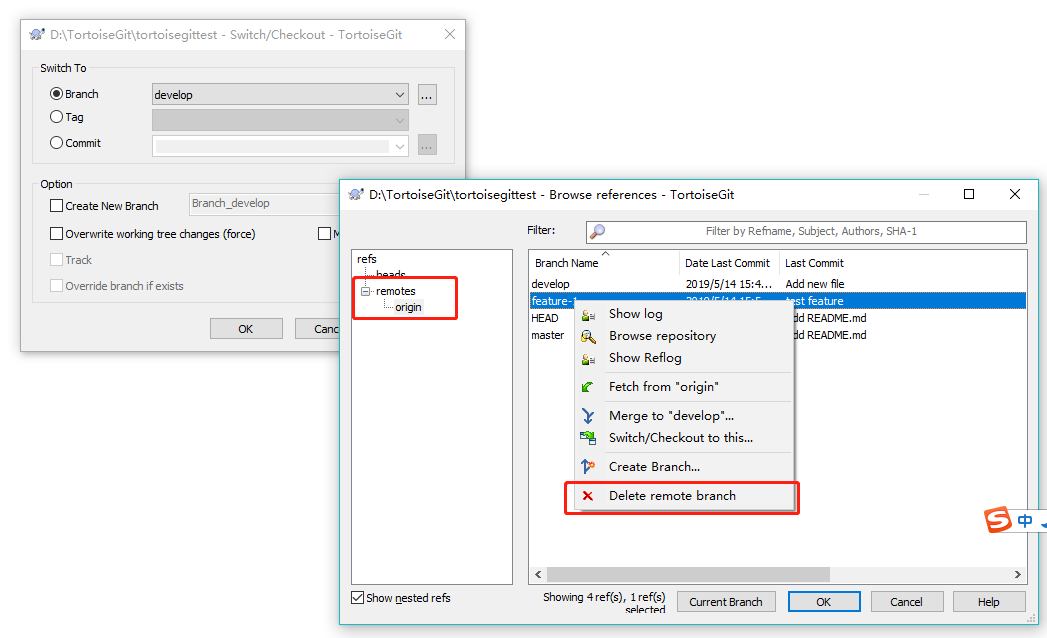
##### 2.4.2.5.6 删除本地以及远程分支



进入switch/checkout选项



删除本地分支



删除远端分支

### 2.4.3 项目管理人员（项目经理）

项目管理人员主要包括项目经理，架构师、开发经理、核心开发人员(能够Code Review，能够处理Pull Request)。

项目管理人员主要负责Git中项目的维护、项目分支的管理、Pull Request的处理，持续集成等。项目管理人员在项目中的权限为Maintainer，管理者权限。

需要注意的是Gitlab中的Merge Request与Git中的Pull Request是一个意思。

#### 2.4.3.1 分支管理

项目管理员主要职责：

1. 在项目启动时，项目管理员在Git中创建master和develop分支；
2. 项目管理员需要对feature分支合入develop进行审核
3. 项目管理员需要处理将hotfix分支合入到develop分支以及master分支；
4. 一个研发功能完成后，项目管理员需要从develop分支创建release分支进行对外发布测试。完成后需要将release分支合入到develop分支以及master分支；
5. 项目管理员对master分支上合入或发出的版本打Tag。

#### 2.4.3.2 发布版本完成后的处理

release分支测试无误后，将 release 的内容，发起code Review，并将代码分别合并到 master 和 develop 中。然后删除release分支。

#### 2.4.3.3 为master分支打tag

release分支和入到master分支后，需要基于 master打Tag

|  |
| --- |
| git checkout master # 切换到 master git tag -a "1.0" -m "First stable release" # 针对当前所在的分支打tag git push --tags # push 到远端库中 |

项目经理添加tag时，版本号直接以数字开头，如1.1.0(版本号前不加’v’)

#### 2.4.3.4 线上紧急bug的处理

对外发布后，发现了 bug，需要修复，流程如下：

1. 从master分支上拉出了一个维护分支

2. 研发人员在分支上进行bug修复，修复完成后提交Pull Request

3. 审核Pull Request后合并维护分支到master和develop分支

|  |
| --- |
| git checkout master # 切换到 master git pull # 用 pull 命令更新（同步） master分支内容到本地 git checkout -b hotfix-issue-#001 master # 基于 master 本地拉出一个 hotfix 分支 ##修复完成后 git checkout master # 切换到 master git pull # 更新（同步）代码 git merge --no-ff hotfix-issue-#001 # 合并代码 git push # 将当前分支（master）的内容 push到远端仓库 git tag -a "1.1" -m "Fix issue 001" master # 将 master 分支打上 tag git push --tags # push 到远端库中 git checkout develop # 切换到 develop git pull # 更新（同步）代码 git merge --no-ff hotfix-issue-#001 # 合并代码 git push # 将当前分支（develop）的内容  push 到远端仓库 git branch -d issue-#001 # 删除本地分支 |

#### 2.4.3.5 Code Review

无论进度有多么紧迫，Code Review的过程都需要做。所有提出的问题根据进度的紧迫程度、问题的严重程度、改动成本等决定问题的解决顺序，并将Code Review的问题记录在缺陷跟踪管理系统内。

##### 2.4.3.5.1 Code Review 的规则

1. 不合并没有完成审核的Pull request；
2. Pull request的发起人要主动地推动Pull request的审核，Leader也会密切关注Pull request审核的进度，在需要的时候及时介入；
3. 任务的完成被定义为：代码编写完成，经过自测，提交的PR经过审核并且合并到相应的分支；
4. 发起Pull request以后，请将Pull request的链接在Tchat上发给代码审核者，以此通知对方及时进行审核。

##### 2.4.3.5.2 Pull Request 的说明

Pull Request也可以简称PR，在Gitlab中称为Merge Request。关于PR需遵守如下原则：

1. 任务完成才能提交PR；
2. PR应该在一个工作日内被合并或者被拒绝；
3. PR在有严重问题（包括但不限于架构问题、安全问题、设计问题）或者任务无效的情况下会被拒绝；
4. 严禁一个PR里面有多个任务，除非它们是紧密关联的；
5. PR提交之后只允许针对Review发现问题再次提交代码;除非有充足的理由，严禁在同一个PR中再次提交其它任务的代码；
6. 改动的范围、解决的问题，需要代码审查的人留意那些影响比较大的改动要特别说明。如果对基础、公共的组件进行了改动，一定要特别留意。

##### 2.4.3.5.3 审核人员邀请原则

1. 在创建PR时，Reviewers（审核人）一栏里主要填写“必需审核人”。只有这些人审核都通过，才允许合并；
2. 除了“必需审核人”外，还有一些其它审核人，可以在Description里做为“邀请审核嘉宾”@进来；
3. 主干分支间的合并，如Develop => Master，则需要把整个团队（开发+QA）都列为“必需审核人”。

**必须审核人**，可能包括以下人选：团队Leader、产品架构师、对此问题比较熟悉的（之前一直负责这部分业务的同事）、此PR解决的问题对他影响比较大（比如认领的任务依赖此PR的同事）。

**其它审核人**，包括但不限于：需要知悉此处代码改动的人但又不必非要其审核通过的同事、可以从这个PR中学习的同事。

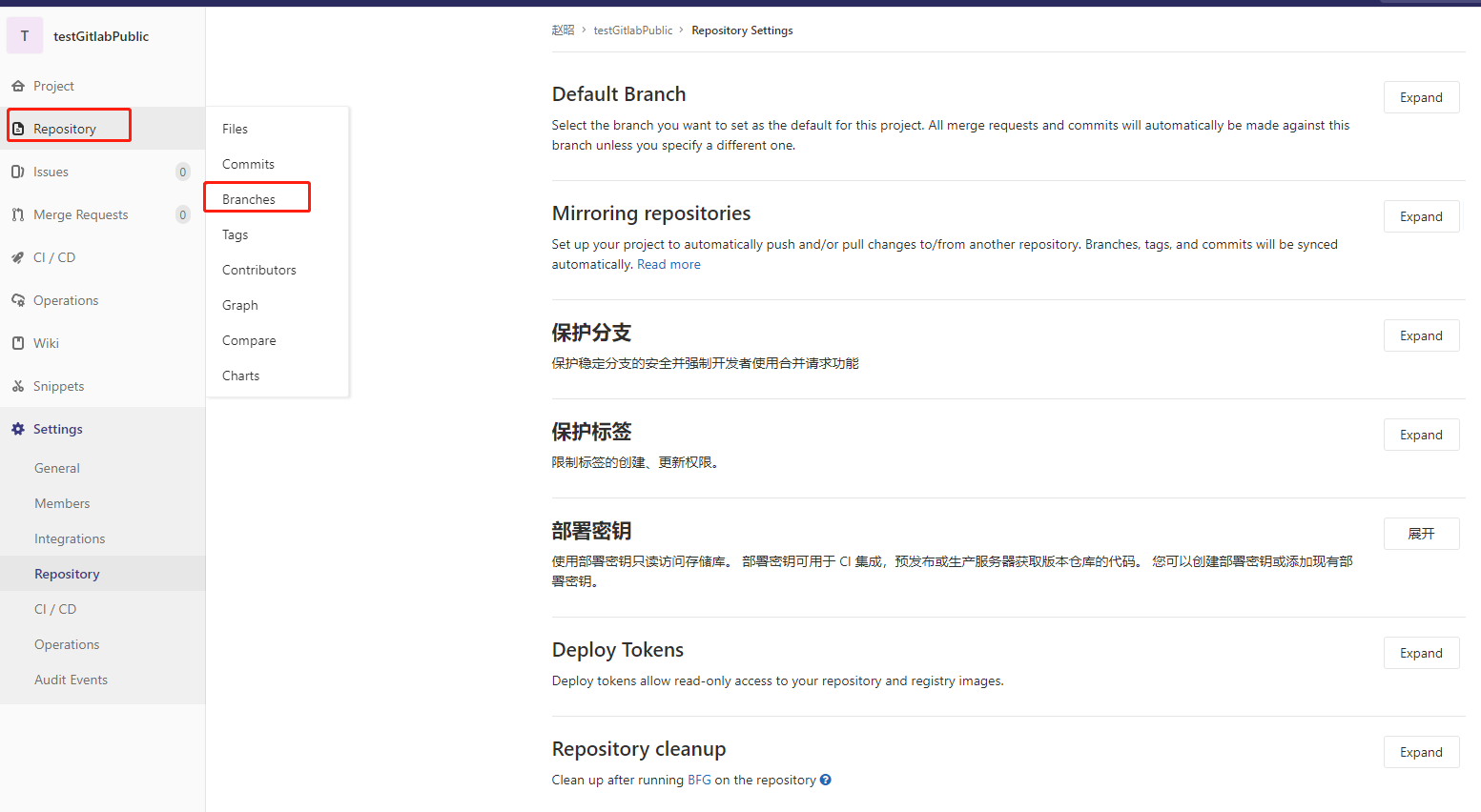
对于Bug修复之类的改动，或者影响较小的改动，审核人员可以授权团队内的某个资深开发人员代表他们进行审核。

#### 2.4.3.6 cherry pick

项目管理者在处理分支功能合入时，有可能会遇到只想合入某些而不是全部功能，这时可用Git中的 git cherry-pick功能来挑选合入的功能。关于cherry-pick可以参考下面的链接：

https://blog.csdn.net/qq\_32452623/article/details/79449534

#### 2.4.3.7 GitLab分支的设置



项目的分支设置：

Default Branch：可以设置项目默认的分支，如果不指定其他分支，默认的合并请求和提交请求都在默认分支上。这里设置为develop。

保护分支：保护稳定分支的安全并强制开发者使用合并请求功能。保护分支的规则如下：

* 除了主程序员外，禁止其他所有人员创建分支
* 除了主程序员外，禁止其他所有人员推送
* 禁止**任何人** 强制推送此分支
* 禁止**任何人** 删除此分支

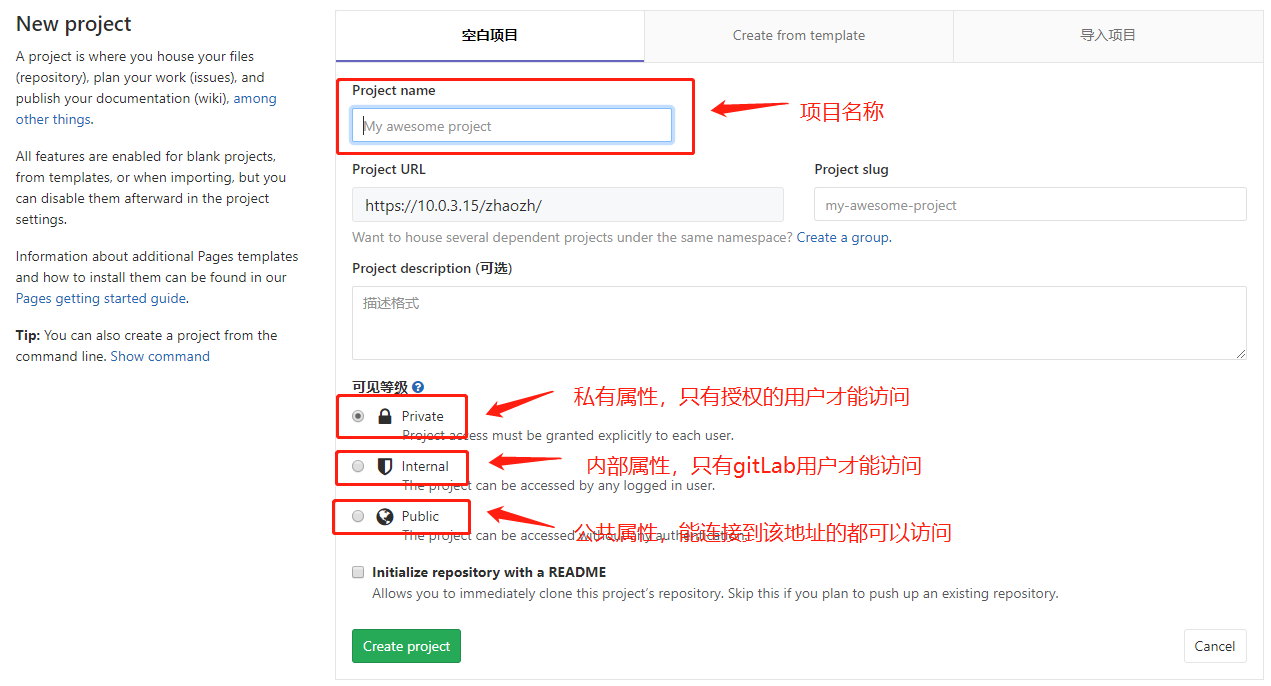
develop和master需被设置为保护分支。

### 2.4.4 技术管理人员（技管部）

技术管理人员主要负责Git中项目的创建，项目人员添加以及权限的管理。所有操作都在GitLab页面中进行。技术管理人员在项目中的权限为Owner，即所有者权限。公司的项目均创建在sinosun用户下面，对外的url格式为：

https://host /{项目组}/{子项目组}/{项目名称}

#### 2.4.4.1 创建项目



也可以先创建Groups后，在Groups中创建项目。

项目按照Groups组创建项目的示例如下：

yqt项目：

|  |
| --- |
| <https://git.sinosun.com/yqt/client/android.git>  <https://git.sinosun.com/yqt/client/ios.git>  <https://git.sinosun.com/yqt/client/windows.git>  https://git.sinosun.com/yqt/server/server.git  yqt是group名，client是子项目组名称，android为项目名称 |

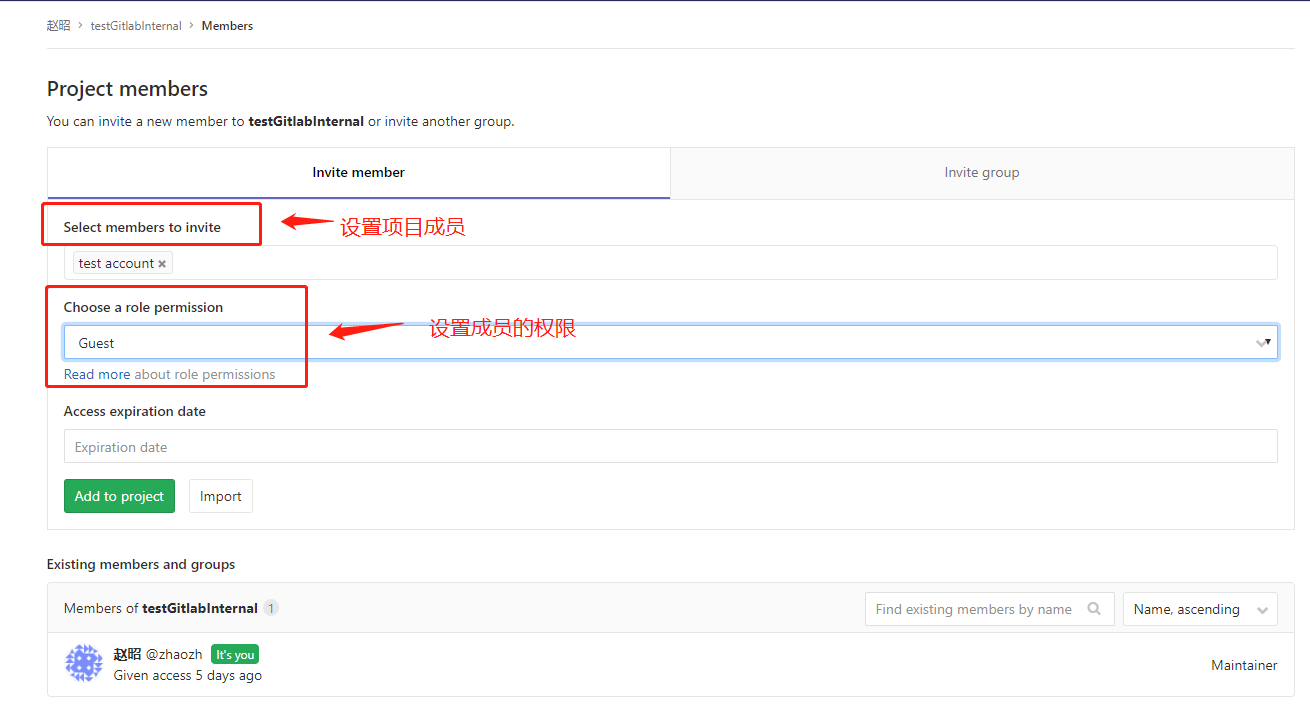
纸纹项目：

|  |
| --- |
| <https://git.sinosun.com/paperprint/client.git>  <https://git.sinosun.com/paperprint/server.git> |

mstp项目：

|  |
| --- |
| <https://git.sinosun.com/mstp/sdk/sdk.git>  <https://git.sinosun.com/mstp/server/server.git> |

#### 2.4.4.2 添加项目成员并赋予权限



项目成员的添加除了使用invite member之外，还可以使用invite group添加。

### 2.4.5 运维人员

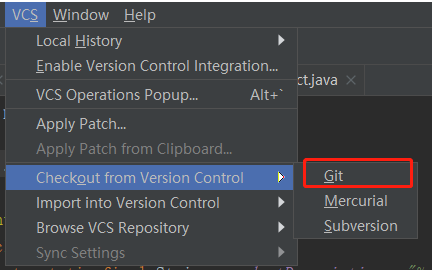
运维人员主要使用root账号来维护GitLab系统，包括GitLab服务状态的检查、GitLab上项目统计、GitLab用户账号的创建、安全认证、数据备份等。

### 2.4.2 开发人员（IDEA）

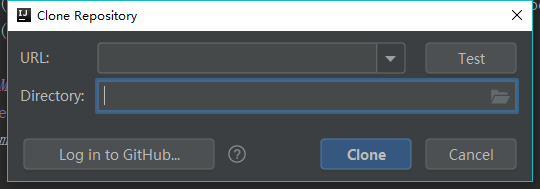
开发人员在使用IDEA工具进行开发时，可以使用IDEA自带的VCS管理，使用IDEA之前需要完成2.4.1.1以及2.4.1.2章节准备工作。



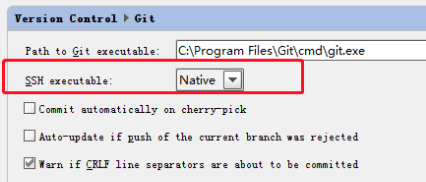
开发人员使用VCS中Checkout from Version Control -> Git选项来获取Git项目



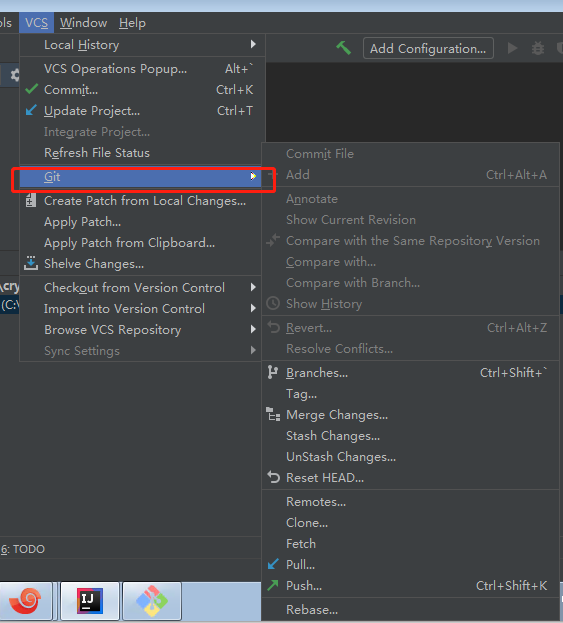
点击Git选项后，URL中填写需要clone的Git仓库地址，填写完成后可以点击Test按钮进行测试，Directory中填写本地保存路径。



需要关注的是对于2019之前的IDEA版本使用兆日的Gitlab时，需要在设置->version Control -> Git 中将 ssh executable更改为Native选项。



仓库clone完成后，可以在IDEA的VCS->Git中查看到常用的Git命令，



## 2.5 Git中大文件的处理

由于Git在每一个commit时都会变动过的文件全部保存外加未变动文件的引用，这样如果在文件系统中有一些大的二进制文件，比如图片、视频等，那么Git的仓库将变得很大，clone这个仓库时也会耗用越来越多的时间。下面提供一些方法来解决这些问题。

### 2.5.1 查看项目中特定分支

**浅克隆：**

为了更快、更节省时间也更节约磁盘空间，第一个解决办法是使用Git进行浅克隆。通过浅克隆可以只克隆某个库最后的历史记录。

|  |
| --- |
| git clone --depth depth remote-url # depth代表要获取的历史提交的深度 |

浅克隆使用时有很多限制：

1. 不能从浅克隆版本库克隆出新的版本库；
2. 其他版本库不能从浅克隆获取提交；
3. 其他版本库不能推送提交到浅克隆版本库；
4. 不要从浅克隆版本库推送提交至其他版本库，除非确认推送的目标版本库包含浅克隆版本库中缺失的全部历史提交，否则会造成目标版本库包含不完整的提交历史导致版本库无法操作；
5. 在浅克隆版本库中执行合并操作时，如果所合并的提交出现在浅克隆历史中，则可以顺利合并，否则会出现大量的冲突，就好像和无关的历史进行合并一样。

由于浅克隆有上述限制，因此浅克隆一般用于对远程版本库的查看和研究。

**只克隆一个分支：**

可以通过只克隆某一个分支来限制历史记录的数量：

|  |
| --- |
| git clone URL --branch branch\_name --single-branch [folder] # 只克隆一个branch分支 |

### 2.5.2 处理巨大二进制资产的库

#### 2.5.2.1 Git gc

Git调用git gc命令进行垃圾回收，此命令会做很多工作：收集所有松散对象并将它们存入packfile，合并这些packfile进一个大的packfile，然后将不被任何commit引用并且已存在一段时间（数月）的对象删除。

|  |
| --- |
| git gc --auto |

#### 2.5.2.2 使用子模块

使用子模块的方式，将项目中的大文件作为子模块关联到项目中。

### 2.5.3 Git-LFS

Git-LFS是Git的一个扩展，并没有改变Git的工作方式。它将用户所标记的大文件保存至另外的仓库，而在主仓库仅保留其轻量级指针。在用户检出版本时，根据指针的变化情况下更新对应的大文件，而不是在本地保存所有版本的大文件。

操作如下：

|  |
| --- |
| git lfs install # 开启lfs功能 git lfs track "\*.exe" # 追踪所有后缀为exe的文件 git lfs track # 查看追踪的文件 git add . # gitattributes文件添加 git commit git push origin # 将gitattributes推送到远端 git lfs ls-files # 可以显示当前跟踪的文件列表 ## 本地添加一个.exe后缀的大文件 git add . # 本地添加一个.exe后缀的文件 git commit  git push origin # 上传到远端 |

Git-LFS对后缀名，具体文件名均可以指定。

### 2.5.4 稀疏检出

稀疏检出就是本地版本库检出时不检出全部，只将指定的文件从本地版本库检出到工作区，而其他未指定的文件则不予检出，这样可以防止拉取无关的代码或文件。

具体做法可以参考链接：

<https://www.jianshu.com/p/547def473ed9>

https://zhgcao.github.io/2016/05/11/git-sparse-checkout/

## 2.6 Git中的常用工具

### 2.6.1 Git Bash

官方提供的Git工具，安装地址：\\fs\install\git\Git-2.21.0-64-bit.exe

### 2.6.2 Git中的gui工具

Gitgui：

|  |
| --- |
| git gui #调出gui工具 |

sourcetree：

下载地址：https://www.sourcetreeapp.com/

TortoiseGit：

下载地址：https://tortoisegit.org/download/

### 2.6.3 Git与IDE工具

Xcode、eclipse、Visual Studio、Visual Studio Code等开发软件均有Git插件。

## 2.7 项目中Git的常用配置

### 2.7.1 gitignore

在使用Git的过程中，一些比如日志，临时文件，编译的中间文件等不要提交到代码仓库，这时就要设置相应的忽略规则，来忽略这些文件的提交。Git提供了.gitignore文件，只要在这个文件中申明哪些文件不希望添加到Git中去，这样当你使用git add .的时候这些文件就会被自动忽略掉。具体使用如下：

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提交时就会自动忽略这些文件。github上的gitignore项目中提供了直接可用的.gitignore文件。

gitignore文件地址：https://github.com/github/gitignore

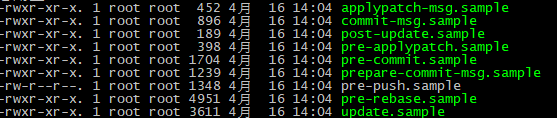
gitignore忽略文件的原则是：

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。

### 2.7.2 git hook

钩子都被存储在 Git 目录下的 hooks 子目录中，也即绝大部分项目中的 .git/hooks 。 当使用 git init初始化一个新版本库时，Git 默认会在这个目录中放置一些示例脚本。这些脚本除了本身可以被调用外，它们还透露了被触发时所传入的参数。Git的钩子程序可以用Ruby、Python或其它语言编写。

在.git/hooks 文件夹下有钩子脚本的范例:



其中commit-msg文件可以规范提交日志

## 2.8 仓库源代码目录图

源代码使用标准的目录结构，目录结构如下

|  |
| --- |
| |- xxx.git 【repo】git仓库  |- conf 【dir】保存仓库配置  |- dependency 【dir】保存微服务之间的依赖关系，以及数据库脚本  |- dependency.yaml 依赖关系描述文件  |- mysql-init-data.sql 初始数据（最新版本），用于执行完 schema.sql 后，导入初始数据。mysql版本  |- mysql-schema.sql 建库脚本（最新版本），全量表结构，用于从 0 开始建库、建表。mysql版本  |- mysql-upgrade.py 数据库升级脚本，包含了每一个版本之间的升级过程。每个版本的升级，使用一个函数来表示。mysql版本  |- oracle-init-data.sql 初始数据（最新版本），用于执行完 schema.sql 后，导入初始数据。oracle版本  |- oracle-schema.sql 建库脚本（最新版本），全量表结构，用于从 0 开始建库、建表。oracle版本  |- oracle-upgrade.py 数据库升级脚本，包含了每一个版本之间的升级过程。每个版本的升级，使用一个函数来表示。oracle版本  |- doc 【dir】接口文档、设计文档等  |- 接口文档  |- 设计文档  |- src 【dir】源码目录  |- test 【dir】存放测试相关内容  |- build.gradle 打包脚本 |

## 2.9 Git日志编写规范

### 2.9.1 Git 中日志的作用

Git中标准化的日志可以给用户使用带来很多便利性，主要如下：

1. 提供历史信息，方便快速浏览

比如，使用下面的命令显示上次发布后的变动，每个commit占据一行。只需要看行首，就知道commit的目的。

|  |
| --- |
| git log <last tag> HEAD --pretty=format:%s # 显示上个tag到目前版本的变动日志 |

1. 可以过滤特定的commit(比如功能开发等)，便于快速查找信息

下面的命令仅显示本次发布新增加的功能。

|  |
| --- |
| git log <last release> HEAD --grep feature # 显示上个release到目前版本的新增功能日志 |

1. 可以从commit 生成ChangeLog

ChangeLog 是发布新版本时，用来说明与上一个版本差异的文档

### 2.9.2 Commit message的格式

提交Commit message 包括三个部分：标题，描述，编号。任何一行都不得超过72个字符（或100个字符）。这是为了避免自动换行影响美观。

#### 2.9.2.1 标题

标题部分只有一行，包括三个字段：类别（必需）、范围（可选）和简述（必需）。

1. 类别

类别用于说明 commit 的类别，只允许使用下面7个标识。

|  |
| --- |
| feat：新功能（feature）  fix：修补bug  docs：文档（documentation）  style： 格式（不影响代码运行的变动）  refactor：重构（即不是新增功能，也不是修改bug的代码变动）  test：增加测试  chore：构建过程或辅助工具的变动 |
|  |

如果类别为feat和fix，则该 commit 需要出现在 Changelog 之中。其他情况（docs、chore、style、refactor、test）由建议是不要放在Changelog中。

1. 范围

用于说明 commit 影响的范围，比如文件模块，消息模块等等，视项目不同而不同。

1. 简述

commit 目的的简短描述，不超过50个字符。

结尾不加句号（.）。

#### 2.9.2.2 提交描述

描述 部分是对本次 commit 的详细描述，可以分成多行。

有两个注意点。

(1) 使用第一人称现在时;

(2) 应该说明代码变动的动机，以及与以前行为的对比。

#### 2.9.2.3 提交编号

提交的禅道编号，新功能填写禅道上的任务编号，故障解决填写对应禅道故障编号。

#### 2.9.2.4 撤销提交

如果当前 commit 用于撤销以前的 commit，则必须以revert:开头，后面跟着被撤销 Commit 的 标题。

|  |
| --- |
| revert: feat(): 添加xxx功能  回滚提交:667ecc1654a317a13331b17617d973392f415f02. |

Body部分的格式是固定的，必须写成此”回滚提交: $hash”的.，其中的hash是被撤销 commit 的 SHA 标识符。

如果当前 commit 与被撤销的 commit，在同一个发布（release）里面，那么它们都不会出现在 Changelog 里面。如果两者在不同的发布，那么当前 commit，会出现在 Changelog 的Reverts小标题下面。

### 2.9.3 日志举例

以一个修改bug的提交日志为例，日志的模板如下：

|  |
| --- |
| fix(消息模块):修改android sdk无法收发消息的故障  //空一行  问题描述：由于使用新的json库解析消息ID字段出错导致该问题  //空一行  禅道编号: bug#12345 |

## 2.10 银行产品在Git中操作

### 2.10.1 银行产品环境准备

银行产品Git操作流程图请参考下图：



上图中的操作Git仓库的操作通过脚本来完成。

#### 2.10.1.1 脚本环境初始化

使用脚本操作Git脚本必须要准备以下环境：

* 1. python 2.7.16
  2. gitlab以及Gitpython的python第三方库
  3. git本地配置(可以参考2.4.1章节配置)

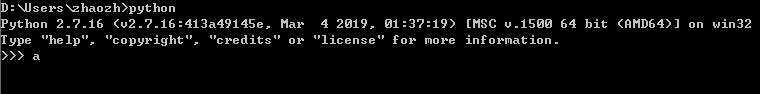
##### 2.10.1.1.1 安装python环境

python版本请选择2.7.16版本(目前写文档时2.x系列的最新版本)，请勿使用3.x系列版本。3.x系列与2.x系列无论在包的使用还是语法上均有不同。

python的windows或linux安装包均由蔡郁田提供。

安装完成python后请将python安装目录添加进系统环境变量中。

打开控制台，输入 python 则显示以下类似画面表示安装成功。以windows为例：



Windows版本python安装后自带pip，linux的pip需要手动安装。安装包同样由蔡郁田提供。安装完成后请将pip添加到环境变量。Pip为python安装第三方包的工具。

##### 2.10.1.1.2 安装使用脚本所需的第三方包

进入脚本目录后depend目录下由gitlab文件夹以及git-python文件夹。

gitlab文件夹为python使用gitlab api的第三方包。

Git-python文件夹内为python使用git命令的第三方包。

1、安装gitlab依赖包(以windows为例，脚本放置在c盘根目录)：

打开cmd在命令行中进入gitlab文件夹: cd /d c: \git-script\depend\gitlab

使用命令将依赖包一次性安装，包名中间‘空格’隔开：

pip install certifi-2019.9.11-py2.py3-none-any.whl chardet-3.0.4-py2.py3-none-any.whl idna-2.8-py2.py3-none-any.whl python-gitlab-1.12.1.tar.gz requests-2.22.0-py2.py3-none-any.whl six-1.12.0-py2.py3-none-any.whl urllib3-1.25.6-py2.py3-none-any.whl

安装成功界面有提示，如果提示pip错误，有可能pip未添加到环境变量。其他错误请根据提示解决。

2、安装git-python依赖包

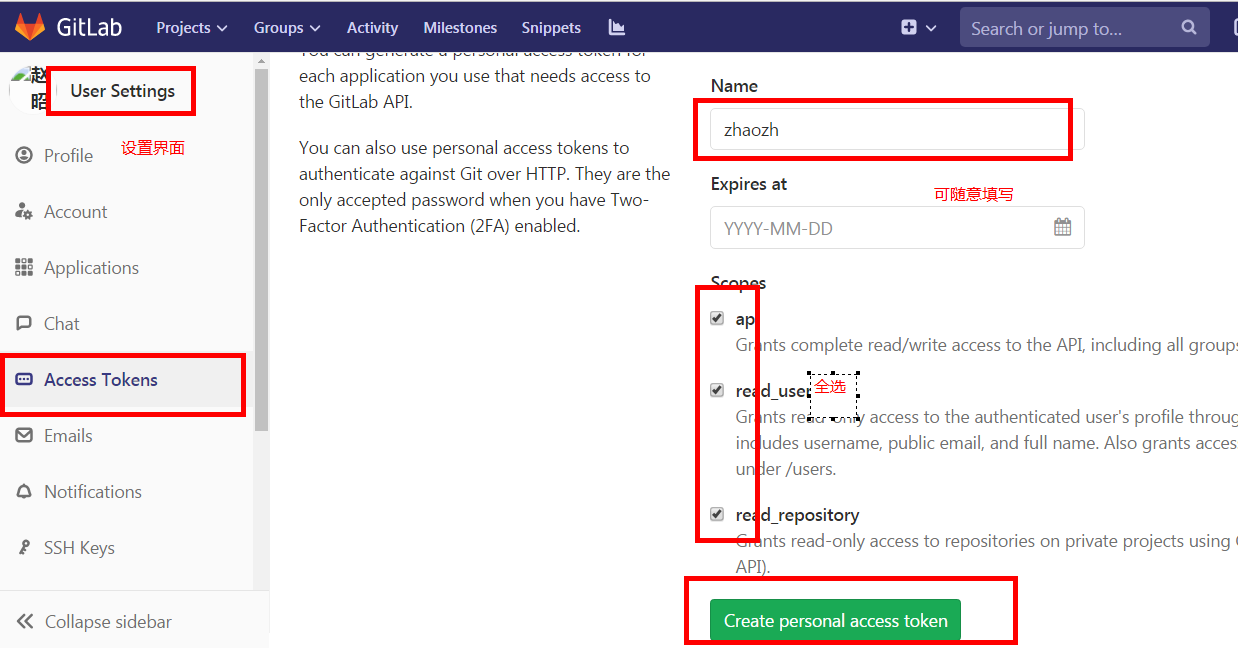
打开cmd在命令行中进入gitlab文件夹: cd /d c: \git-script\depend\ git-python

使用命令将依赖包一次性安装，包名中间‘空格隔开’：

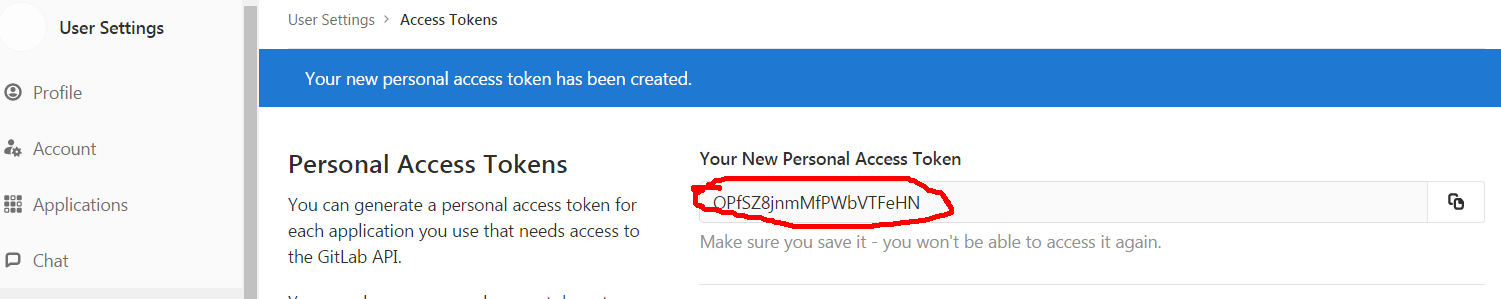
pip install gitdb2-2.0.6-py2.py3-none-any.whl GitPython-2.1.14-py2.py3-none-any.whl smmap2-2.0.5-py2.py3-none-any.whl

##### 2.10.1.1.3 获取access token

access token是api调用gitlab的凭证，获取方式如下图：



点击绿色按钮可以生产access token字符串，请妥善保存。



#### 2.10.1.2 从arsenal产品中fork出银行产品代码

新增一个银行产品后，需要在Git仓库上根据arsenal仓库fork出一份银行产品代码，可以使用脚本batchForkProject.py脚本。

执行 python batchForkProject.py 命令

输入需要fork的arsenal仓库地址以及目的的仓库地址。

fork时有可能出现错误，出现错误后需要脚本使用者手动检查仓库错误原因并修改，错误有可能为一下几种

|  |  |
| --- | --- |
| 错误类型 | 修改方式 |
| develop分支不存在 | 需要基于master分支新建develop分支 |
| .gitmodules文件更新失败 | 需要根据项目情况手动更新.gitmodules文件 |
| 项目为空项目 | 删除空项目 |

#### 2.10.1.3 银行代码自动打包、部署、测试

使用自动化工具，执行batchCloneProject.py，将银行项目代码clone到本地执行自动打包，自动部署以及自动测试

#### 2.10.1.4 银行代码项目外带

通过2.11.3的测试后，该银行产品代码可用于开发。当银行产品代码需要上传到银行的代码仓库时，需要将银行产品代码Clone到本地，执行batchCloneProject.py并压缩为zip包后带到银行现场。

执行 python batchCloneProject.py 命令

#### 2.10.1.5 产品代码上传到银行仓库

银行现场为Git仓库：

产品代码zip包展开后，执行createBankGitRepo.py将代码上传到银行Git仓库。需要注意上传机器也需要配置2.11.1脚本环境。脚本中如果打印错误，请手动解决。

执行 python createBankGitRepo.py命令

银行现场为svn仓库：

产品代码zip包展开后，使用脚本删除所有的.git目录，使用svn工具将代码传到仓库中

#### 2.10.1.6 银行环境验证

上传到银行环境的代码需要自动化验证产品代码能否打包成功，能否部署以及通过自动化测试。

### 2.10.2 银行产品研发场景

完成银行产品环境准备后，可以根据以下方法开始研发过程

#### 2.10.2.1 银行现场研发场景

银行现场的研发场景可以分为以下几类，基本的流程请参考下图：



##### 2.10.2.1.1 银行现场开发网络可以同时连接兆日Git仓库与银行内网Git仓库

在这种场景下，银行前方开发人员在开发完代码后首先与兆日科技Git仓库进行同步，可以使用git remote add 命令增加银行仓库的远端地址，设置远端地址别名，通过git 的命令将代码同步到银行产品Git仓库中。

##### 2.10.2.1.2 银行现场开发网络只能连接银行内网Git仓库

兆日科技研发电脑开发：

这种场景下，银行前方人员需要在兆日科技提供的研发电脑上进行研发，研发过程中在可以与兆日科技Git仓库网络中与该仓库定期同步，完成功能点的开发后将代码传入到银行产品Git仓库，传入的方法可以参考2.10.22章节。

银行提供的虚拟机环境开发：

在这种场景下，银行前方人员在银行提供的虚拟机环境开发，研发人员需要按照功能点的完成情况定期将完成的代码通过跳板机拷贝到外部，然后同步到兆日科技Git仓库，方法可以参考2.10.22章节。

#### 2.10.2.2 银行现场代码如何与兆日公司Git仓库同步

对于需要在银行现场开发的项目，需要保证银行现场代码与兆日公司代码的同步。银行项目开发过程中遇到的开发环境如下：

* 1. 兆日公司环境：参与银行项目的兆日内部开发工程师，提交代码到兆日公司Git仓库
  2. 银行办公环境：银行现场办公环境，该环境通常可以连接兆日公司的Git仓库，同步Git代码
  3. 银行开发环境：银行现场开发环境，该环境在银行内网，无法连接兆日公司Git仓库，开发时需要搭建银行项目Git仓库。

如果银行的开发环境可以连接兆日Git仓库，开发过程与在家里开发一致，这里不做阐述。由于银行开发环境通常无法连接兆日公司Git仓库，以下分场景给出解决方案。

##### 2.10.2.2.1 银行内网代码可以从跳板机定时拷出

研发代码开发在银行开发环境，将内网调试完成的代码定期拷贝到银行办公环境，git中提供了两个命令可以简化同步代码的复杂性。

###### 2.10.2.2.1.1 git bundle 数据打包

git bundle 数据打包，可以把git仓库打包成一个文件，基于该文件可以还原仓库，还原仓库后将该仓库与兆日仓库同步。

|  |
| --- |
| $ git bundle create repo.bundle HEAD master # 打包为master分支上所有提交，也可以指定commitId来打包 |

生成一个名为 repo.bundle 的文件，该文件包含了所有重建该仓库 master 分支所需的数据。

|  |
| --- |
| $ git clone repo.bundle repo # 基于这个文件恢复仓库 |

有时候我们需要只需要打包部分提交

|  |
| --- |
| $ git bundle create commits.bundle master ^9a466c5 # 打包^9a466c5之后的提交 |

打包完成后，使用者检查这个文件是否是一个合法的 Git 包，是否拥有共同的祖先来导入。

|  |
| --- |
| $ git bundle verify |

查看包里可以导入哪些分支

|  |
| --- |
| $ git bundle list-heads |

从bundle包中导入提交，创建othermaster分支

|  |
| --- |
| $ git fetch xxx.bundle master:othermaster |

同步本地master分支与othermaster分支

|  |
| --- |
| $ git merge othermaster |

###### 2.10.2.2.1.2 git format-patch 生成补丁

将银行开发中的提交打包成补丁，然后在办公环境的代码库中打上该补丁，然后同步到兆日科技Git仓库。

git format-patch 将每次的提交生成补丁，比较好的做法是使用git rebase –i 修改commit提交后再生成补丁

git format-patch的命令如下：

|  |
| --- |
| $ git format-patch HEAD^ #生成最近的1次commit的patch  $ git format-patch HEAD^^ #生成最近的2次commit的patch  $ git format-patch HEAD^^^ #生成最近的3次commit的patch  $ git format-patch HEAD^^^^   #生成最近的4次commit的patch  $ git format-patch <r1>..<r2>      #生成两个commit间的修改的patch（包含两个commit. <r1>和<r2>都是具体的commit号)  $ git format-patch -1 <r1>         #生成单个commit的patch  $ git format-patch <r1>            #生成某commit以来的修改patch（不包含该commit）  $ git format-patch --root <r1>　　 #生成从根到r1提交的所有patch |

git am：

|  |
| --- |
| $ git apply --stat 0001-limit-log-function.patch   　  # 查看patch的情况  $ git apply --check 0001-limit-log-function.patch    # 检查patch是否能够打上，如果没有任何输出，则说明无冲突，可以打上  (注：git apply是另外一种打patch的命令，其与git am的区别是，git apply并不会将commit message等打上去，打完patch后需要重新git add和git commit，而git am会直接将patch的所有信息打上去，而且不用重新git add和git commit,author也是patch的author而不是打patch的人)  $ git am 0001-limit-log-function.patch         # 将名字为0001-limit-log-function.patch的patch打上  $ git am --signoff 0001-limit-log-function.patch        # 添加-s或者--signoff，还可以把自己的名字添加为signed off by信息，作用是注明打patch的人是谁，因为有时打patch的人并不是patch的作者  $ git am ~/patch-set/\*.patch　　　　# 将路径~/patch-set/\*.patch 按照先后顺序打上  $ git am --abort                     # 当git am失败时，用以将已经在am过程中打上的patch废弃掉(比如有三个patch，打到第三个patch时有冲突，那么这条命令会把打上的前两个patch丢弃掉，返回没有打patch的状态)  $ git am --resolved            #当git am失败，解决完冲突后，这条命令会接着打patch |

**如果打Patch的过程中发生了冲突（conflicts），怎么办？**

解决patch冲突的过程是：

如果不想打这一系列patch了，直接：git am --abort。

(1) 根据git am失败的信息，找到发生冲突的具体patch文件，然后用命令git apply --reject <patch\_name>，强行打这个patch，发生冲突的部分会保存为.rej文件（例如发生冲突的文件是a.txt，那么运行完这个命令后，发生conflict的部分会保存为a.txt.rej），未发生冲突的部分会成功打上patch

(2) 根据.rej文件，通过编辑该patch文件的方式解决冲突。

(3) 废弃上一条am命令已经打了的patch：git am --abort

(4) 重新打patch：git am ~/patch-set/\*.patchpatch

##### 2.10.2.2.2银行内网代码只能拷入不能拷出

代码开发在银行办公环境开发，可以直接连接兆日公司Git仓库，开发完成后将代码copy到银行内网环境。由于调试在银行内网，会造成研发同事每日需要多次提交copy申请将代码放入内网，开发效率较低。

该方案只适用银行内网代码只能拷入不能拷出场景。