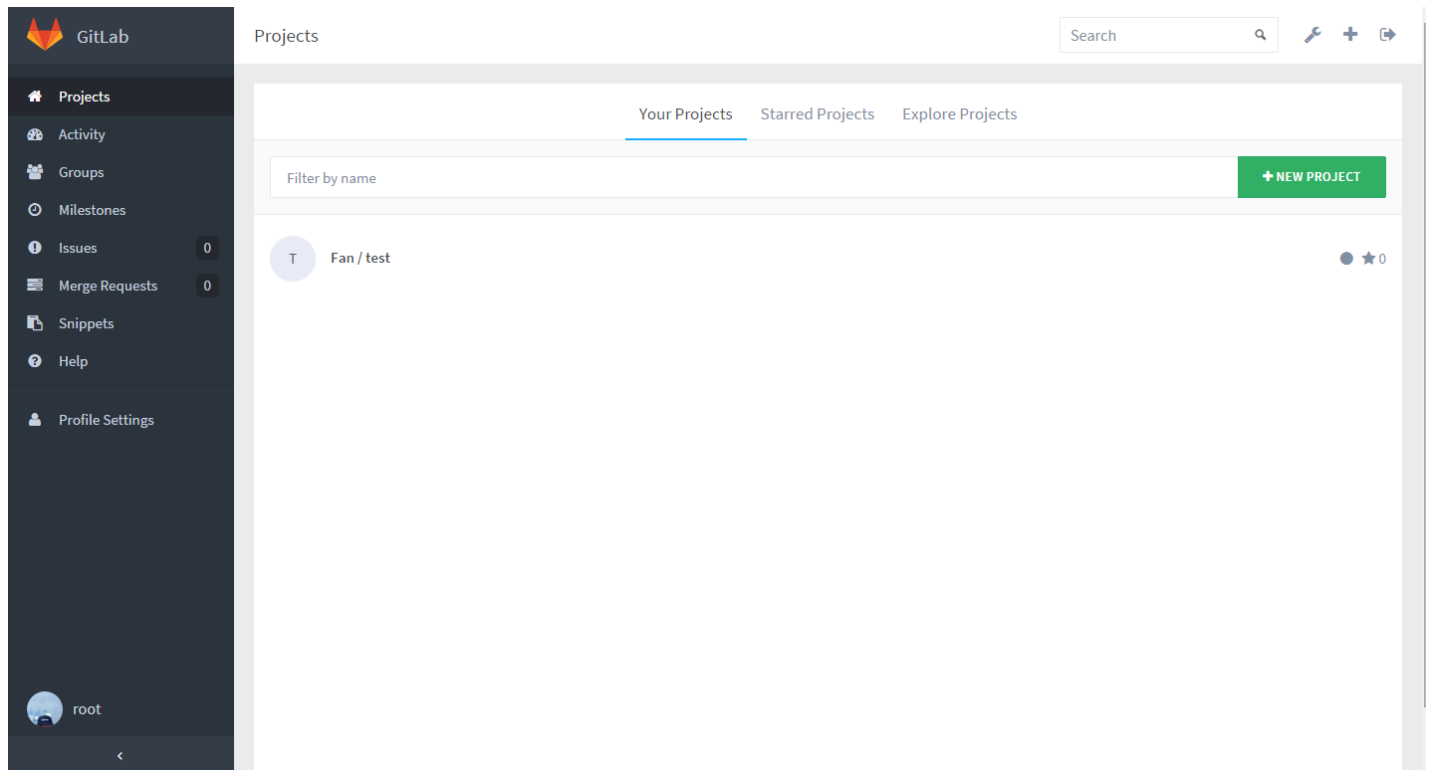


Introduction to GitLab for teams

一、什么是GitLab

GitLab是一个利用Ruby on Rails开发的开源应用程序，使用Git作为代码管理工具，可完美的运行部署在团队自己的服务器上，团队成员可通过Web界面进行访问公开的或者私人项目。GitLab包括Git仓库管理、代码审查、问题跟踪、Wiki等[更多功能](#)。GitLab搭配GitLab CI，能更简单的实现持续集成和自动部署。GitLab还具备团队协同工作功能，如讨论问题，制定里程碑，代码审查等。此外，GitLab还可以方便的集成如JIRA、Slack、Hipchat等系统。



GitLab界面一览

二、为什么选择Git而不是SVN

目前团队使用的版本控制系统是SVN，所有开发者统一在一个SVN版本库中检出代码、开发和提交代码，没有过多的分支，操作相对简单，通过文件夹力度可以轻松管理权限等是它的优势，但是，为什么我们要选择SVN呢？因为——

What SVN cannot do, Git can

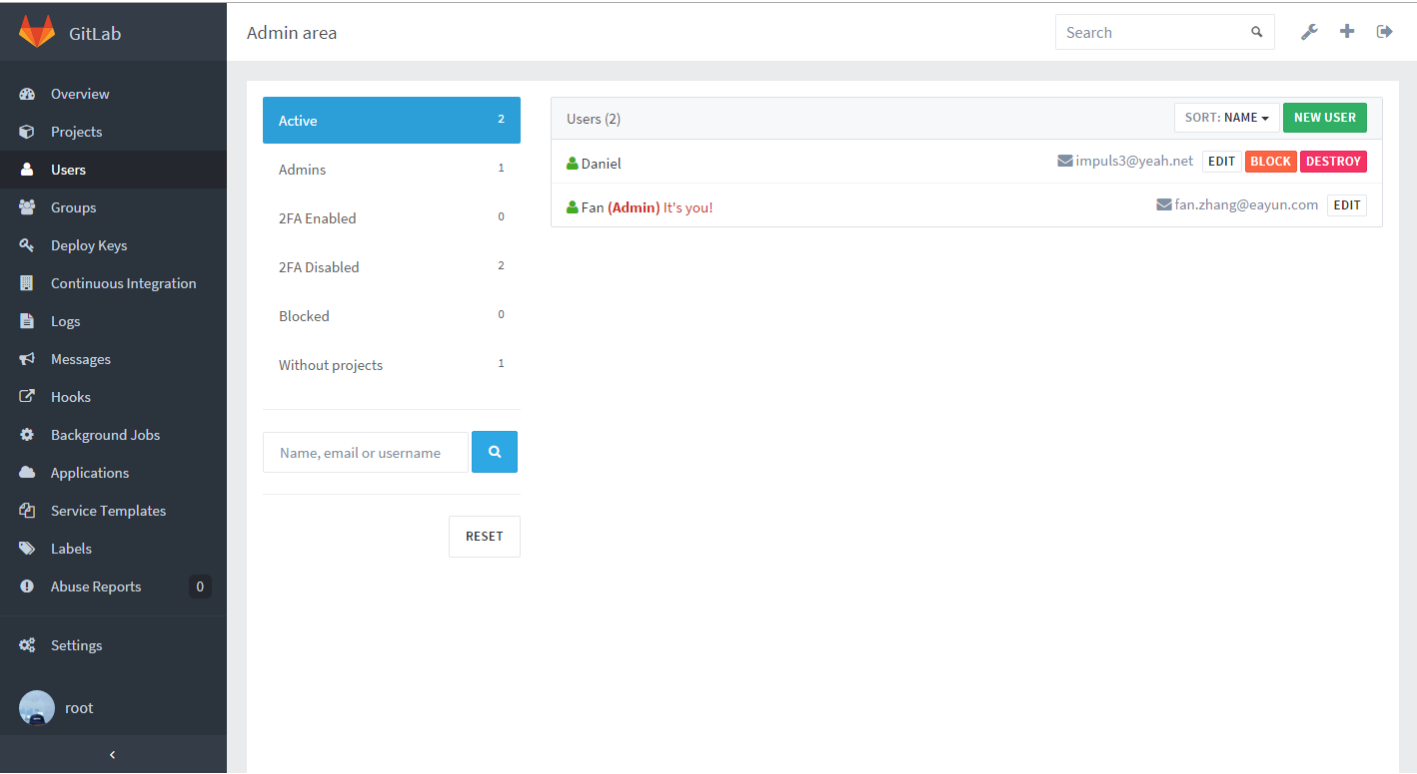
- Git是分布式的，SVN不是。
 - 这是Git区别于其他非分布式的版本控制系统最核心的区别。Git和SVN一样，都有自己的集中式版本库或者服务器，但是，Git更倾向于被使用于分布式模式，也就是每个开发人员在中心版本库/服务器上check out代码后，在自己的机器上克隆一个自己的版本库。在中心版本库/服务器宕机或者身处没有网络的环境，你仍然能够提交文件，查看历史版本记录，创建分支等等。此外，Git包括提交、文件内容等都通过SHA-1哈希保证数据的完整性，从而保证了版本库的安全。
- Git分支和SVN的分支大有不同。
 - SVN中的分支并没有什么特别的，仅仅是版本库中的另外一个目录。而Git的分支却更加灵活有趣。开发者可以在同一个工作目录下快速的切换不同的分支，同时，分支的合并、权限控制相比SVN更精细可控。
- Git可以实现更好的发布控制。针对同一个项目，Git可以设置不同层级的版本库（多版本库），或者通过不同的分支（多分支）实现对发布的控制。
 - 设置只有发布管理员才有权限推送的版本库或者分支，用于稳定发布版本的维护。
 - 设置只有项目经理、模块管理员才有权限推送的版本库或者分支，用于整合测试。
- Git具备对合并更好的支持，更少的冲突，和好的冲突解决。
 - Git的基于DAG（有向非环图）的设计比SVN的线性提交提供更好的合并追踪，避免不必要的冲突，提高工作效率。这是开发者选择Git、抛弃SVN的重要理由。

上面只是列举了几项Git和SVN的不同，也是几项Git略优于SVN的地方。更多的内容，[点击这里](#)进行查看。下面就会针对一些典型场景，结合GitLab做一下操作演示。

三、GitLab典型场景及操作示例

1. 用户管理

在GitLab中，管理员可以早Users中轻松的对用户进行添加、移除、编辑、锁定、权限分配（是否作为Admin访问GitLab）、部署密钥等控制，如下图：



此外，用户也可以通过注册使用系统，如下图：



GitLab Community Edition

Open source software to collaborate on code

Manage git repositories with fine grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki.

Existing user? Sign in

☐ Remember me[Forgot your password?](#)[SIGN IN](#)

New user? Create an account

[SIGN UP](#)

[Didn't receive a confirmation email? Request a new one.](#)

由于GitLab配置了邮件服务，所以注册可通过邮箱进行双重认证（Two-factor Authentication）。

Confirmation instructions ★

EayunCloud

发给 fan.zhang

发件人: EayunCloud <eayuncloud@eayun.cn>

收件人: fan.zhang <fan.zhang@eayun.com>

时间: 2016年1月20日 (周三) 11:30

大小: 2 KB

Welcome Fan!

You can confirm your account through the link below:

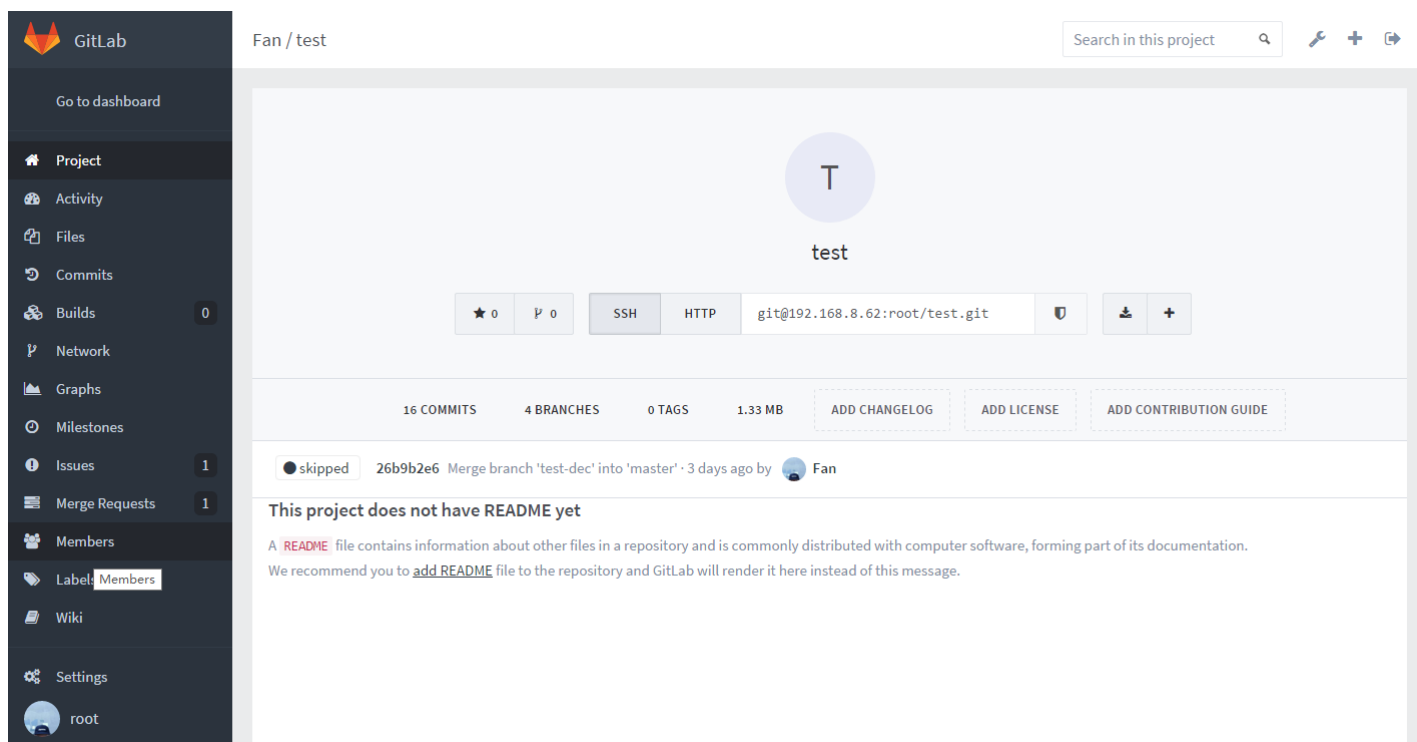
[Confirm your account](#)

2. 权限管理

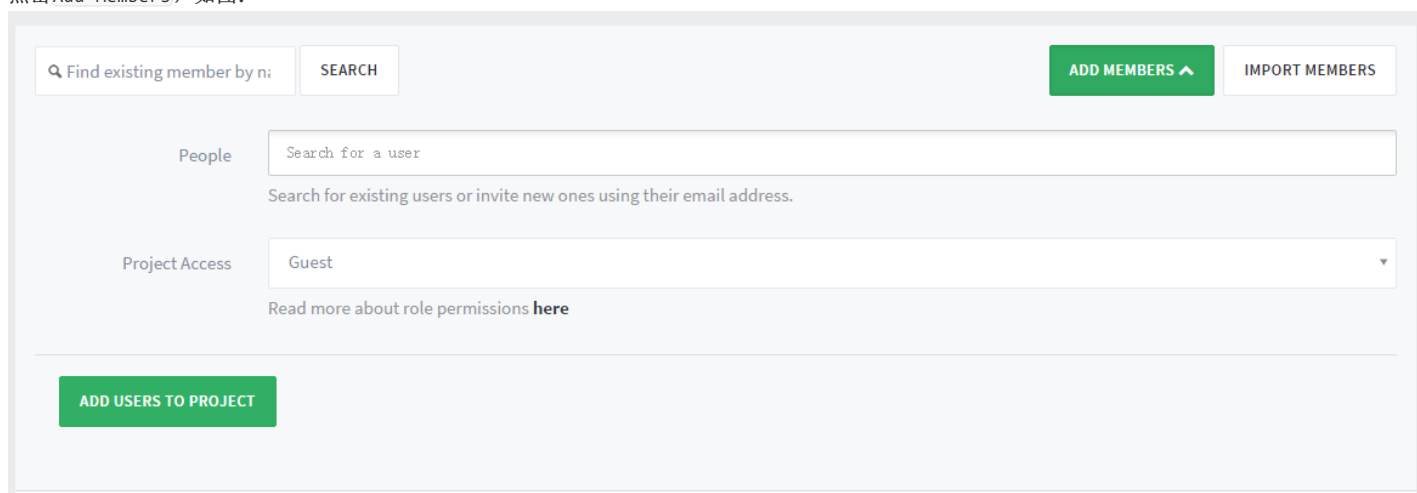
1) 指定成员Level控制项目权限

GitLab通过管理项目中参与用户的级别来管理项目的权限。具备master或者owner权限的用户才可以给项目添加或者导入用户。

首先，打开需要添加成员的项目，点击左侧的Members按钮，如图：



点击Add Members，如图：



添加的成员需要是系统中的用户，Project Access来控制当前成员对该项目的权限。具体权限如下表：

Project

Action	Guest	Reporter	Developer	Master	Owner
Create new issue	✓	✓	✓	✓	✓
Leave comments	✓	✓	✓	✓	✓
Pull project code		✓	✓	✓	✓
Download project		✓	✓	✓	✓
Create code snippets		✓	✓	✓	✓
Manage issue tracker		✓	✓	✓	✓
Manage labels		✓	✓	✓	✓
Manage merge requests			✓	✓	✓
Create new merge request			✓	✓	✓
Create new branches			✓	✓	✓
Push to non-protected branches			✓	✓	✓
Force push to non-protected branches			✓	✓	✓
Remove non-protected branches			✓	✓	✓
Add tags			✓	✓	✓
Write a wiki			✓	✓	✓
Create new milestones				✓	✓
Add new team members				✓	✓
Push to protected branches				✓	✓
Enable/disable branch protection				✓	✓
Turn on/off prot. branch push for devs				✓	✓
Rewrite/remove git tags				✓	✓
Edit project				✓	✓
Add deploy keys to project				✓	✓
Configure project hooks				✓	✓
Switch visibility level					✓
Transfer project to another namespace					✓
Remove project					✓
Force push to protected branches					
Remove protected branches					

Group

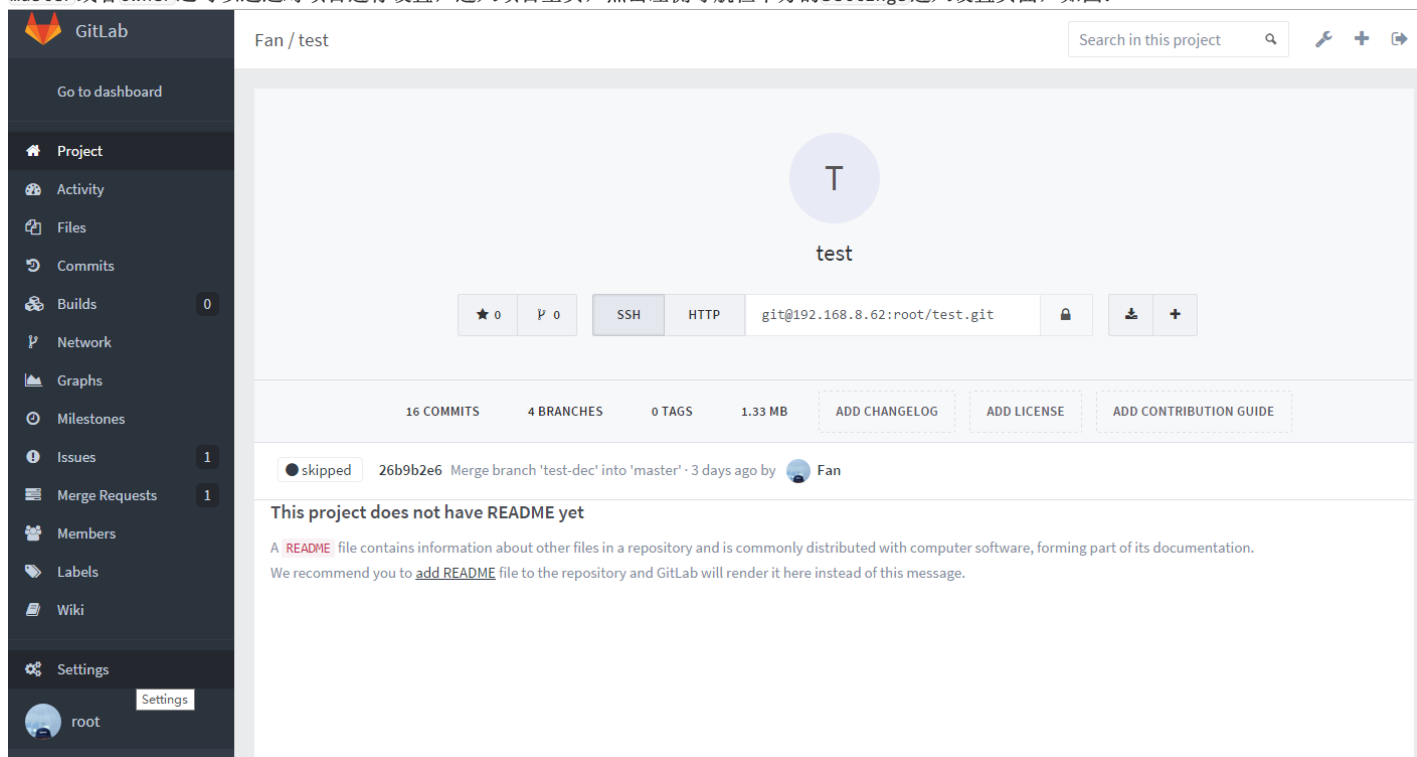
In order for a group to appear as public and be browsable, it must contain at least one public project.

Any user can remove themselves from a group, unless they are the last Owner of the group.

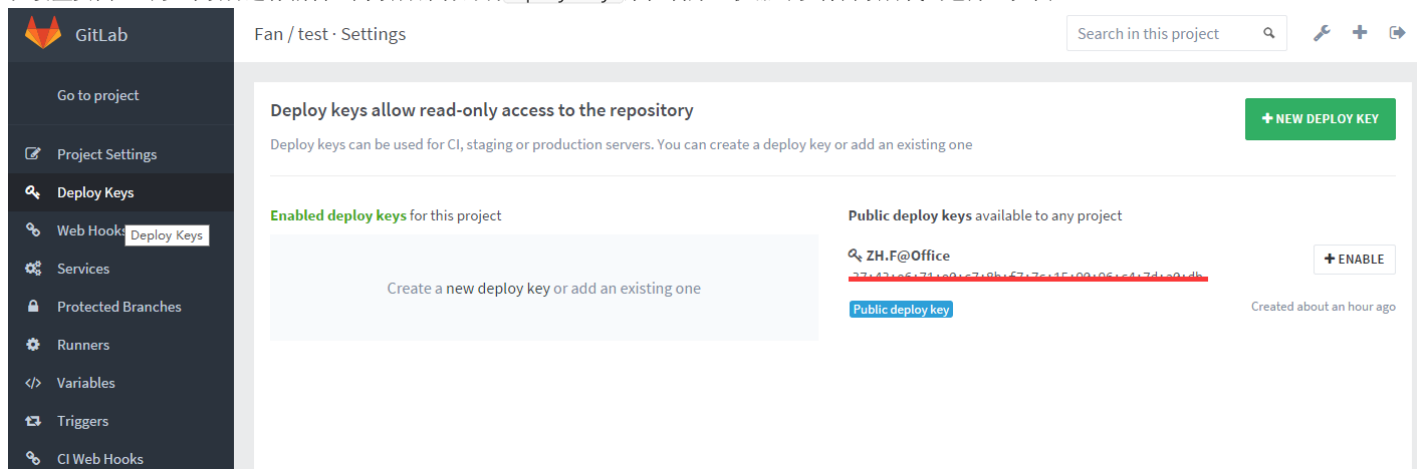
Action	Guest	Reporter	Developer	Master	Owner
Browse group	✓	✓	✓	✓	✓
Edit group					✓
Create project in group				✓	✓
Manage group members					✓
Remove group					✓

2) Deploy Keys控制项目权限

master或者owner还可以通过对项目进行设置，进入项目主页，点击左侧导航栏下方的Settings进入设置页面，如图：



在设置页面，可以对项目进行编辑，为项目部署密钥Deploy Keys来控制那些机器可以访问项目代码仓库，如图：



此时可以添加新的公钥或者在管理员界面添加的公钥针对这个项目进行Enable。

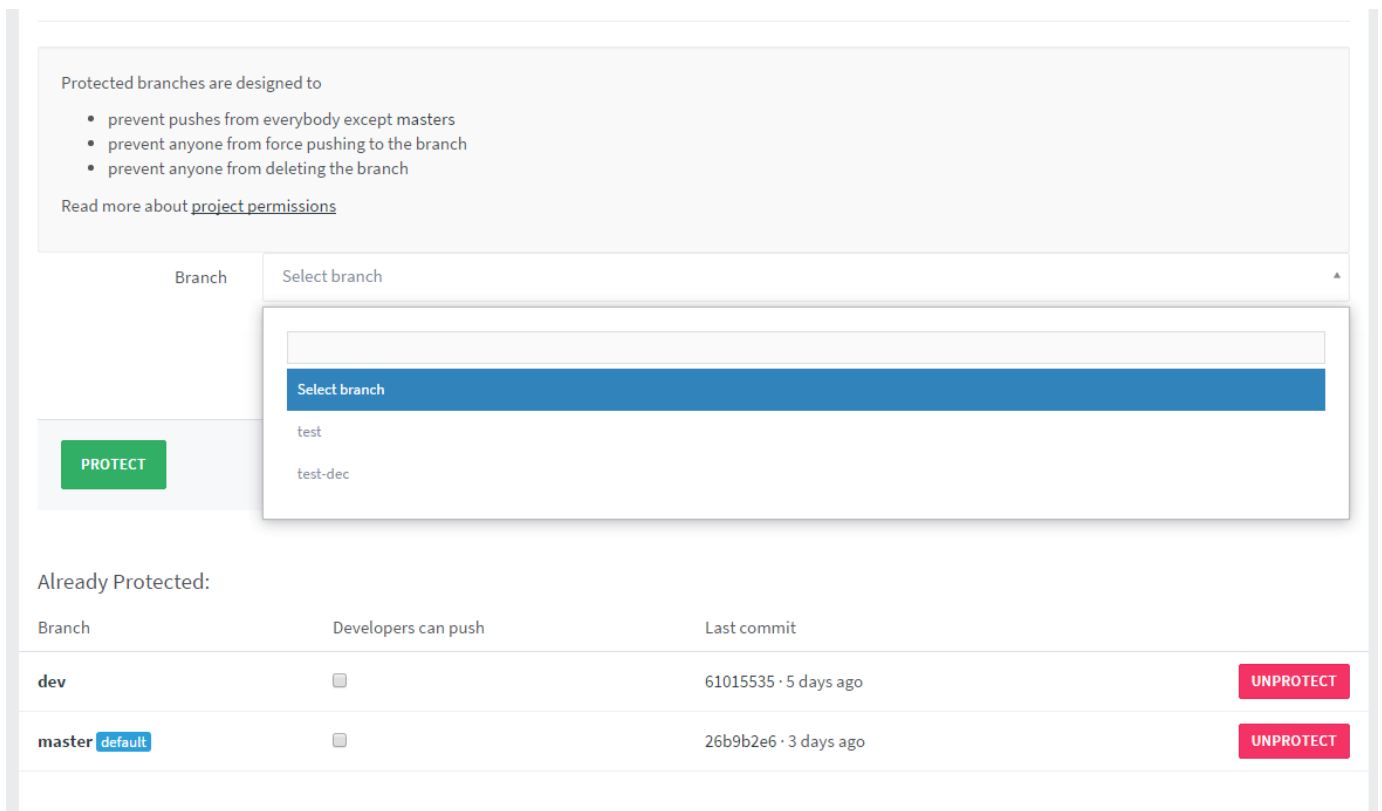
这里的SSH密钥可以结合TortoiseGit提供另一层面的权限控制，不再需要用户帐号这一套体系。比如项目检出分支是按照每个版本一个分支的做法，所有开发者基于这个分支进行开发、测试、修改bugs，所有开发者不需要登录GitLab自己检出分支等操作，所有的分支操作由管理员进行操作。后面的开发者部署章节会基于当前设想的一种 workflow 做详细的操作说明。

3) 保护分支控制分支权限

此外，还可以对项目的分支进行保护，即Protected Branches，受保护的分支设计的目的有三个：

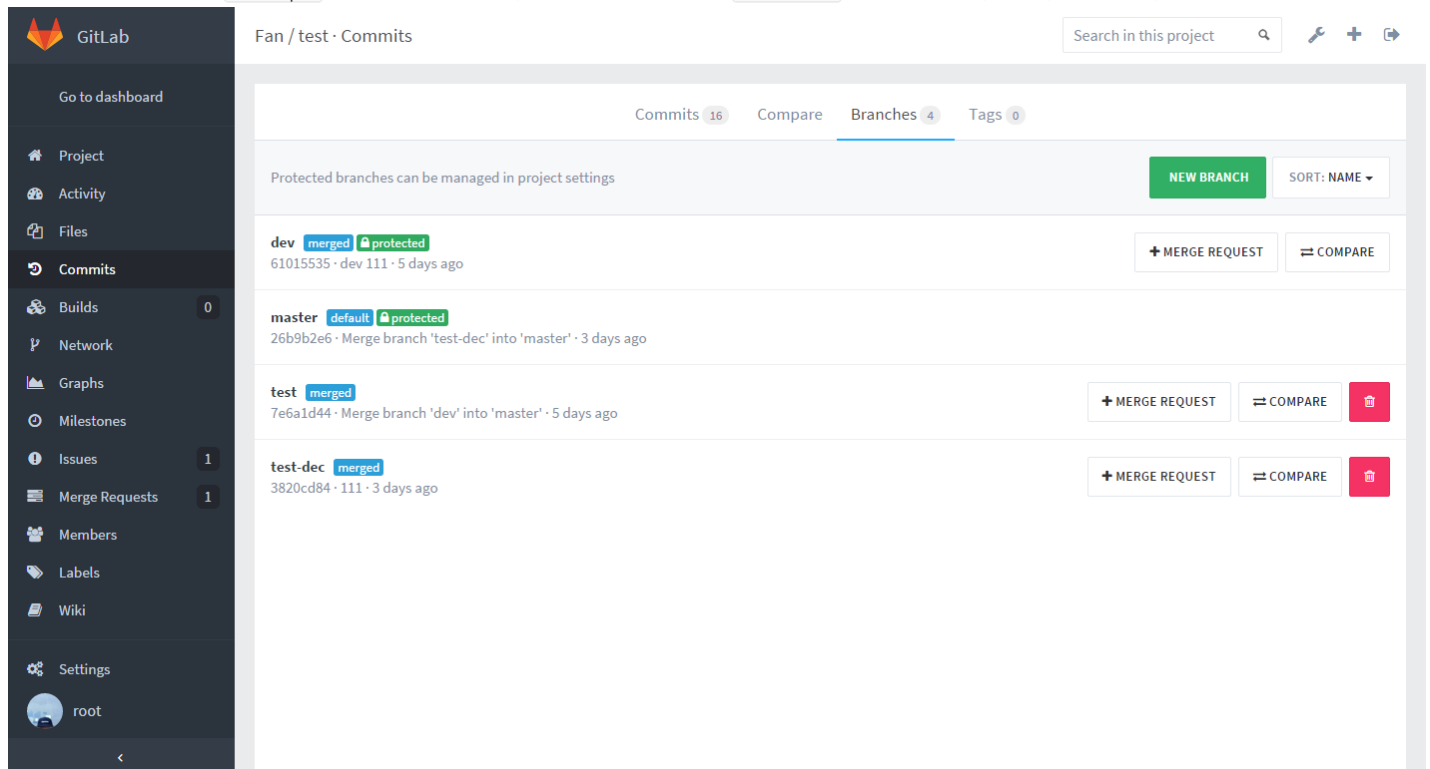
- 除masters之外，防止其他任何人push操作
- 避免任何人该分支中强制push
- 避免任何人删除该分支

关于项目级权限控制，参见上面的具体权限图标。下图是保护分支页面：



3. 分支管理

项目中参与用户级别在Developer及以上就可以创建分支，在项目首页，点击n BRANCHES（n表示当前有几个分支）进入分支管理页面，如下图：



在这里，你可以看到项目现有的分支，可以通过点击MERGE REQUEST对现有的分支发起与某个分支的合并请求，点击COMPARE将现有分支与某个分支进行比较，点击垃圾桶图标对非保护的分支进行删除（注意上图中，dev和master均为protected分支，无法被删除）。

点击NEW BRANCH，进入创建分支页面

New branch

Name for new branch

enter new branch name

Create from

existing branch name, tag or commit SHA

CREATE BRANCH

CANCEL





















选择在某个分支的基础上创建新的分支，创建完毕直接可以在项目的分支中看到该分支。在项目左侧导航的Network可以查看到当前项目的基于某个主分支（可切换选择）的所有分支图形化展示，如图所示：

4. 集成JIRA/Redmine

master或者owner可为项目添加Services，如集成GitLab CI（GitLab的持续集成服务），集成JIRA、Redmine等Issue追踪管理服务等，如图：

Project services

Project services allow you to integrate GitLab with other applications

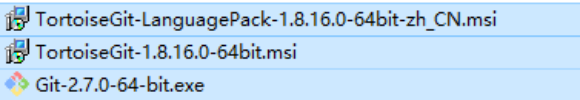
	Service	Description	Last edit
	Asana	Asana - Teamwork without email	6 days ago
	Assembla	Project Management Software (Source Commits Endpoint)	6 days ago
	Atlassian Bamboo CI	A continuous integration and build server	6 days ago
	Buildkite	Continuous integration and deployments	6 days ago
	Campfire	Simple web-based real-time group chat	6 days ago
	Custom Issue Tracker	Custom issue tracker	6 days ago
	Drone CI	Drone is a Continuous Integration platform built on Docker, written in Go	6 days ago
	Emails on push	Email the commits and diff of each push to a list of recipients.	6 days ago
	External Wiki	Replaces the link to the internal wiki with a link to an external wiki.	6 days ago
	Flowdock	Flowdock is a collaboration web app for technical teams.	6 days ago
	Gemnasium	Gemnasium monitors your project dependencies and alerts you about updates and security vulnerabilities.	6 days ago
	GitLab CI	Continuous integration server from GitLab	about 1 hour ago
	HipChat	Private group chat and IM	6 days ago
	Irker (IRC gateway)	Send IRC messages, on update, to a list of recipients through an Irker gateway.	6 days ago
	JIRA	Jira issue tracker	6 days ago
	JetBrains TeamCity CI	A continuous integration and build server	6 days ago
	PivotalTracker	Project Management Software (Source Commits Endpoint)	6 days ago
	Pushover	Pushover makes it easy to get real-time notifications on your Android device, iPhone, iPad, and Desktop.	6 days ago
	Redmine	Redmine issue tracker	4 days ago
	Slack	A team communication tool for the 21st century	6 days ago

详细的内容参见：

- [集成JIRA](#)
- [集成Redmine](#)

四、开发者本地Git环境安装和搭建

开发者本地需要一下三个软件：



其中，TortoiseGit的汉语语言包不做强制要求。

下载地址在这里，可以现在合适你的版本：

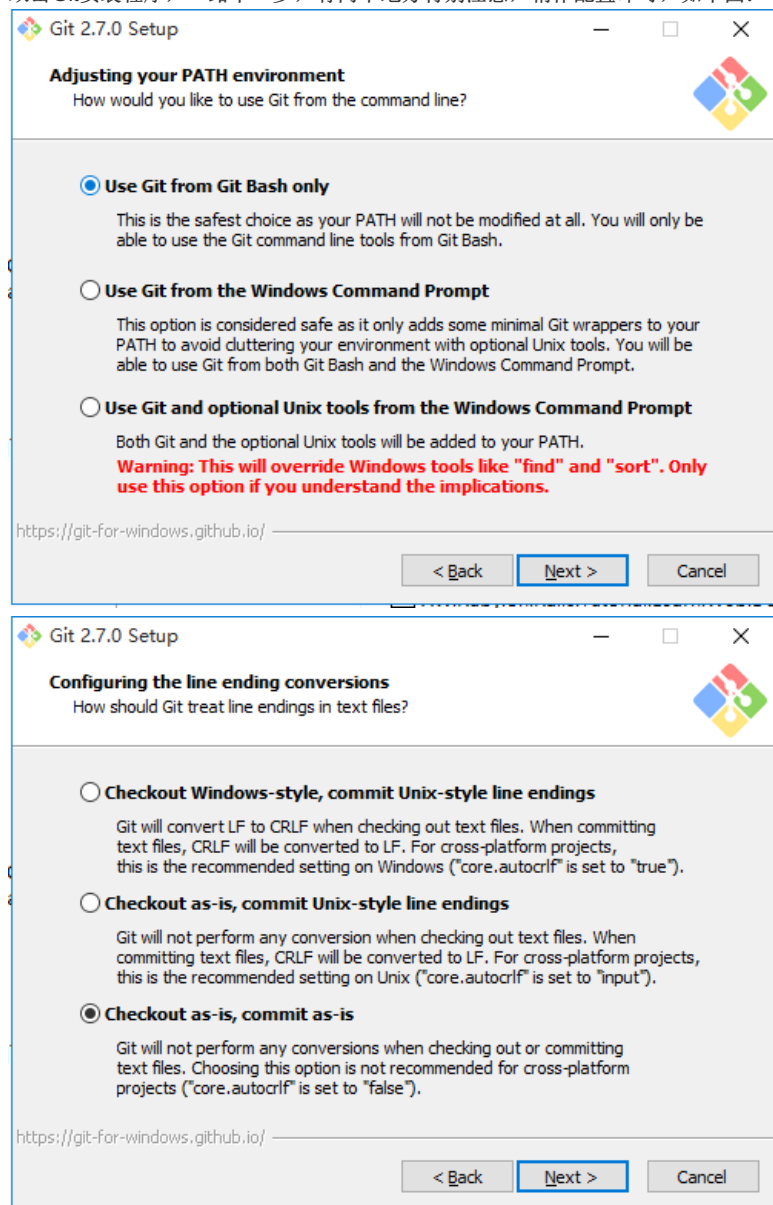
- [Git](#)
- [TortoiseGit](#)

如果网页打不开，可以在我的百度网盘下载（64bit）：

- [点这里](#)，提取密码：35g3

1. Git安装

双击Git安装程序，一路下一步，有两个地方特别注意，稍作配置即可，如下图：



2. TortoiseGit安装

双击TortoiseGit-1.8.16.0-64bit.msi选择安装路径，一路下一步即可完成安装。

3. 生成密钥

在Git安装目录下，找到Git Bash，输入ssh-keygen -t rsa，按照提示输入回车，生成SSH私钥和公钥，如下图：

此时，可以在C:/Users/YourUserName/.ssh/下找到公钥和私钥，如图：

id_rsa	2016/1/26 10:31	文件	2 KB
id_rsa.pub	2016/1/26 10:31	PUB 文件	1 KB

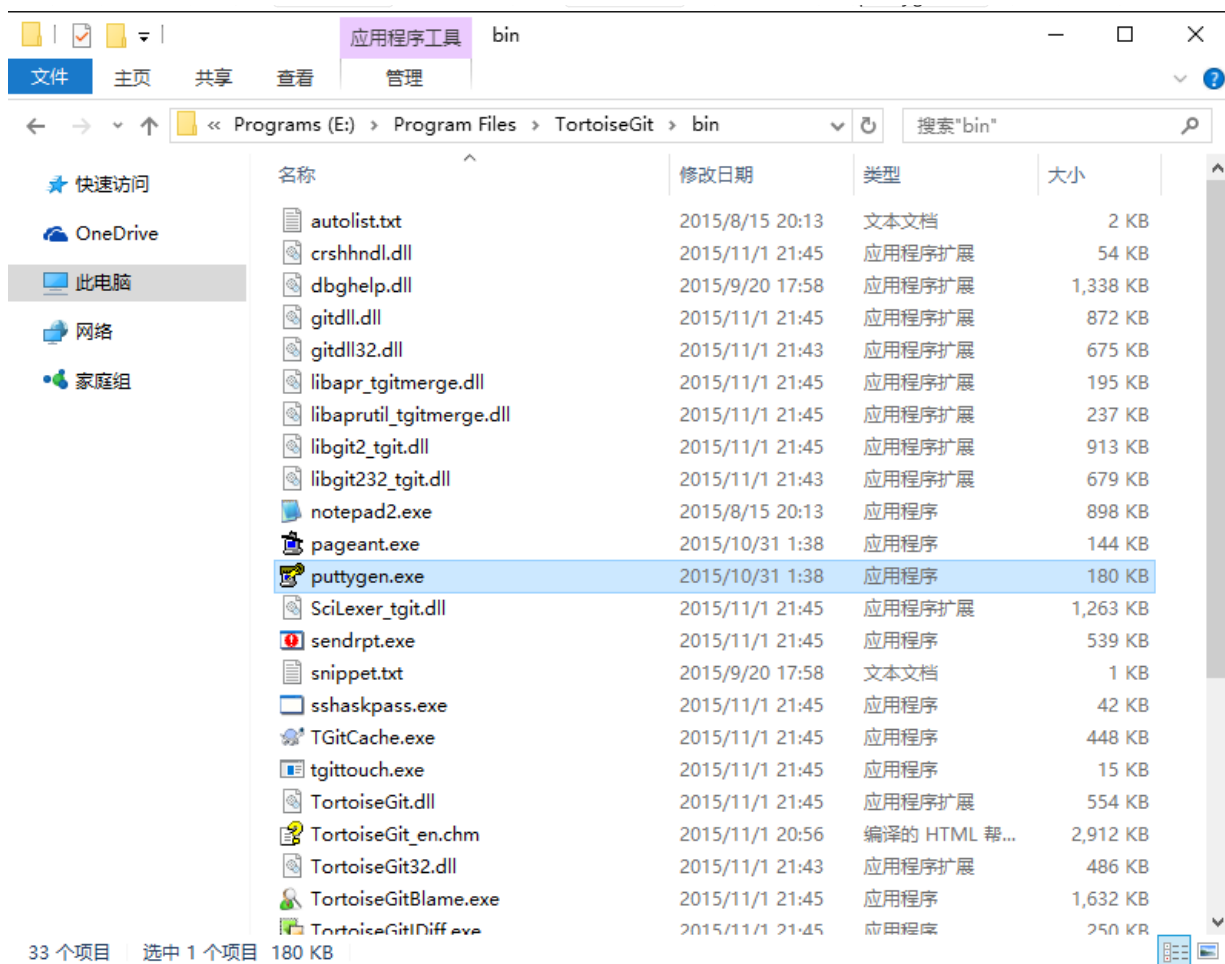
其中，*.pub文件是公钥，请大家改成自己的姓名如zhangfan.pub并发给管理员，由管理员统一对大家的权限进行开通。

在Git Bash下，进行个人用户设置，执行如下命令：

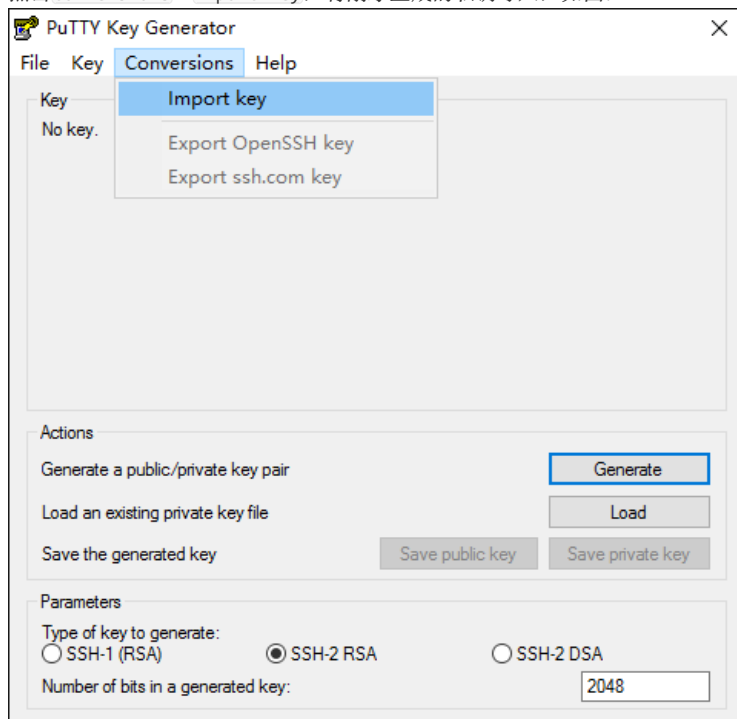
```
git config --global user.name zhangfan
git config --global user.email "fan.zhang@eayun.com"
git config --global push.default simple
```

Tips 记得把用户名和邮箱改成自己的用户名和邮箱。

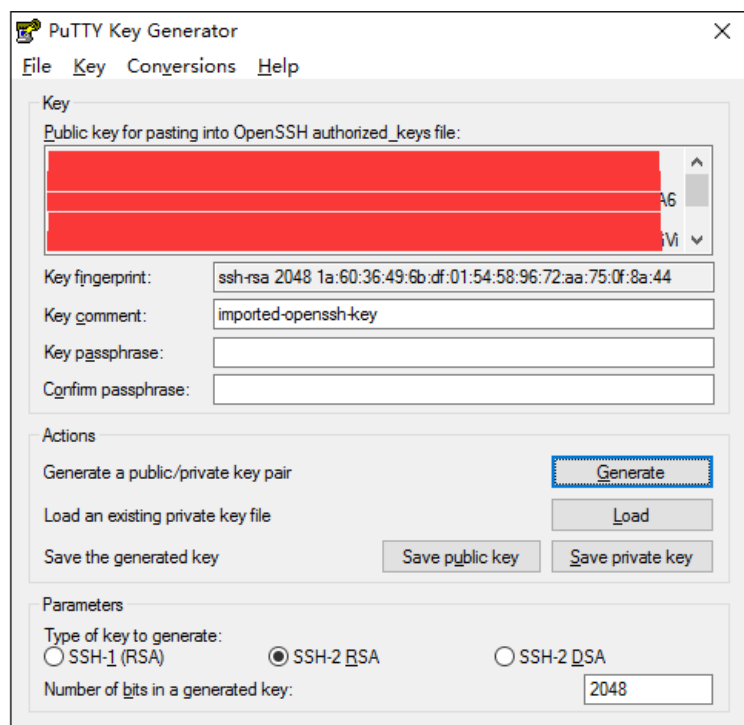
配置完Git后，我们需要使用TortoiseGit生成PPK文件。在TortoiseGit安装目录下，找到puttygen.exe，并运行它，如图：



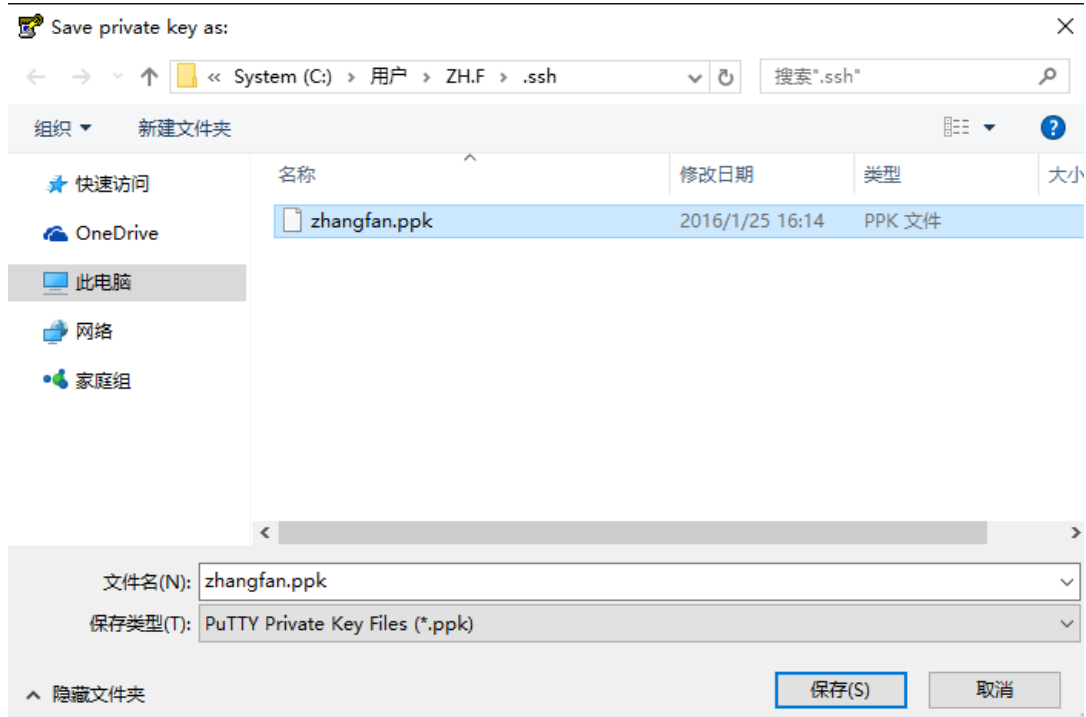
点击Conversions->Import key，将刚才生成的私钥导入，如图：



导入后如图：



点击Save private key，点击确认，以自己的名字命名，讲*.ppk文件保存，如图：



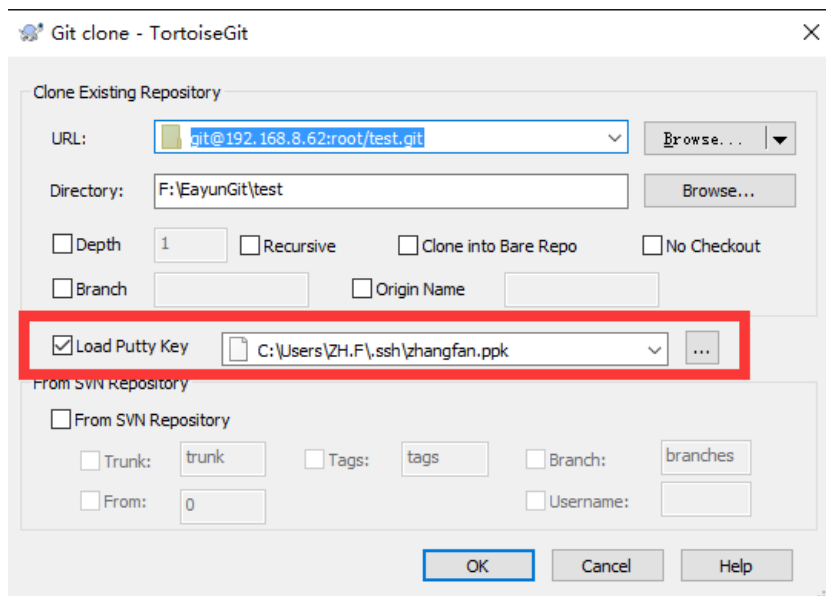
至此，密钥生成工作全部结束，我们可以clone代码库到本地啦~如果重装系统或者更换电脑，还需重新生成。

4. Clone项目

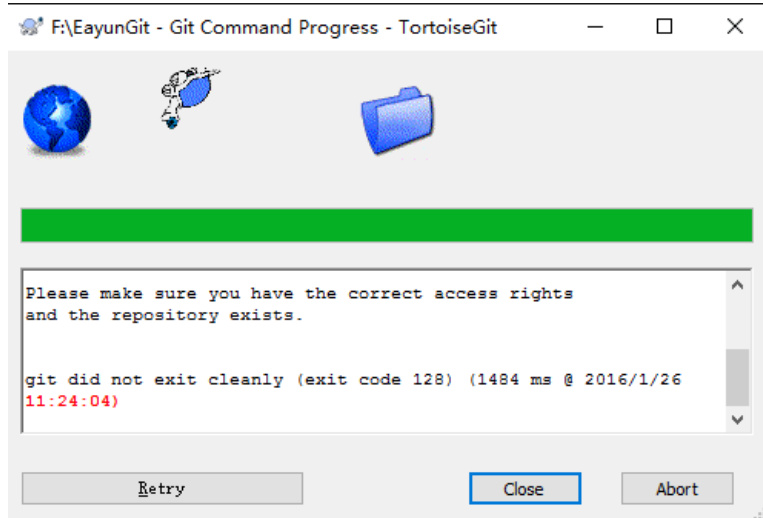
在本地创建文件夹，用作本地的代码库。例如我将文件夹EayunGit作为本地的代码仓库：

EayunDaily	2016/1/22 16:38	↓
EayunGit	2016/1/25 16:18	↓
EayunSVN	创建日期: 2016/1/20 14:16	2016/1/22 13:07 ↓
EayunWorkFile	大小: 4.69 KB	2016/1/25 16:12 ↓
EayunWorksp...	文件夹: test	2016/1/25 14:22 ↓

在该文件夹上，点击鼠标右键，点击Git Clone，输入代码库地址，检查本地代码库目录是否正确，并载入前面生成的ppk文件，如图：

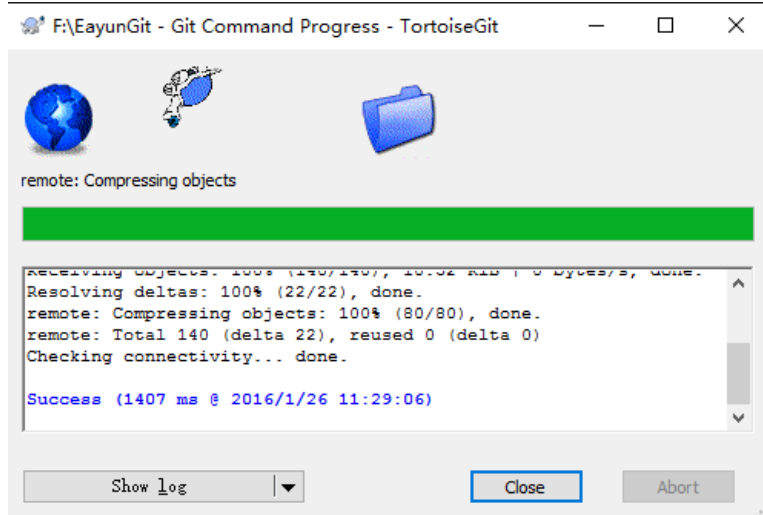


点击确定，如果你并没有该项目的权限，则会遇到下图所示的情况：



请联系管理员。

如果管理员给你开通了权限（即前面提到的Enable deploy key），且本地代码仓库并没有克隆过该项目代码库，则可以正常执行，如图：



此时，本地代码库克隆完毕。

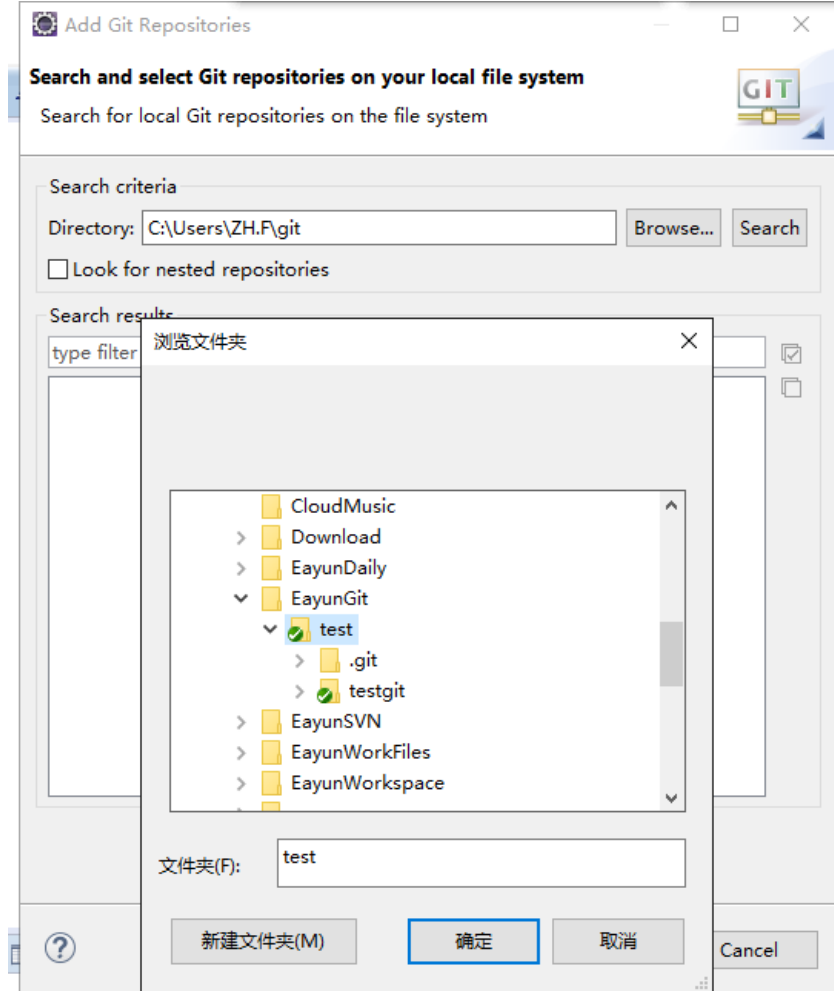
5. 导入IDE中开发和提交

打开IDE（这里以Eclipse/MyEclipse为例，IntelliJ IDEA不做介绍），依次点击Window->Show View->Git->Git Repositories，打开Git资源库视图。

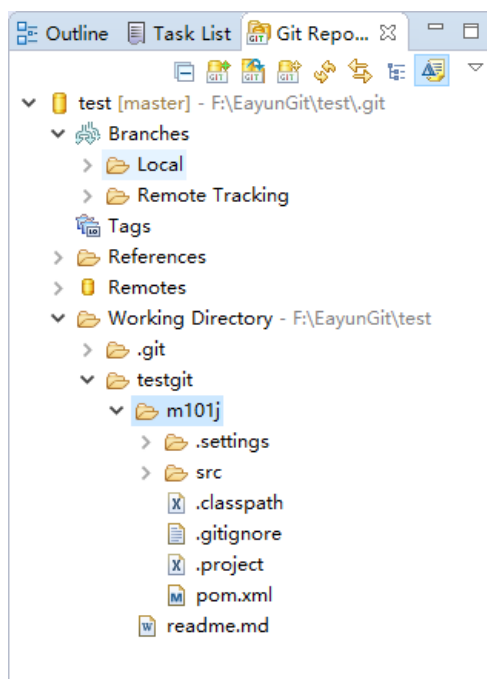
Select one of the following to add a repository to this view:

-  [Add an existing local Git repository](#)
-  [Clone a Git repository](#)
-  [Create a new local Git repository](#)

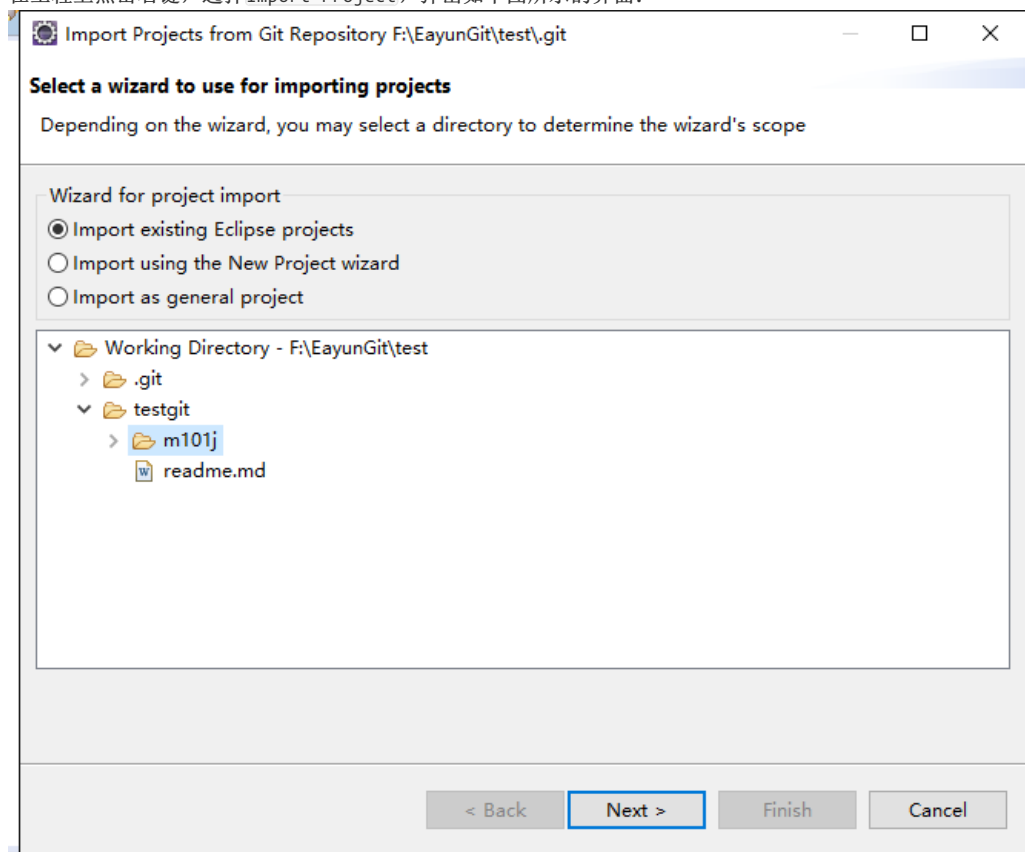
点击Add an existing local Git repository（添加本地Git资源库），选中上面配置的本地代码库，如图：



添加完毕后，可以看到本地的代码库树形结构展示（仅包含一个测试项目），如图：



在工程上点击右键，选择Import Project，弹出如下图所示的界面：



将项目导入到IDE中就可以进行开发了，提交、同步等操作和SVN类似，不同与SVN的是，Git中的提交细分为两步：

- commit 仅把修改commit到本地代码库
- push to upstream 讲本地的提交push到远程Git仓库

下面举一个提交的例子：

首先，在本地对代码进行了变更，此时在工程前面多了一个>表示工程有变更：

```

m101j [test master 11]
├── src/main/java
│   ├── edu.m101j
│   │   └── edu.m101j.week2
│   │       └── Homework.java
│   │           └── Homework
│   ├── src/test/java
│   ├── JRE System Library [J2SE-1.5]
│   ├── Maven Dependencies
│   ├── src
│   ├── target
│   └── pom.xml

```

接着，我们讲变更提交：

Commit Changes to Git Repository

Commit message

commit to master

Author: zhangfan <fan.zhang@eayun.com>

Committer: zhangfan <fan.zhang@eayun.com>

Files (1/1)

type filter text

Status	Path
<input checked="" type="checkbox"/>	testgit/m101j/src/main/java/edu/m101j/week2/Homework.java

Open [Git Staging](#) view

Commit and Push **Commit** Cancel

输入提交信息（强烈建议每次提交都注明提交信息），注意，在Author和Committer部分可以看到跟前面我们配置的git --global user.name和user.email一致。可以选择点击Commit提交到本地而不Push到远端。

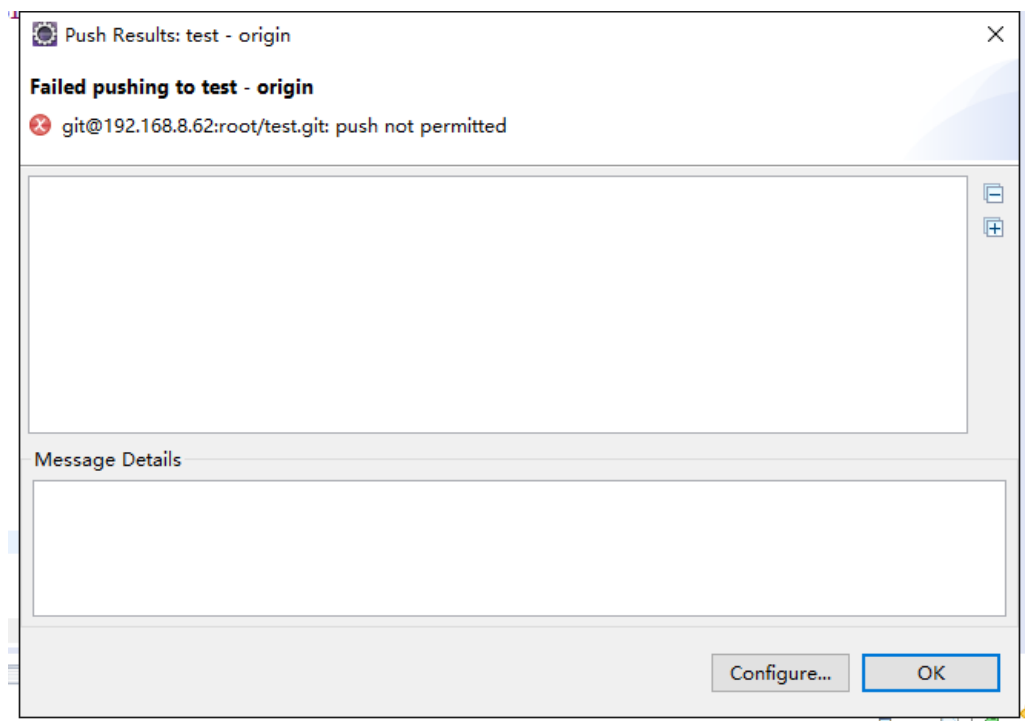
提交后发现工程名字后面括号中有一个12，表示本地有两个提交未Push到远端：

```

m101j [test master 12]
├── src/main/java
│   ├── edu.m101j
│   │   └── edu.m101j.week2
│   │       └── Homework.java
│   │           └── Homework
│   ├── src/test/java
│   ├── JRE System Library [J2SE-1.5]
│   ├── Maven Dependencies
│   ├── src
│   ├── target
│   └── pom.xml

```

让我们执行下Push操作，由于当前分支是master，这是一个受保护的分支，所以Push to upstream会失败：



那么接下来，就要说一下切换分支。假设这样一种应用场景和工作流的组织分配——

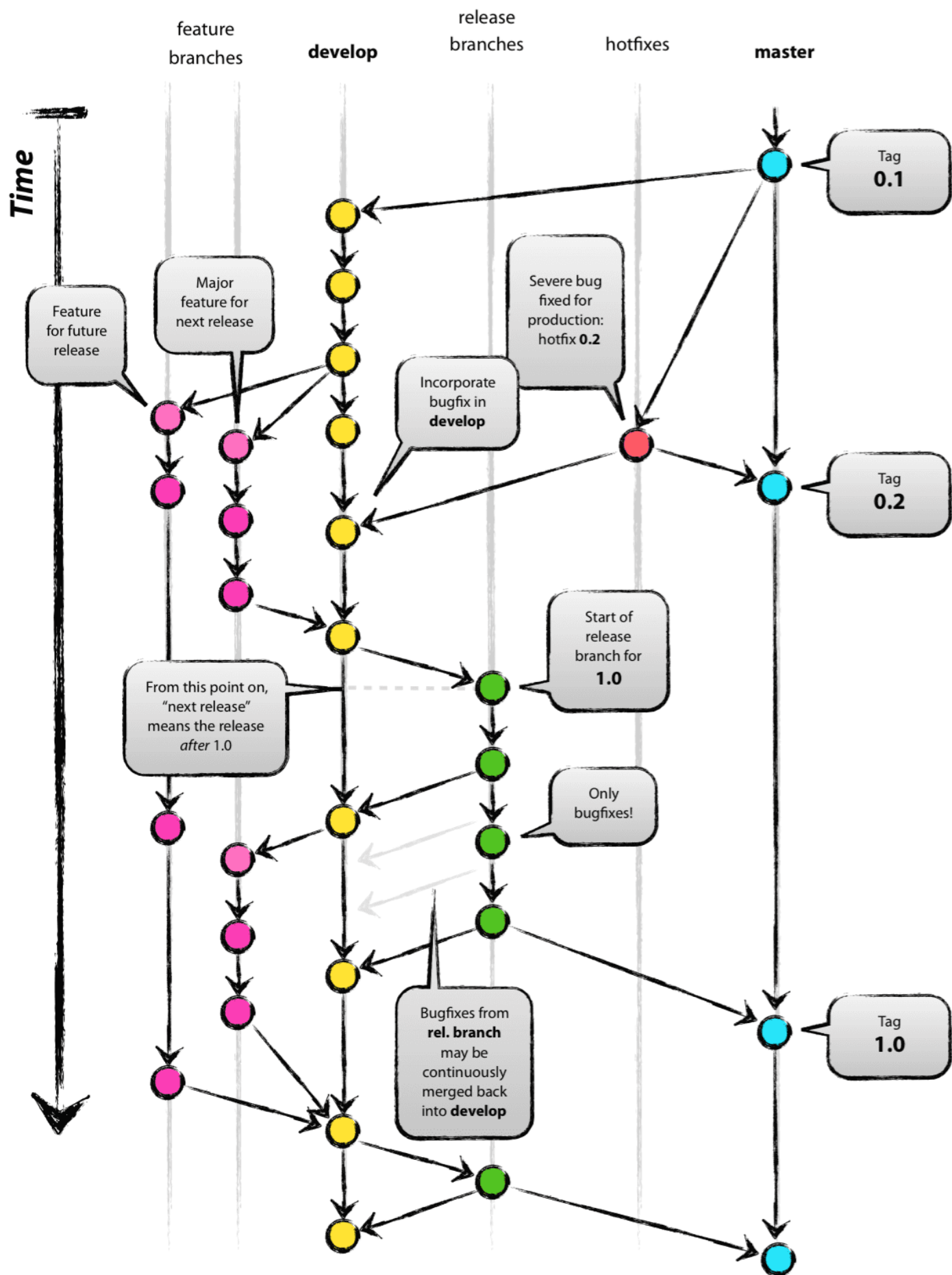
- `master` 是稳定版本分支，与线上版本完全一致；
- `dev` 表示开发分支，基于 `master` 分支检出，用于各版本分支的拉取和合并；
- 各版本分支，都是基于 `dev` 分支，一个版本的分支生命周期从开发、测试到测试完毕持续。

所有的分支检出合并操作均由管理员进行操作，开发开始前，管理员准备好该版本的分支，所有开发者切换到该分支上进行开发，测试和修复bug过程均在这一个分支上进行（可以理解为这个分支等同于之前的SVN），待测试完毕，讲该分支设置为Protected Branch，所有人没有Push权限，待管理员讲分支合并完毕，在拉出新迭代版本的分支前，所有人可以在dev分支上进行提交变更。新的版本分支拉取之后，新的开发任务就如此进行下去。中间过程也可以把Code Review进行起来，等等。

当然，工作流的使用还可以按实际需求灵活安排。

五、一个成功的Git branch模型

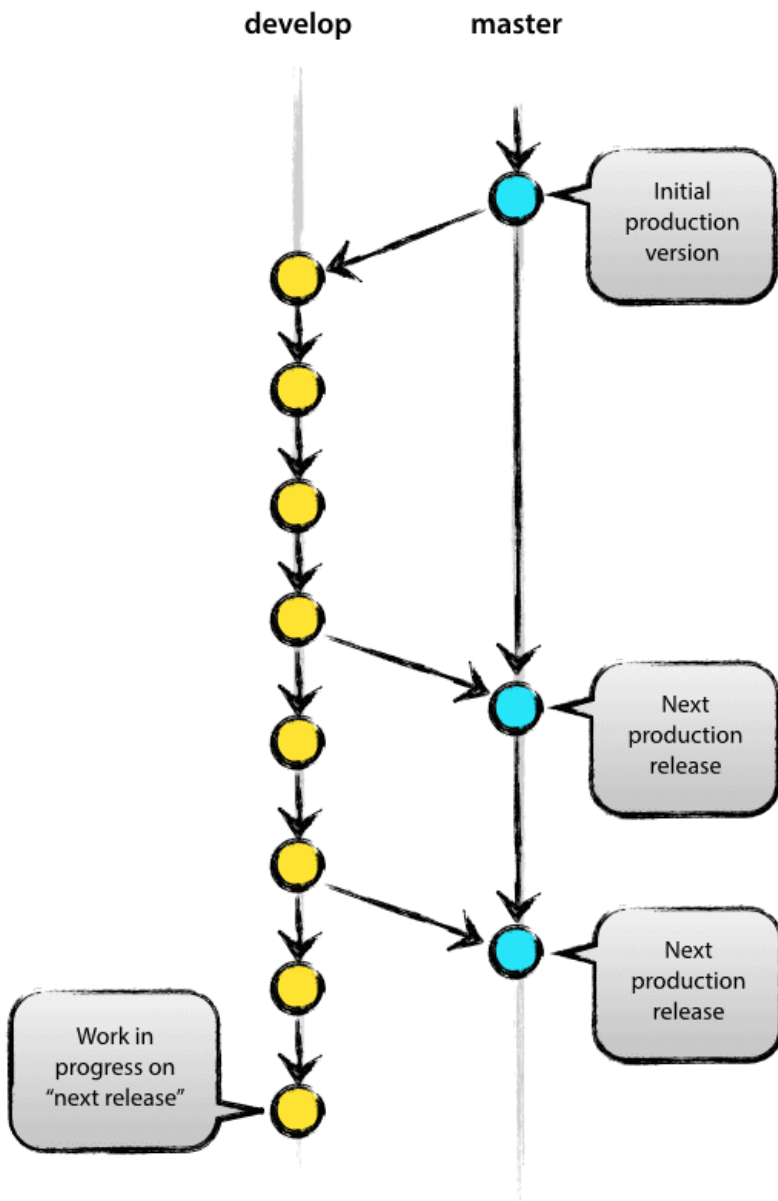
先来张图：



1. 主要分支

在这个模型中，中央仓库持有两个生命周期无限长的主要分支：

- `master`
- `develop`



我们认为，`origin/master`这个主要分支上，源码的HEAD永远保持生产就绪的状态。`origin/develop`这个主要分支的源码HEAD则永远代表了最新提交的开发变更，所以也被称为是“集成分支”。该分支可以用于每晚的自动化构建所使用。

当`develop`分支的代码能够到达一个稳定点，并且已经准备好进行版本发布，所以的变更应当合并到`master`上，并且用版本号标注。具体操作后详细谈到。

因此，每当变更最终合并到`master`分支，这就是一个新的生产版本。对待这个分支，要极其严格，所以理论上讲，可以使用一个Git hook脚本来进行自动化构建，每当有新内容提交到`master`，脚本自动将软件发布到成产环境。

2. 支持性分支

在这个模型中，有各类支持性分支来协助团队成员的并行开发，方便跟踪功能特性，准备生产版本和快速修复生产问题。与主要分支不同的是，这三个支持性分支是有限生命周期的，最终会被移除。

这里使用的三类分支分别是：

- 功能特性分支（Feature branches）

- 发布用分支（`Release branches`）
- 补丁分支（`Hotfix branches`）

这三类分支目的明确，所以对于这些分支的源分支和合并的目标分支具有十分严格的规则。当然，这三类分支也仅仅是分支而已，并没有特别的地方。

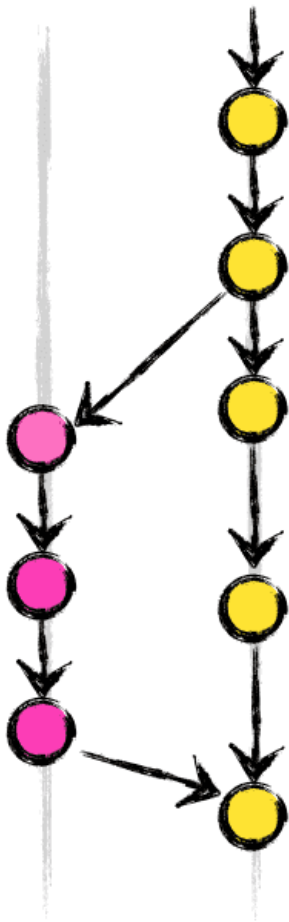
1) 功能特性分支

分支来源: `develop`

合并目标: `develop`

命名惯例: 除`master`、`develop`、`release-*`或者`hotfix-*`之外的任何名字均可

feature
branches **develop**



功能特性分支（或者有时被称作专题分支）被用于开发接下来或者将来版本的新功能、新特性。当开始开发一项功能时，目标发布用分支并未明确，但只要功能在开发中，这个分支就存在，最终会合并回`develop`（意味着即将发布的版本中一定会包含该功能）或者被废弃（这当然是一种令人十分失望的情况）。

功能特性分支仅存在与开发的代码仓库，并不在`origin`。

创建一个功能特性分支

当着手开发新功能时，先在开发分支上检出新分支：

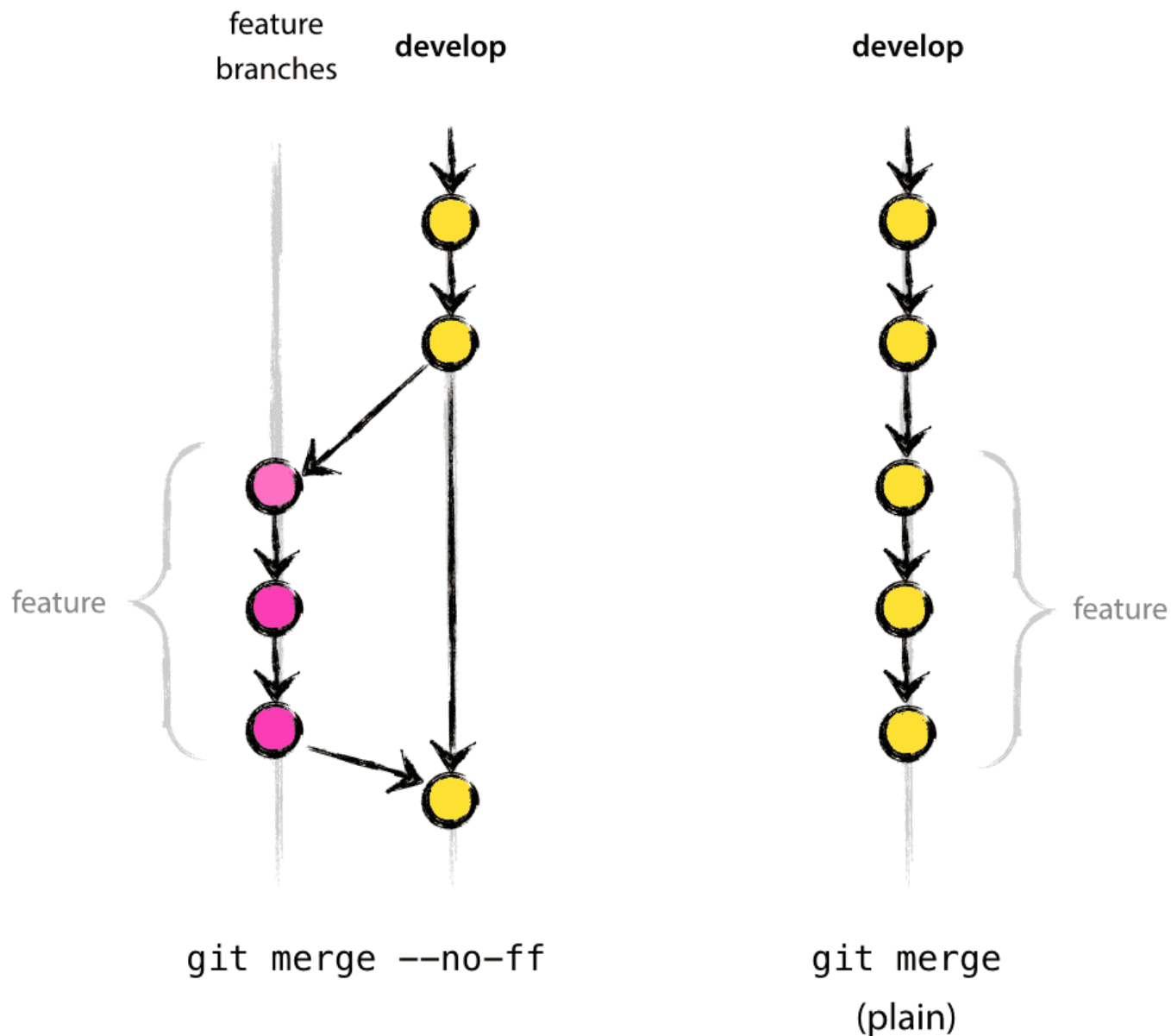
```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

将完成的功能合并到开发分支上

完成的功能特性被合并到develop分支上，表示该功能要添加到即将发布的版本中：

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

`--no-ff`表示合并总是创建新的提交对象，这样可以避免在合并分支时丢失历史信息，对比图如下：



显而易见，这就是证据啊，证据！☐

2) 发布用分支

分支来源: `develop`
合并目标: `develop`和`master`
命名惯例: `release-*`

发布用分支用于支持生产环境新版本，如修改少数的缺陷，准备版本发布的元数据（如版本号，构建日期等）。做完这些操作之后，`develop`分支便可

以为了下个大版本接受这些新功能了。

将发布用分支从 `develop` 分支上检出的关键时刻是在开发几乎完全可以反映新功能理想状态的时候。此时，至少下个版本要发布的功能所在的功能分支要合并到 `develop` 上，而功能发布在将来的版本中则可以暂时不合并，等待下一次发布用分支的检出。

在发布用分支拉出时，就需要给其分配一个版本号。而此后的 `develop` 分支上的变更都将反映这个版本。

创建一个发布用分支

发布用分支在 `develop` 分支上检出。举例来讲，目前我们的生产环境版本是 1.1.5，马上就要发布一个大版本。`develop` 分支已经准备就绪，我们决定将下一个版本的版本号为 1.2。所以我们拉出一个发布用分支，命名需要反映新的版本号：

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

创建完新分支之后，变更版本号（这里的 `bump-version.sh` 脚本用于修改文件版本号，当然，针对不同的场景，也可手动变更版本号）。

该分支会存在一段时间，这段时间内，该分支允许修改缺陷（而不是在 `develop` 上面）。在该分支上禁止添加新特性。最终，该分支必须合并到 `develop`、

完成一个发布用分支

当发布用分支已经准备就绪可以发布一个现实的版本，我们仍然有很多工作要做。首先，将发布用分支合并到 `master`（切记，所以提交到 `master` 内容一定是一新版本）。接着，提交到 `master` 上的变更必须添加标记（如使用版本号等进行标记），用于将来参考。最后，在这个发布用分支上进行的更改需要合并回 `develop`，以保证将来的版本包含缺陷的修复。

在 Git 中的前两步：

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

此时，版本已发布，并且已标记。

Tips: 你可以使用 `-s` 或者 `-u <key>` 来加密标记。

为了保留发布用分支的变更，需要合并回 `develop` 分支：

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

这一步可能也会产生冲突，所以，解决冲突并且提交。

此时，我们可以移除该发布用分支：

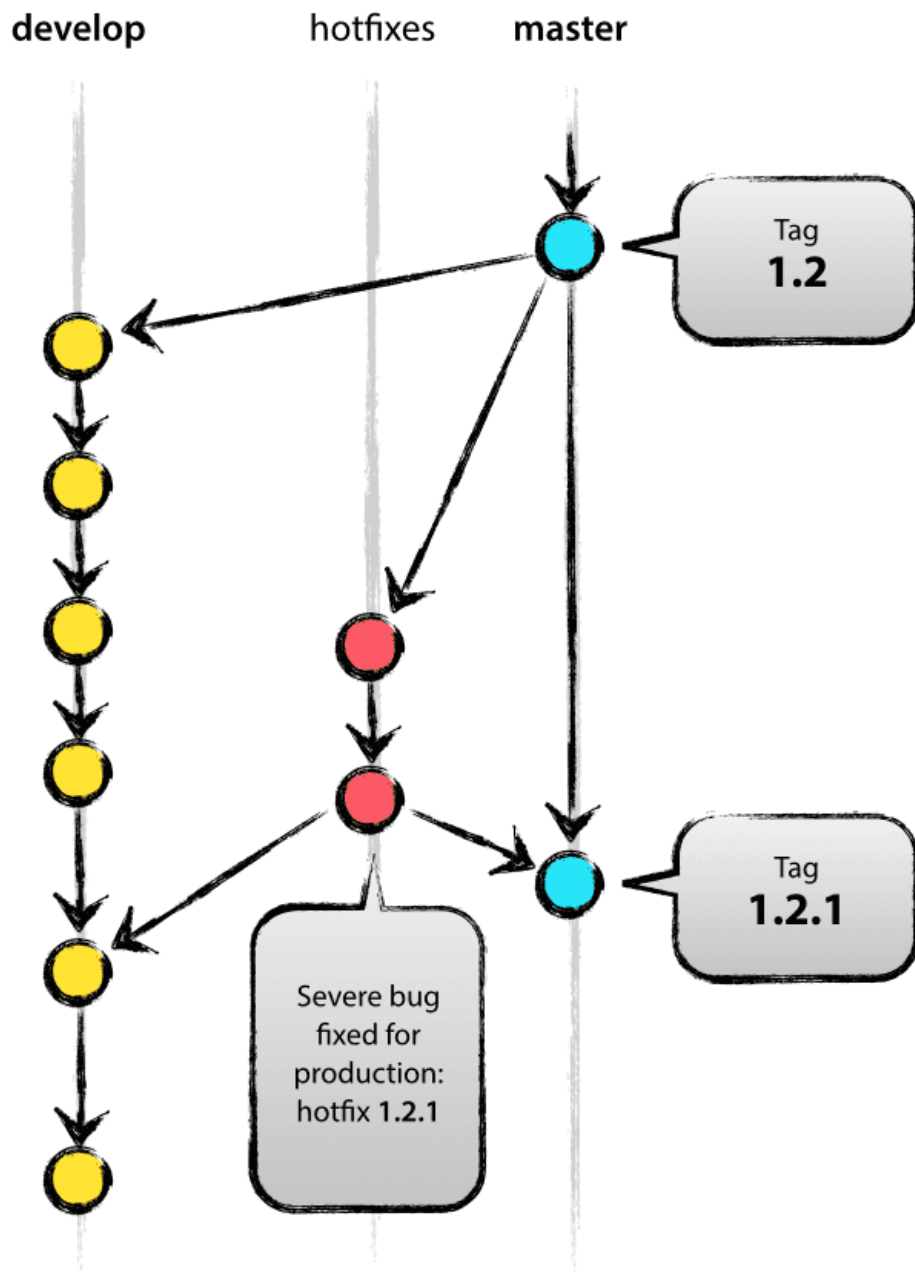
```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe)
```

3) 补丁分支

分支来源: `master`

合并目标: `develop` 和 `master`

命名惯例: `hotfix-*`



这类分支与发布用分支很类似，不过补丁分支的产生是为了快速响应生产环境中的紧急问题。当线上遭遇紧急缺陷需要立刻解决，则需要在对应标记的`master`分支上拉出一个补丁分支。

在某一位或者几位开发者修复线上问题的同时，`develop`分支可以继续进行。

创建一个补丁分支

补丁分支在`master`上拉出，举例来说，1.2版本是目前的线上版本，由于一个严重bug造成宕机的情况出现，但是目前`develop`分支上的变更还不够稳定，此时，我们可以使用补丁分支，先来解决紧急问题：

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

不要忘记增加版本号。

然后，修复bug并提交变更。

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

结束使用一个补丁分支

修复bug之后，补丁分支必须合并到`master`，同时，也需要合并到`develop`，确保在下一个版本中包含bug的修复。此时的操作与发布用分支完全一致。

首先，更新`master`并且标注版本：

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

接着，合并到`develop`：

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

这个规则存在一个例外情况：如果发布用分支当前存在，则需要将补丁分支合并到发布用分支，而不是`develop`，因为该发布用分支最终会合并到`develop`（如果`develop`分支立刻需要这个bug得到修复，而等不到发布用分支结束，则你需要小心谨慎的将修正合并到未准备就绪的`develop`分支上）。

最后，移除这个临时分支：

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

六、结语

基于Git分布式版本控制系统的GitLab为我们提供了更精细的权限管理、更灵活的工作流，使我们在开发中对权限的控制，对版本的控制，Code Review，协同工作等变的更便捷有序。第五节的Git分支模型，是[Vincent Driessen](#)写于2010年的文章，各大社区比如StackOverflow、伯乐在线都能看到对这个模型的赞赏和推荐，值得参考。