Gitlab 使用教程

版本	描述	日期	作者
V1.0.0	初版	2016-12-20	邓冰寒
V1.0.1	新增安装方法，新增目录	2017-01-03	邓冰寒
V1.0.2	增加ssh-keygen使用提示 修改分支管理说明	2017-01-03	邓冰寒
V1.0.3	新增特别说明,新用户必读黄色高亮部分	2017-03-11	邓冰寒

目录

Gitlab 使用教程 1

1．Git 客户端安装 3

1.1 安装方法 3

2．Git 环境配置 3

2.1. 设置 hosts 3

2.2. Gitlab 客户端 3

2.3. 注册新用户 3

2.4. 激活你的账户 4

2.5. 登陆 4

2.6. 设置密钥 4

2.7. 上传公钥 5

2.8 编辑 Git 配置文件 7

3. Git 使用说明 8

3.1 创建分支 8

3.2 查看创建的分支 8

3.3 查看本地文件修改状态 8

3.4 代码审查 8

3.5 代码提交 8

4. Git 分支管理策略 9

4.1 概述 9

4.2 主分支 Master 10

4.3 开发分支 Development 11

4.4 临时性分支 12

4.5 功能分支 12

4.6 预发布分支 13

4.7 修补 bug 分支 13

1 . Git 客户端安装

1.1 安装方法

Windows 用户请下载 https://github.com/git-for-windows/git/releases/download/v2.11.0.windows.1/Git-2.11.0-64-bit.exe

Ubuntu 用户直接在终端下安装 sudo apt-get install git

Mac 用户请使用 homebrew 安装，在终端下运行 brew install git

2 . Git 环境配置

2.1. 设置 hosts

Windows用户编辑：c:\windows\system32\drivers\etc\hosts,

Linux/Mac用户编辑: sudo nano /etc/hosts

在里面添加下面一行：

172.16.5.22 gitlab.vpclub

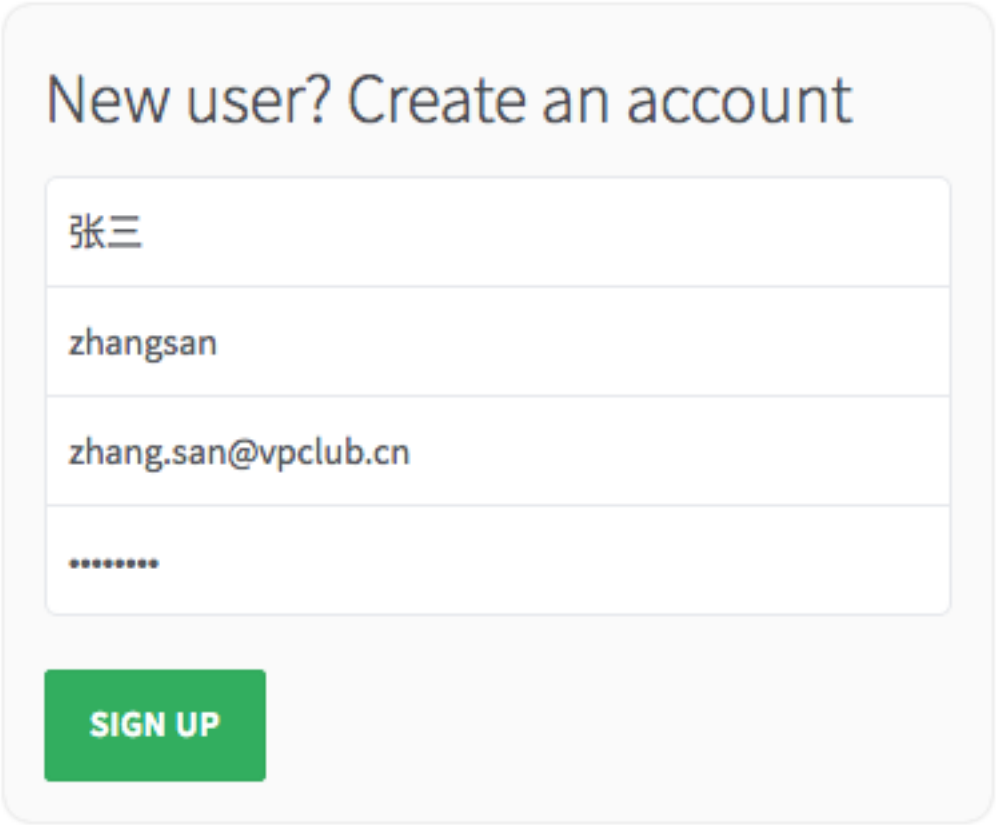
2.2. Gitlab 客户端

在浏览器输入http://gitlab.vpclub:8022

2.3. 注册新用户

特别注意下面的格式，**名字用中文**，**用户名使用全小写拼音，并使用公司邮箱**。

如果你的名字是：张三，用户名是你的全名拼音：zhangsan, 邮箱是使用公司邮箱，不要用私人邮箱：zhang.san@vpclub.cn. 密码请记住，要求 8 位，并包含英文字母大、小写及数字，如：pA5wOldy，如果你没安装要求注册，你将无法登录 Gitlab。



2.4. 激活你的账户

接下来去邮箱查看你的邮件，你很快会收到来自 Gitlab 的邮件,按照提示激活你的账户。（注：Gitlab邮件服务不是太好，收不到邮件请QQ或微信通知我直接激活，我的微信号：john-deng）

2.5. 登陆

现在你可以登陆 http://gitlab.vpclub:8022

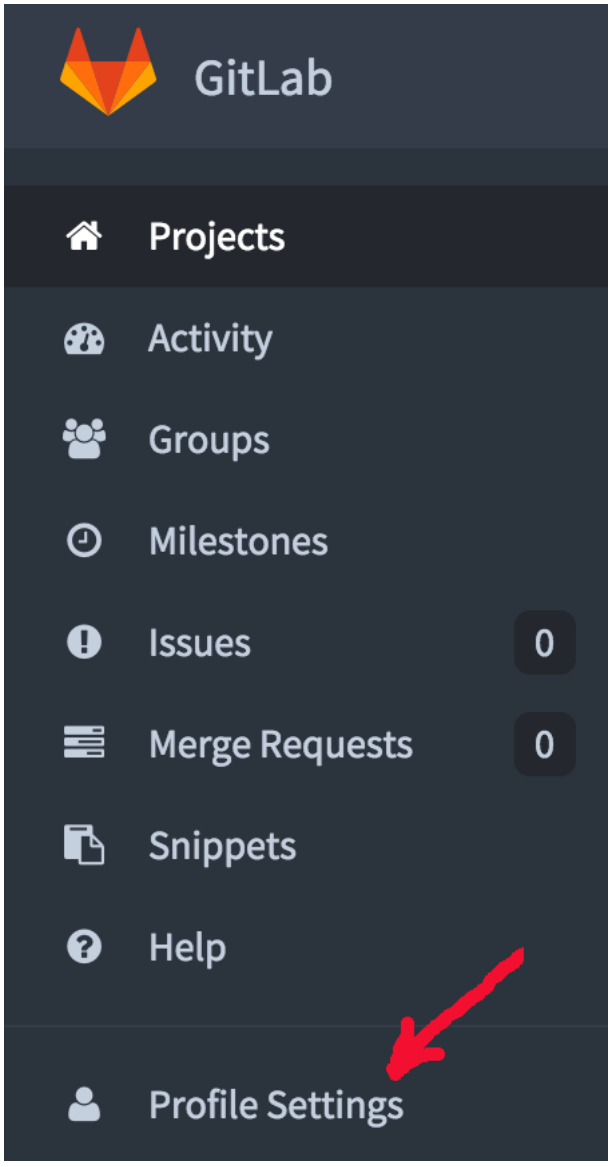
2.6. 设置密钥

回到操作系统命令提示符下（windows用户请用git bash终端）生成ssh密钥及设置git用户信息，注意如果你在运行ssh-keygen是设置了密码，那么你以后的git操作也要输入密码，否则这里可以直接回车，即运行ssh-keygen不使用密码。

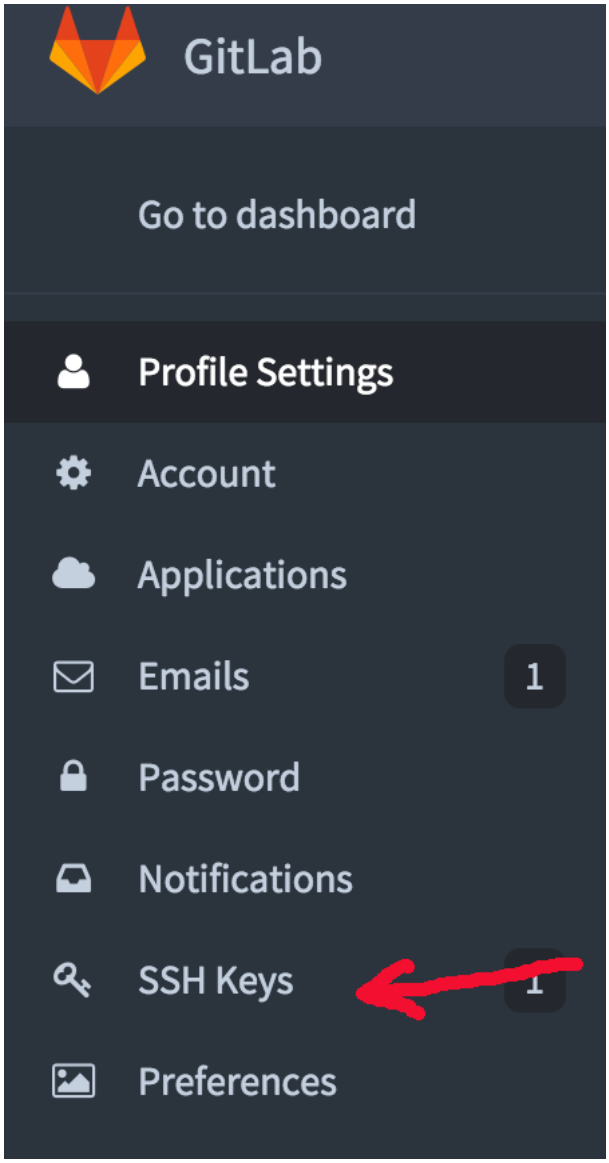
```
ssh-keygen -t rsa -b 4096 -C "zhang.san@vpclub.cn"
git config --global user.name "Zhang San"
git config --global user.email "zhang.san@vpclub.cn"
```

2.7. 上传公钥

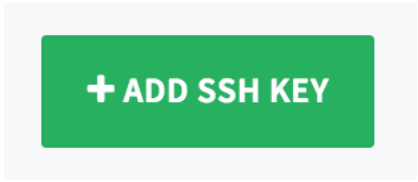
登录gitlab后，先点击 Profile Settings >



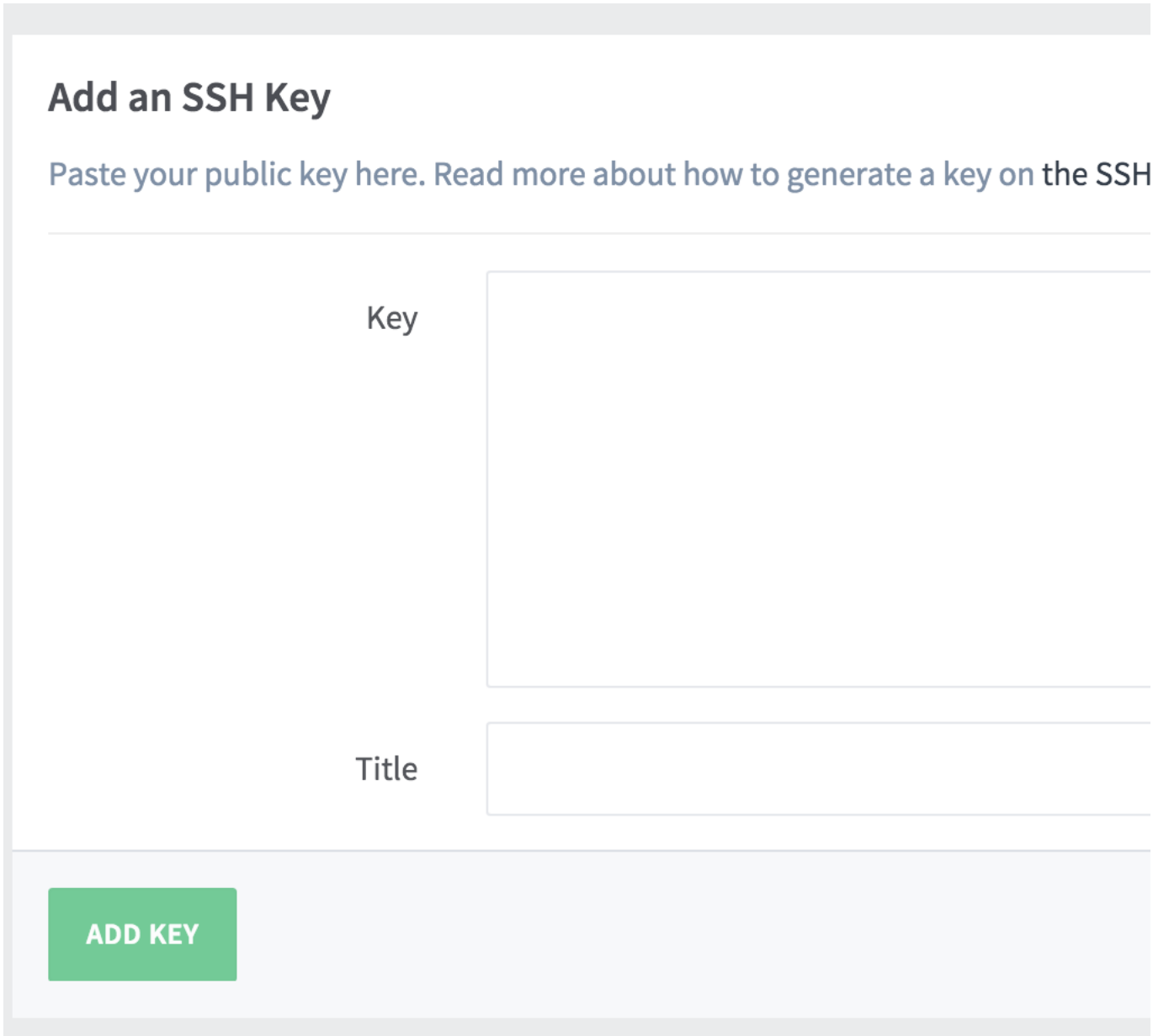
->



->



复制刚才生成好的ssh公共密钥 (Windows：c:\Users\yourname\.ssh\id_rsa.pub; Linux/Mac: ~/.ssh/id_rsa.pub)的文本内容并粘贴到下面文本框内：



2.8 编辑 Git 配置文件

在.ssh目录 （Windows：c:\Users\yourname\.ssh\config; Linux/Mac: ~/.ssh/config）下面添加config配置文件，编辑文件内容，添加以下内容：

```
Host gitlab.vpclub
    User git
    Hostname 172.16.5.22
    Port 22022
    IdentityFile ~/.ssh/id_rsa
```

到现在为止，你已经完成了gitlab的设置，接下来你可以克隆项目到本地进行开发了。

git clone git@gitlab.vpclub:demo/git-tutorial.git

关于 git 的使用方法请参考：https://git-scm.com/book/zh/v2

3. Git 使用说明

下面是实际开发过程中最常用的 git 代码分支管理策略。

3.1 创建分支

从 development 分支创建自己的 feature 分支，这里取名为 order-detail

```
git checkout -b order-detial development
```

3.2 查看创建的分支

```
git branch –a
```

3.3 查看本地文件修改状态

接下来就可以开发了，开发完成后用 git status 查看更改过的文件

```
git status
```

3.4 代码审查

找到代码审查人审查你的代码是否合格
首先运行测试用例给代码审查人确认；
解说逐个文件的更改给审查人审查。

3.5 代码提交

审查没有问题后提交代码到 feature 分支

```
git add .
git status
```

git commit
(press i to input below text)

[Description]: Your comments.
[Reviewer]: The name of reviewer

(press esc and :wq to commit)

把提交到代码推到远程服务器

git push origin order-detial

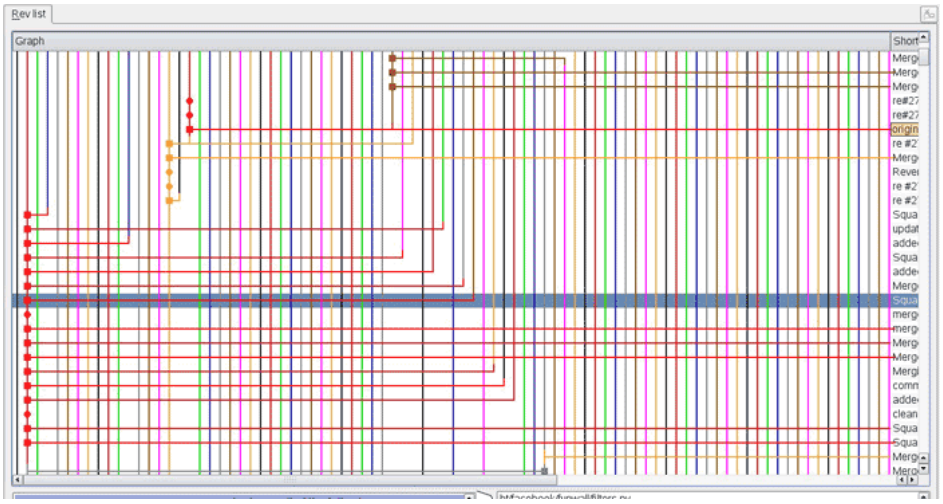
最后到 http://gitlab.vpclub:8022 确认代码是否提交到远程分支

4. Git 分支管理策略

4.1 概述

如果你严肃对待编程，就必定会使用"版本管理系统"（Version Control System）。
眼下最流行的"版本管理系统"，非 Git 莫属。

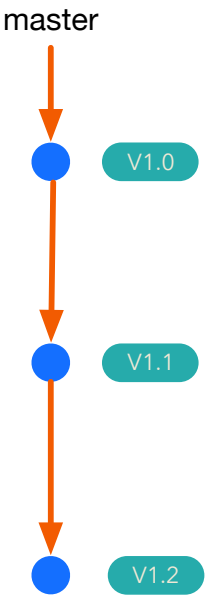
相比同类软件，Git 有很多优点。其中很显著的一点，就是版本的分支（branch）和合并（merge）十分方便。有些传统的版本管理软件，分支操作实际上会生成一份现有代码的物理拷贝，而 Git 只生成一个指向当前版本（又称"快照"）的指针，因此非常快捷易用。
但是，太方便了也会产生副作用。如果你不加注意，很可能会留下一个枝节蔓生、四处开放的版本库，到处都是分支，完全看不出主干发展的脉络。



Vincent Driessen 提出了一个分支管理的策略，我觉得非常值得借鉴。它可以使得版本库的演进保持简洁，主干清晰，各个分支各司其职、井井有条。理论上，这些策略对所有的版本管理系统都适用，Git 只是用来举例而已。如果你不熟悉 Git，跳过举例部分就可以了。

4.2 主分支 master

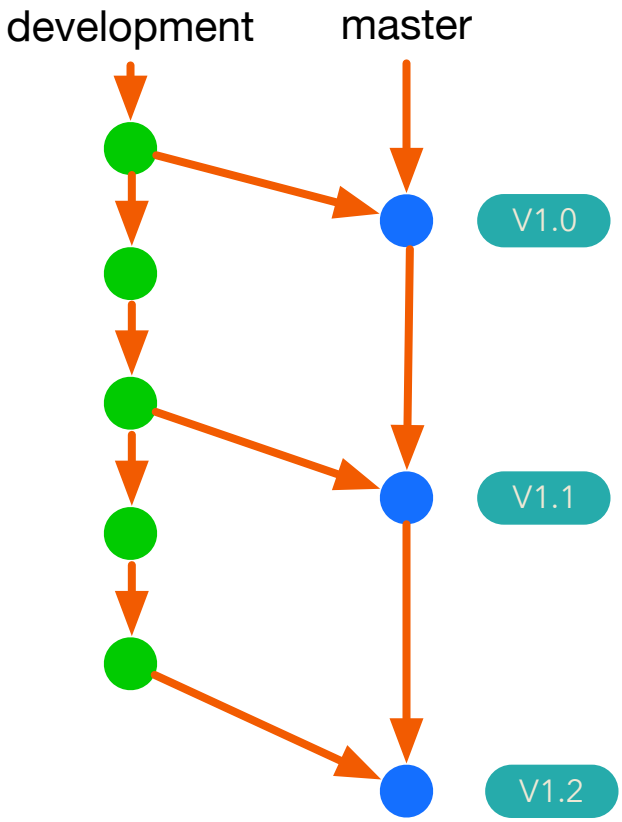
首先，代码库应该有一个、且仅有一个主分支。所有提供给用户使用的正式版本，都在这个主分支上发布。



Git 主分支的名字，默认叫做 Master。它是自动建立的，版本库初始化以后，默认就是在主分支在进行开发。

4.3 开发分支 development

主分支只用来分布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做 development，开发功能模块应该新建 feature 分支，单元测试通过后合并到 development 分支，合并完成后 feature 分支到生命周期结束，应该删除该分支。



这个分支可以用来生成代码的最新隔夜版本（nightly）。如果想正式对外发布，就在 Master 分支上，对 Development 分支进行"合并"（merge）。

Git 创建 Development 分支的命令：

```
git checkout -b development master
```

将 Development 分支发布到 Master 分支的命令：

```
# 切换到 Master 分支
git checkout master
# 对 Development 分支进行合并
git merge --no-ff development
```

这里稍微解释一下，上一条命令的--no-ff 参数是什么意思。默认情况下，Git 执行"快进式合并"（fast-farward merge），会直接将 Master 分支指向 Develop 分支。

使用--no-ff 参数后，会执行正常合并，在 Master 分支上生成一个新节点。为了保证版本演进的清晰，我们希望采用这种做法。关于合并的更多解释，请参考 Benjamin Sandofsky 的《Understanding the Git Workflow》。

4.4 临时性分支

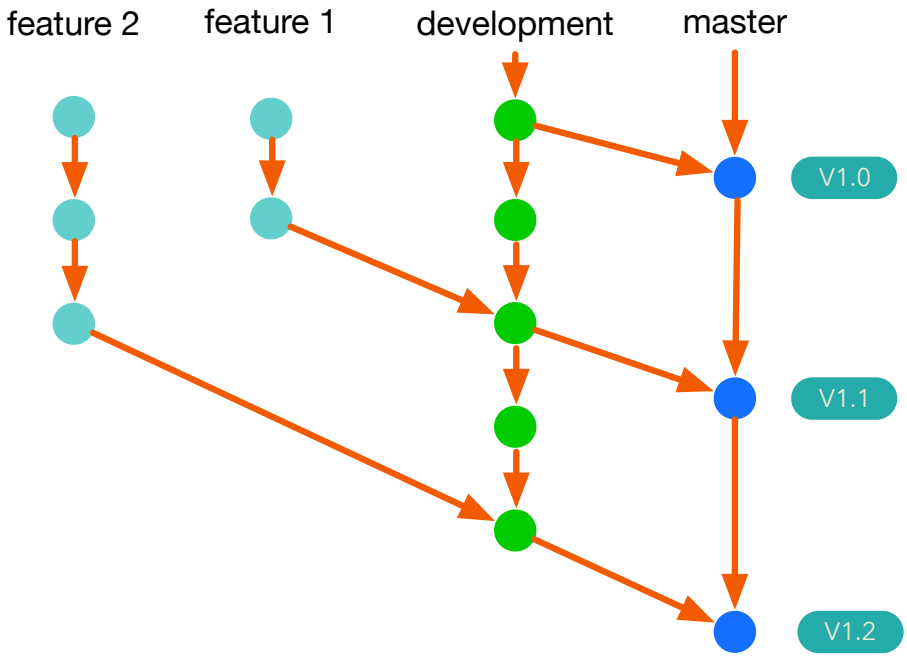
前面讲到版本库的两条主要分支：Master 和 Development。前者用于正式发布，后者用于日常开发。其实，常设分支只需要这两条就够了，不需要其他了。
但是，除了常设分支以外，还有一些临时性分支，用于应对一些特定目的的版本开发。临时性分支主要有三种：

* 功能（feature）分支
* 预发布（release）分支
* 修补 bug（fixbug）分支

这三种分支都属于临时性需要，使用完以后，应该删除，使得代码库的常设分支始终只有 Master 和 Development。

4.5 功能分支

接下来，一个个来看这三种"临时性分支"。
第一种是功能分支，它是为了开发某种特定功能，从 Development 分支上面分出来的。开发完成后，要再并入 Development。



功能分支的名字，可以采用 feature-*的形式命名。
创建一个功能分支：

```
git checkout -b feature-x development
```

开发完成后，将功能分支合并到 development 分支：

```
git checkout development
git merge --no-ff feature-x
```

删除 feature 分支：

```
git branch -d feature-x
```

4.6 预发布分支

第二种是预发布分支，它是指发布正式版本之前（即合并到 Master 分支之前），我们可能需要有一个预发布的版本进行测试。
预发布分支是从 Development 分支上面分出来的，预发布结束以后，必须合并进 Development 和 Master 分支。它的命名，可以采用 release-*的形式。
创建一个预发布分支：

```
git checkout -b release-1.2 development
```

确认没有问题后，合并到 master 分支：

```
git checkout master
git merge --no-ff release-1.2
# 对合并生成的新节点，做一个标签
git tag -a 1.2
```

再合并到 development 分支：

```
git checkout development
git merge --no-ff release-1.2
```

最后，删除预发布分支：

```
git branch -d release-1.2
```

4.7 修补 bug 分支

最后一种是修补 bug 分支。软件正式发布以后，难免会出现 bug。这时就需要创建一个分支，进行 bug 修补。
修补 bug 分支是从 Master 分支上面分出来的。修补结束以后，再合并进 Master 和 Development 分支。它的命名，可以采用 fixbug-*的形式。

创建一个修补 bug 分支：

```
git checkout -b fixbug-0.1 master
```

修补结束后，合并到 master 分支：

```
git checkout master
git merge --no-ff fixbug-0.1
git tag -a 0.1.1
```

再合并到 development 分支：

```
git checkout development
git merge --no-ff fixbug-0.1
```

最后，删除"修补 bug 分支"：

```
git branch -d fixbug-0.1
```