

# Git&GitLab 使用及规范

---

## Git 安装配置及基本使用

1. 从[官网](#)下载安装包，手动完成安装。
2. 打开 **Git Bash** 命令行工具，执行命令 `ssh-keygen -t rsa -C Email-Address` 生成一个密钥对。
3. 登录到 [GitLab](#)，点击右上角你的用户头像，点击 **Edit Profile settings**，点击 **SSH Keys**，点击 **Add SSH Key**，填写 **Title** 栏，复制用户目录下 `.ssh/id_rsa.pub` 文件的内容到 **Key**，点击 **Add Key**。
4. 点击右上角的 **New project**，填写完成后点击 **Create project** 新建一个仓库，点击 **Activity**，点击 **SSH** 后复制 **SSH** 边上栏里的地址。
5. 打开 **Git Bash** 命令行工具，切换到一个合适的目录，使用命令 `git clone '刚才复制的 URL'` 克隆创建的仓库。
6. 进入目录 `cd 仓库名`，执行命令 `gitconfig --global user.email your-email`，`gitconfig --global user.name your-name`，设置你的个人信息。
7. 执行命令：  
`echo "# Description" > README.md`，添加一个文件  
`git status`，查看当前状态，发现有未跟踪文件  
`git add .`，当前目录所有文件添加到暂存区  
`git diff`，比较当前工作区和暂存区有何不同  
`git status`，查看当前状态，发现有文件未提交  
`git commit -m "注释"`，把暂存区内容提交到本地仓库  
`git push -u origin master`，把本地仓库的提交推送到远程仓库  
`git log`，查看提交日志
8. 模拟远程更新  
登录到 [GitLab](#)，点击 **Files** 标签，点击 **README.md** 文件，点击 **Edit** 按钮修改文件内容，添加新行：`"* 远程修改文件记录"`，可以点击 **Preview** 预览，也可以填写注释后直接保存。

## 9. 执行命令:

`git pull`, 拉取远程仓库的更新提交, 并做自动合并, 可能会出现冲突

`git fetch`, 获得远程仓库的更新提交, 不做自动合并, 需要手动合并

`git merge origin/master`, 把远程更新提交手动合并到本地 **master** 分支

## 10. 配置命令别名

`gitconfig --global alias.co checkout`, 配置检出命令别名

`gitconfig --global alias.br branch`, 配置分支命令别名

`gitconfig --global alias.ci commit`, 配置提交命令别名

`gitconfig --global alias.st status`, 配置当前状态命令别名

`gitconfig --global alias.df diff`, 配置比较命令别名

`gitconfig --global alias.pl pull`, 配置拉取命令别名

`gitconfig --global alias.pu push`, 配置推送命令别名

`gitconfig --list`, 查看配置列表

`gitconfig --global credential.helper store`, 配置记住密码, 以 HTTPS 试使用时

## 11. 其它

- Git 管理大项目一段时间后会比较慢, 可以使用 `gitgc` 命令清理一下
  - Git 不管理空目录, 如果本地有空目录, 则始终不会提交到仓库中
  - Git 会根据文件的相似性自动识别重命名操作, 底层其实也是先 **delete** 后 **add**
  - 可在 `.git` 同级目录添加 `.gitignore` 文件, 文件中可写入忽略文件信息, 查看状态和提交时均会自动忽略
  - 1.7 版好像支持了像 `svn` 那样只检出部分目录的功能, 详细请移步[官网文档](#), 另附 [Git 稀疏检出教程一例](#)
  - 推荐官方书籍: [Pro Git](#)
  - [官方手册地址请戳我](#), 注: `git help 'command'` 显示的就是官方手册 **Manual Page**.
  - 还有一些其它功能, 比如: **rebase**, **rm**, **reset**, **remote**, **blame**, **cherry-pick**, **mergetool**, **difftool** 等本教程还没涉及, 以后会逐步更新
-

## Git 本地分支管理

### 1. 分支的创建、合并、删除、比较

`git branch`, 显示所有分支

`git branch b1`, 从当前分支创建一个叫 **b1** 的分支

`git checkout b1`, 切换到 **b1** 分支

`git checkout -b b1`, 相当于以上两条命令的组合

`git checkout master`, 切换到 **master** 主分支

`git merge b1`, 把 **b1** 分支的代码合并到 **master** 上

`git branch -d b1`, 删除 **b1** 分支, 不能在删除分支上执行

`git diff branch-1 branch-2`, 比较两个分支的不同

`git diff --name-only branch-1 branch-2`, 只显示两分支不同的文件的名称

---

## Git 远程分支管理

### 1. 远程分支的创建、合并、删除

`git push origin dev`, 将本地分支 **dev** 推送到 **origin** 远程仓库

`git pull origin dev`, 拉取远程仓库 **origin** 的 **dev** 分支并自动合并

`git push origin :dev`, 删除远程仓库 **origin** 上的远程分支

`git push origin --delete dev`, 1.7 版之后可以使用这个命令。

**Tips :** `tag` 的远程管理类似远程分支管理

---

## Git Tag 标签管理

### 1. 标签的创建、删除

`git tag t1`, 从当前分支创建一个名为 **t1** 的标签

`git tag -m '注释' t1`, 创建一个带有注释的标签

`git tag -d t1`, 删除名为 **t1** 的标签

---

## Git Log 日志

`git log`, 查看历史日志

`git log --graph`, 以基于文本的图形显示合并轨迹

`git log --pretty=oneline`, 一行显示日志简要信息

`git log --pretty=format:"%h - %an, %ar : %s"`, 以指定格式查看日志, `format` 参数请移步[官方手册](#)

`git log --pretty="%h - %s" --author=gitster --since="2008-10-01" --before="2008-11-01" --no-merges`, 以指定格式显示指定日期区间和指定提交者的日志, 不含合并提交

`git log -p -2 --stat`, `-p` 显示文件差异, `-2` 显示最近两次提交, `--stat` 显示修改行数统计信息

---

## GitLab 权限管理

GitLab 有五种身份权限, 分别是:

- **Owner** 项目所有者, 拥有所有的操作权限
- **Master** 项目的管理者, 除更改、删除项目元信息外其它操作均可
- **Developer** 项目的开发人员, 做一些开发工作, 对受保护内容无权限
- **Reporter** 项目的报告者, 只有项目的读权限, 可以创建代码片断
- **Guest** 项目的游客, 只能提交问题和评论内容

具体参见 [GitLab 权限](#), 为项目添加成员时可指定成员的身份权限。

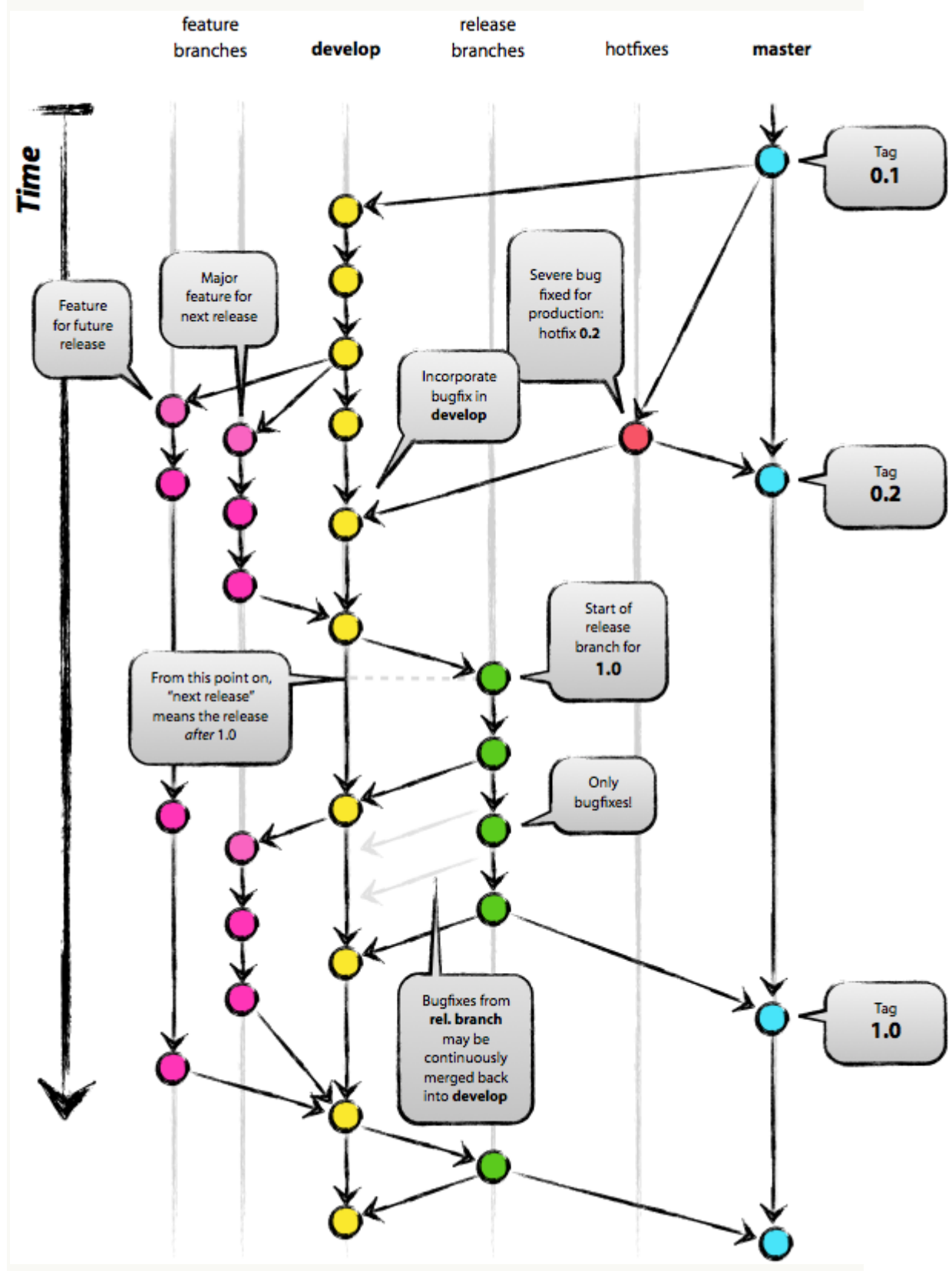
---

## 命名规则

- 每次提交必须写明注释，如果是修复 Bug，请加上 Bug 号
  - 创建特性分支，名称要以 **f-** 开头，加上特性名
  - 创建发布分支，名称要以 **r-** 开头，加上预发布版本号
  - 创建 Bug 修复分支，名称要以 **b-** 开头，加上 Bug 号
  - 创建标签，名称要以 **t-** 开头，加上发布版本号
  - 合并分支时必须使用 **--no-ff** 参数（禁止以快进方式合并），以保留合并历史轨迹
- 

## 分支模型

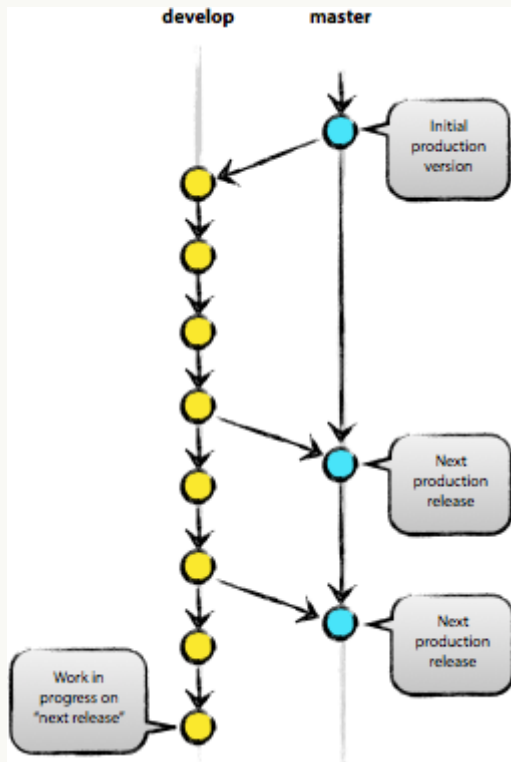
整体流程图:



## 主要分支（保护分支）

- **master** 主分支，稳定代码，为生产环境做准备的
- **develop** 开发分支，为开发服务

分支关系类似下图：



## 辅助分支

### 特性分支

从 **develop** 分支创建，用于特性开发，完成后要合并回 **develop** 分支。

操作过程：

`git checkout -b newfeature develop`, 从 **develop** 分支创建 **newfeature** 特性分支

`git checkout develop`, 开发完成后，需要合并回 **develop** 分支，先切换到 **develop**

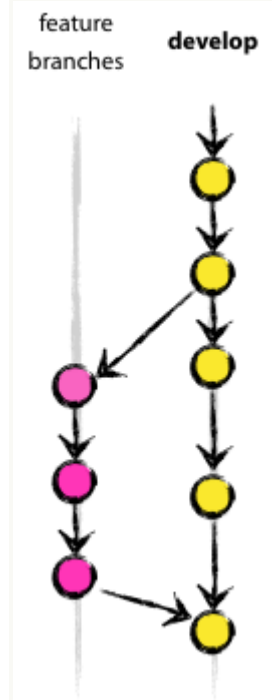
## 分支

`git merge --no-ff newfeature`, 合并回 `develop` 分支, 必须加 `--no-ff` 参数

`git branch -d newfeature`, 删除特性分支

`git push origin develop`, 把合并后的 `develop` 分支推送到远程仓库

分支关系类似下图:



---

## 发布分支

从 `develop` 分支创建, 用于预发布版本, 允许小 `bug` 修复, 完成后要合并回 `develop` 和 `master`。

操作过程:

`git checkout -b release-1.2 develop`, 创建一个发布分支

`git checkout master`, 切换到 `master` 分支, 准备合并

`git merge --no-ff release-1.2`, 把 `release-1.2` 分支合并到 `master` 分支

`git tag 1.2`, 从 `master` 分支打一个标签

`git checkout develop`, 切换到 `develop` 分支, 准备合并

`git merge --no-ff release-1.2`, 把 `release-1.2` 分支合并到 `develop` 分支

`git branch -d release-1.2`, 删除这个发布分支



## 修复分支

从 **master** 分支创建，用于生产环境上的 Bug 修复，完成后要合并回 **develop** 和 **master**。

操作过程：

`git checkout -b hotfix-1.2.1 master`，从 **master** 分支创建一个 Bug 修复分支

`git checkout master`，切换到 **master** 分支，准备合并

`git merge --no-ff hotfix-1.2.1`，合并到 **master** 分支

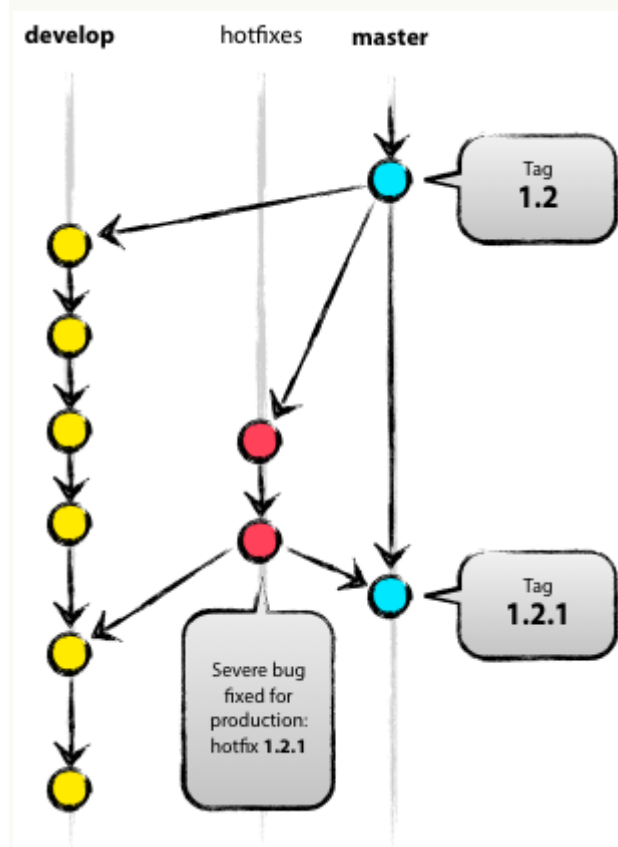
`git tag 1.2.1`，为 **master** 分支创建一个标签

`git checkout develop`，切换到 **develop** 分支，准备合并

`git merge --no-ff hotfix-1.2.1`，合并到 **develop** 分支

`git branch -d hotfix-1.2.1`，删除 **hotfix-1.2.1** 分支

分支关系类似下图：

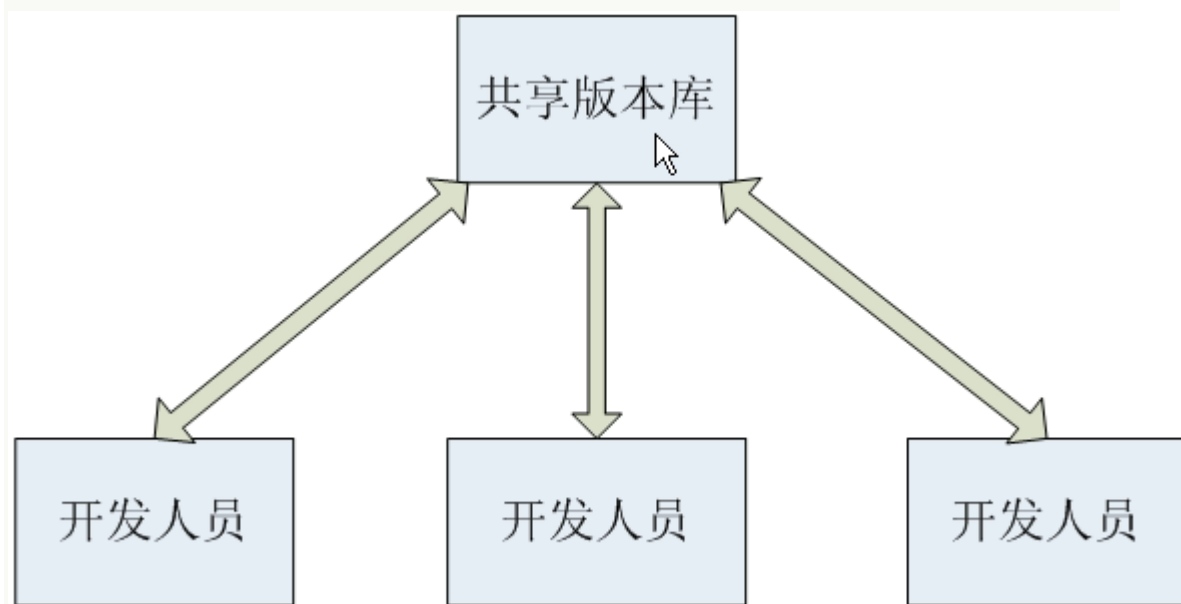


## Git 协同模型

---

### SVN 式集中协同模型

适用于小型项目，参与人员较少的项目，每个开发者均可向仓库推送代码



### 金字塔模型

适用于大型项目，参与人员较多，并且等级划分严明，代码需要逐级审核的项目  
仅核心开发人员可以向仓库推送代码，开发人员只能从仓库拉取代码，开发人员的代码需先推送给核心开发人员审核通过后，合并之后才能推送，一般情况下是

使用 GitHub 的 Pull Request 的方式

