

# git 项目管理二

---

## 修订记录

---

版本号	修订日期	修订人	备注
1.0	2015/6/9	罗展昭	初稿。
1.1	2015/7/12	罗展昭	加入 <code>git-flow</code> 简介 描述内容。
1.2	2015/7/16	罗展昭	加入 <code>SmartGit</code> 安装配置 描述内容。
1.2	2015/7/16	罗展昭	加入 <code>功能提交日志规范</code> 描述内容。
1.3	2016/9/11	罗展昭	加入 <code>Git-Flow</code> 使用 章节内容。

## 1. 概述

---

`git-flow` 是在 Git 命令集基础上，工作流程围绕项目发布定义了严格的分支模型。`git-flow` 没有在 Git 命令集基础上增加任何新的概念或命令。`git-flow` 在基于 Git 命令集基础上，明确了各个分支角色，并且定义了使用场景和用法。

本公司 git 项目管理，遵循 `git-flow` 模型标准。本文主要简单介绍 `git-flow` 模型原理，以及在项目过程中，如何通过 `SmartGit` 进行 `git-flow` 模型下的代码管理。

## 2. 读者范围

---

- git 项目代码管理人员

## 3. 分布式代码管理模型 git-flow 简介

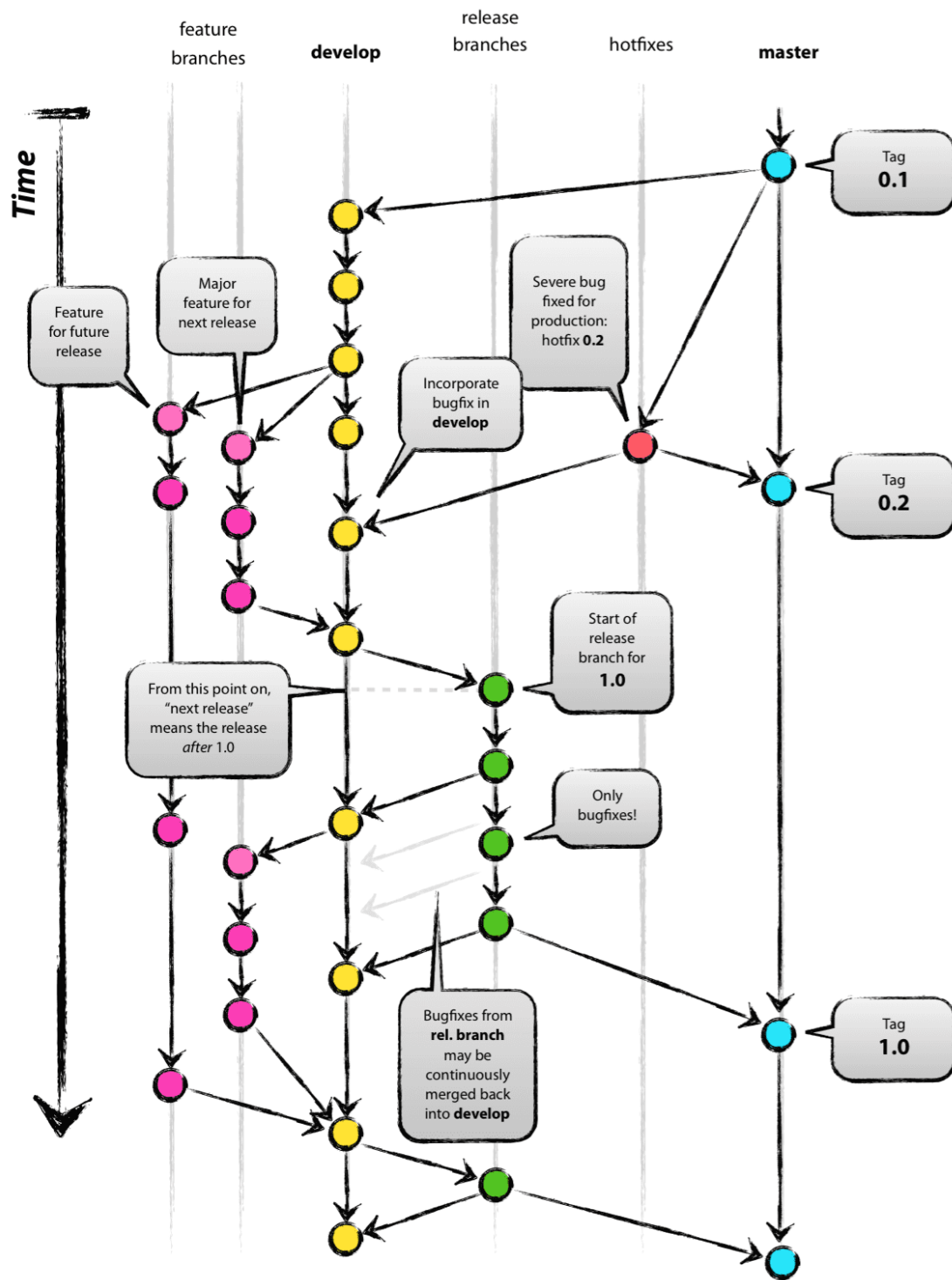
---

### 3.1. git-flow 是什么

---

2010年5月，在一篇名为 [一种成功的 git 分支模型](#) 的博文中，作者 nvie 介绍了一种在 git 之上的软件开发模型。通过利用 git 创建和管理分支的能力，为每个分支设定具有特定的含义名称，并将软件生命周期中的各类活动归并到不同的分支上。实现了软件开发过

程不同操作的相互隔离。这种软件开发的模型被作者称为 git-flow。



## 3.2. git-flow 中的分支定义

git-flow 模型中定义了主分支和辅助分支两类分支。其中主分支用于组织与软件开发、部署相关的活动；辅助分支组织为了解决特定的问题而进行的各种开发活动。

## 3.2.1. 主分支

主分支是所有开发活动的核心分支。所有的开发活动产生的输出物最终都会反映到主分支的代码中。主分支分为 `master` 分支和 `develop` 分支。

- `master` 分支  
`master` 分支上存放的应该是随时可供在生产环境中部署的代码（Production Ready state）。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，最好添加对应的版本号标签（TAG）。
- `develop` 分支  
`develop` 分支是保存当前最新开发成果的分支。通常这个分支上的代码也是可进行每夜间发布的代码（Nightly build）。因此这个分支有时也可以被称作整合分支（Integration branch）。

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，通过了所有的测试后，并且代码已经足够稳定时，就可以将所有的开发成果合并回 `master` 分支了。对于 `master` 分支上的新提交的代码建议都打上一个新的版本号标签（TAG），供后续代码跟踪使用。

因此，每次将`develop`分支上的代码合并回`master`分支时，我们都可以认为一个新的可供在生产环境中部署的版本就产生了。通常而言，“仅在发布新的可供部署的代码时才更新`master`分支上的代码”是推荐所有人都遵守的行为准则。基于此，理论上说，每当有代码提交到`master`分支时，我们可以使用Git Hook触发软件自动测试以及生产环境代码的自动更新工作。这些自动化操作将有利于减少新代码发布之后的一些事务性工作。

## 3.2.2. 辅助分支

辅助分支是用于组织解决特定问题的各种软件开发活动的分支。辅助分支主要用于组织软件新功能的并行开发、简化新功能开发代码的跟踪、辅助完成版本发布工作以及对生产代码的缺陷进行紧急修复工作。这些分支与主分支不同，通常只会在有限的时间范围内存在。

辅助分支包括：

- 用于开发新功能时所使用的 `feature` 分支；
- 用于辅助版本发布的 `release` 分支；
- 用于修正生产代码中的缺陷的 `hotfix` 分支。

以上这些分支都有固定的使用目的和分支操作限制。从单纯技术的角度说，这些分支与 `git` 其他分支并没有什么区别，但通过命名，我们定义了使用这些分支的方法。

### feature 分支

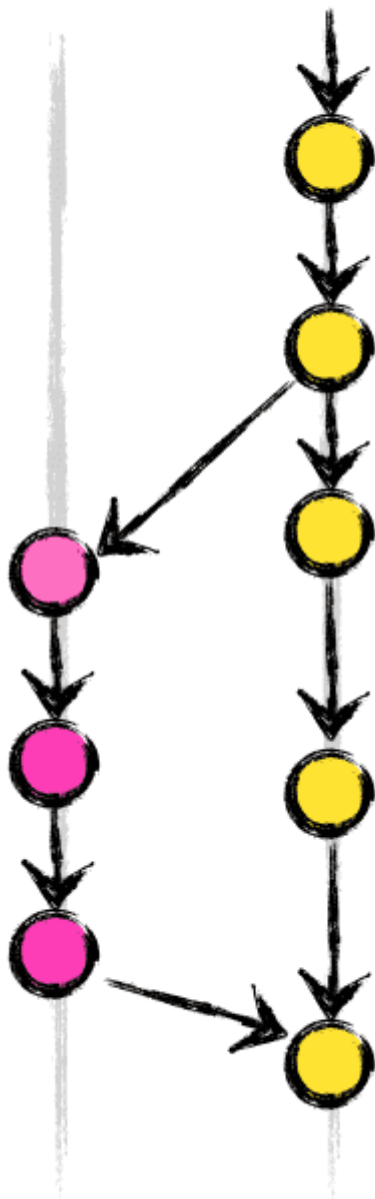
**feature** 分支（有时也可以被叫做“topic分支”）通常是在开发一项新的软件功能的时候使用，这个分支上的代码变更最终合并回 **develop** 分支或者干脆被抛弃掉（例如实验性且效果不好的代码变更）。

**feature** 分支代码除特殊情况外，只保存在开发者自己的代码库中而不提交到主代码库里。

遵循规则：

- **feature** 分支从 **develop** 分支派生；
- 完成时必须合并回 **develop** 分支；
- 分支的命名可以使用除 **master**，**develop**，**release-\***，**hotfix-\*** 之外的任何名称，命名惯例 **feature-\***。

**feature**  
**branches**      **develop**



## release 分支

release分支是为发布新的产品版本而设计的。在这个分支上的代码允许做小的缺陷修正、准备发布版本所需的各项说明信息（版本号、发布时间、编译时间等等）。

通过在release分支上进行这些工作可以让develop分支空闲出来以接受新的feature分支上的代码提交，进入新的软件开发迭代周期。当develop分支上的代码已经包含了所有即将发布的版本中所计划包含的软件功能，并且已通过所有测试时，我们就可以考虑准备创建release分支了。而所有在当前即将发布的版本之外的业务需求一定要确保不能混到release分支之内（避免由此引入一些不可控的系统缺陷）。

成功的派生了release分支，并被赋予版本号之后，develop分支就可以为“下一个版本”服务了。所谓的“下一个版本”是在当前即将发布的版本之后发布的版本。版本号的命名可以依据项目定义的版本号命名规则进行。

遵循规则：

- 从 `develop` 分支派生；
- 完成时必须分别合并到 `develop` 分支和 `master` 分支；
- 分支命名惯例：`release-*`

## hotfix 分支

除了是计划外创建的以外，`hotfix` 分支与 `release` 分支十分相似：都可以产生一个新的可供在生产环境部署的软件版本。

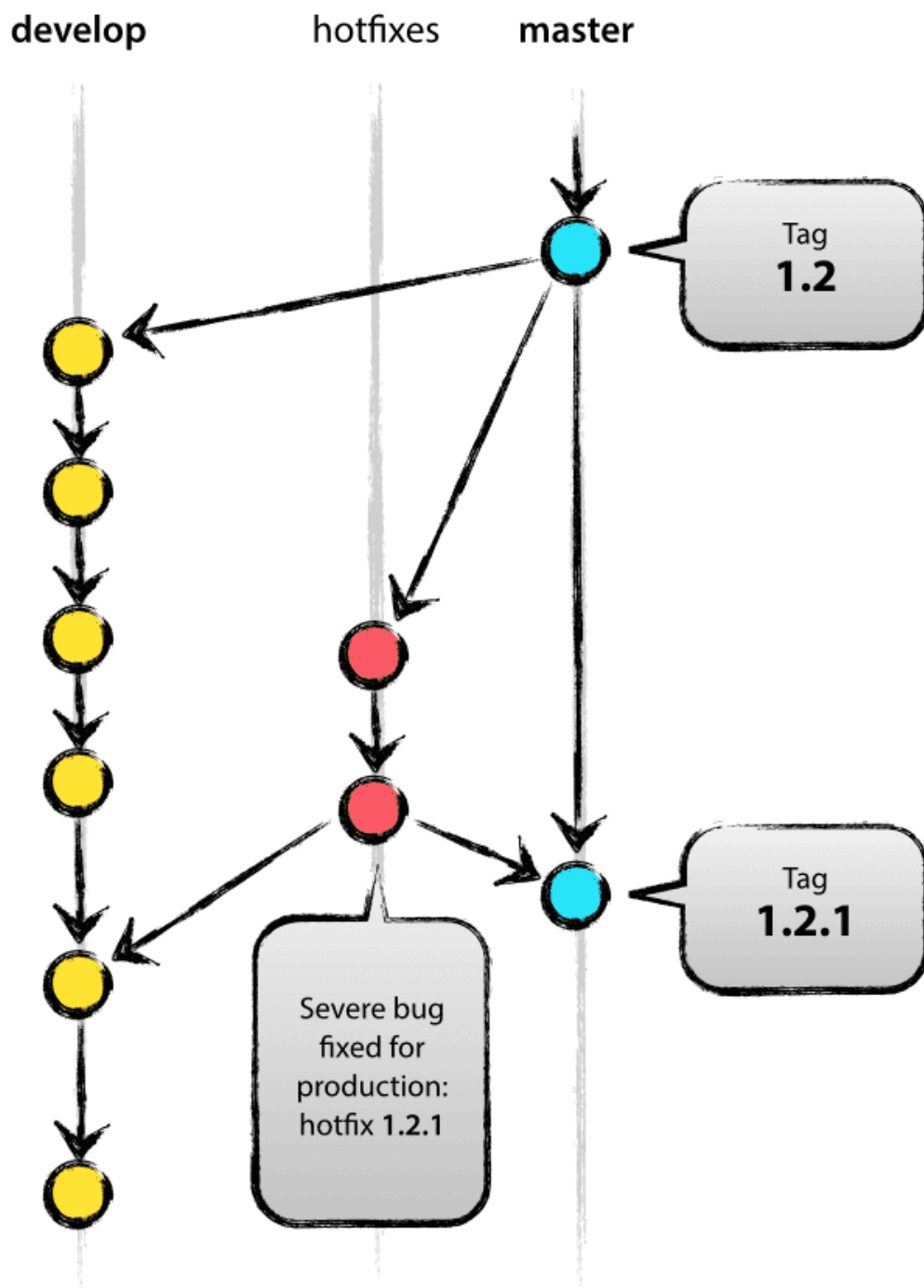
当生产环境中的软件遇到了异常情况或者发现了严重到必须立即修复的软件缺陷的时候，就需要从 `master` 分支上指定的 TAG 版本派生 `hotfix` 分支来组织代码的紧急修复工作。

这样做的显而易见的好处是不会打断正在进行的 `develop` 分支的开发工作，能够让团队中负责新功能开发的人与负责代码紧急修复的人并行的开展工作。

遵循规则：

- 可以从 `master` 分支派生；
- 必须合并回 `master` 分支和 `develop` 分支；

- 分支命名惯例: `hotfix-*`。



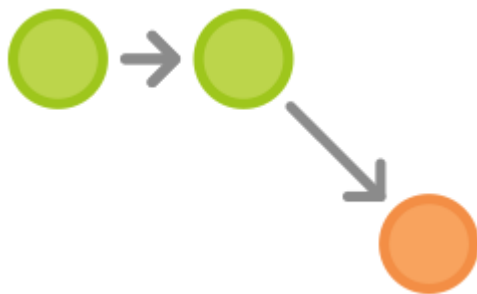
## 4. Git-Flow 使用

---



## 4.1. Git-Flow 开发实例

### 创建develop分支



第一步是给默认的master配备一个develop分支。

一种简单的做法是：让一个开发者在本地建立一个空的develop分支，然后把它推送到服务器。

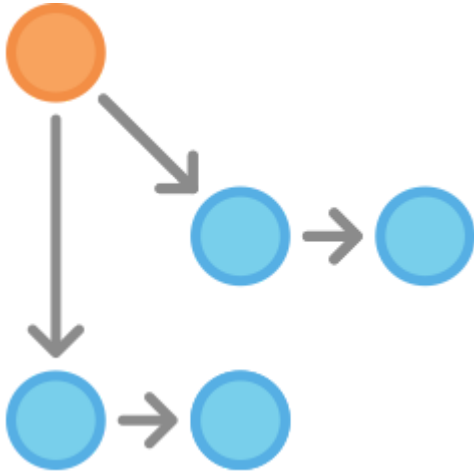
```
git branch develop
git push -u origin develop
```

develop分支将包含项目的所有历史，而master会是一个缩减版本。现在，其他开发者应该克隆（clone）中央仓库，并且为develop创建一个追踪分支。

```
git clone ssh://user@host/path/to/repo.git
git checkout -b develop
origin/develop
```



###A和B开发新功能



分别开发新功能开始。他们俩各自建立了自己的分支。

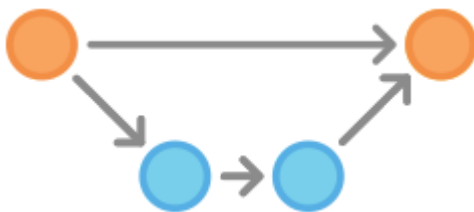
注意，他们在创建分支时，父分支不能选择master，而要选择develop。

```
git checkout -b some-feature develop
```

他们俩都在自己的功能开发分支上开展工作。通常就是这种Git三部曲：edit, stage, commit:

```
git status
git add <some-file>
git commit
```

## A把他的功能开发好了

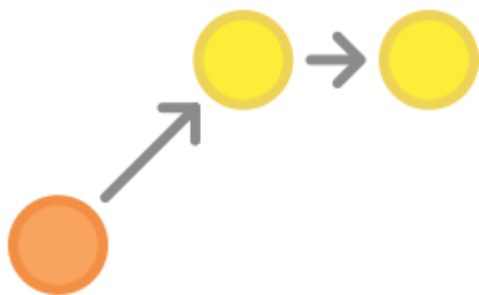


在提交过几次代码之后，A觉得他的功能做完了。如果她所在的团队使用“拉拽请求”，此刻便是一个合适的时机——她可以提出一个将她所完成的功能合并入develop分支的请求。要不然，她可以自行将她的代码合并入本地的develop分支，然后再推送到中央仓库，像这样：

```
git pull origin develop
git checkout develop
git merge <some-feature>
git push
git branch -d <some-feature>
```

第一条命令确保了本地的develop分支拥有最新的代码——这一步必须在将功能代码合并之前做！注意，新开发的功能代码永远不能直接合并入master。必要时，还需要解决在代码合并过程中的冲突。

## A开始准备一次发布

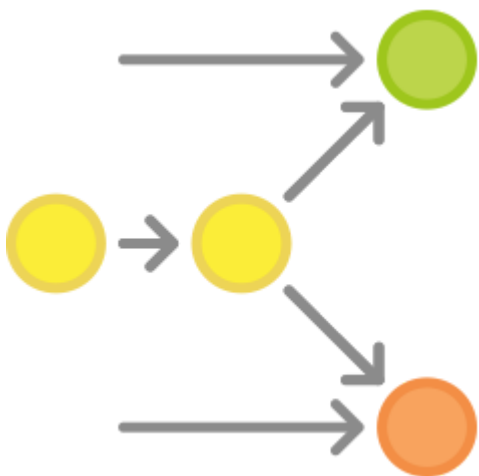


尽管B还在忙着开发他的功能，A却可以开始准备这个项目的第一次正式发布了。类似于功能开发，她使用了一个新的分支来做产品发布的准备工作。在这一步，发布的版本号也最初确定下来。

```
git checkout -b release-0.1 develop
```

这个分支专门用于发布前的准备，包括一些清理工作、全面的测试、文档的更新以及任何其他准备工作。它与用于功能开发的分支相似，不同之处在于它是专为产品发布服务的。一旦A创建了这个分支并把它推向中央仓库，这次产品发布包含的功能也就固定下来了。任何还处于开发状态的功能只能等待下一个发布周期。

## A完成了发布



一切准备就绪之后，A就要把发布分支合并入master和develop分支，然后再将发布分支删除。注意，往develop分支的合并是很重要的，因为开发人员可能在发布分支上修复了一些

关键的问题，而这些修复对于正在开发中的新功能是有利的。再次提醒一下，如果A所在的团队强调代码评审（Code Review），此时非常适合提出这样的请求。

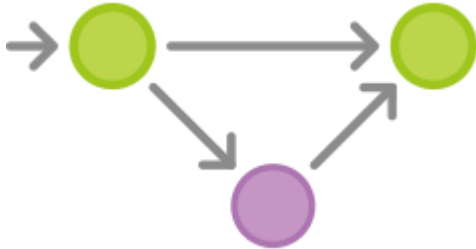
```
git checkout master
git merge release-0.1
git push
git checkout develop
git merge release-0.1
git push
git branch -d release-0.1
```

发布分支扮演的角色是功能开发（develop）与官方发布（master）之间的一个缓冲。无论什么时候你把一些东西合并入master，你都应该随即打上合适的标签。

```
git tag -a 0.1 -m "Initial public release" master
git push --tags
```

Git支持钩子（hook）的功能，也就是说，在代码仓库里某些特定的事件发生的时候，可以执行一些预定义的脚本。因此，一种可行的做法是：在服务器端配置一个钩子，当你把master推送到中央仓库或者推送标签时，Git服务器能为产品发布进行一次自动的构建。

## 用户发现了一个bug



当一次发布完成之后，A便回去与B一起开发其他功能了。突然，某个用户提出抱怨说当前发布的产品里有一个bug。为了解决这个问题，A（或者B）基于master创建了一个用于维护的分支。她在这个分支上修复了那个bug，然后把改动的代码直接合并入master。

```
git checkout -b issue-#001 master
# Fix the bug
git checkout master
git merge issue-#001
git push
```

跟用于发布的分支一样，在维护分支上的改动也需要合并入develop分支，这一点是很重要的！因此，小马务必不能忘了这一步。随后，她就可以将维护分支删除。

```
git checkout develop
git merge issue-#001
git push
git branch -d issue-#001
```

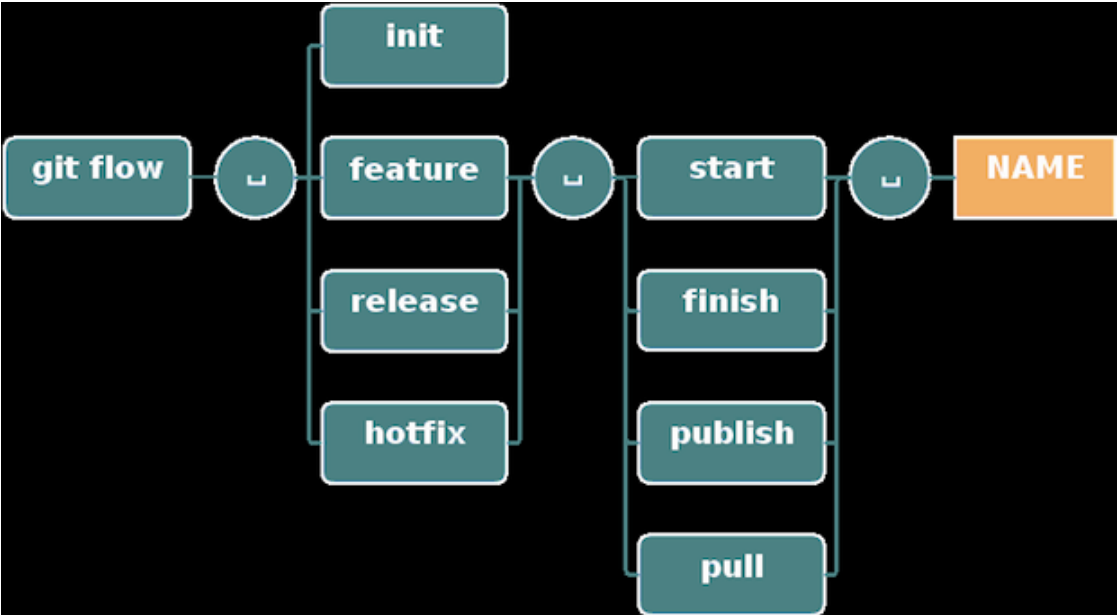
# 4.2. 使用 git-flow script 简化命令

上面介绍的是 git-flow 的详细过程。实际上，如果对 git-flow 流程足够熟悉可以直接使用命令执行。  
但是命令普遍羞涩，入门困难，可以使用 git-flow script 对其进行了封装简化。

## 安装

```
wget -q -O - --no-check-certificate
https://github.com/nvie/gitflow/raw/develop/contrib/gitflow-installer.sh |
bash
```

## 使用



功能	命令
初始化	git flow init
开始新 Feature	git flow feature start MYFEATURE
发布一个Feature（也就是push到远程）	git flow feature publish MYFEATURE

功能	命令
获取发布的Feature	<code>git flow feature pull origin MYFEATURE</code>
完成一个Feature	<code>git flow feature finish MYFEATURE</code>
开始一个Release	<code>git flow release start RELEASE [BASE]</code>
Publish一个Release	<code>git flow release publish RELEASE</code>
发布 Release	<code>git flow release finish RELEASE</code>
提交 Release 标签	<code>git push --tags</code>
开始一个 Hotfix	<code>git flow hotfix start VERSION [BASENAME]</code>
发布一个Hotfix	<code>git flow hotfix finish VERSION</code>

```
git flow init
```

这个命令会进行一些默认的配置，可以自动创建上面介绍的所有分支：master、develop、feature、release、hotfix等分支。

完成后当前所在分支就变成 develop。任何开发都必须从 develop 开始。

当进行新功能开发的时候：

```
git flow feature start some_awesome_feature
```

完成功能开发之后：

```
git flow feature finish some_awesome_feature
```

该命令将会把 feature/some\_awesome\_feature 合并到 develop 分支，然后删除功能 (feature)分支。

将一个 feature 分支推到远程服务器

```
git flow feature publish some_awesome_feature
```

或者

```
git push origin feature/some_awesome_feature
```

当你的功能点都完成时（需要发布新版本了），就基于develop创建一个发布(release)分支。

```
git flow release start v0.1.0
```

当你在完成 (finish) 一个发布分支时，它会把你所作的修改合并到 master 分支，同时合并回 develop 分支，所以，你不需要担心你的 master 分支比 develop 分支更加超前。当系统出现问题的时候，需要进行紧急修改的时候，就好基于 master 创建一个维护 (hotfix) 分支。

```
git flow hotfix start v0.1.0
```

当你在完成 (finish) 一个维护分支时，它会把你所作的修改合并到 master 分支，同时合并回 develop 分支。

## 5. 使用 SmartGit 实现 git-flow 模型管理

---

git-flow 模型是一个很健全的分布式管理模型，但实现起来其指令逻辑有着一定的复杂性。

由于 git-flow 的广泛推广，目前很多线程的 git 客户端都具备 git-flow 指令功能，这里推荐使用 [SmartGit](#) 作为 git-flow 模型实践工具。

### 5.1. SmartGit 下载

---

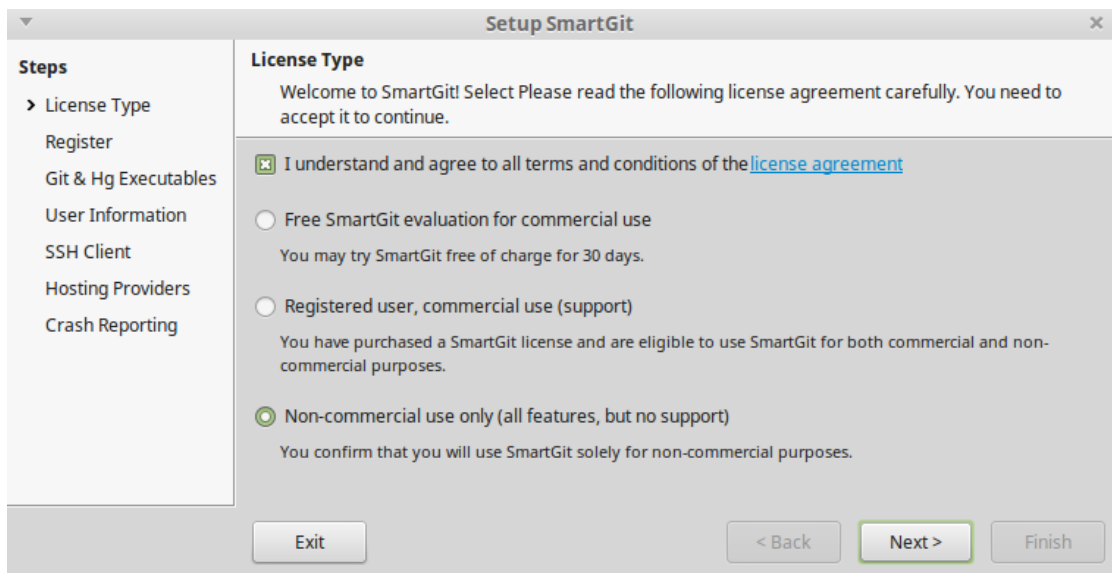
SmartGit 安装客户端可以通过局域网 FTP 服务器下载（用户名/密码：down）。

- [Windows 版本](#)
- [macOS 版本](#)
- [Ubuntu 版本](#)

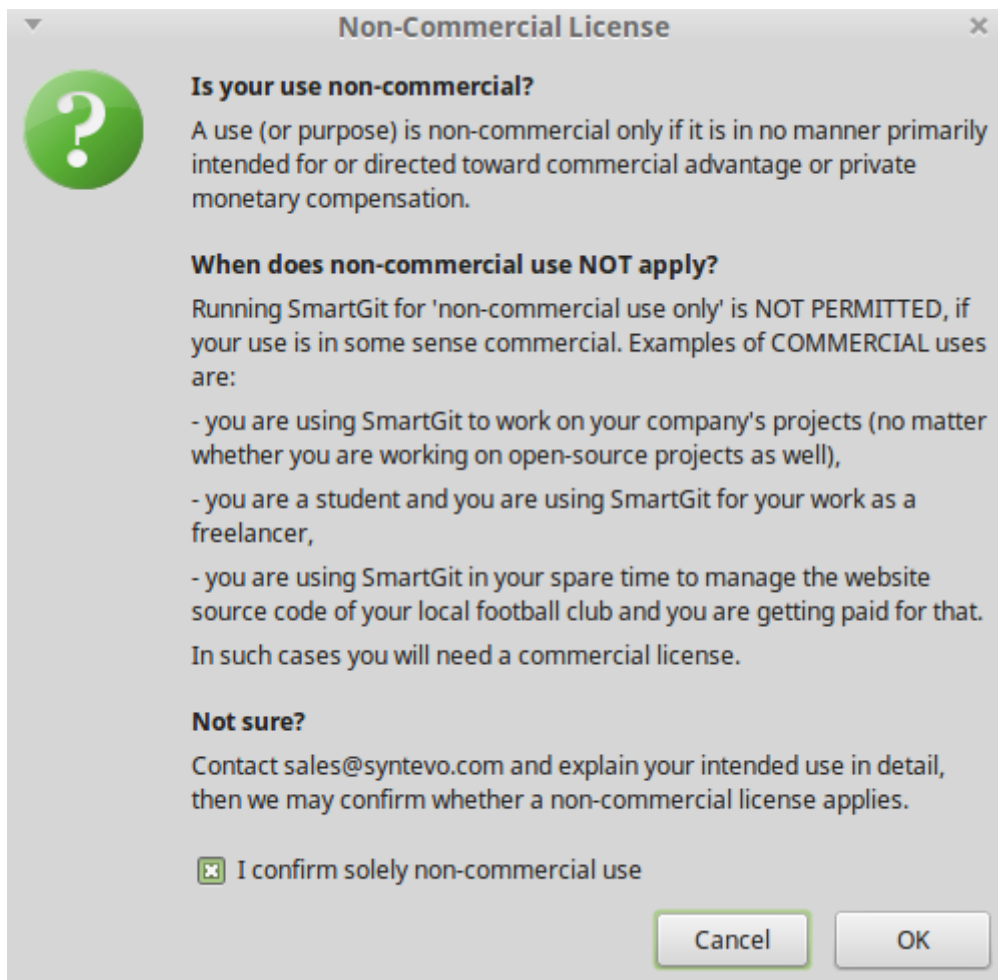
### 5.2. SmartGit 安装配置

---

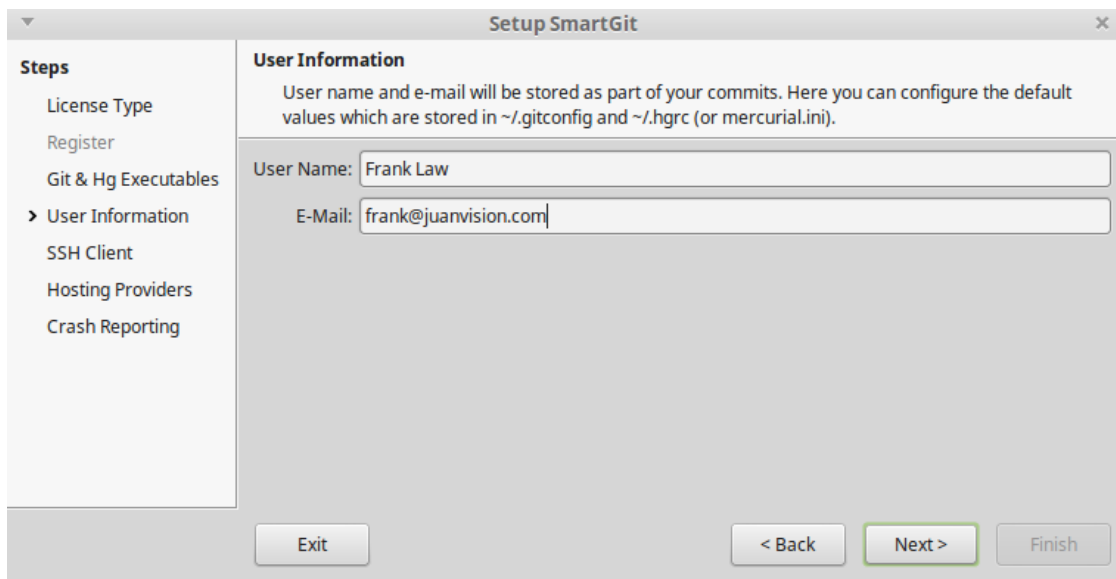
配置时选择 License Type 最后一项非商用模式 Non-commercial user only(all features, but no support)。



然后当然需要同意它各种的条款。

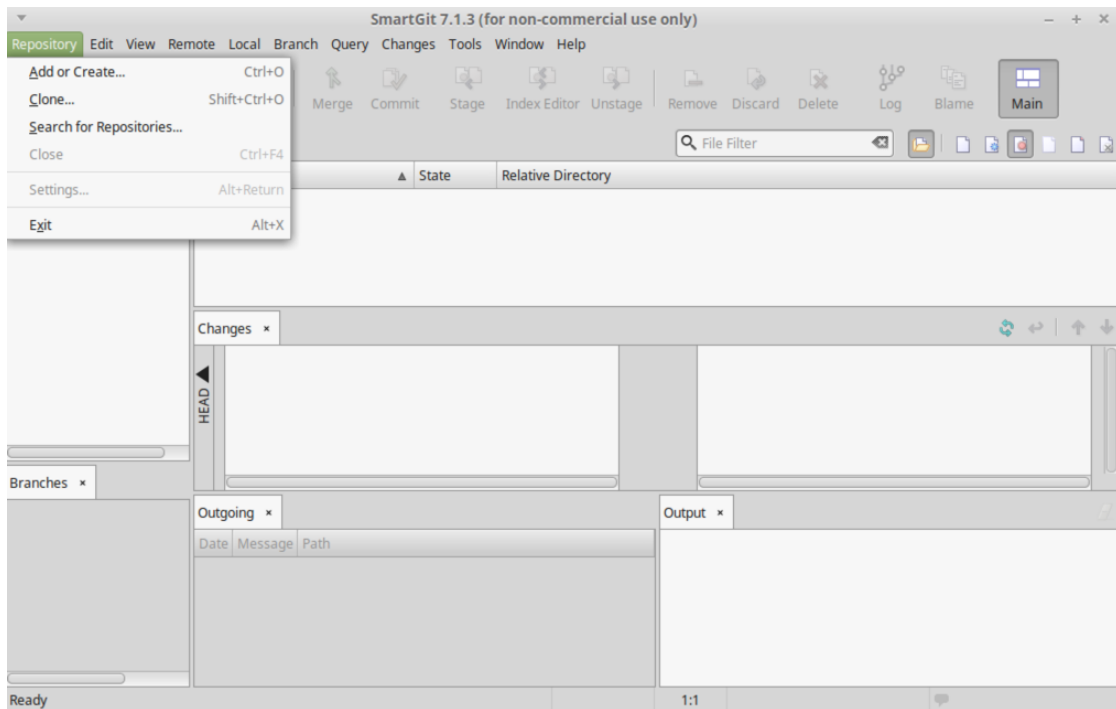


输入 git 中使用的的用户名邮箱，此名称将会显示在 git 的日志作者标签中。



随后一直 Next 至配置完成即可。

初始化完成以后，通过 Repository -> Add or Create 新建或者创建一个仓库，通过 Repository -> Clone 签入一个仓库。

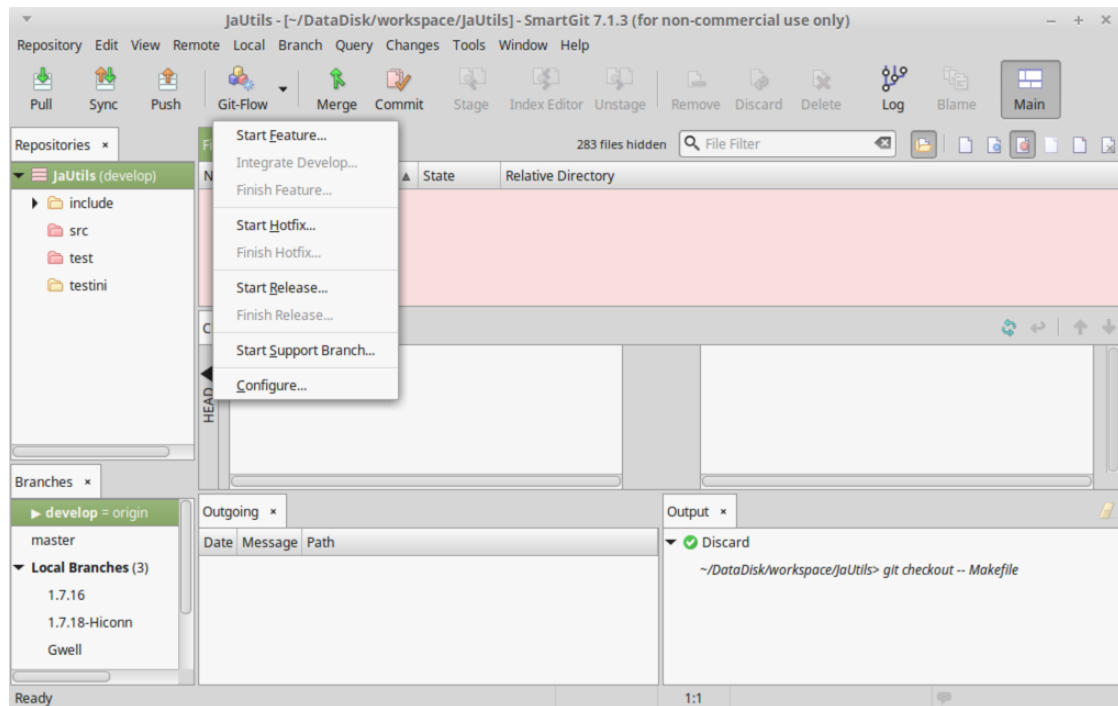


## 5.3. SmartGit 应用 git-flow 模型

### 5.3.1. 配置 git-flow

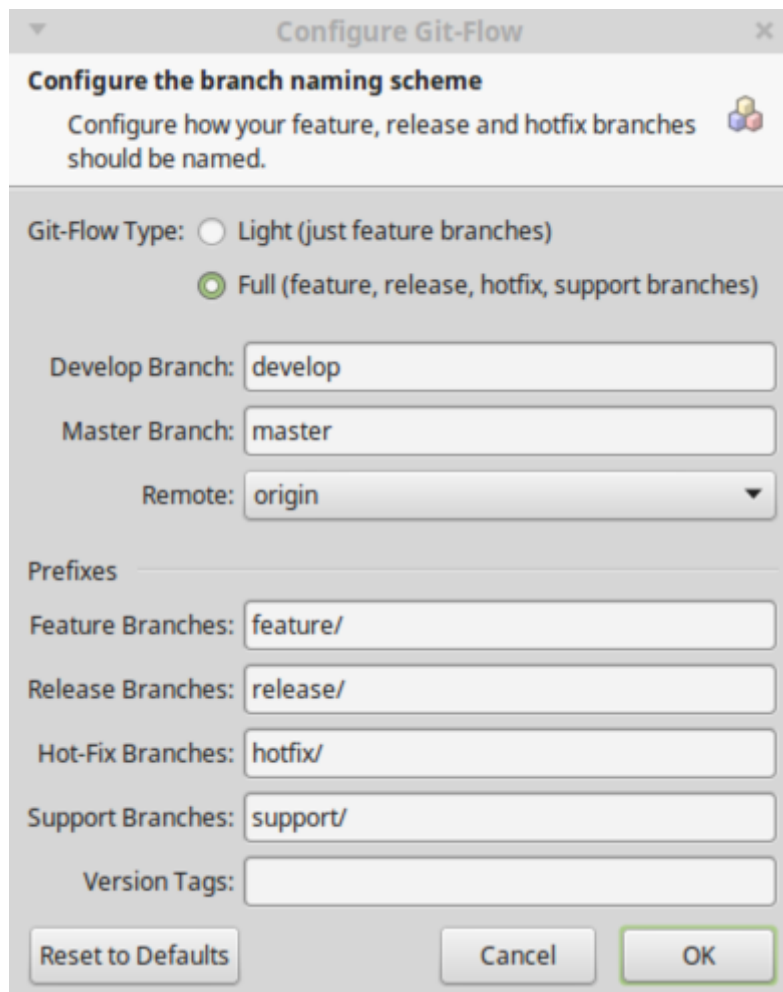


在 SmartGit 管理界面下，通过 `Menu > GitFlow > Configure`，进入 git-flow 配置界面。



在配置窗口，选择 `Full` 版本的 git-flow 才能满足我们的源码管理需要。  
在配置窗口中，可以指定每个 git-flow 分支的命名，这对于学习项目管理来说更加灵

活，SmartGit 会默认创建标准的路径名称，一般默认情况下不需要修改。



配置确定后，SmartGit 会在本地自动创建 `master` 和 `develop` 分支，之后便可使用 `git-flow` 进行源码管理。

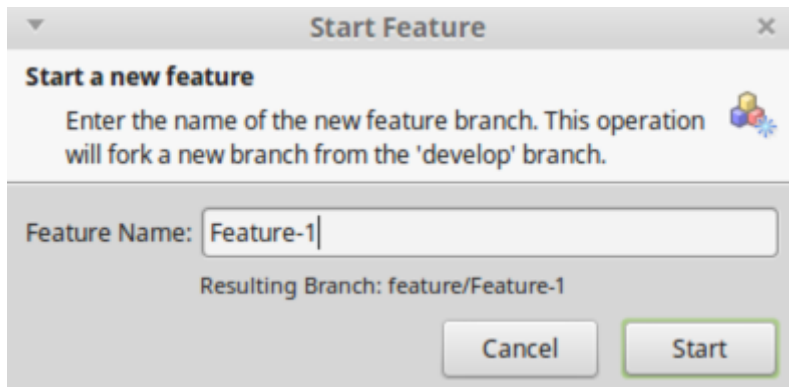
## 5.3.2. 功能开发

在源码管理过程中，必定遇到一个功能增加的需求，在 `git-flow` 模型下通过创建 `feature` 分支实现。

### 创建功能分支

如果当前 `develop` 分支，单纯点击 `GitFlow` 按钮即可创建一个 `feature` 分支，具体可以通过 SmartGit 可以很方便地实现，通过点击 `Menu->Git-Flow->Start Feature` 创建一

个 **feature** 分支，当然处女座的您可以为此功能分支作一个严格的命名。

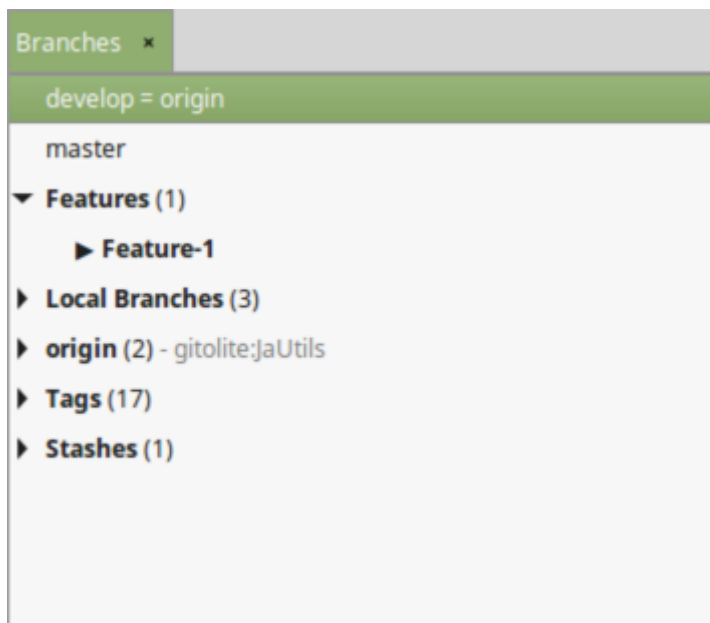


注意：

SmartGit 在 git-flow 模型下建议用户在 **master** 分支以外的分支提交代码，如果当前分支很不巧是 **master** 分支，SmartGit 会对用户有一个温馨的提示:.)。



创建成功后，在 **Branches** 窗口可以观察到 **Features** 树下多出一个分支。



在功能开发过程中，大部分时间在 **feature** 分支上操作提交，这样可以有效避免直接在 **develop** 分支上协同操作产生冲突的问题。

## 完成功能分支

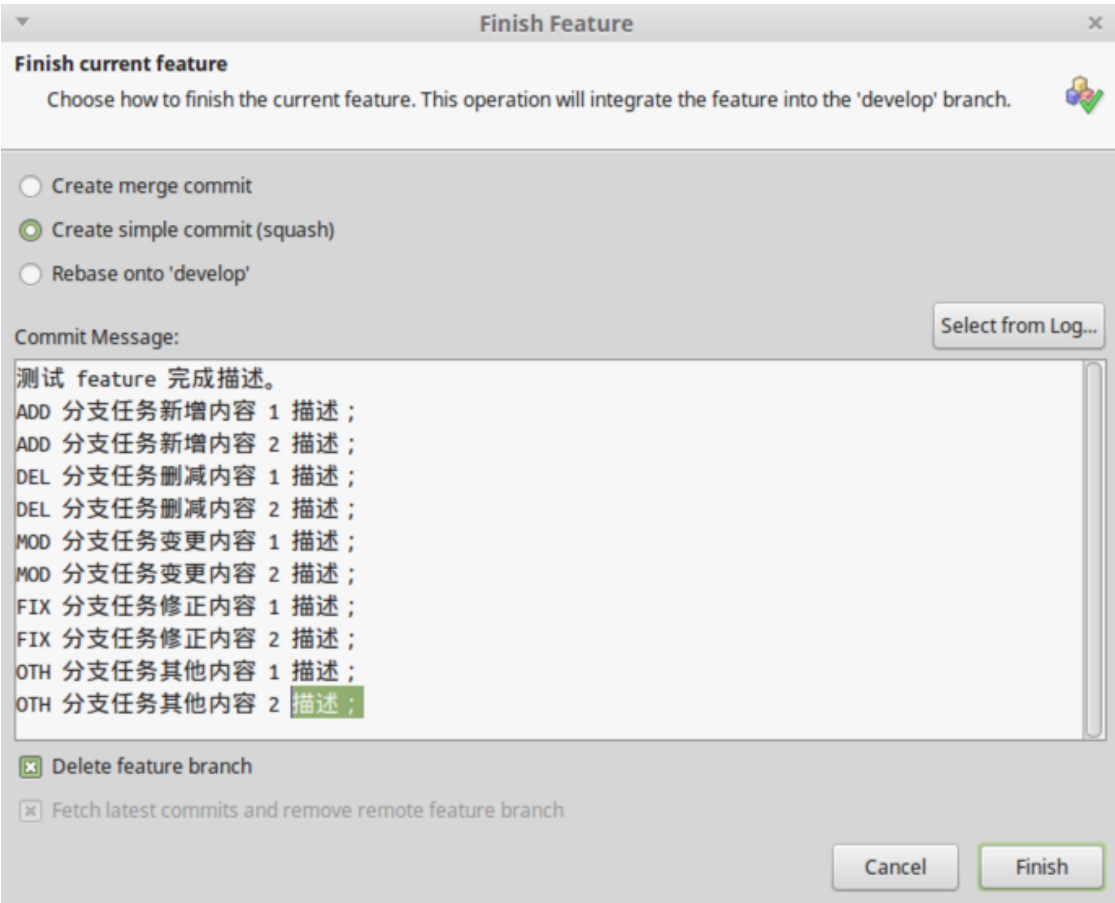
功能开发完成时，确认功能测试完成后，须要把功能分支合并到 **develop** 分支上，这时候可以使用 **Menu -> GitFlow -> Finish Feature** 选项。

在 **Finish Feature** 窗口中须要选择提交方式，这里只选择前面两种 **Create merge commit** 和 **Create simple commit(squash)**。

选择 **Create merge commit** 方式时，SmartGit 会在 **develop** 分支上保留 **feature** 分支迭代过程中的每个版本日志（内部使用 **git --no-ff**）方式，选择 **Create simple commit**

方式时，SmartGit 会去掉 `feature` 分支上的所有迭代日志，只在 `develop` 分支上保留提交日志。

根据项目实际情况去选择哪种提交方式，当功能内容较多，逻辑较复杂时，建议使用第一种提交方式，保留每次 `feature` 迭代日志，便于检查，当功能实现内容较少逻辑简单时，建议时候第二种方式，只需要言简意赅记录功能实现日志，降低阅读者翻查日志的难度。



## 功能提交日志规范

`feature` 分支提交必须按照严谨的日志规范，便于版本管理以及问题定位。

```
功能概述。
ADD 功能细节一；
ADD 功能细节二；
DEL 功能细节三。
```

日志分为两部分，一部分是 `功能概述`，一句话完成，描述功能的大致内容以 `。` 结束，占据一行。另一部分是 `功能变更细节`，此部分根据提交的复杂性可选，如果功能简单通过概述可以描述清楚则不需要功能细节内容。

`功能细节` 内容由多条信息组成，每条信息占一行，以 `；` 结束，最后一条信息以 `。` 结束。

细节信息中包括两部分，分别是信息分类和信息内容，中间用空格隔开。消息分类根据变

更的内容分为五类，分别是：新增（**ADD**）、修改（**MOD**）、删减（**DEL**）、修正（**FIX**）和其他（**OTH**），在对应的信息前描述。

## 解决冲突

略。

注意：

在功能开发过程中，有可能需要您紧急的把功能中的某个部分合并到 **develop** 分支，这时候可以使用 **Menu -> GitFlow -> Integrate Develop** 选项实现分支整合。



## 5.3.3. 版本发布

略。



## 5.3.4. 紧急修复

在版本已经发布以后，难免会遇到一个情况。一个功能逻辑存在缺陷或漏洞，需要紧急修复，这个时需要用到 **hotfix** 分支。

**hotfix** 分支从 **master** 分支上创建，因此创建前应该先把分支切换到 **master** 分支上。

选项 **Menu -> GitFlow -> Start Hotfix** 可以创建一个 **hotfix** 分支。



创建后见 **Branches** 列表框中 **Hotfixes** 分支列表。



当紧急修复完成后，选项 **Menu -> GitFlow -> Finish Hotfix** 完成，SmartGit 会分别尝试合并 **hotfix** 分支到 **develop** 和 **master** 分支上，并要求您提交日志和标注标签，此过程和版本发布类似。



注意：

在 **hotfix** 分支合并到 **develop** 分支的过程中，有可能产生冲突，此时 SmartGit 不会对 **hotfix** 进行删除，可以用用于检查，检查完毕解决冲突以后，用户需要手动提交 **develop** 分支上的内容。

最后，同步本地仓库到服务器 :)。

# 参考文献

---

- [SmartGit 图形化客户端简易使用教程](#)
- [A Successful Git Branching Model](#)
- [Using SmartGit to Follow the GitFlow Branching and Workflow model](#)