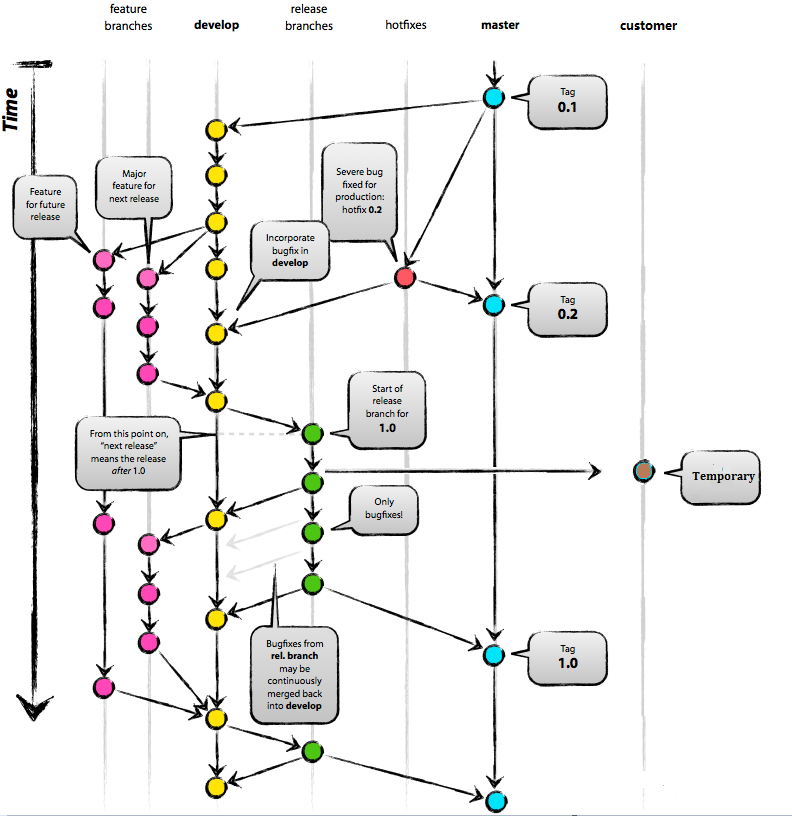
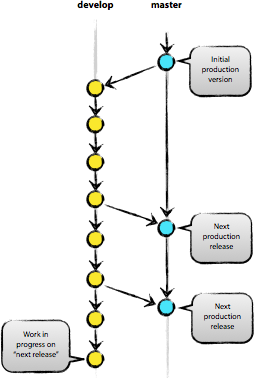
## 一、分支协作示意



它主要体现了Git对我们源代码版本的管理，下面将针对各分支之间协作展开介绍。

**二、**主分支



在核心部分，研发模型很大程度上靠其他现有模型支撑的。中心库有2个可一直延续的分支：

* master分支
* develop分支

每个Git用户都要熟悉原始的master分支。与master分支并行的另一个分支，我们称之为develop分支。

我们把原始库/master库认作为主分支，HEAD的源代码存在于此版本中，并且随时都是一个预备生产状态。

我们把origin/develop库认为是主分支，该分支HEAD源码始终体现下个发布版的最新软件变更。有人称这个为“集成分支”，而这是每晚自动构建得来的。

当develop分支的源码到达了一个稳定状态待发布，所有的代码变更需要以某种方式合并到master分支，然后标记一个版本号。如何操作将在稍后详细介绍。

所以，每次变更都合并到了master，这就是新产品的定义。在这一点，我们倾向于严格执行这一点，从而，理论上，每当对master有一个提交操作，我们就可以使用Git钩子脚本来自动构建并且发布软件到生产服务器。

备注：每次合并到master分支时，需要在此分支打一个tag，标示程序对应的版本和更新的内容。

**三、**辅助性分支

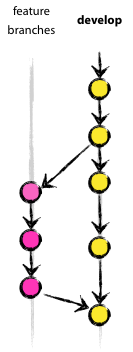
我们的开发模型使用了各种辅助性分支，这些分支与关键分支（master和develop）一起，用来支持团队成员们并行开发，使得易于追踪功能，协助生产发布环境准备，以及快速修复实时在线问题。与关键分支不同，这些分支总是有一个有限的生命期，因为他们最终会被移除。

我们用到的分支类型包括：

* 功能分支 feature
* 发布分支 release
* 热修复分支 hotfixes
* 客户分支 customer

每一种分支有一个特定目的，并且受限于严格到规则，比如：可以用哪些分支作为源分支，哪些分支能作为合并目标。从技术角度来看，这些分支绝不是特殊分支。分支的类型基于我们使用的方法来进行分类。它们理所当然是普通的Git分支。

**3.1 feature分支**



可能是develop分支的分支版本，最终必须合并到develop分支中。

分支命名规则：feature-项目名称\_功能标示。

功能分支（有时被称为topic分支）通常为即将发布或者未来发布版开发新的功能。当新功能开始研发，包含该功能的发布版本在这个还是无法确定发布时间的。功能版本的实质是只要这个功能处于开发状态它就会存在，但是最终会或合并到develop分支（确定将新功能添加到不久的发布版中）或取消（譬如一次令人失望的测试）。

功能分支通常存在于开发者的软件库，而不是在源代码库中。

**3.1.1创建一个功能分支**

开始一项功能的开发工作时，基于develop创建分支。

|  |  |
| --- | --- |
| 1 | $ git checkout -b myfeature develop |
| 2 | Switched to a new branch "myfeature" | |

**3.1.2合并一个功能到develop分支**

完成的功能可以合并进develop分支，以明确加入到未来的发布：

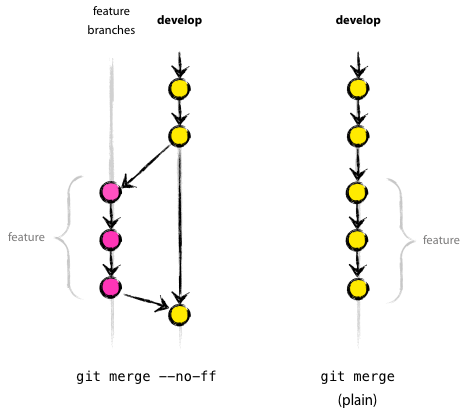
|  |  |
| --- | --- |
| 1 | $ git checkout develop |
| 2 | Switched to branch 'develop' | |

|  |  |  |
| --- | --- | --- |
| 3 | $ git merge --no-ff myfeature | |
| 4 | Updating ea1b82a..05e9557 |

|  |  |
| --- | --- |
| 5 | (Summary of changes) |
| 6 | $ git branch -d myfeature | |

|  |  |  |
| --- | --- | --- |
| 7 | Deleted branch myfeature (was 05e9557). | |
| 8 | $ git push origin develop |

--no-ff标志导致合并操作创建一个新commit对象，即使该合并操作可以fast-forward。这避免了丢失这个功能分支存在的历史信息，将该功能的所有提交组合在一起。 比较:

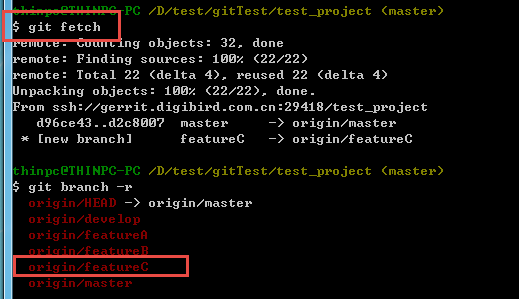


后一种情况，不可能从Git历史中看到哪些提交一起实现了一个功能——你必须手工阅读全部的日志信息。如果对整个功能进行回退 (比如一组提交)，后一种方式会是一种真正头痛的问题，而使用--no-ffflag的情况则很容易.

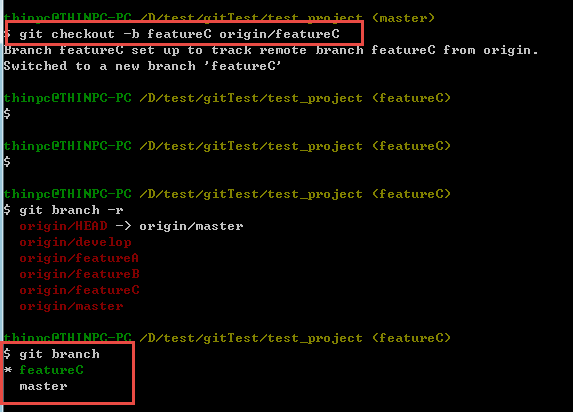
**3.1.3功能分支开发流程总结**

当项目组需要开发一个新功能模块时，可按以下流程展开：

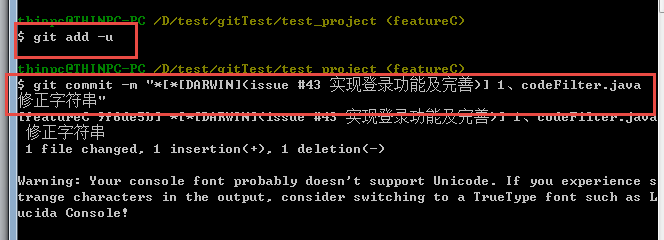
1. 由研发人员创建功能分支
2. 开发人员从服务器clone代码到本地（第一次），并且迁出功能分支到本地进行开发，以“featureC”功能分支为例：
   1. 使用 git fetch命令获取最新的分支数据



* 1. 创建本地featureC 分支：git checkout -b featureC origin/featureC



1. 开发人员可将功能模块细分为多个功能单元（便于小粒度的审核），每完成一个功能单元，进行一次提交，或者多次提交后rebase为一次提交



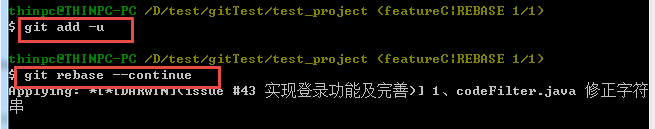
注意git add –u 指add 已加入版本库修改的文件，commit需要按代码提交注释规范进行编写，见：\\DSSERVER\Share\8\_项目相关\文档\规范.docx

1. 开发人员更新远程功能分支，并与本地代码进行rebase,如果有冲突则解决冲突



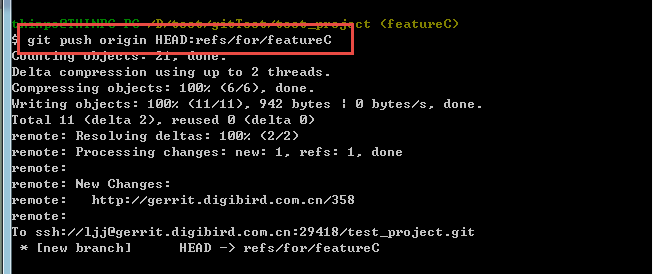


解决冲突然后使用add 和git rebase –continue命令完成rebase操作



1. 开发人员push代码到远程服务器

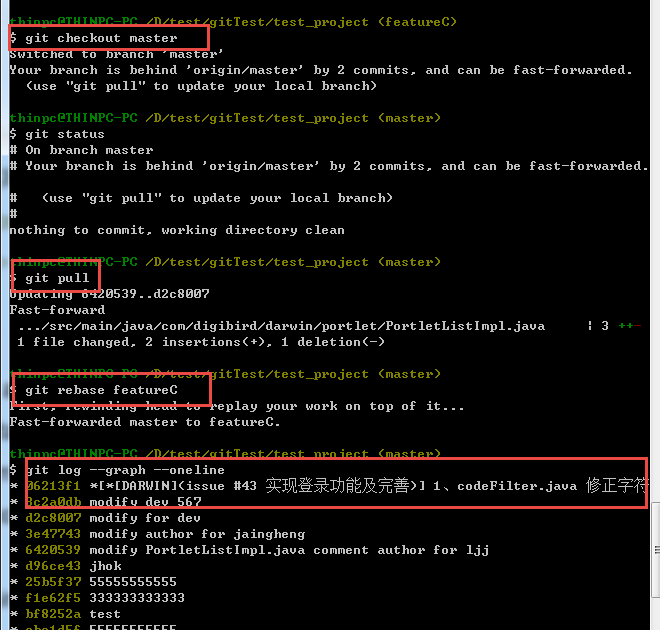
命令： push origin HEAD:refs/for/feature

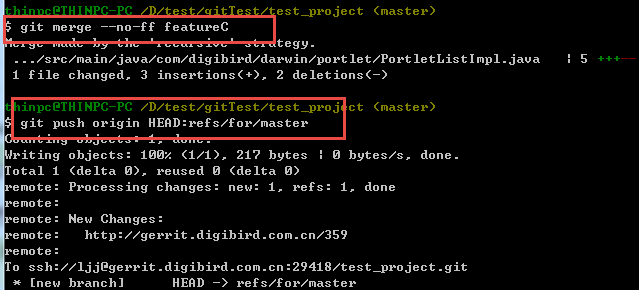


1. 集成服务进行集成测试，其它开发人员进行代码审核，如果审核失败，则开发人员修正问题后，使用amend commit提交代码后，再次push代码到远程服务器



1. 一个功能模块开发完毕后发送合并请求，由测试人员将特性分支并入develop分支，通知配置管理员删除远程分支；





注意这个例子使用的master分支，实际为Develop分支

1. 配置管理员删除远程分支，一个特性分支生命周期结束；

**3.2 Release分支**

Release分支可能从develop分支分离而来，但是一定要合并到develop和master分支上，它的习惯命名方式为：release-目标版本号（备注：dsoc项目组需要增加项目名称）。

Release分支是为新产品的发布做准备的。它允许我们在最后时刻做一些细小的修改。他们允许小bugs的修改和准备发布元数据（版本号，开发时间等等）。当在Release分支完成这些所有工作以后，对于下一次打的发布，develop分支接收features会更加明确。

从develop分支创建新的Release分支的关键时刻是develop分支达到了发布的理想状态。至少所有这次要发布的features必须在这个点及时合并到develop分支。对于所有未来准备发布的features必须等到Release分支创建以后再合并。

在Release分支创建的时候要为即将发行版本分配一个版本号，一点都不早。直到那时，develop分支反映的变化都是为了下一个发行版，但是在Release分支创建之前，下一个发行版到底叫0.3还是1.0是不明确的。这个决定是在Release分支创建时根据项目在版本号上的规则制定的。

**3.2.1创建一个release分支**

Release分支是从develop分支创建的。例如，当前产品的发行版本号为1.1.5，同事我们有一个大的版本即将发行。develop 分支已经为下次发行做好了准备，我们得决定下一个版本是1.2（而不是1.1.6或者2.0）。所以我们将Release分支分离出来，给一个能够反映新版本号的分支名。

|  |  |
| --- | --- |
| 1 | $ git checkout -b release-1.2 develop |
| 2 | Switched to a new branch "release-1.2" | |

|  |  |
| --- | --- |
| 3 | $ ./bump-version.sh 1.2 |
| 4 | Files modified successfully, version bumped to 1.2. | |

|  |  |
| --- | --- |
| 5 | $ git commit -a -m "Bumped version number to 1.2" |
| 6 | [release-1.2 74d9424] Bumped version number to 1.2 |

|  |  |
| --- | --- |
| 7 | 1 files changed, 1 insertions(+), 1 deletions(-) |

这个新分支可能会存在一段时间，直到该发行版到达它的预定目标。在此期间，bug的修复可能被提交到该分支上（而不是提交到develop分支上）。在这里严格禁止增加大的新features。他们必须合并到develop分支上，然后等待下一次大的发行版。

**3.2.2 完成一个Release分支**

当一个release分支准备好成为一个真正的发行版的时候，有一些工作必须完成。首先，release分支要合并到master上（因为每一次提交到master上的都是一个新定义的发行版，记住）。然后，提交到master上必须打一个标签，以便以后更加方便的引用这个历史版本。最后，在release分支上的修改必须合并到develop分支上，以便未来发行版也包含这些bugs的修复。

在Git中的前两步是：

|  |  |
| --- | --- |
| 1 | $ git checkout master |
| 2 | Switched to branch 'master' | |

|  |  |  |
| --- | --- | --- |
| 3 | $ git merge --no-ff release-1.2 | |
| 4 | Merge made by recursive. |

|  |  |  |
| --- | --- | --- |
| 5 | (Summary of changes) | |
| 6 | $ git tag -a 1.2 |

发行版现在已经完成，为以后引用打上标签。

为了是修改保持在release分支上，我们需要合并这些到develop分支上去，在Git上：

|  |  |
| --- | --- |
| 1 | $ git checkout develop |
| 2 | Switched to branch 'develop' | |

|  |  |  |
| --- | --- | --- |
| 3 | $ git merge --no-ff release-1.2 | |
| 4 | Merge made by recursive. |

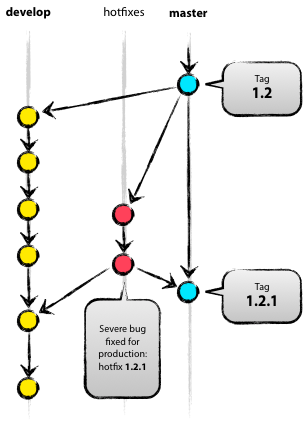
|  |  |
| --- | --- |
| 5 | (Summary of changes) |

这个步骤可能会导致合并冲突（可能由于改变版本号更是如此）。如果是这样，修复它然后提交。

现在我们真正的完成了，这个release分支将被删除，因为我们不再需要它了。

|  |  |
| --- | --- |
| 1 | $ git branch -d release-1.2 |
| 2 | Deleted branch release-1.2 (was ff452fe). | |

**3.3 hotfixes分支**



可以基于master分支，必须合并回develop和master分支。  
分支名约定：hotfix-\*

热修复分支与发布分支很相似，他们都为新的生成环境发布做准备，尽管这是未经计划的。他们来自生产环境的处于异常状态压力。当生成环境验证缺陷必须马上修复时，热修复分支可以基于master分支上对应与线上版本的tag创建。

其本质是团队成员（在develop分支上）的工作可以继续，而另一个人准备生产环境的快速修复。

**3.3.1创建一个修补bug分支**

hotfix branch(修补bug分支)是从Master分支上面分出来的。例如，1.2版本是当前生产环境的版本并且有bug。但是开发分支（develop）变化还不稳定。我们需要分出来一个修补bug分支（hotfix branch）来解决这种情况。

|  |  |
| --- | --- |
| 1 | $ git checkout -b hotfix-1.2.1 master |
|  | Switched to a new branch "hotfix-1.2.1" | |

|  |  |
| --- | --- |
| 3 | $ ./bump-version.sh 1.2.1 |
|  | Files modified successfully, version bumped to 1.2.1. | |

|  |  |
| --- | --- |
| 5 | $ git commit -a -m "Bumped version number to 1.2.1" |
| 6 | [hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1 | |

|  |  |
| --- | --- |
| 7 | 1 files changed, 1 insertions(+), 1 deletions(-) |

分支关闭的时侯不要忘了更新版本号(bump the version)

然后，修复bug，一次提交或者多次分开提交。

|  |  |
| --- | --- |
| 1 | $ git commit -m "Fixed severe production problem" |
| 2 | [hotfix-1.2.1 abbe5d6] Fixed severe production problem | |

|  |  |
| --- | --- |
| 3 | 5 files changed, 32 insertions(+), 17 deletions(-) |

**3.3.2创建一个hotfix分支**

完成一个bugfix之后，需要把bugfix合并到master和develop分支去，这样就可以保证修复的这个bug也包含到下一个发行版中。这一点和完成release分支很相似。

首先，更新master并对release打上tag：

|  |  |
| --- | --- |
| 1 | $ git checkout master |
| 2 | Switched to branch 'master' | |

|  |  |  |
| --- | --- | --- |
| 3 | $ git merge --no-ff hotfix-1.2.1 | |
| 4 | Merge made by recursive. |

|  |  |  |
| --- | --- | --- |
| 5 | (Summary of changes) | |
| 6 | $ git tag -a 1.2.1 |

下一步，把bugfix添加到develop分支中：

|  |  |
| --- | --- |
| 1 | $ git checkout develop |
| 2 | Switched to branch 'develop' | |

|  |  |  |
| --- | --- | --- |
| 3 | $ git merge --no-ff hotfix-1.2.1 | |
| 4 | Merge made by recursive. |

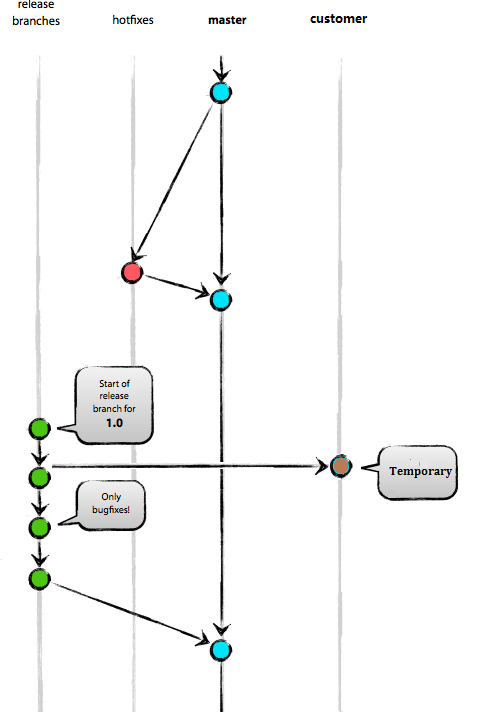
|  |  |
| --- | --- |
| 5 | (Summary of changes) |

规则的一个例外是：**如果一个release分支已经存在，那么应该把hotfix合并到这个release分支，而不是合并到develop分支。**当release分支完成后， 将bugfix分支合并回release分支也会使得bugfix被合并到develop分支。（如果在develop分支的工作急需这个bugfix，等不到release分支的完成，那你也可以把bugfix合并到develop分支）

最后，删除临时分支：

|  |  |
| --- | --- |
| 1 | $ git branch -d hotfix-1.2.1 |
| 2 | Deleted branch hotfix-1.2.1 (was abbe5d6). | |

**3.4 customer分支**



基于release分支，此分支如果有严重缺陷，客户不愿意升级版本时，可在此分支上创建hotfixes分支来修复缺陷，修复完缺陷将其代码合并到release分支或者develop分支。  
分支名约定：customer-\*

当使用customer分支对应程序版本的客户全部升级到其他分支版本时，此customer分支生命周期结束。