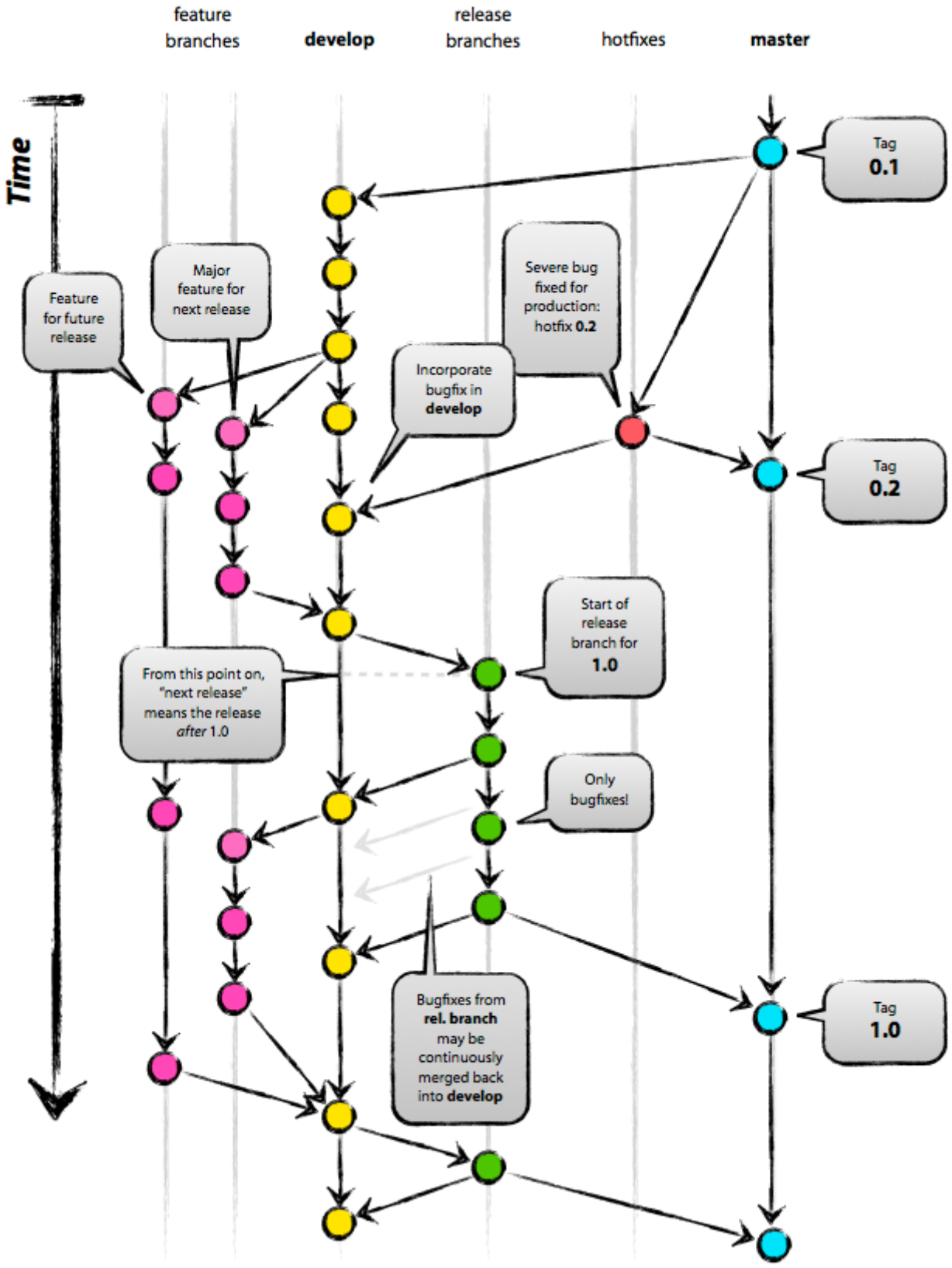
https://www.cnblogs.com/cnblogsfans/p/5075073.html

在这篇文章中，我提出一个开发模型。我已经将这个开发模型引入到我所有的项目里（无论在工作还是私人）已经一年有余，并且它被证明是非常成功的。我打算写这些已经很久了，但我一直找不到时间来做，现在终于有时间了。我不会讲任何项目的具体细节，仅是关于分支策略和释放管理相关内容。



它主要体现了Git对我们源代码版本的管理。

为何是Git？

对于Git与其他集中式代码管理工具相比的优缺点的全面讨论，请参见这里。这样的争论总是喋喋不休。作为一个开发者，与现今的其他开发工具相比较，我更

喜欢Git。Git真得改变了开发者对于合并和分支的思考。我曾经使用经典的CVS/Subversion，然而每次的合并/分支和其他行为总让人担惊受怕（“小心合并里的冲突，简直要命！”）。

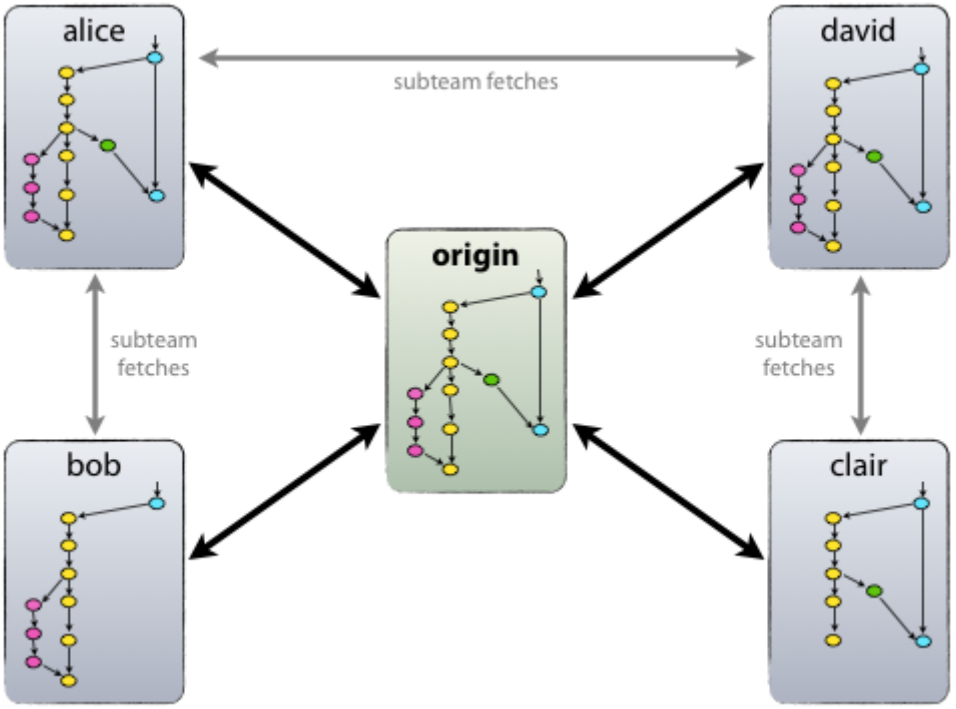
但是对于Git来说，这些行为非常简单和搞笑，它们被认为是*日常*工作中的核心部分。例如，在很多CVS/Subversion书里，分支与合并总是在后面的章节中被讨论（对于高级用户使用），然而在每个Git书中，在第3章就已经完全涵盖了（作为基础）。

简单和重复的特性带来的结果是：分支与合并不再是什么可以害怕的东西。分支/合并被认为对于版本管理工具比其他功能更重要。

关于工具，不再多说，让我们直接看开发模型吧。这个模型并不是如下模型：在管理软件开发进度方面，面对每个开发过程，每个队员必须按一定次序开发。

分布式而非集中式

对于这种分支模型，我们设置了一个版本库，它运转良好，这是一个"事实上"版本库。不过请注意，这个版本库只是被*认为*是中心版本库（因为Git是一个分布式版本管理系统，从技术上来讲，并没有一个中心版本库）。我们将把这个版本库称为原始库，这个名字对所有的Git用户来说都很容易理解。

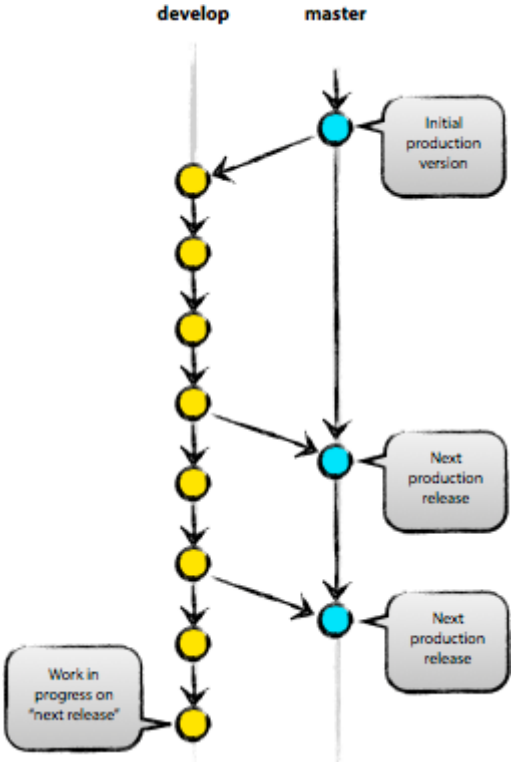


每个开发者都对origin库拉代码和提交代码。但是除了集中式的存取代码关系，每个开发者也可以从子团队的其他队友那里获得代码版本变更。例如，对于2个或多个开发者一起完成的大版本变更，为了防止过早地向origin库提交工作内

容，这种机制就变得非常有用。在上述途中，有如下子团队：Alice和Bob，Alice和David，Clair和David。

从技术上将，这意味着，Alice创建了一个Git的远程节点，而对于Bob，该节点指向了Bob的版本库，反之亦然。

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。

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。

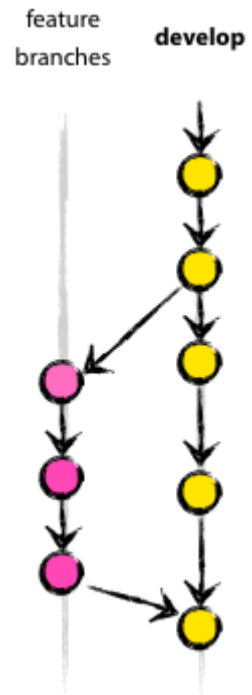
我们用到的分支类型包括：

- 功能分支
- 发布分支
- 热修复分支

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。我们马上将进行演练。

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。

功能分支



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：除了master、develop、release-*、orhotfix-*之外，其他命名均可。

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。

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。

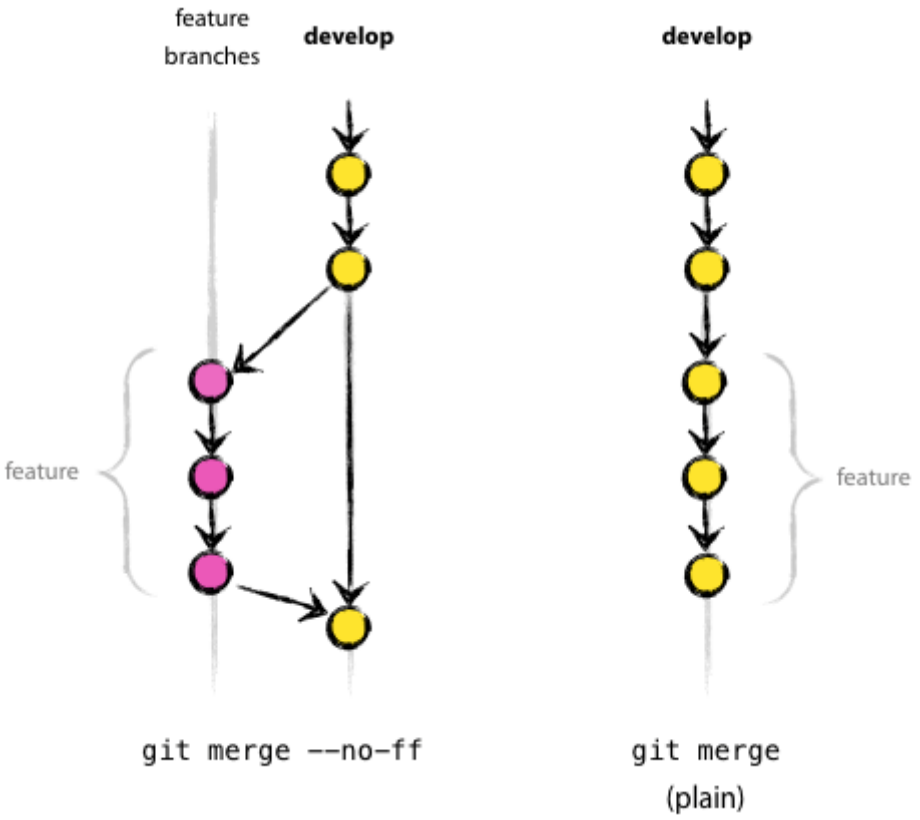
```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

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：

```
$ 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.

是的，它会创建一个新的（空）提交对象，但是收益远大于开销。

不幸的是，我还没找到一种方法，让--no-ff时作为合并操作的默认选项，但它应该是可行的。

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-*。

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。

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。

```
$ 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(-)
```

创建新分支以后，切换到该分支，添加版本号。这里，bump-version.sh 是一个虚构的shell脚本，它可以复制一些文件来反映新的版本（这当然可以手动改变--目的就是修改一些文件）。然后版本号被提交。

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