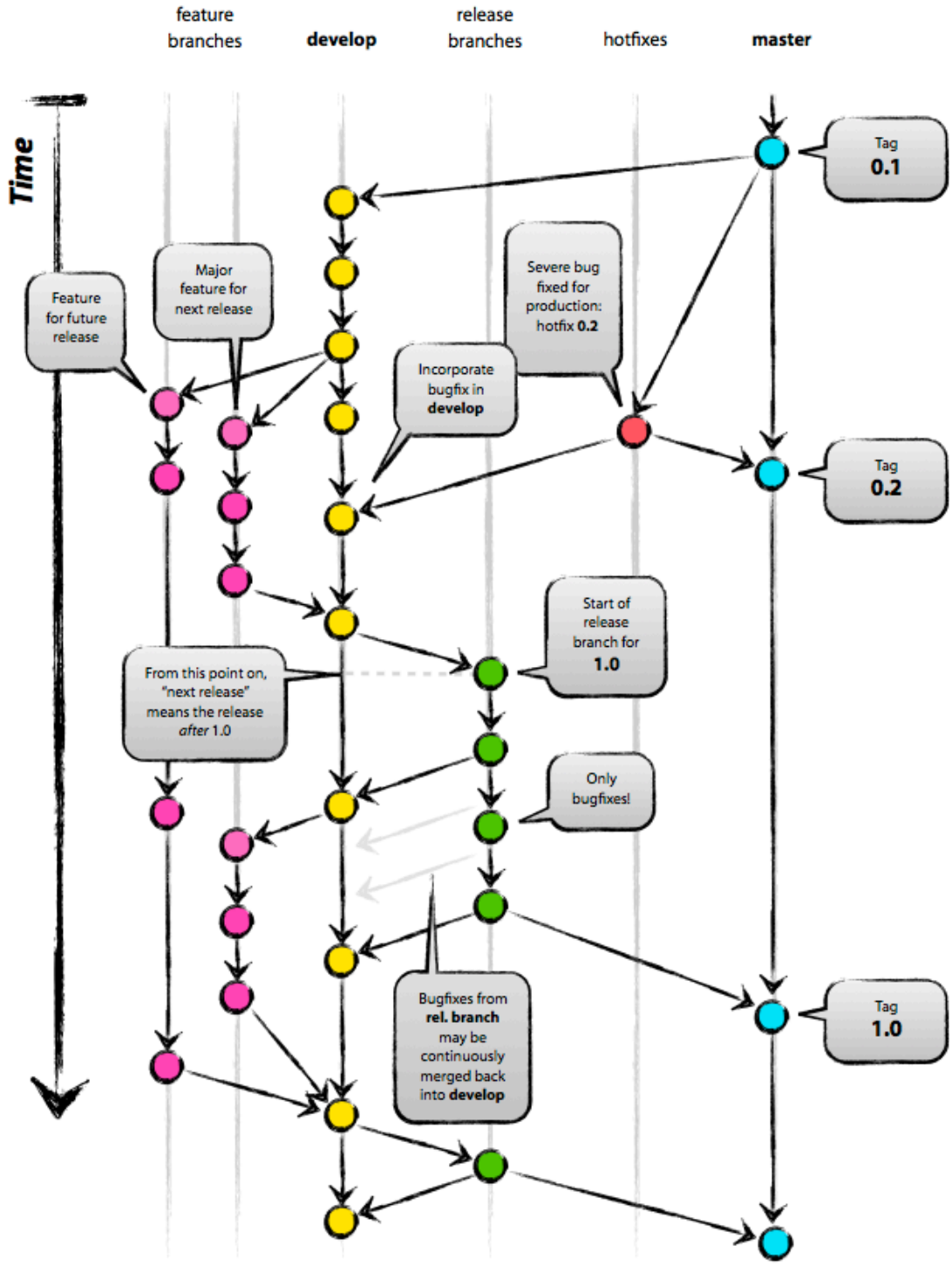
一个成功的 Git 分支模型

本文中我会展示一种开发模型，一年前该模型就已经被我用在所有的项目中（包括工作中的项目和私有项目），结果是非常成功的。我早就想为此写点东西，可直到现在才有时间。本文不会讲述任何项目的细节，只会涉及到分支策略和发布管理。



本文使用 Git 作为所有源码的版本控制工具。

为什么是 Git?

要全面了解 Git 与其它集中式版本控制系统相比的优劣，可以参考这个页面。这方面的争论可谓是硝烟弥漫。作为一个开发者，所有这些工具中我最钟情于 Git。Git 的的确确改变了人们考虑合并及分支的方式。在我之前所处的经典 CVS/Subversion 世界中，合并/分支总是被认为是有点可怕的事情（“小心合并冲突，丫会恶心到你”），因此你只应偶尔干这种事情。

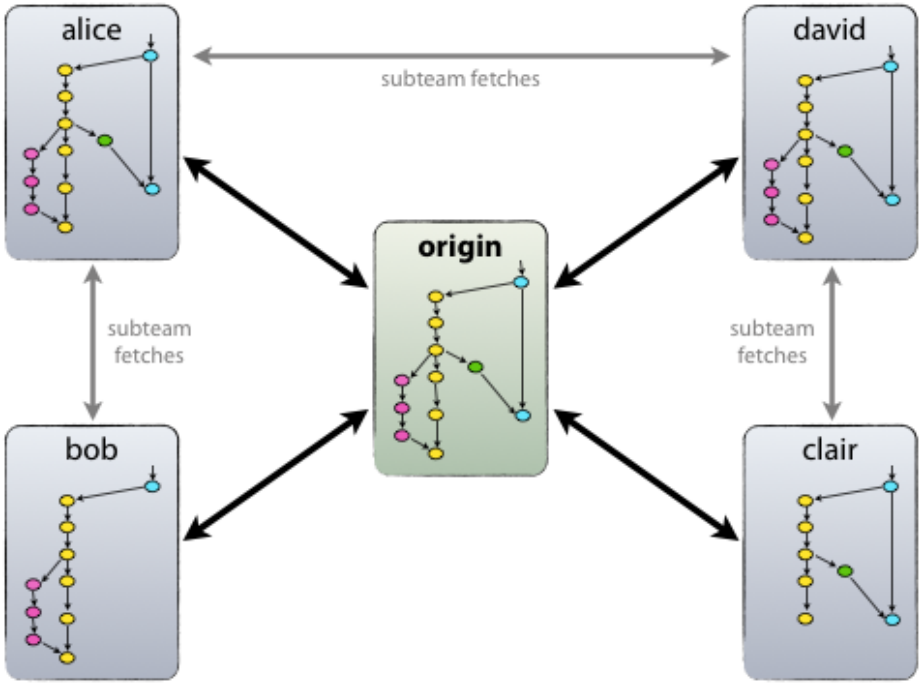
但有了 Git，这类事情就变得非常简单，分支及合并甚至被认为是你日常版本控制操作的核心之一。例如，在 CVS/Subversion 的书中，分支及合并往往在后面的章节才被介绍（针对高级用户），但在每一本 Git 的书中，该内容已经在前 3 章中介绍（基础）。

简单及易重复性带来的好处就是，分支及合并变得不再可怕。版本控制工具本该帮助我们方便的进行和分支及合并操作。

简单介绍下工具后，让我们来看开发模型。我讲介绍的模型本质上只是一组步骤，每个团队成员都必须遵循这些步骤以形成一个可靠管理的软件开发过程。

去中心化但仍保持中心化

在这个分支模型中我们使用的，且被证实工作得很好的仓库配置，其核心是一个中心“真理”仓库。注意只有该仓库才被认为是中心库（由于 Git 是 DVCS [分布式版本控制系统]，在技术层面没有中心库这一东西）。之后我们用 origin 指代该仓库，因为大多数 Git 用户都熟悉这个名称。

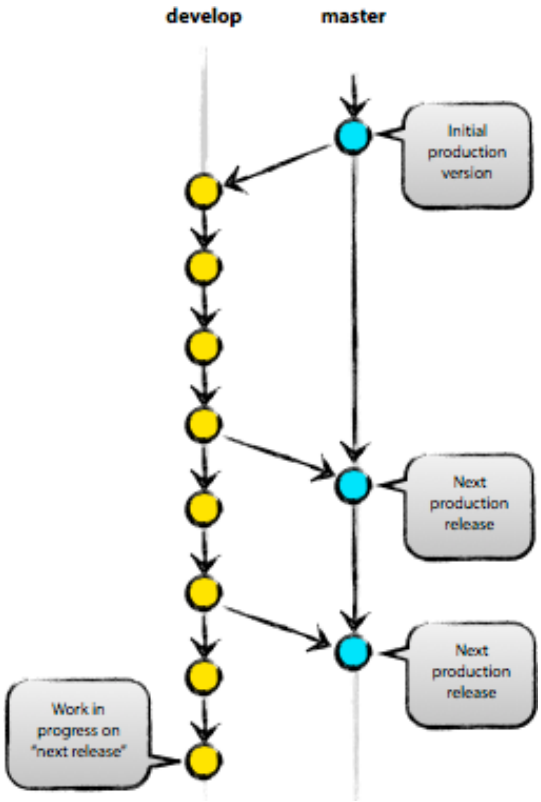


每个开发者都对 origin 做 push 和 pull 操作。不过除了这种中心化的 push-pull

关系外，每个开发者还可以从其他开发者或者小组处 pull 变更。例如，可能两个或更多的开发者一起开发一个大的特性，在往 origin 永久性的 push 工作代码之前，他们之间可以执行一些去中心化的操作。在上图中，分别有 Alice 和 Bob、Alice 和 David、Clair 和 David 这些小组。

从技术上来说，这仅仅是 Alice 定义一个 Git remote，名字为 bob，指向 Bob 的仓库，反过来也一样。

主要分支



此开发模型的核心主要受现有的模型启发。中心仓库包含了两个主要分支，这两个分支的寿命是无限的：

- master
- develop

每个 Git 用于都应该熟悉 origin 上的 master 分支。与 master 分支平行存在的，是另外一个名为 develop 的分支。

我们认为 origin/develop 分支上的 HEAD 源码反映了开发过程中最新的提交变更。有人会称之为“集成分支”。该分支是自动化每日构建的代码源。

当 develop 分支上的源码到达一个稳定的状态时，就可以发布版本。所有 develop 上的变更都应该以某种方式合并回 master 分支，并且使用发布版本号打上标签。稍后我们会讨论具体操作细节。

因此，每次有变化被合并到 master 分支时，根据定义这就是一次新的产品版本发布。我们趋向于严格遵守该规范，所以理论上来说，每次 master 有提交时，我们都可以使用一个 Git 钩子（hook）脚本来自动构建并部署软件至产品环境服务器。

支持性分支

紧接着主要分支 master 和 develop，我们的开发模型使用多种支持性分支来帮助团队成员间实现并行开发、追踪产品特性、准备产品版本发布、以及快速修复产品问题。与主要分支不同的是，这些分支的寿命是有限的，它们最终都会被删除。

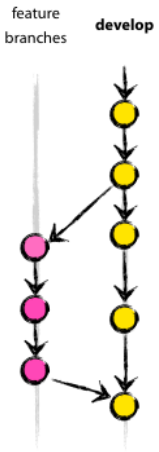
我们会用到的分支有这几类：

- 特性分支（feature branch）
- 发布分支（release branch）
- 热补丁分支（hotfix branch）

上述每种分支都有特定的用途，它们各自关于源自什么分支、合并回什么分支，都有严格的规定。稍后我们逐个进行介绍。

从技术角度来说，这些分支一点都不“特殊”。分支按照我们对其的使用方式进行分类。技术角度它们都一样是平常的 Git 分支。

特性分支



可能的分支来源：develop

必须合并回：develop

分支命令约定：任何除 master, develop, release-*, 或 hotfix-*以外的名称

特性分支（有时也被称作 topic 分支）是用来为下一发布版本开发新特性。当开始开发一个特性的时候，该特性会成为哪个发布版本的一部分，往往还不知道。特性分支的重点是，只要特性还在开发，该分支就会一直存在，不过它最终会被

合并回 develop 分支（将该特性加入到发布版本中），或者被丢弃（如果试验的结果令人失望）。

特性分支往往只存在于开发者的仓库中，而不会出现在 origin。

创建一个特性分支

开始开发新特性的时候，从 develop 分支创建特性分支。

$ git checkout -b myfeature develop

Switch to a new branch “myfeature”

合并完成的特性回 develop

完成的特性应该被合并回 develop 分支以将特性加入到下一个发布版本中：

***$* git checkout develop**

Switch to branch ‘develop’

$ git merge –no-ff myfeature

Updating ea1b82a..05e9557

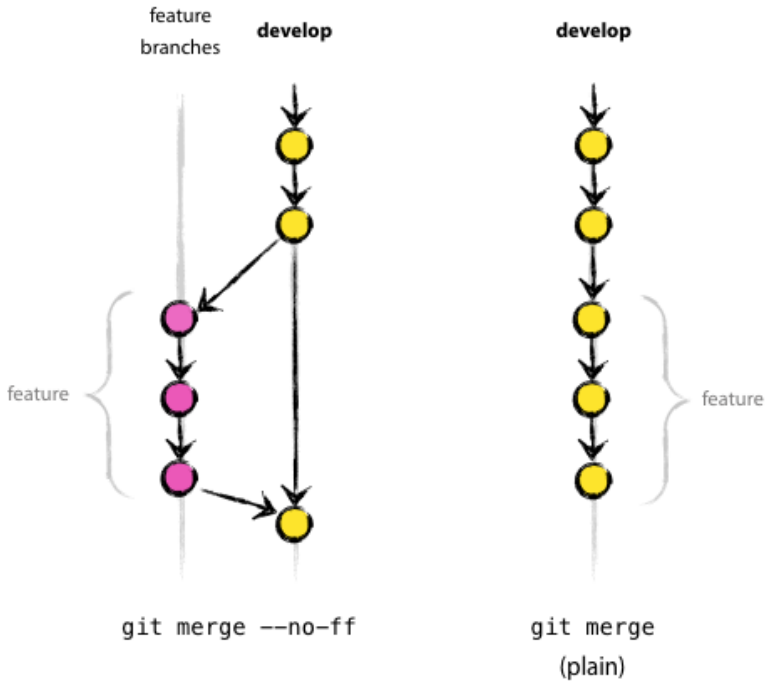
(Summary of changes)

$ git branch -d myfeature

Deleted branch myfeature (was 05e9557).

$ git push origin develop

上述代码中的–no-ff 标记会使合并永远创建一个新的 commit 对象，即使该合并能以 fast-forward 的方式进行。这么做可以避免丢失特性分支存在的历史信息，同时也能清晰的展现一组 commit 一起构成一个特性。比较下面的图：



在第 2 张图中，已经无法一眼从 Git 历史中看到哪些 commit 对象构成了一个特性——你需要阅读日志以获得该信息。在这种情况下，回退（revert）整个特性（一组 commit）就会比较麻烦，而如果使用了–no-diff 就会简单很多。

是的，这么做会造成一些（空的）commit 对象，但这么做是利大于弊的。

可惜的是，我没能找到方法让–no-diff 成为默认的 git merge 行为参数，但其实应该这么做。

发布分支

可能的分支来源：develop

必须合并回：develop 和 master

分支命名约定：release-*

发布分支为准备新的产品版本发布做支持。它允许你在最后时刻检查所有的细节。此外，它还允许你修复小 bug 以及准备版本发布的元数据（例如版本号，构建日期等等）。在发布分支做这些事情之后，develop 分支就会显得比较干净，也方便为下一大版本发布接受特性。

从 develop 分支创建发布分支的时间通常是 develop 分支（差不多）能反映新版本所期望状态的时候。至少说，这是时候版本发布所计划的特性都已经合并回了 develop 分支。而未来其它版本发布计划的特性则不应该合并，它们必须等到当前的版本分支创建好之后才能合并。

正是在发布分支创建的时候，对应的版本发布才获得一个版本号——不能更早。在该时刻之前，develop 分支反映的是“下一版本”的相关变更，但不知道这“下一版本”到底会成为 0.3 还是 1.0，直到发布分支被创建。版本号是在发布分支创建时，基于项目版本号规则确定的。

创建一个发布分支

发布分支从 develop 分支创建。例如，假设 1.1.5 是当前的产品版本，同时我们即将发布下个版本。develop 分支的状态已经是准备好“下一版本”发布了，我们也决定下个版本是 1.2（而不是 1.1.6 或者 2.0）。因此我们创建发布分支，并且为其赋予一个能体现新版本号的名称：

$ git checkout -b releases-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(-)

创建新分支并转到该分支之后，我们设定版本号。这里的 bump-version.sh 是一个虚构的 shell 脚本，它修改一些本地工作区的文件以体现新的版本号。（当然这也可以手动完成——这里只是说要有一些文件变更）接着，提交新版本号。

新的发布分支可能存在一段时间，直到该版本明确对外交付。这段时间内，该分支上可能会有一些 bug 的修复（而不是在 develop 分支上）。在该分支上添加新特性是严格禁止的。新特性必须合并到 develop 分支，然后等待下一个版本发布。

结束一个发布分支

当发布分支达到一个可以正式发布的状态时，我们就需要执行一些操作。首先，将发布分支合并至 master（记住，我们之前定义 master 分支上的每一个 commit 都对应一个新版本）。接着，master 分支上的 commit 必须被打上标签（tag），以方便将来寻找历史版本。最后，发布分支上的变更需要合并回 develop，这样将来的版本也能包含相关的 bug 修复。

前两步在 Git 中的操作：

$ git checkout master

Switched to branch ‘master’

$ git merge –no-ff release-1.2

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2

现在版本发布完成了，而且为未来的查阅提供了标签。

提醒： 你可能同时也会想要用 -s 或者 -u <key> 来对标签进行签名。

为了能保留发布分支上的变更，我们还需要将分支合并回 develop。在 Git 中：

$ git checkout develop

Switched to branch ‘develop’

$ git merge –no-ff release-1.2

Merge made by recursive.

(Summary of changes)

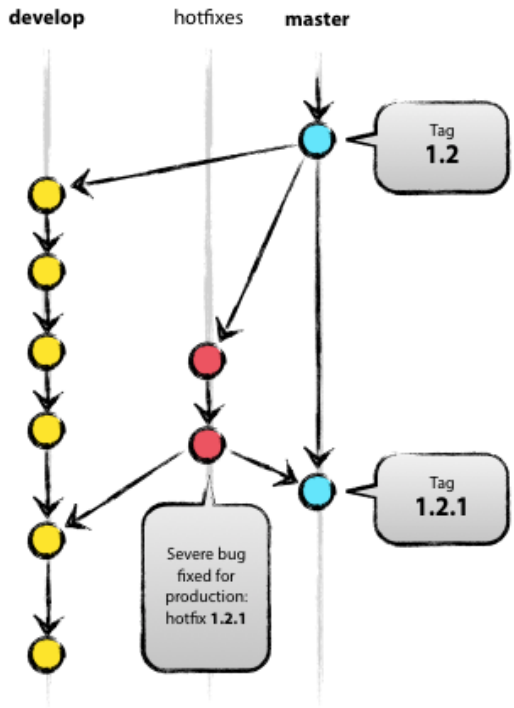
这一操作可能会导致合并冲突（可能性还很大，因为我们改变了版本号）。如果发现，则修复之并提交。

现在工作才算真正完成了，最后一步是删除发布分支，因为我们已不再需要它：

$ git branch -d release-1.2

Deleted branch release-1.2 (was ff452fe).

热补丁分支



可能的分支来源：master

必须合并回：develop 和 master

分支命名约定：hotfix-*

热补丁分支和发布分支十分类似，它的目的也是发布一个新的产品版本，尽管是不在计划中的版本发布。当产品版本发现未预期的问题的时候，就需要理解着手处理，这个时候就要用到热补丁分支。当产品版本的重大 bug 需要立即解决的时候，我们从对应版本的标签创建出一个热补丁分支。

使用热补丁分支的主要作用是（develop 分支上的）团队成员可以继续工作，而另外的人可以在热补丁分支上进行快速的产品 bug 修复。

创建一个热补丁分支

热补丁分支从 master 分支创建。例如，假设 1.2 是当前正在被使用的产品版本，由于一个严重的 bug，产品引起了很多问题。同时，develop 分支还处于不稳定状态，无法发布新的版本。这时我们可以创建一个热补丁分支，并在该分支上修复问题：

$ git checkout -b hotfix-1.2.1 master

Switched to a new branch “hotfix-1.2.1″

$ ./bump-version.sh 1.2.1

Files modified successfully, version bumped to 1.2.1.

$ git commit -a -m “Bumped version number to 1.2.1″

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

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

不要忘了在创建热补丁分之后设定一个新的版本号！

然后，修复 bug 并使用一个或者多个单独的 commit 提交。

$ git commit -m “Fixed severe production problem”

[hotfix-1.2.1 abbe5d6] Fixed severe production problem

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

结束一个热补丁分支

修复完成后，热补丁分支需要合并回 master，但同时它还需要被合并回 develop，这样相关的修复代码才会同时被包含在下个版本中。这与我们完成发布分支很类似。

首先，更新 master 分支并打上标签。

$ git checkout master

Switched to branch ‘master’

$ git merge –no-ff hotfix-1.2.1

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2.1

提醒： 你可能同时也会想要用 -s 或者 -u <key> 来对标签进行签名。

接着，将修复代码合并到 develop：

$ git checkout develop

Switched to branch ‘develop’

$ git merge –no-ff hotfix-1.2.1

Merge made by recursive.

(Summary of changes)

这里还有个例外情况，**如果这个时候有发布分支存在，热补丁分支的变更则应该合并至发布分支，而不是 develop**。将热补丁合并到发布分支，也意味着当发布分支结束的时候，变更最终会被合并到 develop。（如果 develop 上的开发工作急需热补丁并无法等待发布分支完成，这时你也已经可以安全地将热补丁合并到 develop 分支。）

最后，删除临时的热补丁分支：

$ git branch -d hotfix-1.2.1

Deleted branch hotfix-1.2.1 (was abbe5d6).

小结

虽然这个分支模型中没有什么特别新鲜的东西，但本文起始处的“全景图”事实上在我们的项目中起到了非常大的作用。它帮助建立了优雅的，易理解的概念模型，使得团队成员能够快速建立并理解一个公用的分支和发布过程。