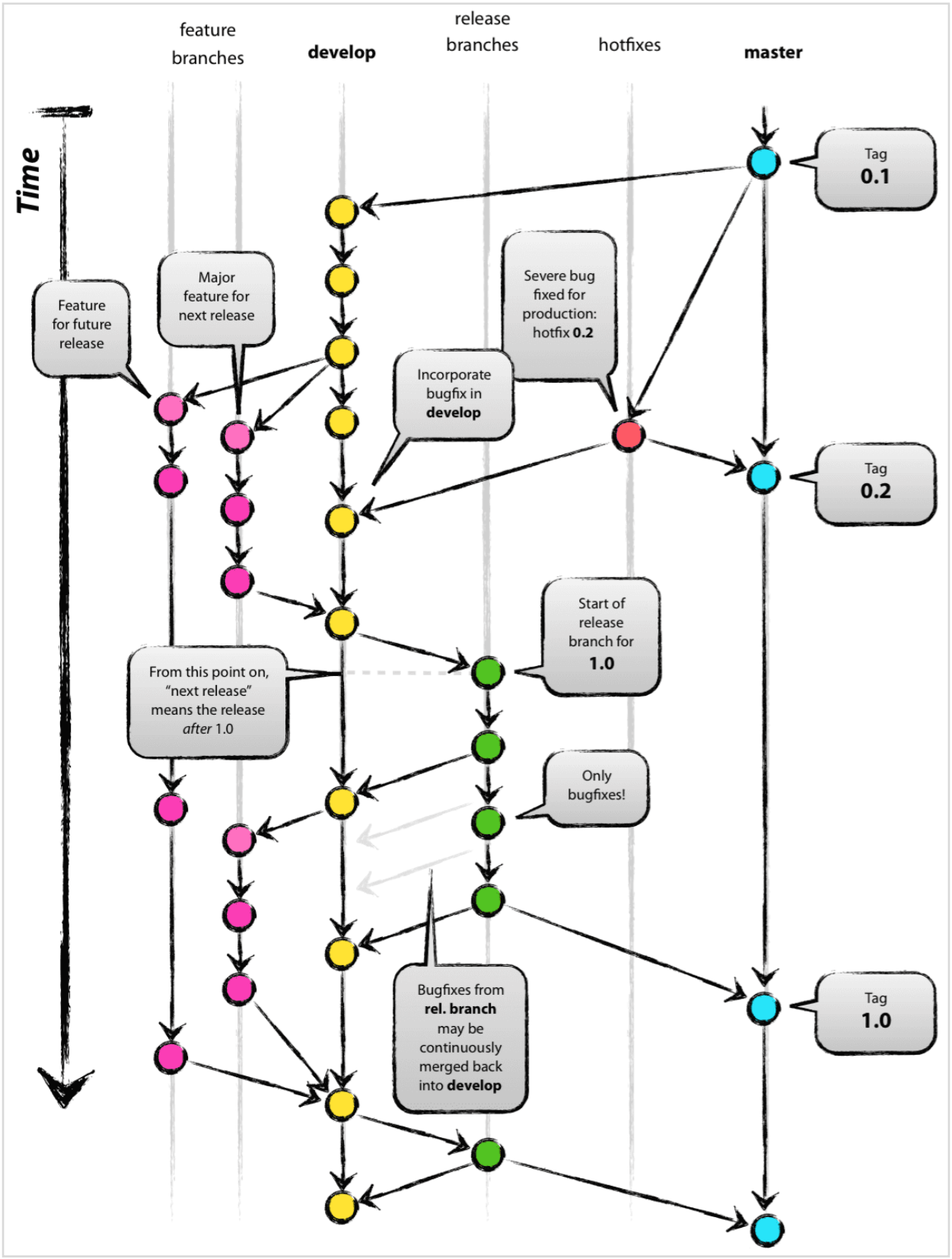
Git 工作流



git flow 完整图示

Git 开发模式本质上是一套流程，团队每个成员遵守这套流程以确保完成可控的软件开发过程。

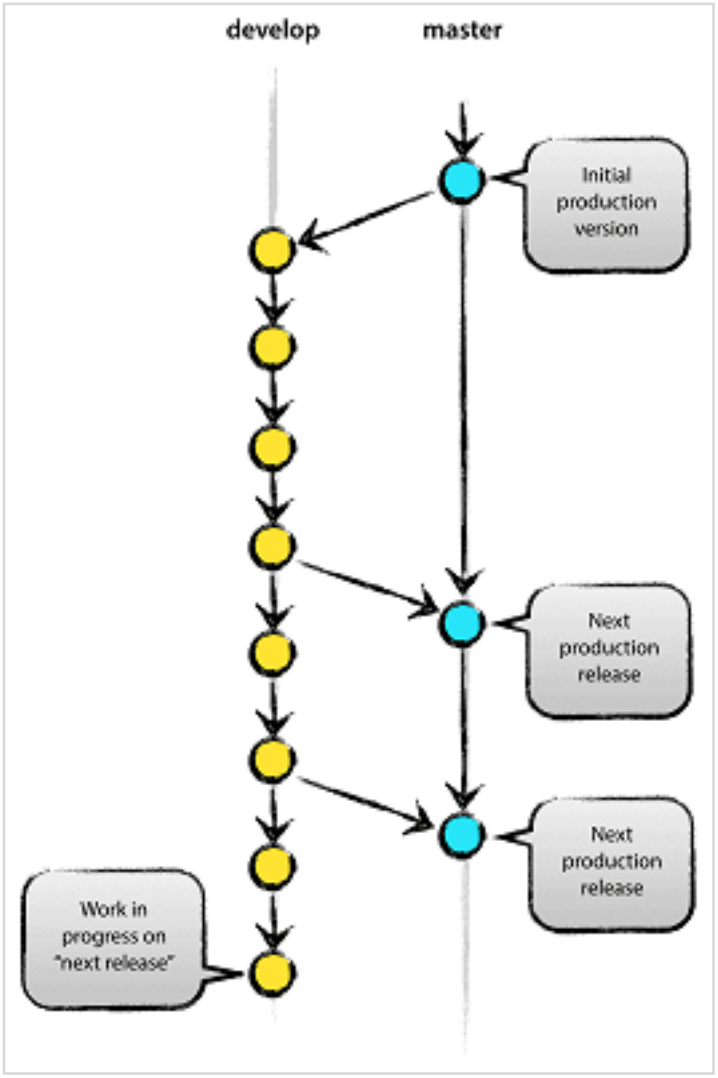
原文参考

1.主要分支

在远程仓库中有两个主要分支的生命期可以无限长，分别是：

* `Master` *

* `Develop` *



develop 和 master 关系图

master 分支（*origin/master*）

代码仓库中有且仅有的一条主分支，默认为 master ，在创建版本库时会自动创建。所有提供给用户使用的正式版本的源码，都会在这个分支上发布。也就是说主分支 master 用来发布重大版

本。

develop 分支（*origin/develop*）

日常开发工作都会在 develop 分支上面完成。develop 分支可以用来生成代码的最新隔夜版本（nightly builds）。

创建`develop`分支

```
$ git checkout -b develop master \#push develop 到远程仓库 $ git push origin develop
```

当我们在`develop`上完成了新版本的功能，最终会把所有的修改`merge`到 *`master`* 分支。针对每次 *`master`* 的修改都会打一个 Tag 作为可发布产品的版本号。

2.辅助分支

开发过程中不可能项目人所有都在一个`develop`分支中开发，版本管理会很混乱。所以除了主要分支外，我们还需要一些辅助分支来协助团队成员间的并行开发。

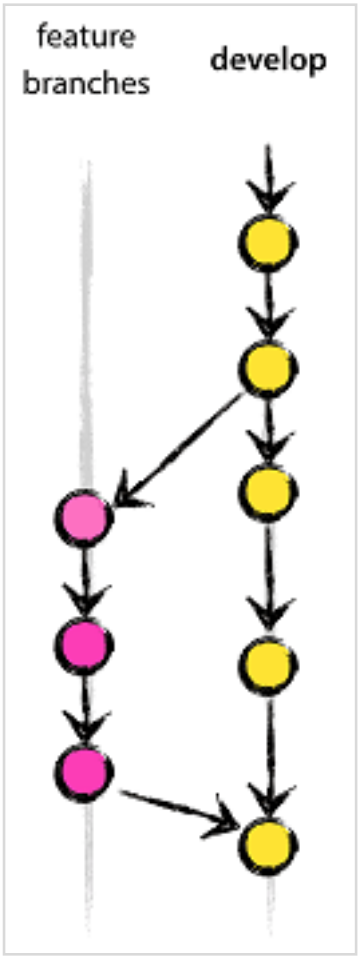
所用到的辅助分支大体分三类：

- *Feature branches*（功能分支）
- *Release branches*（预发布分支）
- *Hotfix branches*（热修复分支）

通过分支名我们能知道各类型分支都有特定作用，对于他们各自的起始分支和最终的合并分支也都有严格规定。呼，虽然可能会麻烦点，但让人一目了然的效果还是很诱人的。

下面逐一介绍下各类型分支的创建使用和移除方法，过程中我在`Github`中创建一个虚拟的项目用来熟悉整个流程，或许你也可以像我一样做一遍。哈，动手总会有意外收获嘛。废话少说，继续正题～

2.1.Feature branches（功能分支）



feature branches

应用场景：

当要开始一个新功能的开发时，我门可以创建一个`Feature branche`。等待这个新功能开发完成并确定应用到新版本中就合并回`develop`，那么如果不是就会被很遗憾的丢弃。。。

应用规则：

1. 从`develop`分支创建，最终合并回`develop`分支;
2. 分支名：feature-*;

Tips：这里很多地方说用 *feature-** 的方式命名，因为公司项目中用的 *feature-** 方式，也就习惯了，其实意思是一样的。

(1).Creat a feature branch

```
$ git checkout -b feature-test develop
```

do something in `feature-test` branch

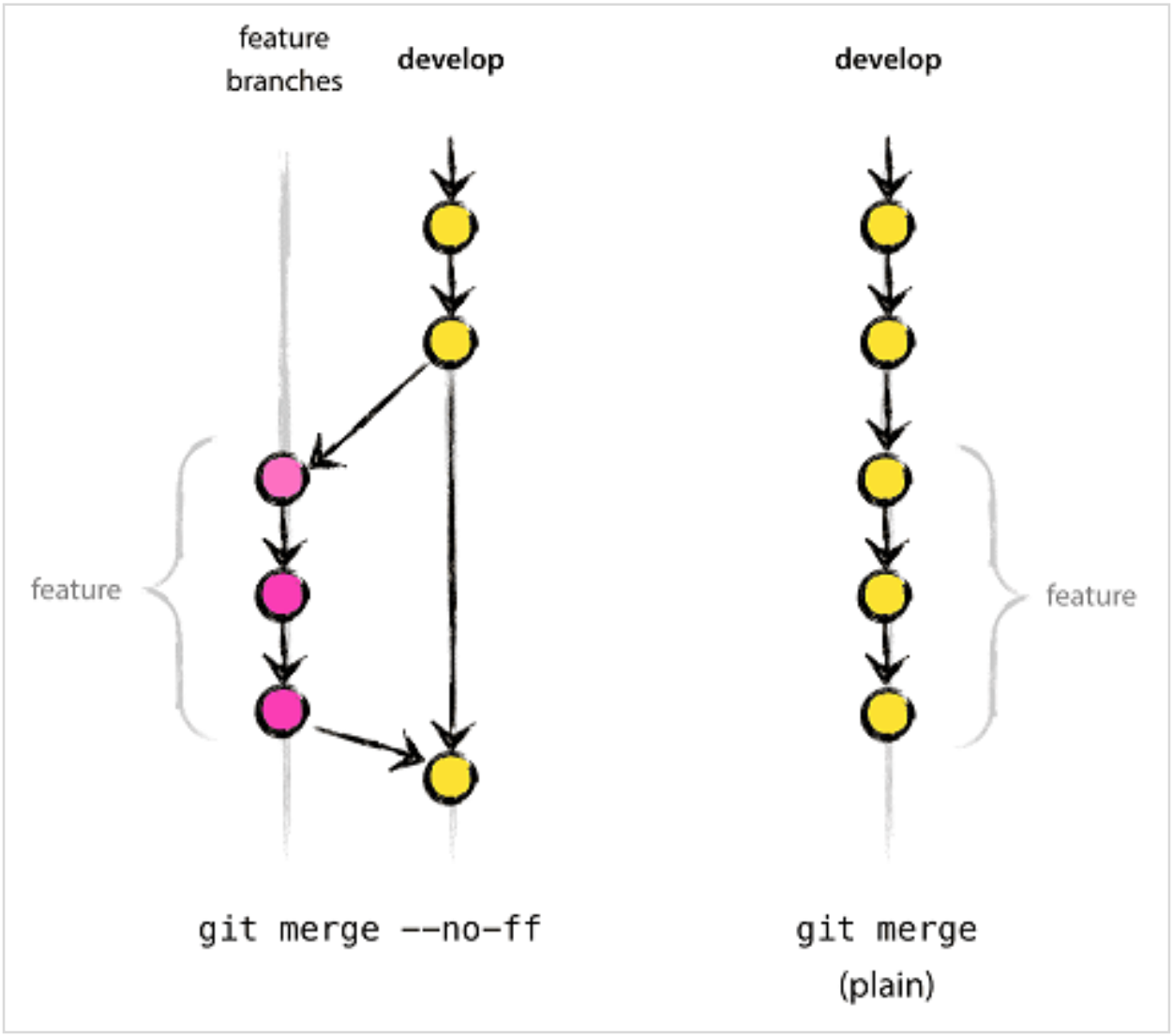
push 本地 `feature-test` 到远处代码库；

```
$ git push origin feature-test
```

(2).切换到 *develop* 合并 *feature-test*

```
$ git checkout develop $ git merge --no-ff feature-test
```

"- -no-ff" 的作用是创建一个新的 "commit" 对象用于当前合并操作。这样既可以避免丢失该功能分支的历史存在信息，又可以集中该功能分支所有历史提交。并且如果想回退版本也会比较方便。



git merge --no-ff 图示

(3).移除本地和远程仓库的 *feature-test* 分支

```
$ git branch -d feature-test $ git push origin --delete feature-test
```

2.2.Release branches（预发布分支）

应用场景：

"Release branches" 用来做新版本发布前的准备工作，在上面可以做一些小的 bug 修复、准备发布版本号等等和发布有关的小改动，其实已经是一个比较成熟的版本了。另外这样我们既可以在预发布分支上做一些发布前准备，也不会影响 "develop" 分支上下一版本的新功能开发。

应用规则：

1. 从 `develop` 分支创建，最终合并回 `develop` 和 `master`；
2. 分支名：release-*；

(1).Creat a release branch

```
$ git checkout -b release-1.1 develop \#push 到远程仓库（可选） $ git push origin release-1.1
```

do something in `release-1.1` branch

(2).切换到 *master* 合并 *release-1.1*

```
$ git checkout master $ git merge --no-ff release-1.1 $ git tag -a 1.1 $ git push origin 1.1
```

当我们的 `release-1.1` 的 Review 完成，也就预示着我们可以发布了。打上相应的版本号，再 *push* 到远程仓库。

(3).切换到 *develop* 合并 *release-1.1*

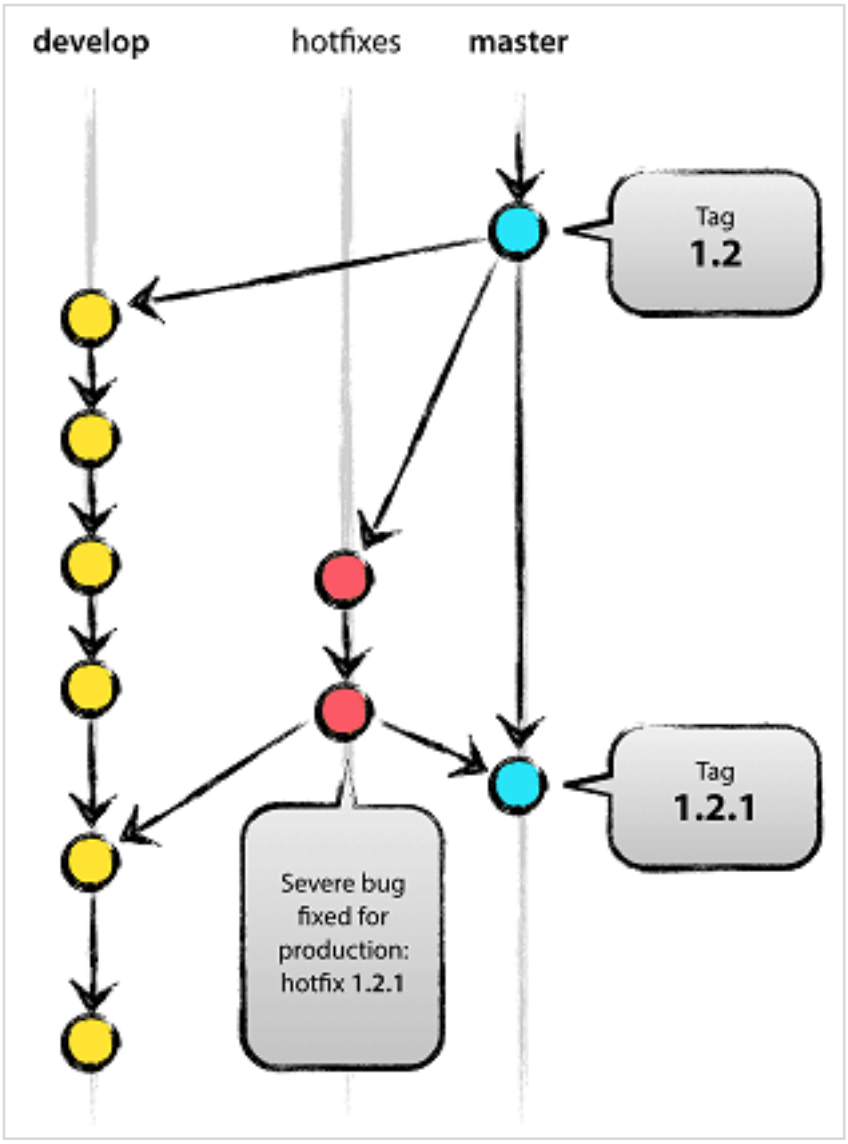
预发布分支所做的修改同时也要合并回 *develop*

```
$ git checkout develop $ git merge --no-ff release-1.1
```

(4).移除本地和远程仓库的 *release-1.1*

```
$ git branch -d release-1.1 $ git push origin --delete release-1.1
```

2.3.Hotfix branches（热修复分支）



Hotfix branches 图示

应用场景：

"Hotfix branches" 主要用于处理线上版本出现的一些需要立刻修复的 bug 情况.

应用规则：

1. 从 `master` 分支上当前版本号的 `tag` 处切出，也就是从最新的 `master` 上创建，最终合并回 `develop` 和 `master`；
2. 分支名：hotfix-*；

(1).Creat a fixbug branch

```
$ git checkout -b fixbug-1.1.1 master \#push 到远程仓库（可选） $ git push origin fixbug-1.1.1
```

do something in `fixbug-1.1.1` branch

(2).切换到 *master* 合并 *fixbug-1.1.1*

```
$ git checkout master $ git merge --no-ff fixbug-1.1.1 $ git tag -a 1.1.1 $ git push origin 1.1.1`
```

bug 修复完成，合并回 `master` 并打上版本号；

(3).切换到 *develop* 合并 *fixbug－1.1.1*

```
$ git checkout develop $ git merge --no-ff fixbug-1.1.1
```

(4).移除本地和远程仓库的 *fixbug-1.1.1*

```
$ git branch -d fixbug-1.1.1 $ git push origin --delete fixbug-1.1.1
```

> 参考内容：
> Git分支管理策略
> A successful Git branching model
> #开发规范/工作流#