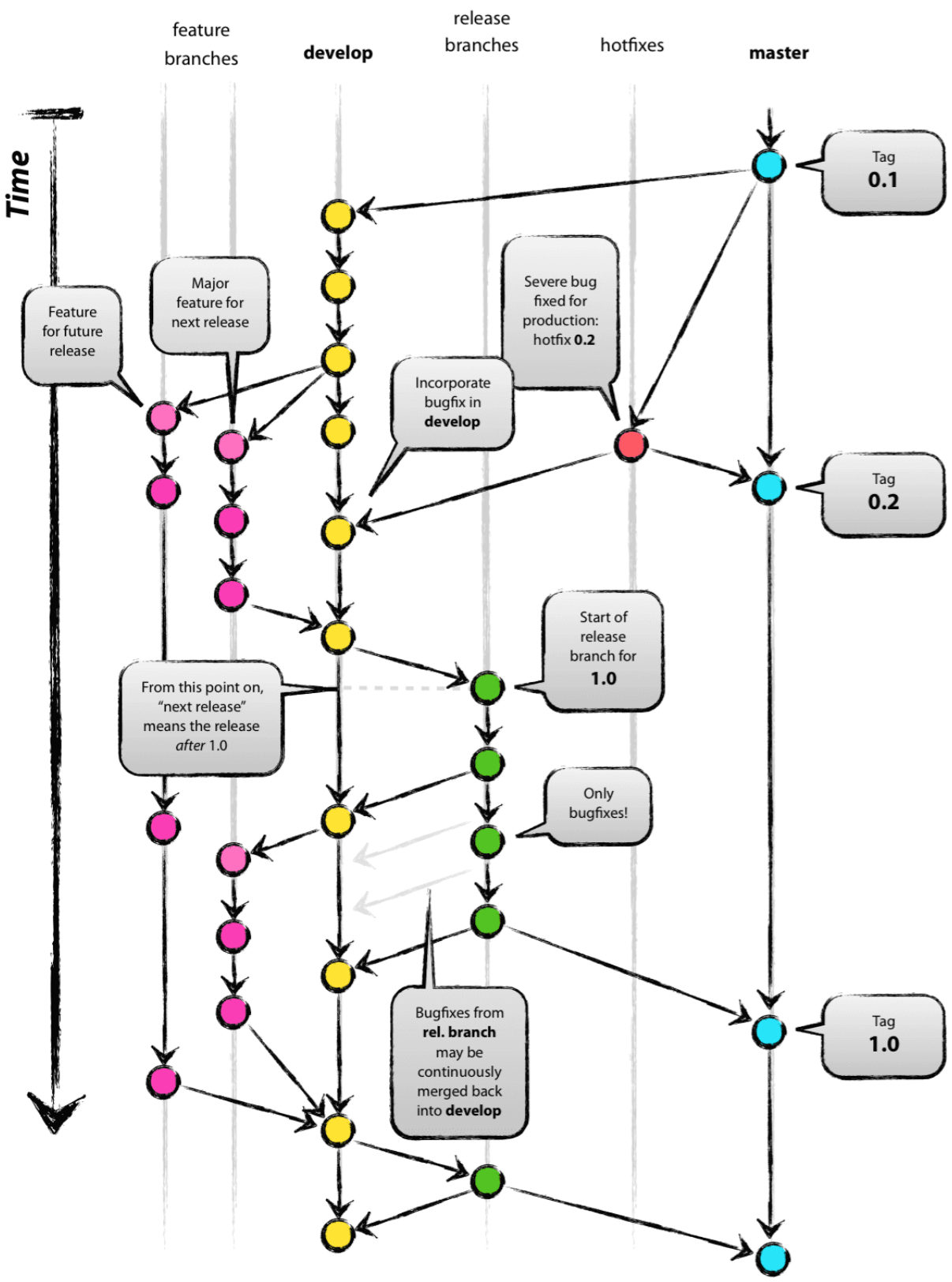
Un modèle de versionnement efficace avec Git

16 juillet 2018 Sodifrance Agilité & méthodes

Dans le contexte actuel, nous travaillons à plusieurs sur les mêmes projets qui peuvent durer plusieurs mois, et mêmes années. Pour chaque projet, nous avons une version de production, de développement et parfois une version pour des nouvelles fonctionnalités complexes qui doivent être isolées du reste du développement. Il faut donc utiliser un bon workflow. Mes recherches m'ont mené à cet article.

Git a permis d'avoir une nouvelle façon de penser pour les développeurs en ce qui concerne les branches et les fusions. Avec Git, ces actions sont simples, elles sont abordées dès le chapitre 3 du Git Book. En raison de leur nature simple et répétitive, ce n'est pas quelque chose qui doit vous effrayer. Je vais à présent vous présenter une méthode qui tire parti des points forts de Git.

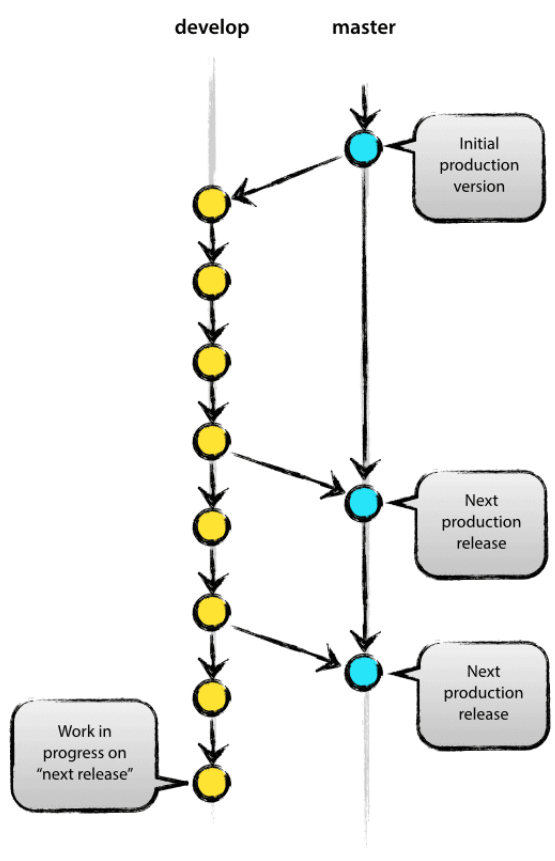
Si toute l'équipe respecte ces règles, on se retrouve avec un dépôt propre, un historique lisible et cohérent et un processus simplifié pour le déploiement en production.



Les branches principales

- master
- develop

La branche master est très familière pour un utilisateur de Git. La branche master représente l'état du code en production. Les bonnes pratiques veulent qu'aucun développeur ne fasse l'erreur de travailler dans master qui est considéré comme un reflet du code en production. Il existe une autre branche appelée develop. La branche supplémentaire develop permet de gérer les développements pour les versions à venir, c'est ce qu'on appelle souvent la branche d'intégration.




```
$ git branch develop
$ git checkout develop
```

Après ces commandes, on peut à présent contribuer au code et le commit dans la branche develop, sans que cela n'impacte notre code en production. Ceci permet d'éviter les effets de bords, et de ne pas déployer un code non validé ou testé sur la branche master.Une fois que le code source dans la branche develop

est testé et validé, tous les changements doivent être fusionnés dans la branche master avec un numéro de version.

Si vous travaillez pour une fonctionnalité estimée à 1 mois de développement, certains de vos commits ne seront pas légitimes car la fonctionnalité ne sera pas finalisée. Dans ce cas là, il faudra créer une branche spécifique.

Les branches pour les développements lourds

Contrairement aux branches master et develop, ces branches ont une durée de vie limitée car elles seront effacées au final. Ces branches permettent de travailler de manière détachée de l'équipe car elles ne seront pas appliquées sur develop et ça ne va pas polluer le code source. En général, ce type de branche reste locale à la machine du développeur. Une branche de develop se crée à partir de develop et sera fusionnée dans develop.

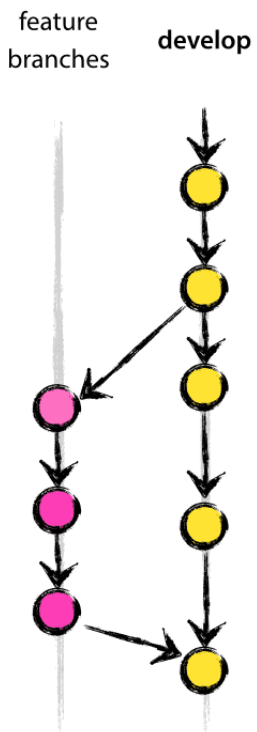
Nous pouvons y trouver

- les branches pour les fonctionnalités
- les branches pour les versions
- les branches de correctifs

Chacune de ses branches a un but bien précis et des règles. Nous allons les voir.

Les branches de fonctionnalité

Peuvent provenir de develop.
Doivent être fusionnées dans develop.
Convention de nommage tout sauf master, develop, release-* ou hotfix-*.



Comme son nom l'indique, elles sont utilisées pour développer de nouvelles fonctionnalités.Cette branche doit exister tant que la fonctionnalité est en cours de développement, et sera finalement fusionnée dans la branche develop (pour ajouter la fonctionnalité à la prochaine version) ou abandonnée (dans le cas d'une expérience décevante).

```
$ git checkout –b feature/foo develop
```

Nous avons à présent une nouvelle branche feature/foo basée sur develop. Nous pouvons coder et commiter et ensuite une fois que la fonctionnalité est terminée l'intégrer à develop.

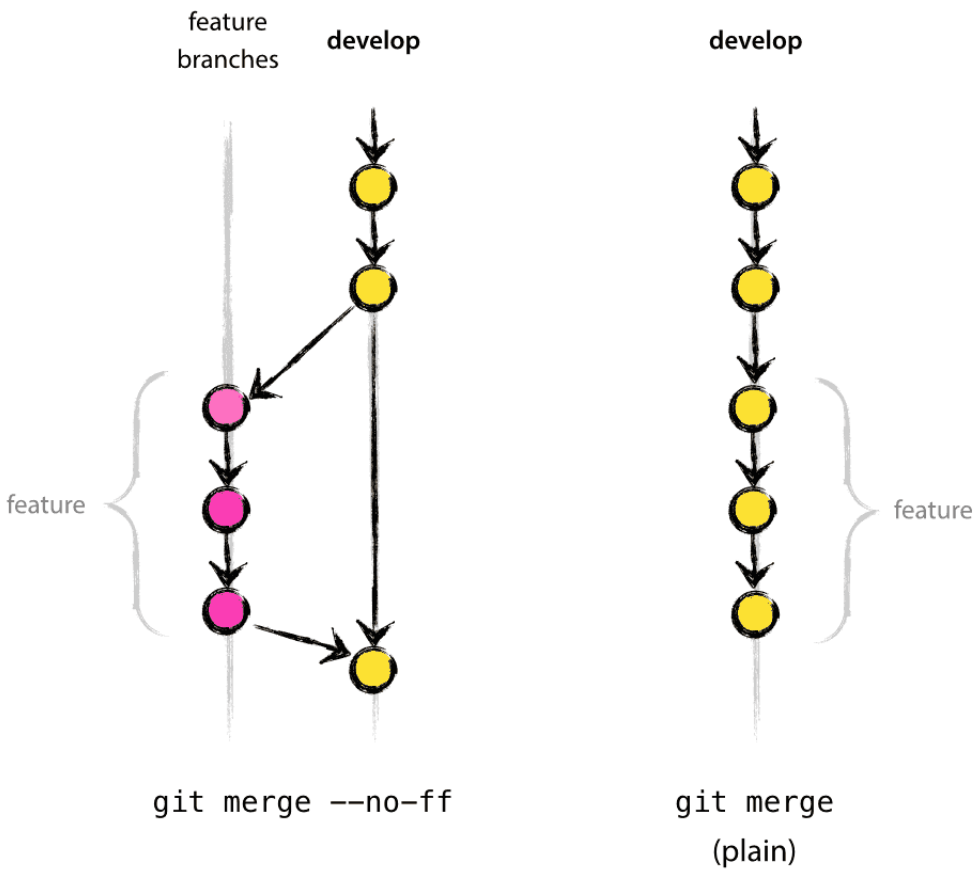
```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff feature/foo
Updating ea1b82a..05e9557
(Summary of changes)

$ git branch -d feature/foo
Deleted branch myfeature (was 05e9557).

$ git push origin develop
```

Nous sommes retourner dans la branche develop dans laquelle on demande à git de fusionner notre branche feature. L'option--no--ff permet de forcer la création d'un commit même si la fusion aurait pu se faire avec un fast forward. Ceci permet de garder une trace dans l'historique du développement dans une branche dédiée. Une fois fusionée, nous supprimons la branche de feature devenue inutile et on push les modifications sur le serveur central.



Les branches de versions

Peuvent provenir de develop.
Doivent être fusionnées dans develop et master.
Convention de nommage release-*

Comme son nom l'indique les branches de version servent à préparer les nouvelles versions de production. Elles permettent la correction d'anomalies mineures et la préparation des méta-données pour une version (numéro, date de compilation, etc.). Grâce à la branche version, la branche develop peut incorporer des fonctionnalités pour la prochaine version majeure. Nous créons la branche version depuis develop quand develop reflète l'état désiré de la nouvelle version.Toutes les fonctionnalités visées pour la version à venir doivent être fusionnées dans develop à ce moment-là. Toutes les fonctionnalités visées pour les prochaines versions ne le sont pas. C'est précisément au début d'une branche de version que la prochaine version se voit assigner un numéro de version, et pas avant. Avant ce moment, la branche develop reflète les changements pour la prochaine version mais sans forcément savoir si cette prochaine version sera la 0.3 ou la 1.0, du moins pas avant que la branche de version soit créée. Cette décision est prise au début de la nouvelle branche de version et dépend des règles mises en place sur les changements de numéros de version.

Créer une branche de version

Les branches de version sont créées à partir de la branche develop. Par exemple, disons que la version 1.1.5 est la version courante de production et que nous avons une nouvelle version majeure qui arrive. L'état de develop est prêt pour la prochaine version et nous avons décidé que cela deviendra la version 1.2 (plutôt que 1.1.6 ou 2.0). Donc nous bifurquons et donnons à la branche de version un nom qui reflète le nouveau numéro de version :

```
$  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(-)
```

Dans cette exemple, après avoir crée une nouvelle branche, et nous être positionné dessus, nous incrémentons le numéro de version. Ici, bump-version.sh est un script shell fictif qui change quelques fichiers dans notre dossier de travail pour refléter la nouvelle version. Cela peut être un changement manuel, le propos étant que certains fichiers changent. Puis, le changement de numéro de version est enregistré.

Cette nouvelle branche pourra exister pendant un moment, jusqu'à ce que la version soit déployée pour de bon. Pendant ce temps, des correctifs pourront être appliqués à cette branche (plutôt que sur la branche develop), et par conséquent en attendant la prochaine version majeure.

Une fois que la version est terminée, quelques actions doivent être effectuées. La branche est fusionnée dans master. Ensuite, ce commit sur master doit être tagué pour pouvoir faire simplement référence par la suite à cette version historique. Enfin, les changements fais sur la branche de version doivent être reversés dans develop, afin que les version futures contiennent aussi ces correctifs.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Résumé des changements)
$ git tag -a 1.2
```

La version est maintenant terminée et taguée pour toute référence future. Vous pouvez tout aussi bien vouloir utiliser les options -s or -u pour signer votre tag de manière chiffrée. Pour garder les changements effectués dans la branche de version, nous avons besoin de les fusionner dans develop.

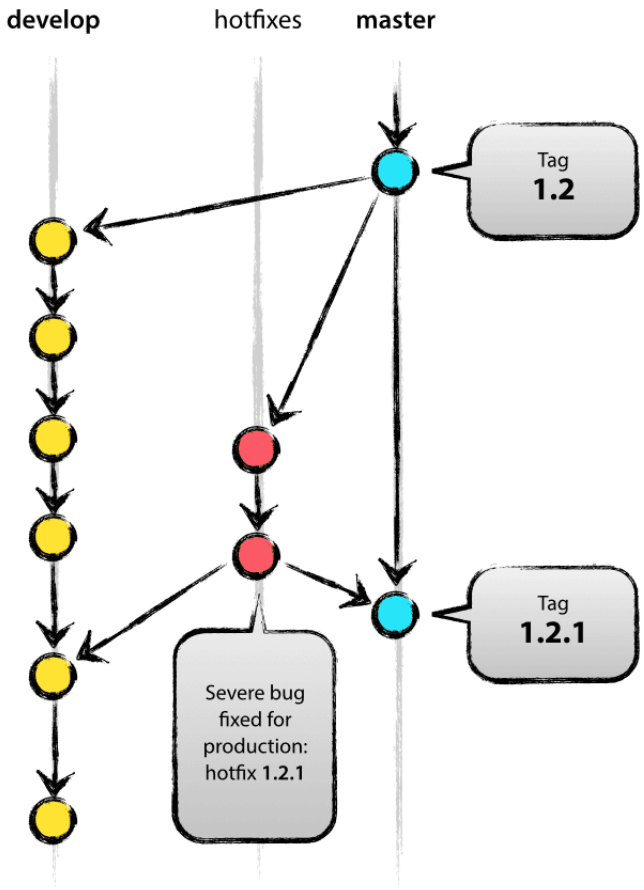
```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Résumé des changements)
```

Cette étape peut mener à une conflit de fusion (probablement même, puisque nous avons changé le numéro de version. Si c'est le cas, corrigez-le et faîtes un commit. Maintenant nous avons vraiment terminé et la branche de version peut être supprimée, puisque nous n'en avons plus besoin :

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

Les branches correctives

Peuvent provenir de master
Doivent être fusionnées dans develop et master
Convention de nommage hotfix-*.



Les branches de correctifs ressemblent aux branches de version, elles sont destinées à préparer une nouvelle version de production, bien que non planifiées. Elles viennent de la nécessité d'agir immédiatement sur un état indésirable d'une version en production. Quand une anomalie critique doit être résolu immédiatement, une branche de correctif peut être crée à partir du tag correspondant sur la branche master qui marque la version de production. L'objectif est que le travail des membres de l'équipe sur la branche develop puisse continuer, pendant qu'une autre personne prépare un correctif rapide pour la production.

Les branches de correctif sont créées à partir de la branche master. Par exemple, disons que la version 1.2 est la version qui tourne actuellement en production et que plusieurs anomalies posent problème. Dans le même temps les modifications effectuées sur develop sont encore instables. Nous pouvons alors créer une branche de correctif et commencer à corriger le problème :

```
$ 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(-)
```

N'oubliez pas de modifier le numéro de version après la création de la branche ! Ensuite, corrigez le bug et enregistrez le correctif dans un ou plusieurs commits séparés.

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

Finalisation d'une branche de correctif

Lorsque c'est terminé, le correctif a besoin d'être reporté dans master, mais aussi dans develop, de manière à garantir que le correctif sera également inclus dans la prochaine version. C'est en tout point similaire à la manière dont les branches de version sont finalisées. Premièrement, mettez à jour master et taguez la version :

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Résumé des modifications)
$ git tag -a 1.2.1
```

Vous pouvez tout aussi bien vouloir utiliser les options -s or -u pour signer votre tag de manière chiffrée. Ensuite, déployez également le correctif dans develop :

```
$  git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Résumé des modifications)
```

La seule exception à la règle ici est que, lorsqu'une branche de version en cours de développement existe, le correctif a besoin d'être reporté dans cette branche de version, plutôt que dans develop. Le report de l'anomalie dans la branche de version finira par être reporté dans develop à son tour, lorsque la branche de version sera terminée. (Si le travail en cours sur develop nécessite que l'anomalie soit corrigée immédiatement sans attendre la fin de la branche de version, vous pouvez tout aussi bien reporter dès à présent le correctif également sur develop.)

Enfin, supprimer la branche temporaire :

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Conclusion

Bien qu'il n'y ait rien de bien révolutionnaire dans ce modèle de développement. C'est un modèle facile à appréhender et il permet de développer une appropriation collective des processus de branches et de versions. J'espère avoir pu vous éclairer sur les différentes branches de git et leurs utilisations. Ce workflow devrai permettre à vos dépots de rester cohérents, de pouvoir gérer indépendamment les avancées fonctionnelles, les patchs et releases sans finir la journée avec un mal de tête.

Pour aller plus loin, vous pouvez lire cette introduction à Git-flow.