Git
Gestion des branches et des sources

Sylvain Bouveret, Grégory Mounié

2019

Ce document peut être téléchargé depuis l'adresse suivante :

http://systemes.pages.ensimag.fr/www-unix/avance/seance1-git-bonus/tp1-flow.pdf

1 Introduction

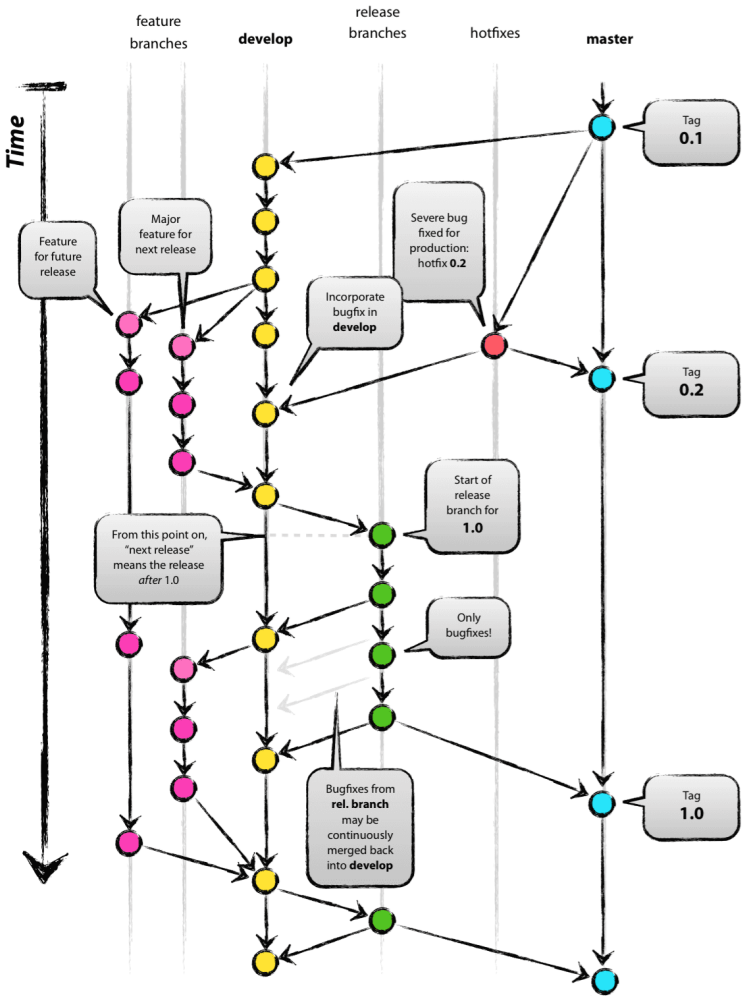

FIGURE 1 – Le modèle de branches de git-flow de Vincent Driessen http://nvie.com/posts/a-successful-git-branching-model/

Le but premier de ce TP est de vous familiariser avec l'utilisation des branches. Nous allons donc créer des branches, sauter d'une branche à l'autre et faire de nombreuses fusions (merge). L'autre point du TP sera de vous amener à manipuler plusieurs entrepôts distants simultanément. Un serveur «public»

diffusera la branche «master» tandis qu'un serveur de «développement» stockera l'ensemble des autres branches.

2 Git et les branches

2.1 Mise en place

Ce TP sera lui-aussi réalisé en équipe de deux utilisateurs (Alice et Bob dans le reste du sujet). Les explications sont écrites pour 2 utilisateurs pour simplifier, mais il peut y avoir un nombre quelconque de coéquipiers.

La première étape, sur votre PC, est de construire d'abord un dépôt minimaliste. Ce dépôt sera copié, plus tard, sur `gitlab.ensimag.fr`.

Alice crée sur sa machine le dépôt local initial.

```
alice@laptop1$ mkdir alice-et-bob
alice@laptop1$ cd alice-et-bob
alice@laptop1$ git init .
alice@laptop1$ mon_editeur_prefere fichier.txt # mettre quelques lignes
alice@laptop1$ git add fichier.txt
alice@laptop1$ git commit
```

À chaque étape, pensez à afficher le graphe de commit pour en suivre l'évolution, soit en mode texte avec :

```
git log --all --graph --oneline
```

soit en mode graphique avec :

```
gitk --all
```

Suivant les modifications que vous ferez à `fichier.txt`, il faudra peut-être aussi gérer quelques conflits.

2.2 Créer deux dépôts vides sur `gitlab.ensimag.fr`

Bob va, lui, créer deux dépôts vides sur `gitlab.ensimag.fr`.

Référez vous au TP1 pour créer le groupe de développeurs, nommé en fonction de vos login, comme par exemple `alice_bob`, si il n'existe pas déjà, et deux projets vides dans ce groupe.

Les deux projets se nommeront :

1. `projet2-public`
2. `projet2-dev`

2.3 Ajouter deux sources distantes (remote) et y pousser le code de départ

Alice ajoute deux sources distantes (remote) à son dépôt et pousse sa version initiale dans les deux dépôts.

```
alice@laptop1$ cd alice-et-bob
alice@laptop1$ git remote add origin
↪  ssh://git@gitlab.ensimag.fr/alice_bob/projet2-public.git
alice@laptop1$ git remote add dev
↪  ssh://git@gitlab.ensimag.fr/alice_bob/projet2-dev.git
alice@laptop1$ git remote -v
alice@laptop1$ git push origin master
alice@laptop1$ git push dev master
```

Maintenant, *Bob* peut cloner un des deux dépôts et ajouter l'autre source, lui aussi.

```
bob@laptop2$ git clone ssh://git@gitlab.ensimag.fr/alice_bob/projet2-public.git
bob@laptop2$ cd projet2-public
bob@laptop2$ git remote -v
bob@laptop2$ git remote add dev ssh://git@gitlab.ensimag.fr/alice_bob/projet2-dev.git
bob@laptop2$ git remote -v
```

2.4 Ajouter des tags sur la version «master»

Alice ajoute le tag `v0.0` à la version initiale de la branche «master». Elle devra écrire un message associé au tag (option -a `-annotate`). Cela permet d'associer au tag une date de création, l'identité du créateur, et éventuellement sa signature GnuPG.

```
alice@laptop1$ git tag -a v0.0
```

Mais ce tag n'existe encore que localement. *Alice* pousse le tag dans les deux dépôts. Elle peut le faire en poussant uniquement le tag, ou en poussant tous les tags.

```
alice@laptop2$ git push origin v0.0
alice@laptop2$ git push --tags dev
```

Bob récupère alors le tag de manière implicite lorsqu'il fait un git pull. Cela ne fonctionne de manière implicite que pour les tags concernant des objets présents ou tirés dans le dépôt de *Bob*.

```
bob@laptop2$ git pull
```

Bob crée un second commit en modifiant `fichier.txt` et ajoute le tag annoté `v1.0`. Il pousse sa modification et le tag dans les deux dépôts.

```
bob@laptop2$ mon_editeur_prefere fichier.txt
bob@laptop2$ git add -p fichier.txt
bob@laptop2$ git commit
bob@laptop2$ git tag -a v1.0
bob@laptop2$ git push --tags dev
bob@laptop2$ git push --tags
bob@laptop2$ git push
bob@laptop2$ git push dev
```

Le dépôt public «origin» de *Bob* est son dépôt par défaut.

Alice peut maintenant le récupérer. Pour ce faire, elle va indiquer «origin» comme dépôt par défaut de sa branche «master».

```
alice@laptop1$ git branch --set-upstream-to=origin/master master
alice@laptop1$ git pull
```

Elle a noté que l'étiquette «remote/dev/master» est toujours sur le premier commit.

2.5 Ajouter une branche «develop» dans dev

Pour faciliter la maintenance et le développement, *Alice* ajoute une branche «develop».

```
alice@laptop1$ git checkout -b develop
alice@laptop1$ git branch
* develop
  master
```

puis elle ajoute un commit sur cette branche et la pousse dans «dev». Par défaut, Git pousse la branche courante. Elle en profite pour indiquer à Git que c'est «dev» qui sera le dépôt par défaut où pousser cette branche

```
alice@laptop1$ mon_editeur_prefere fichier.txt
alice@laptop1$ git add -p
alice@laptop1$ git commit
alice@laptop1$ git push --set-upsteam dev
```

Bob récupère la branche `develop` et bascule sa position courante (HEAD) dessus.

```
bob@laptop2$ git pull dev
```

Un «pull» c'est deux opérations : un «fetch» qui récupère les nouvelles informations du dépôt, ici, la branche «develop» et son commit, et un «merge».

Git vous indique que comme la branche «master» n'est pas configurée pour suivre «dev» (elle est configurée pour suivre «origin»), il faut préciser la branche pour le «merge».

```
bob@laptop2$ git checkout develop
```

Si tout va bien, Git indique que la branche est configurée pour suivre l'évolution de la branche de même nom sur «dev».

2.6 Faire une branche de fonctionnalité et la fusionner

Bob veut implanter une nouvelle fonctionnalité. Il vérifie qu'il est bien sur la branche «develop». Il ajoute une branche («topic1»), code la fonctionnalité et la publie dans «dev».

```
bob@laptop2$ git status
On branch develop
Your branch is up to date with 'dev/develop'.
...
bob@laptop2$ git checkout -b topic1
bob@laptop2$ mon_editeur_prefere fichier.txt
bob@laptop2$ git add -p
bob@laptop2$ git commit
bob@laptop2$ git push --set-upstream dev topic1
```

Alice veut implanter elle-aussi implanter une nouvelle fonctionnalité. Elle le fait dans la branche «topic2» qui part de «develop». Elle enregistre elle-aussi son commit et le publie dans «dev».

```
alice@laptop1$ git status
On branch develop
Your branch is up to date with 'dev/develop'.
...
alice@laptop1$ git checkout -b topic2
alice@laptop1$ mon_editeur_prefere fichier.txt
alice@laptop1$ git add -p
alice@laptop1$ git commit
alice@laptop1$ git push --set-upstream dev topic2
```

Après l'avoir testé rigoureusement *Bob* veut fusionner «topic1» dans «develop».

```
bob@laptop2$ git checkout develop
bob@laptop2$ git merge topic1
bob@laptop2$ git push
bob@laptop2$ gitk --all
```

Regardez attentivement le graphe de commit. Git n'a pas créé de commit pour ce merge. Il a juste déplacé la tête. Le terme Git désignant ce type de merge sans commit est «fast-forwarding».

Alice a une idée pour améliorer cette contribution. Elle récupère le travail de *Bob* (en utilisant juste le fetch), ajoute un commit dans la branche «topic1» mais elle veut garder une trace de la fusion. Elle utilise l'option `-no-ff` lors du merge. Elle devra aussi mettre à jour sa branche «develop» avant.

```
alice@laptop1$ git fetch dev
alice@laptop1$ git checkout topic1
alice@laptop1$ mon_editeur_prefere fichier.txt
alice@laptop1$ git add -p
alice@laptop1$ git commit
alice@laptop1$ git checkout develop
alice@laptop1$ git pull
alice@laptop1$ git merge --no-ff topic1
alice@laptop1$ git push
alice@laptop1$ gitk --all
```

Cette fois-ci, Git a demandé un message de merge et a créé un commit supplémentaire.

La forme finale du graphe de commits dépend de l'usage, ou non, de l'option `-no-ff`. Dans le reste du sujet, *Alice* et *Bob* utiliserons cette option pour tracer les dates, et les responsables des «merge», dans les branches «master» et «develop».

2.7 Repositionner topic2 avec rebase

Bob reprend le travail *Alice* sur le «topic2». Il rebase la branche sur la nouvelle version de «develop». Il aura peut-être à gérer des conflits. Il faudra aussi faire un pull (pour le merge) après le rebase avant de pouvoir faire le push.

```
bob@laptop2$ git fetch dev
bob@laptop2$ git checkout topic2
bob@laptop2$ git rebase develop
bob@laptop2$ git pull
bob@laptop2$ git push
bob@laptop2$ gitk --all
```

2.8 La cachette

Alice récupère les modifications de «topic2» et continue les modifications de `fichier.txt`. Elle note alors un horrible bug et ce bug existe aussi dans la version «master».

Comme elle est au milieu de sa modification, elle ne veut pas faire un commit qui ne compilerait pas. Elle ne veut pas non plus perdre son travail.

Elle va donc sauvegarder le travail en cours dans la cachette (stash), pour repartir du dernier commit et elle le reprendra après la correction du bug.

```
alice@laptop1$ git checkout topic2
alice@laptop1$ git pull
alice@laptop1$ mon_editeur_prefere fichier.txt # modification incomplète
alice@laptop1$ git status
alice@laptop1$ git stash
alice@laptop1$ git status
alice@laptop1$ git stash list
```

2.9 Faire une correction de bug

Pour faire faire la correction *Alice* reprend le travail depuis la version v1.0 de «master». Elle crée alors une nouvelle branche, «hotfix1», pour y écrire la correction.

```
alice@laptop1$ git checkout v1.0 # attention tête détachée !
alice@laptop1$ git checkout -b hotfix1
```

Elle modifie `fichier.txt` et crée un commit dans «hotfix1».

```
alice@laptop1$ mon_editeur_prefere fichier.txt
alice@laptop1$ git add -p
alice@laptop1$ git commit
alice@laptop1$ gitk --all
```

Elle fusionne ensuite ce commit avec «master» et «develop», en prenant soin de bien créer un nouveau commit dans les deux cas. Le nouveau commit de «master» sera tagué `v1.1` et publié sur «origin». Elle pourrait avoir à gérer quelques conflits.

```
alice@laptop1$ git checkout master
alice@laptop1$ git merge --no-ff hotfix1
alice@laptop1$ git tag -a v1.1
alice@laptop1$ git push --tags
alice@laptop1$ git push
alice@laptop1$ git checkout develop
alice@laptop1$ git merge --no-ff hotfix1
alice@laptop1$ git push
alice@laptop1$ gitk --all
```

2.10 Sortir de la cachette

Alice retourne dans la branche «topic2». Elle repositionne la branche par rapport à «develop» et réinsère les changements sur lesquels elle travaillait. Elle aura peut-être quelques conflits à résoudre. Il faudra aussi fusionner la version distante et locale de «topic2».

```
alice@laptop1$ git checkout topic2
alice@laptop1$ git rebase develop
alice@laptop1$ git pull
alice@laptop1$ git stash list
alice@laptop1$ git stash pop
```

Ensuite *Alice* finit ses modifications, enregistre un commit et fusionne la branche dans «develop» en prenant soin de bien créer un nouveau commit.

```
alice@laptop1$ mon_editeur_prefere fichier.txt
alice@laptop1$ git add -p
alice@laptop1$ git commit
alice@laptop1$ git checkout develop
alice@laptop1$ git merge --no-ff topic2
alice@laptop1$ git push
alice@laptop1$ gitk --all
```

Alice remarque que la branche distante de «topic2» est en retard. Pour pousser l'ensemble des distantes branches en retard (comme «topic2») *Alice* effectue

```
alice@laptop1$ git push --all
```

2.11 Nouvelle version majeure

Bob va publier la nouvelle version. Il récupère les modifications et crée une nouvelle branche `release2` partant de `develop`. Dans cette branche il réalise un nouveau commit corrigeant quelques problèmes. Il fusionne ensuite cette branche dans «master» et «develop» en créant un nouveau commit dans les deux cas. La nouvelle version du «master» est publiée avec le tag v2.0.

```
bob@laptop2$ git fetch dev
bob@laptop2$ git checkout develop
bob@laptop2$ git pull
bob@laptop2$ git checkout -b release2
bob@laptop2$ mon_editeur_prefere fichier.txt
bob@laptop2$ git add -p
bob@laptop2$ git commit
bob@laptop2$ git checkout master
bob@laptop2$ git pull
bob@laptop2$ git merge --no-ff release2
bob@laptop2$ git tag -a v2.0
bob@laptop2$ git push --tags
bob@laptop2$ git push
bob@laptop2$ git checkout develop
bob@laptop2$ git merge --no-ff release2
bob@laptop2$ git push
bob@laptop2$ gitk --all
```

3 En conclusion

Le but de cette session était de vous montrer que les branches sont très faciles d'emploi et qu'il n'y a aucune raison de ne pas en abuser car la fusion n'est pas un problème. L'utilisation de deux dépôts différents simultanément n'est pas très compliquée non plus.

Il existe encore de nombreuses autres commandes liées à cette gestion des branches, notamment "cherry-pick" et l'effacement de tags ou de branches à distance. Mais maintenant vous devriez avoir toutes les clefs pour les comprendre.

Pour aller plus loin, ou revenir sur certains concepts :

— https://git-scm.com/book/ Pro Git, un livre couvrant les concepts et les usages courants

— http://git-scm.com contient une très riche documentation : vidéos, références, des tutoriels, des livres libres (Pro Git, Git Magic, Git Internals) ou pas.