



Ce qu'il faut savoir !

IN'TECH INFO - 2016

Que savez-vous sur le  
« *version control* » ?

• • •

# Au programme

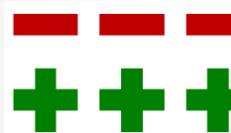
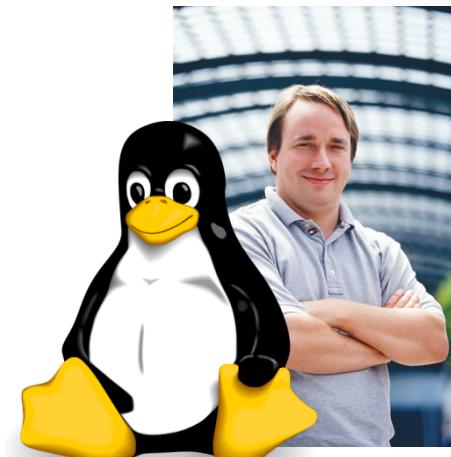
- I. Historique
- II. Les principes de base
- III. Commits
- IV. Les branches
- V. Opérations sur les branches
- VI. Les remotes
- VII. Workflow de fonctionnement
- VIII. Un outil : Git Extensions
- IX. Github
- X. Ressources



# Historique



BITKEEPER



2005



2013

1.8.2

# Les principes de base

Système de gestion de contenu ... distribué !

Fortes garanties contre la corruption !  
(accidentelles ou malveillantes)

Fonctionnement interne très simple  
« *The stupid content tracker* »

90% des opérations sont quasi instantanées  
« *Git will make you think that the gods of speed have blessed Git with unworldly powers* »

# Systèmes distribués

## Les systèmes de gestion de versions

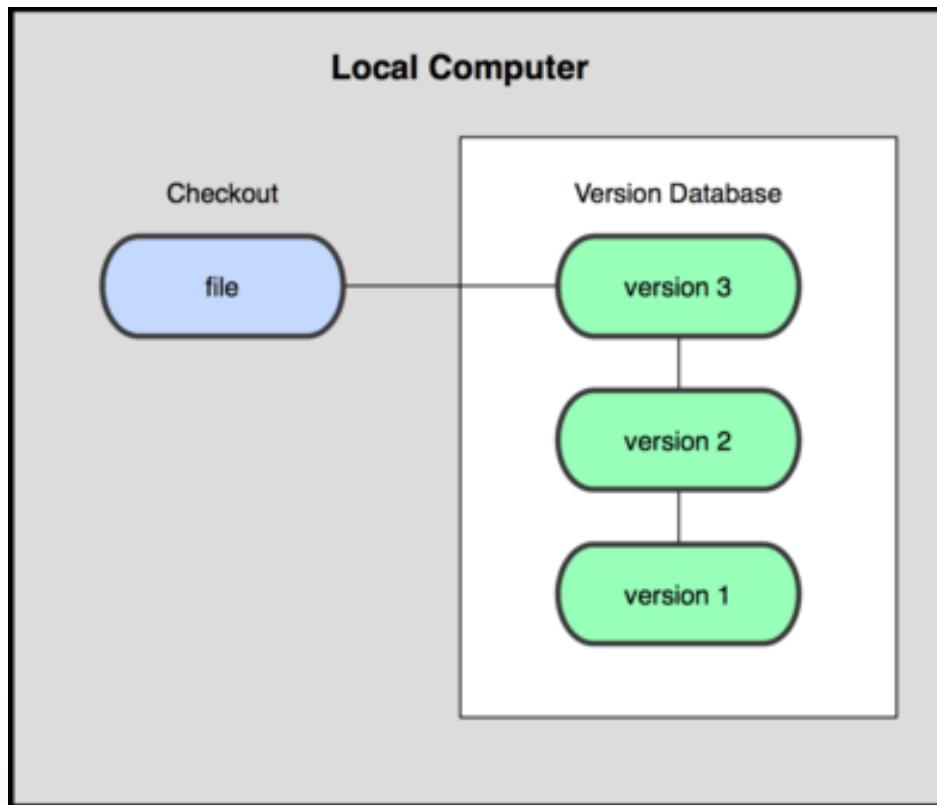
Local Version Control Systems

Centralized Version Control Systems

Distributed Version Control Systems

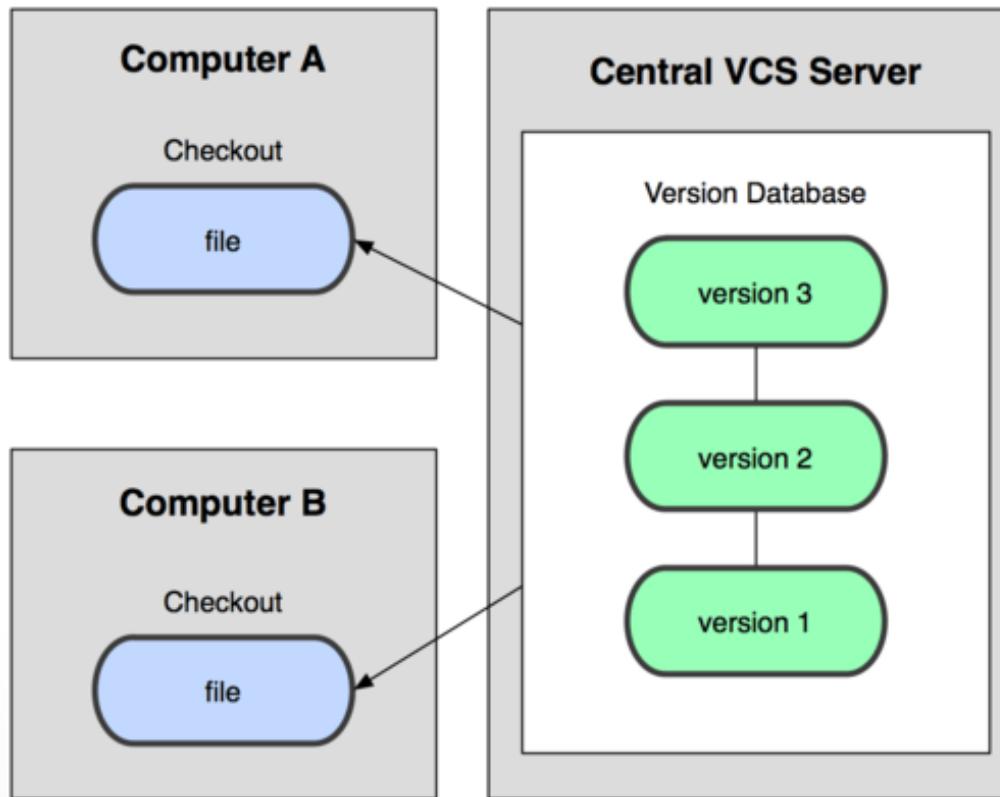
# Systèmes distribués

## Local Version Control Systems



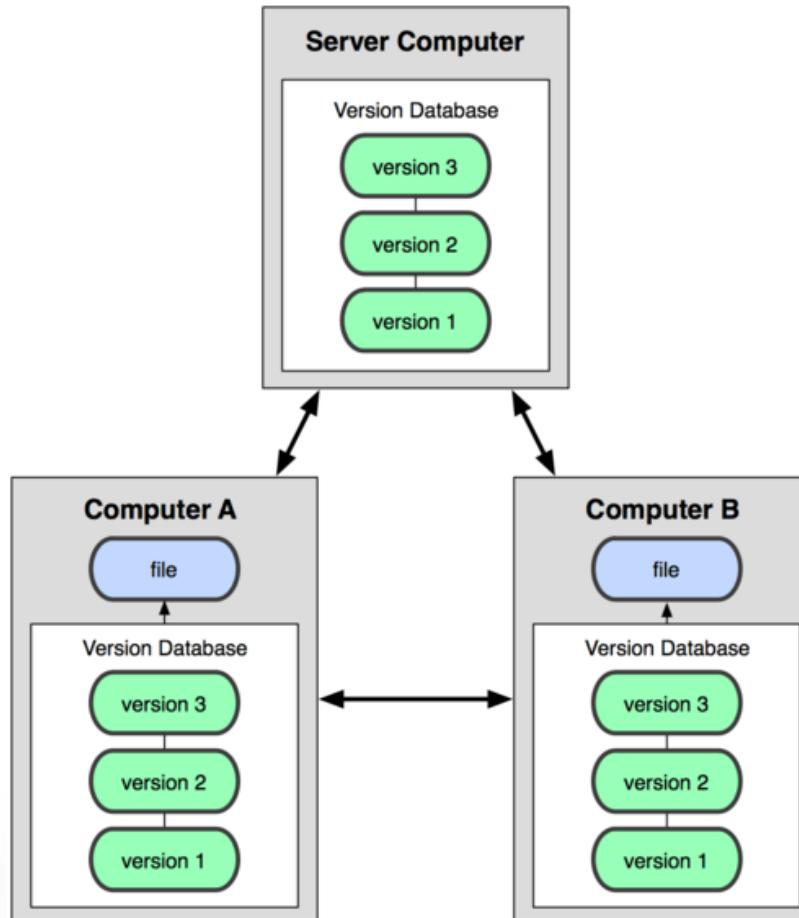
# Systèmes distribués

## Centralized Version Control Systems



# Systèmes distribués

## Distributed Version Control Systems



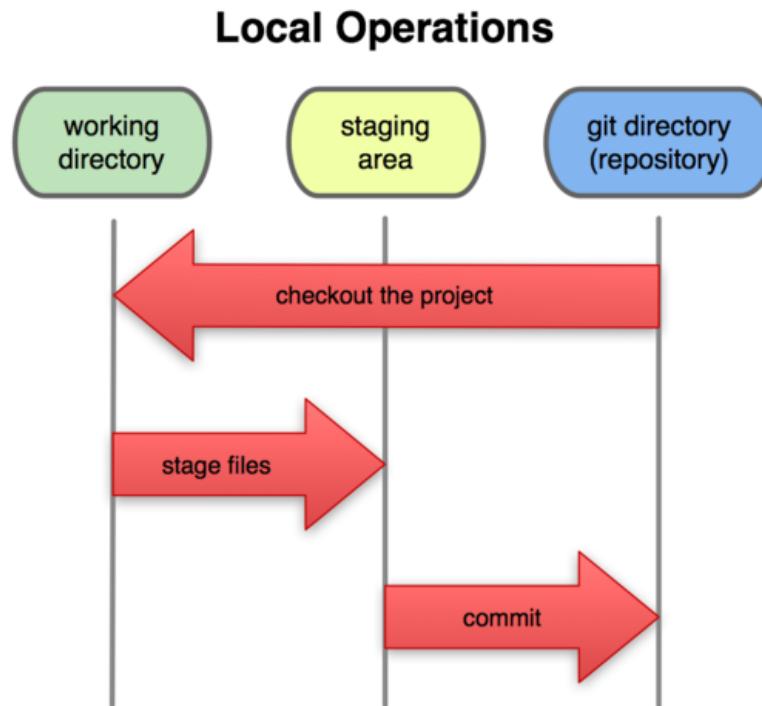
# Systèmes distribués

- Travailler en « hors-ligne »
- Collaboration !
  - Les commits peuvent ne pas gêner les autres développeurs (ou pas!)
  - Vous pouvez avoir confiance dans les autres développeurs (ou pas!)
  - Vous pouvez avoir confiance en votre hébergement (ou pas!)
- La gestion des releases
  - Gestion des cycles concurrents sont simplifiés

# Processus de commit

• • •

# Index ou *Staging area*



# Index ou *Staging area*

- Git gère trois états de fichier :
  - **Committed** : Fichier géré par Git, versionné, n'ayant subit aucun changement.
  - **Modified** : « *Committed* », mais ayant subit des modifications.
  - **Staged** : fichier « *modified* » marqué pour le prochain commit, dans l'état dans lequel il a été « *staged* ».
- Pourquoi cette étape supplémentaire ?
  - C'est au moment de « *stage* » les modifications que les objets sont créés.
  - Cela permet de choisir facilement ce qu'on veut inclure dans le commit.
  - Cela permet aussi, dans des opérations de « *reset* » d'avoir une étape intermédiaire au commit ... pour ne pas surcharger l'historique et garder un repository propre.

# Zoom sur un commit

- Elément central dans Git
- L'auteur doit avoir renseigné son nom, prénom et email (configurable pour la machine, ou uniquement pour un repository particulier).
- Doit être **clair, compréhensible et utile !**
- Doit toujours contenir **un commentaire**, et de préférence dans le format suivant :

[Première ligne = Résumé du commentaire, comme l'objet d'un mail]

[Saut de ligne]

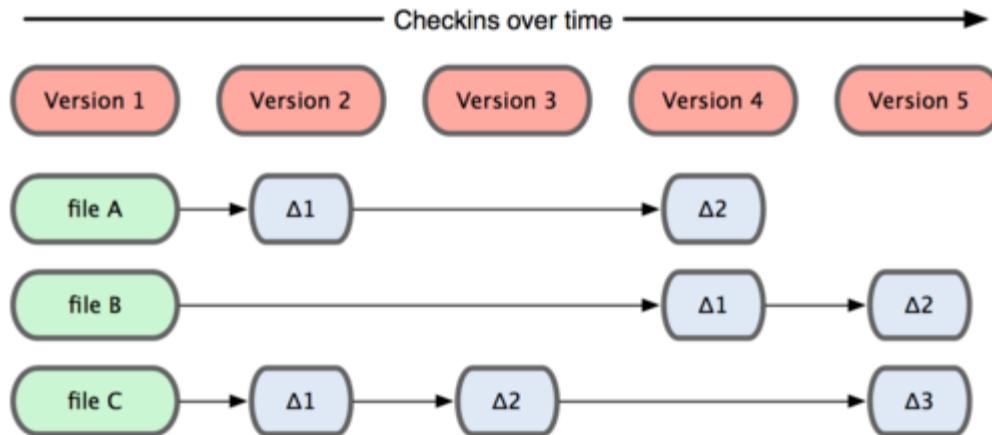
[Contenu détaillé du commit, avec des sauts de lignes pour raconter tout ce qu'il faut savoir sur les modifications que vous avez apportées au repository ...]

# Plusieurs commits



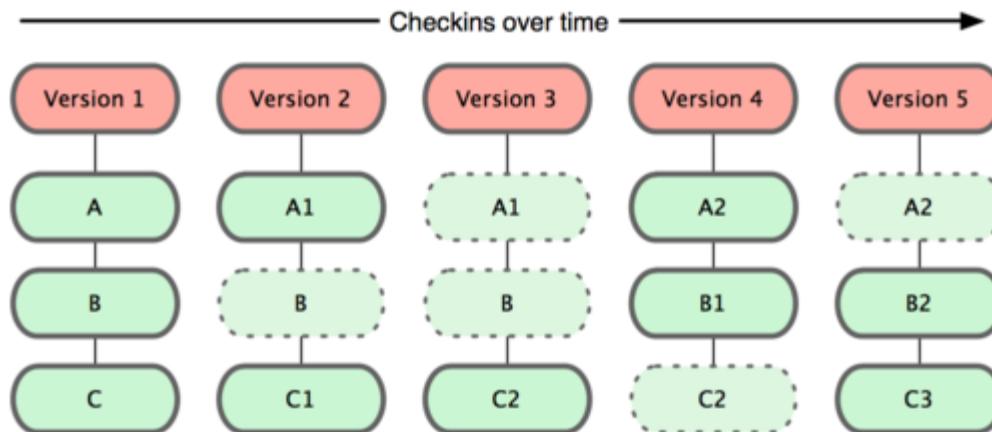
# Snapshots vs différences

Subversion (...) = différences



# Snapshots vs différences

Git (...) = snapshots



# Snapshots vs différences

- Git fait des snapshots complets de chaque donnée pour chaque version
- Git ne fait qu'ajouter des données (ou presque)
- Git gère l'intégrité des données
- Git fait toutes les opérations en local

# Stockage de données

Maintenant il y a un souci avec la quantité de données !

Il y a trop de donnée !

2 versions d'un fichier de 4Ko  
prendraient normalement 8Ko

Les fichiers sont comparés/compressés et stockés  
dans des « packfiles »

2 versions presque identiques de 4Ko  
Prendrons environs 4Ko

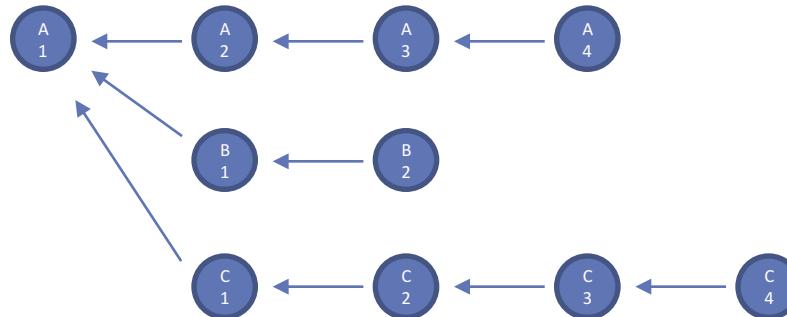
# Qu'est-ce qu'une ref ?

- Un sha1 (ou même juste le début d'un sha1)
- Un fichier nommé dans lequel est stocké un sha1
- Un *refspec*
- ...

Pas grand-chose finalement !

# Plusieurs commits

# Plusieurs fonctionnalités

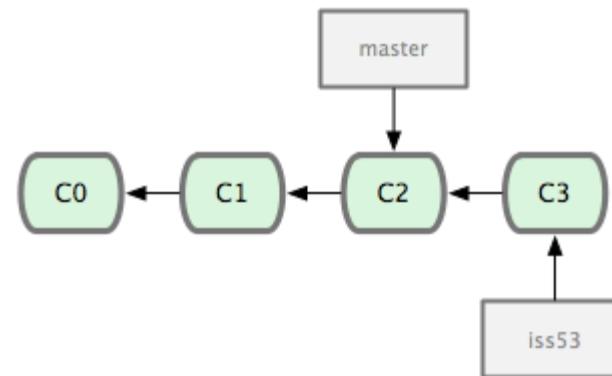


# Les branches

- Une branche = un fichier nommé avec le nom de la branche, contenant un sha1 d'un commit.
- Ce fichier est mis à jour à chaque commit dans la branche, toujours pour pointer le dernier commit. L'historique est remonté en parcourant le graph des commits.
- Une branche = un sha1 = un ref ... donc vous pouvez utiliser les noms des branches pour faire référence à des commits.

Une branche dans Git = 40 octets ! Abusez en !

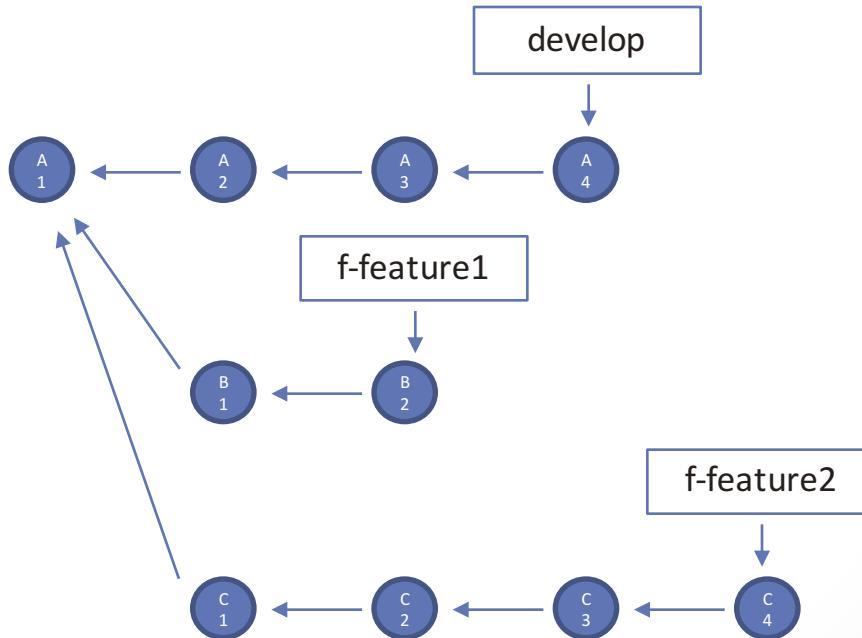
# Les branches



# Les tags

- Un tag = une branche qui ne bouge pas ...
- Un tag peut aussi être annoté : un tag qui pointe sur un objet tag. Cet objet tag pointe sur le commit taggé à l'origine. Il permet de stocker des commentaires.

# Plusieurs branches



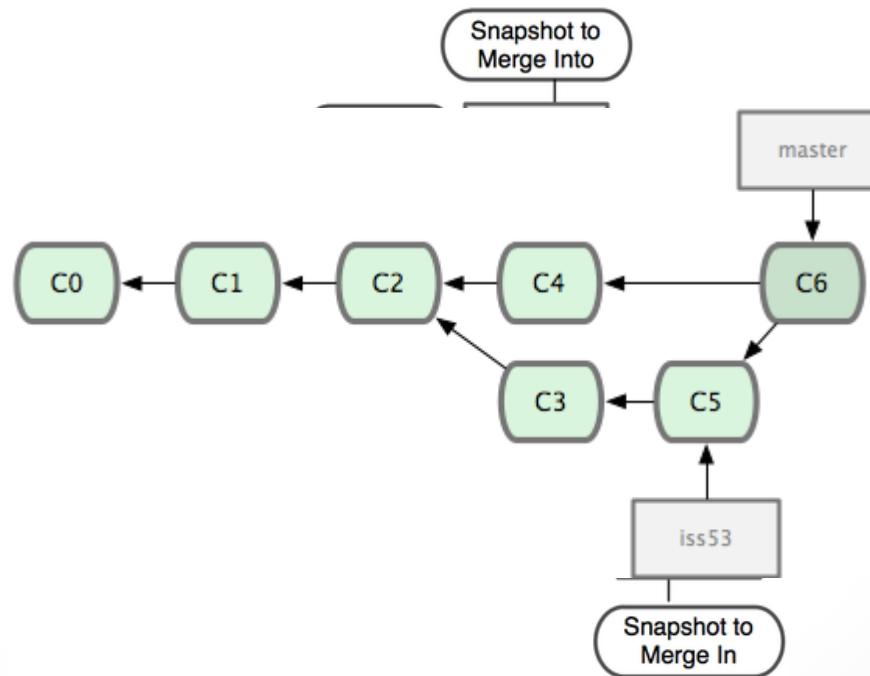
# Intégrations de branches

• • •

Les opérations de fusion : **merge** et **rebase**

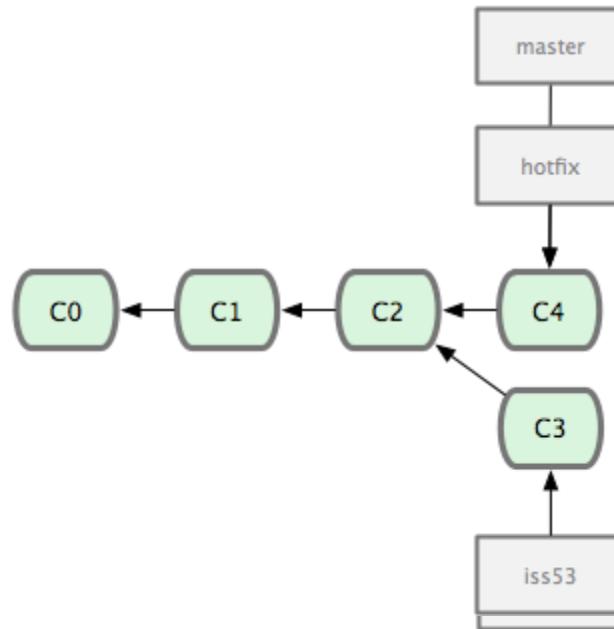
# git merge

## Opération de merge

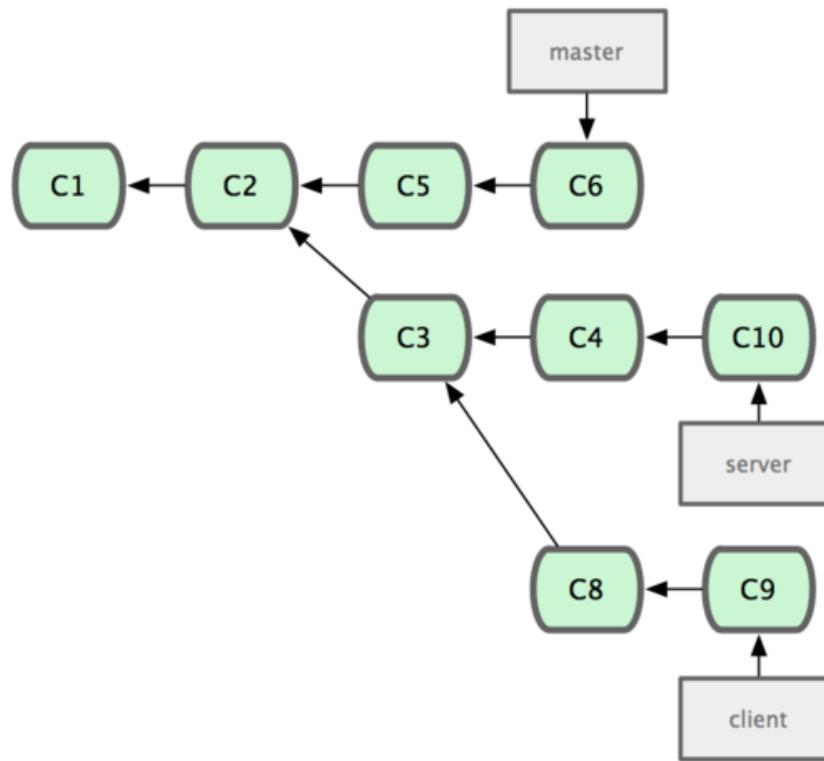


# git merge

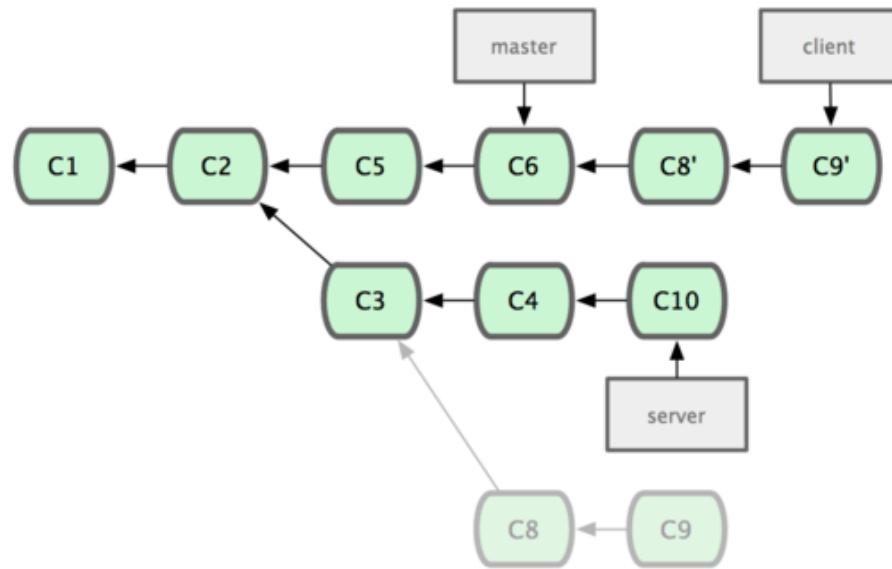
Opération de merge, avec optimisation : **fast forward**



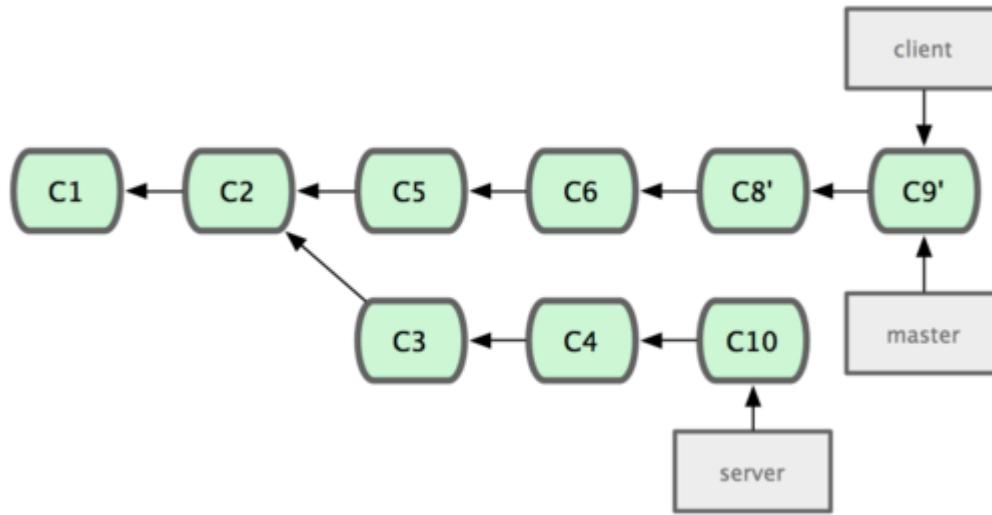
# git rebase



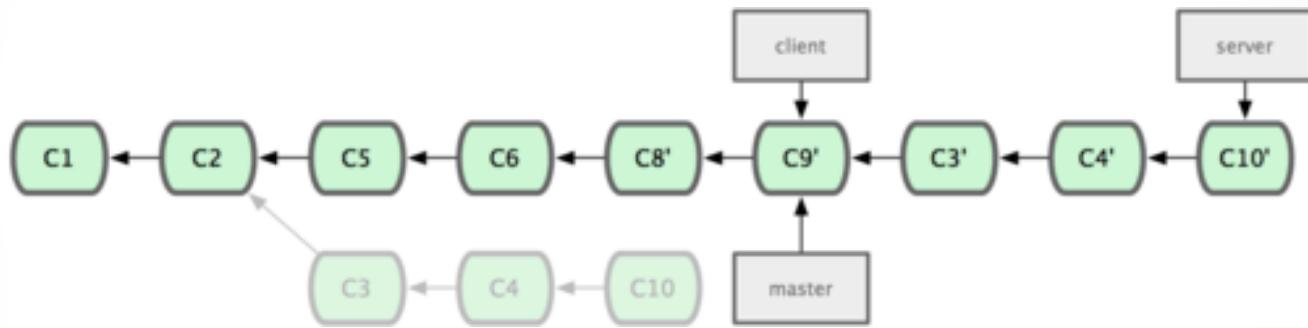
# git rebase



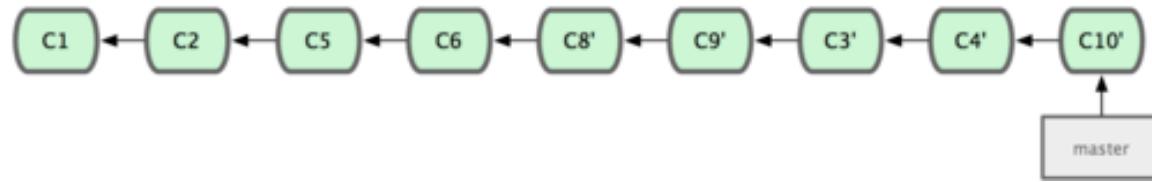
# git rebase



# git rebase



# git rebase

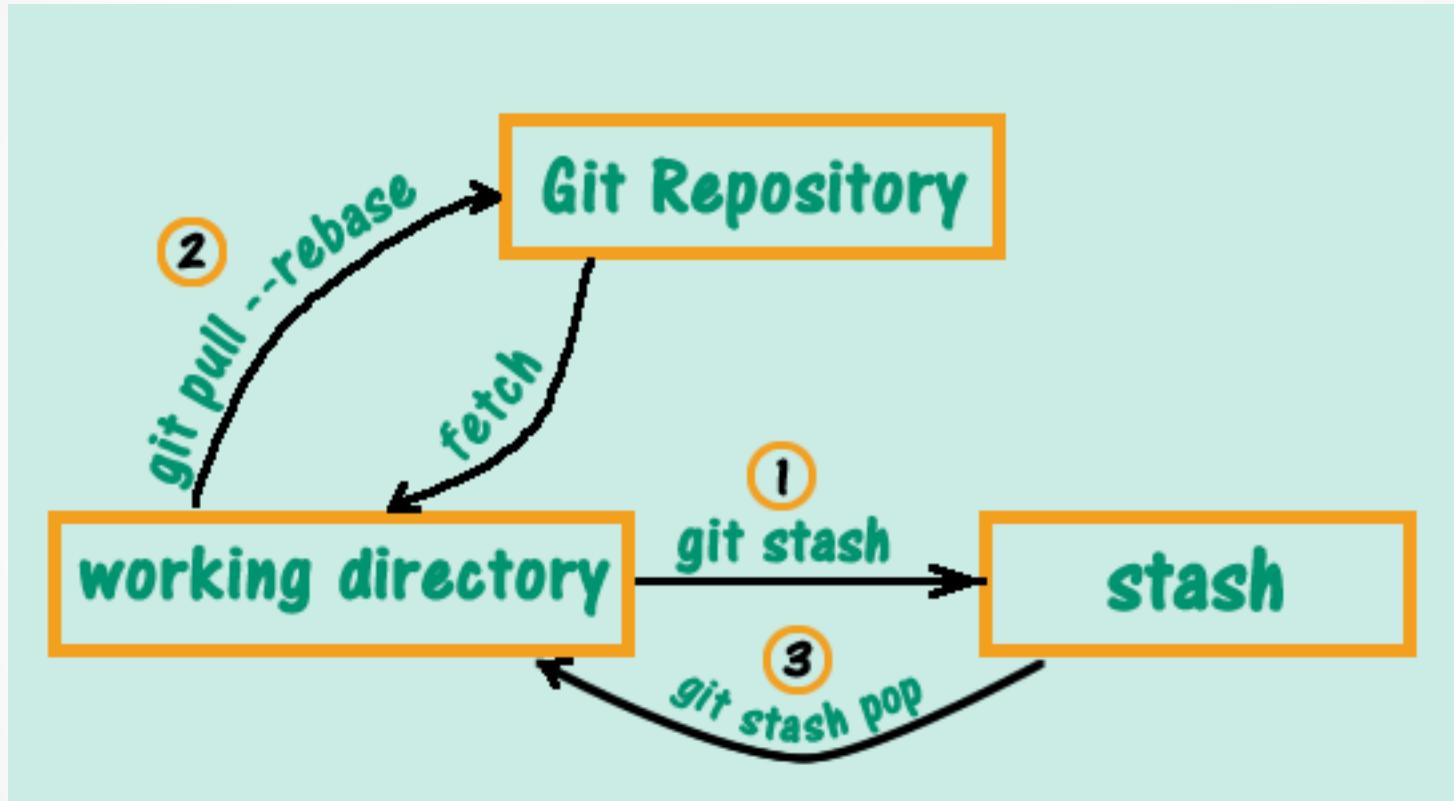


# git rebase

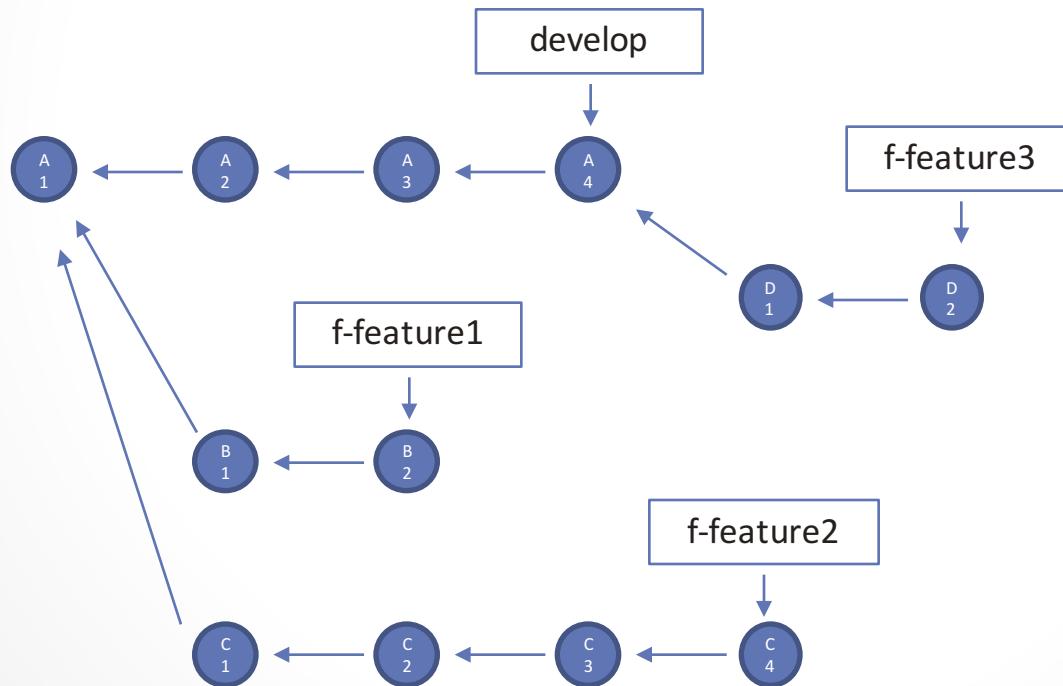


« Si vous considérez le fait de rebaser comme un moyen de nettoyer et réarranger des *commits* avant de les pousser et si vous vous en tenez à ne rebaser que des *commits* qui n'ont jamais été publiés, tout ira bien. Si vous tentez de rebaser des *commits* déjà publiés sur lesquels les gens ont déjà basé leur travail, vous allez au devant de gros problèmes énervants. »

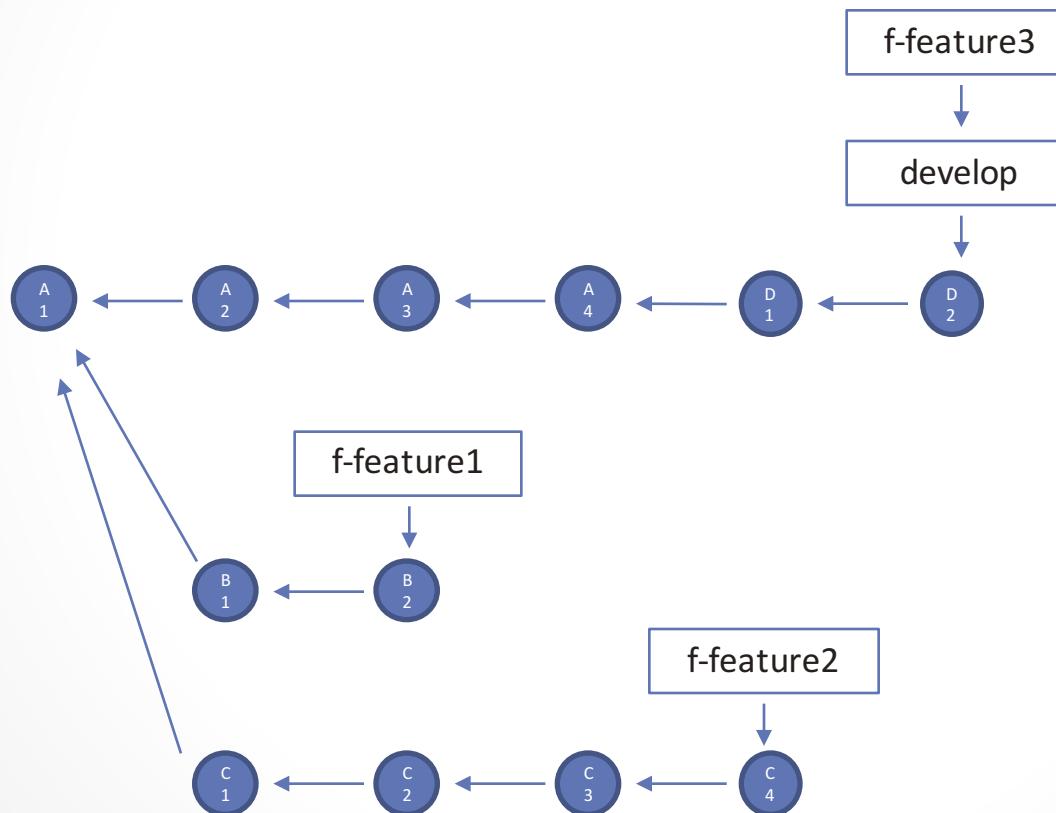
# stash



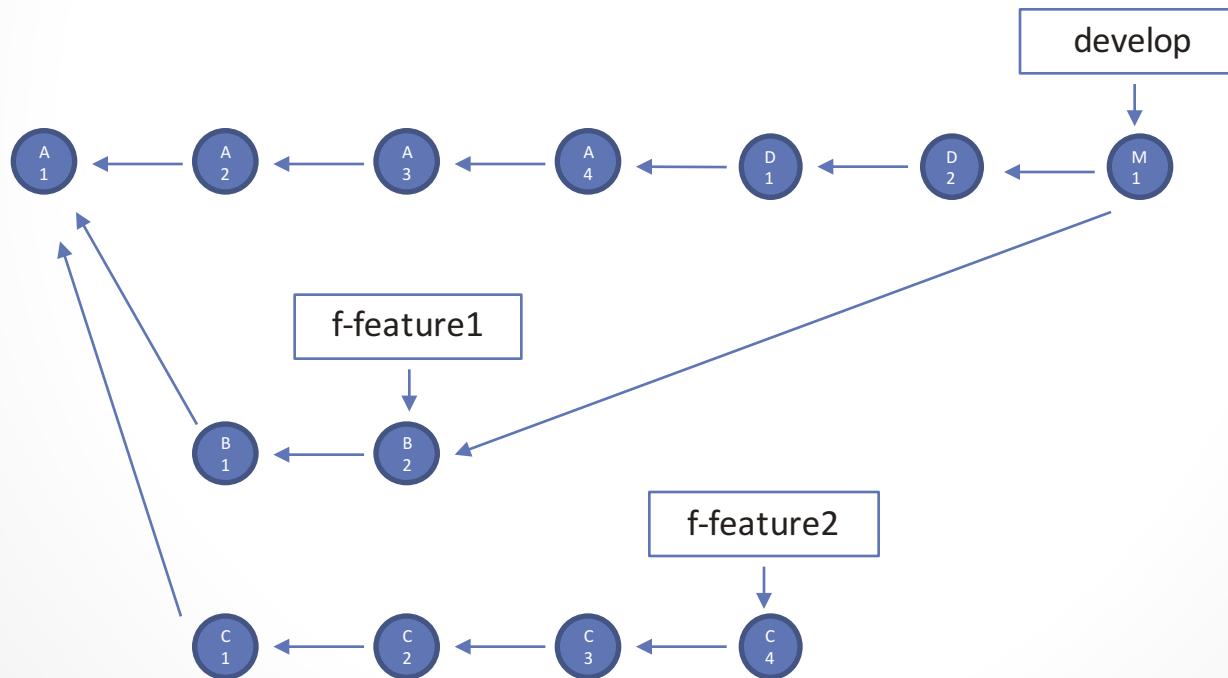
# Plusieurs branches



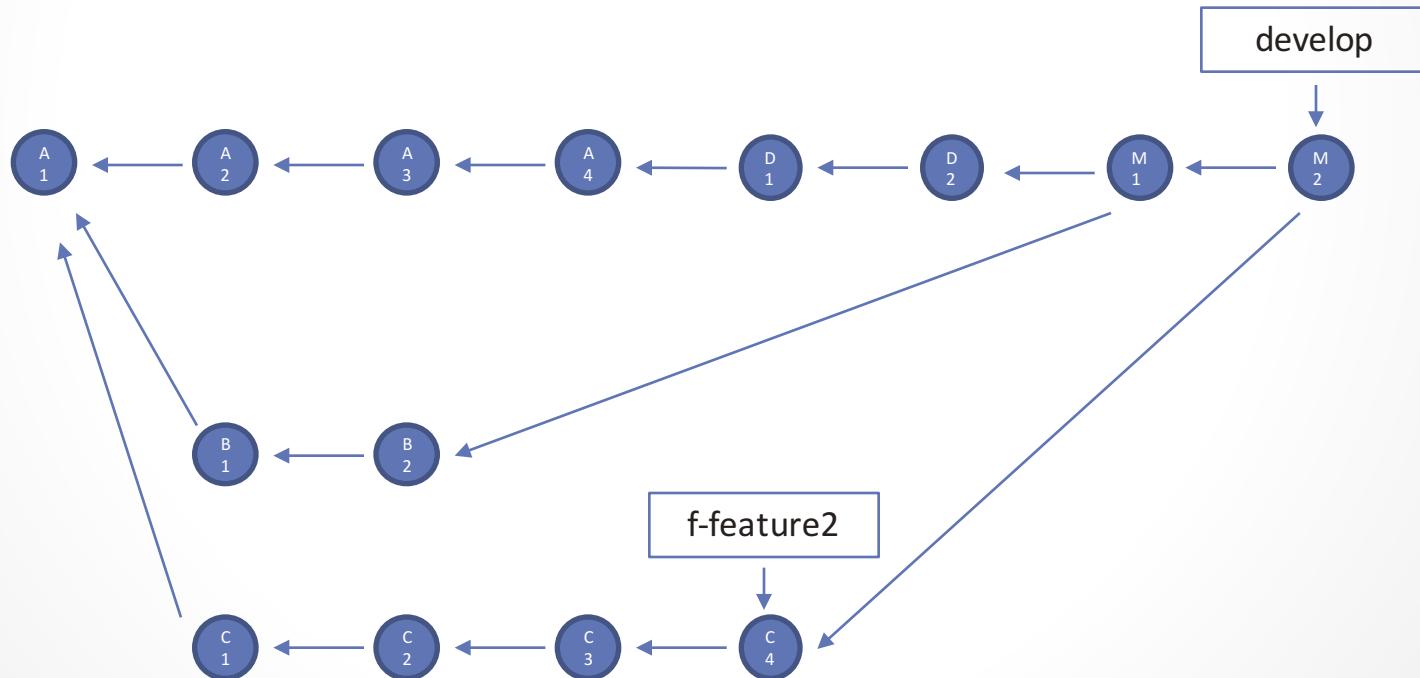
# Plusieurs branches



# Plusieurs branches

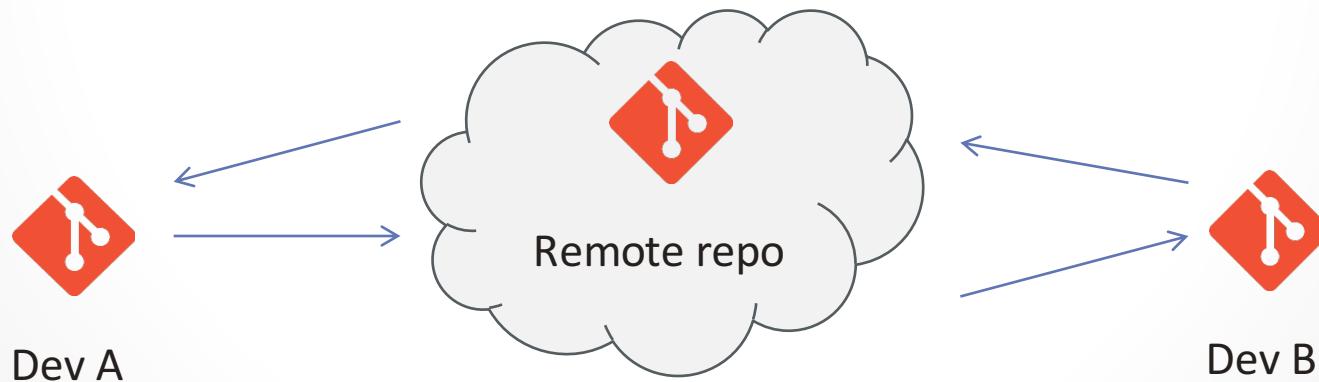


# Plusieurs branches



# Les remotes

- Collaborer ?
- Communiquer entre deux repository ?
- Bref ... centraliser un minimum une équipe ?
- Le remote est appelé par défaut « origin »



# Surtout !



≠



git

# Exemples de services



GitLab

 Visual Studio Online



# Workflows

• • •

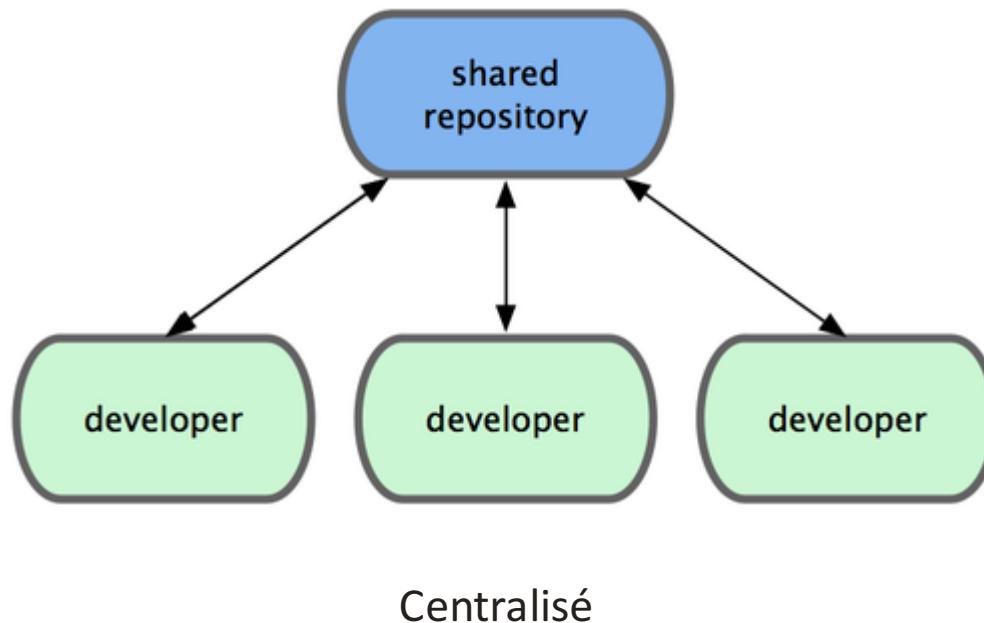
Git dans la vraie vie

# Les différents workflows

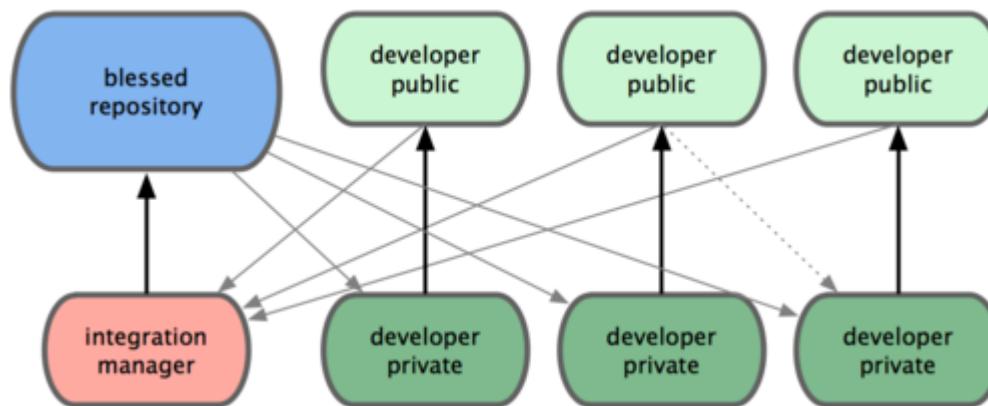
- Macro : Organisation d'équipe
  - Comment les repository échangent, qui a les responsabilités de quoi
- Micro : Organisation personnelle
  - Comment un développeur organise son repository local, et comment travail-t-il avec

Choix important, souvent à faire du début d'un projet et qui ne change pas.  
Equivalent au choix d'un framework, ou d'une méthodologie de projet !

# Workflows macro

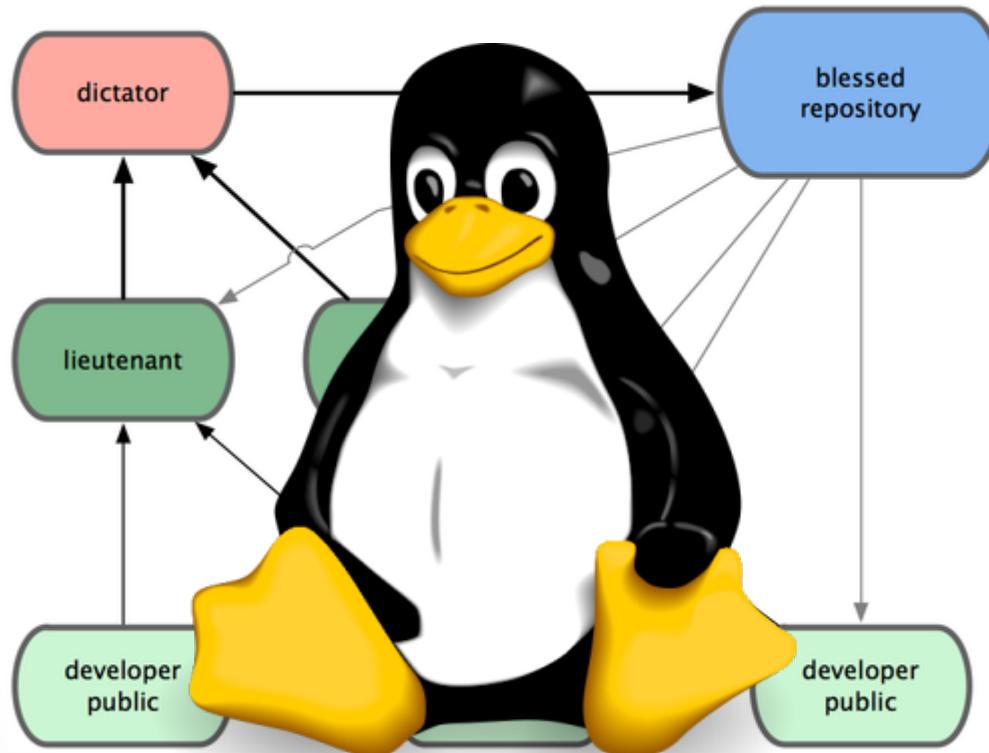


# Workflows macro



Manager d'intégration

# Workflows macro

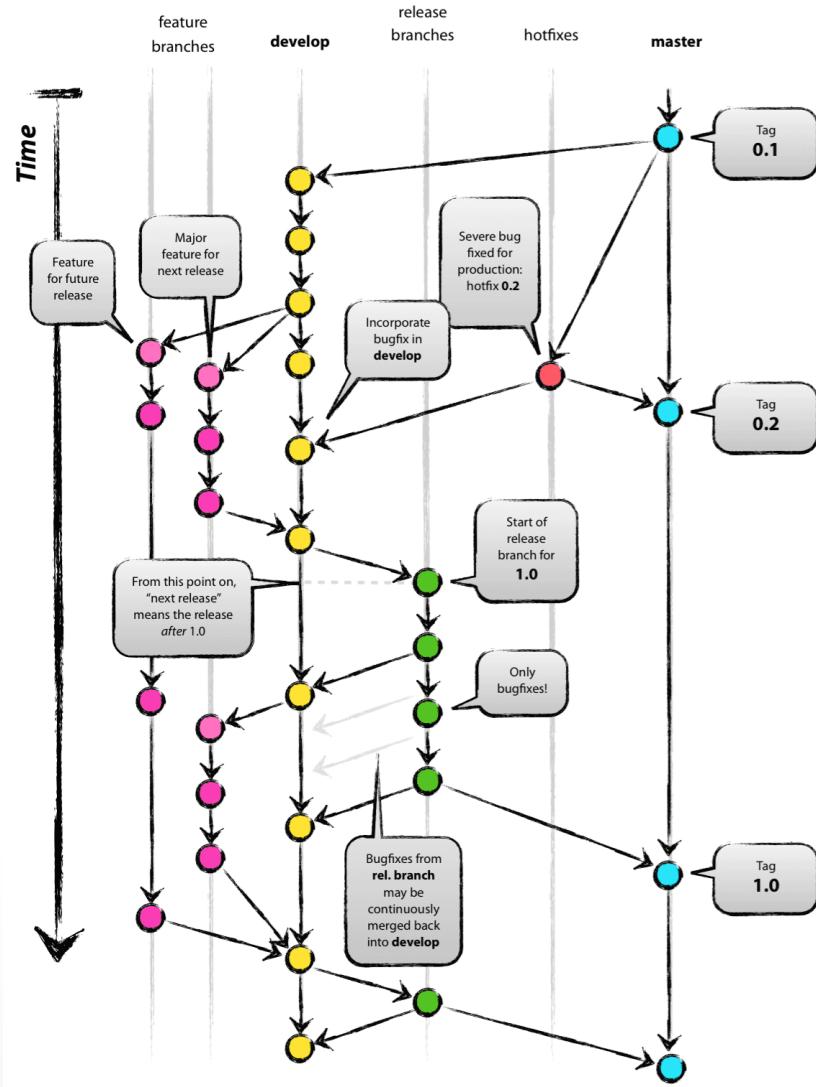


Dictateur et lieutenants d'intégration

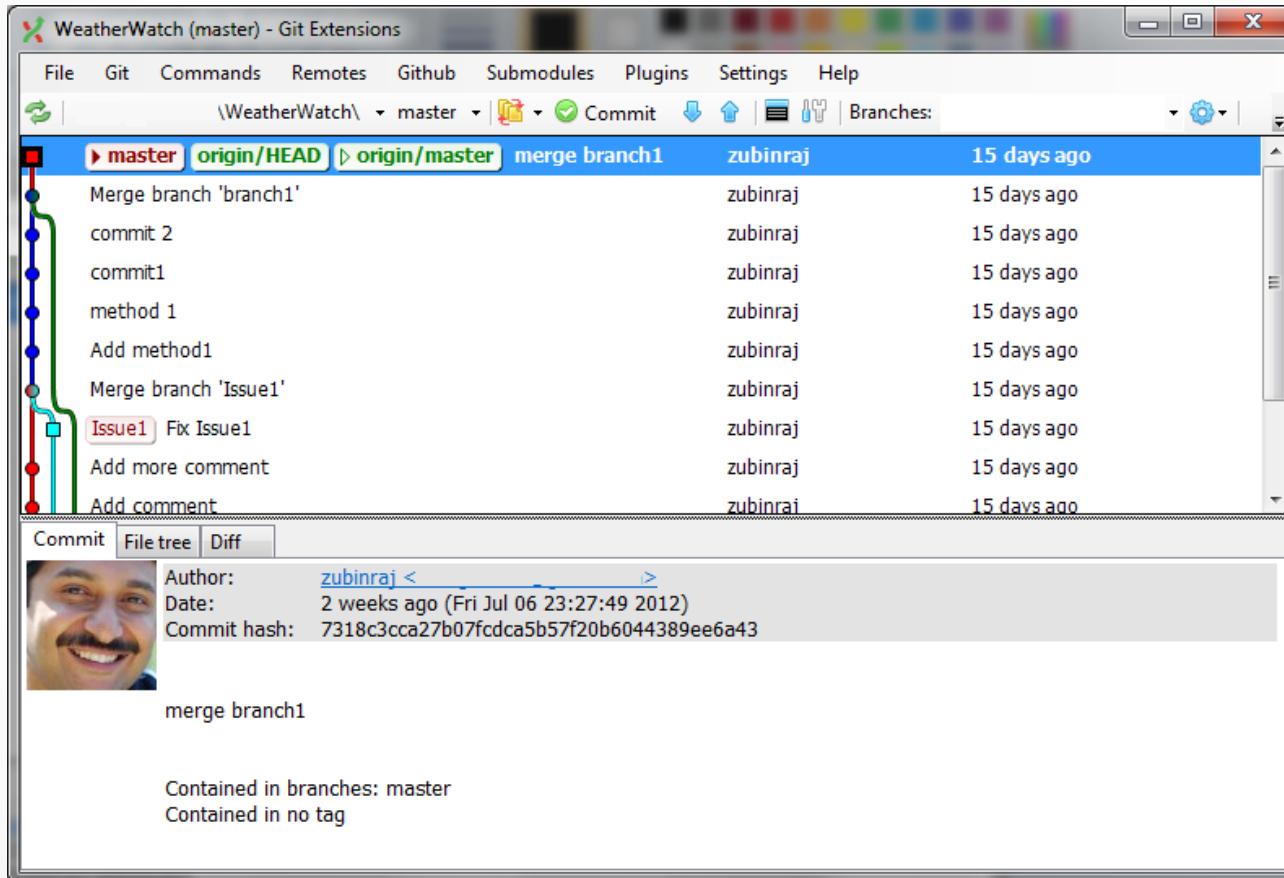
# Workflow micro

Gitflow – A successful one

# Workflow micro



# Git Extensions



<http://sourceforge.net/projects/gitextensions/>

# Hands on !

• • •



facebook



Microsoft

vmware



redhat

LinkedIn

mozilla

Né en 2008

8M+ 14M+ utilisateurs

~~~20,5M+~~ 35M+ repositories

« GitHub is the best place to share code with friends, co-workers, classmates, and complete strangers. Over a million people use GitHub to build amazing things together. »

# Hands on !

• • •

# Ressources

• • •

# Cheat sheet

- <http://zrusin.blogspot.fr/2007/09/git-cheat-sheet.html>

## Git Cheat Sheet

<http://git.or.cz/>

Remember: `git command --help`

Global Git configuration is stored in `$HOME/.gitconfig` (`git config --help`)

### Create

From existing data  
`cd ~/projects/myproject`  
`git init`  
`git add .`

From existing repo  
`git clone ~/existing/repo ~/new/repo`  
`git clone git://host.org/project.git`  
`git clone ssh://you@host.org/proj.git`

### Show

Files changed in working directory  
`git status`

Changes to tracked files  
`git diff`

What changed between \$ID1 and \$ID2  
`git diff $ID1 $ID2`

History of changes  
`git log`

History of changes for file with diffs  
`git log -p $file $directory/`

Who changed what and when in a file  
`git blame $file`

A commit identified by \$ID  
`git show $id`

A specific file from a specific \$ID  
`git show $id:$file`

All local branches  
`git branch`

(star \* marks the current branch)  
`git branch`

### Concepts

#### Git Basics

master : default development branch  
origin : default upstream repository  
HEAD : current branch  
HEAD^ : parent of HEAD  
HEAD~4 : the great-great grandparent of HEAD

#### Revert

Return to the last committed state  
`git reset --hard` ⚠ you cannot undo a hard reset

Revert the last commit  
`git revert HEAD` Creates a new commit

Revert specific commit  
`git revert $id` Creates a new commit

Fix the last commit  
`git commit -a --amend` (after editing the broken files)

Checkout the \$id version of a file  
`git checkout $id $file`

#### Branch

Switch to the \$id branch  
`git checkout $id`

Merge branch1 into branch2  
`git checkout $branch2`  
`git merge $branch1`

Create branch named \$branch based on the HEAD  
`git branch $branch`

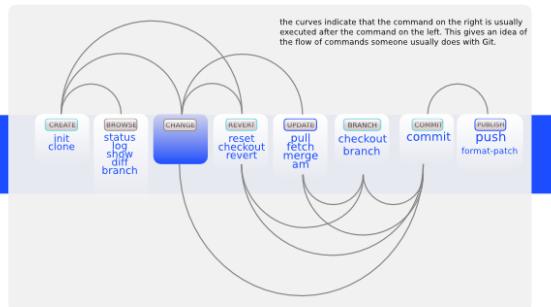
Create branch \$new\_branch based on branch \$other and switch to it  
`git checkout -b $new_branch $other`

Delete branch \$branch  
`git branch -d $branch`

### Cheat Sheet Notation

\$id : notation used in this sheet to represent either a commit id, branch or a tag name  
\$file : arbitrary file name  
\$branch : arbitrary branch name

### Commands Sequence



### Update

Fetch latest changes from origin  
`git fetch` (but this does not merge them)

Pull latest changes from origin  
`git pull` (does a fetch followed by a merge)

Apply a patch that some sent you  
`git am -3 patch.mbox` (in case of a conflict, resolve and use  
`git am --resolved`)

### Publish

Commit all your local changes  
`git commit -a`

Prepare a patch for other developers  
`git format-patch origin`

Push changes to origin  
`git push`

Mark a version / milestone  
`git tag v1.0`

### Useful Commands

#### Finding regressions

`git bisect start` (to start)  
`git bisect good $id` \$id is the last working version  
`git bisect bad $id` \$id is a broken version  
`git bisect bad/good` (to mark it as bad or good)  
`git bisect visualize` (to launch gitk and mark it)  
`git bisect reset` (once you're done)

Check for errors and cleanup repository  
`git fsck`  
`git gc -prune`

Search working directory for foo()  
`git grep "foo()"`

#### Resolve Merge Conflicts

To view the merge conflicts  
`git diff` (complete conflict diff)  
`git diff --base $file` (against base file)  
`git diff --ours $file` (against your changes)  
`git diff --theirs $file` (against other changes)

#### To discard conflicting patch

`git reset --hard`  
`git rebase --skip`

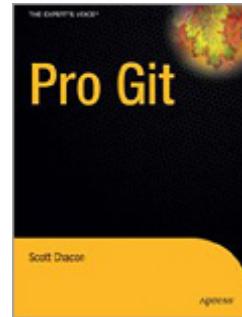
#### After resolving conflicts, merge with

`git add $conflicting_file` (do for all resolved files)  
`git rebase --continue`

Font Awesome  
Icons by [Font Awesome](#). Icons are made by [Font Awesome](#).

# Docs et liens

- <http://git-scm.com>
- <http://nvie.com/posts/a-successful-git-branching-model/>
- <http://git-scm.com/book>



- <http://learn.github.com/p/intro.html>
- <http://gitfr.net/>
- <https://groups.google.com/forum/?fromgroups#!forum/gitfr>