

Git : cheatsheet

1 Configuration globale

```
Nom      $ git config --global user.name "Your Name"
Email    $ git config --global user.email "foo@bar.be"
Couleur  $ git config --global color.ui auto
```

2 Création du dépôt

Dans le dossier principal du projet :

```
$ git init
```

Ou bien faire un clone d'un projet existant :

```
$ git clone <url>
```

3 Commandes de base

Voir l'état actuel du dépôt :

```
$ git status
```

Voir l'historique :

```
$ git log [-p]
```

```
$ git lola
```

Voir les modifications en cours :

```
$ git diff
```

4 Création d'un *commit*

Marquer ce qu'il faut mettre dans le *commit*

En fonction du type de modification, il faut utiliser la commande appropriée.

```
Modifier  $ git add <file>1
Ajouter   $ git add <file>
Supprimer $ git rm <file>
Déplacer  $ git mv <old_name> <new_name>
```

¹ Si on veut marquer toutes les modifications (mais pas les nouveau fichiers), il y a un raccourci : `$ git commit -a`.

Écrire le message du commit et le créer

```
$ git commit
```

L'option `-m` peut être utilisée pour passer le message directement :

```
$ git commit -m "message du commit"
```

5 Branches

Créer une branche

```
$ git branch <name>
```

Changer de branche

```
$ git checkout <branch>
```

Merge une branche

Merge la branche courante avec `other_branch`.

```
$ git merge <other_branch>
```

Rebase une branche

Rebase la branche courante « au dessus » de `other_branch`.

```
$ git checkout <other_branch>
```

Prendre les modifications distantes

```
$ git pull
```

Envoyer ses modifications

```
$ git push
```



Git : un gestionnaire de versions intelligent

Benoit Daloze et Sébastien Wilmet

16 avril 2012

+++ git



Les gestionnaires de versions

Commandes de base, créer un commit

Gérer plusieurs branches

Autres fonctionnalités



Les gestionnaires de versions

Commandes de base, créer un commit

Gérer plusieurs branches

Autres fonctionnalités

Moyens primitifs

Gérer un projet de programmation sans gestionnaire de versions :

- ▶ S'envoyer l'entièreté du code par mail ;
- ▶ Partage de fichiers sur un serveur (dropbox, ...) ;
- ▶ S'envoyer des *patches* :
 - ▶ commande `diff` : différence entre deux fichiers/dossiers ;
 - ▶ commande `patch` : appliquer le *patch* (la *diff*).
- ▶ etc.

Moyens primitifs

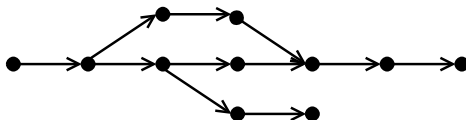
Gérer un projet de programmation sans gestionnaire de versions :

- ▶ S'envoyer l'entièreté du code par mail ;
- ▶ Partage de fichiers sur un serveur (dropbox, ...) ;
- ▶ S'envoyer des *patches* :
 - ▶ commande `diff` : différence entre deux fichiers/dossiers ;
 - ▶ commande `patch` : appliquer le *patch* (la *diff*).
- ▶ etc.

Pas très pratique !

Buts d'un gestionnaire de versions

- ▶ Faciliter la gestion d'un projet de programmation ;
- ▶ Garder l'historique de toutes les modifications (*commits*) ;
- ▶ Travailler en équipe ;
- ▶ Avoir des branches de développement :
 - ▶ pour développer une fonctionnalité séparément ;
 - ▶ pour une certaine version (2.4.0 → 2.4.1 → ...).



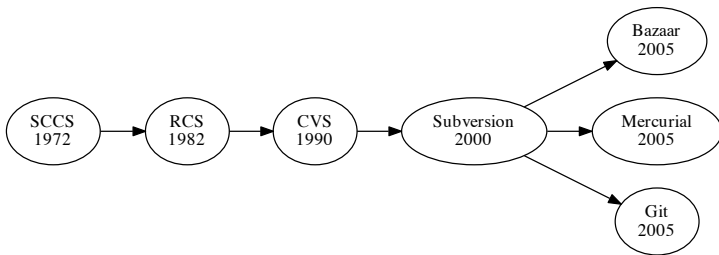
Micro-commits

Un *commit* = **une** modification bien particulière

Avantages :

- ▶ Plus facile à comprendre pour les autres
- ▶ Possibilité d'annuler un changement facilement (`git revert`)
- ▶ Trouver l'origine d'un bug (`git bisect`)
- ▶ ...

Historique des gestionnaires de versions



Subversion (SVN) est **centralisé** :

- ▶ Un serveur central contient toutes les données ;
- ▶ Beaucoup de requêtes entre le client et le serveur (assez lent) ;
- ▶ Besoin d'une connexion internet pour travailler.

Git est **décentralisé/distribué** :

- ▶ Toutes les données sont sur notre machine ;
- ▶ Les opérations sont très rapides ;
- ▶ Connexion internet seulement pour les *pull* et *push*.

Git est **décentralisé/distribué** :

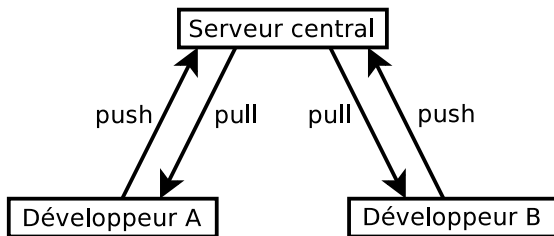
- ▶ Toutes les données sont sur notre machine ;
- ▶ Les opérations sont très rapides ;
- ▶ Connexion internet seulement pour les *pull* et *push*.

Git est aussi **plus puissant** et **plus flexible** :

- ▶ Pour la gestion des branches ;
- ▶ Possède de nombreuses fonctionnalités plus avancées.

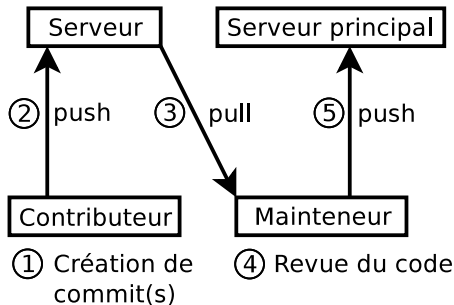
Serveur central avec Git

- ▶ Une manière simple de travailler en équipe ;
- ▶ Accès en écriture pour tous les développeurs.



Git est décentralisé

- ▶ Seul les mainteneurs ont accès en écriture ;
- ▶ Les contributeurs font des *pull requests*.





Les gestionnaires de versions

Commandes de base, créer un commit

Gérer plusieurs branches

Autres fonctionnalités

Créer le dépôt Git

Pour un nouveau projet :

```
$ mkdir project  
$ cd project/  
$ git init
```


Créer le dépôt Git

Pour un nouveau projet :

```
$ mkdir project  
$ cd project/  
$ git init
```

Pour un projet existant :

```
$ git clone git://example.net/project  
$ cd project/
```

Créer le dépôt Git

Pour un nouveau projet :

```
$ mkdir project  
$ cd project/  
$ git init
```

Pour un projet existant :

```
$ git clone git://example.net/project  
$ cd project/
```

Répertoire caché .git/ (unique) :

```
$ ls .git/  
config  description  HEAD  hooks/  info/  objects/  refs/
```

États d'un fichier

Untracked

- ▶ non pris en compte par Git

États d'un fichier

Untracked

- ▶ non pris en compte par Git

Unmodified/Committed

- ▶ aucune modification

États d'un fichier

Untracked

- ▶ non pris en compte par Git

Unmodified/Committed

- ▶ aucune modification

Modified

- ▶ fichier modifié
- ▶ pas pris en compte pour le prochain commit

États d'un fichier

Untracked

- ▶ non pris en compte par Git

Unmodified/Committed

- ▶ aucune modification

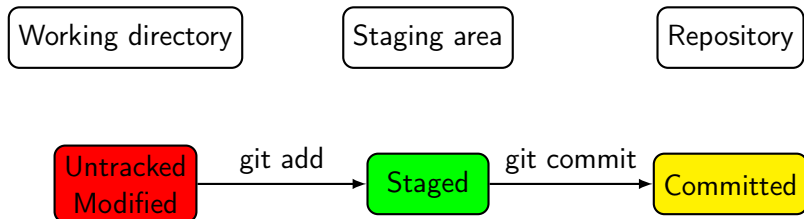
Modified

- ▶ fichier modifié
- ▶ pas pris en compte pour le prochain commit

Staged

- ▶ fichier ajouté, modifié, supprimé ou déplacé
- ▶ pris en compte pour le prochain commit

États d'un fichier



Créer un nouveau fichier

```
$ echo hello > README
```

État du fichier : ***untracked***

Créer un nouveau fichier

```
$ echo hello > README
```

État du fichier : *untracked*

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
nothing added to commit but untracked files present
```

Créer un nouveau fichier

```
$ git add README
```

État du fichier : *untracked* → *staged*

Créer un nouveau fichier

```
$ git add README
```

État du fichier : *untracked* → *staged*

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#
```

Créer le commit

```
$ git commit  
[écrire le message du commit]
```

Créer le commit

```
$ git commit  
[écrire le message du commit]
```

```
$ git log  
commit c3aab8bb6cca644162a4fa82df283682717da3d4  
Author: Your Name <foo@bar.be>  
Date:   Wed Feb 1 15:19:03 2012 +0100
```

Titre du commit (pas trop long)

Plus longue description.
Ligne vide après le titre.

Modifier un fichier

État du fichier README : *unmodified*

Modifier un fichier

État du fichier README : *unmodified*

```
$ echo world >> README
```

État : *modified*

Modifier un fichier

État du fichier README : *unmodified*

```
$ echo world >> README
```

État : *modified*

```
$ git add README
```

État : *staged*

Modifier un fichier

État du fichier README : *unmodified*

```
$ echo world >> README
```

État : *modified*

```
$ git add README
```

État : *staged*

```
$ git commit
```

État : *committed*

Diff

Voir les modifications avant de créer un commit.

```
$ echo new-text > README
```

```
$ git diff
diff --git a/README b/README
index 2e85c45..4320c6f 100644
--- a/README
+++ b/README
@@ -1,2 +1 @@
-hello
-world
+new-text
```

La liste des commandes :

```
$ git help
```

Page de manuel d'une commande :

```
$ git help <cmd>
```

```
$ git help <cmd> --web
```

```
$ git <cmd> --help
```

Résumé des commandes

- ▶ `git init`
- ▶ `git clone`
- ▶ `git status`
- ▶ `git add <file>`
- ▶ `git rm/mv <file>`
- ▶ `git commit`
- ▶ `git log`
- ▶ `git diff`
- ▶ `git help`



Les gestionnaires de versions

Commandes de base, créer un commit

Gérer plusieurs branches

Autres fonctionnalités

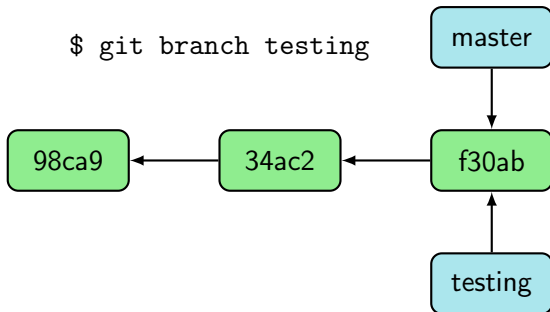
Créer une branche

Une branche :

- ▶ Pointe vers un commit ;
- ▶ Le pointeur « avance » quand on crée un commit.

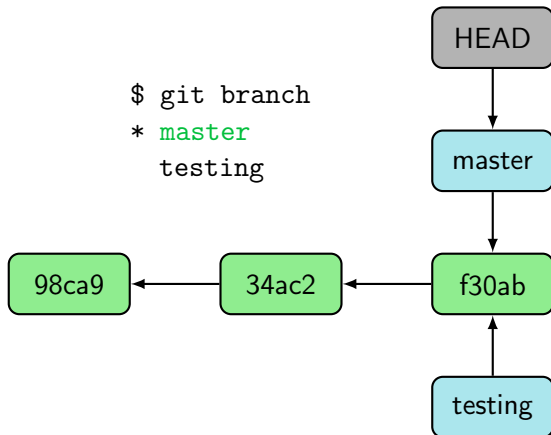
Un commit pointe vers son ou ses commit(s) parent(s).

Créons une branche :

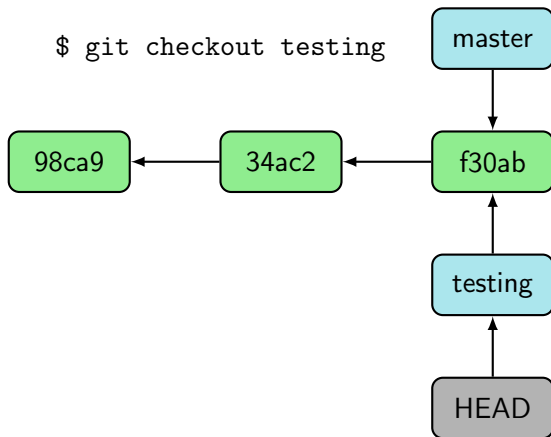


HEAD

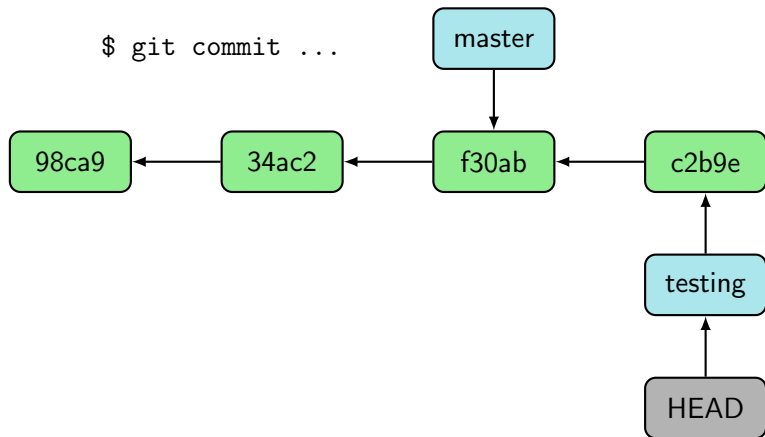
Le pointeur HEAD pointe vers la branche courante.



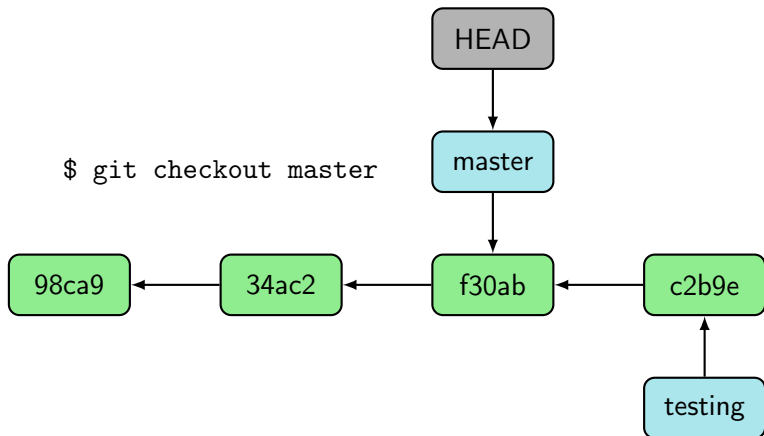
Changer de branche



Créer un commit sur testing

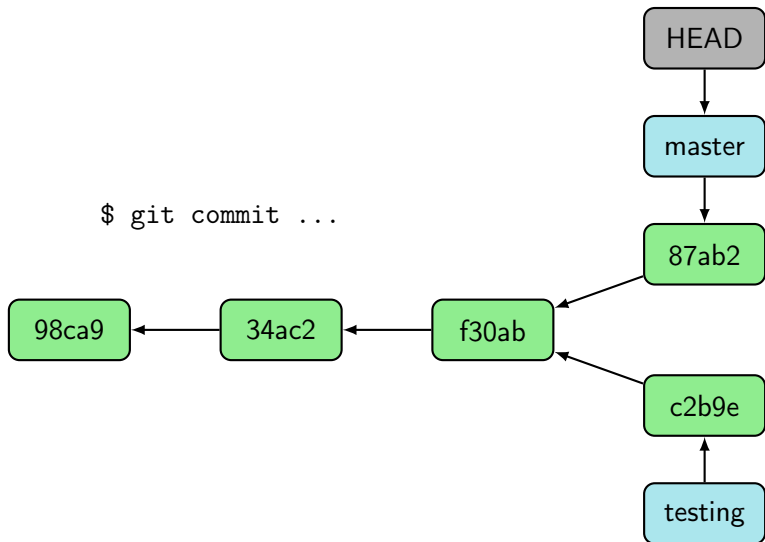


Revenir sur master

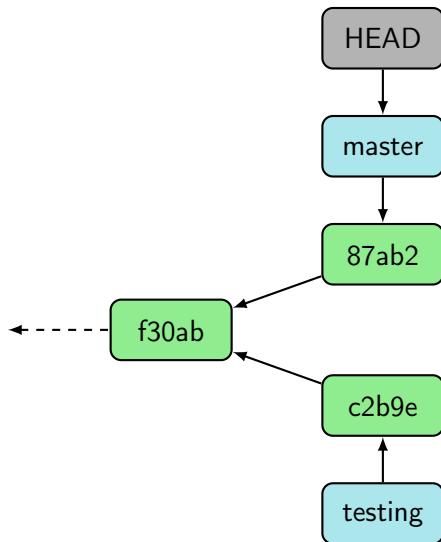


Créer un commit sur master

`$ git commit ...`

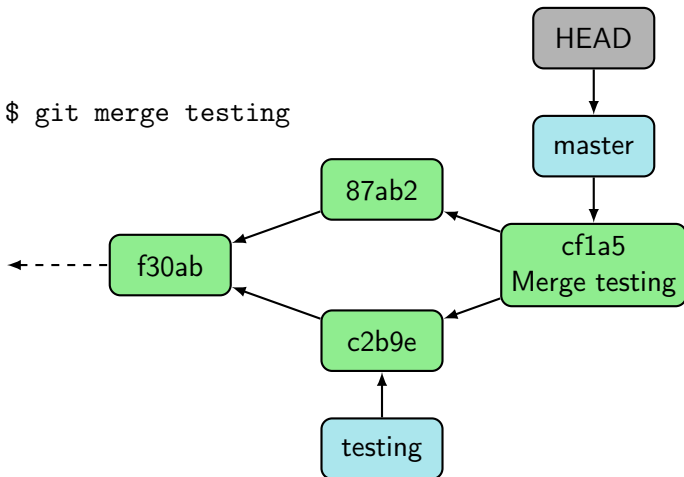


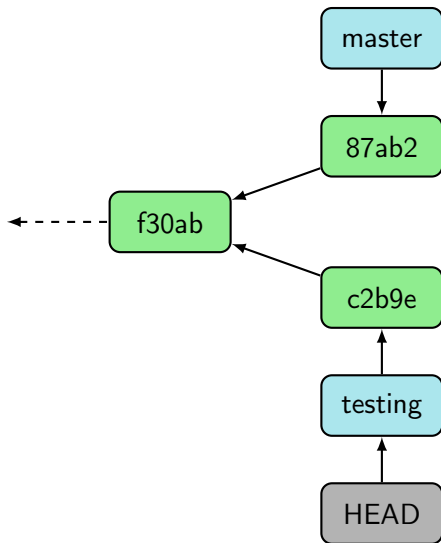
Divergence



Merge

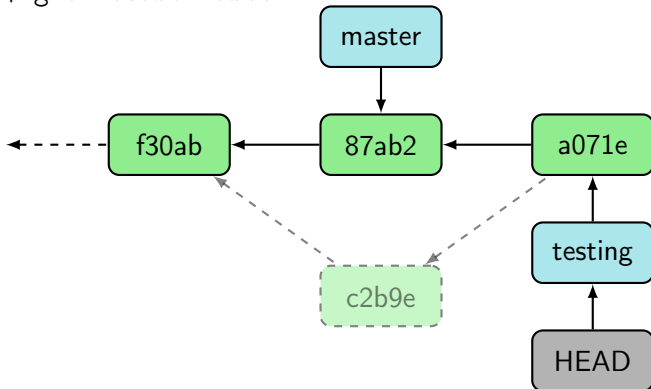
\$ git merge testing





Rebase

```
$ git rebase master
```



Remotes

Remote : référence vers un dépôt distant.

```
$ git remote -v
origin git://git.kernel.org/pub/scm/git/git.git (fetch)
origin git://git.kernel.org/pub/scm/git/git.git (push)
```

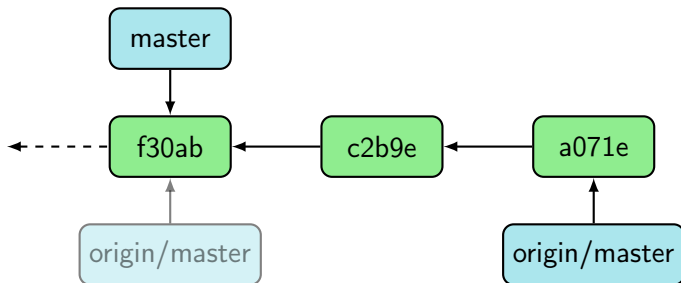
```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/man
  remotes/origin/master
  remotes/origin/next
```


git pull [remote] [branch]

```
git pull origin master
```

- ▶ git fetch origin
- ▶ git merge origin/master

```
$ git fetch origin
```

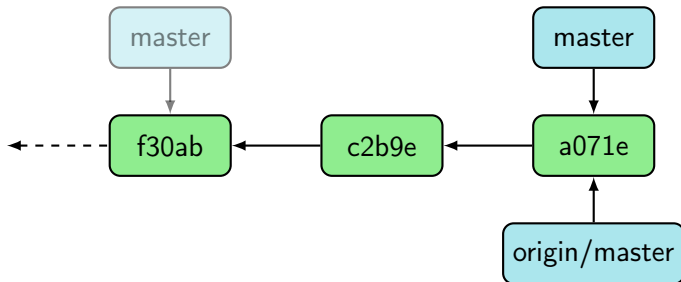


git pull [remote] [branch]

```
git pull origin master
```

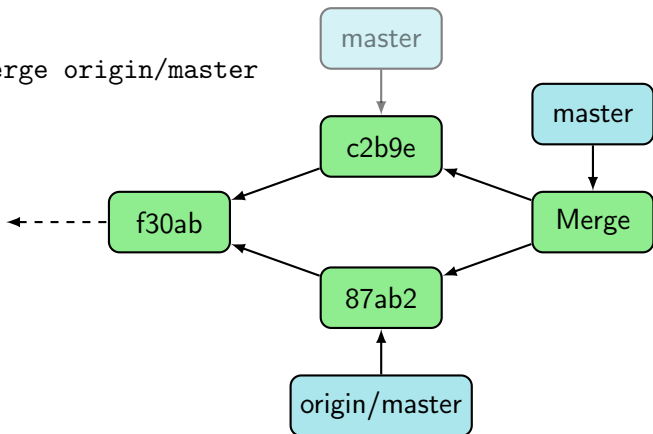
- ▶ git fetch origin
- ▶ git merge origin/master

```
$ git merge origin/master
```



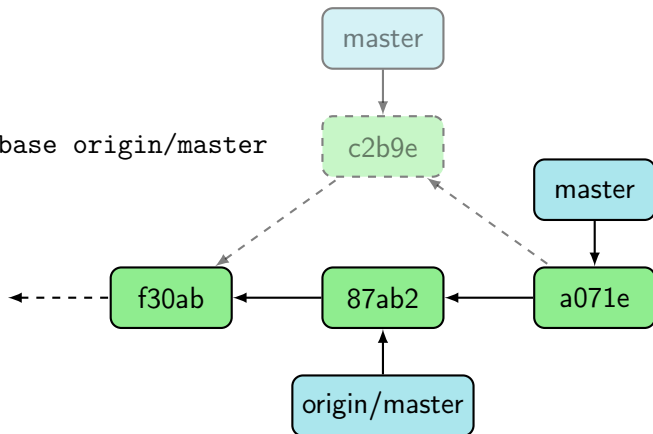
git pull : merge

\$ git merge origin/master



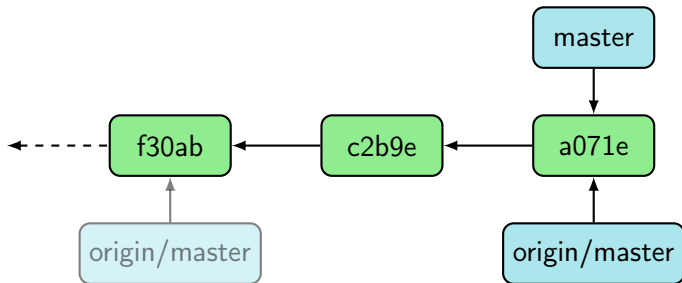
git pull --rebase

\$ git rebase origin/master



```
git push [remote] [branch]
```

```
$ git push origin master
```



Commandes principales

browse

- ▶ `git status`
- ▶ `git log`
- ▶ `git diff`

commit

- ▶ `git add`
- ▶ `git commit`

branch

- ▶ `git branch / checkout`
- ▶ `git merge / rebase`

remote

- ▶ `git pull`
- ▶ `git push`



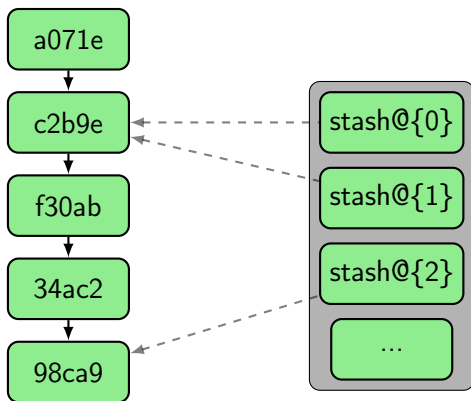
Les gestionnaires de versions

Commandes de base, créer un commit

Gérer plusieurs branches

Autres fonctionnalités

git stash



git stash

git stash

- ▶ save [message]

git stash

git stash

- ▶ save [message]
- ▶ list
- ▶ show [stash]

git stash

git stash

- ▶ save [message]
- ▶ list
- ▶ show [stash]
- ▶ apply [stash]
- ▶ pop [stash]

git stash save

```
$ git status
# On branch master
# Changes not staged for commit:
#
#       modified:   main.c
#
no changes added to commit (use "git add"/"git commit -a")

$ git stash save
Saved working directory and index state WIP on master: 8540fea
HEAD is now at 8540fea message

$ git status
# On branch master
nothing to commit (working directory clean)
```

git stash list, show

```
$ git stash list
stash@{0}: WIP on master: 8540fea message
```

```
$ git stash show -p
diff -git a/main.c b/main.c
index f2ad6c7..2f773ae 100644
--- a/main.c
+++ b/main.c
@@ -1,3 +1,5 @@
+#include <stdio.h>
+
int main() {
    printf("Hello, world!");
    return 0;
}
```

Commandes avancées

git grep

- ▶ `grep(1)` sur les fichiers pris en compte par git

git tag

- ▶ crée un *tag*, une référence fixe vers un commit

git revert

- ▶ crée un *commit* qui annule un autre

git blame

montre qui a modifié le fichier, ligne par ligne

```
$ git blame git.c
85023 (Junio C Hamano      2006-12-19  1) #include "builtin.h"
2b11e (Johannes Schindelin 2006-06-05  2) #include "cache.h"
fd5c3 (Thiago Farina       2010-08-31  3) #include "exec_cmd.h"
fd5c3 (Thiago Farina       2010-08-31  4) #include "help.h"
575ba (Matthias Lederhofer 2006-06-25  5) #include "quote.h"
d8e96 (Jeff King           2009-01-28  6) #include "run-command.h"
8e49d (Andreas Ericsson    2005-11-16  7)
822a7 (Ramsay Allan Jones   2006-07-30  8) const char git_usage_string[]
f2dd8 (Jon Seymour         2011-05-01  9)      "git [--version] [--ex
```

git reset

Change le pointeur de la branche courante

```
$ git reset HEAD^ # recule la branche d'un commit
```

Unstaged changes after reset:

```
M      README
```

```
$ git status
```

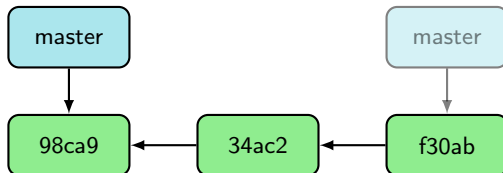
```
# On branch master
```

```
# Changes not staged for commit:
```

```
#
```

```
#      modified:   README
```

```
$ git reset --hard 98ca9
```



git commit --amend

Pour modifier le dernier *commit* :

- ▶ ajouter une modification ;
- ▶ modifier le message du *commit*.

```
$ edit some_file  
$ git add some_file  
$ git commit --amend
```

git rebase --interactive

Permet de modifier l'historique

```
$ git rebase -i HEAD~3
```

```
pick 4eeeb5 bulk-checkin: allow the same data to be multiply hashed
pick 127b177 bulk-checkin: support chunked-object encoding
pick 973951a chunked-object: fallback checkout codepaths
```

```
# Rebase 617312b..973951a onto 617312b
```

```
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

git reflog

Log des opérations sur les *commits*

```
$ git reflog
```

```
b0d66 HEAD@{0}: commit (amend): add a setting to require a filter to be
97395 HEAD@{1}: checkout: moving from master to man
b0d66 HEAD@{2}: rebase: aborting
9cda8 HEAD@{3}: rebase -i (pick): grep: drop grep_buffer's "name" param
b9ef9 HEAD@{4}: rebase -i (pick): convert git-grep to use grep_source in
837de HEAD@{5}: rebase -i (pick): grep: make locking flag global
84f3d HEAD@{6}: checkout: moving from master to 84f3d
b0d66 HEAD@{7}: pull: Fast-forward
f6b50 HEAD@{8}: cherry-pick: add a TODO
98c05 HEAD@{9}: reset: moving to HEAD~
e11ee HEAD@{10}: checkout: moving from master to pu
77eac HEAD@{11}: commit: add a TODO
75f43 HEAD@{12}: commit: use the correct format identifier for unsigned
f88cc HEAD@{13}: pull --rebase: git grep
07873 HEAD@{14}: pull : Fast-forward
f70f7 HEAD@{15}: clone: from git://git.kernel.org/pub/scm/git/git.git
```

git add --patch

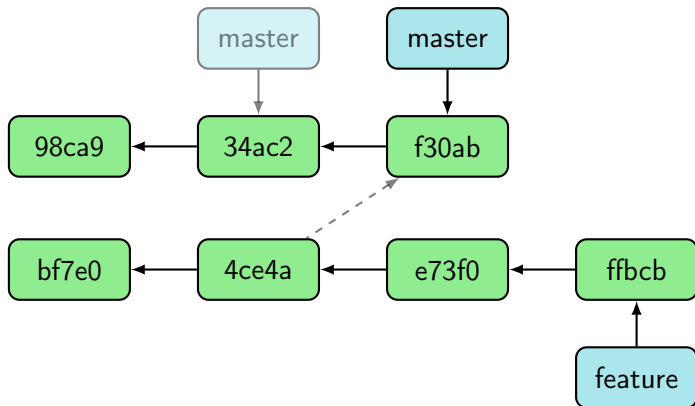
Permet d'ajouter une partie des modifications d'un fichier

```
$ git add -p
diff --git a/README b/README
@@ -1,5 +1,7 @@
 1
+2
 3
+4
Stage this hunk [y,n,q,a,d,/,s,e,?]? s
Split into 2 hunks.
@@ -1,2 +1,3 @@
 1
+2
 3
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? n
@@ -2,4 +3,5 @@
 3
+4
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
```

git cherry-pick

Applique un *commit* sur la branche courante.

```
$ git cherry-pick 4ce4a
```



git bisect

Permet de trouver l'origine d'un bug

```
$ git bisect start bad_commit good_commit  
Bisecting: 20 revisions left to test after this (roughly 4 steps)  
[e4f3edc] Sync with maint
```

```
$ git bisect run ./mytests  
Bisecting: 9 revisions left to test after this (roughly 3 steps)  
[42226d] pack-objects: remove bogus comment
```

```
...  
Bisecting: 0 revisions left to test after this (roughly 1 step)  
[9a4749] checkout -m: no need to insist on having all 3 stages
```

```
8280f2e0a0f4776c4b3008c9172fc0a3e7361534 is the first bad commit  
commit 8280f2e0a0f4776c4b3008c9172fc0a3e7361534
```

```
Author: eregon <mail@me>
```


```
Date: Sat Feb 18 17:36:47 2012 +0100
```

Noooo! You found my hidden bug!

Services d'hébergement

- ▶ GitHub
- ▶ Gitorious
- ▶ Bitbucket, assembla (dépôts privés gratuits)
- ▶ En INGI (voir le wiki)
`http://wiki.student.info.ucl.ac.be/index.php/Git`

Liens

- 
- ▶ Les manpages : `git help [--web] <cmd>`
 - ▶ <http://git-scm.com/book> : Pro Git book
 - ▶ <http://www.siteduzero.com/informatique/tutoriels/gerez-vos-codes-source-avec-git>
 - ▶ <http://eregon.me/git/> : Slides, aide-mémoire, exercices

Exercices sur git

Benoit Daloze

Sébastien Wilmet

16 avril 2012

1 Échauffement

Ouvrez un terminal (sous Windows, lancez "Git bash" depuis le menu Démarrer). Commencez par vous identifier à git (à ne faire qu'une fois) :

```
$ git config --global user.name "Your Name"
$ git config --global user.email "foo@bar.be"
$ git config --global color.ui auto
```

Puis créez un dépôt git :

```
$ mkdir exercices
$ cd exercices
$ git init
```

Le premier but est de créer un *commit*. Comme il nous faut quelque chose à mettre dedans, créez un fichier :

```
$ echo "I'm learning git"> README
```

Faites un `git status` pour voir où vous en êtes. Git liste votre fichier comme *untracked* et vous indique qu'il faut utiliser `git add` pour prendre en compte ce fichier :

```
$ git add README
```

`git status` indique maintenant le fichier dans la partie *Changes to be committed*. Créez donc ce *commit*. L'option `-m` permet de spécifier le titre du *commit* directement dans la ligne de commande. Si l'option `-m` n'est pas utilisée, un éditeur de texte s'ouvre¹.

```
$ git commit -m "first commit"
```

Vous pouvez maintenant vérifier que votre dossier (*working directory*) est propre avec `git status`. Vérifiez que le *commit* a bien été créé, en utilisant `git log`, avec l'option `-p` pour voir les changements.

Modifiez le fichier README et regardez la *diff* :

```
$ git diff
```

La modification n'est pas encore dans la *staging area*, c'est-à-dire qu'il n'y a encore rien pour le prochain *commit*. Si vous voulez commiter le changement, nous avons vu qu'il est possible de faire un `git add` suivi d'un `git commit`. Cependant, il est possible de le faire en une seule étape, grâce à :

```
$ git commit -a
```

Quand vous faites cela, vérifiez bien avant avec `git status` ainsi que `git diff` que tous les changements sont OK pour faire le *commit*.

1. En console l'éditeur de texte peut être vim, nano, etc. Sous Unix cela dépend de la variable d'environnement EDITOR, que vous pouvez modifier si besoin : `export EDITOR=nano`

2 Branches, *merges* et autres *rebases*

Créez une branche `test` :

```
$ git branch test
$ git branch
```

Créez un *commit* sur `master` qui modifie le `README`.

Passez sur la branche `test` :

```
$ git checkout test
```

Créez un *commit* qui rajoute un *nouveau* fichier (pour ne pas qu'il y ait de conflits plus tard).

Voyez maintenant où vous en êtes avec :

```
$ git log --graph --decorate --oneline --all
```

Comme c'est une commande pratique, vous pouvez en faire un alias :

```
$ git config --global alias.lola "log --graph --decorate --oneline --all"
$ git lola
```

Il est temps de faire un *merge* pour intégrer la branche `test` dans la branche `master` :

```
$ git checkout master
$ git merge test
$ git lola
```

Un *rebase* aurait permis d'avoir un historique linéaire. Annulez d'abord le dernier *commit*, qui est le *merge* (attention commande dangereuse, vérifiez bien avec `git lola` ou `git log` que vous vous trouvez bien sur le bon *commit* !) :

```
$ git reset --hard HEAD~1
```

Vous êtes normalement revenu à l'état précédent, comme si le *merge* n'avait pas eu lieu.

Faites maintenant le *rebase* :

```
$ git checkout test
$ git rebase master
$ git lola
```

La branche `test` se trouve maintenant juste au-dessus de `master`, le *merge* se fera en « *fast-forward* » :

```
$ git checkout master
$ git merge test
```

OK toujours vivant ? Compliquons un peu les choses : faites un *commit* sur `master` qui modifie une certaine ligne d'un fichier. Allez sur la branche `test` et créez un autre *commit* qui modifie exactement la même ligne. Le *merge* donnera un conflit. En voici un exemple :

Avant	hello world
Branche <code>master</code>	goodby world
Branche <code>test</code>	Hello, world!
Merge	Goodby, world!

Éditez le fichier pour régler le conflit, et suivez les instructions données par la commande `merge`. Une fois terminé, la branche `test` ne sert plus à rien, vous pouvez la supprimer :

```
$ git branch -d test
```

3 Pour aller plus loin

Faites des modifications, puis imaginez que vous voulez changer de branche. Il vous faut pour cela un dossier propre. Utilisez `git stash save`, changez de branche et puis revenez et faites un `git stash pop`.

Changez votre dernier *commit* en ajoutant une autre modification à l'aide de `git commit --amend`.

Essayez d'appliquer un *commit* dont vous trouverez la référence avec `git log` sur la branche courante en utilisant `git cherry-pick`.

Voyez un peu ce que vous avez fait avec `git reflog`, et profitez-en pour satisfaire votre curiosité en inspectant une référence avec `git show`.

Notez que `git show` permet facilement de voir un fichier à une référence donnée avec la syntaxe `git show ref:file`.

Comparez la taille de votre répertoire avec du `-ks` avant et après `git gc`. `git` possède un *garbage collector*, qui permet de mieux arranger le contenu du dépôt.

Faites plusieurs modifications dans le même fichier, pas forcément contiguës. Essayez d'en ajouter qu'une partie avec `git add -p`. Pouvez-vous même le faire lorsque les lignes se suivent ? (indice : e)

Créez quelques commits dans une branche. Inversez l'ordre de certains commits avec `git rebase -i HEAD~n`, qui sélectionne les n derniers commits et vous propose différentes actions à faire sur ceux-ci, par exemple renommer le message, fusionner plusieurs commits, changer l'ordre, etc. Regardez le résultat grâce à `git lola`.

Faites quelques modifications et faites un `git add`. Vous aimeriez bien voir la diff de ces modifications prêtes à être incluses dans le prochain *commit* (la *staging area*). Trouvez l'option de `git diff` permettant de faire cela.

Imaginez que vous vous rendez compte qu'il y a un bug quand vous testez le dernier *commit*, et qu'auparavant ce bug n'existait pas. Vous regardez l'historique et vous trouvez un *commit* où vous êtes certain que le bug n'existait pas, le plus récent possible. Avec `git bisect`, il est alors facile de trouver le *commit* qui a introduit le bug. Pour démarrer la recherche dichotomique, faites :

```
$ git bisect start HEAD <bad-commit>
```

Pour l'exercice, assurez-vous qu'entre HEAD et le « bad-commit », il y ait au moins une petite dizaine de commits. `git` va maintenant vous emmener à travers plusieurs *commit* que vous allez devoir marquer comme bon ou mauvais. Utilisez :

```
$ git bisect bad
$ git bisect good
```

Une fois terminé, `git` vous donne le premier *commit* qui introduit le bug. `git lola` vous permet de voir les *commits* que vous avez marqués. Terminez par :

```
$ git bisect reset
```

Pour les slides et un exemple de fichier `$HOME/.gitconfig` : <http://eregon.me/git/>