Miximum

Le blog d'un ingénieur Web freelance.

Enfin comprendre Git : le tutoriel complet

9 juillet 2013 Tutoriel

J'utilise Git quotidiennement depuis plus de dix ans. Bien que Git soit un outil extrêmement puissant, il n'est pas très intuitif. Sans bien comprendre les mécanismes internes du logiciel, on se retrouve vite coincé. Par conséquent, voici un tutoriel ulta-détaillé pour bien appréhender les principes et les principales commandes de Git.

Ce billet fait partie d'une série d'articles sur Git. Qu'est-ce que Git et comment s'en servir ? Quelles sont les commandes les plus utiles ? Comment éviter les pièges courants ? Quelles astuces pour gagner du temps ?

- Découvrir Git : introduction et premiers pas
- ➔ **Enfin comprendre Git : le tutoriel complet**
- Git rebase : qu'est-ce que c'est ? Comment s'en servir ?
- Débusquer une régression avec git bisect
- Activer la coloration avec git
- Ignorer des fichiers avec git
- Créer un patch avec git
- Utiliser git-svn pour interfacer Git avec un depot subversion
- Utiliser Git pour travailler sur un dépot CVS

Je dois pourtant reconnaître que Git n'est pas forcément l'outil le plus abordable qui soit. Toutes ces commandes bizarres ! Toutes ces options apparemment redondantes ! Cette documentation cryptique ! Et ce worflow de travail, qui nécessite 18 étapes pour pousser un

patch sur le serveur. Tel un fier et farouche étalon des steppes sauvages, Git ne se laissera approcher qu'avec cisconspection, et demandera beaucoup de patience avant de s'avouer dompté.

Tenez, prenez l'exemple suivant :

Comment j'annule une modification d'un fichier ?

```
git checkout
```

Ok, comment je change de branche ?

```
git checkout
```

Ok, et comment je créé une nouvelle branche ?

```
git checkout
```

Mmm... Ok, et comment je supprime une branche ?

```
git branch -d ma_branche
```

D'accord, et comment je supprime une branche distante ?

```
git push origin :ma_branche
```

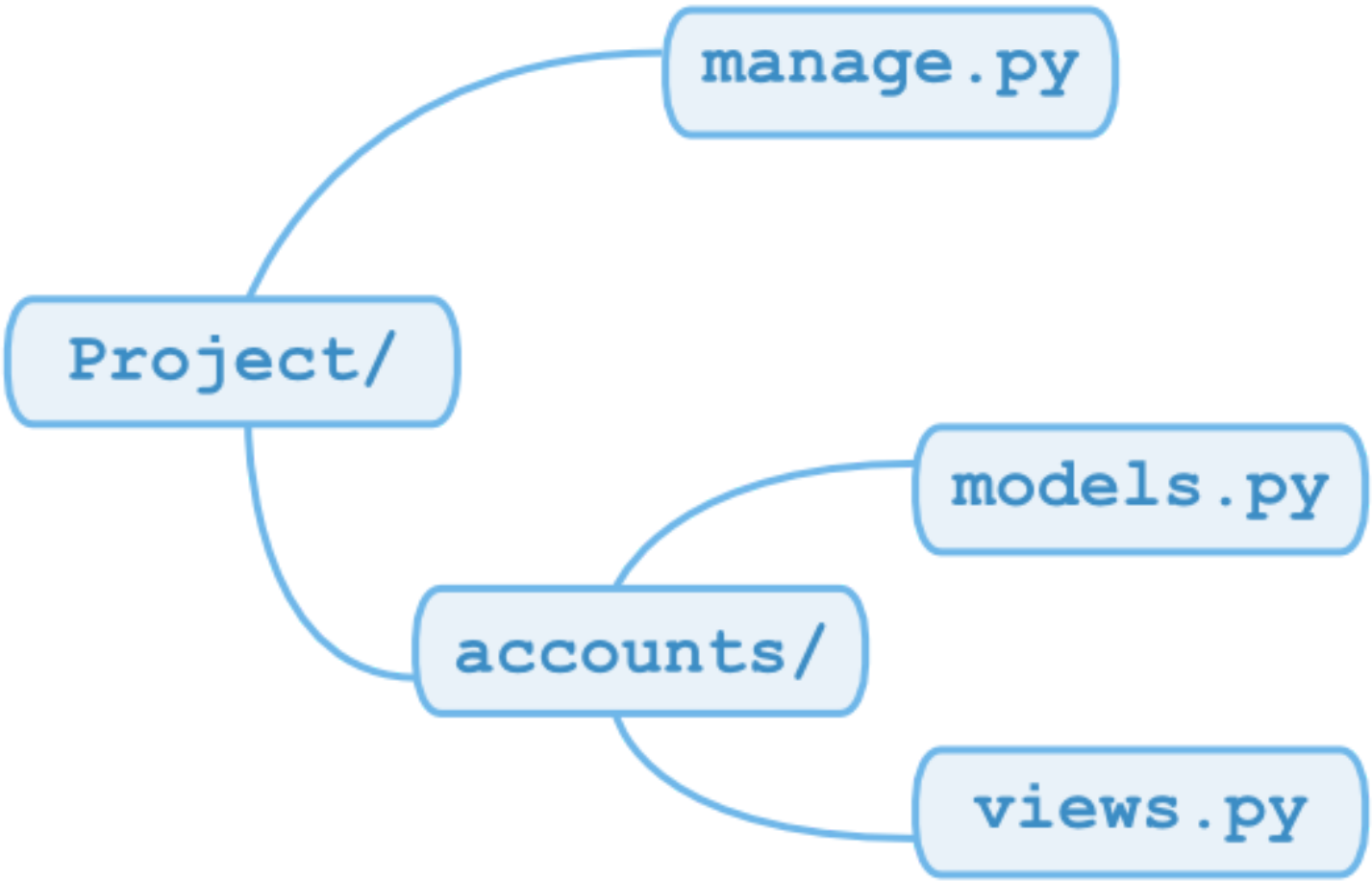
Heu...

À vrai dire, il est trés difficile de percevoir la logique de ce qu'on fait si on n'a pas un minimum de compréhension du fonctionnement interne de Git. De plus, la plupart des commandes sont plutôt bas niveau, ce qui explique leurs effets qui peuvent paraître sans rapports entre eux.

Tâchons de comprendre un peu mieux la bête pour mieux la maîtriser.

Comprendre les zones et le workflow de travail

Imaginons que vous soyez en train de travailler sur un quelconque projet, constitués d'une arborescence de fichiers tout à fait classique.

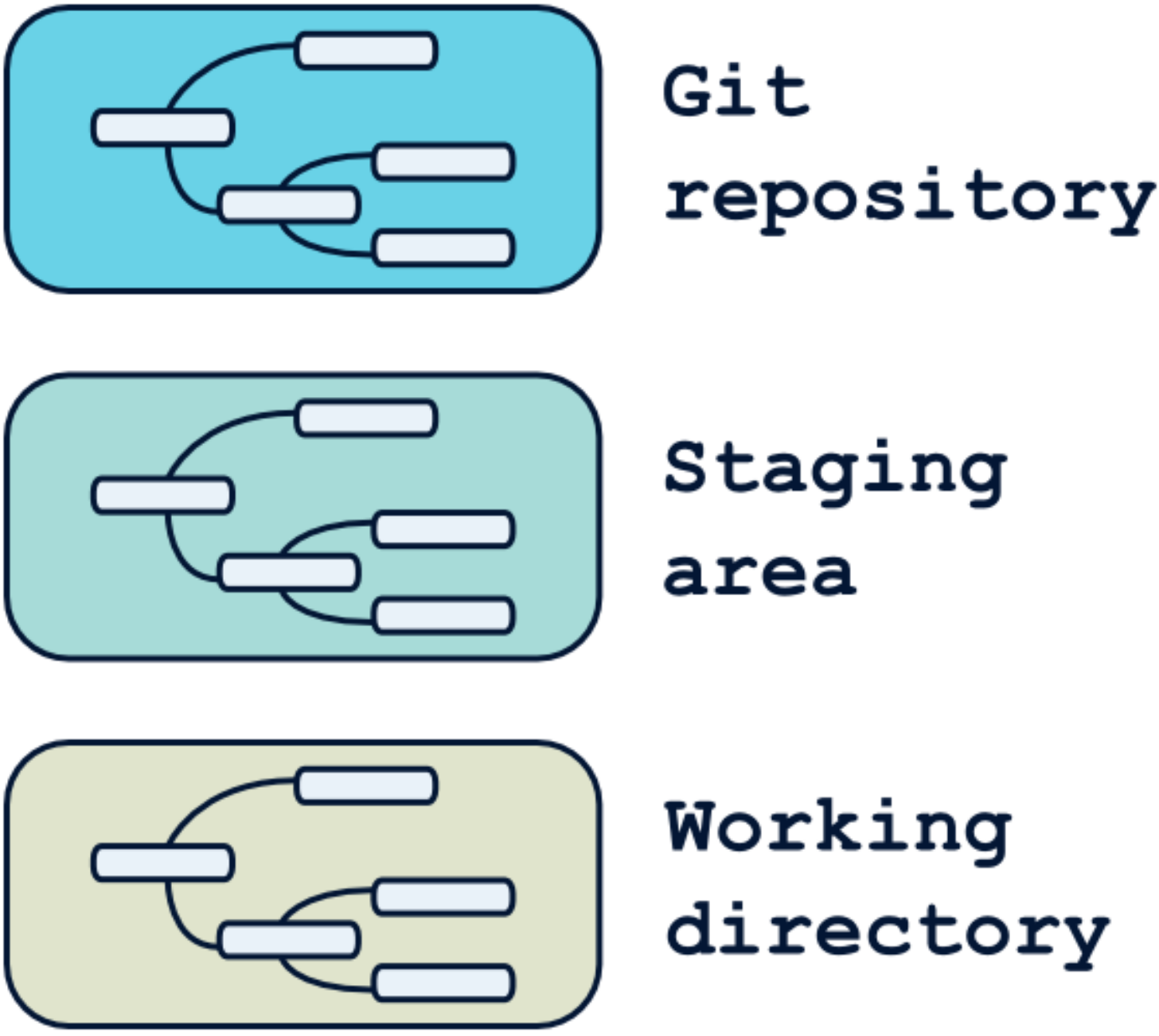


Si votre projet est géré avec Git, on peut grosso-modo considérer que votre arborescence n'est pas stockée une, mais **trois** fois.

D'abord, les fichiers eux-mêmes sur votre disque dur, que vous éditez grâce à votre éditeur préféré. C'est votre répertoire de travail, ou *Working Directory* en angliche.

Ensuite, dans une mystérieuse zone spéciale que l'on appelle l'index, ou la zone de staging.

Enfin, dans la base de données de Git est stockée l'arborescence de votre projet telle qu'elle était lors de votre dernier commit.



Pourquoi trois zones, et pas seulement deux ? Quelle est donc cette mystèrieuse « staging area » ?

Qu'est-ce qu'un bon commit ?

Laissez-moi digresser quelque peu pour rappeler qu'un bon commit est un commit *atomique* :

- il ne concerne qu'une chose et une seule ;
- il est le plus petit possible (tout en restant cohérent).

Pourtant, lorsqu'on travaille sur une fonctionnalité, il n'est pas rare d'en profiter pour corriger une petite faute d'orthographe par ci, un petit bug qui trénouillait par là, et on se retrouve avec un répertoire de travail contenant des modifications totalement indépendantes. Ces modifications doivent alors faire l'objet de commits séparés, et c'est à ça que sert la zone de staging : préparer le prochain commit, en y ajoutant ou retirant des fichiers (ou portions de fichiers) sans toucher à votre répertoire de travail.

Certains y verront sans doute un travail superflu bon à satisfaire les instincts pervers des aficionados d'attouchements intimes sur les diptères. Il n'en est rien, et une fois qu'on y a goûté, il est tout simplement impossible de revenir en arrière.

Commandes de base

Le processus de commit avec Git est donc celui-ci :

1. je développe en modifiant / déplaçant / supprimant des fichiers ;
2. quand une série de modification est cohérente et digne d'être commitée, je la place dans la zone de staging ;
3. je vérifie que l'état de ma zone de staging est satisfaisant ;
4. je committe ;
5. et on répète jusqu'à… euh… ben, la fin quoi.

Pour copier un fichier du répertoire de travail vers la zone de staging, on utilise *git add*.

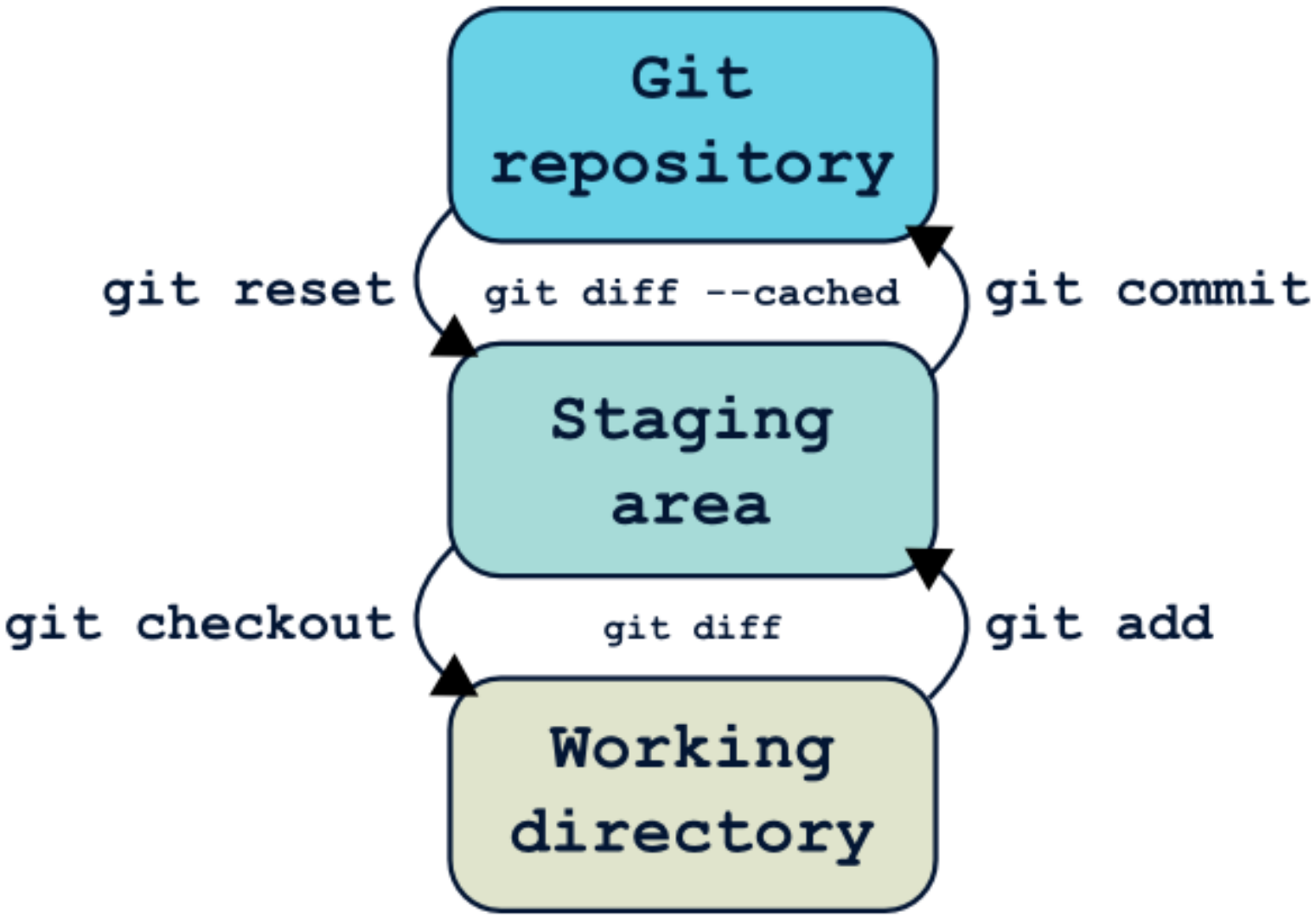
Pour sauvegarder la zone de staging dans le dépôt git et créer un nouveau commit, on utilise *git commit*.

Pour copier un fichier du dépôt Git vers la zone de staging, on utilise *git reset*.

Pour copier un fichier du staging vers le working directory (donc supprimer les modifications en cours), on utilise *git checkout*.

Pour visualiser les modifications entre le répertoire de travail et la zone de staging, on utilise *git diff*.

Pour visualiser les modifications entre la zone de staging et le dernier commit, on utilise *git diff --cached*.



Et comment sait-on quels fichiers sont différents d'une zone à l'autre ? Grâce à *git status* :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD ..." to unstage)
#
#   modified:   accounts/forms.py
#   modified:   accounts/models.py
#
# Changes not staged for commit:
#   (use "git add ..." to update what will be committed)
#   (use "git checkout -- ..." to discard changes in working directory)
#
#   modified:   accounts/urls.py
#   modified:   accounts/views.py
#
```

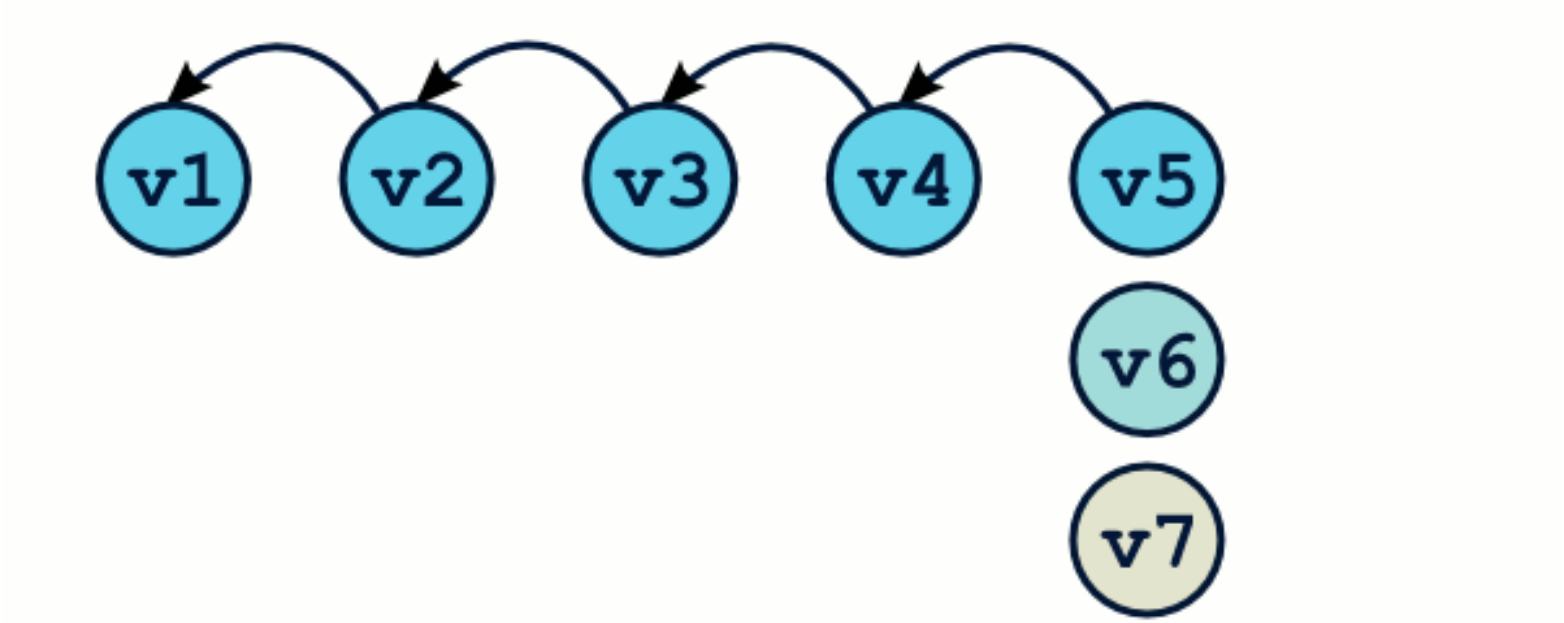
Un arbre de commits

Tout ce pataquès juste pour un commit ? Oui ! Ça peut paraître beaucoup de travail aux habitués de svn (note : ce billet est assez ancien, il a été écrit à une époque ou Git n'était pas encore hégémonique), mais j'affirme que, qu'il travaille seul ou en équipe, un développeur se doit de traiter son historique de projet avec soin, et de s'assurer de la propreté de chaque commit.

Maintenant, que se passe-t-il lorsque les commits s'enchaînent ? En fait, il suffit de comprendre qu'un commit n'est rien d'autre qu'une structure qui contient :

- des méta-données (auteur, date, etc.) ;
- une référence vers un (ou plusieurs) commit parent ;
- une copie de l'arborescence du projet au moment du commit.

Créer un commit, c'est donc rajouter une entrée dans la base de données Git. Et oui, vous avez bien lu : à chaque commit, Git stocke (de manière compressée) l'intégralité des fichiers qui ont été modifiés depuis le commit précédent. Reconstruire le projet à un instant T à partir de l'historique est donc rapide comme l'éclair.



Avec Git, l'historique du projet n'est ni plus ni moins qu'un graphe, qu'on pourra fouiller grâce à la commande *git log*.

```
$ git log
commit 67d6a5214ad4b259407ec7836b9d729f9f7de731
Author: Thibault Jouannic
Date:   Fri Jul 5 10:45:50 2013 +0200

    create reminders admin module

commit 05ca141b9e982c7d04100c37300da4209305b900
Author: Thibault Jouannic
Date:   Fri Jul 5 10:29:56 2013 +0200

    Create user admin module

commit 0fb74654c708a01bfaec8d552437e9f655bd325d
Author: Thibault Jouannic
Date:   Thu Jul 4 15:46:34 2013 +0200

    upgrade pymill version

…
```

Comprendre les branches

Sous d'autres systèmes comme svn, la gestion des branches est souvent pénible et laborieuse, ce qui décourage les développeurs qui ne les utilisent que trés occasionnellement (pour ne pas dire jamais). Avec Git, travailler avec des branches est un tel plaisir qu'on aurait tort de ne pas les utiliser. En fait, les branches sont une fonctionnalité basique, pas un truc « avancé » comme je l'entends parfois.

On utilise les branches tout le temps, ou presque. Pour tester une fonctionnalité ; pour isoler un développement un peu long ; pour mettre quelques commits de côté ; pour développer sans péter la branche principale ; pour corriger un bug sans impacter le développement d'une fonctionnalité parallèle. Bref ! il y a pleins de raisons d'utiliser les branches.

Qu'est-ce qu'une branche ? Attention, accrochez-vous, la définition qui va suivre est difficile à comprendre du premier coup.

Une branche n'est qu'une étiquette qui pointe vers un commit.

Quoi ?! C'est tout ? Et oui ! Si vous ne me croyez pas, tapez la commande suivante à la racine d'un dépôt git :

```
$ cat .git/refs/heads/master
67d6a5214ad4b259407ec7836b9d729f9f7de731
```

La branche master (par défaut la branche principale et seule branche d'un dépôt) n'est qu'une étiquette qui pointe vers un commit. C'est un simple fichier qui ne contient rien d'autre que l'identifiant (un hash SHA1) d'un commit. Dans le jargon Git, cette notion de nom référençant un commit s'appelle une « référence ». Un autre exemple de référence nous est donné par les tags.

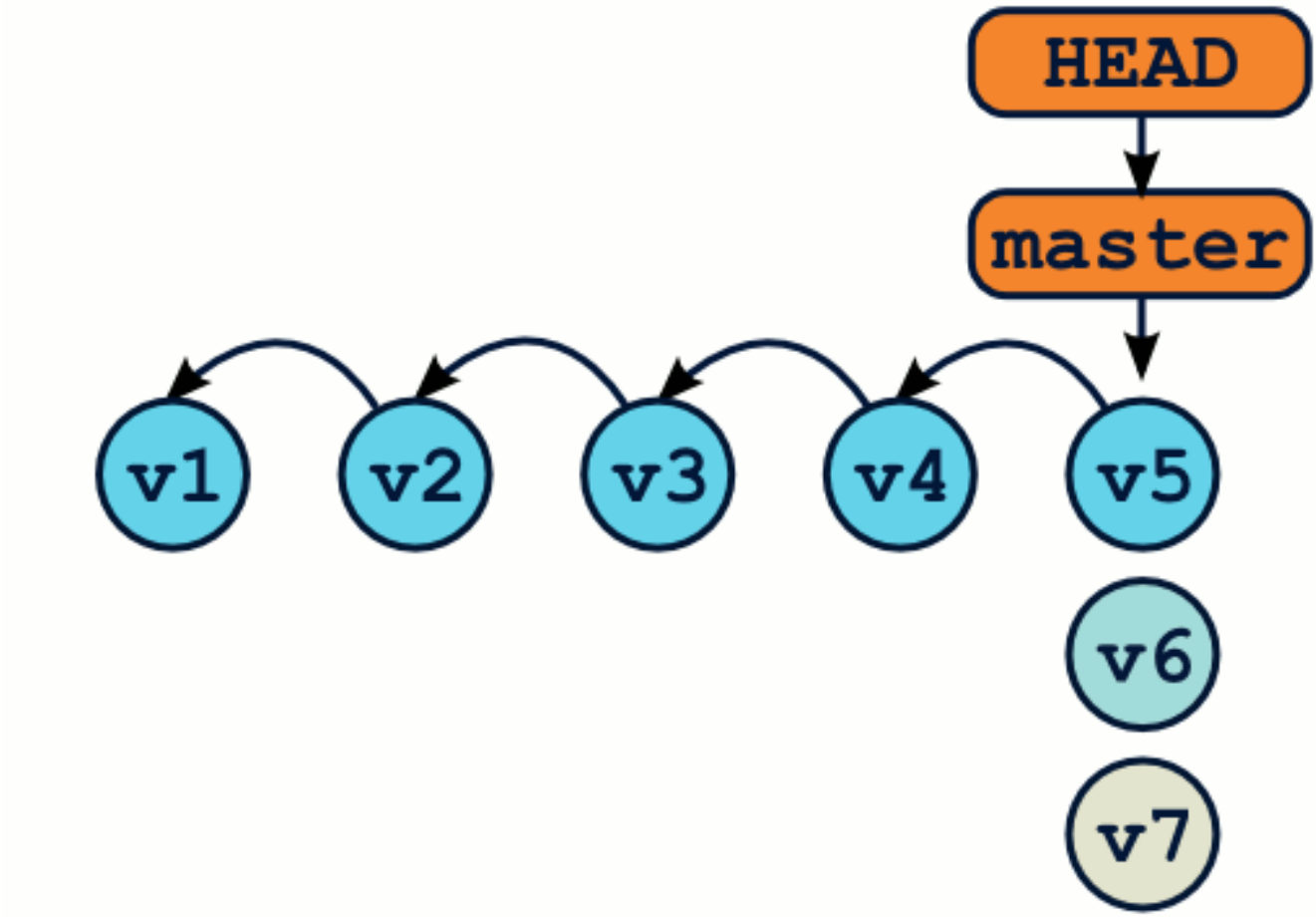
Avec Git, il existe une référence spéciale qui s'appelle *HEAD*. La référence HEAD pointe vers le commit qui sera le parent du prochain commit. C'est clair ? Non ? Vous allez comprendre à la prochaine illustration.

La plupart du temps, HEAD ne pointe pas directement vers un identifiant de commit, mais plutôt vers une branche, e.g *master*.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Quand vous créez un nouveau commit, il se passe ceci :

1. un nouvel objet commit est créé, avec pour parent le commit pointé par HEAD ;
2. la branche pointée par HEAD pointe maintenant vers ce nouveau commit ;
3. et c'est tout.



Manipuler les branches

On va principalement manipuler les branches grâce à deux commandes :

git branch permet de créer, lister et supprimer des branches.

git checkout permet de déplacer la référence HEAD, notamment vers une nouvelle branche.

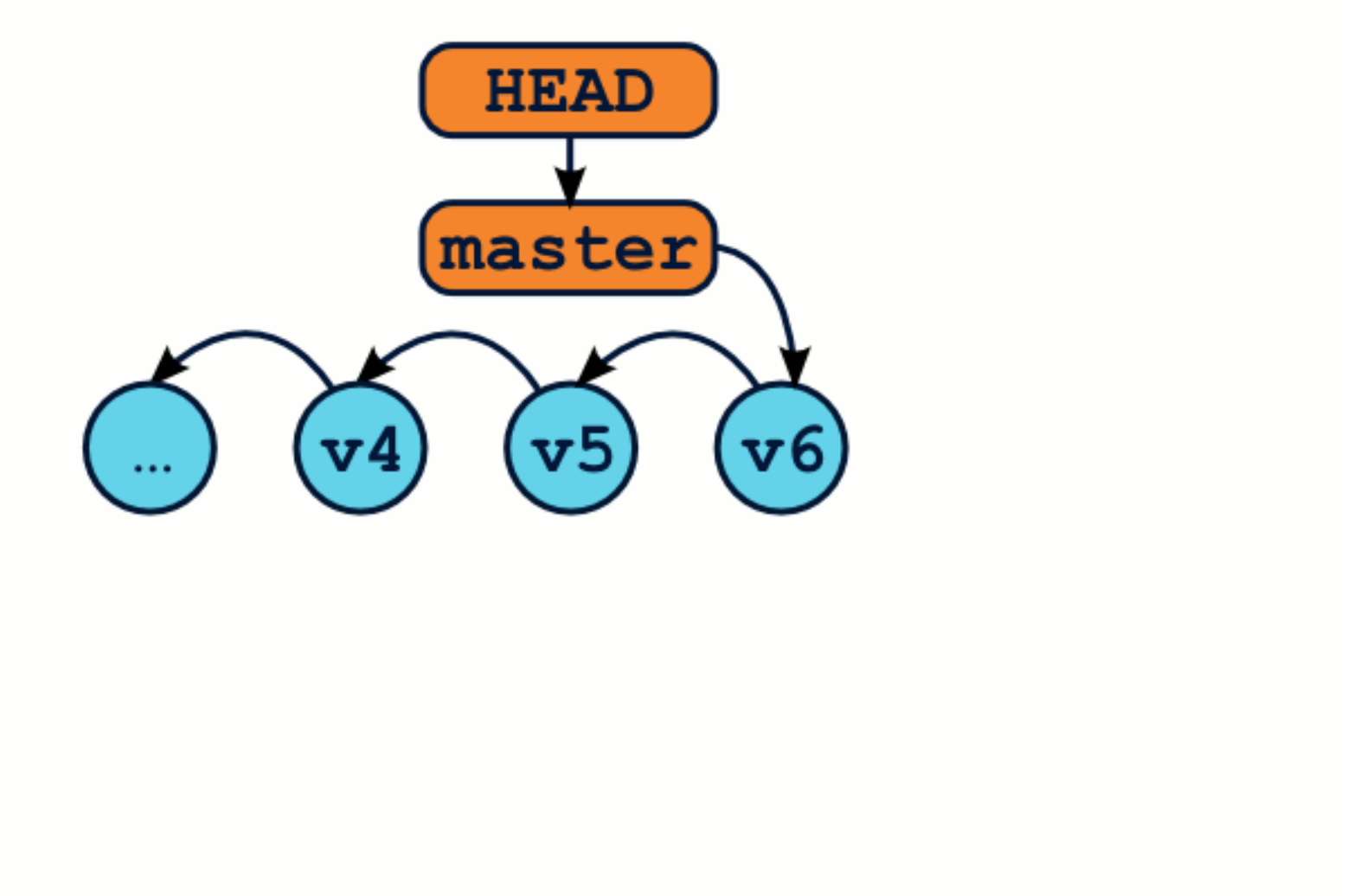
Créer une branche

Voici de manière schématique comment on créé une branche :

```
$ git branch test  # créé la branche "test"
$ git checkout test  # Déplace HEAD sur "test"
```

Notez qu'en général, on utilise le raccourci suivant, qui est trés exactement équivalent aux deux commandes précédentes :

```
$ git checkout -b test
```



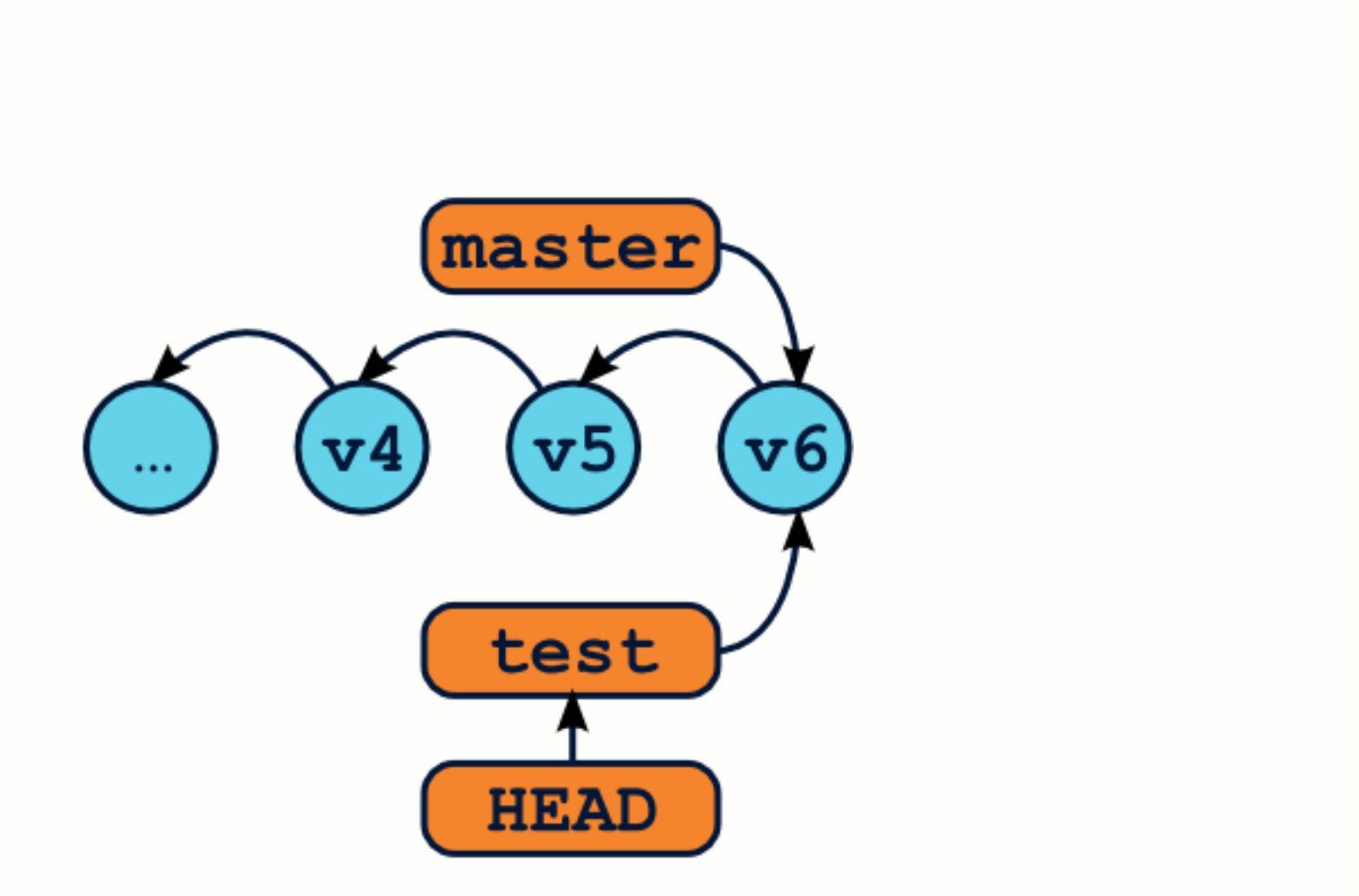
Travailler sur des branches

Créer des branches va nous permettre de créer des flots de développement parallèles. Chaque commit sera ajouté à la chaîne des commits de la branche courante. On connaît la branche courante grâce à la commande *git branch*.

```
$ git branch
* master
  test
```

Voici un workflow de travail classique, ainsi que les commandes associées :

```
$ # Je me trouve actuellement sur la branche "test"
$ git commit …  # Je travaille sur ma branche en créant des commits
$ git checkout master  # Retournons sur la branche master
$ git commit …  # je travaille sur ma branche master
```



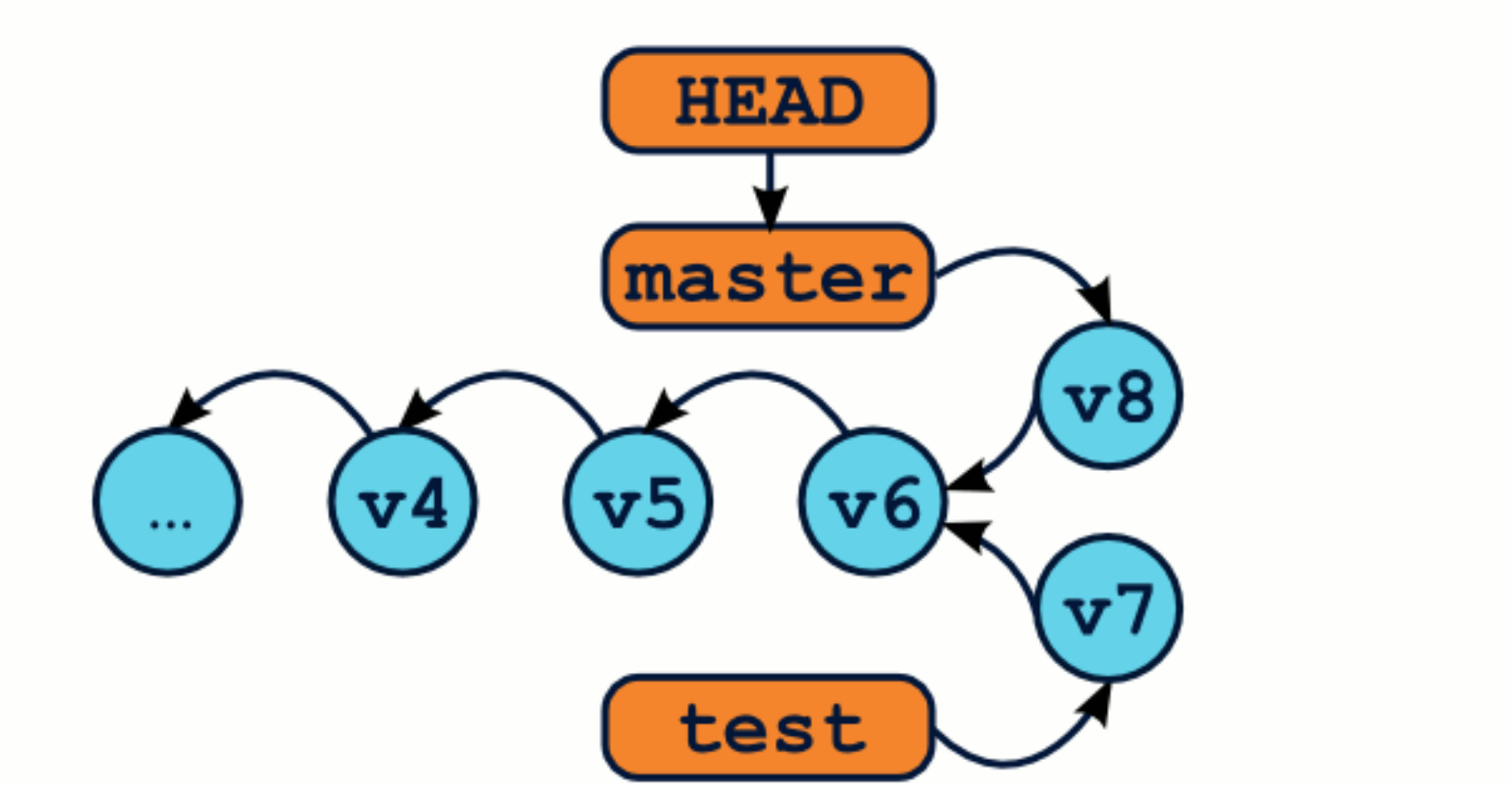
Fusionner des branches

Avoir des branches divergeantes, c'est bien beau, mais il faudra bien fusionner toutes ces modifications dans une seule et même arborescence, n'est-ce pas ? C'est à ça que servent les fusions, ou *merge* dans le jargon.

Lorsqu'on fusionne deux branches, Git va intégrer toutes les modifications contenues sur chaque branche dans une seule et même arborescence. Il va alors créer un commit qui aura deux parents. Une délégation de l'UMP aurait réclamé l'obligation pour chaque commit de

disposer d'un papa et d'une maman, sans effet jusqu'à présent.

```
$ git checkout master  # On se place sur la branche qui va "recevoir" l
$ git merge test  # FuuuuuuuuuuuuSion !
$ git branch -d test  # Une fois fusionnée, notre branche ne sert plus,
```



Résoudre les conflits de fusion

Il peut arriver que la fusion soit impossible à réaliser sans intervention manuelle. Ce sera le cas si chaque branche apporte des modifications différentes au même endroit dans le même fichier. On dit qu'il y a conflit.

Dans ce cas là, Git interrompt le merge et insère des marqueurs dans les fichiers conflictuels. Il nous faudra éditer ces fichiers manuellement (ou à l'aide d'une interface spécifique), avant de poursuivre la fusion.

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified
```

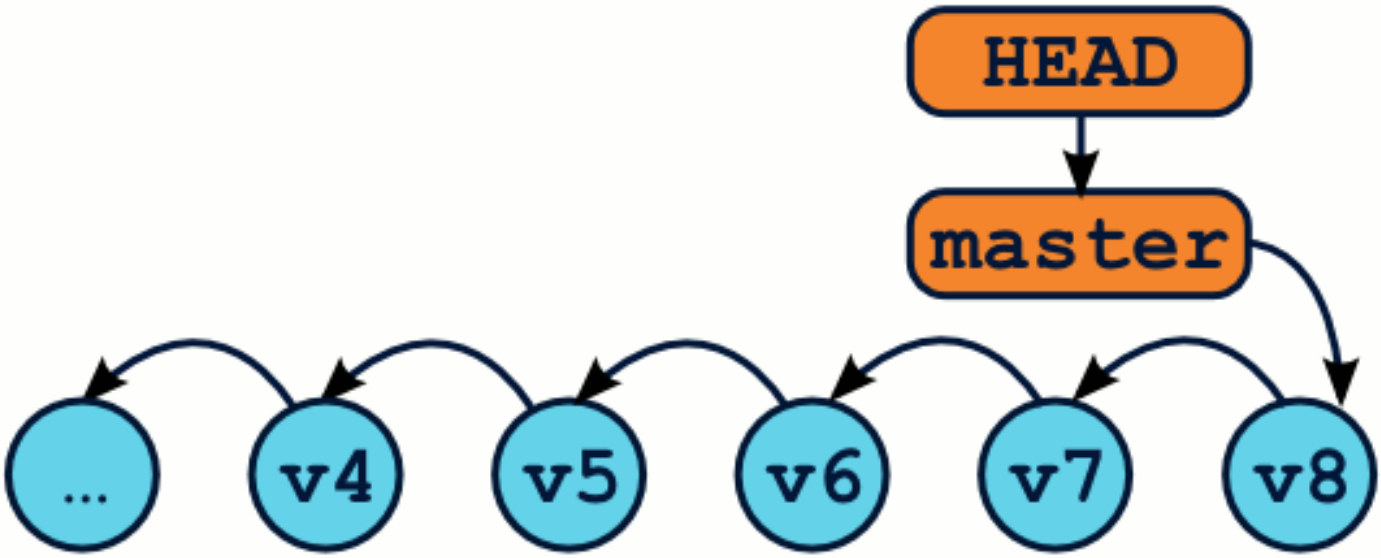
Le protocole précis de résolution est trés bien décrit dans la documentation de la commande *merge*.

```
$ git help merge
```

J'ai des petits problèmes dans ma plantation

Il est trés utile de comprendre que des branches ne sont rien que des étiquettes qui pointent vers des commits. On s'aperçoit alors qu'il est littéralement possible de créer des branches à n'importe quel moment, même depuis un ancien commit.

```
$ git checkout -b test v5
$ Commit… commit… commit…
```

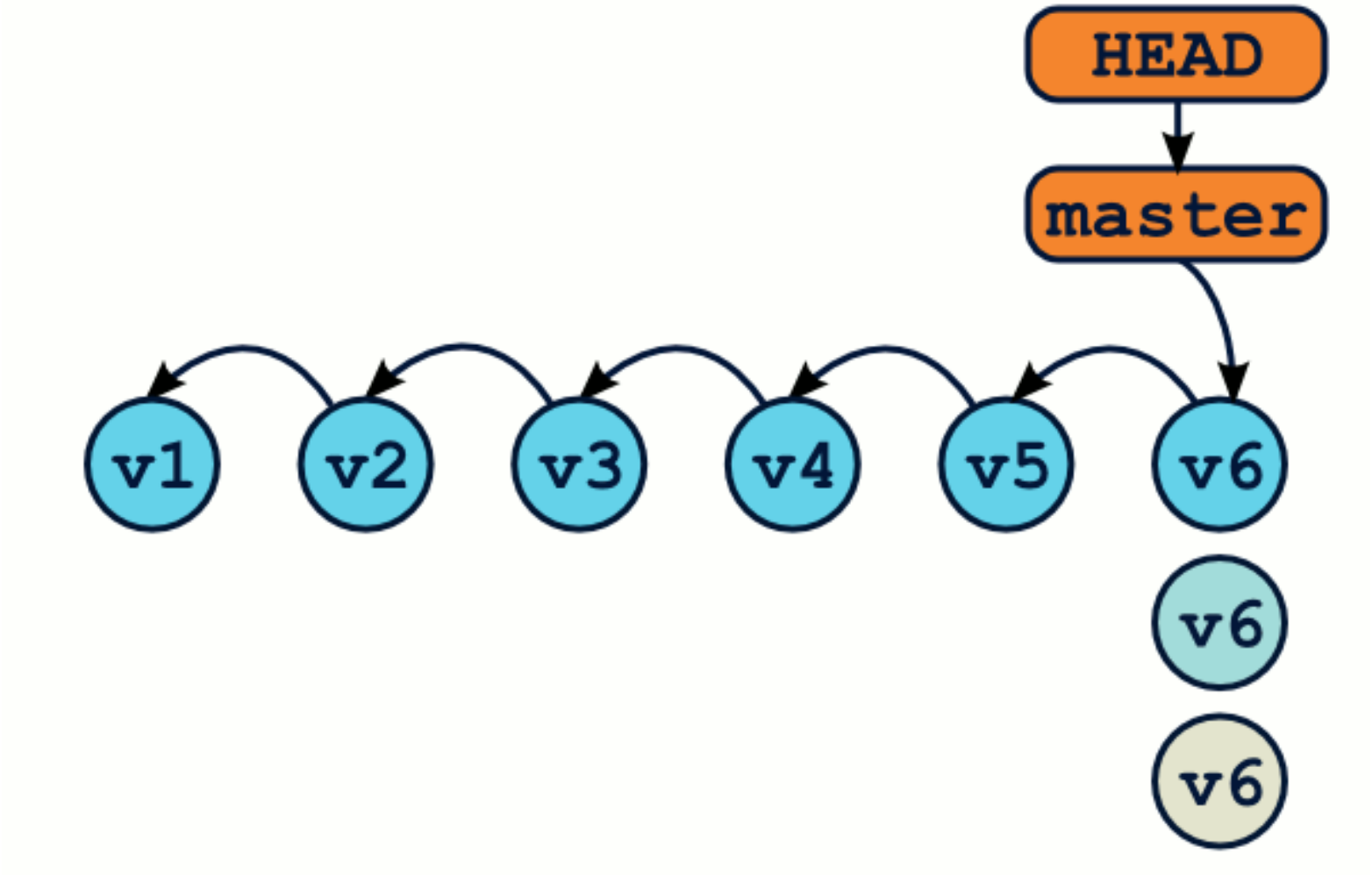


L'état DETACHED HEAD ou *Tête détachée*

Certaines séries de manipulations peuvent parfois laisser votre dépôt Git dans un état dit de *tête détachée*, ce qui est souvent source de confusion. N'y voyons là aucune allusion à un quelconque élément de l'histoire de la révolution française, l'explication est toute autre.

Jusqu'à maintenant, notre référence HEAD a toujours pointé vers une branche, vous vous rappelez ? Ainsi, la commande *git checkout ma_branche* déplace notre HEAD vers la référence « ma_branche ».

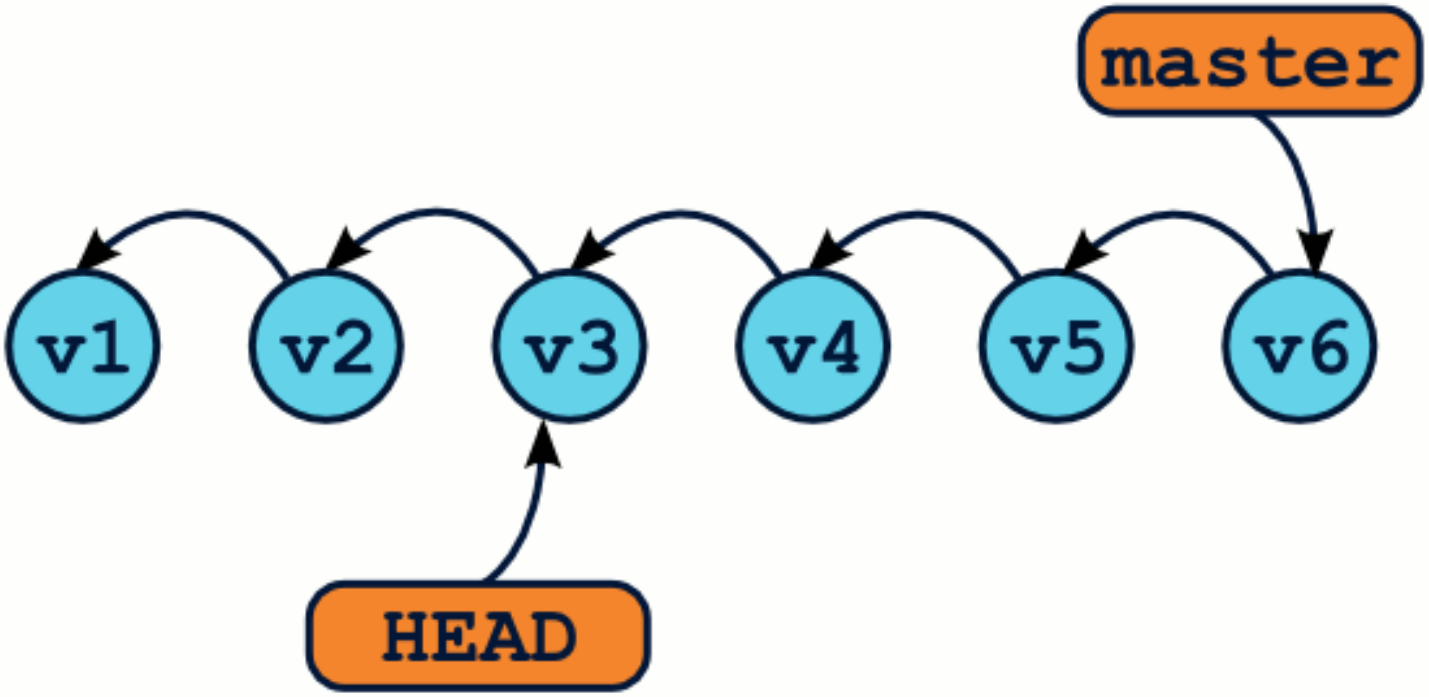
Mais que se passe-t-il si, en lieu et place d'une référence, nous passons directement un identifiant de commit à la commande *git checkout* ?



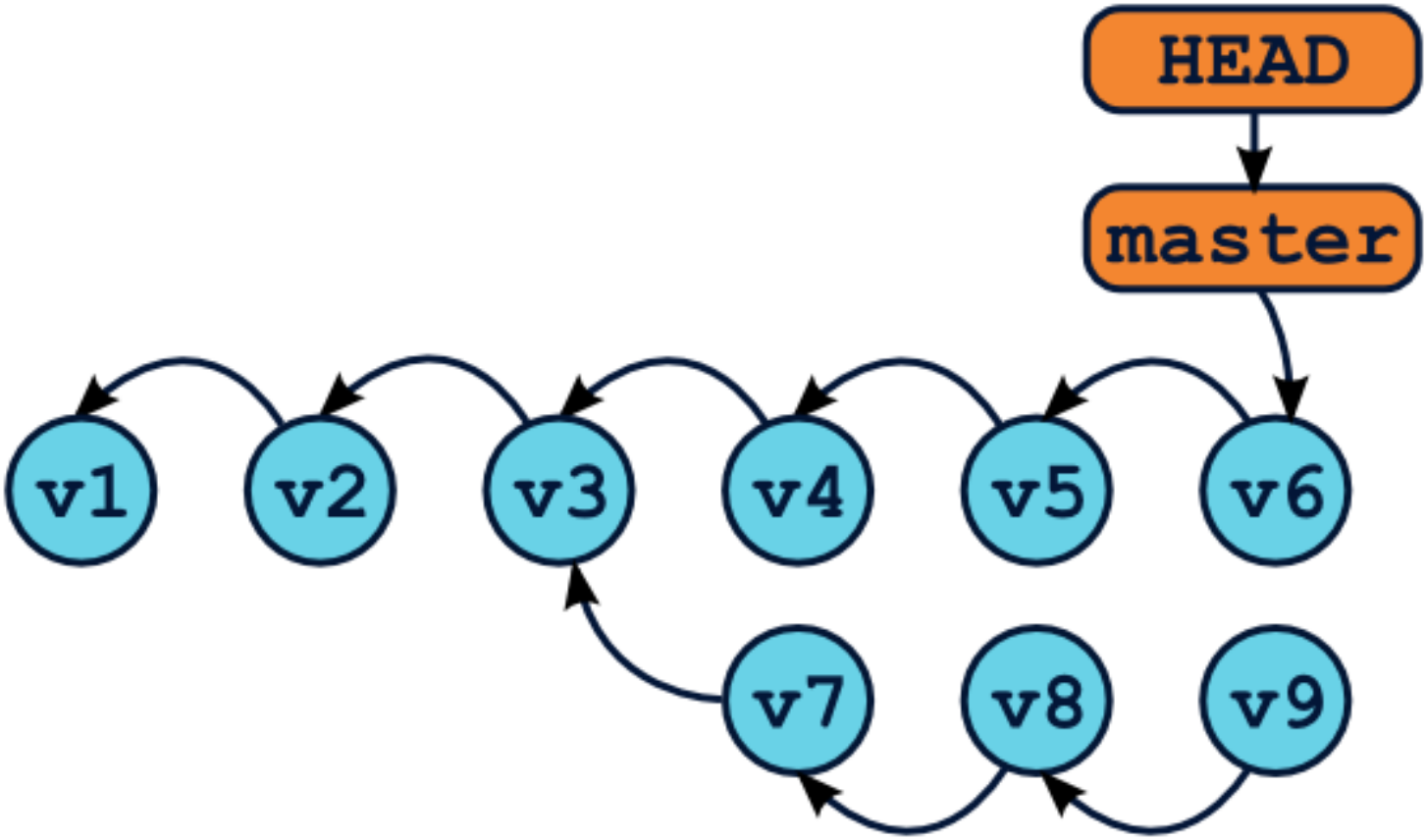
On se retrouve dans ce qu'on appelle l'état *Detached HEAD* : on travaille directement sur un commit, plus sur une branche.

```
$ git branch
* (no branch)
  master
```

L'ajout de nouveaux commits se fera de la manière habituelle.



En revanche, que se passe-t-il si nous utilisons la commande *git checkout* pour retourner sur notre branche *master* ?



OMG ! WTF ! Il semblerait bien que nous ayons perdus des commits ! En effet, la commande *git log* n'affichera que les commits de la branche master (de v1 à v6), mais pas les v7, v8 et v9 ! Et à moins de connaître l'identifiant sha1 exact de ces commits, il semble impossible de les récupérer. Ils ont tout bonnement disparu.

Heureusement, Git dispose d'un outil qui va nous sauver la vie : il s'agit du *reflog*. Git conserve un log de tous les déplacements de la référence HEAD, accessible grâce à la commande *git reflog*.

```
$ git reflog
67d6a52 HEAD@{0}: checkout: moving from 05ca141 to master
05ca141 HEAD@{1}: commit: Create user admin module
0fb7465 HEAD@{2}: commit: upgrade pymill version
8f4c5ba HEAD@{3}: commit: Added log message on new reminder creation
bf15474 HEAD@{4}: checkout: moving from master to bf15474
…
```

Bingo ! La première ligne nous indique la référence du commit juste avant le déplacement de HEAD vers master. Nous pouvons alors créer une branche vers ce commit qui nous permettra de le retrouver plus tard.

```
$ git branch test 05ca141  # Créé une branche, et reste sur master
```

Si récupérer ces commits ne vous intéresse pas, alors laissez les choses en l'état. Les commits qui ne sont accessibles à travers aucune référence sont régulièrement supprimés par le garbage collector de Git.

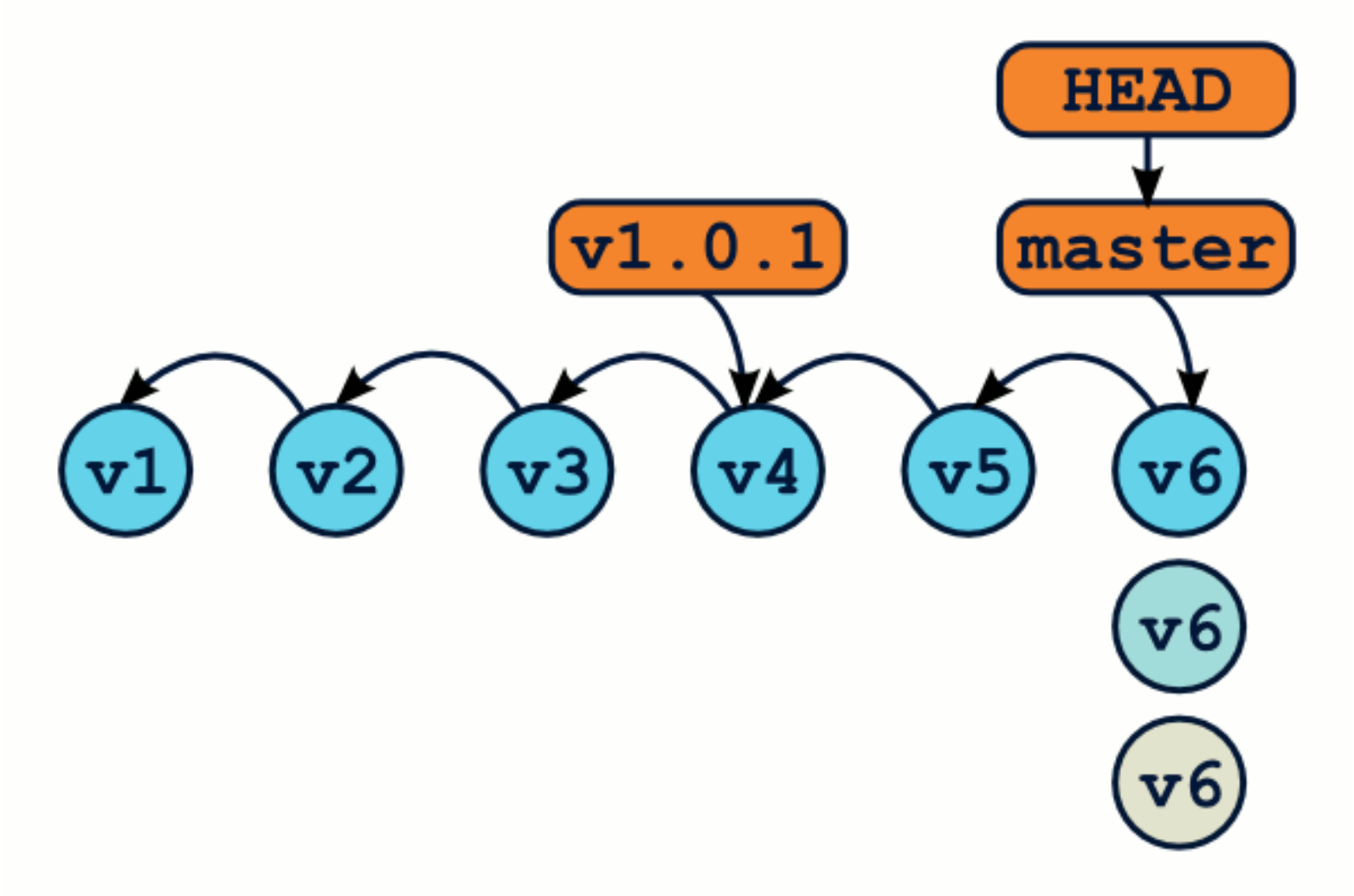
Comprendre cette satanée commande *git checkout*

Si vous avez bien tout suivi, vous avez remarqué que la commande *git checkout* est utilisée à toutes les sauces et semble produire des résultats différents selon les cas.

Il y a en fait deux façons principales d'utiliser la commande *git checkout* :

```
$ git checkout <commit ou branche>  # 1) Avec un identifiant de commit
$ git checkout  <commit ou branche> <repertoire ou fichier >   # 2) Ave
$ git checkout  <repertoire ou fichier> # Équivalent à git checkout HE
$ git checkout  # Équivalent à git checkout HEAD
```

Dans le premier mode d'utilisation (sans spécifier de fichier ou répertoire), Git va simplement déplacer la référence HEAD, et mettre à jour les deux arborescences de la zone de staging et du répertoire de travail. Si vous avez des modifications en cours, elles ne seront pas écrasées, et Git tentera de fusionner la version courante du fichier avec celle correspondant à la nouvelle position de HEAD.



Ainsi, si vous souhaitez récupérer dans votre répertoire de travail votre projet dans l'état ou il était au moment du tag v1.0.1, vous utiliserez la commande *git checkout v1.0.1*, et vous pourrez revenir à l'état précédent grâce à *git checkout master*. On comprend ainsi pourquoi *git checkout* permet de passer d'une branche à l'autre. Par ailleurs, utiliser l'option *-b* permet en plus de créer une nouvelle branche au nouvel emplacement de HEAD.

Le deuxième mode d'utilisation (en spécifiant un fichier ou répertoire) fait grosso-modo la même chose, à quelques exceptions près :

1. la référence HEAD n'est pas déplacée ;

2. la zone de staging n'est pas touchée ;
3. le working directory sera mis à jour à partir de la zone de staging ;
4. l'effet de la commande est limitée au fichier ou répertoire spécifié ;
5. les fichiers en cours de modification seront écrasés purement et simplement.

En gros, la commande *git checkout v1.0.1 accounts* écrase le répertoire *accounts* de votre répertoire de travail avec la version de ce répertoire telle qu'elle était au moment du commit correspondant au tag *v1.0.1*. Clair ?

Voici quelques exemples d'utilisation :

```
$ git checkout .  # Supprime toutes les modifications en cours qui ne s
$ git checkout accounts/  # Supprime toutes les modifications de fichie
$ git checkout v1.0.1 accounts/  # Récupère dans le rép. de travail le
```

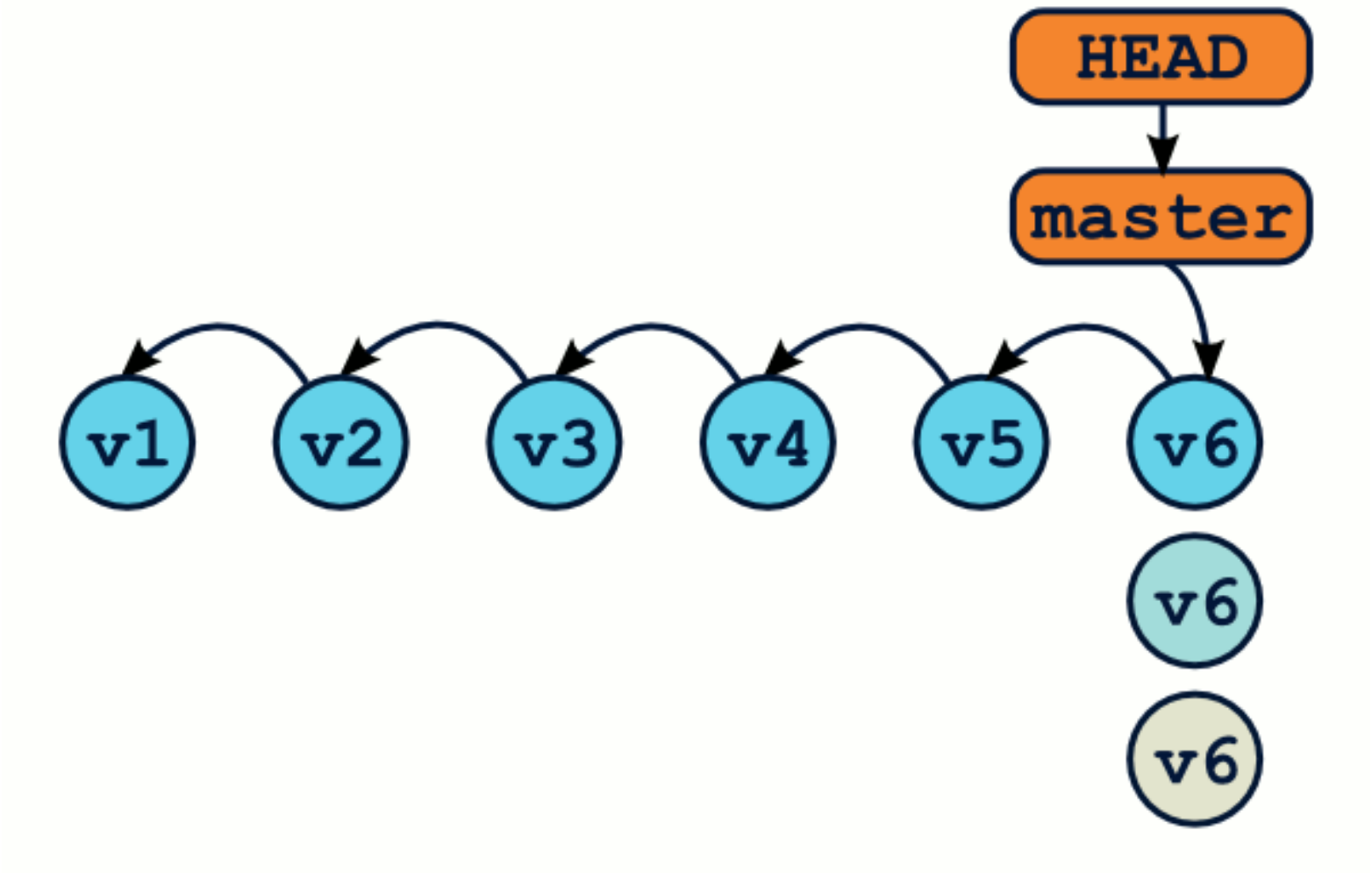
Un peu tordu ? J'en conviens volontiers. Attendez ! attendez ! Il nous reste à nous attaquer à la commende *reset*.

Comprendre la commande *git reset*

Vous avez aimé *git checkout* ? Vous allez adorer *git reset* ! Le principe de cette commande est grossièrement similaire, sauf qu'en plus de déplacer la référence HEAD, *git reset* déplace également la référence de la branche courante. De même, *git reset* dispose de deux modes d'utilisation : avec ou sans spécifier de chemin de fichier.

Par ailleurs, on peut préciser à la commande quelles zones devront être mises à jour après le déplacement de HEAD.

```
$ git reset v1.0.1 --soft  # Déplace HEAD… et c'est tout !
$ git reset v1.0.1 --mixed  # Déplace HEAD, et met à jour le staging. C
$ git reset v1.0.1 --hard  # Déplace HEAD, met à jour le staging et le
```



Attention, contrairement à *git checkout*, *git reset --hard* écrasera votre répertoire de travail même si vous avez des modifications en cours. Il est donc possible de perdre du travail.

Si vous spécifiez un chemin de fichier en plus, alors la commit fonctionnera de manière similaire, à quelques exceptions près :

1. la référence HEAD ne sera pas déplacée ;
2. les modifications seront limitées au chemin de fichier spécifié ;

Quelques exemples d'utilisation :

```
$ git reset  # Équivalent à git reset --mixed HEAD, supprime toutes les
$ git reset --hard  # Supprime toutes les modifications par rapport au
$ git reset --hard v1.0.0  # Supprime tous les commits depuis la v1.0.0
$ git reset HEAD^  # Annule le dernier commit, mais laisse le répertoi
$ git reset accounts/  # Annule les modifications sous accounts/ qui so
$ git reset --hard accounts/  # Cette combinaison d'option est interdit
```

C'est plus clair ?

Des questions ?

Bon, j'espère que ces quelques éclaircissements vous auront permis d'appréhender Git d'une manière moins empirique. Il parait aussi que je suis capable de donner de chouettes formations sur Git, alors si vous…

- …aimeriez mieux comprendre Git,
- …souhaitez convertir votre entreprise à Git,
- …galérez pendant cette conversion,

n'hésitez pas, contactez-moi. Et si vous avez d'autres questions, n'hésitez pas à les poser, je me ferai (peut-être) un plaisir d'y répondre.