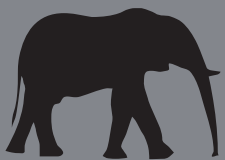
Git à 100%

mémento

Raphaël Hertzog
Pierre Habouzit



ISBN / 978-2-212-13438-4

EYROLLES

Conception : Nord Compo

Cohérence et taille des commits : enregistrez des changements cohérents et minimaux
Un commit doit contenir des **changements cohérents, qui vont ensemble**. Ainsi les corrections de deux bogues différents doivent en règle générale se retrouver dans deux commits différents. De plus, les petits commits sont bien plus **faciles à relire** pour les autres développeurs. En cas de problème, cela permet aussi de retrouver plus facilement le changement fautif (grâce à `git bisect` notamment).
Si des changements hétérogènes sont mélangés dans votre répertoire de travail, enregistrez-les par petits bouts :
`git add -p`, `git checkout -p`, `git rebase -1`, `git reset -p`.

Rythme des commits : faites des commits fréquents et prévenez les conflits
Cela permet d'enregistrer des changements cohérents et de petite taille, et de partager votre code plus souvent avec les autres développeurs.
En intégrant vos développements en cours plus régulièrement, vous prévenez les conflits importants qui arrivent souvent en cas d'intégration tardive d'une branche. Avoir peu de commits contenant beaucoup de modifications, et ne les partager que rarement, génère des conflits plus gros et plus difficiles à résoudre.
Enfin, avant de partager votre code, utilisez si nécessaire `git rebase -i` pour réordonner, reformuler et réagréger des *commits* afin de présenter un historique lisible aux autres programmeurs, dans lequel chaque *commit* représente un ensemble minimal et cohérent de modifications clairement documentées.

N'enregistrez pas du travail à moitié fini : utilisez `git stash` *nom*
et `git stash apply`
Un *commit* doit contenir un travail « complet ». Ceci n'est pas en opposition avec le fait qu'il faut faire des commits de petite taille : il ne s'agit pas de faire en un seul commit une fonctionnalité compliquée, mais de découper la fonctionnalité en plusieurs petites tâches, et d'enregistrer un changement par petite tâche.
Ne faites pas de commit le soir avant de quitter le bureau juste pour avoir un répertoire de travail propre. Dans ce genre de cas, recourez à `git stash`, car les commits temporaires ont la fâcheuse tendance à se retrouver dans les dépôts centraux à cause de `git push` trop hâtifs par exemple…

Documentation : écrivez de bons messages de commit
La gestion de versions perd son sens si les messages de commit sont de mauvaise qualité. Les messages s'adressent tant aux autres qu'à vous-même, ils seront lus par toutes les personnes qui vont ausculter l'historique et se demander invariablement « Pourquoi ce changement ? Qu'est-ce qui a changé ? » plutôt que « Comment fonctionne le code ? » .
Un bon message de commit commence par un résumé d'une cinquantaine de caractères qui décrit le changement. Suit une ligne vide qui le sépare du message détaillé, qui sera aussi long que nécessaire. Un bon message précise donc : 1) ce qui a **motivé** le changement (bogue, nouvelle fonctionnalité, nettoyage, optimisation…) ; 2) ce qui a **changé** par rapport à la version précédente.
Ce n'est pas le lieu pour décrire l'implémentation en détail, elle doit l'être dans le code sous forme de commentaires.

N'enregistrez jamais de changements non fonctionnels : testez avant de commiter !
Avant de publier un commit, il est critique de vérifier que le code en est correct (tests, règles de programmation, etc.), d'autant plus s'il s'agit de séries de changements remises en forme en utilisant `git rebase -i` car une mauvaise manipulation au cours de cette opération peut introduire des régressions (même une régression uniquement présente sur des commits intermédiaires peut nuire en rendant l'usage de `git bisect` plus difficile).
En d'autres termes, testez vos changements, l'un après l'autre. Ne publiez pas des changements mal testés, vous risqueriez de faire perdre un temps précieux aux autres programmeurs et de rendre des commandes comme `git bisect` totalement inutiles.

HÉBERGEMENTS **Git en ligne : github, gitorious…**
Des services tels **GitHub.com** et **Gitorious.com** proposent d'héberger vos dépôts Git en ligne. Ils sont généralement gratuits pour les projets de logiciel libre. Chaque dépôt est avant tout associé à un utilisateur, et il n'est pas rare d'avoir plusieurs centaines de copies d'un dépôt pour un logiciel populaire. C'est la conséquence naturelle du modèle de développement associé : toute contribution externe est réalisée dans une copie du dépôt (`fork`) sur une branche (publique) que l'on peut ensuite soumettre pour intégration (*via* une `pull request`).
La création de dépôts s'effectue par l'interface web mais ils sont ensuite gérés directement *via* git et un dépôt distant accessible par SSH (l'utilisateur doit donc enregistrer sa clé publique SSH auprès de ces services).

Bonnes pratiques

N'hésitez pas à user et abuser des branches
Git a dès l'abord été conçu pour faciliter la création de branches et surtout leur fusion. Des changements peuvent ainsi rester totalement indépendants ; cela ouvre de nombreuses perspectives de travail :
- branches de correctifs **intégrables dans plusieurs versions** du logiciel ;
- **expérimentations sans risque** (il est facile de « jeter » une branche, bien moins de se débarrasser de commits entrelacés avec le développement « normal ») ;
- **branches par fonctionnalités**, pour que les autres développeurs puissent facilement tester un sous-ensemble de vos changements et vous faire des retours dessus ;
- **branches temporaires de travail collaboratif** sur une fonctionnalité avec un autre développeur avant la soumission finale, etc.

Choisissez des règles de travail et conventions communes : adoptez le même workflow
La souplesse de Git (par opposition à des systèmes de gestion de versions centralisés comme CVS, Subversion, perforce) est l'une de ses forces... mais elle peut devenir un cauchemar si tous les développeurs ne travaillent pas de la même manière. Établissez donc quelques règles dans la façon de travailler :
- utilisation d'un **dépôt central** ;
- nommage des différentes branches « officielles » ;
- règles pour soumettre les *patches* (directement, par soumission sur une liste de diffusion, *via* des outils de relecture…), etc.

Choisissez des outils qui s'intègrent bien avec Git
Le système de gestion de versions est la clef de voûte d'une bonne gestion de projet ; préférez ceux des autres outils classiques (gestion de tickets, relecture de code, intégration continue, etc.) qui s'intègrent bien avec Git, pour automatiser des tâches répétitives, sources d'erreurs multiples. Choisissez par exemple un système de gestion de tickets qui « comprenne » Git pour « fermer » les tickets depuis les messages de commit afin d'éviter des oublis humains. Utilisez un outil de relecture de code sachant manipuler directement un dépôt Git, pour éviter des erreurs lors de l'intégration d'une branche.

POUR ALLER PLUS LOIN **Outils de l'écosystème Git**

- **Interfaces graphiques**
  Pour Windows, **Git Extensions** s'intègre dans l'explorateur :
  http://sourceforge.net/projects/gitextensions/
  Pour Mac OS X, **GitX** est une application indépendante :
  http://gitx.laullon.com/
- **Gestion de patches**
  StGit gère les patches *à la* quilt dans Git : **http://www.procode.org/stgit/**
  TopGit gère les *topic* branches liées entre elles : **http://repo.or.cz/w/topgit.git**
- **Hébergement et contrôle d'accès à des dépôts Git**
  Gitorious : http://gitorious.org/gitorious/mainline
  Gitolite : http://github.com/sitaramc/gitolite
- **Revue de code**
  Gerrit : http://code.google.com/p/gerrit/

Code éditeur : G13438
ISBN : 978-2-212-13438-4
Conception : Nord Compo

9,90 €