Annexe 1

Créer un dépôt

git init / **git clone** / **git config**

Ce tutoriel donne un aperçu de la procédure à suivre pour configurer un dépôt avec le contrôle de version Git. Il vous explique comment initialiser un dépôt Git pour un projet (nouveau ou existant). Vous trouverez ci-dessous des exemples de workflow pour des dépôts créés en local et clonés à partir de dépôts distants. Ce guide part du principe que vous maîtrisez les bases de l'interface de ligne de commande.

Ce guide couvre les grands points suivants :

- Initialiser un nouveau dépôt Git
- Cloner un dépôt Git existant
- Commiter une version modifiée d'un fichier dans le dépôt
- Configurer un dépôt Git pour la collaboration à distance
- Commandes de contrôle de version Git courantes

À l'issue de ce module, vous serez en mesure de créer un dépôt Git, d'utiliser des commandes Git courantes, de commiter un fichier modifié, d'afficher l'historique de votre projet et de configurer une connexion avec un service d'hébergement Git (Bitbucket).

Qu'est-ce qu'un dépôt Git ?

Un dépôt Git est un entrepôt virtuel de votre projet. Il vous permet d'enregistrer les versions de votre code et d'y accéder au besoin.

Initialisation d'un nouveau dépôt : git init

Pour créer un nouveau dépôt, utilisez la commande `git init`. `git init` est une commande unique que vous utilisez durant la configuration initiale du nouveau dépôt. Lorsque vous exécutez cette commande, un nouveau sous-répertoire `.git` est créé dans votre répertoire de travail actuel. De même, une branche principale est créée.

Contrôle de version d'un projet existant avec un nouveau dépôt Git

Cet exemple part du principe que vous disposez déjà d'un dossier de projet et que vous aimeriez créer un dépôt dans celui-ci. Lancez d'abord `cd` dans le dossier de projet racine, puis exécutez la commande `git init`.

```
cd /path/to/your/existing/code git init
```

Lorsque vous pointez `git init` vers un répertoire de projet existant, l'étape d'initialisation présentée ci-dessus est également exécutée. Elle est toutefois circonscrite à ce répertoire de projet.

```
git init
```

Pour de plus amples informations sur git init, reportez-vous à la page `git init`.

Cloner un dépôt existant : git clone

Si un projet est déjà configuré dans un dépôt centralisé, la commande clone est la plus courante pour obtenir un clone de développement local. À l'instar de `git init`, le clonage est généralement une opération ponctuelle. Lorsqu'un développeur obtient une copie de travail, toutes les opérations de contrôle de version sont gérées par l'intermédiaire de son dépôt local.

git clone

`git clone` permet de créer une copie ou un clone de dépôts distants. Vous transmettez une URL de dépôt à `git clone`. Git prend en charge quelques protocoles réseau différents et les formats d'URL correspondants. Dans cet exemple, nous utiliserons le protocole SSH de Git. Les URL SSH de Git suivent ce modèle : `git@NOM_HOTE:NOM_UTILISATEUR/NOM_DEPOT.git`

Exemple d'URL SSH de Git : `git@bitbucket.org:rhyolight/javascript-data-store.git` où les valeurs de modèle désignent :

- `NOM_HOTE: bitbucket.org`

- `USERNAME: rhyolight`
- `NOM_DEPOT: javascript-data-store`

Cette commande récupère la dernière version des fichiers du dépôt distant sur la branche master et les ajoute à un nouveau dossier. Le nom du nouveau dossier correspond à celui du dépôt, dans ce cas `javascript-data-store`. Le dossier contient tout l'historique du dépôt distant ainsi qu'une nouvelle branche master.

Pour de plus amples informations sur l'utilisation de `git clone` et les formats d'URL Git pris en charge, reportez-vous à la pqge git clone.

Enregistrer des changements dans le dépôt : git add et git commit

Maintenant que vous avez cloné ou initialisé un dépôt, vous pouvez commiter les changements de version de fichier dans celui-ci. L'exemple suivant part du principe que vous avez configuré un projet dans `/path/to/project`. Dans cet exemple, les étapes à effectuer sont les suivantes :

- Redéfinissez les répertoires sur `/path/to/project`
- Créez un nouveau fichier `CommitTest.txt` avec pour contenu ~"contenu test pour le tutoriel Git"~

- Ajoutez le fichier `CommitTest.txt` à la zone de staging du dépôt avec la commande git add
- Créez un nouveau commit ; son message doit décrire le travail effectué dans le commit

```
cd /path/to/project echo "Contenu test pour le tutoriel Git" >> CommitTest.txt git add CommitTest.txt git commit -m "CommitTest.txt ajouté au dépôt"
```

Après avoir exécuté cet exemple, le fichier `CommitTest.txt` est désormais ajouté à l'historique de votre dépôt, et ce dernier suit les futures mises à jour apportées au fichier.

Cet exemple a introduit deux commandes Git supplémentaires : `add` et `commit`. Cet exemple est très restreint, mais les deux commandes sont abordées plus en détail sur les pages git add et git commit. Autre cas d'usage courant de `git add` : l'option `--all`. Lorsque vous exécutez `git add --all`, tous les fichiers modifiés et non trackés du dépôt sont ajoutés, et l'arborescence de travail du dépôt est mise à jour.

Collaboration dépôt à dépôt : git push

Il est essentiel de comprendre que le concept de « copie de travail » de Git est très différent de la copie de travail que vous obtenez en faisant un check-out du code source à partir d'un dépôt SVN. Contrairement à SVN,

Git ne fait aucune distinction entre les copies de travail et le dépôt centralisé. Ce sont des dépôts Git à part entière.

Cela rend la collaboration avec Git fondamentalement différente d'avec SVN. Alors que SVN dépend de la relation entre le dépôt centralisé et la copie de travail, le modèle de collaboration de Git repose sur l'interaction de dépôt à dépôt. Au lieu de checker une copie de travail dans le dépôt centralisé de SVN, vous pouvez faire un push ou un pull des commits d'un dépôt à un autre.

Bien entendu, rien ne vous empêche de donner un sens spécial à certains dépôts Git. Par exemple, en désignant simplement un dépôt Git comme le dépôt « centralisé », il est possible de répliquer un workflow centralisé avec Git. Cette opération s'effectue selon des conventions et non par raccordement câblé au logiciel de contrôle de version lui-même.

Comparaison des dépôts bruts et clonés

Si vous avez utilisé `git clone` dans la section « Initialisation d'un nouveau dépôt » précédente pour configurer votre dépôt local, ce dernier est déjà défini pour la collaboration à distance. `git clone` configure automatiquement votre dépôt avec une branche remote qui pointe vers l'URL Git à partir de laquelle vous l'avez cloné. Cela signifie qu'une fois les changements apportés à un fichier commités, vous pouvez lancer une commande `git push` pour pusher ces changements vers le dépôt distant.

Si vous avez utilisé la commande `git init` pour créer un nouveau dépôt, vous ne disposerez pas de dépôt distant vers lequel pusher vos

changements. Lors de l'initialisation d'un nouveau dépôt, un modèle courant consiste à accéder à un service d'hébergement Git comme Bitbucket pour y créer un dépôt. Le service fournit une URL Git que vous pouvez ensuite ajouter à votre dépôt Git local avant de faire un `git push` vers le dépôt hébergé. Après avoir créé un dépôt distant auprès du service de votre choix, vous devrez mettre à jour votre dépôt local par le biais d'un mappage. Nous aborderons ce processus dans le guide de configuration et d'installation ci-dessous.

Si vous préférez héberger votre propre dépôt distant, vous devrez configurer un dépôt brut. Les commandes `git init` et `git clone` prennent toutes deux en charge l'argument `--bare`. Le cas d'usage le plus courant pour les dépôts bruts consiste à créer un dépôt Git distant centralisé.

Configuration et installation : git config

Une fois le dépôt distant configuré, vous devrez ajouter une URL de dépôt distant à votre commande `git config` locale et définir une branche upstream pour vos branches locales. La commande `git remote` vous offre un tel utilitaire.

```
git remote add
```

Cette commande mappe le répertoire distant qui se trouve dans vers une réf de votre dépôt local disponible sous . Après avoir mappé le dépôt distant, vous pouvez pusher les branches locales vers celui-ci.

```
git push -u
```

Cette commande pushe la branche du dépôt local sous vers le dépôt distant qui se trouve à .

Pour plus de détails sur la commande `git remote`, reportez-vous à la page `git remote`.

Outre configurer une URL de dépôt local, vous devrez peut-être aussi définir des options de configuration Git globales comme username ou email. La commande `git config` vous permet de configurer votre installation Git (ou un dépôt individuel) à partir de la ligne de commande. Elle est capable de définir les informations relatives à l'utilisateur, les préférences ou encore le comportement d'un dépôt. Vous trouverez plusieurs options de configuration courantes ci-dessous.

Git enregistre les options de configuration dans trois fichiers distincts, ce qui vous permet de définir l'étendue des options sur des dépôts individuels (local), des utilisateurs (global) ou le système global (système) :

- Local : `/.git/config` – paramètres spécifiques du dépôt.
- Global : `/.gitconfig` – paramètres spécifiques de l'utilisateur. Les options contenant le flag --global sont stockées à cet endroit.
- Système : `$(prefix)/etc/gitconfig` – paramètres à l'échelle du système.

Définissez le nom d'auteur à utiliser pour tous les commits dans le dépôt actuel. Typiquement, vous utiliserez le flag `--global` afin de définir les options de configuration pour l'utilisateur actuel.

```
git config --global user.name
```

Définissez le nom de l'auteur à utiliser pour tous les commits par l'utilisateur courant.

Lorsque vous ajoutez l'option `--local` ou que vous ne transmettez pas d'option de niveau de configuration, `user.name` sera défini pour le dépôt local courant.

```
git config --local user.email
```

Définissez l'adresse e-mail de l'auteur à utiliser pour tous les commits par l'utilisateur courant.

```
git config --global alias.
```

Créez un raccourci pour une commande Git. Cet utilitaire très performant permet de créer des raccourcis personnalisés pour des commandes Git fréquemment utilisées. Exemple simpliste :

```
git config --global alias.ci commit
```

Cela crée une commande `ci` que vous pouvez exécuter comme un raccourci de la commande `git commit`. Pour en savoir plus sur les alias Git, reportez-vous à la page `git config`.

```
git config --system core.editor <éditeur>
```

Cette commande définit l'éditeur de texte utilisé par des commandes comme `git commit` pour tous les utilisateurs de la machine actuelle. L'argument devrait être la commande qui lance l'éditeur souhaité (par ex., vi). Cet exemple présente l'option `--system`. L'option `--system` définit la configuration de tout le système, c'est-à-dire tous les utilisateurs et les dépôts sur une machine. Pour de plus amples informations sur les niveaux de configuration, consultez la page git config.

```
git config --global --edit
```

Ouvrez le fichier de configuration globale dans un éditeur de texte pour le modifier manuellement. Vous trouverez un guide détaillé qui vous explique la procédure à suivre pour configurer un éditeur de texte pour Git sur la page git config.

Discussion

Toutes les options de configuration sont stockées dans des fichiers en texte brut. La commande `git config` n'est donc qu'une interface de ligne de commande pratique. Généralement, vous devez uniquement configurer une installation Git quand vous commencez à travailler sur une nouvelle machine de développement. Dans la plupart des cas, vous utiliserez le flag `--global`. Exception importante : lorsque vous écrasez l'adresse e-mail de l'auteur. Vous souhaiterez peut-être définir votre adresse e-mail personnelle pour les dépôts source personnels et ouverts et votre adresse e-mail professionnelle pour les dépôts professionnels.

Git enregistre les options de configuration dans trois dossiers distincts, ce qui vous permet de définir l'étendue des options sur des dépôts individuels, des utilisateurs ou le système global :

- `/.git/config` – Paramètres spécifiques du dépôt.
- `~/.gitconfig` – paramètres spécifiques de l'utilisateur. Les options contenant le flag --global sont stockées à cet endroit.
- `$(prefix)/etc/gitconfig` – Paramètres à l'échelle du système.

Si les options de ces fichiers entrent en conflit, les paramètres locaux remplacent les paramètres utilisateur (s'appliquent à l'échelle du système). Si vous ouvrez l'un de ces fichiers, vous obtiendrez quelque chose de semblable à ce qui suit :

```
[user] name = John Smith email = john@example.com [alias] st = status co = checkout br = branch up = rebase ci = commit [core] editor = vim
```

Vous pouvez modifier manuellement ces valeurs de sorte qu'elles aient le même effet que `git config`.

Exemple

La première chose que vous devez faire après avoir installé Git est lui indiquer votre nom/e-mail et personnaliser certains des paramètres par défaut. Une configuration initiale type peut ressembler à ceci :

Dites à Git qui vous êtes avec `git config`

```
git --global user.name "John Smith" git config --global user.email john@example.com
```

Sélectionnez votre éditeur de texte préféré

```
git config --global core.editor vim
```

Ajoutez des alias de type SVN

```
git config --global alias.st status git config --global alias.co checkout git config --global alias.br branch git config --global alias.up rebase git config --global alias.ci commit
```

Cette commande génère le fichier `~ /.gitconfig` de la section précédente. Pour plus de détails sur la commande git config, reportez-vous à la page git config.

Enregistrer des changements

git add git commit git diff Git Stash .gitignore

Lorsque vous travaillez dans Git ou dans un autre système de contrôle de version, le concept d'« enregistrement » est un processus plus nuancé que dans un outil de traitement de texte ou d'autres applications d'édition de fichiers traditionnelles. Dans l'univers des logiciels, l'expression « enregistrer » classique est synonyme de « commiter » dans Git. Un commit est l'équivalent Git d'un enregistrement. Ce dernier devrait être considéré comme une opération sur un système de fichiers, qui est utilisée pour écraser un fichier existant ou en créer un. Ou, le commit Git est une opération qui agit sur un ensemble de fichiers et de répertoires.

Dans Git, l'enregistrement de changements diffère par rapport à SVN. Les commits ou « check-ins » SVN sont des opérations qui effectuent un push distant vers un serveur centralisé. Cela signifie qu'un commit SVN a besoin d'un accès à Internet pour enregistrer complètement les changements apportés au projet. Les commits Git peuvent être capturés et accumulés en local, avant d'être pushés vers un serveur distant au besoin à l'aide de la commande `git push -u origin master`. La différence entre les deux méthodes ? Une différence fondamentale dans les conceptions d'architecture. Git est un modèle d'application distribuée, tandis que SVN est un modèle centralisé. Les applications distribuées sont généralement plus solides, car elles ne disposent pas de point de défaillance unique comme un serveur centralisé.

Les commandes `git add`, git status et git commit sont toutes utilisées en combinaison pour enregistrer un instantané de l'état actuel d'un projet Git.

Git dispose d'un mécanisme d'enregistrement supplémentaire appelé le « stash ». Le stash est une zone de stockage temporaire pour les changements qui ne sont pas prêts à être commités. Il fonctionne sur le répertoire de travail, la première des trois arborescences, et possède des options d'utilisation étendues. Pour en savoir plus, consultez la page git stash.

Un dépôt Git peut être configuré de sorte à ignorer des fichiers ou des répertoires spécifiques. Cela empêche ainsi Git d'enregistrer des changements apportés à du contenu ignoré. Git propose plusieurs méthodes de configuration qui gèrent la liste « ignore ». La configuration « ignore » de Git est abordée plus en détail sur la page git ignore.

git add

La commande `git add` ajoute un changement dans le répertoire de travail à la zone de staging. Elle informe Git que vous voulez inclure les mises à jour dans un fichier particulier du commit suivant. Cependant, `git add` n'impacte pas le dépôt de manière significative. Les changements ne sont pas réellement enregistrés jusqu'à ce que vous exécutiez git commit.

Conjointement à ces commandes, vous avez également besoin de git status pour voir l'état du répertoire de travail et de la zone de staging.

Fonctionnement

Les commandes `git add` et git commit composent le workflow Git de base. Ce sont les deux commandes que chaque utilisateur de Git doit comprendre, quel que soit le modèle de collaboration de son équipe. Elles permettent d'enregistrer les versions d'un projet dans l'historique du dépôt.

Le développement d'un projet repose sur le processus de base qui s'articule autour de trois étapes : l'édition, le staging et les commits. En premier lieu, vous modifiez vos fichiers dans le répertoire de travail. Lorsque vous êtes prêt à enregistrer une copie de l'état actuel du projet, vous stagez les changements avec `git add`. Quand vous êtes satisfait de l'instantané stagé, vous le commitez dans l'historique du projet avec `git`

`commit`. La commande git reset permet d'annuler un commit ou un instantané stagé.

Outre `git add` et `git commit`, une troisième commande, git push, est essentielle à l'exhaustivité du workflow Git collaboratif. `git push` permet d'envoyer les changements commités vers des dépôts distants à des fins de collaboration. Ainsi, d'autres membres de l'équipe peuvent accéder à un ensemble de changements enregistrés.

La commande `git add` ne doit pas être confondue avec `svn add`, qui ajoute un fichier dans le dépôt. En effet, `git add` fonctionne sur le niveau le plus abstrait des changements. Ceci signifie que `git add` doit être appelé à chaque fois que vous modifiez un fichier, tandis que `svn add` ne doit être appelée qu'une seule fois pour chaque fichier. Cela peut sembler redondant, mais ce workflow facilite considérablement la bonne organisation d'un projet.

Zone de staging

La fonction principale de la commande `git add` consiste à promouvoir les changements en attente dans le répertoire de travail vers la zone de `staging Git`. La zone de staging est l'une des fonctionnalités uniques de Git. Vous aurez peut-être besoin de temps pour la comprendre si vous migrez depuis un environnement SVN (voire Mercurial). Cela vous aidera de la considérer comme une zone tampon entre le répertoire de travail et l'historique du projet. La zone de staging est considérée comme l'une des «

trois arborescences » de Git, avec le répertoire de travail et l'historique des commits.

Au lieu de commiter tous les changements apportés depuis le dernier commit, la zone de staging vous permet de regrouper les changements liés dans des instantanés ciblés avant de les commiter dans l'historique du projet. Autrement dit, vous pouvez apporter différents changements à des fichiers distincts, revenir en arrière et les séparer en commits logiques en ajoutant des changements liés au staging, puis les commiter un par un. Comme dans tout système de contrôle de version, il est important de créer des commits atomiques afin qu'il soit plus facile de suivre les bugs et d'inverser des changements avec un impact minime sur le reste du projet.

Options communes

```
git add
```

Permet de stager tous les changements dans `<fichier>` pour le commit suivant.

```
git add
```

Permet de stager tous les changements dans `<répertoire>` pour le commit suivant.

```
git add -p
```

Permet de commencer une session de staging interactif vous permettant de choisir des parties d'un fichier à ajouter au commit suivant. Un bloc de changements apparaîtra, et vous serez invité à entrer une commande. Utilisez `y` pour stager le bloc, `n` pour l'ignorer, `s` pour le diviser en blocs plus petits, `e` pour le modifier manuellement et `q` pour quitter.

Exemples

Lorsque vous commencez un nouveau projet, `git add` a la même fonction que `svn import`. Pour créer un commit initial du répertoire actuel, utilisez les deux commandes suivantes :

```
git add . git commit
```

Une fois que votre projet est prêt et qu'il est fonctionnel, de nouveaux fichiers peuvent être ajoutés en transmettant le chemin à `git add` :

```
git add hello.py git commit
```

Les commandes ci-dessus permettent également d'enregistrer des changements apportés aux fichiers existants. À nouveau, Git ne fait pas la distinction entre les changements de staging effectués dans les nouveaux fichiers et ceux effectués dans les fichiers déjà ajoutés au dépôt.

Inspecter un dépôt grâce à git status

git status git tag git blame

git status

La commande `git status` affiche l'état du répertoire de travail et de la zone de staging. Elle vous permet de voir les changements qui ont été stagés, ceux qui ne l'ont pas été, ainsi que les fichiers qui sont suivis par Git. La sortie de l'état n'affichera *pas* les informations sur l'historique du projet commité. Pour cela, vous devez utiliser `git log`.

Commandes Git connexes

- git tag
  - Les tags sont des réfs qui pointent vers des emplacements spécifiques de l'historique Git. `git tag` est généralement utilisée pour capturer un point de l'historique associé à une version marquée (c.-à-d., v1.0.1).
- git blame
  - `git blame` a pour fonction générale d'afficher les métadonnées d'auteur associées à des lignes de commit spécifiques dans un fichier. Cela permet d'examiner des points spécifiques de l'historique d'un fichier et de répondre aux questions « quel code a été ajouté à un dépôt », « comment » et « pourquoi ».
- git log

- La commande `git log` affiche des instantanés commités. Elle vous permet de lister l'historique du projet, de le filtrer et de rechercher des changements spécifiques.

Utilisation

```
git status
```

Répertorie les fichiers stagés, non stagés et non suivis.

Discussion

La commande `git status` est relativement simple. Elle vous montre simplement ce qui se passe avec `git add` et `git commit`. Les messages d'état comprennent également des instructions pertinentes pour effectuer ou annuler un stage de fichiers. Vous trouverez ci-dessous un exemple de sortie affichant les trois catégories principales d'un appel `git status` :

```
# Sur la branche principale # Changements à commiter : # (Utiliser "git reset HEAD ..." pour annuler le staging) # #Modifié : hello.py # # Changements non stagés à commiter : # (Utiliser "git add ..." pour mettre à jour ce qui sera commité) # (Utiliser "git checkout -- ..." pour annuler les changements dans le répertoire de travail) # #Modifié : main.py # # Fichiers non suivis : # (Utiliser "git add ..."pour inclure ce qui sera commité) # #hello.pyc
```

Ignorer des fichiers

Les fichiers non suivis sont généralement répartis en deux catégories. Il peut s'agir de fichiers qui ont simplement été ajoutés au projet et qui n'ont

pas encore été commités, ou de fichiers binaires compilés comme `.pyc`, `.obj`, `.exe`, etc. Bien qu'il soit sans aucun doute avantageux d'inclure le premier type de fichiers dans la sortie `git status`, le second peut créer des difficultés pour voir ce qui se passe réellement dans votre dépôt.

C'est la raison pour laquelle Git vous permet d'ignorer totalement les fichiers en plaçant les chemins dans un fichier spécial nommé `.gitignore`. Tous les fichiers que vous souhaitez ignorer doivent figurer sur une ligne distincte, et le symbole * peut être utilisé comme un caractère générique. Par exemple, le fait d'ajouter ce qui suit dans un fichier `.gitignore` à la racine de votre projet empêchera les modules Python compilés d'apparaître dans `git status` :

```
*.pyc
```

Exemple

Il est recommandé de vérifier l'état de votre dépôt avant de commiter les changements pour que vous ne commitiez pas accidentellement quelque chose sans le vouloir. Cet exemple indique l'état du dépôt avant et après le stage et le commit d'un instantané :

```
# Modifiez hello.py git status # hello.py est répertorié sous "Changes not staged for commit" git add hello.py git status # hello.py est répertorié sous "Changes to be committed" git commit git status # Rien à commiter (répertoire de travail propre)
```

La première sortie d'état affiche le fichier comme étant non stagé. L'action `git add` est reflétée dans le second `git status`, et la sortie d'état finale vous indique qu'il n'y a rien à commiter (le répertoire de travail correspond

au commit le plus récent). Certaines commandes Git (p. ex., `git merge`) nécessitent que le répertoire de travail soit propre pour éviter que vous n'écrasiez accidentellement des changements.

Git log

La commande `git log` affiche des instantanés commités. Elle vous permet de lister l'historique du projet, de le filtrer et de rechercher des changements spécifiques. Alors que `git status` vous permet d'inspecter le répertoire de travail et la zone de staging, `git log` fonctionne uniquement sur l'historique commité.

Vous pouvez personnaliser la sortie du journal de différentes manières, par exemple en filtrant simplement les commits ou en les affichant dans un format entièrement définissable par l'utilisateur. Certaines des configurations les plus courantes de `git log` sont présentées ci-dessous.

Utilisation

```
git log
```

Affiche l'historique complet des commits en utilisant le formatage par défaut. Si la sortie nécessite plusieurs écrans, vous pouvez utiliser la touche `Espace` pour faire défiler et la touche `q` pour quitter.

```
git log -n
```

Limite le nombre de commits à . Par exemple, `git log -n 3` affichera seulement trois commits.

```
git log --oneline
```

Rassemble tous les commits sur une seule ligne. Cette commande est utile pour obtenir un aperçu général de l'historique du projet.

```
git log --stat
```

En plus des informations `git log` ordinaires, incluez les fichiers qui ont été modifiés, et le nombre relatif de lignes qui ont été ajoutées ou supprimées de chacun d'entre eux.

```
git log -p
```

Affiche le patch représentant chaque commit. Cette commande affiche une comparaison complète de tous les commits, c'est la vue la plus détaillée que vous pouvez avoir de votre historique de projet.

```
git log --author=""
```

Recherche les commits d'un auteur en particulier. L'argument peut être une chaîne brute ou une expression régulière.

```
git log --grep=""
```

Recherche les commits avec un message de commit qui correspond à (chaîne brute ou expression régulière).

```
git log ..
```

Affiche uniquement les commits qui surviennent entre et . Les deux arguments peuvent être un ID de commit, un nom de branche, un `HEAD` ou tout autre type de référence de version.

```
git log
```

Affiche uniquement les commits qui comprennent le fichier spécifié. C'est un moyen facile de voir l'historique d'un fichier spécifique.

```
git log --graph --decorate --oneline
```

Quelques options utiles à prendre en compte. L'option --graph dessine un graphique basé sur le texte des commits à gauche des messages de commit. L'option --decorate ajoute les noms des branches ou des options des commits affichés. L'option --oneline indique les informations de commit sur une ligne unique, ce qui offre un aperçu rapide des commits.

Discussion

La commande `git log` est l'outil de base de Git pour explorer l'historique d'un dépôt. Vous l'utiliserez lorsque vous rechercherez une version spécifique d'un projet ou lorsque vous voudrez savoir quels changements seront introduits en faisant un merge dans une branche de fonctionnalité.

```
commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7 Author: Jean Dupond
```

Elle est relativement simple, mais la première ligne mérite quelques explications. La chaîne de 40 caractères après `commit` est une somme de contrôle SHA-1 du contenu du commit. Elle a deux fonctions. Tout d'abord,

elle assure l'intégrité du commit. S'il est corrompu, le commit génère une somme de contrôle différente. Elle sert également d'ID unique pour le commit.

Cet ID peut être utilisé dans des commandes comme `git log ..` pour référencer certains commits. Par exemple, `git log 3157e..5ab91` affiche tous les éléments entre les commits portant les ID `3157e` et `5ab91`. En dehors des sommes de contrôle, les noms de branche (examinés dans le module Branches) et le mot-clé HEAD sont d'autres méthodes courantes pour référencer des commits individuels. `HEAD` renvoie toujours au commit actuel, qu'il s'agisse d'une branche ou d'un commit spécifique.

Le caractère ~ permet de créer des références relatives au parent d'un commit. Par exemple, `3157e~1` renvoie au commit avant `3157e`, et `HEAD~3` est l'arrière grand-parent du commit actuel.

Toutes ces méthodes d'identification ont un objectif commun : vous permettre d'effectuer des actions en fonction de certains commits. La commande `git log` est généralement le point de départ de ces interactions. Elle vous permet en effet de trouver les commits avec lesquels vous voulez travailler.

Exemple

La section *Utilisation* contient de nombreux exemples de `git log`, mais gardez à l'esprit qu'il est possible de combiner plusieurs options dans une commande :

```
git log --author="Jean Dupond" -p hello.py
```

Cette commande affichera un diff complet de tous les changements apportés par Jean Dupond dans le fichier `hello.py`.

La syntaxe .. est un outil très utile pour comparer des branches. L'exemple suivant affiche un bref aperçu de tous les commits situés dans `some-feature` et en dehors de `master`.

```
git log --oneline master..some-feature
```

Annuler les commits et les changements

Git checkout git clean Git revert git reset git rm

Dans cette section, nous allons aborder les stratégies et les commandes d'annulation Git disponibles. En premier lieu, il est important de noter que Git ne dispose pas de système d'annulation traditionnel comme ceux présents dans une application de traitement de texte. Il sera utile de vous abstenir de mapper des opérations Git à un modèle mental d'annulation traditionnel, quel qu'il soit. En outre, Git possède sa propre nomenclature pour les opérations d'annulation, et il convient de l'évoquer durant une discussion. Cette nomenclature inclut des termes comme reset, revert, checkout, clean, et bien d'autres.

Une métaphore amusante consiste à considérer Git comme un utilitaire de gestion des chronologies. Les commits sont des instantanés d'un point dans le temps ou de points d'intérêt tout au long de l'historique d'un projet. En outre, il est possible de gérer plusieurs chronologies en utilisant des branches. Lorsque vous faites des annulations dans Git, vous reculez généralement dans le temps ou vous revenez à une autre chronologie dans laquelle les erreurs n'ont pas été commises.

Ce tutoriel présente toutes les compétences nécessaires pour utiliser les versions précédentes d'un projet logiciel. Tout d'abord, il montre comment explorer d'anciens commits, puis il explique la différence entre le revert de commits publics dans l'historique du projet et le reset des changements non publiés sur votre machine locale.

Retrouvez des éléments perdus : comment passer en revue d'anciens commits

L'idée sous-jacente de tout système de contrôle de version est de stocker des copies « sécurisées » d'un projet afin que vous n'ayez plus à vous préoccuper de corrompre irrémédiablement votre base de code. Une fois que vous avez créé un historique de projet pour les commits, vous pouvez passer en revue et revoir tous les commits de l'historique. La commande `git log` constitue l'un des meilleurs outils pour passer en revue l'historique d'un dépôt Git. Dans l'exemple ci-dessous, nous utilisons git log

pour obtenir la liste des derniers commits effectués dans une bibliothèque graphique open source populaire.

```
git log --oneline e2f9a78fe Replaced FlyControls with OrbitControls d35ce0178 Editor: Shortcuts panel Safari support. 9dbe8d0cf Editor: Sidebar.Controls to Sidebar.Settings.Shortcuts. Clean up. 05c5288fc Merge pull request #12612 from TyLindberg/editor-controls-panel 0d8b6e74b Merge pull request #12805 from harto/patch-1 23b20c22e Merge pull request #12801 from gam0022/improve-raymarching-example-v2 fe78029f1 Fix typo in documentation 7ce43c448 Merge pull request #12794 from WestLangley/dev-x 17452bb93 Merge pull request #12778 from OndrejSpanel/unitTestFixes b5c1b5c70 Merge pull request #12799 from dhritzkiv/patch-21 1b48ff4d2 Updated builds. 88adbcdf6 WebVRManager: Clean up. 2720fbb08 Merge pull request #12803 from dmarcos/parentPoseObject 9ed629301 Check parent of poseObject instead of camera 219f3eb13 Update GLTFLoader.js 15f13bb3c Update GLTFLoader.js 6d9c22a3b Update uniforms only when onWindowResize 881b25b58 Update ProjectionMatrix on change aspect
```

Chaque commit possède une empreinte d'identification SHA-1 unique. Ces identifiants sont utilisés pour explorer la chronologie commitée et réexaminer les commits. Par défaut, `git log` affiche uniquement les commits pour la branche actuellement sélectionnée. Il est tout à fait possible que le commit que vous recherchez se trouve sur une autre branche. Vous pouvez afficher tous les commits sur toutes les branches en exécutant la commande `git log --branches=*`. La commande git branch permet d'afficher et d'explorer d'autres branches. Lorsque vous appelez la commande `git branch -a`, celle-ci renvoie une liste de tous les noms de branche connus. L'un de ces noms de branche peut ensuite être consigné à l'aide de la commande `git log`.

Une fois que vous avez trouvé une référence de commit au point dans l'historique que vous souhaitez explorer, vous pouvez utiliser la commande `git checkout` pour explorer ce commit. `git checkout` est une méthode simple pour « charger » l'un de ces instantanés sauvegardés sur votre machine de développement. Durant le cours normal du développement,

l'élément `HEAD` pointe généralement vers la branche `master` ou une autre branche locale, mais lorsque vous faites un check-out d'un ancien commit, `HEAD` ne pointe plus vers une branche. Il pointe directement vers un commit. On parle alors d'état « `HEAD` détaché », qui peut être illustré comme suit :

Lorsque vous faites le check-out d'un ancien fichier, vous ne déplacez pas le pointeur `HEAD`. Ce dernier reste sur la même branche et le même commit, à l'état « HEAD détaché ». Vous pouvez ensuite faire un commit de l'ancienne version du fichier dans un nouvel instantané comme vous le feriez pour tout autre changement. Utiliser `git checkout` dans un fichier permet donc de revenir à une ancienne version d'un fichier donné. Pour plus d'informations sur ces deux modes, consultez la page git checkout.

Afficher une ancienne version

Pour cet exemple, partons du principe que vous avez commencé à développer quelque chose d'un peu fou, mais que vous hésitez encore à le conserver ou pas. Pour vous aider à prendre une décision, vous souhaitez examiner l'état du projet avant le début de votre test. Tout d'abord, vous devez rechercher l'ID de la version que vous souhaitez afficher.

```
git log --oneline
```

Supposons que votre historique de projet ressemble à ceci :

```
b7119f2 Continue doing crazy things 872fa7e Try something crazy a1e8fb5 Make some important changes to hello.txt 435b61d Create hello.txt 9773e52 Initial import
```

Vous pouvez utiliser `git checkout` pour afficher le commit « Make some import changes to hello.txt » comme suit :

```
git checkout a1e8fb5
```

Ainsi, l'état de votre répertoire de travail correspond à celui du commit `a1e8fb5`. Vous pouvez visualiser les fichiers, compiler le projet, effectuer des tests et même modifier des fichiers sans craindre de compromettre l'état actuel du projet. Rien de ce que vous ferez ici ne sera enregistré dans votre dépôt. Pour poursuivre le développement, vous devez revenir à l'état « actuel » de votre projet :

```
git checkout master
```

Cela suppose que vous développiez sur la branche `master` par défaut. Une fois que vous êtes revenu sur la branche `master`, vous pouvez utiliser git revertou git reset pour annuler les changements indésirables.

Annuler un instantané commité

Techniquement, il existe plusieurs stratégies différentes pour annuler un commit. Les exemples suivants partent du principe que nous possédons un historique des commits qui ressemble à ce qui suit :

```
git log --oneline 872fa7e Try something crazy a1e8fb5 Make some important changes to hello.txt 435b61d Create hello.txt 9773e52 Initial import
```

Nous allons nous concentrer sur l'annulation du commit `872fa7e Try something crazy`, qui était peut-être un peu trop insensé.

Comment annuler un commit avec git checkout

À l'aide de la commande `git checkout`, nous pouvons faire un check-out du commit précédent, `a1e8fb5`, pour rétablir le dépôt à un état antérieur au commit insensé. Lorsque vous faites le check-out d'un commit donné, le dépôt est placé à l'état « `HEAD détaché` ». Cela signifie que vous ne travaillez plus sur une branche. À l'état détaché, tous les nouveaux commits que vous ferez seront orphelins lorsque vous repassez à une branche établie. Les commits orphelins seront supprimés par la commande garbage collection de Git. Cette dernière s'exécute à intervalles fixes et supprime les commits orphelins de façon permanente. Pour éviter leur suppression par la commande garbage collection, nous devons nous assurer de travailler sur une branche.

À l'état `HEAD` détaché, nous pouvons exécuter la commande `git checkout -b nouvelle_branche_sans_commit_insensé`. Une nouvelle branche appelée `nouvelle_branche_sans_commit_insensé` va ainsi être créée, et l'état sera modifié en conséquence. Le dépôt se trouve maintenant sur une nouvelle chronologie dans laquelle le commit `872fa7e` n'existe plus. À ce stade, nous pouvons continuer de travailler sur cette nouvelle branche dans laquelle le commit `872fa7e` n'existe plus et la considérer comme « annulée ». Malheureusement, si vous avez besoin de

la branche précédente, qui était peut-être votre `master`, cette stratégie d'annulation n'est pas appropriée. Voyons un peu quelques autres stratégies d'annulation. Pour plus d'informations et d'exemples, consultez notre discussion approfondie sur git checkout.

Comment annuler un commit public avec git revert

Supposons que nous sommes de retour dans notre historique des commits d'origine. Celui-ci inclut le commit `872fa7e`. Essayons cette fois de faire un revert d'une annulation. Si nous exécutons la commande `git revert HEAD`, Git crée un nouveau commit avec l'inverse du dernier commit. Un nouveau commit est alors ajouté à l'historique de branche actuel, qui ressemble maintenant à cela :

```
git log --oneline e2f9a78 Revert "Try something crazy" 872fa7e Try something crazy a1e8fb5 Make some important changes to hello.txt 435b61d Create hello.txt 9773e52 Initial import
```

À ce stade, nous avons techniquement annulé le commit `872fa7e`. Même si le commit `872fa7e` existe encore dans l'historique, le nouveau commit `e2f9a78` est l'inverse des changements apportés dans `872fa7e`. Contrairement à notre précédente stratégie de check-out, nous pouvons continuer d'utiliser la même branche. Cette solution est satisfaisante. C'est la méthode d'annulation idéale pour travailler avec des dépôts publics partagés. Si vous devez conserver un historique Git organisé et minimal, cette stratégie peut ne pas être satisfaisante.

Comment annuler un commit avec git reset

Pour cette stratégie d'annulation, nous allons poursuivre avec notre exemple de travail. git reset est une commande poussée à usages et fonctions multiples. Si nous appelons `git reset --hard a1e8fb5`, l'historique des commits est réinitialisé sur ce commit spécifique. Lorsque vous exécutez la commande `git log` pour examiner l'historique des commits, celui-ci ressemble à ce qui suit :

```
git log --oneline a1e8fb5 Make some important changes to hello.txt 435b61d Create hello.txt 9773e52 Initial import
```

La sortie de journal indique que les commits `e2f9a78` et `872fa7e` n'existent plus dans l'historique. À ce stade, nous pouvons continuer de travailler et de créer des commits comme si les commits « insensés » n'avaient jamais eu lieu. Cette méthode d'annulation de changements est la plus « propre » pour l'historique. La commande git reset est idéale pour les changements locaux ; elle ajoute cependant des complications lorsque vous travaillez avec un dépôt distant partagé. Si nous avons un dépôt distant partagé vers lequel nous avons fait un push du commit `872fa7e` et si nous essayons d'exécuter la commande `git push` sur une branche dans laquelle l'historique a été réinitialisé, Git s'en aperçoit et renvoie une erreur. Git suppose que la branche dont vous faites un push n'est pas à jour en raison de ses commits manquants. Dans ces scénarios, vous devriez préférer `git revert` comme méthode d'annulation.

Annuler le dernier commit

Dans la section précédente, nous avons vu différentes stratégies pour annuler des commits. Toutes sont également applicables au commit le plus récent. Mais, dans certains cas, vous ne pourrez peut-être pas supprimer ou restaurer le dernier commit. Il est possible qu'il ait été réalisé prématurément. Dans ce cas, vous pouvez modifier le commit le plus récent. Quand vous avez fait d'autres changements dans le répertoire de travail et que vous les avez stagés à des fins de commit à l'aide de git add, vous pouvez exécuter `git commit --amend`. Cette commande va demander à Git d'ouvrir l'éditeur système configuré pour vous permettre de modifier le dernier message de commit. Les nouveaux changements seront ajoutés au commit modifié.

Annuler des changements non commités

Avant que les changements ne soient commités dans l'historique du dépôt, ils résident dans l'index de staging et dans le répertoire de travail. Vous devrez peut-être annuler des changements dans ces deux zones. L'index de staging et le répertoire de travail sont des mécanismes de gestion des états internes à Git. Pour des informations plus détaillées sur le fonctionnement de ces deux mécanismes, consultez la page git reset qui les explore en détail.

Le répertoire de travail

Le répertoire de travail est généralement synchronisé avec le système de fichiers local. Pour annuler des changements dans le répertoire de travail, vous pouvez modifier des fichiers comme vous le feriez normalement à l'aide de votre éditeur préféré. Git propose plusieurs utilitaires pour vous aider à gérer le répertoire de travail. La commande git clean est un utilitaire pratique pour annuler des changements dans le répertoire de travail. En outre, `git reset` peut être appelée avec l'option `--mixed` ou `--hard`. Elle applique alors un reset sur le répertoire de travail.

L'index de staging

La commande git add permet d'ajouter des changements à l'index de staging. `git reset` est principalement utilisée pour annuler les changements apportés à l'index de staging. Un reset avec l'option `--mixed` déplace tout changement en attente de l'index de staging vers le répertoire de travail.

Annuler des changements publics

Lorsque vous travaillez en équipe sur des dépôts distants, des considérations supplémentaires entrent en ligne de compte lors de l'annulation de changements. `git reset` devrait généralement être considérée comme une méthode d'annulation locale. Un reset devrait être utilisé lorsque vous annulez des changements dans une branche privée. Vous isolez ainsi nettement la suppression de commits des autres branches susceptibles d'être utilisées par d'autres développeurs. Des problèmes se posent lorsqu'un reset est exécuté sur une branche partagée et qu'un push est réalisé à distance sur cette branche à l'aide de `git push`. Git bloque alors le push dans ce scénario, en arguant que la branche pushée n'est pas à jour par rapport à branche distante, car elle ne contient pas tous les commits.

La méthode privilégiée pour annuler un historique partagé est `git revert`. Un revert est plus sûr qu'un reset, car il ne supprime pas les commits d'un historique partagé. Un revert conserve les commits à annuler et crée un commit qui inverse le commit indésirable. Cette méthode est plus sûre pour la collaboration à distance, car un développeur travaillant à distance peut ensuite faire un pull de la branche et recevoir le nouveau commit restauré, lequel annule le commit indésirable.

Réécrire l'historique

git commit --amend et autres méthodes de réécriture de l'historique

Introduction

Ce tutoriel aborde diverses méthodes de réécriture et de modification de l'historique Git. Git utilise plusieurs méthodes différentes pour enregistrer les changements. Nous aborderons les atouts et les faiblesses de chacune d'elles, et nous expliquerons comment les utiliser. Ce tutoriel développe certaines des raisons les plus courantes pour lesquelles les instantanés commités sont écrasés et vous explique comment éviter les pièges associés à cette opération.

La principale tâche de Git est de s'assurer que vous ne perdiez jamais un changement commité. Git est également conçu pour vous donner le contrôle total de votre workflow de développement. Il vous permet notamment de définir avec exactitude l'apparence de votre historique de projet, mais il crée également des conditions pouvant entraîner la perte de commits. Git fournit ses commandes de réécriture de l'historique en se dégageant de toute responsabilité en cas de perte de contenu découlant de leur utilisation.

Git comprend plusieurs mécanismes de stockage de l'historique et d'enregistrement des changements, parmi lesquels : commit `--amend`, `git rebase` et `git reflog`. Ces options permettent de personnaliser le workflow. À l'issue de ce tutoriel, vous maîtriserez des commandes qui vous permettront de restructurer vos commits Git et vous serez en mesure d'éviter les pièges courants lors de la réécriture de l'historique.

Modification du dernier commit : `git commit --amend`

La commande `git commit --amend` permet de modifier facilement le commit le plus récent. Elle vous permet de combiner les changements stagés avec l'ancien commit au lieu de créer un commit totalement nouveau. Elle peut également être utilisée pour modifier le message de commit sans changer son instantané. Cependant, la modification ne se contente pas de changer le dernier commit. Elle le remplace totalement, ce qui signifie que le commit modifié constitue une toute nouvelle entité qui dispose de sa propre réf. Pour Git, elle ressemblera à un tout nouveau commit, marqué par un astérisque (*) dans le schéma ci-dessous. Il existe quelques cas d'usage courants de `git commit --amend`. Nous les aborderons dans les sections suivantes.

Modifier le dernier message de commit Git

```
git commit --amend
```

Imaginons que vous venez de faire un commit et que vous avez fait une erreur dans votre message. L'exécution de cette commande lorsqu'aucun élément n'est stagé vous permet de modifier le message du commit précédent sans modifier son instantané.

Des commits prématurés apparaissent en permanence au cours de votre développement quotidien. On peut facilement oublier de stager un fichier

ou formater le message de commit de manière incorrecte. Le flag `--amend` est un moyen pratique de corriger ces erreurs mineures.

```
git commit --amend -m "Message de commit mis à jour"
```

En ajoutant l'option `-m`, vous pouvez transmettre un nouveau message à partir de la ligne de commande sans être invité à ouvrir un éditeur.

Modifier des fichiers commités

L'exemple suivant illustre un scénario Git commun. Imaginons que nous avons édité des fichiers que nous souhaitons commiter dans un instantané unique, mais que nous avons oublié d'ajouter l'un des fichiers lors de la première tentative. Pour réparer l'erreur, il suffit de stager l'autre fichier et de faire un commit avec le flag `--amend` :

```
# Vous modifiez hello.py et main.py git add hello.py git commit # Vous réalisez que vous avez oublié d'ajouter les changements de main.py git add main.py git commit --amend --no-edit
```

Le flag `--no-edit` vous permet de modifier votre commit sans changer son message. Le commit obtenu remplacera le commit incomplet. Il sera structuré comme si nous avions commité des changements apportés à `hello.py` et `main.py` dans un instantané unique.

Ne pas modifier les commits publics

Les commits modifiés sont en réalité des commits entièrement nouveaux. Les anciens commits ne se trouveront plus dans votre branche actuelle. Les conséquences associées sont les mêmes que pour la réinitialisation

d'un instantané public. Évitez de modifier un commit sur lequel repose le travail d'autres développeurs. Cette situation est déroutante, et il est difficile de revenir en arrière.

Récapitulatif

Pour résumer, `git commit --amend` vous permet de sélectionner le dernier commit afin d'y ajouter de nouveaux changements stagés. Vous pouvez ajouter ou supprimer des changements à partir de la zone de staging afin de les appliquer avec un commit `--amend`. Si aucun changement n'est stagé, `--amend` vous invite tout de même à modifier le dernier message de journal du commit. Utilisez `--amend` avec précaution dans les commits partagés avec d'autres membres de l'équipe. Lorsque vous modifiez un commit partagé avec un autre utilisateur, vous devrez peut-être résoudre des conflits de merge, une opération chronophage.

Modifier des commits plus anciens ou plusieurs commits

Pour modifier des commits plus anciens ou plusieurs commits, vous pouvez utiliser `git rebase` afin de combiner une séquence de commits dans un nouveau commit de base. Dans le mode standard, `git rebase` vous permet de réécrire l'historique, c'est-à-dire d'appliquer automatiquement des commits de votre branche de travail actuelle à l'élément HEAD de la branche transmise. Puisque vos nouveaux commits

vont remplacer les anciens, il est important de ne pas utiliser `git rebase` sur des commits publics. Sinon, votre historique de projet disparaîtra.

Lorsqu'il est important de conserver un historique de projet propre, vous pouvez ajouter l'option `-i` à la commande `git rebase` pour exécuter un `rebase interactif`. Vous avez ainsi la possibilité de modifier des commits individuels du processus plutôt que de déplacer tous les commits. Pour de plus amples informations sur le rebase interactif et des commandes rebase supplémentaires, reportez-vous à la page git rebase.

Modifier des fichiers commités

Durant un rebase, la commande edit ou `e` interrompt la lecture du rebase à ce commit, ce qui vous permet d'apporter des changements supplémentaires avec `git commit --amend`. Git interrompt la lecture et affiche le message suivant :

```
Stopped at 5d025d1... formatting You can amend the commit now, with git commit --amend
Once you are satisfied with your changes, run git rebase --continue
```

Messages multiples

Chaque commit Git normal comprend un message de journal qui explique ce qu'il s'est passé dans le commit. Ces messages fournissent de précieux renseignements qui sont ajoutés à l'historique de projet. Durant un rebase, vous pouvez exécuter quelques commandes dans des commits pour modifier leurs messages.

- reword ou r arrête la lecture du rebase et vous permet de réécrire le message de commit.

- Si vous lancez la commande squash ou s durant la lecture du rebase, tous les commits marqués par `s` seront interrompus, et vous serez invité à modifier les messages de commit séparés dans un message combiné. Pour plus de détails, reportez-vous à la section sur le squash de commits ci-dessous.
- La commande fixup ou f a le même effet de combinaison que la commande squash, à l'exception près que le fixup de commits n'interrompt pas la lecture du rebase pour ouvrir un éditeur dans lequel les messages de commit seront combinés. Les messages des commits marqués par f seront supprimés et remplacés par le message du commit précédent.

Squasher des commits pour nettoyer l'historique

La commande squash ou `s` montre la véritable utilité du rebase. squash vous permet de spécifier les commits à merger dans les commits précédents. C'est ainsi que vous obtenez un « historique propre ». Durant la lecture du rebase, Git exécute la commande rebase spécifiée pour chaque commit. Dans le cas du squash de commits, Git ouvre l'éditeur de texte que vous avez configuré et vous invite à combiner les messages de commit spécifiés. Le processus global peut être illustré comme suit :

Notez que les commits modifiés à l'aide d'une commande rebase possèdent un ID différent des commits d'origine. Les commits marqués d'une coche recevront un nouvel ID si les commits précédents ont été réécrits.

Les solutions d'hébergement Git modernes telles que Bitbucket proposent désormais des fonctionnalités de « squash automatique » lors du merge. Ces fonctionnalités effectuent automatiquement un rebase et un squash des commits d'une branche lorsque vous utilisez l'interface utilisateur des solutions d'hébergement. Pour de plus amples informations, reportez-vous à l'article « Faire un squash de commits lors d'un merge de branche Git avec Bitbucket ».

Récapitulatif

git rebase vous donne la possibilité de modifier votre historique, alors que le rebase interactif vous permet de le faire sans laisser de traces. Vous avez ainsi la liberté de faire des erreurs, de les corriger et d'affiner votre travail tout en conservant un historique de projet propre et linéaire.

Le filet de sécurité : git reflog

Les journaux de référence ou reflogs constituent un mécanisme utilisé par Git pour enregistrer les mises à jour appliquées aux pointes des branches et à d'autres références de commit. Le reflog vous permet de revenir sur les commits même s'ils ne sont pas référencés par une branche ou un tag. Une fois l'historique réécrit, le reflog contient des informations sur l'ancien état des branches et vous permet d'y revenir si nécessaire. À chaque mise à jour de la pointe de votre branche quelle qu'en soit la raison (en changeant de branche, en faisant un pull de nouveaux changements, en réécrivant l'historique ou simplement en ajoutant de nouveaux commits), une nouvelle

entrée est ajoutée au reflog. Dans cette section, nous verrons brièvement la commande `git reflog` et nous étudierons quelques cas d'usage courants.

Utilisation

```
git reflog
```

Cette commande affiche le reflog pour le dépôt local.

```
git reflog --relative-date
```

Cette commande affiche le reflog avec les informations de date relative (p. ex., deux semaines plus tôt).

Exemple

Pour comprendre `git reflog`, nous allons étudier un exemple concret.

```
0a2e358 HEAD@{0}: reset: moving to HEAD~2 0254ea7 HEAD@{1}: checkout: moving from 2.2 to master c10f740 HEAD@{2}: checkout: moving from master to 2.2
```

Le reflog ci-dessus montre un check-out de la branche master sur la branche 2.2 et l'opération inverse. À partir de là, nous faisons un hard reset d'un commit plus ancien. La dernière activité est représentée dans la partie supérieure, nommée `HEAD@{0}`.

Si vous êtes revenu en arrière sans le vouloir, le reflog contiendra un master commit qui pointait vers `(0254ea7)` avant que vous ne déplaciez accidentellement les deux commits.

```
git reset --hard 0254ea7
```

Avec git reset, il est désormais possible de changer la branche master et de rétablir le commit à son état précédent. C'est votre filet de sécurité en cas de modification accidentelle de votre historique.

Attention : le reflog ne constitue un filet de sécurité que si les changements ont été commités dans votre dépôt local et qu'il suit seulement les déplacements de la pointe de la branche du dépôt. De plus, les entrées reflog ont une date d'expiration. Par défaut, les entrées reflog expirent au bout de 90 jours.

Pour de plus amples informations, reportez-vous à notre page git reflog

SOURCE DU TUTO :
https://www.atlassian.com/fr/git/tutorials/setting-up-a-repository

FIN DU TUTO BITBUCKET