Git でよく使うコマンド一覧

git version

コマンド	説明
`git version`	Git のバージョンを出力する

git clone

コマンド	説明
`git clone {リポジトリの URL}`	対象リポジトリのデフォルトブランチをクローンする
`git clone --depth {深さ} {リポジトリの URL}`	対象リポジトリのデフォルトブランチを指定したコミット数で切り詰めてクローンする
`git clone -b {ブランチ名} {リポジトリの URL}`	対象リポジトリの対象ブランチをクローンする

`git clone --depth 1 {リポジトリの URL}` で、履歴が多いリポジトリをクローンする時間を短縮できます。

git remote

コマンド	説明
`git remote`	リモートリポジトリの一覧を出力する
`git remote add {リモートリポジトリ名} {リポジトリの URL}`	対象リポジトリをローカルのリモートリポジトリに追加する
`git remote rename {旧リモートリポジトリ名} {新リモートリポジトリ名}`	ローカルの対象リモートリポジトリをリネームする
`git remote remove {リモートリポジトリ名}`	対象リモートリポジトリをローカルから削除する

git branch

コマンド	説明
`git branch`	ローカルブランチの一覧を出力する（チェックアウト中のブランチに * が付く）
`git branch -a`	ローカルブランチとリモートブランチの一覧を出力する
`git branch {ブランチ名}`	対象ブランチを新規作成する（チェックアウトしない）
`git branch -d {ブランチ名}`	対象ブランチを削除する
`git branch -d -f {ブランチ名}`	対象ブランチを **強制** 削除する
`git branch -D {ブランチ名}`	対象ブランチを **強制** 削除する

コマンド	説明
`git branch -m {旧ブランチ名} {新ブランチ名}`	対象ブランチをリネームする

git checkout

コマンド	説明
`git checkout {ブランチ名}`	対象ブランチに切り替える
`git checkout -b {ブランチ名}`	対象ブランチを新規作成し、切り替える
`git checkout {ファイルパス}`	ワークツリーにある対象ファイルの変更を取り消す
`git checkout .`	ワークツリーにある全ファイルの変更を取り消す

2.23.0 で `git switch` と `git restore` コマンドが追加され、`git checkout` コマンドを使わなくても済むようになりました。

git switch

コマンド	説明
`git switch {ブランチ名}`	対象ブランチに切り替える
`git switch -c {ブランチ名}`	対象ブランチを新規作成し、切り替える
`git switch --detach refs/tags/{タグ名}`	対象タグに切り替える

git restore

コマンド	説明
`git restore {ファイルパス}`	ワークツリーにある対象ファイルの変更を取り消す
`git restore .`	ワークツリーにある全ファイルの変更を取り消す
`git restore --source {コミット ID} {ファイルパス}`	対象ファイルの変更を対象コミットに戻す

git diff

コマンド	説明
`git diff`	**ワークツリー** にあるファイルの差分を出力する
`git diff --cached`	**インデックス** にあるファイルの差分を出力する

git status

コマンド	説明
`git status`	変更したファイルの一覧を出力する
`git status -s`	`git status` を短い形式で出力する
`git status -s -b`	短い形式でもブランチとトラッキングを出力する

git add

コマンド	説明
`git add {ファイルパス 1} {ファイルパス 2}...`	対象ファイルをインデックス（コミット対象）に追加する
`git add -A`	変更した全ファイルをインデックスに追加する
`git add -p {ファイルパス}`	対象ファイルをハンク単位でインデックスに追加する

git reset

コマンド	説明
`git reset HEAD {ファイルパス}`	ステージングにある対象ファイルをワークツリーに戻す（`git add {ファイルパス}` を取り消す）
`git reset HEAD`	ステージングにある全ファイルをワークツリーに戻す（`git add -A` を取り消す）

git commit

コマンド	説明
`git commit`	指定したエディタでメッセージを書き、インデックスにある全ファイルをコミットする
`git commit -m "{メッセージ}"`	メッセージを付け、インデックスにある全ファイルをコミットする

「指定したエディタ」とは、`.gitconfig` の `editor` で指定しているエディタのことです。

```
[core]
    editor = nvim
```

git revert

コマンド	説明
`git revert HEAD`	直前のコミットを元に戻すコミットを作成する
`git revert {コミット ID}`	対象コミットを元に戻すコミットを作成する

git push

コマンド	説明
`git push origin {ローカルブランチ名}`	対象ローカルブランチを `origin` にプッシュする
`git push -d origin {リモートブランチ名}`	対象リモートブランチを `origin` から削除する
`git push origin {タグ名}`	対象タグを `origin` にプッシュする
`git push -d origin {タグ名}`	対象タグを `origin` から削除する

コマンド	説明
`git push -f`	強制プッシュする
`git push --force-with-lease`	強制プッシュする（ブランチのアップストリームが変更されている場合などは拒否する）

git fetch

コマンド	説明
`git fetch origin`	`origin` から最新の履歴を取得する
`git fetch --prune origin`	リモートリポジトリ上に存在しなくなったブランチなどの参照を削除し、`origin` から最新の履歴を取得する

git rebase

コマンド	説明
`git rebase origin/{ブランチ名}`	`origin` にある対象ブランチを、チェックアウト中のブランチへリベースする
`git rebase --continue`	リベースを続ける
`git rebase --quit`	リベースを中止する
`git rebase --abort`	リベースを強制終了する

git merge

コマンド	説明
`git merge origin/{ブランチ名}`	`origin` にある対象ブランチを、チェックアウト中のブランチへマージする
`git merge --squash origin/{ブランチ名}`	`origin` にある対象ブランチのコミットをひとつにまとめ、チェックアウト中のブランチへマージする

git cherry-pick

コマンド	説明
`git cherry-pick {コミット ID}`	対象コミットをチェリーピックする
`git cherry-pick {始点の 1 つ前のコミット ID}..{終点のコミット ID}`	始点から終点までのコミットをチェリーピックする
`git cherry-pick {始点のコミット ID}^..{終点のコミット ID}`	始点から終点までのコミットをチェリーピックする
`git cherry-pick --skip`	現在のコミットをスキップして、残りのシーケンスを続ける
`git cherry-pick --quit`	失敗したチェリーピックを取り消す

git stash

コマンド	説明
`git stash list`	スタッシュの一覧を出力する
`git stash show`	スタッシュの変更を出力する
`git stash push -m "{メッセージ}"`	メッセージを付け、変更をスタッシュにプッシュする
`git stash pop`	スタッシュの変更をワークツリーに戻す（スタッシュから **消える** ）
`git stash apply`	スタッシュの変更をワークツリーに戻す（スタッシュから **消えない** ）
`git stash drop`	スタッシュから変更を削除する

2020/03/12 現在、`git stash save` は非推奨です。

git log

コマンド	説明
`git log`	ログを出力する
`git log --graph --name-status --pretty=format:"%C(red)%h %C(green)%an %C(Cyan)%ad %Creset%s %C(yellow)%d%Creset"`	ぼくのかんがえたさいきょうのぎっとろぐ

git blame

コマンド	説明
`git blame {ファイルパス}`	対象ファイル全体の各行ごとに、最後に編集したリビジョンと作者を表示する
`git blame -L {開始行} {ファイルパス}`	対象ファイルの開始行から末尾までを各行ごとに、最後に編集したリビジョンと作者を表示する
`git blame -L ,{終了行} {ファイルパス}`	対象ファイルの先頭から終了行までを各行ごとに、最後に編集したリビジョンと作者を表示する
`git blame -L {開始行},{終了行} {ファイルパス}`	対象ファイルの開始行から終了行までを各行ごとに、最後に編集したリビジョンと作者を表示する

git show

コマンド	説明
`git show {コミット ID}`	対象コミットのメッセージとテキストの差分を表示する

git tag

コマンド	説明
`git tag {タグ名}`	タグを付ける
`git tag -d {タグ名}`	タグを削除する

git rm

コマンド	説明
`git rm {ファイルパス}`	対象ファイルを **削除し** 、Git の管理外にする
`git rm --cached {ファイルパス}`	対象ファイルを **削除せず** 、Git の管理外にする
`git rm -r {フォルダパス}`	対象フォルダ以下を全削除し、Git の管理外にする

Git でよく使うテクニック

対象ファイルを一部分のみインデックスに追加する

`git add {ファイルパス}` で対象ファイルの全変更をインデックスに追加できます。

一部の変更のみ追加したいときは `git add -p {ファイルパス}` を使います。

```
$ git add -p {ファイルパス}
...
(1/3) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]?
```

変更は「hunk（ハンク）」という単位で扱います。

ハンクごとに対話式で聞かれるので、基本的には「 y（追加する）」と「 n（追加せずそのまま）」を入力します。

各操作の意味を載せます。

```
(1/3) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
e - manually edit the current hunk
? - print help
```

rebase 時に発生したコンフリクトを解消する

rebase 時に発生したコンフリクトは、手動で解消後に rebase --continue を実行する必要があります。

```
# 1. fetch→rebase する
$ git fetch origin
$ git rebase origin/master

# 2.コンフリクトが発生したら、手動で解消して add する
$ git add Foo.swift

# 3. `rebase --continue` を実行する
$ git rebase --continue

# 4. 対象ブランチを強制プッシュする
$ git push -f origin master
```

強制プッシュしているので、タイミングにはご注意ください。

間違えてプッシュしたコミットを取り消す

```
$ git reset --hard HEAD^
$ git push -f
```

強制プッシュしているので、タイミングにはご注意ください。

直前のコミットメッセージを変更する

直前のコミットメッセージは git commit --amend で変更できます。

テキストエディタが開くので、コミットメッセージを変更して保存します。

```
$ git commit --amend
$ git push -f
```

すでにプッシュしている場合、強制プッシュする必要があります。

コミット ID が変わる点も注意です。

過去のコミットの Author と Committer を一括で変更する

Git でコミットするユーザーを間違え、まとめて修正したい場合に使えるテクニックです。

私は SourceTree が .gitconfig にユーザー名とメールアドレスを追加したことに気づかずコミットし、修正するために使いました。

以下はコミッターが特定のメールアドレスのコミットのみ変更しています。

必要に応じて if 文の条件を変更してください。

```
$ git filter-branch --commit-filter '
if [ "$GIT_COMMITTER_EMAIL" = "{変更したいコミッターのメールアドレス}" ];
  then
    GIT_COMMITTER_NAME="{変更後のコミッター名}";
```

```
    GIT_AUTHOR_NAME="{変更後の作者名}";
    GIT_COMMITTER_EMAIL="{変更後のコミッターのメールアドレス}";
    GIT_AUTHOR_EMAIL="{変更後の作者のメールアドレス}";
    git commit-tree "$@";
  else
    git commit-tree "$@";
  fi' HEAD
$ git push -f
```

コミットの取り消しと同様、強制プッシュしているので、タイミングにはご注意ください。

2021/05/11 現在、**`git filter-branch` は安全性とパフォーマンスの問題により非推奨** です。
`git filter-repo` などを代わりに使いましょう。
https://github.com/newren/git-filter-repo/

タグを付け替える

タグを直接付け替えることはできないので、タグを削除して同名で付け直します。

```
# ローカルでタグを削除して付け直す
$ git tag -d {タグ名}
$ git tag {タグ名}

# リモートリポジトリからも削除し、付け直したタグをプッシュする
$ git push -d origin {タグ名}
$ git push origin {タグ名}
```

ちなみに GitHub でリリース済みのタグを付け直すと、リリースがドラフトになりました。

リモートトラッキングの参照が残ってフェッチできない

`git fetch origin` を実行すると、以下のエラーが発生することがあります。

```
$ git fetch origin
error: cannot lock ref 'refs/remotes/origin/feature/foo': 'refs/remotes/origin/feature' exists; cannot create 'refs/remotes/origin/feature/foo'
```

`feature` ブランチがあって `feature/foo` ブランチを作成できないためです。
`git branch -d feature` を実行するのみだと、リモートトラッキングの参照が残ってしまいます。

```
$ git branch -d feature
$ git branch -a
remotes/origin/feature
```

`git fetch --prune origin` を実行することで、フェッチ前にリモートに存在しなくなったリモートトラッキングの参照を削除するため、エラーが解消されます。

```
$ git fetch --prune origin
 - [deleted]         (none)     -> origin/feature
```

```
 * [new branch]            feature/foo        -> origin/feature/foo
```

壊れた ref を無視する警告を解消する

強制プッシュ後に `git branch -a` を実行すると、以下の警告が発生することがあります。

```
$ git branch -a
warning: ignoring broken ref refs/remotes/origin/HEAD
* main
  remotes/origin/main
```

処理はまだ理解していませんが、私は以下のコマンドを実行することで解消しました。

```
$ git symbolic-ref refs/remotes/origin/HEAD refs/remotes/origin/main
$ git fetch --prune
$ git gc
Enumerating objects: 663, done.
Counting objects: 100% (663/663), done.
Delta compression using up to 8 threads
Compressing objects: 100% (257/257), done.
Writing objects: 100% (663/663), done.
Total 663 (delta 392), reused 653 (delta 386), pack-reused 0
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

GitHub でフォーク元のリポジトリにアクセスする

リモートリポジトリにフォーク元のリポジトリを追加することで、アクセスできるようになります。

```
$ git remote add upstream {フォーク元のリポジトリの URL}
```

あとは `origin` と同様、`fetch` → `rebase` or `merge` などを実行すれば、フォーク元のリポジトリの変更を取得できます。

```
$ git fetch upstream
$ git checkout develop
$ git rebase upstream/develop
```

おまけ：エイリアスの設定

`.gitconfig` の `[alias]` セクションでエイリアスを設定できます。
私は `diff --cached` を `dfc` に設定するなど、短いコマンドで実行できるようにしています。
私の `.gitconfig` は以下にアップロードしており、随時更新しているので、よかったら参考にしてください。
https://github.com/uhooi/dotfiles/blob/master/.gitconfig

余談ですが、私は `.bashrc` で `git` を `g` で実行できるように設定しています。
つまり先ほどのコマンドは `g dfc` で実行できます。
https://github.com/uhooi/dotfiles/blob/master/.bashrc#L13

Git コマンドは補完が効くので、覚えていないうちはエイリアスを設定せず、ある程度覚えたら設定するのがオススメです。