**[git pull を強制し、リモートでローカルを上書きする方法](http://www-creators.com/archives/1097)**

[Git](http://www-creators.com/archives/category/git)

最終更新：2018-07-13 by Joe

git pull して、リモートブランチの最新に合わせようとしたら・・、あれ？コンフリクト・・？なにこれ、うまくいかない！「git push -f origin masterして強制Pushはできたのに。git pull -f origin master的な強制コマンドはないの？！」

とにかくリモートに合わせたい。そんなあなたのための、解決方法と解説です。

**目次**

* [1 git pull で、ローカルを強制上書きする方法](http://www-creators.com/archives/1097#git_pull)
* [2 git pull について、おさらい](http://www-creators.com/archives/1097#git_pull-2)
  + [2.1 git fetch origin mater について](http://www-creators.com/archives/1097#git_fetch_origin_mater)
  + [2.2 git merge origin/master について](http://www-creators.com/archives/1097#git_merge_originmaster)
* [3 git pullの失敗理由と、その解決](http://www-creators.com/archives/1097#git_pull-3)
  + [3.1 git mergeを強制する](http://www-creators.com/archives/1097#git_merge)
  + [3.2 一撃必殺、git reset --hard](http://www-creators.com/archives/1097#git_reset_w_hyphenhard)
* [4 ローカルの変更を保存しつつ、リモートの最新に強制的に合わせる](http://www-creators.com/archives/1097#i)
  + [4.1 ローカルの変更を保持しつつ、リモートで強制上書き](http://www-creators.com/archives/1097#i-2)
* [5 git pull の強制の関連情報](http://www-creators.com/archives/1097#git_pull-4)

**git pull で、ローカルを強制上書きする方法**

実際には「git pull --force」というオプションは存在しませんので、別のコマンドが必要になります。

**ローカルのmasterを、強制的にリモートのmasterに合わせる**

// 1) リモートの最新を取ってきておいて・・

$ git fetch origin master

// 2) ローカルのmasterを、リモート追跡のmasterに強制的に合わせる！

$ git reset --hard origin/master

Copy

「git pull の強制」というよりは、要は「reset」という方が正しいですね。

もちろん、git reset --hardは、手元にある作業ツリーとインデックスの変更内容は、すべてふっとんで消えてなくなりますので、実行前は注意して慎重に行って下さい。

一般的にgitでは、「コミットされていない変更」は、一度失うともう帰ってこないですので、不安な人は必ず実行前に、git statusして、作業ツリーの状態を確認して下さい。もし、作業ツリーとインデックスを別の場所に退避しておきたかったら「git stash」などがあります。

* [git stashで変更を一時的に退避する。git stash まとめ。](http://www-creators.com/archives/1583)

**git pull について、おさらい**

前提知識として、Git で「pull」というのは「fetch して、mergeする」を立て続けに行うためのコマンドでした。「push の反対だから pull 」って感じで名前が付けられたかと思うのですが、実際の動作が必ずしもきっちり対象に（反対に）なっていない点が、若干初学者の混乱のもとになっている気もします。

すなわち、いつものpullコマンド、

$ git pull origin master

Copy

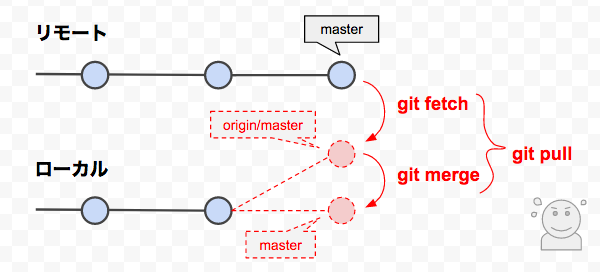
上記コマンドは、この２つのコマンドのショートハンドのようなものです。

$ git fetch origin master

$ git merge FETCH\_HEAD

Copy

※「FETCH\_HEAD」は、要は「origin/master」がマージされると解釈すれば問題無いです。fetch実行時、その後のマージ実行のため、取得したリモート側のブランチ名（refs）とコミット名との対を、.git/FETCH\_HEADに記述します（これはgitユーザは意識する必要はありません）



**git fetch origin mater について**

「git fetch origin master」は、リモートレポジトリ「origin」にあるブランチ「master」を、ローカルのリモート追跡ブランチに反映させる。という意味のコマンドです。

**リモート追跡ブランチ・・？**

「リモート追跡ブランチ」と聞いて「えーと、なんだっけ・・・」の人は、Git のリモートの概念をへの理解が浅い可能性がありますので、こちらも補足しておきます。

下記のコマンドを打てば、すべてのローカルブランチが表示されますが・・

// すべてのローカルブランチを表示する

$ git branch -a

Copy

一度でも git fetch origin master, もしくは、git pull origin master したことのあるローカルレポジトリ環境では、おそらくアウトプットは、のように表示するはずです。

\* master

remotes/origin/master

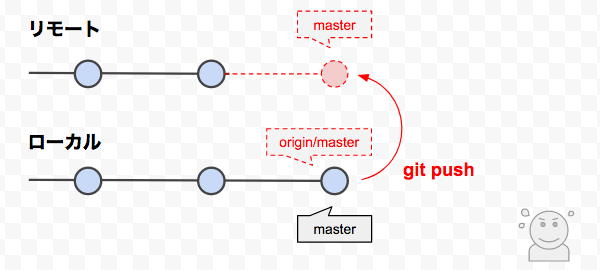
Copy

この「remotes/xxxx/xxxx」となっているブランチ名が、すなわち「リモート追跡ブランチ」です。

これはローカルにあるブランチの一種ですが、通常あなたがコミットを行う類のブランチではなく、常に対応するリモートブランチからfetch して更新される対象とするためのブランチとして区別されています。

gitマニュアルでは「A remote tracking branch」とされていますので「リモートトラッキングブランチ」とそのまま呼んだりもします。リモート追跡ブランチは、通常のローカルブランチのようにチェックアウトして作業する事はできません。

ちなみに、git push 成功時には、（上流ブランチとして設定されていれば）自動的に対応する追跡ブランチの参照が更新されます。



もし上流ブランチが設定されていない場合は「git push -u origin master」などのuオプションで設定できます。

**git merge origin/master について**

fetchしたら、あとはmergeです。

mergeコマンドにおいて、リモート追跡ブランチを指定することができます。

「git merge origin/master」は『「origin」という名前のリモートレポジトリ上にあるブランチ「master」の追跡ブランチ、すなわち「origin/master」を現在のブランチにマージする』という意味です。

**git pullの失敗理由と、その解決**

git pull origin materが失敗しているときは、おそらく origin/master　→ master のマージがコンフリクトしているときが多いはずです。

普通のチームの開発では、リモートのブランチを「正」とする事がほとんどだと思いますので、ローカルと噛み合わない場合「あ〜、めんどくさいからずぱっと強制したい・・」となるわけです。

**git mergeを強制する**

git mergeがコンフリクトする時は、コンフリクトを解決するのが教科書通りですが、もし完全に origin/master 側が「正」としているのであれば、「git merge --theirs」などというコマンドもありますが、そもそもマージによって、新しいコミットが作られてしまうので、これは選択肢ではありません。

ここでは、より強力な強制力を持つ「git reset 」を利用して、強制的に一致させます。

**一撃必殺、git reset --hard**

現在のブランチの状態を「強制的に」他のブランチの状態に合わせるときの方法として、「**git reset --hard** 」があります。reset --hard は、現在のブランチの状態を強制的に対象に合わせるコマンドでしたね。

git reset --hard <コミットへの参照>

Copy

resetコマンドは３種類が代表的です。

**git resetのモード**

| **コマンドとオプション** | **動作** |
| --- | --- |
| git reset --soft | 現在のbranchの先頭 だけをリセット |
| git reset git reset --mixed | 現在のbranchの先頭と、インデックスをリセット |
| git reset --hard | 現在のbranchの先頭、インデックスと作業ツリーを全部リセット |

以上です。

うまくgit pull を強制（？）できたでしょうか。

**ローカルの変更を保存しつつ、リモートの最新に強制的に合わせる**

おまけです。

上記の方法では、git reset --hard により、ワーキングツリーの状態を完全に破棄してしまいます。もし、破棄しないでおいてあとで確かめたり、変更内容を利用したい時、別途、残しておくことができます。

**ローカルの変更を保持しつつ、リモートで強制上書き**

// （１）ローカルの変更をコミットしておく。

$ git commit .

// （２）ローカルのmasterブランチの名前を変えておく

$ git branch -m prev-master

// （３）リモート追跡ブランチを最新にする。

$ git fetch origin

// （４）ローカルのmasterブランチを新規に作る

$ git checkout master

（４）の出力

Branch master set up to track remote branch master from origin.

Switched to a new branch 'master'

Copy

これまでのmasterブランチを名前を変え、新しくmasterブランチを作成します。（４）の時点で、自動的にリモート追跡ブランチ「origin/master」を起点にして新しいmasterブランチが生成されます。(指定されたブランチ名がリモートトラッキングブランチに見つかったときのgitによる補助的な振る舞いです)

さておき、たぶんこんなかんじになるはずです。

$ git log --all --decorate=full --graph --oneline

// 結果

\* HEAD -> refs/heads/master, refs/remote/origin/master

| \* refs/heads/prev-master

|/

\*

\*

Copy

これで、自分のローカル変更を別の名前で保存できたことになります。

ともすれば、「自分の実の子であるmasterちゃんを戸籍変更して、別の女との間に新しくできた子に、前の子の名前をつけて、その子として育てる」という、なんだか推理サスペンス的な操作にも見えます。一抹の後ろめたさを感じてしまいます。

もちろん、コミットを作らなくても、git stashを使っても大丈夫です。でも、ブランチにしたほうが、なんだかクリーンで、管理しやすかったりもします。そのへんは好みや状況に応じてどうぞ。

以上です。

**git pull の強制の関連情報**

リモートのブランチや、リモート追跡ブランチとローカルとの操作は、最初慣れるまで戸惑う事が多いです。こちらの記事も思い当たる方は是非どうぞ。

* [gitでリモートブランチを削除するコマンド](http://www-creators.com/archives/1062)
* [git commitしたコミットを取り消す、やり直す、修正する。](http://www-creators.com/archives/1116)
* [git resetを取り消して、元に戻す方法。](http://www-creators.com/archives/2025)

こちらは、git pull について全般的にまとめた記事です。

* [git pull コマンドの使い方と、主要オプションまとめ](http://www-creators.com/archives/2295)

**gitコマンド 備忘録**

[Git](https://qiita.com/tags/Git)

32

この記事は最終更新日から1年以上が経過しています。

**はじめに**

タイトル通り、「あれって何だったっけな？」の時のための自分用メモ。  
記載コマンドは -help した時の表記となるべく同じようになるように記述しておく。

更新は随時行っていく。

**ブランチ操作**

**リモートブランチを作成/更新**

**チェックアウト中のローカルブランチをそのままpushする場合**

git push <repository> <refspec>

**リモートブランチを指定してpushする場合**

git push <repository> <localbranch>:<remotebranch>

**ローカルブランチの作成**

**ブランチを作成すると同時にチェックアウト**

git checkout -b <branch> [<repository>/]<branch>

 git checkout -B **で実行すると、同名のブランチが既に存在していた場合は上書きされてしまうので注意**

**upstreamの設定をしながらブランチを作成する**

git branch -t <branchname> [<repository>/]<target\_branchname>

**空のブランチを作成する**

git checkout --orphan <newbranch>

**ブランチの削除**

**ローカルブランチを削除**

git branch -d <branchname>

**ローカルブランチを強制的に削除**

git branch -D <branchname>

**リモートリポジトリーからブランチを削除**

git push --delete <repository> <refspec>

**ブランチにupstreamを設定する**

git branch -u <target\_branchname>

**ブランチのリネーム**

**通常**

git branch -m <oldbranch> <newbranch>  
 コミットしていない変更がある場合はリネームできない

**強制的にリネーム**

git branch -M <oldbranch> <newbranch>

**リモート操作**

**現在のリモートリポジトリーの状態を表示**

git remote show <repository>

**リモートリポジトリーに関する情報を更新**

**テスト実行**

git remote prune --dry-run <repository>

**反映**

git remote prune <repository>

**リモートブランチの情報を更新**

**通常**

git fetch <repository>

**ブランチの削除情報も含めて更新**

git fetch --prune <repository>

**tag**

**作成**

**コメント付き**

git tag -a <tag\_name> (<target\_branch>)

**リモートリポジトリーに反映**

git push <repository> (<tagname>|refs/tags/<tagname>)

**対話的ステージング**

git add -i

**commit**

**ステージされている変更をコミット**

git commit

**全ての変更をコミット**

git commit -a

**stash**

**保存**

git stash save <message>  
 コマンドラインからの入力になってしまうので、日本語が化ける

**一覧表示**

git stash list

**反映**

**current branchに反映してstashを削除**

git stash pop stash@{0}

**current branchに反映してstashを保持**

git stash apply stash@{0}

**削除**

**指定したstashを削除**

git stash drop stash@{0}

**全てのstashを削除**

git stash clear

**既にバージョン管理されているファイルの変更を無視したい(update-index)**

**assume-unchanged**

**設定する**

git update-index --assume-unchanged <target\_file>

**元に戻す**

git update-index --no-assume-unchanged <target\_file>

**設定状態の確認**

git ls-files -v | grep ^h  
 *assume-unchanged* 設定のファイルは状態が **小文字** で表示される

**skip-worktree**

**設定する**

git update-index --skip-worktree <target\_file>

**元に戻す**

git update-index --no-skip-worktree <target\_file>

**設定状態の確認**

git ls-files -v | grep ^S  
 *skip-worktree* 設定のファイルは状態が **S** と表示される

**※assume-unchanged と skip-worktree の違い※**

**assume-unchanged**

そのファイルが作業ツリー上で変更されている場合でも git はその変更を無視して変更されていないとみなす。

* merge の時にマージ元の内容で上書きされる。
* reset --hardした時も上書きされる。

**skip-worktree**

そのファイルが作業ツリー上で変更されている場合には git はその変更を保つ。

* merge の時にローカルの状態が優先されて保持される。
* reset --hardしたとしても状態は保持される。

**その他のコマンド**

**コミットしてあるけどキャッシュから削除して最初から存在しなかったことにする**

git rm --cached ./

**特定のモジュールを存在しなかったことにする**

**ファイル**

git filter-branch --tree-filter 'rm -f <file>' HEAD

**ディレクトリ**

git filter-branch --tree-filter 'rm -rf <dir>' HEAD

**graph**

git log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n'' %C(white)%s%C(reset) %C(dim white)- %an%C(reset)' --all

**conflict発生時に、コマンドラインからサクッとマージ**

 マージをするというよりは、競合が起きたモジュールをどちらかのコミットに完全に寄せてしまう対処の仕方

**マージ元を採用**

git checkout --theirs .

**マージ元を採用 -ファイル単位-**

git checkout --theirs <file>

**マージ先(workディレクトリ)を採用**

git checkout --ours .

**マージ先(workディレクトリ)を採用 -ファイル単位-**

git checkout --ours <file>