



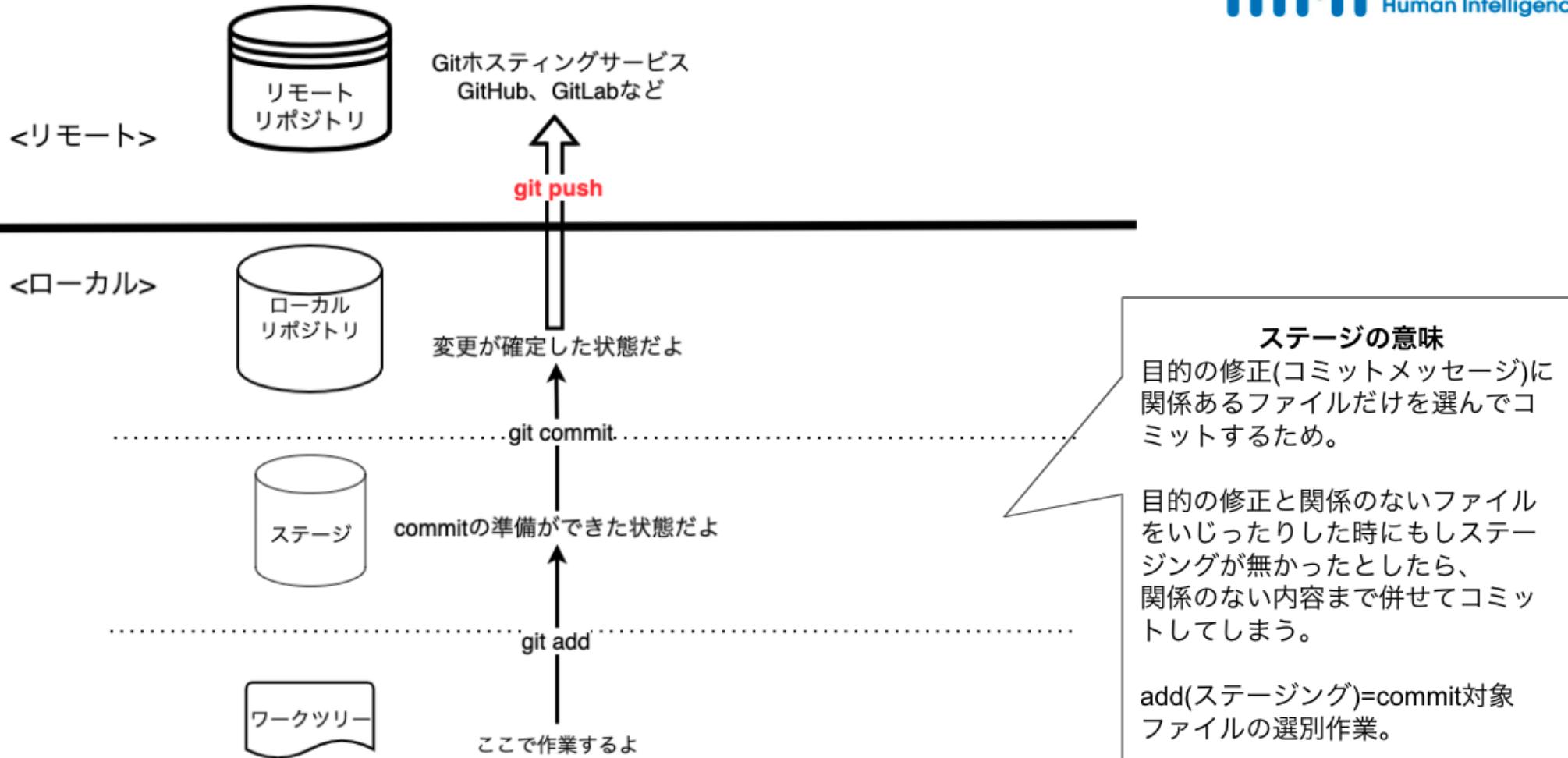
# Git 勉強会 3 日目

基本コマンド & ハンズオン ② ~チーム開発編~  
2022/03/28

# ここまでまとめ:リポジトリの構成(1日目)



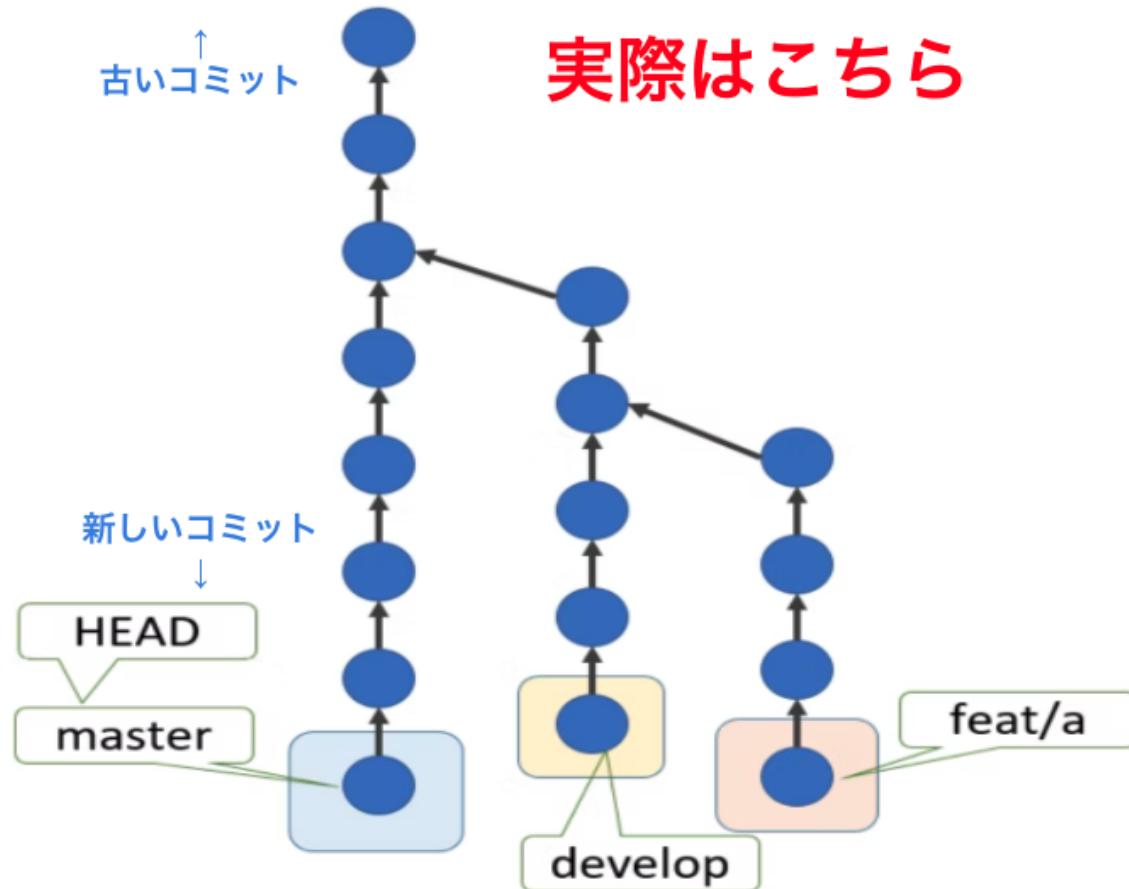
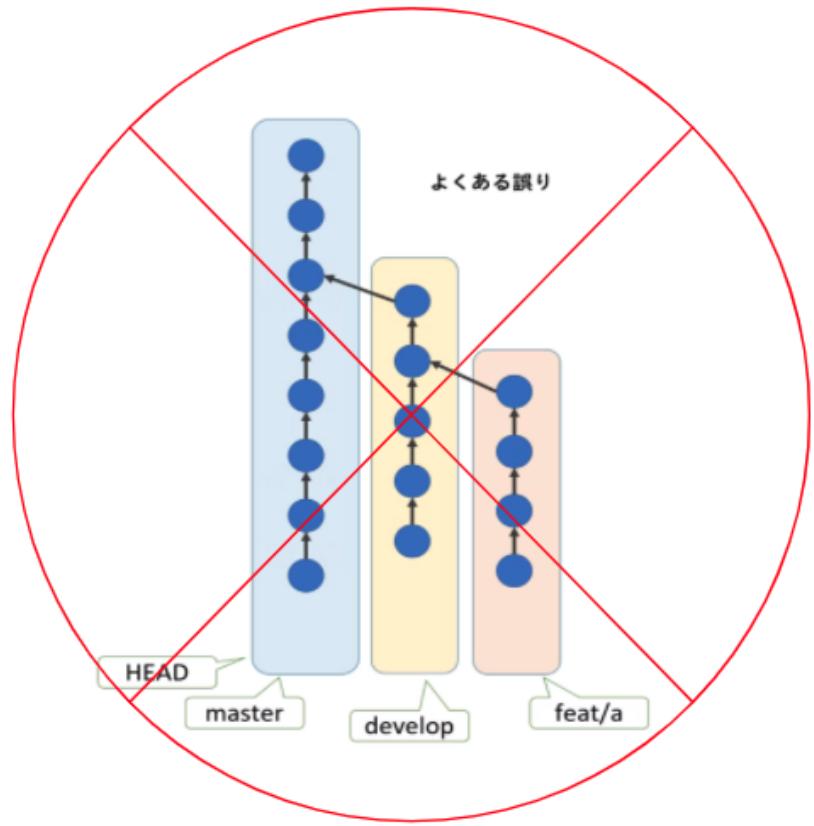
## 1.b. リポジトリの構成



# ここまでまとめ: ブランチ・HEAD の実態(1日目)



## 1.d. ブランチ、HEADとは



ブランチとはコミットを指すポインタに過ぎない

[GitのHEADとは何者なのか - Qiita](#) より

## ここまでまとめ: Git コマンド ~個人開発編~(2日目)

- clone ★
- config ★
- init
- remote
- branch ★
- switch (checkout) ★
- status ★
- add ★
- commit ★
- push ★
- mv
- rm
- log ★
- diff

# クイズ

Q. 次のユースケースではどんなコマンドを実行すればいいでしょうか？

1. 新しいブランチ feature を作成し、そのまま移動したい
2. git merge を実行したらコンフリクトしてしまった。状況を確認し、次に取るべき行動の選択肢を知りたい
3. 変更した内容を記録(コミット)したい。同時にコミットメッセージも付けたい
4. 直前のコミットメッセージを修正したい(やや難)

# クイズ(解答例)

Q. 次のユースケースではどんなコマンドを実行すればいいでしょうか?

# 1. 新しいブランチを作成し、そのまま移動したい

\$ git switch -c feature # git checkout -b feature でも可

# 2. git merge を実行したらコンフリクトしてしまった。状況を確認し、次に取るべき行動の選択肢を知りたい

\$ git status

# 3. 変更した内容を記録（コミット）したい。同時にコミットメッセージも付けたい

\$ git commit -m "任意のメッセージ"

# 4. 直前のコミットメッセージを修正したい（やや難）

\$ git commit --amend -m "修正後のメッセージ"

# 本日のゴール

頭の中に「こんなときはこうする」というインデックスをぼんやりと作ること

- Git の各サブコマンドの存在を知ること
- 各サブコマンドのユースケースを知り、Git で躊躇したときに本資料を見返そうと思い付けること

→ この資料は辞書として使っていいってほしいので、完全に理解しようとしないで OK です！



→手を動かしてブランチを指定するより (Learn Git Branching にアリレンジ)

- コマンド説明(★付きコマンドのみ当日説明します)

- pull ★
- fetch ★
  - 主なオプション
- merge ★
- rebase ★
- コラム:merge と rebase の違い
- コラム:merge と rebase はどっちがいい?
- コラム:Fast-forward merge と Non Fast-forward merge
- stash ★
- restore (checkout) ★
- reset ★
- revert ★
- コラム:reset と revert の違い
- コラム:reset と revert はどっちがいい?
  - 参考
- コラム:HEAD^ と HEAD~ の違い
- cherry-pick ★
- blame
- tag
- reflog

- ハンズオン

- 【参考】ターミナルでよく使うコマンド ①
- 【参考】ターミナルでよく使うコマンド ②

# 3日目アジェンダ(今日はこっち)

- Git 基本コマンド ② ~チーム開発編~
  - fetch ★
  - merge ★
  - rebase ★
  - pull ★
  - stash ★
  - restore (checkout) ★
  - reset ★
  - revert ★
  - cherry-pick ★
  - blame
  - tag
  - reflog

# 復習&予習タイム ~今日のコマンドをより理解するため に~

- origin develop と origin/develop の違い
- pull と fetch + merge の違い
- ブランチとは

## 参考記事:

- [git pull と git pull –rebase の違いって?図を交えて説明します! ★ とても分かりやすいのでおすすめ](#)
- [Git で「追跡ブランチ」って言うのやめましょう - Qiita](#)
- [origin master と origin/master の違い - Qiita](#)



## クイズ:origin develop と origin/develop の違い

以下のコマンドで指定している対象ブランチ、何が違うか分かりますか？

### ケース 1. fetch + merge で指定する対象ブランチ

```
$ git fetch origin develop # fetch では origin develop を指定してるけど  
$ git merge origin/develop # merge では origin/develop を指定している・・・?
```

### ケース 2. merge で指定する対象ブランチ

```
$ git merge origin/develop # このコマンドと
```

```
$ git merge develop # このコマンドの意味の違いは・・・?
```

違い分かりますか？

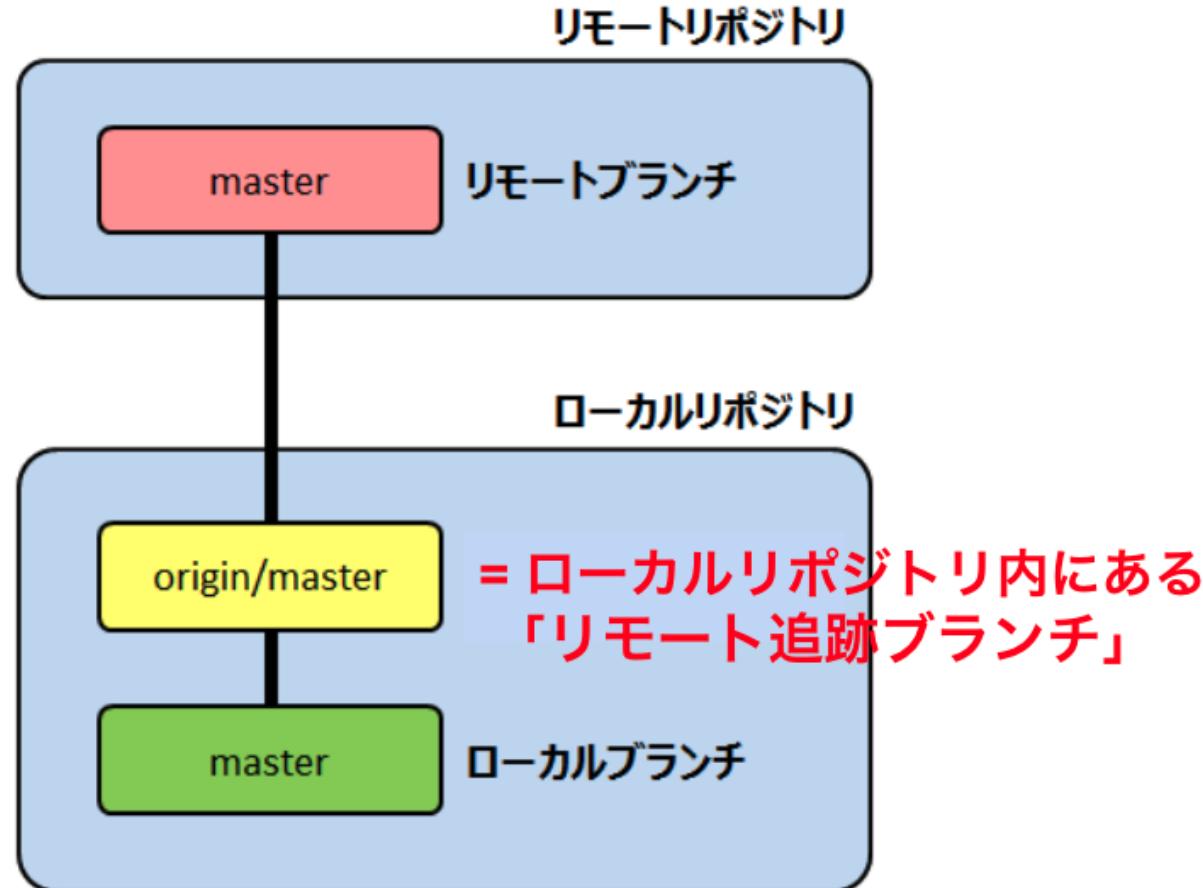
# 復習: origin(リモートリポジトリ)と origin/develop(リモート追跡ブランチ) git

## 1.e. originとは



もう一度確認

“origin”  
= リモートリポジトリに  
付けた名前



[gitのfetchとpullの違いについて: 小糸空間](#) より

## クイズ:origin develop と origin/develop の違い(解答)

origin develop

- origin という名前で管理しているリモートリポジトリの develop ブランチ

origin/develop

- リモートリポジトリ origin の develop ブランチを追跡する、ローカルリポジトリ内にあるリモート追跡ブランチ

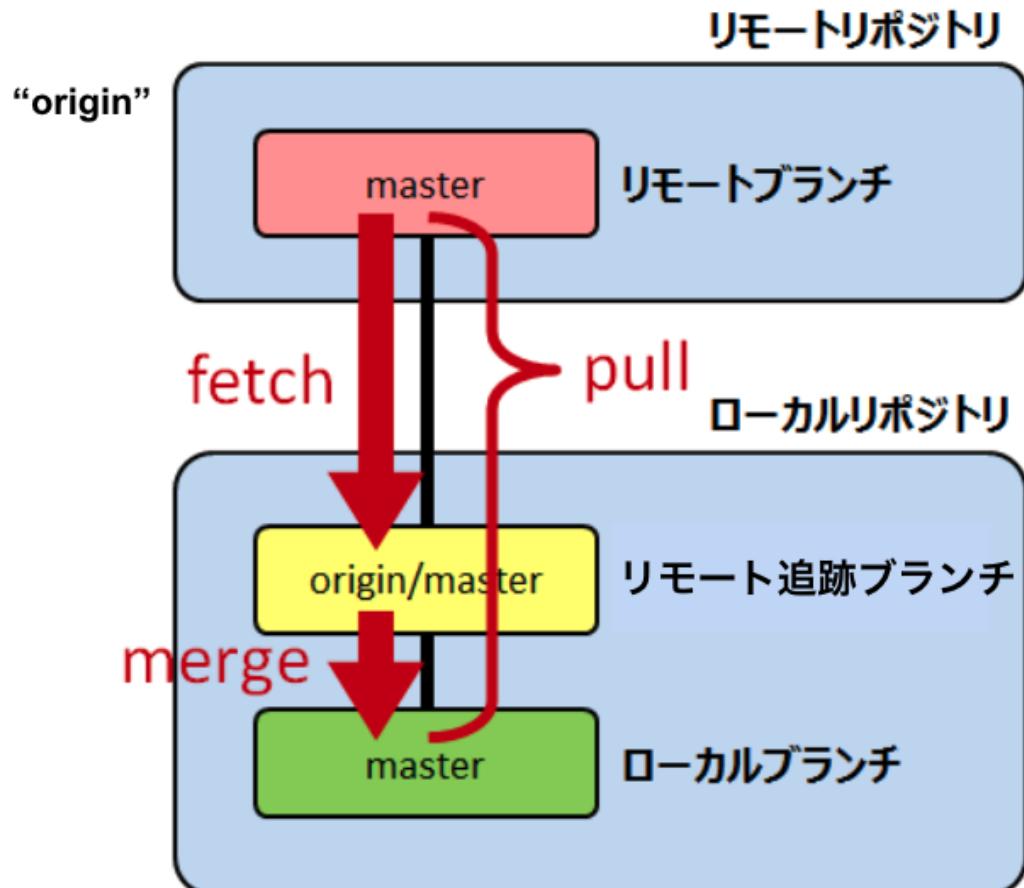
# 復習:pull と fetch + merge の違い



## 1.e. originとは



リモート追跡ブランチ (origin/<ブランチ名>) の役割



リモート追跡ブランチのおかげで、fetchコマンドで最新のリモートの内容をリモート追跡ブランチに取り込んで確認できる

リモート追跡ブランチが無ければ、リモートブランチの内容を確認できずにローカルブランチに取り込むこととなる

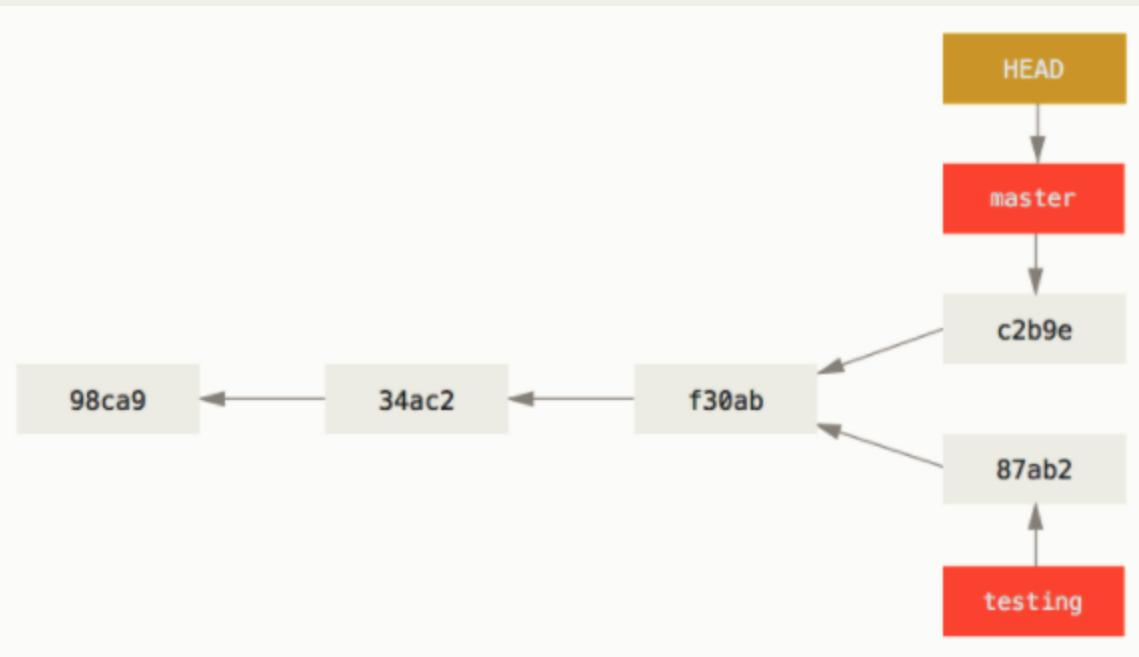
[gitのfetchとpullの違いについて: 小糸空間](#) より

# ブランチとは

- コミット履歴を分岐する機能。1つのプロジェクトからブランチを分岐させることにより、プロジェクト本体に影響を与えるずに複数機能を安全に並行開発することが可能となる

## ブランチの実態

- 「コミットを指すポインタ」のこと。ただ単に特定のコミット ID を指差しているだけ

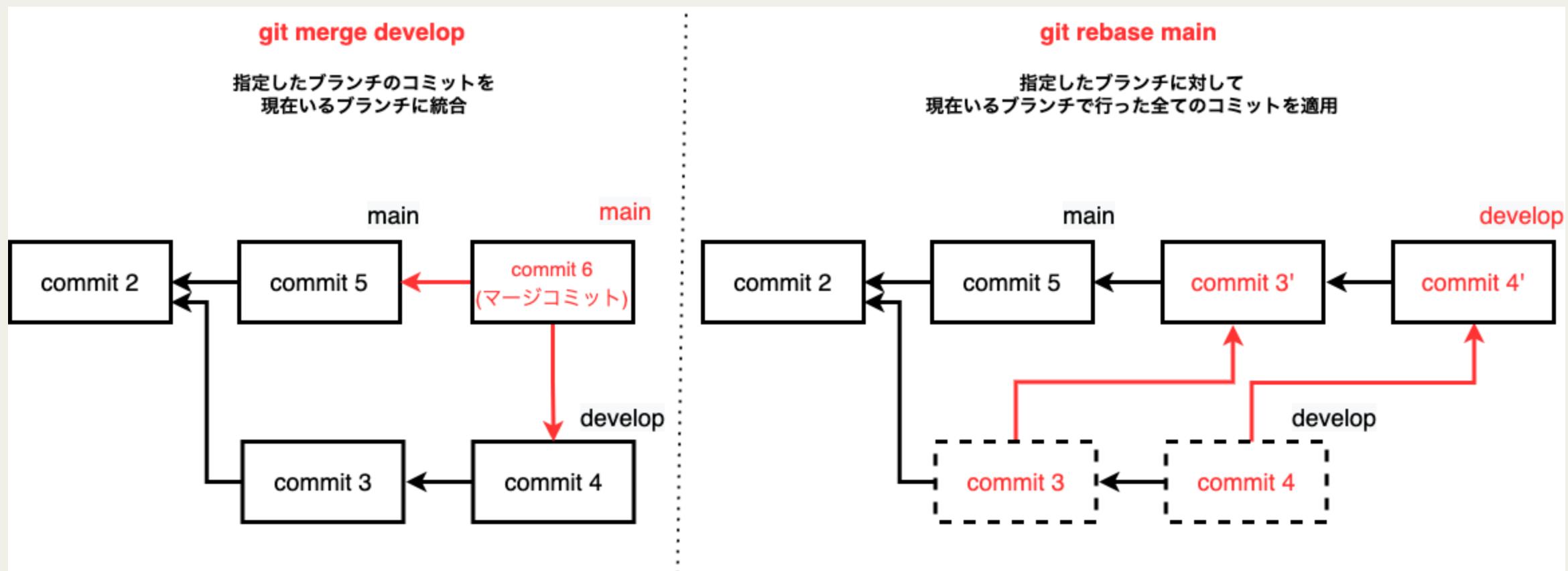


# マージとは

- 分岐したコミット履歴を統合する機能

## マージの種類

統合の手段には merge と rebase がある(後述)



## 手を動かしてイメージを掴んでみよう (Learn Git Branching にチャレンジ)

ブランチの枝分かれ・統合やコミットの動きをイメージするため、以下をみんなでやってみましょう！

- [Learn Git Branching](#)
  - 「1: Git のコミット」の 1~4 までみんなでやりましょう

ここでは、下記の種類の git コマンドを学ぶことができます。

- commit
- branch
- checkout (switch)
- merge
- rebase

コマンド説明(★ 付きコマンドのみ当日説明します)

# pull ★

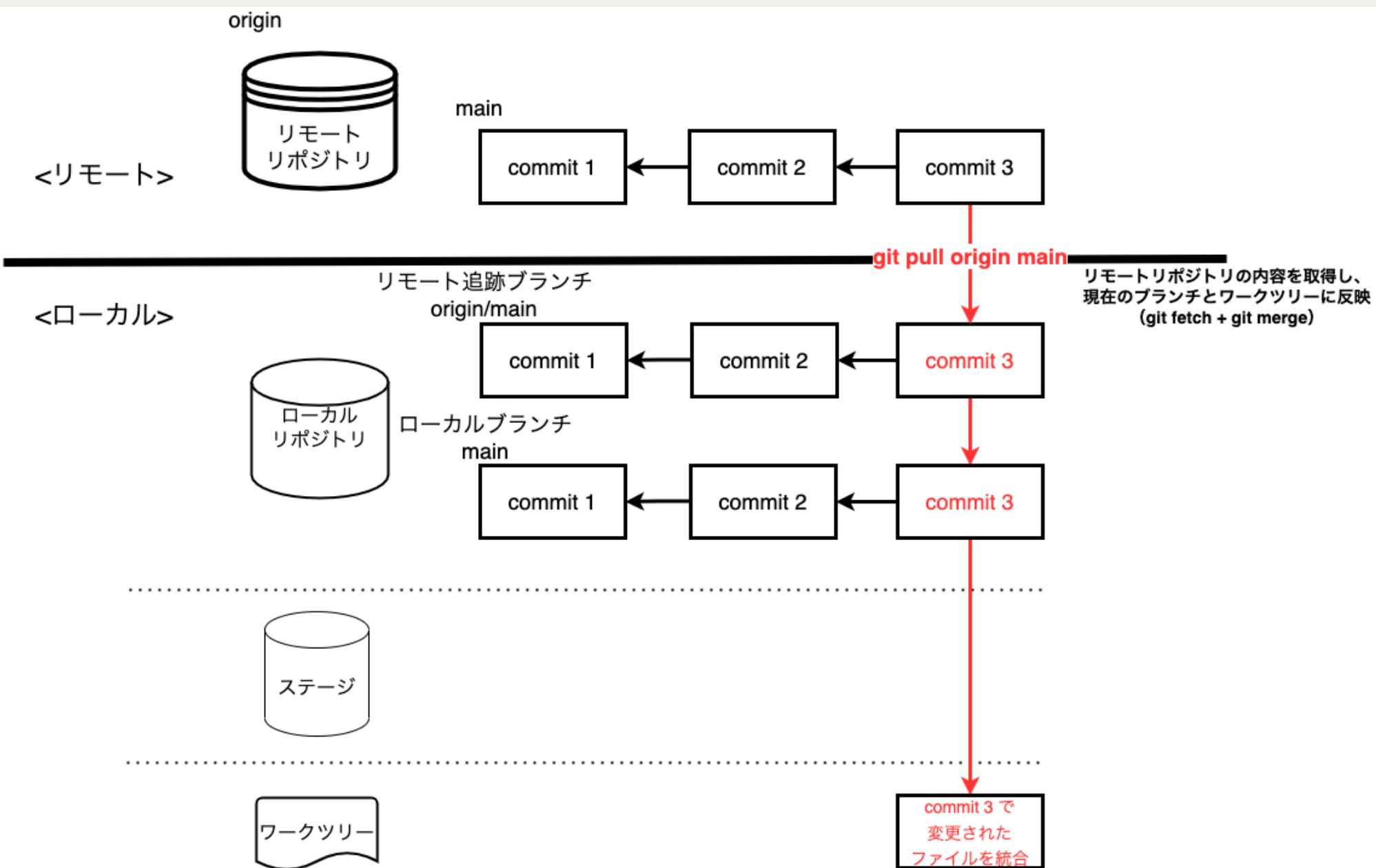
## 機能

- リモートリポジトリの内容を取得し、現在のブランチに取り込む( `git fetch` + `git merge` )

## ユースケース

- リモートリポジトリの最新情報をローカルリポジトリに取り込みたい

# イメージ





## 主なオプション

- `-r | --rebase` : フェッチ後に現在のブランチを上流ブランチの上にリベース
  - `git fetch + git rebase`

## コマンド例

```
$ git switch <ブランチ名> # まず pull したいブランチへ切り替える
$ git pull origin <ブランチ名> # リモートリポジトリの内容を取得してマージ。上流ブランチを設定している場合は git pull で OK
$ git pull -r origin <ブランチ名> # リモートリポジトリの内容を取得してリベース。
```

## 参考

- [git-pull – Git コマンドリファレンス\(日本語版\)](#)
- [git pull コマンドの使い方と、主要オプションまとめ](#)
- [【初心者向け】git fetch、git merge、git pull の違いについて - Qiita](#)
- [git pull と git pull –rebase の違いって?図を交えて説明します!](#)
- [Git のコミットメッセージを後から変更する方法をわかりやすく書いてみた](#)

# fetch ★

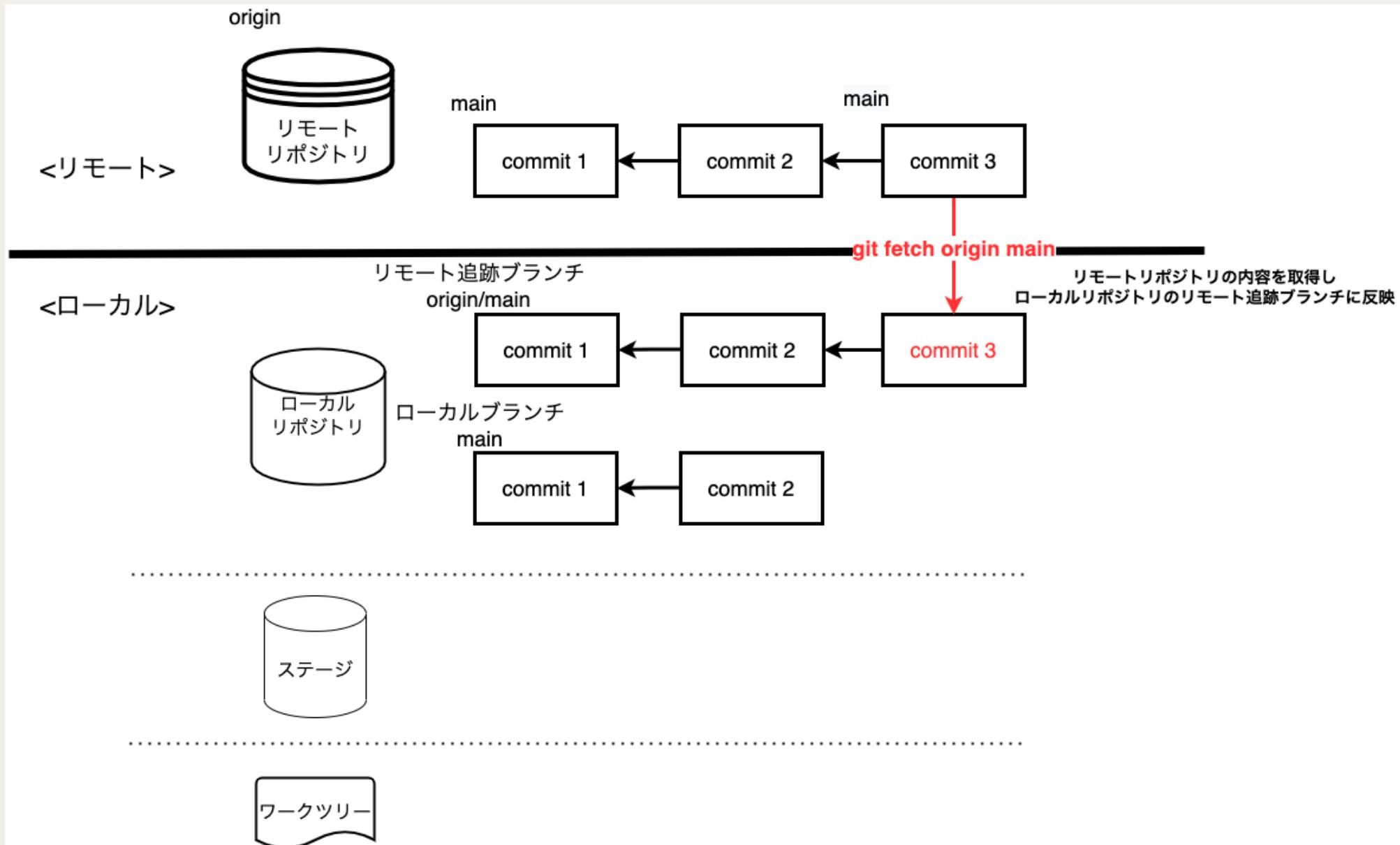
## 機能

- リモートリポジトリの内容を取得する
  - 取得した内容はローカルのリモート追跡ブランチに反映するが、ワークツリーには反映しない

## ユースケース

- 他者が作成したブランチに切り替えたいので、リモートリポジトリの最新状態を取得したい
- pull だとワークツリーまで更新してしまうので、とりあえずリモートの状態を確認したい

# イメージ



## 主なオプション

- `-p | --prune` : リモートリポジトリで削除されたブランチをローカルリポジトリのリモート追跡ブランチに反映
  - 常に `--prune` を有効にする場合は `git config --global fetch.prune true` を実行して設定を更新
- `--all` : すべてのリモートリポジトリの内容を取得
  - リモートリポジトリが `origin` しかない場合は使うことはない

## コマンド例

```
$ git fetch # リモートリポジトリの内容をローカルのリモート追跡ブランチに反映  
$ git fetch --prune # リモートリポジトリで削除されたブランチをローカルリポジトリに反映
```

## 参考

- [git-fetch – Git コマンドリファレンス\(日本語版\)](#)
- [git pull と git pull –rebase の違いって?図を交えて説明します!](#)
- [Git で「追跡ブランチ」って言うのやめましょう - Qiita](#)
- [リモートで消されたブランチが手元で残ってしまう件を解消する - Qiita](#)
- [git pull のとき常に prune するための設定 - Qiita](#)

# merge ★

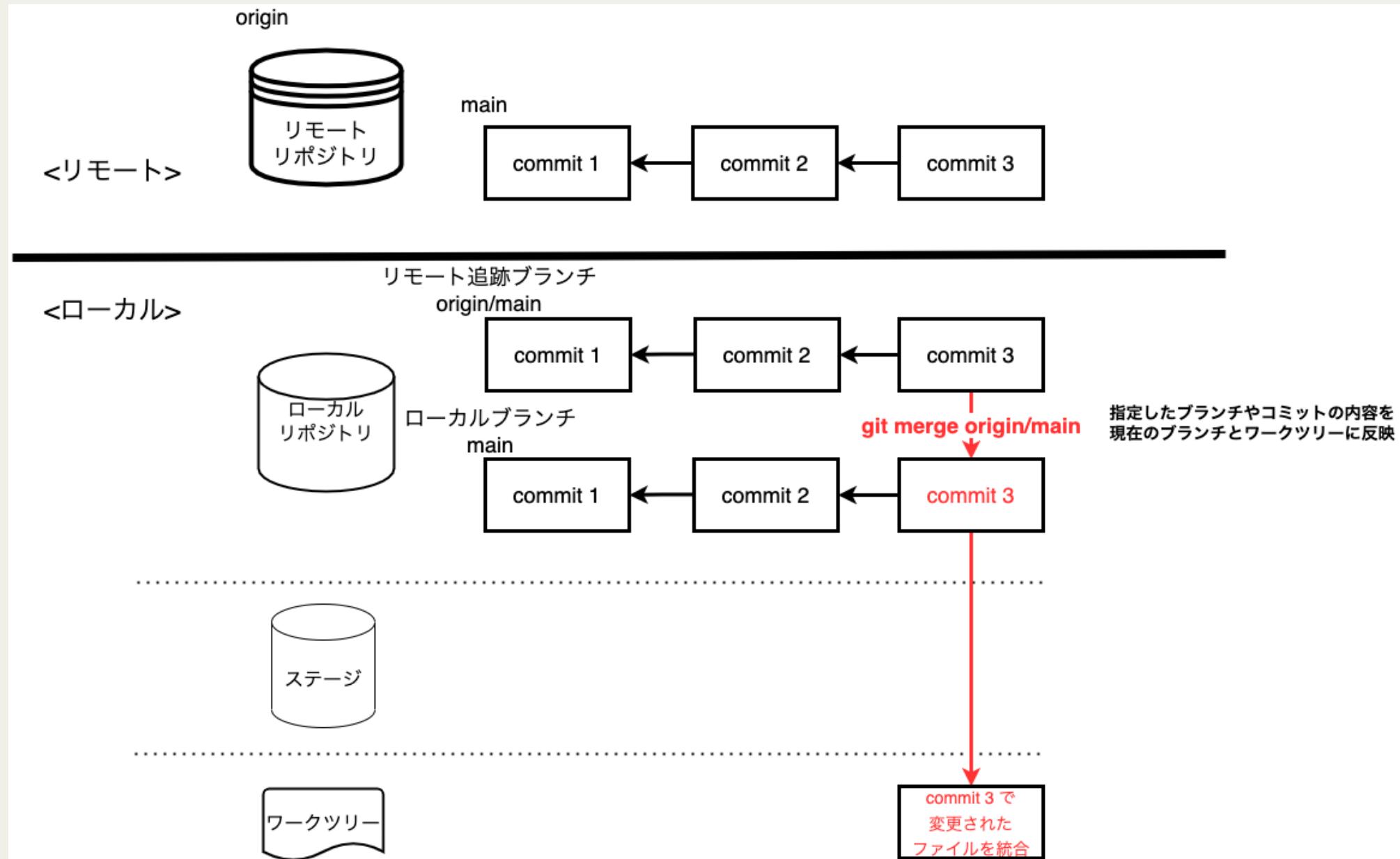
## 機能

- 他のブランチやコミットの内容を現在のブランチに取り込む

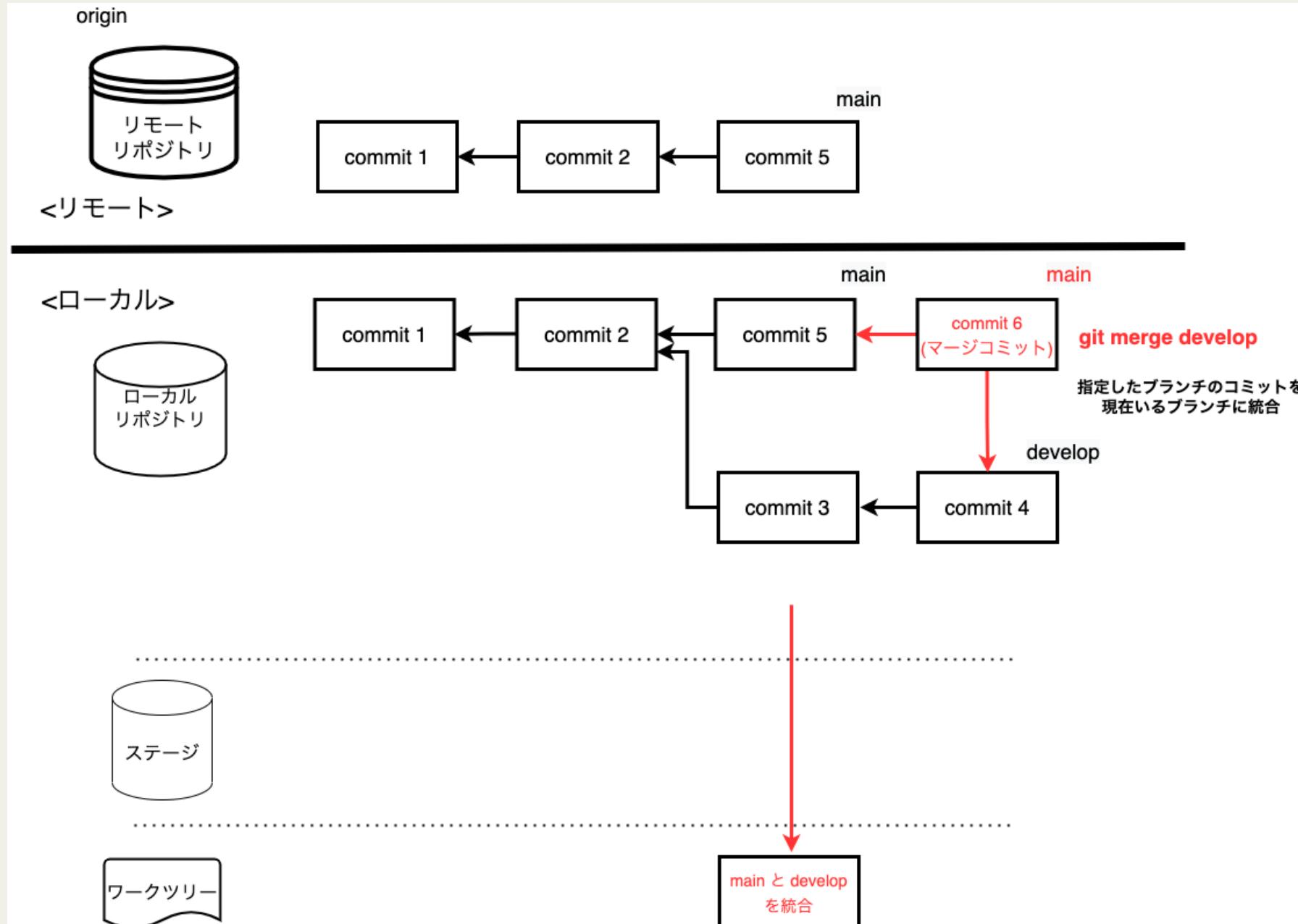
## ユースケース

- 2つのブランチを1つに統合したい。どのブランチでどんな変更があったかという記録はコミット履歴に残したい
- fetchした内容をワークツリーに反映したい。どのブランチでどんな変更があったかという記録はコミット履歴に残したい
- マージ時にコンフリクトしたので、競合を解決したい

## イメージ (1. リモート追跡ブランチをマージ)



## イメージ (2. ブランチをマージ)



## 主なオプション

- `--no-ff` : fast-forward merge が発生する条件であっても必ずマージコミットを作成
- `--continue` : コンフリクト解決後、マージを続行
- `--abort` : コンフリクト解決を中止し、マージ前の状態に再構築

## コマンド例

```
$ git switch <マージ先のブランチ名> # マージ先のブランチに移動
$ git merge <マージ元のブランチ名> # 指定したブランチを現在いるブランチに取り込み
$ git merge --no-ff <マージ元のブランチ名> # fast-forward merge が発生する条件であっても必ずマージコミットを作成
```

## 参考

- [git-merge – Git コマンドリファレンス\(日本語版\)](#)
- [git pull と git pull –rebase の違いって?図を交えて説明します!](#)
- [git の merge --no-ff のススメ](#)

# rebase ★

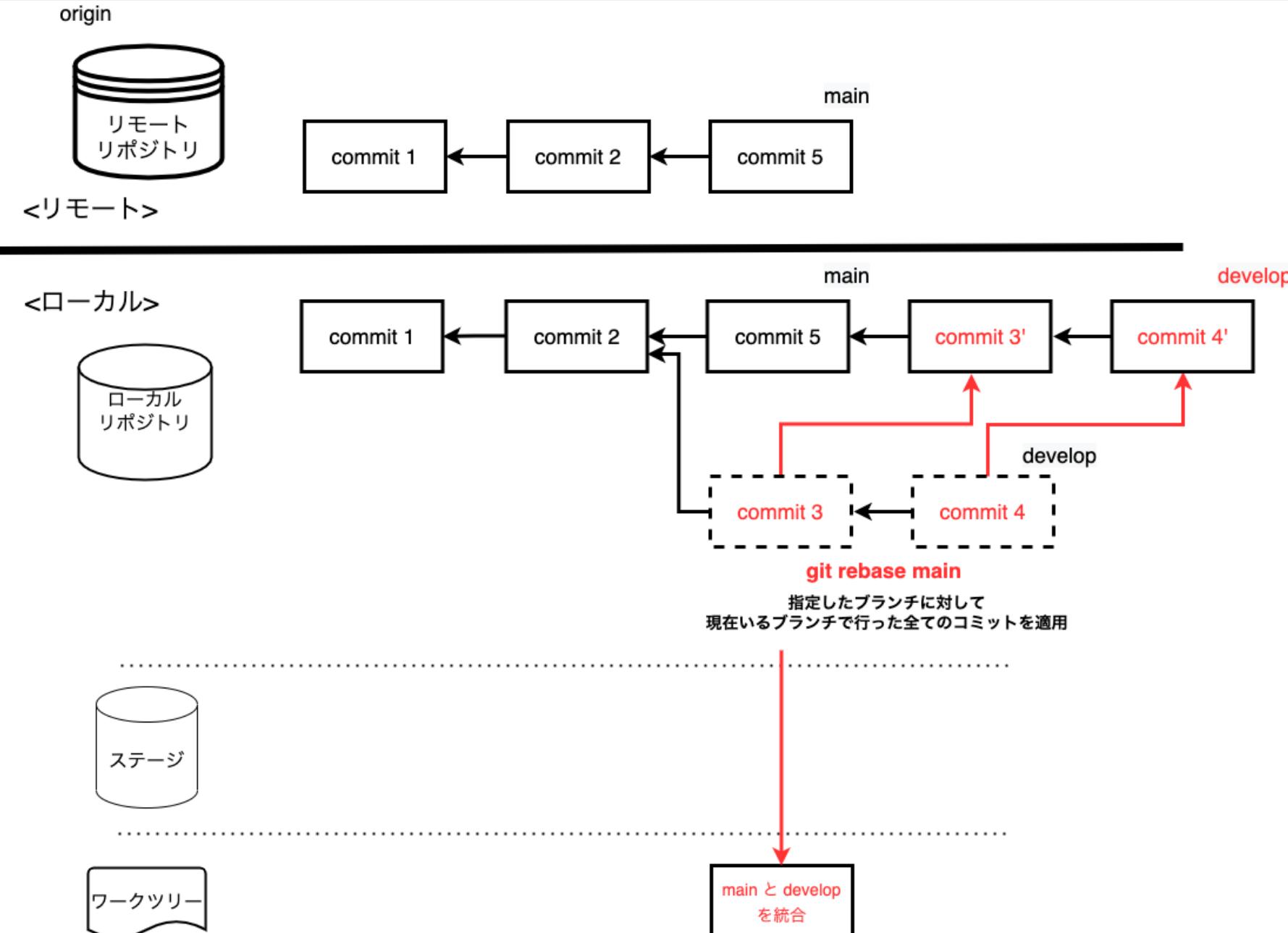
## 機能

- コミットを再適用する(ブランチの分岐点を変更したり、コミットの順番を入れ替えたりできる)
  - re(変える) + base(基点)=コミットの基点を付け変える

## ユースケース

- 2つのブランチを1つに統合したい。コミットログはきれいに(一直線に)保ちたい(マージコミットを作りたくない)
- fetchした内容をワークツリーに反映したい。コミットログはきれいに(一直線に)保ちたい
- リベース時にコンフリクトしたので、競合を解決したい
- 過去のコミットを編集したい
  - 複数のコミットをひとつにまとめたい
  - 2つ以上前のコミットを修正したい

# イメージ



## 主なオプション

- `--continue` : コンフリクト解決後、リベースを続行
- `--abort` : コンフリクト解決を中止し、リベース前の状態に再構築
- `-i` | `--interactive` : 対話モードで過去のコミットを編集(コミットメッセージの修正やコミットの統合など)

## コマンド例

```
$ git switch <付け替えたいのブランチ名>
$ git rebase <マージ元のブランチ名> # 指定したブランチに対して現在いるブランチで行った全てのコミットを適用
$ git rebase -i HEAD~4 # 最新から 4 つ分のコミットを修正・統合
```

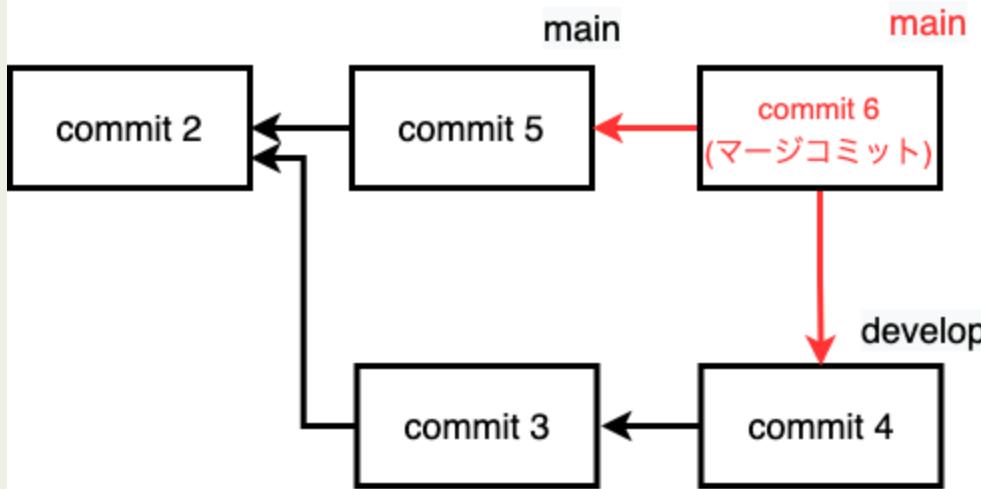
## 参考

- [git-rebase – Git コマンドリファレンス\(日本語版\)](#)
- [git pull と git pull –rebase の違いって?図を交えて説明します!](#)
- [git rebase を初めて使った際のまとめ - Qiita](#)
- [rebase -i でコミットをまとめる - Qiita](#)
- [【やっとわかった!】git の HEAD^ と HEAD~ の違い - Qiita](#)
- [git rebase 失敗した時の対処法 - Qiita](#)

# コラム:merge と rebase の違い

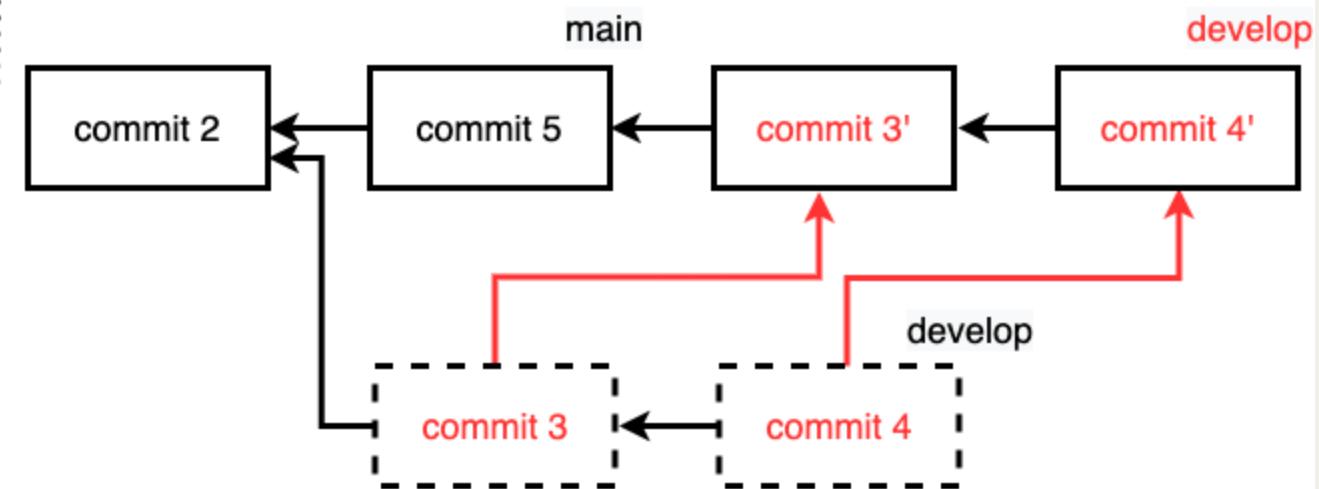
**git merge develop**

指定したブランチのコミットを  
現在いるブランチに統合



**git rebase main**

指定したブランチに対して  
現在いるブランチで行った全てのコミットを適用



## イメージ

- merge: 現在のブランチの上に他のブランチの内容を取り込む
- rebase: 取り込みたいブランチの上に今のブランチの内容を乗せる

# コラム:merge と rebase はどっちがいい?

A. 慣れないうちは merge の方が無難。あとはチームの運用方針次第。

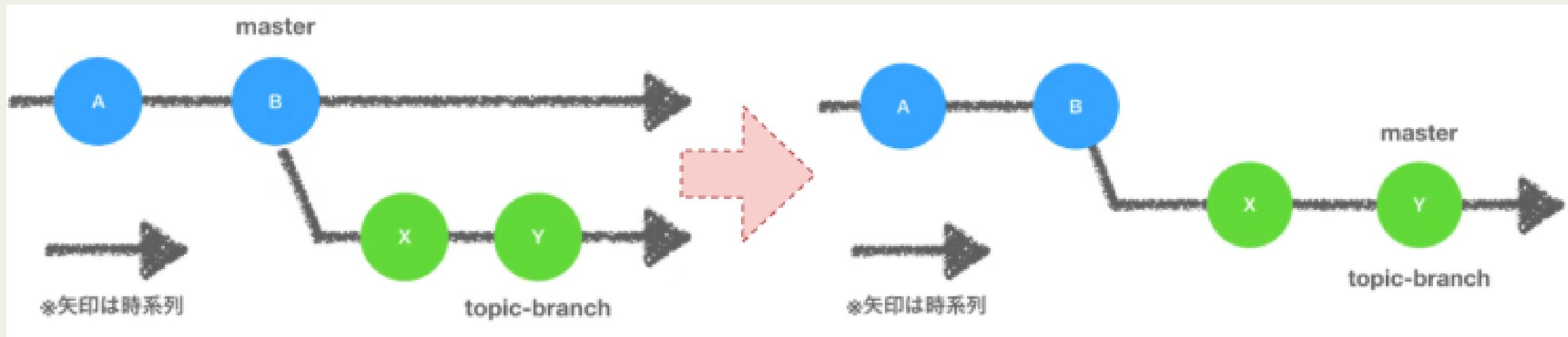
-	merge	rebase
メリット	<ul style="list-style-type: none"> <li>- どのブランチからどんなコミットを取り込んだのか履歴を追える(merge no-ff を強制する運用の場合)</li> <li>- MR(PR) レビューの文脈が残る</li> </ul>	<ul style="list-style-type: none"> <li>- 履歴を一直線に保つことができる</li> </ul>
デメリット	<ul style="list-style-type: none"> <li>- マージコミットがあると履歴が見づらい (<code>git log --no-merges</code> でマージコミットを除いて履歴表示すれば解決?)</li> </ul>	<ul style="list-style-type: none"> <li>- リモートに push 済の変更を rebase した場合は force-push が必須</li> <li>- コミッター・コミット ID が変わってしまう</li> <li>- コンフリクトの解決が煩雑(コミットを 1 つずつ適用し、それぞれで解決が必要なため。小さい単位でコンフリクト解決する方が分かりやすいという意見もある)</li> </ul>

## 参考

- あなたは merge 派?rebase 派?綺麗な Git ログで実感したメリット (記事に対する @kazuho さんのコメント)
- Git の rebase と merge の挙動の違いを GitHub を用いて検証してみた - Qiita
- Github での Web 上からのマージの仕方 3 種とその使いどころ - Qiita
- なぜ git rebase をやめるべきか
- 【Git】将来の自分を救うのは、rebase だと僕は思うよ

## Fast-forward merge

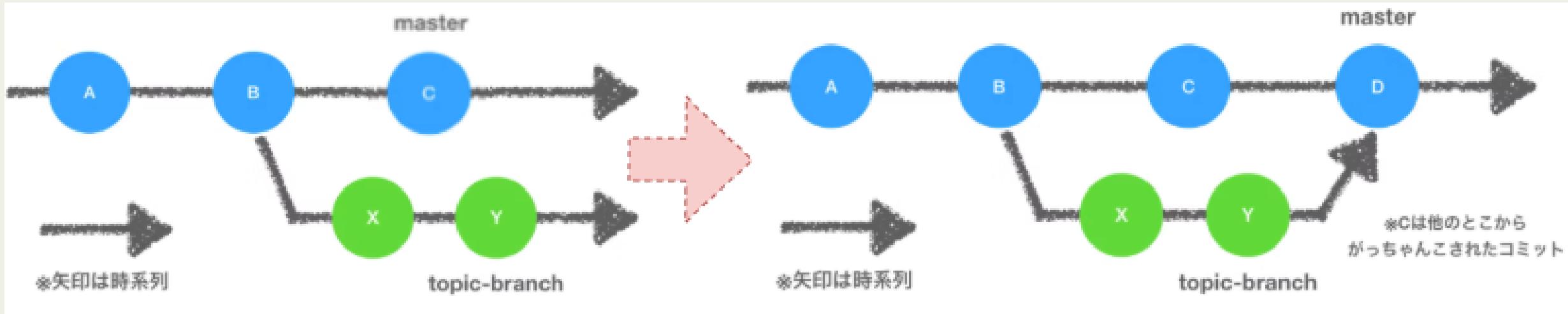
- 早送りマージ
- 「マージされるブランチがマージの基点にいるとき」に発生する
- Git はマージ作業を省略し、マージされるブランチのポイントをマージするブランチに移動するだけとなる
  - つまり、マージする際に新しいコミット(=マージコミット)を作成しない
- メリットは、絶対にコンフリクトが起きない(マージ作業をしていないので)
- デメリットとして、「ブランチをマージした」という履歴が残らない。(マージを間違えて元に戻すときに苦労する)
- `git merge --no-ff` オプションを付けると Fast-forward が発生する条件でも Non Fast-forward となる
  - (オプション付け忘れ防止のため `git config --global --add merge.ff false` と設定するといい)



## Non Fast-forward merge



- 早送りじゃないマージ(マージコミットを作成)
- 「マージされるブランチがマージの基点にいないとき(既に枝分かれしているとき)」に発生する
- 「どのブランチからどんな変更(どのコミット)をマージしたか」をコミット履歴として持つ
- 間違えて意図せぬブランチをマージしてしまった場合、マージコミットを取り消すだけで戻せる
- マージ対象のそれぞれのコミットで同じ箇所の修正がなければ Auto merge, あれば Conflict となる



(Git の rebase と merge の挙動の違いを GitHub を用いて検証してみた - Qiita より引用)

## 参考

- 3.2 Git のブランチ機能 - ブランチとマージの基本
- git merge の動作仕様と、主要オプションのまとめ
- Git の rebase と merge の挙動の違いを GitHub を用いて検証してみた - Qiita
- こわくない Git

# stash ★

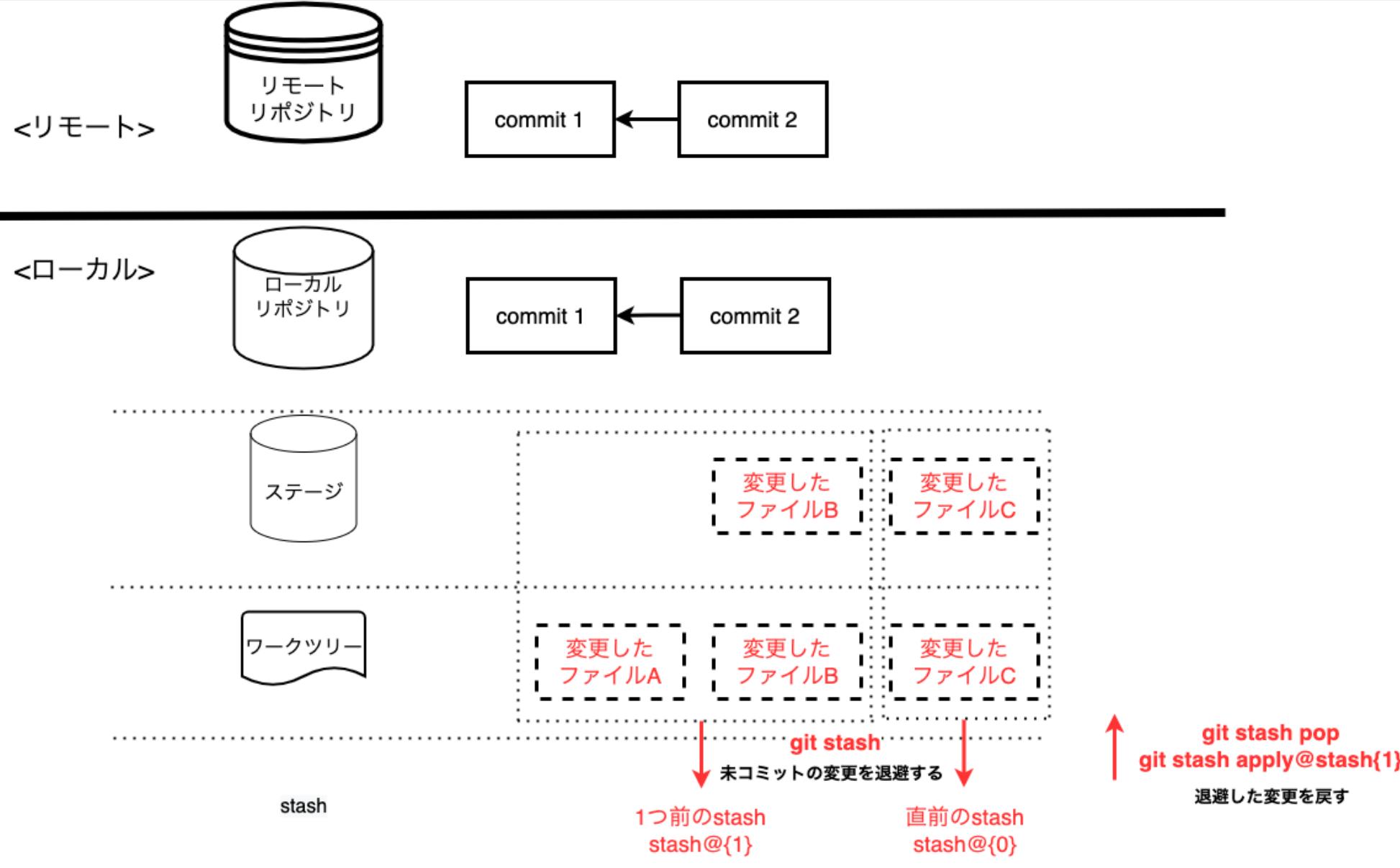
## 機能

- 未コミットの変更を退避する

## ユースケース

- 他ブランチに切り替えたいが、作業が中途半端なのでコミットはしたくない

# イメージ



## 主なオプション

- `-u | --include-untracked` : 追跡対象に含まれていないファイル(新規作成ファイルなど)も含めて退避

## コマンド例

```
$ git stash -u # 新規作成ファイルも含めて変更を退避
$ git stash push -m "<メッセージ>" -u # メッセージを付けて変更を退避
$ git stash list # 退避した作業の一覧を見る
$ git stash pop # 直前に退避した変更を戻す。退避していたデータは削除する。
$ git stash apply stash@{N} # 直前の次から数えてN番目に退避した変更を戻す。直前の変更は N=0。stash@{N} を省略した場合は直前に stash した情報 (N=0) を戻す
$ git stash show stash@{N} -p # 直前の次から数えてN番目に退避した変更の詳細を見る。
$ git stash drop stash@{N} # 直前の次から数えてN番目に退避していたデータを削除する
```

## 参考

- [git-stash Documentation](#)
- [【git stash】コミットはせずに変更を退避したいとき - Qiita](#)

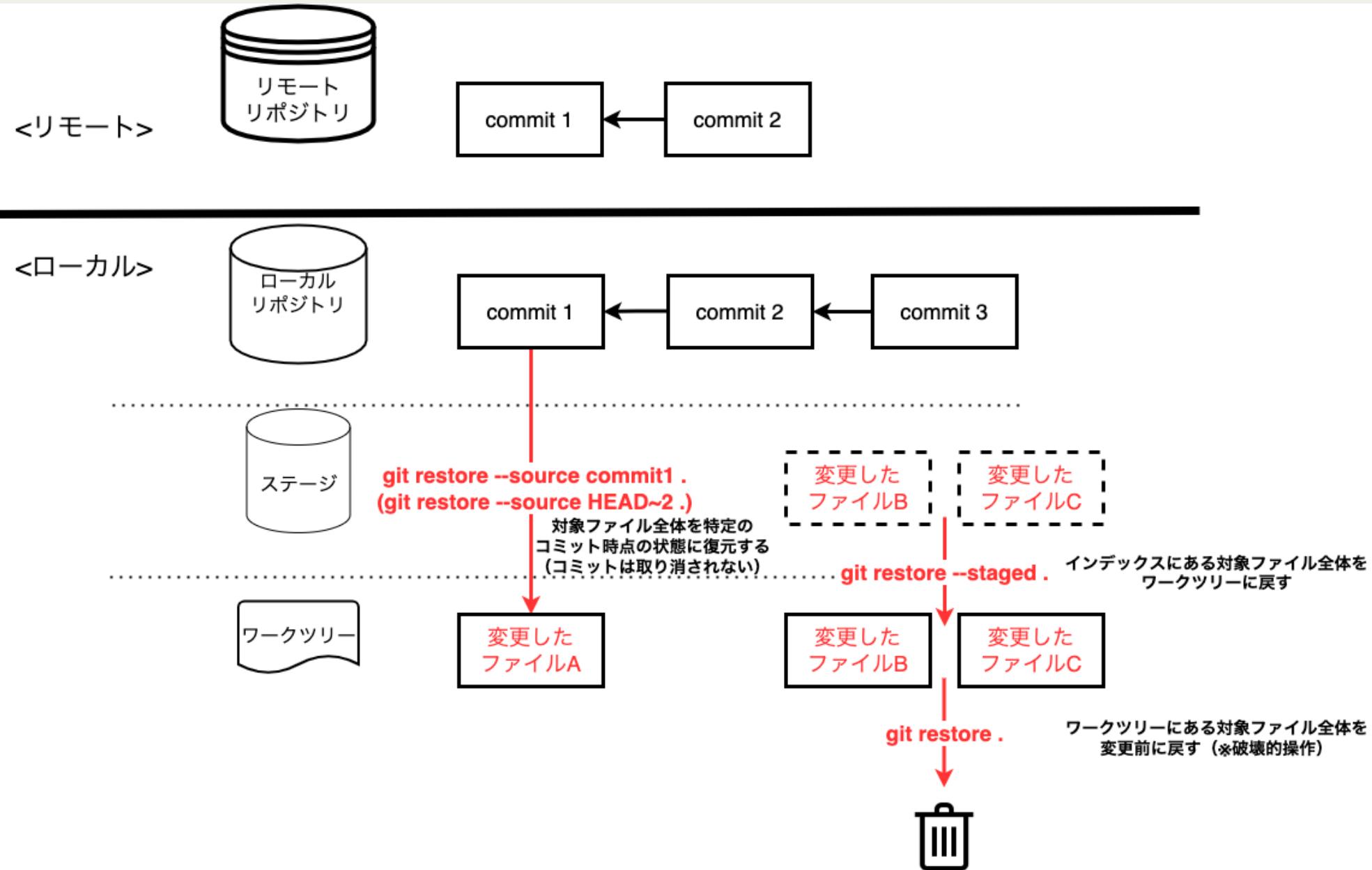
# restore (checkout) ★

## 機能

- ファイルを指定した状態に復元する

## ユースケース

- add をしたけど取り消したい
- 特定のコミットの時点に戻したい



## 主なオプション

- `-S | --staged` : インデックスを復元
- `-s | --source` : 指定したコミットの状態にワークツリーファイルを復元(reset と異なりコミットは取り消されない)

## コマンド例

```
git restore <ファイル名> # 特定のファイルを保存前に戻す。破壊的操作。git checkout -- <ファイル名> と同じ  
git restore --staged . # 対象ファイル全体を add される前に戻す。git reset . と同じ  
git restore --source <コミットID> <ファイル名> # 特定のファイルを特定のコミット時点に戻す。git checkout <コミットID> -- <ファイル名> と同じ  
git restore --source HEAD~2 . # 対象ファイル全体を2つ前のコミット時点の状態に復元する
```

## 備考

- `restore` は Git バージョン 2.23.0 でリリース (2019/08/16)
- `checkout` は複数の役割を兼ね備えてしまっているため、こちらの方が直感的に理解しやすい

## 参考

- [git-restore – Git コマンドリファレンス\(日本語版\)](#)
- [git switch と restore の役割と機能について - Qiita](#)
- [これからは git restore を使ってみようかな](#)

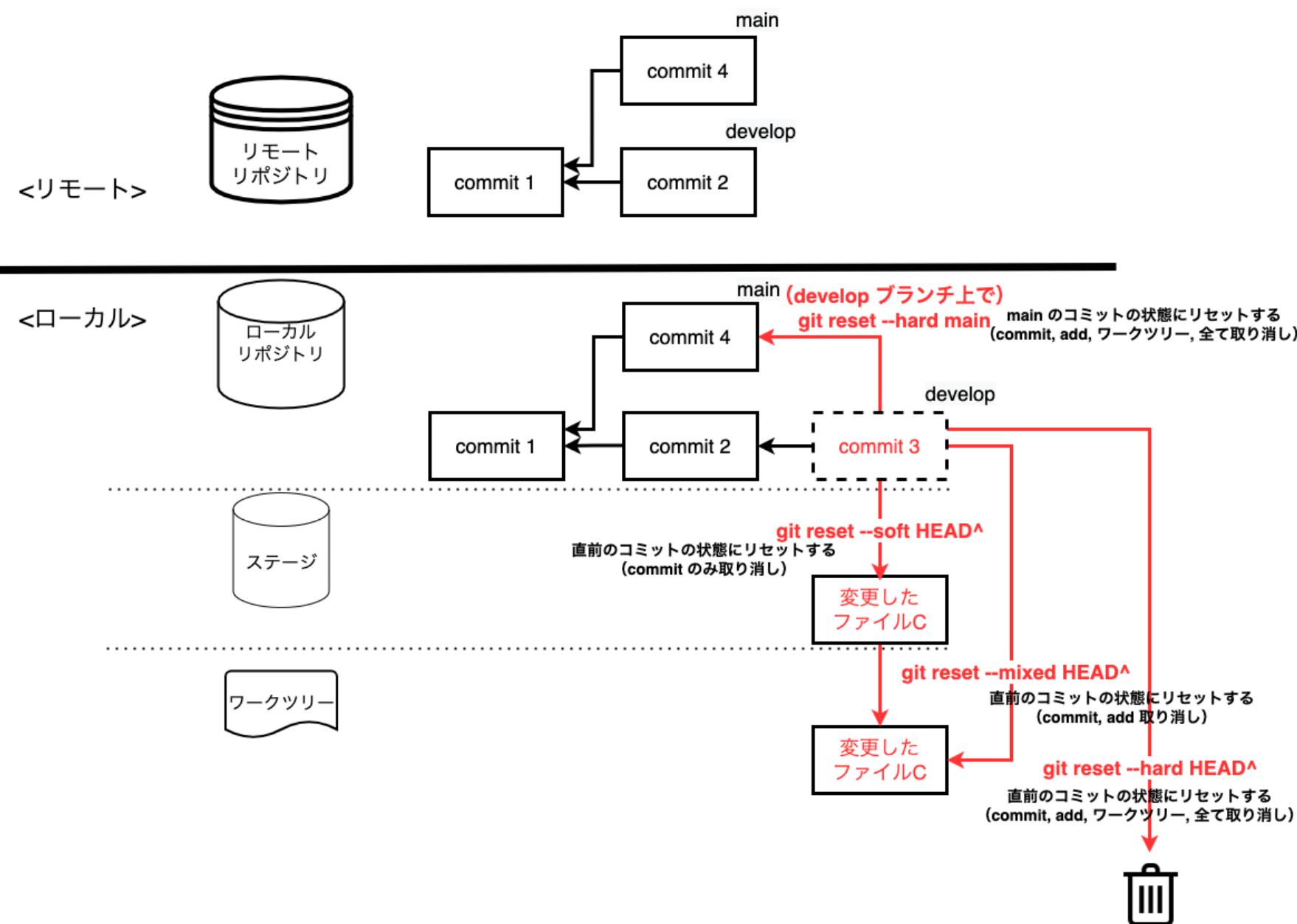
# reset ★

## 機能

- 指定したファイルを特定のコミットの状態まで戻す
  - 実際の挙動としては HEAD の位置を変更することで、特定のコミット時点の状態へと移動している
  - オプションによってインデックス、ワークツリーの内容も変更できる

## ユースケース

- 直前のコミットを取り消したい (**push する前**)
- 目的のブランチに切り替える前に誤って pull してしまったのを取り消したい



## 主なオプション

- `--soft` : インデックスとワークツリーはそのままで、指定したコミットの状態へリセット (commit のみ取り消し)
- `--mixed` : ワークツリーはそのまま、インデックスを指定したコミットの状態へリセット (commit と add を取り消し。デフォルト)
- `--hard` : インデックスとワークツリーを指定したコミットの状態へリセット (現在のファイル変更も含めて取り消し)



## コマンド例

```
$ git reset --soft HEAD^ # 直前の commit のみ取り消す。現在のファイル変更はそのまま。HEAD^ は直前のコミットを意味する
$ git reset --mixed HEAD^ # 直前の commit, add を取り消す。現在のファイル変更はそのまま。
$ git reset --hard HEAD^ # 直前の commit, add, 現在のファイル変更も全部取り消す（破壊的操作）
$ git reset --hard <コミットID> # 指定したコミットの状態に戻す
$ git reset --hard <ブランチ> # 指定したブランチの状態に戻す（現在いるブランチに戻す場合は下と同じ）
$ git reset --hard ORIG_HEAD # 直前の reset を取り消す（最新の状態に戻る）
```

## 備考

- `ORIG_HEAD` : HEAD が移動するコマンドを実行した際に、コマンド実行時の HEAD の位置を記録しているポインタ

## 参考

- [git-reset – Git コマンドリファレンス \(日本語版\)](#)
- [\[git reset \(--hard/--soft\)\]ワークツリー、インデックス、HEAD を使いこなす方法](#)
- [第 6 話 git reset 3 種類をどこよりもわかりやすい図解で解説!【連載】マンガでわかる Git ~コマンド編~](#)
- [git reset についてもまとめてみる](#)

## revert ★

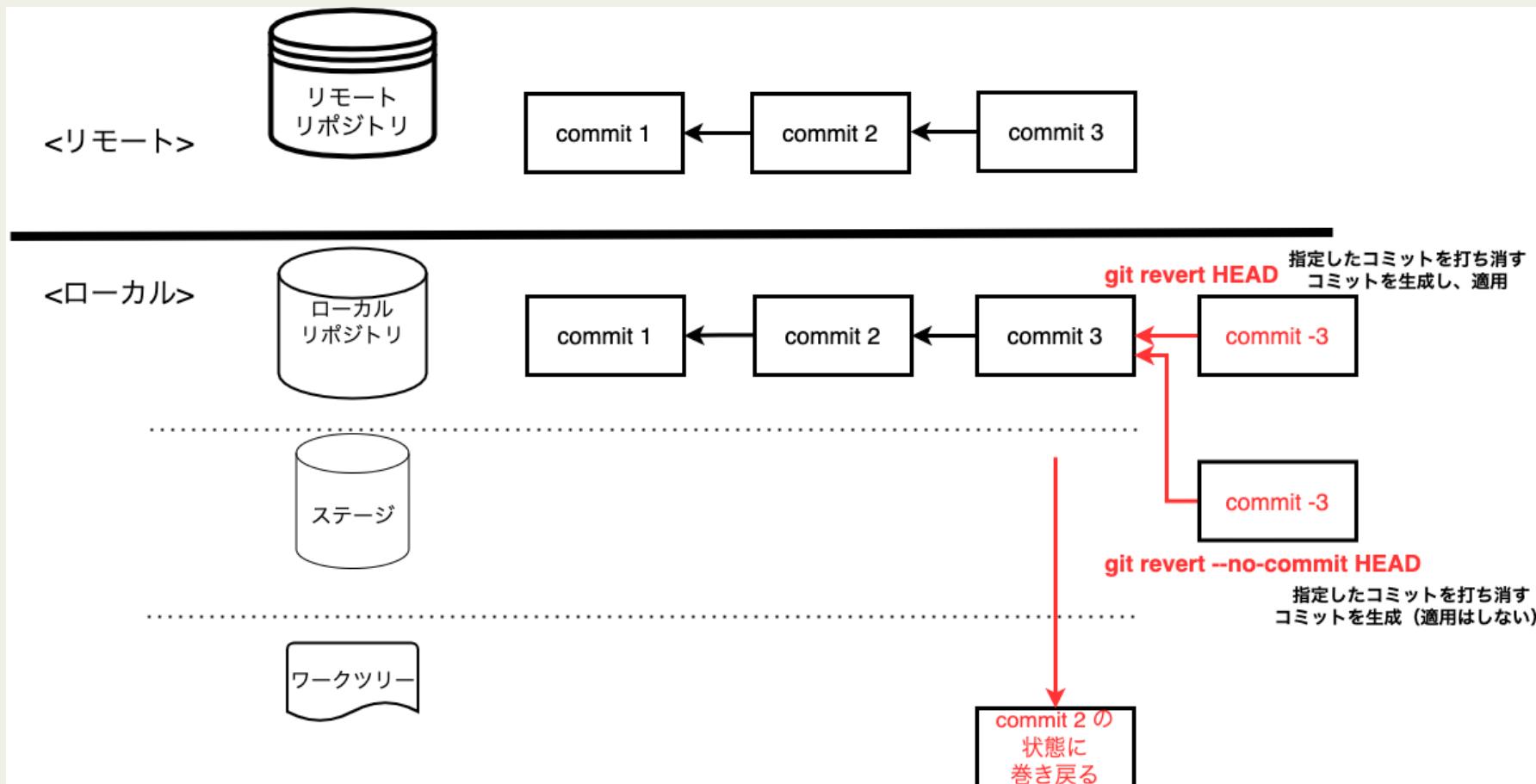
### 機能

- ・ 指定したコミットの内容を打ち消すコミットを生成し、現在いるブランチに適用する

### ユースケース

- ・ 直前のコミットを取り消したい (push した後)

## イメージ





## 主なオプション

- `-e | --edit` : コミットメッセージを編集する (デフォルト)
- `--no-edit` : コミットメッセージを編集しない
- `-n | --no-commit` : 打ち消しコミットを生成するが、適用しない
- `-m | --mainline` : マージコミットを打ち消す場合
  - オプションの引数に 1 か 2 を指定する。1 が「マージされた側のブランチ」、2 が「マージする側のブランチ」
  - マージしたコミットそのものを revert するときは 1 になるので、基本的に 1 を使う
- `--continue` : コンフリクト解決後、マージを続行
- `--abort` : コンフリクト解決を中止し、マージ前の状態に再構築

## コマンド例

```
$ git revert HEAD # 直前のコミットを打ち消すコミットを生成し、適用
$ git revert -n <コミットID> # 特定のコミットを打ち消すコミットを生成（適用はしない。適用するときは git commit）
$ git revert HEAD~N # 直前からN個前の範囲で複数コミットを打ち消すコミットを生成し、適用
$ git revert <コミットID A>..<コミットID B> # A から B の範囲で複数コミットを打ち消すコミットを生成し、適用
$ $ git revert -m 1 <マージコミットID> # マージコミットを打ち消し。1が「マージされた側のブランチ」、2が「マージする側のブランチ」。
```

## 参考

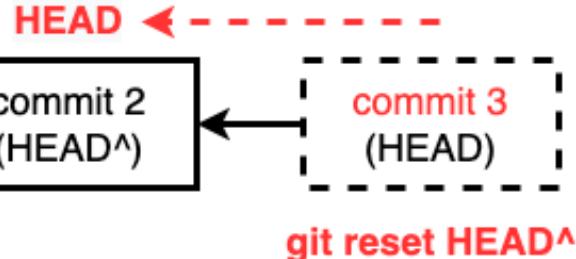
- [git-revert Documentation](#)
- [【git コマンド】いまさらの revert - Qiita](#)
- [Git revert と reset について - Qiita](#)

# コラム:reset と revert の違い



## git reset

指定したコミットの状態に戻す



HEAD<sup>^</sup> が指すコミットの状態に戻す  
(HEAD は 1つ前のコミットに移動する)

## git revert

指定したコミットを打ち消すコミットを生成する



HEAD が指すコミットを打ち消すコミットを生成する  
(HEAD は生成したコミットに移動する)

いずれのコマンドを実行しても、結果的に手元のソースコードは同じ状態となる  
(コマンド実行時点で HEAD<sup>^</sup> が指していたコミットの状態)

## イメージ

- reset: 指定したコミットの状態に戻す (指定したコミットより新しいコミットは削除する)
  - 直前のコミットに戻りたい場合は、1つ前のコミット( `HEAD^` )を指定する
  - `$ git reset HEAD^`
- revert: 指定したコミットを打ち消すコミットを生成する
  - 直前のコミットに戻りたい場合は、現在のコミット( `HEAD` )を打ち消すように指定する
  - `$ git revert HEAD`

# コラム:reset と revert はどっちがいい?

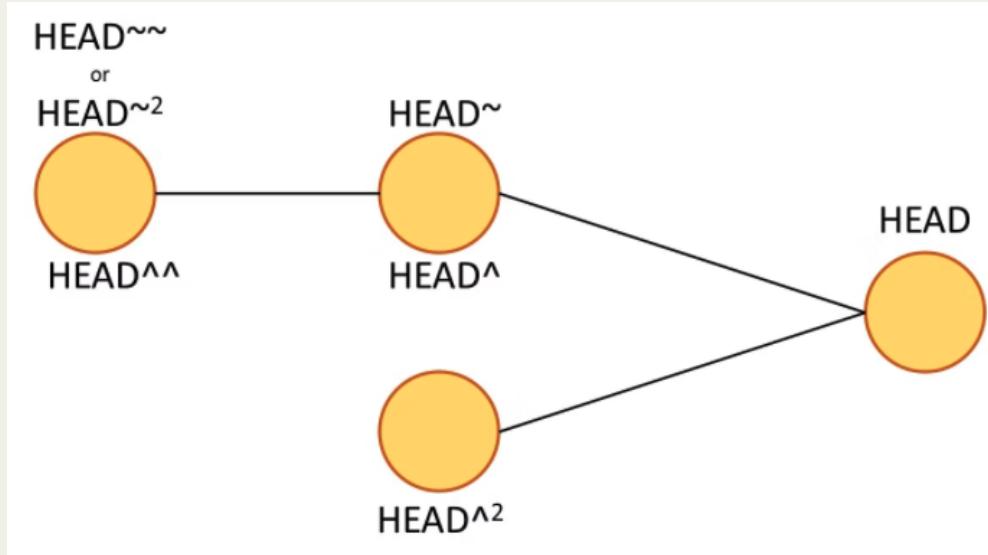
A. リモートリポジトリに push 済(他者に共有済)であれば revert。あとはチームの運用方針次第。

-	reset	revert
メリット	<ul style="list-style-type: none"><li>- 誤ったコミット自体を削除できるのでコミット履歴が見やすい</li><li>- HEAD の位置を大幅に移動することができる</li></ul>	<ul style="list-style-type: none"><li>- コミット自体を削除するわけではないので、安全にコミットを元に戻すことができる</li><li>- 誤った履歴が残ることで revert 自体の取り消しも容易に行うことができる</li></ul>
デメリット	<ul style="list-style-type: none"><li>- チームで開発にて他の人が同じブランチで作業していた場合に、コンフリクトがおきたり、エラー発生の原因となる可能性がある</li><li>- コミット自体を削除してしまうため、復元作業に少し手間がかる</li></ul>	<ul style="list-style-type: none"><li>- 誤った履歴が残ってしまうので、コミットログが見づらくなる</li><li>- 直前のコミットより前のコミットを revert する場合、コミットの整合性が取れなくなる</li></ul>

## 参考

- Git revert と reset について - Qiita
- 【git】revert と reset の違い

# コラム: HEAD^ と HEAD~ の違い



- HEAD : 今自分が作業しているブランチ(コミット)を示すポインタ (@ は HEAD の別名(エイリアス)で同じ意味)
- ^ (キャレット): (複数分岐がある親コミットの中から)N 個前の親コミットを指定 (マージコミットの分岐元コミットを指定するときに使用)
- ~ (チルダ): (複数続いている直線コミットの中から)N 世代前のコミットを指定。HEAD~N で N 世代前のコミットを指定
  - HEAD~~ と HEAD~2 は同じコミットを指すが、HEAD^^ と HEAD^2 はそれぞれ違うコミットを指す

## 参考

- Git の HEAD とは何者なのか - Qiita
- 【やっとわかった!】git の HEAD^ と HEAD~ の違い - Qiita
- 7.1 Git のさまざまなツール - リビジョンの選択

# cherry-pick ★

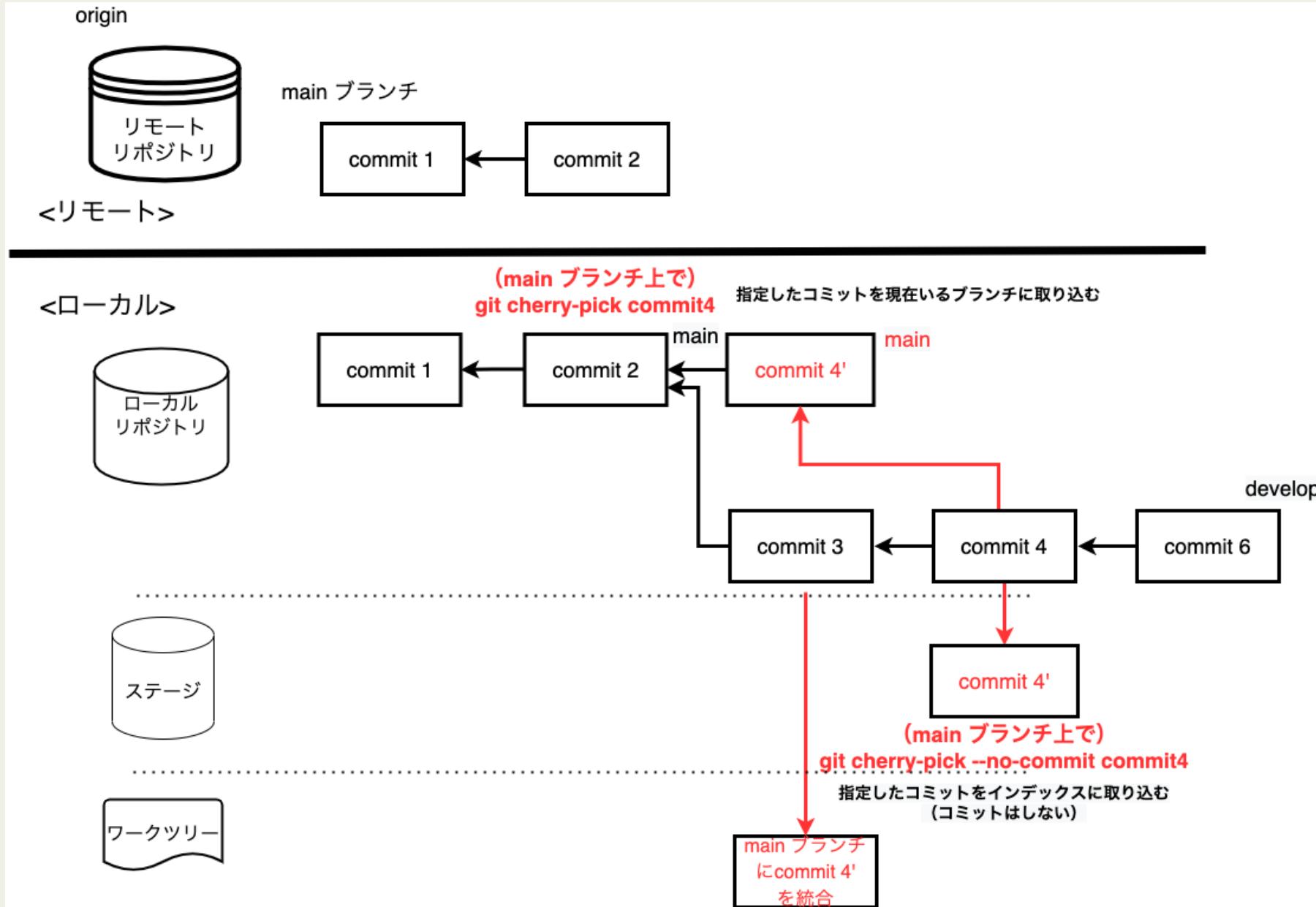
## 機能

- 指定したコミットの内容を現在いるブランチに取り込む

## ユースケース

- 他のブランチの特定のコミットを現在のブランチに取り込みたい
  - 特定のバージョンで開発した機能を他のバージョンにも適用(出荷)したい
  - 複数ブランチを並行して開発している状況で、あるブランチがマージされたので、最新の統合ブランチから新たにトピックブランチを切り、先行して開発が進んでいる他のブランチから必要なコミットのみを取ってきたい

# イメージ



## 主なオプション

- `-e | --edit` : コミットメッセージを編集する
- `-n | --no-commit` : 指定したコミットを取得するが、適用しない
- `--continue` : コンフリクト解決後、マージを続行
- `--abort` : コンフリクト解決を中止し、マージ前の状態に再構築

## コマンド例

```
$ git cherry-pick <コミットID> # 特定のコミットを現在のブランチに取り込み、適用  
$ git cherry-pick -n <コミットID> # 特定のコミットをインデックスに取り込む（適用するときは git commit）  
$ git cherry-pick <コミットID A> <コミットID B> # AとBのコミットを現在のブランチに取り込み、適用  
$ git cherry-pick <コミットID A>..<コミットID B> # AからBの範囲のコミットを現在のブランチに取り込み、適用
```

## 参考

- [git-cherry-pick Documentation](#)
- [git cherry-pick を完全マスター!特定コミットのみを取り込む方法](#)
- [git で他ブランチの特定のコミットを取り込む方法 - Qiita](#)

# blame

## 機能

- 指定されたファイルの各行に最終更新者・日時などの情報を表示する

## ユースケース

- この行はいつ誰がどういった理由で変更したのか調べたい
  - 不具合が発生しているコードに実行することで、発生元の MR(PR) やチケットを特定できる
  - 意図が不明瞭なコードやこれから修正したいコードに実行することで、当該コードが追加された MR(PR) やチケットからコードの背景や意図を理解できたり、影響範囲調査の助けになったりする

## 主なオプション

- `-L` : 最終コミットを表示する行の範囲を指定

## コマンド例

```
$ git blame <ファイル名> # 特定のファイルについて各行ごとに最終コミット情報を表示する  
$ git blame -L 40,50 <ファイル名> # 40～50行目の最終コミット情報を表示する  
$ git blame -L 40,+10 <ファイル名> # 40行目から10行分の最終コミット情報を表示する
```

## 備考

- 最近のエディタでは拡張機能を入れると `blame` の結果が表示される(Visual Studio Code では [GitLens](#))

## 参考

- [git-blame Documentation](#)
- [インデントコミットで真犯人がわからなくなつた場合の git blame - Qiita](#)
- [GitLens — Git supercharged - Visual Studio Marketplace](#)

# tag

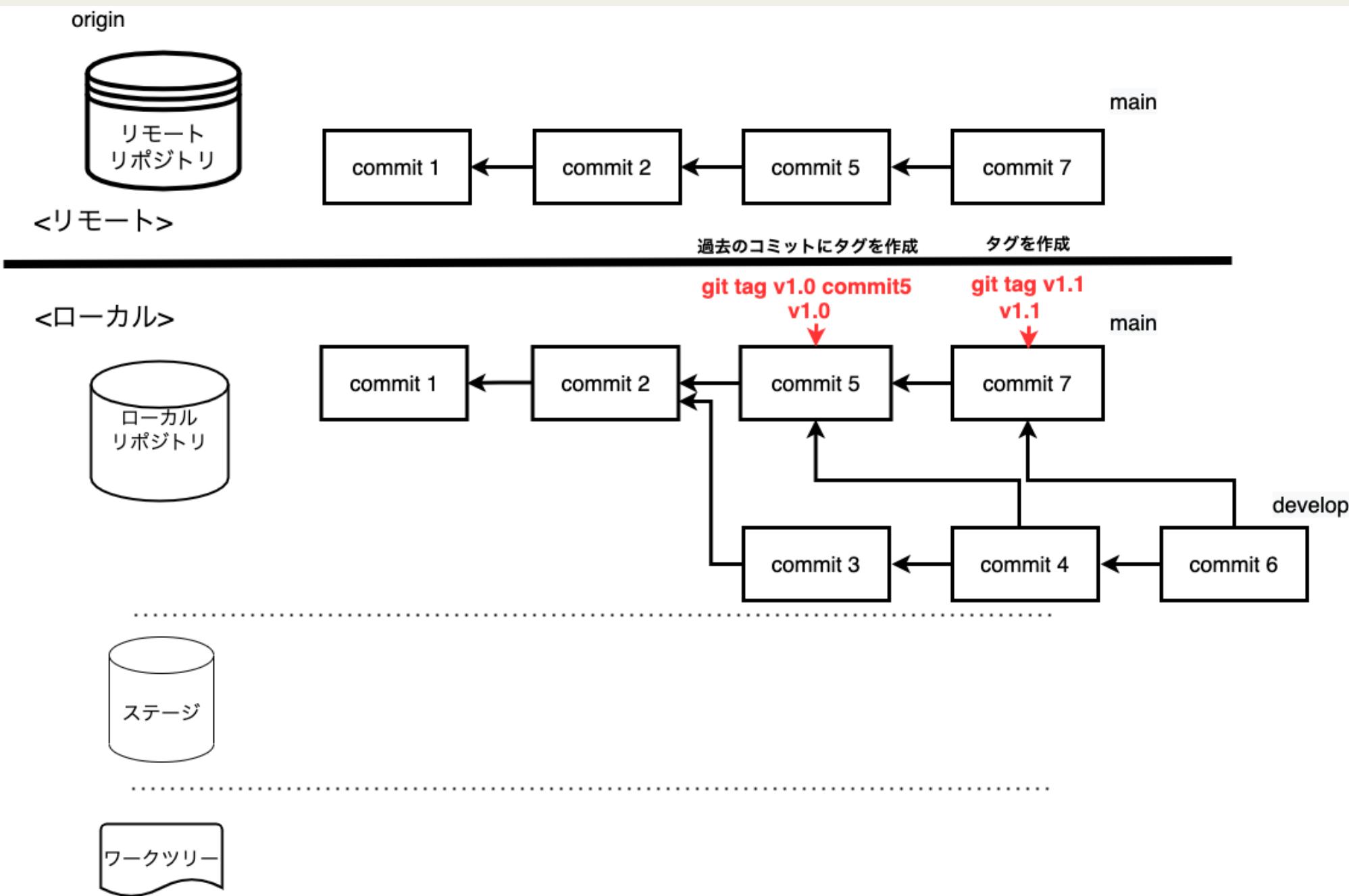
## 機能

- コミットにタグを付ける、削除する、一覧表示する

## ユースケース

- リリースした時点のソースコードに名前を付けたい

# イメージ



## 主なオプション

- `-a | --annotate` : 注釈付きのタグを作成
- `-m | --message` : メッセージを付与
- `-l | --list` : タグを一覧表示

## コマンド例

```
$ git tag <タグ名> # コメント（注釈）なしでタグを作成する
$ git tag -a <タグ名> -m '<タグのコメント>' # コメント（注釈）付きでタグを作成
$ git tag -a <タグ名> -m '<タグのコメント>' <コミットID> # 過去のコミットにタグを作成
$ git push origin <タグ名> # タグをリモートリポジトリに反映する
$ git tag -l # タグを一覧表示する
```

## 参考

- [git-tag – Git コマンドリファレンス\(日本語版\)](#)
- [git tag の使い方まとめ - Qiita](#)

# reflog

## 機能

- git の操作履歴を表示する
  - 過去に行った branch の移動や commit、pull や reset などの操作履歴を見ることができる

## ユースケース

- コミット履歴だけでなく HEAD やブランチの動きなど細かいところまで確認したい
- 誤って意図と異なるコミットへ `git reset --hard` をしてしまったので戻したい

## 主なオプション

- `--date=default` : 操作日時を絶対時刻で表示

## コマンド例

```
# $ git reset --hard を失敗してしまったとき
$ git reflog # コミット履歴や HEAD・ブランチの参照に関する変化ログを表示
$ git reflog --date=default # 日時付きで表示
$ git reset HEAD@{N} --hard # reflog 確認した元に戻したいコミットを指定し、reset --hard で戻す
```

## 参考

- [git-reflog Documentation](#)
- [git reflog についてまとめてみる](#)
- [【git reflog】ブランチの復元やりセットしたコミットを元に戻す方法](#)
- [git reflog を日時で参照する](#)

# ハンズオン

## 事前準備

以下が終わっていない人は実施をお願いします。

1. ローカル端末 Git( v2.23.0 以上推奨) をインストールしておいてください。(ターミナルで `git --version` を実行して v2.23.0 以上が表示されれば OK です)
  - i. [【参考】Git インストール手順 < Windows 向け >](#)
  - ii. [【参考】Git Bash を手動でアップデートする方法【Git for Windows】](#)
2. ハンズオン用に GitHub や GitLab で個人用のリモートリポジトリを作成しておいてください。
  - i. [【参考】GitLab での個人リポジトリの作成方法 \(From 赤池さん\)](#)
  - ii. [【参考】【GitLab】プロジェクト\(リポジトリ\)を作成する - Qiita](#)
  - iii. [【参考】【超入門】GitHub リポジトリの作り方 - Qiita](#)
3. 勉強会2日目スライドのハンズオン ①(既にリモートに存在するリポジトリをコピー - `git clone`)を完了してください。
4. 勉強会2日目スライドのハンズオン ②(Git 開始時の基本設定 - `git config`)を完了してください。

利用するコマンドの説明スライドを復習しながら進めましょう!

## 【参考】ターミナルでよく使うコマンド ①

cd

- ・ディレクトリを移動します。

ls

- ・ディレクトリの内容を表示します。ls -a コマンドで、隠しファイルを含めたディレクトリ全内容を表示します。

mkdir

- ・ディレクトリを新規作成します。

touch

- ・ファイルを作成します

rm

- ・ファイルを削除します。

## 【参考】ターミナルでよく使うコマンド ②

cp

- ファイルをコピーします。

mv

- ファイルの移動とファイル名の変更を行います。

cat

- ファイルの中身を表示します。

echo

- 画面に文字列を表示します。

### >> (リダイレクト(追記))

- コマンドの出力結果をファイルへ追記します。

- echo コマンドと組み合わせて、echo "文字列" >> ファイル名 で指定した文字列をファイルに追記します。

# ハンズオン⑥:リモートリポジトリの内容を取得し、現在のブランチに取り込む

リモートリポジトリの内容を取得し、現在チェックアウトしているブランチに取り込んでみましょう。  
pull でもできますが、今回は fetch + merge を使ってやってみましょう。  
(次頁に解答例があります)

## 事前準備

1. GitHub/GitLab で事前準備したハンズオン用リポジトリをブラウザで表示
2. 自分がどのブランチのコードを開いているかを確認(ハンズオンでは main を例にします)
3. リモートリポジトリにファイル index.html を作成してコミット(中身は `<h1>Git Exercise</h1>` を書く)
  - i. "Add file", もしくは "+" ボタンをクリックして追加
  - ii. "Edit file" でファイル内本文に `<h1>Git Exercise</h1>` を追記
  - iii. コミットメッセージに "Add index.html" と記載してコミット

## 操作手順例

1. ハンズオン用ディレクトリへ移動
2. リモートの変更内容を取得したいブランチへ切り替え
3. リモートリポジトリの内容をリモート追跡ブランチに反映
4. ローカルブランチとリモート追跡ブランチの差分を確認 (ヒント: `diff <ブランチ名> origin/<ブランチ名>`)
5. リモート追跡ブランチの内容をローカルリポジトリに反映
6. リモートリポジトリから取り込んだコミットとマージコミットがコミット履歴に反映されていることを確認
7. リモートリポジトリの内容がローカルに反映されていることを確認



## ハンズオン⑥:リモートリポジトリの内容を取得し、現在のブランチに取り込む (解答例)

```
$ cd git-exercise # ハンズオン用ディレクトリへ移動（※ git-exercise は例です）
$ git switch main # リモートの変更内容を取得したいブランチへ切り替え

$ git fetch # リモートリポジトリの内容をリモート追跡ブランチに反映
$ git diff main origin/main # ローカルブランチとリモート追跡ブランチの差分を確認（エディタで vim が開いた場合は `:q` で終了）
$ git merge origin/main # リモート追跡ブランチの内容をローカルリポジトリに反映（エディタで vim が開いた場合は `:x` で保存して終了）
# 上記 fetch + merge の手順は git pull origin main でも可

$ git log # リモートリポジトリから取り込んだコミットとマージコミットがコミット履歴に反映されていることを確認（エディタで vim が開いた場合は `:q` で終了）
$ ls # リモートリポジトリの内容がローカルに反映されていることを確認
```

## ハンズオン⑦:トピックブランチを作成してローカル内でマージ

ローカルリポジトリで統合ブランチ(ex. develop)からトピックブランチ(ex. feature1)を作成し、トピックブランチで適当な修正を行ってコミットしてみましょう。  
その後、統合ブランチに戻ってトピックブランチをマージしてみましょう。  
(次頁に解答例があります)

### 操作手順例

1. ハンズオン用ディレクトリへ移動
2. ベースブランチとする develop ブランチへ切り替え(未作成の場合は git switch -c develop で作成)
3. feature1 ブランチを作成して切り替え
4. feature1.html というファイルを作成
5. "<h1>feature1</h1>" という内容を feature1.html に追記
6. 変更内容をインデックスに追加
7. "Add feature1.html" というメッセージを付けてインデックスに追加した内容をコミット
8. ベースブランチへ切り替え
9. ベースブランチと feature1 ブランチの差分を確認 ( ヒント : diff <ブランチ名 A> <ブランチ名 B> )
10. feature1 ブランチを統合
11. feature1 ブランチでのコミットとマージコミットがコミット履歴に反映されていることを確認
12. feature1 ブランチの内容がベースブランチに反映されていることを確認

## ハンズオン⑦:トピックブランチを作成してローカル内でマージ(解答例)

```
$ cd git-exercise # ハンズオン用ディレクトリへ移動 (※ git-exercise は例です)
$ git switch develop # ベースブランチとする develop ブランチへ切り替え (未作成の場合は git switch -c develop で作成)

$ git switch -c feature1 # feature1 ブランチを作成して切り替え
$ touch feature1.html # feature1.html というファイルを作成 (エディタ上でファイルを新規作成しても OK)
$ echo "<h1>feature1</h1>" >> feature1.html # "<h1>feature1</h1>" という内容を feature1.html に追記 (エディタ上で追記しても OK)

$ git add . # 変更内容をインデックスに追加
$ git commit -m "Add feature1.html" # "Add feature1.html" というメッセージを付けてインデックスに追加した内容をコミット

$ git switch develop # develop ブランチへ切り替え (※ develop は例です)
$ git diff develop feature1 # develop ブランチと feature1 ブランチの差分を確認
$ git merge feature1 # feature1 ブランチを統合 (エディタで vim が開いた場合は `:x` で保存して終了)

$ git log # feature1 ブランチでのコミットとマージコミットがコミット履歴に反映されていることを確認 (エディタで vim が開いた場合は `:q` で終了)
$ ls # feature1 ブランチの内容が develop ブランチに反映されていることを確認
```

# ハンズオン⑧:ローカルの変更内容を一時退避



ローカルリポジトリで適当な修正を行った後に変更を一時退避してみましょう（次頁に解答例があります）

## 操作手順例

1. ハンズオン用ディレクトリへ移動
2. develop ブランチへ切り替え
3. stash-test ブランチを作成して切り替え
4. "<p>stash test</p>" を既存の feature1.html の本文に追記 (feature1.html がない場合は作成・コミット)
5. 変更を一時退避
6. stash.html というファイルを作成
7. 新規作成ファイルも含めて変更を一時退避
8. 現在の状態を確認する。 `nothing to commit, working tree clean` となっていることを確認
9. 退避した作業の一覧を確認
10. 直前から 2 番目に退避した変更の詳細を確認(※ 直前の変更は N=0)
11. 直前から 2 番目に退避した変更を復元(※ 直前の変更は N=0)
12. 現在の状態を確認。 `Changes not staged for commit: modified: feature1.html` となっていることを確認
13. feature1.html の内容に "<p>stash test</p>" があることを確認
14. 直前に退避した変更を復元。その際に退避していたデータは削除する
15. ディレクトリ内のファイルを確認
16. 退避した作業の一覧を確認。直前の退避データが削除されていることを確認
17. stash.html を削除

## ハンズオン⑧:ローカルの変更内容を一時退避(解答例)

```
$ cd git-exercise # ハンズオン用ディレクトリへ移動 (※ git-exercise は例です)
$ git switch develop # develop ブランチへ切り替え (※ develop は例です。未作成の場合は git switch -c develop で作成)

$ git switch -c stash-test # stash-test ブランチを作成して切り替え
#   (feature1.html がない場合は $ touch feature1.html で作成し、add, commit をしておく)
$ echo "<p>stash test</p>" >> feature1.html # "<p>stash test</p>" を既存の feature1.html の本文に追記 (エディタ上で追記しても OK)
$ git stash # 変更を一時退避

$ touch stash.html # stash.html というファイルを作成 (エディタ上でファイルを新規作成しても OK)
$ git stash -u # 新規作成ファイルも含めて変更を一時退避
$ git status # 現在の状態を確認。`nothing to commit, working tree clean` となっていることを確認
$ git stash list # 退避した作業の一覧を確認

$ git stash show stash@{1} -p # 直前から2番目に退避した変更の詳細を確認 (直前の変更は N=0) 。"<p>stash test</p>" という差分が表示されている
$ git stash apply stash@{1} # 直前から2番目に退避した変更を復元 (直前の変更は N=0)
$ git status # 現在の状態を確認。`Changes not staged for commit: modified: feature1.html` となっていることを確認
$ cat feature1.html # feature1.html の内容に "<p>stash test</p>" があることを確認

$ git stash pop # 直前に退避した変更を復元。その際に退避していたデータは削除する
$ ls # ディレクトリ内のファイルを確認。stash.html が作成されていることを確認
$ git stash list # 退避した作業の一覧を確認。直前の退避データが削除されていることを確認
$ rm stash.html # stash.html を削除 (エディタ上でファイルを削除しても OK)
```

## ハンズオン⑨:ローカルで誤ってコミットした内容を元に戻す

ローカルリポジトリで誤ってコミットした内容を元に戻してみましょう（次頁に解答例があります）

### 操作手順例

1. ハンズオン用ディレクトリへ移動
2. develop ブランチへ切り替え
3. reset-test ブランチを作成して切り替え
4. feature2.html というファイルを作成
5. 変更内容をインデックスに追加
6. "Add feature2.html" というメッセージを付けてインデックスに追加した内容をコミット
7. commit のみ取り消し。現在のファイル変更はそのまま
8. 現在の状態を確認
  - i. Changes to be committed: new file: feature2.html となっていることを確認
9. "Add feature2.html" というメッセージを付けてインデックスに追加した内容をコミット
10. commit, add, 現在のファイル変更も全部取り消し
11. 現在の状態を確認
  - i. nothing to commit, working tree clean となっていることを確認

## ハンズオン⑨:ローカルで誤ってコミットした内容を元に戻す(解答例)

```
$ cd git-exercise # ハンズオン用ディレクトリへ移動（※ git-exercise は例です）
$ git switch develop # develop ブランチへ切り替え（※ develop は例です。未作成の場合は git switch -c develop で作成）

$ git switch -c reset-test # reset-test ブランチを作成して切り替え
$ touch feature2.html # feature2.html というファイルを作成（エディタ上でファイルを新規作成しても OK）
$ git add . # 変更内容をインデックスに追加
$ git commit -m "Add feature2.html" # "Add feature2.html" というメッセージを付けてインデックスに追加した内容をコミット
$ git reset --soft HEAD^ # commit のみ取り消し。現在のファイル変更はそのまま。HEAD^ は直前のコミットを意味する
$ git status # 現在の状態を確認。`Changes to be committed:           new file:   feature2.html` となっていることを確認

$ git commit -m "Add feature2.html" # "Add feature2.html" というメッセージを付けてインデックスに追加した内容をコミット
$ git reset --hard HEAD^ # commit, add, 現在のファイル変更も全部取り消し（破壊的操作）
$ git status # 現在の状態を確認。`nothing to commit, working tree clean` となっていることを確認
```

# ハンズオン⑨:ローカルで誤ってコミットした内容を元に戻す(応用編)



ローカルリポジトリで2つ以上前のコミット状態に戻したり、戻す操作自体を取り消したりしてみましょう  
(次頁に解答例があります)

## 操作手順例

(ハンズオン⑨の基本編が終わった状態から)

1. feature2.html というファイルを作成
2. 変更内容をインデックスに追加
3. "Add feature2.html" というメッセージを付けてインデックスに追加した内容をコミット
4. feature3.html というファイルを作成
5. 変更内容をインデックスに追加
6. "Add feature3.html" というメッセージを付けてインデックスに追加した内容をコミット
7. コミット履歴を確認
  - i. 2つのコミットがコミット履歴に反映されていることを確認
8. 2つ前のコミットの状態へリセット
9. コミット履歴を確認
  - i. 2つのコミットがコミット履歴から消えていることを確認
10. 直前の reset を取り消し
11. コミット履歴を確認
  - i. 2つのコミットがコミット履歴に反映されていることを確認
12. develop ブランチの状態へリセット
13. develop ブランチに切り替え

## ハンズオン⑨:ローカルで誤ってコミットした内容を元に戻す(応用編)(解答例)

```
# (ハンズオン⑨の基本編が終った状態から)
$ touch feature2.html # feature2.html というファイルを作成（エディタ上でファイルを新規作成しても OK）
$ git add . # 変更内容をインデックスに追加
$ git commit -m "Add feature2.html" # "Add feature2.html" というメッセージを付けてインデックスに追加した内容をコミット
$ touch feature3.html # feature3.html というファイルを作成（エディタ上でファイルを新規作成しても OK）
$ git add . # 変更内容をインデックスに追加
$ git commit -m "Add feature3.html" # "Add feature3.html" というメッセージを付けてインデックスに追加した内容をコミット
$ git log # コミット履歴を確認。2つのコミットがコミット履歴に反映されていることを確認（エディタで vim が開いた場合は `:q` で終了）

$ git reset --hard HEAD~2 # 2つ前のコミットの状態へリセット
$ git log # コミット履歴を確認。2つのコミットがコミット履歴から消えていることを確認（エディタで vim が開いた場合は `:q` で終了）

$ git reset --hard ORIG_HEAD # 直前の reset を取り消し
$ git log # コミット履歴を確認。2つのコミットがコミット履歴に反映されていることを確認（エディタで vim が開いた場合は `:q` で終了）

$ git reset --hard develop # develop ブランチの状態へリセット
$ git switch develop # develop ブランチに切り替え
```

# ハンズオン ⑩: 別ブランチの特定のコミットを取り込む

ローカルリポジトリで別ブランチにある特定のコミットを取り込んでみましょう（次頁に解答例があります）

## 操作手順例

1. ハンズオン用ディレクトリへ移動
2. develop ブランチへ切り替え
3. cherry-test ブランチを作成して切り替え
4. feature4.html というファイルを作成
5. 変更内容をインデックスに追加
6. "Add feature4.html" というメッセージを付けてインデックスに追加した内容をコミット
7. feature5.html というファイルを作成
8. 変更内容をインデックスに追加
9. "Add feature5.html" というメッセージを付けてインデックスに追加した内容をコミット
10. コミット履歴を確認。"Add feature5.html" のコミット ID をコピーして控えおく
11. develop ブランチへ切り替え
12. pick-test ブランチを作成して切り替え
13. "Add feature5.html" のコミットを現在のブランチに取り込み
14. コミット履歴を確認。"Add feature5.html" のコミットが反映されていることを確認
15. ディレクトリ内のファイルを確認。feature5.html が作成されていることを確認



## ハンズオン ⑩: 別ブランチの特定のコミットを取り込む(解答例)

```
$ cd git-exercise # ハンズオン用ディレクトリへ移動（※ git-exercise は例です）
$ git switch develop # develop ブランチへ切り替え（※ develop は例です。未作成の場合は git switch -c develop で作成）

$ git switch -c cherry-test # cherry-test ブランチを作成して切り替え
$ touch feature4.html # feature4.html というファイルを作成（エディタ上でファイルを新規作成しても OK）
$ git add . # 変更内容をインデックスに追加
$ git commit -m "Add feature4.html" # "Add feature4.html" というメッセージを付けてインデックスに追加した内容をコミット

$ touch feature5.html # feature5.html というファイルを作成（エディタ上でファイルを新規作成しても OK）
$ git add . # 変更内容をインデックスに追加
$ git commit -m "Add feature5.html" # "Add feature5.html" というメッセージを付けてインデックスに追加した内容をコミット
$ git log # コミット履歴を確認。"Add feature5.html" のコミットID をコピーして控えおく

$ git switch develop # develop ブランチへ切り替え
$ git switch -c pick-test # pick-test ブランチを作成して切り替え
$ git cherry-pick <上記で控えたコミットID> # "Add feature5.html" のコミットを現在のブランチに取り込み
$ git log # コミット履歴を確認。"Add feature5.html" のコミットが反映されていることを確認
$ ls # ディレクトリ内のファイルを確認。feature5.html が作成されていることを確認
```

# 本日のゴール(再掲)

頭の中に「こんなときはこうする」というインデックスをぼんやりと作ること

- Git の各サブコマンドの存在を知ること
- 各サブコマンドのユースケースを知り、Git で躊躇したときに本資料を見返そうと思い付けること

→ この資料は辞書として使っていいってほしいので、完全に理解しようとしないで OK です！

# 次回

次回は **チーム開発体験編** です!  
おたのしみに!