



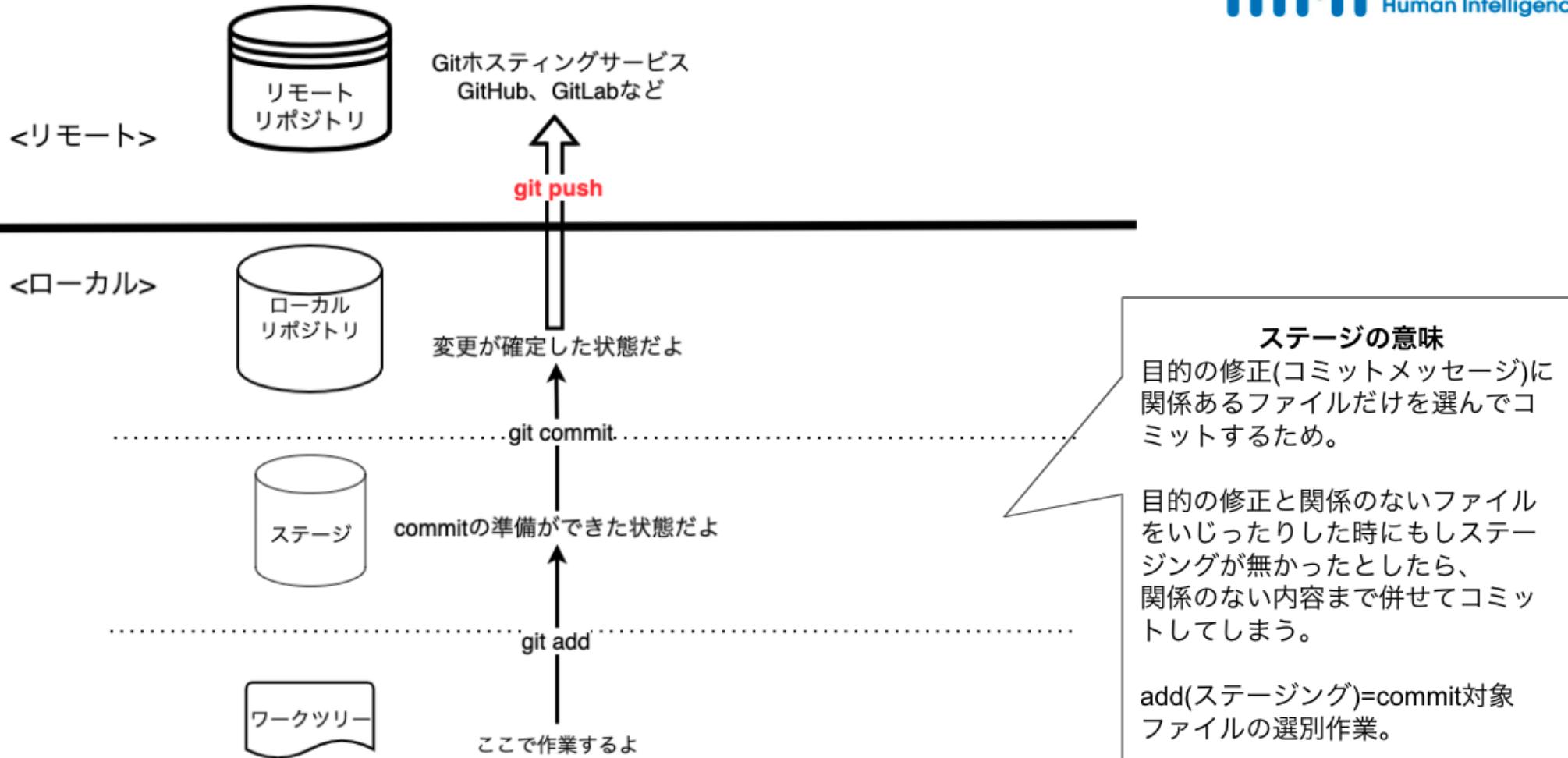
# Git 勉強会 3 日目

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)



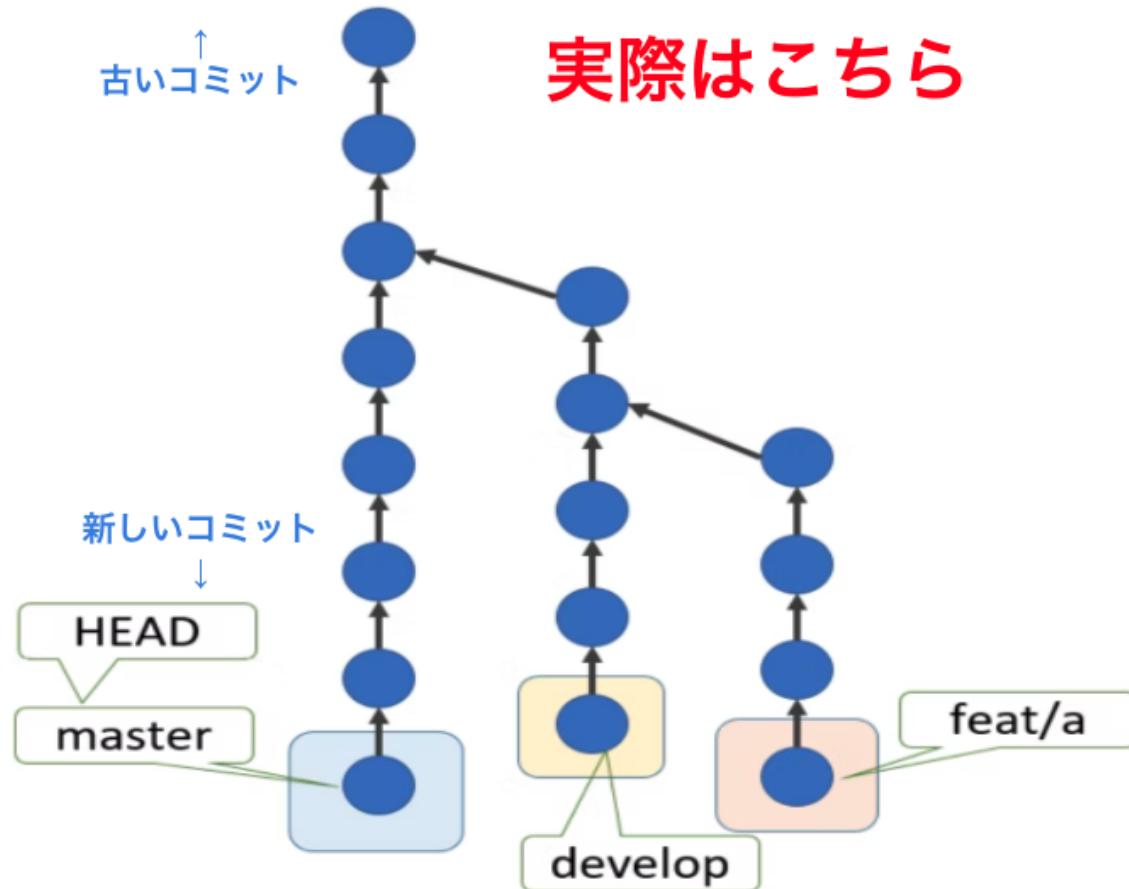
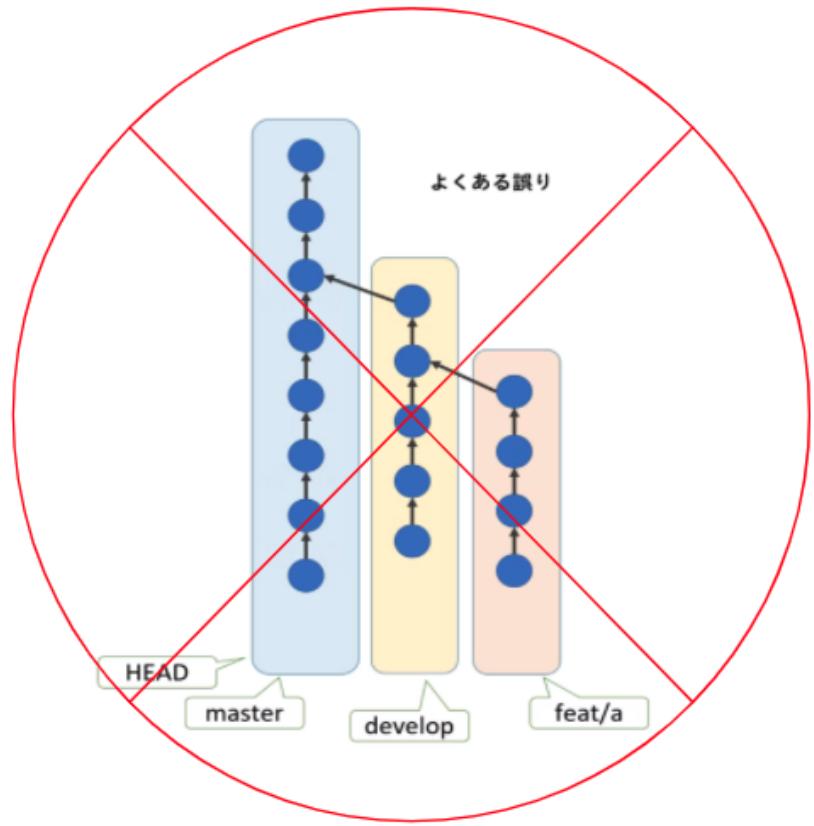
## 1.b. リポジトリの構成



# 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



[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！

# 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

## 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

以下で指定している対象ブランチ、何が違うか分かりますか？

## 1. fetch + merge で指定する対象ブランチ

```
$ git fetch origin develop
```

```
$ git merge origin/develop
```

## 2. merge で指定する対象ブランチ

```
$ git merge origin/develop
```

```
$ git merge develop
```

違い分かれますか？

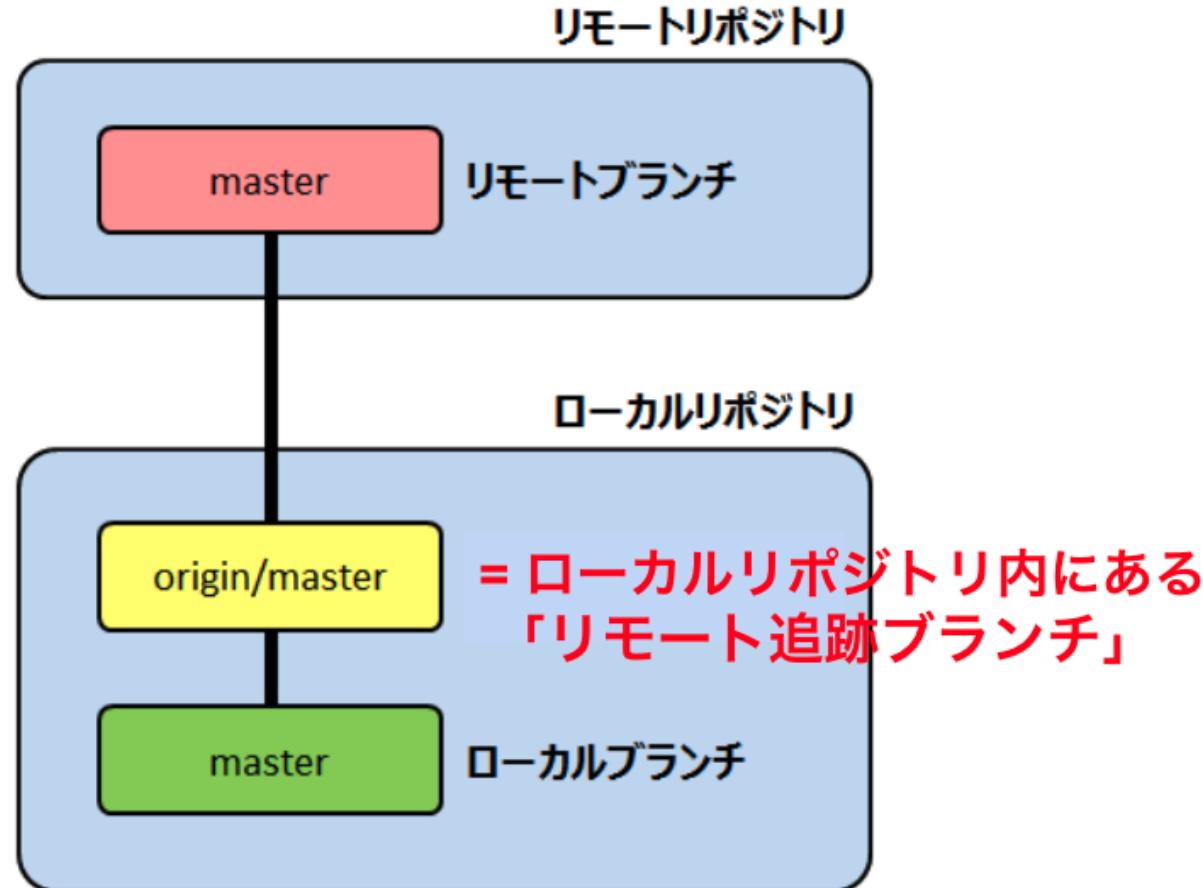
# 復習: origin(リモートリポジトリ)と origin/develop(リモート追跡ブランチ) git

## 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

## 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

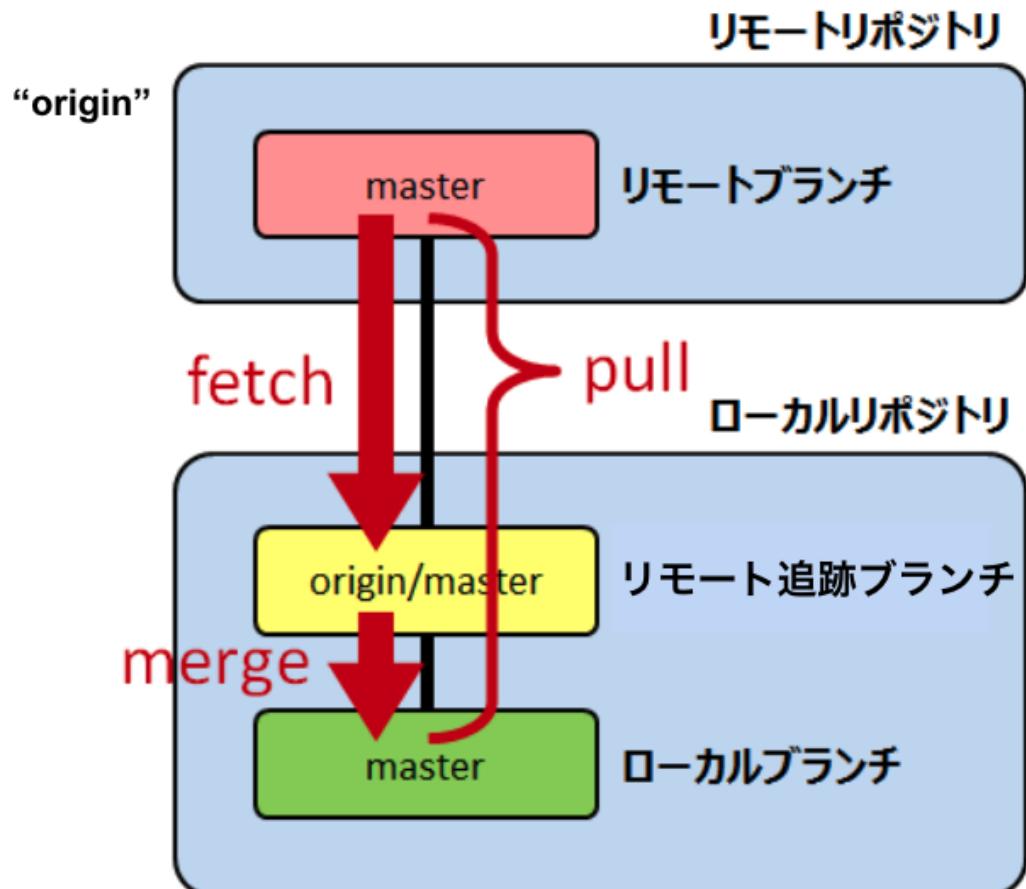
# 復習:pull と fetch + merge の違い



## 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



// TODO: Fast-forward merge と Auto merge の違いについて(不要?)

コマンド説明(★ 付きコマンドのみ当日説明します)

# pull

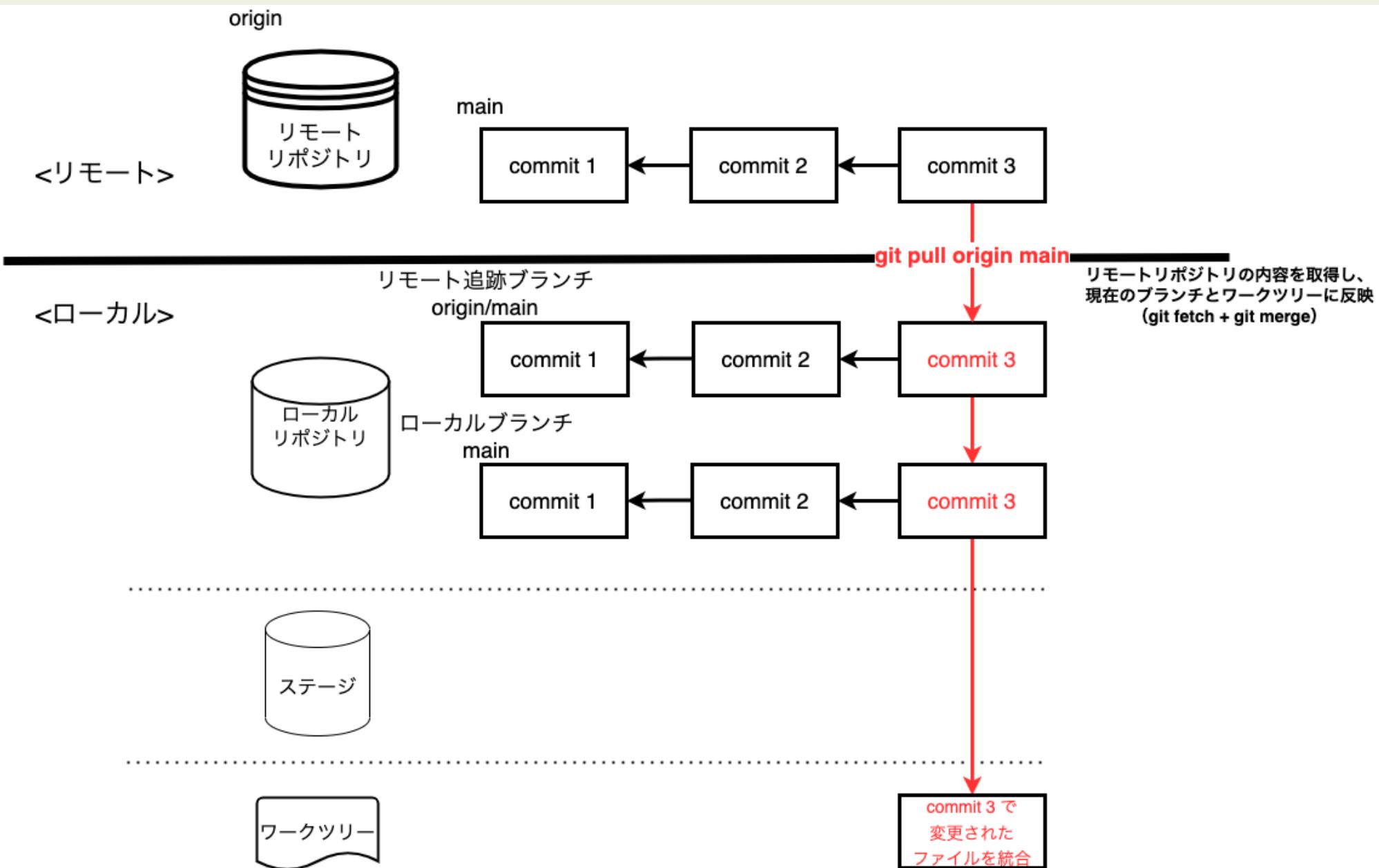
## 機能

- 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

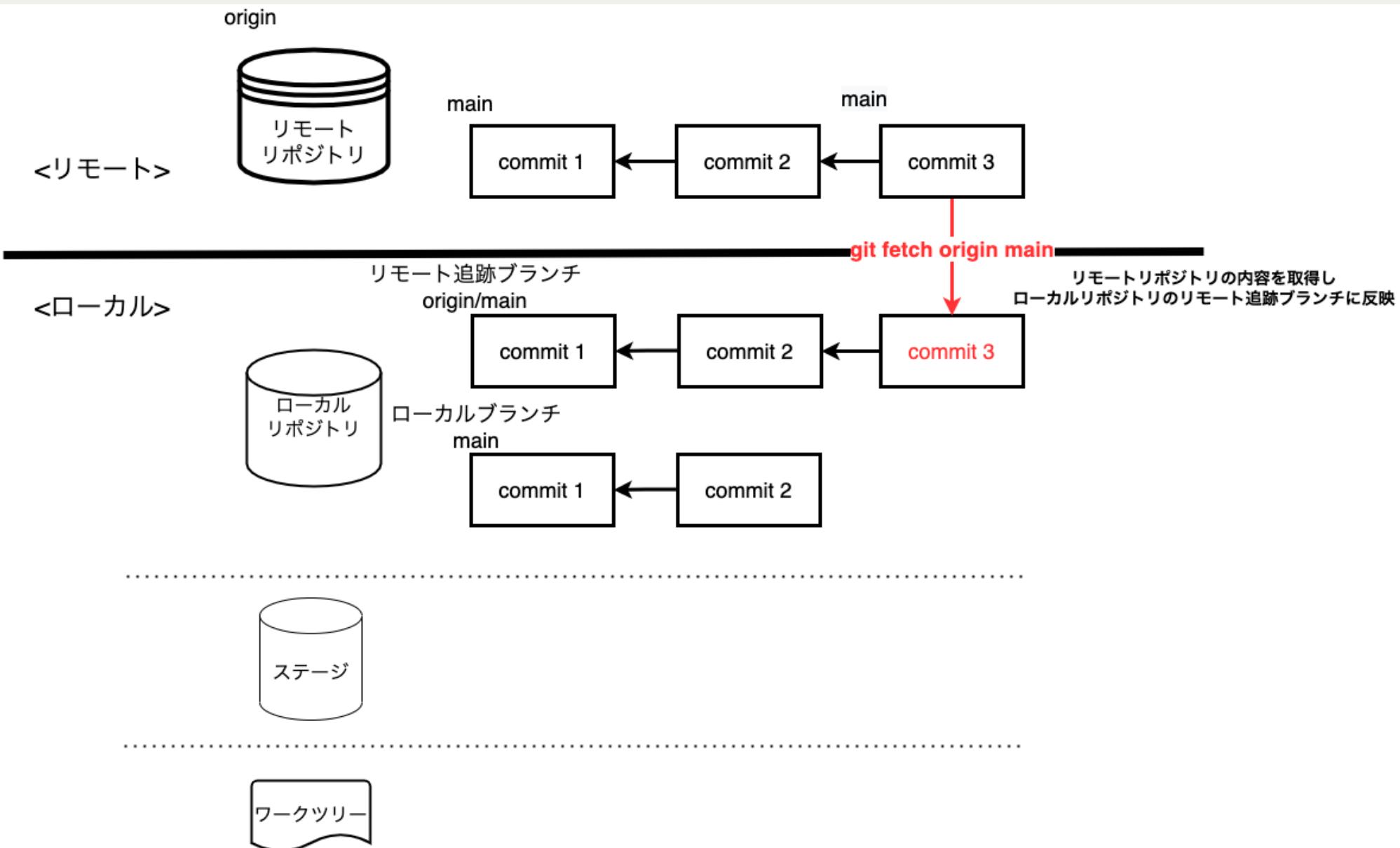
## 機能

- 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





## 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
```

## 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](#)

# merge

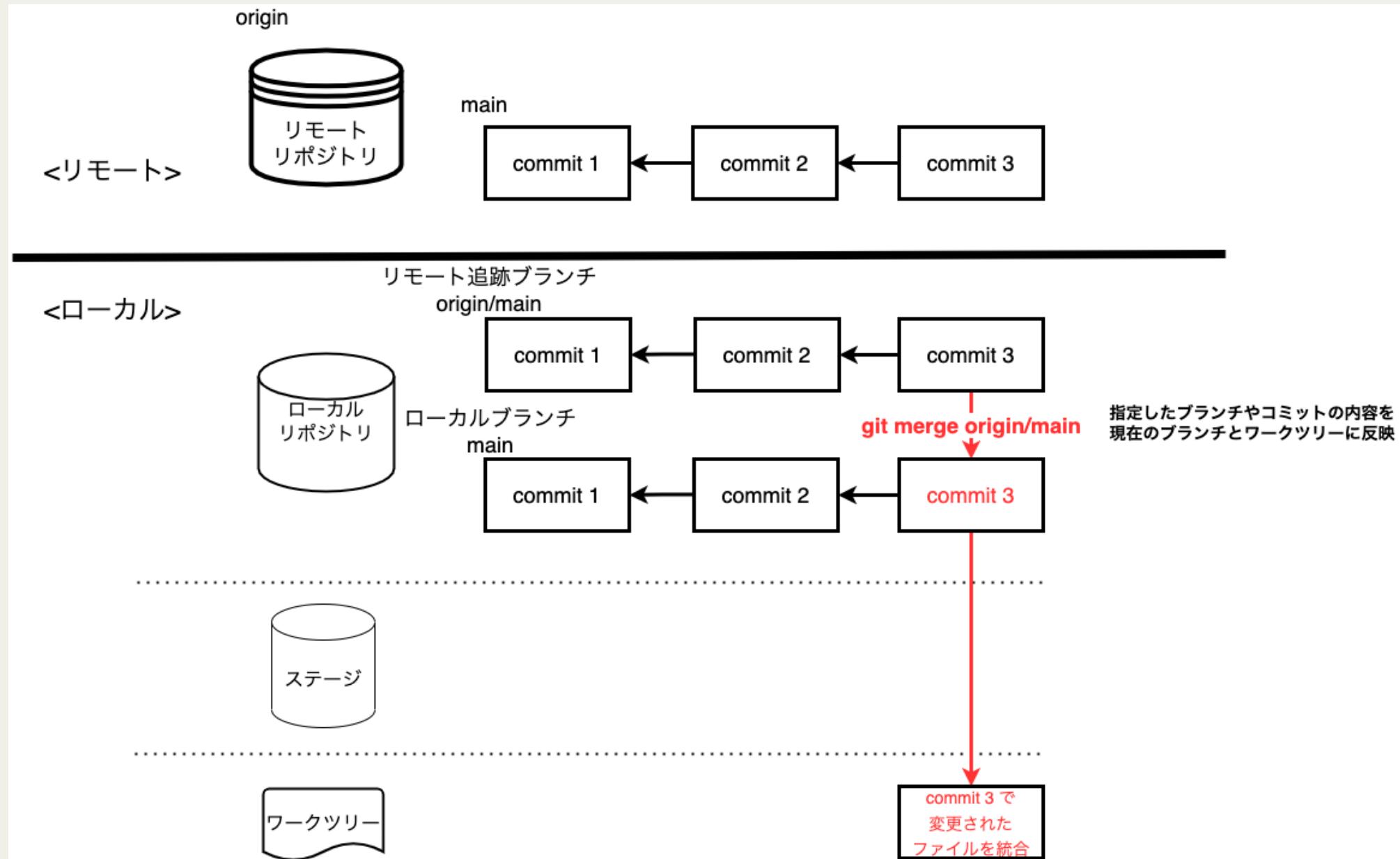
## 機能

- 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

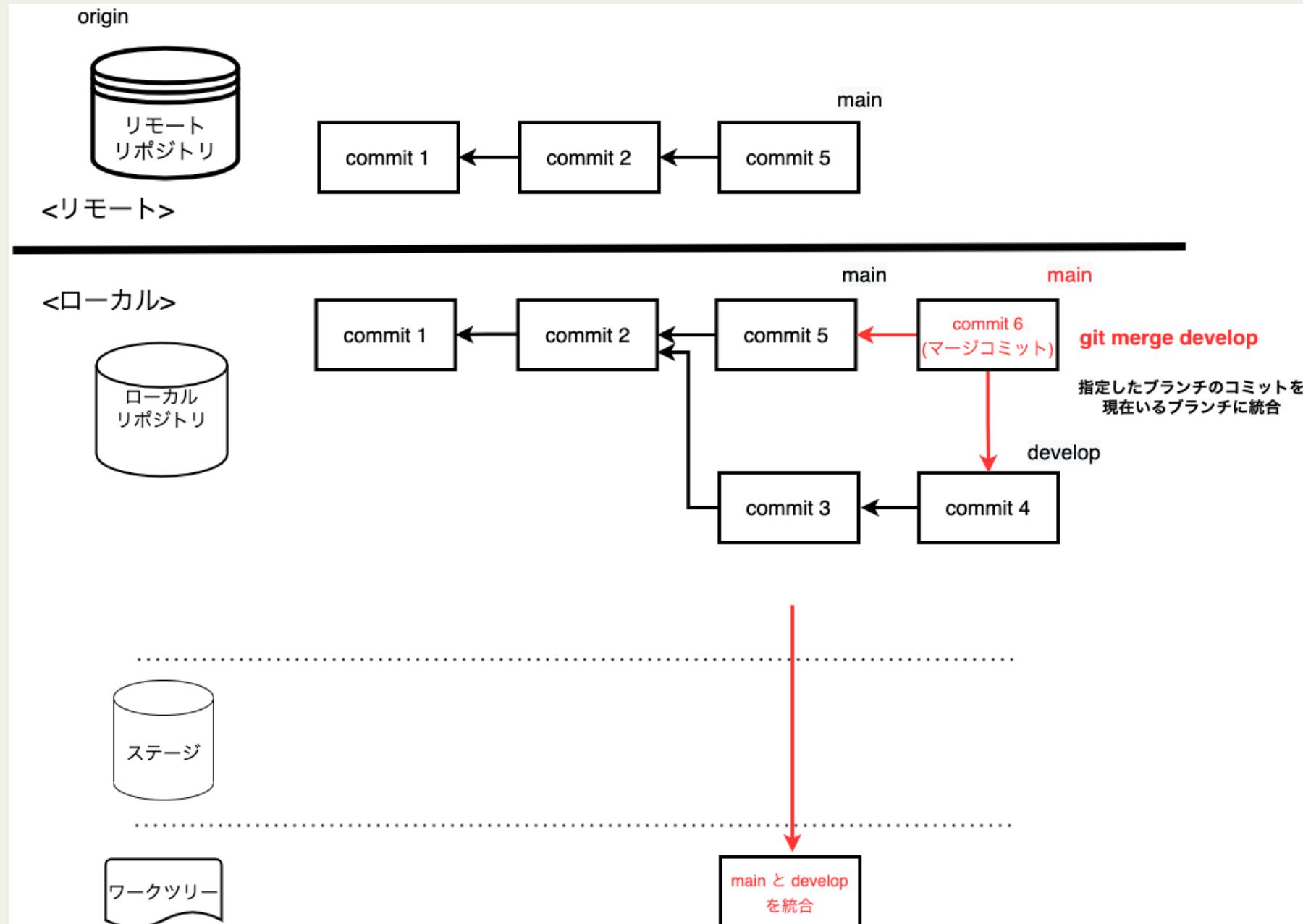
## ユースケース

- fetch した内容をワークツリーに反映したい(ブランチをマージしたという記録をコミット履歴に残したい)
- 2 つのブランチを 1 つに統合したい
- 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 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>
$ git merge <マージ元のブランチ名>
```

## 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!](#)

# rebase

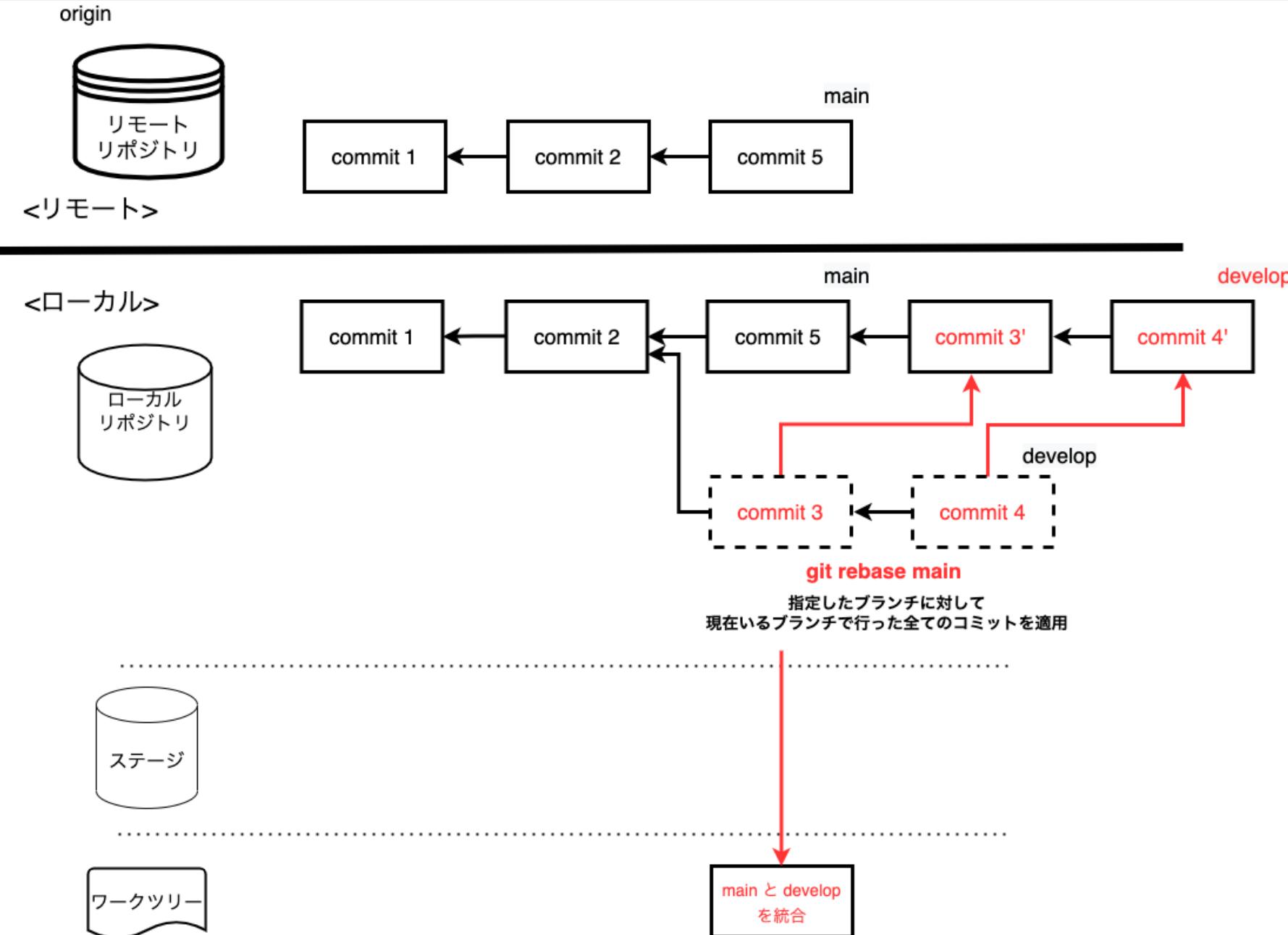
## 機能

- ・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つに統合したいが、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が、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
- ・複数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` : 複数のコミットを統合

## 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 はどっちがいい?

## A. 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</li><li>- 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</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></ul>

※ よく聞く merge のメリットで「コンフリクトの解決が比較的簡単」(rebase だとコミットを 1 つずつ適用し、それぞれで解消が必要なため)というものがあるが、個人的には rebase の方がコンフリクト解消は簡単な気もする。巨大なマージコミットを相手にコンフリクト解消するのは文脈の理解がしづらいので。

## 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
  - 上記記事に対する @kazuho さんのコメント
- git の merge --no-ff のススメ
- 【Git】将来の自分を救うのは、rebase だと僕は思うよ

# stash

## 機能

- 未コミットの変更を退避する

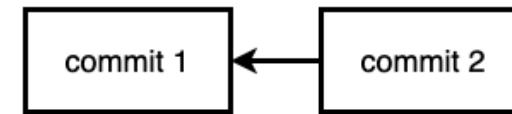
## ユースケース

- 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

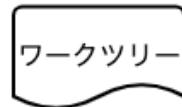
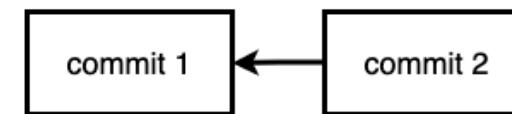
# イメージ



<リモート>



<ローカル>



stash

1つ前のstash  
stash\${1}

直前のstash  
stash\${0}

変更した  
ファイルB

変更した  
ファイルC

変更した  
ファイルA

変更した  
ファイルB

変更した  
ファイルC

git stash

未コミットの変更を退避する

git stash pop  
git stash apply@stash{1}

退避した変更を戻す

## 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 -u push "<メッセージ>" # 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} # 直前からN番目に退避した変更を戻す。stash@{N} を省略した場合は直前に stash した情報を戻す
$ git stash show stash@{N} -p # 直前からN番目に退避した変更の詳細を見る
$ git stash drop stash@{N} -p # 直前から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

# イメージ



<リモート>



<ローカル>



`git restore --source commit1 .  
(git restore --source HEAD~2)`

対象ファイル全体を特定の  
コミット時点の状態に復元する

変更した  
ファイルB  
変更した  
ファイルC

`git restore --staged .`

インデックスにある対象ファイル全体を  
ワークツリーに戻す

変更した  
ファイルB  
変更した  
ファイルC

`git restore .`

ワークツリーにある対象ファイル全体を  
変更前に戻す（※破壊的操作）



## 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

// TODO: checkout の例を追記

```
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

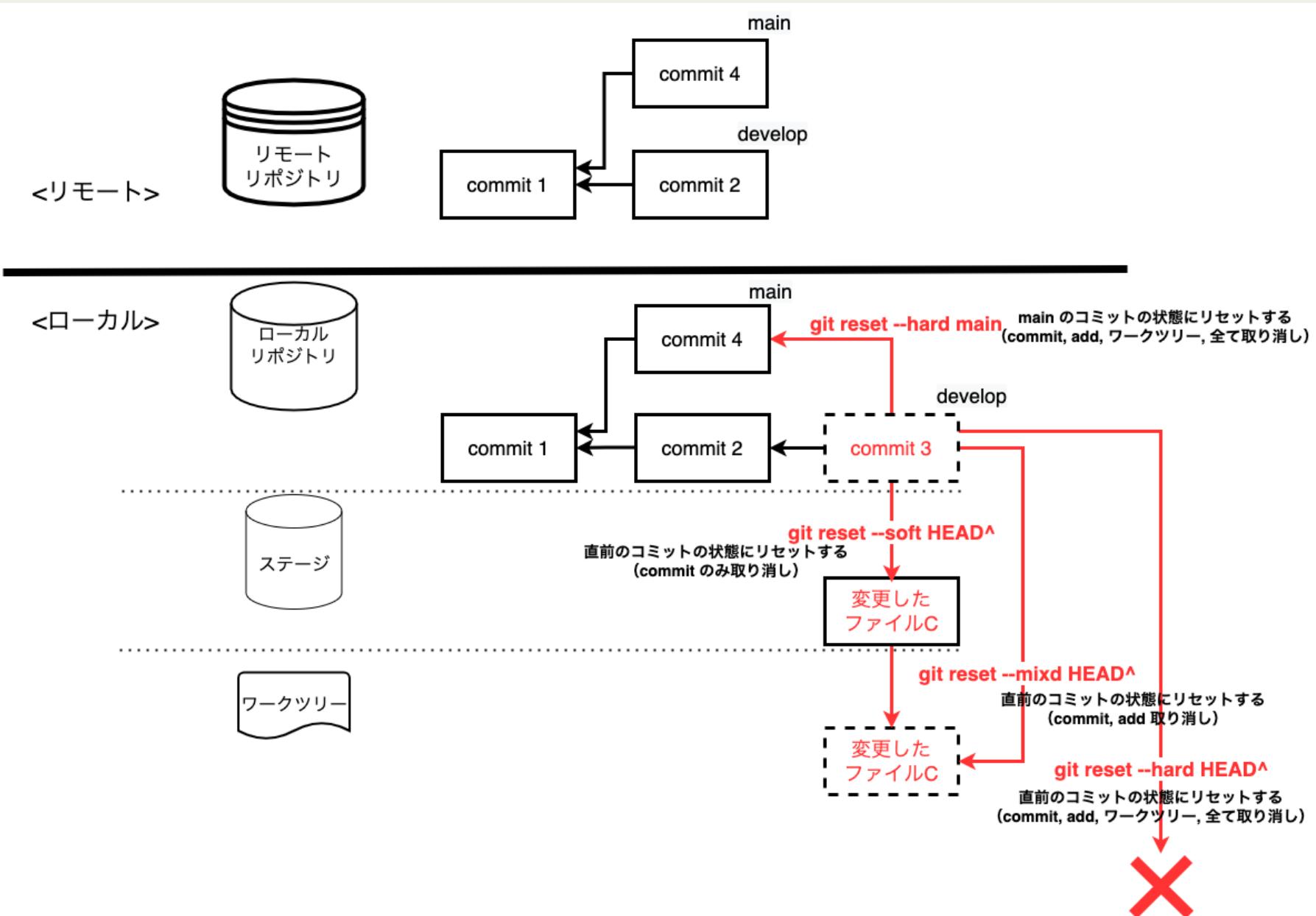
### 機能

- ファイルをインデックスから削除し、特定のコミットの状態まで戻す

### 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

# イメージ



## 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戻す (**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全部取り消し**)

## コマンド例

```
$ git reset --soft HEAD^ # 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^ # 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^ # 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 昔のコミットのハッシュ値 # 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)
```

## 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～](#)

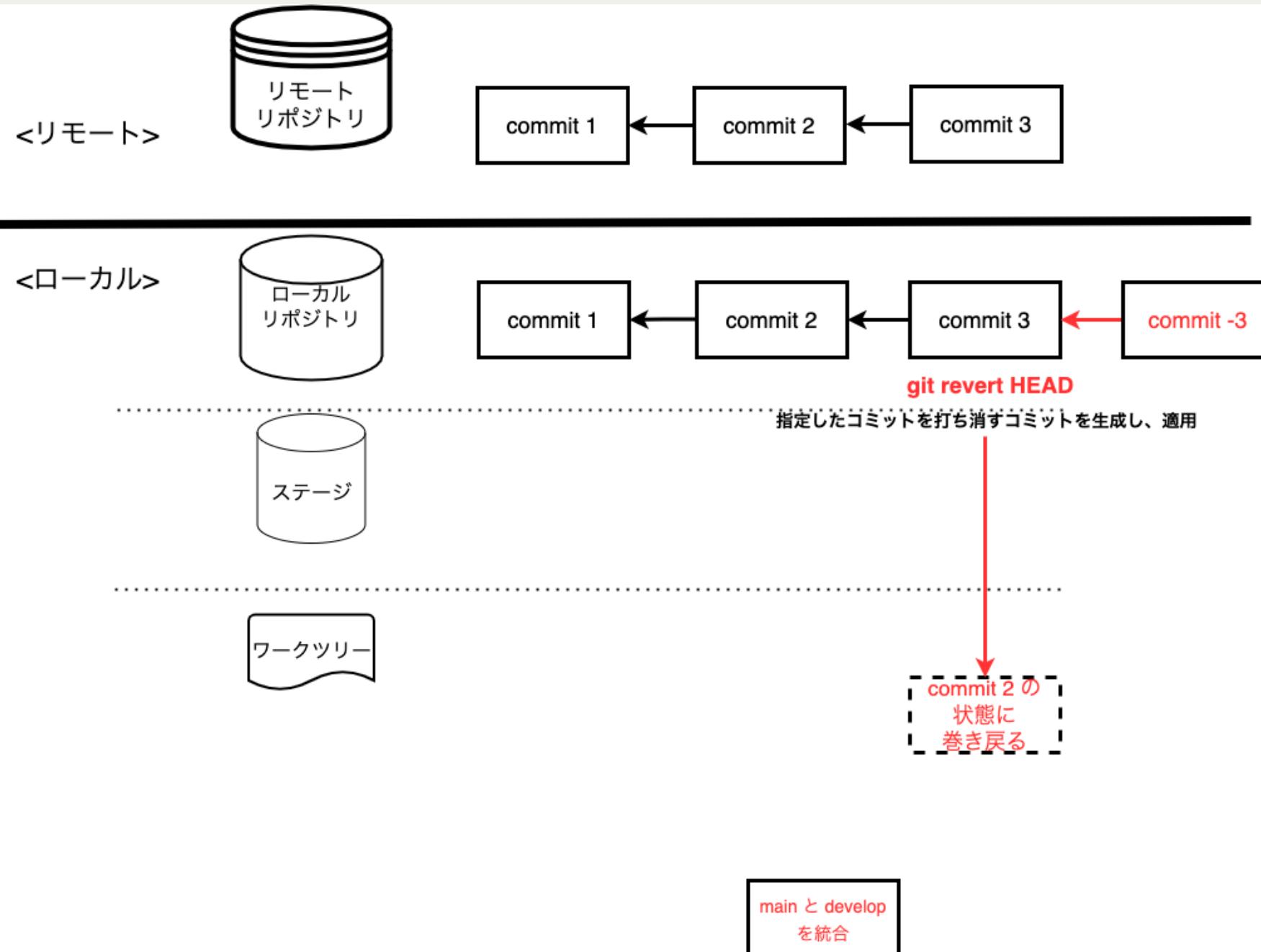
## revert

### 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)





## 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

## 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
```

## 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](#)

# cherry-pick

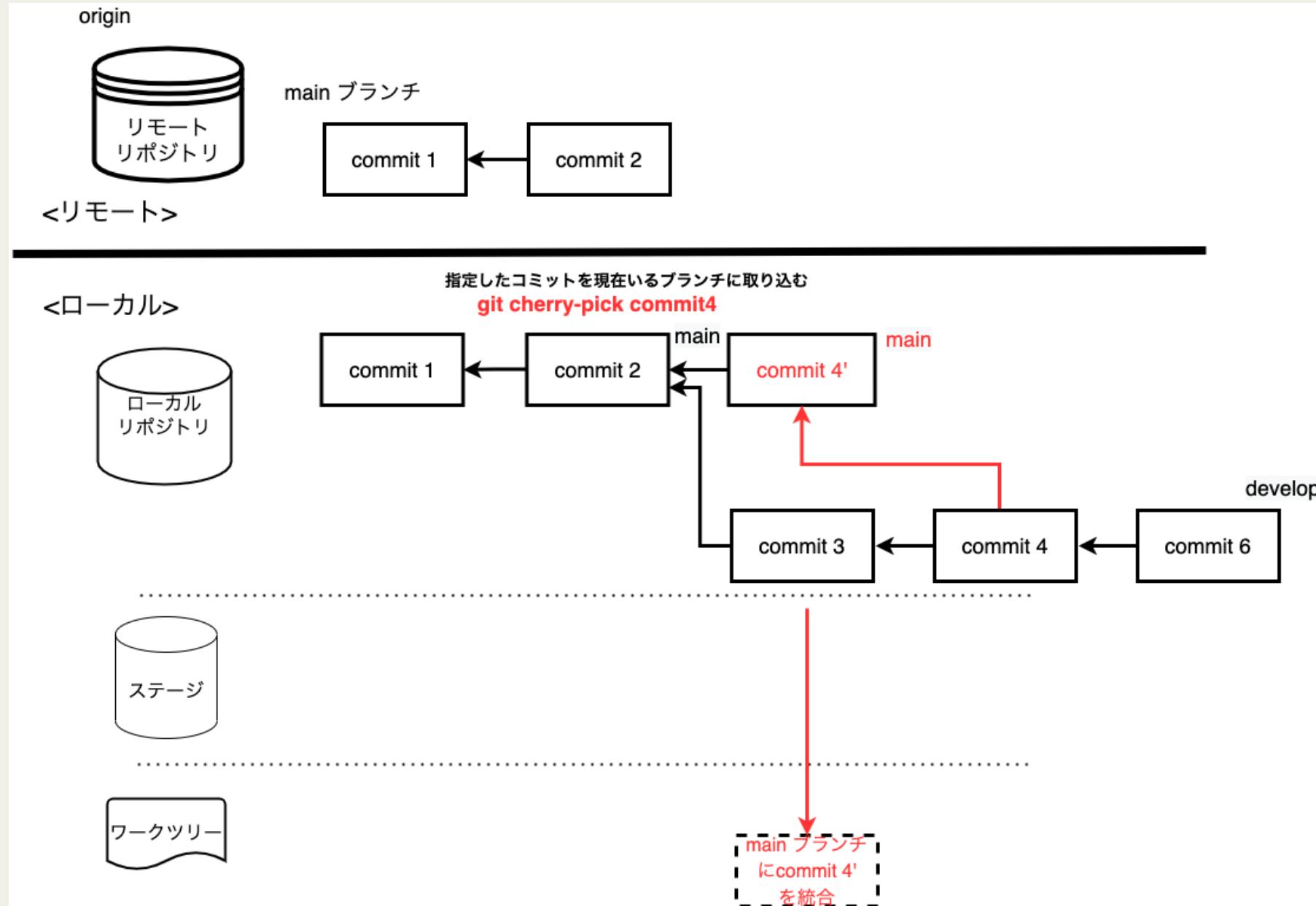
## 機能

- 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(複数 PR を並行してレビューしているときなど)

## 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` : 指定したコミットを取得するが、適用しない

## 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> # 特定のコミットを現在のブランチに取り込む（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

- この行は誰が変更したのか調べたい

## 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

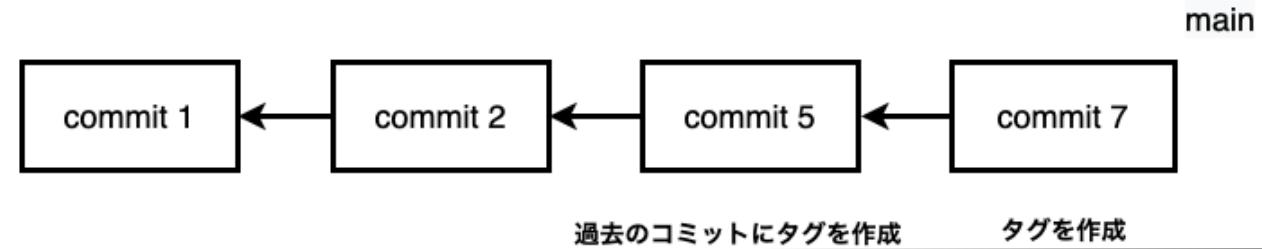
# イメージ



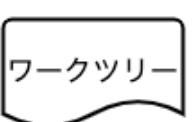
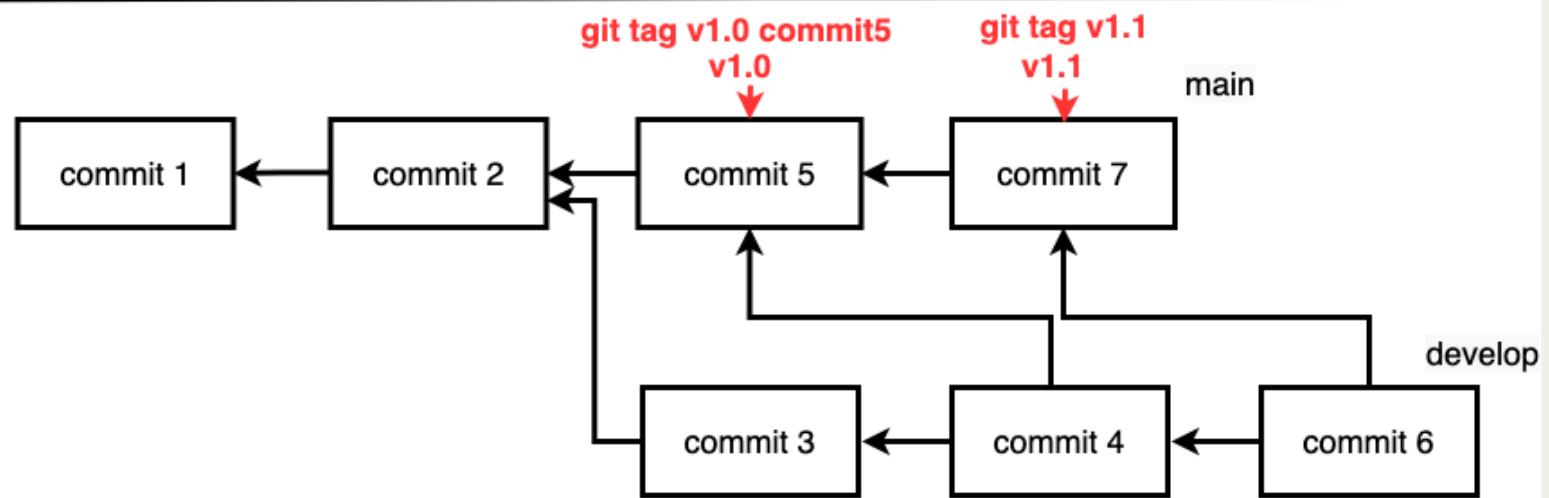
origin



<リモート>



<ローカル>



## 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

- ・ ブランチのヒントやその他の参照がローカルリポジトリで更新された時期を記録する

## 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
- ・ 間違えて `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 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@{1} --hard # reflog で元に戻したいコミットを指定して、reset --hard で戻す
```

## 参考

- [git-reflog Documentation](#)
- [いざという時のための git reflog - Qiita](#)
- [初心者から一步抜け出すための Git の業 ~ git reflog - Qiita](#)
- [git reflog を日時で参照する](#)

# ハンズオン

// TODO: コンテンツ作成(★は一緒にやる予定)

1. pull ★
2. fetch + merge
3. merge (branch) ★
4. rebase ★
5. stash ★
6. reset ★
7. revert
8. cherry-pick ★



# 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！