数値計算屋のためのGit入門

2020/04/10

慶應義塾大学理工学部物理情報工学科
渡辺

Gitとは何か

バージョン管理システムの一つ

(Version Control System: 略してVCS)

- バージョンを管理してくれる
- 変更点を後から見やすくしてくれる
- 多人数による開発を容易にしてくれる

プログラム開発現場ではVCSの導入は必須

いまどきVCSを使ってない開発現場は無い……と思う

バージョン管理システムのご利益

☑ 編集の歴史を保存し、いつでも過去に戻ることができる

「しまった！」を「なかったこと」にできる
「以前は動いてたのに」を再現できる

☑ バックアップがわりになる

※ リモートリポジトリを別サーバに用意した場合

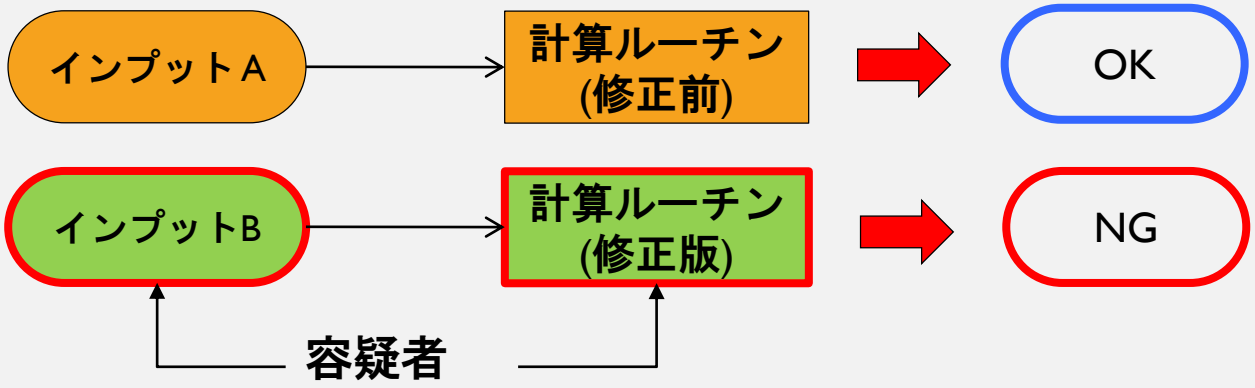
修士論文提出直前にPCが壊れた
USBに保存してたデータが読めなくなった

↑ こういう悲劇を防ぐ

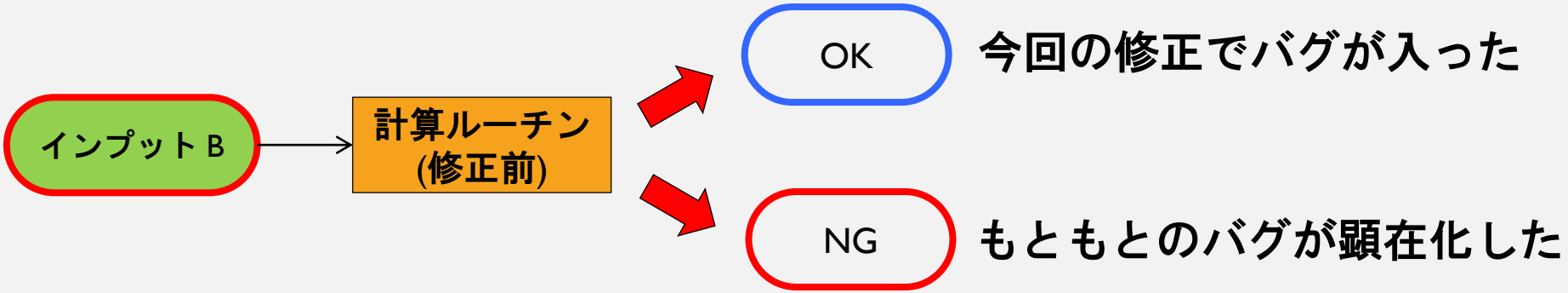
バージョン管理システムのご利益

機能を追加し、別のインプットを与えたら計算が失敗した

→機能を追加したことによるバグ？もともとバグっていたものが顕在化？



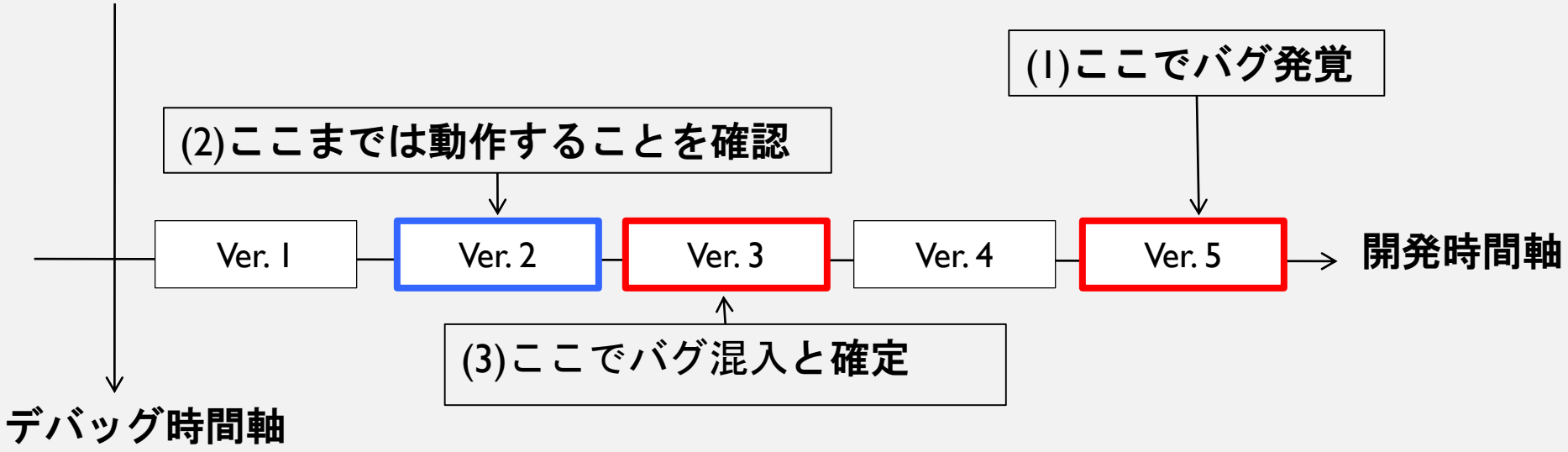
機能追加前のソースを取って来て、Input Bを食わせる



バージョン管理をしていると、問題の切り分けが容易

バージョン管理システムのご利益

いつの間にかバグが入っていて、いつ入ったバグかわからない
→「歴史」を二分探索



Ver. 2とVer. 3の差分をとれば、何が原因かがすぐにわかる

「容疑者」を絞るのは捜査の基本

開発時間のほとんどはデバッグ

年に二編論文を書きたい→ 半年で一つの研究を完結させたい

半年

調査：先行研究の調査や、計算手法についての調査 (1ヶ月)
開発＋計算：プログラム開発、計算の実行(4ヶ月)
執筆：結果の解析＋論文執筆＋投稿 (1ヶ月)

実態は・・・

デバッグの時間を減らすことが最も効果的な「高速化」
バージョン管理システムはデバッグ時間を減らす強力なツール

Gitは簡単？

正直な話、Gitは簡単ではない

- コマンドが多い
- 使い方に自由度が高い(人によって違う)
- よくわからない状態になりがち

慣れるまで時間がかかると思って、根気よく使ってみよう
使い慣れると、無い生活は考えられません

とりあえず覚えたいコマンド

ローカルリポジトリの操作

- git init
- git add
- git commit

状態や歴史の確認

- git status
- git diff
- git log

ブランチの操作

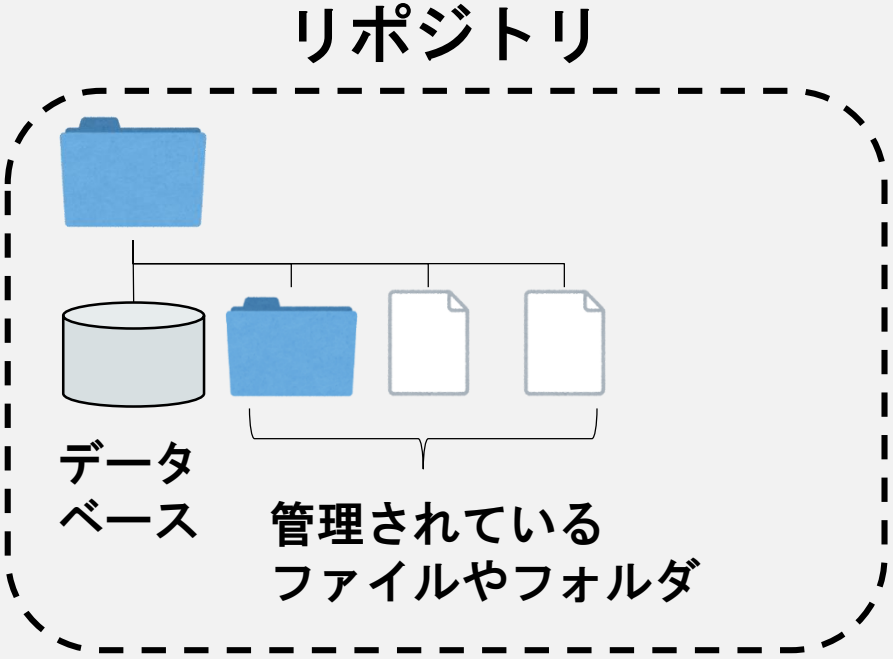
- git checkout
- git merge

リモートとのやりとり

- git clone
- git remote
- git fetch
- git push

「とりあえず」でも
こんなにある

Gitの仕組み

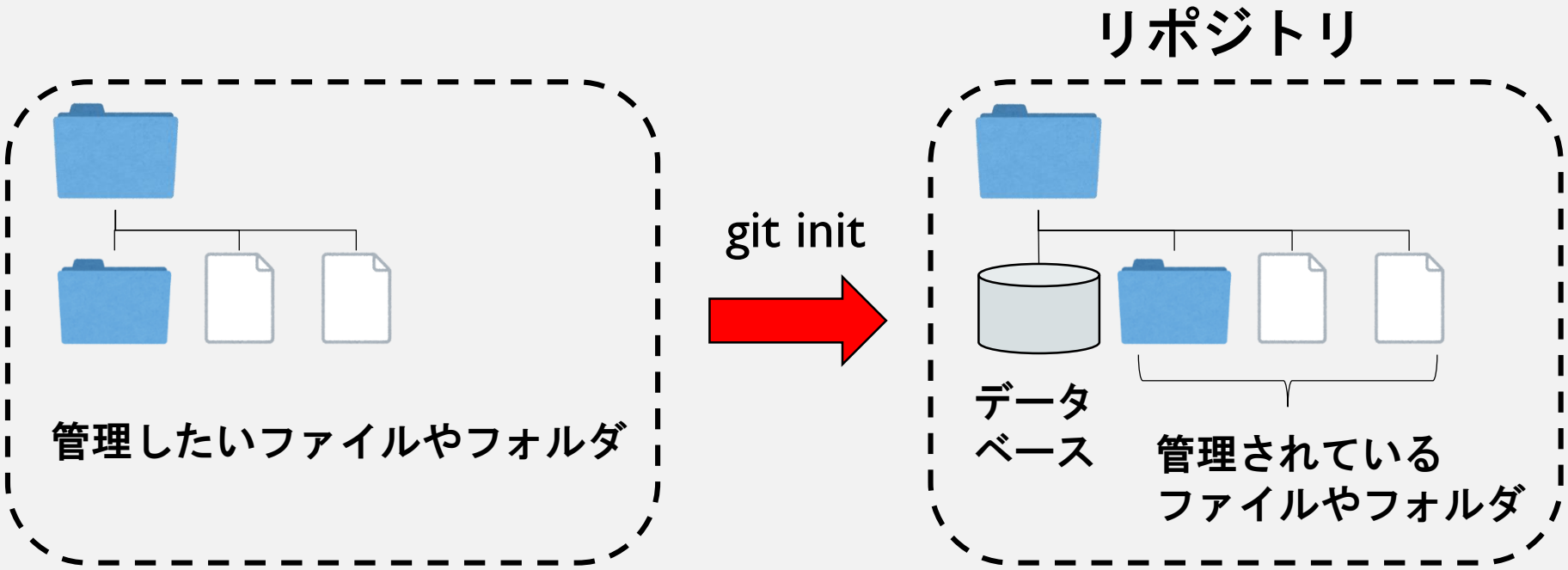


- 「リポジトリ」という単位で管理する
- リポジトリごとに「データベース」がある
- データベースには「歴史」が保存される
- 「コミット」により、「歴史」が追加される

※ データベースの実体は .git というディレクトリに保存されている

git init - 管理を始める

管理したいファイルを含むディレクトリで git initを実行する



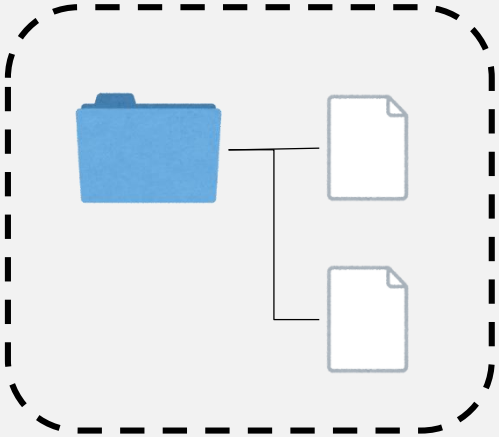
※ 実際にはgit addしてからgit commitしないとgitの管理下に入らない

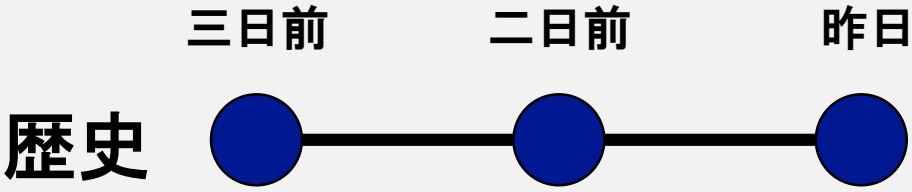
git commit - コミットとは

Gitでは「歴史」を丸と線で表現する

- 丸：ある時点の「状態」
- 線：二つの状態の関係(差分)

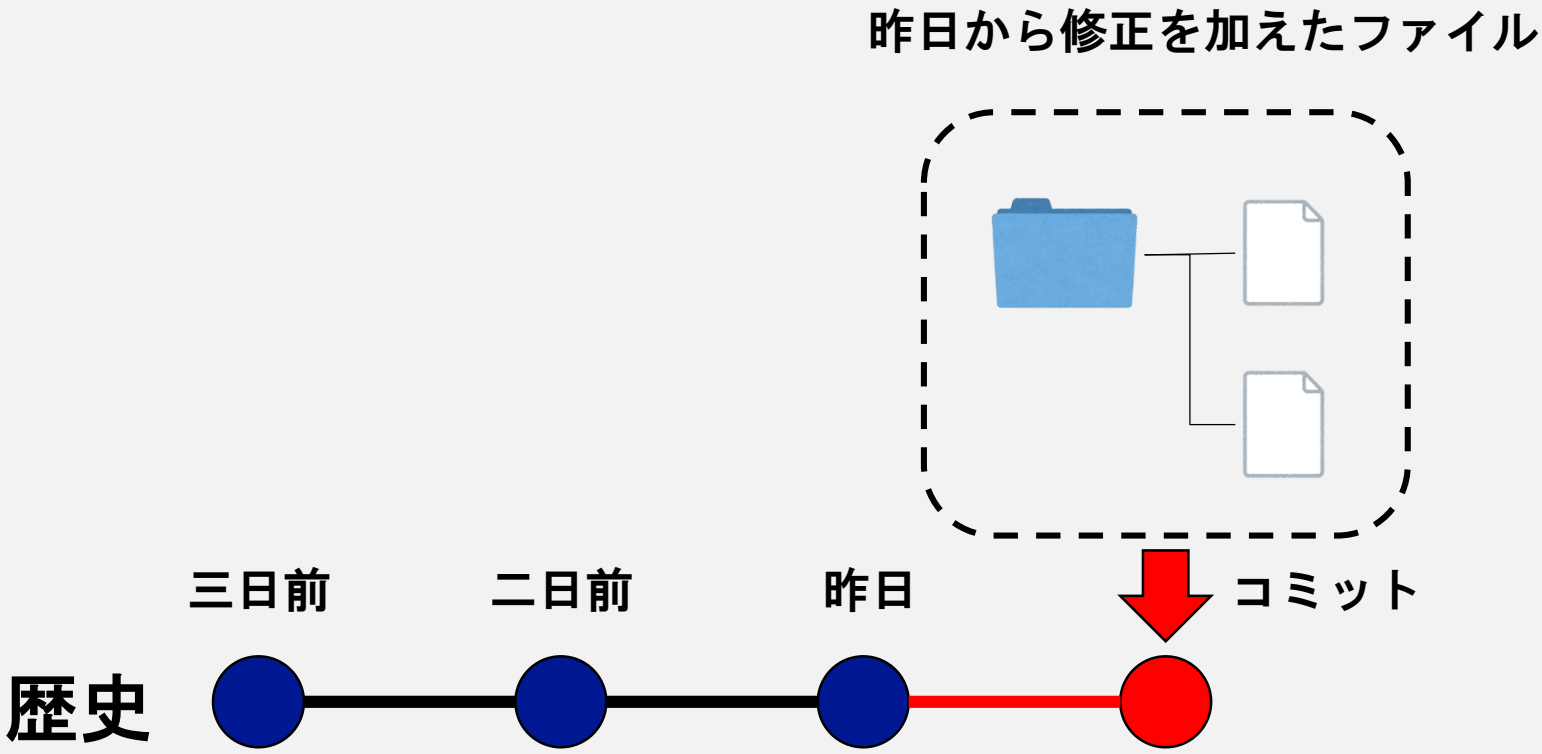
昨日から修正を加えたファイル



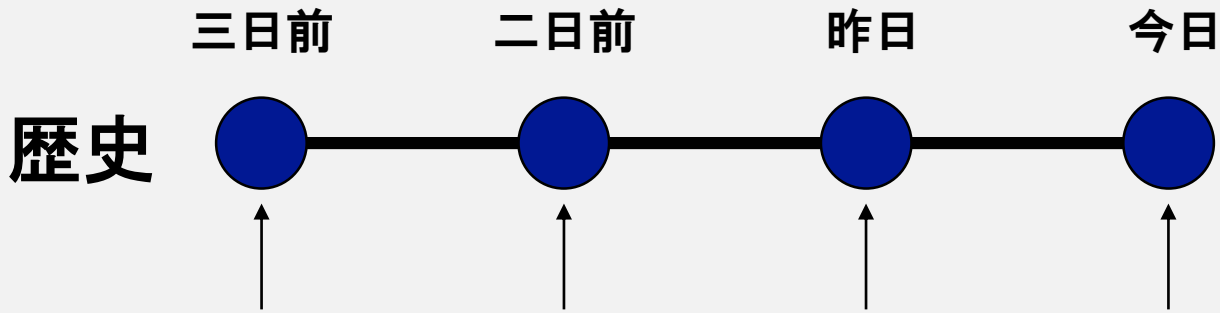


git commit - コミットとは

コミット：現在の状態を保存して「歴史」に加える



git commit - コミットとは



この玉それぞれを「コミット」と呼ぶ

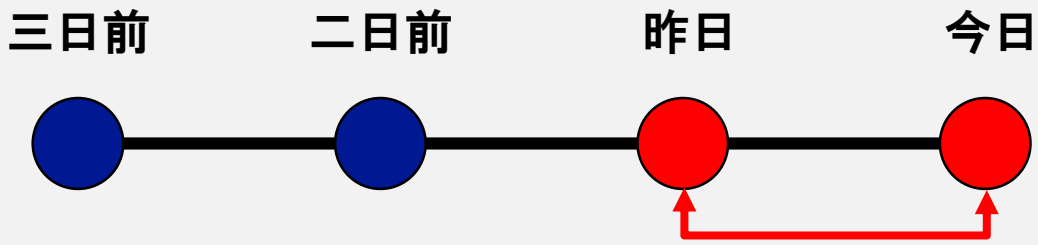
この玉を新たに作る作業を「コミットする」と呼ぶ

commit (名詞) : Gitの歴史のある「点」(スナップショット)
commit (動詞): Gitの歴史に新たにスナップショットを付け加えること

A Git Glossary https://git-scm.com/docs/gitglossary

歴史があるとできること

git diff - 任意の二点の「差分」が取れる




```
$ git diff HEAD^
  これは昨日書いた文章です。
-これは今日削除した文章です。
+これは今日追加した文章です。
```

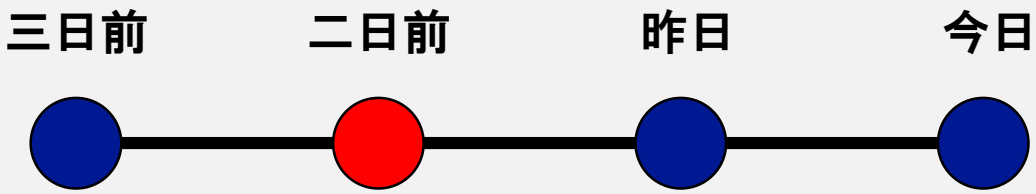
白地：変更なし
赤字：削除
緑字：追加

自分がいつどこを修正したか確認できて便利

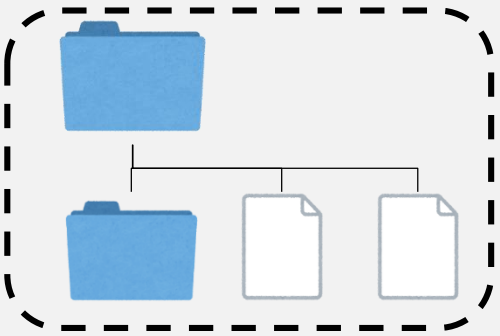
歴史があるとできること

git checkout - 任意の時間に戻ることができる

```
$ git checkout –b twodaysago HEAD^^
```



二日前の状態



「いつの間にか動かなくなった」時に「この時点では動く」ことを確認できる

歴史があるとできること

git log – これまでの履歴を確認できる

```
$ git log
commit 1ee64f77e9f32a947b0774eb2c82cd8da59aed40 (HEAD -> master)
Author: H.Watanabe <kaityo@users.sourceforge.jp>
Date:   Fri Apr 10 19:42:01 2020 +0900

    test2.txtを追加

commit 2a2ae2c7f601bf3d2a6d727745e57fa4a7de83b0
Author: H.Watanabe <kaityo@users.sourceforge.jp>
Date:   Fri Apr 10 19:41:26 2020 +0900

    test.txtを追加

commit 1a2da617b848413daee9b2880c2f7e6d201ed2b9
Author: H.Watanabe <kaityo@users.sourceforge.jp>
Date:   Fri Apr 10 19:41:06 2020 +0900

    initial commit
```

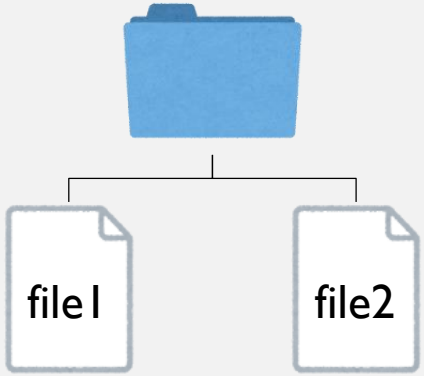
誰が、いつ、どんな修正をしたかを確認できる

git add

gitには、三種類のエリアがある

git init 直後の状態

ワークツリー	インデックス	リポジトリ



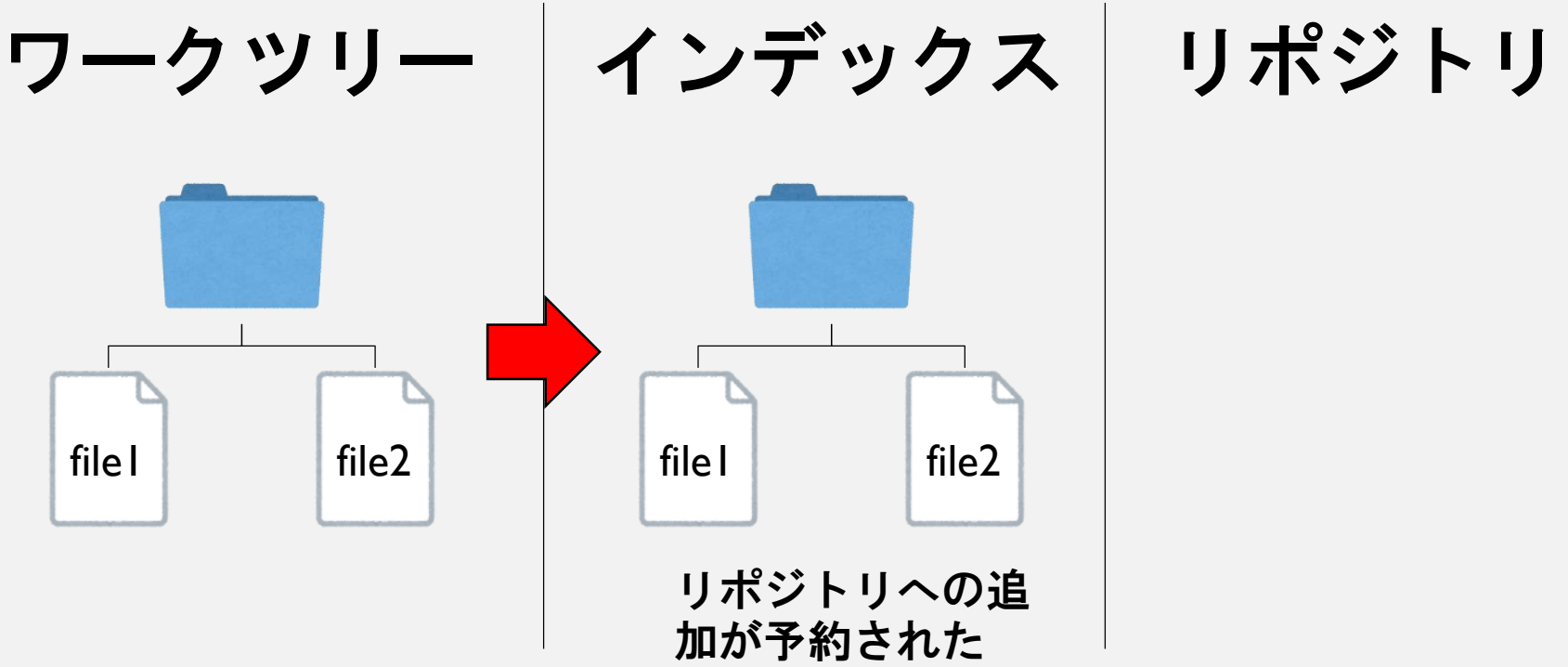
まだGit管理下に
置かれていない

いつやるの？Git入門 https://www.slideshare.net/matsukaz/git-17499005

git add

「git add ファイル名」により、インデックスに入る

git add file1 file2



いつやるの？Git入門 https://www.slideshare.net/matsukaz/git-17499005

git add

git commitにより、変更がリポジトリに記録される

git commit –m “file1とfile2を追加”

いつやるの？Git入門 https://www.slideshare.net/matsukaz/git-17499005

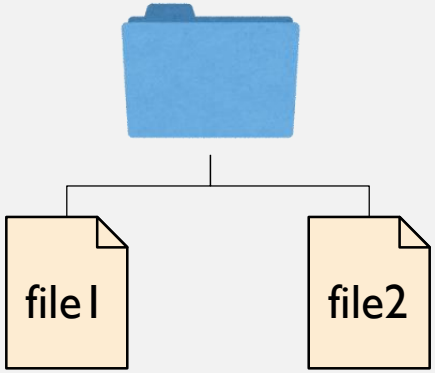
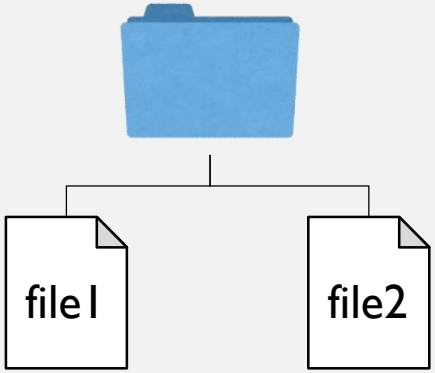
git add

Q: なぜインデックスがあるか？

A: 複数の修正がある時、一部の修正を選んでコミットを作るため

git add

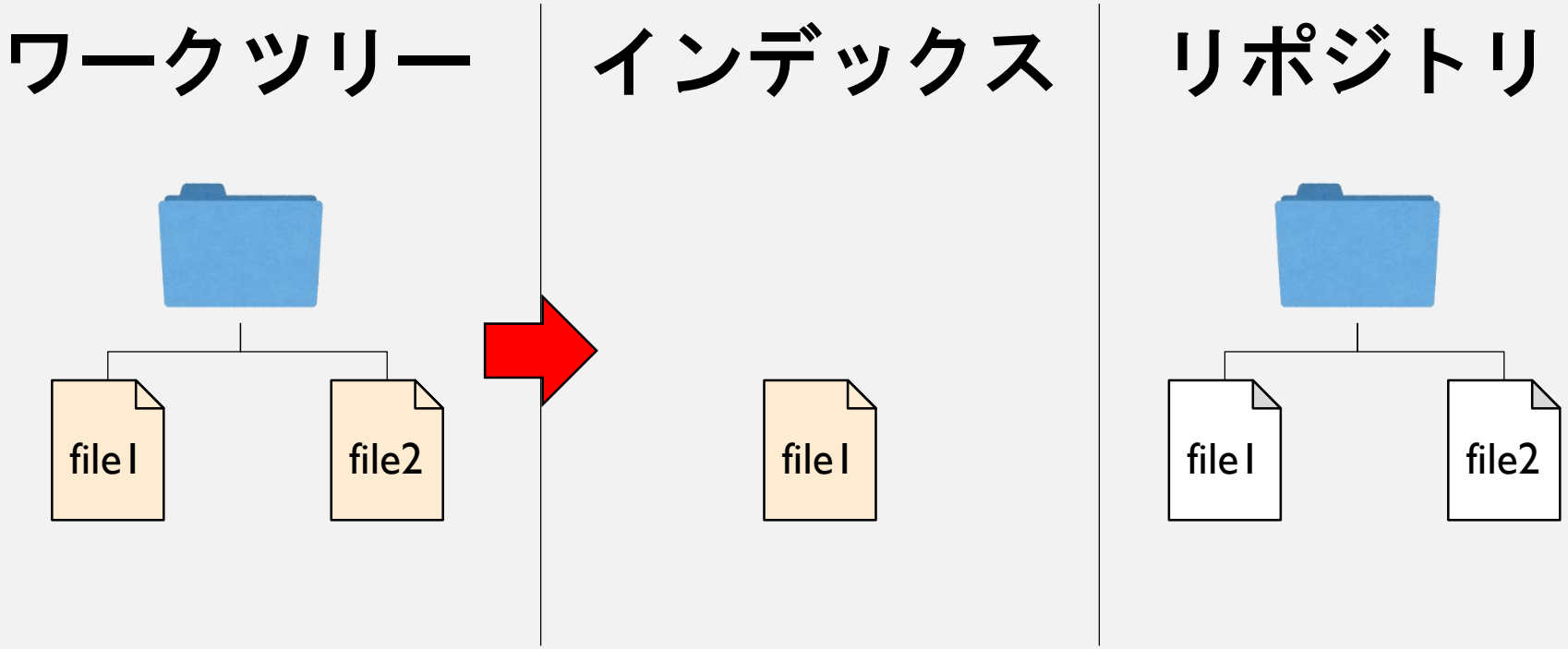
最後にコミットした状態から、file1とfile2を修正した

ワークツリー	インデックス	リポジトリ
		

git add

file1だけaddする

git add file1

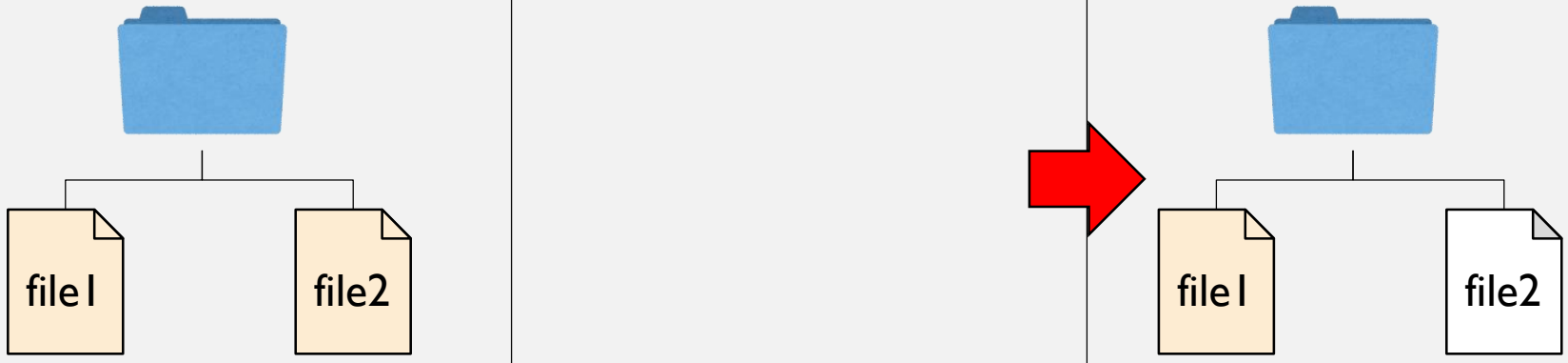


git add

コミットする

git commit –m “file1を修正”

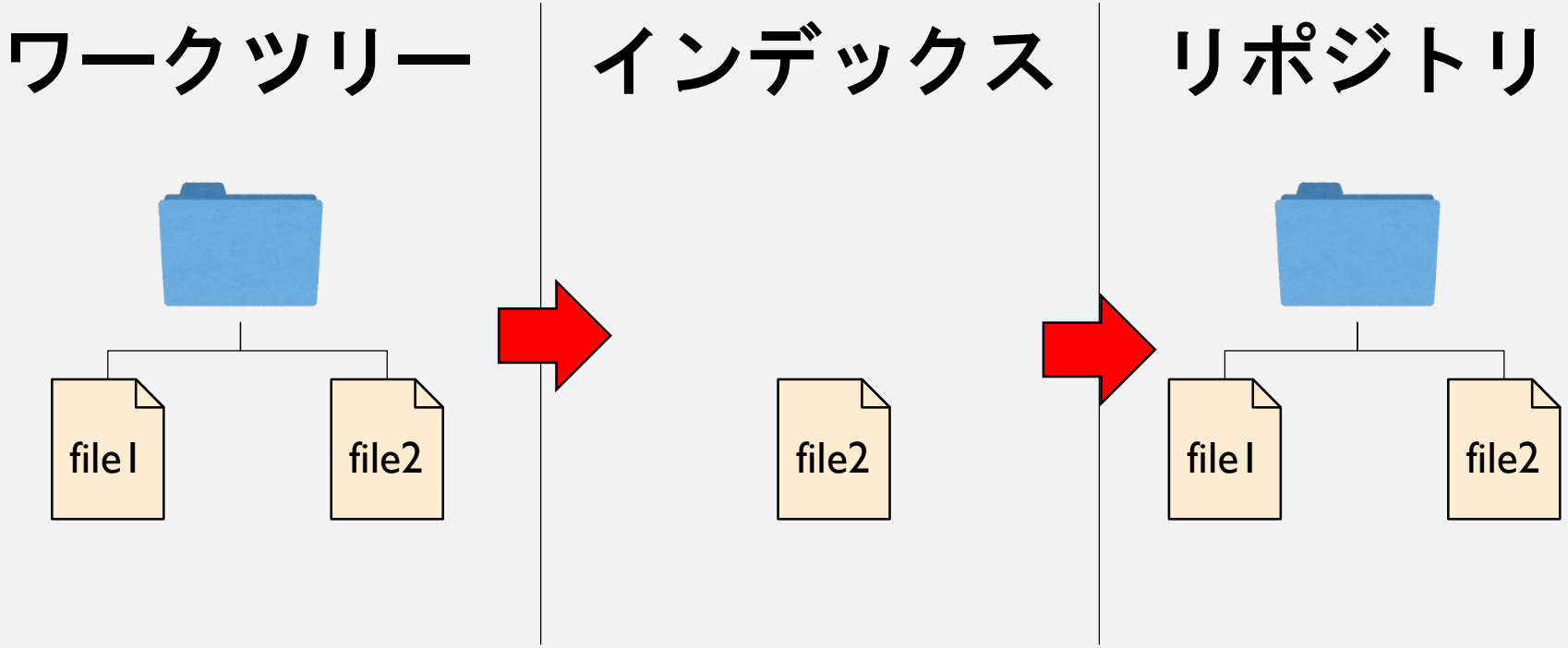
ワークツリー | インデックス | リポジトリ



git add

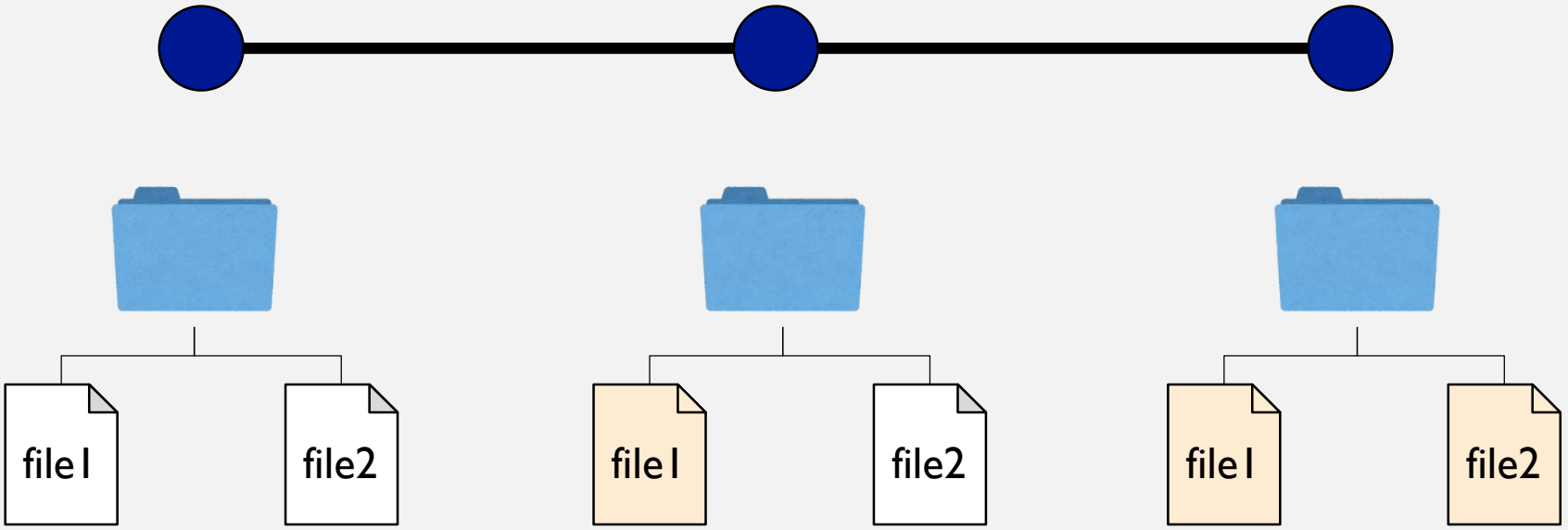
file2も同様にする

git add file2
git commit –m “file2を修正”



git add

こんな歴史ができあがった

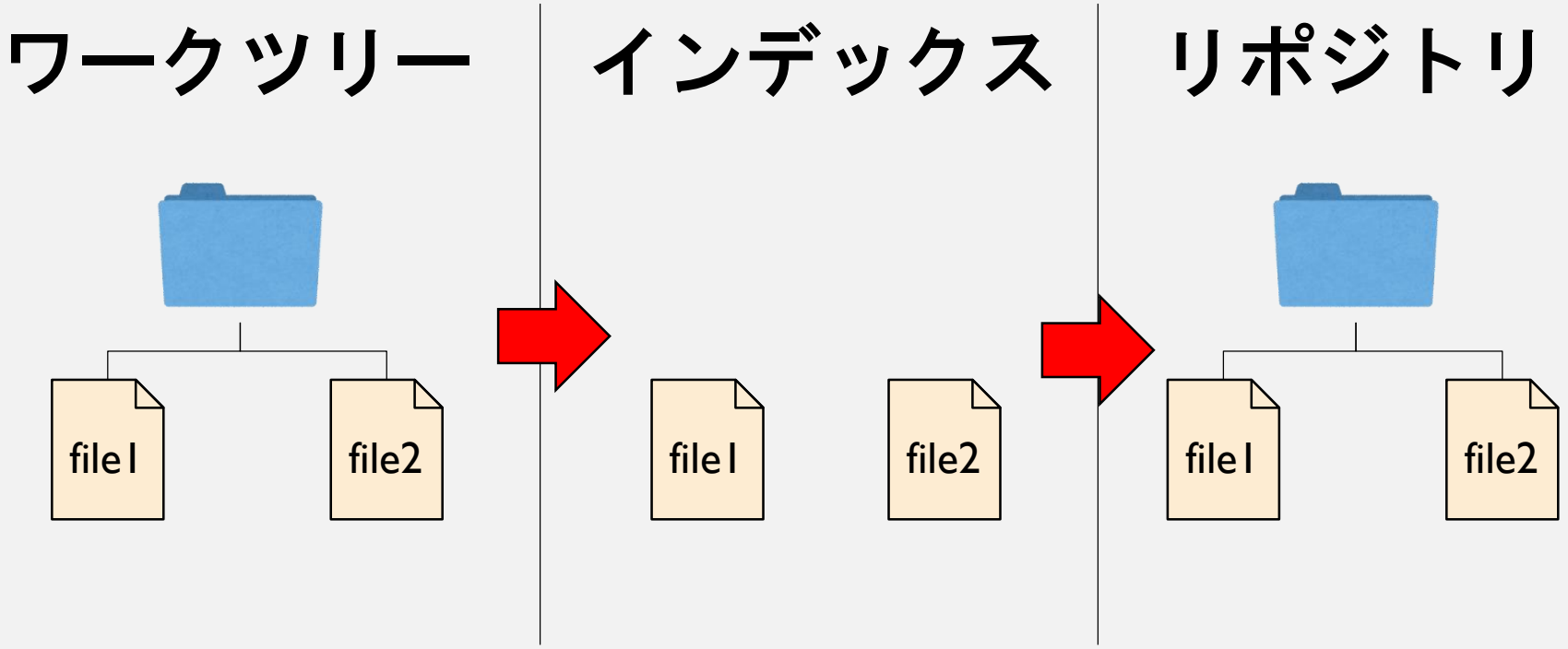


file1とfile2を追加　　file1を修正　　file2を修正

Gitでは積極的に歴史を作成、改変する

git commit -a

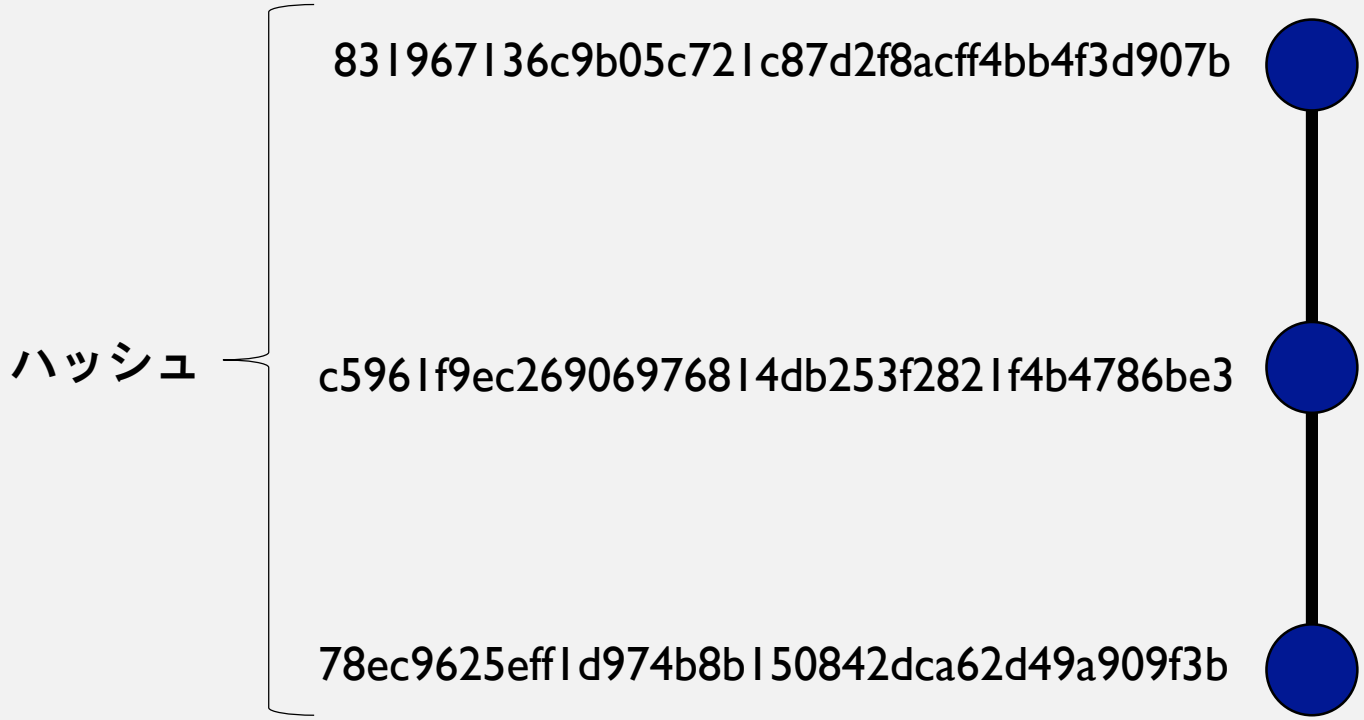
git commit –a オプションで
「修正があったファイルを全てコミットに含める」
ことができる(git addを省略できる)



一人で使っている時は、慣れるまではこれで良いと思う

ブランチ

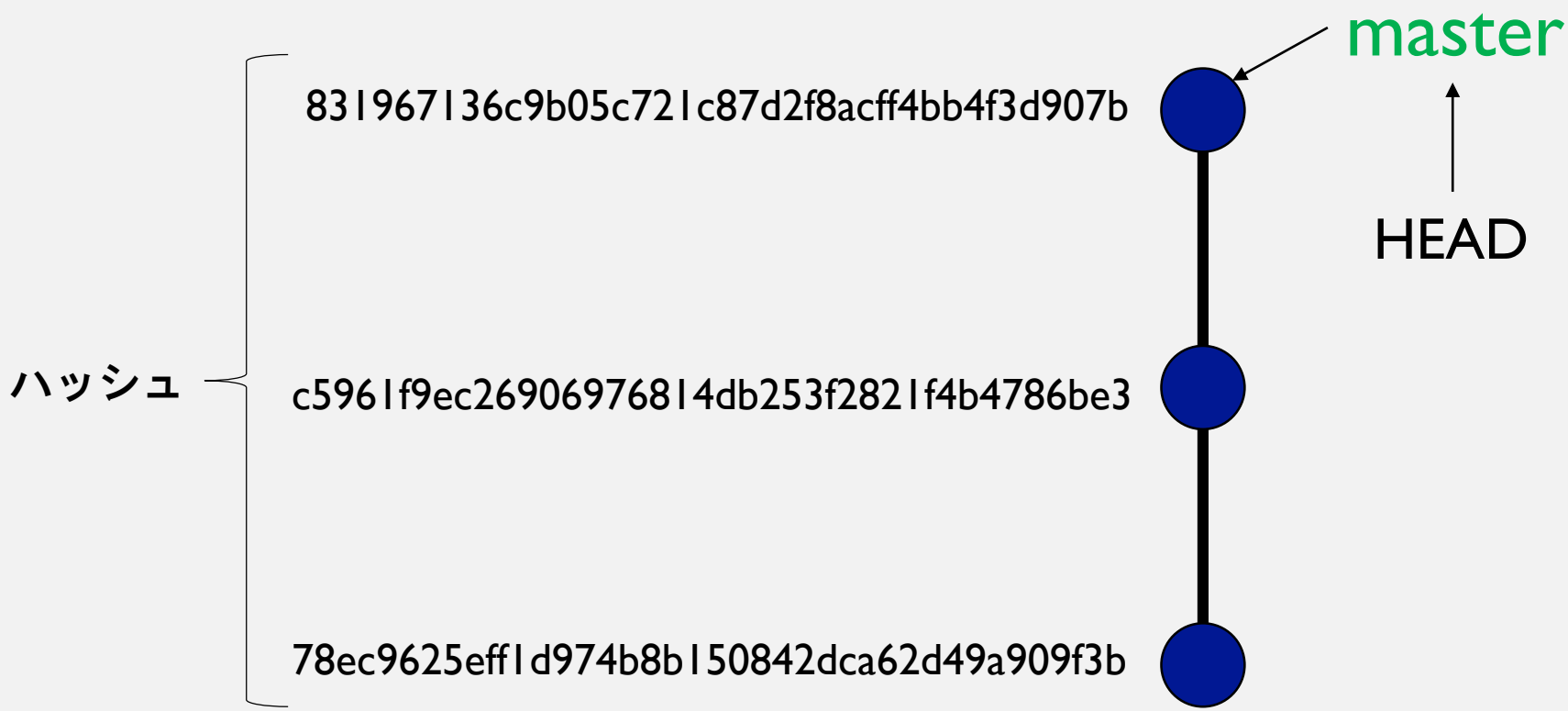
コミットが作成されると自動的に「ハッシュ」と呼ばれる識別子がつく



ブランチ

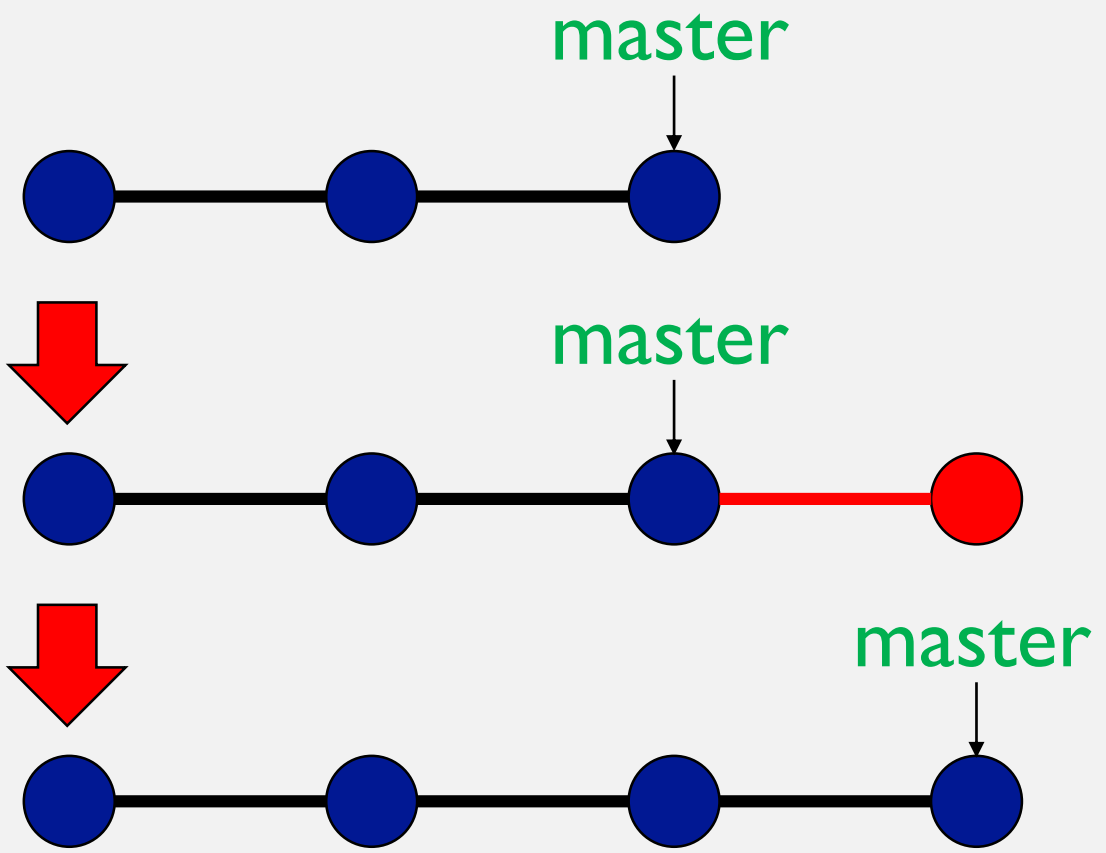
ブランチとはコミットにつけられた「別名」

master: 最初に作成されるブランチ
HEAD: いま自分が見ているブランチ



ブランチ

「現在自分がいるブランチ」は、コミットすると自動的に動く



現在の状態

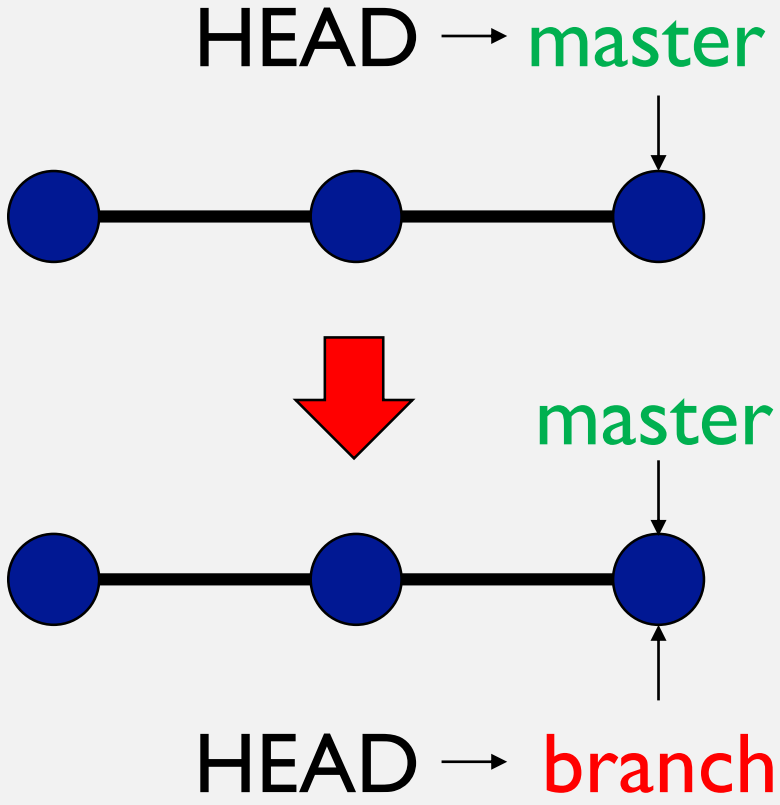
コミットを作成

コミットを作成

ブランチ

「git checkout –b *branchname*」により

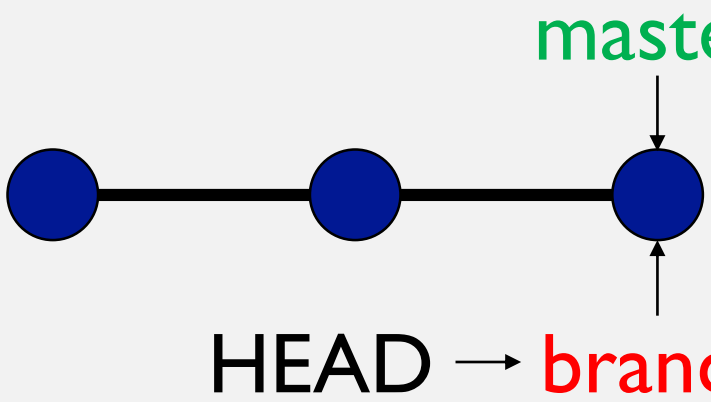
1. branchnameという名前のブランチを作り
2. そのブランチに自分(HEAD)が移る



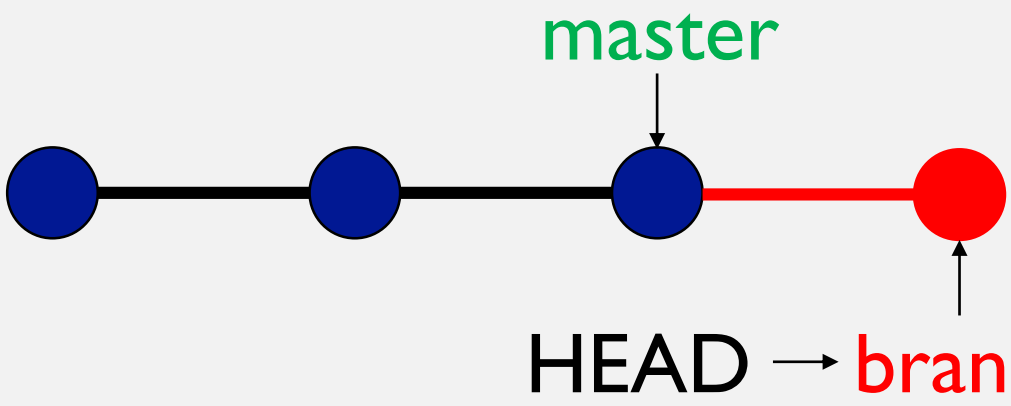
git checkout –b branch

ブランチ

「コミット」により動くのは、HEADがある(自分が今見ている)ブランチ



banch上でコミットした

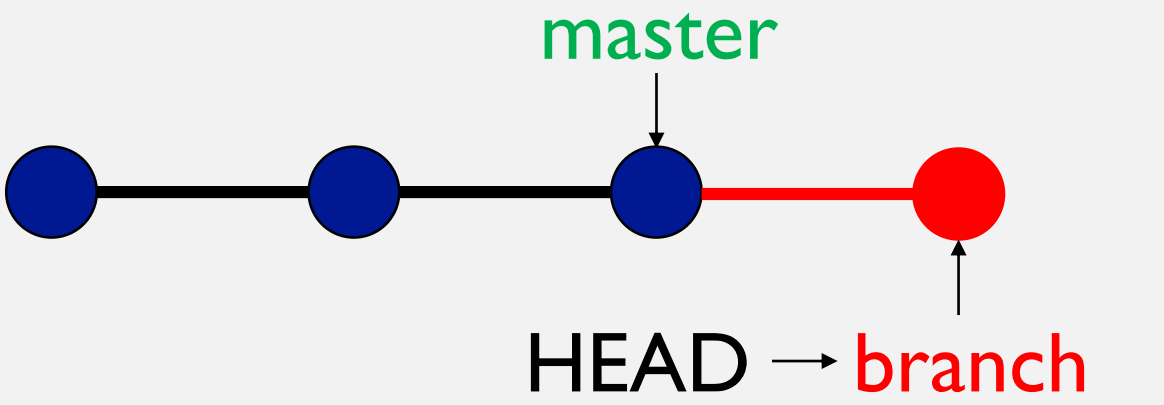


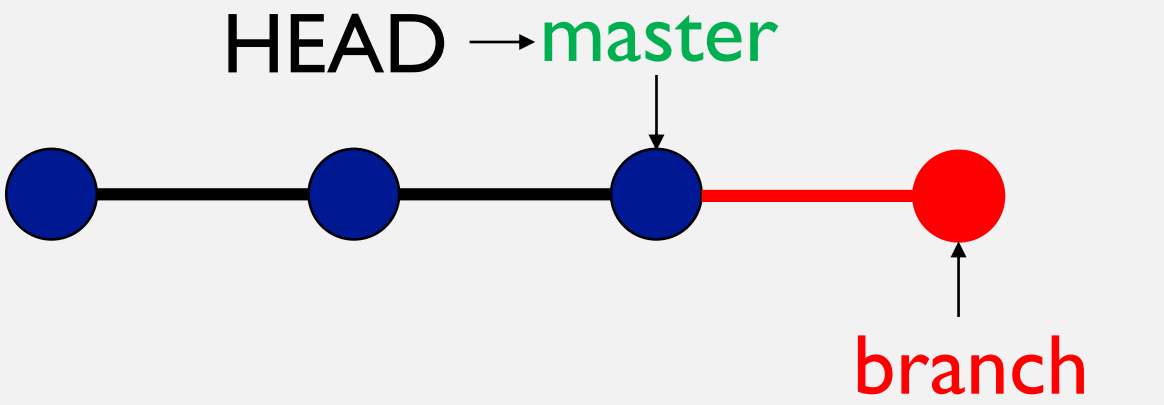
コミットが作られ、branchと
HEADが動く(masterはそのまま)

ブランチ

git checkout *branchname*

自分が見るブランチ(HEADが指すブランチ)をbranchnameに変更する





今見ているブランチが
masterに変わった

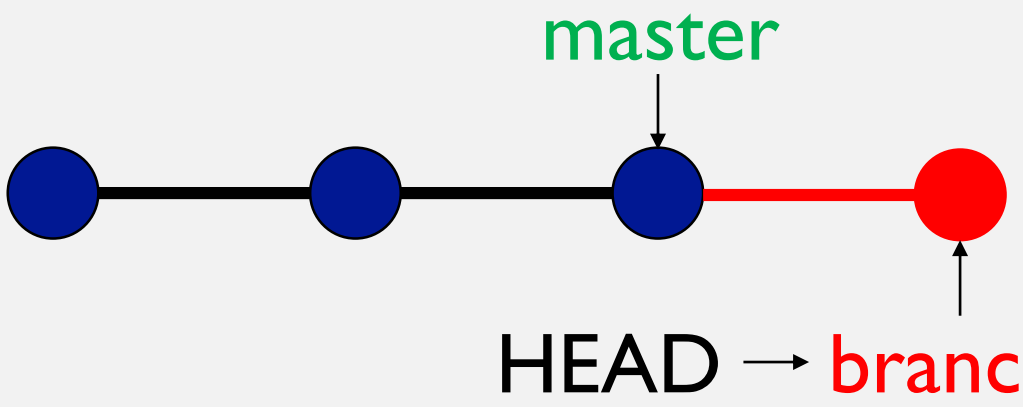
マージ

異なる二つのコミットから、新たなコミットを作ること

git merge *branchname*

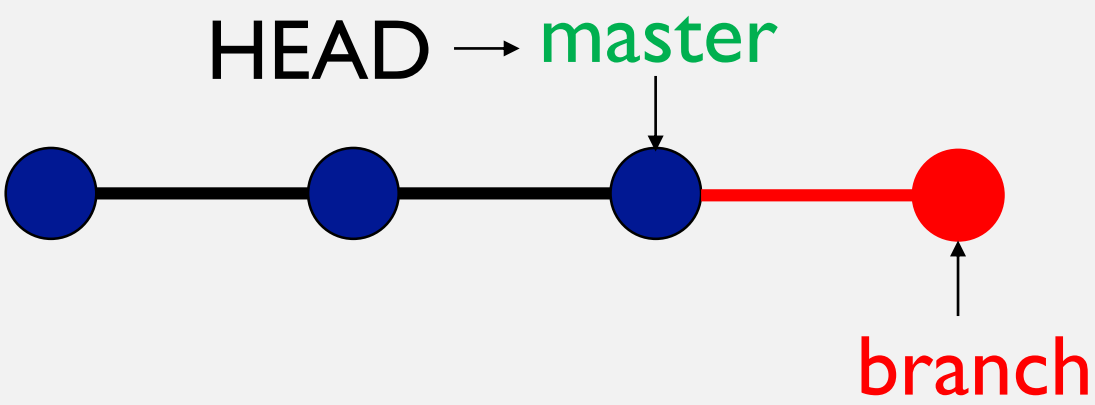
現在自分が見ているブランチに、branchnameの変更を取り込む
この時、「歴史が分岐しているかどうか」で動作が変わる

マージ (Fast Forward)



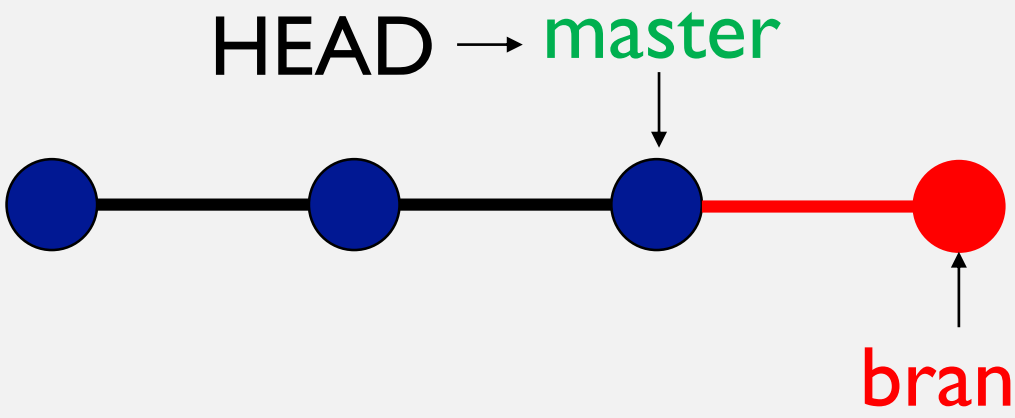
現在、自分はbranchにいて、
masterよりも進んだ状態

git checkout master



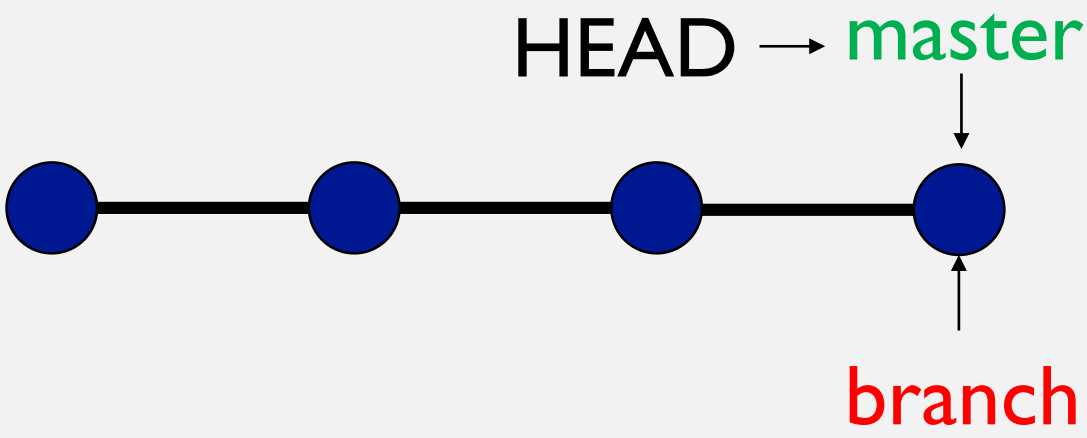
masterブランチに移動した

マージ (Fast Forward)



git merge branch

masterブランチに
branchの修正を取り込む

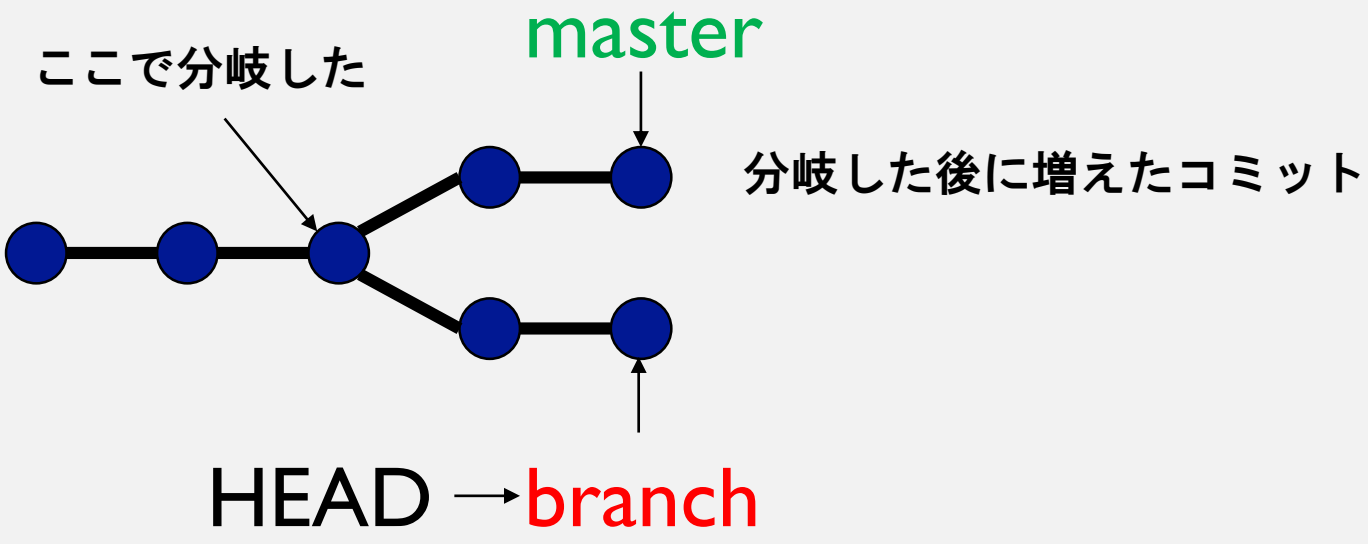


ブランチが移動するだけ
(Fast Forward)

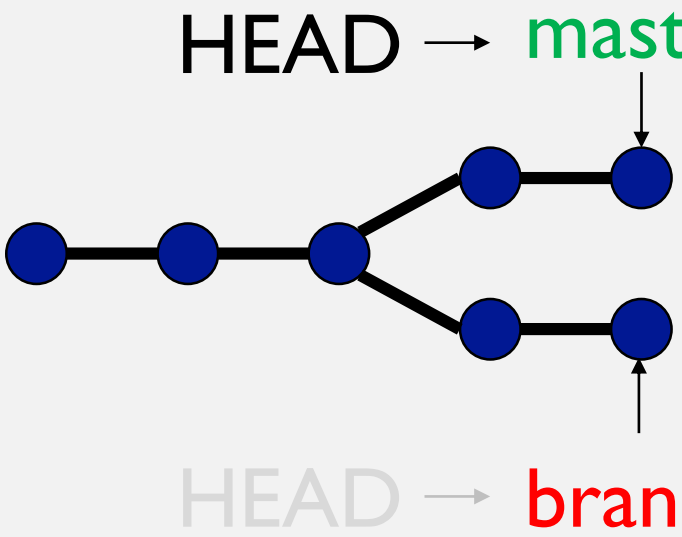
マージ (non-fast-forward)

歴史が分岐している場合

自分がブランチで作業している間に
masterブランチにコミットが増えていた

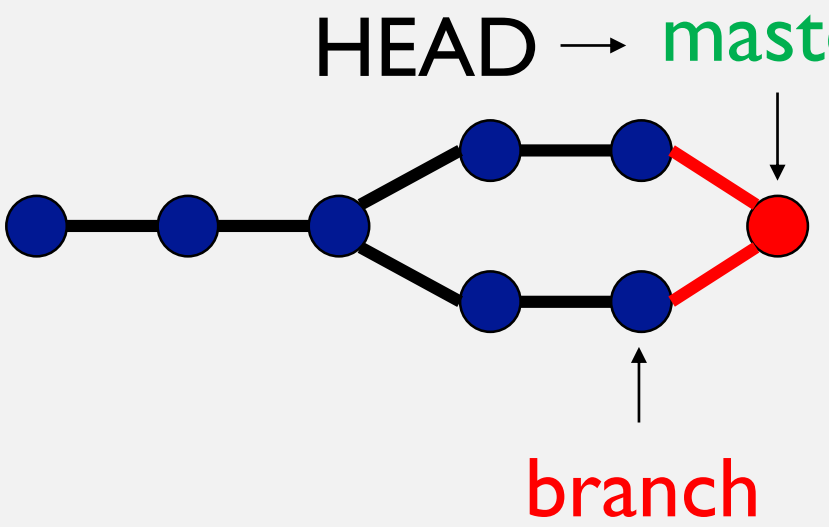


マージ (non-fast-forward)



git checkout master

まず、masterブランチに移る



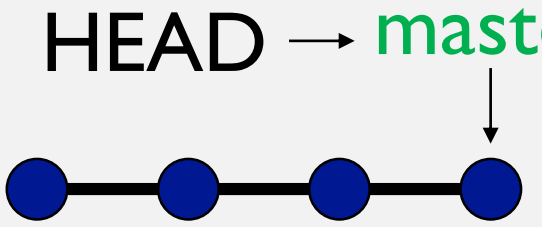
git merge branch

1. branchの修正を取り込み
2. 新たなコミットができて
3. master/HEADがそこを指す

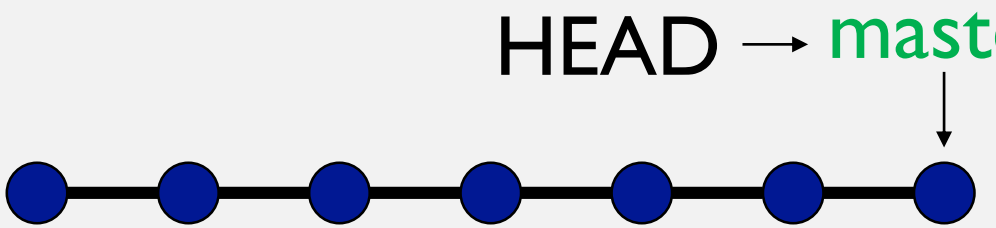
なぜブランチを使うか？

A.「修正」をまとめるため

masterブランチだけで作業していると…



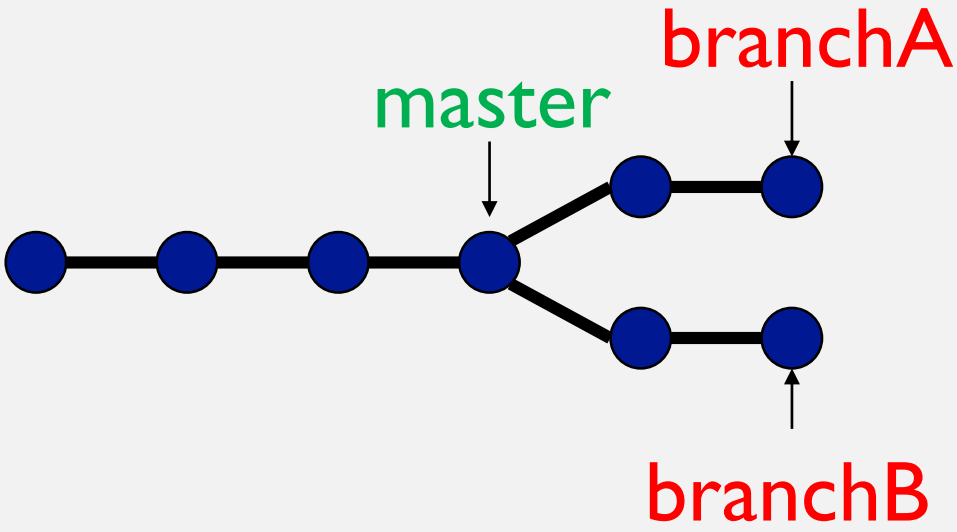
機能Aと機能Bを実装するぞ



機能Bがバグったんだけど、
何が原因かわからない…

なぜブランチを使うか？

「まとまり」ごとにブランチを分けて作業



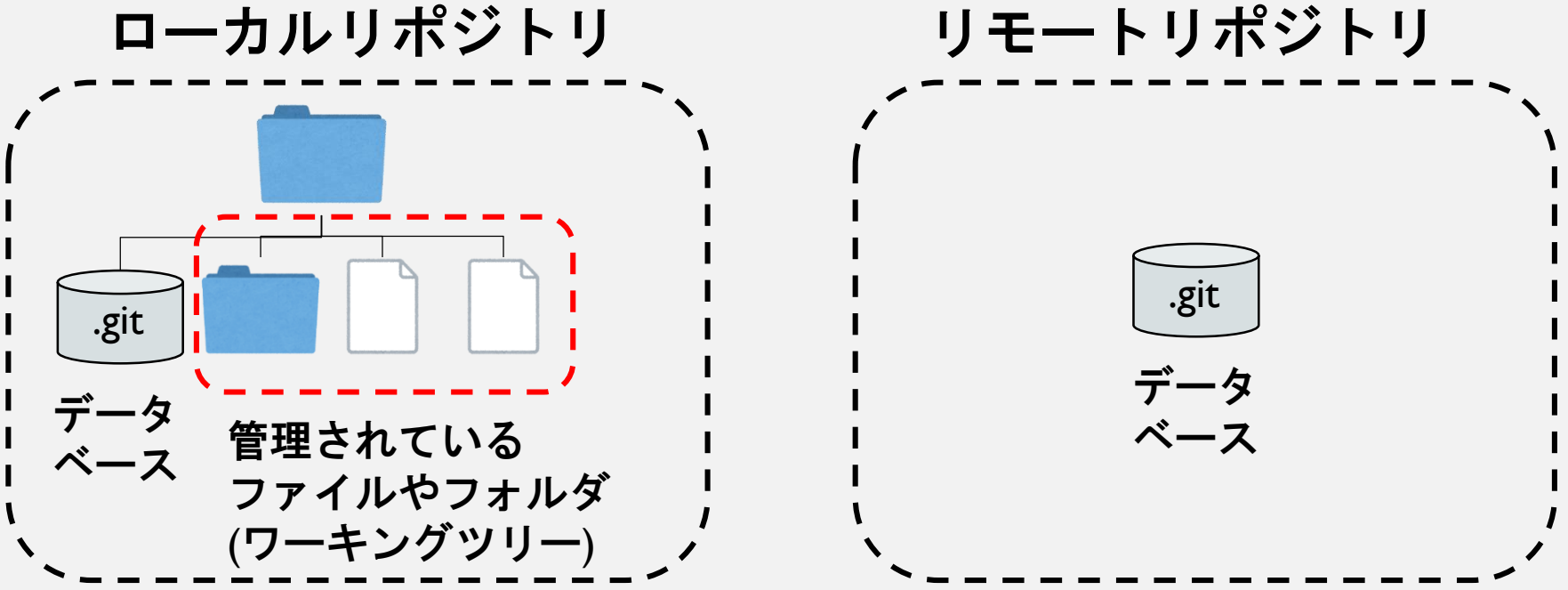
機能Bでバグったけど、
少なくとも機能Aは無関係

Gitでは、ブランチを気軽に作ったり消したりしながら開発する

リモートリポジトリ

リモートリポジトリとは、データベースだけのリポジトリ

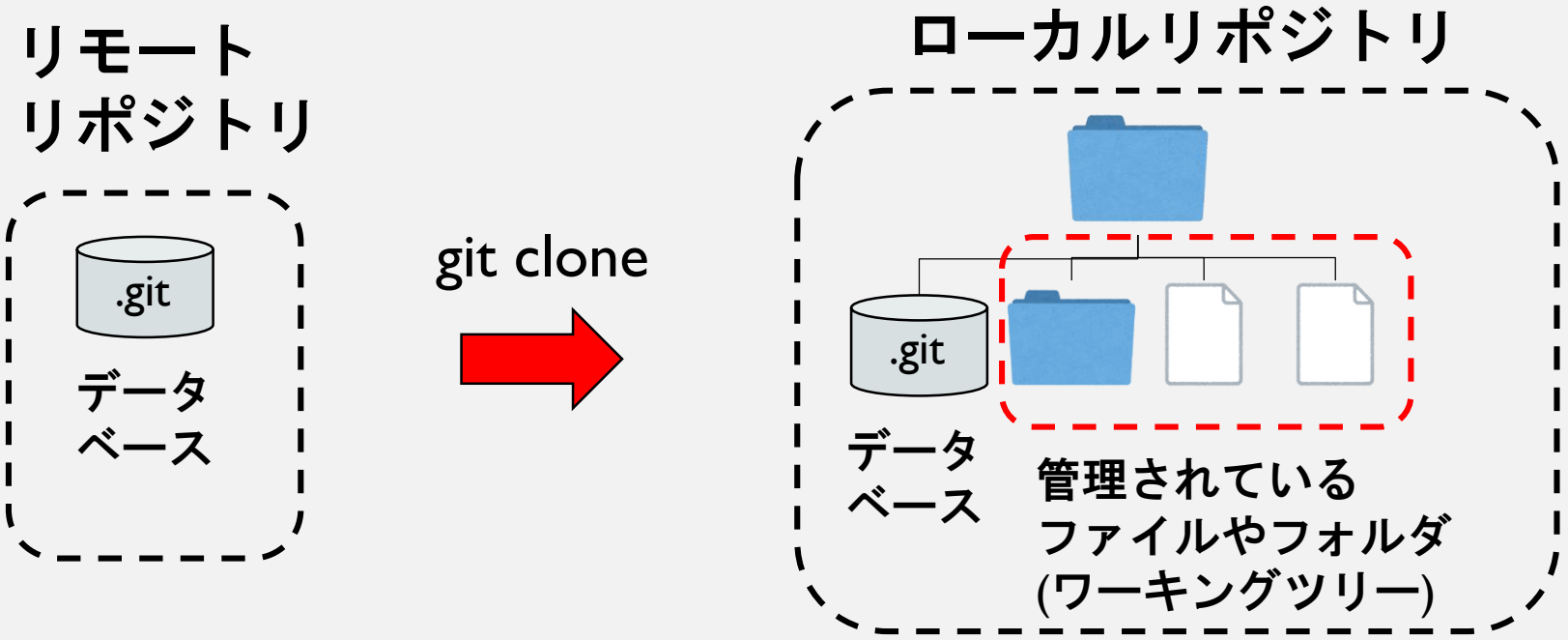
(ベアリポジトリ)



ワーキングツリーを含まず
.gitディレクトリだけを含むと思えばよい

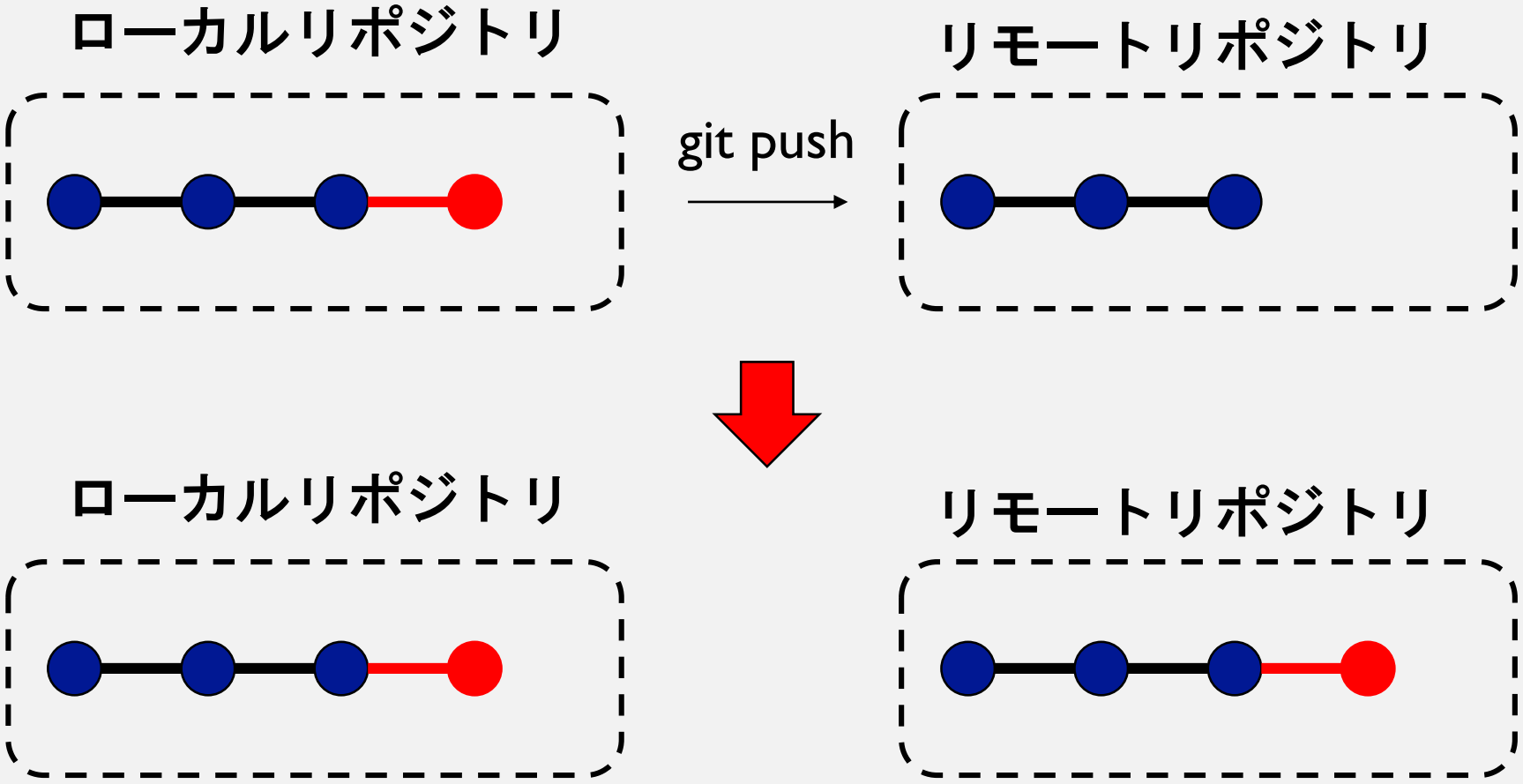
git clone

リモートからデータベースをローカルにコピーし、
ワーキングツリーを展開する



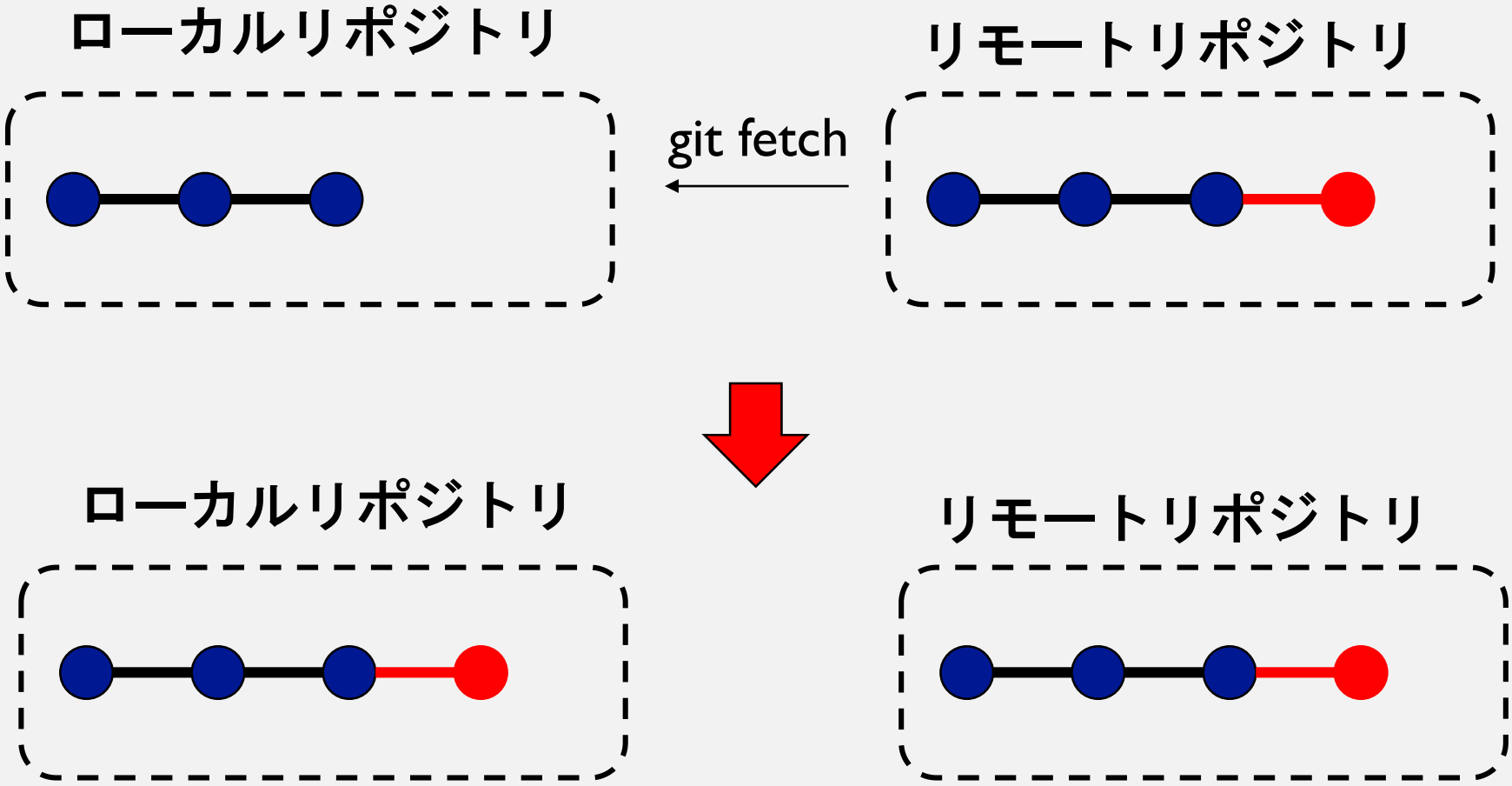
git push

ローカルで変更された「歴史」をリモートに反映させる



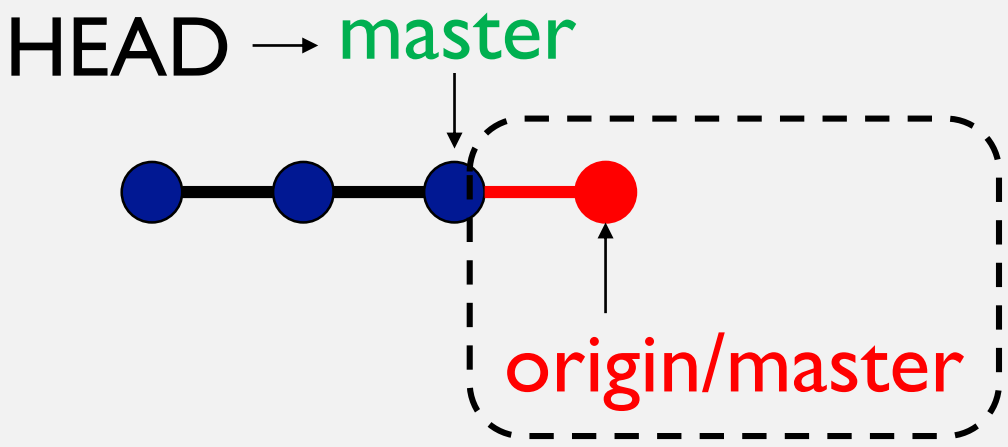
git fetch

リモートで変更された「歴史」をローカルに取ってくる



git fetch

git fetchしてもローカルのmasterやHEADは修正されない

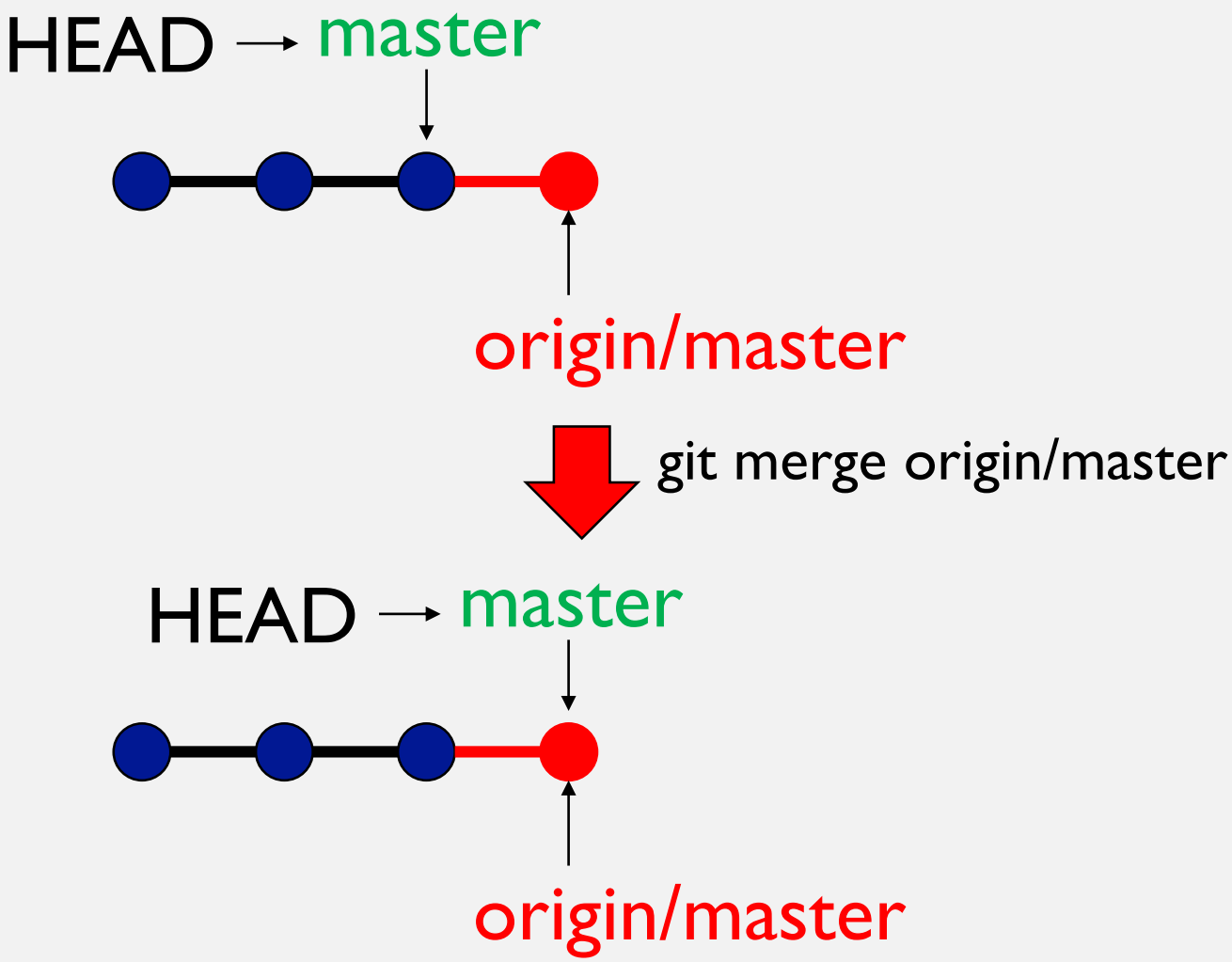


git fetchで追加された部分

※ リモートのブランチは origin/branch名という名前にすることが多い

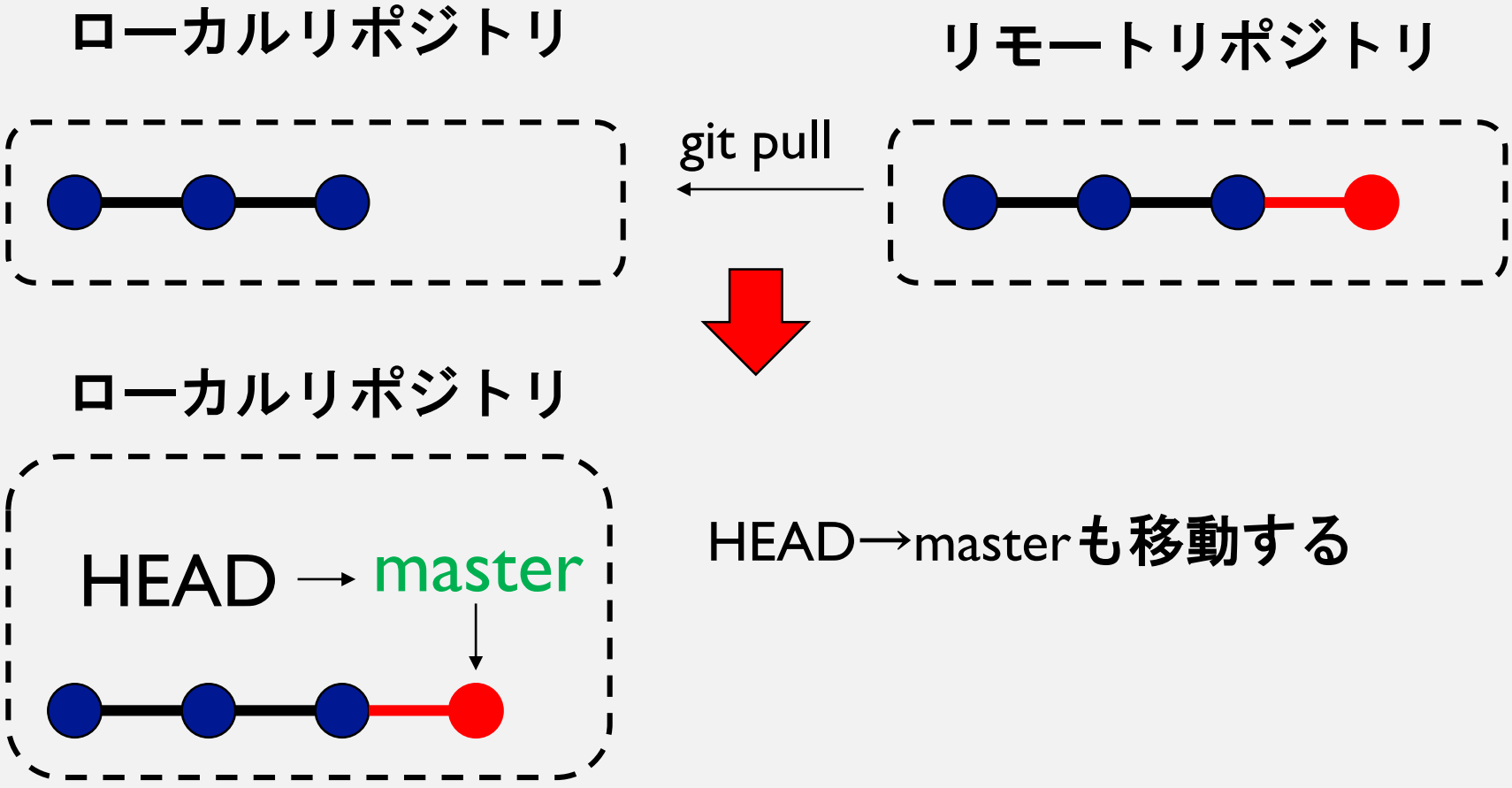
git fetch

リモートのブランチをマージすることで修正を取り込む



git pull

git fetchしてgit merge origin/masterを一度に行う



事故が起こりやすいので、慣れるまでgit pullは使わない方がよい

まとめ

Gitは慣れるまではそこそこ大変

しかし、使わない場合に比べて
開発効率は倍以上になる ※ 個人差あり

使わない場合は人生を2倍以上損している

※ 個人差あり

普段から少しずつ使って慣れましょう